SbornDB – documentazione

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: gestione della memoria, strutture dati, corpo centrale e parser. Ogni parte verrà analizzata nel dettaglio e alla fine verranni discussi i costi computazionali di tempo e spazio.

---

2. Gestione della memoria

2.1 Premessa

Un reale database, di quelli che possiamo trovare in sistemi enterprise e di produzione, possono immagazzinare una grande quantità di dati. In fase di progettazione abbiamo deciso di emulare, per quanto possibile, le capacità di questi sistemi, cercando il miglior compromesso tra complessità di implementazione e prestazioni con dataset corposi.

2.2 Assunzioni

1. I file di testo devono mantenere la struttura indicata nelle specifiche del progetto.
2. Non si possono creare altri file su disco, per salvare dati in forma più comoda all'uso con certe strutture dati.
3. La gestione della memoria deve essere efficiente come il resto del sistema. Il database mantiene i dati su disco secondo le specifiche del progetto, in RAM vengono usate strutture dati viste al corso per avere ottime prestazioni.

2.3 Scopo

Limitare l'utilizzo di RAM per mantenere alte le prestazioni di tutto il sistema, senza rallentare le query.

2.4 Definizione delle caratteristiche

1. Possibilità di definire una soglia di memoria, sopra la quale è necessario deallocare per avere altra memoria.
2. Deve permettere al database di caricare almeno una tabella per volta.
3. Migliorare il processo di allocazione e deallocazione tramite un sistema di caching basilare.

2.5 Descrizione

2.5.1 Perchè un limite di memoria

Supponendo un utilizzo con dataset grandi, un sistema di caching ben realizzato può fornire speed-up importanti, limitando i caricamenti da disco. Se non gestito, il sistema di caching arriverebbe ad avere tutti i dati in RAM, senza caricare nulla da disco e potenzialmente rallentando l'intera macchina. Considerando un caching completo una strada non sempre percorribile, il sistema di limitazione della memoria serve a prevenire eccessivi rallentamenti, eliminando le tabelle dalla RAM secondo un criterio temporale. In caso di pessime prestazioni con cambi repentini di tabelle, si suggerisce di aumentare il limite di memoria.

2.5.2 Modalità di implementazione

Il sistema usa un heap con update dinamico per approssimare una coda di priorità con update dinamico. La chiave dell'heap è un intero che indica l'indice dell'ultima query su quella tabella. La tabella è memorizzata come valore. La memoria utilizzata da ogni tabella viene sommata in un contatore generale, usato per i confronti con il limite impostato.

2.5.3 Operazioni nell'heap a seguito delle query

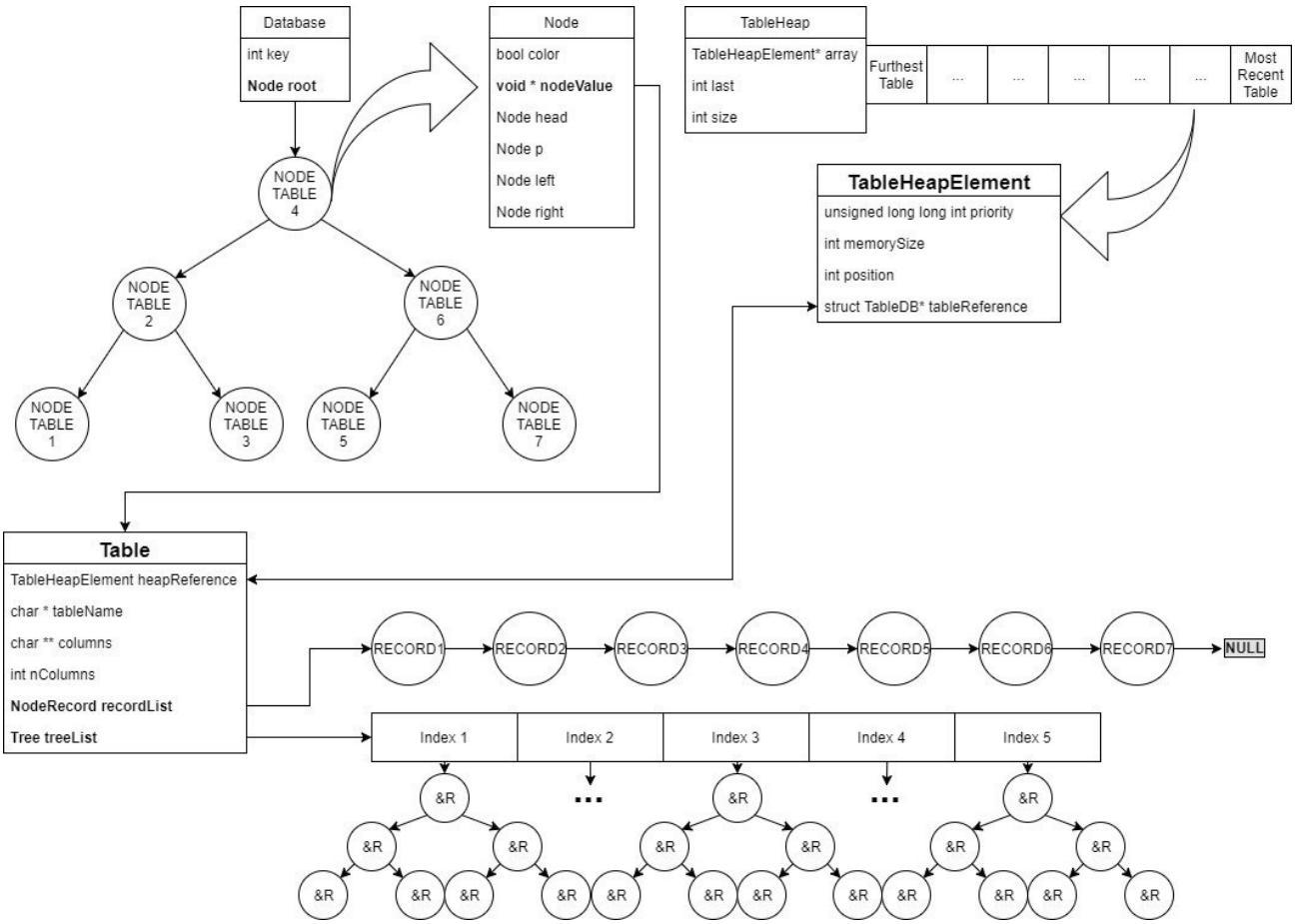
Ad ogni query, la chiave della tabella interessata viene aggiornata, spostando la tabella in una foglia dell'heap. Quando la memoria raggiunge il limite impostato, il sistema procede a deallocare tabelle dalla radice dell'heap, fino a quando non c'è abbastanza spazio a disposizione, o rimane una sola tabella.

2.5.4 Update Dinamico con complessità logaritmica

Per poter effettuare l'update dinamico sull'heap con complessità logaritmica, è necessario un riferimento reciproco tra l'elemento nell'heap e la tabella corrispondente: nel nodo dell'heap è presente un puntatore alla tabella e vice versa. L'heap è una struttura dati vista a lezione che è stata aumentata per gestire l'update dinamico in `O(log n)`. Per la implementazione si è preso spunto dal paper *A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms* redatto da O. Tamir, A. Morrison e N. Rinetzky [1], osservando in particolare la sezione numero 3 : *"A Sequential Heap with Mutable Priorities"*.

3. Database e Strutture dati

Il database è accessibile da un unico puntatore dichiarato globale e statico. È stata fatta questa scelta in quanto si assume di lavorare sempre su un unico database, seguendo le linee guida del progetto. In questo modo si evita di appesantire lo stack di sistema a ogni chiamata di funzione. Segue uno schema grafico della sua struttura interna e la spiegazione di ogni elemento di cui è composto.



Nel database vengono utilizzate:

1. Linked List
2. Red-Black Tree
3. Heap con Update Dinamico

3.1 Linked List

Le linked list utilizzate sono implementate nel modo classico.

3.2 Red-Black Tree

I Red-Black Tree (d'ora in poi chiamati anche RBT) utilizzati sono stati modificati per l'applicazione specifica. L'implementazione delle funzioni core si è ispirata allo pseudo-codice presente sul libro di testo consigliato dal docente "Introduction to Algorithms" di Cormen, Leiserson, Rivest e Stein.

3.2.1 Struct RBT

La struct utilizzata per rappresentare l'albero contiene:

1. Puntatore alla radice `struct RBTNode * root`
2. Chiave intera `int key` . Il valore della key può essere:
    - Key == -2 : indica che il RBT contiene tabelle
    - Key == -1 : non utilizzato, riservato dalla funzione `searchColumnIndex()`
    - 0 ≤ Key ≤ MAX_INT : indica che il RBT contiene puntatori a record, e il valore di key rappresenta l'indice della colonna nella tabella.

3.2.2 Struct RBTNode

La struct utilizzata per rappresentare i nodi dell'albero contiene:

1. Colore del nodo `bool color`
2. Puntatore alla radice `struct RBTree * head`
3. Puntatore al nodo padre `struct RBTree * p`
4. Puntatore al nodo destro `struct RBTree * r`
5. Puntatore al nodo sinistro `struct RBTree * l`
6. Puntatore al record o alla tabella `void * nodeValue`

3.2.3 Uso

La struct Database è un RBT contenente le tabelle caricate in memoria.

3.3 Heap con Update Dinamico

L'Heap è stato modificato, aggiungendo la possibilità di fare Update Dinamici in tempo logaritmico, ispirandosi al paper redatto da O. Tamir, A. Morrison e N. Rinetzky [1], osservando in particolare la sezione numero 3: "A Sequential Heap with Mutable Priorities".

3.4 Strutture

3.4.1 Struttura TableDB

La struttura tabella (`struct TableDB`) contiene:

1. Il nome della tabella `char * name`
2. I nomi delle colonne `char ** columns`
3. Il numero delle colonne `int nColumns`
4. La lista di record `struct Record * recordList`
5. L'array degli RBT `struct RBTree* treeList`

L'array degli RBT contiene tanti RBT quante sono le colonne della tabella, e ognuno di questi mantiene i record ordinati per quel campo.

3.4.2 Struttura ParseResult

`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.

3.4.3 Struttura QueryResultElement

`struct QueryResultElement` è una struttura che serve a contenere il risultato delle query. È una semplice linked list, in cui il campo “occurrence” viene utilizzato nelle query “group by” per memorizzare le occorrenze del valore e il campo “nodeValue” punta al record a cui si vuol riferire questo nodo della lista.

---

4. Corpo centrale

Il corpo centrale è diviso in tre parti principali:

1. Parse della query
2. Recupero della tabella
3. Esecuzione della query

A cui si aggiungono tre fasi minori:

- Inizializzazione del database
- Controllo query *(tra parsing ed esecuzione della query)*
- Deallocazione della memoria *(dopo aver eseguito la query)*

4.1 Inizializzazione

Viene inizializzata la struttura che conterrà 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 Parsing

Il parsing viene gestito dalla funzione `parseQuery()` , che ritorna una `struct ParseResult` con tutte le informazioni contenute nella query. In caso quest'ultima fosse malformata, l'esecuzione viene interrotta e la proprietà `success` del `ParseResult` avrà valore `false` .

4.3 Recupero della tabella

La tabella richiesta dalla query viene cercata in RAM dalla funzione `searchTableDb()` . Se la ricerca ha esito positivo, viene aggiornata la chiave associata alla tabella nella coda di priorità per la gestione della memoria. In caso di esito negativo, la tabella viene caricata da disco tramite `loadTableFromFile()` . Alla fine della fase di recupero della tabella, si ottiene una `struct Table` contenente l'esito dell'operazione e, in caso sia positivo, la tabella.

4.4 Esecuzione Query

Viene eseguita la query.

4.4.1 Controllo della query

Ora che si hanno tutte le informazioni, si procede a un check della validità delle informazioni richieste dalla query.

> E.g.: Le colonne richieste esistono.

4.4.2 Create Table

Una query `CREATE TABLE` valida prevede che non venga caricata nessuna tabella. Viene generato il file su disco e caricato in memoria tramite la stessa procedura applicata alle altre tabelle.

4.4.3 Insert Into

Il record specificato viene aggiunto prima nel file su disco, poi nelle strutture in memoria.

4.4.4 Selezione

Attraverso la funzione `querySelect()` viene creata una `struct QueryResultList`, che contiene il risultato della query. Il contenuto viene loggato tramite una chiamata a `generateLog()` e la struttura deallocata.

4.5 Deallocazione della memoria

Al termine dell'esecuzione vengono eliminate le strutture superflue, in questo caso la `struct ParseResult`.

5. Parser

Il parser deve gestire un numero limitato di possibili query, ha quindi una struttura molto semplice. Tutta l'operazione di parsing viene eseguita in tempo lineare rispetto alla lunghezza della query.

L'entry point del parser è la funzione:

```
ParseResult parseQuery (char * query);
```

Durante l'esecuzione viene usato direttamente un puntatore `char *`, che segue il progresso del parsing. Il parser restituisce un puntatore alla `struct ParseResult`, così composta:

```
struct ParseResult {
    bool success;
    char * tableName;
    int queryType;
    int querySelector;
    int nColumns;
    char ** columns;
    char ** fieldValues;
    char * keyName;
    char * key;
    int order;
    int parseErrorCode;
};
```

dove:

1. `bool success` : contiene l'esito dell'analisi.
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:
    - Create Table: `-1`
    - Select con filtro Where: `0`
    - Select con filtro Order By: `1`
    - Select con filtro Group By: `2`
    - Insert Into: `3`
    - Select senza filtri: `4`
    - 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, xhe corrisponde al numero di puntatori presenti nel 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:
    - Equal: `0`
    - Greater: `1`
    - Lesser: `2`
    - Greater Equal: `3`
    - Lesser Equal: `4`
    - No Operator: `5`

8. `char * keyName` : usato nelle query di tipo:
    - Select con filtro Where, per contenere il nome del campo interessato dalla condizione where.
    - Select con filtro Group By, per contenere il nome del campo per cui raggruppare i record.
    - 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 controlla la condizione.
10. `int order` : usato solo nelle query di tipo Select con filtro Order By, contiene l'ordine desiderato. Può assumere i valori:
    - `ASC` : 0
    - `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 può facilmente risalire al punto in cui si è manifestato il problema. I codici di errore sono suddivisi in classi:
    - `0-99` : Errori Generali
    - `101-199` : Errori durante il parsing di una query "Create Table"
    - `201-299` : Errori durante il parsing di una query "Insert Into"
    - `301-399` : Errori durante il parsing della parte senza filtri di una query "Select"
    - `401-499` : Errori durante il parsing della parte finale di una query "Select Where"
    - `501-599` : Errori durante il parsing della parte finale di una query "Select Group By"
    - `601-699` : Errori durante il parsing della parte finale di una query "Select Order By"

6 Riferimenti esterni:

[1] *A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms*, by O. Tamir, A. Morrison e N. Rinetzky.