Progettazione Fisica

13 / 12 / 2016

Progettazione Fisica

- *Ingresso*:
  - Schema logico della base di dati
  - Caratteristiche del sistema scelto
  - Previsioni sul carico applicativo (queries)

- *Uscita*:
  - Strutture fisiche utilizzate (struttura primaria per ciascuna relazione, eventuali indici secondari)

Progettazione Fisica

- Operazioni più costose:
  - **Selezione** (accesso ad uno o più record sulla base di uno o più attributi)
  - **Join**

Queste operazioni sono molto più efficienti se esistono indici sui campi interessati (*primari* o *secondari*)

Progettazione Fisica

- **Strategie**:
  - La chiave primaria sarà quasi sempre coinvolta in operazioni di selezione o di join
    => spesso utile costruire un indice
    => valutare se utilizzarlo come primario

  - Indici su altri attributi spesso coinvolti in selezioni o join.
    - *B+-tree*: accesso **logaritmico**, utile per intervalli
    - *Hash*: accesso **diretto**, non utile per intervalli

Approccio Sistematico

- Si supponga di avere le operazioni $O_1, O_2, \ldots, O_n$
- Ciascuna con la frequenza $f_1, f_2, \ldots, f_n$
- Per ogni operazione è possibile definire un costo di esecuzione c_i (*numero di accessi a memoria secondaria*)
- Il costo può variare a seconda delle strutture fisiche scelte

La progettazione fisica = minimizzare il costo complessivo:

$$\sum_{i=1}^{n} c_i \, f_i$$

Consideriamo la relazione IMPIEGATO(Matricola, Cognome, Nome, DataNascita) con un numero di tuple pari a N = 10 000 000 abbastanza stabile nel tempo (pur con inserimenti e cancellazioni) e una dimensione di ciascuna tupla pari a L = 100 byte, di cui K = 2 byte per la chiave (Matricola) e C=15 byte per Cognome.

Supponiamo di avere a disposizione un DBMS che permetta strutture fisiche disordinate, ordinate e hash e che preveda la possibilità di definire indici secondari e un sistema operativo che utilizzi blocchi di dimensione B = 2000 byte con puntatori a blocchi di P = 4 byte.

Supponiamo che le operazioni principali siano le seguenti:

- O_1 – ricerca sul numero di matricola con frequenza f_1 = 2000 volte al minuto
- O_2 – ricerca sul cognome (o una sua sottostringa iniziale, abbastanza selettiva, in media una sottostringa identifica S = 10 tuple) con frequenza f_2 = 100 volte al minuto

Effettuare la progettazione fisica per identificare le strutture primarie e secondarie.

Considerazioni:

1. E’ comunque necessaria una struttura ad accesso diretto per matricola e cognome, visto che una scansione sequenziale sarebbe troppo costosa.
2. Non è possibile usare una struttura hash per Cognome, perché si vuole cercare per sottostringa (si può quindi considerare un indice primario)
3. Per la matricola si può utilizzare una struttura hash (ricerca diretta e struttura stabile nel tempo), oppure un indice (primario o secondario) in alternativa.

Abbiamo quindi 2 alternative da valutare:

- Struttura hash su Matricola - indice secondario su Cognome
- Indice primario su Cognome - secondario su Matricola

Ora possiamo calcolare i costi delle operazioni nei due casi, successivamente moltiplicare i costi per le frequenze per trovare l’alternativa migliore.

- $C_{1,A}$ – accesso diretto utilizzando l'hash (costo = 1)
- $C_{2,A}$ – richiede la visita dell'albero di Cognome + accessi diretti ai dati.

  Fanout nodi intermedi dell'albero su cognome
  ~100 = (2000 bytes / (15 byte + 4 byte))

  Profondità albero su cognome
  $\log_{100}$ 10 000 000 = 3.5 (quindi **4 accessi** per raggiungere foglie)

  + (mediamente) S = 10 accessi alla struttura primaria per recuperare le tuple (ogni ricerca accede in media a10 tuple)

  costo totale = 14

- $C_A = C_{1,A} \times f_1 + C_{2,A} \times f_2 =$

  $= 1 \times 2000 + 14 \times 100 =$ **3400**

- $C_{1,B}$ – richiede la visita dell'albero di Matricola (secondario) + accesso

  Fanout nodi intermedi dell'albero su Matricola
  ~330 = (2000 bytes / (2 byte + 4 byte))

  Profondità albero su cognome
  $\log_{330} 10\,000\,000 = 2.78$ (quindi **3 accessi** per raggiungere foglie)

  + 1 accessi alla struttura primaria (chiave primaria)

  costo totale = 4

- $C_{2,B}$ – richiede la visita dell'albero di Cognome (primario)

  Fanout nodi intermedi dell'albero su Cognome ~= 100

  Le foglie contengono 2000 byte / 100 byte = 20
  Quindi ci sono 10 000 000 / 20 = 500 000 foglie

  Profondità albero su cognome (senza foglie)
  $\log_{100}$ 500 000 = 2.85 (quindi **3 accessi + 1** per raggiungere foglie)

  costo totale = 4

- $C_B = C_{1,B} \times f_1 + C_{2,B} \times f_2 =$

  $= 4 \times 2000 + 4 \times 100 =$ **8 400**

Quindi viene scelta l'alternativa A (2800 accessi al minuto < 10 400)

Cosa succederebbe se f1 e f2 fossero invertite (f1 = 100, f2 = 2000) ?

Quindi viene scelta l'alternativa A (2800 accessi al minuto < 10 400)

Cosa succederebbe se f1 e f2 fossero invertite (f1 = 100, f2 = 2000) ?

- $C_A = C_{1,A} \times f_1 + C_{2,A} \times f_2 =$

  $= 1 \times 100 + 14 \times 2000 =$ **28 100**

- $C_B = C_{1,B} \times f_1 + C_{2,B} \times f_2 =$

  $= 4 \times 100 + 4 \times 2000 =$ **8 400**

In questo caso si sceglierebbe l'alternativa B.

Es. 2 A table STUDENT(RegNo, Name, City) has 100K tuples in 7K blocks with an entry-sequenced organization. There are B +-tree indexes on Name and City, both of depth 3, with the ability to reach 5 tuples in average for each value of Name, and 40 tuples in average for each value of City. Consider the following SQL query:

```
select * from Student
where Name in (Name1, Name2, ..., Namen)
  and City in (City1, City2, ..., Cityc)
```

Determine the optimal strategy for query execution based on the number of values (**n**,**c**) of Name and City listed in the where clause query, considering these four cases:

1) (**n**,**c**) = (10, 10)
2) (**n**,**c**) = (1000, 10)
3) (**n**,**c**) = (10, 1000)
4) (**n**,**c**) = (1000, 1000)

One search based on 1 Name value costs : 3 (tree nodes) + 5 (primary blocks) = 8 i/o operations
One search based on 1 City value costs : 3 (tree nodes) + 40 (primary blocks) = 43 i/o op.s

Scenarios / Used Index	Name	City	Seq. Scan
1)	10 x 8 = **80**	10 x 43 = 430	7.000
2)	1.000 x 8 = 8.000	10 x 43 = **430**	7.000
3)	10 x 8 = **80**	1.000 x 43 = 43.000	7.000
4)	1.000 x 8 = 8.000	1.000 x 43 = 43.000	**7.000**

Es. 3 A table USER(Email, Password, LastName, FirstName) contains 128K users and is stored on 25K blocks, with a primary hash organization on the primary key.

A table REVIEW(Email, ISBN, Date, Rating, ReviewText) has instead 4M tuples and is organized with a primary B +-tree (ie, the tuples are entirely contained in the leaves of the tree) with the email as a key; the average fan-out is equal to 35 and leaf nodes occupy 1M blocks. Estimate the cost of implementing the join between the two tables using the most efficient technique.

Strategy 1

A strategy with Reviews scanned in email order, with a lookup for each such email:

4 + 1M = 1M i/o to scan the external table

128K i/o to lookup all the users (only once per user, as the reviews are in email order)

Overall: 1M + 128K = 1.13 M i/o

Strategy 1

A strategy with Reviews scanned in email order, with a lookup for each such email:

4 + 1M = 1M i/o to scan the external table

128K i/o to lookup all the users (only once per user, as the reviews are in email order)

Overall: 1M + 128K = 1.13 M i/o

Strategy 2

Scanning the Users and looking up the reviews by means of the B+ would cost more:

25 K to scan the users

Cost of 1 lookup : 4 intermediate nodes ($\log_{35}$ 1M ~= 4)
+ 9 leaf nodes to collect the 32 (4M / 128K) reviews by each user
(4M / 1M = 4 reviews per block, 32/4 = 8, we read one additional block because ranges will not start at the beginning of the block)

Overall: 25K + 128K x (4 + 9) = 1.7 M i/o

A tale of two Joins (*31 Jan 2012*)

STUDENT (Matricola, FirstName, LastName, BirthDate, …)

contains 150K students on 10K blocks in a primary hash over attribute Matricola.

EXAM (Matricola, CourseId, Date, Grade)

contains 2M tuples on 8K blocks in a entry-sequenced structure
we also have a secondary B+ index over Matricola, with fanout 200.

Estimate and compare the cost (in terms of number of disk accesses) of executing the query

```
select *
from Student S join Exam E on S.Matricola = E.Matricola
```

a) with a **hash join** in which Exam is re-structured in a suitable primary hash storage
b) with a **nested loop join** with Student as **external** table.
c) with a **nested loop join** with Student as **internal** table.

(Ignore collisions and caching)

STUDENT has 15 tuples per block, each hash-based access costs 1 i/o.

EXAM has 256 tuples per block, and has no hash-based access.

We need to build a hash structure (in order to ba able to use it!). This requires scanning the whole table, extracting each tuple and inserting it into the right «bucket». The number of buckets is of course the same, as the hash funcion is the same (e.g., *Matricola mod 10.000*): these buckets however will be rather «empty» w.r.t. the blocks in the previous representation (the storage grows overall from 8K to 10K blocks).

We therefore need to perform **8K i/o ops** in order to ***read*** and as much as **4M i/o ops to *update***, as the right bucket must be found for each exam, and the corresponding block needs to be read (moved to main mamory), updated with the new exam, and then re-written to disk.

The **hash-join** per se then costs **2 * 10K = 20K i/o**, to an overall cost of **4M + 28K i/o**

(*a remarkable cost, that should be payed off by the efficient future executions of the same join)*

Calcoliamo ora il costo del nested loop con ESAME come tabella **interna** (cioè nel caso in cui leggiamo un blocco della tabella STUDENTE (iterazione esterna) e per ogni studente cerchiamo in ESAME (iterazione interna) tutti gli esami sostenuti da quel particolare studente. Leggiamo un nuovo blocco di STUDENTE solo dopo aver esaurito tutti gli studenti nel blocco corrente).

L'albero ha profondità media pari a circa 3 ed è relativamente sparso (ha spazio per circa 8M valori di chiave). La **ricerca degli esami di un dato studente** costa quindi **3 accessi ai nodi dell'albero**, **più 1** accesso ai blocchi della struttura primaria **entry-sequenced** per recuperare tutti i campi dell'esame (trascuriamo il caso in cui i puntatori agli esami di uno stesso studente siano ripartiti su più di un nodo foglia dell'albero). In totale pagheremo 3 accessi per ogni studente e uno per ogni esame, cioè **3*150K + 2M accessi**.

Il caricamento di tutti i blocchi di STUDENTE nel**l'iterazione esterna costa inoltre 10K** accessi, per cui in totale avremo = **2M + 460K accessi**.

È un costo inferiore a quello di costruirsi la struttura ad-hoc (poco più di metà), il quale però può ancora considerarsi un buon investimento se, conservando la struttura creata, velocizza poi sostanzialmente le esecuzioni dello stesso join (*20K contro 2,5M*).

Calcoliamo infine il costo del nested loop con ESAME come tabella **esterna** (che a prima vista sembra l'opzione più sensata, dato che gli accessi random basati su matricola alla struttura primaria a hash dello STUDENTE costano molto meno di quelli a ESAME basati sull'indice B+ (1 contro 4)).

Possiamo **scandire tutta la tabella ESAME pagando solo 8K** accessi per la sua lettura, sfruttando la struttura entry-sequenced (non avrebbe senso, infatti, scandire in sequenza le foglie dell'indice B+ e accedere ai blocchi primari "uno studente alla volta", dato che avere gli esami in ordine di matricola non ci aiuta), e dobbiamo poi **accedere 2M volte alla struttura hashed**, una volta per ogni esame, a recuperare i dati dello studente (più volte lo stesso studente, ogni volta che ci imbattiamo in un suo esame – ma stiamo trascurando gli effetti della cache), per un totale di **2M+8K accessi**.

Il costo è minore, ma comunque dello stesso ordine di grandezza di quelli precedentemente calcolati. Un ottimizzatore "miope" potrebbe continuare a scegliere questa ultima opzione senza mai "imbarcarsi" nell'impresa di costruire la struttura hash-based, che invece si rivela strategicamente la più conveniente se il join viene ripetuto frequentemente (come è probabile).