The Niagara Internet Query System

Jeffrey Naughton
Jianjun Chen
Jaewoo Kang
Naveen Prakash
Jayavel Shanmugasundaram
Ravishankar Ramamurthy
Yuan Wang
Rushan Chen

David DeWitt
Leonidas Galanis
Qiong Luo
Feng Tian
Chun Zhang
Bruce Jackson
Anurag Gupta

David Maier
Kristin Tufte
Computer Science Department
Oregon Graduate Institute

Computer Sciences Department
University of Wisconsin-Madison

Abstract: Many projections envision a future in which the Internet is populated with a vast number of Web-accessible XML files—a "World-Wide Database". Recently, there has been a great deal of research into XML query languages to enable the execution of database-style queries over these XML files. However, merely being an XML query-processing engine does not render a system suitable for querying the Internet. A truly 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. It finds relevant XML documents by using a novel collaboration between the Niagara XML-QL query processor and the Niagara "text-in-context" XML search engine. To handle infinite streams and data sources with unpredictable rates, it supports a "get partial" operation on *blocking* operators in order to produce partial query results, and *inserts synchronization packets* at critical points in the operator tree to guarantee the consistency of (partial) results. The Niagara Internet Query System is public domain software that can be found at http://www-db.cs.wisc.edu/niagara/. Category: Research.

1 Introduction

At the time of the writing of this article in early 2000, even a cursory glance at the research and trade press shows that XML is generating intense interest. There are venture funds created for the sole purpose of funding XML-related business opportunities; there are numerous large consortia dedicated to defining and standardizing DTDs for various communities; all major DBMS vendors have announced XML support in their products; and every major database research conference has at least one session devoted to XML issues. While HTML is still by far the dominant format for publishing data on the web, it appears very likely that in the near future a large portion of the web-accessible documents will be published in XML. From a database point of view, one of the most exciting opportunities this raises 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 need not have to specify the XML files that should be consulted for its answer. This flexibility is a radical departure from the way current database systems work; in SQL terminology, it amounts to supporting a "FROM *" construct in the query language. However, we think this is essential because a truly useful system will allow the 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. XML files fetched from some sites may come much more slowly than files fetched from other sites; furthermore, some of these "files" may actually be streams (consider a stock ticker news feed). These possibilities mean that the system must not "hang" waiting for a slow site, and that users must be able to request the result "so far" without having to wait for all the input streams to terminate, even in the presence of inherently blocking operations such as average, sum, nest and negation. Once again, this is a radical departure from the way conventional DBMS operate. The Niagara Internet Query System is designed to have these characteristics.

To support the "From *" clause, Niagara uses a novel collaboration between its XML query processor and its "text-in-context" XML search engine. When the XML query processor receives a query, expressed in a modified version of the XML-QL query language [8] (in XML-QL terminology, we have an "IN *" construct), it first parses the query to extract a search engine query. The search engine query is expressed in the Search Engine Query Language (SEQL, pronounced "seek-ul"), which supports Boolean combinations of predicates that specify containment relationships between XML elements and their contents. The SEQL query extracted from the XML-QL query is passed to the Niagara Search Engine. The Niagara Search Engine is an inverted list system optimized for evaluating SEQL. It works by crawling the web off-line, and building an index of all the XML files it encounters. The Search Engine evaluates the SEQL query utilizing the index and returns to the XML-QL query engine a list of URLs for the XML files that could possibly contribute answers to the original XML-QL query.

Once the Search Engine has returned a list of URLs, the Query Engine begins fetching the documents referenced by the URLs (first checking to see if it has copies of "hot" documents in the local cache). As we mentioned previously, it is possible that these documents could arrive at different rates, and after different initial delays. Furthermore, documents from different sites could feed into different leaves of the query evaluation tree. Finally, it is possible that some of the "documents" in question are really non-ending streams, which means a user could wait forever for an answer if the query contains blocking operators. A solution to these problems is to provide partial results to users. The Niagara Query Engine allows users to ask at any time "give me the results so far."

Supporting this kind of request over potentially streaming input sources imposes some constraints on the operators used to implement the query engine. First, all multi-operand operators are implemented in such a way that they can process data on any of their inputs at any time. This flexibility ensures that operators do not stall waiting for slow input streams when there are other (faster) inputs. Second, traditionally blocking operators (for example, hash join) are implemented using a non-blocking alternative (such as symmetric hash join or one of its variants 0[17][19]) when possible. If no non-blocking alternative is possible (for operators such as nest, average and except), the operator must support a "get partial" command. Upon receiving a "get partial" command, the operator passes to its successor in the tree the result it has computed at the point it receives the "get partial" command. Third, each operator handles changes (updates and deletions, in addition to insertions) because a partial result input may change or may no longer be valid.

Unfortunately, this combination of streaming processing and get partial commands introduces the possibility of inconsistency if the operator "tree" is really a DAG, since the computation on one branch of

the DAG may reflect input data that has not yet propagated up another branch of the DAG. To eliminate this sort of inconsistency, the Niagara Query Engine uses synchronization packets inserted at appropriate places in the DAG in response to "get partial" commands.

The Niagara Query Engine and Search Engine are public domain software, available at http://www-db.cs.wisc.edu/niagara/. The user interface provided with the query engine does not currently export the "get partial" command, although all the machinery necessary to support it is built into the prototype and has been tested and evaluated using a special-purpose interface.

The remainder of this paper is organized as follows. Section 2 gives a very 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, the SEQL language, and how SEQL queries are extracted from XML-QL queries. It also gives an example of the Search Engine and the Query Engine cooperating to answer queries. Section 4 covers the implementation of the Niagara Query Engine, and discusses the design of its operators and how it supports the "get partial" command. We give our conclusions in Section 5.

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

2.1 XML and XML-QL

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. Further information on XML can be found in [4][7].

