PROYECTO II TinySQLDb

Adrian P. Ramírez & Jose Pablo A. Mata

Ingeniería en Computadores

Algoritmos y Estructuras de Datos I (CE 1103)

Leonardo Araya Martinez

Cartago, Costa Rica

2024

Tabla de contenido

1. Introducción

El proyecto TinySQLDb tiene como objetivo diseñar un motor de base de datos relacional sencillo que nos permita comprender los principios fundamentales detrás de los motores de bases de datos. El proyecto fue desarrollado utilizando el lenguaje de programación **C#** y está diseñado para manejar operaciones básicas como **crear tablas**, **insertar datos**, **buscar información** y **actualizar datos**. El sistema se desarrolla con el fin de simular algunas funcionalidades básicas de motores de bases de datos comerciales, tales como MySQL, lo cual, permite entender cómo se estructuran y procesan consultas SQL a nivel interno.

2. Descripción del Problema

El problema que este proyecto busca resolver es la creación de un sistema que permite ejecutar consultas SQL, y almacenar los resultados en un formato organizado. En otras palabras, necesitábamos un sistema en el que los usuarios puedan realizar operaciones como **crear bases de datos**, **crear tablas**, **insertar información en las tablas**, **buscar datos específicos** y **actualizar datos**.

La principal dificultad era hacer que estas operaciones fueran eficientes y que el sistema pudiera realizar las búsquedas rápidamente, sin perder tiempo, incluso cuando la cantidad de datos almacenados aumentará. Para eso, había que implementar un sistema que pudiera **organizar** bien los datos y **acelerar las búsquedas**.

3. Descripción de la Solución

3.1 Cliente en PowerShell

Para permitir que los usuarios puedan escribir consultas SQL y ver los resultados, implementamos un **cliente** en **PowerShell**. PowerShell es una herramienta de línea de comandos que permite ejecutar comandos en la computadora. En este caso, el cliente lee las consultas SQL que los usuarios ingresan y las envía al servidor, que es la parte del sistema que realiza las operaciones en la base de datos.

El cliente de PowerShell toma el archivo donde están escritas las consultas SQL, se conecta al servidor usando una **IP** y un **puerto**, y envía las consultas para que el servidor las ejecute. Luego, muestra los resultados al usuario en formato de tabla, que es fácil de leer.

Alternativa considerada: Se consideró crear una interfaz gráfica, pero PowerShell fue elegido por ser más rápido de implementar y fácil de usar para este tipo de proyecto.

Limitaciones: Al ser un sistema de línea de comandos, no es tan visual como una interfaz gráfica, lo que puede hacer que el uso sea menos intuitivo para algunos usuarios.

Problemas encontrados: Tuvimos algunos problemas con la lectura de los archivos de consultas, especialmente al dividir las sentencias SQL correctamente cuando estaban mal formateadas. También fue necesario ajustar la manera en que se muestra el resultado para hacerlo más legible.

3.2 Procesamiento de Consultas

El procesamiento de consultas es la parte más importante del proyecto. Aquí es donde el servidor recibe la consulta SQL, verifica que esté bien escrita, y luego ejecuta la operación. Si todo está bien, el servidor devuelve el resultado al cliente.

Se diseñó un **Query Processor** como procesador de consultas, que es un conjunto de funciones que revisan si la consulta está bien escrita, es decir, que las tablas y columnas existen por ejemplo, y que luego ejecuta la operación en los datos almacenados. Este procesador también es responsable de optimizar las búsquedas para que sean rápidas.

Alternativa considerada: Se puede usar una biblioteca de SQL ya existente que manejara las consultas, pero la idea del proyecto es crear un propio procesador para aprender cómo funciona internamente un sistema de bases de datos.

Limitaciones: El sistema solo acepta un conjunto limitado de comandos SQL, por lo que no tiene todas las funciones avanzadas que podrías encontrar en sistemas como MySQL.

Problemas encontrados: Al principio, validar las consultas fue complicado porque algunas no estaban bien estructuradas, y eso hacía que el sistema fallara. También se tuvieron que hacer muchos ajustes para optimizar las búsquedas y que fueran rápidas. Algunas funcionaron pero otras tuvieron que quedarse como estaban porque al optimizar líneas de código, se dañaba el resto.

3.3 Creación de bases de datos

Se implementó la operación **`CREATE DATABASE`** con la que se crean las bases de datos de manera eficiente. El proceso consiste en recibir una consulta SQL, verificar si la base de datos ya existe, y en caso contrario, crear una nueva entrada en el **System Catalog**.

Alternativas consideradas: Se pensó almacenar las bases de datos en memoria, pero se optó por usar el sistema de archivos para garantizar la persistencia.

Limitaciones: El sistema no maneja bases de datos distribuidas, por lo que la escalabilidad es limitada.

Problemas encontrados: Durante el desarrollo, no se encontraban las rutas de acceso a los archivos donde se guardan las bases, debido a que las carpetas se guardaban en OneDrive por eso se decidió para las carpetas del proyecto al Disco duro y se logró solucionar el problema.

3.4 Creación de tablas

El comando **`CREATE TABLE`** fue implementado para permitir la creación de tablas dentro de una base de datos seleccionada. La tabla es registrada en el System Catalog y las columnas definidas son almacenadas en un archivo independiente.

Alternativas consideradas: Se analizó el uso de bibliotecas externas para gestionar las tablas, pero la idea es construir una solución interna para tener mayor control sobre el proceso.

Limitaciones: La creación de tablas con un gran número de columnas puede afectar el rendimiento.

Problemas encontrados: Se intento corregir las afectaciones que puede provocar crear muchas columnas tratando de optimizar el código pero al hacerlo produjo errores en los que ya habíamos logrado, por tanto, por falta de tiempo mejor decidimos devolver el código a como estaba antes sin intentos de optimizaciones.

3.5 Gestión de índices

Los **índices** en bases de datos nos permiten realizar búsquedas rápidas en tablas que contienen grandes cantidades de datos. En este proyecto, implementamos índices basados en **B-Trees**, una estructura de datos que es eficiente en este caso, y aseguran de que los datos siempre estén bien organizados, de manera que, sin importar cuántos elementos agreguemos, siempre podamos encontrar, insertar o borrar un elemento en **pocos pasos**. Estos pasos son proporcionales al "logaritmo" del número de datos existentes ya sea si es

una tabla como en nuestro proyecto, si existen muchos datos, los pasos adicionales que necesitas no crecen muy rápido.

Implementación de índices con B-Trees

Un **B-Tree** es un árbol balanceado que permite organizar los datos de manera que se minimicen las operaciones de búsqueda, inserción y eliminación. A diferencia de otros tipos de árboles, un B-Tree está diseñado para ser eficiente con grandes cantidades de datos, asegurando que todas las hojas estén al mismo nivel, lo que garantiza búsquedas rápidas.

En nuestro proyecto, la operación **CREATE INDEX** permite crear un índice sobre una columna de una tabla. El B-Tree se encarga de organizar los valores de esta columna, permitiendo que las búsquedas en dicha columna se realicen en tiempo *O (Log n)*.

Alternativas consideradas

Durante el desarrollo, se evaluaron alternativas como el uso de **árboles binarios (BST)**, pero se optó por los **B-Trees** debido a sus características mas avanzadas en manejo de cantidades mayores de datos.

Limitaciones

Una de las principales limitaciones de nuestro sistema es que solo se permite un índice por columna. Esto puede ser un problema si se quiere realizar consultas eficientes en varias columnas de una misma tabla.

Problemas encontrados

Implementar los **B-Trees** presentó problemas en cuanto a ubicar los datos en las tablas creadas anteriormente por el script, ya que al parecer no estaba implementando correctamente el comando WHERE

3.6 Almacenamiento de Datos

El **almacenamiento de datos** es cómo el sistema guarda la información en el disco. En lugar de usar un sistema de base de datos existente, como MySQL, creamos nuestro propio sistema que guarda los datos en archivos binarios en la computadora. Cada tabla de la base de datos se guarda en un archivo separado.

Implementación:

Se diseñó un **Stored Data Manager** que se encarga de leer y escribir los datos en el disco. Este gestor también es responsable de asegurarse de que los índices estén siempre actualizados.

Alternativa considerada:

Podríamos haber utilizado una base de datos en memoria (es decir, que guarda los datos solo mientras el programa está funcionando), pero eso haría que los datos se pierdan al cerrar el programa. Por eso, decidimos que el sistema guarde los datos en el disco para que se mantengan guardados y borrarlos manualmente.

Limitaciones:

El sistema no está diseñado para manejar grandes volúmenes de datos de manera eficiente, porque el manejo de archivos puede ser lento si hay mucha información.

Problemas encontrados:

Hubo problemas de rendimiento al manejar tablas muy grandes, ya que el sistema de almacenamiento como que no puede leer y escribir archivos de manera rápida.

3.7 Operaciones CRUD (Insertar, Seleccionar, Eliminar, Actualizar)

En este proyecto, implementamos las operaciones básicas que cualquier base de datos debe tener, conocidas como **CRUD**: **Crear (Insert)**, **Leer (Select)**, **Actualizar (Update)** y **Eliminar (Delete)**. Estas son las operaciones que nos permiten agregar, modificar, eliminar o consultar datos en una tabla.

3.7.1 Operación INSERT

La operación **`INSERT INTO`** se utiliza para agregar nuevos datos a una tabla. Es como si estuviéramos llenando una fila en una hoja de cálculo. Primero verificamos que la tabla exista y luego nos aseguramos de que los datos que queremos agregar coincidan con los tipos de datos de las columnas (números, texto, fechas, etc.).

Problema encontrado: Al principio, tuvimos problemas porque no estábamos validando bien los tipos de datos, como que no insertaba bien los datos. Lo arreglamos asegurándonos de que los datos que insertamos siempre coincidan con lo especificado en el script.

3.7.2 Operación SELECT

`SELECT` es probablemente la operación más importante porque es la que nos permite consultar los datos almacenados en la base de datos. Con ella, podemos filtrar resultados usando **`WHERE`** para ver solo los datos que necesitamos y también podemos ordenarlos.

Problema encontrado: Tuvimos problemas con la comparación de datos, especialmente con cadenas de texto. Si no estaban bien formateadas (por ejemplo, mayúsculas y minúsculas), la operación no funcionaba como esperábamos. La solución fue convertir todas las cadenas a minúsculas antes de compararlas, para evitar errores.

3.7.3 Operación DELETE

DELETE nos permite eliminar datos de una tabla. Si no especificamos una condición con **WHERE**, borra todo. Si usamos **WHERE**, solo borra las filas que coinciden con la condición.

Problema encontrado: Al principio, buscar las filas que debíamos eliminar era muy lento cuando había muchas filas. Lo solucionamos creando **índices** en las columnas para que las búsquedas fueran más rápidas, lo que ayudó mucho a mejorar el rendimiento.

3.7.4 Operación UPDATE

UPDATE sirve para modificar los datos de una fila existente en la tabla. Esta operación es muy útil cuando queremos cambiar algo que ya hemos insertado, por ejemplo, si un dato es incorrecto. Se especifica qué columna queremos actualizar y qué condición debe cumplirse con **WHERE** para que solo se actualicen elementos específicos de ciertas filas.

Problema encontrado: Tuvimos un problema con **UPDATE** porque no siempre encontrábamos las filas que queríamos modificar. Esto pasaba porque las listas donde se guardaban los datos no comparaban bien los valores. Se trato solucionar limpiando los datos, como quitando espacios, ponerlos en minúsculas antes de buscar las filas, pero aun así no lograba encontrar los datos o actualizarlos.

4. Diseño General