CS2702
BASE DE DATOS II

Proyecto 1
Organización de Archivos y Control de Concurrencia

Prof. Heider Sánchez

Giordano Alvitez
Juan Vargas
Roosevelt Ubaldo

Contents

1 Introducción

1.1 Objetivo

La finalidad del proyecto es lograr un análisis comparativo entre dos tipos de índices (B+ Tree y Static Hashing). Una punto importante del proyecto es que hemos desarrollado la implementación de ambos índices basados en disco y de manera genérica (usando templates).

1.2 Definiciones previas

1. **B+ Tree:** Es un tipo de árbol similar al árbol B-Tree con 2 diferencias principales. La primera es que las hojas del B+ Tree se encuentran enlazadas con las hojas adyacentes. La segunda diferencia es que al insertar un valor al árbol B+ Tree, el valor insertado se agrega a la hoja y en caso existe un overflow al realizar la inserción, se genera un split en la hoja que se propaga hacia arriba (bottom_up) hasta que encuentre un nodo que no haga overflow. En la propagación el valor que se propaga se mantiene aún en la hoja y en los nodos que se dividen hacia arriba. El valor que se propaga hacia arriba puede ser distinto en cada nivel. Si bien en la implementación original del B+Tree solo se posee un enlace al nodo siguiente, en la implementación realizada también se consideró un enlace al nodo anterior.

2. **Static Hashing:** Este tipo de índice se vale de una función hashing la cual permite transformar el valor de la llave primaria a un valor fácilmente manejable por la estructura, usualmente del tipo long (ya que va a ser una dirección de memoria). El valor generado servirá luego para localizar en que bucket (espacio de memoria en disco), será localiza el un nuevo registro.

1.3 Descripcion del Dominio de Datos

La data trabajada en el proyecto proviene de la plataforma Kagle, uno de los mayores repositorios de dataset online. En este caso, la data corresponde a accidentes automovilísticos de los Estados Unidos entre febrero del 2016 y diciembre del año 2019. En la sección referencias se puede encontrar el enlace donde se encuentra la data. Cada registro esta conformado por 49 campos de los cuales hemos decidido trabajar por comodidad con solo 4 además de la llave primaria que sería el campo ID. Los campos seleccionados fueron: Description, City, State y Weather_Condition.

1.4 Resultados que se esperan obtener

La base teórica de ambos índices demuestran que ambos logran un gran desempeño al momento de la búsqueda y recuperación de registros desde disco. Sin embargo, al almacenar un nuevo registro existe un delay por operaciones que se realizan dentro de las estructuras con las que se trabaja cada tipo de índice. La compensación recibida en las búsqueda bien vale aún trabajar con los índices a pesar del retraso en las inserciones.

2 Implementación

2.1 Estructura Principal

Para la implementación del proyecto usamos C++ y hemos seguido el paradigma de programación orientada a objetos, por este motivo hemos trabajado las diferentes implementaciones

de índices en clases separadas. Sin embargo, hay clases que son usadas por ambos índices como **DiskManager** que se encarga de realizar las operaciones de lectura y escritura de manera genérica.

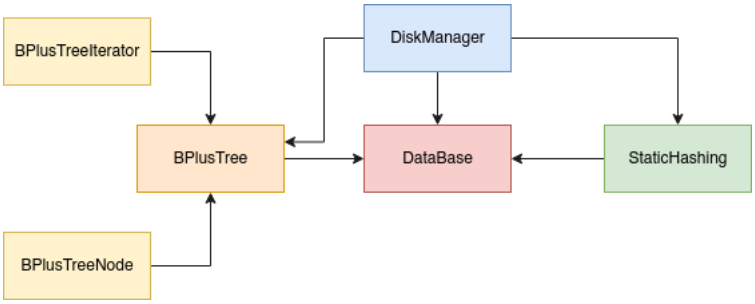


Figure 1: Diagrama de dependencias principal

Para utilizar ambos índices, existe una clase central llamada **DataBaseManer** que se encarga de recibir los registros e Insertarlos de acuerdo a un índice previamente seleccionado, o en su defecto sin ningún índice.

2.2 Disk Manager

Esta clase sobre-escribe la función **fstream** para facilitarnos el acceso y escritura a disco. Por tal motivo, se le suministra como entradas el nombre de un archivo de texto y un booleano *reset* para truncar o no el archivo.

Los métodos que se implementaron para esta clase son:

- **write_record:** este método escribe un objeto genérico llamado registro en una posición en disco previamente ingresada.
- **retrieve_record:** este método recupera un objeto registro dada una posición de memoria previamente ingresada.
- **write_record_to_ending:** este método recibe un registro y lo escribe al final del archivo.

2.3 B+Tree

En esta implementación usamos el **order** como valor para definir el número máximo de elementos que puede contener un nodo y para un mejor manejo se creo una clase para los nodos por separado. En esta clase hemos trabajado con templates para hacerla genérica al tipo de dato de la key. Cada nodo posee un booleano si es hoja o no, un array de keys, un array de hijos, un contador de keys ocupadas, un puntero para el siguiente y el nodo anterior. Si el nodo es hoja, se guarda en un array la posición en disco del registro que corresponde a dicha key. Para un mayor entendimiento de la estructura se puede observar la siguiente imagen:

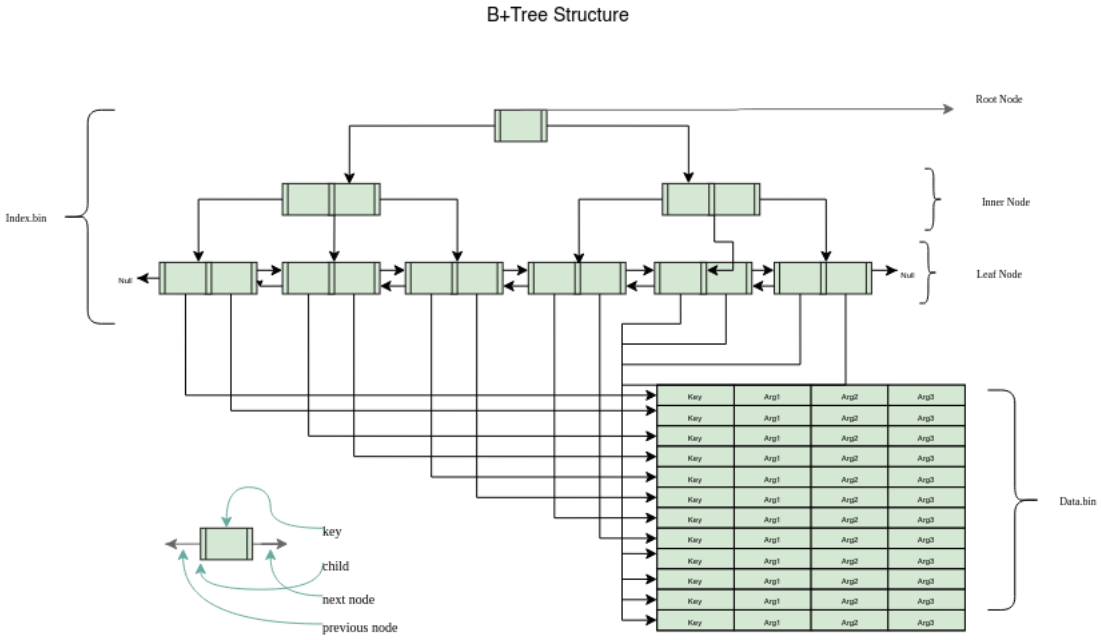


Figure 2: Estructura de la implementación del índice B+Tree

Para facilitarnos los splits de los nodos cuando están en overflow, hemos permitido que cada key pueda Insertar un elemento extra al número permitido y al cabo de dicha inserción ejecutamos una función que nos indica si el nodo está o no en overflow. Si está en overflow ejecutamos un split del nodo. Dado que el split no es el mismo para el nodo padre o los nodos hojas, hemos creado un split especial para cada uno de ellos.

Al momento de hacer el split de un nodo hoja, hemos seguido la convensión de **left based split**, es decir copiamos en el nodo izquierdo el valor de la key que estamos promoviendo al nodo padre. En el caso del split de los nodos padres, solo se promueve la key, mas no se copia en el nodo izquierdo.

Por otro lado, para facilitarlos las consultas por rango, hemos creado una clase llamada **BPlusTreeIterator** que nos permite movernos hacia adelante o hacia atrás a nivel de los nodos hojas. Para ello hemos implementado los métodos *begin()*, *end()*, *null()* y los operadores $++$ y $--$.

En síntesis, los métodos públicos para esta clase son los siguientes:

- **Insertar:** este método se encarga de recibir una key para ser Insertarada en el índice.
- **showTree:** este método nos permite mostrar en consola el árbol que se ha formado.
- **print:** este método nos permite mostrar en consola solo los nodos hojas,
- **begin():** este método retorna un iterador con la posición del primer nodo hoja.
- **end():** este método retorna un iterador con la posicoón del último nodo hoja.
- **null():** este método nos permite saber si nos hemos excedido de los límites hacia adelante o atrás cuando estámos haciendo uso de las operaciones $++$ o $--$

- **operator++:** este método mueve el iterador al siguiente key, si ya hemos llegado al último key de un nodo, nos movemos al siguiente nodo.
- **operator−−:** este método mueve el iterador al key anterior, si ya hemos llegado al key anterior al primero, nos movemos al nodo anterior.
- **operator∗:** este método retorna el valor de la key para dicha posición del iterador
- **getRecordId:** este método retorna el valor de la posición en disco al cual pertenece la key para dicha posición del iterador
- **operators ==, !=, =:** nos permite comprobar la igualdad o diferencia de dos iteradores, o asignar el valor de un iterador a otro iterador.

2.4 Static Hashing

Dentro de la implementación estamos incluyendo los valores de **global depth (gd)** y **full depth (fd)** , los cuales nos darán la posicion del bucket respectivo a un registro a un registro junto a la **función hash** y la cantidad máxima que cada bucket podrá almacenar en disco. Igual que la clase B+Tree hemos trabajado con templates para que permita flexibilidad al momento de trabajar con el tipo de llave primaria.

Cada **bucket** le corresponde un campo NextBucket en caso un bucket complemete su máxima capacidad, entonces el nuevo registro tendra que ser almacenado en el siguiente bucket libre. Igualmente se cuenta con 2 arreglos, uno para almacenar la dirección de cada registro y otro para almacenar la llave primaria de cada registro tambien. Los bucket son almacenados de manera sequencial y la dirección de cada uno corresponde al índice con el cual trabajan.

Como ya se menciono previamente, Static Hashing trabaja con una **funcion hashing**, esta función permitira generar un hash que trabajara como índice el cual nos ayudara a definir la posición del bucket en que corresponde un registro especifico. Dentro de la función se recibe el valor de la llave primaria. Se realiza un calculo con la llave primaria para poder generar el hash respectivo. En nuestro caso, la función hash para generar un hash especifico es la siguiente:

$$Hash = f(key\_value) = key\_value\%gd$$

La siguiente figura muestra como static hashing trabaja. La función hashing necesita del valor de la llave primaria, luego cálcula el valor hash que seria 4 en la figura y este valor hash sirve como índice para localizar el bucket especifico a un registro. Si el bucket se encuentra lleno procede al siguiente bucket conectado, hasta encontrar uno que no haya alcanzado su máximo capacidad de almacenamiento.

