The Niagara Internet Query System

Jeffrey Naughton, David DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, Rushan Chen

Abstract

Recently, there has been a great deal of research into XML query languages to enable the execution of database-style queries over XML files. However, merely being an XML query-processing engine does not render a system suitable for querying the Internet. A useful system must provide mechanisms to (a) find the XML files that are relevant to a given query, and (b) deal with remote data sources that either provide unpredictable data access and transfer rates, or are infinite streams, or both. The Niagara Internet Query System was designed from the bottom-up to provide these mechanisms. In this article we describe the overall Niagara architecture, and how Niagara finds relevant XML documents by using a collaboration between the Niagara XML-QL query processor and the Niagara "text-in-context" XML search engine. The Niagara Internet Query System is public domain software that can be found at http://www-db.cs.wisc.edu/niagara/.

1 Introduction

One of the most exciting opportunities presented by the emergence of XML is the ability to query XML data over the Internet. In our view, a query system for web-accessible Internet data should have the following characteristics. First, the query itself should not have to specify the XML files that should be consulted for its answer. This flexibility is a departure from the way current database systems work; in SQL terminology, it amounts to supporting a "FROM *" construct in the query language, and ideally would allow a user to pose a query, and get an answer if the query is answerable from any combination of XML files anywhere in the Internet. Secondly, a useful query system cannot assume that all the streams of data feeding its operators progress at the same speed, or even that all of the data streams feeding its operators will terminate. Once again, this is a departure from the way conventional DBMS operate. The Niagara Internet Query System is designed to have these characteristics. In this paper we describe how Niagara supports queries that do not explicitly name source XML files; support for network resident and streaming data is described elsewhere [STD+00].

The remainder of this paper is organized as follows. Section 2 gives a brief overview of XML and XML-QL and presents the overall top-level architecture of the Niagara Internet Query System. Section 3 describes the text-in-context Niagara Search Engine and the SEQL language, while Section 4 describes how XML-QL queries are evaluated. We conclude in Section 5.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 XML, XML-QL, and Top-Level Architecture of Niagara

2.1 XML and XML-QL

```
<book>
   <title> Java Programming </title>
   <price> 40 </price>
   <author id = “gosling”>
          <name>
               <firstname> James </firstname>
               <lastname> Gosling </lastname>
          </name>
   </author>
</book>
```

Figure 1: An Example XML Document

```
WHERE <book>
                <title> Java Programming </title>
                <author>
                    <lastname> $l </lastname>
                </>
          </> IN *
CONSTRUCT <lastname> $l </lastname>
```

Figure 2: An Example XML-QL Query

Extensible Markup Language (XML) is a hierarchical data format for information representation and exchange in the World Wide Web. An XML document consists of nested element structures, starting with a root element. Element data can be in the form of attributes or sub-elements. Figure 1 shows an XML document that contains information about a book. In this example, there is a book element that has three sub-elements — title, price and author. The author element has an id attribute with value “gosling” and is further nested to provide name information. Much more information about XML and related standards can be found on the W3C Web Site, http://www.w3c.org.

There are many semi-structured query languages that can be used to query XML documents, including XML-QL [DFF+99], Lorel [AQM+97], UnQL [BDH+96] and XQL [R98], and what appears to be the emerging standard, XQuery [CFR+01]. Each of these query languages has a notion of path expressions for navigating the nested structure of XML. In Niagara, we initially chose XML-QL as the query language, although we are currently switching to XQuery. Figure 2 shows an XML-QL query to determine the last name of the author of a book having the title “Java Programming.” The query is executed over the XML documents specified in the “IN” construct and the last names thus selected are nested under a <lastname> element. As mentioned in the introduction, one of the design goals of the Niagara query system is to allow the user the flexibility to query the web without having to explicitly specify each individual XML file to be queried. We thus extend XML-QL to support the “IN *” construct, whereby the query is (conceptually) executed over all the XML files present in the World Wide Web.

