

## Elaborazione delle interrogazioni / 31-03

Un sistema migliore si basa sulla qualità e velocità delle interrogazioni:

Il sistema cerca l'algoritmo migliore per suddividere le query; per renderla più leggibile al sistema essa viene tradotta nella notazione algebrica

$$\Pi_{B,D} [G_{R.A = "c"} \wedge S.E = 2 \wedge R.C = S.C] \quad (R \times S)$$

↓      ↓      ↓  
③      ②      ①

(l'espressione algebrica rappresenta un algoritmo logico di esecuzione delle opere su tabella e può essere rappresentata con un albero)

$$\begin{array}{c} \Pi_{B,D} \\ | \\ G_{R.A = "c"} \wedge S.E = 2 \wedge R.C = S.C \\ | \\ X \\ / \quad \backslash \\ R \quad S \end{array}$$

LQP

PIANO DI ESECUZIONE LOGICO:

albero che rappresenta l'espressione algebrica, le cui foglie sono le tabelle coinvolte

di fatto conosce solo

Posso utilizzare un'espressione algebrica diversa ma equivalente e quindi costruire un piano logico alternativo

$$\Pi_{B,D} [\sigma_{R.A = "c"}(R) \bowtie \sigma_{S.E = 2}(S)]$$

$\Pi_{B,D}$

|

$\bowtie$

/ \

$\theta_{R,A} = "c"$      $\theta_{S,C} = 2$

|

|

R

S

Posso avere due o più algoritmi:

qual è il più efficiente (e quindi sceglie il DBMS)?

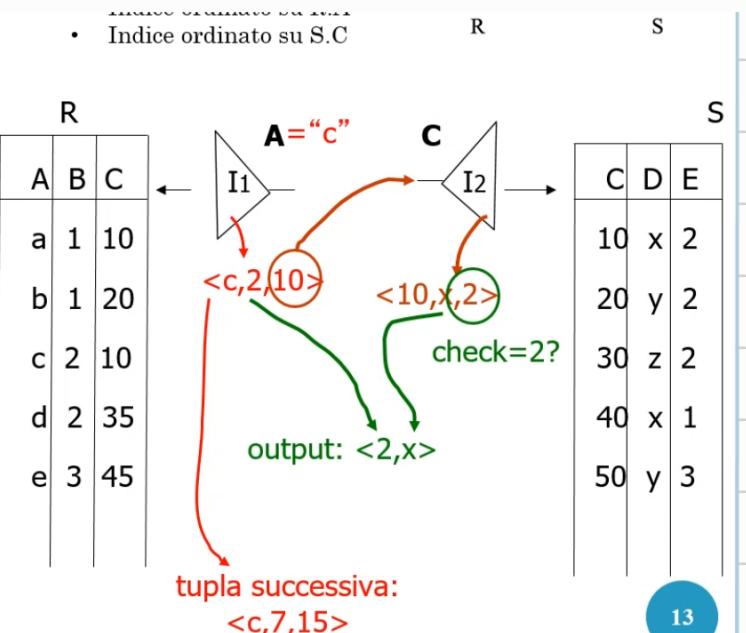
Si vogliono ridurre gli accessi a disco in generale. Nota che il prodotto cartesiano porta dietro più tuple da filtrare. Mentre nel secondo caso filtro un sottoinsieme di tuple per poi effettuarne un join. = risultati intermedi più piccoli.

I piani logici forniscono un'indicazione ma non sono esse stesse un algoritmo. È necessario passare dal piano logico a quello fisico: un algoritmo che implementa il piano logico tenendo conto di come prender DB organizzata, memorizzata e opera sui record.  
→ In output ottiene un solo piano fisico.

Si sfruttano gli indici che permettono di concentrarsi su porzioni di tabella.

#### Piano fisico

- si usa l'indice su R.A per selezionare i record corrispondenti alle tuple di R con R.A = "c"
- per ogni valore di R.C trovato, si usa l'indice su S.C per trovare i record corrispondenti alle tuple in join
- si eliminano i record di S tali che S.E ≠ 2
- si concatenano di record di R e S risultanti, proiettando su B e D e si restituisce la tupla
- si ripete il procedimento per ogni altra tupla restituita dall'indice su R.A



13

Il compito del query processor è individuare il piano fisico più efficiente e partire da quello logico e uno fisico generico in input

Allo stesso modo dei piani logici, posso avere molti piani fisici (consistenza) (molti algoritmi). Si utilizzano le strategie di costo minore.

Tale costo e' basato e ottimizzato sugli accessi e disco.

Nuovamente le apparenze, effettuare un calcolo ausiliario per scegliere il piano di esecuzione ottimale, non e' cosi' otto, in termini di tempo, puo utilizzare un piano fisico non ottimizzato per elaborare le query.

