ARGOMENTO 11

Algoritmi di JOIN e costi

Il JOIN

- Finora tutte le operazioni analizzata riguardavano una singola tabella (relazione)
- Tuttavia, una delle operazioni più frequenti (e costose!) in SQL è il JOIN tra tabelle
- Per semplicità, nel seguito faremo riferimento al JOIN tra due tabelle R(A,B) e S(C,D) con condizione B = C ovvero:

$$R \underset{R.B=S.C}{\bowtie} S$$

R è chiamata *outer table*
S è chiamata *inner table*

A	B
...	7
...	9
...	6
...	15
...	20

R

C	D
15	...
10	...
7	...
9	...
20	...

S

JOIN

Assunzioni di partenza:

- Il costo del JOIN è inteso come il numero di pagine che l'algoritmo dovrà leggere e/o scrivere per calcolare il risultato del JOIN
- Via via che le tuple vengono prodotte dall'algoritmo di JOIN, esse vengono mostrate a video all'utente e non riscritte su disco (quindi non avremo costi legati alla scrittura del risultato su disco)

Nested loop JOIN

- E' l'algoritmo più semplice e immediato per eseguire un JOIN

- Si basa sull'uso di due cicli **for**:
  1. **foreach** tuple *t* in R **do:**
  2. (a) **foreach** tuple *t'* in S **do**:

  (b) **if** *t.B* = *t'.C* **then** output the tuple *t,t'*

- Questo algoritmo dovrà leggere per R volte 1+|S| tuple, ovvero |R| + |R|*|S| tuple

Nested loop JOIN

- Come sappiamo, il DBMS non legge singole tuple, ma intere pagine che contengono un certo numero di tuple.
- Quindi la versione più realistica del NLJ è:
  1. **foreach page** *P* di R **do:**
  2. (a) **foreach** pagina *P' di* S **do**:

  (b) per ogni coppia di tuple $t \in P$, $t' \in P'$ *tali che* $t.B = t'.C$ stampa a video la tupla *t,t'*
- Sostanzialmente, il NLJ è ripetuto per ogni coppia di pagine (una di R e una di S) caricate in memoria dall'algoritmo

Costo del Nested Loops JOIN

- In analogia a quanto fatto per il costo delle altre operazioni su singola tabella, il costo del NLJ è:

$$\text{Costo}_{\text{NestedLoopJoin}} = P_R + P_R \cdot P_S$$

- Da notare che l'ordine delle tabelle conta conviene sempre usare la tabella con cardinalità minore come *outer table*!

- Il costo del NLJ (versione *naif*) è molto elevato. Se P_R e P_S hanno valore 1.000, il costo del NLJ sarà di 1.001.000 di operazioni di I/O!!

(Sort-)Merge JOIN

- Il secondo algoritmo assume che le R e S siano ordinate rispetto ai valori degli attributi su cui intendiamo fare il JOIN (B e C rispettivamente):

A	B
...	6
...	7
...	9
...	15
...	20

R

C	D
7	...
9	...
10	...
15	...
20	...

S

Merge JOIN

A	B
...	6
...	7
...	9
...	15
...	20

R

C	D
7	...
9	...
10	...
15	...
20	...

S

- Possiamo quindi immaginare un algoritmo MOLTO più efficiente per calcolare il JOIN:
  1. Prendiamo il primo valore di B in R
  2. Iniziamo a scorrere i valori di C in S:
     a) se troviamo lo stesso valore (anche più volte), stampiamo a monitor la nuova tupla (tuple)
     b) Se troviamo un valore più alto, ripetiamo il processo invertendo il ruolo di R e S

Merge JOIN

A	B
...	6
...	7
...	9
...	15
...	20

R

C	D
7	...
9	...
10	...
15	...
20	...

