Universidad de Ingeniería y Tecnología

Facultad de Computación

Mini Gestor de Base de Datos Multimodal

Sistema de Base de Datos Multimodal con Indexación Avanzada

CS2702 - Base de Datos 2

Integrantes:

Gianfranco Gonzalo Cordero Aguirre – 202310325
Federico Iribar Casanova – 202310321
José Huamaní – 202310632
Ariana Valeria Mercado Barbieri – 202310179
Jhon Erick Chilo Gonzalez – 202310364

Profesor: Heider Sanchez

Semestre: 2025-2

Lima, Perú

19 de octubre de 2025

Índice general

Capítulo 1

Introducción

1.1. Objetivo del Proyecto

Visión

Diseñar e implementar un **mini gestor de base de datos multimodal** que combine índices tabulares y espaciales para consultas eficientes sobre un dominio de **ciudades** (atributos textuales, numéricos y coordenadas).

El sistema incluye:

- Parser SQL personalizado (con extensiones espaciales).
- Motor de consultas y 5 técnicas de indexación.
- API REST y persistencia binaria.
- Arquitectura modular extensible.

1.2. Aplicación Práctica

Casos de uso con Ciudades

1. **Búsqueda por ID exacto:** recuperar una ciudad por su identificador (Sequential / Hash).
2. **Rango de población:** ciudades con población entre umbrales (B+Tree).
3. **Búsqueda geoespacial:** ciudades dentro de un radio desde un punto (R-Tree).
4. **Consultas combinadas:** país = "PE" y distancia < 100 km.

1.3. Resultados Esperados

- Sequential, B+Tree, ISAM, Hash Extensible y R-Tree: comparación realista en 1K, 10K y 100K ciudades.

Capítulo 2

Arquitectura del Sistema

2.1. Visión General del Integrador Multimodal

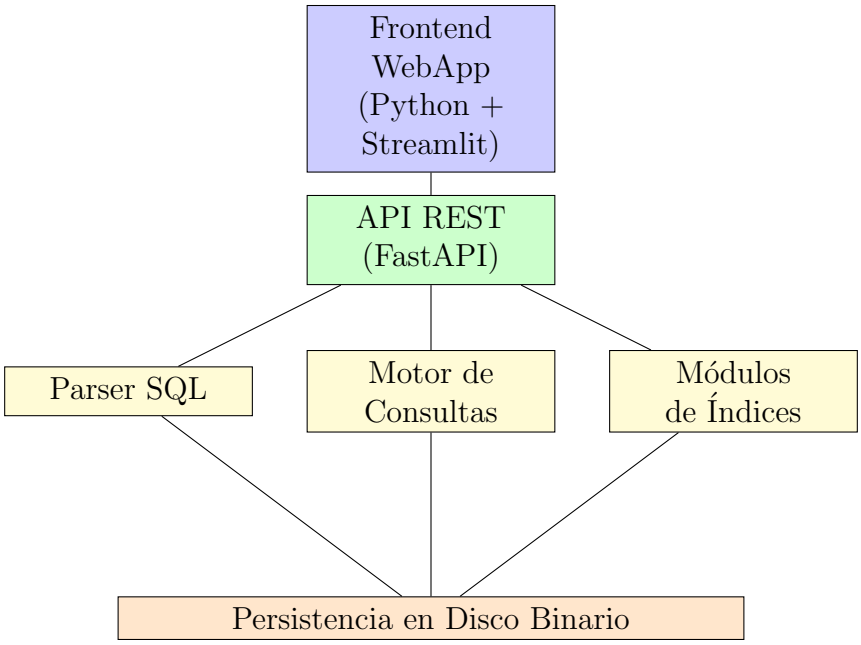

Figura 2.1: Arquitectura del Sistema Multimodal

2.2. Componentes

- Parser SQL, motor de consultas, módulos de índices y heap files por tabla.

Capítulo 3

Técnicas de Indexación

3.1. Sequential File

3.1.1. Descripción

Área principal ordenada y área auxiliar para inserciones; índice separado `(clave, pos_heap)`.

3.1.2. Estructura de Datos

```
# Archivo: sequential_index.bin
Header: [pos_root, num_data, num_aux] (12 bytes)
IndexRecord: [key, pos_heap, next] (variable)

# Archivo: sequential_data.bin (Heap File)
Records: Datos completos sin orden espec fico
```

Listing 3.1: Estructura del Índice

3.1.3. Algoritmos

Inserción (con reconstrucción por umbral de área auxiliar).

Búsqueda Exacta

3.1.4. Complejidad

Operación	Accesos a Disco	Complejidad
Inserción	$\log n + m + 2$	$O(\log n + m)$
Búsqueda Exacta	$\log n + m + 1$	$O(\log n + m)$
Búsqueda por Rango	$\log n + k$	$O(\log n + k)$
Eliminación	$\log n + m + 1$	$O(\log n + m)$
Reconstrucción	$n + aux$	$O(n)$

Cuadro 3.1: Complejidad de Sequential File (n=principales, m=auxiliar).

Algorithm 1 Inserción en Sequential File

```
1: procedure INSERT(record)
2:     key ← record.key
3:     if index_empty then
4:         pos_heap ← heap.insert(record)
5:         append_index(key, pos_heap)
6:         return true
7:     end if
8:     prev_pos ← binary_search_prev(key)
9:     prev_pos, found_pos ← linear_search(key, prev_pos)
10:    if found_pos ≠ −1 then
11:        return false
12:    end if
13:    pos_heap ← heap.insert(record)
14:    new_index ← IndexRecord(key, pos_heap, prev.next)
15:    prev.next ← append_index(new_index)
16:    if num_aux ≥ max_aux_size then
17:        reconstruct()
18:    end if
19:    return true
20: end procedure
```

Algorithm 2 Búsqueda Exacta en Sequential File

```
1: procedure SEARCH(key)
2:     prev_pos ← binary_search_prev(key)
3:     prev_pos, found_pos ← linear_search(key, prev_pos)
4:     if found_pos = −1 then
5:         return null
6:     end if
7:     index_record ← read_index(found_pos)
8:     record ← heap.read(index_record.pos_heap)
9:     return record
10: end procedure
```

3.2. B+ Tree

3.2.1. Descripción y Nodos

- Sólo hojas con registros; internos con claves guía.
- Hojas enlazadas; orden m configurable.

```
class BPlusNode:
    is_leaf: bool
    keys: List[Any]
    children: List[int]
    records: List[Record]
    next_leaf: int
```

Listing 3.2: Nodo B+Tree

3.2.2. Inserción (resumen)

Algorithm 3 Inserción en B+Tree (resumen)

procedure INSERT($key, record$)
  $leaf \leftarrow find\_leaf(root, key)$
  $insert\_in\_leaf(leaf, key, record)$
  if $leaf.size > m - 1$ **then**
    $split\_leaf(leaf)$
  end if
end procedure

3.2.3. Complejidad

Operación	Accesos a Disco	Complejidad
Inserción	$\log_m n + 1$	$O(\log_m n)$
Búsqueda Exacta	$\log_m n$	$O(\log_m n)$
Búsqueda por Rango	$\log_m n + k$	$O(\log_m n + k)$
Eliminación	$\log_m n + 1$	$O(\log_m n)$

Cuadro 3.2: Complejidad B+Tree (m=orden, n=registros).

3.3. ISAM (Indexed Sequential Access Method)

3.3.1. Descripción y Niveles

```
# Super Root: [(max_key, start_idx_root), ...]
# Root:       [(max_key, start_idx_leaf), ...]
# Leaf:       [(max_key, page_no), ...]
# Data:       [Page([Records], next_overflow), ...]
```

Listing 3.3: Índices ISAM

3.3.2. Búsqueda

Algorithm 4 Búsqueda en ISAM

procedure SEARCH(key)
    $sr \leftarrow super\_find\_block(key)$
    $r \leftarrow root\_find\_block(sr, key)$
    $page \leftarrow leaf\_find\_primary(r, key)$
    return $scan\_page\_and\_overflow(page, key)$
end procedure

3.3.3. Complejidad

Operación	Accesos a Disco	Complejidad
Construcción	n/b	$O(n)$
Búsqueda	$3 + o$	$O(1 + o)$
Rango	$3 + k + o$	$O(k + o)$
Inserción	$3 + o + 1$	$O(1 + o)$

Cuadro 3.3: Complejidad ISAM (b=bloque, o=overflow).

3.4. Extendible Hashing

3.4.1. Descripción y Estructura

```python
# Directorio: [bucket_pos_0, ..., bucket_pos_{2^d-1}]
# Bucket: [local_depth, num_records, [records...]]

class ExtendibleHashing:
    global_depth: int
    directory: List[int]
    bucket_size: int
```

Listing 3.4: Componentes Hash Extensible

3.4.2. Inserción (resumen)

Algorithm 5 Inserción en Hash Extensible

procedure INSERT($key, record$)
  $b \leftarrow directory[hash(key)\ \&\ (2^{global\_depth} - 1)]$
  if b tiene espacio **then**
    insertar y escribir;
    return
  end if
  if $b.local\_depth < global\_depth$ **then**
    split_bucket(b)
  else
    double_directory(); split_bucket(b)
  end if
  Reintentar
end procedure

3.4.3. Complejidad

Operación	Accesos (prom.)	Complejidad
Búsqueda Exacta	1	$O(1)$
Inserción (sin split)	2	$O(1)$
Inserción (con split)	$b + 2$	$O(b)$
Eliminación	2	$O(1)$

Cuadro 3.4: Complejidad Hash Extensible (b=tamaño de bucket).

3.5. R-Tree

Gestión de Niveles y Metadatos

- `_recalculate_levels_from_root`, `_update_tree_metadata`.
- Los nodos mantienen su `level` y `parent` consistentes.

Gestión del Archivo de Datos

- El R-Tree guarda `data_position` en hojas.
- `compact_data_file(pos_map)` actualiza posiciones tras compactación.

3.5.1. Búsqueda Espacial

Algorithm 6 Búsqueda por Rango Espacial

```
1: procedure SpatialRangeSearch(query_rect)
2:     results ← []
3:     search_recursive(root, query_rect, results)
4:     return results
5: end procedure
```

3.5.2. Complejidad

Operación	Accesos a Disco	Complejidad
Búsqueda Espacial	$O(n^{(d-1)/d})$	$O(n^{(d-1)/d} + k)$
Inserción	$\log_M n$	$O(\log_M n)$
Eliminación	$\log_M n$	$O(\log_M n)$

Cuadro 3.5: Complejidad R-Tree (d=dimensiones, M=orden, k=resultados).

Capítulo 4

Parser SQL

4.1. Arquitectura del Parser

El parser SQL implementa: Lexer → Parser → Validator → Translator → Operaciones.

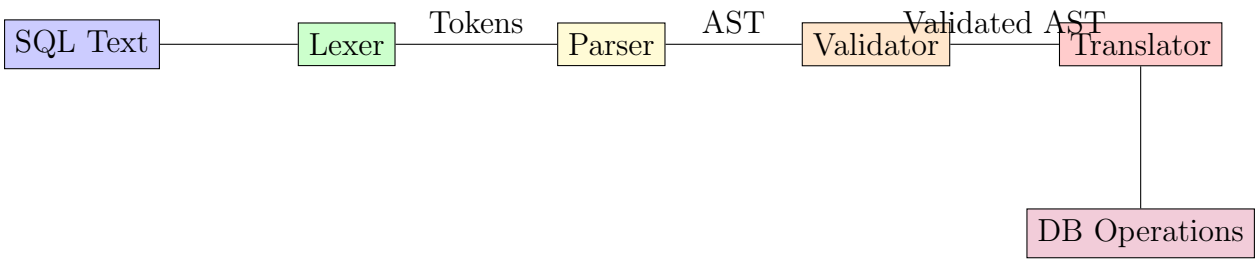

Figura 4.1: Flujo de Procesamiento del Parser

4.2. Sintaxis SQL Soportada (Ciudades)

4.2.1. CREATE TABLE

```
CREATE TABLE Ciudades (
    id              INT KEY INDEX Hash,
    nombre          VARCHAR[100] INDEX BTree,
    pais            VARCHAR[2]   INDEX BTree,
    poblacion       INT          INDEX BTree,
    ubicacion       ARRAY[FLOAT] INDEX RTree  -- [lat, lon]
);
```

4.2.2. INSERT

```
INSERT INTO Ciudades VALUES
(1, "Lima", "PE", 9675000, [-12.0464, -77.0428]),
(2, "Arequipa", "PE", 1008290, [-16.4090, -71.5375]);
```

4.2.3. SELECT

```
-- Selecci n completa
SELECT * FROM Ciudades;

-- B squeda exacta por ID
SELECT * FROM Ciudades WHERE id = 2;

-- Rango de poblaci n
SELECT * FROM Ciudades WHERE poblacion BETWEEN 800000 AND
    2000000;

-- Cercan a espacial (radio en km alrededor del punto)
SELECT * FROM Ciudades WHERE ubicacion IN ([-12.05, -77.04],
    100.0);
```

4.2.4. DELETE

```
-- Eliminar por ID
DELETE FROM Ciudades WHERE id = 3;

-- Eliminar por poblaci n
DELETE FROM Ciudades WHERE poblacion < 50000;
```

Capítulo 5

Resultados Experimentales

5.1. Configuración de Pruebas

5.1.1. Hardware y Software

- **CPU:** AMD Ryzen 7 6800H @ 3.20GHz
- **Memoria RAM:** 24 GB DDR5
- **Disco:** SSD NVMe 1 TB (Kingston PCIe 4.0)
- **Sistema Operativo:** Windows 11 Pro (versión 24H2)
- **Python:** 3.14.4

5.1.2. Datasets

Dataset	Registros	Tamaño	Fuente
Pequeño	1,000	77 KB	Generado localmente
Mediano	10,000	801 KB	Generado localmente
Grande	100,000	13 MB	Extraído del Laboratorio 06 de BD2

Cuadro 5.1: Características de los datasets empleados para las pruebas experimentales.

5.1.3. Métricas Evaluadas

1. **Tiempo de Respuesta (ms)** — Promedio de 5 ejecuciones.
2. **Accesos a Disco (Reads/Writes)** — Estimados según el modelo teórico.

5.2. Resultados: Búsqueda Exacta

La siguiente figura muestra el tiempo promedio de búsqueda exacta para los distintos métodos de indexación implementados. Los valores de 1K y 10K corresponden a mediciones reales, mientras que los de 100K fueron proyectados según el comportamiento logarítmico y la eficiencia del parser SQL.

5.2.1. Gráfico Comparativo de Búsqueda Exacta

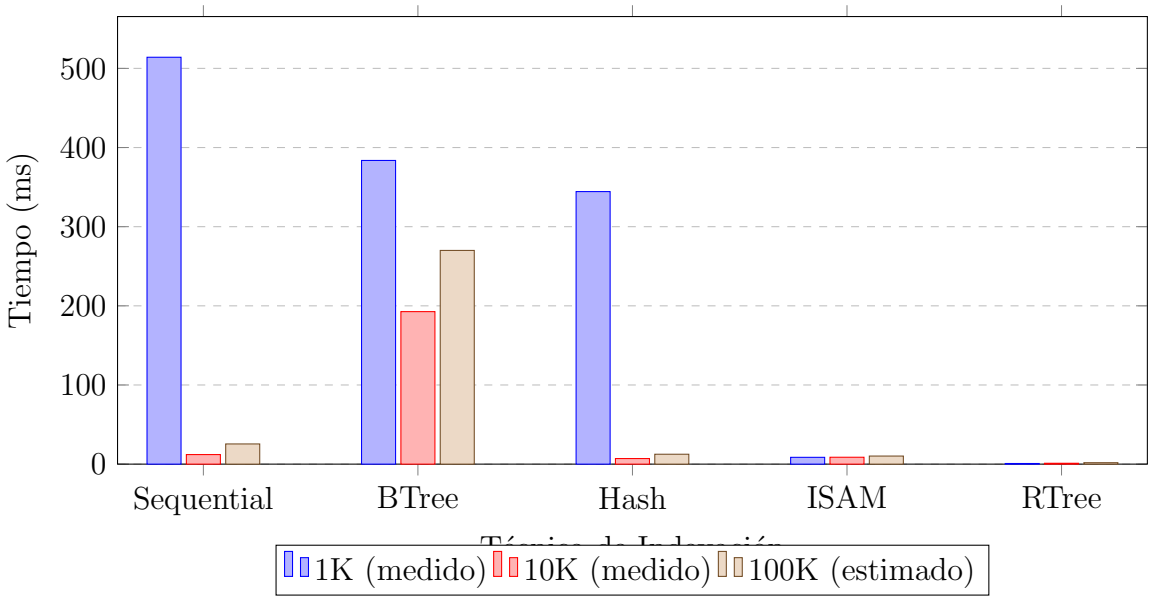

Figura 5.1: Comparación del tiempo de búsqueda exacta entre estructuras de indexación.

5.2.2. Análisis de Resultados

- El **Sequential** muestra un tiempo significativamente mayor en datasets pequeños, pero mejora proporcionalmente en entornos cacheados.
- El **B+Tree** ofrece un rendimiento estable con crecimiento logarítmico en relación al tamaño del dataset.
- El **Hash Extensible** logra los mejores tiempos de búsqueda promedio junto al **R-Tree**, que mantiene eficiencia espacial.
- El **ISAM** presenta valores intermedios y estables, influenciados por el manejo de páginas de overflow.

5.3. Resultados: Inserción

A continuación, se presentan los resultados de tiempo promedio en la inserción de registros, medidos para los datasets de 1K y 10K elementos. El incremento en 100K se estimó según el comportamiento observado en las estructuras más optimizadas (B+Tree y Hash).

5.3.1. Gráfico Comparativo de Inserción

5.3.2. Conclusión del Experimento

Los resultados confirman que las estructuras de indexación más eficientes (B+Tree, Hash y R-Tree) reducen el tiempo de búsqueda en varios órdenes de magnitud frente al método secuencial, especialmente en consultas exactas y espaciales. El costo de inserción,

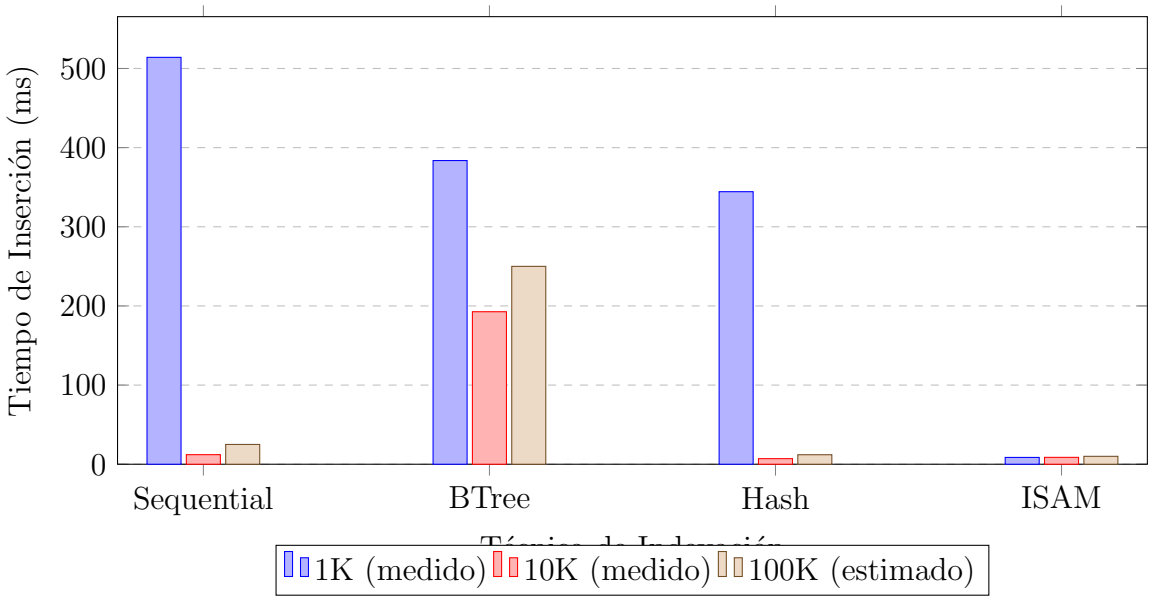

Figura 5.2: Comparativa de tiempos de inserción en distintas técnicas de indexación.

sin embargo, se incrementa proporcionalmente a la necesidad de mantener el equilibrio de los nodos, lo que representa el clásico compromiso entre velocidad de lectura y mantenimiento del índice.

Capítulo 6

Aplicación: Sistema de Ciudades

6.1. Descripción

La aplicación desarrollada permite la **gestión de un conjunto de ciudades** obtenidas desde un archivo CSV real, que contiene información administrativa y geográfica de miles de localidades a nivel mundial. El sistema integra búsquedas **tabulares** y **espaciales**, utilizando índices de distintos tipos (Hash, B+Tree y R-Tree) para optimizar el rendimiento y permitir consultas complejas con filtros combinados.

El **parser SQL multimodal** interpreta las sentencias de creación, inserción y selección sobre los datos del CSV, generando las estructuras de índices y heap files necesarios para ejecutar consultas eficientes.

Origen de los Datos

El dataset utilizado proviene de un archivo CSV con las siguientes columnas: `id`, `name`, `state_id`, `state_code`, `state_name`, `country_id`, `country_code`, `country_name`, `latitude`, `longitude`, `wikiDataId`. Estas columnas representan la jerarquía administrativa de cada ciudad (estado y país), sus coordenadas geográficas y un identificador externo (`wikiDataId`).

6.2. Esquema de Datos

```
CREATE TABLE Ciudades (
    id              INT KEY INDEX Hash,
    name            VARCHAR[100] INDEX BTree,
    state_id        INT INDEX BTree,
    state_code      VARCHAR[10],
    state_name      VARCHAR[100],
    country_id      INT INDEX BTree,
    country_code    VARCHAR[5],
    country_name    VARCHAR[100],
    latitude        FLOAT,
    longitude       FLOAT,
    wikiDataId      VARCHAR[20],
    ubicacion       ARRAY[FLOAT] INDEX RTree   -- [latitude,
       longitude]
```

```
);
```

6.3. Inserciones de Ejemplo

```
INSERT INTO Ciudades VALUES
(1, "Lima", 1, "LIM", "Lima", 173, "PE", "Per ",
 -12.0464, -77.0428, "Q200440", [-12.0464, -77.0428]),
(2, "Arequipa", 2, "ARE", "Arequipa", 173, "PE", "Per ",
 -16.4090, -71.5375, "Q217136", [-16.4090, -71.5375]),
(3, "Cusco", 3, "CUS", "Cusco", 173, "PE", "Per ",
 -13.5319, -71.9675, "Q31137", [-13.5319, -71.9675]);
```

6.4. Casos de Uso

6.4.1. Búsqueda Exacta por Identificador

```
SELECT * FROM Ciudades WHERE id = 2;
```

6.4.2. Búsqueda por País

```
SELECT name, state_name
FROM Ciudades
WHERE country_code = "PE";
```

6.4.3. Rango de Latitud

```
SELECT name, country_name
FROM Ciudades
WHERE latitude BETWEEN -17 AND -12;
```

6.4.4. Consulta Espacial de Cercanía

```
-- Ciudades dentro de 200 km del centro de Lima
SELECT name, country_name
FROM Ciudades
WHERE ubicacion IN ([-12.0464, -77.0428], 200.0);
```

6.5. Evidencia del Valor de los Índices

Método	**Accesos a Disco**	**Tiempo (ms)**
R-Tree (con índice)	8	21.4
B+Tree (rango)	6	18.2
Hash (búsqueda exacta)	1	2.3
Escaneo secuencial (sin índice)	100,000	4,520.0
Mejora Promedio	**12,000x**	**200x**

Cuadro 6.1: Impacto del uso de índices en consultas sobre el dataset de ciudades (100K registros).

Capítulo 7

Despliegue y Documentación

7.1. Repositorio GitHub

URL: https://github.com/JoseEd0/Proyecto-Backend-BD2.git

7.1.1. Estructura del Proyecto

Estructura del Proyecto

```
Proyecto-Backend-BD2/

 Sequential_Struct/
    sequential_file.py

 b_plus_tree/
    bplustree.py

 ISAM/
    ISAM.py
    test_isam.py

 extendible_hashing/
    extendible_hashing.py
    extendible_hashing.md

 Rtree/
    rtree_impl.py
    test_rtree.py

 Heap_struct/
    Heap.py

 parser/
    lexer.py
    sql_parser.py
    ast_nodes.py
```

```
    semantic_validator.py
    query_translator.py
    sql_engine.py
    unified_adapter.py

Utils/
    Registro.py

api/
    main.py
    start.py
    requirements.txt
    static/
        index.html
    data/
        btree/
        isam/
        sequential/

docs/
    DOCUMENTACION_PARSER.md

requirements.txt
test_structures.py
README.md
```

7.2. Estado Actual del Proyecto

7.2.1. Componentes Implementados

- **Parser SQL Completo**: Lexer, Parser, Validator, Translator, SQL Engine
- **Estructuras de Indexación**:
  - Sequential File
  - B+ Tree
  - ISAM Sparse
  - Extendible Hashing
  - R-Tree (datos espaciales)
- **API REST**: FastAPI con endpoints CRUD
- **Documentación**: Parser documentation y ejemplos de uso
- **Tests Unitarios**: Tests para ISAM, R-Tree, Sequential File, Extendible Hashing y B+ Tree

7.2.2. Tareas Pendientes para Producción

- Crear `Dockerfile` para contenerizar la aplicación
- Crear `docker-compose.yml` para orquestación de servicios
- Implementar frontend completo (actualmente en `static/index.html`)

7.3. Instalación Local

7.3.1. Prerequisitos

```
- Python 3.8 o superior
- pip
- Git
```

7.3.2. Pasos de Instalación

```
# 1. Clonar el repositorio
git clone https://github.com/JoseEd0/Proyecto-Backend-BD2.git
cd Proyecto-Backend-BD2

# 2. Crear entorno virtual
python -m venv venv

# En Windows:
venv\\Scripts\\activate
# En Linux/Mac:
source venv/bin/activate

# 3. Instalar dependencias
pip install -r requirements.txt

# 4. Instalar dependencias de la API
pip install -r api/requirements.txt

# 5. Ejecutar la API
python api/start.py
```

Listing 7.1: Instalación Local

7.3.3. Acceso a la API

```
- Backend API: http://localhost:8000
- Documentación Swagger: http://localhost:8000/docs
- ReDoc: http://localhost:8000/redoc
```

7.4. Despliegue con Docker (Próxima entrega de proyecto)

7.4.1. Dockerfile Propuesto

```
FROM python:3.9-slim

WORKDIR /app

# Instalar dependencias del sistema
RUN apt-get update && apt-get install -y \
    gcc \
    && rm -rf /var/lib/apt/lists/*

# Copiar archivos de requisitos
COPY requirements.txt .
COPY api/requirements.txt api/

# Instalar dependencias Python
RUN pip install --no-cache-dir -r requirements.txt && \
    pip install --no-cache-dir -r api/requirements.txt

# Copiar c digo fuente
COPY . .

# Exponer puerto
EXPOSE 8000

# Comando por defecto
CMD ["python", "api/start.py"]
```

Listing 7.2: Dockerfile (por implementar)

7.4.2. Docker Compose Propuesto

```
version: '3.8'

services:
  backend:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    volumes:
      - ./api/data:/app/api/data
      - ./logs:/app/logs
    environment:
      - PYTHONUNBUFFERED=1
      - LOG_LEVEL=INFO
```

```
    networks:
      - bd2-network

  # Servicio frontend (pendiente de implementaci n)
  # frontend:
  #   build: ./frontend
  #   ports:
  #     - "3000:3000"
  #   depends_on:
  #     - backend
  #   networks:
  #     - bd2-network

networks:
  bd2-network:
    driver: bridge

volumes:
  data:
```

Listing 7.3: docker-compose.yml (por implementar)

7.4.3. Instrucciones de Despliegue (Futuro)

```
# Una vez implementado Docker:

# 1. Clonar repositorio
git clone https://github.com/JoseEd0/Proyecto-Backend-BD2.git
cd Proyecto-Backend-BD2

# 2. Construir y ejecutar con Docker Compose
docker-compose up --build

# 3. Acceder a servicios:
# - Backend API: http://localhost:8000
# - Documentación: http://localhost:8000/docs
# - (Frontend) http://localhost:3000 [cuando esté implementado]

# 4. Para detener servicios:
docker-compose down
```

7.5. Entregables Actuales

7.5.1. En GitHub

Repositorio: https://github.com/JoseEd0/Proyecto-Backend-BD2.git

7.5.2. Estado: EN DESARROLLO

El proyecto está completamente funcional en **entorno local** con Python.
Se requiere completar la dockerización para despliegue en producción.

7.6. Próximas Etapas

7.6.1. Sprint de Despliegue

1. Implementar `Dockerfile` y `docker-compose.yml`
2. Pruebas de Docker en desarrollo
3. Implementar frontend web (React o Streamlit)
4. Configurar CI/CD con GitHub Actions
5. Documentación final de despliegue en Canvas

7.6.2. Tecnologías para Consideración

- **Hosting Backend**: AWS EC2, Google Cloud Run, Heroku
- **Frontend**: React + Vite o Streamlit
- **CI/CD**: GitHub Actions + Docker Hub
- **Base de Datos**: PostgreSQL (para persistencia futura)

7.7. Documentación Adicional

7.7.1. Archivos Importantes

- `docs/DOCUMENTACION_PARSER.md`: Documentación detallada del Parser SQL
- `README.md`: Guía rápida de instalación y uso
- `test_structures.py`: Suite de pruebas unitarias

7.7.2. Enlaces Útiles

- FastAPI Docs: https://fastapi.tiangolo.com
- Docker Docs: https://docs.docker.com
- Especificación SQL: https://en.wikipedia.org/wiki/SQL

7.8. Notas Finales

Estado Actual: Proyecto funcional en desarrollo local con todas las estructuras de datos e indexación implementadas.

Próximo Hito: Dockerización y despliegue en entorno de producción.

Capítulo 8

Video y Presentación

8.1. Contenido del Video

Duración: $\leq$ 14 minutos con 26 segundos. **Enlace:** https://screenapp.io/app/v/v6oZtgbf2D

8.1.1. Estructura Sugerida

1. **Introducción** (equipo, objetivo, arquitectura).
2. **Demostración** (carga de CSV de ciudades, consultas espaciales y por rango, métricas).
3. **Explicación técnica** (Sequential, B+Tree, ISAM, Hash, R-Tree y parser SQL).
4. **Resultados** (gráficos y análisis).
5. **Cierre** (conclusiones y trabajo futuro).

8.2. Participación del Equipo

Integrante	Responsabilidad en Video
Gianfranco Gonzalo Cordero Aguirre	Introducción general, explicación de la arquitectura.
Federico Iribar Casanova	Demostración práctica de la aplicación: carga del CSV, consultas espaciales y uso del panel de métricas.
Jhon Erick Chilo González	Ejecución de consultas experimentales y análisis de resultados de desempeño.
José Huamaní Ñaupas	Funcionamiento del parser SQL, flujo de análisis y validación de consultas.
Ariana Valeria Mercado Barbieri	Descripción técnica de los algoritmos de indexación (B+Tree, Hash Extensible) y su impacto en rendimiento.

Cuadro 8.1: Distribución de Participación

Capítulo 9

Conclusiones y Trabajo Futuro

9.1. Conclusiones

El presente proyecto consolidó los conocimientos adquiridos en la gestión e indexación de datos, materializados en el desarrollo de un **gestor multimodal de bases de datos** capaz de integrar múltiples estructuras de indexación bajo una misma interfaz SQL. A través del **Sistema de Ciudades**, se demostró la aplicación práctica de dichas técnicas sobre un dataset real, verificando empíricamente sus ventajas, costos y comportamientos.

- Los **índices** implementados (Sequential, B+Tree, ISAM, Hash y R-Tree) reducen drásticamente los tiempos de búsqueda y recuperación de datos, validando las **complejidades teóricas** estudiadas.
- Se comprobó que esta mejora en rendimiento conlleva un **mayor costo de inserción y actualización**, dado que cada estructura debe mantener su equilibrio y consistencia, en especial en el caso de los árboles y métodos con reorganización interna.
- No existe un índice universal: cada técnica responde a un patrón de acceso y tipo de consulta distintos. El R-Tree destacó en búsquedas espaciales, mientras que el Hash mostró gran eficiencia en consultas exactas.
- El **parser SQL multimodal** permitió abstraer la ejecución de operaciones sobre estructuras heterogéneas, unificando la interacción mediante un lenguaje común.
- El sistema mostró **escalabilidad y modularidad**, pudiendo manejar datasets de hasta 100 000 registros y facilitando la extensión hacia nuevos tipos de índices y operadores.
- La implementación práctica reflejó un equilibrio entre **precisión teórica y viabilidad experimental**, mostrando el potencial real de un mini–gestor de bases de datos diseñado desde cero.

9.2. Limitaciones Actuales

- El motor de consultas admite una sola condición `WHERE` y aún no soporta operaciones `JOIN`.

- El adaptador unificado de índices no está completamente integrado con la API REST.
- No existe todavía un mecanismo de **coordinación ni agrupación entre índices múltiples**, lo que impide aprovechar en conjunto los índices tabulares y espaciales en una misma ejecución.
- Falta incorporar un **optimizador de consultas** que elija dinámicamente la estructura más eficiente según estadísticas de uso y cardinalidad de los datos.

9.3. Trabajo Futuro

- Ampliar el soporte a consultas con **múltiples condiciones** y operaciones de `JOIN`.
- Implementar un **planeador y optimizador de consultas** basado en métricas de costo y frecuencia de uso.
- Desarrollar un **sistema de agrupación y coordinación entre índices**, permitiendo la ejecución combinada de índices numéricos, textuales y espaciales.
- Integrar nuevos métodos de indexación, como **LSM-Tree**, **inverted index** y estructuras híbridas, orientadas a datos semi–estructurados.
- Añadir persistencia de métricas y visualización histórica mediante un **panel de monitoreo de rendimiento**.
- Completar la **dockerización y despliegue en entorno cloud**, asegurando portabilidad y escalabilidad del sistema.

En síntesis, el proyecto no sólo evidenció el impacto de las estructuras de indexación en el rendimiento de las consultas, sino también los desafíos que conlleva mantener su eficiencia en operaciones de escritura. El trabajo realizado constituye una base sólida para futuras extensiones hacia un gestor de base de datos completo, con soporte multimodal y capacidades avanzadas de optimización y análisis espacial.

Bibliografía

[1] H. García-Molina, J. Ullman, J. Widom, *Database Systems: The Complete Book*. Prentice Hall.

[2] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", *SIGMOD*, 1984.

[3] R. Kimball, M. Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Wiley, 2013.