XML Research at UTA

© Leonidas Fegaras
XML Lab

Web site:  http://lambda.uta.edu/xlab/
Location:  GACB 107
Faculty:  Leonidas Fegaras, Ramez Elamsri, David Levine
Students:  Anthony Okorodudu, Ranjan Dash,
          Lekhendro Lisham, Shahina Ferdous
Graduates:  Sujoe Bose, Weimin He

Current projects:
• Querying and updating XML views – data integration
• Processing of continuous historical queries over XML update streams
• Load shedding for XML stream engines
• Joining XML streams
• Search engines for web-accessible XML documents
• Fine-grained dissemination of XML data in publisher/subscriber systems
Goal: Handle continuous XQueries over continuous streamed XML data
- Embedded updates in the streams
- Exact rather than approximate answers
- Produce continuous results, even when the results are not complete

Problem: most interesting operations are blocking and/or require unbounded state
- grouping & aggregation
- predicate evaluation
- sorting
- sequence concatenation
- backward axis steps

We want to address the blocking problem differently
- Display the current result of the blocking operation continuously in the form of an update stream
  - incoming vs. generated updates
Traditional Stream Processing

- Typically, a stream consists of numerical values or relational tuples
- Focuses on a sliding window
  - fixed number of tuples, or
  - fixed time span
- Extracts approximate results
- Uses a small (bounded) state
- Examples:
  - top-k most frequent values
  - group-by SQL queries (OLAP)
  - data stream mining
Our View of XML Update Streams

- A continuous (possibly infinite) sequence of XML tokens with embedded updates
  - Usually, a finite data stream followed by an infinite stream of updates
  - three basic types of tokens: `<tag>`, `</tag>`, `text`
  - the target of an update is a stream subsequence that contains zero, one, or more “complete” XML elements
  - the source is also a token sequence that contains complete XML elements
  - updates are embedded in the data stream and can come at any time
    - update events can be interleaved with data events and with each other
    - each event must now have an id to associate it with an update
  - updated regions can be updated too
  - to update a stream subsequence, you wrap it in a Mutable region
  - three types of updates:
    - replace
    - insertBefore
    - insertAfter
<table>
<thead>
<tr>
<th>id</th>
<th>Event</th>
<th>equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>&lt;a&gt;</code></td>
<td><code>&lt;a&gt;</code></td>
</tr>
<tr>
<td>1</td>
<td><code>&lt;b&gt;</code></td>
<td><code>&lt;b&gt;</code></td>
</tr>
<tr>
<td>1</td>
<td>StartMutable(2)</td>
<td><code>&lt;c&gt;</code></td>
</tr>
<tr>
<td>2</td>
<td><code>&lt;c&gt;</code></td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>&lt;/c&gt;</code></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>EndMutable(2)</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td><code>&lt;/b&gt;</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>StartInsertBefore(3)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>&lt;c&gt;</code></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td><code>&lt;c&gt;</code></td>
</tr>
<tr>
<td>3</td>
<td><code>&lt;/c&gt;</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>EndInsertBefore(3)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><code>&lt;/a&gt;</code></td>
<td><code>&lt;a&gt;</code></td>
</tr>
</tbody>
</table>
Continuous Queries

• Need to decide: snapshot or temporal stream processing?
  • Snapshot: after a replace update, the replaced element is forgotten
  • Temporal: “some” of the replaced elements are kept
    – we may have repeated updates on a mutable region, forming a history list
    – each version has a time span (valid begin/end times)
    – the versions kept are determined at run time from the temporal components of the query that process that region

• Query language: XQuery with temporal extensions
  e?t           time projection               “give me the version before t secs”
  e#v           version projection           “give me the past v version”
  e?[t]         time sliding window          “give me all versions the last t secs”
  e#[v]         version sliding window       “give me the v latest versions”

• The default is “current snapshot” (version #0 at time 0)
• Much finer grain for historical data than sliding windows
Continuous Results

- Our stream engine is implemented as a pipeline
  - each pipeline stage performs a very simple task
- The final pipeline stage is the **Result Display** that displays the query results continuously
  - the display is a editable text window (a GUI), where text can be inserted, deleted, and replaced at any point
  - when an update is coming in the input stream, it is propagated through the result display, where it causes an update to the display text!
- Why?
  - This is what you really want to see as the result of a query
    - eg, in a stock ticker feed stream, where updates to ticker values come continuously
  - It leads to “optimistic evaluation” where results are displayed immediately, to be retracted or modified later when more information is available
    - addresses the “blocking problem”
    - minimizes caching
Snapshot Example

