Departamento de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla
Avda Reina Mercedes, s/n. 41012 SEVILLA
Fax : 95 455 71 39. Tlf: 95 455 71 39. E-mail: lsi@lsi.us.es

XQuery

J. J. Gutiérrez, M. J. Escalona, M. Mejías, J. Torres, D.Villadiego

Universidad de Sevilla

Lenguajes y Sistemas Informáticos

España

{javierj, escalona, risoto, torres}@lsi.us.es

Sevilla, Febrero de 2005

HOJA DE CONTROL DE MODIFICACIONES

Número	**Fecha**	**Descripción**	**Autores**
0	06 – 2.004	Desarrollo Inicial	Los autores
1	03 – 2005	Correcciones del proceso de evaluación	Los autores

Índice.

1. Introducción.

Extensible Markup Language[1] (más conocido como XML) es un formato para almacenar datos extremadamente versátil, utilizado para representar una gran cantidad distinta de información como, por ejemplo, páginas web (XHTML), mensajes web entre aplicaciones (SOAP), libros y artículos, datos de negocio, representación de bases de datos relacionales, interfaces de usuario (XUL), transacciones financieras y bancarias, partidas de ajedrez, recetas de cocina, trazas del sistema, gráficos vectoriales (SVG), sindicación de contenidos (RSS), archivos de configuración, documentos de texto (OpenOffice.org) y manuscritos en griego clásico.

Un conjunto de datos expresados en XML es una cadena de texto donde cada uno de los datos está delimitado por etiquetas de la forma <T> ... </T> o se incluye como atributo de una etiqueta de la forma <T A="..."> ... </T>. Mediante esta representación es posible expresar también la semántica de cada uno de los datos.

A continuación se muestra un conjunto de datos en XML que contiene información sobre un libro titulado "TCP/IP Illustrated", escrito por "Stevens, W.", editado por "Addison-Wesley" y cuyo precio es 69.95.

```
<bib>
    <libro>
        <titulo>TCP/IP Illustrated</titulo>
        <autor>
           <apellido>Stevens</apellido>
           <nombre>W.</nombre>
     </autor>
        <editorial>Addison-Wesley</editorial>
        <precio> 65.95</precio>
    </libro>
</bib>
```

En XML las etiquetas se estructuran en forma de árbol n-ário. En la figura 1 se muestra la estructura en árbol del conjunto de datos referidos al libro.

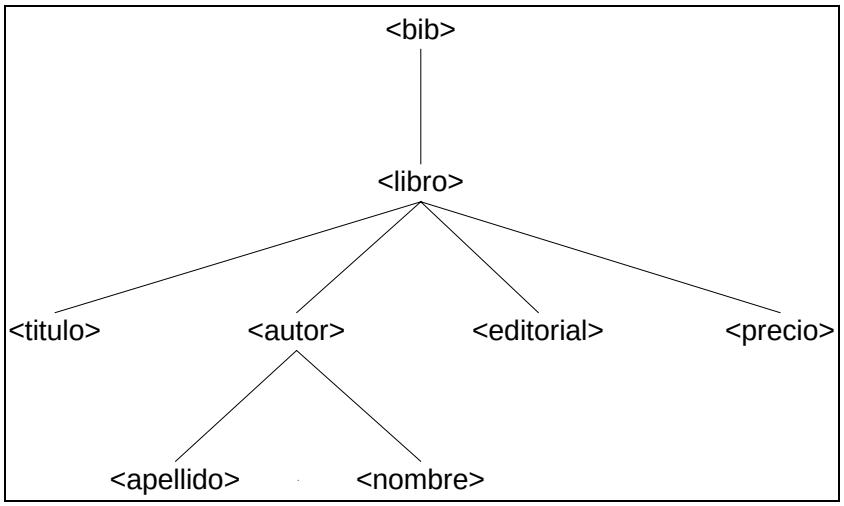

Figura 1. Representación en forma de árbol de un conjunto de etiquetas XML.

1.1. ¿Por qué necesitamos XQuery?.

Actualmente, XML se ha convertido en una herramienta de uso cotidiano en los entornos de tratamiento de información y en los entornos de programación. Sin embargo, a medida que se emplea en un mayor número de proyectos de complejidad y tamaño crecientes y la cantidad de datos almacenados en XML aumenta, se comprueba que, las herramientas más habituales para manipular desde un programa un árbol con un conjunto de datos en XML, los parsers SAX y DOM[1], no son prácticas para manejar grandes y complejas colecciones de datos en XML.

El principal problema a la hora de procesar colecciones de datos en XML a bajo nivel mediante parsers DOM y SAX es la necesidad de escribir una gran cantidad de código para realizar el procesamiento. Este código consiste, básicamente, en recorrer un árbol, para un parser DOM, o detallar una lista de acciones según la etiqueta que se encuentre para parser SAX.

Posteriormente a los parsers DOM y SAX, ha aparecido una nueva familia de herramientas conocidas genéricamente como bindings[2], que, si bien en ciertos aspectos son tremendamente útiles y ahorran una gran cantidad de trabajo, no son útiles en todas las situaciones en las que se puede utilizar XML.

Otra solución existente en la actualidad es el estándar XSLT[3] que permite definir transformaciones sobre colecciones datos en XML en otros formatos, como HTML o PDF y las herramientas que le dan soporte. Sin embargo XSLT sólo es aplicable en los casos en los que desee obtener una representación distinta de los datos, no cuando se desea localizar una información concreta dentro de dicho conjunto de datos.

Un ejemplo de un escenario donde no es aplicable ninguna de las herramientas comentadas hasta ahora (parsers, binding y XSLT) lo encontramos a la hora de consultar la información en los actuales servidores de bases de datos que ya incorporan

[1] Para una descripción del modo de trabajo y características principales de los parsers SAX y DOM consultar el apéndice I.

[2] Para más información sobre la manera de trabajar y un listado de herramientas de binding consultar el apéndice II.

funciones para poder obtener los datos en XML o bases de datos XML nativas (Native XML Database). El escenario de uso más común se da cuando tenemos que realizar alguna búsqueda en un documento o conjunto de documentos con gran cantidad de datos en XML, o bien hemos de trabajar sólo sobre un subconjunto de todos los datos XML y queremos obtener dicho subconjunto de una forma sencilla, para evitar trabajar con toda la colección de datos.

Dar solución a este escenario de uso mediante código escrito con la ayuda de un parser SAX es rápido y sencillo de desarrollar si la consulta y la estructura de la información no tienen una gran complejidad. A medida que la consulta va ganando en complejidad, es necesario definir comportamientos más complejos a ejecutar cuando se encuentre una etiqueta y se necesita un mayor número de almacenamientos temporales de datos. Otro problema es que el código está fuertemente ligado a la estructura de los datos en XML, por lo que cualquier cambio en la consulta o en la estructura de los datos obliga a volver a escribir el código. Además el código resultante es muy poco parametrizable y reutilizable por lo que para desarrollar una consulta similar habría que volverla a escribir desde el principio, sin poder aprovechar nada de lo escrito. Dar solución a este escenario de uso mediante código escrito con la ayuda un parser DOM añade, a los problemas de reusabilidad, complejidad y fuerte acoplamiento anteriores, la exigencia de tener cargar los datos en memoria antes de comenzar el recorrido, lo cual puede resultar imposible para grandes volúmenes de datos.

Las herramientas de bindings favorecen la sencillez en el proceso de carga en memoria de los datos, pero, aplicadas a este escenario de uso, no facilita la tarea a la hora de escribir una consulta. Primero, muchos bindings necesitan los documentos de definición de los documentos XML (DTDs o XML-Schemas) para poder generar las clases y el código de vinculación, por lo que si no se tienen y la estructura de la información es compleja puede resultar que el tiempo que ganamos con el uso de estas herramientas lo perdamos escribiendo el DTD o XML-Schemas correspondiente. También existen herramientas que pueden trabajar sin los DTD o XML-Schemas, pero entonces es necesario escribir a mano todas las clases necesarias para almacenar la colección de datos en XML. Además estas herramientas siguen teniendo el problema de que requieren cargar en memoria todos los datos. Más aún, lo que estamos haciendo es cambiar un conjunto de datos expresados mediante etiquetas XML por un conjunto de datos expresados por los atributos de un conjunto de objetos, por lo que sigue siendo necesario escribir la algoritmia necesaria para esa consulta, algoritmia poco reutilizable en posteriores consultas.

A la vista de todo lo comentado hasta ahora, se hace necesaria la aparición de un lenguaje que permita definir de forma rápida y compacta, consultas o recorridos complejos sobre colecciones de datos en XML los cuales devuelvan todos los nodos que cumplan ciertas condiciones. Este lenguaje debe ser, además, declarativo, es decir, independientemente de la forma en que se realice el recorrido o de donde se encuentren los datos. Este lenguaje ya existe y se llama XQuery.

A lo largo de este artículo vamos a mostrar, con profusión de ejemplos, las características principales de este lenguaje y vamos a mostrar como utilizar uno de los motores open-source existentes para incorporar capacidad de procesado de consulta XQuery en nuestras aplicaciones.

2. Definición de XQuery.

De manera rápida podemos definir XQuery con un símil en el que XQuery es a XML lo mismo que SQL es a las bases de datos relacionales.

XQuery[4][5] es un lenguaje de consulta diseñado para escribir consultas sobre colecciones de datos expresadas en XML. Abarca desde archivos XML hasta bases de datos relacionales con funciones de conversión de registros a XML. Su principal función es extraer información de un conjunto de datos organizados como un árbol n-ário de etiquetas XML. En este sentido XQuery es independiente del origen de los datos.

XQuery es un lenguaje funcional, lo que significa que, en vez de ejecutar una lista de comandos como un lenguaje procedimental clásico, cada consulta es una expresión que es evaluada y devuelve un resultado, al igual que en SQL. Diversas expresiones pueden combinarse de una manera muy flexible con otras expresiones para crear nuevas expresiones más complejas y de mayor potencia semántica.

XQuery está llamado a ser el futuro estándar de consultas sobre documentos XML. Actualmente, XQuery es un conjunto de borradores[3] en el que trabaja el grupo W3C[20]. Sin embargo, a pesar de no tener una redacción definitiva ya existen o están en proceso numerosas implementaciones de motores y herramientas que lo soportan.

2.1. Requerimientos técnicos de XQuery.

El grupo de trabajo en XQuery del W3C[20] ha definido un conjunto de requerimientos técnicos para este lenguaje. Los más importantes se detallan a continuación.