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

Figure 2. Example XML Document

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

Figure 1. XML-QL query to find all books with the title "Java Programming"

There are many semi-structured query languages that can be used to query XML documents, including XML-QL [8], Lorel [1], UnQL [2] and XQL [20] (from Microsoft). Each of these query languages have a notion of path expressions for navigating the nested structure of XML. In Niagara, we use XML-QL as the query language mainly because it can specify joins and complex result construction naturally. XML-QL uses a nested XML-like structure to specify the parts of a document to be selected and specifies the structure of the result XML document using a result template. 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 earlier, 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

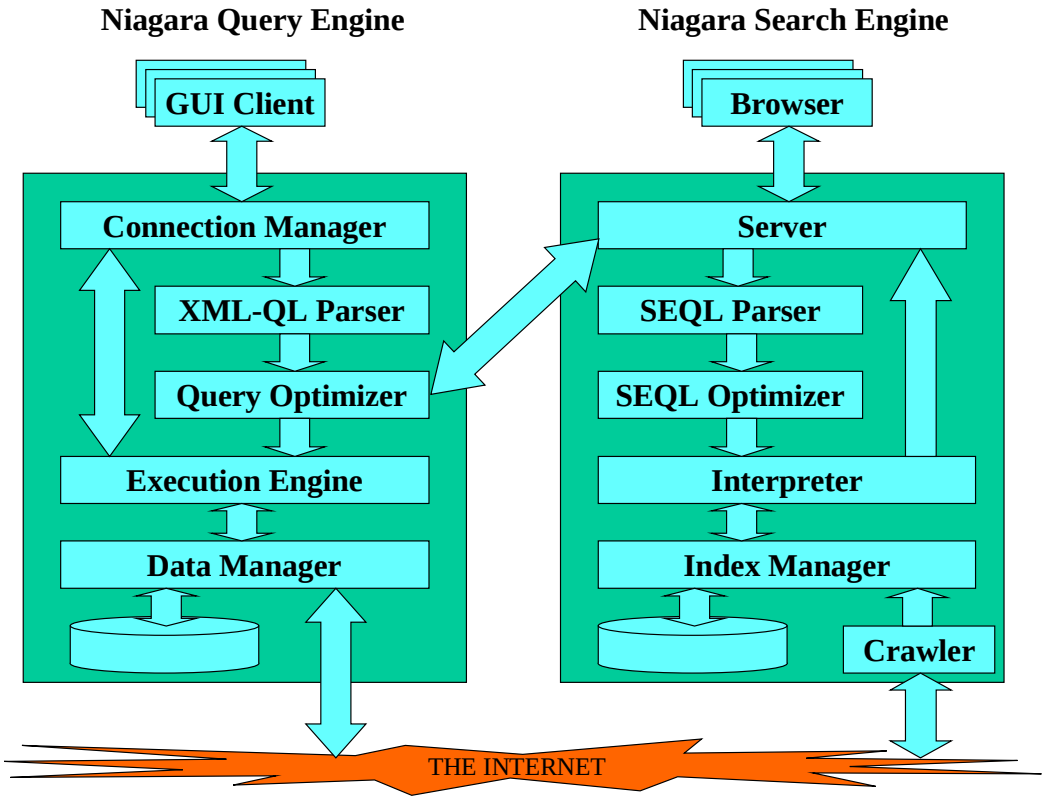

Figure 3: Architecture of the Niagara System

As mentioned earlier, the Niagara system has two main components – the Search Engine, which returns the URLs of the XML files relevant to a query, and the Query Engine, which evaluates XML-QL queries. This section provides an architectural overview of each sub-system and illustrates how the sub-systems interact.

Figure 3 shows the high level architectural overview of the Niagara Internet Query System. Potentially many users can 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 each client. Each query received by the connection manager is then 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). This reduction is done by extracting a search engine query from the XML-QL query, which asks for the URLs of the XML files that could possibly be relevant to the final result, from the user query.

This search engine query is sent for execution to the search engine, which responds with a list of URLs that the query engine needs to query over. The user query is further optimized by the query engine and then executed. 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 and when they become available. Users can request partial 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 search engine uses inverted indexes to efficiently determine the URLs of relevant XML files and these indices are consulted during SEQL query execution. 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 that constantly crawls the World Wide Web (in the background) for XML files.

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 (in potentially 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

As mentioned earlier, 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 own search engine, the Niagara "Text-in-Context" Search Engine.

There are a lot of search engines on the web: AltaVista, Hotbot, NorthernLight, Google, and many others. Why do we need another search engine? The answer is that each of these commercial search engines focus on HTML files, rather than XML files. (Rightly so; for the time being, the XML on the web is dwarfed by the HTML on the web. XML search engines are starting to appear, but they are still in their infancy. GoXML, http://www.goxml.com, for example, is a standard search engine that allows post-processing of results based upon XML tags in the documents.) However, Niagara has as its goal the querying of XML, and fortunately, XML documents support much more powerful and precise searches than HTML documents.

When using an existing search engine, one can essentially ask only "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.

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, as shown by queries Q1 and Q2 in Table 1. Complex SEQL queries can be built from the atomic SEQL queries using the binary operators "**contains"**, "**containedin"** and "**is"** (see Q3, Q4). In addition, SEQL supports a proximity operator and a numeric comparison operator. The proximity operator (added for compatibility with existing search engines) returns the URLs of the XML files that satisfy a restriction on the distance between two words contained in the file (see Q5). The numeric comparison operator returns the URLs of those files that have a numeric-valued element satisfying the comparison condition (see Q6).

To allow the composition of more elaborate queries, SEQL 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. (See Q7, Q8.) 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 (see Q9). SEQL is a highly orthogonal language and so a SEQL expression can appear anywhere a literal (string, number or element name) can appear. This orthogonality allows one to build up complex queries.

