BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

METODOLOGÍA PARA ACCESO Y PROCESAMIENTO DE DATOS ESPACIALES UTILIZANDO R^+-TREES

TESIS PROFESIONAL
QUE PARA OBTENER EL TÍTULO DE
MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA
HUGO ALEJANDRO RAMOS CRUZ

ASESOR
DRA. MARÍA JOSEFA SOMODEVILLA GARCÍA

PUEBLA PUE. **AGOSTO 2006**

INTRODUCCIÓN

El espacio y el tiempo son dos características de cualquier objeto del mundo real, de esta forma un objeto se caracteriza por su posición y área que ocupa en cualquier punto de tiempo. En la actualidad existen muchas aplicaciones que requieren manejar de forma eficiente los atributos espaciales de un objeto para poder modelar, analizar y dar soluciones a problemas que involucran este tipo de datos.

Aplicaciones como Sistemas de Información Geográfica (GIS), Diseño Asistido por Computadora (CAD), Sistemas de Información Multimedia, sistemas para la observación de transportes, medioambiente, demografía así como bibliotecas digitales fueron los principales promotores del impulso a la investigación en esta área.

Los sistemas manejadores de bases de datos espaciales proporcionan modelos y algoritmos para el manejo eficiente de datos espaciales utilizando los distintos mecanismos de acceso multidimensional. Estos manejadores pueden dar solución a los distintos tipos de consultas espaciales.

Gran cantidad de estas aplicaciones requieren modelar ambientes que tienen una alta complejidad en la definición de sus atributos y restricciones. En un sistema informático estos datos espaciales se representan por puntos, líneas, polígonos, regiones, etc., que se les conoce con el nombre de objetos espaciales. Para responder a consultas relacionadas con propiedades espaciales, se implementan algoritmos eficientes sobre índices espaciales creados a partir de esos objetos.

Este trabajo está enfocado a analizar e implementar los algoritmos básicos para el acceso y procesamiento de datos espaciales mediante los métodos de indexamiento multidimensionales, específicamente cuando utilizamos los R^+-trees como estructura de datos y utilizando los Rectángulos de Mínimo Acotamiento (MBR) para representar de forma aproximada al objeto.

Los R^+-trees se clasifican como la estructura de datos que dan soporte al método de indexamiento por recorte. Esta técnica descompone el espacio de manera jerárquica, los objetos son almacenados en las hojas mientras los nodos intermedios facilitan las búsquedas con la característica que no permite solapamiento entre ellos.

Como un resultado del análisis y adecuamiento de las técnicas y métodos expuestos anteriormente se propone una metodología para el acceso y procesamiento de datos espaciales. Esta metodología incluye la descripción de los algoritmos básicos que nos permitan el almacenamiento, búsquedas y respuestas eficientes a la mayor cantidad de consultas espaciales.

OBJETIVO GENERAL

Proponer una metodología que permita el manejo de datos espaciales así como de las principales relaciones topológicas (consulta de punto y de ventana) además de poder interactuar con la base de datos a través de los procesos de *join* y *semijoin* espacial, usando como estructura de datos a los R^+-trees.

OBJETIVOS PARTICULARES

- Plantear una propuesta para el indexamiento de objetos espaciales a través de una implementación de R^+-trees utilizando los *MBR´s* de los objetos.
- Diseñar la metodología para el acceso de datos espaciales, así como los algoritmos para el procesamiento de consultas espaciales.
- Diseñar el algoritmo del *Join* espacial basado en un *semijoin*.
- Implementación y pruebas de la metodología propuesta con datos espaciales reales.
- Resaltar la ventaja en términos de espacio de almacenamiento y accesos a la base de datos, entre el uso de un *join* y un *semijoin* espacial.
- Uso de un lenguaje de programación orientado a objetos que permita el uso de clases, así como para su reutilización en trabajos futuros.

ORGANIZACIÓN DE LA TESIS

El trabajo de tesis se encuentra organizado en seis capítulos, en el Capítulo I se da un panorama sobre las bases de datos espaciales, los distintos mecanismos de indexamiento existentes.

En el Capítulo II se define un dato espacial así como sus principales características, además se describe a detalle la clasificación de los Métodos de Acceso Multidimensional así como las Estructuras de Datos Espaciales más importantes.

La clasificación de las consultas espaciales así como su descripción a través de un ejemplo son consideradas en el Capítulo III.

La metodología de indexamiento usando R^+-trees propuesta, así como los algoritmos necesarios para el almacenamiento, búsqueda y consultas espaciales son presentados en el Capítulo IV.

La implementación de la propuesta así como el comportamiento de un ejemplo clásico de los métodos de indexamiento espacial son reportados en el Capítulo V, los cuales nos servirán para poder dar un grado de confianza al trabajo presentado.

Finalmente se presentan las conclusiones obtenidas respecto al plan de trabajo presentado, así como los retos y trabajo futuro.

CAPÍTULO I

Marco Teórico

Este trabajo está enfocado en plantear una propuesta de metodología para trabajar con datos espaciales mediante la utilización de R^+-trees. Las bases de datos espaciales en un contexto general almacenan objetos que tienen características espaciales que los describen.

El estudio de las bases de datos espaciales tiene más de tres décadas de continua investigación, tiempo en el cual se han desarrollado métodos de acceso multidimensional que proporcionan un eficiente indexamiento de objetos espaciales en bases de datos.

Uno de los principales objetivos de un sistema de gestión de base de datos es proporcionar métodos de acceso y algoritmos eficientes para el procesamiento de consultas. Para aumentar el rendimiento de dichos sistemas surgen los métodos de acceso. En este sentido los índices unidimensionales más estudiados han sido los B-tree [20] y sus variaciones.

El uso de índices unidimensionales basa su hecho en tener un punto de referencia que identifique de forma única al objeto que representa. Esta misma idea se extiende para los índices multidimensionales, en este sentido existen diversidad de formas en las cuales representar objetos de más de una dimensión; ejemplos de estas son: representar un objeto por el área que tiene, mediante las curvas de *space filling* [22], su centroide, alguna esquina del objeto entre otras.

Una segunda estrategia consiste en extender el índice de unidimensional a multidimensional, y es en este sentido en el que trabajaremos. Dado que resulta más general y tiene ventajas como poder representar al objeto de una manera más parecida a lo real.

En base a este grupo de índices se modifican sus algoritmos para permitir indexar objetos multidimensionales que se les conoce con el nombre de *familia de R-trees* (estructuras de datos arbóreas basadas en rectángulos de mínimo acotamiento, balanceados en altura y almacenados en disco), entre los cuales están el R-tree original [8], R+-tree [10], R*-tree [9], X-tree [21], entre los más importantes.

El término *dato espacial*, agrupa objetos multidimensionales como los son: puntos, líneas, polígonos, cubos entre algunos otros. Un objeto espacial, por tanto, ocupa cierta región en el espacio, la cual será representada por su ubicación y límite.

Para aumentar el rendimiento en la recuperación de los objetos espaciales se pueden definir índices espaciales. Dichos índices representan la ubicación de los objetos espaciales que están almacenados en la base de datos. Sin embargo, para simplificar el manejo de dichos índices se suele utilizar algún tipo de aproximación de los objetos espaciales. El rectángulo de mínimo acotamiento (MBR) de esos objetos espaciales es la aproximación más utilizada. Dichos MBR´s representan la clave de indexación que se utiliza para generar los índices que pertenecen a la familia de R-trees.

Un objeto espacial es un tipo especial de dato, el cual tiene características no-estándar que hacen su manejo complejo. De acuerdo a [1] las características que definen un objeto espacial son: su *estructura compleja*, son frecuentemente *dinámicos*, las bases de datos espaciales tienden a ser *grandes*, *no hay un álgebra espacial estándar* y los operadores espaciales son *no cerrados*.

Los métodos de acceso multidimensional en principio se clasifican en: basados en memoria principal y basados en almacenamiento secundario. El requerimiento de los usados en memoria principal los hacía inservibles en muchas aplicaciones donde el volumen de datos hace necesario el uso de memoria secundaria. Sin embargo la importancia de estas estructuras radica

en haber sido las primeras en considerar la posibilidad de indexado multidimensional.

El segundo grupo de métodos de acceso multidimensional almacenados en memoria secundaria, hace necesario que los Sistemas de Administración de Bases de Datos Espaciales ofrezcan mecanismos y operadores espaciales que permitan realizar almacenamiento, búsqueda y consultas espaciales eficientes, existe una clasificación de estos métodos que de acuerdo a [2] son:

1. Métodos de Acceso al Punto (PAM) que generalmente organizan los datos en cubetas, donde cada una corresponde a una página de disco y a un sub-espacio del universo, así mismo éstos métodos se categorizan en:
    a. Métodos de Acceso de Hashing Multidimensional: estos métodos usan hashing de 1-dimensión para indexar puntos de d-dimensiones. Ejemplos de este método son el Grid File [3] y EXCELL [4].
    b. Métodos de Acceso Jerárquico: estos métodos usan estructuras de datos jerárquicas para manejar los datos. Los principales ejemplos son el Quadtree [5], el k-d-tree [6] y el k-d-B-tree [7].
2. Métodos de Acceso Espacial (SAM) que son extensiones de los PAM´s, usados para cubrir las necesidades espaciales, se clasifican en:
    a. Métodos de mapeo de objetos: mapean objetos geométricos en puntos de un espacio multidimensional.
    b. Métodos de limitación de objetos: descomponen el espacio de manera jerárquica. Los objetos son almacenados en las hojas de una estructura jerárquica y los nodos intermedios facilitan las búsquedas. Los métodos más importantes son el R-tree [8] y el R*-tree [9].
    c. Métodos de recorte: son extensiones de los métodos anteriores con la característica que entre nodos del mismo nivel no existe solapamiento entre objetos. El R+-tree [10] es el más conocido dentro de esta clasificación.

d. Múltiples Capas: particionan el espacio más de una vez y cada una de ellas forma una capa. Un método característico es el Multi-layer Grid File [11].

En la Figura 1.1 se muestra una tabla comparativa de los métodos de acceso multidimensional basado en memoria secundaria, así como sus estructuras de datos más significativas de cada grupo.

Métodos de Acceso a Puntos (PAM)		Métodos de Acceso Espacial (SAM)	
Hashing	*Jerárquicos*	*Baja Dimensión*	*Alta Dimensión*
Grid File	k-d-tree	R-tree	X-tree
Excell	k-d-B-tree	R+-tree	
		R*-tree	

Figura 1.1 Clasificación general de los métodos de acceso en almacenamiento secundario

Existen gran cantidad de aplicaciones que hacen uso de datos con atributos espaciales, donde algunos de ellos no tiene límites bien definidos o no pueden ser determinados de forma concisa. Ejemplo de este tipo de objetos son los océanos, ríos, regiones climáticas, densidad de poblaciones, valles y montañas. Esta situación ocasiona que el manejo de los objetos espaciales con todas sus características sea para las bases de datos espaciales computacionalmente difíciles de manejar.

El Rectángulo de Mínimo Acotamiento [12] es el rectángulo más pequeño que cubre mínimamente al objeto. El Rectángulo de Mínimo Acotamiento para nuestro caso será 2-dimensional, de esta forma una entrada MBR será de la forma (x_{min}, y_{min}, x_{max}, y_{max}) que representan las coordenadas de la esquina inferior izquierda y la esquina superior derecha del objeto.

De esta forma nuestro trabajo consistirá en proponer una metodología que permita integrar los distintos algoritmos de inserción, borrado, y consultas espaciales en los mecanismos de indexamiento multidimensional,

específicamente cuando ocupemos el método de recorte que tiene como estructura de datos a los R^+-trees.

El trabajo está basado en los algoritmos de indexamiento del R^+-tree, con la característica de almacenar los Rectángulos de Mínimo Acotamiento de cada objeto respecto a su paquete, dicha aproximación estará dada por la implementación de los MBR en los paquetes de indexamiento.

Por otro lado, existen operaciones que son de vital importancia debido a que relacionan dos o más tablas de la base de datos espacial, dicho operador es el join. El join espacial [23] sobre dos conjuntos de objetos espaciales devuelve todos los pares de objetos que se solapan entre si, donde cada elemento del par pertenece a un conjunto diferente. Introducimos también una modificación al join que reduce a la mitad los accesos a la base de datos, originando una respuesta más rápida a la consulta, dicha modificación es el semijoin.

El semijoin [19] tiene sus orígenes en las bases de datos distribuidas. La transferencia de tuplas de un nodo *R* a otro *S* resultaba muy costosa, por esta razón la modificación fue enviar solo la columna de la relación R con la que se va a realizar el join al otro nodo. Una vez en este nodo se lleva a cabo el join con la relación *S* alojada en este nodo, las columnas implicadas en el resultado se envían nuevamente al nodo inicial y se realiza el join con la relación *R*. Esta idea es adaptada para modificar el join espacial y aplicarla a los árboles que representan una relación espacial.

En los Capítulos II y III de este documento se detallan los métodos y procedimientos esbozados en este marco teórico.

CAPÍTULO II
Mecanismos de Indexado Espacial

Introducción

Los índices son archivos auxiliares usados para acelerar la búsqueda de un dato. Los registros en un índice solo tienen dos campos: el valor de la llave y la dirección de una página en el archivo de datos. Los registros en un archivo de índices están regularmente ordenados y pueden ser organizados más a fondo usando estructura de datos especializadas como son el B-tree, el R-tree y el Grid File entre otros [14].

Debido a que los registros están ordenados en un archivo de índices, las búsquedas binarias pueden ser usadas para realizar consultas en los archivos de índices incluso si el archivo no estuviera ordenado. Además, el archivo de índice tiende a ser pequeño y en consecuencia la búsqueda es más rápida. Una vez que el índice apropiado es hallado, el registro de datos es extraído recuperando el archivo de datos de la página de disco apuntada por el registro índice.

2.1 Datos Espaciales

2.1.1 Características de los datos espaciales

Los datos espaciales son considerados como una tipo especial de dato, pues tienen algunas características no estándar que hacen su manejo complejo para la base de datos. Estas características las podemos definir como:

a) *Los objetos espaciales tienen estructura compleja*: Un simple punto, o un conjunto arbitrario de polígonos pueden representar un objeto espacial. Las tuplas en una base de datos relacional con tamaño fijo no son recomendables para almacenar dicha variedad de formato de datos. Como resultado, las operaciones espaciales (como intersección, unión) son computacionalmente *más costosas* que las operaciones estándar de un RDBMS.

b) *Los datos espaciales son frecuentemente dinámicos*: Esta característica de los datos espaciales requiere estructuras de datos robustos para inserciones, borrado y actualización de objetos.
c) *Las bases de datos espaciales tienden a ser grandes*: El número de objetos en un mapa geográfico, o en un circuito VLSI a menudo demandan gran cantidad de gigabytes de almacenamiento. La integración de memoria secundaria en estructuras de datos espaciales es por consiguiente necesario.
d) *No hay un álgebra espacial estándar*: No hay definido un conjunto de operadores espaciales estándar, éstos usualmente dependen del dominio de la aplicación de la base de datos específica.
e) *Los operadores espaciales son no cerrados*: La intersección de dos objetos espaciales, por ejemplo, pueden regresar un conjunto de puntos, líneas o regiones.

Otra importante característica concernientes a los datos espaciales, es que al ser multidimensionales hace difícil aplicar métodos de indexamiento tradicionales como los B-trees o hashing lineal.

2.1.2 Requerimientos de los métodos de acceso espacial

Las propiedades de los datos espaciales mencionados anteriormente hacen el diseño de los métodos de acceso espacial una tarea laboriosa, que tienen una variedad de requerimientos a cubrir como lo son [1]:

a) *Dinámicos:* Debido a que los objetos espaciales pueden ser insertados y eliminados de la base de datos espacial en cualquier orden dado, los métodos de acceso deben guardar o actualizar continuamente las operaciones sobre estos cambios.
b) *Manejo de memoria secundaria/terciaria:* Los métodos de acceso espacial necesitan integrar eficientemente el almacenamiento secundario y terciario.
c) *Soporte a varias operaciones:* Un amplio rango de operaciones espaciales deben ser soportados.

d) *Independencia de los datos de entrada y de las secuencias de inserción:* El rendimiento de un método de acceso no debe depender ni del tipo de dato de entrada, ni del orden en el cual se insertan.
e) *Escalabilidad:* Los métodos de acceso deben adaptarse bien al crecimiento de las bases de datos.
f) *Eficiencia en el espacio y el tiempo:* Los métodos de acceso espacial deben funcionar rápido, con un rendimiento logarítmico en el peor caso, dado cualquier conjunto de datos de entrada. El indexado debe ser pequeño y debe garantizar una eficiente utilización del espacio.

2.2 Clasificación de los Métodos de Acceso Multidimensional

Gaede [2] clasifica los métodos de acceso a datos multidimensionales en *Métodos de Acceso al Punto* (PAM) y *Métodos de Acceso Espacial* (SAM). Los PAM fueron diseñados en primera instancia para realizar búsquedas espaciales sobre bases de datos de puntos; las bases de datos que almacenan únicamente puntos multidimensionales pero que no tienen una extensión espacial. Por otro lado los SAM manejan objetos que, aparte de su posición en el espacio, tienen características espaciales (forma). Tales objetos son líneas, polígonos, o poliedros de mayores dimensiones.

