Università degli Studi di Roma Tre

Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche

Progetto 1 per "Tecnologie e Architetture per la gestione dei dati"

Analisi del Comportamento delle Interrogazioni su un Database Universitario

Autore
Luca Borrelli

Matricola
559443

Sommario

Il presente documento descrive il progetto volto alla creazione e analisi di un database universitario.
Il progetto ha come obiettivo la sperimentazione del comportamento delle interrogazioni (query) sul database in condizioni differenti: **senza indice** e, in seguito, con l'applicazione di un indice `BTREE` sul campo `corso.id`.

Indice

Elenco delle figure

Elenco delle tabelle

0.1 Struttura del Database

Il database **universita** è composto da tre tabelle principali:

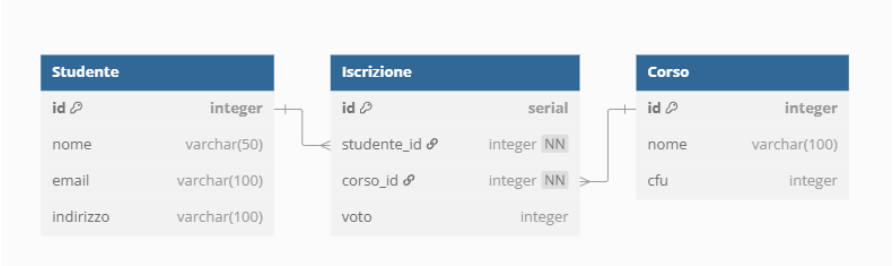

Figura 1: Diagramma ER del database universitario.

- **Studente**: Archivia i dati anagrafici degli studenti, comprendendo un identificativo univoco, il nome, l'indirizzo email e l'indirizzo fisico.
- **Corso**: Contiene le informazioni relative ai corsi offerti, tra cui un identificativo univoco, il nome descrittivo del corso e il numero di CFU (Crediti Formativi Universitari) assegnati.
- **Iscrizione**: Funziona da tabella ponte per rappresentare la relazione molti-a-molti tra studenti e corsi; oltre a registrare le associazioni, memorizza il voto ottenuto dallo studente nel corso.

La relazione tra le tabelle è definita in modo da permettere l'esecuzione di join per recuperare informazioni complete su studenti e corsi.

0.2 Tecnologie e Approccio

Il progetto è stato sviluppato utilizzando:

- **Python** come linguaggio di programmazione.
- Il modulo `psycopg2` per la connessione e l'interazione con PostgreSQL.
- **Faker** per la generazione di dati sintetici, utili a popolare il database con un numero significativo di record (50.000 studenti, 5.000 corsi, 2.000.000 iscrizioni).
- Script SQL per la creazione delle tabelle e, successivamente, per la creazione degli indici.

L'approccio adottato prevede la popolazione del database, l'esecuzione di query con `EXPLAIN ANALYZE` per analizzare il piano di esecuzione e il monitoraggio delle statistiche I/O (blocs hit e physical read) pre e post l'aggiornamento delle statistiche tramite il comando `ANALYZE`.

0.3 Analisi delle Performance delle Interrogazioni

In questa sezione vengono analizzati i risultati ottenuti dall'esecuzione di due query specifiche, confrontando il comportamento del sistema nelle seguenti configurazioni:

1. **Configurazione NO_INDEX**: Nessun indice sul campo `corso_id`.
2. **Configurazione CON_INDEX**: Presenza di un indice `BTREE` sul campo `corso_id` (attraverso l'indice `idx_iscrizione_corso_btree`).

0.3.1 Descrizione delle Query

1. **Query 1: Join e Selezione**

   ```
   EXPLAIN ANALYZE SELECT s.nome, s.email, i.voto FROM Studente
   s JOIN Iscrizione i ON s.id = i.studente_id WHERE i.corso_id =
   10;
   ```

   La query esegue una join tra `Studente` e `Iscrizione` filtrando per un valore specifico di `corso_id`.

2. **Query 2: Range Scan su `corso_id`**

   ```
   EXPLAIN ANALYZE SELECT s.nome, s.email, i.voto FROM Studente
   s JOIN Iscrizione i ON s.id = i.studente_id WHERE i.corso_id BETWEEN
   100 AND 200 ORDER BY i.corso_id;
   ```

   Questa query effettua una join applicando un filtro su un intervallo di valori e ordinando il risultato in base a `corso_id`.

0.3.2 Caso NO_INDEX

Aspettative Teoriche:

- **Assenza di Indice:** Senza un indice, le query si basano su scansioni sequenziali e join tramite hash, con conseguente elevato carico I/O.
- **Effetto dell'`ANALYZE`:** L'aggiornamento delle statistiche tramite `ANALYZE` dovrebbe migliorare marginalmente il piano di esecuzione, riducendo tempi e I/O.

Risultati Ottenuti:

PRE-ANALYZE (NO_INDEX)

- **Query 1:**
  - Piano: `Gather` → `Hash Join` con scansioni parallele (Parallel Seq Scan su entrambe le tabelle).
  - Statistiche I/O: Differenza `Num Pin`: 6911, `Num Read`: 0.
  - Tempo di Esecuzione: ≈ 97.8 ms.
- **Query 2:**
  - Piano: `Gather Merge` con `Sort` su `corso_id` e `Hash Join`.
  - Statistiche I/O: Differenza `Num Pin`: 6929, `Num Read`: 0.
  - Tempo di Esecuzione: ≈ 110.0 ms.

