Tarea 1 CC4102
Implementación y Experimentación de R-Trees en Memoria Secundaria

Javiera Born
Nicolás Lehmann
Estefanía Vidal

15 de Octubre de 2012

1. Descripción del Problema

En la primera tarea del curso se busca implementar un R-Tree en memoria secundaria. Esta estructura consta de nodos que contienen, dado un parámetro b, entre b y $2b$ rectángulos en cada nodo, correspondiendo cada nodo con una página de disco, con el fin de minimizar las visitas a los nodos al realizar operaciones. Se busca que la estructura almacene los rectángulos que representan datos reales en las hojas, y los llamados *Minimum Bounding Rectangles (MBRs)*, rectángulos que acotan de manera mínima a aquellos de nivel inferior, de los rectángulos de estos nodos almacenados hacia arriba, llegando hasta la raíz. En estos árboles se tendrán tres operaciones:

1.1. Inserción

La inserción consiste en agregar un nuevo rectángulo al R-Tree. Si hay espacio disponible en el nodo correspondiente donde realizar la inserción, es decir, hay menos de $2b$ rectángulos, entonces sólo se inserta y actualizan los MBRs. En caso contrario, se realiza la operación *split*, que separará el nodo en dos nodos, de tamaños b y $b+1$: para ello se determinan dos rectángulos que estarán uno en cada nodo, y luego se verifica, entre todos los restantes, cuál agranda menos el MBR de alguno de los dos, y se agrega a ese grupo,

iterando hasta que se agregan todos. Se utilizan dos métodos para encontrar los primeros rectángulos:

Método 1: Consiste en tomar todos los pares de rectángulos, y ver cuál genera el MBR con el área máxima.

Método 2: Consiste en tomar todos los pares de rectángulos, y ver cuál tiene una mayor distancia.

1.2. Búsqueda

Para la búsqueda simplemente se recorre el árbol viendo los MBRs que intersectan con el rectángulo buscado, llegando hasta la raíz, donde se encuentran aquellos que efectivamente insersectan.

1.3. Borrado

El borrado elimina un rectángulo identificado por la hoja en que está. Análogamente a la inserción, puede eliminarse si todavía quedan b elementos en el nodo, pero si hay menos entonces se debe utilizar un método para ajustar los nodos para que cumplan la estructura del R-Tree.

Método 1: Consiste en verificar hasta dónde la eliminación hace *underflow* en los padres y borrar esos nodos, y luego reinsertar todos los rectángulos que quedan huérfanos en el camino.

Método 2: Consiste en verificar si el hermano puede donar un rectángulo —posible si es que tiene más de b elementos. Si no, se une el hermano y el nodo actual, generando un nodo de $2b-1$ elementos, y actualizando este nodo menos hacia arriba. En ambos casos se actualizan los MBRs hacia arriba.

2. Hipótesis

Se espera que las operaciones en la implementación del R-Tree tarden tiempos acordes a aquellos calculados teóricamente para esta estructura de datos.

Inserción: Consistirá, primero, en un recorrido en altura del árbol visitando los rectángulos de cada nodo, que toman en promedio $\mathcal{O}(1)$. Dependiendo del método que se utilice en caso de *overflow*, se tendrá que el costo total de la inserción será $\mathcal{O}(1)$ con el primer método, y $\mathcal{O}(1)$ con el segundo método.

Búsqueda: Consiste en recorrer el árbol por todos aquellos caminos que intersecten con el rectángulo buscado. Así, se tiene que en el peor caso se visitan todos los rectángulos de cada nodo, obteniendo $\mathcal{O}(1)$, con un promedio de $\mathcal{O}(1)$.

Borrado: En este caso primero buscamos el elemento, y luego se utizará alguno de los dos métodos para lidiar con el *underflow*. En el primer caso tendremos $\mathcal{O}(1)$, dado que subiremos y luego reinsertaremos los elementos de el subárbol correspondiente, y en el segundo caso se tiene $\mathcal{O}(1)$.

Se espera también observar un *overhead* en las operaciones de inserción y borrado, debido a la modificación contínua de archivos por parte del algoritmo: esta interacción con el sistema de archivos debería traer un costo asociado.

Finalmente, se espera un escalamiento estasble respecto al orden de los algoritmos, pues la optimalidad teórica de los parámetros de cada uno no se basa en el número de elementos con los que se trata, sino que en parámetros de la unidad de almacenamiento secundario y del tamaño de la memoria principal.

3. Diseño Experimental

El lenguaje utilizado para desarrollar la estructura de datos y sus algoritmos fue C. Para ello, se dividió el proyecto en varias partes y sub-estructuras. Se incluyen en esta entrega los archivos que implementan los algoritmos, los archivos con las estructuras de datos, ejemplos de prueba y un *Makefile* para compilar todo.

