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

Tarea 1

Manejo de overflows en R-Trees implementados en memoria Secundaria

Integrantes: Vicente Illanes
Gabriela Mendoza
José Vergara
Profesores: Pablo Barcelo
Gonzalo Navarro
Auxiliar: Dustin Cobas
Ayudantes: Francisco Sanhueza
Ignacia Parra
Nicolás Higuera

Fecha de entrega: 6 de mayo 2019
Santiago, Chile

Índice de Contenidos

Lista de Figuras

1. Introducción

El acceso a memoria secundaria en un computador es una operación costosa, que llega a ser órdenes de magnitud más lenta que el acceso a memoria principal. Este es un hecho que se debe tomar en cuenta a la hora de diseñar y analizar el costo de algoritmos que estén estrechamente relacionado con esta operación.

En esta tarea se implementa la estructura de datos *R-tree* que permite manipular *rectángulos* en memoria secundaria. En el contexto del estudio de operación de algoritmos computacionales, interesa estudiar los tiempos de construcción y búsqueda de la estructura, con diferente número de inserciones y bajo dos heurísticas de rebalanceo. Así como, comparar el espacio utilizado y porcentaje de llenado en las páginas de disco.

En el presente informe se prueban dos heurísticas de inserción, *Greene's Split* y *Linear Split*, con un número n de inserciones, con $n \in \{2^9, ..., 2^{25}\}$. Por cada conjunto de tamaño n insertado en el *RTree,* se genera $n/10$ rectángulos al azar para hacer consultas de búsqueda, documentando en cada caso el tiempo y cantidad promedio de accesos a disco.

2. Marco teórico

Un *R-Tree* es un árbol estable, similar a un *B-Tree* que permite almacenar, buscar y representar información multidimensional. Estos árboles son frecuentemente utilizados para la búsqueda e inserción de rectángulos (información en 2 dimensiones), un ejemplo de uso en la realidad es la localización de objetos indexados geográficamente. Por ejemplo, se podría usar un *R-Tree* para encontrar todos los hospitales en un radio de diez kilómetros alrededor de la posición actual.

Cada nodo de este árbol puede almacenar un valor máximo de M rectángulos, (por lo general M es el tamaño de una página de disco), y un mínimo de m. Debido a estas restricciones, un nodo puede sufrir un rebalse, al insertar el dato M+1 sobre éste. Por lo que se debe manejar el rebalse, dividiendo el nodo con *overflow* en dos nuevos nodos, utilizando diferentes heurísticas de *splitting*, para mantener la condición. Se utilizan las siguientes heurísticas para el manejo de *overflow*:

Linear Split: Se escogen los 2 rectángulos más alejados como miembros iniciales de cada nuevo nodo. Luego, uno por uno, se insertan los rectángulos en el nodo cuyo *MBR* experimente el menor aumento. Se debe tener cuidado de que ambos nodos tengan al menos m rectángulos.

Greene's Split: Se eligen los 2 rectángulos más distantes, de la misma forma que en *Linear Split*. Luego se define la dirección de corte, como el eje perpendicular al eje que tenía la mayor distancia normalizada. Se ordenan los M + 1 rectángulos según esta dirección (por ejemplo lado inferior en eje y). Una vez ordenados, los primeros $M/2 - 1$ son insertados en el primer nodo, el resto se insertan en el segundo.

Otros invariantes presentes en la estructura son:

- Cada nodo representa un *MBR*, que es el rectángulo más pequeño que contiene los rectángulos de sus hijos.
- Los rectángulos que corresponden a datos, se almacenan en las hojas del árbol. Estos nodos también representan el *MBR* de estos rectángulos.
- Todas las hojas se encuentran a la misma profundidad.

3. Hipótesis

Se espera que en la búsqueda la heurística con mejor desempeño sea Greene's split. Comparando los algoritmos, ambos eligen de igual manera los miembros iniciales de los nuevos nodos, pero, se diferencian en como distribuyen los rectángulos restantes en estos nodos. Por un lado, *Linear Split* elige al azar el próximo rectángulo, mientras que *Greene's split*, ordena los rectángulos según un eje e inserta la mitad de los rectángulos en un primer nodo, y la otra mitad en el segundo. Luego, se espera que las búsquedas en un árbol que ocupe la heurística *Greene's split* sean más rápidas, ya que tiende a insertar rectángulos cercanos en el mismo nodo, cada vez que ocurre un *overflow*. Además, como se ordenan los rectángulos en memoria principal no debiese considerarse este aspecto al calcular el tiempo del algoritmo. Bajo el mismo argumento, *Greene's* split debería tener menos accesos a disco en promedio, que *linear split*.

