Ottimizzazione di interrogazioni

Ottimizzazione di interrogazioni

- abbiamo visto finora come organizzare i dati in un DB
- normalmente le decisioni sulle strutture da allocare sono determinate durante la **progettazione fisica** della base di dati
- la modifica di tali strutture in seguito può essere costosa
- quindi quando una interrogazione è presentata al sistema occorre determinare il modo più efficiente per eseguirla usando le strutture disponibili

Ottimizzazione di interrogazioni

- nell'elaborazione di interrogazioni si trasforma quindi una query in un **query plan**
- Esempio:

  SELECT B,D
  FROM R,S
  WHERE R.A = "c" $\wedge$ S.E = 2 $\wedge$ R.C=S.C

Ottimizzazione di interrogazioni

R	A	B	C
	a	1	10
	b	1	20
	c	2	10
	d	2	35
	e	3	45

S	C	D	E
	10	x	2
	20	y	2
	30	z	2
	40	x	1
	50	y	3

Ottimizzazione di interrogazioni

- Risposta:

B	D
2	x

- Come eseguiamo tale interrogazione?
- Una possibile strategia è:
  - fare il prodotto Cartesiano
  - selezionare le tuple
  - effettuare la proiezione

Ottimizzazione di interrogazioni

RXS

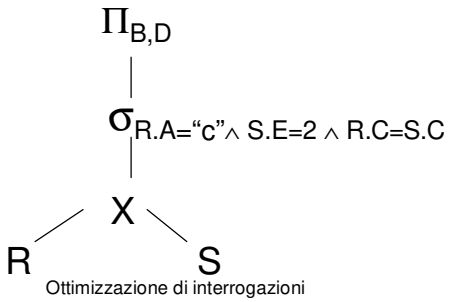
	R.A	R.B	R.C	S.C	S.D	S.E
	a	1	10	10	x	2
	a	1	10	20	y	2
	.					
	.					
→	C	2	10	10	x	2
	.					
	.					

Ottimizzazione di interrogazioni

- I piani di esecuzione possono essere descritti per mezzo dell'algebra relazionale
- il piano descritto è

$$\Pi_{B,D}\,[\,\sigma_{R.A=\text{“c”}\,\wedge\, S.E=2\,\wedge\, R.C\,=\,S.C}\,(RXS)]$$

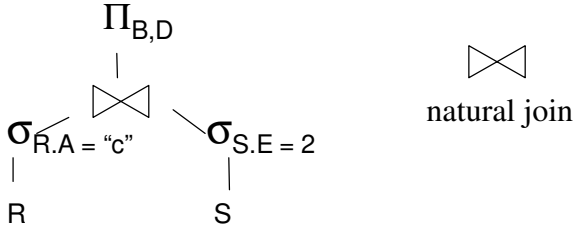
- Piano I



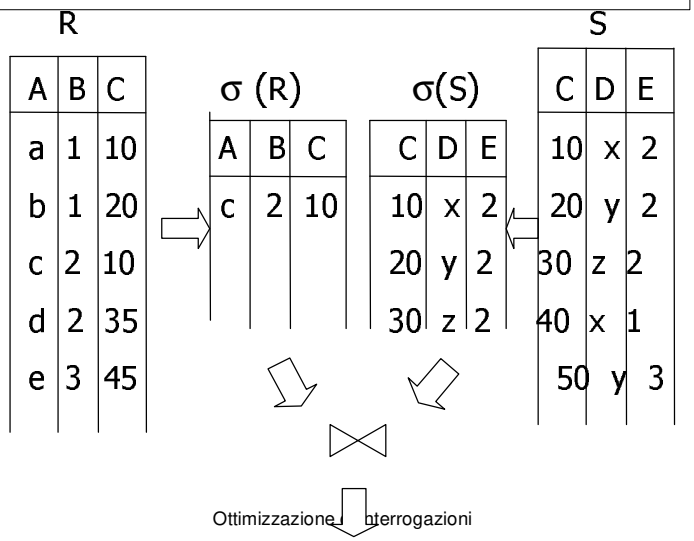
Ottimizzazione di interrogazioni

- Altra possibile strategia

Piano II



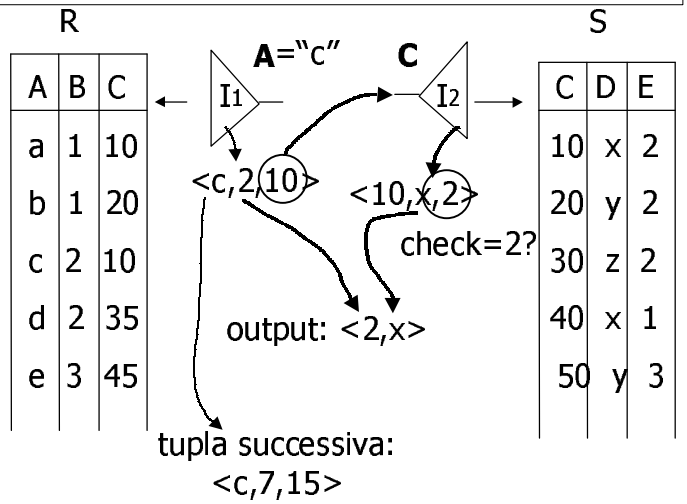
Ottimizzazione di interrogazioni



Ottimizzazione di interrogazioni

- Piano III
- si utilizzano gli indici R.A e S.C
  - si usa l'indice R.A per selezionare le tuple di R con R.A = "c"
  - per ogni valore di R.C trovato si usa l'indice su S.C per trovare le tuple che matchano
  - si eliminano le tuple di S tali che S.E ≠ 2
  - si concatenano le tuple di R e S risultanti, proiettando su B e D

Ottimizzazione di interrogazioni



A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

Ottimizzazione di interrogazioni

- per interrogazioni complesse esistono più strategie possibili
- il costo di determinare la strategia ottima può essere elevato
- il vantaggio in termini di tempo di esecuzione che se ne ricava è tuttavia tale da rendere preferibile eseguire l'ottimizzazione

Ottimizzazione di interrogazioni

Esempio

- Consideriamo le relazioni
  - Studenti(MatrS,Nome,Ind,AltreInfo)
  - Esami(Corso,MatrS,Voto,Data)
- Supponiamo di voler trovare il nome degli studenti e la data degli esami per gli studenti che hanno sostenuto BD con 30
  - SELECT Nome,Data
  - FROM Studenti NATURAL JOIN Esami
  - WHERE Corso ='BD' AND Voto =30

Ottimizzazione di interrogazioni

- Consideriamo un database con 2.000 studenti e 20.000 esami, di cui 500 di BD e di questi solo 50 con 30 (consideriamo solo la scansione sequenziale delle relazioni)
- Se si fa il prodotto cartesiano delle due relazioni, si ottiene una relazione temporanea con 40.000.000 tuple, da queste si estraggono poi le 50 tuple desiderate (costo proporzionale a 120.000.000 accessi)
- Se si selezionano i 50 esami di BD con 30 e poi si fa il join di questa relazione temporanea con Studenti si ha un costo proporzionale a 120.050

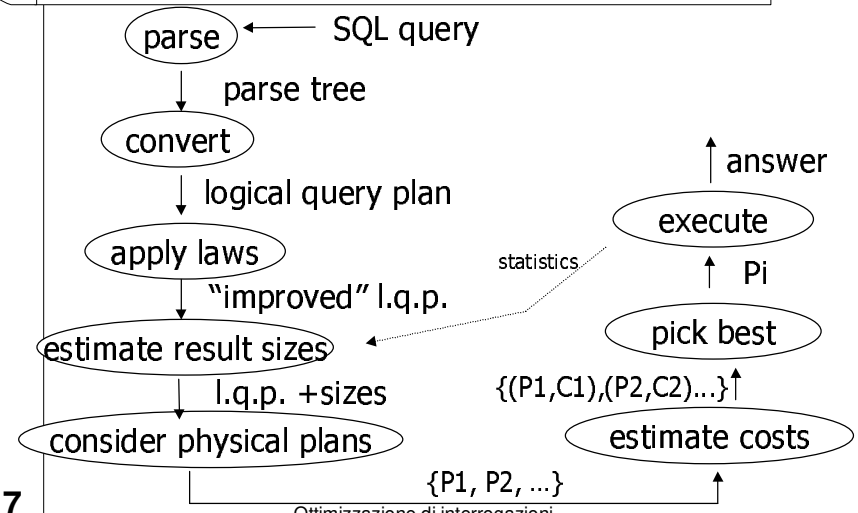
Passi nell'esecuzione di un'interrogazione

- **Parsing**
  - Viene controllata la correttezza sintattica della query e ne viene generata una rappresentazione interna (parse tree)
- **Trasformazioni algebriche**
  - La query viene trasformata in una query equivalente ma più efficiente da eseguire (ci si basa sulle proprietà dell'algebra relazionale)
  - Esempi:
    - eseguire le operazioni di selezione prima possibile
    - evitare join che rappresentano prodotti Cartesiani

Efficienza - Passi nell'esecuzione di un'interrogazione

- **Selezione della strategia**
  - si determina in modo preciso come la query sarà eseguita (per esempio si determina che indici si useranno)
  - la scelta della strategia è fatta principalmente in base al numero di accessi a disco
- **Esecuzione della strategia scelta**
  - è possibile eseguire alcuni dei passi a tempo di compilazione del programma (DB2 e System R usano questa strategia) o a tempo di esecuzione (Oracle usa questa strategia)

Passi nell'esecuzione di un'interrogazione



Passi nell'esecuzione di un'interrogazione

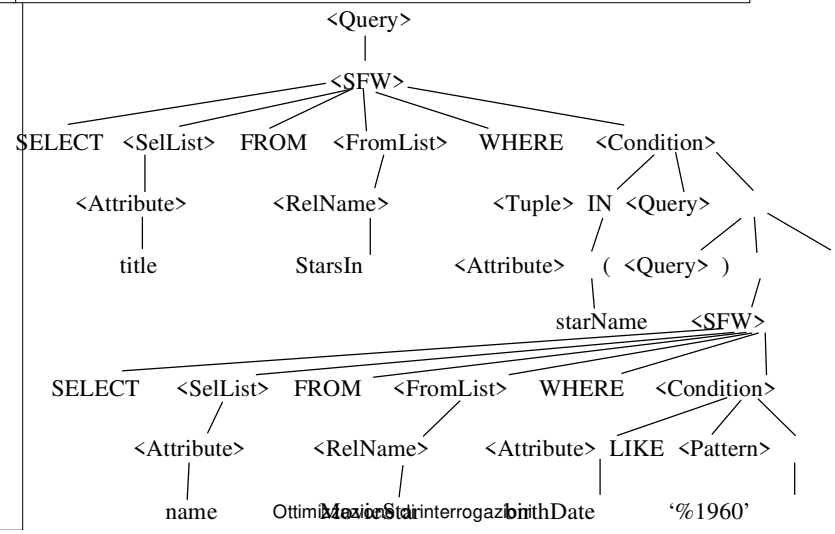
- Esempio: query SQL

```
SELECT title
FROM StarsIn
WHERE starName IN (
      SELECT name
      FROM MovieStar
      WHERE birthdate LIKE '%1960');
```

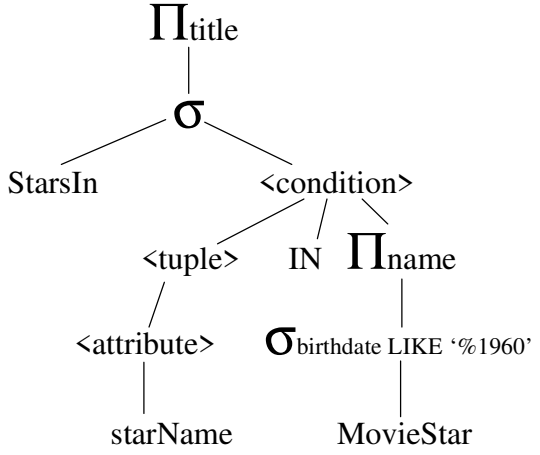
Trovare i film in cui hanno recitato attori nati nel 1960

Schema:
StarsIn(title,year,starName)
MovieStar(name, address, gender,birthdate)

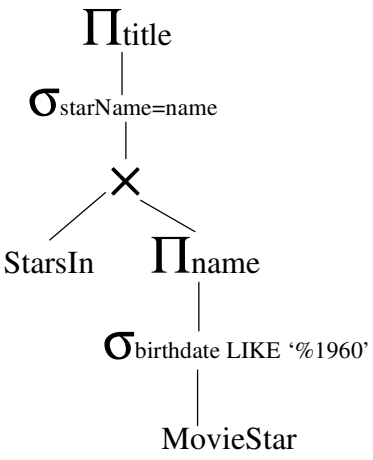
Passi nell'esecuzione di un'interrogazione - parse tree



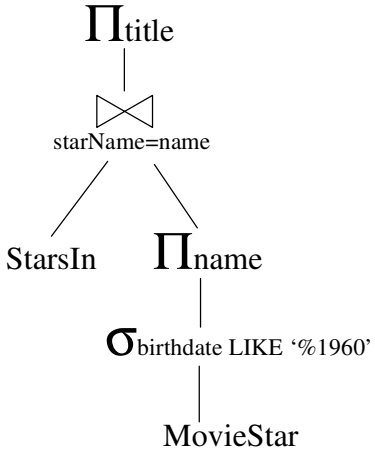
Passi nell'esecuzione di un'interrogazione - algebra



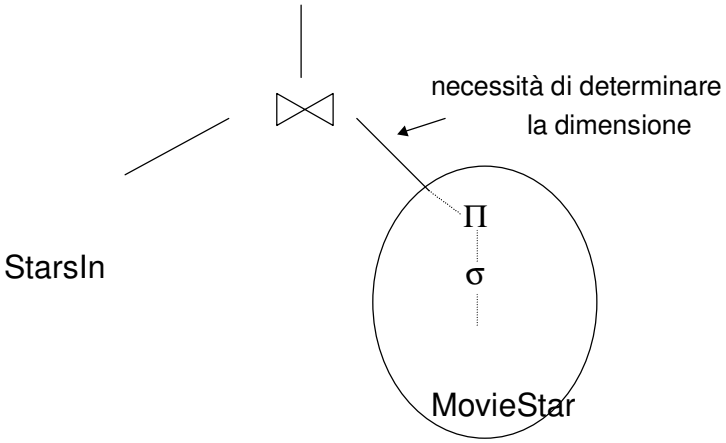
Passi nell'esecuzione di un'interrogazione: logical query plan



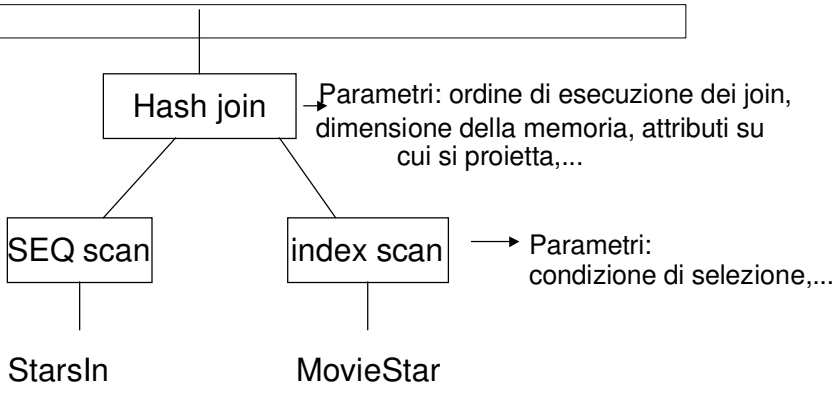
Passi nell'esecuzione di un'interrogazione: improved logical query plan



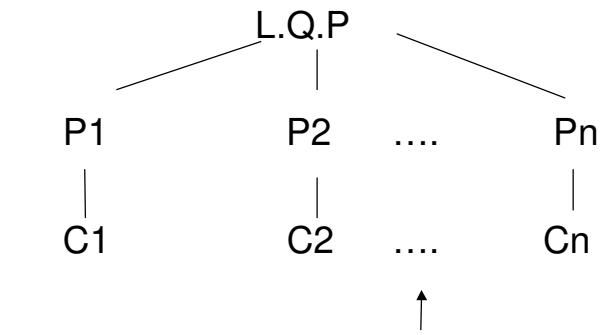
Passi nell'esecuzione di un'interrogazione:stima della dimensione del risultato



Passi nell'esecuzione di un'interrogazione: physical plan



Passi nell'esecuzione di un'interrogazione: stima dei costi



Scelta del piano di esecuzione migliore

Ottimizzazione di interrogazioni

- Un piano di esecuzione di un'interrogazione consiste quindi in:
  - un ordine di esecuzione delle varie operazioni relazionali
  - una strategia per l'esecuzione di ogni operazione
- in quanto segue vedremo:
  - come ordinare i dati su memoria secondaria (external sorting)
  - la realizzazione degli operatori relazionali (diverse strategie possibili)
  - i vari passi di ottimizzazione
    - ottimizzazione logica (regole di riscrittura)
    - determinazione dei piani di esecuzione e loro costi

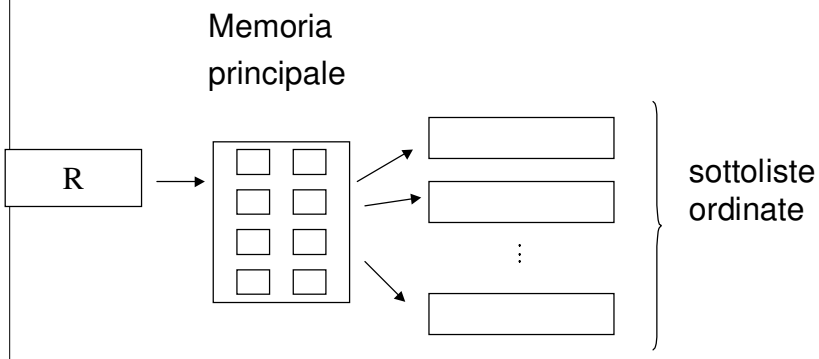
External sorting

- È spesso necessario ordinare i dati
  - perché l'interrogazione da eseguire contiene una clausola ORDER BY
  - perché avere i dati ordinati permette di eseguire altre operazioni in modo più efficiente (es. eliminazione dei duplicati, raggruppamento, join)
- non si possono usare algoritmi di ordinamento classici perché i dati da ordinare sono troppi per stare in memoria principale
- vediamo due approcci
  - two-phase multiway merge sort
  - uso di B+ tree

External sorting - Two phase multiway merge sort

- idea: si ordinano separatamente porzioni di dati che stanno in memoria e poi se ne effettua il merge
- Fase 1:
  - si riempie tutta la memoria principale disponibile con blocchi della relazione originale da ordinare
  - si ordinano i record in main memory (con uno degli algoritmi di ordinamento "classici", es. quicksort)
  - si scrivono i record ordinati su nuovi blocchi di memoria secondaria formando una sottolista ordinata
- alla fine della prima fase tutti i record della relazione sono stati letti una volta in memoria principale e sono parte di una sottolista ordinata, di dimensioni pari a quelle della MM, che è stata scitta su disco

External sorting - Two phase multiway merge sort



External sorting - Two phase multiway merge sort

- Esempio:
  - supponiamo di avere una relazione R che contiene 10.000.000 tuple (ognuna da 100b)
  - supponiamo che 50 Mb di memoria principale siano disponibili come blocchi di buffer (su 64Mb totali)
  - supponiamo che i blocchi siano di 4096
  - in memoria possiamo memorizzare 12.800 blocchi
  - in un blocco possiamo far stare 40 tuple da 100b
  - la relazione occupa così 250.000 blocchi
  - quindi riempiamo la memoria 20 volte
  - e abbiamo fatto 250.000 * 2 operazioni di I/O (molto più del costo di ordinare in memoria 10.000.000 record)
  - in realtà bisogna tenere conto che i blocchi non saranno distribuiti a caso sul disco (almeno quelli su cui si è scritto)

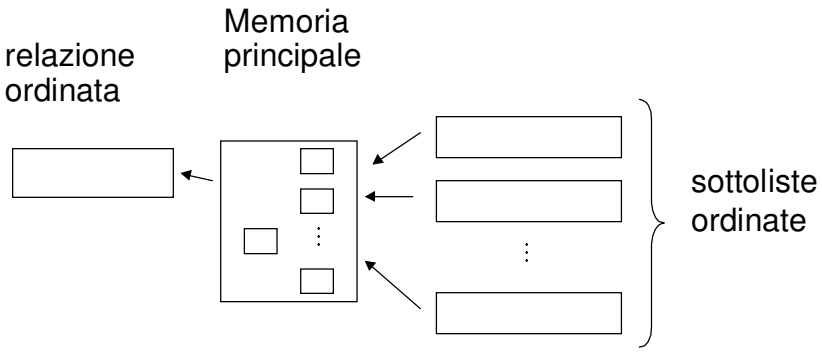
External sorting - Two phase multiway merge sort

- Fase 2:
- si effettua il merge delle sottoliste ordinate
- si potrebbe effettuare a due a due come nel merge sort classico, ma questo richiederebbe di effettuare $2 \log_2 n$ operazioni di I/O per n sottoliste
- approccio migliore: leggo in memoria il primo blocco di ogni sottolista
- posso utilizzare tutti blocchi del buffer tranne uno che mi serve per l'output

External sorting - Two phase multiway merge sort

- Fase 2 (segue):
- si effettua quindi un ciclo fino a che non si sono esaurite tutte le sottoliste ordinate in cui:
  - trovo il valore più piccolo tra i primi elementi di ogni lista
  - lo sposto nel blocco di output
  - se il blocco di output è pieno lo scarico su disco e lo reinizializzo
  - se il blocco da cui ho estratto l'elemento è vuoto, leggo il prossimo blocco di quella sottolista
  - se quella sottolista è terminata, lascio vuoto quel blocco di buffer
- nel nostro esempio si effettuano altre 250.000 * 2 operazioni di I/O

External sorting - Two phase multiway merge sort



External sorting - Two phase multiway merge sort

- Nel nostro esempio avevamo 20 sottoliste ordinate e 12.800 blocchi di buffer, è quindi possibile fare il merge delle 20 liste in un'unica passata
- supponiamo di avere
  - blocchi di dimensione B bytes
  - M bytes di memoria a disposizione
  - record di R bytes
  - il numero di blocchi è quindi M/B
  - quindi riesco ad effettuare un'unica fase di merge se ho meno di M/B - 1 sottoliste
  - ogni volta che riempiamo la memoria ordiniamo M/R record
  - quindi con questo approccio possiamo ordinare
    (M/R) * ((M/B) -1) record cioè circa M^2/RB record

External sorting - Two phase multiway merge sort

- nel nostro esempio sarebbero 6.1 miliardi di record, pari a 0.6 Terabyte
- se abbiamo relazioni più grandi (non molto comune in memoria secondaria attuale …) si può comunque aggiungere un terzo passo:
  - si usa l'approccio visto per ordinare gruppi di M^2/RB record, che vengono scritti come sottoliste ordinate
  - si riapplica la fase 2 a tali sottoliste, ordinando in un'unica relazione (M/B) - 1 di tali liste
- in questo modo riusciamo ad ordinare M^3/RB^2 record, che occupano M^3/B^2 blocchi
- nel nostro esempio 75 mila miliardi di record che occupano 7500 petabyte …

External sorting

- Sono possibili miglioramenti all'approccio visto:
  - blocked I/O (si leggono/scrivono gruppi di pagine)
  - double buffering (per non sprecare tempo di CPU mentre si effettua I/O)
  - inoltre possibilità di ordinare in parallelo

External sorting - Uso di B+ tree per ordinamento

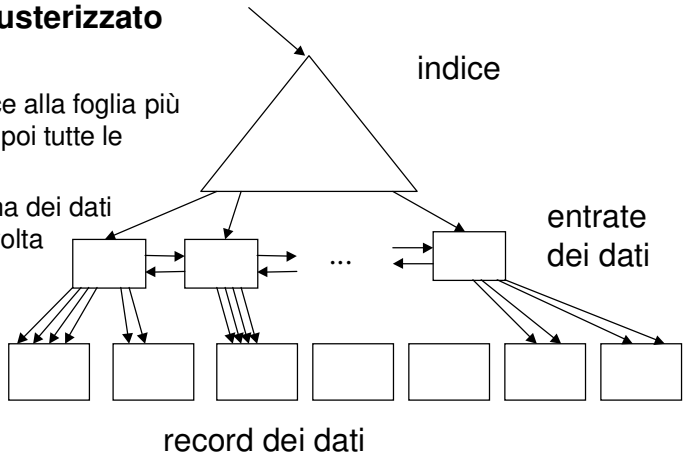
- Se la relazione da ordinare ha un indice di tipo B+ tree sull'attributo su cui vogliamo ordinare si può pensare di effettuare l'ordinamento attraversando le pagine foglia dell'indice
- se l'indice è clusterizzato è una buona idea
- se l'indice non è clusterizzato può essere una pessima idea

External sorting - Uso di B+ tree per ordinamento

B+ tree clusterizzato

costo:

- dalla radice alla foglia più a sinistra, poi tutte le foglie
- ogni pagina dei dati un'unica volta

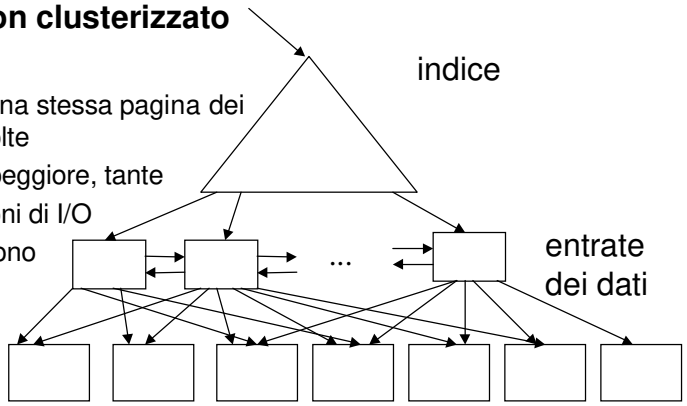


External sorting - Uso di B+ tree per ordinamento

B+ tree non clusterizzato

costo:

- torno su una stessa pagina dei dati più volte
- nel caso peggiore, tante operazioni di I/O quanti sono i record!



più avanti vedremo analisi più precisa del costo di accedere ad una relazione attraverso un indice non clusterizzato

record dei dati

Elaborazione degli operatori relazionali

- Innanzitutto bisogna precisare che faremo riferimento all'algebra relazionale, ma con alcune importanti differenze:
  - delle operazioni su insiemi (unione, intersezione, differenza) esiste versione su set e versione su bag, a seconda che si eliminino o meno i duplicati
  - la proiezione ha anch'essa due varianti, una delle quali è seguita da un'operazione di eliminazione dei duplicati
  - è prevista un'operazione di ordinamento
  - è prevista un'operazione di raggruppamento

Elaborazione degli operatori relazionali

- Le tecniche che si utilizzano per sviluppare gli algoritmi per i vari operatori sono essenzialmente tre:
  - **iterazione:** si esaminano le tuple della relazione di input sequenzialmente (scansione sequenziale)
  - **indici:** se è specificata una selezione o una condizione di join si usa un indice per esaminare solo le tuple che soddisfano la condizione
  - **partizionamento:** partizionando le tuple in base ad una chiave di ordinamento, si può decomporre un'operazione in operazioni meno costose sulle partizioni

    esempi: *ordinamento* e *hashing*

Elaborazione degli operatori relazionali

- un **cammino di accesso** ci permette di descrivere un modo alternativo di ritrovare le tuple di una relazione
- un cammino di accesso è

  (1) una scansione sequenziale, oppure

  (2) un indice più una *corrispondente* condizione di selezione (detta *predicato di ricerca*), cioè tale che:
  - esiste un indice utilizzabile per trovare le tuple che soddisfano la condizione
  - l'indice è utile per ridurre il costo della verifica della condizione (es. se condizione C+D=1.000 oppure D $\neq$ 20, con C e D attributi con indici, gli indici non aiutano a limitare il numero di tuple da visitare)

Elaborazione degli operatori relazionali

- La scansione sequenziale è sempre un possibile cammino di accesso
- se l'operazione è una selezione o un join, ed esistono gli indici corrispondenti, si possono avere diversi cammini di accesso
- il costo di un cammino di accesso è il numero di pagine visitate (sia pagine dell'indice che pagine dei dati) se si usa tale cammino di accesso per ritrovare le tuple desiderate
- in caso di più cammini di accesso si vuole scegliere quello di costo minimo (ma questo lo vedremo bene più avanti)

Elaborazione degli operatori relazionali

- Costo = numero di operazioni di I/O (non consideriamo il tempo di CPU)
- nel determinare il costo delle varie operazioni non terremo per ora in considerazione il costo di scrittura dell'output
- questo perché
  - tale costo non dipende dalla strategia utilizzata, ma solo dalla dimensione dell'output, che è uguale per tutte le strategie
  - non è detto che si voglia effettivamente scrivere su disco l'output

Elaborazione degli operatori relazionali

- Relazioni di esempio

  customer (c-name, street, c-city)

  deposit (b-name, account-number, c-name, balance)
- T(deposit) = 10,000
- T(customer) = 200
- deposit clusterizzato, con 20 tuple di deposit per blocco, B(deposit) = 500
- customer clusterizzato, con 20 tuple di customer per blocco, B(customer) = 10

Elaborazione degli operatori relazionali

- Notazione:
  - T(R) = numero di tuple della relazione R
  - B(R) = numero di blocchi/pagine della relazione R
- Osservazione importante:
  - abbiamo parlato diverse volte di indice/relazione clusterizzata e di clusterizzazione (clustering)
  - fare attenzione ai diversi significati

Elaborazione degli operatori relazionali - Selezione

- Consideriamo innanzitutto selezioni con condizioni atomiche della forma $\sigma_{attr\ op\ valore}(R)$
- consideriamo come esempio l'interrogazione

```
SELECT *
FROM deposit
WHERE balance > 10,000
```

- **Scansione sequenziale**: si scandisce l'intera relazione, si controlla la condizione su ogni tupla e si aggiunge la tupla al risultato se la condizione è soddisfatta
- il costo è di B(R) operazioni di I/O, nel nostro caso 500

Elaborazione degli operatori relazionali - Selezione

- Se non abbiamo indici e il file non è ordinato, la scansione sequenziale è l'unica strategia a disposizione
- Se invece abbiamo un indice *corrispondente* sull'attributo *attr* di selezione, allora possiamo utilizzare l'indice per trovare le entrate dei dati che soddisfano la selezione, e attraverso tali entrate accedere solo alle pagine dei dati che contengono questi dati
- Se l'indice è di tipo hash, allora può essere utilizzato solo se *op* è uguaglianza

Elaborazione degli operatori relazionali - Selezione

- Il costo dipende dal numero di tuple che soddisfano la condizione di selezione (vedremo più avanti come può essere stimato) e dal fatto che l'indice sia o meno clusterizzato
- in ogni caso è il costo di determinare le entrate dei dati (foglie dell'indice) che soddisfano la condizione più il costo di accedere ai corrispondenti blocchi dei dati (tipicamente maggiore)
- esattamente come nel caso dell'ordinamento, mentre per un indice clusterizzato ogni blocco dei dati viene visitato al più una volta, per gli indici non clusterizzati si può dover tornare sullo stesso blocco più volte

Elaborazione degli operatori relazionali - Selezione

- È però possibile (e importante!) evitare di visitare lo stesso blocco più volte per lo stesso valore della chiave di ricerca
  - si trovano le foglie dell'indice che soddisfano la condizione
  - si ordinano i rid dei record dei dati da reperire, in modo che i rid di record nello stesso blocco siano vicini
  - si accede ai record corrispondenti in ordine
- in tal modo si accede ad ogni blocco un'unica volta per ogni valore della chiave di ricerca (anche se si può comunque accedervi diverse volte, per valori diversi della chiave di ricerca)

Elaborazione degli operatori relazionali - Selezione

- Consideriamo ora il caso generale di selezione
- es. (b-name= “Chester” AND balance > 10,000) OR account-number= 18894 OR c-name = “Smith”
- queste condizioni sono innanzitutto convertite in forma normale congiuntiva (CNF)
- es. (b-name= “Chester” OR account-number= 18894 OR c-name = “Smith”) AND (balance > 10,000 OR account-number= 18894 OR c-name = “Smith”)

Elaborazione degli operatori relazionali - Selezione

- consideriamo i congiunti che non contengono OR (detti *fattori booleani*)
- gli elementi di questi congiunti se sono falsa rendono falsa tutta l’interrogazione
- è quindi possibile accedere solo alle tuple delle relazione che soddisfano tale condizione, senza dover esaminare le altre tuple
- nel caso ci siano più indici applicabili, sono possibili due approcci
- es. b-name = “Chester” AND c-name = “Smith” AND balance > 10,000 AND account-number = 18894, con indici su b-name, c-name e account-number

Elaborazione degli operatori relazionali - Selezione

- **Primo approccio**
- si utilizza comunque un solo indice
  - si determina il costo dell'accesso utilizzando i vari indici (che dipende dalla selettività del predicato e dal fatto che l'indice sia clusterizzato o meno)
  - si sceglie l'indice con costo minimo
  - si accede alle tuple della relazione attraverso tale indice
  - sulle tuple si verificano le altre condizioni
- nell'esempio, se si sceglie l'indice su account-number per ritrovare le tuple, bisogna poi controllare su ogni tupla ritrovata che b-name = "Chester" AND balance > 10,000 AND c-name = "Smith"

Elaborazione degli operatori relazionali - Selezione

- **Secondo approccio**
- si utilizzano tutti gli indici disponibili
  - si estraggono dalle foglie dell'indice i rids dei record che soddisfano la condizione
  - si effettua l'intersezione di tali insiemi di rid
  - si ritrovano i record e si verifica la parte rimanente della condizione
- nell'esempio, si utilizzano tutti gli indici per ritrovare i rid, si intersecano i tre insiemi di rid ottenuti, si ritrovano le tuple corrispondi e si controlla su ogni tupla ritrovata se balance > 10,000

Elaborazione degli operatori relazionali - Selezione

- Se invece il congiunto contiene una disgiunzione (quindi non è un fattore booleano) non è possibile utilizzare eventuali indici sugli attributi che compaiono nel congiunto come cammino di accesso
  - nota: c-name = “Smith” OR c-name = “Johnson”, in cui le condizioni in OR sono sullo stesso attributo, è equivalente a c-name IN (“Smith”, “Johnson”) e si può utilizzare
- in realtà, se ogni termine della disgiunzione ha un indice che è un predicato di ricerca quello che si potrebbe fare è ritrovare le tuple candidate utilizzando gli indici e poi farne l’unione
- la maggior parte dei sistemi esistenti non gestisce in modo efficiente le condizioni che contengono disgiunzioni

Elaborazione degli operatori relazionali - Proiezione

- Consideriamo la proiezione algebrica, quindi corrispondente a una interrogazione SELECT DISTINCT
- per implementare la proiezione bisogna:
  - rimuovere gli attributi che non compaiono nella proiezione
  - eliminare i duplicati
- il secondo passo è quello più difficile/costoso
- ci sono due algoritmi possibili, uno basato sull’ordinamento e uno basato sull’hashing (entrambi quindi basati sulla tecnica di partizionamento)

Elaborazione degli operatori relazionali - Proiezione

- **Approccio basato sull'ordinamento**
  - si accede sequenzialmente ad R per ottenere un insieme di tuple che contengono solo gli attributi desiderati
  - si ordina questo insieme di tuple con uno degli algoritmi visti usando tutti gli attributi come chiave per l'ordinamento
  - si scandisce il risultato ordinato, eliminando le tuple duplicate (che sono adiacenti)
- il costo totale è O(B(R)logB(R))

Elaborazione degli operatori relazionali - Proiezione

- in realtà l'approccio può essere migliorato integrando le operazioni nell'algoritmo di ordinamento (external merge sort)
- si modifica la prima fase in modo da eliminare gli attributi non richiesti nella proiezioine (in tal modo la dimensione delle tuple da ordinare diminuisce)
- si modifica la seconda fase in modo da eliminare i duplicati
- costo:
  - nella prima fase: B(R) letture, scrivo lo stesso numero di tuple, ma di dimensione minore
  - nella seconda fase: stesse letture, meno scritture perché non scrivo tuple duplicate (la differenza dipende dal numero di tuple duplicate)

Elaborazione degli operatori relazionali - Proiezione

- **Approccio basato sull'hashing**
- memoria principale organizzata in B buffer
- *fase di partizionamento:*
  - si legge R usando un input buffer
  - per ogni tupla, si eliminano i campi su cui non si proietta e si applica una funzione hash *h1* su tutti i campi rimasti per scegliere uno dei B - 1 buffer di output
- alla fine abbiamo B - 1 partizioni (di tuple con solo i campi richiesti)
- due tuple in partizioni diverse sono sicuramente distinte

Elaborazione degli operatori relazionali - Proiezione

- *Fase di eliminazione dei duplicati:*
- si legge ogni partizione e si costruisce una hash table in memoria, usando una funzione *h2* (diversa da *h1*!) su tutti i campi
- se una tupla va a finire nello stesso bucket di una tupla esistente, si controlla che non sia un duplicato e se lo è la si elimina
- lo scopo di utilizzare *h2* è di distribuire le tuple in una partizione in buffer diversi, in modo da minimizzare le collisioni
- se la partizione non sta in memoria, si può applicare ricorsivamente l'algoritmo alla partizione

Elaborazione degli operatori relazionali - Proiezione

- costo:
  - nella prima fase: B(R) letture, scrivo lo stesso numero di tuple, ma di dimensione minore
  - nella seconda fase: stesse letture, meno scritture perché non scrivo tuple duplicate (la differenza dipende dal numero di tuple duplicate)
- l'approccio basato sull'ordinamento si comporta meglio di quello basato sull'hashing se ci sono molti duplicati o se la distribuzione hash non è molto uniforme, inoltre ha il side effect di ordinare la relazione, che può essere utile per operazioni successive $\Rightarrow$ è il più utilizzato
- se si deve proiettare solo su un attributo su cui si ha un indice (denso), si possono accedere solo le foglie dell'indice invece del file dei dati (index-only scan)

Elaborazione degli operatori relazionali - Join

- Consideriamo il join naturale tra due relazioni, con un solo attributo in comune
- relazioni di esempio

  customer (c-name, street, c-city)

  deposit (b-name, account-number, c-name, balance)
  - T(deposit) = 10,000
  - T(customer) = 200
  - deposit clusterizzato, con 20 tuple di deposit per blocco, B(deposit) = 500
  - customer clusterizzato, con 20 tuple di customer per blocco, B(customer) = 10
- SELECT * FROM customer NATURAL JOIN deposit

Elaborazione degli operatori relazionali - Join

- **Iterazione semplice (simple nested loop)**
- si accede ad una tupla di deposit (*outer relation*) e si confronta con ogni tupla di customer (*inner relation*)

```
for each tuple d in deposit do
    begin
        for each tuple c in customer do
            begin
              test pair (d,c) to see if  a tuple should be
              added to the result
            end
    end
```


Elaborazione degli operatori relazionali - Join

- ogni tupla di deposit è letta una sola volta; questo può richiedere un massimo di 10,000 accessi (T(R))
- poiché le tuple sono clusterizzate allora il costo diminuisce sensibilmente, il numero totale di accessi per la relazione deposit è B(R) cioè 500
- per quanto riguarda le tuple di customer queste vengono accedute per ogni tupla di deposit; quindi ogni tupla di customer viene acceduta 10,000 volte
- poichè le tuple di customer sono 200, il totale degli accessi sarebbe T(R) * T(S) = 2,000,000
- poichè le tuple di customer sono clusterizzate, il totale degli accessi si riduce a T(R) * B(S) =100,000
- il costo totale nel caso in cui entrambe le relazioni sono clusterizzate è B(R) + (T(R) * B(S)) = 500 + 100,000 = 100,500

Elaborazione degli operatori relazionali - Join

- **Iterazione orientata ai blocchi (block nested loop)**
- è possibile migliorare la strategia precedente se si elaborano le relazioni sulla base dei blocchi e non delle tuple
- è utile principalmente quando le tuple di una stessa relazione sono clusterizzate

Elaborazione degli operatori relazionali - Join

```
for each block Bd of  deposit do
   begin
        for each block Bc of  customer do
                begin
                   for each tuple b in Bd do
                          begin
                             for each tuple c in Bc do
                                begin
                                   test pair (b,c) to see if  a tuple should
                                   be added to the result
                                end
                             end
                end
        end
```

Elaborazione degli operatori relazionali - Join

- Questa strategia esegue il join esaminando un intero blocco di tuple di deposit alla volta
- il costo dell'accesso a deposit è lo stesso della strategia di iterazione semplice (i.e. 500)
- dobbiamo comunque scandire la relazione customer più volte (ad un costo di 10 a scansione)
- tuttavia rispetto alla strategia di iterazione semplice, dobbiamo scandire la relazione customer tante volte quanti sono i *blocchi* di deposit (B(R)) e non quante sono le tuple di deposit (T(R))
- il numero di scansioni della relazione customer è 500; quindi il costo totale di accesso a customer è 500*10=5000 accessi
- il costo totale di questa strategia è quindi B(R) + (B(R) * B(S)) = 500 + 5000 = 5500

Elaborazione degli operatori relazionali - Join

- La scelta di deposit come outer relation e customer come inner è stata arbitraria
- se avessimo usato customer come outer relation e deposit come inner il costo finale della strategia sarebbe stato 5010
- un vantaggio nell'usare la relazione più piccola come inner è che se la relazione è piccola abbastanza può risiedere tutto il tempo in MM
- se ad esempio customer fosse abbastanza piccola da risiedere in memoria, la strategia richiederebbe solo 500 accessi per leggere deposit e 10 per leggere customer, per un totale di 510 (B(R) + B(S))

Elaborazione degli operatori relazionali - Join

- **Uso di indici (index nested loop)**
- se una delle due relazioni ha un indice sull'attributo di join, conviene renderla inner e sfruttare il join
- supponiamo che esista un indice sull'attributo c-name della relazione customer
- consideriamo la strategia di iterazione semplice
- data una tupla d di deposit non è più necessario scandire l'intera relazione customer, ma è sufficiente eseguire una ricerca sull'indice con il valore di c-name dato da d[c-name]

Elaborazione degli operatori relazionali - Join

```
for each tuple d of  deposit do
   begin
    for each tuple c of  customer s.t. c[c-name] = d[c-name] do
                begin
                        add pair (b,c) to the result
                end
   end
```

Elaborazione degli operatori relazionali - Join

- si devono eseguire 500 accessi per leggere la relazione deposit
- se ogni blocco dell'indice contiene 20 entrate, poiché T(customer) = 200 la scansione dell'indice ha un costo di 2 accessi
- il costo di accesso ai dati varia a seconda che l'indice sia o meno clusterizzato
- nel primo caso tipicamente ho un unico accesso
- nel secondo caso, posso arrivare ad avere un accesso per ogni tupla matching
- rispetto all'iterazione semplice, in ogni caso, è necessario eseguire 3 accessi per ogni tupla di deposit, invece di 200; il costo totale della strategia nel caso peggiore è di 30,500 accessi

Elaborazione degli operatori relazionali - Join

- **Merge join**
- nel caso in cui nessuna delle due relazioni sia piccola abbastanza da poter risiedere in MM è possibile eseguire efficientemente il join se entrambe le relazioni sono ordinate in base al valore dell'attributo di join
- relazioni customer e deposit ordinate in base all'attributo c-name
- l'operazione di merge-join richiede di associare inizialmente un puntatore ad ogni relazione
- i puntatori inizialmente puntano alla prima tupla di ogni relazione
- poichè le tuple sono ordinate in base all'attributo di join ogni tupla viene letta esattamente una volta

Elaborazione degli operatori relazionali - Join

- il costo totale è B(R) + B(S), nel caso delle relazioni customer e deposit il costo totale sarebbe 510 accessi
- l'algoritmo non richiede che la relazione stia tutta in MM; è sufficiente che tutte le tuple con lo stesso valore dell'attributo di join stiano in MM
- questo è possibile anche per relazioni di ampie dimensioni
- lo svantaggio principale di questo metodo è che richiede che le relazioni siano ordinate
- tuttavia, poichè è molto efficiente può convenire ordinare le relazioni prima di eseguire il join

Elaborazione degli operatori relazionali - Join

```
pd := address of first tuple of deposit;
pc := address of first tuple of customer;
while (pc ~= null) do
    begin
        t_c :=tuple to which pc points;
        S_c:= {t_c};
        set pc to point to  next tuple of customer;
        done := false;
        while (not done) do
            begin
                t_c ':=tuple to which pc points;
                if t_c [customer-name] =t_c '[customer-name]
                then begin
                    S_c:= S_c U {t_c'};
                    set pc  to point to next tuple of customer;
                end
                else done:=true;
            end
```

Elaborazione degli operatori relazionali - Join

```
td:=tuple to which pd points;
        while (td [customer-name] < tc [customer-name]) do
                begin
                        set pd to point to  next tuple of deposit;
                        td:=tuple to which pd points;
                end
        while (td [customer-name] = tc [customer-name]) do
                begin
                        for each t in Sc do
                                begin
                                  compute t |x| td and add this to the result;
                                end
                        set pd to next tuple of deposit;
                        td:=tuple to which pd points;

                end
end
```


Elaborazione degli operatori relazionali - Join

D	N	B			C	N	D	
	a1	b1	i1	← pd		a1	c1	i6 ← pc
	a1	b2	i2			a1	c3	i7
	a2	b4	i3			a3	c5	i8
	a2	b5	i4			a4	c7	i9
	a5	b7	i5			a5	c9	i10

- tc=<a1,c1> Sc={<a1,c1>}

Elaborazione degli operatori relazionali - Join

D	N	B			C	N	D	
	a1	b1	i1	← pd		a1	c1	i6
	a1	b2	i2			a1	c3	i7 ← pc
	a2	b4	i3			a3	c5	i8
	a2	b5	i4			a4	c7	i9
	a5	b7	i5			a5	c9	i10

- tc'=<a1,c3> e poiché tc'[N]=tc[N] Sc={<a1,c1>, <a1,c3>}

Elaborazione degli operatori relazionali - Join

D	N	B			C	N	D	
	a1	b1	i1	← pd		a1	c1	i6
	a1	b2	i2			a1	c3	i7
	a2	b4	i3			a3	c5	i8 ← pc
	a2	b5	i4			a4	c7	i9
	a5	b7	i5			a5	c9	i10

- tc'=<a3,c5>, tc'[N]<>tc[N], td=<a1,b1>
  - td[N]<tc[N] ? NO
  - td[N]=tc[N] ? SI
- Si esegue il join tra la tupla td corrente e tutte le tuple in Sc
- R={<a1,b1,c1>, <a1,b1,c3>}

Elaborazione degli operatori relazionali - Join

D	N	B			C	N	D	
	a1	b1	i1			a1	c1	i6
	a1	b2	i2	← pd		a1	c3	i7
	a2	b4	i3			a3	c5	i8 ← pc
	a2	b5	i4			a4	c7	i9
	a5	b7	i5			a5	c9	i10

- td=<a1,b2>
- td[N]=tc[N] ? SI
- R = {<a1,b1,c1>, <a1,b1,c3>, <a1,b2,c1>, <a1,b2,c3>}

Elaborazione degli operatori relazionali - Join

D	N	B			C	N	D	
	a1	b1	i1			a1	c1	i6
	a1	b2	i2			a1	c3	i7
	a2	b4	i3	← pd		a3	c5	i8 ← pc
	a2	b5	i4			a4	c7	i9
	a5	b7	i5			a5	c9	i10

- td=<a2,b4>
- td[N]=tc[N] ? NO
- tc=<a3,c5> Sc={<a3,c5>}

Elaborazione degli operatori relazionali - Join

D	N	B		C	N	D	
	a1	b1	i1		a1	c1	i6
	a1	b2	i2		a1	c3	i7
	a2	b4	i3 ← pd		a3	c5	i8 ← pc
	a2	b5	i4		a4	c7	i9
	a5	b7	i5		a5	c9	i10

- tc'=<a4,c7>
- tc[N]=tc'[N]? NO
- td[N] < tc[N] ? SI (td[N]=a2 tc[N]=a3)

Elaborazione degli operatori relazionali - Join

D	N	B		C	N	D	
	a1	b1	i1		a1	c1	i6
	a1	b2	i2		a1	c3	i7
	a2	b4	i3		a3	c5	i8 ← pc
	a2	b5	i4 ← pd		a4	c7	i9
	a5	b7	i5		a5	c9	i10

- td=<a2,b5>
- td[N] <tc[N]? SI (td[N]=a2 tc[N]=a3)

Elaborazione degli operatori relazionali - Join

D	N	B		C	N	D	
	a1	b1	i1		a1	c1	i6
	a1	b2	i2		a1	c3	i7
	a2	b4	i3		a3	c5	i8 ← pc
	a2	b5	i4		a4	c7	i9
	a5	b7	i5 ← pd		a5	c9	i10

- td=<a5,b7>
- td[N]<tc[N]? NO (td[N]=a5 tc[N]=a3)
- td[N]=tc[N]? NO

Elaborazione degli operatori relazionali - Join

D	N	B		C	N	D	
	a1	b1	i1		a1	c1	i6
	a1	b2	i2		a1	c3	i7
	a2	b4	i3		a3	c5	i8
	a2	b5	i4		a4	c7	i9 ← pc
	a5	b7	i5 ← pd		a5	c9	i10

- td=<a5,b7>
- td[N]<tc[N]? NO (td[N]=a5 tc[N]=a4)
- td[N]=tc[N]? NO

Elaborazione degli operatori relazionali - Join

D	N	B		C	N	D	
	a1	b1	i1		a1	c1	i6
	a1	b2	i2		a1	c3	i7
	a2	b4	i3		a3	c5	i8
	a2	b5	i4		a4	c7	i9
	a5	b7	i5 ← pd		a5	c9	i10 ← pc

- tc=<a5,c9> td[N]=tc[N]? SI
- <a5,b7,c9> viene aggiunto al risultato

Elaborazione degli operatori relazionali - Join

- Se ognuno degli insiemi Sc è abbastanza piccolo da stare in memoria, ogni blocco delle due relazioni viene acceduto un'unica volta, quindi il costo è B(R) + B(S)
- se le relazioni non sono già ordinate, la fase di merge (fase 2) dell'algoritmo di external merge sort può essere combinata con il merge richiesto dal join
- si ottengono cioè le sottoliste ordinate per entrambe le relazioni e si carica in memoria il primo blocco di ogni sottolista di ogni relazione, il merge delle sottoliste e il controllo della condizione di join vengono effettuati contemporaneamente

Elaborazione degli operatori relazionali - Join

- **Hash join**
- in una **partitioning phase** sia R che S sono partizionate usando una funzione hash *h*, in modo tale che le tuple di R nella partizione *i* matcharanno solo con tuple di S nella partizione *i*
- in una **matching phase** si legge una partizione di R, e ad ogni elemento si applica una funzione hash *h2* (diversa da *h*), si scandisce quindi la corrispondente partizione di S cercando le tuple che matchano
- il costo totale (se non ci sono overflow delle partizioni) è 3 * (B(R) + B(S))

Elaborazione degli operatori relazionali - Join

```
/* partitioning phase - h hash function that distribute to 1..k*/
for each tuple  d of  deposit do
    read d and add it to buffer page h(d)
for each tuple  c of  customer do
    read c and add it to buffer page h(c)
for l  = 1, …, k do                    /* matching phase */
    begin
        for each tuple  d of partition l of  deposit do
            read d and insert into hash table using h2(d)
        for each tuple  c of partition l of customer do
            begin
                read c and insert into hash table using h2(c)
                for matching deposit tuples d add (c,d) to the result
            end
    end
```

Elaborazione degli operatori relazionali - Join

- Condizioni di join più generali
- uguaglianza su più di un attributo
  - si può usare un indice su uno qualsiasi degli attributi
  - in merge e hash join, si deve ordinare/partizionare su tutti gli attributi di join
- disuguaglianze
  - si può usare l'index nested loop se l'indice è di tipo B+ tree
  - merge e hash join non sono applicabili
  - probabilmente in questo caso l'iterazione orientata ai blocchi è il metodo più efficiente

Elaborazione degli operatori relazionali - Operazioni insiemistiche

- L'intersezione e il prodotto Cartesiano sono casi speciali del join
- l'unione (senza duplicati) e la differenza sono simili
- consideriamo l'unione
- approccio basato su ordinamento:
  - si ordinano entrambe le relazioni (sulla combinazione di tutti gli attributi)
  - si scandiscono le relazioni ordinate e se ne fa il merge (eliminando i duplicati)
- alternativa: applicare la fase 2 dell'external merge sort alle sottoliste ordinate ottenute dall'applicazione della fase 1 ad entrambe le relazioni

Elaborazione degli operatori relazionali - Operazioni insiemistiche

- approccio basato su hashing:
  - si partizionano R e S usando una funzione hash *h*
  - per ogni partizione di S, si costruisce una tabella hash in memoria, usando una funzione *h2*
  - si scandisce la corrispondente partizione di R e si aggiungono le tuple solo se non sono duplicate

Elaborazione degli operatori relazionali - Operazioni aggregate

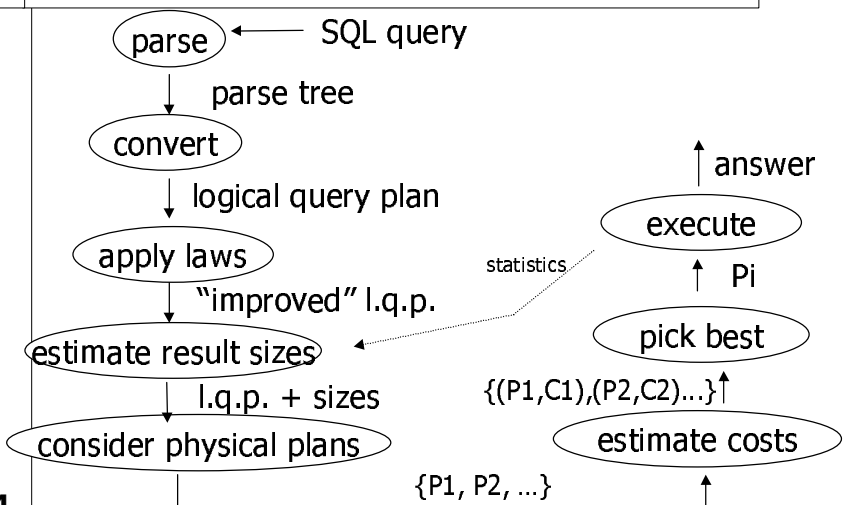
- Senza raggruppamento:
  - in generale, richiede la scansione dell'intera relazione
  - è necessario tenere alcune running information per calcolare le funzioni di gruppo

  - SUM totale dei valori esaminati
  - AVG totale e numero dei valori esaminati
  - COUNT numero dei valori esaminati
  - MIN valore più piccolo esaminato
  - MAX valore più grande esaminato

Elaborazione degli operatori relazionali - Operazioni aggregate

- con raggruppamento:
  - si può ordinare sugli attributi su cui si raggruppa, poi scandire la relazione e calcolare gli aggregati per ogni gruppo
  - può essere migliorato combinando la fase di ordinamento e quella di calcolo degli aggregati
  - approccio simile basato su hashing sugli attributi su cui si raggruppa

Ottimizzazione di interrogazioni



Ottimizzazione di interrogazioni

- **Logical query plan:** espressione algebrica (nell'algebra estesa) per l'interrogazione, rappresentata come albero
- **Physical query plan:** albero in cui, oltre all'ordine di esecuzione delle varie operazioni, vengono precisati:
  - algoritmo per la valutazione di ogni operazione
  - cammino di accesso per ogni relazione acceduta
  - eventuali fasi di ordinamento inserite
  - come i risultati delle operazioni intermedie sono passate alle operazioni successive (materialized vs pipelined)

Ottimizzazione di interrogazioni Pipelining

- Quando un'interrogazione è composta da diversi operatori, il risultato di un operatore può essere *pipelined* all'operatore successivo, senza creare una relazione temporanea in cui scrivere il risultato intermedio
- se invece l'output di un operatore viene salvato in una relazione temporanea si dice che tale risultato è *materializzato*
- il pipelining del risultato di un operatore all'operatore successivo permette di risparmiare il costo di scrivere il risultato intermedio e di rileggerlo successivamente
- poiché tale costo può essere significativo si preferisce il pipelining alla materializzazione se l'algoritmo per l'operatore lo permette

Ottimizzazione di interrogazioni Pipelining

- Se ad esempio si deve effettuare il join naturale di tre relazioni (A ⋈ B) ⋈ C si può effettuare il pipelining del risultato del primo join con il secondo
- mano a mano che si ottiene una tupla di A ⋈ B la si utilizza per effettuare il join con C, con una tecnica di tipo nested loop (con A ⋈ B come relazione outer) o con l'uso di un indice
- questo approccio ha il grande vantaggio di non dover scrivere il risultato di A ⋈ B in un file temporaneo perché le tuple di questa relazione vengono prodotte e direttamente consumate una pagina per volta

Ottimizzazione di interrogazioni Iterator interface

- Il piano di esecuzione di un'interrogazione è un albero di operatori relazionali ed è eseguito chiamando gli operatori in un qualche ordine (eventualmente interleaved)
- ogni operatore ha uno o più input e un output, che sono anch'essi nodi dell'albero
- per semplificare il codice che deve coordinare l'esecuzione di un piano di esecuzione, gli operatori che formano i nodi di tale albero tipicamente supportano una **iterator interface** uniforme, che nasconde i dettagli interni di implementazione di ogni operatore

Ottimizzazione di interrogazioni Iterator interface

- Tale interfaccia prevede le seguenti operazioni:
- **open** che inizializza lo stato dell'iteratore allocando i buffer per il suo input e output, ed è anche usata per passare argomenti (ad es. condizioni di selezione) che possono modificare il comportamento dell'operatore
- **get_next** viene effettuata su ogni elemento di input e esegue il codice specifico per l'operatore per elaborare le tuple in input, scrivendo le tuple di output nel buffer di output dell'operatore
- **close** che dealloca le informazioni di stato

Ottimizzazione di interrogazioni Iterator interface

- Tale interfaccia consente facilmente di effettuare il pipelining del risultato: la decisione di effettuare il pipelining o materializzare le tuple in input è incapsulata nel codice specifico dell'operatore che processa le tuple
- se l'algoritmo per l'operatore permette il pipelining le tuple non vengono materializzate e vengono elaborate in pipelining, altrimenti se l'algoritmo deve esaminare più volte le tuple in input, queste vengono materializzate
- tale decisione, come gli altri dettagli dell'implementazione degli operatori, è nascosta dall'iterator interface dell'operatore

Ottimizzazione di interrogazioni Iterator interface

- Tale interfaccia permette anche di incapsulare metodi di accesso quali indici (B+ o hash)
- esternamente tali metodi di accesso possono essere visti semplicemente come operatori che producano una sequenza di tuple di output
- in questo caso il metodo **open** può essere utilizzato per passare la condizione di selezione che corrisponde al cammino di accesso

Ottimizzazione di interrogazioni System R

- Gli ottimizzatori di interrogazioni degli attuali DBMS relazionali commerciali sono stati fortemente influenzati dalle scelte effettuate dall'ottimizzatore di interrogazioni del System R dell'IBM
- le scelte principali del System R includono:
  - l'uso di statistiche sul contenuto della base di dati per stimare il costo di valutazione di un piano di esecuzione
  - la scelta di considerare solo piani con join binari in cui la relazione inner è una relazione di base (questa euristica consente di ridurre il numero di piani alternativi da considerare)
  - la decisione di concentrarsi su interrogazioni SQL senza sottointerrogazioni e di trattare le sottointerrogazioni in un modo ad-hoc

Ottimizzazione di interrogazioni System R

  - la decisione di non effettuare l'eliminazione dei duplicati durante la proiezione, ma come una fase successiva se richiesto dalla clausola DISTINCT
  - un modello dei costi che tiene conto del costo di CPU oltre al costo di I/O
- in quello che vedremo seguiremo queste scelte, tranne che per il modello dei costi, per cui considereremo solo il costo di I/O

Ottimizzazione di interrogazioni Formato delle interrogazioni

- Per il momento, consideriamo interrogazioni senza sottointerrogazioni, cioè della forma

  SELECT Lista Attributi
  FROM Lista Relazioni
  WHERE condizione in CNF
  [GROUP BY Lista Attributi]
  [HAVING Condizione]
- vedremo poi come si possono trattare le interrogazioni con sottointerrogazioni

Ottimizzazione di interrogazioni Equivalenze algebriche

- Un blocco di interrogazione SQL può essere visto come un'espressione algebrica che consiste nel prodotto cartesiano delle relazioni nella clausola FROM, le selezioni nella clausola WHERE e le proiezioni nella clausola SELECT
- le equivalenze algebriche ci permettono di convertire il prodotto cartesiano in join, di scegliere un diverso ordine per effettuare i join, di portare all'interno selezioni e proiezioni
- trasformiamo un'espressione algebrica in un'interrogazione equivalente, ma che può essere valutata in modo più efficiente

Ottimizzazione di interrogazioni Equivalenze algebriche

- due espressioni dell'algebra relazionale sono dette equivalenti se, per ogni possibile relazione in input, producono lo stesso risultato in output

- **Selezione**

$$\sigma_{P1}(\sigma_{P2}(e)) \equiv \sigma_{P2}(\sigma_{P1}(e)) \equiv \sigma_{P1 \text{ AND } P2}(e))$$

  - *permette di gestire cascate di selezioni e stabilisce la commutatività della selezione*

Ottimizzazione di interrogazioni
Equivalenze algebriche

- **Proiezione**

$$\Pi_{A1,....,An}(\Pi_{B1,....,Bm}(e)) \equiv \Pi_{A1,....,An}(e)$$

  - *permette di gestire cascate di proiezioni*
  - si noti che {A1,.....,An} $\subseteq$ {B1,.....,Bm} affinchè la cascata sia legale

Ottimizzazione di interrogazioni
Equivalenze algebriche

- **Prodotto Cartesiano e join**
- commutatività
  - $e1 \rhd\lhd_F e2 \equiv e2 \rhd\lhd_F e1$
  - $e1 \rhd\lhd e2 \equiv e2 \rhd\lhd e1$
  - $e1 \times e2 \equiv e2 \times e1$
- associatività
  - $(e1 \rhd\lhd_{F1} e2) \rhd\lhd_{F2} e3 \equiv e1 \rhd\lhd_{F1} (e2 \rhd\lhd_{F2} e3)$
  - $(e1 \rhd\lhd e2) \rhd\lhd e3 \equiv e1 \rhd\lhd (e2 \rhd\lhd e3)$
  - $(e1 \times e2) \times e3 \equiv e1 \times (e2 \times e3)$

Ottimizzazione di interrogazioni
Equivalenze algebriche

- **Selezioni, proiezioni e join**
- Commutazione di selezione e proiezione
  - se una selezione con predicato P coinvolge solo gli attributi A1,.....,An, allora

$$\Pi_{A1,....,An}(\sigma_P(e)) \equiv \sigma_P(\Pi_{A1,....,An}(e))$$

  - più in generale, se il predicato P coinvolge anche gli attributi B1,.....,Bm che non sono tra gli attributi A1,.....,An allora

$$\Pi_{A1,....,An}(\sigma_P(e)) \equiv \Pi_{A1,....,An}(\sigma_P(\Pi_{A1,....,An,B1,....,Bm}(e)))$$

Ottimizzazione di interrogazioni
Equivalenze algebriche

- **Selezioni, proiezioni e join**
- Commutazione di selezione e prodotto Cartesiano
  - se una selezione con predicato P coinvolge solo gli attributi di e1, allora

$$\sigma_P(e1 \times e2) \equiv \sigma_P(e1) \times e2$$

  - come conseguenza, se P=P1 AND P2 dove P1 coinvolge solo gli attributi di e1 e P2 quelli di e2,

$$\sigma_P(e1 \times e2) \equiv \sigma_{P1}(e1) \times \sigma_{P2}(e2)$$

  - inoltre se P1 coinvolge solo attributi di e1, mentre P2 coinvolge attributi di e1 e di e2

$$\sigma_P(e1 \times e2) \equiv \sigma_{P2}(\sigma_{P1}(e1) \times e2)$$

Ottimizzazione di interrogazioni
Equivalenze algebriche

- **Selezioni, proiezioni e join**
- si può inoltre trasformare una selezione ed un prodotto cartesiano in un join, in accordo alla definizione di join

$$\sigma_P(e1 \times e2) \equiv e1 \rhd\lhd_P e2$$

- Commutazione di proiezione e prodotto Cartesiano
  - Sia A1,....,An una lista di attributi di cui gli attributi B1,....., Bm siano attributi di e1, e i rimanenti C1,.....,Ck siano attributi di e2

$$\Pi_{A1,....,An}(e1 \times e2) \equiv \Pi_{B1,....,Bm}(e1) \times \Pi_{C1,....,Ck}(e2)$$

Ottimizzazione di interrogazioni
Equivalenze algebriche

- **Altre equivalenze (unione e differenza)**
- commutatività e associatività dell'unione
  - $(e1 \cup e2) \cup e3 \equiv e1 \cup (e2 \cup e3)$
  - $e1 \cup e2 \equiv e2 \cup e1$
- commutazione di selezione e unione

$$\sigma_P(e1 \cup e2) \equiv \sigma_P(e1) \cup \sigma_P(e2)$$

- commutazione di selezione e differenza

$$\sigma_P(e1 - e2) \equiv \sigma_P(e1) - \sigma_P(e2) \equiv \sigma_P(e1) - e2$$

- commutazione di proiezione e unione

$$\Pi_{A1,....,An}(e1 \cup e2) \equiv \Pi_{A1,....,An}(e1) \cup \Pi_{A1,....An}(e2)$$

Ottimizzazione di interrogazioni
Equivalenze algebriche - euristiche

- Una volta costruito il logical query plan della query, le regole viste vengono utilizzate, secondo delle euristiche, per trasformarlo in un improved logical query plan che si spera sia più efficiente da eseguire
- un'alternativa sarebbe quella di considerare diversi equivalenti logical query plans e stimare il costo dei diversi physical plans associati ai vari piani logici considerati
- in genere questa alternativa non viene utilizzata per limitare il numero di piani da valutare

Ottimizzazione di interrogazioni
Equivalenze algebriche - euristiche

- le euristiche si basano sull'idea di anticipare il più possibile le operazioni che permettono di ridurre la dimensione dei risultati intermedi: selezione e proiezione
- l'ordine di esecuzione dei join, invece, viene deciso nella fase successiva, sulla base delle informazioni relative alla dimensione delle relazioni e alla valutazione del costo dei diversi ordini di esecuzione

Ottimizzazione di interrogazioni
Equivalenze algebriche - euristiche

- **Euristiche: selezione**
- *Eseguire le operazioni di selezione* (σ) *il più presto possibile*
- *Trasformare espressioni della forma* $\sigma_{P1\ AND\ P2}(e)$ *in espressioni della forma* $\sigma_{P1}(\sigma_{P2}(e))$ *dove P1 e P2 sono predicati ed e è una espressione algebrica*
- a volte è però conveniente portare prima all'esterno e poi di nuovo all'interno la selezione
- esempio: $(\sigma_{A=v}(R)) \rhd\lhd S$

  se S ha anch'essa attributo A l'espressione equivalente più efficiente è

  $(\sigma_{A=v}(R)) \rhd\lhd (\sigma_{A=v}(S))$

Ottimizzazione di interrogazioni
Equivalenze algebriche - euristiche

- **Euristiche: proiezione**
- *Eseguire le operazioni di proiezione* (π) *il più presto possibile*
- Si possono inoltre introdurre ulteriori proiezioni nell'espressione, gli unici attributi da non eliminare sono quelli che
  - appaiono nel risultato della query
  - sono necessari in operazioni successive
- non sempre è però conveniente introdurre queste ulteriori proiezioni

Ottimizzazione di interrogazioni
Equivalenze algebriche - euristiche

- esempio: $\pi_A (\sigma_{B = v} (R)$
  potrebbe essere trasformata in
  $\pi_A (\sigma_{B = v} (\pi_{A,B}(R))$
  se è presente un indice su B per R applicare la proiezione a tutta la relazione fa "perdere tempo" rispetto al ritrovare tramite l'indice le tuple che soddisfano la condizione $B = v$ e poi applicare solo a queste la proiezione

$\Rightarrow$ si usa "conventional wisdom" tenendo presente che nessuna trasformazione è <u>sempre</u> buona

Ottimizzazione di interrogazioni
Stima del costo di esecuzione

- A questo punto abbiamo scelto un logical query plan, che deve essere trasformato in un physical plan
- questo in genere viene fatto considerando diversi possibili piani fisici che realizzano il piano logico scelto, valutando il costo di ognuno di essi e scegliendo il piano fisico di minor costo (*cost-based enumeration*)
- per stimare il costo di un piano di esecuzione, per ogni nodo nell'albero:
  - si stima il costo di effettuare l'operazione corrispondente
  - si stima la dimensione del risultato (che sarà l'input di operatori successivi) e si determina se è ordinato

Ottimizzazione di interrogazioni
Stima del costo di esecuzione

- dato un piano logico, i corrispondenti piani fisici sono ottenuti fissando, per ogni piano fisico:
  - un ordine di esecuzione per le operazioni associative e commutative (join, unione, intersezione)
  - un algoritmo per ogni operatore nel piano logico
  - operazioni addizionali (es. scansione, ordinamento) che non sono indicate nel piano logico
  - il modo in cui i risultati intermedi sono passati da un operatore al successivo (es. materializzati o pipelining)

Ottimizzazione di interrogazioni
Statistiche

- Per la determinazione dei costi delle varie operazioni, il DBMS mantiene nei cataloghi di sistema alcune informazioni statistiche sui dati contenuti nelle relazioni
- 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
  - Max(A,R) e Min(A,R) valori minimo e massimo dell'attributo A nella relazione R

Ottimizzazione di interrogazioni Statistiche

- per ogni indice I
  - K(I) numero di entrate (valori di chiave) dell'indice I
  - L(I) numero di pagine dell'indice I (pagine foglia nel caso di indice B+)
  - H(I) altezza dell'indice I
- tali statistiche sono aggiornate in seguito al caricamento delle relazioni o alla creazione di un indice e in seguito solo periodicamente
  - aggiornarle in seguito ad ogni modifica ai dati è troppo costoso, le stime dei costi sono approssimate comunque quindi si accetta che tali valori non siano completamente accurati
- molti DBMS prevedono un comando UPDATE STATISTICS per richiedere esplicitamente il loro aggiornamento (alternativa: valutazione incrementale)

Ottimizzazione di interrogazioni Stima della dimensione del risultato

- **Proiezione**
- la dimensione di $\pi_{A1, \ldots, An}(R)$ può essere calcolata esattamente come

  T(R)* (S(A1,R) + … + S(An,R))

  [n.b. la proiezione non elimina automaticamente i duplicati]
- se vogliamo tenere conto dell'eliminazione dei duplicati si può stimare la dimensione di $\pi_A(R)$ in

  V(A,R) * S(A,R)

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione**
- Il numero di tuple restituito da una selezione $\sigma_P(R)$ dipende da quante tuple di R soddisfano il predicato P
- il rapporto tra le tuple di R che soddisfano P e tutte le tuple di R (cioè la probabilità che una tupla di R soddisfi P) viene chiamata *fattore di selettività del predicato* P ed indicata con F(P)
- è possibile stimare il fattore di selettività assumendo l'uniformità di distribuzione dei valori di ogni attributo, cioè sotto l'ipotesi che ogni valore appaia con la stessa probabilità
- si stima poi che $\sigma_P(R)$ selezioni un numero di tuple pari a T(R) * F(P)

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione - Stima del fattore di selettività**
- F(A=v) = 1/V(A,R)

  [se per qualche motivo V(A,R) non è noto, si assume F=1/10]
- F(A>v) = (Max(A,R) - v)/(Max(A,R)-Min(A,R))
- F(A<v) = (v-Min(A,R))/(Max(A,R)-Min(A,R))

  [se non si conoscono i valori di max e min dell'attributo o l'attributo non è numerico, si stima F=1/3]
- F(A **IN** $(v_1, v_2,.....,v_N)$)= N * F(A=v)
- F(A **BETWEEN** (v_1, v_2))= $(v_2 - v_1)$/(Max(A,R)-Min(A,R))

  [se non si conoscono i valori di max e min dell'attributo o l'attributo non è numerico, si stima F=1/4]

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione - Stima del fattore di selettività**
- F(A1 = A2) = 1/ MAX (V(A1,R), V(A2,R))
  [questa stima assume che per ogni valore dell'attributo con meno valori, esista un valore corrispondente nell'attributo con più valori - se i valori non sono noti si assume 1/10]
- $F(C_1 \text{ AND } C_2) = F(C_1) * F(C_2)$
- $F(C_1 \text{ OR } C_2) = F(C_1) + F(C_2) - F(C_1) * F(C_2)$
- F(NOT C) = 1 - F(C)
- per come è stato definito, è chiaro che tanto più il fattore di selettività è minore di uno, tanto più è selettivo il predicato a cui si riferisce

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

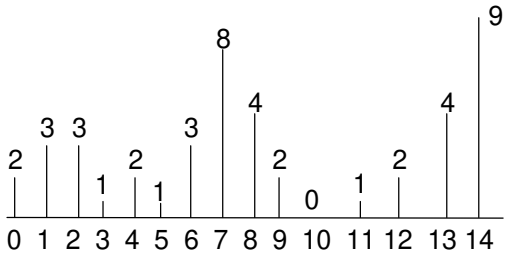
- **Selezione - Istogrammi**
- le stime viste si basano sull'assunzione che tutti i valori sono equamente probabili, e che non vi siano correlazioni tra i valori di attributi diversi
- in alcune situazioni non è realistico assumere l'equiprobabilità dei valori
- per esempio se consideriamo la relazione deposit e l'attributo b-name, ci si può aspettare che le filiali più grandi abbiano più depositi e quindi alcuni nomi di filiale appariranno con maggiore probabilità di altri

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione - Istogrammi**
- allo stesso modo nell'esempio degli impiegati è probabile che vi sia una correlazione tra la mansione di un impiegato e il suo stipendio
- tuttavia entrambe le ipotesi sono buone approssimazioni in molti casi
- solo recentemente sono state sviluppate tecniche più sofisticate basate sul mantenimento di statistiche più dettagliate (istogrammi dei valori di un attributo), che si stanno diffondendo nei sistemi commerciali

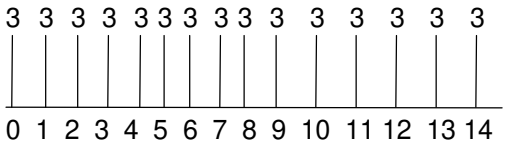
Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione - Istogrammi**
- per approssimare in modo più accurato la distribuzione dei valori dell'attributo A si può pensare di mantenere più informazioni che non il numero di valori assunti, valore minimo, valore massimo



Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione - Istogrammi**
- con l'assunzione di distribuzione uniforme viene approssimato come segue

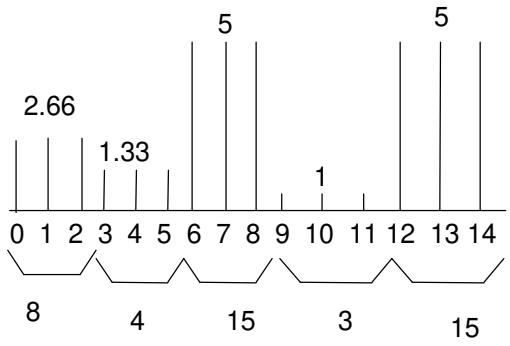


Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione - Istogrammi**
- un'istogramma è una struttura dati mantenuta dal DBMS per approssimare la distribuzione dei dati
- il range dei valori assunti dall'attributo A viene diviso in subrange detti bucket e per ogni bucket si determina quante sono le tuple di R con valori di A in quel bucket
- sono possibili due diversi modi di determinare i bucket: *equiwidth* (i subrange contengono tutti lo stesso numero di valori di A) e *equidepth* (i subrange contengono tutti lo stesso numero di tuple di R)
- inoltre a volte sono anche mantenuti i valori più frequenti e il loro numero di occorrenze (es. 7: 8 e 14: 9)

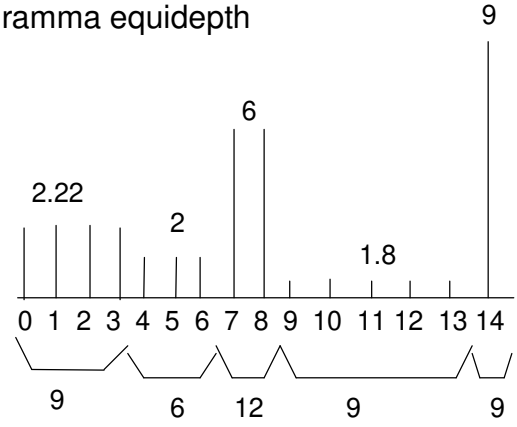
Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione - Istogrammi**
- istogramma equiwidth



Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione - Istogrammi**
- istogramma equidepth



Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Selezione - Istogrammi**
- consideriamo la condizione age > 13
- dalla distribuzione D dei valori si vede che il risultato è costituito da 9 tuple
- con l'ipotesi di equiprobabilità dei valori si stimano 1/15*45 = 3 tuple
- con l'istogramma equiwidth si stimano 1/3 * 15 = 5 tuple (la distribuzione all'interno del bucket si assume uniforme)
- con l'istogramma equidepth si stimano correttamente 9 tuple
- in generale gli istogrammi equidepth forniscono stime migliori

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Prodotto Cartesiano**
- la dimensione di $R \times S$ può essere calcolata esattamente come
  T(R)*T(S)* (S(R)+S(S))

- **Join**
- la dimensione del risultato di un theta-join viene stimata considerandolo come un prodotto Cartesiano seguito da una selezione
- consideriamo invece separatamente il join naturale

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Join naturale**
- Siano r1 e r2 due relazioni di schema rispettivamente R1 e R2
- se $R1 \cap R2 = \emptyset$, allora $r1 \bowtie r2$ è lo stesso di $r1 \times r2$ e si può usare la stessa formula di stima usata per il prodotto Cartesiano
- se $R1 \cap R2$ è una chiave per R1, allora una tupla di r2 è connessa tramite il join con al più una tupla di r1; quindi $T(r1 \bowtie r2) \leq T(r2)$
- se $R1 \cap R2$ non è una chiave né di R1 né di R2 bisogna effettuare un'altra stima

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Join naturale**
- Consideriamo una tupla t di r1, e supponiamo che $R1 \cap R2 = \{A\}$
- le tuple di r2 che hanno t[A] come valore per A sono T(r2)/V(A,r2)
- la tupla t produce T(r2)/V(A,r2) tuple di $r1 \bowtie r2$
- se consideriamo tutte le tuple in r1, allora otteniamo
  T(r1)*T(r2)/V(A,r2) *(i)*
- invertendo i ruoli di r1 e r2 nella stima, otteniamo
  T(r1)*T(r2)/V(A,r1) *(ii)*
- le due stime coincidono se V(A,r1) = V(A,r2)

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Join naturale**
- Se V(A,r2) ≠V(A,r1) allora esiste qualche tupla *dangling* che non partecipa al join
- in questo caso il minore dei due valori costituisce una stima migliore
- esempio (1): T(r1) = 4, T(r2) = 6 , V(A,r1)=2, V(A,r2)=2

r1	A	B
	a1	b1
	a1	b2
	a2	b3
	a2	b4

r2	A	C
	a1	c1
	a2	c2
	a1	c3
	a1	c6
	a2	c5
	a2	c7

T(r1)*T(r2)/V(A,r2) = (4*6)/2=12

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Join naturale**
- Esempio (2): T(r1) = 6, T(r2) = 6, V(A,r1)=3, V(A,r2)=2

r1	A	B
	a1	b1
	a1	b2
	a2	b3
	a2	b4
	a3	b2
	a3	b3

r2	A	C
	a1	c1
	a2	c2
	a1	c3
	a1	c6
	a2	c5
	a2	c7

- T(r1)*T(r2)/V(A,r2) = (6*6)/2=18
- T(r1)*T(r2)/V(A, r1) = (6*6)/3=12

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Join naturale**
- La stima T(r1)*T(r2)/max(V(A,r1),V(A,r2)) precedente non vanno molto bene se le due relazioni hanno pochi valori in comune per l'attibuto in comune; in tal caso la stima della dimensione del join risulta troppo alta
- nella pratica, d'altra parte, questi casi non si verificano molto spesso
- nel caso in cui si dovessero avere molte tuple dangling, occorre applicare dei fattori di correzione

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Join naturale**
- se le due relazioni hanno più attributi in comuni, questi vengono considerati come indipendenti
- supponiamo che $R1 \cap R2 = \{A,B\}$

  la stima diventa quindi

$$\frac{T(r1)*T(r2)}{\max(V(A,r1),V(A,r2))* \max(V(B,r1),V(B,r2))}$$

- in generale la stima si ottiene quindi moltiplicando T(r1) per T(r2) e dividendo per il massimo tra V(X,r1) e V(X,r2) per ogni attributo X in comune tra R1 e R2

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Join naturale**
- nel caso di join naturale di più di due relazioni, si considera il prodotto del numero di tuple di ogni relazione coinvolta nel join e, per ogni attributo A che compare in almeno due relazioni, lo si divide per tutti i V(A,Ri) tranne il minimo
- es. R(a,b,c), S(b,c,d), U(b,e)
  - T(R) = 1000, T(S) = 2000, T(U) = 5000
  - V(a,R) =100, V(b,R) = 20, V(c,R) =200
  - V(b,S) = 50, V(c,S) = 100, V(d,S) = 400
  - V(b,U) = 200, V(e,U) = 500

  tuple risultanti 1000*2000*5000 / 50*200*200

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Altre operazioni**
- per le altre operazioni non è facile stimare la cardinalità del risultato
- unione
  - su bag T(R1 U R2) = T(R1 + R2)
  - su insiemi varia tra T(R1 + R2) e max (T(R1),T(R2))
  - in genere si considera la media dei due
- intersezione
  - varia tra 0 e min (T(R1),T(R2))
  - in genere si considera la media dei due
- differenza
  - varia tra T(R1) e T(R1) - T(R2)
  - in genere si considera la media dei due

Ottimizzazione di interrogazioni
Stima della dimensione del risultato

- **Altre operazioni**
- eliminazione dei duplicati
  - varia tra 1 e T(R) [in realtà 0 se R è vuota]
  - un'altra stima è il prodotto dei V(Ai,R)
  - ragionevolmente si usa il più piccolo tra T(R)/2 e il prodotto dei V(Ai(R))
- raggruppamento
  - si può fare lo stesso discorso
  - ragionevolmente si usa il più piccolo tra T(R)/2 e il prodotto dei V(Ai(R)) per gli attributi Ai su cui si raggruppa

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- Data un'interrogazione un ottimizzatore dovrebbe enumerare tutti i possibili piani per eseguirla e valutarne il costo, per poi selezionare quello di costo minimo
- abbiamo già detto che già in fase di ottimizzazione logica si utilizzano alcune euristiche per non dover valutare tutti i possibili piani logici
- in quanto segue discutiamo quali piani vengono valutati e come viene determinato il loro costo
- cominciamo ad esaminare il caso di interrogazioni su una singola relazione e poi quello del join

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione**
- bisogna scegliere il cammino di accesso alla relazione
- un piano possibile è la scansione sequenziale, che ha sempre costo B(R)
- abbiamo visto che altri possibili cammini di accesso sono attraverso predicati di ricerca che siano fattori booleani della condizione di selezione
- supponiamo di utilizzare un solo indice per volta

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione**
- per determinare i cammini di accesso è necessario:
  - analizzare la condizione per ricavarne tutti i fattori booleani
  - individuare i predicati di ricerca fra i fattori booleani
  - valutare il costo di accesso con l'impiego dell'indice, per ogni predicato di ricerca
  - scegliere il predicato di ricerca più conveniente, cioè quello con costo minimo

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione**
- stimiamo in modo più preciso il costo dell'accesso con indice
- in questo caso il costo è esprimibile come CI + CD
  (costo di accesso all'indice più costo di accesso ai dati)
- indicato con C_A un predicato di ricerca, e dato $F(C_A)$ fattore di selettività di tale predicato, il costo di accesso all'indice I allocato su A, è dato da:

  $CI = F(C_A) * L(I)$

  (quindi $F(C_A)$ dà anche la stima della frazione di foglie accedute)

  [n.b. consideriamo indici B+ e trascuriamo l'accesso ai nodi intermedi dell'albero, altrimenti andrebbe aggiunto H(I)]

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione**
- La stima del costo di accesso ai dati è più complessa e dipende dalla proprietà dell'indice di essere clusterizzato oppure no
- indice clusterizzato

  $CD = F(C_A) * B(R)$
- nel caso di indice non clusterizzato
  - poichè la lista di riferimenti è ordinata, non si può verificare il caso che per un valore dell'attributo A una pagina venga visitata più volte
  - una stessa pagina può però essere visitata più volte per valori dell'attributo compresi in un certo intervallo (più liste di TID da visitare)

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione**
- indice non clusterizzato

  CD = Numero liste di TIDs * Pagine da visitare per ogni lista
- Numero di liste di TIDs = $F(C_A) * V(A,R)$
- Pagine da visitare per ogni lista = $\Phi(NTK(A),B(R),T(R))$

  dove Φ è la funzione di Yao

$$\Phi = B(R) * [1 - \prod_{i=1}^{NTK(A)} \frac{T(R) - (T(R)/B(R)) - i + 1}{T(R) - i + 1}]$$

  e NTK(A) = T(R)/V(A,R) è il numero medio di TID associati ad un valore dell'attributo

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione**
- indice non clusterizzato

  CD = $F(C_A) * V(A,R) * \Phi(NTK(A), B(R), T(R))$
- si noti che $\Phi(NTK(A), B(R), T(R))$ è sicuramente minore o uguale di tutti i suoi argomenti:
  - ogni lista di TID è lunga NTK(A), quindi al più si accedono NTK(A) pagine diverse
  - R occupa B(R) pagine, quindi sicuramente non si accedono più di B(R) pagine per ogni valore di A
  - R ha T(R) tuple in tutto!

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione**
- il costo globale di accesso è quindi
- B(R) se si sceglie la scansione sequenziale
- $CA = F(C_A)^* [L(I) + B(R)]$

  se si sceglie un indice I clusterizzato
- $CA = F(C_A)^* [L(I) + V(A,R) * \Phi(NTK(A), B(R), T(R))]$

  se si sceglie un indice I non clusterizzato

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione**
- Se si usa un indice per accedere alle tuple della relazione, le tuple vengono restituite nell'ordine dei valori dell'attributo su cui è costruito l'indice
- l'interrogazione può richiedere le tuple in un dato ordine, ad esempio se contiene una clausola ORDER BY o GROUP BY
- in questo caso è opportuno distinguere tra vie di accesso (indici o scansione sequenziale) che producono l'ordinamento richiesto e vie di accesso che non lo producono
- per le seconde è necessario calcolare anche il costo di ordinamento della relazione risultato

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione - Esempio**
- Si consideri la relazione

  Impiegato(Codice, Nome, Qualifica, Stipendio, Progetto, Città)

  supponendo che:
  - la relazione sia ordinata su Stipendio, e sia costituita da 2.000 tuple, contenute in 100 pagine, ciascuna delle quali contiene 20 tuple
  - T(Impiegati)=2000
  - B(Impiegati)=100

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione - Esempio**
  - esista un indice I clusterizzato su Stipendio
    - V(Stipendio,Impiegati)=200
    - L(I) = 9
    - Min(Stipendio) = 1.200.000 Max(Stipendio)=2.700.000
  - esista un indice I' non clusterizzato su Qualifica
    - V(Qualifica,Impiegati)=40
    - L(I') = 7
- Si vuole selezionare il Nome e lo Stipendio dei programmatori che guadagnano tra 1.500.000 e 2.000.000, e cha lavorano al progetto "Banche" o "Comuni"

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione - Esempio**
- l'interrogazione è formulata come segue

```
SELECT Nome, Stipendio
FROM Impiegati
WHERE  Qualifica = "Programmatore" AND
        Stipendio  BETWEEN (1.500.000, 2.000.000)
        AND Progetto IN ("Banche"; "Comuni")
```

- strategie alternative
  - a) scansione sequenziale
  - b) accesso con indice I (su Stipendio)
  - c) accesso con indice I' (su Qualifica)

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione - Esempio**
- i fattori di selettività sono:
  - F(Qualifica = "Programmatore") = 1/40=0.025
  - F(Stipendio BETWEEN (1.500.000, 2.000.000)) = 0.33
- NTK(Qualifica) = T(Impiegati)/V(Qualifica,Impiegati) =2.000/40=50
- i costi per le tre alternative sono:
  - a) $CA_{sequenziale}$= 100
  - b) $CA_{Qualifica}$= 0.025*[7 + 40 *Φ(50,100, 2000)]=40
  - c) $CA_{Stipendio}$= 0.33 *(9 + 100) = 36
- viene quindi preferito l'indice su Stipendio

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Interrogazioni su una singola relazione - Esempio**
- Supponiamo ora di avere anche un indice I" non clusterizzato su Progetto
  - V(Progetto,Impiegato) = 300 e L(I") = 16

  si ha
  - F(Progetto IN ("Banche", "Comuni”))=2/300=0.0066
  - NTK(Progetto)=2000/300 = 7 da cui
  - $CA_{Progetto}$= 0.0066 * [16 + 300 *Φ(7,100, 2000)] = 1 + 14=15
- in questo caso l'indice su Progetto è da preferire a quello su Stipendio

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Join**
- nel determinare i piani per l’esecuzione di un’interrogazione che coinvolge più relazioni, collegate tramite join, un aspetto importante è l’ordine di esecuzione dei join
- tale ordine, insieme alla tecnica di join utilizzata per ogni join, determina il piano di esecuzione
- nella rappresentazione ad albero del piano, si utilizza la convenzione che il figlio sinistro è la relazione outer e il figlio destro è la relazione inner

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Join**
- poiché considerare tutte le possibili alternative può essere costoso una soluzione comune è concentrarsi su piani **left-deep**
- in un albero left-deep il figlio destro di ogni nodo etichettato con un join è una relazione di base
- la motivazione per considerare solo alberi left-deep è che tali alberi permettono di generare piani di esecuzione fully-pipelined: le relazioni inner devono comunque essere materializzate quindi un piano in cui la relazione inner sia il risultato di un join forza a materializzare il risultato del join

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Join**
- per restringere lo spazio della ricerca non vengono considerate permutazioni che implicano dei prodotti cartesiani: in altra parole si cerca di eseguirli il più tardi possibile
- l'approccio seguito consiste nel determinare una soluzione ottima per ogni sottoinsieme dei join della query, e poi determinando il modo migliore per eseguire il join della relazione composita ottenuta con una ulteriore relazione
- questo approccio parte dal considerare i join di due relazioni ed ogni passo incrementa il numero di relazioni

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Join**
- Una soluzione consiste di:
  - una lista ordinata di relazioni su cui si esegue il join
  - il metodo di join usato (nested loop, merge join)
  - un piano indicante come ogni relazione deve essere acceduta (indice, scansione sequenziale)
  - indicazione sulla necessità di ordinamento delle relazioni prima dell'esecuzione del join

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Join**
- Un aspetto importante riguarda l'ordinamento delle tuple nelle relazioni
- si definisce **ordinamento interessante** un ordinamento su attributi che compaiono in clausole di GROUP BY e di ORDER BY, e sugli attributi di join
- tale nozione è importante perché un piano parziale non ottimale, ma che ritrova le tuple ordinate rispetto ad un ordinamento interessante, potrebbe poi portare ad un piano globale ottimale

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Join**
- L'albero di ricerca è costruito tramite iterazione sul numero di relazioni "joined"
- al primo passo, si determina il modo più efficiente per accedere ogni singola relazione per ogni ordinamento interessante e per il caso non ordinato
- al secondo passo, si determina il modo migliore per eseguire il join di ogni singola relazione con ogni altra singola relazione (alcune combinazioni non sono valutate in base all'euristica di ritardare l'esecuzione del prodotto Cartesiano)
- il secondo passo produce soluzioni per eseguire i join di coppie di relazioni

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Join**
- Al terzo passo si determina il modo migliore per eseguire i join di tre relazioni
- il terzo passo viene eseguito considerando ogni relazione composita, ottenuta al passo precedente, e determinando il modo migliore per collegarla con ogni altra relazione (sempre limitandosi a piani left-deep e utilizzando l'euristica del prodotto Cartesiano)
- si procede in questo modo fino a che tutte le relazioni sono state considerate

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Join**
- dopo che tutte le soluzioni complete sono determinate (il join di tutte le relazioni è eseguito), l'ottimizzatore sceglie la soluzione meno costosa che restituisce le tuple nell'ordine richiesto (se è stato specificato un ordine)
- se esiste una soluzione che produce le tuple secondo l'ordinamento richiesto, non deve essere eseguito alcun ordinamento extra per le clausole di ORDER BY e GROUP BY, a meno che la soluzione ordinata sia più costosa della soluzione più economica non ordinata con il costo addizionale dell'esecuzione dell'ordinamento

Ottimizzazione di interrogazioni
Enumerazione e scelta dei piani

- **Join - Esempio**
- Schema:
  Sailors(sid,sname,rating,age)
  Reserves(sid,bid,day,rname)
- indici:
  - su bid (B+) per Reserves
  - su rating (B+) e sid (hash) per Sailors
- Query

```
SELECT sname
FROM Reserves NATURAL JOIN Sailors
WHERE bid = 100 AND rating > 5
```

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Join - Esempio**
- Passo 1
- si considerano i metodi di accesso per Sailors: indice su sid, indice su rating, scansione sequenziale, tenendo conto della condizione di selezione rating > 5
- supponiamo che l'indice su rating sia la meno costosa, ritrova le tuple ordinate su rating
- poiché gli altri metodi non producono le tuple ordinate vengono scartati
- si considerano i metodi di accesso per Reserves: indice su bid e scansione sequenziale, tenendo conto della condizione di selezione bid = 100

Ottimizzazione di interrogazioni Enumerazione e scelta dei piani

- **Join - Esempio**
- Passo 2
- si considera il piano per Reserves e il suo join (come outer) con Sailors
- in questo caso si cercano tuple di Sailors con rating > 5 e sid = v, in questo caso poiché la condizione di uguaglianza è più selettiva, conviene accedere a Sailors attraverso l'indice su sid
- tutti i possibili metodi di join vengono considerati, compreso il merge join (aggiungendo un passo in cui le relazioni vengono ordinate su sid)
- lo stesso viene ripetuto per Sailors come outer e Reserves come inner

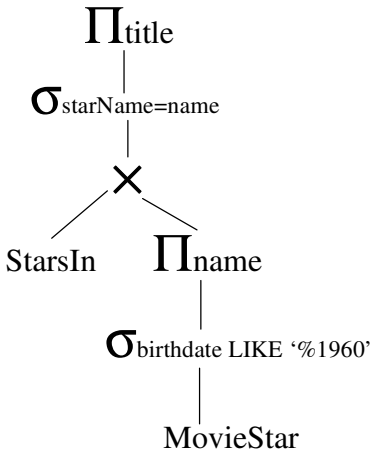
Ottimizzazione di interrogazioni
Sottointerrogazioni

- Se possibile le interrogazioni che contengono sottointerrogazioni vengono trasformate in interrogazioni equivalenti senza sottointerrogazioni
- esempio

```
SELECT title
FROM StarsIn
WHERE starName IN (
      SELECT name
      FROM MovieStar
      WHERE birthdate LIKE '%1960');
```

utilizzando rappresentazione intermedia con "two argument selection" viene rappresentata come

Ottimizzazione di interrogazioni
Sottointerrogazioni



Ottimizzazione di interrogazioni
Sottointerrogazioni

- Altrimenti (ad es. sottointerrogazioni scalari) l'idea è comunque che una sottointerrogazione viene valutata e il valore ottenuto dalla sua interrogazione sostituito nell'interrogazione principale
- le sottointerrogazioni correlate devono essere invece valutate in riferimento al valore della tupla candidata esaminata nell'interrogazione principale
- l'ottimizzatore ha molti meno margini di ottimizzazione (ad esempio si può usare nested loop ev. con indice, con la sottointerrogazione come inner, ma non altre strategie)

Ottimizzazione di interrogazioni
Sottointerrogazioni

- Non tutti gli ottimizzatori trattano in maniera sofisticata le sottointerrogazioni (e sono in grado di riconoscere quando una interrogazione può essere riscritta senza utilizzo di sottointerrogazioni)
- quindi, poiché molto spesso una query con sottointerrogazione è equivalente a una query senza sottointerrogazione, è preferibile utilizzare la seconda
- esempio:
  - vogliamo trovare i nomi dei sailor che hanno riservato la barca con bid 103
  - queste tre query sono equivalenti, ma in alcuni DBMS la query (a) può essere valutata più efficientemente della (b) che a sua volta può essere valutata più efficientemente della (c)

Ottimizzazione di interrogazioni
Sottointerrogazioni

```
(a)   SELECT sname
      FROM Sailors, Reserves
      WHERE Sailors.sid = Reserves.sid AND bid = 103
(b)   SELECT sname
      FROM Sailors
      WHERE sid IN (SELECT sid
                    FROM Reserves
                    WHERE bid = 103)
(c)   SELECT sname
      FROM Sailors S
      WHERE EXISTS  (SELECT *
                     FROM Reserves
                     WHERE sid = S.sid AND bid = 103)
```


Progettazione fisica e tuning

- Dopo la progettazione concettuale e logica dello schema con le metodologie viste, e la definizione del livello esterno (viste) abbiamo gli schemi *logico* e *esterno* della base di dati
- bisogna però ancora scegliere gli indici e decidere le strategie di memorizzazione delle relazioni
- inoltre gli schemi logici ed esterni ottenuti dalla progettazione possono dover essere rivisti per esigenze legate alle prestazioni

Progettazione fisica e tuning

- il primo passo per effettuare questa fase di progettazione è capire il *workload*:
  - quali sono le query più importanti e con che frequenza vengono eseguite
  - quali sono gli aggiornamenti più importanti e con che frequenza vengono eseguiti
  - le prestazioni desiderate/necessarie per tali interrogazioni e aggiornamenti

Progettazione fisica e tuning

- Per ogni query nel workload bisogna chiedersi:
  - a quali relazioni accede?
  - quali attributi ritrova?
  - quali attributi sono coinvolti in condizioni di selezione/join? quanto selettive saranno queste condizioni?
- Per ogni update nel workload bisogna chiedersi:
  - quali attributi sono coinvolti in condizioni di selezione/join? quanto selettive saranno queste condizioni?
  - il tipo di update (INSERT/DELETE/UPDATE) e gli attributi che saranno coinvolti nell'aggiornamento

Progettazione fisica e tuning

Decisioni da prendere

- quali indici creare
  - quali relazioni devono avere indici
  - su quali attributi
  - più di un indice per relazione
- per ogni indice, di che tipo deve essere
  - clusterizzato? hash/tree? denso/sparso?
- bisogna modificare lo schema?
  - considerare schemi normalizzati alternativi (nel decomporre in BCNF ci sono varie alternative)
  - `disfare' alcuni passi di decomposizione ed accontentarsi di una forma normale minore? (*denormalizzazione)*
  - partizionamento orizzontale, replicazione, viste ...

Progettazione fisica e tuning

Scelta degli indici

- Un possibile approccio è:
  - si considerano una alla volta le query più importanti
  - si considera il piano di esecuzione migliore utilizzando gli indici attualmente disponibili
  - se con un indice addizionale si avrebbe un piano di esecuzione migliore, lo si aggiunge
- prima di creare un indice, però, bisogna anche considerare l'impatto degli aggiornamenti sul workload!
- trade-off:
  - gli indici possono velocizzare le query ma rallentare gli aggiornamenti
  - inoltre richiedono spazio su disco

Progettazione fisica e tuning

Scelta degli indici

- gli attributi che appaiono in una clausola WHERE sono candidati come chiavi per un indice
  - se la condizione è di uguaglianza può essere meglio un indice hash
  - se la condizione è di tipo range si preferisce un indice ad albero
  - la clusterizzazione su tale attributo è particolarmente utile in caso di query tipo range, ma può essere utile anche per l'uguaglianza se ci sono duplicati
- bisogna cercare di scegliere indici che siano utilizzabili nel maggior numero possibili di query
- un solo indice per relazione può essere clusterizzato, per sceglierlo ci si basa sulle interrogazioni importanti che trarrebbero maggior vantaggio dalla clusterizzazione

Progettazione fisica e tuning

Scelta degli indici

- se la clausola WHERE di un'interrogazione importante contiene diverse condizioni si considerano anche indici con chiave multi-attributo
  - se si hanno selezioni di tipo range, l'ordine degli attributi deve essere scelto attentamente in modo da corrispondere a quello del range
  - si noti anche che questo tipo di indici a volte permettono strategie index-only per query importanti
    - si noti anche che per le strategie index-only, il clustering non è importante!

Progettazione fisica e tuning

Scelta degli indici

- Quando si considera una condizione di join:
  - un indice hash sulla relazione inner è molto utile per la strategia Index Nested Loop
    - dovrebbe essere clusterizzato se l'attributo di join non è una chiave per la relazione inner e si devono ritrovare le tuple inner (cioè non basta index-only scan)
  - *un indice clusterizzato di tipo* B+ tree sull'attributo di join è utile per la strategia sort-merge

Progettazione fisica e tuning

Esempio

```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  D.dname='Toy' AND E.dno=D.dno
```

- un indice hash su *D.dname* permette la selezione di 'Toy'
  - non serve quindi un indice su D.dno
- un indice hash su *E.dno* permette di ritrovare le tuple (inner) di Emp che matchano per ogni tupla (outer) di Dept selezionata
- se la clausola WHERE includesse: `` ... AND E.age=25'' ?
  - si potrebbero ritrovare le tuple di Emp usando l'indice su *E.age*, e poi effettuare il join cercando le tuple di Dept che soddisfano la condizione su *dname* (analogo alla strategia utilizzando l'indice su *E.dno)*
  - se si ha già un indice su *E.age*, questa query non motiva l'aggiunta di un indice su *E.dno*

Progettazione fisica e tuning

Esempio

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE E.sal BETWEEN 10000 AND 20000
  AND E.hobby='Stamps' AND E.dno=D.dno
```

- Emp dovrebbe essere la relazione outer
  - potrebbe essere utile un indice hash su *D.dno*
- che indici si dovrebbero costruire su Emp?
  - si potrebbe usare un indice di tipo B+ tree su *E.sal* O un indice su *E.hobby*
  - uno solo di tali indici è necessario e qual è meglio dipende dalla selettività delle condizioni
    - in generale (ma non è detto!), le selezioni con uguaglianza sono più selettive di quelle di tipo range
- si noti come la scelta degli indici sia guidata dai piani che ci aspettiamo che l'ottimizzatore considerà per la query
- bisogna avere chiaro come funzionano gli ottimizzatori!

Progettazione fisica e tuning

Clustering

- si può usare un indice di tipo B+ tree su E.age
  - quanto selettiva è la condizione?
  - l'indice è clusterizzato?

```
SELECT E.dno
FROM Emp E
WHERE E.age>40
```

- interrogazione con GROUP BY
  - se molte tuple hanno *E.age* > 10, l'uso dell'indice su *E.age* seguito dall'ordinamento delle tuple ritrovate può essere costoso
  - un indice clusterizzato su *E.dno* può essere meglio

```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age>10
GROUP BY E.dno
```

- query di uguaglianza con duplicati:
  - è vantaggioso clusterizzare su *E.hobby*

```
SELECT E.dno
FROM Emp E
WHERE E.hobby=Stamps
```

Progettazione fisica e tuning

Clustering

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname='Toy' AND E.dno=D.dno
```

- Il clustering è molto importante quando si accedono le tuple inner nell'index nested loop
  - l'indice su *E.dno* dovrebbe essere clusterizzato
- se la clausola WHERE fosse

  WHERE E.hobby='Stamps' AND E.dno=D.dno
  - se molti impiegati collezionano francobolli, potrebbe essere più efficiente il Sort-Merge join
  - un indice *clusterizzato* su D.dno sarebbe utile
- *conclusione*: la clusterizzazione è utile quando si devono ritrovare più tuple

Progettazione fisica e tuning

Indici multi-attributo

- per ritrovare record di Emp con *age*=30 AND *sal*=4000, un indice su <*age,sal*> è meglio di un indice su *age* o un indice su *sal*
  - tali indici sono anche detti indici *compositi* o *concatenati*
  - la scelta della chiave dell'indice è ortogonale al clustering etc.
- se la condizione è: 20<*age*<30 AND 3000<*sal*<5000:
  - indici ad albero clusterizzati su <*age,sal*> o <*sal,age*> sono la soluzione migliore
- se la condizione è: *age*=30 AND 3000<*sal*<5000:
  - un indice clusterizzato su <*age,sal*> è molto meglio di un indice <*sal,age*>
- gli indici compositi sono però più grossi e vengono aggiornati più spesso

Progettazione fisica e tuning

Piani Index-Only

- un certo numero di query può essere risolto senza ritrovare tuple da una o più delle relazioni coinvolte se è disponibile un indice opportuno

<*E.dno*>

```
SELECT D.mgr
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<*E.dno*>

```
SELECT E.dno, COUNT(*)
FROM Emp E
GROUP BY E.dno
```

<*E.dno,E.sal*> (ad albero)

```
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno
```

<*E. age,E.sal*> o <*E.sal, E.age*> (ad albero)

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
 E.sal BETWEEN 3000 AND 5000
```


Progettazione fisica e tuning

- La progettazione di una base di dati consiste in molte fasi: *analisi dei requisiti, progettazione logica e fisica, raffinamento dello schema, progettazione fisica e tuning*
  - in generale, queste fasi devono essere iterate più volte per raffinare la progettazione di una base di dati e decisioni prese in una fase possono influenzare scelte in un'altra fase
- per sviluppare una buona progettazione bisogna capire la natura del *workload* dell'applicazione e i requisiti prestazionali
  - quali sono le query e gli aggiornamenti più frequenti/importanti? quali attributi/relazioni coinvolgono?

Progettazione fisica e tuning

- Gli indici devono essere scelti in modo tale da velocizzare qualche query importante (e eventualmente qualche aggiornamento)
  - il mantenimento di un indice causa un overhead sugli aggiornamenti all'attributo chiave
  - si dovrebbero scegliere indici che possono essere utili in più query, se possibile
  - gli indici che consentono strategie index-only sono particolarmente utili
  - la decisione sul clustering è molto importante: solo un indice per relazione può essere clusterizzato
  - l'ordine dei campi negli indici compositi può essere importante

Progettazione fisica e tuning

Tuning dello schema logico

- anche la scelta dello schema logico dovrebbe essere guidata dal workload, oltre che da aspetti legati alla ridondanza:
  - si può decidere di scegliere uno schema in 3NF piuttosto che in BCNF
  - il workload può influenzare le scelte fatte nello scomporre una relazione in 3NF o BCNF
  - si può scomporre ulteriormente uno schema BCNF
  - si può *denormalizzare* (cioè disfare un passo di normalizzazione), o si possono aggiungere campi ad una relazione
  - si può effettuare una *scomposizione orizzontale*
- se queste modifiche vengono fatte dopo che la base di dati è stata popolata si parla di *evoluzione di schema*

Progettazione fisica e tuning

Tuning delle interrogazioni

- se un'interrogazione è più lenta di quel che ci si aspetta bisogna controllare se le statistiche sono troppo vecchie
- a volte più succedere che il DBMS non stia eseguendo il piano che si pensa - tipicamente questo succede per:
  - selezioni che coinvolgono valori nulli
  - selezioni che coinvolgono espressioni aritmetiche o su stringhe
  - selezioni che coinvolgono condizioni in OR
  - mancanza di caratteristiche quali strategie index-only o alcuni metodi di join o stima inesatta delle dimensioni
- si deve controllare quale piano viene effettivamente utilizzato e aggiustare la scelta degli indici o riscrivere l'interrogazione

Riferimenti

- Raghu Ramakrishnan, Johannes Gehrke "Database Management Systems - Second Edition" McGraw Hill, 2000
- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom "Database System Implementation" Prentice Hall, 2000