UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

ELABORACIÓN DEL NÚCLEO DE UN SISTEMA MANEJADOR DE BASES DE 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 :

EGAR ARTURO GARCÍA CÁRDENAS

DIRECTORA DE TESIS: DRA. AMPARO LÓPEZ GAONA

México D.F. 2005

A mis padres María de Consuelo Cárdenas Viillordo y
Salomón García Salinas.

A mi amada Ana Patricia Gómez Mayén.

A mi hermano Eric García Cárdenas.

A la Doctora Amparo López Gaona.

Índice general

Índice de figuras

Índice de cuadros

Introducción

La información es el más valioso de los recursos, basta con ver los precios elevados de las licencias que los grandes productores de manejadores de bases de datos venden y la cantidad de dinero que gastan las grandes compañías para obtenerlos. Esto anterior es porque un manejador de bases de datos es una herramienta que brinda un gran apoyo para el manejo de información, una herramienta de este tipo resulta prácticamente invaluable.

Las bases de datos semiestructurados han surgido en los últimos años como una alternativa para superar las dificultades que presentan otros tipos de bases de datos, como las relacionales y orientadas a objetos. Estas dificultades están asociadas con información sumamente flexible (es decir, que no se apega a los formatos establecidos), en constante cambio, cuyas necesidades son difíciles de predecir y se encuentra dispersa en gran variedad de fuentes.

En la actualidad los intentos por construir manejadores para datos con este tipo de características más bien se enfocan hacia el almacenamiento de documentos XML, los cuales tienen cierta simitud con los datos semiestructurados [3]. Lore [13] es un prototipo de un manejador de datos semiestructurados y XML desarrollado en la universidad de Stanford, está basado en un manejador orientado a objetos. Existen sistemas específicos para el almacenamiento, tratamiento y publicación de documentos XML [18] como dbXML [19] y Tamino [20]. También, algunos grandes manejadores de bases de datos relacionales como Oracle, SQL Server y DB2 incluyen mecanismos para el tratamiento de XML.

Debido a las propiedades que ofrecen los datos semiestructurados, la idea de tener un manejador enfocado a las bases de datos semiestructurados es interesante y prometedora. En la actualidad, no existe ninguno que alcance

un alto desempeño como para ser utilizado en aplicaciones grandes o que pueda sustituir a los ya establecidos para bases de datos relacionales.

Diseñar y construir un sistema manejador de bases de datos es una labor titánica. Las grandes organizaciones que producen este tipo de software ocupan a cientos o miles de personas para su desarrollo e invierten grandes cantidades de tiempo en investigación.

El objetivo de este trabajo es el diseño e implementación de los módulos básicos que conforman un sistema manejador de bases de datos semiestructurados, a este conjunto de componentes se le llama el núcleo. Este trabajo es una aportación en la construccion de un sistema manejador de bases de datos semiestructurados de alto desempeño.

Para lograr el objetivo se requirió de mucho tiempo de investigación, reflexión, diseño e implementación por parte del autor. Los productos principales de este trabajo son:

- El diseño de los componentes que conforman el núcleo, que son: la interfaz, el manejo de almacenamiento, el control de concurrencia y el sistema de recuperación.
- La implementación de la interfaz, el manejo de almacenamiento y el control de concurrencia.

Este documento se enfoca principalmente en los aspectos teóricos y el diseño de los componentes. La implementación realizada se incluye en el CD adjunto. El presente documento está estructurado de la siguiente manera:

- En el primer capítulo, se exponen los conceptos de dato semiestructurado, base de datos semiestructurados y se muestran sus ventajas para el almacenamiento de información.
- En el segundo capítulo, se discute acerca de la importancia del conocimiento en la actualidad y el uso de los datos semiestructurados para su representación.
- En el tercer capítulo, se explica lo que es el núcleo de un sistema manejador de bases de datos semiestructurados y se definen una serie de

operaciones básicas para su comunicación con otros módulos de software.

- En el cuarto capítulo, se presenta la construcción realizada para resolver el problema de almacenamiento. Esta construcción consta de tres niveles: Primitivo, Avanzado y de Base de Datos. Se explican las decisiones tomadas considerando la naturaleza de los datos semiestructurados para lograr un alto desempeño.

- En el quinto capítulo, se exponen los conceptos de concurrencia y recuperación, basados en el concepto de transacción. Se explican las construcciones realizadas para el control de concurrencia y el diseño del sistema de recuperación, así como, las decisiones tomadas para su funcionamiento considerando las propiedades de los datos semiestructurados.

- En el apéndice A, se exponen algunas cuestiones referentes al proceso de implementación como la organización del código y la metodología de desarrollo. También, se presentan los resultados de una prueba realizada para evaluar el desempeño de los componentes presentados.

- Finalmente, se exponen las conclusiones, las aportaciones y los trabajos futuros.

Capítulo 1

El Modelo de Datos Semiestructurados

Este capítulo tiene el fin de presentar el concepto de dato semiestructurado. Se explican las características fundamentales de los datos semiestructurados y los métodos para representarlos. También se introduce el concepto de base de datos semiestructurados, se abordan sus propiedades y sus componentes.

1.1. El concepto de dato semiestructurado

Para una gran variedad de problemas se requiere hacer uso de información con una estructura irregular. Esta información puede sufrir un crecimiento imprevisto y debe soportar la falta de cierta información. Los datos semiestructurados se han venido desarrollando como una alternativa para manejar información con este tipo de características.

El uso de modelos relacionales y orientados a objetos para tratar información de tipo no rígido presenta ciertas limitaciones. Por ejemplo, si se usan bases de datos relacionales podría requerirse que se agregaran frecuenemente nuevos campos a las tablas, que se utizara una gran cantidad de valores nulos o en su defecto la elaboración de un diseño demasiado complejo.

Cuando se tiene la necesidad de representar información, frecuentemente lo primero que se hace es definir una estructura para los datos y después se crean instancias de dicha estructura. Es decir, existe una separación entre la definición de la estructura o tipo del dato y su valor. En los *datos semiestructurados* dicha separación no existe, la estructura del dato se describe junto con su valor, es decir, los *datos semiestructurados* son auto-descriptivos. Sin embargo, se tiene que asumir la existencia de ciertos *tipos primitivos de datos*, los tipos primitivos de datos que se tienen para los propósitos de este trabajo son:

- Números enteros.
- Números reales o de punto flotante.
- Cadenas de caracteres.
- Secuencias de bits.
- BLOBs. Que son objetos largos orientados a bytes, como puede ser un video o música.
- CLOBs. Objetos largos orientados a caracteres, como puede ser un libro.

Intuitivamente un *dato semiestructurado* puede ser:

- Un dato primitivo.
- Un conjunto finito de otros datos semiestructurados etiquetados con una cadena de caracteres.

Se asume que un dato semiestructurado tiene un *identificador* único.

Se puede decir también que un dato semiestructurado *no primitivo* es un conjunto que contiene parejas consistentes de una etiqueta y un dato semiestructurado. Un dato semiestructurado se puede contener así mismo de manera directa o recursiva.

Un dato semiestructurado puede contener un mismo dato semiestructurado con varias etiquetas o varios datos semiestructurados con la misma

etiqueta. El orden no importa en los datos semiestructurados. Obsérvese que el conjunto vacío con un identificador asociado es un dato semiestructurado.

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

Todo dato semiestructurado pertenece a una familia. En toda familia de datos semiestructurados existe un dato semiestructurado que es ancestro de todos los demás de la familia y se le llama *raíz*.

Como se puede ver los datos semiestructurados son muy flexibles en cuanto a la estructura. Las contenciones entre ellos se pueden anidar a una profundidad arbitraria.

1.1.1. Sintaxis

Los datos semiestructurados pueden describirse usando una sintaxis sencilla. La que aquí se presenta es bastante común y su objetivo es describir listas asociativas etiqueta-valor [3].

La especificación de la sintaxis en BNF es la siguiente:

```
<ssd-expr> ::= <valor> | &<oid> <valor> | &<oid>
<valor> ::= <valor-primitivo> | <valor-complejo>
<valor-complejo> ::= { <lista-valores> }
<lista-valores> ::= <etiqueta>: <ssd-expr> |
                    <etiqueta>: <ssd-expr>, <lista-valores>
<oid> ::= [a..zA..Z1..9_]+
<etiqueta> ::= [a..zA..Z1..9_]+
```

El símbolo <`valor-primitivo`> toma la sintaxis usada para representar datos de tipo primitivo.

La descripción de un dato semiestructurado está dada por el símbolo <`ssd-expr`>, a ésta se la llama *ssd-expresión*. El símbolo <`oid`> se refiere a un identificador que debe ser único para cada dato. El símbolo <`valor`> representa el contenido de un dato semiestructurado que puede ser un dato primitivo o un conjunto de datos semiestructurados etiquetados. El símbolo <`valor-complejo`> representa un conjunto de datos semiestructurados etiquetados cuyo contenido se dá en <`lista-valores`>.

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

Para que un dato semiestructurado esté bien descrito a través de una ssd-expresión s se requiere que s sea consistente. Una ssd-expresión s es consistente si y sólo si:

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

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

Con la sintaxis definida anteriormente, se describen familias de datos semiestructurados empezando por una raíz.

Ejemplo 1.1.1 Se describe un dato semiestructurado para representar la información de algunos países. El dato semiestructurado consta a su vez de tres datos semiestructurados etiquetados como país. Éstos a su vez contienen otros datos primitivos que representan la información del país, tales como el nombre, la capital, la moneda y el idioma. Obsérvese cómo un dato semiestructurado puede contener varios con la misma etiqueta.

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

Ejemplo 1.1.2 El siguiente dato semiestructurado consta de otros datos semiestructurados que representan lenguajes o paradigmas de lenguajes de programación. Estos últimos a su vez contienen datos primitivos que representan lenguajes que concuerdan con el paradigma o contiene otros datos semiestructurados que corresponden a subparadigmas. Obsérvese cómo la información que contienen los datos semiestructurados es muy variable de uno a otro.

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

Ejemplo 1.1.3 En el siguiente dato semiestructurado, se representan las relaciones de parentesco entre cuatro personas. Obsérvese cómo las personas y sus identificadores se definen una sola vez. Las relaciones de parentesco se

definen por medio de los identificadores, así no se requiere definir más de una vez cada persona.

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

1.1.2. Representación con gráficas

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

Los datos semiestructurados se representan utilizando gráficas dirigidas con aristas etiquetadas [3]. Cada dato semiestructurado corresponde a un único vértice de la gráfica. Los datos primitivos corresponden a hojas (vértices que no tienen aristas de salida). A los vértices que corresponden a datos primitivos se les asocia el valor del dato primitivo. El identificador de un dato semiestructurado se coloca en su vértice correspondiente.

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

Ejemplo 1.1.4 En las figuras 1.1, 1.2 y 1.3 se muestran las gráficas de los datos semiestructurados descritos en los ejemplos 1.1.1, 1.1.2 y 1.1.3 respectivamente.

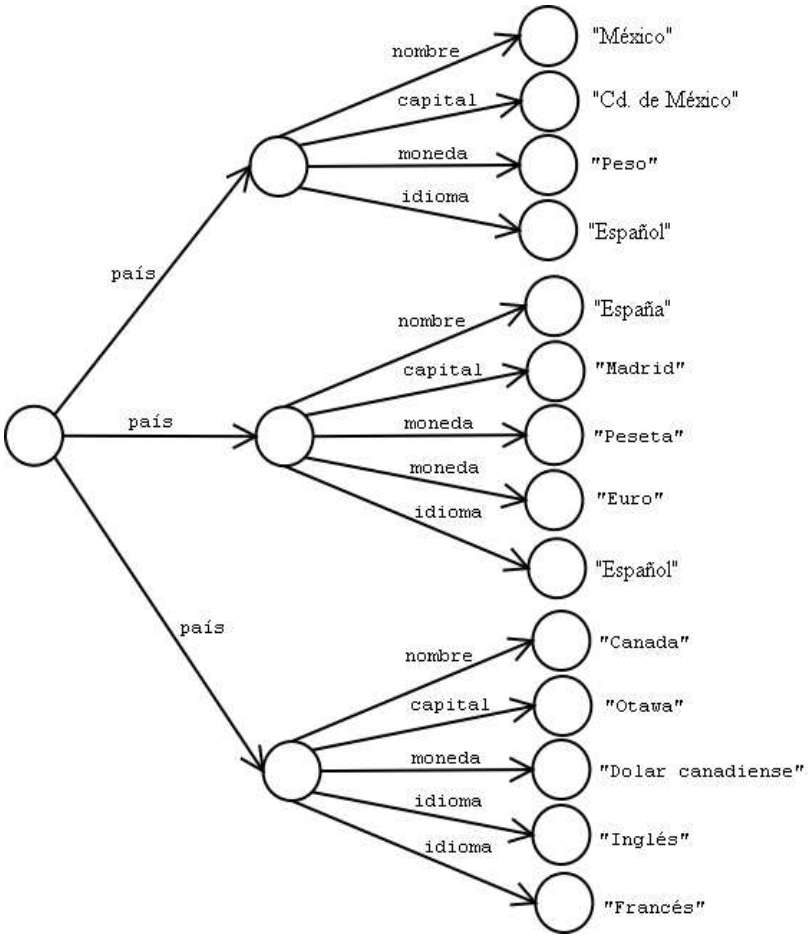

Figura 1.1: Representación gráfica del dato semiestructurado del ejemplo 1.1.1

1.2. Bases de datos semiestructurados

El concepto de *base de datos* involucra tres entidades que son: una colección de datos, un sistema manejador y un modelo para representar los datos. Si el modelo que se sigue para representar los datos está basado en el modelo de datos semiestructurados, entonces se trata de una *base de datos semiestructurados*.

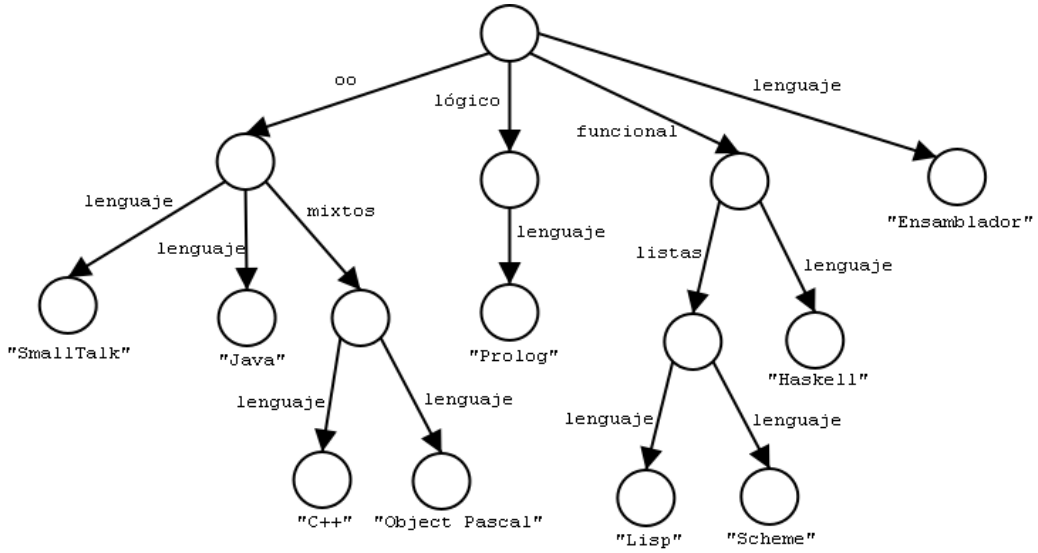

Figura 1.2: Representación gráfica del dato semiestructurado del ejemplo 1.1.2

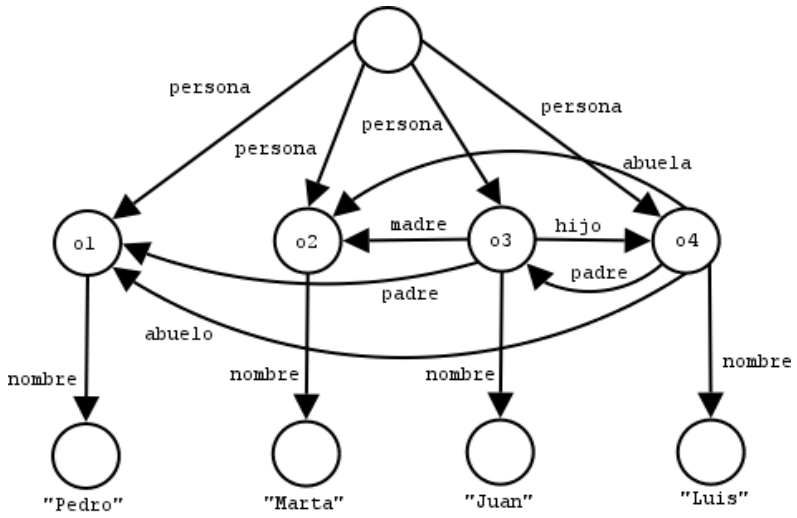

Figura 1.3: Representación gráfica del dato semiestructurado del ejemplo 1.1.3

Las bases de datos trabajan sobre un modelo teórico que especifica la forma en que se representan los datos y las operaciones que se les puede aplicar. Por ejemplo las bases de datos relacionales utilizan el modelo relacional que consiste de tablas o relaciones y las operaciones están dadas por el álgebra relacional. Un *sistema manejador de bases de datos* consiste en un conjunto de módulos de software que se encargan del manejo de la colección de datos de acuerdo con el modelo utilizado, la colección de datos es lo que comunmente se le llama la *base de datos* [1].

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

1.2.1. Las ssd-tablas

Una *base de datos semiestructurados* se puede ver como una colección de datos semiestructurados. Sin embargo, es necesario contar con algunos datos semiestructurados distinguidos, a través de los cuales se pueda acceder a todos los demás. Una *ssd-tabla* es un dato semiestructurado distinguido mediante un nombre o etiqueta. Este dato semiestructurado es la raíz o el ancestro de otros datos semiestructurados, los cuales están contenidos y pueden ser accedidos o modificados a través de dicha raíz. A la colección de *ssd-tablas* se la llama *diccionario de datos*.

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

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

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

Las *ssd-tablas* proporcionan los puntos de acceso o de partida para la manipulación de la información en la base de datos.

Ejemplo 1.2.1 En la figura 1.4 se muestra un ejemplo de una base de datos semiestructurados. Ésta consta de tres *ssd-tablas* llamadas *cursos*, *profesores* y *publicaciones*. Estas *ssd-tablas* se refieren a los datos semiestructurados con identificadores $c1$, $m1$ y $p1$ respectivamente. Estos datos son la raíz de otros más. Obsérvese cómo las *ssd-tablas* pueden compartir datos y todos los datos semiestructurados en la base de datos tienen un identificador distinto.

