Data di consegna: 07/09/2018

# Progetto laboratorio di Algoritmi e Strutture dati

*A.A. 2017/2018*

Baiardi Martina (Mat. 825688),
Alessandro Lombardini (Mat. 841177 )

Ambiente di sviluppo: Visual Studio 2015 e 2017

Compilatore: MSBuild, predefinito di Visual Studio per C/C++

# Sommario

# REQUISITI

Il progetto svolto consiste nell'implementazione di un software in linguaggio ANSI C in grado di simulare le funzionalità di un Data Base Management System.

## CARATTERISTICHE GENERALI DEL PROGETTO

Il programma riceve in *input* una stringa alla volta dall'utente, la quale richiama i concetti base delle *query MySql*.

La stringa viene interpretata dal programma ed eseguita, *memorizzando* i vari dati su *file di testo*. L'interpretazione avviene attraverso la presenza di *parole specifiche*, che permettono di riconoscere quale richiesta l'utente sta facendo.

Ogni file di testo rappresenta una tabella di dati, dove il *nome del file*, assegnato dall'utente, corrisponde al *nome della tabella*.

## LE QUERY

Le query che il programma deve essere in grado di interpretare si suddividono in tre tipologie:

1. Creazione di una nuova tabella;
2. Inserimento di campi in una tabella preesistente;
3. Ricerca all'interno delle tabelle.

Ogni volta che viene chiamata una query di ricerca, i risultati ottenuti devono obbligatoriamente essere scritti su un file di testo, chiamato *query_results.txt*.

La funzione che elabora la stringa e gestisce l'esecuzione di ogni tipologia di query si chiama *executeQuery()*.

# IMPLEMENTAZIONE

## SCHEMA GENERALE

La struttura di esecuzione del programma è la seguente: ogni stringa scritta dall'utente viene passata alla funzione *executeQuery()*, la quale riconosce, attraverso l'utilizzo della libreria <string.h>, la presenza o meno di parole chiave delle query, in modo da poter poi inviare queste ultime alla funzione d'esecuzione specifica. Per cercare di ridurre al massimo gli errori al tempo d'esecuzione, prima di cercare le corrispondenze, la *executeQuery()* chiama una procedura, chiamata *correct_query(),* che permette di correggere errori minori di battitura da parte dell'utente.

Una volta che viene riconosciuta la query dalla stringa digitata, quest'ultima viene passata alla funzione d'esecuzione specifica, anch'essa divisa in piú parti.

## FUNZIONI IMPLEMENTATE PER ESEGUIRE LE QUERY

### 1. CREAZIONE DI UNA TABELLA - bool create_table(char *query);

Funzione che gestisce la creazione di un nuova tabella.

*Stringa di esempio:*
*CREATE TABLE nome_tabella (nome_colonna1,nome_colonna2,…)*

Questa stringa ordina al programma di creare una nuova tabella.

La creazione di una tabella corrisponde alla creazione di un file di testo in formato .txt il cui nome corrisponde a *nome_tabella*.

Per prima cosa la funzione estrae il nome della tabella che si vuole creare, chiamando la funzione *copy_table_name()*, e chiede un'apertura di questa tabella in lettura; in questo modo si puó identificare, attraverso il controllo sul puntatore al suddetto file, se la tabella era precedentemente esistente. In tal caso viene restituito false alla funzione principale senza procedere oltre.

In caso contrario il nuovo file creato conterrà una sola riga di testo, in questo formato:

*TABLE nome_tabella COLUMNS nome_colonna1,nome_colonna2,...;*

## 2. INSERIMENTO DATI - bool insert_into(char *query);

Funzione che gestisce l'inserimento di una riga di dati (istanza) all'interno di una tabella giá esistente.

*Stringa di esempio:*
*INSERT INTO nome_tabella (nome_colonna1,nome_colonna2,...) VALUES (valore1,valore2,...)*

Il primo passo dell'esecuzione, come per la CREATE TABLE, è una chiamata alla *copy_table_name()*, per prelevare il nome della tabella dalla stringa, procedendo poi al controllo.

In caso negativo viene immediatamente restituito false, concludendo l'esecuzione, viceversa si procede con il controllo sull'equivalenza delle colonne inserite con quelle date nella definizione della tabella. Infatti i parametri contenuti nella prima coppia di parentesi devono essere coerenti con quelli dichiarati al momento della creazione della tabella, nonché presenti nella prima riga di intestazione del file.

*Siccome, prima di effettuare l'inserimento, la query ha subito una correzione in merito a spazi superflui lasciati dall'utente,* ***la insert_into() non prevede l'inserimento di un campo spazio (' ')***. *Se si vuole inserire un campo senza un preciso valore, allora deve essere scritto* ***NULL*** *nella posizione scelta.* In caso contrario verrà restituito errore.