- XQuery debe ser un lenguaje declarativo. Al igual que SQL hay que indicar que se quiere, no la manera de obtenerlo.
- XQuery debe ser independiente del protocolo de acceso a la colección de datos. Una consulta en XQuery debe funcionar igual al consultar un archivo local que al consultar un servidor de bases de datos que al consultar un archivo XML en un servidor web.
- Las consultas y los resultados deben respetar el modelo de datos XML
- Las consultas y los resultados deben ofrecer soporte para los namespace[4].
- Debe ser capaz de soportar XML-Schemas y DTDs y también debe ser capaz de trabajar sin ninguno de ellos.
- XQuery debe poder trabajar con independencia de la estructura del documento, esto es, sin necesidad de conocerla.
- XQuery debe soportar tipos simples, como enteros y cadenas, y tipos complejos, como un nodo compuesto por varios nodos hijos.
- Las consultan deben soportar cuantificadores universales (para todo) y existenciales (existe).

[3] Para ver el conjunto completo de borradores y sus URLs consultar el apéndice IV.

[4] Un namespace[1] es un espacio de definición de etiquetas que permite que dos diseñadores puedan utilizar el mismo nombre de etiqueta con dos significados semánticos distintos sin que entren en conflicto. Dos etiquetas con el mismo nombre declaradas en namespaces diferentes se consideran dos etiquetas diferentes.

- Las consultas deben soportar operaciones sobre jerarquías de nodos y secuencias de nodos.
- Debe ser posible en una consulta combinar información de múltiples fuentes.
- Las consultas deben ser capaces de manipular los datos independientemente del origen de estos.
- Mediante XQuery debe ser posible definir consultas que transformen las estructuras de información originales y debe ser posible crear nuevas estructuras de datos.
- El lenguaje de consulta debe ser independiente de la sintaxis, esto es, debe ser posible que existan varias sintaxis distintas para expresar una misma consulta en XQuery.

Aunque XQuery y SQL puedan considerarse similares en casi la totalidad de sus aspectos, el modelo de datos sobre el que se sustenta XQuery es muy distinto del modelo de datos relacional sobre el que sustenta SQL, ya que XML incluye conceptos como jerarquía y orden de los datos que no están presentes en el modelo relacional. Por ejemplo, a diferencia de SQL, en XQuery el orden es que se encuentren los datos es importante y determinante, ya que no es lo mismo buscar una etiqueta <B> dentro de una etiqueta <A> que todas las etiquetas <B> del documento (que pueden estar anidadas dentro de una etiqueta <A> o fuera).

XQuery ha sido construido sobre la base de Xpath[3]. Xpath[5] es un lenguaje declarativo para la localización de nodos y fragmentos de información en árboles XML. XQuery se basa en este lenguaje para realizar la selección de información y la iteración a través del conjunto de datos.

2.2. Colección de datos de ejemplo.

En los ejemplos de consultas que veremos a lo largo de este trabajo, vamos a trabajar con un conjunto de datos compuesto por fichas de varios libros almacenadas en un archivo local llamado “libros.xml”, cuyo contenido se muestra a continuación.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<bib>
    <libro año="1994">
        <titulo>TCP/IP Illustrated</titulo>
        <autor>
           <apellido>Stevens</apellido>
           <nombre>W.</nombre>
     </autor>
        <editorial>Addison-Wesley</editorial>
        <precio> 65.95</precio>
    </libro>

    <libro año="1992">
        <titulo>Advan Programming for Unix environment</titulo>
        <autor>
```

[5] Para más información sobre XPath consultar el apéndice III y el apéndice IV.

```
            <apellido>Stevens</apellido>
            <nombre>W.</nombre>
      </autor>
         <editorial>Addison-Wesley</editorial>
         <precio>65.95</precio>
    </libro>

    <libro año="2000">
         <titulo>Data on the Web</titulo>
         <autor>
            <apellido>Abiteboul</apellido>
            <nombre>Serge</nombre>
      </autor>
         <autor>
            <apellido>Buneman</apellido>
            <nombre>Peter</nombre>
      </autor>
         <autor>
            <apellido>Suciu</apellido>
            <nombre>Dan</nombre>
      </autor>
         <editorial>Morgan Kaufmann editorials</editorial>
         <precio>39.95</precio>
    </libro>

    <libro año="1999">
         <titulo> Economics of Technology for Digital TV</titulo>
         <editor>
                 <apellido>Gerbarg</apellido>
                 <nombre>Darcy</nombre>
                 <afiliacion>CITI</afiliacion>
         </editor>
             <editorial>Kluwer Academic editorials</editorial>
         <precio>129.95</precio>
    </libro>

</bib>
```

A continuación se muestra el contenido del DTD correspondiente al archivo "libros.xml".

```
<!ELEMENT bib (libro* )>
<!ELEMENT libro (titulo,(autor+ | editor+ ),editorial, precio )>
<!ATTLIST libro year CDATA #REQUIRED >
<!ELEMENT autor (apellido, nombre )>
<!ELEMENT editor (apellido, nombre, afiliacion )>
<!ELEMENT titulo (#PCDATA )>
<!ELEMENT apellido (#PCDATA )>
<!ELEMENT nombre (#PCDATA )>
<!ELEMENT afiliacion (#PCDATA )>
<!ELEMENT editorial (#PCDATA )>
```

```
<!ELEMENT precio (#PCDATA )>
```

En algunas consultas también necesitaremos combinar la información contenida en el archivo “libros.xml” con la información contenida en el archivo “comentarios.xml”. El contenido de este archivo se muestra a continuación.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<comentarios>
    <entrada>
        <titulo>Data on the Web</titulo>
        <precio>34.95</precio>
        <comentario>
     Un libro muy bueno sobre bases de datos.
        </comentario>
    </entrada>
    <entrada>
        <titulo>Advanced Programming in the Unix
environment</titulo>
        <precio>65.95</precio>
        <comentario>
     Un libro claro y detallado de programación en UNIX.
        </comentario>
    </entrada>
    <entrada>
        <titulo>TCP/IP Illustrated</titulo>
        <precio>65.95</precio>
        <comentario>
     Uno de los mejores libros de TCP/IP
        </comentario>
    </entrada>
</comentarios>
```

A continuación se muestra el contenido del DTD correspondiente al archivo "comentarios.xml".

```
<!ELEMENT comentarios (entrada*)>
<!ELEMENT entrada   (titulo, precio, comentario)>
<!ELEMENT titulo   (#PCDATA)>
<!ELEMENT precio   (#PCDATA)>
<!ELEMENT comentario  (#PCDATA)>
```

3. Consultas en XQuery.

Una consulta en XQuery[4][5] es una expresión que lee una secuencia de datos en XML y devuelve como resultado otra secuencia de datos en XML.

Un detalle importante es que, a diferencia de lo que sucede en SQL, en XQuery las expresiones y los valores que devuelven son dependientes del contexto. Por ejemplo los nodos que aparecerán en el resultado dependen de los namespaces, de la posición donde aparezca la etiqueta raíz del nodo (dentro de otra, por ejemplo), etc.

En XQuery las consultas pueden estar compuestas por cláusulas de hasta cinco tipos distintos. Las consultas siguen la norma FLWOR (leído como flower), siendo FLWOR las siglas de For, Let, Where, Order y Return. A continuación, en la tabla 1, se describe la función de cada bloque:

For	Vincula una o más variables a expresiones escritas en XPath, creando un flujo de tuplas en el que cada tupla está vinculada a una de las variable.
Let	Vincula una variable al resultado completo de una expresión añadiendo esos vínculos a las tuplas generadas por una cláusula for o, si no existe ninguna cláusula for, creando una única tupla que contenga esos vínculos.
Where	Filtra las tuplas eliminando todos los valores que no cumplan las condiciones dadas.
Order by	Ordena las tuplas según el criterio dado.
Return	Construye el resultado de la consulta para una tupla dada, después de haber sido filtrada por la cláusula where y ordenada por la cláusula order by.

Tabla 1. Posibles cláusulas en una consulta XQuery.

En XQuery, cuando usamos el térmico tupla, nos estamos refiriendo a cada uno de los valores que toma una variable.

A continuación, en la figura 2, se muestra gráficamente el orden en que se ejecuta cada cláusula de una consulta y los resultados de cada una:

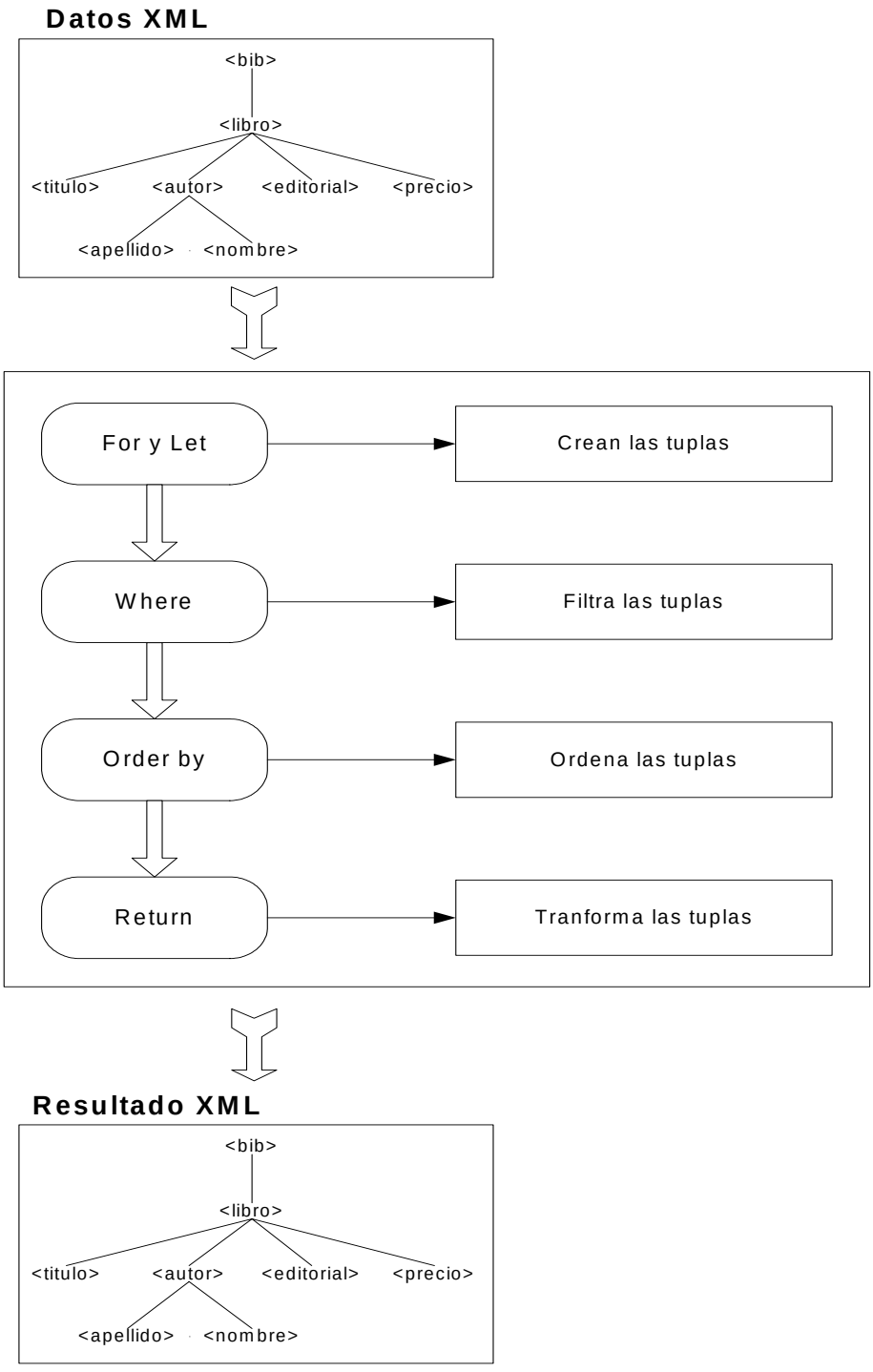

Figura 2. Orden de ejecución y resultado de las cinco cláusulas posibles.