2.2.1 Métodos de Acceso al Punto (PAM)

Estos métodos generalmente organizan los datos de punto en cubetas, cada una de éstas corresponde a una página de disco y algún sub-espacio del universo. La cubeta, usualmente rectilínea es indexada tanto por estructuras de datos planas como jerárquicas. Existen muchos PAM que son híbridos y por tanto no pueden ser clasificados en un solo grupo. En [2] se reporta la siguiente clasificación de éstos métodos:

- Métodos de Acceso de Hashing Multidimensional. Estos métodos usan hashing de dimensión-1 para indexar puntos de n-dimensiones. Sin embargo no existe un orden total de los objetos n-dimensionales dentro de una dimensión, éstos métodos usan técnicas heurísticas para asegurar que dos objetos que son cercanos uno del otro en el espacio

multidimensional, sean indexados en la misma cubeta o en una cubeta cercana. Ejemplos de este métodos son el Grid File [3] y EXCELL [4].

- Métodos de Acceso Jerárquicos. Estos métodos usan estructuras de datos jerárquicos para manipular datos de punto. Los PAM que pertenecen a esta categoría son el Quadtree [5], el k-d-tree [6], y el k-d-B-tree[7].

Los métodos de acceso multidimensional frecuentemente usan las llamadas *curvas de relleno del espacio* (space-filling) para conservar la proximidad espacial cuando se ordenan puntos multidimensionales dentro de un espacio de 1-dimensión. Estas técnicas sugieren un orden total de los objetos espaciales, asegurando con una alta probabilidad que si dos objetos se encuentran cercanos en el espacio original entonces ellos estarán cercanos en el orden total.

2.2.2 Métodos de Acceso Espacial (SAM)

Los métodos de acceso al punto no pueden ser usados directamente para manejar objetos con extensión espacial. Los métodos de acceso espacial son a menudo extensiones de los PAM, usados para cubrir esta necesidad. En [2] se clasifican estos métodos de acuerdo a las técnicas que ellos usan para ampliar los PAM, esta es la siguiente:

- Métodos de Mapeo de Objetos. Estos métodos mapean objetos geométricos en un espacio de dimensión mayor. Por ejemplo, un rectángulo en R^2 puede ser visto como un punto en R^4. De esta forma usan PAM existentes para manejar los puntos. Una alternativa usada por algunos métodos es la descomposición de objetos geométricos en objetos simples (por ejemplo rectángulos) y ordenarlos usando curvas *space-filling*.
- Métodos de Limitación de Objetos (regiones traslapadas). Constituyen los más populares SAM, estos métodos descomponen el espacio de forma jerárquica. Los objetos son almacenados en las hojas de las estructuras jerárquicas, y los nodos intermedios facilitan las búsquedas eficientes. Los nodos en un mismo nivel pueden solaparse, así el número de rutas

que habrán de seguirse en la búsqueda de un objeto puede variar. Los métodos de limitación de objetos más prometedores son el R-tree [8] y el R^*-tree [9].

- Métodos de Truncamiento. Estos métodos usan estructuras de datos jerárquicas, como lo hacen los métodos de limitación de objetos, pero éstos usan truncamiento de objetos para prevenir el solapamiento de nodos intermedios en un mismo nivel. De esta manera se garantiza que solo una ruta de la estructura jerárquica tendrá que ser seguida para buscar un objeto. Los objetos son recortados y almacenados en algunos nodos para garantizar su característica de no-traslape. Ejemplo de este método es el R^+-tree [10].
- Múltiples Capas. Los métodos de múltiples capas particionan el espacio en más de una ocasión y cada partición es referida como una capa. Las capas son organizadas de manera jerárquica, y las regiones particionadas dentro de una misma capa no se traslapan. Cada capa puede usar un algoritmo diferente para particionar el espacio. Un objeto es almacenado en una región de la capa más baja posible en la jerarquía que pueda almacenar el objeto sin recortarlo. Un método de acceso de múltiples capas característico es el Multi-layer Grid File [11].

2.3 Estructuras de Datos Espaciales

A continuación se presenta una descripción detallada de los métodos más importantes de acceso al punto y espaciales, de forma especial los árboles ya que constituyen la estructura de datos que se utilizará para el desarrollo de la metodología propuesta en este trabajo.

2.3.1 Métodos de Acceso Hashing

2.3.1.1 El Grid File y sus variantes

El Grid File es una variación de los métodos grid, los cuales se basan en el requerimiento de que las líneas de división de celdas deben ser equidistantes. Su meta es recuperar registros a lo más en dos accesos a disco y manejar eficientemente las consultas de rango. Esto se hace usando un

directorio grid que consiste de bloques grid que son análogos a las celdas de los métodos de grid-fijos. Todos los registros dentro de un bloque grid son almacenados en una misma cubeta. Sin embargo, algunos bloques grid pueden compartir una cubeta tan grande como la unión de estos bloques que formen un rectángulo de dimensión-k en el espacio de registros. Aunque las regiones de las cubetas son disjuntas, juntas alcanzan el espacio de registros. Para garantizar que los datos son siempre encontrados a los más en dos accesos a disco para consultas de empate exacto, el grid en si es almacenado en memoria principal, representado por *d* arreglos unidimensionales llamados *escalas*.

La Figura 2.1 muestra un *Grid File* con cubetas de capacidad para cuatro datos de tipo punto. El centro de la figura muestra el directorio con las escalas sobre el eje *x* y el eje *y*. Para responder a una consulta exacta, primero usamos las escalas para localizar la celda que contiene el punto buscado. Si la celda grid correspondiente no está en memoria principal, un acceso a disco es necesario, y la celda cargada contendrá una referencia a la página donde posiblemente encontraremos el dato que responda la consulta.

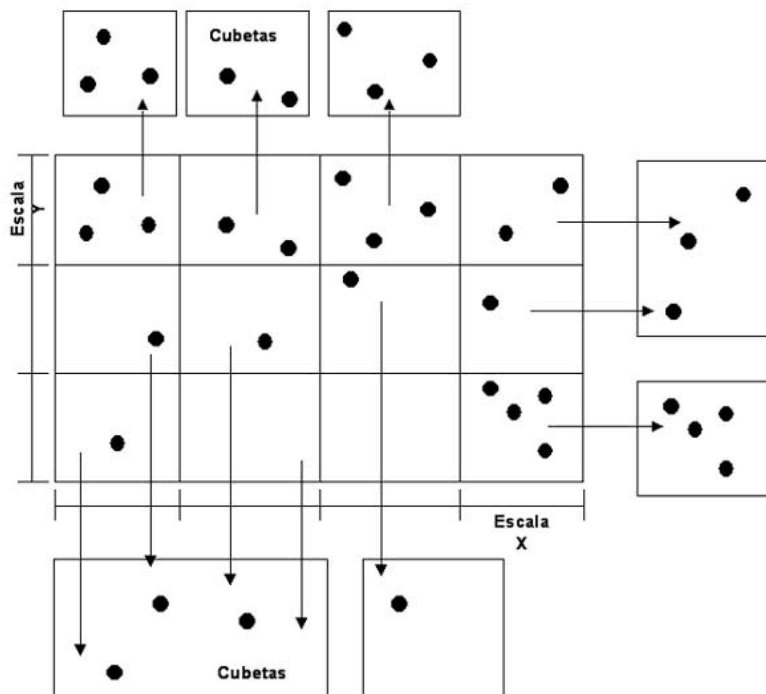


Figura 2.1 Grid File

2.3.1.2 Otros métodos hashing

Estrechamente relacionado con el Grid File está el método EXCELL propuesto por Tamminen [4]. Es un árbol binario junto con un directorio en la forma de un arreglo que proporciona acceso por cálculo de direcciones. También puede ser visto como una adaptación del hashing extendido para datos de puntos multidimensionales. En contraste con el Grid File, donde la partición de hiperplanos debe ser arbitrariamente espaciada, el método EXCELL descompone el universo de forma regular; todas las celdas grid son de igual tamaño. Para mantener esta propiedad bajo la presencia de inserciones, cada nueva división resulta en tener todas las celdas y en consecuencia en el doble del tamaño del directorio.

Otro método hashing considerado es el *Grid File de dos-niveles* [17]. La idea básica es usar un segundo grid file para manejar el directorio grid. El primer de los dos niveles es llamado *directorio raíz*, el cual es una versión rústica del segundo nivel, el actual directorio grid. Las entradas del directorio raíz contienen apuntadores a las páginas del directorio del nivel más bajo, los cuales contiene apuntadores a las páginas de disco. El hecho de tener un segundo nivel, hace que las divisiones sean a menudo confinadas a las regiones de subdirectorios sin afectar demasiado sus alrededores.

El *twin grid file* [18] es otro método hashing que intenta incrementar el espacio de utilización comparado con el original grid file por medio de la introducción de un segundo grid file. La relación entre estos dos grid files no es jerárquica pero es más balanceada. Ambos grid files cubren todo el universo. La distribución de los datos a lo largo de los dos archivos es realizada dinámicamente. Si el número de puntos en una cubeta excede el límite, el *twin grid file* intenta redistribuir los puntos entre los dos grid files. La transferencia de puntos de un archivo primario P a uno secundario S debe conducir a un desbordamiento en S. Sin embargo también puede causar un *underflow* en P, lo cual debe llevar a la fusión de cubetas y por tanto una reducción de cubetas en P. El objetivo principal de la modificación es minimizar el número total de cubetas en los dos grid files P y S.

2.3.2 Quadtrees

Los Quadtrees son uno de las primeras estructuras de datos para datos de dimensiones grandes. Fueron desarrolladas por Finkel y Bentley en 1974 [5]. Desde entonces, centenares de artículos tratan acerca de Quadtrees.

Un Quadtree es un árbol con raíz en el cual los nodos internos tienen cuatro hijos. Cada nodo en un Quadtree corresponde a un cuadro. Si un nodo *v* tiene hijos, entonces sus correspondientes cuadros son los cuatro cuadrantes del cuadro *v*. Esto implica que los cuadros de las hojas en conjunto forman una subdivisión del cuadro de la raíz, a esto llamaremos *subdivisión Quadtree*. La Figura 2.2 muestra un ejemplo de un Quadtree y sus correspondientes subdivisiones. Los hijos de la raíz están etiquetados como NE, NW, SW y SE.

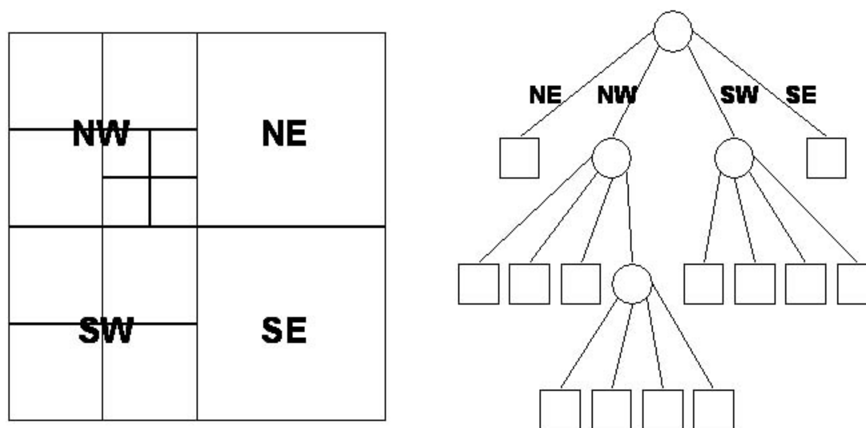


Figura 2.2 Un Quadtree y sus subdivisiones

La definición recursiva de un Quadtree inmediatamente nos lleva a un algoritmo recursivo: dividir el cuadro actual en cuatro cuadrantes donde cada cuadrante marca una nueva entrada, de esta forma se construye recursivamente un nuevo Quadtrees por cada entrada con su conjunto de datos de entrada asociados.

Los Quadtrees pueden ser diferenciados bajo las siguientes bases:

a) El tipo de datos que utilizan para representar
b) El principio que dirige el proceso de descomposición
c) La resolución (variable o no)

Actualmente los Quadtree son usados para datos de tipo punto, áreas, curvas, superficies y volúmenes. La descomposición debe ser en partes iguales en cada nivel, o puede ser manipulado por la entrada. La resolución de la descomposición se puede fijar con anterioridad, o se puede administrar por las propiedades de los datos de entrada.

2.3.3 El k-d-tree y sus variantes

Una de las estructuras de datos multidimensionales más prominentes es el *k-d-tree*[6]; un árbol de búsqueda binario que almacena puntos de un espacio de dimensión-k. En cada nodo intermedio, el k-d-tree divide el espacio de dimensión-k en dos partes por un hiperplano de dimensión-(k-1). La dirección del hiperplano alterna entre las *k* posibilidades de un nivel del árbol al otro. Cada división del hiperplano contiene al menos un punto, el cual es usado para representar el hiperplano en el árbol. La Figura 2.3 ilustra un 2-d-tree con algunos puntos en él.

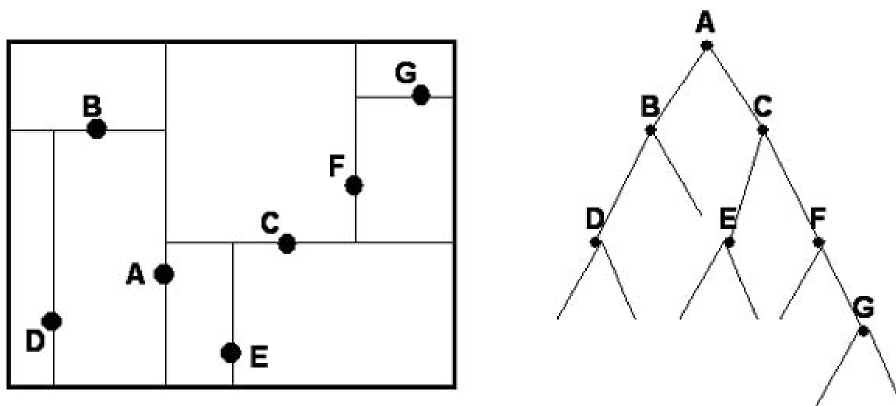


Figura 2.3 División de los puntos y su k-d-tree correspondiente

Las búsquedas e inserciones de nuevos nodos son realizadas de forma directa. El borrado puede causar una reorganización del sub-árbol formado debajo del nodo eliminado, lo cual resulta más complicado. La estructura del árbol depende en gran medida del orden de inserción de los puntos de entrada. Otra desventaja de los k-d-tree es que como la división de hiperplanos es definida por la posición de los puntos, el árbol resulta frecuentemente no balanceado.

El *k-d-B-tree*[7] combina propiedades del k-d-tree y del B-tree para cubrir esta debilidad. De igual forma usa hiperplanos para dividir el espacio; esta vez más de una hiperplano divide el nodo del árbol en un número correspondientes de regiones disjuntas. Todos los nodos del árbol corresponden a una página de disco. Una hoja almacena los datos de punto que son localizados en su respectiva partición de la hoja definida. Así como los B-tree, el k-d-tree está perfectamente balanceado, sin embargo, no asegura la utilización eficiente de almacenamiento. La Figura 2.4 muestra como una distribución de puntos puede ser almacenada en un k-d-B-tree.

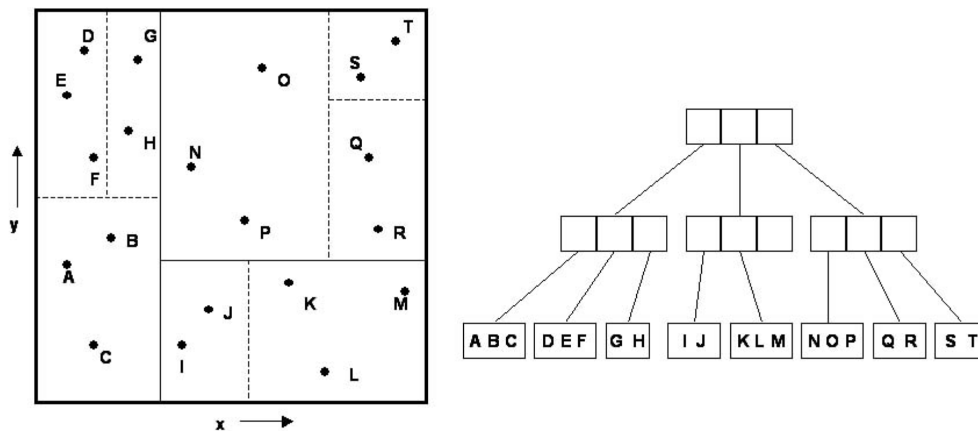


Figura 2.4 Ejemplo de un k-d-B-tree

2.3.4 R-trees y sus variantes

Los R-trees[8] son estructuras de datos jerárquicas, utilizados para el indexado eficiente de objetos multidimensionales con extensión espacial. Los R-trees son usados para almacenar, en lugar del objeto espacial original, su *Rectángulo de Mínimo Acotamiento (MBR)*. El MBR de un objeto n-dimensional es definido como el rectángulo mínimo de dimensión-n que contiene el objeto original. Al igual que los B-trees, los R-trees son balanceados y aseguran la eficiente utilización de almacenamiento.

Los R-trees manejan MBRs y no los objetos reales debido a sus características complejas, por esta razón ellos no pueden responder completamente a una consulta, a menos que el objeto en la base de datos sea igual a su MBR. En general, ellos son usados para solucionar eficientemente el

paso de *filtrado* de una consulta, esto es, hallando el objeto cuyo MBR intersecte con el MBR del objeto de consulta.

El R-tree de Guttman [8] es el padre de todas las variantes de R-trees. Cada nodo del R-tree corresponde a una página de disco y a un rectángulo n-dimensional. Cada nodo interno contiene entradas de la forma:

(ref, rect)

Donde *ref* es la dirección del nodo hijo y *rect* es el MBR que cubre todas las entradas que apunta ese nodo. Las hojas contienen entradas de la misma forma, donde *ref* apunta a un objeto en la base de datos y *rect* es el MBR de ese objeto. En la Figura 2.5 se muestra un R-tree para un conjunto de rectángulos de 2 dimensiones.

