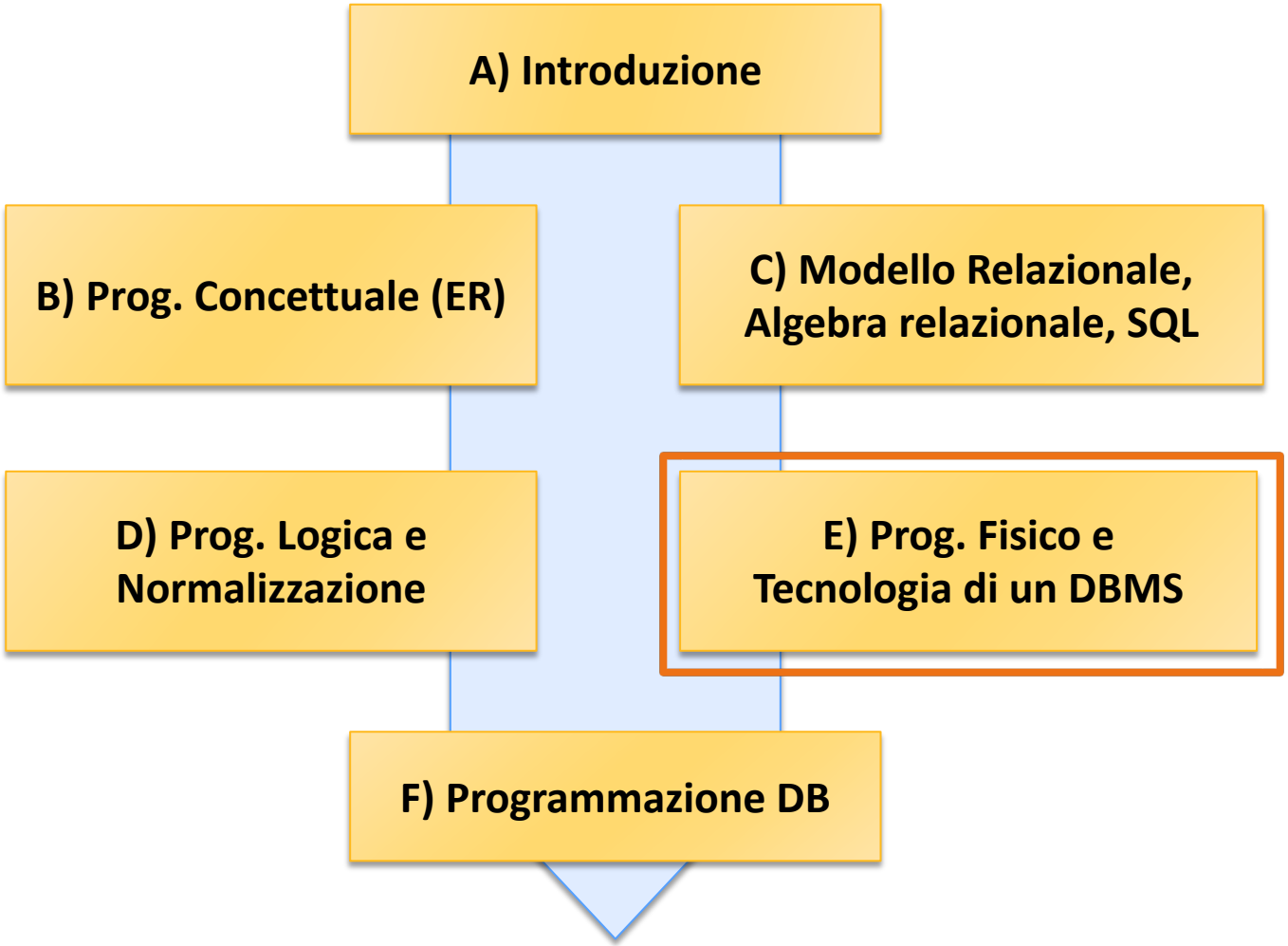
Basi di Dati

Calcolo del Costo di Accesso ai Dati (Parte 2)

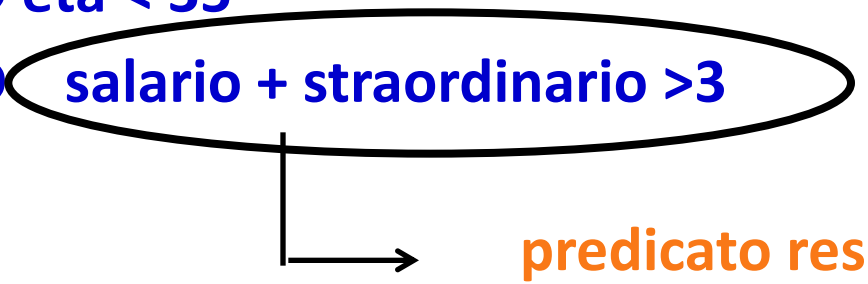
Basi di Dati – Dove ci troviamo?



Uso degli indici

- **Le tuple del risultato vengono poi controllate con i predicati residui:**

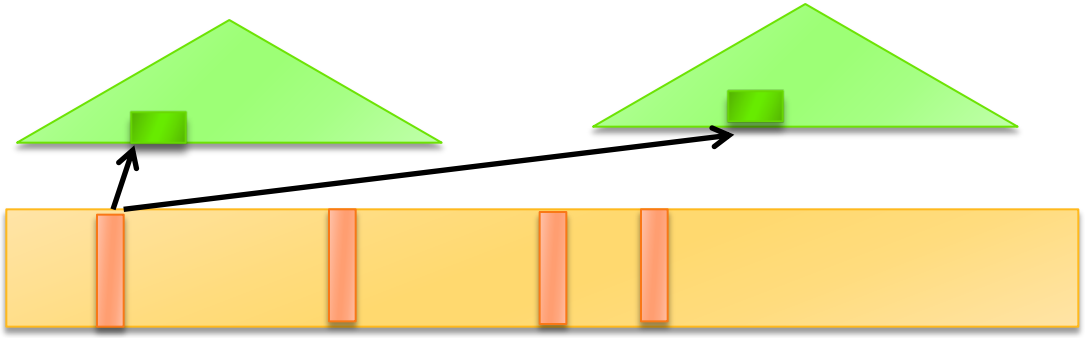
```
SELECT * FROM IMPIEGATI
WHERE lavoro = 'fattorino'
AND età < 35
AND salario + straordinario >3
```



La selettività dei predicati residui è difficile da calcolare

Uso degli indici

- **Perché non mettere indici su tutti gli attributi?** **Il query optimizer potrebbe poi scegliere.**
- **Gli indici devono essere mantenuti:**
  caso DELETE: eliminazione di TID

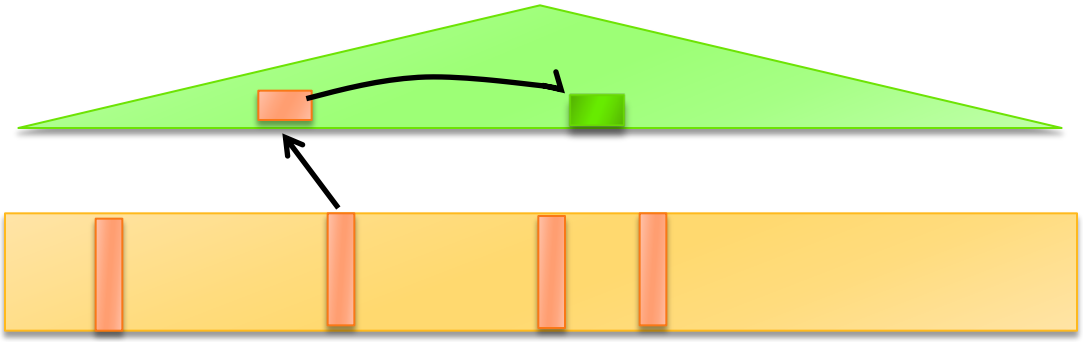


Costo = N_{ind} x 2 x E **(N_{ind} : numero indici)**

2 operazioni di scrittura (foglia e blocco dati), per ogni tupla e per ogni indice

Uso degli indici

caso UPDATE: spostamento di TID



Costo = N_{ind} x 4 x E (**N_{ind}: numero indici**)

2 operazioni di lettura (foglia e blocco dati) e 2 di scrittura (idem) per ogni tupla e per ogni indice

Un numero indici troppo elevato comporta un eccessivo costo di modifica della relazione

Analisi del caso di join

- **scopo:**
  - **valutare quale sia la migliore strategia di accesso per interrogazioni SQL nel caso di join**
  - **i criteri di valutazione servono anche a prendere decisioni sull'ordinamento delle relazioni e quali indici costruire**
  - **I criteri che vedremo sono in linea con i metodi e le scelte utilizzati dai query-optimizer dei DBMS relazionali**

Uso degli indici in caso di join

```
SELECT cognome, salario
FROM impiegati as IMP, dipartimenti as DIP
WHERE lavoro = 'fattorino'
AND sede= 'milano'
AND imp.dno = dip.dno
```

con: impiegati (matr, cognome, nome, lavoro, dno, …)
dipartimenti (dno, nome, sede, …)

impiegati.dno = dipartimenti.dno
(o dipartimenti.dno = impiegati.dno)
è argomento di ricerca ?

Uso degli indici in caso di join

Esecuzione del JOIN con il metodo nested loops:

```
{loop 1}:per ogni tupla della relazione IMP
                    se soddisfa i predicati su IMP allora
{loop 2}:  per ogni tupla della relazione DIP
                    se soddisfa i predicati su DIP e
                    se soddisfa il predicato di jOIN
                              allora la giunzione delle due
                    tuple fa parte del risultato
          fine
fine
```

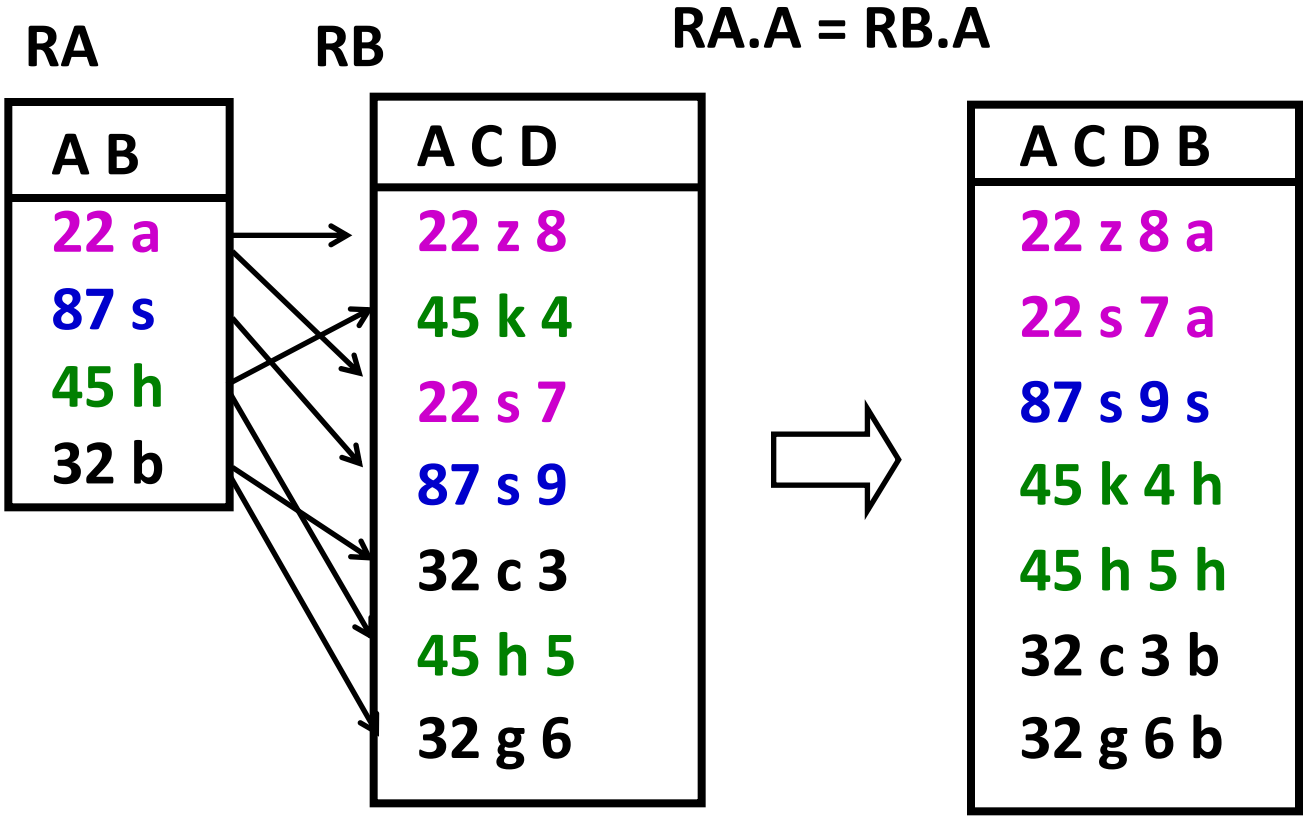
Nell'esempio: sono possibili due sequenze:

IMP ⇒ DIP (cioè IMP esterna, DIP interna)

DIP ⇒ IMP (viceversa)

Uso degli indici in caso di join

Esecuzione del JOIN con il metodo nested loops:



Uso degli indici in caso di join

- un predicato di join è utilizzabile come argomento di ricerca (ed è utilissimo!) solo se è possibile sostituire un valore al posto di uno dei due attributi.
  - se nell'esecuzione del join si visita prima impiegati allora per l'accesso a dipartimenti

    valore = dipartimenti.dno SI

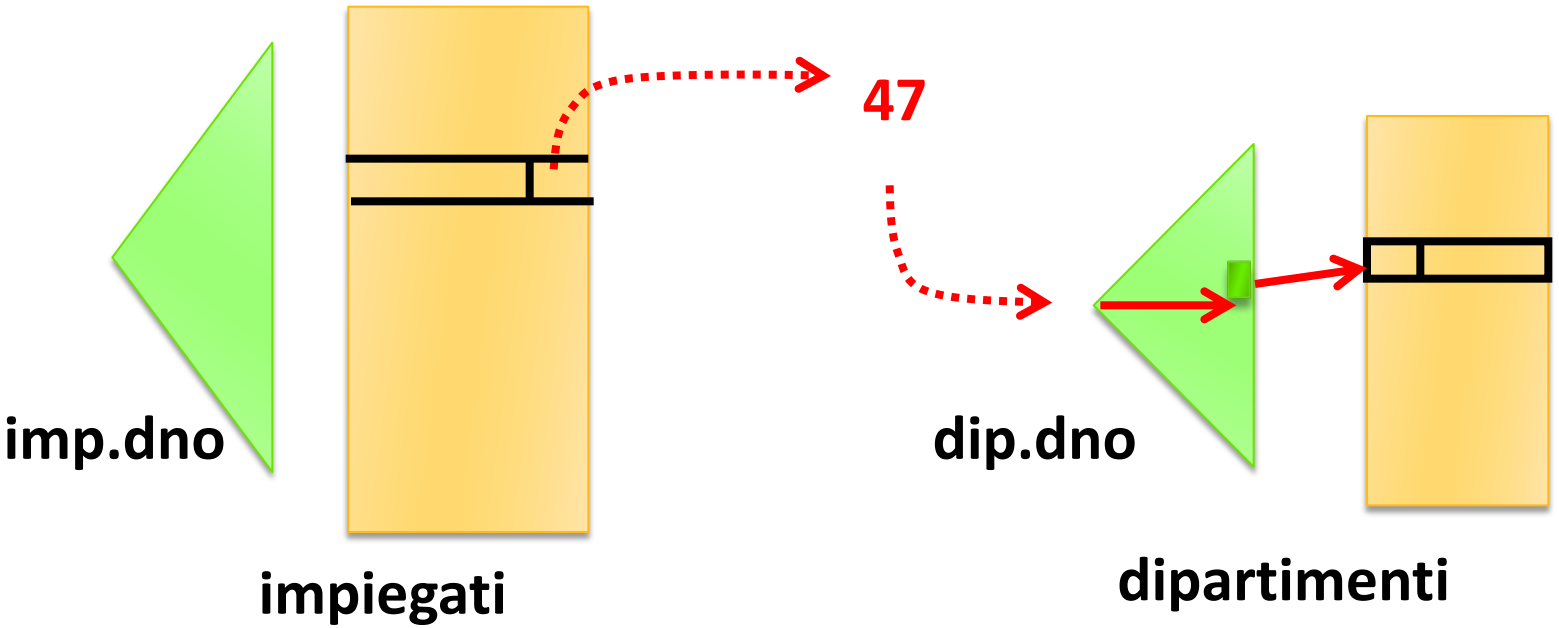
    (il valore è quello trovato in una delle tuple di impiegati)

    per impiegati sarebbe: impiegati.dno = ? NO

Uso degli indici in caso di join

nella sequenza impiegati ⟹ dipartimenti

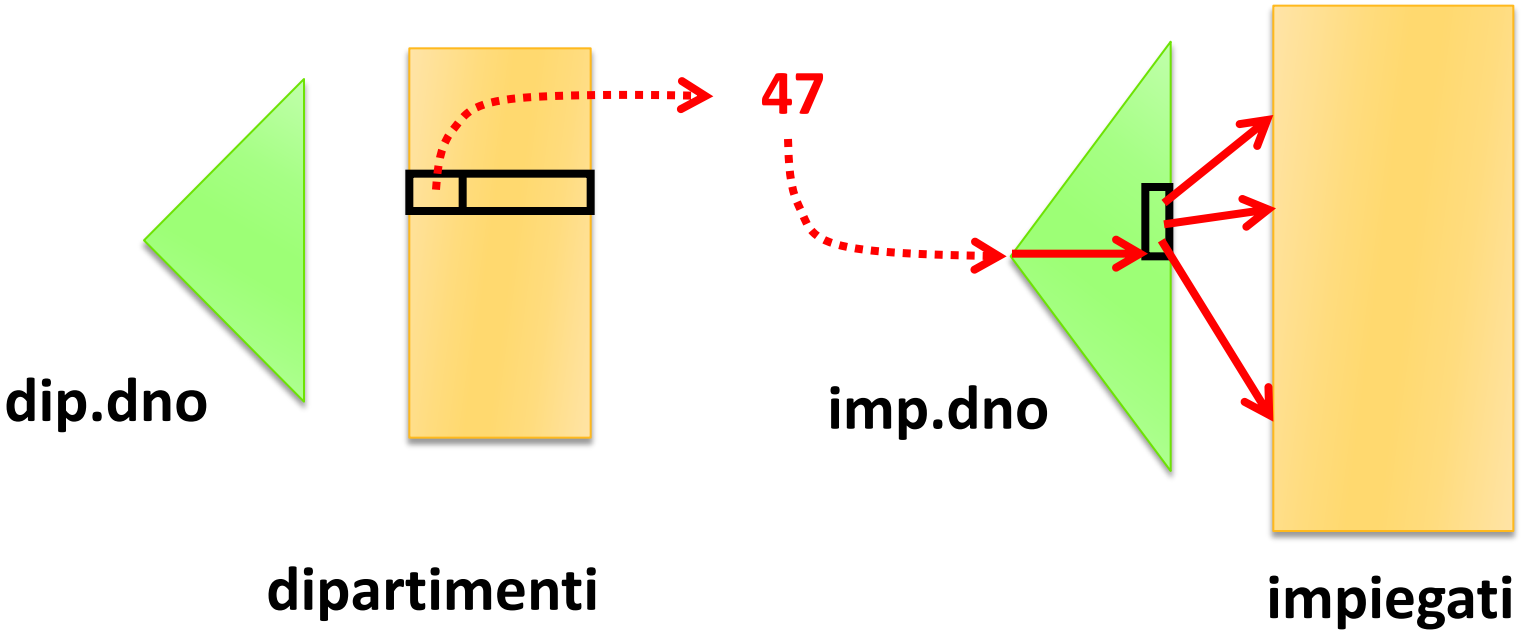
imp.dno = ? e dip.dno = valore



Uso degli indici in caso di join

nella sequenza dipartimenti ⟹ **impiegati**

dip.dno = ? **e** **imp.dno = valore**



Costo di join

In generale, considerando il metodo nested loop:

Costo di JOIN: $C_{join} = C_{R1} + E_{R1} \times C_{R2}$

(dove R1 è la relazione esterna, R2 quella interna)

Esercizio: costo di accesso (join)

Esercizio, con le relazioni:

prodotti (pn, pnome, tipo, pj),

NT = 2000, NB= 400

progetti (pj, nome, dno),

NT = 200, NB= 100

gli indici su:

dno con $NK_{dno} = 20$, $NF_{dno} = 5$

tipo con $NK_{tipo} = 100$, $NF_{tipo} = 10$

prodotti.pj con $NK_{prod.pj} = 200$, $NF_{prod.pj} = 15$

progetti.pj con $NK_{prog.pj} = 200$, $NF_{prog.pj} = 10$

Esercizio: costo di accesso (join)

Ed il JOIN :

SELECT pj, nome

FROM prodotti, progetti

WHERE dno = 136

AND tipo = 'AA'

AND prodotti.pj = progetti.pj

calcoliamo il costo nel caso

prodotti ⇒ progetti e

nessun indice è clustered

Esercizio: costo di accesso (join)

prodotti ⇒ progetti

accesso a prodotti:

C_{seq} = 400,

F_{tipo} = 1 / 100, E_{tipo} = 2000/100 = 20

$C_{tipo} = \lceil 10 / 100 \rceil + \lceil 2000 / 100 \rceil = 21$

l'indice su pj non si può usare su prodotti

accesso a progetti:

C_{seq} = 100, F_{dno} = 1 / 20, E_{dno} = 200 / 20 = 10

$C_{dno} = \lceil 5 / 20 \rceil + 10 = 11$ (indice su dno)

$C_{progetti.pj}$ = 1 + 1 = 2 (indice su pj)

Esercizio: costo di accesso (join)

con l'accesso a prodotti si ottengono

E_{tipo} = 20 tuple che soddisfano tipo = 'AA'

quindi per 20 volte si fa l'accesso a progetti

per trovare le tuple che soddisfino dno = 136 ed il predicato di join prodotti.pj = progetti.pj

in conclusione:

$C_{join} = C_{prodotti} + E \times C_{progetti}$

$C_{join} = C_{tipo} + E \times C_{progetti.pj} = 21 + 20 \times 2 = 61$

$C_{seq\text{-}join} = C_{seq\text{-}prodotti} + E \times C_{seq\text{-}progetti} =$

$= 400 + 20 \times 100 = 2400$

Esercizio: costo di accesso (join)

progetti ⇒ prodotti

accesso a progetti:

$E_{dno} = 200 / 20 = 10$

$C_{dno} = 11$ **(già calcolato)**

l'indice su pj non si può usare su progetti

accesso a prodotti:

$C_{tipo} = 21$ **(già calcolato)**

$C_{prodotti.pj} = \lceil 15 / 200 \rceil + 10 = 11$ **(indice su pj)**

$C_{join} = C_{dno} + E_{dno} \times C_{prodotti.pj} =$

$11 + 10 \times 11 = 121$

Conclusione: è preferibile la sequenza prodotti ⇒ progetti

Costo di accesso

- Gli ottimizzatori generalmente scartano a priori le sequenze che contengono prodotti cartesiani perché potrebbero dare luogo a E intermedie troppo elevate, anche se questo non sempre è vero.
- Gli ottimizzatori valutano tutte le sequenze dove due relazioni consecutive sono collegate da predicati di join e scelgono la migliore.
- Esistono molti altri algoritmi di join oltre al nested loops che comunque è uno dei più adoperati e risulta in linea di massima molto efficace in presenza degli indici opportuni.

Costo di accesso

Conclusioni:

- **senza indici il DBMS relazionale ha prestazioni scadenti**
- **l'ordinamento e gli indici migliorano di molto le prestazioni**
- **un eccesso di "indicizzazione" è nocivo per DB con elevato carico di modifiche**
- **bisogna trovare un compromesso sensato**

Costo di accesso

Statement SQL utili:

EXPLAIN <query>

descrive il piano d'accesso scelto dall'ottimizzatore

RUNSTATS (update statistics)
aggiorna le statistiche nei cataloghi

Riferimenti Bibliografici

Fabio Grandi:

“Esercizi di Basi di Dati”.

Progetto Leonardo, Esculapio, 2 edizione, 2015

contenuti:

Esercizi risolti e ampiamente commentati di:

- interrogazione e manipolazione SQL
- ottimizzazione piani di accesso

Formulario costi di accesso

Costo di accesso

- $E = \lceil \prod_i F_i * NT \rceil$

	F:	default
A = val	$1 / NK_A$	<= 1/10
A IN (val_1, val_2 ... val_n) (OR)	n / NK_A	<= 1/2
A > val	$(max_A\text{-}val) / (max_A\text{-}min_A)$	<= 1/3
A < val	$(val\text{-}min_A) / (max_A\text{-}min_A)$	<= 1/3
A BETWEEN val_1 AND val_2	$(val_2\text{-}val_1) / (max_A\text{-}min_A)$	<= 1/4
A = NOT val	$1\text{-}1 / NK_A$	
A = B	$1 / max(NK_A, NK_B)$	
pred1 OR pred2	$F_{pred1} + F_{pred2} - F_{pred1}F_{pred2}$	

Costo di accesso

- C_A = 1 foglia + 1 blocco = 2 E=1

  indice clustered / unclustered

- $C_A = \lceil F * NF \rceil + \lceil F * NB \rceil$ E>1

  indice clustered

- $C_A = \lceil F * NF \rceil + \lceil F * NT \rceil$ E>1

  indice unclustered

- $C_A = \Sigma_k \lceil F_k * NF_k \rceil + \lceil \Pi_k F_k * NT \rceil$ E>1

  più indici unclustered (unione TID) (!)

- $C_{Join}(R_1,R_2) = C_A(R_{ext}) + E * C_A(R_{int})$ (Nested loops join)