Proyecto 1
Organización e Indexación Eficiente de Archivos y Datos Multidimensionales

Alberto Domenic Rincon Espinoza

Mayo 2025

Índice

1. Introducción

- Este proyecto tiene como objetivo implementar, analizar y aplicar diferentes técnicas de indexación tradicionales y espaciales, ademas de compararlas y evaluar su desempeño tanto en eficiencia temporal como espacial (CPU y DISCO).

- La aplicación desarrollada permite gestionar y consultar un dataset geoespacial de calidad del aire que describe niveles de contaminación (**PM2.5**, **Ozono (O_3)**, **Monóxido de Carbono (CO)**) con coordenadas geográficas de ciudades a nivel mundial. Este dataset resulta de la fusión de dos fuentes: una con mediciones de contaminantes por país y otra con datos de localización (latitud/longitud) de ciudades, lo que permite realizar análisis ambientales con precisión espacial.

- Para optimizar el rendimiento en las consultas, se integraron diversas técnicas de indexación:

  - **R-Tree**, para búsquedas espaciales eficientes, como recuperar todas las estaciones dentro de un radio geográfico o encontrar las más cercanas a un punto dado.
  - **ISAM** para consultas por fecha de medición o por rangos de concentración de contaminantes.
  - **Extendible Hashing**, para accesos rápidos por identificador de ciudad o estación.

2. Técnicas Implementadas

2.1. Resumen de Técnicas de Organización e Indexación

- **Sequential File:** Mantiene el archivo de datos ordenado físicamente, esto permite una búsqueda de rango y unitaria eficiente, además no requiere de espacio adicional más que el puntero al siguiente registro lógico. Sin embargo se debe mantener un espacio auxiliar para las inserciones, cuando este espacio se llena se debe reconstruir el archivo de datos ordenado.

- **ISAM-Sparse Index:** A partir de un conjunto de datos se construye un árbol estático que en los nodos internos mantiene las llaves y punteros que indican una relación de comparación (caso comun node(ptr[i-1]) $\leq$ key[i] $\leq$ node(ptr[i])), y en los nodos hoja (archivo de datos) se mantienen bloques con registros referenciados por el indice. Este enfoque permite una búsqueda por rango y unitaria eficiente y no requiere de reconstrucción, sin embargo requiere de espacio adicional fijo y también puede tener bloques de overflow lo cual degrada en cierta medida las operaciones de búsqueda.

- **Extendible Hashing:** Mantiene un archivo directorio el cual tiene 2^G entradas, donde G es la profundidad global, este directorio apunta a buckets los cuales contienen registros. Cada entry del directorio apunta a una cantidad de buckets que depende de la profundidad local, para acceder a un bucket se tiene que aplicar una funcion hash a una key y extraer los G bits mas a la derecha. Cuando se inserta y no hay espacio en el

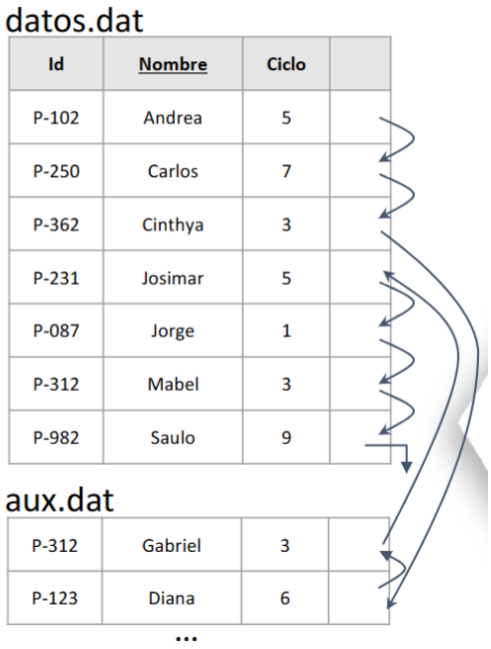

Figura 1: Sequential File con espacio auxiliar

bucket se realiza la operación de split el cual aumenta la profundidad local del bucket, si la profundidad local supera a la global se tiene que aumentar en uno la profundidad global y reacomodar el directorio. Este indice permite una búsqueda unitaria eficiente y además su complejidad temporal no se ve comprometida por el crecimiento en la cantidad de registros. Sin embargo, este indice no permite búsquedas por rango eficientes y el espacio adicional (directorio) crece a medida que la cantidad de datos aumenta.

- **R-Tree:** Es una estructura de árbol balanceado diseñada para datos multidimensionales. Agrupa objetos cercanos espacialmente en rectángulos mínimos (MBR) y los organiza jerárquicamente. Cada nodo contiene un conjunto de MBRs que pueden solaparse, junto con punteros a hijos (nodos internos) o registros (nodos hoja). Soporta eficientemente búsquedas por rango y espaciales. Su inserción puede requerir divisiones y fusiones de nodos para mantener el árbol balanceado.

2.2. Algoritmos Implementados

2.2.1. Sequential File

- Inserción: Se hace un recorrido secuencial siguiendo el orden del puntero next, si hay espacio en el espacio ordenado se inserta, sino, se inserta en el espacio auxiliar, si no hay espacio en el espacio auxiliar se reconstruye el file en orden y se inserta el registro en el espacio auxiliar.

- Búsqueda unitaria: Como el espacio principal esta ordenado físicamente se realiza una búsqueda binaria, luego, si no se encuentra se busca en el espacio auxiliar.

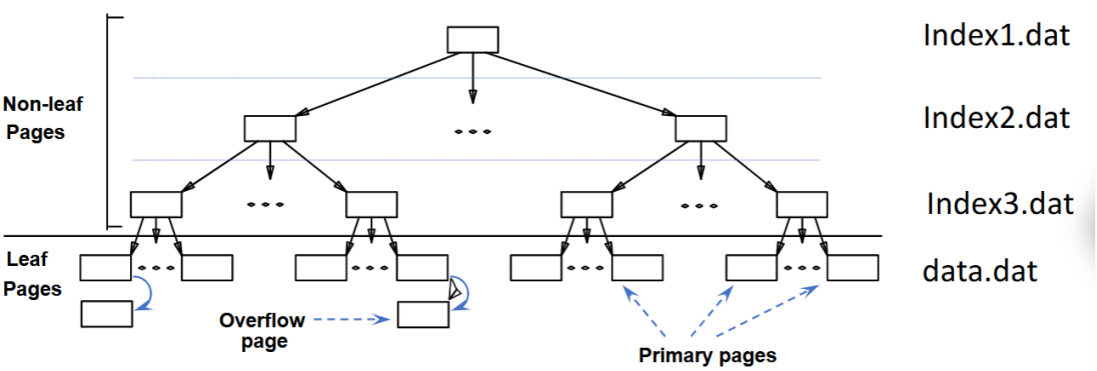

Figura 2: Indice estatico ISAM

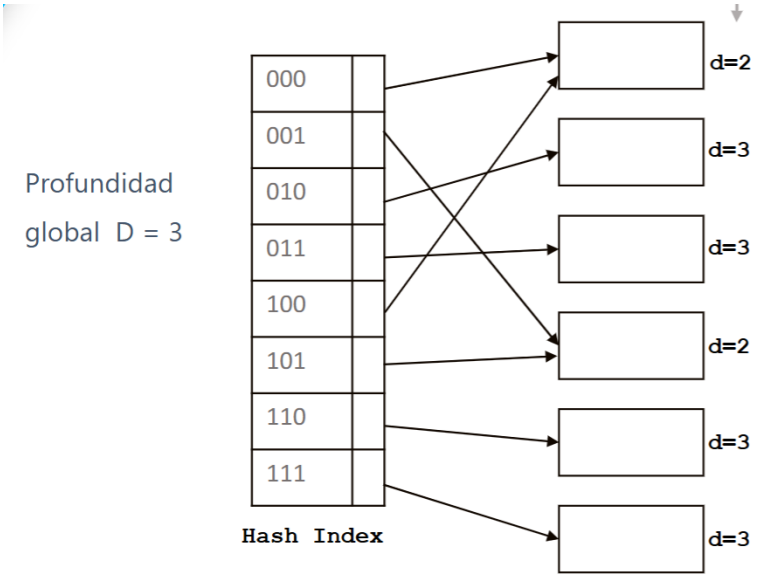

Figura 3: Extendible hashing index

- Búsqueda por rango: Se realiza una búsqueda unitaria sobre begin_key luego se recorre con el puntero next hasta encontrar end_key.

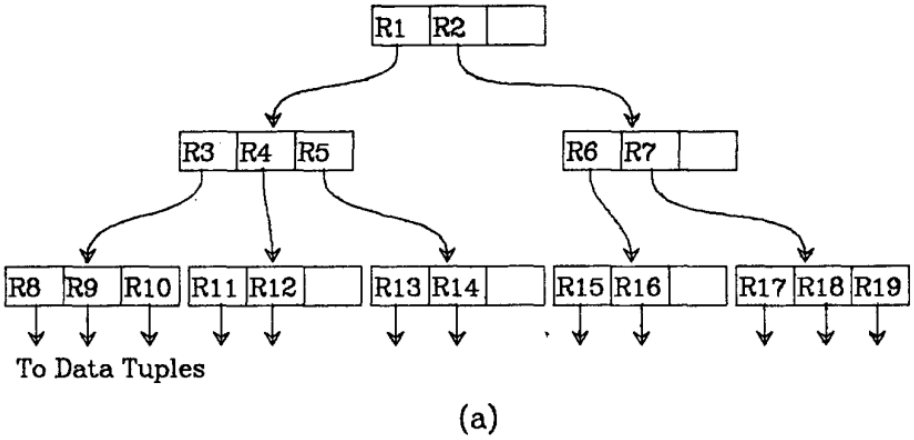

Figura 4: R Tree index

2.2.2. ISAM-Sparse Index

- Inserción: Se busca en el indice raiz y el indice intermedio a que bloque de indexación pertenece el registro a insertar, luego se añade el registro al bloque, si no hay espacio se crea un bloque de desbordamiento y se enlaza.
- Búsqueda unitaria : primero se compara la key en el nodo raíz para saber a que bloque del siguiente nivel de indexación pertenece, luego se busca a que bloque de datos pertenece la key y se busca en el bloque y si tiene overflow, también buscar en el bloque desbordado.
- Búsqueda por rango : se realiza una búsqueda unitaria de begin_key para obtener el bloque mínimo y otra búsqueda unitaria de end_key para obtener el bloque máximo, luego se recorre desde el bloque mínimo hasta el bloque máximo insertando rn una lista temporal todos los registros que esten dentro del rango.

2.2.3. Extendible Hashing

- Inserción : Se calcula el hash de la key, luego se extraen los G(Global depth) bits mas a la derecha, se ubica en el directorio el bucket perteneciente al hash. Si el bucket esta lleno se realiza la operación de split aumentando L(Local depth) en uno e insertar el registro en el bucket correspondiente, si L es mayor que G entonces se debe aumentar G en uno y re acomodar el directorio.
- Búsqueda unitaria : Se calcula el hash y se ubica el bucket en el directorio, y se busca el registro en el bucket.

2.2.4. R-Tree

- Inserción: Se realiza una búsqueda descendente desde la raíz para encontrar la hoja más adecuada según el criterio de mínima expansión del MBR. Si el nodo hoja tiene

espacio, se inserta el nuevo MBR; si está lleno, se realiza un split del nodo utilizando heurísticas como el algoritmo de Greene o Quadratic Split, y se propaga el cambio hacia arriba si es necesario.

- Búsqueda KNN: Se utiliza una cola de prioridad (heap) para explorar primero los nodos más cercanos al punto de consulta. Se calcula la distancia mínima entre el punto y los MBRs, y se visitan los nodos en orden creciente de esa distancia hasta encontrar los k vecinos más cercanos.
- Búsqueda por rango: Se realiza un recorrido recursivo desde la raíz visitando todos los nodos cuyos MBRs intersectan el área de consulta. En los nodos hoja, se evalúa si cada objeto intersecta con el rango y se agregan a los resultados los que cumplen la condición.

2.3. Análisis Comparativo Teórico

2.3.1. Sequential File

- Inserción : Escaneo secuencial $O(n + k)$, insersion $O(1)$, complejidad final $O(n + k)$
- Búsqueda unitaria: Búsqueda binaria $O(lg(n))$, búsqueda en el espacio auxiliar $O(lg(n))$, ya que $k = lg(n)$, complejidad final $O(lg(n))$.
- Búsqueda por rango: Búsqueda binaria de begin_key $O(lg(n))$, recolección de registros entre el rango $O(k)$ donde k es la cantidad de registros en el rango. Complejidad final $O(lg(n) + k)$.

2.3.2. ISAM-Sparse Index

- Inserción: Acceso a índice raíz e intermedio $O(\log_b(n))$ (donde b es el branching factor), acceso secuencial al bloque de datos $O(1)$, pero si hay desbordamiento puede implicar acceso en cadena de bloques $O(t)$ (donde t es la cantidad de bloques desbordados). Complejidad final promedio $O(\log_b(n))$, peor caso $O(\log_b(n) + t)$.
- Búsqueda unitaria: Acceso jerárquico a través de índices $O(\log_b(n))$, búsqueda secuencial en el bloque de datos $O(1)$, y búsqueda en cadena de desbordamiento $O(t)$. Complejidad final promedio $O(\log_b(n))$, peor caso $O(\log_b(n) + t)$.
- Búsqueda por rango: Dos búsquedas unitarias $O(\log_b(n))$, y recorrido de bloques de datos (posiblemente incluyendo desbordamientos) $O(k)$ donde k es el número de registros dentro del rango. Complejidad final $O(\log_b(n) + k)$.

2.3.3. Extendible Hashing

- Inserción: Cálculo de hash $O(1)$, acceso a bucket $O(1)$, en caso de split (cuando bucket está lleno) se realiza redistribución local $O(m)$, donde m es la cantidad de elementos en el bucket, y posiblemente duplicación del directorio si $L > G$, lo cual puede implicar tiempo adicional $O(2^G)$. Complejidad promedio $O(1)$, peor caso $O(2^G)$ durante redistribuciones.

- Búsqueda unitaria: Cálculo de hash y acceso directo al bucket $O(1)$, búsqueda lineal dentro del bucket $O(1)$ (si el tamaño del bucket es pequeño y constante). Complejidad final $O(1)$.

2.3.4. R-Tree

- Inserción: Recorrido desde raíz hasta hoja para encontrar el nodo óptimo $O(\log_M(n))$ (siendo M el número máximo de hijos por nodo), posible propagación por splits hasta la raíz. En el peor caso (cuando hay splits hasta la raíz), la complejidad es $O(\log_M(n))$. Complejidad promedio $O(\log_M(n))$.

- Búsqueda KNN: Búsqueda heurística con heap, visitando los nodos según la mínima distancia estimada. Complejidad en promedio $O(\log_M(n) + k\log(n))$ donde k es la cantidad de vecinos requeridos.

- Búsqueda por rango: Visita a los nodos cuyos MBR intersectan con el rango, coste depende del número de nodos visitados. Complejidad promedio $O(\log_M(n) + k)$, peor caso $O(n)$ si todos los nodos intersectan con el rango.

2.4. Parser SQL

2.4.1. Gramática utilizada

programa ::= sentencia

sentencia ::= `CREATE TABLE` id `(` def_columnas `)`
| `CREATE TABLE` id `FROM FILE` cadena `USING INDEX` id `(` cadena `)`
| `SELECT` columns `FROM` id condicion_where
| `INSERT INTO` id `VALUES (` valores `)`
| `DELETE FROM` id `WHERE` condicion

def_columnas ::= columna | columna `,` def_columnas

columna ::= id tipo opciones_col

tipo ::= `INT` | `VARCHAR[`num`]` | `DATE` | `TEXT` | `FLOAT` | `ARRAY[ FLOAT ]`

opciones_col ::= ε | `KEY` | `KEY INDEX` tipo_indice | `INDEX` tipo_indice

tipo_indice ::= `SEQ` | `BTREE` | `RTREE` | `AVL` | `ISAM` | `HASH`

columns ::= `*` | id | id `,` columns

condicion_where ::= ε | `WHERE` condicion

condicion ::= id `=` valor
| id `BETWEEN` valor `AND` valor
| id `IN (` valor `,` valor `)`

valor ::= num | floatval | cadena

valores ::= valor | valor `,` valores

3. Resultados Experimentales

3.1. Descripción del Dataset

Nombre del Campo	Descripción	Tipo de Dato
Country	Nombre del país	VARCHAR(50)
City	Nombre de la ciudad	VARCHAR(50)
AQI Value	Valor general del Índice de Calidad del Aire	INT
AQI Category	Categoría del índice general (ej. Good, Moderate)	VARCHAR(20)
CO AQI Value	Valor del AQI basado en monóxido de carbono (CO)	INT
CO AQI Category	Categoría del AQI para CO	VARCHAR(20)
Ozone AQI Value	Valor del AQI basado en ozono	INT
Ozone AQI Category	Categoría del AQI para ozono	VARCHAR(20)
NO2 AQI Value	Valor del AQI basado en dióxido de nitrógeno (NO)	INT
NO2 AQI Category	Categoría del AQI para NO	VARCHAR(20)
PM2.5 AQI Value	Valor del AQI basado en partículas finas PM2.5	INT
PM2.5 AQI Category	Categoría del AQI para PM2.5	VARCHAR(20)
lat	Latitud geográfica del lugar	FLOAT
lng	Longitud geográfica del lugar	FLOAT

Cuadro 1: Diccionario de datos del conjunto de calidad del aire

3.2. Comparación de Desempeño

- Métricas: accesos a disco, tiempo de ejecución.
- Técnicas comparadas.

Figura 5: Comparación de rendimiento entre técnicas de indexación

3.3. Análisis de Resultados

Interpretación de las métricas obtenidas y conclusiones preliminares.

4. Pruebas de Uso y Presentación

- Interfaz gráfica desarrollada.
- Casos de uso presentados en la demo.
- Evidencia de cómo los índices mejoran el rendimiento.

Figura 6: Captura de pantalla de la interfaz gráfica

5. Conclusiones

- Reflexiones sobre el trabajo realizado.
- Limitaciones encontradas.
- Posibles mejoras o trabajos futuros.

Repositorio y Video

- **Repositorio GitHub:** `https://github.com/usuario/proyecto1`
- **Video de presentación:** `https://youtube.com/enlace_video`