2.2 Top-Level Architecture of the Niagara Query System

Figure 3 illustrates the main components of the Niagara Internet Query System. Users craft XML-QL queries using a graphical user interface and send them to the query engine for execution. The connection manager in the query engine accepts queries for execution and is also responsible for maintaining sessions with clients. Each query received by the connection manager is parsed and optimized.

A crucial step in the optimization process is reducing the number of XML files to be consulted in order to produce the result of the query (especially in view of the “IN *” construct). Niagara accomplishes this reduction by extracting a search engine query from the XML-QL query. This search engine query is sent for execution to the search engine, which responds with a list of potentially relevant URLs. During execution, data from the relevant input sources is asynchronously fetched from the Internet by the data manager (if it is not already cached), and the results of execution are streamed to the user as they become available. Users can request partial

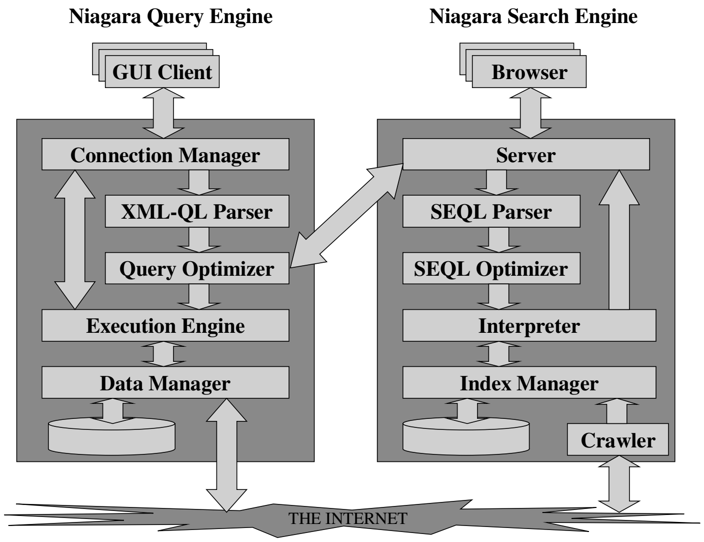

Figure 3: Niagara System Architecture

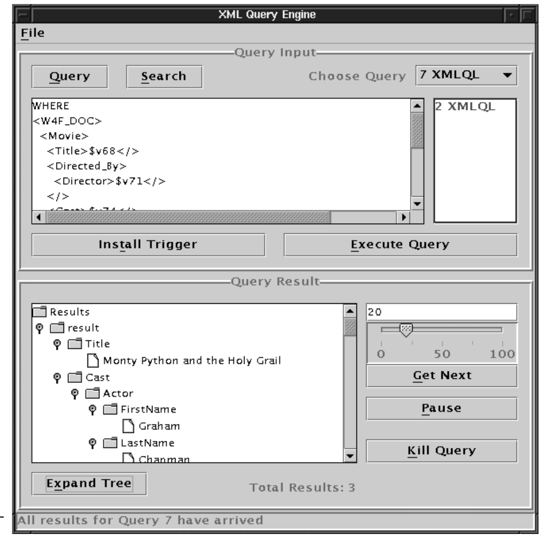

Figure 4: Query result screen shot.

results at any time during the execution. In this case, the execution engine returns the "result so far" while at the same time processing new input data coming off the Internet.

The Niagara search engine, in addition to its role in answering user XML-QL queries, can also be used as a stand-alone search engine for XML documents over the World Wide Web. The query interface to the search engine, in either case, is SEQL (Search Engine Query Language). SEQL queries are parsed and optimized in the search engine and the search engine interpreter executes the optimized execution plan. The results of the query execution are returned to the query engine or user. The inverted indices used for efficient SEQL execution are built and updated by the index manager. The index manager receives information about new and updated XML files from a crawler.

The GUI is a Java application that can also be run as an applet in a web browser. In addition to providing a simple graphical user interface to generate XML-QL, it is also capable of handling multiple concurrently executing XML-QL or SEQL queries. Both the query engine and search engine are also implemented in Java and are structured as multi-threaded servers. Since the query engine and search engine are individual servers, they run as separate processes (potentially on different machines). The next two sections describe the working of the search engine and the query engine in more detail.

3 The Niagara "Text-in-Context" Search Engine

A novel feature of the Niagara Internet Query System is that users do not need to specify source XML files in their queries. Rather, it is the responsibility of the system itself to examine the query, and from the query to determine the set of XML files that could possibly contribute to an answer to the query. Niagara does this through the use of its search engine, the Niagara "Text-in-Context" Search Engine.

When using an existing Internet search engine, the most common query is "find all the documents that contain these keywords." It is possible to construct more advanced searches based upon such properties as proximity and simple Boolean combinations of keywords. However, it is impossible to query based upon the role of the keyword in a document, because that information is not even available in the document itself. XML changes all this.

As a simple, admittedly contrived example, consider searching for all documents that have information about departures of a ship named "Montreal." One could go to one of the existing search engines and ask for the URLs of all documents that contain the string “Montreal,” with predictable results — the query will return thousands of documents, most of which have nothing to do with the ship named “Montreal.” Using the Niagara text-in-context search engine, one can instead ask for “all documents that contain departure elements that contain shipname elements that contain the string ’Montreal’.” It should now be clear what we mean by “text-in-context” — rather than just searching for words in documents, we search for containment relationships between elements and other elements (e.g., “departure element contains shipname element”) and also containment relationships between elements and strings (e.g., “shipname element contains the string ’Montreal”’).

The preceding paragraph is overly simplistic — a user looking for departures of ships named Montreal with a traditional search engine would be unlikely to search only on “Montreal,” they would be far more likely to search for “Montreal, ship, departure.” This will yield a far more focussed search than just giving the keyword “Montreal.” It is an open question how this search would fare when compared to the XML structural search. The structural search is more precise, but the value of this precision can only be determined empirically as we gain experience with the engine. Since in our experiments the structural approach is not significantly slower with respect to query evaluation, we have decided to use it.

3.1 SEQL

In this section we describe Search Engine Query Language (SEQL), the language executed by the search engine, briefly describe how the search engine evaluates SEQL, and how the XML-QL engine and the search engine interact. This process is described in more detail in [NDM+00].

SEQL is a simple language designed to specify patterns that can be matched by XML files. The output of a SEQL query is the list of URLs of the files that match the query. An atomic SEQL query is a word or an XML element tag. Such a query returns the URLs of the XML files containing the word or XML element tag, respectively. Complex SEQL queries can be built from the atomic SEQL queries using the binary operators “**contains**,” “**containedin**” and “**is**” In addition, SEQL supports a proximity operator and a numeric comparison operator.

SEQL also supports the standard Boolean connectives “**and**,” “**or**,” and “**except**,” which represent the intersection, union and difference of the results of the simple SEQL queries, respectively. Finally, SEQL supports a special construct “**conformsto**”, which is used to restrict the result to only the URLs of those XML files that are declared to conform to a given DTD.

3.2 Evaluating SEQL

We now turn our attention to the efficient evaluation of SEQL queries in the Niagara search engine. As mentioned in Section 2.2, the search engine has two logical parts, the crawler and the SEQL execution engine.

The crawler locates XML documents in the web. Since at the time of writing this paper there are relatively few XML files on the web, to evaluate the system we manually built a local collection of XML files, and then used this crawler to crawl the local subtree and “find” these files. The crawler passes URLs of XML files to the search engine to be indexed.

The indices used by the search engine are variants of inverted lists used for information retrieval. There are three categories of inverted lists used by the search engine, which are element lists, word lists and DTD lists. The search engine maintains one element list for every unique XML element name encountered in the crawled XML files. Each entry (or “posting”) of the element list has the form (fileId, beginId, endId) where fileId identifies a file containing the XML element, beginId specifies the beginning position of the element’s begin tag in that file, and endId specifies the ending position of the element’s end tag in the same file.

The search engine also maintains one word list for each unique text word encountered in the crawled XML files. Each posting in a word list for a given word is of the form (fileId, position) where fileId identifies a file containing the word and position indicates the position of the word in that file. Finally, the search engine maintains one DTD list for each unique DTD that the crawled XML files conform to. Each posting in a DTD list for a given DTD is of the form (fileId), where fileId identifies a file that conforms to that DTD. All three types of lists are maintained sorted by fileId to enable the efficient execution of SEQL queries.

4 Generating and Evaluating XML-QL Queries

In this section we describe how users pose queries using the Niagara GUI interface, and how those queries are evaluated by the query engine.

4.1 Extracting SEQL from XML-QL

The Niagara query engine sends SEQL queries to the Niagara search engine to determine the XML files over which to run an XML-QL query. To do so, the XML-QL query engine extracts a SEQL query (or queries) from the original XML-QL query. The XML-QL query engine extracts SEQL during query optimization.

The SEQL extraction process does not extract all possible constraints from the query; rather it uses heuristics to avoid generating SEQL that would be likely to be extremely inefficient to evaluate. The goal of the generated SEQL is to produce a superset of the URLs that need to be consulted to evaluate the XML-QL. It would be optimal if the "superset" were exactly the set actually required, but for efficiency of SEQL evaluation we do not always achieve this goal. Evaluating tradeoffs between the cost of the SEQL query and the precision with which it returns useful URLs is an interesting direction for future work.

4.2 A User Interface for XML Querying

In a classical relational database management query-builder interface, the basic approach is to start with the database schema. Clearly something analogous is needed for querying XML (no user is going to type in XML-QL!), but if a query is being posed over "all the XML files on the Internet" where is the schema?

In our system, we have taken the simple approach that this schema information is derived from document type descriptors (DTDs). Both the XML-QL engine and the text-in-context search engine have graphical user interfaces. To build a query, a user starts by selecting a set of DTDs to work with. After these DTDs have been selected, the GUI displays element names and users can do standard "point and click" or "drag and drop" query building over these DTDs. Once the query has been specified, and the user clicks on "submit query," an XML-QL query is generated (in the case of the XML-QL engine) or a Search Engine Query Language Query is generated (in the case of the text-in-context search engine.)

Note that the resulting query will not run solely over documents conforming to the DTDs selected by the user (unless the user specifies that this is desired by including a "conformsto" clause). Rather, the DTDs are used to generate a candidate set of tags over which to query. Any document that can match the query over these tags will be used in answering the query, whether or not it conforms to the DTD.

Clearly this is only a first step to building a good user interface. What is needed is a higher level mapping from user concepts to XML element vocabularies. We regard this area as important for future research. It is our hope that this sort of mapping will be done at a higher level than the query engine, in a layer that "understands" user-level concepts and can map them to schema information stored as DTDs or XML Schemas.

4.3 A Simple Query

For clarity and brevity of exposition, we present a very simple query. Consider the DTD in Figure 5 that describes XML documents representing movies. The elements are self-explanatory, with the exception of W4F_DOC, which is an element added by the wrapper that converted this data to XML [SA99].

```
<?xml encoding="ISO-8859-1"?>
<!ELEMENT W4F_DOC (Movie)>
<!ELEMENT Movie
(Title,Year,Directed_By,Genres,Cast)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
<!ELEMENT Directed_By (Director)*>
<!ELEMENT Director (#PCDATA)>
<!ELEMENT Genres (Genre)*>
<!ELEMENT Genre (#PCDATA)>
<!ELEMENT Cast (Actor)*>
<!ELEMENT Actor
        (FirstName,LastName)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
```

Figure 5: Example Movie DTD.

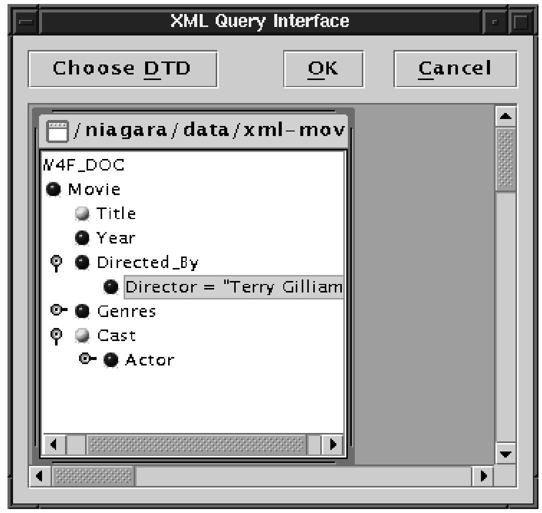

Figure 6: Niagara Query Interface Example.