S

Esempio:

- Partiamo dal valore 6 di B
- Iniziamo a scorrere i valori di C in S e troviamo 7 (caso b)
- Scartiamo il 6 e ripartiamo dal valore 7 in S
- Iniziamo a scorrere i valori di B maggiori di 6 in R
- Troviamo 7: *match*! Restituiamo la tupla …, 7, 7, …
- Scegliamo il valore successivo a 7 in R (cioè 9)
- Ripartiamo dall'inizio e procediamo fino a che non ci sono più valori rimasti

Merge JOIN

- Quante tuple abbiamo dovuto considerare? Solo |R|+|S|, dato che non abbiamo mai dovuto leggere due volte la stessa tupla
- In termini di pagine, il merge JOIN richiede di leggere $P_R + P_S$ pagine
  - Se $P_R = P_S = 1.000$, il numero di pagine da leggere sarà 2.000 e non 1.001.000!
- Tuttavia, il prezzo da pagare è quello di ordinare le tuple delle R e S … e questa operazione è ovviamente piuttosto costosa

Algoritmi di *external merge sort*

- Si tratta di algoritmi che si applicano quando non è possibile caricare in memoria interna (RAM) tutti i dati da riordinare e quindi ci si deve appoggiare su una memoria esterna (es. disco)
- Il costo è calcolato come segue:
  - Sia P_R il numero di pagine necessarie per memorizzare la tabella R
  - Sia B il numero di pagine che possono essere caricate in RAM (in genere nell'ordine delle centinaia)
  - Il costo di riordinare R è $2 \cdot P_R \cdot \left(\left\lceil \log_{B-1} \left\lceil \frac{P_R}{B} \right\rceil \right\rceil + 1 \right)$

Costo del merge JOIN

- Se R e S non sono già ordinate, il costo dell'algoritmo di merge JOIN sarà:

$$\text{Costo}_{\text{SortMergeJoin}} = \text{Costo per ordinare R} + \text{Costo per ordinare S} + P_R + P_S$$

- I costi sono in generale molto minori rispetto al Nested Loops JOIN, ma va tenuto conto del fatto che in alcuni casi ci sono ottimizzazioni che possono rendere quest'ultimo vantaggioso
  - Esempio: se S è sufficientemente piccola da poter essere caricata tutta in RAM, il DBMS la terrà in memoria e il costo del NLJ diventa $P_R + P_S$

Hash JOIN

- Utilizza *hash tables* per velocizzare il JOIN di due tabelle
- L'idea è di usare una funzione di *hashing* sui valori degli attributi di R e S usati nel JOIN per dividere le tuple delle due tabelle in bucket
  - se *t* e *t'* possono essere messe in JOIN, allora *t.B* = *t.C* e quindi valori di *t.B* e *t.C* devono avere lo stesso *hash* (ovvio, dato che sono uguali)
  - NB: da questo **non** segue invece che se due valori hanno lo stesso hash, allora essi sono uguali!

Hash JOIN

- Partiamo sempre dalle tabella R e S:

A	B
...	7
...	9
...	6
...	15
...	20

R

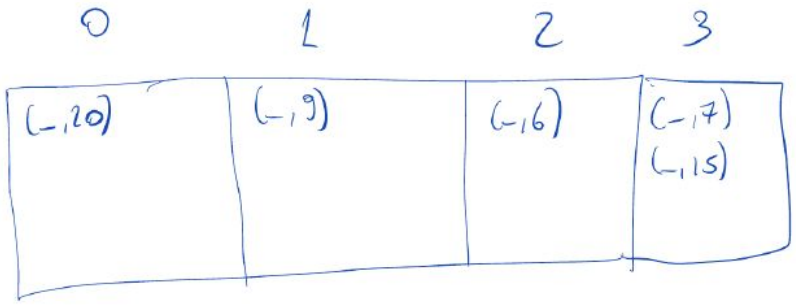
C	D
15	...
10	...
7	...
9	...
20	...