Con respecto al tiempo de la construcción del árbol, se espera que estos valores no sean muy distintos entre las heurísticas. Se espera que las inserciones cuesten muy poco cuando no sea necesario hacer accesos a disco, o todos los datos estén en un solo nodo. Mientras que para datos grandes, el tiempo va a ser proporcional a la altura del árbol, ya que se deben traer por lo menos $O(h)$ con $h = log_M n$ (n número de hojas), páginas de disco, lo cual es la operación más costosa de la inserción.

Se espera que el porcentaje de llenado de las páginas de disco sea alto mientras el dataset es pequeño, ya que los split disminuyen este porcentaje. Sin embargo con un n muy grande, los datos insertados ya no serán muy distintos unos de otros por lo cual el árbol será más denso, con una tendencia a mantener estable el espacio usado.

4. Diseño

La solución implementada es desarrollada en el lenguaje Java, aprovechando que éste tiene la capacidad de exportar objetos en forma de archivos con la función **ObjetOutputStream**, y cargarlos nuevamente en memoria con **ObjetInputStream**. El principal desafío de trabajar con Java es lidiar con su *garbage collector*, ya que para cerciorarse de que una variable sea destruida por el recolector de basura debemos encerrarla dentro de *scopes*, o pasarle una referencia *null*, para que ya no ocupe espacio en memoria.

La solución implementada cuenta con dos clases abstractas; **AbstractRectangulo** y **AbstractNodo** que implementan las interfaces **IRectangulo** e **INodo**, respectivamente. La primera clase representa un rectángulo interno a un nodo, que puede ser de tipo **Dato** o **MBR**, mientras que un nodo puede ser del tipo **NodoHoja** o **NodoInterno** y contiene un arreglo de *IRectangulos*. Las clases que extienden *AbstractNodo* tienen un campo entero **Id** que representa el nombre del archivo donde se guarda este nodo en disco. Mientras que las clase *MBR* también tiene un campo entero *Id*, pero este corresponde al nodo hijo al cual apunta este *MBR*.

4.1. Inserción en R-Trees

Para la inserción de rectángulos en un *RTree*, se implementa una búsqueda *DFS*, usando una pila en memoria principal. La pila permite guardar una referencia al camino tomado para llegar a una hoja, lo cual será útil al momento de arreglar conflictos de tipo *overflow* o de invariantes. Para lograr esto, la pila guarda referencias a un nodo, que corresponden a pares con los valores **(Id: int, Visititado: booleano)**, donde el parámetro *visitado* indica si el algoritmo ya ha traído el nodo a memoria principal en la anterioridad, e *Id* representa el identificador del nombre del archivo de dicho nodo en disco. El algoritmo sigue la siguiente lógica:

- Inicialmente la pila tiene un sólo elemento, que corresponde a una referencia a la raiz.
- Mientras la pila no esté vacía se busca una hoja, cargando un nodo en memoria con **stack.peek()**. Si se trata de un nodo interno, se marca como visitado, y se agrega a la pila el *Id* del rectángulo por el cual se debe descender, con el *booleano* visitado seteado en *false*.
- Si el nodo en memoria es una hoja, entonces se inserta el dato en ésta, y se saca su referencia de la pila con **stack.pop()**. Si se pudo insertar el dato en la hoja sin problemas entonces retornamos, si no, n momento todos los nodos de la pila han sido visitados, por lo cual debemos propagar *overflows* o conflictos producto de esta inserción.
- Para manejar los problemas luego de la inserción continuamos cargando nodos a memoria con *stack.peek()*, los nodos cargados corresponden a aquellos por los cuales se descendió. Propagamos los cambios al nodo, lo guardamos en disco, y quitamos su referencia de la pila con *stack.pop()*. Continuamos con el siguiente nodo si es que todavía existen conflictos y quedan nodos en la pila.

4.2. Implementación de Heurísticas

En primer lugar, para ambas heurísticas se debe elegir los rectángulos más distantes en el nodo. Para hacer esto se usan cuatro rectángulos auxiliares que almacenan el rectángulo que tiene su lado derecho más a la izquierda, el que tiene lado izquierdo más a la derecha, lo mismo para los lados superior e inferior, esto se decide en base a comparaciones de los lados de los rectángulos del nodo que tiene M+1 elementos. Luego se normaliza por el rango del eje (distancia entre los lados más alejados del eje) y nos quedamos con quienes tengan mayor distancia normalizada en un mismo eje.

