Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamendo de Ciencias de la Computación

Diseño y análisis de algoritmos

Tarea 1

Memoria secundaria: R-Tree

Integrantes: Fabián Souto H.
Francisco Clavero H.
Profesores: Gonzalo Navarro
Auxiliares: Jorge Bahamonde
Ayudantes: Sebatián Ferrada
Willy Maikowski

Fecha de entrega: 8 de noviembre de 2016
Santiago, Chile

Índice de Contenidos

Lista de Figuras

Lista de Tablas

1. Introducción

Un R-Tree es un árbol parecido a un B-Tree, que sirve para poder manejar rectángulos en memoria secundaria. Se puede *buscar* e *insertar* rectángulos. Por ejemplo la búsqueda, dado un rectángulo C retorna la cantidad de rectángulos que intersectan con él.
El árbol debe preservar un conjunto de invariantes:

- Cada nodo de la estructura es un rectángulo: el MBR (*minimum boundig rectangle*) de los rectángulos que cuelgan de él.
- Los datos se encuentran en las hojas.
- Cada nodo del R-Tree almacena al menos m rectángulos (llamaremos a este el invariante m) y no puede tener más de M rectángulos (lo llamaremos invariante M).
- Todas las hojas están a la misma profunidad en el árbol.

El objetivo de este informe es mostrar una implementación de un R-Tree, realizada en lenguaje *Java* en donde se ocupan dos heurísticas distintas para la inserción: *Linear Split* y *Greene's Split*.
Además será motivo de interés poder analizar el comportamiento del árbol con estas dos heruísticas distintas, analizando la operación de búsqueda después de haber hecho las inserciones ocupando alguna de las dos.
Parte importante, para hacer una implementación más eficiente de esta estructura, es poder agregar un *buffer* que permite tener una cantidad fija de nodos del árbol en memoria, así evitando acceder a disco cada vez que se necesite una parte de él. Se analizarán las diferencias respecto a no tener un *buffer* o similar a tener uno de tamaño 1.

Posterior a explicar la implementación, se expondrán experimentos llevados a cabo con cantidades de datos superiores a 2^9 generados al azar, respetando cotas inferiores y superiores; luego se estudiarán en profundidad y se analizarán los resultados de forma de hacer un contraste con las hipótesis creadas.

2. Implementación

La implementación del árbol consta de 9 clases y 2 interfaces. Las siguientes secciones abarcan en mayor detalle cada una de éstas.

2.1. Interfaces

2.1.1. Rectangle

Diremos que toda componente del árbol es un rectángulo, pues la raíz tiene un MBR (*Minimum Bounding Rectangle*) que es un rectángulo, lo mismo con cualquier nodo y obviamente un dato es en sí mismo un rectángulo.

Todo rectángulo tiene un MBR, se puede buscar dado un rectángulo, se puede contar los accesos a disco que hace dada la búsqueda de un rectángulo y se puede optener su *path*, que es el path hacia su archivo en disco.

2.1.2. Splitter

El splitter es el encargado de manejar los casos de *overflow*. Cuando un nodo tenga *overflow* acude a su splitter para que pueda solucionar el problema. Implementa alguna heurística. Se encarga de mantener el invariante M y el invariante m.

Un splitter lo único que hace es *split* dada una lista de nodos/datos que repartir y dos nodos donde insertarlos.

2.2. Clases

2.2.1. Data

Es una clase que implementa la interfaz *Rectangle* y representa un dato, es decir, un rectángulo en concreto. Dentro de las cosas que puede hacer es obtener su área, ver si intersecta con otro rectángulo dado o si es igual a otro rectángulo (coinciden en dimensiones y coordenadas).

2.2.2. Node

Es la clase *core* del árbol, implementa *Rectangle*. Un nodo sabe cuales son los m y M que debe tener para cumplir sus invariantes, sabe quién es su *Splitter*, sabe el *path* a sus hijos (para cargarlos a memoria) y sabe su propio *path*.

Dentro de las cosas importante que puede hacer un *Node*, aparte de implementar a *Rectangle*, es calcular cuánto crecerá su MBR si insertamos un rectángulo dado, insertar un rectángulo dado (llama recursivamente al hijo que tiene que crecer menos), hacer *split* en caso de overflow, actualizar su MBR dada una inserción, etc.

2.2.3. RootNode

Hereda de *Node*. Se implementó esta clase pues en caso de overflow, la clase node arroja una excepción, en donde el padre es el encargado de mandarlo a hacer el split y recibir los dos nuevos nodos que contienen lo que contenía el nodo con overflow. ¿Y qué hacemos en la raíz? Esa es la función de esta clase, que simplemente en caso de tener overflow, él mismo se actualiza y setea sus antiguos hijos a los dos nuevos que contienen todos los datos de la antigua raíz del overflow, así el árbol crece hacia la raíz.

2.2.4. LeafNode

Hereda de *Node*. Dentro de las cosas que hace, la mayor diferencia con *Node*, es la inserción, pues en este caso no puede insertar en el rectángulo que menos crezca, pues esos ya son datos, no son nodos. Entonces es el encargado de tomar el nuevo *Data*, guardarlo en disco, agregarlo a sus hijos, los datos que tiene ese nodo, y en caso de overflow generar la excepción.

2.2.5. RTree

Es la clase que se ocupa para manejar un R-Tree. Obviamente sabe sus parámetros m, M, *Splitter* y quién es su raíz. Tiene variables y métodos *static* de forma que cualquier nodo pueda llamar a las funciones para el manejo de los accesos a disco.

Dentro de las cosas que permite están:

- Obtener un nuevo *path* para algún archivo nuevo que se debe guardar en disco.
- Insertar datos en el árbol.
- Realizar la operación de búsqueda (simplemente hace *this.root.search(C)*).
- Realizar estadísticas: contar accesos a disco, cantidad de rectángulos, cantidad de nodos, altura y porcentaje de uso.

Pero lo más importante, es que maneja los accesos a disco y al *buffer*. Tiene un *buffer* de tamaño fijo que permite tener cierta cantidad de nodos en memoria, permitiendo ahorrar tiempo. Si algún nodo quiere acceder a algún objeto tiene que llamar *RTree.getObj(path)*, donde *path* es un *String* que representa la ubicación del archivo que se quiere leer para carga el objeto. De esta manera vemos si el objeto está en el buffer (en tiempo constante, cada archivo tien un index) y si está lo retornamos, si no, lo *deserializamos* (cargamos en memoria el objeto tal cual se guardó (serializó) la última vez).

Algo similar es lo que pasa con *save*, donde le pasamos un *Rectangle C* y lo guarda en el *buffer*, viendo antes si lo que estaba en el buffer era él o no, en caso de que no fuera él ahí recién guardamos (*serializamos*) el objeto en memoria secundaria.

2.2.6. Otras clases

Además de las clases mencionadas tenemos las clases que implementaron los Splitters, una implementa Linear Split y otra implementa Greene's Split. También está una clase *GeneralException* que

ocupamos para arrojar excepciones generales, principalmente para los casos de overflow. Y como último está la clase de experimentación, que abordaremos en esa sección y una carpeta que incluye diversos test que ocupamos para ir comprobando la funcionalidad de nuestra implementación.

3. Experimentación

Para la experimentación se creó una clase *Experiment*, que permite generar logs de los experimentos, mostrar en pantalla las estadísticas que lleva y además generar un archivo CSV para poder hacer un análisis más profundo con *python*. Tiene métodos para formatear la muestra de tiempo, para generar datos y generar árboles.

Un *experiment* recibe la cantidad de datos con los cuales se desea experimentar y llama a *experimentTree* donde ejecuta el experimento sobre un árbol. Lo hace dos veces para probar con el mismo n (cantidad de datos) pero con las dos heurísticas.

El experimento primero toma el tiempo de creación del árbol (insertar todos los datos), luego calcula el espacio de uso promedio, después toma el tiempo de cuanto demora en completar todas las *queries* y finalmente calcula cuantos accesos a disco se tuvieron que hacer para completar esas *queries*.

3.1. Hipótesis

3.1.1. Cota superior inserción

Se tienen n elementos en el árbol. Se desea insertar un nuevo elemento. Para insertar el elemento se debe llegar a una hoja, si hay n elementos en el árbol entonces tiene altura $\log_M(n)$, eso toma llegar a una hoja. Ahora para elegir la hoja correcta por la cual bajar se debe iterar por sobre todos los hijos del nodo para elegir el que corresponde (el que su MBR aumenta mínimamente), en el peor caso tiene M hijos sobre los cuales iterar. O sea llegar a una hoja toma $M \cdot \log_M(n)$, tenemos que iterar sobre M hijos $\log_M(n)$ veces.

3.1.2. Peor caso inserción

Una cota fácil sería pensar que tenemos que insertar n hijos, entonces tenemos $O(nM \log_M(n))$, pero no resulta muy preciso.

Algo más concreto es ver que el nodo $i+1$ para insertarse toma (en el peor caso) $M \log_M(i)$. Por lo tanto sumando desde el primero al último resulta:

$$M \sum_{i=2}^{n} \log_M(i-1)$$

$$M \log_M(\prod_{i=1}^{n-1} i)$$

$$M \log_M((n-1)!)$$

Luego, ocupando *Stirling* sabeos que $(n-1)! = O(n^2)$:

$$O(M \log_M(n^2))$$

Pero a esto no le hemos sumado el costo de que ocurra un split. Ambas heurísticas ocupadas cargan una sola vez los datos en memoria que necesitan para hacer el split y luego escriben los nodos resultantes, o sea hacen $(M + 1 + 2) = (M + 3) = O(M)$ I/Os.

Una opción fácil también sería pensar que se hace split cada vez que insertamos un nodo, pero esto no se acerca mucho a la realidad, busquemos un peor caso más ajustado.

En el peor caso, el adversario quiere que hagamos la mayor cantidad de *splits*. Por lo tanto nos dará elementos para insertarlos uno tras otro de manera que al bajar por el árbol se escoge el nodo más lleno y se inserta en la hoja más llena. Los datos insertados son tales que al ocurrir un split los nodos se reparten los hijos de forma que uno de los dos queda como el peor caso posible, o sea con $(M - m)$ hijos, con lo cual con m hijos más se llenará de nuevo.

Ahora podemos ver que cada m inserciones se produce un *split* en la hoja más cargada. Siguiendo este régimen de inserción veamos cuantos splits se requieren para armar un árbol de n elementos.

Notemos que un árbol de n elementos con este régimen de inserción tendrá una altura $h = O(\log_M(n))$.

Digamos que el costo de *split* de una hoja será $s(1) = O(M)$ y que ocurre cada m inserciones. Así el *split* del padre de una hoja, será $s(2) = m \cdot s(1) = O(mM)$ tomando en cuenta todos los splits que tuvieron que pasar para llegar a él; esto es que el split del padre de una hoja cuesta m splits de su hija, así para tener un árbol de altura 2 tenemos que hacer $s(2)$ y pagar esos splits. Análogamente podemos ver que $s(i) = m^{i-1}M$.

Sabiendo que la raíz será el nivel h, para obtener un árbol de altura $h = O(\log_M(n))$ tenemos que hacer todos los splits que cueste llegar a hacer $s(h)$, o sea tenemos que pagar $s(h) = s(\log_M(n)) = m^{\log_M(n)-1}M$.

De esta forma podemos ver que el costo de crear un árbol será el costo de insertar todos los elementos (toma $O(M \log_M(n^2))$ más todos los splits que se tuvieron que hace que cuestan $s(\log_M(n))$. El costo final será:

$$O(M \log_M(n^2) + m^{\log_M(n)-1}M)$$

3.1.3. Peor caso queries

En el peor de los casos está intersectado con todos, o sea debe cargar todos los nodos del árbol, que son $M^h = M^{\log_M(n)}$, por lo que el costo en el peor caso será $O(M^{\log_M(n)})$.

3.2. Resultados

3.2.1. Resultados numéricos

3.2.2. Gráficos

3.2.3. Análisis y conclusión

El primer punto a notar, es que como todos podíamos esperar entre mayor el n mayores son los accesos a disco y mayor es la demora.
Para nuestra sorpresa, las cotas inferiores que habíamos pensado no se ajustaron bien a los resultados experimentales.

En el caso de las consultas es difícil poder determinar una cota justa, puesto que deberíamos calcular la esperanza de la cantidad de elementos que se intersectan con un rectángulo generado al azar. Esa sería la cantidad de accesos a disco. Pero podemos ver que nunca supera el peor caso que mencionamos, que es acceder a todos los nodos.

Esto se puede notar en el gráfico **??**, donde al dividir por esta cota superior se invierten los puntos. Esto quiere decir que crecía más lento que en el peor caso. Nuestra cota inferior resultó en un caso que practicamente no sucede nunca.
En el caso del tiempo de creación es donde nos llevamos la mayor sorpresa. Probamos con el peor caso que habíamos calculado que era más justo que el fácil pero resultaba que la curva se mantenía en proporciones, indicando que funcionaba peor que ese peor caso. Después probamos con el peor caso más fácil, que es pensar que hace la peor inserción posible las n veces y además por cada inserción hacía un *split*, pero para nuestra no grata sorpresa, esto aún quedaba corto.

Una hipótesis es que no hayamos considerado el tiempo de CPU del algoritmo, puesto como solo tomábamos en cuenta los I/Os. Puede ser que Java y su manera de procesar la estructura haya llevado a que aportara una parte no despreciable en términos de costo. A estas alturas no alcanzamos a rehacer todos los experimentos y todos los análisis, podría quedar como problema abierto.