ITCR

Ingeniería en Computadores

Algoritmos y Estructuras de Datos 1

Documentación Proyecto 2

Estudiantes:
Esteban Campos Abarca - 2022207705
Jafet Díaz Morales - 2023053249

II Semestre 2024

Tabla de Contenidos

Introducción

En este documento se presenta la implementación con paradigma de orientación a objetos de un motor de base de datos sencillo con C# con el fin de familiarizarse con el uso de una base de datos relacional y con el objetivo de hacer un buen uso de estructuras de datos jerárquicas como lo son los árboles. Además, el proyecto tiene el fin de fomentar la creatividad y las buenas prácticas de programación (documentar, uso de UML, etc.)

Breve Descripción del Problema

El proyecto consiste en implementar un sistema administrador de bases de datos sencillo el cual tiene los siguientes componentes:

1. Cliente
2. Servidor

a. Interfaz API
b. Query processing
c. Stored Data Manager

En general el comportamiento es el siguiente:

1. El cliente envía el requisito mediante un socket al API Interface.
2. API interface procesa el mensaje y envía la sentencia SQL al query processor quitando cualquier elemento específico del protocolo de comunicación.
3. Query processor valida que la sentencia sea correcta utilizando el system catalog (por ejemplo, que la base de datos exista, que la tabla exista, que las columnas sean válidas, entre otras) y coordina su ejecución apoyándose en Stored Data Manager.
4. Stored Database Manager accede a los archivos que contienen los datos y realiza la consulta como tal.

Descripción de la solución

Se procederá a explicar la solución en referente a cada requerimiento.

Requerimiento 0:

El cliente se implementa como un módulo de Powershell 7 que se carga en Powershell y que permite ingresar comandos para realizar operaciones sobre la base de datos. Existe una función llamada Execute-MyQuery con los siguientes parámetros:

- QueryFile: Indica un path donde se encuentra el archivo con sentencias SQL por ejecutar.
- Port: puerto en el que el interfaz API escucha
- IP: dirección IP en el que el interfaz API escucha

Por ejemplo: Execute-MyQuery –Query .\Script.tinysql -Port 8000 –IP 10.0.0.2

Los resultados se muestran en la terminal en formato de tabla.

Requerimiento 1:

Existe un solo archivo .json (SystemCatalog.json) que contiene la información de toda la base de datos. Al iniciar el programa SystemCatalog.json es leído y convertido en un objeto DatabaseManger llamado dm. La clase DatabaseManger tiene de atributo solo una List<Database>.

Database es una clase que tiene de atributos:

string name
List<Table>

Table (hay dos clases que se llaman Table. [están Metadata.Table y DataStructures.Tables.Table, la que usa SystemCatalog es la de Metadata]) tiene atributos:

string name
string PK (primary key)
string Index (esto debe ser igual al nombre de una de las columnas de List<Column> columns)
List<Column> columns

Column es una clase que tiene:

string name (e. g. ID)
string type (e. g. INTEGER)

Cuando una tabla o base de datos se crea o se actualiza, se modifica dm y se usa la función su SystemCatalog.SaveChanges() para reflejar ese cambio en el JSON.

Requerimiento 2:

Para las queries o sentencias SQL se usa la clase estática QueryProcessing. QueryProcessing tiene una función public static string Execute(string query) que la recibe y con base en eso llama otra función a la que le pasa la query y que también devuelve un string. Execute devuelve ese string.

Todas las funciones que llama QueryProcessing.Execute(string query) descomponen la query original (del inicio hasta el punto y coma) en partes.

Para CREATE DATABASE <database name> QueryProcessing.CreateDatabase(string query) hace dos cosas:

(modifica el JSON): que dm (el objeto global DatabaseManager) añada un objeto Database a su List<Database> databases y que guarde el resultado
(crea una carpeta): crea la carpeta con el nombre <database name> dentro de la carpeta SystemCatalog.

NOTA: existe una carpeta SystemCatalog en la que está toda la información de las bases de datos. Abriendo la carpeta, lo primero que se ve son carpetas con los nombres de las databases y el archivo SystemCatalog.json.

Requerimiento 3:

Para SET DATABASE <database name> existe una variable global en Globals (que es en donde existe el objeto dm también) que se llama string set_database.

Cuando una query que inicia por SET DATABASE llega a QueryProcessing.Execute(string query) se llama QueryProcessing.SetDatabase(string query) que simplemente cambia ese string global set_database por el nombre de la database <database name>

Requerimiento 4:

Para CREATE TABLE (la sintaxis incluye declaración de la llave primaria) igual que antes pasa por query processing y se redirige a una función QueryProcessing.CreateTable(string query). Esta función separa cada parte de la query y luego con eso instancia un objeto tabla (DataStructures.Tables.Table) que tiene varias funciones que sirven para los requerimientos 007, 008, 009 y 010.

clase DataStructures.Tables.Table:
atributos:
List<string> cols
List<string> column_types
string primary_key
string table_name

Es importante aclarar que el objeto tabla se puede crear con dos constructores, uno para obtener una tabla nueva a partir de la información de Create Table; y otro, a partir de un archivo guardado en la base de datos. Esta segunda opción es la que se usa para tablas preexistentes y que no estén indexadas.

Requerimiento 5:

DROP TABLE <table name>. Igual que antes pasa por query processor y luego llama a una función que se llama DropTable(string query) que borra el archivo de texto de la tabla usando el Globals.datapath, Globals.set_database y <table name>. Además borra la tabla del System Catalog a través de Globals.dm.

Requerimiento 6:

Existe la variable List<Index> indexes en Globals que contiene objetos tipo Index. Estos representan una tabla indexada mediante un nombre de base de datos, el nombre de la columna por la que se indexa (que debe ser la llave primaria para asegurar evitar entradas repetidas en el índice) y un objeto de la clase TableTree. Este último es un árbol binario de búsqueda cuyos nodos son filas de tabla y que ordena sus nodos según su llave primaria. Cuando una sentencia requiere de una tabla, primero se busca si está existe en Globals como índice.

Requerimiento 7:

SELECT <columns> FROM <table name>. Similarmente, existe una función Select en QueryProcessor que descompone la query en strings. Una lista de strings de las columnas deseadas, el WHERE, y el ORDER BY. Estos los usa en la función de la clase tabla, GetSubtable(List<string> columns, string where, string order_by) que genera un objeto tabla de menor dimensión según se especifica. En este objeto Tabla se llama la función ShowString() para devolver el contenido de la tabla como un string ordenado y retornarlo fuera de Select() y del QueryProcessing.

Requerimiento 8:

UPDATE <column> FROM <table name>. Esta función se maneja mediante la función Update de la clase QueryProcessing. Descompone la query y por cada repetición de la forma “SET <column name> = <value>”, llama a la función Update de la clase tabla. Seguidamente retorna la tabla como string.

Requerimiento 9:

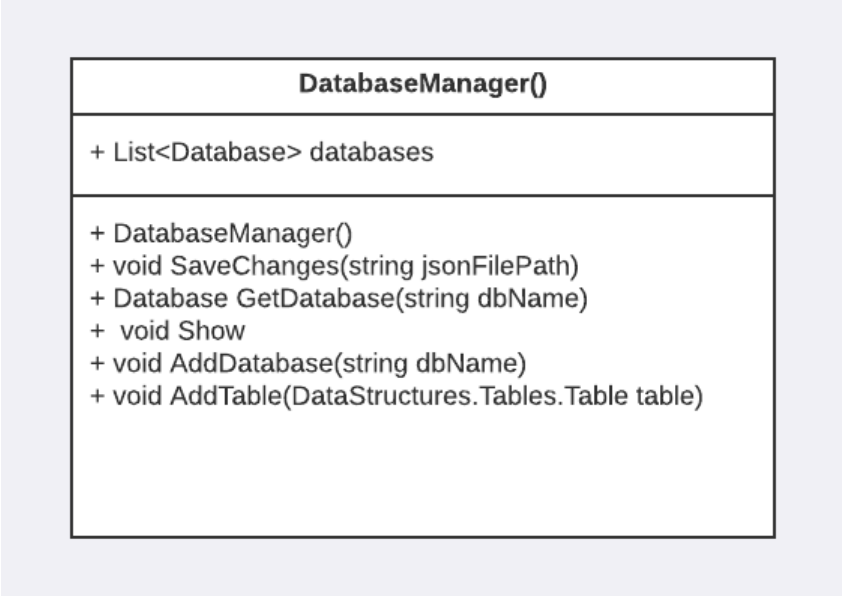
DELETE FROM <table name> WHERE. Existen las funciones Delete(string query) y Delete(string where) tanto en el QueryProcessor como en la clase tabla. La primera extrae el where y consigue la tabla correcta; la segunda ejecuta el comando y posteriormente se devuelve el string de la tabla modificada.

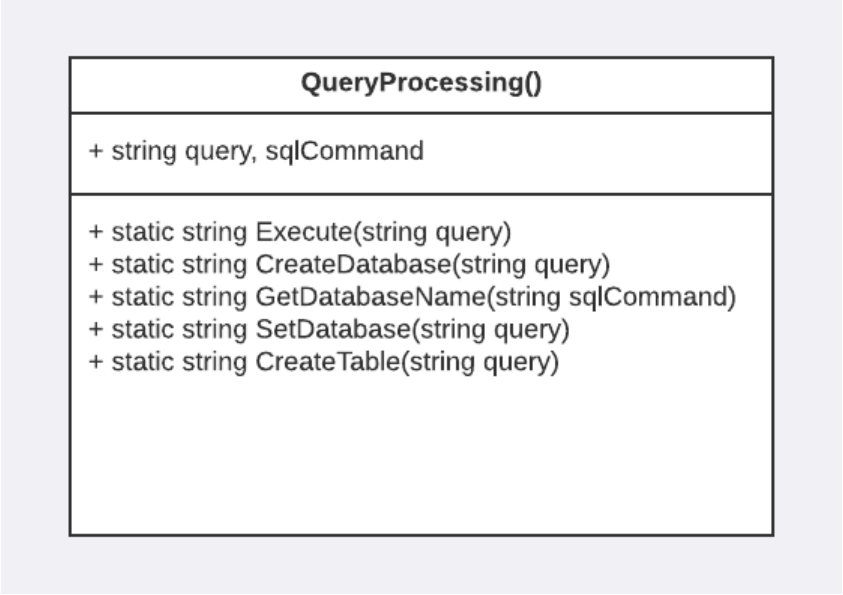
Requerimiento 10:

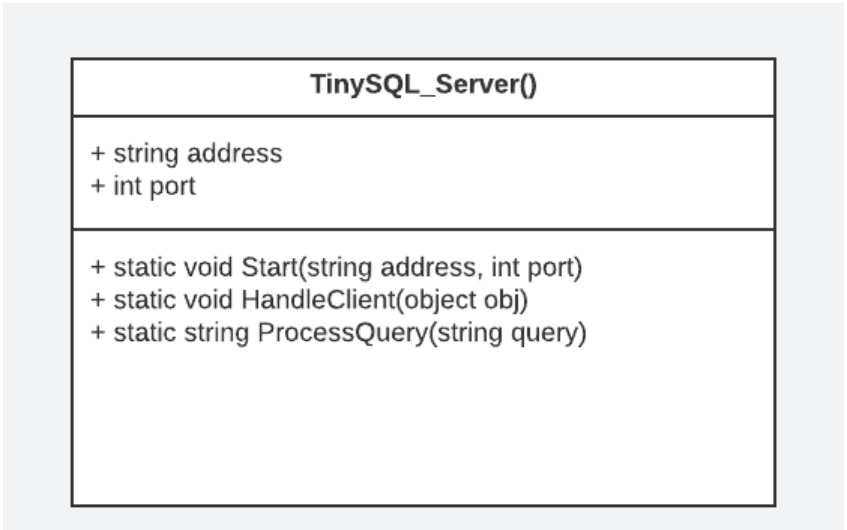
INSERT INTO <table name> (<value1>, <value2>, …, <value n>,). Nuevamente, existe la función InsertInto(string query) en el QueryProcessor. Separando los valores a insertar, InsertInto() elige la tabla buscada en los archivos e inserta una nueva fila con los valores en forma de string, seguidamente genera la tabla del archivo para que procese los datos en los tipos correctos y con esta genera el resultado final en string.

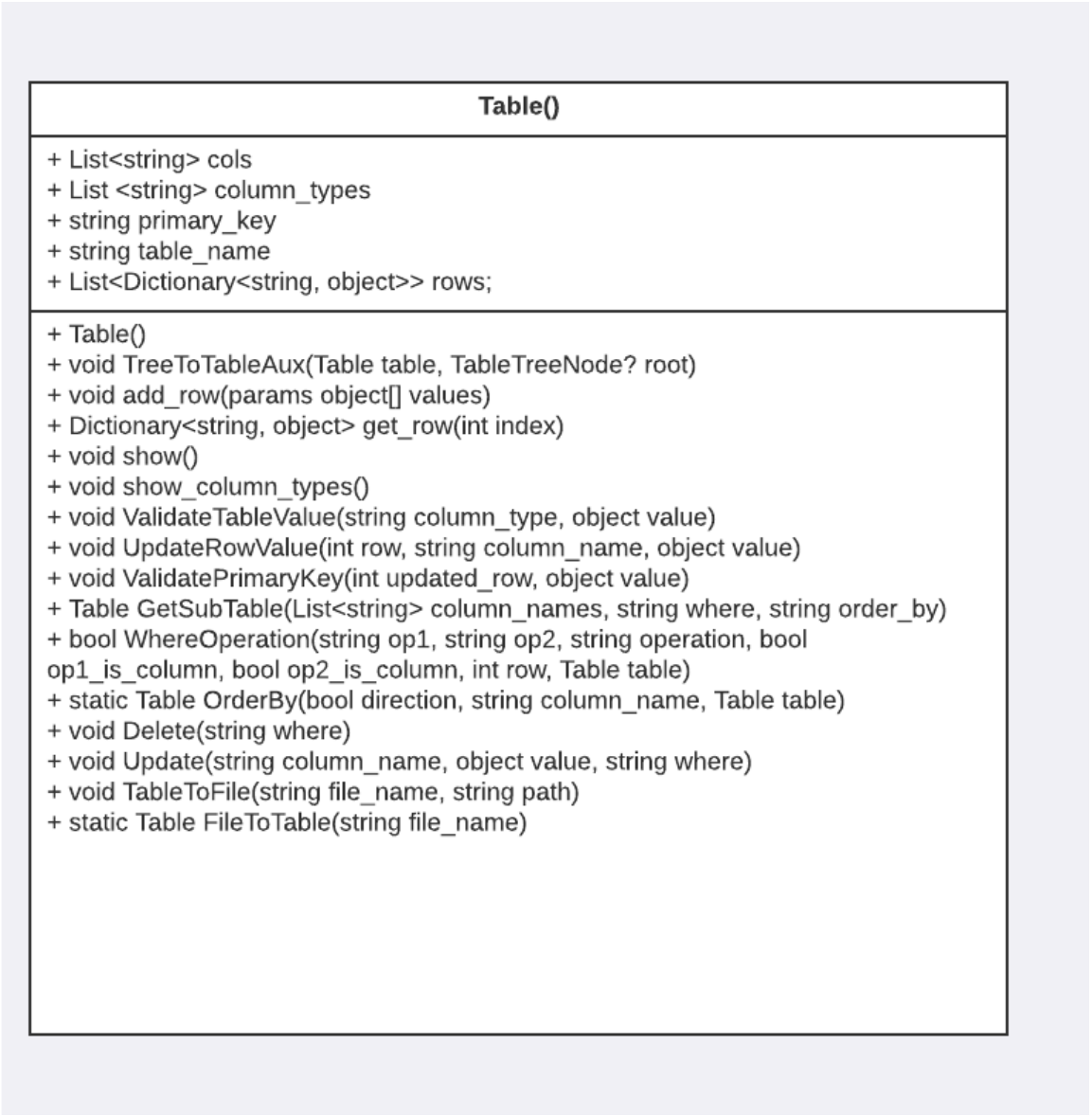
Diagramas UML

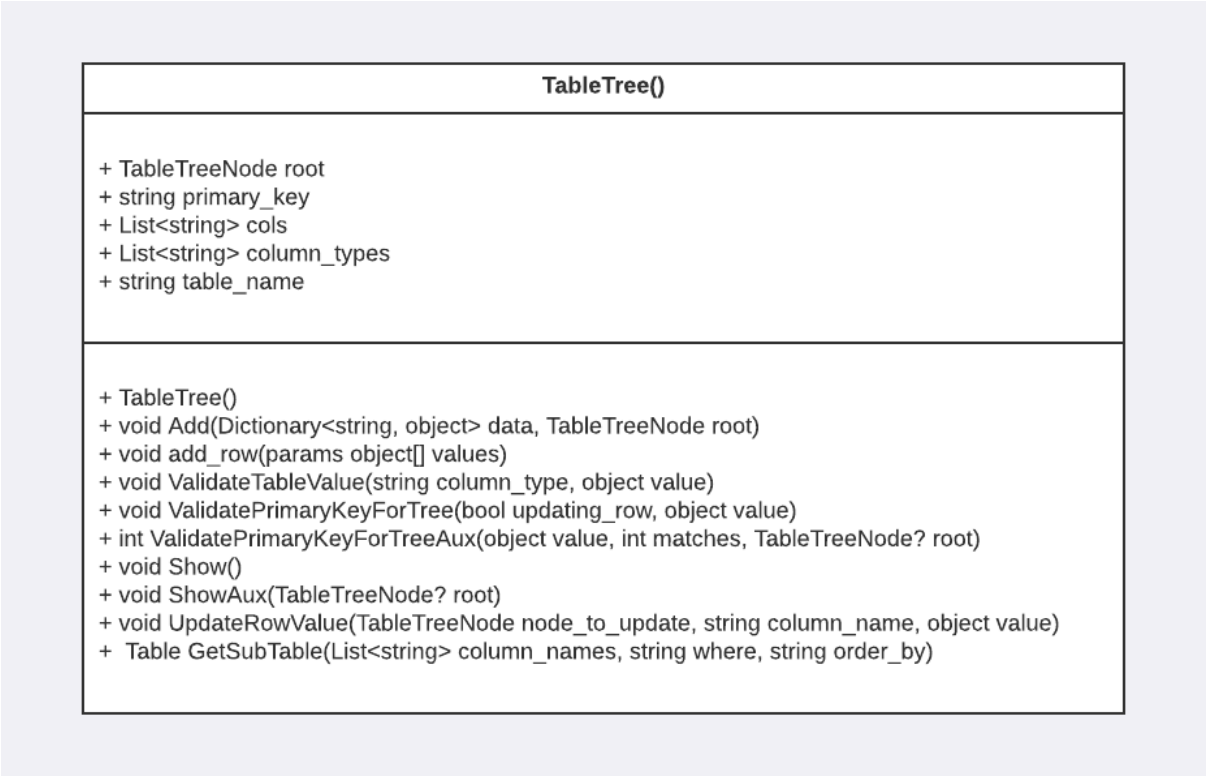
A continuación se muestran los diagramas UML con las clases más importantes del proyecto.






TinySQL_Server()
+ string address
+ int port
+ static void Start(string address, int port)
+ static void HandleClient(object obj)
+ static string ProcessQuery(string query)


Table()
+ List<string> cols
+ List <string> column_types
+ string primary_key
+ string table_name
+ List<Dictionary<string, object>> rows;
+ Table()
+ void TreeToTableAux(Table table, TableTreeNode? root)
+ void add_row(params object[] values)
+ Dictionary<string, object> get_row(int index)
+ void show()
+ void show_column_types()
+ void ValidateTableValue(string column_type, object value)
+ void UpdateRowValue(int row, string column_name, object value)
+ void ValidatePrimaryKey(int updated_row, object value)
+ Table GetSubTable(List<string> column_names, string where, string order_by)
+ bool WhereOperation(string op1, string op2, string operation, bool
op1_is_column, bool op2_is_column, int row, Table table)
+ static Table OrderBy(bool direction, string column_name, Table table)
+ void Delete(string where)
+ void Update(string column_name, object value, string where)
+ void TableToFile(string file_name, string path)
+ static Table FileToTable(string file_name)


TableTree()
+ TableTreeNode root
+ string primary_key
+ List<string> cols
+ List<string> column_types
+ string table_name
+ TableTree()
+ void Add(Dictionary<string, object> data, TableTreeNode root)
+ void add_row(params object[] values)
+ void ValidateTableValue(string column_type, object value)
+ void ValidatePrimaryKeyForTree(bool updating_row, object value)
+ int ValidatePrimaryKeyForTreeAux(object value, int matches, TableTreeNode? root)
+ void Show()
+ void ShowAux(TableTreeNode? root)
+ void UpdateRowValue(TableTreeNode node_to_update, string column_name, object value)
+ Table GetSubTable(List<string> column_names, string where, string order_by)