3

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

UN LENGUAJE DE CONSULTA PARA BASES DE DATOS SEMIESTRUCTURADOS

T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACION

PRESENTA

EGAR ARTURO GARCIA CARDENAS

FACULTAD DE CIENCIAS
UNAM

DIRECTORA DE TESIS:
DRA. AMPARO LOPEZ GAONA

FACULTAD DE CIENCIAS
SECCION ESCOLAR 2002

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.

M. EN C. ELENA DE OTEYZA DE OTEYZA
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

Un lenguaje de consulta para bases de datos semiestructurados.

realizado por Egar Arturo García Cárdenas

con número de cuenta 9429700-5, quién cubrió los créditos de la carrera de Ciencias de la Computación.
Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	Dra. Amparo López Gaona	
Propietario	Mat. Salvador López Mendoza	
Propietario	Dr. Francisco Hernández Quiroz	
Suplente	Dra. Hanna Oktaba	
Suplente	M. en C. Ma. Guadalupe E. Ibargüengoitia González	

Consejo Departamental de Matemáticas.

Dra. Amparo López Gaona

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

A mi mamá, Ma. del Consuelo Cardenas Villordo.

A mi papá, Salomón García Salinas.

Agradecimientos

Llegando a este punto es momento de agradecer a aquellas personas que han aportado algo significativo en mi vida y en mi trayectoria académica, de alguna manera una parte de este trabajo también es de ellos.

En primer lugar quiero agradecer a mis padres: Consuelo y Salomón; por todo su apoyo, comprensión, dedicación y paciencia; por todo lo que me han brindado; por su cariño incondicional en todo momento; y bueno, todo lo que pueda decir sería insuficiente, así que con esta tesis quisiera brindarles un pequeño homenaje. También quiero agradecer a mi hermano Eric por su compañía y apoyo todos estos años.

Un agradecimiento especial a mi asesora de tesis Amparo y a su esposo Salvador, ambos maestros y amigos, por todo su apoyo y múltiples consejos no solo en lo académico sino en muchos aspectos de la vida. En particular agradezco a Amparo por su confianza y por todas lo que ha aportado en mi vida.

Gracias a mis sinodales Salvador, Francisco, Lupita y Hanna por sus exhaustivas revisiones y correcciones; a Salvador por sugerir el nombre del lenguaje de consulta que es el tema principal de esta tesis.

Agradezco a todos aquellos maestros que me han brindado parte de sus conocimientos y de su confianza a lo largo de mi vida, en especial a aquellos que recuerdo con especial cariño: A la maestra Martita en el jardín de niños; a las maestras Rosita, María Luisa y Alicia en la primaria, a Vianey y Patricia en la secundaria; a Ismael, Chucho, Elizabeth y Eva Lidia en la prepa; a Amparo, Salvador, Patricia Duran, Eliza, Flor de María, Carmen y Luis Colavita en la facultad; al profesor Jaime Escamilla del Karate y a mi amigo e instructor de gimnasio Victorino (Vic).

Agradezco a los amigos por compartir momentos felices, momentos tristes y un sin fin de aventuras: A mi amigo desde el jardín de niños Andrés; a mis amigos de la primaria Cesar y Daniel; a mis amigos de la secundaria Erick y Francisco Mastache; a mis amigos de la prepa Victor, Mariel, Jassia, Ana, Celeste, Oscar (Brandom), Sandra, Agustín, Nancy, Marcela, Israel (El Groovy), Quetza, Alfredo, Isaid Basilio,

Azalea, Jessica, Silvia, Alejandra, Berenice, Diana, Omar, Gretel, Yadira, Socorro e Isabel; a mis amigos de la facultad Alejandra, Oscar, Dorian, Ewi, Gustavo, Reina, Nayeli, Alexei, Mireya, Oscar, Jerónimo, Dulce, Claudia, Ivette, Melisa, Cesar, Marisol, Selene, Rosita, Oscar Ruíz, Gustavo, Canek, Edgar, Rafael, Erick, Karina, Citlali, Liliana, Maribel, Lucio, Leif, Julio, y Karla. En especial quiero agradecer a Alexei, Mariel, Jassia, Yadira, Karla y Alejandra por ser los mejores y verdaderos amigos que alguien puede tener.

El amor es una parte primordial de la vida, quizá más que el alimento, así que no puedo dejar de agradecer a las mujeres que (en su momento) amé: Gisela, Olivia, Yazmin y Ana Patricia. Si bien me hicieron pasar momentos tormentosos, también me dierón toda la inspiración que necesitaba y un poco mas.

A mi numerosa familia por estar siempre ahí. A mis abuelos: Alfonso y Gullermina y los que ya no están: Macario y Consuelo. A mis tíos y tías: Bertha, María, Herlinda, Eduardo, Alfonso, Silvia, Teresa, Socorro, Catalina y Adolfo. A mis primos y primas: Teresa (Tete), Beto, Felix (El Gato), Silvia, Jessica, Sofia, Ernesto, Alejandro, Angel, Barbara, Fabian, Edibaldo, Lupita, Sara, Guillermina (La Dona), Sabino y Macario. A mis tíos políticos Florencio (El Checho), Baldemar (El Guali) y a todos los demás...

Tal vez esté fuera de lugar pero también quisiera dar la gracias al siempre fiel compañero de toda la vida, con el cual he vivido un sin fin de aventuras: El auto de mi familia, la camioneta Fairmont 82 roja, la máquina a la que más cariño le tengo.

Y en general gracias a todos aquellos que encontré a lo largo de este camino y que brindarón algo a mi vida.

A la UNAM, en especial a la Facultad de Ciencias, agradezco todos los conocimientos y satisfacciones que me ha brindado a lo largo de esta etapa de mi formación profesional.

A todos, un millón de gracias.

Índice general

Índice de figuras

Índice de cuadros

Introducción

En una gran cantidad de dominios, los problemas no se deben a una falta de información, sino que existe demasiada, en constante cambio, con una estructura irregular y dispersa en varias fuentes. La Web es un claro ejemplo de ello.

En la Web se puede encontrar todo tipo de información y representada de distintas formas, como imágenes, música, documentos, animaciones, etc. En los documentos existentes en la Web resulta común encontrar referencias a otras fuentes de información en el mismo u otro documento. El lenguaje predominante en la Web es HTML, este lenguaje ha ido creciendo de manera acelerada, acarreando con ello el surgimiento de una serie de problemas. Por otra parte, XML es un metalenguaje que ha surgido para resolver varios de los problemas que se presentan con el uso de HTML. Alrededor de XML existe gran variedad de tecnologías que facilitan el tratamiento, intercambio y presentación de la información. Se pretende que dentro de algún tiempo XML sea el formato predominante en la Web y otros muchos dominios.

La diferencia entre un documento y un almacén de datos, es que los almacenes de datos poseen mecanismos para elegir una parte de la información de acuerdo a las necesidades de quien la requiera, así como mecanismos para modificar la información ya existente. En los documentos la información permanece casi estática. La mayor parte de la información en la Web se utiliza como documentos, pero para algunos propósitos es útil considerarla como almacenes de datos, por ejemplo al realizar busquedas de algún tema.

Un lenguaje de consulta marca la diferencia entre documento y almacén de datos, el lenguaje de consulta incluye mecanismos para el manejo de la información. En la actualidad muchas aplicaciones utilizan XML como un formato para el almacenamiento de datos, lo que ha traído el surgimiento de algunos lenguajes de consulta para XML.

En muchas aplicaciones, se requiere que la información utilizada posea una flexibilidad enorme, es decir, que se pueda manejar información faltante, así como la incorporación de nueva que no se esperaba previamente. El uso de bases de datos

relacionales u orientadas a objetos para manejar este tipo de información, implicaría gran cantidad de dificultades en el diseño y el almacenamiento de los datos, debido a su naturaleza rigida.

El modelo de datos semiestructurados, así como sus bases de datos, se ha venido desarrollando con la intención de tomar los conocimientos que se tienen acerca de bases de datos tradicionales para aplicarlos a información que no posee una estructura fija.

La información en la Web y en muchos otros dominios se puede modelar con datos semiestructurados. El tener un lenguaje de consulta consistente y poderoso para el manejo de datos semiestructurados, traería muchos beneficios respecto al tratamiento de la información en estos dominios.

Para estudiar las propiedades de los datos semiestructurados y definir claramente un lenguaje de consulta para ellos, es necesario ordenar y dar formalidad al concepto de dato semiestructurado. Un formalismo matemático cumple con este propósito, así se plasma la idea intuitiva de este concepto en términos de construcciones formales.

Los objetivos de este trabajo son: Proponer un formalismo matemático para el modelo de datos semiestructurados que se apegue a la naturaleza e idea intuitiva de dicho concepto y diseñar un lenguaje de consulta para bases de datos semiestructurados que sea claro, consistente y con alto poder expresivo.

Para lograr los objetivos, este trabajo se divide en cinco capítulos. En el capítulo 1 se exponen los conceptos de dato semiestructurado y de base de datos semiestructurados. En el capítulo 2 se presenta un formalismo para los datos semiestructurados basado en teoría de conjuntos, se describe un formalismo basado en gráficas y se muestra la equivalencia entre ambos. En el capítulo 3 se describe el lenguaje *Ssquirel*, que es el lenguaje de consulta para bases de datos semiestructurados que se propone. En el capítulo 4 se exponen las principales características de XML y las tecnologías a su alrededor. En el capítulo 5 se explican las similitudes existentes entre XML y los datos semiestructurados, para luego analizar algunos de los lenguajes de consulta más conocidos para estos tipos de información.

La información que se encuentra contenida en las siguientes páginas se obtuvo de libros, artículos, manuales, documentos en la Web y muchas horas de reflexión e inspiración. Los conocimientos que se manejan tienen que ver con una gran variedad de disciplinas como bases de datos, lenguajes formales, teoría de conjuntos, teoría de gráficas, lógica matemática, estructuras de datos, etc.

Capítulo 1

El modelo de datos semiestructurados

Para una gran variedad de problemas se requiere hacer uso de información con una estructura irregular, esta información puede sufrir un crecimiento imprevisto y debe tolerar la falta de alguna información. Los datos semiestructurados se han venido desarrollando como una alternativa para menejar información con este tipo de caracteristicas.

El uso de modelos relacionales y orientados a objetos para tratar información de tipo no rígido presenta ciertas limitaciones. Por ejemplo si se usan bases de datos relacionales podría requerirse que a cada rato se agregaran nuevos campos a las tablas, que se utizara una gran cantidad de valores nulos, o en su defecto la elaboración de un diseño demasiado complejo; esto puede provocar el desperdicio de espacio de almacenamiento, redundancia de información y multiples dificultades en su manejo.

1.1. El concepto de dato semiestructurado

Frecuentemente cuando se tiene la necesidad de representar información lo primero que se hace es definir una estructura para los datos y después se crean instancias para esa estructura, es decir existe una separación entre la definición de la estructura o tipo del dato y su valor. En los *datos semiestructurados* dicha separación no existe, la estructura del dato se describe junto con su valor.

Los *datos semiestructurados* son autodescriptivos, sin embargo, se tiene que asumir la existencia de ciertos *tipos primitivos de datos* como cadenas de caracteres, números,

imágenes, fechas, horas, etc.

Intuitivamente un *dato semiestructurado* puede ser: Un dato de tipo primitivo o un conjunto finito de parejas formadas por una *etiqueta* y un dato semiestructurado. Una *etiqueta* es simplemente un nombre que se le da a un dato semiestructurado para referirse a él dentro de otro. Una pareja formada por una etiqueta y un dato semiestructurado se dice que es un *dato semiestructurado etiquetado.*

Un dato semiestructurado puede contener un mismo dato semiestructurado etiquetado de distintas maneras o varios datos semiestructurados etiquetados con una misma etiqueta. Cabe mencionar que el conjuto vacío es un dato semiestructurado, esto se puede ver por vacuidad.

En concreto un dato semiestructurado es:

- Un dato de tipo primitivo.
- Un conjunto finito de datos semiestructurados etiquetados.

En un conjunto de datos semiestructurados cada uno de sus miembros se debe identificar de manera única, para este proposito se usan los *identificadores*, a un *identificador* le corresponde un dato semiestructurado y cada dato semiestructurado tiene un identificador único. De esta manera un dato semiestructurado está dado de manera única por su identificador. En algunas ocaciones los identificadores no se hacen explicitos, sin embargo se asume su existencia.

Los identificadores se pueden ver como nombres globales y únicos, mientras que las etiquetas se pueden ver como nombres locales con posible repetición.

En los datos semiestructurados se puede establecer una jerarquía de parentesco. Dado un dato semiestructurado A y uno B, se puede decir que A es *padre* de B o bien que B es *hijo* de de A, si A contiene a B. Un dato semiestructurado C es *ancestro* de D si C es padre de D o si existen $C_1 ... C_k$ con $k > 0$ tales que C_{i+1} es padre de C_i para $i = 1, ..., k-1$, C es padre de C_k y C_1 es padre de D. Similarmente D es *descendiente* de C si C es ancestro de D.

Una *familia de datos semiestructurados* es una colección de datos semiestructurados tal que existe un miembro que es ancestro de todos los demás de la colección, a éste se le conoce como *raíz*, notese que una raíz de una familia de datos semiestructurados no necesariamente es única.

Como se puede ver los datos semiestructurados son muy flexibles en cuanto a la estructura, las contenciones entre ellos se pueden anidar a una profundidad arbitraria. Para aclarar la idea de dato semiestructurado veanse los ejemplos 1.2.1, 1.2.2 y 1.2.3.

1.2. Sintaxis

Los datos semiestructurados pueden ser descritos usando una sintaxis simple. La que aquí se presenta es muy común y resulta muy familiar a los programadores de Lisp, la idea es describir listas asociativas etiqueta-valor [1].

La especificación de la sintaxis en BNF es la siguiente:

```
<ssd-expr> ::= <ssd> | &<oid> <ssd> | &<oid>
<ssd> ::= <primitivo> | <no-primitivo>
<no-primitivo> ::= { <lista-ssds> }
<lista-ssds> ::= <etiqueta>: <ssd-expr> |
                 <etiqueta>: <ssd-expr>, <lista-ssds>
<oid> ::= [a..zA..Z1..9_]+
<etiqueta> ::= [a..zA..Z1..9_]+
```

El símbolo <primitivo> toma la sintaxis usada para representar datos de tipo primitivo como enteros, números de punto flotante, cadenas de caracteres, según los tipos de datos primitivos que se asuman.

La descripción de un dato semiestructurado está dada por el símbolo <ssd-expr>, a ésta se la llama *ssd-expresión*. El símbolo <oid> se refiere a un identificador que debe ser único para cada dato, <ssd> representa el contenido de un dato semiestructurado que puede ser un dato primitivo o un conjunto de datos semiestructurados etiquetados, <no-primitivo> representa un conjunto de datos semiestructurados etiquetados cuyo contenido se da en <lista-ssds>.

Un identificador o se dice que está *definido* en una ssd-expresión s si s es de la forma $\&o\ v$ para algún v o si s es de la forma $\{l_1 : e_1, ..., l_n : e_n\}$ y o está *definido* en una de las ssd-expresiones $e_1, ..., e_n$. Se dice que un identificador o es una *referencia* en una ssd-expresión s si s es de la forma $\&o$ o si s es de la forma $l_1 : e_1, ..., l_n : e_n$ y o es una *referencia* en una de las ssd-expresiones $e_1, ..., e_n$.

Una ssd-expresión s es consistente si y sólo si:

- Todo identificador está definido a lo más una vez en s.
- Todo identificador que es una referencia en s está definido en s.

Para que un dato semiestructurado esté bien descrito a través de una ssd-expresión s se requiere que s sea consistente.

Con las referencias se representan anidamientos arbitrarios y se evita describir más de una vez un mismo dato semiestructurado. Basta con definir el identificador de un dato semiestructurado una sola vez y luego si aparece en otro lugar solamente se hace una referencia a él por medio de dicho identificador. Aunque no se requiere que para cada dato se defina un identificador, es útil pensar que lo tiene aunque éste no se haga explícito.

Con la sintaxis definida anteriormente se describe una familia de datos semiestructurados empezando por una raíz.

Ejemplo 1.2.1 Se describe un dato semiestructurado para representar la información de algunos países, el dato semiestructurado consta a su vez de tres datos semiestructurados etiquetados como país, éstos a su vez contienen otros datos primitivos que representan la información del país como el nombre, la capital, la moneda y el idioma. Obsérvese como un dato semiestructurado puede contener varios con la misma etiqueta.

```
{
  pais: { nombre:  "México",
          capital: "Cd. de México",
          moneda:  "Peso",
          idioma:  "Español",
        },
  pais: { nombre:  "España",
          capital: "Madrid",
          moneda:  "Peseta",
          moneda:  "Euro",
          idioma:  "Español",
        },
  pais: { nombre:  "Canadá",
          capital: "Otawa",
          moneda:  "Dólar canadiense",
          idioma:  "Inglés",
          idioma:  "Francés",
        }
}
```

Ejemplo 1.2.2 El siguiente dato semiestructurado consta de otros datos semiestructurados que representan lenguajes o paradigmas de lenguajes de programación, los cuales a su vez contienen datos primitivos que representan lenguajes que concuerdan con el paradigma o contiene otros datos semiestructurados que corresponden a subparadigmas. Obsérvese como la información que contienen los datos semiestructurados es muy variable de uno a otro.

```
{ oo: {lenguaje: "SmallTalk",
       lenguaje: "Java",
       mixtos: { lenguaje: "C++",
                 lenguaje: "Object Pascal"}
      },
  logico: { lenguaje: "Prolog" },
  funcional: { listas: { lenguaje: "Lisp",
                         lenguaje: "Scheme"},
               lenguaje: "Haskell"
             },
  lenguaje: "Ensamblador"
}
```

Ejemplo 1.2.3 En el siguiente dato semiestructurado se representa la relación de parentesco entre cuatro personas. Obsérvese como las personas y sus identificadores se definen una sola vez, las relaciones de parentesco se definen por medio de los identificadores, así no se requiere definir más de una vez cada persona.

```
{ persona:&o1 { nombre:"Pedro" },
  persona:&o2 { nombre:"Maria" },
  persona:&o3 { nombre: "Jose",
                padre: &o1,
                madre: &o2,
                hijo: &o4 },
  persona:&o4 { nombre:"Luis",
                padre:&o3,
                abuelo:&o1 }
}
```

1.3. Representación con gráficas

Es muy común y resulta de gran utilidad representar los datos semiestructurados por medio de gráficas. Ello da una visión esquemática de un dato semiestructurado, que en la mayoría de las ocasiones resulta más cómoda que la descripción con la sintaxis anterior.

Los datos semiestructurados se representan utilizando gráficas dirigidas con aristas etiquetadas [1]. Cada dato semiestructurado corresponde a un único vértice de la gráfica, los datos primitivos corresponden a hojas (vértices que no tienen aristas de salida), a los vértices que corresponden a datos primitivos se les asocia el valor del dato primitivo. Si es explicito, el identificador de un dato semiestructurado usualmente se coloca en su vértice correspondiente.

Una arista (v_1, v_2) está en la gráfica si y sólo si el dato a quien representa v_1 es padre del dato a quien representa v_2, la etiqueta de la arista (v_1, v_2) será la etiqueta que corresponde al dato representado por $v2$ dentro del dato representado por v_1. Se debe notar que la gráfica resultante es conexa y tiene raíz, es decir, un vértice del cual existe un camino dirigido a todos los demás la gráfica. La gráfica resultante no necesariamente es un árbol, puede tener ciclos.

Ejemplo 1.3.1 En las figuras 1.1, 1.2 y 1.3 se muestran las gráficas de los datos semiestructurados descritos en los ejemplos 1.2.1, 1.2.2 y 1.2.3 respectivamente.

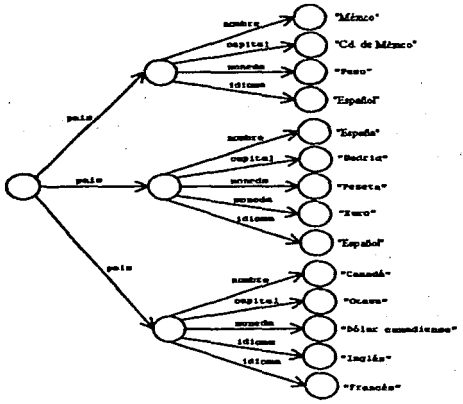


Figura 1.1: Representación gráfica del dato semiestructurado del ejemplo 1.2.1.

1.4. Bases de datos semiestructurados

El concepto de *base de datos* consta de tres partes que son: El modelo para representar los datos, la colección de datos y el sistema manejador.

El *modelo de datos* es una abstracción de las caracteristicas de los elementos que componen la información. Consta de una serie de propiedades que poseen los elementos de información y posiblemente operaciones que se pueden aplicar a ellos. Por ejemplo, las bases de datos relacionales utilizan el modelo relacional que consta de tablas o relaciones [20, 11].

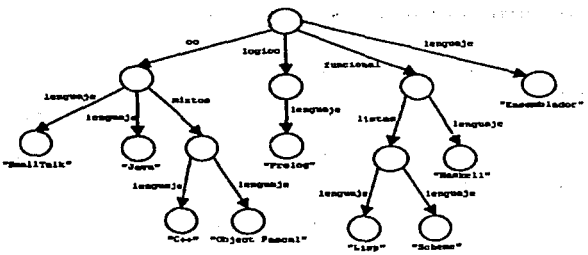


Figura 1.2: Representación gráfica del dato semiestructurado del ejemplo 1.2.2.

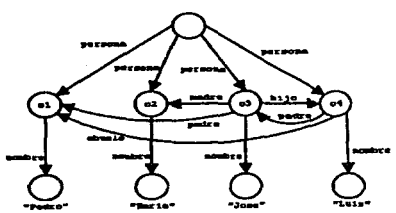


Figura 1.3: Representación gráfica del dato semiestructurado del ejemplo 1.2.3.

La *colección de datos* es la información requerida por los usuarios, consiste de datos interrelacionados que siguen el modelo de datos tomado.

El *sistema manejador* consiste de un conjunto de programas que permiten la representación, almacenamiento, acceso y manipulación de la colección de datos [20].

Si el modelo que se sigue para representar los datos está basado en el modelo de datos semiestructurados, entonces se trata de una *base de datos semiestructurados.*

En esta sección se propone un modelo para la descripción y construcción de bases de datos semiestructurados. Este modelo tiene el propósito de especificar claramente la manera de organizar la información que se maneja en las bases de datos de este tipo.

1.4.1. Descomposición en ssd-tablas

Una *base de datos semiestructurados* es una colección de datos semiestructurados. Es necesario contar con algunos datos semiestructurados distinguidos, a través de los cuales se pueda acceder a todos los demás. Una *ssd-tabla* es simplemente un dato semiestructurado distinguido mediante un nombre o etiqueta. Este dato semiestructurado es la raíz o el ancestro de otros datos semiestructurados, los cuales pueden ser accesados o modificados a través de la *ssd-tabla*.

Una base de datos semiestructurados se puede considerar como una colección de ssd-tablas, cuyos nombres, por supuesto, no se pueden repetir.

Las ssd-tablas en una base de datos semiestructurados pueden compartir datos con otras ssd-tablas. Incluso se puede dar el caso que un dato semiestructurado esté en más de una ssd-tabla.

Todos los datos semiestructurados en una base de datos semiestructurados deben tener identificadores distintos, si los datos son distintos. No se permite que dos datos distintos tengan el mismo identificador aunque pertenezcan a ssd-tablas distintas.

Las ssd-tablas proporcionan un punto de partida para la manipulación de la información en la base de datos.

Ejemplo 1.4.1 En la figura 1.4 se muestra una base de datos semiestructurados. Ésta consta de tres ssd-tablas llamadas `cursos`, `profesores` y `publicaciones`. Estas ssd-tablas se refieren a los datos semiestructurados con identificadores $c1$, $m1$ y $p1$ respectivamente, estos datos semiestructurados son la raíz de otros más. Obsérvese como las ssd-tablas pueden compartir datos y todos los datos semiestructurados en la base de datos tienen un identificador distinto.

1.4.2. Redundancia y falta de información

Al construir bases de datos relacionales lo primero que se hace es utilizar el modelo entidad-relación para modelar la información que se va a representar. En esta fase se describen los objetos o entidades de las cuales se va a componer la información y la relación que existe entre ellas, esto se trabaja de manera conceptual. Posteriormente, del modelo entidad-relación se pasa al modelo relacional para el almacenamiento de dicha información, la información se almacena en relaciones o tablas y la información referente a un tipo de entidades se reparte en una o más tablas. De esta manera es posible que exista información redundante en las tablas. Para evitar la redundancia las relaciones tienen que pasar por procesos de normalización. En resumen, el diseño

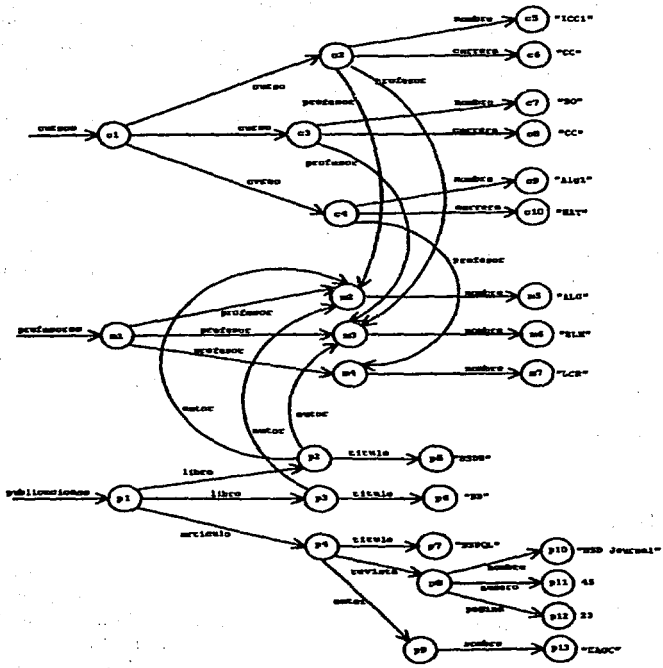


Figura 1.4: Base de datos semiestructurados.

de una base de datos relacional consta de dos niveles, en uno se describen las entidades y sus relaciones y en otro la forma en que se almacenan.

En el modelo de datos semiestructurados los objetos o entidades de las cuales consta la información son precisamente datos semiestructurados, las relaciones entre ellos son relaciones de contención con etiquetas, así que la forma de describir las entidades también da un modelo del almacenamiento de los datos. En las bases de datos semiestructurados un objeto es único y describe su propia información, por lo cual en el modelo de datos semiestructurados la redundancia de la información no es un problema importante.

En las bases de datos relacionales la falta de información corresponde a la presencia de valores nulos en las tablas, en la mayoría de los casos el permitir el uso de valores nulos en las tablas tiene el propósito de simplificar el diseño de la base de datos, el impedir que los valores nulos aparezcan puede complicar tremendamente el diseño debido a que las entidades poseen una estructura rígida. Sin embargo el uso de valores nulos tiene varios problemas como son espacio desperdiciado y la introducción de una lógica trivalente.

