Laboratorio 04 EDA

Pablo Luis Carazas Barrios

November 2023

1 Introduction

En el ámbito de las estructuras de datos, se han propuesto numerosas soluciones con el objetivo de lograr búsquedas rápidas y eficientes, a la vez que mantienen una organización efectiva de los datos. El R-tree ha sido una solución inicialmente prometedora, pero a medida que la investigación avanzaba, se descubrió que era posible mejorar la propuesta original del R-tree. Como resultado, se creó el R*-tree, una versión mejorada del R-tree diseñada para mejorar aspectos como consultas, consumo de memoria y otros.
En este informe, se llevará a cabo una comparación entre la función Insert original del R*-tree y una versión propuesta, en busca de identificar similitudes y diferencias clave entre ambas implementaciones.

2 Insert Original

La implementación de la función `Insert` presentada en el paper original 'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles' tiene la siguiente forma:

Algorithm Insert Data
1D1 Llamar `Insert` desde un nivel hoja.

Algorithm Insert
I1 Llamar `ChooseSubTree` poniendo el nivel como parametro para así encontrar un nodo N apropiado donde poner E
I2 Si N tiene menos de M entradas colocar E en N. Si N tiene M entradas llamar a `OverflowTreatment` con el nivel N como parametro
I3 Si se llamó a `OverflowTreatment` y se realizó algún split progpagar `OverflowTreatment` si es necesario, Si `OverflowTreatment` hizo split en una raiz se debe crear una nueva raiz.
I4 Ajustar todos los MBR correspondientes

Algorithm ChooseSubtree
CS1 Establecer N como el nodo raíz.
CS2 Si N es una hoja,
retornar N
de lo contrario,
Si los punteros a los hijos en N apuntan a hojas,
elegir la entrada en N cuyo rectángulo necesita la menor superposición para incluir el nuevo rectángulo de datos. Resuelve empates eligiendo la entrada cuyo rectángulo necesita la menor ampliación de área y luego la entrada con el rectángulo de menor área.
de lo contrario,
elige la entrada en N cuyo rectángulo necesita la menor ampliación de área para incluir el nuevo rectángulo de datos. Resuelve empates eligiendo la entrada con el rectángulo de menor área.
CS3 Establecer N como el nodo hijo señalado por el puntero al nodo hijo de la entrada elegida. Repetir desde CS2.

Algorithm OverflowTreatment
OT1 Si el nivel no es el nivel raíz y esta es la primera llamada a `OverflowTreatment` en el nivel dado

durante la inserción de un rectángulo de datos, entonces
invocar `Reinsert`
de lo contrario,
invocar `Split`
fin

Algorithm Split
S1 Invocar `ChooseSplitAxis` para determinar el eje, perpendicular al cual se realiza la división.
S2 Invocar `ChooseSplitIndex` para determinar la mejor distribución en dos grupos a lo largo de ese eje.
S3 Distribuir las entradas en dos grupos.

Algorithm ChooseSplitAxis
CSA1 Para cada eje,
Ordenar las entradas primero por el valor inferior y luego por el valor superior de sus rectángulos y determinar todas las distribuciones como se describe anteriormente. Calcular S, la suma de todos los valores de margen de las diferentes distribuciones.
fin
CSA2 Elegir el eje con el valor mínimo de S como eje de división.

Algorithm ChooseSplitIndex
CSA1 A lo largo del eje de división elegido, elegir la distribución con el valor de superposición mínimo. Resolver empates eligiendo la distribución con el valor de área mínimo.

Algorithm Reinsert
RI1 Para todas las entradas M+1 de un nodo N, calcular la distancia entre los centros de sus rectángulos y el centro del rectángulo delimitador de N.
RI2 Ordenar las entradas en orden decreciente de sus distancias calculadas en RI1.
RI3 Eliminar las primeras p entradas de N y ajustar el rectángulo delimitador de N.
RI4 En la ordenación definida en RI2, comenzando con la distancia máxima (= reinserción lejana) o la distancia mínima (= reinserción cercana), invocar `Insert` para reinserir las entradas.

En resumen la función `Insert` desempeña un papel fundamental en la inserción eficiente de nuevos datos. Comienza eligiendo un nodo apropiado mediante `ChooseSubTree` y, en caso de que el nodo esté lleno, se activa `OverflowTreatment`, que decide si reinsertar o dividir el nodo. La función `Split` se encarga de la división, determinando el eje de corte y distribuyendo las entradas. Este proceso asegura que los Rectángulos delimitadores mínimos (MBR) se ajusten correctamente. En caso de necesitar reinserción, `Reinsert` maneja la ubicación óptima de las entradas. En conjunto, estas funciones garantizan una gestión eficiente y equilibrada de datos en un R*-tree, lo que es esencial para consultas espaciales efectivas.

3 Codigo

A continuacion se mostrará la plantilla base que usa la clase RStarTree

```
template <typename LeafType, std::size_t dimensions,
    std::size_t min_child_items, std::size_t max_child_items>
class RStarTree {
public:
    // Métodos
    RStarTree(); // Constructor
    ~RStarTree(); // Destructor
    void insert(const LeafType leaf, const BoundingBox& box); // Insertar un objeto
    std::vector<LeafWithConstBox> find_objects_in_area(const BoundingBox& box); // Buscar objetos en
    void delete_objects_in_area(const BoundingBox& box); // Eliminar objetos en una región
    void write_in_binary_file(std::fstream& file); // Guardar en archivo binario
```

```
    void read_from_binary_file(std::fstream& file); // Leer desde archivo binario

private:
    // Atributos
    Node* tree_root; // Nodo raíz del árbol
    std::size_t size_; // Número de hojas
    std::unordered_set<int> used_deeps; // Profundidades utilizadas

    // Clases internas
    struct Node; // Nodo interno del árbol
    struct Leaf; // Hoja del árbol
    struct SplitParameters; // Parámetros para dividir nodos

    // Métodos privados
    Node* choose_leaf_and_insert(Leaf* leaf, Node* node, int deep = 0);
    Node* choose_node_and_insert(Node* node, Node* parent_node, int required_deep, int deep = 0);
    Node* choose_subtree(Node* node, const BoundingBox& box);
    Node* overflow_treatment(Node* node, int deep);
    Node* split(Node* node);
    void forced_reinsert(Node* node, int deep);
    SplitParameters choose_split_axis_and_index(Node* node);
    void delete_tree(Node* node);
    void write_node(Node* node, std::fstream& file);
    void write_leaf(Leaf* leaf, std::fstream& file);
    Node* read_node(std::fstream& file);
    Leaf* read_leaf(std::fstream& file);
};
```

3.1 Estructuras Internas

3.1.1 Cuadro Delimitador (BoundingBox)

El **Cuadro Delimitador** se utiliza para representar un rectángulo que delimita objetos en un espacio de múltiples dimensiones. Contiene coordenadas mínimas y máximas en cada dimensión y se utiliza para definir el espacio ocupado por un grupo de objetos.

3.1.2 Parte del Árbol (TreePart)

La clase base **Parte del Árbol (TreePart)** se utiliza para representar tanto nodos internos como hojas en el árbol R*-Tree. Cada `TreePart` tiene un `BoundingBox` asociado para describir el espacio que abarca.

3.1.3 Nodo (Node)

La clase **Nodo (Node)** representa un nodo interno en el árbol R*-Tree. Contiene una colección de punteros a otras instancias de `TreePart`, que pueden ser nodos internos o hojas. Además, tiene un indicador llamado `hasLeaves` que indica si sus hijos son hojas o nodos internos.

3.1.4 Hoja (Leaf)

La clase **Hoja (Leaf)** representa una hoja en el árbol R*-Tree. A diferencia de los nodos internos, una hoja contiene un valor de objeto. Las hojas son las unidades finales en las que se almacenan los datos reales.

3.1.5 Parámetros de División (SplitParameters)

La estructura **Parámetros de División (SplitParameters)** se utiliza para almacenar información relacionada con la división de nodos. Puede incluir detalles como el índice óptimo para la división,

el eje de división preferido y el tipo de división (superior o inferior). Estos parámetros son esenciales para dividir nodos de manera efectiva y mantener la estructura del árbol.

Estas estructuras internas son fundamentales para la implementación del árbol R*-Tree y son utilizadas para gestionar y organizar eficazmente los datos en el espacio multidimensional. Las hojas almacenan los datos reales, mientras que los nodos internos actúan como organizadores y proporcionan una estructura jerárquica para acelerar las consultas espaciales. Los cuadros delimitadores (BoundingBox) son esenciales para definir y gestionar el espacio ocupado por los objetos en el árbol.

3.2 Metodos utilizados por Insert

3.2.1 Función `insert`

La función `insert` desempeña un papel crucial en la inserción eficiente de nuevos datos en el árbol R*-Tree. Su lógica se puede resumir de la siguiente manera:

- Incrementa el contador de tamaño del árbol.
- Crea un nuevo objeto `Leaf` con el valor del objeto a insertar y su área delimitadora.
- Si el árbol está vacío, crea un nuevo nodo raíz y agrega la hoja al nodo raíz.
- Si el árbol no está vacío, utiliza la función `choose_leaf_and_insert` para determinar dónde insertar el nuevo elemento en el árbol.
- Realiza un seguimiento de las profundidades utilizadas en el proceso de inserción y las borra al final.

La función `insert` es esencial para garantizar una inserción eficiente de datos en el árbol R*-Tree y para mantener su estructura organizada y equilibrada.

3.2.2 Función `choose_leaf_and_insert`

La función `choose_leaf_and_insert` es un componente esencial para la inserción eficiente de datos en un árbol R*-Tree. Su lógica se puede resumir de la siguiente manera:

- Actualiza el cuadro delimitador del nodo `node` al extenderlo para incluir el cuadro delimitador de la hoja `leaf`.
- Si `node` tiene hojas como hijos, agrega la hoja `leaf` al conjunto de hojas dentro del nodo.
- Si `node` no tiene hojas como hijos, llama recursivamente a `choose_leaf_and_insert` para determinar la ubicación adecuada para insertar la hoja `leaf` en el subárbol descendiente. La ubicación se determina utilizando la función `choose_subtree`.
- Si la llamada recursiva devuelve un nuevo nodo en lugar de una hoja, agrega este nuevo nodo al conjunto de hijos del nodo actual.
- Si el número de elementos en el nodo supera `max_child_items`, se activa la función `overflow_treatment` para dividir el nodo, asegurando que el árbol se mantenga balanceado.
- Devuelve `nullptr` si no se realizó ninguna división o `new_node` si se insertó un nuevo nodo en lugar de una hoja.

La función `choose_leaf_and_insert` utiliza la función `choose_subtree` para determinar la ubicación óptima para insertar la hoja `leaf`, lo que es fundamental para garantizar una inserción eficiente en el árbol R*-Tree y mantener su estructura equilibrada.

3.2.3 Función `choose_subtree`

La función `choose_subtree` desempeña un papel esencial en la inserción de nuevos elementos en un árbol R*-Tree. Su lógica se puede resumir de la siguiente manera:

- La función toma como argumento un nodo y un cuadro delimitador (`BoundingBox`) que representa la región en la que se desea insertar un nuevo elemento.
- Comienza por crear un vector de punteros llamado `overlap_preferable_nodes` que almacenará los nodos que tienen superposición con el cuadro delimitador dado.
- Si el nodo actual tiene hojas como hijos, se busca el nodo con la menor superposición con el cuadro. Si hay varios nodos con la misma superposición mínima, se selecciona el que tenga la menor ampliación de área.
- Si los hijos del nodo no son hojas, se mantienen todos en el vector `overlap_preferable_nodes`.
- A continuación, se busca un nodo que minimice el aumento de área al agregar el cuadro delimitador dado.
- Se selecciona el nodo con la menor ampliación de área. Si hay varios nodos con la misma ampliación mínima, se elige el que tenga el área más pequeña.
- Finalmente, si aún hay varios nodos con la misma área mínima, se selecciona el que tenga el área más pequeña de todos.

La función `choose_subtree` juega un papel crucial en la determinación del lugar óptimo para insertar un nuevo elemento en el árbol R*-Tree. Esto asegura que los nuevos elementos se agreguen de manera eficiente y se mantenga la organización de la estructura del árbol.

3.2.4 Función `overflow_treatment`

La función `overflow_treatment` desempeña un papel fundamental en el manejo de situaciones de desbordamiento de nodos en un árbol R*-Tree. Su lógica se puede resumir de la siguiente manera:

- La función toma como argumentos un nodo y una profundidad (`deep`).
- Comienza verificando si la profundidad actual (`deep`) no se ha utilizado previamente en el proceso de inserción y si el nodo no es la raíz del árbol. Si se cumplen estas condiciones, se procede a realizar una reinserción forzada (`forced_reinsert`).
- En caso de que no se realice una reinserción forzada, se divide el nodo actual (`split(node)`) para eliminar los hijos innecesarios y obtener una ubicación con ellos.
- Si el nodo es la raíz del árbol, se agrega un nivel adicional al árbol. Se crea un nuevo nodo que se convierte en la nueva raíz, y los nodos originalmente raíz y el recién dividido se convierten en hijos de la nueva raíz.
- En otros casos, se devuelve la ubicación donde se deben insertar los hijos en el arreglo del padre del nodo.

La función `overflow_treatment` es esencial para mantener el equilibrio y la organización del árbol R*-Tree cuando se supera el número máximo de elementos en un nodo. Puede llevar a cabo reinserciones o divisiones de nodos según sea necesario, y garantiza que la estructura del árbol se mantenga adecuadamente.

3.2.5 Función `forced_reinsert`

La función `forced_reinsert` juega un papel clave en el proceso de reinserción forzada de algunos de los hijos de un nodo en el árbol R*-Tree. Su lógica se puede resumir de la siguiente manera:

- La función toma como argumentos un nodo y una profundidad (`deep`).
- Calcula un valor de porcentaje (`p`) que representa la fracción de hijos que se eliminarán del nodo actual.
- Determina el número de hijos que serán eliminados, que es el producto del tamaño total de los hijos del nodo y el valor de porcentaje (`p`).
- Ordena los hijos del nodo según la distancia entre sus centros y el centro del cuadro delimitador del nodo (`node`).
- Selecciona y almacena los hijos que serán eliminados en un vector llamado `forced_reinserted_nodes`.
- Elimina los hijos seleccionados del nodo y ajusta su cuadro delimitador.
- Marca la profundidad (`deep`) como utilizada en el proceso de inserción.
- Actualiza el cuadro delimitador del nodo con los hijos restantes.
- Si los hijos del nodo son hojas, se vuelven a insertar utilizando el método `choose_leaf_and_insert` en la raíz del árbol.
- Si los hijos son nodos internos, se vuelven a insertar en el árbol utilizando el método `choose_node_and_insert` hasta una profundidad específica.

  La función `forced_reinsert` se encarga de optimizar el espacio y la organización del árbol mediante la eliminación y reinserción de algunos de los hijos del nodo, lo que ayuda a evitar situaciones de desbordamiento y a mantener una estructura equilibrada.

3.2.6 Función `choose_node_and_insert`

La función `choose_node_and_insert` desempeña un papel fundamental en la inserción de nuevos nodos internos en el árbol R*-Tree. Su lógica se puede resumir de la siguiente manera:

- La función toma como argumentos un nodo interno (`node`), el nodo padre (`parent_node`), una profundidad requerida (`required_deep`), y una profundidad actual (`deep`).
- Ajusta el cuadro delimitador del nodo padre (`parent_node`) para incluir el cuadro delimitador del nodo interno (`node`).
- Si la profundidad actual (`deep`) es igual a la profundidad requerida (`required_deep`), lo que indica que se ha alcanzado la profundidad deseada en el árbol, agrega el nodo interno al vector de hijos del nodo padre. Esto se utiliza para mantener el árbol perfectamente equilibrado.
- Si la profundidad actual es menor que la profundidad requerida, la función sigue descendiendo en el árbol, utilizando la función `choose_subtree` para encontrar el nodo hijo adecuado y llamándose recursivamente a sí misma hasta alcanzar la profundidad requerida.
- Si la función no devuelve un nuevo nodo durante el proceso de inserción, se retorna `nullptr`.
- Si la cantidad de hijos del nodo padre supera `max_child_items`, se activa la función `overflow_treatment` para dividir el nodo padre.
- Finalmente, se retorna `nullptr` si no se ha realizado una división en el nodo padre.

  La función `choose_node_and_insert` se encarga de gestionar la inserción de nodos internos en el árbol R*-Tree y garantiza que la estructura del árbol se mantenga equilibrada y eficiente para consultas espaciales.

3.2.7 Función `split`

La función `split` es esencial en la división de nodos en el árbol R*-Tree, lo que contribuye a mantener su estructura equilibrada y eficiente para consultas espaciales. Su lógica se puede resumir de la siguiente manera:

- La función toma un nodo interno (`node`) como entrada.
- Utiliza la función `choose_split_axis_and_index` para seleccionar el índice y el eje óptimos para la división.
- Ordena los elementos del nodo según el eje y el tipo seleccionados en el paso anterior.
- Crea un nuevo nodo interno (`new_Node`) que actuará como uno de los nodos resultantes de la división.
- El nuevo nodo recibe una parte de los elementos del nodo original, comenzando desde el índice calculado en el paso 2. El número de elementos que se transfieren es determinado por la fórmula: `max_child_items + 1 - min_child_items - params.index`.
- Actualiza las propiedades del nuevo nodo, incluyendo si sus hijos son hojas o nodos internos, y ajusta su cuadro delimitador (`box`).
- Actualiza el cuadro delimitador del nodo original (`node`) y elimina los elementos que se transfirieron al nuevo nodo.
- La función retorna el nuevo nodo resultante de la división.

  La función `split` es crucial para dividir los nodos internos en el árbol R*-Tree, lo que garantiza que la estructura del árbol se mantenga balanceada y eficaz para la realización de consultas espaciales.

3.2.8 Función `choose_split_axis_and_index`

La función `choose_split_axis_and_index` desempeña un papel esencial en la selección del eje y el índice óptimos para dividir un nodo interno en el árbol R*-Tree. Su lógica se puede resumir de la siguiente manera:

- La función toma un nodo interno (`node`) como entrada.
- Itera a través de todas las dimensiones (ejes) del espacio de datos.
- Para cada dimensión, considera dos tipos de divisiones: una en el borde inferior del espacio y otra en el borde superior.
- Ordena los elementos del nodo (`node->items`) según el valor de la coordenada en el eje actual y el tipo de división (inferior o superior).
- Calcula las áreas y los márgenes de las dos regiones resultantes al dividir el conjunto de elementos en un punto específico (varía según `k`) en el rango entre `min_child_items` y `max_child_items`.
- Elige el punto de división que minimice la suma de márgenes de las dos regiones (`min_margin`).
- Registra el índice de división (`params.index`), el eje de división (`params.axis`) y el tipo de división (inferior o superior) (`params.type`) que resultaron en el menor margen.
- Retorna un objeto `SplitParameters` que contiene estos valores óptimos.

  La función `choose_split_axis_and_index` es crucial para determinar la estrategia de división más eficiente, lo que asegura que el árbol R*-Tree mantenga su estructura balanceada y efectiva para consultas espaciales.

4 Comparación

Para una comparación más precisa, es importante tener en cuenta que la estructura entre el código C++ y el seudocódigo difiere en algunos aspectos. Estas diferencias pueden dar lugar a cambios tanto simples como complejos en la lógica y en los pasos de ejecución.

4.1 Función `Insert`

La función `Insert` en el código C++ y el seudocódigo son similares en términos de la lógica general para la inserción de elementos en un árbol R*-Tree. Ambos siguen los siguientes pasos generales:

- Encuentran el nodo apropiado para insertar el nuevo elemento.
- Si el nodo tiene menos de M entradas, insertan el elemento en el nodo. Si el nodo tiene M entradas, invocan la función `OverflowTreatment` para manejar el desbordamiento (reinserción o división).
- Si `OverflowTreatment` causó la división de la raíz, crean una nueva raíz.
- Ajustan todos los rectángulos de cobertura en la ruta de inserción para que sean MBR que encierran los rectángulos de los hijos.

Diferencias clave:

- El código C++ incorpora un control de las profundidades utilizadas (`used_deeps`) y la actualización de los rectángulos de cobertura en la ruta de inserción, que no se mencionan explícitamente en el seudocódigo.

4.2 Función `ChooseSubtree`

La función `ChooseSubtree` en el código C++ y el seudocódigo siguen la misma lógica subyacente para elegir el nodo apropiado para la inserción. Ambos consideran la superposición de rectángulos y la ampliación de área. Las diferencias más notables son:

- El código C++ considera un caso especial si solo hay un nodo preferido con la menor superposición, mientras que el seudocódigo no menciona este caso explícitamente.
- El código C++ utiliza el tipo `axis_type` y las funciones `value_of_axis` para manejar el valor del eje, que no están presentes en el seudocódigo.

4.3 Función `OverflowTreatment`

La función `OverflowTreatment` en el código C++ y el paper siguen una lógica similar de reinserción o división de nodos en caso de desbordamiento. Ambos manejan el caso de la raíz y aplican la estrategia de reinserción o división de manera recursiva. Las diferencias clave son:

- El código C++ verifica las profundidades utilizadas (`used_deeps`) y realiza la reinserción forzada (forced reinsert) en ciertas condiciones, mientras que el paper se refiere a la reinserción como "Reinsert."
- El código C++ incluye un caso especial para cuando el nodo es la raíz del árbol y se debe crear un nuevo nivel.

4.4 Función `Split`

La función `Split` en el código C++ y el paper son muy similares en términos de dividir un nodo en dos grupos a lo largo de un eje. Ambos calculan el eje y el índice óptimos para la división y redistribuyen las entradas. Las diferencias más notables son:

- El código C++ utiliza un objeto `SplitParameters` para almacenar información sobre el eje y el índice de división, mientras que el seudocódigo no describe una estructura similar explícitamente pero deja entendido que se puede dejar eso como un parametro o algo similar.
- El código C++ ordena los elementos en función del valor del eje, lo que no se especifica en el paper.

4.5 Función `ChooseSplitAxis`

La función `ChooseSplitAxis` en el código C++ y el paper determina el eje de división óptimo. Ambos utilizan una estrategia similar para minimizar el margen o la superposición. Las diferencias clave son:

- El código C++ utiliza el objeto `SplitParameters` para almacenar información sobre el eje y el tipo de división (inferior o superior). Esta información no se describe explícitamente en el paper.

4.6 Función `ChooseSplitIndex`

La función `ChooseSplitIndex` en el código C++ y el paper elige la distribución óptima en dos grupos a lo largo del eje de división. Ambos se centran en minimizar la superposición y el área. Las diferencias clave son:

- El código C++ utiliza el objeto `SplitParameters` para determinar el eje de división y el tipo de división (inferior o superior), mientras que el seudocódigo no describe esta estructura de manera explícita.

4.7 Función `Reinsert`

La función `Reinsert` en el código C++ y el paper tienen una lógica similar para reinserir elementos que fueron eliminados del nodo debido al desbordamiento. Ambos calculan distancias y reinserción selectiva de elementos. Las diferencias clave son:

- El código C++ utiliza el valor `p` para determinar el porcentaje de elementos que se eliminarán y reinserirán, mientras que el paper no establece un valor específico para esto.
- El código C++ utiliza el tipo `axis_type` y las funciones `dist_between_centers` y `value_of_axis`, funciones que no existen en el paper.

4.8 Función `ChooseNodeAndInsert`

Esta funcion no existe como tal en el codigo pero es una version adaptada de la funcion `ChooseSubtree` cumpliendo con lo necesario para esta version adaptada de la estructura:

- El código C++ incluye un control de las profundidades utilizadas (`used_deeps`).
- El código C++ utiliza el tipo `axis_type` y las funciones `stretch` y `value_of_axis`.

5 Conclusiones

Como se puede evidenciar, la implementacion que se nos ha dado para analizar tiene muchos factores adicionales que no son nombrados en el paper original, pero esto se puede deber a que como el paper muestra un pseudocodigo por lo que en idea es simple de entender pero este puede traer muchas cosas adicionales consigo siendo una de estas que el codigo genera y usa una estructura adicional que se encarga de analizar los cambios respecto a splits, esto puede facilitar muchos factores de implementacion, facilita el acceso y uso de los valores a tomar en cuenta.

Puede que exista mejores formas de implementarlo pero esta trata de cumplir con el algoritmo propuesto en el mismo paper con las diferencias encesarias para que funcione de manera eficiente para este lenguaje.

6 Bibliografia

Beckmann, N., Kriegel, H. P., Schneider, R., & Seeger, B. (1990, May). The R*-tree: An efficient and robust access method for points and rectangles. In Proceedings of the 1990 ACM SIGMOD international conference on Management of data (pp. 322-331).

Codigo Original del R*Tree