Instituto Tecnológico de Costa Rica

Ingeniería en computadores

Segundo semestre 2024

Fecha: 8 de septiembre de 2024

Algoritmos y estructuras de datos 1

Grupo #2

Segundo proyecto

Estudiantes:

Pablo Esteban Rivera Redondo Carnet: 2023395989

Saddy Guzman Rojas Carnet: 2023088184

Contenido

Introducción

En la presente documentación se expondrá la descripción del problema y de la solución que fue encontrada para este segundo proyecto programado del curso, el cual fue realizado en parejas, se explicara la solución presentada, así como alternativas consideradas y problemas encontrados durante el desarrollo del proyecto.

El proyecto consiste en la elaboración de un sistema de manejo de bases de datos además de la implementación de un árbol binario para optimizar las búsquedas en las bases de datos.

Descripción del problema

El problema principal sobre este proyecto es el diseñar un mini motor de base de datos relacional simple con el fin de lograr comprender mejor el funcionamiento interno de sistemas como MySQL, Este programa llamado TinySQL debe ser capaz de recibir y procesar sentencias en formato SQL y a partir de ellas ejecutar las operaciones necesarias relacionadas a la gestión de datos, como INSERT, SET, DELETE, etc. que crean, eliminan o muestran los datos. Además, se debe implementar la capacidad del manejo de índices con árboles para que las consultas sean más rápidas y en general el rendimiento de la base de datos sea mejor.

El desafío como tal radica en la poca experiencia del equipo desarrollando programas en C# complejos y en que es y cómo funciona una base de datos relacional. Además de eso el desafío de construir un sistema capaz de manejar creación de base de datos, gestionar tablas y columnas y realizar consultas sobre esas tablas y crear índices para acelerar las búsquedas y todo utilizando principios de estructuras de datos, programación orientada a objetos y principios de programación en C#.

Preliminar a la Descripción de la Solución

1.Inicio del proyecto

Antes de iniciar el proyecto el grupo discutió cuál podría ser el acercamiento o estrategia que se usaría para realizar el proyecto, se leyeron los requerimientos y se llegó al primer planteamiento de que el proyecto trataba de 4 módulos “independientes”, el Powershell, el Api Interface, el Query Processor y el Store Data Manager. El PowerShell cumpliendo el rol de cliente, el Api interface como intermediario entre servidor y cliente, el Query como el que procesa las validaciones y el Store Data Manager que se encarga de interactuar directamente con los archivos.

A partir de este planteamiento algo inmaduro sobre los 4 módulos, pero sin realmente conocer que hace que y como interactúan entre ellos se dividió el proyecto en 2, Una Persona trabajando en el Powershell y el Store Data Manager y otra personas trabajando en el Query Processor y el Api interface.

2. Problema en medio del proyecto

Debido a problemas en el desarrollo por el mal planeamiento se llego a la conclusión de que era complejo trabajar en módulos que dependían tanto de los demás sin estar trabajando en estos otros, además de esto hubieron problemas de tiempo y sincronización para trabajar en ciertas partes en ambos lados y por lo general los módulos iban desfasados unos de otros así que fue imposible conectarlos.

3.Reinicio

Luego del problema en el desarrollo del proyecto se tomó la decisión de empezar de cero pero con un enfoque diferente. Se iniciaria desde una base de datos muy sencilla y simple pero funcional y a partir de esta se irían agregando funciones,nuevas operaciones,nuevas entradas, pero todo poco a poco y ambos integrantes del grupo podrían trabajar en todo el código a la vez a la hora de implementar una nueva función. Y también se establecieron reglas como que solo se puede hacer un commit una vez esté asegurado que esa nueva operación funciona por completo y no genera errores en el resto del código.

Descripción de la Solución

1.Comprendimiento de la base de datos y la primer versión de TinySQL

El primer paso sería comprender que es una base de datos y cómo funciona al menos por encima, para ellos se visualizaron videos donde se explicaba el concepto básico de una base de datos, sus principales componentes y funcionamiento, enfocándose en la interacción cliente-servidor. De estos videos se aprendió sobre las bases necesarias sobre el proyecto como que los datos deberían almacenarse,recuperarse o manipularse, la forma en la que se escriben las sentencias SQL y cómo se manejan o operan, como que el INSERT mete una fila de datos o el DELETE eliminar una fila de datos.

