Universidad de Costa Rica

Facultad de Ingeniería

Escuela de Computación e Informática

CI-1312 Bases de Datos I

Tarea Programada I

Profesor

Rodrigo Bartels

Elaborado por:

Abellán Jiménez Mariana B50031
Aguilar Corrales Vladimir B50120

7 de abril de 2017

Índice general

0.1. Introducción

0.1.1. Resumen

La siguiente documentación es acerca de un sistema gestor para bases de datos que recibe archivos de formato csv, el cual tiene un formato en específico y solo sirve para realizar consultas de lectura sobre el mismo. Se implementó en dos fases, la primera donde la búsqueda de los datos se hacía de manera manual, a diferencia de la segunda, en la cual se buscaban estructuras de datos que lograran hacer las búsquedas más eficientes, en términos de tiempo.

Capítulo 1

Descripción del Sistema

La aplicación desarrollada se encarga de permitirle al usuario ver datos de un archivo de formato csv (comma separated values). El archivo contiene en la primera línea los nombres de los campos, en la segunda línea el tipo de dato de cada columna con base en este nomenclatura:

1. String: hilera de caracteres
2. int: número entero
3. double: número de punto flotante
4. date: fecha
5. boolean: valor booleano de true o false

Después de esta línea las demás filas son los registros de cada cliente.

Se le debe permitir al usuario realizar consultas de igualdad, diferente, mayor que, menor que, menor o igual, mayor o igual y en un rango cualquiera de los campos definidos en el archivo.

También debe permitir hacer consultas compuestas, en este caso, que sean conjuntivas o disyuntivas (OR y AND).

Capítulo 2

Implementación

En el siguiente capítulo se abordarán cuestiones ya relacionadas con la implementación del sistema de consultas.

2.1. Descripción

La aplicación tiene dos versiones, ambas difieren en términos de eficiencia en tiempo y espacio, ya que una es eficiente en tiempo y no en espacio y la otra es de manera contraria. A partir de ahora, cuando se hable de la versión ineficiente, es refiriéndose a la versión ineficiente en tiempo.

La versión ineficiente se compone de 19 clases, separadas por paquetes según su funcionalidad:

- file: En este se encuentra la clase FileLoader, que se encarga de abrir el archivo que se quiere de acuerdo a la ruta que indicó el usuario. Además según se le solicite, va pidiendo las filas para que se pueda verificar que cumpla con los requisitos solicitados por el usuario.
- filter: Existe una clase abstracta llamada FilteringOperation que contiene la forma para decidir que operación simple se puede hacer y además establece los métodos de los diferentes tipos de operaciones para que las otras clases de acuerdo a el tipo de dato realice la operación.
- main: Contiene la clase Main donde está la ejecución inicial del programa y el Menú donde se implementa la interfaz por línea de comando para el usuario.
- parser: Hay una clase abstracta Parser que establece el método parse que las clases que heredan implementar para hacer el parseo del archivo de acuerdo con el tipo de datos.
- query: Se ubican las clases QueryExecutor, QueryResult y QueryParameters. La primera se encarga de la lógica para poder ejecutar las consultas y devolver la clase QueryResult donde existe una lista con los resultados de las consultas realizadas. QueryParameters contiene los parámetros que se ocupan para poder realizar la consulta.

Para la versión eficiente se redujo a 14 clases la implementación, ya que se elimino el paquete filter y se convirtió en una sola clase. Más detalles de la implementación de cada una en la parte de Análisis.

Capítulo 3

Problemas y retos en implementación

El primer reto, que en realidad se viene enfrentando desde cursos anteriores, fue el diseño inicial de la aplicación, ya que por mala costumbre se empieza con implementación y en el camino ver como se va diseñando la aplicación. Luego de varias consultas se logró obtener un diseño aceptable para empezar a programar

En realidad, la implementación de la versión ineficiente no tuvo una mayor dificultad, gracias al diagrama de clases realizado, lo único fue buscar dentro del API de Java como realizar ciertas operaciones como separar según un símbolo (en este caso por comas) y abrir el archivo, es decir, cuestiones de sintaxis.

Otro problema curioso que se tuvo fue con el Scanner para obtener los datos del usuario, ya que en algunos casos, si se escribía un entero y luego iba una cadena de texto, hacía un salto de una vez y no dejaba escribir nada. La solución encontrada y para facilitar el asunto fue tener dos Scanner, uno para ingresar números y otros cadenas de texto.

Una vez terminada la parte ineficiente, para la versión mejorada se tuvo que pensar e investigar cuál podría ser la mejor opción para implementar estructuras de datos y que las mismas fueran eficientes en tiempo y que implicaciones en espacio requería tener velocidad a la hora de hacer consultas.

Cuando se encontró un posible escenario ideal, se tuvo que buscar si Java poseía implementaciones de las estructuras que se querían utilizar y ver la forma correcta de utilizarlas, esto se logró gracias al API en línea de este lenguaje de programación.

Capítulo 4

Análisis

La versión ineficiente se planteó mediante el algoritmo de fuerza bruta, es decir, recorrer todo el archivo para verificar que cada entrada cumpliera o no con las solicitudes del usuario. Esto en todo caso de búsqueda se hará en un tiempo de O(n), lo que en archivos grandes podría resultar muy perjudicial en términos de tiempo.

Se realizó una prueba con un archivo con aproximadamente un millón de líneas y efectivamente para cada consulta que se quería realizar el resultado duraba algunos segundos en ser desplegado, aproximadamente 15 segundos en promedio, por lo que resulta bastante tedioso que dure ese tiempo solo para averiguar el resultado de una búsqueda.

En términos de espacio, la idea es que el archivo fuera cargando a memoria lo que iba a consultar y si coincidía con los términos de búsqueda, que lo almacene y sino que lo descarte, liberando también la memoria al final de la ejecución, por lo que en términos de almacenamiento la aplicación es bastante eficiente.

Sin embargo, como se vio en el curso de Estructuras de Datos y Análisis de Algoritmos, para hacer una mejora en una situación, se requiere sacrificar otra. En este caso, si queremos mejorar los tiempos de ejecución, se debe sacrificar espacio, en el caso de la aplicación en cuestión, es espacio en memoria.

Para la parte eficiente, se deseo implementar la estructura TreeMap para almacenar un registro de los datos en memoria.

Se tiene un ArrayList de TreeMap, donde cada TreeMap representa una columna del archivo. La estructura por columna tiene como llave una entrada de la columna y como valores tiene una lista de listas, donde cada lista representa una fila del archivo que coincide con la llave. Por ejemplo, si existe una columna de apellido, el TreeMap tendrá una llave con un apellido en específico y como valores tendra la lista con todas las filas que coinciden con ese apellido

Según la documentación de Java, el TreeMap garantiza tiempos logaritmicos para los containsKey y de get, por lo que la eficiencia se ve aumentada notablemente, ya que de hecho a la hora de ser pruebas, el tiempo que dura haciendo una consulta es de milisegundos porque de forma natural en lo que se dura es imprimiendo los datos con las coincidencias.

Sin embargo, siempre se tiene que hacer un recorrido por todos los datos para acomodarlos en estos TreeMaps, esto se hace al inicio de la ejecución del programa cuando se pide la dirección del archivo csv. Para esta ejecución se dura en promedio 15 segundos, pero es un gato en tiempo que no es tan tedioso ya que se hace una sola vez, y no como en la versión ineficiente que se realizaba cada vez que se realiza una consulta.

Como los datos son cargados a memoria, la cantidad de gasto de la misma es bastante notable con respecto a el de la otra solución ineficiente en tiempo, por lo que se puede observar que realmente el sacrifico es importante.

Una de las posibles mejoras que se podría hacer a la aplicación, es cambiar la estructura de cuando una columna es de Strings y de boolean por un hashmap, ya que el mismo

asegura tiempos constantes para las operaciones de get y contains, lo cual mejoraría notablemente estas columnas en cuestión de tiempo de acceso a los datos.

Con este trabajo y la materia analizada en clase, se pudo observar el nivel de complejidad que lleva un DBMS, ya que si bien la implementación de lo realizado no fue tan complicado, el planteamiento y el diseño para realizarlo fue importante. Además, esta aplicación solo revisa archivos csv con un orden específico de los datos y solo se pueden consultar, de aquí la importancia de un DBMS en el cual se pueden almacenar los datos sin importar su formato, ya que estos tienen el objetivo de ser de propósito general (hablando de DBMS como SQL Server, Oracle, etc), esto por su puesto que debe implicar un trabajo mucho más pesado en el diseño y la implementación de esta pieza de software.

Otro punto importante es que esta aplicación no maneja seguridad de acceso a los datos, ya que cualquiera puede accesarlos, cosa que un DBMS decente sí debería manejar.

En general, este trabajo sirvió de herramienta para resaltar la importancia de la manipulación de los datos de una manera correcta y eficiente, para que el usuario no sufra con retrasos en manipularlos ni que la integridad de los mismos se vea comprometida, ya que por ejemplo, se debieron hacer muchas validaciones para que la aplicación funcionara correctamente y no se cayera por cualquier cosa, y aún así se podría decir que hacen falta más validaciones.

Además, se pudo observar la complejidad que puede implicar la implementación y la arquitectura que lleva un pequeño DBMS y como a gran escala esto se puede complicar para hacerlo más genérico y de propósito general.