## Passi dell'elaborazione

L'idea e': passare dalla query  $\rightarrow$  traduzione  $\rightarrow$  ottimizzazione logica  
 $\rightarrow$  ottimizzazione fisica  $\rightarrow$  esecuzione del piano fisico  
"riduzione piano fisico migliore"

- Nel parsing della query (per controllare la correttezza sintattica SQL) viene rappresentato attraverso un parseTree fino alle stringhe inserite.
- Da tale rappresentazione sintattica si calcola un'espressione algebrica immediata (canonica). A sua volta rappresentata in un elenco e da questo deduce un secondo elenco oltronetivo (o piu') con l'obiettivo di elaborare (fare prime, poi ritrova) le selezioni sulle relazioni.
- Da qui si sceglie il piano fisico piu' efficiente, che riduce gli accessi al disco, considerando tutti i piani fisici e calcolandone il costo
  - $\rightarrow$  stima dimensioni dei risultati  $\rightarrow$  oltronero statistiche contenute del sistema nei cataloghi
  - $\rightarrow$  enumerazione dei piani fisici
  - $\rightarrow$  stima dei costi
  - $\rightarrow$  scelta del piano migliore

Per ogni nodo (operazione algebrica) viene corrisposto un algoritmo che possono far riferimento a indici. Può essere necessario inserire ulteriori nodi (operazioni), in particolare ordinamenti.

Inoltre si decide se memorizzare i risultati intermedi o passarli all'operatore successivo.

## Formato delle interrogazioni - ottimizzazione logica

Vengono usati operatori logici (algebra relazionale) estesi agli operatori SQL per trovare un'espressione algebrica alternativa.

Si utilizzano delle equivalenze algebriche, come regole di riduzione per ridurre la dimensione delle relazioni ottenute dagli operatori; non si usano dati statistici sul DB fisico e questo liello

espr. alg

$e_1$  ed  $e_2$  sono dette equivalenti se per ogni input  $\Delta$  restituisce uno stesso risultato quando eseguite su  $\Delta$  indipendentemente dello stato di  $\Delta$

• selezione :

commutativa	;	produce risultato intermedio	;	non lo produce
-------------	---	------------------------------	---	----------------

$$\sigma_{p_1}(\sigma_{p_2}(c)) = \sigma_{p_2}(\sigma_{p_1}(c)) \equiv \sigma_{p_1 \text{ AND } p_2}(c)$$

• proiezione : se  $\{A_1, \dots, A_n\} \subseteq \{B_1, \dots, B_m\}$

$$\pi_{A_1 \dots A_n}(\pi_{B_1 \dots B_m}(c)) = \pi_{A_1 \dots A_n}(c)$$

$$\text{es. } \pi_A(\pi_{ABC}(c)) = \pi_A(c)$$

inutile  $\{A\} \subseteq \{A, B, C\}$

- selezione e proiezione in cascata se  $P \subseteq \{A_1, \dots, A_n\}$

$$- \pi_{A_1, \dots, A_n}(G_P(e)) = G_P(\pi_{A_1, \dots, A_n}(e))$$

$\hookrightarrow$  prima filtro via delle colonne non ho A

$$\pi_{A, B}(G_{A=3}(e)) = G_{A=3}(\pi_{A, B}(e)) \quad \text{ma} \quad \pi_B(G_{A=3}(e)) \neq G'_{A=3}(\pi_B(e))$$

$$- \pi_{A_1, \dots, A_n}(G_P(e)) = \pi_{A_1, \dots, A_n}(G_P(\pi_{A_1, \dots, A_n, B_1, \dots, B_m}(e)))$$

se il predicato coinvolgesse anche attributi  $B_1, \dots, B_m$

- prodotto cartesiano e selezioni

$$- G_P(e_1 \times e_2) \equiv G(e_1) \times e_2 \quad \text{se la selezione coinvolge solo gli attributi di } e_1$$

$$- G_{P_1 \text{ AND } P_2}(e_1 \times e_2) \equiv G_{P_1}(e_1) \times G_{P_2}(e_2) \quad \text{se } P_1 \text{ coinvolge solo gli attributi di } e_1 \text{ e } P_2 \text{ di } e_2$$

$$- G_{P_1 \text{ AND } P_2}(e_1 \times e_2) \equiv G_{P_2}(G_{P_1}(e_1) \times e_2) \quad \text{se } P_1 \rightarrow e_1 \text{ ma } P_2 \text{ sia } e_1 \text{ che } e_2$$

- prodotto cartesiano e proiezione

$$\pi_{\underbrace{A_1, \dots, A_n}_{\text{list. attr.}}} (e_1 \times e_2) \equiv \pi_{\underbrace{B_1, \dots, B_m}_{\text{attr. di } e_1}}(e_1) \times \pi_{\underbrace{C_1, \dots, C_K}_{\text{attr. di } e_2}}(e_2)$$