- **XQuery:**

```xml
<books>{
    for $b in stream("books")//biblio[publisher="Wiley"]/books
    where $b/author/lastname="Smith"
    order by $b/price
    return <book>{ $b/title, $b/price }</book>
}</books>
```

- This is what you see in the display:

```xml
<books>
    <book><title>All about XML</title><price>35</price></book>
    <book><title>Querying XML</title><price>120</price></book>
...
```
A Temporal Query

- Display all stocks whose quotation increased at least 10% since the last time, sorted by their rate of change:

```xml
<quotes>{
    for $q in stream("tickers")//ticker
    where $q/quote > $q/quote#1 * 1.1
    order by ($q/quote - $q/quote#1) div $q/quote
    return <quote>{ $q/name, $q/quote }</quote>
}
```
Another Temporal Query

- Suppose a network management system receives two streams from a backbone router for TCP connections
  - one for SYN packets, and
  - another for ACK packets that acknowledge the receipt
- identify the misbehaving packets that, although not lost, their ACK comes more than a minute late

```xml
for $a in stream(“ack”)//packet
where not (some $s in stream(“syn”)//packet?[60]
  satisfies $s/id = $a/id
  and $s/srsIP = $a/destIP
  and $s/srcPort = $a/destPort)
return <warning>{ $a/id, $a/destIP, $a/destPort }</warning>
```
Yet Another …

- Radar detection system
  - A swiping antenna monitors communications between vehicles
    - sweeping rate: 1 round/sec
  - Determines the time of the communication, the angle of antenna, and the frequency of signal

- Locate the position of a vehicle by correlating the streams of two radars:

```xml
for $r$ in stream("radar1")//event?[1],
   $s$ in stream("radar2")//event?[1]
where $r/frequency = s/frequency$
return <position>{ triangulate($r/angle,$s/angle) }</position>
```
Problems

- Most interesting operations are blocking and/or require unbounded state
  - predicate evaluation
  - sorting
  - sequence concatenation
  - backward axis navigation
- If we are not careful, history lists may be arbitrarily long
  - need to truncate them based on
    - whether a region is mutable or not (mutability analysis)
    - query requirements
    - client interests
Our Approach

- **Pessimistic evaluation**: at all times, the query display must always show the correct results up to that point
- **Optimistic evaluation**: display any possible output without delay and later, if necessary, retract it or modify it to make it correct
  - far more powerful than lazy evaluation

How?
- Generated and incoming updates are propagated through the evaluation pipeline until they are processed by the display
- They may cause changes to the states of the pipeline stages

Examples:
- **Event counting**: instead of waiting until we count all events, we generate updates that continuously display the counter “so far”
- **Predicate evaluation**: assume the predicate is true, but when you later find that it is false, retract all output associated with this predicate
- **Sorting**: wrap each element to be sorted around an update that inserts it into the correct place to the element sequence “so far”
Contributions

- Instead of eagerly performing the updates on cached portions of the stream, we propagate the updates through the pipeline
  - … all the way to the query result display
  - the display prints the results continuously, replacing old results with new

- Other approaches:
  - continuously display approximate answers by focusing on a sliding window over the stream

- Our approach:
  - generate exact answers continuously in the form of an update stream

- But the propagated updates may affect the state of the pipeline operators
  - developed a uniform methodology to incorporate state change

- Used this update processing framework to unblock operations and reduce buffering
  - let the operations themselves embed new updates into the stream that retroactively perform the blocking parts of the operation
  - why? because “later” is often better than “now”
State Transformers

- Each stage in the query evaluation pipeline implements a state transformer
  - Input: a single event and a state S
  - Output: a sequence of events and a new state S’
- Implemented as a function from an event to a sequence of events that destructively modifies the state
  - can be used in both pull- and push-based stream processing
- The state transformers need only handle the basic events: <tag>, </tag>, text, and begin/end of stream
  - the update events are handled in the same way for all state transformers
  - it requires only one function for each state transformer:
    \[
    \text{adjust}(s1,s2,s3)
    \]
    “if state s2 is replaced by s3, adjust the succeeding state s1 accordingly”
  - each state transformer is wrapped by a fixed function that handles update events by modifying the state using the adjust function, while passing the basic events to the state transformer
Example: Event Counting