Lo segundo fue comprender que era Tiny SQL,que hace y cómo lo hace,siendo este un ejemplo inicial muy incompleto de cómo debería estar estructurado el Proyecto,Se comprendio que era un esqueleto inicial o una guia. Pero con cosas como los sockets o la conexión servidor-cliente ya programada y definida lo cual ahorraría gran cantidad de tiempo.

2.Establecimiento del Plan inicial

Una vez comprendido el proyecto y que se requiere y con la experiencia de intento anterior se trazó el plan mencionado anteriormente en Reinicio. Donde se establece puntualmente que incorpora la primer versión y como. Además de esto se plantearon preguntas que habia que responder antes de iniciar a programar.

El documento del plan inicial incluía:

Mini Test de servidor SQL

Solo va a recibir 3 sentencias, INSERT, DELETE Y SELECT.

Por lo que ya va a tener hecha una “Base de datos” que va a ser una carpeta con un .txt en la que va a hacer las búsquedas el Store Data Manager.

El programa se compone únicamente de 4 Partes:

- **El Powershell** para enviar las consultas,recibirlas e imprimir en consola la tabla o fila consultada. El usuario escribe la consulta en SQL y el powershell la pasa a JSON para enviársela al Api

- **El Api interface:** Recibe la consulta en formato JSON y la pasa a SQL para procesarla,le envia la consulta en SQL al Query,

- **EL Query** solo verifica que este bien escrita y se la pasa al Store data manager(no hay validación de si la tabla existe)

- **El Store Data Manager**:Retorna el resultado de la consulta o la operación de insertar o borrar. Aun no se como devuelve el resultado si lo recibe directamente el api al hacer la cadena api-query-store. O el store se lo tiene que pasar directamente al api.

La entrada que se va a recibir siempre esta compuesta de un ID que es int, y un nombre de comida que es un varchar y un día que es igual varchar

Ejemplo: 1 Hamburguesa Martes

Consultas antes de empezar el proyecto:

-Debería tener un parser? Es necesario para consultas tan simples o se puede evitar en este caso?

-Como el Api interface debería recibir el resultado del store data manager.

-Con insert,select y delete es suficiente para mostrar el funcionamiento básico de una base de datos sin complicar mucho el proyecto, o hay algúna operación extremadamente esencial que se debería agregar?

3.Inicio del desarrollo el Query Processor:

Se decidió empezar programando el Query Processor, Esto debido a que es el corazón del proceso, se supuso que el API únicamente va a llamar al query pasando como entrada una sentencia en formato SQL.

¿Entonces qué hace el Query Processor?

Este lee la sentencia y en base a esta se asegura que este bien escrita la consulta y a partir de ahí valida que la estructura sea correcta,los datos recibidos sean los correctos y las tablas y base de datos existan, una vez validada llama a la operación requerida donde esta maneja una lógica a parte para realizar operaciones como parsear la consulta antes de enviarla al store para facilitar el trabajo del store y que este solo haga las operaciones con los datos en las bases de datos.

Primera parte el Query:

Se conservó la estructura de la base inicial de manejar una clase principal del query que solo se base en un método execute, se define una variable currentDatabase que se define con el SET y cada metodo se llama segun con que empiece la sentencia:

```
public class SQLQueryProcessor
{
    // Variable para almacenar la base de datos actual
    private static string currentDatabase = null;

    1 referencia
    public static (OperationStatus, List<string>) Execute(string sentence)
    {
        // SET DATABASE
        if (sentence.StartsWith("SET DATABASE", StringComparison.OrdinalIgnoreCase))
        {
            var dbName = sentence.Split("DATABASE")[1]?.Trim(' ', ';');
            if (string.IsNullOrEmpty(dbName))
            {
                Console.WriteLine("Error: No se encontró el nombre de la base de datos.");
                return (OperationStatus.Error, null);
            }

            // Actualizamos la base de datos actual
            currentDatabase = dbName;
            Console.WriteLine($"Base de datos actualizada: {currentDatabase}");
            return (OperationStatus.Success, null);
        }

        // CREATE DATABASE (esta parte la agregamos)
        if (sentence.StartsWith("CREATE DATABASE", StringComparison.OrdinalIgnoreCase))
        {
            var dbName = sentence.Split("DATABASE")[1]?.Trim(' ', ';');
            if (string.IsNullOrEmpty(dbName))
            {
                Console.WriteLine("Error: No se encontró el nombre de la base de datos.");
                return (OperationStatus.Error, null);
            }

            // Llamamos a la operación CreateDatabase para crear la base de datos
            var status = new CreateDatabase().Execute(dbName); // Aquí solo pasamos el nombre de la base de datos
            return (status, null);
        }
```