3.1. Generación de Instancias y Metodología

Para probar los algoritmos se generaron rectángulos con datos aleatorios cuyas coordenadas estuviesen entre 0 y 500000, y tamaño de cada lada entre 0 y 100. Para asegurar una buena distribución se utiliza el método `drand48()`, que entrega un valor flotante de doble precisión entre 0 y 1 con distribución uniforme, calculando primero x_1 y y_1 entre 0 y 500000, luego calculando el tamaño del lado, uniforme entre 0 y 99, y a partir de eso x_2 y y_2, verificando que esté dentro del rango solicitado. Para insertar y buscar, simplemente se genera un rectángulo y se inserta o se busca en el árbol. En cambio, para el borrado, se va accediendo por posiciones aleatorias j del nodo desde la raíz

hasta llegar a una hoja, recalculando el j en cada paso: entonces se borra el elemento en la posición j del nodo. Con esto se asegura que existe el elemento a borrar y se obtiene la información del nodo.

3.2. Determinación de Parámetros

Resulta necesario calcular el valor que debe tomar b para cumplir las características pedidas en el enunciado. Para ello debemos analizar las estructuras de datos utilizadas en la implementación.

```
typedef struct{
        float x1;
        float x2;
        float y1;
        float y2;
}rect;
```

Es fácil ver que al ser un rectángulo simplemente cuatro floats, su tamaño será de $16B$.

```
typedef struct{
        rect *r;
        int child;
}nodeVal;
```

En el caso de esta estructura, veremos que contiene un puntero a un rectángulo (que ya sabemos es de tamaño $16B$ y un int, de $4B$, obteniendo así un tamaño de $20B$.

```
typedef struct{
        int size;
        int address;
        nodeVal *values[2*b+1];
        rect *MBR;
        int leaf;
}node;
```

Ahora finalmente podemos considerar aquello que nos interesa, que es el nodo. Tenemos que se compone de 2 enteros (pues `address` no se guarda en

disco: es el nombre del archivo), un `rect` ($16B$) y b `nodeVals`, que corresponden a $2b * 20B$, resultando en que el tamaño total debe ser $24 + 40bB$. Se debe determinar ahora el tamaño de página del disco. Para ello, se utiliza el comando `sysconf(_SC_PAGESIZE)`, que nos arroja como resultado un tamaño de 4096 bytes. Despejando, se obtiene que el b a utilizar debe ser aproximadamente 100.

3.3. Experimentación con Parámetros Fijos

Una vez determinados los parámetros a utilizar, se procedió a realizar ejecuciones de éstos utilizando distintas cantidades de elementos generados aleatoriamente. Tal como se pide en el enunciado, se utilizó $n \in \{2^9, 2^{12}, 2^{15}, ..., 2^{27}\}$ y $k \in \{1, 3, 7\}$, manteniendo así fijo $m = n/(2k)$ intercalando las operaciones de inserción, búsqueda y borrado. Se utilizaron las secuencias $i^{km}f^{km}$ para determinar la mejor inserción, $(i^k d^k i^k)^m f^{km}$ para determinar el mejor borrado, y finalmente $(idi)^{km} f^{km} (did)^{km}$ para probar la estructura de datos completa con todas las operaciones. Los parámetros que se midieron en cada experimento fueron dos: el tiempo por cada secuencia de operaciones tomada, utilizando el comando `time` para obtener tiempo de sistema, de usuario y real, y se midieron la cantidad de accesos a páginas de disco, medido cada vez que se abre un nodo a memoria principal o se escribe a disco.

3.3.1. Dificultades de Experimentación

- La primera dificultad surgió al terminar el desarrollo y comenzar la generación de rectángulos aleatorios, obteniéndose casos que no se habían considerado y que ilustraron errores de implementación que no se habían detectado antes.

- Se encontraron problemas de diseño en la escritura de archivos, puesto que los valores no se escribían como enteros (en representación binaria), si no que como `char[]`, lo que hacía mucho más lenta la escritura y lectura. Al solucionar este problema se redujeron los tiempos a aproximadamente la mitad.

- Inicialmente se tomaron pruebas tomando 10 muestras de cada combinación de k y n, lo cual rápidamente condujo a que los algoritmos corrieran demasiado tiempo (en 2^{18} se corrió la prueba por más de 8

horas sin terminar), por lo que se redujeron las muestras a 5 y posteriormente a 2 según el crecimiento del n.

3.3.2. Descripción del Hardware Utilizado

4. Resultados

5. Interpretación y Conclusiones

6. Anexos