Figure 6 shows a screen shot of the Niagara GUI specifying the query “retrieve the Movie title and the Cast of movies directed by Terry Gilliam” over the DTD in Figure 5. The XML-QL query that is generated in response to the query specified in the GUI is shown in Figure 7. Figure 8 shows the SEQL query the query engine extracts from this XML-QL query. In response to the SEQL query, the search engine returns three URLs, which have been filtered from over 600 XML documents that conform to the “movies” DTD. These URLs are passed to the query engine, which evaluates the XML-QL query, giving the result shown in Figure 4.

```
WHERE
<W4F_DOC>
 <Movie>
  <Title>$v68</>
  <Directed_By>
   <Director>$v71</>
  </>
  <Cast>$v74</>
 </>
</>
IN "*" conform_to
 "http://www.cs.wisc.edu/niagara/data/
    xml-movies/movies.dtd",
 $v71 = "Terry Gilliam"
CONSTRUCT
<result>
 <Title>$v68</>
 <Cast>$v74</>
</>
```

Figure 7: Generated XML-QL.

```
(W4F_DOC CONTAINS
        (Movie CONTAINS
              ((Directed_By
                CONTAINS (Director
                  IS "Terry Gilliam"))
                        AND
                        (Title AND Cast)
                )
        )
)
conformsto
   "http://www.cs.wisc.edu/niagara/data/
       xml-movies/movies.dtd"
```

Figure 8: SEQL Extracted from Terry Gilliam query.

5 Conclusion

The Niagara Internet Query System is designed to enable users to pose XML queries over the Internet. It differs from traditional database systems in (a) how it decides which files to use as input, (b) how it handles input sources that have unpredictable performance or may be infinite streams or both, and (c) in its support for large numbers of triggers. In this paper we have focussed on the interaction between the search engine and the query engine; other aspects of the system are described elsewhere [CDT+00, STD+00].

The Niagara project is on-going; much more information about the system is available from the project homepage, http://www.cs.wisc.edu/niagara.

Acknowledgements

Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by NSF Awards CDA-9623632 and ITR 0086002.

References

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, "The Lorel Query Language for Semistructured Data," International Journal on Digital Libraries, 1(1), pp. 68-88, April 1997.

[BDH+96] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, "A Query Language and Optimization Techniques for Unstructured Data," Proceedings of the 1996 ACM SIGMOD Conference, Montreal, Canada, June 1996.

[CFR+01] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, M. Stefanescu, "XQuery: A Query Language for XML." W3C Working Draft, available at http://www.w3c.org/TR/xquery/.

[CDT+00] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", Proceedings of the 2000 ACM SIGMOD Conference, Dallas, TX, May 2000.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, "XML-QL: A Query Language for XML", Proceedings of the Eighth International World Wide Web Conference, Toronto, May 1999.

[NDM+00] J. Naughton, D. DeWitt, D. Maier, et al., "The Niagara Internet Query System." Available from http://www.cs.wisc.edu/niagara/Publications.html.

[R98] Jonathan Robie, "The Design of XQL", Texcel Research, http://www.texcel.no/whitepapers/xql-design.html, November 1998.

[SA99] Sahuguet, Arnaud and Fabien Azavant. Web Ecology, "Recycling HTML pages as XML documents using W4F", Proceedings of the 1999 WebDB Workshop, June 1999.

[STD+00] J. Shanmugasundaram, K. Tufte, D. DeWitt, D. Maier, J. Naughton, "Architecting a Network Query Engine for Producing Partial Results", Lecture Notes in Computer Science, Vol. 1997, Springer-Verlag Publishers, 2001. A short version of this paper appeared in the proceedings of WebDB 2000 and is also available from http://www.cs.wisc.edu/ niagara/papers/partialResultsPerformance.pdf.