Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Tesis de Licenciatura

"CX-XML: Un Lenguaje de consulta para XML"

Alumnos
Ilyef, Patricio LU 121/91
Prividera, Alejandro LU 731/89

Director
Alejandro Vaisman

INDICE

Abstract

En este trabajo presentamos un nuevo lenguaje de consulta que hemos denominado CXXML. Este lenguaje permite recuperar información de documentos XML con una sintaxis similar a SQL.
Este lenguaje de consulta está basado en el álgebra estándar para XML -versión del 15 de Febrero de 2001- propuesto por el World Wide Web Consortium (W3C) (www.w3.org) permitiendo una clara definición semántica del lenguaje y facilitando el proceso de optimización. El lenguaje propuesto es más poderoso que otros lenguajes de manipulación de datos semiestructurados (Lorel, UnQL y XQL) ya que permite, entre otras funcionalidades, realizar operaciones de ordenamiento y agrupación como así también proyección de varios elementos y/o atributos.

Capítulo 1

Introducción

Hoy en día la WEB provee un estándar simple y universal para el intercambio de información. El principio central es descomponer la información en unidades que puedan ser nombradas y transmitidas. Esta unidad de información es normalmente un archivo que tiene una estructura y que es creado por un usuario y compartido con otros definiendo su nombre en la forma de URL.
El éxito de la WEB es derivado del desarrollo de HTML (HyperText Marckup Language) [ABS00], una forma de estructurar texto para presentaciones visuales. HTML describe tanto el diseño y el formato del texto como las referencias a otros documentos mediante el uso de hipervínculos.
El hecho de compartir información, desde el punto de vista de las bases de datos, pasa por definir lenguajes de consulta para acceder a los datos y por definir los mecanismos necesarios para controlar la concurrencia en los accesos y mantener la integridad de los mismos.
Como ejemplo del intercambio de información en la WEB [ABS00], supongamos una organización que publica datos financieros, los cuales se encuentran en alguna base de datos relacional. Las páginas WEB que muestran esta información lo hacen mediante comandos SQL desplegando el resultado en páginas HTML.
Por otro lado, tenemos otra organización que quiere obtener un análisis financiero a partir de los datos publicados pero sólo tiene acceso a las páginas HTML. La única posibilidad que tienen es escribir un software que parsee cada una de las páginas y las convierta en una estructura adecuada para el análisis. Esto trae aparejado dos problemas: si se cambia el formato o la posición de los datos puede llegar a fallar el programa de parseo; y lo que es peor, para obtener toda la información el software deberá realizar repetidos requerimientos a las páginas HTML.

El lenguaje **XML** (eXtensible Markup Language) es el primer paso hacia la convergencia de estas dos vistas sobre estructuras de información [ABS00]. Es el nuevo estándar adoptado por el World Wide Web Consortium (W3C) que complementa a HTML para el intercambio de datos en la WEB.
A diferencia de lo que hace HTML, el propósito principal de XML es representar datos *semi-estructurados*, no describir formatos de texto. Como cada vez más empresas eligen dar acceso a sus bases de datos e intercambiar información con organizaciones afines a sus negocios, XML se está convirtiendo en una herramienta adecuada para lograrlo Deutsch *et al*[DFFLS98].
Un documento XML es una secuencia de *elementos*, cada uno consistente en texto libre y/u otros elementos encerrados entre *tags*. Un tag está formado por los caracteres "<" y ">" y el texto para identificarlo, por ej: <persona>. Los tags deben equipararse, es decir <persona> debe tener un </persona>, y anidar adecuadamente. Un documento XML con esas propiedades se llama documento *bien formado*. A su vez, XML nos permite asociar *atributos* a los elementos. Los *atributos* se definen como pares (nombre, valor) por ej: year="1999". Se pueden definir diferentes tipos de atributos pero sus valores deben ser siempre strings y estar encerrados entre comillas. Además, un atributo solamente puede aparecer una vez dentro de un tag.
Más aún, es posible producir datos XML con estructuras complejas sin tener que definir un esquema [DFFLS98].

El siguiente ejemplo muestra cómo catalogar distintos libros en una librería.

```
<bib>
     <book year="1999" isbn="1-55860-622-x">
          <title>Data on the Web</title>
          <author>Abiteboul</author>
          <author>Buneman</author>
          <author>Suciu</author>
     </book>
     <book year="2001" isbn="1-xxxxx-xxx-z">
          <title>XML Query</title>
          <author>Fernandez</author>
          <author>Suciu</author>
     </book>
</bib>
```

Se pueden apreciar los distintos elementos -book- encerrados entre tags. Cada elemento `book`, a su vez, tiene definidos dos atributos -`year` e `isbn`- que lo identifica. En el primer caso los valores son `"1999"` y `"1-55860-622-x"` y en el segundo, `"2001"` y `"1-xxxxx-xxx-z"`.

A veces, puede resultar de gran utilidad tener alguna especificación de la estructura de los datos XML. Esto puede no tener sentido cuando los documentos son pequeños porque los mismos pueden ser recorridos fácilmente, pero cuando los documentos son extremadamente grandes se vuelve algo imprescindible. Para ello se usan los *Descriptores de Tipos de Documentos* (*DTD*). Si bien los DTDs son similares a los esquemas en las bases de datos relacionales y orientadas a objetos, son menos restrictivos y permiten mayor variedad de estructuras.

Por lo tanto, en este escenario de intercambios de datos en la WEB, XML puede contribuir a resolver el problema de proveer una interfaz robusta con herramientas de parseo estables que sean independientes de cualquier formato de muestreo pero no resolverá varios problemas como por ej: cómo se extraerán los datos de grandes documentos XML; cómo se intercambiarán los datos, enviando un documento XML o enviando un query; cómo se intercambiarán datos entre comunidades con diferentes formatos de DTDs o cómo se integrarán datos de diferentes documentos XML. Todos estos problemas fueron resueltos en el ámbito de las bases de datos mediante el uso de lenguajes de consulta, ya sean relacionales u orientados a objetos [DFFLS98]. Sin embargo, estos lenguajes no se adaptan inmediatamente a XML, porque los datos XML difieren de los relacionales u orientados a objetos asemejándose bastante al *modelo de datos semiestructurado*.

Es por eso, que nuestro trabajo presenta un nuevo lenguaje de consulta llamado *CXXML*. Este lenguaje permite recuperar información de documentos XML con una sintaxis similar a SQL. Está basado en el álgebra estándar para XML propuesto por el World Wide Web Consortium (W3C) (www.w3.org) que permite definir claramente la semántica del lenguaje y facilita el proceso de optimización, al permitir una traducción del lenguaje al álgebra sobre la cual se sustenta.

CXXML intenta aumentar la declaratividad de los lenguajes de consulta para XML existentes.
Por ej. en CXXML buscar todos los elementos author de los elementos book del ejemplo anterior se escribe de la siguiente forma:

```
Select author from /bib/book
```

El lenguaje propuesto es más poderoso que otros lenguajes (Lorel, UnQL y XQL) ya que permite realizar operaciones de ordenamiento y agrupación como así también proyección de varios elementos y/o atributos, no implementadas en dichos lenguajes.

En síntesis, las contribuciones de esta tesis son la definición de un nuevo lenguaje de consulta sobre documentos XML con una sintaxis declarativa y la implementación de este lenguaje, incluyendo una interfaz de consulta.
Esta implementación soporta entre otras características:

- Proyección de atributos
- Proyección de listas de elementos o atributos
- Agrupamiento (groupby) con funciones de agregación
- Ordenamiento (orderby)

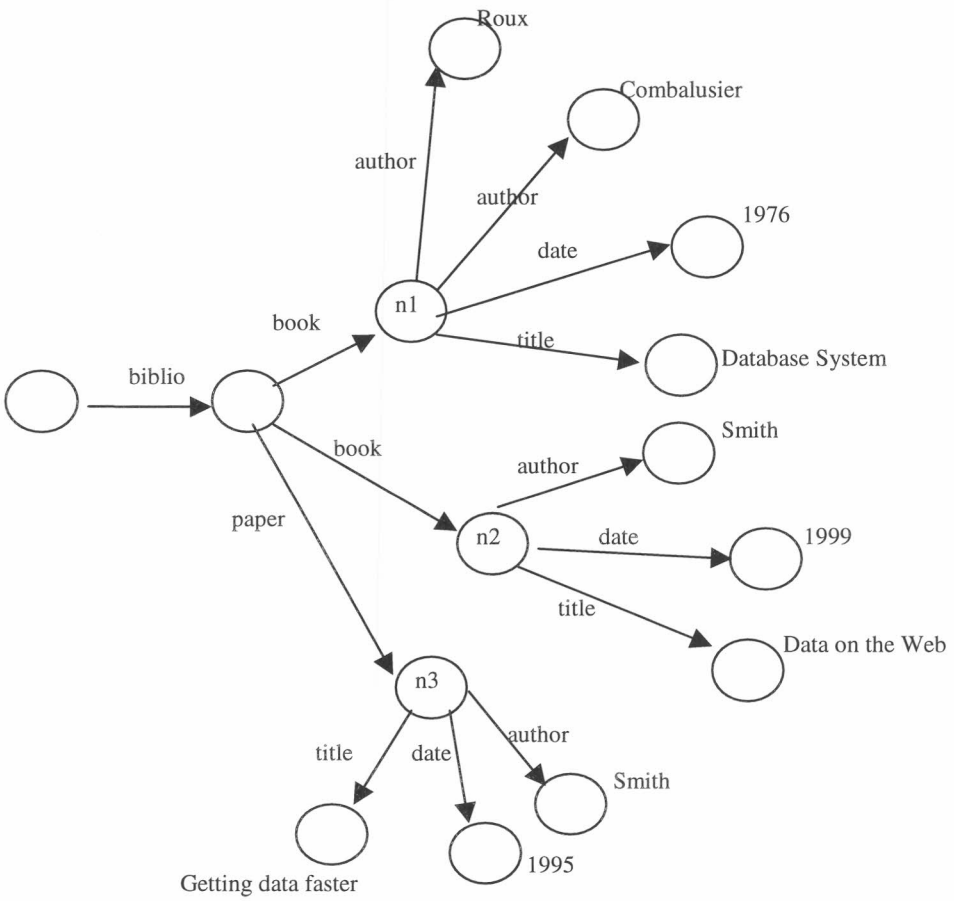
La organización de esta tesis es la siguiente: en el capítulo 2 se presenta un resumen del estado del arte en datos semiestructurados y XML En el capítulo 3 describimos el álgebra estándar para XML propuesta por el W3C; mientras que en el capítulo 4 definimos nuestro lenguaje de consulta. En el capítulo 5 presentamos la implementación del lenguaje. En el capítulo 6 se discuten las pruebas realizadas sobre el prototipo, concluyendo en el capítulo 7.

Capítulo 2

Estado del Arte

2.1 Datos semiestructurados

Los datos semiestructurados surgieron hace unos años como tema de estudio por varias razones. Primero, existen fuentes de datos como la WEB, las cuales nos gustaría tratar como bases de datos pero que no pueden ser limitadas por un esquema. Segundo, podría ser deseable tener un formato extremadamente flexible para el intercambio de datos entre bases de datos dispares. Tercero, incluso cuando estamos tratando con datos estructurados puede ser de gran ayuda verlos como semiestructurados para el propósito de navegarlos.
En los datos semiestructurados, la información que está asociada normalmente con un esquema está contenida dentro de los datos. Esto se debe a que los mismos no tienen un esquema fijo y su estructura puede ser irregular o incompleta, con lo cual la información acerca de los datos hay que buscarla *dentro* de los datos. Por este motivo estos datos son denominados "auto descriptibles" (término que usaremos para referirnos a datos sin estructura) Los datos semi-estructurados se representan como un grafo dirigido o un árbol. Cada eje del árbol es etiquetado con un nombre. Los nodos internos forman los *elementos* y los nodos externos u *hojas* tienen el valor asociado al eje que los vincula.
Un ejemplo de esto es:



Los datos semiestructurados nos dan la flexibilidad para procesar (guardar, consultar, etc.) cualquier dato y nos permiten cambiar su estructura fácilmente. Sin embargo, tienen varios inconvenientes:

- El almacenamiento tiende a ser ineficiente, ya que el esquema necesita ser replicado con cada ítem de datos.
- Las consultas son difíciles de evaluar: incluso una simple consulta puede requerir recorrer todo el grafo.
- Las consultas son difíciles de expresar: el usuario tiene que basarse en información no documentada acerca de los datos para poder formular consultas importantes.

2.2 XML

XML (eXtensible Markup Language) es el nuevo estándar aprobado por el World Wide Web Consortium (W3C) que muchos creen se convertirá en el formato de intercambio de datos de facto para la Web [SUC98]. XML fue diseñado específicamente para describir contenido, a diferencia de HTML que fue diseñado para describir la presentación de los datos [ABS00].
XML es una representación textual de los datos [ABS00]. El componente básico es el *elemento*, es decir, una pieza de texto limitado por *tags* como <persona> y </persona>. Dentro de un elemento podemos tener texto plano, otros elementos o una mezcla de ambos. Consideremos el siguiente ejemplo:

```
<persona>
     <nombre>Patricio</nombre>
     <edad>30</edad>
     <dni>22675500</dni>
     <direccion>diagonal norte 567</direccion>
</persona>
```

La expresión <persona> es llamada *start-tag* y </persona> es llamada *end-tag*. Estos tags deben estar *balanceados*, es decir deben estar cerrados en sentido inverso al que fueron abiertos, como los paréntesis. Los tags en XML son definidos por los usuarios, a diferencia de HTML en donde son predefinidos. El texto embebido entre un start-tag y un end-tag se llama *elemento* y la estructura entre los tags se llama *contenido*.
Todos los datos son tratados como texto. Esto se debe a que XML evolucionó como un lenguaje de marcado de documentos, y los datos son el texto del documento. Estos datos usualmente son referenciados como PCDATA (Parsed Charater Data).
XML nos permite asociar *atributos* dentro de los elementos. Los *atributos* se definen como pares (nombre, valor) por ej: year="1999". Se pueden definir diferentes tipos de atributos pero sus valores deben ser siempre strings y estar encerrados entre comillas. Además, un atributo solamente puede aparecer una vez dentro de un tag.
Existen pocas restricciones que debemos tener en cuenta: los tags deben anidar en forma apropiada y los atributos deben ser únicos. Si un documento XML cumple con estas restricciones decimos que el documento está *bien formado*. Que un documento esté bien formado nos permite asegurar que los datos XML podrán ser parseados en un árbol etiquetado.

XML comparte varias características de los datos semiestructurados: su estructura puede ser irregular, no siempre es conocida de antemano, y puede cambiar varias veces en el tiempo sin previo aviso[SUC98].
Es más poderoso que HTML en tres aspectos:

- Los usuarios pueden definir nuevos tags a voluntad
- Las estructuras de los documentos pueden anidarse hasta cualquier nivel
- Cualquier documento XML puede contener una descripción opcional de su gramática para su uso por aplicaciones que necesitan realizar validaciones de estructura.

Es importante aclarar que XML no es un superconjunto de HTML, ya que, por ejemplo, éste último no requiere que todos los tags tengan sus correspondientes tags de cierre.
Los datos en XML se agrupan en elementos definidos por tags y los elementos pueden ser anidados. Son auto-descriptibles y tienen varias similitudes con los datos semiestructurados. La analogía puede ser sumarizada en la siguiente tabla:

Modelo de datos semi-estructurado	**XML**
Atributos	Tags
Objetos	Elementos
Valores atómicos: string, int, float, video, etc.	Character data (CDATA) strings únicamente

Sin embargo, es importante resaltar que XML es solamente un lenguaje de marcado, y no tiene asociado un modelo de datos. La analogía entre XML (sintaxis) y el modelo de datos semiestructurado (semántica) tiene algunos puntos conflictivos que detallamos a continuación.
Orden. Los documentos XML son por definición ordenados mientras que el modelo de datos semiestructurados es desordenado.
Atributos. Los elementos XML pueden contener *atributos* Cada tag puede tener varios atributos de la forma (atributo=valor) donde el valor siempre es string. Aplicar la misma idea en el modelo de datos semiestructurado no es tan sencillo.
Links. Los documentos XML pueden tener links a elementos dentro de otros documentos XML. Ignorar los links en el modelo de datos lógico puede resultar en grandes faltas de eficiencia.

XML es, como su predecesor SGML, un metalenguage utilizado para definir otros lenguajes. Sin embargo, es mucho más simple y sencillo que SGML. XML es un lenguaje de marcado que no especifica ni las etiquetas (tags), ni la gramática del lenguaje. El conjunto de etiquetas de un lenguaje de marcado define las etiquetas de marcado que tienen significado para un analizador del lenguaje. Por ejemplo, HTML tiene un conjunto estricto de etiquetas permitidas. Se puede usar el tag <table> pero no el tag <silla> porque el primero tiene un significado específico para una aplicación que use los datos y se utiliza para indicar el inicio de una tabla en HTML; la segunda no tiene ningún significado específico. La gramática de un lenguaje de marcado define la correcta utilización de las etiquetas del lenguaje.
XML, al no definir ni las etiquetas ni la gramática, es completamente extensible; como indica su propio nombre. Esto es lo bueno de XML: se permite definir el contenido de los datos de la forma que se desee mientras sea conforme a la estructura general que XML requiere.

2.3 DTD

Un *document type definition -o Descriptor de Tipos de Documentos- (DTD)* sirve como gramática para el documento XML en cuestión, y es parte del lenguaje XML. Incluso un DTD puede servir como esquema del documento [ABS00]. En otras palabras, un DTD especifica qué elementos pueden aparecer en un documento y cómo pueden anidarse [DFFLS98].
Consideremos por ejemplo un documento XML consistente de una cantidad arbitraria de elementos de tipo "person", de la forma:

```
<db>
     <person>
          <name>Alan</name>
          <age>42</age>
          <email>agb@abc.com</email>
     </person>
     <person>
          ...
     </person>
     ...
</db>
```

Un DTD posible podría ser

```
<!DOCTYPE db[
     <!ELEMENT db (person*)>
     <!ELEMENT person (name, age, email)>
     <!ELEMENT name (#PCDATA)>
     <!ELEMENT age (#PCDATA)>
     <!ELEMENT email (#PCDATA)>
]>
```

La primera línea dice que el elemento raíz es <db>. Las siguientes líneas son declaraciones diciendo que <db> puede contener un número arbitrario de elementos <person>, cada uno de los cuales puede contener elementos <name>, <age> y <email>.
Estos elementos sólo pueden contener datos character. La declaración *person** es una expresión regular que indica que puede existir cualquier número de elementos person.

Un DTD es precisamente una gramática libre de contexto para el documento. En particular, la declaración anterior impone que <name>, <age> y <email> aparezcan en ese orden en un elemento <person>.
Un documento XML que tiene un DTD asociado se denomina *válido*. Es decir, los elementos pueden anidarse solamente de la forma que describe el DTD y sólo puede tener atributos permitidos por el DTD.
Como mencionamos anteriormente, los DTDs imponen orden. Otra limitación que tienen es que no manejan la noción de tipos atómicos. Por ejemplo, nos gustaría decir que el contenido del campo <age> (edad) fuese entero; sin embargo el único tipo de datos posible es #PCDATA, lo que significa string.

2.4 SCHEMA

XML Schema es definido por W3C [W3CSCHEMA] para definir clases de documentos XML y mejorar las limitaciones que tienen los DTDs. El Schema es frecuentemente usado para describir un documento XML. A continuación veremos un ejemplo de una orden de compra en XML y su correspondiente Schema asociado.

```
<purchaseOrder orderDate="1999-10-20">
      <shipTo country="US">
           <name> Alice Smith </name>
           <street> 123 Maple Street </street>
           <state> CA </state>
      </shipTo>
      <billTo country="US">
           <name> Robert Smith </name>
           <street> 8 Oak Avenue </street>
           <state> PA </state>
      </billTo>
      <comment>Hurry, my lawn is going wild!</comment>
      <items>
           <item partNum="872-AA">
                <productName>Lawnmower</productNAme>
                <quantity>1</quantity>
                <USPrice>148.95</USPrice>
           </item>
           <item partNum="926-AA">
                <productName>Baby Monitor</productNAme>
                <quantity>1</quantity>
                <USPrice>148.95</USPrice>
                <shipDate>1999-05-21</shipDate>
           </item>
      </items>
 </purchaseOrder>
```

Ahora veremos el Schema asociado al ejemplo anterior:

```
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
      <xsd:sequence>
           <xsd:element name="shipTo" type="USAddress"/>
           <xsd:element name="billTo" type="USAddress"/>
           <xsd:element name="items" type="Items"/>
      </xsd:sequence>
      <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
<xsd:complexType name="USAddress">
      <xsd:sequence>
           <xsd:element name="name" type="xsd:string"/>
           <xsd:element name="street" type="xsd:string"/>
           <xsd:element name="state" type="xsd:string"/>
      </xsd:sequence>
```

```
            <xsd:attribute name="country" type="xsd:NMTOKEN"
        fixed="US"/>
      </xsd:complexType>
      <xsd:complexType name="Items">
            <xsd:sequence>
                  <xsd:element name="item" minOcurrs="0"
            maxOccurs="unbounded">
              <xsd:complexType name="Items">
                  <xsd:sequence>
                              <xsd:element name="productName"
                 type="xsd:string"/>
                              <xsd:element name="quantity">
                                    <xsd:simpleType>
                                          <xsd:restriction
                    base="xsd:positiveInteger">
                        <xsd:maxExclusive value="100"/>
                     </xsd:restriction>
                   </xsd:simpleType>
                </xsd:element>
              <xsd:element name="USPrice" type="xsd:decimal"/>
              <xsd:element name="shipDate" type="xsd:date"/>
            <xsd:sequence>
          </xsd:complexType>
      </xsd:element>
    <xsd:sequence>
</xsd:complexType>
```

Los elementos del Schema de mayor relevancia son : `element`, `complexType` y `simpleType`, que determinan la apariencia de los elementos y su contenido en la instancia del documento XML de la orden de compra. Existe una diferencia básica entre tipos complejos, que permiten contener elementos y atributos, y tipos simples que no contienen atributos ni elementos.
Se pueden definir elementos opcionales, como es el caso de `comment` dentro del elemento `PurchaseOrderType`.
También se puede aplicar restricciones de ocurrencias, con el uso de los atributos `minOccurs` y `maxOccurs`.
Por último se presentan varios tipos simples como ser `string` y `date`, que son *built-in* en Schema. Existen muchos tipos simples built-in y tambien derivados de ellos.
Con el Schema se definen las características de un documento XML y se presenta una mejora a las limitaciones del DTD.

2.5 SAX

SAX (Simple Api for XML) proporciona un marco basado en eventos para analizar datos XML, es decir, realizar el proceso de leer detenidamente el documento, desglosando los datos en partes utilizables. SAX lee el flujo de datos XML, lo parsea y detecta eventos a medida que interpreta los tags. Estos eventos son enviados de vuelta a la aplicación que invocó al parser [ABS00]. Por ejemplo, define métodos tales como startDocument() y endElement(). SAX proporciona el *medio* para analizar un documento XML.

2.6 JDOM

JDOM es una API hecha en Java para el Modelo de Objetos de Documento (DOM en inglés, definido por W3C). Mientras que SAX sólo proporciona acceso a los datos de un documento XML, DOM está diseñado para proporcionar un medio para manipularlos. DOM proporciona una representación de un documento mediante un árbol. De esta forma, resulta sencillo lograr el recorrido y manipulación de los datos ya que esa estructura es muy conocida por todos los lenguajes de programación y en especial, Java.
DOM carga un documento XML entero en memoria, almacenando todos los datos como *nodos (nodes)*, de forma que se puede acceder a todo el documento entero rápidamente. Además de la interfaz Node, define subclases Element (elemento), Attribute (atributo) y CharacterData (dato) todas éstas heredando de la clase Node. La clase Node define métodos para acceder a los componentes de un nodo. Por ejemplo, parentNode() devuelve el nodo padre; childnodes() devuelve un conjunto de hijos. Adicionalmente, existen métodos que modifican los nodos y permiten crear nodos nuevos.
Existe otra clase llamada Document que define métodos para trabajar con documentos XML. Los más importantes son new() que define un documento nuevo; getRootElement() que devuelve la raíz de un documentoy setRootElement() que permite cargar un árbol en un documento.
JDOM es una representación en Java de un documento XML, mucho más facil de usar que DOM, permitiendo una forma fácil y eficiente de leer, manipular y escribir XML [JDOM]. Es una alternativa de DOM y SAX, pero los integra muy bien a ambos. Usa cualquier parser para construir documentos, en particular, tiene clases preparadas para soportar eventos tipo SAX para construir arboles JDOM.

2.7 XPath

El propósito principal de XPath es referenciar partes de un documento XML [CD99]. Para ello provee facilidades para el manejo de cadenas de texto, números y condiciones lógicas. XPath utiliza una sintaxis totalmente distinta a la de XML para facilitar su uso en los valores de los atributos de los documentos XML. Su nombre proviene del uso de una notación como la usada en las *URLs* (Universal Resource Locator) para navegar a través de la estructura jerárquica de un documento XML.
XPath modela un documento XML como un árbol de nodos. Existen distintos tipos de nodos incluyendo nodos de tipo *elemento*, *atributo* y *texto*.
La construcción sintáctica básica es la *expresión*. Una clase importante de expresiones es *location path*. Esta selecciona un conjunto de nodos relativos al nodo contexto. El resultado de evaluar una expresión location path es un conjunto de nodos seleccionados por el location path.
Existen dos tipos de location path: *relativos* y *absolutos*. Un location path *relativo* consiste de una secuencia de uno o más *pasos* separados por '/'. Cada paso selecciona un conjunto de nodos relativos a un nodo contexto. Por ejemplo:

```
child::div/child::para
```

selecciona los elementos `para` hijos de los elementos `div`, hijos a su vez del nodo contexto.
Un location path *absoluto* consiste de un '/' opcional seguido de un location path relativo. El símbolo '/' por sí solo selecciona el nodo raíz del documento conteniendo el nodo contexto. Por ejemplo:

```
/biblio/book
```

selecciona los elementos `book` hijos de los elementos `biblio` que son hijos del nodo raíz.
En los ejemplos anteriores mencionamos a la función `child::` la cual devuelve los hijos del elemento que recibe como parámetro. Como ésta, existen otras funciones que dan soporte a la implementación del lenguaje como `attribute::` que selecciona el atributo que recibe como parámetro del nodo contexto y `ancestor::` que devuelve todos los elementos ancestros del tipo que recibe como parámetro del nodo contexto.

2.8 Lorel

Lorel surgió como un lenguaje de consulta dentro del marco del proyecto TSIMMIS y fue diseñado específicamente para datos semiestructurados. Una característica importante de este lenguaje es que explota la estructura de los datos cuando ésta se encuentra presente, pero no requiere estructuras uniformes para dar respuestas correctas [QRSUW97]. Soporta objetos y relaciones entre objetos.
Una sentencia típica en Lorel sería:

```
SELECT Frodos.Group.Name
FROM Frodos
WHERE Frodos.Group.Category = "Opera"
```

Y la respuesta a la consulta:

```
Answer
     Name "palo Alto Savoyards"
```

La respuesta siempre se *empaqueta* dentro de un objeto llamado por default `Answer`.
Los *path expressions* forman la base de las consultas en Lorel. En este lenguaje, los path expressions se definen como una secuencia de etiquetas separadas por puntos. El ejemplo mostrado anteriormente contiene dos path expressions: uno en la cláusula SELECT y otros en la cláusula WHERE.
La cláusula FROM salvo para el caso de la utilización de *wildcars* (*, ?) es opcional y redundante ya que los path expressions deben comenzar con el objeto mencionado en la cláusula FROM.
Lorel no maneja chequeo de tipos con lo cual permite comparar objetos de distinto tipo o usar comparadores que no están definidos para un tipo dado; este tipo de comparaciones simplemente devuelven *false*.
Tiene definido todo tipo de operaciones exceptuando la proyección de atributos, el agrupamiento con funciones de agregación y el ordenamiento.

2.9 XML-QL

Es un lenguaje propuesto al W3C en 1998 como solución a los distintos problemas que se plantean al momento de realizar consultas sobre documentos XML. El lenguaje tiene una construcción `select-where` similar a SQL e implementa algunas de las

características de los lenguajes recientemente implementados para datos semi-estructurados [DFFLS98].
Un ejemplo de este lenguaje es:

```
WHERE <libro>
          <editor><nombre>Addison-
Wesley</nombre></editor>
          <titulo>$t</titulo>
          <autor>$a</autor>
       </libro> IN www.a.b.c/bib.xml
CONSTRUCT $a
```

Informalmente, esta consulta obtiene cada elemento `<libro>` del documento XML especificado luego de la palabra clave IN, que tenga por lo menos un elemento `<titulo>`, un elemento `<autor>` y un elemento `<editor>` cuyo elemento `<nombre>` sea igual a Addison-Wesley. Para cada valor obtenido, liga las variables `t` y `a` a cada par de valores `titulo` y `autor`. El reultado es la lista de autores ligados a `a`. Todas las variables están precedidas por un símbolo '$' para distinguirlas de los strings literales.
El ejemplo anterior sirve para listar los elementos que estamos buscando. Si la última sentencia la reemplazáramos por:

```
CONSTRUCT <result>
               <autor>$a</autor>
               <titulo>$t</titulo>
          </result>
```

Obtendríamos como resultado un nuevo documento que agrupa los pares de elementos `autor-titulo` en un nuevo elemento llamado `result`.
Adicionalmente a la selección de elementos, permite realizar consultas sobre los atributos, operaciones de *ordenamiento*, *agrupamiento y joins*.

2.10 UnQL

UnQL, es un lenguaje relacionado con Lorel [AQM+97] , que tambien fue diseñado para consultar datos semi-estructurados. UnQL esta basado en el uso de *patterns*, que permite relacionar variables con nodos, permitiendo al usuario expresar en forma concisa condiciones complejas sobre la estructura [ABS00]. A continuación damos un ejemplo de una selección:

```
select title: X
where {biblio: {paper: {title: X, year: Y}}} in db,
     Y > 1989
```

Nota: db es la raiz del documento

Otra característica importante de UnQL es un poderoso constructor llamado *traverse*, que permite reestructurar arboles para una determinada profundidad. Esta operaciones

no existen en otros lenguajes, que primariamente fueron diseñados para consultar solamente.
Para construir grafos, UnQL tiene los siguientes operadores [SY98]:

Constructores de grafos :

- { } : un grafo vacio
- { l => g } : l es una etiqueta de eje y g es un grafo
- g1 U g2 : g1 y g2 son grafos

Operadores adicionales :

- select
- if _ then _ else _
- traverse (reestructuración de arboles a una determinada profundidad)
- @ (expresa operaciones sobre estructura cíclicas)

Se dice que UnQL es un lenguaje de consulta para sitios Web, para datos semi-estructurados o para grafos.

2.11 Xquery

Xquery es un lenguaje de consulta para XML diseñado por W3C XML Query Working Group, definido en [W3CXQ]. Fue diseñado para realizar consultas concisas y fáciles de entender. Deriva de un lenguaje de consulta llamado *Quilt* [RFC00] y además incluye características de Xpath 1.0, XQL, XML-QL, SQL y OQL.
Su sistema de tipos está basado en XML Schema [W3CSCHEMA] y es un lenguaje fuertemente tipificado, aunque permite al usuario deshabilitar esta opción.
A continuación mostraremos ejemplos de uso, en notación abreviada para path expressions y posicionado en el nodo raíz del documento:

- la sentencia **para** selecciona todos los elementos hijos de tipo **para**
- la sentencia * selecciona todos los elementos hijos del elemento raíz
- **@name** selecciona el atributo **name**
- **para[1]** selecciona el primer hijo **para**
- **//list/member** selecciona todos los elementos **member** hijos de **list**
- **//book1/author eq "Kennedy**" selecciona todos los elementos **author** hijos de **book** con valor igual a **"Kennedy"**

A partir de las construcciones sintácticas definidas en el lenguaje y estructuras de control, variables, etc. , se pueden construir aplicaciones para realizar diversas clases de consultas, como ser Ordenamiento, Join's, Secuencias o Agrupamiento.

2.12 XSL

XSL es un lenguaje para expresar hojas de estilo (styleSheet) en XML, creado por W3C [W3CXSL]. Su rol primario es permitir a los usuarios escribir transformaciones desde XML a HTML. Como lenguaje de transformaciones, también puede servir para aplicaciones de datos [ABS00], por lo que solo presentaremos un análisis sobre esta característica. XSL no es relacionalmente completo, como lo es XML-QL. No permite hacer joins y en general el poder expresivo de las transformaciones es limitado. XSL consiste de tres partes: XSL Transformations (XSLT), un lenguaje para tranformar documentos XML; XML Path Languaje (Xpath) y XSL Formatting Object, un vocabulario en XML para especificar formatos. Un styleSheet XSL especifica la presentación de una clase de documentos XML, describiendo cómo una instancia de esa clase es tranformada en otro documento XML usando el vocabulario de formateo.
Un programa XSL es un conjunto de reglas. Veremos a continuación un ejemplo:

```
<bib>
     <book>
          <title>t1</title>
          <author>a1</author>
          <author>a2</author>
     </book>
     <paper>
          <title>t2</title>
          <author>a3</author>
          <author>a4</author>
     </paper>
     <book>
          <title>t3</title>
          <author>a5</author>
          <author>a6</author>
          <author>a6</author>
     </book>
</bib>
```

El siguiente programa XSL retorna todos los títulos de libros y papers:

```
<xsl:template>
     <xsl:apply-templates/>
</xsl:template>
<xsl:template match="/bib/*/title">
 <result>
     <xsl:value-of/>
 </result>
</xsl:template>
```

Cada par <xsl:template> construye una regla. En la segunda regla es donde vemos la selección de los títulos.

El resultado tiene la forma :

```
<result> t1 </result>
<result> t2 </result>
<result> t3 </result>
```

XSL no maneja variables, por lo que permite hacer programas concisos pero limita el poder expresivo, por ejemplo, no permite realizar Joins.

2.13 XQL

XQL es un lenguaje de consulta para XML, basado en el lenguaje XML-QL de W3C y a su vez enriquecido con definiciones de lenguajes estándares como ser SQL y OQL.
La sentencia característica de este lenguaje es la estructura select-from-where similar a SQL y OQL [W3CXQL]. A continuación presentamos un ejemplo de una selección en XQL:

```
select $book.author
from bib:URL www.a.b.c/bib.xml, book:$bib.book
where $book.publisher.name = "Addison-Wesley"
```

Como se ve en el ejemplo, XQL trabaja con *path expressions* para referenciar a los documentos XML o bien trabaja directamente con la URL. En ambos casos define variables de tipo elemento, como `bib` o `book`, para referenciarlos y utilizarlos en otras cláusulas de la sentencia, como ser el `select` o el `where`.
Para poder proyectar varios elementos, tiene un operador de construcción de elmentos llamado `result`, que se antepone a los elementos a proyectar, como veremos en el siguiente ejemplo:

```
select result:<$book.author, $book.title>
from bib:URL www.a.b.c/bib.xml, book:$bib.book
where $book.publisher.name = "Addison-Wesley"
```

XQL da soporte también a funciones de agrupamiento y ordenamiento, con cláusulas similares a SQL, como ser `groupby` y `orderby`. Presentamos a continuación ejemplos de las mismas:

```
select result:<$book.author, $book.title>
from bib:URL www.a.b.c/bib.xml, book:$bib.book
where $book.publisher.name = "Addison-Wesley"
groupby $book.title

select result:<$book.author, $book.title>
from bib:URL www.a.b.c/bib.xml, book:$bib.book
where $book.publisher.name = "Addison-Wesley"
orderby $book.title
```

Otras características importantes de este lenguaje son que permite realizar la operación de Join, realizar sub-querys, definir funciones y dar soporte a DTDs o Schemas.

2.14 Resumen

En este capítulo describimos los datos semiestructurados, sus ventajas y desventajas y la forma que se utiliza para representarlos. Mostramos qué es XML, cuál es el formato de un documento XML y cuál es su relación con los datos semiestructurados. Detallamos las razones por las cuales XML no es considerado un superconjunto de HTML y por qué se lo considera un lenguaje totalmente extensible. Explicamos para qué sirven los DTDs y los SCHEMAs y por qué son necesarios para los documentos XML.
Finalmente describimos distintas herramientas, como SAX y JDOM, y lenguajes de consulta, como Xpath, Lorel, XML-QL, UnQL, Xquery, XSL y XQL que se utilizan en la actualidad para extraer información de documentos XML.

Capítulo 3

Un álgebra standard para XML

En el mundo de las bases de datos, es común traducir un lenguaje de consulta a algún álgebra; esto pasa en SQL, OQL y NRA entre otros. El propósito del álgebra es doble. Primero, es usada para darle una semántica al lenguaje, por lo tanto las operaciones del álgebra deberían estar bien definidas. Segundo, el álgebra es usada para realizar optimización de consultas, con lo cual el álgebra debe poseer un conjunto rico de leyes.
El álgebra definida por el W3C [W3CXQA] es lo suficientemente poderosa para capturar la semántica de muchos lenguajes de consulta para XML, y las leyes definidas ahí incluyen analogías con la mayoría de las leyes del álgebra relacional.
El álgebra mencionada define los siguientes operadores: Proyección, Iteración. Selección, Join, GroupBy, OrderBy y Agregación -sum, avg, count, max, min-.
Para describir cada una de ellas, usaremos el ejemplo XML que se muestra a continuación, referido al contenido de una biblioteca:

```
<bib>
     <book year="1999" isbn="1-55860-622-x">
          <title>Data on the Web</title>
          <author>Abiteboul</author>
          <author>Buneman</author>
          <author>Suciu</author>
     </book>
     <book year="2001" isbn="1-xxxxx-xxx-z">
          <title>XML Query</title>
          <author>Fernandez</author>
          <author>Suciu</author>
     </book>
</bib>
```

Este ejemplo tiene un elemento raíz `<bib>` que contiene dos elementos `<book>`. Cada uno de estos elementos tiene definidos dos atributos `year` e `isbn`, y está compuesto, a su vez, por un elemento `<title>` y uno o varios elementos `<author>`.
Un elemento `<book>` representa un libro. El elemento `<title>` representa el nombre del libro y el elemento `<author>` representa un autor del mismo. Los atributos `year` e `isbn` representan el año de publicación y su número de identificación, respectivamente.
El álgebra usa una notación similar en apariencia y significado al path de navegación usado en Xpath. Todos los ejemplos que veremos a continuación constan de dos partes. La primera es una expresión del álgebra. La segunda, a continuación del símbolo "➔" es el resultado de la expresión.

3.1 Proyección

Esta operación es análoga a la proyección en álgebra relacional. Por ejemplo, la siguiente expresión devuelve todos los elementos `author` contenidos en el elemento `book`, contenidos a su vez en el elemento `bib`.

```
bib/book/author
➔    author["Abiteboul"],
     author["Buneman"],
     author["Suciu"],
     author["Fernandez"],
     author["Suciu"]
```

Nótese que en el resultado, el orden de `author` del documento se mantiene como así también los elementos duplicados.
Así como accedemos a los elementos, podríamos acceder a los atributos de los elementos.

```
bib/book/@year/data()
➔    1999
     2001
```

La palabra reservada `data()` se usa para acceder al dato atómico mientras que el símbolo '@' sirve para identificar a los atributos.

3.2 Iteración

Otra operación común es la de iterar sobre elementos en un documento para que su contenido pueda ser transformado en nuevo contenido.
A continuación mostramos un ejemplo de cómo procesar cada elemento `book` para listar el `autor` antes del `título`, y remover el `año` y el `isbn`.

```
FOR b IN bib/book DO
     book [b/author, b/title]
➔    book[
          author["Abiteboul"],
          author["Buneman"],
          author["Suciu"],
          title["Data on the Web"]
     ],
     book[
          author["Fernandez"],
          author["Suciu"],
          title["XML Query"]
     ]
```

La expresión FOR itera sobre todos los elementos book en bib, y relaciona la variable b a cada uno de esos elementos. Para cada elemento relacionado a b, la expresión interna construye un nuevo elemento book conteniendo el autor del elemento book seguido de su título. Los elementos transformados aparecen en el mismo orden que en bib.
En general, el valor de una expresión FOR es un *bosque*. Si el cuerpo de la expresión FOR genera un bosque, entonces todos los bosques son concatenados.
Por ejemplo, la expresión:

```
FOR b IN bib/book DO
     b/author
```

es exactamente equivalente a la expresión bib/book/author.

3.3 Selección

Este operador selecciona elementos que satisfacen algún predicado. La selección se expresa con la cláusula WHERE. Por ejemplo, si queremos seleccionar todos los libros que fueron publicados antes del año 2000.

```
FOR b IN bib/book DO
     WHERE  b/@year/data() <=2000 DO b

➔ book [
     @year [1999],
     @isbn ["1-55860-622-x"],
     title["Data on the Web"],
     author["Abiteboul"],
     author["Buneman"],
     author["Suciu"],
     ]
```

La expresión FOR itera sobre todos los elementos book en bib, y relaciona la variable b a cada uno de esos elementos. Si el dato del atributo year cumple con la condición de ser menor o igual a "2000" entonces se devuelve como resultado el elemento relacionado a b.

En general, una expresión de la forma:
WHERE e_1 DO e_2
Es convertida a la forma
IF e_1 THEN e_2 ELSE ()
Donde e_1 y e_2 son expresiones. El símbolo () es una expresión que significa la secuencia vacía.

Por lo tanto, de acuerdo con esta regla, la expresión de arriba se traduce a

```
FOR b IN bib/book DO
```

```
IF  b/@year/data() <=2000 THEN b ELSE ()
```

3.4 Distinct

El operador *distinct* recibe como parámetro un *bosque* y devuelve como resultado uno nuevo sin elementos repetidos.
La siguiente expresión selecciona todos los autores de libros eliminando repetidos.

```
distinct (bib/book/author/data())

➔   author["Abiteboul"],
    author["Buneman"],
    author["Suciu"],
    author["Fernandez"]
```

En nuestro ejemplo la función `distinct()` recibe como argumento los valores de los elementos `author` -Abiteboul, Buneman, Suciu, Fernandez y Suciu- y devuelve un nuevo conjunto sin repeticiones -Abiteboul, Buneman, Suciu, Fernandez.

3.5 Join

Para ilustrar este operador definimos un segundo documento que define críticas de libros:

```
<reviews>
    <book>
        <title>XML Query</title>
        <review>A very fine book</title>
    </book>
    <book>
        <title>Data on the Web</title>
        <review>this is great!</review>
    </book>
</reviews>
```

Supongamos que queremos obtener en un único documento los datos de cada libro junto a sus respectivas críticas. En el álgebra, esto se expresa de la siguiente manera:

```
FOR b IN bib/book DO
    FOR r IN reviews/book DO
        WHERE b/title/data() = r/title/data() DO
            Book[b/title, b/author, r/review]

➔book[
    title["Data on the Web"],
```

```
        author["Abiteboul"],
        author["buneman"],
        author["Suciu"],
        review["A very fine book"]
        ],
    book[
        title["XML Query"],
        author["Fernandez"],
        author["suciu"],
        review["this is great!"]
        ]
```

La primera expresión FOR itera sobre todos los elementos book contenidos en bib, y relaciona la variable b a cada uno de esos elementos. La segunda expresión FOR itera sobre todos los elementos book contenidos en reviews, y relaciona la variable r a cada uno de esos elementos. Si el dato del elemento title asociado a b cumple con la condición de ser igual al dato del elemento title asociado a r, se devuelve como resultado un nuevo elemento book que contiene los elementos title y author relacionados a b y el elemento review relacionado a r.

Nótese que el FOR exterior determina el orden del resultado. A diferencia de los lenguajes relacionales en los cuales la optimización del join se puede hacer en cualquier orden, esto no ocurre en el álgebra para consultas de XML, es decir, cambiar el orden de las expresiones FOR cambiará el resultado.

3.6 GroupBy

A menudo es útil agrupar elementos en un documento XML. Por ejemplo, la consulta "Obtener todos los títulos de los libros contenidos en bib agrupados por autor", se expresa en el álgebra de la siguiente forma:

```
FOR a IN distinct (bib/book/author/data()) DO
     Biblio[
          Author[a],
          FOR b IN bib/book DO
               FOR a2 IN b/author/data() DO
                    WHERE a = a2 DO
                         b/title
     ]

➔biblio[
     author["Abiteboul"],
     title["Data on the Web"]
  ],
  biblio[
     author["Buneman"],
     title["Data on the Web"]
  ],
```

```
biblio[
   author["Suciu"],
   title["Data on the Web"],
   title["XML Query"],
],
biblio[
   author["Fernandez"],
   title["XML Query"],
]
```

La expresión `distinct (bib/book/author/data())` produce un *bosque* de elementos de autores sin duplicados. El `FOR` externo liga la variable `a` a cada valor del elemento `author`. Para cada uno de esos valores, el primer `FOR` interno selecciona los elementos `book` que se encuentran en `bib` y los relaciona a la variable `b`; el segundo `FOR` interno relaciona la variable `a2` a cada valor del elemento `author` ligado a la variable `b` y compara el valor de la variable `a` con el de la variable `a2`. Si son iguales devuelve el elemento `title` ligado a la variable `b` junto al elemento `author` ligado a la variable `a`. Este par de elementos se crean dentro de un nuevo elemento llamado `biblio`.

3.7 Sort

Para ordenar un bosque, el álgebra provee una expresión `sort`, cuyo formato es:

`sort Var IN` Exp_1 `by` Exp_2.

La variable `Var` recorre los ítems en el bosque Exp_1 y ordena esos ítems usando el valor clave definido por Exp_2.
Por ejemplo, la siguiente expresión ordena los libros que se encuentran en `reviews` por sus títulos.

```
Sort b IN reviews/book by b/title

➔book [
     title ["Data on the web"],
     review["this is great!"]
  ]
  book[
     title["XML Query"],
     review["A very fine book"]
  ]
```

Nótese que la variable `b` aparece en la expresión clave `b/title`.

3.8 Agregación

El álgebra define cinco funciones de agregación built-in: avg, count, max, min y sum. Esta expresión selecciona libros que tienen más de dos autores:

```
FOR b IN bib/book DO
     WHERE count(b/author)>2 DO b

➔book[
     @year[1999],
     @isbn["1-55860-622-x"],
     title["Data on the web"],
     author["Abiteboul"],
     author["Buneman"],
     author["Suciu"]
  ]
```

Todas las funciones de agregación toman un bosque y devuelven un valor entero; `count` devuelve el número de elementos de un bosque.

3.9 Unión

Si bien Frankhauser *et al* [FFMRSW00] no hacen mención explícita a la operación *Union*, sí la mencionan Fernandez *et al* [FR01] que definen la operación *Union* de la siguiente manera:

$$\text{Union} : (\{ S_1 \}, \{ S_2 \}) \rightarrow \{S_3\}$$

donde la función Union crea un nuevo bosque desordenado, conteniendo la unión disjunta de cada uno de sus argumentos. En nuestro caso, S_1 y S_2 son cualesquiera de las expresiones vistas con anterioridad. De esto se deduce que esta operación consta de tres pasos: los dos primeros resuelven las operaciones definidas para S_1 y S_2, mientras que el tercero realiza la Union propiamente dicha.

Por ejemplo, si definimos a S_1 como:

```
FOR b IN bib/book DO
     WHERE  b/@year/data() <=2000 DO b
```

Y definimos a S_2 como:

```
FOR b IN bib/book DO
     WHERE  b/@year/data() >2000 DO b
```

El resultado de realizar `Union(`S_1`, `S_2`)` será:

```
➔ book [
     @year [1999],
     @isbn ["1-55860-622-x"],
     title["Data on the Web"],
     author["Abiteboul"],
```

```
    author["Buneman"],
    author["Suciu"],
    ]
  book [
    @year [2001],
    @isbn ["1-xxxxx-xxx-x"],
    title["XML Query"],
    author["Fernandez"],
    author["Suciu"],
    ]
```

3.10 Intersección

De la misma forma que en el punto 3.10, en [FFMRSW00] no se hace una mención explícita a la operación *intersección*, en cambio sí se la menciona en [FR01]. Allí la operación *Intersect* se define de la siguiente manera:

```
Intersect : ( { S1 }, { S2}) ➔ {S3}
```

La función *Intersect* crea un nuevo bosque desordenado, conteniendo valores que ocurren en el primer y segundo argumento. En nuestro caso, S_1 y S_2 son cualesquiera de las operaciones vistas en este capítulo. De esto se deduce que esta operación consta de tres pasos: los dos primeros resuelven las operaciones definidas para S_1 y S_2, mientras que el tercero realiza la intersección propiamente dicha.

Por ejemplo, si definimos a S_1 como:

```
FOR b IN bib/book DO
    WHERE  b/@year/data() >1999 DO b
```

Y definimos a S_2 como:

```
FOR b IN bib/book DO
    WHERE  b/@year/data() <=2001 DO b
```

El resultado de realizar `Intersect(`S_1`, `S_2`)` será:

```
➔ book [
    @year [2001],
    @isbn ["1-xxxxx-xxx-x"],
    title["XML Query"],
    author["Fernandez"],
    author["Suciu"],
    ]
```

Para poder definir estas operaciones es necesario contar con un operador de igualdad para elementos XML. El XML Query Data Model [FR] define el operador de igualdad de nodos '= =', donde el valor será verdadero si y solo si los dos nodos son exactamente el mismo nodo, siendo dicho operador dependiente de la implementación elegida. El detalle de cómo implementamos dicho operador se encuentra en la sección [5.4] del capítulo *Implementación*.

3.11 Relacionando proyección con iteración

Los ejemplos anteriores usan el operador "/" libremente, pero de hecho el álgebra usa "/" como una abreviatura de expresiones construídas a partir de operadores de más bajo nivel: expresiones FOR, la función children, y la expresión match.

Por ejemplo, la expresión:

```
book/author
```

es equivalente a la expresión:

```
FOR v IN children(book) DO
     Match v
          Case a: author[AnyComplexType] DO a
          ELSE()
```

Acá la función children devuelve un bosque consistente de los hijos de los elementos book, es decir, un elemento title y tres elementos author (se preserva el orden). La expresión FOR liga la variable v sucesivamente a cada uno de esos elementos. Luego, la expresión match selecciona una rama en función del tipo de v. Si es un elemento author entonces la primer rama es evaluada, de lo contrario la segunda. Si la primer rama es evaluada, devuelve a. Si la segunda rama es evaluada, devuelve la secuencia vacía ().

Para componer varias expresiones usando /, usamos nuevamente expresiones FOR. Por ejemplo, la expresión:

```
bib/book/author
```

Es equivalente a la expresión:

```
FOR b IN children(bib) DO
Match b
     Case b: book[AnyComplexType] DO
          FOR d IN children(b) DO
          Match  d
               Case a: author[AnycomplexType] DO a
               ELSE()
     ELSE()
```

La expresión FOR itera sobre todos los elementos book en bib y liga la variable b a cada uno de esos elementos. Para cada uno de esos elementos ligados a b, la expresión interna devuelve todos los elementos author en b, y los bosques resultantes son concatenados juntos en orden.

En general, una expresión de la forma e/a se convierte a la forma:

```
FOR v1 IN (e) DO
     FOR v2 IN children (v1) DO
     match v2
          case v3: a[AnyComplexType] DO v3
          ELSE ()
```

donde e es un elemento, a es un nombre de un elemento, y v_1, v_2 y v_3 son variables auxiliares.

De acuerdo a esta regla, la expresión `bib/book` se traduce a:

```
FOR b1 IN bib DO
     FOR b2 IN children (b1) DO
match b2
     case c1: book[AnyTypeof] DO c1
          ELSE ()
```

Y la expresión `biblio/book/author` se traduce a

```
FOR b1 IN (FOR b2 IN children (biblio) DO
          match b2
               case c1: book[AnyTypeof] DO c1
               ELSE ()) DO
     FOR b3 IN children(b1) DO
          match b3
               case c2: author[AnyTypeof] DO c2
               ELSE ()
```

3.12 Simplificación de Expresiones

El álgebra maneja algunas leyes que permiten definir reglas que simplifican las expresiones, eliminando expresiones *for* o *match* innecesarias. Si tomamos el último caso, la expresión simplificada es:

```
FOR b1 IN children(biblio) DO
Match b1
     Case c1: book [] DO
          FOR b2 IN children(c1) DO
               match b2
                    case c2: date[AnyTypeof] DO c2
               ELSE ()
```

Las leyes que relacionan las expresiones reducibles a expresiones equivalentes, se detallan a continuación:

```
(1) e / Wild        → FOR v1 IN e DO
                        FOR v2 IN children (v1) DO
                          match v2 case v3 : Wild[AnyType] DO v3
                          ELSE ()
```

```
(2) e / @Wild        → FOR v1 IN e DO
                          FOR v2 IN children (v1) DO
                            match v2 case v3 : @Wild[AnyType] DO
                       v3
                            ELSE ()

(3) e / data()       → FOR v IN e DO
                           match children(v)
                           case v : AnySimpleType DO v
                         ELSE ()

(4) WHERE e1 DO e2   → IF e1 THEN e2 ELSE ()
```

En la regla (1), la expresión de proyección `e/Wild` es reescrita como una expresión `FOR`, la cual liga la variable `v1` a cada elemento en el bosque de `e`. Para cada uno de los elementos ligados a `v1`, busca los hijos y los relaciona a la variable `v2`. Si el valor de `v2` es un elemento `Wild` lo devuelve como resultado.
De forma similar, la regla (2) reescribe la proyección de atributos `e/@Wild` como una expresión `match` embebida en una expresión `FOR`.
La regla (3) reescribe la expresión `e/data()` iterando sobre los items de `e`. La expresión `match` anidada devuelve los hijos de `e` si `e` es AnySimpleType. De lo contrario, devuelve la secuencia vacía.
La regla (4) simplemente reescribe la expresión `WHERE` como una expresión `IF - THEN - ELSE`.

3.13 Resumen

En este capítulo analizamos mediante ejemplos las distintas operaciones definidas en el álgebra estándar para XML. Mostramos la relación entre la proyección y la iteración, lo que nos permitió entender en detalle la forma en que se recorren los elementos del documento. Y, finalmente, detallamos las leyes que nos permiten reducir expresiones complejas en otras equivalentes más fáciles de manipular.

Capítulo 4

El lenguaje CXXML

Si bien hoy en día otros lenguajes de consulta como Lorel, UnQL y XQL realizan consultas sobre documentos XML, éstos no se basan en el álgebra estándar para XML propuesta por el World Wide Web Consortium (W3C) (www.w3.org). Además, no definen operadores para realizar ordenamiento de datos ni agrupamiento como así tampoco proyección de varios elementos y/o atributos.
Para solucionar los problemas antedichos, proponemos un nuevo lenguaje, *CXXML,* que permite recuperar información de documentos XML con una sintaxis similar a SQL. Está basado en el álgebra estándar para XML permitiendo una clara definición semántica del lenguaje y facilitando el proceso de optimización. El lenguaje propuesto es más poderoso que los lenguajes mencionados en el párrafo anterior, ya que permite realizar operaciones de ordenamiento, agregación y agrupamiento como así también proyección de varios elementos y/o atributos. CXXML no explota la información contenida en los DTDs asociados a los documentos XML. Esto permite realizar consultas sobre una amplia gama de documentos XML pero repercute en el análisis semántico de las sentencias.

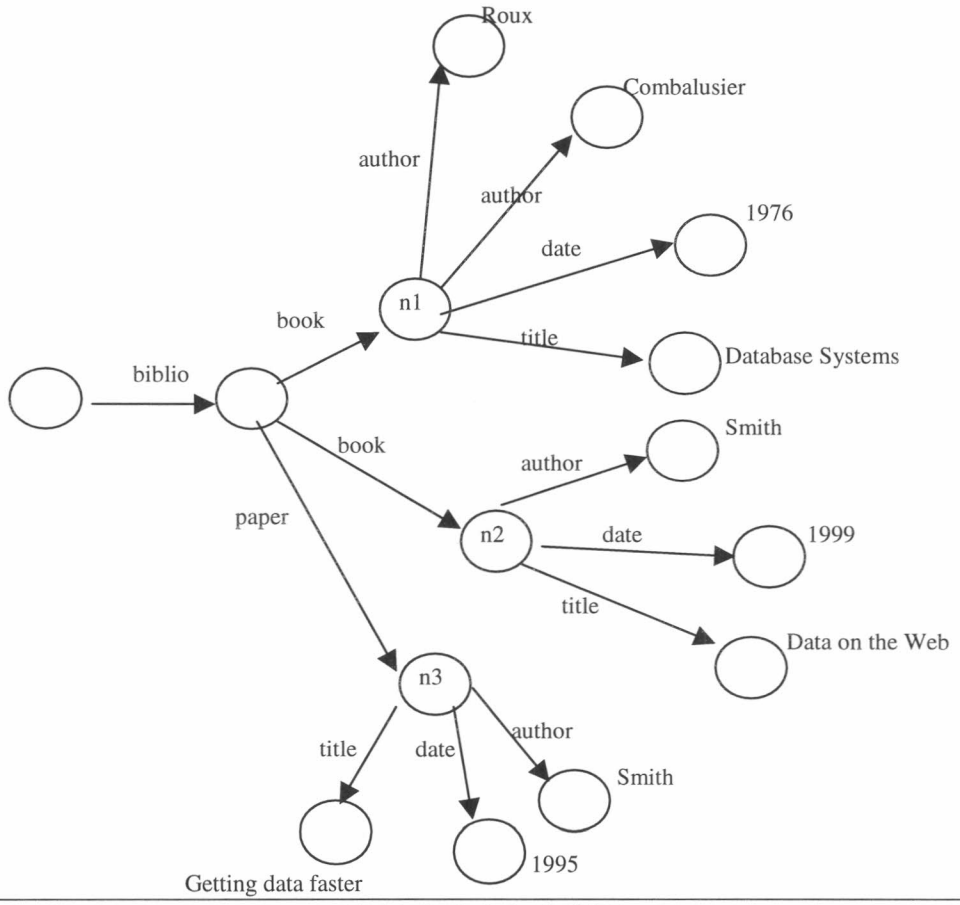
Esta sección explica la sintaxis de CXXML. Está dividida en dos partes, la primera describe el ejemplo que se tomará como base para explicar las sentencias; y la segunda describe en detalle cada una de las operaciones permitidas en el lenguaje. En el caso de que alguna de estas operaciones tenga validaciones de semántica especiales, las mismas se explicarán aparte.

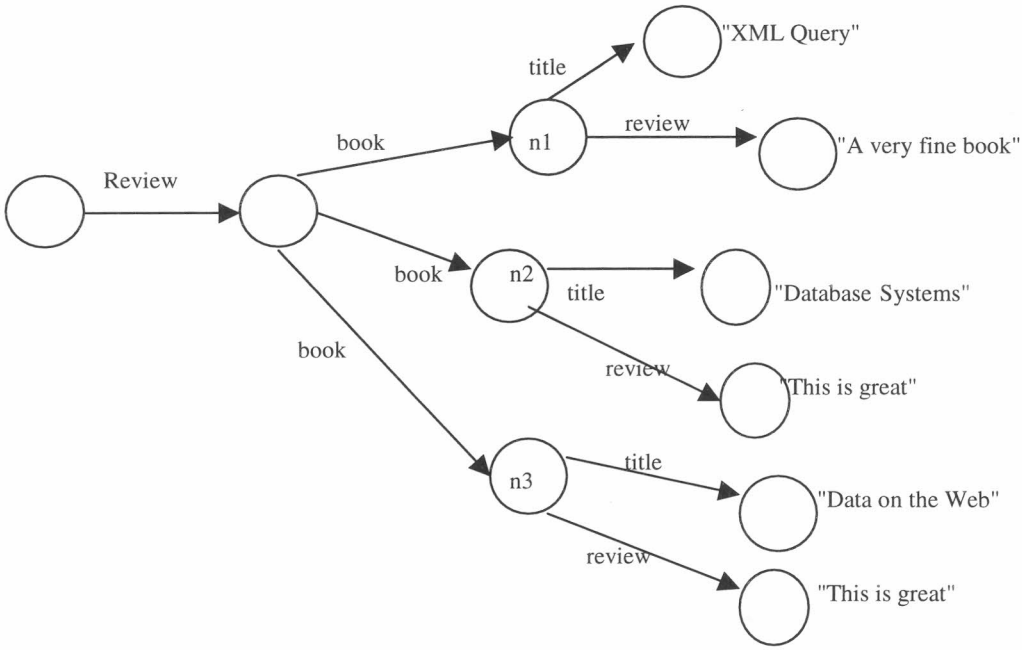
Para poder analizar la sintaxis y la semántica de nuestro lenguaje de consulta usaremos como ejemplo base el siguiente modelo:

```
<biblio>
     <book isbn="1-55860-622-X" country=USA>
          <author>Roux</author>
          <author>Combalusier</author>
          <date>1976</date>
          <title>Database Systems</title>
          <price>40.25</price>
     </book>
     <book isbn="1-55860-630-X" country=USA>
          <author>Smith</author>
          <date>1999</date>
          <title>Data on the Web </title>
          <price>83.00</price>
     </book>
     <paper>
          <author>Smith</author>
          <date>1995</date>
          <title>Getting data faster</title>
     </paper>
</biblio>
```

```
<review>
     <book>
          <title>XML Query</title>
          <review>A very fine book</review>
     </book>
     <book>
          <title>Database systems</title>
          <review>this is great</review>
     </book>
     <book>
          <title>Data on the Web</title>
          <review>this is great</review>
     </book>
</review>
```

El primer documento xml representa una base de libros almacenada en una biblioteca. En esta biblioteca se guarda el título del libro (`title`), su autor (`author`), el año de publicación (`date`) y el precio (`price`). Cada elemento book tiene a su vez dos atributos que indican el número de referencia de publicación (`isbn`) y el país de origen (`country`).
El segundo documento representa una crítica a una serie de libros. Sólo se guarda el título del libro (`title`) y su crítica (`review`).
Este modelo se puede representar gráficamente utilizando grafos de la siguiente forma:





La sintaxis del lenguaje CXXML está basada en la utilizada en SQL y tiene definidas las siguientes operaciones de consulta: proyección, selección, distinct, join, groupby, orderby, union e intersección. Además tiene las funciones de agregación sum, avg, max, min y count. Para más detalle referimos al lector a la sección *Apéndice*.
Como diferenciador de los demás lenguajes, CXXML permite realizar operaciones de ordenamiento, agregación y agrupamiento como así también proyección de varios elementos y/o atributos. Si bien en el álgebra sólo se menciona la proyección de un elemento o atributo, CXXML extiende el concepto a varios elementos y/o atributos.
Por ejemplo, en CXXML la selección de una lista de atributos se escribe de la siguiente forma:

```
Select @isbn, @country from /biblio/book
```

El símbolo "@" se utiliza para diferenciar los atributos de los elementos. Si quisieramos seleccionar una lista de elementos, deberíamos escribir:

```
Select author, date, title from /bilio/book
```

Una operación de agrupamiento en CXXML tendría la forma:

```
Select title, date from /biblio/book groupby author
```

También se puede asociar una función de agregación a la operación anterior:

```
Select count (author) from /biblio/book groupby author
```

Finalmente, la operación de ordenamiento se escribe en nuestro lenguaje de la siguiente forma:

```
Select title, date from /biblio/book orderby date
```

4.1 Proyección

En nuestro lenguaje de consulta definimos la proyección de la siguiente forma:

select L from P

Donde P es un path expression de la forma $/p_1/p_2/.../p_n$ y L es una lista de elementos o atributos de la forma $l_1, l_2, l_3, ...l_m$ que pertenecen a p_n. El * es un caso particular que se refiere a todos los elementos de p_n.

Ejemplo 1.
"Seleccionar todos los libros de la colección biblio"

```
select book
from /biblio
```

Según el álgebra mencionada en el capítulo [3] y teniendo en cuenta la sección [3.11] el código resultante para esta consulta será el siguiente:

```
for b1 in children(biblio) do
match b1
     case c1: book[AnyTypeof] do c1
     ELSE ()
```

Esta sentencia se traduce internamente en CXXML en el siguiente código:

```
(1)   b0_i = demo.xmlIn().getRootElement();
(2)   b1= demo.children(b0_i,"biblio");
(3)   FOR (int i1=0; i1<b1.size(); i1++){
    (4)    b1_i = (Element) b1.get(i1);
    (5)    b2= demo.children(b1_i);
    (6)    demo.crearElementoVacio();
    (7)    FOR (int i2=0; i2<b2.size(); i2++){
        (8)     b2_i = (Element) b2.get(i2);
        (9)     IF (demo.isTipoElemento(b2_i,"book")){
                (10) demo.asignarTMP(b2_i);}
    (11)   }
    (12)   demo.asignar();
(13)  }
(14)  demo.saveDatos("Query.xml");
```

La línea (1) relaciona la variable `b0_i` con la raíz del documento que se está analizando. El objeto `demo` es un objeto de la clase *cxxml.resultado*, que tiene como variables de instancia los documentos a consultar y el documento resultado.
La línea (2) busca todos los hijos del elemento "`biblio`" y los almacena en `b1`. A continuación se itera sobre `b1` (3) y se guarda en la variable `b1_i` el elemento que se encuentra en la posición `i1` de `b1` (4).
Para cada uno de los elementos encontrados se buscan los hijos y se guardan en la variable `b2` (5).

En la línea (6) se crea un nuevo elemento vacío en el que se colocará el resultado.
La línea (7) muestra cómo se itera sobre cada uno de los elementos contenidos en b2. Para cada uno de ellos (8), se pregunta si es el tipo de elemento buscado (9). De ser así, se lo asigna al elemento nuevo creado con anterioridad (10).
La línea (12) asigna el resultado final al nuevo elemento y la (14) guarda el resultado en formato XML.
Las líneas (11) y (13) sólo cierran los ciclos abiertos en líneas anteriores.

La respuesta a la consulta anterior es:

```
➔<root>
     <parent>
          <book>
               <author>Roux</author>
               <author>Combalusier</author>
               <date>1976</date>
               <title>Databse System</title>
          </book>
     </parent>
     <parent>
          <book>
               <author>Smith</author>
               <date>1999</date>
               <title>Data on the Web </title>
          </book>
     </parent>
</root>
```

El resultado de la consulta debe tener un elemento raíz, que nosotros denotamos <root>. También definimos para nuestro lenguaje el elemento <parent> que es utilizado a fines de contener en un solo elemento, el o los elementos proyectados.

Ejemplo 2.
"Obtener la fecha, título y autor de todos los libros que se encuentran en biblio"

```
select date, title, author
from /biblio/book
```

Lo que genera el siguiente código:

```
(1)  b0_i = demo.xmlIn().getRootElement();
(2)  b1= demo.children(b0_i,"biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
  (4)  b1_i = (Element) b1.get(i1);
  (5)  b2= demo.children(b1_i,"book");
  (6)  FOR (int i2=0; i2<b2.size(); i2++){
    (7)  b2_i = (Element) b2.get(i2);
    (8)  b3= demo.children(b2_i);
    (9)  demo.crearElementoVacio();
    (10) FOR (int i3=0; i3<b3.size(); i3++){
```

```
        (11) b3_i = (Element) b3.get(i3);
        (12) IF (demo.isTipoElemento(b3_i,"date")){
          (13) demo.asignarTMP(b3_i);}
        (14) IF (demo.isTipoElemento(b3_i,"title")){
          (15) demo.asignarTMP(b3_i);}
        (16) IF (demo.isTipoElemento(b3_i,"author")){
          (17) demo.asignarTMP(b3_i);}
      (18) }
      (19) demo.asignar();
    (20) }
(21) }
(22) demo.saveDatos("Query.xml");
```

Como en el ejemplo anterior, las líneas que van de la (1) a la (6) sirven para realizar la navegación en el documento hasta el nodo que estamos buscando. Una vez posicionados allí, se toman todos los hijos (7) y (8) y se itera sobre cada uno de ellos (10) y (11), buscando los elementos detallados en la sentencia. Si se encuentra alguno de esos elementos (líneas 12 a 17) se lo guarda en una variable temporal hasta que es guardado definitivamente en el elemento a grabar (19).

Nótese que tanto en este ejemplo como en el anterior se utiliza un método llamado `children` que puede recibir como parámetro un argumento o dos dependiendo de la utilización que le querramos dar. Si recibe un solo parámetro, es decir `children(x)`, devuelve todos los hijos del nodo `x`. Si recibe dos parámetros, `children(x,y)`, devuelve todos los hijos de `x` que son de tipo `y`.

La respuesta que generará CXXML será la siguiente:

```
➔<root>
      <parent>
            <date>1976</date>
            <title>"Database Systems"</title>
            <author>"Roux"</author>
            <author>"Combalusier"</author>
      </parent>
      <parent>
            <date>1999</date>
            <title>"Data on the Web"</title>
            <author>"Smith"</author>
      </parent>
</root>
```

Al comienzo de esta sección mencionamos que el lenguaje permitía realizar selecciones de elementos como así también de atributos. Veamos un ejemplo con atributos.

Ejemplo 3.
"Obtener el valor isbn de todos los libros que se encuentran en biblio"

```
select @isbn
from /biblio/book
```

Lo que genera el siguiente código:

```
(1)  String atrib = "";
(2)  e = demo.xmlIn().getRootElement();
(3)  b1 = demo.children(e, "Biblio");
(4)  FOR (int i1=0; i1<b1.size(); i1++) {
   (5)  b1_i = (Element) b1.get(i1);
   (6)  b2 = demo.children(b1_i,"Book");
   (7)  FOR (int i2=0; i2<b2.size(); i2++) {
      (8)  demo.crearElementoVacio();
      (9)  b2_i = (Element) b2.get(i2);
      (10) atrib = demo.atributo(b2_i, "isbn");
      (11) demo.asignarTMP("isbn",atrib);
      (12) demo.asignar();
   (13) }
(14) }
(15) demo.saveDatos("Query.xml");
```

Con respecto a los ejemplos anteriores, el ejemplo 3 muestra dos diferencias significativas. La primera es la línea (1) en la cual se inicializa una variable que luego será utilizada para la comparación de los atributos. La segunda son las líneas (10) y (11) que se utilizan solamente en el caso de atributos. La línea (10) busca el valor que contiene la variable b2_i y lo asigna a la variable atrib. La línea (11) asigna a la variable temporal el resultado anterior.

La respuesta generada por CXXML será:

```
➔<root>
     <parent isbn="1-55860-622-X" />
     <parent isbn="1-55860-630-X" />
</root>
```

Por último veremos un ejemplo de cómo buscar todos los elementos de un documento usando el operador *.

Ejemplo 4.
"Obtener todos los libros de la biblioteca"

```
select *
from /biblio/book
```

Lo que genera el siguiente código:

```
(1)  b0_i = demo.xmlIn().getRootElement();
(2)  b1= demo.children(b0_i,"biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
   (4)  b1_i = (Element) b1.get(i1);
   (5)  b2= demo.children(b1_i,"book");
```

```
   (6)  FOR (int i2=0; i2<b2.size(); i2++){
      (7)  b2_i = (Element) b2.get(i2);
      (8)  b3= demo.children(b2_i);
      (9)  demo.crearElementoVacio();
      (10) FOR (int i3=0; i3<b3.size(); i3++){
         (11) b3_i = (Element) b3.get(i3);
         (12) demo.asignarTMP(b3_i);}
         (13) demo.asignar();
   (14) }
(15) }
(16) demo.saveDatos("Query.xml");
```

La diferencia principal con los casos anteriores es que en este ejemplo, al estar buscando todos los elementos, no se contempla ningún caso particular dentro del ciclo de la línea (10).

4.2 Selección

En nuestro lenguaje de consulta definimos la selección de la siguiente forma:

select L from P where C

Donde P es un path expression de la forma $/p_1/p_2/.../p_n$ y L es una lista de elementos -o atributos- de la forma $l_1, l_2, l_3, ...l_m$ que pertenecen a p_n. El * es un caso particular que se refiere a todos los elementos de p_n. C es una condición lógica cuyos elementos deben pertenecer a p_n y es de la forma c_1 *op* c_2 *ob*...*ob* $c_{(n-1)}$ *op* c_n donde *op* es un operador lógico y *ob* es un operador booleano AND u OR.

Ejemplo 5.
"Obtener los títulos y autores de todos los libros que se encuentran en biblio cuya fecha de publicación sea mayor o igual a 1980"

```
select title, author
from /biblio/book where date>=1980
```

Lo que genera el siguiente código:

```
(1)  b0_i = demo.xmlIn().getRootElement();
(2)  b1= demo.children(b0_i,"biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
   (4)  b1_i = (Element) b1.get(i1);
   (5)  b2= demo.children(b1_i,"book");
   (6)  FOR (int i2=0; i2<b2.size(); i2++){
      (7)  b2_i = (Element) b2.get(i2);
      (8)  b3= demo.children(b2_i);
      (9)  IF ((demo.mayorIgual(b3,"date",1980)) ){
         (10) demo.crearElementoVacio();
         (11) FOR (int i3=0; i3<b3.size(); i3++){
```

```
            (12) b3_i = (Element) b3.get(i3);
            (13) IF (demo.isTipoElemento(b3_i,"title")){
               (14) demo.asignarTMP(b3_i);}
            (15) IF (demo.isTipoElemento(b3_i,"author")){
               (16)    demo.asignarTMP(b3_i);}
         (17) }
         (18) demo.asignar();
      (19) }
   (20) }
(21) }
(22)  demo.saveDatos("Query.xml");
```

En el ejemplo 4, la principal diferencia con los demás es la línea (9) en donde se establece la condición del `where` de la sentencia que se está ejecutando. Ahí se define el método `mayorIgual()` que recibe como parámetros el nombre del campo que se está comparando (en nuestro ejemplo "`date`") y el valor contra el cual se realiza la comparación (en nuestro ejemplo `1980`).
El resto de las líneas mantienen la misma secuencia que en la proyección.

La respuesta generada por CXXML será:

```
➔<root>
     <parent>
          <title>"Data on the Web"</title>
          <author>"Smith"</author>
     </parent>
</root>
```

4.3 Distinct

Veamos ahora cómo seleccionar los elementos L de un documento XML que se encuentra en el path P cumpliendo con la condición C y pidiendo además que no haya elementos repetidos. Sea P un path expression de la forma $/p_1/p_2/.../p_n$, L una lista de elementos de la forma $l_1, l_2, l_3, ...l_m$ que pertenecen a p_n y C una condición lógica cuyos elementos deben pertenecer a p_n y ser de la forma mencionada en el punto 4.2. La condición lógica C no es requisito indispensable de la operación *distinct* pero la incluímos a los efectos prácticos.

select distinct L from P where C

Ejemplo 6.
"Seleccionar los distintos autores de libros cuya fecha de edición sea mayor a 1969."

```
select distinct author
from /biblio/book
where date >= 1970
```

Esta sentencia genera el siguiente código:

```
(1)  b0_i = demo.xmlIn().getRootElement();
(2)  b1= demo.children(b0_i,"biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
   (4)  b1_i = (Element) b1.get(i1);
   (5)  b2= demo.children(b1_i,"book");
   (6)  FOR (int i2=0; i2<b2.size(); i2++){
      (7)  b2_i = (Element) b2.get(i2);
      (8)  b3= demo.children(b2_i);
      (9)  IF ((demo.mayorIguall(b3,"date",1970)) ){
         (10) demo.crearElementoVacio();
         (11) FOR (int i3=0; i3<b3.size(); i3++){
            (12) b3_i = (Element) b3.get(i3);
            (13) IF (demo.isTipoElemento(b3_i,"author")){
               (14) demo.asignarTMP(b3_i);}
         (15) }
         (16) demo.asignar();
      (17) }
   (18) }
(19) }
(20) demo.distinct();
(21) demo.saveDatos("Query.xml");
```

Así como en el álgebra se define una función distinct() que recibe como parámetro un *árbol* y devuelve un árbol sin elementos repetidos (ver 3.4), en CXXML al finalizar la operación de proyección o selección (pasos 1-19) se realiza la operación `distinct()` que recorre el resultado almacenado en el objeto `demo` y saca los valores repetidos. Para más información acerca del algoritmo `distinct()` usado, referimos al lector a la sección [5.3] del capítulo *Implementación*.

La respuesta CXXML será:

```
➔<root>
     <parent>
          <author>Roux</author>
          <author>Combalusier</author>
          <author>Smith</author>
     </parent>
</root>
```

4.4 Join

En CX-XML el join entre dos documentos XML se escribe de la siguiente forma:

select L from A.P_1, B.P_2 where C

Siendo P_1 y P_2 path expressions, de la forma $/p_1/p_2/.../p_n$ y $/q_1/q_2/.../q_m$, que están ligados a las variables A y B respectivamente. La condición C es una operación lógica binaria entre un elementos de P_1 y uno de P_2 de la forma a.l_1<oplógico>b.l_2 y L es la lista de elementos de la forma $a.l_1,\ a.l_2,...,b.l_{q-1},\ b.l_q$

Ejemplo 7.

"Supongamos que queremos armar un nuevo documento que tenga los datos de los libros más su crítica."

```
select a.title, a.author, b.review
from a./biblio/book, b./review/book
where a.title=b.title
```

Esta sentencia de nuestro lenguaje, genera el siguiente código Java:

```
(1)  b0_i = demo.xmlIn().getRootElement();
(2)  c0_i = demo.xmlIn2().getRootElement();
(3)  b1= demo.children(b0_i,"biblio");
(4)  FOR (int i1=0; i1<b1.size(); i1++){
  (5)  b1_i = (Element) b1.get(i1);
  (6)  b2= demo.children(b1_i,"book");
  (7)  FOR (int i2=0; i2<b2.size(); i2++){
    (8)  b2_i = (Element) b2.get(i2);
    (9)  b3= demo.children(b2_i);
    (10) c1= demo.children(c0_i,"review");
    (11) FOR (int j1=0; j1<c1.size(); j1++){
      (12) c1_i = (Element) c1.get(j1);
      (13) c2= demo.children(c1_i, "book");
      (14) FOR (int j2=0; j2<c2.size(); j2++){
        (15) c2_i = (Element) c2.get(j2);
        (16) c3= demo.children(c2_i);
        (17) IF((demo.igual(b3,"title",c3,"title")) ){
          (18) demo.crearElementoVacio();
          (19) FOR (int i3=0; i3<b3.size(); i3++){
            (20) b3_i = (Element) b3.get(i3);
            (21) IF (demo.isTipoElemento(b3_i,"title")){
              (22) demo.asignarTMP(b3_i);}
            (23) IF
              (demo.isTipoElemento(b3_i,"author")){
              (24) demo.asignarTMP(b3_i);}
          (25) }
          (26) FOR (int j3=0; j3<c3.size(); j3++){
            (27) c3_i = (Element) c3.get(j3);
            (28) IF
              (demo.isTipoElemento(c3_i,"Review")){
              (29) demo.asignarTMP(c3_i);}
          (30) }
          (31) demo.asignar();
        (32) }
      (33) }
    (34) }
```

```
   (35) }
(36)  }
(37)  demo.saveDatos("Query.xml");
```

La implementación consiste en recorrer el primer árbol hasta llegar al nivel buscado (líneas 1 a 8) y a continuación, hacer lo mismo con el segundo árbol (líneas 9 a 15) Una vez posicionados en dichos nodos, se realiza una comparación tomando la condición definida en el `where` de la sentencia (16). Si la condición se cumple, se crea un nuevo elemento vacío (17) y se recorre cada uno de los documentos buscando los elementos `title` y `author` (líneas 18 a 24) y `review` (líneas 25 a 29) respectivamente.
Nótese que el orden del resultado viene dado por el orden de los *for,* a diferencia del álgebra relacional en donde el join es conmutativo.

La respuesta que generará CXXML será la siguiente:

```
➔<root>
     <parent>
          <title>Database System</title>
          <author>Roux</ author>
          <author>Combalusier</ author>
          <review>This is great</review>
     </parent>
</root>
```

4.5 Group_by

Veamos cómo seleccionar de un documento XML los elementos L, de la forma l_1, l_2, l_3,...l_m, que se encuentran en el path expression P, de la forma $/p_1/p_2/.../p_n$, y que pertenecen a p_n, , agrupados por un elemento G_1 que pertenece a p_n y está incluído en L. Tanto en el álgebra como en CXXML se permite solo un elemento asociado a la función de agrupación.
A diferencia de la operación definida en el álgebra relacional, en CXXML el operador groupby permite realizar adicionalmente "cortes de control". En SQL, la operación de agrupamiento siempre se utiliza con una función de agregación asociada mientras que en nuestro caso esto no es necesario lo que aumenta el poder expresivo del lenguaje.

select L from P groupby G_1

Ejemplo 8.
"Supongamos que queremos seleccionar todos los autores de libros y sus títulos que se encuentran en la biblioteca agrupados por su autor."

```
select author, title
from /biblio/book
groupby author
```

Que a su vez se traduce en el siguiente código:

```
(1)  b0_i = demo.xmlIn().getRootElement();
(2)  b1= demo.children(b0_i,"biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
   (4)  b1_i = (Element) b1.get(i1);
   (5)  b2= demo.children(b1_i,"book");
   (6)  FOR (int i2=0; i2<b2.size(); i2++){
      (7)  b2_i = (Element) b2.get(i2);
      (8)  b3= demo.children(b2_i);
      (9)  demo.crearElementoVacio();
      (10) FOR (int i3=0; i3<b3.size(); i3++){
         (11) B3_i = (Element) b3.get(i3);
         (12) IF (demo.isTipoElemento(b3_i,"author")){
            (13) demo.asignarTMP(b3_i);}
      (14) }
      (15) demo.asignar();
   (16) }
(17) }
(18) demo.distinct();
(19) demo.setXmlIn2RootElement(demo.xmlOut().getRootElement
   ());
(20) demo.setXmlOut(new Document(new Element("root")));
(21) e1 = demo.xmlIn().getRootElement();
(22) e2 = demo.xmlIn2().getRootElement();
(23) b0 = demo.children(e2);
(24) FOR (int i0=0; i0<b0.size(); i0++){;
   (25) b0_i = (Element) b0.get(i0);
   (26) demo.crearElementoVacio();
   (27) demo.asignarTMP(b0_i);
   (28) b1= demo.children(e1,"biblio");
   (29) FOR (int i1=0; i1<b1.size(); i1++){
      (30) b1_i = (Element) b1.get(i1);
      (31) b2= demo.children(b1_i, "book");
      (32) FOR (int i2=0; i2<b2.size(); i2++){
         (33) b2_i = (Element) b2.get(i2);
         (34) b3 = demo.children(b2_i);
         (35) IF ((demo.igual(b3,"author",b0_i)) ){
            (36) FOR (int i3=0; i3<b3.size(); i3++){
               (37) b3_i= (Element) b3.get(i3);
               (38) IF (demo.isTipoElemento(b3_i,"title")){
                  (39) demo.asignarTMP(b3_i);}
            (40) }
         (41) }
      (42) }
   (43) }
   (44) demo.asignar();
(45) }
(46)  demo.saveDatos("Query.xml");
```

Se puede ver que en este caso la operación se divide en dos partes. La primera consiste en realizar una proyección del elemento `author` y al resultado aplicarle la operación de *distinct()* (líneas 1 a 18) Esta operación sirve para obtener todos los elementos `author` no repetidos lo que nos servirá para la segunda parte.
En las lineas (19) y (20) se pasa el documento obtenido en la consulta anterior como segundo documento a consultar (`xmlIn2`) y se inicializa el documento sobre el cual volcaremos el resultado.
La segunda parte (línea (25) a la (42)) consiste en realizar una selección de los títulos correspondientes a los autores obtenidos en la consulta anterior. Para ello, se recorre el primer documento `xmlIn1`, tantas veces como elementos `author` haya en `xmlIn2`, seleccionando aquellos elementos `title` que aparezcan en el path `/biblio/book` y cuyo autor (elemento `author`) sea igual (35) al correspondiente a la iteración actual (b0_i).

De esta forma, el resultado queda ordenado por el elemento de agrupamiento.

La respuesta CXXML será:

```
➔<root>
     <parent>
          <author>Roux</author>
          <title>Database System</title>
     </parent>
     <parent>
          <author>Combalusier</author>
          <title>Database System</title>
     </parent>
     <parent>
          <author>Smith</author>
          <title>Data on the Web</title>
          <title>Getting Data Faster</title>
     </parent>
</root>
```

Ejemplo 9.
"Obtener la cantidad de autores y el título de cada libro agrupado por su título."

```
select title, count(author)
from /biblio/book
groupby title
```

Esto genera el siguiente código:

```
(1)  e = demo.xmlIn().getRootElement();
(2)  b1 = demo.children(e, "Biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++) {
  (4)  b1_i = (Element) b1.get(i1);
  (5)  b2= demo.children(b1_i,"book");
  (6)  FOR (int i2=0; i2<b2.size(); i2++){
    (7)  b2_i = (Element) b2.get(i2);
```

```
      (8)  b3= demo.children(b2_i);
      (9)  demo.crearElementoVacio();
      (5)  FOR (int i3=0; i3<b3.size(); i3++) {
        (6)  B3_i = (Element) b3.get(i3);
        (7)  IF (demo.isTipoElemento(b3_i,"title")){
          (8)  demo.asignarTMP(b3_i);}
      (9)  }
   (10) }
   (11) demo.asignar();
(12) }
(13) demo.distinct();
(14) demo.setXmlIn2RootElement(demo.xmlOut().getRootElement
   ());
(15) demo.setXmlOut(new Document(new Element("root")));
(16) e1 = demo.xmlIn().getRootElement();
(17) e2 = demo.xmlIn2().getRootElement();
(18) b0 = demo.children(e2);
(19) float agg_variable = 0;
(20) float count_avg_elementos = 0;
(21) FOR (int i0=0; i0<b0.size(); i0++) {
   (22) b0_i = (Element) b0.get(i0);
   (23) demo.crearElementoVacio();
   (24) demo.asignarTMP(b0_i);
   (25) b1 = demo.children(e1, "biblio");
   (26) agg_variable = demo.setear_var_agregacion("count");
   (27) count_avg_elementos = 0;
   (28) FOR (int i1=0; i1<b1.size(); i1++) {
     (29) b1_i = (Element) b1.get(i1);
     (30) b2 = demo.children(b1_i);
     (31) IF (demo.igual(b2,"title", b0_i)){
       (32) FOR (int i2=0; i2<b2.size(); i2++) {
         (33) b2_i = (Element) b2.get(i2);
         (34) IF (demo.isTipoElemento(b2_i,"author")){
           (35) agg_variable =
             demo.agg_operacion("count", b2_i ,
             agg_variable);
         (36) }
     (37)    }
     (38) }
   (39) }
   (40) demo.asignarElementoAgregacion("count",agg_variable
     );
   (41) demo.asignar();
(42) }
(43) demo.saveDatos("Query.xml");
```

La primera parte de la consulta es idéntica al ejemplo anterior. Sin embargo, la segunda parte tiene algunas diferencias introducidas por la función de agregación. La primer diferencia aparece en las líneas (20) y (21) en donde se inicializan los contadores necesarios para la operación. La segunda se encuentra en la línea (27), en donde se inicializa para cada valor de `title`, la variable `agg_variable`, según el tipo de

operación a realizar. En este caso que estamos trabajando con un *count*, toma el valor 0. El resto de las funciones se inicializan con este valor, salvo la funcion *max*, que toma el maximo valor número que soporte la plataforma usada.
En las línea (36) se modifica el valor de la variable en cuestión, incrementando en una unidad el valor de la variable `agg_variable`, en nuestro ejemplo *count*. Para el resto de las funciones, la modificación de dicho valor estará relacionado con el tipo de función de agregación elegida.
Al finalizar el ciclo que cuenta los autores de un determinado titulo, en la línea (41), se crea un elemento nuevo, que toma como nombre `count`, y como contenido el valor de `agg_variable`.
La segunda parte (línea (25) a la (42)) consiste en realizar una selección de los títulos correspondientes a los autores obtenidos en la consulta anterior. Para ello, se recorre el primer documento `xmlIn1`, tantas veces como elementos `author` haya en `xmlIn2`, seleccionando aquellos elementos `title` que aparezcan en el path `/biblio/book` y cuyo autor (elemento `author`) sea igual (35) al correspondiente a la iteración actual (b0_i).

La respuesta que generará esta consulta será:

```
➔<root>
     <parent>
          <title>Database System</title>
          <count>2</count>
     </parent>
     <parent>
          <title>Data on the Web</title>
          <count>1</count>
     </parent>
</root>
```

4.6 Order_By

Para seleccionar de un documento XML los elementos L, de la forma l_1, l_2, l_3,...l_m que se encuentra en el path expression P, de la forma $/p_1/p_2/.../p_n$ y pertenecen a p_n , ordenados por un elemento L_1 que pertenece a p_n y que está incluído en L realizamos lo siguiente:

select L from P orderby L_1

Ejemplo 10.
"Obtener todas las críticas de los libros ordenadas por el título de los mismos."

```
select review, title
from /review/book
orderby title
```

El código que genera esta consulta es:

```
(1)  b0_i = demo.xmlIn().getRootElement();
(2)  b1= demo.children(b0_i,"Review");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
   (4)  b1_i = (Element) b1.get(i1);
   (5)  b2= demo.children(b1_i,"Book");
   (6)  FOR (int i2=0; i2<b2.size(); i2++){
      (7)  b2_i = (Element) b2.get(i2);
      (8)  b3= demo.children(b2_i);
      (9)  demo.crearElementoVacio();
      (10) FOR (int i3=0; i3<b3.size(); i3++){
         (11) b3_i = (Element) b3.get(i3);
         (12) IF (demo.isTipoElemento(b3_i,"Review")){
            (13) demo.asignarTMP(b3_i);}
         (14) IF (demo.isTipoElemento(b3_i,"Title")){
            (15) demo.asignarTMP(b3_i);}
         (16) demo.asignar();
      (17) }
   (18) }
(19) }
(20) demo.sort("Title");
(21) demo.saveDatos("Query.xml");
```

Se puede ver claramente que el código generado por CXXML se divide en dos partes: la primera realiza la *selección* de los elementos (líneas 1 a 19) y la segunda realiza un *sort()* al resultado anterior tomando como parámetro de entrada el elemento sobre el cual se realizará el ordenamiento (20). De esta forma se logra reutilizar el código (en la generación de la selección) y alinear el lenguaje con el álgebra (al definir un método *sort*).
Para más información acerca del algoritmo `sort()` usado, referimos al lector a la sección [5.7] del capítulo *Implementación*.

La respuesta CXXML será la siguiente:

```
➔<root>
     <parent>
          <title>Data on the Web</title>
          <review>This is great</review>
     </parent>
     <parent>
          <title>Database Systems</title>
          <review>This is great!</review>
     </parent>
     <parent>
          <title>XML Query</title>
          <review>A very fine book</review>
     </parent>
</root>
```

4.7 Union

La operación de unión se escribe en nuestro lenguaje de consulta de la siguiente forma:

select L from P Union Select J from R

Siendo L y J listas de elementos de la forma $l_1, l_2, l_3,...l_r$ y $j_1, j_2, j_3,...j_s$ respectivamente que se encuentran en los path expressiona P y R, de la forma $/p_1/p_2/.../p_n$ y $/r_1/r_2/.../r_m$ y pertenecen a p_n y r_m.

Ejemplo 11.
"Listar el título y la crítica de todos los libros que se encuentran en biblio y review"

```
     select title
     from /biblio/book
     union
     select review
     from /review/book

(1)  b0_i = demo.xmlIN().getRootElement();
(2)  b1= demo.children(b0_i,"biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
  (4)  b1_i = (Element) b1.get(i1);
  (5)  b2= demo.children(b1_i);
  (6)  demo.crearElementoVacio();
  (7)  FOR (int i2=0; i2<b2.size(); i2++){
    (8)  b2_i = (Element) b2.get(i2);
    (9)  IF (demo.isTipoElemento(b2_i,"title")){
      (10) demo.asignarTMP(b2_i);}
  (11) }
  (12) demo.asignar();
(13) }
(14) Document xmlTemp;
(15) xmlTemp = new Document(new Element("root"));
(16) xmlTemp.setRootElement(demo.xmlOUT().getRootElement())
  ;
(17) demo.setXmlOut(new Document(new Element("root")));
(18) demo.setXmlInRootElement(demo.xmlIN2().getRootElement(
  ));
(19) b1= demo.children(b0_i,"review");
(20) FOR (int i1=0; i1<b1.size(); i1++){
  (21) b1_i = (Element) b1.get(i1);
  (22) b2= demo.children(b1_i);
  (23) demo.crearElementoVacio();
  (24) FOR (int i2=0; i2<b2.size(); i2++){
    (25) b2_i = (Element) b2.get(i2);
    (26) IF (demo.isTipoElemento(b2_i,"review")){
      (27) demo.asignarTMP(b2_i);}
  (28) }
```

```
   (29) demo.asignar();
(30) }
(31) demo.setXmlIn2RootElement(demo.xmlOUT().getRootElement
   ());
(32) demo.setXmlInRootElement(xmlTemp.getRootElement());
(33) listIn = demo.xmlIN().getRootElement().getChildren();
(34) listIn2 =
   demo.xmlIN2().getRootElement().getChildren();
(35) demo.union(listIn,listIn2);
(36) demo.saveDatos("Query.xml");
```

Se puede ver que el código generado por CXXML se divide en tres partes: la primera (líneas 1 a 13) realiza el primer query para obtener todos los títulos de los libros ubicados en `biblio`; la segunda (líneas 19 a 30) realiza el segundo query para obtener todas las críticas de `review`; y la tercera consiste en invocar el método `union()` (35) recibiendo como parámetro los arboles obtenidos en los pasos anteriores y generando un único árbol final con la union de ellos.
Debemos resaltar dos detalles en la implementación: existe un paso intermedio entre las líneas (14) y (18), en el que el documento resultado `xmlOut`, es copiado a un documento XML temporal llamado `xmlTemp`. Luego se carga en `xmlIn1` el documento `xmlIn2`, para procesar la segunda parte de la consulta de `review` . Finalmente, entre las lineas (31) y (34), antes de llamar a la función de `union()`, se reordena las variables de instancia del objeto `demo`, correspondiente a la clase `cxxml.resultado`. En `xmlIn1`, queda el resultado de la primer consulta (32) y en `xmlIn2` el correspondiente resultado de la segunda consulta (31). Luego en (33) y (34) se cargan ambos documentos en listas para realizar la union en (35). Este manejo con listas se debe a que JDOM referencia a documentos mediante listas.
Para más información acerca del algoritmo `union()` usado, referimos al lector a la sección [5.4] del capítulo *Implementación*.
La operación de unión para XML, basada en la idea de manejar datos semi-estructurados, permite unir estructuras de documentos totalmente diferentes, por lo que no queda restringido la aplicación de esta operación a elementos del mismo tipo.

La respuesta CXXML será la siguiente:

```
➔<root>
     <parent>
          <title>Database System</title>
     </parent>
     <parent>
          <title>Data on the Web</title>
     </parent>
     <parent>
          <title>XML Query</title>
     </parent>
     <parent>
          <review>This is great</review>
     </parent>
     <parent>
          <review>This is great!</review>
```

```
        </parent>
        <parent>
            <review>A darn fine book</review>
        </parent>
    </root>
```

En el resultado anterior la semántica del lenguaje permitió la unión disjunta de estructuras distintas. En la sección 6 de experimentos se verán más ejemplos de esta operación.

4.8 Intersection

La operación de intersección se escribe en nuestro lenguaje de consulta de la siguiente forma:

select L from P intersection Select J from R

Siendo L y J listas de elementos de la forma l_1, l_2, l_3,...l_r y j_1, j_2, j_3,...j_s respectivamente que se encuentran en los path expressiona P y R, de la forma $/p_1/p_2/.../p_n$ y $/r_1/r_2/.../r_m$ y pertenecen a p_n y r_m.

Ejemplo 12.
"Obtener los libros que se encuentran en los documentos biblio y review"

```
select book
from /biblio
intersection
select book
from /review
```

```
(1)  b0_i = demo.xmlIN().getRootElement();
(2)  b1= demo.children(b0_i,"biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
  (4)  b1_i = (Element) b1.get(i1);
  (5)  b2= demo.children(b1_i);
  (6)  demo.crearElementoVacio();
  (7)  FOR (int i2=0; i2<b2.size(); i2++){
    (8)  b2_i = (Element) b2.get(i2);
    (9)  IF (demo.isTipoElemento(b2_i,"book")){
      (10) demo.asignarTMP(b2_i);}
  (11) }
  (12) demo.asignar();
(13) }
(14) Document xmlTemp;
(15) xmlTemp = new Document(new Element("root"));
(16) xmlTemp.setRootElement(demo.xmlOUT().getRootElement())
  ;
(17) demo.setXmlOut(new Document(new Element("root")));
```

```
(18) demo.setXmlInRootElement(demo.xmlIN2().getRootElement(
  ));
(19) b1= demo.children(b0_i,"review");
(20) FOR (int i1=0; i1<b1.size(); i1++){
   (21) b1_i = (Element) b1.get(i1);
   (22) b2= demo.children(b1_i);
   (23) demo.crearElementoVacio();
   (24) FOR (int i2=0; i2<b2.size(); i2++){
      (25) b2_i = (Element) b2.get(i2);
      (26) IF (demo.isTipoElemento(b2_i,"book")){
         (27) demo.asignarTMP(b2_i);}
   (28) }
   (29) demo.asignar();
(30) }
(31) demo.setXmlIn2RootElement(demo.xmlOUT().getRootElement
  ());
(32) demo.setXmlInRootElement(xmlTemp.getRootElement());
(33) listIn = demo.xmlIN().getRootElement().getChildren();
(34) listIn2=demo.xmlIN2().getRootElement().getChildren();
(35) demo.intersection(listIn,listIn2);
(36) demo.saveDatos("Query.xml");
```

La única diferencia entre este ejemplo y el anterior es la línea (35) en la cual se invoca al método `intersection()` pasandole como parámetros los dos arboles que surjen de realizar las respectivas proyecciones sobre ambos documentos.
Para más información acerca del algoritmo `intersection()` usado, referimos al lector a la sección [5.5] del capítulo *Implementación*.

La respuesta CXXML será la siguiente:

```
➔<root>
     <parent>
          <title>Database System</title>
     </parent>
     <parent>
          <title>Data on the Web</title>
     </parent>
     <parent>
          <title>XML Query</title>
     </parent>
  </root>
```

El resultado obtenido, muestra los 3 libros que existen en ambos documentos. En la sección 6 de experimentos se verán más ejemplos de esta operación.

4.9 Funciones de Agregacion : Count, Min, Max, Sum, Avg

Como vimos en la sección 3.8, el álgebra para XML define 5 funciones built-in de agregación : avg, count, max, min y sum. Todas las funciones toman un bosque con un campo repetitivo y retornan un valor entero.

En el siguiente ejemplo, la función `count()` retorna el número de elementos del bosque.

Ejemplo 13.
"Contar la cantidad de libros que figuran en la base de la biblioteca"

```
select count(book)
from /biblio
```

El código generado a partir de esta consulta es:

```
(1)  b0_i = demo.xmlIn().getRootElement();
(2)  b1= demo.children(b0_i,"biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
  (4)  b1_i = (Element) b1.get(i1);
  (5)  b2= demo.children(b1_i);
  (6)  demo.crearElementoVacio();
  (7)  FOR (int i2=0; i2<b2.size(); i2++){
    (8)  b2_i = (Element) b2.get(i2);
    (9)  IF (demo.isTipoElemento(b2_i,"book")){
    (10)      demo.asignarTMP(b2_i);}
  (11) }
  (12) demo.asignar();
(13) }
(14) demo.count("book");
(15) demo.saveDatos("Query.xml");
```

El lector puede apreciar que la única diferencia entre éste ejemplo y el de la proyección de la sección 4.1, es la línea (14) en la cual se invoca al método `count()` pasándole como parámetro el nombre del campo que queremos contar.

La respuesta CXXML será:

```
<count>2</count>
```

Ejemplo 14.
"Sumar el precio de todos los libros de la biblioteca".

```
Select sum(Price)
from /biblio/book
```

El código que genera esta consulta es:

```
(1)  b0_i = demo.xmlIn().getRootElement();
(2)  b1= demo.children(b0_i,"biblio");
(3)  FOR (int i1=0; i1<b1.size(); i1++){
```

```
(4)  b1_i = (Element) b1.get(i1);
(5)  b2= demo.children(b1_i, "book");
(6)  FOR (int i2=0; i2<b2.size(); i2++){
   (7)  b2_i = (Element) b2.get(i2);
   (8)  b3= demo.children(b2_i);
   (9)  demo.crearElementoVacio();
   (10) FOR (int i3=0; i3<b3.size(); i3++){
      (11) b3_i = (Element) b3.get(i3);
      (12) IF (demo.isTipoElemento(b3_i,"Price")){
      (13)    demo.asignarTMP(b3_i);}
   (14) }
   (15) demo.asignar();
(16) }
(17) }
(18) demo.sum();
(19) demo.saveDatos("Query.xml");
```

La respuesta será:

```
<Sum>123.25</Sum>
```

En (14) finalizada ya la iteración, el método `sum()` crea un elemento etiquetado con el tipo de función de agregación solicitada y se le incluye como valor del elemento, la suma de todos los valores obtenidos.

El resto de las funciones de agregación se plantean igual, reciben un bosque como entrada y se itera sobre el campo solicitado, creando como resultado un documento de un único elemento, rotulado con el nombre de la función de agregación y tomando com valor el resultado de dicha operación.

4.10 Generación de Código

Los ejemplos anteriores muestran intuitivamente la generación del código fuente a partir de una sentencia CXXML. En esta sección formalizaremos lo anterior.

Como mostramos en la sección [3.11] existe una relación entre el path expression que estamos analizando y los distintos ciclos `FOR` que se generan a partir de él. Por cada nivel que se recorre, se genera un ciclo `FOR` que itera sobre los elementos de ese nivel. Teniendo en cuenta esto y los ejemplos mostrados en la sección [4.1] podemos inferir que el código generado para una *proyección* de la forma:

select L from P

donde P es un path expression de la forma $/p_1/p_2/.../p_n$ y L es una lista de elementos o atributos de la forma $l_1, l_2, l_3, ...l_m$ que pertenecen a p_n, es el siguiente:

```
b0_i = demo.xmlIn().getRootElement();
b1= demo.children(b0_i,"p1");
FOR (int i1=0; i1<b1.size(); i1++){
      b1_i = (Element) b1.get(i1);
      b2= demo.children(b1_i,"p2");
      FOR (int i2=0; i2<b2.size(); i2++){
            b2_i = (Element) b2.get(i2);
            ...
            bn= demo.children(b1_i,"pn");
            FOR (int in=0; in<bn.size(); in++){
                  bn_i = (Element) bn.get(in);
                  bm= demo.children(bn_i);
                  demo.crearElementoVacio();
                  FOR (int im=0;im<bm.size(); im++){
                        bm_i = (Element) bm.get(im);
                        IF (demo.isTipoElemento(bm_i,"l1")){
                              demo.asignarTMP(bm_i);}
                        IF (demo.isTipoElemento(bm_i,"l2")){
                              demo.asignarTMP(bm_i);}
                        ...
                        IF (demo.isTipoElemento(bm_i,"lm")){
                              demo.asignarTMP(bm_i);}
                  }
                  demo.asignar();
            }
            ...
      }
}
demo.saveDatos("Query.xml");
```

Nótese que por cada elemento perteneciente al path expression de la cláusula `FROM` se genera un ciclo `FOR` que recorre todos los hijos de ese elemento, obtenidos mediante el método `children()`. Una vez que nos posicionamos en el último elemento del path expression se generan tantas sentencias `IF` como elementos se pidan en la cláusula `SELECT`. Esto difiere del álgebra en que no usamos sentencias `match` y `case` dado que no son soportadas por el lenguaje de implementación. Por tal motivo fueron reemplazadas por sentencias `IF`.
Como veremos a continuación la operación de *selección* está íntimamente relacionada con la *proyección*. Si tomamos una sentencia de la forma:

select L from P where C

el código generado es el siguiente:

```
b0_i = demo.xmlIn().getRootElement();
b1= demo.children(b0_i,"p1");
FOR (int i1=0; i1<b1.size(); i1++){
      b1_i = (Element) b1.get(i1);
      b2= demo.children(b1_i,"p2");
      FOR (int i2=0; i2<b2.size(); i2++){
            b2_i = (Element) b2.get(i2);
            ...
            bn= demo.children(b1_i,"pn");
```

```
            FOR (int in=0; in<bn.size(); in++){
                  bn_i = (Element) bn.get(in);
                  bm= demo.children(bn_i);
                  IF (C){
                        demo.crearElementoVacio();
                        FOR (int im=0;im<bm.size(); im++){
                              bm_i = (Element) bm.get(im);
                              IF (demo.isTipoElemento(bm_i,"l1")){
                                    demo.asignarTMP(bm_i);}
                              IF (demo.isTipoElemento(bm_i,"l2")){
                                    demo.asignarTMP(bm_i);}
                              ...
                              IF (demo.isTipoElemento(bm_i,"lm")){
                                    demo.asignarTMP(bm_i);}
                        }
                        demo.asignar();
                  }
            }
            ...
      }
}
demo.saveDatos("Query.xml");
```

Nótese que la estructura del código generado es la misma que la utilizada en la proyección con la diferencia del agregado de la condición del `where` antes del último `FOR`.
Esto muestra una característica muy importante en la generación de código: la *reutilización*. Esta característica facilita la generación de código para el resto de las operaciones como veremos a continuación.

La generación de código para la operación *distinct* reutiliza lo realizado para la proyección o selección, según el caso, y agrega al final una sentencia `demo.distinct()` -ver ejemplo 6 de la sección [4.3]- que es la que finalmente elimina los elementos repetidos.

En el caso del *Join* la generación es más complicada pero mantiene lo mencionado anteriormente. Teniendo en cuenta que se accede a dos paths, el programa genera dos proyecciones anidadas: la primera para posicionarse en el nodo correspondiente al primer path, y la segunda para realizar lo mismo con el segundo path. Una vez hecho esto, se genera la condición como en el caso de la selección y finalmente se realiza la selección de los elementos.

Para la generación del groupby primero se genera el código correspondiente a la proyección o selección que precede al operador groupby. Luego se le aplica la operación de distinct() y a continuación se toma el resultado obtenido y se realiza una selección, como vimos al principio de esta sección, tomando como condición que el elemento analizado sea igual al elemento de agrupamiento.

La generación del orderby es similar a la realizada para la operación distinct() ya que reutiliza lo realizado para la proyección o selección y agrega al final una sentencia `demo.sort()` -ver ejemplo 10 de la sección [4.10]- que es la que ordena finalmente los elementos.

4.11 Resumen

En este capítulo definimos la sintáxis de las operaciones definidas en CXXML. Ejemplificamos cada una de ellas y mostramos el código generado a partir de ellas con el objetivo de establecer una clara relación entre nuestro lenguaje y el álgebra estándar definido para XML.
En el próximo capítulo explicaremos su implementación.

Capítulo 5

Implementación

En este capítulo se plantea cómo fue el desarrollo del prototipo de CXXML. Se analizarán distintas herramientas existentes y se justificará la opción elegida, para el lenguaje de implementación y el soporte de acceso y representación de datos XML.
Se detallará la arquitectura final de la aplicación y en la última sección se verán las especificaciones de los algoritmos planteados en el álgebra.

5.1 Implementación de CX-XML

Nuestro lenguaje de consultas fue realizado en Java, basados en el concepto de que Java es código portable y XML representa datos portables [M01]. Java asegura que cualquier sistema operativo y hardware con una máquina virtual Java (JVM) puede ejecutar su bytecode compilado. Otra característica importante para la elección de este lenguaje, es que proporciona el conjunto más robusto de API, analizadores, procesadores y herramientas para utilizar XML, que el resto de los lenguajes de programación existentes.
Dentro de las APIs existentes para XML (SAX, DOM, JAXP, JDOM, etc.), nuestra elección fue JDOM para manipular y representar los datos, combinado con SAX para gestionar la entrada y salida de datos.
JDOM define una visión muy elemental de un documento XML, con retroalimentación y soporte a otras APIs, como ser SAX y DOM. Incluye también un comportamiento estándar de entrada y salida para crear un objeto Document JDOM, a partir de los datos existentes en XML y escribe los datos del Document en cualquier destino.

Las versiones de API utilizados en la implementación de CX-XML son:

- JDOM beta6
- SAX 2.0, de Apache.Xerces, para construir documentos JDOM

La eficiencia de SAX y JDOM, está analizada en [BENCHXML02], donde se las comparó con otras tecnologías de acceso y representación existentes en la actualidad. Las herramientas analizadas fueron:

```
Crimson v1.1
Xerces v1.2.0 (SAX)
JDOM b6
dom4j v0.2
Electric XML v1.4
```

Los puntos analizados en dicho ensayo son:

1) Tiempo de Construcción del Documento
2) Uso de Memoria del Documento
3) Tiempo de Recorrido del Arbol
4) Tiempo de generación de Texto
5) Tiempo y espacio de Serialización

En el test se demostró que las herramientas usadas en CX-XML -*Jdom b6 y SAX*- fueron las mejores en los ítems 1, 3 y 5. En los puntos 2 y 4, JDOM y SAX arrojaron un resultado promedio respecto al resto de las herramientas testeadas.
Por último, la interfaz gráfica fue realizada en MS-Visual Basic, dado que no se obtuvo un buen rendimiento con las herramientas gráficas de Java.

5.2 Análisis de Alternativas

Otras alternativas de implementación, diferentes a las descritas en la sección anterior, fueron analizadas previamente al desarrollo e implementación del sistema. Estas son: (a) implementación con Visual Basic y DOM; (b) implementación con C++.

(a) Implementación con Visual Basic y DOM

Se analizó la posibilidad de utilizar Microsoft Visual Basic y sus librerías asociadas para representar y manipular XML, basadas en DOM.
Analizando ventajas, se distingue la posibilidad de un desarrollo del sistema bastante rápido, por la naturaleza del lenguaje de contar con instrucciones típicas de un lenguaje de alto nivel. Por otro lado, esa facilidad de construir el sistema en forma rápida, perturba el objetivo de generar un código intermedio abierto y optimizable, por lo que fue desechada esta opción.

(b) Implementación con C++

Esta opción, se aproxima mucho a las intenciones del proyecto, pero el manejo del documento XML requiere un esfuerzo de desarrollo mayor para la interacción con XML, dado que habría que armar la estructura necesaria para representarlo y manipularlo. Otra desventaja importante que encontramos es el hecho de que C++ no genera código portable, como JAVA, por lo que habría que manejar distintas versiones del sistema, según la arquitectura en la que se implemente.

5.3 Arquitectura

A continuación observaremos un esquema de la arquitectura usada para CXXML:

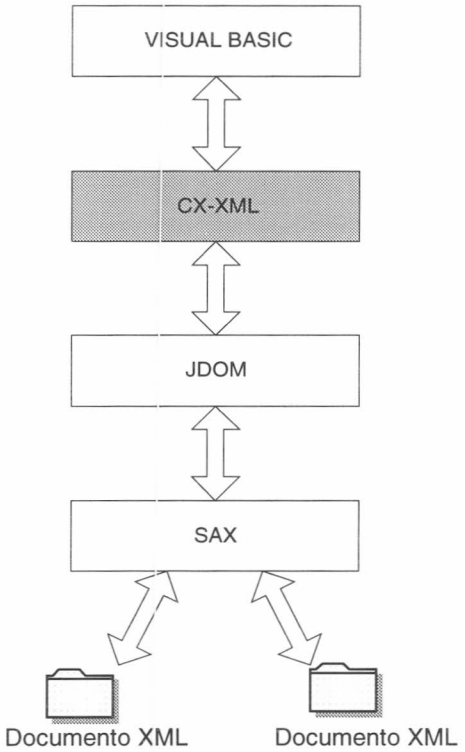
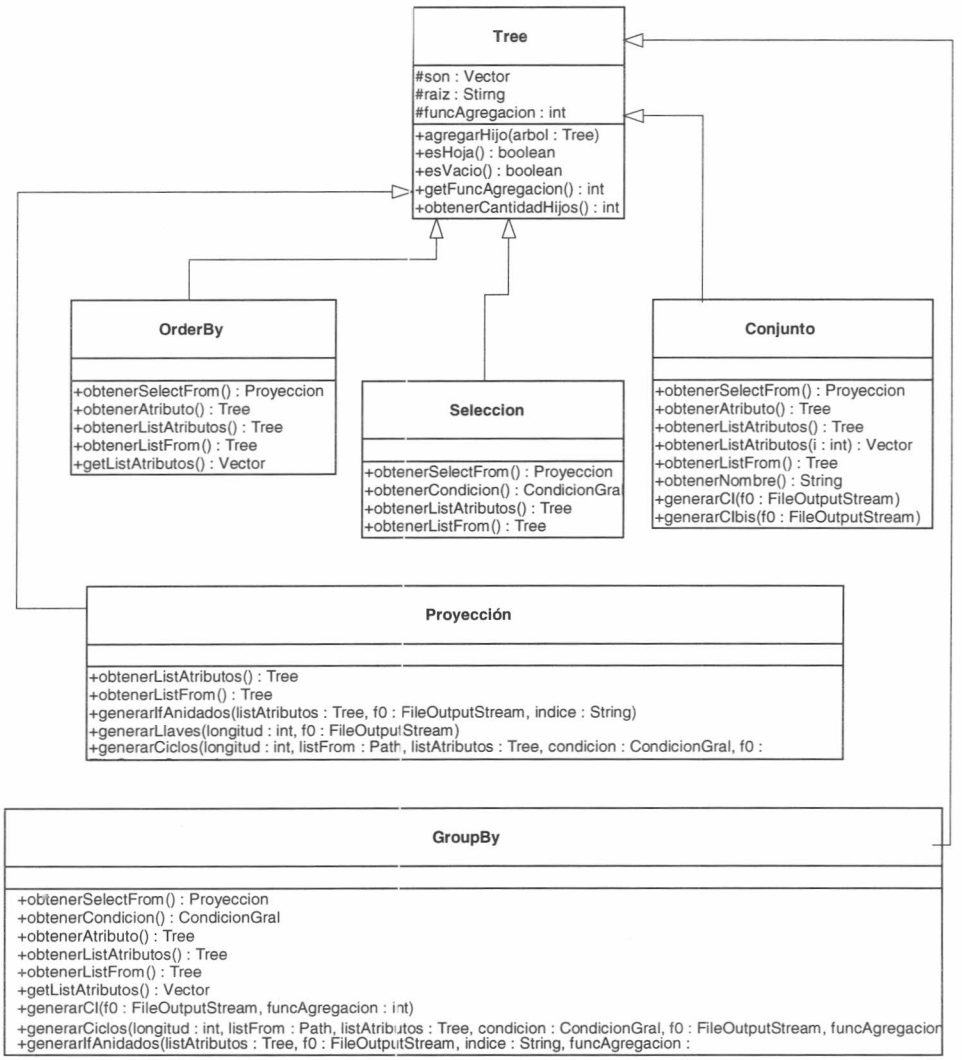


Figura 1 - Arquitectura CX-XML

En la figura 1 observamos las capas en las que se divide nuestra arquitectura: la interfaz del usuario realizada en Visual Basic 6.0; nuestro lenguaje de consultas CX-XML realizado en Java 3.0; y por último JDOM y SAX.
La forma en la que interactúan estos componentes es la siguiente: Cada vez que un usuario ingresa una consulta y el archivo XML a consultar, CXXML recibe mediante una llamada estos datos y realiza el análisis sintáctico y semántico de la sentencia. Para realizar el análisis sintáctico se utiliza un Parser propio. Mientras que para acceder a los documentos XML se utiliza SAX y JDOM. Si la validación es exitosa genera el código fuente Java necesario para obtener el resultado. Este código -definido en el álgebra estándar de XML- tiene embebido métodos definidos en las herramientas mencionadas con anterioridad para acceder a los datos.
Una vez finalizado este paso, la interfaz permite compilar y ejecutar el código fuente obteniendo como resultado un nuevo documento XML con el resultado buscado. Nuevamente se utilizan en este paso métodos definidos en JDOM y SAX para crear el documento final.

5.4 Diagrama de Clases

A continuación mostramos las principales clases creadas utilizadas en el proyecto y su relación entre sí.



Descripción de la clase Tree

La clase Tree es la base de nuestra implementación. Con ella se genera el árbol sintáctico que se utiliza para analizar la correctitud de la sentencia. Todas las operaciones soportadas por CXXML heredan de esta clase su comportamiento.

Los principales métodos de la clase son:

- `agregarHijo`: agrega un elemento al nodo del árbol.
- `esHoja`: indica si un nodo es hoja del árbol.
- `esVacío`: indica si el árbol tiene elementos.

- `getFuncAgregacion`: devuelve la función de agregación embebida en la sentencia.
- `ObtenerCantidadhijos`: devuelve la cantidad de hijos que posee un árbol.

Descripción de la clase Proyección

La clase Proyección es la clase principal usada en nuestro lenguaje. Como todas las sentencias implementadas tienen al menos una proyección, ella se utiliza para generar el árbol que representa dicha proyección.

Los principales métodos de la clase son:

- `obtenerListAtributos`: devuelve la lista de atributos o elementos que se encuentran luego de la cláusula `SELECT`.
- `obtenerListFrom`: devuelve la lista de path expressions que se encuentran luego de la cláusula `FROM`.
- `generarIfAnidados`: genera dentro del código las sentencias *if* anidadas que sirven para recuperar los elementos o atributos solicitados en la cláusula `SELECT`.
- `generarLlaves`: genera dentro del código las llaves que cierran los ciclos.
- `generarCiclos`: genera dentro del código los ciclos que permiten recorrer el documento.

Descripción de la clase Selección

La clase Selección se utiliza para implementar las operaciones de ese tipo. A los efectos prácticos la condición fue limitada a un máximo de dos operadores lógicos de la forma: c_1 *op* c_2 *ob* c_3 *op* c_4 siendo op un operador lógico y ob un operador booleano *AND u OR*.

Los principales métodos de la clase son:

- `obtenerSelectFrom`: devuelve la proyección que se encuentra embebida en la sentencia, antes de la cláusula `WHERE`.
- `obtenerCondicion`: devuelve la condición especificada luego de la cláusula `WHERE`.
- `obtenerListAtributos`: devuelve la lista de atributos o elementos que se encuentran luego de la cláusula `SELECT` . Este método invoca al método de mismo nombre de la clase Proyeccion.
- `obtenerListFrom`: devuelve la lista de path expressions que se encuentran luego de la cláusula `FROM`. Este método invoca al método de mismo nombre de la clase Proyeccion.

Descripción de la clase GroupBy

La clase GroupBy se utiliza para implementar las operaciones de agregación.

Los principales métodos de la clase son:

- `obtenerSelectFrom`: devuelve la proyección que se encuentra embebida en la sentencia, antes de la cláusula `WHERE`. Este método invoca al método de mismo nombre de la clase Selección.
- `obtenerCondicion`: devuelve la condición especificada luego de la cláusula `WHERE`. Este método invoca al método de mismo nombre de la clase Selección.
- `obtenerAtributo`: devuelve el atributo asociado a la cláusula `GROUPBY`.
- `obtenerListAtributos`: devuelve la lista de atributos o elementos que se encuentran luego de la cláusula `SELECT`. Este método invoca al método de mismo nombre de la clase Seleccion.
- `obtenerListFrom`: devuelve la lista de path expressions que se encuentran luego de la cláusula `FROM`. Este método invoca al método de mismo nombre de la clase Seleccion.
- `generarCI`: genera las líneas de código específicas para esta operación.
- `generarCiclos`: genera dentro del código los ciclos que permiten recorrer el documento.
- `generarIfAnidados`: genera dentro del código las sentencias *if* anidadas que sirven para recuperar los elementos o atributos solicitados en la cláusula `SELECT`.

Descripción de la clase OrderBy

La clase OrderBy se utiliza para implementar las operaciones de ordenamiento.

Los principales métodos de la clase son:

- `obtenerSelectFrom`: devuelve la proyección que se encuentra embebida en la sentencia, antes de la cláusula `WHERE`. Este método invoca al método de mismo nombre de la clase Selección.
- `obtenerAtributo`: devuelve el atributo asociado a la cláusula `GROUPBY`.
- `obtenerListAtributos`: devuelve la lista de atributos o elementos que se encuentran luego de la cláusula `SELECT`. Este método invoca al método de mismo nombre de la clase Seleccion.
- `obtenerListFrom`: devuelve la lista de path expressions que se encuentran luego de la cláusula `FROM`. Este método invoca al método de mismo nombre de la clase Seleccion.

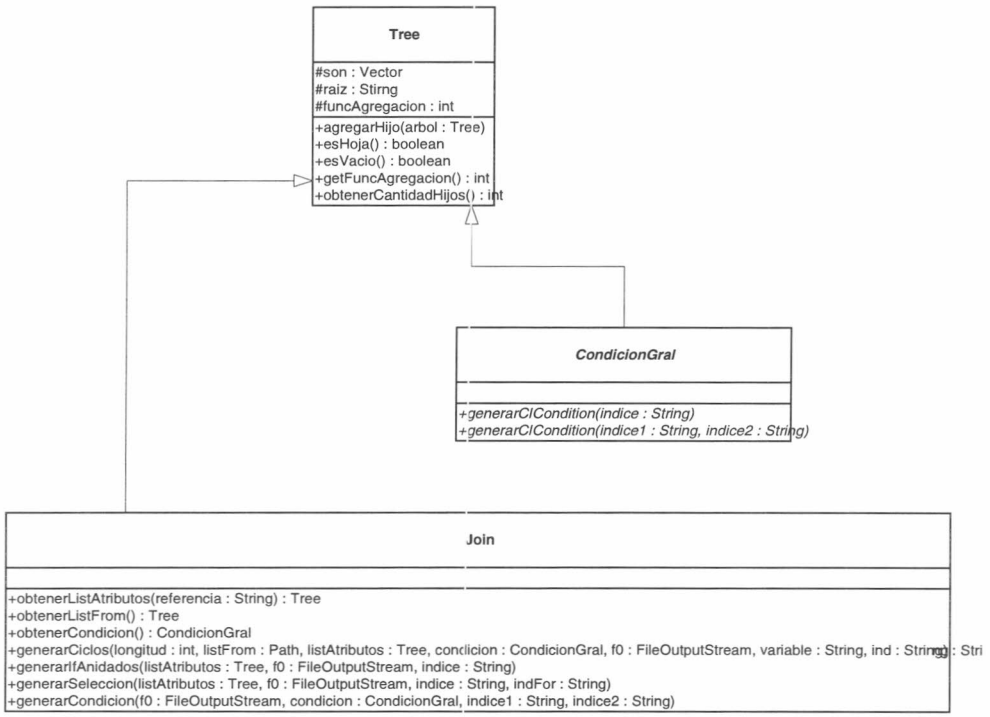
Descripción de la clase Conjunto

La clase Conjunto se utiliza para implementar las operaciones union e intersection.

Los principales métodos de la clase son:

- `obtenerSelectFrom`: devuelve la proyección que se encuentra embebida en la sentencia, antes de la cláusula `WHERE`. Este método invoca al método de mismo nombre de la clase Selección.
- `obtenerAtributo`: devuelve el atributo asociado a la cláusula `GROUPBY`.

- `obtenerListAtributos`: devuelve la lista de atributos o elementos que se encuentran luego de la cláusula `SELECT`. Este método invoca al método de mismo nombre de la clase Seleccion.
- `obtenerListFrom`: devuelve la lista de path expressions que se encuentran luego de la cláusula `FROM`. Este método invoca al método de mismo nombre de la clase Seleccion.
- `generarCI`: genera las líneas de código específicas para esta operación.
- `generarCIbis`: genera las líneas de código específicas para esta operación.



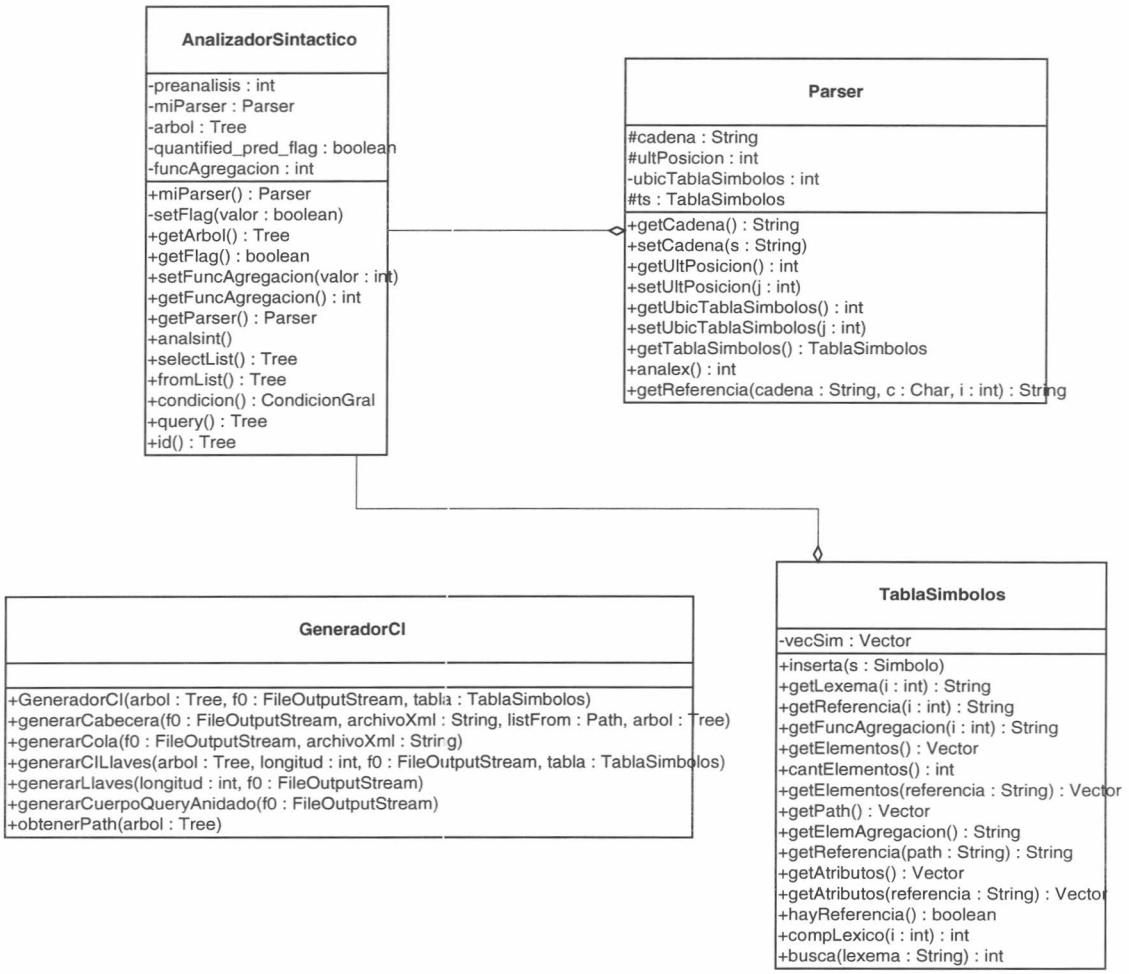
Descripción de la clase Join

La clase Join se utiliza para implementar las operaciones de junta entre documentos.

Los principales métodos de la clase son:

- `obtenerListAtributos`: devuelve la lista de atributos o elementos que se encuentran luego de la cláusula `SELECT`. Este método invoca al método de mismo nombre de la clase Seleccion.
- `obtenerListFrom`: devuelve la lista de path expressions que se encuentran luego de la cláusula `FROM`. Este método invoca al método de mismo nombre de la clase Seleccion.
- `obtenerCondicion`: devuelve la condición especificada luego de la cláusula `WHERE`. Este método invoca al método de mismo nombre de la clase Selección.
- `GenerarCiclos`: genera dentro del código los ciclos que permiten recorrer el documento.
- `generarIfAnidados`: genera dentro del código las sentencias *if* anidadas que sirven para recuperar los elementos o atributos solicitados en la cláusula `SELECT`.

- `generarSeleccion`: genera el código referido a la operación de selección embebida dentro de ésta.
- `generarCondicion`: genera dentro del código la sentencia *if* asociada a la condición `WHERE`.



Descripción de la clase AnalizadorSintactico

La clase AnalizadorSintactico se utiliza para realizar el análisis sintáctico de la sentencia CXXML ingresada. Está íntimamente relacionada con las clases Parser y TablaSimbolos que discutiremos más adelante.

El principal método de la clase es:

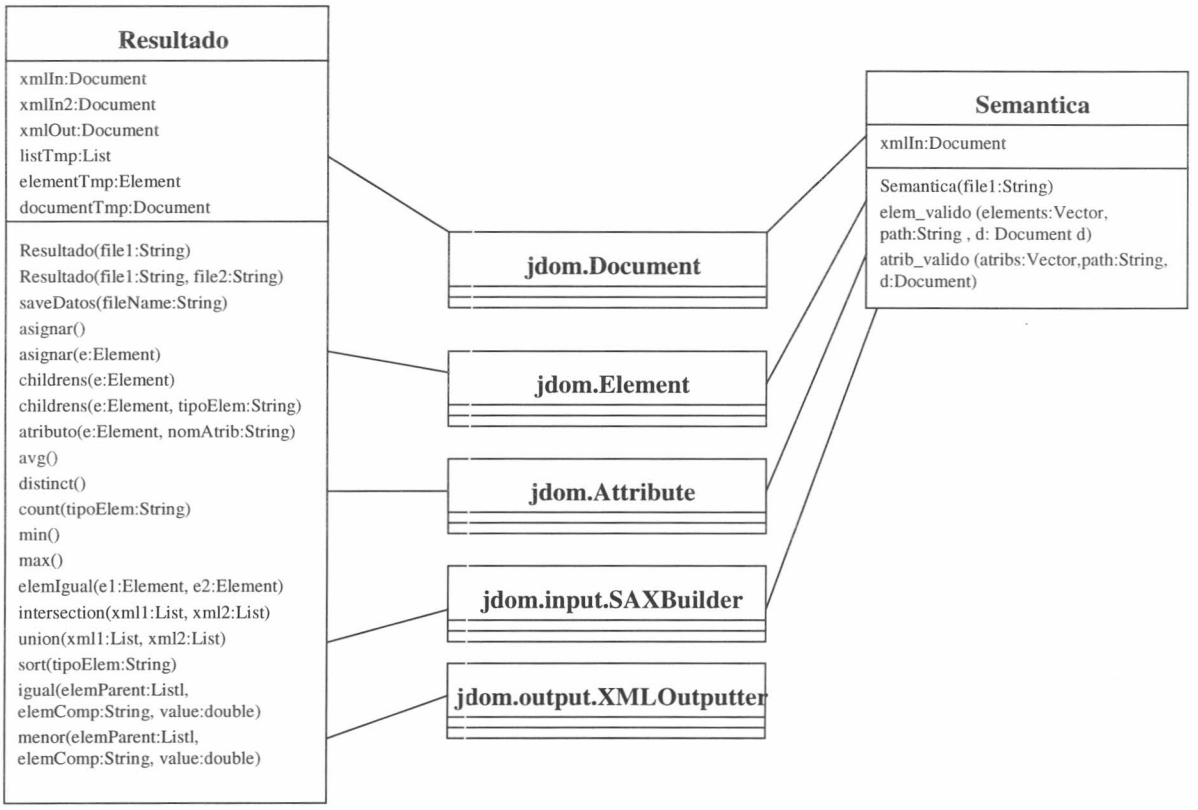
- `analsint`: realiza el análisis sintáctico de la sentencia devolviendo el resultado a la clase *main*. Este método invoca a su vez a otros métodos necesarios para el análisis de la sentencia como ser: *selectList(), fromList(), condicion(), query()* y *id()*.

Los principales métodos de la clase son:

- `children` : devuelve todos o parte de los hijos de un elemento dado
- `asignar`: agrega un elemento a un documento
- `distinct`: implementación del algoritmo Distinct para eliminación de elementos duplicados.
- `count, max, min, avg, sum`: Implementación de las funciones de agregación.
- `igual`: implementación del algoritmo "==", para la comparación de dos elementos en forma recursiva
- `mayor, menor, igual, mayor_igual, menor_igual`: métodos de comparación de valores de elementos, usados generalmente en la condición del `where`.
- `sort` : implementación del algoritmo de ordenamiento para dar soporte al order_by
- `union, intersection`: Implementación de las operaciones de conjuntos.
- El resto de los métodos son funciones necesarias para la implementación de los métodos anteriores, para asignaciones temporarias, para acceder y grabar datos, etc.

5.5 Interfaz CX-XML

La interfaz construida para realizar consultas CXXML, permite al usuario ingresar una sentencia de consulta y el archivo XML a consultar; CXXML realiza el análisis sintáctico y semántico de aquella y genera el código fuente necesario para obtener el resultado. Este código -definido en el álgebra estándar de XML- tiene embebido métodos para acceder a los datos definidos en las herramientas mencionadas con anterioridad.
El reconocedor del lenguaje propiamente dicho y el soporte a realizar consultas CXXML está realizado en Java. Para ello se definió un parser para reconocer las sentencias del lenguaje y un generador de código intermedio Java, arrojando como resultado el código correspondiente a la consulta ingresada como se define en secciones anteriores. Dicho código, luego se compila y ejecuta, obteniendo el resultado de la consulta en formato XML.
La aplicación que hace de interfaz de usuario fue realizada en Visual Basic 6.0. La misma permite manipular el lenguaje CXXML, cargar documentos XML, visualizar código intermedio Java, visualizar errores sintácticos y/o semánticos y visualizar el documento resultado de la ejecución de la consulta propiamente dicha.
A continuación se observa la pantalla principal de la aplicación:



Descripción de la clase Semantica

La clase Semántica tiene el rol de dar soporte a la interfaz para el chequeo semántico de la sentencia escrita en CX-XML. El objeto instanciado, tiene una sola variable de instancia de llamada xmlIN de tipo Jdom.Document

Los principales métodos de la clase tienen las siguientes características:

- Validar que el path del `from` sea válido. Para ello alcanza con encontrar el primer path que encuentre. Es una función con recursividad total sobre todo el árbol
- Validar si un elemento pertenece a un path
- Validar si un atributo pertenece a un elemento
- Validación para elementos de `GroupBy`

Descripción de la clase Resultado

La clase Resultado es una de las principales clases del sistema. Da el soporte a todas las funciones que son usadas en el código intermedio de la consulta, totalmente relacionado con lo definido en el álgebra.
Al instanciar el objeto Resultado, las variables de instancia que se crean son 3 objetos JDOM.Document, los 2 primeros (xmlIN1 y xmlIN2) son para cargar los documentos a consultar y el tercero (xmlOUT) es donde quedará almacenado el documento resultado.

Descripción de la clase Parser

La clase Parser se utiliza para realizar el parser de la sentencia CXXML ingresada. Su función es la de devolver a la instancia de la clase AnalizadorSintactico un token nuevo cuando ésta lo solicita.

El principal método de la clase, además de los proyectores, es:

`analex`: analiza la cadena de caracteres ingresada y busca el siguiente token válido para devolverlo al analizador sintáctico.

Descripción de la clase TablaSimbolos

La clase TablaSimbolos se utiliza para guardar todos los datos asociados con cada uno de los símbolos que el Parser detectó. Estos datos son: lexema, referencia -si el símbolo tienen asociada una referencia- y función de agregación -si el símbolo tiene asociada una función de agregación.

Descripción de la clase GeneradorCI

La clase GeneradorCI se utiliza para generar el código de todas las sentencias CXXML ingresadas.

Los principales métodos de la clase son:

- `generarCabecera`: es el método principal de la clase. Genera el archivo en el cual se guardará el código generado y realiza la invocación al resto de los métodos.
- `generarCola`: genera las últimas líneas del código que son comunes a todas las sentencias CXXML.
- `generarCI`: genera el código en función de la sentencia ingresada.
- `generarCILlaves`: genera las llaves que cierran los casos especiales.
- `generarLlaves`: genera las llaves que cierran los ciclos comunes a todas las sentencias CXXML.
- `generarCuerpoQueryAnidado`: genera las líneas de código para el caso puntual de tener un query anidado.

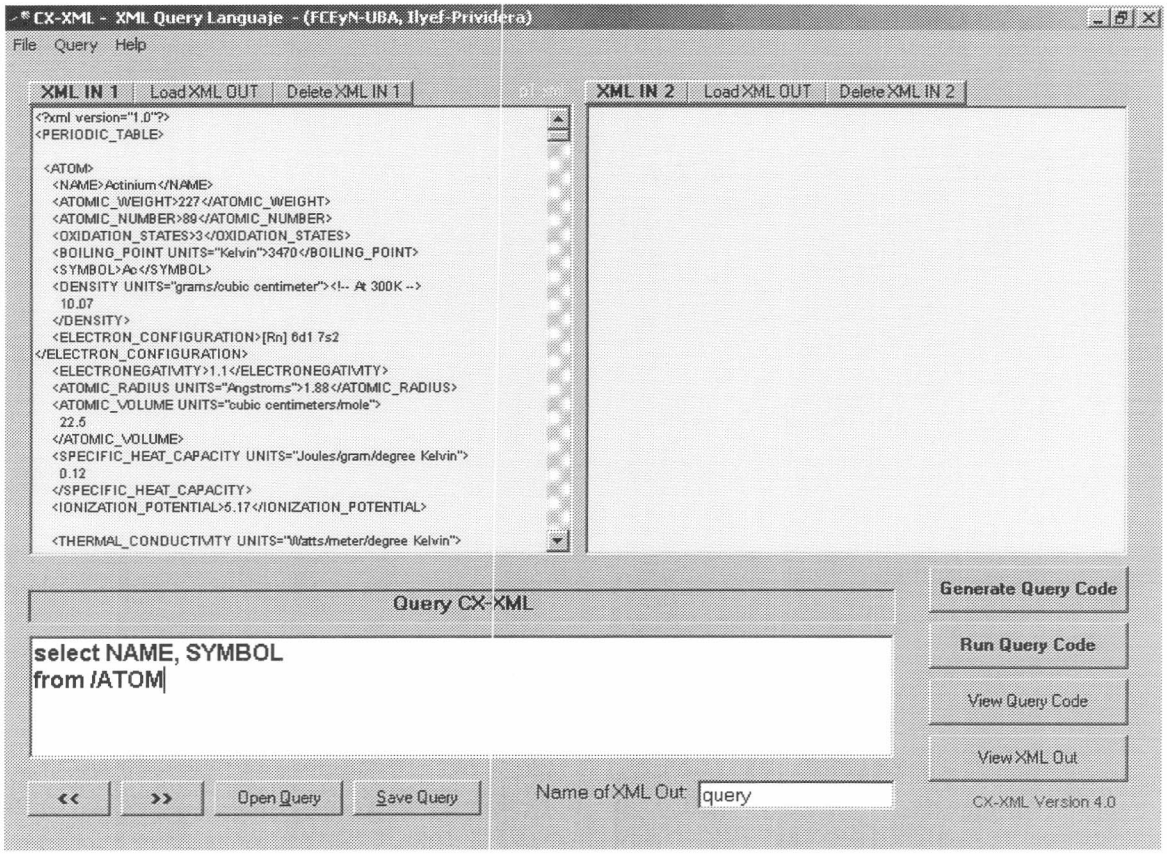


figura 2 – Interface CXXML

La aplicación consta de tres frames detallados a continuación:

XML IN 1: Primer documento a consultar. Este campo es requerido.

XML IN 2: Para los casos en los que sea necesario, aquí se puede cargar el segundo documento a consultar.

Query CX-XML: Editor de consulta, en formato multilínea donde se escribirá la consulta en lenguaje CX-XML

A su vez, existen ocho botones cuyas funciones se detallan a continuación:

XML IN 1: Carga el primer documento XML (XML IN 1)

XML IN 2: Carga el segundo documento XML si fuese necesario (XML IN 2)

LOAD XML OUT: Carga en el Frame correspondiente, el XML OUT resultante de la última consulta ejecutada.

Generate Query Code: Generador de Código Intermedio Java con el código correspondiente a la consulta escrita en CX-XML.

Run Query Code: Compilación y Ejecución del Código Intermedio Java.

View Query Code: Visor del código Intermedio Java generado o Visor de Errores sintácticos o semánticos. Por defecto levanta el archivo con el editor de Windows.

View XML Out: Visor del XML OUT resultante de la última consulta ejecutada. Por defecto levanta el archivo XML, en el editor de Windows.

Name of XML Out: Permite especificar el nombre con el que se grabará el resultado de la consulta.

<< : Itera en retroceso sobre consultas CX-XML escritas anteriormente.

>> : Itera en avance sobre consultas CX-XML escritas posteriormente.

Open Query: Permite levantar consultas CX-XML de disco, en formato de texto con extensión .cxq

Save Query: Permite grabar en disco consultas CX-XML en formato de texto, con extensión .cxq

Generación de Código Intermedio

El paso siguiente a escribir la consulta es realizar la Generación de Código Intermedio Java (Generate Query Code) con el código correspondiente a la consulta escrita en CX-XML.
Esta acción tiene como resultado el código Java intermedio o bien la notificación de errores sintácticos o semánticos (detallados más adelante en la presente sección)
Este proceso se realizará desde el shell de DOS, mediante la ejecución de una secuencia de comandos en forma automática, llamado *main.bat*, parametrizada con los nombres de documentos a consultar y la consulta CXXML propiamente dicha. A continuación mostramos el contenido de *main.bat*:

```
(1)  cd c:/testcxxml
(2)  java -cp cxxml.jar;xerces.jar;jdom.jar; cxxml.main -
  q%1 -a%2 -a%3
(3)  write.exe c:/testcxxml/query.java
(4)  exit
```

En la línea (1) observamos que la aplicación se posiciona en el directorio de trabajo. En (2) se ejecuta en JAVA la aplicación *cxxml.main* (clase principal), incluyendo las librerías *cxxxml.jar* (todas las clases de la tesis), *xerces.jar* (clases de SAX), *jdom.jar* (clases de JDOM) y recibiendo como parámetros el texto de la consulta (-q%1), y los documentos a consultar (-a%1 y –a%2).
En (3) se invoca al editor de Windows para mostrar la clase *query.java* resultante del punto (2), que muestra el código Java de la consulta, o también los errores sintácticos o semánticos si hubiesen existido.

Ejecución de Código

Luego de que se genere correctamente el código intermedio Java, será necesario ejecutar ese código para obtener los datos resultantes de la consulta (Run Query Code)
Este proceso se realizará desde el shell de DOS, mediante la ejecución de una secuencia de comandos en forma automática, llamado *query.bat*, que toma como entrada la clase *Query.java*, obtenida en el paso anterior. A continuación mostramos el contenido de *query.bat*:

```
(1)  cd c:/testcxxml
(2)  javac    -classpath    cxxml.jar;xerces.jar;jdom.jar;
  Query.java
(3)  java -cp cxxml.jar;jdom.jar;xerces.jar; Query
(4)  write.exe query.xml
```

En la línea (1) observamos que la aplicación se posiciona en el directorio de trabajo de la aplicación.
En (2) se compila el código Java definido en *Query.java*. Para ello se ejecuta el compilador JAVA, *javac.exe* y se incluyen las librerías *cxxxml.jar* (todas las clases de la tesis), *xerces.jar* (clases de SAX), *jdom.jar* (clases de JDOM), generándose la clase compilada *Query.class*
En la línea (3) se ejecuta en JAVA, la clase *Query.class* generada anteriormente, devolviendo el resultado de la consulta CXXML en el documento resultado *query.xml* (xmlOUT).
En (4) se invoca al Editor de Windows para ver el archivo *query.xml* (xmlOut)

5.6 Algoritmos

En esta sección se presentan los algoritmos para especificar funciones u operadores, usados en el capítulo 4, escritos en pseudo-código para independizarlo del lenguaje en que se deseen implementar. En la especificación del álgebra se referencian estas funciones, definiendo su significado, pero dejando la libertad de implementarlo según las características del lenguaje de desarrollo. *Estos algoritmos pueden ser mejorados o modificados en la implementación elegida, sin que cambie la definición del lenguaje CXXML.*

Algoritmo '=='

El XML Query Data Model [FR] define el operador de igualdad de nodos '==', donde el valor será verdadero sí y solo sí los dos nodos son exactamente el mismo nodo, siendo dicho operador dependiente de la implementación elegida.
Definimos un método de comparación de elementos que verifica la información que consideramos más relevante, haciendo una recursión general sobre los elementos a comparar, pero sin llegar a ser total. Los componentes que tenemos en cuenta son los contenidos de elementos, de atributos, y su organización estructural. Es importante destacar que la comparación es por los valores y su organización, por lo que no se tiene en cuenta para la comparación los nombres de elementos o atributos, dándole mayor

flexibilidad a las operaciones del lenguaje. De esta manera, se plantea un algoritmo de igualdad eficiente y útil a la vez. Este operador es importante para el resto de los algoritmos desarrollados en las próximas secciones, que lo usan para sus operaciones de igualdad entre elementos.
A continuación se presenta la especificación del operador "==":

```
funcion ==(element1, element2) : lógico
 lógico compAtrib, compElem
 Lista children1 = children(element1)
 Lista children2 = children(element2)
 Lista attributs1 = attributs(element1)
 Lista attributs2 = attributs(element2)
 int cantHijos1 = tamaño(children1)
 int cantHijos2 = tamaño(children2)
 si ((cantHijos1=cantHijos2) y
    (tamaño(attributs1) = tamaño(attributs2))) entonces
  compAtrib = compAtrib(attributs1,attributs2)
  compElem = (data(element1) = data(element2))
  si (compAtrib y compElem) entonces
   /* llega a este nivel si el contenido de elemento y
   /* atributos es identico
   si (cantHijos1>0) entonces
        /* llega a este nivel si tiene hijos el elemento actual
        i=1
      mientras ((i<=cantHijos1) y
          (==(children1(i), children2(i)))) hacer
        /* si los elementos hijos(i) cumplen con el ==,
        /* avanza al proximo elemento hijo del mismo nivel
            i=i+1
        fin
        si i>tamaño(children1) entonces
            /* si se recorrieron todos los elementos hijos(i)
            /* fueron iguales y retorna verdadero
            retornar Verdadero
        sino
            /* sino se recorrieron todos los hijos es falso
            retornar Falso
        fin
   sino
        /* devolvera verdadero dado que no tiene mas hijos y su
        /* contenido de elemento y sus atributos son iguales
        retornar Verdadero
   fin
   sino
   /* no cumple con la comparacion del valor del elemento y
   /* el contenido de sus atributos
    retornar Falso
   fin
  sino
   retornar Falso
  fin
fin
```

La función "==" recibe como parámetro dos elementos, denominémoslos `element1` y `element2` y devuelve un valor lógico verdadero si ambos elementos son iguales y falso en caso contrario. Para ello obtiene primero una lista de los elementos hijos de cada

elemento usando la función `children` definida en la sección 3.11 y luego una lista de los atributos de los elementos a comparar.
Para entrar en el ciclo principal de la función se chequea que sea igual la cantidad de atributos y de elementos hijos para `element1` y `element2`, utilizando la función `tamaño` que devuelve la cantidad de objetos que contiene el parámetro de entrada. Si se cumple con la igualdad numérica, se compara la igualdad de contenido; para el caso de los atributos, la comparación de contenido se hace usando la función `compAtrib` definida más adelante en esta sección; para comparar el contenido de los elementos se usa la primitiva `data()` definida en el álgebra. Si los valores de los atributos y el texto del elemento son idénticos, se avanza a la segunda instancia de decisión, donde se compara en forma recursiva cada elemento hijo de element1, con su correspondiente en element2, devolviendo como resultado la conjunción de valores de verdad resultantes de cada comparación; como resultado final alcanza con que un solo valor sea falso para que toda la comparación resulte falsa; si todos los pasos fueron arrojando resultados verdaderos, el resultado será que ambos elementos son iguales.
A continuación se detalla la función `compAtrib`, que recorre 2 listas de atributos (attributs1 y attributs2) y devuelve el valor de verdad de comparar texto de a pares de atributos:

```
funcion compAtrib(attributs1, attributs2): lógico
 lógico valueReturn = True
 int cantAtrib1 = tamaño(attributs1)
 int i = 1
 mientras ((i<= cantAtrib1) y
      (data(attributs1(i))= data(attributs2(i)))) hacer
      /* cicla mientras los valores de los atributos sean
      /* idénticos y las iteraciones no supere la cantidad
      /* de atributos
       i = i+1
 fin
 si (i>cantAtrib1) entonces
  /* todos los atributos fueron de idéntico valor y la
  /* condición de salida del mientras fue por haberlos
  /* recorrido a todos
  retornar Verdadero
 sino
  retornar Falso
 fin
fin
```

Algoritmo Distinct

La función `distinct` tiene como objetivo, dado un documento, obtener otro documento sin elementos duplicados. La igualdad entre elementos, esta definido por el operador "==" definido anteriormente. Se describe a continuación la semántica de esta operación:

pre-condición: dado un Documento D, con elementos $e_1 ... e_n$

`distinct (D)` → `D'`

post-condición : se obtiene un nuevo documento D', con elementos $e'_1..e'_p$, con p <= n . Para todo los elementos $e'_i \in D'$, con i = 1..p, vale que $e'_i == e_j$ (j=1..n), para algún j, con $e_j \in D$ También cumple que $e'_i \neg== e'_k$ (k=1..p, k<>i), $e'_k \in D'$. El operador ==, es el definido anteriormente en esta misma sección .

A continuación presentamos la especificación del algoritmo:

```
funcion distinct(elementRoot): documento
  Lista b = elementRoot
  Lista v
  lógico encontroElemento = Falso
  v(1) = b(1)
  para cada b(i) con i=2..tamaño(b) hacer
    encontroElemento = Falso
    para cada v(j) con j=1..tamaño(v) hacer
          si ==(b(i), v(j)) entonces
             encontroElemento = Verdadero
                salir para
          fin
    fin
    si ¬ encontroElemento
     añadir b(i) a v
    fin
   fin
   distinct = v
fin
```

Observamos en la especificación anterior, que la función recibe un documento `elementRoot`, manejándolo como la lista `b` y creando una lista nueva `v`, donde se irá almacenando el documento sin elementos repetidos.
Luego de inicializar `v(1)` con el primer elemento de `b`, se itera sobre este último agregándose a `v` el elemento `b(i)`, si y solo si no existiese en `v`.
Una vez que se terminó de recorrer todo `b`, se retorna `v` como resultado de la función.

Algoritmo Union

La función union se implementó tomando como entrada dos documentos XML, y devolviendo como resultado la unión disjunta de cada uno de ellos. Se describe a continuación la semántica de esta operación:

pre-condición: dado un Documento D, con elementos $e_1...e_n$, y un documento D' con elementos $e'_1...e'_m$

`union (D, D')` ➔ `D"`

post-condición : se obtiene un nuevo documento D" con p elementos (p>0) tal que : $((\forall e, e \in D \Rightarrow e \in D") \wedge (\forall e', e' \in D' \Rightarrow e' \in D") \wedge$ (sean $e"_1,..,e"_p$ los elementos de D", $\forall j, \forall k, 1 \leq j \leq p, 1 \leq k \leq p, k \neq j \Rightarrow \neg (e"_j == e"_k)$))

Nota: se utiliza el operador de comparación de elementos '==' definido en esta misma sección, acorde a la definición de la unión del XML Query Data Model [FR01]

A continuación presentamos la especificación del algoritmo:

```
funcion union(xml1, xml2): documento
 documento xmlOUT
 concatenar xml2 a xml1
 añadir distinct(xml1) a xmlOUT
 retornar xmlOUT
fin
```

Primero se concatenan los documentos de entrada `xml1` y `xml2`. Luego se utiliza la función `distinct` para eliminar los elementos duplicados de esa concatenación.

Algoritmo Intersection

A continuación se especifica la función `intersection()`, que dados dos documentos retorna los elementos en común de ambos. Se describe a continuación la semántica de esta operación:

pre-condición: dado un Documento D, con elementos $e_1...e_n$, y un documento D' con elementos $e'_1...e'_m$

`intersection (D, D')` → D"

post-condición : se obtiene un nuevo documento D" con p elementos (p>=0) tal que: $((\forall e, e \in D, e \in D' \Rightarrow e \in D'') \wedge$ (sean $e''_1,..,e''_p$ los elementos de D", $\forall j, \forall k, 1 \leq j \leq p, 1 \leq k \leq p, k \neq j \Rightarrow \neg (e''_j == e''_k)$))

Nota: al igual que en la definición de la unión, se utiliza el operador de comparación de elementos '==' definido en esta misma sección, acorde a la definición de la intersección del XML Query Data Model [FR01]

A continuación presentamos la especificación del algoritmo:

```
funcion intersection(xml1, xml2): documento
      documento xmlOUT
      para cada xml1(i) en xml1 hacer
            para cada xml2(j) en xml2 hacer
             si ==(xml1(i),xml2(j)) entonces
                  añadir xml1(i) en xmlOUT
             fin
            fin
      fin
      retornar distinct(xmlOUT)
fin
```

Observamos en el código anterior, que se itera sobre los documentos de entrada, `xml1` y `xml2`, agregando al documento temporario `xmlOUT` aquellos elementos que están en ambos documentos. Como esta iteración puede contener elementos duplicados, al final se aplica la función `distinct`.

Algoritmo Sort

La función `sort`, establece un ordenamiento entre los elementos de un documento resultante de alguna consulta realizada en nuestro lenguaje. Como todos los resultados de nuestras consultas generan un tipo genérico de elemento, que llamamos `parent`, el algoritmo `sort` asume esta estructura, y realiza la comparación por algún elemento hijo de `parent`. Dicha comparación se realiza al invocar a la función `menor`, que se detalla mas adelante.
A continuación se realiza una especificación de la función `sort` usado en la sección [4.6]:

```
funcion sort(xmlIN, campo_Orden): documento
 documento xmlOUT
 posMinimo = 0
 si tamaño(xmlIN) > 1 entonces
  para cada xmlIN(j) con j=0..(tamaño(xmlIN)-1) hacer
   elemMin = xmlIN(0)
   posMinimo = 0
   para cada xmlIN(i) con i=1..(tamaño(xmlIN-1) hacer
        si menor(xmlIN(i),elemMin, campo_Orden) entonces
             posMinimo = i
             elemMin = xmlIN(i)
        fin
  fin
  añadir elemMin a xmlOUT
  eliminar xmlIN(posMinimo)
  si tamaño(xmlIN) > 1 entonces
        salir para
  sino
        añadir xmlIN(0) a xmlOUT
  fin
 fin
 sino
  añadir xmlIN a xmlOUT
 fin
 retornar xmlOUT
fin
```

Observamos en código anterior 2 grandes iteraciones, utilizando un documento temporario xmlOUT para ir almacenando los elementos, partiendo del mínimo encontrado, para luego eliminarlo del documento xmlIN y así seguir buscando el siguiente mínimo, hasta que xmlIN tenga un solo elemento.
Como se dijo antes la función menor, es la que determina si un elemento es menor a otro dado un campo de comparación. A continuación se especifica dicha función:

```
funcion menor (element1, element2, campoOrden): lógico
```

```
logico returnValor = Falso
elemento e1
elemento e2
e1 = children (element1, campoOrden)
e2 = children (element2, campoOrden)
si (data(e1) < data(e2)) entonces
     retornar Verdadero
sino
     retornar Falso
fin
fin
```

En la especificación anterior observamos que se obtienen los elementos hijos de tipo `campoOrden`, de ambos elementos de comparación y luego se analizan los datos de los mismos, comparando por menor al contenido de ambos.

5.7 Resumen

En este capítulo se presentó y justificó la elección como herramienta de desarrollo e implementación del sistema, a JAVA y JDOM. También se presentó el diseño de las clases componentes del sistema y una breve descripción de las mismas.
Finalizando se presentaron los algoritmos para especificar los operadores más relevantes del álgebra y su implementación para nuestro lenguaje, como ser el operador de igualdad de elementos, de operaciones de conjuntos y funciones de ordenamiento.

Capítulo 6

Experimentación

Se realizaron una serie de experimentos, con el objetivo de comparar performance sobre diversos tipos de documentos. Usamos como métrica de performance el tiempo de procesamiento de las consultas. Adicionalmente se buscó conocer el límite máximo dentro del cual el sistema se mantiene estable y sea razonable el tiempo de procesamiento de las consultas. Luego de mostrar los resultados obtenidos, se discutirán algunos puntos importantes, que sentaran una base para trabajos futuros.

6.1 Plataforma de experimentación

El experimento fue realizado utilizando el prototipo CXXML sobre procesador Pentium III, 600mhz, 128Mb de memoria RAM y sistema operativo Microsft Windows 2000 Professional. Nuestro experimento usa 4 fuentes de datos, 3 de los cuales son los usados en [BENCHXML02] y el último un documento representativo de base de datos de texto. Ellos son:

- periodic_table.xml (tabla periódica de elementos)
- much_ado.xml (obra de Shakespeare)
- xml.xml (especificación de XML)
- nt.xml (texto religioso)

A continuación la tabla 4.1 muestra las características de los archivos que son la fuente del experimento:

	Tabla	**Shakespeare**	**Especif.**	**Base de Datos de Texto**				
Documento	pt	much_ado	xml	nt	nt2	nt4	nt8	nt16
Tamaño (MB)	0.1	0.2	0.15	1	2	4	8	16
Profundidad de Path	Bajo	Bajo	Alto	Bajo	Bajo	Bajo	Bajo	Bajo
Nodos XML	112	978	275	16810	33618	67234	134470	268938
Longitud promedio de contenido	Bajo	Medio	Medio	Alto	Alto	Alto	Alto	Alto

Table 6.1: Parametros del Experimento

En todos los casos los resultados analizados se hacen sobre el tiempo de compilación del código intermedio y la ejecución del mismo. El tiempo de generación del código intermedio se mantiene constante, durando aproximadamente 2 segundos su generación para todos los casos, por lo que no se muestra en los resultados obtenidos.

6.2 Ensayo sobre tabla periódica

Se realizó el experimento sobre el archivo `periodic_table.xml`, que representa la tabla de elementos periódicos en XML. Los elementos contienen variedad de atributos y su estructura es representativa de muchos documentos XML de negocios. El tamaño del archivo es de 114Kb y un total de 112 elementos de primer nivel únicamente.
En este primer ensayo se realizaron una serie de consulta que abarcan la mayoría de las consultas que se pueden realizar en el sistema con el objetivo de conocer los tiempos promedio de respuesta de las mismas y empezar a plantear las primeras discusiones sobre los resultados obtenidos.

El siguiente documento XML, es una muestra que representa la tabla periódica de elementos:

```
<PERIODIC_TABLE>
  <ATOM>
    <NAME>Actinium</NAME>
    <ATOMIC_WEIGHT>227</ATOMIC_WEIGHT>
    <ATOMIC_NUMBER>89</ATOMIC_NUMBER>
    <OXIDATION_STATES>3</OXIDATION_STATES>
    <BOILING_POINT UNITS="Kelvin">3470</BOILING_POINT>
    <SYMBOL>Ac</SYMBOL>
    <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
      10.07
    </DENSITY>
    <ELECTRON_CONFIGURATION>[Rn] 6d1 7s2
      </ELECTRON_CONFIGURATION>
    <ELECTRONEGATIVITY>1.1</ELECTRONEGATIVITY>
    <ATOMIC_RADIUS UNITS="Angstroms">1.88</ATOMIC_RADIUS>
    <ATOMIC_VOLUME UNITS="cubic centimeters/mole">
      22.5
    </ATOMIC_VOLUME>
    <SPECIFIC_HEAT_CAPACITY UNITS="Joules/gram/degree Kelvin">
      0.12
    </SPECIFIC_HEAT_CAPACITY>
    <IONIZATION_POTENTIAL>5.17</IONIZATION_POTENTIAL>

    <THERMAL_CONDUCTIVITY UNITS="Watts/meter/degree Kelvin">
    <!-- At 300K -->
      12
    </THERMAL_CONDUCTIVITY>
  </ATOM>
      ...
 </PERIODIC_TABLE>
```

También fueron usados los siguientes documentos auxiliares (ptJoin.xml y pt_conj.xml) para las operaciones de Join, Union e Intersection:

```
<ptJoin>
 <parent>
      <NAME>Actinium</NAME>
      <ORIGEN>ARGENTINA</ORIGEN>
 </parent>
 <parent>
      <NAME>Aluminum</NAME>
      <ORIGEN>ARGENTINA</ORIGEN>
```

```
  </parent>
  <parent>
       <NAME>Americium</NAME>
       <ORIGEN>URUGUAY</ORIGEN>
  </parent>
...
  <parent>
       <NAME>Zirconium</NAME>
       <ORIGEN>SUDAN</ORIGEN>
       </parent>
</ptJoin>


<pt_conj>
  <ATOM>
    <NAME>Aluminum</NAME>
        ...
<pt_conj>
```

El documento pt_conj.xml, contiene 4 elementos de tipo ATOM, siendo los 3 primeros "Aluminum" y el cuarto "Antimony". El primero y el cuarto elemento son idénticos a sus correspondientes en la tabla periódica, mientras que el segundo tiene el valor de un elemento cambiado y el tercero difiere en el valor de un atributo, a fines de testear eficiencia y correctitud sobre la implementación de operaciones de conjuntos, las cuales tienen su fundamento en el operador "==" definido.

Las consultas realizadas fueron las siguientes.

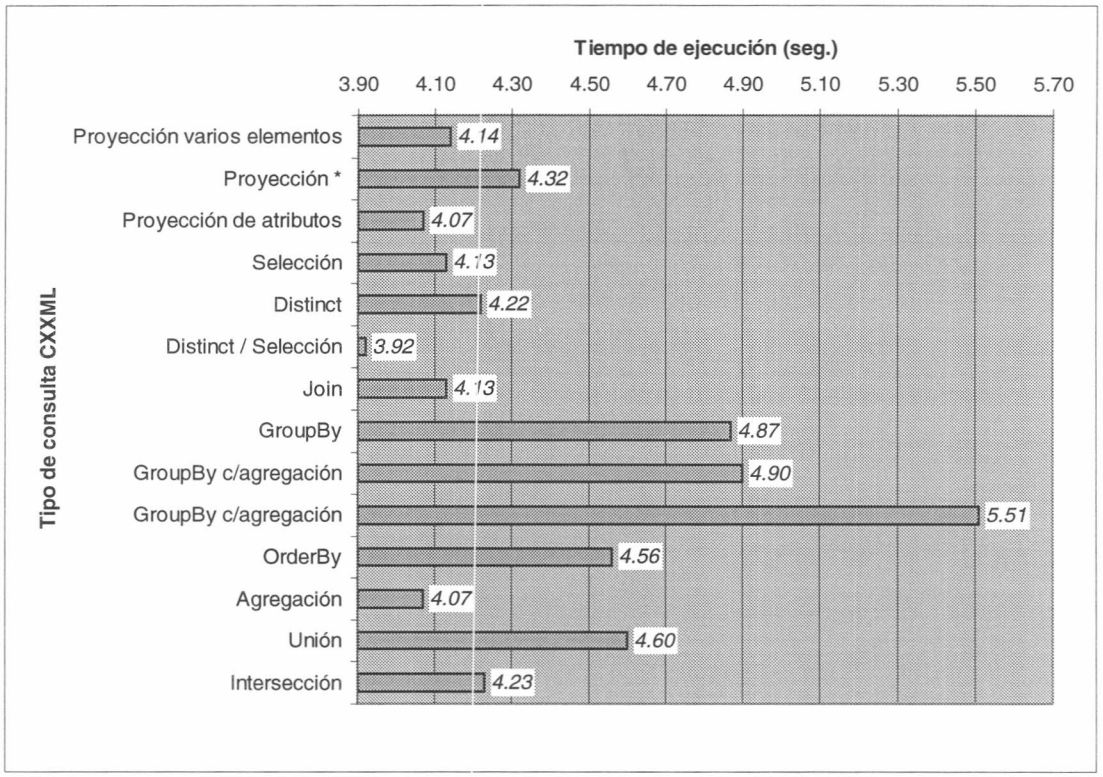
Nro.	*Tipo.*	***Descripción***
1	Proy.	*"Obtener nombre de átomo, número atómico y electronegatividad de los átomos"* **select NAME, ATOMIC_NUMBER,** **ELECTRONEGATIVITY** **from /ATOM**
2	Proy.	*"Obtener todos los átomos"* **select * from /ATOM**
3	Proy.	*"Obtener las unidades en que esta medida la densidad de los átomos"* **select @UNITS from /ATOM/DENSITY**
4	Selec.	*"Seleccionar nombre y radio atómico de aquellos átomos cuya electronegatividad sea mayor o igual a 2"* **select NAME, ATOMIC_RADIUS from /ATOM** **where ELECTRONEGATIVITY >= 2**
5	Dist.	*"Seleccionar las distintas electronegatividades existentes en los átomos"*

		select distinct ELECTRONEGATIVITY from /ATOM
6	Dist.	*"Seleccionar de la consulta anterior, aquellas electronegatividades mayor que 3"* **select distinct ELECTRONEGATIVITY from /parent** **where ELECTRONEGATIVITY > 3**
7	Join	*"Dado 2 documentos, el primero el de la tabla periódica de elementos y el segundo los lugares de origen de los elementos (ptJoin.xml), obtener el nombre y el lugar de cada tipo de átomo"* **select a.NAME, b.ORIGEN** **from /ATOM.a, /parent.b** **where a.NAME = b.NAME**
8	Group	*"Obtener los nombres de los átomos que tienen el mismo estado de oxidación, agrupados por dicho campo"* **select OXIDATION_STATES, NAME from /ATOM** **groupby OXIDATION_STATES**
9	Group	*"Dado un estado de oxidación, obtener la cantidad de átomos que tienen ese estado"* **select OXIDATION_STATES, count(NAME)** **from /ATOM groupby OXIDATION_STATES**
10	Group	*"Dado un valor de volumen atómico, obtener la suma de electronegatividades de todos los átomos para dicho valor"* **select ATOMIC_VOLUME,** **sum(ELECTRONEGATIVITY) from /ATOM** **groupby ATOMIC_VOLUME**
11	Order	*"Seleccionar nombre y número atómico de los átomos ordenado por el número atómico"* **select NAME, ATOMIC_NUMBER from /ATOM** **orderby ATOMIC_NUMBER**
12	Agreg	*"Contar la cantidad de nombres de átomos"* **select count(NAME) from /ATOM**

13	Conj.	*"Tomando como segundo documento a pt_conj.xml, obtener la union disjunta de los átomos"* **select NAME from /ATOM** **union** **select NAME from /parent**
14	Conj.	*"Buscar los atomos en común entre la tabla periódica y pt_conj.xml"* **select NAME from /ATOM** **intersection** **select NAME from /parent**

Resultados

A continuación se muestra un cuadro comparativo de los tiempos que demoró en compilarse el código JAVA de la consulta (1 segundo aproximadamente) y la ejecución de la misma, clasificados por tipo de operación.



Como se observa en el gráfico anterior, los tiempos de respuesta rondan los 4 segundos, en la mayoría de las consultas realizadas. El mayor tiempo observado, entre 5 y 6 segundos, se obtuvo en las funciones de GroupBy, dada sus características de trabajar como un corte de control acorde lo plantea el álgebra.

6.3 Ensayo sobre obra de Shakespeare

El segundo experimento fue realizado sobre el archivo much_ado.xml, que representa la obra de Shakespeare titulada "Much Ado about Nothing". Tiene un tamaño de 198Kb, no tiene atributos y es una estructura completamente plana. Es un representante de una base de datos de texto, por tener campos string muy extensos.

A continuación se muestra una breve porción del documento much_ado.xml:

```
<?xml version="1.0"?>
<PLAY>
  <TITLE>Much Ado about Nothing</TITLE>
  <FM>
    <P>Text placed in the public domain by Moby Lexical
  Tools,1992</P>
    <P>SGML markup by Jon Bosak, 1992-1994.</P>
    <P>XML version by Jon Bosak, 1996-1998.</P>
    <P>This work may be freely copied and distributed
  worldwide.</P>
  </FM>
  <PERSONAE>
    <TITLE>Dramatis Personae</TITLE>
    <PERSONA>DON PEDRO, prince of Arragon.</PERSONA>
    <PERSONA>DON JOHN, his bastard brother.</PERSONA>
    <PERSONA>CLAUDIO, a young lord of Florence.</PERSONA>
    <PERSONA>BENEDICK, a young lord of Padua.</PERSONA>
    <PERSONA>LEONATO, governor of Messina.</PERSONA>
    <PERSONA>ANTONIO, his brother.</PERSONA>
    <PERSONA>BALTHASAR, attendant on Don Pedro.</PERSONA>
    <PGROUP>
      <PERSONA>CONRADE</PERSONA>
      <PERSONA>BORACHIO</PERSONA>
      <GRPDESCR>followers of Don John.</GRPDESCR>
    </PGROUP>
    <PERSONA>FRIAR FRANCIS</PERSONA>
    <PERSONA>DOGBERRY, a constable.</PERSONA>
    <PERSONA>VERGES, a headborough.</PERSONA>
    <PERSONA>A Sexton.</PERSONA>
    <PERSONA>A Boy.</PERSONA>
    <PERSONA>HERO, daughter to Leonato.</PERSONA>
    <PERSONA>BEATRICE, niece to Leonato.</PERSONA>
    <PGROUP>
      <PERSONA>MARGARET</PERSONA>
      <PERSONA>URSULA</PERSONA>
      <GRPDESCR>gentlewomen attending on Hero.</GRPDESCR>
    </PGROUP>
    <PERSONA>Messengers, Watch, Attendants, &amp;amp;c. </PERSONA>
  </PERSONAE>
  <SCNDESCR>SCENE  Messina.</SCNDESCR>
  <PLAYSUBT>MUCH ADO ABOUT NOTHING</PLAYSUBT>
  <ACT><TITLE>ACT I</TITLE>
    <SCENE><TITLE>SCENE I.  Before LEONATO'S house.</TITLE>
    <STAGEDIR>Enter LEONATO, HERO, and BEATRICE, with a
    Messenger</STAGEDIR>
    <SPEECH>
      <SPEAKER>LEONATO</SPEAKER>
      <LINE>I learn in this letter that Don Peter of Arragon</LINE>
      <LINE>comes this night to Messina.</LINE>
    </SPEECH>
```

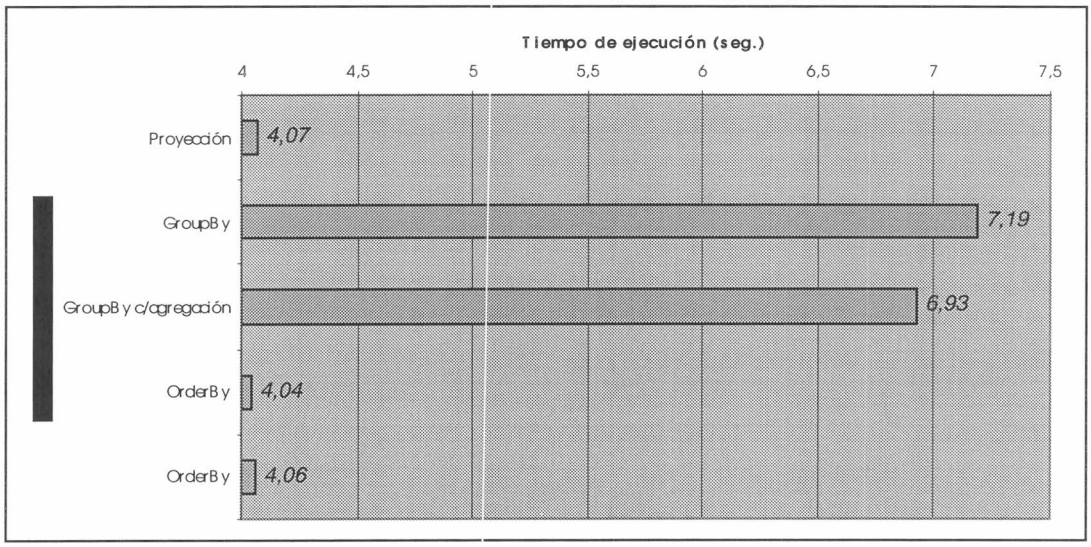
```
      <SPEECH>
        <SPEAKER>Messenger</SPEAKER>
        <LINE>He is very near by this: he was not three leagues
        off</LINE>
        <LINE>when I left him.</LINE>
      </SPEECH>
    ...
      </SCENE>
    </ACT>
  </PLAY>
```

Se realizó un conjunto más acotado de consultas, analizando la respuesta del sistema ante el incremento en los niveles de profundidad del path y testeando el comportamiento al manejo de volúmenes importantes de texto:

Nro.	*Tipo.*	***Descripción***
1	Proy.	*"Obtener el nombre de los personajes de la obra"* **select PERSONA from /PERSONAE**
2	Group	*"Obtener todas las lineas de la obra que tiene que decir cada personaje, agrupado por personaje"* **select SPEAKER, LINE** **from /ACT/SCENE/SPEECH** **groupby SPEAKER**
3	Group	*"Calcular la cantidad de lineas de la obra que tiene que decir cada personaje"* **select SPEAKER, COUNT(LINE)** **from /ACT/SCENE/SPEECH** **groupby SPEAKER**
4	Order	*"Obtener todos los nombres de los personajes ordenados alfabeticamente"* **select PERSONA** **from /PERSONAE** **orderby PERSONA**
5	Order	*"Dada la consulta 2, obtener todos los personajes ordenados alfabeticamente"* **select SPEAKER** **from /parent/parent** **orderby SPEAKER**

Resultados

A continuación se muestra un cuadro de tiempos de respuesta, clasificados por tipo de operación.



Observamos que los tiempos se mantienen similares a los del ensayo 1, las mayores demoras se encuentran en las funciones de agrupación.

6.4 Ensayo sobre especificación de XML

El tercer ensayo realizado fue sobre el archivo `xml.xml` que representa la especificación misma de XML con su referencia de DTD removida, necesario para la versión de CX-XML usada en el test. Tiene un tamaño de 159Kb, contiene un alto nivel de anidamiento, gran cantidad de atributos, gran volumen en texto de elementos o atributos y sus datos están organizados en forma no uniforme.

A continuación se muestra una breve porción del documento xml.xml:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<spec>
  <header>
        <title>Extensible Markup Language (XML) 1.0</title>
        <version />
              <w3c-designation>REC-xml-19980210</w3c-designation>
              <w3c-doctype>W3C Recommendation</w3c-doctype>
      <pubdate>
                    <day>10</day>
                    <month>February</month>
                    <year>1998</year>
              </pubdate>
      <publoc>
    ...
  </header>
  <body>
```

```
      <div1 id="sec-intro">
        <head>Introduction</head>
        <p>
          Extensible Markup Language, abbreviated XML, describes a
          class of data objects called
          <termref def="dt-xml-doc">XML documents</termref>
          and partially describes the behavior of computer programs
          which process them. XML is an application profile or
          restricted form of SGML, the Standard Generalized Markup
          Language
          <bibref ref="ISO8879" />
          By construction, XML documents are conforming SGML
          documents.
        </p>
        <p>
                        ...
        </body>
        ...
          <member>
            <name>John Tigue, DataChannel</name>
          </member>
        </orglist>
      </inform-div1>
    </back>
  </spec>
```

También fue usado un documento auxiliar para las consultas de tipo Join, cuya estructura vemos a continuación:

```
<root>
  <parent>
    <email>jeanpa@microsoft.com</email>
                <country>ARGENTINA</country>
          </parent>
  <parent>
                <email>tbray@textuality.com</email>
                <country>BRASIL</country>
          </parent>
  <parent>
                <email>cmsmcq@uic.edu</email>
                <country>URUGUAY</country>
          </parent>
  <parent>
                <email>error@non.com</email>
                <country>USA</country>
          </parent>
    </root>
```

Las consultas que se realizaron, están orientadas a analizar la respuesta del sistema ante paths de varios niveles y análisis de datos en una estructura no uniforme.

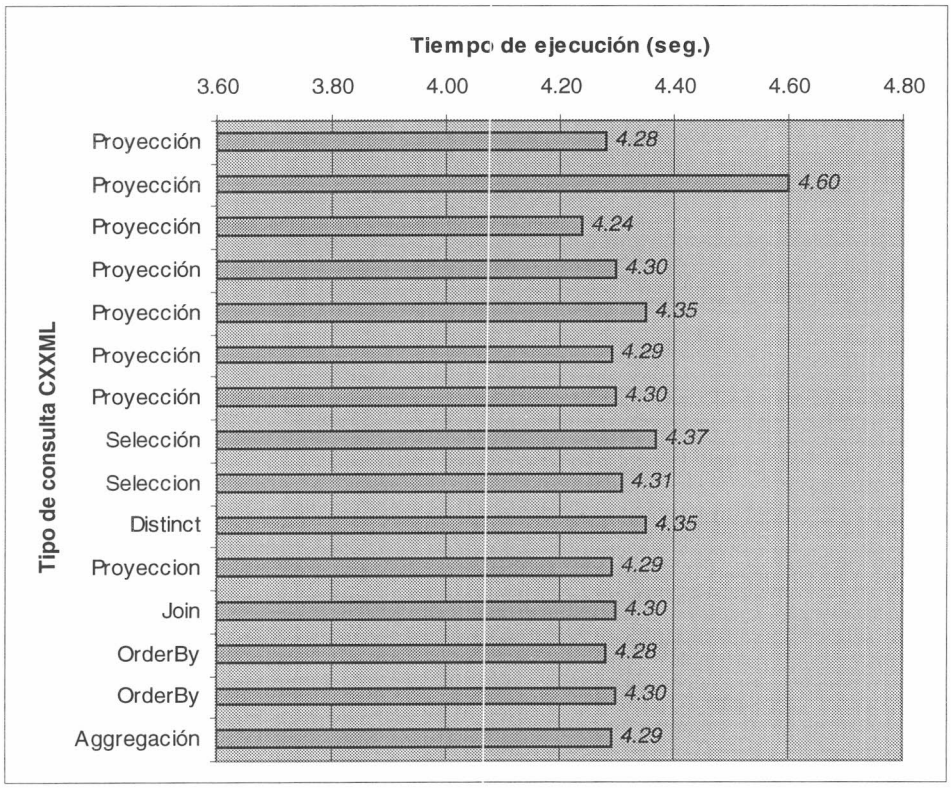
Nro.	*Tipo.*	***Descripción***
1	Proy.	*"Seleccionar nombre y email de los autores del documento"* **select name, email from**

		/header/authlist/author
2	Proy.	*"Seleccionar nombre y email de los autores del documento"* **select name, email from /header/authlist/author**
3	Proy.	*"Obtener toda las referencias de la especificación"* **select * from /back/div1/div2/blist**
4	Proy.	*"Obtener todas las definición de terminos referentes a xml"* **select @id from /body/div1/p/termdef**
5	Proy.	*"Obtener toda la bibliografía referente a la especificación de xml"* **select bibl from /back/div1/div2/blist**
6	Proy.	*"Seleccionar todos los titulos y parrafos del cuerpo principal de la especificación de xml"* **select head, p from /body/div1**
7	Proy.	*"Seleccionar los terminos y su identificador, del cuerpo principal de la especificación"* **select @id, @term from /body/div1/p/termdef**
8	Proy.	*"Obtener los roles de los miembros del W3C XML Working Group"* **select role from /back/inform-div1/orglist/member**
9	Selec.	*"Seleccionar toda la introducción"* **select * from /body/div1 where head = Introduction**
10	Selec.	*"Seleccionar los parrafos de la introducción"* **select p from /body/div1 where head = Introduction**
11	Dist.	*"Obtener todos los códigos sin repetir de la definición de terminos del cuerpo de la especificación de xml"*

		select distinct code from **/body/div1/div2/p/termdef**
12	Proy.	*"Seleccionar nombre y email de los autores"* **select name, email from** **/header/authlist/author**
13	Join	*"Dados 2 documentos, el primero la especificación de xml y el segundo un conjuto de emails y pais de origen, obtener nombre y pais de los autores de la especificación de xml"* **select a.name, b.country** **from /parent.a, /parent.b** **where a.email = b.email**
14	Order	*"Obtener todos los encabezados del cuerpo de la especificación de xml, ordenados alfabeticamente"* **select head from /body/div1 orderby head**
15	Order	*"Obtener la definición de terminos del cuerpo del documento ordenado alfabeticamente"* **select termref from /body/div1/p orderby termref**
16	Agreg	*"Contar la cantidad de terminos definidos en el cuerpo del documento"* **select count(termref) from /body/div1/p**

Resultados

A continuación se muestra un cuadro de tiempos de respuesta, clasificados por tipo de operación.



Como se observa en el cuadro anterior, los tiempos de respuesta siguen el patrón de los ensayos anteriores, donde los archivos de entrada tienen el mismo volumen, por lo que en la sección siguiente haremos pruebas de esfuerzo, variando el tamaño del mismo.

6.5 Ensayo sobre base de datos de texto

En este experimento se trabajará sobre el archivo `nt.xml`. Es un exponente de base de datos de texto, representante de un texto religioso escrito en XML. El tamaño del archivo es de 1Mb, con una cantidad de 16810 nodos XML, conteniendo strings extensos y un bajo nivel de anidamiento.

Este ensayo analiza la capacidad del sistema para mantenerse estable ante la presencia de documentos de gran tamaño. Para ello el archivo `nt.xml` (tamaño = 1 Mb), fue concatenado en forma sucesiva, haciendo crecer su tamaño en forma exponencial. De esta forma se creo el archivo `nt2.xml`, que es la concatenación del contenido original de nt.xml, quedando siempre un único nodo raíz (`<tstnt>`) y de tamaño 2Mb. De esta manera se crearon archivos de 4, 8 y 16Mb (`nt4.xml, nt8.xml nt16.xml`), sobre los cuales se aplico el mismo conjunto de consultas.

A continuación se muestra una breve porción del documento nt.xml:

```
<tstmt>
  <coverpg>
    <title>The New Testament</title>
    <title2>One of a group of four religious works marked up for
    electronic publication from publicly available sources</title2>
```

```
    <subtitle>
      <p>SGML version by Jon Bosak, 1992-1994</p>
      <p>XML version by Jon Bosak, 1996-1998</p>
      <p>The XML markup and added material in this version are
      Copyright © 1998 Jon Bosak</p>
    </subtitle>
      .
      .
      .
  <bookcoll>
    <book>
      <bktlong>The Gospel According to SAINT MATTHEW.</bktlong>
      <bktshort>Matthew</bktshort>
      <chapter>
                <chtitle>Chapter 1</chtitle>
          <v>The book of the generation of Jesus Christ, the son of
        David, the son of Abraham.</v>
          <v>Abraham begat Isaac; and Isaac begat Jacob; and Jacob
        begat Judas and his brethren;</v>
          <v>And Judas begat Phares and Zara of Thamar; and Phares
        begat Esrom; and Esrom begat Aram;</v>
          <v>And Aram begat Aminadab; and Aminadab begat Naasson; and
        Naasson begat Salmon;</v>
                .
                .
                .
      </chapter>
     </book>
    </bookcoll>
  </tstmt>
```

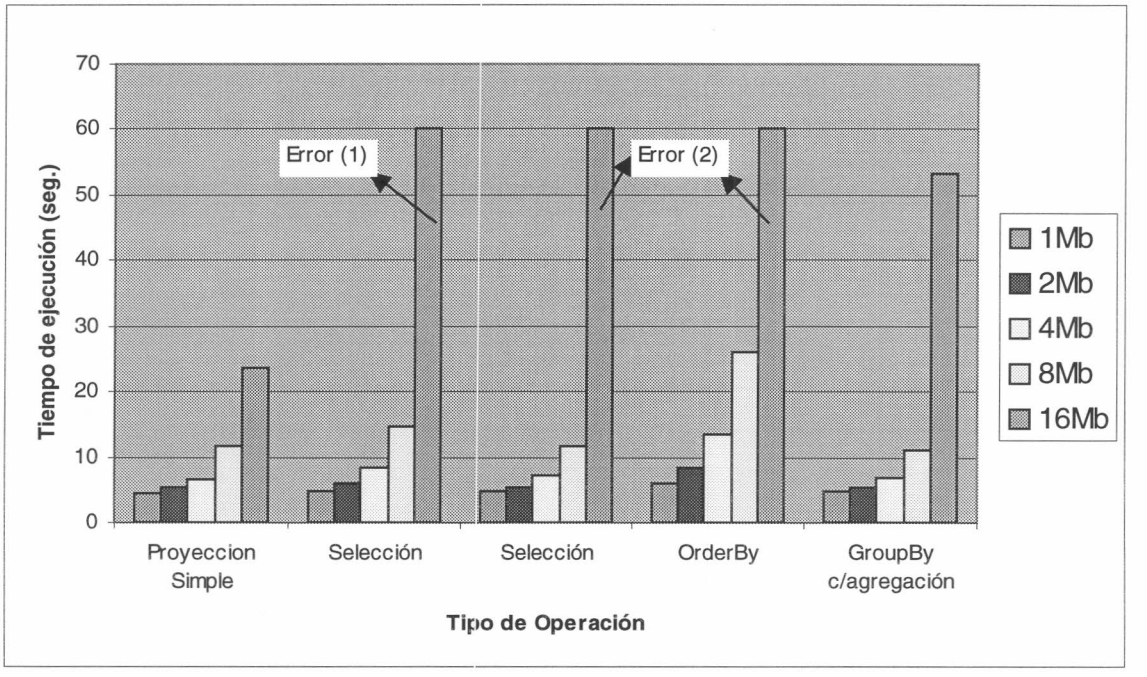
Consultas realizadas:

Nro.	*Tipo.*	***Descripción***
1	Proy.	*"Recuperar todos los títulos del nuevo testamento"* **select title** **from /coverpg**
2	Selec.	*"Seleccionar todo lo que escrito por San Lucas"* **select *** **from /bookcoll/book** **where bktshort = 'Luke'**
3	Selec.	*"Recuperar todos los primeros capitulos de todos los libros"* **select *** **from /bookcoll/book/chapter** **where chtitle = 'Chapter 1'**
4	Order	*"Mostrar todos los libros y su contenido*

		ordenado por autor" **select *** **from /bookcoll/book** **orderby bktshort**
5	Group	*"Contar los capitulos escritos por cada autor"* **select bktshort, count(chapter)** **from /bookcoll/book** **groupby bktshort**

Resultados

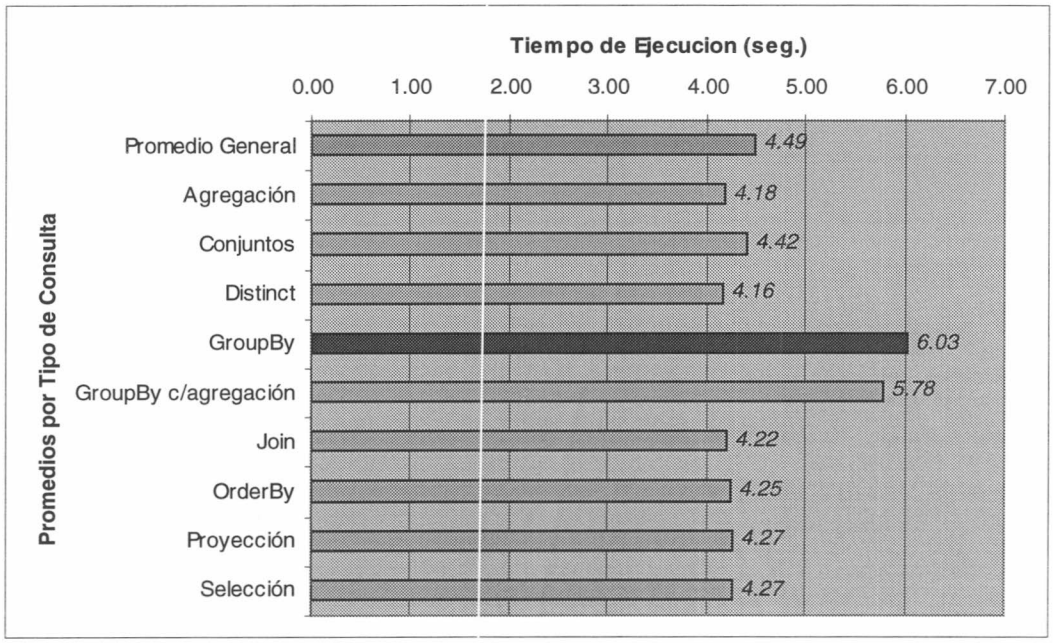
A continuación se muestra un cuadro de tiempos de respuesta, clasificados por tipo de operación y subdividido por el archivo consultado. Por ejemplo, en el eje X, para la primera operación "Proyección Simple" se presentan 5 valores, correspondientes a la misma consulta sobre los 5 archivos de entrada (nt.xml, nt2.xml, ..., nt16.xml)



Los tiempos de respuesta obtenidos se incrementaron en forma gradual de acuerdo al tamaño del archivo. El sistema se mantuvo estable hasta que se empezó a testear el archivo de 16Mb (nt16.xml). Las operaciones en el que el resultado no es de gran tamaño, el sistema respondió siguiendo el patrón de los ensayos anteriores. En el caso de las consultas 2, 3 y 4, donde el resultado es de gran tamaño, se presentaron errores de manejo de memoria. Dichos errores se presentan en el gráfico con los rótulos Error(1) y Error(2). El primer error es propio de JAVA, por manejo excesivo de memoria; el caso del Error(2), se debió a que JDOM no tuvo capacidad, también por problemas de manejo de memoria, para escribir el archivo resultado en disco.

6.6 Resumen

Como resultado de los experimentos realizados en el presente capítulo, se observó que el sistema sigue un patrón de tiempos de respuesta según el tipo de consulta que se procese. A continuación se presenta un gráfico con un promedio por tipo de operación de los 3 primeros ensayos.



Se observa que en general el promedio del tiempo de respuesta es similar para todas las operaciones, siendo la de mayor tiempo usado la operación de GroupBy. Estos tiempos crecen linealmente con el tamaño del archivo de entrada.
Quedó demostrado con el ensayo final, con las secuencias de archivos de la sección 6.5, que una de las principales limitaciones del sistema es el manejo de memoria, ya que el documento es totalmente cargado en memoria. Queda planteado para una implementación más eficiente que JDOM trabaje con solo una parte del documento en memoria y el resto lo deje en disco, para el manejo de grandes volúmenes de información, ya sea para el ingreso del documento a consultar como para almacenar resultados intermedios de ciertas consultas.

Capítulo 7

Conclusiones

En este capítulo se presentarán las conclusiones del proyecto. Se analizan limitaciones del lenguaje propuesto, se discuten algunas características de los experimentos realizados y se presentan puntos a tratar en futuros trabajos.

7.1 Limitaciones del lenguaje

Basados en los experimentos realizados, podemos observar algunas limitaciones a la expresividad del lenguaje, producto del planteo que hace el álgebra para el manejo de las consultas y que nuestro lenguaje respeta.

Consideremos nuevamente la consulta 1 de la sección 6.3:

```
"Obtener el nombre de los personajes de la obra"
select PERSONA from /PERSONAE
```

Esta consulta genera un resultado basado en lo que especifica el álgebra, pero presenta un inconveniente en el caso que al resultado obtenido se le quisiera aplicar una nueva consulta, como pueden ser funciones de ordenamiento o de agregación principalmente. Veamos el resultado:

```
<root>
   <parent>
      <PERSONA>DON PEDRO, prince of Arragon.</PERSONA>
      <PERSONA>DON JOHN, his bastard brother.</PERSONA>
      <PERSONA>CLAUDIO, a young lord of Florence.</PERSONA>
      <PERSONA>BENEDICK, a young lord of Padua.</PERSONA>
      <PERSONA>LEONATO, governor of Messina.</PERSONA>
      <PERSONA>ANTONIO, his brother.</PERSONA>
      <PERSONA>BALTHASAR, attendant on Don Pedro.</PERSONA>
      <PERSONA>FRIAR FRANCIS</PERSONA>
      <PERSONA>DOGBERRY, a constable.</PERSONA>
      <PERSONA>VERGES, a headborough.</PERSONA>
      <PERSONA>A Sexton.</PERSONA>
      <PERSONA>A Boy.</PERSONA>
      <PERSONA>HERO, daughter to Leonato.</PERSONA>
      <PERSONA>BEATRICE, niece to Leonato.</PERSONA>
      <PERSONA>Messengers, Watch, Attendants, &amp;amp;c. </PERSONA>
   </parent>
</root>
```

Los elementos PERSONA, son hijos de un único elemento PARENT, ya que en el documento original eran hijos de un mismo elemento PERSONA. Sobre esta representación no funcionan como sería deseado las funciones de agregación o de ordenamiento, ya que no toma a los elementos PERSONA como elementos independientes.
Queda abierto a implementar una función especial o a redefinir el álgebra para independizar los elementos PERSONA. A continuación se muestra el resultado que se hubiese querido obtener para poder realizar un mejor análisis de datos:

```
<root>
   <parent>
      <PERSONA>DON PEDRO, prince of Arragon.</PERSONA>
   </parent>
   <parent>
      <PERSONA>DON JOHN, his bastard brother.</PERSONA>
   </parent>

     ...
    <parent>
      <PERSONA>Messengers, Watch, Attendants, &amp;amp;c. </PERSONA>
     </parent>
</root>
```

Relacionado con el ejemplo anterior, también se plantea la necesidad de convertir en elementos a los atributos resultado de una proyección. Por ejemplo, dada la siguiente consulta, que es la proyección de atributos sobre un documento XML representante de una biblioteca:

"Obtener el género de cada libro"

```
select @genre from /book
```

Obteniendo el siguiente documento como resultado:

```
<root>
   <parent genre="autobiography" />
   <parent genre="novel" />
   <parent genre="philosophy" />
</root>
```

Sería de mucha utilidad plantear una función que convierta en elementos dichos atributos, para un posterior análisis. En el ejemplo anterior la función a definir, tendría como resultado el siguiente documento, donde los atributos `genre`, se convierten en elementos:

```
<root>
    <parent>
          <genre> autobiography </genre>
    </parent>
    <parent>
          <genre> novel </genre>
    </parent>
    <parent>
          <genre> philosophy </genre>
    </parent>
</root>
```

A continuación recordamos la consulta 3 de la sección 6.5:

"Recuperar todos los primeros capítulos de todos los libros"

```
select *
from /bookcoll/book/chapter
where chtitle = 'Chapter 1'
```

Supongamos que se desea recuperar el primer capítulo escrito por Matthew, en CXXML seria necesario realizar 2 consultas: la primera que recupere los libros escritos por Matthew:

```
select chapter
from /bookcoll/book
where bktshort = 'Matthew'
```

Luego, sobre el resultado de la consulta anterior habría que recuperar todos los capítulos cuyo título es "Chapter 1":

```
select *
from /parent/chapter
where chtitle = 'Chapter 1'
```

Si se planteara una extensión del álgebra y consecuentemente enriqueciendo el lenguaje, se podría definir la consulta anterior en un solo paso haciendo una función que permita establecer condiciones sobre otras profundidades de path, quedando una consulta del siguiente estilo:

```
select *
from /bookcoll/book/chapter
where ..bktshort = 'Matthew' and chtitle = 'Chapter 1'
```

Como se ve en el ejemplo anterior, queda planteado para un trabajo futuro la definición e implementación del operador ".." para navegar sobre elementos de un path de nivel menor al actual del `from`.

7.2 Comparación con otros lenguajes

A modo de resumen de características de CXXML se presenta a continuación un cuadro comparativo con otros lenguajes de consulta de XML:

	CX-XML	LOREL	XML-QL	UnQL	Xquery	XQL
Proyección de elementos	SI	SI	SI	SI	SI	SI
Proyección de atributos	SI	NO	SI	NO	SI	NO
Selección	SI	SI	SI	SI	SI	SI
Funciones de agregación	SI	SI	NO	NO	NO(2)	NO
Groupby	SI	SI(1)	SI	SI(1)	NO(2)	SI
SubQuery	SI	SI	SI	NO	NO(2)	SI
Join	SI	SI	SI	NO	NO(2)	SI
Groupby con funciones de agregación	SI	NO	NO	NO	NO(2)	NO
Orden	SI	NO	SI	NO	NO(2)	SI

Operaciones de conjunto	SI	SI	NO	NO	SI	NO
== (comparación de elementos complejos)	SI	SI	NO	NO	SI	NO
Query sobre estructuras no conocidas	NO	SI	NO	NO	NO	NO
Transformación de Elementos y/o Atributos en nuevos Elementos	NO	NO	NO	SI	NO[2]	NO
Consulta sobre elementos pertenecientes a otros niveles del Path del From	NO	NO	NO	NO	SI	SI

[1] pueden ocurrir elementos duplicados al agrupar

[2] podria cumplir con la funcionalidad solicitada construyendo aplicaciones con el lenguaje, pero no esta definida atómicamente la expresión en cuestión

7.3 Trabajos en curso

En la actualidad se está trabajando sobre nuevos lenguajes de consulta que atacan varios de los puntos mencionados en este trabajo. CXQuery, por ejemplo, es un nuevo lenguaje declarativo que permite realizar consultas y actualizaciones sobre documentos XML. A diferencia de CXXML, permite definir vistas sobre los documentos y utiliza la sintáxis y la semántica de los lenguajes de consulta basados en restricciones [RC02]. Adicionalmente utiliza los DTDs para especificar el esquema del documento de salida.

7.4 Conclusiones

En este trabajo se presentó un nuevo lenguaje de consulta para archivos XML basado en el álgebra estándar de W3C. El mismo introduce mejoras sustanciales comparado con otros lenguajes, como quedó reflejado en la sección anterior, además de utilizar una sintaxis similar a la de SQL. El código intermedio que generan las consultas *CXXML*, alineadas con el álgebra de W3C, deja abierto trabajos a futuro para la optimización de los principales algoritmos del lenguaje. Si bien los tiempos de ejecución de las distintas consultas no son óptimos, esto se debe en parte a las herramientas que usamos para acceder físicamente a los datos y a la forma en la cual recorremos el archivo que está alineada con la especificación del álgebra pero que podría optimizarse. Dejamos planteada ésta y otras mejoras para implementaciones futuras.

Apéndice

Gramática CXXML

La gramática asociada a nuestro lenguaje de consultas es la siguiente:

<query> → **SELECT** [**DISTINCT**] *<select_first>*
FROM *<from_list>*
[**WHERE** *<condition>*]
[**GROUP_BY** *<group_by_list>*]
[**UNION** *<last_query>*]
[**INTERSECTION** *<last_query>*]

<query_with_order> → *<query>*
[**ORDER_BY** *<order_by_list>* [**DESC**]]

<select_first> → *
| *<select_list>*

<select_list> → *<select_expr>, <select_list>*
| *<aggregate> ([<var> .] <atributo>)*
| *<aggregate> (*)*

<select_expr> → [*<var>* .] *<atributo>*

<from_list> → *<path_expr>*[.*<var>*], *<path_expr>*[.*<var>*]
| *<path_expr>*[.*<var>*]
| *<path_expr>*[.*<var>*], *<last_query>*[*<var>*]
| *<last_query>* [*<var>*]

<group_by_list> → [*<var>* .] *<atributo>*

<last_query> → **SELECT** [**DISTINCT**] *<select_first>*
FROM *<last_from_list>*
[**WHERE** *<simple_pred>*]
[**GROUP_BY** *<group_by_list>*]
[**ORDER_BY** *<order_by_list>*]

<last_from_list> → *<path_expr>* [.*<var>*], *<path_expr>* [.*<var>*]
| *<path_expr>* [.*<var>*]

<order_by_list> → *<order_by_listr>, <order_by_expr>*
| *<order_by_list>*

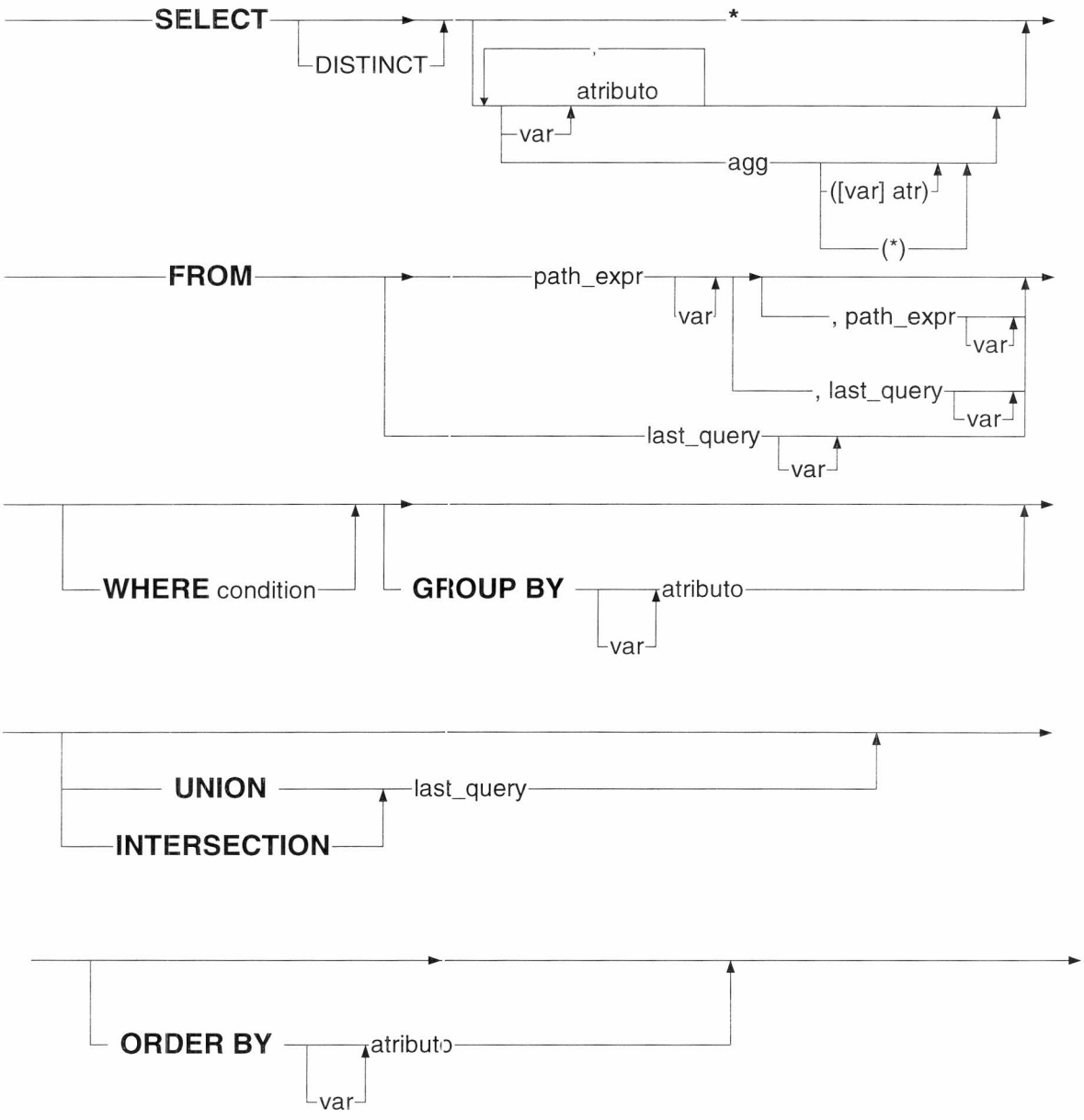
<order_by_expr> → *<order_by_listr>, <order_by_expr>*
[*<var.>*] *<atributo>*

<var>	→	Identifier
<atributo>	→	Identifier
<path_expr>	→	*<label>*
	|	*<label>* / *<path_expr>*
<label>	→	String restricted char set
	|	#
	|	**?**
<condition>	→	*<predicate>***OR** *<predicate>*
	|	*<predicate>***AND** *<predicate>*
	|	(*<predicate>*)
<predicate>	→	*<simple_pred>*
	|	*<quantified_pred>*
<simple_pred>	→	*<simple_operand>* *<comp_op>* *<simple_operand>*
<quantified_pred>	→	*<simple_operand>* *<comp_op>* **ANY** *<last_query>*
		<simple_operand> *<comp_op>* **ALL** *<last_query>*
		<simple_operand> **IN** *<last_query>*
		<simple_operand> **NOT IN** *<last_query>*
<simple_operand>	→	[*<var>* .] *<atributo>*
	|	*<constant>*
<constant>	→	*<digito>* *<label>*
	|	"*<label>*"
<digito>	→	**0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
<comp_op>	→	**= | < | <= | > | >= | !=**
<aggregate>	→	**COUNT | MAX | MIN | SUM | AVG**

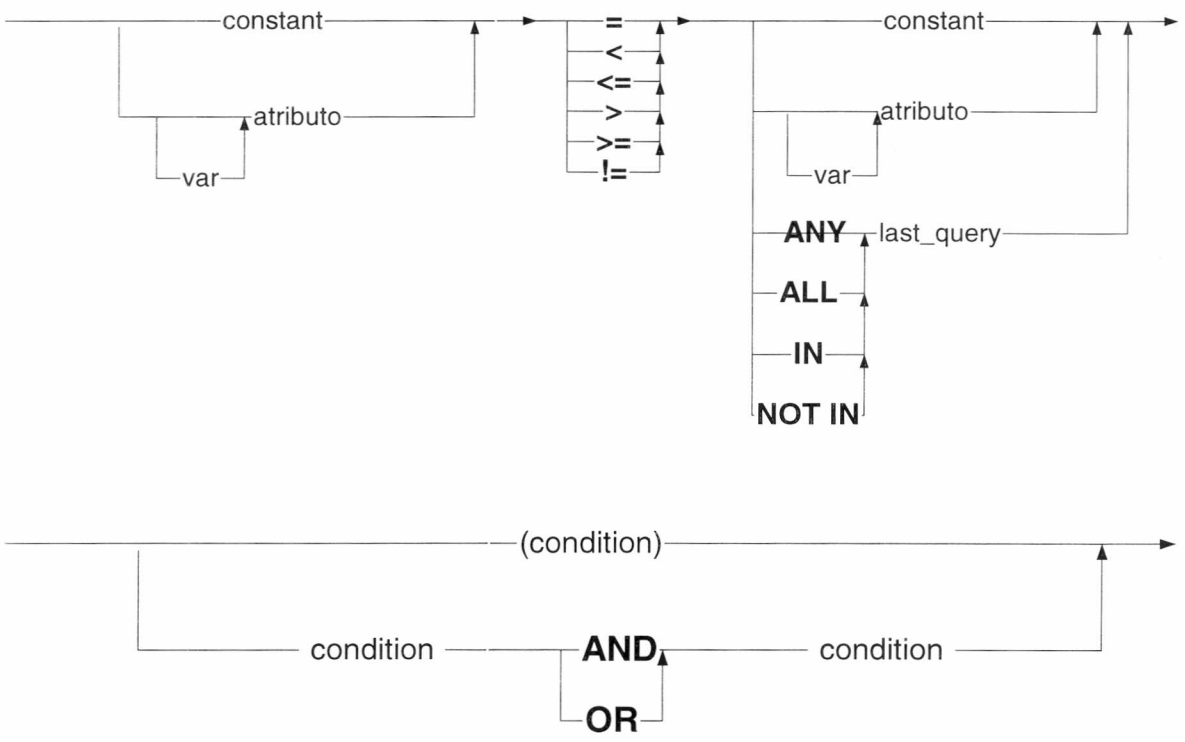
Sintaxis

La sintaxis asociada a la gramática es la siguiente:

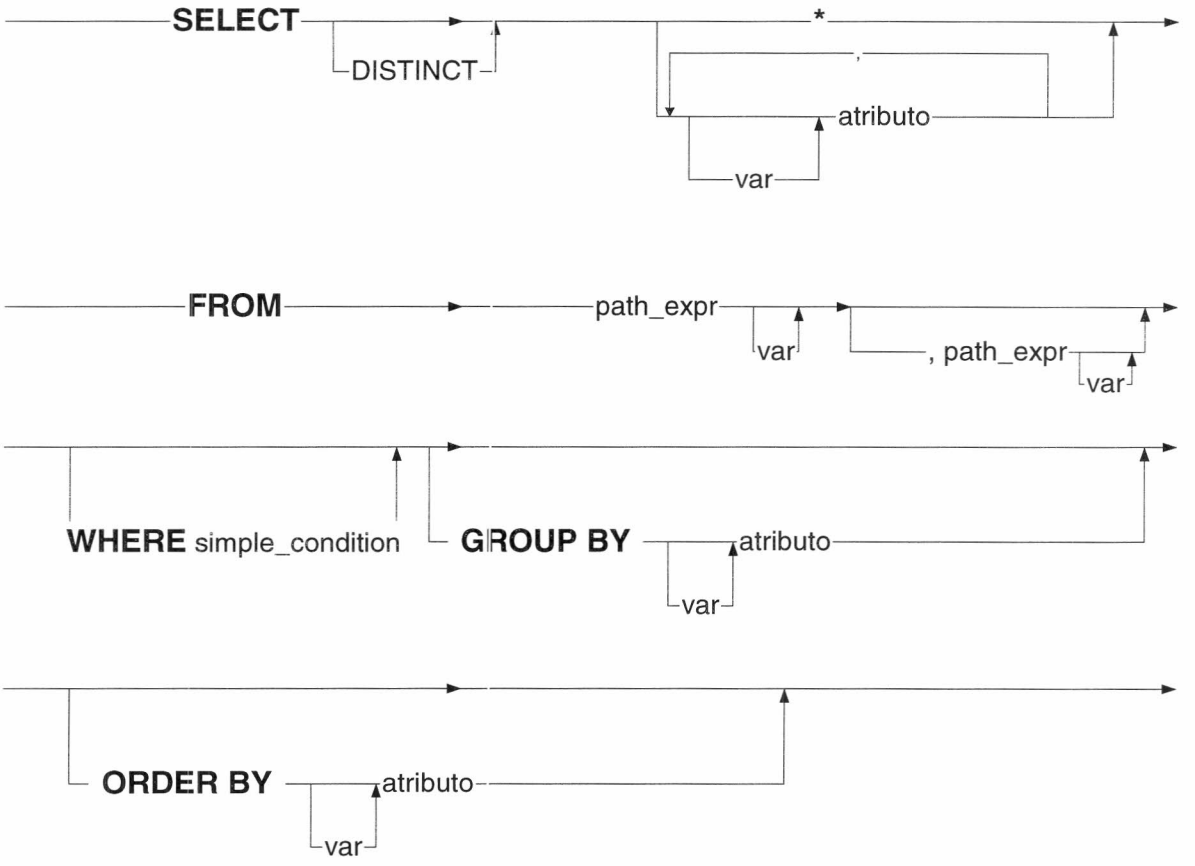
- SELECT



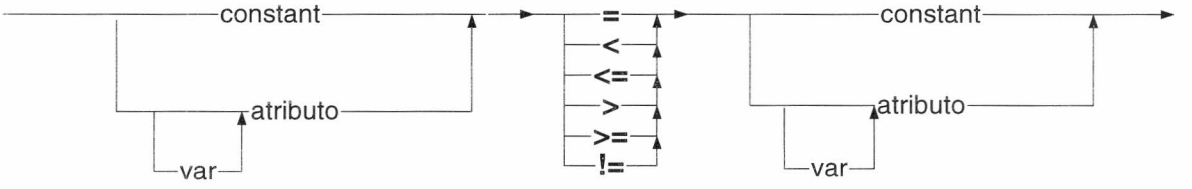
- CONDITION


constant
atributo
var
=
<
<=
>
>=
!=
constant
atributo
var
ANY
ALL
IN
NOT IN
last_query
(condition)
condition
AND
OR
condition

- SUBQUERY



- SIMPLE CONDITION



Bibliografía

[ABS00] Abiteboul, S., Buneman, P., Suciu D. Data on the Web - From Relations to Semistructured Data and XML. Morgan Kaufmann Publishers, 2000.

[AQM+97] Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J. The Lorel query language for semistructured data. *Intl. J. On Digital Libraries*, 1(1):68-88, April 1997.

[BENCHXML02] http://www.sosnoski.com/opensrc/xmlbench accedida el dia 04-03-2002

[BUN97] Buneman, P. Semistrcutured Data, PODS 1997: 117-121 Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona. ACM Press 1997, ISBN 0-89791-910-6

[BCD89] Bancilhon, F., Cluet, S., and Delobel, C. Query languages for object-oriented database system: the O_2 proposal. In Proc. Of the Second Intl. Work. On Data Base Programming Languages, 1989.

[BDHS96] Buneman, P., Davidson, S., Hillebrand, G., Suciu, D. A query language and optimization techniques for unstructured data. In SIGMOD, pages 505-516. Montreal, 1996.

[CD99] Clark, J., DeRose, S., editores. XML Path Language (XPATH), 1999. - Version 1.0 - http:http://www.w3.org/TR/1999/REC-xpath-19991116, 1999.

[DFFLS98] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Suciu, D. XML-QL: A query Language for XML, 1998. http: //www.w3.org/TR/1998/NOTE-xml-ql-19980819

[FFMRSW00] Fankhauser, P., Fernandez, M., Malhotra, A., Rys, M., Simeon, J., Wadler, P., editores. The XML Query Algebra, 2000 - http://www.w3.org/tr/2000/wd-query-algebra-20001204

[FR01] Fernandez, M., Robie, J. editores. XML Query Data Model, 2001 - http://www.w3.org/TR/2001/WD-query-datamodel-20010215

[JDOM] http://www.jdom.org

[QRSUW97] Quass, D., Rajaraman, A., Yehoshua, S., Ullman, J., Widom, J. Querying Semistructured Heterogeneous Information, 1997. University of Stanford. http://www.db.stanford.edu/tsimmis

[RC02] Revesz, P., Chen, Y. CXQuery: A novell XML Query Language, 2002. University of Nebraska. http://www.ssgrr.it/en/ssgrr2002w/papers/216.pdf

[RCF00] Robie, J., Chamberlin, D., Florescu, D. Quilt: an XML Query Language – http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html

[SUC98] Suciu, D. Semistructured Data and XML. The 5th International Conference of Foundations of Data Organization (FODO'98), Kobe, Japan, November 12-13, 1998

[SY98] Sagiv, P., Yariv T., UnQL Presentation, 1998.
http://www.cs.huji.ac.il/~sdbi/1998/yarivi/sdbi

Alfredo Reino Romero - Universidad de Colima. México. Introducción a XML versión 2.0. http://www.ibium.com/alf/xml/index.asp

The World Wide Web Consortium (W3C). http://www.w3.org

[W3CXQ] http://www.w3.org/TR/xquery

[W3CXQL] XQL: A Query Language for XML Data.
http://www.w3.org/TandS/QL/QL98/pp/flab.txt

[W3CXQA] The World Wide Web Consortium (W3C)
http://www.w3.org/TR/2000/WD-query-algebra-20001204

[W3CXSL] The Extendible Stylesheet Languaje (XSL), http://www.w3.org/Style/XSL/

[W3CSCHEMA] World-Wide Web Consortium XML Schema , Estructuras
http://www.w3.org/TR/xmlschema-1