- The state is an integer counter, \( \text{count} \)

- A blocking state transformer, \( f(e) \):
  
  ```java
  if e is a text event
    count = count+1
    return []
  else if e is end-of-stream
    return [ "count value" ]
  ```

- A non-blocking state transformer:
  
  ```java
  if e is begin-of-stream
    return [ startMutable(id), "0", endMutable(id) ]
  else if e is a text event
    count = count+1
    return [ startReplace(d), "count value", endReplace(id) ]
  ```

- The adjust function is:
  
  ```java
  adjust(s1,s2,s3).count = s1.count+(s3.count-s2.count)
  ```
XPath Steps

- The state transformers of simple XPath steps are trivial to implement
  - their adjust function is the identity
    \[ \text{adjust}(s_1, s_2, s_3) = s_1 \]
- **Example: the Child step** (/tag)
  - **state:**
    - need a counter \texttt{nest} to keep track of the nesting depth, and
    - a flag \texttt{pass} to remember if we are currently passing through or discarding events
  - **logic:**
    - when we see the event \texttt{<tag>} at \texttt{nest}=1, we enter pass mode and stay there until we see \texttt{</tag>} at \texttt{nest}=1
    - when in pass mode, we return the current event
    - otherwise, we return [ ]
- Handles most XQueries
- Currently, only snapshot queries
- Tested on two datasets:
  - XMark 224MB artificial data
  - DBLP 318MB real data
- Throughput: between 1 and 14 MB/s
Updating XML Views

- XML is frequently used for data integration
  - Most data sources can export their data in XML form
  - XQuery can work on multiple data sources and allows complex integration schemes

- Querying XML views has already been addressed by others
  - SilkRoute and XPeranto were two early systems
  - Many commercial DB systems already do this

- A large body of work on the relational view update problem
- Very little work on updating XML views
  - There is already a W3C proposal for XML updates
  - Interested in updates to XML data generated by relational databases

- Problem: XML data generated by data sources may be transformed through multiple layers of views
  - Need to propagate these updates back to the views bypassing the views
  - Translate XQuery updates to SQL updates
Virtual vs. Materialized Views

- A view in relational databases is a single table derived from one or more base tables or other previously defined views
  
  ```sql
  create view myView as
  select dname, count(*)
  from department join employee on dnumber=dno
  group by dname
  ```

- A virtual view is a view that does not exist in physical form
  - Queries over views are translated to queries over base tables
  - View updates are translated to base table updates

- A materialized view involves physically storing the result of the view in a temporary table
  - Very efficient querying
  - Problem: keep the materialized view up-to-date
  - Incremental updates: when new tuples are inserted, removed, or updated in the base tables, some tuples are changed in the materialized view table
Relational Views

- Not all views are updatable
  - 'with check option' in SQL
  - A view over a single base table is updatable if it contains the primary key
  - Groups with joins, groupby, and aggregations are generally not updatable

- XML views
  - XML publishing of relational data is an XML virtual view
    - Good choice if base data are dynamic (they change frequently)
    - ... or the view data are large
  - XML native storage system is a materialized view
    - Good choice if base data are static
Data Integration

- An integrated distributed database supporting a global schema
- Usually done by middleware that unifies the individual local schemas into a global schema and includes data from all local sites
- Sometimes, the mapping considers replication and horizontal/vertical fragmentation
  - In general, it needs schema matching
  - Eg, person name is the same as employee name
- It must provide a mapping from global to local schemas
- Users submit a single global query
  - Location transparency
  - The global query is translated to local queries and the results are integrated based on the mapping
  - Semi-joins are often used for efficiency
XML Data Integration

- db1: XML wrapper
- db2: XML wrapper
- db3: XML wrapper
- db4: XML wrapper

Data integration

User query/update
XML Updates

- Updates are XQuery expressions that return ()
- There are some restrictions about where an update can occur
- Examples:

```xml
replace $d//gradstudent[name/lastname="Smith"]/gpa
with 3.7

insert <zip>12345</zip>
into $d//gradstudent[name/lastname="Smith"]/address

for $e in $d//employee
return replace $e/salary with $e/salary*1.5
```
Propagating Updates through XML Views

