QUADRATIC SPLIT Y LINEAR SPLIT EN R-TREES

Mario Garrido

Universidad de Chile
Santiago, Chile

ABSTRACT

En el presente trabajo se estudia el desempeño de construcción y búsqueda en un *R-tree* utilizando 2 algoritmos de *split*: *Quadratic Split* y *Linear Split*. Como su nombre indica, el primer algoritmo tiene un costo cuadrático $\mathcal{O}(M^2)$, con M el número máximo de entradas de cada nodo, pero produce una estructura más compacta y con menos *overlaps*, lo que se traduce en menos accesos a disco, mientras que el algoritmo lineal es más rápido, pero el aumento de accesos a disco necesarios para concretar una búsqueda transfiere el costo en tiempo a esta última operación.

1. INTRODUCCIÓN

Los objetivos perseguidos en el presente trabajo son los siguientes:

- Realizar una implementación de *R-tree* en memoria secundaria.
- Realizar una implementación de 2 algoritmos de *split*: *Quadratic Split* y *Linear Split*, tal como fueron descritos por Guttman[1].
- Realizar experimentos que permitan comprender mejor los siguientes puntos:
  1. Costos de tiempo y espacio en la construcción del árbol para cada tipo de *split*.
  2. Porcentaje de llenado de las páginas de disco utilizadas para almacenar el árbol.
  3. Costos de tiempo y accesos a disco de la operación de búsqueda en el árbol.
- Concluir respecto a las hipótesis.

Las hipótesis a estudiar son las siguientes:

- Considerando que el costo de *Quadratic Split* es cuadrático en el parámetro *M*, debido a que realiza una operación cuadrática sobre el número de entradas a dividir en el nodo para encontrar el par que minimiza el área desperdiciada del *MBR* que alberga a ambos, se espera que este costo se traduzca en un árbol cuyas ramas tienen *MBRs* más compactos, lo que resulta en búsquedas más eficientes con menor cantidad de accesos a disco.
- Debido a la naturaleza aleatoria de *PickNext* para *Linear Split* se espera que el árbol construído tenga *MBRs* menos compactos y que esto resulte en una mayor cantidad de lecturas a disco necesarias para completar una operación de búsqueda. También se espera que este algoritmo sea mucho más rápido que el anterior para la construcción del árbol.

2. IMPLEMENTACIÓN

Para realizar la implementación se utilizó el lenguaje C. Se siguió la estructura descrita por Guttman en el trabajo original[1], por lo que existen, en ambas implementaciones, los algoritmos *PickNext*, *PickSeeds*, *Insert*, *Search*, *AdjustTree* y *ChooseLeaf*. Adicionalmente se construyeron métodos que permitieran almacenar en disco y leer desde disco los nodos y la estructura completa: *Get_Node*, *Save_Node*, *dump_tree* y *Load_Tree*.

Para realizar la escritura a disco del árbol se decidió guardar cada nodo en un archivo separado, conteniendo el archivo el *struct* completo correspondiente al nodo y sus campos, esto implica que se incurrió en un costo, por nodo, de 20 bytes, correspondiente al costo de los 4 enteros que guardan información crucial del nodo, y de *M**20 bytes, para almacenar los *Entry*. El tamaño de página del sistema utilizado es de 4096 bytes, por lo que el parámetro *M* se fijó en 203, resultando cada nodo en un peso final de 4080 bytes, con lo que, en condiciones de alineación ideales, solo sería necesario un acceso a disco para recuperar el nodo completo. Considerando que el parámetro *m* debe corresponder a un 40 % de *M* se fijó en 81.
Con el objetivo de simplificar las pruebas se consideró que el dato a almacenar por cada entrada sería un entero. De ser necesario almacenar un tipo de dato arbitrario se puede utilizar ese entero para referirse a un archivo en disco, tal como se hizo en las funciones *Get_Node* y *Save_Node*. Un detalle importante a considerar de la implementación actual es que no se almacena el nodo raíz en disco, este nodo se mantiene en el árbol, el cual se encuentra en memoria principal. En

caso de utilizar la función *dump_tree* se procede a guardar el nodo raíz y el *struct* Tree. Es un detalle importante ya que para algunas búsquedas se realizan 0 lecturas a disco, ya que es suficiente la información de las entradas de la raíz para desistir de la búsqueda.

3. QUADRATIC SPLIT

Para estudiar el comportamiento de la implementación se estudió la construcción de un árbol con $m = 2$ y $M = 3$. El método *print_basic_example()* inserta los siguientes valores al árbol:

1. $\{2.0, 2.0, 4.0, 4.0\}$
2. $\{3.0, 5.0, 5.0, 6.0\}$
3. $\{5.0, 2.0, 6.0, 3.0\}$
4. $\{5.0, 4.0, 7.0, 5.0\}$
5. $\{0.0, 0.0, 1.0, 1.0\}$
6. $\{1.0, 0.0, 2.0, 4.0\}$
7. $\{2.0, 0.0, 3.0, 5.0\}$
8. $\{1.0, 6.0, 2.0, 7.0\}$
9. $\{5.0, 0.0, 7.0, 1.0\}$
10. $\{6.0, 0.0, 7.0, 4.0\}$
11. $\{3.0, 0.0, 4.0, 1.0\}$
12. $\{6.0, 6.0, 7.0, 7.0\}$
13. $\{1.0, 1.0, 3.0, 3.0\}$
14. $\{2.0, 2.0, 3.0, 3.0\}$
15. $\{2.3, 3.3, 2.6, 3.7\}$

Correspondientes al valor del punto inferior izquierdo y punto superior derecho, respectivamente. Cada entrada contiene como dato el entero correspondiente a su puesto de inserción (e.g: $\{2.0, 0.0, 3.0, 5.0\}$ contiene el dato 7). Estas inserciones producen la estructura descrita en la figura 1. Cada línea representa un *Entry* en el nodo. *Root* 6 indica que es una entrada del nodo raíz, el cual corresponde al sexto nodo creado. En el caso de las hojas se agrega, además, la información del valor almacenado como hijo.

```
Root 6, (0.0000,0.0000),(3.0000,7.0000)
  INNER 2, (1.0000,0.0000),(2.0000,7.0000)
    LEAF 1, Stored: 8 (1.0000,6.0000),(2.0000,7.0000)
    LEAF 1, Stored: 6 (1.0000,0.0000),(2.0000,4.0000)
  INNER 2, (2.0000,0.0000),(3.0000,5.0000)
    LEAF 4, Stored: 7 (2.0000,0.0000),(3.0000,5.0000)
    LEAF 4, Stored: 14 (2.0000,2.0000),(3.0000,3.0000)
    LEAF 4, Stored: 15 (2.3000,3.3000),(2.6000,3.7000)
  INNER 2, (0.0000,0.0000),(3.0000,3.0000)
    LEAF 8, Stored: 5 (0.0000,0.0000),(1.0000,1.0000)
    LEAF 8, Stored: 13 (1.0000,1.0000),(3.0000,3.0000)
Root 6, (2.0000,0.0000),(7.0000,7.0000)
  INNER 5, (3.0000,4.0000),(7.0000,7.0000)
    LEAF 0, Stored: 4 (5.0000,4.0000),(7.0000,5.0000)
    LEAF 0, Stored: 2 (3.0000,5.0000),(5.0000,6.0000)
    LEAF 0, Stored: 12 (6.0000,6.0000),(7.0000,7.0000)
  INNER 5, (3.0000,0.0000),(7.0000,4.0000)
    LEAF 3, Stored: 9 (5.0000,0.0000),(7.0000,1.0000)
    LEAF 3, Stored: 10 (6.0000,0.0000),(7.0000,4.0000)
    LEAF 3, Stored: 11 (3.0000,0.0000),(4.0000,1.0000)
  INNER 5, (2.0000,2.0000),(6.0000,4.0000)
    LEAF 7, Stored: 1 (2.0000,2.0000),(4.0000,4.0000)
    LEAF 7, Stored: 3 (5.0000,2.0000),(6.0000,3.0000)
```

Fig. 1. Árbol después de las 15 inserciones con *Quadratic Split*

Se realizaron las siguientes búsquedas de prueba:

- *Query:* $\{1.0, 1.0, 3.0, 3.0\}$. Arroja como valores encontrados: 6, 7, 14, 5, 13, 11 y 1. Realiza 7 lecturas a disco.
- *Query:* $\{2.0, 2.0, 3.0, 3.0\}$. Arroja los valores encontrados: 6, 7, 14, 13 y 1. Realiza 7 lecturas a disco.
- *Query:* $\{2.3, 3.3, 2.6, 3.7\}$. Arroja los valores encontrados: 7, 15 y 1. Realiza 4 lecturas a disco.

4. LINEAR SPLIT

De forma análoga al caso anterior se verificó la estructura de la figura 2, para las mismas inserciones.

Las pruebas demostraron lo siguiente:

- *Query:* $\{1.0, 1.0, 3.0, 3.0\}$. Arroja como valores encontrados: 11, 6, 5, 7, 13, 14 y 1. Realiza 8 lecturas a disco.
- *Query:* $\{2.0, 2.0, 3.0, 3.0\}$. Arroja los valores encontrados: 6, 7, 13, 14 y 1. Realiza 8 lecturas a disco.
- *Query:* $\{2.3, 3.3, 2.6, 3.7\}$. Arroja los valores encontrados: 7, 15 y 1. Realiza 5 lecturas a disco.

Ya se aprecia, en esta etapa temprana, un incremento en la cantidad de lecturas necesarias.

```
Root 6, (3.0000,0.0000),(7.0000,7.0000)
  INNER 2, (3.0000,4.0000),(7.0000,7.0000)
    LEAF 0, Stored: 2 (3.0000,5.0000),(5.0000,6.0000)
    LEAF 0, Stored: 4 (5.0000,4.0000),(7.0000,5.0000)
    LEAF 0, Stored: 12 (6.0000,6.0000),(7.0000,7.0000)
  INNER 2, (3.0000,0.0000),(7.0000,4.0000)
    LEAF 1, Stored: 10 (6.0000,0.0000),(7.0000,4.0000)
    LEAF 1, Stored: 9 (5.0000,0.0000),(7.0000,1.0000)
    LEAF 1, Stored: 11 (3.0000,0.0000),(4.0000,1.0000)
Root 6, (0.0000,0.0000),(3.0000,7.0000)
  INNER 5, (1.0000,0.0000),(2.0000,7.0000)
    LEAF 3, Stored: 8 (1.0000,6.0000),(2.0000,7.0000)
    LEAF 3, Stored: 6 (1.0000,0.0000),(2.0000,4.0000)
  INNER 5, (0.0000,0.0000),(3.0000,5.0000)
    LEAF 4, Stored: 5 (0.0000,0.0000),(1.0000,1.0000)
    LEAF 4, Stored: 7 (2.0000,0.0000),(3.0000,5.0000)
    LEAF 4, Stored: 13 (1.0000,1.0000),(3.0000,3.0000)
Root 6, (2.0000,2.0000),(6.0000,4.0000)
  INNER 9, (2.0000,2.0000),(3.0000,3.7000)
    LEAF 8, Stored: 15 (2.3000,3.3000),(2.6000,3.7000)
    LEAF 8, Stored: 14 (2.0000,2.0000),(3.0000,3.0000)
  INNER 9, (2.0000,2.0000),(6.0000,4.0000)
    LEAF 7, Stored: 3 (5.0000,2.0000),(6.0000,3.0000)
    LEAF 7, Stored: 1 (2.0000,2.0000),(4.0000,4.0000)
```

Fig. 2. Árbol después de las 15 inserciones con *Linear Split*

5. RESULTADOS EXPERIMENTALES

Los experimentos fueron realizados en un Ultrabook HP, procesador I5-3317U, 8Gb de ram corriendo a 1600 Mhz y un disco SSD Crucial BX100 de 250Gb. El sistema operativo utilizado fue *Ubuntu* 14.04.3 y el compilador fue *GCC* 4.8.4. Los parámetros a medir, respecto a cada algoritmo de *split*, corresponden a:

- Tiempo de construcción del árbol.
- Espacio total utilizado en disco por la estructura.
- Porcentaje de llenado del árbol y de las hojas (Fill).
- Número total de lecturas a disco para la construcción del árbol.
- Número total de escrituras a disco para la construcción del árbol.
- Tiempo promedio de búsqueda.
- Número promedio de lecturas para encontrar las entradas.

Para cada iteración *i* se realiza lo siguiente:

1. Se genera un dataset de 2^i entries. [1]
2. Se inicializan los contadores designados para medir los parámetros a 0.[2]
3. Se crea el árbol y se le inserta todo el dataset, midiendo el tiempo en nanosegundos.[2]
4. Se calculan, una vez construído el árbol, el resto de los parámetros que no involucran a las búsquedas.[2]
5. Se asigna, nuevamente, las lecturas a 0.[2]
6. Se realizan las búsquedas y se miden los tiempos.[2]
7. Se calculan los parámetros restantes correspondientes a las búsquedas.[2]

Se utiliza $i \in \{9, 10, 11, ..., 22, 23, 24, 25\}$. Para generar el dataset se siguen las indicaciones dadas en el enunciado. Las búsquedas son de tamaño $\frac{n}{10}$, donde n es el número de entradas insertadas al árbol. La mitad de estas búsquedas corresponden a rectángulos que se encuentran insertos en el árbol y la otra mitad a rectángulos generados aleatoriamente.

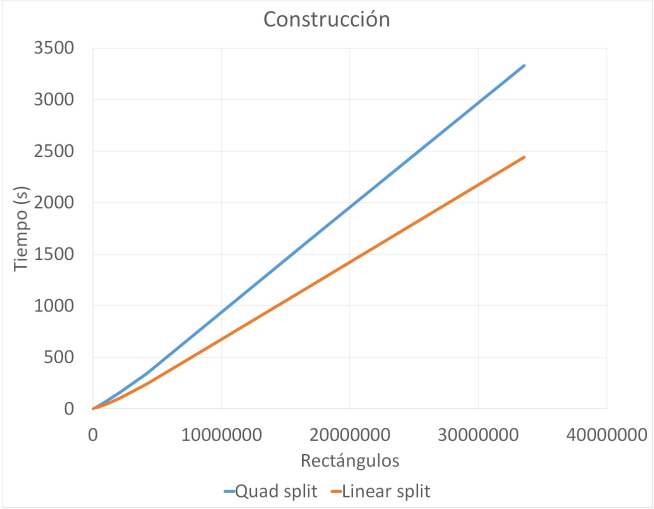

Fig. 3. Comparación de tiempos de construcción.

La Figura 3 muestra los tiempos de construcción para ambas heurísticas, donde se aprecia un incremento cercano a $\approx 35\,\%$ en el tiempo de construcción de *Quad split* respecto a *Linear split*. El espacio en disco utilizado por cada heurística, en contraste, no varía tanto (Figura 4), lo que da indicios de que las diferencias estructurales de ambas, en términos de nodos adicionales, no son tan grandes.

[1] Con la función *create_rectangles*.

[2] Paso de función *insert_data_to_tree*

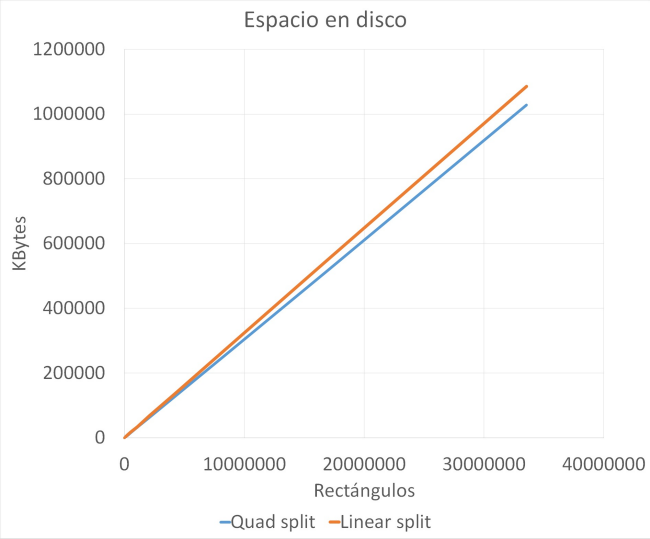

Fig. 4. Comparación de tamaño en disco.

Pese a que el tiempo de construcción utilizando *Quad split* es mayor, que el tiempo de *Linear split*, el costo de la heurística no logra dominar el tiempo del algoritmo completo, ya que se utiliza para hacer los *splits* pero el resto se mantiene igual, con sus costos no despreciables en tiempo (producto de la interacción con el disco).

La diferencia en los tamaños de los árboles en disco se va acentuando a medida que incrementa el número de entradas almacenadas. Cuando se trata de pocas entradas (e.g: 2^9) la aridad del árbol hace que unos pocos nodos puedan almacenar la totalidad de los datos, lo que no deja mucho espacio para grandes diferencias en la cantidad de nodos totales.

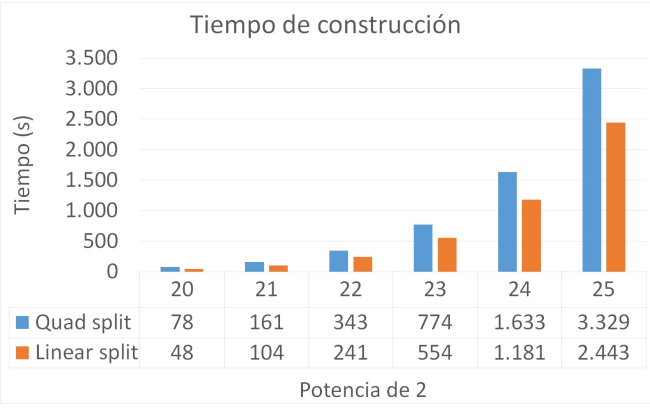

Fig. 5. Tiempo de construcción para 2^{20} a 2^{25}.

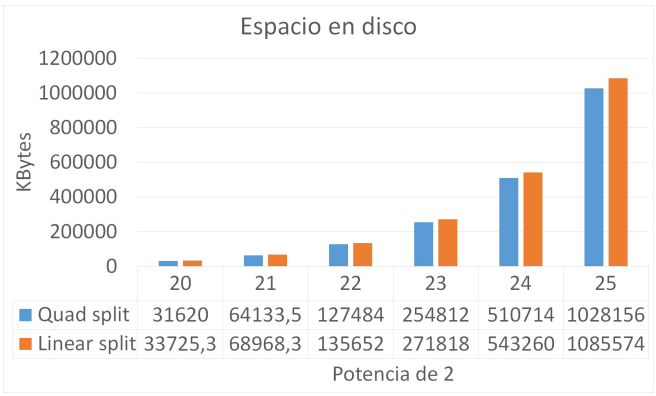

Fig. 6. Comparación de tamaño en disco para 2^{20} a 2^{25}.

Sin embargo, cuando estudiamos el llenado de los nodos de los árboles notamos que desde 2^{11} se marca una diferencia substancial. Si solo se consideran las hojas estas no se alejan demasiado del 60 %, pero en el caso del árbol completo se puede llegar a $\approx$ 50 % con *Linear split*, mientras que *Quad split* tiene a oscilar cercano al $\approx$ 66 %, exceptuando el caso 2^9.

Incluso los peores casos se alejan del llenado mínimo del 40 %.

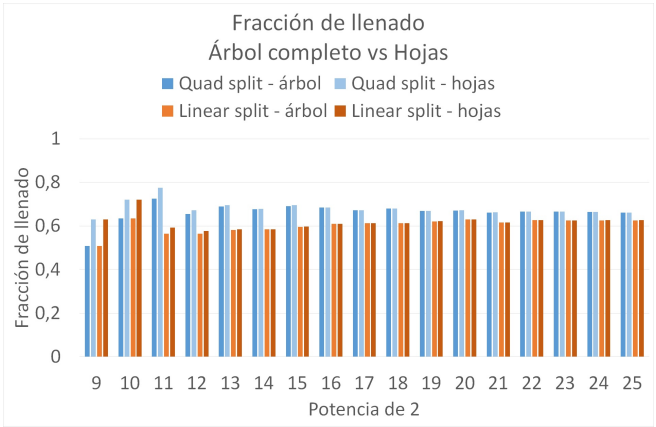

Fig. 7. Llenado de todos los nodos del árbol y de nodos hoja.

Un mayor porcentaje de llenado para *Quad split* concuerda con los datos de tamaño en disco que producen las heurísticas, esto es evidencia de que, por tener menos nodos, *Quad split* produce un árbol más compacto.

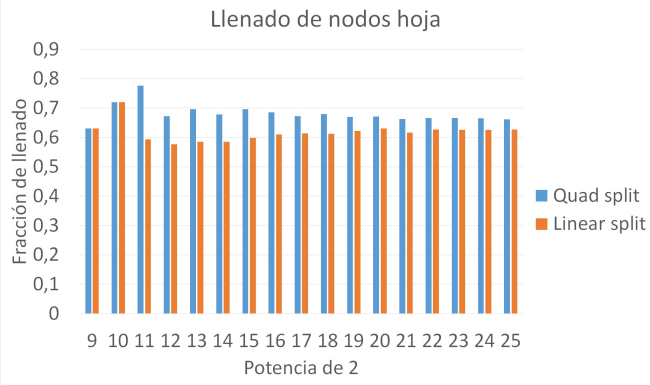

Fig. 8. Llenado de nodos hoja.

Vale la pena preguntarse si el tamaño más compacto del árbol generado por *Quad split* podría producir un mayor *overlap* de los *MBRs* contenidos en nodos distintos (a un mismo nivel - nodos hermanos) lo que podría generar un mayor costo en la búsqueda, producto de los caminos extra que habría que recorrer para responder una *query*.

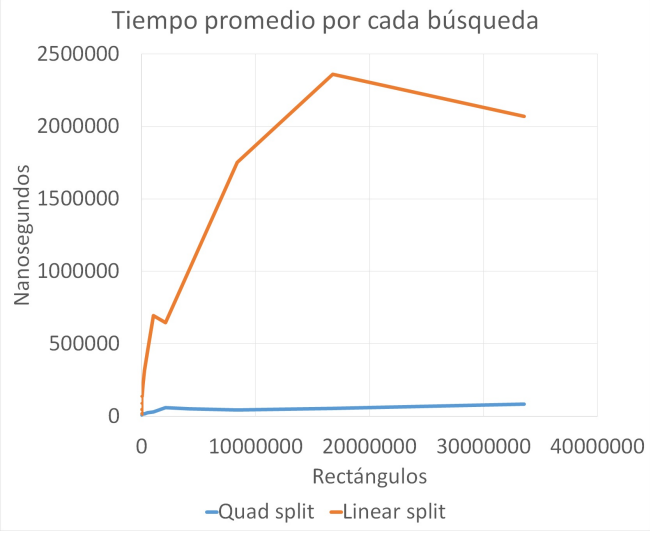

Fig. 9. Tiempo promedio por cada búsqueda.

Hay que considerar que la estrategia de *Quad split* es reducir el área desperdiciada de cada *MBR*, lo que disminuye directamente la probabilidad de generar *overlaps*, esto hace pensar que la compactibilidad del árbol no solo no es a expensas del tiempo de búsqueda, sino que es directamente una consecuencia de una mejora en la separación de los *MBRs*.

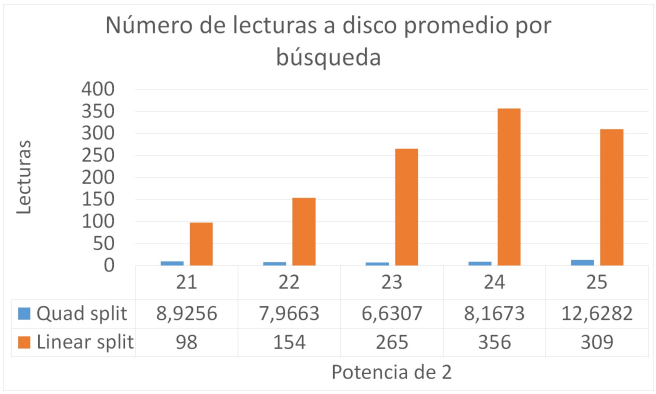

Fig. 10. Número promedio de lecturas a disco para una búsqueda.

