Tarea Programada 1: Manipulación y consulta de datos

Rodrigo Acuña Fernández (B20051)
Paola Ortega Saborío (B55204)

Jueves 6 de Abril del 2017

Escuela de Ciencias de la Computación e Informática

Universidad de Costa Rica

Índice general

Capítulo 1

Problema

El objetivo principal de esta tarea programada fue implementar una aplicación que permitiera desplegar datos desde un archivo de texto y permitir a un usuario hacer consultas sobre dicho archivo. Ésta tarea se implementó en Java, y los archivos de texto utilizados como protocolización a la hora de desarrollar el programa fueron los archivos csv (comma separated values). Los requerimientos de la tarea fueron los siguientes:

1. Interfaz por consola, no gráfica.
2. Permitir al usuario indicar la ruta del archivo por cargar.
3. Realizar consultas de igualdad, desigualdad y rango (con sus respectivas excepciones detalladas en el enunciado). Además debe permitir al usuario realizar consultas complejas.
4. Procesar la consulta y mostrarla con los registros que cumplen, y su respectiva cantidad.

Como parte de la tarea, había que implementar dos versiones de la tarea. Primero había que desarrollar una primera versión de la tarea, luego hacer un análisis de rendimiento sobre esa versión y finalmente implementar una versión mejorada y más eficiente de la aplicación.

Capítulo 2

Implementación

2.1. Descripción de la solución implementada

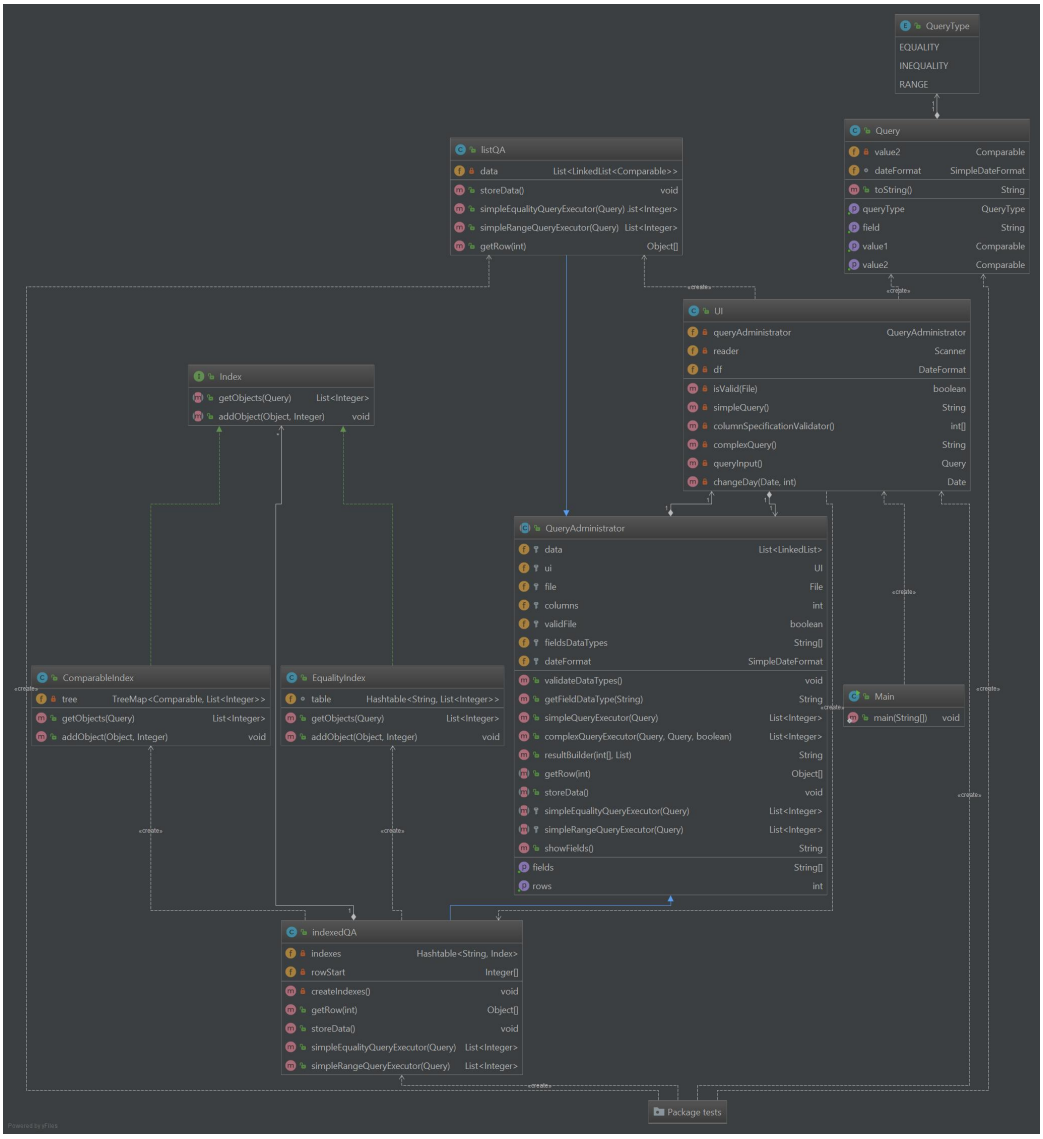
Para solucionar el problema se implementaron dos clases principales: Una interfaz UI que interactúa con el usuario para que el mismo le indique cual es el archivo que se desea consultar, y a su vez le permite hacer todo tipo de consultas, una vez que el archivo ya haya sido procesado. La interfaz se encarga también de validar que el archivo indicado sea existente y que el usuario ingrese consultas coherentes. Para lograr representar las consultas hechas por el usuario se implementaron dos clases pequeñas Query la cual almacena el campo que se desea buscar, el valor o rango de valores buscados y el QueryType la cual es una clase enumerada que indica el tipo de consulta que se esta realizando. Se definieron tres tipos de consulta: EQUALITY cuando el usuario quiere buscar las entradas con un valor específico, INEQUALITY cuando el usuario desea todos los valores distintos a un valor específico y finalmente RANGE cuando el usuario desdea. La otra clase principal es una clase abstracta llamada QueryAdministrator que se encarga de construir la base de datos, procesar las consultas recibidas de la interfaz y devolver el resultado. Esta clase es abstracta ya que se implementaron dos versiones distintas que la extienden, una versión inicial ineficiente listQA y una versión eficiente indexedQA.

La versión ineficiente utiliza como estructura de datos para almacenar las entradas de la tabla una lista de listas enlazadas. Cada lista representa una entrada o fila del archivo de la tabla. La listas son de objetos para poder almacenar todos los tipos distintos de datos soportados por el programa. Cuando se hace una consulta esta versión debe recorrer fila por fila verificando si la fila cumple los requerimientos especificados en la consulta.

La versión eficiente construye en cambio un índice por campo, Index es

una interfaz extendida por dos clases ComparableIndex, la cual se utiliza en campos cuyo tipo de dato soporta todos los tipos de consulta descritos anteriormente(Integer,Double,Date), y EqualityIndex para tipos de datos que solamente soportan consultas por igualdad y desigualdad(String,boolean). Estos índices implementan solamente dos métodos, uno para agregar entradas y otro para consultar por valores. Otra diferencia importante es que los índices no almacenan el objeto completo si no más bien almacenan la fila en que ocurre el valor. Esto permite que no sea necesario subir todo el archivo a memoria principal, si no más bien solo extraer las filas que coinciden con el resultado de la consulta. Para que la lectura del archivo fuera más eficiente esta versión utiliza lectura por bytes, guarda un puntero al byte en que inicia cada fila.

2.2. Diagrama de clases



2.3. Manual de usuario

Dado a la estaticidad del programa en sí al tener que trabajar con un archivo de formato predeterminado, el usuario debe asegurarse que el archivo que suba sea compatible con la implementación del programa. Es importante destacar las siguientes cosas:

1. Tal como sugiere el tipo de archivo requerido, cada uno de los tipos de datos deben estar separados por comas. Además, cada uno de los datos deben estar separados por un espacio de retorno.

2. En la primera línea se deben detallar las columnas principales. La segunda línea debe indicar el tipo de dato de cada columna.

3. Los tipos de datos son çase sensitive", por lo tanto es importante que el usuario deletrée bien el tipo de dato que solicita por columna. Los tipos de datos se escriben de la siguiente forma:

   a) String

   b) int

   c) double

   d) date

   e) bool

4. Los valores booleanos deben escribirse solamente como "True" o "False".

De igual forma, la interfaz de usuario se encarga de indicar paso a paso cómo se debe digitar la información, e intenta atrapar todos los errores posibles. A continuación se muestra un ejemplo de cómo tiene que digitarse la información de las primeras dos líneas del archivo:

2.4. Pruebas utilizadas

Para probar el programa se realizaron dos pruebas idénticas, en donde la única variación fue la versión del QueryAdministrator utilizada. En primera instancia se probó la versión ineficiente que utiliza una lista de listas enlazadas y posteriormente se hicieron exactamente las mismas consultas en la versión eficiente y se comparó que los resultados fueran los mismos. Inicialmente se realizaron pruebas sencillas para verificar que el programa leyera el archivo correctamente, que almacenara el formato(los campos y su tipo de dato) de la manera adecuada y que finalmente almacenara los datos en su estructura de datos utilizada. Posteriormente se procedió a probar los métodos de procesamiento de consultas. Se realizaron 6 consultas simples y luego se combinaron 4 de ellas para formar 2 consultas complejas.

En las siguiente sección se muestran imágenes con los resultados de algunas consultas simples y de una compleja.

2.5. Resultados obtenidos

En las siguientes dos páginas se mostrarán pantallazos de los resultados en la ejecución de las pruebas realizadas para las dos versiones del programa.

```
Male values not equal to: True
A total of 7 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Elly        False       23/08/1979      66      124.72
Fran        False       27/07/2006      66      115.23
Gwen        False       16/06/1982      64      121.61
Kate        False       9/12/1968       69      139.6
Myra        False       22/09/2005      62      98.95
Page        False       31/08/1998      67      135.25
Ruth        False       19/09/2006      65      131.15

Name values equal to: Neil
A total of 1 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Neil        True        29/09/2000      75      160.25

Birth values equal to: 26/02/2006
A total of 1 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Carl        True        26/02/2006      70      155.63

Birth entries in range: [01/01/2000 , 01/01/2010]
A total of 7 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Dave        True        18/03/2000      72      167.53
Neil        True        29/09/2000      75      160.25
Jake        True        8/9/2003        69      143.85
Myra        False       22/09/2005      62      98.95
Carl        True        26/02/2006      70      155.63
Fran        False       27/07/2006      66      115.23
Ruth        False       19/09/2006      65      131.15


Complex queries...
First query:
Male values not equal to: True
AND
Second query:
Weight entries in range: [100.0 , 150.0]

A total of 6 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Elly        False       23/08/1979      66      124.72
Fran        False       27/07/2006      66      115.23
Gwen        False       16/06/1982      64      121.61
Kate        False       9/12/1968       69      139.6
Page        False       31/08/1998      67      135.25
Ruth        False       19/09/2006      65      131.15
```

Figura 2.1: ListQA

```
Male values not equal to: True
A total of 7 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Elly        False       23/08/1979      66      124.72
Fran        False       27/07/2006      66      115.23
Gwen        False       16/06/1982      64      121.61
Kate        False       09/12/1968      69      139.6
Myra        False       22/09/2005      62      98.95
Page        False       31/08/1998      67      135.25
Ruth        False       19/09/2006      65      131.15

Name values equal to: Neil
A total of 1 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Neil        True        29/09/2000      75      160.25

Birth values equal to: 26/02/2006
A total of 1 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Carl        True        26/02/2006      70      155.63

Birth entries in range: [01/01/2000 , 01/01/2010]
A total of 7 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Carl        True        26/02/2006      70      155.63
Dave        True        18/03/2000      72      167.53
Fran        False       27/07/2006      66      115.23
Jake        True        08/09/2003      69      143.85
Myra        False       22/09/2005      62      98.95
Neil        True        29/09/2000      75      160.25
Ruth        False       19/09/2006      65      131.15


Complex queries...
First query:
Male values not equal to: True
AND
Second query:
Weight entries in range: [100.0 , 150.0]

A total of 6 entries match your query.
These are the matching results to your query:
Name        Male        Birth       Height      Weight
Elly        False       23/08/1979      66      124.72
Fran        False       27/07/2006      66      115.23
Gwen        False       16/06/1982      64      121.61
Kate        False       09/12/1968      69      139.6
Page        False       31/08/1998      67      135.25
Ruth        False       19/09/2006      65      131.15
```

Figura 2.2: IndexedQA

Capítulo 3

Análisis

En la primera versión del programa se utilizó, como estructura de datos, una lista de listas enlazadas. Se decidió utilizar dicha estructura porque era la más simple de implementar y se apegaba fuertemente a la lógica del archivo. Como resultado, logramos manejar datos en el programa utilizando solamente una gran estructura de datos.

En términos de tiempo, se sabe que el tiempo promedio de búsqueda a una columna específica se da en tiempo lineal. Además, se debe agregar el hecho de que cada una de las columnas tiene datos cuyo acceso también es lineal, en promedio. Por lo tanto, el tiempo de acceso a la hora de realizar consultas, o el acceso de datos en general, es $\mathcal{O}(n^2)$ como caso promedio. Por lo tanto, al utilizar esta versión del programa con archivos csv que contienen muchas columnas o un peso fuerte en datos, el tiempo de búsqueda en consultas y de despliegue de datos puede extenderse por una gran cantidad de tiempo.

La primer versión es ineficiente tanto en uso de memoria como en uso del espacio, por lo que es posible mejorar el uso de recursos en ambas dimensiones. Para la versión eficiente se implementaron cambios que mejoraron tanto en tiempos de inserción y búsqueda de consultas como en uso del espacio por parte del programa.

El problema principal en la primera versión en cuanto a uso de memoria es que al utilizar una lista de listas enlazadas se guarda cada dato como un objeto completo en memoria. Por esta razón el requerimiento de memoria es tan grande como el tamaño del archivo, o peor por el ' 'overhead' ' del lenguaje de programación. Esto puede ser una limitación a la hora de trabajar con documentos muy grandes, de miles o millones de entradas, ya que implica tener una duplicación innecesaria de la información. La versión eficiente utiliza índices, uno por cada campo, que permiten ubicar rápidamente la ocurrencia de un valor en el archivo sin necesariamente guardar el objeto completo. Debido a esto fue necesario implementar un método que leyera

solamente una fila del archivo. Si se utilizaba un Buffered Reader corriente, el ahorro en memoria implicaba un gasto extra en recorrer todo el archivo hasta la fila deseada por lo que se implementó un método de lectura por Bytes en la versión eficiente que permite saltar al punto del archivo deseado sin recorrerlo.

Las mayores mejoras de la segunda versión son en cuanto a tiempo de procesamiento de consultas. El propósito principal de la utilización de índices fue mejorar los tiempos de búsqueda por tipo de dato. Concretamente, el cambio realizado es que en la versión eficiente se consulta el índice por las filas en que aparecen (o no), el valor o rango de valores deseados y luego se extraen las filas del archivo. Esto evita recorrer todo el archivo linealmente, como previamente se detalló. La razón por la cual se implementaron dos tipos de índices es debido a que algunos tipos de Datos soportan tanto búsqueda por igualdad o desigualdad como búsqueda por rango, en cambio otros tipos de Datos solo soportan búsqueda por igualdad o desigualdad y se pueden beneficiar de una estructura de datos distinta. La separación por índices permite, por lo general, administrar la información de una manera más controlada y concisa.

En la segunda versión del programa, se buscó sacar provecho a las cualidades de los distintos tipos de datos para encontrar estructuras que se acoplaran de la mejor manera a las mismas. Como se detalla en el capítulo 2, se colocaron cada uno de los tipos de índices en un hashtable general, lo cual permite un acceso ideal de $\mathcal{O}(1)$. La estructura de datos dentro de cada índice es relativo al tipo de dato que se detalla en el documento que el usuario pasa. A los objetos de tipo Comparable, tales como los enteros, doubles o fecha, se les puede sacar provecho con un árbol rojinegro (o en el caso de Java, un TreeMap). Esto reduce los tiempos de inserción y búsqueda a ser $\mathcal{O}(log(n))$. Para los tipos de datos de Equality, tales como String y boolean, se indexaron su acceso en un Hashmap. Como resultado, la inserción de los datos es en tiempo constante, mientras que la búsqueda a los datos es en $\mathcal{O}(n)$, ya que se tiene que revisar por cada celda el nombre del String que contiene antes de saber su ubicación.

En cuanto a utilización de espacio, el hecho de no guardar el objeto completo en memoria principal puede representar una reducción importante en uso de este recurso. Sin embargo es importante destacar que el nivel de mejora es dependiente del archivo de entrada. El utilizar punteros a filas representa una reducción de espacio solamente si el puntero es de menor tamaño que el objeto a quien se apunta. Esto se cumple para tipos de datos como String (Dependiendo del largo de la hilera) y Date, y no necesariamente para Integers, Doubles, en donde el puntero puede ser del mismo tamaño que el objeto en sí. Esto implica que dependiendo de la distribución de los tipos de datos

de los campos de el archivo de entrada existirá una mejora en cuanto a espacio o no. Sería necesario realizar más pruebas con distintas distribuciones de tipos de datos y contrastar el uso de memoria por parte de ambas versiones para llegar a conocer si la mejora es significativa. En términos prácticos esto significa que el uso de memoria es $\mathcal{O}(n)$ en el peor casos para la versión eficiente, y es necesario hacer pruebas para identificar el caso promedio.

En conclusión, ambas implementaciones tienen sus ventajas y sus desventajas. Se puede tomar como ejemplo el método getRow(). En la primera versión, el acceso se logra en menos pasos, ya que simplemente devuelve en forma de arreglo lo que está en una posición lineal de la lista principal. Sin embargo, cargar todo esto a memoria puede ser sumamente costoso si los datos son abundantes. En la segunda versión se logra hacer lo mismo que en la primera en más pasos, ya que tiene que calcular la posición en el archivo deseado antes de sacar la línea, separarla y meterla en un arreglo. Sin embargo, en el fondo la segunda implementación logra encontrar la posición de la línea deseada de una sola vez, sin necesidad de cargar todas las líneas a memoria. La compensación de tiempo/espacio entre las dos implementaciones y la dependencia de archivos en el programa hace que lleguemos a una sola conclusión: las DBMS son esenciales para las necesidades de tanto los programadores como los usuarios de bases de datos.

Capítulo 4

Problemas y retos encontrados

Uno de los principales retos a la hora de programar fue la categorización lógica de las consultas, se sabía que habían muchos tipos distintos de consultas, simples y complejas, buscando un solo valor o un rango de valores, por tipo de dato, etc. Esto dejaba distintas maneras de agruparlas ya que sería tedioso e ineficiente tener que crear una clase por cada tipo distinto de consulta y posteriormente tener que escribir código que procese cada una de ellas. Al final se lograron agrupar por tipo de dato y consecuentemente tipo de dato soportado.

A la hora de programar la lectura por bytes en la versión eficiente se presentó un problema inesperado, inicialmente se estaba asumiendo que al finalizar la línea solo había que descartar un byte correspondiente al caracter de cambio de línea "\". Sin embargo, estaba imprimiendo saltos extraños y dividiendo incorrectamente las frases al asignarlas a vectores. Examinando más detenidamente qué se estaba leyendo del archivo se descubrió que el cambio de linea se hacía con dos caracteres "\r \n " por lo que se estaba asignando uno de los mismos a un String y al imprimirse cortaba las líneas.

No solamente se presentaron retos en la implementación de la solución, sino que también a la hora de tratar con la interfaz. Se tuvieron que agregar una cantidad exagerada de validaciones cada vez que se esperaba que el usuario digitara algo. Entre esas validaciones se tenía que verificar el tipo de dato de entrada, si era un dato enumberable que estuviera en el rango de opciones, etc. La inmutabilidad del archivo hizo que el programa se tuviera que acoplar a la forma del archivo, y de esa misma forma se le tuvo que exigir al usuario una sola forma de digitar la información.

Finalmente otro reto encontrado al implementar la solución es que el programa sea más flexible, entendiendo flexibilidad como el soporte de muchos y variados tipos de formatos y tipos de datos. En el estado actual el programa está muy hecho "a la medida" para una estructura y tipos de datos definidos,

cambios mínimos en el archivo fuente generarían que el programa no genere el resultado correcto o peor aún se caiga. Un ejemplo concreto de esto fue a la hora de buscar archivos para probar el programa, nos dimos cuenta que a veces se utiliza "TRUE" en vez de "true", y este mínimo cambio implicaría que ya el archivo hay que modificarlo para que sea compatible con nuestro programa, de igual manera si el formato de fecha es distinto al utilizado por nuestro programa. En general aumentar la compatibilidad y flexibilidad de el programa nos dimos cuenta es complicado y laborioso.