- A view is expressed as an XQuery
  - The XQuery input are relational tables mapped to a default XML schema
  - The XQuery output is the XML view
- Data integration is done with XQuery over views
- You may have multiple layers of views and integration mappings
- XQuery updates must be over updatable views
  - They must be able to map back to the original data sources
- Goal: Given an XQuery update over XML views, we want to
  - Propagate the updates back to the data sources (relational tables)
  - Translate the XQuery updates to SQL updates
- Correctness: The updated data sources, when mapped through the views, must yield the same XML data as if we had updated the XML data directly
- Problem: cascaded updates (when view is a join)
View Example

- Relational schema of the DBLP database:
  
  \[\text{Dblp}(\text{id, parent})\]
  
  \[\text{Inproceedings}(\text{id, parent, key, number, title, year})\]
  
  \[\text{Author}(\text{id, parent, author})\]
  
  \[\text{Cite}(\text{id, parent, label, cite})\]

- It is mapped to the DTD:
  
  \[
  \text{<!element dblp (inproceedings*)>}
  \]
  
  \[
  \text{<!element inproceedings (number,title,year,author+,cite*)>}
  \]
  
  \[
  \text{<!attlist inproceedings key ID #REQUIRED>}
  \]
  
  \[
  \text{<!element cite (#PCDATA)>}
  \]
  
  \[
  \text{<!attlist cite label #IDREF>}
  \]
  
  ...
The View Mapping

for $d$ in SQL[ Dblp ]
return <dblp>{
  for $i$ in SQL[ Inproceedings ]
  where $i$/parent = $d$/id
  return <inproceedings key="{$i/key/data()}">{
    <number>{$i/number/data()}</number>,
    <title>{$i/title/data()}</title>,
    <year>{$i/year/data()}</year>,
    for $a$ in SQL[ Author ]
    where $a$/parent = $i$/id
    return <author>{$a/author/data()}</author>,
    for $c$ in SQL[ Cite ]
    where $c$/parent = $i$/id
    return <cite label="{$c/label/data()}">{
      SQL[ Inproceedings* ][key=$c/cite]/title/data()
    }</cite>
  }<inproceedings>
}<dblp>

Unique data source: SQL[ T ]
  Allows updates to propagate through T
Common data source: SQL[ T* ]
  Doesn't allow update propagation

SQL[[Cite]] is row*, where:
<!element row (id,parent,label,cite)>
An example of an XQuery update:

for $i in $view//inproceedings[author="John Smith"]
where $i/title = "XML for Dummies."
return replace $i/year with 2009

How do we map these updates back to the database?

Need to generate the following SQL update:

update Inproceedings
set year = 2009
where title='XML for Dummies.'
and id in (select parent from Author where author='John Smith')

What is missing here is a link between the data generated from views and the source that produced this data
for $d$ in SQL[ Dblp ]
return <dblp>{
    for $i$ in SQL[ Inproceedings ]
    where $i$/parent = $d$/id
    return <inproceedings key="{"$i$/key/data()"">{
        <number id="{$i$/id/data()" source="Inproceedings.number">{$i$/number/data()"}</number>,
        <title id="{$i$/id/data()" source="Inproceedings.title">{$i$/title/data()"}</title>,
        <year id="{$i$/id/data()" source="Inproceedings.year">{$i$/year/data()"}</year>,
        for $a$ in SQL[ Author ]
        where $a$/parent = $i$/id
        return <author id="{$a$/id/data()" source="Author.author">{$a$/author/data()"}</author>,
        for $c$ in SQL[ Cite ]
        where $c$/parent = $i$/id
        return <cite id="{$c$/id/data()" source="Cite.cite" label="{$c$/label/data()"">{
            SQL[ Inproceedings* ][key=$c$/cite]/title/data()
        }</cite>
    }<inproceedings>
}<dblp>
Lineage Tracing

- Need to establish a link between the update destination and the data sources used to form its value
- Lineage: the description of the origins of a piece of data
- The lineage can be statically inferred from the views and these annotations can be easily embedded in the XQuery code at compile-time to propagate the lineage at run-time
  - Only the id attribute needs to be embedded
- Use lineage tracing to propagate information about the origins of every updatable data piece through every expression in a query, to be used when this piece is to be updated
- Lineage tracing is done statically by conservatively embedding lineage attributes to constructions in all views used by the query
- We do a static analysis that it is able to embed the correct lineage attributes to all updatable constructions and determine, at compile-time, whether an XQuery update is valid or not
Contributions