POST-ANALYZE (NO_INDEX)

- **Query 1:**
  - Piano: Simile a quello pre-analyze (Gather + Hash Join).
  - Statistiche I/O: Differenza `Num Pin`: 5968, `Num Read`: 0.
  - Tempo di Esecuzione: ≈ 85.8 ms.
- **Query 2:**
  - Piano: Rimane invariato (Gather Merge, Sort, Hash Join).
  - Statistiche I/O: Differenza `Num Pin`: 6974, `Num Read`: 0.
  - Tempo di Esecuzione: ≈ 104.0 ms.

0.3.3 Caso CON_INDEX (Presenza di un Indice BTREE su `corso_id`)

Aspettative Teoriche:

- **Utilizzo dell'Indice:** L'indice BTREE dovrebbe permettere l'utilizzo di scansioni indicizzate (Bitmap Index Scan) per filtrare i record su `corso_id`, riducendo il numero di blocchi letti e accelerando l'esecuzione della query.
- **Miglioramento delle Performance:** In particolare, per la Query 1, ci si aspetta una riduzione significativa dei tempi di esecuzione, mentre per la Query 2 il beneficio dovrebbe essere evidente nella fase di recupero dei dati per il range scan.

Risultati Ottenuti:

PRE-ANALYZE (CON_INDEX)

- **Query 1:**
  - **Piano di Esecuzione:**
    - `Hash Join` utilizzato in combinazione con un `Bitmap Heap Scan` su `iscrizione`, che sfrutta il `Bitmap Index Scan` sull'indice `idx_iscrizione_corso_btree`.
  - **Statistiche I/O:** Differenze `Num Pin` e `Num Read` pari a 0.
  - **Tempo di Esecuzione:** $\approx$ 18.2 ms.
- **Query 2:**
  - **Piano di Esecuzione:**
    - `Sort` seguito da `Hash Join` con un `Bitmap Heap Scan` che utilizza il `Bitmap Index Scan` sull'intervallo di `corso_id`.
  - **Statistiche I/O:** Differenze `Num Pin` e `Num Read` pari a 0.
  - **Tempo di Esecuzione:** $\approx$ 99.5 ms.

POST-ANALYZE (CON_INDEX)

- **Query 1:**
  - **Piano di Esecuzione:** Il medesimo piano (Hash Join con Bitmap Heap Scan e Bitmap Index Scan) ma con tempi di esecuzione ridotti.
  - **Statistiche I/O:** Differenze in `Num Pin` e `Num Read` pari a 0.
  - **Tempo di Esecuzione:** $\approx$ 11.1 ms.
- **Query 2:**
  - **Piano di Esecuzione:** Il piano prevede un `Sort` seguito da un `Hash Join` che impiega il `Bitmap Index Scan` per il range di `corso_id`.
  - **Statistiche I/O:** Differenze in `Num Pin` e `Num Read` pari a 0.
  - **Tempo di Esecuzione:** $\approx$ 54.1 ms.

Commento Comparativo:

- **Effetto dell'Indice:** L'introduzione dell'indice `BTREE` su `corso_id` ha consentito l'uso di `Bitmap Index Scan` e, conseguentemente, un accesso più rapido ai dati rispetto alla scansione sequenziale.
- **Miglioramenti Osservati:**
  - Per la **Query 1**, il tempo di esecuzione è sceso da circa 18.2 ms in PRE-ANALYZE a 11.1 ms in POST-ANALYZE.
  - Per la **Query 2**, il tempo di esecuzione è migliorato sensibilmente, passando da circa 99.5 ms a 54.1 ms.
- **Statistiche I/O:** In entrambi i casi, le differenze nelle statistiche I/O risultano pari a 0, evidenziando che l'uso dell'indice ha eliminato l'accesso a blocchi aggiuntivi, ottimizzando così l'interrogazione.

0.4 Conclusioni

Il progetto ha evidenziato come l'assenza di un indice sul campo `corso_id` induca PostgreSQL a ricorrere a scansioni sequenziali e join basati su hash, con tempi di esecuzione relativamente elevati. L'aggiornamento delle statistiche tramite `ANALYZE` ha apportato miglioramenti modesti.

L'introduzione di un indice `BTREE` sul campo `corso_id` ha determinato un netto miglioramento delle performance:

- L'accesso ai record è avvenuto tramite `Bitmap Index Scan` e `Bitmap Heap Scan`, riducendo i tempi di esecuzione in entrambe le query.
- Il passaggio da tempi di esecuzione mediamente più elevati (fino a 110 ms) nella configurazione NO_INDEX a valori notevolmente inferiori (fino a 54 ms) in presenza dell'indice conferma l'importanza di una corretta indicizzazione per query di selezione e range scan.

Questi risultati sottolineano come l'ottimizzazione delle performance in un database relazionale possa essere significativamente influenzata dalla presenza di indici, soprattutto in scenari con grandi volumi di dati.