A Distributed Query Engine for XML-QL

Paramjit Oberoi and Vishal Kathuria
University of Wisconsin-Madison
{param,vishal}@cs.wisc.edu

Abstract: *This paper describes a distributed Query Engine for executing queries on XML documents. It supports complex pattern matching and a single pattern can be used for not only expressing selections and projection but also for a joins of one part of the document with another parts of the same document. The patterns can contain path expressions and tag variables as well. It is a multithreaded pipelined query engine with each operator running as a separate thread. A single query engine can execute multiple queries at the same time and execution of a single query can be distributed across different servers. The distributed servers are totally symmetric and queries can be submitted to any server. The server conforms to XML and proposed XML-QL standards and all communication among servers follow these specifications. This makes sure that even the query engines from other vendors can participate in a distributed query execution.*

1 Introduction

XML (eXtensible Markup Language) [1] is a subset of SGML that has been proposed by W3C. The goal of XML is to provide many of the SGML benefits not available in HTML and to provide a language that is easier to learn and use. It is used to create data in a semi-structured format and it is possible to do much more complex queries on XML documents than is it possible on HTML. These features make XML an excellent language for storing data, especially on internet. This paper describes a Query Engine for XML documents which can execute queries specified in XML-QL, a query language for writing queries for XML documents.

The paper is organised as follows. Section 2 gives a brief description of XML-QL. If you are familiar with XML-QL then you can skip this section. Section 3 describes the features implemented by our query engine (hereafter referred to as QE) that are used in extracting the data from XML documents. Section 4 describes the features that can be used to create a resulting XML document from the data extracted by the engine. Section 5 and 6 describe motivation and methodology for distributed operation.

2 XML-QL: A Query Language for XML documents

XML-QL uses element patterns to match data in an XML document. an example is

```
WHERE
<book>
  <title> $t </>
</> IN "http://www.a.com/book.xml
```

This query would match for the given pattern in book.xml and return all the book titles. XML-QL allows the result to be produced as yet another XML document. It allows the format of the resulting XML document to be specified using a CONSTRUCT clause. Therefore a query like

```
WHERE
<book>
  <title> $t </>
</> IN "http://www.a.com/book.xml
CONSTRUCT
<mein_book>
  $t
</>
```

would result in an XML document like

```
<queryresult>
   <mein_book>
      Java Programmming Language
   </>
   <mein_book>
      Linux Kernel Hackers Guide
   </>
</>
```

3 Data Extraction

3.1 Pattern Matching

The QE matches the simple patterns like the one described in section 2. Such patterns can be used to specify selections and projections on the documents. Pattern matching provides very powerful and flexible semantics that allow the user to express a wide variety of queries in a compact fashion.

3.2 Self Joins

Suppose there is an XML document appleton.xml containing the given and family names of the people living in Appleton. The data required is a list of all strings that are used as a given name and as a family name. This query can be represented by the follwing pattern.

```
WHERE
<personnel>
  <person>
    <name>
      <given> $g </>
    </>
  <person>
    <name>
      <family> $g </>
    </>
  </>
</>
in  "http://www.wi.com/appleton.xml"
CONSTRUCT
<name> $g </>
```

Speaking in relational terms, this query is a projection of appleton.xml on the field *given* (say resulting in `doc1`) and a projection on field family (say resulting in `doc2`) and then a join on `doc1` and `doc2`. There are many complex self join queries possible with the patterns. This query was a vertical split of the document and join of the resulting parts. One could easily do horizontal split and join of the resulting parts. The query engine also handles the case where there is a series of horizontal and vertical splits into different parts and there is an n-way join between those parts. For example suppose a user is interested in finding a string which is a manager's given name, a workers family name and a child's name in GE. The query can be expressed as

```
WHERE
  <personnel>
    <person>
      <name>
        <given> $g </>
      </>
      <position> Manager </>
    </>
    <person>
      <name>
        <family> $g </>
      </>
      <position> Worker </>
    </>

    <person>
      <child>
        <name> $g </>
      </>
    </>
  </>
in  "http://www.wi.com/ge.xml"
CONSTRUCT
<name> $g </>
```

3.3 Path Expressions

The tag inside a specified pattern can also be a regular path expression. Our query engine implements all the path expressions that are specified in the XML-QL specification [2].