- Our framework can handle any kind and any number of XML views and integration mappings, as long as they are expressed in XQuery.
- It can handle any XQuery update whose destination has lineage that can be traced back to an updatable data source.
- Uses *lineage types* that can concisely represent the lineage of any XML tree by linking the textual content of the XML leaves to the data sources that generated them.
- Provides a static algorithm that infers the lineage type of any XQuery expression and determines the translatability of the updates.
- Provides a static algorithm that embeds lineage annotations to constructions to be evaluated at run-time.
  - These annotations are propagated to the update destination at run-time, and are used to reflect the updates back to their data sources.
Lineage Types

- An XQuery expression returns a value that has a certain type (e.g., an XML Schema)
- We need to link these values to their sources
- ... so an XQuery expression may also have a lineage type that links its XML leaves to the data sources that created them
- Consider again:

  ```xml
  for $i in $view//inproceedings[author="John Smith"]
  where $i/title = "XML for Dummies."
  return replace $i/year with 2009
  ```

- The lineage type of variable $i is:

  ```xml
  Inproceedings { @key: Inproceedings.key,
  number: Inproceedings.number,
  title: Inproceedings.title,
  year: Inproceedings.year,
  ∀ Author ⇒ author: Author.author,
  ∀ Cite ⇒ cite { @label: Cite.label, ∀ Inproceedings* ⇒ Inproceedings.title } }
  ```
Translating XQuery Updates

- The goal is to find the lineage type of the update destination.
- If the lineage type is empty or is not unique, the update is wrong.
- Otherwise, we know exactly which table column to update (this is the implicit source attribute) and the id of the row.
- From the last example, the update destination of replace $i/year with 2009 has lineage type Inproceedings.year.
- The update is then translated into:
  
  ```
  update Inproceedings set year=2009 where id=$i/id/data()
  ```
  where i is equal to $i/id/data() (the propagated id).
- When the update destination has a composite lineage type, we generate multiple SQL updates guided by the lineage type.
Lineage Tracing Rules

- How do we infer the lineage type of an XQuery expression?
- Simple typing rules – far simpler than XQuery type inference
- Examples:
  - The lineage type of SQL[T], where T is a relational table with columns \( A_1, ..., A_n \), is:
    \[
    \forall T \Rightarrow \text{row } \{ A_1 : T.A_1, ..., A_n : T.A_n \}
    \]
  - If we know that an XQuery expression \( e \) has lineage type \( t \), then the expression \(<A>{e}</A>\) has lineage type \( A \{ t \} \)
  - Let \( \forall T \Rightarrow t_1 \) be the lineage type of \( e_1 \) in
    \[
    \text{for } v \text{ in } e_1 \text{ return } e_2
    \]
    and let \( t_2 \) be the type of \( e_2 \), assuming that the type of \( v \) is \( t_1 \),
    then the for-loop has lineage type \( \forall T \Rightarrow t_2 \)
Optimizations

- A view transforms a large part of the underlying data source into XML, while a casual user query or update may access only a small part of this view.
- Normalization rules can eliminate those parts of the view that do not contribute to the query result:
  - normalization will fuse the query with the view
  - will eliminate most parts of the intermediate data
  - the resulting query will work directly on the source data bypassing the view
- Examples of normalization rules:
  \[
  \langle A \rangle \{e\}\langle /A\rangle/\text{child::}B \rightarrow e/\text{self::}B
  \]
  \[
  \langle A \rangle \{e\}\langle /A\rangle/\text{self::}A \rightarrow e
  \]
- Need to promote relevant SQL predicates into SQL queries, and fuses SQL queries in pairs using transformation rules.
- Need to embed complex SQL queries inside XQuery.
Embedding SQL in XQuery

- Need special syntax to embed SQL code in XQuery
- It must evaluate the SQL query and convert the result to XML
  - To create deeply-nested XML data from flat relational data, we need to do an implicit group-by
- Consider the SQL query:
  
  ```sql
  select *
  from Inproceedings i join Author a on a.parent=i.id
  where i.title=‘XML for Dummies.’ and a.author=‘John Smith’
  ```

- You can't nest Author elements inside Inproceedings using SQL, but you can do it using code:
  
  ```xquery
  { <inproceedings> a.*,
    { <author> b.* </author> | b in result, b.id=a.id } }</inproceedings>
  | a in result }
  ```
Embedding SQL in XQuery

- The XQuery syntax:
  
