Universidad de Carabobo

Facultad Experimental de Ciencias y Tecnología

Departamento de Computación

Simulador De Datos Médicos

Autor:

Martin Natera 3044534

Curso:

Arquitectura del computador

Fecha:

10 de octubre de 2025

1. Descripción del problema

El proyecto consiste en el desarrollo de un sistema en C++ para simular procesamiento de datos medicos en memoria con persistencia de datos por medio de una base de datos y utilizando Boost Multi-Index para búsquedas eficientes. Los objetivos principales incluyen:

- Implementar un sistema de persistencia de datos usando una base de datos ligera
- Utilizar Boost Multi-Index para búsquedas rápidas O(log n) en memoria
- Sincronizar datos entre memoria RAM y almacenamiento persistente
- Gestionar información por medio de estructuras
- Permitir búsquedas eficientes por múltiples criterios simultáneos

El software se desarrolla para entornos médicos simulados donde es crucial mantener registros de pacientes de manera organizada y eficiente, permitiendo acceso rápido a la información y garantizando la persistencia de los datos.

2. Análisis de la solución

Cómo se organizan y almacenan los datos

Los datos se organizan en tres niveles principales:

1. **Almacenamiento en memoria**: Los pacientes se almacenan en un contenedor Boost Multi-Index que mantiene múltiples índices automáticos para búsquedas rápidas.
2. **Almacenamiento persistente**: Los datos se guardan en LevelDB usando el ID del paciente como clave y los metadatos como valor en formato delimitado.
3. **Interfaz de usuario**: La clase `MenuPrincipal` gestiona la interacción con el usuario mediante un sistema de menús jerárquico.

Justificación de las decisiones tomadas

- **STL/vector**: Permite el uso de estructura de almacenamiento en memoria de los datos obtenidos.
- **Boost Multi-Index**: Permite múltiples índices automáticos sobre los mismos datos, proporcionando búsquedas O(log n) sin duplicación.
- **LevelDB**: Base de datos clave-valor eficiente y ligera, ideal para aplicaciones embebidas.
- **Índices múltiples**: Cinco índices diferentes (ID, nombre, modalidad, sexo, secuencial) para cubrir todos los casos de uso.

3. Estructuras de datos

Estructura PacienteData para Boost Multi-Index

```
    struct PacienteData {
      string patientID;           // Identificador unico (indice unico)
      string patientName;         // Nombre (indice ordenado)
      string studyDate;           // Fecha del estudio
      string modality;           // Modalidad (indice agrupado)
      string sex;                // Sexo (indice agrupado)
      long long tamanoArchivo;   // Tamano del archivo

      // Conversiones desde/hacia DataPaciente
      PacienteData(const DataPaciente& dp);
      DataPaciente toDataPaciente() const;
    };

```

Contenedor Multi-Index con 5 índices

```
    typedef multi_index_container<
    PacienteData,
    indexed_by<
    ordered_unique<member<PacienteData, string, &PacienteData::patientID
    >>,      // Indice 0: ID unico
    ordered_non_unique<member<PacienteData, string, &PacienteData::
    patientName>>, // Indice 1: Nombre
    ordered_non_unique<member<PacienteData, string, &PacienteData::
    modality>>,    // Indice 2: Modalidad
    ordered_non_unique<member<PacienteData, string, &PacienteData::sex
    >>,         // Indice 3: Sexo
    sequenced<>
             // Indice 4: Secuencial
    ]
    > PacienteContainer;

```

Clase SistemaPacientes con Boost Multi-Index

```
    class SistemaPacientes {
      private:
      PacienteContainer pacientesContainer;  // Contenedor multi-index
      LevelDBManager leveldb;                // Gestor de base de datos
      public:
      // Busquedas O(log n) usando diferentes indices
      vector<DataPaciente> buscarPorNombre(const string& nombre) const;
      vector<DataPaciente> buscarPorID(const string& id) const;
      vector<DataPaciente> buscarPorModalidad(const string& modalidad)
    const;
      vector<DataPaciente> buscarPorSexo(const string& sexo) const;
      DataPaciente* buscarExactoPorID(const string& id); // O(log n)
    };

```

Estructuras y algoritmos aplicados

1. **Boost Multi-Index Container**: Contenedor principal con 5 índices automáticos
2. **ordered_unique**: Índice único por ID (clave primaria)
3. **ordered_non_unique**: Índices no únicos para nombre, modalidad y sexo
4. **sequenced**: Índice que mantiene el orden de inserción
5. **LevelDB**: Base de datos clave-valor para persistencia

Algoritmos de Búsqueda

Algorithm 1 Búsqueda por ID (Clave Primaria)

```
function buscarExactoPorID(id)
    index ← get(0)                                   ▷ Índice por ID
    it ← find(index, id)
    if it ≠ end(index) then
        return it.toDataPaciente()
    end if
    return nullptr
end function
```

Algorithm 2 Búsqueda por Nombre (Búsqueda Parcial)

```
function buscarPorNombre(nombre)
    resultados ← {}
    nombreBusqueda ← aMinusculas(nombre)
    index ← get(1)                                   ▷ Índice por nombre
    for it ← begin(index) to end(index) do
        nombrePaciente ← aMinusculas(it.patientName)
        if find(nombrePaciente, nombreBusqueda) ≠ string :: npos then
            push_back(resultados, it.toDataPaciente())
        end if
    end for
    return resultados
end function
```

Algoritmos de Inserción y Eliminación

Algorithm 3 Inserción de Paciente

```
function agregarPaciente(paciente)
    if existePaciente(paciente.getPatientID()) then
        return                                       ▷ Paciente ya existe
    end if
    insert(pacientesContainer, PacienteData(paciente))
    if leveldb.isConnected() then
        guardarPaciente(leveldb, paciente)
    end if
end function
```

4. Algoritmos de LevelDB

Operaciones de Persistencia

Algorithm 4 Guardar Paciente en LevelDB

```
1: function GUARDARPACIENTE(id, nombre, fecha, modalidad, sexo, tamano)
2:     pacienteData ← nombre + "|" + fecha + "|" + modalidad + "|" + sexo + "|" +
   to_string(tamano)
3:     status ← Put(db, WriteOptions(), id, pacienteData)
4:     if notstatus.ok() then
5:         return false
6:     end if
7:     return true
8: end function
```

Algorithm 5 Búsqueda por Campo en LevelDB

```
1: function BUSCARPACIENTESPORCAMPO(campoIndex, valor)
2:     resultados ← {}
3:     valorBusqueda ← aMinusculas(valor)
4:     it ← NewIterator(db, ReadOptions())
5:     for it.SeekToFirst(); it.Valid(); it.Next() do
6:         campos ← parseCampos(it.value())
7:         if |campos| > campoIndex then
8:             campoValor ← campos[campoIndex]
9:             if aMinusculas(campoValor).find(valorBusqueda) ≠ string :: npos
   then
10:                push_back(resultados, formatearResultado(it, campos))
11:            end if
12:        end if
13:    end for
14:    return resultados
15: end function
```

5. Gestión en Memoria con STL/Vector

Uso de STL Vector

El código utiliza `std::vector` principalmente para:

- **Almacenamiento temporal**: Durante el procesamiento de archivos y parsing de datos
- **Resultados de búsqueda**: Retorno de múltiples pacientes que coinciden con criterios
- **Manejo de campos**: División de strings en operaciones de parsing
- **Recolección de datos**: Acumulación temporal de información antes de procesar

```cpp
#include <vector>

// Almacenamiento temporal durante procesamiento
vector<string> campos;
string campo;
size_t inicio = 0;
size_t fin = linea.find('|');

while (fin != string::npos) {
  campo = linea.substr(inicio, fin - inicio);
  campos.push_back(campo);  // Uso de vector para almacenamiento temporal
  inicio = fin + 1;
  fin = linea.find('|', inicio);
}

```

Listing 1: Uso de vector en el parsing de datos

6. Boost Multi-Index: Arquitectura Avanzada

El sistema requiere múltiples formas de acceso a los datos de pacientes simultáneamente:

Tipo de Acceso	**Requisito**	**Implementación**
Acceso rápido por ID	Clave primaria	Índice único ordenado
Búsqueda por nombre	Búsqueda parcial	Índice no único ordenado
Filtrado por modalidad	Agrupamiento	Índice no único ordenado
Filtrado por sexo	Agrupamiento	Índice no único ordenado
Orden de inserción	Secuencial	Índice secuencial

Comparativa de complejidades

Operación	**Vector (antes)**	**Multi-Index (ahora)**
Búsqueda por ID	O(n)	O(log n)
Búsqueda por nombre	O(n)	O(log n)
Búsqueda por modalidad	O(n)	O(log n + k)
Inserción	O(1)	O(log n)
Eliminación	O(n)	O(log n)

Cuadro 1: Comparativa de complejidades algorítmicas

Ventajas de la Implementación Multi-Index

1. **Eficiencia en Búsquedas**: Cada índice mantiene su propia estructura ordenada con complejidad $O(\log n)$

2. **Acceso Multi-dimensional**: Múltiples formas de acceder a los mismos datos sin duplicación

3. **Actualización Automática**: Los índices se mantienen consistentes automáticamente
4. **Memoria Optimizada**: Los datos se almacenan una vez, los índices son referencias
5. **Tipado Seguro**: Acceso type-safe a través de diferentes índices

7. Persistencia con LevelDB

Selección de LevelDB:

LevelDB fue seleccionado por las siguientes razones:

- **Base de Datos Embedida**: No requiere servidor externo
- **Alto Rendimiento**: Optimizada para operaciones de lectura/escritura
- **Persistencia Clave-Valor**: Modelo simple que se adapta a los requisitos
- **Escalabilidad**: Maneja eficientemente grandes volúmenes de datos
- **Consistencia**: Garantiza la integridad de los datos
- **Manejo de Fallos**: Recuperación automática después de caídas

Esquema de Almacenamiento

Estructura Clave-Valor:

- **Clave**: `patientID` (string)
- **Valor**: `nombre|fecha|modalidad|sexo|tamano` (string delimitado)

Limitaciones:

- **Búsquedas Secundarias**: Requieren escaneo completo de la base de datos
- **Parseo Manual**: Necesidad de parsear strings para acceder a campos individuales
- **No hay Índices Secundarios**: LevelDB solo tiene índice por clave primaria
- **Consistencia Eventual**: Entre memoria y persistencia

8. Pruebas realizadas

Descripción de los casos de prueba

1. **Carga de archivo de ejemplo con 200 pacientes**:
    - Se creó automáticamente un archivo con 200 pacientes de prueba
    - Verificación: Todos los pacientes se cargaron correctamente en el Multi-Index
    - Persistencia: Los datos se guardaron en LevelDB
    - Índices: Los 5 índices se actualizaron automáticamente

2. **Búsquedas eficientes con diferentes índices**:
    - Búsqueda por nombre: Usando índice ordenado (O(log n))
    - Búsqueda por ID: Usando índice único (O(log n))
    - Búsqueda por modalidad: Usando equal_range (O(log n + k))
    - Búsqueda por sexo: Agrupación automática por índices
    - Verificación: Resultados consistentes entre todos los índices

9. Análisis

Este código representa un simulador avanzado de gestión de datos médicos que emula el comportamiento de sistemas hospitalarios reales mediante una arquitectura híbrida en memoria y persistente. Como simulador, no solo almacena información de pacientes sino que genera datos médicos sintéticos realistas, incluyendo tamaños de archivo basados en modalidades de estudio específicas (CT, MRI, XRAY), fechas de estudio formateadas y perfiles de pacientes diversos. La capacidad de cargar desde archivos de texto con formato compacto permite simular escenarios de migración de datos o integración con sistemas legacy, mientras que la generación automática de tamaños de archivo por modalidad reproduce fielmente las características de almacenamiento típicas en entornos médicos reales.

La simulación de múltiples patrones de acceso concurrente se logra mediante la implementación de Boost Multi-Index, que permite emular cómo diferentes roles médicos (radiólogos, administradores, especialistas) accederían a los mismos datos desde perspectivas distintas. El sistema simula búsquedas por ID para acceso rápido en emergencias, por nombre para consultas administrativas, por modalidad para análisis especializados y por sexo para estudios epidemiológicos, manteniendo en todo momento la coherencia de datos que sería crítica en un entorno médico real. Esta multi-dimensionalidad en el acceso simula eficientemente las demandas concurrentes de un hospital operativo.

La persistencia con LevelDB añade otra capa de simulación realista, emulando cómo los sistemas médicos garantizan la durabilidad de datos críticos ante fallos del sistema. El esquema clave-valor simula bases de datos NoSQL utilizadas en healthcare moderno, mientras que las operaciones de sincronización entre memoria y disco reproducen los mecanismos de consistencia que prevendrían la pérdida de información médica vital. La capacidad de realizar búsquedas directas en LevelDB versus búsquedas en memoria simula los trade-offs entre velocidad y persistencia que los administradores de sistemas médicos enfrentan diariamente.

Como simulador de pruebas, este código permite evaluar estrategias de optimización antes de su implementación en entornos productivos, demostrando cómo la combinación de contenedores especializados en memoria con bases de datos embedidas puede resolver problemas complejos de gestión de datos médicos. La arquitectura sirve como banco de pruebas para comprender cómo escalarían las operaciones con miles de pacientes y cómo diferentes estrategias de indexación impactarían el rendimiento en situaciones simuladas de alta demanda, todo ello sin riesgo para pacientes reales.