La información contenida en las Figuras 9 y 10 da cuenta de la diferencia substancial en la cantidad de lecturas a disco por cada búsqueda, lo que se traduce en una gran diferencia de tiempo. *Linear split* no solo produce un árbol menos compacto sino que el *overlap* de los *MBRs* es mayor.

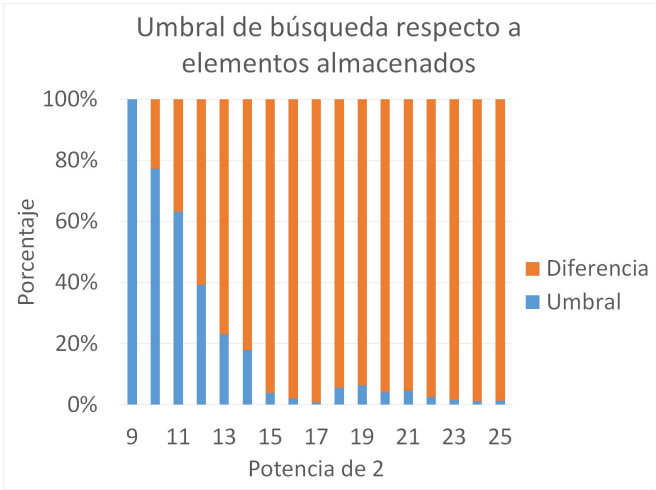

Fig. 11. Umbral de número de búsquedas desde las que es conveniente utilizar *Quad split* respecto a la cantidad de entradas del árbol, considerando el tiempo.

Logramos establecer que la diferencia en tamaño de los árboles no es considerable (para los valores estudiados, pero vale notar que esta diferencia solo parece crecer a medida que crece el número de entradas), pero las diferencias en tiempos de búsqueda y en tiempos de construcción sí lo son. El tiempo de ventaja que obtiene *Linear split* en la construcción se va perdiendo paulatinamente con cada búsqueda, por lo que desde cierto umbral de número de búsquedas es más provechoso utilizar *Quad split*. Cuando las entradas del árbol son pocas este umbral corresponde a un porcentaje importante de los datos almacenados, pero se hace menos significativo a medida que los árboles crecen, como se muestra en la Figura 11.

Por ejemplo, para 2^{10} entradas sería necesario realizar más de 790 (≈ 77 % de las entradas almacenadas) consultas para que sea más conveniente, en términos de tiempo, utilizar *Quad split*.

6. CONCLUSIONES

Si bien, la implementación utilizando un archivo por cada nodo logra ser funcional no es recomendable, en contraste a una implementación que use un solo archivo con un offset, ya que la alta cantidad de nodos que se deben crear para almacenar las entries merman la capacidad del sistema operativo de administrar los archivos.

El menor costo asociado a la construcción del árbol que se obtiene con el algoritmo Linear Split puede resultar atractivo, pero se paga muy caro con todas las queries de búsqueda subsiguientes, por lo que sería preferible, en el largo plazo, utilizar Quadratic Split para producir MBRs que sean más compactos y también árboles más compactos, si es que el espacio es una limitante.

La cantidad de lecturas que realiza *Linear split* parece brutal cuando se considera la altura del árbol, incluso en el peor caso (aridad m = 81), signo de estar explorando órdenes de magnitud más caminos candidatos que en un árbol creado con *Quad split*.

Si bien, existe un costo adicional considerable respecto a almacenar los datos directamente (de 20 *bytes* por Entry en la implementación actual), el tiempo utilizado para contestar las queries es mucho menor gracias a la baja cantidad de accesos a disco necesarios para resolverlas, por lo que constituye espacio bien gastado.

Los experimentos apoyaron las hipótesis originales, y permitieron obtener conclusiones adicionales, por lo que se consideran exitosos.

7. REFERENCES

[1] A. Guttman, R-Trees: a dynamic index structure for spatial searching", Proceedings of the 1984 ACM SIGMOD International conference on management of data, vol. 14, no. 2, pp. 47-57, June 1984.