1.2.2. Redundancia y falta de información

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

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

En las bases de datos relacionales, la falta de información corresponde a la presencia de nulos en las tablas. El permitir el uso de nulos en las tablas, en la mayoría de los casos tiene el propósito de simplificar el diseño de la base de datos. El impedir que los nulos aparezcan puede complicar tremendamente el diseño debido a que las entidades poseen una estructura rígida. Sin embargo,

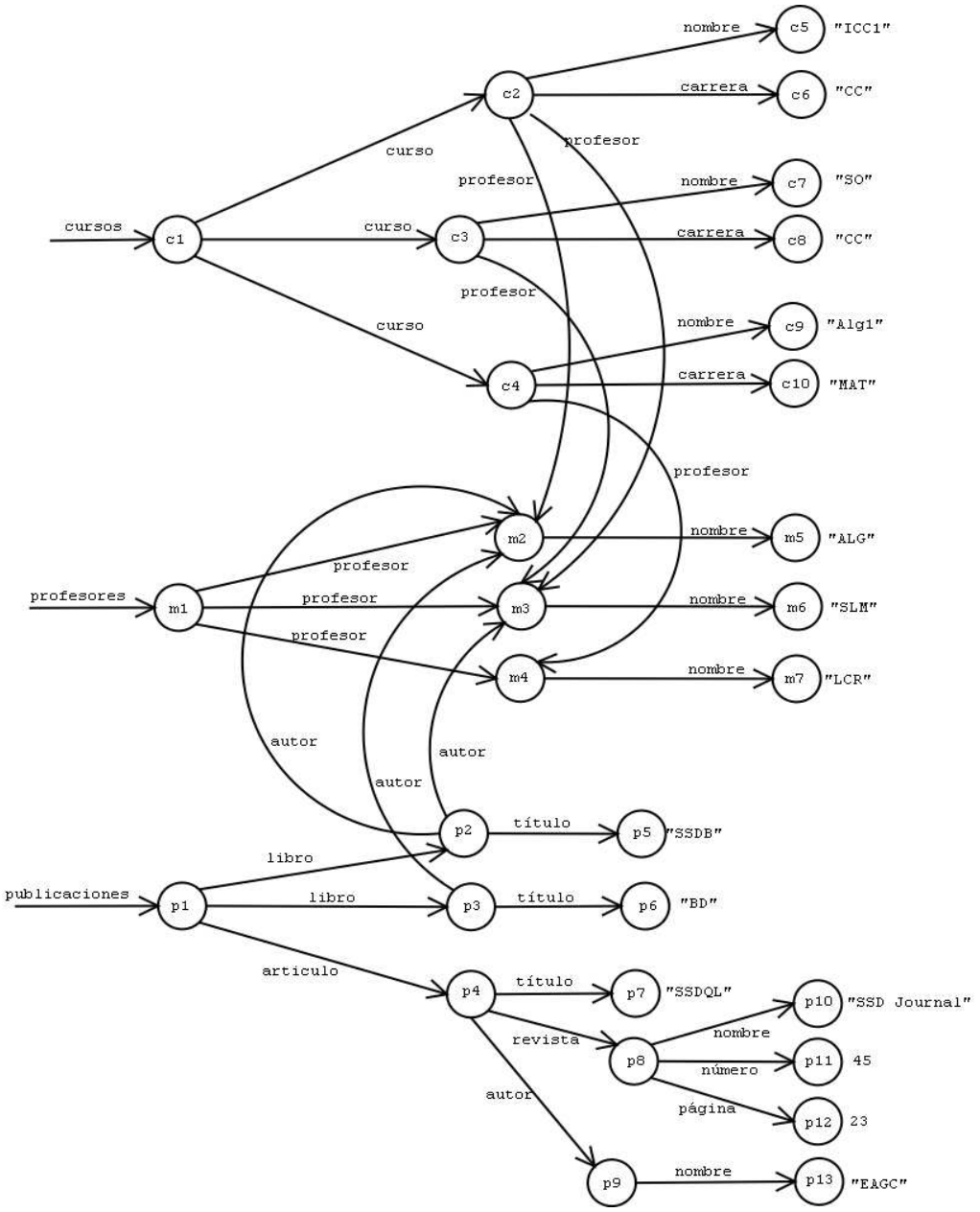

Figura 1.4: Base de datos semiestructurados.

el uso de nulos tiene varios problemas como son espacio desperdiciado y la introducción de una lógica trivalente.

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

Resumen

En este capítulo se presentó el concepto de dato semiestructurado y sus características. Se mencionaron dos formas de representarlos una mediante una sintaxis y la otra mediante gráficas. Se explicó lo que es una base de datos semiestructurados y finalmente las propiedades de éstas respecto a la redundancia y falta de información.

La naturaleza y caracteristicas de los datos semiestructurados, los hacen factibles para ser utilizados en dominios en donde otros modelos, presentan dificultades. Para utilizar los datos semiestructurados con un enfoque de almacenamiento, fue necesario definir el concepto de base de datos semiestructurados y sus componentes.

En el siguiente cápitulo se discute acerca de las necesidades actuales de información, la importancia del conocimiento y el uso de los datos semiestructurados como una alternativa a estas cuestiones.

Capítulo 2

Datos Semiestructurados y Representación del Conocimiento

En este capítulo se discute el concepto de conocimiento, la importancia que el conocimiento tiene en la actualidad y el uso de los datos semiestructurados como una alternativa para su representación.

2.1. Importancia del conocimiento

Los sistemas computacionales, al igual que todas las máquinas han servido al hombre como una extensión de sus propias capacidades. La mayoría de las máquinas extienden las capacidades físicas del hombre. Los sistemas computacionales extienden sus capacidades mentales.

Las necesidades y capacidades de los sistemas de cómputo han ido cambiando con el paso de los años. El propósito de los primeros sistemas de cómputo era realizar una gran cantidad de cálculos numéricos. Tiempo después las computadoras podían almacenar grandes cantidades de datos de manera persistente en dispositivos de almacenamiento secundario. Individuos y organizaciones utilizaban los sistemas de cómputo para almacenar grandes cantidades de información. Se requirieron formas eficientes para organizar,

manipular y recuperar la información. Fue así como surgieron los sistemas de bases de datos.

Los sistemas de bases de datos se encargaron de conjuntar herramientas de software con el propósito de simplificar la tarea de manipular información. Surgió el modelo relacional de bases de datos, bajo este modelo se resolvieron muchos de los problemas de eficiencia y se introdujo un modelo matemático para representar la información a almacenar. El modelo relacional está muy apegado a la forma en como se almacena la información físicamente aunque se puede apoyar del modelo entidad-relación para darle una semántica a ésta.

Hoy en día las computadoras están muy apegadas a la humanidad. Una gran mayoría de tareas realizadas por el hombre se apoyan en los sistemas de cómputo. Las necesidades han cambiado, se tiene una gran cantidad de información disponible y dispersa en una enorme cantidad de sitios. La cantidad de información es colosal así que se tiene que hacer uso de sistemas computacionales para obtenerla , interpretarla y analizarla. De aquí que, se requiera una comunicación entre sistemas más que entre humanos.

Es así como han surgido tecnologías como XML, Web semántica, minería de datos, minería de la Web, datos semiestructurados, etc. Esta necesidad ha provocado el surgimiento de modelos de información más apegados a la semántica que al almacenamiento físico. Es decir, es necesario hacer más pequeña la brecha entre estructura y semántica, o entre información y conocimiento.

Conocimiento, inteligencia, aprendizaje y razonamiento son conceptos muy relacionados, intuitivamente se sabe qué son, pero resulta muy difícil definirlos con precisión. Su significado puede dar lugar a una gran cantidad de trabajos y temas de discusión.

Uno de los principales objetivos que se busca en la ciencias de la computación, es la creación de máquinas inteligentes, la inteligencia Artificial es la encargada de este propósito. Lo que busca la inteligencia artificial es que las computadoras puedan realizar tareas que por el momento hace mejor el hombre [10] (es decir requieren de inteligencia humana). Por ejemplo: diagnosticar una enfermedad, jugar ajedrez, conducir un automóvil, etc. La inteligencia artificial también se encarga de tareas que son difíciles tanto para el hombre como para las máquinas.

Muchas de las tareas que se considera necesitan de inteligencia se han resuelto utilizando técnicas clásicas de computación, por lo que ha surgido el debate acerca de si se trata de verdadera inteligencia o no. Los límites de la inteligencia artificial no están claramente delimitados, tal vez, para comprender su verdadero significado primero sea necesario comprender los mecanismos fisiológicos del cerebro humano y los procedimientos de la mente. Hoy en día la inteligencia artificial se ocupa de dos grandes problemas: el desarrollo de buenos algoritmos de búsqueda y la representación del conocimiento [10].

2.2. El concepto de conocimiento

El conocimiento se construye sobre la información y la información sobre los datos [9] (ver fig. 2.1). Los datos en una computadora se pueden ver como secuencias de bytes, la información brinda un significado y una organización a los datos. Las bases de datos se ubican en el nivel de información. En el nivel de conocimiento se interpreta la información. En varios casos se puede generar información nueva, a este proceso se la llama inferencia o razonamiento.

Figura 2.1: Niveles de conocimiento

Existen dos tipos de conocimiento: el conocimiento a priori y el conocimiento a posteriori. El conocimiento a priori se considera como verdad absoluta o universal, es decir, consta de una serie de hechos que no se pueden negar (como los axiomas matemáticos). El conocimiento a posteriori es aquel en el que la verdad o falsedad se puede verificar por medio de ciertos mecanismos.

Del término conocimiento surge también el término *Base de Conocimiento*, los sistemas expertos están construidos con bases de conocimiento. Regularmente una base de conocimiento consta de un conjunto de hechos (conocimiento a priori) y un conjunto reglas de inferencia (conocimiento a posteriori). La representación del conocimiento consiste en la construcción de estructuras computacionales para apoyar estos dos fines.

2.3. Representación del conocimiento

Al igual que los datos semiestructurados, los conceptos en la mente humana no poseen una estructura rígida. Los conceptos se enriquecen, se eliminan, se modifican o se adaptan dependiendo de la interacción con la realidad. Los conceptos se pueden mezclar y entrelazar unos con otros. Para la representación del conocimiento, se utilizan varias estructuras como: reglas, redes semánticas, marcos, guiones, gráficas conceptuales, etc. En esta sección se propone el uso de datos semiestructurados como una alternativa para representar conocimiento.

Los datos semiestructurados almacenan información de manera sumamente flexible, pueden almacenar información incompleta y no especificada. De aquí que, el uso de bases de datos semiestructurados y sus operaciones pueden ser de gran ayuda para la construcción de bases de conocimiento.

Una base de datos semiestructurados puede ser utilizada para representar hechos. Las operaciones sobre la base de datos se pueden utilizar para implementar los mecanismos de inferencia, dependiendo del dominio en el que se encuentre el problema a resolver.

Las redes semánticas son gráficas dirigidas, en donde los nodos representan objetos y las aristas representan vínculos o conceptos. Esto también se puede representar en los datos semiestructurados.

Ejemplo 2.3.1 En la figura 2.2 se muestra una red semántica, el dato semiestructurado equivalente se muestra en la figura 2.3.

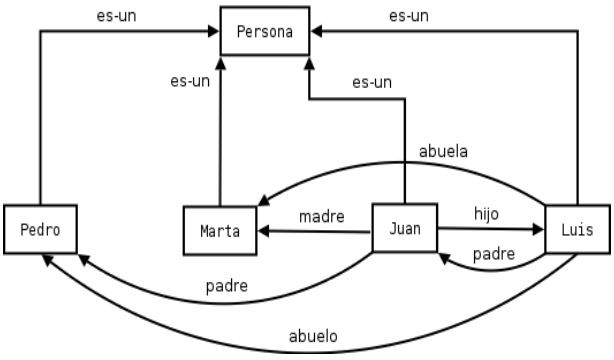

Figura 2.2: Red semántica

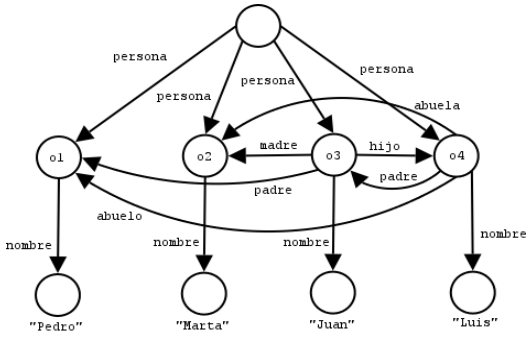

Figura 2.3: Dato semiestructurado equivalente a la red semántica en la figura 2.2

Los marcos son estructuras de datos formadas por *ranuras*[1] y *rellenos*[2] (estos son equivalentes a los campos y valores de un registro en una base de datos relacional orientada a objetos), un marco define un objeto con una serie de características. Los datos semiestructurados al tener subdatos etiquetados se pueden utilizar también de esta manera.

Ejemplo 2.3.2 El marco siguiente es equivalente al dato semiestructurado de la figura 2.4.

[1]Ranura es la traducción del término en inglés *slot*.
[2]Relleno viene del término en inglés *facet*

Ranuras	Rellenos
fabricante	Ford
modelo	Mustang
año	1982
transmición	manual
motor	V8
color	rojo

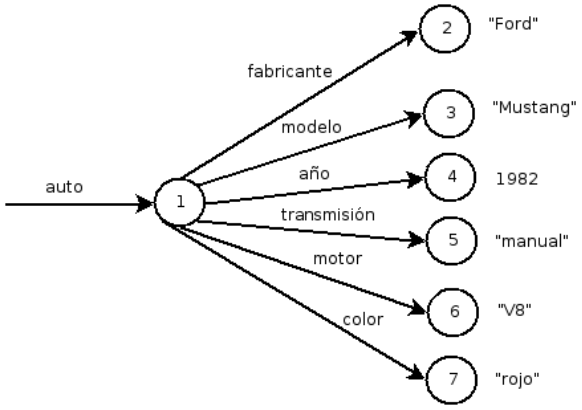

Figura 2.4: Dato semiestructurado equivalente a un marco

Una forma en la cual los datos semiestructurados se pueden utilizar perfectamente es en la descripción de taxonomías.

Ejemplo 2.3.3 La figura 2.5 representa una taxonomía muy sencilla, donde se puede ver que el pingüino aunque sea un ave, no vuela.

Se ha visto como los datos semiestructurados presentan una funcionalidad similar a otras estructuras comúnmente utilizadas en inteligencia artificial, para representar conocimiento. Así se puede concluir que, el tener bases de datos semiestructurados puede apoyar el desarrollo de sistemas basados en conocimiento.

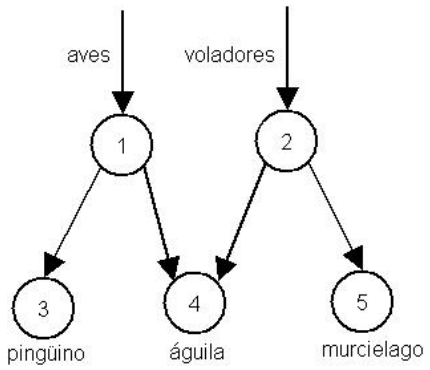

Figura 2.5: Representación de una taxonomía

Resumen

En este capítulo se ha discutido acerca de las necesidades actuales de información. Se explicó el concepto de conocimiento y se propuso el uso de datos semiestructurados como una alternativa para su representación. También se ejemplificó, cómo los datos semiestructurados pueden proveer una funcionalidad similar a las estructuras comúnmente utilizadas en inteligencia artificial, para representar conocimiento.

Durante la realización de este trabajo, se encontró que los datos semiestructurados poseen propiedades, que se pueden utilizar para afrontar los problemas actuales de información. Sus mismas propiedades permiten utilizarlos como una alternativa a las estructuras clásicas que se utilizan para representar el conocimiento.

Los capítulos subsecuentes tratan sobre el diseño de los módulos básicos que conforman un manejador de bases de datos semiestructurados. Dicha herramienta podría materializar las ventajas antes presentadas, para su utilización en aplicaciones del mundo real.

Capítulo 3

El Núcleo de un Sistema Manejador de Bases de Datos Semiestructurados

En este capítulo, se define lo que es el núcleo de un sistema manejador de bases de datos semiestructurados y sus componentes. También, se aborda el problema de diseñar una interfaz por medio de la cual el núcleo pueda ser utilizado.

La interfaz consta de una serie de operaciones básicas o primitivas. Estas operaciones deben de ser lo suficientemente expresivas para poder realizar todas las operaciones que el usuario pudiera requerir y al mismo tiempo deben ser lo suficientemente sencillas para poder permitir su implementación de manera accesible y eficiente, en colaboración con los demás componentes del núcleo.

3.1. Descripción del núcleo

Los sistemas manejadores de bases de datos incorporan una gran variedad de herramientas que facilitan el uso y administración de las bases de datos. Todas estas herramientas están construidas sobre un núcleo funcional que maneja las funciones básicas que provee el manejador de bases de datos.

De acuerdo con el diseño realizado en este trabajo, el *núcleo del sistema manejador de bases de datos* consta de los siguentes componentes básicos. La arquitectura se muestra en la figura 3.1.

- Interfaz.
- Manejo del almacenamiento.
- Control de concurrencia.
- Sistema de recuperación.

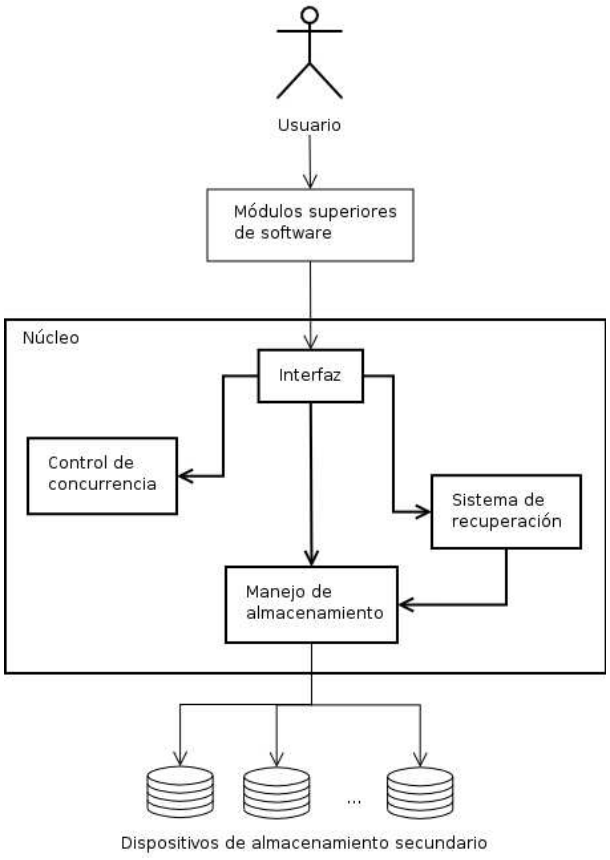

Figura 3.1: Arquitectura del núcleo

La interfaz se encarga de brindar una serie de métodos para que el núcleo pueda se utilizado por otros módulos de software, ésta se aborda en el presente capítulo.