Se non si verificano errori, la tabella viene aggiornata con l'inserimento della riga richiesta, la quale verrà scritta in questo modo:

*ROW valore1,valore2,valore3,...;*

## 3. RICERCA ALL'INTERNO DELLE TABELLE

Per effettuare una ricerca all'interno delle tabelle, viene utilizzato il comando SELECT.

Ogni volta che viene utilizzato questo comando, viene creato un file chiamato *query_results.txt*, il quale conterrà la query richiesta e il rispettivo output ottenuto. Si avrà un risultato solo quando la query andrà a buon fine. Se il file è già esistente, verrà aggiornato.

### 3.1. bool select_all(char *query); e bool select_columns(char *query);

Funzioni che permettono l'esecuzione di una ricerca senza filtri. Quest'ultima puó essere fatta in due modi, digitando le seguenti stringhe:

*SELECT * FROM nome_tabella*

*SELECT nome_colonna1,nome_colonna3,... FROM nome_tabella*

La prima visualizza tutte le colonne contenute nella tabella specificata (*select_all()*), mentre la seconda solo le colonne scelte (*select_columns()*).

La *select_all()* non utilizza la struttura dati, in quanto basta prelevare il nome della tabella dalla stringa e leggere tutto il file che viene aperto per eseguirla.

Infatti questa funzione chiama la *put_query_results()* per salvare nel file *query_results.txt* la stringa data in input e la *extract_table()* che estrae dalla stringa il nome_tabella e apre il relativo file, restituendolo.

Se queste due funzioni restituiscono un puntatore a file con valore *NULL*, significa che non sono andate a buon fine e di conseguenza viene interrotta l'esecuzione con valore *false*.

Se i controlli vanno a buon fine viene stampato tutto il contenuto del file, compresa la sua intestazione, sia a video che nel file *query_results.txt*.

Una volta raggiunta la fine, i due file vengono chiusi e viene restituito valore *true*.

La *select_columns()* utilizza invece la struttura dati per:

- semplificare i controlli relativi alla corrispondenza tra le colonne preesistenti in tabella e quelle richieste nella query
- semplificare la stampa dei loro contenuti

Come per la *select_all()* e tutte le altre query di selezione, le prime funzioni che richiama sono la *put_query_results()* e la *extract_table()*. Una volta effettuati questi passaggi e verificato che non siano stati commessi errori, si procede con la chiamata alla funzione che crea la struttura dati, cioè la *make_structure()*.

Se la funzione di creazione della struttura dati non va a buon fine, allora il nome della struttura, cioè *table*, ha valore NULL. In questo caso si interrompe l'esecuzione e viene restituito valore *false*.

Successivamente alla creazione della struttura, viene chiamata la procedura che stampa le colonne richieste della struttura dati creata: la *view()*.

Una volta effettuati i controlli sulla sua esecuzione, si puó dire che la funzione ha avuto esito positivo. Si conclude cosí chiudendo i file di testo precedentemente aperti e la struttura dati, chiamando l'apposita procedura *free_structure()*.

### 3.2. bool select_where(char *query);

Funzione che permette l'esecuzione della ricerca con filtro WHERE. Questa funzione viene attivata con la seguente sintassi:

*SELECT ... FROM nome_tabella WHERE condition*

Come precedentemente descritto, viene immediatamente chiamata la funzione *put_query_results()* con il relativo controllo che sia andata a buon fine. Peró, prima di chiamare la *extract_table()*, viene estratta dalla query la condizione della clausola WHERE e salvata in una stringa di supporto, chiamata *condition*. Una volta controllato che la *extract_query()* abbia avuto buon esito si procede con la creazione della struttura.

Prima di passare alla stampa del contenuto della struttura, quest'ultima viene elaborata chiamando la procedura *clean_structure()*. Tramite quest'ultima vengono effettuati i controlli sulla condizione salvata precedentemente nel vettore *condition* e la relativa colonna di tutte le istanze della struttura dati. Le istanze che non rispettano la condizione vengono eliminate, restituendo cosí una tabella contenente solamente le righe che soddisfano la clausola.

Controllato che la *clean_structure()* abbia avuto buon fine, cioè controllando che il parametro con il numero delle righe (*num_row*) sia diverso da -1, si procede con la stampa attraverso la *view()*. Se anche questa ha avuto esito positivo, si conclude con la chiusura dei file e la chiamata della *free_structure()*, restituendo *true*.

Per quanto riguarda l'interpretazione del tipo di dato da confrontare (cioè se si tratta di un intero o di una stringa), viene sfruttata la funzione *atoi()*, la quale permette di convertire una stringa in intero. Se la tipologia di dato è mista, cioè contiene sia caratteri alfanumerici che numeri interi, la colonna verrà trattata come se contenesse stringhe. Ad esempio:

SELECT nome_colonna FROM tabella WHERE nome_colonna == 18e

Questa colonna viene interpretata dal programma come contenente caratteri stringhe. Nel caso in cui la colonna non contenga stringhe ma interi,

la procedura *clean_structure*() riconosce l'incompatibilità e interrompe l'esecuzione, in quanto non è possibile confrontare un numero con una stringa.