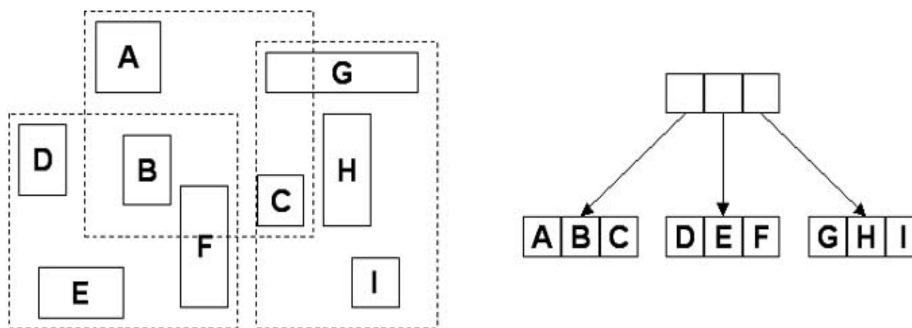


Figura 2.5 Un R-tree para un conjunto de rectángulos

El resto de propiedades del R-tree incluyen las siguientes:

- Sea M el número de entradas que puede contener un nodo, y sea *m* el número mínimo de entradas por nodo, $2 \leq m \leq$ [M/2]. Cada nodo contiene entre *m* y M nodos, a menos que sea la raíz. Si el número de entradas en un nodo cae por debajo de *m* entradas después de un borrado, el nodo es borrado y el resto de las entradas son distribuidas a lo largo de los nodos del mismo nivel.
- La raíz contiene al menos 2 entradas, a menos que sea una hoja.
- El árbol es altamente balanceado; cada hoja tiene la misma distancia desde la raíz. El peso del árbol es a lo más [$\log_m$ N] para n registros indexados.

Las búsquedas en un R-tree son hechas en forma similar al B-tree. Tanto para consultas de punto como regiones, las trayectorias donde *rect* intersecte con el objeto de búsqueda es seguido. A diferencia del B-tree, el R-tree no garantiza que el seguimiento de una sola dirección sea suficiente para encontrar el objeto, esto debido a que los MBRs de las entradas en los mismos niveles pueden solaparse entre ellos. En el peor caso, el algoritmo de búsqueda tendrá que visitar todas las páginas en orden para contestar a la consulta.

La inserción de un objeto en un R-tree, incluye inserción de su MBR al R-tree junto con una referencia del objeto al campo *ref* de la nueva entrada. Solo una ruta hay que seguir para insertar la nueva entrada en una hoja del árbol. Si el MBR del objeto intersecta varias entradas en un nodo intermedio, lo añadimos al nodo hijo donde al insertar el MBR crece menos. El objeto es insertado solo en una hoja y si causa desbordamiento en la página de la hoja, dividimos la página en dos. Esta división puede ser propagada a los nodos antecesores. Si una inserción causa que la página de la hoja crezca en demasía, la ajustamos correctamente y propagamos los cambios hacia arriba.

El borrado en un R-tree requiere un empalme exacto de la consulta con el objeto en primera instancia. Si el objeto es hallado en una hoja, éste es borrado. Nuevamente el borrado puede causar una modificación en la estructura del árbol, pues puede causar que la página de la hoja donde fue borrado el objeto sufra *underflow* (el número de entradas sea menor que *m*). En este caso, el nodo entero es borrado y todas las entradas son almacenadas en un *buffer* temporal, y reinsertados en el árbol. Así como la inserción, la eliminación puede afectar el MBR de la página. En este caso, propagamos el cambio hacia arriba a lo largo del camino de búsqueda.

2.3.1.3 El R^+-Tree

El R^+-tree fue introducido por Sellis [10] como una forma de solucionar el problema de búsquedas ineficientes que se presenta cuando nodos del mismo nivel se intersectan en el R-tree. Como solución directa el R^+-tree propone usar

truncamiento, es decir no hay solapamiento entre nodos intermedios del árbol en un mismo nivel, y objetos que intersectan más de un MBR en un nivel específico son truncados y almacenados en distintas páginas. Como resultado de esta acción, consultas de punto en un R^+-tree requieren recorrer solo una ruta en el árbol. El precio a pagar es el incremento del requisito de almacenamiento del árbol. La Figura 2.6 ilustra un R^+-tree para los rectángulos de la Figura 2.5. Los rectángulos G y F son truncados y almacenados dos veces en el árbol.

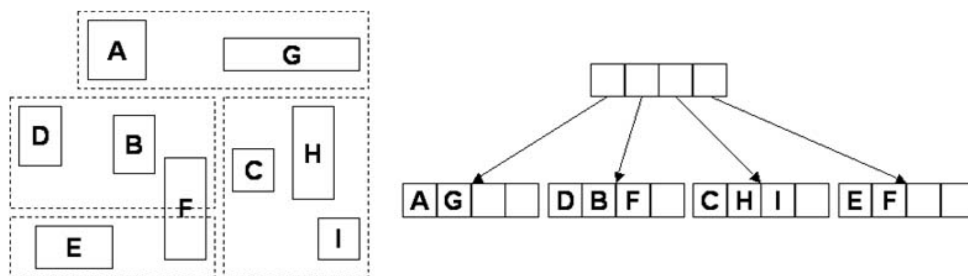


Figura 2.6 Un R+-tree, presentado recorte de 2 objetos

La inserción requiere seguir trayectorias múltiples del árbol, puesto que el objeto a insertar puede intersectar más de un nodo intermedio, y sus partes truncadas deben ser insertadas en distintas hojas bajo esos nodos. Como la inserción de las partes del objeto pueden agrandar el MBR de la página, el algoritmo de inserción tiene que prevenir posibles solapamientos entre páginas hermanas.

La eliminación de un objeto es realizada primero encontrando las páginas que contienen fragmentos del objeto y después removiendo esos fragmentos. Si ocurre *underflow*, tratamos de unir el nodo con sus hermanos. En algunas ocasiones esto no es posible sin la pérdida de la propiedad de ser disjuntos los nodos del árbol, por tanto el R^+-tree no asegura una utilización del almacenamiento mínimo.

2.3.1.4 El R*-Tree

Algunas debilidades de los algoritmos originales de la inserción del R-tree estimularon a Beckmann a trabajar en una versión mejorada del R-tree.

Esta versión (R*-tree) incluye una nueva política de inserción, que mejora significativamente el funcionamiento del árbol. El objetivo principal de esta política es minimizar el solapamiento de regiones en nodos hermanos en el árbol. Una ventaja directa de esto es la minimización de las trayectorias que tiene que seguir una búsqueda para encontrar el objeto en el árbol. Las ventajas de este nuevo algoritmo de inserción pueden resumirse como las siguientes:

- Mientras que recorre la ruta de inserción, el algoritmo de inserción sigue los nodos cuyo MBR tiene el mínimo incremento de solapamiento; así, se mejora el funcionamiento de la búsqueda.
- Siempre que una nueva entrada tenga que ser almacenada en un nodo completo, el nodo no está dividido necesariamente, pero algunas entradas son borradas y reinsertadas en nodos hermanos. Las entradas para reinsertar son elegidas de tal forma que la distancia desde el centro del MBR a éstas sea máxima. Esta característica del algoritmo incrementa la utilización de almacenamiento y mejora la calidad de la partición, haciéndola independiente de la secuencia de inserción.
- El algoritmo para dividir a un nodo es totalmente diferente a su equivalente del R-tree. En primera, el algoritmo decide el eje con respecto al cual se llevará a cabo la división. Entonces, las proyecciones del MBR sobre el eje de división son ordenadas de acuerdo al valor de sus puntos final izquierdo. Esta secuencia puede ser dividida en dos sub-secuencias, en $M-2m+1$ formas. Este algoritmo alcanza una mejor calidad de la partición de los MBR sobre el árbol.

Este panorama nos proporciona las bases teóricas para poder entender los métodos de acceso multidimensional, en particular la familia de R-trees. El R^+-tree brinda características interesantes para la recuperación eficiente de objetos en un solo recorrido. Por esta razón nuestro trabajo esta enfocado a la implementación de algoritmos basados en esta estructura de datos.

CAPÍTULO III

Procesamiento de Consultas Espaciales

Introducción

Como se mencionó en el capítulo anterior, no existe un álgebra espacial estándar o un lenguaje de consulta espacial estándar. Los lenguajes de consulta para bases de datos espaciales dependen en gran medida del dominio de la aplicación, es por esta razón que la estandarización para cubrir todas las consultas espaciales ha fallado. El resultado de una consulta espacial en la base de datos es usualmente un conjunto de objetos espaciales que satisfacen los requerimientos de la consulta.

3.1 Tipos de consultas espaciales

3.1.1 Consultas de Selección Espacial

Los tipos de consultas que pueden ser aplicados a bases de datos espaciales de acuerdo a [2] son:

a) Consultas de Coincidencia Exacta. Este tipo de consulta encuentra todos los objetos de la base de datos que tiene *exactamente la misma extensión espacial* que el objeto *O* de la consulta espacial.

b) Consulta de Punto. Encuentra todos los objetos de la base de datos que *contienen* al punto P de la consulta. En la Figura 3.1 se muestra un ejemplo de este tipo de consulta, dónde el área sombreada sería la respuesta a la consulta del punto que contiene.

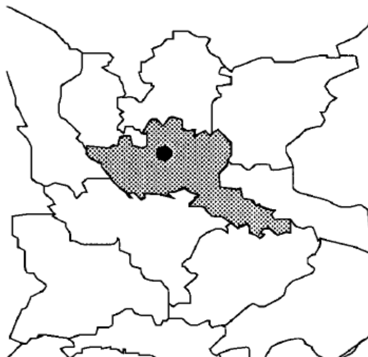


Figura 3.1 Consulta tipo Punto

c) Consulta de Ventana ó Consulta de Rango. Estas consultas recuperan todos los objetos de la base de datos que tienen *al menos un punto en común* con la ventana d-dimensional W de la consulta. Ejemplo de esta consulta se muestra en la Figura 3.2

Figura 3.2 Ejemplo de Consulta de Ventana

d) Consulta de Intersección ó Consulta de Región ó Consulta de Solape. Halla todos los objetos de la base de datos que tienen *al menos un punto en común* con el objeto O de la consulta. Tenemos como ejemplos la Figura 3.3 donde a diferencia de la Figura 3.2 la consulta no es solo un rectángulo sino una región.

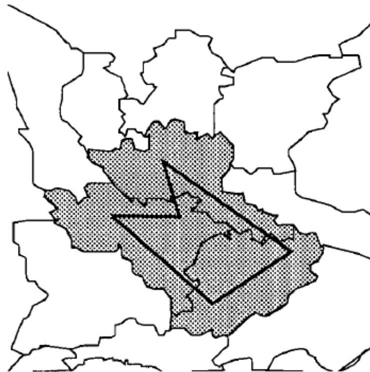


Figura 3.3 Ejemplo de Consulta de Región

e) Consulta de Encierro. Encuentra todos los objetos de la base de datos que *incluyen* un objeto de consulta *O.* Un objeto *a* se dice que es incluido por un objeto *b* si y sólo si cualquier punto de *a* es parte del objeto *b*. La Figura 3.4 muestra un ejemplo de este tipo de consulta.

Figura 3.4 Ejemplo de Consulta de Encierro

f) Consulta de Contención. Estas consultas encuentran todos los objetos de la base de datos que *son encerrados* por un objeto de consulta *O*. Un ejemplo de este tipo de consulta se muestra en la Figura 3.5

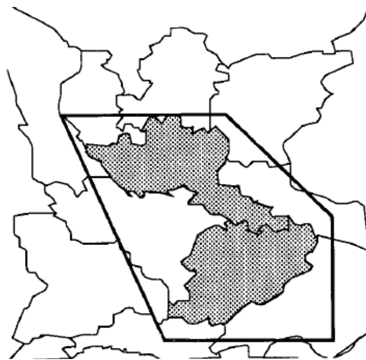


Figura 3.5 Consulta de Contención

g) Consulta de Adyacencia. Recupera todos los objetos de la base de datos que *son adyacentes* al objeto de consulta *O*. Dos objetos se dicen adyacentes si ellos tienen límites comunes, pero uno no encierra al otro. La Figura 3.6 ilustra un ejemplo de consulta de adyacencia donde la respuesta son todas las regiones que tienen un límite en común.

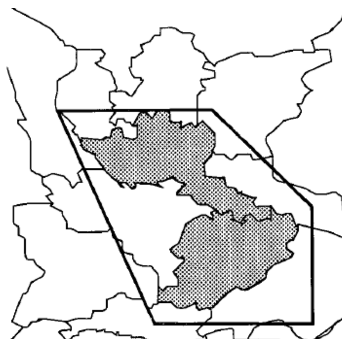


Figura 3.6 Consulta de Adyacencia

h) Consulta de Vecino más Cercano. Encuentra todos los objetos de la base de datos que *tienen una distancia mínima* al objeto de consulta *O*. La distancia entre los objetos espaciales es generalmente definida como la distancia entre sus puntos más cercanos.

3.1.2 Junta Espacial (*Join)*

Además de las consultas de selección, donde un conjunto de objetos que satisfacen el criterio de la consulta son seleccionados de una base de datos relacional, el *join* espacial es una operación común importante en las bases de datos espaciales. Una relación θ-join de dos relaciones R_1 y R_2 sobre la columna i $\in$ R_1, j $\in$ R_2, es llamada *junta espacial* si la i-ésima columna de la relación R_1 y la j-ésima columna de R_2 son atributos espaciales y θ es un predicado espacial.

La operación de join más común es el *join de intersección*, donde θ es el operador de intersección. Brinkhoff en [15] define el *MBR-spatial-join* como un paso intermedio de filtrado para calcular la intersección del join entre dos relaciones. El *MBR-spatial-join* es la intersección del rectángulo mínimo de dimensión *d* (Rectángulo de Mínimo Acotamiento) que contiene los objetos de la junta. La idea principal sobre la cual trabaja este proceso es que si los MBRs de dos objetos no se intersectan, sus objetos exactos tampoco lo harán. Esta propiedad puede ser usada para eliminar pares de objetos que no pertenecen a la junta espacial, simplemente comprobando la intersección de sus MBRs. Si se asume que objetos espaciales en ambas columnas de la relación de la junta son organizadas en R^*-trees, éstas sugieren algunas técnicas heurísticas que realizan eficientemente el *MBR-spatial-join* en términos de CPU y tiempo de Entrada/Salida.

La Figura 3.7 muestra la forma de realizar un join de intersección de dos relaciones en tres pasos. Primero, se aplican algoritmos de [15] para calcular el MBR-join de las relaciones. El siguiente paso del join es aplicar nuevamente los filtros geométricos baratos que identifican pares de objetos que no se intersectan definitivamente (*false hit*) y pares de objetos que si lo hacen (*hits*),

disminuyendo el número de parejas candidatas que necesitarán un procesamiento posterior. Para identificar los *false hits*, se aplica una aproximación convexa más precisa para cada objeto en un conjunto candidato, y una nueva prueba de intersección es aplicada. Los *hits* son filtrados comprobando la intersección sobre el Máximo Rectángulo de Acotameinto (MBR) con los objetos en cada pareja candidata. El MBR de un objeto es definido como el máximo rectángulo d-dimensional que puede ser incluido en la extensión del objeto espacial. Si los MBRs de dos objetos intersectan entonces los objetos también lo harán. Después del segundo paso del proceso, las restantes parejas candidatas tendrán que ser procesadas usando un algoritmo geométrico más costoso computacionalmente (*plane sweep*) que se aplicará sobre la extensión espacial exacta del objeto.

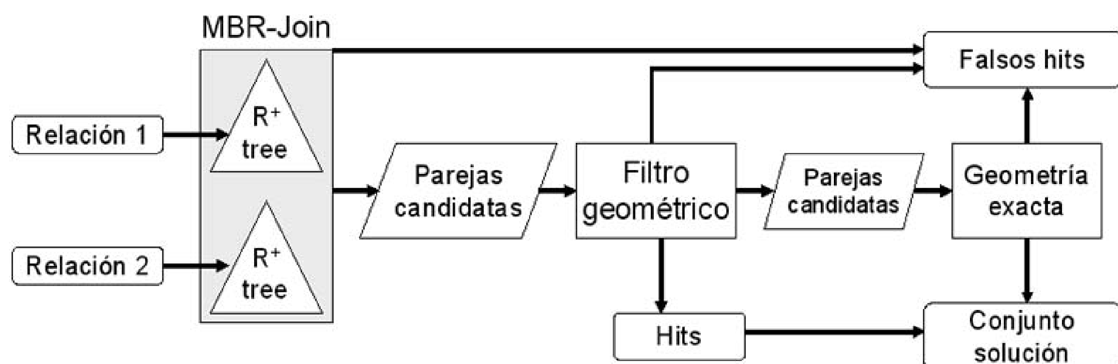


Figura 3.7 Proceso de Join Espacial

3.2 Semijoin espacial

En un principio el operador semijoin fue propuesto para reducir el costo de transmisión en bases de datos distribuidas [16] ya que el join es la operación más costosa en bases de datos. El operador θ-semijoin de una relación R con otra relación S en la condición de join $R._A\ \theta\ S._B$ (A y B representan los campos a comparar en la condición de join) es la relación $R' \subseteq R$ tal que todos los registros de R´ satisfacen la condición del join, donde θ es un operador de comparación escalar (<, >, <=, >=, =). El join de R y S, que están localizados en R_A y S_B respectivamente, se puede realizar utilizando un semijoin de 3 pasos:

a) S es proyectada sobre el atributo del join en S_B y el conjunto resultante de valores distintos S´, es transmitido a R_A.

b) El semijoin de R y S´ se realiza en R_A para producir R´, el cual es enviado a S_B.

c) El join de R´ y S´ es hecho en S_B para producir el resultado del join.

En la figura 3.8 se muestra el resultado de un semijoin aplicado a dos tablas (S = Employee, R = Department) para la siguiente consulta: R_{Loc_MBR} θ S_{Loc_MBR} AND DeptName = 'Sales' (Recuperar todos los empleados que trabajen en el departamento *SALES*). La ubicación de los departamentos está representada por el atributo Loc_MBR de las tablas Employee y Department.

Employee			
Emp_id	Name	DeptName	Loc_MBR
3415	Harry	Finance	MBR_1
2241	Sally	Sales	MBR_2
5134	Mary	Production	MBR_3
3401	George	Finance	MBR_1
2022	James	Sales	MBR_2

Department		
DeptName	Manager	Loc_MBR
Finance	Richard	MBR_1
Sales	Harriet	MBR_2
Production	Charles	MBR_3

Employee θ Department			
Emp_id	Name	DeptName	Loc_MBR
2241	Rally	Sales	MBR_2
2022	James	Sales	MBR_2

Fig 3.8 Semijoin R_{Loc_MBR} θ S_{Loc_MBR} AND DeptName = 'Sales'

El semijoin de las tablas anteriormente mostradas se realiza de la siguiente forma. La relación S es proyectada sobre los atributos del join: DeptName y Loc_MBR. Tenemos dos condiciones: primero que el departamento sea 'SALES' y segundo que la ubicación (Loc_MBR) sea la misma en ambas tablas. El conjunto de elementos distintos forma S´, el cual ahora será comparado con la relación R. El segundo paso realiza la comparación de S´ con la relación R, dicha comparación nos dará una relación con muy pocos datos que formarán R´. Finalmente R´ es comparada con la relación S, donde solo recuperaremos aquellas tuplas que coincidan con la pequeña relación R´.

CAPÍTULO IV

Metodología Propuesta para Acceso y Procesamiento de Datos Espaciales

El R^+-Tree es una extensión directa del B-tree en k-dimensiones. La estructura de dato es un árbol altamente balanceado que consiste de nodos internos y hojas. Los objetos son almacenados en los nodos hojas y los nodos intermedios son construidos por rectángulos que agrupan estos datos en el nivel inferior.

En este capítulo se hace un análisis del R^+-tree propuesto en 1987 por Sellis [10] cuya estructura de datos da soporte a la inserción, eliminación y consultas espaciales. Además se presentan algunas modificaciones realizadas en el algoritmo de empaquetamiento así como en el procesamiento de consultas para poder incorporar un join a través de un semi-join.

4.1 Características del R^+-tree

Como mencionamos anteriormente un R^+-tree consiste de nodos intermedios y hojas, los cuales tienen distintas formas. Un *nodo hoja* es de la forma:

(oid, RECT)

Donde *oid* es un identificador del objeto y es usado para referenciar a un objeto en la base de datos, *RECT* es usado para describir los límites de los objetos y en nuestro caso particular utilizaremos el Rectángulo de Mínimo Acotamiento (MBR). Un *nodo intermedio* es de la forma:

(p, RECT)

En este caso *p* es un apuntador a un nodo de nivel inferior del árbol y *RECT* será el MBR que cubra completamente a los objetos que haga referencia.

Dadas las características de los nodos que componen al árbol, describimos las siguientes propiedades de los R^+-tree:

a) Para cada entrada (*p, RECT*) en un nodo intermedio, el subárbol con raíz en el nodo apuntado por *p* contiene un rectángulo *R* si y sólo si *R* es cubierto por *RECT*. La única excepción es cuando *R* es un rectángulo en un nodo hoja; en este caso *R* debe ajustarse con *RECT.*

b) Para cualesquiera dos entradas (p_1, $RECT_1$) y (p_2, $RECT_2$) de un nodo intermedio, el traslape entre $RECT_1$ y $RECT_2$ es cero.

c) La raíz tiene al menos dos hijos a menos que sea una hoja.

d) Todas las hojas del árbol están al mismo nivel (altura).

La figura 4.1 a) muestra las relaciones entre objetos así como su agrupamiento y la figura 4.1 b) representa la estructura de un R^+-tree y el contenido de sus nodos.

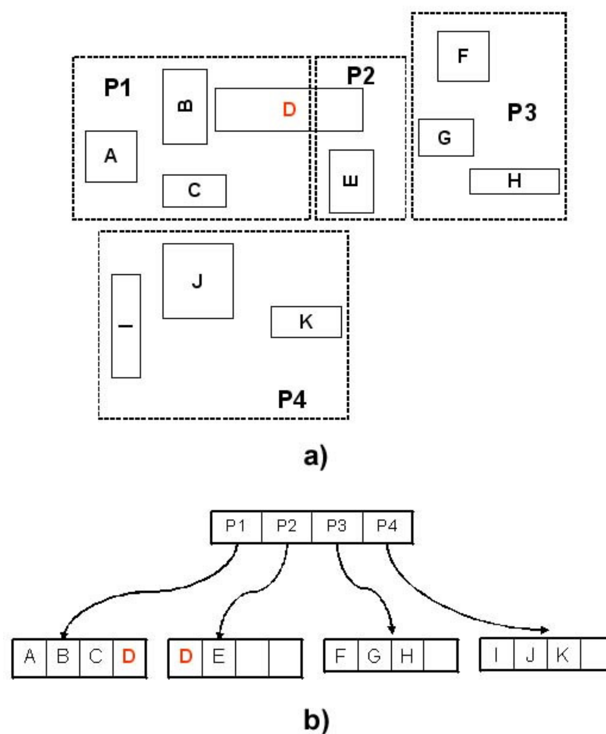


Figura 4.1 Agrupamiento y representación de un R^+-tree

En la figura 4.2 se muestra la metodología propuesta para el acceso y procesamiento de datos espaciales, el diagrama consistente de secuencias de bloques que se ejecutarán de acuerdo a la operación realizada.

Como primera acción tenemos la creación de una Base de Datos Espacial, para esto tenemos dos opciones: a través de un empaquetamiento o mediante la inserción de dato a dato, una vez que la base contiene elementos, podemos realizar eliminaciones de objetos, realizar consultas de tipo point o window query, y finalmente podemos realizar un join o semijoin tomando los árboles R^+ de las relaciones involucradas.

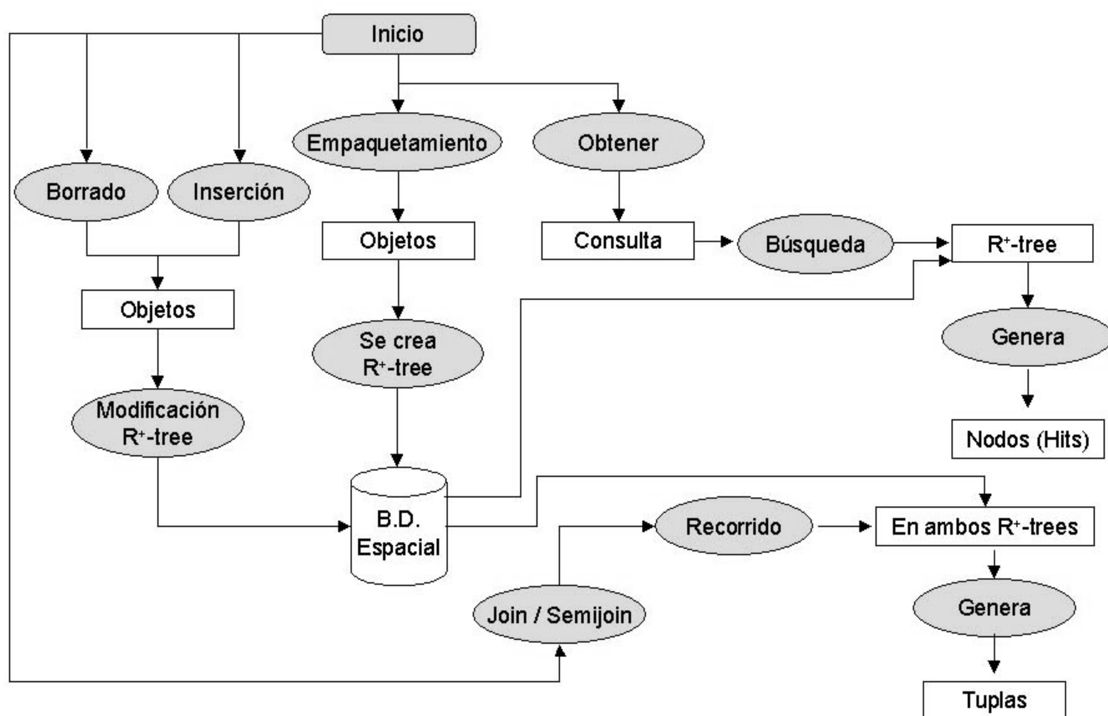


Figura 4.2 Metodología propuesta para acceso y procesamiento de datos espaciales

En las siguientes secciones se describen los algoritmos básicos empleados para la implementación de dicha metodología.

4.2 Búsqueda en el R^+-tree

El algoritmo de búsqueda es similar al usado en los R-trees. La idea es descomponer el espacio de búsqueda en sub-regiones disjuntas y por cada una de éstas descender por el árbol hasta que el dato sea o no encontrado en las hojas. Una diferencia importante con los R-trees es que en éstos las sub-regiones pueden solaparse, lo que llevaría a búsquedas más costosas pero debido a la característica de no traslape entre regiones del mismo nivel evitamos dicho trabajo extra.

A partir de esta sección denotaremos una entrada de la forma (p, RECT) por E, la referencia a RECT por EMBR y la referencia a un hijo o a una tupla por EP.

Algoritmo de Búsqueda (Window Query)

Como resultado encontrará todos los objetos que intersecten con la window query W, iniciando la búsqueda en el nodo R de un R^+-tree.

B1 [Búsqueda en Nodos Intermedios]

Si R no es una hoja, entonces para cada entrada E de R verificar si EMBR traslapa W. En este caso ejecutar Búsqueda de ($W \cap EMBR$) iniciando en el subárbol con raíz EP.

B2 [Búsqueda en Nodos Hojas]

Si R es una hoja, comprobar todos los objetos EMBR en R y regresar aquellos que intersecten con W.

4.3 Inserción en el R^+-tree

La inserción de un nuevo rectángulo en un R^+-tree es hecha buscando en el árbol y agregando el rectángulo en los nodos hojas. La diferencia con el algoritmo correspondiente a los R-trees es que el rectángulo de entrada puede ser agregado *a más de* un nodo hoja. Finalmente cuando se presente desbordamiento de la capacidad del nodo hoja, éste se dividirá y los efectos se propagarán hacia el padre y de éste a sus hijos. Esta última actualización se

debe a que al dividir al padre puede causar una división del espacio lo cual afectará a sus nodos hijos. El algoritmo de Inserción se muestra a continuación.

Algoritmo de Inserción

Dado un R^+-tree con raíz en el nodo R y un dato de entrada IR, el algoritmo encuentra donde IR debe ir y lo agrega al nodo hoja correspondiente.

I1 [Búsqueda en Nodos Intermedios]

Si R no es una hoja, entonces para cada entrada E de R verificar si EMBR solapa IR. En este caso invocar Insertar IR en el nodo apuntado por EP.

I2 [Inserción en el Nodo Hoja]

Si R es una hoja, agregar IR en R. Si después que el Rectángulo es insertado en R y éste tiene más entradas que el límite, llamar al método SplitNodo(R) para reorganizar el árbol.

4.4 Borrado en el R^+-tree

El borrado de un rectángulo en un R^+-tree es realizado en primer lugar localizando el rectángulo que debe ser eliminado después removiéndolo del nodo o nodos hojas. La razón por la cual más de un rectángulo podría ser borrado de los nodos hojas es que la rutina de *Inserción* puede introducir más de una copia para un rectángulo ya insertado. El siguiente algoritmo muestra la forma en que una entrada es borrada del árbol.

Algoritmo de Borrado

Como entrada recibimos un R^+-tree con raíz R y un MBR IR que será borrado

BR1 [Búsqueda en Nodos Intermedios]

Si R no es hoja entonces para cada entrada E de R probar si EMBR se solapa con IR. De ser así Borrar IR del subárbol con raíz en EP.

BR2 [Borrar elemento del Nodo Hoja]

Si R es una hoja, elimina IR de R (si es que se encuentra) y ajusta el MBR que contiene a los MBR´s restantes.

4.5 División de Nodos en el R^{+}-tree

Cuando un nodo desborda la capacidad del nodo necesitamos de un algoritmo que produzca dos nuevos nodos. Dado que requerimos que los dos sub-nodos cubran el espacio y además sean áreas disjuntas, lo primero que tenemos que analizar es una "buena" partición (vertical u horizontal) que descomponga el espacio en dos regiones.

Algoritmo Partición

Dado un conjunto S de MBR´s y el *ff* (factor de relleno) de la primera región, el algoritmo nos regresará un nodo R conteniendo los MBR´s de la primera sub-región y un conjunto S´ con los restantes elementos.

P1 [No se requiere partición]

Si la lista S tiene igual o menos elementos que *ff* entonces no se realiza descomposición del espacio y un nodo R que contiene estos elementos es creado y el algoritmo termina.

P2 [Calcular coordenadas X e Y mínimas]

Sea Ox y Oy las coordenadas *x* e *y* mínimas de los MBR contenidos en la lista.

P3 [Barrido a lo largo del eje X]

(Cx, x_cut) = Barrido("x", Ox, ff), donde Cx es el costo de dividir el espacio a lo largo del eje X.

P4 [Barrido a lo largo del eje Y]

(Cy, y_cut) = Barrido("y", Oy, ff), donde Cy es el costo de dividir el espacio a lo largo del eje Y.

P5 [Escoger el punto de partición]

Seleccionar el eje que regresó el costo más pequeño de Cx y Cy, dividir el espacio y distribuir los rectángulos y sus divisiones. Crear un nodo R que almacene todos los MBR´s de la primera sub-región. Asignemos a S´ los MBR´s faltantes y regresar (R, S´).

Podemos notar que este algoritmo utiliza el procedimiento *Barrido*, el cual es usado para explorar los MBR´s e identificar puntos donde es posible particionar el espacio, dicho algoritmo se muestra enseguida.

Algoritmo Barrido

Recibe el eje sobre el cual se realizará el barrido, el punto donde iniciará y el número de objetos a agrupar, como salida regresamos el costo de agrupar los primeros *ff* MBR así como la coordenada más grande sobre el eje elegido.

BRD1 [Encontrar los primeros ff MBR´s]

Empezando de Oxy, selecciona los primeros ff MBR´s de la lista ordenada.

BRD2 [Evaluación de la selección]

Calcular el costo total de la característica con la cual se está midiendo la organización de los MBR´s (vecino más cercano, mínima cobertura, entre otras). Regresar el costo y la coordenada X o Y más grande de los ff MBR´s elegidos (En nuestro caso costo es la suma de las distancias entre los objetos).

Notemos que al realizar el proceso de división podría ser necesario la propagación hacia abajo. Una vez descritos los algoritmos de Partición y Barrido que serán usados para la división del espacio se describe enseguida el algoritmo de *Split*.

Algoritmo SplitNodo

Dado un nodo R, el algoritmo encuentra una partición para el nodo a dividir, creando dos nuevos nodos, y si es necesario, propaga la división hacia arriba o abajo según corresponda.

SN1 [Encuentra una partición]

Particiona R usando la rutina *Partición* descrita anteriormente. Sean RECT el MBR y P el apuntador asociado al nodo R. También sea S_1 y S_2 las dos regiones resultantes después de la partición, crear dos nodos $N_1 = (p_1, RECT_1)$ y $N2 = (p_2, RECT_2)$ que serán los nodos resultantes de aplicar el algoritmo a R, donde $RECT_i = RECT \cap S_i$, para i = 1,2

SN2 [Poblar los nuevos nodos]

Poner en Ni todos los nodos E de R cuyo EMBR este cubierto totalmente por $RECT_i$ (i=1,2). Para aquellas entradas que solapan ambas sub-regiones:

a) Si R es un nodo hoja, poner E en ambos nodos
b) De otra forma, usar SplitNodo para recursivamente dividir los nodos hijos a lo largo de la partición. Sea (p_{k1}, $RECT_{k1}$) y (p_{k2}, $RECT_{k2}$) los dos nodos después de dividir (p_k, $RECT_k$) donde $RECT_{k1}$ cabe completamente en $RECT_i$ (i=1,2). Agregar estos dos nodos al correspondiente nodo Ni.

SN3 [Propagación de la división hacia arriba]

Si R es la raíz, crear una nueva raíz con solo dos hijos N1 y N2. De otra forma crear un nodo PR que represente el padre del nodo R, reemplazar R en PR con N1 y N2. Si PR tiene más entradas que las permitidas, invocar a SplitNodo (PR).

4.6 Algoritmo de Empaquetamiento

Una operación importante en los R^+-trees es el empaquetamiento inicial de un árbol. Esto es especialmente útil cuando tenemos un archivo con datos (MBR´s) y un sistema es requerido para construir un R^+-tree en base a éste.

El algoritmo *Empaquetamiento* es descrito como una extensión de [19], modificado para aceptar cualquier criterio de agrupación mencionados anteriormente. El *fill-factor* determina que tan poblado estarán los nodos del árbol. Mientas mejor empaquetado este el árbol, más rápidas serán las búsquedas. Por tanto, si la base de datos espacial es relativamente estática (no sufre muchos cambios) es altamente deseable empaquetar el árbol a su máxima capacidad.

Algoritmo Empaquetamiento

Dado un conjunto S de MBR´s y un factor de relleno ff para el árbol, el procedimiento recursivamente empaca las entradas de cada nivel del árbol.

E1 [No es necesario empacar]

Si la cardinalidad de S es menor o igual al factor de relleno ff, construir la raíz R del R^+-tree y regresarlo.

E2 [Inicialización]

Sea AN=*empty*, AN contiene el conjunto de MBR´s del siguiente nivel a ser empaquetados.

E3 [Particionar el espacio]

(R, S´) = *Partición(S, ff)*. Si estamos dividiendo nodos internos y algunos de los MBR´s tienen que ser divididos debido a la partición elegida, recursivamente propagar los cambios hacia arriba y en caso de ser necesario también realizar los cambios hacia abajo. AN = Añadir (AN, R).

Continuar con el paso E3 hasta que la lista S´ este vacía.

E4 [Recursivamente Empacar los Nodos Intermedios]

Regresar *Empaquetamiento (An, ff).*

4.7 Join Espacial

El Join Espacial (Junta) es una de las más importantes operaciones para combinar objetos espaciales entre varias relaciones. Los métodos de acceso espacial son los encargados de permitir realizar las juntas espaciales, entra las principales razones tenemos:

- En aplicaciones espaciales, asumimos que casi siempre un índice espacial existe en una relación espacial. Por tanto, es interesante aprovechar estos índices para realizar comparaciones entre estás relaciones.
- Muchos algoritmos eficientes diseñados para *Joins* naturales, como algunos basados en tablas hash, no pueden ser aplicados a relaciones espaciales, por lo cual los métodos de acceso espacial ofrecen una oportunidad para investigar y realizar estudios que permitan relacionarlas.
- Dado que los métodos de acceso espacial utilizan rectángulos de mínimo acotamiento como parámetro para representar los objetos, la junta espacial requerirá casi siempre acceso a objetos en una ventana dada.

4.7.1 Definición y tipos de Junta Espacial

Dados dos relaciones espaciales $A = \{a_1,.., a_n\}$ y $B = \{b_1,.., b_m\}$, la relación θ-junta espacial de A con B es el conjunto de objetos espaciales $C = \{c_1, .., c_p\}$ tal que C_i es la unión de una tupla a_i con una tupla b_j que cumplen la condición a_i *oper* b_j, dónde específicamente en nuestro caso *oper* es el operador de traslape.

Sean A y B dos relaciones espaciales indexadas y Id una función que asigna un identificador único a cada objeto espacial en una base de datos espacial, los más importantes tipos de junta espacial de acuerdo a [15] son los siguientes:

- *MBR-spatial-join*: compara todas las parejas $Id(a_i)$, $Id(b_j)$ cuya intersección sea distinta del vacío ($MBR(a_i) \cap MBR(b_j) \neq \emptyset$).
- *ID-spatial-join*: compara todas las parejas $Id(a_i)$, $Id(b_j)$ cuyas características originales se traslapan ($a_i \cap b_j \neq \emptyset$).
- *Object-spatial-join*: compara los objetos $a_i \cap b_j$ que cumplan la característica $a_i \cap b_j \neq \emptyset$.

En nuestro caso particular, implementaremos el algoritmo que ocupa el MBR de los objetos, ya que tiene menor costo computacional y obtiene una buena aproximación sin la necesidad de manejar todos los atributos espaciales del objeto a priori.

Algoritmo SpatialJoin

Dados 2 nodos (R, S) correspondientes a 2 árboles.

Para todas las entradas E_S del nodo S hacer

    Para todas las entradas E_R del nodo R, tal que ($E_S MBR \cap E_R MBR \neq \emptyset$)

        Si (R es Nodo Hoja) entonces

            Reporta (E_R, E_S)

        Sino

            Leer _ página ($E_R P$), Leer _ página ($E_S P$)

            SpatialJoin ($E_R P$, $E_S P$)

    Fin del For

Fin del For

Cuando en el algoritmo anterior mencionamos **Reporta (E_R, E_S),** nos referimos a que en este punto el algoritmo estará en condiciones de responder al join a través del recuperar las tuplas apuntadas por E_R y E_S.

El algoritmo anterior está diseñado para árboles que tienen la misma altura, es decir el algoritmo se basa en el hecho de que si los MBR´s de dos entradas de directorios E_R y E_S no se intersectan, no existirá una pareja (E_RMBR, E_SMBR) de datos que se intersecten en el subárbol inferior apuntado por E_RP y E_SP. Por el contrario, si intersectan se llama recursivamente el algoritmo con los subárboles correspondientes hasta llegar al nivel de las hojas y en este caso comparar que datos se solapan y reportar estos datos.

Cuando los árboles no sean de la misma altura, supongamos que la altura de R es mayor que S. Sea N_R un nodo directorio (intermedio) del árbol R, N_S el nodo de datos (hoja) del árbol S y consideremos la junta espacial entre el nodo N_R y N_S.

La idea de este caso es realizar consultas de ventana en los subárboles con raíz en N_S usando como dato de consulta a los MBR´s del nodo N_R. Primero, comparamos todos los pares (E_R, E_S) que se intersectan. La forma de considerar las consultas de ventana se realizarán de la siguiente forma: para cada entrada E_S, realizamos una consulta de ventana sobre E_RP siempre que $E_R MBR \cap E_S MBR \neq \emptyset$ (es decir se intersectan).

4.8 Semijoin Espacial

El semi-join es un operador relacional que reduce el costo de esfuerzo requerido para el procesamiento en las consultas que envuelven operaciones binarias [19]. Esto es logrado mediante la previa selección de los datos relevantes para contestar la consulta y de esta forma reduciendo el tamaño de la operación de las relaciones involucradas.

El operador *semi-join* toma la junta de dos relaciones espaciales, R y S, y después proyecta los datos correspondiente al dominio de la relación R. Esto

es, recupera aquellas tuplas en R que hacen *join* con algunas tuplas en S. De forma alterna, podemos ver al semi-join como una generalización de la restricción, es decir restringe a R por valores que aparecen en el dominio de la Junta de S.

En nuestro caso para la implementación del semijoin utilizaremos el algoritmo del join natural con la siguiente variante: en la condición para saber si es un nodo hoja, en caso afirmativo, no se reportará la tupla con que contiene ambas relaciones, sino solo la tupla correspondiente a la primera relación así como los campos solicitados, la modificación se muestra en el siguiente algoritmo.

Algoritmo Semijoin Modificado

Dados 2 nodos (R, S) correspondientes a 2 árboles.

Para todas las entradas E_S del nodo S hacer

    Para todas las entradas E_R del nodo R, tal que ($E_S MBR \cap E_R MBR \neq \emptyset$)

        Si (R es Nodo Hoja) entonces

            Reporta_Semijoin (E_R)

        Sino

            Leer _ página ($E_R P$), Leer _ página ($E_S P$)

            SpatialJoin ($E_R P$, $E_S P$)

    Fin del For

Fin del For

La operación **Reporta_Semijoin (E_R)** representa la modificación más importante ya que a diferencia del join, el semijoin solo tendrá que ir a la base de datos espacial por una tupla y no por dos como en el join. Dicha modificación reduce el tiempo de respuesta así como el espacio de almacenamiento.

En el siguiente capítulo describimos la implementación de los algoritmos a través de un lenguaje de programación orientado a objetos. Asimismo mostramos un ejemplo a lo largo de todo el proceso.

CAPÍTULO V

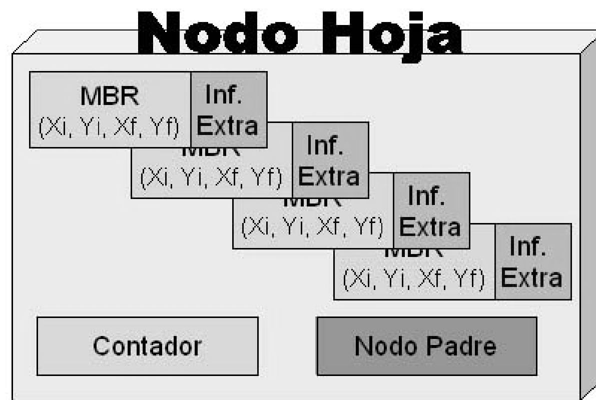
Implementación y Análisis de Resultados

Hasta el capítulo anterior hemos detallado las características que definen un R^+-Tree así como sus algoritmos básicos. En este capítulo definiremos los detalles de implementación así como los principales procedimientos que permiten trabajar un árbol en base a los MBR´s de los objetos a indexar. Al mismo tiempo iremos ilustrando dichos procedimientos con una serie de pruebas (ejemplo clásico de la literatura orientada a mecanismos de indexamiento espacial), que muestren el comportamiento correcto del árbol y sus algoritmos de inserción, búsqueda, borrado así como las implementaciones de join espacial y semijoin espacial.

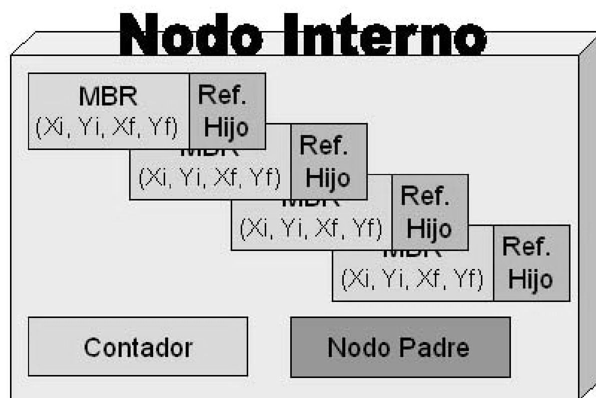
5.1 Diseño del R^+-tree

La implementación del árbol se realizó en un lenguaje de programación orientado a objetos (Java) ya que permite el manejo de referencias a través de instancias por medio de las clases, lo cual hace más fácil el manejo de memoria dinámica que la que utiliza los apuntadores.

El modelo de la estructura de clases que utilizaremos para representar el árbol R^+ se muestra en la Figura 5.1 a) para el nodo hoja y en la 5.2 b) para el nodo interno del árbol. Cabe destacar que este diseño es similar al R-tree dado que no cambia el formato de los nodos, lo que lo hace diferente son el método de *Split* y el empaquetamiento inicial.



a)



b)

Figura 5.1 Estructura de Dato utilizado para el R^+-tree

De forma general el diseño de la implementación del R^+-tree tendrá como componente a la aplicación principal la cual hará el llamado a las distintas clases: RPTree, Nodo_Interno, Nodo_Hoja y Generales. Esta última clase será la encargada de invocar los métodos de *Split*, así como las demás que puedan ser comunes a las demás clases. Por otra parte, RPTree es la clase encargada de inicializar el árbol, insertar, borrar y buscar datos espaciales. Dicho diseño de clases se muestra en la Figura 5.2

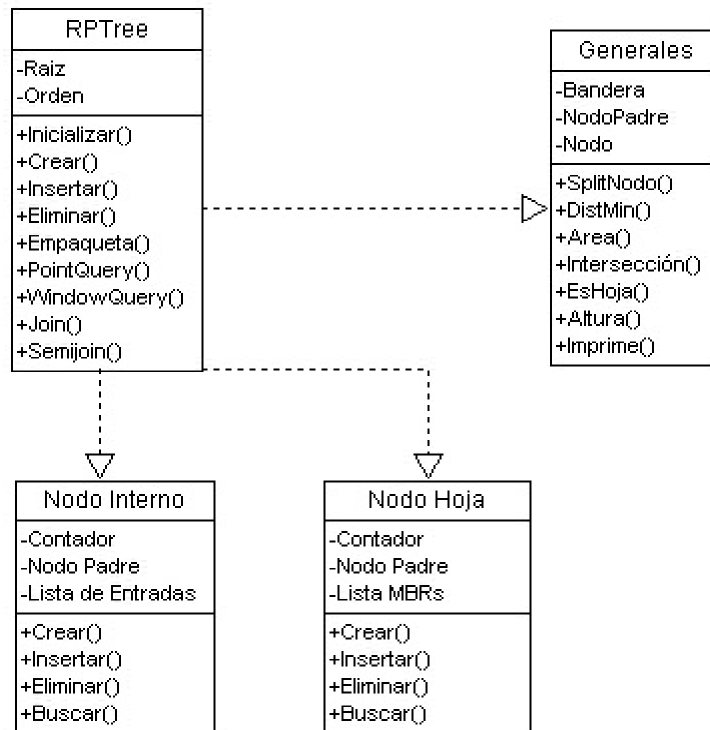


Figura 5.2 Diagrama de clases

5.2 Empaquetado en un R^+-tree

Como mencionamos en el capítulo anterior, el proceso de empaquetamiento es uno de los más importantes ya que en ocasiones necesitaremos, a partir de un archivo de datos, poder construir un árbol que tenga las características necesarias para poder trabajar con el de forma adecuada. El proceso inicia con un conjunto de objetos a indexar, en este caso asignamos estas entradas en las hojas para comenzar a construir el árbol en forma ascendente, es decir primero empaquetamos los objetos y a partir de estos construimos nuevos paquetes que serán auxiliares para el proceso de construcción del árbol R^+.

Nuestro criterio para asignar objetos en un mismo paquete es agrupar aquellas entradas cuya distancia entre sus puntos medios de la arista inferior sea la mínima. Basamos este hecho en la distancia euclidiana, que es la medida más común en este tipo de criterios.

Supongamos que tenemos el caso que se ilustra en la Figura 5.3 donde tenemos 8 objetos a indexar y sea un árbol de orden M (*M = 3*). En esta

situación el primer paso es agrupar en conjuntos los M elementos que tengan distancia mínima entre ellos, no importando si pudieran solaparse los paquetes que contienen a estos objetos.

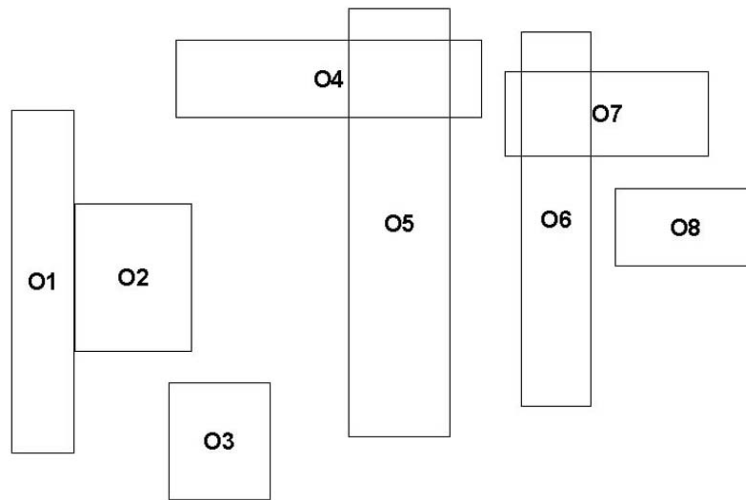


Figura 5.3 Conjunto de objetos a empaquetar

En la Figura 5.4 mostramos el agrupamiento que resultaría de aplicar el algoritmo de *Partición*, sin que esto nos asegure un empaquetamiento lo suficientemente adecuado para un R^+-tree, ya que podría presentar solapamiento entre los paquetes formados (A, B, C). Este problema se solucionará con una rutina que verificará el traslape entre paquetes formados que mostraremos más adelante. Los triángulos en la figura 5.4 representan el cálculo de las distancias mínimas entre los objetos.