El manejo de almacenamiento provee las construcciones y mecanismos necesarios para el almacenamiento persistente de las bases de datos en dispositivos de almacenamiento secundario, que por lo general son discos duros. El manejo de almacenamiento se trata con detalle en el capítulo 4

El control de concurrencia permite que varios usuarios pueden acceder al mismo tiempo a una base de datos sin que las acciones que realicen afecten la integridad de la información. El control de concurrencia se aborda en el capítulo 5.

El sistema de recuperación permite que el sistema manejador de bases de datos se pueda recuperar y dejar la base de datos en un estado consistente después de haber ocurrido un fallo. El sistema de recuperación se trata en el capítulo 5.

El objetivo de un sistema manejador de bases de datos es simplificar al usuario la manipulación de bases de datos, sin que se tenga que preocupar por cuestiones como la eficiencia, la concurrencia o la corrupción de la información. En lugar de esto, el usuario simplemente se tiene que ocupar de las operaciones que le brinda el modelo de datos que utiliza su base de datos.

3.2. La interfaz del núcleo y las primitivas

Para interactuar con módulos superiores de software (como el compilador o intérprete del lenguaje de consulta) el núcleo de un manejador de bases de datos semiestructurados debe proveer una serie de operaciones básicas o *primitivas* para su funcionamiento. En el caso de bases de datos relacionales estas están dadas por el álgebra relacional. Este trabajo también tiene la intención de definir un conjunto de operaciones primitivas para las bases de datos semiestructurados, las cuales se exponen a continuación.

3.2.1. Creación de datos semiestructurados

Las operaciones para creación de datos son necesarias para la incorporación de nuevos datos semiestructurados a la base de datos. Debido a que se tienen varios tipos de datos semiestructurados, es necesario contar con multiples tipos de operaciones de creación. Todas las operaciones de creación regresan el identificalor del dato recién creado para su posterior utilización. Las operaciones para creación de datos definidas para este sistema manejador de bases de datos semiestructurados son:

- `CREATE_INTEGER`. Crea un dato primitivo simple correspondiente a un número entero. A esta operación se le tiene que proporcionar el valor del entero a crear. Por ejemplo: `CREATE_INTEGER(4)`.
- `CREATE_REAL`. Crea un dato primitivo simple correspondiente a un número real o de punto flotante. También se le tiene que pasar el número de punto flotante a crear. Por ejemplo: `CREATE_INTEGER(3.1416)`.
- `CREATE_STRING`. Crea un dato primitivo simple correspondiente a una cadena de caracteres. Se debe proporcionar el valor del dato. Por ejemplo `CREATE_STRING("hello world")`.
- `CREATE_BYTESEQUENCE`. Crea un dato primitivo simple correspondiente a una secuencia de bytes. Se le debe pasar el valor del dato. Por ejemplo: `CREATE_BYTESEQUENCE(0,5,3)`.
- `CREATE_BLOB`. Crea un dato semiestructurado de tipo primitivo correspondiente a un BLOB, los BLOBs son objetos largos orientados a bytes. Ésta operación no recibe parámetros simplemente crea el dato para posteriormente ser llenado. Los BLOBs y CLOBs se manipulan de una manera similar a como se hace con los archivos de acceso aleatorio.
- `CREATE_CLOB`. Crea un dato semiestructurado de tipo primitivo correspondiente a un CLOB, los CLOBs son objetos largos orientados a caracteres. Al igual que la anterior, ésta operación no recibe parámetros, simplemente crea el dato para posteriormente ser llenado.
- `CREATE_NON_PRIMITIVE`. Crea un dato semiestructurado de tipo no primitivo. Ésta operación no recibe parámetros, simplemente crea el dato vacío para posteriormente ser llenado.

Ejemplo 3.2.1 Considérese la figura 3.2. Al aplicar `CREATE_REAL(3.1416)` a la base de datos que se muestra en la parte a) se obtiene el identificador 4, que es el que el sistema ha asignado al dato creado (ver parte b)).

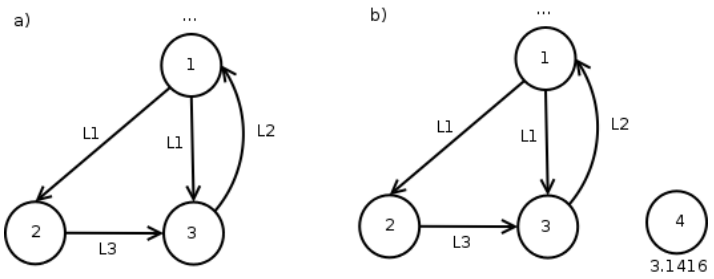

Figura 3.2: Ejemplo de creación de un dato semiestructurado

3.2.2. Manejo del contenido de los datos

Las primitivas para el manejo de datos son las que se encargan de la modificación de la información almacenada en la base de datos semiestructurados. Las primitivas para el manejo de contenido son:

- `ADD`. Trabaja con datos semiestructurados no primitivos, añade un dato con una etiqueta especifica al contenido de otro. Recibe el identificador del dato cuyo contenido será modificado, la etiqueta y el identificador del dato a incluir.

  Ejemplo 3.2.2 Considérese la figura 3.3. Al aplicar `ADD(1,L1,4)` a la base de datos que se muestra en la parte a), queda como se muestra en la parte b). Nótese que es importante el orden en el que se colocan los parámetros.

- `REMOVE_LABEL`. Trabaja con datos semiestructurados no primitivos. Lo que hace es eliminar del contenido de un dato todos los subdatos que estén etiquetados con una etiqueta dada. Recibe el identificador del dato cuyo contenido será modificado y la etiqueta a remover.

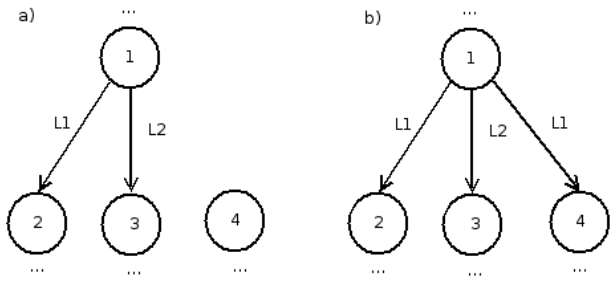

Figura 3.3: Ejemplo de la primitiva `ADD`

Ejemplo 3.2.3 Considérese la figura 3.4. Al aplicar la operación primitiva `REMOVE_LABEL(1,L1)` a la base de datos que se muestra en la parte a), queda como la parte b).

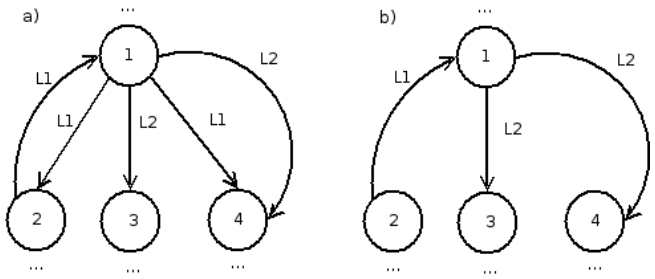

Figura 3.4: Ejemplo de la primitiva `REMOVE_LABEL`

- `REMOVE_ID`. Trabaja con los datos semiestructurados no primitivos. Lo que hace es eliminar del contenido de un dato todos los subdatos que correspondan a un identificador dado. Recibe el identificador del dato cuyo contenido será modificado y el identificador a remover.

  Ejemplo 3.2.4 Al aplicar `REMOVE_ID(1,4)`, a la base de datos que se muestra en la parte a) de la figura 3.5, queda como se muestra en la parte b).

- `REMOVE`. Trabaja con datos no primitivos. Elimina del contenido de un dato el subdato con etiqueta e identificador específicos. Recibe el

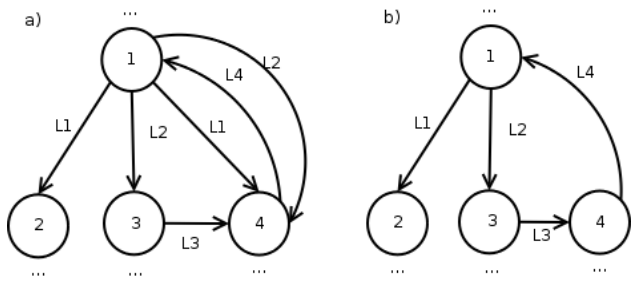

Figura 3.5: Ejemplo de la primitiva `REMOVE_ID`

identificador del dato cuyo contenido será modificado, la etiqueta y el identificador del dato a remover.

Ejemplo 3.2.5 Al aplicar `REMOVE(1,L1,4)`, a la base de datos que se muestra en la parte a) de la figura 3.6, queda como se muestra en la parte b).

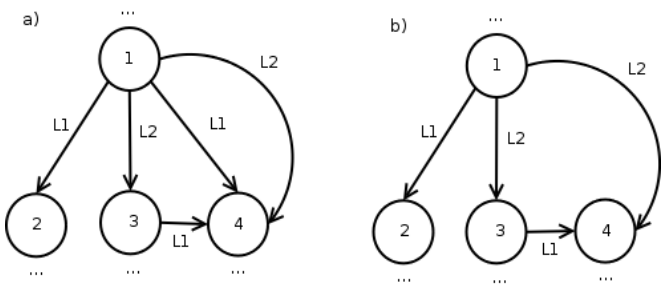

Figura 3.6: Ejemplo de la primitiva `REMOVE`

- `GET_TYPE`. Para un dato específico, regresa un número entero que corresponde con el tipo del dato. Recibe el identificador del dato a analizar. El número que regresa puede ser alguna de las siguientes constantes:
  - `NON_PRIMITIVE`: Para los datos no primitivos.
  - `INTEGER`: Para los números enteros.
  - `REAL`: Para los números reales.

- `STRING`: Para las cadenas de caracteres.
- `BYTE_SEQUENCE`: Para las secuencias de bytes.
- `BLOB`: Para los BLOBs.
- `CLOB`: Para los CLOBs.

- `IS_PRIMITIVE`. Para un dato semiestructurado específico, regresa verdadero si es primitivo o falso si no lo es. Recibe el identificador del dato a verificar.

- `GET_CONTENT`. Regresa el contenido de un dato semiestructurado, dado su identificador. Como se tienen diferentes tipos de datos, lo que ésta primitiva puede regresar es variado, es decir, se trata de un método polimórfico. En caso de tipos no primitivos de datos se regresa una colección de datos semiestructurados etiquetados, en caso de tipos primitivos simples regresa su valor y en el caso de LOBs se regresa un descriptor para manipularlos. Al momento de implementación se regresa una estructura estándar, con la cual se pueda tener acceso a todos los tipos.

3.2.3. Verificación de parentescos

Las primitivas de verificación de parentescos son útiles para verificar las relaciones existentes entre los datos. Los datos semiestructurados por naturaleza están llenos de relaciones de parentesco lo cual da mucha flexibilidad para representar información. La primitivas para la verificación de parentescos son:

- `CONTAINS`. Recibe un identificador, una etiqueta y otro identificador. Verifica sí el dato correspondiente al primer dato contiene al dato correspondiente al segundo etiquetado con la etiqueta dada.

  Ejemplo 3.2.6 Considérese la figura 3.7. Al aplicar la operación primitiva `CONTAINS(1,L2,3)` se obtiene verdadero y al aplicar la operación `CONTAINS(1,L1,3)` se obtiene falso.

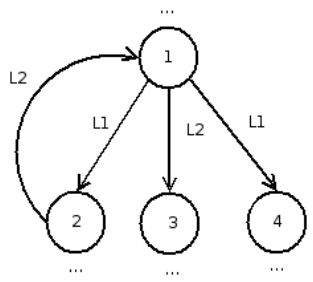

Figura 3.7: Ejemplo para la aplicación de las primitivas `CONTAINS` y `BELONGS`

- `CONTAINS_ID`. Recibe dos identificadores de datos semiestructurados. Verifica sí el dato correspondiente al primer identificador es padre del correspondiente al segundo.

  Ejemplo 3.2.7 Considérese la figura 3.7. Al aplicar `CONTAINS_I(1,3)` se obtiene verdadero, al aplicar `CONTAINS_ID(3,1)` se obtiene falso, al aplicar `CONTAINS_ID(1,2)` se obtiene verdadero y al aplicar la operación `CONTAINS_ID(2,1)` tambien se obtiene verdadero.

- `CONTAINS_LABEL`. Recibe un identificar y una etiqueta. Verifica sí el dato correspondiente al identificador contiene algún dato con la etiqueta dada.

  Ejemplo 3.2.8 Considérese la figura 3.7. Al aplicar la operación primitiva `CONTAINS_LABEL(1,L1)` se obtiene verdadero, al aplicar la operación `CONTAINS_LABEL(1,L3)` se obtiene falso.

- `BELONGS`. Recibe dos identificadores de datos semiestructurados. Verifica si el dato correspondiente al primer identificador es hijo del correspondiente al segundo.

  Ejemplo 3.2.9 Considérese la figura 3.7. Al aplicar `BELONGS(3,1)` se obtiene verdadero, al aplicar `BELONGS(1,3)` se obtiene falso, al aplicar `BELONGS(2,1)` se obtiene verdadero y al aplicar `BELONGS(1,2)` también se obtiene verdadero.

- `GET_PARENTS`. Regresa una colección que contiene los identificadores de los padres de un dato semiestructurado especificado. Recibe el identificador del dato del cual se desea obtener la lista de padres.

  Ejemplo 3.2.10 Al aplicar `GET_PARENTS(2)` a la base de datos de la figura 3.8, se obtiene la colección de identificadores $\{1, 2, 3\}$.

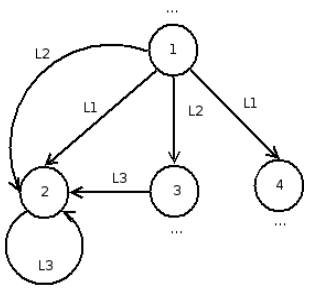

Figura 3.8: Ejemplo para la aplicación de la primitiva `GET_PARENTS`

3.2.4. Manejo del diccionario de datos

El diccionario de datos es un componente muy importante de las bases de datos semiestructurados, es aquí donde se definen las *ssd-tablas*. Las cuales sirven como puntos de acceso para toda la información contenida en la base de datos. Las primitivas para el manejo del diccionario de datos son:

- `ADD_SSDTABLE`. Añade una nueva *ssd-tabla* al diccionario de datos. Recibe el nombre de la *ssd-tabla* y el identificador del dato que corresponde a la raíz.

  Ejemplo 3.2.11 Considérese la figura 3.9. Aplicando la operación primitiva `ADD_SSDTABLE(T3,4)` a la base de datos de la parte a), el resultado es como se muestra en la parte b).

- `REMOVE_SSDTABLE`. Elimina una *ssd-tabla* del diccionario. Recibe el nombre de la *ssd-tabla*.

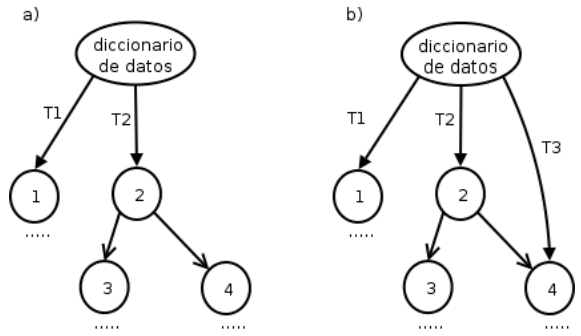

Figura 3.9: Ejemplo de la aplicación de la primitiva `ADD_SSDTABLE`

Ejemplo 3.2.12 Considérese la figura 3.10. Aplicando la operación primitiva `REMOVE_SSDTABLE(T3)` a la base de datos de la parte a), el resultado es como se muestra en la parte b).

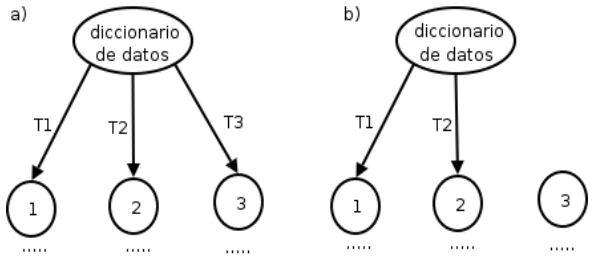

Figura 3.10: Ejemplo de la aplicación de la primitiva `REMOVE_SSDTABLE`

- `GET_ROOT`. Regresa el identificador correspondiente a la raíz de una *ssd-tabla*. Recibe el nombre de la *ssd-tabla*.

  Ejemplo 3.2.13 Si se aplica `GET_ROOT(T3)` a la base de datos de la figura 3.11, se obtiene el identificador 3.

- `REMOVE_SSDTABLE`. Recibe un identificador y elimina las *ssd-tablas* del diccionario que tengan como raíz al identificador dado.

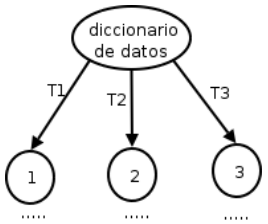

Figura 3.11: Ejemplo para la aplicación de la primitiva `GET_ROOT`

3.2.5. Eliminación de datos

Debido a la gran cantidad de parentescos que se pueden presentar en una base de datos semiestructurados, la eliminación de datos es un proceso delicado. La eliminación de un dato puede causar la eliminación de otros o la presencia de datos inalcanzables. Para la implementación de la eliminación, también, se tiene que prestar mucho cuidado para evitar inconsistencias en la base de datos. Solamente se incluye una primitiva de borrado que se explica a continuación:

- `DROP`. Elimina un dato semiestructurado de la base de datos, para ello recibe el identificador. Para evitar inconsistencias, se asume que el dato a borrar está aislado, es decir no tiene padres ni hijos. Para los borrados más complejos de datos se deja la responsabilidad a los módulos superiores.

  Ejemplo 3.2.14 En la figura 3.12, se muestra la eliminación del dato con identificador 3, en la parte a) se muestra el estado original y en la b) el estado después del borrado.

3.2.6. Manejo de transacciones

El concepto de transacción se explica con detalle en el capítulo 5. Básicamente las transacciones ofrecen mecanismos para agrupar operaciones de manera atómica, cada operación que se realiza en la base de datos está incluida dentro de una transacción. El manejador al nivel de interfaz se encarga de

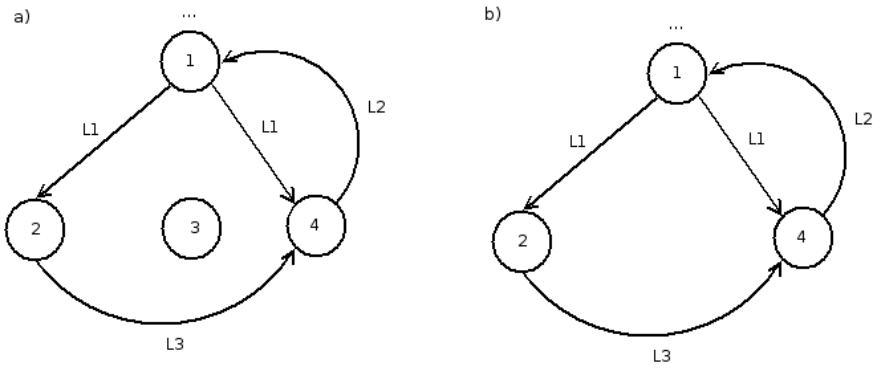

Figura 3.12: Ejemplo de la aplicación de la primitiva `DROP`

crear nuevas transacciones según las primitivas que realice el usuario. A continuación se describen las primitivas para el manejo de transacciones, cada una de ellas causa que la transacción actual termine.

- `COMMIT_TRANSACTION`. Ordena al manejador que comprometa la transacción actual. Es decir, informa que la transacción se ha completado con éxito y que debe mantener los cambios hechos a la base de datos.
- `ROLLBACK_TRANSACTION`. Ordena al manejador deshacer la transacción actual. Es decir, los cambios hechos a la base de datos por la transacción actual tienen que ser deshechos por el manejador.

La implementación de las primitivas discutidas en este capítulo se traducen en operaciones efectuadas en los demás módulos que conforman el núcleo del sistema manejador de bases de datos semiestructurados. Estas primitivas definen la interfaz, que deberán utilizar otros módulos de software para interactuar con el núcleo

Resumen

En este capítulo, se definió lo que es el núcleo de un sistema manejador de bases de datos semiestructurados. Se explicaron de manera general sus

componentes: interfaz, manejo de almacenamiento, control de concurrencia y sistema de recuperación.

Se expusieron una serie de primitivas para interactuar con el núcleo. Estas primitivas son el resultado de un proceso de diseño, en el que también se consideraron los demás componentes. Las primitivas fueron cuidadosamente diseñadas para mantener un equilibrio entre expresividad y factibilidad de implementación. Finalmente las primitivas expuestas fueron implementadas en el módulo de software correspondiente (véase el apéndice A).

El siguiente capítulo, aborda el problema de almacenar los datos semiestructurados de manera persistente, en dispositivos de almacenamiento secundario. Esto tiene el propósito de que las operaciones realizadas por las primitivas se mantengan en la base de datos.

Capítulo 4

El manejo de Almacenamiento

En este capítulo se aborda el problema más importante de este trabajo, que es el almacenamiento de bases de datos semiestructurados en dispositivos de almacenamiento secundario. El almacenamiento de bases de datos semiestructurados involucra varios subproblemas como: el uso de memoria intermedia, la agrupación de múltiples dispositivos, el uso de particiones crudas y cocidas, el manejo del espacio libre, la manera de representar los datos semiestructurados a un bajo nivel de almacenamiento y la incorporación de objetos largos (BLOBs y CLOBs). Todos estos problemas se tratan en este capítulo.

El *manejo de almacenamiento* provee al manejador de bases de datos el almacenamiento persistente de la información contenida en la base de datos. Lo que concierne en este trabajo es el diseño y construcción del manejo de almacenamiento orientado hacia las bases de datos semiestructurados. Para lograr dicho propósito se diseñó e implementó un mecanismo de almacenamiento a varios niveles: Primitivo, Avanzado y de Base de Datos.

4.1. El Almacenamiento Primitivo

La mayoría de los lenguajes de programación, permiten el acceso a los dispositivos de almacenamiento secundario byte por byte. Sin embargo, internamente el almacenamiento de información en estos dispositivos se rea-

liza mediante unidades de información de un tamaño específico. A nivel de hardware (generalmente hablando de discos duros) esta unidad es llamada *sector*. Un sector es la mínima unidad de información que se puede leer o escribir en el disco, el tamaño del sector varía según el tipo de disco, ejemplos de tamaño de sector son 512b, 1K, 2K, 4K, etc.

Los sistemas operativos implementan también sus propias unidades mínimas de lectura-escritura. A nivel de sistema operativo estas unidades son llamadas *bloques*, un bloque puede estar formado por uno o varios sectores, dependiendo de la configuración del sistema de archivos. Generalmente los sistemas operativos tienen la opción de ajustar el tamaño de bloque que usarán dependiendo de sus necesidades. Por ejemplo para utilizar el sistema de cómputo como un servidor de bases de datos se suele aumentar el tamaño de bloque hasta 32K o más, dependiendo de la cantidad de información que se espera almacenar. El tamaño de bloque más común es de 4K.

Para el manejo del almacenamiento los sistemas manejadores de bases de datos tienen dos posibilidades principales:

- Utilizar el sistema de almacenamiento que brinda el sistema operativo, a través, del sistema de archivos.
- Implementar su propio sistema de almacenamiento.

Cuando el manejador de bases de datos implementa su propio sistema de almacenamiento, también, se define una unidad mínima de lectura-escritura. En el contexto del manejador de bases de datos, esta unidad es llamada *página*. Una página puede estar formada por uno o varios bloques.

En los sistemas manejadores de bases de datos, el tamaño de la página se ajusta de acuerdo a las necesidades de espacio que requieren las bases datos a almacenar. Para grandes cantidades de información se usan tamaños de página grandes y para cantidades relativamente pequeñas de información se utilizan tamaños de página pequeños. El objetivo de esto es el de minimizar los accesos a disco que realizará el manejador. Un tamaño de página grande puede incrementar los tiempos de lectura-escritura al dispositivo de almacenamiento secundario, debido a que depende del tamaño de bloque que use el sistema operativo; pero también permite reducir el número de accesos a disco, si se tiene una buena organización de la información y del espacio libre.

4.1.1. Manejo de Memoria Intermedia

Para aumentar el rendimiento del almacenamiento, los sistemas computacionales utilizan *buffers*. Un buffer es una área de memoria de almacenamiento temporal. En este desarrollo se utiliza un buffer para almacenar una cierta cantidad de páginas. Para atender las peticiones de entrada-salida de los procesos, primero se verifica que la página se encuentre en el buffer, de no ser así, se tiene que realizar un acceso a disco para recuperar la página. Como el tamaño del buffer es limitado no todas las páginas pueden mantenerse en él. Para elegir qué páginas mantener en el buffer y qué páginas no, se utilizan los *algoritmos de reemplazo de páginas* [8, 7]. El objetivo de estos algoritmos es que se realice la menor cantidad posible de reemplazos de página, lo que se refleja también en una baja cantidad de accesos a disco. Un acceso a disco es aproximadamente 10000 veces más lento que un acceso a memoria, de aquí que, la elección de un algoritmo de reemplazo de páginas adecuado se refleja en una mejora de la velocidad de almacenamiento.

Existen varios algoritmos de reemplazo de páginas con distintas propiedades. La elección del más adecuado depende del comportamiento de la base de datos, en muchos casos esta elección depende de un largo tiempo de observación por parte del administrador. En la implementación del sistema que compone este trabajo se pueden elegir los algoritmos: *LRU*, *MRU*, *FIFO*, *LIFO* y *CLOCK*. El algoritmo de reemplazo de páginas más comúnmente utilizado es el LRU y es el que se utiliza por omisión en este sistema. Se deja la posibilidad de utilizar algún otro algoritmo, en caso de determinarse que funciona mejor con el comportamiento del sistema manejador de bases de datos semiestructurados.

A continuación se explican con detalle los algoritmos de reemplazo de páginas:

- LRU (Least Recent Used). Con este algoritmo, se reemplaza la página menos recientemente utilizada. Las páginas se pueden formar en una fila, de tal manera que, cada que se utilice una página ésta pasa a la cabeza. Cuando se solicite una página que no se encuentra en el buffer, se toma la del final de la fila y se reemplaza por la página pedida, la cual pasa a la cabeza de la fila. El principio básico de este algoritmo dice que lo que ocurrió en el pasado refleja lo que sucederá en el futuro,

es decir, si una página ha sido utilizada recientemente es muy probable que pronto se vuelva a utilizar. Este es el algoritmo más comúnmente usado en los sistemas operativos y en los manejadores de bases de datos.

- MRU (Most Recent Used). Este algoritmo es el opuesto al anterior. Aquí la página que se utilizo al último es la que se reemplaza. Cuando se usa una página ésta se coloca a la cabeza de la fila, la cabeza de la fila es la siguiente a reemplazar. Este algoritmo resulta conveniente en el caso de que se requieran datos que sólo se acceden una vez durante un proceso.

- FIFO (First In First Out). En este algoritmo, se mantiene una fila de las páginas que se han usado de acuerdo a su orden de llegada. Cuando llega una nueva página ésta se coloca al final de la fila. Cuando se requiere un reemplazo de página, se elimina la que se encuentra a la cabeza y la nueva se coloca al final. A diferencia del algoritmo LRU, cuando se requiere una página que se encuentra en el buffer ésta no cambia de posición en la fila.

- LIFO (Last In First Out). En este algoritmo, las páginas nuevas se colocan al final de la fila y la última página en la fila es la que será reemplazada. A diferencia del algoritmo MRU en caso de que la página requerida se encuentre en el buffer, no se modifica su posición en la fila.

- CLOCK. Este algoritmo es llamado del reloj o LRU con segunda oportunidad. La idea es que por cada página se asocia un bit adicional de referencia. Se suele implementar con una lista circular. Se mantiene un apuntador a la página a reemplazar, si dicha página tiene el bit apagado entonces se reemplaza; si el bit se encuentra prendido entonces el bit se apaga y el apuntador se recorre al siguiente elemento de la lista. El proceso se repite hasta encontrar una página adecuada para reemplazar. La nueva página ocupa el lugar de la reemplazada, se prende su bit y el apuntador se recorre al siguiente elemento. Cuando se usa una página que se encuentra en el buffer simplemente se prende su bit.

El proceso de *paginación* consiste básicamente en buscar la página solicitada en el buffer, si no se encuentra se tiene que traer del dispositivo de almacenamiento secundario y reemplazarla por una existente en el buffer. Para esto último se usa el algoritmo de reemplazo de páginas. Los sistemas

computacionales brindan las funciones de paginación por medio del hardware usando circuitos especializados, sin embargo, los manejadores de bases de datos tienen que recurrir a técnicas de software para implementarlos. Esto debido a que los manejadores funcionan sobre el sistema operativo, que se encarga del manejo del hardware restringiendo a los procesos para su uso. El buffer del manejador de bases de datos, tiene que realizar funciones muy similares a las del hardware de paginación, pero específicamente para la información contenida en las bases de datos, utilizando para ello la memoria. Para localizar una página solicitada, el hardware de paginación está diseñado de tal manera que no tarde mucho más que un acceso a memoria, pero en software se tienen que utilizar estructuras de datos para efectuar la búsqueda lo más rápidamente posible.

En este trabajo, se utiliza una estructura de datos doble formada por una tabla de dispersión (hash) y una lista circular doblemente ligada (ver fig. 4.1). La tabla de dispersión sirve para localizar las páginas dentro del buffer de acuerdo al número o dirección de página. La lista circular sirve para implementar los diversos algoritmos de reemplazo de páginas (LRU, MRU, FIFO, LIFO o CLOCK). También por cada página se almacenan dos bits: uno de referencia y otro de escritura. El bit de escritura se prende solamente cuando se realiza un cambio al contenido de la página, de tal manera que una página cuando es retirada del buffer se actualiza en el disco sólo si su bit de escritura está prendido. De esta manera, se ahorran más accesos a disco.

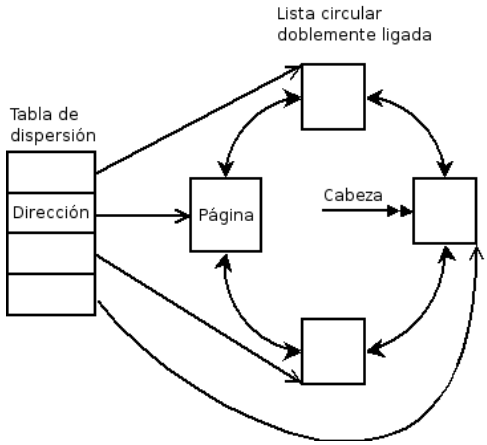

Figura 4.1: Implementación del buffer

Para implementar los algoritmos de búsqueda utilizando la lista circular, se tiene que variar un poco la forma de reorganizar su estructura después de un reemplazo. La lista circular mantiene una cabeza; cuando se utiliza LRU o FIFO, el elemento a reemplazarse es el que se encuentra antes de la cabeza, es decir, el último. Cuando se usa MRU o LIFO se reemplaza la cabeza. Cuando se usa CLOCK, la cabeza es la candidata a reemplazarse y ésta se va recorriendo hasta encontrar la página apropiada para el reemplazo. Cuando se usa una página que se encuentra en el buffer, si se usa MRU o LRU ésta se reposiciona antes de la cabeza, si se usa MRU se convierte en la nueva cabeza.

4.1.2. Uso de Múltiples Dispositivos

En ocasiones, el tamaño de una base es tan grande que no se puede almacenar en un solo dispositiv. Algunos manejadores de bases de datos tienen mecanismos para agrupar varios dispositivos en una misma base de datos. Generalmente los dispositivos de almacenamiento más que corresponder a un disco corresponden a una partición. Existen dos tipos de particiones:

- *Particiones crudas*. Son aquellas que el manejador de bases de datos controla directamente, sin utilizar el sistema de archivos brindado por el sistema operativo. El manejador de bases de datos da formato a este tipo de particiones.
- *Particiones cocidas*. Son aquellas que se montan sobre el sistema de archivos que brinda el sistema operativo. El sistema operativo es el que se encarga de dar formato a estas particiones.

Lo más eficiente para un manejador de bases de datos es el uso de particiones completamente crudas, sin intervención alguna del sistema operativo, ya que así podría implementar sus propias estructuras especializadas para su propósito. El manejador de bases de datos es un programa que se tiene que ejecutar sobre el sistema operativo. El sistema operativo implementa su estructura de almacenamiento para múltiples programas con propósitos distintos, lo cual no es lo más adecuado para el manejador de bases de datos. Existen varias alternativas para el uso de particiones crudas y cocidas por parte del manejador de bases de datos, las dos más radicales son:

- El manejador de bases de datos usa el sistema de archivos dado por el sistema operativo.

- El manejador de bases de datos usa y da su propio formato a las particiones a utilizar.

Hay muchas alternativas intermedias a éstas, que son las que los manejadores generalmente usan. Por ejemplo una de ellas es el uso de un archivo (dado por el sistema de archivos) de gran tamaño y ahí crear el formato propio del manejador. No todos los sistemas operativos permiten el uso de particiones crudas, como Windows. Otros sistemas como los UNIX si permiten su uso y además se pueden utilizar como los otros dispositivos, es decir, se montan en el sistema de archivos y se tratan como si fueran archivos de acceso aleatorio.

La solución para agrupar múltiples dispositivos de almacenamiento, ya sean crudos o cocidos de este trabajo está basada en la que usa INFORMIX[1] [22]. Ésta se basa en el uso de *chunks*[2]. Un chunk es un archivo o parte de él, existen dos tipos de chunks: *chunks flexibles* y *chunks fijos*. Los chunks flexibles se construyen sobre un archivo cuyo tamaño no está determinado. Los chunks fijos están sobre una parte de tamaño fijo en un archivo.

La idea de los chunks flexibles es que se utilicen bajo el sistema de archivos dado por el sistema operativo, sujeto a sus respectivos límites (en particiones cocidas), tratando de dejar de lado la administración del almacenamiento. Esto es viable para el uso de bases de datos pequeñas. Los chunks fijos requieren de una administración detallada del almacenamiento, pero en sistemas como UNIX permiten la incorporación de particiones crudas.

El sistema manejador de bases de datos semiestructurados trata a los chunks como si fueran dispositivos de almacenamiento distintos, de esta manera, se pueden agrupar espacios incluso en diferentes discos o particiones. Los chunks son tratados como secuencias de páginas, para agrupar los diversos dispositivos, simplemente se define el grupo de chunks a utilizar. Para

[1]INFORMIX es un sistema manejador de bases de datos relacionales desarrollado por IBM, que ha sido muy utilizado en aplicaciones donde se requiere almacenar grandes cantidad de datos.

[2]Una traducción para la palabra *chunk* podría ser *pedazo*. Sin embargo, por abuso del lenguaje y dado que en la literatura dicho término aparece en inglés, a lo largo de este trabajo se utilizará *chunk*.

acceder a una página en específico se da el número de chunk y el número de página dentro de éste.

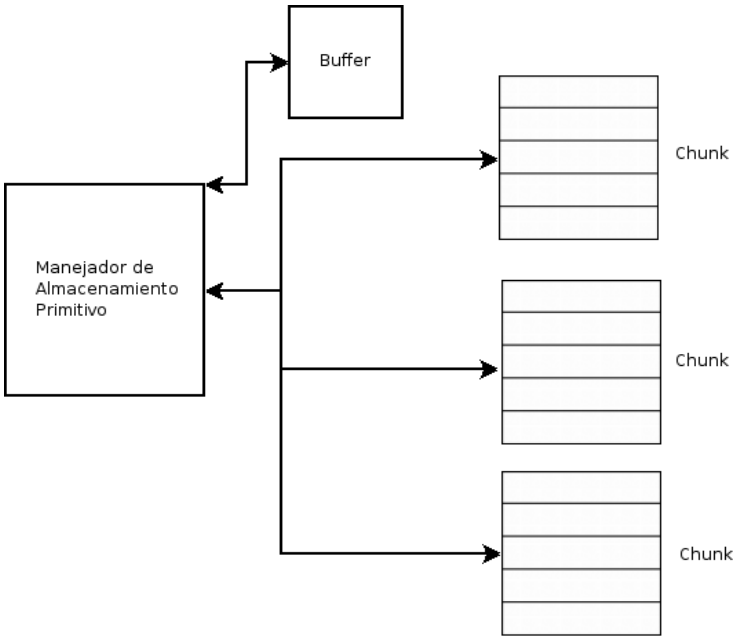

Figura 4.2: Almacenamiento primitivo

El almacenamiento primitivo está compuesto del grupo de chunks, el buffer para el manejo de la memoria intermedia y el manejador del almacenamiento primitivo. El manejador de almacenamiento primitivo se encarga de los remplazos de página y de implementar los mecanismos de entrada-salida (ver fig. 4.2). En el buffer se almacenan páginas de los diferentes chunks. En el módulo de software correspondiente se pueden elegir: el tamaño de página, el tamaño del buffer, el algoritmo de reemplazo de páginas y los chunks a utilizar.

4.2. El almacenamiento Avanzado

En las bases de datos semiestructurados, se requiere del almacenamiento de estructuras de tamaño no fijo, esto debido a la naturaleza de los datos.

La mayoría de las técnicas que se utilizan para guardar en disco estructuras de tamaño variable, consisten en distribuirlas en cajones de tamaño fijo. Esto con el fin de facilitar el acceso, el manejo de espacio libre y evitar la fragmentación externa.

Bajo la filosofía anterior, se diseñó el modelo de almacenamiento que se expone en esta sección. Su propósito es el de ser utilizado para las bases de datos semiestructurados, aunque puede ser usado para algún otro propósito que lo requiera. El *almacenamiento avanzado* está basado en tres niveles de organización:

- *Segmento.*
- *Espacio.*
- *Unidad.*

Un segmento es un arreglo de espacios y un espacio es un arreglo de unidades. La unidad corresponde a los cajones de tamaño fijo, en donde se va a distribuir la información. Una unidad puede estar libre u ocupada (ver fig. 4.3). Las unidades son del mismo tamaño para cada segmento, aunque unidades de segmentos distintos pueden tener tamaños distintos. El tamaño de la unidad, se determina según las necesidades de espacio de lo que se requiera almacenar en ella (por ejemplo se pueden almacenar: nodos de un árbol, páginas de un archivo, descriptores de datos, etc).

Para tener acceso a una unidad en particular, se deben especificar: el número de segmento, el número de espacio dentro del segmento y el número de unidad dentro del espacio. En este nivel de almacenamiento, se proporcionan mecanismos para leer o escribir en una unidad, encontrar una unidad libre, liberar u ocupar una unidad.

Para el propósito de almacenar bases de datos semiestructurados, la idea es tener cinco segmentos por cada base de datos para almacenar los siguientes componentes principales:

- Descriptores de datos.
- Contenidos de datos.

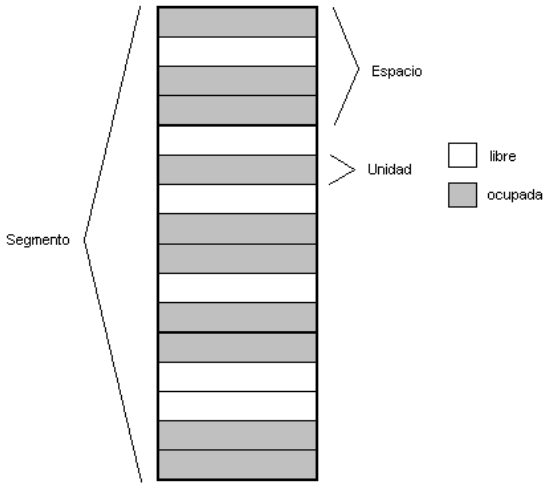

Figura 4.3: Modelo del almacenamiento avanzado

- Colecciones de padres.
- Contenidos de objetos largos (LOBs).
- Diccionario de datos.

En las unidades de estos segmentos, se encuentran las pequeñas partes que conforman los respectivos componentes.

4.2.1. Implementación del almacenamiento avanzado

Para implementar el modelo anterior, se hace uso del almacenamiento primitivo. Los espacios corresponden a chunks y las unidades se colocan dentro de las páginas. La restricción es que el tamaño de la unidad no debe exceder el tamaño de la página.

Cada espacio corresponde con un chunk, internamente un espacio tiene tres niveles de organización:

- *Área.*

- *Extensión.*
- *Página de unidades.*

Las áreas y extensiones fueron concebidas con el fin de implementar el manejo del espacio libre. El manejo del espacio libre hace uso intensivo de los *mapas de bits*. Un espacio está compuesto de áreas, una área está compuesta de extensiones y una extensión está compuesta de páginas de unidades. Una página puede ser una página de unidades, un mapa de bits o un descriptor, como se explicará a continuación.

La primera página del espacio es un descriptor, que contiene el número de la última unidad utilizada. Se asume que las unidades con números más grandes están libres. Este número se utiliza para calcular la parte usada del espacio sin necesidad de formatearlo completamente, solamente se da formato conforme el espacio se utiliza. Esto también es conveniente cuando no se sabe cual será el tamaño total del espacio, es decir, cuando se usan chunks flexibles.

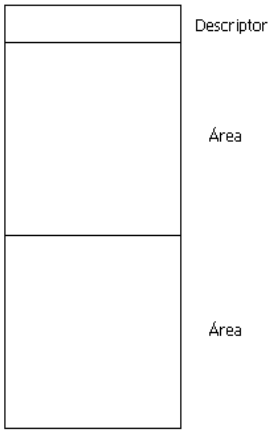

Figura 4.4: Distribución de un espacio

Un espacio consta de un descriptor y de áreas seguidas una de otra (ver fig. 4.4). La primera página de un área es un mapa de bits, que indica las extensiones que están disponibles y las que no. La posición de un bit en el mapa corresponde con la posición de la extensión en el área, un cero indica

que la extensión está disponible y un uno que está llena. Después de este mapa de bits, aparecen las extensiones una seguida de otra (ver fig. 4.5).

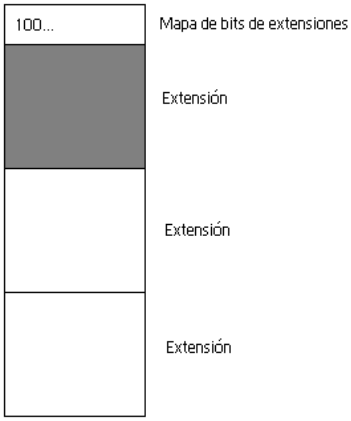

Figura 4.5: Distribución de una área

La primera página de una extensión es un mapa de bits, que indica qué páginas de unidades están disponibles y cuáles no. Un cero indica que la página de unidades está disponible y un uno que está llena. De igual manera, la posición del bit en el mapa corresponde con la posición de la página de unidades en la extensión. Después del mapa de bits, aparecen las páginas de unidades una después de otra (ver fig. 4.6).

Dentro de las páginas de unidades, puede haber una o más unidades. Si el tamaño de unidad permite que quepa más de una en una página, entonces hay un mapa de bits al final de la página de unidades, si solamente cabe una unidad entonces no aparece mapa de bits (ver fig. 4.7). El mapa de bits dentro de una página de unidades indica las unidades que están libres y las que están ocupadas, el cero indica que la unidad está libre y el uno que está ocupada. La posición en el mapa de bits indica la posición de la unidad dentro de la página de unidades.

El tamaño de las extensiones y áreas depende del tamaño de página. Si el tamaño de página es P (en bytes), el tamaño de una extensión es $(8P+1)P$, porque con una página se puede almacenar el estado de $8P$ páginas y se requiere una página para el mapa de bits. Análogamente el tamaño de un

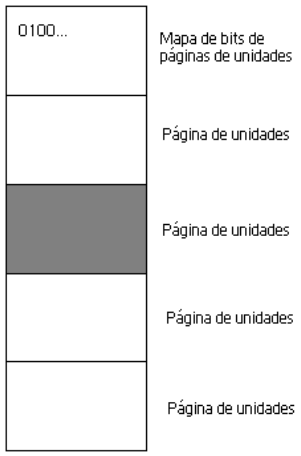

Figura 4.6: Distribución de una extensión

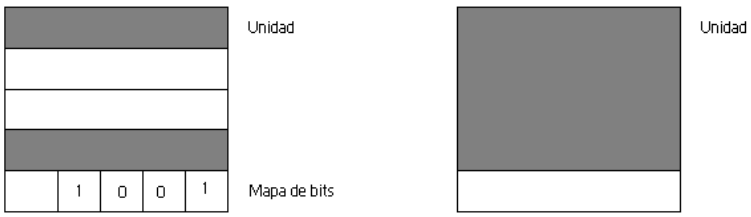

Figura 4.7: Páginas de unidades

área es $(8P(8P+1)+1)P$, ya que un área puede almacenar el estado de $8P$ extensiones y cada extensión tiene $8P+1$ páginas.

Para localizar una unidad libre en una área se tiene que acceder a su mapa de bits y encontrar una extensión disponible, después acceder al mapa de bits de la extensión y buscar una página de unidades disponible, ya que en la página de unidades se puede obtener una unidad libre. Si se supone que cada acceso a una página requiere de un acceso a disco, entonces localizar una unidad libre en un área toma tres accesos a disco, si no hay unidades libres sólo toma uno.

El cuadro 4.1 muestra algunos tamaños de página y el tamaño de área correspondiente. Se puede apreciar que con pocos accesos a disco se puede organizar el espacio libre en una gran cantidad de espacio. Por ejemplo, con una página de 4K sólo se necesitan tres accesos a disco para encontrar una unidad libre en 4 terabytes de información.

Tamaño de página	Tamaño de área
128 B	128.125122070 MB
256 B	1.000488520 GB
512 B	8.001953602 GB
1 KB	64.007813454 GB
2 KB	512.031251907 GB
4 KB	4.000122074 TB
8 KB	32.000488289 TB
16 KB	256.001953140 TB
32 KB	2.000007629 PB
64 KB	16.000030518 PB
128 KB	128.000122070 PB
256 KB	1.000000477 HB
512 KB	8.000001907 HB
1 MB	64.000007629 HB
2 MB	512.000030518 HB
4 MB	4.000000119 ZB

Cuadro 4.1: Tamaños de páginas y áreas

El acceso a una unidad conociendo su número dentro de un espacio, sólo requiere de un acceso a disco, ya que se puede calcular la página correspondiente considerando el número de área y extensión.

Los accesos a disco descritos anteriormente, no siempre se pueden alcanzar si se usan particiones cocidas, hay que considerar el número de accesos que el sistema operativo tiene que realizar para llegar a la página. En las particiones crudas se debe considerar el tamaño de sector. Sin embargo, es una buena manera de medir el desempeño del modelo de almacenamiento.

Las direcciones de página requieren 8 bytes dentro del espacio, debido a que es el máximo direccionable por un lenguaje de programación. Las direc-

ciones del espacio requieren de 4 bytes, debido a que es el máximo tamaño que soporta el lenguaje de programación para las tablas de chunks en memoria. Así que, para encontrar una página dentro de un segmento, se requieren de 12 bytes. Este tamaño de dirección, prácticamente no impone un límite para el tamaño de los espacios en los sistemas actualmente existentes.

4.3. Almacenamiento de una Base de Datos Semiestructurados

Para almacenar los componentes principales de una base de datos semiestructurados, se utilizan cinco segmentos del almacenamiento avanzado, estos segmentos son:

- *ID*. Para almacenar los descriptores de los datos.
- *CONTENT*. Donde se almacenan los contenidos de los datos semiestructurados que componen la base de datos.
- *LOB*. Donde se almacenan los contenidos de los objetos largos (LOBs).
- *PARENTS*. Donde se almacenen las colecciones de padres de los datos semiestructurados.
- *DICTIONARY*. Donde se almacena el diccionario de datos, que contiene los nombres de las *ssd-tablas* y sus raices.

4.3.1. El segmento ID

En el segmento ID se almacenan los descriptores de los datos semiestructurados que conforman la base de datos. Los datos pueden ser no primitivos o primitivos, los datos primitivos pueden ser simples u objetos largos (LOBs). Intuitivamente se puede decir que los datos simples se pueden manipular en memoria y los objetos largos son demasiado grandes para ser manipulados en memoria. Los datos no primitivos son conjuntos o colecciones de otros datos semiestructurados etiquetados.

Cada dato semiestructurado posee un identificador único, para encontrar rápidamente un dato semiestructurado la mejor opción es que su identificador corresponda con su posición física, es decir, donde se encuentra su descriptor. En el segmento ID, se almacenan los descriptores de los datos semiestructurados, cada unidad representa un descriptor, el número de espacio junto con el número de unidad dentro del espacio conforman el identificador para el dato descrito.

Los descriptores tienen que almacenar el tipo de dato, el número de *ssd-tablas* de la cuales el dato es raíz (esto es el número de referencias que el diccionario hace hacia él), el apuntador a su colección de padres y el apuntador a su contenido.

Se espera que casi todos los datos semiestructurados tengan solamente un padre, así que, en el descriptor se puede reservar un espacio para guardar el identificador de un padre, esto con el fin de ahorrar espacio en el segmento PARENTS con los respectivos ahorros de acceso a disco que esto trae. También se espera que un poco más de la mitad de los datos semiestructurados sean de tipo primitivos, así que de igual manera se puede reservar en el descriptor un espacio para almacenar el contenido de un dato primitivo con el fin de ahorrar espacio en el segmento CONTENT y los respectivos ahorros de accesos a disco que esto trae.

Al tener tres tipos de datos semiestructurados se deben tener tres tipos de descriptores, un descriptor de dato ocupa un total de 96 bytes, así que, el tamaño de unidad en el segmento ID es de 96 bytes. A continuación se explica el contenido de estos descriptores muchas de las partes almacenadas se explicarán con más detalle en las secciones siguientes.

Para los tres tipos de descriptor se debe almacenar:

- El número de referencias del diccionario. Ocupa 8 bytes, que corresponden al máximo establecido para el número de *ssd-tablas*.
- El identificador del primer padre, se usa un identificador nulo en caso de que el dato no tenga padre. Ocupa 12 bytes.
- Una dirección que apunta a una unidad en el segmento PARENTS y que corresponde a la raíz del árbol usado para la colección de padres. Ocupa 12 bytes, que es el tamaño requerido para direccionar unidades.

- Una dirección que apunta a una unidad en el segmento PARENTS y que corresponde a la cabeza de la lista usada para la colección de padres. Ocupa 12 bytes.
- El tipo de dato, dado por alguna de las constantes: `NON_PRIMITIVE`, `INTEGER`, `REAL`, `STRING`, `BYTE_SEQUENCE`, `BLOB` o `CLOB`. Ocupa 4 bytes.

Para los descriptores de datos primitivos simples se requiere además almacenar:

- Una dirección a una unidad en el segmento CONTENT, que representa una extensión en caso de que el contenido del dato no quepa en el descriptor. Ocupa 12 bytes.
- El tamaño del dato. Ocupa 4 bytes, que corresponden al máximo dado por el lenguaje de programación para almacenar un dato en memoria.
- Los primeros 32 bytes del contenido del dato.

Para los descriptores de datos primitivos de tipo LOB se requiere además almacenar:

- Una dirección a una extensión en el segmento LOB, donde se guarda el resto del contenido del dato que no se guarda en el descriptor. Ocupa 12 bytes.
- El tamaño del dato. Ocupa 8 bytes, que corresponde al máximo dado por el lenguaje de programación para direccionar un archivo de acceso aleatorio.
- Los primeros 28 bytes del contenido del dato.

Para los descriptores de datos no primitivos se requiere además almacenar:

- Una dirección a una unidad en el segmento CONTENT, que corresponde a la raíz del árbol ordenado por identificador, que se usa para la colección de datos semiestructurados etiquetados que representa el contenido. Ocupa 12 bytes.

- Una dirección a una unidad en el segmento CONTENT, que corresponde a la cabeza de la lista ordenada por identificador, que se usa para la colección de datos semiestructurados etiquetados que representa el contenido. Ocupa 12 bytes.

- Una dirección a una unidad en el segmento CONTENT, que corresponde a la raíz del árbol ordenado por etiqueta, que se usa para la colección de datos semiestructurados etiquetados que representa el contenido. Ocupa 12 bytes.

- Una dirección a una unidad en el segmento CONTENT, que corresponde a la cabeza de la lista ordenada por etiqueta, que se usa para la colección de datos semiestructurados etiquetados que representa el contenido. Ocupa 12 bytes.

4.3.2. El segmento CONTENT

El segmento CONTENT se encarga de almacenar el contenido de los datos semiestructurados, con excepción de los de tipo LOB. Para los datos primitivos simples, se almacenan cadenas de extensiones que forman el contenido. Para los datos no primitivos, se tiene que almacenar la colección de datos semiestructurados etiquetados que forman su contenido.

La implementación de las colecciones de datos semiestructurados etiquetados tiene varias cuestiones a considerar. Se espera que en una base de datos semiestructurados poco menos de la mitad de los datos sean de tipo no primitivo y casi todos ellos con un número pequeño de subdatos. Sin embargo, se espera que unos pocos datos (las raices de las *ssd-tablas*) tengan un número muy grande de subdatos y sean usados frecuentemente. Algunas operaciones que provee el manejador afectan directamente el contenido de los datos (el borrado y la inserción), otras operaciones requieren localizar los subdatos de la manera más eficiente posible (las consultas, comparaciones de parentesco, etc.). Las localizaciones se pueden requerir por medio del identificador o de la etiqueta de los subdatos. Dentro de un dato es posible encontrar varios subdatos con una misma etiqueta o el mismo dato con distintas etiquetas. Es necesario utilizar una estructura de datos que soporte estas características y pueda lograr un desempeño aceptable.

La solución presentada en este trabajo, es el uso de una estructura de datos cuádruple para implementar una colección de datos semiestructurados etiquetados. Esta estructura está formada por dos árboles-AA y dos listas doblemente ligadas, ésto para ordenar los subdatos por identificador y por etiquetas. Los árboles sirven para localizar rápidamente un dato y las listas para el manejo de repeticiones y la recuperación secuencial del contenido. Tanto las listas como los árboles están ordenados uno por identificador y otro por etiqueta.

Los *árboles-AA* [11] son una variante de los árboles rojo-negro. Los árboles-AA, rojo-negros y AVL son árboles binarios balanceados, es decir que su altura es de $O(\log n)$. Los árboles AVL tiene una altura de $\log_2 n$, los rojo-negros y AA una altura de $2\log_2 n$. La desventaja que presentan los árboles AVL y rojo-negros es la complejidad de implementar sus operaciones, sobre todo la del borrado. Los árboles-AA proveen una implementación más sencilla y son ideales para tener árboles balanceados que requieren de borrado, así que, son ideales para su uso en operaciones sobre datos semiestructurados. Como los árboles-AA son binarios permiten que en sus nodos se puedan combinar otras estructuras de datos.

Los árboles B y B^+, son las estructuras de datos más comúnmente utilizadas en los sistemas de bases de datos relacionales para ordenar los registros de una tabla. Éstos resultan ser muy eficientes para organizar grandes cantidades de datos. Sin embargo, para el uso en datos semiestructurados no son tan convenientes como los árboles-AA, debido a que al no ser binarios no se pueden combinar otras estructuras. Además, los nodos de los arboles B y B^+ generalmente ocupan una página completa y se estima un desperdicio de la mitad del espacio que ocupan. Como se espera que la mayoría de los datos no primitivos tengan pocos subdatos, esto generaría un gran desperdicio de espacio.

En un mismo nodo de la colección de datos semiestructurados etiquetados, se puede incluir la información necesaria para mantener las cuatro estructuras de datos. Para mantener las cuatro estructuras de datos, para cada nodo se tienen que mantener los apuntadores derecho e izquierdo, el apuntador al padre y la altura del nodo para cada árbol, además de los apuntadores anterior y siguiente para cada lista. Debido a que se requiere mantener las cuatro estructuras de datos, es necesario que cada nodo posea el apuntador al padre (en cada árbol). Esto debido a que después de un borrado se requiere

mantener la consistencia de la colección y el apuntador al padre ayuda a este fin, aunque el algoritmo se complica un poco.

Cada nodo de la estructura cuádruple se encuentra en una unidad del segmento CONTENT. Las estructuras cuádruples poseen cuatro nodos distinguidos, a través de los cuales se tiene acceso a ella, estos son dos raíces y dos cabezas de lista. Las direcciones a estos nodos se tienen que almacenar en el descriptor (en el segmento ID) del dato no primitivo que posee la colección.

En el segmento CONTENT, se almacenan los nodos de las estructuras cuádruples que conforman las colecciones de datos semiestructurados etiquetados y las extensiones de contenido. Las extensiones de contenido se utilizan en caso de que los datos primitivos simples no quepan totalmente en el descriptor o que las etiquetas no quepan totalmente en los nodos de la estructura cuádruple. Tanto las extensiones como los nodos se colocan en unidades del segmento CONTENT. Estas unidades tienen un tamaño de 200 bytes.

Un nodo de la estructura cuádruple contiene lo siguiente:

- La dirección de una unidad en el mismo segmento, que corresponde al nodo anterior en la lista doblemente ligada ordenada por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo siguiente en la lista doblemente ligada ordenada por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo izquierdo en el árbol-AA ordenado por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo derecho en el árbol-AA ordenado por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo padre en el árbol-AA ordenado por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La altura del nodo en el árbol-AA ordenado por identificador. Ocupa 1 byte.

- La dirección de una unidad en el mismo segmento, que corresponde al nodo anterior en la lista doblemente ligada ordenada por etiqueta. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo siguiente en la lista doblemente ligada ordenada por etiqueta. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo izquierdo en el árbol-AA ordenado por etiqueta. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo derecho en el árbol-AA ordenado por etiqueta. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo padre en el árbol-AA ordenado por etiqueta. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La altura del nodo en el árbol-AA ordenado por etiqueta. Ocupa 1 byte, como la altura del árbol es logarítmica y las unidades se direccionan con 12 bytes no se requiere más.
- El identificador del dato semiestructurado etiquetado al que corresponde el nodo en la colección. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde a la extensión de la etiqueta en caso de que se requiera o una dirección nula si no se requiere. Ocupa 12 bytes.
- El tamaño de la etiqueta. Ocupa 4 bytes, que corresponde al máximo que da el lenguaje de programación para manipular la etiqueta en memoria.
- El contenido de la etiqueta correspondiente al dato semiestructurado etiquetado al que corresponde el nodo o al menos sus primeros 50 bytes.

Las extensiones de contenido requieren almacenar lo siguiente:

- La dirección de una unidad en el mismo segmento, que corresponde a otra extensión por si ésta no es suficiente o una dirección nula si no se requiere. Ocupa 12 bytes.
- El contenido de la extensión con un máximo de 188 bytes.

4.3.3. El segmento LOB

Los objetos largos (LOBs) intuitivamente son aquellos que por su gran tamaño no se pueden manipular en memoria. Hay dos tipos de objetos largos: BLOBs (que están orientados a bytes) y CLOBs (que están orientados a caracteres). Los objetos largos son tratados de manera similar a los archivos de acceso aleatorio, con mecanismos para leer o escribir bytes en una posición específica. Ejemplos de LOBs son imágenes, videos, música, textos largos, etc.

El segmento LOB, se encarga de almacenar el contenido de los objetos largos. Para aprovechar al máximo el espacio, el tamaño de las unidades en este segmento es igual al tamaño de página usado por el manejador. Si P es el tamaño de página se toma $D = [P/12]$, D es el número de direcciones de unidades que caben en una página (que es del mismo tamaño que una unidad en el segmento LOB).

Para almacenar el contenido de un LOB se utilizan 5 tipos de unidades (ver fig. 4.8):

- Unidad de datos. Almacena parte del contenido de los objetos largos, a lo más P bytes.
- Unidad indirecta. Contiene D direcciones de unidades de datos.
- Unidad doblemente indirecta. Contiene D direcciones de unidades indirectas.
- Unidad triplemente indirecta. Contiene D direcciones de unidades doblemente indirectas.

- Unidad de índice. Consta de $D-4$ direcciones de unidades de datos, una dirección a una unidad indirecta, una dirección a una unidad doblemente indirecta, una dirección a una unidad triplemente indirecta y una dirección a otra unidad de índice.

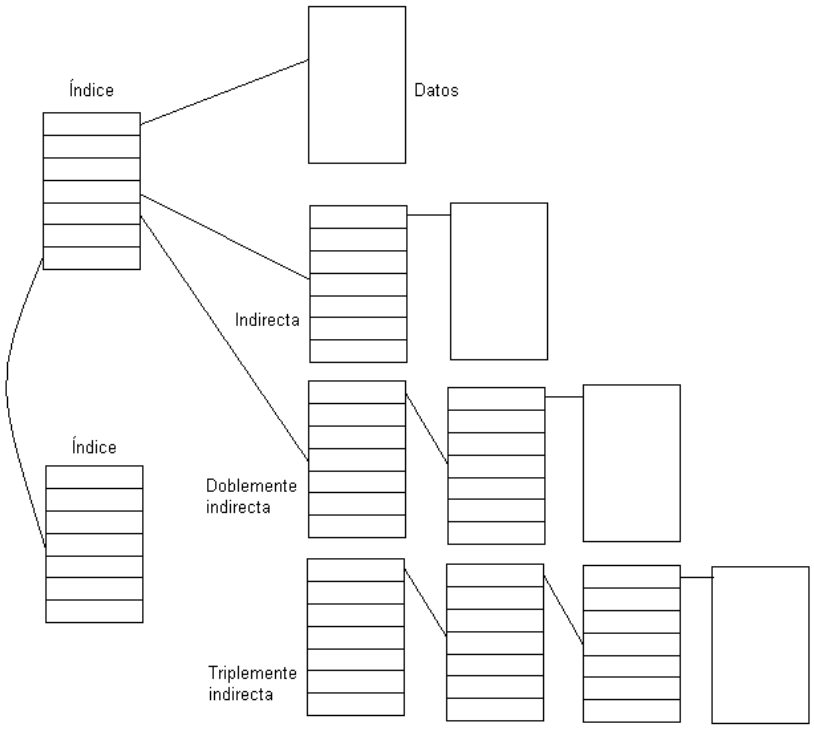

Figura 4.8: Organización para el almacenamiento de LOBs

Con una sola unidad de índice, se puede direccionar un contenido de tamaño $(D-4)P + DP + D^2P + D^3P$. En el cuadro 4.2 se muestran algunos valores para distintos tamaños de página. Si el espacio no fuese suficiente se pueden crear cadenas de unidades de índice. El descriptor del LOB en el segmento ID apunta a la primera unidad de índice para el contenido del LOB. Las direcciones que no se requieran dentro de los distintos tipos de unidades contendrán valores nulos.

Tamaño de página	Tamaño direccionable
128 B	184 KB
256 B	2 MB
512 B	40 MB
1 KB	607 MB
2 KB	10 GB
4 KB	152 GB
8 KB	2 TB
16 KB	38 TB
32 KB	607 TB
64 KB	9 PB
128 KB	152 PB
256 KB	2 EB
512 KB	38 EB
1 MB	607 EB
2 MB	9 ZB
4 MB	152 ZB

Cuadro 4.2: Tamaños de páginas y espacio de LOB direccionable por unidad de índice

4.3.4. El segmento PARENTS

En el segmento PARENTS, se almacenan las colecciones de padres que poseen los datos semiestructurados. Como se mencionó anteriormente se espera que la mayoría de los datos semiestructurados tengan un solo padre o una pequeña cantidad de ellos, aunque puede haber casos en lo que ésto no sea así. Es importante saber cuáles son los padres de un dato semiestructurado, para lograr mayor eficiencia y mantener la consistencia de algunas de las operaciones definidas en el capítulo 3.

Para implementar las colecciones de padres también se utiliza el árbol-AA y la lista doblemente ligada. Con ello se evita el desperdicio excesivo de espacio que se podría presentar con el uso de otras estructuras. Para los padres sólo se requiere una estructura doble, ordenada por identificador debido a que solo se tienen que conocer los identificadores de los padres.

Las unidades del segmento PARENTS son una combinación de un nodo de un árbol-AA ordenado por identificador y un nodo de una lista doblemente ligada ordenada con el mismo criterio. La raíz del árbol-AA y la cabeza de la lista correspondientes a la colección de padres se almacenan en el descriptor del dato en el segmento ID.

Las unidades del segmento PARENTS tienen un tamaño de 73 bytes y contienen:

- La dirección de una unidad en el mismo segmento, que corresponde al nodo anterior en la lista doblemente ligada ordenada por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo siguiente en la lista doblemente ligada ordenada por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo izquierdo en el árbol-AA ordenado por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo derecho en el árbol-AA ordenado por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo padre en el árbol-AA ordenado por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La altura del nodo en el árbol-AA ordenado por identificador. Ocupa 1 byte.
- El identificador del dato semiestructurado al que corresponde el nodo en la colección. Ocupa 12 bytes.

4.3.5. El segmento DICTIONARY

El diccionario de datos básicamente almacena los nombres de las *ssd-tablas* y los respectivos identificadores de los datos semiestructurados que

corresponden a sus raíces. El diccionario de datos se puede ver como una colección de datos semiestructurados etiquetados, la etiqueta corresponde al nombre de una *ssd-tabla* y no se puede repetir.

El segmento DICTIONARY almacena un descriptor de la colección de datos etiquetados, los nodos que conforman esta colección y las extensiones para los nombres de las *ssd-tablas*. Al igual que ocurre con el contenido de los datos no primitivos, se utiliza una estructura cuádruple formada por dos árboles-AA y dos listas doblemente ligadas para ordenar la colección por identificador y por nombre de las *ssd-tablas*.

Las unidades en el segmento DICTIONARY tienen un tamaño de 200 bytes, al igual que las del segmento CONTENT. La diferencia con el segmento CONTENT es, que en el segmento CONTENT se guardan varias colecciones y en el segmento DICTIONARY sólo se guarda una.

El descriptor ocupa la primera unidad del segmento DICTIONARY y contiene lo siguiente:

- La dirección a una unidad en el mismo segmento, que correspondea la raíz del árbol ordenado por identificador, que se usa para la colección que forma el diccionario. Ocupa 12 bytes.
- La dirección a una unidad en el mismo segmento, que correspondea la cabeza de la lista ordenada por identificador, que se usa para la colección que forma el diccionario. Ocupa 12 bytes.
- La dirección a una unidad en el mimo segmento, que correspondea la raíz del árbol ordenado por nombre, que se usa para la colección que forma el diccionario. Ocupa 12 bytes.
- La dirección a una unidad en el mismo segmento, que correspondea la cabeza de la lista ordenada por nombre, que se usa para la colección que forma el diccionario. Ocupa 12 bytes.

Los nodos de la estructura cuádruple que forma el diccionario contienen lo siguiente:

- La dirección de una unidad en el mismo segmento, que corresponde al

nodo anterior en la lista doblemente ligada ordenada por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.

- La dirección de una unidad en el mismo segmento, que corresponde al nodo siguiente en la lista doblemente ligada ordenada por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo izquierdo en el árbol-AA ordenado por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo derecho en el árbol-AA ordenado por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo padre en el árbol-AA ordenado por identificador. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La altura del nodo en el árbol-AA ordenado por identificador. Ocupa 1 byte.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo anterior en la lista doblemente ligada ordenada por nombre. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo siguiente en la lista doblemente ligada ordenada por nombre. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo izquierdo en el árbol-AA ordenado por nombre. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo derecho en el árbol-AA ordenado por nombre. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde al nodo padre en el árbol-AA ordenado por nombre. Se usa un valor nulo si no se requiere. Ocupa 12 bytes.

- La altura del nodo en el árbol-AA ordenado por nombre. Ocupa 1 byte.
- El identificador del dato semiestructurado correspondiente a la raíz de la *ssd-tabla* dada por el nodo. Ocupa 12 bytes.
- La dirección de una unidad en el mismo segmento, que corresponde a la extensión del nombre en caso de que se requiera o una dirección nula si no se requiere. Ocupa 12 bytes.
- El tamaño del nombre. Ocupa 4 bytes.
- El contenido del nombre correspondiente a la *ssd-tabla* descrita en el nodo o al menos sus primeros 50 bytes.

Las extensiones de nombre requieren almacenar lo siguiente:

- La dirección de una unidad en el mismo segmento, que corresponde a otra extensión por si ésta no es suficiente o una dirección nula si no se requiere. Ocupa 12 bytes.
- El contenido de la extensión con un máximo de 188 bytes.

Se ha visto cómo las bases de datos se distribuyen en cinco componentes principales, para ser almacenadas en segmentos del modelo de almacenamiento avanzado (ver fig. 4.9). Con ello se aprovechan las capacidades descritas anteriormente para los niveles de almacenamiento avanzado y primitivo.

Resumen

En este capítulo se presentaron los principales aspectos del diseño del manejo de almacenamiento para las bases de datos semiestructurados. El manejo de almacenamiento se construye en tres niveles: Almacenamiento Primitivo, Almacenamieto Avanzado y Almacenamiento de Bases de Datos.

En el almacenamiento primitivo, se define el tamaño de página que es la unidad mínima de lectura-escritura que utilizará el manejador. Se incorpora el manejo de memoria intermedia, se resuelve el problema de la agrupación de múltiples dispositivos y el uso de particiones crudas y cocidas.

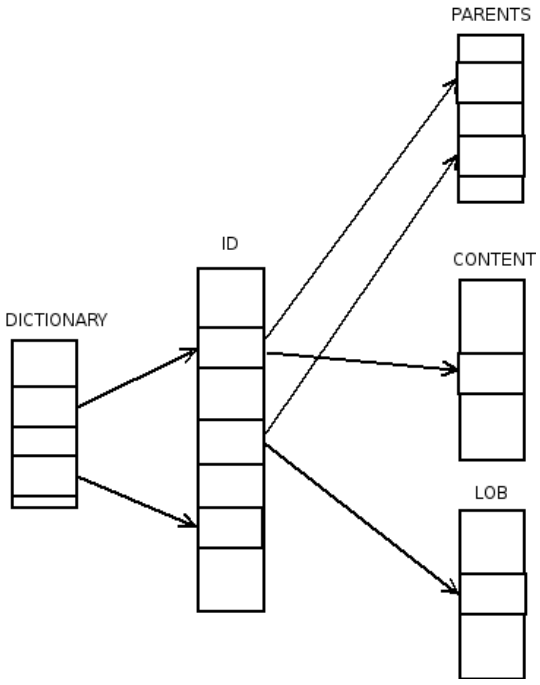

Figura 4.9: Relaciones de los segmentos que conforman una base de datos semiestructurados

En el almacenamiento avanzado, se provee un modelo de almacenamiento en tres niveles de organización: Segmento, Espacio y Unidad. En el almacenamiento avanzado es donde se controla el espacio libre.

En el almacenamiento de bases de datos, los datos semiestructurados se transforman en estructuras especializadas como listas, árboles-AA, tablas de índices, etc. Para implementar su funcionalidad y poder ser almacenadas en el almacenamiento avanzado.

El diseño del manejo de almacenamiento fue un proceso complicado, debido a la gran cantidad de detalles a considerar y a la gran interdependencia que guarda con los demás componentes del núcleo. La implementación fue un proceso delicado, pero se logró concluir exitosamente (ver apéndice A).

En el siguiente capítulo, se aborda el problema de permitir que la base de datos sea utilizada por varios usuarios o procesos al mismo tiempo. Esto

también, involucra el problema de que no se presenten inconsistencias a las estructuras tratadas en este capítulo.

Capítulo 5

Concurrencia y Recuperación

Los sistemas manejadores de bases de datos, tienen la capacidad de permitir a varios usuarios acceder simultáneamente a las bases de datos, sin que ocurran conflictos entre ellos y sin causar pérdída o corrupción de la información. También tienen un mecanismo que permite al sistema recuperarse en caso de alguna falla, con lo cual se asegura la consistencia de la información contenida en las bases de datos.

En este capítulo, se aborda el problema de diseñar los componentes que permitan la incorporación de las capacidades mencionadas anteriormente. Estos componentes son el control de concurrencia y el sistema de recuperación.

5.1. Transacciones

Una *transacción* es una serie de operaciones a una base de datos que se agrupan en una sola unidad lógica de ejecución [1]. Un sistema manejador de bases de datos debe asegurar que la ejecución de transacciones se realice adecuadamente, es decir, que se ejecute completamente o en caso de falla no se ejecute en absoluto. Además, debe de asegurar que en el caso de transacciones concurrentes no se distorsionen unas a otras.

Una transacción está definida por dos declaraciones: *inicio de Transacción* y *fin de Transacción*. La transacción, consta de todas las operaciones a la base de datos que se ejecutan entre estas dos declaraciones.

Las transacciones constan de cuatro propiedades llamadas ACID (por sus siglas en inglés):

- *Atomicidad*: Todas las operaciones de la transacción se realizan adecuadamente en la base de datos o ninguna de ellas se ejecuta.
- *Consistencia*: La ejecución de una transacción conserva la consistencia de la información en la base de datos.
- *Aislamiento*: Aunque se ejecuten transacciones concurrentemente, el sistema garantiza que los resultados son como si se ejecutaran de manera secuencial, es decir, una después de otra.
- *Durabilidad*: Tras la finalización de una transacción, los cambios realizados a la base de datos permanecen.

Una transacción que no termina con éxito su ejecución (a causa de un error o por decisión del usuario), se denomina *transacción abortada*. Para asegurar la propiedad de atomicidad, una transacción abortada no debe tener efecto en la base de datos, así que, cualquier cambio que la transacción abortada haya hecho debe deshacerse. Una vez deshechos los cambios, se dice que la transacción se ha retrocedido. Una transacción que termina con éxito, se dice que es una *transacción comprometida*. Una transacción comprometida ha hecho cambios en la base de datos dejándola en un estado consistente.

Las transacciones pueden estar en uno de los siguientes estados (ver fig. 5.1):

- *Activa*: La transacción permanece en este estado durante su ejecución.
- *Parcialmente Comprometida*: Después de ejecutarse la última instrucción.
- *Fallida*: Tras descubrir que no se puede continuar la ejecución normal de la transacción.

- *Abortada*: Después de haber retrocedido la transacción y reestablecido la base de datos a su estado anterior.
- *Comprometida*: Tras completarse con éxito la transacción.

Se puede considerar que el acceso a la base de datos se lleva a cabo mediante dos operaciones básicas:

- Leer(X).
- Escribir(X).

X es un dato, objeto o elemento de la base de datos (hablando de bases de bases de datos relacionales X puede ser un registro, una tabla, el diccionario de datos, toda la base de datos, etc). X puede representar un elemento a diversos niveles dentro de la base de datos. Las operaciones anteriores se traducen a operaciones de entrada-salida, en los dispositivos de almacenamiento secundario o en la memoria intermedia.

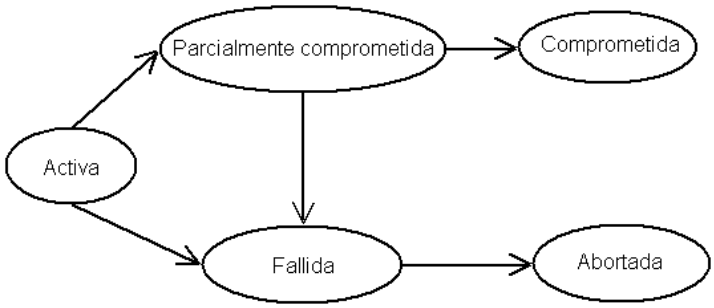

Figura 5.1: Estados de una transacción

5.1.1. Concurrencia

La *concurrencia* es el mecanismo que permite ejecutar varias transacciones al mismo tiempo en las bases de datos. El sistema manejador tiene que asegurar que al ejecutarse transacciones concurrentes, se tiene que mantener la propiedad de aislamiento, es decir, que el resultado es como si las transacciones se ejecutaran de manera secuencial.

Aunque las transacciones se pueden ejecutar de manera concurrente, las operaciones básicas a la base de datos se ejecutan de manera secuencial. A la secuencia de ejecución de estas operaciones básicas se le llama *plan*. Un plan representa el orden cronológico en el cual se ejecutan las operaciones básicas a la base de datos [1] (ver fig. 5.2).

T_1	T_2
$leer(A)$	
$A = A - 10$	
$escribir(A)$	
	$leer(A)$
	$A = A * 0{,}9$
	$escribir(A)$
$leer(B)$	
$B = B + 10$	
$escribir(B)$	

Figura 5.2: Ejemplo de plan

Un plan es *secuencial* cuando todas las operaciones de cada transacción aparecen juntas en el plan, sin traslaparse con las de otra transacción (ver fig. 5.3).

T_1	T_2
$leer(A)$	
$A = A - 10$	
$escribir(A)$	
$leer(B)$	
$B = B + 10$	
$escribir(B)$	
	$leer(A)$
	$A = A * 0{,}9$
	$escribir(A)$

Figura 5.3: Ejemplo de plan secuencial

Cuando se ejecutan transacciones concurrentes, el plan no tiene por qué ser secuencial. De hecho, la distribución de las operaciones puede ser muy va-

riada y ciertas distribuciones pueden causar que la base de datos quede en un estado inconsistente. El sistema manejador de bases de datos se debe encargar de que los planes resultantes dejen la base de datos en un estado consistente.

5.1.2. Secuencialidad

Un plan es *secuenciable* si el resultado que produce a la base de datos es equivalente al de un plan secuencial. Por ejemplo, el plan de la figura 5.2 es equivalente al de la figura 5.3. Los planes secuenciales mantienen la propiedad de aislamiento. Así que, el manejador de bases de datos se puede asegurar de que los planes que se ejecuten sean secuenciables.

La técnica más usada por los manejadores de bases de datos para garantizar la secuencialidad de los planes es la *secuencialidad en cuanto a conflictos*, la cual se describe a continuación [1].

Sea P un plan con dos instrucciones consecutivas I_i e I_j, pertenecientes a las transacciones T_i y T_j respectivamente, con $i \neq j$. Si I_i e I_j operan sobre el mismo elemento Q, se dice que I_i e I_j tienen conflicto si ocurre alguna de las siguientes:

- $I_i = leer(Q)$ e $I_j = escribir(Q)$.
- $I_i = escribir(Q)$ e $I_j = leer(Q)$.
- $I_i = escribir(Q)$ e $I_j = escribir(Q)$.

Sea P un plan con dos instrucciones consecutivas I_i e I_j, pertenecientes a las transacciones T_i y T_j respectivamente, con $i \neq j$. Si I_i e I_j no tienen conflicto estas se pueden intercambiar de orden, generando así un plan P' que es equivalente a P en cuanto a conflictos.

Un plan P es equivalente a P' en cuanto a conflictos, si P' se puede obtener desde P mediante una serie de intercambios de operaciones consecutivas que no tengan conflicto.

Un plan P es secuenciable en cuanto a conflictos, si es equivalente en cuanto a conflictos a un plan secuencial, es decir, si P se puede transformar

en un plan secuencial por medio de intercambios de operaciones consecutivas no conflictivas. Por ejemplo, el plan en la figura 5.4 es secuenciable en cuanto a conflictos ya se puede transformar en un plan secuencial.

T_1	T_2
$leer(A)$	
$escribir(A)$	
	$leer(A)$
$leer(B)$	
	$escribir(A)$
$escribir(B)$	
	$leer(B)$
	$escribir(B)$

T_1	T_2
$leer(A)$	
$escribir(A)$	
$leer(B)$	
$escribir(B)$	
	$leer(A)$
	$escribir(A)$
	$leer(B)$
	$escribir(B)$

Figura 5.4: Ejemplo de plan secuenciable en cuanto a conflictos

El intercambio de operaciones no conflictivas, garantiza que el resultado de el plan generado mantiene exactamente los mismos cambios a la base de datos que el original. De esta manera, se garantiza que un plan secuenciable en cuanto a conflictos es secuenciable. Mediante técnicas que se explicarán más adelante, el sistema manejador de bases de datos garantiza que los planes resultantes son secuenciables en cuanto a conflictos, con lo que se asegura la propiedad de aislamiento.

5.1.3. Recuperabilidad

Si una transacción falla (ya sea por un error de software, un error de hardware, a voluntad del usuario o por la razón que sea) es necesario deshacer el efecto de dicha transacción, para asegurar la propiedad de atomicidad. Cuando se permite la concurrencia se puede dar el caso de que la transacción T_j dependa de T_i y T_i falle, en este caso T_j se tiene que deshacer también. Una *transacción no recuperable* es aquella que no se puede deshacer porque otra que a ha sido completada depende de ella.

Ejemplo 5.1.1 Considérese la figura 5.5. Supóngase que la transacción T_2 se compromete inmediatamente después la instrucción $leer(A)$ y la transacción

T_1 se aborta justo después de $leer(B)$, en este caso no se puede deshacer la escritura de A hecha por T_1 ya que T_2 ya leyó A. Aquí T_1 es una transacción no recuperable.

T_1	T_2
$leer(A)$	
$escribir(A)$	
	$leer(A)$
$leer(B)$	

Figura 5.5: Ejemplo de transacción no recuperable

Un *plan recuperable*, es aquel en el que para todo par de transacciones T_i y T_j, tales que T_j lee datos escritos previamente por T_i, la operación de comprometer de T_i aparece antes que la de T_j.

Aunque un plan sea recuperable, puede darse el caso de que se tengan que retroceder varias transacciones para recuperar correctamente el estado previo a un fallo. A este fenómeno se le llama retroceso en cascada.

Ejemplo 5.1.2 Considérese la figura 5.6. Si la transacción T_1 falla, entonces también se debe deshacer la transacción T_2, ya que lee un dato que escribió T_1. Por razones similares, también se debe deshacer T_3. Aquí se aprecia un retroceso en cascada

T_1	T_2	T_3
$leer(A)$		
$leer(B)$		
$escribir(A)$		
	$leer(A)$	
	$escribir(A)$	
		$leer(A)$

Figura 5.6: Ejemplo de dependencia que puede provocar un retroceso en cascada

Tanto los planes no recuperables como los retrocesos en cascada son no deseables para un sistema manejador de bases de datos, ya que provocan problemas de consistencia y de rendimiento.

5.2. El control de concurrencia

En los manejadores de bases de datos, existe una gran variedad de técnicas para el control de concurrencia: protocolos de bloqueo, protocolos de marcas temporales, protocolos de validación, etc. La técnica más comúnmente usada es el uso de protocolos de bloqueo.

Una forma de asegurar la secuencialidad es exigir que el acceso a los datos se haga con exclusión mutua, es decir, mientras una transacción modifica un elemento ninguna otra puede acceder a él. Para lograr dicho fin se utilizan los *bloqueos*, una transacción puede acceder a un elemento de datos solamente si posee un bloqueo sobre dicho elemento. Se tienen dos modos de bloqueo:

- *Compartido*. Si una transacción obtiene un bloqueo en modo compartido para un elemento, entonces puede leer pero no escribir en él.
- *Exclusivo*: Si una transacción obtiene un bloqueo en modo exclusivo para un elemento, entonces puede tanto leer como escribir en él.

Cuando una transacción quiere acceder a un elemento, es necesario que solicite un bloqueo sobre ese elemento, el modo de bloqueo depende de la operación que se necesite realizar. El manejador de concurrencia concede el bloqueo o deja a la transacción en espera hasta que se le pueda otorgar dicho bloqueo. Para otorgar los bloqueos el manejador de concurrencia utiliza una función de compatibilidad que es como sigue:

- Si T_1 solicita un bloqueo en modo exclusivo sobre el elemento D, entonces el bloqueo se otorga solamente si D no está bloqueado por otra transacción.
- Si T_1 solicita un bloqueo en modo compartido sobre D, entonces el bloqueo se otorga sólo si D no está bloqueado en modo exclusivo por otra transacción.

El simple hecho de bloquear los elementos y liberar el bloqueo después del acceso no garantiza la secuencialidad, para garantizarla se utiliza el *protocolo de bloqueo de dos fases* [1]. Dicho protocolo exige que cada transacción tenga las siguientes fases:

- *Fase de Crecimiento*: Donde una transacción puede obtener bloqueos, pero no liberarlos.
- *Fase de Decrecimiento*: Donde una transacción puede liberar bloqueos, pero no obtener uno nuevo.

Inicialmente una transacción entra en la fase de crecimiento, donde adquiere los bloqueos. Una vez que libera un bloqueo, entra en la fase de decrecimiento donde no puede pedir más bloqueos.

El protocolo de bloqueo de dos fases asegura que los planes resultantes sean secuenciables y evita los planes no recuperables, pero se pueden presentar retrocesos en cascada. Para evitar los retrocesos en cascada, se usa el protocolo de bloqueo estricto de dos fases. En éste último, se exige que además de que el protocolo sea de dos fases, una transacción mantenga todos sus bloqueos exclusivos hasta que se complete.

Debido a que en la mayoría de los casos la ejecución de una transacción no se puede conocer a priori, no se puede saber cuando ya no se va a ocupar un bloqueo en modo exclusivo. Por esta razón se utiliza otra variante llamada el *protocolo de bloqueo rigurosos de dos fases*, aquí se exige que todos los bloqueos se mantengan hasta que la transacción se comprometa, es decir, la fase de decrecimiento entra al momento de comprometer la transacción.

La mayoría de los manejadores de bases de datos usan el protocolo de bloqueo riguroso de dos fases. Sin embargo, los protocolos de bloqueo presentan otro problema que es la existencia de *bloqueos mutuos*. Esto ocurre cuando dos o más transacciones están esperándose entre si. Por ejemplo: Cuando T_1 tiene un bloqueo en A y T_2 un bloqueo en B, pero T_1 está esperando que se libere B y T_2 que se libere A.

La solución que se usa en este trabajo para tratar los bloqueos mutuos, es la capacidad de poder definir un tiempo máximo de espera. Si el tiempo se supera entonces la transacción se debe retroceder.

5.2.1. El protocolo de bloqueo dual para datos semiestructurados

El control de concurrencia para este trabajo está implementado al nivel semántico de datos semiestructurados. Cuando se accede a un dato semiestructurado se pueden realizar distintas formas de lectura o escritura. Se puede modificar el contenido del dato o su colección de padres.

Ejemplo 5.2.1 Considerese la figura 5.7. Cuando se agrega el dato B al contenido del dato A, se modifica el contenido de A y la colección de padres de B, pero el contenido de B y la colección de padres de A permanecen intactos. Si se colocaran bloqueos exclusivos en A y B, no se permitiría que otra transacción modificara el contenido de B o la lista de padres de A.

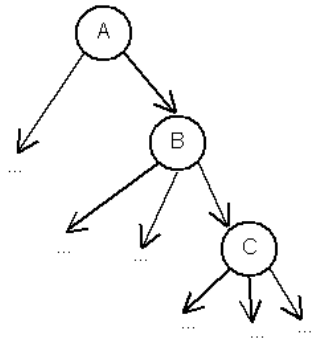

Figura 5.7: Dato semiestructurado para ejemplificar el uso del protocolo de bloqueo dual para datos semiestructurados

Para elevar el nivel de concurrencia, se considera que un dato semiestructurado esta formado por dos entidades distintas en lugar de una sola. Un dato semiestructurado está compuesto de *vista* y de *contenido*, los bloqueos se pueden aplicar a cualquiera de ellas sin que afecte a la otra.

Intuitivamente la vista indica la presencia de un dato semiestructurado, técnicamente la vista está representada por su colección de padres y las referencias del diccionario hacia el dato. La vista y el contenido de un dato semiestructurado son independientes entre sí (aunque pertenezcan al mismo dato, en el almacenamiento se puede delimitar una separación exacta entre

ellos), por esta razón un protocolo de bloqueo riguroso de dos fases aplicado garantiza la secuencialidad en cuanto a conflictos, evita las transacciones no recuperables y los retrocesos en cascada. En el ejemplo anterior, se podrían aplicar bloqueos exclusivos al contenido de A y a la vista de B, pero para la vista de A y el contenido de B no se solicitan bloqueos, lo que permite que otras transacciones puedan usarlos sin que afecte el resultado de la operación.

Ejemplo 5.2.2 Considérense la figura 5.7 y las transacciones mostradas en la figura 5.8. T_1 agrega B al contenido de A y la transacción T_2 elimina a C del contenido de B, al mismo tiempo. Estas dos transacciones se pueden ejecutar de manera concurrente y son secuenciables en cuanto a conflictos. Si el dato B se tratara como una sola entidad, estas transacciones no podrían ser concurrentes.

T_1	T_2
$bloquear(contenido, A, exclusivo)$	
	$bloquear(contenido, B, exclusivo)$
$bloquear(vista, B, exclusivo)$	
	$bloquear(vista, C, exclusivo)$
$agregar(A, l, B)$	
	$remover(B, C)$

Figura 5.8: Ejemplo de uso del protocolo de bloqueo dual para datos semiestructurados

Cuando se crean o eliminan datos semiestructurados, éstos requieren de un bloqueo exclusivo para la vista y uno para el contenido. También se debe considerar el diccionario de datos como otro elemento de la base de datos semiestructurados, que se puede bloquear en modo exclusivo o compartido.

5.3. El sistema de recuperación

El *sistema de recuperación* se encarga de dejar la base de datos en un estado consistente, en caso de que ocurra algún fallo. Existen fallos de los cuales el sistema manejador de bases de datos se puede recuperar, como una terminación espontánea del sistema manejador o una suspensión de corriente

eléctrica, es aquí donde entra el sistema de recuperación. Hay otros fallos de los cuales no es posible recuperarse, como la destrucción total del sistema de cómputo o del disco, en estos casos es conveniente seguir una política de respaldos.

El método más popular para construir el registro de recuperación es el uso de un *registro histórico* [1, 2], las operaciones que se efectúan a la base de datos primero se escriben en el registro histórico y posteriormente se realizan los cambios a la base de datos. A esto se le conoce como protocolo *WAL* (que es una abreviatura de la frase en inglés: *Write-Ahead Log*). De esta manera si ocurre un fallo el registro histórico tiene la información necesaria para la recuperación.

En este trabajo el sistema de recuperación trabaja a nivel del almacenamiento avanzado, a diferencia del control de concurrencia que se implementa a nivel semántico de datos semiestructurados que es más alto.

Como se vió anteriormente hay cuatro operaciones que se pueden realizar a las unidades en el almacenamiento avanzado:

- Lectura.
- Escritura.
- Liberación.
- Ocupación.

Sólo la lectura no causa modificaciones en la base de datos.

En el registro histórico se pueden almacenar 7 tipos de registros:

- Transacción iniciada.
- Ocupación de unidad.
- Liberación de unidad.
- Escritura de unidad.
- Transacción comprometida.

- Transacción abortada.
- Punto de revisión.

En el registro histórico se usan dos segmentos, uno que contiene los registros llamado LOG y otro que contiene detalles de registros llamado DET. Estos segmentos a diferencia de los del almacenamiento avanzados, sólo se consideran como una secuencia de bytes y no de páginas o unidades.

El contenido general de los descriptores de los registros es:

- Tipo de registro. Ocupa 2 bytes.
- Número de transacción. Ocupa 8 bytes.
- Número de segmento. Ocupa 4 bytes.
- Número de espacio dentro del segmento. Ocupa 4 bytes.
- Número de unidad dentro del espacio. Ocupa 8 bytes.
- Una dirección del segmento DET para los detalles del registro. Ocupa 12 bytes.

En total cada registro ocupa 38 bytes.

Los registros de inicio de transacción, transacción comprometida y transacción abortada, sólo requieren el tipo de registro y el número de transacción. Los registros de liberar y ocupar unidad no requieren de la dirección al segmento DET. Los registros de punto de revisión no requieren de los números de segmento, espacio y unidad. Para los registros de escritura de unidad en el segmento DET se almacena:

- El desplazamiento de la unidad.
- La longitud a escribir.
- El valor anterior.
- El valor nuevo.

Para los registros de punto de revisión en el segmento DET se almacena:

- El número de transacciones activas al momento.
- La lista de transacciones activas.

El sistema de recuperación realiza dos operaciones básicas para cada transacción T:

- *Rehacer* T.
- *Deshacer* T.

El proposito del sistema de recuperación es detectar las transacciones que no se alcanzaron a completar (en este caso se le aplica Deshacer) y las que sí se completaron pero los cambios no se hicieron efectivos en la base de datos (en este caso se le aplica Rehacer).

Cuando se realiza un cambio a la base de datos, primero se efectúa en la memoria intermedia. Así que, aunque en el registro histórico se almacenen los cambios, éstos pueden aún no estar reflejados físicamente, para ello se utiliza el punto de revisión. Cada determinado tiempo el manejador escribe lo que hay en memoria intermedia a los dispositivos físicos, al hacer ésto se escribe un registro de punto de revisión en el registro histórico. En dicho registro se escribe la lista de transacciones activas.

Al deshacer una transacción, las unidades descritas en el registro histórico se restauran al valor anterior, las unidades liberadas se ocupan y las ocupadas se liberan. Al rehacer una transacción las unidades descritas en el registro histórico se fijan a su nuevo valor, las unidades liberadas se liberan nuevamente y las ocupadas se ocupan de nuevo.

El algoritmo para recuperarse de un fallo se describe a continuación [1], para ello se supondrá que se tienen dos listas de transacciones: *listaDeshacer* y *listaRehacer*.

- Se examina el registro histórico desde el final hacia atrás hasta que se encuentre un punto de revisión, para cada registro de transacción comprometida, la transacción correspondiente se añade a la *listaRehacer*.

- Para cada registro de transacción iniciada, la transacción correspondiente se añade a *listaDeshacer*, si es que no aparece en *listaRehacer*.
- Ahora cada transacción que aparece en la lista del punto de revisión, se añade a *listaDeshacer* si es que no está en *listaRehacer*.
- Lo siguiente es rehacer las transacciones que se encuentran en *listaRehacer* y deshacer las que se encuentren el *listaDeshacer*.
- Para deshacer las transacciones de *listaDeshacer*, se recorre nuevamente el registro histórico desde el final hacia atrás y se ejecuta la operación deshacer para cada registro que pertenezca a una transacción de esta lista. Este proceso termina cuando se encuentra el registro de inicio de transacción para todas las transacciones de *listaDeshacer*.
- Para rehacer las transacciones de lista rehacer, se parte del último punto de revisión y se recorre el registro histórico hacia adelante, ejecutando la operación rehacer para cada registro perteneciente a una transacción de *listaRehacer*.

Resumen

En este capítulo, se presentan los conceptos de transacción, concurrencia, secuencialidad y recuperabilidad. Se explicó el uso de los protocolos de bloqueo y en particular el protocolo de bloqueo riguroso de dos fases para el control de la concurrencia. Para el caso especial de los datos semiestructurados, se diseñó un protocolo que tiene el propósito de aumentar la concurrencia en este tipo de datos, a este protocolo se le llamó: protocolo de bloqueo dual para datos semiestructurados. El control de concurrencia fue diseñado en base al protocolo anterior y se logró completar su implementación (véase el apéndice A).

También en este capítulo se presentaron las características que debe tener un sistema de recuperación. Se diseñó el sistema de recuperación para el núcleo del sistema manejador de bases de datos semiestructurados, el sistema de recuperación se basa en la utilización de un registro histórico. El sistema de recuperación involucra problemas de manejo de memoria intermedia y reciclaje de espacio. Desafortunadamente, la implementación del sistema de

recuperación no se alcanzó a completar por cuestiones de tiempo, sin embargo, su diseño está muy involucrado con la funcionalidad de los demás componentes del núcleo.

Apéndice A

Consideraciones de Implementación

En el presente apéndice se describen algunos aspectos acerca del proceso de implementación. Se exponen algunas decisiones tomadas para este fin. Se explica la manera en que está organizado el código, se describe la metodología de desarrollo y finalmente se presentan los resultados de una prueba de desempeño realizada.

A.1. Lenguaje de programación

El lenguaje de programación que se eligió fue Java, la razón para ello es que Java ofrece algunas ventajas sobre otros lenguajes como son:

- Es multiplataforma.
- Permite un buen manejo de contenidos binarios.
- Permite un manejo flexible de archivos de acceso aleatorio.
- Permite la ejecución de procesos simultáneos, sin importar la plataforma.
- Provee un manejo flexible y robusto de la memoria.

- Posee una gran variedad de estructuras de datos en su distribución.
- La documentación se encuentra disponible de manera sencilla.
- Posee herramientas para simplificar el proceso de elaboración de documentación.

Java es un lenguaje que se compila y se interpreta, la ejecución de un programa en un lenguaje interpretado es más lenta que tener directamente el código ejecutable. Sin embargo, en un manejador de bases de datos lo que mas repercute en el tiempo de ejecución son las operaciones de entrada-salida que controla el sistema operativo. Así que, la eficiencia del manejador de bases de datos más bien depende de los algoritmos internos para lograr el menor número de accesos a disco posibles. Por ello es preferible sacrificar un poco de tiempo de ejecución a cambio de las ventajas antes mencionadas.

Para el desarrollo se utilizo la versión 1.5.0_01 de Java, aunque solamente se utilizaron las construcciones básicas del lenguaje, por lo cual el desarrollo debe funcionar en versiones anteriores.

A.2. Organización del código

Los componentes se programaron utilizando la metodología orientada a objetos, para lograr una organización modular y extensible. El código se organizó en cinco paquetes principales, que se describen a continuación (para obtener información más detallada se debe consultar el CD adjunto).

- `ssdbms`: Donde se tiene la definición de las APIs para utilizar los datos semiestructurados y para utilizar el núcleo del sistema manejador de bases de datos semisestructurados. Consta de los siguientes subpaquetes:
  - `ssd`: Donde se definen los objetos para la representación de datos semiestructurados.
  - `manager`: Donde se define el comportamiento del núcleo de sistema manejador de datos semiestructurados.

    - `manager.primitives`: Donde se definen los métodos que representan las primitivas tratadas en el capítulo 3.

- `storage`: En este paquete se tiene la implementación del almacenamiento primitivo y el almacenamiento avanzado, los cuales se explicaron en el capítulo 4. Incluye los siguientes subpaquetes:

    - `primitiveStorage`: Donde se implementan las construcciones diseñadas para el almacenamiento primitivo.
    - `advancedStorage`: Donde se implementan las construcciones diseñadas para el almacenamiento avanzado.
    - `intermediateStorage`: Provee mecanismos para el acoplamiento entre el almacenamiento primitivo y el avanzado.
    - `tools`: Provee una serie de macanismos para apoyar el uso del almacenamiento.

- `DbStorage`: En este paquete se encuentra la implementación del almacenamiento de bases de datos y sus respectivas estructuras, como se explicó en el capítulo 4. Incluye los siguientes subpaquetes:

    - `driver`: Donde se mapean las estructuras del almacenamiento de bases de datos al almacenamiento avanzado.
    - `manager`: Donde se implementan los mecanismos para el manejo de las estructuras del almacenamiento de bases de datos.
    - `primitive`: Donde se incorporan las estructuras del almacenamiento de bases de datos para implementar las primitivas de la interfaz.

- `concurrency`: En este paquete se implementa el control de concurrencia utilizando el protocolo de bloqueo dual para datos semiestructurados, este problema se trató en el capítulo 5.

- `kernelInterface`: Aquí se hace uso de los otros paquetes para implementar la intefaz y sus respectivas primitivas para interactuar con el núcleo, éstas se trataron en el capítulo 3.

De las clases generadas por el código fuente, vale la pena destacar las siguientes:

- `PrimitiveStorageManager` en el paquete `storage.primitiveStorage`. Es la que se encarga del manejo del almacenamiento primitivo.
- `AdvancedStorageManager` en el paquete `storage.advancedStorage`. Se encarga del manejo del almacenamiento avanzado.
- `DbManager` en el paquete `DbStorage.primitives`. Se implementan las primitivas utilizando las estructuras definidas para el almacenamiento de base de datos.
- `ConcurrencyManager` en el paquete `concurrency`. Implementa el control de concurrencia.
- `DbInterface` en el paquete `kernelInterface`. Se implementan las primitivas de la interfaz del núcleo del sistema manejador de bases de datos semiestructurados utilizando los demás módulos.

Los comentarios, nombres de clases, funciones, paquetes y demás componente en el código fueron realizados en inglés. La razón para ello es que el código fuente está dirigido a programadores y desarrolladores, el idioma estándar para este tipo de público es el inglés. De esta manera, en un futuro el desarrollo puede ser accesible para la gente interesada sin importar su nacionalidad.

El desarrollo consta de 77 archivos de código en Java, en total son aproximadamente 11500 líneas de código (sin contar comentarios). El código fuente ocupa 609KB de espacio y el código compilado en Byte Code ocupa 254KB. Se generaron 142 clases en 15 paquetes.

En el CD adjunto se incluye el código fuente en el directorio `src/`, la documentación del código se encuentra en el directorio `doc/` y el código fuente compilado se encuentran en el archivo `ssdbmsKerlnel.jar`.

A.3. Metodología de desarrollo

Durante el desarrollo se siguieron los siguientes pasos:

- En principio se realizó el análisis de requerimientos.

- Se realizó un diseño general del cuál se obtuvieron los 4 componentes principales del núcleo.
- Se realizó un diseño más detallado de cada componente obteniendo así sus respectivos subcomponentes y también se buscó el acoplamiento de todos los componentes.
- Se implementaron los componentes, en este proceso se siguió una metodología ascendente. Es decir, se implementaban primero los paquetes más simples y después los más complejos (que requerían de otras paquetes). En total fueron 15 paquetes para cada uno de ellos se realizaron los siguientes pasos:
  - Se complementaba o corregía el diseño.
  - Se diseñaban y elaboraban las rutinas de prueba y validación.
  - Se programaba el componente.
  - Se realizaban las pruebas.
  - Se corregían los errores pertinentes.
  - Se documentaba el código.
- Finalmente se realizaron las pruebas generales para verificar la integración, la funcionalidad y el desempeño.

A.4. Prueba de desempeño

En el cuadro A.1 se muestran los resultados de una prueba de desempeño realizada. Dicha prueba consistió en crear cierto número de datos semiestructurados y registrar el tiempo en milisegundos que al programa le toma. En la primera columna aparece el número de datos a crear. En la segunda columna aparece el tiempo que tarda el manejador de almacenamiento sin utilizar el control de concurrencia. En la tercera columna aparece el tiempo que tardó el manejador de almacenamiento junto con el control de concurrencia. Para la cuarta columna cada dato creado se añade al contenido de un dato no primitivo creado previamente. En la quinta columna aparece el

tiempo que tardó el manejador de bases de datos relacionales Postgres[1] en crear el número de registros especificado en la primera columna.

En los resultados obtenidos por la prueba de desempeño, se puede apreciar que los tiempos obtenidos por este trabajo son más pequeños que los de Postgres. Ésta no es una comparación del todo válida ya que ambos sistemas tratan tipos de datos de distinta naturaleza, sin embargo, sirve para dar una idea del desempeño obtenido, que se puede apreciar es bastante satisfactoria.

Datos	Almacenamiento	Con concurrencia	Con adición	Postgres
1000	500	437	2169	46031
10000	3860	4313	43766	414188
100000	138328	145125	972015	3492906

Cuadro A.1: Prueba de desempeño

Resumen

En este apéndice, se presentaron los detalles de implementación de la elaboración de los componentes para el núcleo del sistema manejador de bases de datos semiestructurados. La implementación realizada como parte del trabajo, se incluye en el CD adjunto.

Se explicó y justificó la elección del lenguaje de programación utilizado. Se explicó la manera en la que se encuentra organizado el código fuente. Se expuso la metodología de desarrollo seguida en el proceso de construcción del software. Finalmente se mostraron los resultados de una prueba de desempeño, la cual produjo resultados satisfactorios.

[1]Postgres es un sistema manejador de bases de datos relacionales, es gratuito y de código abierto, es muy utilizado en los ámbitos universitarios y gubernamentales.

Conclusiones y Perspectivas

El presente trabajo forma parte de una serie de investigaciones que tienen como propósito el estudio de los datos semiestructurados y la elaboración de herramientas para su utilización. Una de los principales herramientas que se desea obtener es un sistema manejador para bases de datos semiestructurados. En este trabajo se abordó el problema de diseñar y construir los componentes básicos que conforman dicho sistema. Este conjunto de componentes es conocido como el núcleo, dichos componentes son: La interfaz, el manejo de almacenamiento, el control de concurrencia y el sistema de recuperación. En la realización de este trabajo se invirtió en investigación, diseño e implementación. El proceso de diseño de los componentes del núcleo fue particularmente difícil y delicado, debido a la gran interdependecia que los componentes guardan entre sí, en el diseño de cada componente se tienen que considerar los demás.

Las conclusiones obtenidas de la realización de este trabajo son:

- El modelo de datos semiestructurados y sus bases de datos son útiles para enfrentar las necesidades actuales de información, lo cual se puede concluir de los temas abordados en los capítulos 1 y 2.

- El núcleo puede ser utilizado por otros módulos de software, ya que se construyó una interfaz que incluye un conjunto de operaciones básicas o primitivas creadas para la manipulación de las bases de datos semiestructurados almacenadas. La descripción de las primitivas se abordó en el capítulo 3.

- Se logró un almacenamiento nativo de datos semiestructurados, ya que la construcción aquí presentada partió de las unidades básicas de los

sistemas de cómputo y no se basó en los paradigmas o modelos de almacenamiento que utilizan otros sistemas de bases de datos. En el capítulo 4 se abordó la construcción del sistema de almacenamiento de bases de datos semiestructurados.

- El sistema de almacenamiento posee un buen desempeño y es factible para ser utilizado en aplicaciones que requieran almacenar una gran cantidad de información, lo cual se puede concluir a partir de las capacidades de almacenamiento que se muestran en las tablas 4.1 y 4.2 además de los resultados de la prueba de desempeño en la tabla A.1.

- El mecanismo construido para el control de concurrencia es factible para su utilización en datos semiestructurados, ya que se creó el protocolo de bloqueo dual para datos semiestructurados, que resuelve los problemas que se presentan al utilizar un protocolo de bloqueo común causados por la naturaleza de los datos semiestructurados. Esto fue tratado en el capítulo 5.

- La implementación realizada es modular y extensible, ya que la programación se desarrolló utilizando la metodología orientada a objetos y el lenguaje de programación Java que utiliza estas características. Los detalles de implementacion se trataron en el apéndice A.

Las principales aportaciones de este trabajo en el campo de las bases de datos son:

- La creación de una serie de primitivas para manipular la información en una base de datos semiestructurados, lo que es el equivalente al álgebra relacional en las bases de datos relacionales.

- El diseño de los componentes que forman el núcleo de un sistema manejador de bases de datos semiestructurados. En la actualidad no existe un manejador de bases de datos semiestructurados que sea suficientemente estable, los sistemas similares que se dedican al almacenamiento de documentos XML se enfocan más bien hacia la publicación en la web de este tipo de información (repositorio) en lugar de enfocarse al almacenmiento y tratamiento de grandes cantidades de información (base de datos).

- La construcción (implementación) de un mecanismo para el manejo de almacenamiento nativo de los datos semiestructurados, con un desempeño mayor al que poseen los mecanismos no nativos que se utilizan para este fin.
- La creación de un protocolo de bloqueo para resolver los problemas que presentan los protocolos de bloqueo comunes debido a la naturaleza de los datos semiestructurados.

El trabajo que queda por delante es completar el sistema manejador de bases de datos semiestructurados y aplicarlo a los dominios que requieran del manejo de este tipo de información (se han identificado problemas en biología, bibliotecología, integración de información, etc.). Como trabajos futuros se consideran:

- La implementación del sistema de recuperación.
- La construcción del sistema de respaldos.
- La construcción del sistema de seguridad.
- La construcción de herramientas para facilitar el uso y administración del manejador.
- La construcción de las APIs para la comunicación con diversos lenguajes de programación.
- La optimización de los componentes del núcleo.

Bibliografía

[1] A. Silberschatz, H. Korth & S. Sudarshan. *Database System Concepts*. McGraw-Hill. 4a. edición, 2002.

[2] R. Ramakrishnan & J. Gehrke. *Database Management Systems*. McGraw-Hill. 3a. edición, 2003.

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

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

[5] R. Elmasri & S. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 3a. edición, 2000.

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

[7] A. Tanenbaum. *Sistemas Operativos Modernos*. Prentice Hall. 1992.

[8] A. Silberschatz, P. Galvin & Greg Gagne. *Sistemas Operativos*. Limusa Wiley. 6a edición, 2002.

[9] J. Giarratano & G. Riley. *Sistemas Expertos, principios y programación*. Prentice Hall. 3a edición, 2001.

[10] E. Rich & k. Knight. *Inteligencia Artificial*. McGraw-Hill. 2a edición, 1994.

[11] M. Weiss. *Estructuras de Datos en Java*. Addison Wesley. 2000.

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

[13] A. Bergholz. *Lore Tutorial*. Incluido en la distribución de Lore. http://www-db.stanford.edu/lore. Mayo, 2002

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

[15] S. Abiteboul. *Quering Semi-Structured Data.* International Conference on Database Theory, vol. 6, pp.1-18, 1997.

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

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

[18] M. Graves. *Designing XML Databases.* McGraw-Hill, 2002.

[19] *http://www.dbxml.com*. Sitio en la Web de dbXML. Abril, 2005.

[20] *http://www1.softwareag.com/Corporate/products/tamino*. Sitio en la Web de Tamino. Abril, 2005.

[21] *Introduction to IBM DB2 Universal Database.* IBM Data Managemet Solutions Edication Services, 2003.

[22] *Overview of IBM Informix Dynamic Server.* International Business Machines Corporation, 2001.

[23] *SQL Server y Servicios de datos.* Microsoft Corporation, 2000.

[24] K. Loney & M. Theriault. *Oracle 9i, Manual del administrador.* McGraw-Hill, 2002.

[25] E. García. *Un Lenguaje de Consulta para Bases de Datos Semiestructurados.* Tesis de Licenciatura, Facultad de Ciencias, UNAM, 2002.

Índice alfabético