SbornDB – documentazione

# 0. Indice

SbornDB

[0. Indice 1](#__RefHeading___Toc1150_3069259507)

[1. Premessa 2](#__RefHeading___Toc1025_3069259507)

[2. Gestione della memoria 2](#__RefHeading___Toc1027_3069259507)

[2.1 Premessa 2](#__RefHeading___Toc1029_3069259507)

[2.2 Assunzioni 2](#__RefHeading___Toc1031_3069259507)

[2.3 Scopo 2](#__RefHeading___Toc1033_3069259507)

[2.4 Definizione delle caratteristiche 2](#__RefHeading___Toc1035_3069259507)

[2.5 Descrizione 3](#__RefHeading___Toc1037_3069259507)

[3. Strutture dati 4](#__RefHeading___Toc1039_3069259507)

[4. Corpo centrale 5](#__RefHeading___Toc1152_3069259507)

[4.1 Fase di Inizializzazione 5](#__RefHeading___Toc1467_3069259507)

[4.2 Fase di Parsing 5](#__RefHeading___Toc1469_3069259507)

[4.3 Fase di Ricerca 5](#__RefHeading___Toc1471_3069259507)

[4.4 Fase di Esecuzione 6](#__RefHeading___Toc1473_3069259507)

[4.4.1 Fase di Controllo 6](#__RefHeading___Toc1475_3069259507)

[4.4.2 Creazione 6](#__RefHeading___Toc1477_3069259507)

[4.4.3 Inserimento 6](#__RefHeading___Toc1479_3069259507)

[4.4.4 Selezione 6](#__RefHeading___Toc1481_3069259507)

[4.5 Fase di Liberazione della memoria 6](#__RefHeading___Toc1483_3069259507)

[5. Parser 6](#__RefHeading___Toc1043_3069259507)

# 1. Premessa

In questo file sono racchiuse tutte le specifiche sul funzionamento del database, sui suoi costi in termini di velocità e di memoria.

Il progetto è diviso in diverse parti: gestione della memoria, strutture dati, corpo centrale e parser.

Di seguito ogni parte verrà analizzata nel dettaglio. Infine vi è una sezione dedicata all’analisi dei costi computazionali in ordine di tempo e spazio richiesto.

# 2. Gestione della memoria

## 2.1 Premessa

Data la grande quantita’ di dati che possono essere immagazzinati in un database reale, si e’ pensato di gestire questo problema (in maniera notevolmente ridotta e parziale) anche nel progetto.

## 2.2 Assunzioni

Si e’ assunto che i file di testo contenenti le tabelle dovessero rimanere formattati come da specifiche di progetto.

Inoltre, si e’ assunto che non potessimo creare altri file per salvare le strutture dati usate per consultare le tabelle.

Detto questo, e’ stato deciso di creare una sistema per la gestione della memoria utilizzata dal programma.

## 2.3 Scopo

- Cercare di limitare l’utilizzo della memoria da parte del programma in modo da non sovraccaricare la RAM e quindi generare un conseguente calo di prestazioni di tutto il sistema

- Poter comunque utilizzare tutte le funzionalita’ presenti nelle specifiche di progetto

- Mantenere efficienza in termini di tempo di esecuzione

## 2.4 Definizione delle caratteristiche

- Utilizzo della memoria “heap” per memorizzare le tabelle durante l’esecuzione del programma (colonne, dati, strutture, etc…)

- Permettere la memorizzazione di almeno una tabella per volta, per poterci eseguire query e inserimenti

- Definire una soglia massima di memoria utilizzata, sopra la quale non si puo’ piu’ allocare ma bisogna liberare memoria

- Limitare la allocazione/deallocazione di memoria utilizzando un sistema di cacheing basilare

## 2.5 Descrizione

Si e’ pensato di imitare in funzionamento di un sistema di cacheing temporale, ovvero assegnare delle priorita’ alle tabelle ed eliminare, in caso di eccessivo uso della memoria, quelle che non vengono utilizzate da piu’ tempo.

Ovviamente, questo sistema basilare puo’ non rivelarsi ottimo a livello di quantita’ di allocazione/deallocazione, ma e’ molto piu’ probabile che tutte le query/inserimenti fatti su una tabella vengano eseguiti uno dopo l’altro piuttosto che distanziati.

Il sistema e’ stato implementato come un heap con update dinamico. Come chiave si e’ utilizzato in intero (esso descrive il numero della operazione eseguita ed cresce con il numero delle operazioni creazioni/inserimenti/query) e come valore la tabella stessa. Il funzionamento e’ simile a una coda di priorita’ con update dinamico.

Viene anche tenuto traccia della quantita’ di memoria utilizzata dal programma per gestire le tabelle.

Quando una tabella viene creata, questa viene allocata e inserita nella coda. Siccome l’intero che rappresenta la chiave dell’heap descrive l’istante in cui la tabella viene usata, questa tabella avra’ la chiave maggiore di tutte le altre all’interno dell’heap.

La coda di priorita’ mantiene in testa la tabella piu’ “vecchia” ossia quella utilizzata meno di recente ossia quella con chiave minore.

Quando viene fatta una qualsiasi operazione su una tabella gia’ esistente nell’heap, la sua chiave viene aggiornata (update dinamico).

Quando deve essere allocata memoria per una tabella ma non c’e’ abbastanza spazio per farlo, viene eliminata la tabella in cima alla coda fino a che non rimane abbastanza spazio per quella che deve essere allocata in quel momento.

Siccome l’heap ha bisogno della proprieta’ di update dinamico (con complessita’ logaritmica) e’ necessario un riferimento uno ad uno con tra elemento nell’heap e tabella corrispondente (puntatore all tabella nell’elemento dell’heap e puntatore all’elemento dell’heap nella tabella).

L’heap e’ una struttura dati vista a lezione, e’ stata aumentata per gestire l’update dinamico in O(log n).

Per la implementazione si e’ preso spunto da un paper redatto da O. Tamir, A. Morrison and N. Rinetzky (link al paper https://www.cs.tau.ac.il/~mad/publications/opodis2015-heap.pdf ), osservando in particolare la sezione numero 3 : “A Sequential Heap with Mutable Priorities”

# 3. Strutture dati

In questa sezione sono descritte le varie strutture dati utilizzate e come sono collegate tra loro.

Le principali strutture dati utilizzate sono:

* Linked list
* Red Black Tree
* Heap con Update Dinamico

La struttura delle Linked List non varia dalla regolare implementazione.

I Red Black Trees sono personalizzati per il progetto. Nella struttura RBTree è presente un puntatore alla radice dell’albero ``` struct RBTNode\* root; ``` e una chiave ``` int key; ```. La chiave può assumere due diversi significati: può indicare che sia un albero di tabelle o un albero di puntatori a record. Nel primo caso è indicato assumendo valore -2, mentre nel secondo caso assume valori tra 0 e MAX\_INT, che indicano l’indice della colonna di ogni tabella. Questa variabile non assume mai valore -1 in quanto è il valore ritornato dalla funzione “searchColumnIndex()” nel caso in cui la colonna richiesta non è stata trovata.

Per la implementazione dell’heap si rimanda alla precedente sezione “Gestione della Memoria”.

*Il database è accessibile da un unico puntatore dichiarato globale.* È stata fatta questa scelta in quanto si assume di lavorare sempre sullo stesso database, seguendo le linee guida del progetto. In questo modo si evita di passare il puntatore al database in ogni funzione alleggerendo lo stack di sistema.

La struttura Database è un Red Black Tree nei quali nodi sono salvate le tabelle caricate in memoria.

La struttura RBTNode contiene i puntatori di una normale implementazione di un nodo di un red black tree: colore, testa dell’albero, padre, figlio destro e figlio sinistro. Inoltre contiene il puntatore ```void \* nodeValue;```, il quale contiene o la tabella o il puntatore al record a cui si riferisce il nodo stesso (come riportato precedentemente è possibile capire il tipo del puntatore nodeValue a seconda della chiave nella testa dell’albero).

La struttura tabella (```struct TableDB```) contiene: il nome della tabella (“name”), i nomi delle colonne (“columns”), il numero delle colonne (“nColumns”), la lista di record (“recordList”) e l’array di alberi di puntatori ai record che permettono di mantenere ordinati i record per ogni colonna. Infine contiene un puntatore heapReference !!!—GIACK—!!!.

È utile approfondire, nella struttura tabella, il funzionamento degli alberi di puntatori ai record e della lista di record. È oltretutto necessario chiarire che i record sono salvati fisicamente solo nella lista di record. Per ogni colonna della tabella è presente un red black tree, che tiene ordinato ogni record per ogni colonna della tabella. Nel momento dell’inserimento di un record, infatti, si inserisce un nodo RBT in ogni albero nella tabella, contenente il puntatore al nuovo record inserito in recordList.

Le rimanenti strutture dati sono alquanto facili da comprendere, ma se ne riporta una breve spiegazione in seguito.

```struct ParseResult``` ha un campo per ogni possibile informazione di cui può essere necessario fare il parse e un campo di controllo (```bool success```) che indica se esso è andato a buon fine o meno.

```struct QueryResultElement``` è una struttura che serve a contenere il risultato delle query. È una semplice linked list in cui c’è il campo “occurrence” utilizzato nel groupby per capire quante volte si ripete un elemento e il campo nodeValue punta al record a cui si vuol riferire questo nodo della lista.

# 4. Corpo centrale

Il corpo centrale del main e’ diviso in tre parti principali:

1. Parsing della stringa
2. Ricerca tabella nel database (ed eventuale caricamento in memoria)
3. Esecuzione della query/creazione/inserimento

Inoltre sono presenti tre fasi minori:

1. Fase di inizializzazione (eseguita alla prima chiamata della funzione)
2. Fase di controllo (eseguita prima della esecuzione per controllare la query)
3. Fase di liberazione della memoria (eseguita alla fine di ogni chiamata della funzione)

## 4.1 Fase di Inizializzazione

Viene inizializzata la struttura che conterra’ il database e l’heap per la gestione della memoria.

Viene eseguita solo una volta, alla prima chiamata della funzione “executeQuery” e poi viene saltata.

## 4.2 Fase di Parsing

Viene chiamata la funzione “parseQuery” che, appunto, fa il parsing della query passata per argomento alla funzione “executeQuery”.

Essa ritorna una struttura chiamata “ParseResult” che contiene il risultato del parsing (a buon fine o meno) e tutte le informazione necessarie per eseguire la query/creazione/inserimento.

Se la queryString non era correttamente formattata (quindi il parse non ha successo), viene interrotta l’esecuzione con ritorno a false.

## 4.3 Fase di Ricerca

Viene ricercata la tabella sul cui si vuole lavorare nel database tramite la funzione “searchTableDb”. Se viene trovata, viene aggiornata la sua chiava nellla coda di priorita’ per la gestione della memoria.

Altrimenti si prova a caricarla da file con la funzione “loadTableFromFile”.

Questa fase rilascia una struttura di tipo “Table” che puo’ contenere o no una tabella in base al successo del caricamento.

## 4.4 Fase di Esecuzione

Viene eseguita la query/creazione/inserimento vera e propria.

### 4.4.1 Fase di Controllo

Prima di tutto, viene controllata la query/creazione/inserimento ora che si conoscono tutte le informazione della tabella a cui si sta facendo riferimento (se e’ stata caricata in memoria dalla fase precedente).

Dopodiche’ la query viene eseguita in base al suo tipo.

### 4.4.2 Creazione

Se la tabella non esiste (quindi non era stata caricata), viene creata. Prima viene generato il file e poi viene creata la struttura della tabella nel database.

### 4.4.3 Inserimento

Se la tabella esiste (quindi e’ stata caricata), viene eseguito l’inserimento di un record. Prima viene aggiornato il file e poi aggiunto il record nella struttura della tabella a database.

### 4.4.4 Selezione

Se la tabella esiste, viene eseguita la selezione. Attraverso la funzione “querySelect” viene creato una struttura di tipo “QueryResultList” contenente il risultato della query.

Dopodiche’ viene generato il log della query eseguita, chiamando la funzione “generateLog”.

Viene poi liberata la memoria contenente la struttra della “QueryResultList”.

## 4.5 Fase di Liberazione della memoria

Vengono liberate tutte le strutture utilizzate per la query in corso. Nel nostro caso solo una che rimane ancora non liberata, ossia il “ParseResult”

# 5. Parser

Il parser deve gestire un numero limitato di possibili query, ed ha quindi una struttura molto semplice.

Tutta l'operaIone di parsing viene eseguita in tempo lineare rispetto alla lunghezza della query.

L'entry point del parser e' la funzione `ParseResult parseQuery (char \* query)`, dalla quale vengono chiamate varie funzioni per capire il tipo di query e farne l'analisi. Durante l'esecuzione viene usato direttamente un puntatore `char \*`, che segue il progresso lungo la query tramite degli incrementi.

La funzione `parseQuery` restituisce un puntatore `ParseResult` alla `struct ParseResult`, cosi' composta:

```

struct ParseResult {

bool success;

char \* tableName;

int queryType;

char \*\* columns;

int nColumns;

char \*\* fieldValues;

int querySelector;

char \* keyName;

char \* key;

int order;

int parseErrorCode;

};

```

dove:

1. `bool success` contiene il risultato dell'analisi, e di conseguenza la validita' sia della query che dei dati nella struttura

2. `char \* tableName` contiene il nome della tabella su cui agisce la query

3. `int queryType` contiene il codice numerico identificativo del tipo di query, tra i possibili seguenti:

a. Create Table -1

b. Select con filtro Where 0

c. Select con filtro Order By 1

d. Select con filtro Group By 2

e. Insert Into 3

f. Select senza filtri 4

g. No Query 6

4. `char \*\* columns` contiene i puntatori di tipo `char \*` ai nomi delle colonne interessate dalla query.

5. `int nColumns` contiene il numero di colonne interessate dalla query, e di conseguenza anche il numero di puntatori presenti dentro il campo `char \*\* columns`.

6. `char \*\* fieldValues` usata solo nelle query di tipo Insert Into, contiene `nColumns` puntatori di tipo `char \*` alle stringhe contenenti i valori da inserire

7. `int querySelector` usato solo in query di tipo Select con filtro Where, contiene il codice numerico identificativo dell'operatore:

a. Equal 0

b. Greater 1

c. Lesser 2

d. Greater Equal 3

e. Lesser Equal 4

f. No Operator 5

8. `char \* keyName` usato nelle query di tipo:

a. Select con filtro Where, per contenere il nome del campo interessato dalla condizione where.

b. Select con filtro Group By, per contenere il nome del campo per cui raggruppare i record.

c. Select con filtro Order By, per contenere il nome del campo per cui ordinare i record.

9. `char \* key` usato solo nella query di tipo Select con filtro Where, per contenere il valore da paragonare quando si verifica la condizione della query.

10. `int order` usato solo nelle query di tipo Select con filtro Order By, contiene l'ordine desiderato. Puo' assumere i valori:

a. ASC 0

b. DESC 1

11. `int parseErrorCode` contiene un identificativo numerico hardcoded nel sorgente, unico al punto in cui il parse ha fallito l'esecuzione e terminato prematuramente. Controllando il codice si puo' facilmente risalire al punto in cui si e' manifestato il problema. I codici di errore sono suddivisi in classi:

a. 0-99 : Errori Generali

b. 101-199: Errori durante il parsing di una query "Create Table"

c. 201-299: Errori durante il parsing di una query "Insert Into"

d. 301-399: Errori durante il parsing della parte senza filtri di una query "Select"

e. 401-499: Errori durante il parsing della parte finale di una query "Select Where"

f. 501-599: Errori durante il parsing della parte finale di una query "Select Group By"

g. 601-699: Errori durante il parsing della parte finale di una query "Select Order By"