



Ottimizzazione delle interrogazioni

Prof. Matteo Golfarelli
Alma Mater Studiorum - Università di Bologna

Per approfondimenti:

- Ciaccia, Maio. Lezioni di basi di dati: pp 377-434
- Sistemi di basi di dati-Complementi R.A. Elmasri and S.B. Navathe
- ORACLE 11g – Performance Tuning Guide

Introduzione

- L'ottimizzazione delle performance (*tuning*) è l'insieme delle attività che mirano a massimizzare le prestazioni di un sistema in relazione agli obiettivi preposti e nel rispetto dei vincoli di sistema.
- Gli obiettivi da perseguire variano a seconda del tipo di sistema considerato:
 - **Sistemi OLTP:** massimizzazione del throughput (quantità del lavoro svolto in un'unità di tempo).
 - **Sistemi OLAP:** minimizzazione del tempo di risposta all'utente (response time).

Introduzione

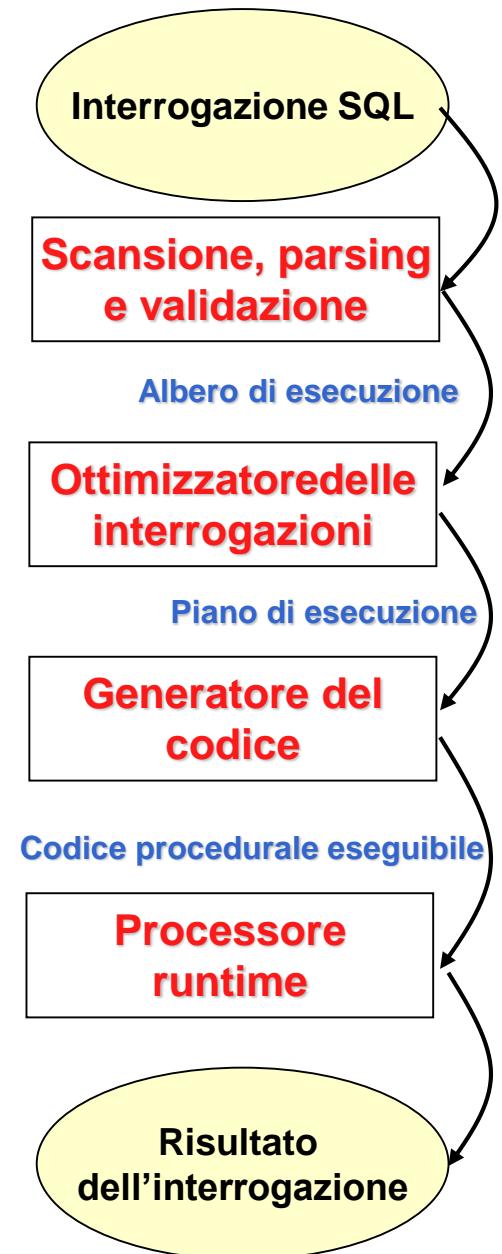
- L'ottimizzazione delle prestazioni è un fattore che deve essere preso in considerazione in tutte le fasi della progettazione e dell'implementazione:
 - Progettisti delle basi di dati attraverso una corretta modellazione.
 - Progettisti delle applicazioni stabilendo il giusto compromesso tra complessità ed efficienza delle funzioni.
 - Implementatori delle applicazioni attraverso un utilizzo attento delle risorse di sistema.
 - **Database Administrator** che devono continuamente verificare le mutate necessità degli utenti e identificare colli di bottiglia non individuabili durante la realizzazione.

I compiti dell'amministratore

- Per poter svolgere il proprio compito l'amministratore deve conoscere al meglio le modalità di funzionamento del DBMS
 - *Quali indici ?*
 - *Quali piani di accesso ?*
 - *Di quante risorse necessita una certa operazione ?*
 - *Su quali componenti si può intervenire ?*
- Il principale componente da studiare per poter rispondere a queste domande è l'**ottimizzatore**.

Fasi dell'esecuzione

- Il Parser esegue l'analisi lessicale e sintattica dell'interrogazione (**parsing**) e di convalida degli elementi referenziati (**check semantico**)
- Il parser produce una prima rappresentazione interna non dichiarativa chiamata: **albero di esecuzione**
- L'ottimizzatore trasforma l'albero di esecuzione al fine di ottimizzare le performance del sistema
- L'ottimizzatore sceglie quali algoritmi utilizzare per implementare gli operatori di algebra relazionale presenti nell'albero di esecuzione
- Il generatore del codice traduce l'albero di esecuzione creato in codice procedurale vero e proprio



Check semantico: un esempio

- La fase di check semantico si basa sulle informazioni presenti nel Data Dictionary
- Supponendo di avere la seguente query:

```
SELECT P_Name from Part;
```

- In fase di check semantico verranno eseguite interrogazioni del tipo:

```
SELECT * FROM DBA_TABLES WHERE  
TABLE_NAME = 'PART' AND OWNER = 'USERSI'
```

```
SELECT * FROM DBA_TAB_COLUMNS WHERE  
TABLE_NAME = 'PART' AND COLUMN_NAME='P_Name' AND OWNER = 'USERSI'
```

Traduzione dell'interrogazione

- SQL è un linguaggio dichiarativo ossia specifica **cosa** deve essere recuperato ma non **come** ciò deve essere fatto.
- Il **come** è un compito che viene demandato al DBMS che traduce l'interrogazione in un equivalente espressione in *algebra relazionale estesa* che verrà in seguito ottimizzata
- L'algebra relazionale deve essere estesa per elaborare interrogazioni che prevedano operazioni non previste nell'algebra "standard": aggregazione, ordinamento, ecc.
- Le interrogazioni SQL vengono divise in blocchi che formano le unità base traducibili in operatori algebrici e ottimizzabili
- Un blocco di un'interrogazione contiene una singola espressione SELECT-FROM-WHERE-GROUP BY-HAVING

Uno DB di esempio

IMP(I_SSN, I_Nome, I_Cognome, I_DataN, I_Indirizzo, I_Sesso,
I_Stipendio, I_Super:IMP, I_ND:DIP)

DIP(D_Numero, D_Nome, D_SSNDir:IMP, D_DataInizio)

SEDI DIP(S_Numerodip:DIP, S_Sede)

PROG(P_Numero, P_Nome, P_Sede, P_Dipartimento:DIP)

LAVORA SU(L_SSNImp:IMP, L_Prog:PROG)

PERSONA A CARICO(C_SSN:IMP, C_NomeP, C_Sesso, C_DataN, C_Parentela)

Traduzione dell'interrogazione

$$\pi_{I_cognome, I_nome} (\sigma_{I_Stipendio > c} (IMP))$$

```
select I_Cognome, I_Nome  
from IMP  
where I_Stipendio > (select MAX(I_Stipendio)  
from IMP  
where I_ND = 5);
```

$$c = \sum_{\text{MAX}(I_Stipendio)} (\sigma_{I_ND=5} (IMP))$$

- Il blocco interno, corrispondente all'interrogazione nidificata richiede un'operazione dell'algebra estesa
- L'ottimizzatore sceglierà un piano di esecuzione per ogni blocco
- L'ottimizzazione delle interrogazioni nidificate correlate risulta ovviamente più complesso

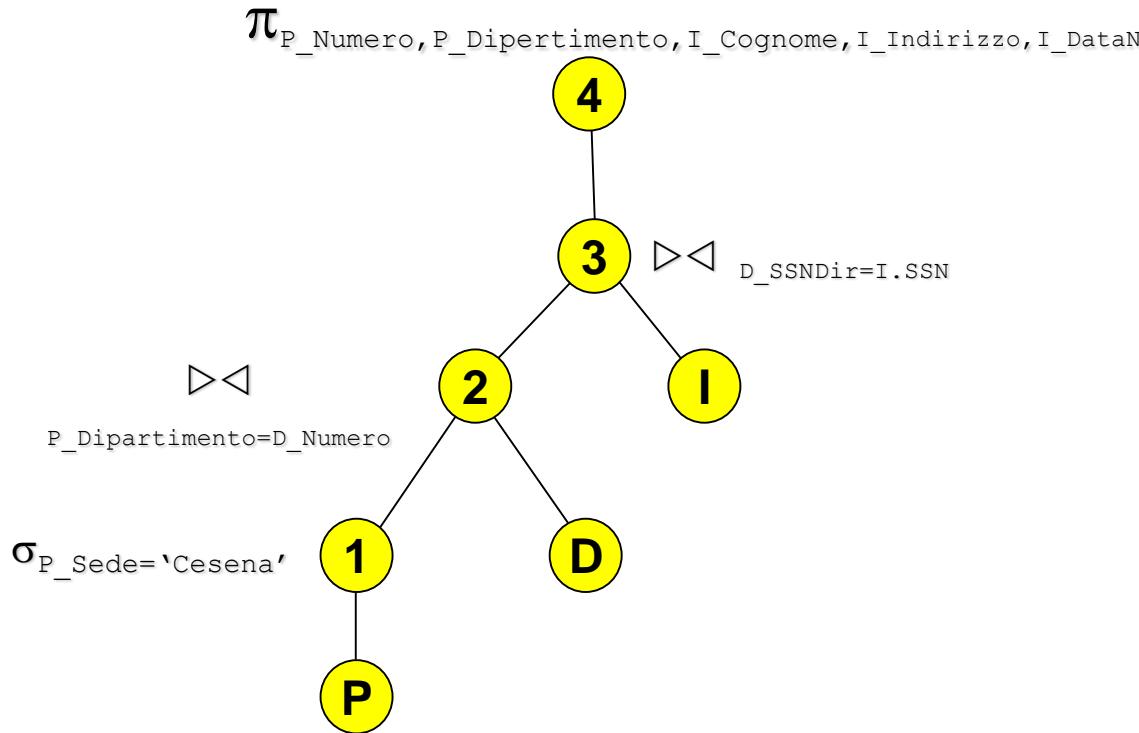
L'albero di esecuzione

- Un albero di esecuzione è una struttura ad albero che corrisponde a una espressione dell'algebra relazionale estesa.
 - I suoi nodi foglia sono le relazioni di input dell'interrogazione
 - I suoi nodi interni rappresentano le operazioni dell'algebra relazionale estesa

```
select P_Numer, P_Dipartimento, I_Cognome, I_Indirizzo, I_DataN  
from PROG, DIP, IMP  
where P_Dipartimento=D_Numer AND D_SSNDir=I.SSN AND  
P_Sede='Cesena'
```

$$\pi_{P_Numer, P_Dipertimento, I_Cognome, I_Indirizzo, I_DataN}((\sigma_{P_Sede='Cesena'}(PROG) \bowtie_{P_Dipartimento=D_Numer} (DIP)) \bowtie_{D_SSNDir=I.SSN} (IMP))$$

L'albero di esecuzione



- L'albero di esecuzione rappresenta uno specifico ordine delle operazioni
 - Il nodo (1) deve essere disponibile per iniziare l'operazione (2)
 - Il nodo (2) deve essere disponibile per iniziare l'operazione (3)

L'albero di esecuzione

- L'albero di esecuzione così specificato non è ancora direttamente eseguibile da un processore run-time poiché le singole operazioni specificate nei suoi nodi possono essere eseguite in molti modi diversi
 - Accesso a *tabella*: sequenziale, con indice
 - *Join*: nested loop, sort merge, hash join
 - ...
- La comprensione di un albero di esecuzione richiede la conoscenza delle operazioni che possono essere contenute nei suoi nodi

Gli indici

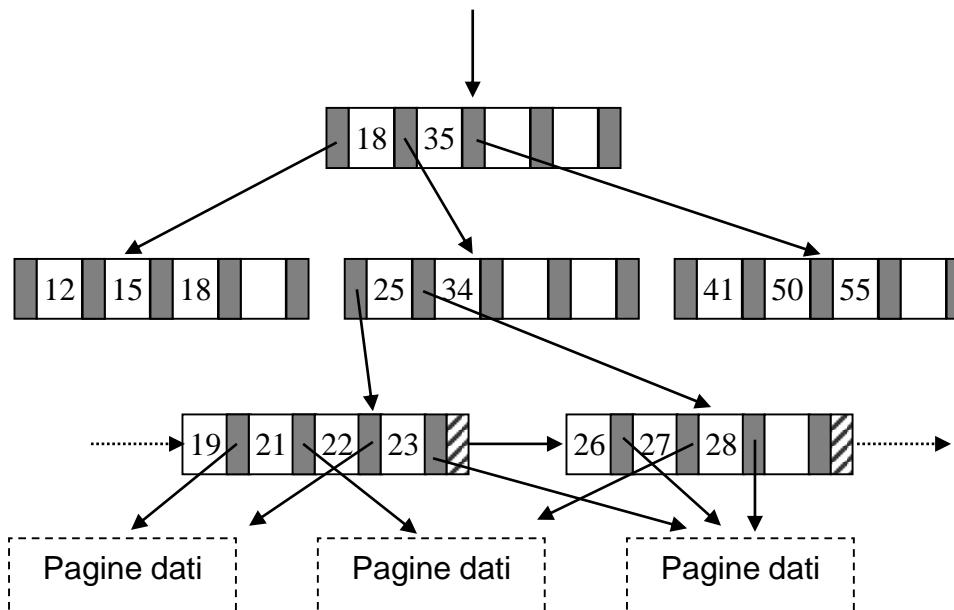
- Un indice è una **struttura dati accessoria** per accedere velocemente ai dati delle relazioni sulla base di una chiave di ricerca
- L'idea di base è quella di associare ai dati una “tabella” nella quale la tupla i -esima memorizza una coppia del tipo (k_i, p_i) dove:
 - k_i è un valore della chiave di ricerca su cui l'indice è costruito
 - p_i è un riferimento al record con valore di chiave k_i
- I DBMS dispongono di **più tipologie di indici** ognuna adatta a un particolare tipo di dato o a un particolare tipo di accesso
 - Indici B+Tree
 - Indici Bitmap
 - Indici di join

Gli indici

- Più indici possono essere usati contemporaneamente nell'ambito della stessa query
- La scelta di **quali e quanti indici utilizzare** è (normalmente) demandata all'ottimizzatore ed è trasparente agli utilizzatori
- La scelta di **quali e quanti indici costruire** è compito del progettista e dell'amministratore del sistema

I B⁺Tree

- Un indice **B⁺-Tree** per un attributo c di una relazione R è un albero bilanciato che consente accessi associativi alle tuple di R in base ai valori della chiave c . Le foglie dell'albero, collegate tra loro in sequenza, contengono i puntatori ai record (*RID - Row Identifier*), mentre i nodi interni costituiscono una “mappa” per consentire una rapida localizzazione delle chiavi.



I B⁺Tree

- I B⁺Tree possono essere:
 - **Primary index:** un solo record per ogni valore di chiave
 - **Secondary index:** più record per ogni valore di chiave. In quest'ultimo caso le foglie contengono la lista di RID relative alle tuple che presentano un particolare valore di chiave.
- Per ricercare un particolare valore v
 - si discende l'albero in base alla mappa dei puntatori;
 - tramite la foglia individuata si accede al blocco dati in cui sono memorizzati i record che presentano il valore di chiave v .
- Per eseguire una ricerca su un intervallo di valori $[a, b]$ è sufficiente scendere nell'albero utilizzando a come valore di chiave, quindi seguire la sequenza di nodi foglia fino a che non si incontra un valore superiore a b .

I B⁺Tree

- Ogni nodo di un B⁺Tree può ospitare un numero variabile di separatori (o chiavi) compreso tra g (grado) e $2 \cdot g$ a seconda della sua dimensione D , e della lunghezza dei puntatori $\text{len}(p)$ e dei separatori stessi $\text{len}(k)$:

$$2g \times \text{len}(k) + (2g+1) \times \text{len}(p) \leq D \rightarrow g = \left\lfloor \frac{D - \text{len}(p)}{2(\text{len}(k) + \text{len}(p))} \right\rfloor$$

- La dimensione di un B⁺Tree dipende da molteplici fattori (es. lunghezza dei separatori, lunghezza delle RID, dimensione della pagina di disco) e può variare a causa delle operazioni di bilanciamento. Una stima approssimativa della dimensione può essere fornita dal numero delle foglie dell'albero che è pari a:

$$NL = \left\lceil \frac{NK \cdot \text{len}(k) + NR \cdot \text{len}(p)}{D \cdot u} \right\rceil$$

- dove NK è il numero di valori distinti di chiave, NR è il numero delle tuple da indicizzare, $\text{len}(k)$ e $\text{len}(p)$ rappresentano rispettivamente la lunghezza delle chiavi delle tuple e dei puntatori ai blocchi dati, infine u rappresenta il fattore di riempimento dei nodi.