  SQL[ row: ( i.@id, i.year, nest: ( a.@id, author ) ),
       from Inproceedings i join Author a on a.parent=i.id
       where i.title=`XML for Dummies.' and a.author=`John Smith'
       order by i.id  ]

- Evaluates the SQL query:
  
  select i.id, i.year, a.id, a.author
  from Inproceedings i join Author a on a.parent=i.id
  where i.title=`XML for Dummies.' and a.author=`John Smith'
  order by i.id

- ... and generates XML data of type row* where:
  
  <!element row (year,nest*)>
  <!attlist row id CDATA>
  <!element nest (author) >
  <!attlist nest id CDATA>
Predicate Promotion

- Need to convert XQuery predicates into SQL predicates
- Given a binding from $v$ to an SQL header $h$ and an XQuery predicate $p$, split $\text{split}_{v=h}(p)$ returns $(p_1,p_2)$ where $p_1$ is the part of $p$ that can be converted to an SQL predicate and $p_2$ is the rest
- For example,

  \[
  \text{split}_{v=\text{row:}(t.A,t.B)}(v/A=2)
  \]
  returns the SQL predicate $t.A=2$
- Same for joins:

  \[
  \text{split}_{v=s:(t.A,t.B),w=r:(s.C)}(v/B=w/C)
  \]
  returns the SQL predicate $t.B=s.C$
Optimizing Embedded SQL

- Promoting XQuery predicates inside SQL:

  for $v$ in SQL[ $h$, from $f$ where $p'$ ]
  where $p$ return $e$

  $\rightarrow (p1,p2) = \text{split}_{v=h}(p)$

  for $v$ in SQL[ $h$, from $f$ where $p'$ and $p1$ ]
  where $p2$ return $e$

- Fusing two SQL queries:

  for $v1$ in SQL[ $h1$, $q1$ ]
  where $p1$
  return $F$(for $v2$ in SQL[ $h2$, $q2$ ]
  where $p2$ return $e2$)

  $\rightarrow (p12,p22) = \text{split}_{v1=h1,v2=h2}(p2)$

  for $v1$ in SQL[ row: ( $x$.*, nest: ( $y$.* ) ),
  from $s1$ $x$ left outer join $s2$ $y$ on $p12$
  order by $x$.id ]
  where $p1$
  return $F$(for $v2$ in $v1/nest$
  where $p22$ return $e2$)
Other Current Projects at XML Lab

- **Load shedding for XML stream engines**
  - when stream data arrive faster than you can process
  - we can handle small fluctuations by queuing events
    - eventually, we may have to remove elements from the queue
  - removing queued elements improves quality of service but may affect the quality of data (decreases the accuracy of the query results)
  - unlike relational streams, queued XML elements can be of any size
  - selecting a victim from the queue must be faster than processing the element but intelligent enough to maximize quality of data

- **Joining XML streams**
  - typical evaluation: symmetric hash join
  - all events from both stream must be cached
  - non-blocking but unbounded
  - needs intelligent shedding of “cold” events
    - based on past history
    - but also on knowledge about the future (punctuations)
Other Current Projects at XML Lab

- Search engines for XML documents
  - Given an XPath or XQuery
    - find the top ranked web-accessible XML documents that match the query and
    - return the results of evaluating the queries against these documents
  - Uses full-text syntax extensions to XQuery
    //article[author/lastname = "Smith"][title ~ "XML" and "XQuery"]/title
  - Far more precise than keyword queries handled by web search engines
  - Other approaches use inverted indexes for both content and structure
  - We use content and structure synopses for document filtering
    - structural summary matching
    - containment filtering
    - relevance ranking based on both TF*IDF scoring and term proximity
  - Application: indexing and locating XML documents in a P2P network
Other Current Projects at XML Lab

- Fine-grained dissemination of XML data in a publisher/subscriber system
  - Publishers disseminate XML data in stream form to millions of subscribers
  - Subscribers have profiles (XPath queries) and expect to receive from publishers at least those XML data that match their profiles
    - How do we avoid flooding the network by sending all data to all subscribers?
    - How do we utilize the profiles so that only relevant data go to subscribers?
  - Need a middle-tier, consisting of an overlay network of brokers that discriminately multicast XML fragments based on profiles
    - Self adjustable, scalable to both data volume and number of subscribers
      - we are currently looking at tree overlays and P2P networks
  - Conservative dissemination:
    - Makes sure that all relevant fragments will reach interested subscribers
    - but it may also send irrelevant fragments