Linear Split: Se escogen los 2 rectángulos iniciales para cada nodo como se explico anteriormente. Luego, uno por uno, se insertan los rectángulos en el nodo cuyo *MBR* experimente el menor aumento en su área, esto se hizo usando la función *mbr.difArea(rec)* que entrega cuánto tuvo que crecer *mbr* para agregar el rectángulo *rec* . Se debe tener cuidado de que ambos nodos tengan al menos m rectángulos. Por lo que si en algún momento, la suma entre la cantidad de elementos por insertar y la cantidad de elementos en uno de los dos nodos es igual a m, se insertan todos los elementos a ese nodo, independientemente de cuanto crece el *mbr*.

Greene's Split: Una vez elegido los rectángulos, nos fijamos en el eje que se está considerando. Luego se mantiene la dirección de corte con un booleano verdadero para el eje y, y falso para el eje x. Se ordenan los M + 1 rectángulos usando *MergeSort* según el booleano. Una vez ordenados los primeros $M/2 - 1$ son insertados en el primer nodo, el resto se insertan en el segundo.

4.3. Búsqueda en R-Trees

De forma similar a la inserción se utiliza DFS para recorrer el árbol y traer todos los rectángulos contenidos por el rectángulo dado. También utilizamos una pila para implementar la versión iterativa del algoritmo, con los siguientes pasos:

- Recibimos el rectángulo dado para la búsqueda como *mbr*.
- Inicializamos *datos* como un arreglo vacío.
- Insertamos el id de la raíz en la pila.
- Mientras la pila no este vacía:
  - Sacamos el primer elemento de la pila con *pila.pop()*
  - Cargamos el nodo en memoria.
  - Si es una hoja entonces agregamos todos los rectángulos que esten contenidos en *mbr* a *datos*.
  - Si es un nodo interno entonces apilamos todos los ids de los rectángulos que intersectan a *mbr*.
- Retornamos *datos*.

5. Diseño experimental

5.0.1. Medidas de rendimiento

Para realizar los experimentos de rendimiento, se crea un programa *Main* que crea datos aleatorizados que serán insertados sobre los dos *RTree*, uno instanciado con heurística *linear* y otro con *Greene*. Estos datos son creados con vértices en el rango [0, 500000], y ancho y largo en [0, 100]. El programa *Main* hace lo siguiente :

- Antes de realizar las inserciones sobre los árboles finales, se realiza una inserción *dummy* de 2^{10} datos para alcanzar *warmstate*.
- En un ciclo *for* se itera sobre $i \in \{7, ..., 17\}$, dentro de este ciclo hay un *for* anidado que itera para $j < 2^i - 2^{i-1}$, creando datos y agregándolos a una lista
- Luego se insertan los datos en el árbol actual, contando el tiempo con **System.currentTimeMillis()**.
- Sobre cada uno de estos árboles se realiza la búsqueda, y se cuenta el tiempo de la misma forma que en la inserción.
- Se obtiene los milisegundos transcurridos restando al tiempo final, el tiempo de inicio.

5.1. Supuestos de ejecución

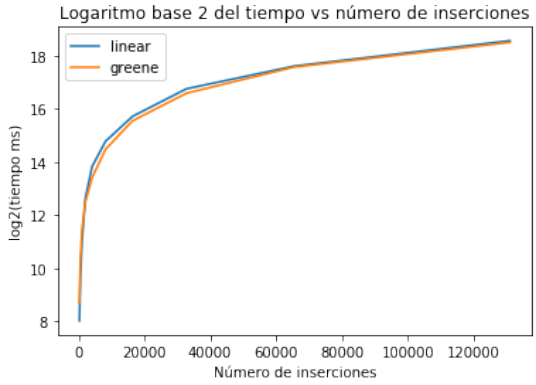
.
Los experimentos se ejecutan en un computador con las siguientes características:

- Sistema Operativo: Ubuntu 18.04 64 bits
- RAM: 12 GB
- Tamaño del disco: partición de 196 GB
- Tipo de Disco: Disco mecánico HDD (SATA), 5400 RPM.
- Tamaño página de disco 4096 bytes ó 4kB
- Compilador de la JVM: java version 1.8.0_191