3.4 Predicates

Each clause in the query has a pattern as well as a predicate associated with it. A predicate is an arbitrary boolean expression involving variables as well as constants. The standard comparison operators <,>,=,!=,<=,>= are supported for both numbers as well as strings. In addition, the boolean operators AND, OR and NOT can also be used. The predicate is evaluated for each set of values of the variables in the query that the pattern matcher produces. If the predicate evaluates is true, that set is sent to the next operator in the query; otherwise it is rejected.

4 The CONSTRUCT clause

This clause is the final step in the execution of any query. The construct clause consists of a template (which is very similar to a pattern) that specifies how the result is to be formatted. Although XML-QL does not require the result to be a valid XML document, the parser we use enforces this restriction. The construct clause usually involves both text as well as variables. The variables may be bound to strings, numbers, individual element nodes in the document or sets of element nodes. As each tuple is received by the CONSTRUCT operator, it builds an in-memory representation on the XML document. The document is formatted to make it easily readable and is finally written out to disk.

Usually the result template has static text as the names of the various tags and variables as the content enclosed by tags. However, XML also supports tag variables: these allow the names of the tags to be variables. Our implementation has limited support for tag variables. We allow the tag name to be a variable, but we do not allow the tag name to be an expression involving both text and variables.

4.1 Skolem Functions

Skolem functions are analogous to the *group by* clause in SQL. Skolem functions allow grouping of multiple related "tuples" under the same tag. However, due the flexible nature of XML, the functions can sometimes lead to inconsistencies. Our implementation checks for these inconsistencies as it is producing the result and ignores the skolem functions when they are inconsistent. As the results of the query are received by the construct operator, in addition to constructing the in-memory XML representation, it also constructs a hash table of node ID's that involve skolem functions. Whenever a tag with a skolem function is encountered, this hash table is searched to find the correct nodes.

Skolem functions make it impossible to start writing out the result untill the entire document has been processed. This can be a major problem when a single query is distributed across multiple servers.

5 Architecture

The QE follows a client-server architecture. The server waits on a particular port for requests from clients. The clients and servers communicate using sockets. The moment a request comes to a server, it spawns a separate thread to service that request. A single server can execute multiple queries concurrently and is always available to accept more requests if there are sufficient resources on the machine it is running on.

6 Distributed Operation

6. 1 Why Distributed Operation?

6.1.1 Performance

Queries like searching for the cheapest car of a given specification on the internet take quite a bit of time to execute becuase of the sheer volume of data that needs to be scanned for finding the result. This might increase the response time to the user beyond acceptable limits. Since data on internet is distributed across a huge number of servers, this distribution could be exploited to achieve parallelism in query execution and reduce the response time to the user. One can expect a reasonable speedup because of the following reasons:

- Each machine has limited computational resources. Distributing the query makes more computational power available for the execution of the query
- A single server has limited bandwidth. This bandwidth could become a bottleneck if the query involves a large number of big documents. If the query is running on different servers on different parts of the internet, the cumulative bandwidth of the servers can be utilized

Superlinear Speedup

The above reasons can lead to linear speedup at best. Another reality about the internet that can make the super linear speedup possible is the fact that internet is not symmetrically connected. There are clusters of high connectivity connected with each other with relatively low bandwidth available between the clusters. For example if most of the data one is interested in is in London and Berlin. One can run the query on one server in Madison or ship part of the query to another server in Chicago, gaining a speedup of two (if lucky). If the subqueries are shipped to a server in London and another in Berlin and just ship the results over the trans-atlantic line, a speedup of more than two is very likely.