	SEQL Query	Result
Q1	"Java"	All XML files that contain the text string "Java."
Q2	book	All XML files that contain a "book" element.
Q3	book **contains** "Java"	All XML files that contain a "book" element that contains the text string "Java"
Q4	title **is** "Java Programming"	All XML files that contain a "title" element whose content is "Java Programming"
Q5	**distance**("Java", "Programming") < 5	All XML files that contain the text string "Java" and "Programming" less than 5 words apart
Q6	price < 30	All XML files that contain a "price" element whose contents is a numeric value less than 30
Q7	"Java" **and** "Programming"	All XML files that contain the text strings "Java" and "Programming"
Q8	book **contains** ("Java" and "Programming")	All XML files that contain a "book" element that contains the text strings "Java" and "Programming"
Q9	book **contains** "Java" **and conformsto** "http://www.cs.wisc.edu/bib.dtd"	All XML files that contain the text string "Java" and conform to the DTD "http://www.cs.wisc.edu/bib.dtd"

Table 1

It is important to note that the SEQL text-in-context query language subsumes the expressive power of standard, HTML search engine query languages. SEQL queries that are built using only strings and Boolean operators, thereby ignoring all tags, have the same basic semantics that regular search engine query languages have over plain text documents (although SEQL currently has none of the sophisticated facilities for ranking results that are incorporated into standard search engines.)

3.2 Evaluating SEQL

Given SEQL query specifications, we now turn our attention to the efficient evaluation of these queries in the Niagara search engine. As mentioned in Section 2.2, the search engine has two logical parts – the first part deals with crawling the web and indexing all the encountered XML files while the second part deals with exploiting these indices to optimize the execution of SEQL queries.

The Crawler component of the search engine is essentially used to locate XML documents in the web. Since at the time of writing this paper there are so 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 [21]. 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. An element list associated with a given element name stores information about the files that contain the XML element, and the position of the XML element in those files. More precisely, 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. For example, if the list for the <book> element contains a posting (26, 5, 10), this would mean that the file with id 26 has a book element that begins at position 5 and ends at position 10. The position in a document is simply the word number in the document.

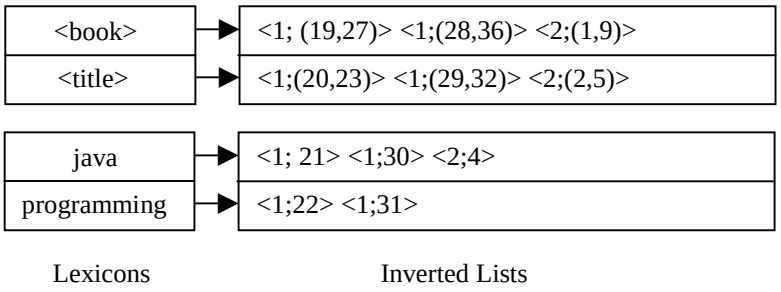

Figure 4: Example illustrating element and word lexicons and the associated inverted lists

Similar to element lists, the search engine 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. Figure 4 gives a logical view of a simple inverted index. It shows two kinds of lists, one for elements and one for text words. This index indicates that there is a "<book>" element in document 1 from word number 19 to 27, and that the book element occurs a second time in document 1 from word number 28 to 36. It appears again in document 2 from word number 1 to 9. Similarly, the word "java" appears in document 1 at word numbers 21 and 30 and in document 2 at word number 4. The search engine also 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.

When the search engine receives a SEQL query for execution, it parses it into a tree of binary and unary operators, where each operator corresponds to a SEQL construct such as "contains" or "and". Each operator operates on its input(s), which are inverted list(s), and produces an output, which is also an inverted list. As an example, consider SEQL query Q3 in Table 1. The search engine will create an operator tree with a single operator corresponding to "contains". This operator will fetch the element list for the <book> element and the word list for the text string "Java", and merge the two. The merge logic will output a file with Id *fid* if and only if there is a book posting of the form (fid, beginElemPos, endElemPos) and a word posting of the form (fid, beginWordPos) and $beginElemPos < beginWordPos < endElemPos$.

3.3 Extracting SEQL from XML-QL

As described in Section 2.2, 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. This section describes the extraction process.

The XML-QL query engine extracts SEQL during query optimization using the logical plan produced by the XML-QL parser, and also a data structure called "the schema". The schema data structure contains information about parent and child relationships between tags, as well as all the selection predicates from the original XML-QL query. The following two types of restrictions are extracted from the schema and are used to construct SEQL queries:

(1) selection predicates, for example, <title> Java Programming </title>

(2) containment relationships among tags, for example, <book><title>…</title></book>

Selection predicates are translated into **is** and numerical comparison operations, while containment relationships are translated into **contains** or **containedin** operations. Boolean relationships among predicates in the original XML-QL query are generally maintained in the translated SEQL query.

The extraction process starts with the translation of the XML-QL predicates into SEQL predicates, because of their ready availability in the "select nodes" of the logical query plan. A simple predicate in XML-QL takes the form:

expression op value

where "expression" is a regular path expression, "op" is any operator supported by the query engine, and "value" is some constant. Of course, simple predicates can be used as building blocks for more complex predicates through the use of recursion. XML-QL predicates yield both SEQL selection predicates and containment constraints.

We now discuss the process of translating XML-QL predicates to SEQL queries. 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.

The constraints used in translating XML-QL predicates to SEQL are listed below. Examples of the translations are provided in Table 2. In this table, each XML-QL predicate is shown being translated to a single SEQL query. In practice, as in the example in Section 3.4.2, these SEQL queries are combined, when appropriate, to make a larger SEQL query to maximize the filtering of XML files (URLs).

(1) XML-QL predicates involving negation are not extracted and added to the SEQL query (see T1 and T2 in Table 2).

(2) Numeric predicates are only extracted if they are equality predicates. Inequality predicates are not extracted and are not "pushed" to the search engine because they are potentially expensive for

the search engine to evaluate and are not expected to significantly restrict the number of URLs the search engine returns to the query engine.

(3) If a predicate involves a tag variable (that is, a range variable, such as $1 in Figure 2), the predicate is ignored. It is possible that in some cases analysis could restrict the tag variable to a small set of element domains, which could then appear in the SEQL predicate, but we do not currently perform this analysis.