Los datos semiestructurados son flexibles en cuanto a la estructura, un dato semiestructurado puede contener un número arbitrario de datos semiestructurados inclusive con la misma etiqueta, así que si algo se requiere simplemente se incluye y si no se requiere no se incluye, de esta manera no es necesario el uso de los valores nulos y con la implementación adecuada se evita el desperdicio del espacio que éstos producen.

El conjunto vacío es un dato semiestructurado y éste se puede pensar como un valor nulo, pero como éste puede tener una etiqueta es posible que represente la presencia de algo, de forma similar a una bandera o como el uso de los elementos vacíos en XML o HTML. También el conjunto vacío puede corresponder a un dato que espera que se le agreguen otros datos. En conclusión, la presencia del conjunto vacío no significa una falta de información.

1.4.3. Vistas

En bases de datos relacionales, la idea de *vista* es asociada con una *tabla virtual.* Dicha tabla virtual tiene el propósito de restringir la información que se presenta a los usuarios. Una vista se construye a partir de una fórmula aplicada a la base de datos, a la vista se le asocia un nombre como si se tratara de otra tabla. Si la base de datos sufre modificaciones, éstas se deben reflejar en la vista, es por ello que la mayoría de los sistemas de bases de datos guardan la fórmula o *definición de la vista,* en vez del resultado de su evaluación, por lo tanto la vista se calcula cada vez que se requiere. Algunos otros sistemas permiten que la vista ya calculada sea la que se guarde, a estas se les denomina *vistas materializadas*; sin embargo con las vistas materializadas resulta difícil que la vista refleje los cambios que se hacen en la base de datos, aunque se gana en tiempo de respuesta [20].

Si un sistema permite modificaciones a las vistas, estas modificaciones se deben traducir a modificaciones en la base de datos, lo cual trae consigo algunos problemas significativos. Es por ello que la mayoría de los sistemas no permiten modificaciones a las vistas, ya que un cambio en una vista puede afectar a varias tablas.

En el modelo de bases de datos semiestructurados que aquí se propone, se consi-

deran dos tipos de vistas: Las *vistas abstractas* y las *vistas materializadas*, también se asume que las vistas no se pueden modificar.

Se define una *vista* de una base de datos semiestructurados como una *ssd-tabla virtual* que se construye a partir de una fórmula o *definición de vista* que se aplica a la base de datos semiestructurados.

Las *vistas abstractas* son las que se calculan cada vez que se hace uso de ellas. Si la base de datos sufre cambios, al recalcular la vista se verán reflejados los cambios.

Las *vistas materializadas* son las que se calculan una sola vez y se guarda su estructura, cada vez que se usa una vista de este tipo se usa la estructura ya creada. En la estructura de la vista se mantienen una serie de relaciones entre datos semiestructurados de la base de datos. Un cambio en la base de datos semiestructurados se ve como una serie de modificaciones a sus datos semiestructurados, así que estos cambios se ven directamente reflejados en la estructura de la vista. Debido a la naturaleza de los datos semiestructurados la implementación de este tipo de vistas resulta más sencilla que en las bases de datos relacionales.

En la sección 3.4.3 se verá como definir vistas en una base de datos semiestructurados.

Ejemplo 1.4.2 Considérese la base de datos semiestructurados que se muestra en la figura 1.5, la cual consta de una sola ssd-tabla llamada `profs`. Supongase que la formula o definición de una vista consiste en elegir el dato semiestructurado etiquetado como `nombre` de los datos semiestructurados etiquetados como `maestro`. En la figura 1.6 se muestran una vista abstracta en la parte (a) llamada `v1` y una vista materializada en la parte (b) llamada `v2`, ambas obtenidas a partir de dicha fórmula, estas dos vistas tienen la misma estructura.

Supongase que la ssd-tabla `profs` se modifica de tal manera que la etiqueta `profesor` se remplaza por la etiqueta `maestro` y el dato semiestructurado con identificador $a4$ es eliminado, esto se muestra en la figura 1.7. Si la vista abstracta dada por la formula anterior se utiliza después de este cambio, queda como en la figura 1.8(a), ya que ésta se evalúa de nuevo. La vista materializada queda como en la figura 1.8(b), ésta no se vuelve a calcular. Como el dato semiestructurado con identificador $a9$ ha sido eliminada con los cambios a la tabla, este dato también se elimina de la vista.

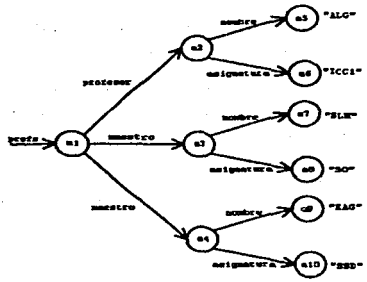


Figura 1.5: Base de datos semiestructurados del ejemplo 1.4.2.

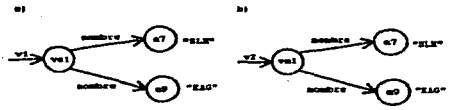


Figura 1.6: Vistas de la base de datos semiestructurados de la figura 1.5.

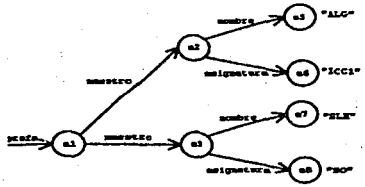


Figura 1.7: Base de datos semiestructurados modificada del ejemplo 1.4.2.

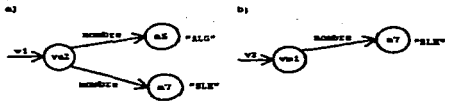


Figura 1.8: Vistas de la base de datos semiestructurados de la figura 1.7.

Capítulo 2

Un formalismo para los datos semiestructurados

Los formalismos matemáticos se utilizan para plasmar la idea intuitiva de un concepto en términos de construcciones formales. De esta manera las propiedades del modelo intuitivo se pueden estudiar a través de una representación sintáctica de éste.

Sin un formalismo matemático un concepto solamente existe en la imaginación y está sujeto a la interpretación particular que cada quien tenga de él, pudiendo en algunos casos variar enormemente de una persona a otra. Un formalismo matemático toma la esencia de las propiedades de un concepto intuitivo, para transformarla en construcciones formales. De esta manera, la interpretación que cada individuo tenga del concepto estará guiada por su especificación en términos formales.

Los formalismos existentes para los datos semiestructurados se basan en árboles[10], gráficas dirigidas [2, 8, 18, 21] o en gramáticas libres de contexto [1, 16]. Las gramáticas libres de contexto simplemente describen la sintaxis.

El concepto de dato semiestructurado explicado en el capítulo anterior se construye a partir del concepto intuitivo que se tiene de conjuntos, con algunas variantes. En este capítulo se propone un formalismo para los datos semiestructurados basado en la teoría de conjuntos tradicional. Así se apega más a la idea intuitiva y a la naturaleza de los datos semiestructurados que los otros formalismos. También se presenta un formalismo basado en gráficas y su equivalencia con el formalismo basado en conjuntos. El uso de gráficas es de suma utilidad para dar una representación visual o esquemática de los datos semiestructurados.

El formalizar los datos semiestructurados permite estudiar de manera organizada y sistematizada las propiedades de éstos, lo cual se puede reflejar en el desarrollo de una

gran variedad de aplicaciones útiles en el tratamiento e intercambio de información.

Los formalismos matemáticos, en particular uno para datos semiestructurados, trae consigo una serie de problemas como es el caso de la existencia de indecidibles. Sin embargo, esto no impide que su utilización sea de gran utilidad para fines prácticos.

2.1. Un modelo formal para los datos semiestructurados

Intuitivamente un dato semiestructurado es un dato primitivo o un conjunto cuyos elementos son otros datos semiestructurados etiquetados, posiblemente incluyéndose el mismo, uno que lo contenga, uno que contenga a otro que lo contenga, etc. Se asume que cada dato semiestructurado tiene un identificador único.

Para iniciar con la formalización se define el concepto de *D-familia*. Una *D-familia* se puede ver como una colección de datos, estos pueden ser primitivos o no-primitivos, un dato no-primitivo se representa como una triada que consta de una etiqueta, un identificador y un dato dado por el identificador.

Definición 1 Dada $S = \langle P, I, L, D, R, T \rangle$. S es una *D-familia* si y sólo si:

- P es un conjunto finito de conjuntos no vacíos.
- I es un conjunto finito no vacío.
- L es un conjunto finito.
- D es un conjunto finito de conjuntos.
- R es una función
$$I \rightarrow D \cup \bigcup_{p \in P} p$$
tal que, $R(R^{-1}(D)) = D$.
- Para todo $d \in D$, y para todo $x \in d$ existen $i \in I$ y $l \in L$ tales que $x = (l, i, R(i))$.
- T es un subconjunto de I, tal que:
$$R(T) \subseteq \bigcup_{p \in P} p$$
y
$$R(I - T) = D$$

P representa el conjunto de tipos de datos primitivos usados, los cuales se pueden construir utilizando la teoría de conjuntos tradicional, por ejemplo números naturales, enteros, reales, cadenas de caracteres (que se pueden ver como secuencias finitas de números naturales), etc. I representa el conjunto de identificadores. L es el conjunto de etiquetas. D representa un conjunto de datos no-primitivos. La función R asocia un dato con cada identificador. T representa los identificadores para los datos primitivos.

En una *D-familia* es posible tener un orden jerárquico, basado en la forma en que están contenidos los datos, de esta manera se tienen padres, hijos, ancestros, descendientes y es posible tener una raíz la cual es un ancestro común para todos. La siguiente definición formaliza estos conceptos de acuerdo a la definición de *D-familia*. El parentesco se establece por medio de los identificadores, ya que por medio de estos es como se caracteriza un dato.

Definición 2 Dada $S = \langle P, I, L, D, R, T\rangle$, una *D-familia*, y dados $i, j \in I$:

- i es padre de j si y sólo si $R(i) \in D$ y existe $l \in L$ tal que $(l, j, R(j)) \in R(i)$.
- i es un hijo de j si y sólo si j es padre de i.
- i es un ancestro de j si y sólo si existen $i_1, ..., i_n \in img(R)$, con $n \geq 2$ tales que $i_1 = i$, $i_n = j$ y para cada $k = 1, ..., n-1$ i_k es padre de i_{k+1}.
- i es descendiente de j si y sólo si j es ancestro de i.
- i es una raíz de S si y sólo si para todo $k \in I - \{i\}$, i es ancestro de k.

Ejemplo 2.1.1 Para aclarar la idea de *D-familia* se presenta este ejemplo. Sean:

$$
\begin{aligned}
P &= \{\mathbf{N}\} \\
I &= \{1, 2, ..., 11\} \\
L &= \{a, b, c\} \\
D &= \{A_1, A_2, A_3, A_4, A_5\} \\
A_1 &= \{(a, 3, A_4), (b, 4, 0)\} \\
A_2 &= \{(a, 5, A_3), (a, 6, A_5)\} \\
A_3 &= \{(b, 7, A_2), (c, 8, 8)\} \\
A_4 &= \{(a, 9, 0)\} \\
A_5 &= \{(a, 10, A_3), (b, 11, 5)\} \\
R &= \{(1, A_1), (2, A_2), (3, A_4), (4, 0), (5, A_3) \\
&\quad (6, A_5), (7, A_2), (8, 8), (9, 0), (10, A_3), (11, 5)\} \\
T &= \{4, 8, 9, 11\}
\end{aligned}
$$

$\langle P, I, L, D, R, T\rangle$ es una *D-familia*. En este caso 1 es padre de 4 y 2 es padre de 5.

La definición de *D-familia* permite la existencia de grupos de datos aislados, es decir, grupos de datos que no tienen parentesco con otros datos de la misma *D-familia*.

En la noción intuitiva de dato semiestructurado todos los miembros de una familia tienen un ancestro común, de esta manera, existe una relación de parentesco entre cada par de miembros de la familia, o dicho de otra manera, no existen grupos aislados. La siguiente definición refleja esta noción. Se toman las *D-familias* que tengan una raíz o ancestro común a todos los miembros, a éstas se les llama *SSD-familias*.

Definición 3 Una *D-familia* $S = \langle P, I, L, D, R, T\rangle$ es una *SSD-familia* si y sólo si existe $r \in I$ tal que r es raíz de S.

De esta manera, un *dato semiestructurado* puede ser definido como un miembro de una *SSD-familia*. Desde este punto solamente se trabajará con *SSD-familias*.

Definición 4 Dada $S = \langle P, I, L, D, R, T\rangle$, una *SSD-familia*, se dice que s es un *dato semiestructurado* si y sólo si $s \in img(R)$.

Ejemplo 2.1.2 El presente ejemplo muestra cómo el conjunto vacío es un dato semiestructurado.
Dados $P = \{\mathbf{N}\}$, $I = \{1\}$, $L = \phi$, $D = \{\phi\}$.
Si $R = \{(1, \phi)\}$ y $T = \phi$ entonces $\langle P, I, L, D, R, T\rangle$ es una *SSD-familia*. En éste ejemplo 1 es la raíz.

Ejemplo 2.1.3 Aquí se muestra un caso más práctico:
Sean:

$$
\begin{aligned}
P &= \{\text{Strings}, \mathbf{N}\} \\
I &= \{1, ..., 8\} \\
L &= \{\text{maestro}, \text{nombre}, \text{materia}\} \\
D &= \{A, B, C\} \\
A &= \{(\text{maestro}, 2, B), (\text{maestro}, 3, C)\} \\
B &= \{(\text{nombre}, 4, \text{“ALG”}), (\text{materia}, 5, \text{“BD”})\} \\
C &= \{(\text{nombre}, 6, \text{“SLM”}), (\text{materia}, 7, \text{“ICC1”}), (\text{materia}, 8, \text{“SO”})\} \\
R &= \{(1, A), (2, B), (3, C), (4, \text{“ALG”}), (5, \text{“BD”}), \\
&\quad (6, \text{“SLM”}), (7, \text{“ICC1”}), (8, \text{“SO”})\} \\
T &= \{4, 5, 6, 7, 8\}
\end{aligned}
$$

$\langle P, I, L, D, R, T\rangle$ es una *SSD-familia* con 1 como raíz.

2.2. Datos semiestructurados como gráficas

La definición siguiente formaliza el concepto de *SSD-gráfica*. Esta definición puede ser usada para dar una definición alternativa de dato semiestructurado a través de gráficas. El concepto de gráfica dirigida con raíz se extiende para tener etiquetas en las aristas y datos primitivos en algunas hojas.

Definición 5 $G = \langle V, E, P, L, tag, H, val\rangle$ es una *SSD-gráfica* si y sólo si

- V es un conjunto finito no vacío.
- $E \subseteq V \times V$.
- La gráfica dirigida (V, E) tiene una raíz.
  Es decir, existe $r \in V$ tal que para todo $v \in V - \{r\}$ existen $v_1, ..., v_n \in V$, con $n \geq 2$ tales que, $v_1 = r$, $v_n = v$ y para todo $i = 1, ..., n-1$, $(v_i, v_{i+1}) \in E$.
- P es un conjunto de conjuntos no vacíos.
- L es un conjunto no vacío.
- tag es una función $E \rightarrow \mathcal{P}(L)$ tal que $|tag(e)| \geq 1$ para todo $e \in E$.
- H es un subconjunto de $hojas(V, E)$.
  Donde $hojas(V, E) = \{v \in V \mid \neg\exists y(y \in V \wedge (v, y) \in E)\}$.
- val es una función
$$val : H \rightarrow \bigcup_{p \in P} p$$

Los conjuntos V y E corresponden a los vértices y aristas de la gráfica dirigida, P es el conjunto de tipos primitivos de datos, L es el conjunto de etiquetas, tag es la función que asocia etiquetas a las aristas, H es el conjunto de hojas correspondientes a datos primitivos y val es la función que asocia datos primitivos a las hojas en H.

La siguiente definición introduce el concepto de *SSD-gráfica* asociada a una *SSD-familia*. En la gráfica cada vértice es un dato semiestructurado y cada padre tiene aristas hacia sus hijos.

Definición 6 Dada $S = \langle P, I, L, D, R, T\rangle$, una *SSD-familia*, se dice que $G = \langle V, E, P, L, tag, H, val\rangle$ es una *SSD-gráfica asociada a* S si y sólo si

- Existe una función biyectiva $f : I \rightarrow V$ tal que, para todo $i, j \in I$, $l \in L$, $(l, j, R(j)) \in R(i)$ si y sólo si $(f(i), f(j)) \in E$ y $l \in tag((f(i), f(j)))$.

- $f(T) = H$.
- Para todo $u \in H$, $val(u) = R(f^{-1}(u))$.

Ejemplo 2.2.1 En la figura 2.1 se muestra un ejemplo de una *SSD-gráfica* asociada con la *SSD-familia* del ejemplo 2.1.3.

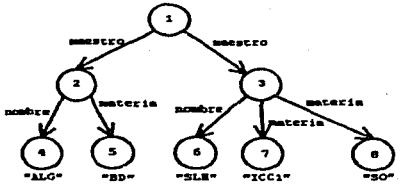


Figura 2.1: SSD-gráfica asociada a la SSD-familia del ejemplo 2.1.3.

La definición que sigue establece formalmente el isomorfismo de dos *SSD-gráficas*. Intuitivamente significa que las gráficas son iguales con excepción de sus vértices, es decir, las gráficas tienen la misma forma y representan exactamente lo mismo. El teorema que sigue a la definición establece que las *SSD-gráficas* asociadas a una misma *SSD-familia* son isomorfas.

Definición 7 Dadas $G = \langle V, E, P, L, tag, H, val\rangle$ y $G' = \langle V', E', P', L', tag', H', val'\rangle$ *SSD-gráficas*, se dice que G y G' son *isomorfas* si y sólo si

- Existe una función biyectiva $f : V \to V'$.
- $P = P'$.
- $L = L'$.
- Para todo $u, v \in V$, $(u, v) \in E$ si y sólo si $(f(u), f(v)) \in E'$.
- $f(H) = H'$
- Para todo $v \in H$, $val(v) = val'(f(v))$.
- Para todo $u, v \in V$, si $(u, v) \in E$ entonces $tag(u, v) = tag'(f(u), f(v))$.

Teorema 1 Dada $S = \langle P, I, L, D, R, T\rangle$ una *SSD-familia*, si $G = \langle V, E, P, L, tag, H, val\rangle$ y $G' = \langle V', E', P, L, tag', H', val'\rangle$ son *SSD-gráficas asociadas a* S entonces G y G' son isomorfas.

Demostración Como G y G' son gráficas asociadas a S, entonces, existen funciones biyectivas $f : I \to V$ y $f' : I \to V'$ tales que:

$$\begin{aligned} &\forall i \forall j \forall l (i, j \in I \wedge l \in L \Rightarrow \\ &\quad (l, j, R(j)) \in R(i) \Leftrightarrow (f(i), f(j)) \in E \wedge l \in tag(f(i), f(j))) \quad (2.1) \\ &\forall i \forall j \forall l (i, j \in I \wedge l \in L \Rightarrow \\ &\quad (l, j, R(j)) \in R(i) \Leftrightarrow (f'(i), f'(j)) \in E' \wedge l \in tag'(f'(i), f'(j))) \quad (2.2) \end{aligned}$$

y también

$$\forall u (u \in H \Rightarrow val(u) = R(f^{-1}(u))) \quad (2.3)$$

$$\forall u (u \in H' \Rightarrow val'(u) = R(f'^{-1}(u))) \quad (2.4)$$

1. De lo anterior, se construye una función biyectiva $f_1 : V \to V'$, donde $f_1 = f' \circ f^{-1}$.
2. Por definición $f(T) = H$ y $f'(T) = H'$. Como f es biyectiva, $f^{-1}(H) = T$. De aquí que:
$$f_1(H) = f'(f^{-1}(H)) = f'(T) = H' \quad (2.5)$$
3. Ahora, dados $u, v \in V$. Supóngase que $(u, v) \in E$.
Se tiene que para todo $l \in tag(u, v)$, $(l, f^{-1}(v), R(f^{-1}(v))) \in R(f^{-1}(u))$, por 2.1.
De 2.2, $(f'(f^{-1}(u)), f'(f^{-1}(v))) \in E'$.
Como $f_1 = f' \circ f^{-1}$ se tiene que $(f_1(u), f_1(v)) \in E'$.
Se ha probado que si $(u, v) \in E$ entonces $(f_1(u), f_1(v)) \in E'$. De manera similar se prueba la contrapuesta, es decir, si $(f_1(u), f_1(v)) \in E'$ entonces $(u, v) \in E$. Teniendo esto, se puede concluir que $(u, v) \in E$ si y sólo si $(f_1(u), f_1(v)) \in E'$.
4. De 2.5 se puede concluir que $w \in H$ si y sólo si $f_1(w) \in H'$. Luego, dado $w \in H$, de 2.3 y 2.4 se tiene que:
$$\begin{aligned} val(w) &= R(f^{-1}(w)) \\ &= R((f'^{-1} \circ f') \circ f^{-1}(w)) \\ &= R(f'^{-1}(f' \circ f^{-1}(w))) \\ &= R(f'^{-1}(f_1(w))) \\ &= val'(f_1(w)) \end{aligned}$$
5. Para concluir la demostración, dados $x, y \in V$ tales que $(x, y) \in E$. Dado $l \in L$ tal que $l \in tag(x, y)$. Entonces
$$\begin{aligned} R(f^{-1}(x)) &= R((f'^{-1} \circ f') \circ f^{-1}(x)) \\ &= R(f'^{-1}(f' \circ f^{-1}(x))) \\ &= R(f'^{-1}(f_1(x))) \end{aligned}$$

y

$$\begin{aligned} R(f^{-1}(x)) &\ni (l, f^{-1}(y), R(f^{-1}(y))) \\ &= (l, (f'^{-1} \circ f')(f^{-1}(y)), R((f'^{-1} \circ f')(f^{-1}(y)))) \\ &= (l, f'^{-1}(f' \circ f^{-1}(y)), R(f'^{-1}(f' \circ f^{-1}(y)))) \\ &= (l, f'^{-1}(f_1(y)), R(f'^{-1}(f_1(y)))) \end{aligned}$$

esto significa que

$$(l, f'^{-1}(f_1(y)), R(f'^{-1}(f_1(y)))) \in R(f'^{-1}(f_1(x)))$$

de 2.2

$$(f'(f'^{-1}(f_1(x))), f'(f'^{-1}(f_1(y)))) \in E'$$

y

$$\begin{aligned} l &\in tag'(f'(f'^{-1}(f_1(x))), f'(f'^{-1}(f_1(y)))) \\ &= tag'(f_1(x), f_1(y)) \end{aligned}$$

Se ha probado que si $l \in tag(x, y)$ entonces $l \in tag'(f_1(x), f_1(y))$, de manera similar se puede probar el inverso. De ésta manera se concluye que $tag(x, y) = tag'(f_1(x), f_1(y))$

De lo anterior se concluye que G y G' son *isomorfas.* □

De las *SSD-gráficas* asociadas a una *SSD-familia* dada, hay una que tiene una importancia significativa: la *gráfica equivalente*, en esta gráfica el conjunto de identificadores de la *SSD-familia* se toma como el conjunto de vértices de la *SSD-gráfica*, así, la jerarquía de los datos en la *SSD-familia* es exactamente reflejada en la gráfica de acuerdo a los identificadores.

Definición 8 Dada $S = \langle P, I, L, D, R, T \rangle$ una *SSD-familia*, se dice que $G = \langle I, E, P, L, tag, H, val \rangle$ es una *SSD-gráfica equivalente a* S si y sólo si

- Para todo $i, j \in I$, $l \in L$, $(l, j, R(j)) \in R(i)$ si y sólo si $(i, j) \in E$ y $l \in tag(i, j)$.
- $T = H$.
- Para todo $u \in H$, $val(u) = R(u)$.

Es facil verificar que dada una *SSD-familia* existe una *SSD-gráfica* equivalente a ella, esto es lo que dice el siguiente teorema cuya demostración se puede obtener facilmente a partir de la definición anterior, simplemente el conjunto de vértices de la *SSD-gráfica* es el conjunto de identificadores de la *SSD-familia* y las aristas se construyen de tal manera que reflejen el parentesco entre ellos.

Teorema 2 Dada $S = \langle P, I, L, D, R, T\rangle$ una *SSD-familia*, existe G una *SSD-gráfica equivalente* a S.

El siguiente teorema establece que si dos *SSD-gráficas* son equivalentes a una misma *SSD-familia* entonces son exactamente la misma gráfica, esto quiere decir que la *SSD-gráfica* equivalente a una *SSD-familia* es única.

Teorema 3 Dada $S = \langle P, I, L, D, R, T\rangle$ una *SSD-familia*, si $G = \langle I, E, P, L, tag, H, val\rangle$ y $G' = \langle I, E', P, L, tag', H', val'\rangle$ son *SSD-gráficas equivalentes a* S entonces

- $E = E'$.
- $tag = tag'$
- $H = H'$
- $val = val'$

Demostración Como G y G' son *SSD-gráficas equivalentes a* S, entonces:

$$\begin{aligned}
\forall i\forall j\forall l\,(i,j \in I \wedge l \in L \;&\Rightarrow \\
(l, j, R(j)) \in R(i) \;&\Leftrightarrow\; (i,j) \in E \wedge\; l \in tag(i,j)) && (2.6)\\
\forall i(i \in H \;&\Rightarrow\; val(i) = R(i)) && (2.7)\\
\forall i\forall j\forall l(i,j \in I \wedge l \in L \;&\Rightarrow \\
(l, j, R(j)) \in R(i) \;&\Leftrightarrow\; (i,j) \in E' \wedge\; l \in tag'(i,j)) && (2.8)\\
\forall i(i \in H' \;&\Rightarrow\; val'(i) = R(i)) && (2.9)
\end{aligned}$$

1. Dado $e \in E$ con $e = (u, v)$ donde $u, v \in I$ y dado $l \in L$, tal que, $l \in tag(e)$. De 2.6 $(l, v, R(v)) \in R(u)$ y de 2.8 $(u, v) \in E'$, de lo cual se concluye que $e \in E'$. Por lo tanto $E \subseteq E'$. De manera similar se prueba que $E' \subseteq E$ y por ello $E = E'$.
2. Dado $e \in E$ con $e = (u, v)$ donde $u, v \in I$ y dado $l \in L$.
   Supóngase que $l \in tag(e)$, de 2.6, $(l, v, R(v)) \in R(u)$ y de 2.8 se concluye que $l \in tag'(e)$.
   Ahora, supóngase que $l \in tag'(e)$, de 2.8 $(l, v, R(v)) \in R(u)$ y de 2.6 se concluye que $l \in tag(e)$. Esto significa que $tag(e) = tag'(e)$.
   Con esto se puede concluir que $tag' = tag$.
3. Por definición $T = H$ y $T = H'$, por lo tanto $H = H'$.