6.1.2 Economic Factors

A large number of internet businesses make money out of the data they own. They either charge money on per query basis (micro money) or through advertising. Such businesses will never part with their whole database stored in their XML files. Very likely, they would run a query engine on their machine and make it available to users to send requests to. They would either charge money for each query or embed advertising in the result XML documents. If a user wants a search on all sites that keep the old car sales data, the only way a query engine can execute this query is by distributing sub queries to each of these sites

6.1.3 Technical Factors

It might be impossible for a query engine to fetch a document (say www.amazon.com/book_data.xml) because Amazon doesn't have any such XML document. They might be just using this name to represent a table book_data in their RDBMS. They, however, may make available a frontend that take XML queries on www.amazon.com/book_data.xml and translate it into a SQL query on the table book_data and execute it on their RDBMS and then package the result table in an XML format. As in the previos case, the queries on www.amazon.com/book_data.xml can be executed only by Amazon query engine and none else.

6.2 Cost Model

We used a simple cost model to decide which sub-queries to ship. Each query engine has a static table called location table. For each XML document, it stores the hostname of the server which is best suited for executing queries on that document. eg an entry looks like www.amazon.com/book_data.xml engine.amazon.com. This table does not store the information about all the XML documents but only those which are really huge or which cannot be fetched because of economic or technical factors. According to our cost model, shipping XML documents listed in the location table is expensive and shipping other documents doesn't cost anything. Our model also ignores the cost of shipping intermediate results and the overhead cost of starting up a large number of servers.

6.3 How a Query is Distributed

Since the servers are symmetric the server that first receives the query acts as a query coordinator. The client server architecture allows the query coordinator to act like a client and send subqueries to the other servers. The protocol is that whenever a server receives a query, it does not send the result of the query directly but returns a URL where it is going to place the results. The coordinator parses the query and looks at the names of XML documents that are needed in this query.

Case 1. If none of the needed documents are in the location table, this server goes ahead and fetches these documents and executes the query itself.

Case 2. If there are some documents in the location table and this itself is the server for those documents then it behaves like in case 1.

Case 3. If there are some documents in the location table and the preferred server for those documents is not the coordinating server then the coordinator pushes the pattern matching for each of the documents to their respective servers.

The coordinator does not push the CONSTRUCT clause to any of the servers. All the final data comes back to the coordinator which then assembles the resulting XML doucument. The coordinator does not receive any intermediate results--it only receives the final results. The intermediate results are directly transferred among servers.

Due to compatibility reasons, servers communicate among each other only using the standard XML and XML-QL and all documents, including intermediate results, are accessed only through URL's. The intermediate results are recast into XML format and shipped. The receiving server parses the document and continues its part of the query.

Our query engine is able to handle the cases when multiple documents in a single inClause belong to different hosts and when documents of different inClauses belong to different hosts.

7 Conclusions

In this paper, we describe the implementation of a distributed query engine for XML-QL. It includes a brief description of the features implemented. We talked about the motivation behind implementing a distributed query engine and our cost model. The process of distributing sub-queries and deciding which sub-query to send where was breifly described.

8 References

[1] Tim Bray (Textuality and Netscape) Jean Paoli (Microsoft) C. M. Sperberg-McQueen (University of Illinois at Chicago), Extensible Markup Language (XML) 1.0.

W3C Recommentdation.

[2] Alin Deutsch (University of Pennsylvania), Mary Fernandez (AT&T Labs), Daniela Florescu (INRIA), Alon Levy (University of Washington), Dan Suciu (AT&T Labs), XML-QL. QL'98.

[3] Document Object Model (DOM) Level 1 Specification, Version 1.0 W3C Recommendation, 1 October, 1998

[4] Jon Bosak. XML, Java, and the Future of the Web. Mar 1997.

[5] Sun Microsystems. Java Project X: Java Services for XML Technology.

[6] IBM Alphaworks. XML Parser for Java.