(4) A predicate involving a non-numeric value is always translated to an **is** operation, whereas a predicate involving a numeric value is translated to a numerical predicate. (See T3 and T4 in Table 2.)

	XML-QL Predicate	**SEQL Query**
T1	A = 10 and not (B = "hello")	A = 10
T2	A = 10 or not (B = "hello")	null (nothing corresponding to this predicate appears in the SEQL query)
T3	<title> Java Programming </title>	title **is** "Java Programming"
T4	<year> 1999 </year>	year = 1999

Table 2

Another interesting issue is how to process predicates involving regular path expressions, that is, predicates on path expressions that may contain embedded "*", "?", "+", or "|" operators. These are handled as described below and illustrated in Table 3.

(1) A path expression without "*", "?", "+", or "|", is translated into containment. (see P1, Table 3) .

(2) The components of the regular path expression with "?", "*" are ignored, but not components with "+." (see P2 and P3, Table 3).

(3) "|" is translated into OR (see P4, Table 3).

These rules are recursively applied, and A, B, and C themselves can be a complex path expressions.

	Path Expression	**SEQL Query**
P1	A.B	A **contains** B
P2	A.B*.C	A **contains** C
P3	A.B+.C	A **contains** B **contains** C
P4	A.(B|C)	(A **contains** B) **or** (A **contains** C)

Table 3

Once all the predicates have been extracted, we are still not finished, since there are potentially other containment relationships that can be extracted from the schema data structure. Walking this parse structure generates these relationships. For example, if an XML-QL query had an expression such as

WHERE <book><title> "Java Programming" </title></book>

the SEQL corresponding to this would be

book **contains** title **is** "Java Programming"

The next section contains a complete example of an XML-QL query and the SEQL that is generated.

3.4 An End-to-End Example

We close this section with two examples that follow two XML-QL queries from the time they are generated by a user until the answer has been computed. We begin by discussing the user-interface issues that arise in building a system such as Niagara.

3.4.1 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. From this schema, a user can decide which tables to query, which attributes to include in the result, any join or selection predicates, and so forth. 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?

The Niagara XML-QL query processor and text-in-context search engine do not need any schema information to evaluate queries. Neither engine "interprets" the element names and constants used in queries, they just "match" these element names and constants with the tags and element values found in the XML files over which they are querying. However, once again, this flexibility does not mean that the user does not need schema information. To the contrary, in order to pose a query, the user must know some element names, or he or she has nothing to refer to in the query.

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.

3.4.2 XML-QL Query Processing Example

Next we give an example of how the XML-QL query process works. For clarity and brevity of exposition, we present a very simple query. Consider the DTD in Figure 5 that describes XML documents

```
<?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)>
```

Figure 5: Movie DTD

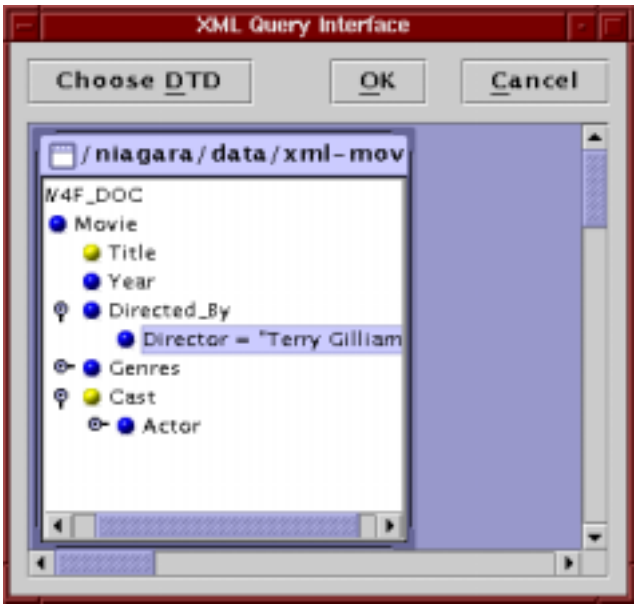

Figure 6: Screen shot of query interface

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 [22].

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 9.

From Figure 9 we can also see some additional features of the Niagara GUI. The buttons "Query" and "Search" toggle between the XML-QL and the Search Engine query-building GUIs. The "XMLQL" button displays the XML-QL for the current query; the box below this button lists the currently defined queries, which can be chosen for execution by selecting them. The Install Trigger button refers to the trigger engine portion of Niagara, which is described elsewhere [6]. The lower pane of the GUI deals with query results. The window in Figure 6 is currently displaying the results of the movie query. The "Get Next" button returns the next 20 results to the screen (in this case there are only three results), while the Pause and Kill Query buttons control query execution.

4 The Niagara Query Engine, Streaming, and Partial Results

To this point, we have focussed on how Niagara determines a relevant set of XML files for a given XML-QL query. We have treated the XML-QL query engine as a black box. Furthermore, for clarity of exposition, we have treated all the XML files in the examples as if they are located local to the system running the XML-QL query processor and the search engine. In this section we focus on the query engine itself, and further focus on the novel features of the query engine that are designed to make it effective in an Internet environment.

Using the Niagara search engine to determine the URLs of the XML files relevant to a given query is but the first step in providing a query capability over the Internet. The data in the XML files identified to be relevant has to further be processed by the query processor to produce an XML result. Several challenges arise because the XML files of interest can be distributed, and because evaluating an XML-QL query often requires extensive use of "blocking" operations such as "nest". A simplistic solution to the first problem is to have all the XML documents cached at the query engine. Though this approach is viable under certain conditions, it cannot be used in general because (a) it is not scalable, especially in view of the dramatic growth in the content and size of the Internet, (b) some of the web content is highly dynamic, such as auction bids, that makes caching them impractical, and (c) some of the XML documents of interest may actually be "continuous feeds", such as stock quotes, that cannot be cached. This implies that the Niagara query engine needs to deal with geographically distributed data sources, potentially reachable only through slow and unreliable communication channels.