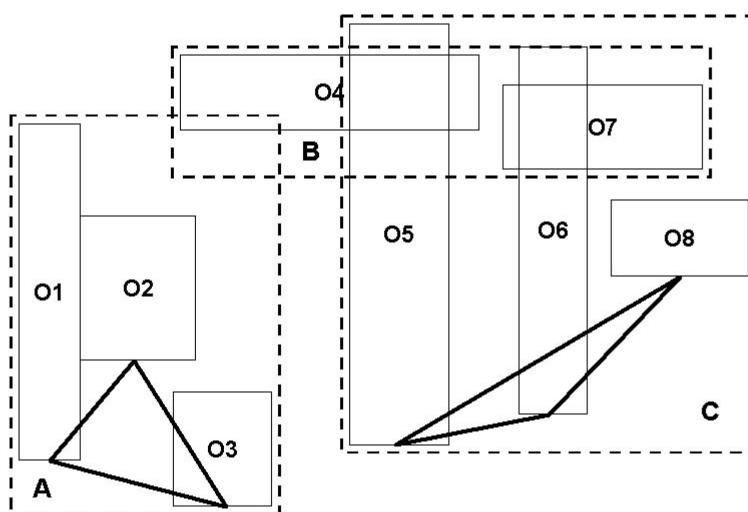


Figura 5.4 Primeros paquetes construidos

El siguiente pseudocódigo muestra parte del procedimiento que se sigue para realizar esta construcción de paquetes. En primer plano si existen menos de M elementos (factor de relleno del nodo) se agrupan en un solo nodo y el algoritmo termina, en caso contrario se procede a calcular las distancias de todas las parejas de M elementos para poder obtener el primer paquete con distancia mínima entre sus elementos. De esta forma se continúa hasta finalizar con todos los elementos o se alcance el primer caso descrito.

```
Public Partición (Vector Lista, int ff) {
  Si (Lista.size() <= ff)       // ff Factor de Relleno
    Crear un nodo con estos elementos
  Sino
    Ordenar Lista en base al eje X
    Obtener la Distancia Mínima (DminX) entre ff elementos
    Ordenar Lista en base al eje Y
    Obtener la Distancia Mínima (DminY) entre ff elementos
    Si (DminX < DminY)     //  Elegimos el ordenamiento de menor costo
      Crear un nodo con los ff elementos del ordenamiento en X
    Sino
      Crear un nodo con los ff elementos del ordenamiento en Y
} Fin de Partición
```

El algoritmo que utilizamos para calcular las distancias mínimas se basa como mencionamos anteriormente en la distancia euclidiana y consiste en encontrar las distancias entre todos los ff (fill factor) elementos consecutivos. De esta forma el algoritmo es de orden lineal por lo cual no representa una carga computacional importante a la aplicación. El pseudocódigo utilizado se muestra en el siguiente procedimiento.

```
Public double DMin (Vector vcopia, int ff) {
  MIN = infinito        //  máxima distancia
  Para cada elemento de la Lista
    Toma los primeros ff elementos de la Lista y calcula Distancia entre ellos
```

```
    //Distancia = √((x2−x1)^2+(y2−y1)^2)
    Si (Distancia < MIN)
      MIN = Distancia
      Guardar el índice del primer elemento del conjunto
  Fin del Para
  Regresa (MIN, índice del primer elemento del conjunto)
} Fin de Dmin
```

Retomando la situación presentada en la Figura 5.4 donde creamos grupos de 3 elementos con posible solapamiento, el siguiente pseudocódigo verificará que dicho traslape se solucione agrandando el área de un paquete y recortando el otro. De esta forma se realizará un ajuste en los paquetes, ya que al agrandar el área de uno de ellos provocará que algún o algunos objetos se vinculen con este paquete. Por el contrario en el paquete que disminuye de área podría presentarse la situación que algún elemento ya no sea cubierto por esta área. Al final del pseudocódigo mostramos la figura 5.5 donde observamos el nuevo acomodo de los paquetes.

```
Public Part_paquetes (Vector Lista, Vector Dirs) {
  F = 0; M = F +1
  Mientras (F < Lista.size()) {
    Si (Hay traslape entre MBR´s de Lista(F) y Lista(M))
      Obtener áreas de los paquetes
      PMP se agranda      // PMP: Paquete Más Pequeño
      PAQ se reduce       // PAQ: Paquete Restante
      Se reacomodan los objetos pertenecientes a cada paquete
    Incrementamos (M)
    Si (M >= Lista.size()) Incrementamos (F) y M = F + 1
} Fin de Part_paquetes
```

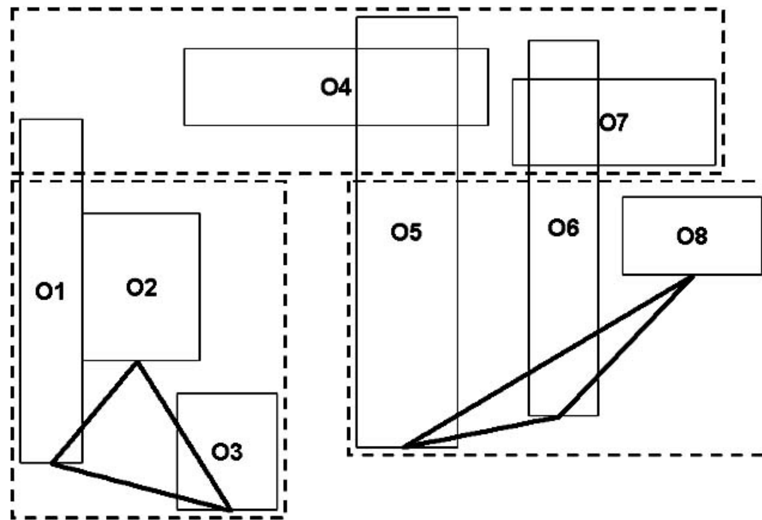


Figura 5.5 Resultado del procedimiento Part_paquetes

Una vez que tenemos paquetes formados y disjuntos podemos pasar a la última parte del proceso de empaquetamiento, lo cual es verificar que los grupos creados no rebasen la capacidad máxima de almacenamiento (llamado en el programa ff). El procedimiento consiste en tomar los primero ff objetos de algún paquete que rebase el límite y dividir este conjunto en dos, así sucesivamente hasta obtener paquetes que a lo más tengan ff elementos. En la figura 5.6 tenemos la organización que genera este proceso.

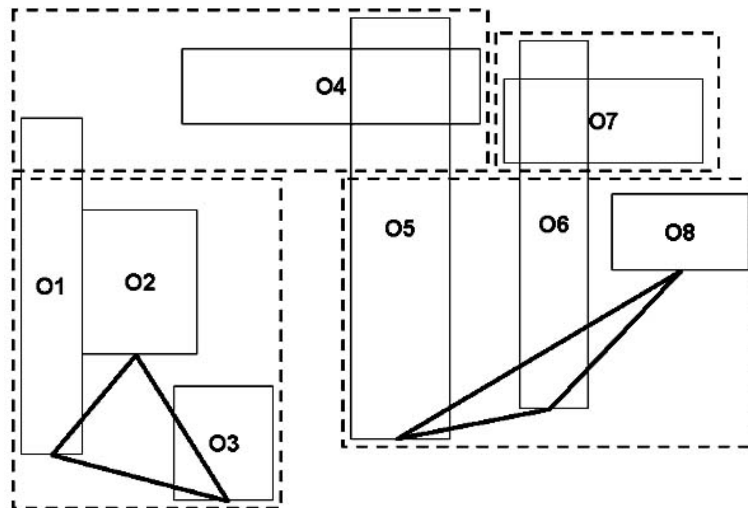


Figura 5.6 Se generan paquetes disjuntos

Finalmente este procedimiento es iterativo y no termina con el resultado mostrado en la Figura 5.6, ya que como se nota los paquetes son disjuntos pero supera la cantidad máxima de paquetes que puede contener un nodo. En este caso aplicamos nuevamente el proceso de Partición y en caso de quedar

paquetes traslapados utilizamos el proceso de Part_paquetes. La figura 5.7 a) muestra el resultado final de la organización de los objetos y los paquetes, la figura 5.7 b) representa el árbol R^+ creado a partir de esta organización.

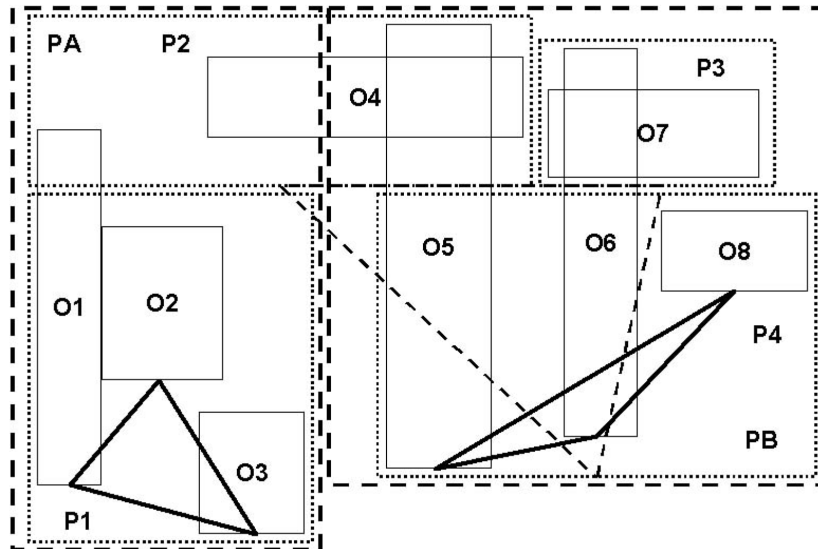


Figura 5.7 a) Organización final de los objetos

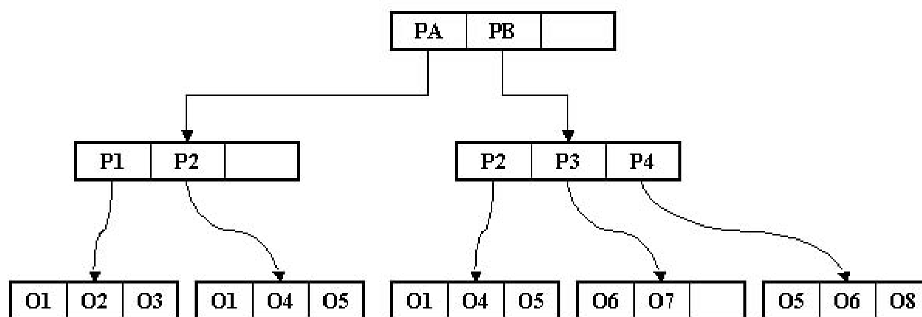


Figura 5.7 b) Árbol R^+ generado a partir de la estructura pasada

5.3 Inserción en un R^+-tree

La inserción en un árbol R^+ se efectúa de forma similar a lo realizado en un árbol R, la diferencia radica en que en este tipo de árboles podemos indexar un objeto en más de un paquete debido a que nuestro propósito es realizar una búsqueda con el recorrido de un solo trayecto.

La inserción de una nueva entrada se realiza colocando ésta en un nodo hoja, dicha ubicación se apoya con el uso de los nodos intermedios que determinará si la entrada se indexa en un solo nodo o requiere el almacenamiento en más de uno. Cuando se inserta en el nodo hoja se verifica si no rebasa el límite de almacenamiento, en caso de hacerlo se llama al procedimiento de SplitNode que se encargará de la división de este nodo. El siguiente pseudocódigo ilustra el procedimiento general de inserción.

```
Public Insertar (Nodo nodo, Entrada dato) {
  Si (No Es Hoja(nodo))    //Estamos en un nodo intermedio
    Para cada entrada E del nodo
      Si (Hay Traslape entre el dato y E)
        Insertar (E.hijo, dato)
  Sino        //Estamos en una hoja
    Nodo.clave[nodo.conta] = dato
    Nodo.conta++
    Si (nodo.conta > orden)      //orden = ff
      SplitNodo (nodo)
} Fin de Insertar
```

Retomando la estructura final mostrada en la figura 5.7 a), realizaremos inserciones que ilustren el comportamiento de las divisiones que se producen así como del crecimiento del árbol. Supongamos que queremos insertar el elemento O9 como se muestra en la figura 5.8, el procedimiento de inserción en primer lugar comprueba en cual de los dos paquetes iniciales debe ser insertado (PA, PB). Dicha elección se basa en el hecho que el objeto O9 intersecta con el paquete PB.

Una vez ya en el paquete PB, y considerando que no es un nodo hoja verificamos en cual de los 3 paquetes (P2, P3, P4) debe ser indexado; en este caso el objeto se insertará en el paquete P3 debido a que es con este paquete con quien intersecta. Finalmente una vez que descendimos hasta el nodo hoja que contiene a los elementos O6 y O7, dado que existe todavía espacio para

almacenar un objeto más la inserción termina y tenemos el árbol que se muestra en la Figura 5.9.

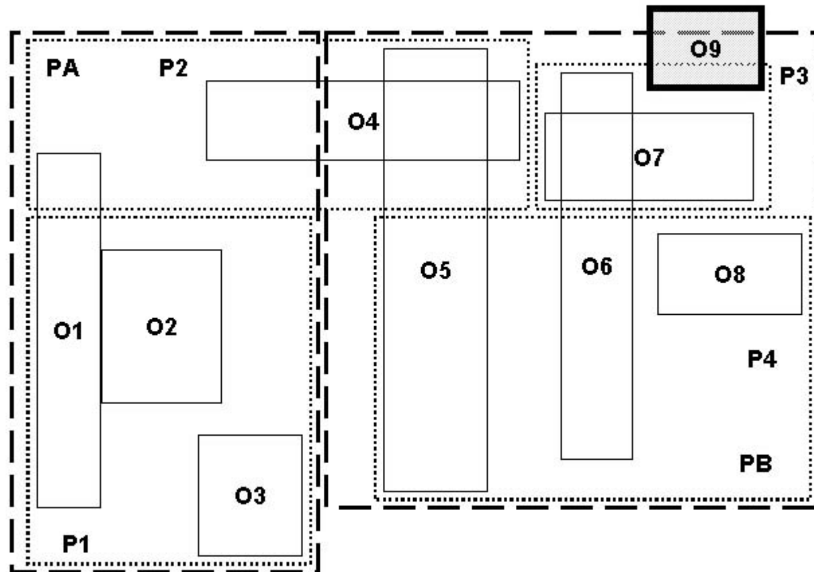


Figura 5.8 Inserción del Objeto O9

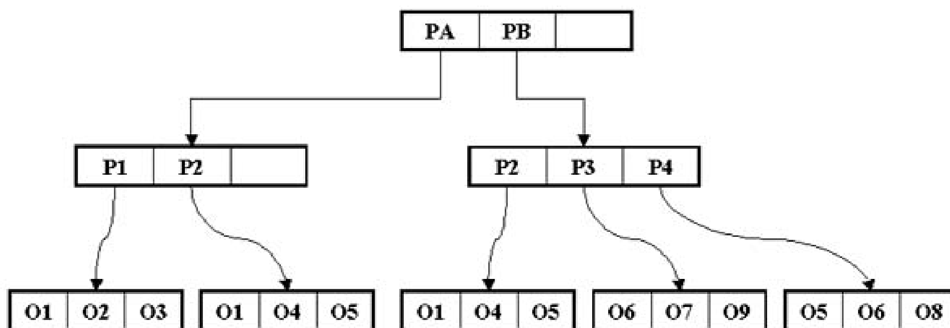


Figura 5.9 Árbol resultante de la inserción del objeto O9

Ahora supongamos que tenemos el caso que se muestra en la Figura 5.10 en dónde se quiere insertar el Objeto O10, en dicha figura se observan los cambios realizados por la inserción pasada, así como la redimensión de los paquetes que sufren cambios por la inserción.

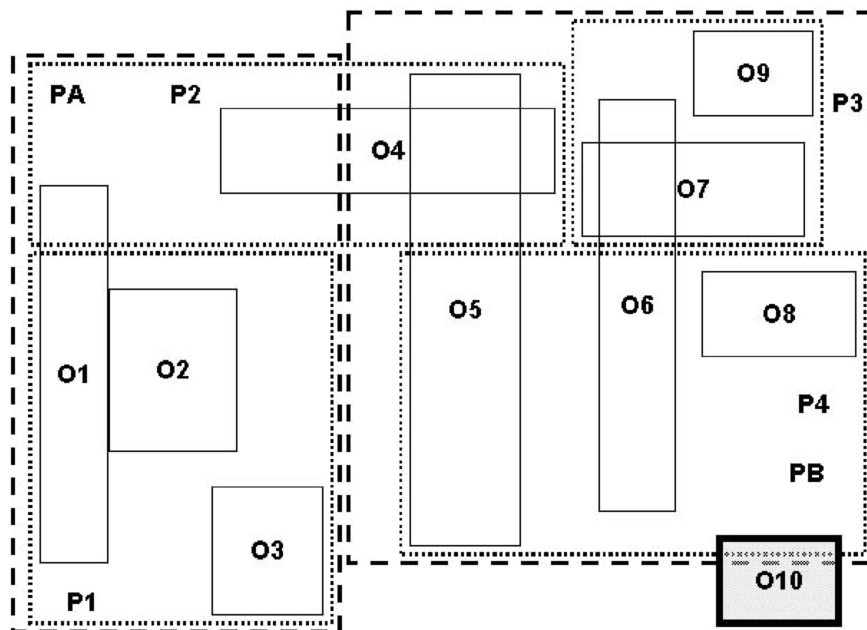


Figura 5.10 Inserción del Objeto O10

Para efectos de la estructura, lo primero que hacemos al tratar de insertar el objeto O10 es recorrer el primer nodo y en este caso descender por el hijo del paquete PB, ya en este nodo verificamos que corresponde insertarlo en el paquete P4; de la misma forma descendemos e insertamos el objeto O10 en el nodo hoja que contiene a los elementos O5, O6 y O8. Dado que nuestro árbol puede almacenar a los más 3 objetos, se manda a llamar al método SplitNodo para este nodo, el pseudocódigo se muestra a continuación.

```
Public SplitNodo (Nodo nodo) {
  Utilizar el método Partición para crear 2 nodos //N1 , N2
  Para cada entrada E del nodo
    Si (E esta cubierto completamente por N1)
      Insertamos E en N1
    Sino Si (E esta cubierto completamente por N2)
      Insertamos E en N2
    Sino Si (EsHoja(R))
      Insertamos E en ambos nodos
    Sino SplitNodo (Nodo.hijo)    //Realizamos los cambios hacia abajo
  Si (EsRaiz(R))
    Raíz = new Nodo()      //Tenemos una nueva raíz
```

```
    Asignamos los dos nodos a la raíz
  Sino
    PR = R.padre
    Asignamos los dos nodos creados a PR
    Si (PR.conta > orden)
      SplitNodo(PR)
} Fin de SplitNodo
```

Dado que el nodo rebasa el límite de almacenamiento se divide en dos nodos y esto a su vez provoca que el nodo padre (Nodo con Objetos P2, P3 y P4) se incremente por la creación de un nuevo paquete (P5). De esta forma este nodo se ve afectado por el método SplitNodo. Finalmente, llegamos a la raíz y dado que se crea un nuevo paquete se almacena en la raíz, pero dado que no rebasa el límite de almacenamiento el procedimiento termina y el Objeto O10 queda indexado, dicha representación del árbol se muestra en la Figura 5.11

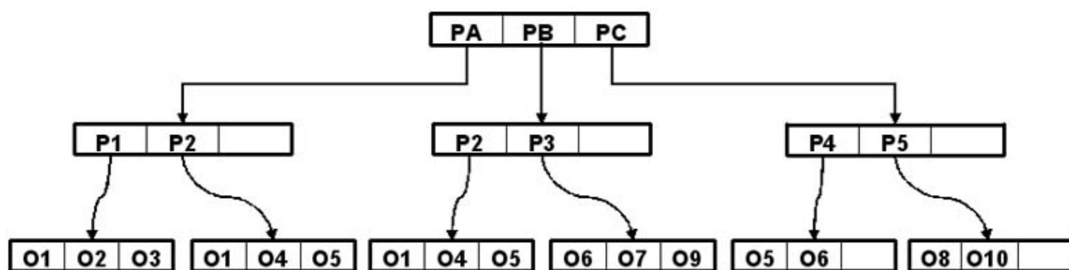


Figura 5.11 Árbol resultante de la inserción del objeto O10

5.4 Borrado en un R+-tree

La eliminación de objetos en un árbol R+-tree se reduce a una búsqueda del objeto a eliminar y una vez encontrado este elemento, borrarlo de los nodos que hacen referencia a él. La dificultad del borrado en este tipo de estructura se presenta cuando la cantidad de objetos que tiene el nodo se ve reducido a menos del límite inferior permitido debido a la omisión de un objeto. El procedimiento se muestra en el siguiente pseudocódigo.

```
Public Eliminar (Nodo raiz, Entrada dato) {
  Si (No EsHoja(raiz))
    Para cada entrada E del nodo
      Si (Overlap(E, dato))
        Eliminar (E.hijo, dato)
  Sino        //Es una hoja
    Si (Encontrado(raiz, dato))
      raiz.conta—
      Ajustamos el MBR de raiz.padre
      Si (Vacio(raíz))
        Borrar nodo y nodo.padre
    Sino
      "No existe el elemento a eliminar"
}  Fin de Eliminar
```

Para nuestra metodología asumiremos que el número de eliminaciones es una operación poco frecuente por tal razón no tendremos un limite inferior, así que cuando realicemos una eliminación el nodo que lo contenga no sufrirá cambio excepto por la redimensión del MBR del nodo padre si es que lo amerita. Cuando el nodo tenga solo un elemento y éste sea borrado el procedimiento será eliminar el nodo y la referencia padre, realizando los cambios necesarios en forma ascendente.

Realicemos algunas eliminaciones tomando como base la figura 5.11. Como primer elemento borremos el objeto O5 el cual se encuentra en los paquetes P2 y P4 que además no afecta la estructuradle árbol. Un segundo elemento a borrar es el objeto O6, el cual se encuentra indexado en los paquetes P3 y P4. En el paquete P3 no existe problema ya que tenemos objetos que permiten mantener intacto dicho nodo, pero en el paquete P4 al eliminar el objeto O5 el nodo queda sin elementos, por esta razón se elimina el nodo así como su referencia padre que es la entrada P4. A su vez el nodo queda solo con un elemento que dada las restricciones anteriores no implica un reacomodo del árbol. El resultado de las eliminaciones de estos objetos se observan en las figuras 5.12 a) y 5.12 b).

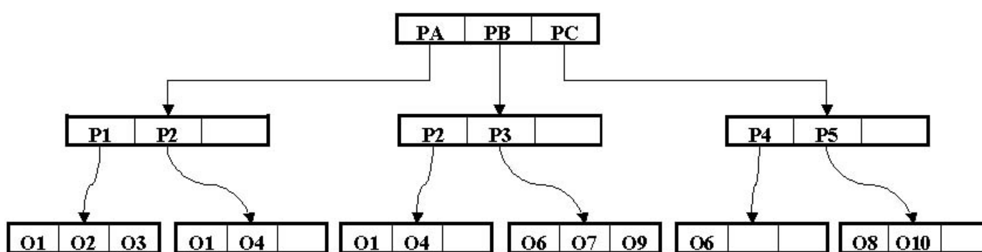


Figura 5.12 a) Árbol resultante de la eliminación del objeto O5

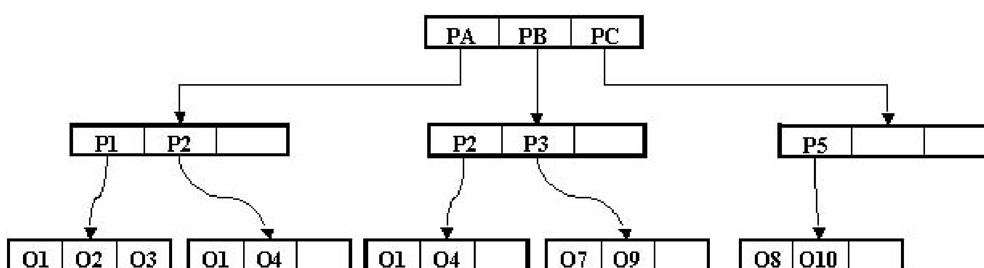


Figura 5.12 b) Árbol resultante de la eliminación del objeto O6

5.5 Búsqueda en un R⁺-tree

Uno de los objetivos de una base de datos espacial es poder dar respuesta a cuestionamientos del tipo topológicos. En este sentido podemos resaltar 2 operaciones básicas como son la consulta de punto (Point Query) y la consulta de ventana (Window Query). El primer tipo de operación consiste únicamente en verificar en que parte de las entradas de los nodos se presenta solape, continuando de esta forma hasta llegar a los nodos hojas. Una vez en esta instancia daremos como resultado aquellas entradas en las cuales si se presente traslape, el pseudocódigo se muestra a continuación.

```
public PointQuery (Nodo raiz, Punto dato) {
  Si (No EsHoja(raiz))
    Para cada entrada E del nodo
      Si (Overlap(E.MBR, dato))
        PointQuery (E.hijo, dato)
  Sino
```

```
    Para cada entrada E del nodo
      Si (Overlap(E.MBR, dato))
        Imprimir "E.MBR intersecta con la Point Query"
} //Fin de PointQuery
```

La segunda operación de consulta en árboles R^+-tree es la Window Query y básicamente en esta se basan las demás operaciones topológicas ya que siguen el mismo procedimiento hasta llegar a los nodos hojas. En éstos cambia la comprobación; es decir si los objetos se solapan parcialmente, si se solapan completamente y algunas otras variaciones. En el siguiente pseudocódigo mostramos la Window Query ya que a partir de ésta las demás consultas topológicas representan un cambio mínimo.

```
public WindowQuery (Nodo raiz, Rect dato) {
  Si (No EsHoja(raiz))
    Para cada entrada E del nodo
      Si (Overlap(E.MBR, dato))
        WindowQuery (E.hijo, Intersección(E.MBR, dato))
  Sino
    Para cada entrada E del nodo
      Si (Overlap(E.MBR, dato))
        Imprimir "E.MBR intersecta con la Window Query"
} //Fin de PointQuery
```

En la figura 5.13 mostramos un ejemplo de una Window Query basada en la representación gráfica de los objetos y en la figura 5.14 indicamos el recorrido de la trayectoria que hace el algoritmo para dar respuesta a dicha consulta.

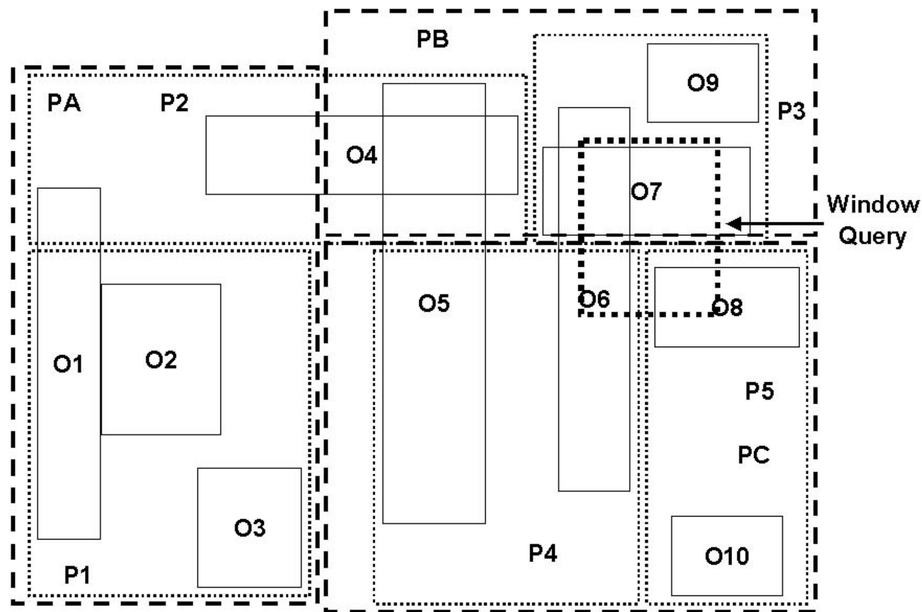


Figura 5.13 Window Query para la estructura

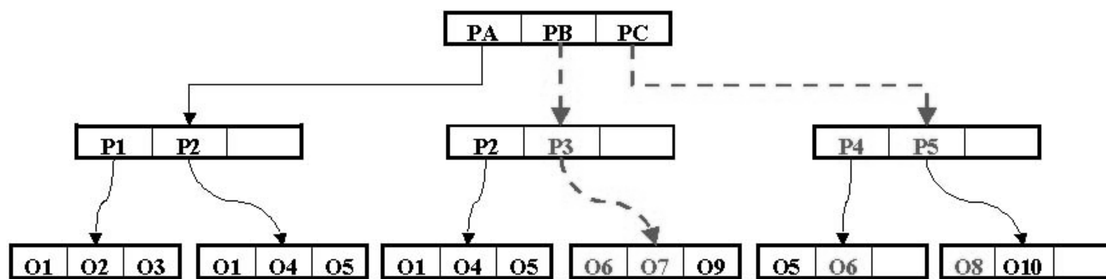


Figura 5.14 Trayectorias usadas para resolver la Window Query

5.6 Join y Semijoin Espacial

Finalmente analizamos la implementación del *join* y *semijoin* espacial. Dichas operaciones son de gran importancia para combinar objetos espaciales entre varias relaciones. El join espacial tiene dos variantes en base a su altura, el primero que mostramos en el siguiente pseudocódigo da por hecho que las alturas de los dos árboles son iguales, esto conlleva que las comparaciones se dan en los nodos con la misma altura. El descenso se hace hasta llegar a los nodos hojas y una vez en éstas, los objetos que coincidan con la condición de join podrán recuperar las tuplas que son apuntadas por cada entrada y unirlas para formar un nuevo grupo que será el conjunto que responde al join.

```
Public MBRJoin1 (Nodo N1, N2) {        //Árboles con la misma altura
  Para cada entrada E en N1
    Para cada entrada S en N2
      Si (Overlap(E.MBR, S.MBR))
        Si (EsHoja(N1))
          Escribe "E y S corresponden a una tupla del join"
        Sino
          MBRJoin1(E.hijo, S.hijo)
} //Fin del MBRJoin1
```

La segunda variante del join es cuando las alturas de los árboles son distintas. En este caso cuando en alguno de los árboles lleguemos a sus nodos hojas, el proceso continuará pero en la llamada recursiva se mantendrán fijas las entradas que pertenecen a los nodos hojas. En el siguiente pseudocódigo se muestra la variante que se hace al método anterior para poder resolver ambos casos.

```
Public MBRJoin2 (Nodo N1, N2) {        //Árboles con distinta altura
  Para cada entrada E en N1
    Para cada entrada S en N2
      Si (Overlap(E.MBR, S.MBR))
        Si (EsHoja(N1) AND EsHoja(N2))
          Escribe "E y S corresponden a una tupla del join"
        Sino
          Si (EsHoja(N1) AND NoEsHoja(N2))
            MBRJoin2(N1, S.hijo)
          Si (NoEsHoja(N1) AND EsHoja(N2))
            MBRJoin2(E.hijo, N2)
          Si (NoEsHoja(N1) AND NoEsHoja(N2))
            MBRJoin2(E.hijo, S.hijo)
} //Fin del MBRJoin2
```

El semijoin a diferencia del join reduce el esfuerzo y costo computacional, debido a que una vez encontrados en los árboles las entradas

de los nodos que cumplen con la condición de join, solo recuperaremos las tuplas de una sola relación. Esto en primer lugar evita la unión de tuplas de dos relaciones que implicaría mayor capacidad de almacenamiento así como complejidad. En segundo lugar una vez referenciadas las tuplas que pertenecen al conjunto solución podemos reducir la magnitud de atributos a través de una poda que puede estar dada en la consulta inicial.

El siguiente pseudocódigo muestra la variación sobre el algoritmo de join espacial para implementar el semijoin, cabe destacar que éste algoritmo está diseñado para procesar árboles con igual o distinta altura.

```
Public MBRSemijoin (Nodo N1, N2) {     //Árboles con distinta altura
  Para cada entrada E en N1
    Para cada entrada S en N2
      Si (Overlap(E.MBR, S.MBR))
        Si (EsHoja(N1) AND EsHoja(N2))
          Escribe "E corresponden a una tupla del semijoin"
        Sino
          Si (EsHoja(N1) AND NoEsHoja(N2))
            MBRSemijoin(N1, S.hijo)
          Si (NoEsHoja(N1) AND EsHoja(N2))
            MBRSemijoin(E.hijo, N2)
          Si (NoEsHoja(N1) AND NoEsHoja(N2))
            MBRSemijoin(E.hijo, S.hijo)
} //Fin del MBRSemijoin
```

En la figura 5.15 a) se muestra el estado de dos tablas en una base de datos espacial: la tabla Departamentos (*E*) contiene el departamento así como un atributo espacial que representa su ubicación, la tabla SectoresPuebla (*S*) contiene la ubicación geográfica de los sectores que se ubican en la ciudad de Puebla así como su director encargado.

Al realizar la consulta "*Recuperar los departamentos que pertenecen al Sector Puebla*" sobre las relaciones *E* y *S* (figura 5.15 a), mediante un join y un

semijoin obtenemos las relaciones intermedias *E´* y *S´* (figura 5.15 b) que contienen el atributo que será parte de la condición. Cabe destacar que esta parte corresponde a los árboles R^+ de indexado correspondientes a cada relación, en la figura 5.15 c) mostramos el resultado que devuelve tanto el join como el semijoin espacial a la consulta dada.

Departamentos (E)			
Depto_id	Nombre	Personal	Depto_MBR
3415	Sistemas	12	O1
5151	Rec. Hum.	5	O2
5453	Administracion	23	O3
9131	Contabilidad	7	O4
5452	Control Calidad	10	O5
8763	Rel. Públicas	15	O6
1943	Finanzas	4	O7
2006	Vinculación	6	O8

SectoresPuebla (S)		
Sec_id	Director	Sec_MBR
101	Juan	A
102	Pedro	B
103	Alejandro	C
104	Fernando	D
105	Arturo	E
106	Cecilia	F
107	Patricia	H
108	Karina	G
109	Ricardo	I
110	Pablo	J
111	Hugo	K

Figura 5.15 a) Relaciones espaciales

E´
Depto_MBR
O1
O2
O3
O4
O5
O6
O7
O8

S´
Sec_MBR
A
B
C
D
E
F
H
G
I
J
K

Figura 5.15 b) Relaciones Intermedias generadas

Resultado del Join

Departamentos ⊝ SectoresPuebla						
8763	Rel. Públicas	15	O6	108	Karina	G
1943	Finanzas	4	O7	108	Karina	G
1943	Finanzas	4	O7	109	Ricardo	I

Resultado del Semijoin

Departamentos ⊝ SectoresPuebla			
8763	Rel. Públicas	15	O6
1943	Finanzas	4	O7

Figura 5.15 c) Resultados a la consulta

En la figura 5.16 se muestran las dos estructuras de objetos indexados que corresponden a las relaciones anteriores. Los objetos indexados de la relación Departamentos están etiquetados por O_N (N número de objeto) y la relación SectoresPuebla por las primeras once letras del alfabeto. Al aplicar el join sobre ambas relaciones notamos que el resultado serán aquellas tuplas en donde sus objetos (que representan ubicación geográfica) se intersecten, en nuestro caso las tuplas *O6-G, O7-G* y *O7-I*.

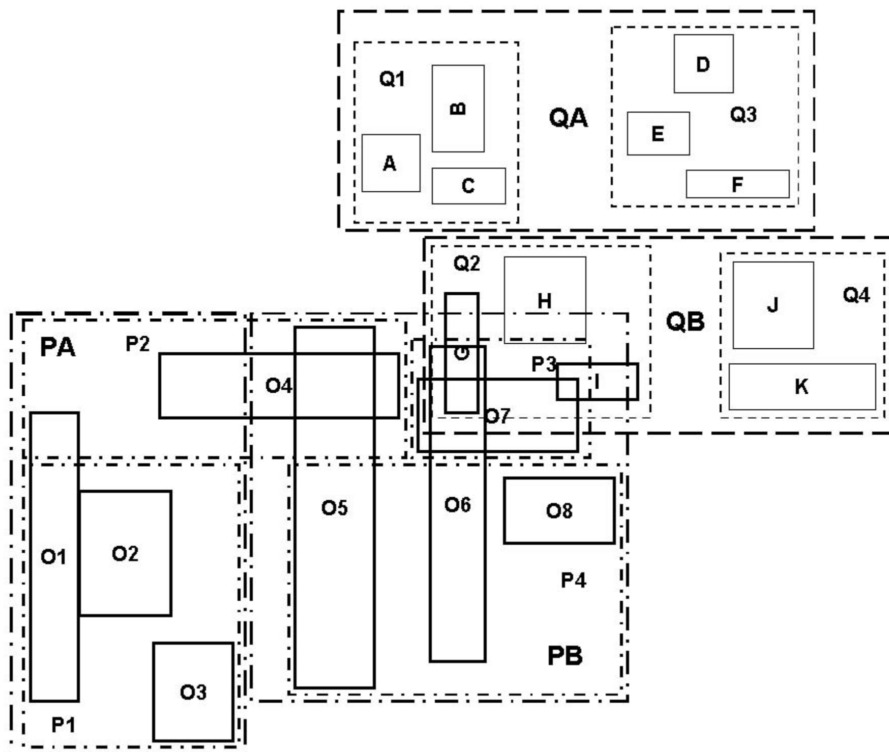


Figura 5.16 Relaciones perteneciente al semijoin

La figura 5.17 representa los árboles R^+ de las relaciones mostradas en la figura anterior así como los recorridos que el algoritmo hace para determinar la respuesta. El proceso es similar en el semijoin, excepto que éste solo reportará como resultado las tuplas *O6* y *O7* pertenecientes a la primera relación.

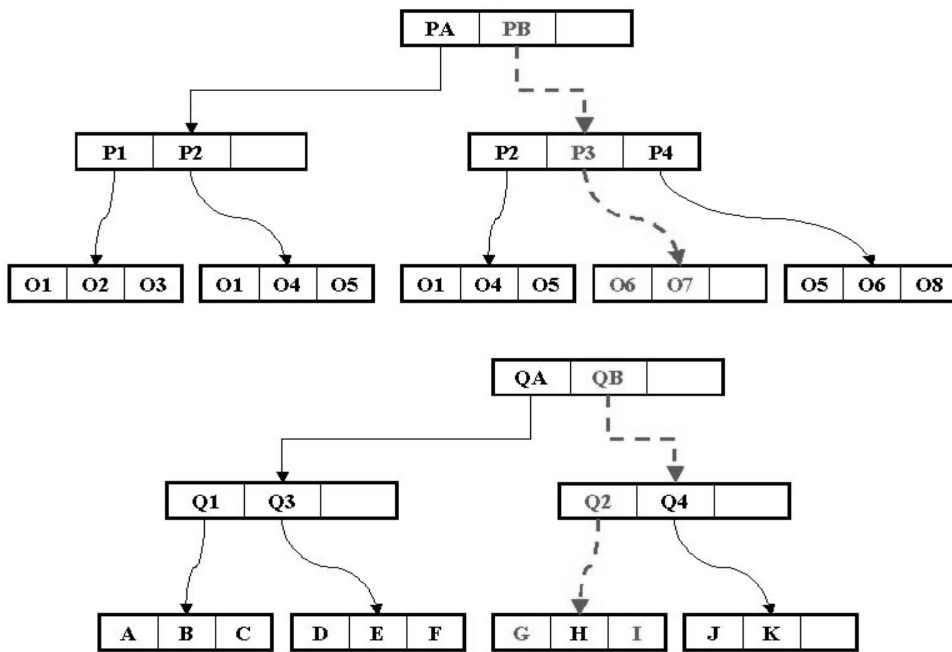


Figura 5.17 Árboles R+. Trayecto seguido por el *join*

5.7 Análisis de resultados entre el Join y Semijoin Espacial

La principal diferencia presente entre el join y semijoin es la cantidad de accesos a la base de datos. Mientras que en el join por cada tupla resultante del proceso tenemos que hacer dos accesos a la base de datos, uno a cada relación que conforma el join, en el semijoin solo tenemos que hacer el acceso a una de las dos relaciones presentes.

Aunado a que en el proceso de semijoin podemos podar los campos que no tienen mucha importancia para la consulta, provoca una menor cantidad de espacio de almacenamiento a utilizar por el resultado del semijoin. Esto es; una vez identificadas las tuplas que satisfacen la condición de semijoin podemos eliminar características (campos) que no sean requeridas para poder responder a la consulta original.

En la figura 5.18 podemos distinguir la ventaja en término de almacenamiento y eficiencia que implica el uso del semijoin. Mientras mayor sea el número de tuplas resultantes de la condición de join mejor será el rendimiento si usamos el método de semijoin ya que los accesos a la base de datos serán reducidos a la mitad mínimamente, en comparación con el join

natural. Aún más si consideramos que internamente se crea una tabla temporal para alojar los resultados, en nuestro caso también habrá ahorro en el espacio de almacenamiento, debido a que podremos tener el mismo número de tuplas pero con una menor cantidad de atributos, ya que en el semijoin solo consideramos los campos de una sola relación.

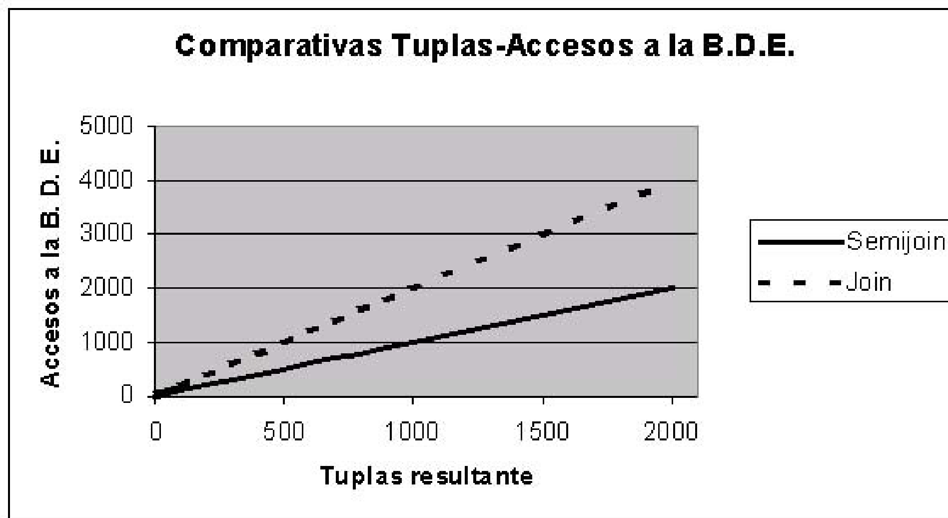


Figura 5.18 Tabla comparativa Join - Semijoin

Así como se decrementa el número de accesos a la base de datos espacial, también reducimos el espacio en disco necesario para almacenar la relación temporal creada por el semijoin. Mientras el join crea una relación temporal con la junta de las dos tuplas de las relaciones involucradas así como con todos sus atributos (campos), el semijoin solo tendrá las tuplas que cumplan la condición de traslape de la primera relación.

Finalmente, podemos afirmar que las pruebas anteriores ofrecen fundamentos para poder asegurar que el manejo de datos espaciales a través de árboles R^+ es factible, ya que proporciona algoritmos para su manejo y consultas, así como el beneficio que tiene la modificación del join (semijoin) que reduce tiempo de respuesta y espacio de almacenamiento.

CONCLUSIONES

Existen infinidad de aplicaciones que pueden adaptarse para ocupar una base de datos espacial y poder tener una mejor representación de sus objetos. A su vez, las BDE permiten realizar consultas y/o búsquedas en base a propiedades espaciales como lo son intersección entre regiones, adyacencia y consultas en base a distancia, entre las operaciones más significativas.

En este trabajo se presenta una investigación acerca del diseño y la implementación de un mecanismo de indexado espacial que permita ser una herramienta para el manejo y procesamiento de este tipo de tareas. Se presentan como estructura de datos a los R^+-trees, ya que sus características permiten ventajas sobre las demás variedades de árboles, siendo la más significativas el no traslape entre nodos del mismo nivel.

Se plantean modificaciones a los algoritmos de inserción, búsqueda, borrado y algunas consultas topológicas básicas en el manejo de datos espaciales. Además se presenta la implementación del join espacial utilizando R^+-trees con su variante de semijoin que permita un manejo eficiente de almacenamiento y accesos a la base de datos.

Parte importante del trabajo consiste en proponer un método de empaquetamiento eficiente que permita crear un árbol R^+ lo suficientemente bueno en términos de agrupación. Esto es, que sus nodos intermedios en un mismo nivel del árbol no presenten intersección entre ellos y segundo que el árbol no crezca en demasía. Para dicho proceso ocupamos como criterio de agrupación la distancia euclidiana que existe entre ellos.

Por otro lado implementamos la operación de join así como su variante de semijoin. Dicha operación es de las más importantes en bases de datos ya que permite vincular dos relaciones obteniendo resultados de mayor interés para una aplicación en particular. Además la operación join siempre ha sido controversial por su elevado costo computacional.

De esta manera el objetivo general puede afirmarse que queda cumplido, así como los objetivos específicos que resultan de esta investigación.

La principal aportación de este trabajo es la creación de una metodología para la implementación de un R^+-tree con sus operaciones más importantes entre las que destacan: empaquetamiento, inserción, borrado y búsqueda de objetos indexados, las operaciones topológicas básicas y el join con su variante del semijoin.

El manejo de un lenguaje orientado a objetos me permitió en primera instancia el aprendizaje y práctica del mismo, el conocimiento de los conceptos de la programación orientada a objetos como los son las clases, la abstracción, instancia de clases, polimorfismo y una de las características más importantes: la reutilización de código.

Por último, mi perspectiva de los mecanismos de acceso a datos espaciales (y en general multidimensionales) se fortaleció con la investigación de los mismo y con la implementación de los principales algoritmos para el funcionamientos de los R^+-trees.

TRABAJO FUTURO

Existen distintos métodos de indexado de objetos n-dimensionales, en nuestro caso ocupamos el método de recorte para poder trabajar con objetos bidimensionales, pero no está restringido solo a este tipo, por tanto entre los trabajos por realizar están:

- Realizar modificaciones a los algoritmos para llevar a cabo pruebas con objetos de dimensiones mayores.
- El principio de los árboles R^+ es no permitir intersección entre nodos de un mismo nivel, en nuestras pruebas utilizamos ejemplos en donde el traslape entre objetos no se presenta en demasía. Por tanto resultaría importante experimentar el prototipo con este tipo de situaciones.
- La inclusión de la lógica difusa en los árboles R permitiría diferentes tipos de agrupación al utilizar el empaquetamiento. Además, esta proporcionaría un enfoque distinto que permitiría adaptar otro tipo de circunstancias de la vida real a las bases de datos espaciales como es el concepto de proximidad.
- Utilización de la metodología para trabajar con una aplicación de la vida real (geográfica, biológica), y realizar pruebas, en base a esto determinar que tan significativos son los resultados comparados con un análisis hecho por un especialista del tema.

BIBLIOGRAFÍA

[1] Hee Kap Ahn, Nikos Mamoulis, Ho Min Wong, *"A Survey on Multidimensional Access Methods,"* UU-CS-2001-14, Mayo 2001.

[2] Gaede, V., Günther, O., "*Survey on Multidimensional Access Methods,"* Technical Report ISS-16. Department of Economics and Business Administration, Humboldt University Berlin, 1997.

[3] Nievergelt, J., H. Hinterberger, K. Sevcik, *"The grid file: An adaptable, symmetric multikey file structure,"* Proc 3th. ECI Conf., New York, pp. 236-251, 1981.

[4] Tamminen, M., *"The extendible cell method for closest point problems,"* BIT 22, pp. 27-41, 1982.

[5] Finkel, R. A. y J. L. Bentley, *"Quead trees: a data structure for retrieval on composite keys,"* 1974.

[6] Bentley, J. L., *"Multidimensional Binary Search Trees Used For Associative Searching,"* Communications of the ACM, 18(9), 509-517, 1975.

[7] Robinson, J. T., *"The K-D-B-tree: A Search Structure For Large Multidimensional Dynamic Indexes,"* In Proc. ACM SIGMOD Internactional Conference on Management of Data, pp. 10-18, 1981.

[8] Guttman, A., *"R-trees: A Dynamic Index Structure for Spatial Searching,"* In Proc. ACM SIGMOD International Conference on Management of Data, pp. 47-57, 1984.

[9] Beckmann, N., H.-P. Kriegel, R. Schneider, B. Seeger, *"The R^*-tree:An Efficient and Robust Access Methods for Points and Rectangles,"* In Proc. ACM SIGMOD International Conference on Management od Data, pp. 322-331, 1990.

[10] Sellis, T., N. Roussopoulos, C. Faloutsos, *"The R+-tree: A dynamic index for multidimensional objects,"* In Proc. 13th. Int. Conference on Very Large Data Bases, pp. 507-518, 1987.

[11] Six, H., P. Widmayer, *"Spatial searching in geometric databases,"* In Proc. 4th. IEEE Int. Conf. On Data Eng., pp. 496-503.

[12] Kingston Centre for GIS, Kingston University, *"Introduction to GIS and Geospatial Data,"* UK, 2001.

[13] Somodevilla, María J., Petra, Frederick, *"Indexing Mechanisms to Query FMBRs,"* In Proc. NAFP04, 2004.

[14] Shashi Shekhar y Sanjay Chawla, *Spatial Databases (a Tour),* Editorial Prentice Hall, 2003.

[15] Brinkhoff, T., H-P. Kriegel, B. Seeger, *"Efficient Processing of Spatial Joins using R-trees,"* In Proc. ACM SIGMOD International Conference on Management of Data, pp. 237-246, 1993.

[16] P. Bernstein, D. M. Chiu. "*Using semi-joins to solve relational queries."* Journal of the ACM, 28, 1981.

[17] Hinrichs, K. *"Implementation of the grid file: Desing concepts and experience,"* BIT 25, pp. 569-592, 1985.

[18] Hutflesz, A., Six, H. W., Widmayer, P., *"Tein grid files: Space optimizing access schemes,"* In Proc. ACM SIGMOD International Conference on Management Data, pp. 183-190, 1988.

[19] Valduriez Patrick, "*Semi-join Algorithms for Multiprocessor Systems*", Proceedings of the 1982 ACM SIGMOD international conference on Management of data, pp. 225-233, 1982.

[20] D. Comer. "*The Ubiquitous B*-tree". ACM Computing Surveys. Vol. 11, No. 2. pp. 121-137. 1979.

[21] S. Berchtold, D.A. Keim y H.P. Kriegel; *"The X-tree: An Index Structure for High-Dimensional Data"*. International Conference on Very Large Data Bases, (VLDB'96). pp. 28-39. 1996.

[22] Tetsuo Asano, Desh Ranjan, Thomas Roos, Emo Welz; "*Space Filling Curves and their Ise in the Design of Geometric Data Structure*". Lecture Notes in Computer Science. pp. 36-48. 1997.

[23] A. Papadopoulos, P. Rigaux y M. Scholl. "*A Performance Evaluation of Spatial Join Processing Strategies*" Advances in Spatial Databases (SSD´99). Lecture Notes in Computer Science Vol. 1651. pp. 286-307. 1999.

Índice

INTRODUCCIÓN 1

OBJETIVO GENERAL 2

OBJETIVOS PARTICULARES 2

ORGANIZACIÓN DE LA TESIS 3

CAPÍTULO I 4

MARCO TEÓRICO 4

CAPÍTULO II 9

MECANISMOS DE INDEXADO ESPACIAL 9

Introducción 9

2.1 Datos Espaciales 9
2.1.1 Características de los datos espaciales 9
2.1.2 Requerimientos de los métodos de acceso espacial 10

2.2 Clasificación de los Métodos de Acceso Multidimensional 11
2.2.1 Métodos de Acceso al Punto (PAM) 11
2.2.2 Métodos de Acceso Espacial (SAM) 12

2.3 Estructuras de Datos Espaciales 13
2.3.1 Métodos de Acceso Hashing 13
2.3.1.1 El Grid File y sus variantes 13
2.3.1.2 Otros métodos hashing 15
2.3.2 Quadtrees 16
2.3.3 El k-d-tree y sus variantes 17
2.3.4 R-trees y sus variantes 18
2.3.1.3 El R^+-Tree 20
2.3.1.4 El R*-Tree 21

CAPÍTULO III 23

PROCESAMIENTO DE CONSULTAS ESPACIALES 23

Introducción 23

3.1 Tipos de consultas espaciales 23

3.1.1 Consultas de Selección Espacial 23
3.1.2 Junta Espacial (*Join*).. 26

3.2 Semijoin espacial 27

CAPÍTULO IV 29

METODOLOGÍA PROPUESTA PARA ACCESO Y PROCESAMIENTO DE DATOS ESPACIALES.. 29

4.1 Características del R^+-tree 29

4.2 Búsqueda en el R^+-tree 32

4.3 Inserción en el R^+-tree 32

4.4 Borrado en el R^+-tree 33

4.5 División de Nodos en el R^+-tree 34

4.6 Algoritmo de Empaquetamiento 36

4.7 Join Espacial 37
4.7.1 Definición y tipos de Junta Espacial.. 38

4.8 Semijoin Espacial... 39

CAPÍTULO V 41

IMPLEMENTACIÓN Y ANÁLISIS DE RESULTADOS 41

5.1 Diseño del R^+-tree 41

5.2 Empaquetado en un R^+-tree 43

5.3 Inserción en un R^+-tree 48

5.4 Borrado en un R^+-tree 52

5.5 Búsqueda en un R^+-tree 54

5.6 Join y Semijoin Espacial 56

5.7 Análisis de resultados entre el Join y Semijoin Espacial 61

CONCLUSIONES.. 63

TRABAJO FUTURO... 65

BIBLIOGRAFÍA 66