4. Finalmente, dado $u \in H$, como $H = H'$, por 2.7 y 2.9 se tiene que:

$$\begin{aligned} val(u) &= R(u) \\ &= val'(u) \end{aligned}$$

de donde se puede concluir que $val = val'$.

De lo anterior se concluye que G y G' son *equivalentes a* S. □

2.3. Existencia de SSD-familias

Hasta este momento no se ha probado la existencia de las SSD-familias, y desde luego, tampoco la existencia de los datos semiestructurados. La teoría de conjuntos permite la creación de nuevos conjuntos a través de conjuntos previamente construidos, la teoría de conjuntos no puede ser usada para probar la existencia de conjuntos mutuamente recursivos [14, 28], como es el caso de los datos semiestructurados.

Para librar esta limitación, a la teoría de conjuntos de Zermelo-Fraenker con axioma de elección (ZFC) y suponiendo que no se cuenta con el axioma de regularidad o fundación [17, 28], se le agregará un nuevo axioma que garantizará la existencia de las *SSD-familias*. Las gráficas se pueden construir usando la teoría de conjuntos tradicional y muchas veces son usadas para representar estructuras mutuamente recursivas, de esta manera, las *SSD-gráficas* servirán de apoyo para la construcción de *SSD-familias* y así eliminar la limitación. El axioma es el siguiente:

Axioma SSD Dada $G = \langle V, E, P, L, tag, H, val \rangle$ una *SSD-gráfica*, existe una única *SSD-familia* $S = \langle P, V, L, D, T, R \rangle$ tal que G es una *SSD-Gráfica equivalente* a S.

La unicidad es asumida como axioma debido al hecho de que la teoría de conjuntos no posee mecanismos para probar la igualdad entre conjuntos mutuamente recursivos.

Gracias al axioma anterior y a los teorema que establecen la existencia y unicidad de la *SSD-gráfica* equivalente a una *SSD-familia*, se puede trabajar indistintamente con la *SSD-gráfica* o con la *SSD-familia*. Así, se tiene una completitud en el sentido de que toda *SSD-gráfica* define una *SSD-familia* y viceversa.

Existen otras formas de definir conjuntos mutuamente recursivos, como son el uso de la coinducción y los hiperconjuntos (conjuntos que permiten contenciones a profundidades arbitrarias) [14]. La introducción de un nuevo axioma crea el problema de

verificar la consistencia del sistema, sin embargo, el axioma anterior resulta un teorema en la teoría de hiperconjuntos, se ha demostrado que la teoria de hiperconjuntos es consistente si lo es la teoria de conjuntos tradicional (ZFC), sin embargo, no existe un resultado que garantice la consistencia de ZFC.

La razón por la que se eligió la introducción de un nuevo axioma en lugar de utilizar hiperconjuntos, es que para el propósito de este trabajo basta con garantizar la existencia de las *SSD-familias*, la teoría de hiperconjuntos permite construir estructuras más generales pero la axiomatización que requiere tiene una mayor complejidad conceptual.

Capítulo 3

Un lenguaje de consulta para bases de datos semiestructurados

Un lenguaje de consulta da a los programadores[1] un medio para manejar la información contenida en una base de datos, el lenguaje debe proveer mecanismos para recuperar la información de la base de datos, así como mecanismos para modificar, crear y destruir información.

En el diseño de un lenguaje de consulta se tienen que considerar algunos aspectos como son:

- Poder expresivo. Se refiere a la capacidad del lenguaje para realizar lo que el programador quiere o necesita hacer con la información.
- Semántica clara y consistente. La semántica tiene que estar claramente especificada y sin ambigüedades en las construcciones del lenguaje y debe estar claramente reflejada por la sintaxis.
- Capacidad de composición. Es la cualidad del lenguaje de consulta que permite que ciertas expresiones en el lenguaje puedan ser usadas para construir otras, en las cuales su significado está dado en función del significado de suspartes.
- Sintaxis sencilla. Es deseable que el lenguaje de consulta sea sencillo de aprender y comprender para los programadores. Para ello, las construcciones del lenguaje no deben ser demasiadas ni demasiado complicadas. Además deben resultar intuitivas para el programador.

[1] En este contexto el programador se refiere al usuario del lenguaje

En este capitulo se propone el lenguaje llamado *Ssquirel*[2] , que es un lenguaje de consulta para bases de datos semiestructurados como las que se describen en la sección 1.4. La idea es sentar la bases para que en un futuro surja un estándar de un lenguaje de datos semiestructurados, que debe tener el nombre *SSDQL* (el uso de este nombre es demasiado pretencioso en este momento).

La sintaxis de *Ssquirel* tiene cierta semejanza con la sintaxis de SQL y OQL, esto es con el propósito de que resulte sencillo de aprender y comprender para los programadores, el significado de las construcciones se puede intuir con el conocimiento previo de estos lenguajes. La semántica está diseñada especialmente para el manejo de datos semiestructurados, no se basa en un modelo relacional o uno orientado a objetos.

Ssquirel posee construcciones para realizar consultas, modificar la información existente en la base de datos semiestructurados, crear y destruir ssd-tablas y vistas.

3.1. Expresiones de camino

Una de las principales características que debe tener un lenguaje de consulta para datos semiestructurados es la habilidad para explorar los datos que se encuentran a una profundidad arbitraria, para este propósito se usan las *expresiones de camino.*

3.1.1. Expresiones de camino simples.

Una expresión de camino es una secuencia finita de etiquetas $l_0.l_1.\ldots.l_n$ en donde l_0 es la etiqueta de una ssd-tabla y $l_1,\ldots,l_n$ son etiquetas para los datos en dicha ssd-tabla.

El resultado de una expresión de camino $l_0.l_1.\ldots.l_n$ es el conjunto de datos semiestructurados S_n tales que existen datos semiestructurados $S_0,...,S_{n-1}$ tales que el dato semiestructurado etiquetado $l_1 : S_1$ está en S_0, $l_2 : S_2$ está en S_1, ..., $l_{n-1} : S_{n-1}$ está en S_n y S_0 es la raíz de la ssd-tabla dada por l_0. Considerando la representación gráfica para datos semiestructurados, una expresión de camino $l_0.l_1.\ldots.l_n$ aplicada a la ssd-tabla T con nombre l_0 es el conjunto de nodos v_n en la gráfica de T tales que existen nodos $v_0, v_1, \ldots, v_{n-1}$ donde v_0 es la raíz distinguida de T, v_{i-1} es padre de v_i

[2] *Ssquirel* se deriva de *Semistructured data query and information retrieval language.* También se deriva de la palabra inglesa squirrel cuyo significado en español es ardilla. Esto es porque la ardilla se la pasa recorriendo los árboles y arbustos de un bosque, con los cuales se puede establecer una analogía respecto a los datos semiestructurados

para $i = 1, \ldots, n$ y l_i es la etiqueta de la arista (v_{i-1}, v_i) para $i = 1, \ldots, n$.

Una expresión de camino se puede considerar como una pequeña consulta que regresa un conjunto de datos semiestructurados, las expresiones de camino no dan la capacidad para construir nuevos datos semiestructurados, pero proporcionan un mecanismo para explorar datos semiestructurados a una profundidad arbitraria, mismo que servirá de base para las construcciones del lenguaje de consulta.

Ejemplo 3.1.1 Considérese la base de datos semiestructurados de la figura 3.1. La expresión de camino `publicaciones.libro.titulo` regresa el conjunto formado por los datos semiestructurados con identificadores $p5$ y $p6$. La expresión de camino `publicaciones.libro.autor` regresa el conjunto formado por los datos semiestructurados con identificadores $m2$ y $m3$. La expresión `profesores.profesor.nombre` solamente regresa el dato semiestructurado con identificador $m7$.

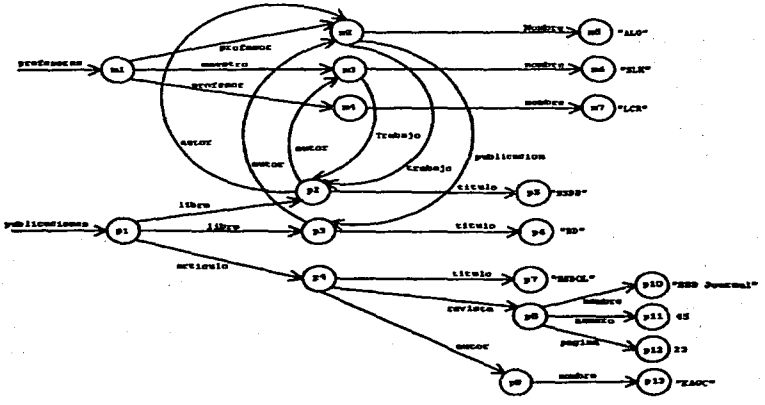


Figura 3.1: Base de datos semiestructurados del ejemplo 3.1.1.

3.1.2. Expresiones de camino extendidas

Para dar mayor flexibilidad a las expresiones de camino se usan expresiones regulares en dos niveles. El primer nivel consiste en el uso de expresiones regulares donde el conjunto de etiquetas se toma como alfabeto. El segundo nivel consiste en

el uso de expresiones regulares tomando como alfabeto el conjunto de caracteres que componen las etiquetas (letras y dígitos). Otra forma de dar flexibilidad es por medio del uso de un comodín el cual puede ser sustituido por cualquier etiqueta o cualquier carácter dependiendo del nivel en el que se esté usando dicho comodín en la expresión; el símbolo # es el que sera usado como comodín. Las *expresiones de camino extendidas* son las que hacen uso de estas características. El uso de expresiones de camino extendidas permite que en una sola expresión se exprese lo que requerirían varias expresiones de camino simples.

Ejemplo 3.1.2 Considérese nuevamente la base de datos semiestructurados de la figura 3.1. La expresión de camino `publicaciones.#.autor` regresa el conjunto formado por los datos semiestructurados con identificadores $m2$, $m3$ y $p9$.

La sintaxis general de una expresión de camino con expresiones regulares es:

```
<e> ::= <etiqueta> |
        <e>|<e>    |
        <e>.<e>    |
        <e>*       |
        <e>+       |
        <e>?       |
        (<e>)      |
```

El operador · sirve para establecer las secuencias de etiquetas en una expresión de camino. El operador * (estrella de Kleene) indica la presencia de cero o más ocurrencias de la expresión dada, el operador + indica la presencia de una o más ocurrencias y el operador ? indica la precencia de una o cero ocurrencias de la expresión dada. El operador | indica que debe aparecer una u otra de las expresiones dadas. Los operadores *, + y ? son los que tienen mayor precedencia, le sigue el operador · y finalmente el operador |.

Cuando la gráfica contiene ciclos, es posible especificar caminos de longitud arbitraria. Utilizando la estrella de Kleene con el comodín se puede generar un número infinito de expresiones de camino. Sin embargo, como la gráfica es finita, el número de datos semiestructurados que la expresión de camino extendida devolverá es también finito.

Ejemplo 3.1.3 Una vez más considérese la base de datos semiestructurados de la figura 3.1. La expresión de camino `publicaciones.#.autor?` regresa el conjunto formado por los datos semiestructurados con identificadores $p2$, $p3$, $p4$, $m2$, $m3$ y $p9$. La expresión de camino `publicaciones.#*` regresa el conjunto formado por todos

los datos semiestructurados de la base de datos excepto $m1$, $m4$ y $m7$. La expresión de camino `profesores.profesor.(Nombre|nombre)` regresa el conjunto formado por los datos semiestructurados con identificadores $m5$ y $m7$. La expresión de camino `profesores.(profesor|maestro).nombre` regresa el conjunto formado por los datos semiestructurados con identificadores $m6$ y $m7$. La expresión de camino `(profesores|publicaciones).(libro|profesor)` regresa el conjunto formado por los datos semiestructurados con identificadores $p2$, $p3$, $m2$ y $m4$. La expresión de camino `#+` regresa el conjunto formado por todos los datos semiestructurados de la base de datos.

También es posible utilizar expresiones regulares para describir las etiquetas. Para eliminar las ambigüedades entre las expresiones regulares para las etiquetas y para las expresiones de camino, las primeras se encerrarán entre apóstrofos.

Ejemplo 3.1.4 Considérese la base de datos semiestructurados de la figura 3.1. La expresión de camino `profesores.profesor.'(N|n)ombre'` regresa el conjunto formado por los datos semiestructurados con identificadores $m5$ y $m7$. La expresión `'(N|n)ombre'` genera las cadenas `Nombre` y `nombre`.

La expresión de camino `publicaciones.'#*o'*` regresa el conjunto formado por los datos semiestructurados con identificadores $p1$, $p2$, $p3$, $p4$, $p5$, $p6$ y $p7$. La expresión `'#o*'` genera las cadenas que terminan con o, las que aquí importan son `libro`, `articulo` y `titulo`.

La expresión de camino `'p#*'.'#*o#?'*` regresa el conjunto formado por los datos semiestructurados con identificadores $m1$, $m2$, $m3$, $m4$, $p1$, $p2$, $p3$, $p4$, $p5$, $p6$, $p7$ y $p9$.

3.2. Consultas

La función principal de un lenguaje de consulta para bases de datos son precisamente las consultas, las cuales extraen información de la base de datos. En el caso de las bases de datos semiestructurados el resultado se expresa en un dato semiestructurado.

En el lenguaje de consulta aquí presentado las consultas se realizan por medio de operaciones para crear nuevos datos semiestructurados y para elegir información de los ya existentes. Para realizar una consulta se parte de los datos semiestructurados dados por las ssd-tablas, las vistas y las constantes. Con ellos se pueden construir otros nuevos a través de las operaciones mencionadas, estos nuevos datos semiestructurados a su vez se pueden operar entre ellos o con los ya existentes, y así sucesivamente

hasta obtener al final un dato semiestructurado que es el resultado de la consulta. Una consulta regresa un nuevo dato semiestructurado que existe en un nivel lógico, es decir, la base de datos no sufre ninguna modificación.

En seguida se describen los elementos que se utilizan para la realización de consultas en una base de datos semiestructurados.

3.2.1. Constantes

Se pueden utilizar dos tipos de constantes: La constante EMPTY y las constantes primitivas. La constante EMPTY se refiere al conjunto vacío como un dato semiestructurado con un identificador asociado. Las constantes primitivas se refieren a datos semiestructurados de tipo primitivo. La sintaxis es:

EMPTY

o

P

donde *P* corresponde con la sintaxis para algún dato primitivo.

Al utilizarse la constante EMPTY se construye un nuevo dato semiestructurado: el conjunto vacío, además el sistema manejador de bases de datos le asignará un identificador único que no esté siendo utilizado por algún otro dato semiestructurado. Obsérvese que pueden existir varios conjuntos vacios pero con identificadores distintos.

De igual forma al utilizarse una constante primitiva se construye un nuevo dato semiestructurado de tipo primitivo con un identificador que no haya sido utilizado.

3.2.2. Operaciones para datos semiestructurados

Para crear nuevos datos semiestructurados a partir de otros, se utilizan operaciones para ellos. Las operaciones que existen son la unión, la proyección, la agrupación, la clonación, las funciones de agregación (la suma, el promedio, el máximo y el mínimo) y las operaciones en tipos primitivos de datos (suma, resta, multiplicación, división y módulo).

Al utilizar una operación con datos semiestructurados, se crea un nuevo dato semiestructurado, para éste se asume que el sistema manejador de bases de datos le asigna un identificador único que no haya sido utilizado por ningún otro dato semiestructurado existente en la base de datos.

3.2.2.1. Unión

La sintaxis es:

$$S_1 \text{ UNION } S_2$$

donde S_1 y S_2 son dos datos semiestructurados.

La unión de dos datos semiestructurados es similar a la unión típica para conjuntos, se produce un nuevo dato semiestructurado que contiene los datos semiestructurados etiquetados contenidos en ambos, es decir, dados $S_1 = \{l_{1,1} : S_{1,1}, ..., l_{1,m} : S_{1,m}\}$ y $S_2 = \{l_{2,1} : S_{2,1}, ..., l_{2,n} : S_{2,m}\}$ dos datos semiestructurados entonces S_1 UNION S_2 es el dato semiestructurado $\{l_{1,1} : S_{1,1}, ..., l_{1,m} : S_{1,m}, l_{2,1} : S_{2,1}, ..., l_{2,n} : S_{2,m}\}$.

Obsérvese que si $l_{1,i} = l_{2,j}$ y $S_{1,i} = S_{2,j}$ para algunas $i \in \{1, ..., m\}$ y $j \in \{1, ..., n\}$ entonces dentro de la unión $l_{1,i} : S_{1,i}$ y $l_{2,j} = S_{2,j}$ son exactamente el mismo dato semiestructurado etiquetado, esto quiere decir que en la unión se considerarán como un solo elemento. Si $l_{1,i} \neq l_{2,j}$ aunque $S_{1,i} = S_{2,j}$ estos son distintos datos semiestructurados etiquetados, es decir, en la unión se consideran como dos elementos.

Ejemplo 3.2.1 Considérense S como el dato semiestructurado de la figura 3.2(a) con identificador $a1$ y T como el dato semiestructurado de la figura 3.2 (b) con identificador $b1$, el resultado de S UNION T es el dato semiestructurado con identificador $r1$ (el cual es asignado por el sistema) que se muestra en la figura 3.2(c).

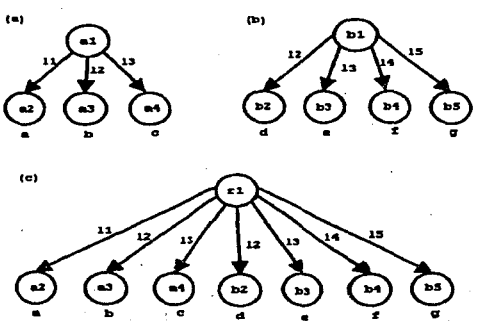


Figura 3.2: Unión de los datos semiestructurados del ejemplo 3.2.1.

Ejemplo 3.2.2 Considérense S como el dato semiestructurado de la figura 3.3(a) con identificador $c1$, T como el de identificador $c2$ y U como el de identificador $c3$, el

resultado de T UNION U es el dato semiestructurado con identificador $r2$ de la figura 3.3(b), el resultado de T UNION S es el dato semiestructurado con identificador $r3$ de la figura 3.3(c), el resultado de S UNION S es el dato semiestructurado con identificador $r3$ de la figura 3.3(d).

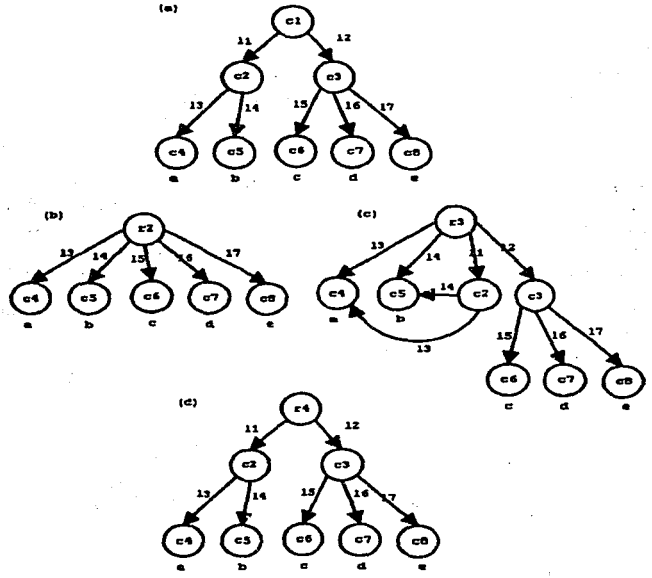


Figura 3.3: Unión de los datos semiestructurados del ejemplo 3.2.2.

Ejemplo 3.2.3 Considérense S como el dato semiestructurado de la figura 3.4(a) con identificador $d2$ y T como el dato semiestructurado con identificador $d3$. El resultado de S UNION T es el dato semiestructurado con identificador $r5$ que se muestra en la figura 3.4(b).

Ejemplo 3.2.4 Considérense S como el dato semiestructurado con identificador $e2$ y T como el de identificador $e3$ de la figura 3.5(a). El resultado de S UNION T es el dato semiestructurado con identificador $r6$ que se muestra en la figura 3.5(b).

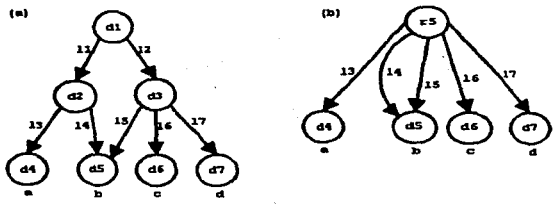


Figura 3.4: Unión de los datos semiestructurados del ejemplo 3.2.3.

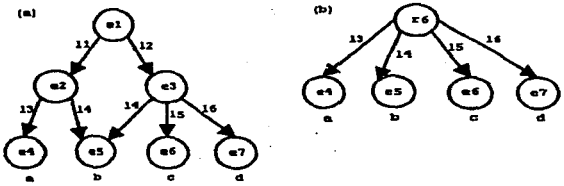


Figura 3.5: Unión de los datos semiestructurados del ejemplo 3.2.4.

Ejemplo 3.2.5 Considérense S como el dato semiestructurado con identificador $f2$ y T como el de identificador $f3$ de la figura 3.6(a). El resultado de S UNION T es el dato semiestructurado con identificador $r7$ que se muestra en la figura 3.6(b).

3.2.2.2. Proyección

En varios dominios, entre ellos las bases de datos relacionales, la proyección es la acción elegir la información que es de utilidad y de eliminar o podar aquella que no lo es.

Para las proyecciones en datos semiestructurados se usan los operadores PICK y TRIM. La sintaxis es:

$$S \text{ PICK } (h_1, ..., h_m)$$

y

$$S \text{ TRIM } (h_1, ..., h_m)$$

donde S es un dato semiestructurado y $h_1, ..., h_m$ son etiquetas.

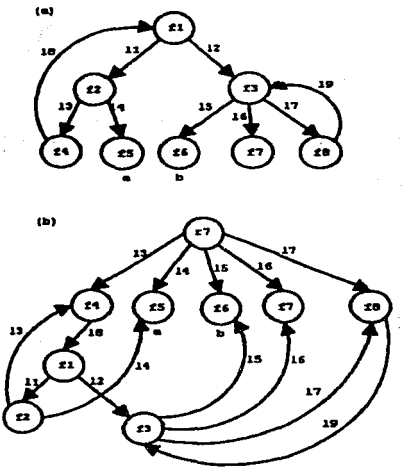


Figura 3.6: Unión de los datos semiestructurados del ejemplo 3.2.5.

El operador PICK aplicado a un dato semiestructurado selecciona los datos semiestructurados etiquetados contenidos en éste y cuyas etiquetas coinciden con las que están dadas como argumentos del operador, con los datos semiestructurados etiquetados elegidos se construye un nuevo dato semiestructurado que los contiene. Si $S = \{l_1 : S_1, ..., l_n : S_n\}$, S PICK $(h1, ..., h_m)$ va a ser el nuevo dato semiestructurado $\{l_i : S_i | \exists i(i \in \{1, ..., n\} \wedge \exists j(j \in \{1, ..., m\} \wedge l_i = h_j))\}$.

El operador TRIM aplicado a un dato semiestructurado elimina los datos semiestructurados etiquetados en éste y cuyas etiquetas coinciden con las que están dadas también como argumentos del operador, con los datos semiestructurados etiquetados elegidos se construye un nuevo dato semiestructurado que los contiene. Si $S = \{l_1 : S_1, ..., l_n : S_n\}$, S TRIM $(h1, ..., h_m)$ va a ser el nuevo dato semiestructurado $\{l_i : S_i | \exists i(i \in \{1, ..., n\} \wedge \forall j(j \in \{1, ..., m\} \rightarrow l_i \neq h_j))\}$.

Ejemplo 3.2.6 Sea S el dato semiestructurado con identificador $r2$ de la figura 3.7(a). El resultado de S PICK $(l3, l4, l5)$ se muestra en la figura 3.7(b), $s1$ es el identificador que el sistema le asigna automáticamente al resultado. El resultado de S TRIM $(l3, l4, l5)$ se muestra en la figura 3.7(c).

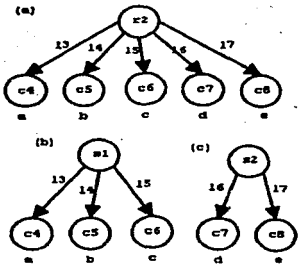


Figura 3.7: Proyección de los datos semiestructurados del ejemplo 3.2.6.

Ejemplo 3.2.7 Considérese S el dato semiestructurado con identificador $r3$ que se muestra en la figura 3.8(a). El resultado de S TRIM $(l3, l2)$ se muestra en la figura 3.8(b).

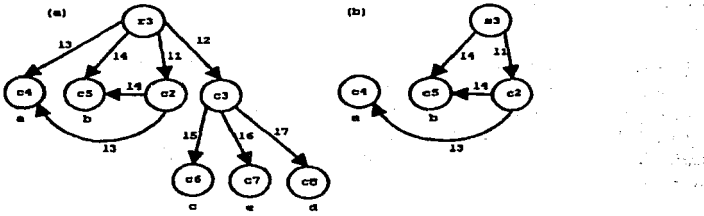


Figura 3.8: Proyección de los datos semiestructurados del ejemplo 3.2.7.

Ejemplo 3.2.8 Considérese S el dato semiestructurado con identificador $r5$ que se muestra en la figura 3.9(a). El resultado de S PICK $(l5, l6, l7)$ se muestra en la figura 3.9(b).

3.2.2.3. Agrupación

La sintaxis de la operación de agrupación es:

$$\{l_1 : S_1, ..., l_n : S_n\}$$

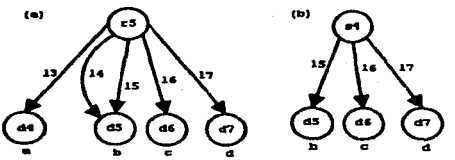


Figura 3.9: Proyección de los datos semiestructurados del ejemplo 3.2.8.

donde $S_1, ..., S_n$ son datos semiestructurados y $l_1, ..., l_n$ son etiquetas.

La agrupación consiste en construir un nuevo dato semiestructurado a partir de uno o varios datos semiestructurados etiquetados, los cuales van a estar contenidos en el nuevo dato. $\{l_1 : S_1, ..., l_n : S_n\}$ es el dato semiestructurado que consta de los datos semiestructurados etiquetados $l_i : S_i$ para $i = 1, ..., n$.

De igual manera que con la unión si $l_{1i} = l_{2j}$ y $S_{1i} = S_{2j}$ para alguna $i \in \{1, ..., m\}$ y alguna $j \in \{1, ..., n\}$ entonces $l_{1i} : S_{1i}$ y $l_{2j} : S_{2j}$ son un solo elemento; en otro caso corresponden a dos elementos en la agrupación.