Aca se muestra la estructura general de el QueryProcessor.

Segunda parte Operaciones:

Cada operación inicia siendo llamada desde el método execute de la clase Query donde este llama a la operacion respectiva que luego de procesar el dato recibido llama al metodo del stored data manager respectivo que realiza la operacion como tal para que atravez de esa cadena Api-Query-Store el Api reciba el resultado de la consulta.

Insert:

La operación Insert tiene la función de procesar una sentencia de inserción de datos SQL INSERT y validarla antes de que los datos sean agregados a la tabla correspondiente en la base de datos. Esta debe recibir 3 parámetros que serán la base de datos, el nombre de la tabla y la sentencia de inserción completa que seria el sentence El formato esperado sería algo como:

INSERT INTO Estudiante VALUES (1, "Isaac", "Ramirez", "Herrera", "2000-01-01 01:02:00").

A partir de esta entrada la operación insert extrae los valores clave de la sentencia buscando la palabra clave VALUES y que hay entre los paréntesis, luego se limpian los caracteres innecesarios como los espacios,comas y comillas y se separan los valores para obtener cada datos que se va a insertar.

Luego se valida que el número de valores extraídos sea exactamente 5 y si el número no es correcto devuelve el error.

Luego se tiene una parte de conversión de datos donde se definen las entradas según como se definieron en la tabla,siendo el primer valor entero,los siguientes varchar y el último DateTime.

Por último ya con todos los valores establecidos se llama el método InsertIntoTable de la clase Store en el Store Data manager para que realice la operación,

```
// Extrae la sección de valores (todo lo que está entre los paréntesis)
var valuesSection = sentence.Split("VALUES")[1]?.Trim(' ', ';', '(', ')');
if (string.IsNullOrEmpty(valuesSection))
{
    Console.WriteLine("Error: No se encontraron los valores.");
    return OperationStatus.Error;
}

// Divide los valores, asegurando que se eliminen las comillas simples o dobles alrededor de las cadenas y la fecha
var values = valuesSection.Split(',')
                          .Select(v => v.Trim(' ', '\'', '"'))  // Eliminar comillas adicionales
                          .ToList();

// Valida que se reciban 5 valores (ID, varchar1, varchar2, varchar3, datetime)
if (values.Count != 5)
{
    Console.WriteLine("Error: Se esperan 5 valores (ID, 3 cadenas de texto y una fecha).");
    return OperationStatus.Error;
}

try
{
    // Procesa cada valor
    int id = int.Parse(values[0]); // ID
    var c1 = values[1];            // Primer varchar
    var c2 = values[2];            // Segundo varchar
    var c3 = values[3];            // Tercer varchar

    // Parsea el valor datetime
    DateTime dateValue = DateTime.Parse(values[4]);

    // Llama al Store Data Manager para insertar en la tabla
    return Store.GetInstance().InsertIntoTable(databaseName, tableName, id, c1, c2, c3, dateValue);
}
catch (FormatException ex)
```

A partir de acá entra en juego el Store Data Manager con la operación InsertIntoTable

Que se encargará de insertar los valores dentro de una tabla específica en una base de datos, se dan 7 entradas siendo el nombre de la tabla y la base de datos,el id, los 3 varchar y el datetime.

Se creó un método para obtener la ubicación del archivo de la tabla para saber donde se debe almacenar los datos.

