Niagra People

- Faculty
  - David J. DeWitt
  - Jeffrey Naughton

- Graduate Students
  - Jianjun Chen
  - Anurag Gupta
  - Vishal Kathuria
  - Naveen Prakash
  - Feng Tian
  - Chun Zhang
  - Leonidas Galanis
  - Jaewoo Kang
  - Qiong Luo
  - Ravishankar Ramamurthy
  - Yuan Wang

- Niagra Alumni
  - Rushan Chen
  - Jun Li
  - Kristin Tuft
  - Bruce Jackson
  - Jayavel Shanmugasundaram

NIAGRA Status Update
Goals of the Project:

- At last year’s meeting, we stated the goals were to:
  - improve the precision of Internet searching
  - allow queries over the whole Internet (the “FROM *” clause)
  - monitor the Internet for changes
- Not (quite) finished yet...

What have we done since?

- Investigated storing and querying XML in an RDBMS (see VLDB paper).
- Completed three prototypes:
  - A “text-in-context” XML search engine.
  - An XML-QL query engine.
  - An XML-QL trigger engine.
- This presentation will cover the prototypes.
Why not use RDBMS?

■ Poor efficiency for certain specialized operations.
■ More important reason:
  – RDBMS: system must know full schema at data load time, vs.
  – NIAGRA: user must know fragment of schema at query time

Niagra Prototype Components
Text-in-Context XML SE

- Rather than ask:
  What are all the documents that contain the string “Montreal”?

- We can ask:
  What are all the documents that contain ship departure information for a ship whose name is “Montreal”?

How it works:

- “Off-line” crawls web to find and index documents.
- Executes “SEQL” queries over index.
- Two uses:
  – stand alone (from GUI), or
  – part of query engine.
XML-QL Query Engine

- Evaluates queries expressed in XML-QL (language developed at ATT.)
- Looks like strange SQL with path expressions (but even uglier.)
- Result is XML

Search Engine vs. XML-QL

Search Engine Query:
Find all XML files with ship departure events where the departing ship’s name is “Montreal”?

XML-QL Query:
What is a list of departure dates for ships named “Montreal”? 
Ex: Fragment of XML file...

```xml
<port>
  <portname> Hong Kong </portname>
  <departure>
    <ship>
      <shipname> Montreal </shipname>
      <cargo> Software CDs </cargo>
    </ship>
    <date> January 1, 2000 </date>
  </departure>
</port>
```

XML-QL Query...

```xml
WHERE <port>
  <portname> $v1 </portname>
  <departure>
    <ship>
      <shipname> “Montreal” </shipname>
    </ship>
    <date> $v2 </date>
  </departure>
</port>

</> content_as $v3
IN “*”
CONSTRUCT <departinfo>
  $v3
</>
```
Important Question

■ Which documents should be consulted to answer an XML-QL query? We support three approaches:
  – Explicitly listed documents (“in foo.xml”)
  – Documents conforming to DTD (“conforms to some_dtd.xml”)
  – Documents that satisfy search engine predicates extracted from query

Example of third approach:

■ Given the previous XML-QL query finding departures of ships named “Montreal”, the system will extract this Search Engine query:

```xml
port CONTAINS
    (portname AND
     departure CONTAINS
        (ship CONTAINS (shipname IS "Montreal")
         AND date))))
```
Control Flow for Query

■ So full flow of typical XML-QL query:
  – User submits XML-QL query
  – System extracts SEQL query from XML-QL, passes it to search engine
  – Search engine returns list of URLs
  – XML-QL engine fetches documents in URL list (aided by local cache), evaluates query
  – Answer returned to the user.

XML-QL Trigger Engine

■ Goal:
  – Allow users to define “triggers” on XML files using XML-QL predicates.
  – Scale to huge numbers of triggers by exploiting “on-the-fly” aggregation of triggers
Rest of presentation...

- More detail about the SE, QE, and Trigger Engine.
- A short demo.
- Wrap-up (future directions, questions.)
- Note: during the “demo session” this afternoon Niagra project members will be available for more in-depth information...

Future work

- Just getting started! major next tasks:
- Rewrite prototypes in C++ (instead of Java)
- Parallelize and run them on cluster:
  - 36 dual 550MHz Pentium IIs
  - 1 GB RAM each (36 GB total)
  - 45 GB disk each (1.6 TB total)
- Distributed query engine.
A “Text-in-Context” XML Search Engine

Traditional Search Engines

- Traditional search engines on the web:
  - Keyword searching
  - Return too many results!
  - Lots of manual work to screen through search results
- What do we do differently?
Our Search Engine

- Search in context (here comes XML!)
- More powerful queries, more accurate results
- Flavors of queries (SEQL) supported:
  - Find books with title “Java Programming” and price less than $50
  - Find titles of articles containing words “XML” and “search” that are less than 5 words apart
  - In the speech spoken by “Antonio”, find the line that contains “merchandise”

- So, how do we process these?

The Workhorse: Inverted Index

- Full text indexing
  - Document considered as a sequence of words
  - Both element names and their contents are indexed
- Lexicons records collections of index terms
  - Three types of lexicons in search engine: Text, Element, DTD
- Inverted lists indicate occurrences of index terms
Inverted Index Structure

- An inverted list records occurrences of an indexed term
  - e.g., <1; (19,27)(28,36)>: element “book” appears in doc1 from word number 19 to 27, as well as word number 28 to 36. (two “book” elements in doc1)
  - e.g., <1; 21,30>: text word “java” appears twice in doc1 at word numbers 21 and 30

- Containment & proximity relationships checked by positions
  - e.g. “book” (<1;20,23>) contains “title” (<1;19,27>)
  - e.g. “java” (<1;21>) appears next to “programming” (<1;22>) in doc1

- Inverted list sorted in increasing order of docno & positions

Why Inverted Index?

- Simple
- Fast on popular and useful information retrieval queries
- Highly tolerant of unstructured data, and data with different structures
- Preserves data source and document boundary
- Good scalability

Suitable for Web Information Processing
SEQL: Search Engine Query Language

1. Find books with title “Java Programming” and price less than 50
   
   
   `book contains (title is “Java Programming” and price < 50)`

2. Find titles of articles containing words “XML” and “search” that are less than 5 words apart
   
   `title containedin (article contains distance (“XML”, “search”) < 5)`

3. Find the line in “Antonio”’s speech that contains “merchandise”
   
   `line contains (“merchandise” containedin speech contains (speaker is “Antonio”))`

SEQL Operators

- A complete set of operations:
  - containment: CONTAINS, CONTAINEDIN
  - boolean: AND, OR, EXCEPT
  - proximity (text words only): IS, DISTANCE
  - numerical: >, >=, <, <=, =
  - DTD conformant: conformsto

- Inputs and outputs of operators are inverted lists
Process of Indexing

- Crawler finds an URL, gives it to SE Server
- SE Server passes URL to Index Manager
- Index Manager:
  - fetches document, assigns doc number
  - parses it into a sequence of index terms
  - puts terms and positions into lexicons and inverted lists
  - merges with rest of index
Query Processing

- SEServer accepts SEQL query from GUI or Query Engine
- Parser parses query, generates execution plan
- SE Operators contacts Index Manager
- Index Manager serves inverted lists
- SE Operators execute query plan

Query Processing

- GUI
- SEQuery
- URLs
- Query Engine
- SE Server
- Crawler
- Parser
- Query Plan Generator
- Execution Operators
- Inverted Index
- Index Manager
- Inverted lists request
- Inverted lists
- SEQuery
- URLs
Future Work

- Scalability with parallel processing
- Expedite indexing process
- Query Optimization
- Support for database queries (selection, projection, path expression, join)
- Crawler

Niagra Query Engine
Query Engine Architecture

- Client
- Connection Manager
  - XML-QL Parser
  - Query Optimizer
  - Execution Engine
  - Data Manager
- Search Engine
- Internet

Connection Manager

- Persistent connection with the client
- Client-Server communication is in XML
- Handles queries for SE, QE and Trigger Manager
Example DTDs

- http://www.publications.com/books.dtd
  ```xml
  <!ELEMENT book (author+, title) >
  <!ELEMENT author (#PCDATA) >
  <!ELEMENT title (#PCDATA) >
  ```

- http://www.publications.com/article.dtd
  ```xml
  <!ELEMENT article (author+, title, year) >
  <!ELEMENT author (#PCDATA) >
  <!ELEMENT title (#PCDATA) >
  <!ELEMENT year (#PCDATA) >
  ```

A Simple XML-QL Query

- WHERE
  ```xml
  <book>
    <author> $a </>
    <title> $t </>
  </>
  <article>
    <author> $a </>
    <year> 1995 </>
  </>
  IN conform_to “http://www.publications.org/article.dtd”
  CONSTRUCT <title> $t </>
  ```

- Give me the name of all books that have an author who wrote an article in the year 1995
XML-QL

- W3 recommended XML-QL
- Three ways of specifying data sources
  - IN "*"
  - IN "*" Conform_to "http://www.publications.org/book.dtd"
  - IN "http://www.bookstore.com/book.xml"
- Supports features like regular expression, tag variables, etc.

Query Engine Execution Flow

Client

Connection Manager
- query string
- XML-QL Parser
- Query Optimizer
- Execution Engine
- Data Manager

Search Engine

Internet
Query Engine

- Multiple Query Threads
- Parser
  - query string -> logical plan
- Optimizer
  - logical plan -> physical plan
- Execution Engine
  - Runs each operator concurrently

XML-QL Parser

- Creates a syntax tree from the XML-QL query
- Generates a logical plan from the syntax tree
- Operators:
  - Data Scan, Scan, Select, Join, Construct
Logical Plan

WHERE
<book>
  <author> $a </a>
  <title> $t </title>
</book>
</browser>
<browser>

CONSTRUCT <title> $t </title>

Select
year = 1995

year

author

book

Data Scan
book.dtd

Join
author = author

title

Select
year = 1995

year

author

article

Data Scan
article.dtd

Query Engine Execution Flow

Client

Connection Manager

XML-QL Parser

logical plan

Query Optimizer

Execution Engine

Data Manager

Search Engine

Internet
Query Optimizer

- Generates the SE Query to be sent to the Search Engine
- Generated SE queries
  - (book CONTAINS (author AND title))
    conformsto "http://www.publications.org/book.dtd"
  - (article CONTAINS ((year = 1995) AND author))
    conformsto "http://www.publications.org/article.dtd"

Query Engine Execution Flow

![Diagram of the Query Engine Execution Flow]

- Client
  - Connection Manager
  - XML-QL Parser
  - Query Optimizer
  - Execution Engine
  - Data Manager
- Search Engine
  - SE query
  - URLs
- Internet
Query Optimizer (contd.)

- Retrieves URLs of qualifying data sources from the SE
- Passes URLs to the data scan node
- Selects an algorithm for executing each operator
- Gives the physical plan to the Execution Engine

Query Engine Execution Flow

- Client
- Connection Manager
- XML-QL Parser
- Query Optimizer
  - physical plan
  - Execution Engine
- Data Manager
- Search Engine
- Internet
Execution Engine

- Creates streams to connect different physical operators
- Puts each physical operator in a physical operator queue
- Operator threads run these operators concurrently

Data Manager

- Handles request for XML files from the execution engine
- Returns DOM trees to the Data Scan operators
- Caches the XML files
Future Work

- Complete query optimizer
- Scalable and distributed implementations
- Different algorithms for operators
- IDREFs, XML Schema
**Motivation**

- **Continuous queries**: persistent XML-QL queries to obtain new results automatically
- **Large amount of frequently changing information on Internet**
- **An example**
  - Whenever the stock price for a company in the Computer Service industry drops by more than 5%, give me its stock price and profile
- **Challenge**: system scalability with arbitrary XML-QL queries
Group optimization

- Key assumption: many queries are similar
- Key advantage: share computation among grouped queries
- Limitations of previous approaches:
  - Can only handle a small number of simple queries
  - Unsuitable for dynamic continuous query environment

Incremental Group Optimization

- Strategy:
  - Queries are classified into groups based on their expression signatures
  - A new query is merged into one or more existing groups (instead of re-grouping all the queries in the system)
Expression Signature

Definition: The same syntax structure in different queries, but with potentially different constant values

Two trivial continuous queries:

Where <Quotes> <Quote>
    <Symbol>INTC</Symbol>
    </> element_as $g
in “http://www.nasdaq.com/quotes.xml” construct $g

Where <Quotes> <Quote>
    <Symbol>MSFT</Symbol>
    </> element_as $g
in “http://www.nasdaq.com/quotes.xml” construct $g

Expression Signature

\[
\text{Quotes.Quote.Symbol in “quotes.xml”}
\]

Group

• Signature
• Constant Table
• Execution Plan

```
File Scan
---

Group Plan
---

File Scan
```

<table>
<thead>
<tr>
<th>Constant Value</th>
<th>Intermediate file name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ints</td>
<td>file_i</td>
</tr>
<tr>
<td>MSFT</td>
<td>file_j</td>
</tr>
</tbody>
</table>

Constant Table
Incremental Grouping Example

Query Split with Intermediate files

- Incremental group optimizer may split a query into multiple queries
- Split operator stores its output tuples into the appropriate intermediate files
- Intermediate files are monitored like data files
- Upper level queries are triggered by the changes on their files
- Key Advantages
  - Avoid unnecessary invocation
  - Be able to utilize a common query engine
Performance Results

1000 tuples modified
All installed queries fired

1000 tuples modified
100 queries fired

Grouping Timer-based Queries

- **NiagraCQ Interface**
  
  CREATE CQ_name  
  XML-QL query  
  DO action  
  {START start_time} {EVERY time_interval} {EXPIRE expiration_time}

- **Timer-based queries:** time_interval > 0 (e.g. 10 min, 1 hour, 1 day etc)

- **Change-based queries:** time_interval = 0

- **Challenges due to different time_intervals**
NiagraCQ Architecture

Conclusions and Future Work

- Incremental group optimization significantly improves system performance
- NiagraCQ can be scaled to support a large number of continuous queries
- A cost model for group optimizer
- Grouping queries with multiple join operators
- Dynamic regrouping
- Distributed continuous query processing