Proyecto 1
Base de Datos II

Andrea Diaz

Octubre 2020

Contents

1 Introducción

1.1 Objetivos

El proyecto consiste en implementar distintos tipos de indexación de datos para luego comparar su performance en diferentes tipos de operaciones.

Los dos tipos de estructuras a evaluar serán el B+ Tree y Extendible Hashing.

1.1.1 B+ Tree

La gran diferencia entre archivos *indexados* y *secuenciales* se encuentra en la visualización de este. En el primero, vemos al archivo como una set de *keys* con las cuales podemos acceder a este mientas que en el secuencial, uno solo puede acceder al archivo *secuencialmente*, ir relazando operaciones en el actual o siguiente archivo que se encuentre en memoria. Los dos métodos tienen sus beneficios y desventajas, es por esto que el B+ Tree es una gran alternativa para obtener lo mejor de estos; indexando los archivos como un B Tree y accediendo a estos de manera secuencial a conveniencia.

1.1.2 Extendible Hashing

Con la introducción del B+ Tree el tiempo de acceso a archivos mejoro a $O(lg_k n)$ y si bien ya es un buen performance, este puede mejorar a $O(1)$ gracias al Hashing.

El core del tiempo constante por operación del Hash se encuentra en la creación de las address cada vez que una nueva key es introducida, con esta se facilita el storing y recovery. Si bien el addressing nos facilita muchas cosas uno de los grandes problemas de este método de indexación es el collision, dos keys pueden generar iguales hash addresses.

1.2 Datos a usar

Se usa un dataset publico de la Kaggle con el que se extrajeron mil, diez mil y veinte mil datos a indexar. Por motivos físicos del computador no se llegaron indexar más datos pues la máquina crasheaba si superaba los treinta mil datos o al tratar de limpiar el archivo, esta se congelaba. Para futuras mejoras a la implementación se recomienda realizar las pruebas en otro dispositivo.

1.3 Resultados esperados

Gracias a la teoría sabemos los tiempos de ejecución a esperar, pero la complejidad de la implementación puede que afecte los resultados en la experimentación. Aun así se comprobara los tiempos de ejecución de cada operación básica.

2 Implementación

La implementación de cada estructura varia pues se profundizo con mayor grado el Extendible Hash para lograr acércanos a los tiempos teóricos. Cabe recalcar que cada función y clase esta explicada dentro de la wiki del repositorio.

2.1 B+ Tree

Para manejar el B+ Tree se utilizaron tres clases, la principal de creación, una estructura nodo y un page manager que realiza las operaciones de archivos. Dentro de la estructura nodo se almacena el page_id, el id de próximo nodo (next_id), un counter de numero de keys, un arreglo para almacenar la data y otro que guarda las direcciones de sus hijos correspondientes.

2.1.1 Inserción

El proceso de inserción es bastantes simple, lee el header del root e inserta el key en el nodo correspondiente dentro del arreglo children. Por cada inserción se verifica si el nodo necesita un split o necesita ser guardado pues es hoja. Como en el B+ Tree las keys insertadas necesitan también indexarse dentro del ultimo nivel de nodos hojas, la función insert se encarga de colocarla dentro la posición necesaria sin duplicar su valor en el registro.

2.1.2 Eliminación

La eliminación de keys sigue la misma lógica de la inserción con la diferencia que ahora en vez de utilizar el split del nodo, ahora utilizaremos el merge. Dentro de la implementación este metodo fue el más retador pues la falta de punteros al padre complicaba las operaciones y la implementación de iteradores fue necesaria para poder completarlo.

2.1.3 Búsqueda

Se lee la header del root y se carga a un nodo temporal para luego poder ser buscado recursivamente dentro del árbol. Esta operación tiene un alto costo pues en cada iteración se tiene que leer el archivo para realizar las comparaciones correspondientes.

2.2 Extendible Hash

La implementación del Extendible Hash fue mucho más complicada y elaborada a comparación de la otra estructura. Se baso altamente en la teoría vista en clase y los libros de referencia del curso. Para lograr este resultado se crearon múltiples estructuras auxiliares

La función de hash funciona de la siguiente manera:

1. calcular el length de la key, pues este puede variar.

2. si el length es impar se aumenta en uno.

3. calculo el hash de la key con la siguiente formula que se repite length veces iterando cada dos elemento

$$hash_key = (hash_key + 100 * key[i] + key[i+1]\%199937$$

Otra función auxiliar que es necesaria para el correcto funcionamiento de la indexación es la encargada de generar las direcciones, su lógica es la siguiente:

1. Hasheo la key.

2. Itero sobre los niveles, haciendo un shift_left a la dirección y un shift_rigth al hash key.

El código a detalle se encuentra en el repositorio.

2.2.1 Inserción

La lógica de la inserción es la siguiente:

1. genero la dirección del key sobre la cantidad actual de niveles.

2. uso ese valor como índice en el array de bucket_addr para obtener el bucket al que pertenece.

3. traigo ese bucket de memoria.

4. reviso si es necesario hacer un split del bucket. Si es así, se encuentra un bucket buddy con espacio disponible y se reparten los keys.

5. inserto la key y su address en el bucket.

2.2.2 Eliminación

La lógica es la siguiente:

1. genero la dirección del key sobre la cantidad actual de niveles.

2. uso ese valor como índice en el array de bucket_addr para obtener el bucket al que pertenece.

3. traigo ese bucket de memoria.

4. elimino la key y su address en el bucket.

5. reviso si es necesario combinar los buckets. Si es así se combinan un bucket buddy con el current.

2.2.3 Búsqueda

1. genero la dirección del key sobre la cantidad actual de niveles.
2. uso ese valor como índice en el array de bucket_addr para obtener el bucket al que pertenece.
3. traigo ese bucket de memoria.
4. obtengo el address del key de ese bucket.

Al implementar esta estructura pude percatarme que mucho del funcionamiento correcto del Hash depende del pack, unpack y el load de los buckets así como sus buffers.

Se recalca que la explicación detallada de cada función se encuentra dentro de la wiki del repositorio.

3 Resultados

Por motivos técnicos, mi computadora solo podía procesar hasta 20mil datos a indexar. El bajo procesamiento del computador puede ser una variable que afectó la ejecución de los programas.

3.1 Pruebas

Todas las pruebas de las funciones se realizaron a través de unit test gracias a la librería de GTest.

3.2 Inserción

3.2.1 B+ Tree

B+ Tree			
Tiempo en ms			
iteración	1k datos	10k datos	20k datos
1	104	2437	4106
2	550	2204	2647
3	571	816	4085
4	106	2126	3202
5	510	814	2630
Promedio	368.2	1679.4	3334

3.2.2 Extendible Hashing

Ahora para el Extendible Hahshing

Extendible Hash			
Tiempo en ms			
iteración	1k datos	10k datos	20k datos
1	15	113	287
2	17	102	267
3	20	105	268
4	15	104	409
5	18	102	268
Promedio	17	103.2	299.8

3.3 Búsqueda

Se extrajo valores aleatorios de los text files y se midió el tiempo de búsqueda en milisegundos.

3.3.1 B+ Tree

B+ Tree			
Tiempo en ms			
iteración	1k datos	10k datos	20k datos
1	aprox 1	aprox 1	aprox 0
2	aprox 0	aprox 0	aprox 1
3	aprox 1	aprox 0	aprox 0
4	aprox 0	aprox 0	aprox 0
5	aprox 1	aprox 0	aprox 0
Promedio	approx 1	aprox 0	aprox 0

B+ Tree			
cantidad de accesos a disco			
iteración	1k datos	10k datos	20k datos
1	11	13	14
2	11	13	14
3	11	13	14
4	11	13	14
5	11	13	14
Promedio	11	13	14

3.3.2 Extendible Hash

Extendible Hash			
Tiempo en ms			
iteración	1k datos	10k datos	20k datos
1	aprox 0	aprox 0	aprox 0
2	aprox 0	aprox 0	aprox 0
3	aprox 0	aprox 0	aprox 0
4	aprox 0	aprox 1	aprox 0
5	aprox 0	aprox 0	aprox 0
Promedio	aprox 0	aprox 0	aprox 0

Extendible Hash			
cantidad de accesos a disco			
iteración	1k datos	10k datos	20k datos
1	2	2	2
2	2	2	2
3	2	2	2
4	2	2	2
5	2	2	2
Promedio	2	2	2

3.4 Eliminación

3.4.1 B+ Tree

B+ Tree			
Tiempo en ms			
iteración	1k datos	10k datos	20k datos
1	aprox 1	aprox 1	aprox 0
2	aprox 0	aprox 0	aprox 1
3	aprox 1	aprox 0	aprox 0
4	aprox 0	aprox 0	aprox 0
5	aprox 1	aprox 0	aprox 0
Promedio	aprox 1	aprox 0	aprox 0

B+ Tree			
cantidad de accesos a disco			
iteración	1k datos	10k datos	20k datos
1	13	14	16
2	12	14	15
3	12	16	14
4	12	14	14
5	12	14	15
Promedio	12.2	14.4	14.8

3.4.2 Extendible Hash

Extendible Hash			
Tiempo en ms			
iteración	1k datos	10k datos	20k datos
1	aprox 1	aprox 1	aprox 0
2	aprox 0	aprox 0	aprox 1
3	aprox 0	aprox 0	aprox 0
4	aprox 0	aprox 0	aprox 1
5	aprox 0	aprox 0	aprox 0
Promedio	aprox 0	aprox 0	aprox 1

Extendible Hash			
cantidad de accesos a disco			
iteración	1k datos	10k datos	20k datos
1	3	3	4
2	3	4	4
3	3	3	5
4	3	3	4
5	3	3	4
Promedio	3	3.2	4.2

3.5 Comparación y Análisis

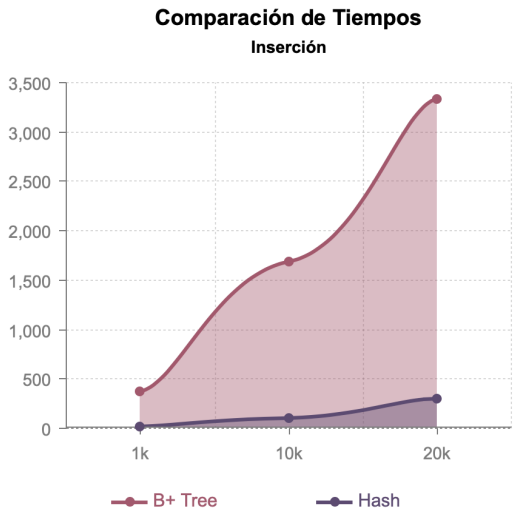

Figure 1: Comparación de tiempos en la inserción de datos

Una de las operaciones con mayores diferencias que pudimos notar fue la de inserción. El B+ Tree necesita mayores pasos para indexar los nuevos datos.

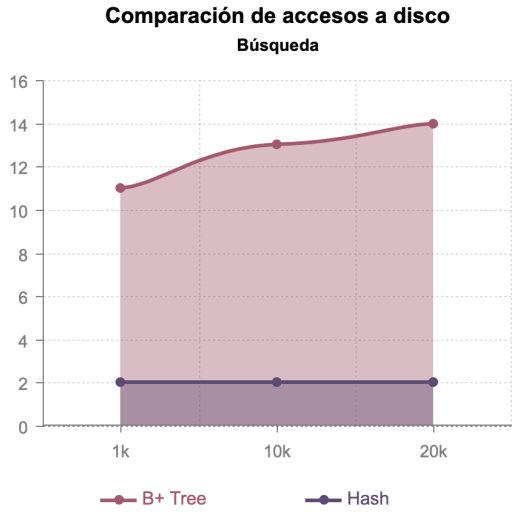

Figure 2: Comparación de accesos a disco en la búsqueda de datos

Como se tenia previsto, la cantidad de accesos a disco en la búsqueda en

el B+ Tree aumenta ligeramente con el tamaño de datos indexados mientras que el Extensible Hash mantiene sus accesos constantes. Este resultado era esperado debido a la gran ventaja que posee el Hash sobre la recuperación de llaves especificas, mientas que el B+ Tree es una gran herramienta en búsquedas por rango cuyos accesos son mínimos comparados con el Hash, que necesitaría operaciones auxiliares para poder realizar este tipo de búsqueda.

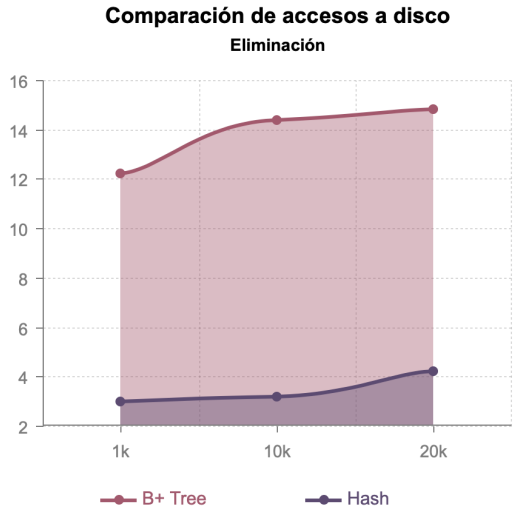

Figure 3: Comparación de accesos a disco en la eliminación de datos

El numero de accesos a disco en la operación de eliminación aumento en el Hash debido a las funciones de merge con buddy buckets para que estos contengan un numero mínimo de keys contenidas, pero aun así su tiempo se mantuvo al mínimo.

Si se comparan las formas de indexación con la otra se nota una clara diferencia, una cosa a tomar en cuenta es la evaluación individual de cada una. Mientras el B+ Tree parece tener un largo gap con los accesos de su contrincante, si solo se fija en su crecimiento por tamaño de dato, podemos darnos cuenta que este no crece demasiado por cantidad de datos.

4 Conclusiones

El proyecto en general fue de por si retador pues toda la implementación se realizó por solo un estudiante pero al analizar los resultados podemos concluir que las métricas coinciden con los tiempos teóricos de cada estructura. Posibles optimizaciones se pueden realizar a nivel de flujo de ejecución y estructuras generales de los auxiliares del Hash como el B+ tree.

Se pudo comprobar que el costo de recuperación de la data es, efectivamente,

mejor en el Extendible Hash que en el B+ Tree, pero el problema de colisiones en la inserción causa que el Hash rinda casi igual que el B+ Tree en algunos casos.

Cabe recalcar que la elección del tipo de indexación depende de los resultados que se quieren obtener para el tipo de operaciones necesarias en su uso practico, mientras que el Hash tiene un tiempo constante de recuperación de llaves especificas, este no es la primera opción si se necesitaran búsquedas de rango repetidamente.