Informe del Proyecto 1

CS2702 – Base de Datos 2

Matias Meneses Sebastian Nieto Guillermo Galvez Zamir Lizardo Jorge Flores
Universidad de Ingeniería y Tecnología (UTEC)
{matias.meneses,hector.nieto,jose.galvez.p,zamir.lizardo,jorge.flores.b}@utec.edu.pe

***Resumen*—El presente trabajo tiene como objetivo implementar y analizar diversas técnicas de indexación como archivos secuenciales, ISAM, B+Tree, Hashing Extensible y R-Tree. Se evalúan los tiempos de ejecución y los accesos a memoria secundaria.**

***Index Terms*—Indexación, B+Tree, ISAM, Hashing Extensible, R-Tree, parser SQL.**

I. Introducción

I-A. Contexto General

En los sistemas de bases de datos, las estructuras de indexación son fundamentales para optimizar el acceso y la manipulación de grandes volúmenes de información. Estas permiten reducir los accesos a memoria secundaria, mejorar los tiempos de búsqueda e incrementar la eficiencia general de las operaciones de inserción y eliminación.

I-B. Objetivo del Proyecto

El objetivo principal del proyecto es implementar y analizar diferentes técnicas de indexación sobre archivos, tanto primarios como secundarios. A través de un entorno controlado, se busca comparar el rendimiento de estructuras como archivos secuenciales, ISAM, B+Tree, Hashing Extensible y R-Tree, evaluando su eficiencia en términos de accesos a memoria y tiempo de ejecución.

I-C. Aplicación Propuesta

Se desarrolló una aplicación modular que permite realizar operaciones de búsqueda, inserción y eliminación mediante un *parser SQL* propio. Esta interfaz simula el funcionamiento de un sistema gestor de bases de datos, facilitando la interacción con los índices implementados y permitiendo analizar su desempeño bajo distintos escenarios de uso.

II. Implementación

II-A. Modelo de Registros

El sistema trabaja con dos tipos fundamentales de registros para optimizar el uso de memoria:

- Record: Representa un registro completo con todos sus campos. Utiliza `struct` de Python para serialización binaria de tamaño fijo. Soporta tipos: `INT`, `FLOAT`, `CHAR`, `ARRAY` (datos espaciales) y `BOOL`.
- IndexRecord: Estructura ligera para índices secundarios con solo dos campos: `index_value` (valor indexado) y `primary_key` (referencia al registro completo). Esta separación mantiene los índices secundarios compactos.

II-B. Clasificación de Índices

- Índices Primarios: Almacenan físicamente los registros completos y retornan objetos `Record`. Técnicas implementadas: Sequential File, ISAM, B+Tree Clustered.
- Índices Secundarios: Indexan campos alternativos (no clave primaria) usando `IndexRecord`s para permitir búsquedas eficientes sobre cualquier campo de la tabla. Técnicas implementadas: Extendible Hashing, B+Tree Unclustered, R-Tree.

II-C. DatabaseManager

Componente central que orquesta todas las operaciones del sistema. Coordina múltiples índices por tabla: un índice primario obligatorio y cero o más índices secundarios opcionales.

Responsabilidades principales:

- Creación y gestión del ciclo de vida de tablas e índices
- Coordinación de operaciones entre índices primarios y secundarios
- Mantenimiento de consistencia referencial entre índices
- Agregación de métricas de rendimiento de operaciones compuestas

II-D. Sistema de Métricas

Cada operación retorna un `OperationResult` con:

- Datos resultantes y tiempo de ejecución
- Contadores de lecturas/escrituras a disco
- Indicador de reorganización de archivos
- Desglose de métricas por índice (primario/secundarios)

El `PerformanceTracker` registra automáticamente cada acceso a disco mediante `track_read()` y `track_write()`, soportando operaciones anidadas para medir operaciones compuestas.

II-E. Flujo de Operaciones y Consistencia

II-E1. Consultas con Índices Secundarios

Las consultas sobre campos secundarios ejecutan un flujo en dos fases:

1. **Búsqueda en índice secundario**: Retorna lista de `primary_keys` que cumplen la condición
2. **Resolución de registros**: Consulta el índice primario con cada clave para obtener los `Record`s completos

Realizamos esto debido a que mantener las posiciones fisicas de los registros es complejo debido a las operaciones como eliminacion, e insercion que puede modificar las posiciones y tendriamos que actualizarlas en todos los indices secundarios.

II-E2. Mantenimiento de Consistencia

Para garantizar integridad referencial, el `DatabaseManager` coordina:

- **Inserción**: Índice primario primero, luego propagación a índices secundarios
- **Eliminación**: Índices secundarios primero (evita referencias huérfanas), luego índice primario.
- **Sincronización automática**: Todos los índices se mantienen actualizados en cada operación.

Los detalles de gestión de memoria secundaria se explican en cada técnica de indexación.

III. Técnicas Utilizadas

Para nuestro proyecto elegimos las siguientes técnicas de indexación, presentadas cada una con su descripción, detalle de implementación y algoritmos.

III-A. Extendible Hashing

Índice basado en el valor hasheado de los records, utilizado para obtener la dirección de guardado del bucket correspondiente, manteniendo una complejidad O(1) a través de todas sus operaciones. No soporta búsqueda por rangos.

III-A1. Detalle de Implementación

- Utilizamos 2 archivos, `dirfile` y `bucketfile`, donde se guarda la información de los directorios que apuntan hacia los buckets, y la información de los buckets.
- Inicialmente, el indice empieza con una profundidad global de 3 y 2 buckets iniciales con profundidad local de 1.
- Se permite encadenamiento de overflow hasta K buckets, por cada bucket principal, en caso no sea posible dividir el bucket principal.
- Identificamos registros eliminados dentro de los buckets como un paquete de caracteres nulos del tamaño del registro.
- Guardamos una free-list de espacios de memoria secundaria de buckets liberados para reutilizar cuando se crean nuevos buckets.
- Se define un mínimo de elementos `MIN_N` como `BLOCK_FACTOR/2`, para los buckets principales. En caso de tener una menor cantidad a `MIN_N`, se llena el bucket principal de los registros en overflow, y se libera los buckets que queden vacios.

III-A2. Algoritmos

III-A2a. Inserción

Para la inserción, obtenemos el bucket a insertar de acuerdo al valor hash de la clave. Insertamos el IndexRecord si es que hay espacio en el bucket. En el caso contrario, cambiaremos de bucket hacia su overflow si existe e intentaremos de nuevo insertar. En caso no haya mas buckets de overflow y no se haya podido insertar, dividiremos los buckets si es posible. En caso no sea posible, si es que no hemos superado el limite K de overflow, crearemos un bucket de overflow donde insertaremos el IndexRecord. En caso de que no sea posible, duplicaremos el directorio y dividiremos los buckets posteriormente para volver a insertar el IndexRecord.

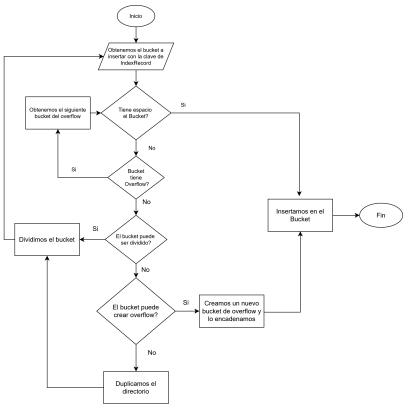

Figura 1: Diagrama de Flujo de Insercion en Extendible Hash

III-A2b. Búsqueda

Para la búsqueda, primero obtenemos el bucket principal a buscar a partir del valor hasheado de la clave. Buscamos dentro del bucket, y cada registro que concuerde con el valor de la clave, guardamos su llave primaria en una lista. Si es que hay overflow, buscamos en todos los buckets de overflow y añadimos las llaves primarias correspondientes a la lista. Si no hay overflow, devolvemos la lista de llaves primarias.

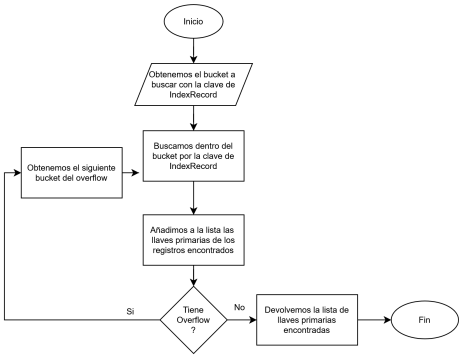

Figura 2: Diagrama de Flujo de Búsqueda en Extendible Hash

III-A2c. Eliminación

Para la eliminacion, tras obtener el bucket donde se encuentra la clave, eliminamos los registros que contienen esa clave, escribiendo caracteres nulos encima de ello. Despues,

en caso haya overflow, realizamos lo mismo en los buckets de overflow. Tras la eliminacion, si el bucket principal tiene menos que la cantidad mínima de elementos definida (BLOCK_FACTOR/2), entonces pasaremos todos los elementos de los buckets de overflow hacia el bucket principal, y si no cabe hacia el overflow. Tras hacer esto, en caso queden buckets de overflow sin elementos, se liberaran. Igualmente si el bucket principal termina sin elementos, se libera y el puntero de directorio que lo apunta sera redireccionado a su bucket hermano. Finalmente se devuelve la lista de llaves primarias de los registros eliminados.

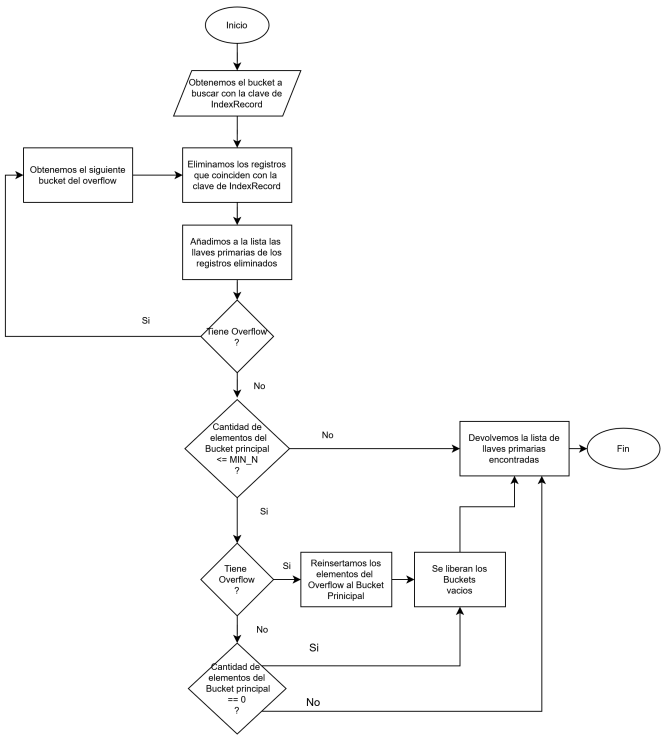

Figura 3: Diagrama de Flujo de Eliminacion en Extendible Hash

III-B. B-Tree+ Clustered

(Describir brevemente la estructura y características del B-Tree+).

III-B1. Detalle de Implementación

(Describir los detalles de implementación del B-Tree+).

III-B2. Algoritmos

(Agregar diagrama de flujo de inserción, búsqueda y eliminación para B-Tree+).

III-C. B-Tree+ Unclustered

(Describir brevemente la estructura y características del B-Tree+).

III-C1. Detalle de Implementación

(Describir los detalles de implementación del B-Tree+).

III-C2. Algoritmos

(Agregar diagrama de flujo de inserción, búsqueda y eliminación para B-Tree+).

III-D. ISAM

Índice secuencial indexado de dos niveles que mantiene un índice raíz (root index) que apunta a páginas del índice hoja (leaf index), las cuales a su vez apuntan a las páginas de datos. Permite búsquedas eficientes mediante navegación jerárquica y soporta overflow chains para manejar desbordamientos cuando las páginas se llenan.

III-D1. Detalle de Implementación

El ISAM implementado utiliza una estructura de tres capas:

- **Root Index**: Primer nivel que contiene entradas que apuntan a páginas del leaf index.
- **Leaf Index**: Segundo nivel con entradas que apuntan directamente a páginas de datos.
- **Data Pages**: Páginas que almacenan los registros ordenados, con soporte para overflow chains cuando se llenan.

Se implementan tres estrategias de manejo de overflow cuando una página de datos se llena:

1. **Split de página de datos**: Si el leaf index tiene espacio, se divide la página llena en dos y se agrega una nueva entrada al leaf index.
2. **Split de leaf index**: Si el leaf index está lleno pero el root index tiene espacio, se divide tanto la página de datos como el leaf index, agregando una nueva entrada al root index.
3. **Overflow chain**: Si ambos índices están llenos, se crea una cadena de overflow enlazando páginas adicionales.

Adicionalmente, se utiliza una free-list para reutilizar páginas liberadas durante eliminaciones.

III-D2. Algoritmos

III-D2a. Inserción

Ante desbordamiento de página, aplica una estrategia en cascada según disponibilidad de espacio en índices: split de página de datos (si leaf index tiene capacidad), split simultáneo de página y leaf index (si root index tiene espacio), u overflow chain reutilizando páginas de la free-list. Dispara reorganización automática cuando el ratio de páginas libres supera 40 % o las cadenas de overflow promedian más de 4 páginas.

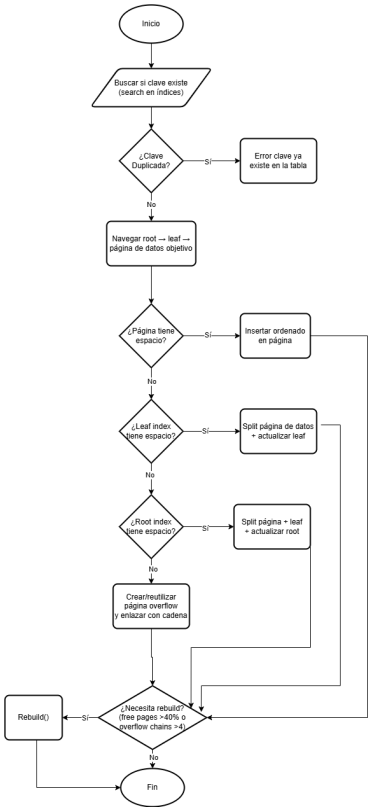

Figura 4: Diagrama de Flujo de Inserción en ISAM

III-D2b. Búsqueda

Utiliza navegación jerárquica en tres niveles: localiza el leaf index mediante búsqueda en root index, identifica la página de datos inicial, y recorre secuencialmente la cadena de overflow (si existe) comparando claves hasta encontrar coincidencia o agotar la cadena.

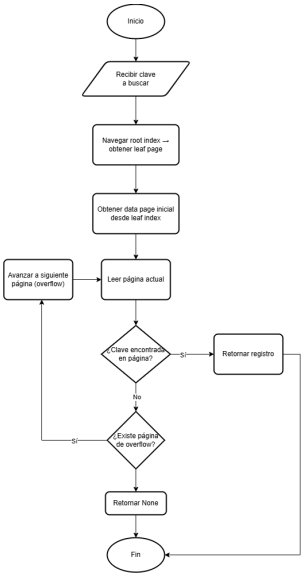

Figura 5: Diagrama de Flujo de Búsqueda en ISAM

III-D2c. Eliminación

Localiza el registro mediante navegación jerárquica y recorrido de overflow chains. Tras eliminar, evalúa si la página queda por debajo del umbral de consolidación (BLOCK_FACTOR/3): intenta fusionar con la página siguiente de la cadena si es posible, libera páginas de overflow vacías agregándolas a la free-list, y dispara rebuild si se superan los umbrales críticos de fragmentación.

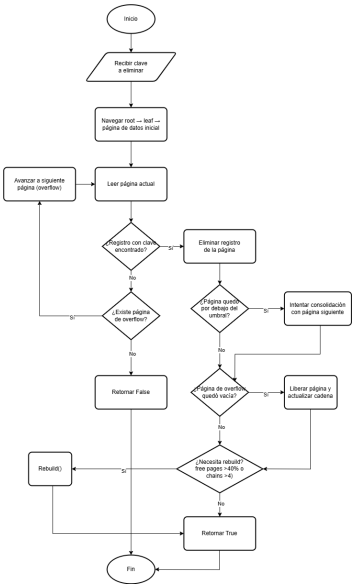

Figura 6: Diagrama de Flujo de Eliminación en ISAM

III-D2d. Búsqueda por rango

Identifica el rango de páginas del leaf index que intersectan con el intervalo de búsqueda mediante recorrido secuencial. Para cada página del leaf index en el rango, recorre todas sus entradas y sus respectivas cadenas de overflow, filtrando registros que cumplan la condición de rango. Finalmente ordena los resultados por clave antes de retornarlos.

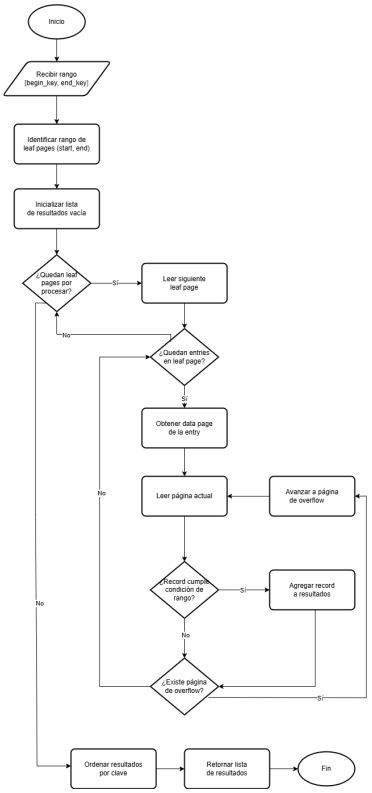

Figura 7: Diagrama de Flujo de Búsqueda por Rango en ISAM

III-E. R-Tree

Índice espacial para datos multidimensionales, basado en rectángulos mínimos. Optimizando consultas de punto, intersección, búsqueda por radio y KNN.

III-E1. Detalle de Implementación

Para el R-Tree, utilizamos una estructura jerárquica basada en Rectángulos Delimitador Mínimos (MBR), la cual permite indexar eficientemente datos espaciales en múltiples dimensiones (es 2D por defecto). Cada registro puntual se representa como un rectángulo.

En nuestra implementación, empleamos la librería `rtree` propia de Python. Cada nodo del árbol contiene un conjunto de MBRs asociados a identificadores (*primary keys*), y las divisiones se producen automáticamente según los algoritmos internos de balanceo y expansión del R-Tree.

Se definen las operaciones básicas de inserción, búsqueda (punto, por radio y KNN) y eliminación, con validaciones para las dimensiones y para los tipos de coordenadas.

III-E2. Algoritmos

III-E2a. Inserción

Para la inserción, se valida que las coordenadas sean una secuencia numérica de dimensión d. Luego se construye el MBR basado en el punto y se inserta en el índice junto con su record primaria. La librería maneja automáticamente las divisiones de nodos en caso de overflow.

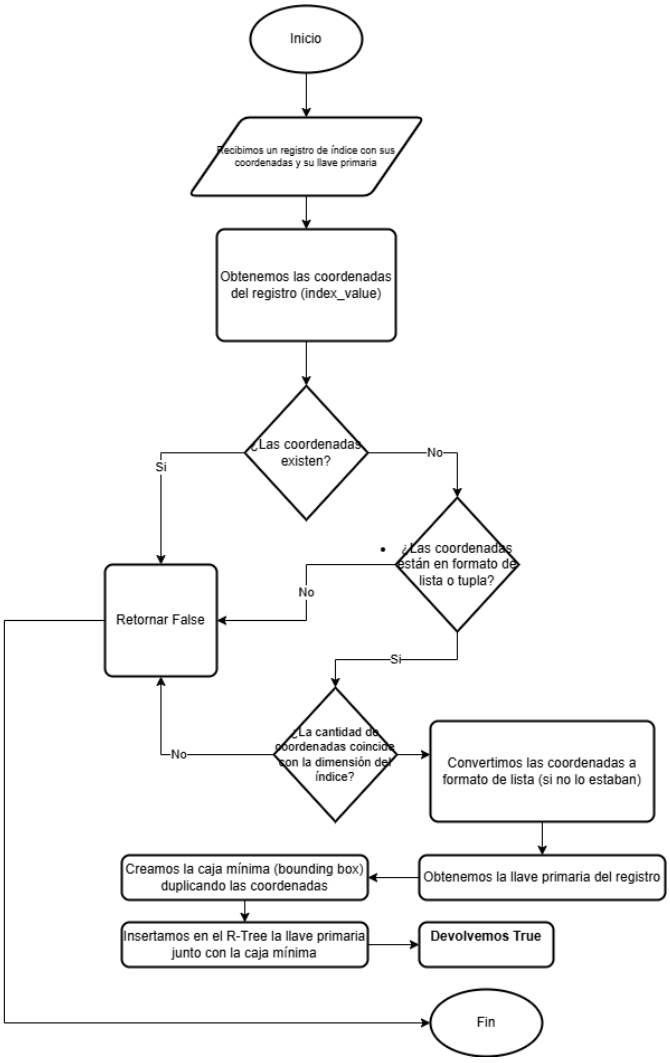

Figura 8: Diagrama de Flujo de Inserción en R-Tree

III-E2b. Búsqueda Puntual

Para la búsqueda puntual, se obtiene el conjunto de registros cuyos MBRs intersectan exactamente con el punto consultado. La operación incluye tanto los nodos hoja como los niveles superiores del árbol cuando aplican.

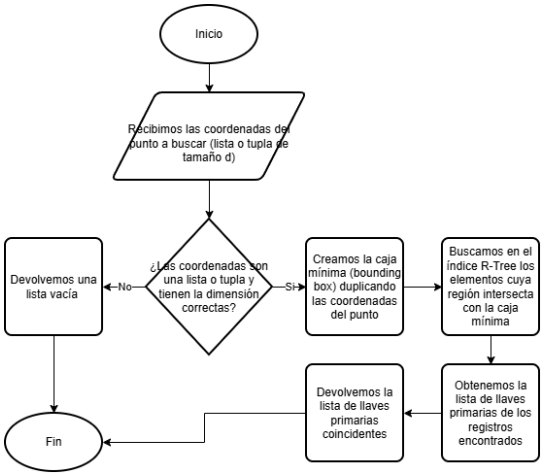

Figura 9: Diagrama de Flujo de la Búsqueda Puntual en R-Tree

III-E2c. Búsqueda por Radio

La búsqueda por radio obtiene todos los objetos cuyo MBR intersecta con la ventana cuadrada definida por el centro y el radio. De manera opcional, puede aplicarse un filtro posterior por distancia euclidiana para refinar los resultados.

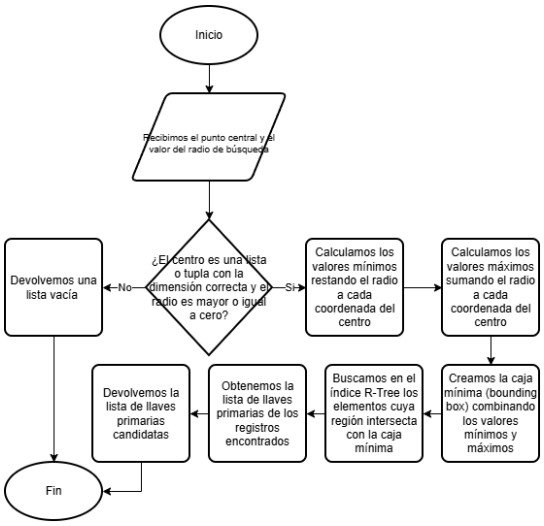

Figura 10: Diagrama de Flujo de la Búsqueda por Radio en R-Tree

III-E2d. Búsqueda k-NN

La búsqueda de los k vecinos más cercanos (k-NN) en un R-Tree permite encontrar los objetos con menor distancia respecto a un punto dado. Para ello, se toma el punto de consulta y se localizan los MBRs más próximos dentro del índice. El proceso recupera un conjunto de candidatos a partir de la operación de vecindad sobre el árbol, de los cuales se seleccionan los k objetos más cercanos según la distancia calculada.

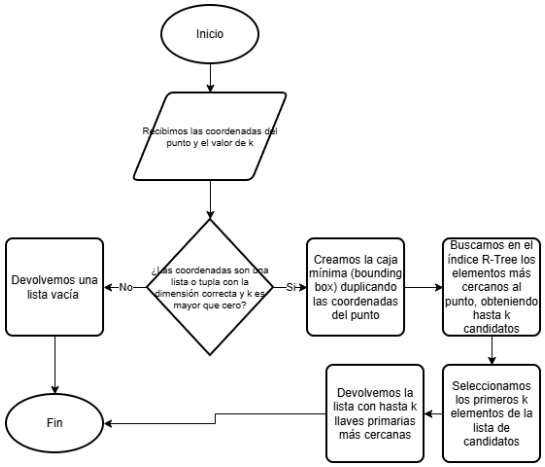

Figura 11: Diagrama de Flujo de la Búsqueda de k vecinos más cercanos en R-Tree

III-E2e. Eliminación

La eliminación elimina uno o varios registros cuyos MBRs coinciden con el punto dado. Si se proporciona la `primary_key`, la operación es directa; en caso contrario, se eliminan todas las coincidencias encontradas en la intersección.

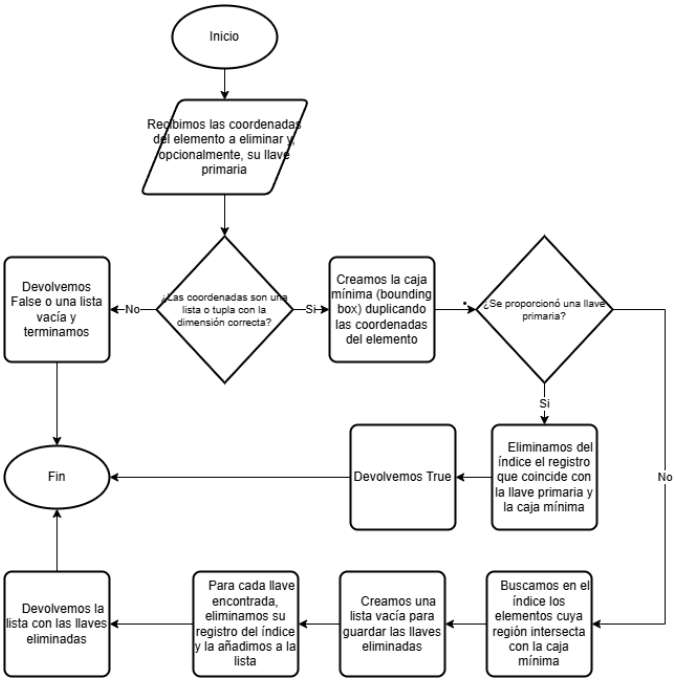

Figura 12: Diagrama de Flujo de la Eliminación en R-Tree

III-F. Sequential File

Estructura secuencial con archivo principal (*main*) ordenado y archivo auxiliar (*aux*) desordenado para inserciones rápidas. Utiliza borrado lógico mediante campo `active` y reorganización automática cuando el aux excede un umbral dinámico k.

III-F1. Detalle de Implementación

El Sequential File implementado utiliza dos archivos físicos:

- **Main**: Archivo principal con registros ordenados por clave y tamaño fijo, optimizado para búsqueda binaria.
- **Aux**: Archivo auxiliar desordenado donde se realizan inserciones en tiempo constante.

El umbral k se calcula dinámicamente como $k = \lfloor \log_2(n) \rfloor$, donde n es el número total de registros activos. Cuando $|aux| > k$, se dispara una reorganización que fusiona ambos archivos en un nuevo main ordenado.

El borrado es lógico mediante el campo booleano `active`. Si los registros eliminados superan el 10 % del total, se ejecuta una reorganización para recuperar espacio y mantener la eficiencia.

III-F2. Algoritmos

III-F2a. Inserción

Para la inserción, primero se verifica que la clave no exista. El registro se agrega al archivo auxiliar en tiempo constante. Si es necesario, se ejecuta una reorganización automática.

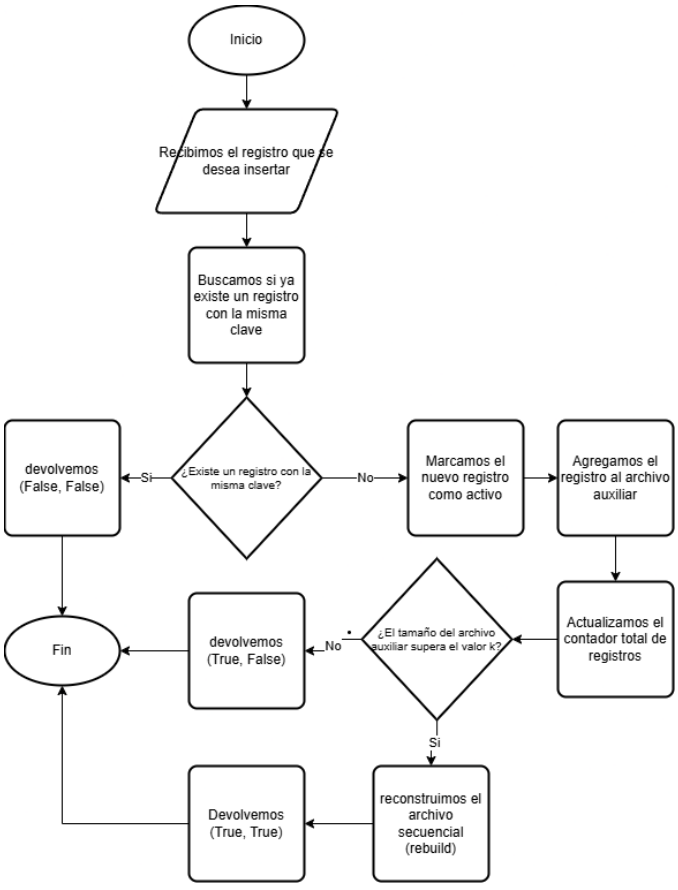

Figura 13: Diagrama de Flujo de la Inserción en el Sequential File

III-F2b. Búsqueda

La búsqueda realiza primero una búsqueda binaria en el archivo main ordenado. Si no encuentra el registro, escanea linealmente el archivo aux hasta localizar la clave buscada.

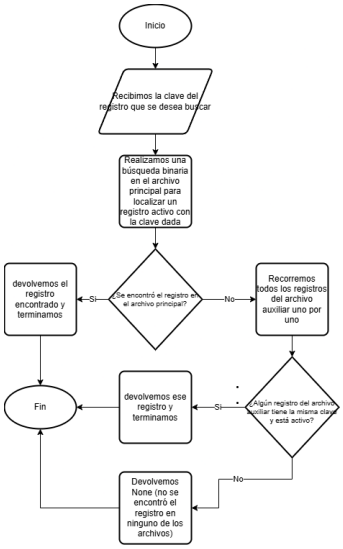

Figura 14: Diagrama de Flujo de la Búsqueda de un registro en el Sequential File

III-F2c. Eliminación

La eliminación es lógica, marcando el campo `active` como falso. Se busca primero en main usando búsqueda binaria y luego en aux con escaneo lineal.

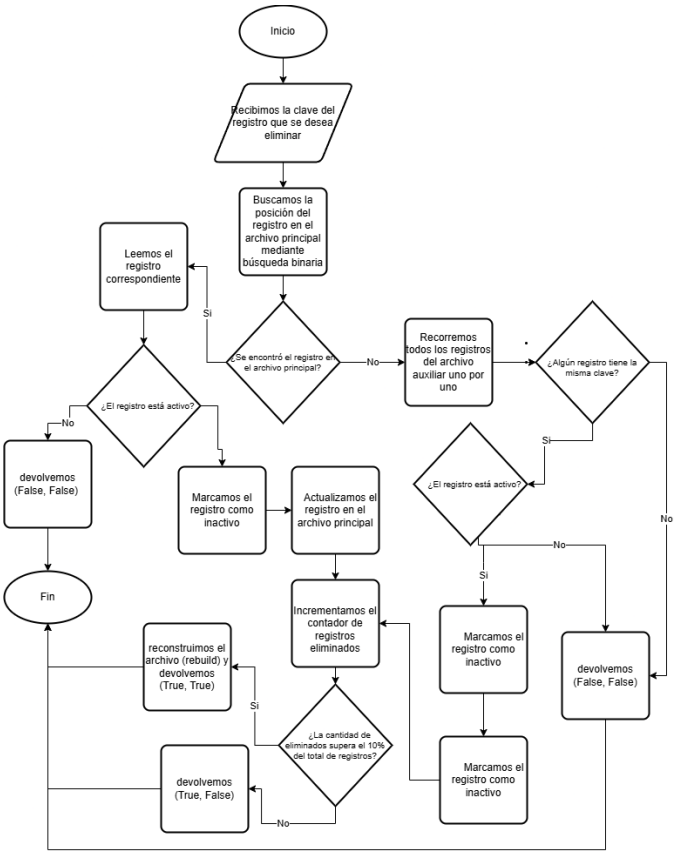

Figura 15: Diagrama de Flujo de la Eliminación (borrado lógico) en el Sequential File

III-F2d. Búsqueda por rango

La búsqueda por rango localiza la posición inicial en main usando lower bound, luego recorre secuencialmente hasta superar el límite superior. Además, filtra los registros del aux que caen dentro del rango y ordena todos los resultados.

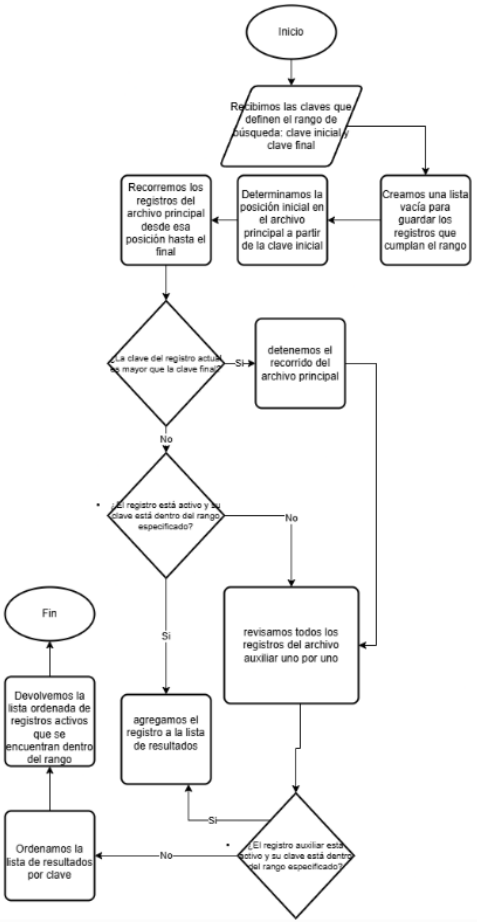

Figura 16: Diagrama de Flujo de la Búsqueda por rango en el Sequential File

III-F2e. Reorganización

La reorganización recopila todos los registros activos de ambos archivos, los ordena y los escribe en un nuevo main. El aux se vacía y se recalcula el umbral k basado en el nuevo total de registros.

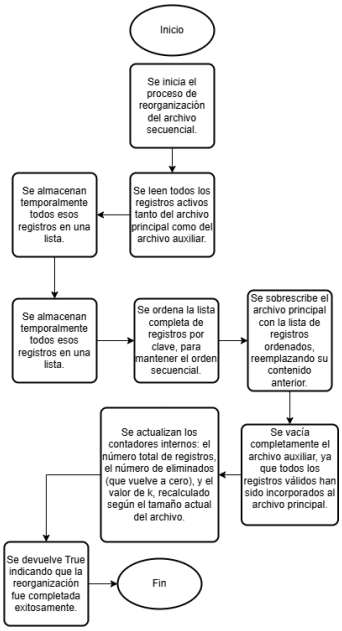

Figura 17: Diagrama de Flujo de la reorganización (Rebuild) del Sequential File

III-G. B+Tree Clustered

III-G1. Detalle de Implementación

Índice donde los datos se almacenan físicamente en las hojas del árbol en orden de clave primaria. El índice y los datos están juntos.

III-G2. Algoritmos

III-G2a. Inserción

La operación de inserción en un B+Tree clustered consiste en agregar un nuevo registro manteniendo el orden físico de los datos por clave primaria. El proceso garantiza que los registros se almacenen secuencialmente en las hojas del árbol, optimizando las consultas por rango y preservando la estructura balanceada del índice.

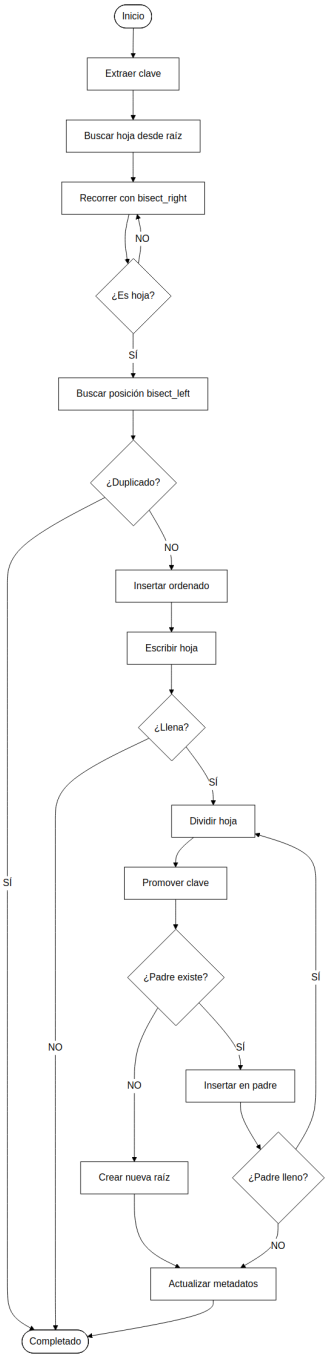

Figura 18: Diagrama de Flujo de la insercion de B+ Tree CLustered

La inserción inicia localizando la hoja adecuada mediante búsqueda binaria desde la raíz. Una vez encontrada la posición correcta, se verifica que no existan claves duplicadas. El registro se inserta manteniendo el orden secuencial y, si la hoja excede su capacidad, se divide en dos para preservar el balance del árbol. Las divisiones pueden propagarse hasta la raíz, requiriendo actualizaciones en los metadatos del índice.

III-G2b. Búsqueda

El algoritmo de búsqueda recorre el árbol desde la raíz hasta la hoja correspondiente usando búsqueda binaria en cada nivel. Al encontrar la hoja, realiza una búsqueda binaria final para ubicar la clave exacta. Si la clave existe, retorna el registro completo inmediatamente; de lo contrario, retorna nulo. La complejidad es logarítmica garantizando acceso eficiente a los datos.

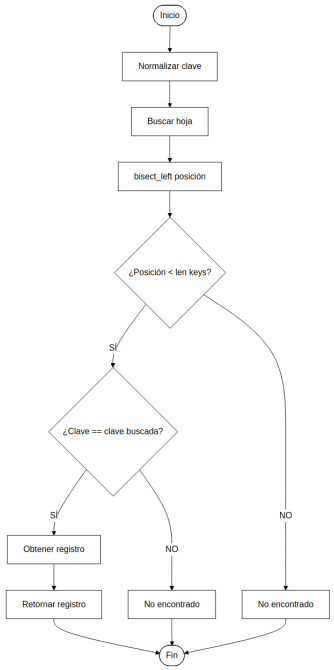

Figura 19: Diagrama de Flujo del search B+Tree Clustered

III-G2c. Eliminación

El algoritmo elimina el registro buscándolo por clave primaria. Si existe, lo remueve de la hoja y verifica underflow. En caso de underflow, realiza préstamo de hermanos o fusión de nodos, propagando los cambios hacia arriba cuando es necesario para mantener el balance del árbol.

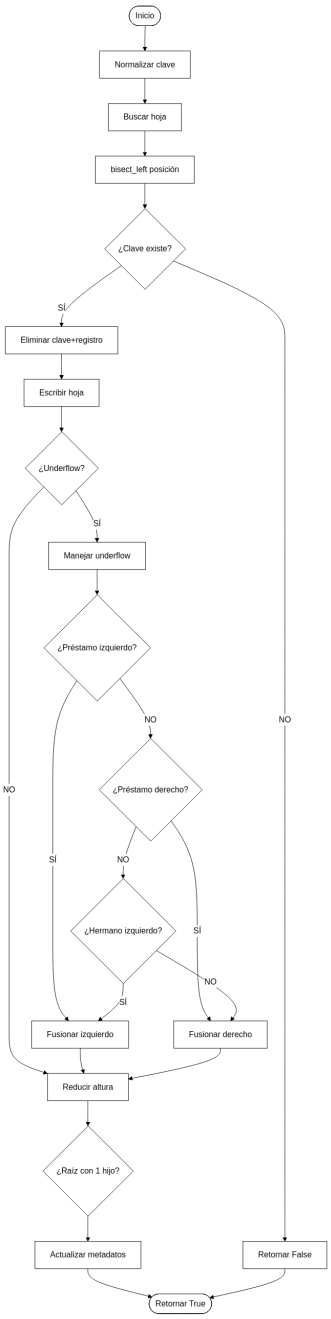

Figura 20: Diagrama de Flujo del delete B+Tree Clustered

III-G2d. Búsqueda por rango

Localiza la hoja inicial que contiene el inicio del rango y recorre secuencialmente las hojas usando los enlaces entre ellas. Colecta todos los registros cuyas claves estén dentro del rango especificado, deteniéndose cuando encuentra una clave mayor al límite superior.

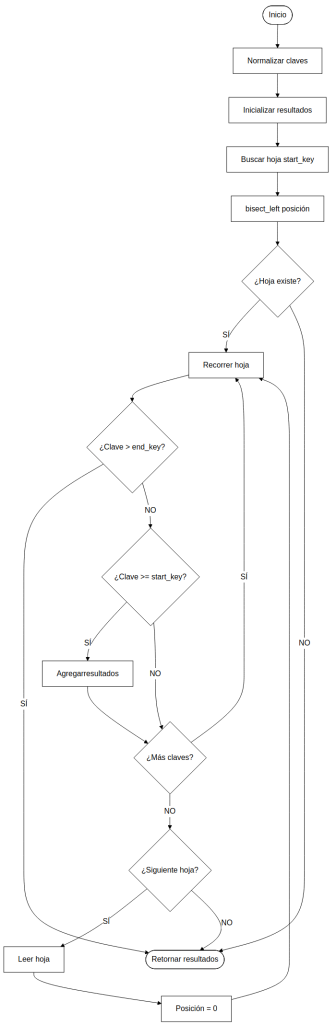

Figura 21: Diagrama de Flujo del range search B+Tree Clustered

III-H. B+Tree Unclustered

III-H1. Detalle de Implementación

Las hojas contienen pares (clave secundaria, clave primaria). Los datos reales están separados.

III-H2. Algoritmos

III-H2a. Inserción

Inserta pares (clave secundaria, clave primaria) manteniendo duplicados ordenados. Para claves secundarias iguales, ordena internamente por clave primaria. Realiza divisiones que preservan grupos de duplicados juntos.

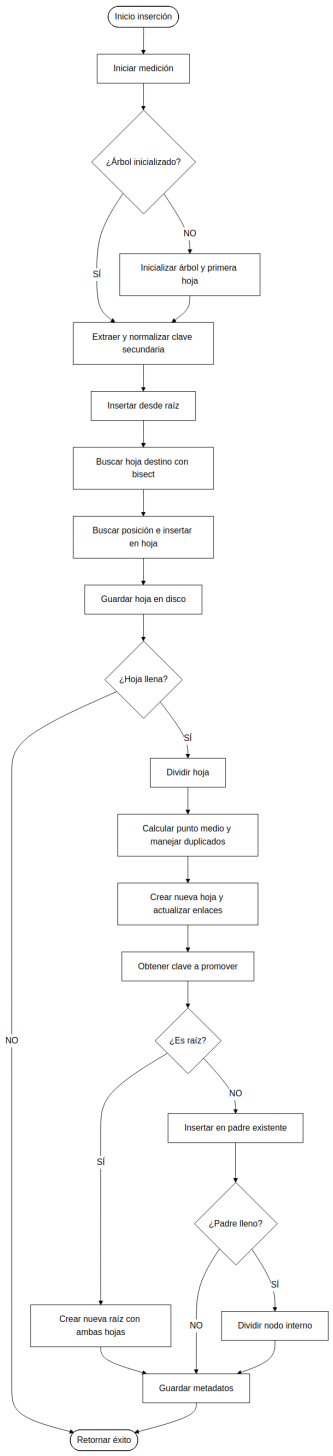

Figura 22: Diagrama de Flujo del insert B+Tree Unclustered

III-H2b. Búsqueda

Busca todas las claves primarias asociadas a una clave secundaria, recorriendo múltiples hojas si es necesario debido a duplicados. Retorna una lista de claves primarias que puede estar vacía o contener múltiples elementos.

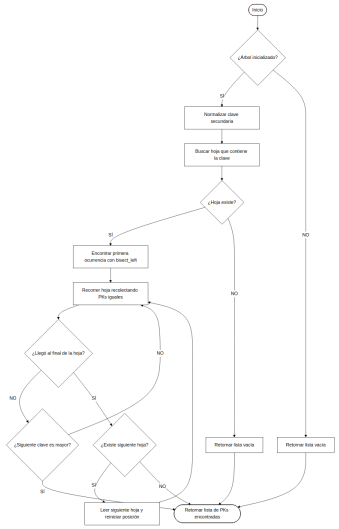

Figura 23: Diagrama de Flujo del search B+Tree Unclustered

III-H2c. Eliminación

Elimina entradas con dos modos: específico (clave secundaria + primaria exacta), busca el par exacto recorriendo todas las hojas que contengan la clave secundaria hasta encontrar la clave primaria coincidente. Una vez encontrado, elimina solo ese par específico y maneja underflow si es necesario.

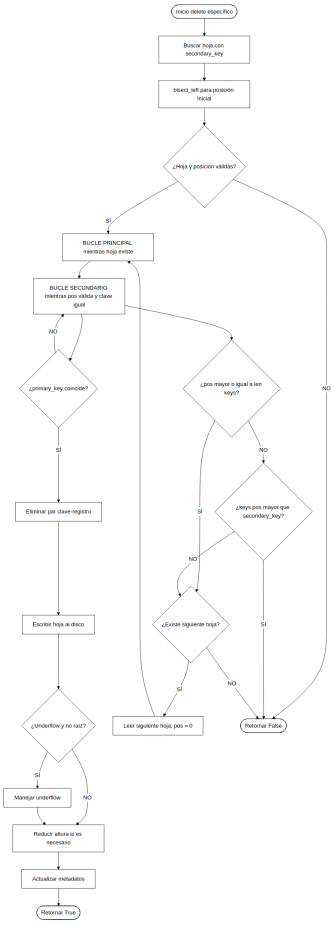

Figura 24: Diagrama de Flujo del delete B+Tree Unclustered

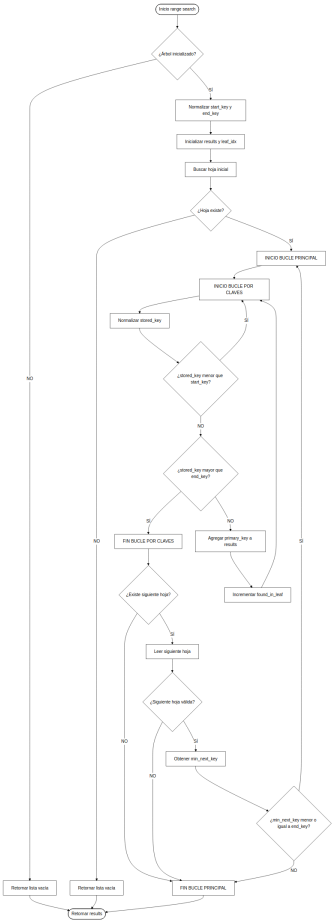

Figura 25: Diagrama de Flujo del range search B+Tree Unclustered

III-H2d. Búsqueda por rango

Encuentra todas las claves primarias dentro de un rango de claves secundarias, considerando que los duplicados pueden estar distribuidos en hojas consecutivas. Retrocede para encontrar la primera ocurrencia real del rango.

IV. Análisis comparativo teórico

IV-A. Complejidad de Operaciones

Las siguientes tablas presentan la complejidad temporal en términos de accesos a memoria secundaria para cada técnica de indexación implementada.

IV-A1. Definición de Variables

- n: número total de registros almacenados
- M: factor de bloque o número de entradas por nodo
- k: tamaño del archivo auxiliar (Sequential File)
- K: número máximo de buckets de overflow (Extendible Hashing)
- r: número de resultados en búsqueda por rango
- h: profundidad global del directorio (Extendible Hashing)

IV-A2. Inserción

Cuadro I: Complejidad de inserción por técnica

Técnica	Caso Promedio	Peor Caso
Sequential File	$O(1)$	$O(n \log n)$ (rebuild)
ISAM	$O(\log n)$	$O(\log n + K)$ (overflow)
B+Tree Clustered	$O(\log_M n)$	$O(M \log_M n)$ (split)
Extendible Hashing	$O(1)$	$O(2^h)$ (directory doubling)
B+Tree Unclustered	$O(\log_M n)$	$O(M \log_M n)$ (split)
R-Tree	$O(\log_M n)$	$O(M \log_M n)$ (split)

IV-A3. Búsqueda Exacta

Cuadro II: Complejidad de búsqueda exacta por técnica

Técnica	Caso Promedio	Peor Caso
Sequential File	$O(\log n + k)$	$O(n)$
ISAM	$O(\log n)$	$O(\log n + K)$ (overflow)
B+Tree Clustered	$O(\log_M n)$	$O(\log_M n)$
Extendible Hashing	$O(1)$	$O(K)$ (overflow)
B+Tree Unclustered	$O(\log_M n)$	$O(\log_M n)$
R-Tree	$O(\log_M n)$	$O(n)$ (overlap)

IV-A4. Búsqueda por Rango

Cuadro III: Complejidad de búsqueda por rango por técnica

Técnica	Caso Promedio	Peor Caso
Sequential File	$O(\log n + r + k)$	$O(n)$
ISAM	$O(\log n + r)$	$O(\log n + r + K)$
B+Tree Clustered	$O(\log_M n + r)$	$O(\log_M n + r)$
Extendible Hashing	—	—
B+Tree Unclustered	$O(\log_M n + r)$	$O(\log_M n + r)$
R-Tree	$O(\log_M n + r)$	$O(n)$ (overlap)

IV-A5. Eliminación

Cuadro IV: Complejidad de eliminación por técnica

Técnica	Caso Promedio	Peor Caso
Sequential File	$O(\log n + k)$	$O(n \log n)$ (rebuild)
ISAM	$O(\log n)$	$O(\log n + K)$ (overflow)
B+Tree Clustered	$O(\log_M n)$	$O(M \log_M n)$ (merge)
Extendible Hashing	$O(1)$	$O(K)$ (overflow)
B+Tree Unclustered	$O(\log_M n)$	$O(M \log_M n)$ (merge)
R-Tree	$O(\log_M n)$	$O(M \log_M n)$ (reinsert)

IV-B. Ventajas, Limitaciones y Casos de Uso

Cuadro V: Comparativa de ventajas, limitaciones y casos de uso recomendados

Técnica	Ventajas	Limitaciones	Casos de Uso
Sequential File	Inserción $O(1)$; rangos eficientes en main ordenado	Rebuild $O(n \log n)$ periódico; aux degrada búsquedas	Datos cronológicos, cargas batch, read-heavy
ISAM	Sin rebalanceo; predecible; simple	Overflow degrada rendimiento; no auto-ajustable	Datos estáticos, archivos de consulta
B+Tree Clustered	Auto-balanceo; $O(\log_M n)$ garantizado; óptimo para rangos	Overhead en splits/merges	Índice primario, transacciones frecuentes
Extendible Hashing	$O(1)$ en búsqueda; crece dinámicamente	No soporta rangos; directory doubling ocasional	Búsqueda exacta por clave
B+Tree Unclustered	Rangos en campos no-clave; múltiples índices	Doble acceso disco (índice + primario)	Índices secundarios con rangos
R-Tree	Consultas espaciales eficientes; KNN y radio	Overlap degrada rendimiento	Datos geoespaciales, proximidad

V. Parser SQL

V-A. Descripción general

El parser SQL interpreta las sentencias del lenguaje y las convierte en estructuras internas que el sistema puede ejecutar. Su función es transformar texto en objetos llamados *planes lógicos*, que representan operaciones como creación de tablas, inserciones o consultas. Se implementó con la librería `Lark`, utilizando una gramática LALR definida manualmente.

V-B. Diseño de la gramática

La gramática fue escrita desde cero en formato `.lark`. En ella se definió un símbolo inicial que permite reconocer múltiples sentencias separadas por ";". Cada sentencia se representa como una regla independiente (`create_table`, `insert_stmt`, `select_stmt`, etc.), lo que facilita su lectura y mantenimiento. El lenguaje soportado incluye:

- `CREATE TABLE`, `LOAD DATA FROM FILE`, `CREATE INDEX`, `DROP TABLE`, `DROP INDEX`, `SELECT`, `INSERT` y `DELETE`.
- Tipos de datos: `INT`, `FLOAT`, `DATE`, `VARCHAR[n]` y `ARRAY[FLOAT]` (2D o n-D).
- Índices: `SEQUENTIAL`, `ISAM`, `BTREE`, `RTREE` y `HASH`.
- Condiciones en `WHERE`: igualdad (=), rango (`BETWEEN`), búsqueda espacial por radio (`IN((x,y),r)`) y búsqueda por vecinos más cercanos (`NEAREST((x,y),k)`).

Durante el análisis, los tokens (números, cadenas, identificadores y puntos espaciales) se normalizan en memoria, garantizando que cada valor llegue correctamente tipado al transformador.

V-C. Transformación a planes lógicos

Una vez generado el árbol de sintaxis, se aplica un transformador que recorre los nodos y los convierte en objetos de Python definidos en `plan_types.py`. Cada tipo de operación SQL tiene su propia estructura, lo que permite mantener independencia entre la interpretación y la ejecución. Por ejemplo:

- **CreateTablePlan**: guarda el nombre de la tabla y las columnas, incluyendo tipo de dato, `KEY` y tipo de índice (si existe).
- **LoadDataPlan**: contiene la ruta del archivo a cargar, el nombre de la tabla y un mapeo opcional para asociar columnas o generar arreglos.
- **SelectPlan**, **InsertPlan** y **DeletePlan**: describen consultas, inserciones y eliminaciones con sus respectivas condiciones o valores.
- **CreateIndexPlan** y **DropTablePlan**: definen acciones administrativas sobre tablas e índices.

Cada literal se convierte automáticamente a su tipo correspondiente (entero, flotante, cadena o nulo) y las coordenadas espaciales se representan como tuplas. El transformador también gestiona los predicados de `WHERE` mediante clases específicas (`PredicateEq`, `PredicateBetween`, `PredicateInPointRadius` y `PredicateKNN`), que permiten al motor reconocer el tipo exacto de condición sin necesidad de analizar texto.

V-D. Funcionamiento general

El flujo del parser es el siguiente:

1. El usuario ingresa una o varias sentencias SQL.
2. La gramática analiza el texto y construye el árbol de sintaxis correspondiente.
3. El transformador recorre el árbol y crea los objetos de plan adecuados.
4. El resultado final es una lista de planes que pueden ser procesados directamente por la base de datos.

El texto SQL se interpreta solo una vez y, a partir de ahí, el sistema trabaja con estructuras internas ya procesadas, lo que mejora la eficiencia y facilita extender el lenguaje con nuevas sentencias sin alterar su estructura principal.

VI. Interfaz Grafica

VI-1. Editor de Consultas

Es aca donde se escribirán las consultas y serán analizadas por el parserSQL para ser ejecutadas. No permite comentarios, las cadenas string utilizan doble comillas como delimitador. Cuenta con botones para ejecutar las consultas, limpiar la base de datos y limpiar el editor.

Figura 26: Interfaz Gráfica: Pagina principal

VI-2. Documentación

En este apartado, se puede apreciar la sintaxis que utiliza nuestro parserSQL para generar consultas. Es ligeramente diferente a PostgreSQL por lo que es altamente recomendable leerla antes de escribir las consultas.

Figura 27: Documentación: Consultas de Búsqueda

VI-A. *Pruebas de uso*

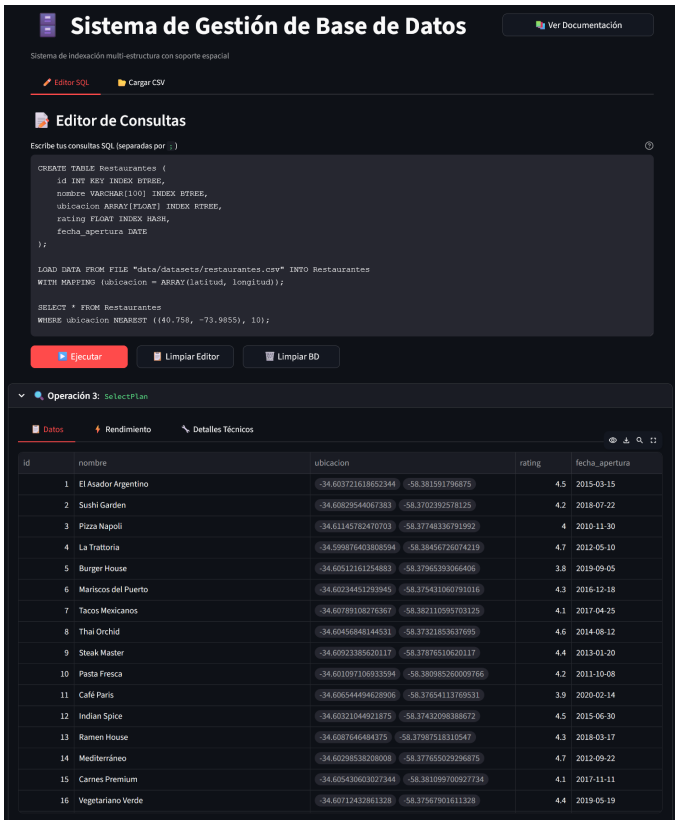

Figura 28: Creacion y carga de datos csv en Tabla Restaurantes

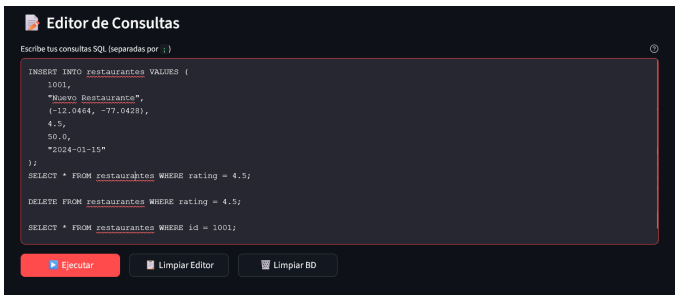

Figura 29: Consultas de Insercion, Busqueda e Eliminacion

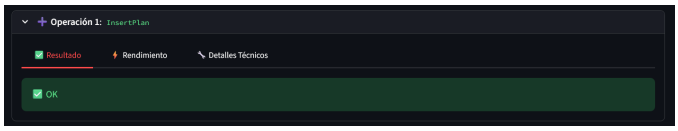

Figura 30: Resultado de Insercion

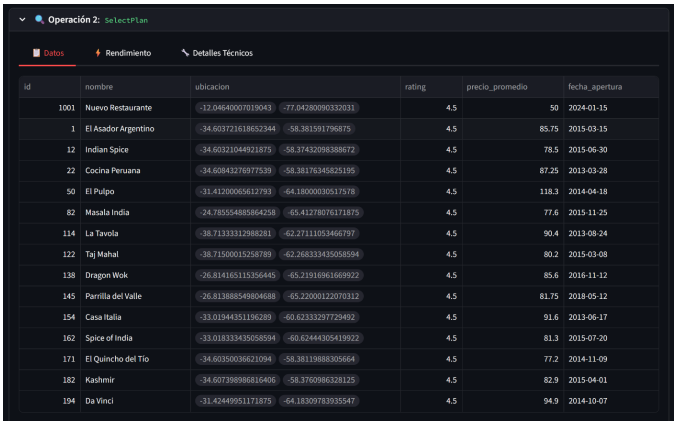

Figura 31: Resultado de Busqueda

Figura 32: Resultado de Eliminacion

VII. Resultados Experimentales

VII-A. *Diseño de los experimentos*

Las pruebas experimentales se realizaron en un entorno Python 3.9 sobre Windows 11, con hardware estándar (Intel Core i7, 16GB RAM, SSD NVMe). Se utilizaron dos datasets: World Cities para comparaciones de índices primarios y secundarios, y NYC Airbnb para índices espaciales.

Se ejecutaron cuatro experimentos comparativos: (1) Índices primarios (Sequential File, ISAM, B+Tree Clustered) en operaciones de inserción, búsqueda exacta y por rango; (2) Índices secundarios para búsqueda exacta (sin índice, Hashing Extensible, B+Tree Unclustered) en el campo ciudad; (3) Índices secundarios para búsqueda por rango (sin índice vs B+Tree Unclustered) en el campo país; y (4) Índices espaciales (R-Tree) para consultas KNN y por radio.

Las métricas evaluadas en cada operación fueron:

- Total de accesos a disco duro (lecturas y escrituras)
- Tiempo de ejecución en milisegundos

Cada experimento se ejecutó con consultas diferentes para obtener promedios representativos.

VII-B. *Resultados obtenidos*

VII-B1. *Experimento 1*

Cuadro VI: Inserción de 41k registros - Índices Primarios

Técnica	Lecturas	Escrituras	Tiempo (ms)
Sequential File	56,571,441	55,617,643	951,162.75
ISAM	250,025	44,946	671,490.27
B+Tree Clustered	125,795	50,638	806,449.86

Cuadro VII: Búsqueda exacta por clave primaria (promedio de 10 búsquedas)

Técnica	Lecturas Promedio	Tiempo Promedio (ms)
Sequential File	16.4	1.98
ISAM	3.0	0.42
B+Tree Clustered	3.0	0.64

Cuadro VIII: Búsqueda por rango en índices primarios (promedio de 5 búsquedas)

Técnica	Lecturas Promedio	Tiempo Promedio (ms)
Sequential File	12,638.2	107.36
ISAM	1,198.0	242.19
B+Tree Clustered	378.2	66.48

VII-B2. Experimento 2

Cuadro IX: Inserción de 41k registros con índices secundarios

Configuración	Lecturas	Escrituras	Tiempo (ms)
Sin Índice Secundario	125,795	50,638	436,264.02
Hashing Extensible	327,410	151,350	836,070.89
B+Tree Unclustered	253,787	102,233	923,343.69

Cuadro X: Búsqueda exacta con índices secundarios (promedio de 20 búsquedas)

Configuración	Lecturas Promedio	Tiempo Promedio (ms)
Sin Índice Secundario	1,228.0	398.27
Hashing Extensible	8.65	0.59
B+Tree Unclustered	8.3	1.19

VII-B3. Experimento 3

Cuadro XI: Inserción de 41k registros con índice secundario para búsqueda por rango

Configuración	Lecturas	Escrituras	Tiempo (ms)
Sin Índice Secundario	125,795	50,638	849,410.68
B+Tree Unclustered	253,670	102,087	1,720,190.18

Cuadro XII: Búsqueda por rango con índices secundarios (promedio de 5 búsquedas)

Configuración	Lecturas Promedio	Tiempo Promedio (ms)
Sin Índice Secundario	1,228.0	187.55
B+Tree Unclustered	2,514.0	129.91

VII-B4. Experimento 4

Cuadro XIII: Inserción de 10k registros con índice espacial R-Tree

Fase	Lecturas	Escrituras	Tiempo (ms)
Carga de Datos	65,107	27,543	228,583.59
Creación de Índice R-Tree	886	21,192	1,915.2
Total de Inserción	**65,993**	**48,735**	**230,498.79**

Cuadro XIV: Búsqueda de k-vecinos más cercanos (k=50) con R-Tree

Técnica	Lecturas	Tiempo (ms)
R-Tree Spatial Index	151	21.91

Cuadro XV: Búsqueda por radio (0.05 grados $\approx$ 5.5 km) con R-Tree

Técnica	Lecturas	Tiempo (ms)
R-Tree Spatial Index	30676	3544.08

VII-C. Discusión y análisis

Los resultados experimentales confirman las predicciones teóricas con matices importantes:

VII-C1. Índices Primarios

- **Sequential File**: Inserción catastrófica por reorganizaciones frecuentes, pero excelente en búsqueda por rango gracias a secuencialidad.
- **ISAM**: Mejor balance en cargas masivas evitando rebalanceos, pero overflow chains degradan consultas por rango.
- **B+Tree Clustered**: Estructura óptima con complejidad logarítmica consistente sin degradación.

VII-C2. Índices Secundarios

- **Búsqueda exacta**: Hashing Extensible logró $O(1)$ real con overhead aceptable. B+Tree Unclustered mostró tiempos ligeramente superiores.
- **Búsqueda por rango**: B+Tree Unclustered justificó su costo. Hashing queda descalificado por incapacidad para rangos.

VII-C3. Índices Espaciales

- **R-Tree**: Eficiencia extraordinaria en KNN, pero overlap de MBRs impacta búsquedas por radio extensas.

VII-D. Aporte de los índices

Los índices transformaron el rendimiento en tres aspectos clave:

- **Reducción de accesos**: Búsqueda exacta mejoró 675× al eliminar escaneos completos.
- **Nuevas capacidades**: Índices espaciales habilitaron consultas geográficas. B+Tree Unclustered permitió rangos en campos secundarios.
- **Escalabilidad**: B+Tree mantuvo $O(\log n)$ consistente. Sequential File e ISAM mostraron deterioro acumulativo.
- **Trade-offs**: Índices secundarios incrementan inserción 2-4×, pero se amortizan con pocas consultas. Óptimos para read-heavy, no para write-heavy.

VIII. Conclusiones

- **No existe una solución perfecta**: Cada técnica brilla en contextos específicos. B+Tree Clustered resultó la más equilibrada para índices primarios, Hashing Extensible dominó en búsquedas exactas, y R-Tree fue esencial para consultas espaciales.

- **Los índices secundarios valen la pena**: Aunque duplican o triplican el tiempo de inserción, una sola búsqueda sin índice puede ser cientos de veces más lenta. En aplicaciones reales con más lecturas que escrituras, son imprescindibles.
- **El mantenimiento automático es crucial**: Sequential File e ISAM se deterioran con el uso por reorganizaciones y overflow. El auto-balanceo del B+Tree mantiene su rendimiento constante, lo que lo hace más confiable a largo plazo.
- **La arquitectura modular simplifica la complejidad**: El `DatabaseManager` permitió coordinar múltiples índices manteniendo la consistencia. El parser SQL propio resultó suficientemente expresivo para operaciones avanzadas.

VIII-A. Mejoras Futuras

Posibles extensiones incluyen optimizar las estrategias de reorganización para reducir rebuilds frecuentes, implementar caché en memoria para datos consultados frecuentemente, agregar índices compuestos para consultas multi-campo, y extender el parser SQL con operaciones JOIN y funciones de agregación como COUNT, AVG y SUM.

VIII-B. Video de presentación

Enlace al video: https://drive.google.com/drive/folders/1d8QkI2vcO49ck5Ipro1T7g6NIEqBobzJ?usp=sharing