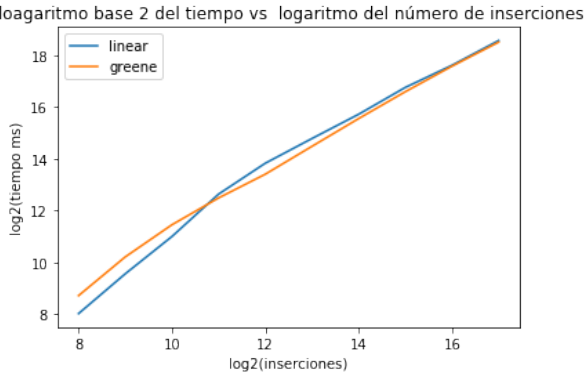
Como se está usando un *FileOutputStream*, se debe verificar cuantos bytes de memoria usa el objeto AbstractNodo, para verificar que no use más de 4 KB en disco. Se obtiene que AbstractNodo ocupa $y(x) = 26 * x + 245$ bytes en disco, donde x es el número de rectángulos insertados. por lo que $M = \frac{(4096-245)}{26} \sim 148$. Debido a esto, n siempre será lo suficientemente grande para tener que hacer accesos a disco, por lo que para percibir los cambios entre trabajar en disco y en memoria, se trabaja parte insertando un dataset de 2^7 rectángulos. Por otro lado, dado las capacidades del disco del computador utilizado, sólo se realizan inserciones en los intervalos $\{2^7, ..., 2^{17}\}$

5.2. Resultados

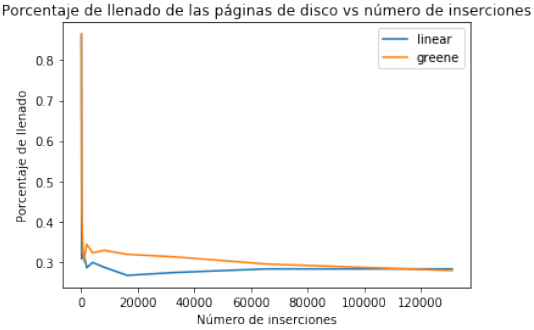
5.2.1. Inserciones vs tiempo y llenado de disco



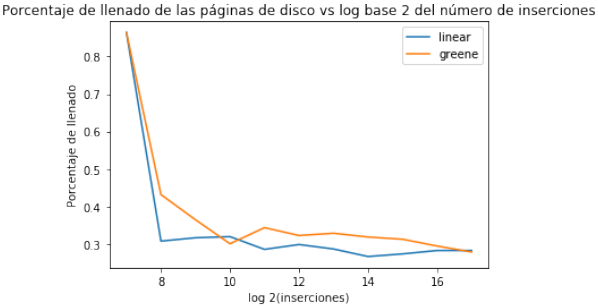
(a)



(b)



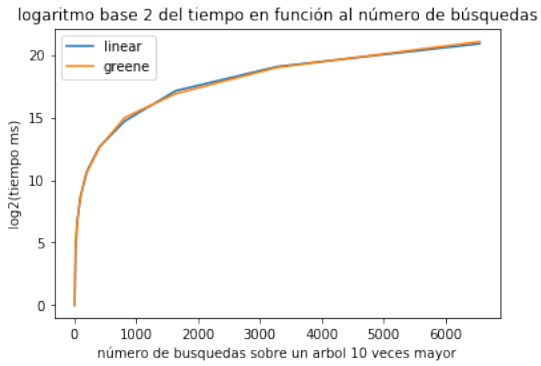
(c)



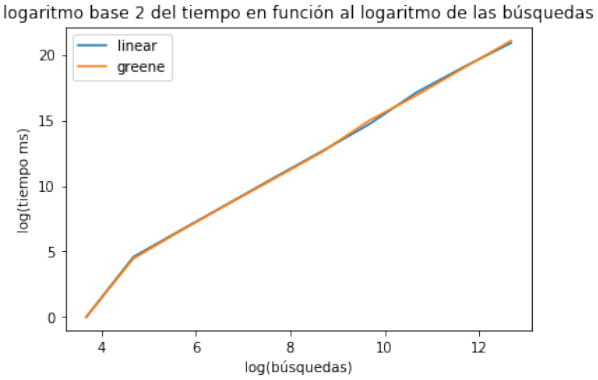
(d)

En los gráficos a y b se puede apreciar que *Greene's split* está por debajo de *Linear split*, lo cual indica que demora menos tiempo en crear el árbol, sin embargo esta diferencia no es muy significativa.
También se aprecia que en los gráficos c y d la curva de *Greene's split* está por sobre la curva de *Linear Split*, por lo que la primera heurística tiene un mejor manejo de las páginas en disco que la segunda.

