VNIVERSIDAD NACIONAL AVFNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

UN PROCESADOR DE CONSULTAS PARA DATOS SEMIESTRUCTURADOS

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS (COMPUTACIÓN)

P R E S E N T A:

ANDRÉS LÓPEZ CAPISTRÁN

DIRECTORA DE LA TESIS: DRA. AMPARO LÓPEZ GAONA

MÉXICO, D.F. **2005.**

Con cariño para mis padres,
por su apoyo en todo momento,
por darme ánimos siempre que lo he necesitado,
por haberme guiado en el camino correcto,
porque sin sus consejos nunca hubiera llegado a esta meta.

Gracias a la Dra. Amparo López Gaona,
por brindarme siempre su ayuda y motivarme para seguir adelante,
por sus continuas revisiones que hicieron de esta tesis un mejor trabajo.

Índice

Introducción

Las bases de datos han ido evolucionando a través del tiempo para satisfacer nuevos requerimientos de búsqueda y almacenamiento de información. Las bases de datos semiestructurados se enfocan en resolver los problemas a los que se enfrentan los desarrolladores de sistemas de *software*, al no tener pleno conocimiento de la estructura que tendrán los datos con los que trabajarán. Estas dificultades se han acentuado con las nuevas tecnologías de comunicación y transmisión de datos por Internet.

Una base de datos es controlada mediante un sistema de *software*, el cual se conoce como sistema administrador de bases de datos. En él, se agrupa un gran número de técnicas enfocadas principalmente a recuperar y almacenar, con mayor eficiencia, los datos de interés. Para especificarle al sistema administrador de bases de datos cuáles son estos datos, se hace uso de un lenguaje especial de consulta, el cual describe de dónde obtenerlos y cómo presentarlos. Además, ofrece mecanismos para crearlos o eliminarlos de la base de datos, pero no define cómo debe realizar estas operaciones. Este trabajo le corresponde al sistema administrador de base de datos.

A nivel de complejidad, un sistema administrador de bases de datos se divide en varios componentes, los cuales se muestran en la figura I. Estos componentes se encargan de una función específica dentro del sistema completo. Existen para: controlar el acceso; controlar la concurrencia; manejar el medio de almacenamiento; interpretar las consultas; etc. Este último componente constituye el procesador de consultas que es el centro de interés de esta tesis, concretamente enfocado para bases de datos semiestructurados

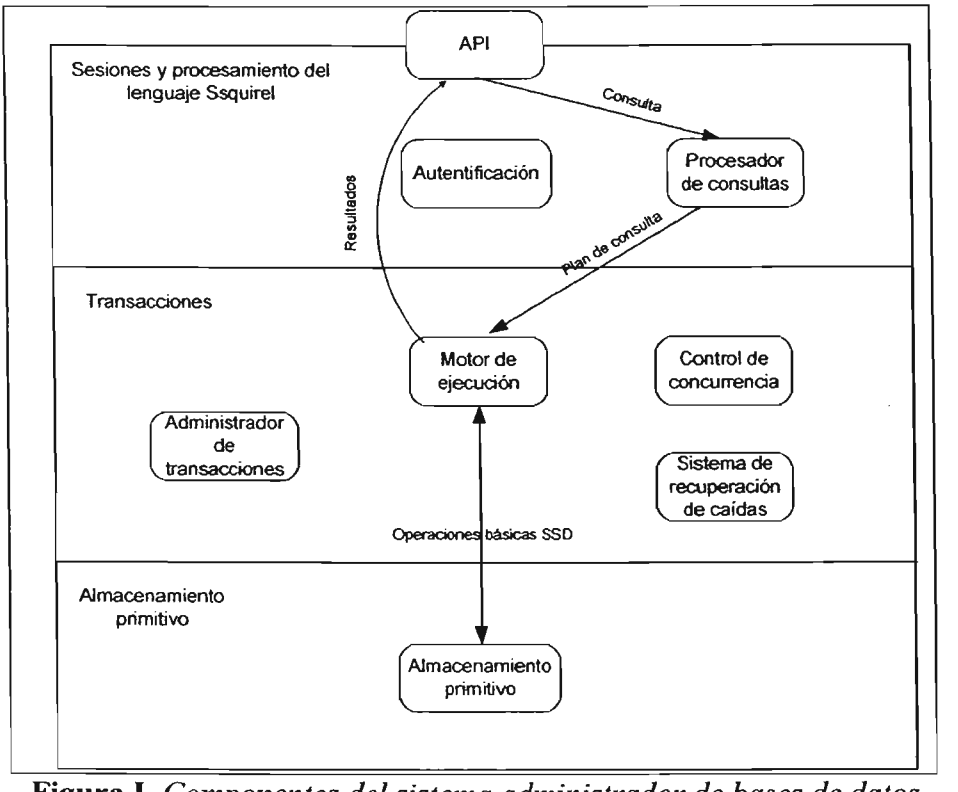


Figura I. *Componentes del sistema administrador de bases de datos semiestructurados*

El procesador de consultas tiene la tarea de convertir una *proposición escrita en un lenguaje de consulta* (llamada *consulta* para simplificar) a un conjunto de instrucciones propias del componente encargado del medio de almacenamiento. Como lenguaje de consulta se utilizó Ssquirel [Garc02], el cual es un lenguaje reciente para bases de datos semiestructurados.

El procesador de consultas está dividido en subcomponentes. Éstos tienen semejanza con las etapas encontradas en los compiladores de lenguajes de programación; es decir, su análisis léxico, sintáctico y semántico, la transformación a código intermedio, optimizaciones y la transformación a código objeto. A diferencia de estos compiladores, el procesador de consultas tiene como salida un código objeto que no es dirigido a una máquina objeto, sino hacia otro componente interno del sistema administrador, el motor de ejecución. Lo anterior, en consecuencia, cambia el enfoque que se da a las optimizaciones, pues no se tiene un control directo de la memoria o, en el caso de las bases de datos, de los accesos a disco. La relación entre las etapas de un compilador

común para lenguajes de programación y los subcomponentes del procesador de consultas desarrollado en esta tesis se puede observar en la Tabla I.

Compilador	*Procesador de consultas*
Análisis léxico	Análisis léxico
Análisis sintáctico	Análisis sintáctico
Análisis semántico	Preprocesador
Código intermedio	Plan lógico de consulta
Optimizaciones	Optimizaciones
Código objeto	Plan físico de consulta

Tabla I. *Relación entre las etapas de un compilador en las del procesador de consultas*

Una consulta pasa por cada uno de estos subcomponentes del procesador de consultas en su camino a ser ejecutada. En este proceso sufre varias transformaciones: pasa de una simple cadena de caracteres a un árbol de análisis sintáctico, el cual realiza un reconocimiento de las partes que conforman a la consulta que se esté evaluando; de ese árbol, se obtiene un plan lógico de consulta, que le sirve al sistema para determinar si es posible crear un plan diferente que sea equivalente pero que pueda recuperar la información con un menor gasto de recursos del sistema; del plan lógico se transforma a uno físico, el cual contiene funciones primitivas del almacenamiento interno. En este último plan se determina el orden de ejecución, es decir, cuáles funciones se deben ejecutar antes que otras.

En cada etapa por la que pasa una consulta a través del procesador de consultas, se tiene muy en cuenta cuál es la mejor opción para ejecutarla, esto es, se realizan optimizaciones. Lo que se busca es mejorar la recuperación de la información de interés descrita por la consulta, ahorrando memoria y/o espacio de almacenamiento y realizando un menor número de accesos a disco, que es el aspecto en el que se hace más énfasis en esta tesis.

Este trabajo tiene el propósito de describir detalladamente la parte teórica de cada uno de los subcomponentes del procesador de consultas, así como llevarlos a la práctica construyendo el procesador de consultas como parte de un sistema administrador de bases de datos semiestructurados. Para ejecutar los resultados obtenidos por el procesador de consultas (el plan físico), también se desarrolla el aspecto básico del motor de ejecución, que es otro componente del sistema administrador de

bases de datos semiestructurados. Este último componente tiene la responsabilidad de manejar los datos temporales generados durante la consulta y de monitorear que siempre exista memoria disponible.

Algunos de los temas desarrollados en esta tesis se basaron en trabajos similares [McWi97, MAG+97, GoWi97, McHu00]. Sin embargo, se realizan varias aportaciones en el área de investigación de bases de datos semiestructurados al proponer nuevos diseños de planes lógicos y físicos empleados dentro del procesador de consultas, así como algunas mejoras a las ideas de optimización utilizadas al manejar este tipo de datos.

El contenido de este documento está clasificado en 6 capítulos y 2 apéndices. El capítulo 1 introduce al ambiente de datos semiestructurados, su historia, la motivación, el modelo de datos, etc. y se presenta brevemente el lenguaje de consulta Ssquirel utilizado en esta tesis.

En el capítulo 2 se expone un panorama general de la arquitectura del sistema administrador de bases de datos semiestructurados, en donde actuará el procesador de consultas, objeto de esta tesis, para después profundizar en este último dando la descripción y misión de cada uno de sus subcomponentes.

En el capítulo 3 se trata en detalle el plan lógico de consulta. Aquí se propone un lenguaje lógico que agrupa varias ideas desarrolladas en el área de datos semiestructurados y se define formalmente cada uno de los operadores lógicos utilizados dentro de este lenguaje. El plan lógico sirve como base para las optimizaciones que son propuestas en el capítulo 4.

En el capítulo 5 se maneja la última transformación que sufre una consulta, la cual es necesaria para obtener el plan físico de consulta. Por último, el capítulo 6 describe el motor de ejecución, los mecanismos que se utilizan para crear un ambiente adecuado al ejecutar una consulta y su función dentro del sistema administrador de bases de datos semiestructurados.

Los apéndices se refieren al análisis léxico y sintáctico del procesador de consultas y a pruebas de rendimientos elaboradas para corroborar el funcionamiento del mismo.

Capítulo 1

Los datos semiestructurados

El concepto de dato semiestructurado surgió al observar que mucha de la información disponible electrónicamente, como páginas Web, tiene una estructura que varía tanto a través del tiempo como entre las distintas fuentes. Almacenar este tipo de información de forma descriptiva y útil en una base de datos usual, como son las relacionales, puede llegar a ser una tarea difícil de completar, pues esta clase de bases de datos no permiten reflejar los cambios en la estructura de la información, es decir, en el esquema. En un sistema administrador de bases de datos relacional no sólo habría que modificar continuamente el esquema de la base de datos para poder manejar este tipo de información, lo que implica el reacomodo de los registros internos para dar o quitar espacio de campos agregados o eliminados, sino que, además, habría el problema de tener posiblemente muchos campos en una relación que no serían utilizados por todas las tuplas, pero aún así, el sistema administrador de bases de datos relacional tendría que dejar reservado el espacio para estos campos, pudiendo llegar a desperdiciar recursos de almacenamiento. Los datos semiestructurados ofrecen un nuevo modelo de almacenamiento capaz de manejar este tipo de situaciones.

La integración de información, es decir, almacenar información proveniente de diferentes fuentes en un solo sistema que pueda aprovecharla, es otro de los puntos que motivó el origen de los datos semiestructurados. Actualmente, la creciente popularidad del XML para compartir información en un formato que se pueda utilizar y manipular por diferentes aplicaciones, ha impulsado también el desarrollo de los datos semiestructurados. Hay similitudes entre XML y datos semiestructurados, dos de ellas son: ambos representan la información en forma de árbol; y han sido objeto del diseño de varios lenguajes de consulta [Suci98, AQM+97, W3C99, W3C04]. Sin embargo, XML se desarrolló específicamente para el intercambio de información y los datos semiestructurados como una extensión a las técnicas de administración de

bases de datos para datos con estructura irregular, desconocida y con cambios frecuentes.

Siendo una colección de datos, un modelo para representar datos y un sistema manejador, las partes que definen a una base de datos, la mayor diferencia entre XML y los datos semiestructurados está en el sistema manejador. Para XML, los datos se almacenan en forma de archivos de texto plano, sin la utilización de índices ni alguna otra técnica desarrollada en las bases de datos, dejando todo el trabajo de encontrar información al programa que procese dichos archivos. Esto último podría llegar a ser un problema en un sistema que maneje gran cantidad de información, pues el tiempo para recuperarla y/o la memoria necesaria para recorrer un documento XML extenso, serían muy grandes. Por ejemplo, existen dos tecnologías muy utilizadas en el recorrido de documentos XML para la extracción de información, SAX y DOM, pero ambas caen en alguno de los problemas descritos anteriormente. SAX recorre el documento XML en una sola dirección, de manera que si, por alguna razón, se necesite retroceder, hay que regresar al inicio del documento, algo que causaría perdida de tiempo para una consulta. Por el otro lado, DOM soluciona el problema de SAX al cargar todo el documento en memoria, pero tratándose de un documento XML extenso, podría no haber suficiente memoria para realizar dicha operación.

No sólo estos detalles surgen cuando se utiliza XML como un almacén directo de datos, sino otros como las transacciones y el manejo de concurrencia, comunes en los sistemas administradores de bases de datos, se deben llevar a cabo separadamente, si es que son necesarios.

1.1 Definición de los datos semiestructurados

Se pueden definir a los datos semiestructurados [Suci98, Abit97, Bune97] como:

- Datos que no se encuentran en bruto ni con una estructura rígida.
- Datos que contienen internamente el esquema al que se restringen, lo que se conoce como datos autodescriptivos.
- Datos con estructura irregular o flexible.

La figura 1.1 ayuda a identificar la posición de los datos semiestructurados entre los datos que normalmente se encuentran en fuentes electrónicas.

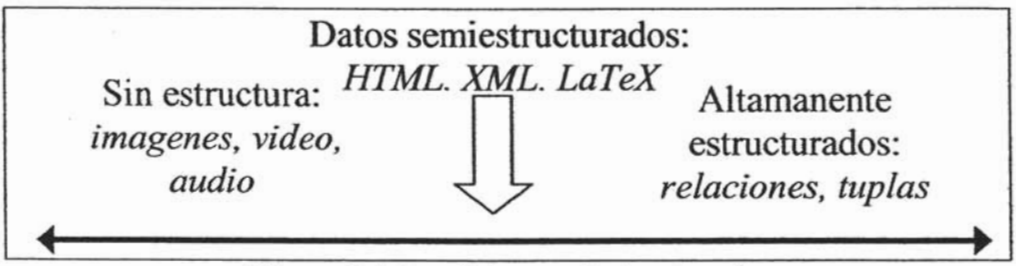


Figura 1.1 *Los datos semiestructurados*

Los datos semiestructurados se describen a ellos mismos, esto es, contienen información referente a su significado. Por ejemplo, los datos de una guía de restaurantes podrían modelarse de la forma siguiente:

```
&12 guía
      &19 restaurante
            &17 categoría gourmet
            &13 nombre Chef Chu
            &14 dirección
                  &44 ciudad DF
                  &16 cp 92310
      &35 restaurante
            &66 categoría vietnamese
            &23 nombre Saigon
            &14 dirección
                  &29 calle Zapata
                  &30 número  30
      &77 restaurante
            &55 precio económico
            &80 nombre El Itacate
```

Este ejemplo de datos semiestructurados nos da información acerca de restaurantes, pero además, contiene información acerca de los datos, un identificador y relaciones padre-hijo. Por ejemplo, el restaurante de nombre "Chef Chu" pertenece a la categoría de restaurantes de gourmet y se ubica en la ciudad de "Palo Alto". Los identificadores (como &35) se describen posteriormente en este capítulo. Por lo anterior, el esquema de una base de datos semiestructurados puede llegar a ser tan grande como los datos en sí, lo que dificulta la distinción entre el esquema y los datos.

Algunos ejemplos de datos que pueden considerarse como semiestructurados son:

- Datos Web: HTML
- Formatos de intercambio de datos, en general:
  - SGML, XML, etc.
- Código: C, LaTeX, etc.

1.2 Particularidades de los datos semiestructurados

A continuación se describen algunas particularidades que hay que tener en mente cuando se habla de datos semiestructurados.

La estructura es irregular.- De forma contraria a como ocurre en las bases de datos relacionales, donde toda la información está estrictamente apegada a un esquema rígido, la estructura de los datos semiestructurados no restringe a la información a seguir una determinada estructura. Retomando el ejemplo anterior de los restaurantes, hay restaurantes que almacenan mayor información que otros, o existen restaurantes con información faltante o en blanco al no contener algún campo definido por otro.

No sólo se da la variación en la estructura de la información, también existe la variación de tipos. Por ejemplo, en el siguiente dato semiestructurado:

```
&1 Ventas
      &11 Casa
            &23 Precio 400000000.50
            &43 Dirección Colonia Centro, México DF
      &79 Casa
            &34 Precio 500000
            &32 Dirección
                  &42 Colonia John F. Kennedy
                  &65 País EUA
```

Tomando en consideración la parte de información que almacena la dirección de las casas, podría especificarse como tipo cadena a la dirección de la casa en venta en México, pero para la de EUA la dirección contiene otros dos atributos, colonia y país, por lo que el tipo para ambos casos varía.

Para profundizar en este punto, se puede observar que el precio de la casa en venta en México tiene una representación como dato flotante, cosa que para la segunda es de tipo entero. Para realizar una comparación entre ambas cantidades, el sistema administrador de bases de datos tendría que convertir el tipo para poder ejecutarla. Es más, podría ser que uno de los dos fuera una cadena y el otro un número, donde de igual forma se tendría que realizar una transformación de alguno de los dos precios antes de poder compararlos. Sin embargo, en un sistema administrador de bases de datos semiestructurados esta conversión se hace implícitamente.

La estructura está implícita.- Los datos semiestructurados son autodescriptivos, ya que la misma información contiene los datos en sí junto con su descripción o estructura y, como se dijo en el punto anterior, esta estructura puede variar o no ser uniforme. Esto da flexibilidad para procesar (almacenar, consultar, etc.) cualquier información y poder manejar cambios en la estructura de los datos sin notarlo; sin embargo, el procesamiento de información en una fuente semiestructurada es más complicado.

El que los datos semiestructurados no tengan restricciones en su estructura presenta algunas desventajas [Suci98a]:

- La información es almacena de forma ineficiente, pues es necesario replicar el esquema con cada elemento de la información, aún cuando la estructura de los datos fuera semejante.
- Las consultas son difíciles de evaluar eficientemente; hasta para las consultas más sencillas sería necesario recorrer toda la información en busca de los datos que cumplan con las características deseadas.
- Las consultas son difíciles de formular, pues los usuarios no disponen de información acerca de los datos, de manera que puedan realizar consultas relevantes.

Es por esto que fue necesario el desarrollo de técnicas de almacenamiento y consulta que ayudarán a mejorar la recuperación de la información, un punto muy importante para los sistemas administradores de bases de datos. En los capítulos 3, 4 y 5 se mostrarán algunas de las técnicas referentes a la evaluación de consultas.

La estructura puede ser parcial.- Es posible encontrar que parte de los datos almacenados en una fuente de datos semiestructurados, su descripción o estructura no se especifica totalmente. Alguna de las dos situaciones siguientes puede ocurrir:

- Datos carentes de estructura.- Este tipo de datos también conocidos como datos en bruto, por ejemplo imágenes, videos u otro tipo de información binaria no manejable por el sistema administrador de bases de datos. Posiblemente contengan un tipo de estructura, pero para recuperarla es necesario procesarla a través de *software* especializado.

- Datos con poca estructura.- Un archivo de texto plano es un buen ejemplo de este tipo de datos, donde la forma de búsqueda más común es encontrar una palabra o enunciado dentro de ellos, por lo que la división que se realiza generalmente es por palabras. Podría haber otro tipo de divisiones, como párrafos, capítulos, etc., pero su uso en búsquedas no es muy común.

Hay que tener cuidado de no confundir este punto con el de "estructura irregular". Aquí se mencionan tipos de datos con poca o casi nula estructura; el punto anterior refiere lo irregular que puede ser la estructura en una misma fuente de datos semiestructurados.

Política de tipos no estricta.- A diferencia de las bases de datos relacionales, que rechazan cualquier actualización si no cumple con el esquema definido, las bases de datos semiestructurados aceptan cualquier tipo de actualización. Esto no sólo ocurre durante las actualizaciones, sino que también se puede encontrar con una situación similar al realizar alguna consulta. Es posible que se necesite recuperar información con una descripción o estructura que no existe en la base de datos, sin embargo, la consulta se aceptará y no devolverá ningún resultado. Esto último no pasa en las bases de datos relacionales, donde toda consulta debe apegarse al esquema.

Esquema a-posteriori.- En una base de datos relacionales primero hay que definir un esquema y después ingresar los datos siguiendo ese esquema, pero en una base de datos semiestructurados sucede de forma totalmente inversa, los datos son los que definen al esquema. La posibilidad de extraer un esquema de una fuente de datos semiestructurados es un tema ha tratar posteriormente.

1.3 El modelo de datos semiestructurados

Una de las primeras formas de representación de datos semiestructurados provino del lenguaje funcional LISP y su sintaxis era muy sencilla:

{name: "Alan", tel: 2157786, email: agb@abc.com}

Esta representación consideraba a los datos semiestructurados como parejas de etiquetas y valores, donde los valores nuevamente podían ser otra estructura de datos semiestructurados [AbBS00]:

{name: {first: "Alan", last: "Black"}, tel: 2157786, email: agb@abc.com}
La utilización de árboles es uno de los mecanismos mayormente utilizados para la representación gráfica de datos semiestructurados. Para los dos ejemplos anteriores sus gráficas corresponden a la figura 1.3.1.

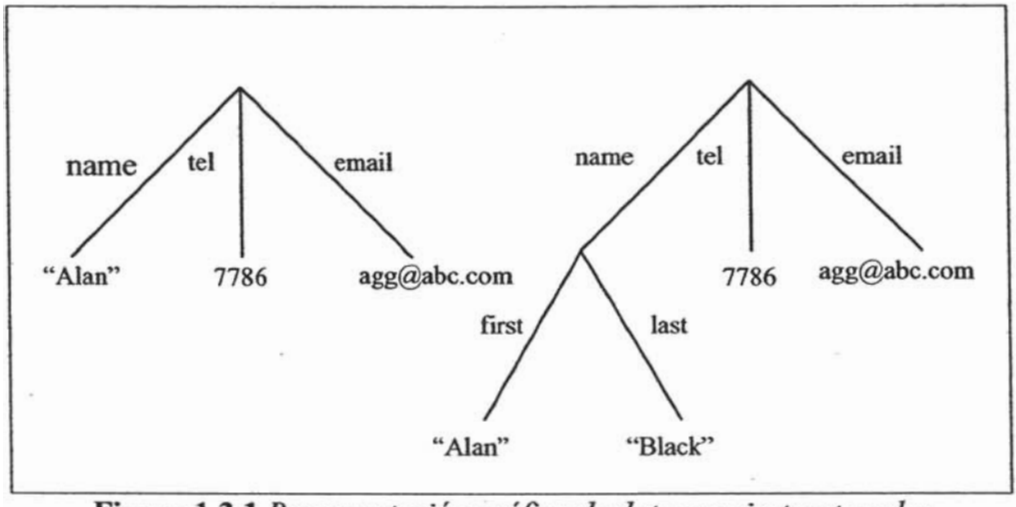


Figura 1.3.1 *Representación gráfica de datos semiestructurados*

Los datos en las hojas de estos árboles reciben el nombre de datos de tipo primitivo. Estos pueden ser: cadenas de caracteres, números, fechas, etc. Es verdad que los datos semiestructurados son autodescriptivos y esta misma característica puede indicar su tipo, pero para el manejo interno de esta información, como al realizar operaciones o comparaciones, es útil tener en consideración la existencia de ciertos tipos primitivos. Es posible que internamente (almacenamiento primitivo) los datos se guarden como cadenas de caracteres, pero se puede dar un cierto patrón a éstos para distinguir entre su tipo primitivo. Por ejemplo, las cadenas de caracteres escribirlas entre comillas, las fechas tendrían formato específico, etc.

Con este modelo se puede definir a los datos semiestructurados como:

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

Posteriormente se desarrollaron otros modelos. Muchos de ellos basados en modelos para bases de datos orientados a objetos, como el ODMG (*Object-Oriented Data Model*) [Catt94] o el OEM (*Object Exchange*

Model) [PaGW95]. Tanto en las bases de datos orientados a objetos como en las de semiestructurados, cada objeto o elemento necesita una identificación propia, cosa que no tenía el modelo anterior, pues las etiquetas podían repetirse a lo largo de la información, por lo que fue necesario agregar identificadores al modelo. Por ejemplo, si se tuviera la siguiente información de clientes representada como datos semiestructurados:

```
{
{nombre: "Juan Carlos", dirección: "Arboleda #4", ciudad:
         "Guadalajara"}
{nombre: "Juan Carlos", dirección: "Lázaro Cárdenas #233",
          ciudad: "Guadalajara"}
}
```

Podría ser la misma persona que tiene dos domicilios o personas totalmente diferentes, cosa que no puede saber el sistema administrador de bases de datos, por lo que debe guardarlos como entidades separadas, pero ¿cómo identificarlos? Algo parecido ocurre en un lenguaje de programación orientado a objetos. Podemos tener varias instancias de la misma clase con el mismo estado, pero el sistema reconoce cada una de ellas por un identificador único que se les asigna. Para los datos semiestructurados se agregó de forma similar un identificador.

Puede haber relaciones entre los datos semiestructurados. En el siguiente ejemplo de datos semiestructurados se muestra cómo utilizar los identificadores para relacionarlos:

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

Las relaciones padre-hijo, madre-hijo, abuelo-nieto se definen con el uso de dichos identificadores como enlaces entre la información. Para no repetir información ya almacenada previamente en la base de datos, por

ejemplo, el padre de José es Pedro, pero como la información de Pedro ya se definió anteriormente, puede usarse escribiendo su identificador.

Aquí entra otro aspecto en consideración. Para que un dato semiestructurado sea consistente debe cumplir lo siguiente:

- Todo identificador debe definirse sólo una vez.
- Todo identificador que es una referencia en el dato semiestructurado debe estar definido.

La figura 1.3.2 muestra la representación gráfica del ejemplo anterior.

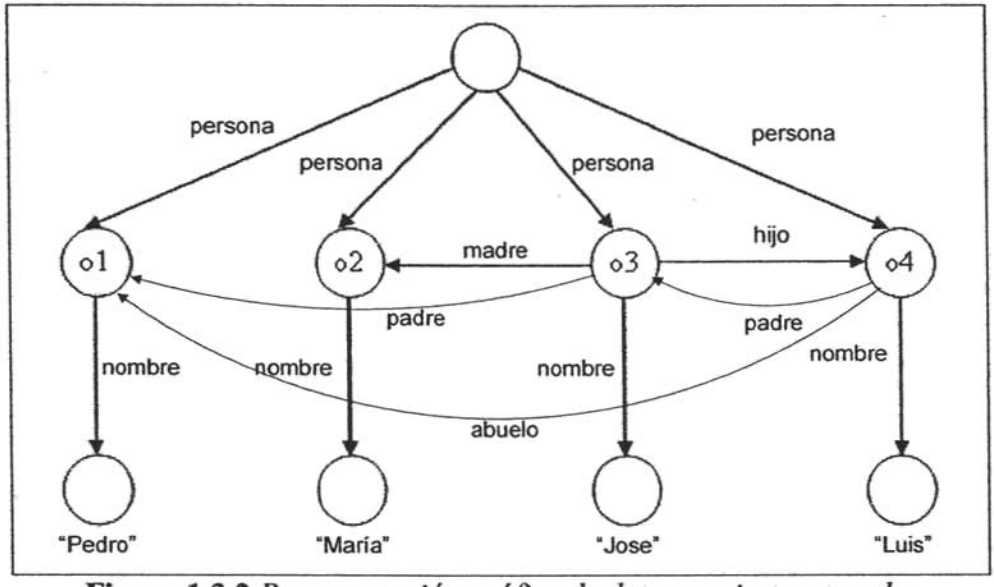


Figura 1.3.2 *Representación gráfica de datos semiestructurados*

Para finalizar con la descripción del modelo de datos semiestructurados, a continuación se proporciona la definición formal [Suci98a]:

Definición 1.3.1 Un dato semiestructurado es $G = (V, E, r, u, F_E)$ donde:

V: Es un conjunto de Vértices

$$V = V_c \cup V_a$$

E: Es un conjunto de aristas

$$E \subseteq V_c \times L_E \times (V - r)$$

r: Es un vértice raíz

$$r \in V$$

u: Es una función

$$u : V_a \rightarrow D$$

F_E: Es un función

$$F_E : E \rightarrow L_E$$

Donde D representa el conjunto de valores atómicos, L_E el dominio de etiquetas de aristas, Vc aquellos vértices internos llamados complejos y Va los nodos atómicos (hojas en la representación gráfica). Básicamente esta definición describe a un dato semiestructurado como un grafo dirigido con etiquetas en las aristas con un vértice especial llamado raíz, al cual no llegan aristas, sólo salen. También existe un conjunto de aristas que sólo pueden tener su punto de origen en un vértice complejo, pues los vértices atómicos representan a datos primitivos (descrito en la definición de la función u), en los cuales ya no se describe la estructura de los datos.

Se podría pensar que un dato semiestructurado puede representarse como un árbol dirigido con etiquetas en las aristas, pues tiene un nodo raíz y los vértices atómicos como hojas; sin embargo, la definición no restringe a un árbol a no tener ciclos (como se presenta en la figura 1.3.2, entre o3 y o4) y un grafo por definición es un árbol sólo si tiene un único camino desde la raíz r a cualquier vértice o nodo v.

1.4 El lenguaje de consulta Ssquirel

Por la misma naturaleza de estructura variable de los datos semiestructurados, es necesario que el lenguaje de consulta sea más flexible que un lenguaje de consulta para bases de datos relacionales, tanto en la revisión de tipos como en el recorrido o acceso a la información. Para el procesador de consultas desarrollado en esta tesis, se utilizó el lenguaje de consulta para datos semiestructurados Ssquirel [Garc02], el cual tiene las siguientes características:

- **Primitivas de consulta estándar.-** Un enunciado común en Ssquirel tiene el formato estándar utilizado en la mayoría de los lenguajes de consulta para bases de datos relacionales, como SQL, de la forma SFW (Select-From-Where). De esta forma se obtiene un lenguaje de consulta que sea sencillo de aprender y comprender para programadores que tengan experiencia en el SQL.
- **Capacidad de composición.-** El resultado de una consulta puede utilizarse dentro de otra, es decir, existe la posibilidad de un anidamiento de expresiones.
- **Capacidad de manipular tanto datos como el esquema.-** Se debe recordar que la distinción entre los datos y el esquema en una base de datos semiestructurados es borrosa, pues al ser autodescriptivos, dentro de los datos se especifica su estructura. Por lo anterior, un lenguaje de consulta para datos semiestructurados debe tener el poder expresivo de procesar (consultar, actualizar, borrar) tanto datos como estructura.

En el modelo de datos semiestructurados utilizado por Ssquirel, además de la definición de un dato semiestructurado como un árbol dirigido con etiquetas en las aristas, se define el concepto de tabla para crear una distinción entre conjuntos de datos dentro de una misma base de datos.

Definición 1.4.1 Cada raíz de un grafo de datos es una Ssd-tabla si existe una flecha etiquetada que apunte a dicha raíz. La etiqueta de la flecha será el nombre de esa Ssd-tabla y a un conjunto de Ssd-tablas se considera como una base de datos semiestructurados.

1.4.1 Expresiones de camino

Una de las características principales que se observa en los lenguajes de consulta para datos semiestructurados (Ssquirel [Garc02], Lorel[AQM+97], UnQL[BDHS96], etc) es la habilidad de alcanzar información a una profundidad arbitraria en el grafo de datos. Para realizarlo, la gran parte de estos lenguajes utilizan las expresiones de camino (o *Path Expressions*) de alguna forma.

Definición 1.4.1.1 Una expresión de camino es una secuencia de etiquetas de arista $l_1.l_2.....l_n$ donde cada l_i con i = 1…n es una etiqueta de arista en el grafo del dato semiestructurado. El resultado de una expresión de camino $l_1.l_2....l_n$ en un grafo de datos es una colección de nodos v_n tales que existen aristas (l_1, r), (r, l_2, v_2), …, (v_{n-1}, l_n, v_n), donde *r* es la raíz.

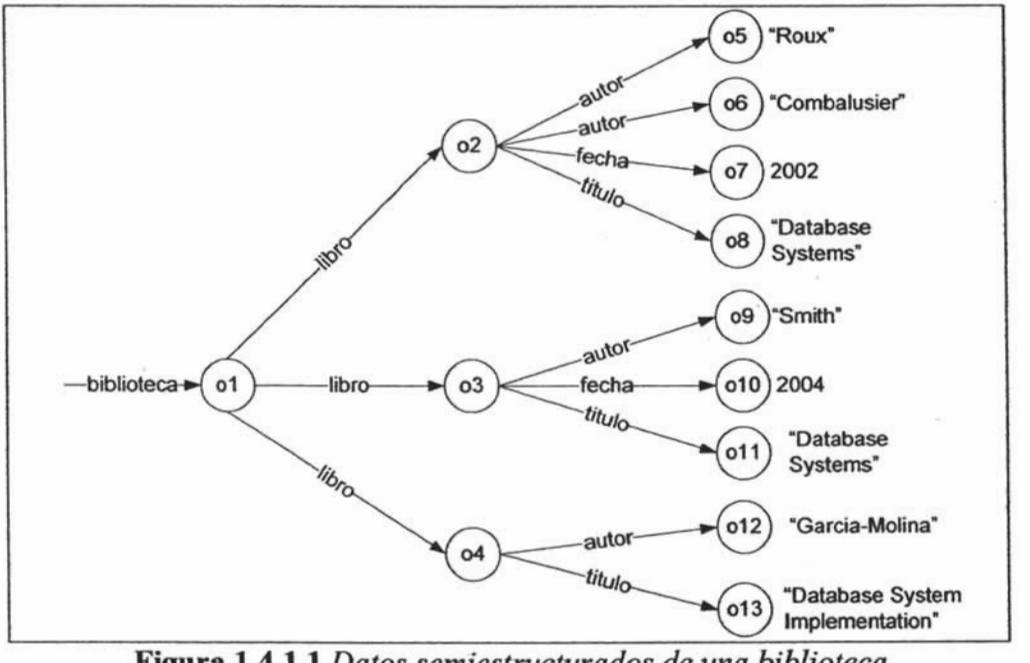


Figura 1.4.1.1 *Datos semiestructurados de una biblioteca*

Ejemplo 1.4.1.1 Se desea evaluar la expresión *biblioteca.libro.autor* sobre el dato semiestructurado de la figura 1.4.1.1.

Los nodos que regresará la expresión *biblioteca.libro.autor* son aquéllos con identificadores o5, o6, o9 y o12. El recorrido para encontrar un nodo del resultado es el siguiente: (biblioteca, o1), (o1, libro, o2), (o2,

autor, o5), como se cumple con todo el camino se regresa el último nodo de la expresión, que en este caso es el nodo con identificador o5. El total de recorridos realizados al evaluar la expresión se muestra en la figura 1.4.1.2.

Cabe señalar que habrá ocasiones en que el resultado contendrá vértices complejos y no sólo atómicos como fue en el caso anterior. Por ejemplo, si sólo se evaluara la expresión de camino sin la última etiqueta, es decir, *biblioteca.libro,* esta expresión regresaría los nodos o2, o3 y o4. Otro detalle a tener en cuenta al construir o evaluar expresiones de camino, es que los datos primitivos no pueden tomarse en cuenta, pues no pertenecen al conjunto de etiquetas. Por ello, construir una expresión de camino *biblioteca.libro.autor.Smith* para la figura 1.4.1.2 no regresaría ningún dato.

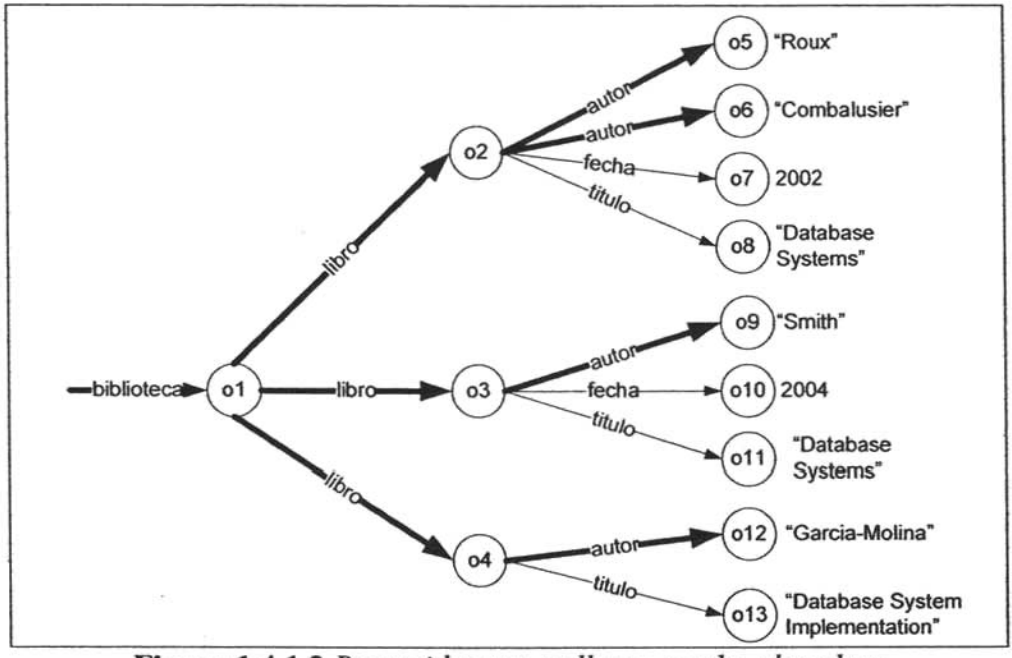


Figura 1.4.1.2 *Recorridos que se llevan a cabo al evaluar la expresión biblioteca.libro.autor*

Existe la posibilidad de que una expresión de camino devuelva nodos repetidos al existir dos caminos iguales hacia un mismo nodo. Esto ocasiona que el utilizar el concepto de conjunto como aquello que devuelve una expresión de camino sea incorrecto, pues un conjunto, por

definición, no contiene elementos repetidos. En el álgebra relacional se extiende el concepto de conjuntos a multiconjuntos o bolsas [UlWi99, DaGK82], para manejar relaciones con tuplas repetidas. Por lo anterior, la definición de una colección es el resultado de la evaluación de una expresión de camino sobre una base de datos. Utilizar el concepto de colección fue con base en lo descrito en [FrHP02], donde menciona que una colección es una generalización de los conceptos de listas, conjuntos y bolsas (multiconjuntos) donde, además de permitir valores duplicados, contiene la propiedad de mantener el orden de sus elementos.

Existen ocasiones en que al escribir una consulta no se tiene conocimiento exacto de la estructura de los datos, o simplemente se desean recuperar partes de información sin importar a qué profundidad del grafo estén. Para dar esta flexibilidad fue necesario agregar nuevos mecanismos a las expresiones de camino, dando como resultado las expresiones de camino extendidas.

Las expresiones de camino extendidas usan expresiones regulares en dos niveles, tomando el conjunto de etiquetas como alfabeto y tomando el conjunto de caracteres que componen las etiquetas como alfabeto. Además, se cuenta con un comodín que representa cualquier etiqueta o cualquier caracter dependiendo del nivel en que se utilice dicho comodín.

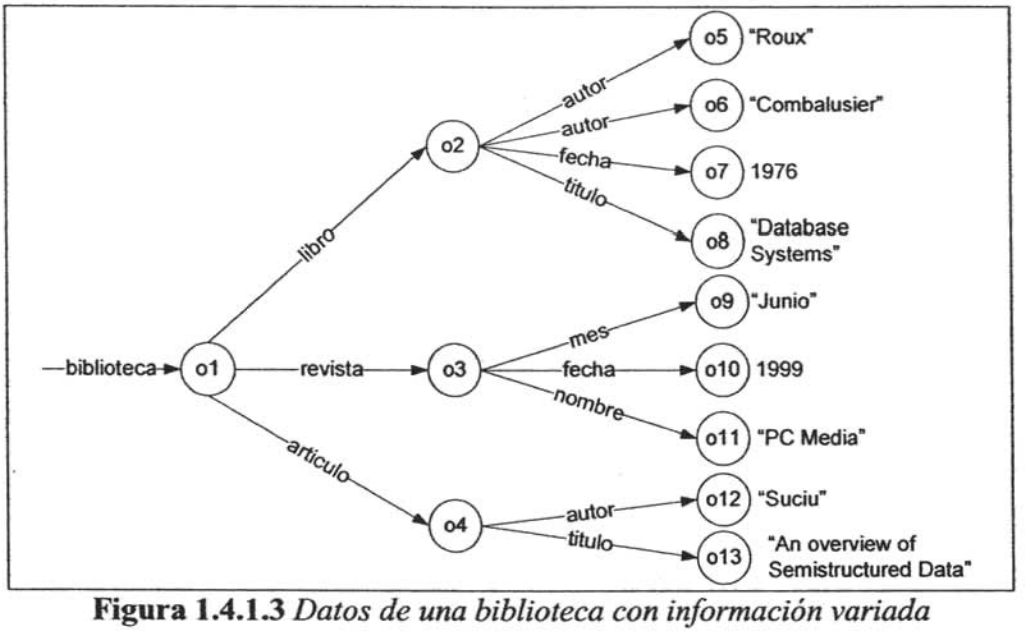


Figura 1.4.1.3 *Datos de una biblioteca con información variada*

Ejemplo 1.4.1.2 Considerando la Ssd-tabla de la figura 1.4.1.3, la expresión de camino extendida a nivel de etiquetas *biblioteca.#.(autor|nombre)* regresa como resultado los nodos con identificadores o5, o6, o11 y o12.

Ejemplo 1.4.1.3 Considerando la misma Ssd-tabla de la figura 1.4.1.3 al evaluar la expresión de camino extendida *biblioteca.'#*o'*, regresa como resultado los nodos con identificadores o2 y o4.

Para más ejemplos de expresiones de camino simples y extendidas se puede consultar la sección 3.1 de la referencia [Garc02].

Como se ha visto, las expresiones de camino sirven para recorrer el grafo de datos al recuperar información específica. En el lenguaje de consulta Ssquirel las expresiones de camino representan una parte esencial para el manejo de datos semiestructurados y tal importancia podría considerarse como la diferencia más significativa entre Ssquirel y SQL. Al momento de diseñar este lenguaje de consulta para datos semiestructurados se procuró hacerlo lo más apegado a SQL y de esta forma tener una sintaxis muy usada y conocida actualmente.

1.4.2 Descripción del lenguaje de consulta Ssquirel

Como en cualquier lenguaje de consultas, Ssquirel define una serie de operadores y funciones. También especifica la sintaxis de los enunciados de consulta, borrado, actualización y definición de datos. A continuación se describen, a grosso modo, distintos tipos de enunciados con algunos ejemplos ilustrativos del lenguaje. Se recomienda al lector interesado dirigirse a [Garc02] donde encontrará una descripción completa de este lenguaje de consulta, aunque este trabajo presenta la sintaxis completa en el Apéndice A.

Un enunciado Select-From-Where similar al utilizado por SQL se emplea en Ssquirel. La evaluación de este enunciado consta de tres etapas. El orden en que se ejecutan fue definido en la descripción del lenguaje Ssquirel [Garc02]. Primero se evalúa la cláusula FROM, después la cláusula WHERE y por último la cláusula SELECT. La sintaxis es la siguiente:

```
SELECT 1: construction
FROM e1 AS X1, …, en AS Xn
WHERE condition
```

Donde cada e_i para i = 1..n representa una expresión de camino y su resultado se asociará a una Ssd-tabla temporal que tendrá como nombre X_i. Cada expresión de camino puede iniciar con el nombre de una Ssd-tabla o vista almacenada en la base de datos, o de alguna tabla temporal creada anteriormente. La condición será una expresión que devuelva falso o verdadero y se podrán utilizar las tablas, vistas o tablas temporales disponibles hasta el momento. Entre algunos de los operadores utilizables para la condición se encuentran:

- LIKE el cual compara un dato primitivo con una cadena, sintaxis
  `<Ssd> LIKE string`
- CONTAINS regresa verdadero si el dato semiestructurado Ssd1 contiene a Ssd2, sintaxis
  `<Ssd1> CONTAIN <Ssd2>`
- BELONGS regresa verdadero si Ssd1 está en Ssd2, sintaxis
  `<Ssd1> BELONG <Ssd2>`
- IS regresa verdadero si Ssd1 y Ssd2 tienen el mismo identificador, sintaxis
  `<Ssd1> IS <Ssd2>`
- OWNS regresa verdadero si Ssd contiene un dato semiestructurado etiquetado con *l*, sintaxis
  `<Ssd> OWN l`
- PRIMITIVE regresa verdadero si Ssd es de tipo primitivo, sintaxis
  `PRIMITIVE <Ssd>`

La cláusula SELECT se encarga de construir un nuevo dato semiestructurado, que será el devuelto por la consulta a manera de resultado. Para hacerlo, se hace uso de las Ssd-tablas y vistas de la base de datos o con tablas temporales $X_1,..,X_n$. Como medio para construir un nuevo dato semiestructurado, Ssquirel ofrece la siguiente serie de instrucciones:

- CLON copia un dato semiestructurado utilizando identificadores que aún no han sido utilizados, sintaxis:
  `CLON <Ssd>`

- AVG crea un nuevo dato primitivo que corresponde al promedio de las cantidades que guardan los hijos de un dato semiestructurado Ssd de entrada, sintaxis:

  `AVG <Ssd>`

- SUM crea un nuevo dato primitivo que corresponde a la suma de las cantidades que guardan los hijos de un dato semiestructurado Ssd de entrada, sintaxis:

  `SUM <Ssd>`

- MAX crea un nuevo dato primitivo que corresponde al valor máximo de las cantidades que guardan los hijos de un dato semiestructurado Ssd de entrada, sintaxis:

  `MAX <Ssd>`

- MIN crea un nuevo dato primitivo que corresponde al valor mínimo de las cantidades que guardan los hijos de un dato semiestructurado Ssd de entrada, sintaxis:

  `MIN <Ssd>`

- COUNT crea un nuevo dato primitivo que corresponde al número de hijos de un dato semiestructurado Ssd de entrada, sintaxis:

  `COUNT <Ssd>`

- PICK crea un nuevo dato semiestructurado que contiene ciertos hijos (aquellos que tienen etiqueta como la de uno de los parámetros de entrada) del dato semiestructurado de entrada, sintaxis:

  `<Ssd> PICK (etiqueta`$_1$`, etiqueta`$_2$`, .., etiqueta`$_n$`)`

- TRIM crea un nuevo dato semiestructurado que no contiene ciertos hijos (aquellos que tienen etiqueta como la de uno de los parámetros de entrada) del dato semiestructurado de entrada, sintaxis:

  `<Ssd> TRIM (etiqueta`$_1$`, etiqueta`$_2$`, .., etiqueta`$_n$`)`

- Operadores Aritmeticos crean un nuevo dato primitivo que contine el resultado la operación, sintaxis:

  `<Ssd> op <Ssd>`

  Donde op representa a uno de los operadores de +, -, *, /, MOD.

- Agrupación ({}) crea un nuevo dato semiestructurado que contendrá uno o varios datos semiestructurados, cada uno con su etiqueta, como hijos, sintaxis:

  `{etiqueta`$_1$`:<Ssd`$_1$`>, .., etiqueta`$_n$`:<Ssd`$_n$`>}`

- UNION crea un nuevo dato semiestructurado que contiene los hijos de ambos datos semiestructurados de entrada, sintaxis:

  `<Ssd`$_1$`> UNION <Ssd`$_2$`>`

Cabe señalar que las expresiones de camino sólo pueden declararse dentro de la cláusula FROM. Por consiguiente, se debe declarar una tabla temporal por cada expresión de camino que se desee utilizar en las otras cláusulas de la consulta, pues dicha tabla tomará su lugar.

Definición 1.4.2.1 Un *ciclo* en la evaluación de una consulta, es la ejecución ordenada de cada una de las cláusulas contenidas en la consulta para una combinación de valores (identificadores de datos semiestructurados) asignados a cada una de las tablas temporales definidas en la cláusula FROM de la consulta.

Existen tantos *ciclos* como combinaciones posibles entre las expresiones de camino $e_1..e_n$ de la cláusula FROM, por cada *ciclo* se evalúan tantas etapas como cláusulas tenga la consulta.

Ejemplo 1.4.2.1 Considerando la base de datos semiestructurados con una sola Ssd-tabla de la figura 1.4.2.1, se evalúa la siguiente consulta:

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

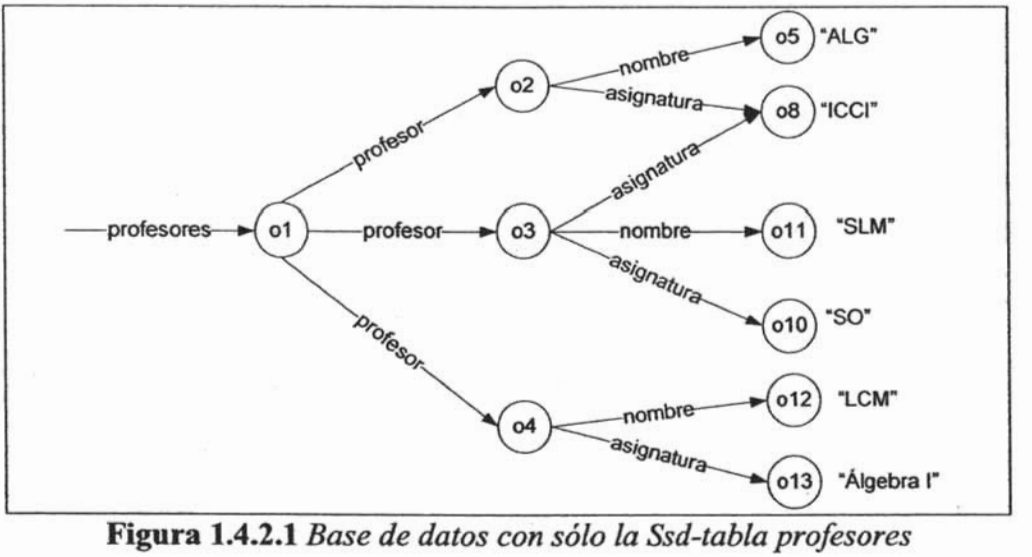


Figura 1.4.2.1 *Base de datos con sólo la Ssd-tabla profesores*

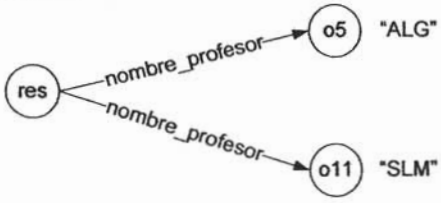
Este ejemplo inicial es sencillo y la evaluación ocurre de la siguiente forma:

- Primer *ciclo*
  - La expresión de camino regresa el nodo con identificado o5 y lo asocia con una tabla temporal de nombre X:

Este resultado lo procesa la cláusula SELECT donde se construye un nuevo dato semiestructurado con la etiqueta nombre_profesor y el contenido en X:

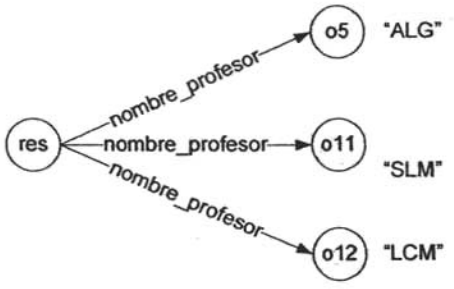
- Segundo *ciclo*
  - La expresión de camino regresa el nodo con identificador o11:

Este resultado se agrega al nuevo dato semiestructurado creado anteriormente:



- Tercer *ciclo*
  - En este último *ciclo* la expresión de camino del FROM regresa el nodo con identificador o12 y lo asigna a la tabla temporal de nombre X:

El resultado final es el siguiente dato semiestructurado, donde el sistema automáticamente asigna el identificador "res".



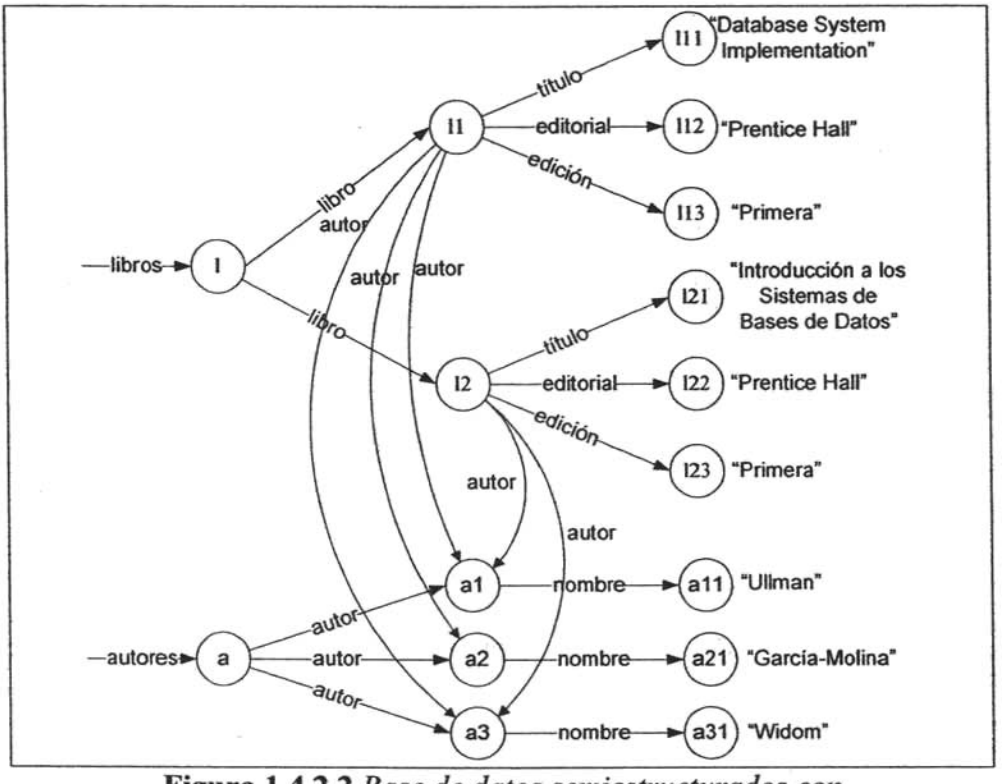


Figura 1.4.2.2 *Base de datos semiestructurados con información de libros y sus autores*

Ejemplo 1.4.2.2 Se evaluará la consulta:

```
SELECT   LibrosEnEquipo: T
FROM     libros.libro AS L,
         L.título as T,
         L.autor.nombre AS N
WHERE    3 = COUNT (L PICK (autor)) and N = "Ullman"
```

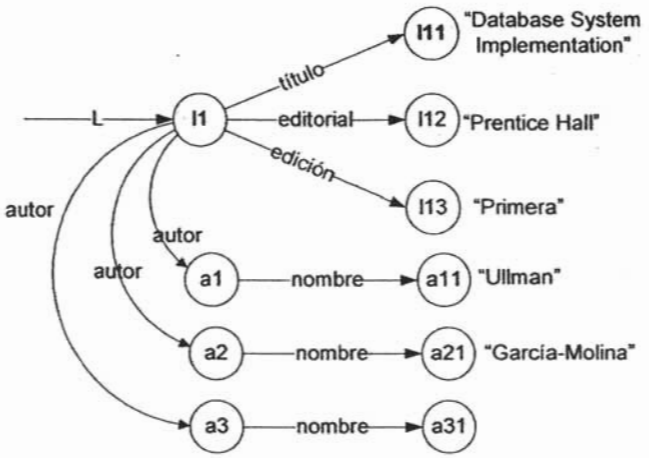
sobre la base de datos semiestructurados de la figura 1.4.2.2.

En esta ocasión se evaluará una consulta más compleja que contiene la cláusula WHERE. Hay que observar que esta consulta tendrá cinco *ciclos*:

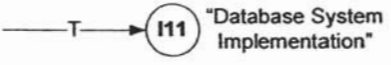
- *libros.libro* devuelve dos identificadores.
  - *L.titulo* devuelve uno por cada libro.
  - *L.autor.nombre* devuelve tres nombres para un libro y dos para el otro.

A continuación se listan los pasos de evaluación.

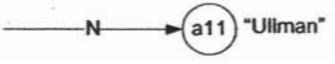
- Primer *ciclo*:
  - Evaluación de la cláusula FROM: La expresión de camino *libros.libro* devuelve el identificador "l1" y se asocia con la tabla temporal llamada L.



Evaluación de la expresión de camino *L.título* la cual devuelve el identificador "l11" y lo asocia con la tabla temporal llamada T.

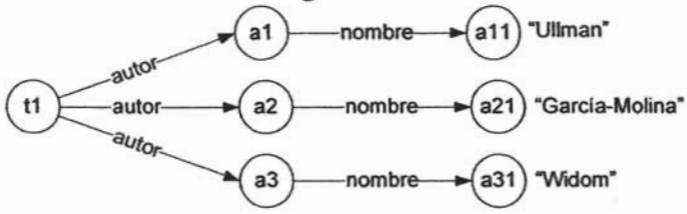


Evaluación de la siguiente expresión de camino *L.autor.nombre* la cual devuelve el identificador "a11" y lo asocia con la tabla temporal llamada N.



- Evaluación de la cláusula WHERE; si devuelve verdadero, se ejecuta la construcción del nuevo dato semiestructurado contenida en la cláusula SELECT.

  - Para la parte de 3 = COUNT (L PICK (autor))
    - L PICK (autor) devuelve el nuevo dato semiestructurado siguiente:



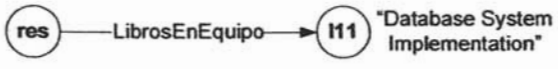
    - COUNT(t1) devuelve el dato primitivo 3.

    - 3 = "dato primitivo 3" es **verdadero**, pues toda constante se toma como un dato primitivo.

  - Para la parte N = "Ullman", ambas partes son datos primitivos del mismo valor, por lo que el resultado es **verdadero**.

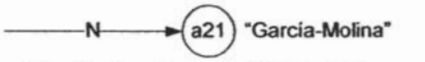
  - Por lo tanto la condición del WHERE es **verdadera**.

- Evaluación de la cláusula SELECT: Se construye el dato semiestructurado de la cláusula SELECT
  - Se agrega el dato semiestructurado "l11" contenido en la tabla temporal T al resultado con etiqueta "LibrosEnEquipo":



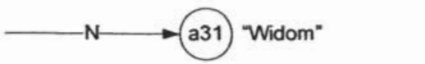
Con esto se concluye este *ciclo*.

- Segundo *ciclo*:
  - Evaluación de la cláusula FROM: La expresión de camino *L.autor.nombre* regresa el siguiente identificador, que es "a21" y se asigna a la tabla temporal N.



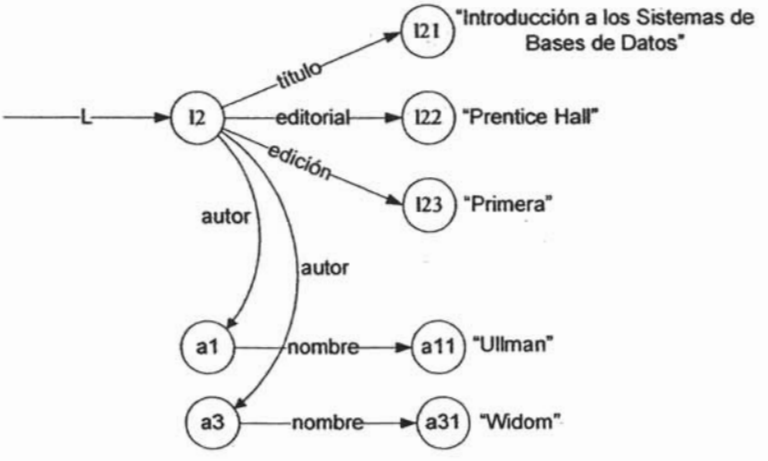
  - Evaluación de la cláusula WHERE:
    - Para la parte de 3 = COUNT (L PICK (autor)), como L no ha cambiado el resultado continúa siendo el mismo: verdadero.
    - Para N = "Ullman", el resultado es **falso**, pues en este instante N contiene el dato primitivo con valor "García-Molina".
    - El resultado final de la condición es **falso**.
  - Evaluación de la cláusula SELECT: Como el resultado de la condición en la cláusula WHERE es falso no se evalúa la construcción del dato semiestructurado de la parte del SELECT. Con esto termina el segundo *ciclo*.

- Tercer *ciclo*:
  - Evaluación de la cláusula FROM: La expresión de camino *L.autor.nombre* regresa el siguiente identificador, el cual es "a31" y se asigna a la tabla temporal N.

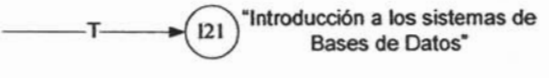


  - Evaluación de la cláusula WHERE:
    - Para la parte de 3 = COUNT (L PICK (autor)), como L no ha cambiado el resultado continúa siendo el mismo: verdadero.
    - Al igual que en el segundo *ciclo*, N es diferente a "Ullman" por lo que el resultado es falso.
    - El resultado final de la condición es **falso**.
  - Evaluación de la cláusula SELECT: No se construye el dato semiestructurado de la parte del SELECT.

- Cuarto *ciclo*:
  - Evaluación de la cláusula FROM: Como la expresión de camino *L.autor.nombre* no encuentra más identificadores diferentes a los regresados, la evaluación vuelve la segunda expresión de camino *L.título* que de igual forma no encuentra más identificadores en el estado actual de la consulta, por lo que retorna a evaluar la expresión *libros.libro*, la cual en esta ocasión devuelve el identificador "l2" y lo asigna a la tabla temporal L.

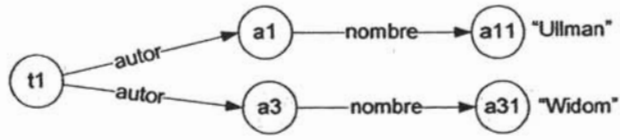


Se prosigue a evaluar de la expresión de camino *L.título* la cual devuelve el identificador "l21" y lo asocia con la tabla temporal llamada T.



Por último al evaluar la expresión de camino *L.autor.nombre* regresa el identificador "a11" y se asigna a la tabla temporal N.

  - Evaluación de la cláusula WHERE:
    - Para la parte de 3 = COUNT (L PICK (autor))
      - L PICK (autor) devuelve el nuevo dato semiestructurado siguiente:



- COUNT (t1) devuelve el dato semiestructurado primitivo 2.

- 3 = "dato primitivo 2" es falso, pues toda constante se toma como un dato primitivo.

▪ Como el primer operando del operador "AND" resultó falso, no es necesario evaluar el segundo operador. Por lo que el resultado final de la condición en el WHERE es falso.

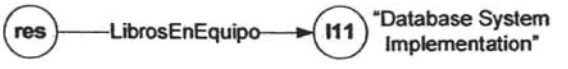
o Evaluación de la cláusula SELECT: Al resultar falsa la condición en la parte del WHERE no se evalúa la cláusula del SELECT. Con esto termina el cuarto *ciclo*.

- Quinto *ciclo*:
  - o Evaluación de la cláusula FROM: La expresión de camino *L.autor.nombre* devuelve el siguiente identificador, "a3" y se asigna a la tabla temporal N.

  - o Evaluación de la cláusula WHERE:
    - ▪ Para la parte de 3 = COUNT (L PICK (autor))
      - Como L no ha cambiado, la evaluación de esta expresión es igual a la del Cuarto *ciclo*, el cual es falso.

    - ▪ Como el primer operando del operador "AND" es falso, el resultado de la expresión también es falso.
  - o Evaluación de la cláusula SELECT: Al ser falsa la condición en la cláusula WHERE no se evalúa la cláusula SELECT. No hay más ciclos, pues al regresar a las expresiones de camino de la cláusula FROM, *L.autor,nombre* no tiene más identificadores diferentes que

regresar, por lo que se evalúa la expresión de camino *L.titulo*, que de igual forma no encuentra más identificadores y por último la expresión *libros.libro*, no localiza más identificadores diferentes a los que ya ha regresado. La evaluación de la consulta termina y devuelve el siguiente dato semiestructurado como resultado:



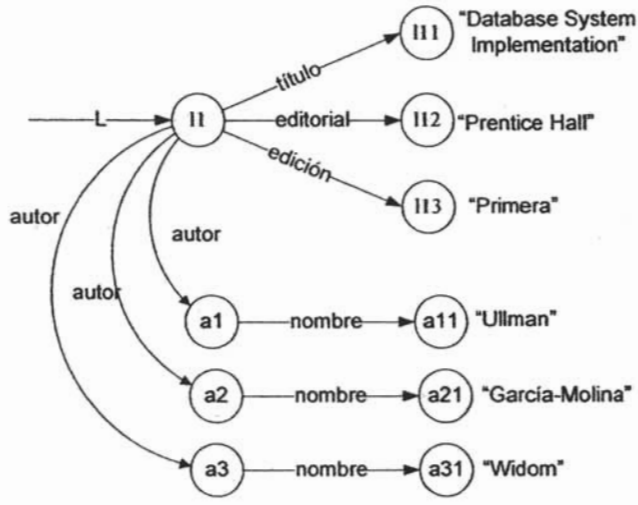
Ejemplo 1.4.2.3 Se desea borrar de la base de datos semiestructurados de la figura 1.4.2.2 los libros que tengan más de 2 autores. El enunciado para realizar esta operación es el siguiente:

```
DELETE L
FROM libros.libro L
WHERE COUNT (L PICK (autor)) > 2
```

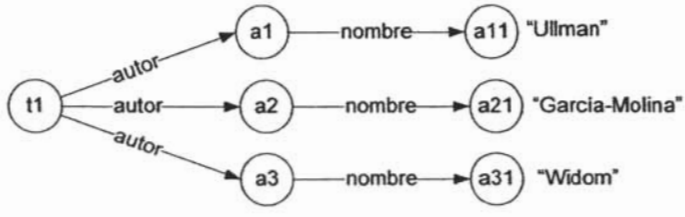
La ejecución de un enunciado para borrar datos es similar al enunciado de consulta Select-From-Where. La diferencia radica en la última etapa, aquí se eliminará de la base de datos aquel dato al que se refiera la tabla temporal especificada en la cláusula DELETE (sólo puede borrarse una de las tablas temporales especificadas en el FROM en un enunciado).

La evaluación de éste enunciado consta de dos *ciclos*, pues la expresión de camino *libros.libro* sólo puede devolver dos identificadores diferentes, l1 y l2. La ejecución se detalla a continuación:

- Primer *ciclo*:
  - Evaluación de la cláusula FROM: La expresión de camino *libros.libro* devuelve el identificador "l1" y se asigna a la tabla temporal L.



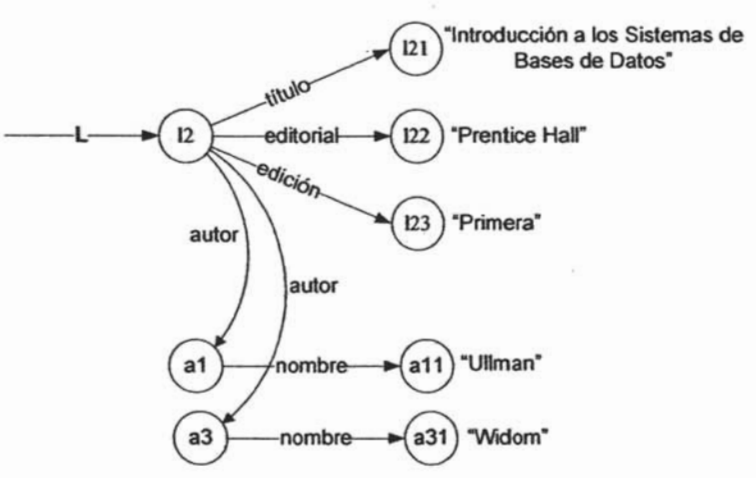
- Evaluación de la cláusula WHERE:
  - L PICK (autor) regresa un nuevo dato semiestructurado:



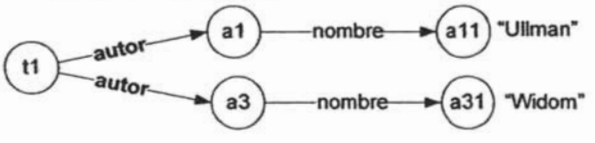
  - COUNT (t1) regresa un nuevo dato semiestructurado primitivo con valor 3.
  - La condición es verdadera pues el dato primitivo con valor 3 es mayor a la constante 2.

- Evaluación de la cláusula DELETE: Como la condición en la cláusula WHERE fue verdadera se marca el dato contenido en L para eliminarse al terminar la ejecución del enunciado de borrado.

- Segundo *ciclo*:
  - Evaluación de la cláusula FROM: La expresión de camino *libros.libro* devuelve el siguiente identificador "l2" y se asigna a la tabla temporal L.



- Evaluación de la cláusula WHERE:
  - L PICK (autor) regresa un nuevo dato semiestructurado:



  - COUNT (t1) regresa un nuevo dato semiestructurado primitivo con valor 2.
  - La condición es falsa pues el dato primitivo con valor 2 no es mayor a la constante 2.
- Evaluación de la cláusula DELETE: El dato contenido en la tabla temporal L (l2) no se marca para su eliminación pues la condición de la cláusula WHERE fue falsa.

- Los datos marcados para su eliminación se borran de la base de datos. La base de datos resultante después de la evaluación de este enunciado aparece en la figura 1.4.2.3.

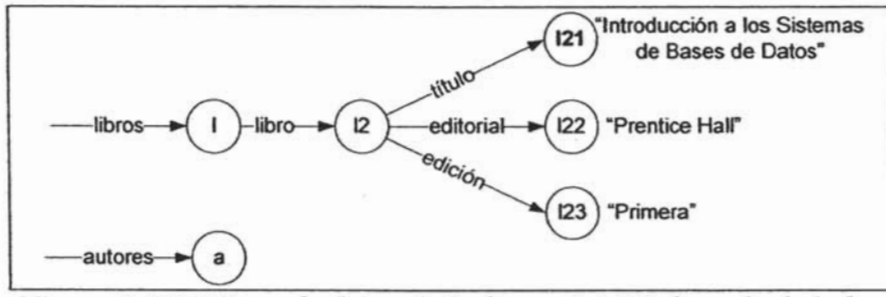


Figura 1.4.2.3 *Base de datos de la figura 1.4.2.2 después de haber borrado aquellos libros con más de dos autores*

Hay que observar en la figura 1.4.2.3 cómo afecta la eliminación de datos de una Ssd-tabla a otros datos, pues los autores de "l1" estaban compartidos con los de "l2", por lo que ahora "l2" no tiene ningún autor especificado.

Ejemplo 1.4.2.4 Se desea agregar al libro con título "Database System Implementation" de la base de datos semiestructurados de la figura 1.4.2.2 la información que es de pasta dura. El siguiente enunciado realiza esta operación:

```
UPDATE L
SET L UNION {pasta: "dura"}
FROM libros.libro L,
     L.título T
WHERE T = "Database System Implementation"
```

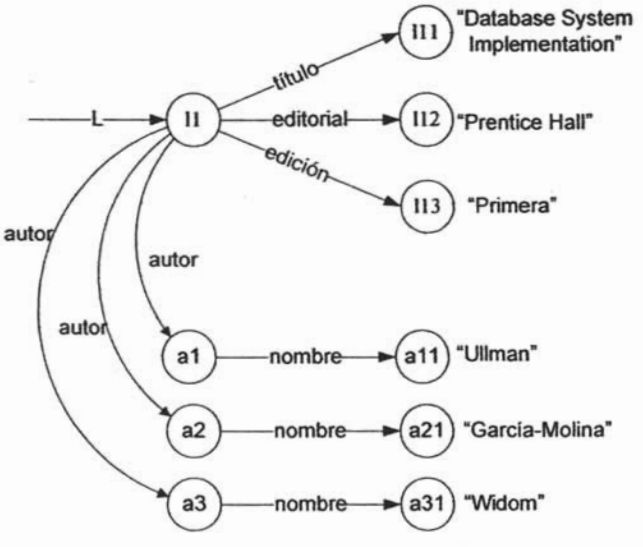
En esta ocasión se trata de un enunciado de actualización que contiene cuatro etapas, pero sólo se ejecutan tres por cada *ciclo*: 1ª. FROM, 2ª WHERE, 3ª UPDATE. El orden de ejecución de cada etapa fue definido en [Garc02]. Estas etapas son similares a las de un enunciado de borrado. Al igual que con la de la cláusula DELETE, la de la cláusula UPDATE sólo marcará los datos contenidos en la tabla temporal que se especifique. Únicamente cuando no queden más *ciclos* por evaluar se ejecutará la cuarta y última, la de la cláusula SET. Aquí todos los datos marcados anteriormente por la cláusula UPDATE se reemplazarán por la construcción definida en la del SET. Esta construcción es similar a la que se encuentra en la cláusula SELECT de un enunciado de consulta, sólo que en esta ocasión no se permite hacer referencia a las tablas temporales especificadas en el FROM, sólo se puede hacer referencia a la marcada por el UPDATE.

La ejecución de este enunciado de actualización sobre la base de datos de la figura 1.4.2.2 se compone de dos *ciclos*:

- *libros.libro* sólo puede devolver dos identificadores diferentes.
- *L.título* regresa un solo identificador por libro.

por lo tanto, la ejecución es la siguiente:

- Primer *ciclo*:
  - Evaluación de la cláusula FROM: La expresión de camino *libros.libro* regresa el identificador "l1" y se asigna a la tabla temporal L:

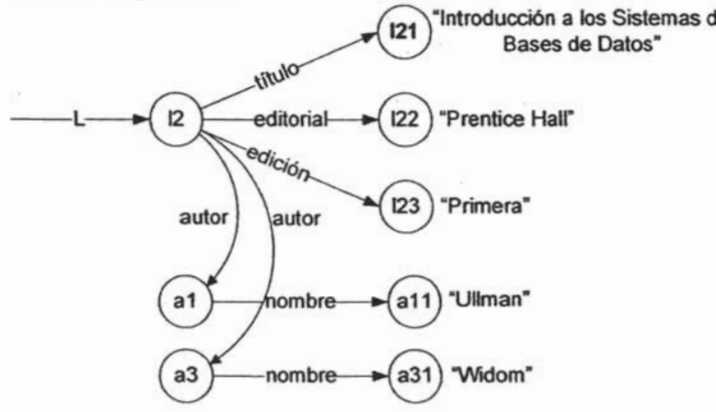


De igual forma la expresión de camino *L.título* regresa el identificador "l11" y se asigna a la tabla temporal T:

  - Evaluación de la cláusula WHERE: La condición T = "Database System Implementation" es verdadera, pues T contiene en este instante el dato primitivo con el mismo valor a la constante.

  - Evaluación de la cláusula UPDATE L: El identificador "l1" se marca para su posterior actualización.

- Segundo *ciclo*:
  - Evaluación de la cláusula FROM: La expresión de camino *L.título* no tiene más identificadores que regresar por lo que se continúa con la expresión de camino anterior. *libros.libro* regresa el identificador "l2" y se le asigna a la tabla temporal L:

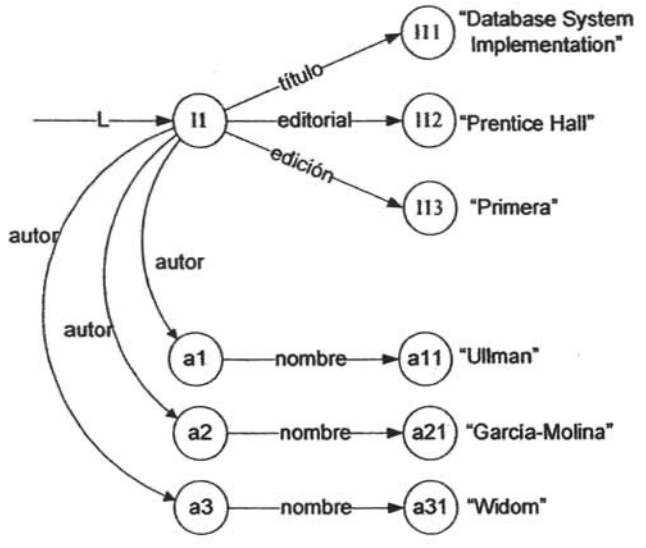


Se procesa de nueva cuenta la expresión de camino *L.título* para la nueva L, ésta regresa el identificador "l21" y se asigna a la tabla temporal con nombre T:

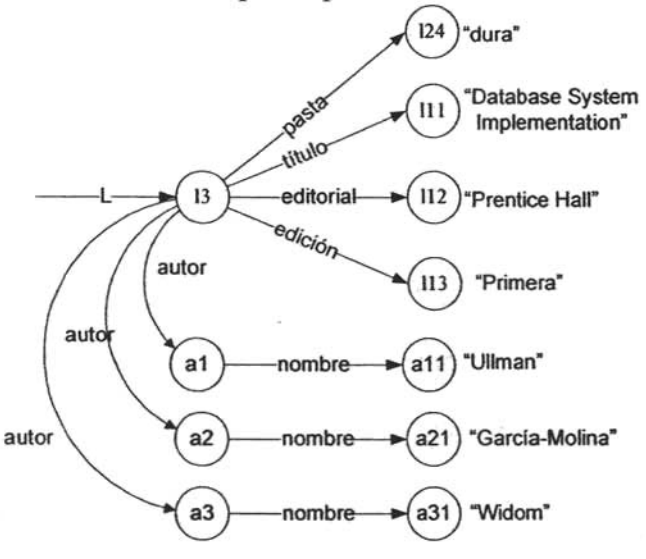
  - Evaluación de la cláusula WHERE: En esta ocasión la condición T = "Database System Implementation" es falsa, pues T contiene el dato primitivo con valor "Introducción a los Sistemas de Bases de Datos".

  - Evaluación de la cláusula UPDATE: Dado que la cláusula WHERE devolvió un resultado falso no se evalúa la cláusula UPDATE.

- Evaluación de la cláusula SET: Se retoman los datos semiestructurados marcados para su actualización, en este caso sólo fue aquel con identificador "l1":



- Se ejecuta la construcción especificada en el SET:
  - L UNION {pasta: "dura"}: Se construye un nuevo dato semiestructurado primitivo con valor "dura" y se une con lo que exista en L mediante una arista con etiqueta "pasta":



La base de datos resultante después de la ejecución de la actualización aparece en la figura 1.4.2.4.

Estos han sido algunos ejemplos de las construcciones propias del lenguaje Ssquirel. Existen otras utilizadas para la definición de datos (Ssd-

tablas) y vistas (abstractas y materializadas). Sin embargo, su estructura tiene como base los tipos de enunciados aquí presentados. Se recomienda consultar [Garc02] para obtener la definición completa del lenguaje y más ejemplos.

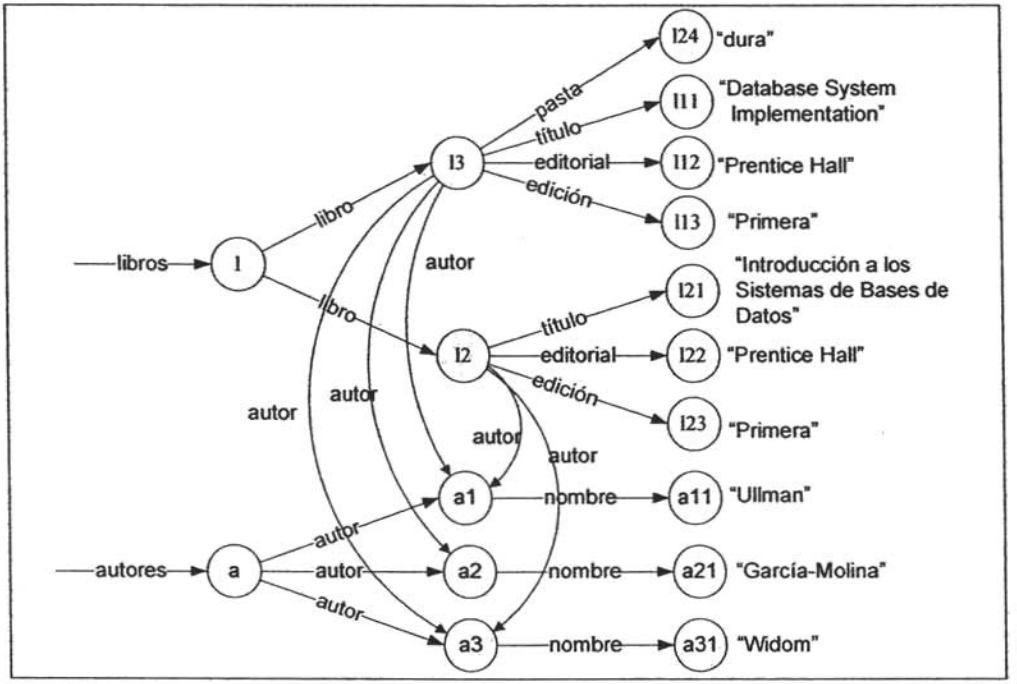


Figura 1.4.2.4 *Base de datos semiestructurados después de la ejecución de la actualización del ejemplo 1.4.2.4*

1.5 Resumen y discusión

Este capítulo introdujo al concepto de dato semiestructurado, cómo se originó, sus características y el modelo que utilizan las bases de datos que los almacenan. También se describió el lenguaje de consulta para datos semiestructurados Ssquirel que maneja el procesador de consultas que se trata en los siguientes capítulos.

Existen varios lenguajes de consulta que ha sido desarrollados para el manejo de datos semiestructurados [Suci98, AQM+97, W3C99, W3C04]. Para el caso de esta tesis, el procesador de consultas utilizará el lenguaje llamado Ssquirel [Garc02]. La elección se basó en que la sintaxis

de Ssquirel está muy apegada a la de SQL, lo que facilita su aprendizaje. Por otra parte, para los demás lenguajes de consulta ya se cuenta con sus respectivos reconocedores (*parsers*) o se emplean dentro de un sistema administrador de bases de datos [MAG+97]. Ssquirel, por ser tan reciente, no cuenta con ningún sistema de *software* que lo utilice, por lo que el procesador de consultas de este trabajo será el primero que reconozca este lenguaje.

¿Por qué es necesario un lenguaje especial para el manejo de datos semiestructurados? La mayor diferencia entre el lenguaje de consulta para datos semiestructurados Ssquirel y SQL para datos relacionales radica en el manejo tanto de los datos como del esquema. En Ssquirel es posible manejar ambas características con el uso de expresiones de camino (*path expressions*). Por lo anterior, estas expresiones constituyen un punto de atención en el desarrollo de esta tesis.

A manera de introducción se mostrarón algunos ejemplos de consultas, con sus respectivos despliegues de las operaciones que se realizan en cada *ciclo*. Comprender la ejecución de una consulta es clave para determinar cómo el procesador de consultas debe realizarlas. También es útil para el descubrimiento de nuevas técnicas de optimización, de forma que modifiquen una primera secuencia de ejecución a otra que requiera menor gasto de recursos del sistema.

Antes de describir qué es y para qué sirve el procesador de consultas, el capítulo siguiente detalla el ambiente en que trabaja y los módulos en los que se compone. Posteriormente, en el resto de la tesis, se profundiza en cada uno de estos módulos explicando cada uno de los procesos que en ellos se realizan.

Capítulo 2

Estructura del procesador de consultas

Previo a la descripción del procesador de consultas, es conveniente contar con la idea general del entorno en que trabaja, es decir, el sistema administrador de bases de datos (SABD). Por esto, inicialmente se muestra la estructura del SABD para posteriormente retomar al procesador de consultas con el objeto de presentar su estructura y función con más detalle.

2.1 Estructura del sistema administrador de bases de datos semiestructurados

Un SABD es una herramienta poderosa para crear, consultar y mantener eficientemente grandes cantidades de datos y permitir que persistan durante largos periodos de tiempo. Debido a la complejidad de un SABD, son varios componentes los que lo integran. Cada uno está encargado de realizar una tarea o función específica, como controlar la concurrencia de usuarios, el manejo de transacciones, etc. Estas funciones generalmente no varían entre los SABD, pues a lo largo del tiempo se ha visto que son necesarias. La variación entre ellos ocurre en el modelo de datos o en los algoritmos internos de cada componente, por lo que es posible tener un diagrama general de componentes de un SABD. La figura 2.1.1 muestra el esquema del sistema administrador de bases de datos semiestructurados (SABDSS), el cual es la base para este trabajo de tesis.

A continuación se describe brevemente cada componente y su función específica dentro del SABD.

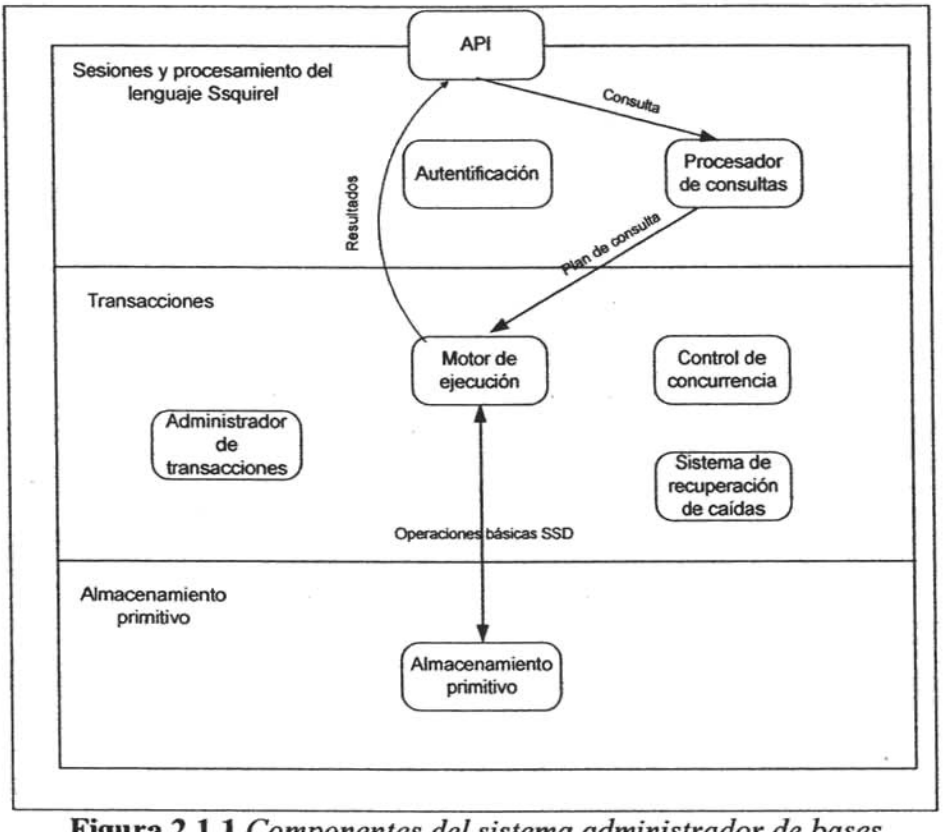


Figura 2.1.1 *Componentes del sistema administrador de bases de datos semiestructurados*

2.1.1 API

La API (Application Program Interface) es el medio de comunicación entre el SABD y el usuario. Como usuario se refiere a aquella entidad que hace uso del sistema, como puede ser otro sistema o aplicación, una persona encargada de la administración de la base de datos, o un programador de aplicaciones que utilice el SABD para ejecutar sus consultas. La API representa para el SABD el punto de entrada de las consultas, así como la parte en donde se regresarán los resultados al usuario.

Inicialmente se desarrolló un API tipo texto o línea de comandos, donde se escriben las consultas a ejecutar por el SABDSS y los resultados se imprimen en pantalla a manera de árboles.

El SABDSS también cuenta con una API gráfica, con la que se pretende ayudar al usuario a conocer la estructura de la base de datos en la que está operando, lo que más adelante describe el capítulo 4.

Estas dos APIs mencionadas (modo texto y modo gráfico) están orientadas al uso por un programador o por un administrador de la base de datos. Para la comunicación entre el SABDSS y otra aplicación de *software*, aún no se ha construido ninguna interfaz, como es la JDBC [SunM05] para el manejo de bases de datos relacionales en el lenguaje de programación JAVA.

2.1.2 Autentificación

Es común que un sistema multiusuario requiera de cierto nivel de seguridad y de acceso a diferentes partes del mismo. Un SABD no es la excepción. Se puede restringir el acceso a determinados usuarios en el manejo de una base de datos almacenada en el sistema, otorgándoles permisos de sólo lectura, de modificación, o de acceso sólo a ciertos datos.

Al momento de escribir esta tesis, las instrucciones y mecanismos de autentificación que tendrá el SABDSS todavía no están definidas y quedan como tema abierto para futuras contribuciones.

2.1.3 Motor de ejecución

El motor de ejecución es el componente encargado de llevar a cabo el plan físico de consulta, es decir, el resultado de convertir un enunciado escrito en un lenguaje de alto nivel –como es en este caso Ssquirel– en una serie de instrucciones primitivas, mismas que se describen en el capítulo 5.

Como se mencionó en el capítulo 1, en una consulta la lista de expresiones de camino definidas en la cláusula FROM así como la construcción del resultado definida por la cláusula SELECT se manejan como tablas temporales. De éstas últimas son de las que está a cargo del motor de ejecución.

Otra de las responsabilidades de este componente es la administración de memoria (*buffer manager*). Generalmente, hablar de una base de datos supone el manejo de gran cantidad de datos y es posible que, para la ejecución de alguna consulta, no se cuente con suficiente

memoria disponible, por lo que este componente utiliza técnicas para su adecuado uso.

Por último, al terminar de ejecutar el plan físico de consulta, el motor de ejecución regresa los resultados al API que se encarga de mostrarlos al usuario. Este componente y las técnicas utilizadas por él serán descritas a profundidad en el capítulo 6.

2.1.4 Administrador de transacciones

Este administrador acepta instrucciones de transacción desde una aplicación, las cuales le dicen cuándo inician y terminan las transacciones.

Una transacción es un grupo de una o más operaciones de consulta o modificación a una base de datos y constituye una unidad de trabajo. Toda transacción debe dejar a la base de datos en un estado consistente, por lo que existen cuatro propiedades a cumplir para garantizarlo:

1. Atomicidad.- Una transacción se debe ejecutar por completo o simplemente no ejecutarse en lo absoluto. Si es terminada o interrumpida por algún error o caída del sistema antes de concluir, la base de datos debe volver al estado que tenía antes de iniciarse dicha transacción.
2. Aislamiento.- Cada transacción debe aparentar ser ejecutada como si ninguna otra estuviera en operación al mismo tiempo.
3. Durabilidad.- Es la condición relativa a que el efecto de una transacción sobre una base de datos no debe perderse una vez que termina por completo.
4. Consistencia.- Antes y después de ejecutar una transacción válida, o varias al mismo tiempo, la base de datos debe mantener un estado de consistencia. Este punto se retomará más adelante en el control de concurrencia.

El administrador de transacciones trabaja conjuntamente con el sistema de recuperación de caídas y el control de concurrencia. El primero utiliza un registro de cambios (*log*) que le permite restaurar la base de datos a un estado consistente y con ello se asegura la atomicidad de las transacciones. El control de concurrencia, por su parte, está encargado del aislamiento y consistencia de las transacciones. Para ello emplea técnicas de planificación, las que a su vez recurren al uso de candados (*locks*) hacia

partes de la información que se están utilizando por una transacción, con el fin de que no sean accesadas por otra.

El administrador de transacciones también tiene bajo su responsabilidad la resolución de interbloqueos (*deadlocks*). Un *deadlock* puede llegar a ocurrir cuando un proceso P_A bloquea un recurso X y un proceso P_B bloquea a su vez un recurso Y; sin embargo, cada proceso no liberará su recurso sin antes acceder al del otro, lo que genera el conflicto.

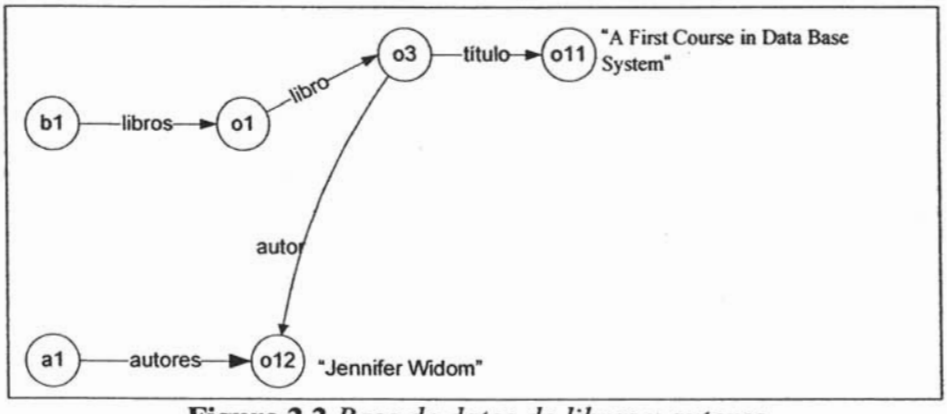


Figura 2.2 *Base de datos de libros y autores*

2.1.5 Control de concurrencia

El proceso general de asegurar que cada transacción sea consistente cuando se ejecuta simultáneamente con otras, se conoce como control de concurrencia [GaUW00].

Cuando un SABD maneja simultáneamente varios usuarios, se requiriere un mecanismo de control que impida que la base de datos quede en estado de inconsistencia. En las bases de datos relacionales, donde toda la información debe estar apegada a un esquema, la inconsistencia es un problema mucho más grave que en las de semiestructurados. En las de semiestruturados sólo se cuenta con la restricción de que los identificadores internos no se repitan en toda la base de datos y que los identificadores a los que se hace referencia en algún otro dato semiestructurado estén definidos en la misma.

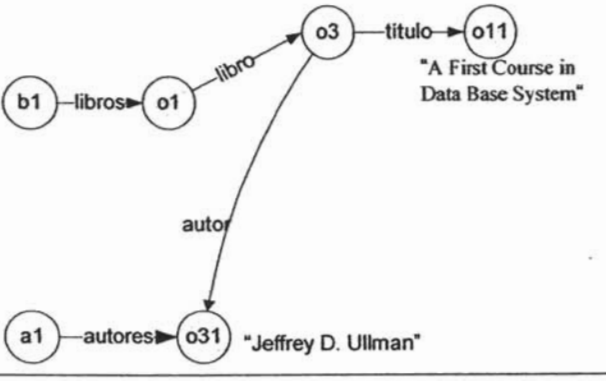
El control de concurrencia forma parte del procesamiento de transacciones, pues cada una de éstas debe parecer que fue ejecutada en aislamiento, es decir, como si el sistema sólo aceptara un usuario a la vez; sin embargo, en un SABD habrá varias transacciones ejecutándose al mismo tiempo. Otras características del procesamiento de transacciones serán presentadas más adelante.

A continuación se muestra un ejemplo de cuándo una base de datos semiestructurados puede caer en estado inconsistente por la carencia de un control de concurrencia.

Ejemplo 2.1.5.1 Existen dos usuarios que desean modificar la información de la base de datos de la figura 2.2. A continuación se presentan las modificaciones que cada usuario realiza y la estructura final que debería tener la base de datos si ambos trabajaran en dos bases de datos separadas con los mismos datos.

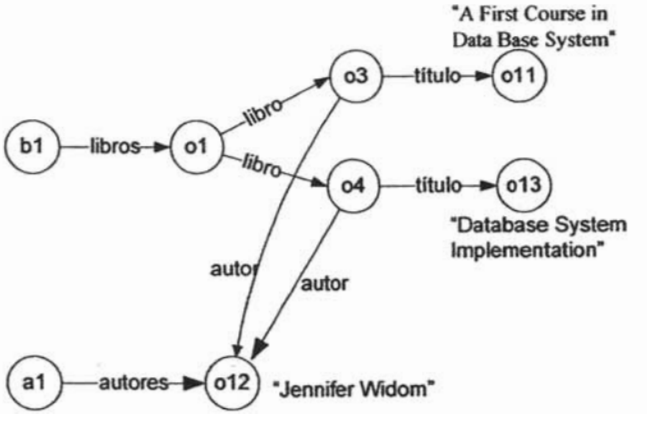
Usuario 1

- Agregar un nuevo autor con nombre "Jeffrey D. Ullman".
- Cambiar la relación "autor" del libro "A First Course in Data Base System" al nuevo autor agregado.
- Eliminar el autor "Jennifer Widom"



Usuario 2

- Agregar un nuevo libro titulado "Database System Implementation"
- Agregar la relación "autor" al libro recién incorporado con el autor "Jennifer Widom"



Si ambos usuarios trabajan sobre la misma base de datos simultáneamente y sin control de concurrencia, podría suceder que los dos ejecuten su última instrucción al mismo tiempo. Uno de ellos elimina el dato semiestructurado con identificador "o12" y el otro hace una referencia a este mismo, lo que deja a la base de datos en un estado de inconsistencia.

En las bases de datos relacionales existen métodos llamados planificadores (schedules), como los seriales y serializables, con los que se maneja el control de concurrencia. Éstos se basan en dar un orden de ejecución al conjunto de instrucciones de las transacciones operándose al mismo tiempo. Los planificadores seriales ejecutan una transacción completa a la vez, en cambio, los planificadores serializables buscan ejecutar varias transacciones simultaneamente ejecutando de forma mezclada el conjunto de instrucciones de cada una. Estos métodos tuvieron como base las técnicas utilizadas por sistemas operativos multiusuarios, razón por la que también manejan los llamados candados a partes de la información que están siendo ocupadas por algún usuario.

El control de concurrencia es un proceso que todo SABD que maneje varios usuarios al mismo tiempo debe tener y, en general, su objetivo es garantizar la consistencia de la base de datos. Sin embargo, las restricciones definidas por el modelo de datos que utiliza ocasionan que los métodos empleados por cada SABD tengan que ser acoplados al mismo.

2.1.6 Sistema de recuperación de caídas

Un SABD debe ser capaz de manejar situaciones que pueden salir mal mientras está en ejecución. Estos problemas ocurren en varios niveles, desde una entrada incorrecta a través del teclado hasta una pérdida total del sistema.

Para una entrada incorrecta, un sistema administrador de bases de datos elacionales cuenta con un conjunto de restricciones (*constrains*) que deben seguir los datos y así detectar fechas incorrectas, longitud de cadenas, números incorrectos, etc. Sin embargo, para un SABDSS el problema no es tan fácil de manejar. Debido a la libertad intrínseca de los datos semiestructurados, no existe definición cercana a una restricción como en una base de datos relacionales, por lo que detectar una entrada incorrecta de datos es tarea no apta para un SABDSS.

Existen otros problemas para los que un SABD debe estar preparado para manejar, como las fallas de los medios de almacenamiento[1] o la pérdida del estado de una transacción, mejor conocida como falla del sistema (*crash*). Esto último ocurre debido a la interrupción inesperada del servicio del SABD, lo que comúnmente sucede a causa de un corte en el suministro de energía eléctrica.

Toda transacción tiene un estado que representa el valor de las variables locales de la misma, como las tablas temporales, resultados o el registro del *ciclo* y etapa en que se encuentra la ejecución. Perder dicho estado de una transacción implica, casi con seguridad, que dejará a la base de datos en estado de inconsistencia. Si esto ocurriera, la transacción no sería terminada, lo cual entra en conflicto con la propiedad de atomicidad, definida exactamente para no caer en este tipo de problemas. Por ejemplo, cuando se establece una actualización como la siguiente:

```
UPDATE C
SET    C + 400
FROM   empleados.capturista.salario as C
WHERE C < 2300
```

donde se aumenta el salario en 400 a los empleados capturistas que ganan menos de 2300, el sistema tendrá tantos *ciclos* como datos de la expresión de camino *empleados.capturista.salario* encuentre. Si en determinado *ciclo* la ejecución falla, no se podrá conocer a cuales de ellos ya se les aplicó el aumento. Ejecutar nuevamente la operación sería un error, pues algunos capturistas podrían recibir doble aumento; si en la primera ocasión tenían como salario 1000, para la segunda tendrían 1400, por lo que de nueva cuenta se les aumentaría el sueldo.

Es cierto que existen modificaciones como la anterior donde puede no ser necesario escribir nada al medio de almacenamiento permanente (disco), pues el número de *ciclos* y, por consiguiente, de datos semiestructurados que se tienen que manejar son tan pocos, que pueden ser modificados en memoria. Solamente se deben grabar en el disco cuando termine la transacción (*commit*). Sin embargo, esta acción queda a cargo del sistema que maneja los buffers de memoria, por lo que no es correcto asegurar que toda transacción se realiza en memoria.

---

[1] El uso de arreglos de discos RAID, copias de seguridad o copias redundantes distribuidas, son algunas de las técnicas utilizadas para manejar problemas de medios de almacenamiento.

El sistema de recuperación de caídas es el encargado de restaurar la base de datos a un estado consistente si alguna transacción no se terminó, ayudando a cumplir con la propiedad de atomicidad. Como un medio para garantizar que las operaciones sean atómicas, hace uso de un registro de eventos importantes (*logging*) realizados sobre la base de datos, tales como capturar un nuevo dato en disco, pues éstos modifican la base de datos realmente. Este tipo de operaciones se pueden encontrar aún en las consultas si es necesario guardar un resultado intermedio.

Cuando una transacción no termina su ejecución por algún motivo, como falla en el sistema o la realización de una operación indebida dentro de la transacción (división entre 0), se tienen dos opciones: deshacer los cambios hechos por la transacción y dejar la base de datos en el punto en que se encontraba antes o guardar la última transacción completa. A estas opciones se les conoce como Undo-logging y Redo-logging respectivamente.

Para realizar un *Undo-logging*, se procede de la siguiente manera: registrar cada inicio de transacción; guardar en el registro los valores que tenía un dato antes de la modificación y almacenarlo en el disco; el nuevo valor del dato grabarlo en el disco; sólo hasta que todos los cambios contenidos en la transacción se hayan realizado y guardado registrar que la transacción se completó escribiendo un *commit* en el registro e inmediatamente guardarlo en disco. Para recuperar una falla de sistema con esta modalidad hay que regresar los valores anteriores, guardados en el registro, de aquellas transacciones que no tienen su *commit*. Una de las desventajas de esta técnica es el aumento de accesos al disco, pues los cambios deben reflejarse lo más pronto posible en él para decir que una transacción se hizo completa.

Efectuar un registro con la técnica de *Redo-logging* implica guardar los valores nuevos, al contrario de lo que sucede con *Undo-logging*. Antes de modificar la base de datos en disco es necesario que en el registro se encuentre la transacción completa, es decir, con su inicio, las modificaciones con sus nuevos valores y su final (*commit*). Para recuperar la base de datos de una falla se toman en cuenta las transacciones completas y se regraban en disco sus movimientos.

Existe una técnica híbrida, pero ésta gasta más espacio en el registro al guardar tanto valores anteriores como nuevos de cada dato. Sin embargo, al recuperar la base de datos se puede elegir entre cual de ellos utilizar dependiendo si la transacción se completó o no.

2.1.7 Almacenamiento primitivo

Este componente recibe las instrucciones primitivas (físicas) provenientes del motor de ejecución que evalúa la consulta. Está encargado de manejar los índices y estructuras en las cuales se almacenan físicamente los datos.

A diferencia de los registros grabados en el medio de almacenamiento secundario (disco) de un sistema administrador de bases de datos relacionales (que tienen un número preestablecido y fijo de campos y cada uno requiere de análisis para poder aprovechar al máximo el tamaño del bloque del disco), para el modelo de datos semiestructurados tal concepto de registro no existe y por ello la distribución de la información tiene que considerarse de forma diferente.

La organización de los datos en el medio de almacenamiento es también un punto muy importante, pues de ésta depende en gran parte qué tan rápido y eficiente sea posible recuperar la información. Por ejemplo, si la información está contenida en un número de cilindros continuos, la recuperación será más rápida que si se encuentra repartida entre varios, debido a que en este último caso la cabeza lectora del disco tendrá que realizar un mayor recorrido.

Como se mencionó en el apartado 1.3, cada dato semiestructurado cuenta con un identificador único, que no se puede repetir en toda la base de datos, por lo que éste representa una buena oportunidad para construir un índice sobre ellos.

Paralelamente a este trabajo, los componentes descritos en este capítulo (con excepción del procesador de consultas y el motor de ejecución) están en desarrollo como parte de otra tesis.

2.2 El procesador de consultas

El procesador de consultas es el encargado de traducir las instrucciones de alto nivel a un plan físico de consulta, el que utiliza instrucciones de bajo nivel llamadas instrucciones primitivas. Este plan físico de consulta pasa al motor de ejecución, que tiene a su cargo ejecutar las instrucciones en un ambiente controlado. Se puede pensar que el procesador de consultas es como un traductor o intérprete. Resulta común encontrar en libros de compiladores [AhSU90] el término intérprete de consultas para bases de datos, pues con cada enunciado se realiza una lectura-traducción-ejecución. Sin embargo, en documentación especializada en el tema [GaUW00] recibe la denominación de procesador de consultas, por lo que fue el término que se utilizó en esta tesis.

El procesador de consultas se divide en los siguientes seis subcomponentes:

- Analizador léxico
- Analizador sintáctico
- Analizador semántico
- Generador del plan lógico de consulta
- Optimizador de consulta
- Generador del plan físico de consulta

El proceso realizado dentro del procesador de consultas aparece en la figura 2.2.1. A continuación se describe cada subcomponente y su función dentro del proceso de conversión de la consulta de un lenguaje de alto nivel a instrucciones primitivas.

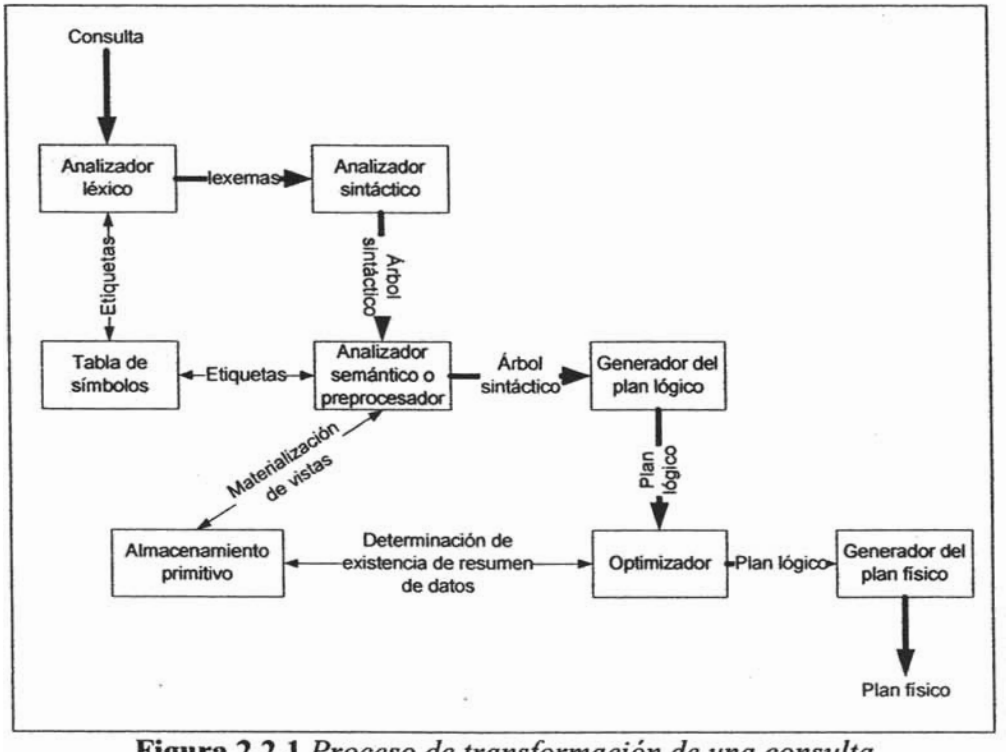


Figura 2.2.1 *Proceso de transformación de una consulta dentro del procesador de consultas*

2.2.1 Analizador léxico

Para iniciar el proceso de consulta, es necesario dividir ésta en la serie de unidades (tokens) que la componen. Además, se debe identificar la categoría a la que pertenece cada una (identificador, palabra reservada, cadena de caracteres, etc.). El analizador léxico es el subcomponente encargado de dividir el texto de entrada, en este caso una consulta en lenguaje Ssquirel, en una secuencia de tokens. Se ignoran los espacios en blanco, saltos de línea y todos aquellos caracteres especiales que, si bien contribuyen a la estética del enunciado, no se utilizan (ni son necesarios) para reconocer que la consulta cumple con las reglas del lenguaje. En ocasiones, estos caracteres ayudan a identificar la separación entre tokens.

Un token es una unidad indivisible de un lenguaje y puede ser una palabra reservada, un identificador, un signo de puntuación, etc. El analizador léxico hace uso de expresiones regulares que describen las

cadenas que representan un token específico. Por ejemplo, una expresión regular que reconozca una palabra que inicie con letra seguida de letras, números o caracteres de subrayado, es la siguiente:

```
(a..Z)( (a..Z) | (0..9) | _ )*
```

La expresión (a..Z) hace referencia a todas las letras del alfabeto (minúsculas y mayúsculas) y a su vez (0..9) a los números del 0 al 9. El signo "|" representa la posibilidad de alternativas. El signo de asterisco localizado al final de toda esta expresión se le conoce como la cerradura de Kleene, con la cual se define que las cadenas que pertenezcan al lenguaje reconocido por esta expresión regular "(a..Z) | (0..9) | _" pueden tener cero o más repeticiones de letras, números o guión bajo, después de una letra. Existen otros operadores para manejar el número de caracteres dentro de las expresiones regulares, como el operador + y ?, que representan una o más ocurrencias y cero o una ocurrencia, respectivamente. En el Apéndice A se muestran todas las expresiones regulares utilizadas por el procesador de consultas.

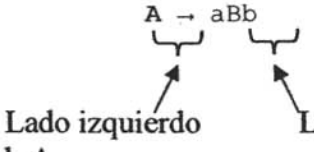
En general, el analizador léxico realiza las siguientes funciones:

- Aisla los tokens que componen el código fuente.
- Identifica el tipo al que pertenece cada token (identificador, constante, palabra reservada, etc.)
- Devuelve una secuencia de símbolos (los cuales representan identificadores, constantes, palabras reservadas o igualmente símbolos como el "+" o el "-") necesarios para el análisis sintáctico.

2.2.2 Analizador sintáctico

Cada consulta debe apegarse a una sintaxis dependiendo de su función (actualización, borrado, consulta, etc.). El analizador sintáctico recibe la secuencia de tokens del analizador léxico y verifica que la estructura sintáctica de la consulta se apegue a las reglas del lenguaje. Por ejemplo, un enunciado de consulta SELECT-FROM-WHERE debe declararse en orden específico, primero la cláusula SELECT, después la FROM y por último la WHERE. Estas reglas, generalmente, se representan por medio de una gramática, en este caso, independiente del contexto. La gramática

consiste en un conjunto de producciones compuestas de un lado izquierdo que "produce" a un derecho, formado de símbolos terminales y no terminales, además de uno especial no terminal llamado inicial. Por ejemplo:



Símbolo inicial: A
Símbolos no terminales: A, B
Símbolos terminales: a, b

Cualquier consulta que no esté apegada a estas reglas será rechazada y el análisis de la misma no continuará.

Existen herramientas que ayudan a desarrollar los analizadores descritos anteriormente (léxico y sintáctico). Por lo general, se ingresan las expresiones regulares para el analizador léxico y la gramática a seguir por el analizador sintáctico y la herramienta devuelve un código de alto nivel del cual, una vez compilado, se obtiene un programa que reconoce el lenguaje descrito por la gramática. Para el procesador de consultas objeto de este trabajo se utilizó como herramienta JavaCC, cuya descripción y funcionamiento así como las expresiones regulares y gramática elaboradas se detallan en el apéndice A.

2.2.3 Analizador semántico

En algunas ocasiones en la literatura sobre sistemas administradores de bases de datos recibe el nombre de pre-procesador. Este subcomponente es el encargado de realizar verificaciones sobre la consulta, a la salida del analizador sintáctico.

Además de verificar que la estructura del enunciado de consulta sea válida, es decir, que esté apegada a las reglas del lenguaje (verificación sintáctica), se efectúa la revisión semántica para comprobar que los recursos utilizados dentro del enunciado sean correctos. Por ejemplo,

comprobar el alcance de las variables internas de la consulta. En resumen, las funciones del analizador semántico son las siguientes:

a) Revisar que cada expresión de camino tenga un inicio válido
b) Preparar una vista no materializada o abstracta para su uso dentro de la consulta

Estas funciones se profundizan a continuación.

2.2.3.1 Revisión de expresiones de camino

Toda expresión de camino debe tener un punto donde inicia el recorrido del árbol de datos, el cual puede ser la etiqueta de una Ssd-tabla contenida dentro de la base de datos, la etiqueta de una vista o el nombre de una tabla temporal válida dentro del alcance.

Ejemplo 2.2.3.1 Considerando la base de datos de la figura 2.2.3.1 se analiza semánticamente la siguiente consulta:

```
SELECT      programador:{P UNION (
                SELECT      proyecto: N
                FROM        proyectos.proyecto AS R,
                            R.nombre AS N,
                            R.participante AS X
                WHERE       X IS P)
FROM        programadores.programador AS P,
            P.salario AS S
WHERE       S > 23000
```

En este enunciado se utiliza un anidamiento de consultas. En la exterior se recuperan todos los *programadores* con un sueldo mayor a 23,000 almacenados en la base de datos, mientras que en la interior se recuperan los *proyectos* en los que cada *programador* participa. Para realizar un enlace entre las dos consultas es necesario compartir una etiqueta de una tabla temporal, que en este caso es "P", por lo que parte del trabajo del pre-procesador es verificar el alcance de las etiquetas declaradas en la cláusula FROM.

En la declaración de las tablas temporales de la cláusula FROM, las expresiones de camino deben iniciar con la etiqueta de una Ssd-tabla, vista o tabla temporal, definida anteriormente. Lo anterior se se cumple en este ejemplo, pues al encontrarse con una expresión de camino la definición de la etiqueta inicial ya se conoce.

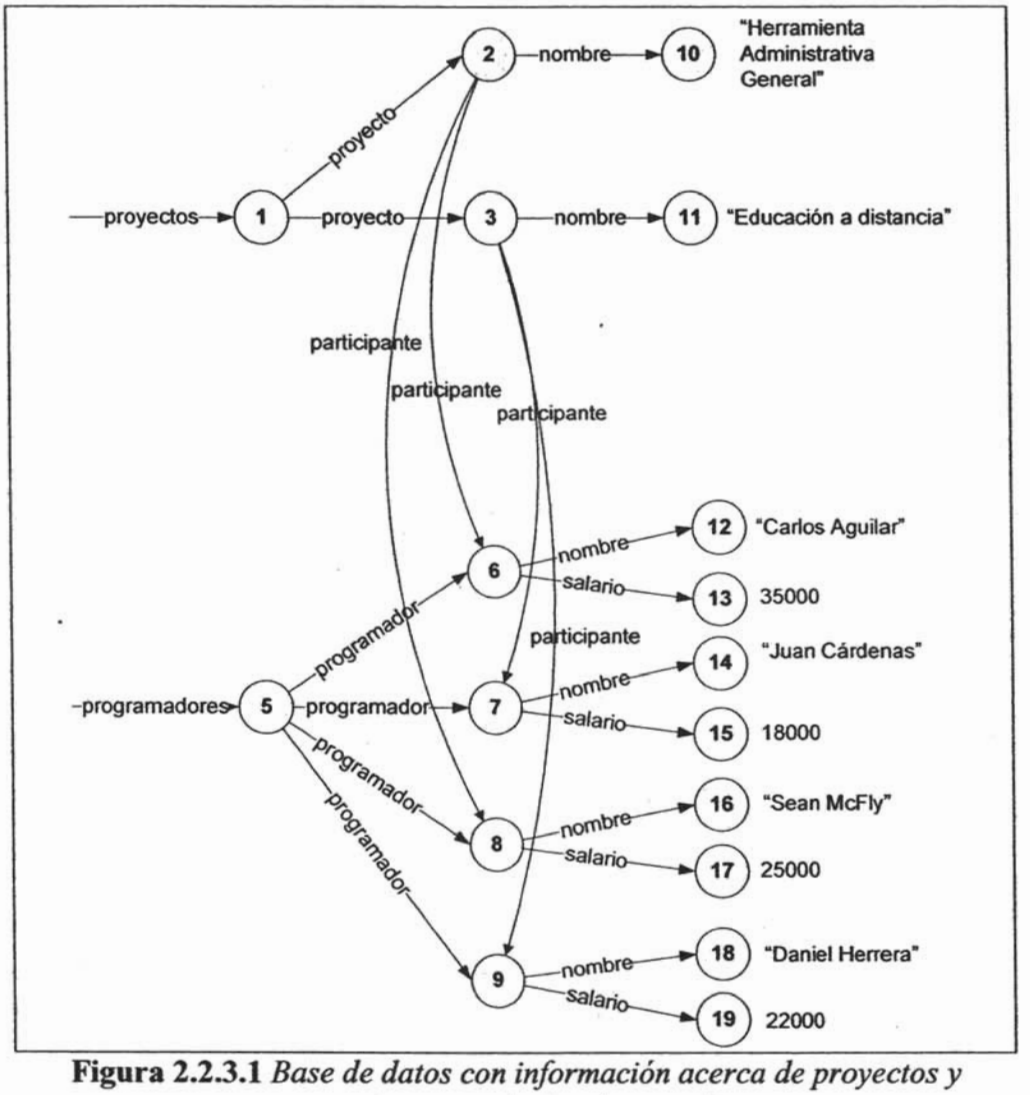


Figura 2.2.3.1 *Base de datos con información acerca de proyectos y recursos humanos destinados a cada uno*

Hay que recordar que la primera cláusula a evaluar es FROM, por lo que al evaluar la consulta interna que se encuentra en la cláusula SELECT la tabla temporal "P" ya se ha definido. En las consultas hay que seguir el

orden de la definición de las tablas temporales, pues si se escribiera la cláusula FROM de la consulta interna de este ejemplo como:

```
FROM  R.participante AS X,
      R.nombre AS N,
      proyectos.proyecto AS R
```

sería incorrecto, ya que al tratar de evaluar la primera expresión de camino R.participante, se tendría definido el alcance de etiquetas como {"proyectos", "programadores", "P"} donde no existe la definición de "R". Esto provocaría caer en error.

En la consulta externa se agrega, al conjunto de etiquetas válidas para un inicio de expresión de camino, la etiqueta "P", esto es {"proyectos", "programadores", "P"}. Por consiguiente, al entrar a la consulta interna se le pasa este nuevo conjunto de etiquetas como si se tratara del conjunto de las Ssd-tablas de la base de datos. Por ello, el conjunto de etiquetas para la consulta interna es: {"proyectos", "programadores", "P", "R", "N", "X"}, e indica que toda tabla temporal definida en una consulta es válida para toda consulta anidada internamente.

Con base en lo anterior, si en la consulta interna se declarase nuevamente una tabla temporal con la etiqueta "P":

```
SELECT      programador:{P UNION (
                              SELECT      proyecto: P
                        FROM  proyectos.proyecto AS R,
                                    R.nombre AS P,
                                    R.participante AS X
                        WHERE X IS P)
FROM  programadores.programador AS P,
          P.salario AS S
WHERE S > 23000
```

no se tomaría la definición más cercana de la tabla temporal, sino que nuevamente sería una consulta incorrecta, pues habría una repetición de etiquetas. Esto aplica no sólo entre etiquetas de tablas temporales, sino además con etiquetas de Ssd-tablas y vistas.

2.2.3.2 Preparación de una vista abstracta

Existen dos tipos de vistas definidas en el lenguaje Ssquirel, las materializadas y las abstractas. Las primeras se ejecutan y guardan su resultado al momento de su creación. Para las segundas sólo se guardan sus definiciones y, antes de que se utilicen por cualquier consulta, deben ejecutarse a modo de obtener los datos que contenga (ejecutar y guardar su resultado). Por lo anterior, las vistas materializadas no reflejan los cambios sobre la base de datos, en cambio las vistas abstractas sí lo hacen.

La preparación de una vista abstracta (ejecutar y guardar la información) es una de las tareas del pre-procesador de consultas.

Ejemplo 2.2.3.2 Si existe una vista abstracta dentro de la base de datos de la figura 2.2.3.1 definida como:

```
CREATE VIEW proy
WITH (SELECT      proyecto : X
       FROM proyectos.proyecto AS X)
```

y sobre ella se evalúa la siguiente consulta:

```
SELECT programador:{P UNION (
                  SELECT      proyecto: N
                  FROM        proy.proyecto AS R,
                              R.nombre AS N,
                              R.participante AS X
                  WHERE X IS P)
FROM  programadores.programador AS P,
           P.salario as S
WHERE S > 23000
```

En este ejemplo se utiliza la vista abstracta definida anteriormente. Por tratarse de este tipo, es necesario ejecutar la consulta descrita por la vista asignando el resultado a una tabla temporal de nombre "proy" y agregarla al alcance de dicha consulta que la invocó. En el caso de vistas materializadas, esto no rige, pues éstas son tratadas como Ssd-tablas, por lo que no es necesario ningún pre-procesamiento antes de poder utilizarlas.

2.2.4 Generador del plan lógico de consulta

Es el subcomponente encargado de realizar la transformación de la consulta a plan lógico. La razón de efectuar ésto es realizar un análisis interno de la misma para lograr alguna optimización. En las bases de datos relacionales, un plan lógico de consulta es aquel constituido sólo por operadores relacionales (selección, proyección, producto cruz, etc). Un plan lógico de consulta generalmente se representa por un árbol de expresión, debido a que varios operadores del álgebra relacional pueden ser combinados aplicando un operador al resultado de uno o más operadores distintos.

A nivel lógico, las optimizaciones se rigen por equivalencias entre operadores. Por ejemplo, en el álgebra relacional existen las reglas asociativas y conmutativas entre los operadores de unión, intersección y producto cartesiano. Otras técnicas, como *push selections down* [GaUW00], son utilizadas en el nivel lógico para la reducción de accesos a disco.

Los operadores utilizados a nivel lógico se definen para representar una tarea simple, no tan elemental como a nivel físico ni tan compleja como lo es una consulta en un lenguaje de alto nivel (como Ssquirel). En otras palabras, al transformar una consulta a plan lógico, se realiza una división en tareas simples. Por ejemplo, las expresiones de camino definidas en la cláusula FROM de una consulta pueden ser escritas de forma sencilla:

```
FROM autor.nombre AS N
     N.ap AS P
     N.am AS M
```

las que al transformarse en un árbol de expresión a nivel lógico, proporcionan una descripción de los pasos a realizar para llevar a cabo la consulta con un punto de vista de cómo lo realizará la computadora, sin entrar en detalles específicos del almacenamiento primitivo. Para este ejemplo en particular se definirá el manejo de los *ciclos* de la consulta.

Como se ha expresado, al transformar una consulta a plan lógico, se realiza una división en tareas simples, lo que la hace manejable por parte de los algoritmos de análisis y optimización que se aplicarán posteriormente. Si se convirtiera directamente la consulta a plan físico, el análisis y aplicación de técnicas de optimización serían más complejos al

tener que revisar un mayor número de instrucciones de las que en realidad se realizan en una tarea simple. Por el otro lado, analizar una consulta directamente escrita en un lenguaje de alto nivel, no permite mucha libertad para efectuar alguna optimización, como por ejemplo en las transacciones, debido al concepto de tarea tan amplio.

Para transformar una expresión de camino a plan lógico, se emplea una simple función de recorrido del árbol, mientras que para el plan físico podrían necesitarse varias llamadas a alguna o algunas funciones. El ejemplo 2.2.4.1 lo demuestra.

Ejemplo 2.2.4.1[2] Transformaciones de una expresión de camino a plan lógico y a plan físico.

Expresión de camino en la cláusula FROM	*Plan lógico*	*Plan físico*
Libros.libro.autor.nombre as X	crearTablaTemp(X, recorrer(Libros.libro.autor. nombre))	crearTabla("X") para cada libro en Libros para cada autor en libro para cada nombre en autor link("X", nombre)

Los operadores lógicos para esta transformación y su manejo por el procesador de consultas para datos semiestructurados de esta tesis, se definirán en el capítulo 3. Estos operadores tienen fundamento en trabajos anteriores efectuados por la Universidad de Stanford [McWi97] y la de Eindhoven [FrHP02], con algunas diferencias debido a variaciones entre el manejo de índices y técnicas de optimización.

Para la transformación de una consulta a un árbol de expresión se utilizó una de las herramientas anexas a JavaCC denominada JJTree. Su descripción y cómo se usó en este trabajo se encuentran en el apéndice A. El diseño de JJTree obedeció al propósito de ayudar a construir el árbol de análisis sintáctico del código reconocido por JavaCC, es decir, el analizador léxico y sintáctico. Un árbol de análisis sintáctico es una representación gráfica de las producciones realizadas para reconocer una consulta (cada nodo es un no terminal y las hojas son terminales) y a partir

[2] Los nombres de los operadores lógicos y físicos de este ejemplo no son los que realmente utiliza el procesador de consultas. Se presentan de esta forma para hacerlo comprensible, sin la necesidad de mencionar la definición de cada uno de los operadores.

de éste se obtiene el árbol de expresión de la consulta, es decir, el plan lógico.

Tanto la descripción de los operadores lógicos como la transformación de un enunciado de consulta en lenguaje Ssquirel a plan lógico, son los temas principales del capítulo 3.

2.2.5 Optimizador de consultas

La salida del generador del plan lógico de consulta (el árbol de expresión) pasa al optimizador de consultas para su análisis en busca de alguna posibilidad de aplicar una técnica que permita ejecutar la consulta con menor número de accesos a disco. Puede ejecutarse una consulta sin pasar por este subcomponente de optimización, pero tal vez requeriría la utilización de un mayor número de accesos a disco que si se aplica una optimización antes de evaluarse. Como se mostrará más adelante, esta diferencia puede llegar a ser de varios millones de accesos menos.

El número de accesos a disco es una de las métricas de mayor importancia cuando se habla del desempeño de un SABD, pues leer datos desde disco es de las tareas que hace más lento al sistema en general. Por consiguiente, la tarea del optimizador de consultas consiste en aminorar este número de accesos. Al llegar la consulta a esta etapa, el optimizador de consultas la analiza y elije qué tipo de técnicas pueden ser aplicadas para mejorar el desempeño del SABD al evaluarla.

La forma de ejecución de una consulta se mostró en la sección 1.4.2, la cual tiene como base el número de *ciclos* y el de *etapas*. El número de *etapas* en cada *ciclo* está en función al tipo de consulta a ejecutar (consulta de datos, modificación de datos, borrado de datos) y en ocasiones una de ellas queda condicionada al resultado de la cláusula WHERE, por lo que se puede agregar que el valor de los datos semiestructurados también define el número de *etapas* en cada *ciclo*. Por otro lado, el número de *ciclos* en cada consulta se define por el número de combinaciones de valores diferentes (identificadores de datos semiestructurados) asignados a cada una de las tablas temporales especificadas en la cláusula FROM. Por lo anterior, en cada *ciclo* se evalúa uno o varios datos semiestructurados que al menos cumplen con encontrarse en algún camino definido en dicha cláusula. Reducir el número de *ciclos* y *etapas* por *ciclo* puede ayudar a mejorar el desempeño

del SABD, aunque no se tenga la seguridad de una reducción en el número de accesos a disco, pues lo que disminuye es el número de instrucciones necesarias para ejecutar la consulta.

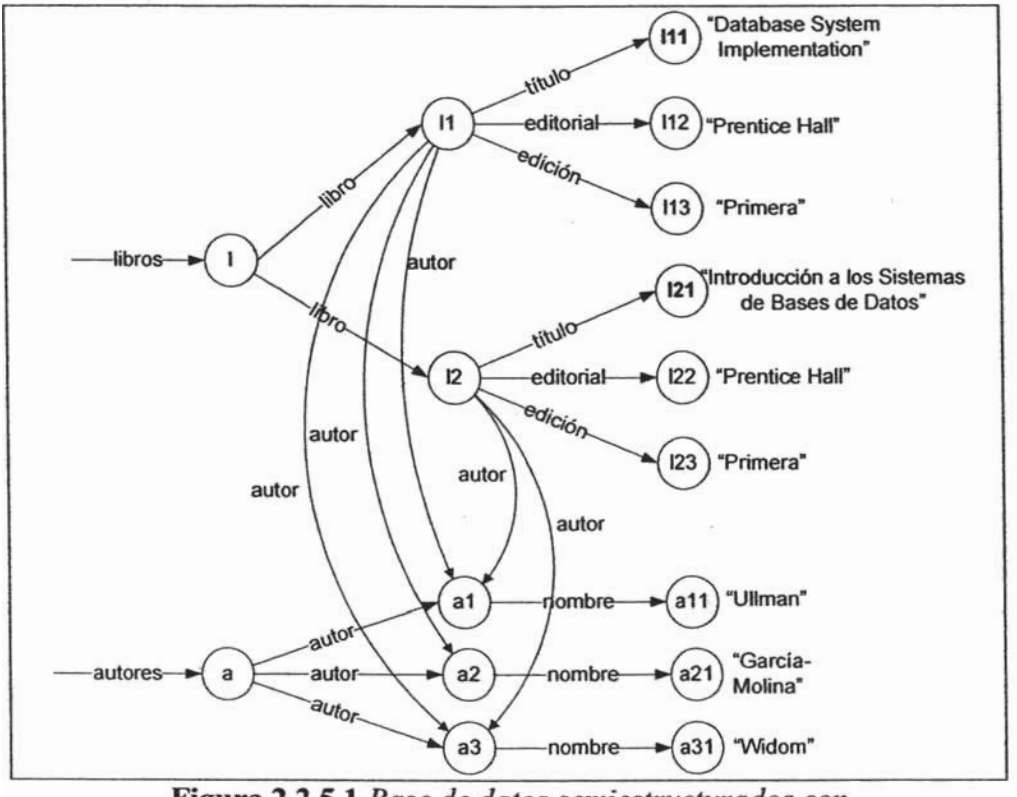


Figura 2.2.5.1 *Base de datos semiestructurados con información de libros y sus autores*

Ejemplo 2.2.5.1

```
SELECT      títulos: T
FROM        libros.libro AS L,
            L.autor.nombre AS A,
            L.título AS T
WHERE       A = "García-Molina"
```

Al evaluar esta consulta en la base de datos de la figura 2.2.5.1, habrá algunos *ciclos* en donde no se cumpla la condición de que el nombre del autor sea "García-Molina" y por ello, no se ejecute el constructor de la cláusula SELECT. Este constructor genera el resultado utilizando la tabla temporal T y adiciona todo lo que contenga al resultado interno, el cual la

consulta devuelve como si fuera otra tabla. Lo anterior puede dar pie a pensar que al no ejecutarse el constructor podrían evitarse algunos accesos a disco si la "tabla" resultado no se actualiza. Sin embargo, de esto último se encarga el motor de ejecución, pues si hay suficiente memoria disponible para realizar el trabajo, no habrá necesidad de guardarlos en el medio de almacenamiento secundario.

Ejemplo 2.2.5.2

```
SELECT      títulos: T
FROM        libros.libro AS L,
            L.autor.nombre AS N,
            L.título AS T
WHERE N = "Buneman"
```

Cuando se ejecuta esta consulta sobre la base de datos de la figura 2.2.5.1, al no efectuarse la *etapa* del constructor de la cláusula SELECT, se ahorra el trabajo de agregar el contenido de T al resultado. Ello no representa ninguna optimización, pues con una evaluación tradicional se obtiene el ahorro. Esto indica que en ninguno de los casos presentados en los ejemplos anteriores, la optimización fue a causa de alguna técnica que pueda llevar a cabo el optimizador de consultas.

Reducir el número de *ciclos* tiene el propósito de eliminar aquéllos en donde se sabe que los datos contenidos en las tablas temporales del FROM no serán parte del resultado; por lo tanto, con los valores actuales del ejemplo la condición en la cláusula WHERE será falsa y, en consecuencia, el constructor en la cláusula SELECT no se ejecutará. En el ejemplo 2.2.5.1, sólo se podría reducir el número de *ciclos* al descartar aquellos libros de los que se tenga conocimiento que no cumplirán con la condición especificada en el WHERE; tal es el caso del libro de nombre "Introducción a los Sistemas de Bases de Datos". Eliminar un *ciclo* requiere hacerlo con un buen análisis, considerando que no se tome en cuenta algún dato que realmente no forme parte del resultado, pues de lo contrario no habría optimización alguna.

Eliminar un *ciclo* en una consulta como la del ejemplo 2.2.3.1, genera un mayor ahorro al no realizar la evaluación de las expresiones de camino definidas en la cláusula FROM de la consulta interna.

Evaluar una expresión de camino significa para el SABD recorrer el grafo de datos semiestructurados en busca de aquellos identificadores que estén localizados en el camino definido por la expresión. La forma simple de ejecutar una expresión de camino, requiere un acceso a disco cada vez que se profundiza en el recorrido de alguna rama del grafo de datos. Por lo general, habrá varias ramas dentro de un mismo grafo, lo que implica que la evaluación total de una expresión de camino puede requerir un número elevado de accesos a disco.

Ejemplo 2.2.5.3

libros.libro.autor.nombre

Al evaluar la expresión de camino anterior sobre la base de datos de la figura 2.2.5.1, el recorrido de forma simple se realiza de la siguiente manera:

1) Recuperar el identificador del dato semiestructurado raíz de la Ssd-tabla con nombre "libros".
2) Recuperar la colección de identificadores de los hijos del punto anterior alcanzados bajo la arista con etiqueta "libro".
3) Para cada elemento de la colección del punto anterior:
    a. Recuperar la colección de identificadores de los hijos alcanzados bajo la arista con etiqueta "autor".
    b. Para cada elemento de la colección del punto anterior:
        i. Recuperar la colección de identificadores de los hijos alcanzados bajo la arista con etiqueta "nombre".
        ii. Para cada elemento del conjunto del punto anterior:
            1. Guardar el identificador.

Para ejecutar esta expresión de camino se necesitaron tres iteradores y cada una de ellos representa la división de ramas en el árbol de datos. Por cada iteración se requiere un acceso a disco para recuperar la colección de hijos que cumplan con el camino especificado en la expresión. Utilizando los identificadores de la base de datos de la figura 2.2.5.1, se tiene la siguiente secuencia de accesos a disco:

I. *Paso 1*: Recuperar raíz: "l"
II. *Paso 2*: Recuperar la colección de identificadores de los hijos de "l" alcanzados bajo la arista "libro": {"l1", "l2"}
III. *Paso 3.a*: Recuperar la colección de identificadores de los hijos de "l1" alcanzados bajo la arista "autor": {"a1", "a2", "a3"}
IV. *Paso 3.a.i*: Recuperar la colección de identificadores de los hijos de "a1" alcanzados bajo la arista "nombre": {"a11"}
V. *Paso 3.a.i*: Recuperar la colección de identificadores de los hijos de "a2" alcanzados bajo la arista "nombre": {"a21"}
VI. *Paso 3.a.i:* Recuperar la colección de identificadores de los hijos de "a3" alcanzados bajo la arista "nombre": {"a31"}
VII. *Paso 3.a*: Recuperar la colección de identificadores de los hijos de "l2" alcanzados bajo la arista "autor": {"a1", "a3"}
VIII. *Paso 3.a.i:* Recuperar la colección de identificadores de los hijos de "a1" alcanzados bajo la arista "nombre": {"a11"}
IX. *Paso 3.a.i:* Recuperar la colección de identificadores de los hijos de "a3" alcanzados bajo la arista "nombre": {"a31"}

El total de accesos necesarios por la expresión de camino *libros.libro.autor.nombre* es nueve. Para una base de datos tan pequeña como la de la figura 2.2.5.1, este número es muy elevado, considerando que el total de aristas contenidas en la Ssd-tabla "libros" son 19; es decir, se trata de casi la mitad de un recorrido completo del árbol de datos para encontrar los identificadores requeridos, que son sólo tres.

Para una base de datos de tamaño considerable, como la del ejemplo 2.2.5.4, evaluar una expresión de camino puede llegar a ser altamente costoso para el SABD, si ella se realiza de forma simple, esto es, sin aplicar alguna técnica de optimización.

Ejemplo 2.2.5.4 Sobre una base de datos como la descrita en la figura 2.2.5.2, determinar el número de accesos a disco necesarios para evaluar la expresión de camino:

Sonylandia.AparatoElectrónico.componente.CircuitoIntegrado.pin.descripción

En este caso se trata de una base de datos con un número de datos semiestructurados más realista. Está compuesta por una Ssd-tabla llamada

"Sonylandia", en la que hay almacenada información de 1,000 aparatos electrónicos, cada uno conformado, en promedio, por 15 componentes, éstos por 200 circuitos integrados, los que a su vez tienen 60 pines. El punto clave en esta evaluación de la expresión de camino es que no todos los pines cuentan con su descripción, pues en total sólo existen 10,000 registradas en la base de datos; por ello, varias de las ramificaciones no llegarán al último paso de la expresión de camino.

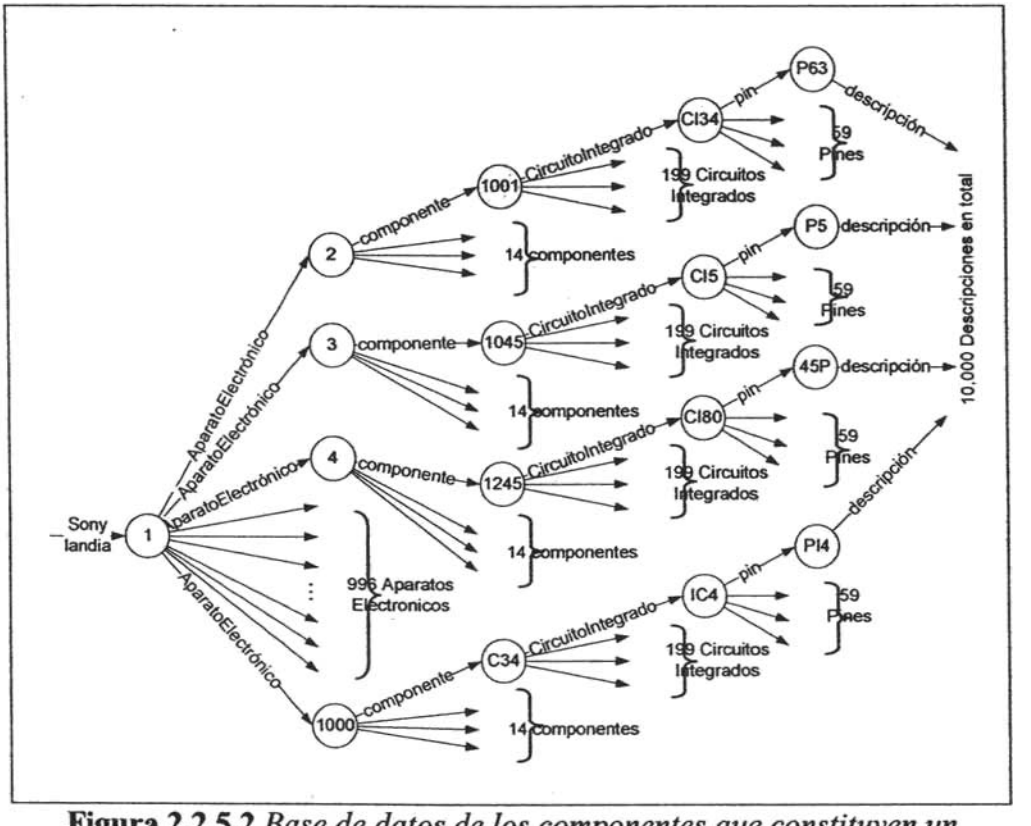


Figura 2.2.5.2 *Base de datos de los componentes que constituyen un aparato electrónico*

La evaluación de forma simple para esta expresión de camino es la siguiente:

1. Recuperar el identificador del dato semiestructurado raíz de la Ssd-tabla con nombre "Sonylandia".
2. Recuperar la colección de identificadores de los hijos del punto anterior alcanzados bajo la arista con etiqueta "AparatoElectrónico".
3. Para cada elemento de la colección del punto anterior:
   3.1. Recuperar la colección de identificadores de los hijos alcanzados bajo la arista con etiqueta "componente".
   3.2. Para cada elemento de la colección del punto anterior:
      3.2.1. Recuperar la colección de identificadores de los hijos alcanzados bajo la arista con etiqueta "CircuitoIntegrado".
      3.2.2. Para cada elemento de la colección del punto anterior:
         3.2.2.1.Recuperar la colección de identificadores de los hijos alcanzados bajo la arista con etiqueta "pin".
         3.2.2.2.Para cada elemento de la colección del punto anterior:
            3.2.2.2.1. Recuperar la colección de identificadores de los hijos alcanzados bajo la arista con etiqueta "descripción".
            3.2.2.2.2. Para cada elemento de la colección del punto anterior:
               3.2.2.2.2.1.Guardar el identificador.

Por cada dato semiestructurado "AparatoElectrónico" que se recupera, se obtienen 15 componentes en promedio, después por cada componente se recuperan 200 circuitos integrados y así sucesivamente. Por cada iteración es necesario un acceso a disco para recuperar la colección de identificadores de datos semiestructurados de los hijos del nodo actual y así continuar al siguiente nivel de la ramificación que se evalúa. El número de accesos a disco requeridos para evaluar esta expresión de camino es:

- 1 acceso para recuperar el identificador de la Ssd-tabla "Sonylandia".
- 1 acceso para recuperar la colección de identificadores de los aparatos electrónicos.
  - 1 acceso por cada aparato electrónico (1,000) para recuperar sus componentes.

- 1 acceso por cada componente (15) contenido en un aparato electrónico (10,000) para recuperar sus circuitos integrados.
  - 1 acceso por cada circuito integrado (200) contenido en cada componente (15) de cada aparato electrónico (10,000) para recuperar sus pines.
    - ❖ 1 acceso por cada pin (60) de cada circuito integrado (200) contenido en cada componente (15) de cada aparato electrónico (10,000) para recuperar las descripciones. Aunque no todos los pines tengan descripción, es necesario el acceso a disco para comprobar que realmente carece de ello.

Total de accesos a disco: 1 + 1 + 1,000 + 1000*15 + 1000*15*200 + 1000*15*200*60

≈ 183 millones de accesos.

Con lo anterior, puede observarse que el mayor número de accesos a disco proviene de la evaluación de expresiones de camino y que de éstas depende en gran parte la evaluación de consultas; es decir, el número de *ciclos* y de su resultado el número de *etapas*. Por esto último, la optimización que se trata en esta tesis se enfoca principalmente a dichas expresiones.

El capítulo 4 describe las técnicas de optimización desarrolladas y utilizadas en el procesador de consultas.

2.2.6 Generador del plan físico de consulta

La última transformación que sufre la consulta es el paso del plan lógico al plan físico, esto es, convertir el árbol de expresión a nivel lógico a una secuencia de instrucciones primitivas. Una instrucción primitiva, a diferencia de un operador lógico, representa una acción básica dentro del SABD, como por ejemplo, recuperar un dato (un nodo del árbol de datos),

borrar un dato, recuperar la información de un dato primitivo, recuperar los hijos de un nodo, etc.

En esta etapa, cada operador lógico se transforma en una o varias instrucciones primitivas que realizan la operación indicada. Además, se determina el orden de ejecución de las operaciones lógicas. La razón de esto radica en que en una consulta, el número de *ciclos* lo determinan las expresiones de camino declaradas en la cláusula FROM, por lo que evaluar dichas expresiones es la primera tarea del plan físico. Junto con las expresiones de camino, se construyen las iteraciones necesarias para recorrer el árbol de datos en busca de la información requerida al evaluar la consulta.

El número de iteradores no siempre será el mismo para un número determinado de expresiones de camino declaradas en la cláusula FROM, esto depende de qué técnicas de optimización se aplicaron en el subcomponente optimizador de consulta. Sin embargo, estas optimizaciones deben reflejarse dentro del árbol de expresión a nivel lógico. De esta forma, al realizar la traducción de lógico a físico existirá la suficiente información para crear un plan físico correcto y óptimo, es decir, que realmente realice la tarea descrita por la consulta y que aproveche las técnicas aplicadas por el optimizador de consultas.

La salida de este subcomponente, el plan físico de consulta, se pone en marcha en el motor de ejecución, el cual administra la memoria y el ambiente de ejecución de la consulta.

La descripción de la transformación que realiza este subcomponente, así como la definición de las instrucciones primitivas del almacenamiento físico, son temas que trata el capítulo 5.

2.3 Resumen y discusión

El esquema del sistema administrador de bases de datos semiestructurados presentado en este capítulo es muy genérico. Sólo obedece al propósito de dar una idea del entorno en que trabaja el procesador de consultas. En la literatura sobre bases de datos [GaUW00], pueden encontrarse otros diagramas con más detalle de los componentes que integran a un sistema administrador de bases de datos así como sus relaciones. Aún si estos

diagramas son para sistemas administradores de bases de datos relacionales, el objetivo de cada componente no cambia a través de los distintos sistemas administradores, lo que cambia es el modelo que ellos utilizan. Esto último, en consecuencia, modifica la forma de trabajar de cada componente al adaptarse al modelo que maneje.

La adaptación de los componentes del sistema administrador (excepto el procesador de consultas y el motor de ejecución) al uso del modelo de datos semiestructurados es tema de otra tesis que se desarrolla paralelamente. Lo que trata en esta disertación es la descripción del trabajo que realiza el procesador de consultas al transformar una consulta a un plan físico cuando se manejan datos semiestructurados.

El procesador de consultas es el componente del sistema administrador de bases de datos de convertir una consulta a instrucciones del almacenamiento primitivo. Este componente es uno de los esenciales dentro del sistema administrador de bases de datos, pues forma parte del camino de ejecución de una consulta (vea figura 2.1.1). Como su función principal es la de traducir un consulta, se revisó documentación acerca de compiladores [AhSU90, Loud04], pues es un tema relacionado. En ella, el enfoque está dirigido al manejo de lenguajes de programación; sin embargo, fue de utilidad para iniciar la investigación.

Después de consultar estas fuentes de información, surgieron dos preguntas: ¿Por qué no traducir directamente la consulta a plan físico? ¿Es necesario contemplar un código intermedio para el caso del procesador de consultas? En documentación especializada en sistemas de bases de datos [GaUW00, SiKS02, MAG+97] mencionan cómo se lleva acabo la traducción dentro del procesador de consultas. Después de analizar lo descrito en esta documentación, se llegó a la conclusión que sí es necesario un código intermedio, llamado plan lógico en el ambiente de bases de datos. Este plan se utiliza para realizar optimizaciones a un nivel que no esté apegado al almacenamiento primitivo, es decir, que puede aplicarse en cualquier sistema administrador que maneje el mismo diseño de plan lógico. En general, todas las operaciones realizadas dentro del procesador de consulta se conciben para mejorar el rendimiento del sistema administrador.

Por lo anterior, la principal diferencia entre un compilador y el procesador de consultas está en que uno maneja lenguajes de programación y el otro el modelo de datos semiestructurados. Por ello, las

etapas que siguen después del analizador semántico se enfocan al manejo de dicho modelo, principalmente para optimizar la ejecución de la consulta.

La función del análisis semántico se limita a validar el alcance de las tablas temporales y el manejo de vistas abstractas. Como el analizar léxico y sintáctico son de carácter general, pues su función es la misma que la que se encuentra en cualquier compilador, se describen en el apéndice A.

En el siguiente capítulo se describe el plan lógico de consulta, el cual es el primer paso dentro del procesador de consultas que se enfoca directamente al uso del modelo de datos semiestructurados.

Capítulo 3

El plan lógico de consulta

La primera acción dentro del procesador sobre una consulta es transformarla de una cadena de caracteres a un plan lógico. En él que se describen aquellas acciones necesarias para ejecutarla y se utiliza para detectar alguna oportunidad de optimización. El plan lógico de consulta consiste en un anidamiento de operadores lógicos que representan tareas más simples dentro de la consulta. Algunos de éstos se basan en los que utiliza el álgebra relacional, debido a la semejanza de sintaxis que existe entre Ssquirel [Garc02] y el SQL. Los demás provienen de la misma definición de Ssquirel pero con una variante en algunos de ellos para poder unirlos al plan lógico.

Los operadores lógicos deben ser capaces de conectarse entre sí para poder expresar consultas más complejas. De este modo, la mayoría de ellos regresan una colección de datos semiestructurados o primitivos definida como:

$$\{d_1, d_2, d_3, \ldots, d_n\}$$

donde cada d_x con $x=1..n$, es el identificador de un nodo (dato semiestructurado o primitivo). Esta colección, como se mencionó en el capítulo 1, puede tener elementos duplicados.

Es necesario también que los operadores trabajen con el mismo tipo de colecciones que éstos regresan, lo que hace posible conectarlos entre ellos (anidamiento de operadores) y así poder construir planes lógicos de consulta de tamaño indefinido.

A partir de este momento, el concepto de nodo e identificador tendrán un significado equivalente. Este capítulo detalla los operadores lógicos que utiliza este plan. La definición de los siguientes operadores busca mejorar y eliminar algunos de los problemas que se encuentran en otras álgebras desarrolladas para este tipo de datos. También, se pretende concebir una muy apegada a la relacional, con la que sea posible adaptar

las técnicas de optimización a nivel lógico ampliamente estudiadas en las bases de datos relacionales.

En el álgebra utilizada por el proyecto Lore [MAG+97], sus operadores se apegan a sus funciones físicas, lo que hace difícil ocuparlos para otro sistema que no utilice su mismo almacenamiento primitivo. Además, su optimización se basa en costos, lo que propició no considerar mejoras a nivel lógico realizando equivalencias entre operadores como lo efectuado en el álgebra relacional.

Un álgebra definida para datos semiestructurados y XML [BeTz99], sólo considera el tipo de consultas de recuperación de datos, sin tomar en cuenta las que modifican el estado de la base de datos. En una publicación más reciente [FrHP02] sobre un álgebra para XML, se observa un trabajo más elaborado y completo, pero aún así no contempla las consultas de modificación de datos. Estos dos últimos trabajos también tienen algunas deficiencias en la definición de sus operadores, básicamente por no implementar sus álgebras en un programa de *software*, algo que sí se realizó con Lore [MAG+97] pero su álgebra no es aplicable a un sistema diferente al suyo.

El procesador de consultas desarrollado con este trabajo implementó el álgebra que se presenta a continuación, la cuál también abarca consultas de actualización a datos semiestructurados.

3.1 Operadores lógicos elementales

Los operadores lógicos tratados en esta sección tienen su origen en la descripción del lenguaje Ssquirel, por lo que sólo se definen brevemente.

3.1.1 Operador unión de datos (∪)

Este operador puede aplicarse a nodos o a colecciones que sólo contienen un nodo. Su definición es la siguiente:

Sintaxis:

$d_1 \cup d_2$

Definición:

Dados los nodos d_1 y d_2, que tienen una colección de hijos definidos como $\{l_{11}:h_{11}, l_{12}:h_{12}, \ldots, l_{1n}:h_{1n}\}$ y $\{l_{21}:h_{21}, l_{22}:h_{22}, \ldots, l_{2m}:h_{2m}\}$ respectivamente, donde cada l:h representa una arista con etiqueta l apuntando al nodo con identificador h, se crea un nuevo nodo que contendrá la colección de hijos definido como $\{ l_{11}:h_{11}, l_{12}:h_{12}, \ldots, l_{1n}:h_{1n}, l_{21}:h_{21}, l_{22}:h_{22}, \ldots, l_{2m}:h_{2m} \}$. En la figura 3.1.1.1 se encuentra una representación gráfica de la definición de la unión.

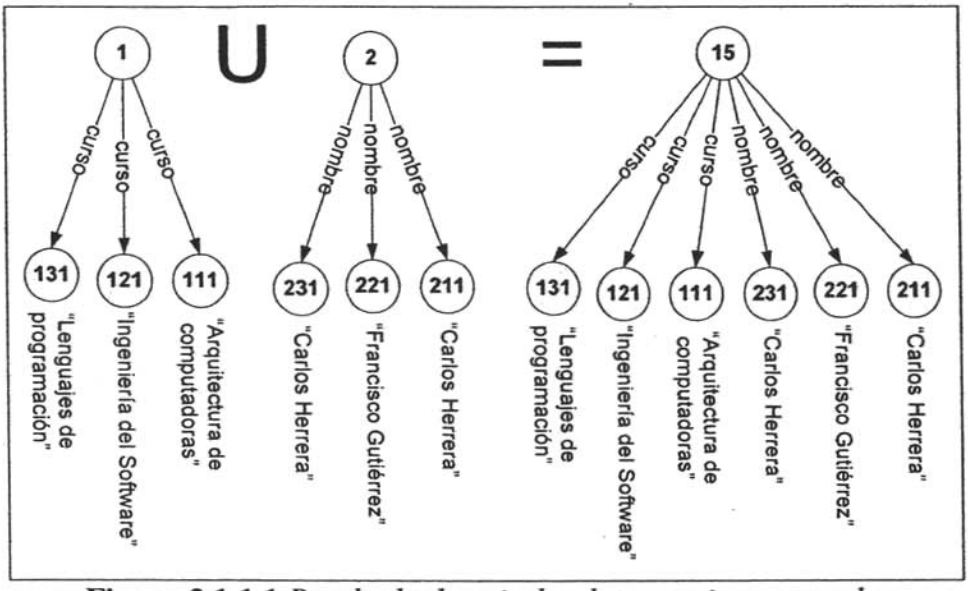


Figura 3.1.1.1 *Resultado de unir dos datos semiestructurados*

3.1.1.1 Operador NewSSD

A partir del operador de unión ($\cup$) se define el operador NewSSD el cual une varios nodos sin necesidad de definir varias uniones.

Sintaxis:

NewSSD(D_1, D_2,D_3,...,D_n)

Descripción:

A partir de *n* colecciones de nodos definidas como:

$$D_1 = \{d_{11}, d1_2, \ldots, d_{1a}\}$$
$$D_2 = \{d_{21}, d_{22}, \ldots, d_{2b}\}$$
$$D_3 = \{d_{31}, d_{32}, \ldots, d_{3c}\}$$
$$\ldots$$
$$Dn = \{d_{n1}, d_{n2}, \ldots, d_{nx}\}$$

regresa el identificador de un nuevo nodo *r* definido como:

$$r = d_{11} \cup d_{12} \cup \ldots \cup d_{1a} \cup d_{21} \cup d_{22} \cup \ldots \cup d_{2b} \cup d_{31} \cup d_{32} \cup \ldots \cup d_{3c} \cup \ldots \cup d_{n1} \cup d_{n2} \cup \ldots \cup d_{nx}$$

3.1.2 Operador unión de colecciones ($\cup_c$)

Este operador, a diferencia del relativo a la unión de datos, agrupa todos los nodos de dos colecciones en una sola, semejante a la unión de conjuntos. Sin embargo, debido a que es posible encontrar elementos duplicados y que existe un orden de los mismos dentro de las colecciones, la definición toma en cuenta estas características:

Sintaxis:

$D_1 \cup_c D_2$

Definición:

Partiendo de dos colecciones D_1, D_2 definidas como $\{d_{11}, d_{12}, \ldots, d_{1n}\}$ y $\{d_{21}, d_{22}, \ldots, d_{2m}\}$ respectivamente, la unión de ellas es la colección *R*:

$$R = \{ d_{11}, d_{12}, \ldots, d_{1n}, d_{21}, d_{22}, \ldots, d_{2m} \}$$

Con esta definición, la unión mantiene el orden de los elementos contenidos en cada colección y, además, preserva los elementos duplicados. Por ejemplo, si un identificador *Z* se repite *r* veces dentro de D_1 y s dentro de D_2, la colección resultante tendrá $r+s$ repeticiones del identificador *Z*.

3.1.3 Operador diferencia de colecciones ($-_c$)

Este operador puede aplicarse a colecciones de datos semiestructurados o primitivos. Su definición es la siguiente:

Sintaxis:

$D_1 -_c D_2$

Definición:

Partiendo de dos colecciones D_1, D_2 definidas como $\{d_{11}, d_{12}, ..., d_{1n}\}$ y $\{d_{21}, d_{22}, ..., d_{2m}\}$ respectivamente, se agrupan los elementos en cada una formando parejas (<nodo>,<repeticiones>) y cambiando ambas colecciones a conjuntos de la forma:

$$C_1 = \{(d_{1a},a_1), (d_{1b},b_1), ..., (d_{1x},x_1)\}$$
$$C_2 = \{(d_{2a},a_2), (d_{2b},b_2), ..., (d_{2x},x_2)\}$$

la diferencia de estos conjuntos se define como:

$$R = \{(d, n) \mid (d, x) \in C_1 \wedge n = exist((d,x), C_2) \}$$

Donde la función exist se define como:

$$exist((d,x), C) = \begin{cases} x & si\ (d,y) \notin C\ con\ alguna\ y \\ max(0, x-y) & si\ (d,y) \in C\ para\ alguna\ y \end{cases}$$

La función *max* devuelve el mayor de dos números. El resultado de $D_1 -_c D_2$ es la colección R_c que es la representación del conjunto R, donde por cada pareja $(d,n) \in R$ el nodo d aparece n veces en R_c.

Para este operador, aunque los duplicados se toman en cuenta, el orden de los elementos se pierde durante la acción.

Ejemplo 3.1.3.1 El resultado de evaluar la expresión $A -_c B$, donde A y B son dos colecciones definidas como:

$$A = \{1,2,2,3,3,3\}$$
$$B = \{2,3,3,4,4\}$$

se determina de la siguiente forma:

- Se construyen los conjuntos C_1 y C_2 que representan las colecciones A y B:

$$C_1 = \{(1,1), (2,2), (3,3)\}$$
$$C_2 = \{(2,1), (3,2), (4,2)\}$$

- Por cada elemento en C_1 se determina su número de repeticiones final mediante la función *exist*:

(1,1): d=1, x=2 → exist((1,1), {(2,1), (3,2), (4,2)})

Como no existe ningún *(1,y)* el número de repeticiones de "1" será igual a la del conjunto original, el cual es 1.

(2,2): d=2, x=2 → exist((2,2), {(2,1), (3,2), (4,2)})

El número de existencias del elemento "2" se determina al restar *x-y*, donde *y=1* debido al elemento *(2,1)* en C_2. "2" tendrá 1 repetición en el conjunto final.

(3,3): d=3, x=3 → exist((3,3), {(2,1), (3,2), (4,2)})

En esta ocasión *(3,2)* existe en C_2, por lo que el número de repeticiones del elemento "3" será 3-2.

$$\therefore R = \{(1,1), (2,1), (3,1)\}$$

El resultado de restar la colección *B* a *A* es:

$$R_c = \{1, 2, 3\}$$

3.1.4 Funciones de agregación

El lenguaje Ssquirel define cinco funciones de agregación (*AVG, MAX, MIN, SUM y COUNT*) y se incorporan al plan lógico debido principalmente a que construyen nuevos datos dentro de la base de datos.

Sintaxis:

Fun(D)

Definición:

Fun representa cualquiera de las cinco funciones mencionadas. Al recibir una colección de nodos $D = \{d_1, d_2, ..., d_n\}$, para cada d_i con $i=1..n$, se crea un nuevo nodo primitivo p, el cual contendrá el promedio, número máximo, mínimo o suma de los hijos de d_i, según la función de que se trate. Devuelve una colección R conteniendo a los nodos p creados.

$$R = \{p \mid p=Fun(d) \wedge d \in D\}$$

3.1.5 Operador clon

Este operador aplica tanto a datos primitivos como semiestructurados. Para los primeros copia su valor, para los segundos realiza una copia de la estructura de todos los datos alcanzados desde aquel que está en clonación.

Sintaxis:

Clon(x)

Definición:

Regresa un nuevo dato c, tal que si x es primitivo con valor v, c será un primitivo con valor v y si x no es primitivo, c será un semiestructurado que clone todos los nodos alcanzados por x.

$$Clon\ (x) = \begin{cases} CreatePrimitive(v) & \text{si } x \text{ es primitivo} \\ c \in N \wedge \forall l \in L \wedge \forall s \in x.l\ (c.l = c.l \cup_c \{Clon(s)\}) & \text{si } x \text{ no es primitivo} \end{cases}$$

Con $N = \Sigma - \beta$, donde Σ representa el conjunto de todos los posibles identificadores dentro de la base de datos, β es el conjunto de todos los identificadores en uso dentro de la base de datos y L es el conjunto de todas las etiquetas definidas en la base de datos. *CreatePrimitive* se describe en la sección 3.3

3.1.6 Operadores aritméticos con datos primitivos

Las operaciones presentadas a continuación, sólo son válidas para datos primitivos. Ca-da una devuelve un nuevo dato primitivo que contiene el valor del resultado de dicha operación.

Sintaxis:	*Definición:*
$d_1 + d_2$	Regresa un nuevo dato primitivo que contiene el resultado de sumar el dato primitivo d_1 más d_2.
$d_1 - d_2$	Regresa un nuevo dato primitivo que contiene el resultado de restar el dato primitivo d_2 al d_1.
$d_1 * d_2$	Regresa un nuevo dato primitivo que contiene el resultado de multiplicar el dato primitivo d_1 por d_2.
d_1 / d_2	Regresa un nuevo dato primitivo que contiene el resultado de dividir el dato primitivo d_1 entre d_2.
d_1 MOD d_2	Regresa un nuevo dato primitivo que contiene el residuo de la división de d_1 entre d_2.

3.2 Operador Project (π)

Este operador permite crear las tablas temporales definidas dentro de la cláusula FROM de una consulta. Como medio para especificar cuáles son los datos que contendrá la tabla temporal, el operador maneja una expresión de camino, que regresa una colección de identificadores. Por lo anterior, no se define un operador lógico para una expresión de camino, puesto que al encontrar una expresión de este tipo, ésta siempre va a regresar una colección de identificadores. La definición del operador *Project* es la siguiente:

Sintaxis:

$\pi(S, l)$

Definición:

Partiendo de una colección de datos semiestructurados $S = \{d_1, d_2, d_3, ..., d_n\}$, para cada d_i con $i=1..n$, se crea un nodo x_i con identificador no utilizado por otro nodo y se crea una arista A_i con origen en x_i y etiqueta l apuntando a d_i. El resultado de esta función es una colección R de datos semiestructurados que contiene los identificadores de cada x_i creado.

$$R = \{ x \mid d \in S \wedge x \in N \wedge x.l = d \}$$

Con $N = \sum - \beta$, donde $\sum$ representa el conjunto de todos los posibles identificadores dentro de la base de datos y β es el conjunto de todos los identificadores ya utilizados dentro de la base de datos.

Ejemplo 3.2.1 La conversión de una cláusula FROM definida como:

```
FROM profesores.profesor.nombre as X
```

a plan lógico es:

π *(profesores.profesor.nombre, X)*

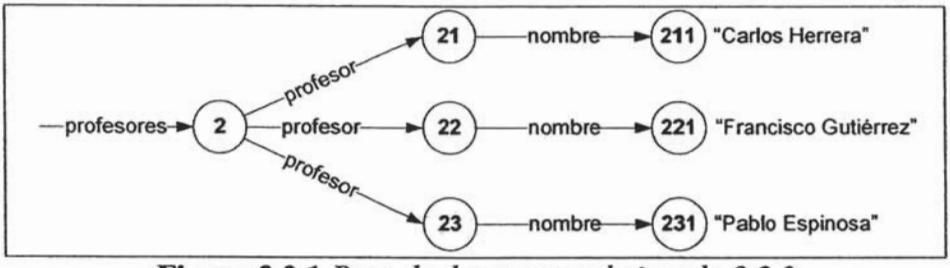


Figura 3.2.1 *Base de datos para el ejemplo 3.2.2*

Ejemplo 3.2.2 Considerando la base de datos de la figura 3.2.1, el plan lógico

π *(profesores.profesor.nombre, X)*

enlaza una nueva arista con etiqueta "*X*" a los nodos de la base de datos con identificadores 211, 221 y 232, dejando momentáneamente a la base de datos en el estado que muestra la figura 3.2.2 mientras se ejecuta la consulta.

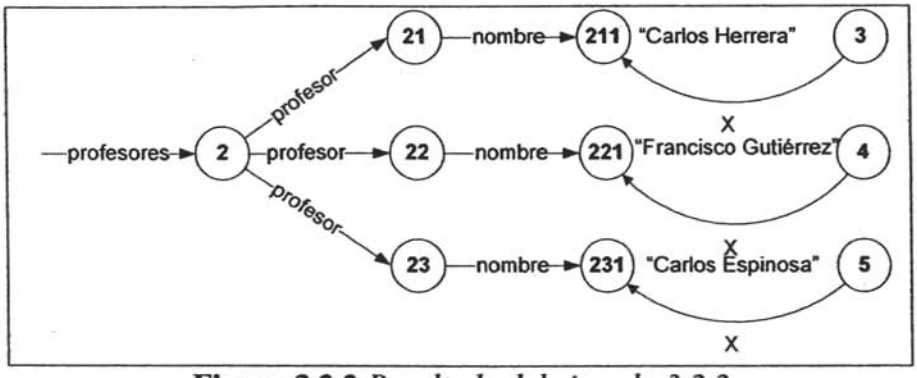
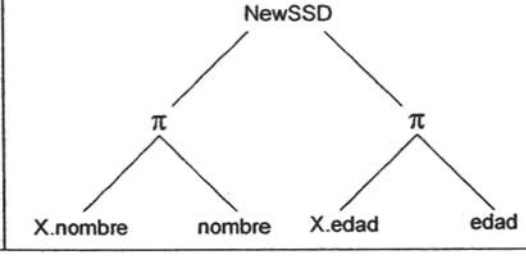


Figura 3.2.2 *Resultado del ejemplo 3.2.2*

Este operador será uno de los más utilizados dentro del plan lógico, pues con él se efectúa gran parte de los enlaces necesarios en la consulta para recuperar, borrar y/o modificar datos. También se emplea para representar la operación realizada por la función *Pick* encontrada en el lenguaje Ssquirel, pues su traducción a plan lógico es la siguiente:

Expresión en lenguaje Ssquirel	***Plan lógico***
`X Pick (nombre, edad)`	

3.2.1 Operador de proyección Trim

Este operador elimina algunos de los hijos de un dato. Fue necesaria su definición debido a que la elección de los hijos de interés (los que van a proyectarse) es diferente para cada uno de los datos, algo que se determina a tiempo de ejecución.

Sintaxis:

Trim(D, L)

Definición:

Dado una colección $D = \{d_1,d_2,...,d_k\}$ y un conjunto $L = \{l_1,l_2,..,l_n\}$, para cada d_i con $i=1..k$ determinar el conjunto $L_t = L_i\text{-}L$ donde L_i es el conjunto de todas las etiquetas de las aristas que salen de d_i. De esta forma, con $L_t = \{l_{t1}, l_{t2}, .., l_{tm}\}$, con $m \leq n$ y el nodo d_i, crear el nodo:

$$r_i = NewSSD(\pi(d_i.l_{t1}, l_{t1}), \pi(d_i.l_{t2}, l_{t2}), ..., \pi(d_i.l_{tm}, l_{tm}))$$

Regresa la colección de nodos $\{r_1,r_2,..,r_k\}$.

3.3 Operador CreatePrimitive

Construye un nuevo nodo primitivo en la base de datos, el cual tiene el valor especificado como parámetro de entrada del operador.

Sintaxis:

CreatePrimitive(X)

Definición:

Con un valor primitivo *X*, crea un nodo con identificador aún no usado en la base de datos y le asigna el valor *X*. Cuando a un nodo primitivo se le agregan aristas de salida (es decir, hijos) dejará de ser un nodo primitivo y perderá el valor *X*.

3.4 Operador producto cruz (x)

Este operador es semejante al encontrado en el álgebra relacional [UlWi99, SiKS02]. En el caso de datos semiestructurados, para realizar el producto cruz se utiliza el concepto de unión definido en la sección 3.1.1.

A continuación se expresa su descripción y en qué situación aparece dentro de una consulta.

Sintaxis:

$D_1 \, X \, D_2$

Definición:

Partiendo de dos colecciones D_1, D_2 definidas como $\{d_{11}, d_{12}, ..., d_{1n}\}$ y $\{d_{21}, d_{22}, ..., d_{2m}\}$ respectivamente, se realiza el producto cruz de dichas colecciones dando como resultado la colección $Z = \{(d_{11},d_{21}), (d_{11},d_{22}), ..., (d_{11},d_{2m}), (d_{12},d_{21}),..., (d_{1n},d_{2m})\}$. Por cada pareja (d_1, d_2) de Z, se efectúa la operación $d_1 \cup d_2$. Se devuelve una colección R con $n*m$ nodos construidos por la unión realizada en cada una de las parejas.

$$R = \{ (d_1 \cup d_2) \mid d_1 \in D_1 \wedge d_2 \in D_2 \}$$

Este operador es útil cuando se unen dos π donde sus expresiones de camino inician con Ssd-tablas no temporales, o cuando la colección D_1 ya se especificó y el π que determina a D_2 contiene una expresión de camino que inicia con una Ssd-tabla no temporal. Por lo anterior, este operador se emplea dentro de la parte del plan lógico que representa la cláusula FROM de la consulta. El resultado de este operador, al ser un producto cruz, tiende a ser grande.

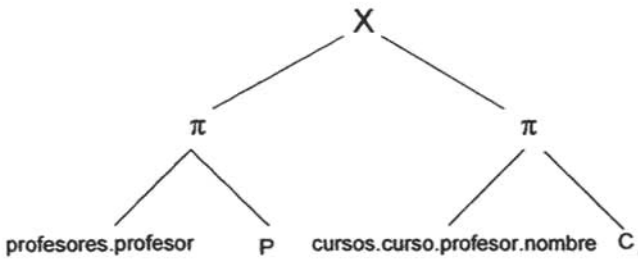
Ejemplo 3.4.1 El plan lógico de la cláusula:

```
FROM profesores.profesor as P,
     cursos.curso.profesor.nombre as C
```

es el siguiente:

$$\pi \, (profesores.profesor, P) \; X \; \pi \, (cursos.curso.profesor.nombre, C)$$

Con el ejemplo anterior, que es pequeño, puede observarse que presentar un plan lógico en forma de anidamiento de funciones dificulta su lectura, por lo que generalmente este plan se presenta en forma de árbol. De este modo el plan lógico anterior queda representado de la siguiente manera:



Una cualidad de los nodos que resultan al realizar un producto cruz es que cada uno de estos nodos lo que representa dentro de la ejecución de la consulta es un *ciclo* de ésta.

3.5 Operador DJoin

Para aquellos casos en que la obtención de una colección de datos semiestructurados depende de otra, se define este operador. Inspirado en el descrito en [ChCM96] con el nombre de *Dependent Join*. A continuación se presenta su descripción y en qué situación aparece dentro de la consulta.

Sintaxis:

DJoin(D_1, E, l)

Definición:

Partiendo de una colección $D_1 = \{d_1, d_2, \ldots, d_n\}$, una expresión de camino $E = l_1.l_2....l_n$ y una etiqueta l se determina la colección:

$$R = \{ (d_1 \cup d_2) \mid d_1 \in D_1 \wedge (d \in d_1.E \wedge d_2 \in N \wedge d_2.l = d) \}$$

Con $N = \Sigma - \beta$, donde Σ representa el conjunto de todo los posibles identificadores dentro de la base de datos y β es el conjunto de todos los identificadores ya utilizados.

Este operador es útil cuando la etiqueta con la que inicia E se define dentro de cada $d_1 \in D_1$, en otro caso, debe usarse el producto cruz. Cuando se emplea el operador *DJoin*, para cada identificador Y de la

primera colección (D_1), se determina la colección de identificadores que son descendientes de éste con algún camino E. Por lo anterior, el resultado, generalmente, tiene un número menor de nodos que en el caso del producto cruz. Cuando para un nodo d de la primera colección D_1, se establece que la colección $d.E$ no tiene ningún elemento, entonces *DJoin* no produce ningún nodo de resultado.

Este operador sirve para enlazar las definiciones de tablas temporales de la cláusula FROM al plan lógico, pero sólo para aquéllas que cumplen la condición descrita anteriormente. El siguiente ejemplo muestra cómo trabaja este operador sobre una base de datos. Se utiliza un número pequeño de identificadores para no crear colecciones que dificulten su lectura.

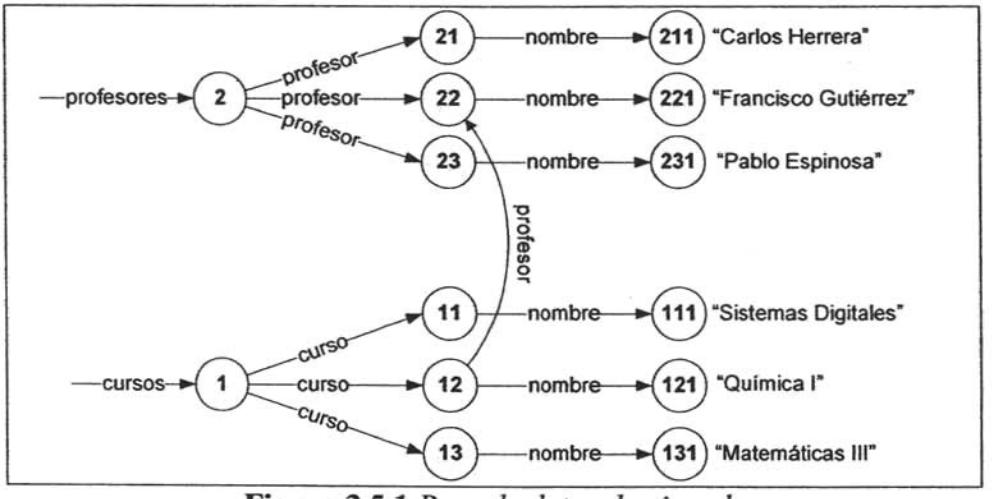
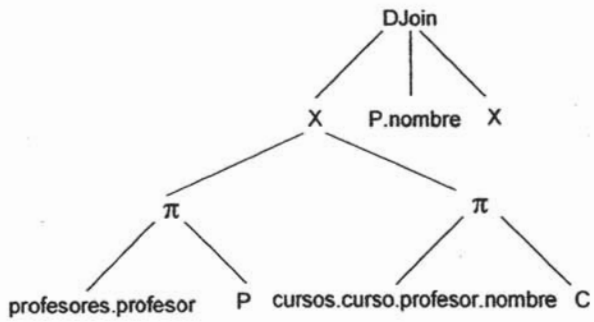


Figura 3.5.1 *Base de datos de ejemplo*

Ejemplo 3.5.1 Con la base de datos de la figura 3.5.1 y la cláusula FROM:

```
FROM profesores.profesor as P,
     cursos.curso.profesor.nombre as C,
     P.nombre as X
```

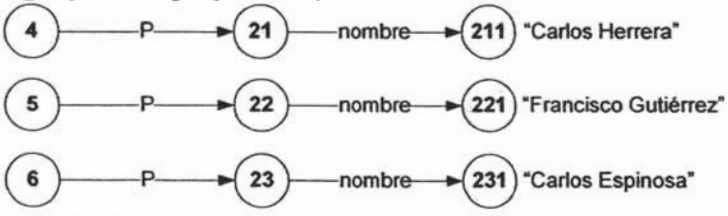
se convierte a plan lógico como:



Se analizan las colecciones creadas por cada operador.
DJoin (X, P.nombre, X)

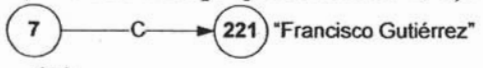
Se procede a determinar la primera colección.

π(profesores.profesor, P) X π(cursos.curso.profesor.nombre, C):
π(profesores.profesor, P):



= {4,5,6}

π(cursos.curso.profesor.nombre, C):

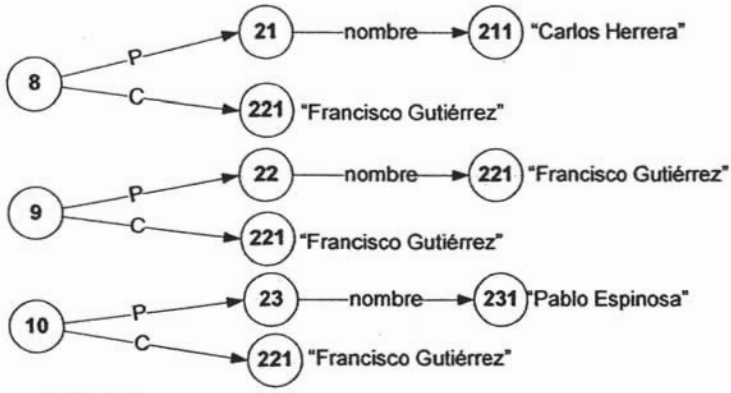


= {7}

{4,5,6} X {7}

Como ambas colecciones no dependen entre sí, se utilizó un producto cruz:

{4,5,6} X {7} = { {4∪7}, {5∪7}, {6∪7}} :



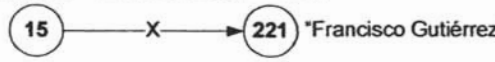
= {8,9,10}

DJoin ({8,9,10}, P.nombre, X):

8: P.nombre = 21.nombre = 211

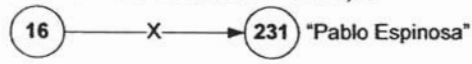
= (8,14)

9: P.nombre = 22.nombre = 221



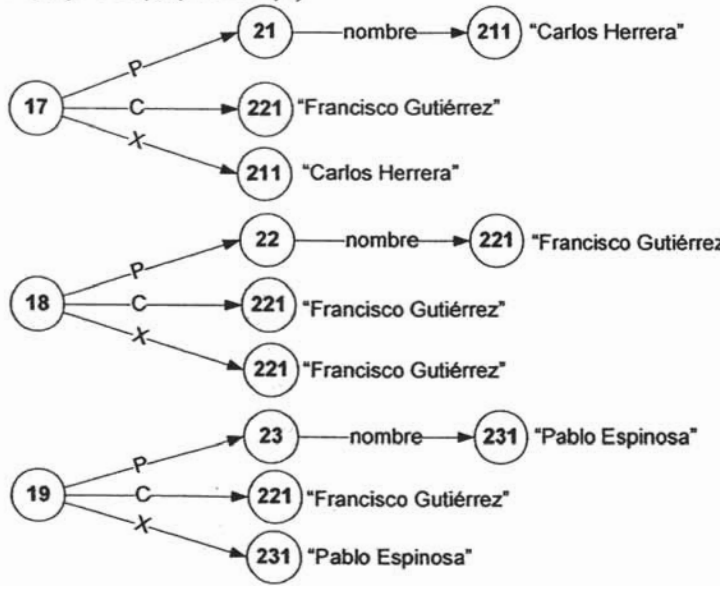
= (9,15)

10: P.nombre = 23.nombre = 231,X



=(10,16)

∴ { (8∪14), (9∪15), (10∪16) } =



dejando la base de datos en el estado que muestra la figura 3.5.2.

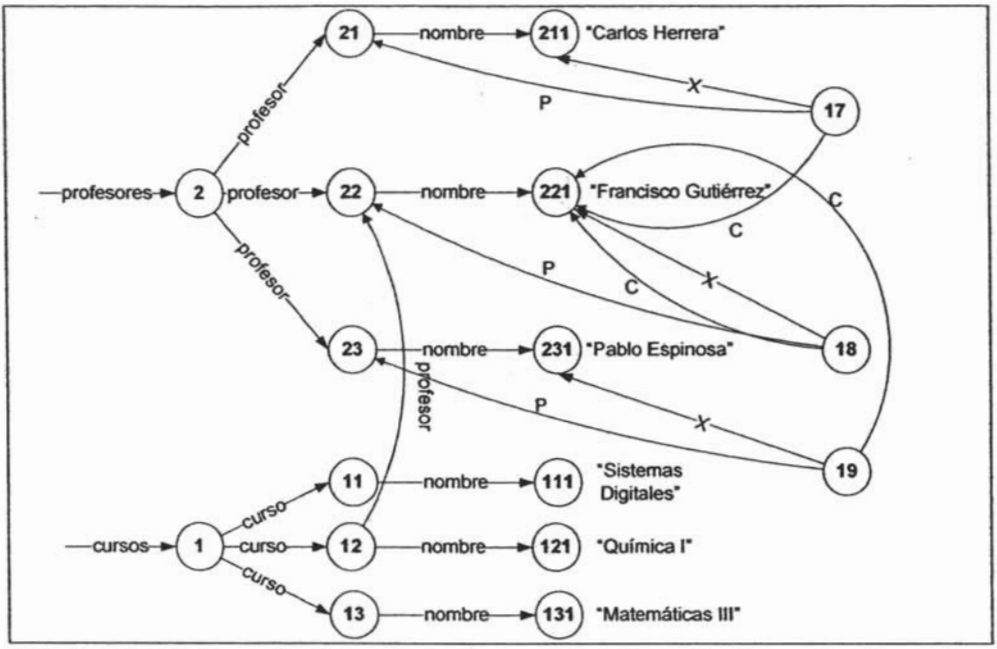


Figura 3.5.2 *Base de datos de la figura 3.5.1 después de se ejecutaron los operadores lógicos del ejemplo 3.5.1*

Una cualidad de los nodos que resultan al realizar un *DJoin*, dígase los nodos 17, 18 y 19 de la figura 3.5.2, es que cada uno de estos nodos lo que representa dentro de la ejecución de la consulta es un *ciclo* de ésta.

3.6 Operador Select (σ)

Este operador proviene de otro del mismo nombre del álgebra relacional y su función es básicamente la misma. Al recibir un conjunto (colección en este caso) de datos, seleccionará aquéllos que cumplan con una condición que se proporciona. Su definición es la siguiente:

Sintaxis:

σ *(D, <condición>)*

Definición:

Dada una colección de datos $D = \{d_1, d_2, ..., d_n\}$, para cada d_i con $i=1..n$, evaluar *<condición>*. El resultado de la función es la colección R de todos los d_i en que la condición es verdadera. La colección R se define como:

$$R = \{d \mid d \in D \wedge condición(d)\}$$

Los operadores condicionales que pueden utilizarse dentro de la condición se especifican en [Garc02] y su conversión a plan lógico queda básicamente como un anidamiento de operadores condicionales. Estos operadores no se definen como operadores lógicos de la consulta, pues no devuelven colecciones, sino un resultado falso o verdadero. Sin embargo, en una condición existe un cambio al pasar a plan lógico, el cual recae en los operandos que representan una consulta anidada. Por ejemplo, en una consulta:

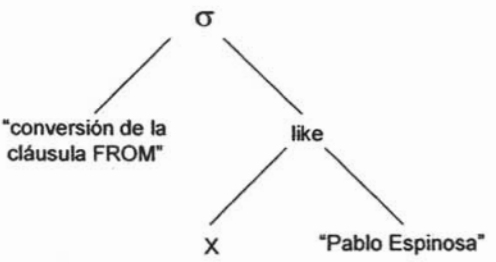
```
SELECT      profesor : P
FROM        profesores.profesor as P
WHERE       2 = COUNT (SELECT curso: C
                       FROM cursos.curso AS C,
                            C.profesor AS M
                       WHERE M IS P)
```

el operando de la función de agregación *COUNT* se transforma en el plan lógico de esa consulta anidada. Hay que recalcar que este tipo de cambio no es exclusivo de los operandos en las condiciones, sino que ocurre donde haya con una consulta anidada.

Al igual que en el álgebra relacional, el operador σ se utiliza para manejar la conversión de la cláusula WHERE a plan lógico. Por lo tanto, la conversión de la cláusula:

```
WHERE X like "Pablo Espinosa"
```

a plan lógico es:



3.7 Operador Map

Este operador, que tiene fundamento en lo que se describe en [FrHP02]. Es útil para aplicar un operador cualquiera a una colección de datos, algo similar a lo que ocurre en la descripción de los operadores lógicos tratados anteriormente, donde se aplica una operación a cada elemento de una colección. Sin embargo, *Map* se define para realizar lo anterior pero con cualquier operación. Su descripción es la siguiente:

Sintaxis:

Map (D, O)

Definición:

Dados una colección de datos semiestructurados o primitivos $D = \{ d_1, d_2, ..., d_n \}$ y un operador lógico O, para cada d_i con $i=1..n$, evaluar O relacionando todo inicio de expresión de camino utilizada dentro del operador O con el nodo d_i y el conjunto de Ssd-tablas contenido en la base de datos; es decir, el alcance dentro del operador O no queda sólo en Ssd-tablas sino lo amplía a lo contenido dentro del nodo d_i. El resultado de este operador es la unión de todas las colecciones devueltas por cada O.

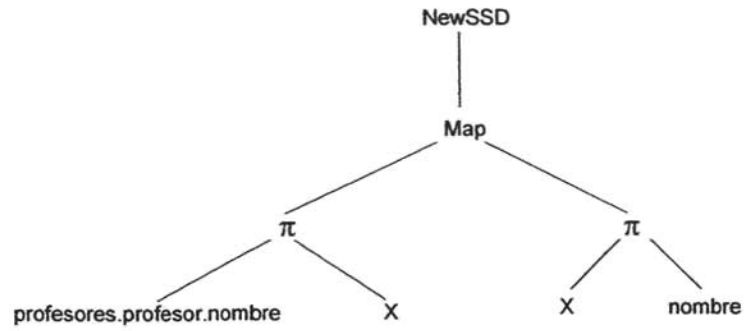
$$R = \{ O\langle d_1 \rangle \cup_c O\langle d_2 \rangle \cup_c ... \cup_c O\langle d_n \rangle \}$$

Con los operadores vistos hasta este punto ya es posible expresar una consulta completa de la forma SELECT-FROM-WHERE con un plan lógico. Los ejemplos 3.7.1 y 3.7.2 muestran las conversiones de este tipo de consultas.

Ejemplo 3.7.1 La representación en plan lógico de la consulta:

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

es la siguiente:



Este plan lógico describe una ejecución de la consulta de la siguiente forma:

- π enlaza a cada identificador que regresa la expresión de camino *profesores.profesor.nombre* con un nodo nuevo temporal *N* mediante una arista con etiqueta "*X*". Luego se regresa la colección de los nodos temporales *N*.

- *Map*: Por cada elemento de la colección devuelta por π

    - Crear una nueva arista con etiqueta "*nombre*" que tiene como origen un nuevo nodo *R* y como destino el nodo obtenido al evaluar la expresión de camino *X*.

    - Regresa la colección de nodos *R* creados en el punto anterior.

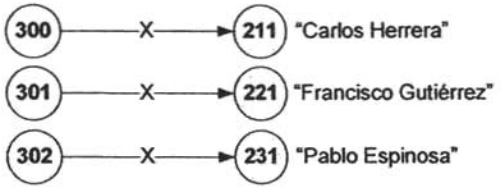
- *NewSSD*: Regresa la unión de cada uno de los nodos *R* contenidos en la colección devuelta por *Map*.

Utilizando la base de datos de la figura 3.5.1, el plan lógico describe el proceso de recuperación de datos como:

1: π*(profesores.profesor.nombre, X)*

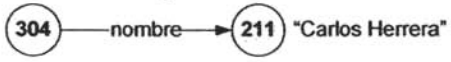
- *profesores.profesor.nombre* = {211, 221, 231}

- Por cada identificador en *profesores.profesor.nombre* crear un nuevo nodo y arista con etiqueta "*X*":

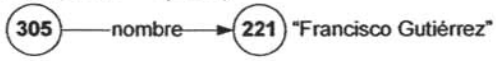


2: *Map({300, 301, 302}, π(X, nombre))*

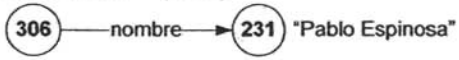
- Por cada identificador de la colección {300, 301, 302}
  - Con 300 se evalúa el operador π, el cual contiene una expresión de camino *X*, donde 300.*X* = {211}. Se crea un nodo nuevo y una arista con etiqueta "*nombre*" que une al nuevo nodo y al nodo 211



  - Con 301, 301.X = {221}

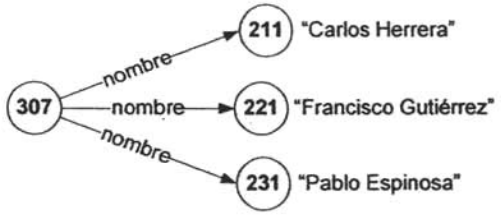


  - Con 302, 302.X = {231}



  - *Map* regresa la colección {304, 305, 306}, donde cada nodo contiene la información recuperada como lo describe la consulta

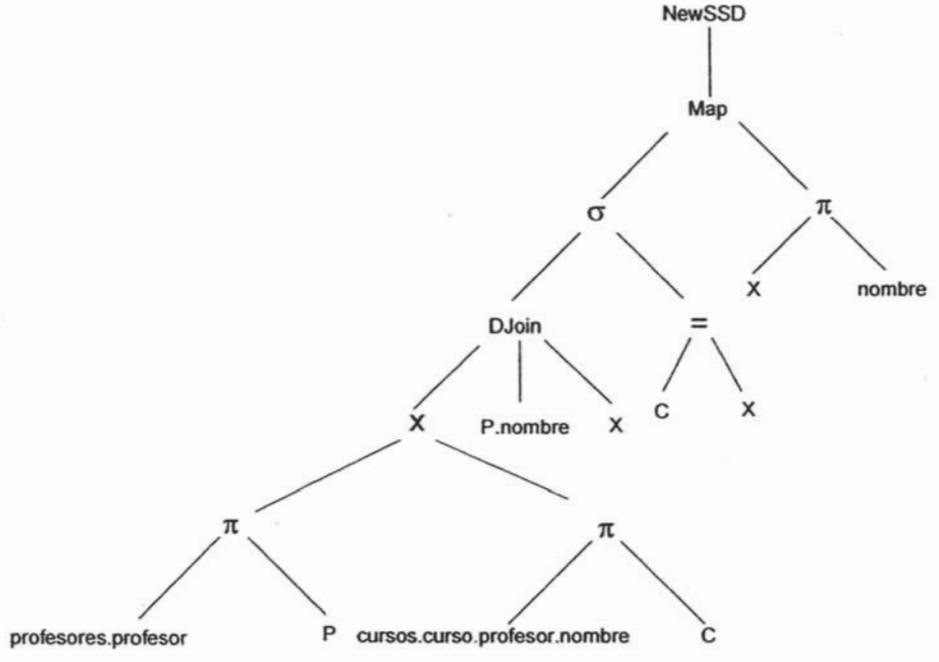
3: *NewSSD* regresa un nuevo nodo 307 resultado de unir los nodos 304, 305 y 306:



Ejemplo 3.7.2 Convertir la siguiente consulta a plan lógico.

```
SELECT      nombre : X
FROM  profesores.profesor as P,
            cursos.curso.profesor.nombre as C,
            P.nombre as X
WHERE C = X
```

Esta consulta regresará aquellos nombres de maestros que estén impartiendo un curso. Es posible obtener este mismo resultado más fácil y directamente con sólo la expresión de camino *cursos.curso.profesor.nombre*, pero en este ejemplo se utiliza esta consulta para así utilizar la cláusula FROM del ejemplo 3.5.1 y de está forma enfocarla a mostrar el trabajo de los otros operadores. El plan lógico para esta consulta es el siguiente:



El proceso de ejecución de este plan adiciona los operadores *σ, DJoin y X* al que se describe en el ejemplo 3.5.1. Utilizando nuevamente la base de datos de la figura 3.5.1, el proceso realizado por los operadores dentro de la cláusula FROM es el mismo que el efectuado en el ejemplo 3.5.1, el cual regresa una colección de identificadores {17,18,19} como los definidos en la figura 3.7.1. Por lo anterior, sólo resta describir el proceso de los operadores *σ, Map y NewSSD* cuando se usa tal base de datos.

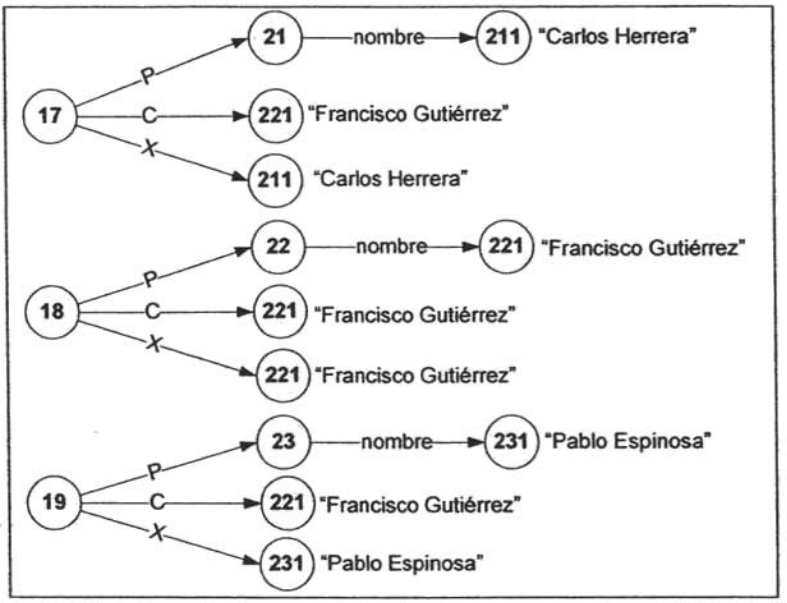
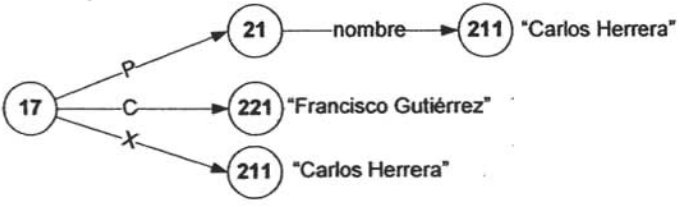


Figura 3.7.1 *Resultado proveniente de los operadores dentro de la cláusula FROM del ejemplo 3.5.1*

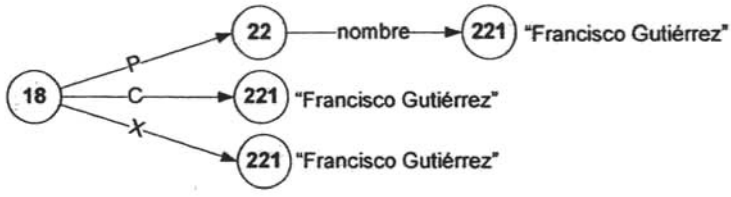
El proceso restante se describe como:

- $\sigma(\{17,18,19\}, C=X)$
  - Para cada nodo de {17,18,19} evaluar la condición C=X
    - Con el nodo que tiene identificador 17, evaluar la condición. *C=X* que contiene dos expresiones de camino como operandos, por lo que la condición se transforma en {221}={211}



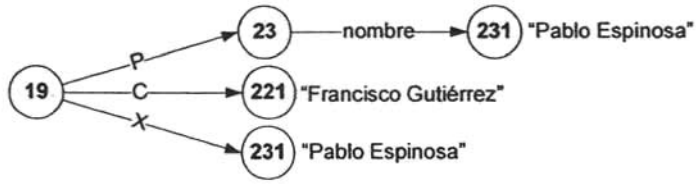
Esta condición resulta falsa y no se regresa dicho nodo 17.

- Con el identificador 18:



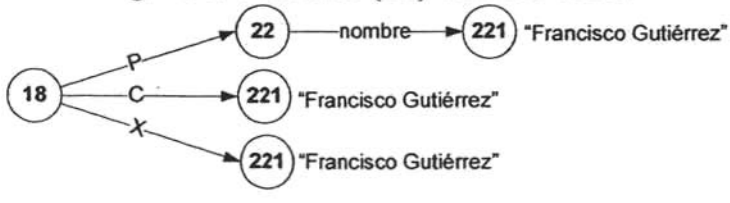
La condición se transforma en {221} = {221} lo cual es verdadero y por lo tanto almacena en la colección de resultado el nodo 18.

- Con el identificador 19:



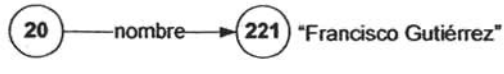
La condición se transforma en {221} = {231} lo cual es falso y por consiguiente no regresa el nodo 19.

- Regresa la colección {18} definido como:



- *Map({18}, π(X, nombre))*

  - La evaluación en este caso trabaja con solo un nodo (18), por lo que al evaluar la expresión de camino *X* regresa el nodo 221, para el cual se crea un nuevo nodo y una arista con etiqueta "*nombre*" que une al nodo 221 con el recién creado:



- *NewSSD({20})*

    - Al ser un solo nodo, no se une con ningún otro, por lo que *NewSSD* regresa el mismo nodo 20.

Por lo tanto, el resultado del proceso descrito por el plan lógico es el nodo con identificador 20, el cual contiene la información que se obtiene al evaluar la consulta con la base de datos de la figura 3.5.1.

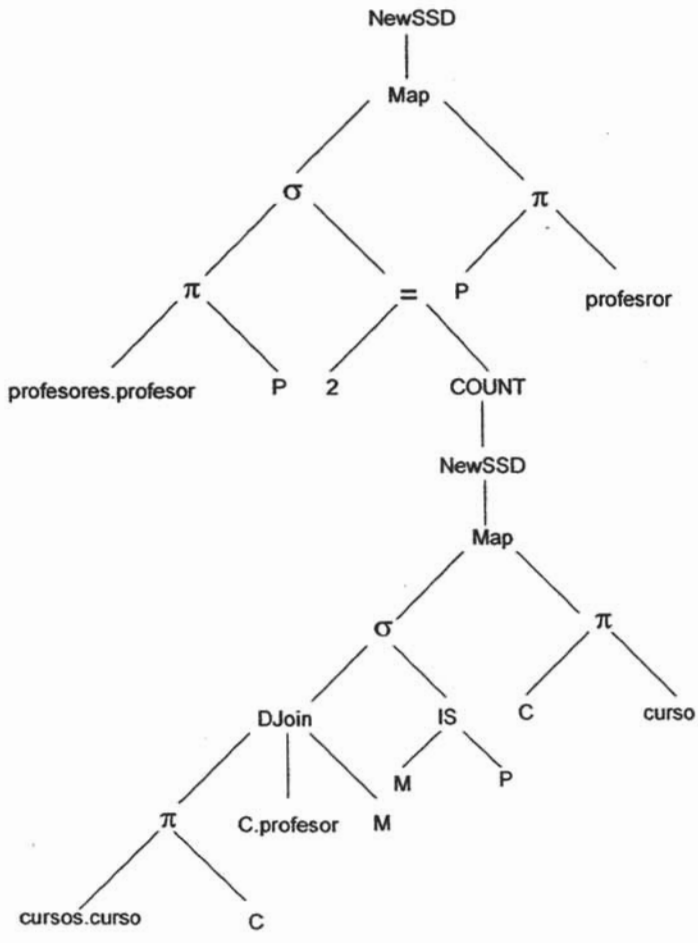
Los ejemplos 3.7.1 y 3.7.2 muestran la transformación de una consulta a plan lógico y el proceso de ejecución descrito por cada plan. Aún siendo consultas pequeñas, como las de estos ejemplos, cumplen con su objetivo de describir la función y comportamiento de un plan lógico de consulta. Sin embargo, el procesador de consultas desarrollado para esta tesis, da la opción al usuario de visualizar el plan lógico por cada consulta que ingrese. Se recomienda al lector interesado probar sus consultas en dicho programa de *software*.

Para finalizar esta sección se presenta un último ejemplo de consulta anidada dentro de una condición y muestra el plan lógico que la describe, en el que puede observase el caso en que se modifica una condición al pasar una consulta a plan lógico. La descripción del proceso de ejecución del plan se basa en la del ejemplo 3.7.2, por lo que no necesita explicarse esa parte en este ejemplo.

Ejemplo 3.7.3 La consulta:

```
SELECT profesor : P
FROM  profesores.profesor as P
WHERE 2 = COUNT (SELECT       curso: C
                 FROM         cursos.curso AS C,
                              C.profesor AS M
                 WHERE        M IS P)
```

es descrita por el siguiente plan lógico de consulta:



3.8 Operador DELTA (δ)

Con el operador δ se define que un dato dentro de una colección de datos semiestructurados es igual a otro si ambos tienen los mismos descendientes. Por ello, en una colección puede haber datos repetidos aún cuando dentro de ésta no existan dos o más nodos con el mismo identificador. Este concepto es diferente al definido por el operador condicional (=) utilizado en consultas, pues en este último la igualdad entre datos se resuelve sólo tomando en cuenta sus identificadores. Para datos primitivos, su equivalencia se define por su valor.

Cuando se utilizan colecciones, éstas aceptan datos duplicados, por lo que generalmente se le da la opción al usuario de eliminar datos repetidos. Para realizar esto, se define este operador, el cual proviene de uno utilizado para el mismo propósito en el álgebra relacional extendida. En el modelo del álgebra relacional puro (selección, proyección, unión, diferencia, producto cruz y renombrado) no existen valores duplicados, pues todos los operadores manejan conjuntos particulares, que son las relaciones. Existen operadores que extienden al álgebra relacional para el manejo de valores repetidos, los cuales ya no manejan conjuntos, sino bolsas (*bags*) [UlWi99] o multiconjuntos. Existen otras propuestas al respecto [DaGK82] que redefinen todos los operadores básicos del álgebra relacional, agregando un extra (δ) para la eliminación de duplicados.

La definición del operador δ para datos semiestructurados es la siguiente:

Sintaxis:

$\delta(D)$

Definición:

Dado una colección de nodos $D = \{d_1, d_2, ..., d_n\}$, regresará un conjunto de nodos $R \subseteq D$ tal que para cada $r \in R$ no exista otro $z \in R\text{-}r$ que cumpla la condición $z=r$ y $z.l = r.l$ para toda etiqueta l

$$R = \{ r \mid r \in D \wedge (\neg \exists z \in (R\text{-}r) \wedge \forall l \in L \ (z.l = r.l \vee z = r)) \}$$

Donde L es el conjunto de todas las etiquetas de arista contenidas en la base de datos.

La selección de nodos a regresar por este operador, no se basa solamente en discriminar aquellos elementos dentro de D que tengan el mismo identificador (si lo tienen entonces es el mismo nodo, por lo que se consideran los mismos hijos), sino que realiza una comparación entre los hijos de dichos nodos. De esta forma se determina que dos nodos n_1 y n_2 son iguales si la colección de hijos H_1 de n_1 es igual a la colección de hijos H_2 de n_2. Para determinar dichas colecciones H_1 y H_2 se hace uso de expresiones de camino que devuelven los hijos de un nodo específico.

La definición de este comportamiento para el operador δ, es necesaria debido a la forma en que funcionan los operadores anteriores. Operadores como π y *DJoin*, por mencionar sólo dos, crean nuevos nodos y aristas que apuntan a los datos de interés que se desea discriminar. Por ejemplo, en un plan lógico definido como:

$$\delta(\pi(\{21\}, X) \cup_c \pi(\{21\}, X))$$

Dado que cada operador π recibe una colección que tiene sólo un elemento (21), únicamente se creará un nodo con identificador no usado y una arista con etiqueta "*X*" que apunte al elemento (21). Es decir, para el primer π y un identificador no utilizado 31, se crea el dato representado como expresión de camino $31.X = \{21\}$ y regresa la colección {31}. Para el segundo π el comportamiento es similar, sólo que en este caso se utiliza un identificador diferente para la creación del nodo, por ejemplo el identificador 32, que se representa como $32.X = \{21\}$, por lo tanto, el segundo π devuelve la colección {32}. Ambos π seleccionaron al mismo nodo (21) pero devolvieron un nodo diferente (el nodo creado). Por lo anterior, para determinar que la información seleccionada por los π es la misma y así seleccionarla sólo una vez, es necesario trabajar con los hijos de dichos datos.

3.9 Operadores lógicos para la modificación de datos

El contenido de la base de datos puede modificarse mediante el borrado, actualización e inserción de información. En este apartado se presentan los operadores lógicos que describen cada uno de estos procesos.

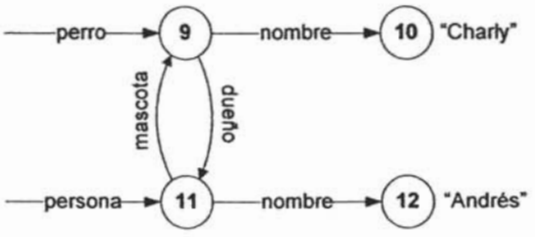
3.9.1 Operador Delete

La eliminación de nodos del grafo en una base de datos semiestructurados requiere revisar, antes de hacerla, que no la deje en un estado de inconsistencia, es decir, debe asegurarse que un dato no tenga una arista que apunte al dato a borrar. Por ello, es necesario analizar que al borrar un nodo, éste no sea descendiente de otro. En relación a la descendencia del mismo nodo, se debe borrar junto con éste, aún cuando ésta sea parte de otra Ssd-tabla distinta a la del nodo a borrar.

Este proceso de borrar nodos de una base de datos semiestructurados es semejante al de borrar, en una base de datos relacionales, una tupla que contiene datos a los que se hace referencia desde otra tupla en la misma base de datos. A lo anterior se le conoce como integridad referencial. Un sistema administrador de bases de datos relacionales, comúnmente, tiene la posibilidad de borrar datos en cascada. Esta técnica de borrado elimina toda tupla que haga referencia a la tupla especificada antes de borrarla. Algo similar debe realizarse en las bases de datos semiestructurados, pero en dos direcciones:

- Borrar toda la descendencia del nodo.- Eliminar los nodos (al eliminar un nodo se debe realizar el mismo proceso) encontrados por las aristas que salen.
- Borrar las referencias a dicho nodo. Eliminar todas las aristas que apuntan a él.

Durante este proceso existe la necesidad de contemplar el manejo de *ciclos* que puedan encontrarse dentro de la base de datos. Por ejemplo, si se desea borrar el nodo 9 del dato presentado en la figura siguiente:



Antes de borrar el nodo 9 hay que eliminar sus hijos, por lo que con la expresión de camino 9.# se recupera la colección de nodos {10, 11}. Con el nodo 10 no existe problema pues es un dato primitivo, pero al intentar borrar el nodo 11 si lo hay, porque antes de borrar el nodo 11 se deben eliminar sus hijos. La expresión 11.# devuelve la colección {9,12}, lo que indica borrar el nodo 9 antes del 11, esto origina caer en un ciclo. Para solucionar este problema existen dos opciones:

- Utilizar el concepto de transacción, que puede dejar a la base de datos en un estado de inconsistencia mientras se realiza, pero al terminarla, la base de datos debe seguir consistente. De esta forma antes de eliminar los hijos del nodo se borra el nodo en sí.

- Borrar antes que la descendencia del nodo todas las aristas que apunten a él.

Cualquiera de las dos es viable, siendo la última más directa al no tener que recurrir a otra técnica para completar la tarea.

Por lo anterior, es necesario describir un operador lógico que defina el proceso de borrado.

Sintaxis:

Delete(D)

Definición:

Dado una colección de datos $D=\{d_1, d_2, d_3, ..., d_n\}$, por cada d_i con $i = 1..n$, hacer:

- $d_i \in n.l \rightarrow n.l=n.l-\{d_i\}\ \forall n \in \beta \wedge \forall l \in L$ - Borrar referencias a d_i
- $Delete(d_i.\#)$ - Borrar descendencia
- $\beta = \beta - d_i$

Donde β es el conjunto de todos los identificadores en uso dentro de la base de datos y L el conjunto de todas las etiquetas que se utilizan en la base de datos.

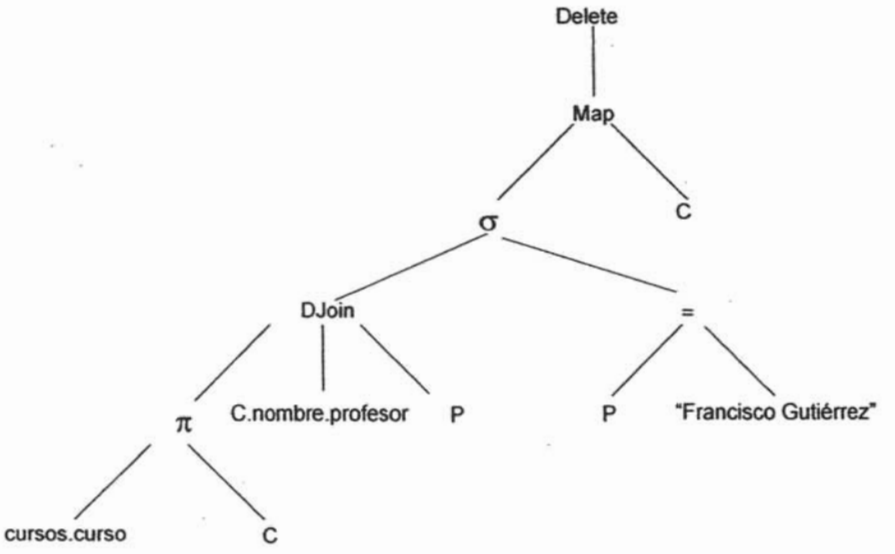
Como se ha visto, por todas las operaciones que implica borrar un nodo de la base de datos, este proceso es uno de los más costosos para un sistema administrador de bases de datos semiestructurados. Para realizarlo, prácticamente se tiene que revisar cada nodo y cada arista en busca de los nodos que hacen referencia al que ha de borrarse, lo cual, para una base de datos que supone maneja gran cantidad de información (nodos), resulta oneroso.

En el siguiente ejemplo 3.9.1.1 se muestra el traslado de una consulta de borrado de datos a plan lógico y la descripción de cómo los elimina.

Ejemplo 3.9.1.1 La consulta:

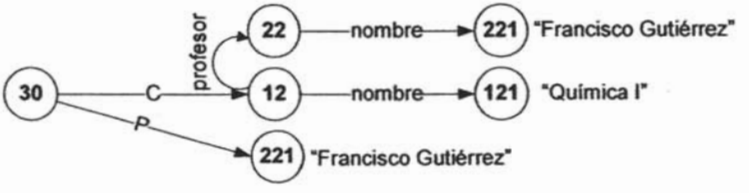
```
DELETE      C
FROM        cursos.curso AS C,
      C.profesor.nombre AS P
WHERE P = "Francisco Gutiérrez"
```

se representa por el plan lógico:



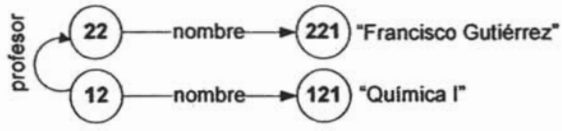
La descripción de la evaluación de este plan lógico considerando una base de datos como la de la figura 3.5.1, es la siguiente.

El *DJoin* dentro del operador σ devuelve una colección con un solo nodo {30} definido como:



El operador σ debe determinar si se satisface la condición *P="Francisco Gutiérrez"* por cada nodo devuelto por el *DJoin*. En este caso sólo es el

nodo con identificador 30, el cual si cumple con la condición. Con 30, sólo se selecciona aquel nodo que se alcanza con la arista que tiene etiqueta C. Por lo tanto, el operador *Map* devuelve el nodo 12:



Para este nodo con identificador 12, se realiza la eliminación de sus nodos:

- *Delete*(12)
  - 1.curso = {11,12,13}$-_c${12}={11,13}
  - *Delete*(12.#) = *Delete*(121,22)
    - Con 121
      - 12 .nombre = {}
      - *Delete*(121.#) = *Delete*({})
      - Liberar el nodo con identificador 121
    - Con 22
      - 12.profesor = {}
      - 2.profesor = {21,22,23}$-_c${22}={21,23}
      - *Delete*(22.#) = *Delete*(221)
        - 22.nombre = {}
        - *Delete*(221.#) = *Delete*({})
        - Liberar el nodo con identificador 221
      - Liberar el nodo con identificador 22
  - Liberar el nodo con identificador 12

3.9.2 Operador Update

Si el proceso de borrar nodos de una base de datos semiestructurados es costoso para el sistema administrador de bases de datos semiestructurados, actualizarlos es algo similar. En este procedimiento tanto se crean nodos nuevos, como se buscan las aristas que apuntan a cierto nodo dentro de toda la base de datos.

Así como en el caso de borrar, también es necesario manejar las referencias de los nodos que se manejen durante la actualización. Por esta razón, procede crear un operador lógico que maneje la actualización de los datos.

Sintaxis:

Update(D, U)

Definición:

Dado dos colecciones $D = \{d_1, d_2, ..., d_n\}$ y $U = \{d_u\}$, se realiza la siguiente operación:

$$\forall d \in D \wedge \forall n \in \beta \wedge \forall l \in L \; (d \in n.l \rightarrow n.l = n.l -_c \{d\} \cup_c U)$$

Donde β es el conjunto de todos los identificadores en uso dentro de la base de datos y L el conjunto de todas las etiquetas.

Lo que hace este operador es redireccionar todas las aristas que apuntan al nodo a actualizar, hacia el nuevo nodo creado con la información que lo reemplaza. El ejemplo 3.9.2.1 muestra el concepto de actualizar un nodo.

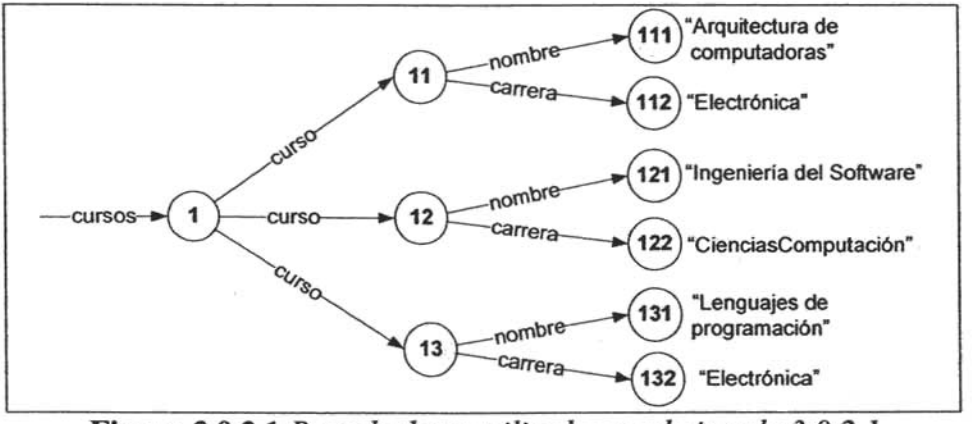
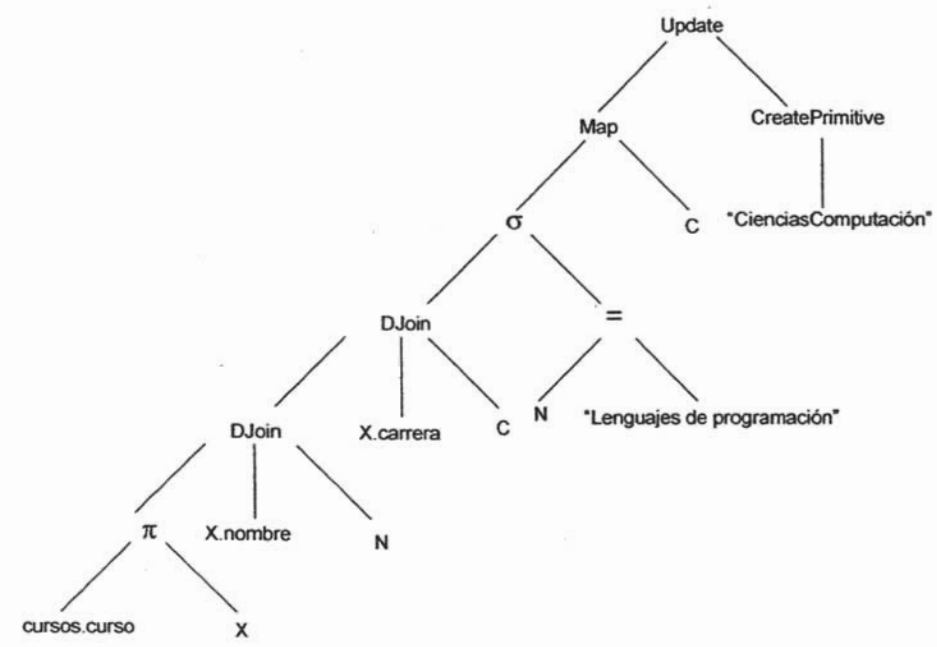


Figura 3.9.2.1 *Base de datos utilizada por el ejemplo 3.9.2.1*

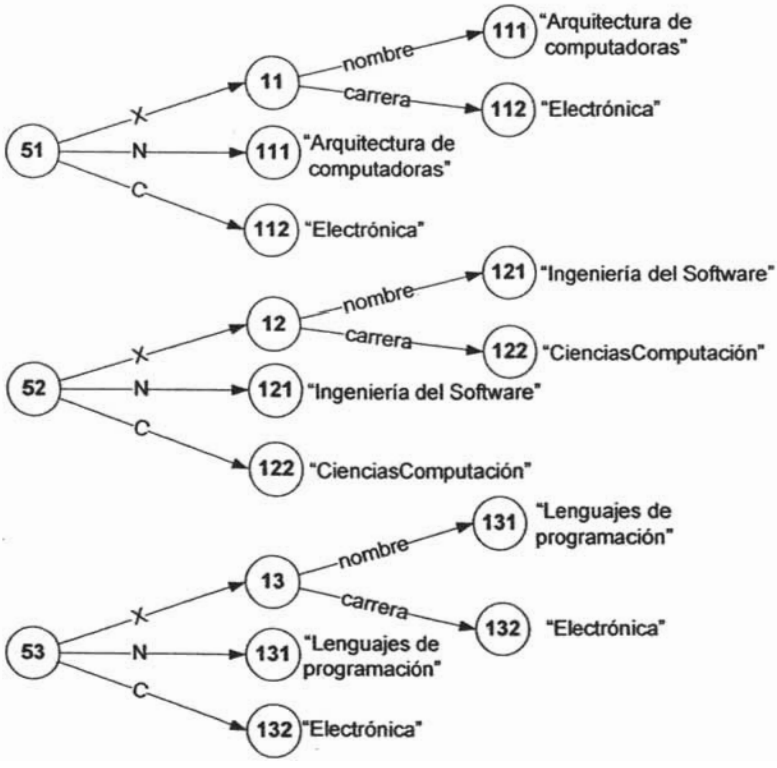
Ejemplo 3.9.2.1 La consulta de modificación de datos:

```
UPDATE      C
SET         "CienciasComputacion"
FROM        cursos.curso AS X,
      X.nombre AS N,
      X.carrera AS C
WHERE N="Lenguajes de programación"
```

puede representarse por un plan lógico definido como:



Si se aplica este plan a la base de datos de la figura 3.9.2.1, el operador *DJoin* interno del operador σ devolvería la colección de nodos {51,52,53} definido como:



El operador σ sólo devolvería el nodo 53 pues es el único que cumple con la condición *N="Lenguajes de programación"*. Enseguida, con el nodo 53 el operador *Map* selecciona aquel nodo con etiqueta "*C*" y devuelve el nodo 132 definido como:

En este punto se obtiene la colección *U* que contendrá al nuevo nodo definido por *CreatePrimitive("CienciasComputación")*, el cual regresa el nodo:

Para completar el proceso, se buscan en toda la base de datos aquellas aristas que hacen referencia a 132, que es el nodo que la consulta intenta actualizar. Los nodos encontrados serían {13,53,54}, cuyas aristas se redireccionan al nuevo dato primitivo 55. El estado de la base de datos después de esta actualización se muestra en la figura 3.9.2.2.

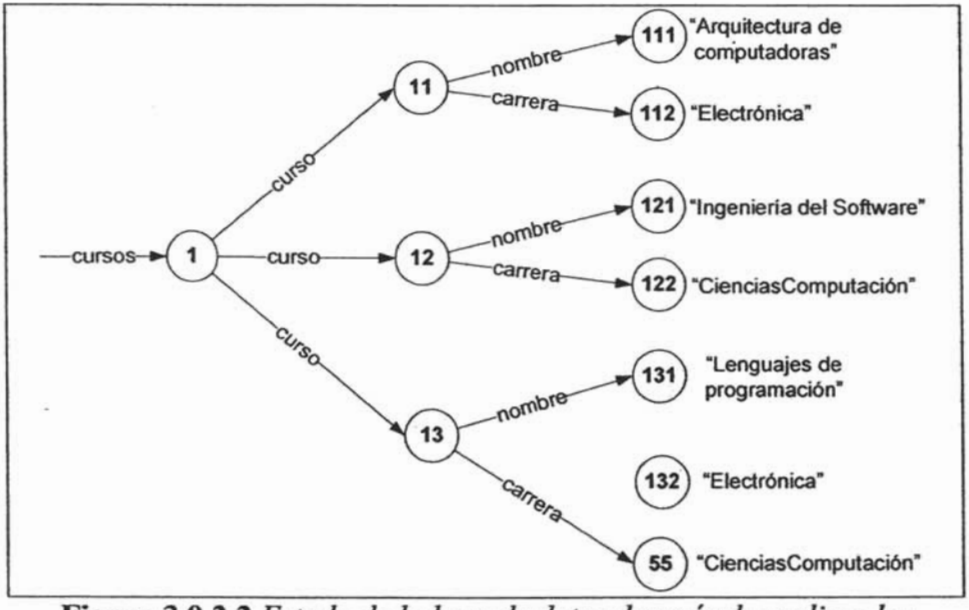


Figura 3.9.2.2 *Estado de la base de datos después de realizar las operaciones descritas por la consulta del ejemplo 3.9.2.1*

Como puede observarse en la figura 3.9.2.2, después de ejecutar una actualización es posible que existan nodos no accesibles desde la raíz de alguna Ssd-tabla y podrían borrarse de la base de datos. Sin embargo, para borrar estos nodos "basura" no se debe seguir la misma filosofía del operador *Delete*, pues se podrían borrar nodos aún accesibles desde alguna Ssd-tabla, por lo que a estos nodos "basura" se les borran sólo sus aristas que salen y el nodo en sí.

La mayoría de los operadores lógicos definidos en este capítulo requieren de la construcción de nuevos nodos; pero como se acaba de mencionar, no todos terminarán siendo parte de la base de datos y, por lo tanto, ser accesibles desde alguna Ssd-tabla. Por esta característica se dice que tales nodos son temporales y su creación física o sólo en memoria depende del motor de ejecución.

3.9.2.1 Inserción de datos

La inserción es un caso especial de la actualización y ocurre cuando dentro de la cláusula SET se utiliza el dato especificado por la cláusula UPDATE. Por ejemplo:

```
UPDATE      C
SET         C UNION {facultad: "Ciencias"}
FROM        cursos.curso as C
```

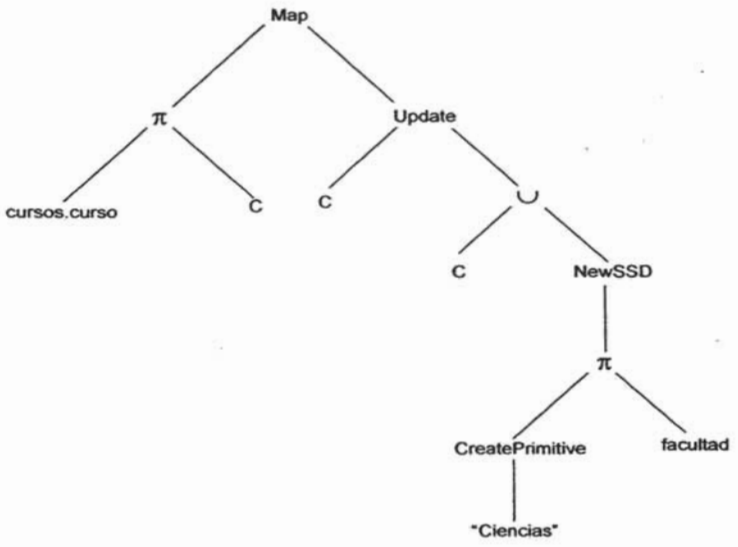
En esta consulta se inserta o agrega un nuevo dato primitivo "*Ciencias*" al nodo *C* mediante una arista con etiqueta "*facultad*". En general, el dato seleccionado para actualizar por la cláusula UPDATE, puede aparecer dentro de la cláusula SET. Por lo anterior, la definición del operador UPDATE debe ampliarse de forma que considere esta modalidad.

Básicamente es el mismo operador, sólo que la traducción a plan lógico se modifica, algo que muestra el ejemplo 3.9.2.1.1.

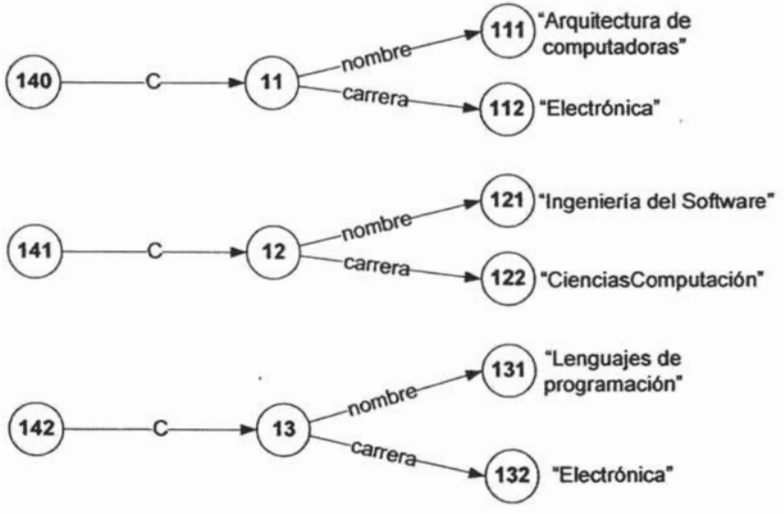
Ejemplo 3.9.2.1.1 La consulta:

```
UPDATE      C
SET         C UNION {facultad: "Ciencias"}
FROM        cursos.curso as C
```

se representa por el plan lógico:



Si se aplica este plan a la base de datos de la figura 3.9.2.1, el operador π regresaría la colección de nodos $D = \{140, 141, 142\}$ definidos como:



Una vez obtenida la colección D del operador π, el operador *Map* evalúa el *Update* para cada nodo en D. *Update* realiza el proceso descrito anteriormente:

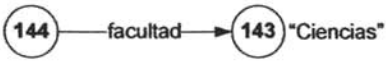
Con 140:

Update(C, C ∪ NewSSD(π(CreatePrimitive("Ciencias"), facultad)))

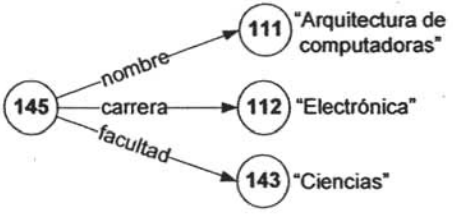
El primer parámetro del *Update* = 140.C = {11}

Update({11} , C ∪ NewSSD(π(CreatePrimitive("Ciencias"), facultad)))

- Obtener la segunda colección:
  - *NewSSD* regresa el nodo 144, el cual se definie como:



  - Se realiza la operación C ∪ {144} = 140.C ∪{144} = {11} ∪ {144} =



∴ la segunda colección es {145}

Update({11}, {145})

- Los nodos que apunta al nodo 11 son: 1 y 140
  - Para el nodo 1:
    - 1.curso = {11, 12, 13}
    - {11} es un subconjunto de {11,12,13}, por lo que se realiza la siguiente operación:
      1.curso = 1.curso – {11} $\cup_c$ {145} = {12,13, 145}
  - Para el nodo 140:
    - 140.C = {11}
    - 140.C = 140.C – {11} $\cup_c$ {145} = {145}

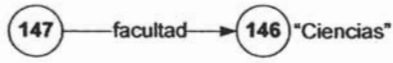
Con 141:

Update(C, C ∪ NewSSD(π(CreatePrimitive("Ciencias"), facultad)))

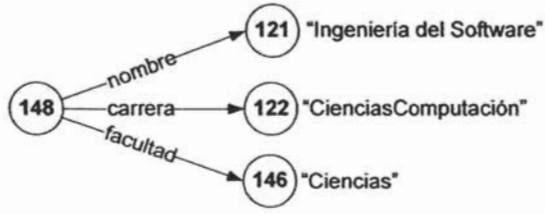
El primer parámetro del *Update* = 141.C = {12}

Update ({12} , C ∪ NewSSD(π(CreatePrimitive("Ciencias"), facultad)))

- Obtener la segunda colección:
  - *NewSSD* regresa el conjunto {147} donde el nodo 147 se define como:



  - Se realiza la operación C ∪ {147} = 141.C ∪ {147} = {12} ∪ {147} =



∴ la segunda colección es {148}

Update ({12}, {148})

- Los nodos que apunta al nodo 12 son 1 y 141
  - Para el nodo 1:
    - 1.curso = {12, 13, 145}
    - {12} es un subconjunto de {12,13,145}, por lo que se realiza la siguiente operación:
      1.curso = 1.curso – {12} $\cup_c$ {148} = {13, 145, 148}
  - Para el nodo 141:
    - 141.C = {12}
    - 141.C = 141.C – {12} $\cup_c$ {148} = {148}

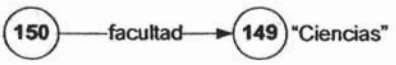
Con 142:

Update(C, C ∪ NewSSD(π(CreatePrimitive("Ciencias"), facultad)))

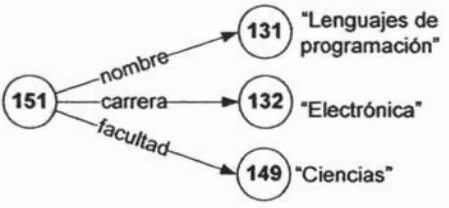
El primero parámetro del *Update* = 142.C = {13}

Update ({13} , C ∪ NewSSD(π(CreatePrimitive("Ciencias"), facultad)))

- Obtener la segunda colección:
  - *NewSSD* regresa la colección {150} donde el nodo 150 esta definido como:



  - Se realiza la operación C ∪ {150} = 142.C ∪ {150} = {13} ∪ {150} =



∴ la segunda colección es {151}

Update({13}, {151})

- Los nodos que apuntan al nodo 13 son 1 y 142
  - Para el nodo 1:
    - 1.curso = {13, 145, 148}
    - {13} es un subconjunto de {13,145, 148}, por lo que se realiza la siguiente operación:

      1.curso = 1.curso – {13} $\cup_c$ {151} = {145, 148,151}

- Para el nodo 142:
  - 142.C = {13}
  - 142.C = 142.C – {13} $\cup_c$ {151} = {151}

El estado de la base de datos después de realizada esta actualización se muestra en la figura 3.9.2.1.1.

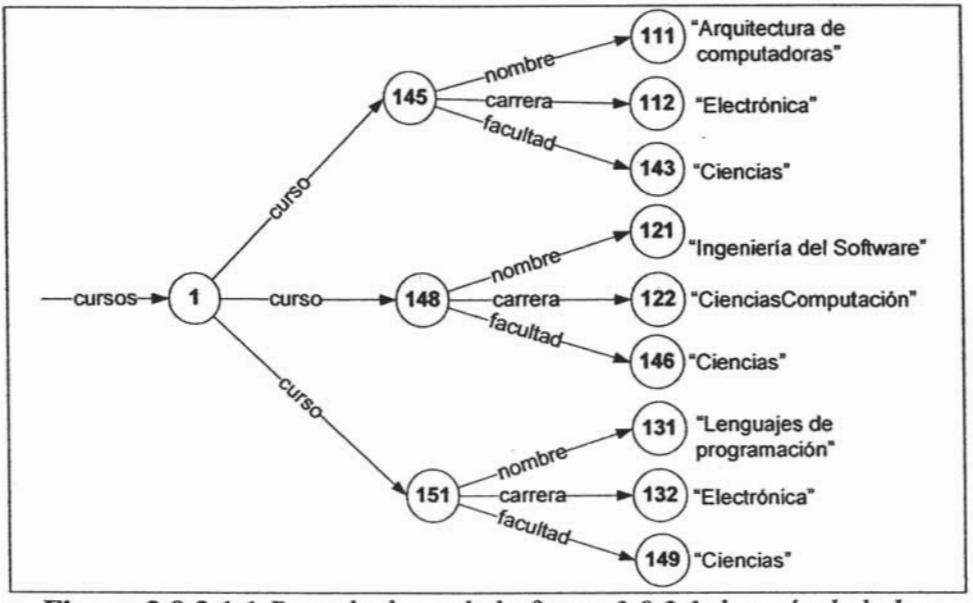


Figura 3.9.2.1.1 *Base de datos de la figura 3.9.2.1 después de haber realizado la actualización del ejemplo 3.9.2.1.1*

3.10 Resumen y discusión

La generación del plan lógico para las consultas tiene como objetivo realizar optimizaciones a un nivel no apegado al sistema de almacenamiento primitivo. Con lo anterior, el plan lógico y, por consiguiente, las técnicas de optimización desarrolladas para este plan pueden ser utilizadas en otro sistema administrador para datos semiestructurados.

En este capítulo se describió cada uno de los operadores utilizados para crear el plan lógico de consulta. Estos operadores tienen fundamento en otras contribuciones a esta área de bases de datos [MAG+97, BeTz99, FrHP02]. Sin embargo, algunas no describen formalmente sus operadores,

lo que da lugar a malintepretaciones. También se observó la carencia en la definición de operadores para el manejo de actualizaciones, pues sólo se concentran en operaciones de consulta. En este capítulo se dio una propuesta de operadores lógicos que describen los procesos de modificación a una base de datos semiestructurados. Tal fue la homogenización de los operadores que hizo posible describir el proceso de inserción de datos, sin la necesidad de construir un nuevo operador que lo definiera.

La concepción de los operadores lógicos se orientó a poder aplicar las técnicas de optimización desarrolladas para el álgebra relacional [SiKS02, GaUW00], como lo es la de *Push Selections Down*. Sin embargo, debido a que esas optimizaciones han sido altamente estudiadas, la dirección que toma esta tesis al respecto se enfoca hacia las partes específicas del ambiente de datos semiestructurados. Por lo anterior, queda como tema abierto demostrar que realmente estos operadores tengan las mismas propiedades conmutativas y asociativas que se encuentran en los relacionales. De la misma forma, demostrar que es posible utilizar las técnicas de *Push Selections Down* y *Push Proyections Down* en un plan lógico que utilice los operadores de este trabajo.

En el siguiente capítulo se explican diferentes técnicas de optimización diseñadas para ayudar a reducir el trabajo que requiere evaluar un plan lógico de consulta. Esta labor demanda muchos recursos en el ambiente de datos semiestructurados, debido a la continua creación de nodos temporales y a la búsqueda de información a través de la base de datos mediante expresiones de camino.

Capítulo 4

Optimización del plan de consulta

Como se mostró en el capítulo 2, la mayor cantidad de accesos a disco ocurre al tratar de encontrar datos semiestructurados requeridos por una consulta mediante expresiones de camino. Esto es debido, principalmente, a la necesidad de recorrer el árbol de datos sin la posibilidad de saltar alguna ramificación, pues podría no tomarse en cuenta algún dato que cumple con los requerimientos de la consulta.

Según se ha mencionado, optimizar la ejecución de una consulta en una base de datos se enfoca a minimizar el número de accesos al almacenamiento secundario o disco. Esto es debido a que recuperar información desde disco, generalmente, es el proceso más costoso (refiriéndose a la duración de la ejecución a causa del tiempo de acceso, tiempo de búsqueda, etc.). Por lo anterior, deben localizarse aquellas partes principales en una consulta que demandan el mayor uso de este recurso para ejecutarse.

Los ejemplos de la sección 1.4.2 del capítulo 1 mostraron el comportamiento de las consultas. En ellos se distingue como el número de *ciclos* de una consulta se determina por el total de identificadores diferentes que se puedan asignar a las tablas temporales especificadas en la cláusula FROM. Estos identificadores son devueltos por las expresiones de camino. Podría pensarse que al reducir los *ciclos* baje el tiempo de ejecución de la consulta; sin embargo, aminorar éstos no es significativo, porque no son realmente los causantes de los accesos a disco.

Con cada *ciclo* se evalúa un identificador (dato semiestructurado) contenido en cada una de las tablas temporales, aún no utilizado y que cumpla con el camino correspondiente. Como se mostró en la sección 2.2.5, estos identificadores se recuperan al evaluar una expresión de camino. Por lo anterior, son estas expresiones las que generan, principalmente, los accesos a disco, por lo que resulta necesario encontrar una forma de mejorar su ejecución.

4.1 Resumen de datos

Con el fin de reducir el número de accesos a disco necesarios para localizar información que cumpla con alguna expresión de camino, se describe una técnica que utiliza un "esquema" de la base de datos y a partir de éste conseguir la colección de identificadores que recupera la expresión de camino. A dicho "esquema" se le denominará *resumen de datos*. Esta sección describe el mismo y su construcción, para dar paso, posteriormente, a la definición de la técnica de optimización.

Definición 4.1.1
Un *camino de etiquetas* es una secuencia de etiquetas de arista $l_1.l_2.....l_n$ donde cada l_i con $i=1...n$ es una etiqueta de arista en el grafo del dato semiestructurado y para cada pareja $l_j.l_{j+1}$ con $j=1...n-1$ existe una conexión entre ellas mediante un nodo.

Definición 4.1.2
Un *resumen de datos* para una Ssd-tabla T es un grafo dirigido G con etiquetas en las aristas, donde para cada camino de etiquetas diferente en T existe un camino equivalente en G y para cada camino de etiquetas en G existe al menos uno en T.

Un *resumen de datos* se enfoca a la estructura de los datos y no a los datos en sí. Esto es, no contiene información sobre los datos primitivos (cadenas, números, fechas, etc.). En vez de ello, define una recopilación de los diferentes caminos que puede tener una Ssd-tabla iniciando desde la raíz.

Un *resumen de datos* define solamente una vez cada camino de etiquetas extraído de la Ssd-tabla origen y para todo camino de etiquetas en el *resumen de datos* se debe encontrar un camino igual en el origen.

Ejemplo 4.1.1 El *resumen de datos* para la Ssd-tabla de la figura 4.1.1 se muestra en la figura 4.1.2.

El nombre de un *resumen de datos* para una tabla de nombre "*X*" será "*X*-". Esta elección fue por razones de implementación y diseño del lenguaje, pues el caracter guión (-) no es válido dentro de una etiqueta (nombre de Ssd-tabla o etiqueta de arista) para la gramática del lenguaje. De esta forma se asegura que el usuario de la base de datos no defina una

tabla con nombre "*X*-"; reservando dicha etiqueta para el *resumen de datos*.

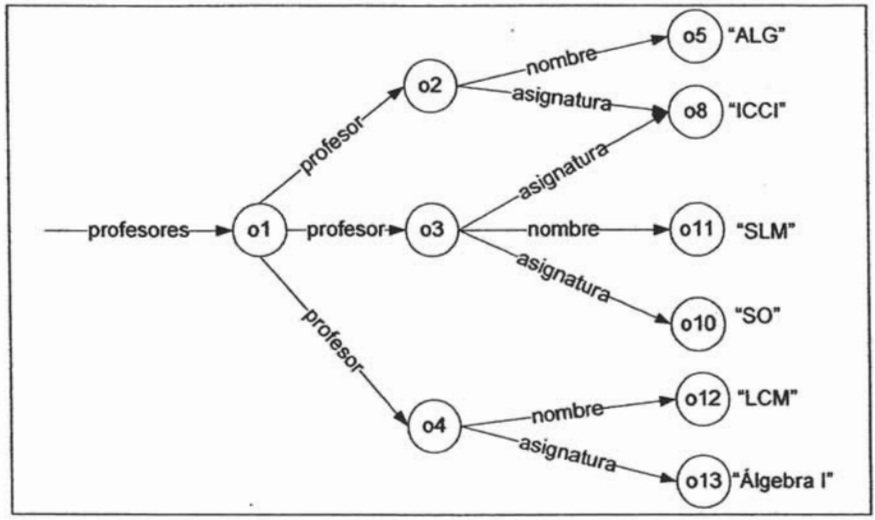


Figura 4.1.1 *Ssd-tabla profesores*

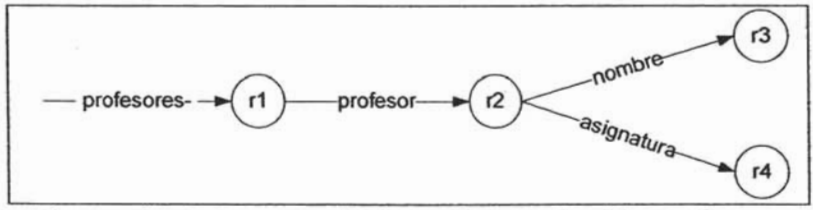


Figura 4.1.2 *Resumen de datos de la Ssd-tabla profesores de la figura 4.1.1*

Ejemplo 4.1.2 Para una Ssd-tabla que comparte información con otra Ssd-tabla, como en la base de datos de la figura 4.1.3, el *resumen de datos* contemplará todo camino de etiquetas que pueda alcanzar a algún nodo desde la raíz de la Ssd-tabla. Por ejemplo, la figura 4.1.4 muestra el *resumen de datos* para la Ssd-tabla libros y la figura 4.1.5 muestra el de la Ssd-tabla autores.

Como puede observarse en los ejemplos 4.1.1 y 4.1.2, es posible representar un *resumen de datos* como un dato semiestructurado. De esta manera puede almacenarse en la misma base de datos junto con las Ssd-tablas.

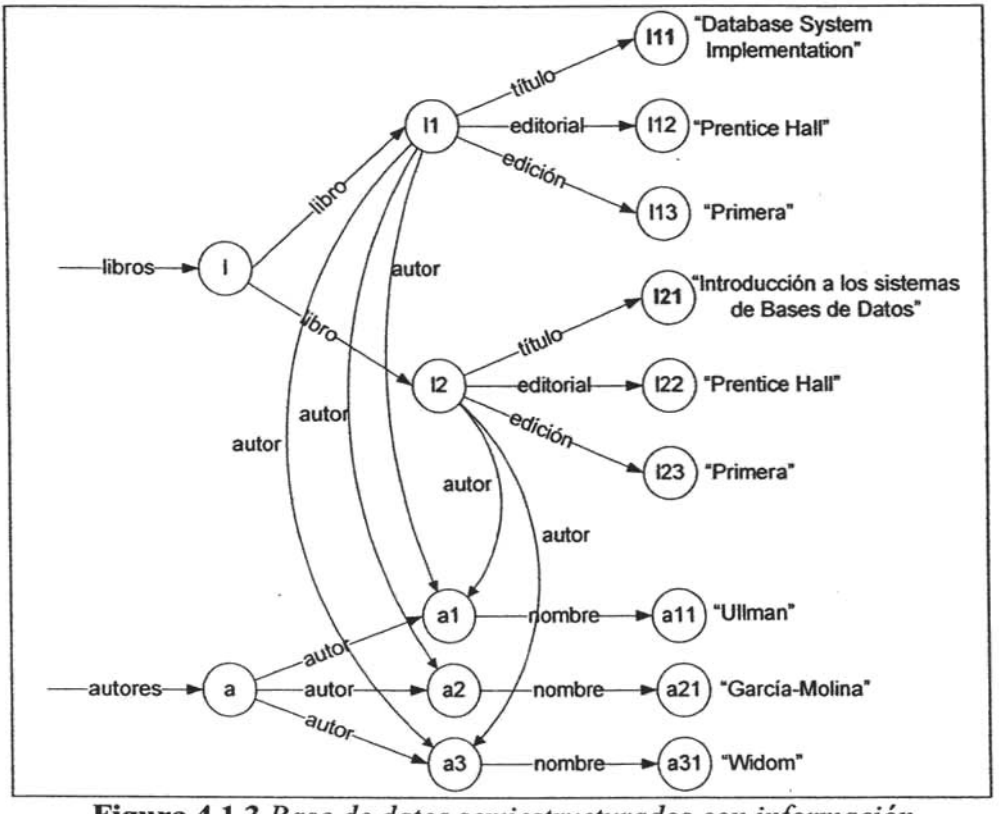


Figura 4.1.3 *Base de datos semiestructurados con información compartida entre Ssd-tablas*

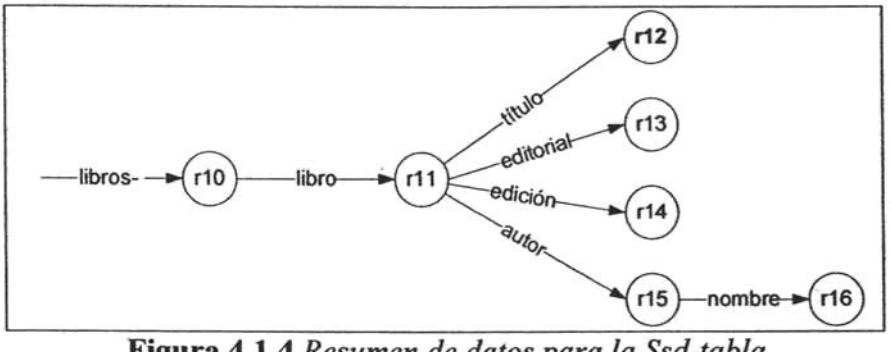


Figura 4.1.4 *Resumen de datos para la Ssd-tabla libros de la figura 4.1.3*

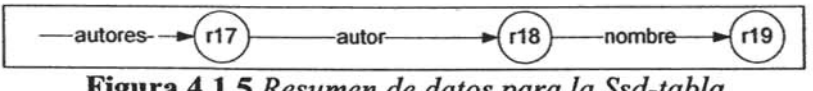


Figura 4.1.5 *Resumen de datos para la Ssd-tabla autores de la figura 4.1.3*

4.1.1 Optimización de expresiones de camino extendidas

Con la información contenida en un *resumen de datos*, puede realizarse una optimización a la evaluación de las consultas, específicamente hablando, a la evaluación de las expresiones de camino extendidas. Este tipo de optimización fue definido en [McWi99], con el nombre de expansión de camino (*Path Expansion*). Este último se basó de igual forma en el trabajo de [FeSu98], donde se hace uso de un *resumen de datos*, como el aquí presentado, con el nombre de *Graphic Schema*.

La optimización para este trabajo se define de la siguiente forma:

Optimización I
Reescribir las expresiones de camino de una consulta para reemplazar aquellas partes que hacen uso de operadores de cerradura (*, +, ?) a nivel de etiquetas, con una expresión de camino equivalente, que se encuentra recorriendo la expresión de camino original sobre el *resumen de datos*, a manera de eliminar dichos operadores.

Ejemplo 4.1.1.1 Utilizando la Ssd-tabla de la figura 4.1.1.1 con su correspondiente *resumen de datos*, evaluar la consulta siguiente:

```
SELECT      Nombres : N
FROM        biblioteca.#*.nombre as N
```

En esta consulta se evalúa una expresión de camino que puede expandirse utilizando la técnica de optimización I. El primer paso consiste en recorrer el camino descrito por la expresión pero dentro del *resumen de datos* de la Ssd-tabla correspondiente. Como en esta ocasión la expresión de camino inicia desde la Ssd-tabla "biblioteca" se usará el resumen de datos llamado "*biblioteca-*".

Al evaluar la sección donde se utiliza la cerradura de Kleene (#*) será necesario explorar el árbol del *resumen de datos* recorriendo todas las ramificaciones posibles, esto es:

- biblioteca-.revistas
- biblioteca-.libros
- biblioteca-.revistas.revista
- biblioteca-.libros.libro

- biblioteca-.revistas.revista.título
- biblioteca-.revistas.revista.número
- biblioteca-.libros.libro.título
- biblioteca-.libros.libro.editorial
- biblioteca-.libros.libro.edición
- biblioteca-.libros.libro.autor
- biblioteca-.libros.libro.autor.nombre

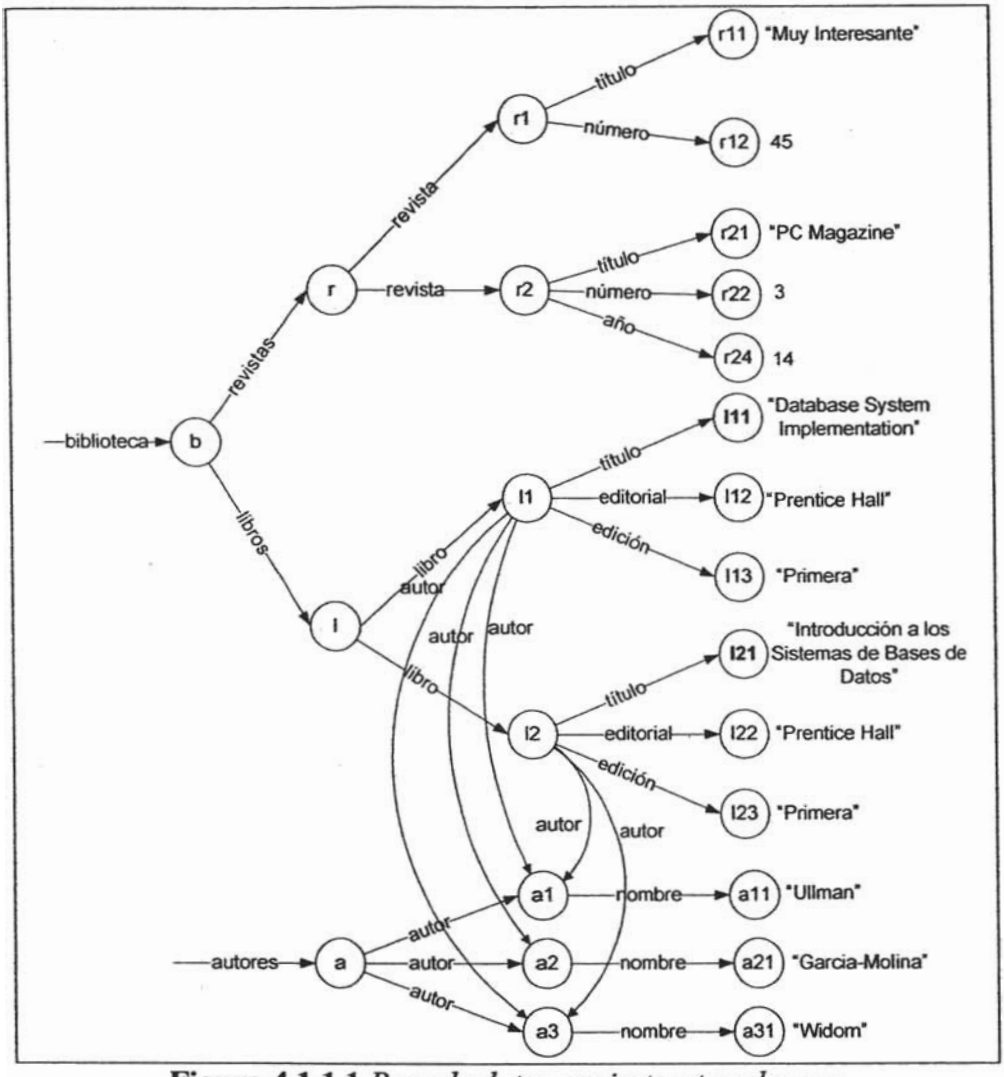


Figura 4.1.1.1 *Base de datos semiestructurados con información de una biblioteca*

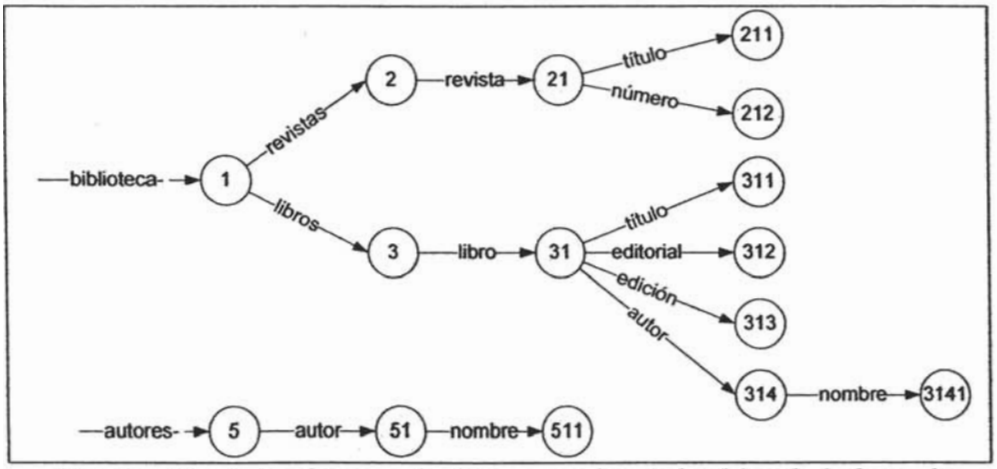


Figura 4.1.1.2 *Resúmenes de datos para las Ssd-tablas de la base de datos de la figura 4.1.1.1*

Con cada una de estas ramificaciones se evalúa cuál de ellas finaliza con la etiqueta "*nombre*". La única ramificación que cumple con lo anterior es *biblioteca-.libros.libro.autor.nombre*. Después de haber identificado los caminos válidos para la expresión de camino original, ésta se reemplaza por el camino recorrido en el *resumen de datos*, reescribiendose la consulta como sigue:

```
SELECT Nombres : N
FROM  biblioteca.(libros.libro.autor).nombre as N
```

Para esta consulta, sin aplicar la técnica de optimización, sería necesario el siguiente número de accesos a disco para ejecutarla:

- 1 acceso para recuperar los hijos de biblioteca
- 1 acceso para recuperar los hijos de revistas
- 2 accesos para recuperar los hijos de cada revista
- 1 acceso para recuperar los hijos de libros
- 2 accesos para recuperar los hijos de cada libro
- 5 accesos para recuperar los hijos de cada autor de cada libro

Total: 12 accesos a disco

En cambio, con la técnica de optimización se requieren:

- 1 acceso para recuperar los hijos de biblioteca
- 1 acceso para recuperar los hijos de libros
- 2 accesos para recuperar los hijos de cada libro
- 5 accesos para recuperar los hijos de los autores de cada libro

Total: 9 accesos a disco

Este ejemplo demuestra el ahorro de accesos al no considerar ramificaciones que no aportarán datos a la consulta. En una base de datos con un mayor número de ramificaciones esta técnica reducirá en mayor cantidad el número de accesos a disco.

El siguiente ejemplo muestra cómo se integran varios caminos válidos dentro de la expresión de camino.

Ejemplo 4.1.1.2 Supóngase que en el *resumen de datos* "biblioteca-" de la figura 4.1.1.2 aumenta el número de ramificaciones quedando como el de la figura 4.1.1.3 y se evalúa la misma expresión de camino del ejemplo 4.1.1.1: *biblioteca.#*.nombre*

En este caso el recorrido completo de la expresión de camino se da en las tres siguientes:

- biblioteca.libros.libro.autor.nombre
- biblioteca.artículos.artículo.autor.nombre
- biblioteca.reportes.reporte.autor.nombre

Por lo que la expansión de la expresión de camino original se realiza substituyendo la expresión "#*" por un conjunto de caminos unidos por una disyunción. Para este ejemplo, el resultado de la expansión es el siguiente:

biblioteca.(libros.libro.autor|artículos.artículo.autor|reportes.reporte.autor).nombre

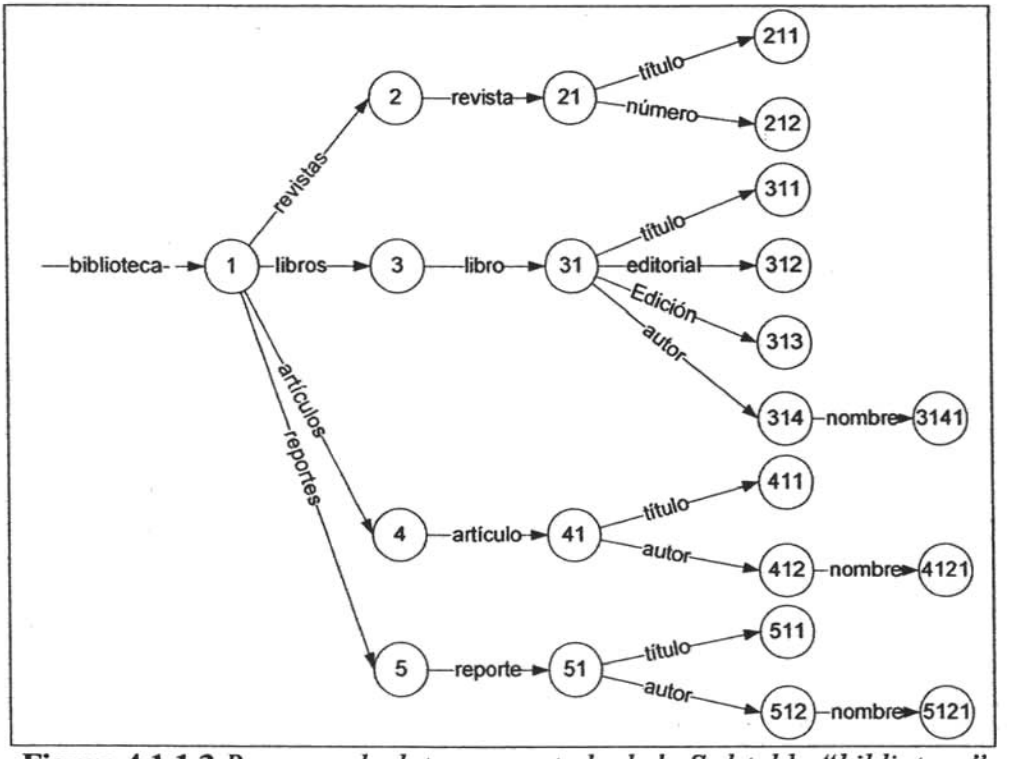


Figura 4.1.1.3 *Resumen de datos aumentado de la Ssd-tabla "biblioteca"*

La expansión de caminos con operadores de cerradura + o ? procede de forma similar. Esta técnica, tiene una falla dentro de un ambiente multiusuario, pues en caso de modificarse el *resumen de datos*, como por ejemplo realizando una actualización, después de haber expandido la expresión de camino, al momento de la ejecución no se tomarán en cuenta las modificaciones realizadas y podría no considerar datos que deberían ser devueltos por la consulta, algo que podría ser vital en sistemas de alta carga de trabajo. Además, en una base de datos que no contenga caminos comunes (dando como resultado un *resumen de datos* exactamente igual que la base de datos en sí, sin datos primitivos), supondrá recorrer el mismo camino dos veces, una para expandir la expresión de camino y otra para recuperar los datos. Estos problemas se retomarán más adelante en este mismo capítulo para solucionarlos.

Existe otra utilidad que puede dársele al *resumen de datos*. Utilizar las expresiones de camino extendidas puede deberse a dos circunstancias: porque se requiere recuperar información abarcando una gran extensión de los datos; o porque quien escribe la consulta no está seguro de donde se encuentran los datos. Tratándose de datos semiestructurados, el segundo

escenario tiene una posibilidad más alta de ocurrir. Esto es a causa, principalmente, de la carencia de un esquema que muestre al usuario la estructura de la información contenida en la base de datos. Es aquí donde el *resumen de datos* puede ayudar, a quien escribe las consultas, a conocerla y así recuperar la parte de información en la que realmente está interesado. En un trabajo similar al aquí presentado [MAG+97], se utiliza un concepto aproximado al *resumen de datos* descrito anteriormente, llamado *Data Guide*. Con él, se ayuda al usuario a escribir expresiones de camino que realmente existan en la base de datos, presentándolo con una interfaz gráfica en forma de árbol de directorios. Esta funcionalidad también se implementa como una herramienta extra al procesador de consultas desarrollado con esta tesis.

4.1.2 Resumen de datos con identificadores

Con lo visto hasta el momento, el *resumen de datos* ayuda a discriminar rutas de expresiones de camino extendidas; sin embargo, para las expresiones de camino normales y para mejorar la recuperación de datos por parte de las mismas, aún no se cuenta con alguna técnica. Es por esto que es necesario definir un *resumen de datos* que contenga más información acerca de los datos para una posible optimización extra.

Definición 4.1.2.1
Un *resumen de datos con identificadores* para una Ssd-tabla T es un grafo dirigido G con etiquetas en las aristas, donde para cada camino de etiquetas diferente en T existe uno equivalente en G y para cada camino de etiquetas en G existe al menos uno en T; asimismo, para cada nodo en G se integra una colección de identificadores de todos los datos semiestructurados que son alcanzables desde la raíz de T a través de L, donde L es el camino de etiquetas descrito desde la raíz de G a dicho nodo.

Un *resumen de datos con identificadores* realiza un enlace entre cada uno de sus nodos con los datos reales en la Ssd-tabla a la cual pertenece, esto es, crea una conexión entre el esquema de la base de datos y los datos demiestructurados. Aún cuando este tipo de resumen contiene más información acerca de los datos, la relativa a las relaciones "padre-hijo" entre datos semiestructurados no se almacena, siendo esta particularidad la razón principal de la existencia de ramificaciones en una Ssd-tabla. Los datos primitivos tampoco entran en este caso.

Ejemplo 4.1.2.1 La colección de identificadores contenidos en cada nodo de un *resumen de datos con identificadores* para la Ssd-tabla "*profesores*" de la figura 4.1.1, puede verse de dos formas diferentes:

1. Como ligas entre los datos y el *resumen de datos* (figura 4.1.2.1):
   Cada liga se crea utilizando una arista con etiqueta "*Data-*"; de esta forma se reserva su uso para esta situación.
2. Como un dato primitivo especial que contiene los identificadores mencionados (figura 4.1.2.2):
   Los datos primitivos se enlazan al nodo del resumen de forma semejante a la del punto anterior, con una arista con etiqueta "*Data-*".

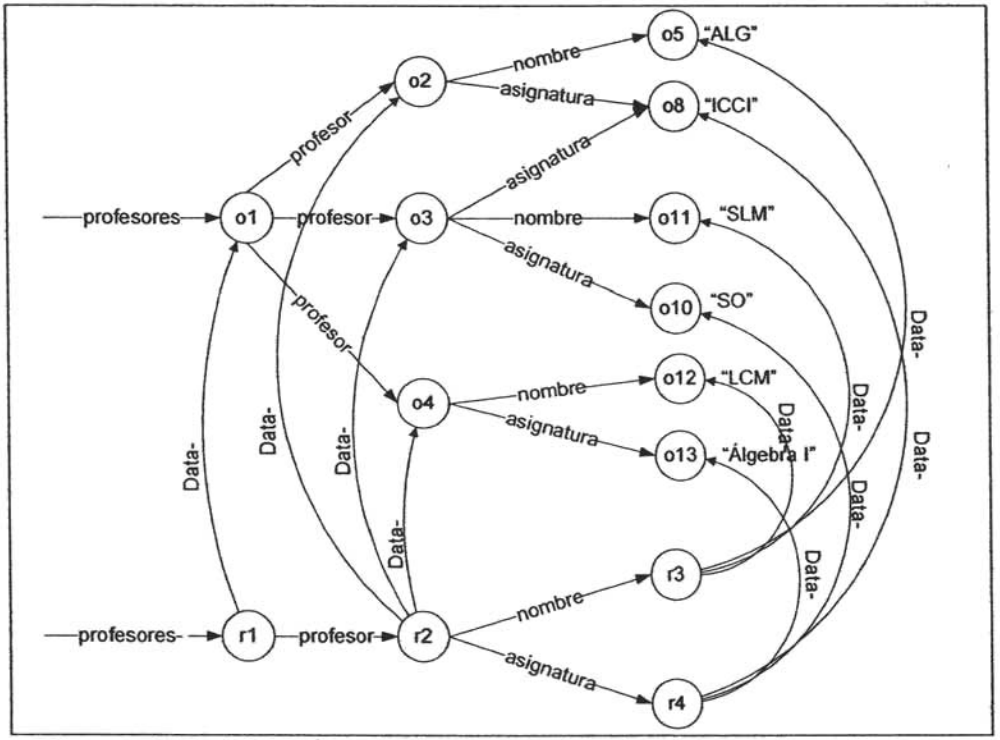


Figura 4.1.2.1 *Resumen de datos con identificadores utilizando aristas para representar las ligas entre él y los datos*

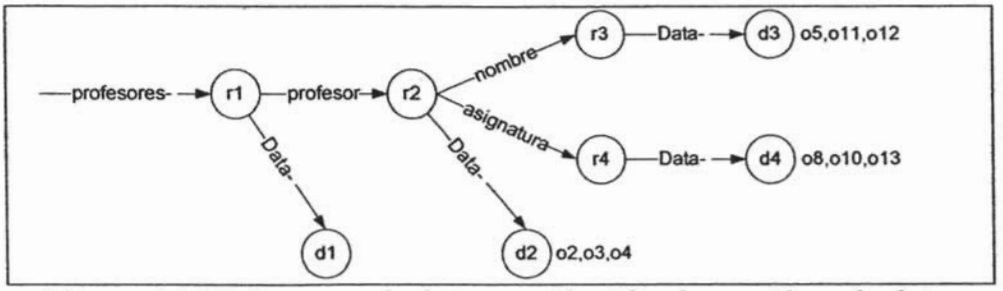


Figura 4.1.2.2 *Resumen de datos con identificadores utilizando datos primitivos para representar las ligas entre él y los datos*

La primera representación es la que utiliza el procesador de consultas de esta tesis, debido a que de este modo será más fácil actualizar datos y localizar la parte del *resumen de datos* que debe ser objeto de cambios y poder reflejarlos. De esta forma, al manejar los accesos como se hace con cualquier referencia (aristas), se estandariza la forma de manipular el *resumen de datos*. En cualquiera de los dos casos hay una disminución de accesos a disco en el recorrido del árbol al evaluar una expresión de camino; sin embargo, la última representación conduce a más problemas para mantener el *resumen de datos* consistente.

4.1.3 Optimización en búsqueda de información

Lo que se persigue al utilizar un *resumen de datos con identificadores* es reunir una mayor cantidad de información acerca de la base de datos. De esta forma, además de poder aprovecharlo para discriminar expresiones de camino que no devolverán ningún dato, también es útil para ayudar al sistema administrador de bases de datos semiestructurados a recuperar los identificadores requeridos por dichas expresiones de camino. Así, no hay necesidad de recorrer el árbol de datos de forma simple como la mostrada en el capítulo 2.

Cuando el sistema administrador de bases de batos semiestructurados recibe una expresión de camino, debe recorrer el árbol siguiendo el camino de etiquetas marcado por dicha expresión. Por la naturaleza intrínseca de los datos semiestructurados, no hay garantía de que todas las ramificaciones lleguen al final del camino de etiquetas marcado por la expresión. Por otro lado, con cada avance en profundidad en el árbol de datos, es necesario un acceso a disco para recuperar los

identificadores de los nodos hijo correspondientes a la etiqueta actual de la expresión de camino en evaluación. Esto implica que puede darse el caso de tener que avanzar en profundidad varias veces antes de concluir que, en la ramificación actual, no existen datos que cumplan con el camino de etiquetas, dando como resultado un uso de recursos innecesario.

Para ejemplificar lo anterior, se presenta una situación donde puede apreciarse con mayor claridad el gasto innecesario de accesos a disco por parte del sistema administrador de bases de datos semiestructurados cuando no se emplea el *resumen de datos con identificadores*.

Ejemplo 4.1.3.1

```
SELECT Mice : X
FROM   Inventario.Edificio.Piso.EquipoCómputo.Desktop.Mouse
       AS X
```

Aplicando esta consulta en la base de datos de la figura 4.1.3.1 —que describe el registro de los números de serie del equipo en inventario dentro de cada edificio de Ciudad Universitaria— la evaluación de la consulta se centra en recuperar los identificadores de aquellos datos semiestructurados que se obtengan al evaluar la expresión de camino *Inventario.Edificio.Piso.EquipoCómputo.Desktop.Mou-se*.

La base de datos de la figura 4.1.3.1 sólo muestra una pequeña parte de la información. Lo importante aquí es distinguir la profundidad en la cual están los datos requeridos por la consulta y que no todas las computadoras de escritorio tienen el registro del número de serie del mouse.

Esta consulta puede ser muy común cuando se piensa en generar reportes sobre una base de datos de inventario, pues lo que puede obtenerse es la cantidad de "ratones" o una lista de números de serie sólo de dicho componente. La respuesta con la que el sistema administrador de bases de datos semiestructurados devuelva los datos, en este caso, es un punto significativo.

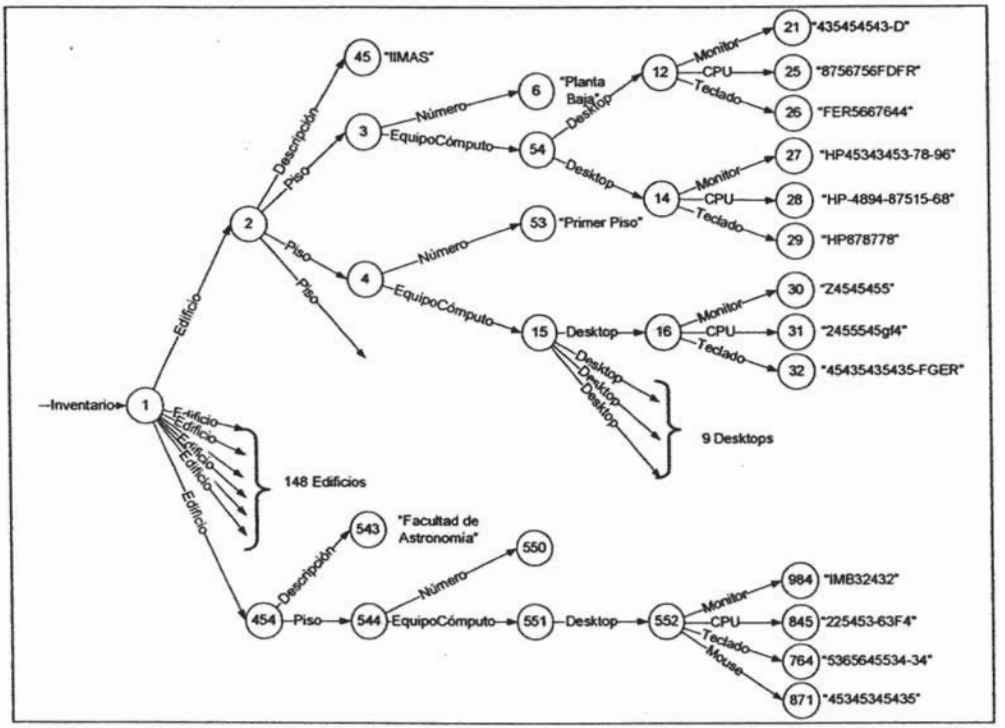


Figura 4.1.3.1 *Base de datos que muestra el registro de inventario del equipo de cómputo existente en Ciudad Universitaria*

En este ejemplo pasa exactamente lo que ya se ha expresado. Antes de que el sistema administrador de bases de datos semiestructurados pueda descartar una ramificación, debe atravesar en profundidad el árbol a un número considerable de niveles, en este caso *Inventario.Edificio.Piso.EquipoCómputo.Desktop*, por lo que son necesarios 6 accesos a disco para poder recuperar los datos del "ratón". Si se considera que existen 150 edificios, con 3 pisos cada uno y en promedio 10 computadoras de escritorio en cada piso, se obtiene un total de 4500 equipos de cómputo. La recuperación de los datos se realiza de la siguiente forma:

- 1 acceso para recuperar el identificador de la Ssd-tabla "Inventario".
- 1 acceso para recuperar los edificios (150 edificios)
- 150 accesos para recuperar los pisos (3 pisos por edificio = 450 pisos)

- 450 accesos para recuperar los Equipos de Cómputo (1 enlace "Equipo de Cómputo por piso = 450)
- 450 accesos para recuperar los Desktop (10 desktop por cada piso = 4500)
- 4500 accesos para recuperar los mouse de cada desktop.

Total: 5552 accesos.

En el supuesto caso de que sólo existieran 300 "ratones" registrados y a lo más uno por Desktop, 4200 de los 4500 accesos del último punto producirían una colección vacía. Por ser el punto donde más accesos son necesarios para encontrar los datos, con sólo considerar esto último se tiene un desperdicio del 93 % del total de accesos a disco.

La recuperación de estos identificadores puede acelerarse utilizando un *resumen de datos con identificadores* sobre la Ssd-tabla "*Inventario*". La optimización se describe de la siguiente forma:

Optimización II
Para toda *expresión de camino* $l_1.l_2. \ldots .l_n$, donde l_1 es una etiqueta con el nombre de una Ssd-tabla contenida en la base de datos que tiene disponible, al momento de evaluarse la expresión, un *resumen de datos con identificadores*, la colección de identificadores de los datos semiestructurados (si los hay) devuelta por la expresión de camino, se recupera recorriendo el *resumen de datos con identificadores*.

Con esta técnica el número de accesos a disco necesarios para evaluar una expresión de camino de la forma $l_1.l_2. \ldots . l_n$, es igual a su longitud $n + 1$.

Ejemplo 4.1.3.2 Evaluar la consulta (del ejemplo 4.1.3.1) pero ahora utilizando la técnica de optimización II.

```
      SELECT Mouses : X
      FROM
Inventario.Edificio.Piso.EquipoCómputo.Desktop.Mouse
            AS X
```

Suponiendo que ahora se cuenta con el *resumen de datos con identificadores* de la figura 4.1.3.2 para la base de datos de la figura 4.1.3.1 (la colección de identificadores por cada nodo no está completa debido a la cantidad de datos contenidos en la Ssd-tabla), para recuperar

los identificadores requeridos por la expresión de camino *Inventario.Edificio.Piso.EquipoCómputo.Desktop.Mouse* se debe recorrer el camino descrito por la expresión en el *resumen de datos con identificadores*. En este ejemplo, se llegará al nodo "r11". De ahí, sólo será suficiente recuperar el dato "Data-" de dicho nodo "r11" para obtener la colección de identificadores deseada. Los accesos a disco requeridos por este ejemplo son en total 7:

- 6 para recorrer el resumen de datos en profundidad siguiendo el camino descrito por la expresión y
- 1 para recuperar el dato semiestructurado primitivo que contiene la colección de identificadores deseada.

La reducción en el número de accesos es de 5552 – 7 = 5545.

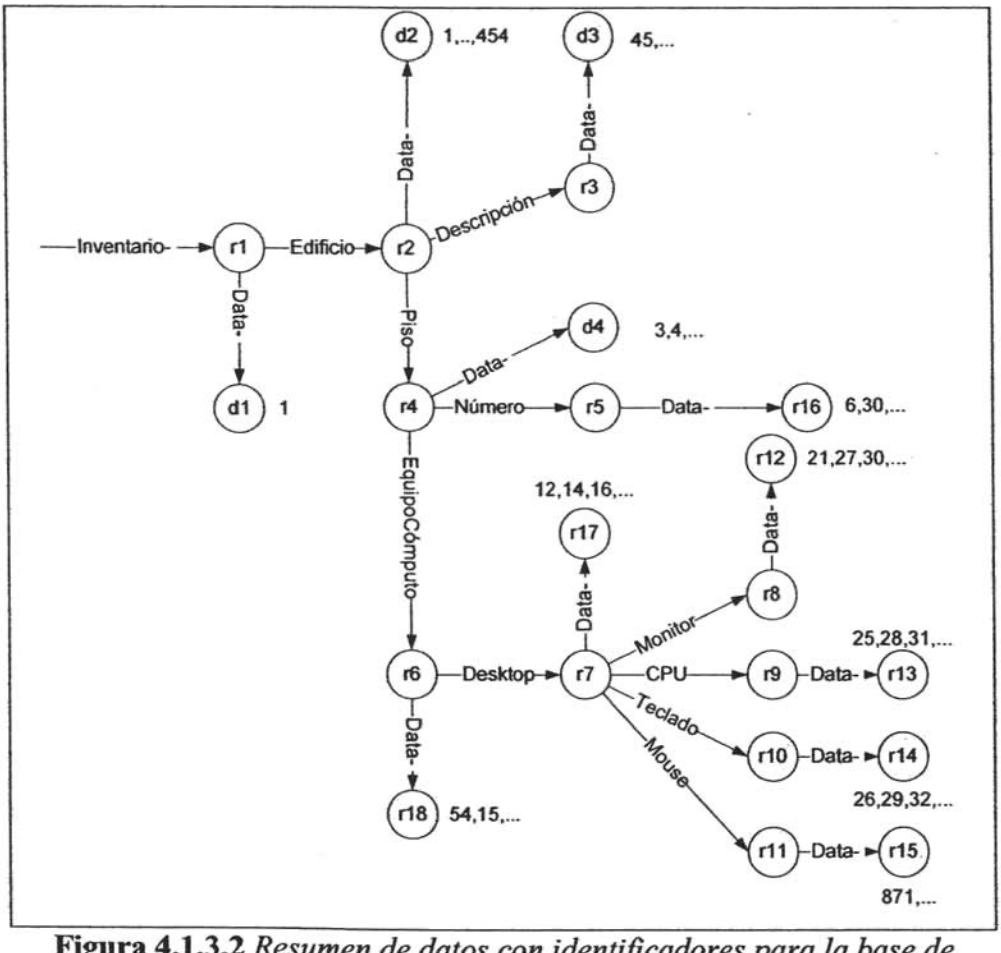


Figura 4.1.3.2 *Resumen de datos con identificadores para la base de datos de la figura 4.1.3.1*

Por lo tanto, para aplicar esta técnica de optimización, el optimizador tiene que analizar cada expresión de camino de la forma $l_1.l_2.$ ... $.l_n$. Si l_1 representa una Ssd-tabla no temporal, debe cambiar la expresión a l_1-$.l_2.$ ... $.l_n.Data$-

4.1.4 Expansión y búsqueda de información

Como se mencionó en la sección 4.1.1, es posible expandir las expresiones de camino extendidas utilizando un *resumen de datos*, siguiendo únicamente las rutas que lleguen al fin del camino. Lo anterior implica realizar la búsqueda de tales rutas en el *resumen de datos* y después aplicar la expresión de camino expandida sobre la base de datos en sí. De esta forma, en algunas ocasiones será necesario recorrer el mismo camino dos veces antes de poder recuperar los datos (identificadores de los datos semiestructurados) cuando sólo existe una ruta definida para una determinada expresión de camino extendida. Esto último ocurre cuando la base de datos no contiene ramificaciones o cuando ésta y el *resumen de datos* contienen la misma estructura definida para el camino a recorrer por la expresión de camino extendida, es decir, una parte de la base de datos no contiene estructura repetitiva que pueda resumirse. En estos casos en lugar de reducir el número de accesos a disco, éstos aumentan.

Con la posibilidad que abre el *resumen de datos con identificadores* de recuperar la información desde él sin necesidad de consultar directamente la Ssd-tabla, los identificadores requeridos por una expresión de camino extendida pueden recuperarse al recorrer el *resumen de datos con identificadores* en busca de expandir la expresión. La técnica de expansión de expresiones de camino se realiza en tiempo de compilación, es decir, antes de ejecutar la consulta en sí, se buscan las rutas válidas para la expresión. Tomando en cuenta que el *resumen de datos* contiene la información de los identificadores alcanzados por un camino determinado, la búsqueda de rutas válidas es posible realizarla en tiempo de ejecución y así la información se busca y recupera con un sólo recorrido de árbol (el del *resumen de datos con identificadores*).

4.1.5 Construcción del resumen de datos

La construcción de un *resumen de datos* es similar a transformar un Autómata Finito No Determinístico (AFND) a uno Determinístico (AFD), es decir, convertir un autómata donde en cada estado existe una o más transiciones con un sólo caracter, a uno donde en cada estado sólo existe una transición con un caracter. Existe un algoritmo muy conocido para realizar la conversión de un AFND a un AFD. Éste se basa en crear nuevos estados que representen al conjunto de estados alcanzados desde un estado del autómata original con un caracter, siempre y cuando todavía no exista tal representación.

Para construir el *resumen de datos* sucede algo parecido. Se toma como estado a cada nodo del árbol de datos y como transición a cada arista con su etiqueta, considerando a esta última como el caracter. El algoritmo desarrollado en este trabajo se basa en el descrito en [GoWi97] y se presenta en la siguiente figura 4.1.5.1.

Según lo mencionado en [GoWi97], el desempeño de este algoritmo puede requerir de un tiempo de ejecución exponencial y aún así, posiblemente, crear un *DataGuide* (para su caso) exponencialmente más largo que los datos origen. Sin embargo, también mencionan que para crear un *DataGuide* a partir de un grafo de datos que tenga estructura de árbol (sin ciclos), esta conversión siempre correrá en tiempo lineal. Dado que un algoritmo que corre en tiempo exponencial no es algo deseable para cualquier sistema de cómputo, es necesario encontrar una forma de mejorarlo.

El problema reside en que por cada colección de nodos alcanzados debe buscarse que no se haya creado dicho nodo anteriormente. Si se realiza esta búsqueda secuencialmente con un vector, en el peor de los casos (donde el AFND "tabla fuente" ya es un AFD "*resumen de datos*") se tendría que recorrer todo el vector en busca de una colección que no existirá. Por ello, mejorar esta búsqueda se convierte en una necesidad.

En dirección a este propósito, al crear un índice sobre la colección de nodos se incrementará la velocidad de búsqueda, reduciendo el tiempo de ejecución del algoritmo. Mediante el uso de una tabla hash para este fin, es posible accesar inmediatamente al registro que contiene la información de qué nodo representa la colección de nodos buscado.

```
// Entrada : o, el identificador de la Ssd-Tabla a la cual
se le construirá su RS
targetHash = tabla global vacía hash, mapeo de colecciones
de nodos origen con representaciones en el RS

CrearRS(o){
      rs = nuevoNodo()
      targetHash.agregar({o}, rs)
      CreaciónRecursiva({o}, rs)
}
CreaciónRecursiva(ssd, rs){
      ssdHijos = colección <etiqueta, id> de todos los hijos
      alcanzables desde ssd
      Para cada etiqueta l diferente en ssdHijos
            lHijos = colección de ids en ssdHijos que son
            alcanzados con la etiqueta l
            nodoRS = tarjetas.buscar(lHijos)
            Si nodoRS ¡= nulo
                  CrearArista(rs,l,nodoRS)
            Si no
                  nodoRS = nuevoNodo()
                  targetHash.agregar(lHijos, nodoRS)
                  CrearArista(rs.l.nodoRS)
                  CreaciónRecursiva(lHijos, NodoRS)
}
```

Figura 4.1.5.1 *Algoritmo para la creación de un resumen de datos sin identificadores*

El algoritmo de la figura 4.1.5.1 aún requiere de modificaciones para crear un *resumen de datos con identificadores*. Aquí es donde pueden considerarse las dos opciones antes presentadas para el enlace de los nodos del *resumen de datos* con los nodos de la Ssd-tabla de donde se obtuvo. Por un lado se cuenta con la información recopilada por la tabla hash, donde se tiene la relación <colección de nodos Ssd-tabla – nodo del *resumen de datos*>. Sin embargo, para utilizar esta información habría que modificar el campo índice y en lugar de ser la colección de nodos debería de ser el nodo que lo representa. De esta forma, al buscar un nodo del *resumen de datos* se tendrá la información de cuáles nodos son los alcanzables desde dicho punto. Esta modificación tendría que realizarse construyendo una nueva tabla hash, debido a que la original se guarda para que sea persistente y ocuparla en el mantenimiento (actualización) del *resumen de datos*, detalle que se mostrará más adelante. En consecuencia, para utilizar esta tabla hash en el enlace "Data-" del *resumen de datos*, es necesario que también sea persistente; lo que requeriría aún más espacio de almacenamiento para cubrir con tales requisitos. Pero, no sólo se

demandan más recursos físicos, sino que al recuperar los identificadores a partir del *resumen de datos*, habría que considerar otra estructura, la de la tabla hash que contiene dichos identificadores. Esta última no podría mantenerse en memoria, pues su tamaño puede ser considerablemente grande, ya que está relacionado con el tamaño del *resumen de datos*. Por lo tanto, cada vez que sea necesario obtener tales identificadores, se tendría que cargar la tabla en memoria, haciendo que el proceso sea más lento. Existe la posibilidad de construirse un sistema completo de acceso a la tabla como el realizado en las bases de datos, pero al contar con dicho sistema (el almacenamiento primitivo), éste permite utilizar en vez de una tabla extra, el mismo *resumen de datos* con ligas (aristas etiquetadas como "*Data-*") a los nodos que representa.

```
// Entrada : o, el identificador de la Ssd-tabla a la cual
se le construirá su RS
targetHash = tabla global vacía hash, mapeo de colecciones
de nodos origen con representaciones en el RS

CrearRS(o){
      rs = nuevoNodo()
      targetHash.agregar({o}, rs)
      CrearArista(rs,"Data-",o)
      CreaciónRecursiva({o}, rs)
      Guardar(targetHash)
}
CreaciónRecursiva(ssd, rs){
      ssdHijos = colección <etiqueta, id> de todos los hijos
      alcanzables desde ssd
      Para cada etiqueta l diferente en ssdHijos
            lHijos = colección de ids en ssdHijos que se
            alcanzan con la etiqueta l
            nodoRS = tarjetas.buscar(lHijos)
            Si nodoRS ¡= nulo
                  CrearArista(rs,l,nodoRS)
            Sino
                  nodoRS = nuevoNodo()
                  targetHash.agregar(lHijos, nodoRS)
                  CrearArista(rs.l.nodoRS)
                  Por cada hijo h en lHijos
                        CrearArista(nodoRS,"Data-",h)
                  CreaciónRecursiva(lHijos, NodoRS)
}
```

Figura 4.1.5.2 *Algoritmo para la creación de un resumen de datos con identificadores*

Considerando todo lo anterior, el algoritmo de creación del *resumen de datos con identificadores* final se muestra en la figura 4.1.5.2.

Este algoritmo se utiliza sólo después del proceso de creación de una Ssd-tabla y hará que sea más lento, lo que no es gravoso considerando que la creación de Ssd-tablas no es un proceso tan común como lo es la consulta de datos, en la cual recae el objetivo de la optimización.

4.1.6 Mantenimiento del resumen de datos

Al realizar una modificación a los datos de alguna Ssd-tabla, es necesario que su *resumen de datos* refleje tales cambios. Una forma de actualizar el *resumen de datos* es volver a crearlo desde cero, algo que resulta excesivo, pues generalmente no se modifica la Ssd-tabla en su totalidad, sino sólo una parte de ella. Por lo anterior, el *resumen de datos* debe modificarse sólo en aquella parte que haya cambiado.

En los casos donde una consulta modifique en su totalidad a una Ssd-tabla, el punto de actualización será la raíz, por ejemplo:

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

Esta consulta marca como punto de actualización la raíz de la Ssd-tabla "*profesores*", lo que lleva a verificar en su totalidad el *resumen de datos* "*profesores-*".

A partir de un punto de actualización, se busca qué nodo del *resumen de datos* es el que lo representa. Sin embargo, debido a que un *resumen de datos* abarca todo dato alcanzable desde la raíz de una Ssd-tabla, existe la posibilidad de que un mismo nodo tenga un enlace desde varios *resúmenes de datos*; por ejemplo, cuando una Ssd-tabla hace referencia a un dato contenido en otra. En tal caso, es necesario recuperar todos los nodos que pertenezcan a algún *resumen de datos* para actualizar este último desde el punto marcado por el nodo.

El algoritmo de actualización del *resumen de datos* se basa en el de creación, con la diferencia de que su inicio es en un punto diferente a la raíz. Además, en esta situación la tabla hash "targetHash" ya contiene la información de la equivalencia entre los nodos del resumen y las colecciones de nodos de la Ssd-tabla origen. Dicho algoritmo se presenta en la figura 4.1.6.1.

```
// Entrada : o, el identificador del nodo actualizado
targetHash = tabla hash, mapeo de colecciones de nodos
origen con representaciones en el RS

ActualizarRS(o){
      Para cada nodo rs que tenga una arista "Data-" al nodo
      o
            targetHash = Recuperar la tabla hash del resumen
            de datos que contiene a rs
            alcanzables = colección de nodos alcanzados
            desde rs con aristas "Data-"
            CreaciónRecursiva(alcanzables, rs)
            Guardar(targetHash)
}
CreaciónRecursiva(ssd, rs){
      ssdHijos = colección <etiqueta, id> de todos los hijos
      alcanzables desde ssd
      Para cada etiqueta l diferente en ssdHijos
            lHijos = colección de ids en ssdHijos que se
            alcanzan con la etiqueta l
            nodoRS = tarjetas.buscar(lHijos)
            Si nodoRS ¡= nulo
                  Si no ya existe una arista desde rs a
                  nodoRS con etiqueta l
                        Si rs tiene una arista que sale con
                        etiqueta l, borrarla
                        CrearArista(rs,l,nodoRS)
            Si no
                  nodoRS = nuevoNodo()
                  targetHash.agregar(lHijos, nodoRS)
                  Si rs tiene una arista que sale con
                  etiqueta l, borrarla
                  CrearArista(rs.l.nodoRS)
                  Por cada hijo h en lHijos
                        CrearArista(nodoRS,"Data-",h)
                  CreaciónRecursiva(lHijos, NodoRS)
      Eliminar toda arista que sale de rs cuya etiqueta no
      se encuentre en ssdHijos
}
```

Figura 4.1.6.1 *Algoritmo de actualización del resumen de datos con identificadores*

Este algoritmo es similar al presentado en [GoWi97], con la diferencia de que en este último se maneja una mayor cantidad de tablas hash, una para la relación conjunto de nodos origen – nodo *DataGuide* que los representa, otra para una relación más específica que la primera, nodo origen – nodo *DataGuide* y una más que representa la primera pero en sentido contrario.

El algoritmo propuesto aquí, elimina la necesidad de tantas tablas hash por el uso de las ligas "*Data-*", con lo que no sólo se salva el almacenar dichas tablas, sino también el manejo de las mismas (recuperarlas y guardarlas).

4.2 Optimización Pipelining

Aunque esta técnica de optimización recae en la forma de ejecución de la consulta (plan físico), se presenta aquí por ser parte de las optimizaciones realizadas en el procesador de consultas.

La mayoría de los operadores lógicos descritos utilizan una colección de datos semiestructurados como su entrada. Cada uno de ellos se maneja como si fuera parte de la base de datos, lo que se conoce como materialización. Sin embargo, es decisión del motor de ejecución grabarlos o mantenerlos en memoria, dependiendo de la cantidad disponible de ésta. Para una consulta que maneje una gran cantidad de información (por ejemplo que en su cláusula FROM se declaren varias tablas temporales y por lo tanto existan varios operadores *X* o *DJoin*), podría no haber suficiente memoria disponible para mantener todos los nodos temporales que se generen, sin la necesidad de grabarlos físicamente en la base de datos. Hacer esto último, generaría un mayor número de accesos a disco, haciendo la ejecución de la consulta más lenta.

Para aminorar el número de nodos temporales generados por la consulta, es posible implementar la técnica de optimización conocida como *Pipelining*. Ésta, describe que cada operador devuelve un sólo dato para que lo consuma el operador que lo recibe y procesa, después el operador regresa el siguiente dato. En este caso, el concepto operador hace referencia a un operador lógico; sin embargo, la técnica *Pipelining* no intenta cambiar la definición de éste, sino la forma en la que se ejecuta la operación que describe. Como se mostrará en el siguiente capítulo, un operador lógico se transforma en uno o más operadores físicos, los cuales ejecutan la operación descrita por el operador lógico. La técnica

Pipelining recae sobre los operadores físicos, pero es posible utilizar los operadores lógicos para fines de demostración de esta técnica.

En los sistemas administradores de bases de datos relacionales, para realizar un *Pipelining* dentro de la ejecución de la consulta es necesario definir un *iterador* para cada operador en el que se desee aplicar la técnica. Este *iterador* devuelve una tupla a la vez. Este trabajo define un operador físico, descrito en el siguiente capítulo, con el nombre `ForEach` el cual cumple con el trabajo de un *iterador*.

En el ambiente del lenguaje Ssquirel y el modelo de los datos semiestructurados, existe una característica intrínseca que facilita implementar la técnica *Pipelining*. Esta particularidad es la definición de un *ciclo* de la consulta. Por cada *ciclo*, todas las cláusulas (a excepción de FROM que es la que lo genera) se ejecutan con un solo conjunto de tablas temporales. De esta forma, se borran, seleccionan, actualizan o se revisa que cumplan una condición. Además, como puede observarse en el capítulo 3, los operadores que más nodos temporales generan, son precisamente los utilizados para representar la cláusula FROM, es decir, el producto cruz (*X*) y el *DJoin*. Por ello, aplicar la técnica de *Pipelining* a estos dos operadores ayudará a reducir considerablemente el uso de memoria para resultados temporales de la consulta.

Como se describió en el capítulo 3, cada nodo contenido dentro de la colección de nodos que recibe cada operador lógico, es un nodo que representa un *ciclo*. Los nodos se construyeron para agrupar las Ssd-tablas temporales y así poder tener repeticiones de las mismas. Sin embargo, al utilizar la técnica de *Pipelining* y sabiendo que sólo se evaluará uno de esos nodos "*ciclo*" a la vez, es innecesario construir un nodo temporal que agrupe a dichas Ssd-tablas temporales, pues en un momento dado de la ejecución, se sabrá que lo que debería estar en tal nodo temporal es el conjunto de Ssd-tablas temporales definidas en ese momento. Hay que recordar que las Ssd-tablas temporales definidas en la cláusula FROM, no son más que aristas que apuntan a un nodo en especifico (lo que devuelve la expresión de camino), donde su etiqueta es el nombre de la Ssd-tabla temporal. Por esto, el nodo temporal construido por el operador *Project*(π) también puede omitirse sin que afecte a la ejecución de la consulta.

Con todo lo anterior se ha eliminado la mayor parte de los nodos temporales utilizados al ejecutar una consulta. Sólo quedan algunas aristas

(las cuales también son en número reducido en comparación al número de ellas creadas sin *Pipelining*) que representan las Ssd-tablas temporales.

Ejemplo 4.2.1 Partiendo de la consulta utilizada para el ejemplo 3.4.2

```
SELECT      nombre : X
FROM        profesores.profesor as P,
            cursos.curso.profesor.nombre as C,
            P.nombre as X
WHERE       C = X
```

donde el número total de nodos temporales requerido para su ejecución es de 14, usando la técnica de *Pipelining* se construyen los siguientes componentes temporales:

- Para la cláusula FROM:
  - 3 aristas representando a las Ssd-tablas temporales. En todo *ciclo* siempre se utilizará la misma cantidad.
- Para la cláusula WHERE:
  - No se genera ningún componente temporal para evaluar su condición.
- Para la cláusula SELECT:
  - 1 nodo temporal para el resultado.
  - 1 arista temporal con etiqueta "nombre" que une al nodo del resultado con el nodo seleccionado (221).

Esta técnica libera al procesador de consultas de crear varios nodos temporales, los que llenan la memoria y reduce el trabajo del motor de ejecución al no tener que vaciar la memoria mediante la escritura a disco de tales nodos temporales.

4.3 Resumen y discusión

El elemento de la consulta en el que más se ha puesto énfasis en este capítulo es las expresiones de camino. Después de ejemplificar porqué son éstas las que más requieren de accesos a disco (si se evalúan sin ninguna optimización), se han propuesto métodos que ayudan a reducir la cantidad de recursos necesarios para evaluarlas.

Se introdujo el concepto de *resumen de datos*, el cual extrae información acerca de la estructura de los datos sin realizar repeticiones. En un principio, sólo era posible realizar una optimización a un grupo limitado de expresiones de camino, expandiendo la parte de ellas donde se hiciera uso de comodines u operadores de cerradura. Además, se detectaron ciertas desventajas al realizar la expansión en un ambiente multiusuario o cuando se utilizan bases de datos de estructura totalmente no homogénea. Por ello, se agregó al *resumen de datos* más información acerca de los datos. Con el propósito de mejorar la recuperación de datos por parte de una consulta, se ligó el *resumen de datos* con los datos en sí, lo que dió origen al *resumen de datos con identificadores*. De esta forma, cualquier expresión de camino que inicie con la etiqueta de una Ssd-tabla puede recuperar la información directamente sobre el *resumen de datos con identificadores*. Así, se ahorra tener que revisar cada ramificación en el grafo de datos en busca de los nodos de interés.

Durante el desarrollo de esta tesis, se observó también que hay un elevado número de componentes temporales que se generan al evaluar una consulta. Con el fin de reducir ésto, se definió cómo y en qué parte de la consulta implementar la técnica *Pipelining*. Con ella es posible reducir la creación de nodos temporales, sobre todo aquéllos que definen un *ciclo* de la consulta.

El objetivo de las técnicas de optimización propuestas en este trabajo es atacar aquellos puntos propios de los datos semiestructurados. Otras optimizaciones desarrolladas para otro tipo de datos, como *Push Selections Down*, pueden ser adaptadas para aplicarse en el plan lógico. Sin embargo, su implementación queda abierta para futuras contribuciones, en busca de concebir un sistema administrador de bases de datos semiestructurados robusto que pueda competir con los actuales sistemas comerciales.

Aún cuando las ideas utilizadas para definir el *resumen de datos* y el *resumen de datos con identificadores* se basaron en trabajos relacionados [GoWi97, McWi99, FeSu98], el aporte hecho aquí radica en la forma de ligar el resumen con la Ssd-tabla a la cual pertenece. En trabajos similares se utilizan otras estructuras de datos para almacenar las colecciones de identificadores de cada nodo del *resumen de datos*. Aquí, esta relación se definió utilizando el mismo modelo de datos semiestructurados. De esta forma, se ahorra espacio de almacenamiento así como la manipulación de otras estructuras ajenas al modelo. Otra ventaja de utilizar la ligadura entre los datos y el resumen es la simplificación de la actualización de este último, pues las mismas ligaduras señalan los puntos donde debe iniciar el reajuste.Después de haberse aplicado las técnicas para optimizar el plan lógico de consulta, es momento de obtener la serie de instrucciones primitivas que realizarán la tarea descrita por plan lógico. El siguiente capítulo describe este proceso, con el cual se termina el trabajo del procesador de consultas.

Capítulo 5

El plan físico de consulta

Semejante a lo ocurrido en el plan lógico de consulta, el físico es un anidamiento de funciones. Cada una ejecuta una operación básica, como crear un nodo, crear una arista, etc. las que, en conjunto, realizan la operación descrita por un operador lógico. Algunas de estas funciones son proporcionadas por el almacenamiento primitivo del sistema administrador de bases de datos semiestructurados. Otras son propias del motor de ejecución y se usan, principalmente, para generar nodos o aristas temporales.

Debido a que las funciones encontradas dentro de un plan físico serán las ejecutables y en razón a que el lenguaje en el que se desarrolló el procesador de consultas es Java, para cada función se define una clase. Los objetos creados por cada clase serán los encargados de ejecutar la consulta con ayuda del motor de ejecución.

En este capítulo se describirá cada una de las funciones del plan físico. Además, se presenta la equivalencia entre un operador lógico y la o las funciones físicas necesarias para ejecutarlo, es decir, cómo se efectúa la transformación de plan lógico a plan físico.

5.1 Funciones del plan físico

Cada función se implementó por una clase. Algunas son funciones primitivas y otras son propias del motor de ejecución. Cada instancia (objeto) de una clase representa una función en el plan físico de consulta. Por ello, se utilizan diagramas de clases para describir gráficamente dichas funciones. Existen tres tipos de clases:

- Las utilizadas para definir los datos (crear o borrar Ssd-tablas y vistas). Figura 5.1.1

- Las que representan funciones que devuelven una colección de datos semiestructurados. Figura 5.1.2
- Las que se utilizan para representar una condición. Figura 5.1.3

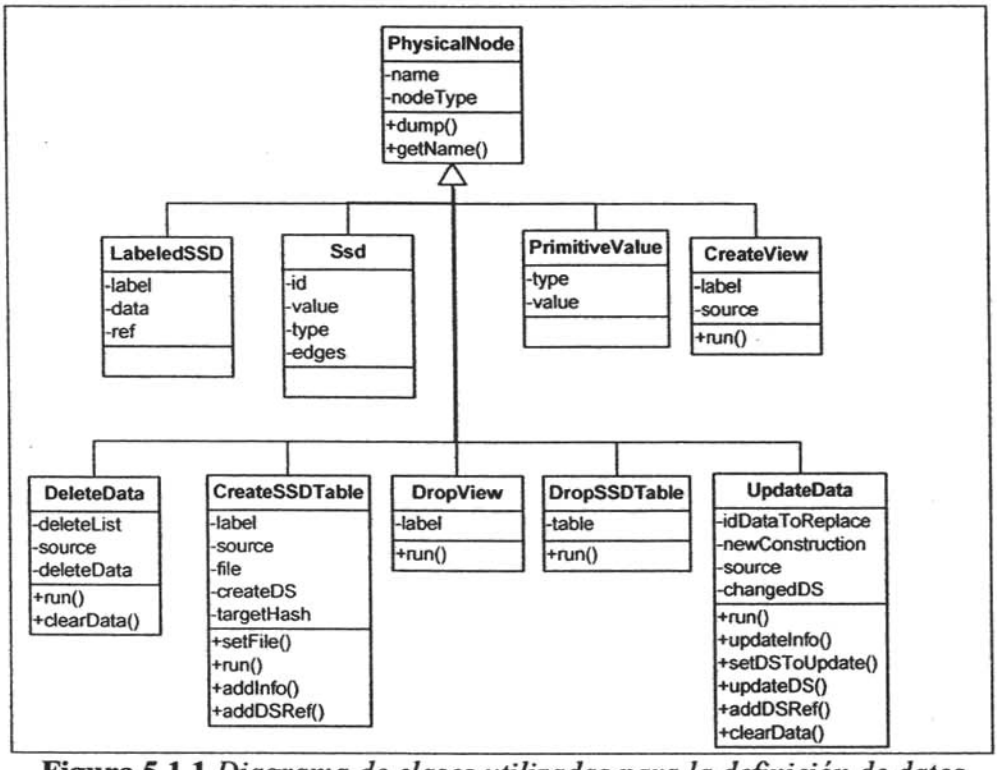


Figura 5.1.1 *Diagrama de clases utilizadas para la definición de datos*

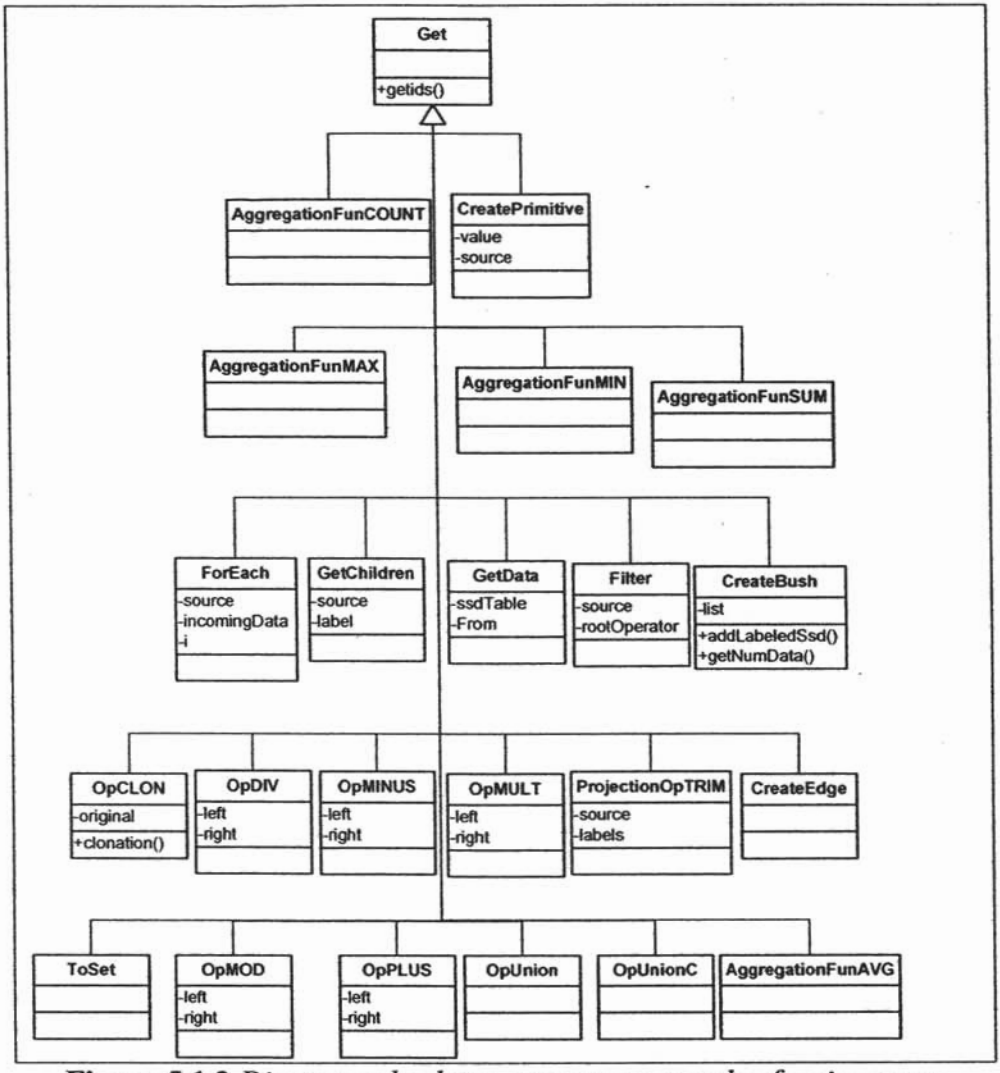


Figura 5.1.2 *Diagrama de clases que representan las funciones que devuelven una colección de datos semiestructurados*

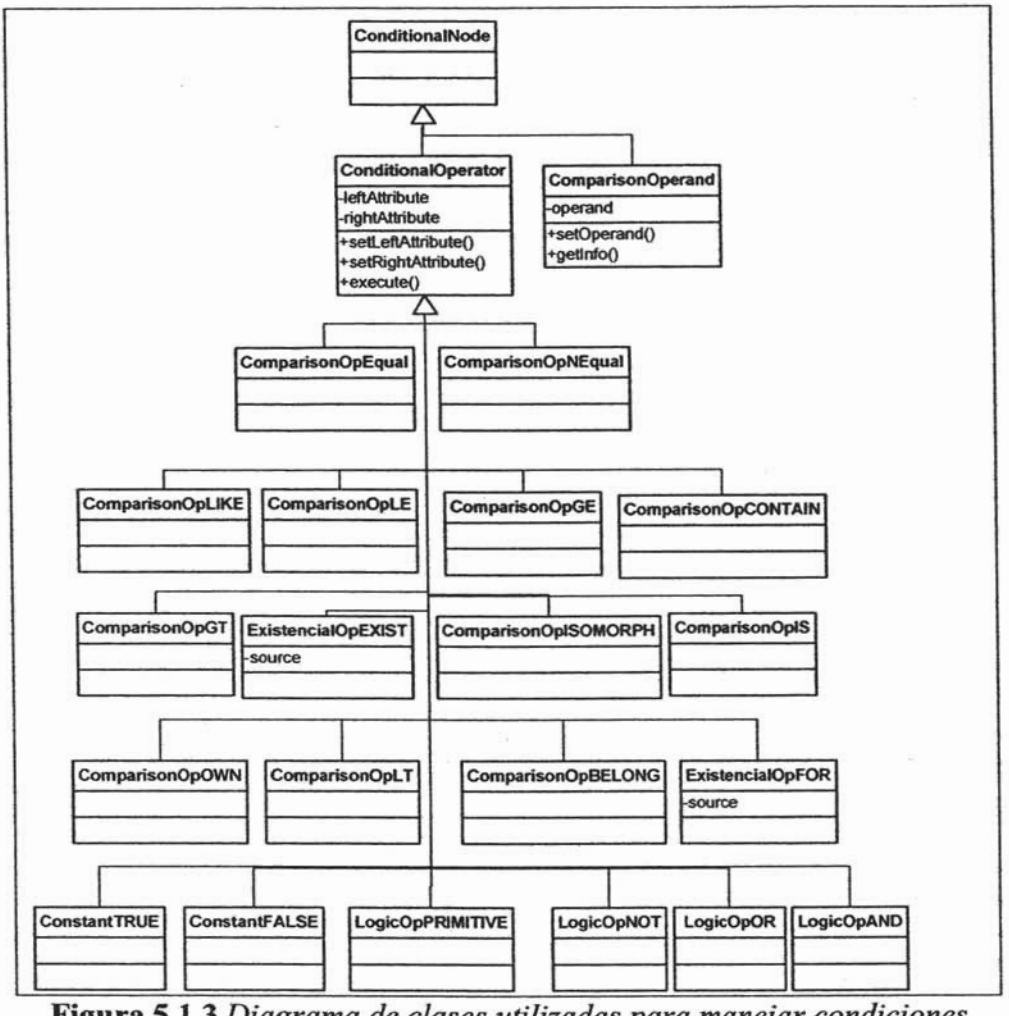


Figura 5.1.3 *Diagrama de clases utilizadas para manejar condiciones*

Los tres tipos de clases pertenecen al plan físico. Cada una realiza una función específica en él, por lo que todas heredan de la clase abstracta `PhysicalNode`. Son derivaciones directas de `PhysicalNode` las clases de la figura 5.1.1, pero las que se encuentran en las figuras 5.1.2 y 5.1.3 tienen una clase de la que heredan las demás. Esto facilita realizar la conexión de herencia. Ver figura 5.1.4.

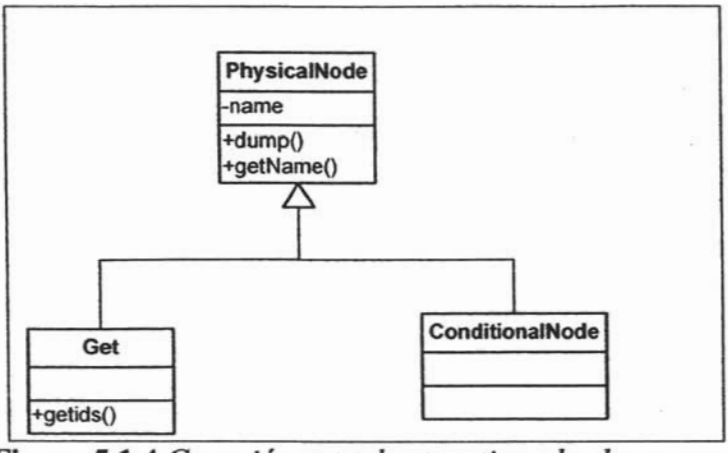


Figura 5.1.4 *Conexión entre los tres tipos de clases como pertenecientes al plan físico*

La clase `PhysicalNode` tiene tres características: un atributo `name` para guardar el nombre de la clase a la que pertenecerá cada objeto creado; un método `getName` para recuperar dicho nombre; un método abstracto `dump` para la impresión de cada objeto. Este último todas las clases deberán implementarlo según lo que cada una represente y/o contenga. Esta impresión da como resultado la presentación del árbol del plan físico de consulta.

El enlace entre las funciones físicas u objetos (instancias de las clases de plan físico) se lleva a cabo mediante la especificación de los atributos de cada clase. Habrá algunas que no necesiten enlazarse con otra porque no lo requieren para trabajar; estas funciones representan las hojas dentro del árbol del plan físico. A continuación se describen cada una de las clases del plan físico.

5.1.1 Clases para la definición de datos

Las clases descritas en esta sección se utilizan para representar datos semiestructurados o para crearlos. Las primeras se forman por:

- `Ssd`
- `PrimitiveValue`
- `LabeledSSD`

y se ocupan, fundamentalmente, dentro de la implementación de otras clases para tener una representación adecuada de los datos semiestructurados en el ambiente Orientado a Objetos. Las segundas se componen por:

- `CreateSSDTable`
- `CreateView`
- `DeleteData`
- `DropSSDTable`
- `DropView`
- `UpdateData`

y definen la implementación esencial para aquellas consultas de creación, actualización o borrado de datos.

5.1.1.1 Clase Ssd

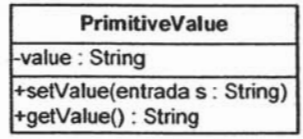
Ssd
-id : long -value : String -type : int -edges : Vector
+setId(entrada i : long) +setValue(entrada data : String) +addChild(entrada label : String, entrada child : Ssd) +getId() : long +getValue() : String +getType() : int +getChild(entrada i : int) : LabeledSSD +dump(entrada prefix : String)

Esta clase representa a un dato semiestructurado con su valor primitivo o la colección de todas las aristas que salen de él, según sea el caso. Sus atributos se describen como:

Atributo	*Descripción*
id	Identificador interno de la base de datos asignado a cada nodo
value	Valor primitivo
type	Tipo de nodo (0 – No primitivo, 1 – Entero, 2 – Real, 3 – Cadena de caracteres
edges	Es un arreglo dinámico que contiene un conjunto de objetos de tipo `LabeledSSD`, cada uno de los cuales representa una pareja <etiqueta>, <dato>

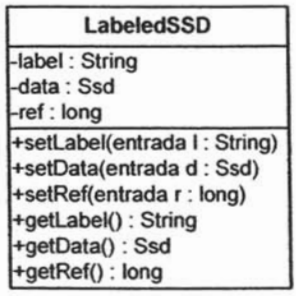
Cuenta también con varios métodos, algunos de ellos son utilizados para establecer y recuperar los atributos antes mencionados. Sin embargo, tratándose de aristas requiere de un proceso no tan directo como en los demás métodos. Para adicionar una arista se utiliza el método `addChild`, el cual requiere de la etiqueta de aquélla arista y el dato semiestructurado al que apuntará. Por ello, que es necesario otro objeto de la misma clase `Ssd`. Internamente el método `addChild` crea un nuevo objeto `LabeledSSD`, que encapsula la pareja <etiqueta>,<dato> y lo agrega al arreglo dinámico `edges`. Para recuperar una arista determinada simplemente se busca en el arreglo.

5.1.1.2 Clase PrimitiveValue



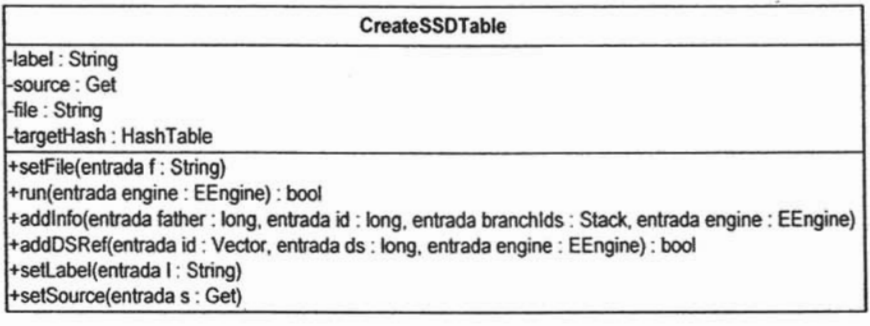
Esta clase sirve para almacenar el valor de cualquier constante definida dentro de una consulta.

5.1.1.3 Clase LabeledSSD



Con esta clase se realiza un encapsulamiento de la información que representa a una arista, es decir, una pareja <etiqueta>,<dato>. Existen dos formas de representar el <dato> destino: mediante la creación de un objeto `Ssd` que reúna las propiedades del dato semiestructurado destino; mediante el atributo `ref`. Hay ocasiones en donde no se tiene toda la información del dato destino y recuperarla requiere una búsqueda en la base de datos. Pero si, en un momento dado, tal información no es necesaria, entonces se usa el atributo `ref` el cual representa el dato destino únicamente con su identificador. Si después es necesario obtener toda la información de este dato, se puede lograr con dicho identificador.

5.1.1.4 Clase CreateSSDTable



Implementa la funcionalidad para crear una nueva Ssd-tabla en la base de datos. Al respecto, es necesario definir los datos que contendrá inicialmente. Éstos, pueden especificarse mediante una expresión similar a la de una consulta o con un archivo externo que debe ser un documento XML válido y bien formado. Para el caso de XML, cada elemento complejo será un dato semiestructurado y los caracteres de texto serán datos primitivos. Los atributos de los elementos del documento XML no son tomados en cuenta en esta implementación. Para los caracteres de

texto se crea un nuevo dato primitivo con ellos y se enlaza al dato al que pertenece mediante una arista con etiqueta del elemento en XML que contiene dicho texto. La sección 6.3 describe el manejo que se da a los elementos XML de contenido mixto.

La descripción de los atributos de esta clase es la siguiente:

Atributo	***Tipo***	***Descripción***
label	Cadena de caracteres	Es el nombre de la nueva Ssd-tabla a crear
source	Get	Obtiene la colección de datos semiestructurados que contendrá la nueva Ssd-tabla si éstos se definen mediante una construcción similar a la de una consulta
targetHash	HashTable	Es la tabla hash que utiliza el algoritmo de creación del resumen de datos.
file	String	Contiene la ruta del archivo XML de donde se obtendrán los datos que contendrá la nueva Ssd-tabla

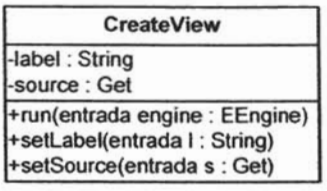
Esta clase contiene varios métodos, pero algunos de ellos sólo se usan para definir los atributos necesarios en la construcción de la Ssd-tabla (`setFile`, `setLabel`, `setSource`). Los más importantes son:

Método	***Descripción***
run	Inicia el proceso de creación de la nueva Ssd-tabla. Verifica que el nombre de la tabla no esté repetido y realiza la secuencia: - Crea un nuevo nodo que será la raíz para la nueva Ssd-tabla en la base de datos y conserva su identificador; utiliza para ello la función primitiva `createNonPrimitive`. - Verifica qué tipo de fuente se utilizará para crear sus datos. - Ingresa los datos al nodo recién creado mediante llamadas al método `addInfo`. - Cuando termina de ingresar los datos al nuevo nodo raíz se crea la nueva Ssd-tabla mediante la función primitiva `addSSDTable` - Creada la Ssd-tabla se elabora su resumen de datos, para lo cual se llama al método `addDSRef`

addInfo	Se trata de una función recursiva, que recibe el identificador del nodo que contendrá la información del segundo identificador recibido. Como es posible utilizar datos ya contenidos dentro de la base de datos, realiza enlaces a dichos datos (aristas que apuntan a los nodos ya creados); en cambio, si la información es nueva, entonces se crean nuevos nodos con las funciones primitivas `createPrimitive` o `createNonPrimitive`, según sea el caso y se crean las aristas correspondientes para unir el primer identificador con nuevos nodos con la función primitiva `add`.
addDSRef	Cuando la nueva Ssd-tabla contiene todos sus datos iniciales, crea su resumen de datos correspondiente. Esta función es la implementación del algoritmo de creación de un resumen de datos con identificadores mostrado en la figura 4.1.5.2.

Después de crear el *resumen de datos* debe guardarse el estado en que queda la tabla hash utilizada. Para ello, se emplea la serialización de objetos de Java, basándose en la guía de la referencia [Will02]. Afortunadamente la clase `HashTable` que proporciona el paquete `java.util` se implementó para guardarse en un archivo con la serialización, por lo que no es necesario realizar mayor trabajo que el especificado en la referencia anterior.

5.1.1.5 Clase CreateView



Para crear una vista materializada se utiliza la clase anterior `CreateSSDTable`, pero tratandose de una vista abstracta, se está frente a un problema más complejo. La referencia [AGM+97] hace mención a la alternativa utilizada en este trabajo, en la cual se materializa (crear una Ssd-tabla con el nombre de la vista que contenga sus datos) la vista abstracta cada vez que es demandada por alguna consulta. Por ello, tiene que guardarse aquella parte de la consulta que especifica los datos que

contendrá la vista para su posterior ejecución. Para realizarlo, podría guardarse esa parte de la consulta original y ejecutarla antes de ser utilizada por la consulta que la demanda para llenar la vista con datos actuales. Lo anterior requiere analizar léxica, sintáctica y semánticamente esa parte de la consulta, además de realizar la transformación a plan lógico y de éste a plan físico. Esto consumiría más tiempo al ejecutar una consulta que utilice una vista abstracta, pues tendrían que ejecutarse dos consultas completas en una: para crear la Ssd-tabla que representa la vista abstracta; para la consulta real.

A efecto de reducir el número de etapas necesarias para ejecutar la materialización de la vista abstracta, se guarda el plan físico de la parte que recupera los datos que contendrá la vista. De esta forma el plan se realiza directamente en el motor de ejecución sin necesidad de efectuar los análisis léxico, sintáctico y semántico ni las conversiones de la consulta para obtener el plan físico.

Para guardar el plan físico se utiliza nuevamente la serialización de objetos de Java; pero las clases de dicho plan no están listas para ello, pues para cada una que se desea serializar debe implementarse la interfaz de persistencia `java.io.Serializable`. Sin embargo, no hay la necesidad de adicionar otros métodos a la clase que implemente dicha interfaz, sólo hay que colocar la referencia en la declaración de la clase.

Como todas las clases del plan físico se derivan de una llamada `PhysicalNode`, implementar la interfaz es más sencillo adicionándola en la definición de dicha clase de la siguiente manera:

```
public abstract class PhysicalNode implements
                                java.io.Serializable{
```

Regresando a la descripción de la clase `CreateView`, en ella se reúne la información necesaria para crear la vista abstracta, es decir, su nombre y la definición de los datos que contendrá. Esta última parte es la que se almacena persistentemente.

Al ejecutarse el método `run`, se guardan los datos de la vista contenida en los atributos utilizando la función primitiva `addView`, la cual recibe como parámetros el nombre de la vista y su definición, justamente lo que contiene la clase `CreateView`. Por ello, `CreateView` representa a la función primitiva `addView`.

5.1.1.6 Clase DeleteData

DeleteData
-deleteList : Vector -deleteData : Get
+run(entrada engine : EEngine) +clearData(entrada id : long, entrada engine : EEngine) +setDelete(entrada d : Get)

Esta clase implementa la metodología de borrar descrita por el operador lógico *Delete* en uno de sus métodos. En general efectúa el proceso definido por las consultas de la forma DELETE-FROM-WHERE para borrar datos. Los atributos de esta clase representan:

Atributo	***Descripción***
deleteList	Es la colección de identificadores de los nodos a ser borrados de la base de datos.
deleteData	Recupera los identificadores a ser borrados de la base de datos.

Esta clase cuenta con dos métodos principales, `run` y `clearData`. El primero es el encargado de marcar la secuencia de borrado y el segundo tiene la función de borrar un identificador y todas sus referencias de la base de datos. La descripción de estos métodos es:

Método	***Descripción***
run	Se recuperan los identificadores del atributo `deleteData` a eliminar y se almacenan en el atributo `deleteList`. Cuando no hayan más ciclos por analizar, se eliminan cada uno de los identificadores contenidos en el atributo `deleteList` con el método `clearData`.
clearData	Implementación del operador lógico Delete. Es el encargado de dejar en estado consistente a la base de datos al borrar un nodo. Como quedó establecido en la sección 3.6.1, para borrar un dato antes deben eliminarse las aristas que apuntan a él, después toda su descendencia y por último el nodo en sí. Exactamente este proceso se realiza utilizando las funciones primitivas: - `get` para obtener la descendencia, - `remove` para borrar las aristas que entran y salen - `drop` para borrar el nodo en sí

El borrar datos no necesita que el *resumen de datos* de la Ssd-tabla modificada se actualice por separado, pues esta operación se realiza de forma directa con el algoritmo descrito en `clearData`. Debido a que el resumen contiene aristas que apuntan a los datos, al borrar las aristas que entran a un nodo, una o varias serán aristas provenientes de algún *resumen de datos*, dejando éste, por consiguiente, en estado actualizado después de ejecutar el proceso. Esta es una de las ventajas que ofrece la representación que se le dio al *resumen de datos*.

5.1.1.7 Clase DropSSDTable

DropSSDTable
-table : String
+run(entrada engine : EEngine) +setTable(entrada label : String) +dump(entrada prefix : String)

Representa la función primitiva `removeSSDTable`. Su funcionamiento es sencillo; con el método `setTable` se le especifica qué Ssd-tabla borrar de la base de datos y con el método `run` llama a `removeSSDTable` del almacenamiento primitivo. También se elimina su *resumen de datos* correspondiente.

Sólo resta mencionar que al borrar una Ssd-tabla no se eliminan los datos de otra a los que se hace referencia dentro de ella.

5.1.1.8 Clase DropView

DropView
-label : String
+run(entrada engine : EEngine) +setView(entrada view : String)

Se utiliza para borrar vistas abstractas y representa a la función primitiva `removeView`, a la cual sólo hay que especificarle el nombre de la vista a eliminar. Si ésta no existe regresa un error. Tratándose de vistas materializadas el borrado se realiza mediante la clase `DropSsd-tabla`.

Al borrar una vista abstracta no desaparece ningún dato de la base de datos, sólo el plan físico de ésta. Caso contrario sucede con las vistas materializadas que se borran usando DropSsd-tabla, donde sí desaparecen datos. Sólo se eliminan los datos contenidos dentro de la Ssd-tabla que

representa a la vista materializada, porque, como se describió anteriormente, cualquier arista que haga referencia a datos de otra Ssd-tabla no se borrarán.

5.1.1.9 Clase UpdateData

UpdateData
-idDataToReplace : Get -newConstruction : Get -changedDS : Vector
+run(entrada engine : EEngine) +updateInfo(entrada father : long, entrada id : long, entrada branchIds : Stack, entrada engine : EEngine) +updateDS(entrada id : long, entrada engine : EEngine) +addDSRef(entrada id : Vector, entrada ds : long, entrada targetHash : HashTable, entrada engine : EEngine) +clearData(entrada id : long, entrada engine : EEngine) +setSet(entrada set : Get) +setidToReplace(entrada id : Get)

Esta es la clase más compleja de todo el procesador de consultas, lo que resulta así por el proceso tan amplio que cubre. Hay que recordar que durante una actualización existen varios procesos:

- Recuperar los identificadores a actualizar (una consulta).
- Construir el nuevo dato que contendrá cada identificador (creación de una Ssd-tabla).
- Reemplazar el contenido de los identificadores a actualizar por el nuevo.
- Actualizar todos los *resúmenes de datos* afectados.
- Eliminar datos no alcanzables (borrado de datos)

La descripción de cada uno de los atributos de esta clase es:

Atributo	***Descripción***
idDataToReplace	Recupera el identificador del nodo a actualizar.
newConstruction	Proporciona el nuevo contenido de cada identificador a actualizar.
changedDS	Es una colección de nodos que pertenecen a algún resumen de datos que debe actualizarse para que refleje los cambios realizados. El método `updateDS` utiliza este atributo.

En esta clase sólo existen dos métodos para establecer los atributos mencionados (`setSet`, `setidToReplace`); los demás son propios para ejecutar la actualización. La descripción de éstos es:

Método	*Descripción*
run	El método principal de la clase. Controla la secuencia de ejecución. El algoritmo de ejecución es el siguiente: Recuperar el identificador del nodo a actualizar. Crear el nuevo contenido del nodo. Limpiar el atributo `changedDS`. Buscar los nodos de algún resumen de datos que deba actualizarse ejecutando el método `setDSToUpdate`. **Redireccionar todas las aristas que apuntan al nodo a actualizar al recién creado mediante el uso del método `updateInfo`.** Actualizar los nodos que pertenecen a algún resumen de datos usando el método `updateDS` Limpiar aquellos nodos que no son accesibles desde ninguna Ssd-tabla usando el método `clearData`. Lo marcado en negrita representa el proceso de actualización descrito en el capítulo 3. Las otras acciones, aparte de las de recuperación de los identificadores a utilizar en la actualización, son necesarias para el mantenimiento del resumen de datos, las cuales son varias, pero, como se mencionó anteriormente, se sacrifica el tiempo necesario para actualizar la base de datos con el fin de realizar las consultas más rápidamente. Esto es a causa de que la prioridad de este procesador de consultas es la recuperación de información.
updateInfo	Realiza la conexión con aristas entre el nodo a actualizar y su nueva información utilizando la función primitiva `add`.
setDSToUpdate	Como se mencionó en el capítulo 4, es posible obtener los nodos que apuntan al nodo a actualizar sin usar varias tablas hash como se utiliza en el algoritmo de actualización de DataGuides en [GoWi97]. Este método realiza tal operación de la forma siguiente: Recuperar todos los padres del nodo a actualizar

	utilizando la función primitiva `getParents`. Por cada padre, encontrar aquéllos que tienen una arista enlazada al nodo en cuestión con etiqueta "Data-" utilizando la función primitiva `get`. Por cada padre que cumpla con la condición anterior, guardar su identificador en el atributo changedDS, pues es un punto de actualización de algún resumen de datos.
updateDS	Implementación de la función "ActualizarRS" del algoritmo de la figura 4.1.6.1.
addDSRef	Implementación de la función "CreaciónRecursiva" del algoritmo de la figura 4.1.6.1.
clearData	Borra los nodos que no tienen acceso desde ninguna Ssd-tabla de la siguiente forma: Se borran todas las aristas que salen del nodo. Se borra en nodo en sí. No se borran los descendientes de dicho nodo pues desde alguna Ssd-tabla podrían haber acceso. Este es el último proceso que se ejecuta en la actualización, aunque no siempre es posible hacerlo y por ello queda "basura" o nodos no alcanzables dentro de la base de datos. Esto hace necesario un proceso de mantenimiento periódico de la misma.

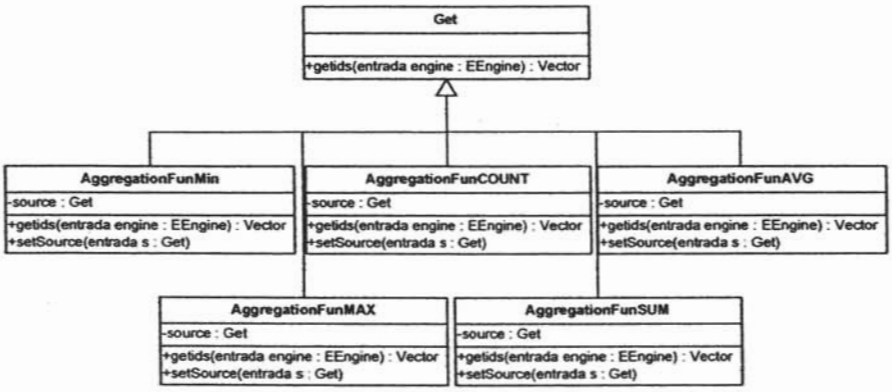
Aunque por estas últimas clases de definición de datos podría pensarse que cada una realiza varias operaciones, esto fue sólo el caso de las clases para eliminar y actualizar datos. Se presentó este tipo de clases (de definición de datos) antes que las demás porque las primeras (`Ssd`, `LabeledSSD`, `PrimitiveValue`) se utilizan para manejar los datos semiestructurados dentro de las clases siguientes.

5.1.2 Clases para las funciones que regresan datos semiestructurados

Este tipo de clases se basan en una abstracta llamada `Get` la cual define un método abstracto `getids`, mismo que sirve para recuperar la colección de datos semiestructurados. Sin embargo, esta clase no define cómo realizarlo, pues eso lo establecen las otras clases que heredan de ésta. Por tratarse de una clase abstracta, no puede ser instanciada y debido a ello no

representa ninguna función del plan físico. Sólo es definida para obligar a todas las clases de este apartado a describir la forma en que devolverán su colección de datos.

5.1.2.1 Clases que implementan las funciones de agregación

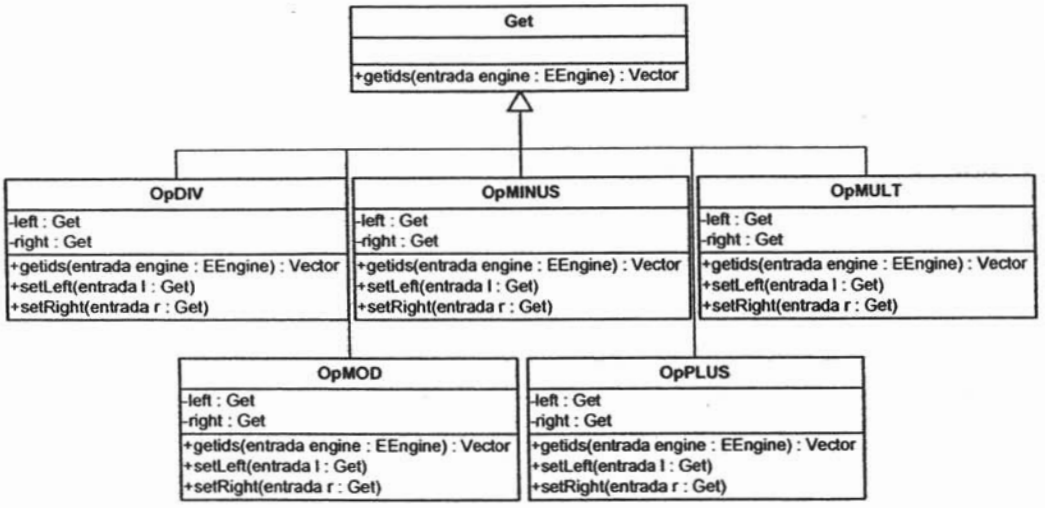


Estas clases se encargan de ejecutar las funciones de agregación definidas por el lenguaje. A cada una de ellas se le debe especificar de cual objeto `Get` recuperaran los datos a evaluar. El atributo `source` es el encargado de administrar los datos de entrada de la función, mismo que por diseño del lenguaje sólo es posible que devuelva un identificador a la vez.

Cada una de estas clases implementa el método `getids`, que devolverá el dato primitivo construido dentro del método que representa el resultado de la función. Todas las funciones, excepto AVG, se pueden aplicar si todos los hijos del nodo recuperado por `source` son primitivos y del mismo tipo (o puros números o puras cadenas de caracteres). Para AVG sólo es posible con números. Al momento de detectar un cambio de tipo primitivo o que un hijo no es primitivo, la evaluación se detiene y no regresa ningún identificador.

Para realizar cada función es necesario hacer uso de la primitiva `get`, que sirve para recuperar los hijos de un nodo y de la función `CreatePrimitive`, la cual crea el dato primitivo que contendrá el resultado y que residirá en memoria, si hay disponible.

5.1.2.2 Clases que implementan las operaciones aritméticas



Cada clase implementa el método `getids` para construir un dato primitivo que contendrá el resultado de la operación. Además, necesita que sus dos operandos (atributos `left` y `right`) devuelvan sólo un dato primitivo. El único operador que maneja cualquier tipo de dato primitivo es el `OpPLUS` (+), los otros sólo son válidos para números enteros o flotantes.

En estas clases es donde se realiza la conversión de tipo de manera automática cuando se evalúan tipos primitivos diferentes. Un número entero puede promoverse a número de punto flotante o a cadena de caracteres y un número de punto flotante puede promoverse a una cadena de caracteres.

Estas clases hacen uso de la función primitiva `CreatePrimitive` para la creación del resultado, el cual residirá en memoria, si hay disponible. En el casó contrario, el resultado se guarda en el medio de almacenamiento secundario. Esta decisión está en manos del motor de ejecución.

5.1.2.3 Clase Clon

OpCLON
-original : Get
+getids(entrada engine : EEngine) : Vector +clonation(entrada source : Vector, entrada branchIds : Stack, entrada engine : EEngine) : Vector +setOriginal(entrada s : Get)

Realiza la clonación de un dato semiestructurado. El dato original se recupera mediante el atributo `original`. El método `getids` regresa el identificador del nuevo dato clonado. La clonación se realiza con el método `clonation` el cual recibe una colección de identificadores a clonar y regresa la colección de los nuevos datos clonados. Cuando se llama por primera vez a dicho método, la colección a clonar sólo contiene un identificador, el recuperado por el atributo `original`.

Clonar un dato semiestructurado involucra a todo su contenido, es decir, toda su descendencia. Esto genera la posibilidad de caer en un ciclo. Por ello, se tiene un registro de los datos contenidos en una ramificación mediante el parámetro `branchIds`. Cada identificador a clonar se revisa si no se clonó anteriormente; si ya se efectuó esta acción, se construye tal ciclo con una arista al nodo clonado, es decir, el identificador de nodo clonado.

Esta clase utiliza la función primitiva `get` para recuperar los hijos de los nodos que se están clonando y `createNonPrimitive` y `createPrimitive` para su creación.

5.1.2.4 Clase CreatePrimitive

CreatePrimitive
-source : Get
+getids(entrada engine : EEngine) : Vector +setSource(entrada s : Get)

Representa la función primitiva del mismo nombre. Obtiene el valor primitivo a crear de su atributo `source`, con el que puede enlazar esta clase con otras tipo `get` para la creación del plan físico de consulta. El método `getids` regresa el identificador del nuevo dato primitivo.

5.1.2.5 Clase OpUNION

OpUNION
-data1 : Get -data2 : Get
+getids(entrada engine : EEngine) : Vector +setData1(entrada d : Get) +setData2(entrada d : Get)

Implementa el operador lógico unión de datos ($\cup$). Los atributos "data1" y "data2" proporcionan los identificadores de los datos semiestructurados que se unirán. El método `getids` devuelve el nuevo dato semiestructurado que contiene la unión de los datos. Esta clase utiliza la función primitiva `get` para recuperar los hijos de los datos semiestructurados y la función física `createResultData`, que se describe en el capítulo 6, para la creación del nuevo dato semiestructurado.

5.1.2.6 Clase OpUnionC

OpUnionC
-data1 : Get -data2 : Get -source : Get
+getids(entrada engine : EEngine) : Vector +setData1(entrada d : Get) +setData2(entrada d : Get) +setSource(entrada s : Get)

Implementa el operador lógico unión de colecciones ($\cup_c$). Los atributos "data1" y "data2" proporcionan los identificadores de los datos semiestructurados que se unirán en una sola colección. El método `getids` devuelve la nueva colección de datos semiestructurados que contiene la unión de las colecciones de entrada.

El atributo `source` se utiliza para aquellas ocasiones en las que es necesario ejecutar alguna definición (como Ssd-tablas temporales) antes de realizar la unión. Esto es especialmente útil cuando se trata de convertir un tipo especial de expresión de camino a plan físico de consulta. En la sección 5.2.1 se describe tal conversión.

5.1.2.6 Clase CreateEdge

CreateEdge
-label : String -from : Get
+getids(entrada engine : EEngine) : Vector +run(entrada engine : EEngine) +setLabel(entrada l : String) +setFrom(entrada value : Get)

Crea una arista temporal en la base de datos con etiqueta especificada en el atributo `label` y apunta al nodo recuperado por el atributo `from`. El método `getids` regresa la pareja <label>,<data> recién creada, encapsulada en un objeto tipo LabeledSSD.

Como se verá en el esquema de traducción de plan lógico a plan físico, esta clase juega un papel muy importante dentro de la consulta, pues realiza parte de la implementación del operador lógico π. Esta clase se desarrolla con el fin de utilizarse junto con la técnica *Pipelining*, para recuperar y devolver un identificador a la vez, por lo que se espera que el atributo `from` devuelva un solo dato en igual forma.

5.1.2.7 Clase ForEachDO

ForEachDO
-op : PhysicalNode -source : Get
+setOp(entrada operator : PhysicalNode) +setSource(entrada s : Get) +getids(entrada engine : EEngine) : Vector

Implementa el operador lógico *Map*, donde el atributo `source` proporciona la colección de datos semiestructurados. Por cada dato en dicha colección, se ejecuta el operador especificado en el atributo `op`, el cual, si es de tipo `Get`, `ForEachDo` devolverá una colección de datos que contendrá todos los datos recuperados por el operador `op` al mandar a llamar su método getids. De otra forma, se emplea el método `run` del operador y `ForEach` no devolverá ningún dato.

5.1.2.8 Clase ProjectionOpTRIM

ProjectionOpTRIM
-source : Get -labels : Vector
+getids(entrada engine : EEngine) : Vector +setSource(entrada s : Get) +addLabel(entrada l : String)

Implementa al operador lógico *Trim*. En el atributo `source` se especifica de dónde obtendrán la colección de datos semiestructurados. Esta clase se desarrollo para ejecutarse con la técnica *pipelinig*, por lo que se espera que el atributo `source` regrese un solo dato a la vez.

En el atributo `labels` se especifica cuales hijos no serán seleccionados del nodo recuperado por `source`. Los que sí se elijan serán devueltos por el método `getids` dentro de un nuevo nodo.

5.1.2.9 Clase GetData

GetData
-ssdTable : String -From : Get
+getids(entrada engine : EEngine) : Vector +setSsdTable(entrada t : String) +setScope(entrada scope : Get)

Recupera el identificador de una Ssd-tabla materializada o temporal. Para cualquiera de los dos casos se llama a una función del motor de ejecución, el cual determinará en qué lugar buscar la Ssd-tabla, en la base de datos o en la parte de la memoria reservada para elementos temporales. El nombre de la Ssd-tabla a recuperar es definido por el atributo `ssdTable`.

Similar a la clase `OpUnion`, `GetData` ofrece la posibilidad de ejecutar otro objeto (posiblemente para declarar tablas temporales) antes de tratar de recuperar el identificador de la tabla, si el atributo `from` es definido. Lo anterior permite la posibilidad de enlazarlo con un subárbol del plan físico de consulta.

5.1.2.10 Clase GetChildren

GetChildren
-source : Get -label : String
+getids(entrada engine : EEngine) : Vector +setSource(entrada s : Get) +setLabel(entrada l : String)

Representa a la función primitiva `get` en un plan físico de consulta. El atributo `source` especifica al dato semiestructurado del cual se obtendrán aquellos datos que están conectados a éste por medio de una arista con etiqueta igual a la que se especifica en el atributo `label`. Esta clase, al igual que varias de esta sección, está preparada para una ejecución *Pipelining*, por lo que se espera que el atributo `source` devuelva un sólo dato cada vez que ejecuta su método `getids`.

Esta clase, junto con `CreateEdge`, forman una parte importante dentro del plan físico de consulta, pues con ellas se realiza casi todo el trabajo de recuperación de identificadores a partir de una expresión de camino.

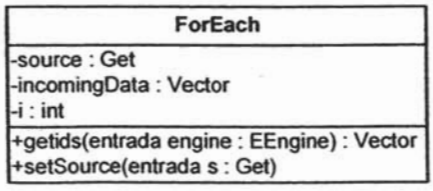
5.1.2.11 Clase CreateBush

CreateBush
-list : Vector
+getids(entrada engine : EEngine) : Vector +addLabeledSsd(entrada data : CreateLabeledSSD) +getNumData() : int

Implementa el proceso descrito por el operador lógico *NewSSD* descrito en el capítulo 3. Su función es unir un grupo de aristas, contenidas en el atributo `list`, en sólo un nodo. Regularmente el grupo de aristas se crea en tiempo de ejecución, por lo que cada elemento del atributo `list` es un objeto del tipo `Get`.

Esta clase es útil para realizar la operación de agrupación definida en [Garc02] y generalmente se usa para construir nuevos datos semiestructurados o el resultado de una consulta.

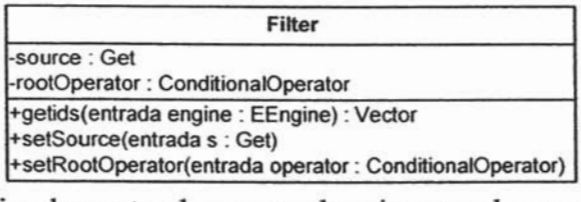
5.1.2.12 Clase ForEach



Con esta clase se implementa el iterador para todas las clases de esta sección. Recupera la colección de datos semiestructurados que devuelve cada clase con su método `getids`, el cual es un `Vector`, y regresa dato por dato. Cuando termine la colección llama nuevamente al método `getids` de la clase original. El objeto de alguna clase `get` se establece en el atributo `source`, el atributo `incomingData` guarda el resultado obtenido al llamar a su método `getids` y el atributo `i` guarda la posición del `Vector` que se ha regresado.

Con este tipo de iterador no es necesario escribir uno por cada clase de esta sección. Las clases de las otras secciones, al no devolver datos, no requieren de un iterador para realizar su ejecución tipo *Pipelining*.

5.1.2.13 Clase Filter



Esta clase implementa el proceso descrito por el operador lógico σ visto en el capítulo 3. Mediante el método `getids`, por cada dato recuperado por el atributo `source` se evalúa la condición; si ésta resulta afirmativa, el dato se devuelve. La condición se establece en el atributo `rootOperator` y es un objeto de tipo `ConditionalOperator`, que se describirá en la sección 5.1.3.

5.1.2.14 Clase ToSet

ToSet
-source : Get
+setSource(entrada s : Get) +getids(entrada engine : EEngine) : Vector +existance(entrada list : Vector, entrada data : LabeledSSD, entrada engine : EEngine) : bool +isomorph(entrada l : Vector, entrada r : Vector, entrada engine : EEngine) : bool

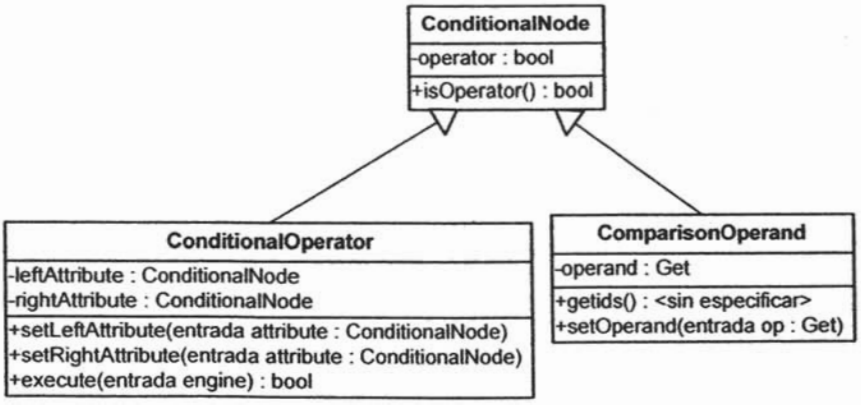
Esta clase implementa el proceso descrito por el operador lógico δ. Devuelve la colección de datos semiestructurados recuperada por el objeto especificado en el atributo `source`, transformada en conjunto, es decir, se eliminan aquellos datos repetidos.

Para realizar la discriminación de datos, se utiliza el método existance, el cual comprueba si no existe un dato dentro de un conjunto mediante el método isomorph. Este último implementa el algoritmo descrito en la figura 5.1.3.2.1 para revisar si dos datos son isomorfos.

5.1.3 Clases utilizadas para las condiciones

Las clases de esta sección son útiles para representar los operadores condicionales encontrados en la cláusula WHERE de la mayoría de las consultas. Sus operandos son objetos de tipo `get` descritos en la sección anterior, por lo que el valor de cada uno es obtenido ejecutando el método `getids`.

Para representar condiciones con un número indeterminado de operadores, se define la siguiente estructura de clases:

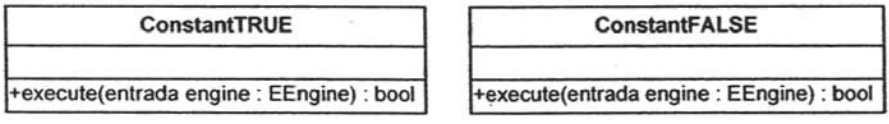


la cual describe que cualquier nodo encontrado dentro de una condición puede ser un operador (`ConditionalOperator`) o un operando (`ComparisonOperand`), de esta forma, los operandos de un operador pueden ser otro operador o bien un objeto que recupera datos.

Como pudo observarse en la sección anterior, la clase `Filter` fija a un objeto tipo `ConditionalOperator` como una condición y es el que ejecuta (método `execute`) y devuelve el resultado booleano. Fijar dicho objeto de tipo `ConditionalNode` podría generar el error de establecer un operando, el cual ni tiene el método de ejecución ni devuelve un resultado booleano.

Las clases descritas en esta sección tienen fundamento en la clase `ConditionalOperator` y deben implementar el método abstracto `execute`, que determina si la condición es falsa o verdadera.

5.1.3.1 Clases para las constantes TRUE y FALSE



Estas clases se ocupan para representar las constantes TRUE y FALSE en una condición. Su implementación es simple y el método `execute` siempre regresa verdadero o falso según sea el caso.

5.1.3.2 Clases para operadores que se aplican a datos semiestructurados

Su diagrama puede observarse en la figura 5.1.3. Las clases de esta sección sólo tiene validez cuando sus operandos son de tipo `get`, restricción que se lleva acabo mediante la gramática del lenguaje. En la tabla siguiente se describe a cada una de ellas:

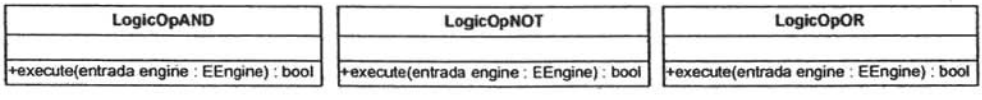
Clase	*Descripción*
ComparisonOpEqual	Realizan la comparación (=, <>, <=, >=, >, <) entre dos datos primitivos. Se utiliza la promoción de tipos de datos, como en el caso de las clases que manejan operadores aritméticos, cuando se compara a dos tipos diferentes de datos.
ComparisonOpNEqual	
ComparisonOpLE	
ComparisonOpGE	
ComparisonOpGT	
ComparisonOpLT	
ComparisonOpLIKE	Realiza la comparación entre un dato primitivo y una cadena de caracteres que puede contener comodines. Para la implementación de esta comparación se usó la clase proporcionada por Java, `String` y su método `matches` descrito en [SunM04].
ComparisonOpCONTAIN	Representa la función primitiva `contains`.
ComparisonOpOWN	Representa la función primitiva `owns`.
ComparisonOpBELONG	Implementa la función `Belong`, utilizando la función primitiva `contains` con los parámetros intercambiados.
ComparisonOpISOMORPH	Implementa el operador `Isomorph`, utilizando el algoritmo de la figura 5.1.3.2.1 para verificar que su estructura sea idéntica.
ComparisonOpIS	Implementa el operador `Is`. Regresa verdadero si ambos operandos tienen el mismo identificador.
ComparisonOpPRIMITIVE	Representa la función primitiva `isPrimitive`.

```
      // conjunto1 - Es un conjunto de tamaño N
      // conjunto2 - Es un conjunto de tamaño M
      // ISOMORPH - Función recursiva para determinar si dos
datos                    semiestructurados tienen estructura
idéntica.

ISOMORPH ( conjunto1, conjunto2) : bool
      Si N <> M
            Regresa falso
      Para i=0..N hacer
            e1 = conjunto1.elemento(i)
            e2 = conjunto2.elemento(i)
            Si isPrimitive(e1) XOR isPrimitive(e2)
                  Regresa falso
            Si isPrimitive(e1)
                  Si e1.valor <> e2.valor
                        Regresa falso
            Si no
                  C1 = conjunto de hijos de e1
                  C2 = conjunto de hijos de e2
                  Si no ISOMPORPH(C1, C2)
                        Regresa falso
      Regresa verdadero
```

Figura 5.1.3.2.1 *Algoritmo utilizado para comparar si dos datos semiestructurados son isomorfos*

5.1.3.3 Clases para operadores lógicos condicionales



Los operandos que utilizan estas clases deben ser del tipo `ConditionalOperator`, al contrario de las clases de la sección anterior. El método `execute` de cada clase ejecuta los operandos para obtener el resultado falso o verdadero. A partir de ellos se evalúa la condición AND, OR o NOT, según sea el caso; para el último, sólo se ocupa uno de los dos operandos de los que se dispone.

5.1.3.4 Clases para operadores existenciales

ExistencialOpEXIST
-source : Get
+execute(entrada engine : EEngine) : bool +setSource(entrada s : Get)

ExistencialOpFOR
-source : Get
+execute(entrada engine : EEngine) : bool +setSource(entrada s : Get)

Las clases `ExistencialOpEXIST` y `ExistencialOpFOR` implementan los operadores condicionales EXIST y FOR ALL, respectivamente. Su ejecución se realiza con el método `execute` y es de la siguiente forma: por cada dato obtenido por el atributo `source`, se evalúa la condición establecida en el atributo leftAttribute heredado de la clase `ConditionalOperator`.

Para la clase `ExistencialOpEXIST`, a la primera evaluación de la condición que resulte verdadera el método `execute` devuelve verdadero, pero si la condición simpre es falsa, entonces devuelve falso.

Para la clase `ExistencialOpFOR`, se realiza un proceso inverso; a la primera evaluación de la condición que resulte falsa, devuelve un resultado falso, pero si todas las evaluaciones resultan verdaderas, entonces devuelve un resultado verdadero.

5.2 Traducción de plan lógico a plan físico

Una vez descritas todas las funciones utilizadas en un plan físico, en esta sección se muestra cómo un plan lógico pasa a ser uno físico. Este cambio se realiza mediante equivalencias de un operador lógico a una o varias funciones físicas, lo que se conocerá como esquema de traducción de lógico a físico.

La tabla 5.2.1 muestra el esquema de traducción para la mayoría de los operadores lógicos a objetos de las clases de la sección anterior. La función TLF(x) (por Traducción de lógico a físico), que se utiliza dentro de la representación del plan físico en la tabla 5.2.1, equivale al plan físico del parámetro "x".

Para cada uno de los operadores condicionales se utiliza una clase particular. Por ello, la traducción de plan lógico a físico se limita a crear los objetos de tipo que marca el operador y establecer sus operandos.

La conversión de una expresión de camino a plan físico y la traducción de los operadores *X* y *DJoin*, son temas descritos en las secciones 5.2.1 y 5.2.2, respectivamente. Es en estos tres elementos del plan lógico donde se aplica el iterador `ForEach` definido anteriormente para la implementación de la técnica *Pipelining*, pues de ellos depende el número de *ciclos* de una consulta.

Operador lógico	***Plan físico***
$d_1 \cup d_2$	*OpUnion union* *union.setData1(TLF(d_1))* *union.setData2(TLF(d_2))*
NewSSD(D_1, D_2,D_3,...,D_n)	*CreateBush bush* *Para i = 1 .. n* *bush.addLabeledSsd(TLF(D_i))*
AVG(D)	*CreatePrimitive nuevoDato* *AggregationFunAVG op* *op.setSource(TLF(D))* *nuevoDato.setSource(op)*
MAX(D)	*CreatePrimitive nuevoDato* *AggregationFunMAX op* *op.setSource(TLF(D))* *nuevoDato.setSource(op)*
MIN(D)	*CreatePrimitive nuevoDato* *AggregationFunMIN op* *op.setSource(TLF(D))* *nuevoDato.setSource(op)*
SUM(D)	*CreatePrimitive nuevoDato* *AggregationFunSUM op* *op.setSource(TLF(D))* *nuevoDato.setSource(op)*
COUNT(D)	*CreatePrimitive nuevoDato* *AggregationFunCOUNT op* *op.setSource(TLF(D))* *nuevoDato.setSource(op)*
Clon(x)	*OpClon clon* *clon.setOriginal(TLF(x))*
$d_1 + d_2$	*CreatePrimitive nuevoDato*

	OpPLUS op *op.setLeft(TLF(d1))* *op.setRight(TLF(d2))* *nuevoDato.setSource(op)*
$d_1 - d_2$	*CreatePrimitive nuevoDato* *OpMINUS op* *op.setLeft(TLF(d1))* *op.setRight(TLF(d2))* *nuevoDato.setSource(op)*
$d_1 * d_2$	*CreatePrimitive nuevoDato* *OpMULT op* *op.setLeft(TLF(d1))* *op.setRight(TLF(d2))* *nuevoDato.setSource(op)*
d_1 / d_2	*CreatePrimitive nuevoDato* *OpDIV op* *op.setLeft(TLF(d1))* *op.setRight(TLF(d2))* *nuevoDato.setSource(op)*
d_1 *MOD* d_2	*CreatePrimitive nuevoDato* *OpMOD op* *op.setLeft(TLF(d1))* *op.setRight(TLF(d2))* *nuevoDato.setSource(op)*
π *(S,* λ*)*	*CreateEdge nuevaTabla* *nuevaTabla.setFrom(TLF(S))* *nuevaTabla.setLabel(*λ*)*
δ *(D)*	*ToSet d* *d.setSource(TLF(D))*
Trim(D, $\lambda_1,\lambda_2,..,\lambda_n$*)*	*ProjectionOpTRIM trim* *trim.setSource(TLF(D))* *Para i=1..n hacer* *trim.addLabel(*λ_i*)*
CreatePrimitive(X)	*CreatePrimitive primitive* *primitive.setValue(X)*
σ *(D, <condición>)*	*Filter filtro* *filtro.setSource(TLF(D))* *filtro.setRootOperator(TLF(<condición>))*
Map (D, O)	*ForEachDO map* *map.setSource(D)*

	map.setOp(O)
Delete(D)	*Delete borrar* *borrar.setDelete(D)*
Update(D, U)	*UpdateData actualizar* *actualizar.setSet(U)* *actualizar.setSource(D)*

Tabla 5.2.1 *Esquema de traducción de operador lógico a su plan físico*

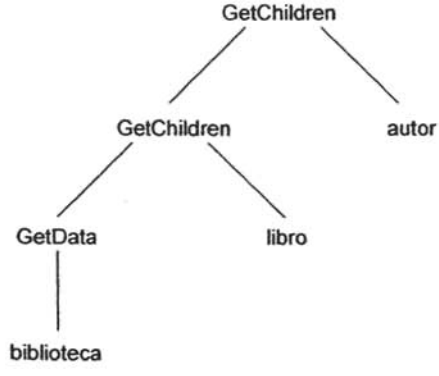
5.2.1 Conversión de una expresión de camino a plan físico

Una expresión de camino de la forma $l_1.l_2.....l_n$ se convierte a plan físico de la siguiente manera:

- Para l_1 utilizar la función GetData:
  GetData get
  get.setSsdTable(l_1)

- Para l_i con i=2..n utilizar la función GetChildren con el atributo `source` como la transformación de l_{i-1}:
  GetChildren get
  get.setSource(TLF(l_{i-1}))
  get.setlabel(l_i)

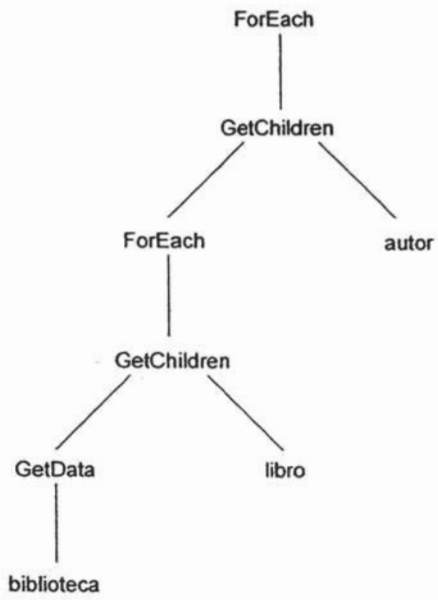
Ejemplo 5.2.1.1 La conversión de la expresión de camino *biblioteca.libro.autor* a plan lógico es:



El lado derecho de la función GetChildren representa la etiqueta y su lado izquierdo su atributo `source`.

Para este tipo de conversión es necesario utilizar el iterador `ForEach`, de modo que cada función vaya devolviendo un dato a la vez y cumplir con la técnica *Pipelining*. El iterador se aplica sobre la función `GetChildren`, pues para `GetData`, al ser una función que obtiene el identificador de una Ssd-tabla, siempre obtendrá un solo dato, por lo que no es necesario aplicar el iterador sobre esta función.

Por lo anterior, el plan físico de la expresión de camino del ejemplo 5.2.1.1 con iteradores, queda de la siguiente forma:



Para las expresiones de camino extendidas que utilizan el operador "|" se emplea un esquema de traducción que se basa en lo que describe la referencia [McWi99] que elimina dicho operador.

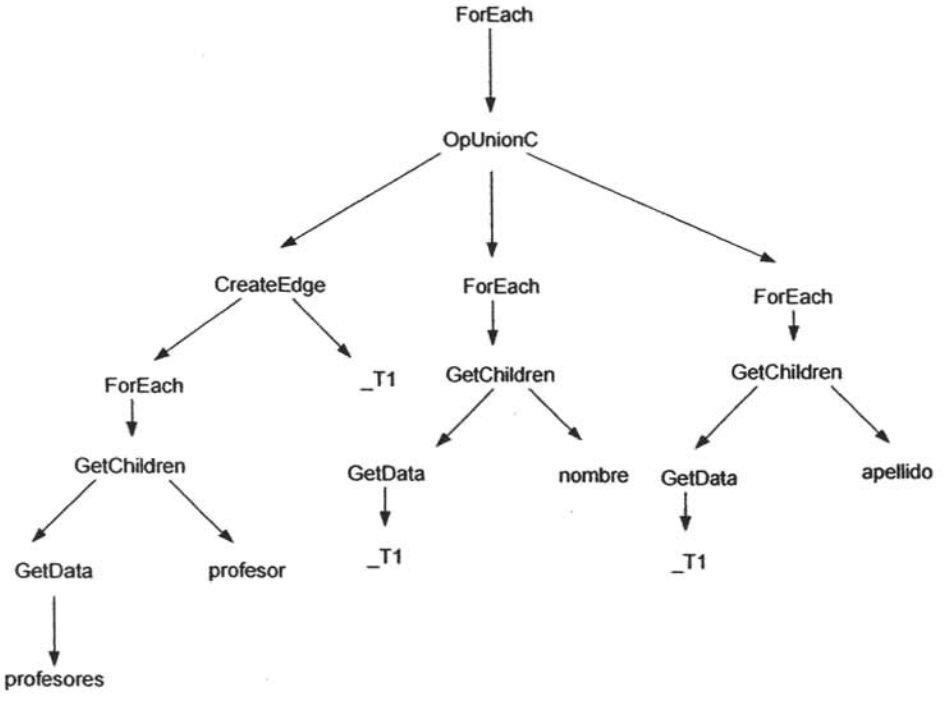
En el esquema de traducción que presenta [McWi99] se crean dos expresiones de camino para cada una de las opciones. Después de obtener los datos de ambas, se realiza una unión entre ellos obteniendo así el resultado final. Por ejemplo, la expresión de camino *biblioteca.libros.libro.autor.(nombre|apellido)* se divide en:

- *biblioteca.libros.autor.nombre*
- *biblioteca.libros.autor.apellido*

Después de obtener los dos planes físicos de cada expresión, se inserta, al final de ellos, la función `OpUnionC`, la cual une las dos colecciones que devuelven las expresiones de camino.

Lo anterior tiene el inconveniente de no tomar en cuenta el avance realizado por las funciones físicas, al recorrer el camino común entre las dos expresiones. Para resolver esta situación, la clase `OpUnionC` contiene un atributo extra llamado `source`, al cual se le especifica, antes de realizar la unión, la parte del plan físico a ejecutarse. En ella se establece el plan físico de la parte de la expresión de camino común para las dos alternativas. De esta forma, se utiliza el recorrido del camino en común en ambos conjuntos de la unión. En el ejemplo siguiente se muestra lo anterior.

Ejemplo 5.2.1.2 La expresión de camino *profesores.profesor.(nombre|apellido)* al convertirla a plan físico queda de la siguiente manera:



La Ssd-tabla temporal "_T1" se crea para representar el identificador recuperado por la expresión de camino común. Esto es posible gracias al iterador. De esta forma, se obtiene identificador por identificador del

camino común. En su momento, cada uno de éstos estará dentro de la Ssd-tabla temporal "_T1". Con lo anterior, las dos opciones tienen un inicio en común y utilizan la información recuperada por la expresión de camino en común.

Para los operadores de ocurrencia (*, +, ?) se definen etiquetas especiales que pueda reconocer la función `GetChildren`. Con ellas, es posible retornar el dato obtenido al atravesar el árbol de datos con la etiqueta especificada, o regresar el dato devuelto por un atributo `source`, esto para el caso del operador ?, o continuar recuperando datos recorriendo el árbol con la misma etiqueta, en el caso del operador +. El operador * se construye utilizando los dos métodos descritos anteriormente.

Lo anterior aplica para las expresiones de camino de primer nivel, en donde el conjunto de etiquetas se toma como alfabeto. Para las de segundo nivel, los operadores |, *, + y ? se aplican a los caracteres de las etiquetas contenidas dentro de la expresión de camino. Para lo anterior, la etiqueta se convierte a una expresión regular de la forma descrita en [SunM04], utilizando el método `matches` de la clase String en Java (algo similar a lo realizado con las cadenas de caracteres utilizadas por el operador lógico LIKE).

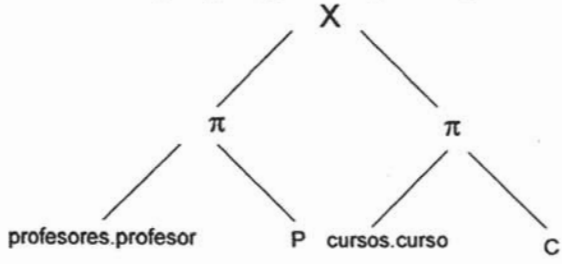
5.2.2 Conversión de los operadores lógicos X y DJoin a plan físico

La conversión de los operadores *X* y *DJoin* resulta laborioso al implementarlo en la forma como se describe en el capítulo 3, pues llevaría a construir más funciones físicas de diferentes tipos de producto cruz (de ciclo anidado, basado en tablas hash, etc.) y de *DJoin*. Esto obligaría a realizar un plan de optimización que compare los diferentes tipos de unión disponibles y determinar cuál es el mejor con base en el número de accesos a disco y tamaño del resultado. Todo lo anterior conduce no sólo a requerir más recursos para almacenar los resultados temporales, sino a construir un optimizador más complejo que posiblemente retarde la ejecución por la búsqueda de cual función aplicar a un producto cruz. En [McWi99a] se describen algoritmos para reducir esta búsqueda y así el proceso de optimización no tome más tiempo del que lleva ejecutar la consulta sin él.

Con la implementación del iterador, es posible evadir ambos problemas. Primero se evita la construcción de resultados temporales de

tamaño considerable al sólo tomar y devolver un dato a la vez. De este modo, se simplifica también la elección de qué algoritmo utilizar para el producto (*X*) o mezcla (*DJoin*). Al manejarse un conjunto de datos tan pequeño (al menos en una de las dos partes), una mezcla basada en el algoritmo de ciclo anidado es suficiente para realizar el trabajo.

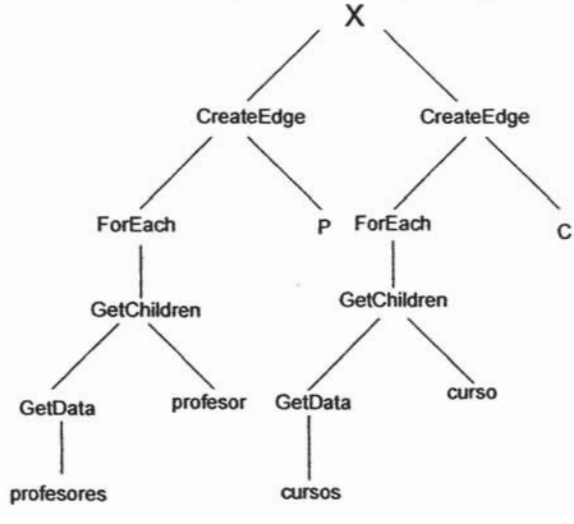
Entonces, para efectuar el producto de los dos operadores contenidos dentro de un producto cruz, se debe integrar el ciclo descrito por uno al del otro. Por ejemplo, para un plan lógico definido como:



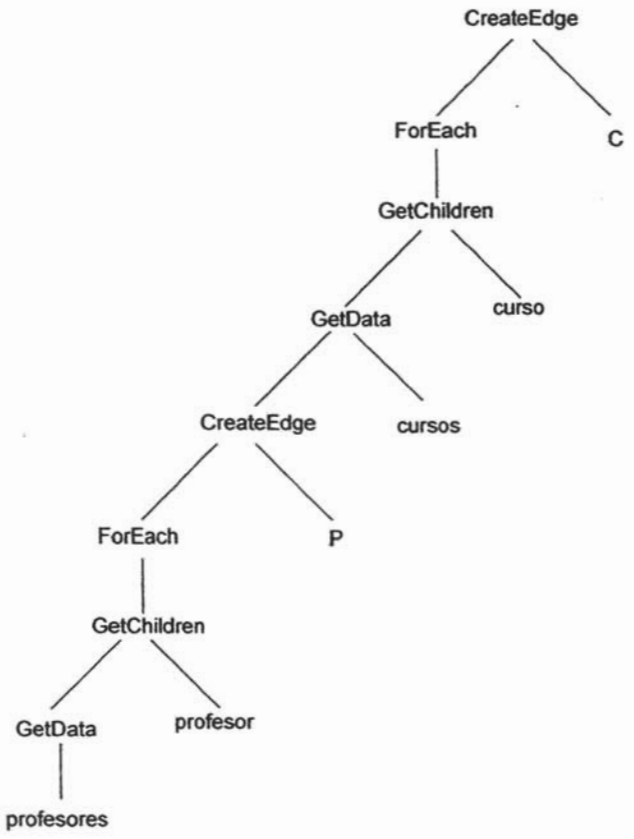
el cual describe una cláusula FROM definida como:

```
FROM        profesores.profesor as P,
            cursos.curso as C
```

al convertirlo a plan físico, el operador *X* debe realizar un producto de las colecciones regresadas por cada π. Los iteradores `ForEach` utilizados por cada expresión de camino ayudan a efectuar el anidamiento de los ciclos (*Nested-Loop Join*) necesarios para obtener dicho producto. Es decir, la primera función a ejecutar del plan físico de uno (el π de la derecha) debe contener al plan físico del otro (el π de la izquierda):

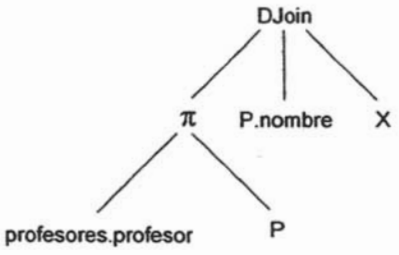


Para realizar el producto de los planes físicos, se incrusta la raíz del plan de la izquierda (`CreateEdge`) dentro de la función más a la izquierda y abajo del árbol del plan de la derecha, es decir, dentro de la función a ejecutar primero. Para este caso es `GetData`. El plan físico final queda de la siguiente forma:

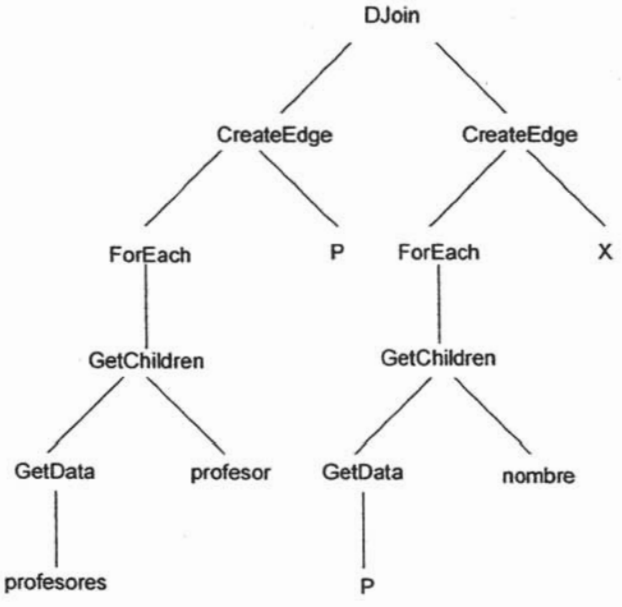


Para el operador lógico *DJoin*, la traducción a plan físico se realiza de forma similar a la del producto cruz. Como puede observarse en la definición del operador *DJoin* en el capítulo 3, el segundo elemento de la unión realizada por cada elemento de la colección de entrada, llámese d_2, es similar al devuelto por un operador π. Por ello, puede utilizarse la misma traducción de un π para la segunda parte del DJoin.

Ejemplo 5.2.2.1 Para un *DJoin* definido como:

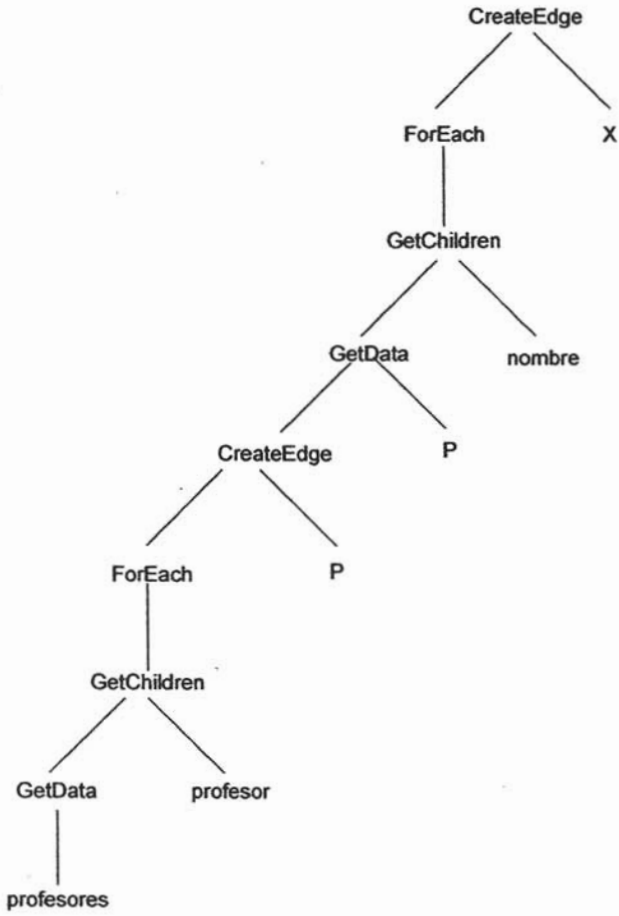


La traducción inicial de las colecciones que manejará es la siguiente:



Para finalizar la traducción del *DJoin*, por cada elemento de la colección de la izquierda hay que determinar la de la derecha. Como está utilizandose el iterador `ForEach`, se asegura que se devolverá elemento por elemento de la colección de la izquierda y no debe utilizarse otro sino hasta haberlo unido con todos los elementos de la colección de la derecha. Por lo anterior, la mezcla de ambas colecciones se realiza de forma similar al del producto cruz. Sin embargo, en el caso de la dependencia entre colecciones definida por el operador *DJoin* se realiza mediante alguna Ssd-tabla temporal — que este caso fue P — lo cual es algo parecido a lo que ocurre en la eliminación de disyunciones en una expresión de camino al traducirla a plan físico.

La traducción final del operador *DJoin* del ejemplo 5.2.2.1 es la siguiente:



Para fines educativos, se recomienda utilizar el procesador de consultas construido junto con esta tesis, el cual muestra todos y cada uno de los planes generados al analizar una consulta.

5.2.3 Implementación de la traducción

Para obtener el plan físico, cada operador debe generar la parte que le corresponde de acuerdo al esquema de traducción presentado anteriormente (tabla 5.2.1). Para ello, se utilizó el patrón de diseño *Visitor* [Cart00, Mart02], el cual define una disciplina elegante para separar un algoritmo de una estructura de objetos.

El plan lógico se representa por una colección de objetos (instancias de clases generadas por JJTree) relacionados unos con otros, lo que construye una estructura arborescente. Todas las clases a las que pertenecen dichos objetos son derivadas de una que especifica el comportamiento básico de un nodo en una estructura arborescente, esto es, especificar el padre y añadir, quitar u obtener los hijos.

La idea primordial de utilizar el patrón de diseño *Visitor*, es que cada una de estas clases tenga un método `Aceptar` que admita un objeto tipo `Visitante`. La clase `Visitante` es una interfaz que tiene un método `Visitar` distinto para cada tipo de clase, el cual se diferencia por el tipo de sus parámetros. El método `Aceptar` de cada clase llama de regreso al método `Visitar` del objeto `Visitante`, correspondiente a la clase.

De esta forma, para agregar un método que se necesite definir a lo largo de todas estas clases, sólo basta crear una clase concreta del tipo `Visitante`, que defina lo que cada método tiene que hacer para cada una de las clases. Desde otro punto de vista, un método que se debería definir para cada una de las clases (definiendo así un comportamiento de la misma) se agrupa en una sola clase.

Por lo anterior, para que cada objeto del plan lógico genere su parte del plan físico correspondiente, se implementó la tabla 5.2.1 como una clase concreta de tipo `Visitante`, a la que se le escribe el código pertinente para cada tipo de objeto del plan lógico. Para la traducción de una expresión de camino, como la descrita en el apartado 5.2.1, se utilizaron clases extras a un nivel más detallado que en el plan lógico, las cuales representan a las partes que conforman una expresión de camino. De esta forma, la traducción se realiza mediante la especificación del método pertinente de cada una de estas clases extras en la clase concreta `Visitante`.

La traducción de los operadores *X* y *DJoin,* descritos en el apartado 5.2.2, se realizó con el método pertinente para la clase `Join` en la clase concreta `Visitante` debido a la semejanza en la traducción de ambos operadores.

Otras tareas también fueron hechas definiendo una nueva clase concreta `Visitante`, dejando las clases originales intactas, como es el caso para desplegar el plan lógico en pantalla.

Uno de los inconvenientes de utilizar el patrón de diseño *Visitor* es que habrá veces en que sea necesario manejar los atributos propios de una clase fuera de ésta, es decir, en la clase Visitante. Esto rompería con el concepto de encapsulamiento del paradigma orientado a objetos. Lo anterior no fue el caso para la implementación de la traducción del plan lógico a físico, debido a la sencillez con la que fue definida cada clase del plan lógico. Pero, como se muestra en este capítulo, las clases del plan físico no cuentan con esta característica, por lo que utilizar el patrón *Visitor* para ejecutar la consulta, llevaría a definir sus atributos internos como públicos, comprometiendo la misma ejecución al abrirlos a cualquier acceso externo. Además, no todas las clases se ejecutan con el mismo método (podría haberse llamado al método de ejecución de igual forma, pero se define con nombre diferente para ayudar a entender el comportamiento de cada clase). Por esta razón, para el caso de las clases del plan físico, no se utilizó el patrón de diseño *Visitor* para especificar los métodos de ejecución.

5.3 Resumen y discusión

En este capítulo se describió cada una de las funciones que intervienen en la ejecución de la consulta. Su diseño se apegó a las que ofrece el almacenamiento primitivo, mismas que se listan en el capítulo 6. Además, se especificó la traducción del plan lógico a plan físico. No obstante que el plan lógico se basó en trabajos anteriores [MAG+97, BeTz99, FrHP02], el plan físico es propio para el sistema administrador de bases de datos semiestructurados en el que este trabajo colabora. Por esta circunstancia, las instrucciones del almacenamiento primitivo se definieron específicamente para este sistema y la traducción del plan lógico al físico fue una tarea esencial en esta tesis. En la investigación realizada se conoció que en la Universidad de Stanford [MAG+97] ofrecen algunos ejemplos de este tipo de traducción. No obstante que su almacenamiento primitivo es distinto al aquí utilizado, su trabajo ayudó a concebir la traducción de lógico a físico en este proyecto. La documentación sobre bases de datos relacionales [GaUW00, SiKS02] también aportó ideas para realizar esta tarea.

En la descripción de la traducción del plan lógico a plan físico se observan dos puntos importantes:

- La eliminación de disyuntivas dentro de las expresiones de camino
- La elección del algoritmo de mezcla a utilizar para realizar el producto cruz y la unión *DJoin*.

Ambos puntos fueron posibles debido a la introducción de la técnica *Pipelining* (tratada en el capítulo 4). El primer punto representa una mejora al procedimiento descrito en [McWi99]. Con lo propuesto aquí, se puede aprovechar el recorrido que realizan las funciones del plan físico para ambas opciones. Ésto se efectúa mediante la creación de una tabla temporal extra que especifique el punto de partida de las alternativas.

Para realizar lo descrito por los operadores lógico *X* y *DJoin* en el plan físico, se utilizó el algoritmo de ciclo anidado (*Nested-Loop*). Cuando se maneja gran cantidad de datos, generalmente el algoritmo de ciclo anidado no es la mejor opción para realizar la mezcla de dos fuentes de datos (colecciones en este caso). Sin embargo, con el uso de la técnica

Pipelining el número de datos manejados en determinado momento se reduce y así emplear el algoritmo *Nested-Loop* es suficiente para realizar la tarea descrita por los operadores *X* y *DJoin*.

Con este capítulo termina la descripción de los componentes que integran el procesador de consultas. Se describieron las etapas por las que pasa una consulta dentro del procesador y se dieron las razones por las que se realiza cada una. Aún cuando el plan físico es un programa ejecutable (dentro del sistema administrador de bases de datos que utiliza funciones proporcionadas por el almacenamiento primitivo), es otro el componente que lo ejecuta, el motor de ejecución.

El almacenamiento primitivo es parte de otro trabajo desarrollado paralelamente con éste. Con la contribución de esta tesis, se cuenta con dos de los más importantes componentes de un sistema administrador de bases de datos semiestructurados, el procesador de consultas y el almacenamiento primitivo. Sin embargo, como puede observarse en la figura 2.1.1, el motor de ejecución es el que realiza la comunicación entre el procesador de consultas y el almacenamiento primitivo. Por lo tanto, es necesario contar con el motor de ejecución para comprobar los resultados del procesador de consultas y del almacenamiento primitivo.

En el siguiente capítulo se describe del motor de ejecución; su funcionamiento y labor que desempeña al ejecutar una consulta dentro del sistema administrador de bases de datos semiestructurados.

Capítulo 6

El ambiente de ejecución

Algunas funciones que utilizan las clases que componen el plan físico de consulta son propias del almacenamiento primitivo, otras son proporcionadas por el motor de ejecución. Esto es debido, principalmente, a la necesidad de éste último de controlar el ambiente (cantidad de memoria disponible, tablas temporales, aristas temporales, etc.) al momento de ejecutar la consulta.

Las llamadas a estas funciones se ejecutarán como si todas pertenecieran al motor de ejecución. Cuando se trate de funciones primitivas, el motor de ejecución creará un puente entre la clase que la llamó y el almacenamiento primitivo. Las funciones que pertenecen al motor de ejecución son:

- Para la creación de datos temporales

```
LabeledSSD createResultData(ListOfLabeledSSD children)
LabeledSSD createTempNode(ID id, ListOfLabeledSSD edges)
LabeledSSD createTempPrimitive( PrimitiveValue v)
```

- Para la creación de aristas temporales

```
LabeledSSD createTempTable (Label l, ID target)
```

- Para verificar si un dato es temporal

```
Bolean isNew(ID ssd)
```

- Para la creación de Ssd-tablas desde un documento XML

```
Bolean addSSDTableFromFile( ID tableId, File file)
```

Las funciones que proporciona el almacenamiento primitivo son:

- Para la creación de nuevos datos

```
ID CreatePrimitive ( PrimitiveValue v )
ID CreateNonPrimitive()
ID CreateNonPrimitive(ListOfLabeledSDD l)
```

- Para la destrucción de datos

```
drop(ID ssd)
```

- Para modificar el contenido de un dato

```
add (ID ssd, LabeledSSD lssd)
add (ID ssd, ListOfLabeledSSD l)
remove (ID ssd, ID ssdc)
remove (ID ssd, Label l)
remove (ID ssd, ListOfLabeledSSD l)
updateToPrimitive (ID ssd, PrimitiveValue v)
```

- Para obtener el contenido de un dato

```
PrimitiveValue getValue (ID ssd)
Int getType (ID ssd)
Bolean isPrimitive (ID ssd)
```

- Para obtener relaciones de parentesco

```
Bolean contains (ID ssd, ID ssdc)
Bolean owns (ID ssd, Label l)
Bolean hasParent (ID ssd, ID ssdp)
```

- Para los hijos de un dato

```
ListOfSSD get (ID ssd, Label l)
```

- Para obtener los padres de un dato

```
ListOfSSD getParents (ID ssd)
```

- Para manipular el diccionario de datos

```
addSSDTable (Label l, ID ssd)
removeSSDTable (Label l)
ID getID (Label l)
addView (Label l, ViewDefinition def)
removeView (Label l)
ViewDefinition getViewDefinition (Label l)
```

Para estas funciones, el tipo `Label` se implementó con la clase `String` de Java. El tipo `ID` es utilizado para los identificadores de los datos semiestructurados, en cuya implementación se empleó el tipo `long` de Java. Aunque en la descripción del lenguaje Ssquirel [Garc02] se recurre a identificadores alfanuméricos (por ejemplo, "a4"), se utilizó un tipo numérico debido a que las comparaciones entre números son más directas que las que se hacen entre cadenas de caracteres. Además, dichos identificadores son internos del sistema administrador de bases de datos, por lo que cambiarles de tipo no modifica la forma de trabajar del sistema para el usuario.

El tipo `ViewDefinition`, utilizado para crear o recuperar una vista abstracta, en la implementación es el tipo `PhysicalNode`, pues como describe el capítulo 5, lo que se guarda en una vista abstracta es su plan físico. El tipo `ListOfLabeledSSD` se instrumentó como un objeto de la clase proporcionada por Java `java.util.Vector`. Ésta se utiliza para crear una colección de objetos de cualquier tipo, como el `LabeledSSD` en este caso.

El motor de ejecución, al igual que las funciones del plan físico de consulta, se creó como una clase y cada instancia de ella representa un motor de ejecución. Por ello, cada función se implementa como un método dentro de esta clase.

Este capítulo describe la forma de ejecución de una consulta, las clases que la controlan y el ambiente creado por el motor de ejecución.

6.1 El motor de ejecución

El motor de ejecución se implementa con la clase llamada `EEngine` (ver figura 6.1.1). El motor inicia la ejecución del plan físico de consulta llamando a los métodos pertinentes dependiendo del tipo al que pertenezca el objeto raíz de dicho plan. Como puede observarse en los diagramas de clase del capítulo 5, los métodos ejecutables de cada clase tienen como parámetro de entrada al motor de ejecución, el cual es necesario para llamar a las funciones físicas o primitivas que representa cada clase.

El método `execute` inicia la ejecución del plan físico de consulta, dependiendo del tipo de objeto especificado como la raíz del plan. Existen dos tipos de ejecución: para recuperar datos y para modificarlos (borrar, actualizar, insertar). Tratándose del primer caso, se ejecutará el método `getids` y `run` para el segundo. Al ejecutar dichos métodos, el objeto raíz llama a los nodos descendientes de él, de los cuales se servirá para buscar, construir o borrar información de la base de datos. Así, cada objeto realizará lo mismo con sus descendientes, hasta llegar a las hojas del plan físico de consulta. La mayoría de las consultas se basan en la recuperación de nodos ya existentes dentro de la base de datos, con los cuales se modificará, agregará o se regresará información. Por lo anterior, las hojas del plan físico, usualmente, son objetos del tipo `GetData`, es decir, son funciones de recuperación de datos.

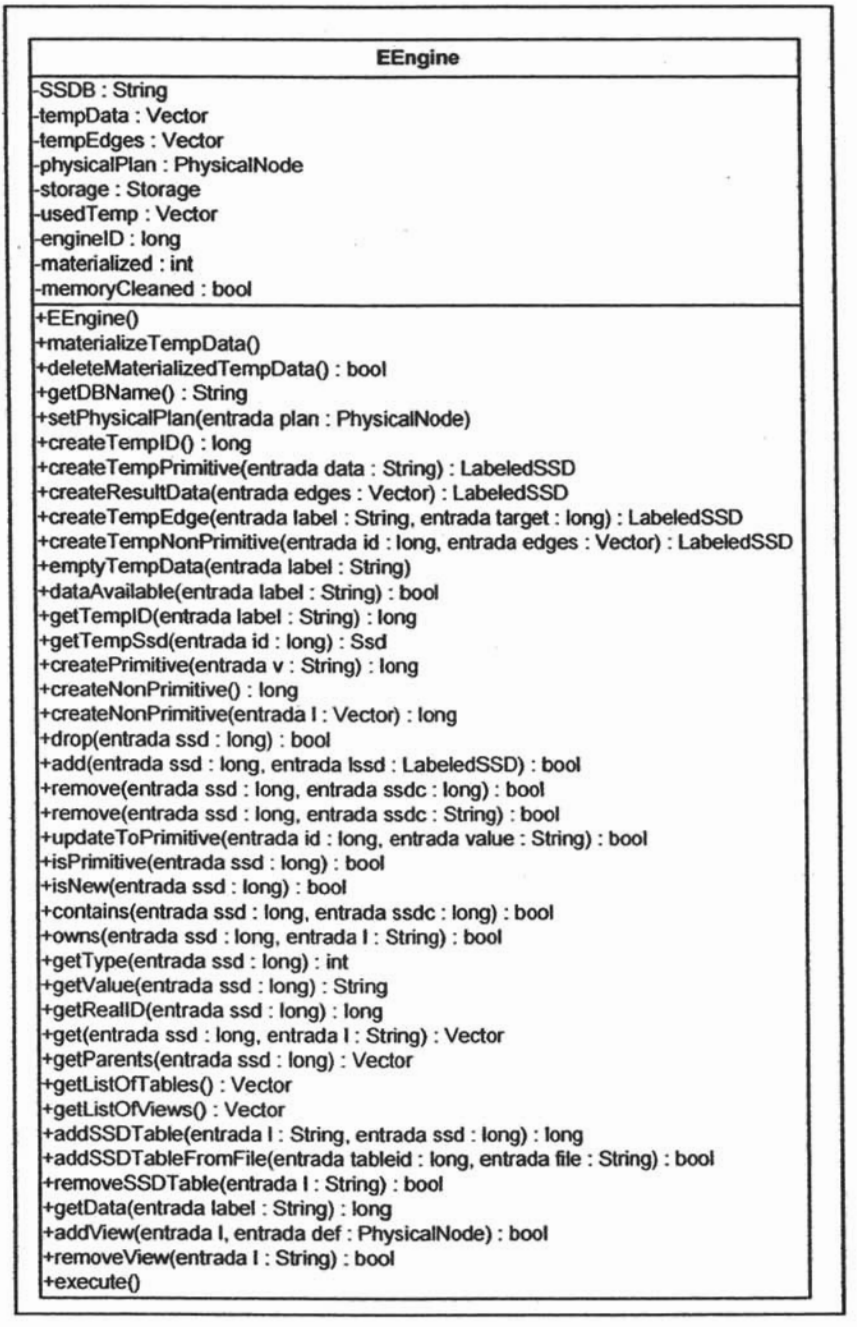


Figura 6.1.1 *Diagrama de la clase EEngine*

Es necesario que todos los objetos ejecuten las funciones físicas o primitivas desde el mismo motor de ejecución, pues dentro de él están definidos los datos temporales para la consulta que esté en ejecución.

6.2 Manejo de datos temporales

Al crearse una instancia de la clase `EEngine`, se inicializan los atributos que contendrán los elementos temporales creados durante la ejecución, que son `tempData`, `usedTemp` y `tempEdges`, para almacenar los nodos temporales, los materializados por la consulta y aristas temporales, respectivamente. En cada una las funciones que recuperan datos (aquéllas que su nombre inicia con "get"), primero se busca el dato en los datos temporales, (atributos `tempData` o `tempEdges`), si no está ahí, entonces la búsqueda se hace en los datos materializados, es decir, dentro del almacenamiento primitivo.

Para todas las funciones de creación de datos temporales, `createTempTable`, `createTempNode`, `CreateTempData`, se monitorea si hay suficiente memoria disponible para la creación del nuevo dato temporal. En el caso de que no sea así, se materializan todos los datos temporales contenidos en los atributos `tempData` y `tempEdges`, creando nuevos nodos o aristas en la base de datos. Lo anterior permite eliminarlos de los atributos temporales al no ser ya necesarios y ello libera espacio en memoria. Cada uno de los datos temporales materializados se guardan en una lista en disco para borrarse después de terminada la ejecución de la consulta. No se eliminan si la propia consulta los materializó al llamar a la función `isNew`, la cual se llama antes de grabar un dato en el almacenamiento secundario. En este caso, los datos temporales materializados se registran en el atributo `usedTemp`; de esta forma, al borrarse no desaparecerán aquellos guardados en la base de datos por la consulta.

Con el objeto de mantener conocimiento de qué datos temporales se materializaron para su posterior eliminación al terminar de ejecutar la consulta, es necesario guardarlos en una lista de nodos y aristas. Pero, dado que la materialización de datos temporales se realiza debido a la carencia de memoria disponible, no es posible hacerlo en ella, sino en disco. Al crear dicha lista, su llenado se realiza en memoria, pero es necesario borrar datos antes de agregarle otro. Es por esto que cada vez que se materializa un dato temporal, éste se elimina de la lista correspondiente (`tempData` o `tempEdges`), liberando espacio en memoria. Después sólo se incorpora el identificador o etiqueta (con el identificar origen y destino), según sea un nodo o una arista temporal la que se

guarde, dentro de la lista, misma que se utilizará para posteriormente eliminarlos.

Al materializar un dato temporal, se hace uso de las funciones primitivas `createPrimitive`, `createNonPrimitive`, `add` y `addSSDTable`. Las dos primeras para la creación de nodos y las dos últimas para la materialización de aristas temporales. Cada una de estas primitivas regresa el identificador asignado a dicho dato dentro de la base de datos. Este identificador es el que se guarda dentro de la lista que se utilizará para borrarlos y no el que tenía asignado el dato cuando estaba en memoria.

Los algoritmos de creación y eliminación de datos temporales materializados se muestran en las figuras 6.2.1 y 6.2.2, respectivamente.

```
// tempEdges - Es un arreglo que contiene la lista de
// aristas
// temporales representadas mediante objetos del tipo
// LabeledSSD.
// tempData - Es un arreglo que contiene la lista de nodos
// temporales
// representados como objetos del tipo Ssd.
// mat - Es la lista que contendrá los identificadores de
// los datos
// temporales que serán eliminados al terminar la
// ejecución
// rel - Es una tabla hash donde se almacena las parejas
// <identificador memoria>, <identificador materializado>
// materializaciones - Variable entera global que almacena
// el número de ocasiones en las que se han materializado
// datos temporales

MaterializarDatosTemporales(){
      Mientras tempData tenga datos
            Nodo = tempData.removerDatoEn(0)
            Si Nodo es primitivo
                  mid = createNonPrimitive(Nodo)
            sino
                  mid = createNonPrimitive(Nodo)
                  Por cada hijo H de Nodo
                        hid = H.id
                        Si rel contiene al identificador de hid
                              hid = rel.key(hid)
                        add(mid, hid)
            rel.agregar(Nodo.id, mid)
            Agregar a la lista mat el identificador mid
      Guardar mat en disco con nombre "nodos-"+materializaciones
```

```
      Vaciar lista mat
      Mientras tempEdges tenga datos
            LabeledSSD dato = tempEdges.removerDatoEn(0)
            createSSDTable(dato.etiqueta,
dato.identificador)
            Agregar a la lista mat la etiqueta dato.etiqueta
      Guardar mat en disco con nombre "aristas-"+
materializaciones
      materializaciones = materializaciones + 1
}
```

Figura 6.2.1 *Algoritmo para la materialización de datos temporales*

```
// materializaciones - Variable entera global que almacena
el número
      // de ocasiones en las que se han materializado datos
temporales

BorrarDatosTemporalesMaterializados{
      i = 0;
      mientras i < materializaciones
            Recuperar lista mat desde el archivo llamado
                  "aristas-"+i
            Mientras mat contenga datos
                  etiqueta = mat.removerDatoEn(0)
                  removeSSDTable(etiqueta)
            Recuperar lista mat desde el archivo llamado
"nodos-
      "+i
            Mientras mat contenga datos
                  Ssd Nodo = mat.removerDatoEn(0)
                  Si nodo no se encuentra en la lista
UsedTemp
                        Hijos = get(Nodo.id, "#")
                        Para cada hijo h en Hijos
                              remove(Nodo.id, h.id)
                        padres = getParents(Nodo.id)
                        Para cada padre p en padres
                              remove(p.id, Nodo.id)
            Borrar archivo "aristas-" + i
            Borrar archivo "nodos-" + i
            i = i + 1
}
```

Figura 6.2.2 *Algoritmo para borrar datos temporales materializados*

6.3 Creación de una Ssd-tabla desde un archivo

Cuando se crea una Ssd-tabla utilizando un documento XML para cargar sus datos, se hace uso del método `addSSDTableFromFile` del motor de ejecución. Este método realiza un análisis al documento mediante el API de SAX para trasladar los datos XML a datos semiestructurados.

Existe una diferencia entre el modelo de documentos XML y el modelo de datos semiestructurados. Mientras que en XML es válido encontrar elementos de contenido mixto, esto es, que tienen información tipo texto y a su vez también elementos internos, en el modelo de datos semiestructurados esto no es válido, pues no existe el concepto de nodo de contenido mixto. En el ambiente de datos semiestructurados, un dato es un primitivo o es un semiestructurado, es decir, contiene información de tipo primitiva (cadenas de caracteres, números enteros, etc.) o contiene aristas con etiquetas que apunten a otro dato.

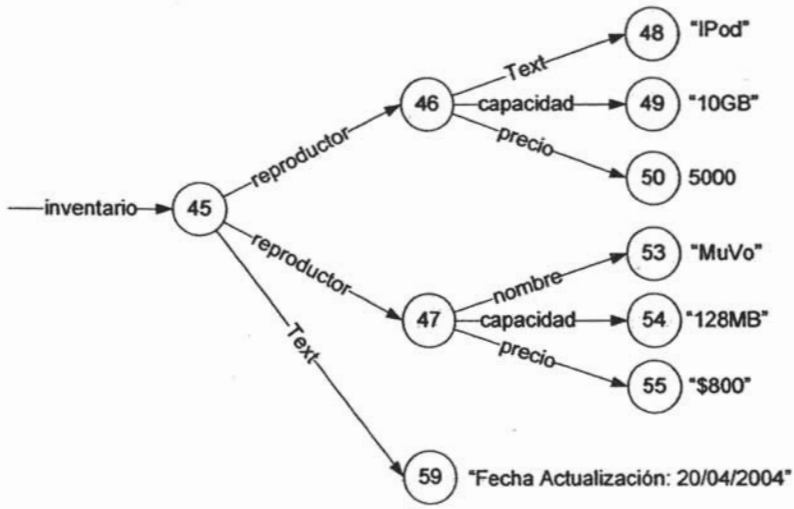
Con base en lo anterior, al realizar la transformación de un documento XML a datos semiestructurados, para todo elemento X de XML que contenga información mixta, se crearán nodos primitivos P para cada contenido tipo texto dentro de X y se conectarán al nodo semiestructurado S construido para representar a X con una arista etiquetada como "Text".

Ejemplo 6.3.1 La representación del documento XML:

```
<?xml version="1.0" ?>
<inventario>
      <reproductor>
            IPod
            <capacidad>
                  10GB
            </capacidad>
            <precio>
                  5000
            </precio>
      </reproductor>
      <reproductor>
            <nombre>
                  MuVo
            </nombre>
            <capacidad>
                  128MB
            </capacidad>
            <precio>
                  $800
```

```
            </precio>
        </reproductor>
        Fecha Actualización: 20/04/2004
</inventario>
```

en datos semiestructurados es:



Como se ha mencionado, la carga de datos desde un documento XML a la base de datos semiestructurados se realiza mediante un análisis con la API de SAX [Deve02]. SAX hace un recorrido secuencial de inicio a fin del documento XML y dispara eventos al encontrar el inicio o el final de un elemento, el de un documento, simplemente texto, etc. Por cada uno de estos eventos, se ejecutan las funciones primitivas pertinentes para el traspaso de los datos. La Tabla 6.3.1 muestra las funciones primitivas que se ejecutan por cada evento ocurrido, utilizando una estructura de datos *pila* para almacenar los elementos creados.

Evento	***Funciones Primitivas***
Al iniciar un elemento de nombre Nname	id = createNonPrimitive(); padreid = pila.pop() add(padreid, Nname.id) pila.push(padreid) pila.push(id)
Al terminar un elemento	pila.pop()
Al encontrar caracteres ch	id = createPrimitive(ch) padreid = pila.pop() Si padreid no tiene hijos updateToPrimitive(padreid, ch)

	sino add(padreid, TEXT.id) pila.push(padreid)

Tabla 6.3.1 *Funciones utilizadas para la carga de documentos XML*

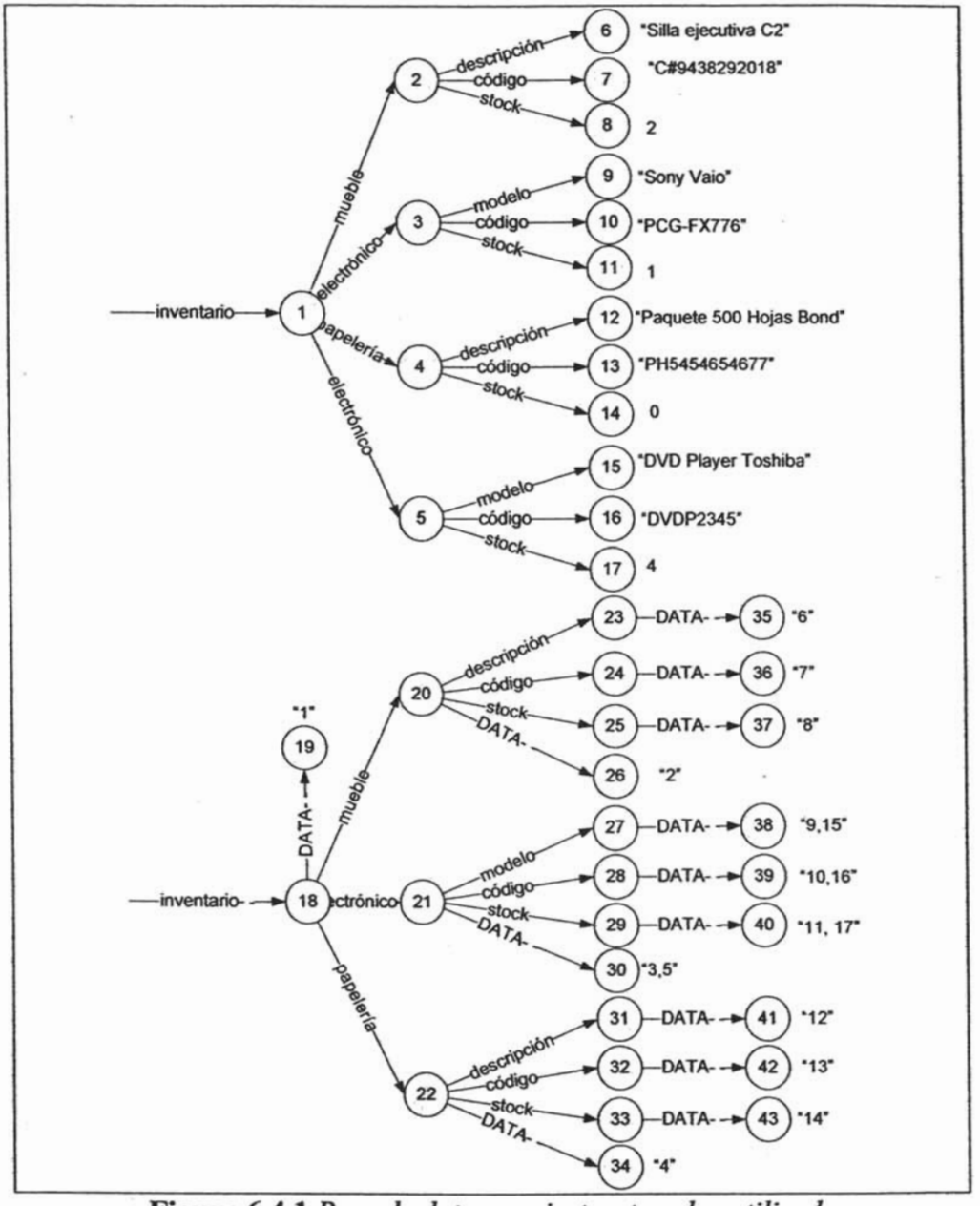


Figura 6.4.1 *Base de datos semiestructurados utilizada por los ejemplos de la sección 6.4*

6.4 Ejecución de una consulta

Una consulta sufre tres transformaciones antes de ejecutarse, desde un enunciado escrito en el lenguaje de consulta Ssquirel hasta un plan físico. Con cada transformación se ha enfocado a mejorar aquellas partes de la consulta que realizan la recuperación de los datos desde la base de datos, es decir, las expresiones de camino. Todas estas transformaciones se llevan a cabo dentro del procesador de consultas, que es un componente del sistema administrador de bases de datos, como muestra la figura 2.1.1.

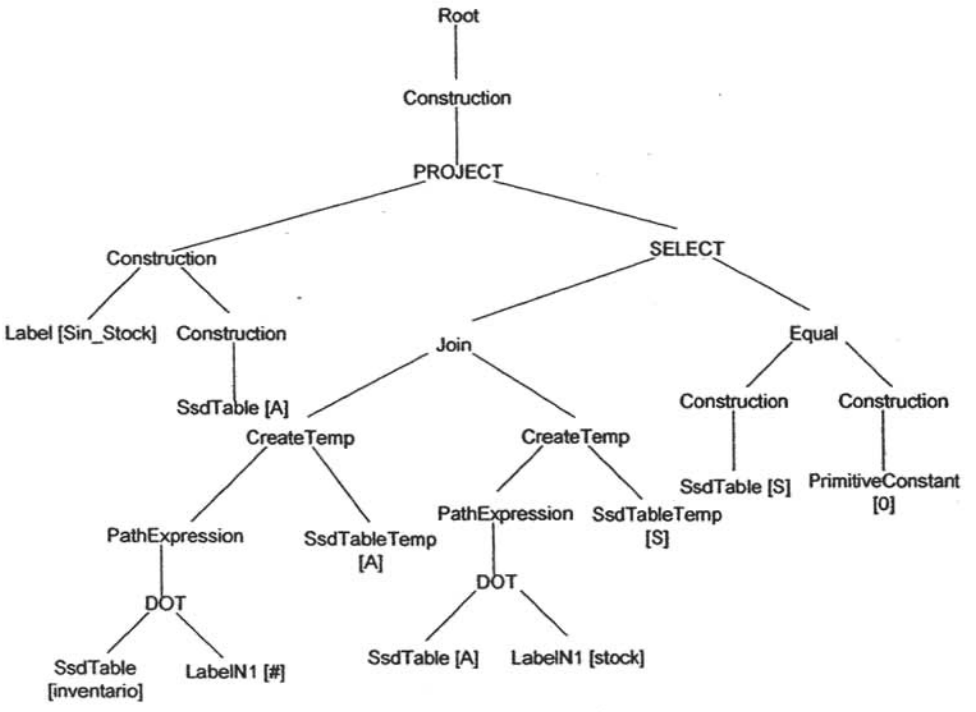
La ejecución del plan físico se realiza por otro componente del sistema administrador de bases de datos, el motor de ejecución, el cual debe comunicarse con otros componentes, como el administrador de transacciones, el control de concurrencia, etc. Estos componentes aún se no han desarrollado, por lo que la interacción ellos y el motor de ejecución no se ha realizado.

La ejecución de una consulta inicia cuando el procesador de consultas la recibe, con lo cual empieza su análisis léxico y sintáctico. Estos análisis devuelven un conjunto de objetos conectados entre sí representando el árbol de análisis sintáctico, que es el árbol de derivación que representa a la consulta de entrada. Esto último lo muestra el ejemplo 6.4.1. Todos los ejemplos de esta sección utilizan la base de datos de la figura 6.4.1.

Ejemplo 6.4.1 El árbol de análisis sintáctico resultante del análisis léxico y sintáctico de la consulta

```
SELECT      Sin_Stock : A
FROM        inventario.# AS A,
            A.stock AS S
WHERE       S = 0
```

la cual encuentra aquellos artículos sin existencias, es el siguiente:



Las reglas de producción que se utilizan para obtener este árbol se encuentran en el apéndice A, pues forman parte del análisis léxico y sintáctico.

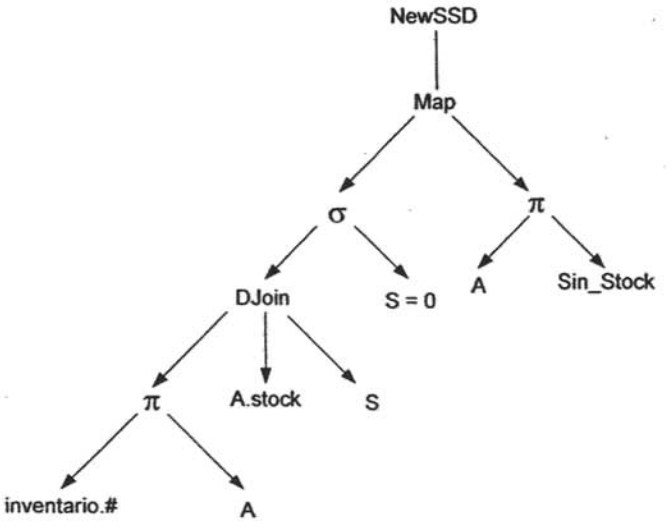
En el árbol de análisis sintáctico existe mucha información que es útil para las siguientes etapas del procesador de consultas y se asemeja en algunas partes al plan físico. Por lo tanto, este árbol se conserva mientras se realizan las acciones dentro del procesador.

A partir del árbol de análisis sintáctico se obtiene el plan lógico de consulta que utiliza sólo los operadores descritos en el capítulo 4. Para obtener el plan lógico es necesario realizar un análisis en el árbol de análisis sintáctico, de forma que se puedan determinar que operadores lógicos utilizar en que producciones. Este análisis es descrito en la tabla 6.4.2. Las producciones que no se definen en la tabla 6.4.2 no se consideran debido a que su conversión de árbol de análisis sintáctico a plan lógico queda de la misma forma.

Nombre de la derivación en el árbol de análisis sintáctico	*Operador lógico*
PROJECT	MAP
SELECT	σ
Join	X o DJoin Para determinar que operador utilizar se debe realizar un análisis en la segunda ramificación en la producción Join. Si es una expresión de camino que inicia con una Ssd-tabla se utiliza el operador X, si inicia con una Ssd-tabla temporal se utiliza el operador DJoin.
CreateTemp	π
Construction	Si tiene dos ramificaciones se utiliza el operador π, si sólo tiene una no se utiliza ningún operador y se pasa la etiqueta de su hijo.
DOT	Se concatenan las etiquetas de los hijos de esta producción mediante el signo "."
UPDATE	Se realiza la transformación definida en la sección 3.9.2.1 utilizando el operador MAP y Update
PathExpression	Se concatenan las etiquetas de los hijos de esta producción
PICK	Se aplica la transformación definida en la sección 3.2 que utiliza el operador π.
DELTA	δ
Root	NewSSD

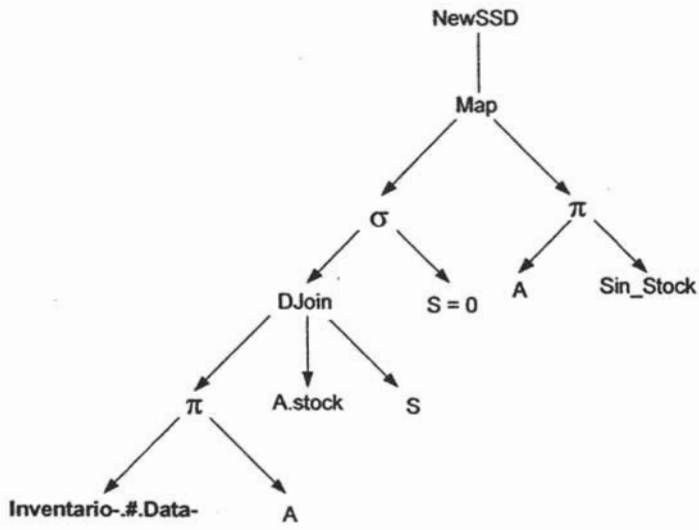
Tabla 6.4.2 *Relación entre producciones del árbol de análisis sintáctico y el plan lógico de consulta*

Ejemplo 6.4.2 El plan lógico del árbol de análisis sintáctico del ejemplo 6.4.1 es:



Al plan lógico se le aplica la técnica de uso del *resumen de datos con identificadores* descrito en el capítulo 4, cambiando el inicio de todas las expresiones de camino que comiencen con una Ssd-tabla a una etiqueta del mismo nombre agregando el caracter guión "-" al final. Además, se le adiciona una etiqueta extra al término de la expresión de camino (Data-).

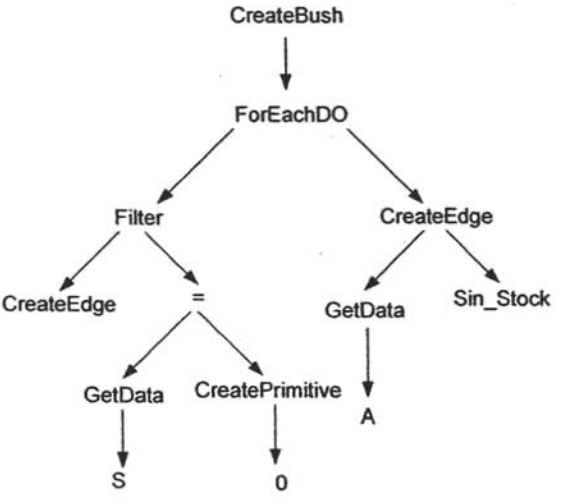
Ejemplo 6.4.3 El plan lógico del ejemplo 6.4.2 después de realizar la optimización de la utilización del *resumen de datos con identificadores* es:



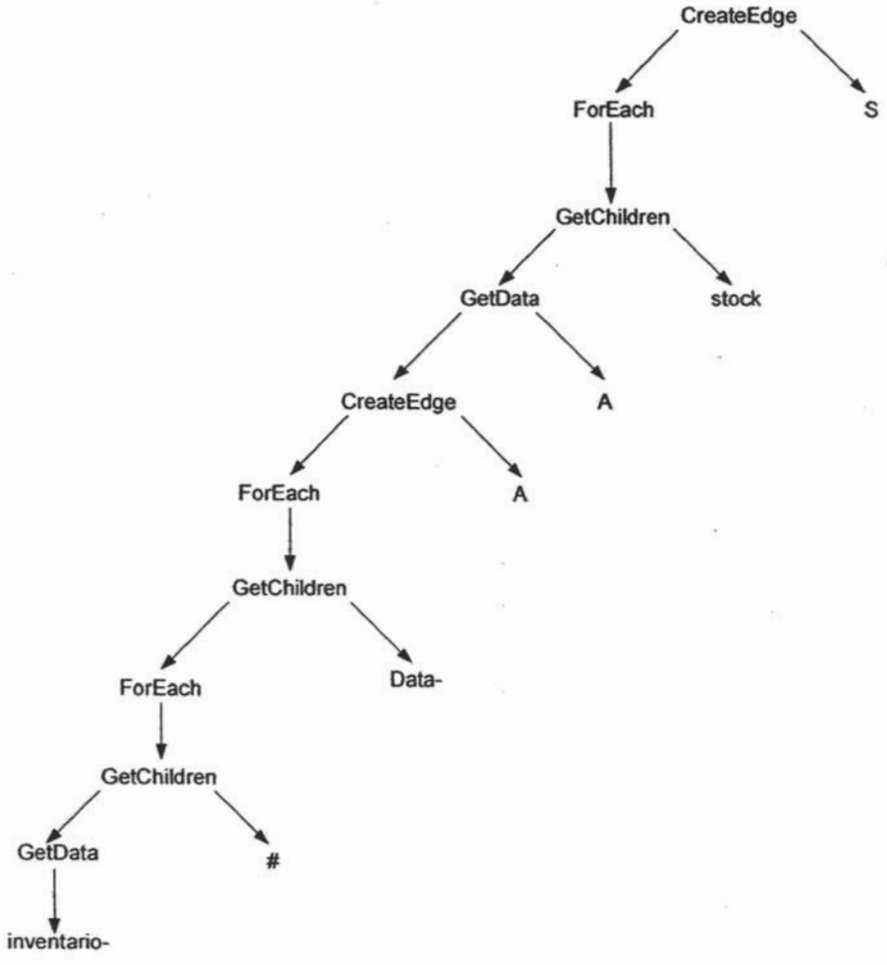
El último paso dentro del procesador de consultas es generar el plan físico de consulta, que consiste en producir un conjunto de objetos relacionados entre ellos, los cuales representan funciones físicas o primitivas. En esta etapa se resuelve la secuencia en la que se llevarán a cabo las acciones de los operadores *X* y *DJoin*, es decir, se construye un tipo de unión de ciclo anidado (*Nested-Loop Join* [GaUW00]). Además, se incluye el iterador `ForEach` en cada parte del plan donde ocurra una recuperación de datos, para realizar un consumo de datos entre operadores utilizando la técnica *Pipelining*.

Ejemplo 6.4.4 El plan físico derivado del plan lógico del ejemplo 6.4.3 se presentará en dos partes, debido al tamaño tan extenso que tienen este tipo de planes.

La parte superior del plan es:



El plan físico restante, que va dentro de la función `CreateEdge` más a la izquierda, es:



El plan físico pasa al motor de ejecución, el cual, después de inicializar el espacio de memoria reservado para ejecutar la consulta, llama el método que inicia la ejecución del plan físico del objeto raíz del mismo. En general, la ejecución del plan físico de consulta se realiza de la siguiente forma:

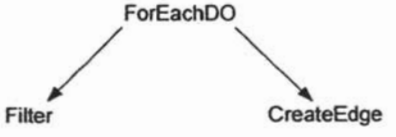
- El motor de ejecución inicializa los vectores `tempData`, `tempEdges` y `usedTemp` que contendrán los elementos temporales generados durante la ejecución de la consulta.
- Se crea el medio de comunicación entre el motor de ejecución y el almacenamiento primitivo, el cual recibirá y ejecutará las funciones primitivas. Este medio se implementa como una

instancia de una clase que contiene la lista de funciones primitivas en forma de métodos, cuyas funciones a realizar cada uno de ellos dependerá del almacenamiento primitivo.
- El plan físico de consulta se pasa al motor de ejecución.
- Dependiendo del tipo de objeto encontrado en la raíz del plan físico, el motor de ejecución llama el método correspondiente para iniciar el proceso de ejecución. Se pasa a ese método la instancia de la clase del motor de ejecución como parámetro, para que todo el plan físico maneje el mismo motor de ejecución y, por consiguiente, el mismo espacio de memoria y los datos temporales.
- El motor de ejecución devuelve los resultados.
- Se borran los datos temporales materializados.

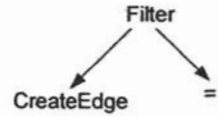
Ejemplo 6.4.5 La ejecución del plan físico del ejemplo 6.4.4, después de las inicializaciones realizadas dentro del motor de ejecución, se efectúa de la siguiente forma:

- El motor de ejecución analiza el tipo del objeto raíz del plan físico y si éste es de tipo `Get` (`CreateBush` hereda de la clase `Get`), significa que se trata de una consulta que devolverá datos. Por ello, llama al método `getids` de ese objeto raíz para iniciar la ejecución.

- `ForEachDO`.- Por cada dato que le proporcione `Filter`, ejecutará el método `getids` de `CreateEdge`. Todos los datos recuperados de este último se regresarán a `CreateBush` como una colección.



  - Recuperar los datos de `Filter`.- Por cada dato devuelto por `CreateEdge`, se revisará si se cumple la condición S = 0 con el estado actual de la base de datos; si es así, entonces devuelve lo obtenido del objeto `CreateEdge`.



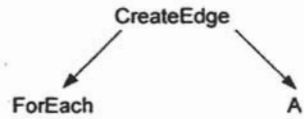
- Recuperar datos de `CreateEdge`.- Aquí inicia el proceso definido por el operador *DJoin* del plan lógico, el cual puede observarse en la segunda figura del ejemplo 6.4.4. Por cada dato semiestructurado *D* devuelto por el objeto `ForEach`, se creará una arista temporal con etiqueta "S" apuntando a *D*

  - Recuperar los datos de `ForEach`: Es un iterador que va devolviendo dato por dato recuperado del objeto descendiente de él, el cual, en este caso, es `GetChildren`.

  - Recuperar los datos de `GetChildren`.- Por cada dato *F* recuperado por el objeto de la izquierda (`GetData`) recuperar aquellos datos conectados a *F* con una arista etiquetada como "stock".

    - Recuperar los datos de `GetData`.- En este caso, existe un objeto descendiente de este (`CreateEdge`), por lo que tiene que ejecutar su método `getids` antes de intentar recuperar el dato semiestructurado al que apunta la Ssd-tabla "A".

      - Recuperar los datos de `CreateEdge`.- Con el dato *D* devuelto por el objeto `ForEach` se creará una arista temporal con etiqueta "A" apuntando a *D*.



- Recuperar los datos de `ForEach`.- Devuelve dato por dato regresado por su objeto descendiente.

- Recuperar los datos de `GetChildren`:

  - Recuperar los datos de `ForEach`:

  - Recuperar los datos de `GetChildren`:

    - Recuperar los datos de `GetData`:

      Este objeto es una hoja dentro del plan físico, el cual recupera el identificador 18 que es la raíz de la tabla "inventario-".

      `GetChildren`.- Recupera la colección de todos los identificadores de los datos conectados al nodo con identificador 18. Regresa la colección {20,21,22} (El nodo 19 no se considera al utilizar comodines (#) como en este caso).

`ForEach`.- Regresaría el nodo 20, pero no cumple con la condición; para acortar la descripción se supondrá que ya se han devuelto los otros identificadores y se pasará directamente al último, el cual es el que aporta datos al resultado, que es el 22.

`GetChildren`.- El nodo con identificador 22 tiene una arista con etiqueta "*Data-*", la cual apunta al nodo con identificador 4. Regresa la colección {4}.
`ForEach`: Regresa el identificador 4.

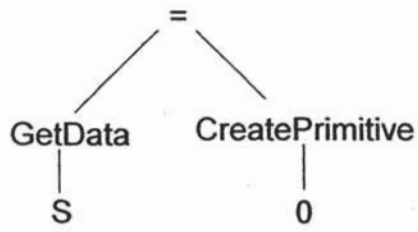
`CreateEdge`.- Crea una Ssd-tabla temporal con etiqueta "A" apuntando al nodo con identificador 4.

`GetData`.- Recupera el nodo raíz de la Ssd-tabla "A", que es el identificador 4.

`GetChildren`.- A partir del nodo con identificador 4, se buscan los que se encuentren conectados a éste con una arista etiquetada como "stock", que es el nodo con identificador 14.

`ForEach`.- Devuelve el identificador 14.
`CreateEdge`.- Crea una Ssd-tabla temporal con etiqueta "S" que apunta al nodo con identificador 14.

`Filter`.- Se evalúa la condición:



= Debe recuperar los identificadores de sus operandos:

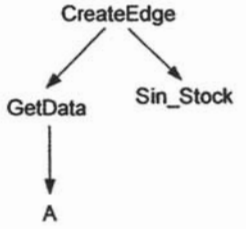
`GetData`.- Recupera el identificador 14 del nodo raíz de la Ssd-tabla S.

`CreatePrimitive`.- Crea un nodo primitivo temporal con valor 0. Devuelve el identificador 44 asignado al nuevo nodo.

Se evalúa la condición; como ambos nodos son primitivos y números, se recupera el valor primitivo de cada uno y debido a que ambos son cero, la condición es verdadera por lo que se devuelve el nodo con identificador 14.

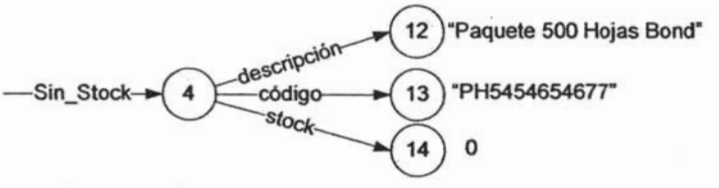
`ForEachDO`.- Una vez obtenidos los datos de `Filter`, se procede a construir el resultado con el objeto `CreateEdge`.

`CreateEdge`.- Crea una nueva arista temporal con etiqueta "Sin_Stock" apuntando al nodo regresado `GetData`.



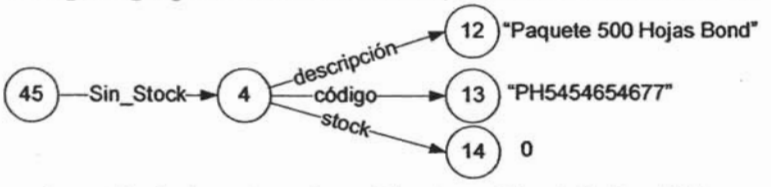
`GetData`.- Recupera el identificador 4 del nodo raíz de la Ssd-tabla "A".

`CreateEdge`.- Crea y devuelve la arista temporal "Sin_Stock" apuntando al nodo con identificador 4:



`ForEachDO`.- Almacena lo que regresó `CreateEdge`. Después de esto, pide al objeto de la izquierda `Filter` más datos y así sucesivamente, pero si ningún objeto tiene más datos para regresar, la ejecución de los objetos descendientes de `Filter` se da por terminada. Entonces, `ForEachDO` devuelve las parejas almacenadas, las cuales son el resultado, que en este caso sólo fue una pareja.

`CreateBush`.- Para todos los datos obtenidos por `ForEachDO` se crea un nuevo nodo que agrupe a todos esos datos, creando un nuevo nodo 45:



el cual es el resultado terminando así la ejecución del plan físico

Con el ejemplo 6.4.5, puede observarse que la ejecución del plan físico de consulta es semejante a la ejecución descrita en el capítulo 1. La diferencia en este ejemplo es cómo inicia el recorrido del árbol desde una Ssd-tabla distinta, la del *resumen de datos con identificadores*.

La ejecución de otro tipo de consultas es similar, se evalúan los objetos hijos antes de hacer la correspondiente del objeto en sí. Para observar los diferentes planes lógicos y físicos creados con otro tipo de consultas (actualización, borrado), se recomienda al lector acudir al procesador de consultas desarrollado con esta tesis.

6.5 Resumen y discusión

El motor de ejecución es otro de los componentes principales del sistema administrador de bases de datos, pues es el que pone en marcha el plan físico de consulta. Su labor central es manejar todo el ambiente en el que se ejecuta una consulta: los nodos temporales, Ssd-tablas temporales y los componentes materializados.

En este capítulo se presentó el algoritmo para la materialización de datos semiestructurados temporales. Este tipo de datos, después de finalizar la ejecución de la consulta, deben ser eliminados de la base de datos física, por lo que también de desarrolló la técnica a seguir para hacerlo.

Existen otros componentes que cooperan con el motor de ejecución (control de concurrencia, transacciones, etc.), con los que no se estableció la conexión, pues al momento de escribir esta tesis y desarrollar el motor no se cuenta con ellos.

Contar con el motor de ejecución permitió realizar pruebas del comportamiento del procesador de consultas en un ambiente más cercano al de un sistema administrador de bases de datos de producción. En el apéndice B se presentan una serie de pruebas enfocadas a este punto.

Conclusiones

El trabajo fundamental de un procesador de consultas es transformar una consulta a una serie de instrucciones llamadas primitivas, propias del almacenamiento interno de la base de datos. En el caso de esta tesis, el enfoque se centró en el ambiente de datos semiestructurados. En base a esto, se utilizó un lenguaje especialmente diseñado para el manejo de este tipo de datos como medio para expresar la consulta y se emplearon primitivas de un almacenamiento interno desarrollado concretamente para guardar dichos datos.

Con el propósito de realizar la transformación, se diseñó una arquitectura interna del procesador de consultas inspirada en los sistemas de bases de datos relacionales y en los compiladores de lenguajes de programación. Por ello, fue necesario definir una primera transformación a la consulta, el plan lógico, de forma que con éste fuera posible identificar aquellas oportunidades para optimizarla. Estas optimizaciones se concretaron después de un largo proceso de investigación documental y tienen como base trabajos que han hecho aportes en este campo, como los de la Universidad de Stanford. El plan lógico pasa después a un plan físico, que toma en consideración las instrucciones del almacenamiento primitivo.

Para obtener el plan lógico, era necesario contar con un lenguaje lógico que manejara datos semiestructurados lo suficientemente expresivo para representar cualquier acción sobre ellos al mismo nivel que lo hace una consulta escrita en el lenguaje Ssquirel. Al observar carencia en la literatura de un lenguaje con estas características —pues, o bien no se definían formalmente (lo que deja lugar a interpretaciones erróneas) o sólo abarcaban consultas para obtención de datos, sin considerar aspectos como borrar o actualizar— surge la necesidad de crear un instrumento que diera respuesta a tal limitante. Ante esta circunstancia, parte esencial de este trabajo de tesis fue diseñar un lenguaje lógico que abarcara todas las posibles consultas, mismo que se propone como una herramienta formal para la investigación de optimizaciones para consultas que manejen datos semiestructurados, adicionales a las que aquí se plantean.

Para desarrollar las optimizaciones del plan lógico fue necesario un análisis profundo del modelo de datos semiestructurados, pues éste posee una característica que difícilmente puede alcanzarse en otros modelos, que es la capacidad de manejar datos que tienen una estructura variable o bien parcialmente desconocida. El manejo de estas fuentes de información es posible hacerlo de forma más natural mediante un modelo de datos semiestructurados, en vez de tratar de amoldar otro modelo para manejar esta información.

En otro tipo de bases de datos el esquema que deben seguir los datos se determina antes de ingresarlos. Pero, en el modelo de datos semiestructurados este no es el caso, pues al ser autodescriptivos, los datos definen su estructura en sí, liberando al desarrollador de la tarea de modelar la base de datos. Sin embargo, esto hace que el trabajo necesario para recuperar datos se convierta en un proceso más costoso para el sistema administrador de bases de datos semiestructurados. Por ello, fue necesario trabajar en maneras para optimizarlo y hacerlo más eficiente.

A través del desarrollo de esta tesis se ha visto que para agilizar la recuperación de información desde una base de datos semiestructurados es necesario contar con un esquema que ayude a determinar la localización y existencia de datos específicos. Para dar solución a ello y así ofrecer un mecanismo de optimización a una consulta, se desarrolló la técnica del *resumen de datos con identificadores*. Esta técnica tiene su base en la creación de un esquema dinámico que se actualiza al modificar la base de datos, para que de esta forma siempre contenga una fiel representación de la estructura de los datos.

Mejorar el plan físico, el cual se basa en la creación de datos temporales y recuperación de hijos, necesitó desarrollar un iterador para la devolución de datos, de modo que los requerimientos de espacio en memoria y, por consiguiente, la materialización de datos se vean reducidos.

Con el fin de realizar el análisis léxico y sintáctico de la consulta, se empleó la herramienta para construcción de compiladores JavaCC, la cual, unida con JJTree, facilitó en la elaboración de estas etapas y el desarrollo de las siguientes del procesador de consultas.

Todo lo anterior permitió alcanzar el propósito central de esta tesis de construir un procesador de consultas para datos semiestructurados, generando para el área de bases de datos las siguientes aportaciones:

- Definición de un lenguaje lógico para el manejo de datos semiestructurados.

  Los operadores para este lenguaje fueron definidos formalmente. Además, se incluye la propuesta de operadores innovadores que consideran la modificación de datos semiestructurados.

- Mejoramiento al algoritmo de resolución de disyuntivas en expresiones de camino.

  Cada opción se divide en varias expresiones y también se separa el recorrido común entre ellas. Esto se logra mediante el uso de una tabla temporal extra, con la que inician las expresiones de cada opción.

- Introducción de la técnica del *resumen de datos con identificadores* para la recuperación de información.

  Se mostró que utilizar el resumen para recuperar la información de aquellas expresiones de camino que inicien con la etiqueta de una Ssd-tabla reduce significativamente el número de accesos a disco. Esto resulta porque el sistema administrador no tiene que considerar cada ramificación en el grafo de datos original. De este modo se obtiene un mayor ahorro pues no se recorren aquellas ramificaciones en las que no se tiene la seguridad que llegarán al final del camino definido por la expresión de camino.

- Elaboración de un *resumen de datos con identificadores* de fácil actualización.

  Al realizar ligaduras entre el resumen y los datos en sí es posible encontrar cuáles resúmenes datos y en qué punto de ellos afecta una actualización. De igual forma, el proceso de borrado definido por el operador lógico *Delete* actualiza automáticamente el *resumen de datos con identificadores* debido al uso de estas ligaduras.

- Definición de la traducción del plan lógico a físico.

Se definió el esquema de traducción para realizar este paso, el cual se basa en relacionar cada operador lógico con la o las funciones físicas que lo representan.

- Aplicación de la técnica *Pipelining* a consultas de datos semiestructurados.

  Al observar la cantidad de nodos temporales creados durante la evaluación de una consulta, se dio pauta para describir cómo y dónde implementar la técnica *Pipelining*. Al hacerlo, se eliminó la necesidad de crear varios de estos componentes temporales.

- Definición de algoritmos para la materialización y borrado de datos semiestructurados temporales.

  El motor de ejecución usa estos algoritmos para el manejo de datos temporales y cumple, de este modo, con una parte esencial de su trabajo.

En la integración del procesador de consultas al resto de los componentes del sistema administrador, se tuvo un avance que consistió en el desarrollo del motor de ejecución. Con ésto fue posible realizar pruebas del comportamiento del procesador de consultas. Con ello, se pudo observar el impacto que tiene sobre el tiempo de ejecución al utilizar o no el *resumen de datos con identificadores* para recuperar la información solicitada por las expresiones de camino. Estas pruebas se presentan en el Apéndice B.

Perspectivas

Las optimizaciones elaboradas en este trabajo se enfocaron a la recuperación de datos semiestructurados por parte de las expresiones de camino. Sin embargo, en materia de optimizaciones, siempre existe la posibilidad de investigar y concebir nuevas técnicas que ayuden a mejorar la ejecución de una consulta. Con base en la teoría aquí presentada, llámese lenguaje lógico e instrucciones físicas, es posible continuar esfuerzos orientados a perfeccionar este procesador de consultas.

El alcance de las pruebas efectuadas en este trabajo (vea apéndice B) sólo llega a comparativas entre el mismo sistema. Comparar qué tan rápido o lento es el sistema administrador de bases de datos semiestructurados —del que es parte el procesador de consultas— contra otros sistemas, es motivo para un trabajo futuro. Se pueden realizar pruebas entre sistemas que manejen el mismo tipo de datos (semiestructurados), como por ejemplo el sistema Lore [MAG+97]. Sin embargo, también sería interesante comparar el sistema de esta tesis contra un sistema administrador de bases de datos relacional, para observar la diferencia en la velocidad de procesamiento.

Realizar la comparación contra otros sistemas administradores no fue posible durante el tiempo de realización de esta tesis, pues el almacenamiento primitivo todavía está en fase de desarrollo.

Aún quedan por explorar otros componentes para tener un sistema administrador de bases de datos semiestructurados robusto, que atienda las necesidades de acceso y seguridad útiles para las aplicaciones de hoy en día, muchas de las cuales ya las cumplen los sistemas administradores de bases de datos relacionales comerciales.

Referencias Bibliográficas

[AbBS00] Abiteboul S., Buneman P., Suciu D. "Data on the Web, From Relations to Semistructured Data and XML", *editorial Morgan Kaufmann Publishers*, 2000.

[Abit97] Abiteboul S. "Querying Semi-Structured Data". *In Proceedings of the International Conference on Database Theory, pp. 1-58,* Delfi, Grecia, Enero 1997.

[AGM+97] Abiteboul S., Goldman R., McHugh J., Vassalos V., Zhuge Y. "Views for Semistructured Data". *In Proceedings of the Workshop on Management of Semistructured Data, pp. 83-90,* Tucson, Arizona, Mayo 1997.

[AhSU90] Aho A.V., Sethi R., Ullman J.D. "Compiladores: Principios, técnicas y herramientas", *editorial Addison Wesley*, 1990.

[AQM+97] Abiteboul S., Quass D., McHugh J., Widom J., Wiener J. "The Lorel Query Language for Semistructured Data". *International Journal on Digital Libraries, volumen 1, número 1, pp. 68-88*, April 1997.

[BDHS96] Buneman P., Davidson S., Hillebrand G., Suciu D. "A Query Language and Optimization Techniques for Unstructured Data". *In Proceedings of the 1996 ACM SIGMOD Internacional Conference on Management of Data, volumen 25, número 2 SIGMOD Record, pp. 505-516, editorial ACM Press*, 1996.

[BeAn03] Berk E.J., Ananian C.S. "JLex: A Lexical Analyzer Generator for Java". *http://www.cs.princeton.edu/~appel/modern/java/JLex/* Revisado el 26 de Abril de 2005.

[BeTz99] Beeri, C., Tzaban, Y. "SAL: An Algebra for Semistructured Data and XML". *In Proc. ACM SIGMOD Workshop on The Web and Databases (WebDB'99), pp.37-42, editorial ACM Press*, 1999.

[Bune97] Buneman P. "Semistructured data". *In Proc. of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 117-121*, Tucson, Arizona, *editorial ACM Press*, Mayo 1997.

[Cart00] Cartwright R. "Notes on Object-Oriented Program Design". *http://www.cs.rice.edu/~cork/book/*
Revisado el 26 de Abril de 2005.

[Catt94] Cattell R. G. G. "The Object Database Standard: ODMG-93". *Editorial Morgan Kaufmann Publisher*, San Francisco, 1998.

[ChCM96] Christophides V., Cluet S., Moerkotte G. "Evaluating Queries with Generalized Path Expressions". *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, pp. 413-422, editorial ACM Press*,1996.

[DaGK82] Dayal U., Goodman N., Katz R.H. "An extended relational algebra with control over duplicate elimination". *Editorial ACM Press*, NY USA, 1982.

[Deve02] DeveloperWorks. "Understanding SAX". *http://ibm.com/developerWorks*
Revisado el 26 de Abril de 2005.

[FeSu98] Fernández M., Suciu D. "Optimizing Regular Path Expression Using Graph Schemas (full, revised version)". *AT&T Labs*. 1998.

[FrHP02] Frasincar F., Houben G, Pau C. "XAL: an Algebra for XML Query Optimization". *In Database Technologies 2002, Thirteenth Australasian Database Conference in Research and Practice in Information Technology, volumen 5, pp. 49-56.* Sociedad Australiana en Computación, 2002.

[Garc02] García Cárdenas E. A. "Un lenguaje de consulta para Bases de Datos Semiestructurados". *Tesis de licenciatura*, Facultad de Ciencias, UNAM, 2002

[Gars03] Garshol L. M. "BNF and EBNF: What are they and how do they work?". *http://www.garshol.priv.no/download/text/bnf.html* Revisado el 26 de Abril de 2005.

[GaUW00] García-Molina H., Ullman J.D., Widom J. "Database System Implementation". *Editorial Prentice Hall,* 2000.

[GoWi97] Goldman R., Widom J. "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases". *In Proc. of the Int'l Conf. on Very Large Databases, pp. 436-445,* 1997.

[Huds99] Hudson S. "CUP Parser Generator for Java". *http://www.cs.princeton.edu/~appel/modern/java/CUP/* Revisado el 26 de Abril de 2005.

[Katz02] Katz H. "JavaCC, parse trees, and the XQuery grammar". *http://www-106.ibm.com/developerworks/xml/library/x-javacc1.html* Revisado el 1 de Diciembre de 2002.

[Katz03] Katz H. "Implementing the XQuery grammar". *http://www.fatdog.com/Extreme.html* Revisado el 26 de Abril de 2005.

[LeYa05] Lex Manual Page. "The Lex & Yacc Page". *http://dinosaur.compilertools.net/* Revisado el 26 de Abril de 2005.

[Loud04] Louden K.C. "Construcción de compiladores, principios y práctica". *Editorial Thomson,* 2004.

[Lund03] Lundberg J. "Getting Started with JavaCC". *Institute of Mathematics and System Engineering*, Växjö University, Septiembre 2003.

[MAG+97] McHugh J., Abiteboul S., Goldman R., Quass D., Widom J. "Lore: A database management system for semistructured data". *Revista SIGMOD Record, volumen 26, número 3, pp. 54-66, editorial ACM Press,* Septiembre 1997.

[Mart02] Martin R. C. "The Principles, Patterns and Practices of Agile Software Development". *Editorial Prentice Hall, primera edición*, Octubre 2002.

[McHu00] McHugh J.G. "Data Management and Query Processing for Semistructured Data". *Tesis doctoral.* Stanford University, Marzo 2000.

[McWi97] McHugh J., Widom J. "Query Optimization for Semistructured Data". *Technical Report No. 4*, Stanford University, 1997.

[McWi99] McHugh J., Widom J. "Compile-Time Path Expansion in Lore". *In Proceeding of the Workshop in Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, Enero 1999.

[McWi99a] McHugh J., Widom J. "Optimizing Branching Path Expressions". *Technical Report*, Standford University, Junio 1999.

[Norv02] Norvell T. S. "The JavaCC FAQ". *http://www.idevelopment.info/data/Programming/java/JavaCC/The_JavaCC_FAQ.htm* Computer and Electrical Engineering Memorial University of Newfoundland. Revisado el 26 de Abril de 2005.

[PaGW95] Papakonstantiou Y., Garcia-Molina H., Widom J. "Object Exchange across heterogeneous information sources". *In Intl. Conf. on Data Engeneering, pp. 251-260,* Taipei, Taiwan, 1995.

[SiKS02] Silberschatz, A., Korth, H.F., Sudarshan, S. "Fundamentos de Bases de Datos". *4ª edición, editorial McGraw-Hill*, 2002.

[Suci98] Suciu D. "Semistructured Data and XML". *In Proceedings of International Conference on Foundations of Data Organization*, 1998.

[Suci98a] Suciu D. "An overview of semistructured data". *SIGACT New, volumen 29, número 4*, pp. 28-38, Diciembre 1998.

[SunM03] Sun Microsystems. "javacc: JavaCC Home". *https://javacc.dev.java.net/* Revisado el 26 de Abril de 2005.

[SunM04] Sun Microsystems. "Class Pattern". *http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html* Revisado el 26 de Abril de 2005.

[SunM05] Sun Microsystems. "JDBC Technology". *http://java.sun.com/products/jdbc/.* Revisado el 26 de Abril de 2005.

[UlWi99] Ullman, J.D., Widom, J. "Introducción a los Sistemas de Bases de Datos". *Editorial Prentice Hall*, 1999.

[W3C99] World Wide Web Consortium (W3C) "Recommendation. XML Path Language (Xpath)". *Versión 1.0.* 16 noviembre 1999.

[W3C04] World Wide Web Consortium (W3C). "XQuery 1.0: An XML Query Language". *http://www.w3.org/TR/2005/WD-xquery-20050404/* Revisado el 26 de Abril de 2005.

[Will02] Williams D. M. "Implement Persistent Objects with Java Serialization". *http://www.devx.com/Java/Article/9931* Revisado el 26 de Abril de 2005.

Apéndice A

Análisis léxico y sintáctico

Actualmente, existen herramientas de *software* que ayudan a generar este tipo de analizadores, a las cuales se les conoce como compiladores para compiladores. Su función es convertir expresiones regulares (para la parte léxica) y una gramática (para la parte sintáctica) a código en un lenguaje de alto nivel. Este último debe compilarse para obtener el programa ejecutable que reconozca si una sentencia pertenece al lenguaje descrito por las expresiones regulares y la gramática.

Dos de las primeras herramientas de este tipo que tuvieron gran aceptación fueron Lex y Yacc [LeYa05]. Éstas transformaban sus entradas —expresiones regulares para Lex, una gramática para Yacc— en código fuente escrito en lenguaje C. Cada una de ellas generaba su propio programa: Lex, un analizador léxico, reconoce los tokens de una secuencia de caracteres; Yacc, un analizador sintáctico, verificar si una secuencia de tokens pertenece a la gramática que se compiló con él. Era tarea del programador unir los dos programas anteriores, de tal forma que la entrada de Lex sea una secuencia de caracteres que los convierte a una secuencia de tokens, misma que debe corresponder a la entrada del programa generado por Yacc.

Para realizar una traducción utilizando herramientas como Lex y Yacc, era necesario mezclar el código C dentro de la gramática del lenguaje, de tal manera que al aceptar una producción se ejecuta el código incrustado. Lo anterior crea una unión entre el análisis sintáctico y el proceso de traducción en sí. Por lo tanto, no es posible hacer una separación entre las fases del compilador y, en consecuencia, depurarlo y mantenerlo se convierte en un proceso tedioso y, en muchas ocasiones, difícil de realizar.

A.1 JavaCC

Para el procesador de consultas que se desarrolló con esta tesis, se planeó utilizar el lenguaje de alto nivel Java. Por ello, se requirió realizar una investigación de las herramientas existentes para crear analizadores léxico y sintáctico que utilicen Java como lenguaje destino.

Jlex [BeAn03] y CUP [Huds99] son herramientas para el lenguaje Java lo que Lex y Yacc son para C, respectivamente. Por tal motivo, caen en los mismos problemas, como son: la necesidad mezclar código entre la definición de la gramática; la conexión entre ambos analizadores es tarea del programador; no generan automáticamente un árbol de análisis sintáctico.

Existe una herramienta que ofrece varias características que ayudan al desarrollo de un compilador, la que, a diferencia de las mencionadas que sólo auxilian en la creación de la parte léxica y semántica, ésta brinda una plataforma de trabajo para el desarrollo de compiladores. Esta herramienta se le conoce como JavaCC[Norv02, SunM03].

JavaCC tiene las siguientes características:

- Analizador léxico especificado mediante expresiones regulares.
- La sintaxis de la gramática que reconoce es una extensión de la notación BNF (forma de Backus-Naur).
- Tiene la posibilidad de generar automáticamente el árbol de análisis sintáctico.
- Ofrece una plataforma orientada a objetos para el manejo de los árboles de análisis sintáctico, que permite realizar una traducción basada en dichos árboles.
- Las especificaciones léxicas y gramáticas se definen en un solo archivo; es decir, la misma herramienta maneja la unión de los analizadores léxico y sintáctico.
- Internacionalización; puede reconocer caracteres de otros países, por ejemplo, las vocales acentuadas en el caso del español, algo que no es muy común en estas herramientas.
- Buena documentación y ejemplos.
- Es una herramienta libre.

JavaCC, a partir de la gramática, construye un analizador sintáctico LL(k>1), esto es, realiza un análisis sintáctico descendente. Este tipo de analizadores no permite utilizar gramáticas recursivas por la izquierda y, como se verá a continuación, la gramática que describe al lenguaje Ssquirel tiene varias partes que son de este modo. Por lo tanto, esto debe eliminarse.

Gracias a que JavaCC da posibilidad de utilizar k símbolos de entrada antes de tomar una decisión del análisis sintáctico, o qué derivación elegir, es posible manejar aquellas partes de la gramática de Ssquirel donde existen dos o más derivaciones con inicio común. Sin embargo, esto hace lento el proceso de análisis sintáctico, pues debe avanzar hasta k símbolos en una producción, antes de determinar si es o no la producción correcta a seguir; si no lo es, debe retroceder nuevamente los k símbolos que había avanzado. JavaCC tiene preestablecido una k=1, esto es, con un sólo símbolo debe determinarse qué derivación seguir, pero para aquellas producciones en donde es necesario más de un símbolo, es posible señalarle a JavaCC hasta cuántos símbolos utilizar antes de tomar una decisión. Esto último se especifica con un comando especial llamado LOOKAHEAD dentro de la gramática, antes de listar las producciones que harán uso de tal característica.

A.1.1 Sintaxis

Las expresiones regulares se escriben con una concatenación de caracteres utilizando signos especiales (*, +, ?, |), todo encerrado en una definición de un token, lo cual etiqueta a la expresión regular, por ejemplo:

```
ab*
```

que representa al lenguaje de cadenas de caracteres que inician con a y terminan con cero o más b's. Esta expresión regular se escribe en JavaCC como:

```
TOKEN: { <ABS : "a"("b")* > }
```

etiquetándola con el nombre de ABS. Es necesario utilizar comillas para todo símbolo que pertenezca al alfabeto del lenguaje que reconoce la expresión regular. El ejemplo anterior es una de las varias formas en las que puede especificarse un token,

Para la gramática, JavaCC define un tipo especial de sintaxis, parecido a declarar métodos de las clases Java. También es posible utilizar la notación extendida de BNF [Gars03], con los operadores *, +, [] para especificar la multiplicidad de un elemento.

Para una gramática escrita en notación BNF:

```
<exp>       ::=   <exp> + <term> | <term>
<term>      ::=   <term> * <term> | <fact>
<fact>            ::=   <dígitos>
```

se transforma en notación JavaCC como:

```
TOKEN : { < DIGITOS : ([0..9])+ >}
void exp() : {}   {
      exp() "+" exp() |
      term()
}

void term() : {} {
term() "*" term() |
fact()
}

void fact() : {} {
<DIGITOS>
}
```

Para más detalles sobre la sintaxis de JavaCC existe documentación muy completa en [SunM03, Lund03].

A.2 JJTree

Por sí sólo, JavaCC no cuenta con la capacidad de crear un árbol de análisis sintáctico, entonces habría que construir la traducción o un árbol en sí, incrustando código Java dentro de la gramática del lenguaje. Como se mencionó, lo anterior genera un problema de depuración y mantenimiento del programa debido a que no existe una separación en el código del análisis sintáctico y el de las fases siguientes.

Para resolver este problema, JavaCC cuenta con un preprocesador llamado JJTree, el cual inserta automáticamente el código de creación del árbol de análisis sintáctico dentro de la gramática del lenguaje. JJTree tiene la ventaja de que su entrada puede ser exactamente igual a la que utiliza JavaCC, por lo que no hay necesidad de realizar cambios al archivo de especificación para poder utilizarlo con JJTree. Sin embargo, JJTree agrega otros comandos especiales que pueden escribirse dentro de la gramática, los cuales se emplean para controlar la creación del árbol de análisis sintáctico.

JJTree, de manera predeterminada, genera un código para crear nodos del árbol por cada no terminal en la gramática del lenguaje. Este comportamiento puede modificarse con los comandos propios de la herramienta. JJTree define una interfaz *Node*, implementada por todos los nodos del árbol, cuya función es proveer métodos para agregar hijos al nodo, especificar el padre del nodo, etc.

Si el comportamiento de crear un nodo por cada no terminal no se modifica, entonces todos los nodos serán de tipo `SimpleNode`, que es una clase proporcionada por JJTree que implementa a la interfaz *Node*. Sin embargo, cuando es conveniente crear una clase para un determinado no terminal, hay que especificar para cual se generará el código de creación del nodo y el nombre de la clase a la que pertenecerá. Por ejemplo, para una definición de un no terminal como:

```
void NT() : { … } { … }
```

se agrega el nombre de la clase después del nombre del no terminal:

```
void NT() #NoTerminal : { … } { … }
```

de esta forma se genera una nueva clase ASTNoTerminal la cual servirá para crear los nodos que representen al no terminal NT. Esta clase debe implementar los métodos de *Node*, por lo que hereda de la clase `SimpleNode` que ya los define.

Para el analizador sintáctico utilizado en el procesador de consultas, se estudió cuáles no terminales necesitaban la construcción de un nodo y, por consiguiente, de una clase, de manera que en cada una de ellas se definiera el código para las transformaciones siguientes de la consulta. La sección A.5 presenta la gramática final.

Al utilizar JJTree, la generación del analizador léxico y sintáctico pasa a ser un proceso de tres etapas:

- Se compila el archivo de especificación con JJTree. (archivo.jjt)
- La salida de JJTree (archivo.jj) se compila con JavaCC.
- La salida de JavaCC (archivo.java) es el código del analizador en Java, el cual se compila, junto con todas las clases derivadas de JJTree, con algún compilador de Java para obtener el ejecutable. (archivo.class)

A.3 Gramática del lenguaje Ssquirel

La figura A.3 presenta la gramática original del lenguaje Ssquirel. Como puede observarse, la parte más complicada de manejar para un analizador sintáctico es la producción "*ssd-construction*". Además, es la de mayor uso dentro de la gramática, pues se ocupa para representar un dato semiestructurado mediante el lenguaje Ssquirel y con ello poder crearlos o especificarlos.

```
<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> TRIM ( <list-labels> ) |
```

```
                                { <list-labeled-ssds> } |
                                CLON <ssd-construction> |
                        <aggregation-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>
<aggregation-function>::=       AVG |
                                MIN |
                                SUM |
                                COUNT
<query> ::=     SELECT <label> : <ssd-construction>
                FROM <list-associations> |
                SELECT <label> : <ssd-construction>
                FROM <list-associations>
                WHERE <condition> |
                SELECT DISTINCT <label> : <ssd-construction>
                FROM <list-associations> |
                SELECT DISTINCT <label> : <ssd-construction>
                FROM <list-associations>
                WHERE <condition>

<delete-statement>      ::=     DELETE <ssd-table-name>
                                FROM <list-associations> |
                                DELETE <ssd-table-name>
                                FROM    <list-association>
                                WHERE <condition>
<update-statemet>       ::=     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>        ::=     <view-definition-statement> |
                                <drop-view-statement>

<view-definition-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 <sdd-table-name> |
       <path-expression> AS <sdd-table-name> , <list-associations>

<condition>::= TRUE |
               FALSE |
               ( <condition> ) |
               <condition> AND <condition> |
               <condition> OR <condition> |
               NOT <condition> |
               <sdd-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-expresssion>      ::=     <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>       ::=     "<characters>" | ""

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

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

<characters>        ::=     <letter> |
                            \" |
                            <letter><characters> |
                            \"<characters>

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

<digit> ::=         [0-9]
```

Figura A.3 *Gramática del lenguaje Ssquirel*

La gramática de la figura A.3 está en notación BNF, pero no extendida, por lo que producciones como <x-label> :: = <x-label>* no significan que puede haber cero o más repeticiones del no terminal <x-label>, sino que al no terminal <x-label> le prosigue el caracter "*".

Las últimas producciones, desde "*numeric-constant*", se implementaron como parte del analizador léxico, así como todas las palabras reservadas (escritas en mayúscula) del lenguaje.

Además de la producción "*ssd-construction*", también son recursivas "*path-expression*", "*condition*" y "*x-label*", por lo que deben modificarse para que las acepte JavaCC como parte de la gramática. De igual forma, se realiza una factorización por la izquierda para utilizar lo menos posible el comando LOOKAHEAD que proporciona JavaCC.

A.4 Manejo de precedencia y asociatividad de operadores

Los problemas de recursión por la izquierda y de elección de producción son muy conocidos en la literatura [AhSU90, Loud04] al igual que los algoritmos de eliminación de recursión por la izquierda y factorización por la izquierda. Sin embargo, la gramática obtenida después de realizar ambos algoritmos, no contempla la precedencia y asociatividad de los operadores utilizados por Ssquirel. La literatura [AhSU90 ejemplo 2.1] da un ejemplo simple de cómo modificar una gramática para que ésta genere un árbol de análisis sintáctico que respete las reglas de precedencia y asociatividad. De esta forma, se facilita el trabajo posterior al tener conocimiento que la gramática no creará ningún árbol que no respete dichas reglas y que, además, su estructura las contemplará para su formación. Por ejemplo, la operación "*4 + 3 * 2*" puede generar dos árboles, los de la figura A.2; de los cuales el A.2(a) es el correcto, pues respeta las normas establecidas para la resolución de operaciones aritméticas.

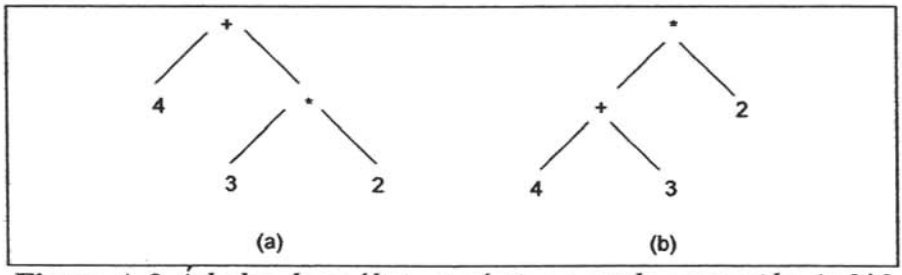


Figura A.2 *Árboles de análisis sintáctico para la operación 4+3*2*

Al obtener un árbol de análisis sintáctico que considere las reglas de precedencia y asociatividad, es posible realizar una evaluación simplemente haciendo un recorrido postorder, es decir, evaluando los descendientes antes de la raíz.

La tabla A.1 contiene la precedencia y asociatividad para los operadores utilizados por la producción "*ssd-construction*". La tabla A.2 define dichas reglas de los operadores utilizados por la producción "*condition*". Para las expresiones de camino representadas por las producciones "*path-expression*" y "*x-label*", los operadores *, + y ? son los de mayor precedencia (se deben ejecutar antes que los demás), le sigue el operador · y el de menor es el operador |.

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

Tabla A.1 *Precedencia y asociatividad de operadores*

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

Tabla A.2 *Precedencia y asociatividad de los operadores condicionales*

La documentación [Katz02, Katz03] trata este problema específicamente para JavaCC y proporciona una solución para modificar una gramática realizando lo siguiente:

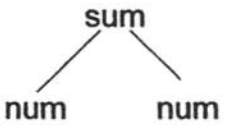
- Eliminar la recursividad por la izquierda
- Factorizar por la izquierda
- Considerar las reglas de precedencia y asociatividad

La solución consiste en aprovechar la estructura de datos que utiliza JJTree al generar el árbol de análisis sintáctico. Internamente, JJTree guarda en una pila los nodos construidos, de manera que al crear el nodo padre, éste recupera sus hijos de dicha pila. Por ejemplo, con la gramática siguiente:

```
TOKEN : < numeros : (["0".. "9"])+>

void suma() #suma : {} { num() "+" num() }
void num() #num : {} { <numeros> }
```

Cuando se elabora el árbol para la expresión 4+7, el nodo "suma" tomará los nodos construídos por los dos no terminales num:



Extendiendo la misma gramática para sumas y multiplicaciones, con lo cual debe manejarse la precedencia de dichos operadores, partiendo de la gramática:

```
TOKEN : < numeros : (["0".. "9"])+>
void suma() #suma : {} { num() ("+"|"*") num() }
void num() #num : {} { <numeros> }
```

que no considera las reglas de precedencia, pues crea un árbol como el de la figura A.2(b), se transforma en:

```
TOKEN : < numeros : (["0".. "9"])+>
void suma() : {} { mult() ("+" mult() #suma(2) )* }
void suma() : {} { num() ("*" num() #mult(2) )* }
void num() #num : {} { <numeros> }
```

la que obliga a cada operador a tener sus dos operandos antes de crear el nodo correspondiente, tomándolos de la pila que almacena los nodos producidos con anterioridad, lo que hace posible elaborar correctamente el árbol de análisis sintáctico, tal y como lo muestra la figura A.2(a). Para más información acerca del manejo de la precedencia y asociación con JJTree, la referencia [Katz03] ofrece varios ejemplos y una explicación más detallada al respecto.

Aplicar esta técnica a una gramática tan extensa y con tantos operadores como la de Ssquirel es una tarea difícil, pero vale la pena pues facilita la traducción de la consulta en etapas posteriores.

A.5 Gramática final en notación de JJTree

A continuación se presenta la gramática final utilizada por el analizador léxico y sintáctico que incluye en primer lugar la parte léxica del lenguaje.

Existen algunos caracteres especiales (como el espacio en blanco y el retroceso) que no son necesarios en el análisis y para no tomarlos en cuenta se escribe lo siguiente:

```
SKIP :
{ " "
| "\t"
| "\n"
| "\r"
| "\f"
}
```

Cada uno de los operadores se define como un token y es posible utilizar el caracter directamente en la gramática. Sin embargo, para no revolver la parte léxica con la semántica, la definición debe hacerse al inicio:

```
TOKEN
{
      < OR: "|">
|     < HYPHEN: "-">
|     < QUESTION: "?">
|     < GT: ">">
|     < EQUAL: "=">
|     < MT: "<">
|     < DASH: "/">
|     < PLUS: "+">
|     < ASTERISK: "*">
}
```

De igual forma sucede con los separadores:

```
TOKEN:
{
      < WILDCARD: "#">
|     < QUOTA: "'">
|     < LPAREN: "(" >
|     < RPAREN: ")" >
|     < LBRACE: "{" >
|     < RBRACE: "}" >
|     < LBRACKET: "[" >
|     < RBRACKET: "]" >
```

```
|       < COMMA: "," >
|       < DOT: "." >
|       < QUOTES : "\"" >
|       < SEMICOLON : ";" >
}
```

Es importante mencionar que el analizador se implementó como no sensible a mayúsculas o minúsculas. Para especificar esto se utiliza la etiqueta "*IGNORE_CASE*" en los tokens que así lo requieran. En el caso de los identificadores obtenidos a tiempo de ejecución, se utiliza el preprocesador para ignorar su formato. A continuación se definen las palabras reservadas del lenguaje:

```
TOKEN [IGNORE_CASE]:    /*
Reserved Words */
{
      < ALL: "ALL">
|     < AND: "AND">
|     < AS: "AS">
|     < AVG: "AVG">
|     < BELONG: "BELONG">
|     < CLON: "CLON">
|     < CONTAIN: "CONTAIN">
|     < COUNT: "COUNT">
|     < CREATE: "CREATE">
|     < CUT: "CUT">
|     < DELETE: "DELETE">
|     < DISTINCT: "DISTINCT">
|     < DROP: "DROP">
|     < EMPTY: "EMPTY">
|     < EXIST: "EXIST">
|     < FALSE: "FALSE">
|     < FILE: "FILE">
|     < FOR: "FOR">
|     < FROM: "FROM">
|     < IN: "IN">
|     < IS: "IS">
|     < ISOMORPH: "ISOMORPH">
|     < LIKE: "LIKE">
|     < MAX: "MAX">
|     < MIN: "MIN">
|     < MOD: "MOD">
|     < MVIEW: "MVIEW">
|     < NOT: "NOT">
|     < ORL: "OR">
|     < OWN: "OWN">
|     < PICK: "PICK">
|     < PRIMITIVE:
"PRIMITIVE">
|     < SELECT: "SELECT">
|     < SET: "SET">
|     < SSDTABLE:
"SSDTABLE">
|     < SUM: "SUM">
|     < TRIM: "TRIM">
|     < TRUE: "TRUE">
|     < TWOPOINTS: ":">
|     < UNION: "UNION">
|     < UPDATE: "UPDATE">
|     < VIEW: "VIEW">
|     < WHERE: "WHERE">
|     < WITH: "WITH">
}
```

Los siguientes tokens representan las constantes utilizadas por el lenguaje, es decir, las cadenas de caracteres, números enteros y números reales. Son la traducción de las producciones:

- <numeric-constant>
- <string-constant>
- <digits>
- <letters-digits>
- <letter>
- <digit>

mostradas en la figura A.3. Estas producciones se tradujeron como expresiones regulares para representarse como token y de esa forma ser reconocibles por el analizador léxico:

```
TOKEN:
{
      < #LETTER: [ "a"-"z" , "A"-"Z"] | "\u00E1" | "\u00E9"
      | "\u00ED" | "\u00F3" | "\u00FA" | "\u00F1">
|     < #DIGIT: ["0"-"9"] >
|     < #UNDERH: "_">
|     < #DIGITS : (<DIGIT>)+ >
|     < NUMERICCONSTANT : (<DOT> <DIGITS>|<DIGITS>
                              (<DOT><DIGITS>)?) >
|     < LETTERSDIGITS : <LETTER>
                        (<LETTER>|<DIGIT>|<UNDERH>) *>
|
      < CHARACTER : ["\u0000" - "\u0021" , "\u0023" -
"\uFFFF"] >
|
      < STRINGCONSTANT : <QUOTES> (<CHARACTER>)* <QUOTES> >
}
```

Con lo anterior se termina la especificación del análisis léxico.

A continuación se definen cada una de las producciones de la gramática en notación JavaCC.

Notación BNF	<statement> ::= <query-statement>; | <modification-statement>; | <data-definition-statement>; | <view-statement>;
Notación JJTree	SimpleNode Statement() #Root : {} { (QueryStatement() | ModificationStatement() | LOOKAHEAD(5) DataDefinitionStatement() | ViewStatement()) <SEMICOLON> { return jjtThis; } }

Como `Statement` es el símbolo inicial, éste devolverá la raíz del árbol de análisis sintáctico, misma que será una instancia de la clase `Root` que hereda de la clase `SimpleNode`. Por lo anterior, esta producción en JJtree inicia con el tipo de objeto que devolverá (`SimpleNode`) y finaliza con el código en Java de retorno de un objeto jjtThis, que representa el nodo creado en la producción actual. Además de esta ocasión, más adelante se incrusta código Java para guardar valores constantes dentro de los nodos del árbol.

Otro detalle a tomar en cuenta con esta producción, es que `Statement` debe ser el símbolo inicial. Para ello, éste tiene que aparecer como la primera producción dentro del archivo de especificación *archivo.jjt*

BNF	`<query-statement> ::= <ssd-construction>`
JJTree	`void QueryStatement() : {} { SsdConstruction() }`

BNF	`<modification-statement> ::= <delete-statement> | <update-statement>`
JJTree	`void ModificationStatement() : {} { DeleteStatement() | UpdateStatement() }`

BNF	`<data-definition-statement> ::= <create-statement> | <drop-statement>`
JJTree	`void DataDefinitionStatement() : {} { CreateStatement() | DropStatement() }`

BNF	`<ssd-construction> ::=`
	`<ssd-table-name> |`
	`<primitive-constant> |`
	`<query>`
	`EMPTY |`

```
        ( <ssd-construction> ) |
        <ssd-construction> UNION <ssd-construction> |
        <ssd-construction> PICK ( <list-labels> ) |
        <ssd-construction> TRIM ( <list-labels> ) |
        { <list-labeled-ssds> } |
        CLON <ssd-construction> |
        <aggregation-function> ( <ssd-construction> ) |
        <ssd-construction> <primitive-op> <ssd-construction>
```

JJTree

```
void SsdConstruction() #Construction: {}     { ( unionExpr() ) }

void unionExpr(): {}{ addExpr() ( LOOKAHEAD(2) <UNION> addExpr()
#Union(2) )* }

void addExpr() : {}{ timesExpr() (
                     LOOKAHEAD(2)(( <PLUS> timesExpr() #Plus(2) )
                     |
                     ( <HYPHEN> timesExpr() #Minus(2) ))
                     )*
               }

void timesExpr() : {} { cutExpr() (
                     LOOKAHEAD(2) (
                            (<ASTERISK> cutExpr() #Mult(2))
                            |
                            (<DASH> cutExpr() #Div(2))
                            |
                            (<MOD> cutExpr() #Mod(2))
                     )
                     )* }

void cutExpr() : {}     { Association() (
                     LOOKAHEAD(2) (
                     (<PICK> <LPAREN> ListLabels() #labels()
<RPAREN> #Pick(2))
                     |

                     (<TRIM> <LPAREN> ListLabels() #labels()
<RPAREN> #Trim(2))
                     )
                     )* }

void Association(): {}{ <LPAREN> SsdConstruction() <RPAREN> |
                        <LBRACE> ListLabeledSsds() <RBRACE> |
                        SsdConsterm()
                      }

void SsdConsterm(): {}{ EmptyStatement() |
                        Query() |
                        ClonStatement() |
                        AgregationStatement() |
                        LOOKAHEAD(2) SsdTableName() |
                        PrimitiveConstant()
                      }

void EmptyStatement() #Empty: {} { <EMPTY> }

void ClonStatement()  #Clon: {}  { <CLON> SsdConstruction() }
```

Esta conversión incluye los operadores de la tabla A.1 y se observa la transformación realizada a la gramática para considerar la precedencia y asociatividad de los mismos.

Por cada nueva producción creada en la notación JJTree el nivel de precedencia aumenta; por lo tanto, tiene que ejecutarse primero, razón por la que debe quedar a más profundidad del árbol. De igual forma, entre más cerca se encuentre el operador de la raíz, menor precedencia tendrá.

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

JJTree	void ListLabels() :
	{}
	{
	Label() [<COMMA> ListLabels()]
	}

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

JJTree	void ListLabeledSsds() #NewSSD:
	{}
	{
	Label() <TWOPOINTS> SsdConstruction() #ConstructSSD(2) [<COMMA> ListLabeledSsdsNext()]
	}
	void ListLabeledSsdsNext():
	{}
	{
	Label() <TWOPOINTS> SsdConstruction() #ConstructSSD(2) [<COMMA> ListLabeledSsdsNext()]
	}

BNF	<aggregation-function>::= AVG | MIN | SUM | COUNT

JJTree

```
void AggregationStatement() : {} {
        <AVG> <LPAREN> SsdConstruction() <RPAREN> #Avg(1)
|
        <MAX> <LPAREN> SsdConstruction() <RPAREN> #Max(1)
|
        <MIN> <LPAREN> SsdConstruction() <RPAREN> #Min(1)
|
        <SUM> <LPAREN> SsdConstruction() <RPAREN> #Sum(1)
|
        <COUNT> <LPAREN> SsdConstruction() <RPAREN> #Count(1)
```

BNF

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

JJTree

```
void Query() : {} {
        <SELECT> (DeltaQuery()
                |     StartQuery() )
}
void EndQuery() #MAP: {} {
        MainConstruction()   <FROM>   ListAssociations()   [   <WHERE>
Condition() #SELECT(2)]
}
void DeltaQuery() #DELTA : {} {
        <DISTINCT> EndQuery()
}
void StartQuery() : {} {
        EndQuery()
}
```

BNF

```
<delete-statement>     ::=     DELETE <ssd-table-name>
                               FROM <list-associations> |
                               DELETE <ssd-table-name>
                               FROM <list-association>
                               WHERE <condition>
```

JJTree

```
void DeleteStatement() #DELETE:
{}
{
        <DELETE> SsdTableName() <FROM> ListAssociations() [ <WHERE>
                Condition() #SELECT(2)]
}
```

BNF

```
<update-statemet>      ::=     UPDATE <ssd-table-name>
                               SET <ssd-construction>
                               FROM <list-associations> |
                               UPDATE <ssd-table-name>
                               SET <ssd-construction>
                               FROM <list-associations>
                               WHERE <condition>
```

JJTree

```
void UpdateStatement() #UPDATE:
{}
{
        <UPDATE> SsdTableName() #DATA(1) <SET> SsdConstruction()
#SET(1) <FROM> ListAssociations() [ <WHERE> Condition() SELECT(2)]
}
```

BNF

```
<create-statement>     ::=     CREATE SSDTABLE <ssd-table-name>
                               WITH <ssd-construction> |
                               CREATE SSDTABLE <ssd-table-name>
                               WITH FILE <string-constant>
```

JJTree

```
void CreateStatement() #CREATE:
{}
{
        <CREATE> <SSDTABLE> Label() <WITH> ( SsdConstruction() |
XmlFile() )
}
void XmlFile() #XMLFILE:
{Token t;}
{
        <FILE> t=<STRINGCONSTANT> { jjtThis.setToken(t.image); }
}
```

BNF

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

JJTree

```
void DropStatement() #DropTable:
{}
{
        <DROP> <SSDTABLE> SsdTableName()
}
```

BNF

```
<view-statement>       ::=     <view-definition-statement> |
                               <drop-view-statement>
```

JJTree

```
void ViewStatement() :
{}
{
        ViewDefinitionStatement()
|
        DropViewStatement()
}
```

BNF

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

JJTree

```
void ViewDefinitionStatement() :
{}
{       LOOKAHEAD(2)
        MView()
        |
        AView()
}

void MView() #CREATE:
{}
{
        <CREATE> <MVIEW> {new DBInfo().setMview(true);} Label()
<WITH> SsdConstruction()
}

void AView() #View:
{}
{
        <CREATE> <VIEW> Label() <WITH> SsdConstruction()
}
```

BNF

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

JJTree

```
void DropViewStatement() #DropView:
{}
{
        <DROP> <VIEW> SsdTableName()
}
```

BNF

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

JJTree

```
void ListAssociations() :
{}
{
        PEAssociation() ( LOOKAHEAD(2) <COMMA> PEAssociation()
#Join(2) )*
}
void PEAssociation() #CreateTemp :
{}
{
  PathExpression() <AS> SsdTableTemp()
}
```

BNF

```
<condition>::= TRUE |
               FALSE |
               ( <condition> ) |
               <condition> AND <condition> |
               <condition> OR <condition> |
               NOT <condition> |
               <sdd-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> )
```

JJTree

```
void Condition() :{}
{       notExpr()
        ( LOOKAHEAD(2) (<AND> notExpr() #And(2) |
                        <ORL> notExpr() #Or(2) )
        )*
}

void notExpr() :{}
{       ExisExpr() |
        <NOT> ExisExpr() #Not(1)
}

void ExisExpr():{}
{       CompOpExpr()
        |
        (       <FOR> <ALL> SsdTableTemp() <IN> SsdConstruction()
<LPAREN>
                        Condition() <RPAREN> #ForAll(3)
        )
        |
        (       <EXIST> SsdTableTemp() <IN> SsdConstruction()
<LPAREN>
                        Condition() <RPAREN> #Exist(3)
        )
}

void CompOpExpr():{}
{       LOOKAHEAD(3)
        ConditionAssociation()
        |
        (       PrimitiveExpr() (
                     LOOKAHEAD(2) (
                        <LIKE> PrimitiveExpr() #Like(2)
                        |<OWN> Label() #Own(2)
                        |<IS> PrimitiveExpr() #Is(2)
                        |<CONTAIN> PrimitiveExpr() #Contain(2)
                        |<BELONG> PrimitiveExpr() #Belong(2)
                        |<ISOMORPH> PrimitiveExpr() #Isomorph(2)
                        |<LT> PrimitiveExpr() #LT(2)
                        |<LT><EQUAL> PrimitiveExpr() #LE(2)
                        |<GT> PrimitiveExpr() #GT(2)
                        |<GT><EQUAL PrimitiveExpr() #GE(2)
                        |<EQUAL> PrimitiveExpr() #Equal(2)
                        |<LT><GT> PrimitiveExpr() #NEqual(2)
                     )
                )*
        )
}

void PrimitiveExpr()  :{}    {
SsdConstruction() | <PRIMITIVE> SsdConstruction() #isPrimitive
        }

void ConditionAssociation()   :{}     {<LPAREN> Condition() <RPAREN>|
                                       ConditionTerm()
                                      }
void ConditionTerm() #bool    :{Token t;}
        {t=<TRUE> { jjtThis.setToken(t.image); }
        |
        t=<FALSE> { jjtThis.setToken(t.image); }
}
```

La producción `condition` en JJTree resuelve el problema de no considerar la precedencia y asociatividad de los operadores condicionales de la tabla A.2 en la gramática original. Al igual que con la producción `SsdConstruction`, se generan nuevas producciones para alejar de la raíz a los operadores condicionales conforme tengan mayor precedencia.

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

JJTree	`void SsdTableName() #SsdTable: {Token t;} { t = <LETTERSDIGITS> { jjtThis.setToken(t.image); } }`
	
	`void SsdTableTemp() #SsdTableTemp: {Token t;}{ t = <LETTERSDIGITS> { jjtThis.setToken(t.image); } }`

Se ocupan dos tipos de producciones para los nombres de las Ssd-tablas: `SddTableName` para especificar que va a utilizarse una Ssd-tabla existente en la base de datos; `SsdTableTemp` para especificar que se creará una nueva Ssd-tabla temporal. Lo anterior ayuda al preprocesador a identificar y validar los nombres de las Ssd-tablas utilizadas dentro de la consulta.

Dentro de estas dos producciones, `SsdTableName` y `SsdTableTemp`, se incrusta código Java necesario para guardar el valor constante (una cadena de caracteres) dentro del nodo creado por JJTree. De igual manera ocurrirá con toda producción que represente una constante.

BNF	`<label> ::= <letters-digits>`
JJTree	`void Label() #Label: {Token t;} { t = <LETTERSDIGITS> { jjtThis.setToken(t.image); } }`

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

JJTree

```
void PrimitiveConstant() #PrimitiveConstant:
{Token t;}
{
        t = <NUMERICCONSTANT> { jjtThis.setToken(t.image); }
|
        t = <STRINGCONSTANT> { jjtThis.setToken(t.image); }
}
```

BNF

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

JJTree

```
void PathExpression() #PathExpression: {}
{
  DotPE1() (<OR> DotPE1() #ORPE(2))*
}

void PathExpression1() : {}
{
  DotPE1() (<OR> DotPE1() #ORPE(2))*
}

void DotPE1() : {}
{
  UnitPE1() ( <DOT> UnitPE() #DOT(2))*
}

void DotPE() : {}
{
  UnitPE() ( <DOT> UnitPE() #DOT(2))*
}

void UnitPE1() : {}
{
  Pexp1()
}

void UnitPE() : {}
{
  Pexp() [
            <ASTERISK> #Kleene(1)
            |
            <PLUS> #OneMore(1)
            |
            <QUESTION> #Nullable(1)
          ]
}

void Pexp1() : {}
{
  <LPAREN> PathExpression1() <RPAREN>
  |
  SsdTableName()
}

void Pexp() : {}
{
  <LPAREN> PathExpressionTail() <RPAREN>
  |
  PeLabel()
}

void PathExpressionTail() : {}
{
  DotPE() (<OR> DotPE() #ORPE(2))*
}
```

También existe una precedencia entre los operadores que se utilizan dentro de las expresiones de camino. Debe validarse también que la primera etiqueta que contenga una expresión de camino no pueda emplear dichos operadores ni etiquetas entre comillas simples, pues esta primera etiqueta

debe ser la de una Ssd-tabla. Ésto hace que la producción para dichas expresiones se vuelva extensa.

BNF

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

JJTree

```
void PeLabel() :
{}
{
  LabelN1()
  |
  <QUOTA> XLabel() <QUOTA>
}
void LabelN1() #LabelN1:
{Token t;}
{
        t = <LETTERSDIGITS>
         {jjtThis.setToken(t.image);}
|
        t = <WILDCARD>
         {jjtThis.setToken(t.image);}
}
```

BNF

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

JJTree

```
void XLabel() #XLabel: {}
{
  UnitX() (<OR> UnitX() #ORXL(2))*
}

void UnitX() #XWord: {}
{
  (XCharacter())*
}

void XCharacter() #XCharacter: {Token t;}
{
  t = <LETTERSDIGITS> {jjtThis.setToken(t.image);}
  |
  t = <WILDCARD> {jjtThis.setToken(t.image);}
  |
  t = <ASTERISK> {jjtThis.setToken(t.image);}
  |
  t = <PLUS> {jjtThis.setToken(t.image);}
  |
  t = <QUESTION> {jjtThis.setToken(t.image);}
  |
  <LPAREN> XLabel() <RPAREN>
}
```

Apéndice B

Pruebas de rendimiento

A continuación se muestran algunos resultados de las pruebas que se realizaron al procesador de consultas relativas al tiempo de ejecución, dividido entre el tiempo que toma convertir una consulta a plan físico (tiempo de análisis) y el tiempo de ejecución de éste. Además, se determina qué impacto tiene el aplicar o no la técnica del *resumen de datos con identificadores* en la ejecución de la consulta, tanto para su análisis como para su ejecución.

Además de mostrar la diferencia de tiempos de ejecución cuando se utiliza o no el *resumen de datos con identificadores*, también se probó el procesador de consultas con diferentes tamaños y tipos de bases datos. En relación al tamaño, se usaron bases de datos pequeñas —con alrededor de 30 nodos y 40 aristas— y bases de datos grandes —con más de 10,000 nodos y aristas— para probar el comportamiento del procesador cuando maneja pocos y muchos datos, respectivamente. En relación al tipo, se experimentó con bases de datos de estructura homogénea y no homogénea, a fin de observar en cuánto ayuda o no el uso del *resumen de datos con identificadores* en cualquiera de los dos casos.

Los resultados se presentan en porcentaje (con un error de ±3%) e indica el grado de mayor rapidez en la realización de una consulta. Para ello, se toma como el 100% al tiempo de la alternativa que tuvo la mayor duración, la que al contrastarla con la del menor tiempo se obtiene el porcentaje.

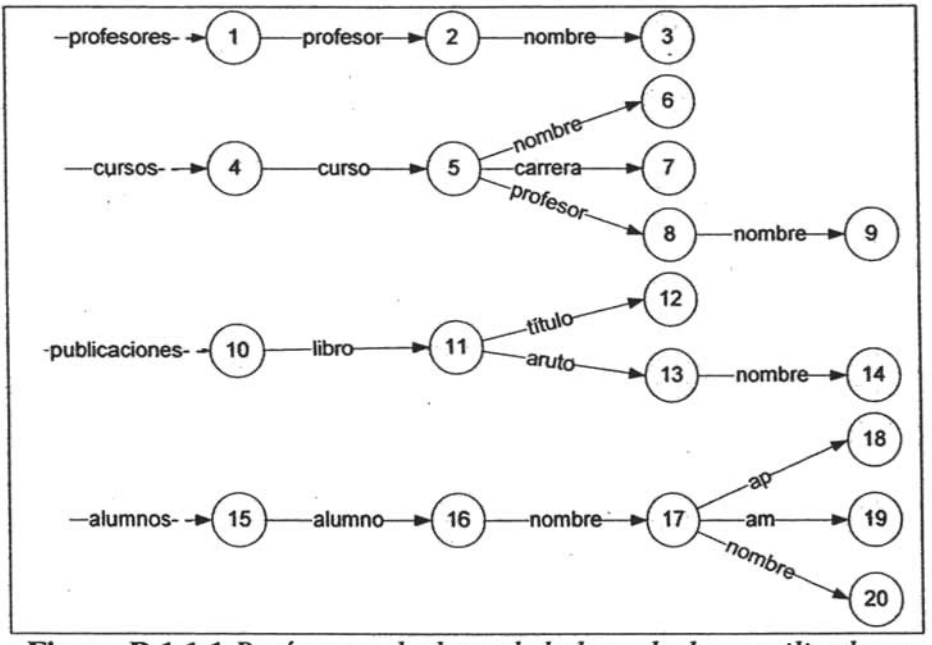


Figura B.1.1.1 *Resúmenes de datos de la base de datos utilizada en el apartado B.1.1*

B.1 Bases de datos pequeñas

B.1.1 Bases de datos con estructura homogénea

Este apartado maneja una base de datos pequeña, cuyos datos comparten una misma estructura, es decir, cada uno tiene una estructura igual a la de los demás datos de una Ssd-tabla.

Para tener conocimiento de la estructura de esta base de datos, en la figura B.1.1.1 se presenta el *resumen de datos* de cada una de las Ssd-tablas contenidas en ella.

En esta prueba, la Ssd-tabla profesores cuenta con sólo tres profesores, la de cursos también tiene tres cursos, la de publicaciones contiene dos libros y existen 24 alumnos en la tabla correspondiente, donde sólo uno tiene su nombre separado por apellidos y nombre. Las consultas realizadas sobre esta base de datos y sus tiempos de ejecución fueron las siguientes:

Consulta			
SELECT FROM	nombre : X profesores.profesor.nombre as X;		
	Tiempo de Análisis	Tiempo de Ejecución	Total
Sin optimización	=	6% más rápido	6% más rápido
Con optimización	=		

Consulta			
SELECT FROM	profesor: X UNION (SELECT publicaciones: P FROM publicaciones.# AS P, P.autor AS A WHERE A IS X) profesores.profesor AS X;		
	Tiempo de Análisis	Tiempo de Ejecución	Total
Sin optimización	=	8% más rápido	8% más rápido
Con optimización	=		

Consulta			
SELECT FROM	ApellidosPaternos : P alumnos.alumno.nombre.ap AS P;		
	Tiempo de Análisis	Tiempo de Ejecución	Total
Sin optimización	=		
Con optimización	=	84% más rápido	84% más rápido

Como puede observarse en los tiempos de ejecución de estas consultas, para una base de datos con un número limitado de ellos, la diferencia entre utilizar o no el *resumen de datos* no es de gran impacto. Sin embargo, hay un cambio notable en la última consulta, donde se pregunta por el único apellido almacenado en la base de datos. Al no tener que recorrer los otros 23 alumnos que no lo tienen registrado, se observa una reducción considerable al momento de ejecutar la consulta. En general, puede

concluirse que aún con una base de datos con estructura no variable entre dato y dato, al haber pocos, realizar el gasto que implica llegar a un *resumen de datos* no muestra una mejora al momento de ejecutar una consulta.

B.1.2 Bases de datos con estructura no homogénea

Para una base de datos en la que el *resumen de datos* tiene una estructura idéntica a la de los datos en sí —no es posible realizar ninguna simplificación en la estructura al crear el resumen— y con un número pequeño de datos (16 artículos), los resultados fueron los siguientes:

Consulta			
SELECT FROM	código : C artículos.#.código as C;		
	Tiempo de Análisis	Tiempo de Ejecución	Total
Sin optimización	=	25% más rápido	25% más rápido
Con optimización	=		

Consulta			
SELECT FROM	autos : A artículos.auto.Modelo as A;		
	Tiempo de Análisis	Tiempo de Ejecución	Total
Sin optimización	=	25% más rápido	25% más rápido
Con optimización	=		

El utilizar el *resumen de datos con identificadores* para este tipo de bases de datos supone un gasto extra al ejecutar la consulta. Ésto es debido a que el número de aristas recorridas para encontrar el dato iniciando desde la raíz de la Ssd-tabla, es menor que comenzando desde el resumen, ya que

para obtener los datos sin utilizarlo se requiere recorrer tres aristas por cada uno, pero ocupándolo, se necesitan recorrer cuatro.

B.2 Bases de datos grandes

B.2.1 Bases de datos con estructura homogénea

La base de datos utilizada en este apartado se obtuvo al cargar datos desde documentos XML, importándolos desde una base de datos relacional a archivos XML. Esta base de datos contiene alrededor de 15,000 nodos y más de 12,000 aristas y la distribución de éstos entre las Ssd-tablas se muestra en la figura B.2.1.1

La base de datos corresponde a un sistema de ventas y en ella se almacena la información de los pedidos efectuados (tabla pedidos), quiénes los realizaron (tabla clientes), los productos que maneja la empresa (tabla productos), las compañías que surten a la empresa (tabla proveedores), las compañías que se utilizan para remitir los pedidos (tabla compañías de envíos) y un registro de los empleados que atendieron a los clientes (tabla empleados).

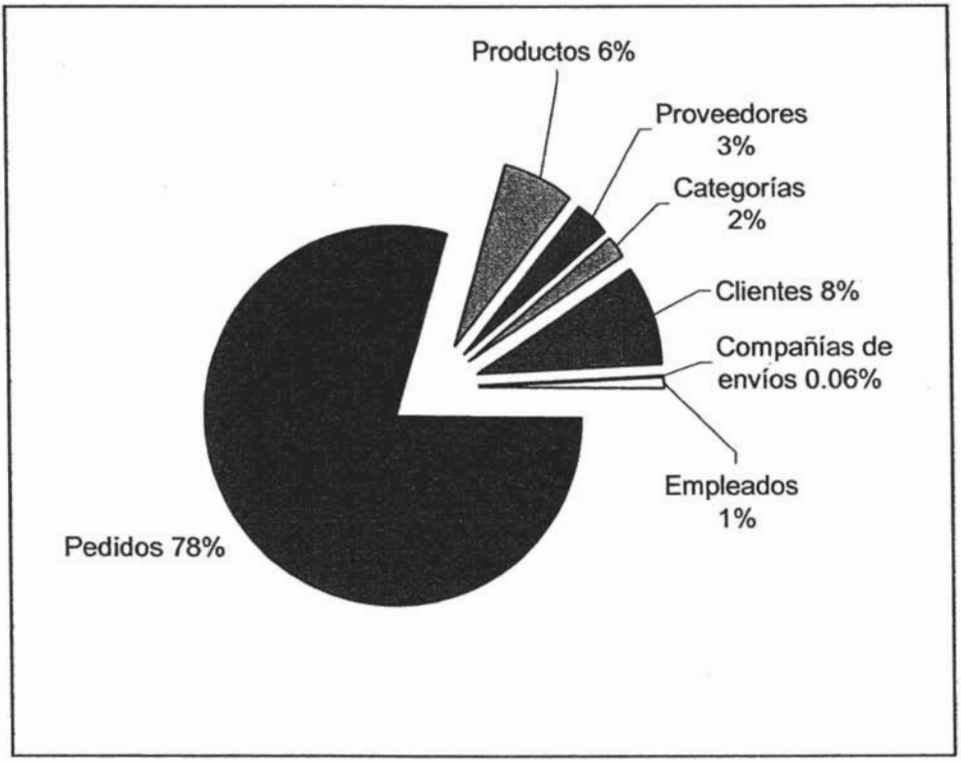


Figura B.2.1.1 *Distribución de información entre las Ssd-tablas*

La circunstancia que se genera al ingresar los datos desde archivos XML a la base de datos semiestructurados es que no se aprovechan las conexiones que pueden hacerse entre los datos semiestructurados sin necesidad de crear nuevos nodos. Es decir, para las referencias foráneas que existen en las bases de datos relacionales es necesario utilizar el identificador de la tupla a la que se haga referencia, mientras que en las bases de datos semiestructurados, para crear esta misma relación, sólo basta generar una arista que conecte ambos datos. Por ejemplo, en la base de datos utilizada en esta sección, la mayor cantidad de nodos y aristas pertenecen a la Ssd-tabla "*pedidos*", debido a que ésta, en la base de datos relacional, realiza la función de unir un cliente con el producto que compró, quién lo atendió y el precio del producto. Por ello, la mayor cantidad de datos que contiene son llaves foráneas, que se usan en las bases de datos relacionales para efectuar la conexión entre los datos. En las bases de datos semiestructurados, este tipo de conexión se realiza mediante el uso de aristas, donde la cantidad de nodos necesarios para representar la misma información que en la relación "pedidos" sería menor. Por lo anterior,

migrar la información de una base de datos relacional a una base de datos semiestructurados requiere de más trabajo que simplemente utilizar un documento XML para realizarlo. Pero ello puede llevarse a cabo automáticamente por un programa que considere las dependencias entre los datos en la base de datos relacional antes de migrar la tupla, teniendo en cuenta el identificador del nodo raíz que representa la tupla de la llave foránea, para crear una arista que apunte a tal identificador cuando se haga uso de dicha llave. Una exploración más a fondo sobre este tema queda como parte de una investigación futura.

Para probar el procesador de consultas y el motor de ejecución cuando éstos manejan un número considerable de nodos, no es prudente hacer los cambios para efectuar una migración adecuada de relacionales a semiestructurados, ya que si se hicieran, la base de datos contendría un significativo menor número de nodos. Las consultas realizadas fueron las siguientes:

Consulta			
SELECT FROM	compañía : C proveedores.proveedores.nombrecompañía as C;		
	Tiempo de Análisis	Tiempo de Ejecución	Total
Sin optimización	=		
Con optimización	=	25% más rápido	25% más rápido

Consulta			
SELECT FROM	destinatarios : D pedidos.pedidos.destinatario as D;		
	Tiempo de Análisis	Tiempo de Ejecución	Total
Sin optimización	=		
Con optimización	=	23% más rápido	23% más rápido

Consulta			
SELECT VentasXVendedor: {nombre: A+" "+V, Ventas: COUNT(SELECT venta: X FROM pedidos.pedidos.IdEmpleado AS X WHERE X = I) FROM empleados.empleados AS E, E.IdEmpleado AS I, E.nombre AS V, E.apellidos AS A;			
	Tiempo de Análisis	Tiempo de Ejecución	Total
Sin optimización	=		
Con optimización	=	25% más rápido	25% más rápido

B.2.2 Bases de datos con estructura no homogénea

Después de realizar las pruebas del apartado B.1.2 puede afirmarse que al utilizar una base de datos de mayor tamaño, pero con una estructura que no contiene repeticiones, no hay cambio en los resultados obtenidos en el apartado anterior. Es decir, emplear el *resumen de datos con identificadores* implica recorrer una arista más que no utilizándolo. Con una base de datos grande sólo pudo comprobarse que el tiempo extra utilizado en recorrer el resumen de datos es inversamente proporcional a la longitud del camino recorrido que se sigue para obtener los datos.

B.3 Resultados

Una vez realizadas estas pruebas, se llega a las siguientes conclusiones:

- En el peor de los casos, utilizar el *resumen de datos con identificadores* implica recorrer una arista extra en el proceso de ejecución.
- En promedio, para bases de datos grandes (más de 10000 nodos y aristas) se tiene un ahorro del 25% en el tiempo de ejecución usando el *resumen de datos con identificadores*.
- Para bases de datos pequeñas (entre 1 y 100 nodos y aristas) no existe una clara diferencia entre el uso o no del *resumen de datoscon identificadores*.
- El tiempo requerido para determinar cuáles expresiones de camino pueden iniciar desde el *resumen de datos con identificadores* no causa un gasto significativo de tiempo en el proceso de ejecución.

Por lo tanto, el uso del *resumen de datos con identificadores* es redituable tratándose de bases de datos grandes.