### 3.3. bool select_group_by(char*query);

Funzione che effettua la ricerca con filtro GROUP BY, chiamata attraverso la scrittura di una query con la seguente sintassi:

SELECT nome_colonnaX FROM nome_tabella GROUP BY nome_colonnaX)

La clausola GROUP BY è utilizzata nelle operazioni di SELECT al fine di raggruppare i valori identici presenti in una o più colonne.

Viene chiamata la funzione *put_query_results*(), la quale va ad inserire la query di input nel file di testo *query_results.txt*. Successivamente si procede con l'identificazione della colonna specificata dalla clausola della query. Prima di effettuare la ricerca della colonna all'interno della tabella, si effettua un controllo su di essa: si verifica che quella richiesta nella clausola GROUP BY sia la stessa richiesta dalla SELECT; se ciò non accade significa che l'utente ha compiuto un errore di sintassi, viene interrotta l'esecuzione del programma e restituito *false*.

Dopo aver estratto il nome della tabella richiesta (viene sfruttata una copia della *extract_table*() modificata a causa della diversa disposizione degli spazi all'interno della query) si procede con l'apertura del file e si passa alla creazione della struttura dati con la funzione *make_structure*().

Se quest'ultima ha avuto esito positivo viene verificato che la colonna richiesta esista e, in tal caso, la struttura viene ordinata richiamando la procedura *mergeSort*() che realizza un ordinamento crescente rispetto alla colonna il cui indice è stato passato come parametro.

La tabella viene ora stampata tramite una copia della procedura *view*(), appositamente modificata per il fine della clausola GROUP BY.

### 3.4. bool select_order_by(char *query);

Funzione che effettua la ricerca con filtro ORDER BY, chiamata attraverso la scrittura di una query con una delle seguenti sintassi:

SELECT ... FROM nome_tabella ORDER BY nome_colonna ASC

SELECT ... FROM nome_tabella ORDER BY nome_colonna DESC

Come per tutte le precedenti query di selezione, viene chiamata per prima la funzione *put_query_results()* e viene verificato che sia andata a buon fine.

In caso positivo, viene verificato che all'interno della query ci sia la presenza della tipologia di ordinamento: ***ascendete*** (***ASC***) o ***discendente*** (***DESC***). Nel caso l'esito sia positivo si procede con l'identificazione della colonna da ordinare richiesta dalla query; se invece non è presente la tipologia di ordinamento la funzione termina l'esecuzione e ritorna il valore *false*. Si procede quindi con l'apertura del file di testo, per la quale viene utilizzata, anche questa volta, una copia della *extract_table()* leggermente modificata a causa della specifica disposizione degli spazi all'interno della query.

Dopo aver controllato che la tabella esista, si procede con la creazione della struttura dati, attraverso la funzione *make_structure*(). Una volta che questa è stata creata con successo, si ricerca al suo interno la corrispondenza con la colonna richiesta dall'utente nella clausola ORDER BY. Se non viene trovata, il programma interrompe l'esecuzione e restituisce *false* dopo aver liberato le celle di memoria dinamica, in caso contrario si procede con la chiamata della procedura *mergeSort*(), che ordina la struttura creata rispetto alla colonna e all'ordine specificati nei parametri.

Una volta ordinata la struttura, si procede con l'applicazione della procedura *view*(), che stampa sia a video che nel file apposito i risultati della query.

Se quest'ultima procedura non ha avuto esito negativo si conclude che la *select_order_by*() ha avuto buon esito, restituendo true.

# FUNZIONI IMPLEMENTATE DI SUPPORTO

### char *copy_table_name(char *input, char* query);

Funzione che ricerca il nome del file, all'interno delle query di creazione di una nuova tabella e di inserimento, e lo restituisce.

### FILE* extract_table(char*query, char *input);

Funzione che ricerca all'interno delle query di selezione il nome della tabella da aprire. Questa ricerca viene effettuata sfruttando la funzione strrchr(), che restituisce un puntatore all'ultima occorrenza del carattere passatole come argomento, passandole il carattere spazio.

Il nome del file viene copiato sul parametro *input* e viene completato con l'estensione ".txt", per poi aprirlo con la funzione fopen().

Se il file richiesto dall'utente non esiste, questa funzione restituirà un puntatore a NULL. In caso contrario viene invece restituito, ma dopo aver aggiornato il parametro *input* rimuovendo l'estensione.

void view(char*** table, char ** columns, int num_columns, int num_row, char*query,char*input, FILE ** results);
Procedura principale per le query di selezione: permette di visualizzare le colonne richieste dall'utente tra tutte quelle presenti in tabella.