Ejemplo 3.2.9 Considérense S como el dato semiestructurado con identificador $g1$, T como el de identificador $h1$, U como el de identificador $i1$ y V como el de identificador $g3$ de la figura 3.10(a). El resultado de $\{l4 : S, l5 : T, l6 : U\}$ se muestra en la figura 3.10(b), el resultado de $\{l4 : S, l5 : V\}$ se muestra en la figura 3.10(c), el resultado de $\{l4 : S, l5 : S\}$ se muestra en la figura 3.10(d), el resultado de $\{l4 : S, l4 : S\}$ se muestra en la figura 3.10(e).

3.2.2.4. Clonación

La sintaxis es:

CLON S

donde S es un dato semiestructurado.

Lo que hace la función CLON aplicada a un dato semiestructurado es construir una copia de éste, pero con otros identificadores que no han sido utilizados. Se asume que el sistema asigna estos identificadores automáticamente.

Ejemplo 3.2.10 Considérese S como el dato semiestructurado con identificador $j1$, de la figura 3.11(a), el resultado de CLON S se muestra en la figura 3.11(b).

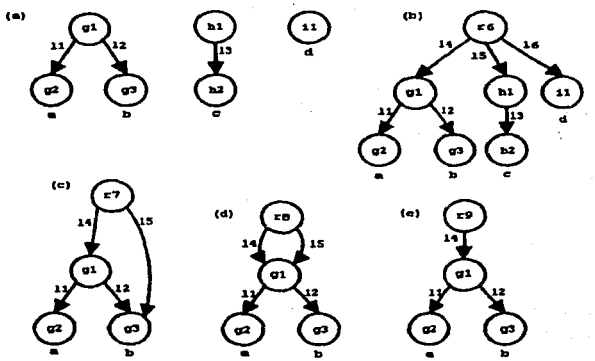


Figura 3.10: Agrupación de los datos semiestructurados del ejemplo 3.2.9.

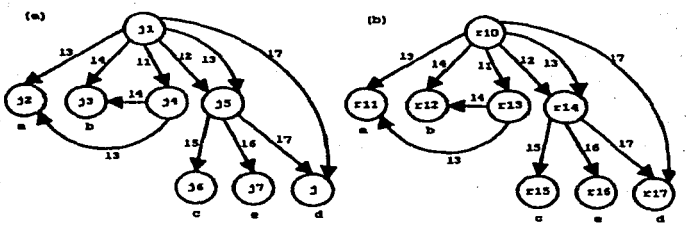


Figura 3.11: Clonación del dato semiestructurado del ejemplo 3.2.10.

3.2.2.5. Operaciones de Tipos Primitivos

Las operaciones de tipos primitivos de datos son +, −, *, /, MOD. La sintaxis es:

$$S_1 + S_2$$

$$S_1 - S_2$$

$$S_1 * S_2$$

$$S_1 / S_2$$

$$S_1 \text{ MOD } S_2$$

donde S_1 y S_2 son datos semiestructurados de tipo primitivo.

Las operaciones de tipos primitivos se aplican a datos semiestructurados de tipo primitivo. Las operaciones dependen del tipo de datos que se utilicen. En este momento sólo se asumirá que se tienen tres tipos de datos primitivos: números enteros, números de punto flotante y cadenas de caracteres. Los operadores, así como los tipos primitivos de datos pueden ampliarse sin ningún problema, pero en este trabajo sólo se asumirán tres tipos primitivos y cinco operadores.

El operador + tiene sentido para cadenas de caracteres, para números enteros y para números de punto flotante, para cadenas de caracteres es la concatenación y para números es la suma tradicional. Los operadores −, * y / tienen sentido para números enteros y de punto flotante, estos son la resta, producto y división tradicionales, la división entre enteros regresa solamente el cociente entero. El operador MOD sólo tiene sentido para números enteros y este regresa el residuo de la división de dos enteros.

En caso de que se trabajen dos operandos de distinto tipo, el del tipo menor se promueve al de tipo mayor, así el operador regresa un dato primitivo del mayor de los tipos. Los números enteros se pueden promover a números de punto flotante o a cadenas de caracteres; los números de punto flotante se pueden promover a cadenas de caracteres. Para promover los números a cadenas de caracteres se usa la representación como cadena de caracteres de estos, por ejemplo, el número 42 se promueve a la cadena "42". Así el menor de los tipos es el de números enteros, le sigue el de números de punto flotante y el mayor es el de cadenas de caracteres.

Una operación de tipo primitivo de datos aplicada a dos datos primitivos regresa un nuevo dato semiestructurado primitivo.

Ejemplo 3.2.11 Considérense S como el dato semiestructurado primitivo con identificador $r19$, de la figura 3.12(a) y T como el de la figura 3.12(b) con identificador $r20$, el resultado de $S + T$ se muestra en la figura 3.12(c).

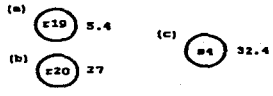


Figura 3.12: Operaciones de tipos primitivos del ejemplo 3.2.11.

3.2.2.6. Funciones de Agregación

Las funciones de agregación pueden ser COUNT, AVG, SUM, MAX, MIN, la sintaxis es:

COUNT S

AVG S

SUM S

MAX S

MIN S

donde S es un dato semiestructurado, pero con algunas restricciones según la función, que se tratarán enseguida.

La función COUNT aplicada a un dato semiestructurado, devuelve un dato semiestructurado primitivo, que corresponde al número de subdatos semiestructurados etiquetados contenidos en el dato semiestructurado dado, es decir, regresa el numero de hijos del dato. Dado $S = \{l_1 : S_1, ..., l_n : S_n\}$, suponiendo que los $l_i : S_i$ son todos distintos, es decir, $l_i \neq l_j$ o $S_i \neq S_j$ para todas $i, j = 1, ..., n$, se tiene que COUNT S es el dato primitivo que corresponde al número n.

Un dato semiestructurado en el cual sus hijos son todos de tipo primitivo, se dice que es un *contenedor de primitivos*. O más formalmente, si $S = \{l_1 : S_1, ..., l_n : S_n\}$ es un dato semiestructurado S es un *contenedor de primitivos* si y sólo si S_i es de tipo primitivo para toda $i = 1, ..., n$.

Las funciones de agregación AVG, MAX, MIN y SUM se aplican solamente a contenedores de primitivos y el resultado es un nuevo dato semiestructurado de tipo primitivo. En caso de que se operen distintos tipos de datos primitivos, se aplican las reglas de promoción de tipos tratadas en la sección anterior.

Dado $S = \{l_1 : S_1, ..., l_n : S_n\}$ un contenedor de primitivos, asumiendo que para todas $i, j = 1, ..., n$, $l_i \neq l_j$ o $S_i \neq S_j$, se tiene lo siguiente:

- AVG S es el nuevo dato semiestructurado primitivo que corresponde al número $(S_1 + ... + S_n)/n$. Esta función de agregación sólo se puede aplicar si cada S_i es un número entero o de punto flotante.
- SUM S es el nuevo dato semiestructurado primitivo que corresponde a la evaluación de $S_1 + ... + S_n$. Esta función de agregación se puede aplicar si cada S_i es un número entero, de punto flotante o una cadena de caracteres.

- MAX S devuelve el dato semiestructurado primitivo que corresponde a la evaluación de máx$\{S_1, ..., S_n\}$. Esta función de agregación se puede aplicar si cada S_i es un número entero, de punto flotante o una cadena de caracteres.
- MIN S devuelve el dato semiestructurado primitivo que corresponde a la evaluación de mín$\{S_1, ..., S_n\}$. Esta función de agregación se puede aplicar si cada S_i es un número entero, de punto flotante o una cadena de caracteres.

Ejemplo 3.2.12 Considérese S como el dato semiestructurado con identificador $j1$, de la figura 3.13(a). El resultado de COUNT S se muestra en la figura 3.13(b).

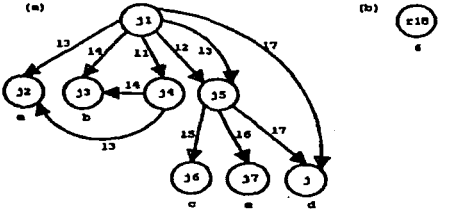


Figura 3.13: Funciones de agregación del ejemplo 3.2.12.

Ejemplo 3.2.13 Sea S el dato semiestructurado con identificador $k1$ de la figura 3.14(a). El resultado de AVG S se muestra en la figura 3.14(b), el resultado de SUM S se muestra en la figura 3.14(c), el resultado de MAX S se muestra en la figura 3.14(d), el resultado de MIN S se muestra en la figura 3.14(e).

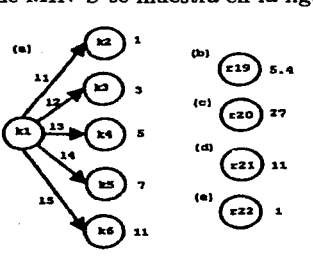


Figura 3.14: Funciones de agregación del ejemplo 3.2.13.

3.2.2.7. Precedencia y asociatividad

Para saber el orden en el que se aplican los operadores anteriores y, por tanto, dónde poner paréntesis y dónde se pueden ahorrar, es necesario tener una precedencia de operadores. La precedencia y asociatividad de los operadores que se utilizan para las consultas y que devuelven un dato semiestructurado se muestra en el cuadro 3.1, en orden de mayor a menor precedencia.

Precedencia	Operador	Asociatividad
10	CLON AVG SUM MAX MIN COUNT	derecha-izquierda
9	PICK TRIM	izquierda-derecha
8	* / MOD	izquierda-derecha
7	+ −	izquierda-derecha
6	{:} (agrupación)	derecha-izquierda
5	UNION	izquierda-derecha

Cuadro 3.1: Precedencia y asociatividad de operadores

Cabe destacar que en el operador de agrupación lo más anidado es lo que se realizan primero.

3.2.3. EL enunciado SELECT-FROM-WHERE

De manera análoga a SQL y OQL, en Ssquirel se debe utilizar el enunciado *SELECT-FROM-WHERE* como la parte central de las consultas. Este enunciado permite seleccionar un conjunto de datos semiestructurados en la base de datos y con ellos construir uno nuevo. Dicho enunciado consta de tres cláusulas: La cláusula SELECT, la cláusula FROM y la cláusula WHERE.

La sintaxis es la siguiente:

SELECT l : *construction*
FROM e_1 AS X_1, ..., e_n AS X_n
WHERE *condition*

donde l es una etiqueta, $e_1, ..., e_n$ son expresiones de camino (con algunas restricciones que se explicarán en breve), $X_1, ..., X_n$ son nombres de variables, *construction* es una construcción de un nuevo dato semiestructurado y *condition* es una condición (ambos también se explicaran en breve). La cláusula WHERE es opcional, las cláusulas SELECT y FROM son obligatorias.

La semántica del enunciado SELECT-FROM-WHERE consta de 3 etapas, la primera es la etapa de la cláusula FROM, la segunda es la etapa de la cláusula WHERE y la tercera es la de la cláusula SELECT.

En la cláusula FROM aparecen $e_1, ..., e_n$, que son expresiones de camino. La expresión de camino e_1 puede comenzar con el nombre de alguna ssd-tabla o vista existente en la base de datos o también con alguna de las ssd-tablas temporales visibles creadas por consultas en un nivel exterior de anidamiento, X_1 va a ser el nombre de una ssd-tabla temporal asociada a un dato semiestructurado que concuerda con la expresión de camino e_1. La expresión de camino e_2 puede comenzar con el nombre de una ssd-tabla o vista existente, una tabla temporal visible de una consulta exterior o X_1. Ahora X_2 va a ser el nombre de una ssd-tabla temporal que se refiere a un dato semiestructurado que concuerda con la expresión de camino e_2. Siguiendo con este proceso, la expresión de camino e_i puede comenzar con el nombre de alguna tabla o vista existente, una tabla temporal visible creada por alguna consulta en un nivel de anidamiento exterior o con alguna de $X_1, ..., X_{i-1}$. Por supuesto X_i será el nombre de la ssd-tabla temporal que concuerda con la expresión de camino e_i, esto para $i = 1, ..., n$. Así se crean $X_1, ..., X_n$ que corresponden a ssd-tablas temporales.

La cláusula WHERE consta de una condición en la cual pueden aparecer los datos semiestructurados dados por las ssd-tablas y vistas existentes en la base de datos, las ssd-tablas temporales visibles dadas por consultas exteriormente anidadas, las ssd-tablas temporales $X_1, ..., X_n$ creadas en el FROM, constantes primitivas, la constante EMPTY y las construcciones a partir de cualquiera de ellas utilizando para ello las operaciones de unión, proyección, agrupación, clonación, operaciones primitivas y funciones de agregación tratadas anteriormente. La condición se basa en las reglas de coerción que se explicarán posteriormente. La condición regresara un valor de falso o verdadero.

La cláusula SELECT es la que se encarga de construir un nuevo dato semiestructurado etiquetado, si la condición en la parte del WHERE resultó verdadera. Para construir un dato semiestructurado etiquetado se usa una etiqueta l y una construcción que es una combinación de las operaciones de unión, proyección, agrupación,

clonación, operaciones primitivas y funciones de agregación aplicadas a las ssd-tablas y vistas existentes en la base de datos, las ssd-tablas temporales visibles dadas por consultas exteriormente anidadas, las ssd-tablas temporales $X_1, ..., X_n$ creadas en el FROM, constantes primitivas, la constante EMPTY y otras construcciones.

Lo descrito anteriormente es un ciclo de 3 etapas y lo que se obtiene es un dato semiestructurado etiquetado. Para el resultado de la consulta se requiere realizar tantos ciclos como combinaciones de valores para $X_1, ..., X_n$ sean posibles (que es un número finito). El resultado de la consulta es un nuevo dato semiestructurado que contiene los datos semiestructurados etiquetados que se obtuvieron en cada uno de los ciclos, el identificador de este nuevo dato al igual que con todas las operaciones tratadas anteriormente no debe haber sido utilizado y se puede asumir que el sistema manejador de bases de datos lo asigna automáticamente.

Ejemplo 3.2.14 Considérese la base de datos semiestructurados que se muestra en la figura 3.15. Ésta sólo consta de una ssd-tabla llamada `profesores`.

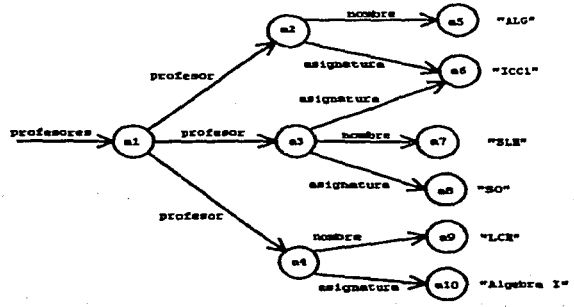


Figura 3.15: Base de datos semiestructurados de una sola ssd-tabla.

Considérese la consulta siguiente:

```
SELECT nombre_profesor: X
FROM   profesores.profesor.nombre AS X
```

Los únicos datos semiestructurados que coinciden con la expresión de camino dada en la parte del FROM son los que tiene identificadores $a5$, $a7$ y $a9$; así que el número de etapas a ejecutar es 3.

En la primera etapa, en la parte del FROM se elige el dato semiestructurado con identificador $a5$. Este dato semiestructurado es asociado a la ssd-tabla temporal X. En la parte del SELECT se construye el dato semiestructurado etiquetado *nombre_profesor* : &$m5$ "ALG", el cual va a formar parte del resultado de la consulta. De la misma manera en la segunda etapa se construye el dato semiestructurado etiquetado *nombre_profesor* : &$m7$ "SLM" y en la tercera etapa se construye *nombre_profesor* : &$m9$ "LCM".

Finalmente el resultado de la consulta es el dato semiestructurado:

```
&sm1 { nombre_profesor:&m5 "ALG",
       nombre_profesor:&m7 "SLM",
       nombre_profesor:&m9 "LCM"}
```

que se muestra en la figura 3.16. Se asume que el sistema asigna el identificador $sm1$ al resultado de la consulta.

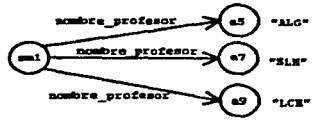


Figura 3.16: Resultado de la consulta del ejemplo 3.2.14.

Ejemplo 3.2.15 En la siguiente consulta a la base de datos de la figura 3.15, se eligen cada uno de los datos semiestructurados con etiqueta `profesor` y se agrupan en un nuevo dato con la etiqueta `maestro`.

```
SELECT maestro: X
FROM   profesores.profesor AS X
```

El resultado se observa en la figura 3.17. Es muy similar a la ssd-tabla original solamente que las etiquetas `profesor` se cambian por `maestro` y la raíz es un nuevo dato semiestructurado creado por el sistema.

Ejemplo 3.2.16 La consulta siguiente aplicada a la base de datos de la figura 3.15, produce el resultado que se muestra en la figura 3.18.

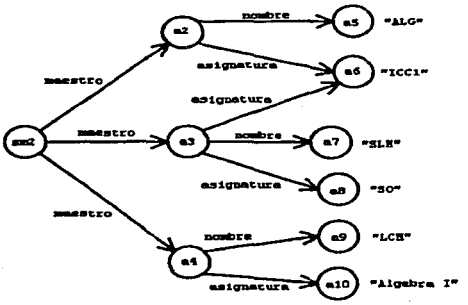


Figura 3.17: Resultado de la consulta del ejemplo 3.2.15.

```
SELECT grupo: {profesor: Y, materia: Z}
FROM   profesores.profesor AS X,
       X.nombre AS Y,
       X.asignatura AS Z
```

En la parte del FROM se utilizan tres ssd-tablas temporales `X`, `Y` y `Z`, los datos semiestructurados a los cuales se pueden enlazar tienen las posibles combinaciones $(a2, a5, a6)$, $(a3, a7, a6)$, $(a3, a7, a8)$ y $(a4, a9, a10)$; así que el número de etapas a considerar es 4. En la parte del SELECT por cada una de las 4 combinaciones se construye un nuevo dato semiestructurado etiquetado como `grupo` y que consta de los datos semiestructurados obtenidos en `Y` y `Z`, etiquetados con `profesor` y `materia`, respectivamente.

Ejemplo 3.2.17 El propósito de la siguiente consulta a la base de datos de la figura 3.15 es agrupar las materias de un profesor en un solo dato semiestructurado. Así cada profesor tendrá un nombre y un dato semiestructurado que contiene sus materias.

```
SELECT profesor: {nombre: Y, materias: X PICK(asignatura)}
FROM   profesores.profesor AS X,
       X.nombre AS Y
```

El resultado de la consulta se muestra en la figura 3.19. Obsérvese como en la parte del SELECT se crean dos nuevos datos por etapa.

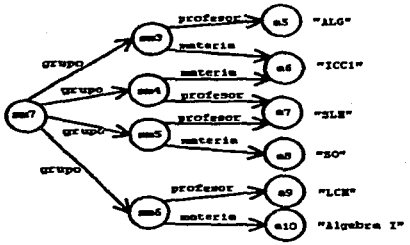


Figura 3.18: Resultado de la consulta del ejemplo 3.2.16.

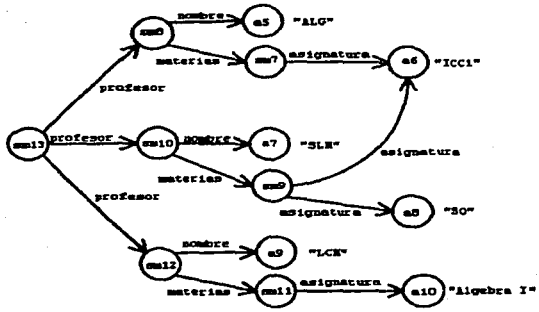


Figura 3.19: Resultado de la consulta del ejemplo 3.2.17.

Ejemplo 3.2.18 En la siguiente consulta a la base de datos de la figura 3.15, se pretende que, por cada profesor, se tenga un dato semiestructurado que contenga su nombre y el número de materias que imparte. El resultado se muestra en la figura 3.20.

```
SELECT profesor: {nombre: Y, num_materias: COUNT (X PICK(asignatura))}
FROM   profesores.profesor AS X,
       X.nombre AS Y
```

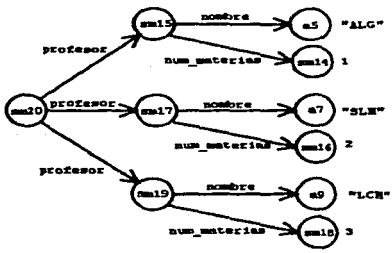


Figura 3.20: Resultado de la consulta del ejemplo 3.2.18.

La palabra SELECT puede ser sucedida de la palabra DISTINCT. En este caso si en dos ciclos distintos se construye esencialmente el mismo dato entonces sólo se tomará uno de ellos para incluirlo en el resultado de la consulta. Para construir un nuevo dato semiestructurado en la parte del SELECT, es posible que no se requieran todos los $X_1, ..., X_n$ de la parte del FROM, sino solamente algunos de ellos. Si $X_{i_1}, ..., X_{i_k}$ es lo mínimo que se requiere para construir el dato semiestructurado entonces dos datos semiestructurados son esencialmente el mismo si se construyen con los mismos valores para $X_{i_1}, ..., X_{i_k}$ aunque los demás tengan distinto valor.

La cláusula WHERE lleva una condición[3] que regresa un valor de verdadero o falso, la condición se puede formar de lo siguiente:

- Las constantes TRUE y FALSE que siempre regresan los valores verdadero y falso respectivamente.
- Operadores aplicados a datos semiestructurados[4] que regresan valores de verdadero o falso, estos operadores se explicarán posteriormente en las reglas de coerción.
- Los operadores AND, OR y NOT que sirven para construir condiciones a partir de otras condiciones, estos operadores siguen las reglas ya conocidas de la lógica proposicional. La sintaxis es:

$$cond_1 \text{ AND } cond_2$$

[3]Estrictamente hablando, una condición es una fórmula bien formada consistente de fórmulas atómicas y operadores lógicos.

[4]Estos operadores corresponden a las formulas atómicas o predicados.

$cond_1$ OR $cond_2$

NOT $cond_1$

donde $cond_1$ y $cond_2$ son condiciones.

- Los operadores FOR ALL y EXIST que tienen la sintaxis:

FOR ALL v IN S (*cond*)

EXIST v IN S (*cond*)

donde S es un dato semiestructurado, v es el nombre de una variable y *cond* es una condición.

Si S es un dato semiestructurado, la condición FOR ALL v IN S (*cond*) va a dar un resultado de verdadero si todos los datos semiestructurados contenidos en S y que intuitivamente son enlazados uno por uno a la variable v hacen verdadera a la condición *cond*. El resultado de la condición EXIST v IN S (*cond*) va a ser verdadero si algún dato semiestructurado contenido en S hace verdadera la condición *cond*, hay que tomar en cuenta que la variable v se enlaza a cada dato contenido en S.

3.2.3.1. Las Reglas de Coerción

Los operadores que se describen a continuación constituyen las reglas de coerción. Éstos se aplican a los datos semiestructurados y regresan algún valor de verdadero y falso

1. Operadores de comparación para tipos primitivos.

   Los operadores < (menor que), > (mayor que), <= (menor o igual), >= (mayor o igual), = (igual), <> (distinto), se aplican a dos datos semiestructurados de tipo primitivo. Su sintaxis es:

   S_1 *operador* S_2

   donde S_1 y S_2 son datos semiestructurados de tipo primitivo y *operador* es uno de los operadores anteriores.

   Recuérdese que por el momento sólo se asumen como tipos primitivos de datos los números enteros, los números de punto flotante y las cadenas de caracteres, aunque éstos también pueden aumentar junto con los operadores de comparación. Los operadores anteriores regresan el valor de verdadero o falso según la semántica conocida para estos operadores y el tipo de datos a los que se aplican.

Si se operan dos datos de tipo distinto el de menor tipo se promueve al de tipo mayor, en el caso de las cadenas de caracteres el resultado del operador depende del orden lexicográfico; en el caso de los tipos numéricos el resultado depende del orden tradicional.

2. El operador LIKE.

   Este operador se aplica a un tipo primitivo de datos y a una cadena de caracteres. Su comportamiento es el mismo que el utilizado para SQL, el tipo primitivo de datos necesariamente debe ser promovido a una cadena de caracteres. La sintaxis es:

   $$S \text{ LIKE } str$$

   donde S es un dato semiestructurado y str es una cadena de caracteres, en str se utilizan los caracteres _ y % como comodines. El primero se remplaza por un carácter y el segundo por cero o más caracteres, también se utiliza el carácter de escape \. El operador S LIKE str va a regresar verdadero si y sólo si S concuerda con str.

3. El operador BELONG.

   El operador BELONG sirve para verificar que un dato semiestructurado pertenece a otro. La sintaxis es:

   $$S_1 \text{ BELONG } S_2$$

   donde S_1 y S_2 son datos semiestructurados.

   Dados S_1 y S_2 datos semiestructurados S_1 BELONG S_2 va a ser verdadero si y sólo si S_1 está en S_2, es decir que existe una etiqueta l tal que el dato semiestructurado etiquetado $l : S_1$ pertenece a S_2, lo cual quiere decir que si $S_2 = \{l_1 : T_1, ..., l_n : T_n\}$ entonces $S_1 = T_i$ para alguna $i \in \{1, ..., n\}$. Evidentemente si S_2 es un dato primitivo el resultado va a ser falso.

4. El operador CONTAIN.

   El operador CONTAIN funciona de manera inversa al operador BELONG, sirve para verificar que un dato semiestructurado contiene a otro, la sintaxis es:

   $$S_1 \text{ CONTAIN } S_2$$

   donde S_1 y S_2 son datos semiestructurados.

   Dados S_1 y S_2 datos semiestructurados, S_1 CONTAIN S_2 va a ser verdadero si y sólo si S_1 contiene a S_2, es decir que existe una etiqueta l tal que el dato semiestructurado etiquetado $l : S_2$ pertenece a S_1, esto quiere decir que si $S_1 = \{l_1 : T_1, ..., l_n : T_n\}$ entonces $S_2 = T_i$ para alguna $i \in \{1, ..., n\}$. Si S_1 es un dato primitivo el resultado va a ser falso.

5. El operador OWN.

   El operador OWN verifica que un dato semiestructurado contiene un dato semiestructurado etiquetado con alguna etiqueta dada. Su sintaxis es:

   $$S \text{ OWN } l$$

   donde l es una etiqueta y S es un dato semiestructurado.

   Dados l una etiqueta y S un dato semiestructurado S OWN l va a regresar verdadero si S contiene un dato semiestructurado etiquetado con l, esto es, si existe T un dato semiestructurado tal que $l : T$ pertenece a S, es decir, si $S = \{l_1 : S_1, ..., l_n : S_n\}$ entonces $l = l_i$ para alguna $i \in \{1, ..., n\}$. Claramente, si S es un dato semiestructurado primitivo el resultado va a ser falso.

6. El operador IS.

   El operador IS sirve para verificar que dos datos semiestructurados sean el mismo. La sintaxis es:

   $$S_1 \text{ IS } S_2$$

   donde S_1 y S_2 son datos semiestructurados.

   S_1 IS S_2 va a ser verdadero si y sólo si S_1 y S_2 tienen el mismo identificador.

7. El operador ISOMORPH.

   El operador ISOMORPH sirve para verificar que dos datos son iguales aunque no sean el mismo, es decir, son isomorfos, la sintaxis es:

   $$S_1 \text{ ISOMORPH } S_2$$

   donde S_1 y S_2 son datos semiestructurados.

   S_1 ISOMORPH S_2 va a ser verdadero si y sólo si S_1 y S_2 son isomorfos, esto se puede verificar construyendo la gráfica que los representa, si ambas gráficas son isomorfas entonces los datos semiestructurados también lo son.

8. El operador PRIMITIVE.

   El operador PRIMITIVE verifica si un dato semiestructurado es de tipo primitivo, su sintaxis es:

   $$\text{PRIMITIVE } S$$

   donde S es un dato semiestructurado.

   PRIMITIVE S va a regresar verdadero si y sólo si S es un dato semiestructurado de tipo primitivo.

En las consultas también se requiere de una precedencia y asociatividad en los operadores que se utilizan. Éstos devuelven un valor booleano, su precedencia y asociatividad se muestra en el cuadro 3.2. En las condiciones también se pueden utilizar los operadores para construir nuevos datos semiestructurados que se utilizan en las consultas. Éstos últimos deben tener una precedencia más alta que los operadores que regresan valores booleanos.

Precedencia	Operador	Asociatividad
	PRIMITIVE	derecha-izquierda
4	< > <= >= = <> LIKE BELONG CONTAIN EXIST IN OWN IS ISOMORPH	izquierda-derecha
3	FOR ALL EXIST	derecha-izquierda
2	NOT	derecha-izquierda
1	AND OR	izquierda-derecha

Cuadro 3.2: Precedencia y asociatividad de los operadores en las condiciones

Obsérvese que los operadores de precedencia 4 se aplican a datos semiestructurados y regresan valores booleanos, por lo cual es imposible hacer una asociación entre ellos. En los operadores de precedencia 3, los que estén más anidados se van a ejecutar primero.

Ejemplo 3.2.19 Considérese la figura 3.21. Sea S_1 el dato semiestructurado que se refiere al de identificador $m1$, S_2 el que se refiere al de identificador $m2$ y así sucesivamente hasta que S_{15} es el dato semiestructurado con identificador $m15$.

- $S_{10} > S_9$ es verdadero.
- $S_5 > S_{11}$ es falso.

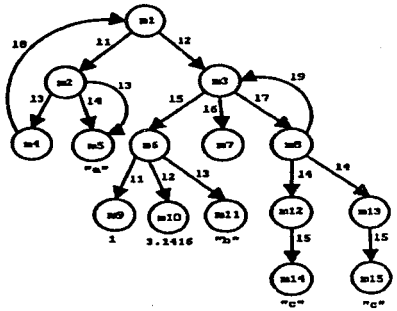


Figura 3.21: Dato semiestructurado del ejemplo 3.2.19.

- S_{14} = S_{15} es verdadero.
- S_5 LIKE "%a%" es verdadero.
- PRIMITIVE S_9 es verdadero.
- PRIMITIVE S_2 es falso.
- PRIMITIVE S_7 es falso, ya que S_7 es el conjunto vacío.
- S_{12} = S_{13} no está definido.
- S_{12} = S_{12} no está definido.
- S_{12} IS S_{12} es verdadero.
- S_{12} IS S_{13} es falso.
- S_{12} ISOMORPH S_{13} es verdadero.
- S_{14} IS S_{15} es falso.
- S_{14} ISOMORPH S_{15} es verdadero.
- S_1 ISOMORPH S_1 es verdadero.
- S_1 ISOMORPH S_3 es falso.
- S_{11} <> S_{15} es verdadero.

- S_{12} <> S_{15} no está definido.
- NOT (S_{12} IS S_{15}) es verdadero.
- S_{14} BELONG S_{12} es verdadero.
- S_{14} BELONG S_8 es falso.
- S_8 BELONG S_3 es verdadero.
- S_3 BELONG S_8 es verdadero.
- S_6 CONTAIN S_9 es verdadero.
- S_6 CONTAIN 1 es falso, recuérdese que las constantes crean un nuevo dato semiestructurado con un identificador que no ha sido utilizado.
- EXIST v IN S_6 (v = 1) es verdadero.
- S_8 OWN $l9$ es verdadero.
- FOR ALL v IN S_6 (PRIMITIVE v) es verdadero.
- S_7 IS EMPTY es falso.
- S_7 ISOMORPH EMPTY es verdadero.
- S_7 = EMPTY no está definido.
- EMPTY IS EMPTY es falso.
- EMPTY ISOMORPH EMPTY es verdadero.
- FOR ALL v IN S_3
  (v ISOMORPH EMPTY
  OR v CONTAIN S_3
  OR EXIST w IN v
  (PRIMITIVE w AND w = 1)) Es verdadero

Ejemplo 3.2.20 Considérese la base de datos semiestructurados que se muestra en la figura 3.22. En ella se tienen tres ssd-tablas llamadas `cursos`, `profesores` y `publicaciones`, las tres ssd-tablas comparten datos semiestructurados en común.

El resultado de la consulta:

```
SELECT publicacion: P
FROM   publicaciones.# AS P
```

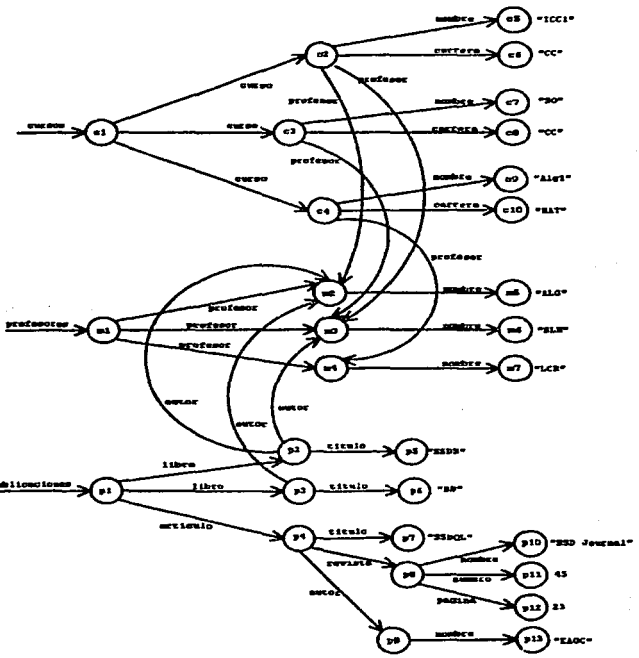


Figura 3.22: Base de datos semiestructurados.

se muestra en la figura 3.23. En un nuevo dato semiestructurado se agrupan las publicaciones con sus respectivos subdatos, entre ellos los autores. Obsérvese que los subdatos pueden al mismo tiempo estar en una ssd-tabla distinta a la de publicaciones.

Ejemplo 3.2.21 El resultado de la siguiente consulta a la base de datos de la figura 3.22, se muestra en la figura 3.24.

```
SELECT profesor: A
FROM   publicaciones.libro AS L
       L.autor AS A
```

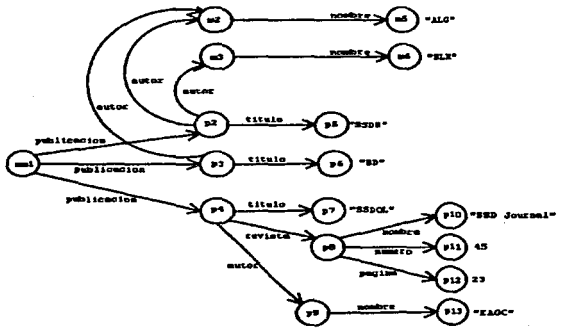


Figura 3.23: Resultado de la consulta del ejemplo 3.2.20.

Lo que se hace es elegir los profesores (de la ssd-tabla `profesores`) que han publicado algún libro (los libros se encuentran en la ssd-tabla `publicaciones`).

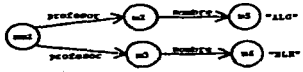


Figura 3.24: Resultado de la consulta del ejemplo 3.2.21.

Ejemplo 3.2.22 En la consulta siguiente aplicada a la base de datos de la figura 3.22, lo que se pretende es elegir los cursos (de la ssd-tabla `cursos`) que son impartidos por 2 profesores.

```
SELECT curso: C
FROM   cursos.curso AS C
WHERE  2 = COUNT (C PICK(profesor))
```

El resultado se muestra en la figura 3.25.

Ejemplo 3.2.23 En la siguiente consulta a la base de datos de la figura 3.22, se crea un nuevo dato semiestructurado que contiene otros datos que constan de un profesor y una de sus publicaciones, es decir se construyen parejas de un profesor y una publicación.

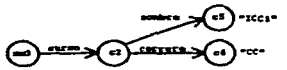


Figura 3.25: Resultado de la consulta del ejemplo 3.2.22.

```
SELECT profesor_publicacion: {profesor: X, publicacion: P}
FROM   profesores.profesor AS X,
       publicaciones.# AS P,
       P.autor AS A
WHERE  A IS X
```

El resultado se muestra en la figura 3.26.

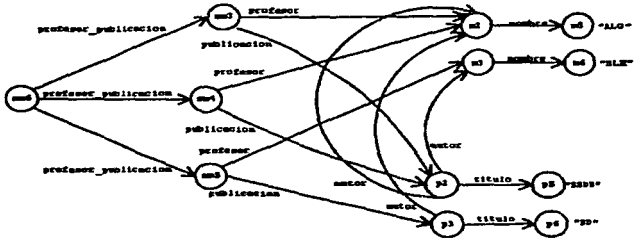


Figura 3.26: Resultado de la consulta del ejemplo 3.2.23.

Ejemplo 3.2.24 La siguiente consulta a la base de datos de la figura 3.22, crea un nuevo dato semiestructurado que contiene los datos de los profesores y sus respectivas publicaciones, es decir, lo que se crea son profesores pero con todas sus publicaciones incluidas.

```
SELECT profesor: X UNION (SELECT publicacion: P
                          FROM publicaciones.# AS P,
                               P.autor AS A
                          WHERE A IS X)
FROM   profesores.profesor AS X
```

El resultado se muestra en la figura 3.27.

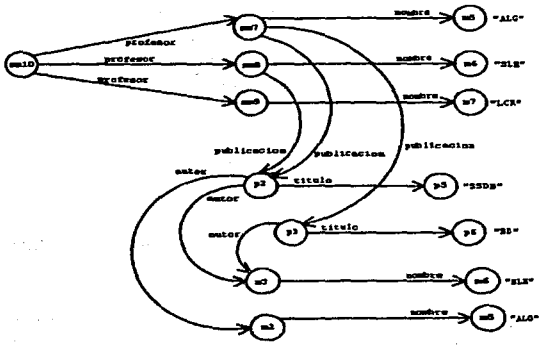


Figura 3.27: Resultado de la consulta del ejemplo 3.2.24.

Ejemplo 3.2.25 La siguiente consulta a la base de datos de la figura 3.22, construye un dato semiestructurado que consta de datos que contienen el nombre de un profesor y el número de cursos que tiene.

```
SELECT profesor: {nombre: N,
                  n_cursos: COUNT (SELECT curso: C
                                   FROM cursos.curso AS C,
                                        C.profesor AS M
                                   WHERE M IS X)}
FROM   profesores.profesor AS X,
       X.nombre AS N
```

Obsérvese como en la subconsulta anidada se eligen los cursos que tiene cada profesor. El resultado se ve en la figura 3.28.

Ejemplo 3.2.26 La siguiente consulta

```
SELECT curso: C TRIM(carrera)
FROM   cursos.curso AS C,
       C.profesor AS A,
       A.nombre AS N
WHERE  N="ALG" OR N="SLM"
```

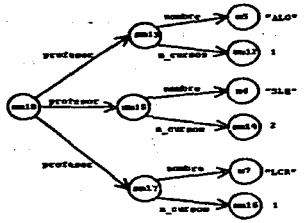


Figura 3.28: Resultado de la consulta del ejemplo 3.2.25.

aplicada a la base de datos de la figura 3.22, elige los cursos impartidos por los profesores con nombre "ALG" y "SLM", pero no incluye el dato correspondiente a la carrera. El resultado se observa en la figura 3.29.

En la parte del FROM se obtienen las combinaciones de identificadores $(c2, m2, m5)$, $(c2, m3, m6)$ y $(c3, m3, m6)$ para C, A y N. Como el operador TRIM construye un nuevo dato semiestructurado en cada etapa, en las primeras dos combinaciones en la parte del SELECT se construye un dato semiestructurado distinto para cada una de ellas. Aunque estos dos nuevos datos en esencia son el mismo, solamente difieren en el identificador.

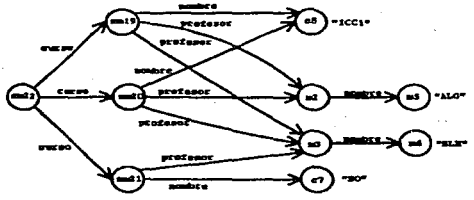


Figura 3.29: Resultado de la consulta del ejemplo 3.2.26.

Ejemplo 3.2.27 La siguiente consulta es similar a la de ejemplo 3.2.26 pero aquí se utiliza la palabra DISTINCT, esto evita que la primera y la segunda combinación de identificadores produzcan dos datos distintos, así solamente se construye uno. Esto se debe a que el dato que se construye en la parte del SELECT solamente depende del valor de C que en la primera y segunda combinación es el mismo. El resultado se muestra en la figura 3.30.

```
SELECT DISTINCT curso: C TRIM(carrera)
FROM    cursos.curso AS C,
        C.profesor AS A,
        A.nombre AS N
WHERE   N="ALG" OR N="SLM"
```

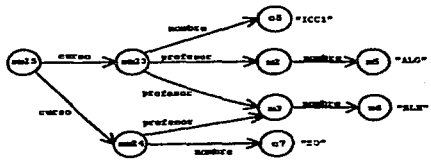


Figura 3.30: Resultado de la consulta del ejemplo 3.2.27.

3.3. Actualización

Las consultas proporcionan información contenida en la base de datos. Para ello devuelven un dato semiestructurado temporal, en las consultas la base de datos no sufre de ninguna modificación. Sin embargo, en ocasiones es necesario hacer cambios a los datos existentes en la base de datos, por ello también se incluye en el lenguaje un mecanismo de actualización de los datos existentes. Los cambios se deben hacer en las ssd-tablas; las vistas no se pueden modificar directamente, aunque los cambios que se hagan en las ssd-tablas se van a reflejar en las vistas y posiblemente en otras ssd-tablas.

Para modificar datos semiestructurados se usan dos operaciones básicas que son el borrado y la edición, las cuales se explican a continuación.

3.3.1. Borrado

Para eliminar subdatos a una profundidad arbitraria en un dato semiestructurado correspondiente a una ssd-tabla, se incluye el enunciado *DELETE-FROM-WHERE* la sintaxis es:

DELETE X_i
FROM e_1 AS X_1, ..., e_n AS X_n
WHERE *condition*

donde $e_1, ..., e_n$ son expresiones de camino, $X_1, ..., X_n$ son nombres de variables y *condition* es una condición. La cláusula WHERE es opcional, las cláusulas DELETE y FROM son obligatorias.

La sintaxis es similar a la utilizada en el enunciado SELECT-FROM-WHERE; el enunciado DELETE-FROM-WHERE también consta de 3 etapas.

La primera etapa es la dada por la cláusula FROM, de igual manera que con el enunciado SELECT-FROM-WHERE a partir de las expresiones de camino $e_1, ..., e_n$ se construyen las nuevas tablas temporales llamadas $X_1, ..., X_n$. La diferencia con la cláusula FROM del enunciado SELECT-FROM-WHERE es que aquí no se permite que las expresiones de camino se refieran a datos semiestructurados en una vista, esto es para evitar que las vistas puedan ser modificadas.

La segunda etapa es la de la cláusula WHERE, aquí hay una condición donde se verifica que las tablas $X_1, ..., X_n$ creadas en la cláusula FROM cumplan con ella, justamente como se hace con el enunciado SELECT-FROM-WHERE.

La tercera etapa es la de la cláusula DELETE, aquí si la condición de la cláusula WHERE resulta verdadera o si no hay cláusula WHERE entonces se elige un X_i creado en la cláusula FROM. Este X_i posteriormente será eliminado junto con sus subdatos anidados.

Lo anterior es un ciclo de tres etapas, al igual que en el enunciado SELECT-FROM-WHERE se ejecutan tantos ciclos como combinaciones de valores de $X_1, ..., X_n$ sean posibles. En cada ciclo se elige un dato semiestructurado. Nótese que un mismo dato se puede elegir en más de un ciclo, pero esto no tiene importancia ya que de todas formas va a ser borrado. Al terminar todos los ciclos se tiene el conjunto de datos semiestructurados que fueron elegidos en los ciclos. Finalmente estos datos semiestructurados son los que se van a borrar junto con sus subdatos arbitrariamente anidados. Obsérvese que si se llega a borrar un dato semiestructurado raíz, la ssd-tabla correspondiente será eliminada. También obsérvese que la ejecución del enunciado DELETE-FROM-WHERE puede eliminar información en varias ssd-tablas.

Ejemplo 3.3.1 En el siguiente enunciado aplicado a la base de datos de la figura 3.22, se elimina el autor de algún articulo cuyo nombre sea "EAGC'.'

```
DELETE P
FROM   publicaciones.articulo AS P,
       P.autor.nombre AS N
WHERE  N="EAGC"
```

El resultado se muestra en la figura 3.31.

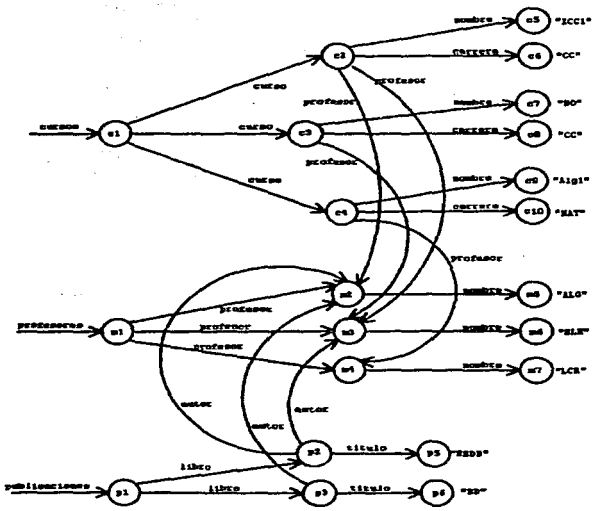


Figura 3.31: Resultado de la actualización del ejemplo 3.3.1.

Ejemplo 3.3.2 En el siguiente enunciado aplicado a la base de datos de la figura 3.22, se eliminan las publicaciones (libros o artículos) cuyo autor tenga nombre "ALG". El resultado se muestra en la figura 3.32.

```
DELETE P
FROM   publicaciones.# AS P,
       P.autor.nombre AS N
WHERE  N="ALG"
```

Nótese como al eliminar las publicaciones también se afecta a los subdatos que se encuentran en otras ssd-tablas.

Ejemplo 3.3.3 Al aplicar el siguiente enunciado a la base de datos de la figura 3.22, se elimina el autor de alguna publicación, si el nombre del autor es "ALG", obsérvese como los datos semiestructurados de otras ssd-tablas se eliminan.

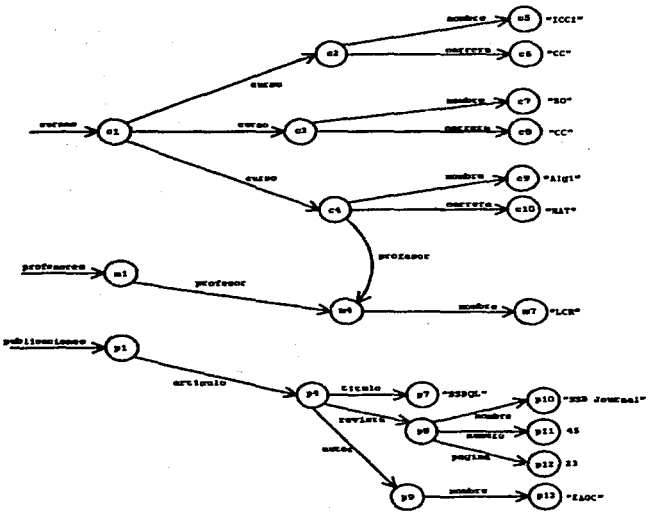


Figura 3.32: Resultado de la actualización del ejemplo 3.3.2.

```
DELETE M
FROM   publicaciones.# AS P,
       P.autor AS M,
       M.nombre AS N
WHERE  N="ALG"
```

El resultado se muestra en la figura 3.33.

3.3.2. Edición

Para cambiar el valor de los datos semiestructurados en una profundidad arbitraria se utiliza el enunciado *UPDATE-SET-FROM-WHERE* la sintaxis es:

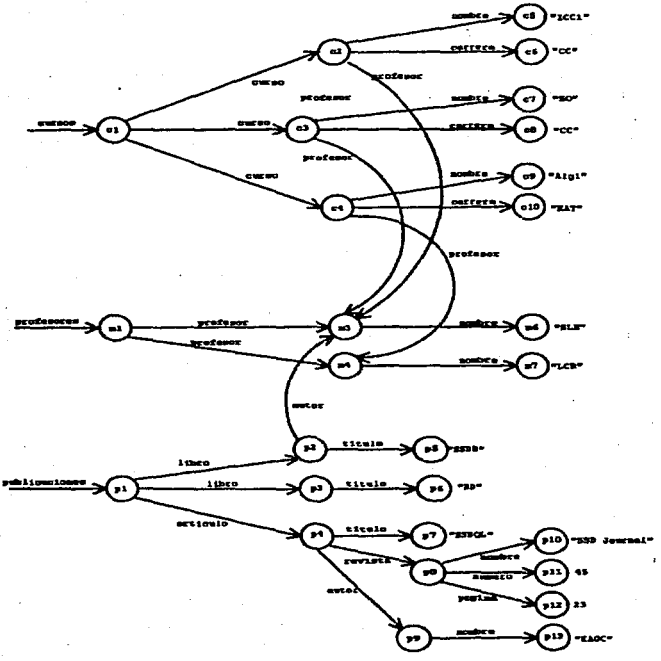


Figura 3.33: Resultado de la actualización del ejemplo 3.3.3.

UPDATE	X_i
SET	S
FROM	e_1 AS X_1, ..., e_n AS X_n
WHERE	*condition*

donde $e_1, ..., e_n$ son expresiones de camino, $X_1, ..., X_n$ son nombres de variables, *condition* es una condición y S es un dato semiestructurado con algunas restricciones que se explicarán a continuación. La cláusula WHERE es opcional, las cláusulas UPDATE, SET y FROM son obligatorias.

La sintaxis, al igual que la semántica, tiene similitudes con los enunciados anteriores. El enunciado UPDATE-SET-FROM-WHERE consta de 4 etapas.

La primera etapa es la dada por la cláusula FROM, de igual manera que con el enunciado SELECT-FROM-WHERE a partir de las expresiones de camino $e_1, ..., e_n$ se construyen las nuevas tablas temporales llamadas $X_1, ..., X_n$. Al igual que con el enunciado DELETE-FROM-WHERE, no se permite que las expresiones de camino se refieran a datos semiestructurados en una vista, esto es para evitar que las vistas puedan ser modificadas.

La segunda etapa es la de la cláusula WHERE, aquí hay una condición donde se verifica que las tablas $X_1, ..., X_n$ creadas en la cláusula FROM cumplan con ella, justamente como se hace con el enunciado SELECT-FROM-WHERE.

La tercera etapa es la de la cláusula UPDATE, aquí si la condición de la cláusula WHERE resulta verdadera o si no hay cláusula WHERE entonces se elige un X_i creado en la cláusula FROM, este X_i posteriormente sera modificado.

Lo anterior es un ciclo de tres etapas. Al igual que en las cláusulas anteriores se ejecutan tantos ciclos como combinaciones de valores de $X_1, ..., X_n$ sean posibles. En cada ciclo se elige un dato semiestructurado. Un mismo dato se puede elegir en más de un ciclo, pero esto no tiene importancia ya que de todas formas se va a reemplazar por el mismo valor. Al terminar todos los ciclos de tres etapas se tiene un conjunto de datos semiestructurados que fueron elegidos en los ciclos. Finalmente estos datos semiestructurados son los que se van a modificar, aquí es donde entra la cuarta etapa, la de la cláusula SET.

En la etapa de la cláusula SET aparece una construcción como la que se utiliza en la cláusula SELECT del enunciado SELECT-FROM-WHERE, pero a diferencia de esta la construcción no puede hacer referencia a todas las $X_1, ..., X_n$ de la cláusula FROM, solamente puede hacer referencia a la X_i dada en la cláusula UPDATE. En la etapa de la cláusula SET todos los datos semiestructurados elegidos van a ser reemplazados por la construcción dada. Esta construcción puede depender de X_i y el identificador va a ser el mismo que posee el dato a ser remplazado. Así los datos elegidos se van a reemplazar por otro valor.

La inserción de datos semiestructurados se puede ver como un caso particular de la edición. Para agregar un dato semiestructurado se utiliza el enunciado UPDATE-SET-FROM-WHERE, en la cláusula SET se debe utilizar una operación de unión entre el dato elegido por la cláusula UPDATE y lo que se quiera insertar en él.

Ejemplo 3.3.4 En el siguiente enunciado aplicado a la base de datos de la figura 3.22, lo que se hace es elegir el dato semiestructurado con identificador *c10*; su valor es reemplazado por "CC". Nótese como el identificador sigue siendo el mismo.

```
UPDATE C
SET    "CC"
FROM   cursos.curso AS X,
       X.nombre AS N,
       X.carrera AS C
WHERE  N="Alg1"
```

El estado de la base de datos después de la actualización se muestra en la figura 3.34.

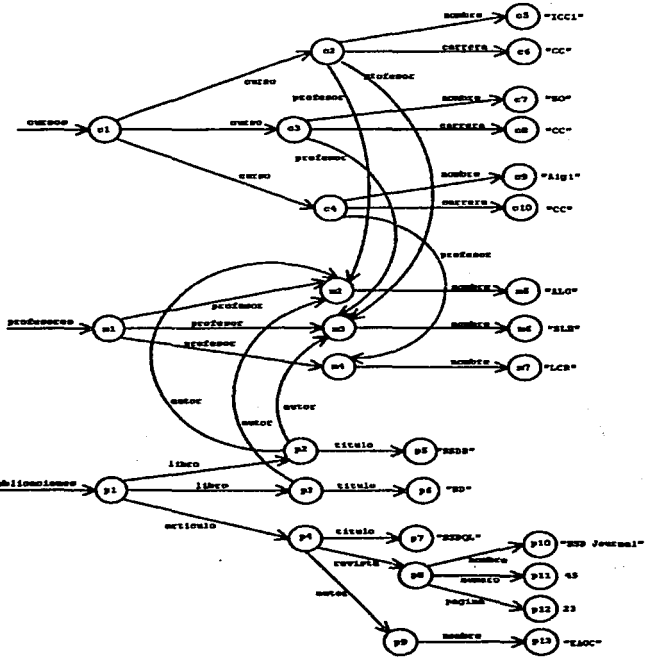


Figura 3.34: Resultado de la actualización del ejemplo 3.3.4.

Ejemplo 3.3.5 Lo que se hace en el enunciado

```
UPDATE X
SET    X UNION {carrera: "CC"}
FROM   cursos.curso AS X,
       X.nombre AS N
WHERE  N="Alg1"
```

aplicado a la base de datos de la figura 3.22, es seleccionar el dato semiestructurado con identificador $c4$, al que se le va a añadir un nuevo dato semiestructurado etiquetado como `carrera` y cuyo valor es "CC". El resultado se muestra en la figura 3.35.

Ejemplo 3.3.6 En el siguiente enunciado aplicado a la base de datos de la figura 3.22, se agrega un dato semiestructurado a la ssd-tabla de profesores. Éste va a ser el que corresponde al ya existente con identificador $p9$ pero etiquetado como `profesor`. El resultado después de la actualización se muestra en la figura 3.36.

```
UPDATE X
SET    X UNION (SELECT profesor: A
                FROM   publicaciones.articulo.autor AS A,
                       A.nombre AS N
                WHERE  N="EAGC")
FROM   profesores AS X
```

Ejemplo 3.3.7 En el siguiente enunciado aplicado a la base de datos de la figura 3.22, lo que se hace es quitar la referencia de autor a las publicaciones cuyo autor tenga nombre "ALG". El resultado se puede ver en la figura 3.37. La idea es similar a la del ejemplo 3.3.2, pero en este caso los datos semiestructurados de otras ssd-tablas no se ven afectados.

```
UPDATE P
SET    P TRIM(autor) UNION (SELECT autor: A
                            FROM   P.autor AS A,
                                   A.nombre AS N
                            WHERE N<>"ALG")
FROM   publicaciones.# AS P,
       P.autor AS M,
       M.nombre AS N
WHERE  N="ALG"
```

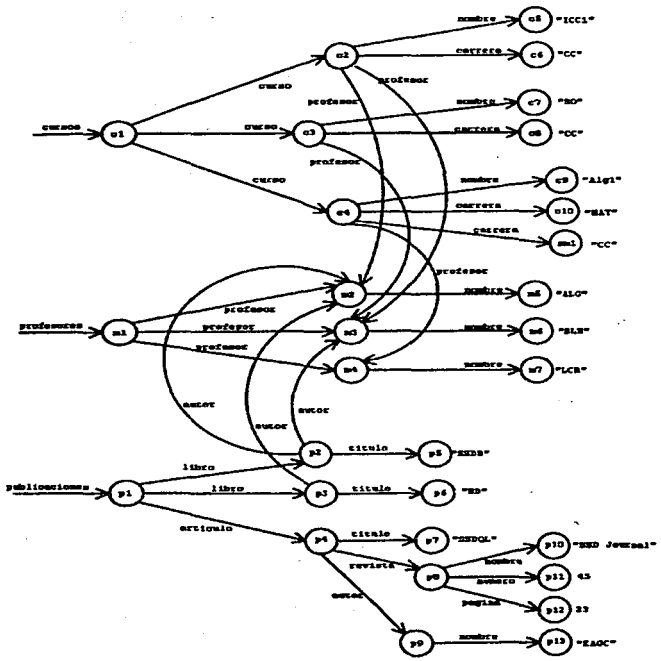


Figura 3.35: Resultado de la actualización del ejemplo 3.3.5.

3.4. Definición de datos

Es de suma utilidad proporcionar mecanismos para agregar información a la base de datos, ya sea para inicializarla o para enriquecerla, así como también dar mecanismos para eliminar la información que ya no sea útil y mecanismos para proteger la información del acceso del mundo. Por ello se introduce un mecanismo de definición de datos que permite la creación de nuevas ssd-tablas o vistas y la eliminación de las ya existentes.

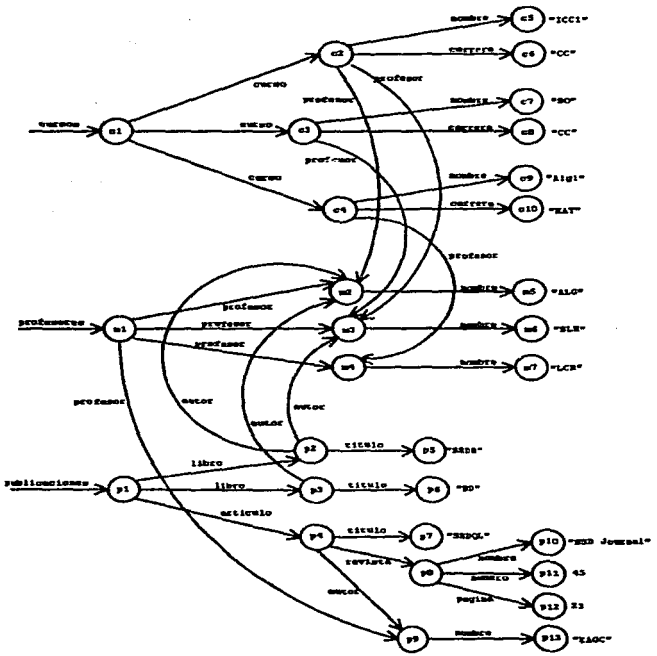


Figura 3.36: Resultado de la actualización del ejemplo 3.3.6.

3.4.1. Creación de ssd-tablas

Para crear una ssd-tabla se utiliza el enunciado *CREATE-SSDTABLE*. La sintaxis es la siguiente:

CREATE SSDTABLE l
WITH S

o

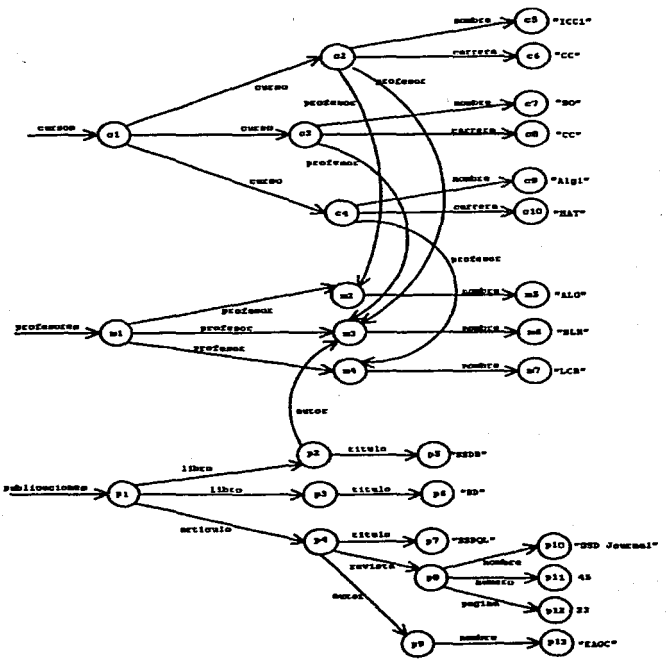


Figura 3.37: Resultado de la actualización del ejemplo 3.3.7.

CREATE SSDTABLE *l*
WITH FILE *file_name*

donde *l* es una etiqueta que va a ser el nombre de la nueva ssd-tabla, *S* es un dato semiestructurados que puede ser resultado de una consulta, el nombre de una ssd-tabla, el nombre de una vista existente o una constante. En la segunda forma del enunciado CREATE-SSDTABLE, *file_name* es la ubicación de un archivo que contiene la descripción de un dato semiestructurado. Con ello se tiene la opción de cargar un dato semiestructurado desde un archivo con un formato compatible con la descripción de datos semiestructurados, como puede ser XML, OEM o con la sintaxis presentada en

la sección 1.2 para describir datos semiestructurados.

En ambos casos se crea una nueva ssd-tabla llamada l. Por supuesto, no debe existir previamente una ssd-tabla con el mismo nombre en la base de datos. El contenido de la nueva ssd-tabla se especifica ya sea por medio de un dato semiestructurado obtenido a partir de la base de datos o por medio de un archivo.

Ejemplo 3.4.1 El enunciado:

```
CREATE SSDTABLE personas
WITH  {persona:{nombre:"EAGC", edad:23},
       persona:{nombre:"ALG"}}
```

aplicado a una nueva base de datos crea la ssd-tabla que se muestra en la figura 3.38.

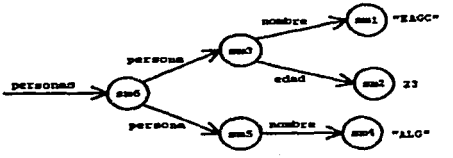


Figura 3.38: Resultado de la creación de una ssd-tabla en el ejemplo 3.4.1.

Ejemplo 3.4.2 El enunciado siguiente aplicado a la base de datos de la figura 3.22 crea la ssd-tabla `cursos2`. El resultado se muestra en la figura 3.39.

```
CREATE SSDTABLE cursos2
WITH  cursos
```

Obsérvese que las ssd-tablas `cursos` y `cursos2` se refieren al mismo dato semiestructurado raíz.

Ejemplo 3.4.3 El enunciado siguiente aplicado a la base de datos de la figura 3.22, crea la ssd-tabla `cursos2`. A diferencia del ejemplo 3.4.2, aquí primero se hace una copia del dato semiestructurado al que se refiere la ssd-tabla `cursos`, incluyendo los datos que se encuentran en otra ssd-tablas, así `cursos` y `cursos2` se refieren a datos semiestructurados distintos.

```
CREATE SSDTABLE cursos2
WITH  CLON cursos
```

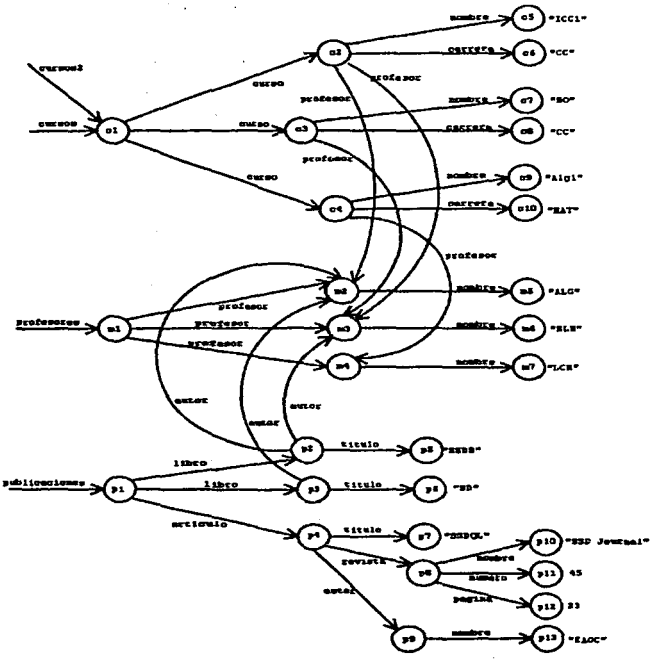


Figura 3.39: Resultado de la creación de una ssd-tabla en el ejemplo 3.4.2.

El resultado se muestra en la figura 3.40.

Ejemplo 3.4.4 El enunciado siguiente aplicado, a la base de datos de la figura 3.22, crea la ssd-tabla personas a partir de una consulta.

```
CREATE SSDTABLE personas
WITH   (SELECT persona: P
        FROM profesores.profesor AS P) UNION
       (SELECT persona: P
        FROM publicaciones.articulo.autor AS P)
```

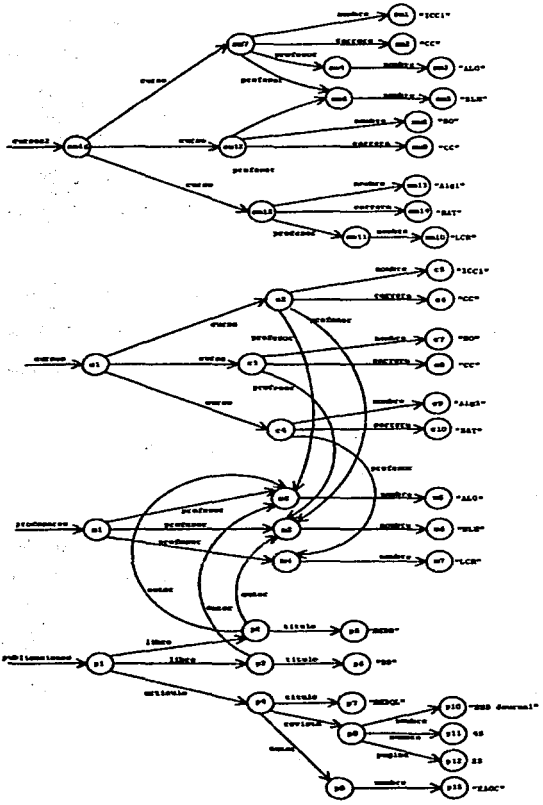


Figura 3.40: Resultado de la creación de una ssd-tabla en el ejemplo 3.4.3.

El resultado se muestra en la figura 3.41.

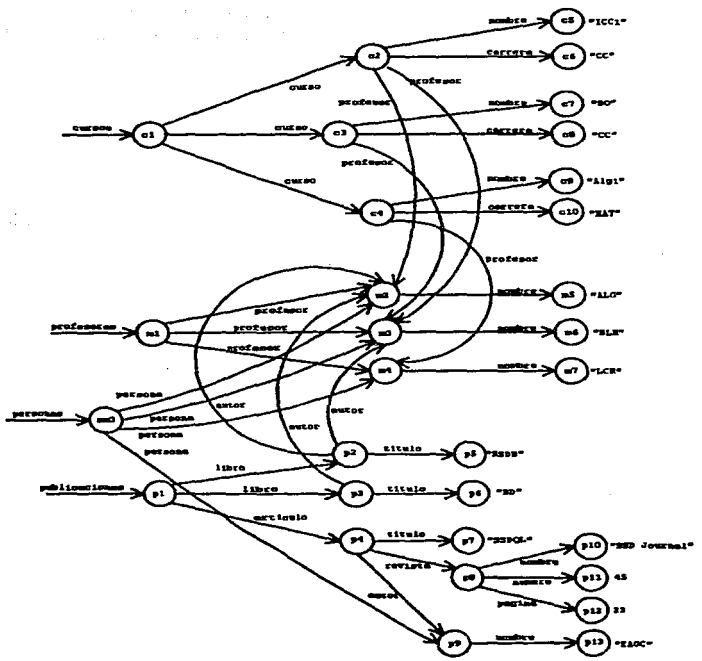


Figura 3.41: Resultado de la creación de una ssd-tabla en el ejemplo 3.4.4.

3.4.2. Destrucción de ssd-tablas

Para eliminar una ssd-tabla existente en una base de datos se usa el enunciado *DROP-SSDTABLE*. La sintaxis es la siguiente:

DROP SSDTABLE *l*

donde *l* es una etiqueta que da el nombre de la ssd-tabla que se va a eliminar.

Al ejecutar DROP SSDTABLE *l*, la ssd-tabla *l* se elimina de la base de datos y se

eliminan los datos semiestructurados que ya no sean referenciables; los datos semiestructurados que sean compartidos por otras ssd-tablas no se eliminan. Evidentemente debe existir previamente una ssd-tabla de nombre *l*.

Ejemplo 3.4.5 Con el enunciado:

```
DROP SSDTABLE cursos
```

aplicado a la base de datos de la figura 3.22, se elimina la ssd-tabla llamada **cursos**, el resultado se puede ver en la figura 3.42. Obsérvese cómo no se eliminan los datos semiestructurados que **cursos** comparte con otras ssd-tablas.

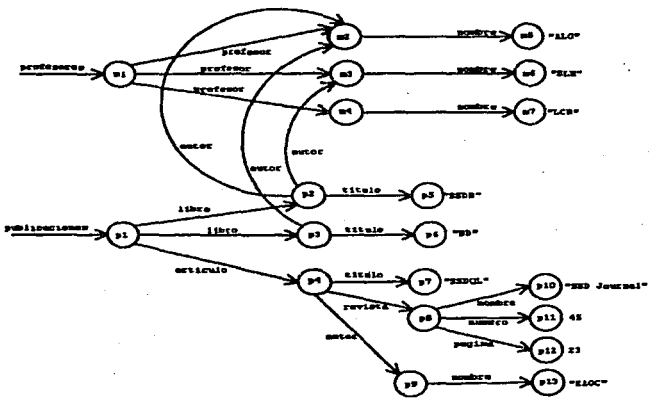


Figura 3.42: Resultado de la destrucción de una ssd-tabla en el ejemplo 3.4.5.

3.4.3. Creación de vistas

Como se vio en la sección 1.4.3 existen dos tipos de vistas: Las *vistas abstractas* y las *vistas materializadas*. Para crear una vista abstracta se utiliza el enunciado *CREATE-VIEW*. La sintaxis es la siguiente:

CREATE VIEW *l*
WITH *S*

Para crear una vista materializada se utiliza el enunciado *CREATE-MVIEW*, la sintaxis es la siguiente:

CREATE MVIEW *l*
WITH *S*

en ambos casos *l* es una etiqueta que va a ser el nombre de la nueva vista, *S* es un dato semiestructurados que puede ser resultado de una consulta, el nombre de una ssd-tabla, el nombre de una vista existente o una constante, *S* es la definición de la vista o fórmula para obtener la vista.

Ejemplo 3.4.6 Para crear las vistas dadas en el ejemplo 1.4.2 de la sección 1.4.3, en las cuales se utilizó la base de datos de la figura 1.5, se utilizan los siguientes enunciados:

```
CREATE VIEW v1
WITH (SELECT nombre: X
      FROM   profs.maestro.nombre AS X)
```

y

```
CREATE MVIEW v2
WITH (SELECT nombre: X
      FROM   profs.maestro.nombre AS X)
```

El resultado aparece en la figura 1.6;

3.4.4. Destrucción de vistas

Para eliminar una vista abstracta o materializada se utiliza el enunciado *DROP-VIEW*. La sintaxis es la siguiente:

DROP VIEW *l*

donde *l* es una etiqueta que da el nombre de la vista que se va a eliminar.

Al ejecutar DROP VIEW *l*, la vista *l* se elimina de la base de datos y se liberan los recursos que ya no sean necesarios. La vista de nombre *l* debe existir previamente.

3.5. Gramática

Para describir la sintaxis del lenguaje se utiliza la gramática libre de contexto[5] que se presenta a continuación. En realidad el lenguaje no es totalmente libre de contexto, lo cual quiere decir que la semántica no se puede derivar directamente de la sintaxis que se describe en la gramática, sino que también se deben considerar las restricciones semánticas explicadas a lo largo de este capítulo.

```
<statement> ::= <query-statement> |
                <modification-statement> |
                <data-definition-statement> |
                <view-statement>

<query-statement> ::= <ssd-construction>

<modification-statement> ::= <delete-statement> | <update-statement>

<data-definition-statement> ::= <create-statement> | <drop-statement>

<ssd-construction> ::= <ssd-table-name> |
                       <primitive-constant> |
                       <query> |
                       EMPTY |
                       ( <ssd-construction> ) |
                       <ssd-construction> UNION <ssd-construction> |
                       <ssd-construction> PICK (<list-labels>) |
                       <ssd-construction> CUT (<list-labels>) |
                       { <list-labeled-ssds } |
                       CLON <ssd-construction> |
                       <agregation-function> <ssd-construction> |
                       <ssd-construction> <primitive-op> <ssd-construction>

<list-labels> ::= <label> |
                  <label> , <list-labels>

<list-labeled-ssds> ::= <label> : <ssd-construction> |
                        <label> : <ssd-construction> , <list-labeled-ssds>

<agregation-function> ::= AVG | MAX | MIN | SUM | COUNT
```

[5]Es usual describir la sintaxis de un lenguaje de consulta o de programación mediante una gramática libre de contexto. Se consideran por separado las restricciones dependientes del contexto, la cuales se infieren de la semántica.

```
<query> ::= SELECT <label> : <ssd-construction>
            FROM <list-associations> |

            SELECT <label> : <ssd-construction>
            FROM <list-associations>
            WHERE <condition>

<delete-statemet> ::= DELETE <ssd-table-name>
                      FROM <list-associations> |

                      DELETE <ssd-table-name>
                      FROM <list-associations>
                      WHERE <condition>

<update-statement> ::= UPDATE <ssd-table-name>
                       SET <ssd-construction>
                       FROM <list-associations> |

                       UPDATE <ssd-table-name>
                       SET <ssd-construction>
                       FROM <list-associations>
                       WHERE <condition>

<create-statement> ::= CREATE SSDTABLE <ssd-table-name>
                       WITH <ssd-construction> |

                       CREATE SSDTABLE <ssd-table-name>
                       WITH FILE <string-constant>

<drop-statement> ::= DROP SSDTABLE <ssd-table-name>

<view-statement> ::= CREATE VIEW <ssd-table-name>
                     WITH <ssd-construction> |
                     CREATE MVIEW <ssd-table-name>
                     WITH <ssd-construction>

<drop-view-statement> ::= DROP VIEW <ssd-table-name> |
                          DROP MVIEW <ssd-table-name>

<list-associations> ::= <path-expression> AS <ssd-table-name> |
                        <path-expression> AS <ssd-table-name> , <list-associations>

<condition> ::= TRUE |
                FALSE |
                ( <condition> ) |
```

```
                <condition> AND <condition> |
                <condition> OR <condition> |
                NOT <condition> |
                <ssd-construction> <comp-op> <ssd-construction> |
                <ssd-construction> LIKE <string-constant> |
                PRIMITIVE <ssd-construction> |
                <ssd-construction> OWN <label> |
                FOR ALL <ssd-table-name> IN <ssd-construction> ( <condition> ) |
                EXIST <ssd-table-name> IN <ssd-construction> ( <condition> )


<primitive-op> ::= + | - | * | / | MOD

<comp-op> ::= < | <= | > | >= | = | <> |
              IS | CONTAIN | BELONG | ISOMORPH

<ssd-table-name> ::= <label>

<label> ::= <letters-digits>

<primitive-constant> ::= <numeric-constant> |
                         <string-constant>

<path-expression> ::= <pe-label> |
                      <path-expression>.<path-expression> |
                      (<path-expression>) |
                      <path-expression>|<path-expression> |
                      <path-expression>* |
                      <path-expression>+ |
                      <path-expression>?

<pe-label> ::= <label> | '<x-label>'

<x-label> ::= <letter> | <digit> | <wild_card> |
              <x-label><x-label> |
              (<x-label>) |
              <x-label>|<x-label> |
              <x-label>* |
              <x-label>+ |
              <x-label>?

<wild_card> ::= #

<numeric-constant> ::= <digits> |
                       <digits>.<digits> |
```

```
                        .<digits>

<string-constant> ::= "<caracters>" | ""

<letters-digits> ::= <letter> |
                     <digit> |
                     _ |
                     <letter><letters-digits> |
                     <digit><letters-digits> |
                     _<letters-digits>

<digits> ::= <digit> | <digit><digits>

<caracters> ::= <caracter> |
                \" |
                <caracter><caracters> |
                \"<caracteres>

<letter> ::= [A-Za-z]

<digit> ::= [0-9]

<caracter> ::= todos los caracteres ASCII imprimibles excepto "
```

ESTA TESIS NO SALE
DE LA BIBLIOTECA

Capítulo 4

XML

En la actualidad XML ha tomado gran importancia en la representación de información en una gran variedad de dominios, sobre todo en la Web. La idea es que en un futuro no muy lejano XML sea el formato que predomine en la Web. En este capítulo se presentan las principales caracteristicas de XML y algunos de los desarrollos tecnológicos a su alrededor.

4.1. Antecedentes

HTML (*HyperText Markup Language*) sin lugar a dudas fue un invento prodigioso. Es el lenguaje de presentación de documentos más exitoso de la historia. Gracias a HTML se ha podido publicar y acceder a una cantidad inimaginable de información. Sin embargo, debido a las altas exigencias que sobrevienen con el crecimiento acelerado de la Web, se provocó que HTML también evolucionara de una forma muy rápida y por desgracia no por el camino más adecuado. A pesar de todas las extensiones que se le han hecho a HTML, aún es un lenguaje rígido e inflexible.

4.1.1. ¿Qué es XML?

El Consorcio de la World Wide Web (W3C) en lugar de extender las capacidades de HTML decidió realizar un nuevo lenguaje que aprovechara todas las ventajas de HTML pero que al mismo tiempo permitiera una mayor flexibilidad. Este lenguaje o mejor dicho meta-lenguaje, es conocido como XML (*eXtensible Markup Language*).

El lenguaje HTML está diseñado para presentar información. Esto quiere decir

que las construcciones de HTML indican como se visualizará la información en el navegador[1]. La idea primordial de XML es la de representar únicamente información, sin tener que preocuparse por la presentación.

No se puede descartar totalmente la presentación de un documento XML; dicha presentación se puede incluir por separado en las llamadas *hojas de estilo*. Actualmente las hojas de estilo se pueden especificar en lenguajes como:

- CSS (*Cascade StyleSheet*) que se usa en HTML.
- DSSSL (*Document Style Semantics and Specification Language*) basado en *Scheme* y usado para SGML.
- XSL (*eXtensible Stylesheet Languaje*) con sintaxis XML, diseñado especialmente para XML.

A los documentos XML también se les puede dar una estructura fija, que se puede incluir por separado o dentro del documento XML a través de una DTD (*Document Type Definition*), que es similar a una gramática en BNF.

De esta manera con, XML se tiene un modelo trifásico en el que se separan la información, la estructura y la presentación.

4.1.2. Origen y objetivos

XML es una forma restringida de SGML (*Standard Generalized Markup Language*, ISO 8879). Fue desarrollado por un grupo de trabajo (originalmente conocido como "SGML Editorial Review Board") bajo los auspicios del Consorcio de la World Wide Web (W3C) en 1996. Fue presidido por Jon Bosak de Sun Microsystems con la participación activa de un grupo especial de interés en XML (previamente conocido como Grupo de Trabajo SGML) también organizado en el W3C. Los objetivos de XML [24] son:

1. Ser directamente utilizable en Internet.
2. Soportar una gran variedad de aplicaciones.
3. Ser compatible con SGML.

[1]Navegador se refiere al software para visualizar documentos HTML, pero no se refiere a alguna implementación en particular.

4. Sencillez en la creación de programas que procesen XML.
5. El número de capacidades opcionales de XML debería ser el mínimo, de ser posible cero.
6. Los documentos XML deberían ser legibles para las personas y razonablemente claros.
7. El diseño del XML debería finalizar de forma rápida.
8. XML debería ser simple pero perfectamente formalizado.
9. Los documentos XML deberían ser sencillos de crear.
10. El nombre de las marcas XML sería de mínima importancia.

4.1.3. XML vs HTML

De manera natural surge la inquietud por comparar a HTML y a XML, en realidad HTML, no puede competir con XML. Como ya se mencionó antes XML no es una extensión de HTML sino una versión reducida de SGML, con pequeñas diferencias. SGML es un meta-lenguaje usado para especificar lenguajes de marcado. HTML es una instancia de SGML. Por otro lado XML siendo un SGML reducido, es de la misma forma un meta-lenguaje para especificar lenguajes de marcado. HTML se puede especificar en términos XML, a esta especificación se la llama XHTML (eXtensible HyperText Markup Lenguage).

En HTML la estructura de los documentos ya está dada y su sintaxis está enfocada a la presentación del documento en el navegador. En XML la estructura de los documentos se define por el usuario. Un documento en XML se enfoca únicamente en la información, dicha información posteriormente se puede presentar transformándola a un lenguaje para presentación, como HTML.

Por ejemplo, si se necesita construir un sitio Web para vender productos a todo el mundo proporcionando el precio de los productos en la divisa del país del comprador, entonces se necesitaría obtener el valor de las divisa periódicamente. Suponiendo que cierto banco publica diariamente en su página el valor de las divisas, se necesitaría consultar la página de dicho banco para obtener el valor de la moneda y realizar la conversión. Si la página del banco tiene un formato HTML, habría que ubicar la porción del documento donde se encuentra el valor de la moneda que se requiere y con un programa extraer dicho valor. Pero si al banco se le ocurre modificar su página para que tenga una buena presentación habría que rehacer el programa que extrajera la información, con los problemas que eso implica. Si la página del banco se

encontrara en formato XML, el programa simplemente tendría que localizar la sección correspondiente a la información requerida en el documento XML sin importarle si se modifica la presentación de dicha página. En la localización incluso no importa que la información no se dé en un lugar exacto.

Con XML se extienden las capacidades de intercambio de información no sólo en la Web sino en un sinnúmero de aplicaciones.

4.2. Sintaxis básica

Un documento XML es simplemente un conjunto de cadenas de caracteres, se pueden diferenciar dos tipos de construcciones: el *marcado* y el *texto*.

Los caracteres reservados en XML son <, >, &, " (comillas) y ' (apóstrofo), los documentos XML son sensibles a mayúsculas y minúsculas.

El texto incluido entre los caracteres < y > o entre los signos & y ; es el marcado, que son las partes del documento XML que el procesador debe entender.

El marcado entre los símbolos < y > se denomina *marca*[2]. Las marcas son definidas por el usuario. Un ejemplo de marca es `<mi_marca>`. El resto es texto contenido en el documento.

Aunque XML es un documento de texto, se puede utilizar para elaborar documentos con otro tipo de contenido. Para ello se pueden usar referencias a sus componentes o una representación textual de ellos.

4.2.1. Elementos

El componente básico de XML es el *elemento*, que es una pieza de texto encerrada por una *marca-inicial* y su respectiva *marca-final*. Una marca-inicial tiene la forma `<mi_marca>` y una marca-final es de la forma `</mi_marca>`. El texto contenido entre una marca-inicial y su correspondiente marca-final, incluyendo a las marcas inmersas, es llamado elemento. Dentro de un elemento puede haber texto u otros elementos, las estructuras entre las marcas es el contenido. El término subelemento se usa para describir la relación entre un elemento y los elementos que lo componen.

Lo que no está encerrado entre los símbolos < y > corresponde al texto del do-

[2]Es la traducción que se manejará de la palabra *tag*.

cumento, el cual se conoce como *PCDATA* (Parsed Character Data), que ha sido cuidadosamente desarrollado para proveer el intercambio de datos en muchos lenguajes.

Un elemento sin contenido se dice que es un *elemento vacío*, los elementos vacíos se pueden abreviar con una sola marca de la forma `<mi_marca/>`.

4.2.2. Atributos

Los elementos pueden tener *atributos*, que describen propiedades que ofrecen información sobre el elemento. Los atributos son parejas nombre-valor que se colocan dentro de la marca-inicial correspondiente al elemento que posee dicho atributo. Después del nombre de la marca se coloca el nombre del atributo seguido por el signo = y el valor del atributo encerrado entre comillas, por ejemplo:

```
<mi_marca atributo1="valor1">
```

Al igual que con las marcas, el usuario puede definir atributos arbitrariamente, pero el valor de un atributo siempre es una cadena de caracteres.

Hay diferencias entre los atributos y las marcas. Un atributo dado solamente puede aparecer una vez dentro de una marca, mientras que los subelementos con una misma marca se pueden repetir. También a los atributos solamente se les asocia una cadena, mientras que la estructura contenida entre una marca-inicial y una marca-final puede contener subelementos. Los atributos revelan el origen de XML como un lenguaje de marcado.

4.2.3. Prólogo

Los documentos XML pueden empezar con un *prólogo*, en el que se da una definición XML y una declaración del tipo de documento (DTD), la cual se verá más adelante. En la declaración XML se proporciona información sobre la versión de XML que se está utilizando (por el momento solamente puede ser la versión 1.0) e información sobre el tipo de codificación de caracteres que se utiliza. Por ejemplo:

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

Ejemplo 4.2.1 El documento XML de la figura 4.1 tiene el propósito de guardar los datos climáticos de algunas ciudades. En cada ciudad se especifican sus datos climáticos agrupados por fecha.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="clima.xsl"?>
<!DOCTYPE clima SYSTEM "clima.dtd">

<clima>
    <ciudad nombre="Cd. de México">
        <fecha dia="02" mes="10" año="2000">
            <maxima>22</maxima>
            <minima>10</minima>
            <promedio>17</promedio>
            <estado>medio nublado</estado>
        </fecha>
        <fecha dia="03" mes="10" año="2000">
            <maxima>18</maxima>
            <minima>8</minima>
            <promedio>14</promedio>
            <estado>tormentas electricas</estado>
        </fecha>
    </ciudad>

    <ciudad nombre="Toluca">
        <fecha dia="02" mes="10" año="2000">
            <maxima>16</maxima>
            <minima>8</minima>
            <promedio>14</promedio>
            <estado>despejado</estado>
        </fecha>
    </ciudad>
</clima>
```

Figura 4.1: Documento XML del ejemplo 4.2.1.

4.2.4. Comentarios

Con los *comentarios* es posible proporcionar información que el procesador XML no tomará en cuenta. Los comentarios empiezan con los caracteres <!-- y terminan con -->. Se pueden colocar en cualquier sitio excepto dentro de las DTDs, marcas y otros comentarios. Por ejemplo:

```
<!-- Esto es un comentario -->
```

Ejemplo 4.2.2 En la figura 4.2 se presenta un documento XML que sirve para especificar los valores de algunas divisas en términos de otras.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Valores de algunas divisas -->

<monedas>
    <moneda>
        <nombre>Dólar</nombre>
        <valor fecha="18/10/2000" monedav="Peso">9.65</valor>
        <valor fecha="19/10/2000" monedav="Peso">9.57</valor>
    </moneda>

    <moneda>
        <nombre>Sol</nombre>
        <valor fecha="19/10/2000" monedav="Peso">3.10</valor>
    </moneda>

    <moneda>
        <nombre>Dólar Canadiense</nombre>
        <valor fecha="18/10/2000" monedav="Peso">6.40</valor>
        <valor fecha="18/10/2000" monedav="Peso">6.43</valor>
        <valor fecha="19/10/2000" monedav="Peso">6.38</valor>
        <valor fecha="19/10/2000" monedav="Dollar">1.6</valor>
    </moneda>
</monedas>
```

Figura 4.2: Documento XML del ejemplo 4.2.2.

4.2.5. CDATA

CDATA (*Character Data*) permite integrar en un documento XML texto que no será interpretado como marcado.

Los CDATA empiezan con los caracteres `<![CDATA[` y terminan con `]]>`. Dentro de ellos se puede colocar lo que sea excepto la cadena final del CDATA. Por ejemplo:

```
<![CADATA[ Lo que sea ] ]>
```

4.2.6. Entidades predefinidas

Para utilizar los caracteres reservados en los documentos XML se utilizan las entidades. Las entidades predefinidas son marcas que se utilizan para representar

caracteres especiales. El usuario puede crear las entidades que desee. En XML hay cinco entidades predefinidas:

1. & (&).
2. < (<).
3. > (>).
4. ' (').
5. " (").

Como se puede observar, las entidades se identifican por ir entre los símbolos & y ;.

4.2.7. Documentos XML bien formados

El W3C ha realizado una especificación de la sintaxis de un documento XML [24], si un documento XML sigue esa especificación, se dice que está *bien formado*.

En concreto, un documento XML está Bien Formado si satisface los tres puntos siguientes:

- Contiene al menos un elemento.
- Hay exactamente un elemento raíz.
- Las marcas están correctamente anidadas, es decir, si una marca-inicial está en un elemento, su respectiva marca-final está en el mismo elemento.

4.3. DTD

Opcionalmente los documentos XML pueden tener una *definición del tipo de documento DTD*, la cual no es más que la descripción de su gramática. Con una DTD se puede establecer la estructura para los documentos XML. En una DTD se usa una sintaxis parecida a la de BNF con expresiones regulares.

Una DTD se puede incluir en el mismo documento XML o bien en un archivo externo. Si se coloca dentro del mismo documento se utiliza la estructura

```
<!DOCTYPE elem_raiz [ ... ]>
```

después de la cláusula !DOCTYPE se coloca el nombre del elemento raíz, entre los paréntesis cuadrados se colocan las declaraciones de elementos, atributos y entidades.

Si la DTD se coloca en un archivo externo, se utiliza una construcción del tipo:

```
<!DOCTYPE elem_raiz SYSTEM "referencia URI">
```

en el archivo al que se refiere la URI se colocan las declaraciones de elementos, atributos y entidades.

4.3.1. Declaración de elementos en la DTD

Las declaraciones de elementos en una DTD definen el nombre y contenido para dicho elemento . Para el contenido se pueden especificar los subelementos que pueden ir dentro del elemento declarado. La declaración de un elemento se incluye dentro de los símbolos < y >, se incluye la cláusula !ELEMENT, el nombre del elemento y después el contenido de la siguiente manera:

- Si el elemento es vacío se coloca la palabra EMPTY.
- Si el contenido no tiene restricciones se coloca la palabra ANY.
- En cualquier otro caso, se indica el contenido entre paréntesis. Se coloca una lista de los elementos hijos con una sintaxis similar a la de las expresiones regulares. La palabra #PCDATA (*Parser Character Data*) se usa para indicar contenido de tipo texto.

A la hora de indicar los elementos hijos (los que están entre paréntesis), se utilizan los caracteres especiales +, *, ? y |, que indican el tipo de uso que se permite de estos elementos.

- + especifica el uso de uno o más elementos del tipo indicado.
- * especifica el uso de ninguno o más elementos del tipo indicado.
- ? especifica el uso de ninguno o un elemento del tipo indicado.
- | equivale a una disyunción lógica, es decir, da la opción de usar uno u otro elemento.

4.3.2. Declaración de atributos en la DTD

La declaración de atributos se coloca dentro de los símbolos < y >. Se coloca la cláusula `!ATTLIST` seguida por:

- El nombre del elemento.
- El nombre del atributo.
- Los posibles valores del atributo, de la siguiente manera:
  - Entre paréntesis y separados por el carácter | significa que el atributo puede tener uno y sólo uno de los valores dados.
  - La palabra `CDATA` se usa para indicar que se puede colocar cualquier valor.
  - La palabra `ID` indica que el valor que se le da es único, es decir un identificador.
  - La palabra `IDREF` se usa para colocar una referencia a un identificador
  - La palabra `IDREFS` se usa para colocar referencias por medio de una lista de identificadores separados por espacios.
- De forma opcional y entre comillas se coloca el valor por omisión del atributo.
- Por último se debe colocar alguna de las palabras:
  - `#REQUIRED`, si el uso del atributo es obligatorio.
  - `#IMPLIED` si el valor del atributo no es necesario.
  - `#FIXED` si el valor debe ser el mismo para cada instancia de un elemento.

Si se requiere más de un atributo, se coloca el nombre del otro atributo seguido de lo ya mencionado.

Ejemplo 4.3.1 En la figura 4.3 se presenta una DTD para el documento XML del ejemplo 4.1. Obsérvese como un elemento clima esta compuesto de una o más ciudades. Una ciudad puede tener cero o más fechas. Las fechas deben tener un promedio, un estado y opcionalmente una máxima y una mínima. El nombre de la ciudad es un atributo del elemento ciudad. El día, mes y año son atributos del elemento fecha.

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

<!ELEMENT clima (ciudad+)>
<!ELEMENT ciudad (fecha*)>
<!ELEMENT fecha (maxima?, minima?, promedio, estado)>
<!ELEMENT maxima (#PCDATA)>
<!ELEMENT minima (#PCDATA)>
<!ELEMENT promedio (#PCDATA)>
<!ELEMENT estado (#PCDATA)>
<!ATTLIST ciudad nombre CDATA #REQUIRED>
<!ATTLIST fecha dia CDATA #REQUIRED
                mes CDATA #REQUIRED
                año CDATA #REQUIRED>
```

Figura 4.3: DTD del documento XML de la figura 4.1.

4.3.3. Declaración de entidades en una DTD

Las entidades dotan de modularidad a los documentos XML. Las entidades predefinidas se usan para los caracteres especiales. Existe una lista de entidades predefinidas de caracteres ISO [24].

De igual forma que en las declaraciones anteriores se usan los símbolos < y > para declarar entidades, se utiliza la palabra `!ENTITY`, seguida del nombre de la entidad y de su valor, por ejemplo:

```
<!ENTITY mi_entidad "entidad XML">
```

En el documento se usa el nombre de la entidad encerrado entre los símbolos **&** y **;**, por ejemplo:

```
&mi_entidad;
```

Se puede declarar el valor de una entidad para referirse a un texto en un archivo externo, cuando se usa la entidad en el documento se incluye todo el texto. La forma de hacer esto es colocando la palabra `SYSTEM` después del nombre de la entidad y en su valor se coloca la referencia al URI del archivo, por ejemplo:

```
<!ENTITY mi_entidad SYSTEM "archivo1.xml">
```

Ejemplo 4.3.2 En la figura 4.4 se presenta un documento XML con DTD incluida. Su propósito es representar la información de ciertas personas y su relación familiar. Obsérvese como se usan las referencias a otros elementos por medio de los atributos.

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

<!DOCTYPE familia [
        <!ELEMENT familia (persona*)>
        <!ELEMENT persona (nombre, edad?, frase?)>
        <!ELEMENT nombre (#PCDATA)>
        <!ELEMENT edad (#PCDATA)>
        <!ELEMENT frase (#PCDATA)>
        <!ATTLIST persona  id       ID     #REQUIRED
                           genero   (M|F)  #REQUIRED
                           padre    IDREF  #IMPLIED
                           madre    IDREF  #IMPLIED
                           hermanos IDREFS #IMPLIED>
        <!ENTITY maxXML "XML es lo máximo">
]>

<familia>
    <persona id="o1" genero="F">
        <nombre>Maria</nombre>
        <edad>60</edad>
        <frase>&maxXML;</frase>
    </persona>
    <persona id="o2" genero="M">
        <nombre>José</nombre>
        <edad>75</edad>
        <frase>soy &gt; los demás</frase>
    </persona>
    <persona id="o3"  genero="M" padre="o2" madre="o1" hermanos="o4 o5">
        <nombre>Pedro</nombre>
        <edad>40</edad>
        <frase>Amor &amp; Paz</frase>
    </persona>
    <persona id="o4" genero="F" padre="o2" madre="o1" hermanos="o3 o5">
        <nombre>Ana</nombre>
        <edad>35</edad>
        <frase><![CDATA[<vida><diversión/></vida>]]></frase>
    </persona>
    <persona id="o5"  genero="M" padre="o2" madre="o1" hermanos="o3 o4">
        <nombre>Juan</nombre>
        <edad>30</edad>
        <frase></frase>
    </persona>
</familia>
```

Figura 4.4: Documento XML del ejemplo 4.3.2.

4.3.4. Documentos XML válidos

Un documento XML es *válido* si, además de ser bien formado, tiene una DTD con la cual cumple. Cabe mencionar que para que sea válido los valores de los atributos

de tipo ID deben ser distintos y los atributos de tipo IDREF y IDREFS deben referirse a identificadores existentes.

4.4. XSL

XSL (eXtensible Stylesheet Language) es un lenguaje que tiene el propósito de describir hojas de estilo para documentos XML, esto es con la finalidad de dar una representación (frecuentemente visual) a los documentos XML, para ello XSL se encarga de transformar documentos XML a otros formatos como PDF, Postscript, VRML, HTML, LaTeX, el mismo XML, etc.

Las hojas de estilo XSL son documentos XML en donde se especifica la manera en la que el documento de entrada se va a transformar en el documento de salida. Para transformar un documento XML se debe considerar la representación de árbol de éste, el árbol se empieza a transformar desde la raíz y a partir de ahí se sigue a los hijos de manera recursiva. Una hoja de estilo XSL está formada de *patrones*[3] y *plantillas*[4]. Los patrones indican lo que se va a transformar y las plantillas indican a que se van a transformar, aplicando las reglas de plantilla el árbol del documento de entrada se transforma en otro árbol para el documento de salida.

Un procesador XSL lleva a cabo dos acciones fundamentales:

- La construcción de un árbol de resultados a partir de un árbol de origen.
- La interpretación del árbol de resultados para fines de formato (presentación de la información).

Estas dos tareas fundamentales corresponden a dos desarrollos tecnológicos de XSL:

- XSLT (XSL Transformations).
- XSLFO (XSL Format Objects).

Estas dos tecnologías se implementan como vocabularios XML.

XSLT se utiliza para transformar documentos XML en otros documentos XML. XSLFO se utiliza para aplicar estilos de formato a los documentos XML con el fin de

[3]Patrón es la traducción que se maneja de *Pattern*.
[4]Plantilla es la traducción de *Template*.

visualizarlos. XSLFO aún se encuentra en fases de desarrollo. Para visualizar documentos XML, la alternativa más usual es transformarlos en documentos HTML por medio de XSLT, pero como HTML no sigue las reglas de XML lo que se utiliza es XHTML, XHTML es la reformulación de HTML para que sea un vocabulario XML.

XSLT es un vocabulario XML para crear hojas de estilo. Como ya se mencionó, las hojas de estilo constan de patrones y plantillas. XSLT utiliza la tecnología XPath para seleccionar los elementos que serán procesados. XPath no es un vocabulario XML, sino que se utiliza para dirigirse a partes de un documento XML.

Para especificar una hoja de estilo en XSL se utiliza el elemento `xsl:stylesheet`, que es el elemento raíz de la hoja de estilo. En el se incluirán todos los patrones y plantillas.

Una plantilla es una estructura XSL que describe la salida a generar. Las plantillas se definen utilizando el elemento `xsl:template`, este elemento es un contenedor de patrones y datos de transformación. El elemento `xsl:template` utiliza un atributo opcional llamado `match` para seleccionar los nodos del árbol XML de entrada que se van a transformar, el valor del atributo `match` es un patrón.

Los patrones son los que determinan las posiciones del documento XML que se van a pasar a través de la plantilla. Un patrón describe un nodo o un conjunto de nodos del árbol XML. La sintaxis que usan los patrones XSL es parecida a la que se utiliza para especificar las rutas en un sistema de directorios: Al inicio de un patrón la barra `/` indica la raíz, el operador `/` selecciona al hijo inmediato y el operador `//` selecciona a los descendientes de un elemento. La sintaxis general es:

```
elemento[filtro]
```

donde el elemento se elige con los operadores `/` y `//` y la marca XML de los elementos. El filtro es una condición que debe cumplir el elemento.

En XSLT se definen varias construcciones que controlan la aplicación de las plantillas en las hojas de estilo XSL.

- El elemento `xsl:value-of`. Se usa para insertar el valor de un elemento o atributo en el documento de salida.
- El elemento `xsl:if`. Se usa para llevar a cabo cotejos condicionales. Este elemento utiliza el atributo `match` para establecer la condición, el valor de `match` es un patrón. Dentro de este elemento se encuentran los datos de transformación que se aplicarán a un elemento si se satisface la condición.

- El elemento `xsl:for-each`. Se utiliza para aplicar una transformación a cada uno de los elementos dados por el patrón que se encuentra en su atributo `select`.
- El elemento `xsl:apply-templates`. Se usa para aplicar plantillas a los elementos dados por el atributo `select`. De esta manera se transforman los documentos recursivamente.
- El elemento `xsl:choose`. Se utiliza para un cotejo condicional múltiple junto con los elementos `xsl:when` y `xsl:otherwise`. En `xsl:when` se utiliza el atributo `test` que contiene un patrón correspondiente a una condición. El elemento `xsl:when` contiene las transformaciones que se realizarán al elemento en caso de que la condición resulte verdadera. El elemento `xsl:otherwise` contiene las transformaciones que se aplicaran en caso de que ninguna de las condiciones de los elementos `xsl:when` se cumpla.

Ejemplo 4.4.1 En las figuras 4.6 y 4.7 se muestra una hoja de estilo para el documento del ejemplo 4.2.1. En la figura 4.5 se muestra como se visualiza en un navegador.

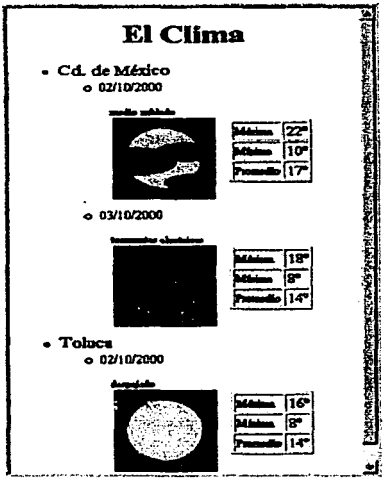


Figura 4.5: Presentación del documento XML del ejemplo 4.2.1 con la hoja de estilo del ejemplo 4.4.1.

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <HTML>
      <BODY STYLE="color=black">
        <CENTER><H1>El Clima</H1></CENTER>
        <UL><xsl:apply-templates select="clima/ciudad"/></UL>
      </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="ciudad">
    <LI/><SPAN STYLE="color=blue;font-size=14pt"><xsl:value-of select="@nombre"/></SPAN>
    <UL><xsl:apply-templates select="fecha"/></UL>
  </xsl:template>

  <xsl:template match="fecha">
    <LI/>
    <SPAN STYLE="color=green;font-size=11pt">
      <xsl:value-of select="@dia"/>/<xsl:value-of select="@mes"/>/<!--
      --><xsl:value-of select="@año"/>
    </SPAN>
    <TABLE CELLSPACING="10">
      <TR>
        <TD>
          <SPAN STYLE="font-size=8pt"><xsl:value-of select="estado"/></SPAN>
          <xsl:choose>
            <xsl:when test="estado = 'despejado'">
              <IMG SRC="Soleado.jpg" ALIGN="left" BORDER="1"/>
            </xsl:when>
            <xsl:when test="estado = 'medio nublado'">
              <IMG SRC="MNublado.jpg" ALIGN="left" BORDER="1"/>
            </xsl:when>
            <xsl:when test="estado = 'tormentas electricas'">
              <IMG SRC="TElectrica.jpg" ALIGN="left" BORDER="1"/>
            </xsl:when>
          </xsl:choose>
        </TD>
        <TD>
          <TABLE BORDER="1">
            <xsl:apply-templates select="maxima"/>
            <xsl:apply-templates select="minima"/>
            <xsl:apply-templates select="promedio"/>
          </TABLE>
        </TD>
      </TR>
    </TABLE>
  </xsl:template>
```

Figura 4.6: Hoja de estilo XSLT para el documento XML del ejemplo 4.2.1 (parte 1).

```
  <xsl:template match="maxima">
    <TR>
      <TD><SPAN STYLE="font-size=9pt">Máxima</SPAN></TD>
      <TD><xsl:value-of select="."/>°</TD>
    </TR>
  </xsl:template>

  <xsl:template match="minima">
    <TR>
      <TD><SPAN STYLE="font-size=9pt">Mínima</SPAN></TD>
      <TD><xsl:value-of select="."/>°</TD>
    </TR>
  </xsl:template>

  <xsl:template match="promedio">
    <TR>
      <TD><SPAN STYLE="font-size=9pt">Promedio</SPAN></TD>
      <TD><SPAN STYLE="color=red"><xsl:value-of select="."/>°</SPAN></TD>
    </TR>
  </xsl:template>

  <xsl:template match="text()"><xsl:value-of select="."/></xsl:template>
</xsl:stylesheet>
```

Figura 4.7: Hoja de estilo XSLT para el documento XML del ejemplo 4.2.1 (parte 2).

Capítulo 5

XML y datos semiestructurados

Los datos semiestructurados y XML presentan ciertas semejanzas que se explican en este cápitulo. De una manera natural se puede pasar de datos semiestructurados a documentos XML y viceversa.

Como se verá más adelante se puede considerar que los documentos XML son un caso particular de datos semiestructurados. Esto quiere decir que el estudio de los datos semiestructurados es aplicable a los documentos XML.

5.1. Datos semiestructurados a XML

La sintaxis básica de XML, se aplica de manera sencilla para representar datos semiestructurados. A partir de la sintaxis dada en la sección 1.2, se puede dar una fórmula para convertir un dato semiestructurado representado en dicha sintaxis a un documento XML. La fórmula es la siguiente:

$$\begin{aligned}
T_1(l:v) &= <l> T_2(v) </l> \\
T_1(l:\&o\ v) &= <l\ \ id="o"> T_2(v) </l> \\
T_1(l:\&o) &= <l\ \ ref="o"/> \\
T_2(valorPrimitivo) &= valorPrimitivo \\
T_2(d_1,...,d_n) &= T_1(d_1)...T_1(d_n) \\
T(s) &= <etiqueta_s> T_2(s) </etiqueta_s> \\
T(\&o\ s) &= <etiqueta_s\ \ id="o"> T_2(s) </etiqueta_s>
\end{aligned}$$

En donde l es una etiqueta; o es un identificador; v es un dato semiestructurado y las d_i pueden ser parejas etiqueta-valor (de la forma $l:v$), etiqueta-identificador (de

la forma $l : \&o$) o triadas etiqueta-identificador-valor (de la forma $l : \&o\ v$); s es una ssd-tabla y *etiqueta*$_s$ representa la etiqueta de s. T es la función que transforma una ssd-tabla a un documento XML.

Cada dato semiestructurado se transforma en un elemento XML, donde la etiqueta del dato semiestructurado se convierte en la marca del elemento XML. Cuando se definen identificadores o se hace referencia a ellos se utilizan los atributos `id` y `ref` para indicar las referencias en el documento XML. Estos atributos se pueden definir como de tipo `ID` o `IDREF` en la DTD del documento XML.

Hay que tomar en consideración que los elementos en XML poseen un orden. Dicho orden no existe en los datos semiestructurados, así que al pasar un dato semiestructurado a XML se le da de un orden. Sin embargo esto no es un problema grave, ya que no se pierde información alguna.

Otra cuestión que hay que considerar es que los documentos XML tienen una estructura de árbol, en la cual cada elemento contiene atributos, texto u otros elementos. Aunque los atributos de alguna manera pueden manejar identificadores y referencias a otros elementos para darle una estructura de gráfica, la realidad es que la mayoría de las aplicaciones (si no es que todas) consideran un documento XML como un árbol.

Ejemplo 5.1.1 Considérese la siguiente ssd-tabla cuyo nombre es `países`

```
{ pais: { nombre:  "México",
          capital: "Cd. de México",
          moneda:  "Peso",
          idioma:  "Español",
        },
  pais: { nombre:  "España",
          capital: "Madrid",
          moneda:  "Peseta",
          moneda:  "Euro",
          idioma:  "Español",
        },
  pais: { nombre:  "Canadá",
          capital: "Otawa",
          moneda:  "Dólar canadiense",
          idioma:  "Inglés",
          idioma:  "Francés",
        }
}
```

su representación en XML es:

```
<paises>
    <pais>
        <nombre>México</nombre>
        <capital>Cd. de México</capital>
        <moneda>Peso</moneda>
        <idioma>Español</idioma>
    </pais>
    <pais>
        <nombre>España</nombre>
        <capital>Madrid</capital>
        <moneda>Peseta</moneda>
        <moneda>Euro</moneda>
        <idioma>Español</idioma>
    </pais>
    <pais>
        <nombre>Canadá</nombre>
        <capital>Otawa</capital>
        <moneda>Dólar Canadiense</moneda>
        <idioma>Inglés</idioma>
        <idioma>Francés</idioma>
    </pais>
</paises>
```

5.2. XML a datos semiestructurados

Los documentos XML poseen una estructura de árbol, los datos semiestructurados se pueden asociar con una estructura de gráfica dirigida con raíz. Desde este punto de vista un documento XML es un caso particular de dato semiestructurado.

En la estructura de árbol de XML el texto, los atributos y los elementos vacíos corresponden a nodos terminales u hojas; los elementos no vacíos corresponden a nodos no terminales, cuyos hijos son nodos correspondientes a otros elementos, atributos o texto.

Se puede transformar un documento XML a un dato semiestructurado de manera recursiva iniciando por el elemento raíz y continuando con los hijos. Cada nodo del árbol XML se va a transformar en un dato semiestructurado de la siguiente manera:

- Si el nodo corresponde a un elemento, la etiqueta del dato semiestructurado va a ser la marca del elemento. Si el elemento es vacío, el dato semiestructurado correspondiente será el conjunto vacío. Si el elemento es no vacío, el dato semiestructurado correspondiente contendrá a los nodos hijos transformados en datos semiestructurados.
- Si el nodo corresponde a un atributo, el dato semiestructurado correspondiente será de tipo primitivo, su etiqueta será el nombre del atributo y su valor será el valor del atributo, de acuerdo a los tipos primitivos que se requieran.
- Si el nodo corresponde a texto, el dato semiestructurado correspondiente será de tipo primitivo, el valor será el mismo texto y la etiqueta debe ser alguna previamente definida para representar texto, por ejemplo pcdata.

Ejemplo 5.2.1 Considérese el siguiente documento XML:

```
<monedas>
    <moneda>
        <nombre>Dolar</nombre>
        <valor fecha="18/10/2000" monedav="Peso">9.65</valor>
        <valor fecha="19/10/2000" monedav="Peso">9.57</valor>
    </moneda>
    <moneda>
        <nombre>Sol</nombre>
        <valor fecha="19/10/2000" monedav="Peso">3.10</valor>
    </moneda>
</monedas>
```

Si se transforma a dato semiestructurado de acuerdo al proceso anterior, se obtiene el siguiente resultado:

```
monedas: { moneda: { nombre: "Dolar",
                     valor: { fecha: "18/10/2000",
                              monedav: "Peso",
                              pcdata: "9.65" },
                     valor: { fecha: "19/10/2000",
                              monedav: "Peso",
                              pcdata: "9.57" },
           moneda: { nombre: "Sol",
                     valor: { fecha: "19/10/2000",
                              monedav: "Peso",
                              pcdata: "3.10" }
         }
```

Hay que tener en cuenta que cuando se pasa un documento XML a un dato semiestructurado el orden se pierde, aunque éste no es un gran problema. Si se deseara se podría solucionar simplemente agregando a cada dato semiestructurado un subdato primitivo cuyo valor sea el índice o posición que ocupaba en el documento XML.

A través de una DTD, XML posee medios para definir identificadores y referencias de elementos, utilizando atributos. Para garantizar que un documento XML describa claramente un dato semiestructurado con todo y referencias e identificadores, es necesario definir claramente una DTD para este propósito. Con ello se podría convertir de forma más precisa un documento XML en un dato semiestructurado, pero esto sólo contemplaría a un subconjunto de los documentos XML, dejando fuera a los que no cumplan con la DTD (que serían demasiados), por lo cual no es una opción viable a menos que la aplicación así lo requiera.

5.3. Documentos XML como almacenes de datos

Hasta ahora XML se ha considerado como un lenguaje para describir documentos, pero nada impide que XML también pueda ser usado para almacenar y tratar información. Desde este punto de vista lo que se tiene son *datos XML*. La diferencia entre documento y dato la da la manera en la que se usa el contenido. El tratar el contenido en forma de datos implica una forma de recuperar o manipular la información, que puede ser a través de un lenguaje de consulta. En los documentos la información permanece casi estática.

Se han propuesto algunos lenguajes de consulta para XML y una gran variedad de sistemas manejadores de bases de datos incorporan mecanismos para el manejo de información en XML.

El utilizar XML como un formato para almacenes de datos proporciona algunas ventajas para el intercambio de información en una gran variedad de aplicaciones, principalmente en la Web.

5.4. Otros lenguajes de consulta

La intensión de esta sección es presentar un breve análisis de los lenguajes XQL, XML-QL, Lorel y XSL, mencionando sus ventajas y desventajas, con el fin de que el lector pueda tener un juicio comparativo con el lenguaje *Ssquirel* presentado en el capítulo 3 y el cual es el tema principal de este trabajo. La sección se concluye presentando un cuadro comparativo que resume las carácteristicas mencionadas

(cuadro 5.1). La mayoría de los lenguajes de consulta que se presentan a continuación no fueron diseñados para el manejo de datos semiestructurados, sino de XML, pero es importante considerarlos debido a las similitudes existentes entre XML y datos semiestructurados. Aquí no se presentan todos los lenguajes de consulta para XML, sino sólo se presentan los más representativos.

5.4.1. XQL

XQL es una sintaxis para seleccionar y filtrar elementos y texto de los documentos XML. XQL se puede considerar como una extensión de la sintaxis para los patrones que utiliza XSL y fue diseñado con la finalidad de ser muy simple y compacto [13].

En XQL se considera un documento XML con una estructura de árbol. Lo que hace una consulta en XQL es devolver un conjunto de nodos del árbol, pero no se indica la manera en que estos son devueltos. La mayoría de las implementaciones los devuelven como un documento XML bien formado, en el cual un elemento raíz es el que engloba a los regresados por la consulta.

El conjunto de elementos en donde se realiza la consulta se llama contexto de la búsqueda. El contexto de la búsqueda puede constar de uno o varios documentos XML o un fragmento bien construido que contenga un elemento raíz.

Una consulta es una cadena construida en función de la sintaxis de XQL. Una consulta consta de un nodo para buscar y de filtros. La sintaxis general es:

```
NodoDeBusqueda[filtros]
```

Para elegir el nodo se utilizan los operadores **/** y **//**. El primero indica una relación padre a hijo y el segundo una relación ancestro a descendiente. El símbolo * es un comodín que se puede reemplazar por cualquier nombre de un elemento. Para devolver los atributos se usa el símbolo @ junto con el nombre del atributo.

Los filtros se especifican entre corchetes ([]). Cuando se añade un filtro a un nodo se añade una cláusula booleana a éste, sólo si la evaluación resulta verdadera el nodo se devolverá. Las instrucciones booleanas utilizan los operadores:

- and. Conjunción lógica.
- or. Disyunción lógica.
- not. Negación lógica.

- eq o =.
- ne o ! =.
- lt. <.
- le. ≤.
- gt. >.
- ge. ≥.

En los filtros se pueden usar varios métodos que permiten la manipulación de los nodos.

- `text()`. Devuelve el texto que hay en un elemento.
- `value()`. Devuelve el valor que hay en un elemento en caso de que manejen otros tipos, si no hay otros tipos es equivalente a `text()`.
- `nodeType()`. Devuelve un valor entero que representa el tipo del nodo: 1 para elemento, 2 para atributo, 3 para texto, 8 para comentario, etc.
- `nodeName()`. Devuelve el nombre de la marca del elemento.
- `index()`. Devuelve la posición de un nodo dentro de su padre.
- `end()`. Devuelve el último nodo de un grupo de nodos iguales.

Ejemplo 5.4.1 Considérese el documento XML de la figura 5.1. La consulta:

```
clima/ciudad/fecha/promedio[value()$gt$15]
```

puede producir el siguiente resultado:

```
<xql:result>
    <promedio>17</promedio>
    <promedio>30</promedio>
</xql:result>
```

Ejemplo 5.4.2 La consulta siguiente:

```
clima/ciudad/@nombre[fecha!estado!text()="Despejado"]
```

```
<clima>
    <ciudad nombre="Cd. de México">
        <fecha dia="02" mes="10" año="2000">
            <maxima>22</maxima>
            <minima>10</minima>
            <promedio>17</promedio>
            <estado>Nublado</estado>
        </fecha>
    </ciudad>

    <ciudad nombre="Zacatecas">
        <fecha dia="02" mes="10" año="2000">
            <maxima>37</maxima>
            <minima>9</minima>
            <promedio>30</promedio>
            <estado>Despejado</estado>
        </fecha>
    </ciudad>

    <ciudad nombre="Veracruz">
        <fecha dia="02" mes="10" año="2000">
            <maxima>24</maxima>
            <minima>10</minima>
            <promedio>15</promedio>
            <estado>Huracan</estado>
        </fecha>
    </ciudad>
</clima>
```

Figura 5.1: Documento XML.

aplicada al documento XML de la figura 5.1 puede producir el siguiente resultado:

```
<xql:result>
    <nombre>Zacatecas</nombre>
</xql:result>
```

XQL es muy sencillo y compacto, pero su principal debilidad es su poder expresivo, el cual es comparable al de las expresiones de camino que se utilizan como base para otros lenguajes, entre ellos Ssquirel. XQL es un lenguaje de consulta para XML, XQL basa su búsqueda en la estructura de árbol de XML, los datos semiestructurados más bien se equiparan con una estructura de gráfica, lo cual de alguna manera lo limita para las consultas en datos semiestructurados. A diferencia de Ssquirel, XQL no ofrece capacidades para construir nuevos datos o modificar los ya existentes.

5.4.2. XML-QL

XML-QL es un lenguaje de consulta para documentos XML. XML-QL combina la sintaxis de XML con las técnicas de otros lenguaje de consulta [1]. Se utilizan en cierto sentido las expresiones de camino para extraer información del dato XML de entrada, se utilizan variables para enlazar la información extraída y se utilizan plantillas para construir la información de salida.

XML-QL está basado en el enunciado `WHERE-CONSTRUCT` en lugar del tradicional `SELECT-FROM-WHERE`. En primer lugar se ejecuta la cláusula `WHERE`, posteriormente se ejecuta la cláusula `CONSTRUCT`. La cláusula `CONSTRUCT` desempeña un papel equivalente a `SELECT` y la cláusula `WHERE` es una combinación de `FROM` y `WHERE`.

La cláusula `WHERE` consta de un patrón cuya sintaxis es la de XML. Es aquí donde se pueden colocar variables, que están precedidas por el símbolo $. El procesador de consultas intenta encontrar la información que se ajuste al patrón dado. Se toma en cuenta que las variables no tienen un valor predefinido. Las variables se pueden usar para sustituir elementos, atributos y marcas.

La cláusula `CONSTRUCT` consta de una plantilla. También de acuerdo a la sintaxis de XML, en dicha plantilla se colocan las variables dadas en la cláusula `WHERE` y con ellas se construyen nuevos datos.

En XML-QL se pueden anidar las consultas, un enunciado `WHERE-CONSTRUCT` puede ser colocado dentro de la cláusula `WHERE` de otra consulta.

XML-QL cuenta con algunas formas para utilizar expresiones regulares en los patrones, con el fin de explorar los datos a profundidades arbitrarias.

En XML-QL existen mecanismos para manipular el orden lineal presente en los documentos XML.

Ejemplo 5.4.3 La consulta siguiente:

```
<ciudades>
  where <clima>
          <ciudad nombre=$N>
              $X
          </ciudad>
        </clima> in "clima.xml"
  construct
        <nombre_ciudad>$N</nombre_ciudad>
</ciudades>
```

aplicada al documento XML de la figura 5.1 produce el siguiente resultado:

```
<ciudades>
    <nombre_ciudad>Cd. de México</nombre_ciudad>
    <nombre_ciudad>Zacatecas</nombre_ciudad>
    <nombre_ciudad>Veracruz</nombre_ciudad>
</ciudades>
```

XML-QL posee una sintaxis sencilla y clara, aunque puede haber cierta confusión en la forma en la que el procesador debe igualar los patrones, ya que no se busca una coincidencia exacta sino más bien a la manera de un cuantificador existencial. La forma de construir nuevos datos es poderosa, aunque no se garantiza que el resultado de una consulta sea un documento XML bien formado. XML-QL asume una estructura de árbol para los documentos XML, por lo cual puede presentar ciertas limitaciones para el manejo de datos semiestructurados, que se pueden asociar con una estructura de gráfica dirigida con raíz y en los cuales se basa Ssquirel. XML-QL ofrece cierta flexibilidad para explorar los datos a profundidades arbitraria, pero la realidad es que las expresiones de camino en otros lenguajes (entre ellos Ssquirel) brindan un mayor poder para este fin.

XML-QL no posee funciones de agregación, ni operaciones de conjunto y sus reglas de coerción son muy limitadas. Por ello su poder expresivo se ve superado por el de Ssquirel. XML-QL tampoco posee construcciones para la modificación de información existente.

5.4.3. Lorel

Lorel es el lenguaje de consulta del sistema Lore (*Light Object Repository*). Fue concebido e implementado en la Universidad de Stanford. Lorel fue diseñado originalmente para las consultas en datos semiestructurados, pero se ha extendido para el manejo de XML [1, 3].

Lorel es una adaptación de OQL para el manejo de datos semiestructurados. Las consultas en Lorel están basadas en el enunciado `SELECT-FROM-WHERE`. La semántica de esta construcción está dada en tres pasos: el primero resuelve la cláusula `FROM`, el segundo la cláusula `WHERE` y el tercero la cláusula `SELECT`. Las cláusulas `FROM` y `WHERE` no son obligatorias.

En la cláusula `FROM` aparecen expresiones de camino cuyo propósito es seleccionar ciertos objetos que son enlazados a variables.

En la cláusula WHERE, los objetos elegidos en la cláusula FROM pasan por una nueva selección por medio de una condición. En esta parte se utilizan las reglas de coerción al estilo de OQL, que pueden hacer uso de una gran variedad de operadores.

La cláusula SELECT hace una nueva selección de los objetos resultantes de las cláusulas FROM y WHERE, aquí se pueden aplicar algunas construcciones para formar nuevos datos, también se pueden usar expresiones de camino para aplicar otra selección. El resultado de la consulta es un dato semiestructurado que contiene a los que se construyen en la cláusula SELECT.

De igual manera que en Ssquirel y otros lenguajes de consulta, las consultas en Lorel se pueden anidar.

Si los datos contienen ciclos, entonces una consulta puede encontrarse con un número infinito de caminos de datos. Lorel cuenta con dos modos para mostrar las respuestas: En el primero se siguen las referencias a objetos y se muestran los objetos que les corresponden; en el segundo únicamente se muestran las referencias a los objetos.

Lorel cuenta con mecanismos para crear nueva información y para actualizar la ya existente. Para actualizar la información se utiliza el enunciado UPDATE-FROM-WHERE, que consiste en elegir los objetos con las cláusulas FROM y WHERE, para después modificarlos con la cláusula UPDATE.

```
biblio:&o1 { libro:&o2 { autor:&o5 "Smith",
                         titulo:&o6 "BD"},
             libro:&o3 { autor:&o7 "Smith",
                         autor:&o8 "Jones",
                         titulo:&o9 "SO"},
             articulo:&o4 { autor:&o10 "Smith",
                            titulo:&o11 "XML"},
                            revista:&o12 { nombre:&o13 "PCW",
                                           año:&o14 4,
                                           numero:&o15 33 }
                          }
           }
```

Figura 5.2: Dato semiestructurado.

Ejemplo 5.4.4 Considérese el dato semiestructurado de la figura 5.2 y la siguiente consulta:

```
SELECT X
FROM biblio.libro X
WHERE X.autor = "Jones"
```

El resultado es el siguiente:

```
{ libro:&o3 { autor:&o7 "Smith",
              autor:&o8 "Jones",
              titulo:&o9 "SO"},
  articulo:&o4 { autor:&o10 "Smith",
                 titulo:&o11 "XML"},
                 revista:&o12 { nombre:&o13 "PCW",
                                año:&o14 4,
                                numero:&o15 33 }
               }
}
```

Al igual que en Ssquirel, las expresiones de camino en Lorel presentan una gran flexibilidad. En Lore las expresiones de camino se pueden usar en las tres cláusulas (SELECT, FROM y WHERE), como se hace en OQL, pero su uso resulta confuso debido a que Lorel está basado en OQL, que fue diseñado para tratar objetos con una estructura fija.

Lorel posee algunas formas para construir un nuevo dato semiestructurado como resultado de una consulta, pero estas construcciones son limitadas. La construcción de nuevos datos en Lorel se ve superada por la de XML-QL y Ssquirel. Lorel tampoco permite el podado de la información. En conclusión, Lorel presenta algunas carencias con respecto a Ssquirel en cuanto a poder expresivo.

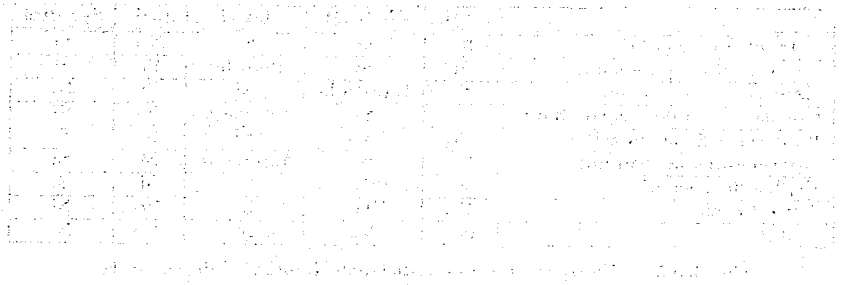
5.4.4. XSL

Como se vió en la sección 4.4, XSL es un lenguaje de definición de hojas de estilo para transformar documentos XML a otro tipo de documentos. De alguna manera XSL se puede considerar como un lenguaje de consulta, ya que se pueden elegir porciones de un documento XML de entrada y mostrar el resultado en otro documento XML. XSL es más limitado que XML-QL, las construcciones de nuevos datos como resultado de una consulta tienen menos poder en XSL que en XML-QL, debido al uso de variables en XML-QL. Como Ssquirel supera el poder expresivo que XML-QL, también supera el de XSL.

XSL considera un documento XML como un árbol y lo transforma en otro árbol de manera recursiva iniciando por la raíz, mientras que XML-QL, Ssquirel y otros lenguajes lo que hacen es dirigirse a los datos, aplicarles una condición y construir nuevos datos. Esto quiere decir que el modelo computacional que sigue XSL es diferente de otros lenguajes.

	XQL	XML-QL	Lorel	XSL	Ssquirel
Estructura de gráfica	No	No	Si	No	Si
Sintaxis clara y sencilla	Si	Si	Regular	Si	Si
Expresiones de camino	Si	Limitada	Si	Si	Si
Construcción de nuevos datos	No	Si	Limitada	Si	Si
Funciones de agregación	No	No	Si	Si	Si
Operaciones de conjunto	No	Si	Limitada	Si	Si
Reglas de coerción	Si	Limitada	Si	Si	Si
Actualización	No	No	Si	No	Si
Vistas	No	No	Si	No	Si

Cuadro 5.1: Comparativo entre distintos lenguajes de consulta.



Conclusiones

A lo largo de las páginas anteriores se han expuesto temas referentes a los datos semiestructurados, los cuales ofrecen una alternativa para solucionar los problemas que requieren de un tratamiento de información flexible y en constante cambio que con otros modelos de bases de datos resultaría bastante complicado.

Hoy en día la Web es la principal fuente de información de muchas áreas del conocimiento y en un futuro cercano lo será de otras más. Los datos semiestructurados ofrecen un apoyo para el tratamiento e intercambio de información en la Web. De aquí que los avances que se logren en este campo se verán reflejados en una mejora de los procesos que involucran información de la Web.

En este trabajo se presentó y se formalizó el concepto de dato semiestructurado y se propuso un modelo para la construcción de bases de datos semiestructurados; todo ello con el fin de sentar las bases para un estudio ordenado y sistematizado de las propiedades de éstos. El formalismo presentado está basado en teoría de conjuntos, de esta manera resulta más apegado al concepto intuitivo de dato semiestructurado expuesto, que utiliza la idea de conjunto.

El objetivo central fue el diseño de Ssquirel, que es un lenguaje de consulta para bases de datos semiestructurados. Su propósito es proporcionar un medio para el tratamiento de la información contenida en este tipo de bases de datos, así como cubrir las deficiencias de los lenguajes existentes de tipo similar. Las virtudes principales de Ssquirel son su poder expresivo y la claridad de su semántica, obtenidas gracias a un diseño basado exclusivamente en el modelo de datos semiestructurados y a todas las construcciones que posee para manipularlos. Así se logró que en muchos aspectos Ssquirel resulte ser superior a los otros lenguajes de consulta existentes.

Como parte complementaria del trabajo se presentaron las características de XML y su similitud con el modelo de datos semiestructurados. Esto resulta particularmente importante debido a que se espera que dentro de algún tiempo XML sea el formato predominante en la Web y en un sinfín de aplicaciones.

Así este trabajo ha llegado a su fin cumpliendo con los objetivos propuestos.

No obstante, el trabajo que queda por delante es enorme y las posibles aplicaciones son interminables. Dentro de los trabajos futuros está el implementar un sistema manejador de bases de datos semiestructurados que use Ssquirel como lenguaje de consulta. El diseño de este sistema trae consigo otros problemas como es el caso de plantear una estructura de almacenamiento, diseñar un procesador y un optimizador de consultas, el manejo de seguridad y transacciones, etc.

Respecto a la parte teórica queda por desarrollar un modelo computacional para expresar las consultas en una base de datos semiestructurados y elaborar un álgebra o un cálculo para datos semiestructurados. Está pendiente establecer formalmente el poder expresivo de Ssquirel en términos del orden de la lógica de predicados que puede expresar, así como su complejidad computacional. También queda estudiar la relación existente con los lenguajes de consulta propuestos para información de tipo similar.

Bibliografía

[1] S. Abitebul, P. Buneman & D. Suciu. *Data on the Web: from Relations to Semistructured data and XML.* Morgan Kaufmann Publishers, 2000.

[2] S. Abiteboul. *Quering semi-structured data.* International Conference on Database Theory, vol. 6, pp.1-18, 1997.

[3] S. Abitebul, D. Quass, J. McHugh, J. Widom & J. Wiener. *The Lorel Query Languaje for Semistructured Data.* Department of Computer Science, Stanford University. Journal on Digital Libraries, 1(1), 1996.

[4] C. Beeri & Y. Tzaban. *SAL: An Algebra for Semistructured Data and XML.* The Hebrew University. WebDB (Informal Proceedings), pp. 37-42, 1999.

[5] A. Bergholz. *Lore Tutorial.* Incluido en la distribución de Lore.

[6] A. Bonifati & S. Ceri. *Comparative Analysis of Five XML Query Languajes.* Dipartimento di Elettronica e Informazione, Politecnico di Milano. SIGMOD Record, vol. 29, num. 1, pp. 68-79, 2000.

[7] N. Bradley. *The XSL Companion.* Addison-Wesley, 2000.

[8] P. Buneman. *Semistructured data.* Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 117-121, 1997.

[9] L. Cardelli, P. Gardner & G. Ghelli. *A Spatial Logic for Querying Graphs.* ICALP'02, 2002.

[10] L. Cardelli. *Describing Semistructured Data.* SIGMOD Record, vol. 20, num. 4, pp. 80-85, 2001.

[11] C.J. Date. *Introducción a los Sistemas de Bases de Datos.* Prentice Hall. 7a edición, 2001.

[12] H. Garcia Molina, J.D. Ullman & J. Widom. *Database System Implementation.* Prentice Hall, 2000.

[13] C. Goldfarb & P. Prescod. *Manual de XML.* Prentice Hall, 1999.

[14] D. Q. Goldin. *Mathematical Models of Interactive Computing.* U.Mass/Boston, http://www.cs.umb.edu/~dqg/talks/umbc.ps.

[15] L. Libkin & L. Wong. *On Representation and Querying Incomplete Information in Databases with Multisets.* Information Proccessing Letters, vol. 56, pp. 209-214, 1995.

[16] J. McHugh & J. Widom. *Query Optimization for XML.* Twenty-Fifth International Conference on Very Large Data Bases, 1999.

[17] E. Mendelson. *Introduction to Mathematical Logic.* Chapman & Hall, 1996.

[18] A. O. Mendelzon & T. Milo. *Formal Models of Web Queries.* Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 134-143, May 1997.

[19] M. Morrison, et al. *XML al Descubierto.* Prentice Hall. 1a. edición, 2000.

[20] A. Silberschatz, H. Korth & S. Sudarshan. *Fundamentos de Bases de Datos.* McGraw-Hill. 3a. edición, 1998.

[21] D. Suciu. *An Overview of Semistructured Data.* SIGART News, vol. 29, num. 4, pp. 28-38, 1998.

[22] Extensible Markup Language (XML). The World Wide Web Consortium (W3C). *http://www.w3.org/XML.*

[23] The Extensible Stylesheet Language (XSL). The World Wide Web Consortium (W3C). *http://www.w3.org/Style/XSL.*

[24] Extensible Markup Language (XML). Version 1.0. W3C Recomendation 10 February 1998. *http://www.w3.org/TR/1998/REC-xml-19980210.*

[25] Extensible Stylesheet Language (XSL). Version 1.0. W3C Recomendation 15 October 2001. *http://www.w3.org/TR/xsl.*

[26] XSL Transformations (XSLT) Version 1.0. W3C Recomendation 16 November 1999. *http://www.w3.org/TR/xslt.*

[27] XML-QL: A Querry Language for XML. Submission to the World Wide Web Consortium 19 August 1998. *http://www.w3.org/TR/1998/NOTE-xml-ql-19980819.*

[28] Eric Weisstein's World of Mathematics. Zermelo-Fraenkel Axioms. *http://mathworld.wolfram.com/AxiomFoundation.html.*

Índice alfabético