I B⁺Tree

- Il B⁺-Tree è una struttura bilanciata, ossia garantisce che l'altezza h dell'albero sia sempre costante per tutti i percorsi dalla radice alle foglie.
- A tal fine vengono utilizzate delle procedure di bilanciamento che si innescano a fronte di cancellazioni (**catenation, underflow**) e inserimenti (**split**).
- A parità del numero di foglie NL , l'altezza di un B⁺-Tree può variare in base al livello di riempimento dei nodi:

$$1 + \lceil \log_{2g+1} NL \rceil \leq h \leq 2 + \left\lfloor \log_{g+1} \frac{NL}{2} \right\rfloor$$

Tutti i livelli prima delle foglie pieni
 $(2g+1)^{h-1} \geq NL$

La radice ha 2 soli puntatori e
ogni livello intermedio $g+1$
 $2 \times (g+1)^{h-2} \leq NL$

Gli indici bitmap

- Un indice bitmap su un attributo è composto da una matrice di bit contenente:
 - Tante righe quante sono le tuple della relazione
 - Tante colonne quanti sono i valori distinti di chiave dell'attributo
- Il bitmap (i,j) è posto a TRUE se nella tupla i -esima è presente il valore j -esimo

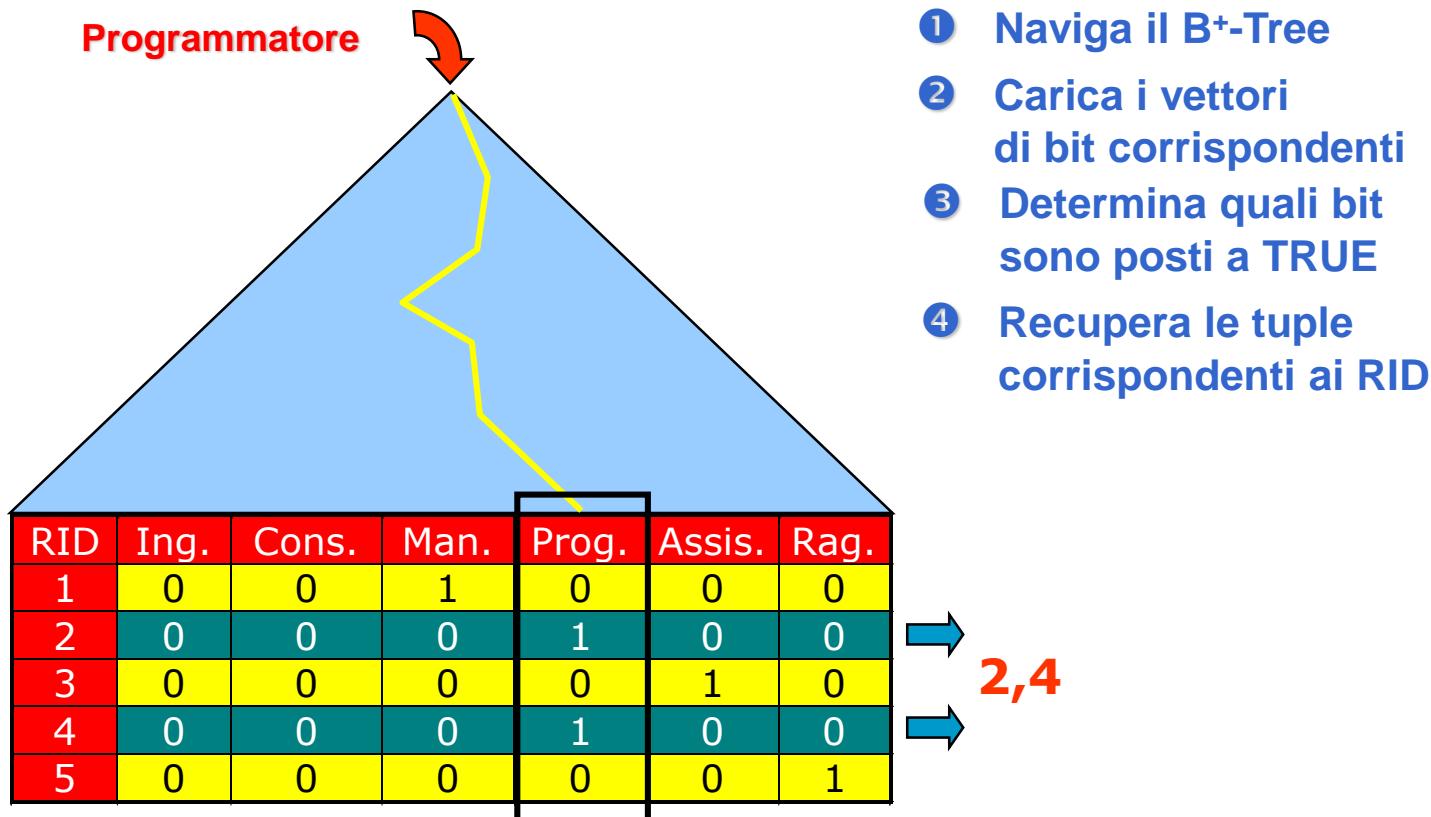
Esempio: Indice sul campo Posizione della tabella impiegati
Ingegnere – Consulente – Manager – Programmatore
Assistente – Ragioniere

**L'impiegato
corrispondente al RID 1 è
un Manager**

RID	Ing.	Cons.	Man.	Prog.	Assis.	Rag.
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	1	0
4	0	0	0	1	0	0
5	0	0	0	0	0	1

Implementazione dei bitmap

- Normalmente i bitmap sono associati a B+-Tree le cui foglie contengono vettori di bit invece di RID

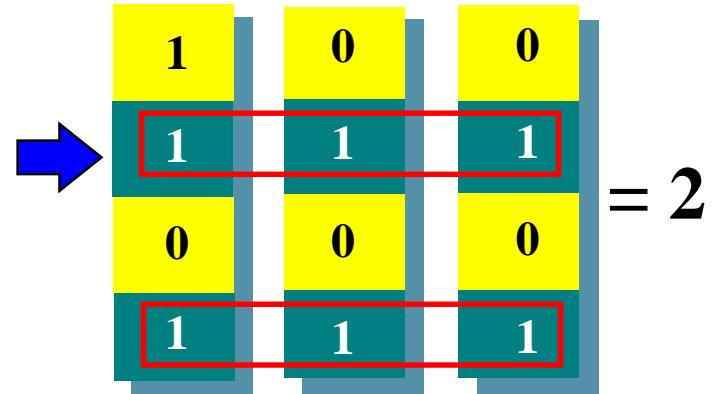


I vantaggi degli indici bitmap

- Lo spazio richiesto su disco può essere molto ridotto
- I/O è molto basso poiché vengono letti solo i vettori di bit necessari
- Ottimi per interrogazioni che non richiedono l'accesso ai dati
- Permettono l'utilizzo di operatori binari per l'elaborazione dei predicati

Esempio: “*Quanti maschi in Romagna sono assicurati ?*”

RID	Sesso	Assic.	Regione
1	M	No	LO
2	M	Sì	E/R
3	F	No	LA
4	M	Sì	E/R



Occupazione su disco

- Gli indici bitmap sono adatti ad attributi con una ridotta cardinalità poiché ogni nuovo valore distinto di chiave richiede un ulteriore vettore di bit
- All'aumentare del numero di chiavi distinte aumenta la sparsità della matrice

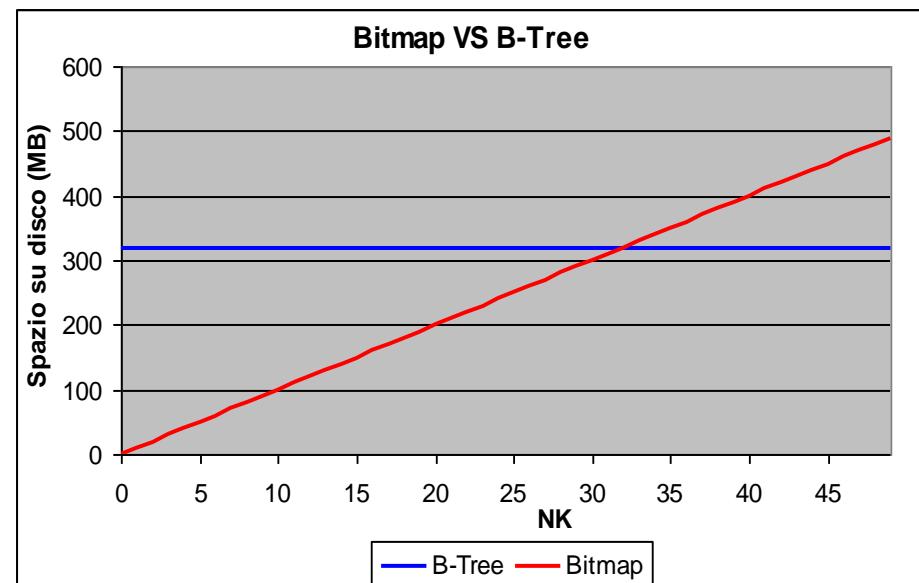
Esempio:

NR=10.000.000

Len(Pointer)= Len(Key) = 4×8 bit

B-tree $NK \times \text{Len(Key)} + NR \times \text{Len(Pointer)}$

Bitmap $NK \times \text{Len(Key)} + NR \times NK / 8$



La compressione delle matrici riduce il fattore di crescita della dimensione

Algoritmi di sort

- Gli algoritmi di ordinamento sono tra i più utilizzati nell'ambito delle operazioni SQL.
- Non vengono utilizzati solo in presenza della clausola ORDER BY ma anche per realizzare operazioni di JOIN, PROIEZIONE (con opzione distinct), GROUP BY
- L'ordinamento può essere evitato quando esiste un indice tale da permettere l'accesso ordinato alle tuple
- Viste le dimensione delle tabelle in gioco, nell'ambito dei DB si utilizzano tecniche di **ordinamento esterno**, ossia che non richiedono di mantenere tutti i dati contemporaneamente nella memoria centrale
- L'algoritmo più utilizzato è il sort-merge

Sort-Merge

- L'algoritmo **ordina** dapprima porzioni di file (**run**) in grado di essere caricate singolarmente in memoria centrale, ad esempio con un algoritmo di QuickSort.
- Le run ordinate e salvate su disco vengono poi **unite** progressivamente, **Z alla volta**, creando porzioni via via più grandi
- Le prestazioni dell'algoritmo dipendono fortemente dalla dimensione del buffer della memoria centrale dove vengono effettivamente eseguite le operazioni di ordinamento e di fusione.

Sort-Merge

- Per semplicità consideriamo il caso base a $Z = 2$ vie, e supponiamo di avere a disposizione solo $NB = 3$ buffer in memoria centrale

g	24
a	19
d	21
c	33
b	14
e	16
r	16
d	31
m	3
p	2
d	7
a	14

Input



Sort interno

a	19
d	21
g	24

b	14
c	33
e	16

d	31
m	3
r	16

a	14
d	7
p	2

1-page runs

a	19
b	14
c	33

d	21
e	16
g	24

a	14
b	14
c	33

d	7
d	21
d	31

a	14
d	7
p	2

2-page runs

3 record per pagina

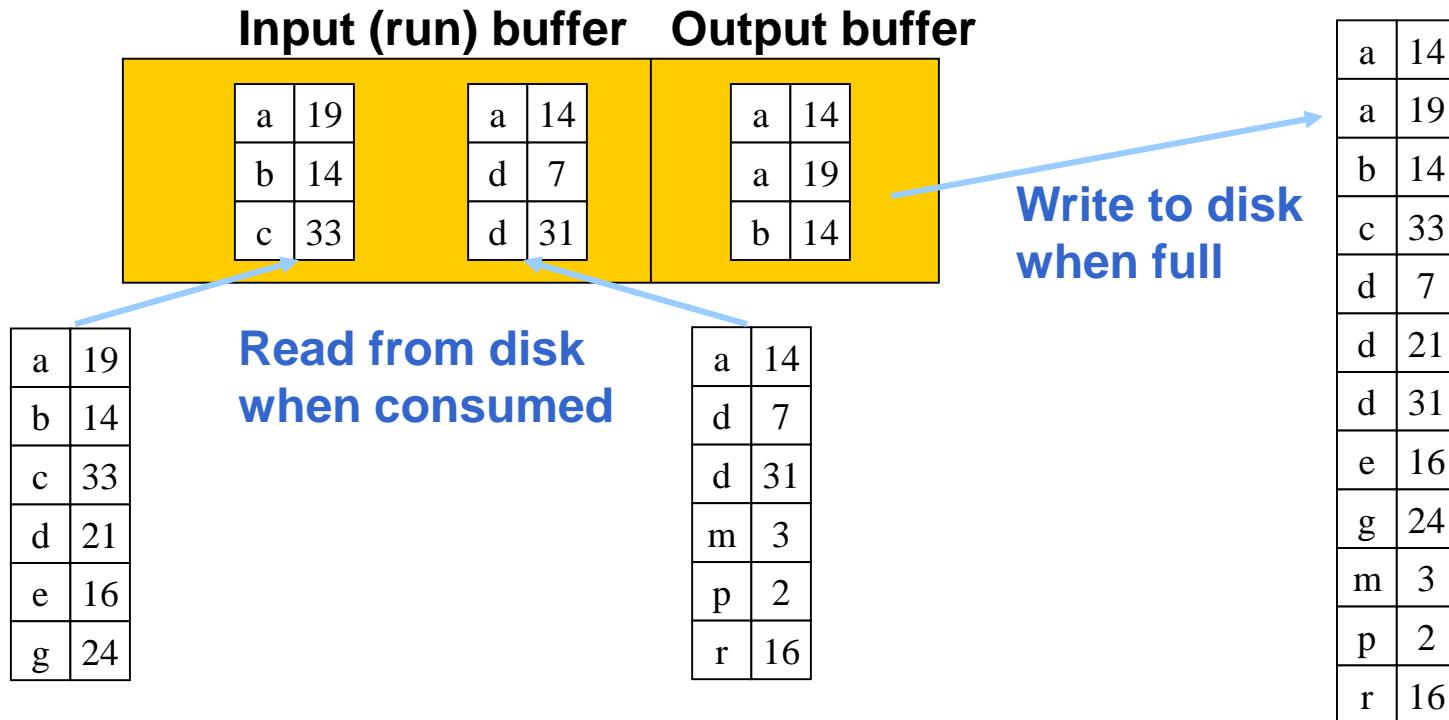
a	14
a	19
b	14
c	33
d	7

Merge

Output

Sort-Merge

- ❑ Nel caso base $Z = 2$ si fondono 2 run alla volta
- ❑ Con $NB = 3$, si associa un buffer a ognuna delle run, il terzo buffer serve per produrre l'output, 1 pagina alla volta
- ❑ Si legge la prima pagina da ciascuna run e si può quindi determinare la prima pagina dell'output; quando tutti i record di una pagina di run sono stati consumati, si legge un'altra pagina della run



Sort-Merge

- Le prestazioni dell'algoritmo dipendono fortemente dalla dimensione del buffer della memoria centrale dove vengono effettivamente eseguite le operazioni di ordinamento e di fusione.
- Nel caso base $Z = 2$ e $NB = 3$ si può osservare che:
 - Nella fase di sort interno si leggono e si riscrivono NP pagine
 - Ad ogni passo di merge si leggono e si riscrivono NP pagine
 - Il numero di passi fusione è pari a $\lceil \log_2 NP \rceil$, in quanto ad ogni passo il numero di run si dimezza
- Il costo complessivo è pertanto pari a:
$$2 * NP + 2 * NP * \lceil \log_2 NP \rceil = 2 * NP * (\lceil \log_2 NP \rceil + 1)$$

Esempio: per ordinare $NP = 8.000$ pagine sono necessarie 224.000 operazioni di I/O; se ogni I/O richiede 20 msec, il sort richiede 4.480 secondi, ovvero circa 1h 15 minuti!