I parametri della procedura sono:

1. ****table,* puntatore alla struttura contenente tutte le righe di dati della tabella.
2. ***columns*, puntatore ad un vettore dinamico contenente i nomi delle colonne presenti nell'intestazione della tabella.
3. *num_columns,* numero delle celle contenute nel vettore *columns.*
4. *num_row* è il numero di righe della struttura dati, nonché il numero di istanze che l'utente ha inserito nella tabella.
5. **query*, stringa inserita dall'utente. Questa viene successivamente elaborata e infine, quando viene inviata alla procedura view, contiene solamente il nome delle colonne richieste dall'utente per la visualizzazione.
6. **input,* nome della tabella selezionata. Questo parametro serve nel momento di stampa dell'intestazione della tabella.
7. ***results*, puntatore al file di testo che ha come nome *query_results.txt*.

Inizialmente, la procedura distingue i due casi principali di visualizzazione, ossia cerca una corrispondenza con il carattere "*" perché, nel caso in cui questo sia presente, allora la stampa deve avvenire di tutte le colonne. Nel caso opposto, invece, effettua il confronto tra i nomi delle colonne richieste e quelle contenute nell'intestazione della tabella, cioè nel vettore *columns*.

Per prima cosa, ogni *nome_colonna* presente nella query viene confrontato con il vettore *columns*, che contiene tutte le colonne presenti in tabella.

Se un *nome colonna* non trova corrispondenza, significa che la colonna richiesta dall'utente non esiste in tabella e di conseguenza è stato fatto un errore di sintassi; l'esecuzione si interrompe e non viene stampato nessun risultato. Per far capire alla funzione richiamante che la view non è andata a buon fine, viene posto a NULL il puntatore al file di output (*results*).

Se invece tutte le colonne trovano corrispondenza, queste vengono memorizzate numericamente rispetto al vettore *columns* e successivamente stampate sia a video che nel file di testo *query_results.txt*.

void clean_structure(char*** table, char ** columns, int num_columns, int *num_row, char*condition);
Procedura principale per la query di selezione con filtro WHERE: questa permette di eliminare le righe che non soddisfano la condizione data dalla stringa in input.

Il parametri ****table*, ***columns*, *num_columns* e *num_row* rappresentano gli stessi dati dei parametri della funzione view() descritti sopra.

Il parametro **condition* è la parte finale di stringa contenuta nella query inserita dall'utente, contenente solo il nome della colonna e la rispettiva condizione.

Per prima cosa viene cercata l'esistenza in tabella della colonna selezionata nella condizione. Se questa non è presente l'esecuzione termina e restituisce il parametro num_row come -1, in modo da permettere alla funzione *select_where()* di riconoscere che è avvenuto un errore.

Successivamente, attraverso l'utilizzo della funzione *strncmp()*, vengono comparati i simboli della condizione con quelli che possono essere interpretati dal programma. Una volta trovata una corrispondenza, la procedura ha compreso quale simbolo deve applicare, quindi va a confrontare che tutte le righe contengano nella colonna identificata un valore coerente con la condizione.

Per ogni simbolo, vengono proposti due blocchi alternativi di procedimento: il primo interpreta la condizione come stringa, il secondo come numero intero. Il riconoscimento della tipologia di dato è compiuto dalla funzione ***atoi()*** fornita dalla libreria *<stdlib.h>* che permette la conversione da stringa a intero. Per verificare la correttezza della conversione viene effettuata una seconda conversione da intero a stringa con la funzione *_itoa()*; se la seconda conversione e la condizione della WHERE combaciano, significa che il numero è un intero, in caso contrario una stringa. Questo ci permettere di verificare univocamente se ci troviamo davanti ad un tipo dato stringa o intero.

*Se il tipo di dato della condizione e il tipo di dato della colonna sono diversi, non è possibile effettuare una comparazione tra questi, pertanto il programma restituirà errore ed interromperà l'esecuzione.*

Solo uno dei blocchi viene eseguito e al suo termine la funzione ritorna alla principale (*select_where()*).

Se la condizione non è verificata viene eliminata l'intera riga dalla struttura.

Se invece non viene trovata una corrispondenza tra i simboli a disposizione, si procede nello stesso modo precedentemente descritto per permettere alla *select_where()* di riconoscere un errore, cioè ponendo a -1 *num_row*.

void correct_query(char*query);
Procedura che viene chiamata prima di passare la stringa alla funzione *executeQuery()*. È stato scelto di introdurla per minimizzare errori al tempo d'esecuzione dovuti a spazi mal inseriti dagli utenti e/o parole chiave scritte in minuscolo al posto del maiuscolo.

FILE* put_query_results(char*query,FILE*results);
Funzione che inserisce all'interno del file *query_results.txt* la stringa scritta (e corretta) dell'utente, prima che venga scomposta per poter procedere con l'esecuzione dei comandi.

# FUNZIONI IMPLEMENTATE PER L'ORDINAMENTO

void mergeSort(char *** arr, int l, int r, int order_colon, int order, int string_or_not);
In generale la *mergeSort*() è una procedura che riprende i concetti dell'algoritmo di ordinamento omonimo. L'idea consiste nel suddividere il blocco di elementi in due parti e poi ordinarli al momento della loro successiva unione tramite la procedura *merge*(). Questo procedimento è applicato ricorsivamente sugli elementi.