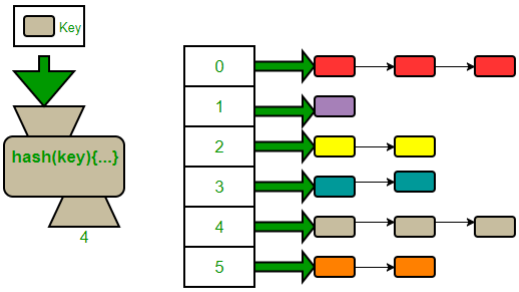


Figure 3: Estructura de la implementación del índice Static hashing

2.5 Data Base Manager

Esta clase se encarga de Insertar, buscar o buscar por rango los registros usando un índice B+Tree, Static Hashing o sin índice. En cualquiera de los casos se emplea un *record manager* para almacenar los registros mientras que los índices se crea en archivos separados de acuerdo al índice seleccionado como se muestra a continuación:

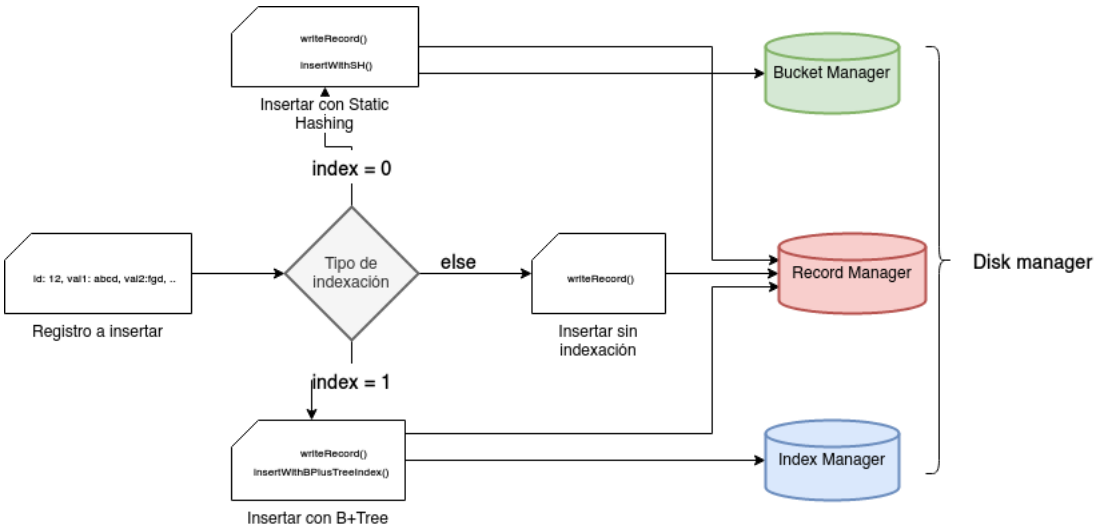


Figure 4: Estructura de la implementación del índice B+Tree

En esta clase disponemos de dos formas de inicialización. La primera, crea por defecto un archivo de índices y de registros, mientras que en la segunda ambos archivos los pasamos como parámetros del constructor. Esto último nos permite tener persistencia en los datos, ya que si usáramos la configuración por defecto siempre se estaría sobre-escribiendo sobre el mismo archivo. En cambio, en la segunda forma podemos volver a utilizar los índices sin necesidad de sobr-escribir los archivos.

Los principales métodos para esta clase son los siguientes:

- **InsertarWithoutIndex():** Insertara un registro de manera secuencial sin utilizar un índice.

- **findWithoutIndex():** busca de manera secuencial un registro dado una key.
- **loadFromExternalFile():** carga data de un archivo externo.
- **InsertarWithBPlusTreeIndex():** Insertara un registro usando un índice B+Tree.
- **readRecord():** lee un registro dada una key usando un índice B+Tree
- **InsertarWithStaticHashing():** Insertara un registro usando un índice con Static Hashing
- **readRecord_SH():** lee un registro dada una key usando Static Hashing.
- **readRecord_SH():** lee un registro dada una key usando Static Hashing.
- **readRecordRange():** realiza una búsqueda por rango usando Static Hashing

Si se desea ver el código fuente y/o un explicación más técnica de cada función implementada se puede dirigir al repositorio del proyecto con este **enlace** en GitHub.

3 Resultados Experimentales

Todas las pruebas se realizaron con diferentes tamaños de registros de una misma data de prueba. Para ello se creo un archivo diferente para cada tamaño. Estas medidas fueron 1k (10^3 registros), 10k (10^4 registros), 100k (10^5 registros) y 1M (10^6 registros) respectivamente. Todas las pruebas se realizaron en una computadora *HP Notebook 15 CoreI5* de 8GB de memoría RAM y 4 núcleos.

3.1 Insertar registros

Se Insertaron diferentes volúmenes de datos usando índexación y sin indexación.

3.1.1 B+Tree

B+Tree de orden 3				
	Tiempo (ms)			
Iteración	Insertar 1k	Insertar 10k	Insertar 100k	Insertar 1M
1	40	506	5754	62000
2	38	544	5442	63556
3	42	592	5248	58902
4	46	486	4986	56574
5	42	469	4745	58336
Promedio	41,6	519,4	5235	59873,6

Table 1: Inserción de 10^3, 10^4, 10^5 y 10^6 registros usando un índice B+Tree.

3.1.2 Static Hashing

Static Hashing				
	Tiempo (ms)			
Iteración	Insertar 1k	Insertar 10k	Insertar 100k	Insertar 1M
1	10	97	1542	30363
2	10	103	1128	39649
3	9	113	1207	53470
4	19	80	1308	88453
5	10	100	1466	71911
Promedio	11,6	98,6	1330,2	56769,2

Table 2: Inserción de 10^3, 10^4, 10^5 y 10^6 registros usando un índice Static Hashing.

3.1.3 Sin Índices

Sin Índices				
	Tiempo (ms)			
Iteración	Insertar 1k	Insertar 10k	Insertar 100k	Insertar 1M
1	4	59	514	7298
2	4	41	426	5559
3	5	43	459	3259
4	5	46	388	3965
5	3	40	465	4464
Promedio	4,2	45,8	450,4	4909

Table 3: Inserción de 10^3, 10^4, 10^5 y 10^6 registros sin índice.

3.2 Buscar un registro dado un valor de key

Para ello en cada prueba se buscó el registro que se encuentra en la mitad de todos los demás. Por ejemplo, si el volumen de datos es de 10^3 se buscó el registro cuyo id es 500, si el volumen es 10^4 se buscó el registro 5000 y así sucesivamente.

3.2.1 B+Tree

B+Tree de orden 3				
Tiempo (ms)				
Iteración	Buscar en 1k	Buscar en 10k	Buscar en 100k	Buscar en 1M
1	∼0	∼0	∼0	∼0
2	∼0	∼0	∼0	∼0
3	∼1	∼0	∼0	∼0
4	∼0	∼0	∼0	∼0
5	∼0	∼1	∼0	∼0
Promedio	∼0	∼1	∼0	∼0

Table 4: Buscar un registro dada una key en 10^3, 10^4, 10^5 y 10^6 registros usando un índice B+Tree.

B+Tree de orden 3				
Accesos a Disco				
Iteración	Buscar en 1k	Buscar en 10k	Buscar en 100k	Buscar en 1M
1	10	13	17	20
2	10	13	17	20
3	10	13	17	20
4	10	13	17	20
5	10	13	17	20
Promedio	10	13	17	20

Table 5: Accesos a disco al buscar un registro dada una key en 10^3, 10^4, 10^5 y 10^6 registros usando índice B+Tree.

3.2.2 Static Hashing

Static Hashing				
Tiempo (ms)				
Iteración	Buscar en 1k	Buscar en 10k	Buscar en 100k	Buscar en 1M
1	∼0	∼0	∼0	∼0
2	∼0	∼0	∼0	∼0
3	∼1	∼1	∼0	∼0
4	∼0	∼0	∼0	∼0
5	∼0	∼0	∼0	∼0
Promedio.	∼0	∼1	∼0	∼0

Table 6: Buscar un registro dada una key en 10^3, 10^4, 10^5 y 10^6 registros usnado un índice Static Hashing.

Static Hashing				
Accesos a Disco				
Iteración	Buscar en 1k	Buscar en 10k	Buscar en 100k	Buscar en 1M
1	2	2	2	4
2	2	2	2	4
3	2	2	2	4
4	2	2	2	4
5	2	2	2	4
Promedio	2	2	2	4

Table 7: Accesos a disco al buscar un registro dada una key en 10^3, 10^4, 10^5 y 10^6 registros usando índice Static Hashing.

3.2.3 Sin Índices

Sin Índices				
Tiempo (ms)				
Iteración	Buscar en 1k	Buscar en 10k	Buscar en 100k	Buscar en 1M
1	3	15	138	1646
2	2	15	121	1321
3	1	14	120	2003
4	2	13	148	1183
5	1	15	141	1538
Promedio	1,8	14,4	133,6	1538,2

Table 8: Buscar un registro dada una key en 10^3, 10^4, 10^5 y 10^6 registros sin usar índices.

Sin Índices				
Accesos a Disco				
Iteración	Buscar en 1k	Buscar en 10k	Buscar en 100k	Buscar en 1M
1	500	5000	50000	500000
2	500	5000	50000	500000
3	500	5000	50000	500000
4	500	5000	50000	500000
5	500	5000	50000	500000
Promedio	500	5000	50000	500000

Table 9: Accesos a disco al buscar un registro dada una key en 10^3, 10^4, 10^5 y 10^6 registros sin usar índices.

4 Análisis y Comparación

4.1 Insertar de registros

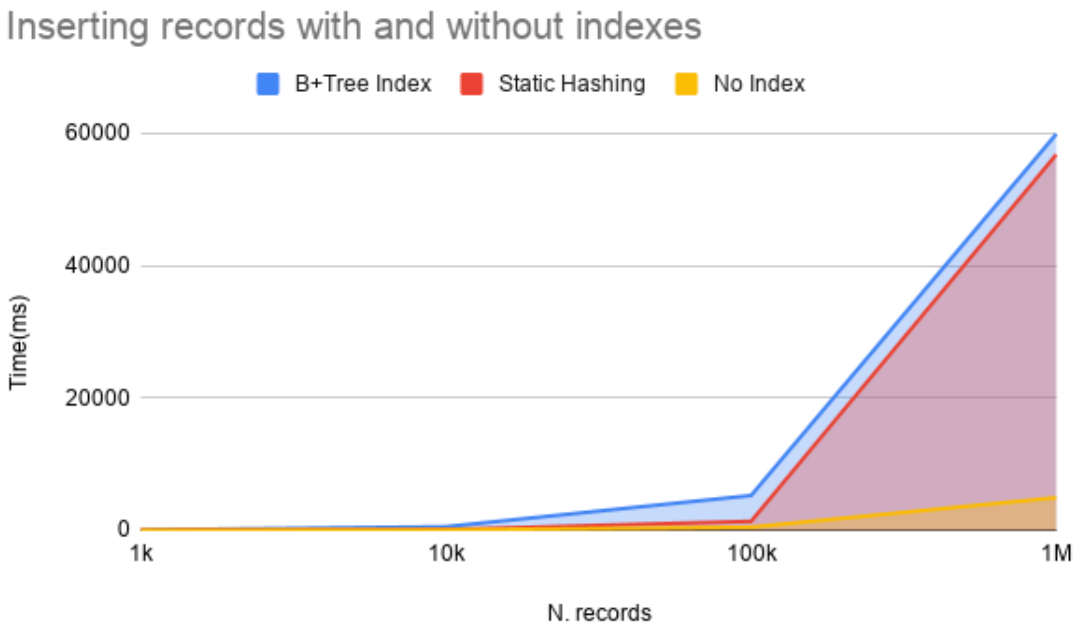


Figure 5: Comparación de la operación de inserción para diferentes volúmenes de datos usando o no indexación.

Como era de esperarse la inserción sin índices es mucho más rápida que la inserción indexada. Esto se debe a que cuando se usa indexación los índices se van actualizando constantemente y esto es más costoso que simplemente insertar un registro sin indexación.

Ahora bien, si centramos las comparaciones a las inserciones indexadas, notaremos que el Static Hashing es ligeramente más rápido que el B+Tree. Una explicación posible a este evento es que el Static Hashing simplemente usa la función hash para obtener la posición que le corresponde al registro y de ahí simplemente lo inserta en dicha posición. En cambio, el índice B+Tree tiene que moverse entre los nodos para encontrar la posición apropiada para la key ingresada, y en algunos casos tendrá que actualizar los índices hacia arriba cuando se produce overflow en un nodo hoja o padre, y como es de esperar estas operaciones consumen tiempo y por lo tanto será más lento que el Static Hashing.

4.2 Buscar un registro dado un valor de key

Es en las operaciones de búsqueda cuando se nota la gran diferencia entre usar o no índices. Como se evidencia en el siguiente gráfico la diferencia es abismal entre no usar un índice y entre usar un índice B+Tree o un Static Hashing.

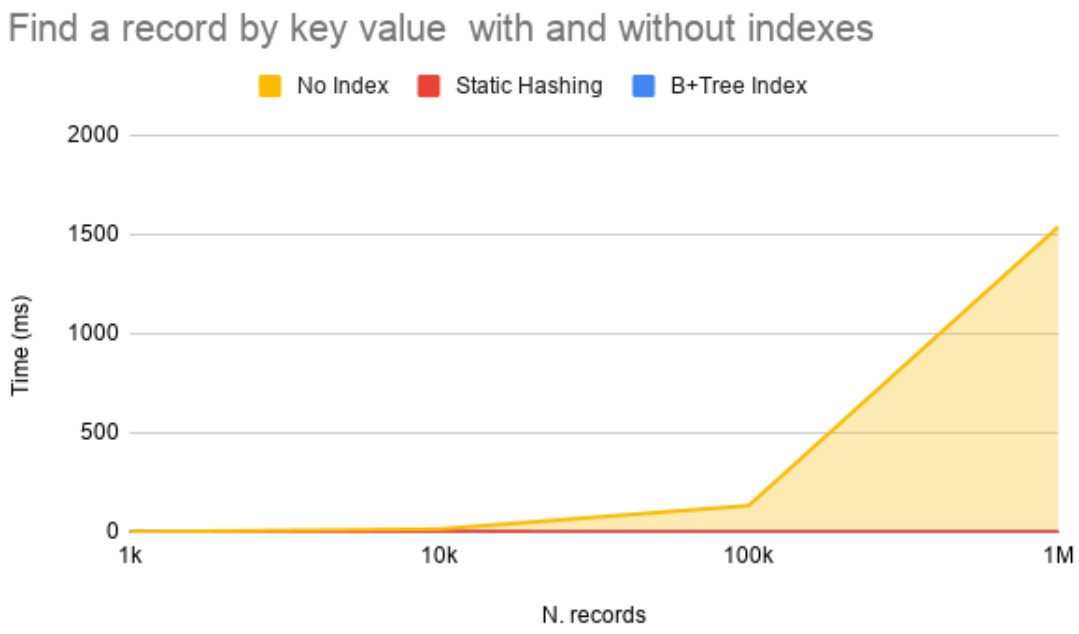


Figure 6: Comparación de la operación de búsqueda dada una key para diferentes volúmenes de datos usando o no indexación.

Sin embargo, hacer una comparación entre indexación y no indexación es bastante injusta, es por ello que centraremos el análisis en las búsquedas usando indexación. Para ello, si volvemos a la tabla 4 y 6, notaremos que el costo en ambos índices es despreciable, en este caso casi cero.

En el caso del B+Tree podemos explicar este fenómeno partiendo del costo computacional de una búsqueda, la cual es $O(log_B N)$, donde B es el orden del árbol y N es la cantidad de registros. Por lo tanto en el peor de los casos tendremos que movernos en $log_B N$ registros, lo cual para una computadora convencional es casi despreciable esa operación.

Por otro lado, en el caso del Static Hashing la búsqueda es incluso más rápida, ya que con la

función hash nos da la posición donde se encuentra el registro y de ahí simplemente accedemos a él en $O(1)$. Sin embargo, dado que las funciones hashing no son perfectas y en muchos casos se producirán colisiones, el tiempo de ejecución de está operación se incrementará, pero de todas maneras el incremento será pequeño en comparación a una búsqueda secuencial.

4.3 Accesos a disco al buscar un registro dado un valor de key

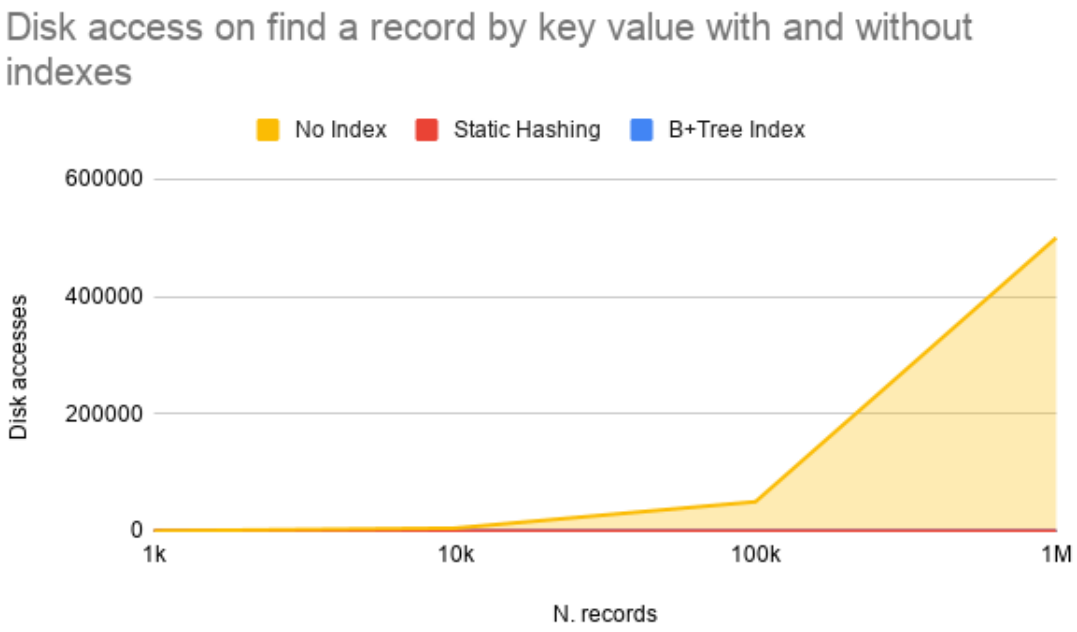


Figure 7: Comparación de la operación de búsqueda de un registro dada una key para diferentes volúmenes de datos usando o no indexación.

Ahora bien si realizamos el análisis en la cantidad de accesos a disco notaremos nuevamente la diferencia abismal entre usar o no usar indexación. Sin embargo, si centramos el análisis en la cantidad de accesos utilizando indexación notaremos que el B+Tree realiza más lecturas a disco que el Static Hashing. Esto se debe a que el B+Tree tiene que moverse desde el nodo raíz hasta los nodos hojas para llegar al registro buscado, lo cual tiene un costo de $log_B N$ accesos a disco, donde B es el orden del árbol y N es la cantidad de registros. En cambio, con el Static Hashing simplemente se realiza un acceso a disco una vez obtenido el hash. Sin embargo, como se mencionó anteriormente las funciones hashing no son perfectas y por lo tanto las colisiones provocarán un aumento en los accesos a disco que puede incluso llegar a ser mayor que en el B+Tree.

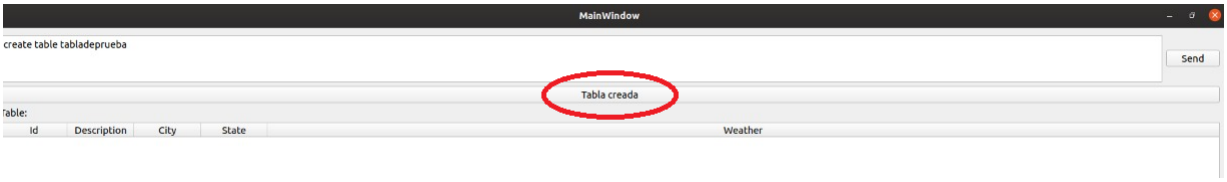
5 Pruebas de Uso

Para probar las consultas diseñamos una aplicación haiendo uso de QtCreator. Las consultas que podemos realizar en la plataforma son las siguientes:

- `CREATE TABLE NOMBRE_DE_LA_TABLA`: Crea una nueva tabla y su índice B+Tree.
- `INSERT INTO NOMBRE_DE_LA_TABLA FROM NOMBRE_DEL_ARCHIVO`: Inserta en una tabla todos los registros de un archivo.
- `INSERT INTO NOMBRE_DE_LA_TABLA VALUES (VALOR1, VALOR2, ...)`: Insertar un nuevo registro en una tabla.
- `SELECT * FROM NOMBRE_DE_LA_TABLA`: Selecciona todos los datos de la tabla.
- `SELECT * FROM NOMBRE_DE_LA_TABLA WHERE KEY = ID_VALUE`: Selecciona el registro correspondiente al id.
- `SELECT * FROM NOMBRE_DE_LA_TABLA WHERE KEY BETWEEN ID_A AND ID_B`: Selecciona todos los registros con id entre A y B.

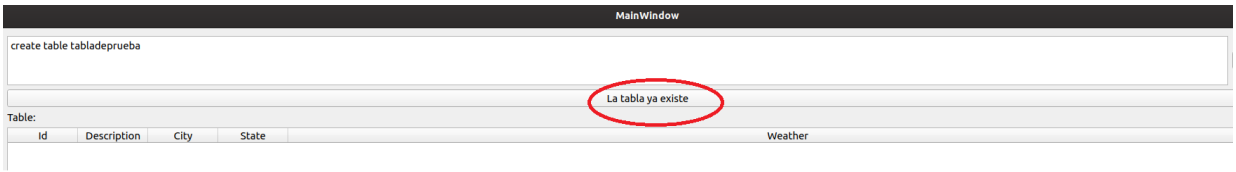
Para cada consulta se recibirá una respuesta en el botón debajo de la consulta. A continuación mostramos los resultados de los diferentes tiempos de consultas.

(a) Creando tabla



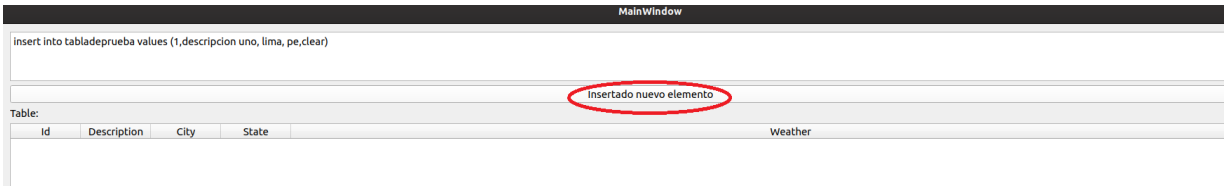
Creamos una tabla que consiste de dos archivos, uno con los records y otro con su índice b+ tree, estos archivos se guardan en la carpeta del proyecto qt.

(b) Creando tabla repetida



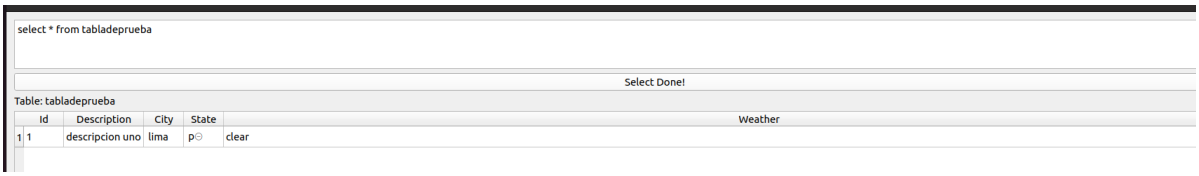
Intentamos crear la misma tabla, pero como ya está dentro de la carpeta del proyecto recibimos una respuesta negativa a la query.

(c) Insertar en una tabla



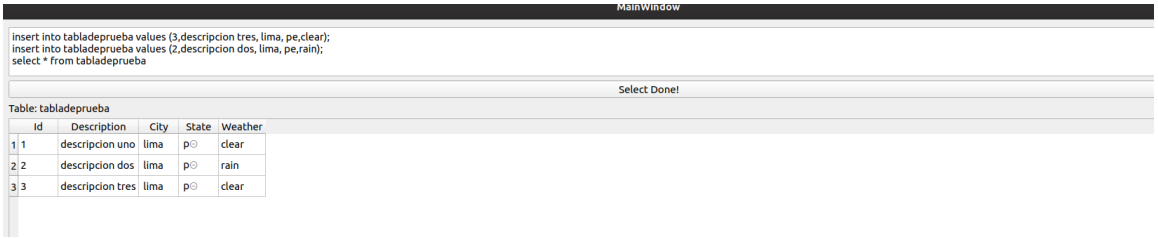
Insertaremos un nuevo registro con los valores `(1, descripcion uno, lima, pe, clear)` en la tabla, recibimos un mensaje de éxito.

(d) Selección en una tabla



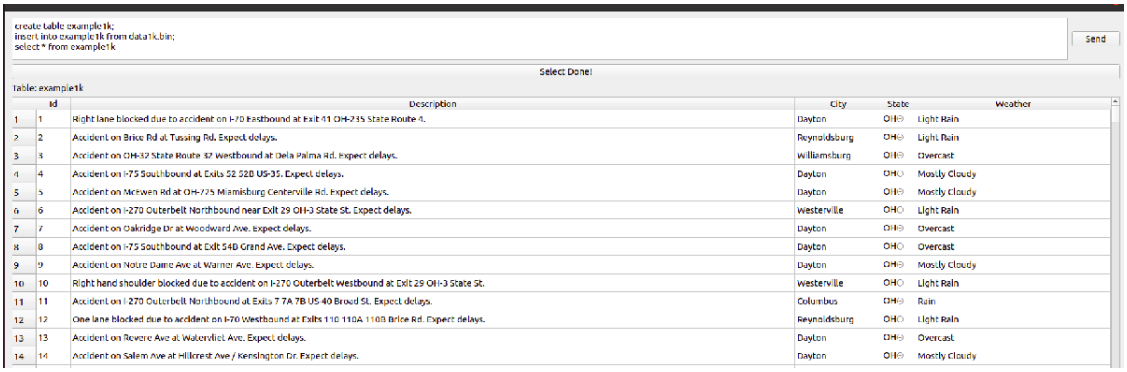
Seleccionamos todos los registros de la tabla, estos se mostrarán en pantalla.

(e) Instrucciones en conjunto



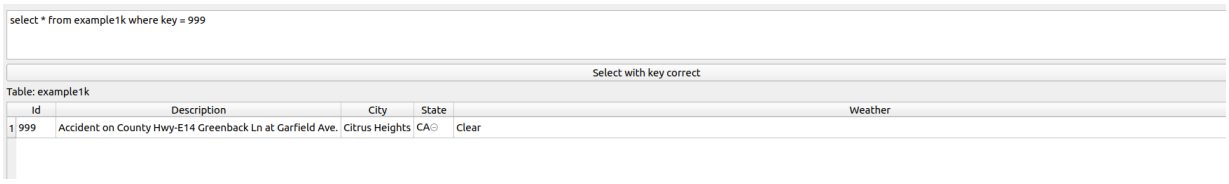
Podemos realizar varias queries juntas, para eso debemos separarlas por ";"

(f) Cargar tabla con un archivo



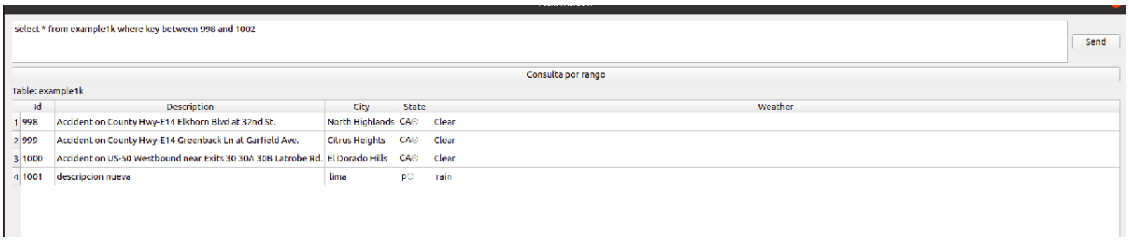
Para llenar una tabla con los valores de un archivo debemos pasarle el nombre de tu nuestro archivo de datos.

(g) Búsqueda por índice



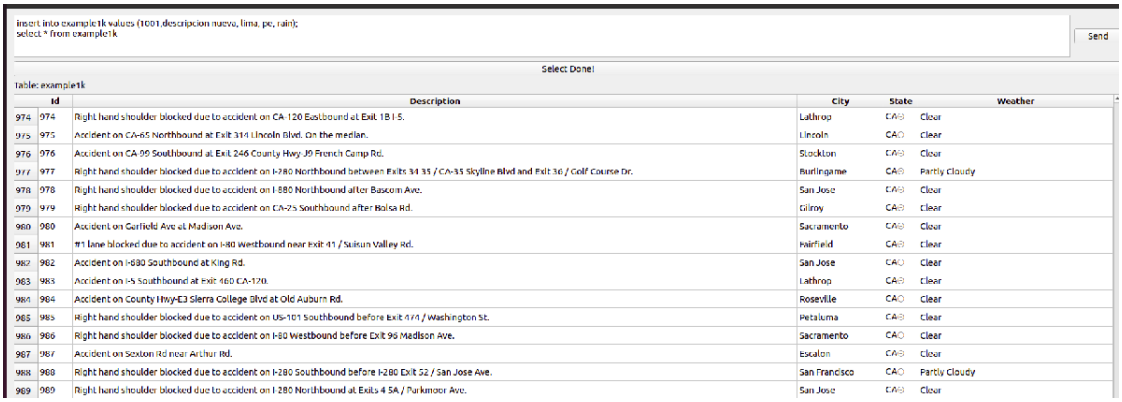
Buscamos un registro cuya primary key sea igual a 999 y la mostramos.

(h) Consulta por rango



Buscamos los registros cuya primary key este entre los valores 998 y 1002 y los mostramos.

(i) Insertar nuevo registro en una tabla con mil datos



Insertaremos un nuevo registro en el archivo con mil registros, este nuevo registro se puede encontrar al final de la tabla.

6 Conclusiones

Podemos concluir que los resultados experimentales fueron los esperados. Las estructuras con las que se trabaja para B+ Tree y Static Hashing han permitido obtener un buen tiempo de respuesta ante consultas sobre un registro o rango de registros. Además, han logrado compensar el retraso que se genera en las inserciones con un excelente performance a nivel de tiempo de ejecución y accesos a disco al realizar las búsquedas.

La comparación entre ambos índices muestra que el Static Hashing logra un mejor rendimiento para la recuperación de un registro en particular. Sin embargo, como ya se explicó previamente las colisiones producto de la función hash mermarán este buen desempeño inicial llegando incluso a ser peor que el B+Tree.

Por otro lado, en la inserción de registros se logra un mejor rendimiento nuevamente con el Static Hashing ya que la función hash y el tamaño del bucket influyen positivamente en su performance. En cambio, en el B+Tree la actualización de los índices es lo que consume más tiempo al realizar la inserción. Por lo tanto, se hace evidente la necesidad de generar mejor

estructuras que nos permitan un mejor tiempo de respuesta al momento de crear y actualizar los índices, y en el caso del Static Hashing una mejor construcción de la función Hash que nos permita distribuciones mucho más uniformes.

Para finalizar, como se hizo evidente a lo largo de la experimentación el uso de indexación influye significativamente en el tiempo de respuesta de las consultas a pesar de la demora inical al momento de insertar los datos.

7 Referencias

- Moosavi, S. (2020, January 17). US Accidents (3.0 million records). Retrieved from https://www.kaggle.com/sobhanmoosavi/us-accidents/data
- Moosavi, Sobhan, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, and Rajiv Ramnath. “A Countrywide Traffic Accident Dataset.”, 2019.
- Moosavi, Sobhan, Mohammad Hossein Samavatian, Srinivasa Parthasarathy, Radu Teodorescu, and Rajiv Ramnath. ”Accident Risk Prediction based on Heterogeneous Sparse Data: New Dataset and Insights.” In proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM, 2019.