Algoritmi e Strutture Dati

Documentazione relativa al progetto di laboratorio

Studente

Alfonsi Andrea

Matr. 0000766102

Data di consegna: 12/12/2018

Ambiente: Windows, Visual Studio 2015

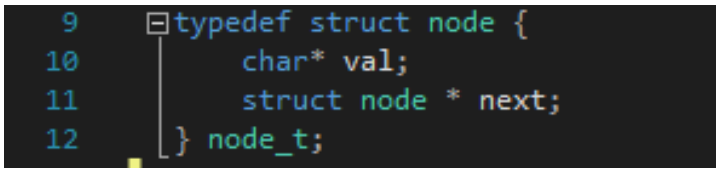
Indice

Strutture Dati pag.2

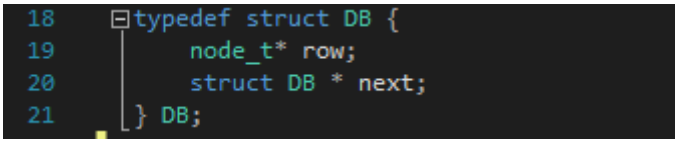
Funzioni e Algoritmi pag.3

Strutture dati utilizzate

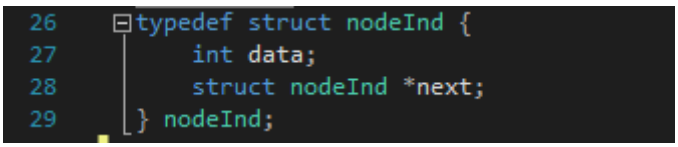
Le strutture dati create per completare questo progetto sono le seguenti:



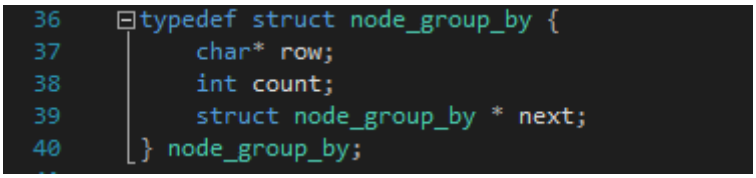
Struttura dati usata per rappresentare il valore di un attributo di un elemento del database.
Comprende un attributo 'val' che corrisponde al valore dell'elemento e un attributo 'next' che punta al successivo elemento della lista.



Struttura dati che rappresenta una 'riga' del database.
In particolare, ha un attributo 'row' che punta al primo elemento della riga e un attributo 'next' che punta alla riga successiva.



Semplice lista con una variabile 'data' di tipo intero e un puntatore all'elemento successivo. Durante lo svolgimento del progetto è stata particolarmente utile per memorizzare temporaneamente gli indici delle colonne richieste.

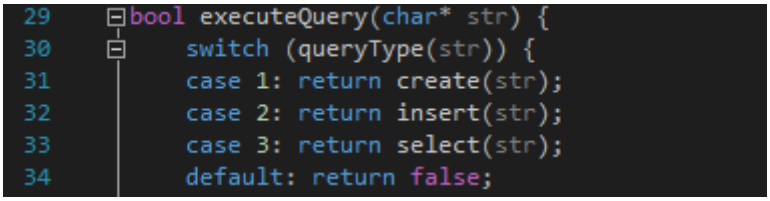


Particolare tipo di struttura dati usata per rappresenta il risultato di una query che utilizza il ‘group by’.
Contiene una variabile 'row' contenente il nome della riga, una variabile 'count' che indica il numero di volte che l'elemento ‘row’ è presente nel database e un puntatore all'elemento successivo.

Funzioni e algoritmi utilizzati

-executeQuery:

Funzione che prende in input una stringa contenente la query da eseguire e che restituisce ‘true’ solo se l’esecuzione della query è andata a buon fine.



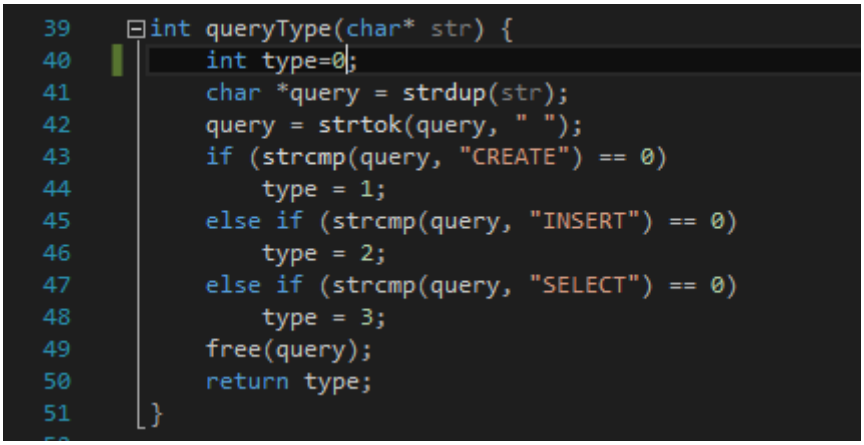
```
29  bool executeQuery(char* str) {
30      switch (queryType(str)) {
31      case 1: return create(str);
32      case 2: return insert(str);
33      case 3: return select(str);
34      default: return false;
```

Come è possibile vedere la funzione usa uno switch e richiama la funzione ‘queryType’ che a sua volta determina il tipo di query da eseguire.

In base al risultato richiamerà poi le rispettive funzioni.

-queryType:

Come detto in precedenza questa funzione ha il compito di determinare il tipo di query richiesta.



```
39  int queryType(char* str) {
40      int type=0;
41      char *query = strdup(str);
42      query = strtok(query, " ");
43      if (strcmp(query, "CREATE") == 0)
44          type = 1;
45      else if (strcmp(query, "INSERT") == 0)
46          type = 2;
47      else if (strcmp(query, "SELECT") == 0)
48          type = 3;
49      free(query);
50      return type;
51  }
```

Per farlo esegue una copia della variabile contenente la query, ne estrae la prima parola usando la funzione ‘strtok’ e resituisce quindi il tipo attraverso una variabile intera denominata ‘type’.

-insert:

Funzione che inserisce una nuova riga all’interno di una tabella.

Cicla tutte le parole della query salvando le informazioni utili come il nome della tabella ed effettuando vari controlli.

```
555     while (words != NULL) {
556         if (i == 2) tableName = strdup(words);
557         else if (i == 3) {
558             char* tmp = strdup(words);
559             tmp++;
560             tmp[strlen(tmp) - 1] = 0;
561             ptr = fopen(strcat(tableName, ".txt"), "r");
562             if (ptr == NULL) return 0;
563             columns = (char*)malloc((17 + strlen(tableName) + strlen(tmp)) * sizeof(char*));
564             fscanf(ptr, "%[^\n]", columns);
565             fclose(ptr);
566             //estrarre dalla prima riga le colonne
567             columns = strtok(columns, " ");
568             while (columns != NULL && j<4) {
569                 if (j == 3) {
570                     columns[strlen(columns) - 1] = 0;
571                     if (strcmp(tmp, columns) != 0) return false;
572                 }
573                 else columns = strtok(NULL, " ");
```

In particolare controlla che le colonne corrispondano e che il numero di elementi inseriti sia lo stesso.

```
602             //se è giusto inserisco in coda al file
603             if (i == j) {
604                 ptr = fopen(tableName, "a");
605                 if (ptr == NULL) return 0;
606                 fprintf(ptr, row);
607                 fclose(ptr);
608                 return true;
609             }
610             else return false;
```

Se non ci sono errori, la funzione procede con l'inserire la nuova riga generata nel file selezionato e restituisce 1.

-save_ds:

Questa funzione ha lo scopo di salvare tutti gli elementi di una tabella in una struttura dati per poi restituirla.

```
53  DB * save_ds(char * table_name) {
54      FILE * ptr = fopen(strcat(table_name, ".txt"), "r");
55      if (ptr == NULL) return 0;
56      int line_size = 300;
57      char * line = calloc(line_size, sizeof(char*));
58      table_name[strlen(table_name) - 4] = 0;
59      DB * DB_head = NULL;
60      DB_head = calloc(1, sizeof(DB));
61      if (DB_head == NULL) return 0;
62      DB * DB_current = DB_head;
63
64      //salto la prima riga
65      fscanf(ptr, "%*[^\n]");
66      char * a = fgetc(ptr);
67
```

Nella prima parte della funzione viene aperto il file, viene saltata la prima riga e inizializzata la struttura dati che andrà a contenere i dati.

```
68      //ciclo ogni riga
69      while (fgets(line, line_size, ptr) != NULL) {
70          char * current_line = strdup(line);
71          current_line[strlen(current_line) - 2] = 0; //rimuovo gli ultimi due caratteri della riga
72          current_line = strtok(current_line, " ");
73          current_line = strtok(NULL, " "); //salto la prima parola così da ottenere gli elementi da inserire nella struttura dati
74
75          current_line = strtok(current_line, ","); //suddivido gli elementi singolarmente
76          DB_current->row = calloc(1, sizeof(node_t));;
77          if (DB_current->row == NULL) return 0;
78          node_t * current_DB = DB_current->row;
79          while (current_line != NULL) { //inserisco gli elementi nella struttura dati
80              current_DB->val = current_line;
81              current_DB->next = calloc(1, sizeof(node_t));
82              current_DB = current_DB->next;
83              current_line = strtok(NULL, ",");
84          }
85          DB_current->next = calloc(1, sizeof(DB));
86          DB_current = DB_current->next;
87      }
88
89      return DB_head;
```

Nella seconda parte invece, viene ciclata ogni riga del file e per ognuna di esse se ne fa una copia che verrà poi ciclata a sua volta in modo da iterare tutti gli elementi di tale riga e da salvarli così nella struttura dati. Il costo sarà di conseguenza il numero di righe moltiplicato per il numero delle colonne.

Infine la struttura dati completa viene restituita.

-filter_columns:

Questa funzione è utilizzata quando è necessario lavorare solo con alcune delle colonne di una tabella.
Prende in input la tabella completa, le colonne da filtrare e il nome della tabella e in output restituisce la tabella filtrata.

```
92      DB * filter_columns(DB * db_head, char * columns1, char * table_name) {
93          char * columns = strdup(columns1);
94          char * words = strdup(columns);
95          columns = strtok(columns, ",");
96          node_t * head = malloc(sizeof(node_t));
97          if (head == NULL) return 0;
98          node_t * current = head;
99
100         //ciclo ogni parola dentro columns e la inserisco nella lista
101         while (columns != NULL) {
102             current->val = columns;
103             columns = strtok(NULL, ",");
104             if (columns != NULL) {
105                 current->next = malloc(sizeof(node_t));
106                 current = current->next;
107             }
108             else current->next = NULL;
109         }
```

Per prima cosa la funzione cicla le colonne e le inserisce in una lista.

```
111     //leggo dal file la prima riga e salvo gli indici
112     table_name = strcat(table_name, ".txt");
113     FILE * ptr = fopen(table_name, "r");
114     if (ptr == NULL) return 0;
115     columns = (char*)malloc((17 + strlen(table_name) + strlen(words)) * sizeof(char*));
116     fscanf(ptr, "%[^\n]", columns);
117     fclose(ptr);
118     columns += 11 + strlen(table_name);
119     table_name[strlen(table_name) - 4] = 0;
120     columns[strlen(columns) - 1] = 0;
121     //ho in head le colonne richieste dalla query, ho in columns le colonne della tabella
122
```

A questo punto il programma apre il file e legge la prima riga in modo da salvare tutte le colonne della tabella. Finito questo si avranno salvate in due strutture diverse sia le colonne richieste dalla query sia le colonne complete della tabella. Successivamente si andrà a creare una lista di tipo nodeInd che avrà lo scopo di contenere gli indici delle colonne richieste.

```
132     //while che salva gli indici delle colonne che mi interessano
133     while (current != NULL) {
134         char * dup_columns = strdup(columns);
135         dup_columns = strtok(dup_columns, ",");
136         cont = 0;
137         int changed = 0;
138         while (dup_columns != NULL && current != NULL) {
139             changed = 0;
140             if (strcmp(current->val, dup_columns) == 0) {
141                 if (curr_q->data != -1) {
142                     curr_q->next = calloc(1, sizeof(nodeInd));
143                     curr_q = curr_q->next;
144                 }
145                 curr_q->data = cont;
146                 changed = 1;
147             }
148             cont++;
149             if (changed == 1)
150                 current = current->next;
151             dup_columns = strtok(NULL, ",");
152         }
153     }
```

Nell'immagine precedente è possibile vedere il ciclo che salva nella nuova lista gli indici delle colonne interessate.

```
161     while (db_current->row != NULL) {
162         curr_q = q;
163         cont = 0;
164         node_t * old_node=db_current->row;
165         node_t * new_node = new_db_current->row;
166         while (old_node != NULL && curr_q!=NULL) {
167             if (curr_q->data == cont) {
168                 new_node->val = old_node->val;
169                 new_node->next = calloc(1, sizeof(node_t));
170                 if (new_node->next == NULL) return 0;
171                 new_node= new_node->next;
172                 curr_q = curr_q->next;
173                 cont = 0;
174                 old_node = db_current->row;
175             }
176             old_node = old_node->next;
177             cont++;
178         }
179         db_current = db_current->next;
180         new_db_current->next = calloc(1, sizeof(DB));
181         new_db_current = new_db_current->next;
182         new_db_current->row = calloc(1, sizeof(node_t));
183     }
```

Infine si andrà a creare una nuova struttura dati di tipo ‘DB’ che conterrà la nuova tabella filtrata.

Il costo nel caso peggiore sarà pari al numero di righe nella tabella moltiplicato per il numero di colonne nella vecchia tabella mentre nel caso migliore sarà uguale al numero di righe per il numero di colonne nella nuova tabella e quindi varierà in base agli indici delle colonne nella nuova tabella.

Successivamente sarà restituita la tabella filtrata.

-operator_to_use:

```
187  int operator_to_use(char * operation) {
188      int type = 0;
189      if (strcmp(operation, "==") == 0)
190          type = 1;
191      else if (strcmp(operation, ">") == 0)
192          type = 2;
193      else if (strcmp(operation, ">=") == 0)
194          type = 3;
195      else if (strcmp(operation, "<") == 0)
196          type = 4;
197      else if (strcmp(operation, "<=") == 0)
198          type = 5;
199      return type;
200  }
```

Questa semplice funzione restituisce un numero che corrisponde all’operazione ricevuta in input ed è utilizzata quando è presente il filtro ‘WHERE’ nella query.

-get_columns:

```
202  char * get_columns(char * table_name) {
203      FILE * ptr = fopen(strcat(table_name, ".txt"), "r");
204      if (ptr == NULL) return 0;
205      char * first_row = calloc(100, sizeof(char));
206      fscanf(ptr, "%[^\n]", first_row);
207      fclose(ptr);
208      first_row += 11 + strlen(table_name);
209      first_row[strlen(first_row) - 1] = 0;
210      table_name[strlen(table_name) - 4] = 0;
211      return first_row;
212  }
```

Questa funzione legge la tabella ricevuta in input, ne salva la prima riga e la filtra in modo da poter restituire tutte le colonne di tale tabella.

-where:

Questa funzione applica il filtro where restituendo poi la tabella filtrata.

```
214  DB * where(DB * db_head, char * elem_1, char * elem_2, char * operation, char * table_name) {
215      //trovo l'indice della colonna a cui devo fare il controllo
216      int cont = 0, index=-1;
217      char * columns = get_columns(strdup(table_name));
218      columns = strtok(columns, ",");
219      while (columns != NULL) {
220          if (strcmp(columns, elem_1) == 0) {
221              index = cont;
222              break;
223          }
224          cont++;
225          columns = strtok(NULL, ",");
226      }
227
```

Per prima cosa la funzione trova l'indice della colonna su cui effettuare il controllo e ne salve l'indice in una variabile denominata 'index',

Successivamente andrà a ciclare tutte le righe della tabella e per ognuna di esse controllerà se il filtro è soddisfatto richiamando la funzione 'operator_to_use'.

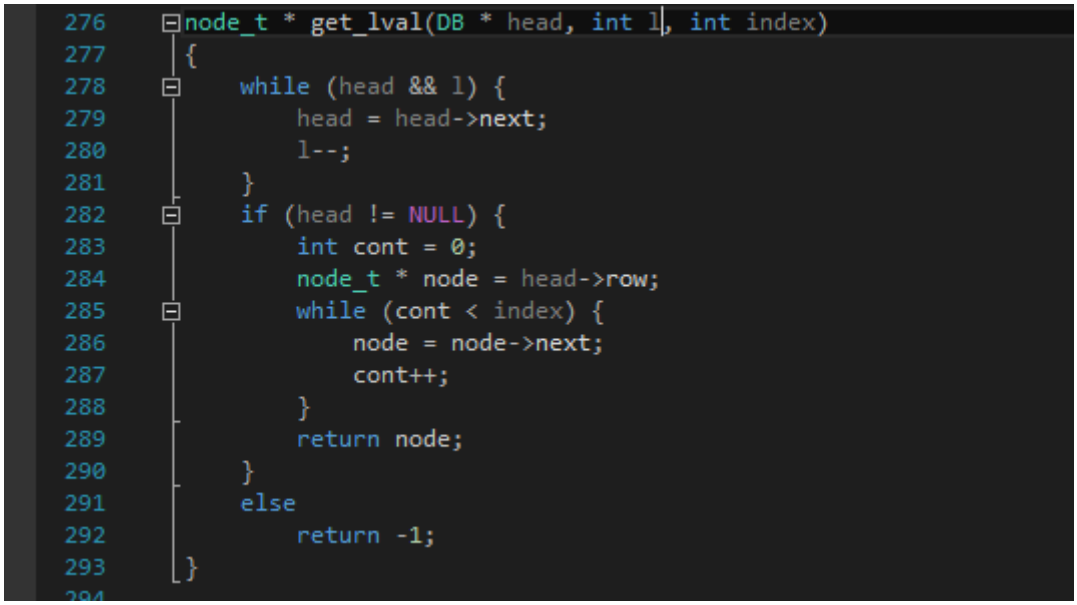
```
242              int satisfied = 0;
243              switch (operator_to_use(operation)) {
244              case 1: if (strcmp(current_node->val, elem_2) == 0)
245                  satisfied = 1;
246                  break;
247              case 2: if (strcmp(current_node->val ,elem_2)>0)
248                  satisfied = 1;
249                  break;
250              case 3: if (strcmp(current_node->val , elem_2)>=0)
251                  satisfied = 1;
252                  break;
253              case 4: if (strcmp(current_node->val , elem_2)<0)
254                  satisfied = 1;
255                  break;
256              case 5: if (strcmp(current_node->val , elem_2)<=0)
257                  satisfied = 1;
258                  break;
259              default: return 0;
260                  break;
261          }
```

In caso il filtro sia soddisfatto si andrà ad aggiungere la riga corrente alla nuova tabella ed infine si restituirà.

```
263      //se il filtro è rispettato, aggiungo la riga corrente al nuovo db
264      if (satisfied == 1) {
265          current_new_db->row = db_current->row;
266          current_new_db->next = calloc(1, sizeof(DB));
267          current_new_db = current_new_db->next;
268      }
269      db_current = db_current->next;
270      complete_db = complete_db->next;
271      }
272
273      return new_db;
274  }
275
```

-get_lval:

Questa funzione è utilizzata nel caso la query richieda un 'ORDER BY' in quanto l'algoritmo scelto per l'ordinamento è stato il 'quicksort' ed in particolare serve per restituire il valore di un particolare nodo della tabella.



```
276  node_t * get_lval(DB * head, int l, int index)
277  {
278      while (head && l) {
279          head = head->next;
280          l--;
281      }
282      if (head != NULL) {
283          int cont = 0;
284          node_t * node = head->row;
285          while (cont < index) {
286              node = node->next;
287              cont++;
288          }
289          return node;
290      }
291      else
292          return -1;
293  }
```

Cicla le righe fino a trovare quella con l'indice richiesto e dopodiché scorre le colonne di tale riga fino a trovare quella desiderata andando poi a restituirla.

Il costo nel caso peggiore può essere il numero di righe sommato al numero di colonne mentre nel caso migliore è 1.

-swap:

Questa funzione è utilizzata durante l'ordinamento nel caso di una query con filtro 'ORDER BY' e serve a cambiare la posizione di due righe della tabella.

```c
void swap(DB * head, int i, int j) {
    DB * tmp = head;
    node_t * tmpival;
    node_t * tmpjval;
    int ti = i;
    while (tmp && i) {
        i--;
        tmp = tmp->next;
    }
    tmpival = tmp->row;
    tmp = head;
    while (tmp && j) {
        j--;
        tmp = tmp->next;
    }
    tmpjval = tmp->row;
    tmp->row = tmpival;
    tmp = head;
    i = ti;
    while (tmp && i) {
        i--;
        tmp = tmp->next;
    }
    tmp->row = tmpjval;
}
```

La funzione comincia ciclando la tabella fino ad arrivare alla prima riga desiderata e la salva un una variabile ‘tmpival’. Poi fa la stessa cosa con la seconda riga e infine ne cambia la posizione.

-quick_sort_list:

```c
DB * quick_sort_list(DB * head, int l, int r, int index, char * order) {
    int i, j;
    char * jval;
    char * pivot;
    i = l + 1;
    if (l + 1 < r) {
        pivot = get_lval(head, l, index)->val; //ricavo il valore per il pivot
        for (j = l + 1; j < r; j++) {
            jval = get_lval(head, j, index)->val;
            if (strcmp(order, "ASC") == 0) {
                if (strcmp(jval, pivot) < 0 && jval != -1) {
                    swap(head, i, j);
                    i++;
                }
            }
            else if (strcmp(order, "DESC") == 0) {
                if (strcmp(jval, pivot) > 0 && jval != -1) {
                    swap(head, i, j);
                    i++;
                }
            }
        }
        swap(head, i - 1, l);
        quick_sort_list(head, l, i, index, order);
        quick_sort_list(head, i, r, index, order);
    }
    return head;
```

Questa funzione esegue l'ordinamento della tabella utilizzando un pivot e mettendo gli elementi più piccoli alla sua sinistra e quelli più grandi alla sua destra (o viceversa nel caso di ordinamento decrescente) facendo anche uso della funzione 'swap' per invertire la posizione delle righe.

Dopo aver ciclato tutti gli elementi ed aver ultimato l'ordinamento restituisce la tabella ordinata.

-order-by:

Per prima cosa, questa funzione trova l'indice della colonna in base alla quale effettuare l'ordinamento.

```
350  DB * order_by(DB * db_head, char * condition, char * order, char * table_name
351      //trovo l'indice della colonna sul quale fare l'ordinamento
352      int cont = 0, index = -1;
353      char * columns = get_columns(strdup(table_name));
354      columns = strtok(columns, ",");
355      while (columns != NULL) {
356          if (strcmp(columns, condition) == 0) {
357              index = cont;
358              break;
359          }
360          cont++;
361          columns = strtok(NULL, ",");
362      }
```

Successivamente conta il numero di elementi(righe) e poi lancia il *quicksort* passandogli tutte le variabili necessarie.

```
364      //conto il numero di elementi
365      DB * tmp = db_head;
366      int n = 0;
367      while (tmp->row) {
368          n++;
369          tmp = tmp->next;
370      }
371      //uso il quick sort
372      db_head = quick_sort_list(db_head, 0, n, index, order);
373
374      return db_head;
```

Infine restituisce la tabella ordinata.

-group_by:

Questa funzione ha lo scopo di effettuare l'operazione di 'group by'. Per farlo comincia eliminando tutte le colonne che non gli interessano (attraverso la funzione 'filter_columns') e creando la nuova struttura dati di tipo 'node_group_by'.

```
377  DB * group_by(DB * db_head, char * condition, char * table_name) {
378
379      //elimino le colonne che non mi interessano
380      db_head = filter_columns(db_head, condition, table_name);
381      //ciclo tutto il db
382      DB * tmp = db_head;
383      node_group_by * new_db = calloc(1,sizeof(node_group_by));
384      node_group_by * new_tmp;
385      int first = 1;
```

Successivamente cicla tutto la tabella e per ogni riga controlla se un elemento con lo stesso valore è gia presente nella nuova struttura dati e in tal caso ne aumenta il valore ‘count’. Altrimenti aggiunge un nuovo nodo alla struttura dandogli valore ‘count’ uguale ad 1.

```
386      while (tmp != NULL && tmp->row->val!=NULL) {
387          new_tmp = new_db;
388          while (new_tmp->row != NULL && first==0){
389              if (strcmp(tmp->row->val, new_tmp->row) == 0) {
390                  new_tmp->count++;
391                  break;
392              }
393              new_tmp = new_tmp->next;
394           }
395          if (new_tmp == NULL || new_tmp->row==NULL) {
396              new_tmp->row = tmp->row->val;
397              new_tmp->count++;
398              new_tmp->next = calloc(1, sizeof(node_group_by));
399          }
400          first = 0;
401          tmp = tmp->next;
402      }
403      return new_db;
```

-print_query_gb:

Funzione con lo scopo di stampare nel file ‘query_results’ il risultato di una query che comprende l’operazione di ‘group by’.

```
406  void print_query_gb(node_group_by * db_head_groupby, char * query, char * table_name, char * column) {
407      FILE * ptr = fopen("query_results.txt", "a");
408      if (ptr == NULL) return 0;
409      fprintf(ptr, "%s;\nTABLE %s COLUMNS %s,COUNT;\n", query, table_name, column);
410      while (db_head_groupby->row != NULL) {
411          fprintf(ptr, "ROW %s,%d;\n", db_head_groupby->row, db_head_groupby->count);
412          node_group_by * tmp = db_head_groupby;
413          db_head_groupby = db_head_groupby->next;
414          free(tmp);
415      }
416      fprintf(ptr, "\n");
417      fclose(ptr);
418  }
```

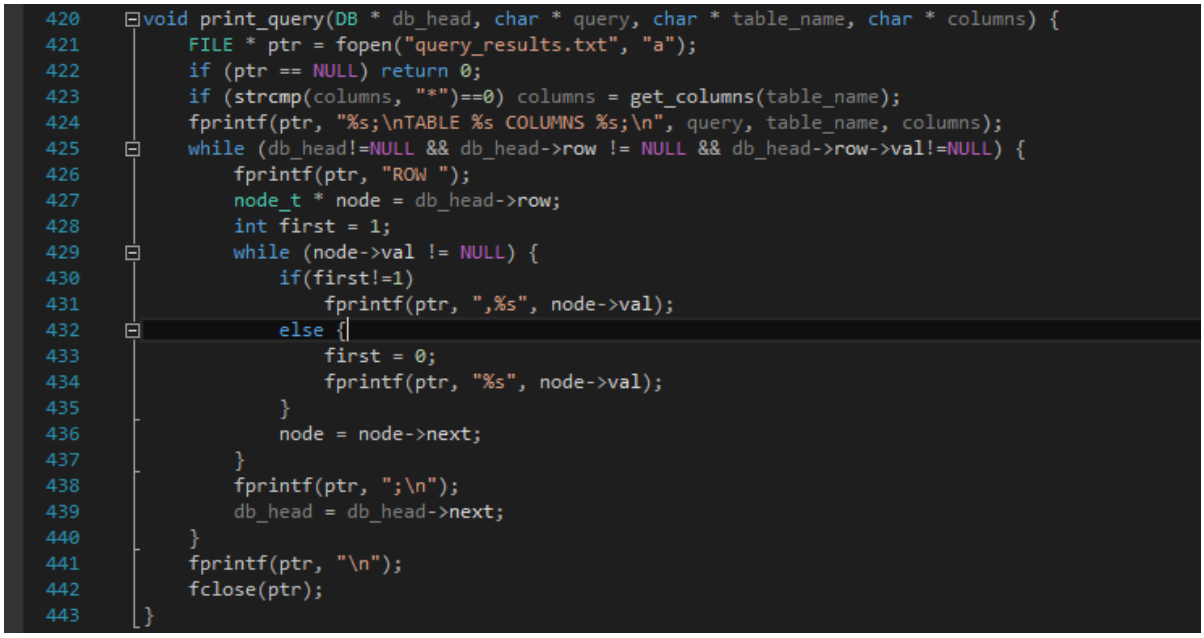
Una volta aperto il file ci stampa la prima riga e poi cicla la struttura dati per tutte le righe seguenti; di conseguenza il costo è pari al numero di nodi nella struttura (che a sua volta equivale al numero di elementi distinti nella colonna della tabella sulla quale si è applicata la query di ‘group_by’).

-print_query:

Questa funzione stampa il risultato di una query (tranne le query con il filtro 'group by').

Per farlo comincia aprendo il file e stampando la prima riga. Nel caso nella query vi fosse un 'SELECT *' le colonne devono essere ricavate attraverso la funzione 'get_columns'.

Dopodiché cicla tutta la struttura dati stampando tutte le sue righe.

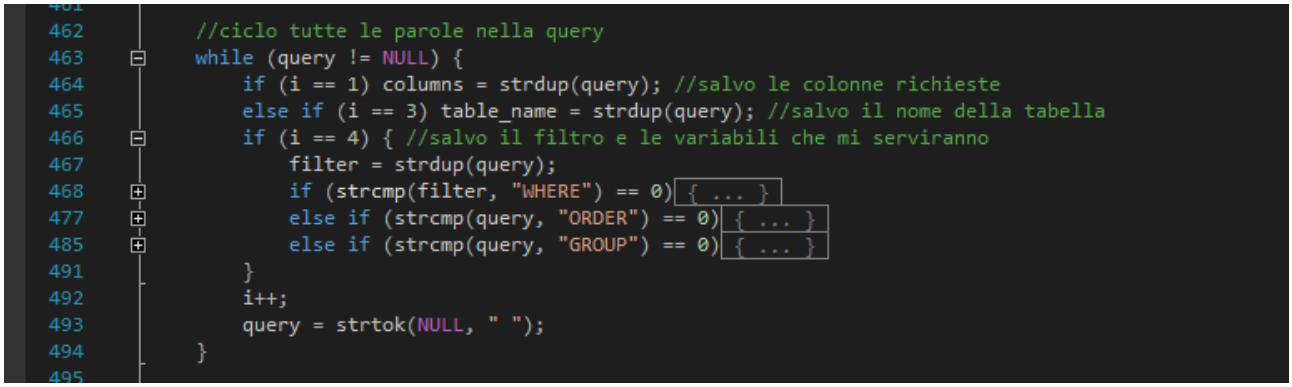


```
420  void print_query(DB * db_head, char * query, char * table_name, char * columns) {
421      FILE * ptr = fopen("query_results.txt", "a");
422      if (ptr == NULL) return 0;
423      if (strcmp(columns, "*")==0) columns = get_columns(table_name);
424      fprintf(ptr, "%s;\nTABLE %s COLUMNS %s;\n", query, table_name, columns);
425      while (db_head!=NULL && db_head->row != NULL && db_head->row->val!=NULL) {
426          fprintf(ptr, "ROW ");
427          node_t * node = db_head->row;
428          int first = 1;
429          while (node->val != NULL) {
430              if(first!=1)
431                  fprintf(ptr, ",%s", node->val);
432              else {
433                  first = 0;
434                  fprintf(ptr, "%s", node->val);
435              }
436              node = node->next;
437          }
438          fprintf(ptr, ";\n");
439          db_head = db_head->next;
440      }
441      fprintf(ptr, "\n");
442      fclose(ptr);
443  }
```

-select:

La funzione select esegue le query di ricerca all'interno delle tabelle.

Per prima cosa cicla tutte le parole della query salvando varie informazioni come il nome della tabella, le colonne interessate dalla query ed eventuali filtri.



```
462      //ciclo tutte le parole nella query
463      while (query != NULL) {
464          if (i == 1) columns = strdup(query); //salvo le colonne richieste
465          else if (i == 3) table_name = strdup(query); //salvo il nome della tabella
466          if (i == 4) { //salvo il filtro e le variabili che mi serviranno
467              filter = strdup(query);
468              if (strcmp(filter, "WHERE") == 0) { ... }
477              else if (strcmp(query, "ORDER") == 0) { ... }
485              else if (strcmp(query, "GROUP") == 0) { ... }
491          }
492          i++;
493          query = strtok(NULL, " ");
494      }
```

A questo punto gestisco gli eventuali filtri richiamando le rispettive funzioni e inviandogli tutte le variabili necessarie.

```
496     //inserisco tutti gli elementi della tabella richiesta in una struttura dati
497     DB * db_head = save_ds(table_name);
498     node_group_by * db_head_groupby=NULL;
499     //gestisco filtri
500     switch (type) {
501     case 1: db_head = where(db_head, elem_1, elem_2, operation, table_name);
502         break;
503     case 2: db_head = order_by(db_head, condition, order, table_name);
504         break;
505     case 3: db_head_groupby = group_by(db_head, condition, table_name);
506         break;
507     }
```

Infine, se necessario, filtro la colonna e poi stampo il risultato della query nel file.

```
509     //rimuovo eventuali colonne non richieste
510     if (strcmp(columns, "*") != 0 && type!=3)
511         db_head=filter_columns(db_head, columns, table_name);
512
513     //stampo il risultato della query nel file
514     if (type != 3)
515         print_query(db_head, str, table_name, columns);
516     else
517         print_query_gb(db_head_groupby, str, table_name, condition);
518
519     return true;
```

-create:

Questa funzione serve a creare una nuova tabella. Una volta ricevuta la query, ogni parola al suo interno viene ciclata salvando inoltre informazioni come il nome della tabella e le colonne. Infine crea la tabella e ne genera la prima riga.

```
522 int create(char* str) {
523     int i = 0;
524     char *tableName = NULL;
525     str = strtok(str, " ");
526     while (str != NULL) {
527         if (i == 2) {
528             tableName = strdup(str);
529         }
530         else if (i == 3) {
531             str++;
532             str[strlen(str) - 1] = 0;
533             FILE* ptr;
534             ptr = fopen(strcat(tableName, ".txt"), "w");
535             if (ptr == NULL) return 0;
536             tableName[strlen(tableName) - 4] = 0;
537             fprintf(ptr, "TABLE %s COLUMNS %s;\n", tableName, str);
538             fclose(ptr);
539
540             return 1;
541         }
542         str = strtok(NULL, " ");
543         i++;
544     }
545     return false;
546 }
```