Universidad de Ingeniería y Tecnología

Departamento de Ciencias de la Computación

Proyecto N°1

Base de Datos 2

Integrantes:
Mitsuo Sebastián Murakami Miyahira
Milloshy Noelia Crisostomo Rodriguez
Christian Maxim Frisancho
Diego Randolp Quispe

Profesor:
Heider Sanchez

September 28, 2024

1 Introduction

El objetivo de este proyecto es desarrollar un sistema de gestión de archivos que implemente las técnicas de organización secuencial, AVL y hashing extensible para optimizar las operaciones de inserción, búsqueda y eliminación sobre archivos de datos reales. De esta forma, mejorar la eficiencia de la recuperación de información mediante el uso de estas estructuras avanzadas de indexación y almacenamiento.

El proyecto utilizará datos extraídos de dos archivos planos obtenidos de Kaggle. El primer archivo contendrá información de una lista de canciones; cada registro contiene el nombre de la canción, el nombre del artista, fecha de lanzamiento de la canción (año y mes), cantidad de listas de reproducción en las que aparece la canción, número total de reproducciones en plataformas de streaming, entre otros. Estos archivos se utilizarán para probar y comparar las tres técnicas de organización seleccionadas: Sequential File, AVL File y extensible hashing.

Se espera que la implementación de las tres técnicas de organización de archivos ofrezca resultados diversos en términos de eficiencia y acceso al almacenamiento secundario. En el caso del Sequential File, la reconstrucción del archivo con el uso de un buffer permitirá mantener el orden físico de los datos, mejorando la inserción de nuevos registros. El AVL File optimizará la búsqueda y eliminación mediante su estructura balanceada, garantizando tiempos de búsqueda logarítmicos. Finalmente, el extensible hashing permitirá búsquedas rápidas por clave, aunque no soportará búsquedas por rango, destacándose por su eficiencia en grandes volúmenes de datos. En conjunto, estas técnicas ofrecerán un manejo optimizado de datos estructurados.

2 Técnicas utilizadas

1. **Sequential File**
   El método de Sequential File es una manera ordenada de guardar la data. Su archivo principal se mantiene siempre ordenado para aplicar búsqueda binaria obteniendo una complejidad de O(log(n)) en la búsqueda de un registro. En caso del archivo auxiliar, tiene un tamaño del logaritmo del tamaño del archivo principal, log(n), con complejidad O(k) la búsqueda ya que este no se encuentra ordenado.

   Algoritmos de inserción, eliminación, búsqueda, búsqueda por rango y reconstrucción

   1.1 Inserción: La inserción en el Sequential File se realiza en el archivo auxiliar. Esto permite mantener el archivo principal de manera ordenada hasta que el archivo auxiliar se llene. El tamaño del archivo auxiliar es log(n), donde n es el tamaño del archivo principal. Al llenarse se llama a la función reconstrucción.

1.2 Eliminación: La eliminación es por el método ***move the last*** donde se mueven todos los registros desde el final hacia el registro a eliminar. Tiene complejidad O(n).

1.3 Búsqueda: La búsqueda es por búsqueda binaria con la llave ***track_name***. En caso no se encuentre en el archivo principal, se realiza una búsqueda lineal en el archivo auxiliar. Se retorna el registro. En caso no sea encontrado, se da un error de búsqueda.

1.4 Búsqueda por rango: En la búsqueda por rango se realiza primero una búsqueda binaria para encontrar el inicio del rango e ir agregando los registros mientras estén dentro del rango, a un vector de registros. Se busca linealmente si hay un registro que esté dentro del rango de búsqueda. Finalmente, se retorna el vector de registros.

1.5 Reconstrucción: Al llamarse a esta función, se va agregando cada registro del archivo auxiliar en el archivo principal. Primero se realiza una búsqueda binaria para encontrar su ubicación, para posteriormente mover los siguientes registros 1 espacio para luego insertar el registro del archivo auxiliar. Así hasta insertar todos los registros del archivo auxiliar.

2. **Árbol AVL (AVL File):**
Este árbol balanceado de búsqueda binaria asegura que el acceso a los registros sea eficiente, con una complejidad de búsqueda, inserción y eliminación de O(log n). Cada nodo del árbol contiene un registro de datos (en este caso, información sobre canciones) y se indexa utilizando un campo único, como el nombre de la canción.

Algoritmos de Inserción, Eliminación y Búsqueda en un Árbol AVL

2.1 Inserción: El proceso de inserción en el árbol AVL sigue las reglas de un árbol binario de búsqueda. Después de insertar el nodo, se verifica el balance del árbol. Si el árbol está desequilibrado, se realizan rotaciones (simples o dobles) para restaurar el equilibrio.