En el siguiente ejemplo de cláusula for, la variable $b tomará como valor cada uno de los nodos libros que contenga en archivo “libros.xml”. Cada uno de esos nodos libros, será una tupla vinculada a la variable $b.

```
for $b in document("libros.xml”)//bib/libro
```

A continuación se muestra un ejemplo de una consulta donde aparecen las 5 clausulas. La siguiente consulta devuelve los títulos de los libros que tengan más de dos autores ordenados por su título.

```
for $b in doc("libros.xml")//libro
let $c := $b//autor
where count($c) > 2
order by $b/titulo
return $b/ titulo
```

El resultado de esta consulta se muestra a continuación.

```
<title>Data on the Web</title>
```

Las barras: "//" no indican comentarios, sino que son parte de la expresión XPath que indica la localización de los valores que tomará la variable $b. En esta consulta la función count() hace la misma función que en SQL, contar el número de elementos, nodos en este caso, referenciados por la variable $c. La diferencia entre la cláusula for y la cláusula let se explica con más detalle en un punto posterior.

Una expresión FLWOR vincula variables a valores con cláusulas for y let y utiliza esos vínculos para crear nuevas estructuras de datos XML.

A continuación se muestra otro ejemplo de consulta XQuery. La siguiente consulta devuelve los títulos de los libros del año 2.000. Como "año" es un atributo y no una etiqueta se le antecede con un carácter "@".

```
for $b in doc("libros.xml")//libro
where $b/@año = "2000"
return $b/titulo
```

El resultado de la consulta anterior se muestra a continuación.

```
<title>Data on the Web</title>
```

A continuación, en la figura 3, se muestra de forma gráfica como se ejecutaría la consulta anterior y los resultados parciales de cada una de sus cláusulas:

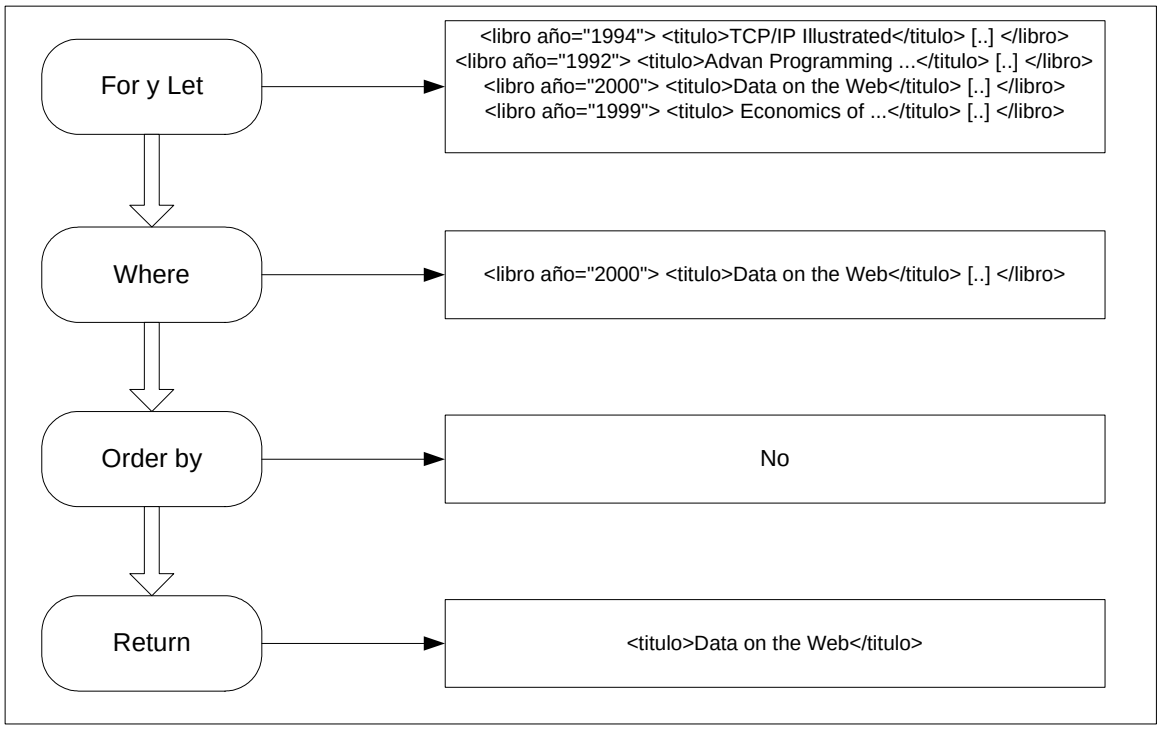

Figura 3. Ejecución de una consulta XQuery con los resultados de cada cláusula.

3.1. Reglas generales.

A continuación, enunciamos una serie de reglas que debe cumplir cualquier consulta escrita en XQuery[4][5]:

- For y let sirven para crear las tuplas con las que trabajará el resto de las cláusulas de la consulta y pueden usarse tantas veces como se desee en una consulta, incluso dentro de otras cláusulas. Sin embargo solo puede declararse una única cláusula where, una única cláusula order by y una única cláusula return.
- Ninguna de las cláusulas FLWOR es obligatoria en una consulta XQuery. Por ejemplo, una expresión XPath, como la que se muestra a continuación, es una consulta válida y no contiene ninguna de las cláusulas FLWOR.

```
doc("libros.xml")/bib/libro/titulo[/bib/libro/autor/apellido='Stevens']
```

Esta expresión XPath, que también es una consulta XQuery válida, devuelve los títulos de los libros que tengan algún autor de apellido ‘Stevens’.

Es posible especificar varios criterios de ordenación en la cláusula order by, separándolos por comas. Los criterios de ordenación se aplican por orden de izquierda a derecha.

3.2. Diferencias entre las cláusulas for y let.

Para ver claramente la diferencia entre una cláusula for y una cláusula let vamos a comenzar estudiando una misma consulta que muestre los títulos de todos los libros almacenados en el archivo “libros.xml”, primero con una cláusula for y, a continuación, con una cláusula let y vamos a detallar que diferencia hay en la información obtenida. La consulta con una cláusula for se muestra a continuación.

```
for $d in doc("libros.xml")/bib/libro/titulo
return
     <titulos>{ $d }</titulos>
```

El resultado de esta consulta se muestra a continuación:

```
<titulos>
     <titulo>TCP/IP Illustrated</titulo>
</titulos>
<titulos>
     <titulo>Advan Programming for Unix environment</titulo>
</titulos>
<titulos>
     <titulo>Data on the Web</titulo>
</titulos>
<titulos>
     <titulo> Economics of Technology for Digital TV</titulo>
</titulos>
```

A continuación repetimos la misma consulta sustituyendo la cláusula for una cláusula let.

```
let $d := doc("libros.xml")/bib/libro/titulo
return
<titulos>{ $d }</titulos>
```

El resultado de esta consulta se muestra a continuación.

```
<titulos>
     <titulo>TCP/IP Illustrated</titulo>
     <titulo>Advan Programming for Unix environment</titulo>
     <titulo>Data on the Web</titulo>
     <titulo> Economics of Technology for Digital TV</titulo>
</titulos>
```

Como se puede ver comparando los resultados obtenidos por ambas consultas, la cláusula for vincula una variable con cada nodo que encuentre en la colección de datos. En este ejemplo la variable $d va vinculándose a cada uno de los títulos de todos los libros del archivo "libros.xml", creando una tupla por cada título. Por este motivo aparece repetido el par de etiquetas <titulos>...</titulos> para cada título. La cláusula let, en cambio, vincula una variable con todo el resultado de una expresión. En este ejemplo, la variable $d se vincula a todos los títulos de todos los libros del archivo "libros.xml", creando una única tupla con todos esos títulos. Por este motivo solo aparece el par de etiquetas <titulos>...</titulos> una única vez.

Si una cláusula let aparece en una consulta que ya posee una o más cláusulas for, los valores de la variable vinculada por la cláusula let se añaden a cada una de las tuplas generadas por la cláusula for. Un ejemplo se muestra en la siguiente consulta:

```
for $b in doc("libros.xml")//libro
let $c := $b/autor
return
<libro>{ $b/titulo, <autores>{ count($c) }</autores>}</libro>
```

Esta consulta devuelve el título de cada uno de los libros de archivo "libros.xml" junto con el número de autores de cada libro. El resultado de esta consulta se muestra a continuación:

```
<libro>
     <titulo>TCP/IP Illustrated</titulo>
     <autores>1</autores>
</libro>
<libro>
     <titulo>Advanced Programming in the UNIX
Environment</titulo>
     <autores>1</autores>
</libro>
<libro>
     <titulo>Data on the Web</titulo>
     <autores>3</autores>
</libro>
<libro>
     <titulo>The Economics of Technology and Content for
Digital TV</titulo>
     <autores>0</autores>
</libro>
```

Si en la consulta aparece más de una cláusula for (o más de una variable en una clásula for), el resultado es el producto cartesiano de dichas variables, es decir, las tuplas generadas cubren todas las posibles combinaciones de los nodos de dichas variables. Un ejemplo se muestra en la siguiente consulta, la cual devuelve los títulos de todos los libros contenidos en el archivo “libros.xml” y todos los comentarios de cada libro contenidos en el archivo “comentarios.xml”.

```
for $t in doc("books.xml")//titulo,
     $e in doc("comentarios.xml")//entrada
where $t = $e/titulo
return <comentario>{ $t, $e/comentario }</comentario>
```

El resultado de esta consulta se muestra a continuación.

```
<comentario>
     <titulo>Data on the Web</titulo>
     <comentario>Un libro muy bueno sobre bases de
datos.</comentario>
</comentario>
<comentario>
     <titulo>Advanced Programming in the Unix
environment</titulo>
     <comentario>Un libro claro y detallado de programación en
UNIX.</comentario>
</comentario>
<comentario>
     <titulo>TCP/IP Illustrated</titulo>
     <comentario>Uno de los mejores libros de
TCP/IP.</comentario>
</comentario>
```

3.3. Funciones de entrada.

XQuery utiliza las funciones de entrada en las cláusulas for o let o en expresiones XPath para identificar el origen de los datos. Actualmente el borrador de XPath y XQuery define dos funciones de entrada distintas, doc(URI) y collection(URI).

La función doc(URI) devuelve el nodo documento, o nodo raíz, del documento referenciado por un identificador universal de recursos (URI). Esta es la función más habitual para acceder a la información almacenada en archivos.

La función collection(URI) devuelve una secuencia de nodos referenciados por una URI, sin necesidad de que exista un nodo documento o nodo raíz. Esta es la función más habitual para acceder a la información almacenada en una base de datos que tenga capacidad para crear estructuras de datos XML.

3.4. Expresiones condicionales.

Además de la cláusula where, XQuery también soporta expresiones condicionales del tipo “if-then-else” con la misma semántica que en los lenguajes de programación más habituales (C, Java, Delphi, etc..). Por ejemplo, la siguiente consulta devuelve los títulos de todos los libros almacenados en el archivo "libros.xml" y sus dos primeros autores. En el caso de que existan más de dos autores para un libro, se añade un tercer autor "et al.".

```
for $b in doc("libros.xml")//libro
return
  <libro>
     { $b/titulo }
     {
       for $a at $i in $b/autor
       where $i <= 2
       return <autor>{string($a/last), ", ",
       string($a/first)}</autor>
     }
     {
       if (count($b/autor) > 2)
       then <autor>et al.</autor>
       else ()
     }
  </libro>
```

El resultado de esta consulta se muestra a continuación.

```
<libro>
     <titulo>TCP/IP Illustrated</titulo>
     <autor>Stevens, W.</autor>
</libro>
<libro>
     <titulo>Advanced Programming in the Unix
Environment</titulo>
     <autor>Stevens, W.</autor>
</libro>
<libro>
     <titulo>Data on the Web</titulo>
     <autor>Abiteboul, Serge</autor>
     <autor>Buneman, Peter</autor>
     <autor>et al.</autor>
</libro>
```

La cláusula where de una consulta permite filtrar las tuplas que aparecerán en el resultado, mientras que una expresión condicional nos permite crear una u otra estructura de nodos en el resultado que dependa de los valores de las tuplas filtradas.

A diferencia de la mayoría de los lenguajes, la cláusula else es obligatoria y debe aparecer siempre en la expresión condicional. El motivo de esto es que toda expresión en XQuery debe devolver un valor. Si no existe ningún valor a devolver al no cumplirse la cláusula if, devolvemos una secuencia vacía con ‘else ()’, tal y como se muestra en el ejemplo anterior.

3.5. Cuantificadores existenciales:

XQuery soporta dos cuantificadores existenciales llamados “some” y “every”, de tal manera que nos permite definir consultas que devuelva algún elemento que satisfaga la condición (“some”) o consultas que devuelvan los elementos en los que todos sus nodos satisfagan la condición (“every”). Por ejemplo, la siguiente consulta devuelve los títulos de los libros en los que al menos uno de sus autores es W. Stevens.

```
for $b in doc("libros.xml")//libro
where some $a in $b/autor
     satisfies ($a/last="Stevens" and $a/first="W.")
return $b/titulo
```

El resultado de esta consulta se muestra a continuación

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix Environment</title>
```

La siguiente consulta devuelve todos los títulos de los libros en los que todos los autores de cada libro es W. Stevens.

```
for $b in doc("libros.xml")//libro
where every $a in $b/autor
satisfies ($a/last="Stevens" and $a/first="W.")
return $b/titulo
```

El resultado de esta consulta se muestra en el siguiente párrafo.

```
<titulo>TCP/IP Illustrated</titulo>
<titulo>Advanced Programming in the Unix Environment</titulo>
<titulo>The Economics of Technology and Content for Digital
TV</titulo>
```

El último título devuelto como resultado de la consulta es un libro que no tiene autores. Cuando un cuantificador universal se aplica sobre un nodo vacío, siempre devuelve cierto, como se ve en el ejemplo anterior.

Otro ejemplo. La siguiente consulta devuelve los títulos de los libros que mencionen “Unix” y “programing” en el mismo párrafo. Si el libro tiene más de un párrafo solo es necesario que aparezca en, al menos, uno de ellos. Esto lo indicamos con la palabra reservada “some” justo a continuación de where.

```
for $b in doc("bib.xml")//libro
where some $p in $b//parrafo satisfies
  (contains($p,"Unix") AND contains($p,"programing"))
return $b/title
```

Otro ejemplo. La siguiente consulta devuelve el título de todos los libros que mencionen “programing” en cada uno de los párrafos de los libros almacenados en “bib.xml”.

```
for $b in doc("bib.xml")//libro
where every $p in $b// parrafo satisfies
  contains($p,"programing")
return $b/title
```

Esta consulta es distinta de la anterior ya que no es suficiente que “programing” aparezca en al menos uno de los párrafos, sino que debe aparecer en todos los párrafos que existan. Esto lo indicamos con la palabra reservada “every” justo a continuación de “where”.

3.6. Operadores y funciones principales.

El conjunto de funciones y operadores soportado por XQuery 1.0 es el mismo conjunto de funciones y operadores utilizado en XPath 2.0 y XSLT 2.0.

XQuery soporta operadores y funciones matemáticas, de cadenas, para el tratamiento de expresiones regulares, comparaciones de fechas y horas, manipulación de nodos XML, manipulación de secuencias, comprobación y conversión de tipos y lógica booleana. Además permite definir funciones propias y funciones dependientes del entorno de ejecución del motor XQuery. Los operadores y funciones más importantes se muestran en la tabla 2 y se detallan a lo largo de este apartado.

Matemáticos:	+, -, *, div(*), idiv(*), mod.
Comparación:	=, !=, <, >, <=, >=, not()
Secuencia:	union (|), intersect, except
Redondeo:	floor(), ceiling(), round().
Funciones de agrupación:	count(), min(), max(), avg(), sum().
Funciones de cadena:	concat(), string-length(), starts-with(), ends-with(), substring(), upper-case(), lower-case(), string()
Uso general:	distinct-values(), empty(), exits()

Tabla 2. Principales operadores y funciones de XQuery

(*) La división se indica con el operador ‘div’ ya que el símbolo ‘/’ es necesario para indicar caminos. El operador ‘idiv’ es para divisiones con enteros en las que se ignora el resto.

El resultado de un operador aritmético en el que uno, o ambos, de los operandos sea una cadena vacía es una cadena vacía. Como regla general el funcionamiento de las cadenas vacías en XQuery es análogo al funcionamiento de los valores nulos en SQL.

El operador unión recibe dos secuencias de nodos y devuelve una secuencia con todos los nodos existentes en las dos secuencias originales. A continuación se muestra una consulta que usa el operador unión para obtener una lista ordenada de apellidos de todos los autores y editores:

```
for $l in distinct-values(doc("libros.xml")
     //(autor | editor)/apellido)
order by $l
return <apellidos>{ $l }</apellidos>
```

El resultado de esta consulta se muestra en el siguiente párrafo:

```
<apellidos>Abiteboul</apellidos>
<apellidos>Buneman</apellidos>
<apellidos>Gerbarg</apellidos>
<apellidos>Stevens</apellidos>
<apellidos>Suciu</apellidos>
```

El operador de intersección recibe dos secuencias de nodos como operandos y devuelve una secuencia conteniendo todos los nodos que aparezcan en ambos operandos.

El operador de sustracción (except) recibe dos secuencias de nodos como operandos y devuelve una secuencia conteniendo todos los nodos del primer operando que no aparezcan en el segundo operando. A continuación se muestra una consulta que usa el operador sustracción para obtener un nodo libro con todos sus nodos hijos salvo el nodo <precio>.

```
for $b in doc("libros.xml")//libro
where $b/titulo = "TCP/IP Illustrated"
return
<libro>
     { $b/@* }
     { $b/* except $b/precio }
</libro>
```

El resultado de esta consulta se muestra a continuación.

```
<libro year = "1994">
     <title>TCP/IP Illustrated</title>
     <author>
      <apellidos>Stevens</apellidos>
      <first>W.</first>
     </author>
     <publisher>Addison-Wesley</publisher>
</libro>
```

La función distinct-values() extrae los valores de una secuencia de nodos y crea una nueva secuencia con valores únicos, eliminando los nodos duplicados. Por ejemplo, la siguiente consulta devuelve todos los apellidos distintos de los autores.

```
for $l in distinct-values(doc("libros.xml")//autor/apellidos)
return <apellidos>{ $l }</apellidos>
```

El resultado de esta consulta se muestra en el siguiente párrafo.

```
<apellidos>Stevens</apellidos>
<apellidos>Abiteboul</apellidos>
<apellidos>Buneman</apellidos>
<apellidos>Suciu</apellidos>
```

La función empty() devuelve cierto cuando la expresión entre paréntesis está vacía. Por ejemplo, la siguiente consulta devuelve todos los nodos libro que tengan al menos un nodo autor.

```
for $b in doc("libros.xml")//libro
where not(empty($b/autor))
return $b
```

El resultado de esta consulta se muestra a continuación.

```
  <libro año="1994">
       <titulo>TCP/IP Illustrated</titulo>
       <autor>
               <apellido>Stevens</apellido>
               <nombre>W.</nombre>
       </autor>
       <editorial>Addison-Wesley</editorial>
       <precio> 65.95</precio>
   </libro>
```

```
    <libro año="1992">
        <titulo>Advan Programming for Unix environment</titulo>
        <autor>
                <apellido>Stevens</apellido>
                <nombre>W.</nombre>
        </autor>
        <editorial>Addison-Wesley</editorial>
        <precio>65.95</precio>
    </libro>

    <libro año="2000">
        <titulo>Data on the Web</titulo>
        <autor>
                <apellido>Abiteboul</apellido>
                <nombre>Serge</nombre>
        </autor>
        <autor>
                <apellido>Buneman</apellido>
                <nombre>Peter</nombre>
        </autor>
        <autor>
                <apellido>Suciu</apellido>
                <nombre>Dan</nombre>
        </autor>
        <editorial>Morgan Kaufmann editorials</editorial>
        <precio>39.95</precio>
    </libro>
```

La función opuesta a empty() es exists(), la cual devuelve cierto cuando una secuencia contiene, al menos, un elemento. Por ejemplo, como la consulta anterior tiene una cláusula where que comprueba una negación sobre empty(), podemos rescribirla usando la función exists() tal y como se muestra a continuación:

```
for $b in doc("libros.xml")//libro
where exists($b/autor)
return $b
```

El resultado de esta consulta es el mismo que el resultado de la consulta anterior.

3.7. Comentarios.

Los comentarios en XQuery, a diferencia de XML, van encerrados entre caras sonrientes, tal y como se muestra a continuación.

```
(: Esto es un comentario, y parece muy feliz :)
```

La sintaxis de comentario usada en XML no es aplicable a XQuery salvo que la consulta se escriba con la sintaxis XQueryX, como se verá más adelante.

3.8. XQueryX.

Como hemos visto, uno de los requerimientos de XQuery es que debe permitir utilizar diferentes sintaxis para redactar las consultas. XQueryX es un ejemplo de sintaxis alternativa para XQuery.

XQueryX es la especificación de una sintaxis alternativa para XQuery que permite definir una consulta XQuery mediante etiquetas XML. Con una sintaxis basada en XML es más sencillo analizar las consultas mediante el uso de herramientas estándares y generar y consultar los contenidos de la consulta. Esto es útil, por ejemplo, para optimizaciones a nivel de código fuente que dependen de la capacidad de inspeccionar de forma rápida y sencilla el código de una consulta.

Sin embargo, el borrador de XQueryX no ha sido actualizado desde su creación en el año 2.001, ni el grupo de trabajo de XQuery ha hecho ningún comentario al respecto sobre esta sintaxis, por lo que su futuro es bastante incierto.

Como ejemplo vamos a ver una sencilla consulta que devuelve la lista de todos los autores de libros. Con la sintaxis de XQuery esta consulta se escribiría como se muestra a continuación:

```
let $autores := /libro/autor
return
   <autores>
   {
      $autores
   }
   </autores>
```

A continuación se muestra la misma consulta escrita con la sintaxis XQueryX.

```
<q:query xmlns:q="http://www.w3.org/2001/06/XQueryx">
   <q:flwr>
      <q:letAssignment variable="$autores">
         <q:step axis="CHILD">
            <q:identifier/>
           <q:step axis="CHILD">
               <q:identifier>libro</q:identifier>
               <q:identifier>autor</q:identifier>
            </q:step>
         </q:step>
      </q:letAssignment>
      <q:return>
         <q:elementConstructor>
            <q:tagName>
               <q:identifier>autores</q:identifier>
            </q:tagName>
            <q:variable>$autores</q:variable>
         </q:elementConstructor>
```

```
         </q:return>
      </q:flwr>
   </q:query>
```

A continuación se muestra el resultado de ambas consultas.

```
<autores>
        <autor>
                <apellido>Stevens</apellido>
                <nombre>W.</nombre>
        </autor>
        <autor>
                <apellido>Stevens</apellido>
                <nombre>W.</nombre>
        </autor>
        <autor>
                <apellido>Abiteboul</apellido>
                <nombre>Serge</nombre>
        </autor>
        <autor>
                <apellido>Buneman</apellido>
                <nombre>Peter</nombre>
        </autor>
        <autor>
                <apellido>Suciu</apellido>
                <nombre>Dan</nombre>
        </autor>
 </autores>
```

3.9. Ejemplos de consultas

Como ya hemos visto, la consulta más sencilla posible en XQuery es una expresión en XPath, por ejemplo:

```
document("libros.xml")//libro[titulo="Ricotta Pie"]
```

La expresión anterior devuelve todos los nodos <libro> que tengan un hijo <titulo> con el valor “Ricotta Pie” del conjunto de datos almacenado en el archivo “libros.xml”. El orden de los libros será, por defecto, el mismo orden en que aparecen en el documento “libros.xml”.

Partiendo del conjunto de datos detallados en el apartado 2.2, vamos a escribir una serie de consultas y a mostrar cuales serian los resultados obtenidos.

Escribir una consulta que obtenga el nombre y el año de todos los libros publicados por Addison-Wesley después de 1991. El código de la consulta será:

```
<bib>
 {
  for $b in doc("libros.xml")/bib/libro
  where $b/editorial = "Addison-Wesley" and $b/@año > 1991
  return
    <libro año="{ $b/@año }">
     { $b/titulo }
    </libro>
 }
</bib>
```

El resultado de esta consulta se muestra a continuación.

```
<bib>
    <libro año="1994">
        <titulo>TCP/IP Illustrated</titulo>
    </libro>
    <libro año="1992">
        <titulo>Advanced Programming in the Unix environment
        </titulo>
    </libro>
</bib>
```

Escribir una consulta que obtenga el título de los libros cuyo precio esté por debajo de 50.00€.

```
for $b in doc("libros.xml")//libro
where $b/precio < 50.00
return $b/titulo
```

El resultado de esta consulta se muestra a continuación.

```
<titulo>Data on the Web</titulo>
```

Escribir una consulta que, por cada libro almacenado en el archivo "libros.xml" devuelva el título del libro, el precio con que consta dicho libro en el archivo "libros.xml" y el precio con que consta ese libro en el archivo "comentarios.xml".

```
<libros-con-precios>
  {
    for $b in doc("libros.xml")//libro,
        $a in doc("comentarios.xml")//entry
    where $b/titulo = $a/titulo
```

```
    return
        <libro-con-precios>
            { $b/titulo }
            <precio-btienda2>{ $a/precio/text() }
             </precio-btienda2>
            <precio-btienda1>{ $b/precio/text() }
             </precio-btienda1>
        </libro-con-precios>
  }
</libros-con-precios>
```

Los resultados de esta consulta se muestran a continuación.

```
<libros-con-precios>
    <libro-con-precios>
        <titulo>TCP/IP Illustrated</titulo>
        <precio-btienda2>65.95</precio-btienda2>
        <precio-btienda1>65.95</precio-btienda1>
    </libro-con-precios>
    <libro-con-precios>
        <titulo>Advanced Programming in the Unix
environment</titulo>
        <precio-btienda2>65.95</precio-btienda2>
        <precio-btienda1>65.95</precio-btienda1>
    </libro-con-precios>
    <libro-con-precios>
        <titulo>Data on the Web</titulo>
        <precio-btienda2>34.95</precio-btienda2>
        <precio-btienda1>39.95</precio-btienda1>
    </libro-con-precios>
</libros-con-precios>
```

Escribir una consulta que, por cada libro con autores, devuelva el título del libro y sus autores. Si el libro no tiene autores pero sí editor, la consulta devolverá el título del libro y la afiliación del editor.

```
<bib>
{
        for $b in doc("libros.xml")//libro[autor]
        return
            <libro>
                { $b/titulo }
                { $b/autor }
            </libro>
}
{
        for $b in doc("libros.xml")//libro[editor]
        return
          <referencia>
```

```
            { $b/titulo }
            {$b/editor/afiliacion}
          </referencia>
}
</bib>
```

Los resultados de esta consulta se muestran a continuación.

```
<bib>
    <libro>
        <titulo>TCP/IP Illustrated</titulo>
        <autor>
            <apellidos>Stevens</apellidos>
            <nombre>W.</nombre>
        </autor>
    </libro>
    <libro>
        <titulo>Advanced Programming in the Unix
environment</titulo>
        <autor>
            <apellidos>Stevens</apellidos>
            <nombre>W.</nombre>
        </autor>
    </libro>
    <libro>
        <titulo>Data on the Web</titulo>
        <autor>
            <apellidos>Abiteboul</apellidos>
            <nombre>Serge</nombre>
        </autor>
        <autor>
            <apellidos>Buneman</apellidos>
            <nombre>Peter</nombre>
        </autor>
        <autor>
            <apellidos>Suciu</apellidos>
            <nombre>Dan</nombre>
        </autor>
    </libro>
    <referencia>
        <titulo>The Economics of Technology and Content for
Digital TV</titulo>
        <afiliacion>CITI</afiliacion>
    </referencia>
</bib>
```

Mostrar los pares de títulos que sean distintos pero tengan el mismo autor o grupo de autores. Hay que tener en cuenta que el orden de aparición de los autores puede variar de un libro a otro.

```
<bib>
{
    for $libro1 in doc("libros.xml")//libro,
        $libro2 in doc("libros.xml")//libro
    let $aut1 := for $a in $libro1/autor
                 order by $a/apellidos, $a/nombre
                 return $a
    let $aut2 := for $a in $libro2/autor
                 order by $a/apellidos, $a/nombre
                 return $a
    where $libro1 << $libro2
    and not($libro1/titulo = $libro2/titulo)
    and deep-equal($aut1, $aut2)
    return
        <libro-par>
            { $libro1/titulo }
            { $libro2/titulo }
        </libro-par>
}
</bib>
```

Los resultados de esta consulta se muestran a continuación.

```
<bib>
    <libro-par>
        <titulo>TCP/IP Illustrated</titulo>
        <titulo>Advanced Programming in the Unix
environment</titulo>
    </libro-par>
</bib>
```

Esta consulta usa la función deep-equal() encargada de comparar secuencias de nodos. Para la función deep-equal() dos consultas son iguales si todos los nodos de la primera secuencia aparecen en la segunda secuencia en la misma posición que en la primera secuencia.

También es posible utilizar XQuery para trasformar datos XML en otros formatos, como HTML, convirtiéndose XQuery en una alternativa más sencilla y rápida de usar que XSLT. A continuación, como ejemplo, se muestra una consulta que crea una tabla HTML con los títulos de todos los libros contenidos en el archivo “libros.xml”.

```
<html> <head> <title> </title>
<body> <table>
{
     for $b in doc("libros.xml")/bib/libro
     return
     <tr> <td> <I> { string( $b/titulo ) } </I> </td> </tr>
```

```
}
</table> </body>
</head>  </html>
```

El resultado de la consulta anterior se muestra a continuación:

```
<html><head><title>  </title>
<body>
<table>
<tr><td><I>TCP/IP Illustrated</I></td></tr>
<tr><td><I>Advan Programming for Unix environment</I></td></tr>
<tr><td><I>Data on the Web</I></td></tr>
<tr><td><I>Economics of Technology for Digital TV</I></td></tr>
</table></body>
</head></html>
```

A continuación, en la figura 4, se muestra como se visualizaría la página anterior en un navegador web:

Figura 4. Resultado de la conversión de un conjunto de datos XML en HTML.

4. XQuery y Java.

A lo largo de este apartado se analizan las características de un motor XQuery open-source escrito en la plataforma Java y vamos a ver como utilizarlo desde nuestras propias aplicaciones. El motor elegido es XQEngine[7], aunque en el punto 4.4 se comentan brevemente otros motores open-source existentes para la plataforma Java.

4.1. XQEngine.

XQEngine es un motor GPL dedicado al indexado y búsqueda de información en ficheros XML cuya sintaxis de consulta es una implementación de XQuery. XQEngine es un componente muy compacto, con un tamaño aproximado de 360 KBs, y por lo tanto, fácilmente integrable dentro de nuestras aplicaciones. La versión actual de esta herramienta es la 0.63.

XQEngine se distribuye con una sencilla aplicación de ejemplo "SampleApp.java" que, además de mostrar con un caso práctico como utilizar este componente en nuestros desarrollos, contiene un sencillo interfaz gráfico que permite ejecutar directamente consultas XQuery y obtener sus resultados.

En los siguientes apartados se describe esta aplicación, la cual es una valiosa ayuda a la hora de escribir y depurar nuestras consultas y, a continuación, se analiza como incluir XQEngine en nuestros programas.

4.2. Ejecutando consultas con XQEngine.

Para ejecutar la aplicación de ejemplo, simplemente hay que escribir “java SampleApp” y obtener así la interfaz de la aplicación mediante la cual se puede escribir y ejecutar consultas XQuery y obtener los resultados. Esta interfaz se muestra en la figura 5.

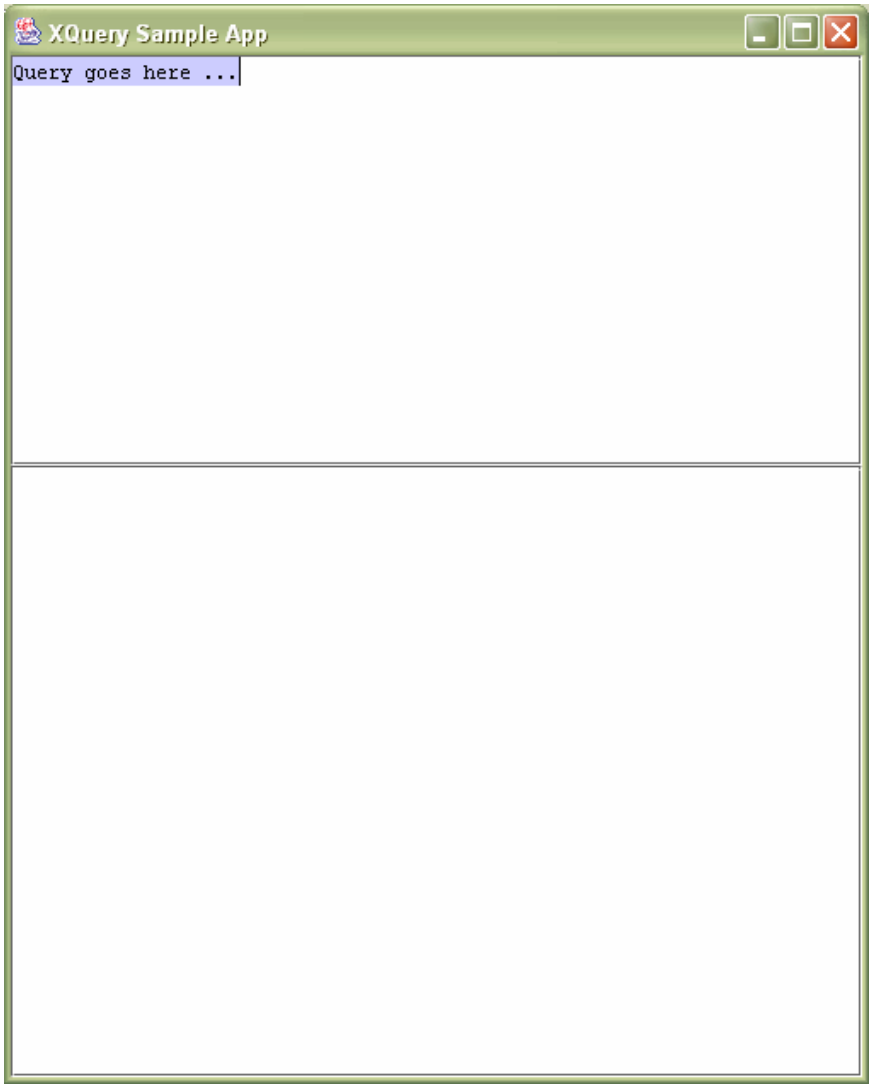

Figura 5. Interfaz de SampleApp

"SampleApp" carga e indexa un conjunto de archivos XML con datos de ejemplo para poder ejecutar consultas sobre ellos. Estos ficheros son: "bib.xml", "reviews.xml", "book.xml", "report1.xml", "bib_2.xml", "nsTest_1.xml", "nsTest_2.xml", "nsTest_3.xml" y "Bug_10jan04_1.xml" y también se distribuyen junto con XQEngine.

La aplicación no ofrece ninguna forma de indicar que archivos XML queremos que se carguen e indexen al inicio. La única manera de agregar nuestros propios archivos es modificar directamente el código de la clase "SampleApp.java". Concretamente, en el método "run()" de dicha clase se indica el nombre de los archivos XML que debe cargar e indexar SampleApp. Para añadir el archivo que venimos usando como base para los ejemplos, "libros.xml", suponiendo que se encuentra en la misma carpeta que los archivos anteriores, añadimos a la clase SampleApp la siguiente línea.

```
public void run()
{
      […..]
      m_engine.setDocument(“libros.xml”);
      […..]
}
```

Ahora, al ejecutar la aplicación, incluirá en su índice de archivos al archivo “libros.xml”.

Para ejecutar una consulta hay que redactarla en la mitad superior de la ventana. Una vez completada la consulta es necesario pulsar dos veces la tecla “Enter” para que la aplicación la ejecute y muestre los resultados en la mitad inferior de la ventana. En la figura 6 se muestra un ejemplo de consulta, en la mitad superior, y de sus resultados, en la mitad inferior.

Figura 6. Ejemplo de consulta y resultados.

Tanto la mitad de edición de la consulta como la mitad que muestra los resultados soportan copiar (CTRL+C) y pegar (CTRL+V), pero no deshacer.

Si la consulta no está bien escrita, en la mitad inferior de la pantalla de la aplicación obtendremos un mensaje de: “InvalidQueryException”. Un mensaje de error más descriptivo se envía a la salida estándar. Por ejemplo, en la figura 7 se muestra el mensaje de error obtenido al intentar ejecutar la consulta anterior sin escribir la cláusula for.

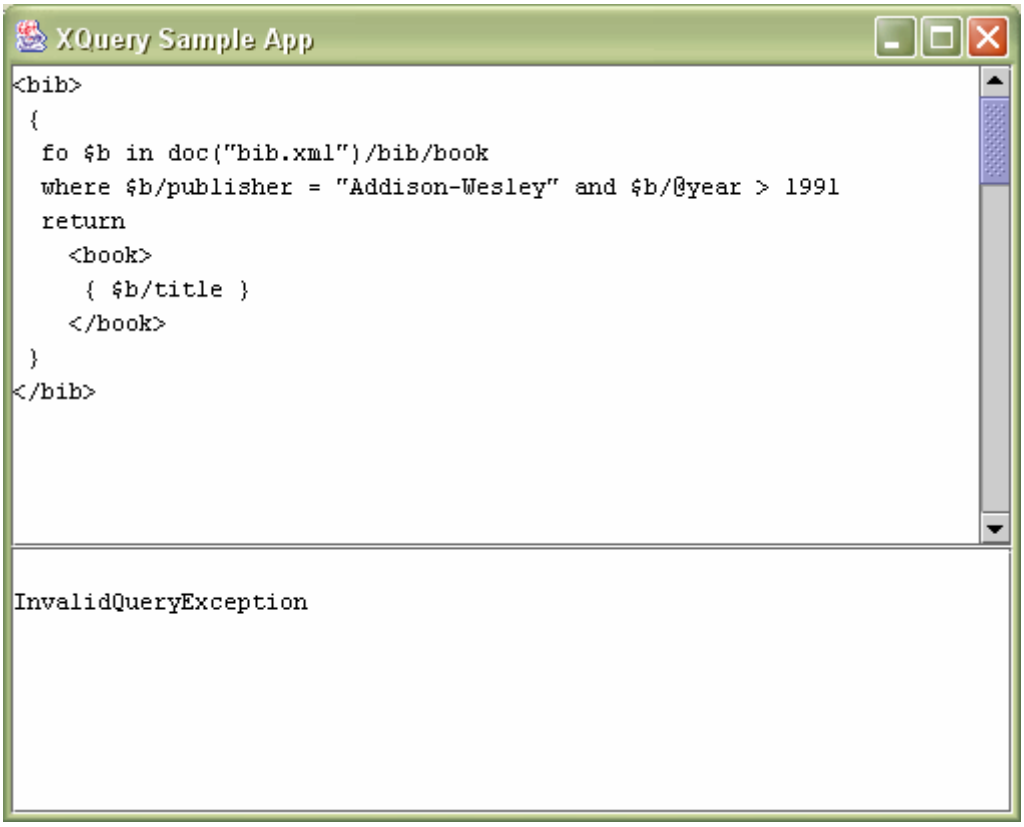

Figura 7. Consulta errónea.

Y en la salida estándar obtenemos el siguiente mensaje, bastante más descriptivo que el anterior.

```
com.fatdog.xmlEngine.exceptions.InvalidQueryException:
Encountered "$" at line 3, column 6
.
Was expecting one of:
    "or" ...
    "and" ...
    "to" ...
    "=" ...
    "is" ...
    "!=" ...
    "isnot" ...
    "<=" ...
    "<<" ...
    ">=" ...
    ">>" ...
    "eq" ...
    "ne" ...
    "gt" ...
    "ge" ...
    "lt" ...
    "le" ...
    "<" ...
```

```
    ">" ...
    "[" ...
    "," ...
    "}" ...
    "/" ...
    "//" ...
    "(" ...
```

Ni esta aplicación ni XQEngine soportan la sintaxis XQueryX para la redacción de las consultas.

4.3. Embebiendo XQEngine en una aplicación Java.

Para mostrar como utilizar XQEngine como componente en nuestras aplicaciones Java vamos a desarrollar un sencillo ejemplo. Este ejemplo ejecutará una consulta sobre el archivo “libros.xml” y mostrará el resultado de la consulta por pantalla.

Para implementar el ejemplo vamos a crear una clase llamada “EjemploXQEngine” que contenga un único método “main” que contendrá todo el código necesario para el ejemplo. El código completo de la clase se muestra a continuación.