S

- Prendiamo ***h(x) = x%N*** come funzione di hash (dove N è il numero di bucket che il DBMS utilizza per indicizzare le tuple)

Hash JOIN

- Il DBMS crea una copia di R e S dove le tuple sono riorganizzate in base alla funzione di hash:



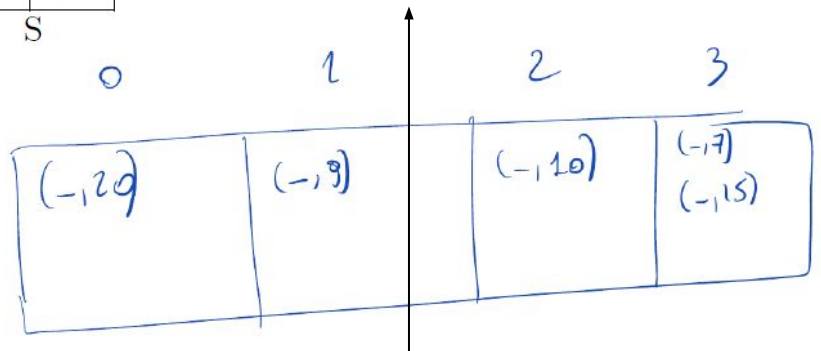
$h(x) = x\%4$

A	B
...	7
...	9
...	6
...	15
...	20

R

C	D
15	...
10	...
7	...
9	...
20	...

S



Hash JOIN

- L'algoritmo procede come segue:
  - Legge e carica in memoria il primo bucket di R e il primo bucket di S
  - Calcola (in memoria) i JOIN delle tuple nei due bucket
  - Passa al successivo bucket di R e di S
- NB: le tuple che sono in un bucket di R possono andare in JOIN solo con tuple che stanno nel bucket corrispondente di S (perché tutte le tuple con lo stesso valore per un attributo sono nello stesso bucket)

Costo dell'hash JOIN

- Il costo totale dell'algoritmo di hash JOIN sarà la somma dei costi di creare le copie delle due tabelle e del costo di fare il JOIN:

$$\text{Costo}_{\text{HashJoin}} = (P_R + 2 \cdot |R|) + (P_S + 2 \cdot |S|) + (P_R + P_S)$$

- Ovviamente, se le tuple sono già distribuite in bucket secondo una funzione di hash, il costo sarà semplicemente $P_R + P_S$
- In base ai valori di P_R , P_S , |R| e |S|, è compito del DBMS scegliere quale metodo conviene utilizzare per ogni caso particolare

Index Nested Loops JOIN

- E' una variante del NLJ in cui si assume che sia presente un indice sulla *inner table*

- L'algoritmo infatti legge tutte le pagine della *outer table* (diciamo R) e poi cerca tramite indice le tuple di S che soddisfano la condizione del JOIN:

1. **foreach** pagina p di R **do**
2. (a) Per ogni tupla $t \in p$, trova tutte le tuple t' in S con $t'.C = t.B$ (tramite l'indice), e stampa t, t';

Index Nested Loops JOIN

- Quindi il costo complessivo è:

$$\text{Costo}_{\text{IndexNestedLoopJoin}} = P_R + |R| \cdot \text{Costo}_{\text{EqSearchS}}$$

dove il costo dell'equality search dipende da che tipo di indice abbiamo sull'attributo di S usato per il JOIN

- NB: il vantaggio si ottiene solo se l'indice è sull'attributo della *inner table*, infatti l'indice non è di alcuna utilità per lo *scan* della *outer table*.

Aggiustamento degli indici

- Obiettivi dell'attività di aggiustamento:
  - Valutare in modo dinamico i requisiti iniziali
  - Riorganizzare gli indici per ottenere la miglior performance possibile
- Motivi per rivisitare le scelte iniziali degli indici:
  - Alcune query possono richiedere troppo tempo per la mancanza di un indice
  - Alcuni indici possono risultare poco utilizzati
  - Alcuni indici possono essere soggetti ad aggiornamenti troppo frequenti

Decisioni progettuali a livello fisico del DB

- Se indicizzare o meno un attributo
  - Es.: l'attributo è una chiave o è usato in una o più query
- Quali attributi indicizzare?
  - Singolo attributo o più attributi?
- Se prevedere un *clustered index*
  - Al massimo uno per tabella!
- Se usare un *hash index* o un *B+ Tree index*
  - Gli indici hash non supportano query su intervalli
- Se usare o meno *hashing* dinamico
  - Adatto per dati molto volatili