Sort-Merge

- Una prima osservazione è che nel passo di sort interno, avendo a disposizione $NB > 3$ buffer, si può utilizzare il buffer per portare a $FS > 1$ la lunghezza di ciascuna run interna, il che porta a un costo di:

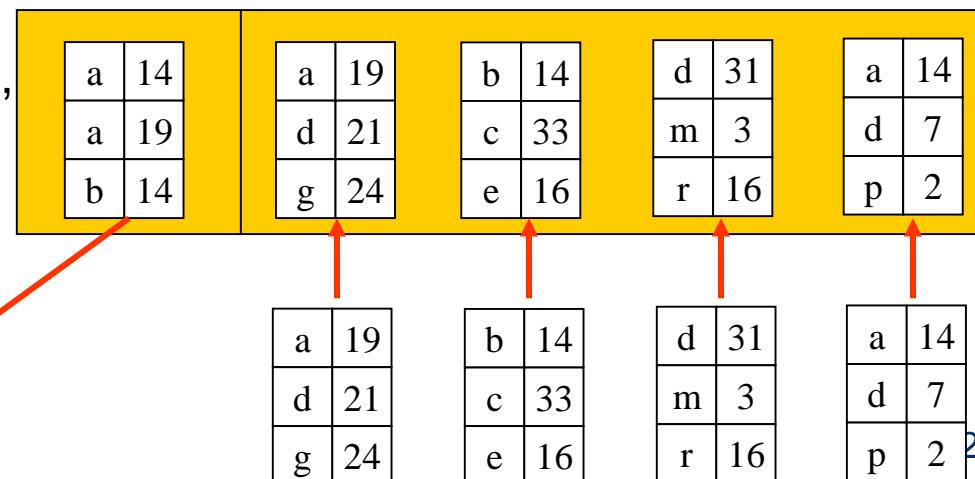
$$2 * NP * (\lceil \lceil \log_2 \lceil NP/FS \rceil \rceil + 1)$$

Esempio: con $NP = 8000$ pagine e $NB = 11$ si può portare $FS=5$ e quindi 192.000 I/O pari a 1 ora e 4 minuti

- Miglioramenti sostanziali si possono ottenere se, avendo $NB > 3$ buffer a disposizione, si fondono $Z = NB - 1$ run alla volta (1 buffer è per l'output)
- In questo caso il numero di passi di fusione è logaritmico in $NB - 1$, il che determina un costo pari a: $2 * NP * (\lceil \log_{NB-1} \lceil NP/NB \rceil \rceil + 1)$

Esempio:

con $NP = 8.000$ pagine
e $NB = 11$ si hanno 64.000 I/O,
per un tempo stimato
pari a 1.280 sec ≈ 21 minuti



Sort-Merge in presenza di selezioni

- Nel caso sulla tabella da ordinare insista un criterio di selezione locale, il numero di tuple da ordinare, EP, sarà ridotto in base al fattore di selettività del predicato Sel(Pred):

$$EP = \lceil NP * Sel(Pred) \rceil$$

$$(NP + EP) + 2 \times EP \times \lceil \log_{NB-1} \lceil EP/NB \rceil \rceil$$

Una curiosità: un tipico DBMS ordina 1 milione di record da 100 byte in 15 minuti. Il record è di 3.5 secondi con un sistema SGI, 12 CPU, 96 dischi e 2 GB di ram

Accesso a tavelle

- Per selezionare i record di una tabella (o file) si possono utilizzare vari algoritmi di ricerca detti algoritmi di scansione (*table scan*). Se l'algoritmo di ricerca prevede l'utilizzo di un indice si parla di scansione basata su indice (*index scan*)
- **Ricerca lineare o scansione sequenziale:** legge ogni tupla della tabella e verifica se soddisfa la condizione di selezione. Si applica se la tabella è disordinata rispetto al campo su cui deve essere fatta la selezione
 - In caso di esistenza del record si accede
 - in media a $(NP+1)/2$ blocchi
 - nel caso peggiore a NP blocchi
 - In caso di non esistenza del record si accede a NP blocchi

Accesso a tavelle

- **Ricerca binaria:** è applicabile se il file è ordinato rispetto al campo su cui deve essere fatta la selezione
 - In caso di esistenza del record si accede
 - in media a $\lfloor \log_2 NP \rfloor$ blocchi
 - nel caso peggiore a $\lfloor \log_2 NP \rfloor + 1$ blocchi
 - In caso di non esistenza del record si accede a $\lfloor \log_2 NP \rfloor + 1$ blocchi
- **Ricerca con indice:** la sequenza di operazioni da svolgere per recuperare i record corrispondenti a un valore di chiave è la seguente
 1. Si discende l'indice fino alle foglie
 2. Si leggono le foglie relative ai valori di interesse
 3. Si accede alle pagine dati indicate dai RID presenti nelle foglie

Accesso a tavelle

- Il costo di esecuzione nel caso di accesso con indice varia a seconda dell'ordinamento della tabella
 - h : altezza dell'indice
 - $\text{sel}(\text{pred})$: selettività dell'eventuale predicato di selezione sull'attributo su cui è costruito l'indice
 - NK : numero di valori distinti della chiave
 - EK : numero di valori di chiave che verificano il predicato di selezione
 - NL : numero di foglie dell'indice
 - NP : numero di pagine di disco della tabella
 - NT : numero di tuple della tabella
- **Indice clustered:** si ha quando le tuple della tabella sono ordinate rispetto all'attributo dell'indice
$$h-1 + \lceil \text{sel}(\text{pred}) \cdot NL \rceil + \text{sel}(\text{pred}) \cdot NP$$
- **Indice Unclustered:** si ha quando le tuple della tabella non sono ordinate rispetto all'attributo dell'indice
$$h-1 + \lceil \text{sel}(\text{pred}) \cdot NL \rceil + EK \cdot \Phi(NT / NK, NP)$$

Accesso a tavelle

- Si ricorda che la formula di Cardenas:

$$\Phi(ER, NP) = NP \cdot (1 - (1 - 1/NP)^{ER}) \leq \min \{ER, NP\}$$

fornisce una stima del numero di pagine, su un totale di NP, che contengono almeno uno degli ER record da reperire

- Si noti che le formule precedenti si applicano sia a indici B+-Tree sia a indici bitmap vista la modalità di memorizzazione di questi ultimi.
 - Ovviamente cambia la modalità di calcolo del numero delle foglie

Tecniche di join: nested loop

- ❑ È l'algoritmo di join più semplice e deriva direttamente dalla definizione dell'operazione.
- ❑ Una delle due relazioni coinvolte è designata come *esterna* e l'altra come *interna*. Supponendo di operare utilizzando due relazioni R e S , R esterna e S interna, e di essere in presenza di due predicati locali F_R su R e F_S su S , l'algoritmo procede ricercando, per ogni tupla t_R della relazione esterna che soddisfa F_R , tutte le tuple di S che soddisfano F_S e che soddisfano il predicato di join F_J .
- ❑ Il costo di esecuzione dell'algoritmo, espresso in numero di pagine di disco lette, è pari a:

$$\text{costo}(R) + ET_R \times \text{costo}(S)$$

dove $\text{costo}(R)$ e $\text{costo}(S)$ misurano i costi di accesso alle relazioni, ET_R è il numero atteso di tuple di R che soddisfano il predicato locale F_R .

Tecniche di join: nested loop

```
open R;
while not EOF(R)
{ CurrR= read(R);
  if (FR(CurrR))           // CurrR verifica il predicato locale
    { open S;
      while not EOF(S)
        { CurrS= read(S);
          if (FS(CurrS) and FJ(CurrR,CurrS)) then
            restituisci CurrR + CurrS;
        }
      next;
      close S;
    }
  next;
}
close R;
```

- ❑ Nel caso più semplice in cui non sono presenti predicati locali e si accede alle relazioni mediante scan sequenziali il costo di esecuzione sarà quindi:

$$\mathbf{NP_R + NT_R \times NP_S}$$

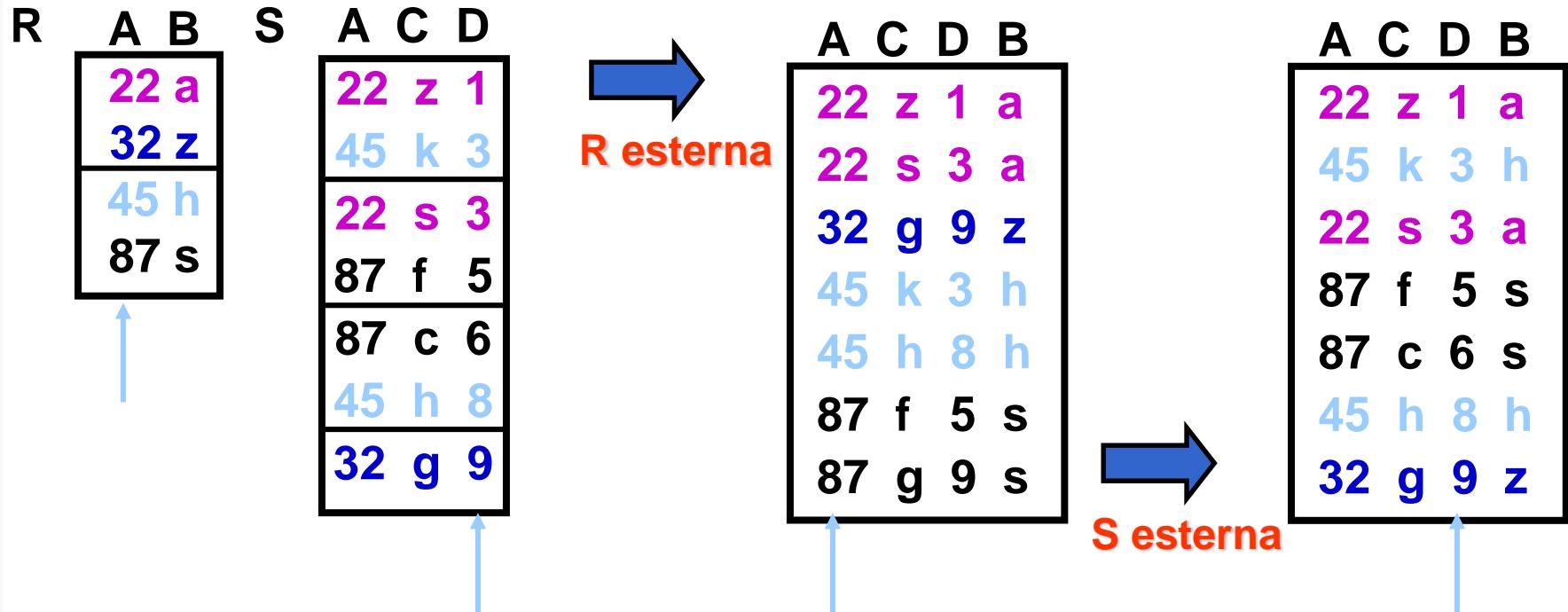
dove NP_R e NP_S sono rispettivamente il numero di pagine di disco necessarie per memorizzare R e S mentre NT_R è il numero di tuple in R .

Scelta della relazione interna

- La scelta della relazione esterna può dipendere da vari fattori:
 - La possibilità di mantenere in memoria centrale tutta la relazione interna porta a scegliere come esterna la relazione con più pagine
 - Se i buffer sono in numero limitato, trascurando il costo di lettura della relazione esterna, si sceglierà R come esterna e S come interna se **NT(R) * NP(S) < NT(S) * NP(R)**, ovvero se:
$$\frac{NT(R)}{NP(R)} < \frac{NT(S)}{NP(S)}$$
che corrisponde a dire che **le tuple di R sono più grandi di quelle di S**
 - Vi sono però altre considerazioni da fare, non meno importanti...

Scelta della relazione interna

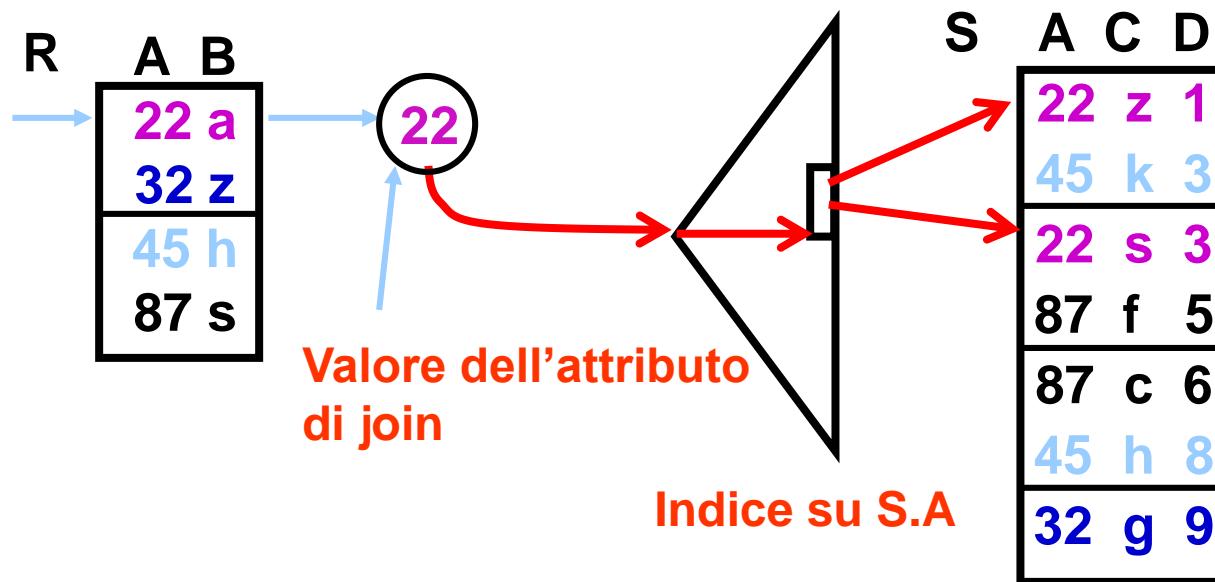
- L'ordine con cui vengono generate le tuple del risultato coincide con l'ordine eventualmente presente nella relazione esterna



Pertanto se l'ordine che si genera è “interessante”, ad esempio perché la query contiene **ORDER BY R.A**, la scelta della relazione esterna può risultarne influenzata

Nested loop in presenza di indici

- Data una tupla della relazione esterna R, la scansione completa della relazione interna S può essere sostituita da una scansione basata su un indice costruito sugli attributi di join di S, secondo il seguente schema:

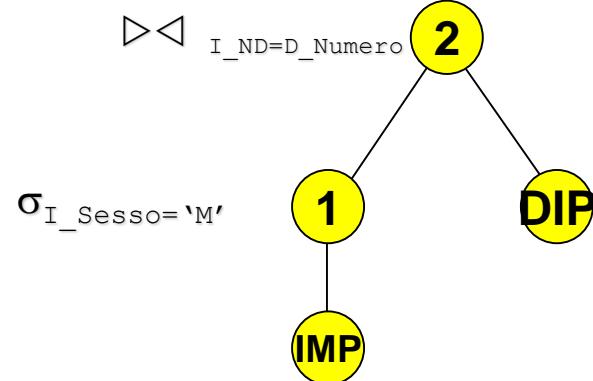


- L'accesso alla relazione interna mediante indice porta in generale a ridurre di molto i costi di esecuzione del Nested Loops Join

Nested loop in presenza di indici

IMP(I_SSNI_Nome,I_Cognome,I_DataN,I_Indirizzo,I_Sesso,I_Stipendio,I_Super:IMP,I_ND:DIP)
DIP(D_Numero,D_Nome,D_SSNDir:IMP,D_Datalnizio)

select I_SSNI_Nome, I_Nome, I_Cognome, D_Nome
from IMP,DIP
where I_ND=D_Numero and I_Sesso='M'



$$\text{costo}(R) + \text{ETR} \times \text{costo}(S)$$

Senza indice

$$\text{costo}(S) = NPS$$

Con indice

$$\text{costo}(S) = h - 1 + \lceil 1 / NK \cdot NL \rceil + \lceil \text{sel(pred)} \cdot NP \rceil$$

clustered

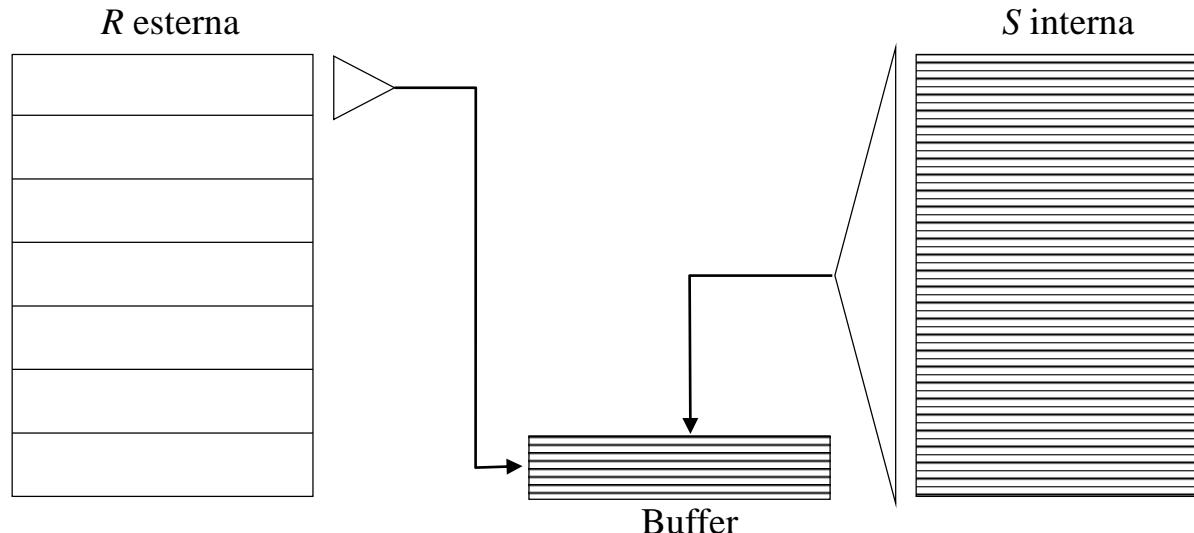
$$\text{costo}(S) = h - 1 + \lceil 1 / NK \cdot NL \rceil + 1 \cdot \Phi(NR / NK, NP)$$

unclustered

Tecniche di join: nested block

- ❑ E' una versione più sofisticata del nested loop, basata sullo stesso principio di esecuzione.
- ❑ Sfrutta un buffer di B pagine in memoria centrale per ridurre il numero di scansioni della relazione interna S . La scansione di S viene eseguita per ogni gruppo di $(B-1)$ pagine della relazione esterna R che vengono preventivamente caricate in memoria
- ❑ Il costo di esecuzione del join in termini di pagine di disco lette si riduce proporzionalmente alla dimensione del buffer. Assumendo l'assenza di prediciati locali diventa :

$$NP_R + \lceil NP_R / (B-1) \rceil \times NP_S$$



Tecniche di join: sort merge

- Sfrutta l'ordinamento delle tuple rispetto all'attributo di join per ridurre il numero dei confronti. Le due tabelle R e S devono essere quindi preventivamente ordinate su tale attributo, ciò consente di scandirle parallelamente alla ricerca di coppie di tuple con lo stesso valore.

```
sort R on cR;
sort S on cS;
CurrR = read(R);
CurrS = read(S);
while not EOF(R) and not EOF(S)
    if (CurrR.cR = CurrS.cS) then
        { restituisci CurrR + CurrS;
          CurrR = read(R);
          CurrS = read(S);
        }
    else if (CurrR.cR > CurrS.cS) then
        CurrS = read(S);
    else
        CurrR = read(R);
next;
close R;
close S;
```

- Se le due relazioni sono già ordinate il vantaggio computazionale nell'utilizzo del sort-merge rispetto al nested loop è evidente poiché le tabelle vengono scandite una sola volta. Quindi il costo di esecuzione è pari a:

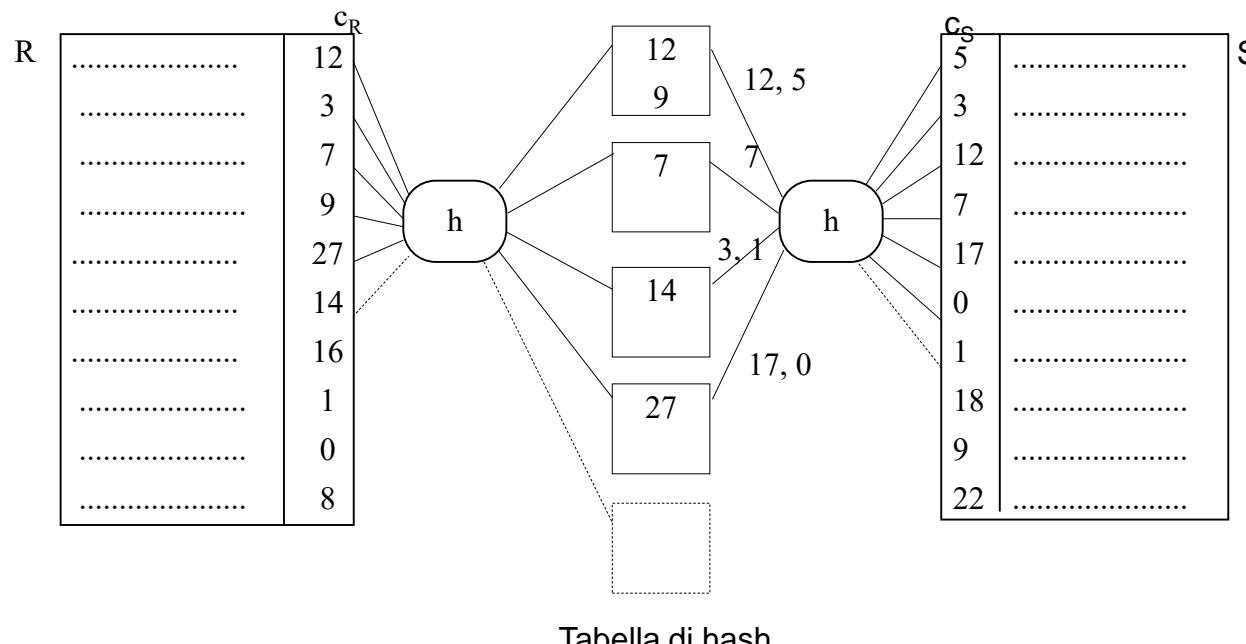
$$\mathbf{NP_R + NP_S}$$

- Nel caso in cui le due relazioni non siano ordinate è necessario valutare il trade-off dato dal costo di ordinamento. Quindi il costo di esecuzione è pari a:

$$\mathbf{Sort(R) + Sort(S) + NP_R + NP_S}$$

Tecniche di join: hash join

- Sfruttando le proprietà delle funzioni di hash è possibile ridurre il numero di confronti da effettuare per eseguire il join senza dover ordinare preventivamente le relazioni.
 - Date due relazioni R e S da porre in equi-join sugli attributi c_R e c_S ($c_R = c_S$)
 1. Si applica ai valori di chiave della prima, detta **relazione di build**, una funzione di hash h che partiziona i valori del dominio di c_R in B frammenti (*bucket*).
 2. Si applica la medesima funzione h ai valori di c_S . La relazione S , detta **relazione di probe**, verrà similmente partizionata e sarà quindi possibile eseguire il join confrontando tra loro solo le tuple appartenenti a partizioni corrispondenti.



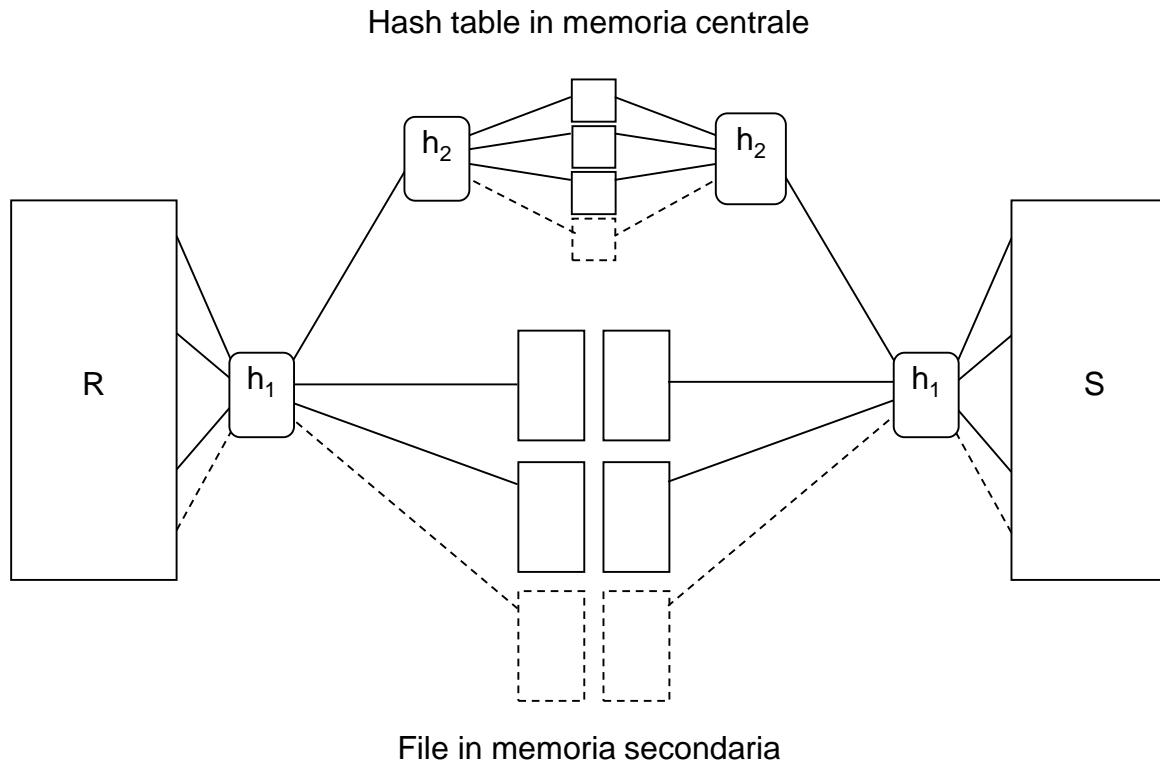
Tecniche di join: hybrid hash join

- Il costo del join può essere stimato come la scansione di entrambe le tabelle e quindi pari a $NP_R + NP_S$
- Qualora, a causa delle sue dimensioni, la tabella di hash (*hash table*) non potesse essere mantenuta in memoria l'hash join dovrebbe essere modificato consentendo la gestione della stessa su memoria di massa.
- L'*hybrid hash join* risolve il problema utilizzano due diverse funzioni di hash h_1 e h_2 : la prima suddivide le tuple delle relazioni in B_1 frammenti di dimensione sufficientemente limitata da far sì che le hash table costruite utilizzando la funzione h_2 possano essere mantenute in memoria centrale.
- Sebbene siano possibili varie forme di ottimizzazione assumeremo per semplicità che il DBMS utilizzi l'HHJ quando nessuna delle due tabelle può essere contenuta in memoria centrale.
 - Supponiamo che $NP_R < NP_S$ e che $NP_R < NB$ allora applichiamo h a R costruiamo la hash table in memoria, poi leggiamo S e applichiamo nuovamente h

Tecniche di join: hybrid hash join

- Con questa tecnica il processo di join può essere così riassunto:
- **Fase 1 build:** R viene letta e alle tuple vengono applicati eventuali predicati locali; alle tuple che risultano eligibili per il join viene applicata la funzione h_1 . Se il valore corrente di c_R è associato al primo dei B_1 bucket la tupla viene mantenuta in memoria e a c_R viene direttamente applicata la funzione h_2 per creare la hash table in memoria centrale; in caso contrario, la tupla viene salvata, in base alla funzione h_1 , in uno dei $B_1 - 1$ file predisposti su disco.
- **Fase 1 probe:** S viene letta e alle tuple vengono applicati eventuali predicati locali; alle tuple che risultano eligibili per il join viene applicata la funzione h_1 . Se il valore corrente di c_S è associato al primo dei B_1 bucket la tupla viene mantenuta in memoria e a c_S viene direttamente applicata la funzione h_2 identificando così il bucket della tabella di hash in memoria centrale con i cui elementi dovranno essere effettivamente effettuati i confronti. Nel caso in cui il valore corrente di c_S non sia associato al primo bucket, la tupla è salvata, in base alla funzione h_1 , in uno dei $B_1 - 1$ file predisposti su disco.
- **Fase $i = 2 \dots B_1$ build:** l' i -esimo file della relazione di build R viene letto in memoria centrale e viene costruita la tabella di hash utilizzando la funzione h_2 .
- **Fase $i = 2 \dots B_1$ probe:** l' i -esimo file della relazione di probe S viene letto in memoria centrale e ai valori di c_S viene applicata la funzione di hash h_2 per determinare il bucket con cui effettuare il confronto.

Tecniche di join: hybrid hash join



- Per l'hybrid hash join il costo di esecuzione può essere stimato come:
 $2(NP_R + NP_S) + NP_R + NP_S = 3(NP_R + NP_S)$

Tecniche di proiezione

- Se l'insieme di attributi su cui proiettare rappresenta una chiave della relazione la proiezione è semplice
 - La cardinalità del risultato è la stessa della relazione di partenza
 - Il risultato si ottiene riportando solo i valori degli attributi di interesse
 - **Pertanto il costo di esecuzione è pari alla scansione delle NP pagine della relazione**
- Se l'insieme di attributi su cui proiettare non rappresenta una chiave della relazione è necessario eliminare le tuple duplicate
 - La cardinalità del risultato è minore di quella della relazione di partenza
 - L'eliminazione viene effettuata mediante ordinamento e mantenimento di una sola delle tuple che presentano gli stessi valori di ordinamento
 - **Pertanto il costo di esecuzione viene stimato come costo di ordinamento della relazione**

Tecniche di aggregazione

- L'operazione di aggregazione (clausola GROUP BY in SQL), non prevista nell'algebra relazionale standard, comporta l'applicazione un operatore di aggregazione a gruppi di tuple con valori omogenei rispetto agli attributi di aggregazione.
- Il risultato si ottiene:
 - Ordinando le tuple sulla base di tutti gli attributi coinvolti nella clausola di GROUP BY
 - Applicando l'operatore agli attributi coinvolti nella formula di aggregazione che presentano gli stessi valori di ordinamento
- **Il costo di esecuzione può essere stimato come costo di ordinamento della relazione**

Tecniche di aggregazione

- In caso di selezione dopo un'aggregazione (clausola HAVING in SQL), può essere necessario calcolare la cardinalità del risultato dell'aggregazione
- Una stima può essere ottenuta utilizzando la formula di cardenas $\Phi(R, N)$ dove:
 - R è il numero di record da aggregare
 - N è il numero di potenziali valori distinti del group-by set a valle del raggruppamento
- Se il valore di R è limitato $\Phi(R, N) < N$. Ossia tra gli R record potrebbero non esserci istanze corrispondenti ad alcuni dei valori del group-by set aggregato

Riassumendo...

- Dimensione di una relazione: $NP = \lceil NR \times \text{len}(t) / (D \times u) \rceil$
- Costo di ordinamento: $2 \times NP \times (\lceil \log_{NB-1} \lceil NP/NB \rceil \rceil + 1)$
- Costo di ordinamento con selezioni : $(NP + EP) + 2 \times EP \times \lceil \log_{NB-1} \lceil EP/NB \rceil \rceil$
- Scansione sequenziale tabelle
 - in media a $(NP+1)/2$ blocchi
 - nel caso peggiore a NP blocchi
 - In caso di non esistenza del record si accede a NP blocchi
- Scansione binaria tabelle
 - in media a $\lfloor \log_2 NP \rfloor$ blocchi
 - nel caso peggiore a $\lfloor \log_2 NP \rfloor + 1$ blocchi
 - In caso di non esistenza del record si accede a $\lfloor \log_2 NP \rfloor + 1$ blocchi
- Accesso a tabelle con indice
 - Numero di foglie del B+-Tree: $NL = \lceil (NK \cdot \text{len}(k) + NR \cdot \text{len}(p)) / (D \cdot u) \rceil$
 - Numero di foglie del Bitmap: $NL = \lceil (NK \cdot \text{len}(k) + NK \cdot NR / 8) / D \rceil$
 - Indice clustered $h - 1 + \lceil EK / NK \cdot NL \rceil + \lceil \text{sel(pred)} \cdot NP \rceil$
 - Indice Unclustered $h - 1 + \lceil EK / NK \cdot NL \rceil + EK \cdot \Phi(NR / NK, NP)$
- Proiezione
 - Mantenendo la chiave della relazione: **NP**
 - Perdendo la chiave della relazione: **assimilabile al costo di ordinamento**
- Raggruppamento: **assimilabile al costo di ordinamento**

Riassumendo: i join

□ Nested loop:

- senza predicato di selezione $NP_R + NR_R \times NP_S$
- con predicato di selezione $NP_R + (sel(pred) \times NR_R) \times costo(S)$
 - Con indice clustered $costo(S) = h - 1 + \lceil 1 / NK \cdot NL \rceil + \lceil sel(pred) \cdot NP \rceil$
 - Con indice unclustered $costo(S) = h - 1 + \lceil 1 / NK \cdot NL \rceil + 1 \cdot \Phi(NR / NK, NP)$

□ Nested block: $NP_R + \lceil NP_R / (B-1) \rceil \times NP_S$

□ Sort merge join

- Tabelle già ordinate $NP_R + NP_S$
- Tabelle da ordinare $Sort(R) + Sort(S) + NP_R + NP_S$

□ Hash join: $NP_R + NP_S$

- Se $NP_R < NB$ oppure $NP_S < NB$

□ Hybrid Hash join: $3(NP_R + NP_S)$

- Se $NP_R \geq NB$ e $NP_S \geq NB$

... e in ORACLE

- Il piano di accesso ai dati viene costruito componendo le modalità di base di accesso alle tabelle. Elenchiamo di seguito i principali e rimandiamo al manuale (Designing and Tuning for Performance – Tabella 12-3) per un elenco completo, necessario all'interpretazione degli alberi di esecuzione
- **Table access:** accede ai dati presenti su una tabella. Può essere dei seguenti tipi
 - **Full:** legge sequenzialmente tutte le righe di una tabella verificando eventuali condizioni sui suoi attributi.
 - **By Rowid:** l'accesso alle singole tuple di una tabella viene effettuato sfruttando il relativo RID che specifica il blocco dati e la posizione all'interno del blocco dati in cui è memorizzata la tupla. Normalmente questo tipo di accesso segue l'accesso a un indice.
 - **By Index RowID:** l'accesso è basato sul RID che è memorizzato in un indice
 - **Sample table scan:** recupera in modo random una porzione di dati da una tabella. Questa modalità è disponibile solo in presenza della clausola sample.

... e in ORACLE

- **Index Scan:** legge i dati da un indice sulla base di una o più delle sue chiavi. Le informazioni contenute nell'indice possono essere sufficienti per rispondere all'interrogazione, in caso contrario vengono recuperate le RID che permetteranno l'accesso alle tabelle. L'index scan può essere dei seguenti tipi:
 - **Unique scan:** utilizzabile se deve essere recuperata una sola tupla
 - **Range scan:** utilizzabile se devono essere recuperate più tuple relative a un range di valori.
 - **Full scan:** utilizzabile se esiste un predicato che coinvolge un attributo dell'indice o, in assenza, di predici per evitare un'operazione di sort.
 - **Fast full scan:** alternativo al full scan viene impiegato quando l'indice contiene tutti i campi necessari all'interrogazione e non è quindi necessario accedere alle tabelle.

... e in ORACLE

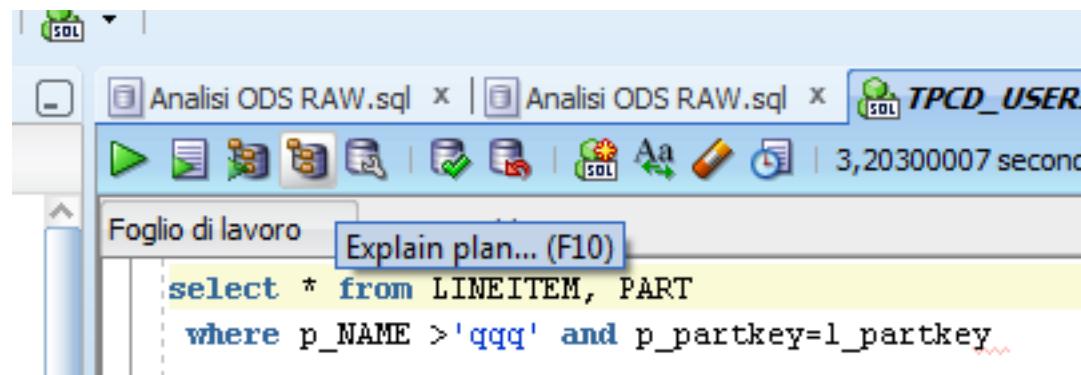
- **Join:** prende in input due insiemi di tuple e restituisce il loro join. In base alla tecnica utilizzata può essere:
 - **Hash**
 - **Nested loops**
 - **Merge join**
- **Sort:** identifica tutte le operazioni di ordinamento. In base al tipo di utilizzo che viene fatto dei dati ordinati si ha:
 - **Aggregate:** restituisce una singola tupla che è il risultato dell'applicazione di un'operazione di raggruppamento a un insieme di tuple
 - **Unique:** utilizzato per l'eliminazione dei duplicati da un insieme di tuple
 - **Group by:** ordina un insieme di tuple in più gruppi in corrispondenza della clausola GROUP BY
 - **Join:** ordina un insieme di tuple in previsione di un'operazione di join
 - **Order by:** ordina un insieme di tuple in corrispondenza della clausola ORDER BY

L'ottimizzatore

- Compito dell'ottimizzatore è costruire un albero di esecuzione ottimizzato a partire dall'albero di esecuzione iniziale
 - **Ottimizzatori rule-based:** utilizzano *regole euristiche* che considerano la struttura dell'albero di esecuzione iniziale e le caratteristiche del DBMS (indici, modalità di accesso ai dati a disposizione) per determinare il piano di esecuzione di costo minimo
 - **Ottimizzatori cost-based:** utilizzano in più *le statistiche* relative all'istanza dei dati per calcolare il costo effettivo di esecuzione dell'informazione
- L'esecuzione di una istruzione SQL su due istanze diverse di uno stesso schema di database potrà utilizzare due piani di esecuzione diversi se si utilizza un ottimizzatore *cost-based*, mentre utilizzerà lo stesso piano se l'ottimizzatore è di tipo *rule-based*.

EXPLAIN PLAN

- **EXPLAIN PLAN** è il comando utilizzato da ORACLE per visualizzare il piano di esecuzione prescelto dall'ottimizzatore.
- La visualizzazione in formato testuale è organizzata in una struttura ad albero che specifica:
 - L'operazione eseguita
 - L'oggetto su cui agisce l'operazione
 - Il numero di righe coinvolte
 - Il costo dell'operazione (non è espresso in unità di misura, è un numero puro calcolato dall'ottimizzatore di Oracle e utilizzabile per confronti)
- Le informazioni relative al piano vengono memorizzate nella PLAN_TABLE che fa parte dello schema

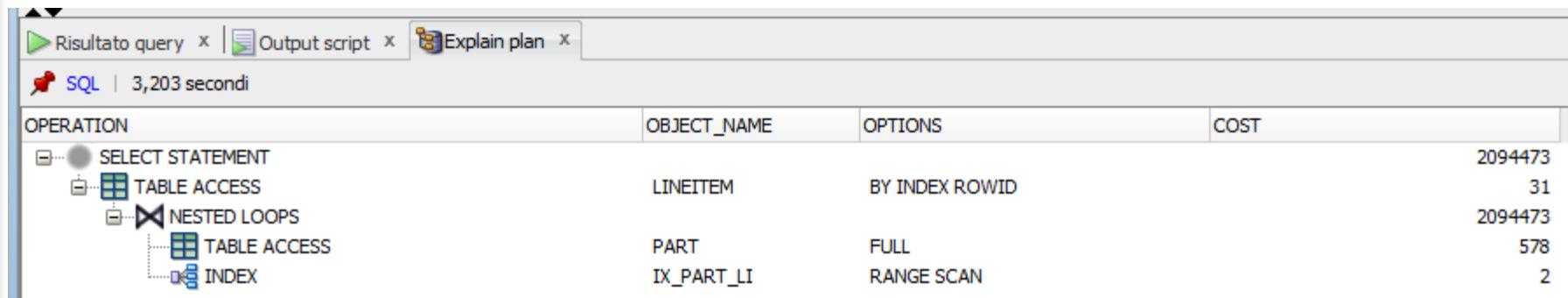


The screenshot shows the Oracle SQL Developer interface. The title bar says "Analisi ODS RAW.sql". The toolbar includes icons for running queries, saving, and zooming. The status bar at the bottom right shows "3,20300007 secondi". The main window has a tab titled "Foglio di lavoro" and a sub-tab titled "Explain plan... (F10)". Below these tabs, a SQL query is displayed:

```
select * from LINEITEM, PART
where p_NAME > 'qqq' and p_partkey=l_partkey
```

EXPLAIN PLAN – un esempio

```
select * from LINEITEM, PART  
where p_NAME >'qqq' and p_partkey=l_partkey
```

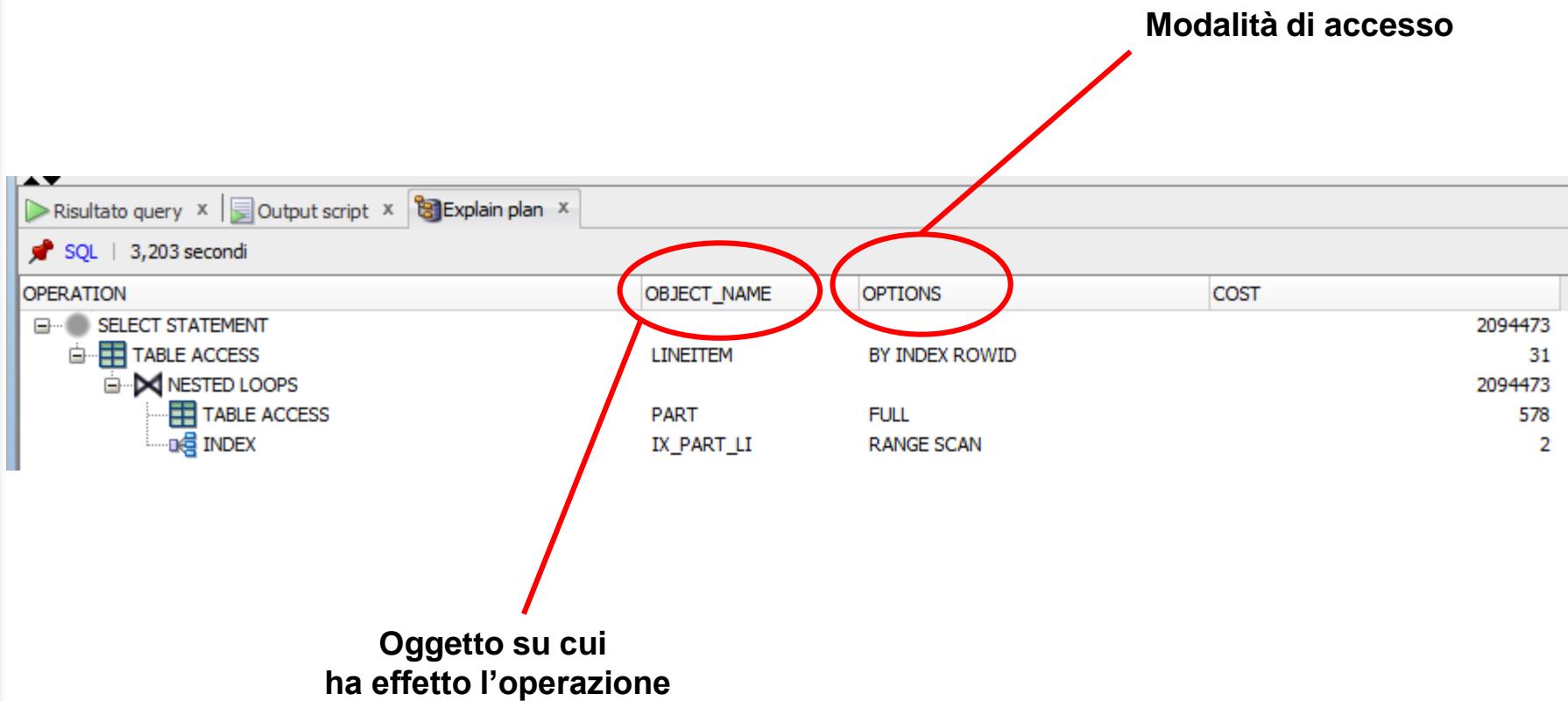


The screenshot shows the Explain plan for the provided SQL query in Oracle SQL Developer. The interface includes tabs for 'Risultato query', 'Output script', and 'Explain plan'. The 'Explain plan' tab is active, displaying the execution plan with the following details:

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			2094473
TABLE ACCESS	LINEITEM	BY INDEX ROWID	31
NESTED LOOPS			2094473
TABLE ACCESS	PART	FULL	578
INDEX	IX_PART_LI	RANGE SCAN	2

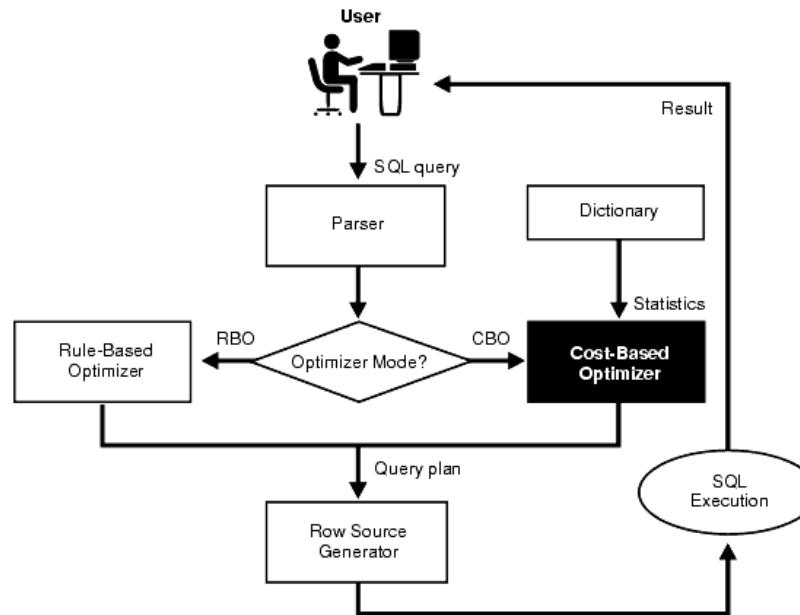
EXPLAIN PLAN – un esempio

```
select * from LINEITEM, PART  
where p_NAME >'qqq' and p_partkey=l_partkey
```



L'ottimizzatore

- In generale, gli ottimizzatori cost-based forniscono soluzioni migliori.
Si utilizza un ottimizzatore rule-based quando:
 - Mantenere le statistiche è troppo oneroso
 - I piani di accesso sono semplici
- Oracle dispone sia di un ottimizzatore rule-based, sia di un ottimizzatore cost-based.



Principi di base per l'ottimizzazione

- Applicare per prime le operazioni che riducono la dimensione dei risultati intermedi
 - Anticipare le operazioni di selezione e di raggruppamento per ridurre il numero delle tuple
 - Anticipare le operazioni di proiezione per ridurre il numero degli attributi
- Applicare per prime le operazioni join e selezione il cui risultato contiene un numero ridotto di tuple
 - Comporta un riordinamento dei nodi dell'albero di esecuzione

EquivALENZA tra alberi di esecuzione

- Più espressioni di algebra relazionale possono essere equivalenti ossia possono determinare lo stesso risultato per ogni stato legale del DB
- Di conseguenza esistono più alberi di esecuzione che determinano lo stesso risultato, ma con costi di esecuzione diversi
- Per passare da un'espressione dell'algebra relazionale all'altra si sfruttano le regole di equivalenza degli operatori
- Nel seguito verranno indicate le regole di base utilizzate nell'ambito dell'ottimizzazione

Regole di trasformazione

1. **Cascata di selezioni:** Una condizione di selezione congiuntiva può essere spezzata in una sequenza di singole operazioni σ

$$\sigma_{c1} \text{ AND } c2 \text{ AND } \dots \text{ AND } cn(R) \equiv \sigma_{c1} (\sigma_{c2} (\dots (\sigma_{cn}(R)) \dots))$$

2. **Commutatività della selezione**

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$$

3. **Cascata di proiezioni:** In presenza di una sequenza di operatori π si possono ignorare tutte le proiezioni fuorché l'ultima

$$\pi_{\text{Lista1}}(\pi_{\text{Lista2}}(\dots(\pi_{\text{Listan}}(R)) \dots)) \equiv \pi_{\text{Lista1}}(R)$$

4. **Commutatività di σ rispetto a π :** se la condizione di selezione c coinvolge solo gli attributi appartenenti alla lista di proiezione $A1, \dots, An$, le due operazioni possono essere commutate

$$\pi_{A1, A2, \dots, An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1, A2, \dots, An}(R))$$

5. **Commutatività di $\triangleright\triangleleft$**

$$R \triangleright\triangleleft_c S \equiv S \triangleright\triangleleft_c R$$

Regole di trasformazione

6. **Commutatività di \bowtie rispetto a σ :** Se tutti gli attributi della condizione di selezione c coinvolgono solamente gli attributi di una delle relazioni su cui viene eseguito il join, le due operazioni possono essere commutate

$$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$$

Alternativamente se la condizione c può essere scritta, sfruttando (1), come $c_1 \text{ AND } c_2$ dove c_1 e c_2 operano rispettivamente su attributi di R e S allora è possibile effettuare la seguente trasformazione

$$\sigma_c(R \bowtie S) \equiv \sigma_{c_1}(R) \bowtie \sigma_{c_2}(S)$$

7. **Commutatività di \bowtie rispetto a π :** Sia la lista di proiezione $L = A_1, \dots, A_n, B_1, \dots, B_n$ dove $A_1, \dots, A_n \in R$ e $B_1, \dots, B_n \in S$, se la condizione di join c coinvolge solamente attributi in L allora le due operazioni possono essere commutate:

$$\pi_L(R \bowtie_c S) \equiv \pi_{A_1, \dots, A_n}(R) \bowtie_c \pi_{B_1, \dots, B_n}(S)$$

Se la condizione di join include oltre a quelli in L ulteriori attributi di R e S: $A_{n+1}, \dots, A_{n+k}, B_{n+1}, \dots, B_{n+k}$. Questi vanno mantenuti sino al join e quindi è necessaria un'ulteriore operazione di proiezione

$$\pi_L(R \bowtie_c S) \equiv \pi_L(\pi_{A_1, \dots, A_n, A_{n+1}, \dots, A_{n+k}}(R) \bowtie_c \pi_{B_1, \dots, B_n, B_{n+1}, \dots, B_{n+k}}(S))$$

Regole di trasformazione

8. **Associatività di \bowtie**

$$(R \bowtie_{c1} S) \bowtie_{c2} T \equiv R \bowtie_{c1} (S \bowtie_{c2} T)$$

9. **Composizione di \times E σ** : se la condizione c di un'operazione di selezione che segue un'operazione prodotto cartesiano rappresenta una condizione di join allora

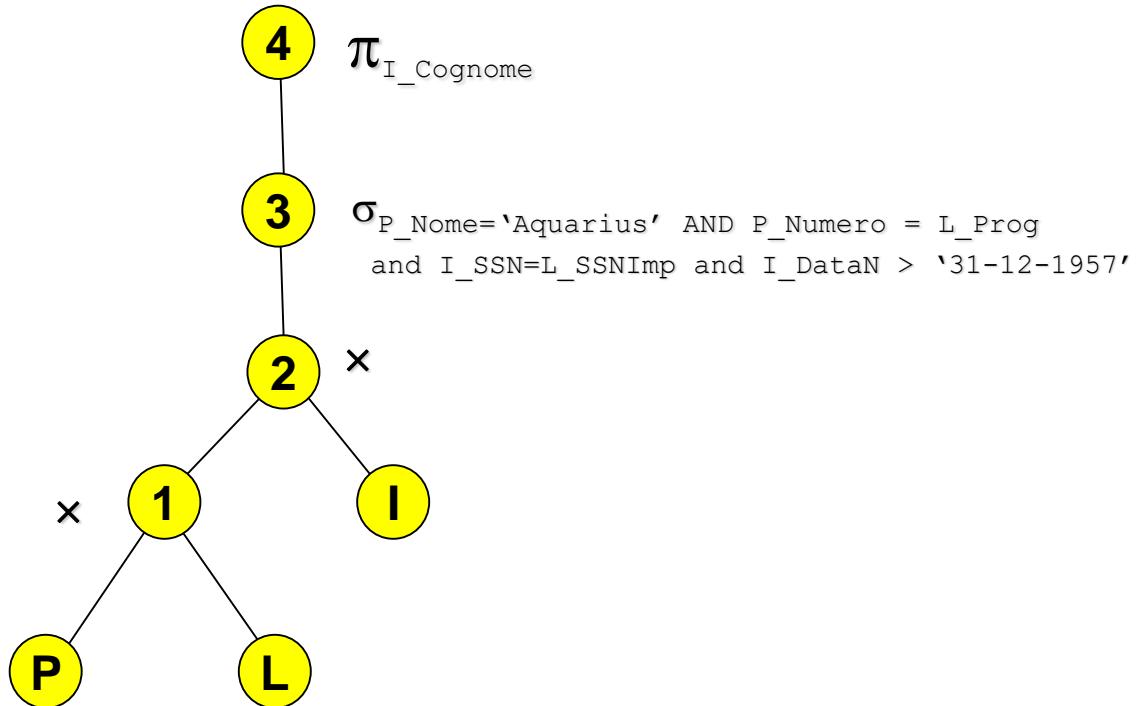
$$\sigma_c (R \times S) \equiv R \bowtie_c S$$

Un algoritmo euristico di ottimizzazione basato su regole

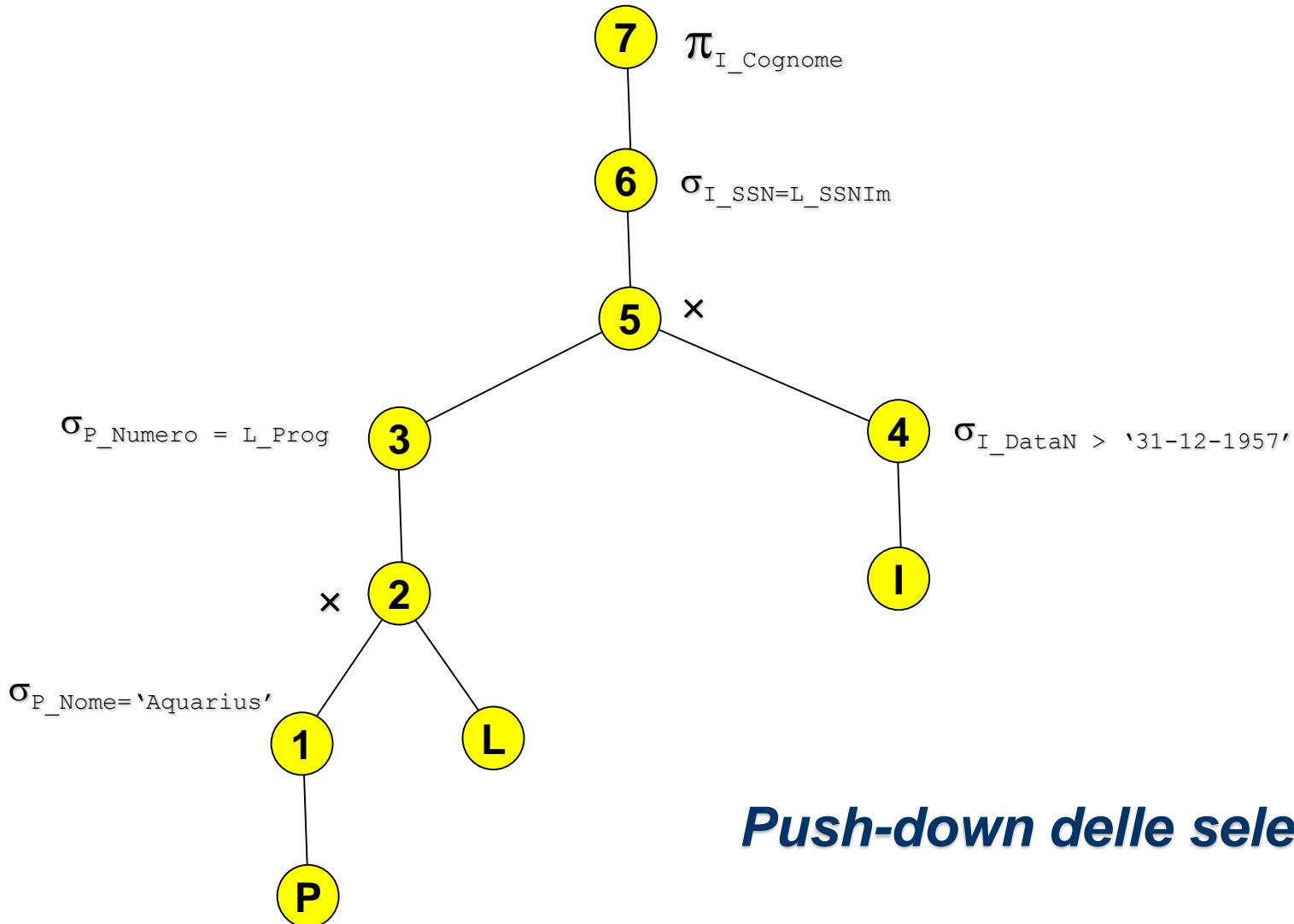
1. **Scomporre i predicati di selezione congiuntiva** utilizzando la regola (1) di equivalenza tra espressioni di AR. In questo modo si garantisce un maggior grado di libertà per lo spostamento delle operazioni di selezione in modo da anticiparle quanto più possibile
2. **Commutare le operazioni di selezione** rispetto alle altre operazioni (regole (2)(4) e (6)), in modo da anticiparle quanto più possibile conformemente a quanto permesso dagli attributi coinvolti nelle selezioni (*push-down selezione*)
3. Usando la regola (9) si **sostituiscano con operazioni di join** le operazioni di prodotto cartesiano seguite da operazioni di selezione
4. Usando le regole (5) e (8) si **modifichi la sequenza di esecuzione delle operazioni di join** in modo da anticipare il join su relazioni in cui insistono **operazioni di selezione più restrittive**
5. Sfruttando le regole (3)(4)(7) si **anticipino quanto più possibile le liste degli attributi di proiezione** creando, quando necessario nuove operazioni di proiezione. Dopo ogni operazione di proiezione, dovrebbero essere mantenuti solamente gli attributi necessari alle operazioni successive e alla generazione del risultato dell'interrogazione (*push-down proiezioni*)

Ottimizzazione euristica – Un esempio

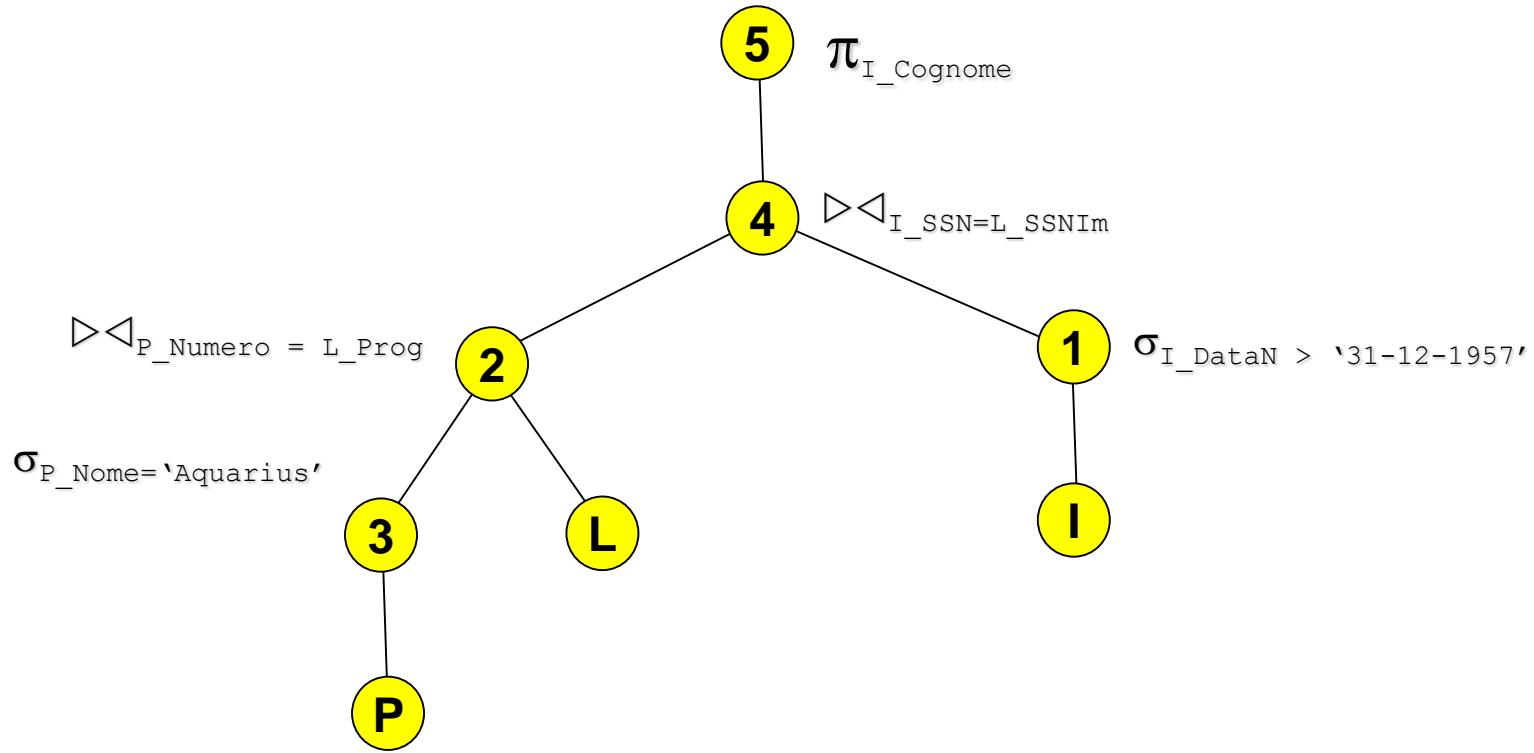
```
select I_Cognome  
from PROG, LAVORASU, IMP  
where P_Nome='Aquarius' AND P_Numero = L_Prog  
      and I_SSN=L_SSNImp and I_DataN > '31-12-1957'
```



Ottimizzazione euristica – Un esempio

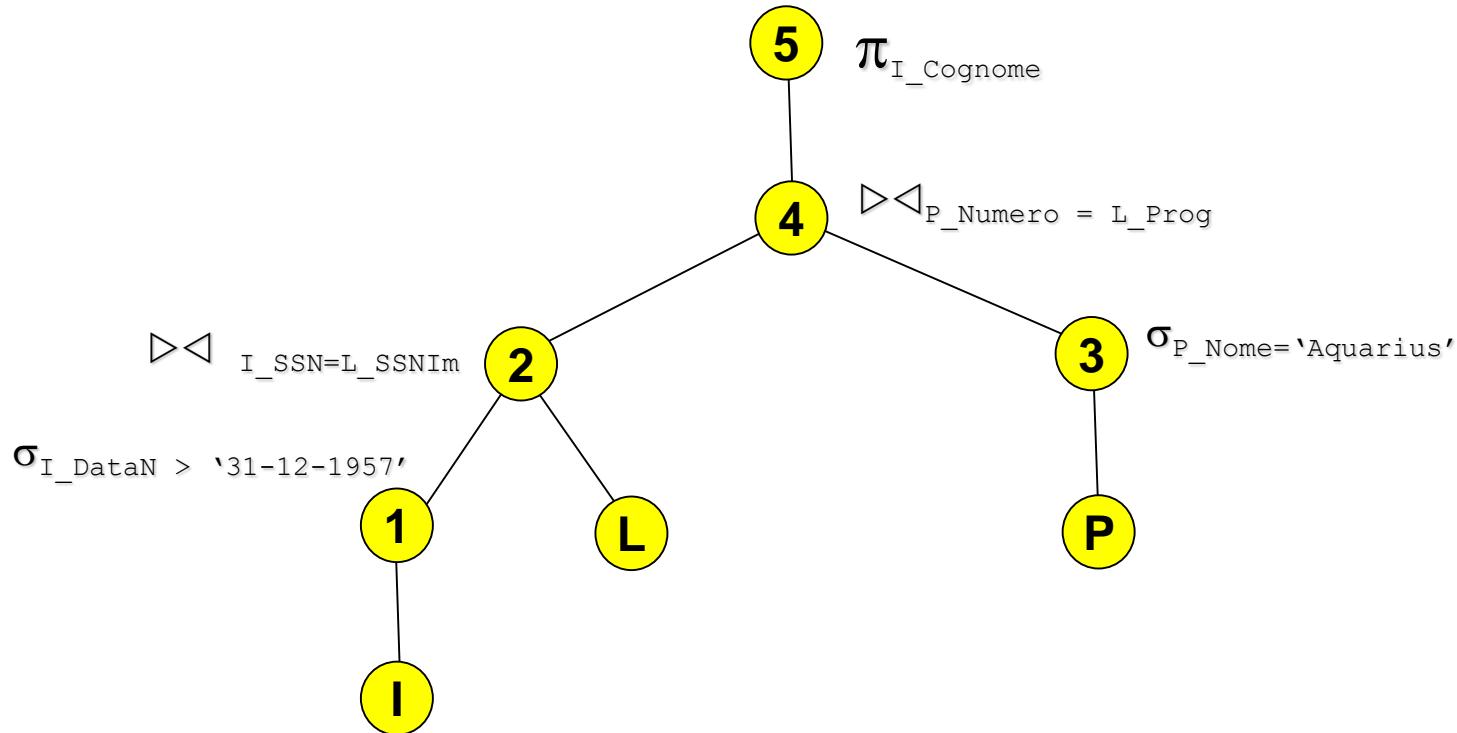


Ottimizzazione euristica – Un esempio



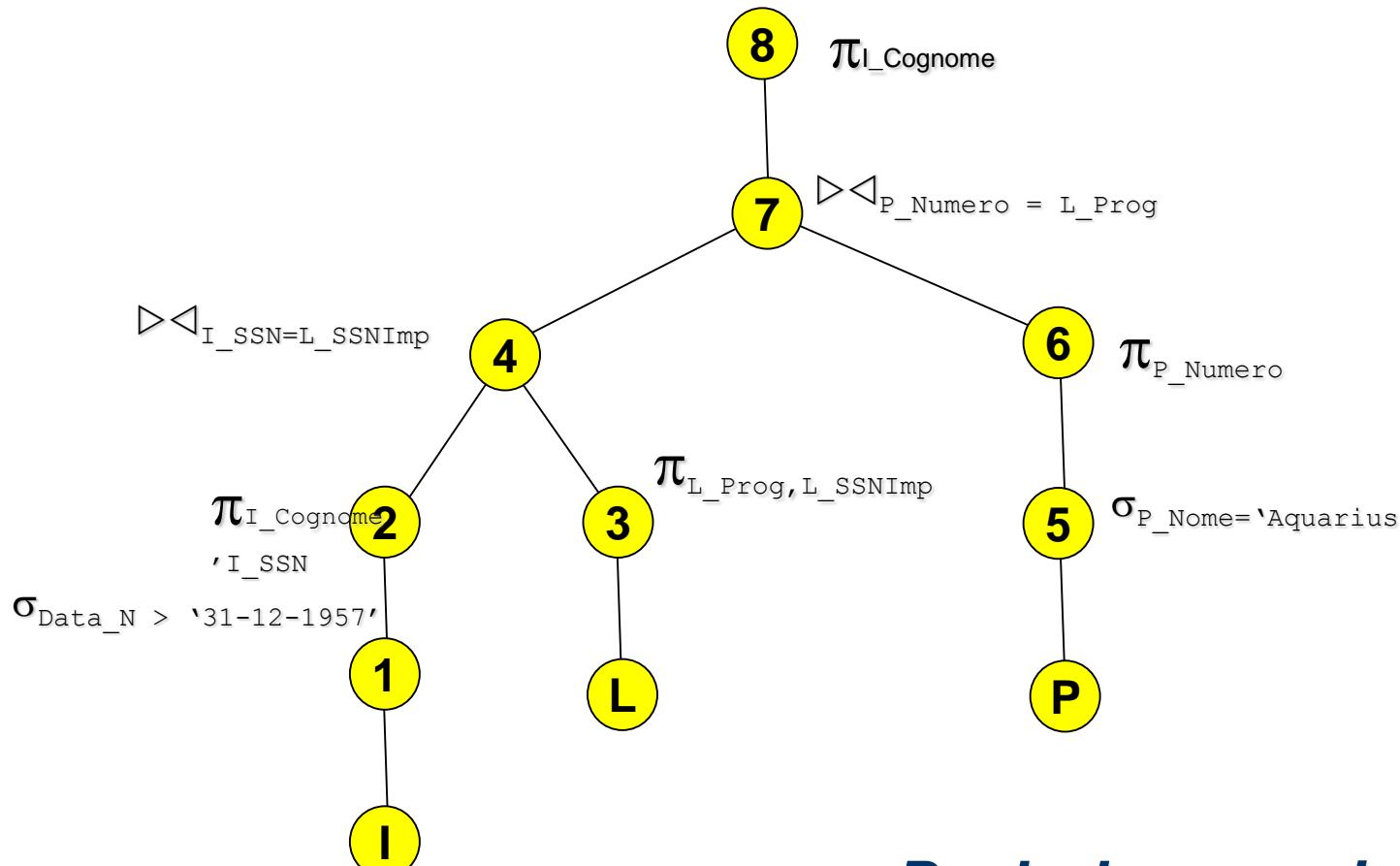
Sostituzione con join

Ottimizzazione euristica – Un esempio



Commutazione join

Ottimizzazione euristica – Un esempio



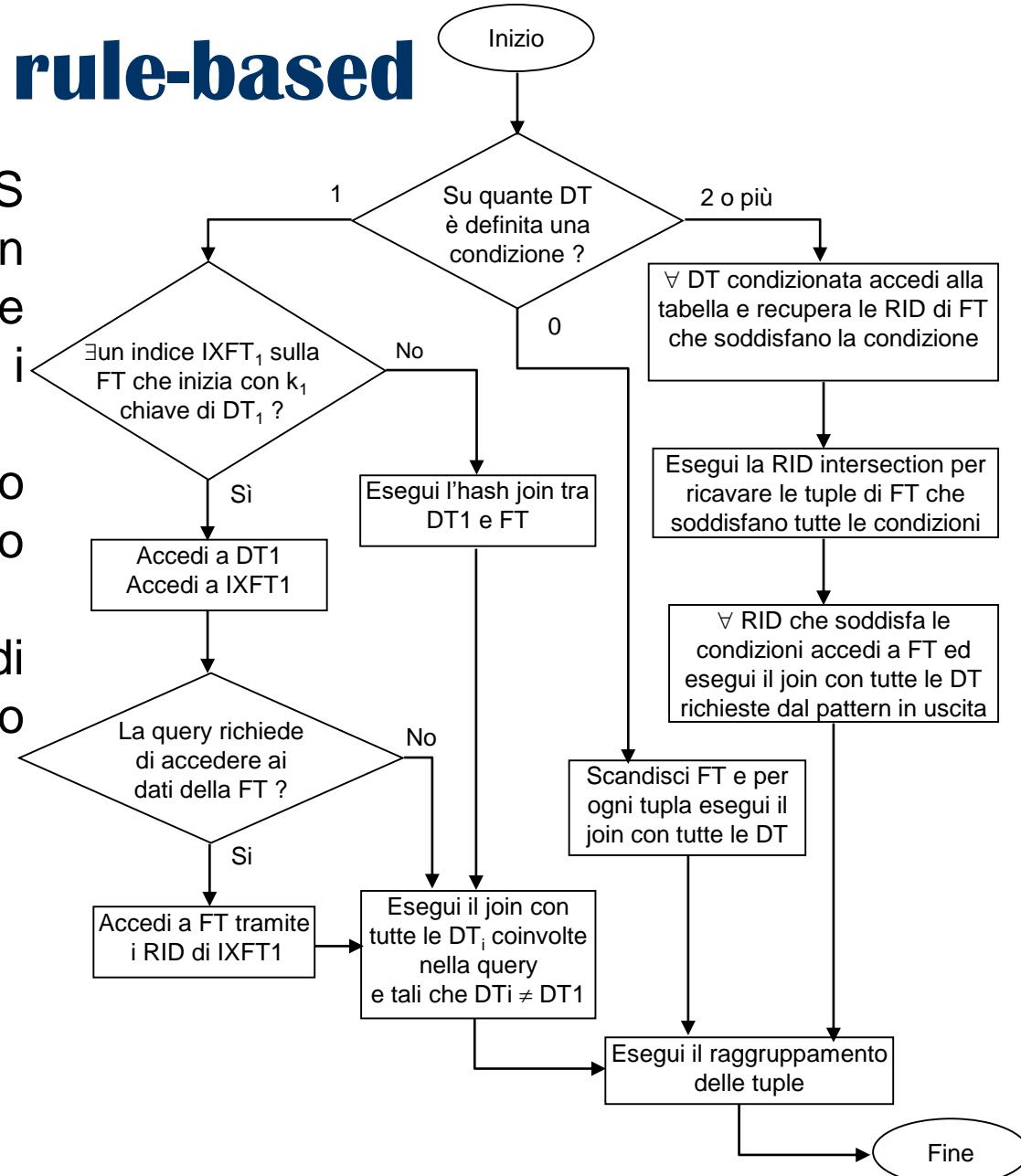
Push-down proiezioni

Ottimizzazione rule-based

- Gli ottimizzatori rule-based definiscono regole euristiche per scegliere le modalità di accesso a tabelle e per determinare gli algoritmi di join da utilizzare
- Per esempio l'ottimizzatore rule-based di ORACLE utilizza 15 modalità diverse per l'accesso a tabelle, al rank più basso del percorso corrispondono costi presunti di accesso inferiori
 - Path 1: Single Row by Rowid
 - Path 2: Single Row by Cluster Join
 - Path 3: Single Row by Hash Cluster Key with Unique or Primary Key
 - Path 4: Single Row by Unique or Primary Key
 - Path 5: Clustered Join
 - Path 6: Hash Cluster Key
 - Path 7: Indexed Cluster Key
 - Path 8: Composite Index
 - Path 9: Single-Column Indexes
 - Path 10: Bounded Range Search on Indexed Columns
 - Path 11: Unbounded Range Search on Indexed Columns
 - Path 12: Sort-Merge Join
 - Path 13: MAX or MIN of Indexed Column
 - Path 14: ORDER BY on Indexed Column
 - Path 15: Full Table Scan

Ottimizzazione rule-based

- I produttori di DBMS rendono note con riluttanza le regole secondo cui operano i propri ottimizzatori.
- Nel seguito è riportato un esempio relativo all'ottimizzatore RULE-BASED di redbrick, in ambito data warehousing



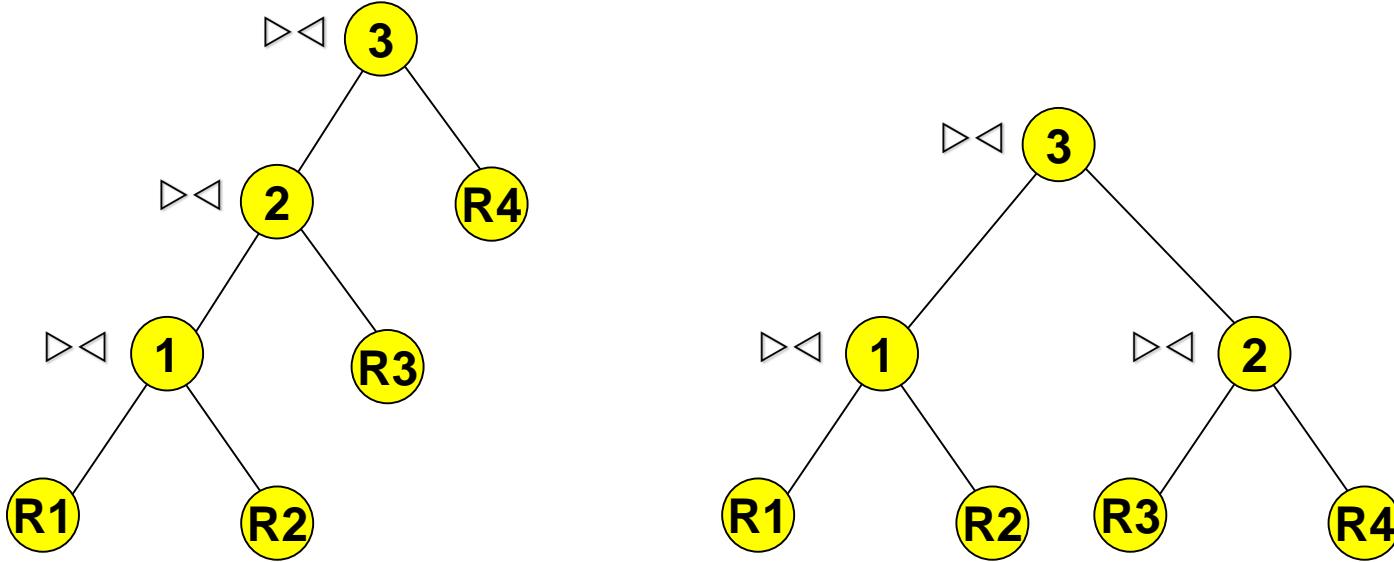
Ottimizzatori cost-based

- Stimano l'effettivo costo di esecuzione di una query mediante funzioni di costo che modellano il comportamento dei diversi algoritmi utilizzati
- Le diverse modalità di esecuzione delle singole operazioni vengono combinate in base alla struttura dell'albero di esecuzione; vista la dimensione dello spazio di ricerca, **il costo di esecuzione viene stimato solo per un sottoinsieme (determinato in base a criteri euristici) dei possibili piani**
- Per poter operare correttamente gli ottimizzatori cost-based devono avere a disposizione statistiche aggiornate sui dati e sugli indici
- Le funzioni di costo determinano un modello semplificato del sistema e così pure le statistiche, quindi le stime che si ottengono potrebbero non essere esatte

Interrogazioni multi-join

- Le regole di trasformazione dell'algebra lineare (5) e (8) permettono di modificare l'ordine dei join e di ottenere espressioni equivalenti.
- Il numero di alberi alternativi aumenta in modo esponenziale rispetto al numero di relazioni presenti in una query
- La stima del costo di esecuzione di ogni possibile albero richiederebbe una cospicua quantità di tempo e risorse.
- Tipicamente gli ottimizzatori cost based si limitano a considerare alberi di esecuzione *left-deep* (profondità a sinistra) o *right-deep* (profondità a destra)
- Un albero **left-deep** è un albero binario in cui il figlio destro di ogni nodo non foglia è sempre una relazione di base
 - Favoriscono l'esecuzione in pipeline dei join senza fare uso di relazioni intermedie: non appena viene prodotta una pagina di tuple risultato del primo join questa può essere utilizzata direttamente come input per i join successivi
 - Il join può sfruttare eventuali strutture dati accessori costruite per le relazioni di base

Interrogazioni multi-join



- Nell'albero di sinistra:
 - Tutte le operazioni di join possono sfruttare eventuali indici costruiti sulle tabelle di base
 - Non appena il join (1) produce una pagina di tuple di risultato si può procedere all'esecuzione del join (2) senza dover memorizzare la pagina su disco
- *Un ulteriore modo per ridurre il numero di alberi da considerare potrebbe essere quello di escludere gli alberi che prevedono un prodotto cartesiano in presenza di un albero che non ne prevede*

Materializzazione vs pipeline

- Un semplice modo di eseguire un piano di accesso composto da diversi operatori consiste nel procedere bottom-up, secondo il seguente schema:
 - Si calcolano innanzitutto i risultati degli operatori al livello più basso dell'albero e si memorizzano tali risultati in relazioni temporanee
 - Si procede quindi in modo analogo per gli operatori del livello sovrastante, fino ad arrivare alla radice
- Tale modo di procedere, detto “**valutazione per materializzazione**”, è altamente inefficiente, in quanto comporta la creazione, scrittura e lettura di molte relazioni temporanee, relazioni che, se la dimensione dei risultati intermedi eccede lo spazio disponibile in memoria centrale, devono essere gestite su disco
 - Nell'esempio precedente i risultati intermedi (1) e (2) potrebbero dar vita a due relazioni temporanee

Materializzazione vs pipeline

- Un modo alternativo di eseguire un piano di accesso è quello di eseguire più operatori in pipeline, ovvero non aspettare che termini l'esecuzione di un operatore per iniziare l'esecuzione di un altro
- Nell'esempio precedente, la valutazione in pipeline opererebbe così:
 - Si inizia a eseguire il primo Join tra R1 e R2. Appena viene prodotta la prima tupla dell'output questa viene passata in input al secondo Join (R2 e R3), che può quindi iniziare la ricerca di matching tuple e quindi produrre la prima tupla del risultato finale della query.
 - La valutazione prosegue cercando eventuali altri match per la tupla prodotta dal primo Join; quando è terminata la scansione di R3, il secondo Join richiede al primo Join di produrre un'altra tupla
- Nel caso di esecuzione in pipeline le stime dei costi viste in precedenza vanno riviste considerando che non tutti i dati vengono letti/scritti su disco quando sono svolte più operazioni in cascata.

Le statistiche

- Per valutare i costi delle varie strategie di esecuzione occorre tenere traccia di alcune informazioni necessarie alle funzioni di costo utilizzate dai DBMS sono:
 - Per le tabelle
 - Numero di tuple di una tabella.
 - Dimensione delle tuple.
 - Numero di blocchi di disco utilizzati
 - Per colonne
 - Valore minimo (*minval*)
 - Valore massimo (*maxval*).
 - Numero di valori distinti (*nval*)
 - Numero di valori nulli
 - Lunghezza media
 - **Distribuzione dei dati (istogrammi)**
 - Per gli indici
 - Numero di foglie
 - Profondità
- Le informazioni relative alle statistiche sono memorizzate sul Data Dictionary nella viste ???_TABLES, ???_TAB_COL_STATISTICS e ???_INDEXES

Le viste del Data Dictionary

```
select TABLE_NAME, TABLESPACE_NAME, NUM_ROWS,BLOCKS,AVG_ROW_LEN  
from USER_TABLES;
```

TABLE_NAME	TABLESPACE_NAME	NUM_ROWS	BLOCKS	AVG_ROW_LEN
CUSTOMER	DATA	150000	6970	158
LINEITEM	DATA	6001215	200485	113
NATION	DATA	25	1	105
ORDERS	DATA	1500000	47087	106
PART	DATA	200000	7655	130
PARTSUPP	DATA	800000	33432	142
PLAN_TABLE	DATA	11683	275	59
REGION	DATA	5	1	113
SUPPLIER	DATA	10000	425	143
TIME	DATA	2557	25	26

Le viste del Data Dictionary

```
Select COLUMN_NAME,NUM_DISTINCT,LOW_VALUE,HIGH_VALUE,NUM_NULLS,AVG_COL_LEN  
from USER_TAB_COL_STATISTICS  
where TABLE_NAME='LINEITEM' ;
```

COLUMN_NAME	NUM_DISTINCT	LOW_VALUE	HIGH_VALUE	NUM_NULLS	AVG_COL_LEN
L_ORDERKEY	1500000	C102	C407	0	6
L_PARTKEY	200000	C102	C315	0	5
L_SUPPKEY	10000	C102	C302	0	4
L_LINENUMBER	7	C102	C108	0	3
L_QUANTITY	50	C102	C133	0	3
L_EXTENDEDPRICE	934059	C20A02	C30B323233	0	6
L_DISCOUNT	11	80	C00B	0	3
.....					

Le statistiche

- Sulla base di queste informazioni, è possibile stimare la selettività di predicati del tipo **attributo oprel valore** (dove gli operatori oprel possono essere $=$, $>$, $<$, \geq , \leq):

$$Sel(= x) = \frac{1}{nval}$$

$$Sel(< x) = Sel(\leq x) = \frac{x - minval}{maxval - minval}$$

$$Sel(> x) = Sel(\geq x) = \frac{maxval - x}{maxval - minval}$$

Le statistiche

- Le precedenti formule sono ottenute assumendo che i valori dell'attributo siano *uniformemente distribuiti* nell'intervallo $[minval, maxval]$.
- Quando, come spesso accade, la distribuzione non è uniforme le formule possono determinare delle forti approssimazioni.
- Uno dei principali passi in avanti nella stima del costo di esecuzione di un'interrogazione è l'introduzione degli *istogrammi* che forniscono una rappresentazione semplificata dell'effettiva distribuzione dei valori per gli attributi delle relazioni di una specifica istanza di database.

Le statistiche

- L'istruzione SQL per calcolare le statistiche è ANALYZE
 - ANALYZE TABLE Prog COMPUTE STATISTICS;
Analizza l'intera tabella
 - ANALYZE TABLE Prog COMPUTE STATISTICS FOR COLUMN P_NOME ;
Restringe l'analisi a un singolo attributo
 - ANALYZE INDEX I1 COMPUTE STATISTICS;
Analizza un indice
- ORACLE mette a disposizione un package specifico DBMS_STATS che permette di effettuare una gestione avanzata delle statistiche. Ad esempio:

```
DBMS_STATS.GATHER_SCHEMA_STATS(OWNNAME=>'USERSI')
```

Gli istogrammi

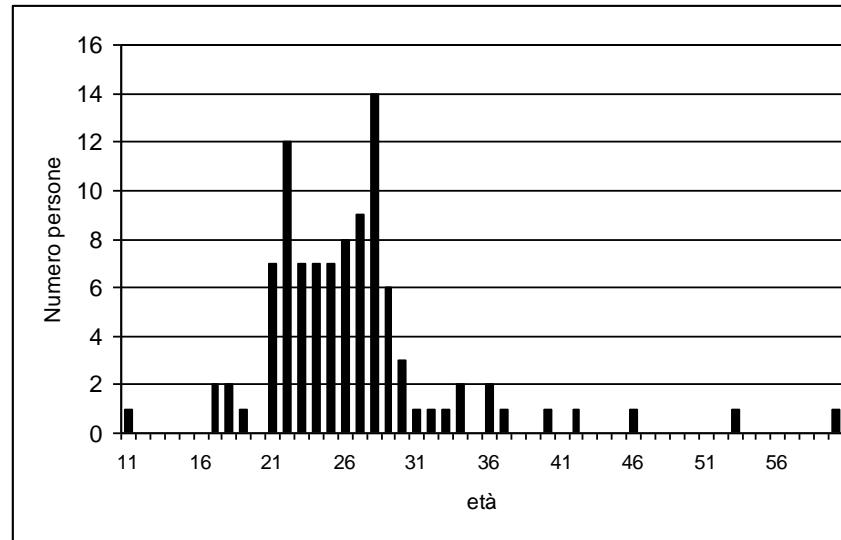
- Un **istogramma** fornisce una rappresentazione semplificata della distribuzione dei valori per gli attributi delle relazioni di una specifica istanza di database.
- La rappresentazione è ottenuta suddividendo lo spazio dei valori in più raggruppamenti o intervalli (**bucket**) e stimando, per ognuno di essi la frequenza media con cui i valori al loro interno si presentano nel database.
 - **Equi-width:** suddividono lo spazio dei valori in intervalli di uguale dimensione.
 - **Equi-height:** la suddivisione è tale che la somma delle frequenze dei valori degli attributi associati a ciascun raggruppamento sia uguale (indipendentemente dal numero di valori che saranno contenuti nell'intervallo).

Gli histogrammi

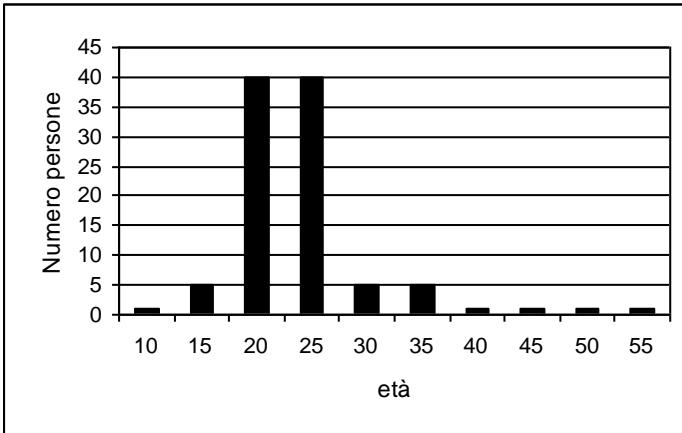
- Gli histogrammi **equi-height forniscono informazioni più accurate**, ma risultano più difficili da calcolare e mantenere aggiornati.
- L'aumento di complessità deriva dal dover calcolare per ogni bucket il range di valori necessario a mantenere costante l'altezza dei bucket
- La scelta della tecnica di costruzione da utilizzare rappresenta il trade-off tra accuratezza della stima da un lato e velocità di aggiornamento e spazio per la memorizzazione delle informazioni dall'altro.
- Le statistiche relative alle sole **cardinalità dei singoli attributi** possono essere viste come un **istogramma degenere** composto da un solo raggruppamento. Ovviamente, alla minor quantità di informazione contenuta in un istogramma degenere corrisponderanno errori più grossolani in fase di stima dei costi.

Gli histogrammi: un esempio

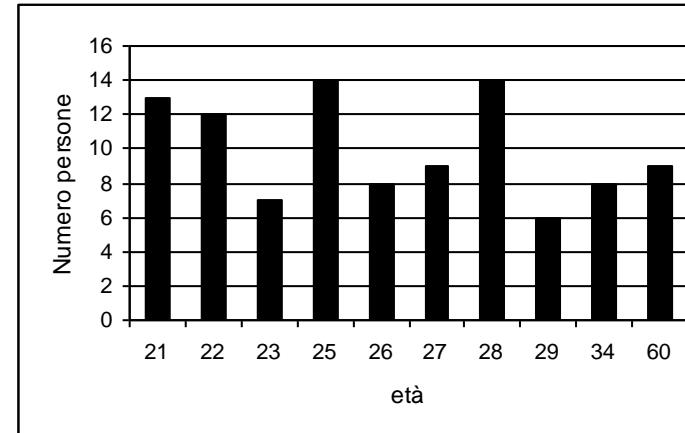
- Distribuzione dei valori dell'attributo età contenuto nella relazione IMPIEGATI di una piccola azienda informatica con 100 dipendenti.



Gli istogrammi: un esempio



Iistogramma
equi-width



Iistogramma
equi-heigh

- Volendo conoscere la porzione di persone con un'età inferiore ai 27 anni si otterrà una stima pari a:
 - Statistiche standard: $Se(\text{età} \leq 27) = \frac{27 - 11}{60 - 11} \approx 0,33$
 - Iistogramma equi-width: $0,46 \leq se(\text{età} \leq 27) \leq 0,86$
 - Iistogramma equi-heigh: $0,54 \leq se(\text{età} \leq 27) \leq 0,63$

Gli histogrammi in ORACLE

- Oracle utilizza histogrammi **equi-height** che vengono costruiti con l'istruzione

```
ANALYZE TABLE nometabella COMPUTE STATISTICS  
FOR ALL COLUMN  
      SIZE valore  
    COLUMNS nomecolonna
```

- Il parametro **SIZE** specifica il numero massimo di bucket da utilizzare.
- Si sconsiglia l'utilizzo degli histogrammi quando:
 - gli attributi hanno distribuzione uniforme
 - sugli attributi vengono espresse condizioni di uguaglianza
- Le informazioni relative agli histogrammi sono memorizzate nel data dictionary nelle viste **???_TAB_COLUMNS** e **???_TAB_HISTOGRAMS**.

Negli esercizi...

- Qualora le informazioni sulle istanze non fossero fornite:
 - La lunghezza media delle tuple va estrapolata dalle statistiche (DBA STUDIO)
 - La lunghezza media dei singoli campi va estrapolata dalle statistiche (Data Dictionary)
 - Il numero di valori distinti di chiave va estrapolato dalle statistiche
 - Il valore minimo e massimo delle chiavi va estrapolato dalle statistiche o mediante query sql (select count(distinct ???)...)
 - Il numero delle foglie degli indici va calcolato in base alle formule date
 - Le stime del numero delle pagine e del numero delle tuple si arrotonda per eccesso.
 - **Si tratta di una scelta semplificativa, che introduce un'approssimazione che può essere anche rilevante: essendo dati stimati non dovrebbero essere arrotondati**

Gli Hints

- Sebbene l'evoluzione degli ottimizzatori li renda sempre più affidabili è spesso necessario apportare delle modifiche al piano di esecuzione proposto dal sistema.
- Tramite gli **hints** (consigli) è possibile specificare:
 - L'obiettivo dell'ottimizzazione
 - Il piano di esecuzione
 - L'ordine dei join
 - La tecnica di join
- La sintassi completa relativa all'utilizzo degli hint è necessaria per lo svolgimento degli esercizi la trovate sul manuale Oracle:

Performance Tuning Guide

Cap. 19 Using optimizer hints

Gli Hints

- In ORACLE gli hints vengono specificati con la sintassi /*+ **hint** */ dove hint appartiene a uno dei seguenti gruppi:
- **Scelta dell'approccio all'ottimizzazione**
 - **ALL_ROWS**: il piano di accesso viene stabilito in modo da massimizzare il throughput del sistema.
 - **FIRST_ROWS**: il piano di accesso viene stabilito in modo da minimizzare il tempo di risposta.
 - **CHOOSE**: il tipo di ottimizzatore da utilizzare (cost-based o rule-based) viene scelto dall'ottimizzatore in base alla presenza delle statistiche.
 - **RULE**: impone l'utilizzo dell'ottimizzatore rule-based.

```
select /*+ FIRST_ROWS */ name from lineitem;
```

Gli Hints

□ Modalità di accesso

- **FULL**: accede alla tabella con modalità full scan
- **ROWID**: accede alla tabella tramite i RID
- **INDEX**: accede alla tabella tramite index scan
- **INDEX_ASC/DESC**: accede alla tabella tramite index scan, i valori dell'indice sono letti in ordine crescente/decrescente
- **INDEX_COMBINE**: impone l'utilizzo di indici bitmap per l'accesso alla tabella
- **NO_INDEX**: impone al sistema di non utilizzare uno specifico insieme di indici per l'accesso alla tabella.

```
select /*+ FULL(lineitem) */ o_clerk, l_quantity  
from lineitem, orders  
where l_orderskey=o_orderskey and o_orderdate=15448;
```

Gli Hints

□ Modalità di join

- **USE_NL**: impone l'utilizzo dell'algoritmo di nested loop per le operazioni di join che coinvolgono la tabella che avrà il ruolo di *inner table*.
- **USE_MERGE**: impone l'utilizzo dell'algoritmo di sort-merge per le operazioni di join che coinvolgono la tabella.
- **USE_HASH**: impone l'utilizzo dell'algoritmo di Hash-join per le operazioni di join che coinvolgono la tabella.
- **LEADING**: la tabella specificata dovrà essere la prima nell'ordine di join
- **ORDERED**: impone che l'ordine di esecuzione del join sia lo stesso in cui compaiono le tabelle nella clausola from.

```
select /*+ ORDERED USE_NL(lineitem,orders) */ o_clerk, l_quantity
from lineitem, orders
where l_orderkey=o_orderkey and o_orderdate=15448;
```

Stima dei costi di esecuzione

- Riprodurre “su carta” il calcolo del costo di esecuzione utilizzato da un DBMS commerciale è pressoché impossibile:
 - Non sono note le funzioni di costo utilizzate
 - Vengono spesso utilizzate semplificazioni euristiche
 - E’ complesso modellare l’utilizzo di statistiche sofisticate quali gli istogrammi
- E’ comunque utile acquisire familiarità con il calcolo dei costi di esecuzione delle interrogazioni:
 - Permette di effettuare stime affidabili nei casi più semplici
 - Permette di capire le scelte dell’ottimizzatore e pertanto
 - Consente di valutare la bontà delle scelte dell’ottimizzatore
 - Consente di valutare come ottimizzare la struttura del database

Stima dei costi di esecuzione

- Il calcolo dei costi di esecuzione prevede le seguenti fasi:
 - Individuazione dei potenziali alberi di esecuzione
 - Valutazione del costo dei diversi algoritmi utilizzabili nei vari nodi
 - Calcolo del costo complessivo