La procedura *mergeSort*() permette di applicare questo particolare algoritmo alla struttura dati da noi utilizzata.

L'algoritmo Merge Sort ha **costo T(n) = Θ(nlogn)** sia nel caso medio che nel caso pessimo, risulta quindi essere un algoritmo ottimo per l'ordinamento.

*mergeSort*() presenta i seguenti parametri di input:

- arr: puntatore alla struttura dati
- l: indice sinistro
- r: indice destro
- order_colon: indice della colonna da ordinare
- order: orientamento desiderato (ascendente o discendente)

- string_or_not: indica se l'ordinamento deve avvenire considerando i valori come stringhe oppure come interi

Gli indici l e r consentono di definire, al momento della divisione in due blocchi, il primo e l'ultimo elemento dei rispettivi blocchi.

Il passo base della ricorsione è dato dalla situazione in cui l'indice sinistro è più grande di quello destro.

A ciascun richiamo si identifica l'indice dell'elemento in posizione centrale al blocco analizzato e viene effettuato un richiamo di tipo ricorsivo alla procedura mergeSort() per entrambi i blocchi venutisi a creare (quello sinistro e quello destro all'identificato).

Successivamente viene chiamata la procedura *merge*() che consente di unire in modo ordinato i blocchi precedentemente creati.

L'ordinamento si differenzia in base alla tipologia di dato da ordinare.

Nel caso in cui tutti i primi caratteri dei valori della colonna da ordinare siano numeri, l'ordinamento viene effettuato per ordine numerico, mentre nel caso in cui ci sia anche solo un carattere alfabetico come primo, l'ordinamento avverrà in base alla codifica nella tabella ASCII.

*String_or_not* è la variabile che specifica questa particolare caratteristica. Al primo richiamo della procedura *mergeSort()* la variabile avrà valore 2, in modo da comunicare al codice che la tipologia di ordinamento deve ancora essere scelta. Vengono analizzati tutti i valori della colonna e scelto l'opportuno metodo grazie alla funzione *atoi()*. Il valore 1 indicherà che si tratta di una stringa mentre il valore -1 che si tratta di un intero.

Sarà poi la procedura *merge()* a tenere conto di questa differenza al momento dello spostamento dei valori.

```
void merge(char *** arr, int l, int m, int r, int order_colon, int order, int string_or_not);
```

La procedura *merge*() viene utilizzata durante il processo di ordinamento della struttura dati. Viene chiamata unicamente dalla procedura *mergeSort*() con il preciso scopo di unire e ordinare gli elementi presenti fra gli indici l e m ed m e r del vettore dinamico arr.

L'indice m definisce i due blocchi da unire: con l'indice ***l*** si indentifica il blocco sinistro mentre con l'indice ***r*** si identifica il blocco destro.

I confronti necessari per lo svolgimento dell'algoritmo Merge Sort vengono effettuati all'interno di questa procedura: essi sono applicati alla colonna il cui indice è il parametro *order_colon*, mentre la tipologia di ordinamento, ascendente oppure discendente, viene stabilita dal parametro *order*.

Il parametro *string_or_not* ha la precisa funzione di indicare se lo spostamento dei valori deve avvenire seguendo un ordinamento di tipo numerico oppure basato sulla codifica ASCII. Il compito di stabilire la tipologia del valore è lasciato alla procedura *mergeSort()*.

# STRUTTURA DATI UTILIZZATA

## IN COSA CONSISTE

La struttura dati consiste in un vettore dinamico di vettori dinamici di stringhe.

Il vettore dinamico è stato il mezzo tramite il quale è stato possibile creare una struttura adatta alle nostre esigenze. L'esigenza era la possibilità di caricare i dati contenuti all'interno del file .txt in memoria, organizzandoli in una struttura capace di adattarsi alla loro dimensione.

L'idea è stata di gestire i record come elementi di un vettore dinamico in quanto il numero di questi è variabile e sconosciuto fino al momento della query.

Il numero di dati legati a ciascun record rappresenta un'incognita e dunque, escludendo strutture non dinamiche (come struct, vettori non dinamici, ecc ...), è stato utile gestire ogni singolo record come un vettore dinamico esso stesso. Nelle celle di quest'ultimo vettore sono contenuti i valori delle colonne.

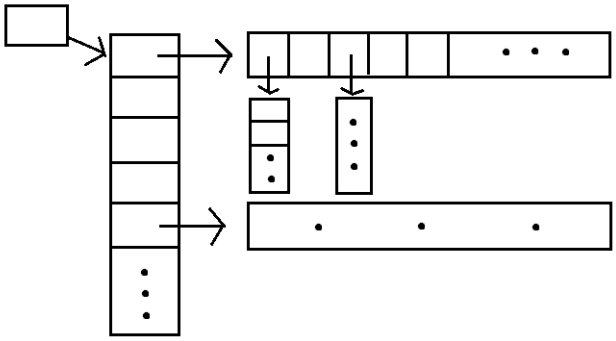
Esiste un vettore che rappresenta le righe e, per ogni riga, un vettore che rappresenta le colonne.

Il linguaggio consente di trattare questa struttura come una matrice e di accedere ad ogni singola cella tramite due valori: indice della riga e indice della colonna. L'elemento raggiunto sarà un puntatore a carattere rappresentante la stringa memorizzata.

Il risultato è estremamente simile alla struttura dei dati memorizzati all'interno dei file del programma: la tabella.

La tipologia di dato della struttura sarà un char ***.

A questa struttura va aggiunto, per completezza, un'ulteriore vettore di stringhe contenente i nomi delle colonne della tabella.



L'accesso alle celle di memoria avviene tramite la notazione nome_struttura[indice_riga] [indice_colonna].

## MOTIVAZIONI DELLA SCELTA

Il problema richiede di lavorare su dati generici con operazioni di ordinamento, raggruppamento e selezione, con e senza filtro.

La struttura dati così realizzata risulta necessaria per quanto riguarda le operazioni di ordinamento e raggruppamento e inoltre consente di applicare le clausole delle SELECT in maniera meno complessa, semplificando il codice. Per tali motivi il vettore dinamico è risultato essere la struttura dati con il rapporto qualità-costo migliore.

Senza rinunciare a ottime prestazioni dal punto di vista di spazio e tempo abbiamo quindi realizzato una struttura dati non estremamente complessa da realizzare e, grazie al linguaggio, semplice da gestire.

L'uso di altre strutture, data la maggiore complessità per la realizzazione e la gestione, non è parso essere favorevole quanto la scelta fatta. Non è risultato esserci un guadagno di tempo e spazio abbastanza consistente per giustificare la scelta di un albero, un heap o una qualsiasi altra struttura dati studiata a lezione.

Le operazioni più complesse per le quali è richiesta la struttura dati, quali ORDER BY e GROUP BY, vengono infatti applicate con costi computazionali pari ad altre strutture anche più complesse.

Il tempo necessario per la costruzione della struttura dati risulta però essere il lato negativo. Data una tabella di n righe e m colonne, questa presenta un costo di $\Theta(n*m)$.

Tendenzialmente il numero di colonne non eccede oltre grandi numeri, e pertanto il costo rimane quindi contenuto e la struttura non perde di

efficienza in particolare quando si parla di tabelle con grandi numeri di record e poche colonne. La differenza è particolarmente significativa nel caso in cui si abbiano pochi record e molte colonne.

Anche la stampa dei valori della struttura è strettamente legata al numero di colonne e righe, ma ciò non è molto diverso da ciò che anche le altre strutture devono affrontare.

La soluzione non è in grado di gestire tabelle dell'ordine di migliaia di righe in quanto l'intera struttura viene caricata in memoria e questa risulta essere ovviamente limitata dal punto di vista fisico.

## FUNZIONI PER LA STRUTTURA DATI

### char *** make_structure(FILE * file, char *** n_colonne, int * num_colonne, int * num_row);

Funzione che crea la struttura dati a partire da un file di testo rappresentante una tabella.

La funzione restituisce il puntatore alla struttura e, al tempo stesso, fornisce all'interno del parametro n_colonne il vettore dinamico contenente i nomi delle colonne.

La prima cosa che la funzione fa è contare il numero di colonne e creare un vettore dinamico di dimensione pari a tale valore. In seguito, se eventuali controlli di allocazione non rilevano errori, inserisce all'interno della struttura le rispettive stringhe rappresentanti le intestazioni delle colonne.

A questo punto l'intero file viene analizzato al fine di contare il numero di record presenti. Si procede creando un vettore dinamico di char** di dimensione pari al numero di righe. Per ciascuna cella di questo vettore viene allocato un vettore dinamico di char* di dimensione pari al numero di colonne.

La *make_structure()* resetta il puntatore a file alla prima riga e analizza, riga per riga, tutti i valori, inserendoli all'interno della struttura appena creata accedendo alla singola cella di memoria tramite la notazione:

*nome_struttura[indice_riga][indice_colonna]*.

### void free_structure(char*** table, char ** columns, int num_columns, int num_row);

Procedura che consente di deallocare tutta la memoria allocata nel momento in cui è stata chiamata la funzione *make_structure*().

# ANALISI DEI COSTI COMPUTAZIONALI

## char *copy_table_name();

Funzione che contiene un solo ciclo che scorre una porzione di stringa contenente il nome della tabella. Il costo prodotto è il seguente:

$$T(n) = O(q)$$

Dove q è la lunghezza della stringa passata.

## FILE* extract_table();

Funzione che ha l'obbiettivo di estrarre ed aggiungere l'estensione al nome del file. Essa compie rispettivamente una *strrchr()* e una *strcat(),* con costo O(q) ciascuna, cioè strettamente inferiore della lunghezza (q) della stringa passata.

Il nome della tabella viene salvato sul parametro input con una *strcpy()*, con costo O(q).

Infine viene effettuato il controllo sulla validità del puntatore a file con costo sempre inferiore o al massimo uguale alla dimensione dell'input passato.

Pertanto, il costo della funzione *extract_table()* risulta essere:

$$T(n) = \Theta(q)$$

Dove q è la lunghezza della stringa passata.

## void view();

Il costo di questa procedura è dato dai cicli che si occupano di stampare il contenuto della tabella di n righe e m colonne.

Nel caso pessimo la tabella viene stampata tutta producendo il seguente costo:

$$T(n)= O(n*m)$$

Dove n è il numero di righe ed m il numero di colonne della tabella.

## void clean_structure();

Siccome viene eseguito solo uno degli 8 blocchi per esecuzione, in quanto questi vengono scartati in base alla costruzione della query, il costo di questa procedura non è più elevato rispetto alle altre.

Viene eseguito un ciclo che si occupa di stabilire, riga per riga, se il record analizzato è valido oppure no.

Se il record risulta non essere valido, viene deallocata tutta la memoria occupata dagli m valori colonna del puntatore a riga, mentre quest'ultimo viene impostato a NULL. Questo passaggio presenta un costo pari a Θ(m).

Nel caso pessimo devono essere scartati tutti i record, dunque:

$$T(n) = O(n^*m)$$

Dove n è il numero di righe ed m il numero di colonne della tabella.

## void correct_query();

il punto che ha costo maggiore è il ciclo che analizza tutta la stringa per controllarne la sintassi.

$$T(n)= \Theta(q)$$

Dove q è la lunghezza della stringa passata.

## FILE* put_query_results();

Funzione che non contiene cicli, semplicemente inserisce una stringa nel file *query_results.txt*.

$$T(n) = costante$$

## void merge();

Unisce due sotto sequenze ordinate con costo:

$$T(n) = \Theta(n)$$

Dove n è il numero di righe della tabella.

## void mergeSort();

Rappresenta l'implementazione dell'algoritmo di ordinamento Merge Sort, basato sulla ricorsione, e sfrutta la tecnica Divide et Impera per ordinare i valori.

La procedura richiama sé stessa due volte, ogni volta su circa la metà dell'input precedente. Essa inoltre chiama la procedura merge() che ha costo Θ(n).
È previsto inoltre che, al primo richiamo, vengano guardati tutti i valori della colonna da ordinare, con un costo di Θ(n).

La procedura presenta quindi un costo di $T(n) = 2T(n/2) + \Theta(n)$

Segue che mergeSort() ha costo:

$$T(n) = \Theta(nlogn)$$

Dove n è il numero di righe della tabella.

## char *** make_structure();

La funzione come primo passo scorre tutte le colonne e, separatamente, tutte le righe con costo rispettivo di Θ(m) e Θ(n).

Un ulteriore costo pari a Θ(m) va poi imputato all'inserimento dei nomi delle colonne all'interno della struttura dedicata.

Per l'inserimento dei valori all'interno della struttura si prevede invece un costo Θ(n*m) in quanto si tratta di inserire un elemento in ogni cella di una matrice n per m.

$$T(n) = \Theta(n*m)$$

Dove n è il numero di righe ed m il numero di colonne della tabella.

## void free_structure();

La procedura dealloca tutte le celle della struttura dati scorrendo n righe e, per ciascuna riga, m colonne con costo Θ(n*m).

Vengono poi deallocate le celle di memoria contenenti i nomi delle colonne con costo Θ(m). Il costo totale prodotto da questa procedura è quindi:

$$T(n) = \Theta(n*m)$$

Dove n è il numero di righe ed m il numero di colonne della tabella.

## bool create_table();

Funzione che richiama:

- copy_table_name();
- strcpy();
- ciclo for che inserisce i nomi delle colonne scritte nella query;

la prima e la seconda funzione han costo Θ(q), mentre il ciclo ha costo o(q) perché setaccia una sola parte della stringa passata.

Il costo della funzione create_table() è:

$$T(n) = \Theta(q)$$

Dove q è la lunghezza della stringa passata.

## bool insert_into();

Funzione che richiama:

- copy_table_name();
- strcat();
- fgets();

- ciclo che conta il numero delle colonne;
- ciclo che confronta il numero delle colonne;
- ciclo che conta il numero di valori;
- ciclo che inserisce la riga all'interno del file;

La funzione presenta un costo fortemente legato al numero di colonne della tabella, quindi:

$$T(n)= \Theta(m)$$

Dove m è il numero di colonne della tabella.

## bool select_all();

Funzione che richiama:

- put_query_results();
- extract_table();
- ciclo che legge tutto il file;

La prima funzione ha costo $\Theta(q)$, la seconda costante, quindi il costo è dato dal ciclo che legge tutto il file di testo, il quale effettivamente legge n righe.

Quindi, il costo prodotto dalla funzione è:

$$T(n)= \Theta(n)$$

Dove n è il numero di righe della tabella.

## bool select_columns();

Funzione che richiama:

- put_query_results();
- extract_table();
- make_structure();
- void view();
- free_structure();

Il costo dominante è dato dalle ultime tre funzioni elencate, le quali hanno lo stesso costo. Pertanto la funzione *select_columns()* avrà costo:

$$T(n) = \Theta(n*m)$$

Dove n è il numero di righe ed m il numero di colonne della tabella.

## bool select_where();

Funzione che richiama:

- put_query_results();

- ciclo che cerca la clausola WHERE ed estrae la condizione; extract_table();
- make_structure();
- clean_structure();
- view();
- free_structure();

Anche in questo caso il costo dominante è dato dalle ultime tre funzioni elencate. Pertanto la funzione *select_where()* avrà costo:

$$T(n) = \Theta(n*m)$$

Dove n è il numero di righe ed m il numero di colonne della tabella.

## bool select_order_by();

Il costo della funzione è dato principalmente dalle funzioni:

- make_structure(): Θ(nm)
- mergeSort(): Θ(nlogn)
- view(): Θ(nm)
- free_structure(): Θ(nm)

Il costo varia al variare del rapporto fra il numero di colonne e il numero di righe. Tanto più saranno le righe, rispetto alle colonne, tanto più il costo propenderà verso un valore O(nlogn); viceversa tanto più saranno le colonne rispetto alle righe, tanto più il costo propenderà verso il valore Θ(n*m).
Logn è ovviamente un'unita che tendenzialmente si manterrà sempre al di sotto di m in quanto una tabella con un milione di record avrà quasi sicuramente più di sei colonne quindi il costo è, di fatto:

$$T(n) = \Theta(n*m)$$

## bool select_group_by();

Funzione che richiama:

- put_query_results();
- strstr(); + ciclo che trova la colonna del group by
- strncpy();
- ciclo che verifica che la colonna sia la stessa della selezione
- memcpy();
- extract_table(); modificata
- make_structure();
- ciclo che cerca la colonna su cui fare il group by
- mergeSort();
- view(); modificata

- free_structure;

La funzione procede esattamente come la precendete: ordina la struttura e poi la stampa. Presenta quindi un costo di almeno Θ(n*m).

Ciò che cambia è l'uso di una *view*() modificata al fine di soddisfare la clausola. Questa prevede un costo pari a Θ(n) quindi il costo della funzione sarà:

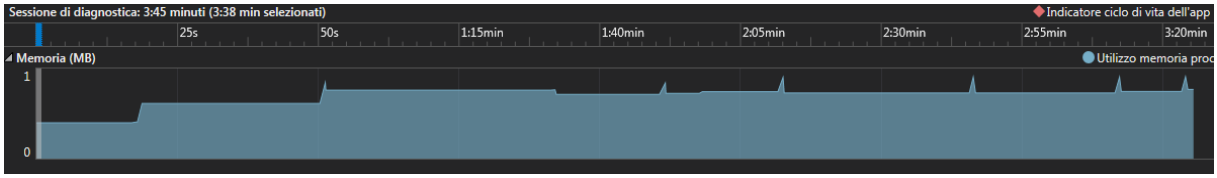
$$T(n) = \Theta(n*m)$$

# ANALISI DELL'IMPIEGO DI TEMPO E MEMORIA

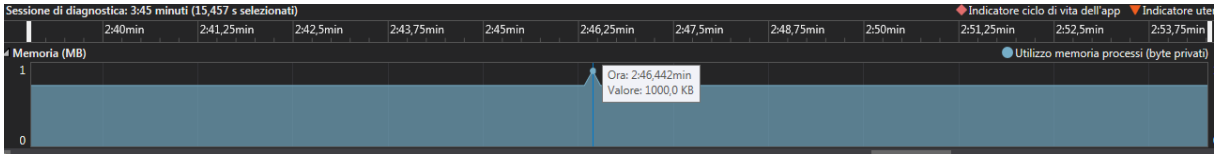
*Per le seguenti analisi è stata utilizzata una tabella di 4 colonne e 11 righe.*

Per quanto riguarda la memoria occupata dal programma abbiamo eseguito un'Analisi di Visual Studio, riscontrando che l'occupazione media di memoria si aggira intorno agli 800KB. Nelle richieste di query particolarmente onerose, la memoria occupata aumenta, ma genera un piccolo scarto rispetto al valore medio, 200KB circa, arrivando ad un'occupazione di 1000KB massimo.

Memoria globale impiegata in una esecuzione di 3 minuti circa:




Memoria occupata per una query onerosa:




Il tempo d'esecuzione delle query si aggira attorno a qualche centesimo di secondo.

Alcuni esempi di query eseguite: