Universidad Nacional Autónoma de México

Facultad de Ciencias

Definición y manejo de la metainformación
de una base de datos

T E S I S

QUE PARA OBTENER EL TÍTULO DE
Licenciado(a) en Ciencias de la Computación
P R E S E N T A N:

Bautista Santiago Crevel
y
Ramírez Vigueras Adriana

Tutor de tesis:
Lic. Manuel Alberto Sugawara Muro

Asesor de tesis:
Dra. Elisa Viso Gurovich

2006

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

VNIVERSIDAD NACIONAL
AVPNOMA DE
MEXICO

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

"Definición y manejo de la metainformación de una base de datos"

realizado por Adriana Ramírez Vigueras y Crevel Bautista Santiago

con número de cuenta 095286819 y 098512982 respectivamente, quienes cubrieron los créditos de la carrera de:

Lic. en Ciencias de la Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	Lic. en C.C. Manuel Alberto Sugawara Muro
Codirectora Propietario	Dra. Elisa Viso Gurovich
Propietario	M. en C. Javier García García
Suplente	M. en I. Ma. de Luz Gasca Soto
Suplente	M. en C. José de Jesús Galaviz Casas

Consejo Departamental de Matemáticas

Dr. Francisco Hernández Quiroz

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

A mi mamá,
tú sabes cuanto te quiero.

Crevel

A mis ejemplos a seguir en la lucha incanzable
con la vida y a quienes debo todo,
a mis padres Esperanza y Marcos, los quiero mucho.

Adriana

Agredecimientos

Crevel

A *Horte*, mi mamá, por apoyarme infinitamente durante toda mi vida, por ser mi principal soporte, en especial estos últimos años. Por darme siempre más de lo necesario y más de lo que esperaba sin pedir nada a cambio. Por guiarme hasta aquí, por cuidarme, por regañarme, por amarme, por educarme ... en fin, por ser como eres, por ser mi madre.

A mi abuelita *Eva*, por ser la más consentidora y comprensiva persona que he conocido. Por darme consejos, aún cuando no se los pidiera.

A mi hermana *Naye*, por ser mi principal complice y amiga. Por cuidarme y quererme tanto aunque no lo merezca.

A mis hermanos *Care* y *Hervin*, por ser los más grandes compañeros de juego que he tenido y porque sé que siempre estarán ahí cuando los necesite.

A mi tío *Abel*, mi tía *Lupe* y mis primos *Leydi* y *Beto*, por permitirme compartir con ellos algunos de los momentos más felices que tengo de mi infancia. También por seguir siendo un apoyo muy cercano.

A mi abuelito *Amancio*, a mis tíos *Uri* y *Nacho*, a mis tías *Nelly*, *Mirza*, *Roci* y *Rosario*, a mis primos, por ser parte de ese gran núcleo que es mi familia.

A mi *padre*, que con sus acciones, me ha enseñado que la vida a veces no es fácil pero hay que enfrentarla.

A mis amigos *Tita*, *Gil* y *Vicky*, por haber sido la mejor compañia durante estos últimos años, tanto en lo personal como en lo escolar. Por compartir con ellos grandes momentos de tensión, educación y por supuesto, de diversión.

A *Manuel Sugawara*, mi tutor, por sus aportaciones e inagotable paciencia durante todo el tiempo en que se trabajo para sacar adelante esta tesis.

A *Elisa Viso*, *José Galaviz*, *Lucy Gasca* y *Javier García*, los mejores profesores que he tenido, por su gran dedicación y apoyo dentro y fuera del salón de clases, así como para la realización de este trabajo.

A *Mónica Leñero*, por haberme dado asilo, apoyo y consejos.

A todos y cada uno de ellos, Gracias.

Agredecimientos

Adriana

Antes que nada quiero agradecer a mis padres por su apoyo incondicional para seguir adelante, sin importar los obstáculos que se presentaran, gracias por todo.

A mi querido hermano *Vale*, de verdad, muchas pero muchas gracias por aceptar ser mi amigo, algunas veces mi consejero y otras tantas mi complice, además por tu apoyo infinito; nunca voy a olvidar que gracias a ti siempre llegué a tiempo a las clases de Javier García.

A mi hermano *Guillermo*, gracias por todos los estímulos que me brindabas durante mi vida académica.

A mi hermana *Esperanza*, por no quererme desde que nací, es broma se que me quieres y me aprecias tanto como yo a ti.

A mi hermano *Marcos*, por aconsejarme estudiar esta carrera, lo cual ha sido la mejor decisión que he tomado en mi vida.

A mi hermano *Sergio*, gracias por compartir tus experiencias conmigo.

A mi hermano *Juan*, por compartir muchas veces el sentido del humor que siempre traes contigo.

A mi cuñada consentida (que conste, eh!) *Gaby* por darme ánimo cuando más lo necesité y ayudarme a iniciar este camino tan difícil que ahora concluyo.

A *Chabela* junto con su tesoro, gracias por darme su amistad incondicional.

Y ya para terminar con esta extensa familia que tengo, pero muy divertida, gracias a todos mis sobrinos, en especial a los más grandes: *Memo*, *Naty*, *Pepé* y *Vivis* por ser mis compañeros incanzables de juego.

Ahora no tengo palabras suficientes para agradecer a mi novio *Gil Bautista García Cano*, quien nunca ha cesado de ayudarme y por ser mi compañero de lucha incanzable en la buenas y en las malas; espero todo salga tal y como lo planeamos, te quiero.

A mi director de tesis por aceptar guiarme en este trabajo que duró casi 4 años, gracias por tu valioso tiempo sin esperar nada a cambio, de verdad mil gracias *Manuel*.

A la mejor profesora de la carrera de Ciencias de la Computación, mi co-directora *Elisa Viso Gurovich*, muchas gracias por aceptar asesorarme y por aconcejarme enfrentar el reto que implica estudiar la maestría.

Al resto de mis sinodales, gracias por el tiempo que dedicaron para revisar este trabajo, con especial admiración y respeto a *Javier García García*, excelente profesor, ejemplo a seguir y a quien debo el gusto por las bases de datos, a *José de Jesús Galaviz Casas* por aceptar ser mi amigo y ayudarme siempre que se lo he pedido, además de su extensa lista de chistes, famosos ya en la facultad y a *Lucy Gazca Soto*, nunca olvidaré tus palabras, que tal vez no recuerdes: "Tu nunca estas sola, eres suficiente para acompañarte a ti misma". Gracias por todo y por las clases en el CDM.

A uno de los profesores (*Gustavo Arturo Márquez Flores*) que me tendio la mano para iniciar mi labor como ayudante de la facultad, gracias por todo y por sus largas pláticas por teléfono dándome siempre consejos muy acertados.

Gracias a mi mejor amigo de la facultad y gran compañero de trabajo por su energía de siempre para segir, si burris (*Crevel Bautista*), eres tú.

Gracias a *Vicky* y *Villedita* por ser tan buenos amigos y con los que he vivido experiencias que ninguno olvidará jamás.

A la mejor jefa que alguien pudiera tener, *Mónica Leñero Padierna*, muchas gracias por los momentos que me has dejado compartir contigo y por apoyarme siempre.

A *Lisbeth Heras* por darme la oportunidad de involucrarme académicamente en la DGSCA, muchas gracias.

Por último, agradezco profundamente a los que directa o indirectamente se vieron involucrados en la realización de este trabajo y a los que olvidé, aunque no intencionalmente.

Muchas gracias a todos.

Índice general

Lista de listados

Lista de figuras

Lista de tablas

Introducción

Durante los últimos años, el desarrollo de la tecnología ha permitido el planteamiento de nuevos problemas, en los cuales se ve involucrado un conjunto de datos para obtener su solución.

Dentro de la computación, la forma de organizar y almacenar dicho conjunto de datos siempre ha sido un tema de interés y uno de los métodos desarrollados para tal fin ha sido la concentración de los mismos en lo que hoy en día se conoce como una base de datos (BD).

Por supuesto, el campo de las bases de datos estaría incompleto si sólo se considerara la forma en la que se organizan los datos para almacenarlos, motivo por el cual también se ha definido una manera de poder acceder a ellos. Esta manera se basa principalmente en el desarrollo de un lenguaje que se denomina *SQL (Structured Query Lenguage)*, de tal manera que al utilizar expresiones construidas a partir de él, se permita obtener un subconjunto de datos de los existentes en la BD.

Una vez descrito a grandes rasgos el concepto de una BD, podemos establecer que en cierto momento, esta se encontrará en alguna de las siguiente fases:

Creación. Etapa en la que se establece la forma en que se organizará el conjunto de datos en cuestión y de qué manera se interrelacionarán estos últimos.

Modificación. El conjunto de datos que contiene la BD podría modificarse en cualquier momento.

Utilización. Se accede a un subconjunto de datos de nuestro intereś dentro de la BD en cierto momento sin que estos sufran modificación alguna.

A la incorporación de los conceptos descritos anteriormente en uno solo se le conoce como Sistema Manejador de Base de Datos (SMBD). Entre los sistemas actuales de este tipo podemos encontrar a *PostgreSQL* y *Oracle*.

Hoy en día es muy común que se utilice un sistema como estos en cualquier organización que necesite concentrar un conjunto de datos y a través de este sistema manipularlos. Sin embargo, hacer uso directo de un SMBD implica tener otros conocimientos aparte del concepto de BD; esto debido a que, como lo mencionamos, se necesita, entre otros

aspectos, conocer SQL, ya que cualquier interacción con el manejador se hace a través de una expresión en este lenguaje. Es aquí donde se impone la restricción de que solamente el usuario que conozca SQL podrá hacer uso de los datos en cuestión.

Por otro lado, alguien que esté involucrado en este campo necesita conocer sus bases o fundamentos, lo cual, dentro de este trabajo, se traduce en entender el modelo relacional (ya que sólo trataremos con BDs de este tipo). Sin embargo, se ha llegado a creer que el conocer SQL significa conocer el modelo relacional y esto es un error. Además, si para algunos principiantes dicho modelo resulta algo abstracto, el manejo de varios de los muchos dialectos que existen hoy en día del lenguaje SQL podría ser un poco más complicado de lograr.

Dado el problema anterior, uno de los objetivos de este trabajo será proporcionar una interfaz por medio de la cual el usuario de un SMBD pueda llevar a cabo la primera etapa de las descritas anteriormente (creación), por medio de la incorporación de un nuevo lenguaje (definido en este trabajo) que represente a los elementos de la BD, de tal manera que dicha interfaz juegue el papel de intermediario entre el usuario y el manejador y proporcione cierta ayuda al generar expresiones adecuadas en el lenguaje SQL.

Al obtener dichas expresiones en SQL, el usuario de esta interfaz podrá implementar su esquema de BD en cualquier SMBD, ya que en principio fueron tomadas en cuenta las definiciones estándar, es decir, sin apegarse a ningún dialecto en particular. Además, se tiene contemplada una definición que permita extender los elementos básicos de tal manera que al hacer uso de ella, el resultado pueda ser utilizado para un SMBD en particular, que es lo mismo que utilizar un dialecto [1].

Hablando más específicamente, los elementos que serán considerados para este fin serán los más comunes o estándares de los que se compone una BD, como son las tablas, las funciones, los *triggers*[2], las vistas y los índices.

La interfaz hace uso de *XML (Extensible Markup Language)*, el cual es un lenguaje que ayuda a la manipulación de información de una manera flexible y sencilla, para definir un lenguaje por medio del cual podemos generar fácilmente las expresiones deseadas en SQL, además de representar en su totalidad el esquema de una BD de manera intuitiva, es decir, podremos tener un documento en XML que describa claramente el esquema de la BD teniendo en cuenta, por ejemplo, que las tablas se componen de atributos y un conjunto de restricciones sobre ellos, lo cual es una clara ventaja sobre el conjunto de instrucciones en SQL que sólo la implementan en el SMBD.

Otra ventaja que nos da el uso de esta tecnología es permitir la introducción de un nuevo elemento, que jugará el papel de comentario para cada uno de los elementos de los

[1] Este trabajo sólo considerará el dialecto para el SMBD Postgresql.

[2] Se hará uso de esta palabra debido a que la traducción literal a "gatillo" o "disparador" no refleja adecuádamente el sentido del término.

cuales se compone la BD, como son las tablas. Si quisiéramos tener algo similar por medio de SQL, primeramente tendríamos que crear los elementos de la BD con las instrucciones adecuadas y posteriormente relacionarlos con sus respectivos comentarios por medio de otra instrucción. Por lo que si quisiéramos consultar el comentario y el nombre del elemento al que pertenece de manera simultánea, tendríamos que ejecutar una consulta (*query*) dentro del SMBD que nos permitiera hacerlo.

Otra tarea que llevará a cabo nuestra interfaz será poder generar expresiones en nuestro lenguaje definido en XML a partir de un esquema ya implementado dentro de un SMBD, lo cual complementa la tarea descrita anteriormente.

Por tanto, la posible disposición de documentos definidos en XML a través de la red que especifiquen BDs permitiría, por ejemplo, que un agente que viajara en ella y que fuera el encargado de hacer búsquedas de BDs con ciertas características, lograra su objetivo al hacer una revisión en tales documentos.

Por medio de tecnologías que permiten la introducción del lenguaje XML, la interfaz se divide en tres capas o niveles, cada una de las cuales llevará a cabo una tarea en particular:

- Especificación del *DDL (Data Definition Language)*. Se construirán las estructuras necesarias por medio del lenguaje XML para llevar a cabo la interacción o comunicación deseada con el SMBD.
- Del DDL al SQL. Tiene como objetivo tomar un documento definido en el DDL y generar su respectiva representación en SQL.
- Del SQL al DDL. Esta etapa tomará un esquema definido en SQL (dentro de un sistema manejador de bases de datos) y lo traducirá al DDL.

Los capítulos 2, 3 y 4 respectivamente serán los encargados de mostrar en detalle las acciones llevadas a cabo para lograr cada uno de los tres niveles anteriores.

La incorporación de XML de acuerdo al proceso anterior, como ya se mencionó, permitirá la obtención de un documento en este lenguaje que represente al esquema de las bases de datos contempladas dentro de un SMBD, lo cual será de utilidad para poder dar soporte a una aplicación del tipo cliente-servidor, que también forma parte de este trabajo.

La aplicación desarrollada tiene como objetivo proporcionar a un usuario información, de manera remota, acerca de las bases de datos y sus elementos, de tal manera que podamos saber cómo está organizada la información contenida en cada una de ellas. El desarrollo y la descripción de dicha aplicación serán tratados dentro del capítulo 5.

Por último, mencionaremos que el conjunto de programas que dan soporte a todo lo desarrollado se realizó en el lenguaje de programación *Java*, no sólo por sus características bien conocidas (como portabilidad o flexibilidad para llevar a cabo interfaces gráficas), sino

también para poder incorporar algunas otras herramientas de las cuales se decidió hacer uso (que serán descritas en los siguientes capítulos) y que se basan en el API proporcionado por dicho lenguaje.

Capítulo 1

Conocimientos Generales

Antes de iniciar con la parte propia del trabajo en cuestión, es pertinente hacer una revisión de los temas que le darán soporte, tanto la parte referente a los modelos *Entidad-Relación* y *Relacional* en bases de datos, los cuales nos marcan el punto de inicio, así como la parte de la tecnología *XML* usada para su desarrollo.

1.1. Modelo Entidad-Relación

En el campo de estudio de las bases de datos una de las fases más importantes para su manejo es la fase de diseño, en la cual es muy útil hacer un modelo de la misma usando una representación semántica que la caracterice.

Una de las representaciones más usadas para la estructura (esquema) de una base de datos es el modelo *Entidad-Relación* (*E/R*), el cual trata de expresar el significado a través de objetos o elementos llamados *entidades*, los cuales a su vez son entrelazados por medio de *relaciones*. Entonces, esta representación tiene como fin hacer una correspondencia directa entre algo que se quiere modelar del mundo real y un esquema conceptual.

Dado que el objetivo de este capítulo no es el de hacer un manual del modelo *Entidad-Relación*, sólo se expondrán conceptos que tienen que ver con éste de manera rápida y concisa, entre los cuales encontramos:

- **Entidad.** Es la abstracción de una cosa u objeto del mundo real que se distingue de otros para un fin específico dentro de la base de datos (BD). Así, un cliente en una zapatería, puede ser definido como una entidad en el modelo *Entidad-Relación*.
- **Atributo.** Es una propiedad que caracteriza a una entidad dentro del modelo; sus valores son atómicos, que en el ejemplo de la entidad cliente serían su nombre, dirección, entre otros.
    - Atributos simples y compuestos. Los atributos simples son los que no se dividen en subpartes y los compuestos, en cambio, sí se subdividen. Por ejemplo, el

nombre de un cliente en una zapatería puede dividirse en apellido paterno, apellido materno y nombre.

- Atributos multivaluados y univaluados. Los atributos multivaluados son aquellos que pueden tomar su valor de un conjunto de cardinalidad mayor a 1 y los univaluados sólo lo toman de un conjunto de cardinalidad 1. Para el caso del cliente, un atributo multivaluado podría ser el teléfono ya que éste podría contar con más de una línea.
- Atributos nulos. Un atributo se determina nulo cuando dentro de la BD el valor para éste es desconocido o simplemente no existe. Por ejemplo, retomando el valor del atributo *teléfono*, un cliente podrá o no tener una línea disponible.

- **Relación.** Una relación es simplemente una asociación entre dos o más entidades. Así, además de un *cliente* podemos tener una entidad *vendedor*, y ambas están relacionadas a través de una *compra*.
- **Llave primaria.** Es un atributo o conjunto mínimo de atributos que caracterizan de manera única a una entidad dentro del diseño. La *clave* del vendedor puede ser un ejemplo.
- **Llave foránea.** Es un conjunto de atributos que dentro de una relación identifican a las entidades que se están relacionando. Este conjunto es precisamente el que forman las llaves primarias de dichas entidades. Un ejemplo para la relación *vender a* son las llaves primarias de las entidades *vendedor* y *cliente*.

1.2. Modelo Relacional

A diferencia del modelo anterior que es una manera sencilla de describir la estructura de los datos, hoy en día las implementaciones de las bases de datos están basadas casi siempre en otro modelo, el modelo *Relacional*.

Este modelo nos permite una sola forma de representar la información: una relación, la cual se puede describir como una tabla (renglones y columnas) con un nombre, y cuyo objetivo será mantener información sobre un conjunto de entidades. Los renglones en la tabla representan una entidad de la relación y las columnas, por su parte, corresponden a cada uno de los atributos del conjunto de entidades. Por ejemplo, la Tabla 1.1 puede ser una relación.

Nombre	*Dirección*	*Clave*
Virginia Teodosio	Abedules No.36	101
Gildardo Bautista	Av. Bombas No.56	102

Tabla 1.1: La relación *Cliente*

Ahora, los siguientes son algunos de los conceptos que el modelo *Relacional* contempla para su descripción:

- **Atributos.** Los atributos de una relación sirven como nombres para las columnas con las que esta cuenta y usualmente describen el significado de las entradas en la columna correspondiente.

- **Esquemas.** El esquema de una relación es simplemente el nombre junto con el conjunto de atributos de ésta. Así, para nuestro ejemplo anterior, el esquema es el siguiente:

  Cliente (Nombre, Dirección, Clave)

  En el modelo relacional, un diseño consiste de uno o más esquemas. El conjunto de esquemas para las relaciones en un diseño se conoce como *esquema de base de datos relacional* o simplemente *esquema de base de datos*.

- **Tuplas.** Son los renglones de una relación (claro, a excepción del que contiene los nombres de los atributos). Entonces, una tupla tiene un componente para cada atributo de la relación. Por ejemplo, los componentes de una de las tuplas para nuestro ejemplo son (*Virginia Teodosio, Abedules No.36, 101*) para los atributos *Nombre, Dirección* y *Clave* respectivamente.

1.3. De diagramas E/R a diseños relacionales

Ya que la mayoría de los sistemas de base de datos comerciales usan el modelo relacional, podríamos suponer que en la fase de diseño se debería usar también el mismo modelo, a diferencia del modelo E/R o algún otro.

Sin embargo, en la práctica es más fácil comenzar con un modelo como lo es el E/R, hacer nuestro diseño y después convertirlo al modelo relacional. La principal razón para hacerlo de esta manera es que el modelo relacional, teniendo una sola forma conceptual que es la relación, tiene algunos aspectos inflexibles que logran manejarse mejor después de haber seleccionado un diseño.

Básicamente, la forma para pasar de un diseño E/R a uno relacional es la siguiente :

- Convertir cada entidad en una relación, con el mismo conjunto de atributos; y

- remplazar una relación (del modelo E/R) por una relación (tabla) cuyos atributos son las llaves primarias de las entidades en cuestión.

Para tener una idea más amplia sobre este proceso, puede consultar [9].

1.4. XML

XML (Extensible Markup Language) ha significado un gran avance en cuanto a la tecnología de web se refiere, ya que ofrece una mayor flexibilidad y simplicidad respecto a la definición de etiquetas de una aplicación así como la auto descripción de las mismas.

XML nació oficialmente en 1998, siendo una ramificación de la tecnología SGML (Standard Generalized Markup Language), persiguiendo objetivos que en esta no se tenían como son:

- Ser un estándar en el intercambio de información a través de la Web.
- Hacer uso de etiquetas específicas de una aplicación, las cuales se definen por medio de documentos (este concepto se tratará en las siguientes secciones).
- Los documentos mencionados en el punto anterior incorporan la característica de auto-descripción de dichas etiquetas.
- Las aplicaciones desarrolladas en este lenguaje pueden "descubrir" el formato de la información y actuar en consecuencia.

La estructura de un documento XML se rige por el uso de elementos, los cuales tienen las siguientes características:

- Constan de etiquetas que aparecen por parejas, una que abre y otra que cierra. Aunque hay una excepción, que el elemento esté definido como vacío, por lo que es suficiente tener una sola etiqueta que finalice con una "/" (como $< fin/ >$).
- Los valores de sus atributos deberán estar colocados entre comillas.
- Existe distinción entre mayúsculas y minúsculas.

Dichas etiquetas delimitan datos para una cierta aplicación, los cuales estarán disponibles para otras tareas. Además, su estructura se define por medio de esquemas (*schemas*), como son los DTDs o XML Schema.

1.4.1. DTDs

Las definiciones de tipos de datos (DTDs) son un conjunto de reglas que definen la estructura de un documento XML y su importancia va más allá de estructurar un documento. Tales definiciones pueden ser compartidas vía red para definir cómo un conjunto de documentos XML debe ser estructurado.

La DTD puede estar incorporada junto con el documento XML o ser un archivo independiente, pero referido a cada documento que vaya de acuerdo a lo estipulado en él (lo cuál es más útil cuando hay varios documentos que deben apegársele).

Por ejemplo, la DTD en el Listado 1.1 nos define un lenguaje para poder especificar por medio de etiquetas la definición de un autómata finito.

Listado 1.1: DTD que define a un autómata finito determinístico. El autómata (finito) está definido por un conjunto de estados, un alfabeto de entrada y una función de transición (línea 2), donde la cardinalidad del conjunto de estados es por lo menos de uno (4) y cada estado tiene la característica de ser inicial, final o normal (5 a 8), tal y como se maneja en un autómata. Con respecto al alfabeto, éste será una cadena de entrada, la cual está constituida por símbolos separados por comas (9); y por último la función de transición define los estados de origen y de destino (10 a 15).

```
1  <?xml encoding="UTF-8" ?>
2  <!ELEMENT Automata (Q, alfabeto, delta)>
3  <!ATTLIST Automata nombre CDATA #REQUIRED>
4  <!ELEMENT Q (estado+)>
5  <!ELEMENT estado (caracteristica)*>
6  <!ATTLIST estado nombre CDATA #REQUIRED>
7  <!ELEMENT caracteristica (#PCDATA)>
8  <!ATTLIST caracteristica tipo (inicial | final | normal) #IMPLIED>
9  <!ELEMENT alfabeto (#PCDATA)>
10 <!ELEMENT delta (transicion)+>
11 <!ELEMENT transicion (de,a)>
12 <!ELEMENT de (#PCDATA)>
13 <!ATTLIST de estado CDATA #REQUIRED simbolo CDATA #REQUIRED>
14 <!ELEMENT a (#PCDATA)>
15 <!ATTLIST a estado CDATA #REQUIRED>
```

Dada la DTD del Listado 1.1, un documento que la ejemplifica es como el que se muestra en el Listado 1.2.

Listado 1.2: Ejemplar del DTD para un autómata que checa paridad.

```
1  <!DOCTYPE Automata SYSTEM "Automata.dtd">
2  <Automata nombre="sucesiones_con_un_numero_par_de_unos">
3    <Q>
4      <estado nombre="PAR">
5        <caracteristica tipo="inicial"/>
6        <caracteristica tipo="final"/>
7      </estado>
8      <estado nombre="NON"/>
9    </Q>
10   <alfabeto> 0,1 </alfabeto>
11   <delta>
12     <transicion>
```

```
13          <de estado="NON" simbolo="0" />
14          <a estado="NON" />
15        </transicion>
16        <transicion>
17          <de estado="NON" simbolo="1" />
18          <a estado="PAR" />
19        </transicion>
20        <transicion>
21          <de estado="PAR" simbolo="0" />
22          <a estado="PAR" />
23        </transicion>
24        <transicion>
25          <de estado="PAR" simbolo="1" />
26          <a estado="NON" />
27        </transicion>
28      </delta>
29    </Automata>
```

No obstante, dentro de una DTD no existe la posibilidad de especificar que los valores de los elementos o atributos se restrinjan a un conjunto en particular, es decir, no podemos definir que sea de un tipo en especial. Por ejemplo, sugongamos que los nombres de los estados fueran solamente cadenas (como NON o PAR) y no números. En un documento que respete la DTD definida anteriormente, aceptaría que el nombre del estado fuera el entero *123*.

La solución a este problema fue proporcionada por *XML Schema*.

1.4.2. XML Schema

XML Schema es un lenguaje de marcas o etiquetas tipificado cuya especificación asume que por lo menos se usan dos documentos de XML:

- Documento esquema. Describe la estructura y tipos del ejemplar del documento.
- Ejemplar del documento. Contiene la información que se va a manejar (en cierto momento).

La diferencia entre estos dos documentos radica principalmente en la idea de clase y objeto respectivamente.

XML Schema, al igual que las DTDs, puede ser usado para definir el esquema de una clase particular de documentos. Sin embargo, XML Schema usa la misma sintaxis que XML, lo cual es una gran ventaja en cuanto a que no se tiene que aprender un lenguaje nuevo para definir la gramática de los documentos XML.

XML Schema ofrece además otras ventajas sobre las DTDs como son el soporte para tipo de datos, modelo de contenido y el modelo de objetos de documentos XML.

Para ejemplificar las características de XML Schema, proponemos el siguiente documento esquema que define a un autómata finito:

Listado 1.3: Esquema que define a un autómata finito determinístico.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >

<xs:element name="Automata" type="AutomataType"/>

<xs:complexType name="AutomataType">
  <xs:sequence>
    <xs:element name="Q" type="QType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="alfabeto" type="xs:string" />
    <xs:element name="delta" type="deltaType" />
  </xs:sequence>
  <xs:attribute name="nombre" type="xs:NCName" use ="required" />
</xs:complexType>

<xs:complexType name="QType">
  <xs:sequence>
    <xs:element name="estado" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="caracteristica" minOccurs="0"
                      maxOccurs="2">
            <xs:complexType>
              <xs:attribute name="tipo" type="tipoType"
                            use="optional" default="normal"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="nombre" use="required">
          <xs:simpleType name="nombreType">
            <xs:restriction base="xs:string">
              <xs:pattern value="[A-Z]+" />
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="tipoType">
```

```
41    <xs:restriction base="xs:string">
42      <xs:enumeration value="inicial"/>
43      <xs:enumeration value="final"/>
44      <xs:enumeration value="normal"/>
45    </xs:restriction>
46  </xs:simpleType>
47
48  <xs:complexType name="deltaType">
49    <xs:sequence>
50      <xs:element name="transicion" type="transicionType"
51                  maxOccurs="unbounded"/>
52    </xs:sequence>
53  </xs:complexType>
54
55  <xs:complexType name="transicionType">
56    <xs:sequence>
57      <xs:element name="de">
58        <xs:complexType>
59          <xs:attribute name="estado" type="xs:string"
60                        use="required" />
61          <xs:attribute name="simbolo" type="xs:string"
62                        use="required" />
63        </xs:complexType>
64      </xs:element>
65      <xs:element name="a">
66        <xs:complexType>
67          <xs:attribute name="estado" type="xs:string"
68                        use="required" />
69        </xs:complexType>
70      </xs:element>
71    </xs:sequence>
72  </xs:complexType>
73
74  </xs:schema>
```

Cualquier documento que defina a un esquema tiene un encabezado particular, en el cual se indica el conjunto de carácteres (*encoding*) a utilizarse por el documento y el espacio de nombres a utilizar (líneas 1 y 2); dicho espacio se define como un conjunto de elementos relacionados y atributos identificados por un nombre común (una URL) y para definirlo se usa el atributo *xmnls* (parte también de la segunda línea del código del esquema anterior).

Ahora bien, hay que hacer notar que el esquema divide a sus elementos en:

- **Simples (***simpleType)*, los cuales son un conjunto de tipos que se representan básicamente como cadenas sin elementos o atributos; entre ellos destacan algunos tipos comunes a lenguajes de programación y otros relativos a XML. Algunos de estos tipos son: *string, double, decimal, integer, boolean, time, date, byte*, etcétera.

- **Complejos (***complexType)*, aquellos que su definición contiene típicamente un conjunto de declaraciones y referencias de elementos y declaraciones de atributos. Los elementos se declaran usando el elemento *element* y los atributos con el elemento *attribute*.

Volviendo al ejemplo, podemos ver que el esquema consiste de un elemento principal, *automata* en la línea 4, el cual está definido como un tipo complejo que está conformado por varios elementos, *Q, alfabeto* y *delta*; los cuales a su vez contienen otros elementos. Sin embargo, podemos ver que el elemento *alfabeto* en la línea 9 ya no sigue este patrón, es decir este elemento es de tipo simple.

Al definir los elementos que conforman al elemento *automata*, observamos que estos contienen atributos que indican el nombre (*name*), tipo (*type*) y número de presencias (*minOccurs* y *maxOccurs*), que tienen los valores por omisión 1; es decir, en la línea 8 podrían no aparecer. Para terminar de describir el elemento *automata* basta mencionar a sus atributos, los cuales están definidos por *attribute*, que contiene los atributos nombre (*name*), tipo (*type*) y si es requerido o no (*use*) y en caso que no sea requerido se tiene la posibilidad de tener un valor por omisión (*default*) - ver línea 24-.

Avanzando sobre el esquema observamos que en la línea 40 se tiene un tipo simple, el cual cuenta con un conjunto de restricciones que determinan la faceta que tomará dicho tipo de datos en una declaración en particular. Por ejemplo, el tipo de dato *string* cuenta con restricciones como *lenght, minlenght, maxlenght, enumeration* y *pattern*. Es aquí donde podemos apreciar una de las diferencias con las DTDs, ya que deseábamos que los nombres de los estados estén formados únicamente por caracteres alfabéticos (línea 31), en este caso sólo mayúsculas.

Hemos hablado acerca de la definición de elementos, sin mencionar como éstos se agrupan. Para proseguir nuestra descripción tocaremos el tema de agrupamiento, el cual puede ser identificado por alguna de las siguientes palabras reservadas dentro del esquema:

- **all**. Agrupa los elementos de manera que todos aparezcan una y sólo una vez, sin importar el orden.
- **choice**. Los elementos agrupados de esta manera podrían aparecer una o varias veces sin importar el orden.
- **sequence**. Se tiene que respetar un orden de definición, es decir, como fueron definidos deben de ser usados.

En la definición de autómata finito del Listado 1.3, en las lineas 7, 16, 49 y 56 se hace uso del agrupamiento *sequence*.

Por último, observemos un ejemplar del documento esquema de la definición de autómata finito en el Listado 1.4.

Listado 1.4: Ejemplar del esquema para un autómata finito que checa paridad.

```
<?xml version='1.0' encoding='UTF-8'?>
<Automata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://caoba.matem.unam.mx_Automata.xsd"
          name="sucesiones_con_un_numero_par_de_unos">
  <Q>
    <estado nombre="PAR">
      <caracteristica tipo="inicial"/>
      <caracteristica tipo="final"/>
    </estado>
    <estado nombre="NON"/>
  </Q>

  <alfabeto>
     0,1
  </alfabeto>
  <delta>
    <transicion>
      <de estado="NON" simbolo="0"/>
      <a estado="NON"/>
    </transicion>
    <transicion>
      <de estado="NON" simbolo="1"/>
      <a estado="PAR"/>
    </transicion>
    <transicion>
      <de estado="PAR" simbolo="0"/>
      <a estado="PAR"/>
    </transicion>
    <transicion>
      <de estado="PAR" simbolo="1"/>
      <a estado="NON"/>
    </transicion>
  </delta>
</Automata>
```

Con lo anterior hemos mostrado las características principales de XML y XML Schema que nos permitirán dar paso a la definición de un nuevo lenguaje en el siguiente capítulo que servirá de base para el desarrollo del resto de este trabajo.

Capítulo 2

Lenguaje de Definición de Datos (DDL)

2.1. Motivación

Actualmente el uso de un sistema manejador de base de datos (SMBD) involucra más que simplemente tener el conocimiento sobre temas teóricos como lo es el modelo relacional. Además, es necesario tener en cuenta que el uso de dicho sistema se rige por la especificación de un lenguaje, que a su vez permite la definición y manipulación de datos: *SQL (Structured Query Language)*.

Algunas de las características de este lenguaje son:

- Está basado en el modelo relacional.
- Es un lenguaje estandarizado, no obstante existen extensiones y variantes.
- Su constante uso ha provocado su desarrollo y es fuente de investigación.

Sin embargo, hay algunos aspectos sobre él que se han convertido en una desventaja, como por ejemplo:

- El resultado de algunas consultas no es estrictamente relacional, ya que se puede tener tuplas duplicadas, por ejemplo.
- El lenguaje cada vez se vuelve más grande en cuanto a sintaxis y funcionalidad, lo cual hace cada vez más pronunciada su curva de aprendizaje (en SQL2003 hay del orden de 300 palabras reservadas).
- Algunas características se instrumentan de distintas maneras en distintas versiones, lo cual impide su portabilidad.

Es por esto que se ha optado por definir otro lenguaje que nos permita hacer la definición de los datos de una manera más simple y que a la vez tenga la misma funcionalidad con respecto a SQL. Para esto, se decidió hacer uso de la tecnología XML debido a que un documento XML es una base de datos, en el sentido estricto de la palabra, es decir, es ya una colección de datos.

Además, como formato de base de datos, XML tiene algunas ventajas. Por ejemplo :

- Sus elementos describen la estructura del documento de manera intuitiva, es decir, al estar hablando del modelo relacional, si vemos un elemento cuyo nombre es *table*, se puede intuir qué es lo que representará.
- Es portátil.
- Los elementos definidos pueden ajustarse a los requerimientos del formato.
- En un esquema se puede incluir elementos definidos en otro esquema con el objeto de obtener una extensión para el primero.
- Cuenta con su propia forma de comentar sus esquemas, la cual sigue el mismo estilo al hacer uso de etiquetas.
- Los datos pueden ser representados en estructuras como árboles o gráficas.

Una vez establecida las herramientas a usar, podemos pasar a definir nuestro DDL.

2.2. Descripción del DDL

La interfaz que servirá de intermediaria entre el usuario y el sistema manejador de la base de datos necesita tener una base o estructura que permita el almacenamiento de los datos que serán manejados durante su uso.

Dado que la parte interna con la que trabajará la interfaz es el SMBD, es conveniente que se mapee la estructura que ocupa el SMBD para el almacenamiento de la información al lenguaje que ocuparemos para esta fase, XML Schema.

Para lograr esto, se definirá un lenguaje de objetos que además de abstraer los elementos involucrados en la estructura que el SMBD requiere para dicho almacenamiento, permitirá una independencia en cuanto a la sintaxis que el manejador utilice, ya que sólo se tendrán que considerar los componentes del modelo relacional y no como estos se definen para su reconocimiento por el SMBD a través de su propia sintaxis.

Dentro de dicha estructura, una de las partes que es importante mapear es la que tiene que ver con los tipos de datos que maneja el SMBD para caracterizar la información, tales como *carácter, entero*, etcétera. XML Schema cuenta con una amplia gama de estos tipos de datos llamados tipos base o primitivos y es por esto que no será necesario redefinirlos

para poder incorporarlos a nuestra interfaz[1].

Otra parte esencial en la estructura de la definición de nuestro lenguaje base será el mapeo de relaciones a la representación usada en un SMBD (tablas). Para lograr implementar el mapeo anterior, ocuparemos la definición de uno de los tipos que permite XML Schema, los tipos de datos complejos (*complexType*), debido que a través de estos se pueden definir estructuras que nos permitan tener una relación directa (en cuanto a forma) con las tablas usadas por el manejador de la base de datos.

El uso de dichos tipos de datos permite la inclusión de elementos, conformados de distintos tipos cada uno, dentro de su definición. Esto nos facilita la implementación de los atributos que contiene una tabla en nuestra estructura por medio de cláusulas de elementos (identificadas por la palabra *element*).

2.3. El modelo Relacional

El objetivo de definir el DDL es lograr que a través de este se pueda definir un documento en XML que permita la representación del modelo relacional y algunas operaciones que un SMBD pueda llevar a cabo sobre el modelo. Generalmente, todo esto se agrupa dentro de esquemas para una base de datos particular.

Dicha definición para el DDL está dada por el esquema del Listado 2.1:

Listado 2.1: Elementos de un esquema en una BD. Al tener un esquema relacional debe ser posible que todo lo que conforme a este y las operaciones con las cuales se interaccione se agrupen dentro de un mismo documento de XML, para lo cual debemos permitir la inclusión de varios elementos como son las tablas (definido a través de un elemento de nombre *table*, por ejemplo), funciones, procedimientos, operaciones ejecutadas automáticamente (*triggers*), vistas y optimización del funcionamiento de la base (índices).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3        xmlns="http://caoba.matem.unam.mx"
4        targetNamespace="http://caoba.matem.unam.mx"
5        elementFormDefault="qualified">
6
7  <xs:element name="database" type="Database"/>
8  <xs:element name="comment" type="nonEmptyString"/>
9
10 <xs:simpleType name="nonEmptyString">
11   <xs:restriction base="xs:string">
```

[1] http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/datatypes.html#built-in-primitive-datatypes

```
12      <xs:minLength value="1" />
13    </xs:restriction>
14  </xs:simpleType>
15
16  <xs:complexType name="Database">
17    <xs:annotation>
18      <xs:documentation xml:lang="en">
19          A database should be formed by schemas.
20      </xs:documentation>
21    </xs:annotation>
22    <xs:sequence>
23      <xs:element ref="comment" minOccurs="0" />
24      <xs:element name="schema" type="Schema"
25                  maxOccurs="unbounded"/>
26    </xs:sequence>
27    <xs:attribute name="name" type="nonEmptyString"
28                  use ="required" />
29  </xs:complexType>
30
31  <xs:complexType name="Schema">
32    <xs:annotation>
33      <xs:documentation xml:lang="en">
34          An schema could be formed by one or more tables,
35          procedures, etc.
36      </xs:documentation>
37    </xs:annotation>
38    <xs:sequence>
39      <xs:element ref="comment" minOccurs="0" />
40      <xs:choice maxOccurs="unbounded">
41        <xs:element name="table" type="Table" minOccurs="0" />
42        <xs:element name="procedure" type="Procedure" minOccurs="0" />
43        <xs:element name="function" type="Function" minOccurs="0" />
44        <xs:element name="trigger" type="Trigger" minOccurs="0" />
45        <xs:element name="view" type="View" minOccurs="0" />
46        <xs:element name="index" type="Index" minOccurs="0" />
47      </xs:choice>
48    </xs:sequence>
49    <xs:attribute name="name" type="nonEmptyString"
50                  use ="required" />
51  </xs:complexType>
```

La inclusión de tales elementos se logra a través de la definición de un tipo complejo que está conformado por la declaración de estos por medio de componentes *element* de XML Schema. La restricción en cuanto al número de ejemplares que habrá dentro del documento para un determinado elemento se establecerá en la definición de estos en el esquema por medio de los atributos *minOccurs* y *maxOccurs*.

2.3.1. Tablas

Dado el punto de partida, comenzaremos por definir la estructura de una tabla (en el Listado 2.2), la cual se declaró anteriormente con el elemento *table* de tipo *Table* (línea 41 en el Listado 2.1).

Listado 2.2: Definición del elemento *table*. La definición del tipo complejo *Table* consta de los elementos básicos de una tabla en un esquema relacional, tales como nombre (definido a través de un atributo para este elemento) y atributos, así como restricciones sobre la tabla (que se relacionan más con su implementación en un SMBD) las cuales tienen características establecidas que fuerzan a definirlos individualmente con un tipo complejo.

```
1  <xs:complexType name="Table">
2    <xs:sequence>
3      <xs:element ref="comment" minOccurs="0" />
4      <xs:element name="attribute" type="Attribute"
5                  maxOccurs="unbounded" />
6      <xs:element name="uniqueGroup" type="UniqueGroup" minOccurs="0"
7                  maxOccurs="unbounded" />
8      <xs:element name="primaryKeyGroup" type="PrimaryKeyGroup"
9                  minOccurs="0" />
10     <xs:element name="foreignKeyGroup" type="ForeignKeyGroup"
11                 minOccurs="0" maxOccurs="unbounded" />
12     <xs:element name="check" type="Check" minOccurs="0"
13                 maxOccurs="unbounded" />
14   </xs:sequence>
15   <xs:attribute name="name" type="nonEmptyString" use="required" />
16 </xs:complexType>
```

La base dentro de cualquier tabla son los atributos que la definen, los cuales darán paso a la formación de tuplas en una BD. Para el DDL, este elemento está definido por el tipo *Attribute* como se ve en el Listado 2.3.

Listado 2.3: Definición del elemento *attribute*. A un atributo se le identifica por sus características para efecto del modelo relacional, tales como un nombre y un tipo. Podría ser que esté establecido un valor por omisión e incluso que se tengan restricciones sobre este.

```
1  <xs:complexType name="Attribute">
2    <xs:sequence>
3      <xs:element ref="comment" minOccurs="0" />
4      <xs:element name="defaultValue" type="nonEmptyString"
5                  minOccurs="0" />
6      <xs:element name="unique" type="Unique" minOccurs="0" />
7      <xs:element name="primaryKey" type="PrimaryKey" minOccurs="0" />
```

```
8       <xs:element name="nillable" type="Nillable" minOccurs="0" />
9       <xs:element name="foreignKey" type="ForeignKey" minOccurs="0"
10                  maxOccurs="unbounded" />
11      <xs:element name="check" type="Check" minOccurs="0"
12                  maxOccurs="unbounded" />
13    </xs:sequence>
14    <xs:attribute name="name" type="nonEmptyString" use="required" />
15    <xs:attribute name="type" type="nonEmptyString" use="required" />
16  </xs:complexType>
```

Podemos agrupar las restricciones para un atributo por los tipos complejos especificados en el Listado 2.4.

Listado 2.4: Definición de las distintas restricciones para un atributo. Un atributo puede ser identificado de manera única, como llave primaria o foránea e incluso puede ocurrir que el valor asignado sea nulo o que cumpla con una condición en particular. Dicha condición podrá ser expresada en uno de distintos dialectos, el cual se indicará a través del atributo *language* (linea 43). También se debe considerar la posibilidad de retrasar la revisión de la restricción al final de la transacción o de hacerlo de forma inmediata (por medio de *Initially*).

```
1  <xs:complexType name="Unique">
2    <xs:sequence>
3      <xs:element ref="comment" minOccurs="0" />
4    </xs:sequence>
5    <xs:attribute name="name" type="nonEmptyString" use="optional" />
6    <xs:attribute name="deferrable" type="xs:boolean"
7                  default="false" />
8    <xs:attribute name="initially" type="Initially"
9                  default="immediate" />
10 </xs:complexType>
11
12
13 <xs:complexType name="PrimaryKey">
14   <xs:sequence>
15     <xs:element ref="comment" minOccurs="0" />
16   </xs:sequence>
17   <xs:attribute name="name" type="nonEmptyString" use="optional" />
18   <xs:attribute name="deferrable" type="xs:boolean"
19                 default="false" />
20   <xs:attribute name="initially" type="Initially"
21                 default="immediate" />
22 </xs:complexType>
23
24
25 <xs:complexType name="Nillable">
```

```
26    <xs:sequence>
27      <xs:element ref="comment" minOccurs="0" />
28    </xs:sequence>
29    <xs:attribute name="name" type="nonEmptyString" use="optional"/>
30    <xs:attribute name="value" type="xs:boolean" default="true"/>
31    <xs:attribute name="deferrable" type="xs:boolean"
32                  default="false"/>
33    <xs:attribute name="initially" type="Initially"
34                  default="immediate"/>
35  </xs:complexType>
36
37  <xs:complexType name="Check">
38    <xs:sequence>
39      <xs:element ref="comment" minOccurs="0" />
40      <xs:element name="definition" type="nonEmptyString" />
41    </xs:sequence>
42    <xs:attribute name="name" type="nonEmptyString" use="optional"/>
43    <xs:attribute name="language" type="nonEmptyString"
44                  use="required"/>
45    <xs:attribute name="deferrable" type="xs:boolean"
46                  default="false"/>
47    <xs:attribute name="initially" type="Initially"
48                  default="immediate"/>
49  </xs:complexType>
```

No obstante, la restriccion más importante que hay que tomar en cuenta para establecer los atributos es la definición de la llave foránea, que agrega nuevas posibilidades dentro de un esquema relacional y hay que considerarlas en el DDL. Al incorporar una llave foránea es posible especificar el atributo al que se hará referencia a través de la llave por medio del atributo *referencesAttribute*, como se aprecia en el Listado 2.5.

Listado 2.5: Definición del tipo *ForeignKey*. También hay que considerar las acciones que se efectuarán en caso de darse un cambio dentro de una de las tablas de la que depende otra u otras (definidas a través de *Action*, cuyo formato se muestra en el Listado 2.6): qué hacer al momento de actualizar un registro dentro de una de las tablas o qué hacer al momento de borrarlo (*onUpdate* y *OnDelete* respectivamente).

```
1  <xs:complexType name="ForeignKey">
2    <xs:sequence>
3      <xs:element ref="comment" minOccurs="0" />
4    </xs:sequence>
5    <xs:attribute name="name" type="nonEmptyString" use="optional"/>
6    <xs:attribute name="referencesTable" type="nonEmptyString"
7                  use="required"/>
8    <xs:attribute name="referencesAttribute" type="nonEmptyString"
9                  use="optional"/>
```

```
10    <xs:attribute name="onDelete" type="Action" default="restrict"/>
11    <xs:attribute name="onUpdate" type="Action" default="restrict"/>
12    <xs:attribute name="deferrable" type="xs:boolean"
13                  default="false"/>
14    <xs:attribute name="initially" type="Initially"
15                  default="immediate"/>
16  </xs:complexType>
```

Listado 2.6: Valores posibles que proporciona el tipo *Action*. Para la ejecución de acciones al momento de actualizar y borrar tuplas, uno puede optar por restringir la acción en caso de que existan tuplas que sean referidas (*restrict*), o bien que la acción se ejecute recursivamente sobre los atributos de las tablas dependientes (*cascade*), que se establezca un valor nulo para dichos atributos (*setNull*) o que simplemente el valor del atributo referente tome su valor por omisión (*setDefault*).

```
1  <xs:simpleType name="Action">
2    <xs:restriction base="nonEmptyString">
3      <xs:enumeration value="restrict"/>
4      <xs:enumeration value="cascade"/>
5      <xs:enumeration value="setNull"/>
6      <xs:enumeration value="setDefault"/>
7    </xs:restriction>
8  </xs:simpleType>
```

Listado 2.7: Posibles valores para el atributo *initially*. El momento en que una restricción sobre los atributos pudiera ser revisada será indicado por *deferrable* (Listado 2.5, línea 12), ya sea que ésta se realice inmediatamente después de un comando ejecutado por el manejador, especificado por *deferrable=false*, o que pueda hacerse al final de una transacción con *deferrable=true*. Y si se activa esta modalidad, pero se desea que el tiempo por omisión en el que se lleve a cabo la verificación de cada restricción sea después de la ejecución de un comando, el valor del atributo *initially* deberá estar definido en *immediate* y no en *deferred*.

```
1  <xs:simpleType name="Initially">
2    <xs:restriction base="nonEmptyString">
3      <xs:enumeration value="deferred"/>
4      <xs:enumeration value="immediate"/>
5    </xs:restriction>
6  </xs:simpleType>
```

Con los elementos anteriores se han cubierto ya las definiciones necesarias que tienen que ver con los atributos que serán parte de una tabla en una BD; ahora sólo

resta mencionar aquellos que formarán el conjunto de restricciones sobre la tabla (ver el Listado 2.2). Las restricciones que influyen sobre una tabla son las definiciones de unicidad (Listado 2.8), llave primaria y/o foránea (listados 2.9 y 2.10, respectivamente) conformadas por más de un atributo, así como el seguimiento de una condición entre los componentes de la misma.

Listado 2.8: Definición del tipo *UniqueGroup*. El conjunto de atributos que conformarán este grupo serán definidos a través del elemento *keyElement*.

```
<xs:complexType name="UniqueGroup">
  <xs:sequence>
    <xs:element ref="comment" minOccurs="0" />
    <xs:element name="keyElement" type="KeyElement" minOccurs="2"
                maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="name" type="nonEmptyString" use="optional" />
  <xs:attribute name="deferrable" type="xs:boolean"
                default="false" />
  <xs:attribute name="initially" type="Initially"
                default="immediate" />
</xs:complexType>

<xs:complexType name="KeyElement">
  <xs:attribute name="name" type="nonEmptyString" use="required" />
</xs:complexType>
```

Listado 2.9: Definición para elemento *primaryKeyGroup*.

```
<xs:complexType name="PrimaryKeyGroup">
  <xs:sequence>
    <xs:element ref="comment" minOccurs="0" />
    <xs:element name="keyElement" type="KeyElement" minOccurs="2"
                maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="name" type="nonEmptyString" use="optional" />
  <xs:attribute name="deferrable" type="xs:boolean"
                default="false" />
  <xs:attribute name="initially" type="Initially"
                default="immediate" />
</xs:complexType>
```

Ya mencionamos que una parte importante dentro de un esquema relacional es la definición de la llave foránea para establecer una relación. Dentro del tipo complejo *Attribute* se dio la posibilidad de definir un solo atributo como llave foránea. Sin embargo, ésta puede estar definida también como un grupo de aquellos. El tipo *ForeignKeyGroup*,

definido en el Listado 2.10, involucra esta opción.

Listado 2.10: Definición del elemento *foreignKeyGroup*. A diferencia de la llave foránea conformada por un solo atributo, la referencia a atributos será por medio del elemento *references*.

```
<xs:complexType name="ForeignKeyGroup">
  <xs:sequence>
    <xs:element ref="comment" minOccurs="0" />
    <xs:element name="attribute" type="KeyElement" minOccurs="2"
                maxOccurs="unbounded" />
    <xs:element name="references" type="KeyElement" minOccurs="0"
                maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="name" type="nonEmptyString" use="optional" />
  <xs:attribute name="referencesTable" type="nonEmptyString"
                use="required" />
  <xs:attribute name="onDelete" type="Action" default="restrict" />
  <xs:attribute name="onUpdate" type="Action" default="restrict" />
  <xs:attribute name="deferrable" type="xs:boolean"
                default="false" />
  <xs:attribute name="initially" type="Initially"
                default="immediate" />
</xs:complexType>
```

Para ejemplificar los elementos recién descritos, tomemos la definición de la tabla *vendedor*, la cual pertenece a un esquema de BD de compras, definida en el Listado 2.11 y cuya representación correspondiente en nuestro DDL se muestra en el Listado 2.12.

Listado 2.11: Ejemplo de la definición de una tabla en SQL.

```
CREATE TABLE "vendedor" (
    clave char(4) PRIMARY KEY NOT NULL,
    curp text CHECK ( char_length(trim(curp)) <> 10 )
         REFERENCES persona ON DELETE CASCADE ON UPDATE CASCADE,
    fecha_nacimiento date,
    sueldo money,
    horario char(10),
    comisiones money
);
```

Listado 2.12: Ejemplo de la definición de una tabla en el DDL.

```
<table name="vendedor">
  <comment>
    Manejara a los vendedores como entes de nuestro sistema
  </comment>
  <attribute name="clave" type="char(4)">
    <comment>
      Identificara al vendedor de forma unica y consta de solo
      4 caracteres (secuencia 0001)
    </comment>
    <primaryKey/>
    <nillable value="false" />
  </attribute>
  <attribute name="curp" type="text">
    <comment>
      hereda este atributo de ''persona'', la cual es
      una generalizacion
    </comment>
    <foreignKey referencesTable="persona" onDelete="cascade"
                onUpdate="cascade"/>
    <check language="PSQL">
      <definition>
           char_length(trim(curp)) &lt;&gt; 10
      </definition>
    </check>
  </attribute>
  <attribute name="fecha_nacimiento" type="date">
    <comment> la fecha de nac. con el formato yymmdd </comment>
  </attribute>
  <attribute name="sueldo" type="money">
    <comment> sueldo base </comment>
  </attribute>
  <attribute name="horario" type="char(10)">
    <comment>
      horario que cubre el vendedor (matutino, vespertino o mixto)
    </comment>
  </attribute>
  <attribute name="comisiones" type="money">
    <comment>
      comisiones para el vendedor, 10% del total de sus ventas
    </comment>
  </attribute>
</table>
```

Ahora bien, una vez terminado con la definición del elemento *table*, seguiremos con el resto de los elementos de *Schema*.

2.4. Operaciones

Ya determinada la estructura de las tablas para el SMBD, en algunas ocasiones es útil definir ciertas operaciones que con base en dicha estructura consigan alterar la información contenida en ellas o simplemente la manipulen.

Dentro de nuestra especificación para el DDL podemos definir funciones, triggers y vistas para una manipulación más estrecha de los atributos.

2.4.1. Funciones y Procedimientos

Habrá operaciones que podrían no ser expresadas en términos de SQL, pero sí en algún lenguaje de propósito general, las cuales serán integradas al esquema por medio de la definición de funciones y/o procedimientos. La especificación del DDL las contemplará por medio de los elementos *function* de tipo *Function* y *procedure* de tipo *Procedure* (definidos en el Listado 2.1) respectivamente.

Listado 2.13: Definición del elemento *function*. La estructura de esta operación es básicamente la misma a la de una función en cualquier lenguaje de programación. La operación cuenta con un nombre, parámetros (opcionales) y un valor de regreso. Además no tiene que ser definida en un lenguaje de programación en particular, ya que cuenta con el atributo *language* para indicar qué lenguaje de programación se está usando. Por último para indicar el cuerpo de la función ocuparemos el elemento *source*.

```
1  <xs:complexType name="Function">
2    <xs:sequence>
3      <xs:element ref="comment" minOccurs="0" />
4      <xs:element name="parameter" minOccurs="0"
5                  maxOccurs="unbounded">
6        <xs:complexType>
7          <xs:attribute name="type" type="nonEmptyString" />
8        </xs:complexType>
9      </xs:element>
10     <xs:element name="returns" type="nonEmptyString"/>
11     <xs:element name="source" type="nonEmptyString"/>
12   </xs:sequence>
13   <xs:attribute name="name" type="nonEmptyString" use="required"/>
14   <xs:attribute name="language" type="nonEmptyString"
15                 use="required"/>
16   <xs:anyAttribute namespace="##other" processContents="lax" />
17 </xs:complexType>
```

Los procedimientos, como sabemos, sólo difieren de las funciones en que no tienen un valor de regreso. Su definición está dada por el Listado 2.14.

Listado 2.14: Definición del elemento *procedure*.

```
<xs:complexType name="Procedure">
  <xs:sequence>
    <xs:element ref="comment" minOccurs="0" />
    <xs:element name="parameter" minOccurs="0"
                maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="type" type="nonEmptyString" />
      </xs:complexType>
    </xs:element>
    <xs:element name="source" type="nonEmptyString" minOccurs="0" />
  </xs:sequence>
  <xs:attribute name="name" type="nonEmptyString" use="required" />
  <xs:attribute name="language" type="nonEmptyString"
                use="required" />
</xs:complexType>
```

En el Listado 2.15 se encuentra un ejemplo de una función definida en SQL, mientras que para mostrar el uso de los elementos definidos en esta sección, su contraparte especificada a través del DDL se muestra en el Listado 2.16.

Listado 2.15: Ejemplo de la definición de una función en SQL.

```
CREATE FUNCTION set_fecha_nac() RETURNS trigger
    AS '
        declare
          fnac text;
        begin
          if new.fecha_nacimiento is null and new.curp is not null
          then
            fnac := substring(new.curp, 5, 6)
            new.fecha_nacimiento := to_date(fnac, 'yymmdd');
          end if;
          return new;
        end; '
    LANGUAGE plpgsql;
```

Listado 2.16: Ejemplo de la definición de una función en el DDL.

```
<function name="set_fecha_nac" language="SQL">
  <returns> trigger </returns>
  <source>
    declare
      fnac text;
    begin
```

```
7         if new.fecha_nacimiento is null and new.curp is not null then
8           fnac := substring(new.curp, 5, 6)
9           new.fecha_nacimiento := to_date(fnac, 'yymmdd');
10        end if;
11        return new;
12      end;
13    </source>
14  </function>
```

2.4.2. Triggers

Otra de las operaciones a incluir es *trigger*, el cual es una acción que invoca a una función que el sistema ejecuta automáticamente como un efecto secundario de una modificación a la base de datos. La definición de esta operación dentro del DDL está dada por el elemento *trigger* (definido en el Listado 2.1) de tipo *Trigger*:

Listado 2.17: Definición del elemento *trigger*. La orden es ejecutada o bien antes o bien después de la modificación, y el tipo complejo *When* es el encargado de indicar el momento. También la modificación tiene que ser sobre alguna relación del esquema y en especial sobre las tuplas; es por esta razón que *Trigger* cuenta con los atributos *target* y *way* para indicar en dónde se ejecutará la acción. Ahora bien, hemos hablado del momento de la ejecución de la acción a tomar olvidándonos un poco de la operación que realizó la modificación en la BD y que generó el evento del *trigger*. Los atributos *onInsert*, *onUpdate* u *onDelete* nos permitirán especificar bajo cual(es) operación(es) de modificación se dará dicha generación del evento para ejecutarse la acción correspondiente.

```
1  <xs:complexType name="Trigger">
2    <xs:sequence>
3      <xs:element ref="comment" minOccurs="0" />
4    </xs:sequence>
5    <xs:attribute name="name" type="nonEmptyString" use="required" />
6    <xs:attribute name="when" type="When" use="required" />
7    <xs:attribute name="target" type="nonEmptyString"
8                  use="required" />
9    <xs:attribute name="way" type="Way" use="required" />
10   <xs:attribute name="handler" type="nonEmptyString"
11                 use="required" />
12   <xs:attribute name="onInsert" type="xs:boolean" use="optional"
13                 default="false" />
14   <xs:attribute name="onUpdate" type="xs:boolean" use="optional"
15                 default="false" />
16   <xs:attribute name="onDelete" type="xs:boolean" use="optional"
17                 default="false" />
18 </xs:complexType>
```

```
19
20 <xs:simpleType name="When">
21   <xs:restriction base="nonEmptyString">
22     <xs:enumeration value="before" />
23     <xs:enumeration value="after" />
24   </xs:restriction>
25 </xs:simpleType>
26
27 <xs:simpleType name="Way">
28   <xs:restriction base="nonEmptyString">
29     <xs:enumeration value="row" />
30     <xs:enumeration value="statement" />
31   </xs:restriction>
32 </xs:simpleType>
```

Para ejemplifcar este elemento, supongamos que al insertar un registro en la tabla *vendedor* (Listado 2.11), se desea establecer el campo *fecha_nacimiento* si es que este no fue definido por lo que desearíamos poder mandar a llamar a la función del Listado 2.15. Para esto, necesitamos definir un trigger como el del Listado 2.18, cuya definición a través del DDL se muestra en el Listado 2.19.

Listado 2.18: Ejemplo de la definición de un trigger en SQL.

```
1 CREATE TRIGGER set_fecha_nacimiento BEFORE INSERT ON vendedor
2     FOR EACH ROW EXECUTE PROCEDURE set_fecha_nac();
```

Listado 2.19: Ejemplo de la definición de un trigger en el DDL.

```
1 <trigger name="set_fecha_nacimiento" when="before" target="vendedor"
2          way="row" handler="set_fecha_nac()" onInsert="true" />
```

2.4.3. Vistas

Ahora, otro elemento que nos falta describir son las vistas, las cuales son tablas temporales (ya que no existen físicamente) que guardan información más específica con respecto a los datos almacenados en las tablas de la base de datos.

Dentro del esquema se incorporó a este elemento con el nombre de *view* del tipo *View* el cual está definido de la manera que se muestra en el Listado 2.20.

Listado 2.20: Definición del elemento *view*. Los datos que se incorporan en una vista se obtienen a través de una consulta (*query* en el idioma inglés) sobre tablas o vistas ya existentes. Este elemento incorporará la definición de dicha consulta a través del texto que se introduzca entre la etiqueta que abra y la que cierre al elemento *statement*. Así también, de igual manera que los *checks*, podrá especificarse un dialecto en particular en el atributo *language*.

```
<xs:complexType name="View">
  <xs:sequence>
    <xs:element ref="comment" minOccurs="0" />
    <xs:element name="statement" type="nonEMptyString" />
  </xs:sequence>
  <xs:attribute name="name" type="nonEmptyString"
                use="required"/>
  <xs:attribute name="language" type="nonEmptyString"
                use="required"/>
</xs:complexType>
```

Como podemos observar, la incorporación de la definición de la consulta para una vista no se hará a través de elementos, sino simplemente por medio de la expresión en algún dialecto del SQL. La razón para esto es que, como ya lo mencionamos, el DDL es un lenguaje para la la definición de datos, mas no para su manipulación, lo cuál es lo que se hace en una consulta.

Para ejemplificar el uso de este elemento, sigamos con nuestro ejemplo del esquema de compras y supongamos que en cierto momento deseamos conocer cuales son los vendedores que tienen un sueldo superior a los 1000 pesos, para lo cual se puede definir una vista ocupando los campos de la tabla *vendedor* (definida en el Listado 2.11). Ambas definiciones en SQL y en el DDL se encuentran en los listados 2.21 y 2.22 respectivamente.

Listado 2.21: Ejemplo de la definición de una vista en SQL.

```
CREATE VIEW sueldos AS
    SELECT clave, sueldo
    FROM vendedor
    WHERE sueldo >= 1000;
```

Listado 2.22: Ejemplo de la definición de una vista en el DDL.

```
<view name="sueldos" language="SQL">
  <statement>
    SELECT clave, sueldo
    FROM vendedor
```

```
5       WHERE sueldo >= 1000;
6     </statement>
7   </view>
```

2.4.4. Índices

Una manera para mejorar el rendimiento en cuanto a consultas en una base de datos es mediante el uso de índices[2], los cuales ayudan a optimizar y conseguir un buen rendimiento global. Dichos índices van a ser representados mediante el elemento *index* de tipo *Index* definido por el Listado 2.23.

Listado 2.23: Definición de tipo para el elemento *index*. La definición de *Index* consta de los elementos *column*, en caso de referirnos a una o varias columnas de la tabla señalada por el atributo *target*, o bien *columnExpression* para indicar la expresión o la función a cosiderar. Otro elemento que forma parte de esta definición es *where*, el cual nos va a indicar las restricciones que deben cumplise antes de usar el índice.

```
1  <xs:complexType name="Index">
2    <xs:sequence>
3      <xs:element ref="comment" minOccurs="0" />
4      <xs:choice>
5        <xs:element name="column" type="nonEmptyString"/>
6        <xs:element name="columnExpression" type="nonEmptyString"/>
7      </xs:choice>
8      <xs:element name="where" type="nonEmptyString" minOccurs="0"/>
9    </xs:sequence>
10   <xs:attribute name="name" type="nonEmptyString" use="required"/>
11   <xs:attribute name="target" type="nonEmptyString"
12                 use="required"/>
13   <xs:attribute name="unique" type="xs:boolean" use="optional"
14                 default="false"/>
15   <xs:anyAttribute/>
16 </xs:complexType>
```

Un elemento que falta definir para los índices es el referente al método que éstos utilizarán para llevar a cabo su tarea; sin embargo, cada SMBD ofrece distintas opciones: por ejemplo, PostgreSQL ofrece los métodos de *btree, hash, rtree* y *gist*. Entonces, dado que la incorporación de éste depende en gran medida del SMBD, lo que se hizo en el esquema para considerarlo es tener la posibilidad de poder definirlos a través de un atributo (línea 15 del Listado 2.23) para este elemento.

[2] Aunque no son parte del estándar, se han vuelto punto importante para los SMBDs en general.

Esta incorporación tiene como idea que la definición de los métodos para un SMBD en particular sea hecha en un archivo de XML Schema con un espacio de nombres para dicho SMBD. Así, para PostgreSQL, dicha definición estaría dada por el código en el Listado 2.24.

Listado 2.24: Definicion del atributo *method* para PostgreSQL.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns="http://caoba.matem.unam.mx/postgresql"
      targetNamespace="http://caoba.matem.unam.mx/postgresql"
      elementFormDefault="qualified">

  <xs:attribute name="method">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="btree" />
        <xs:enumeration value="hash" />
        <xs:enumeration value="rtree" />
        <xs:enumeration value="gist" />
      </xs:restriction>
    </xs:simpleType>
  </xs:atttribute>

</xs:schema>
```

Para ejemplificar lo anterior, retomemos la tabla *vendedor* (Listado 2.11) y supongamos que se desea implementar un índice sobre ella que se base en el atributo *curp* ocupando el método *btree*. La definición para este elemento tanto en SQL como en el DDL se encuentra en los listados 2.25 y 2.26 respectivamente.

Listado 2.25: Ejemplo de la definición de un índice en SQL.

```
CREATE UNIQUE INDEX curp_idx ON vendedor (curp);
```

Listado 2.26: Ejemplo de la definición de un índice en el DDL.

```
<index name="curp_idx" target="vendedor" unique="true"
       pgsql:method="btree"> [3]
  <column> curp </column>
</index>
```

[3] Supongamos que este espacio de nombres fue definido en el documento como *xmlns:pgsql="http://caoba.matem.unam.mx/postgresql*

Con esto damos por terminada la definición del lenguaje que servirá como esquema. Entonces, en los capítulos siguientes se describirá cómo es que se incorporará al resto del trabajo.

Capítulo 3

Generación de SQL a través del DDL

En el capítulo anterior se definió el DDL para representar el esquema de una BD a través de un documento en XML.

Ahora, el siguiente paso será obtener la definición de un esquema de una BD en un SMBD particular dado un documento en XML que lo especifique, es decir, deseamos obtener el conjunto de enunciados necesarios en el lenguaje que el SMBD ocupa para definir la BD en cuestión[1].

Para llevar a cabo esta tarea, lo que se hará será dividir el proceso en las siguientes etapas:

- Hacer un análisis sintáctico y semántico del documento dado en XML.
- Generar los enunciados en el lenguaje que el SMBD ocupa.

El objetivo de la primera etapa será verificar que el documento que define la BD a través del DDL esté construido sintáctica y semánticamente de manera correcta, debido a que en la segunda etapa se actuará de acuerdo al tipo de expresión que se obtenga al leer el documento.

3.1. Análisis Sintáctico y Semántico

Dado el documento en XML que describe la BD, lo que tenemos que hacer primero es leerlo, para lo cual nos auxiliaremos de una herramienta que nos permitirá hacerlo fácilmente: *XMLBeans*.[2]

[1] El SMBD que se ocupará para ejemplificar este proceso será PostgreSQL.
[2] http://xmlbeans.apache.org/

XMLBeans es una herramienta que permite acceder al poder que posee XML al estilo del lenguaje de programación *Java*.

3.1.1. Representación del documento en XML por medio de un árbol

Dado que ya tenemos generado un documento en XML, nuestro interés es que a través de XMLBeans podamos leer dicho documento para que de esta manera podamos hacer la verificación sintáctica y semántica del documento.

XMLBeans permite construir un árbol que representa al documento en XML, para lo cual hacemos básicamente lo siguiente:

```
DatabaseDocument dbDoc = DatabaseDocument.Factory.parse(dbFile);
```

Antes de construir el árbol es posible especificar qué elementos del documento en XML van a ser permitidos en base a nuestra definición del DDL, es decir, se puede establecer una relación con clases definidas en Java y que son la base de la construcción.

3.1.2. Compilación del esquema

Las clases mencionadas son el resultado de una de las funcionalidades que tiene XMLBeans, que es tener la posibilidad de compilar el esquema (el cual fue definido a través de XML Schema) para así obtener tipos por medio de Java que representen a los elementos y los tipos incorporados en dicho esquema, que en nuestro caso son los que definen al DDL [3].

Esta compilación provee una manera de ver a nuestro DDL al estilo *JavaBeans*, además de que si comparamos el contenido del archivo que define al DDL con los tipos generados, se verán naturalmente mapeados. Por ejemplo, retomemos la definición que se estableció para una tabla en nuestro DDL, la cual se puede ver en el Listado 3.1.

Listado 3.1: Definición para una tabla en el DDL.

```
1  <xs:element name="table" type="Table" minOccurs="0"
2                  maxOccurs="unbounded" />
3  ...
4  <xs:complexType name="Table">
5    <xs:sequence>
6      <xs:element ref="comment" minOccurs="0" />
7      <xs:element name="attribute" type="Attribute"
8                  maxOccurs="unbounded" />
9      <xs:element name="uniqueGroup" type="UniqueGroup"
10                 minOccurs="0" maxOccurs="unbounded" />
11     <xs:element name="primaryKeyGroup" type="PrimaryKeyGroup"
12                 minOccurs="0" />
```

[3] Esta tarea puede llevarse a cabo, por ejemplo, a través de *Ant*.

```
13      <xs:element name="foreignKeyGroup" type="ForeignKeyGroup"
14                  minOccurs="0" maxOccurs="unbounded"/>
15      <xs:element name="check" type="Check" minOccurs="0"
16                  maxOccurs="unbounded" />
17    </xs:sequence>
18    <xs:attribute name="name" type="nonEmptyString" use="required"/>
19  </xs:complexType>
```

Entonces, la compilación creará una interfaz con el nombre que se le dió al tipo complejo, es decir *Table*, además de que para cada uno de sus elementos, por ejemplo *comment* y *attribute*, incorporará métodos como los que se muestran en el Listado 3.2.

Listado 3.2: Métodos de acceso para el elemento *comment*.

```
1  public abstract java.lang.String getComment()
2  public abstract void setComment( java.lang.String )
3
4  public abstract mx.unam.matem.caoba.Attribute[] getAttributeArray()
5  public abstract void setAttributeArray(
6                      mx.unam.matem.caoba.Attribute[] attributeArray )
```

Así como también para su atributo (*name*), como se muestra en el Listado 3.3.

Listado 3.3: Métodos de acceso para el atributo *name* de *table*.

```
1  public abstract java.lang.String getName()
2  public abstract void setName( java.lang.String )
```

Además de generar interfaces como esta, para cada tipo complejo se generarán clases *Factory*, que están definidas solamente por métodos estáticos, cuyo propósito es crear instancias de dichos tipos. Precisamente es en estas clases en donde se encuentra el método *parser* utilizado en la sección 3.1.1.

3.1.3. Validación del documento

A pesar de que el método *parse* nos construye un árbol que representa al documento en XML, esto no nos garantiza que los elementos que contiene sigan las restricciones impuestas por la definición del DDL, en primer lugar, así como tampoco otras que son necesarias de acuerdo al esquema de una base de datos.

Entonces, para validar nuestro documento sintáctica y semánticamente de acuerdo al DDL haremos lo especificado en el Listado 3.4.

Listado 3.4: Validando el documento en XML de acuerdo al esquema.

```
1  validateOptions.setErrorListener(errorList);
2  isValid = dbDoc.validate(validateOptions);
3
4  if (!isValid){
5      for (int i = 0; i < errorList.size(); i++){
6          XmlError error = (XmlError)errorList.get(i);
7
8          System.out.println("\n");
9          System.out.println("Message:"+error.getMessage()+"\n");
10         System.out.println("Location of invalid XML:" +
11                     error.getCursorLocation().xmlText()+"\n");
12     }
13
14     throw new InvalidInstanceXMLException(
15                     "Not a valid instance of the schema");
16 }
```

Así, con las clases que se generaron a la hora de la compilación, el método *validate* verifica que la construcción del objeto que lo invoca siga las restricciones impuestas por XML Schema en nuestro DDL, obteniendo un valor de verdadero si se cumplieron tales restricciones y un valor de falso en caso contrario (línea 2 del Listado 3.4).

Básicamente lo que lleva a cabo este método es implementar una máquina de estados (representada por la Figura 3.1) para así verificar que la estructura que se definió en el DDL sea la misma que la definida en el documento en XML[4]. Si la máquina termina en el estado de error, se retornará el valor de falso; sin embargo, si la verificación es exitosa el elemento habrá sido validado.

No obstante, hay restricciones que no pueden ser especificadas dentro del esquema en XML en la definición del DDL y que son parte importante para el esquema de la base de datos.

Una de estas restricciones marca que el conjunto de llave primaria de una tabla no puede contener elementos definidos como nulos, por lo que para cada elemento *table* se tendrán que verificar los elementos *primaryKey, nillable* y *primaryKeyGroup*.

Otra restricción está basada en los siguientes puntos:

- La restricción de llave primaria no puede aparecer dentro de dos elementos *attribute* distintos; y
- No puede aparecer a nivel atributo y a nivel tabla simultáneamente.

[4]La representación de las máquinas de estados para los demás elementos se encuentran en el apéndice.

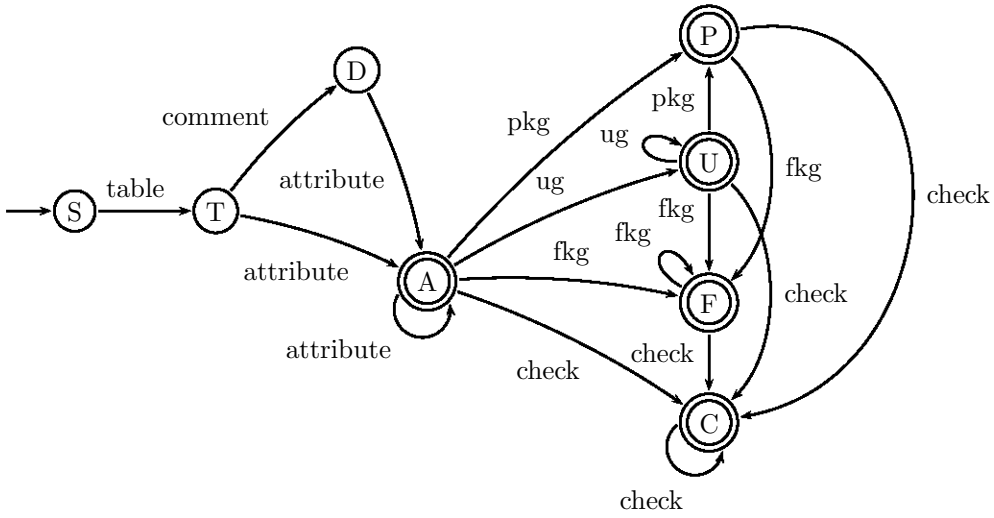

Figura 3.1: Máquina de estados para el elemento *table*.

Para verificar la parte que tiene que ver con estas restricciones tenemos el Listado 3.5.

Listado 3.5: Validando la no nulidad en el conjunto de llave primaria.

```
1  ...
2  Table t = schemas[i].getTableArray(j);
3  nillableElementNames = new ArrayList();
4
5  for(int k = 0; k < t.sizeOfAttributeArray(); k++){
6      Attribute attribute = t.getAttributeArray(k);
7      if(attribute.isSetPrimaryKey()){
8
9          if(t.isSetPrimaryKeyGroup())
10             throw new SemanticException("The table "+
11                         t.getName() + " have two primary keys");
12
13         if(RelationalSchema.isExplicitNillable(attribute))
14             throw new SemanticException("The attribute "+
15                         t.getName()+":" + attribute.getName()+
16                         " can't be primary key and nillable");
17     } else{
18         if(RelationalSchema.isExplicitNillable(attribute))
19             nillableElementNames.add(attribute.getName());
20     }
21 }
22
23 if(t.isSetPrimaryKeyGroup()){
24
```

```
25      PrimaryKeyGroup pkg = t.getPrimaryKeyGroup();
26
27      for(int k = 0; k < pkg.sizeOfKeyElementArray(); k++){
28          if(nillableElementNames.contains(
29                  pkg.getKeyElementArray(k).getName()))
30              throw new SemanticException("The attribute "+
31                                  t.getName()+
32                                  ":"+pkg.getKeyElementArray(k).getName()+
33                                  " can't be primary key and nillable");
34      }
35  }
36  ...
```

La última de las restricciones tiene que ver con el orden en que el creador del documento en XML especifique algunos de sus elementos, como las tablas, ya que si simplemente tomamos nuestro árbol y nos ponemos a generar la salida de manera secuencial, en el resultado probablemente habrá problemas de dependencias.

La solución a este problema requiere de la incorporación de otros procesos, por lo que será tratado aparte en la siguiente sección.

3.2. Dependencias

Hasta este momento, se ha descrito parte del proceso que se sigue para pasar de una base de datos especificada a través del DDL en un documento XML a una que está especificada en SQL. Sin embargo, no hemos tocado el tema de las inminentes dependencias que puede haber entre elementos que la componen.

Con dependencias queremos decir que existen algunos elementos que necesitan hacer referencia a otros dentro del mismo esquema. Por ejemplo, cuando definimos una llave foránea en una tabla, necesitamos indicar sobre qué tabla se hará la referencia. Esto nos dice que para generar dicha tabla, la tabla referida necesita haberse ya definido.

En lo que respecta a las dependencias de los elementos definidos a través de nuestro DDL en este trabajo, se considerarán únicamente para tres de ellos de la siguiente manera:

Tablas, las cuales sólo hacen referencia a otras tablas;
Triggers, los cuales dependen de tablas y funciones; e
Índices, que sólo dependen de las tablas.

Entonces, nuestro verdadero problema es que dado un documento XML, el usuario puede especificar la tabla con la llave foránea en primer lugar y posteriormente la tabla a la que hace referencia, de manera correcta (sintácticamente hablando) y dado que el proceso de verificación descrito en las secciones anteriores es meramente secuencial,

la generación de nuestra salida se llevaría a cabo de manera exitosa; sin embargo, al quererla introducir en nuestro manejador, este nos indicaría seguramente un error, debido al problema de dependencia ya descrito. Por lo tanto, es indispensable incluir un proceso que resuelva dicho problema y que la salida de dicho proceso sea un posible orden de los elementos de nuestro documento en el que todas las dependencias sean resueltas, es decir, para nuestro ejemplo, la salida defina primero a la tabla referida y posteriormente la que la refiere, aunque en el documento esto no haya ocurrido así.

3.2.1. Resolución de las dependencias

Como vimos en la sección anterior, sería común tener un orden parcial entre elementos de nuestro documento XML, esto es, que ciertos elementos tengan que definirse antes que otros y que haya elementos que no tengan relación alguna entre sí. Este tipo de relaciones es fácilmente representada por una gráfica dirigida o digráfica, en la que habrá una arista del vértice u al vértice v si es que el elemento representado por u debe de definirse antes que el representado por v.

Así por ejemplo, supongamos que dentro de un esquema de ventas se tiene un conjunto de vendedores y sucursales. Para modelarlo en una BD, necesitaríamos tener una tabla que controle la información de los vendedores, digamos V, otra para las sucursales, llamada S y otra para relacionarlas (R). Este esquema se ejemplifica en el Listado 3.6.

Listado 3.6: Elementos de un esquema de BD en la que existen dependencias.

```
<table name="vendedor">
  <attribute name="clave" type="char(4)">
    <comment> Clave unica del vendedor </comment>
    <primaryKey/>
    <nillable value="false" />
  </attribute>
   ...
</table>

<table name="sucursal">
  <attribute name="sat" type="char(10)">
    <comment> Clave de registro ante el SAT </comment>
    <primaryKey/>
    <nillable value="false" />
  </attribute>
   ...
</table>

<table name="es_vendedor_de">
  <attribute name="clave" type="char(4)">
    <nillable value="false" />
```

```
22      <foreignKey referencesTable="vendedor" onDelete="cascade"
23                  onUpdate="cascade"/>
24    </attribute>
25    <attribute name="sat" type="char(10)">
26      <nillable value="false" />
27      <foreignKey referencesTable="sucursal" onDelete="cascade"
28                  onUpdate="cascade"/>
29    </attribute>
30    <primaryKeyGroup>
31      <keyElement name="clave"/>
32      <keyElement name="sat"/>
33    </primaryKeyGroup>
34  </table>
```

Como podemos apreciar, la tabla *es_vendedor_de* hace referencia a las tablas *vendedor* y *sucursal* (lineas 22 y 27, Listado 3.6); y la digráfica que representaría la dependencia que hay de la tabla R hacia las otras tablas sería como se muestra en la Figura 3.2.

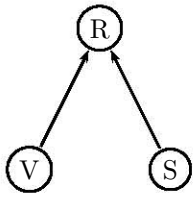

Figura 3.2: Representación de dependencia entre elementos mediante una digráfica.

Entonces, dado nuestro documento en XML con los elementos que conformarán nuestra base de datos, lo que haremos será formar una gráfica que represente la dependencia entre todos sus elementos, de tal manera que al aplicar un algoritmo a ésta podamos resolver todas las dependencias entre sus componentes. Dicho algoritmo se conoce como *Ordenamiento Topológico*, el cual es una especialización de *DFS (Depth First Search)*.

Básicamente decimos que se puede hacer un ordenamieto topológico en un gráfica si es que podemos dibujarla de tal manera que todos sus vértices estén sobre una línea horizontal y todos los arcos (que definen el ordenamiento) vayan de izquierda a derecha, es decir, que podamos asignar un índice i, $1 \leq i \leq n$, a cada vértice de la digráfica de manera que $\nexists\, v_j \rightarrow v_k$ tal que $k < j$.

Retomando nuestro ejemplo anterior y haciendo un ordenamiento sobre él, podríamos obtener algo como la digráfica que se muestra en la Figura 3.3, lo cual, yendo de izquierda a derecha significa que primero deben estar definidas las tablas referentes a los vendedores (V), a las sucursales (S) y posteriormente la tabla que las relacione (R).

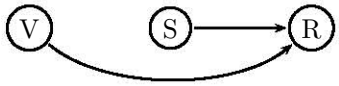

Figura 3.3: Solución al problema de dependencias.

De esta manera, podemos asegurar que al obtener el ordenamiento topológico de los elementos de nuestro documento no tengamos problemas de dependencias a la hora de obtener la salida.

No obstante, el ordenamiento topológico supone que la digráfica no contiene ciclos, los cuales, para nosotros, se traducen en dependencias circulares entre tablas. Por ejemplo, supongamos que tenemos las tablas *empleado* (E) y *departamento* (D), las cuales se relacionan por el hecho de que un empleado trabaja en un departamento y de que un departamento tiene a un empleado como encargado, es decir, tenemos el esquema del Listado 3.7.

Listado 3.7: Elementos de un esquema de BD en la que existe una dependencia circular.

```
1  <table name="empleado">
2    <attribute name="clave" type="char(4)">
3      <comment> Clave unica del empleado </comment>
4      <primaryKey/>
5      <nillable value="false" />
6    </attribute>
7    <attribute name="cve_dep" type="char(4)">
8       ...
9      <foreignKey referencesTable="departamento"  ... />
10   </attribute>
11    ...
12 </table>
13
14 <table name="departamento">
15   <attribute name="cve_dep" type="char(4)">
16     <comment> Clave unica del departamento </comment>
17     <primaryKey/>
18     <nillable value="false" />
19   </attribute>
20   <attribute name="cve_encargado" type="char(4)">
21      ...
22     <foreignKey referencesTable="empleado"  ... />
23   </attribute>
24    ...
25 </table>
```

Además, podemos visualizar los elementos del Listado 3.7 por la digráfica de la Figura 3.4.

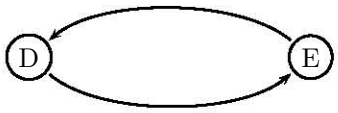

Figura 3.4: Problema de depencia circular.

Por tanto, antes de ejecutar el ordenamiento sobre nuestra digráfica, debemos eliminar los ciclos. Aunque claro, al hacer esto, no debemos restarle importancia a las dependencias que implican los arcos que sean quitados para eliminar los ciclos en la digráfica.

Para eliminar los ciclos primero tenemos que detectarlos, para lo cual se usará DFS. Una vez hecho esto, se podrá ir quitando los arcos necesarios para así obtener una digráfica acíclica en la cual podamos realizar nuestro ordenamiento.

Como se mencionó antes, los arcos removidos representaban referencias a algún otro elemento, los cuales deben verse reflejados también en el esquema generado en SQL, aunque hasta este momento no sea así. La forma de hacerlo es que después de generar la salida de acuerdo al resultado de aplicar el ordenamiento topológico a nuestra digráfica, generemos elementos de SQL que nos permiten modificar algún elemento previamente definido. Así lo único que hacemos a través de éstos es incluir las referencias respectivas a las tablas que estaban definidas por los arcos eliminados. Por ejemplo, dentro de PostgresSQL, dicho elemento se denomina *alter*.

El objetivo del proceso anterior es que podamos obtener elementos en los que ya no tengamos problemas de dependencias circulares, que para nuestro ejemplo anterior se traduce en tener un esquema como el del Listado 3.8, que es la única manera posible en la que se puede introducir una dependencia de este tipo a través de SQL.

Suponiendo que en nuestro ejemplo anterior la arista que se decidió quitar fue la que va de la tabla E a la tabla D (que es la que establece la relación de que cada empleado trabaja en un departamento), el elemento *alter* que permite reincorporarla al esquema sería como el de la línea 15 del Listado 3.8.

Listado 3.8: Esquema de una BD en SQL que contiene una dependencia circular.

```
1 CREATE TABLE "empleado" (
2     clave char(4) PRIMARY KEY NOT NULL ,
3     cve_dep char(4) NOT NULL,
4     ...
```

```
5  ) ;
6
7
8  CREATE TABLE "departamento" (
9      cve_dep char(4) PRIMARY KEY NOT NULL ,
10     cve_encargado char(4) REFERENCES empleado ON DELETE cascade
11                                               ON UPDATE cascade ,
12     ...
13 );
14
15 ALTER TABLE empleado ADD FOREIGN KEY ( cve_dep )
16                     REFERENCES departamento ;
```

Una vez que tenemos un orden correcto con elementos permitidos, podemos pasar a la siguiente fase de nuestra tarea que es precisamente la que genera los enunciadoss en SQL en base a nuestro documento en XML.

3.3. Generación de la salida

Una vez hecho el análisis necesario para identificar los elementos que conforman a nuestro documento que definen a nuestra base de datos, podemos generar la salida, es decir, generar el código necesario en el lenguaje de un manejador de bases de datos específico (que en nuestro caso es PostgreSQL) de tal manera que al introducir el código en nuestro manejador, la base de datos especificada por el documento XML sea generada dentro de él.

Entonces, lo único que se hará será tomar nuestro árbol e ir recorriéndolo de manera que se haga un mapeo de cada uno de los elementos que lo conforman.

Así, retomando de nueva cuenta como ejemplo al elemento *table*, el método que genera la salida deseada luce como se muestra en el Listado 3.9.

Listado 3.9: Generando la salida para el elemento *table*.

```
1  public String table ( Table t ) {
2
3      StringBuffer res = new StringBuffer();
4
5      ...
6
7      res.append("\n \t"+"CREATE TABLE \""+t.getName()+"\" (\n");
8
9      Attribute[] attarray = t.getAttributeArray();
10     for ( int i = 0; i < attarray.length; i++ )
11         res.append ( createAttribute(attarray[i]) );
12
```

```
13      UniqueGroup[] uniarray = t.getUniqueGroupArray();
14      for ( int i = 0; i < uniarray.length; i++ )
15          res.append ( "\n\t  " + createUniqueGroup(uniarray[i]) );
16
17      PrimaryKeyGroup pkg = t.getPrimaryKeyGroup();
18      if ( pkg != null )
19          res.append ( ",\n\t  " + createPrimaryKG(pkg) );
20
21      ForeignKeyGroup[] fkgarray = t.getForeignKeyGroupArray();
22      for ( int i = 0; i < fkgarray.length; i++ )
23          res.append ( ",\n\t  " + createForeignKG(fkgarray[i]) );
24
25      Check[] checks = t.getCheckArray();
26      for ( int k = 0; k < checks.length; k++ )
27          res.append ( ",\n\t  " + createCheck( checks[k] ) );
28
29      res.append("\n \t);\n ");
30
31      return res.toString();
32  }
```

Con esto damos por terminada esta fase de nuestro trabajo, por lo que podemos empezar a describir la que la complementa en el siguiente capítulo.

Capítulo 4

Uso de metainformación para generar DDL

Una vez realizada la fase de la interfaz que toma un documento en XML y entrega una traducción de este en SQL para su ingreso en un SMBD, el siguiente paso a dar es el que va a complementar dicha fase, donde se tomará el diseño de una base de datos en algún SMBD y se convertirá en uno descrito por un documento en XML.

Con el fin de lograr lo anterior, necesitamos tener una manera de que dado un SMBD, podamos obtener la información necesaria para poder formar el esquema de una base de datos de acuerdo al esquema definido por el DDL.

4.1. Catálogos

Como lo mencionamos en el capítulo anterior, este trabajo toma como base al SMBD *PostgreSQL*, en el cual dicha información se encuentra en un conjunto de tablas que se conocen como los *catálogos del sistema*[1].

Los catálogos, no son más que el lugar donde el sistema manejador de bases de datos guarda el esquema de metadatos, tales como información acerca de las tablas y sus atributos, entre otros. Para esa tarea hay comandos en SQL que usa el SMBD para actualizar la metainformación, como por ejemplo *CREATE DATABASE*, que inserta una nueva tupla en el catálogo *pg_database* y a la vez crea la base de datos en disco.

4.2. Proceso de extracción en la metainformación

Con base en los catálogos mencionados en la sección anterior, se puede plantear un proceso con el cual se logre la obtención de la información necesaria a través de la metainformación disponible, de tal manera que sepamos qué elementos conforman una base

[1] http://www.postgresql.org/docs/8.0/interactive/catalogs.html

de datos dada. Tal proceso, planteado para este trabajo, consta de lo siguiente:

- Interactuar con el SMBD de tal manera que podamos obtener la información acerca de los elementos como tablas, vistas, funciones, etcétera que conforman el diseño de una BD.
- Al ir obteniendo la información de cada componente del diseño de la BD, se irán formando los elementos que la encapsulen de acuerdo al DDL, construyendo objetos de la clase correspondiente, tales como *Table*, *View*, *Function*, entre otros[2], de tal manera que cada uno de estos se encargue de formar el segmento del documento en XML que le corresponde con la información que le sea proporcionada.

Para ejemplificar este proceso, el resto del capítulo mostrará su desarrollo para la obtención de las tablas definidas en una BD.

4.3. Extracción de tablas

De acuerdo al primer punto del proceso descrito en la sección anterior, nuestra primer tarea es lograr obtener los nombres de las tablas que se encuentran definidas en un cierto esquema de base de datos y cómo estas fueron declaradas, es decir, de qué atributos constan y con qué restricciones fueron incorporados.

Entonces, lo que haremos será llevar a cabo una serie de consultas (*queries*) al sistema de catálogos del SMBD cuyo resultado sea precisamente la información que necesitamos.

Antes de empezar asumiremos que ya tenemos el nombre de la base de datos y del usuario que tiene control sobre ella.

Así, para la obtención de todos los nombres de las tablas, haremos uso del catálogo *pg_class*, el cual tiene un par de atributos llamados *relkind*, que puede tomar el valor de **r** para indicar que se trata de una tabla y *relname* para obtener el nombre de ésta. Dado lo anterior, la consulta a ocupar se encuentra en el Listado 4.1.

Listado 4.1: Consulta para la obtención de los nombres de las tablas. Esta consulta extrae el nombre de cada una de las tablas definidas dentro del esquema de la base de datos, poniendo el cuidado de no mostrar las tablas que son propias del SMBD (línea 8).

```
1 SELECT s.nspname as nsname, c.oid,
2        c.relname as name,
3        (CASE WHEN c.relkind = 'r' THEN 'Relation' END)::text AS
4        type
5 FROM pg_catalog.pg_class c
```

[2]Definidas en el capítulo 3.

```
6       JOIN
7       pg_catalog.pg_namespace s ON (s.oid =c.relnamespace)
8  WHERE s.nspname NOT IN ('pg_catalog','information_schema',
9                          'pg_toast')
10                 AND c.relkind = 'r'
11 ORDER BY oid;
```

Una vez conocidos los nombres de todas las tablas que pertenecen al esquema, realizamos el paso 2 del proceso de extracción de la metainformación, que es crear la instancia del elemento *Table* (para este caso ver el Listado 4.2), para después en base a este resultado, hacer la extracción de sus atributos, junto con sus propiedades. Para dicha tarea haremos uso del catálogo *pg_attribute* en el cual se encuentra almacenada la información referente a las columnas de las tablas. En particular los atributos:

- *attname*, que almacena el nombre de la columna;
- *atttypid* y *atttypmod* para conocer el tipo de la columna;
- *attNotNull*, es verdadero si la columna tiene restricciones de nulidad o falso en otro caso;
- *atthasdef*, que tiene el valor de verdadero en caso de que la columna tenga definido un valor por omisión y falso en otro caso;
- *attisdropped*, que es verdadero si el atributo se encuentra física pero no lógicamente en la tabla.

Listado 4.2: Crea un ejemplar del elemento table. Ya que se extrajo y se creó un objeto *Table* ejecutaremos el *sql_stament* (línea 7) cuya definición se muestra en el Listado 4.3.

```
1   private void createTable(Connection conn, String oid, String name,
2                            Schema sinc) throws SQLException {
3
4       Table tinc = sinc.addNewTable();
5       tinc.setName(name);
6       ResultSet res = conn.createStatement()
7                       .executeQuery(sql_statement);
8       ...
9   }
```

Listado 4.3: Consulta para la obtención de los atributos de las tablas. Conociendo el identificador de la tabla, digamos *table_oid*, extraemos los nombres y tipos de los atributos que la conforman, así como su nulidad y si tienen definido un valor por omisión.

```
1 SELECT a.attnum, a.attname as attname,
2         pg_catalog.format_type(a.atttypid,a.atttypmod) as atttypname,
3         a.attnotnull, a.atthasdef, a.attisdropped
4 FROM pg_catalog.pg_attribute a
5 WHERE a.attrelid = 'table_oid'::pg_catalog.oid AND
6       a.attnum > 0::pg_catalog.int2
7 ORDER BY a.attrelid, a.attnum;
```

Este catálogo específicamente trata la información general de las columnas de las tablas; por ejemplo sólo sabe si la columna cuenta con un valor por omisión o no, pero no sabe cual es dicho valor. Por esta razón se necesita hacer uso también del catálogo *pg_attrdef*, el cual almacena la definición de dichos valores (ver el Listado 4.4).

Listado 4.4: Consulta para la obtención del valor por omisión para un atributo. En caso de haber determinado que un atributo tiene definido un valor por omisión podemos obtener el valor en cuestión.

```
1 SELECT pg_catalog.pg_get_expr(adbin, adrelid) AS adsrc
2 FROM pg_catalog.pg_attrdef
3 WHERE adrelid = 'table_oid' AND adnum = 'attribute_num';
```

Ya que tenemos las columnas de las tablas junto con las propiedades que representan algunas restricciones, se obtendrán las faltantes (llaves primarias y foráneas, definición de valores específicos, etcétera), haciendo uso del catálogo *pg_constraint* y en particular de sus atributos *conname*, *conkey* y *contype* que nos permiten saber respectivamente el nombre de la restricción, el número de elementos que participan en ella y de qué tipo es. La consulta que nos permite hacerlo se muestra en el Listado 4.5.

Listado 4.5: Consulta para la obtención de restricciones dentro de una tabla.

```
1 SELECT pg_catalog.pg_get_constraintdef(oid) AS consrc,
2         conname as conname, conkey, contype,
3 FROM pg_catalog.pg_constraint
4 WHERE conrelid = 'table_oid';
```

Para el caso que *contype* sea igual a **p** se trata de la restricción de llave primaria, la cual logra la unicidad de cada una de sus tuplas. Para extraerla se hace uso de la consulta del Listado 4.5 por medio del atributo *conkey* (*ckey* en el siguiente listado), siendo su valor el conjunto de atributos que conforman la llave. En nuestro caso si se trata de un solo atributo

se creará una llave primaria a nivel atributo, y si el conjunto tiene cadinalidad mayor que uno se creará una llave primaria a nivel de tabla. Lo anterior se puede ver en el Listado 4.6.

Listado 4.6: Crea un ejemplar de PrimaryKey o PrimaryKeyGroup. Este listado define los métodos que crean las llaves primarias del esquema considerando *ainc* (línea 16) para indicar el único atributo que forma la llave primaria o *atts* (línea 26) para el caso de la llave primaria grupal.

```
1  ...
2  int[] ckey = createAttArray(res.getString("conkey"));
3
4  if (ckey != null && ckey.length == 1) {
5    if(ctype.equals("p"))
6       createPKAttributeConstraint(cname, tinc, att);
7    ...
8  } else if (ckey != null) {
9    if (ctype.equals("p"))
10      createPKTableConstraint(cname, tinc, ckey, atts);
11   ...
12 }
13 ...
14
15 private void createPKAttributeConstraint(String name, Table tinc,
16                                Attribute ainc) throws SQLException {
17    ...
18    PrimaryKey pk = ainc.addNewPrimaryKey();
19    if (name != null)
20        pk.setName(name);
21    if (ainc.isSetNillable())
22        ainc.unsetNillable();
23 }
24
25 private void createPKTableConstraint(String name, Table tinc,
26                                int[] ckey, List atts) {
27
28    PrimaryKeyGroup pk = tinc.addNewPrimaryKeyGroup();
29    pk.setName(name);
30    for(int x = 0; x < ckey.length; x++) {
31        String kname = ((Attribute)atts.get(ckey[x] - 1)).getName();
32        KeyElement kel = pk.addNewKeyElement();
33        kel.setName(kname);
34    }
35 }
```

Además de conocer la unicidad de las tuplas, es importante saber las relaciones que éstas tienen con respecto al resto del esquema; para ello también se hace uso del catálogo *pg_constraint* y su atributo *contype*, que a diferencia de la llaves primarias, toma el valor

de **f**, para indicar a las llaves foráneas. La consulta para determinar lo anterior se muestra en el Listado 4.7.

Listado 4.7: Consulta para la obtención de las llaves foráneas del esquema. Esta consulta obtiene de qué manera se aplicará la actualización o eliminación de las tuplas, así como cuándo vaya a llevar a cabo estas acciones. Para lograrlo, la consulta debe tomar en consideración el identificador de la tabla a la que pertenece la restricción, así como a la que hace referencia (línea 22).

```
1  SELECT n.nspname || '.' || r.relname
2         as relname, a.attname as attname,
3         CASE con.confupdtype WHEN 'c' THEN 'cascade'
4                              WHEN 'n' THEN 'setNull'
5                              WHEN 'd' THEN 'setDefault'
6                              WHEN 'r' THEN 'restrict'
7                              WHEN 'a' THEN 'restrict' END
8                               as attupdate,
9         CASE con.confdeltype WHEN 'c' THEN 'cascade'
10                             WHEN 'n' THEN 'setNull'
11                             WHEN 'd' THEN 'setDefault'
12                             WHEN 'r' THEN 'restrict'
13                             WHEN 'a' THEN 'restrict' END
14                              as attdelete,
15        con.condeferrable as deferrable,
16        CASE con.condeferred WHEN 'f' THEN 'immediate'
17                             WHEN 't' THEN 'deferred' END
18                              as initially
19 FROM pg_class r JOIN pg_attribute a ON (r.oid = a.attrelid)
20      JOIN pg_catalog.pg_namespace n ON (n.oid = r.relnamespace)
21      JOIN pg_constraint con ON (con.connamespace=r.relnamespace)
22 WHERE  con.conrelid= 'table_oid' AND r.oid = con.confrelid;
```

Otra restricción dentro del esquema es que un atributo o un conjunto de atributos cumplan con una expresión dada (*check*). Para obtenerla, así como con las dos restricciones anteriores, se usará *pg_constraint* y *contype* pero el valor de interés para este último será **c**.

Una vez obtenida esta información, podemos construir los objetos correspondientes, como lo muestra el Listado 4.8.

Listado 4.8: Métodos para crear una restricción de tipo *CHECK* en el esquema.

```
1  ...
2  int[] ckey = createAttArray(res.getString("conkey"));
3  if (ckey != null && ckey.length == 1){
4      ...
5      else if (ctype.equals("c"))
```

```
6         createCheckAttributeConstraint(cname, att, res);
7       ...
8   }
9
10  else if (ckey != null) {
11      ...
12      else if(ctype.equals("c"))
13         createCheckTableConstraint(cname,tinc,res);
14   }
15
16  ...
17  private void createCheckAttributeConstraint(String name,
18                            Attribute ainc, ResultSet res)
19                            throws SQLException {
20
21    Check check = ainc.addNewCheck();
22    check.setLanguage("pgsql");
23    check.setName(name);
24    String finalsrc = res.getString("consrc");
25    ...
26    check.setStringValue(finalsrc);
27  }
28
29
30  private void createCheckTableConstraint(String name, Table tinc,
31                           ResultSet res) throws SQLException {
32
33    Check check = tinc.addNewCheck();
34    check.setLanguage("pgsql");
35    check.setName(name);
36    String finalsrc = res.getString("consrc");
37    ...
38    check.setStringValue(finalsrc);
39  }
```

Finalmente para tener todas las propiedades que una tabla contiene hace falta obtener la restricción de que un atributo o conjunto de atributos pueda definirse como único. Siguiendo con nuestro cátalogo *pg_constraint*, el atributo *contype* toma el valor de **u**. Así mismo, después de realizar la consulta respectiva, podemos dar paso a la creación de los objetos correspondientes, esto en base al Listado 4.9.

Listado 4.9: Métodos para crear una restricción de tipo UNIQUE en el esquema.

```
1  ...
2  int[] ckey = createAttArray(res.getString("conkey"));
3   if (ckey != null && ckey.length == 1){
4       ...
```

```
5       else if (ctype.equals("u"))
6           createUniqueAttributeConstraint(cname, att);
7   }
8
9   else if (ckey != null) {
10      ...
11       else if(ctype.equals("c"))
12          createUniqueTableConstraint(cname, tinc, ckey, atts);
13      ...
14   }
15  ...
16
17  private void createUniqueAttributeConstraint(String name,
18                                                Attribute ainc){
19
20          Unique pk = ainc.addNewUnique();
21          if (name != null)
22              pk.setName(name);
23  }
24
25  private void createUniqueTableConstraint(String name,Table tinc,
26                          int[] ckey, List atts) throws SQLException{
27
28          UniqueGroup uniq = tinc.addNewUniqueGroup();
29          uniq.setName(name);
30          for(int x = 0; x < ckey.length; x++) {
31              String kname = ((Attribute)atts.get(ckey[x] - 1))
32                              .getName();
33              KeyElement kel = uniq.addNewKeyElement();
34              kel.setName(kname);
35          }
36  }
```

Con esto damos por terminada la obtención y generación de metainformación entre un documento XML definido por nuestro lenguaje y un esquema creado por el SMBD.

Capítulo 5

Navegación en la metainformación

En los capítulos anteriores dimos una manera en la que podíamos definir una base de datos en un SMBD a través de un documento en XML que la especificara, así como también la forma en la que podríamos generar este documento con base en una base de datos ya definida en el SMBD. En este capítulo, lo que se hará será dar una manera en la que se puede ocupar la metainformación contenida por el archivo en XML con un fin en particular.

5.1. Definición de una aplicación cliente-servidor

Ubiquémonos en el siguiente contexto: tenemos un servidor en el cual están definidas varias bases de datos en un SMBD y alguien (un cliente) desea obtener información sobre ellas, información como qué bases de datos existen o qué elementos contienen. Por tanto, el servidor tendrá que hacer uso de sus recursos para poder atender la solicitud de dicho cliente.

Algunas de las peticiones que un cliente podría hacer son las siguientes:

- Tener una lista de las bases de datos disponibles en el servidor.
- Dada una BD en particular, conocer los esquemas en los cuales está dividida.
- Conocer los elementos (tablas, funciones, etcétera.) que conforman una BD o un esquema en particular de una BD.
- Conocer los elementos de una clase (digamos tablas) que forman parte de una BD.
- Obtener la definición de un elemento que pertenezca a una BD.
- Saber las relaciones que guarda un elemento de una BD con otros elementos definidos dentro de ella.

Dentro de las primeras cuatro peticiones, sería deseable que cada elemento de la lista vaya acompañado de una descripción general.

Dado lo anterior, antes de describir a la aplicación en su conjunto, trataremos con las tareas que serán necesarias del lado del servidor con el fin de resolver estas peticiones.

5.2. Resolución a las peticiones

Es aquí donde podemos introducir el trabajo desarrollado hasta este momento. Podemos plantear tareas específicas en el servidor acordes a las peticiones, de manera que para darles solución se utilicen los documentos en XML que representan a las bases de datos que se encuentran en él. Revisemos cada petición individualmente y veamos lo que será necesario hacer.

5.2.1. Bases de datos disponibles

Dado que el esquema para los documentos en XML con los que hemos trabajado hasta ahora sólo consideran a una BD junto con sus elementos, y teniendo en mente que queremos que la respuesta a una petición se dé con base a un documento de este tipo, es prudente plantear un nuevo esquema que nos ayude a definir un documento en el cual podamos reflejar las bases de datos disponibles por el servidor. Dicho esquema se encuentra en el Listado 5.1.

Listado 5.1: Esquema que permite reflejar las bases de datos en el servidor.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3             xmlns="http://caoba.matem.unam.mx/dbList"
4             targetNamespace="http://caoba.matem.unam.mx/dbList"
5             elementFormDefault="qualified">
6
7  <xs:element name="catalog" type="Catalog" />
8
9  <xs:element name="comment" type="nonEmptyString"/>
10
11 <xs:simpleType name="nonEmptyString">
12   <xs:restriction base="xs:string">
13     <xs:minLength value="1" />
14   </xs:restriction>
15 </xs:simpleType>
16
17 <xs:complexType name="Catalog">
18   <xs:sequence>
19     <xs:element name="database" maxOccurs="unbounded">
20       <xs:complexType>
21         <xs:sequence>
```

```
22              <xs:element ref="comment" minOccurs="0" />
23            </xs:sequence>
24            <xs:attribute name="name" type="nonEmptyString"
25                          use="required" />
26          </xs:complexType>
27        </xs:element>
28      </xs:sequence>
29  </xs:complexType>
30
31  </xs:schema>
```

Así, con este esquema, podremos crear documentos como el del Listado 5.2, en el cual se ve claramente la información que deseábamos reflejar para esta petición.

Listado 5.2: Ejemplar del esquema definido en el Listado 5.1.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <catalog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3           xmlns="http://caoba.matem.unam.mx/dbList"
4           xsi:schemaLocation="http://caoba.matem.unam.mx
5                               dbList.xsd">
6
7  <database name="Zapatita">
8    <comment> Base de Datos que define el
9              sistema utilizado en una zapatería
10   </comment>
11 </database>
12
13 <database name="Mascota">
14   <comment> Base de Datos que define el
15             sistema utilizado en una tienda de mascotas
16   </comment>
17 </database>
18
19 </catalog>
```

Una vez que definimos la respuesta que queremos dar a esta petición, nos falta describir lo que tendrá que llevar a cabo el servidor para generar un documento como el del Listado 5.2. Dicho proceso se puede describir de la sigiente manera:

- El servidor accederá al SMBD y hará una consulta en su sistema de catálogos, obteniendo de esta forma los nombres de las bases de datos disponibles junto con sus descripciones.
- En base al resultado de la consulta, se generará el documento en XML deseado.

La consulta que llevará a cabo la tarea especificada se encuentra en el Listado 5.3.

Listado 5.3: Consulta que obtiene los nombres de las bases de datos disponibles.

```
1 SELECT d.datname as "Name", u.usename as "Owner",
2        pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding"
3 FROM pg_catalog.pg_database d, pg_catalog.pg_user u
4 WHERE d.datdba = u.usesysid
5 ORDER BY 1;
```

5.2.2. Esquemas de una BD

A diferencia de la petición descrita en la subsección anterior, el conocer los esquemas de los que se compone una BD sólo hace que se tome en cuenta a un ente, una BD. Por tanto, retomando el trabajo desarrollado, es posible especificar un documento en XML que refleje solamente esta información en base al esquema definido en el capítulo 2 (Listado 2.1).

Entonces, un ejemplo de un documento en XML que sería una respuesta a una petición por parte del cliente es el mostrado en el Listado 5.4.

Listado 5.4: Documento XML que muestra los esquemas de los que consta una BD.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <database xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3            xmlns="http://caoba.matem.unam.mx"
4            xsi:schemaLocation="http://caoba.matem.unam.mx DDL.xsd"
5            name="Zapatita">
6
7  <schema name="esquema_1">
8    <comment> descripcion del esquema 1 </comment>
9  </schema>
10
11 <schema name="esquema_2" />
12
13 </database>
```

Para poder generar este tipo de documentos, nuestro servidor reutilizará métodos desarrollados en el capítulo 4, en donde se tomaban en cuenta para la salida de un documento en XML todos los elementos de la base de datos; sin embargo, para este caso, sólo se tomará en cuenta a los esquemas.

5.2.3. Elementos que componen una BD o un esquema de esta

Para la petición de conocer únicamente qué elementos conforman una base de datos o un esquema que se encuentre dentro de esta, queremos que el documento en XML que represente la respuesta a esta petición sea como el mostrado en el Listado 5.5.

Listado 5.5: Documento que muestra los elementos de una BD o un esquema.

```
<?xml version='1.0' encoding='UTF-8'?>
<database xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://caoba.matem.unam.mx/firstLevelElems"
          xsi:schemaLocation="http://caoba.matem.unam.mx
                              firstLevelElems.xsd"
          name="base_datos_prueba">
  <comment> Base de datos de prueba </comment>
  <schema name="esquema_1">
    <table name="tabla_1">
      <comment> una tabla en base_datos_prueba </comment>
    </table>
    <table name="tabla_2">
      <comment> otra tabla en base_datos_prueba </comment>
    </table>
    <view name="vista_1">
      <comment>
        una vista en base_datos_prueba que involucra a tabla_1
      </comment>
    </view>
  </schema>
</database>
```

Parecería que un documento como este se define en base al esquema especificado en el capítulo 2 (Listado 2.1); sin embargo, dicho esquema nos restringe a que el interior de un elemento *table*, por ejemplo, no sea vacío; por tanto, es necesario definir un nuevo esquema que nos permita reflejar dicha vacuidad.

El nuevo esquema definido para esta tarea se muestra en el Listado 5.6.

Listado 5.6: Esquema que define al documento de respuesta a la tercera petición.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://caoba.matem.unam.mx/firstLevelElems"
           targetNamespace="http://caoba.matem.unam.mx/
                            firstLevelElems"
           elementFormDefault="qualified">

```

```
<xs:element name="database" type="Database" />
<xs:element name="comment" type="nonEmptyString" />

<xs:simpleType name="nonEmptyString">
  <xs:restriction base="xs:string">
    <xs:minLength value="1" />
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="Database">
  <xs:annotation>
    <xs:documentation xml:lang="en">
        A database should be formed by schemas.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element ref="comment" minOccurs="0" />
    <xs:element name="schema" type="Schema"
                maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="name" type="nonEmptyString"
                use ="required" />
</xs:complexType>

<xs:complexType name="Schema">
  <xs:annotation>
    <xs:documentation xml:lang="en">
        An schema could contain tables, functions, triggers, etc.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element ref="comment" minOccurs="0" />
    <xs:choice maxOccurs="unbounded">
      <xs:element name="table" type="Comment" minOccurs="0" />
      <xs:element name="procedure" type="Comment" minOccurs="0" />
      <xs:element name="function" type="Comment" minOccurs="0" />
      <xs:element name="trigger" type="Comment" minOccurs="0" />
      <xs:element name="view" type="Comment" minOccurs="0" />
      <xs:element name="index" type="Comment" minOccurs="0" />
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="name" type="nonEmptyString" use ="required" />
</xs:complexType>

<xs:complexType name="Comment">
  <xs:sequence>
    <xs:element ref="comment" minOccurs="0" />
  </xs:sequence>
  <xs:attribute name="name" type="nonEmptyString" use="required" />
```

```
57 </xs:complexType>
58
59 </xs:schema>
```

Así como en la petición anterior, se reutilizarán métodos descritos en el capítulo 4, pero ahora considerarando solamente este tipo de elementos.

5.2.4. Elementos de una misma clase que conforman un esquema de una BD

En cuanto a la tarea de seleccionar todos los elementos de una misma clase (por ejemplo, sólo tablas) de una base de datos o un esquema de esta, deseamos que el documento en XML que represente la respuesta a esta petición luzca como el mostrado en el Listado 5.7.

Listado 5.7: Documento con los elementos de una misma clase de una BD o un esquema.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <database xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3            xmlns="http://caoba.matem.unam.mx/sameFirstLevelElems"
4            xsi:schemaLocation="http://caoba.matem.unam.mx
5                                samefirstLevelElems.xsd"
6            name="base_datos_prueba">
7    <comment> Base de datos de prueba </comment>
8
9    <schema name="esquema_1">
10     <table name="tabla_1">
11       <comment> una tabla en base_datos_prueba </comment>
12     </table>
13     <table name="tabla_2">
14       <comment> otra tabla en base_datos_prueba </comment>
15     </table>
16   </schema>
17
18 </database>
```

Si quisiéramos definir un documento como este con base en los esquemas definidos anteriormente, veríamos que ninguno de ellos nos permite hacerlo, por lo que para este tipo de petición también será necesario establecer un esquema. Este esquema es prácticamente el mismo que el de la subsección anterior, salvo que como ahora sólo queremos elementos del mismo tipo, las líneas de la 40 a la 47 (la etiqueta *choice*) cambian por las del Listado 5.8.

Listado 5.8: Esquema que define al documento de respuesta para la la cuarta petición.

```
1       <xs:choice>
2         <xs:element name="table" type="Comment" minOccurs="0"
3                     maxOccurs="unbounded" />
4         <xs:element name="procedure" type="Comment" minOccurs="0"
5                     maxOccurs="unbounded" />
6         <xs:element name="function" type="Comment" minOccurs="0"
7                     maxOccurs="unbounded" />
8         <xs:element name="trigger" type="Comment" minOccurs="0"
9                     maxOccurs="unbounded" />
10        <xs:element name="view" type="Comment" minOccurs="0"
11                    maxOccurs="unbounded" />
12        <xs:element name="index" type="Comment" minOccurs="0"
13                    maxOccurs="unbounded" />
14      </xs:choice>
```

La diferencia radica principalmente en el uso que le damos al atributo *maxOccurs* en ambos esquemas dentro de nuestros elementos.

En cuanto a qué tareas debe llevar a cabo el servidor para poder generar la respuesta a este tipo de peticiones, basta decir que son las mismas que las definidas en el capítulo anterior, pero ahora tomando en cuenta sólo al elemento deseado en la petición.

5.2.5. Descripción de un único elemento dentro de la BD

En esta tarea se pide información más detallada acerca de un único elemento contenido dentro de la BD, es decir, cuál es su definición. Si por ejemplo, se hiciera una petición como esta al servidor, con la tabla *persona* contenida en el esquema *prueba* de la base de datos *zapatita*, el documento que se generaría como respuesta sería el que se encuentra en el Listado 5.9.

Listado 5.9: Documento que muestra la descripción de un solo elemento de una BD.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2
3  <database xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4            xmlns="http://caoba.matem.unam.mx"
5            xsi:schemaLocation="http://caoba.matem.unam.mx DDL.xsd"
6            name="zapatita">
7
8    <schema name="Zapata">
9
10     <table name="persona">
11       <attribute name="curp" type="text">
12         <primaryKey />
13         <check language="PSQL">
```

```
14            <definition>
15              char_length(trim(curp))  &lt;&gt; 10
16            </definition>
17          </check>
18        </attribute>
19        <attribute name= "nombre" type="text"/>
20        <attribute name="calle" type="text"/>
21        <attribute name="numero_exterior" type="int"/>
22        <attribute name="numero_interior" type="int">
23          <nillable value="true"/>
24        </attribute>
25        <attribute name="colonia" type="text"/>
26        <attribute name="codigo_postal" type="int">
27          <nillable value="true"/>
28        </attribute>
29      </table>
30
31    </schema>
32  </database>
```

Como vemos, su estructura es idéntica a la que hemos trabajado en los capítulos 3 y 4, por lo que podemos reutilizar el esquema definido en el capítulo 2 como base para documentos en XML que se generarán como respuesta en esta tarea.

5.2.6. Dependencias de un elemento con respecto a otros dentro de la BD

Como se aprecia en la descripción de las tareas anteriores, estas tienen que ver con los diferentes niveles de granularidad, con respecto a los niveles de los que puede constar el árbol que representa a un documento en XML definido en base al DDL; sin embargo, esta tarea no sigue el mismo patrón. Es por esto que es necesario definir un nuevo esquema que nos permita representar la respuesta a este tipo de petición, esquema que se encuentra en el Listado 5.10.

Listado 5.10: Esquema que define el documento de respuesta a la sexta petición.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
4             xmlns="http://caoba.matem.unam.mx/dependencies"
5             targetNamespace="http://caoba.matem.unam.mx/dependencies"
6             elementFormDefault="qualified">
7
8  <xs:element name="dependencies" type="Dependencies" />
9
10 <xs:complexType name="Dependencies">
```

```
11    <xs:sequence>
12      <xs:element name="table" type="Reference" minOccurs="0"
13                  maxOccurs="unbounded" />
14      <xs:element name="trigger" type="Reference" minOccurs="0"
15                  maxOccurs="unbounded" />
16      <xs:element name="function" type="Reference" minOccurs="0"
17                  maxOccurs="unbounded" />
18      <xs:element name="view" type="Reference" minOccurs="0"
19                  maxOccurs="unbounded" />
20      <xs:element name="index" type="Reference" minOccurs="0"
21                  maxOccurs="unbounded" />
22    </xs:sequence>
23    <xs:attribute name="db" type="nonEmptyString" use="required" />
24  </xs:complexType>
25
26  <xs:complexType name="Reference">
27    <xs:sequence>
28      <xs:element name="reference" maxOccurs="unbounded">
29        <xs:complexType>
30          <xs:attribute name="name" type="nonEmptyString"
31              use="required" />
32          <xs:attribute name="class" type="Class" use="required" />
33        </xs:complexType>
34      </xs:element>
35    </xs:sequence>
36    <xs:attribute name="name" type="nonEmptyString" use="required" />
37  </xs:complexType>
38
39  <xs:simpleType name="nonEmptyString">
40    <xs:restriction base="xs:string">
41      <xs:minLength value="1" />
42    </xs:restriction>
43  </xs:simpleType>
44
45  <xs:simpleType name="Class">
46    <xs:restriction base="nonEmptyString">
47      <xs:enumeration value="table"/>
48      <xs:enumeration value="function"/>
49      <xs:enumeration value="trigger"/>
50      <xs:enumeration value="view"/>
51      <xs:enumeration value="index"/>
52    </xs:restriction>
53  </xs:simpleType>
54
55  </xs:schema>
```

Con este esquema, un documento que sea la respuesta a una petición de este tipo será como el del Listado 5.11.

Listado 5.11: Documento que muestra las dependencias de un elemento dentro de una BD.

```
<?xml version='1.0' encoding='UTF-8'?>

<dependencies xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns="http://caoba.matem.unam.mx/dependencies"
              xsi:schemaLocation="http://caoba.matem.unam.mx
                                  dependencies.xsd"
              db="zapatita" >

  <table name="vendedor">
    <reference name="persona" />
  </table>

  <table name="es_vendedor_de">
    <reference name="vendedor" />
    <reference name="sucursal" />
  </table>

</dependencies>
```

Para resolver peticiones de este tipo hay que buscar dentro del elemento en cuestión a qué otros elementos dentro de la base de datos se hace referencia, por lo que el encontrar tales referencias dependerá del tipo de objeto con el que se esté tratando. Así por ejemplo, para encontrar las dependencias de una tabla en particular, lo que haremos será buscar dentro de su llave foránea con qué tabla o tablas se relaciona.

Como se estableció en el capítulo 3, en el presente trabajo sólo se manejaron las dependencias para tres elementos dados por el DDL: las tablas, los *triggers* y los índices. Sin embargo, el uso de uno de los catálogos con los que cuenta el SMBD *PostgreSQL*, *pg_depend*, nos permitirá obtener las dependencias para otros elementos como lo son las vistas.

5.3. Implementación gráfica

Una vez descrito el conjunto de peticiones y la forma en que se dará solución a cada una de ellas, podemos presentar la forma en la que se incorporarán gráficamente[1].

Como mencionamos anteriormente, la aplicación será del tipo cliente-servidor y por la forma en la que está planteada, sólo necesitaremos darle una cara a la parte del cliente, cara por medio de la cuál este pueda ejecutar alguna de las peticiones consideradas.

Gráficamente, nuestra aplicación luce como en la Figura 5.1, cuya pantalla se divide principalmente en cuatro partes. La primera es la superior izquierda, en la cual el usuario

[1]Esta implementación se hizo en el lenguaje de programación Java.

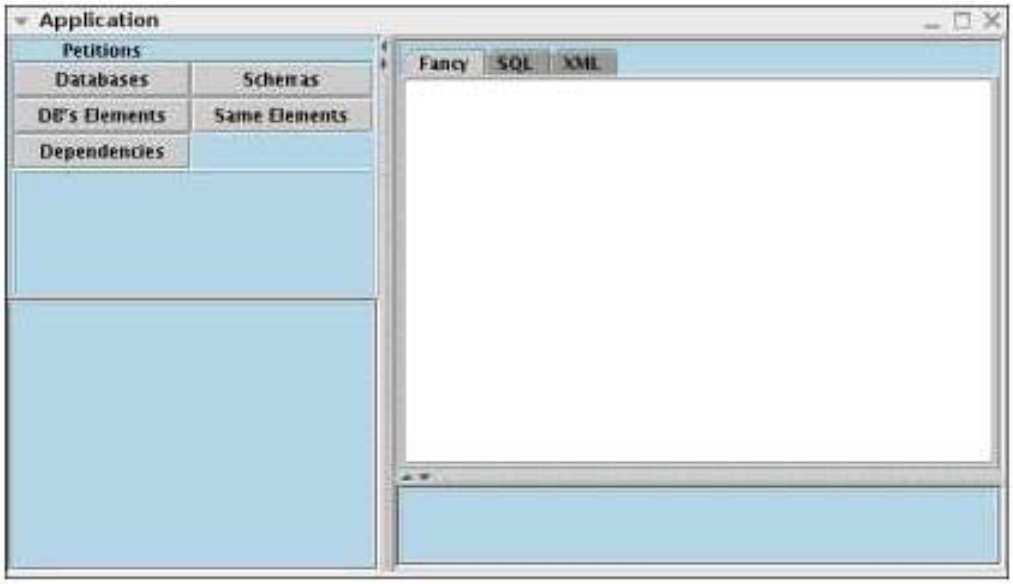

Figura 5.1: Vista general de la aplicación que navega en la metainformación.

podrá solicitar la ejecución de alguna de nuestras peticiones. La segunda, la inferior izquierda, permitirá mostrar un árbol que represente al documento en XML en cuestión de acuerdo a la petición activa en un cierto momento. La superior derecha consta de un conjunto de pestañas rotuladas con las leyendas *XML*, *SQL* y *Fancy View*, que como sus nombres lo indican, presentarán la versión en XML, SQL y otro formato como el del SMBD *PostgreSQL* respectivamente de la petición en curso[2]. Por último, en la parte inferior derecha, se mostrará únicamente el texto que, para cada elemento dentro del documento en XML en cada petición, haya sido incluido como comentario.

Por otra parte, la aplicación también hará uso de las siguientes imágenes para identificar algunos de los elementos que se manejen:

 Representa a un elemento del tipo *database*.

 Identifica a un elemento del tipo *table*.

 Representa a un elemento del tipo *function*.

 Identifica a un elemento del tipo *trigger*.

[2]Se hablará más específicamente de estos elementos en las subsecciones siguientes.

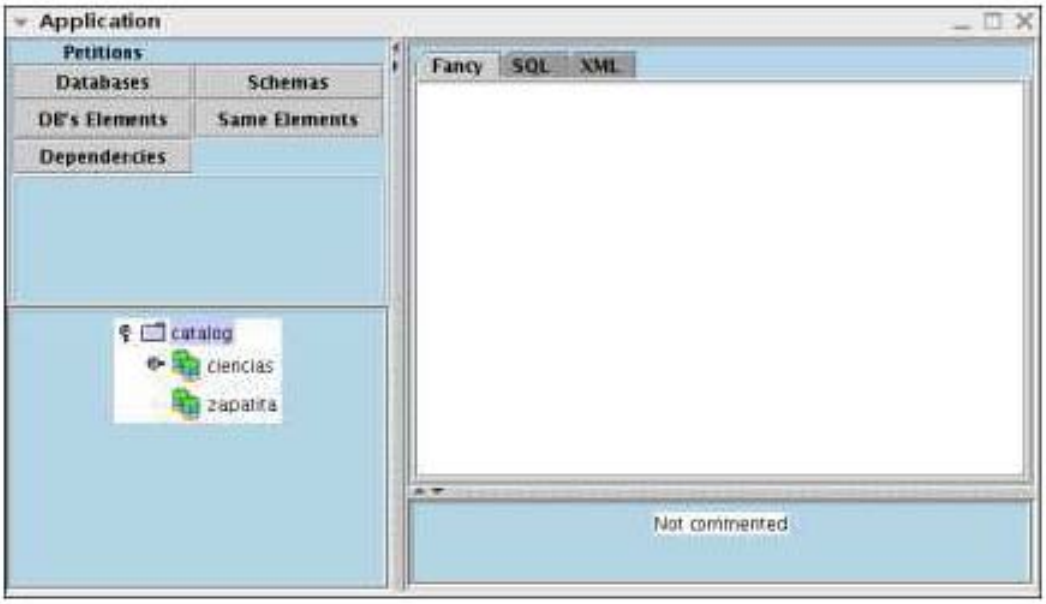

Figura 5.2: Vista al presionar la opción *Databases*.

 Representa a un elemento del tipo *view*.

 Identifica a un elemento del tipo *index*.

 Representa a un elemento del tipo *comment*.

Ahora que ya conocemos a grandes rasgos nuestra aplicación, lo consiguiente será mostrar cómo es que esta cambia en base a las peticiones descritas en la sección 5.2.

5.3.1. Bases de datos disponibles

Como se describió en la subsección 5.2.1, esta tarea tendrá como objetivo mostrar el conjunto de bases de datos disponibles en el servidor al que se tenga acceso desde nuestra aplicación.

Para llevar a cabo dicha tarea, basta con seleccionar la opción *Databases*, lo cual hará que en la pantalla aparezca otro componente (un árbol), como se muestra en la Figura 5.2.

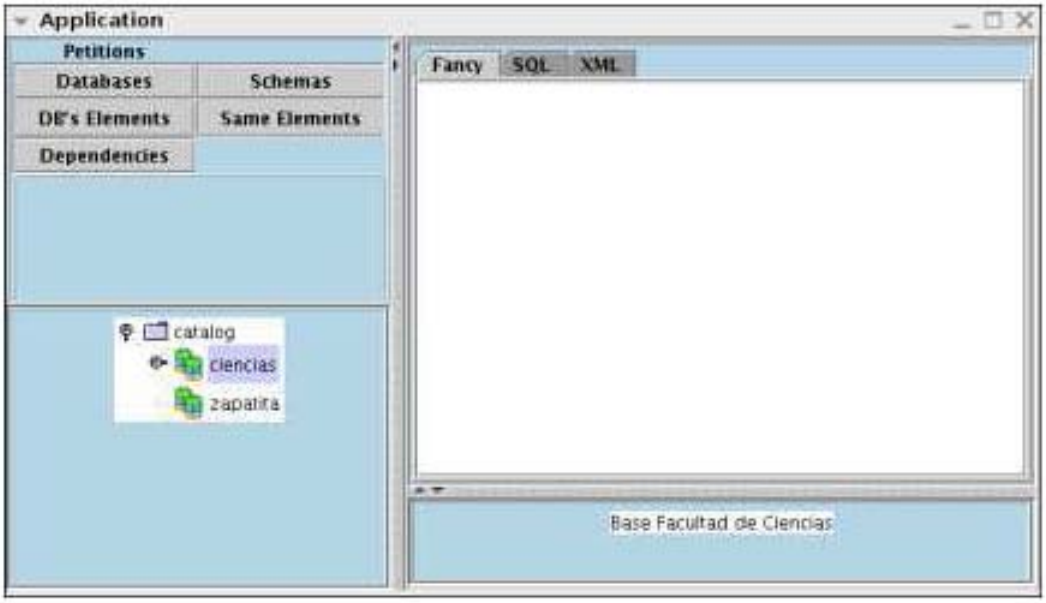

Figura 5.3: Bases de datos disponibles vistas desde la aplicación.

Al inicio el árbol aparecerá de manera comprimida y su objetivo es mostrar de manera jerárquica el documento en XML que representa a nuestra petición, que para ser más concisos, se hará en un conjunto de elementos *database* dentro de uno *catalog*. Por otra parte, cada vez que seleccionamos un elemento distinto dentro del árbol, que corresponde a una BD, deberá de mostrarse su comentario en la sección respectiva, como se aprecia en la Figura 5.3.

5.3.2. Esquemas de una BD

Recordemos que esta petición (descrita en la sección 5.2.2) tiene como objetivo mostrar los esquemas de los que se compone una BD en cuestión. Para tal fin, se incorporó la opción con la leyenda *Schemas*.

El resultado al presionar dicha opción puede variar de acuerdo a lo siguiente:

- Si la petición de la subsección anterior fue activada en primer lugar y alguna de las bases de datos fue seleccionada en el árbol, entonces la aplicación mostrará de manera inmediata los esquemas que correspondan a dicha BD, como se puede ver en la Figura 5.4).
- En otro caso, si no existe una BD de referencia, la aplicación permitirá escoger de una lista aquélla de la que se requiera obtener la información de interés.

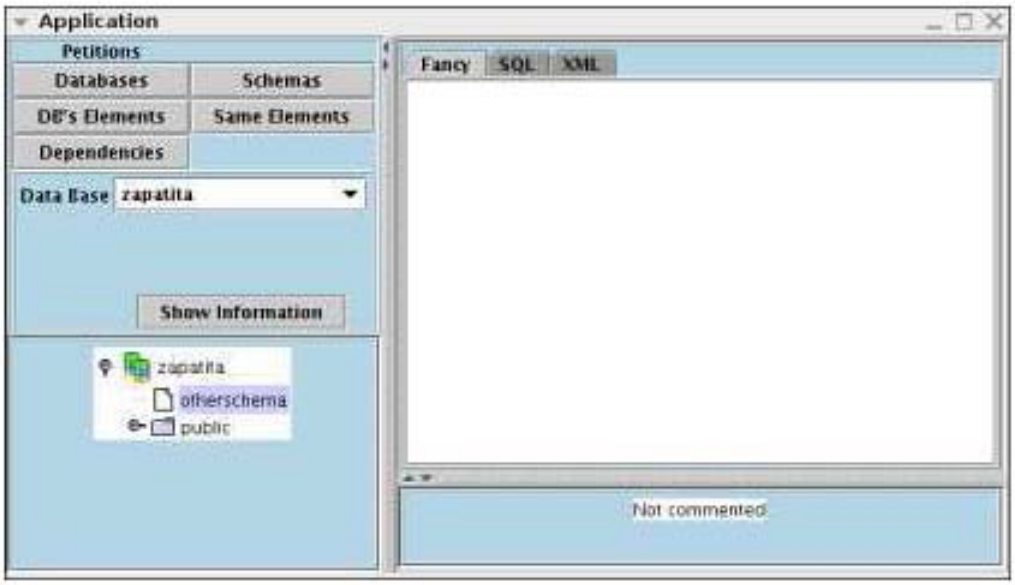

Figura 5.4: Esquemas de una BD vistos desde la aplicación.

No obstante, independientemente de cualquiera de los dos casos que se suscite, la aplicación permitirá seleccionar posteriormente alguna otra BD si así se desea, actualizando la información desplegada al presionar la opción *show*, como en el caso de la Figura 5.5).

Un ejemplo para esta petición dentro de nuestra aplicación se muestra en la Figura 5.4.

5.3.3. Elementos que componen una BD o un esquema de esta

Con lo que respecta a esta petición, recordemos que su objetivo era el de obtener los diferentes elementos de los que se compone una BD o un esquema contenido en esta, es decir, las distintas tablas, funciones, vistas, etcétera. Así, nuestra aplicación cumple este objetivo al presionarse la opción *DB's Elements*.

Al igual que en la petición anterior, el resultado en la pantalla al seleccionar esta opción varía de acuerdo a si fue la primera en ser seleccionada al iniciar la aplicación, para lo cual será indispensable seleccionar la BD y el esquema sobre el cual se desee actuar , que es el caso de la Figura 5.6, o si hubo otras peticiones en las que se haya seleccionado una BD en particular y alguno de sus esquemas, lo cual hará que el árbol muestre de manera directa los elementos requeridos, como en la Figura 5.7.

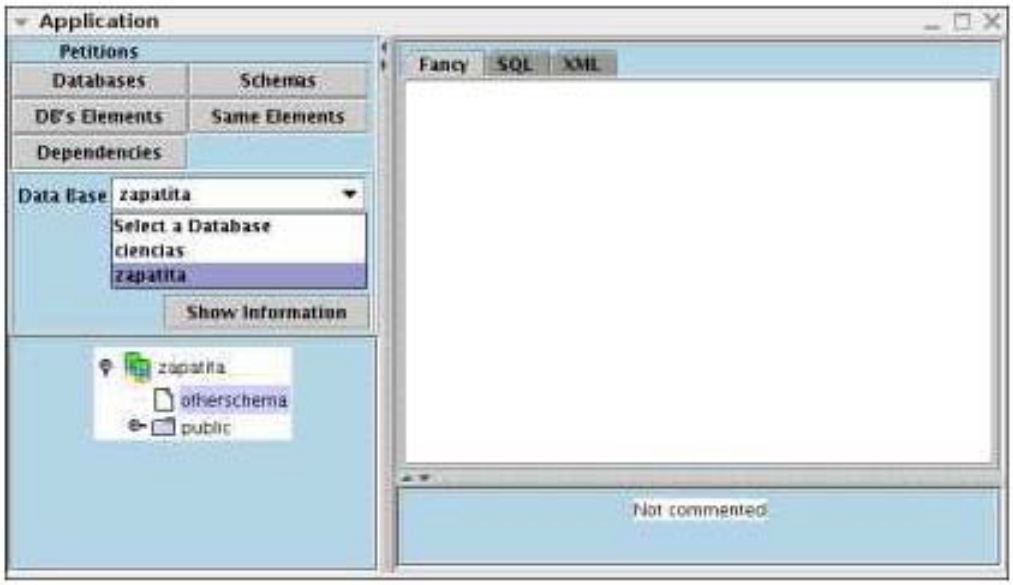

Figura 5.5: Selección de una nueva BD para mostrar sus esquemas.

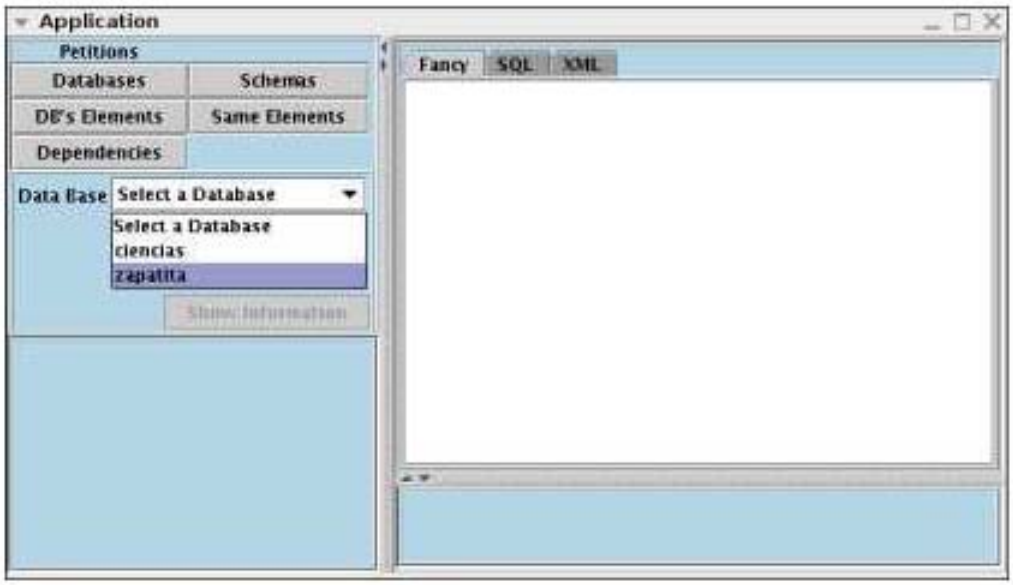

Figura 5.6: Selección de una BD para conocer los elementos de alguno de sus esquemas.

5.3.4. Elementos de una misma clase que conforman un esquema de una BD

Como se mencionó anteriormente, el objetivo de definir esta petición es poder acceder a los elementos de una misma clase que se encuentren definidos dentro de un esquema de

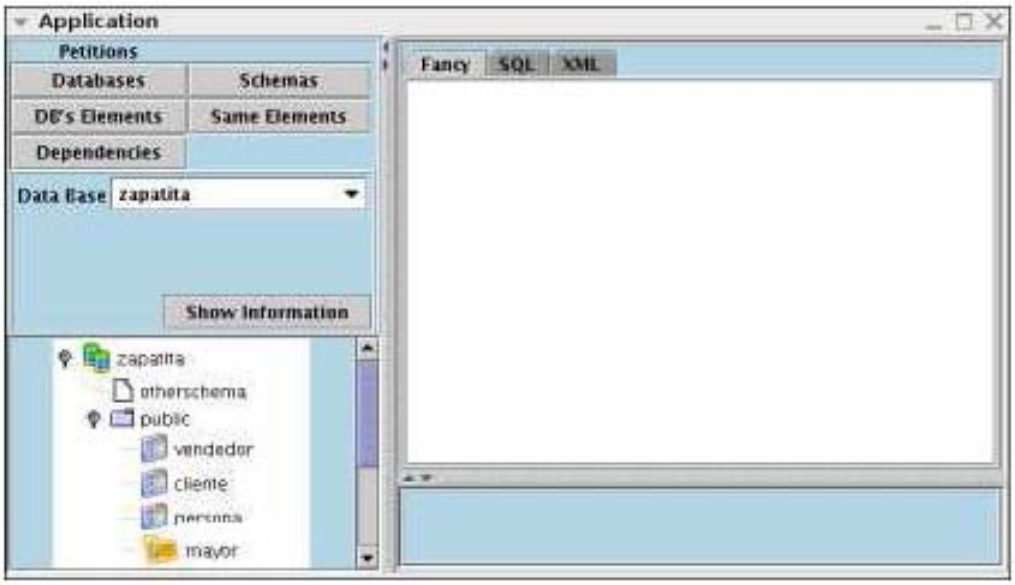

Figura 5.7: Elementos de un esquema en una BD vistos desde la aplicación.

una BD, es decir, consultar solamente las tablas, funciones, *triggers*, vistas ó índices de los cuales esta se conforme.

Para poner en marcha esta tarea, la aplicación cuenta con la opción *Same Elements*, por medio de la cual, al ser la primera en ejecutarse al abrir la aplicación, dará paso a nuevos componentes que pedirán al usuario que seleccione una BD, un esquema de esta y la clase sobre la cual se desea obtener información, como se muestra en la Figura 5.8.

Por otro lado, si hubo alguna petición hecha con anterioridad a esta en la cual se hubiesen seleccionado una BD y alguno de sus esquemas, la selección de la opción que lleva a cabo nuestra tarea sólo demandará al usuario la selección de la clase de elementos deseada, como en la Figura 5.9. Posteriormente, para poder visualizar los datos requeridos, se tendrá que seleccionar la opción *Show Information*.

Para acabar de describir esta tarea, no está de más mencionar que si el usuario desea cambiar de BD o esquema o clase de elementos, lo puede hacer en cualquier momento al presionar alguno de los componentes que contienen respectivamente a los elementos de los que se dispone dentro del servidor.

5.3.5. Descripción de un único elemento dentro de una BD

La aplicación no cuenta con una opción en particular que lleve a cabo esta tarea, ya que esta fue incorporada en las anteriores, de manera que al obtener el árbol jerárquico

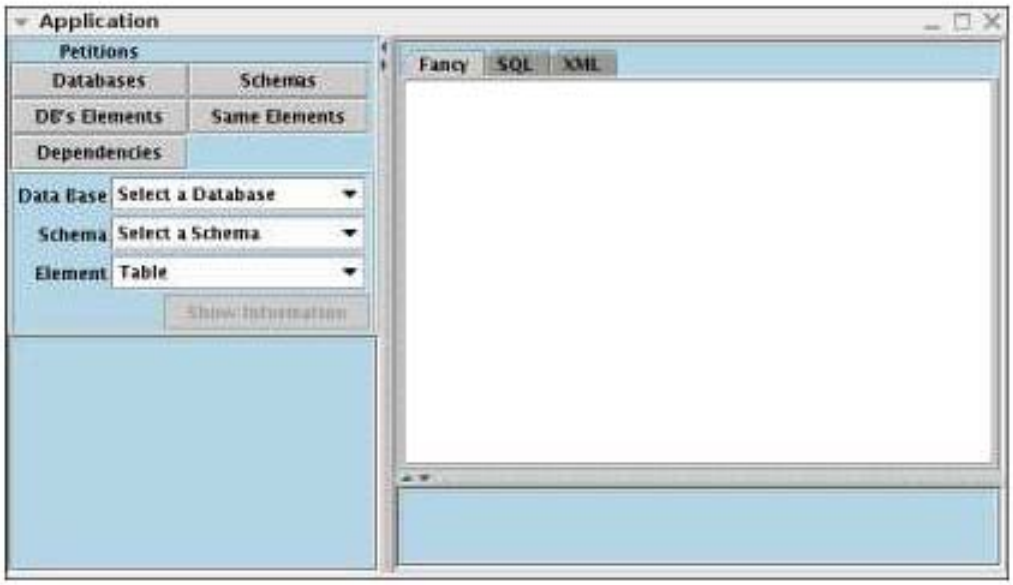

Figura 5.8: Selección de la BD, el esquema y la clase de los elementos a consultar.

y seleccionar a un elemento de él, uno pueda ver su definición. Es aquí donde toman sentido las tres diferentes pestañas que se encuentran del lado derecho de la ventana de la aplicación.

El objetivo de estas pestañas dentro de la aplicación es mostrar la definición de algún elemento por medio de tres vistas distintas, vistas que de alguna u otra manera tienen que ver con la forma en que se representa un elemento en algún SMBD o de acuerdo a el DDL definido en este trabajo.

En general, la selección de cualquier elemento del árbol hará que se active la pestaña rotulada con la leyenda de *XML*; por otro lado, las tres pestañas en conjunto serán activadas solamente cuando el elemento seleccionado del árbol sea una tabla, una función, un *trigger*, una vista o un índice.

La descripción de las tres vistas disponibles es la siguiente:

Fancy Muestra la definición del elemento con un formato más entendible para cualquier usuario.

SQL Muestra la definición que debió seguir el elemento de acuerdo a la sintaxis del lenguaje SQL para ser incorporado a la BD.

XML Muestra la definición del elemento seleccionado por medio de la sintaxis del lenguaje XML, basándose en las etiquetas definidas para él por medio del DDL.

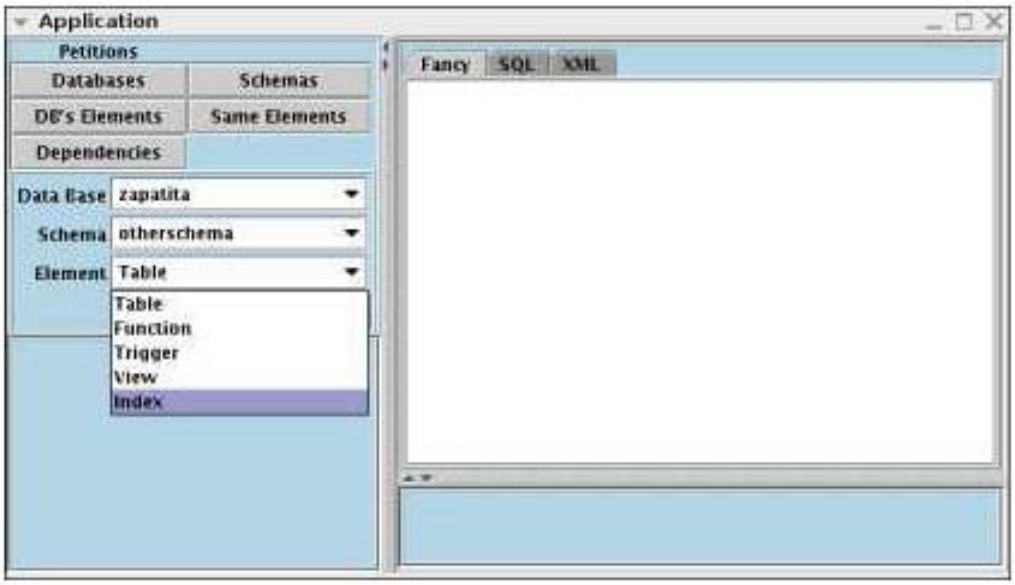

Figura 5.9: Elementos de una misma clase que componen el esquema de una BD.

Para ejemplificar su utilización, supongamos que la tabla *cliente* que pertenece al esquema público de la base de datos *zapatita* fue seleccionada; entonces, las tres distintas vistas se muestran en la figuras 5.10, 5.11 y 5.12, respectivamente.

5.3.6. Dependencias de un elemento con respecto a otros dentro de una BD

Para terminar de cubrir el conjunto de peticiones que se plantearon al principio de este capítulo, sólo falta describir como es que la obtención de dependencias de un elemento que pertenece a una BD se ve reflejado dentro de nuestra aplicación.

Para ejecutar esta petición se cuenta con la opción *Dependencies*, que al presionarla, hará que la aplicación nos permita seleccionar el valor de dos campos necesarios para llevarla a cabo, los cuales son el nombre de una BD y un esquema de esta, como en la Figura 5.13).

Una vez introducidos dichos campos y presionada la opción de *Show Information*, se activará la pestaña que corresponde al formato de XML mostrando la información correspondiente de acuerdo al esquema definido en la sección 5.2.6. Un ejemplo se muestra en la Figura 5.14.

Como también mencionamos en la sección 5.2.6, esta petición no seguía el mismo

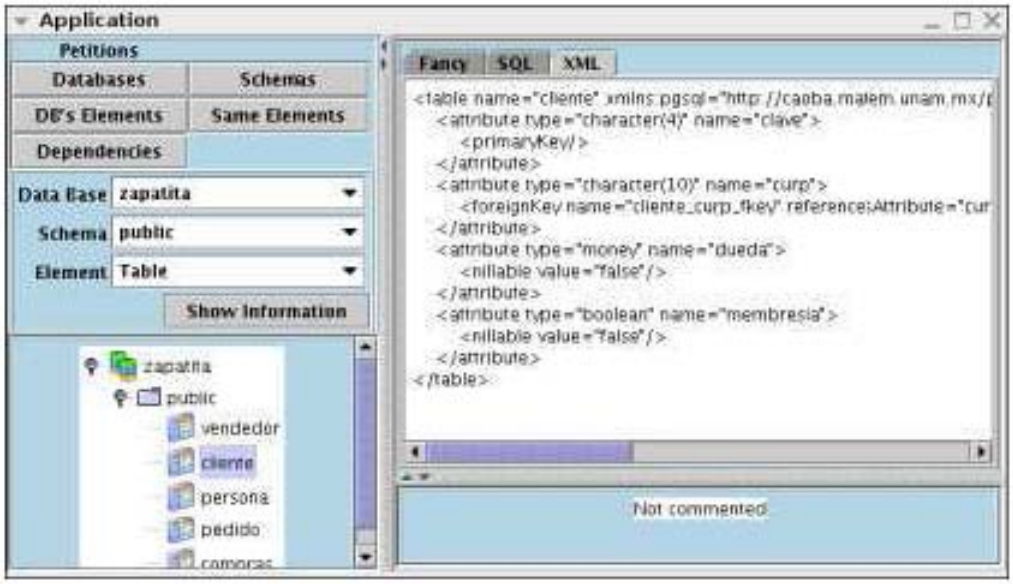

Figura 5.10: Vista tipo *XML* para una tabla.

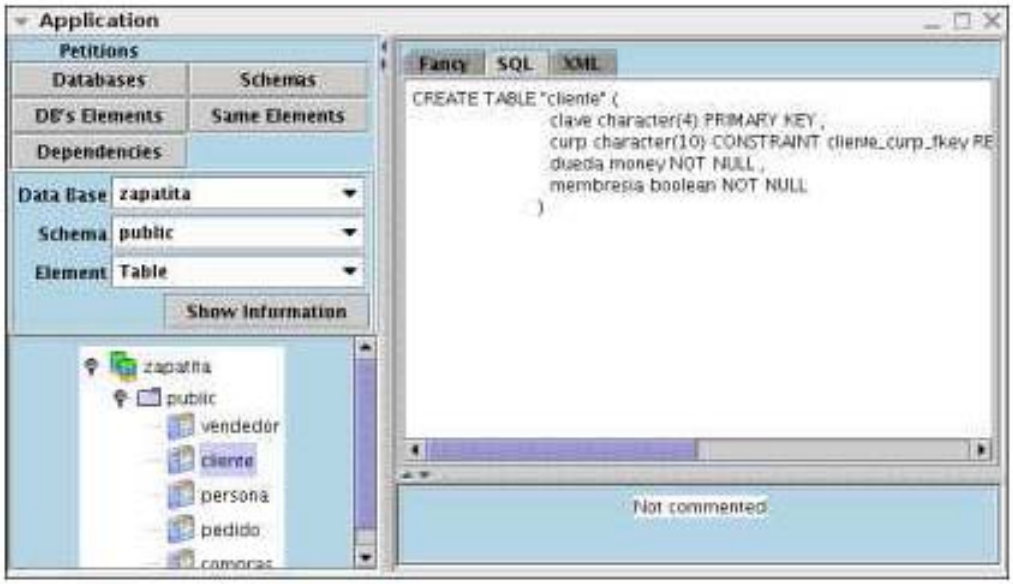

Figura 5.11: Vista tipo *SQL* para una tabla.

patrón que las anteriores; en lo que se refiere a la manera de obtener la información necesaria a través de una serie de consultas también existe una diferencia. Dichas

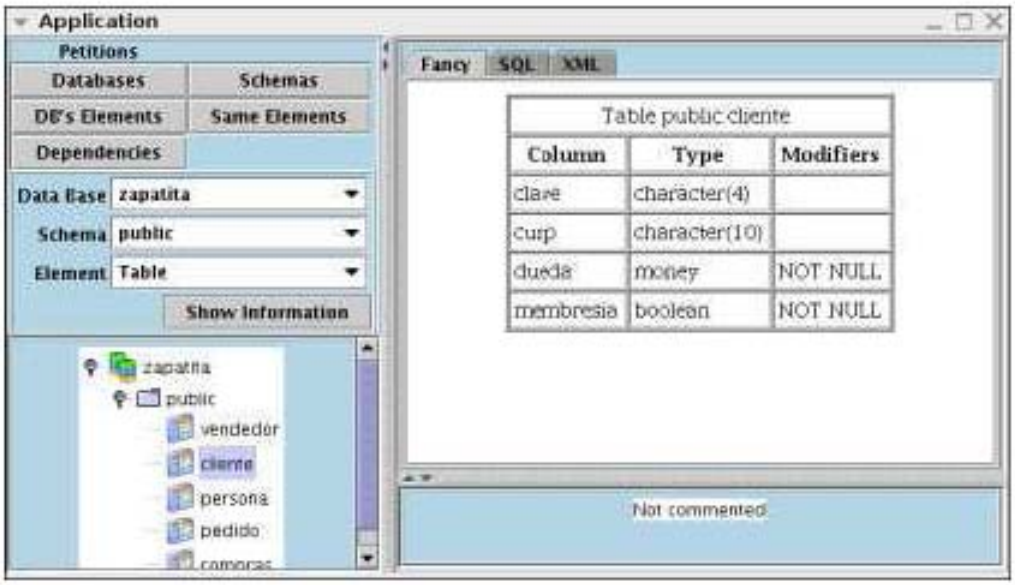

Figura 5.12: Vista tipo *Fancy* para una tabla.

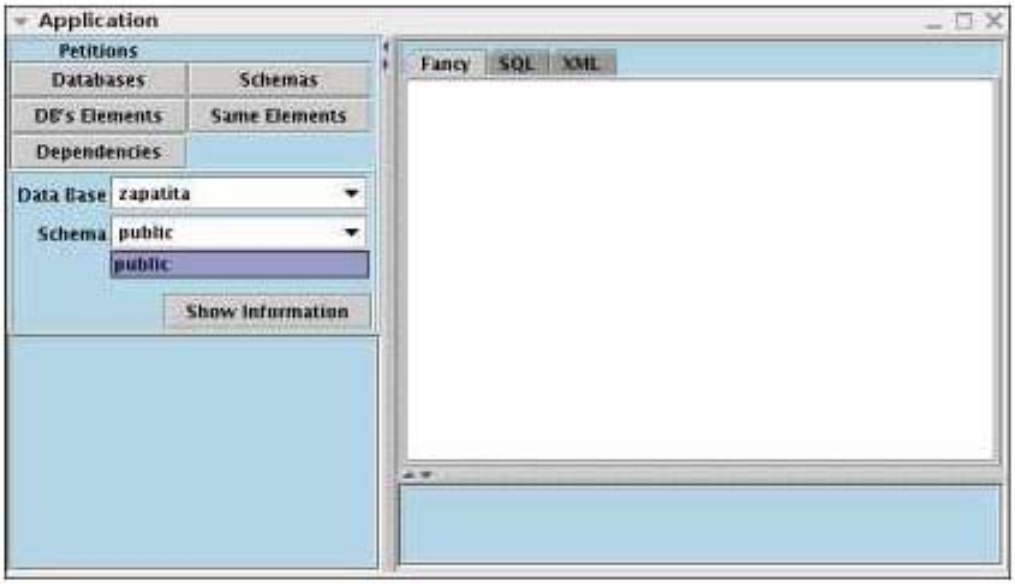

Figura 5.13: Selección de la opción *Dependencies* en la aplicación.

consultas hacen uso de un nuevo catálogo de PostgreSQL llamado *pg_depend*, en el cual el SMBD maneja la metainformación referente precisamente a las dependencias de los

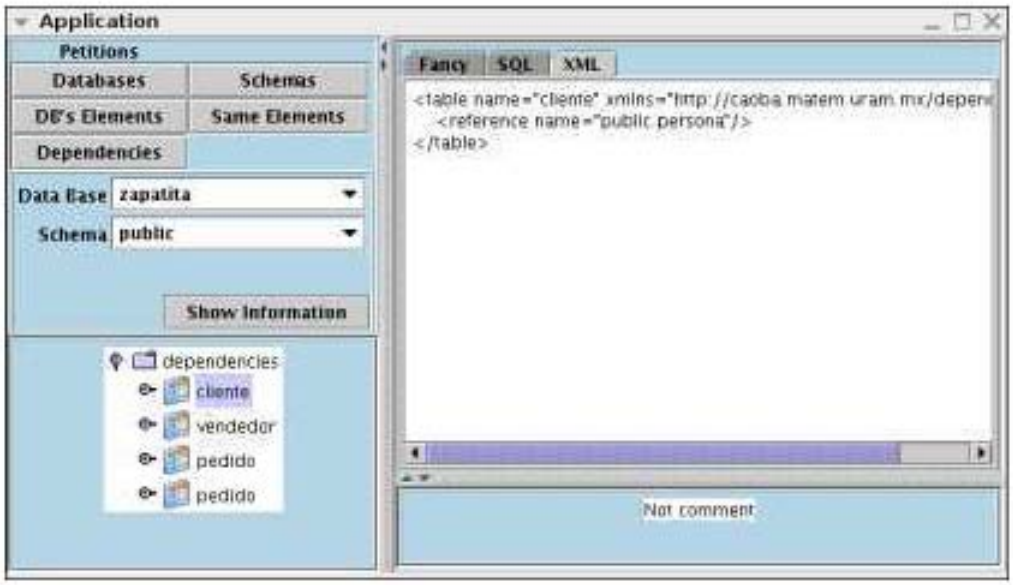

Figura 5.14: Obtención de las dependencias de los elementos de un esquema de una BD.

elementos que estén definidos, por lo que básicamente lo que necesitamos llevar a cabo para resolver nuestra petición es determinar qué elementos necesitamos de *pg_class* para posteriormente empatarlos con los de *pg_depend*.

Es con esto con lo que damos por terminada la descripción del trabajo hecho para esta primera aplicación, haciendo uso de todo lo desarrollado.

Capítulo 6

Modelación basada en objetos

En el capítulo anterior se desarrolló una herramienta cuyo único objetivo era poder obtener información acerca de las BDs existentes dentro de un servidor en particular a través de la navegación dentro de su metainformación, es decir, solamente nos interesábamos en conocer aspectos relacionados con su definición.

Por otro lado, también está el aspecto que tiene que ver en cómo dicha definición de datos puede ser utilizada, ya sea dentro de un SMBD o en alguna aplicación desarollada en algún lenguaje de programación, en particular en uno orientado a objetos, que es precisamente en la cual nos enfocaremos dentro de este capítulo.

6.1. Del modelo relacional al orientado a objetos

Hoy en día, los conceptos de objeto y modelo relacional han sido fuertemente relacionados, esto debido principalmente al planteamiento de si este último puede ser representado por el primero y de qué manera. Entonces, dada una BD, el problema consiste en determinar un conjunto de clases que representen su esquema, para que así, los ejemplares de dichas clases representen a las tuplas contenidas dentro del SMBD.

Básicamente, la necesidad de obtener un conjunto de clases a partir de un conjunto de tablas se debe a la posibilidad de poder interactuar con ellas a otro nivel, es decir, cuando llevamos a cabo un programa, deseamos poder representar todos los elementos involucrados por medio de objetos. Además, basándonos en el modelo orientado a objetos, todo elemento del mundo real es un objeto, en particular las relaciones de las que consta el esquema de una BD, por lo que dicha idea no es ilógica. No obstante, existen corrientes, como en [1], que afirman que aunque es posible hacer dicha representación, esto no es conveniente debido a que el nivel de encapsulamiento de la información no es el mismo.

Para el modelo orientado a objetos, deseamos tener métodos de acceso y asignación (*getters* y *setters*) a través de los cuales podamos obtener o establecer cualquiera de los elementos de los que se compone la clase, es decir, alguno de los atributos que

conformarán una de las tuplas de alguna de las relaciones de nuestra BD; sin embargo, esto, visto al nivel de la BD en un SMBD, no puede llevarse a cabo de la misma manera.

Por otra parte, haciendo una analogía con lo visto en el capítulo 3, recordemos que para generar las clases que permitían la representación de los elementos definidos a través del DDL nos auxiliábamos de XmlBeans, lo cual claramente nos facilitaba la etapa de análisis sintáctico y semántico por medio de la ejecución de una tarea. Otra posible solución para obtener dichas clases era que por nuestra propia cuenta las generáramos, sin embargo, la modificación de alguno de los elementos dentro de la definición del DDL implicaría también la redefinición de una o varias clases, lo cual aparte de ser un posible generador de errores propiciaba un trabajo adicional.

Por lo anterior, el objetivo del trabajo descrito en este capítulo será proporcionar una herramienta por medio de la cual el proceso de obtener el conjunto de clases[1] que representen un modelo de una BD se realice de manera automática.

6.1.1. Obtención de las clases

Ya hemos mencionado que el propósito de esta parte del trabajo será la obtención de clases que representen un esquema de BD; sin embargo, no hemos mencionado cual será nuestra fuente para obtener dicho conjunto. Con tales fines, lo que ocuparemos será un documento en XML que represente dicho esquema y es aquí donde podemos reutilizar una vez mas el trabajo desarrollado hasta este momento de tal manera que podamos obtener tal documento (como se describió en el capítulo 4).

Otro punto que tenemos que mencionar con respecto a la generación de las clases es que sólamente se mapearán los elementos *table* del documento en XML, ya que son estos los que realmente, desde el punto de vista del SMBD, establecen las relaciones que guardan el conjunto de datos de la BD entre sí.

6.1.2. Definición del paquete

Dado un elemento que representa una tabla de una BD, lo primero en lo que tenemos que pensar para generar la clase respectiva es si esta pertenecerá a un paquete en particular. Para determinarlo, nos basaremos en la definición del espacio de nombres del elemento así como del esquema (elemento *schema*) al que pertenece, de tal manera que se puedan tener clases con el mismo nombre, pero no que pertenzcan al mismo esquema.

Así, tomemos como ejemplo el documento del Listado 6.1, en donde el espacio de nombres definido para la tabla *cliente* se encuentra en la línea 3 y el esquema al cual pertenece se definió en la línea 7. Tomando en cuenta todo esto, el paquete al que

[1] Definidas en el lenguaje de programación Java.

pertenecería la clase a la que se mapee la tabla mencionada sería el siguiente:

package mx.unam.matem.caoba.schema_1;

Listado 6.1: Ejemplo de un elemento *table* dentro de un documento XML.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <database xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3            xmlns="http://caoba.matem.unam.mx"
4            xsi:schemaLocation="http://caoba.matem.unam.mx DDL.xsd"
5            name="Example">
6
7  <schema name="schema_1">
8
9    <table name="cliente">
10     ...
11   </table>
12
13   ...
14
15 </database>
```

6.1.3. Definición del nombre de la clase

Ahora, el siguiente paso en la definición de la clase es determinar su firma, que es donde se indica su nombre. Así, continuando con el ejemplo del Listado 6.1, la firma para el elemento *cliente* sería la siguiente:

public class Cliente {

Hay que mencionar que, particularmente en PostgreSQL, una tabla puede ser definida con el nombre equivalente a una cadena de espacios en blanco, que siguiendo la idea establecida en la firma anterior, haría que el nombre de la clase no fuera aceptado por un compilador para *Java*; sin embargo, lo mismo pasaría para tablas cuyos nombres sean dos o cinco espacios respectivamente, siendo que además las tablas en realidad no tendrían el mismo nombre.

Dado lo anterior, se decidió reemplazar cada espacio en blanco por el carácter "_", de tal manera que se logre diferenciar estos casos[2]. Así también, se tomó la decisión de convertir a letra mayúscula el primer carácter del nombre definido dentro del elemento *table* (claro, suponiendo que sea una letra) de tal manera que se siguieran los estándares existentes sobre la definición del nombre de una clase en el lenguaje Java.

[2]Nótese que los nombres de la tablas *ejem tabla* y *ejem_tabla* se mapearan a la misma cadena.

6.1.4. Métodos de acceso y asignación

Los siguientes elementos de nuestro documento XML que hay que reflejar en la definición de la clase son los atributos de los cuales una tabla se compone, que como ya lo hemos mencionado, dicho reflejo se traduce en la incorporación de una variable de clase, además de un método de acceso y otro de asignación por cada uno de los atributos en cuestión.

Así, supongamos que la definición de los atributos de la tabla *cliente* del Listado 6.1 es la que se encuentra en el Listado 6.2.

Listado 6.2: Atributos de los que se compone la tabla *cliente* (definida en el Listado 6.1).

```
1  ...
2
3  <table name="cliente">
4    <attribute name="clave" type="text">
5      <primaryKey />
6      <check language="psql">
7        <definition> length(clave) = 8 </definition>
8      </check>
9    </attribute>
10   <attribute name="cuenta" type="money" />
11   <attribute name="membresia" type="text" />
12   <attribute name="curp" type="text" />
13   <foreignKeyGroup name="cons1" referencesTable="persona">
14     <attribute name="curp" />
15     <attribute name="clave" />
16   </foreignKeyGroup>
17 </table>
18
19 ...
```

Tomemos el primero de sus atributos, es decir, aquel con el nombre *clave* (línea 4). En primer lugar, quisiéramos definir una variable de clase que lo represente, para lo cual es necesario establecer un tipo de acuerdo al lenguaje de programación que estamos utilizando, pero que también vaya acorde al tipo con el que se define dentro de un SMBD. Para este caso, como el atributo se definió con el tipo *text* de acuerdo al sistema manejador, entonces la mejor opción, de acuerdo a los tipos con los que cuenta Java, es que la variable se defina de tipo *String*.

La relación entre los tipos del SMBD y de los que hace uso Java que se utilizará para dar solución al mapeo deseado se muestra en la Tabla 6.1.

Una vez establecida la relación entre los tipos, ya podemos definir la variable de clase y los métodos (de asignación y acceso) que representarán al atributo dentro de la nueva

Tipo SMBD	Tipo Java
varchar, character (varying), char, text	String
bytea	byte[]
smallint	short
integer, serial	int
bigint, bigserial	long
decimal, numeric	java.math.BigDecimal
real	float
double precision	double
date	java.sql.Date
time	java.sql.Time
timestamp	java.sql.Timestamp
boolean	boolean
otro	String

Tabla 6.1: Relación entre los tipos del SMBD y los de Java.

clase. Tales elementos, para el atributo *clave*, se encuentran definidos en el Listado 6.3.

Listado 6.3: Representación del atributo *clave* dentro de una clase.

```
1  // variable de clase
2  String clave;
3
4  /* metodo de asignacion */
5  public void  setClave( String newclave ) {
6      clave = newclave;
7  }
8
9  /* metodo de acceso */
10 public String getClave() {
11     return clave;
12 }
```

Por otro lado, otro elemento que deseamos reflejar es el que tiene que ver con la definición de llaves foráneas dentro de una tabla, lo cual nos conduce a la definición de elementos *foreingnKey* o *foreignKeyGroup* dentro de un elemento *table*.

Para incorporar esto dentro de nuestra clase, lo que haremos será definir una variable de clase más, con sus respectivos métodos de acceso y asignación, por cada uno de los elementos de tipo *foreignKey* o *foreignkKeyGroup* con los que cuente nuestra tabla. El nombre de dichas variables se formará por el prefijo "ref" más el nombre de la tabla a la que se hace referencia, además de que serán del tipo *int*, con lo cual se podrá asignar un

identificador.

Así, siguiendo con el ejemplo definido en el Listado 6.2, este incorpora un elemento de tipo *foreignKeyGroup* en la línea 13, y los elementos necesarios para su incorporación dentro de la clase se muestran en el Listado 6.4.

Listado 6.4: Representación de una llave foránea dentro de una clase.

```
1  // variable de clase
2  int refpersona;
3
4  /* metodo de asignacion */
5  public void setRefpersona( int newrefpersona ) {
6      refpersona = newrefpersona;
7  }
8
9  /* metodo de acceso */
10 public int getRefpersona() {
11     return refpersona;
12 }
```

6.1.5. Clase completa

Con la incorporación de los elementos anteriores, ya tenemos la clase que representa a nuestro elemento *table*.

Siguiendo con nuestro ejemplo de la tabla *cliente*, definida a partir de los listados 6.1 y 6.2, la clase resultante del proceso anterior se muestra en el Listado 6.5.

Listado 6.5: Clase que representa a la tabla *cliente*.

```
1  package mx.unam.matem.caoba.schema_1;
2
3  public class Cliente {
4
5          String clave;
6          String cuenta;
7          String membresia;
8          String curp;
9          int refpersona;
10
11         public void setClave( String newclave ) {
12             clave = newclave;
13         }
14
```

```
15          public String getClave() {
16                  return clave;
17          }
18
19          public void  setCuenta( String newcuenta ) {
20                  cuenta = newcuenta;
21          }
22
23          public String getCuenta() {
24                  return cuenta;
25          }
26
27          public void  setMembresia( String newmembresia ) {
28                  membresia = newmembresia;
29          }
30
31          public String getMembresia() {
32                  return membresia;
33          }
34
35          public void  setCurp( String newcurp ) {
36                  curp = newcurp;
37          }
38
39          public String getCurp() {
40                  return curp;
41          }
42
43          public void  setRefpersona( int newrefpersona ) {
44                  refpersona = newrefpersona;
45          }
46
47          public int getRefpersona() {
48                  return refpersona;
49          }
50  }
```

Con esto damos por terminada la descripción del proceso que planteamos para la obtención de las clases deseadas, y por tanto, de otra aplicación propuesta para nuestro trabajo.

Comentarios Finales

En este trabajo planteamos como punto principal un nuevo lenguaje de definición de datos basado en el modelo relacional, el DDL, por medio del cual se puede representar un esquema de BD y algunas de las operaciones que un SMBD pudiera hacer sobre él. Dicha representación se lleva a cabo de manera sencilla debido a que el DDL da paso a la creación de documentos del tipo XML, los cuales simplemente se forman por etiquetas con determinados atributos. Tal posibilidad de generar estos documentos se debe a que el DDL fue definido a través de XML Schema, el cual a su vez es un lenguaje basado en etiquetas que nos permitió darle la expresividad y flexibilidad descrita en los primeros capítulos de este trabajo a nuestro nuevo lenguaje de datos.

Lo anterior no sólo aportó facilidad y sencillez en cuanto a la representación, sino también a la posibilidad de manejar nuestros esquemas de BD a otro nivel: a través de la red. Es decir, con el uso del DDL se puede generar un documento en XML que represente un esquema de BD y este a su vez puede compartirse a través de la red de tal manera que pueda ser implementado en un SMBD en algún otro sitio distinto. El punto anterior puede también ser apreciado en la descripción de la aplicación descrita en el capítulo 5, en la cual se puede obtener la descripción de una BD remotamente al transmitir un documento XML que la representa.

Hablando un poco sobre los elementos utilizados para el desarrollo de este trabajo, se ocuparon conceptos que tienen que ver con compiladores, así como el uso de gráficas dirigidas y algoritmos sobre estas (*DFS* y *Ordenamiento Topológico*) de tal manera que se obtuviera el resultado deseado.

Trabajo a futuro

Como se mencionó a lo largo del trabajo, se tomó como SMBD base a PostgreSQL y por tanto se definió un apartado en particular para este en el DDL, incorporando algunos elementos que sólo este maneja, con lo cual lográbamos extender nuestro lenguaje base, que es el que maneja elementos estándar con referencia a SQL. Entonces, una buena continuación para nuestro trabajo sería definir algún otro apartado que considere aspectos propios relativos a algún otro SMBD, como lo es Oracle, de tal manera que se pueda obtener un conjunto de instrucciones más acorde a él.

Por último, otro punto a continuar sería utilizar lo ya hecho para implementar una nueva aplicación. Una posible idea podría ser la generación de documentación con un formato en particular de esquemas de BD, ya que recordemos que a través del DDL, era posible incorporar dentro del mismo documento que representa a un esquema anotaciones o comentarios sobre sus elementos.

Apéndice A

La finalidad de este anexo es incluir algunos elementos a los que se hace referencia en los capítulos anteriores de este trabajo, los cuales complementan o recopilan de manera integral a los ya descritos.

A.1. Esquema para el DDL

Listado A.1: Esquema que define al DDL.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3      xmlns="http://caoba.matem.unam.mx"
4      targetNamespace="http://caoba.matem.unam.mx"
5      elementFormDefault="qualified">
6
7  <xs:element name="database" type="Database" />
8  <xs:element name="comment" type="nonEmptyString" />
9
10 <xs:simpleType name="nonEmptyString">
11   <xs:restriction base="xs:string">
12     <xs:minLength value="1" />
13   </xs:restriction>
14 </xs:simpleType>
15
16 <xs:complexType name="Database">
17   <xs:annotation>
18     <xs:documentation xml:lang="en">
19         A database should be formed by schemas.
20     </xs:documentation>
21   </xs:annotation>
22   <xs:sequence>
23     <xs:element ref="comment" minOccurs="0" />
24     <xs:element name="schema" type="Schema"
25                 maxOccurs="unbounded" />
26   </xs:sequence>
27   <xs:attribute name="name" type="nonEmptyString"
```

```
28                  use ="required" />
29 </xs:complexType>
30
31 <xs:complexType name="Schema">
32   <xs:annotation>
33     <xs:documentation xml:lang="en">
34         An schema could be formed by one or more tables,
35         procedures, etc.
36     </xs:documentation>
37   </xs:annotation>
38   <xs:sequence>
39     <xs:element ref="comment" minOccurs="0" />
40     <xs:choice maxOccurs="unbounded">
41       <xs:element name="table" type="Table" minOccurs="0" />
42       <xs:element name="procedure" type="Procedure"
43                   minOccurs="0" />
44       <xs:element name="function" type="Function" minOccurs="0" />
45       <xs:element name="trigger" type="Trigger" minOccurs="0" />
46       <xs:element name="view" type="View" minOccurs="0" />
47       <xs:element name="index" type="Index" minOccurs="0" />
48     </xs:choice>
49   </xs:sequence>
50   <xs:attribute name="name" type="nonEmptyString"
51                 use ="required" />
52 </xs:complexType>
53
54 <xs:complexType name="Table">
55   <xs:sequence>
56     <xs:element ref="comment" minOccurs="0"/>
57     <xs:element name="attribute" type="Attribute"
58                 maxOccurs="unbounded"/>
59     <xs:element name="uniqueGroup" type="UniqueGroup"
60                 minOccurs="0" maxOccurs="unbounded"/>
61     <xs:element name="primaryKeyGroup" type="PrimaryKeyGroup"
62                 minOccurs="0" />
63     <xs:element name="foreignKeyGroup" type="ForeignKeyGroup"
64                 minOccurs="0" maxOccurs="unbounded"/>
65     <xs:element name="check" type="Check" minOccurs="0"
66                 maxOccurs="unbounded" />
67   </xs:sequence>
68   <xs:attribute name="name" type="nonEmptyString"
69                 use="required" />
70 </xs:complexType>
71
72 <xs:complexType name="Attribute">
73   <xs:sequence>
74     <xs:element ref="comment" minOccurs="0" />
75     <xs:element name="defaultValue" type="nonEmptyString"
76                 minOccurs="0" />
```

```
77      <xs:element name="unique" type="Unique" minOccurs="0" />
78      <xs:element name="primaryKey" type="PrimaryKey"
79                  minOccurs="0" />
80      <xs:element name="nillable" type="Nillable" minOccurs="0" />
81      <xs:element name="foreignKey" type="ForeignKey" minOccurs="0"
82                  maxOccurs="unbounded" />
83      <xs:element name="check" type="Check" minOccurs="0"
84                  maxOccurs="unbounded" />
85    </xs:sequence>
86    <xs:attribute name="name" type="nonEmptyString" use="required"/>
87    <xs:attribute name="type" type="nonEmptyString" use="required"/>
88  </xs:complexType>
89
90  <xs:complexType name="Unique">
91    <xs:sequence>
92      <xs:element ref="comment" minOccurs="0" />
93    </xs:sequence>
94    <xs:attribute name="name" type="nonEmptyString" use="optional"/>
95    <xs:attribute name="deferrable" type="xs:boolean"
96                  default="false"/>
97    <xs:attribute name="initially" type="Initially"
98                  default="immediate"/>
99  </xs:complexType>
100
101 <xs:complexType name="PrimaryKey">
102   <xs:sequence>
103     <xs:element ref="comment" minOccurs="0" />
104   </xs:sequence>
105   <xs:attribute name="name" type="nonEmptyString" use="optional"/>
106   <xs:attribute name="deferrable" type="xs:boolean"
107                 default="false"/>
108   <xs:attribute name="initially" type="Initially"
109                 default="immediate"/>
110 </xs:complexType>
111
112 <xs:complexType name="Nillable">
113   <xs:sequence>
114     <xs:element ref="comment" minOccurs="0" />
115   </xs:sequence>
116   <xs:attribute name="name" type="nonEmptyString" use="optional"/>
117   <xs:attribute name="value" type="xs:boolean" default="true"/>
118   <xs:attribute name="deferrable" type="xs:boolean"
119                 default="false"/>
120   <xs:attribute name="initially" type="Initially"
121                 default="immediate"/>
122 </xs:complexType>
123
124 <xs:complexType name="ForeignKey">
125   <xs:sequence>
```

```
126       <xs:element ref="comment" minOccurs="0" />
127     </xs:sequence>
128     <xs:attribute name="name" type="nonEmptyString" use="optional"/>
129     <xs:attribute name="referencesTable" type="nonEmptyString"
130                   use="required"/>
131     <xs:attribute name="referencesAttribute" type="nonEmptyString"
132                   use="optional"/>
133     <xs:attribute name="onDelete" type="Action" default="restrict"/>
134     <xs:attribute name="onUpdate" type="Action" default="restrict"/>
135     <xs:attribute name="deferrable" type="xs:boolean"
136                   default="false"/>
137     <xs:attribute name="initially" type="Initially"
138                   default="immediate"/>
139 </xs:complexType>
140
141 <xs:complexType name="Check">
142     <xs:sequence>
143       <xs:element ref="comment" minOccurs="0" />
144       <xs:element name="definition" type="nonEmptyString" />
145     </xs:sequence>
146     <xs:attribute name="name" type="nonEmptyString" use="optional"/>
147     <xs:attribute name="language" type="nonEmptyString"
148                   use="required"/>
149     <xs:attribute name="deferrable" type="xs:boolean"
150                   default="false"/>
151     <xs:attribute name="initially" type="Initially"
152                   default="immediate"/>
153 </xs:complexType>
154
155 <xs:simpleType name="Action">
156     <xs:restriction base="nonEmptyString">
157       <xs:enumeration value="restrict"/>
158       <xs:enumeration value="cascade"/>
159       <xs:enumeration value="setNull"/>
160       <xs:enumeration value="setDefault"/>
161     </xs:restriction>
162 </xs:simpleType>
163
164 <xs:simpleType name="Initially">
165     <xs:restriction base="nonEmptyString">
166       <xs:enumeration value="deferred"/>
167       <xs:enumeration value="immediate"/>
168     </xs:restriction>
169 </xs:simpleType>
170
171 <xs:complexType name="UniqueGroup">
172     <xs:sequence>
173       <xs:element ref="comment" minOccurs="0"/>
174       <xs:element name="keyElement" type="KeyElement" minOccurs="2"
```

```
175                   maxOccurs="unbounded"/>
176     </xs:sequence>
177     <xs:attribute name="name" type="nonEmptyString" use="optional"/>
178     <xs:attribute name="deferrable" type="xs:boolean"
179                   default="false"/>
180     <xs:attribute name="initially" type="Initially"
181                   default="immediate"/>
182   </xs:complexType>
183
184   <xs:complexType name="KeyElement">
185     <xs:attribute name="name" type="nonEmptyString" use="required"/>
186   </xs:complexType>
187
188   <xs:complexType name="PrimaryKeyGroup">
189     <xs:sequence>
190       <xs:element ref="comment" minOccurs="0"/>
191       <xs:element name="keyElement" type="KeyElement" minOccurs="2"
192                   maxOccurs="unbounded"/>
193     </xs:sequence>
194     <xs:attribute name="name" type="nonEmptyString" use="optional"/>
195     <xs:attribute name="deferrable" type="xs:boolean"
196                   default="false"/>
197     <xs:attribute name="initially" type="Initially"
198                   default="immediate"/>
199   </xs:complexType>
200
201   <xs:complexType name="ForeignKeyGroup">
202     <xs:sequence>
203       <xs:element ref="comment" minOccurs="0"/>
204       <xs:element name="attribute" type="KeyElement" minOccurs="2"
205                   maxOccurs="unbounded"/>
206       <xs:element name="references" type="KeyElement" minOccurs="0"
207                   maxOccurs="unbounded"/>
208     </xs:sequence>
209     <xs:attribute name="name" type="nonEmptyString" use="optional"/>
210     <xs:attribute name="referencesTable" type="nonEmptyString"
211                   use="required"/>
212     <xs:attribute name="onDelete" type="Action" default="restrict"/>
213     <xs:attribute name="onUpdate" type="Action" default="restrict"/>
214     <xs:attribute name="deferrable" type="xs:boolean"
215                   default="false"/>
216     <xs:attribute name="initially" type="Initially"
217                   default="immediate"/>
218   </xs:complexType>
219
220   <xs:complexType name="Procedure">
221     <xs:sequence>
222       <xs:element ref="comment" minOccurs="0"/>
223       <xs:element name="parameter" minOccurs="0"
```

```
224                     maxOccurs="unbounded">
225         <xs:complexType>
226           <xs:attribute name="type" type="nonEmptyString" />
227         </xs:complexType>
228       </xs:element>
229       <xs:element name="source" type="nonEmptyString" />
230     </xs:sequence>
231     <xs:attribute name="name" type="nonEmptyString" use="required"/>
232     <xs:attribute name="language" type="nonEmptyString"
233                     use="required"/>
234   </xs:complexType>
235
236   <xs:complexType name="Function">
237     <xs:sequence>
238       <xs:element ref="comment" minOccurs="0"/>
239       <xs:element name="parameter" minOccurs="0"
240                     maxOccurs="unbounded">
241         <xs:complexType>
242           <xs:attribute name="type" type="nonEmptyString" />
243         </xs:complexType>
244       </xs:element>
245       <xs:element name="returns" type="nonEmptyString"/>
246       <xs:element name="source" type="nonEmptyString"/>
247     </xs:sequence>
248     <xs:attribute name="name" type="nonEmptyString" use="required"/>
249     <xs:attribute name="language" type="nonEmptyString"
250                     use="required"/>
251     <xs:anyAttribute namespace="##other" processContents="lax" />
252   </xs:complexType>
253
254   <xs:complexType name="Trigger">
255     <xs:sequence>
256       <xs:element ref="comment" minOccurs="0"/>
257     </xs:sequence>
258     <xs:attribute name="name" type="nonEmptyString" use="required"/>
259     <xs:attribute name="when" type="When" use="required"/>
260     <!-- el ''on'' del trigger -->
261     <xs:attribute name="target" type="nonEmptyString"
262                     use="required"/>
263     <xs:attribute name="way" type="Way" use="required"/>
264     <!-- el ''execute'' del trigger -->
265     <xs:attribute name="handler" type="nonEmptyString"
266                     use="required"/>
267     <xs:attribute name="onInsert" type="xs:boolean" use="optional"
268                     default="false"/>
269     <xs:attribute name="onUpdate" type="xs:boolean" use="optional"
270                     default="false"/>
271     <xs:attribute name="onDelete" type="xs:boolean" use="optional"
272                     default="false"/>
```

```
273 </xs:complexType>
274
275 <xs:simpleType name="When">
276   <xs:restriction base="nonEmptyString">
277     <xs:enumeration value="before"/>
278     <xs:enumeration value="after"/>
279   </xs:restriction>
280 </xs:simpleType>
281
282 <xs:simpleType name="Way">
283   <xs:restriction base="nonEmptyString">
284     <xs:enumeration value="row"/>
285     <xs:enumeration value="statement"/>
286   </xs:restriction>
287 </xs:simpleType>
288
289 <xs:complexType name="View">
290   <xs:sequence>
291     <xs:element ref="comment" minOccurs="0" />
292     <xs:element name="statement" type="nonEmptyString" />
293   </xs:sequence>
294   <xs:attribute name="name" type="nonEmptyString" use="required"/>
295   <xs:attribute name="language" type="nonEmptyString"
296                 use="required"/>
297 </xs:complexType>
298
299 <xs:complexType name="Index">
300   <xs:sequence>
301     <xs:element ref="comment" minOccurs="0" />
302     <xs:choice>
303       <xs:element name="column" type="nonEmptyString"/>
304       <xs:element name="columnExpression" type="nonEmptyString"/>
305     </xs:choice>
306     <xs:element name="where" type="nonEmptyString" minOccurs="0"/>
307   </xs:sequence>
308   <xs:attribute name="name" type="nonEmptyString" use="required"/>
309   <xs:attribute name="target" type="nonEmptyString"
310                 use="required" />
311   <xs:attribute name="unique" type="xs:boolean" use="optional"
312                 default="false"/>
313   <xs:anyAttribute namespace="##other" processContents="lax" />
314 </xs:complexType>
315
316 </xs:schema>
```

A.2. Máquinas de estados de los elementos de una BD de acuerdo al DDL

A.2.1. Procedimientos

La máquina de estados que representa a este elemento dentro del DDL en el Listado A.1 (línea 220) se muestra en la Figura A.1.

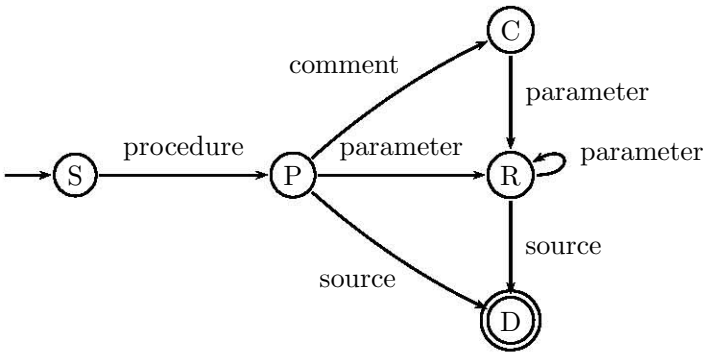

Figura A.1: Máquina de estados para el elemento *procedure*.

A.2.2. Funciones

Por otra parte, la máquina para el elemento *function* (Listado A.1, línea 236) se muestra en la Figura A.2.

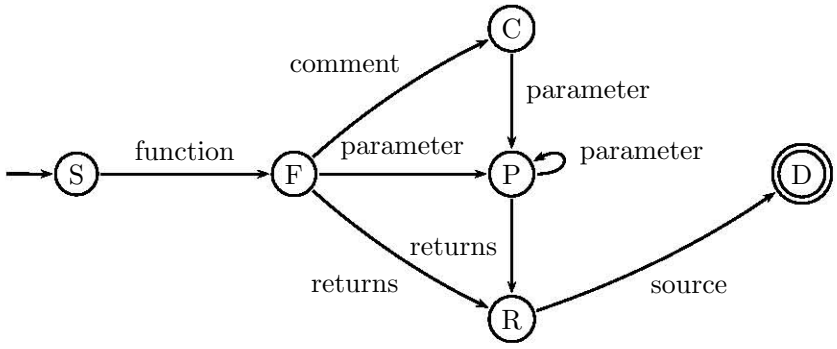

Figura A.2: Máquina de estados para el elemento *function*.

A.2.3. Triggers

En lo que respecta al elemento *trigger* definido en la línea 254 del Listado A.1, su máquina correspondiente se encuentra en la Figura A.3.

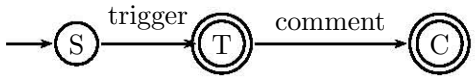

Figura A.3: Máquina de estados para el elemento *trigger*.

A.2.4. Vistas

Este elemento fue definido para el DDL en el Listado A.1 (línea 289) y su máquina de estados se muestra en la Figura A.4.

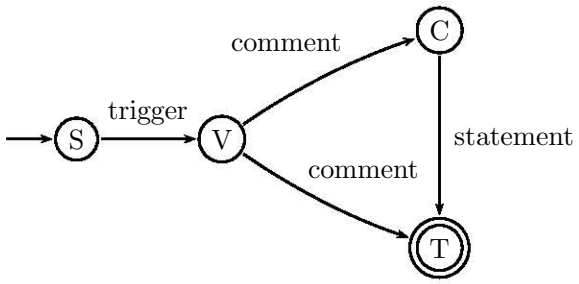

Figura A.4: Máquina de estados para el elemento *view*.

A.2.5. Índices

El último elemento definido para el DDL en el Listado A.1 (línea 299) fué el elemento *index*, cuya máquina de estados se representa en la Figura A.5.

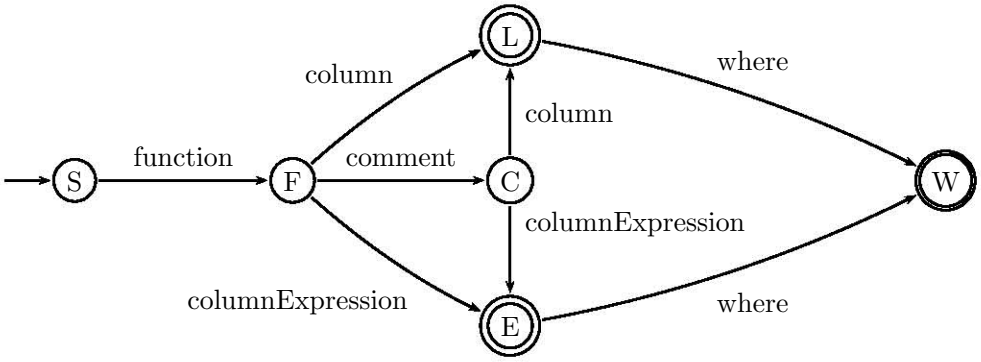

Figura A.5: Máquina de estados para el elemento *index*.

Referencias

[1] S. W. Ambler. *The Object-Relational Impedance Mismatch*, 2002-2004. `www.agiledata.org/essays/impedanceMismatch.html`.

[2] *Cafe con Leche XML News and Resources*, 1998-2006. `www.ibiblio.org/xml/`.

[3] E. Codd. *The Relational Model for Database Management*. Addison Wesley, 1990.

[4] C. Date. *An Introduction to Database Systems*. Addison Wesley, 8th edition, 2003.

[5] H. García-Molina, J. D. Ullman, and J. D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 1st edition, 2001.

[6] E. R. Harold. *XML Bible, Gold Edition*. Hungry Minds, 2001.

[7] E. R. Harold. *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP AND TrAX*. Addison Wesley Professional, 2002.

[8] *PostgreSQL 8.0.6 Documentation*, 1996-2005. `www.postgresql.org/docs/8.0/interactive/index.html`.

[9] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, 4th edition, 2001.

[10] E. Viso and S. Rajsbaum. *Notas de Análisis de Algoritmos*, 1999.

[11] *Apache XMLBeans*, 2004. `xmlbeans.apache.org`.

[12] *Apache XMLBeans 2.0 API Specification*, 2005. `xmlbeans.apache.org/docs/2.0.0/reference/index.html`.

[13] *Apache XML-RPC*, 2001-2005. `ws.apache.org/xmlrpc/index.html`.

[14] *XML-RPC Home Page*. `www.xmlrpc.com`.

[15] *XML Schema*, 2000-2003. `www.w3.org/XML/Schema`.

[16] *XML Schema Part 0: Primer*, 2004. `www.w3.org/TR/xmlschema-0/`.

[17] *XML Schema Part 1: Structures*, 2004. `www.w3.org/TR/xmlschema-1/`.

[18] *XML Schema Part2: Datatypes*, 2004. `www.w3.org/TR/xmlschema-2/`.