- selezioni, prod. cartesiano, join

$$G_P(e_1 \times e_2) = e_1 \bowtie e_2$$

Tutte queste equivalenze sono già descritte all'interno del sistema  
 Nessuna espressione determina l'ordine con cui eseguire più join  
 o prodotti cartesiani in una stessa espressione algebrica nel livello logico  
 → di fatto l'ordine è deciso nelle fasi successive di ottimizzazione  
 fisica sullo base della dimensione delle relazioni e il costo  
 dell'ordine di esecuzione

Trovate heuristiche posso applicare verso che trasformano le espressioni  
 in regole di riscrittura più efficienti (a livello logico significa  
 procedure risultati intermedi più piccoli = l'algoritmo migliore)  
 Si cercano di anticipare le operazioni di selezione e proiezione che  
 riducono i risultati intermedi

### • heuristica 1

Eseguire la operazione di selezione il prima possibile

equivalenze

regola di riscrittura

$$G_p(e_1 \times e_2) \equiv G_p(e_1) \times e_2$$

$$G_p(e_1 \times e_2) \rightarrow G_p(e_1) \times e_2$$

### • heuristica 2

Eseguire la operazione di proiezione il prima possibile

equivalenze

$$\pi_{A_1, \dots, A_n}(G_p(e)) \equiv G_p(\pi_{A_1, \dots, A_n}(e))$$

$$\pi_{A_1, \dots, A_n}(e_1 \times e_2) \equiv \pi_{B_1, \dots, B_m}(e_1) \times \pi_{C_1, \dots, C_n}(e_2)$$

regole di riscrittura

$$\pi_{A_1, \dots, A_n} (G_p(c)) \rightarrow G_p(\pi_{A_1, \dots, A_n}(c))$$

$$\pi_{A_1, \dots, A_n} (e_1 \times e_2) \rightarrow \pi_{B_1, \dots, B_m}(e_1) \times \pi_{C_1, \dots, C_k}(e_2)$$

• **euristica 3**

In introduzione ulteriori proiezioni che eliminano attributi comuni non richiesti delle query

equivalezza

$$\pi_{A_1, \dots, A_n} (G_p(c)) = \pi_{A_1, \dots, A_n} (G_p(\pi_{A_1, \dots, A_n, B_1, \dots, B_m}(c)))$$

regole di riscrittura

$$\pi_{A_1, \dots, A_n} (G_p(c)) \rightarrow \pi_{A_1, \dots, A_n} (G_p(\pi_{A_1, \dots, A_n, B_1, \dots, B_m}(c)))$$

↳ logico attributi non richiesti  
delle query

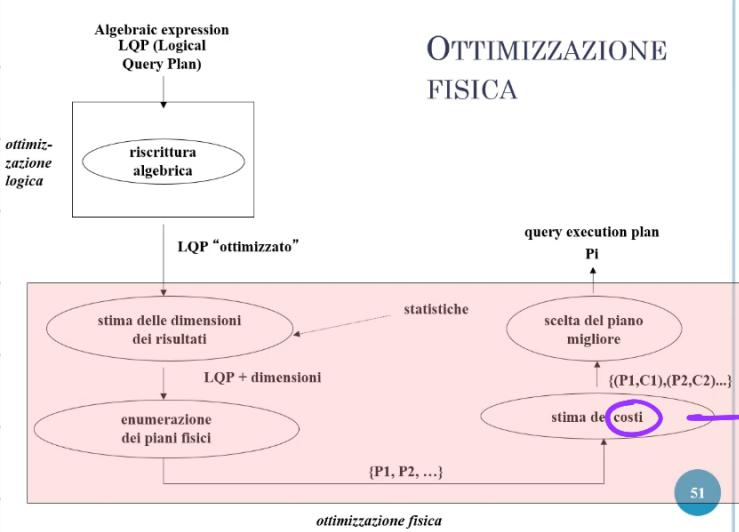
Il risultato dell'ottimizzazione è un piano logico (QP) ottimizzato,  
basato sulle riscritture delle espressioni.

Si cerca di limitare il costo dell'ottimizzazione fisica ponendogli come  
input un QP ottimizzato, molto più conveniente di passare un qualsiasi:  
piano logico (che necessita dell'ottimizzazione fisica di partire da zero  
e ottenere costi di query e selezione di migliore p. logico)

Altre heuristiche supportano le prime 3 sopra, con l'obiettivo  
di gestire condizioni complesse di selezione e proiezione.

d'ordine di esecuzione dei join e prodotti cartesiani non ha regole di riscrittura:  
vengono valutati nella fase di ottimizzazione fisica perché necessita dell'info  
sulle dimensioni delle istanze delle relazioni (ott.logica no statistiche di sistema)