2.2 Eliminacion: La eliminación en un árbol AVL también sigue las reglas de un árbol binario de búsqueda. Una vez eliminado el nodo, se recalcula el balance de todos los nodos antecesores, y se realizan las rotaciones necesarias para mantener el equilibrio.

2.3 Búsqueda: La búsqueda en un árbol AVL es similar a la búsqueda en un árbol binario de búsqueda. Se compara la clave de búsqueda en este caso es el nombre de la música ***track_name*** con la clave en el nodo actual y se sigue el subárbol izquierdo o derecho en función de si la clave es menor o mayor.

2.4 Rotaciones:

2.4.1 Rotación simple:

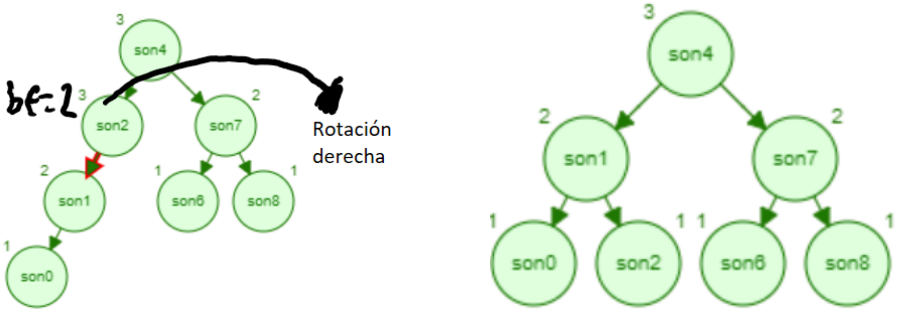
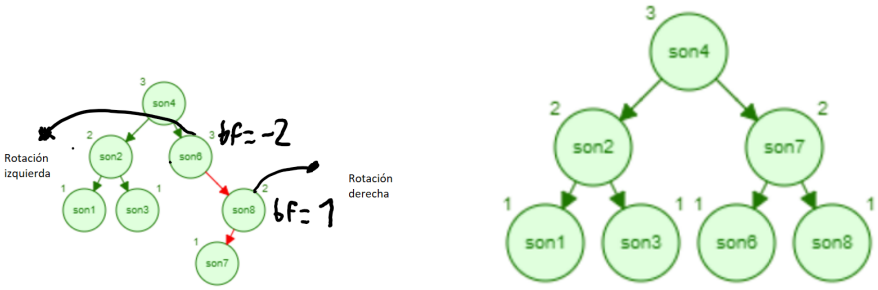


Figure 1: antes y despúes de la rotación

2.4.2 Rotación doble:



(a) figura1 (b) figura2

Figure 2: antes y despúes de la rotación doble

3. **Extendible Hashing**
El extendible hashing es un hash dinámico que se usa para manejar bases de datos que crecen y reducen su tamaño con el tiempo, indexando los registros mediante sus secuencias de bits y poniéndolos en diferentes buckets en base a los 'n' bits desde su least significant bit de su key, en este caso el nombre de la cancion, en binario. En cuanto se insertan varios elementos y los buckets están llenos, se usa la division de buckets ahorrandonos el tiempo de reestructurar todo el hash. Cada bucket contiene varios registros. De esta manera, la búsqueda tiene una complejidad de O(1), ya que se puede acceder directamente al bucket que contiene al registro debido a que el índice tiene el tamaño necesario para manipularlo en RAM. Aunque en el peor caso de overflow es O(k), donde k son los registros de overflow que hay y el registro esta en el ultimo bucket, por lo que se tiene que leer cada uno de estos buckets. La division de buckets es O(k+d) siendo k el numero de registros del bucket lleno y d la profundidad. Finalmente la

eliminacion es O(1) en el mejor caso usando MOVE THE LAST en cada Bucket. Sin embargo se puede llegar hasta una complejidad O(n) cuando hay muchos registros y muchos Buckets.

Inserción: La insercion en el extendible hashing se hace hacia los buckets, en donde tenemos el indice del registro en un codigo binario y en base a sus ultimos numeros insertamos el registro en el bucket corresponiendete, generalmente es O(1) sin embargo hay que considerar el caso cuando el bucket se llena, en este caso la insercion es O(k+d) porque se divide el bucket en 2 buckets nuevos, haciendo que el hash index crezca por ejemplo de 0 a 10 y 00, finalmente tenemos la insercion cuando hay overflow y no se puede dividir es O(k) donde k es el numero de elementos en los overflows.

Eliminacion: La eliminación en el extendible hashing tiene varios casos, dentro de cada bucket se usa Move the last con una complejidad de O(1) en el mejor caso cuando no hay overflow y el registro esta en un bucket directamente referenciado por el Index. El segundo caso es cuando tenemos overflow y nos cuesta buscar el registro O(k) y la eliminacion O(1) siendo en total O(k). El tercer caso ocurre cuando un bucket tiene 0 registros despues de la eliminacion, no tiene next Buckets y su profundidad local es mayor a 1. En esta situacion, se combina este bucket y el bucket que contiene los elementos que su key en binario solo se diferencia en el Most significant bit. Asimismo, al hacer esta combinacion se reorganiza el indice y todos los indices que referenciaban al bucket con 0 registros, ahora referencian a su bucket relacionado. El ultimo caso ocurre cuando el bucket queda vacio y no se puede combinar ya que su profundiad local es 1, por lo que se deja el bucket vacio.

Búsqueda: La busqueda en el extendible hasing sin overflow es O(1) gracias a las propiedades del hash, sin embargo esta puede llegar a ser O(k) cuando hay overflow.

Análisis comparativo

Operación	Sequential File	AVL File	Extendible Hashing
Inserción	O(1) (reconstrucción del archivo O(log(n)*n)	O(log n) (rotaciones para mantener balance)	O(1) promedio, O(n) peor caso
Eliminación	O(n)	O(log n) (rotaciones para mantener balance)	O(1) promedio, O(n) peor caso
Búsqueda	O(log(n)) (binary search)	O(log n)	O(1) promedio, O(n) peor caso

Explicacion del parser SQL:
Se hizo uso de la libreria regex para identificar y procesar distintos tipos de comandos SQL que están definidos en formato de texto. Los comandos SQL reconocidos son:
Para el caso del AVL
INSERT INTO: Para insertar un nuevo registro en el arbol AVL
SELECT *: Para seleccionar y mostrar todos los registros.
SELECT * WHERE track_name = 'X': Para buscar un registro específico por el nombre de la canción.
DELETE WHERE track_name = 'X': Para eliminar un registro específico por el nombre de la canción.

Se definen cuatro expresiones regulares (con std::regex) que corresponden a cada tipo de comando SQL:
insert_regex: Reconoce el comando INSERT INTO con los valores a insertar.
select_regex: Reconoce el comando SELECT * para mostrar todos los registros.
select_specific_regex: Reconoce el comando SELECT * WHERE index = 'X' para buscar un registro específico.
delete_regex: Reconoce el comando DELETE WHERE index = 'X' para eliminar un registro específico. Con respecto al procesamiento de comando SQL una vez que se identifica el tipo de comando se realiza un proceso diferente
Comando INSERT INTO: Se extraen los valores que serán insertados en el registro correspondientes a los campos de la canción (nombre, artista, etc.) y luego se inserta en la estructura
Comando SELECT *: Al reconocer el comando SELECT * se muestra la lista completa de canciones en la base de datos.
Comando SELECT * WHERE index = 'X Se extrae el nombre de la canción de la consulta SQL.Luego, el nombre de la canción se usa para buscar en la estructura de datos segun sea la funcion que se implementó por ejemplo en el caso de AVL es ***tree.search(track_name.c_str())*** si se encuentra el nombre de la cancion se imprime el registro completo. Si no se encuentra, se informa que el registro no existe.
Comando DELETE WHERE index = 'X': El nombre de la canción se pasa al método ***tree.remove(track_name.c_str())***, el cual se encarga de eliminar el registro correspondiente de la estructura. Una vez eliminado el registro, se imprime un mensaje indicando que el registro fue eliminado.

Si la consulta SQL no coincide con ninguna de las expresiones regulares definidas, el sistema imprime un mensaje de error: "Comando SQL no reconocido.". Esto asegura que solo se procesen comandos válidos.

3 Resultados experimentales

Al comparar las tres estructuras de datos, Sequential File, AVL File, y Extendible Hashing, se observan diferencias notables en cuanto a tiempos de operación y accesos a disco, lo que tiene implicaciones directas en el rendimiento dependiendo de los casos de uso.

1. **Sequential File:**
   Se midieron los tiempos de las funciones del Sequential file con la libreria chrono para la inserción, búsqueda, búsqueda por rango y eliminación:

Operación	Time (ms)
Inserción	0.1892
Busqueda	6.035
Búsqueda por rango	6.6747
Eliminación	6.8762

   Con respecto al total de accesos a disco duro estos fueron los resultados

Operacion	Acceso de lectura	Acceso de escritura
Insercion	0	1
Busqueda	10	-
Búsqueda por rango	67	-
Eliminación	16	0

2. **AVL File:** Para el caso del AVL se hizo uso de la libreria chrono donde se implementaron 2 funciones measureInsertionTime() y measureSearchTime() estos fueron los resultados

Operacion	Time (ms)
Insercion	0.0059
Busqueda	0.0013

   Con respecto al total de accesos a disco duro estos fueron los resultados

Operacion	Acceso de lectura	Acceso de escritura
Insercion	19	18
Busqueda	8	-

3. **Extendible Hashing**
   Para medir los timepo de insercion, eliminaicon y busqueda del Extendible Hashing se implemento la funcion measureTimes, la cual dio los siguiente resultados:

Operacion	Time (ms)
Insercion	0.0511
Busqueda	0.1062
Eliminacion	0.0597

En cuanto a los accesos a disco duro, estos fueron los resultados:

Operacion	Acceso de lectura	Acceso de escritura
Insercion	2	1
Busqueda	3	-
Eliminacion	2	2

Discusión de los resultados

- Rendimiento general: El AVL File es claramente la estructura más rápida en términos de tiempos de inserción y búsqueda, lo cual es ideal para aplicaciones que priorizan la velocidad de consulta. Sin embargo, su uso elevado de accesos a disco durante la inserción puede ser un factor limitante.

- Accesos a disco: El Sequential File tiene la ventaja de un menor número de escrituras, pero a costa de tiempos de búsqueda elevados. El Extendible Hashing ofrece un buen equilibrio entre tiempos de operación y accesos a disco, lo que lo hace una opción sólida para sistemas donde el acceso al disco es costoso.
- En general, se podria decir que en el caso de Sequential File es más adecuado para aplicaciones donde la inserción es frecuente y las búsquedas son menos comunes, búsqueda por rango o no requieren rapidez, como en sistemas de almacenamiento a largo plazo. Por otro lado, el AVL File es ideal para aplicaciones donde las operaciones de búsqueda y actualización son frecuentes y, el rendimiento debe ser alto, como bases de datos que requieren rapidez en las consultas. Finalmente, el Extendible Hashing es una buena eleccion para sistemas en los que el acceso a disco es un factor limitante.

4 Pruebas de uso

Por consola: Para mostrar los registros de la tablas se usa el comando
SELECT * FROM songs;

```
> SELECT * FROM songs;
Registros:
Track: "Apna Bana Le (From ""Bhediya"")", Artist: Arijit Singh, Sachin-Jigar, Artist Count: 2, Released Year: 2022, Released Month: 11, Playlists: 86, Streams:
 139836056, Key: A, Mode: Major, Danceability: 59, Cover URL: Not Found
Track: "Besharam Rang (From ""Pathaan"")", Artist: Vishal-Shekhar, Shilpa Rao, Caralisa Monteiro, Ku, Artist Count: 6, Released Year: 2022, Released Month: 12,
 Playlists: 130, Streams: 140187018, Key: G#, Mode: Minor, Danceability: 77, Cover URL: Not Found
Track: "Come Back Home - From ""Purple Hearts""", Artist: Sofia Carson, Artist Count: 1, Released Year: 2022, Released Month: 7, Playlists: 367, Streams: 97610
446, Key: G, Mode: Major, Danceability: 56, Cover URL: https://i.scdn.co/image/ab67616d0000b27326af6664e24916047b68ab81
Track: "I'm Tired - From ""Euphoria"" An Original HBO Se, Artist: Labrinth, Artist Count: 1, Released Year: 2022, Released Month: 2, Playlists: 1888, Streams:
121913181, Key: , Mode: Minor, Danceability: 28, Cover URL: Not Found
Track: "Kesariya (From ""Brahmastra"")", Artist: Pritam, Arijit Singh, Amitabh Bhattacharya, Artist Count: 3, Released Year: 2022, Released Month: 7, Playlists
: 292, Streams: 366599607, Key: , Mode: Major, Danceability: 58, Cover URL: Not Found
```

Para eliminar una cancion se usa el siguiente comando
DELETE FROM songs WHERE track_name = 'positions';

```
> DELETE FROM songs WHERE track_name = 'positions';
Registro eliminado.
```

Para insertar un registro
INSERT INTO songs (track_name, artist_name, artist_count, released_year, released_month, playlists, streams, key, mode, danceability, cover_url) VALUES ('Positions', 'Ariana Grande', 1, 2020, 10, 50, 1000000, 'C', 'major', 80, 'https://example.com/cover.jpg');
Para hacer la búsqueda de una canción

```
> INSERT INTO songs (track_name, artist_name, artist_count, released_year, released_month, playlists, streams, key, mode, danceability, cover_url) VALUES ('Pos
itions', 'Ariana Grande', 1, 2020, 10, 50, 1000000, 'C', 'major', 80, 'https://example.com/cover.jpg');
Registro insertado.
```

SELECT * FROM songs WHERE track_name = 'Positions';

```
> SELECT * FROM songs WHERE track_name = 'Positions';
Track: Positions, Artist:  Ariana Grande, Artist Count: 1, Released Year: 2020, Released Month: 10, Playlists: 50, Streams: 1000000,
 Key:  C, Mode:  major, Danceability: 80, Cover URL:  hhttps://example.com/cover.jp
```