```
import com.fatdog.xmlEngine.ResultList;
import com.fatdog.xmlEngine.XQEngine;
import com.fatdog.xmlEngine.exceptions.*;

import javax.xml.parsers.*;

import org.xml.sax.XMLReader;


public class EjemploXQEngine
{

  public static void main(String[] args)
  {
    String query = "<bib> { "
                 + "  for $b in doc(\"libros.xml\")/bib/libro"
                 + "  where $b/editorial = \"Addison-Wesley\"”
                 + “ and $b/@año > 1991 "
                 + "  return "
                 + "  <libro> { $b/titulo } </libro>"
                 + " } </bib> ";

    XQEngine engine;

    engine = new XQEngine();

    SAXParserFactory spf = SAXParserFactory.newInstance();
    try
```

```
    {
      SAXParser parser = spf.newSAXParser();
      XMLReader reader = parser.getXMLReader();
      engine.setXMLReader( reader );

      engine.setDocument( "libros.xml" );

      ResultList results = engine.setQuery( query );

      System.out.println( results.toString() );
      System.out.println( "----------------------" );
      System.out.println( results.emitXml() );
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }

  }
};
```

En los siguientes párrafos se detalla el funcionamiento del código anterior.

En primer lugar, crear una instancia del motor XQEngine es tan sencillo como declarar una variable de tipo XQEngine y crear un nuevo objeto de ese tipo. Sin embargo, antes de que el motor de XQEngine, pueda funcionar necesita dos elementos: el primero es un parser SAX para leer los documentos XML. En este ejemplo se ha optado por usar el parser SAX de SUN por ser el más difundido ya que viene con la distribución de J2EE, pero otros parsers, como el que se incluye en el componente Xerces[19] de Apache, son perfectamente válidos.

El segundo elemento que XQEngine necesita es el conjunto de archivos XML que contienen los datos sobre los que se van a ejecutar las consultas. XQEngine necesita conocer previamente estos archivos para poder indexarlos antes de realizar ninguna consulta. En este ejemplo solo utilizamos un archivo "libros.xml" pero, en principio, XQEngine no impone ninguna limitación sobre el número de archivos que podamos cargar e indexar.

Una vez obtenido un objeto XQEngine, es posible realizar tantas consultas como se desee. Al ejecutar una consulta, XQEngine devuelve el resultado en un objeto de tipo ResultList. El método de este objeto toString() devuelve las estadísticas de la consulta y el método emitXml() devuelve el resultado como un String .

El resultado de la ejecución de la clase "EjemploXQEngine" se muestran a continuación.

```
ResultList.toString():
NumDocuments   = 1
NumTotalItems  = 1
NumValidItems  = 1

Document id   = -1 ["QUERY_DOCUMENT"]
NumTotalItems = 1
```

```
NumValidItems = 1
[0][0] <bib> [parent=-1]

----------------------

<bib><libro><titulo>TCP/IP
Illustrated</titulo></libro><libro><titulo>Advan Programming fo
r Unix environment</titulo></libro></bib>
```

4.4. Otros motores XQuery open-source.

En este apartado se enumeran los motores XQuery open-source más relevantes y sus características principales.

Qexo:

Qexo[8] es un motor XQuery escrito en Java y con licencia GPL que se distribuye integrado dentro del paquete Kawa.

Saxon:

Saxon[10] es otro motor XQuery escrito en Java que se distribuye en dos paquetes, uno de ellos, Saxon-B es open-source bajo licencia GPL y contiene una implementación básica de XSLT 2.0 y XQuery, mientras que el segundo paquete, Saxon-SA, contiene un procesador completo XSLT y XQuery pero tiene licencia propietaria, aunque está disponible una licencia de evaluación de 30 días.

Qizx/open:

Qizx/open[9] es una implementación Java de la última versión de las especificaciones XQuery. Implementa todas las características del lenguaje excepto la importación y validación de XML-Schemas. Actualmente este es el motor con licencia GPL más completo que existe.

4.5. Otras herramientas relacionadas con XQuery.

En este apartado se describen brevemente otras herramientas relacionadas con XQuery.

Xquark Bridge:

XQuark Bridge[16] es una herramienta que permite importar y exportar datos a bases de datos relacionales utilizando XML. De este modo, XQuark ofrece la posibilidad de manejar estructuras XML y realizar la transformación a objetos de la base de datos, siendo capaz también de realizar el paso contrario. Además de todo esto XQuark es capaz de respetar todas las restricciones de integridad y transformar las relaciones implícitas en los documentos XML en relaciones explícitas en la base de datos y todo ello con un buen rendimiento.

XQuark-Bridge soporta también la consulta y manipulación de los datos en formato XML utilizando el lenguaje XQuery.

XQuark-Bridge soporta MySQL, Oracle, SQLServer y Sybase, tiene una licencia LGPL.

BumbleBee:

BumbleBee[7] es un entorno de prueba automático creado con el fin de evaluar motores de XQuery y validar consultas escritas en sintaxis XQuery. BumbleBee permite evaluar que grado de satisfacción de los borradores del estándar y especificaciones satisface un motor XQuery y comprobar si una versión más moderna de nuestro motor permite seguir ejecutando nuestras consultas.

BumbleBee se distribuye con un conjunto de pruebas ya preparadas y, además, ofrece un entorno sencillo para redactar y ejecutar nuestras propias pruebas.

Actualmente soporta los motores XQuery open-source: Qexo[8], Qizx/open[9] y Saxon[10] y los motores XQuery propietarios: Cerisent[11], Ipedo[12], IPSI-XQ[13] y X-Hive[14].

BumbleBee es software propietario aunque está disponible para descarga una versión de demostración completamente funcional durante 30 días.

5. Conclusiones

XQuery es en la actualidad, y a pesar de estar aún en fase de borrador, una tecnología emergente con grandes expectativas en el mundo de la programación y del tratamiento y manipulación de información como lo demuestra el número de empresas que están apostando por desarrollar implementaciones de motores de consulta basados en XQuery.

Sus principales aplicaciones se pueden resumir en tres grandes grupos.

En primer lugar, recuperar información a partir de conjuntos de datos XML. Hemos expuesto en este trabajo, a través de los ejemplos, como, gracias a un lenguaje sencillo, potente y flexible, es posible recorrer los nodos de un conjunto de datos XML, filtrando aquellos que nos interesen y transformándolos para mostrar la información deseada con la estructura adecuada.

En segundo lugar, transformar unas estructuras de datos XML en otras estructuras que organicen la información de forma diferente. Por ejemplo, es sencillo y fácil obtener a partir de la jerarquía de datos almacenada en el archivo "libros.xml", otra jerarquía de datos donde se almacene una lista con todos los autores y los libros que ha escrito cada autor, tal y como se muestra en la figura 8.

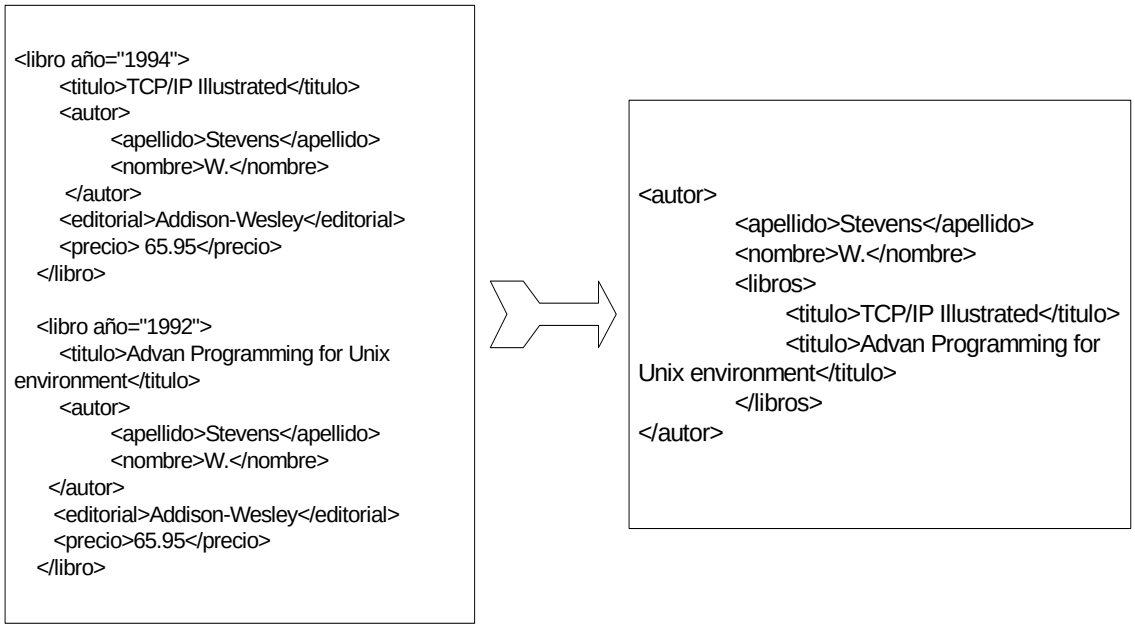

Figura 8. Transformación de una jerarquía en otra mediante XQuery.

Y en tercer lugar, ofrecer una alternativa a XSLT para realizar transformaciones de datos en XML a otro tipo de representaciones, como HTML o PDF. Ya hemos visto en uno de los ejemplos del punto 3.9 lo sencillo que resulta escribir una consulta XQuery para que genere código HMTL.

Actualmente también existe un número variado de implementaciones open-source, lo que nos ofrece libertad de elección, y están lo suficientemente maduras como para poder ser incorporadas con garantías a cualquier proyecto de programación, con la condición de que dicho proyecto mantenga la licencia GPL y siga siendo open-source.

6. Referencias y enlaces.

[1] Marchal, Benoît. *XML by Example*. QUE. 2.001 West 103rd Street, Indianapolis, Indiana 46290.

[2] McLaughlin, Brett. *Java and XML Data Binding*. 2.002. O'Reilly.

[3] Robert Gardner, John; Rendon, Zarella L. *XSLT and XPATH: A Guide to XML Transformations*. 2001. Prentice Hall PTR

[4] Katz, Howard; Chamberlin, Don, et-al. *XQuery from the Experts: A Guide to the W3C XML Query Language. 2.003*. Addison Wesley

[5] Brundage, Michael. *XQuery: The XML Query Language*. 2.004. Addison Wesley

[6] XQEngine. http://xqengine.sourceforge.net/

[7] BumbleBee. http://www.XQuery.com/bumblebee/

[8] Qexo. http://www.gnu.org/software/qexo/

[9] Qizx/open. http://www.xfra.net/qizxopen/

[10] Saxon. http://saxon.sourceforge.net/

[11]Cerisent. http://www.cerisent.com/

[12] Ipedo. http://www.ipedo.com/

[13] IPSI-XQ. http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_e.html

[14] X-Hive. http://xhive.com/

[15] JAXB, API Java de Sun para XML Binding. http://java.sun.com/xml/jaxb/

[16] Xquark. http://xquark.forge.objectweb.org/index.html

[17] SAX project website. http://sax.sourceforge.net/

[18] DOM home page. http://www.w3.org/DOM/

[19] Apache's XML parser. http://xml.apache.org/xerces2-j/

[20] W3C Xquery. http://www.w3.org/TR/XQuery/

Apéndice I. Parses SAX y DOM

En el contexto de este artículo, un parser es una herramienta que procesa una colección de datos organizados en un árbol de etiquetas que sigue la sintaxis XML. Estos parsers son construidos como librerías o componentes para ser utilizarlos en programas que manejen colecciones de datos en XML. A continuación se describen brevemente las características principales de un parser SAX y un parser DOM.

SAX.

SAX[1][17] es un parser dirigido por eventos. Esto significa que cada vez que aparece un elemento XML, como la apertura o cierre de una nueva etiqueta, el parser genera un evento. La programación con un parser SAX consiste en redefinir las funciones de los eventos que nos interese controlar y, para cada evento, comprobar el valor de la etiqueta y actuar en consecuencia.

Las ventajas principales de los parsers SAX son que no necesita una gran cantidad de memoria, ya que solo carga la sección de datos XML con la que está trabajando, y que es rápido y sencillo escribir parsers SAX simples. Sus principales inconvenientes son que depende completamente de la estructura del documento, si cambia la estructura es necesario modificar el código, no es reutilizable de un documento a oro a menos que compartan las mismas etiquetas y estructura de datos y, como se procesa a medida que se lee, no se tiene información de lo que viene a continuación y solo se pueden escribir acciones simples.

DOM.

DOM[1][18], a diferencia del anterior, carga todo el conjunto de datos XML en memoria en la forma de un árbol n-ario. Cualquier operación sobre el conjunto de datos se realiza sobre el árbol que se almacena en memoria.

Como ventajas, es más rápido que SAX ya que trabaja con los datos en memoria y no en el archivo. Además, al tener toda la estructura de datos disponible, es posible obtener información de lo que viene a continuación, retroceder, o hacer varias búsquedas.

Como inconvenientes, es más complejo trabajar con un parser DOM ya que es necesario escribir un recorrido sobre un árbol n-ario, necesita más cantidad de memoria que SAX y es más lento, aunque si es necesario recorrer más de un vez el documento o una parte de él, DOM gana en velocidad.

Apéndice II. XML Binding

Básicamente la idea que se esconde tras el término binding (vincular o vinculación)[2] es tomar un documento que contiene un conjunto de datos expresados en XML y convertirlo en una instancia de un objeto, que puede estar formado a su vez por varios objetos. El proceso de binding está basado en cuatro conceptos que estudiaremos a continuación. Estos conceptos son:

- Generación de código fuente
- Unmarshalling
- Marshalling
- Binding schemas (esquemas de vinculación)

Las herramientas de binding, como primer paso, permiten crear automáticamente un conjunto de clases a partir de los DTDs o XML-Schemas del conjunto de datos. Estas clases serán instanciadas y sus objetos almacenarán los datos del documento XML por la herramienta vinculadora.

Una vez que se han generado las clases ya se puede convertir el documento XML en un conjunto de objetos. Esta operación se conoce con el nombre de unmarshalling. La operación opuesta se conoce con el nombre de marshalling y consiste en almacenar en un documento XML, la información contenida en los atributos de un objeto o conjunto de objetos.

Aunque no es necesario, las herramientas de binding permiten definir binding schemas (esquemas de vinculación), los cuales permiten indicar con mayor precisión las conversiones de tipo o transformaciones de nombres.

Las herramientas de binding tienen la ventaja de que son sencillas y rápidas de utilizar, generan automáticamente la estructura de clases necesaria para cargar en memoria la estructura de datos XML, y nos permite abstraernos de conceptos de bajo nivel, como la lectura y procesamiento del archivo XML. También tienen el inconveniente de que es necesario disponer en DTD o el XML-Schema de la estructura de datos para poder generar el código fuente. Además, el proceso de binding solo realiza la carga en memoria de la estructura de datos XML, al igual que DOM pero, a diferencia de este, organizado como un conjunto de objetos en vez de un árbol n-ario, por lo que es necesario escribir procedimientos de búsqueda de la información dependientes de dicha estructura de objetos.

JAXB[15] es el nombre del API oficial de SUN para binding de archivos XML. Ya existe una implementación de referencia de este API suministrado por la propia Sun. Algunas de las características principales de esta API se detallan a continuación.

- Soporta binding a partir de DTDs y XML Schema.
- Permite personalización del binding.
- Estandariza las interfaces para las clases generadas.
- Soporta validación de documentos.
- Soporta validación de árboles XML.

A continuación, en la tabla 3, se muestran los enlaces a varias herramientas de binding para Java.

Castor	http://www.castor.org/
Xgen	http://www.commerceone.com/developers/docsoapxdk
Breeze	http://www.breezefactor.com/
Zeus	http://zeus.objectweb.org/
XMLBeans	http://xml.apache.org/xmlbeans/

Tabla 3. Herramientas de binding para plataforma Java.

Apéndice III. Xpath.

XPath es un lenguaje de expresión utilizado para referenciar nodos de información en un conjunto de datos XML. XQuery utiliza este lenguaje para las cláusulas for y let de una consulta.

Actualmente tanto XQuery 1.0 como XPath 2.0 están en desarrollo por el mismo grupo de trabajo de la W3C, dado que XQuery hace un uso intensivo de XPath.

Las expresiones XPath tienen un comportamiento similar a las expresiones regulares aplicadas al contexto de un conjunto de datos en XML y, más concretamente, al un conjunto de nodos XML en vez de un conjunto de caracteres.

XPath es muy fácil de entender mediante ejemplos. A continuación mostramos algunas expresiones básicas y lo que se obtiene a partir de ellas:

```
/html/body/h1
```

Selecciona todos los nodos <h1> que son hijos de un nodo <body> que, a su vez, es hijo de un nodo <html> que es el nodo raíz del árbol XML. El resultado será todos los nodos <h1> que cumplan la anterior condición.

```
//h1
```

Selecciona todos los nodos <h1> que aparezcan en cualquier posición del árbol XML. La doble barra indica cualquier profundidad.

```
count(//libro)
```

Devuelve el número de nodos <libro> que aparecen en el documento en cualquier posición.

```
//libro[autor = "Hunter"]
```

Devuelve todos los nodos <libro> que aparezcan en el documento en cualquier posición y que tengan un nodo hijo <autor> con el valor "Hunter". Los corchetes indican un filtro para seleccionar los resultados que cumplan una determinada condición.

```
//libro[@año > 1999]
```

Devuelve todos los nodos <libro> que tengan un atributo "año" con un valor superior a 1999. La arroba indica que "año" no es un hijo (una etiqueta) sino un atributo de la etiqueta libro.

```
//libro[@paginas]
```

Devuelve todos los nodos <libro> que tengan un atributo “paginas”, independientemente del valor de ese atributo.

```
//libro/@paginas
```

Devuelve todos los atributos pagina de los nodos <libro>.

```
(i | b)
```

Devuelve todos los nodos <i> o todos los nodos <b> que encuentre en el nodo contexto. Por defecto el nodo contexto es el nodo raíz del documento.

XPath soporta el uso de predicados, los cuales son condiciones booleanas que permiten seleccionar un subconjunto de los nodos referenciados. La expresión se evalúa por cada nodo encontrado y según sea cierta o falsa, el nodo se selecciona o se descarta. A continuación se muestra un ejemplo:

```
(//servlet | //servlet-mapping) [servlet-name = $servlet]
```

Devuelve todos los nodos <servlet> o <servlet-maping> que tengan un hijo llamado <servelt-name> cuyo valor coincida con el valor de la variable $servlet.

Si el predicado contiene un literal numérico, se trata como si fuera un índice, tal y como muestra el siguiente ejemplo:

```
doc("libros.xml")/bib/libro/autor[1]
```

La expresión anterior devuelve solo el primero nodo autor que encuentre para cada nodo libro.

```
(doc("libros.xml")/bib/libro/autor)[1]
```

La expresión anterior es igual a la que acabamos de ver, excepto por los paréntesis. Estos paréntesis fuerzan a que se evalúe primero el recorrido por los nodos

autor de cada libro, y después se aplique el predicado, por lo que el resultado de esta expresión será el primer autor que aparezca en el archivo “libros.xml”.

```
//key[. = "Tiempo total"]
```

Devuelve todos los nodos <key> que tengan un valor de "Tiempo total.". El carácter "." representa el nodo contexto, lo cual tiene una función similar al operador "this" en lenguajes como C++ o Java.

```
(//key)[1]/texto
```

Devuelve los nodos <texto> del primer nodo <key> del documento.

Apéndice IV. Conjunto de documentos que describen XQuery y XPath.

A continuación se detalla el conjunto de documentos que describen y definen XQuery en su totalidad, junto con la URL donde se encuentra cada documento.

XML Query Requirements.

Este el documento principal del grupo de trabajo y el que contiene el conjunto de requerimientos de XQuery
http://www.w3.org/TR/xmlquery-req

XML Query Use Cases.

Este documento contiene un conjunto de escenarios reales y y varias consultas XQuery para cada uno de esos escenarios.
http://www.w3.org/TR/xmlquery-use-cases

XQuery 1.0: An XML Query Language.

Este es el documento principal del proyecto, el cual presenta el lenguaje y da una perspectiva de todos sus aspectos.
http://www.w3.org/TR/XQuery/

XQuery 1.0 and XPath 2.0 Data Model.

Este documento describe una extensión de conjunto de modelo de datos de XML.
http://www.w3.org/TR/query-datamodel/

XQuery 1.0 and XPath 2.0 Formal Semantics.

Este documento contiene la definición formal algebraica del lenguaje.
http://www.w3.org/TR/query-semantics/

XML Syntax for XQuery 1.0 (XQueryX).

Este documento describe una sintaxis alternativa para poder escribir las consultas en XML.
http://www.w3.org/TR/XQueryx

XQuery 1.0 and XPath 2.0 Functions and Operators Version 1.0.

Este documento detalla las funciones y operadores básicos soportados por XQuery y XPath.
http://www.w3.org/TR/XQuery-operators/

XML Path Language (XPath) 2.0.

Este documento contiene la documentación sobre XPath.
http://www.w3.org/TR/xpath20/

XSLT 2.0 and XQuery 1.0 Serialization.

Este documento expone una primera aproximación a las consideraciones involucradas en la serialización del modelo de datos usado en XQuery y XPath.

http://www.w3.org/TR/xslt-XQuery-serialization/

XML Query and XPath Full-Text Requirements.

Este documento contiene la descripción de los requisitos que una implementación completa debe ser capaz de cumplir.
http://www.w3.org/TR/XQuery-full-text-requirements/

XML Query and XPath Full-Text Use Cases.

Este documento detalla un conjunto de escenarios tomados del mundo real que una especificación completa debe ser capaz de gestionar.
http://www.w3.org/TR/xmlquery-full-text-use-cases/