Luego de eso se formatea el registro en forma de cadena de texto con los cinco valores separados por comas, se abre el archivo txt correspondiente y con StreamWriter en modo adjuntar se agregan los nuevos datos al final del archivo sin sobreescribir el contenido existente.

```
// Método para insertar un nuevo registro en una tabla dentro de una base de datos
1 referencia
public OperationStatus InsertIntoTable(string databaseName, string tableName, int id, string c1, string c2, string c3, DateTime dateValue)
{
    try
    {
        string tableFilePath = GetTableFilePath(databaseName, tableName);
        if (tableFilePath == null) return OperationStatus.TableNotFound;

        // Formatear el registro con los cinco valores
        string record = $"{id},{c1},{c2},{c3},{dateValue:yyyy-MM-dd HH:mm:ss}";

        // Insertar el registro en el archivo de texto
        using (StreamWriter writer = new StreamWriter(tableFilePath, append: true))
        {
            writer.WriteLine(record);
        }

        Console.WriteLine($"Registro insertado correctamente en la tabla {tableName} de la base de datos {databaseName}.");
        return OperationStatus.Success;
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error al insertar: {ex.Message}");
        return OperationStatus.Error;
    }
}
```

Delete:

La operación Delete es una Clase en la parte de operaciones que elimina un registro de una tabla en una base de datos específica, se basa en la condición especifica que donde sea ese ID se borra esa fila.

Se recibe una sentencia tipo:

DELETE FROM tableName WHERE ID = 1;

Que indica que se quiere eliminar donde ID = 1 en la tabla.

Ahora se extrae la condición utilizando la cláusula WHERE para separar la parte ID = X, y una vez extraída esta condición se toma la parte que sigue ID = utilizando Split,y ya con esto tendríamos el valor de la condición que debería ser un entero y con este valor se llama al store data manager.

```
internal class Delete
{
    1 referencia
    public OperationStatus Execute(string databaseName,string tableName, string sentence)
    {
        // Ejemplo de sentencia: "DELETE FROM tableName WHERE ID = 1;"

        // Extrae la condición (ID = X)
        var whereClause = sentence.Split("WHERE")[1]?.Trim(' ', ';');
        if (string.IsNullOrEmpty(whereClause))
        {
            Console.WriteLine("Error: No se encontró la condición para eliminar.");
            return OperationStatus.Error;
        }

        // Extrae la cndicion del ID
        var idString = whereClause.Split('=')[1].Trim();
        if (!int.TryParse(idString, out int id))
        {
            Console.WriteLine("Error: El ID especificado no es válido.");
            return OperationStatus.Error;
        }

        //Llama al Store Data Manager para realizar el borrado
        return Store.GetInstance().DeleteFromTable(databaseName, tableName, id);
    }
}
```

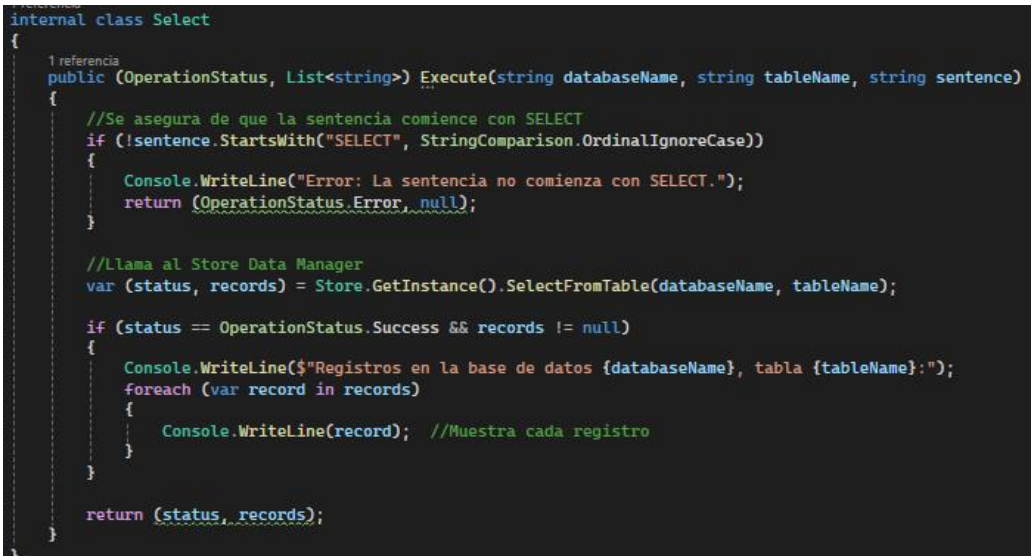
El Método delete del store data manager es el encargado de eliminar ese registro,toma ese ID y primero verifica en que tabla y base de datos se está trabajando.
Se crea un archivo temporal donde se escriben los datos que no pertenecen a ese ID que se quiere eliminar,luego abre el archivo original en moto lectura y el archivo temporal en modo escritura, donde se lee cada registro del archivo de la tabla y donde el ID coincida simplemente no se escribe en el archivo temporal.

Al terminar el archivo temporal reemplaza al original.

```
try
{
    string tableFilePath = GetTableFilePath(databaseName, tableName);
    if (tableFilePath == null) return OperationStatus.TableNotFound;

    var tempFile = Path.GetTempFileName();
    bool found = false;

    using (var reader = new StreamReader(tableFilePath))
    using (var writer = new StreamWriter(tempFile))
    {
        string? line;
        while ((line = reader.ReadLine()) != null)
        {
            var parts = line.Split(',');

            if (int.TryParse(parts[0].Trim(), out int recordId) && recordId == id)
            {
                found = true;
            }
            else
            {
                writer.WriteLine(line);
            }
        }
    }

    File.Delete(tableFilePath);
    File.Move(tempFile, tableFilePath);

    return found ? OperationStatus.Success : OperationStatus.Error;
}
```

Select:

La clase select ejecuta una operación SELECT sobre una tabla en una base de datos especifica, el método execute en la operación verifica que sea una sentencia SQL SELECT y si es simplemente llama al Store pasando el nombre de la tabla y base de datos.

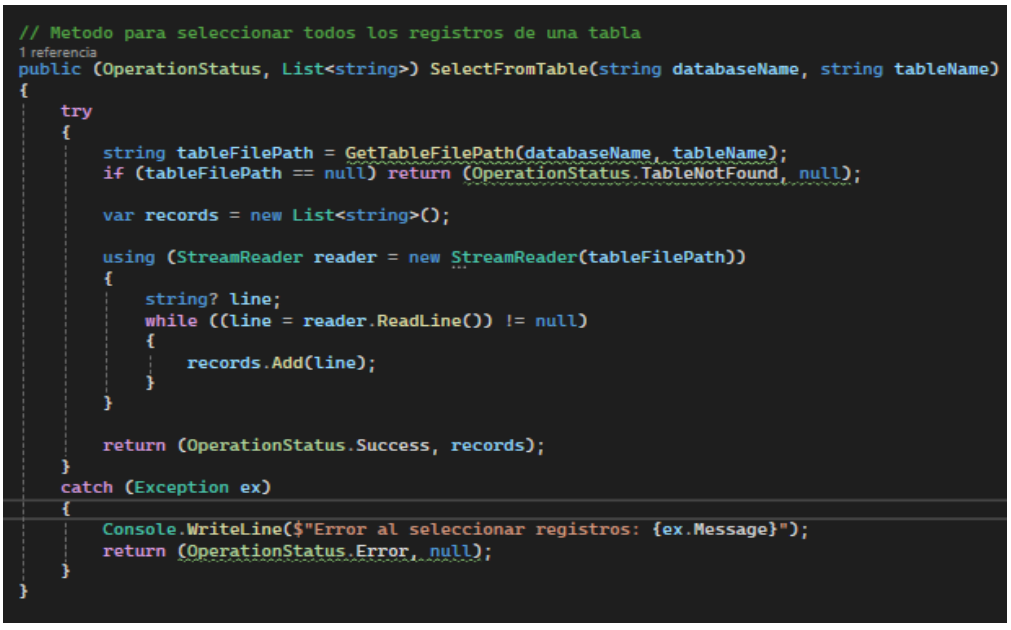


```
internal class Select
{
    1 referencia
    public (OperationStatus, List<string>) Execute(string databaseName, string tableName, string sentence)
    {
        //Se asegura de que la sentencia comience con SELECT
        if (!sentence.StartsWith("SELECT", StringComparison.OrdinalIgnoreCase))
        {
            Console.WriteLine("Error: La sentencia no comienza con SELECT.");
            return (OperationStatus.Error, null);
        }

        //Llama al Store Data Manager
        var (status, records) = Store.GetInstance().SelectFromTable(databaseName, tableName);

        if (status == OperationStatus.Success && records != null)
        {
            Console.WriteLine($"Registros en la base de datos {databaseName}, tabla {tableName}:");
            foreach (var record in records)
            {
                Console.WriteLine(record);  //Muestra cada registro
            }
        }

        return (status, records);
    }
}
```

El método SelectFromTable del store es el encargado de devolver todos los registros de una tabla en una base de datos especifica.

Utiliza un bucle para leer cada linea del archivo hasta que no haya mas datos. Cada linea representa un registro que se añade a una lista “records”

El método retorna una lista de los registros obtenidos



```
// Metodo para seleccionar todos los registros de una tabla
1 referencia
public (OperationStatus, List<string>) SelectFromTable(string databaseName, string tableName)
{
    try
    {
        string tableFilePath = GetTableFilePath(databaseName, tableName);
        if (tableFilePath == null) return (OperationStatus.TableNotFound, null);

        var records = new List<string>();

        using (StreamReader reader = new StreamReader(tableFilePath))
        {
            string? line;
            while ((line = reader.ReadLine()) != null)
            {
                records.Add(line);
            }
        }

        return (OperationStatus.Success, records);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error al seleccionar registros: {ex.Message}");
        return (OperationStatus.Error, null);
    }
}
```

4.Desarrollo del Api Interface

Luego de desarrollar el Query Processor junto con las operaciones y el Store Data manager, se procedió a desarrollar el Api Interface el cual es el que hace la función de servidor.

Recibe las solicitudes en formato JSON del cliente a través de un socket TCP, para luego procesar y enviar las respuestas JSON.

Se define primeramente un serverEndPoint para que el servidor escuche en una dirección y puerto.

El método start pone al servidor en bucle infinito esperando nuevas conexiones y cuando una conexión es aceptada intenta leer el mensaje entrante con GetMessage,convertir el mensaje a un objeto Request que sería pasar el JSON a SQL y procesar la solicitud llamando a ProcessRequest que seria el metodo que llama al Query dando la sentencia como parámetro.

Por último envía la respuesta de vuelta al cliente con SendResponse.

- **El metodo GetMessage**:
  Utiliza NetworkStrem con lo que lee el mensaje recibido desde el cliente
- **El metodo ConvertToRequestObject** "traduce" el mensaje recibido en JSON a SQL que seria el objeto Request.
- **El metodo ProcessRequest** es la parte central del api,es el que envia la solicitud al SQLQueryProcessor que ejecuta la consulta SQL y devuelve un estado y una lista de resultados. Dependiendo del resultado de la operación, se crea un objeto Response con la respuesta apropiada
- **Por ultimo estaria el metodo SendResponse** que toma la respuesta obtenida la serializa a JSON y se envia al cliente mediante NetworkStream.

```
1 referencia
private static Response ProcessRequest(Request requestObject)
{
    try
    {
        var sqlSentence = requestObject.RequestBody;
        var (status, resultList) = SQLQueryProcessor.Execute(sqlSentence);

        return new Response
        {
            Status = status,
            Request = requestObject,
            ResponseBody = status == OperationStatus.Success
                ? (resultList != null ? string.Join(", ", resultList) : "Operation successful")
                : "Operation failed"
        };
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error processing request: {ex.Message}");
        return new Response
        {
            Status = OperationStatus.Error,
            Request = requestObject,  //
            ResponseBody = "Error processing the request."
        };
    }
}
```

Diagrama UML: