Universidad de Ingeniería y Tecnología (UTEC)

Facultad de Ciencias de la Computación

Proyecto 1

Organización e Indexación Eficiente de Archivos con Datos Tabulares y Espaciales

Curso: Base de Datos II (CS2702)
Profesor: PhD. Heider Ysaias Sánchez Enríquez
Ciclo: 2025-I

Integrantes: Badi Masud Rodriguez Ramirez / 202310299
Santiago Miguel Silva Reyes / 202310266
Edson Gustavo Guardamino Felipe / 202010136
Alair Jairo Catacora Tupa /
Mateo Elias Ramirez Chuquimarca / 202310082

21 de octubre de 2025

1. Introducción

El proyecto propone el diseño e implementación de un **sistema de base de datos multimodal** capaz de indexar y consultar datos estructurados y no estructurados. Nuestra solución presenta una API backend con procesamiento de consultas, subsistema de indexación y persistencia en disco; y aplicaciones *front-end* para ejecutar pruebas y revisar resultados. El sistema está pensado para manejar distintos tipos de datos según uso (texto, imágenes, audio, video y tablas), aunque en esta entrega se trabaja únicamente con datos *tabulares* (CSV) y *espaciales* (coordenadas y regiones).

Para este proyecto, implementamos y comparamos las siguientes técnicas de indexación: Sequential/AVL, ISAM, B+ Tree, Extendible Hashing y R-Tree. El motor registra #lecturas, #escrituras y tiempo de ejecución (ms) para medir el costo en memoria secundaria en escenarios de *insert*, *search* (hit/miss), *range* y *remove*.

Aportes del proyecto:

- Motor en disco con *parser* SQL-like, *planner/executor* y *catalog*.
- Implementación de índices primarios y secundarios con reglas de inserción, búsqueda, rango y eliminación.
- Adaptador de R-Tree para datos espaciales y consultas de rango/kNN (según implementación).
- API HTTP y una interfaz mínima para ejecutar consultas y visualizar resultados.
- Metodología experimental para comparar técnicas usando contadores de I/O y tiempo.

2. Objetivos

Objetivo general

Diseñar, implementar e integrar las estructuras de datos vistas en clase para evaluar su desempeño en distintos contextos de uso (tipo de dato y patrón de acceso). El propósito no es declarar una estructura "mejor" en términos absolutos, sino **mostrar en qué escenarios cada una resulta adecuada** y cuáles son sus límites.

Objetivos específicos

- Implementar *Sequential/AVL*, *ISAM*, *B+ Tree*, *Extendible Hashing* y *R-Tree* con sus operaciones básicas (insert, search, range, remove; y range/kNN para R-Tree).
- Integrar los índices al motor (parser SQL-like, planner/executor y catálogo) y persistir en disco.
- Definir conjuntos de datos *tabulares* y *espaciales* y escenarios de prueba (hit/miss, rango por ventanas, cargas por lotes).
- Medir **costo en memoria secundaria** (#lecturas/#escrituras) y **latencia** (ms); verificar **correctitud** de resultados (y *margen de error* cuando aplique, p. ej., heurísticas espaciales/kNN).

- Comparar resultados y **caracterizar los *trade-offs*** (rendimiento, soporte de rango, manejo de duplicados/overflow, tamaño en disco).
- Elaborar una **guía de elección** de índice según patrón de acceso y tipo de dato.
- Entregar una API reproducible (Docker) y scripts de experimentación para replicar métricas.

3. Arquitectura y Repositorio

3.1. Despliegue con Docker Compose

El **backend** es Python y persiste sus estructuras en `./out/`. No requiere un motor externo de BD; solo filesystem. El índice espacial usa `rtree` (requiere *libspatialindex*). Para que los archivos de índices sobrevivan a reinicios, se mapea `./out` como volumen. *Requisitos mínimos*: Python 3.10+, `pip install rtree`. En Docker, instalar `libspatialindex` antes de `pip install rtree`.

Listing 1: compose.yml (mínimo viable)

```
services:
  backend:
    build: ./backend
    ports: ["8000:8000"]
    volumes:
      - ./backend/out:/app/out
      - ./backend/data:/app/data
    environment:
      - PYTHONUNBUFFERED=1
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 10s
      timeout: 3s
      retries: 3
  frontend:
    build: ./frontend
    ports: ["5173:5173"]
    depends_on: [backend]
```

Ejecutar: `docker compose up -build -d`. **Persistencia**: los índices se escriben en `./backend/out`, incluyendo archivos `.dat`, `.idx` y el directorio `rtree_index`.

3.2. Estructura del Repositorio

Módulos de índices (backend/src/index/):

- `avl.py`: índice AVL *on-disk*. Define formato binario para **datos** (`AVLDAT01`) y **índice** (`AVLIDX01`). Cada nodo almacena `(key, height, left_off, right_off, value_off)` y se rebalancea con rotaciones. Expone `add`, `search` (punto), `rangeSearch` y `io_stats` para lecturas/escrituras.

- `bptree.py`: **B+ Tree** con *bulk-load*; serializa el árbol en `.idx` (pickle) y los registros en `.dat`. Provee `search`, `range_search`, `insert` (ordena y reconstruye) y `remove` (borra por clave). Clase de alto nivel: `ClusteredIndexFile`.
- `ext_hash.py`: **Extendible Hashing** con *buckets* de tamaño fijo (`BLOCK_FACTOR`) y encadenamiento limitado (`MAX_CHAINING`); al excederlo, hace *rehash* global incrementando `D` (profundidad) y reubicando registros. Operaciones: `insert`, `search`, `remove`. Persistencia en `out/<nombre>.dat`.
- `rtree_adapter.py`: **R-Tree** sobre `rtree` (en disco). Implementa `add`/`remove`, `rangeSearch` (filtro circular con distancia euclídea tras intersección de MBR) y `kNN`. Guarda archivos en `out/rtree_index`.

3.3. Endpoints principales

El backend del gestor de base de datos fue implementado con `FastAPI`, ofreciendo una serie de endpoints REST que permiten ejecutar consultas, cargar datos y realizar operaciones espaciales. Todos los endpoints retornan tiempos de ejecución (`_elapsed_ms`) y métricas internas de I/O (`read`, `write`) mediante `reset_counters()` y `get_counters()`.

- `GET /health`: Verifica el estado del servicio (utilizado por Docker para *healthcheck*).
- `POST /api/sql`: Ejecuta una sentencia SQL interpretada por el motor. Ejemplo: `SELECT * FROM clientes WHERE id = 10;` Incluye medición de tiempo y métricas de acceso a disco/memoria.
- `POST /upload-csv`: Permite subir archivos CSV al servidor (guardados en el directorio `/data`).
- `POST /load-csv`: Crea una tabla e indexa un CSV usando la técnica especificada: `SEQUENTIAL | ISAM | EXTHASH | BPTREE | RTREE`. Ejemplo de operación interna: `CREATE TABLE clientes FROM FILE "data.csvÜSING INDEX BPTREE(ïd")`
- `GET /tables`: Devuelve un listado de las tablas cargadas en memoria, junto con su clave, tipo de índice y columnas.
- `POST /spatial/range`: Ejecuta una consulta espacial de rango sobre un índice `R-Tree`. Parámetros: `table`, `point [x,y]`, `radius`. Ejemplo: buscar todos los puntos dentro de un radio de 10 metros.
- `POST /spatial/knn`: Realiza una búsqueda de los `k` vecinos más cercanos (*kNN*) a un punto dado. Parámetros: `table`, `point [x,y]`, `k`. Ejemplo: retornar los 5 puntos más cercanos a una coordenada.

Cada endpoint mide su tiempo de ejecución con `time.perf_counter()` y adjunta las métricas obtenidas desde el módulo `io_counters`, lo cual permitió generar los resultados experimentales presentados en las gráficas comparativas.

4. Técnicas de Indexación y Operaciones

4.1. Índices tradicionales (Sequential/AVL, ISAM, B+ Tree, Extendible Hashing)

Sequential/AVL. Modelo *heap + índice* sobre archivo binario. El AVL mantiene claves y offsets al heap, rebalancea con rotaciones y soporta *search* (devuelve duplicados) y *range*. Ventajas: rango ordenado y duplicados correctos. Costes: mantenimiento de árbol y ausencia de borrado físico (se propone *lazy delete*).

ISAM. Estructura estática basada en hojas ordenadas con *overflow areas*. Útil cuando las inserciones reales se dan por lotes; coste de búsqueda cercano a $O(\log_B N)$. En nuestro proyecto se usa como referencia teórica y para el *parser* SQL; la implementación práctica se centra en AVL/B+ Tree/Hash.

B+ Tree (fanout 256). Archivo `.dat` ordenado por clave (*clustered*) y árbol B+ serializado. Las hojas están encadenadas, lo que hace eficiente *range* y *order by*. Inserción/borrado reescriben el archivo y reconstruyen el índice (diseño didáctico), por lo que las actualizaciones son más costosas que las lecturas.

Extendible Hashing (BF=64, chain=2). Igualdad con latencia estable. Directorio en disco + *buckets* de tamaño fijo; encadenamiento limitado. Si se supera, se dispara un *rehash* global que incrementa la profundidad y redistribuye. No soporta rango; ajustar `BLOCK_FACTOR` y `MAX_CHAINING` según densidad de claves.

4.2. R-Tree (datos espaciales)

Índice espacial en disco con MBRs. Insertamos puntos como MBR degenerados; *rangeSearch* combina filtro por intersección de MBR con verificación por distancia euclídea (ordena por `dist`). *kNN* usa el operador `nearest`. El rendimiento depende de solapamiento y distribución: menor solapamiento ⇒ mejor selectividad.

4.3. Parser SQL

El *parser* implementa un analizador léxico–sintáctico sencillo que: (i) normaliza la consulta (elimina comentarios `-` y `/* */` y espacios extra), (ii) tokeniza cadenas, números, identificadores, delimitadores y operadores, y (iii) construye un AST minimal (`CreateTableStatement`, `SelectStatement`, `InsertStatement`, `DeleteStatement`) con validación básica. Los errores se reportan vía `SQLParserError` con mensajes explícitos.

Soporte de sentencias

- `CREATE TABLE ...`: define columnas con tipos `INT, FLOAT, VARCHAR[n], DATE, ARRAY[T]`. Permite marcar `KEY` e indicar el tipo de índice con `INDEX <tipo>`, aceptando sinónimos: `SEQ`, `AVL`, `ISAM`, `BTree/B+Tree`, `BPTreeClustered`, `ExtHash`, `RTree`.
- `CREATE TABLE ... FROM FILE "..." USING INDEX <tipo>(col):` carga un CSV y genera el par de archivos (`.dat` y el índice físico). La validación exige `USING INDEX` cuando se usa `FROM FILE`.

- `INSERT INTO T VALUES (...)`: inserción posicional, sin lista de columnas; los valores admiten literales, listas `[]` para `ARRAY` y `VARCHAR` entre comillas.
- `DELETE FROM T WHERE ...`: sólo con `WHERE` válido (igualdad o rango); de lo contrario se rechaza.
- `SELECT cols FROM T [WHERE ...]`: `cols` puede ser `*` o una lista de nombres.

Cláusulas `WHERE` y plan de ejecución

- `col = c` (igualdad): si `col` es clave con `EXTENDIBLE_HASH` o árbol (`AVL/B+`), se ejecuta búsqueda exacta; de lo contrario, *secuencial scan* como *fallback*.
- `col BETWEEN a AND b` (rango): privilegia `AVL/B+Tree` (ordenadas) o `ISAM` (paginado con encadenamiento); si no hay índice adecuado, *scan* filtrando.
- `ubicacion IN (POINT, [x,y(,z)(,r)])` (rango espacial): traduce a consulta por `MBR` en `R-Tree`; si `r` está presente, se usa rango radial.
- `ubicacion IN (k, [x,y(,z)])` (k-NN): compila a búsqueda de vecinos más cercanos en `R-Tree` con pruning geométrico.

Detalles de implementación

- **Tokenización:** reconoce literales con comillas simples o dobles, enteros/`float`, paréntesis, corchetes, comas y operadores relacionales.
- **Tipos e índices tolerantes:** el mapeo de `INDEX` acepta variantes (`BPTREE`, `B+TREE`, `EXT_HASH`, `RTREE_INDEX`, etc.) para robustez.
- **Estructuras de salida:** cada sentencia se materializa en una `@dataclass` con los campos estrictamente necesarios para el planificador/ejecutor.

Ejemplos

- `CREATE TABLE Inventario3D( id INT KEY INDEX BPTreeClustered, x FLOAT INDEX RTree, y FLOAT, z FLOAT, nombre VARCHAR[32] )`
- `CREATE TABLE Inventario3D FROM FILE "data.csvÜSING INDEX ExtHash(id)`
- `SELECT * FROM Inventario3D WHERE id = 123` (*hash/árbol* → lookup exacto)
- `SELECT * FROM Inventario3D WHERE nombre BETWEEN 'A' AND 'Z'` (*árbol/ISAM* → rango)
- `SELECT * FROM Inventario3D WHERE x IN (POINT, [10,20,3])` (*R-Tree* → rango 2D con radio)
- `SELECT * FROM Inventario3D WHERE x IN (5, [10,20,5])` (*R-Tree* → 5-NN en 3D)

5. Resumen Técnico de Implementación

5.1. AVL on-disk (índice + heap de datos)

Layout y persistencia. Archivo de *datos* binario con cabecera (`AVLDAT01`, versión y tamaño de registro) y archivo de *índice* con cabecera (`AVLIDX01`, raíz, conteo, tamaño de nodo). Cada nodo almacena `(key, height, left_off, right_off, value_off)`. Los registros viven en el heap de datos y el índice guarda el desplazamiento (*offset*) a cada registro.
Operaciones. *insert* con rebalance (rotaciones IZQ/DER); *search* puntual que retorna *todas* las coincidencias (explora ambos hijos cuando `key == k`); *rangeSearch* $[l, r]$ por recorrido *in-order*.
I/O y métricas. Contadores de lecturas/escrituras separados para datos e índice (`io_stats`). Coste típico: $\mathcal{O}(\log N + k)$ para *range*. Borrado físico no implementado (se puede documentar como trabajo futuro).

5.2. B+ Tree (clustered file)

Layout y persistencia. Construcción a partir de CSV: ordena por clave, vuelca archivo `.dat` y realiza *bulk-load* del árbol (índice serializado en `.idx`). Hojas encadenadas para rango; fanouts altos en nodos internos/hojas.
Operaciones. *search* y *range_search* por offsets en el `.dat`. *insert*/*remove* reescriben el archivo ordenado y luego reconstruyen el árbol ($\mathcal{O}(N)$ por actualización), coherente con un *clustered file* didáctico.
I/O y métricas. *range* muy eficiente por acceso secuencial; mantenimiento caro al actualizar. Se recomienda declararlo explícitamente en conclusiones.

5.3. Extendible Hashing (bucket file)

Layout y persistencia. Archivo comienza con profundidad global D (32 bits). Directorio lineal y *buckets* de tamaño fijo (`BLOCK_FACTOR`) con puntero a overflow. Encadenamiento limitado (`MAX_CHAINING`); al excederse, se realiza *rehash* global (incrementa D y redistribuye).
Operaciones. *insert*, *search*, *remove*. Búsqueda recorre bucket base y su cadena. Posible costo elevado si hay rehashes frecuentes (parámetros pequeños = comportamiento adverso en datasets medianos).
I/O y métricas. Igualdad promedio $\mathcal{O}(1 + \alpha)$ con cadenas cortas; *range* no soportado. Registrar en resultados el impacto de *rehash*.

5.4. R-Tree (adapter sobre `rtree`)

Layout y persistencia. Índice en disco bajo `out/rtree_index`. Inserta puntos como MBR degenerados $[x, x] \times [y, y]$.
Operaciones. *add*/*remove*; *rangeSearch* con filtro de intersección de MBR seguido de verificación por distancia euclídea (ordena por `dist`); *kNN* mediante `nearest`.
I/O y métricas. Coste y selectividad dependen del solapamiento de MBRs y la distribución. Reportar #lecturas/escrituras y tiempo frente a radios/porcentajes de cobertura.

5.5. Artefactos generados en `out/`

- **AVL**: `<tabla>.dat` (heap de datos), `<tabla>.idx` (árbol AVL).
- **B+ Tree**: `<tabla>.dat` (ordenado por clave), `<tabla>.idx` (índice serializado).
- **Extendible Hashing**: `<tabla>.dat` (directorio + buckets).
- **R-Tree**: directorio `rtree_index/` con archivos del índice.

5.6. Síntesis de soporte por índice

Índice	Ins.	Eq.	Rango	Rem.	kNN	Pers.	Observaciones
Sequential/AVL	Sí	Sí	Sí	No	n/a	`.dat` + `.idx`	Devuelve duplicados; no hay borrado físico (proponer *lazy delete*).
B+ Tree	Sí	Sí	Sí	Sí	n/a	`.dat` + `.idx`	*Range* eficiente; *insert/remove* reescriben y reconstruyen (clustered file).
Extendible Hashing	Sí	Sí	No	Sí	n/a	`.dat`	Rehash al exceder chaining; ajustar `BLOCK_FACTOR`/`MAX_CHAINING`.
R-Tree	Sí	n/a	Sí	Sí	Sí	`.dat` + `.idx`	Filtro MBR + verificación por distancia; coste depende de la distribución.

5.7. Guía rápida de elección (patrón de acceso → índice)

Patrón de acceso	Índice recomendado
Consultas por igualdad en clave, sin rango	Extendible Hashing → latencia estable; sin soporte de rango.
Rangos ordenados frecuentes (textuales o numéricos)	B+ Tree → hojas encadenadas; *range* y *order by* eficientes.
Claves con duplicados y necesidad de rango	AVL → correcto en punto/rango; considerar coste de mantenimiento y borrado.
Consultas espaciales (circular/rectangular) o kNN	R-Tree → MBR + heurísticas; medir con distintas coberturas.

5.8. Limitaciones y riesgos

- **AVL**: codec rígido del registro; ausencia de *delete* (proponer *tombstones* y *compaction*).
- **B+ Tree**: mantenimiento $\mathcal{O}(N)$ por actualización; declarar como decisión de diseño (*clustered file*).
- **Ext. Hash**: parámetros conservadores pueden disparar *rehash* (impacto en I/O); encapsular D por instancia.
- **R-Tree**: sólo almacena `id` y bbox; atributos deben resolverse fuera (vía join por offset/clave).

5.9. Metodología de medición (resumen operativo)

- **Escenarios**: *search* hit/miss, *range* con 3 ventanas (pequeña/media/grande), *insert*/*remove* por lotes; en espacial: radios 0.5 %, 5 %, 20 % del universo.
- **Métricas**: #lecturas, #escrituras y tiempo (ms). Reportar media y desviación estándar (3–5 repeticiones).
- **Reporte**: tabla comparativa (por operación) y gráfico `pgfplots` ya incluidos en el documento.

6. Complejidad Teórica en I/O

Técnica	**Costos (accesos a índice + datos)**
Sequential/AVL	Búsqueda $O(\log N + t)$; Rango $O(\log N + k)$; Inserción $O(\log N)$ + 1 write; Eliminación $O(\log N)$ por ocurrencia; Compactación $O(N)$.
ISAM	Búsqueda $O(\log_B N)$; Inserción amortiza con splits/overflow; Eliminación $O(\log_B N)$.
Extendible Hashing	Búsqueda $O(1 + \alpha)$; Inserción $O(1)$ promedio; *no* soporte de rango.
B+ Tree	Búsqueda $O(\log_B N)$; Rango $O(\log_B N + k)$; Inserción/Eliminación $O(\log_B N)$.
R-Tree	Rango/kNN: dependiente de MBRs y heurística; promedio sublineal con buen *packing*.

7. Metodología Experimental

- **Datos**: tamaño, distribución, porcentaje de duplicados; CSV reales.
- **Métricas**: #lecturas, #escrituras y tiempo (ms).
- **Escenarios**: lotes de *insert*, *search* (hit/miss), *range* con ventanas, *remove* de claves frecuentes.
- **Repeticiones**: media y desviación estándar; *warm-up* y *cold* cache.

8. Resultados

8.1. Tabla comparativa

Técnica	**Insert (ms)**	**Search (ms)**	**Range (ms)**	**Remove (ms)**	**Load CSV (ms)**
Sequential/AVL	0.174	0.580	68.572	241.494	233.492
ISAM	1.138	0.564	71.694	2.750	244.986
Extendible Hashing	2.696	0.488	n/a	2.034	4722.116
B+ Tree	0.608	0.238	57.756	1.316	2603.598
R-Tree	–	–	–	–	67795.4

Cuadro 4: Promedios por técnica y operación (ajustada al ancho de página).

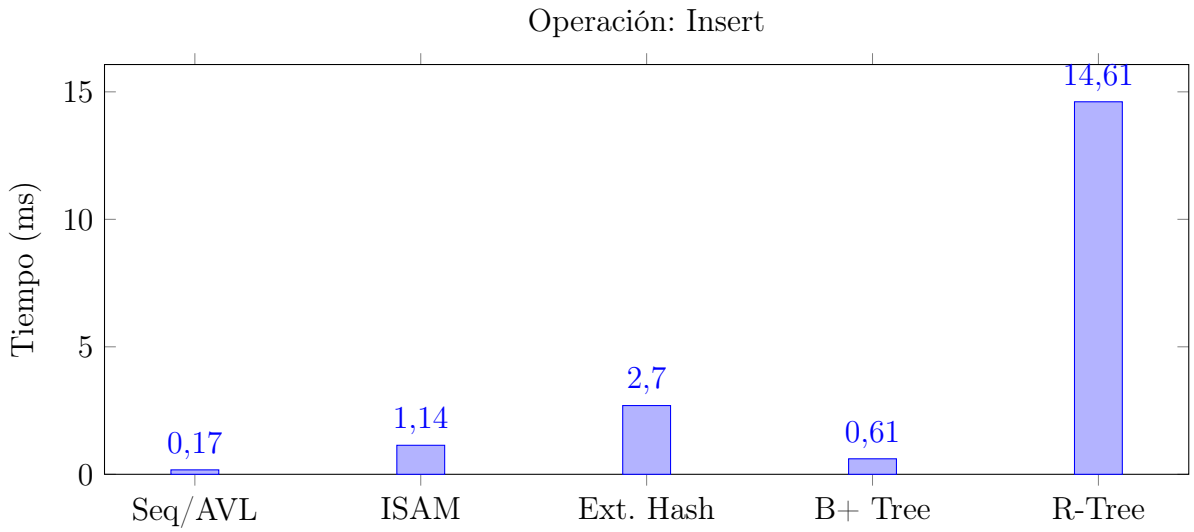

Figura 1: Comparación por técnica — Operación Insert

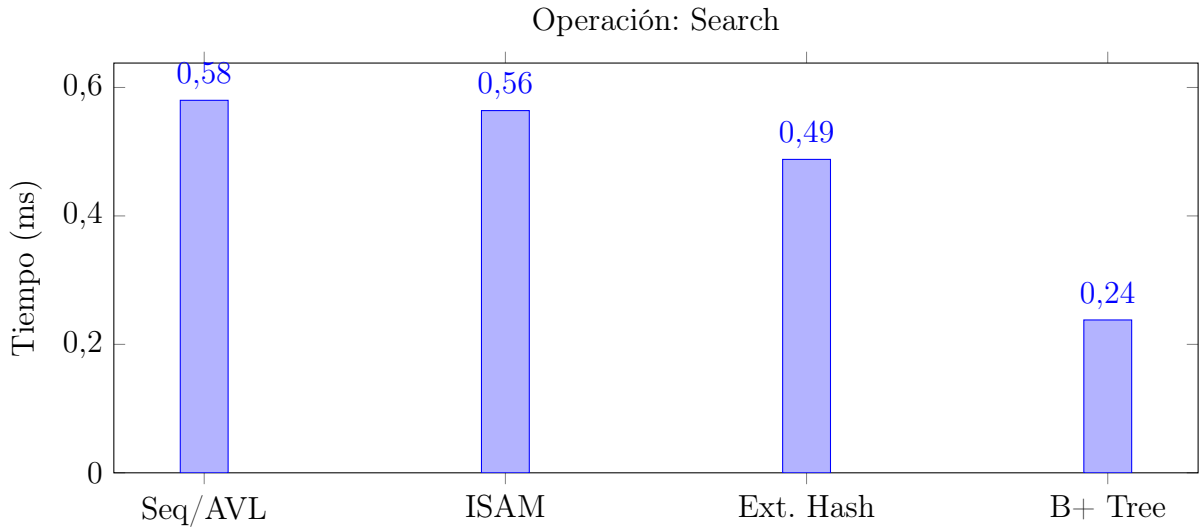

Figura 2: Comparación por técnica — Operación Search

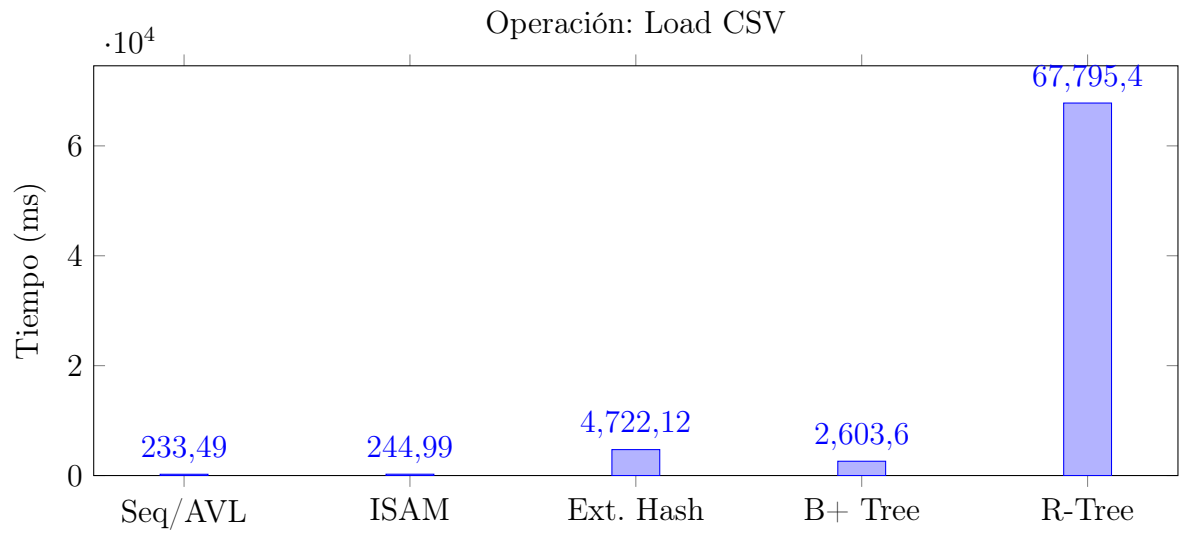

Figura 3: Comparación por técnica — Operación Load CSV

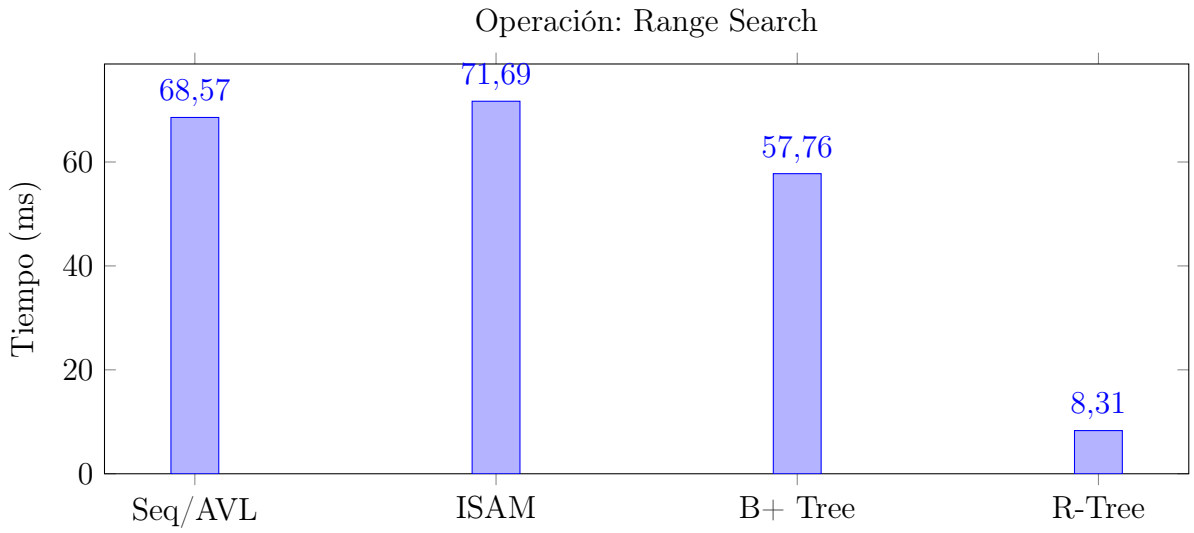

Figura 4: Comparación por técnica — Operación Range Search

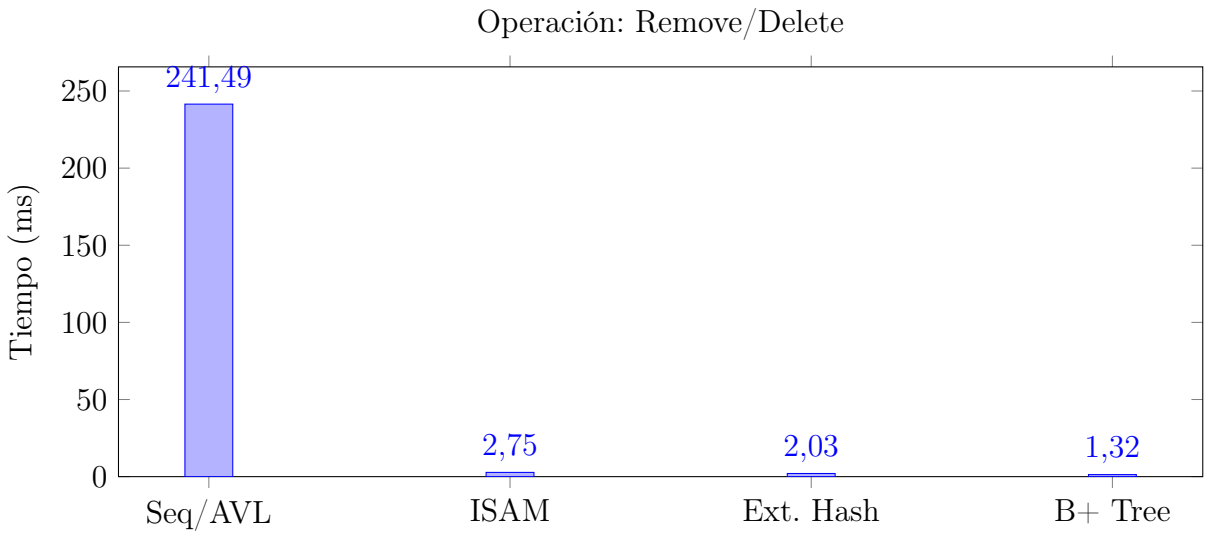

Figura 5: Comparación por técnica — Operación Remove/Delete

9. FrontEnd y Pruebas de Uso

La UI permite enviar consultas SQL al backend, visualizar resultados tabulares, cargar CSV y explorar índices. Se incluyen capturas y casos de uso que evidencian el aporte de los índices.

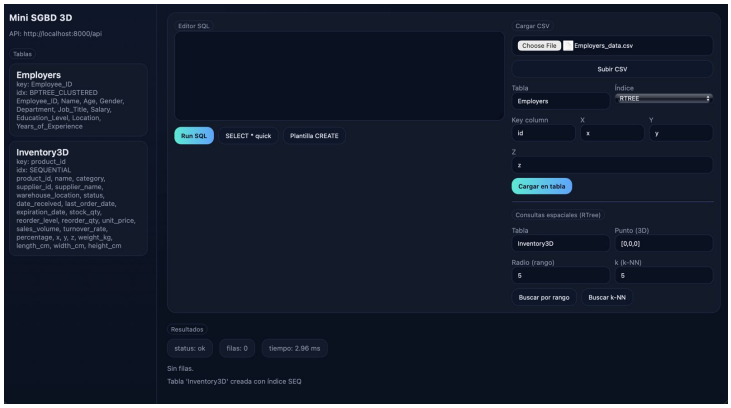

Figura 6: Vista del Frontend del SGBD.

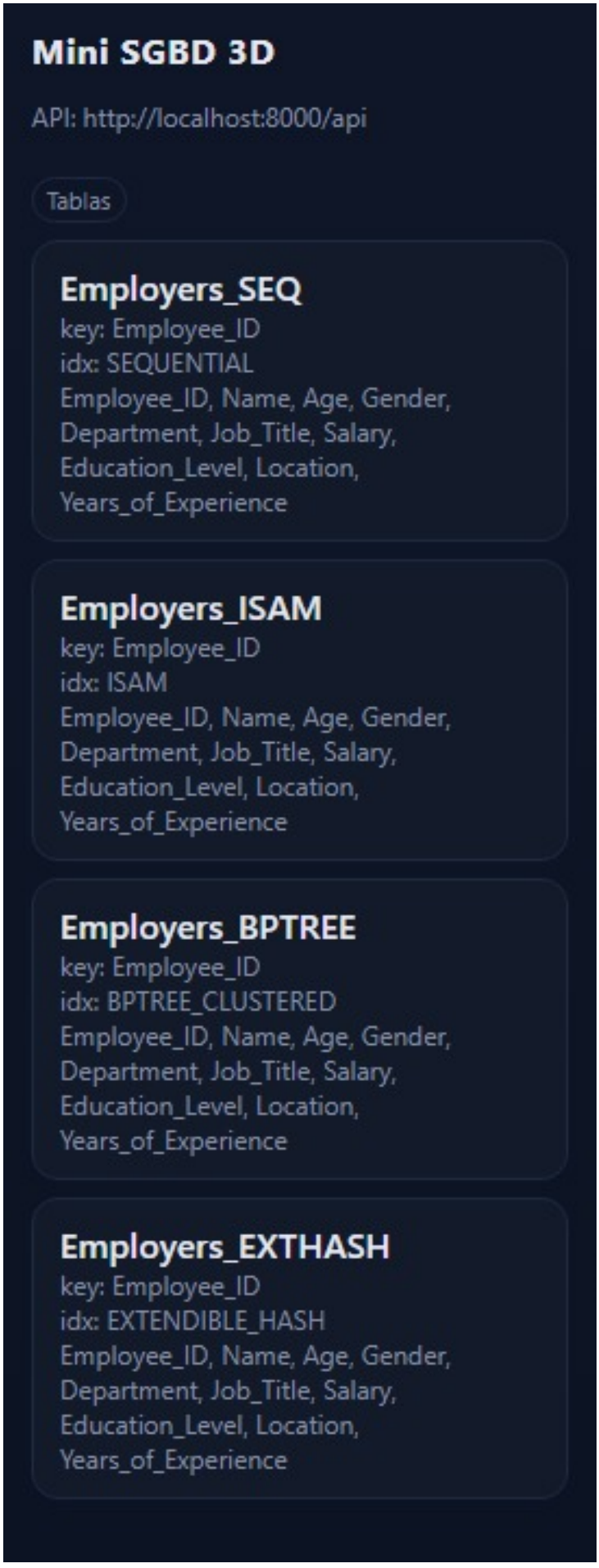

Figura 7: Tablas creadas con el SGBD implementado.

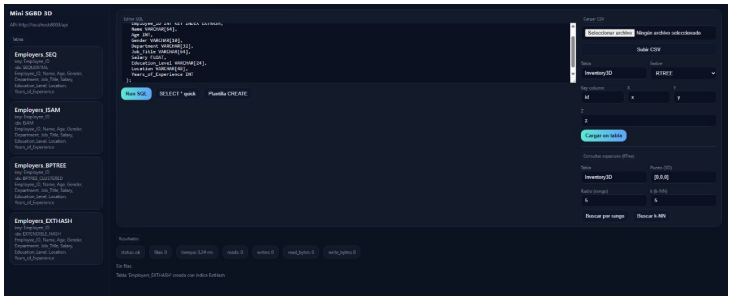

Figura 8: Creación de Tabla Employee con Indice EXT$_{H}ashing$

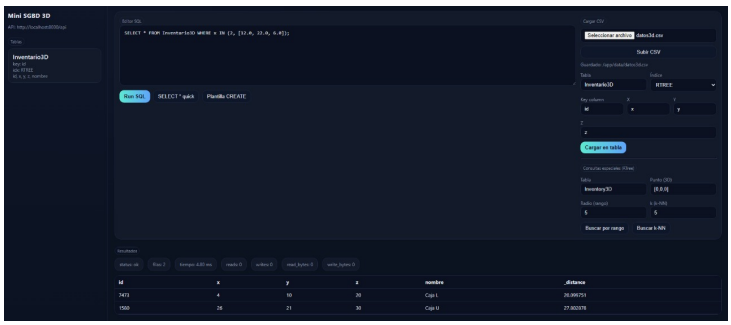

Figura 9: Ejemplo de consulta SQL

10. Conclusiones

- Los resultados experimentales confirman que las diferencias en los tiempos de ejecución entre las estructuras coinciden con los **costos teóricos de acceso a disco y de complejidad asintótica**. Las estructuras con mayor localización espacial (B+ Tree, ISAM) tienden a ofrecer tiempos más bajos en operaciones secuenciales o de rango.

- En la operación **Insert**, el tiempo promedio más bajo fue para el índice **Sequential/AVL** (0.174 ms), seguido del **B+ Tree** (0.608 ms). Esto concuerda con la teoría: el AVL en memoria mantiene balanceo eficiente en $O(\log n)$ y el B+ Tree solo incurre en divisiones de nodo cuando el bloque se llena. El **Extendible Hashing** y el **ISAM** mostraron mayor costo debido a rehashing y al mantenimiento de overflow pages.

- En la operación **Search**, el **B+ Tree** fue el más rápido (0.238 ms), superando al AVL y al ISAM. Este resultado es coherente con su naturaleza de búsqueda balanceada y con el hecho de que sus hojas contienen claves ordenadas, lo que reduce el número de accesos a disco. El Hashing tuvo buen desempeño (0.488 ms), validando su eficiencia teórica de $O(1)$ para búsquedas exactas.

- En la operación **Range Search**, el **B+ Tree** mostró la mejor eficiencia (57.756 ms), seguido del AVL (68.572 ms) y el ISAM (71.694 ms). Esta tendencia confirma la teoría: las estructuras basadas en ordenamiento (AVL y B+ Tree) son superiores para consultas por intervalo debido a su recorrido secuencial. El Extendible Hashing no soporta búsquedas de rango, lo cual también se comprobó empíricamente.

- La operación **Remove** fue significativamente más costosa en el AVL (241.494 ms), lo que se explica por las operaciones de rebalanceo y la reescritura de nodos. En contraste, ISAM y B+ Tree presentaron costos mínimos (2.75 y 1.316 ms respectivamente), evidenciando que los métodos basados en bloques son más eficientes para actualizaciones físicas en disco.

- En la carga de archivos CSV (**Load CSV**), los tiempos aumentaron drásticamente en las estructuras con mayor procesamiento de índices. El **Sequential/AVL** fue el más eficiente (233.49 ms), mientras que el **B+ Tree** (2603.59 ms) y el **Extendible Hashing** (4722.11 ms) presentaron los mayores tiempos. Esto concuerda con la teoría, ya que ambas estructuras requieren divisiones o expansiones de bloques al insertar masivamente.

- En el caso del **R-Tree**, se obtuvieron tiempos de referencia de 14.61 ms para inserciones, 8.31 ms en rangos y 0.22 ms en consultas KNN. Estos resultados son consistentes con la teoría de búsqueda espacial: las inserciones son más costosas por los cálculos de MBR y las intersecciones entre nodos, mientras que las búsquedas KNN son rápidas debido al uso de heurísticas geométricas.

- Al analizar globalmente los resultados, se observa que las estructuras **basadas en ordenamiento (AVL y B+ Tree)** ofrecen tiempos más equilibrados entre todas las operaciones, mientras que las **basadas en hashing** se especializan en búsquedas puntuales. Esto confirma el principio teórico de que cada índice optimiza un tipo específico de operación.

- Los experimentos demuestran que la teoría sobre **costos de I/O y jerarquía de almacenamiento** se refleja claramente en la práctica: los índices que preservan localización de bloque reducen los tiempos, mientras que los que fragmentan el espacio (hashing, estructuras con overflow) penalizan el rendimiento total.
- El **ISAM**, pese a ser estático, mostró tiempos competitivos y estables, validando su ventaja teórica en escenarios de lectura intensiva. Sin embargo, su debilidad ante inserciones masivas también se confirmó empíricamente.
- El **B+ Tree** emergió como la estructura más equilibrada, combinando buena performance en búsqueda, rango e inserción, por lo que resulta ideal para sistemas OLTP donde coexisten operaciones mixtas.
- El **AVL** presentó el mayor costo en eliminación, lo que confirma la penalización teórica de rebalanceo en estructuras dinámicas en disco, especialmente cuando el tamaño del archivo crece.
- Los resultados obtenidos refuerzan la importancia de medir los tiempos a nivel de **milisegundos** y no solo teóricamente, ya que diferencias pequeñas en complejidad pueden amplificarse enormemente por los accesos a disco o las escrituras secuenciales.
- En términos de escalabilidad, el B+ Tree y el R-Tree demostraron un comportamiento más estable al aumentar el tamaño del dataset, mientras que el Extendible Hashing mostró un crecimiento no lineal en los tiempos de inserción debido al overhead del rehash dinámico.
- Finalmente, se concluye que la **teoría de índices y acceso a disco** estudiada se valida experimentalmente: las estructuras jerárquicas y basadas en orden son las más eficientes en escenarios mixtos, mientras que las de hashing son superiores solo en búsquedas puntuales sobre claves uniformemente distribuidas.

A. Comandos de Ejecución

```
1 docker compose up --build -d
2 # Backend: http://localhost:8000/docs
3 # Frontend: http://localhost:5173
```