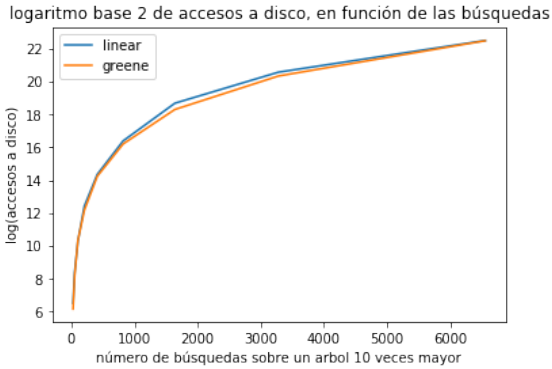
5.3. Número de búsquedas vs tiempo y accesos a disco



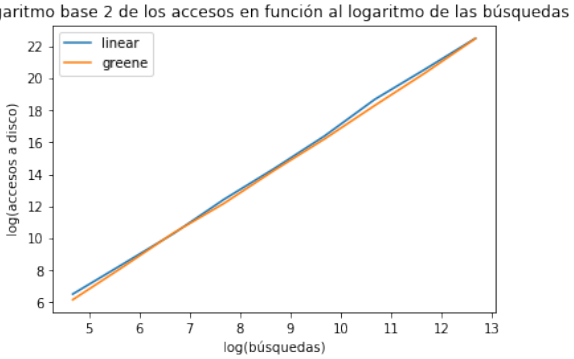
(a)



(b)



(c)



(d)

En cuanto a la búsqueda, no se distingue muy bien cuál de los dos métodos es mejor (gráficos a y b). Mientras que en el número de accesos a disco se aprecia que el método *Greene's split* tiene un mejor desempeño, como se planteó en la hipótesis inicial.
Cabe destacar que en los gráficos de tiempo contra búsqueda, al igual que en los de tiempo contra inserciones, cuando el dataset es muy pequeño, el tiempo es cercano a cero, ya que todas las operaciones se realizan en memoria principal.

6. Análisis e Interpretación

Los gráficos de la sección anterior nos muestran que ambas heurísticas para el manejo de overflow tienen un comportamiento lineal en cuanto al tiempo que demora la inserción de los elementos, es decir, el tiempo que toman en insertar todos los datos es directamente proporcional al número de estos.

Por otra parte, los gráficos de cantidad de accesos a disco vs cantidad de búsquedas también nos muestran que estas variables poseen una relación directamente proporcional.

Finalmente, los gráficos ligados al porcentaje de llenado de las páginas de disco señalan que en un principio (menor cantidad de datos) estas están llenas casi en su totalidad, pero que rápidamente, al aumentar la cantidad de datos, disminuye drásticamente la cantidad de espacio utilizado. En este apartado podemos ver que Greene's split conserva sus páginas más llenas que linear split, lo que implica que el primero tiene sus datos menos dispersos y explica que sea mejor en la búsqueda e inserción de datos, ya que se debe acceder a menos nodos para rescatar los datos que se buscan y llegar a la hoja en la que se quiere insertar un nuevo dato.

Podemos ahora analizar a qué podría deberse el comportamiento de estas heurísticas. Ciertamente estos algoritmos comparten la forma en la que eligen los rectángulos iniciales para los nuevos nodos, al realizar la partición del nodo original que sufrió el overflow, pero difieren en la forma de repartir los rectángulos restantes, de forma que Green's split asignaría la misma cantidad de rectángulos a los dos nodos nuevos (uno de ellos tiene solo uno adicional), mientras que linear split podría dejar a los nodos un tanto más desbalanceados respecto a la cantidad de rectángulos, este último punto debe ser la causa de las diferencias en los gráficos para el llenado de páginas. Por otra parte, como planteamos en nuestra hipótesis, *Greene's Split* al mantener los rectángulos más cercanos juntos induce a que una búsqueda deba acceder a menos nodos para recuperar aquellos rectángulos que están juntos y contenidos dentro del rectángulo de búsqueda, lo que hace que esta sea más rápida.

7. Conclusiones

Mediante los experimentos anteriores se prueba que *Greene's Split* es una mejor heurística que *Linear split*, teniendo mejor manejo de disco, y mejores tiempos de ejecución.

Además se verifica que existe una relación lineal entre el tiempo de ejecución y el número de inserciones. Esto se debe a que el número de I/O es del orden de la altura O(h) para la inserción, con una cota inferior $\Omega(2h)$ ya que en el peor caso debe descender y ascender por el árbol. Mientras que para la búsqueda esta cota es del orden O(kh) con k una constante, ya que depende del número de intersecciones, la cantidad de caminos que se deben tomar.
En ambos casos se verifica que la búsqueda es más costosa que la inserción ya que se deben traer más páginas de disco.

Tras finalizar esta entrega se toma en cuenta que las operaciones en disco son extremadamente costosas, y utilizar optimizaciones puede mejorar en gran medida su rendimiento.