August 1999

# Extraction of Web Information Using W4F Wrapper Factory and XML-QL Query Language

Deepali Bhandari
*University of Pennsylvania*



University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-99-20.

# Extraction of Web Information Using W4F Wrapper Factory and XML-QL Query Language


## Abstract

In many ways, the Web has become the largest knowledge base known to us. The problem facing the user now is not that the information he seeks is not available, but that it is not easy for him to extract exactly what he needs from what is available. It is also becoming clear that a top down approach of gathering all the information, and structuring it will not work, except in some special cases. Indeed, most of the information is present in HTML documents structured only for visual content. Instead, new tools are being developed that attack this problem from a different angle.

XML is a language that allows the publisher of the data to structure it using markup tags. These mark-up tags clarify not only the visual structure of the document, but also the semantic structure. Additionally, one can make use of a query language XML-QL to query XML pages for information, and to merge information from disparate XML sources. However, most of the content of the web is published in HTML. The W4F system allows us to construct wrappers that retrieve web pages, extract desired information using the HTML structure and regular expression search and map it automatically to XML with its XML-Gateway feature.

In this thesis, we investigate the W4F/XML-QL paradigm to query the web. Two examples are presented. The first is the Internet Movie Database, and we query it with the idea of understanding the power of these systems. The second is the NCBI BLAST server, which is a suite of programs for biomolecular sequence analysis. We demonstrate that there are real life instances where this paradigm promises to be extremely useful.



## Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-99-20.

EXTRACTION OF WEB INFORMATION
USING W4F WRAPPER FACTORY AND XML-QL QUERY LANGUAGE

Deepali Bhandari
Philadelphia, Pennsylvania
(August, 1999)

## Abstract

In many ways, the Web has become the largest knowledge base known to us. The problem facing the user now is not that the information he seeks is not available, but that it is not easy for him to extract exactly what he needs from what is available. It is also becoming clear that a top down approach of gathering all the information, and structuring it will not work, except in some special cases. Indeed, most of the information is present in HTML documents structured only for visual content. Instead, new tools are being developed that attack this problem from a different angle.

XML is a language that allows the publisher of the data to structure it using markup tags. These mark-up tags clarify not only the visual structure of the document, but also the semantic structure. Additionally, one can make use of a query language XML-QL to query XML pages for information, and to merge information from disparate XML sources. However, most of the content of the web is published in HTML. The W4F system allows us to construct wrappers that retrieve web pages, extract desired information using the HTML structure and regular expression search and map it automatically to XML with its XML-Gateway feature.

In this thesis, we investigate the W4F/XML-QL paradigm to query the web. Two examples are presented. The first is the Internet Movie Database, and we query it with the idea of understanding the power of these systems. The second is the NCBI BLAST server, which is a suite of programs for biomolecular sequence analysis. We demonstrate that there are real life instances where this paradigm promises to be extremely useful.

## Acknowledgements

First, I would like to thank my advisor, Dr. Val Tannen for always being generous with ideas, and for his constant encouragement. Arnaud Sahuguet and Val helped me define my thesis topic, and made numerous invaluable suggestions along the way. Arnaud, and Alin Deutsch were always available to answer questions on W4F, and XML-QL, and also helped set up and debug my system. I am also grateful to Professors Susan Davidson, and Peter Buneman for introducing me to database systems, and to Hartmut Liefke for helping with XML. Finally, I would like to thank my parents, and my husband for their support.

# Table of Contents

**1. INTRODUCTION ..... 1**

1.1 QUERYING THE WEB ..... 1
1.2 ADDING STRUCTURE TO DATA ON THE WEB ..... 2
*1.2.1 Semi-structured data and XML ..... 2*
*1.2.2 Example: ..... 3*
1.3 XML-QL ..... 4
1.4 W4F: WYSIWYG WEB WRAPPER FACTORY ..... 5
1.5 A ROAD MAP ..... 5

**2. XMLIZING AND QUERYING THE INTERNET MOVIE DATABASE ..... 6**

2.1 INTRODUCTION ..... 6
2.2 IMDB AS A TEST BED ..... 6
*2.2.1 The XML-QL alternative ..... 6*
2.3 BUILDING AN HTML WRAPPER USING W4F ..... 9
2.4 QUERIES USING XML-QL ..... 9
2.5 CONCLUSION ..... 13

**3. XMLIZING AND QUERYING BLAST OUTPUT ..... 14**

3.1 INTRODUCTION TO BLAST: ..... 14
3.2 XML VS. OTHER SCHEMAS ..... 17
3.3 IMPLEMENTATION OF THE BLAST EXAMPLE: ..... 17
3.4 XMLIZING BLAST OUTPUT ..... 17
3.5 IMPLEMENTING BLAST QUERIES IN XML-QL. ..... 19

**4. TYING IT ALL TOGETHER: THE IMPLEMENTATION ..... 21**

**5. REFERENCES ..... 23**

**6. APPENDIX 1 : W4F WRAPPER FOR IMDB ..... 24**

**7. APPENDIX 2 : W4F WRAPPER FOR BLAST ..... 27**

# 1. Introduction

In many ways, the Web has become the largest knowledge base known to us. Information on any topic conceivable is available, literally at our fingertips. Surely, a key reason for the success of the Web as a data resource is its heterogeneity. With relative ease, any individual or institution is able to publish information for the entire world to access at the click of a button. And everyone has embraced this idea. Individuals maintain home pages with personal data. Institutions use the web to make information accessible to the public. Many journals now publish their articles electronically. Airlines, train companies provide schedules and fares on-line. Organizations use the web as a bulletin board to exchange ideas and information, and to coordinate their efforts. Universities can list their schools, course offerings and student transcripts. Corporations of course see in the web, the potential for direct sales, and customized advertisements tailored to the individual. For the individual, the web has become a source of all the information that the rest of the world is putting up.

## 1.1 Querying the Web

The problem facing the user now is not whether the desired information is available, but rather, how to effectively extract what is needed from what is available. The *de-facto* standard for publishing information on the web is HTML. A few features make HTML the language of choice. Data (typically text, but also pictures, and sound) is enclosed by mark up tags which instruct a *browser* on one of the following:

1. how to display the enclosed text
2. which application to use with the enclosed data.
3. add a hyper-link to another document on the web.

Certainly, hyper-links are a powerful query/browsing mechanism, and it allows a user to quickly find his way to the desired information by traversing appropriate links. The user traverses the link by clicking on a button or highlighted text. In a database view of the world, the highlighted text is a keyword, or a query, so as to speak. Clicking on the hyper-link is akin to launching the query, and the new page served up is the answer to that query. Even this limited query mechanism has been quite successful, and has given rise to the popular sub-culture of ‘web surfing’.

There are a few other, more powerful query mechanisms available to the user. First, organizations that publish information on the web often store it locally using a powerful relational or object oriented database. The information is then ‘published’ by either dumping it in a series of hyper-linked documents, or by providing certain ‘canned’ queries that a user can customize by entering values in a form, and then pressing a button to run the query. For structured data, this is useful but still quite limited. First, the user does not

usually have the full querying power of the relational database. For example, Amazon.com possibly maintains all its music CD records in a database. Likewise, IMDB maintains a database of movies. Consider the query:

**Query:** *Find all the movies in which Dustin Hoffman acted for which a Soundtrack album is available*

This is a simple join across two tables, but there is no way to actually accomplish it on the Web, other than to do it manually. Second, this approach is quite useless for unstructured data that cannot be stored in tables. For example, suppose one wants all the newspaper articles on the effectiveness of anti-oxidants in preventing cancer. Even though the newspapers are all on-line, and maintain archives, there is no easy way to retrieve this.

For unstructured data, the best querying mechanism currently available is *key-word search.* Sophisticated web-crawlers regularly scan the web, retrieving and indexing words. A search for a keyword results in a dump of all pages containing that word. As the web grows the number of pages containing a key-word is also growing to a point where searching them manually is no longer practical.

## 1.2 Adding Structure to data on the Web

Right now, almost all the sources of information on the Web are disparate in the sense of the way their data schema is defined. For Online Finance risk management analysis to work effectively, the decision support systems that they rely need to get information in a unified way. Vast resources available in the form of digital libraries need data integration.

For unstructured data, two things need to be done to make querying more effective. First, we must provide a mechanism for the publisher of the data to provide some structure to the information being presented, and second, we need to provide tools to be able to parse and query the semi-structured data. A popular way to add structure to data is by publishing in XML(e.g a Chemical Markup Language for exchanging data about molecules , Open Financial Exchange for exchanging financial data between banks , MathML etc.)

The simplicity, extensibilty and self-describing aspect of XML, will not only facilitate exchange data over the Web but also solve the problems of data unifications required to exploit the maximum potential of Web applications. In the following section we put forward some of the essentials of XML and illustrate them with an example

### 1.2.1 Semi-structured data and XML

XML or eXtensible Markup language was proposed by the SGML working group of the W3 consortium. Recall that HTML was developed to describe how a web browser should arrange text, images on a page. It has a fixed tag set and does not convey structure of the document. XML was developed to forgo all of the above mentioned shortcomings of HTML.

A complete description of XML is beyond the scope of this document (See[8] for a detailed description). From our perspective, the key idea in XML is that of mark-up tags that clarify not only the visual (as in HTML) but also the semantic structure of a document. Along with the tags are a set of rules that impose structure on the tags. Unlike previous languages like SGML however, the DTD can often be inferred from the document, and does not have to be explicitly supplied. One can think of an XML document has having a tree like structure with root node and edges. The leaf nodes correspond to the text portion of the data. Edges correspond to the markup tags or *elements*, while interior nodes correspond to *attributes* of the elements.

This simple idea is actually quite powerful, as it allows the publisher of the data to add structure in a completely flexible manner. Perhaps the best way to understand the power of XML is through an example.

### 1.2.2 Example:

Consider the following:
*Imagine that you are an on-line seller of computers and accessories. Not only do you sell your computers on-line, you also buy them on-line from different manufacturers. Note that you must compete not only with other online stores, but also with the Computer manufacturers themselves. You compete primarily by making your shop a one-stop shopping experience, where any customer can find machines and accessories that fit their needs and budgets.*

In order to stay competitive, you must have the following capabilities, each of which is best served by XML, and an associated query language.

1. **Data Extraction:** Visit each of the computer manufacturer's site, down-load their product descriptions and prices. Often, the information might be available only as unstructured, or semi-structured text (ex., expert reviews of a line of computers). Structured databases are not very good at storing such data.
2. **Data Integration/ Data Restructuring:** The information collected from each manufacturer's site must now be integrated. For example, a potential buyer might be interested in seeing details of all ink-jet printers with a print resolution of 360 X 180 dpi. In order to allow such queries, all the data must first be collected from all the manufacturers, and integrated. In a structured database, this would entail the substantial overhead of building and maintaining local tables. With XML, integration of information across different web sources would be fairly easy. Different buyers might want to see different views of the data. Continuing with the previous example, some buyers might want to see the price along with the print quality, others might want to see print speed, and still others might want to see online reviews of the products. Further, the same user might need to reconfigure the data to get the best view for his needs. With XML, it is possible for the user to down-load a large XML document as the result of the first query, and make subsequent queries to that XML document directly on the client machine. This

takes load off the network, and your computer. To take another example, a customer in India might want to see all products priced in Rupees instead of dollars. XML would allow the customer to download the data, and restructure it locally using XML-QL, without the need for extensive customized code.

3. **Schema browsing:** Standard databases differentiate between data and the schema for storing that data, and it is not clear just by looking at the data, what the schema is. On the other hand, in XML, the schema is given by the structure of the tags surrounding the data, and can be inferred computationally from the data. This allows a user to 'query on the fly'.

### 1.3 XML-QL

The previous example explains XML will change the way we think about the Web. However, in order to exploit the power of XML, we need a query language. The query language should address questions such as as :

- *What techniques and tools should exist for extracting data from large XML documents?*
- *How will XML data translate between different DTD's?*
- *How will XML data from multiple sources be integrated?*

In this thesis, we adopt XML-QL ([3]) as the query language for XML documents. XML-QL is very influenced by recent research and implementations of Query languages for semistructured data (See [9]). XML-QL queries make implicit assumptions about how the XML data is modeled, which is an XML Graph. Only leaf nodes in the XML Graph contain values. This particular model ignores element order for the sake of simplicity and order independent optimizations in the query processor. Next, XML-QL uses element patterns to match data in an XML document. A query in XML-QL is a regular expression over elements, and variables (represented by strings preceded by $). It looks for the r.e. pattern in the XML graph, and for each match it binds the variables in the regular expression to the text data. This a SELECT query is relaively straightforward to implement. "Joins" can be expressed by matching two or more elements that contain the same value. This helps in producing an integrated view of different XML sources. Other, more advanced features include

1. The use of tag variables which allow a variable to be bound to any edge (element) in an XML Graph.
2. Regular path expressions which can specify element paths of arbitrary depth through an XML Graph.

We will defer further discussion on XML-QL to the next section, where its power will be illustrated by examples. For a formal description, see [3].

### 1.4 W4F: WysiWyg Web Wrapper Factory

W4F is a toolkit that allows the fast generation of Web wrappers [4,5]. Wrapper generation consists of *Retrieval* of an HTML page via GET or POST methods, followed by construction of HTML parse tree according to the HTML hierarchy. Information can then be *Extracted* declaratively using a set of rules applied on the parse tree. A nested string list (NSL) data structure is used as the datatype to represent extracted information internally.

Additionally, in places where HTML tags cannot be exploited to obtain information , W4F has incorporated the usage of regular expression patterns. Finally, NSL's are converted into Java base types, and for more complex structures, the user can define his own mappings.

The crux of building a wrapper lies in the definition of the extraction rules. In order to extract information, the user can trace the path along the tree defined by nested HTML tags, and finally match a regular expression the text at the leaf of the tree. W4F facilitates the identification of this path by providing a WysiWig interface ([4]) in which the HTML document is annotated in a way that the user can get the HTML path by clicking the appropriate text.

### 1.5 A Road Map

We have not been comprehensive in our description of XML-QL, and W4F. In this thesis, we focus on these systems from a users' perspective. In particular, we explore the use of W4F and XML-QL as tools for querying the Web. Our methodology is as follows: Use W4F to retrieve an HTML page, extract information from it, and to XMLize the extracted information. Next, use XML-QL to query the XML files.

We focus on two Web resources, the Internet Movie Database (http://www.imdb.com), and the NCBI BLAST server. In Chapters 2, and 3, we describe our work on XMLizing and querying IMDB, and BLAST respectively. The work in chapter 2 is motivated by a desire to understand the power of W4F, and XML-QL as a tool for querying the web. In contrast, Chapter 3 focuses on some real-life problems for which W4F, and XML-QL is perhaps the best solution. Finally, in chapter 4 we provide details on how the entire system is tied together.

# 2. XMLizing and querying the Internet Movie Database

## 2.1 Introduction

The Internet movie database provides useful and up to date information about movies freely. It currently covers over 170,479 titles with over 2,462,516 filmography entries and is being updated continually. The front end is an HTML form that allows users to search the database for keywords such as actors. The site is provided by Amazon.com, and a movie entry is hyperlinked to entries in Amazon on the corresponding books and Soundtracks.

## 2.2 IMDB as a test bed

IMDB provides an interesting case study to get familiar with the functionality of W4F, and XML-QL. We jump straight to an example:

**Query:** *Find all movie directors who have directed both John Travolta and Bruce Willis.*
The query mechanism at IMDB is too limited to allow this directly. One possible manual solution is as follows:

1. Use keyword search to retrieve all the movies of John Travolta, and then, movies of Bruce Willis.
2. Scan the movie descriptions to get two lists of all the directors for the two actors.
3. Find the intersection of the two lists.

If we had to repeat this many times for other actor pairs, we would need to do this computationally by downloading the various pages, and parsing them to get the directors. The parser should be able to take advantage of the inherent structure of the HTML document, as well as keywords (such as *director*) in the text. Next, we would either have to write code to find the intersection of the two lists, or load the two lists into a table to get the intersection. One can appreciate that both solutions are rather cumbersome for someone who is just 'surfing' the net for information, even if that person is computer savvy.

### 2.2.1 The XML-QL alternative

Suppose next that the IMDB pages were published not in HTML, but in XML. Figure 1 gives a possible DTD for the page corresponding to the movies of an actor.

```
<!Doctype Actor [
<!Element Movie(Name, Dir, Categories, Soundtrack?)>
<!Element Name PCDATA>
<!Element Dir PCDATA>
<!Element Categories(genre*)>
<!Element Soundtrack (PCDATA, price)>
<!Element price PCDATA>
<!Element genre PCDATA>
]>
```

**Figure 1: DTD for IMDB example**

```
<?xml version="1.0"?>
<Root>
  <Movie><name>Cosm (1999)</>
         <dir>Jan de Bont</>
         <Categories>
            <genre>Sci-Fi</>
         </>
  </>
  <Movie><name>Joan of Arc (1999/I)</>
        <dir>Luc Besson</>
        <Categories>
           <genre>Drama</>
           <genre>War</>
        </>
  </>
                :
                :
  <Movie><name>Wag the Dog (1997)</>
         <dir>Barry Levinson</>
         <Categories>
            <genre>Comedy</>
         </>
         <Soundtrack>
            "THANK HEAVEN FOR LITTLE GIRLS" Written by Alan Jay Lerner, Frederick Loewe
Performed by Maurice Chevalier
            <price> 11.67 </>
         </>
   </>
                      :
                      :
```

**Figure 2: XML document for Dustin Hoffman's movies**

An XML document that conforms to this DTD is shown in Figure 2, and describes the movies of Dustin Hoffman. Each movie is enclosed within the <movie> tag, and contains Elements like Name, Dir, Categories, and Soundtrack. The ? symbol in front of Soundtrack in the DTD implies 0 or 1 occurrence of this tag. As seen in Figure 2, "Wag the Dog" has a Soundtrack tag, but "Joan of Arc" does not. The Categories tag refers to 0 or more occurrences of genre (represented by a "*"). "Joan of Arc" in Figure 2 can be categorized as being in the genre of Drama, or the genre of War.

So far, we haven't said anything about querying. Figure 3 gives part of the query that gets all the directors who have directed both Dustin Hoffman, and Bruce Willis. We assume that all the movies of Bruce Willis are in bruce.xml. Correspondingly hoffmanDir.xml contains a list of all the directors who directed Dustin Hoffman in a movie. The query in figure 3 looks at each director that has directed Bruce Willis. Each director is captured in $d by matching the `<Dir> $d </>` tag in the document. Next, the condition

`<XML.Movie.dir>$d</> in "file:hoffmanDir.xml"`

ensures that the same director has directed Hoffman. Each such name $d added to the list being output. We avoid getting duplicate entries in our final list by the use of skolem functions. Each director is given a unique Object identifier OID, and each OID is included only once in the final list. Note that this query is analogous to a Natural Join Query for a Relational database.

```
function query () {

 construct
         <dirlist ID=F($d)>
                <dir> $d </>
         </>
 where
         <Root.Movie>$m</> in
            "file:bruce.xml",
         <dir>$d</> in $m,
         <XML.Movie.dir>$d</> in
            "file:hoffmanDir.xml"
}
```

**Figure 3: XML-QL query for directors who have directed Hoffman and Willis.**

In this discussion, we have assumed the availability of XML documents containing all the movies that Bruce Willis acted in, as well as a document containing all the directors who have directed Dustin Hoffman. We generated these files automatically using the W4F system.

### 2.3 Building an HTML Wrapper using W4F

In Appendix 1, we describe a W4F script to extract information from IMDB pages. The task is sub-divided into three parts, RETRIEVAL, EXTRACTION, and JAVACODE. We consider the Retrieval rules first.

For the IMDB example, all our queries start by typing the name of an actor, and getting all the movies in which that actor has performed. The *get*() method is called with the name of the actor and it does the GET query "`http://us.imdb.com/Name?$act_nm$`" to get the HTML page corresponding to all the movies of that actor. In IMDB, each movie name is hyper-linked to another page which contains all the information for that movie. The pages for each of these movies are extracted iteratively using the *get_next_url* () method. For obtaining the Soundtrack information for each movie, first the Sales page related to the movies is fetched by calling the *getSales* () method whose input is the movie title. Next, the URL for the Soundtrack page is extracted into the variable "url_ST". This is then passed on to "getSTurl" method, which retrieves the page with Soundtrack information such as title, price, availability etc.

Next we discuss Extraction. For every movie page corresponding to a movie that the actor has performed in, we extract the movie title, director and genre are extracted using the definitions shown in Appendix 1. Likewise, the information corresponding to the Soundtrack, such as title, price is extracted from the Soundtrack page. To give an example of the extraction consider the following:

```
ST_price = html.body[0].table[1].tr[0].td[2].font[2].b[1].font[0].txt;
```

The Soundtrack price is extracted from the Soundtrack page by considering the structure of the HTML document, and following the nested (and ordered ) tags. Thus, here, we look at the second <table> tag in the body of the document. We consider the third column (td[2]) of the first row (tr[0]) of that table, and so on till we come to the desired text. At first glance, the task of extracting HTML paths for the Extraction rules might seem tedious. However, these paths were directly obtained using a WysiWyg interface provided by W4F ([4]).

Once the information is extracted into the appropriate structures, it is printed to a temporary file after being marked with XML tags. The XML file seen in Figure 2 was created automatically in this fashion.

### 2.4 Queries using XML-QL

In the remaining part of this section, we use the W4F/XML-QL paradigm to query HTML pages from IMDB. The queries are not perhaps exciting in themselves, but *are* representative of the way one might want to use the web. We implemented these queries with the idea of understanding the strengths and limitations of these two systems. For example, consider the following query:

**Query:** *Find all the soundtracks to movies in which Dustin Hoffman acted, and which cost less than $20.00.*

While XML-QL can allow fairly sophisticated queries, the current implementation does not allow comparison operators and therefore, this query.

The following queries were also implemented:

**Query**: Return all movies of an actor grouped by genre.

```
function query () {
//  group by genre, using Skolem functions
 {
  construct <genre ID=F($g)>
                <type>$g</>
                <name>$n</>
            </>
  where  <Root.Movie>$m</Movie> in "file:actor.xml",
          <name>$n</name> in $m,
          <Categories.genre>$g</genre> in $m
 }
}
```

**Figure 4: Group all movies of an actor by genre**

**Implementation** :

See Figure 3. XML-QL uses element patterns to match data in the document. Here $m variable will grab everything delimited by the Tags <Movie>,</Movie>. For every such movie the name is captured in the $n variable and the other condition is that $n should be in $m. Further, $g grabs the genre for the movie. Typically for a movie there are more than one genres possible. This time, we make use of the object identifiers (OID's) and Skolem functions to group movies by their genre. F is a Skolem function and it generates a new identifier for distinct values of $g. If, at a later time, $g is bound to same value (meaning there exists another movie with same genre), then the query will not construct another genre but append name of the movie to the genre element.

**Query:** *All movies grouped by directors ( Similar to Genre)*

**Query:** *Movies released in 1996.*

**Implementation:** See Figure 4.

```
function query () {
   construct  <Movie>$n</>
   where
    <Root.Movie>$m</> in "file:actor.xml",
    <name>".*(1996).*"</> in $m,
    <name>$n</name> in $m
}
```

**Figure 5: Movies released in 1996**

This query is otherwise straightforward, except that it has made use of a regular expression inside the name tag. The symbol '.' matches any character in the text and '*' means zero or more characters. So if the string "(1996)" appears anywhere inside the name tag, the name of that movie would be returned by the query. Note that this query is similar to a Selection query in a RDB. However, the Selection is not done on a predefined field, but on unstructured text within a Tag. Thus this query combines features of Selection as well as keyword search.

Regular expressions, something missing in conventional query languages, are also useful for Schema browsing. For example, suppose the user knows that the genre of the movie is marked by the <genre> tag, but doesn't know where this tag appears in the document. Figure 5 describes a query he can run to get all the different genres in which an actor has performed. Note the regular expression which looks for the <genre> tag at any depth from the root of the document. However, this query is quite computationally inefficient.

```
function query () {
construct  <genre>$g</>
where <Root._*.genre>$g</>
in "file: actor.xml"
}
```

**Figure 6: All the different genres in which the actor has performed**

**Query:** *Movies of the actor for which soundtracks are available (The title and price of the Soundtrack are also returned).*

**Query:** *Movies of the actor which have the genre Drama AND Thriller.*

**Query:** *Movies of the actor which have genre Drama OR Romance.*

**Implementation:** See Figure 6. The query is simplified if we first create an auxiliary file "genreout.xml" containing all movies grouped by genre. Thus, in this file, there is a tag <genre> containing the genre name, as well as the names of all movies in that genre. Each movie is marked by a <name> tag. We first construct a list of movie names that are in the genre Romance. Each movie is assigned a unique OID using Skolem functions. Next we construct the list of movie names in the genre Drama. The use of skolem functions

guarantees that only movies that are not in Romance will be assigned a new OID. Therefore, in the end, we get a non-redundant set of movies.

```
function query () {
   {
        construct <movie ID=M($n)>
                            <name>$n</>
                          </>
         where   <XML.genre>$g</> in "file:genreout.xml"
                     <name>$n</> in $g,
                    <type>"Romance"</> in $g
     }
     {
         construct <movie ID=M($n)>
                          <name>$n</>
                        </>
         where    <XML.genre>$g</> in "file:genreout.xml",
                     <name>$n</> in $g,
                     <type>"Drama"</> in $g
       }
}
```

**Figure 7: Movies with genre Romance OR Drama**.

**Query:** *Directors who have directed more than one movie.*

**Implementation:** This is not as easy to obtain as the corresponding count by operator in SQL. We implemented this by first grouping all the movies of the actor by director name, and storing the result in the auxiliary file dirout.xml. In dirout.xml, each group is surrounded by a <movie> tag. Within the <movie> tag, there is one <dir> tag for the director, and one or more <name> tags for the movies directed by that director. We know that a director has directed the actor in more than one movie, if we can find a movie $m, with director $d, and at two movie names $n1, and $n2, such that the text($n1) is not the same as text($n2) directors. This is implemented in Figure 8. Thus Aggregate operators are possible but cumbersome in XML-QL.

```
function query () {
  construct
         <dir>$d</>
   where   <XML.movie>$m</> in "file:dirout.xml"
               <name>$n1</> in $m,
               <name>$n2</> in $m,
               <dir>$d</> in $m,
               text($n1)!=text($n2)
}
```

**Figure 8: Get all directors who directed more than one movie of an actor.**

### 2.5 Conclusion

The goal of this section was to understand the power of W4F/XML-QL as a paradigm for querying on the web, using a standard example. We conclude that XML-QL is quite flexible in its query mechanism. Many of our queries on IMDB were motivated by typical queries on a relational database, like Select, Project, Join an so on. In particular, the use of regular expressions in the tag structure allow us to query without a complete description of the schema.

However, it is not as convenient for some fundamental operations like Not in (Set difference), arithmetic comparisons, and aggregate operators on Group By. It is possible that subsequent versions will address these difficulties.

There are a few other minor problems.

1. If the XML page has a non alphanumeric character, like accents in the German language, the XML-QL parser raises an error and dies.
2. In practice we often limited the program to only retrieve the first 10 movies of the actor, because the full query took so much time that Netscape timed out.

The W4F system is invaluable in XMLizing existing HTML documents. It offers a clean mechanism to retrieve static and dynamically generated HTML pages from the Web, and its Nested String List structure allows us to exploit the HTML tags and patterns in the text to extract the information we need. One observation we would like to make is that both XML-QL and W4F use markup tags to get a tree-like structure from an HTML or XML document. Perhaps the underlying representation for both should be unified into a single structure. This might make the W4F/XML-QL paradigm even stronger.

# 3. XMLIZING AND QUERYING BLAST OUTPUT

## 3.1 INTRODUCTION TO BLAST:

*BLAST* is a collection of programs designed to search a database for strings similar to a query string ([1]). In the context of Biological databases, the query and database are either DNA or amino acid sequences. For our purpose, we can think of DNA as a string over a 4 letter alphabet, and a protein or amino acid sequence as a string over a 20 letter alphabet. Similarity between two strings is measured as follows: the two strings are lined up against each other (*an alignment*). Each pair of symbols is scored by looking up a standard *Scoring Matrix*. The score of the alignment is the sum of scores of each aligned pair. The database sequences are sorted by decreasing scores of their alignment with the query sequences, and all sequences that score over a user defined threshold are returned, along with the alignment.

We used the NCBI BLAST server ([1]) for our experiment. The key portions of a typical BLAST output are shown in a BLASTP example below (See Figure 9.). A chicken gene X protein (232 amino acids) is compared to the SwissProt protein database ([2]). In this case, one of the SwissProt sequences highly similar to chicken gene X is the human plasminogen activator (415 amino acids). The first line in the output identifies the Query sequence. From that point, the sequence is always referred to as QUERY. Next, the database being scanned is identified. Finally, a header line for each database sequence that 'hits' the query sequence is displayed. In this case, we see the header

```
>sp|P05120|PAI2_HUMAN PLASMINOGEN ACTIVATOR INHIBITOR ....
```

From this point onward, the database sequence is referred to as the `SBJCT`. After the header line for the SBJCT, the statistics of the alignment are shown, followed by the alignment itself. Note that the sequences are not aligned across their entire length. Instead, only the high scoring locally aligned regions are displayed. In this case, two sequences match in more or less contiguous regions. Chicken [1..89] matches human [180..268], Chicken [90..151] against human [272..333], and chicken [155..232] against human [338..415]. Consider the HSP matching chicken [1..89] against human [180..268]. The alignment is 89 amino acids long. 38 of the 89 pairs are *identical* in the alignment. An additional 12 are not identical but sufficiently similar physiochemically, making a total of 50 *positives*. The *score* of this alignment is 176. The probability that a randomly chosen substring from a SwissProt sized database (`13,464,008` letters) gets the same score is estimated to be 1.8e-65, implying that this hit is statistically significant. The statistics of the

alignment are followed by the alignment itself, which is easy to understand, and shows the locations of the identities, and the positives.

```
Query =  pir|A01243|DXCH  232 Gene X protein - Chicken (fragment)
                   (232 letters)

 Database:  SWISS-PROT Release 29.0
          38,303 sequences; 13,464,008 total letters.

 :
 :

>sp|P05120|PAI2_HUMAN PLASMINOGEN ACTIVATOR INHIBITOR-2, PLACENTAL (PAI-2)
                        (MONOCYTE ARG- SERPIN).
                        Length = 415

 Score = 176 (80.2    bits), Expect = 1.8e-65, Sum P(4) = 1.8e-65
 Identities = 38/89 (42%), Positives = 50/89 (56%)

Query:     1 QIKDLLVSSSTDLDTTLVLVNAIYFKGMWKTAFNAEDTREMPFHVTKQESKPVQMMCMNN 60
             +I +LL   S D DT +VLVNA+YFKG WKT F  +     PF V   +  PVQMM +
Sbjct:   180 KIPNLLPEGSVDGDTRMVLVNAVYFKGKWKTPFEKKLNGLYPFRVNSAQRTPVQMMYLRE 239

Query:    61 SFNVATLPAEKMKILELPFASGDLSMLVL 89
               N+  +   K +ILELP+A      L+L
Sbjct:   240 KLNIGYIEDLKAQILELPYAGDVSMFLLL 268

 Score = 165 (75.2 bits), Expect = 1.8e-65, Sum P(4) = 1.8e-65
 Identities = 33/78 (42%), Positives = 47/78 (60%)

Query:   155 ANLTGISSAESLKISQAVHGAFMELSEDGIEMAGSTGVIEDIKHSPESEQFRADHPFLFL 214
             AN +G+S    L +S+  H A ++++E+G E A  TG +   +      QF ADHPFLFL
Sbjct:   338 ANFSGMSERNDLFLSEVFHQAMVDVNEEGTEAAAGTGGVMTGRTGHGGPQFVADHPFLFL 397

Query:   215 IKHNPTNTIVYFGRYWSP 232
             I H  T  I++FGR+ SP
Sbjct:   398 IMHKITKCILFFGRFCSP 415

 Score = 144 (65.6 bits), Expect = 1.8e-65, Sum P(4) = 1.8e-65
 Identities = 26/62 (41%), Positives = 41/62 (66%)

Query:    90 LPDEVSDLERIEKTINFEKLTEWTNPNTMEKRRVKVYLPQMKIEEKYNLTSVLMALGMTD 149
             + D  + LE +E  I ++KL +WT+ + M +  V+VY+PQ K+EE Y L S+L ++GM D
Sbjct:   272 IADVSTGLELLESEITYDKLNKWTSKDKMAEDEVEVYIPQFKLEEHYELRSILRSMGMED 331

Query:   150 LF 151
              F
Sbjct:   332 AF 333
```

**Figure 9: Portion of a Blast output (Blastp of chicken Gene X against the SwissProt database).**

A typical BLAST run may find anywhere from zero to a few hundred hits to a query sequence. Biologists are interested in BLASTing their query sequences for many reasons, partially enumerated below:

**Novelty:**

A biologist might be interested in checking if their newly discovered gene is entirely 'new' or if someone else has already discovered it. (S)he could run a BLASTN query against a DNA database, and look only for matches that show 100% identity, and no gaps.

**Function:**

One way to infer the function of an unknown protein is to BLAST against a protein database to get similar sequences which putatively have the same function. So one could want all sequences with a certain score.

**Secondary Function:**

It could be that the biologist knows that a specific region of the query sequence is important for its function. In that case, the biologist is only interested in the database sequences that match this sub-region of the query sequence, and see if the matched portion is annotated as a known protein domain in the database.

**Matches to Expressed Sequence Tags:**

With a novel cDNA sequence, one often does not know if one has the full length clone. By comparing the novel sequence to similar ESTs, and checking whether most of them are 5' or 3', one can get an idea of the true extent of the clone.

**Parameter Sensitivity:**

When the parameters of the Blast search (mismatch penalties etc.) are changed, the search often yields different results. Database sequences that are hit with one set of parameters but not the others are often interesting because they represent a distant non-obvious homology. Thus a biologist might be interested in a querying the database with different parameters, and looking at sequences which are hit only with specific parameters.

**Other Information:**

The scientist viewing the hits often needs to follow other links to get a full picture. For example, if his query matches sequences in the EST database, he might be interested in following the links to see whether the ESTs are human or mouse, and which tissues (brain, whole embryo) they were extracted from.

### 3.2 XML vs. Other Schema

The key element in all of the queries above is that they are all views on a DYNAMICALLY generated data set. The BLAST output is the result of running the program, and is not saved to a database. It is used mostly for browsing. Occasionally, the biologist may want save all or part of it to his local disk. However, the output can sometimes be often hundreds of pages long, and the flexibility of seeing only a portion of the result offers a distinct advantage over looking at the entire output. One could achieve a similar level of flexibility using a relational database. However,

1. As the Blast output is dynamically generated, one would have to load a table each time a new Blast run is performed. On the other hand, the result often become useless after a few days and the corresponding rows must then be deleted.
2. Typically, one has to write a wrapper that will parse the BLAST output into appropriate fields. The wrapper needs to be rewritten each time a new version of BLAST is released. With XML, it could be that the NCBI site will produce an XMLized output eliminating the need for wrappers. With browsers that can display XML documents, the user can browse and query on the fly.
3. The unstructured portion of the data such as the alignments cannot be usefully stored and queried in structured databases. One could use languages such as Perl to search in the unstructured data, but the effort of writing a Perl parser, that both queries the data and displays it properly is beyond an average biologist. In contrast, XML query languages exploit the tagged structure of the document, and make both querying and displaying the result of the query relatively simple.
4. If the Biologist also wants to extract other information by following links, the effort in merging different XML documents is also minimal compared to using Perl or other similar schemes.

### 3.3 Implementation of the BLAST example:

Our implementation has three stages.
1. XMLize the output of the BLAST program.
2. Writing XML-QL queries.
3. Integration of the system using a Web interface.

### 3.4 XMLizing BLAST Output

The desired DTD for the Blast output is fairly straightforward, and given in Figure 8.

```
<!Doctype BLAST[
<!Element Sequence* (EXPECT, SCORE, IDENTITIES?,
GAP, SPECIES?, TXT, QBEGIN, QEND, SBEGIN, SEND,
TISSUE?)>
<!Element  EXPECT PCDATA>
<!Element  SCORE PCDATA>
<!Element  IDENTITIES PCDATA>
<!Element  QBEGIN PCDATA>
<!Element  QEND PCDATA>
<!Element  SBEGIN PCDATA>
<!Element  SEND PCDATA>
<!Element  TISSUE PCDATA>
<!Element  GAP PCDATA>
<!Element  GAP PCDATA>
]>
```

**Figure 10: DTD for a BLAST document**

We need to create XML documents for BLAST outputs that conform to the DTD shown above. Again, we use the W4F system to do this conversion. Appendix 2 describes the W4F code for a BLAST output. In the case of the BLAST example, the output is essentially text, and the extraction is not as easy as one would like.

As it turns out each 'hit', which includes the header for the database sequence, the statistics of the alignment, and the alignment itself, is surrounded by a <PRE> tag. We use this by extracting information from the contents of all <PRE> tags after the first two. As shown in the W4F code, all the statistics of the alignment and the alignment itself are captured by matching regular expressions in the text field.

We illustrate this with the example that extracts alignQE, which is an array of strings representing the Query portion of the alignment. In Figure 8, this would correspond to the two lines.

```
QIKDLLVSSSTDLDTTLVLVNAIYFKGMWKTAFNAEDTREMPFHVTKQESKPVQMMCMNN
SFNVATLPAEKMKILELPFASGDLSMLVL
```

As with all other information, we extract the text from each <PRE> tag after the first two. Then we consider the portion of the text that begins with the string "Query". This text is split by all the intermediate occurrences of Query using the keyword "Que", so that each fragment has one line of the alignment. Finally, we match the text portion of each line after first demarcating the coordinate numbers on either side.

```
alignQE=html.body[0].form[0]->pre[3-].txt,match /(Query.*)/,split /Que/,
                  match /ry\\:\\s*\\d+[\\sa-zA-Z\\-]+([\\d]+)/;
```

Another interesting feature of the W4F implementation for BLAST is that we need to extract species and tissue information from other genbank documents. The BLAST output does not contain information on which species a database sequence comes from, and the tissue library from which that sequence was extracted. In the Retrieval rules, the procedures *getorgspN* (), and *getorgspP* () take as input an accession number that identifies the sequence uniquely, and retrieve the Nucleotide and Protein database pages, respectively that contain the species and tissue information. This information is then extracted from the page.

### 3.5 Implementing BLAST queries in XML-QL.

Most of the functionality described at the end of Section 3.1 has been implemented. In Figure 9, we get all sequences in the database that are 100% identical over the length of the alignment. Figure 10 checks if any 3' Expressed Sequence Tags (ESTs) match the query fragment. The query in Figure 11 checks if the query cDNA sequence is a complete transcript (or just a fragment) by seeing if the similar sequences hit match across their entire length. To explain the query, note that we are capturing the length of the database sequence in a <LENGTH> tag, and the beginning and end of its alignment in the <SBEGIN>, and <SEND> tag respectively. Now, the query sequences matches the database sequence across its entire length only if the <SBEGIN> tag is 1, and the text in the <SEND> and <LENGTH> tags is identical.

In Figure 12, we group all the hits according to species. This is useful because if there are a lot of species that contain the same gene, that could imply that the gene is ancient and probably codes for a protein that is essential to proper cellular function. Finally in Figure 13, we check if there is any hit from a sequence in Embryonic Tissue to check if the query gene is expressed in embryos. As mentioned earlier, the species and tissue information is not reported in the BLAST output, and the XML page conatisn ist because it has merged the BLAST output with information from genbank.

```
function query () {
   construct
            <SEQ>
               $s
            </SEQ>
   where
          <W4F_DOC.LIST.SEQ>$s</SEQ> in
                        "file:blastout.xml",
          <IDENTITIES>100</IDENTITIES> in $s
   }
}
```

**Figure 11: Get all sequences that are 100% identical to the query.**

```
function query ()
{
  construct <SEQ>
             $s
            </SEQ>
  where
      <W4F_DOC.LIST.SEQ> $s </SEQ> in
                    "file:blastout.xml",
      <TXT>".*3'.*"</TXT>in $s

}
```

**Figure 12: Find all similar sequences that are 3' ESTs**

```
function query ()
{
  construct <SEQ>
             $s
            </SEQ>
  where
      <W4F_DOC.LIST.SEQ>$s</SEQ> in
                         "file:blastout.xml",
      <LENGTH>$l</LENGTH> in $s,
      <SBEGIN>1</SBEGIN>in $s,
      <SEND>$x</SEND> in $s,
      text($x)=text($l)
}
```

**Figure 13: Find all database sequences that match the query sequence over their entire length.**

```
function query () {
//  group by species, using Skolem functions
{
  construct
        <LIST ID=F($sp)>
               <SPECIES>$sp</SPECIES>
               <SEQ>$t</>
        </>
  where
        <W4F_DOC.LIST.SEQ>$s</SEQ>in
                    "file:blastout.xml",
        <SPECIES>$sp</SPECIES> in $s,
        <TXT>$t</TXT> in $s
}
```

**Figure 14: Group all hits by their species**

```
function query ()
// Get hits which have tissue type "embryo"
{
  construct
          <SEQ>$s</>
  where
         <W4F_DOC.LIST.SEQ>$s</SEQ>in "file:blastout.xml",
         <TISSUE>"embryo"</TISSUE> in $s
}
```

**Figure 15 Get all cDNA hits from embryonic tissue.**

## 4. Tying it all together: the implementation

Both the IMDB, and the BLAST examples explore the W4F/XML-QL paradigm for extracting Web pages. In this chapter, we describe the implementation details of our system. The scheme is essentially the same for both IMDB, and BLAST, and in the following we limit our discussion to the BLAST implementation, with the understanding that the arguments hold for either.

A schematic diagram describing the implementation is shown below in Figure 15. At the beginning, the main HTML page is displayed. This is similar to the NCBI BLAST page, and contains:

1. A text area in which the user can cut and paste a query sequence.
2. A choice of programs from the BLAST suite. We limit this choice to BLASTN (nucleotide query sequence against a nucleotide database), and BLASTP (protein query against a protein database).
3. A choice of Databases limited to swissprot, nr for proteins, and dbEST for DNA.

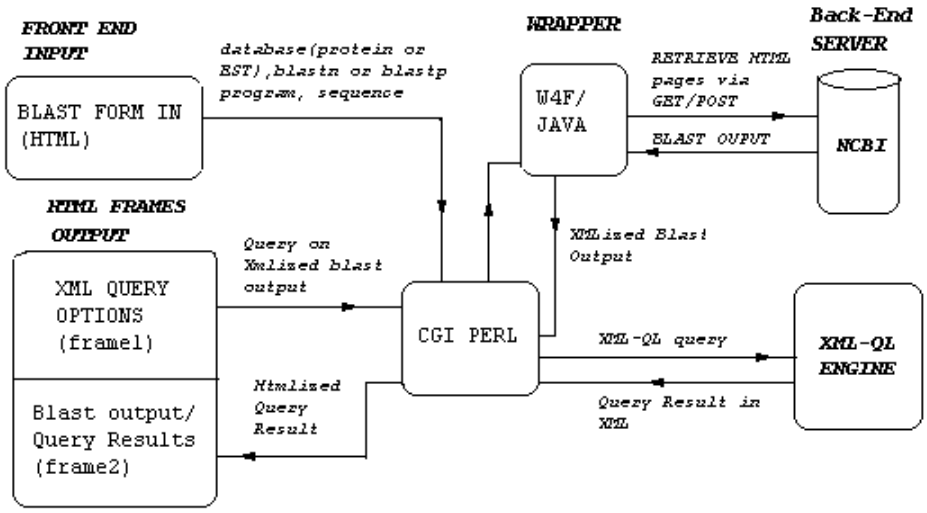



**Figure 15: A schematic of the Implementation**

When the user pushes the submit button, the query sequence, the choice of database and program is sent via a Perl CGI script to a Java program generated by the W4F script. This program queries the NCBI BLAST server to execute the appropriate BLAST query, and to retrieve the resulting BLAST output. Next, it scans through the HTML BLAST output, using Extraction rules to extract the alignments and statistics. For every database sequence that matches the query sequence, it queries GenBank to retrieve the entry and extract the species and tissue information. All this information is then XMLized and stored in an auxiliary file.

The Perl CGI script that had called the JAVA script now outputs an HTML screen, with two frames. The user sees these two frames, with the lower one displaying an Htmlized version of the resulting XML document (This will not be needed with XML enabled browsers), and the other displaying options for XML queries. Currently we do not offer users the option of typing in their own XML queries. The options offered roughly correspond to the functionality described in section 3.4. When the user selects one of these options, a second Perl CGI script is called that executes the selected XML-QL query. The result of this query is another XML file, and its HTMLized version is now displayed in the second frame.

## 5. References

[1.] S. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman. Basic local alignment search tool. *J Mol Biol* 1990 Oct 5;215(3):403-10.

[2.] A. Bairoch, R. Apweiler, The SWISS-PROT protein sequence data bank and its supplement TrEMBL in 1999. *Nucleic Acids Res* 1999 Jan 1;27(1):49-54.

[3.] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu. XML-QL: A Query Language for XML. W3C Note (1998).

[4.] A. Sahuguet, F. Azavant, W4F: a WysiWyg Web Wrapper Factory. Univ. of Pennsylvania Technical Report (1998).

[5.] A. Sahuguet, F. Azavant. W4F, 1998. http://db.cis.upenn.edu/W4F.

[6.] D. Schach, J. Lapp, and J. Robie. XML Query Language (XQL), 1998. QL'98 - The Query Languages Workshop.

[7.] L. Wall, T. Christiansen, R. L. Schwartz. Programming Perl. O'Reilly & Associates, 1996.

[8.] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, 1998. http://www.w3.org/TR/1998/REC-xml-19980210

[9.] S.Abiteboul and D. Quass and J.McHugh and J. Widom and J. Wiener, The Lorel Query Language for Semistructured Data, International Journal on Digital Libraries, vol. 1, no. 1, pp. 68-88,4/1997

## 6. Appendix 1 : W4F wrapper for IMDB

```
Schema{
      mov_url[] data;
      String title;
      String Check_ST;
      String url_ST;
      String ST_price;
      String ST_title;
      String [] genres;
      String director;
}

EXTRACTION_RULES {

       data = html.body[0].table[2].tr[0].td[1].div[0].ol[0].li[0-10]
                ( .a[0].getAttr(href) // url for movies
                );
       genres = html.body->td[i:0].a[*].txt
                where html.body->td[i].b[0].txt = "Genre";
       title = html.body[0].table[1].tr[0].td[2].table[0].tr[0].td[0].h1[0].strong[0]
               .txt;
       Check_ST = html.body[0].table[1].tr[0].td[2].div[0].p[0].a[1].txt;
       url_ST = html.body[0].table[1].tr[0].td[2].div[0].dl[0].dd[1].a[0]
                .getAttr(href);
       ST_price = html.body[0].table[1].tr[0].td[2].font[2].b[1].font[0].txt;
       ST_title = html.body[0].table[1].tr[0].td[2].font[0].b[0].pcdata[0].txt;
       director = html.body[0].table[1].tr[0].td[2].table[0].tr[1].td[1].table[0].
                     tr[1].td[0].a[0].txt;
}

RETRIEVAL_RULES {

  get(String act_nm){
      GET "http://us.imdb.com/Name?$act_nm$";
  }
  getnext_url(String next){
      GET "http://us.imdb.com$next$";
  }
  getsoundTr(String mv_title){
      GET "http://us.imdb.com/Soundtracks?$mv_title$";
  }
  getSales(String mv_title){
      GET "http://us.imdb.com/Sales?$mv_title$";
  }
  getSTurl(String st_url){
      GET "http://us.imdb.com/$st_url$";
  }
}
```

```
JAVACODE {
  public static void main(String args[]) throws Exception {
    BufferedReader stdin=new BufferedReader(new InputStreamReader(System.in));
    String message,message1;
    String str=new String("/sales");
    message=stdin.readLine();
    StringTokenizer reader=new StringTokenizer(message,"=");
    message1=reader.nextToken();
    message1=reader.nextToken();
    StringTokenizer reader1=new StringTokenizer(message1,"+");
    String[] a;
    a=new String[2];
    int j=0;
    while(reader1.countTokens()>0 && j<2){
      a[j]=new String(reader1.nextToken());
      j++;
    }
    String actor=a[1]+","+"+"+a[0];
    movie y=(movie.get(actor)) ;
    int i,l;
    movie x;
    movie z;
    movie s;
    movie t;
    System.out.println("<?xml version="+"\"1.0\""+"?>");
    System.out.println("<Root>");
    for(i=0;i<y.data.length;i++){
      l=((y.data[i].url).length());
      x=movie.getnext_url(y.data[i].url);
      if (x.title!=null){
          System.out.println("  "+"<Movie><name>"+x.title+"</name>");
          if(x.director!=null){
              System.out.println("  "+"<dir>"+x.director+"</dir>");
          }
          int k=0;
          if (x.genres!=null){
              System.out.println("<Categories>");
              while (k<x.genres.length){
                 System.out.println("<genre>"+x.genres[k]+"</genre>");
                 k++;
              }
              System.out.println("</Categories>");
           }
           s=movie.getSales((y.data[i].url).substring(7,l));
```

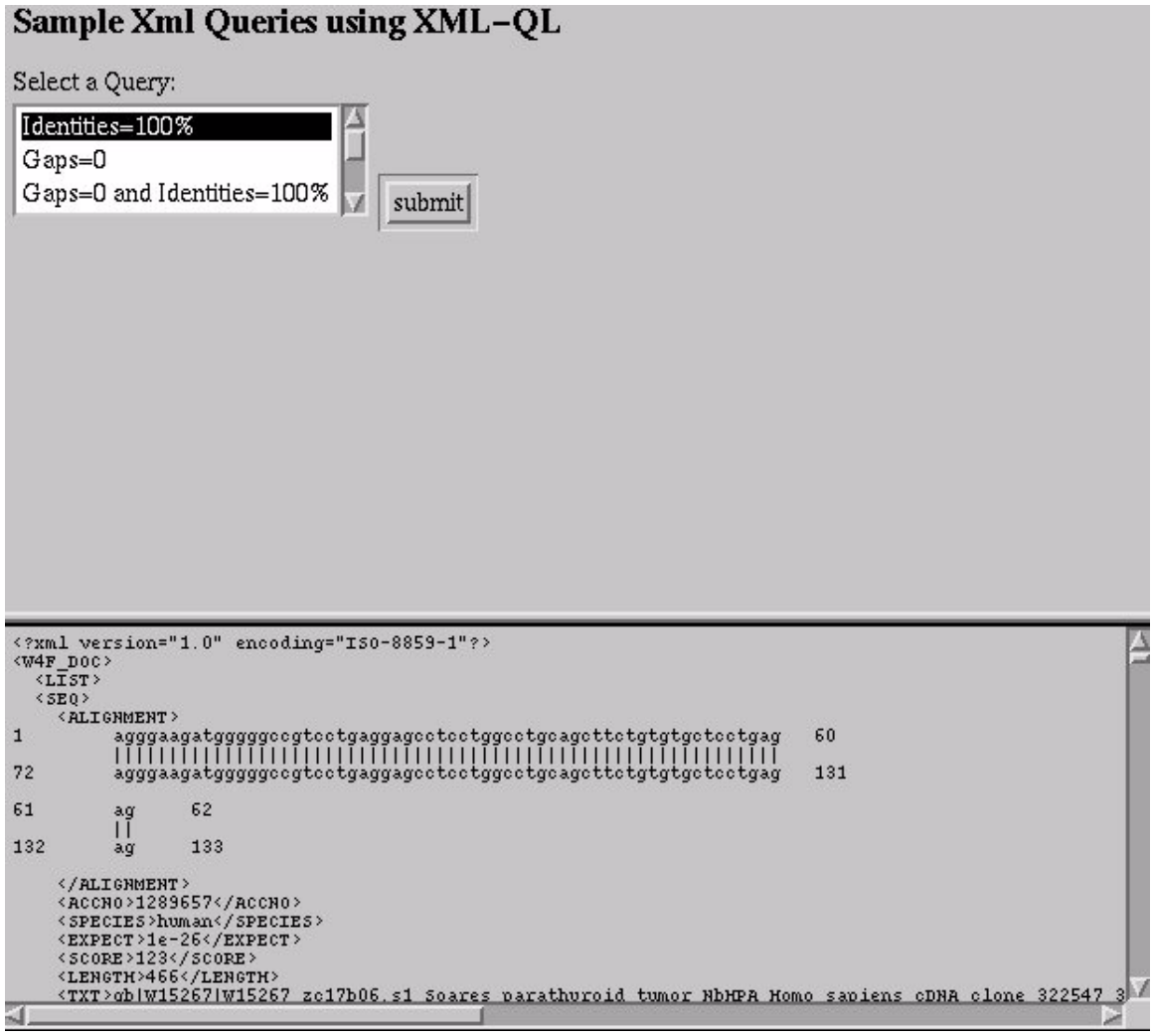
```
            if(s.url_ST!=null){
               if(str.equals((s.url_ST).substring(0,6))){
                  t=movie.getSTurl(s.url_ST);
                  if (t.ST_price !=null && t.ST_title!=null){
                      System.out.println("            "+"<Soundtrack>"+t.ST_title
                                    +"<price>"+t.ST_price+"</price></Soundtrack>");
                  }
               }
            }
            System.out.println("</Movie>");
        }
      }
      System.out.println("</Root>");
    }
}
```

## 7. Appendix 2 : W4F wrapper for BLAST

```
SCHEMA {

        Seque[] seq1;
        String[] tissue;
        String[] species;
        String[] accno;
        String[][] alignQ;
        String[][] alignQB;
        String[][] alignQE;
        String[][] alignS;
        String[][] alignSB;
        String[][] alignSE;
        String[][] alignM;
}

EXTRACTION_RULES {
   seq1=html.body[0].form[0]->pre[3-] (.txt, match/(.*)Length/
                           # .txt, match /Score\\s=\\s*(\\d+.\\d+)/
                           # .txt,match /Expect\\s*=\\s*(\\S+)/
                           # .txt,match /Identities\\s*=\\s*\\d+[^\\d]\\d+\\s*[^d](\\d+)/
                           # .txt,match /Gaps\\s*=\\s*(\\d+)/
                           # .txt,match /Length\\s*=\\s*(\\d+)/
                           # .txt,match /Query:\\s*(\\d+)/          // QBegin
                           # .txt,match /Sbjct:\\s+(\\d+)/          // SBegin
                            # .txt,match /Query.*\\s+(\\d+).*Sbjct/  // QEnd
                           # .txt,match /Sbjct.*\\s+(\\d+)/         // SEnd
                           );
   alignQ=html.body[0].form[0]->pre[3-].txt,match /(Query.*)/,split /Que/,
                  match /ry\\:\\s*\\d+([\\sa-zA-Z\\-]+)[\\d]+/;
   alignQB=html.body[0].form[0]->pre[3-].txt,match /(Query.*)/,split /Que/,
                  match /ry\\:\\s*(\\d+)[\\sa-zA-Z\\-]+[\\d]+/;
   alignQE=html.body[0].form[0]->pre[3-].txt,match /(Query.*)/,split /Que/,
                  match /ry\\:\\s*\\d+[\\sa-zA-Z\\-]+([\\d]+)/;
   alignS=html.body[0].form[0]->pre[3-].txt,match /(Sbjct.*)/,split /Sbj/,
                  match /ct\\:\\s*\\d+([\\sa-zA-Z\\-]+)[\\d]+/;
   alignSB=html.body[0].form[0]->pre[3-].txt,match /(Sbjct.*)/,split /Sbj/,
                  match /ct\\:\\s*(\\d+)[\\sa-zA-Z\\-]+[\\d]+/;
   alignSE=html.body[0].form[0]->pre[3-].txt,match /(Sbjct.*)/,split /Sbj/,
                  match /ct\\:\\s*\\d+[\\sa-zA-Z\\-]+([\\d]+)/;
   alignM=html.body[0].form[0]->pre[3-].txt,match /(Query.*)/,split /Que/,
                  match /ry\\:\\s*\\d+[\\sa-zA-Z\\-]+[\\d]+(.*)Sbjct/;
   accno=html.body[0].form[0]->pre[3-].a[0].getAttr(name);
   tissue=html.body[0].form[0].pre[0].pcdata[*].txt,
                  match /FEATURES.*tissue_type="([^"]+)"/;
   species=html.body[0].form[0].pre[0].pcdata[*].txt,
                  match  /\\s+SOURCE\\s*([^\\.]+)\\./;



}

RETRIEVAL_RULES {

        get(String p1,String p2,String p3,String p4){
               METHOD: POST;
               URL: "http://www.ncbi.nlm.nih.gov/cgi-bin/BLAST/nph-newblast";
               PARAM:"PROGRAM"=$p1$,"DATALIB"=$p2$,"FSET"="isset",
                     "OVERVIEW"="isset+CHECKED","INPUT_TYPE"=$p3$,"SEQUENCE"=$p4$ ;
        }

        getorgspN(String Acc){
               GET "http://www.ncbi.nlm.nih.gov/htbin-
        post/Entrez/query?form=6&dopt=g&db=n&uid=$Acc$";

        }

        getorgspP(String Acc){
               GET "http://www.ncbi.nlm.nih.gov/htbin-
```

```
                    post/Entrez/query?form=6&dopt=g&db=p&uid=$Acc$";
       }

}



JAVACODE {


  public static void main(String args[]) throws Exception {
    blast4 x;
    int k = 0;

    x=blast4.get(args[0],args[1],args[2],args[3]);
    blast4.MyXmlHd();

    while(k<x.seq1.length){
    System.out.println("  <SEQ>");

    System.out.println("    <ALIGNMENT>");
    int l=0;
/*
    if (x.alignQ[1]==null ){
    System.out.println("AlignQ is null");
}else{
System.out.println(x.alignQ[1][1]);
}*/


    while(l<x.alignQ[k].length) {

      System.out.println(x.alignQB[k][l]+"\t"+x.alignQ[k][l]+"\t"+x.alignQE[k][l]);
      System.out.println("\t"+x.alignM[k][l]+"\t");
      System.out.println(x.alignSB[k][l]+"\t"+x.alignS[k][l]+"\t"+x.alignSE[k][l]);
      System.out.println(" ");
      l++;
    }
System.out.println("    </ALIGNMENT>");
// System.out.println("accno"+x.accno[k]+"prg"+args[0]);
 blast4.orgspacc(x.accno[k],args[0]);

 blast4.MyXml(x.seq1[k]);
 k++;
}
blast4.MyXmlTl();

}

public static void MyXml(Seque s){
  System.out.println("    <EXPECT>"+s.expect+"</EXPECT>");
  System.out.println("    <SCORE>"+s.score+"</SCORE>");
  System.out.println("    <LENGTH>"+s.seqlen+"</LENGTH>");
  System.out.println("    <TXT>"+s.seqtxt+"</TXT>");
  System.out.println("    <GAPS>"+s.gaps+"</GAPS>");
  System.out.println("    <IDENTITIES>"+s.identities+"</IDENTITIES>");
  System.out.println("    <QBEGIN>"+s.qbegin+"</QBEGIN>");
  System.out.println("    <SBEGIN>"+s.sbegin+"</SBEGIN>");
  System.out.println("    <QEND>"+s.qend+"</QEND>");
  System.out.println("    <SEND>"+s.send+"</SEND>");
  System.out.println("  </SEQ>");
}

public static void MyXmlHd(){
  System.out.println("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>");
  System.out.println("<W4F_DOC>");
  System.out.println("  <LIST>");
}

public static void orgspacc(String acc,String prg) throws Exception{

  if (acc!= null) {
  System.out.println("    <ACCNO>"+acc+"</ACCNO>");
  blast4 y1;
  String Prg=new String(prg);
  if (Prg.equals("blastp")){
```

```
      y1=blast4.getorgspP(acc);
//    System.out.println("got the page");
  }else{
    y1=blast4.getorgspN(acc);
  }
  int i,j;
  i=0;
  if(y1.tissue!=null){
    while(i<y1.tissue.length){
      if (y1.tissue[i]==null){
        i++;
      }else{
        System.out.println("    <TISSUE>"+y1.tissue[i]+"</TISSUE>");
        i++;
      }
    }
  }
if(y1.species!=null){
  j=0;
  while(j<y1.species.length){
    if (y1.species[j]==null){
      j++;
    }else{
      System.out.println("    <SPECIES>"+y1.species[j]+"</SPECIES>");
      j++;
    }
  }
}else System.out.println("Species is null");

}
}
public static void MyXmlTl(){
  System.out.println("  </LIST>");
  System.out.println("</W4F_DOC>");
}

}
```


Sample Xml Queries using XML-QL
Select a Query:
Identities=100%
Gaps=0
Gaps=0 and Identities=100%
submit
<?xml version="1.0" encoding="ISO-8859-1"?>
<W4F_DOC>
<LIST>
<SEQ>
<ALIGNMENT>
1 agggaagatgggggccgtcctgaggagcctcctggcctgcagcttctgtgtgctcctgag 60
72 agggaagatgggggccgtcctgaggagcctcctggcctgcagcttctgtgtgctcctgag 131
61 ag 62
132 ag 133
</ALIGNMENT>
<ACCNO>1289657</ACCNO>
<SPECIES>human</SPECIES>
<EXPECT>1e-26</EXPECT>
<SCORE>123</SCORE>
<LENGTH>466</LENGTH>
<TXT>gb|W15267|W15267 zc17b06.s1 Soares parathyroid tumor NbHPA Homo sapiens cDNA clone 322547 3