# Ottimizzazione fisica



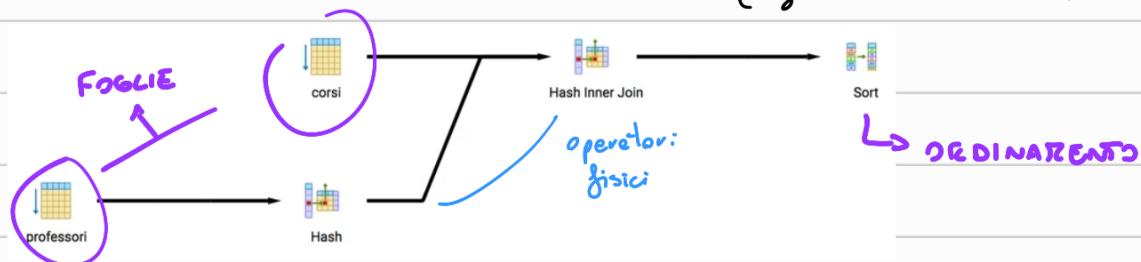
dove il costo si valuta

- \* accessi al disco

- complessità delle operazioni:

- si cercano quali algoritmi utilizzare per implementare a livello fisico gli op. logici
- come elaborare il piano complessivo

Ogni piano di esecuzione fisico si compone di una serie di operatori fisici  
comessi ad albero le cui foglie sono le relazioni di base del piano logico  
(algoritmi di accesso alle relazioni base)



- l' **operatore fisico** è una particolare implementazione di un operatore logico
- esistono diversi algoritmi di realizzazione
- gli algoritmi possono basarsi su diverse informazioni concrete a livello fisico
- gli algoritmi possono utilizzare dati statistici

Si possono raggruppare in:

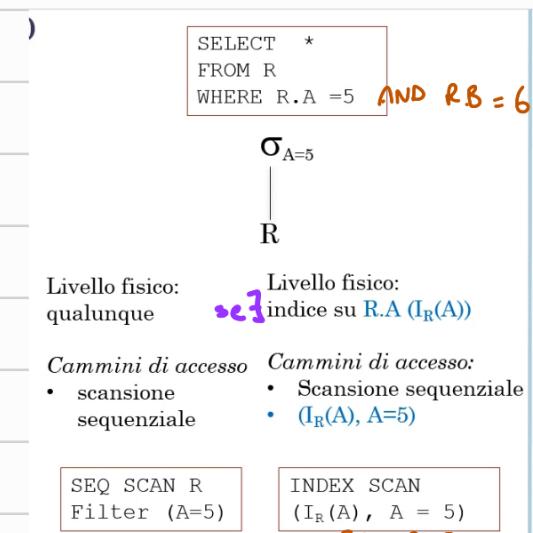
- op di iterazione: si esaminano i record (le tuple) sequentialmente uno allo allo

- op di indice: > utilizzano un indice presente a livello fisico per implementare un'operazione
- op di partizionamento: quando applicate a un insieme di tuple, suddividono queste in portioni più piccole, applicano singolarmente l'operatore e poi fondono il risultato

Il costo dell'algoritmo dipende dalle op. T/S (accessi al disco)

Non si considera il costo di scrittura poiché si è no ogni risultato di qualsiasi algoritmo viene memorizzato (oppure non viene memorizzato effetto)

Un **cammino di accesso** descrive come accedere al file di dati per ricevere tuple di interesse nelle relazioni di base nella foglie (cammino di accesso diretto nodo foglia)



Y ✓ FILTER B=6 \*  
possibili  
cammini  
( $\hookrightarrow$  filtro  
rimuovi con  
una selezione)

Per cui scegliere  
(su qualsiasi relazione base)

esempio di due cammini di accesso

\* oppure uso  $T_R(B) B=6$ , FILTER A=5 se  $T_R(B)$   
è cammino di accesso

→ o uno  
→ o l'altro  
} segnala la presenza dell'indice  
e livello fisico

non posso usare l'indice nel caso  
in cui la condizione di selezione non  
suggerisce l'indice (es.  $<>$  disegualità).  
l'indice è un sostanz. di tuple, non le  
(ovviamente, è solo una guida)

Per i booleani si usa il **fattore booleano**: quelle condizioni che se false  
rendono falsa tutta l'interrogazione atomica

Quindi se filtrate per prima posso rimuovere molte più tuple che alla fine non avrebbero rispettato la condizione interamente, es.  $(A \text{ or } B) \text{ AND } C$  (avrei prima filtro C, che fare A or B e poi C); questo determina un consumo di accesso più efficiente [es.  $I_R(C), C > 10$  Filter  $(A=5 \text{ or } B=6)$ ]

/ 07-04 (a. verso) / 23-04

$\Rightarrow$  vale sempre la scon sep.  
Possano essere usati più indici = utilizzare due consumi di accesso

