Basi di Dati
Architettura dei DBMS: l'ottimizzatore fisico

Corso B

Ottimizzatore fisico

- Ottimizzazione delle interrogazioni per selezione
- Ottimizzazione dei join

Ottimizzatore

L'ottimizzatore ha due fasi

- L'**ottimizzatore logico** lavora sull'albero di parsificazione
- L'**ottimizzatore fisico** considera l'albero prodotto dall'ottimizzazione logica e scegliere gli algoritmi da abbinare ai nodi operativi

Esistono più algoritmi per ogni tipo di nodo operativo

Ottimizzazione delle selezioni

Per semplicità trascuriamo la presenza di OR nel predicato di selezione

Consideriamo la seguente selezione

$$\sigma_{\psi \wedge p1 \wedge p2 \wedge \ldots \wedge ph}(T)$$

Esempio:

STUDENTI(MATR*,Nome*,DataNascita*,Genere,Indirizzo)

$$\sigma_{Nome='Piero' \wedge DataNascita>'1990' \wedge Indirizzo='TO'}(T)$$

(*) Attributi con indice

Ottimizzazione delle selezioni

STUDENTI(MATR*,Nome*,DataNascita*,Genere,Indirizzo)

$$\sigma_{\text{Nome='Piero'} \wedge \text{DataNascita>'1990'} \wedge \text{Indirizzo='TO'}}(T)$$

Alcuni predicati sono risolubili con l'indice (Nome='Piero' e DataNascita>'1990'), altri no (Indirizzo='TO')

$$\sigma_{\psi \wedge p1 \wedge p2 \wedge \ldots \wedge ph}(T)$$

ψ: componente non risolubile (non esiste nessun indice che aiuti a risolvere il predicato)

$p_1 ... p_h$: predicati risolubili (su attributi con indice)

Prima strategia

Alcuni DBMS utilizzano tutti gli indici disponibili, ovvero sfruttano gli indici per ottenere gli insiemi di RID

$$\{RID\}_{p1}, \{RID\}_{p2}, \ldots, \{RID\}_{ph}$$

Successivamente calcolano l'intersezione degli insiemi di RID:

$$R_p = \{RID\}_{p1} \cap \{RID\}_{p2} \cap \ldots \cap \{RID\}_{ph}$$

L'insieme R_p di RID viene portato in memoria centrale e filtrato in base al predicato ψ

Costo della prima strategia

Il costo della prima strategia è dato dal costo di accesso agli indici, mentre il costo di accesso al predicato ψ viene ignorato, in quanto applicato su record già presenti in memoria centrale

Seconda strategia

Altri DBMS invece usano una strategia diversa:

- Considerano l'ipotesi di scansione sequenziale che costa $N_{page}(T)$
- Degli indici a disposizione scelgono quello che conviene di più, trascurando gli altri
- Calcolano quindi i costi di accesso C_{p1}, C_{p2},…, C_{ph}
- L'ottimizzatore sceglie la tecnica di accesso con l'indice più selettivo
- Tutte le altre condizioni vengono verificate in memoria centrale

Prestazioni a confronto

- Alcuni benchmark mostrano che i DBMS che usano un solo indice hanno un'efficienza confrontabile con quella dei DBMS che usano tutti gli indici
- Questi ultimi hanno degli overhead maggiori dovuti all'elaborazione delle liste di RID
- In ogni caso, se il costo dell'accesso sequenziale è minore del costo con indici, gli indici non vengono usati da nessuna delle due strategie
- Se un indice non viene mai usato da un DBMS, ci sono solo costi di mantenimento senza benefici

Gli algoritmi di join

L'operatore di join è il più critico in tutti i DBMS, perché comporta confronti tra tuple di due tavole

Esistono diverse classi di algoritmi di join

Consideriamo il caso generico del theta join tra due tavole R ed S:

$$R \bowtie_{\Theta} S$$

Nested loop

La prima tecnica (Nested Loop) la più semplice che si possa immaginare e consiste nella comparazione delle tuple di R con le tuple di S