Previous approaches to querying widely dispersed data have concentrated either on the modification of query plans so that the network delays can be (partially) hidden from the user [18] or on the implementation of such as joins using non-blocking implementations 0[17]. While these approaches partially address the problems of low network bandwidth and unavailable sites, they do not address the general problem because a query may have some intrinsically blocking operations, such as nest and average, which traditionally need to see all of the input before producing any output. This problem assumes special significance in the context of querying and constructing XML documents because they are heavily nested and the nest operation is inherently blocking. In such cases, the user will have to wait until all the input XML data has been received in order to see even the first result.

```
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
```

Figure 7. XML-QL query generated in response to the query shown in Figure 6.

```
(W4F_DOC CONTAINS
      (Movie CONTAINS
           ((Directed_By
             CONTAINS (Director
              IS "Terry Gilliam"))
                  AND
                  (Title AND Cast)
           )
      )
)
```

Figure 8. SEQL query generated by the Query Engine for the XML-QL query in

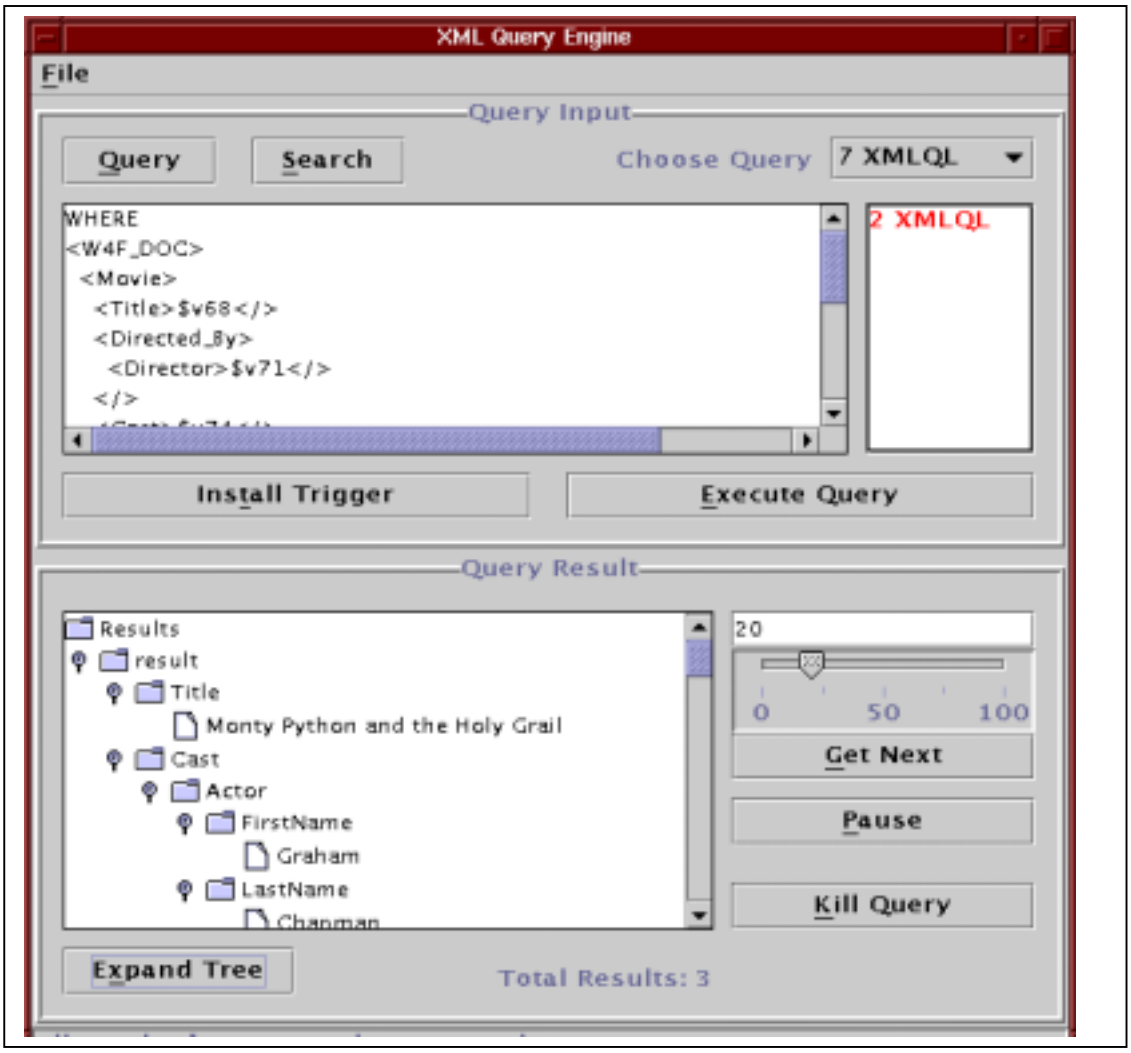

Figure 9: Screen shot showing result of query from Figure 6.

To solve this problem, Niagara is architected to provide partial results to users. Thus, users can see incomplete results of queries as they are executed over slow, unreliable sites or when the queries are long running (or never terminate!). For example, if a user wants to find all BMW cars that cost less that 90% of the average price of cars in its class, except those that appear on salvage lists, it may be desirable to display initial results before all the documents having price and salvage car information have been processed. As we have said, previous solutions to the problem of producing partial results [9][16] are for specific aggregate operations and limit the aggregate operations to appear at the top of the query plan tree or be nested at most one level. As a result, these techniques fail to extend to important blocking operations such as nest and negation that can appear anywhere in the query. Thus queries such as the one above cannot be handled in a partial-result mode by existing techniques even assuming a centralized database system.

The Niagara query engine has been designed from the start to deal with remote data sources and at the same time to provide users the flexibility to see the partial results of computation at any time. The core of

its architecture is a general framework for producing partial results for queries involving any blocking operator. A key feature of this framework is that it provides a mechanism to ensure consistent partial results with unambiguous semantics even in the face of massive distribution of the input data sources. The framework is also general enough to allow blocking and non-blocking operators to be arbitrarily intermixed in the query tree. The rest of this section describes these novel features of the Niagara query engine in more detail. The first part of the discussion focuses on the architecture of the query engine while the second part delves into the implementation details of query engine operators. In the interest of space, we omit the discussion of more traditional query engine components, such as the execution engine and the query optimizer.

4.1 The Niagara Query Engine Architecture for Producing Partial Results

In order to architect a query engine to produce partial results over widely distributed data sources, we first need to formally define the semantics of the partial results of a query.

Definition: Let Q be a query with n inputs and let $Q(I_1, \ldots, I_n)$ represent the result of query Q on inputs $I_1, \ldots, I_n$. A *partial result* of the query Q, given inputs $PI_1, \ldots, PI_n$ is $Q(PI_1, \ldots, PI_n)$, where, for $1 \leq i \leq n$, $PI_i \subseteq I_i$.

Intuitively, a partial result of a query on a given set of inputs is the result of the query on a (possibly) different set of inputs such that each input in the new set is a subset of the corresponding input in the old set.

Given the above definition of partial results, the following questions immediately come to mind. Is it possible to use traditional query engine architectures to produce partial results? If not, then what modifications are needed? Are there any new issues that arise? The following sections are devoted to answering these questions. We first briefly outline the structure of traditional query engines. We then identify some key properties of operators, not supported by traditional query engine architectures that are crucial for producing partial results. We also identify consistency anomalies that can arise during partial result production and propose a solution to tackle this problem. The result is a flexible, general architecture that produces consistent partial results.

4.1.1 The Traditional Query Engine Architecture

In traditional query engines, queries are executed by a collection of operators, each of which transform its input stream(s) and produce an output stream. The inputs to each operator are monotonically increasing, that is, data is only added to the streams, never updated or removed.

In many cases, the operator graph structure is a tree, but not always. As an example, consider a query that asks for all cars that have a price less than 90% of the average price of cars in their category. A graph representation of the operator tree of this query is shown in Figure 10 (the average computes the average price of cars in each category and the join relates it to the price of individual cars). If the query were executed over a network where the information about cars is present at one site and all the processing (Average and Join) are performed at another site, then there is wasteful communication because Car information is transmitted twice over an expensive network. A more efficient solution is to send Car information just once and "replicate" it at the site where the processing is performed. This solution is shown in Figure 11. Note that the operator graph is no longer a tree but a Directed Acyclic Graph (DAG).

Common sub-expressions within a single query can also give rise to DAG operator graphs for the same reason.

Given the traditional query engine architecture described above, what needs to change in order to produce partial results? The architecture described above is inappropriate or inadequate for two main reasons: (a) unsuitable assumptions about the properties of operators and (b) lack of synchronization support for producing consistent partial results, particularly for DAG operator graphs. We discuss and propose solutions for these problems in the next few sections.

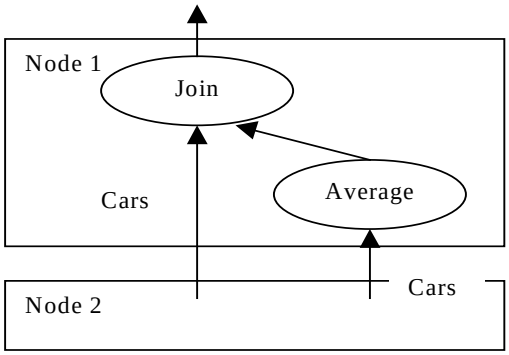

Figure 10

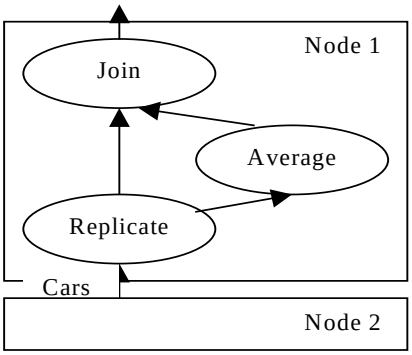

Figure 11

4.1.2 Properties of Operators for Producing Partial Results

Traditional implementations of operators are not suitable for partial result production because they do not satisfy some key characteristics required for producing partial results. The following three properties summarize the key requirements for such operators:

1) **Flexible Input Property:** Operators should not stall waiting for input from a particular input stream if there is some input available on another input stream. The motivation behind this property is that in a network environment, traffic delays may be arbitrary and data in some input streams may arrive earlier than data in other input streams and it may be impossible to determine this information a priori.

2) **Anytime Property:** At any time, each blocking operator should be able to put the "current" result, based on the data seen so far on its input stream(s), into its output stream. Note that non-blocking operators always put the current results based on the input streams immediately, thus satisfying the Anytime property.

3) **Non-Monotonic Input-Output Property:** Each operator has to deal with input streams (and produce output streams) that are not monotonically increasing. This is a direct result of requiring inherently blocking operators such as nest, groupby, and aggregates to produce partial results. Consider the output stream of a nest operator that nests a set of (author, book) pairs on author. An initial partial result is produced by the nest operator, which includes a group for an author named Smith containing two books *Database Design* and *Object Databases for the New Millennium*. By the time the second partial result request is issued, the tuple (Smith, *New Book*) has been received and processed by the nest operator, so Smith's group now contains three books. Thus the nest operator must somehow

communicate to the operator above it the addition of the book *New Book* to Smith's group. This requires changes (not just addition) to be propagated via streams.

Note that the flexible input and the anytime properties are required for the currency of partial result, while the non-monotonic input property is required for the correctness of partial results. In any case, the design and implementation of operators satisfying the properties above is crucial in designing a flexible system capable of producing partial results and are discussed in more detail in Section 4.2.

4.1.3 Synchronization

In addition to more flexible operator properties, the fact that the operator graph can be a DAG has important implications for the architecture of a system designed to produce partial results. The definition of partial results requires that the partial output be the result of executing the query on a subset of the inputs. This limitation implies that any data item that is replicated must contribute to the partial result along *all* possible paths to the output or not contribute at all to the output (along any path). This restriction is necessary to avoid anomalies such as selecting cars below the average price, without the car's price itself being used to compute the average (see Figure 10). Note that this is not an issue when constructing only final results because each operator produces results based on all of the inputs it sees. This potential inconsistency is an artifact of the flexibility we desire of interrupting input streams to produce partial results.

A related issue also arises when an operator logically produces more than one output data item corresponding to a single input data item. This could happen, for instance, in a Join operator when a single tuple from one input stream joins with more than one tuple from the other input stream and produces many output tuples. Another example where this can arise is while unnesting many set sub-elements (say employees) from a single element (say department) in XML documents. In these cases

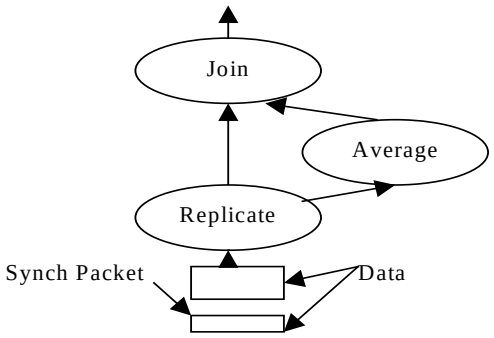

Figure 12: Use of synchronization packets

again, we need to ensure that the partial query result includes the effects of all or none of the output data items that correspond to a single input data item.

We ensure consistent partial results using the idea of *synchronization packets*. Conceptually, these packets are inserted in the input streams of the query whenever a partial result is desired. This situation is shown in the figure above. These synchronization packets are replicated whenever a stream is replicated and their main function is to "synchronize" input streams at well-defined points so that operators see

consistent partial inputs. More precisely, each operator sees all the data "before" the synchronization packets in the production of partial results. In Figure 12 above, the Join operator thus sees all the data before the synchronization packets in the production of partial results. This ensures that every data item reaching the join directly from replicate is also reached through average (and vice versa). The problem now is to determine how the synchronization information is to be propagated up the operator graph. We achieve this by enforcing the following rules for producing partial results at every node in the operator graph:

- If there is a synchronization packet received through an input stream of an operator, then no further inputs are taken from that input stream until synchronization packets are received through all input streams
- Once synchronization packets are received from all input streams of an operator, the operator puts its partial results (in the case of a blocking operator) and synchronization packets into its output stream(s). It is important to note that the Anytime property of blocking operators is used here in partial result/synchronization packet propagation.

  The following theorem shows that these rules are sufficient to guarantee consistent partial results.

Theorem: *If the synchronization packets are inserted into the input streams when partial results are desired, the synchronization rules guarantee that the partial results produced are consistent.*

We omit the proof of this theorem due to space considerations. The key idea in the proof is to use induction on a topological ordering of the operators in the graph.

4.1.4 Partial Request Propagation and Generation

In the discussion in the previous section, we assumed that synchronization packets are inserted into the input streams when partial results are required. However, one of the primary requirements for producing partial results is that the user or application be able to convey this information to the operators in the operator graph. Typically, the user or application has access only to the output stream of the top-level operator. This is because the operators of the query can be distributed at various sites, connected by network streams. What is thus required is a mechanism to be able to propagate user or application requests down the operator graph. This can be achieved by propagating control messages "down stream", all the way to the base of the operator graph. Once the control messages reach the base of the operator graph, there has to be some mechanism to intercept the partial result request control messages and insert synchronization control messages. This is provided by means of a "partial" operator.

Partial operators are added to the base of the operator graph and they perform two simple functions: (1) propagate the data from the input stream unchanged to the output stream and (2) on receiving a partial result request from an output stream, they send a synchronization packet to their output streams. Besides providing an automatic way of handling synchronization packets, partial operators introduce optimization possibilities. The key observation is that partial operators can be moved higher up in the operator graph (and even merged) under certain conditions. This "transformation" of the operator graph is likely to lead to better response times and more efficient execution. The response times are likely to be better because the partial result request and synchronization packets do not have to travel far along the operator graph – this benefit can be substantial in the presence of network streams. Execution is also likely to be more

efficient because (a) there is less overhead for partial result request and reply and, more importantly, (b) there is no need to synchronize operators that are below partial operators in the operator graph.

4.2 The Implementation of the Niagara Query Operators

The previous section described a general architecture and identified certain abstract properties that query operators need to satisfy in order to produce partial results. We now turn out attention to an implementation of query operators that satisfy the required properties. As mentioned earlier, we use non-blocking, flexible input implementations for operators wherever possible. For example, joins are implemented using symmetric hash join and symmetric nested loops join algorithms (or their variants). The algorithms in this section extend such flexible input, non-blocking algorithms to satisfy the non-monotonic input-output property and further, identify blocking operator implementations satisfying all three desirable operator properties.

The key to designing operators having the non-monotonic input-output property is to have them process changes (insertions, updates and deletions) in the input and produce associated changes as output. In this sense, the Niagara query operators are similar to the differential operators of the CQ Project [11], in which standard operators including selects, projects, and outer joins are extended to handle changes. Our system, however, handles changes (insertions, deletions, and updates) as the query is being executed as opposed to [11], which proposes a model for periodic re-execution of continual (trigger) queries. This extension gives rise to new techniques for handling changes as the operator is in progress. In the following sections, we provide brief descriptions of the standard differential algorithms and add a description of a new differential nest operator to illustrate the implementation of a differential blocking operator. We also contribute by commenting on the implementation of the differential operators.

In order to illustrate the working of the query operators, consider the example in Figure 13 where all the books of an author are nested and then joined with the author information. In order to produce partial results, the nest operator must be able to produce the "difference" between sets of partial results and the join operator must be able to process that "difference". We accomplish this by having all operators produce and consume tuples that consist of the old tuple value and the new tuple value, as in [11]. Since the partial results produced by blocking operators consist of differences from previously propagated results, each tuple produced by a blocking operator is an insert, delete or update. For inserts, the old value is null; for deletes, the new value is null; and for updates both the old and new value are valid. Scan operators produce all inserts. Note that a stream of inserts is identical to a stream of tuples in a traditional database system. Unless otherwise stated, updates are processed, without significant performance loss, as deletes followed by inserts.

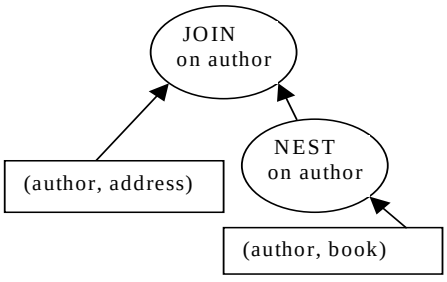

Figure 13

We now describe the implementation of various operators in the Niagara query engine. In the interest of space, we have only picked some representative operators and have not included details of other operators such as average, except , etc. These operator have "bag" semantics. They may input or output the same value several times, and must keep track of the number of times they've received a given tuple (and not remove it totally until an equal number of deletes have been received).

Differential Select: Differential Select propagates inserts (deletes) if and only if the new (old) value meets the selection criteria.

Differential Join: *O*ur implementation of Differential Join is based on a symmetric hash join; extensions to other join algorithms are straightforward. A Differential Join joining relations A and B works as follows. Upon receipt of an insert of a tuple τ into relation B, τ is joined with all tuples in A's hash table and the joined tuples are propagated as inserts to the next operator in the tree. Finally τ is inserted in the hash table for relation B for joining with all tuples of A received in the future. Upon receipt of a delete of a tuple τ from relation B, τ is joined with all tuples in A's hash table and the joined tuples are propagated as deletes to the next operator in the tree.

Differential Project: Differential Project propagates updates as updates, deletes as deletes and inserts as inserts with appropriate attributes projected out.

Differential Nest: Differential Nest is described in relation to a hash-based nest; extensions to other nest implementations are straightforward. Inserts are treated just as tuples would be in a traditional nest operation and are inserted into the appropriate entry in the hash table. For deletes, Differential Nest probes the hash table to find the affected entry and removes the deleted tuple from that entry. For updates, if the grouping value is unchanged, the appropriate entry is pulled from the hash table and updated, otherwise, the update is processed as a delete and insert. All changes are propagated only upon receipt of the subsequent partial result request.

5 Conclusions

The Niagara Internet Query System is designed to enable users to pose XML queries over the Internet. It differs radically from traditional database systems in (a) how it decides which files to use as input, and (b) how it handles input sources that have unpredictable performance or may be infinite streams or both. We have completed the prototypes described in this paper and made them available from our web site.

A great deal of future work remains. We are in the process of translating the prototypes from Java to C++. Our experience with the Java versions have convinced us that we cannot get the performance we desire without making the change. We are also investigating building parallel versions of the search engine and query engine, in order to handle very large data volumes and large numbers of queries.

Acknowledgements

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

References

[1] 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.

[2] 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.

[3] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, C. M. Sperberg-McQueen, L. Wood, J. Clark, "W3C XML Specification DTD", http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0", http://www.w3.org/TR/REC-xml.

[5] Sergey Brin, Lawrence Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine", Proceedings of the Seventh International World Wide Web Conference, Brisbane, Australia, April 1998.

[6] 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 (to appear).

[7] R. Cover, "The SGML/XML Web Page", http://www.oasis-open.org/cover/xml.html.

[8] 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.

[9] J. M. Hellerstein, P. J. Haas, H. Wang, "Online Aggregation", Proceedings of the 1997 ACM SIGMOD Conference, Tuscon, AZ, May 1997.

[10] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, D. S. Weld, "An Adaptive Query Execution System for Data Integration", Proceedings of the 1999 ACM-SIGMOD Conference, Philadelphia, PA, June 1999.

[11] L. Liu, C. Pu, R. Barga, T. Zhou, "Differential Evaluation of Continual Queries", Proceedings of the 16th International Conference on Distributed Computing Systems. May, 1996, Hong Kong.

[12] L. Liu, C. Pu, R. Barga, and T. Zhou, "Differential Evaluation of Continual Queries", Technical Report TR-95-17, Department of Computer Science, University of Alberta, 1995.

[13] V. Raman, B. Raman, J. M. Hellerstein, "Online Dynamic Reordering for Interactive Data Processing", Proceedings of the 1999 VLDB Conference, Edinburgh, Scotland, September 1999.

[14] Arnaud Sahuguet, Fabien Azavant, "Looking at the Web through XML Glasses", Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems, Edinburgh, Scotland, September, 1999.

[15] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities", Proceedings of the 1999 VLDB Conference, Edinburgh, Scotland, September 1999.

[16] K. Tan, C. H. Goh, B. C. Ooi, "Online Feedback for Nested Aggregate Queries with Multi-Threading", Proceedings of the 1999 VLDB Conference, Edinburgh, Scotland, September 1999.

[17] T. Urhan, M. J. Franklin, "XJoin: Getting Fast Answers from Slow and Bursty Networks", University of Maryland Technical Report, UMIACS-TR-99-13, 1999.

[18] T. Urhan, M. J. Franklin, L. Amsaleg, “Cost Based Query Scrambling for Initial Delays”, Proceedings of the 1998 SIGMOD Conference, Seattle, WA, June 1998.

[19] A. N. Wilschut, P. M. G. Apers, “Data Flow Query Execution in a Parallel Main Memory Environment”, Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991), Miami Beach, Florida, December, 1991.

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

[21] D. E. Knuth, The Art of Computer Programming. Volume III: Sorting and Searching, Addison-Wesley, 1973.

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