AND fra folderi booleani  $\rightarrow$  INTERSEZIONE di consumi \*

OR  $\sim$   $\rightarrow$  UNIONE di consumi



non necessari avere su  
folderi booleani



INTERSEZIONE

INDEX SUN /

\ INDEX SUN

$I_R(A), A=5$

$I_R(B), B=6$

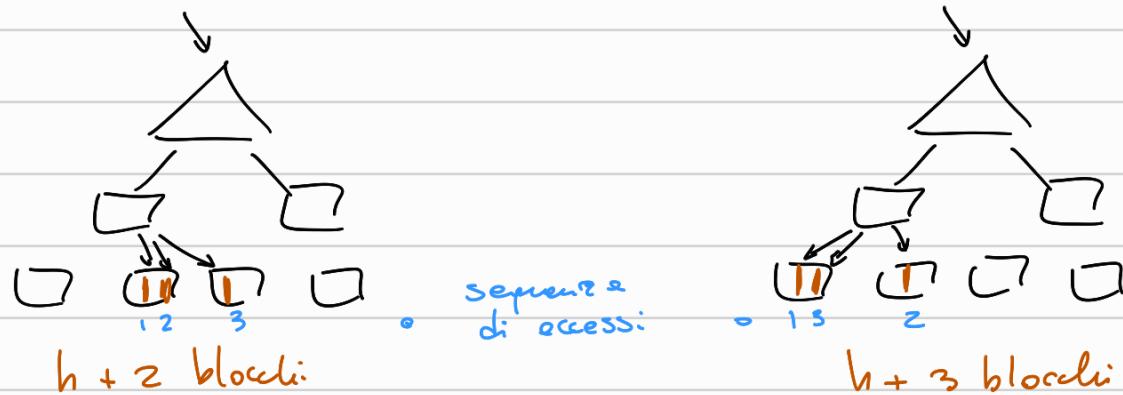


Costi

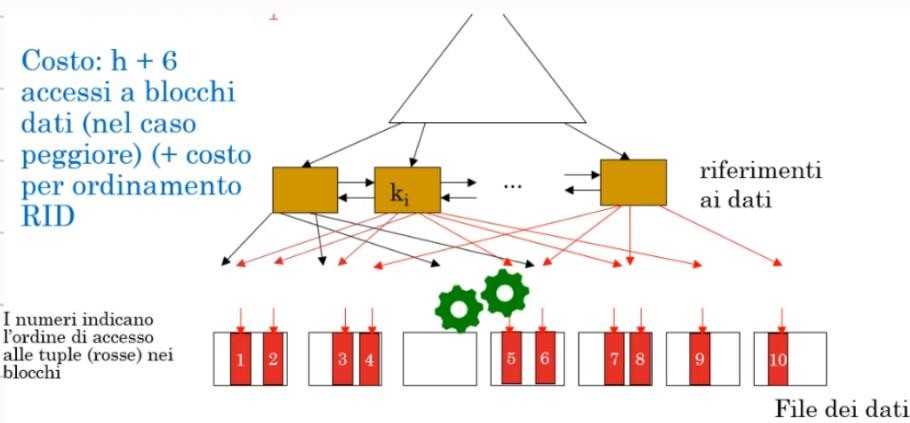
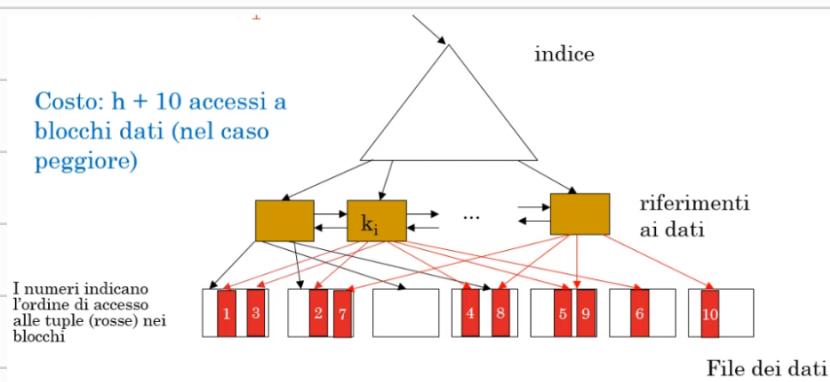
- Scansione sequenziale: necessaria in assenza di indici  
 $\rightarrow$  deve accedere a tutti i blocchi di file per R
- Accesso con indice: dipende del tipo di indice e del modo in cui le tuple che soddisfano la condizione di selezione
  - $\rightarrow$  costante  $\times$  log N
  - $\rightarrow$   $O(\log N)$  per ordinati (ad albero)
  - + coste accesso ai blocchi di dati
  - Dipende se clusterizzato (ordinato rispetto alle chiavi di ricerca)



Ciò permette di memorizzare le informazioni su blocchi dati vicini in modo sequenziale. Se non ilusterato rischio di occedere a stesso blocco dati più volte



Ancora peggio se volessi individuare un range ( $1 > k_i$ ) con indice non clusterizzato  
Alcuni sistemi in pres. caso ri-ordinano i puntatori in modo tale che l'accesso sia ordinato rispetto ai puntatori, ottenendo accessi contigui per blocchi vicini.



per la selezione

Per l'elaborazione dei nodi intermedi posso usare solo una scansione sequenziale, non posso usare gli indici perché così sono costruiti sulla struttura memoria e non sui risultati intermedi. (ho a runtime)

# Implementazione della proiezione nei nodi interni

→ SELECT

mentre i duplicati

→ SELECT DISTINCT

rimuove i duplicati

## • SELECT

Può utilizzare un alg. iterativo (scans. sep.) che toglie gli attributi che non compaiono nella proiezione. Non uso indice

Può integrare con una soluzione (che puoi usare un indice)

## • SELECT DISTINCT

Può seguire il passo precedente

Ma l'eliminazione dei duplicati è più costosa, puoi seguire:

- ordinamenti

- indici

(- hash)

→ ordinamento: si accede sequenzialmente, poi ordina le tuple, infine scandiscono il risultato ordinato (tuple uguali adiacenti)

Ner è efficiente (ordinamento non costa lineare ma logn)

(solo se partizione su veler. di base)

↑  
ordinato

→ indici: se ha un indice "su A", i suoi valori saranno nel livello foglie, un alg. può scendere e quell'unico livello e prelevare i dati attraverso i puntatori direttamente dell'indice associato ad A che vogliamo proiettare.

# Implementazione dell'ordinamento / 02-05

order by : l'ordinamento permette di rendere più efficienti operazioni successive.

Non posso usare gli alg classici perché i dati sono troppi presto in memoria principale. Posso usare

→ mergesort esterno a due fasi: ① (senza indice)

→ uso BT tree ②

① l'idea è ordinare un file di  $B(R)$  blocchi

nella memoria (buffer) ho solo  $NP + 1 \leq B(R)$  pagine disponibili

→ ordino separatamente pezzi di file di input, grandi quanto le pagine disponibili e poi ne viene fatto il merge (fase 2)

Nel buffer effettua un sort "interno", es. un quicksort fra:

record già presenti, i risultati vengono salvati in un file temporaneo

(→ file di dati a buffer)

② l'idea è ordinare sul livello foglie

→ buone idee se indice clusterizzato

→ pessime eff. indici (oltre pagg. occede a tutti i record)

grado su una pagina più volte (cost.  $B(R)$ )

Se clusterizzato posso muovere più efficientemente valori discendenti

# Implementazione del join

## Operazioni costosa

Richiede  $T(R) * T(S)$  (dove  $T = \text{tuple}$ ) confronti

Si cerca di evitare confronti:

- nested loop join ①
- index nested join ②
- merge join ③
- hash join ④

① ogni tuplo di  $R^{\text{outer}}$  si confronta con tutte quelle di  $S^{\text{inner}}$   
→ costo  $B(R) + T(R) * B(S)$   
 $\downarrow$   
= blochi di mem sec.

Se invece la relazione  $S$  (inner) è più piccola e contenuta in memoria  
centra il costo divenne  $B(R) + B(S)$ , senza dover ogni volta  
(buffer) accedere al file di dati in memoria  
ecco una volta per ogni tupla del buffer

Variante: accedo il primo blocco di  $R$  e per ogni tupla del blocco

bloch nested loop: calcolo il join con i blocchi di  $S$ . Costo  $B(R) + B(R) * B(S)$

loop

l'accesso ai blocchi di  $S$  è solo uno per blocco di  $R$   
(un "join per blocco"). Il risultato è uguale ma l'ordine può  
essere diverso da quello delle tuple di  $S$

② Supposte l'esistenza di un indice su  $A \in S$

Dato una tupla di  $R$  non è necessario scendere tutta  $S$  bensì si ricorre  
solo sull'indice di  $S$  per verificare l'ugualanza sul valore di attributo  $A$

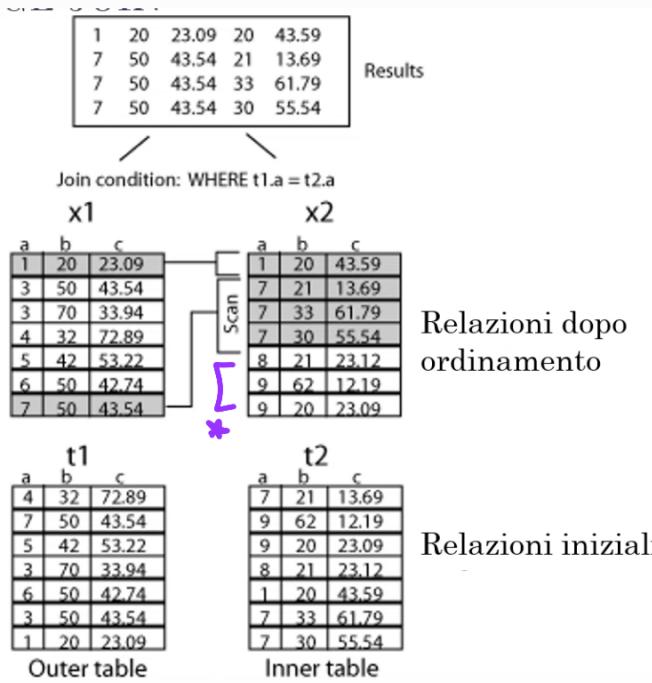
Caso:  $B(R) \times T(R)$  . Caso accesso con comune ( $t_s(A)$ ,  $A$  è tupla) <sup>a 5</sup>  
di accesso (scandiscono  $R$  e  $T$ )

Anche qui dipende se l'indice è ordinato/hash e se clusterizzato

- 3 applicabile solo quando le relazioni input sono ordinate  $\Rightarrow$  perciò  
all'ottimizzo di gain  
→ idea simile al merge sort, esito confronti utili evitano gli  
input ordinati

Caso:  $B(R) \times B(S)$  e si accede sequenzialmente alle relazioni

Soltanente usato x equi-join, altimenti perde di vantaggio



\* esito confronti utili su presto tuple, sicuramente x, non le ha

se non ordinato, devo ordinare le relazioni  
(caso aggiuntivo)

- 4 l'hash-join riduce il n° di confronti tramite una funzione di hash, applicabile solo in caso di equi-join, non necessita di indici

se  $t_r.A = t_s.B$  allora  $H(t_r.A) = H(t_s.B)$

e viceversa

oltre di gain

dove  $R.A = S.B$  se  $PJ = R.A \Delta S.B$

Si desidera quindi che se due tuple sono associate e valori diversi delle funzioni di hash (applicate agli attributi di join), allora sicuramente le tuple non possono essere in join.

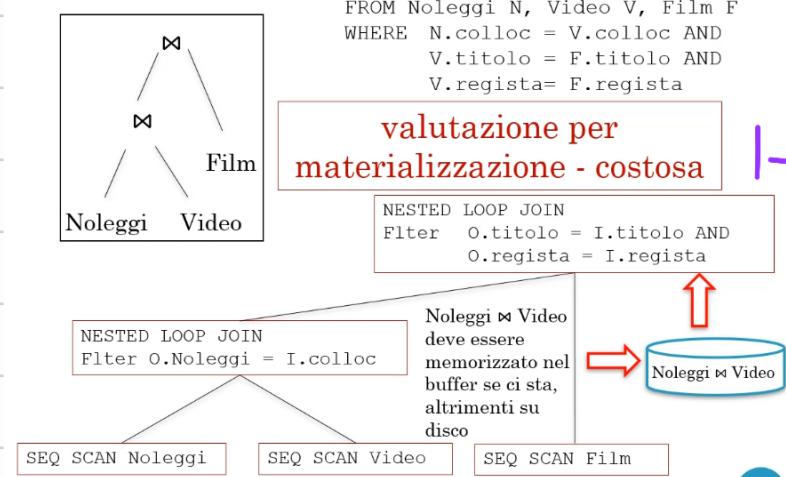
- R e S vengono partizionate sulla base dei valori di H
- le righe di tuple viene solo tra partizioni relative allo stesso valore di H (partizioni fatte da bucket)

(costo:  $B(R) + B(S)$ ) per effettuare il partizionamento.  
conveniente, spesso scelto dai sistemi

Esistono anche altri per le op di intersezione, prodotti, Cartesiane e operazioni di tipo contenevole analoghe.  
caso speciale del join

## Ottimizzazione fisica e materializzazione

### ESEMPIO



```

    SELECT P.ProjNo, e.EmpNo, D.*
    FROM Noleggi N, Video V, Film F
    WHERE N.colloc = V.colloc AND
          V.titolo = F.titolo AND
          V.regista = F.regista
  
```

valutazione per  
materializzazione - costosa

NESTED LOOP JOIN  
Filter O.titolo = I.titolo AND  
O.regista = I.regista

Noleggi & Video  
deve essere  
memorizzato nel  
buffer se ci sta,  
altrimenti su  
disco

NESTED LOOP JOIN  
Filter O.Noleggi = I.colloc

SEQ SCAN Noleggi    SEQ SCAN Video    SEQ SCAN Film

→ i risultati intermedi sono input  
di operazioni successive.  
Se grandi devono essere  
soltati (materializzati) su  
buffer (o disco)  
(oppure costoso)

Un modo alternativo è eseguire più operatori in pipeline ovvero non aspettare  
che termini un operatore per iniziare l'altro: una tupla intermedia viene via  
via passata all'op. successivo quando il primo l'ha elaborata, senza memorizzarla  
nel frattempo

(le velocità di base sono già moltiplicate)

Il pipeline si applica alla relazione outer, la inner è sempre materializzata perché utilizzata ad ogni ciclo per i confronti. Inoltre si applica solo se si effettuano dei cicli sulle relazioni (nested loop semplici, o blocchi, index) (se non c'è possibile scrive in memoria centrale)

## Stima del costo del piano di esecuzione

- approssimazione bottom-up: si stima il costo di accesso + costo nodo (operazione) + dimensione risultati intermedi (input di op. successivi)

Il sistema basa le sue stime su statistiche contenute nei cataloghi di sistema, per ogni relazione:

- Per la determinazione della stima dei costi delle varie operazioni e della dimensione del risultato, si utilizzano dati contenuti nei cataloghi di sistema
- Per ogni relazione R:
  - $T(R)$  numero di tuple nella relazione R
  - $B(R)$  numero di blocchi della relazione R
  - $S(R)$  dimensione di una tupla della relazione R in bytes (per tuple a lunghezza fissa, altrimenti si usano valori medi)
  - $S(A,R)$  dimensione dell'attributo A nella relazione R
  - $V(A,R)$  numero di valori distinti per l'attributo A nella relazione R
  - $\text{Max}(A,R)$  e  $\text{Min}(A,R)$  valori minimo e massimo dell'attributo A nella relazione R

per ogni indice:

- Per ogni indice I:
  - $K(I)$  numero di entrate (valori di chiave) dell'indice I
  - $L(I)$  numero di pagine foglia dell'indice I (se ordinato)
  - $H(I)$  altezza dell'indice I (se ordinato)
- Tali statistiche sono aggiornate alla creazione di un indice e in seguito solo periodicamente
  - aggiornarle dopo ogni modifica ai dati è troppo costoso
  - Poiché le stime dei costi sono approssimate comunque si accetta che tali valori non siano completamente accurati
- Molti DBMS prevedono un comando (ANALYZE in PostgreSQL) per richiedere esplicitamente il loro aggiornamento

Tali statistiche sono aggiornate in fase di creazione (relazione/indice),  
poi solo periodicamente.

Indirizzi infatti ad ogni modifica è costoso, ci si appoggia sulle statistiche  
comunque approssimate

Se invece è necessario lavorare su statistiche aggiornate si può  
fare con ANALYZE (PostgreSQL)

• Come si dimo il n° risultati di tuple de una selezione?

Il sistema calcola il **fattore di selettività**:

- serve a stimare lo dim del risultato
- e' la probabilità di presa una tuple generica di una relaz. R  
queste soddisfici il predicato P delle selezioni

Dunque  $\sigma_C$  selettività = 1 daff.  $F(P)$

$P=1$  od es.  $\sigma(A=1 \text{ OR } A \neq 1)$

$P$  più alto = risultato di dim maggiore

il n° di tuple risultato selezionato e'  $T(P) \cdot F(P)$

es  $\sigma_{P=1}(R)$

\* Caso: general:

$$F(P) = \frac{\text{cas possibili}}{\text{cas totali}} = \frac{1}{V(A, R)}$$

solo un valore  
(cas totali)

$\int$  altri valori di A in R

(se ha solo un valore,  $P=1$  e infatti  
può prendere quello)

- Come limitare il nr di piani fissi di cui stimare il costo?

Per  $N$  join ho  $N!$  piani da considerare!

Il sistema esclude i piani sicuramente con costo non minore

→ utilizza evasione (analogo all'ottimizzazione logica)

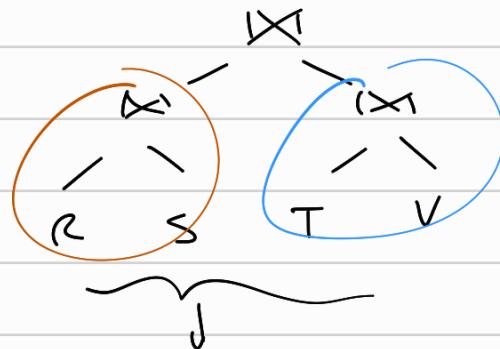
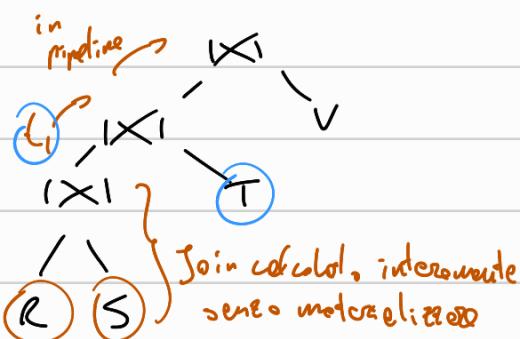
PIANO LEFT DEEP

si considerano operatori pipeline

$R \bowtie S \bowtie T \bowtie V$

left deep

non left deep



usare pipeline significa

che ogni risultato di un join  
è confrontato con un altro

(bisogna materializzare sia un join o l'altro!)

e viceversa

in pipeline prima le parti  
di X sono materializzate, ma  
la X e' già per sé relazione  
di base, nessun risultato  
intermedio usato

→ non ha materializzazioni

ogni parte