Nell'architettura delle basi di dati, possiamo immaginare la tecnica di implementazione in questo modo:

per ogni pagina R_i di R

    per ogni pagina S_h di S

        Calcola il join di $R_i \bowtie_\Theta S_h$

Nested loop: costo

C'è un loop esterno sulle pagine di R

Il loop interno è sulle pagine di S che vengono quindi lette tutte più volte (una per ogni pagina di R)

Il join tra R_i ed S_h viene eseguito in memoria centrale, quindi ha un costo irrilevante

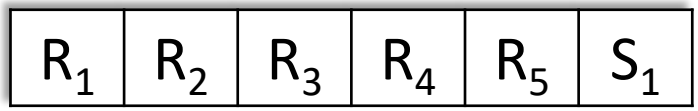
Sul piano dei costi, il DBMS trasferisce in tutto:

$$\text{Costo} = N_{page}(R) + N_{page}(R) \cdot N_{page}(S)$$

Sembra quindi molto costoso, ma si può usare un accorgimento per ridurre in modo sensibile il costo

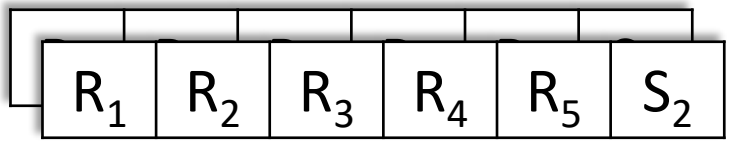
Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S



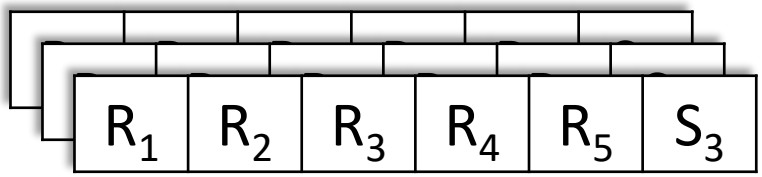
Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S



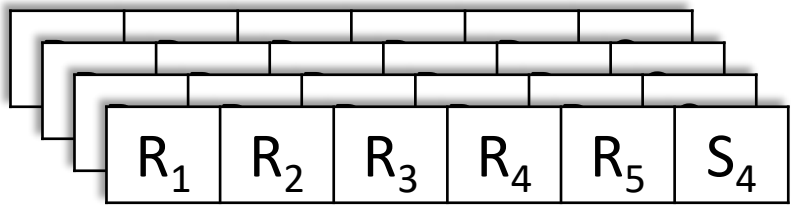
Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S



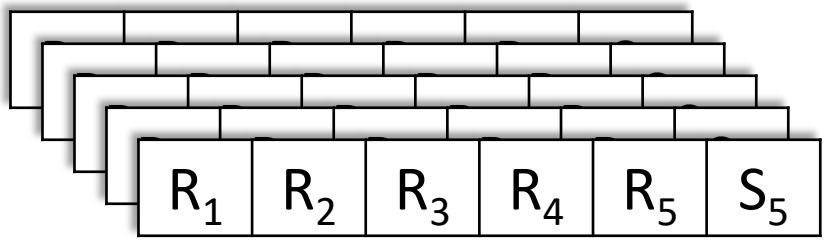
Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S



Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S

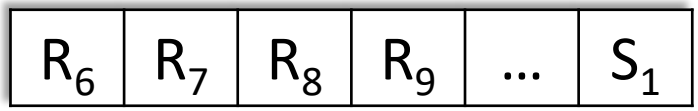


Fino a quando ho esaurito tutta la tavola S

Poi carico le restanti pagine di R

Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S

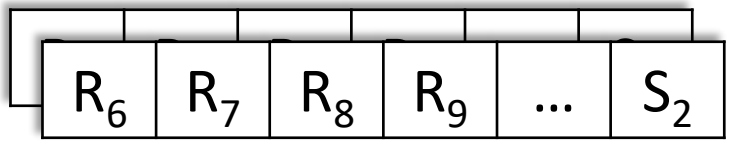


Fino a quando ho esaurito tutta la tavola S

Poi carico le restanti pagine di R

Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S

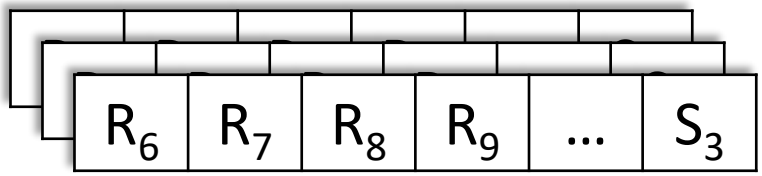


Fino a quando ho esaurito tutta la tavola S

Poi carico le restanti pagine di R

Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S

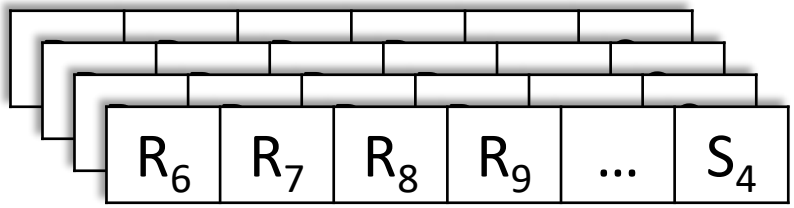


Fino a quando ho esaurito tutta la tavola S

Poi carico le restanti pagine di R

Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S

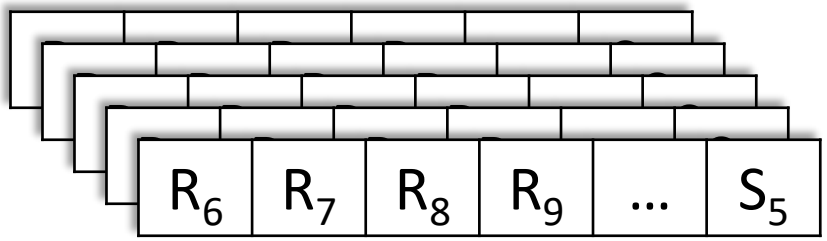


Fino a quando ho esaurito tutta la tavola S

Poi carico le restanti pagine di R

Nested loop: implementazione

Nel buffer abbiamo B pagine: Usiamo B – 1 pagine per caricare le pagine di R e la B-esima pagina per scorrere tutte le pagine di S



Fino a quando ho esaurito tutta la tavola S

Poi carico le restanti pagine di R

Nested block: costo

Questa versione prende il nome di **nested block join**

Anziché caricare una pagina di R e poi eseguire il loop interno, si caricano blocchi di B – 1 pagine di R, e per ogni blocco eseguo un loop completo su S

Il costo di questa implementazione è quindi:

$$\text{Costo} = N_{page}(R) + \lceil N_{page}(R)/\mathbf{(B-1)} \rceil \cdot N_{page}(S)$$

Dato che i buffer reali sono molto capienti (centinaia di pagine) il costo si abbatte notevolmente

Nested block: ottimizzazione

Il costo di questa implementazione è quindi:

$$\text{Costo} = N_{page}(R) + (N_{page}(R)/\mathbf{(B-1)}) \cdot N_{page}(S)$$

Il join è commutativo ($R \bowtie_\Theta S$ è equivalente a $S \bowtie_\Theta R$)

L'ottimizzatore può decidere quale relazione considerare come esterne e quale come annidata

Quale sarà la scelta del DBMS?

Nested block: ottimizzazione

Il costo di questa implementazione è quindi:

$$\text{Costo} = N_{page}(R) + (N_{page}(R)/\mathbf{(B-1)}) \cdot N_{page}(S)$$

Il join è commutativo ($R \bowtie_\Theta S$ è equivalente a $S \bowtie_\Theta R$)

L'ottimizzatore può decidere quale relazione considerare come esterne e quale come annidata

Il DBMS sceglierà la relazione **con meno pagine** come relazione esterna

Nested loop con indice

Consideriamo il caso generico del theta join tra due tavole R ed S:

$$R(\ldots A\ldots) \bowtie_{\Theta(A,B^*)} S(\ldots B\ldots)$$

con l'ipotesi che sull'attributo B esista un indice

Il discorso può essere esteso al caso di condizione su più attributi

Immaginiamo un indice di tipo B+Albero su B

Nested loop con indice

L'algoritmo del nested loop con indice è il seguente:

per ogni pagina R_i di R

    per ogni tupla $t \in R_i$

        calcola $S' = \sigma_{B \, \Theta \, t[A]}(S)$

        se $|S'| > 0$ **allora**

            restituisce $t \times S'$

Esempio

STUDENTI(MATR,Nome,DataNascita,Genere,Indirizzo)

ESAMI(MATR*,Corso,Voto,DataEsame)

$$\text{STUDENTI} \bowtie_{\text{S.MATR=E.MATR}} \text{ESAMI}$$

(*) attributi con indice

1. Si carica una pagina di STUDENTI
2. Si prende una tupla di STUDENTI
3. Si prende il valore della tupla in corrispondenza di A (t[A])
4. Si effettua un'interrogazione su S che (può sfruttare l'indice su MATR): selezione delle tuple di S che soddisfano l'uguaglianza tra B e la costante t[A]
5. Si restituisce la giustapposizione della tupla t con le tuple di S' che soddisfano l'interrogazione del punto 4

Nested loop con indice: costo

Il costo dell'algoritmo è dato dalla scansione sequenziale di R, più |R| volte il costo di accesso per ottenere S'

$$\text{Costo} = N_{page}(R) + CARD(R)\cdot(C_I(S') + C_D(S'))$$

Che vantaggio danno in pratica?

Esempio di calcolo

STUDENTI(MATR,Nome,DataNascita,Genere,Indirizzo)

ESAMI(MATR*,Corso,Voto,DataEsame)

I = INDEX(MATR, ESAMI)

con (dati numerici ragionevoli)

- CARD(STUDENTI) = 50.000 studenti
- N_{page}(STUDENTI) = 1000 pagine
- VAL(MATR,STUDENTI) = CARD(STUDENTI) = 50.000 valori
- CARD(ESAMI) = 500.000 esami (10 esami per ogni studente)
- N_{page}(ESAMI) = 2500 pagine (la tupla esami è più corta)
- N_{foglie}(I) = 800 pagine (circa 60 studenti per foglia) con una media di 10 RID per ogni chiave (10 esami per ogni studente)

Esempio: nested block

Considero B=100

Prendo come relazione esterna STUDENTI (la più piccola), quindi:

$$\text{Costo} = N_{page}(R) + \lceil N_{page}(R)/\mathbf{(B-1)} \rceil \cdot N_{page}(S)$$

$$\text{Costo} = 1000 + \lceil 1000/100 \rceil \cdot 2500 = 26.000 \text{ accessi}$$

Esempio: nested block con indice

Esaminiamo il costo con l'indice:

$$\text{Costo} = N_{page}(R) + CARD(R)\cdot(C_I(S') + C_D(S'))$$

$C_I(S') = \lceil N_{foglie}(I)\cdot(1/VAL(MATR,ESAMI)) \rceil$

$C_I(S') = \lceil 800/50.000 \rceil = 1$

$C_D(S') = \min\{E_{reg}, N_{page}(ESAMI)\}$

$E_{reg} = CARD(ESAMI)\cdot 1/VAL(MATR,ESAMI) = 10$

$C_D(S') = \min\{10,2500\} = 10$

Costo = $1000 + 50.000 \cdot (1 + 10)$ = più di 500.000 accessi

Il costo è **enormemente superiore** al costo del nested block!

Considerazione

A cosa serve allora l'indice?

I sistemi DBMS tradizionali sono tarati per l'attività transazionale, dove le operazioni da compiere sul DB sono mirate a poche tuple (esampio: registrazione di **un** esame di **uno** studente, stampa di **un** certificato per **uno** studente)

I join massivi (**tutti** gli studenti con **tutti** gli esami) si usano solo quando si fanno statistiche, una tantum

Esercizio

Consideriamo la richiesta del certificato esami superati da parte di uno studente

STUDENTI(MATR*,Nome,DataNascita,Genere,Indirizzo)

ESAMI(MATR*,Corso,Voto,DataEsame)

I_1 = INDEX(MATR, STUDENTI), I2 = INDEX(MATR, ESAMI)

$$\sigma_{MATR='X'}(STUDENTI) \bowtie_{S.MATR=E.MATR} ESAMI$$

già ottimizzata dall'ottimizzatore logico

Questa è una tipica interrogazione transazionale rispetto alla quale gli ottimizzatori danno le migliori prestazioni

Esercizio

Quando costa la selezione? (il risultato è una sola tupla)

$$C_\sigma = C_{I1} + C_D(\text{MATR},\text{STUDENTI})$$

$C_{I1} = \lceil 1/\text{CARD}(\text{STUDENTI}) \cdot N_{foglie}(I_1) \rceil$

Immaginiamo un valore $N_{foglie}(I_1)=180$

(foglie di 4KB, data entry di 10Byte: 400 chiavi, ma ipotizziamo un carico parziale del 70%)

$$C_{I1} = \lceil 1/50.000 \cdot 180 \rceil = 1$$

Esercizio

Quando costa la selezione? (il risultato è una sola tupla)

$$C_\sigma = C_{I1} + C_D(\text{MATR},\text{STUDENTI})$$

$C_D = \min\{E_{reg}, N_{page}(\text{STUDENTI})\}$

$E_{reg} = \text{CARD}(\text{STUDENTI}) \cdot 1/\text{CARD}(\text{STUDENTI}) = 1$

$$C_D = \min\{1,1000\} = 1$$

Il costo complessivo $C_\sigma = 2$ accessi

Esercizio

Quando costa il join?

$$C_{\bowtie} = N_{page}(\sigma) + CARD(\sigma)\cdot(C_{I2} + C_D(MATR,ESAMI))$$

Possiamo immaginare che l'interrogazione abbia memorizzato la tupla risultato della selezione in una pagina temporanea, quindi $N_{page}(\sigma) = 1$

$CARD(\sigma) = 1$

$C_{I2} = 1$ (già calcolato in precedenza)

$C_D(MATR,ESAMI) = 10$ (già calcolato in precedenza)

$$C_{\bowtie} = 1 + 1(1 + 10) = 12 \text{ (meno di mezzo secondo)}$$

Esercizio

Il costo dell'esecuzione del join senza gli indici (nested block) è invece:

$$C_{\bowtie} = N_{page}(\sigma) + \lceil N_{page}(\sigma)/\mathbf{(B-1)} \rceil \cdot N_{page}(ESAMI)$$

$$C_{\bowtie} = 1 + \lceil 1/100 \rceil \cdot 2500 = 2501$$

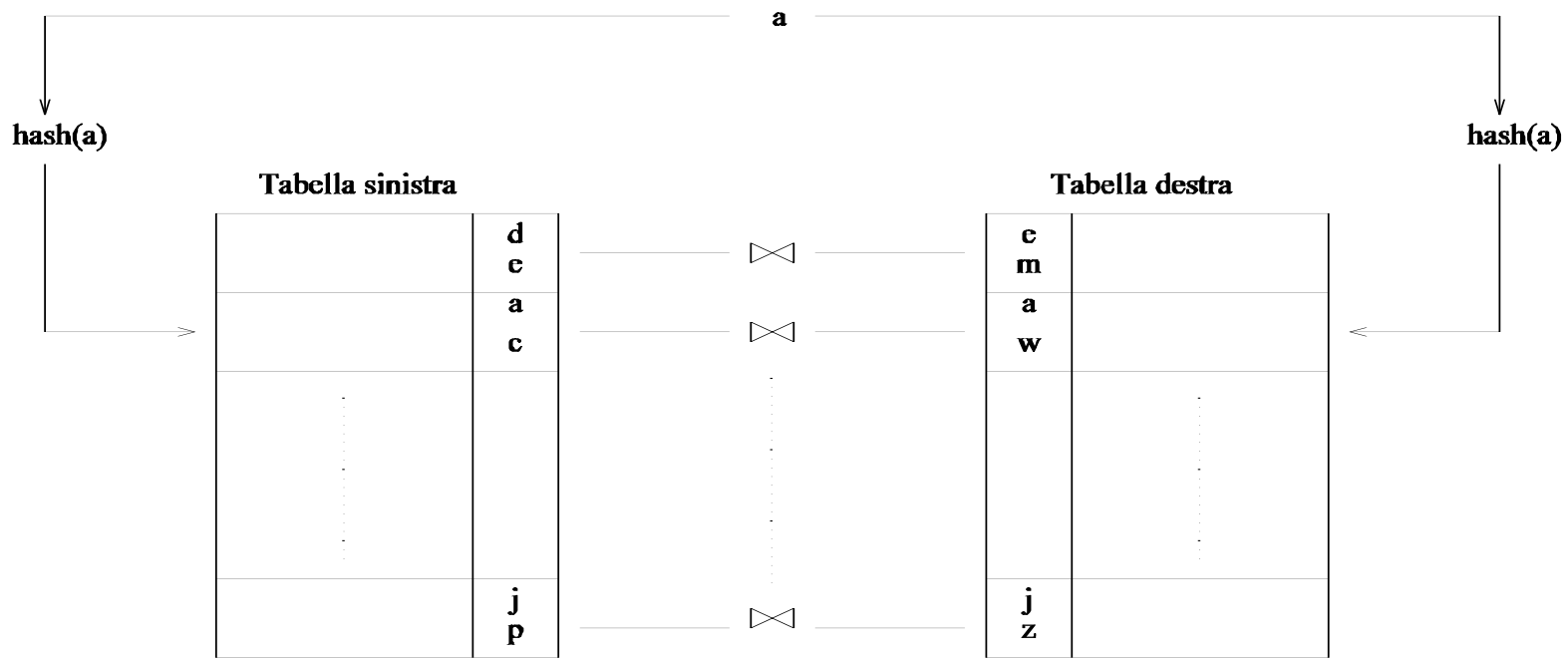
(200 volte meno efficiente)

Altri algoritmi di join

- **Hash-join**: algoritmi basati sulle funzioni di hash e utilizzati solo negli equi-join

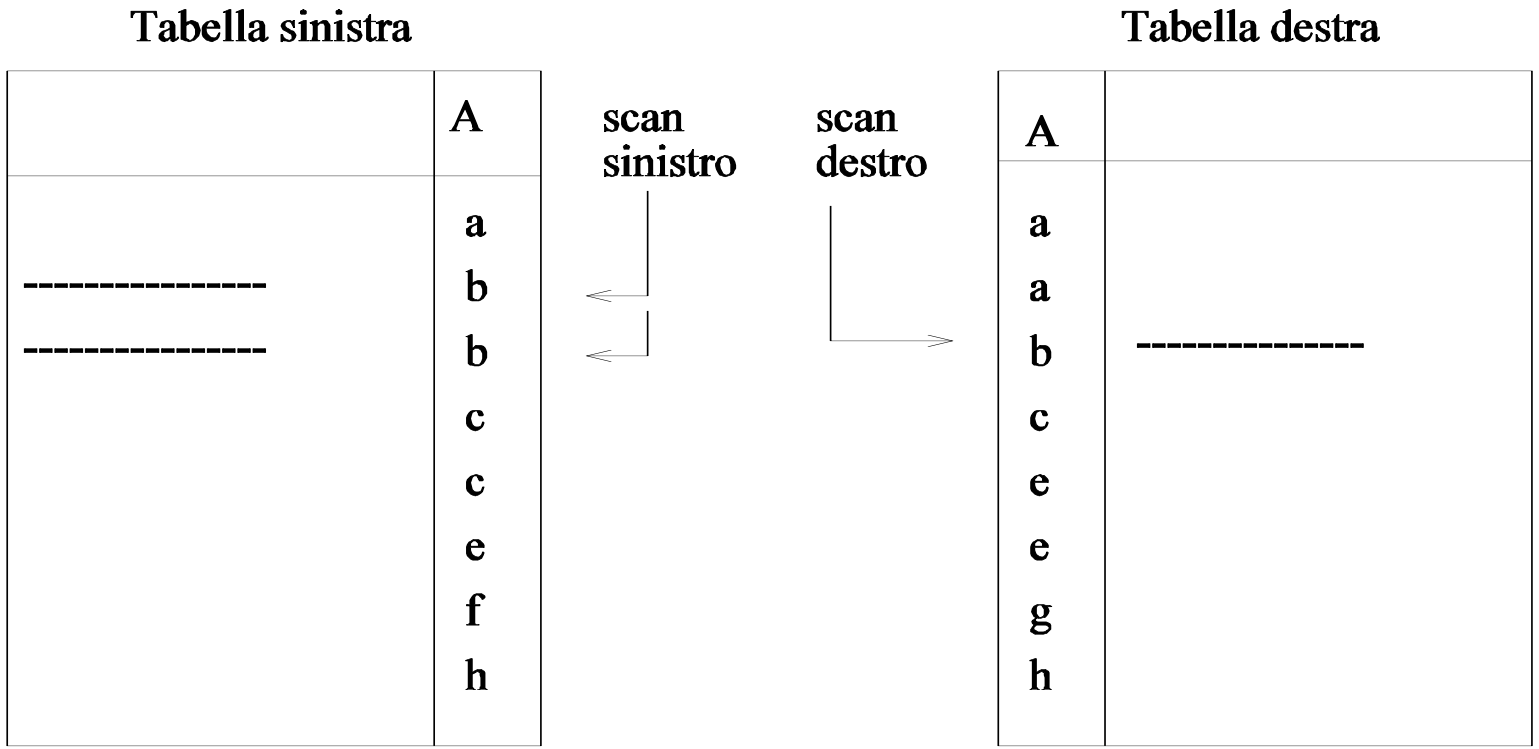
- **Sort-merge join**: algoritmi basati sull'ordinamento e la fusione di risultati parziali

Hash join



Può essere usato solo per l'equi-join.
Viene applicata una funzione di hash sugli attributi di join di ogni tupla delle due tabelle. In questo modo si partizionano le due tabelle: ogni partizione contiene le tuple che hanno lo stesso valore di hash. Dato che valori uguali saranno in partizioni corrispondenti, il join viene fatto solo tra le partizioni con lo stesso valore di hash

Sort-merge join



Si scandiscono contemporaneamente le due tabelle, che sono ordinate sugli attributi di join.

Se le due tabelle sono già ordinate fisicamente, è l'algoritmo più efficiente. Altrimenti, si possono usare indici adeguati.

Il risultato è generalmente ordinato.

Algoritmi di join

- Il DBMS sceglie l'opportuno algoritmo di join in base alla presenza di indici, alla possibilità di caricare un operando in memoria centrale, al tipo di join e all'eventuale richiesta di avere un risultato ordinato.
- Per questi motivi, un programmatore deve delegare al DBMS l'esecuzione del join e non «simularlo» con loop annidati all'interno di un programma.

Ottimizzatore fisico: conclusioni

L'ottimizzatore fisico considera ovviamente anche altri operatori:

- Unione e differenza insiemistica
- Proiezioni
- Richieste di ordinamento (**order by**)
- Presenza del **distinct**