

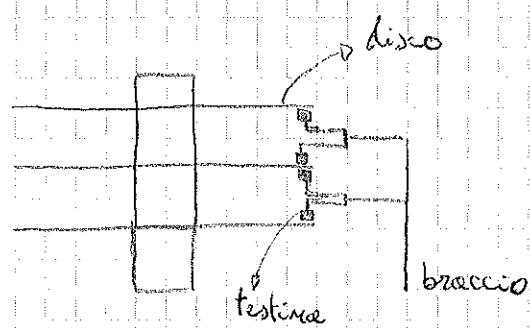
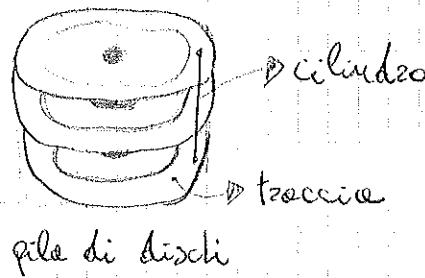
# Architettura delle basi di dati

- Principali differenze tra dischi magnetici ed ottici:

- |               |  |
|---------------|--|
| <u>OTTICI</u> | <ul style="list-style-type: none"><li>• CD ~ 700 Mb</li><li>• DVD ~ 15 Gb</li><li>• Blu-ray (HD DVD) ~ 50 Gb</li><li>• HDV ~ 500 Gb (uso di tecniche holografiche)</li></ul> |
|---------------|--|

- |                  |  |
|------------------|--|
| <u>MAGNETICI</u> | <ul style="list-style-type: none"><li>• creati per applicazioni di gestione dati, in sostituzione dei nastri magnetici</li><li>• e addele perforate</li><li>• 1 Tb</li><li>• dischi e teste mobili</li></ul> |
|------------------|--|

Una memoria a dischi consiste di due pile di dischi metallici, magnetizzati su entrambe le superfici, che ruota a velocità costante. L'accesso alle info avviene tramite un certo numero di testine di I/O poste sull'esterno di un braccio meccanico che si sposta radialmente rispetto alle superfici magnetiche del disco:



## Carakteristiche:

• Una traccia è la parte del disco che può essere utilizzata senza spostare le testine, essa è organizzata in settori di dimensioni fisse dipendenti dal tipo di disco (tutte le tracce, le tracce, sono di diam. uguale (hanno più o meno settori) anche se le loro capacità di memorizzazione sono diverse (comincia la densità di memorizzazione))

• Una traccia è logicamente suddivisa in blocchi (block logici), essi rappresentano l'unità di dati trasferibili fra le memorie permanente e quelle temporanea; la dim. dei blocchi è un multiplo di quella dei settori ( $2\text{ Kb} \div 8\text{ Kb}$ )

• l'insieme delle tracce sulle superfici dei dischi che possono essere usate senza spostare le testine prende il nome di cilindro. N.B. solo una testina allo stesso può essere utilizzata per trasferire i dati.

### Tempi

• Il tempo di trasferimento dei dati dalla memoria secondaria (permanente, cioè i dischi) alla memoria principale (temporanea) è influenzato da molti fattori: rotazione dei dischi per estrarre il cilindro desiderato, posizionamento delle testine sui blocchi, trasferimento.

• In dettaglio abbiamo:

① tempo di posizionamento (seek time),  $t_s$  è il tempo necessario per posizionare le testine sul cilindro desiderato; questo è il tempo dominante ( $5 \div 15$  msec) nell'operazione di trasferimento.  
Notiamo che se i dati che ci interessano si trovano tutti sullo stesso cilindro il costo di seek è praticamente zero; per questo motivo i dati correlati andrebbero sempre messi sullo stesso cilindro o in cilindri contigui.

② tempo di latenza (rotational latency),  $t_R$  è il tempo di attesa dovuto alla rotazione delle superfici per portare un blocco sotto le testine (la quale è già posizionata sulla traccia desiderata, vedi precedente). Questo parametro quindi dipende dalla velocità di rotazione dei dischi ( $4200 \div 15000$  rpm)

③ tempo di trasferimento di un blocco (block transfer time),  $t_B$  è il tempo impiegato per trasferire un blocco in memoria temporanea una volta posizionata la testina all'inizio del blocco. Essenzialmente dipende dal tipo di bus adottato ( $\approx 100 \div 300$  Mb/sec)

→ quindi il tempo medio per trasferire un blocco in memoria temporanea è:

$$(t_s + t_R + t_B)$$

→ mentre il tempo medio per trasferire  $K$  blocchi contigui (cilindro) è:

$$(t_s + t_R + K \cdot t_B)$$

## Architettura RAID

È composta da un array (battery) di dischi e teste mobili, e un'architettura che garantisce affidabilità, correttezza e velocità rispetto il semplice utilizzo di più dischi insieme; un suo aspetto chiave è la ridondanza:

• Mirroring: si hanno due dischi con lo stesso contenuto; le scritture sono contemporanee ma danno migliori solo le prestazioni in lettura.

• Codici di parità: un'architettura RAID con codici di parità e chiamata anche architettura STRIPPING o RAID di Cicllo 3:



i settori sono distribuiti orizzontalmente, grazie alla tecnica dello STRIPPING, in modo da favorire l'accesso parallelo anche allo stesso file.



codici di parità, permettono di recuperare i dati (1 errore alla volta)

## Schedulazione del disco

Assumendo che ci sia una coda di richieste di settori del disco, vogliamo riorganizzare tale coda per minimizzare il tempo di seek (fornendo alle testine il meno possibile e quindi maximizzando il throughput). Mantenendo egualmente il tempo di risposta alle richieste.

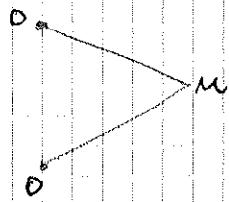
### Strategie:

① FCFS (First Come First Served ~ FIFO): non garantisce la minimizzazione del tempo di seek.

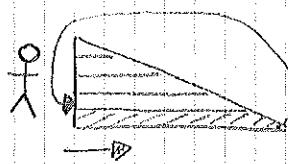
② SSTF (Short Seek Time First): è una strategia greedy la quale quindi non garantisce l'egualità dei tempi di risposta. Vengono fornite le tracce sulla parte centrale del disco perché sono equidistanti dagli estremi dello stesso.

• i tempi di risposta relativi a richieste differenti, sono eguali.

- ④ SCAN/C-SCAN: la testina viaggia dalla traccia 0 → n e poi da n → 0 servendo le richieste mai meno che le incontra: tutti verranno serviti in non più di due scan del disco.



- ⑤ C-SCAN: variante dello SCAN; dopo essere arrivati alla traccia n si ritorna alla traccia 0 senza servire nessuno → si dimostra che questo rende più equi i tempi di risposta.



Come se si dovesse riempire una stanza eliminando lo spazio

- ⑥ LOOK / C-LOOK: ottimizzazioni di SCAN/C-SCAN; non si arriva fino alla traccia n ma solo fino all'ultima richiesta

## Sistema Operativo

Fornisce le variazioni di file come stringhe di byte sequenziali; in realtà quello che si ha è una percezione virtuale del file esistente nel disco fisso. sequenzialmente (dipende dalla politica di allocazione), in ogni caso l'utente deve poter scorrere il file dall'inizio o da una posizione specifica.

## Tecniche di allocazione dello spazio

- ① Allocazione a blocchi (512 byte ad esempio)

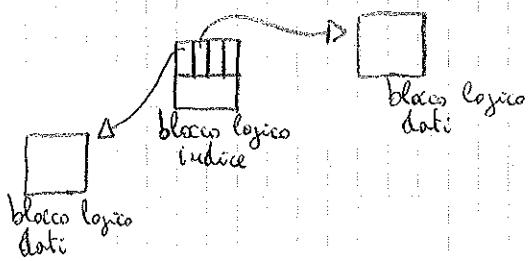
- ② Allocazione contigua: i settori corrispondenti ai pezzi del file sono messi jettivamente in modo contiguo, in questo modo si paga di più il tempo di lettura mentre quello di seek è minimo sia per accesso sequenziale che random. Svantaggio: dal momento che i file sono dinamici può capitare che l'estensione di uno di questi non sia più possibile a causa della mancanza di settori liberi e contigui da allocazioni; in questo caso bisogna cercare settori liberi in altre parti del disco ed allocare al file ma ciò comporta una gran moleta del disco che può diventare tale da impedire l'utilizzo dello stesso → si risolve comprattando lo spazio (op. molto costosa)

④ Allocazione linkata: l'intero disco è suddiviso in blocchi, ogni blocco del file punta al successivo. Svantaggi: onde se non c'è spazio di spazio si paga molto il tempo di accesso (\*) perché non si può controllare come vengono occupati i settori inoltre se si vuole un accesso random, questo non può essere soddisfatto perché onde se si vuole solo il 3° blocco, non sappiamo dove si trova (in quale settore), quindi si deve partire sempre dall'inizio:



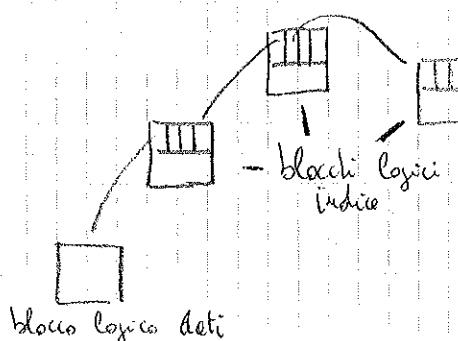
⑤ Allocazione indexed: sono richiesti sempre almeno due accessi al disco, uno all'indice e uno al blocco dati ma si può fare in modo che il blocco indice stia interamente in memoria in modo da avere sempre disp.

Allocazione indexed classica: si mantiene un blocco indice che mantiene i puntatori ai primi  $n$  blocchi del file



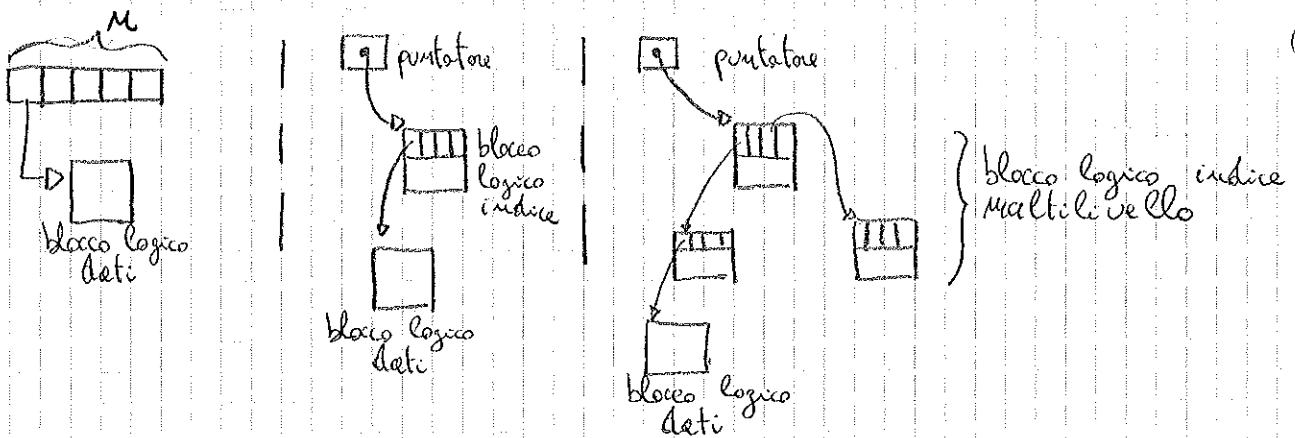
- la lettura sequenziale può essere molto costosa in termini di accessi
- la lettura random è immediata visto che basta leggere l'indirizzo del blocco indice e accedere al blocco dati
- quando si fa un overflow l'ultimo puntatore del blocco indice viene fatto puntare ad un nuovo blocco indice.

- Allocazione indexed geografica: cresce il num. di accessi necessari per ovviare ai dati ma per file grandi è comunque utile.



⑥ il tempo di accesso è il tempo medio necessario per localizzare un blocco in memoria permanente (8÷15 msec)

## Allocazione ibrida:



## Esempio FILE DESCRIPTOR di UNIX

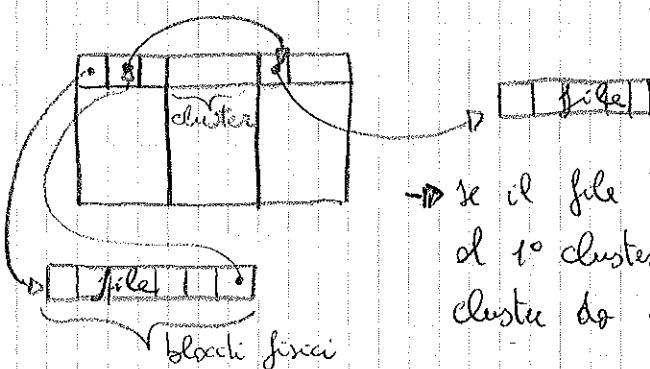
si hanno:

- 12 puntatori diretti ai blocchi logici del file
- 1 puntatore ad un index block
- 1 " " " " " " " " " " " " due livelli
- 1 " " " " " " " " " " " " tre livelli

Per i primi 12 blocchi si paga un accesso solo, se si accede oltre si paga di più

## Esempio FAT di WINDOWS

assomiglia ad un index block ma è diverso per disco quindi le tavole crescono con il device



- Se il file deve essere esteso si aggiunge un puntatore al 1° cluster allocato al file facendolo puntare al 2° cluster da aggiungergli
- Una lettura sequenziale del file corrisponde ad una lettura sequenziale sui cluster: lo testina si sposta solo quando si cambia cluster.

- Consideriamo solo i costi di I/O e non quelli di CPU, definendo quindi costo di accesso ad un file il numero di accessi e pagine per eseguire una certa operazione.
- Una pagina è un'unità di accesso elementare per il DB (corrisponde ai block logici) e va dai 512 byte ai 16 Kb; esse vengono mappate sui blocchi fisici del device come accade per i blocchi logici del S.O.
- I tempi di accesso ad una pagina dipendono sia i tempi fisici (rotazione dei dischi, etc.), dal meccanismo di scheduling e dalla tecnica di allocazione dello spazio.

## Algebra Relazionale

È un linguaggio non procedurale (non dice come fare le cose) il quale possiede degli operatori che, applicati su relazioni, producono altre relazioni.

- Selezione:  $\sigma_c(R)$  restituisce una relazione dello stesso tipo di R i cui elementi sono le copie degli elementi di R che soddisfano il predicato c:  

$$\sigma_c(R) = \{ t \mid t \in R \wedge c(t) \}$$
- Proiezione:  $\Pi_L(R)$  dove  $L = \{A_1, A_2, \dots, A_m\}$  è un insieme di attributi. Restituisce una relazione di tipo  $A_1:T_1, A_2:T_2, \dots, A_m:T_m$  in cui gli elementi sono le copie degli elementi di R proiettati sugli attributi  $A_1, A_2, \dots, A_m$ . Perché questa relazione contiene meno attributi di R è possibile che si presentino dei duplicati i quali vengono eliminati.  

$$\Pi_{A_1, A_2, \dots, A_m}(R) = \{ t[A_1, A_2, A_3, A_4, \dots, A_m] \mid t \in R \}$$

- Prodotto Cartesiano:  $R \times S$  restituisce una relazione del tipo  $\{A_1:T_1, A_2:T_2, \dots, A_l:T_l, A_{l+1}:T_{l+1}, \dots, A_{l+m}:T_{l+m}\}$  con elementi ottenuti concatenando ogni tupla di R con tutte quelle di S.

$$R \times S = \{ t u \mid t \in R \wedge u \in S \}$$

$$\text{con } R = \{A_1:T_1, A_2:T_2, \dots, A_l:T_l\}$$

$$S = \{A_{l+1}:T_{l+1}, \dots, A_{l+m}:T_{l+m}\}$$

Q30:  $R \setminus S$  il quale  $\setminus$  non è un operatore primitivo, infatti si può ottenere da  $S^c (R \times S)$  inizialmente

Considereremo la risoluzione di questi operatori utilizzando solo accessi sequenziali, senza cioè strutture di velocizzazione come gli indici, e senza considerare né i costi di CPU, né i costi per produrre il risultato.  
(vedo 20 parte del corso per struttura degli indici)

### $S^c(R)$

```
for (i=0; i<|R|; i++)
{
    Tuple t = read (R, i)
    if (t. cond(c))
        append (T, t)
}
```

↳ rel. temporanea

costo accesso a tuple =  $|R|$ , costo scansione  
costo accesso a pagina =  $P_R$ , numero di pagine delle rel. R che di solito è  $\ll |R|$

### $\Pi_L(R)$

senza rimozione dei duplicati:

```
for (i=0; i<|R|; i++)
{
    Tuple t = read (R, i)
    t. eliminateDuplicates (L)
    append (T, t)
}
```

↳ rel. temporanea

costo accesso a tuple =  $|R|$   
costo accesso a pagina =  $P_R$

Se si vogliono eliminare i duplicati si devono eseguire le operazioni aggiuntive di sort e di rimozione; i costi aumentano ~~perché~~ a causa del costo molto elevato del sort, per questo motivo in SQL l'eliminazione dei duplicati deve essere richiesta esplicitamente con  DISTINCT.

costo totale

$P_R + \text{costo sort} + \sim P_R$

↳ tempo per scandire la relazione ordinata ed eliminare i duplicati.

N.B.

i duplicati vanno tolti mentre si fa l'ordinamento e non dopo

## External Sort Merge

È un algoritmo di ordinamento che ha due complessità dipendente dal numero di pagine e non dal numero di tuple; correttistiche:

- ① cerca di ottenere il massimo di località, lavorando solo su cose che stanno in memoria
- ② la sua complessità è  $\lceil \log_{B-1} p \rceil$  dove  $p$  è il num. di pagine della tabella e  $B$  è la dimensione di un buffer in mem. centrale (temporanea) composto da frames in cui ogni frame può contenere una pagina. Il costi di molti passaggi della complessità dipende dalla dim del buffer, se abbiamo un file  $< B-1$  pagine oppure molto meno, la disposizione questa tende ad essere lineare.
- ③ abbiamo bisogno di un buffer di almeno 3 frames

### Passi dell'algoritmo

siamo  $p$  le pagine di un file non ordinato,  $B$  la dim. del buffer

#### 1° FASE

- leggo le prime  $B-1$  pagine del file e le metto sul buffer, il buffer viene ordinato in memoria senza ulteriori costi di I/O applicando un qualche metodo (questo ordinamento è obbligatorio visto la sottosezione di merge richiede che i sottofile siano ordinati)

#### ~~Passo alla $B^{\text{th}}$ pagina~~

- si applica la sottosezione di merge ottenendo un unico sottofile ordinato
- si passa alle  $B-1$  pagine successive ottenendo un altro sottofile ordinato; in questo modo si ottengono esattamente  $\lceil \frac{p}{B-1} \rceil$  file ordinati

#### 2° FASE

- allo stesso modo si processano i singoli sottofile creati dalla fase precedente, si ottengono  $\lceil \frac{p}{(B-1)^k} \rceil$  sottofile ordinati.

#### K° FASE

- si ottengono  $\lceil \frac{p}{(B-1)^k} \rceil$  e ci si ferma se  $\frac{p}{(B-1)^k} = 1$ , cioè se si ottiene un solo file ordinato; questo significa che  $(B-1)^k = p$  e che  $K = \lceil \log_{B-1} p \rceil$

Poiché ogni file costa  $ZP$  ( $p$  per leggere e  $p$  per scrivere) ed abbiamo  $K$  file si ha che il costo totale dell'external sort merge è  $ZP \cdot K$  cioè

$$ZP \lceil \lg_{B-1} p \rceil$$

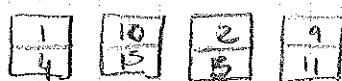
nel caso peggiore del tipo  $O(n \lg n)$

$B-1 = 2$  (buffer a 3 frames)

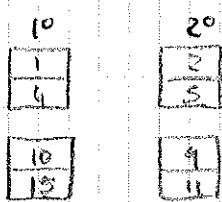
il costo è  $ZP \log_2 p$  che è

### Esempio SOTTOFASE DI MERGE

- consideriamo un file di 4 pagine "scamposto" in due file di due pagine ciascuno ordinati al loro interno ma non tra di loro:

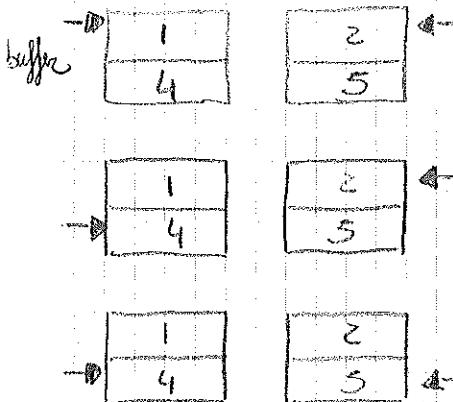


unico file di 4 pag.



due sottofile di due pagine

- buffer di 3 frames



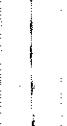
=> output



=> output



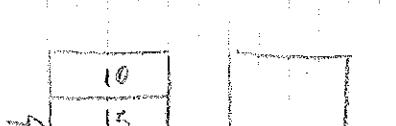
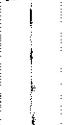
=> output



=> output



=> output



=> output



unico file ordinato

# di accessi = dimensione dei due file in pagine,  $p$  per leggerle e  $p$  per scrivere,  
TOT =  $ZP$

## Esempio Completo

file di 4 pagine non ordinato

buffer di 3 frames

il file viene considerato a blocchi di B-1 (cioè 2) pagine, queste vengono caricate nel buffer, ordinate e messe:

buffer

3	5	
2	10	

ordinato in mem. senza nessun costo aggiuntivo

buffer

1	4	
7	9	

vengono caricati gli altri B-1 blocchi nel buffer

la sotto fase di merge produce il primo sotto file di due pag. ordinato

2	3
5	10

nessun ordinamento interno è necessario; la sotto fase di merge produce il secondo file di due pagine ordinato

1	4
7	9

poiché i blocchi del file originale sono finiti si passa alla 2° fase. **(NB)** con un buffer di 3 frames la 1° fase produce B-1 file (cioè 2) quindi la 2° fase ne produrrà uno soltanto.

considerando i sotto file ottenuti precedentemente si riapplica la sotto fase di merge:

2	1
3	4

2	3
3	4

1	4
7	9

1	2
3	4

3	7
9	10

si ottiene il file ordinato  $\Rightarrow$

etc.

## ⑤ R $\bowtie$ S

Esempio 3. Scegliere dei metodi che risolvono l'equijoin e sono:

- ①- Metodi basati sul prodotto cartesiano: nested-loop, nested-block, nested-scans
- ②- Metodi basati sull'ordinamento: Merging-scans (Sort-Merge)
- ③- Metodi basati sul partizionamento: recursive hash partitioned joins (Segmentation)

### 1- Nested-loop

In prima approssimazione si può ottenere una scansione per tuple: per ogni tuple delle 1° rel. (quella esterna) si scandiscono tutte le tuple della 2° (quella interna) corrispondente, quindi si applica la condizione di filtro per la selezione che, in questo caso trattandosi di equijoin, corrisponde a quelli legati per le tuple sull'attributo di join.

Questa implementazione rappresenta esattamente la corrispondenza  $R \bowtie S \equiv \pi_C(R \times S)$

```

for (i=0; i<|R|; i++)
{
    Tuple t = read(R, i)
    for (j=0; j<|S|; j++)
    {
        Tuple t' = read(S, j)
        Tuple tc = t + t'
        if (tc.condizione (C))
            Append tc
    }
}

```

si lavora una tuple alla volta  
osservando che ogni accesso costi un I/O  
quindi il costo è:

$$\rightarrow \text{COSTO} = K_R + K_R \cdot K_S$$

dove

$$K_R = |R|, \quad K_S = |S|$$

$$+ K_R \cdot K_S = \text{per ogni tuple di } R, \text{ leggo quelle di } S$$

$$+ K_R = \text{tuple di } R$$

L'equijoin quindi può essere visto in due modi: il primo è applicando il nested-loop quindi calcolando prodotto cartesiano e selezione, il secondo andando a considerare, nel ciclo interno, solo le tuple con lo stesso valore su join (senza la necessità di calcolare l'intero prodotto cartesiano).

Una variante del nested-loop prevede una scansione per pagine; la complessità è ancora più elevata, il vantaggio è che non si fanno I/O per tuple sulle stesse pagine  $\rightarrow \text{COSTO} = P_R + K_R \cdot P_S$  dove  $P_R$  e  $P_S$  rappre-

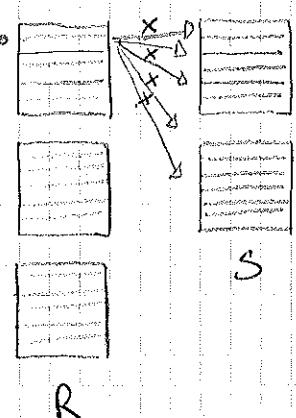
il numero di pagine per la relazione R ed S rispettivamente.

N.B. In generale combinare le relazioni interne con quelle esterne porta a risultati di costo diversi

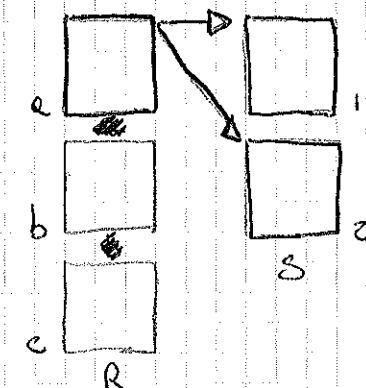
~~È stato scoperto che i costi sono molto simili~~

## 1- Nested-block

Vengono memorizzate in memoria 2 pagine, una per relazione; il prodotto cartesiano viene fatto non più a tuple ma a pagine:



nested loops (Nl)



nested-block

```
for (i=0; i<Pr; i++)
```

```
{
```

```
    Page p = Read (R, i)
```

```
    for (j=0; j<Ps; j++)
```

```
{
```

```
    Page p' = Read (S, j)
```

```
    process (p, p', c)
```

```
}
```

```
process (p, p', c)
```

```
{
```

```
    for Vt ∈ P
```

```
{
```

```
        for Vt' ∈ P'
```

```
{
```

```
            tuple tc = t + t'
```

```
            if (Condizione (c))
```

```
                Append Write (tc)
```

```
}
```

```
}
```

```
}
```

Nel momento in cui leggiamo una pagina di R e iniziamo a leggere quelle di S, facciamo un prodotto cartesiano con tutte le tuple di S che si trovano su quelle pagine, direttamente, perché p e p' si trovano già in memoria. La soluzione del costo è sempre quadratica ma in termini di accessi è → **COSTO** =  $P_R + P_R \cdot P_S$  cioè migliore delle precedenti

## R-Nested-blocks (metodo di RIM)

È un metodo che sfrutta tutta la memoria a disposizione. Considerando come buffer un B-frame.

Spizziamo logicamente la rel. S in  $\lceil \frac{Ps}{B-1} \rceil$  segmenti di  $B-1$  pagine ciascuno (è solo un modo di organizzare l'accesso ad S), otteniamo  $\lceil \frac{Ps}{B-1} \rceil$  segmenti totali. La rel. S viene quindi letta a blocchi di  $B-1$  pagine alla volta:

$B-1$  pagine di S



ultimo frame del buffer, usato per fare una scansione completa della rel. R; ad ogni passo si fa il join tra le  $B-1$  pagine di S e quelle di R

### pseudo-codice

for  $i$  segn  $\in S$

    l leggi segn (corvo le pagine nel buffer)  
    utilizza il frame rimanente per scordare R  
    effettua il join a pagine

g

    si coricano le prime  $B-1$  pagine nel buffer

    si corica nell'ultimo frame la prima pagina di R

    si effettua il join a pagine come in nested-block tra le pagine di R e tutte le  $B-1$  pagine di S (per sempre a copiare); il waiting rispetto al nested-block è che in questo caso si hanno non  $\lceil \frac{Ps}{B-1} \rceil$  ma  $B$  pagine dorate niente in memoria

    mantenendo le  $B-1$  pagine di S si corica la seconda pagina di R nell'ultimo frame del buffer e si ripete il join. Questo processo viene ripetuto fino ad esaurire tutte le pagine di R; dopodiché si coricano oltre  $B-1$  pagine di S e si ricomincia di nuovo.

$$\Rightarrow \text{COSTO} = Ps + \lceil \frac{Ps}{B-1} \rceil \cdot Pr$$

    L'intera rel R viene scordata per ogni segmento di S

Inoltre notiamo che maggiore è la memoria disponibile (dim. del buffer), minore è il costo:

•  $B-1 = 1 \Rightarrow$  nested blocks

•  $B-1 = 2$  i tempi si dimezzano rispetto nested block

•  $P_S \leq B-1$  costo  $\approx P_S + P_R$

→ il nested scans è il miglior metodo per calcolare il prodotto cartesiano.

### Esempio

R:

$$P_R = 200$$

dim. pagina = 4 Kb (naturalmente vale per entrambe le relazioni!)

$$K_R = 4000$$

dim. relazione = 800 Kb (200 pagine  $\times$  4 Kb)

S:

$$P_S = 100$$

$$K_S = 1000$$

dim. relazione = 400 Kb (100 pagine  $\times$  4 Kb)

Esempio. Tempi di elaborazione del join R<sub>A,B</sub>S oppure S<sub>B,R</sub>

- metodo NL tuple 
$$\begin{aligned} NL &= K_R + K_R \cdot K_S = 4.004.000 \\ NL &= K_S + K_S \cdot K_R = 4.001.000 \end{aligned} \quad \left. \right\} 11 \text{ ore}$$

- metodo NL pagina 
$$\begin{aligned} NL &= P_R + K_R \cdot P_S = 400 \cdot 200 \approx 66 \text{ minuti} \\ NL &= P_S + K_S \cdot P_R = 800 \cdot 100 \approx 33 \text{ minuti} \end{aligned}$$

si nota come nel nested-loop a pagina concorre soltanto esterna la relazione che occupa di meno

- metodo NB:  $P_R + P_R \cdot P_S = 20.200 \approx 3 \text{ minuti}, 20 \text{ recordi}$

- metodo NS:  $P_R + P_S = 300 \approx 3 \text{ secondi}$

in questo caso è naturale che sia  $P_S \leq B-1$  poiché è sufficiente un buffer di 101 pagine (404 Kb) per far stare le relazioni S interamente in memoria (100 pagine) + una pagina per scandire R

## Z - Merging - joins (chiamato anche Sort-Merge)

Si considerano due relazioni ordinate allo stesso modo nell'attributo di join; il join viene interpretato come un'operazione di ricerca nello joiner relazione interna delle tuple con lo stesso valore di join rispetto le tuple della relazione esterna, piuttosto che come un'operazione di prodotto cartesiano e successiva relazione (nested-loop).

Probl. Il costo dell'algoritmo è lineare solo se entrambe le relazioni sono ordinate e senza valori duplicati; se è necessario ordinare si paga un costo aggiuntivo di  $\text{zp} \lceil \log_{\text{base}} p \rceil$  per relazione (la complessità quindi diventa nell'ordine di  $n \log n$ ), se ci sono dei duplicati il costo varia tra quello lineare e quello per il nested-loop.

Esempio, relazioni ordinate senza duplicati

R	S
1	2
2	7
5	9
7	10
9	

le tuple vengono confrontate e inserite in output solo se i loro valori sono uguali; essendo le rel. ordinate il valore 1 sicuramente non porta a join (xcl si sarebbe trovato subito un match); si avvia il puntatore al minor valore tra i due.

R	S
1	2
2	7
5	9
7	10
9	

output: 2 ~ in questo caso inseriamo entrambi i puntatori xcl assicuriamo che non ci siano duplicati (vedi altro esempio per caso con i valori duplicati)

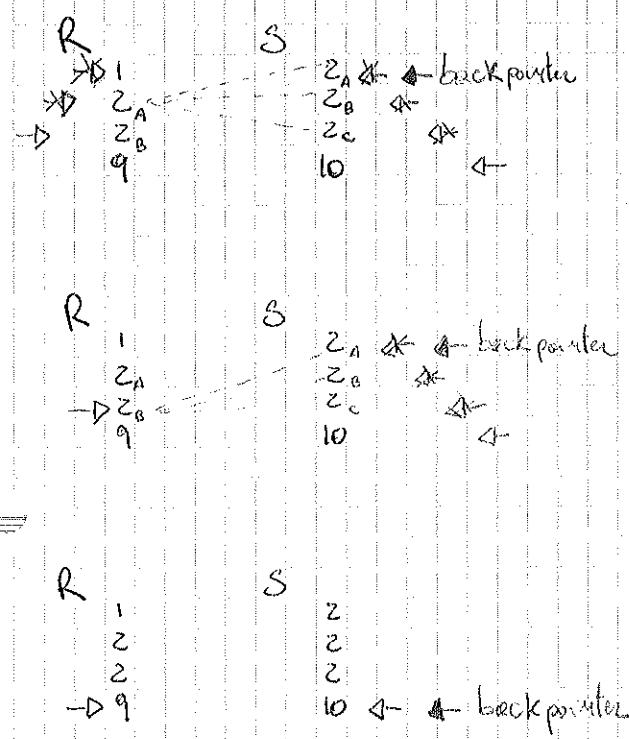
R	S
1	2
2	7
5	9
7	10
9	

output 2, 7, 9

COSTO:  $P_R + P_S$  notiamo che in questo caso S non viene nemmeno esaminata del tutto

## Esempio, relazioni ordinate con duplicati

Si mantengono 3 puntatori, due alle tuple come il caso precedente ed uno per fare Backtracking



output: Z<sub>AA</sub>, Z<sub>AB</sub>, Z<sub>AC</sub>  
In questo momento  
il Z<sub>A</sub> di R non può più essere matching  
con elementi di S, allora si avanza il  
puntatore di R; poiché il valore (Z<sub>B</sub>) è  
uguale al precedente il ~~puntatore~~ del puntatore  
di S viene ripristinato grazie al  
back pointer.

output: Z<sub>AA</sub>, Z<sub>AB</sub>, Z<sub>AC</sub>, Z<sub>BA</sub>, Z<sub>BB</sub>, Z<sub>BC</sub>; ormai  
una volta il puntatore di R viene  
avanzato solo in S non ci sono più  
valori da matching; poiché il nuovo valore  
(9) di R è diverso dal precedente, il  
back pointer di S si sposta all'attuale  
puntatore della stessa.

Il merging-sous quindi, almeno nel caso dei duplicati, si comporta come il nested-loop infatti, per ogni tuple di R, si ricorda la sottoinsieme di S.

Nel caso peggiore in cui tutti i dolori sono uguali il costo diventa proprio quello del nested-loop a pagina; quindi possiamo affermare che;

$$PR + PS \leq \text{COSTO} \leq PR + KR \cdot PS$$

e meno di ordinamenti necessari per le relazioni, nell'isolto caso il costo <sup>non</sup> aumenterebbe oltre la stessa quadraticità data da  $PR + KR \cdot PS$

- 3- Prima di trarre le 3° famiglie di metodi per risolvere l'equi-join occorre esaminare la gestione del buffer.



## Gestione del Buffer

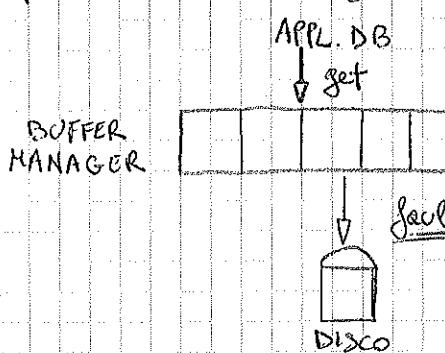
integrazione con articolo: "Buffer Management in Relational Database Systems"

ACM TODS 11:4 DIC. 86

Sacco, Schkolnick

soltando i paragrafi [3.3 ÷ 3.6], 4.2

Che buffer è un'area di memoria centrale costituita da B pagine, ciascuna dei quali può contenere una pagina; esso viene gestito come una memoria virtuale:



Le pagine richieste dall'applicazione non viene trovata nel buffer, essa quindi deve letto dal disco e caricata → si dovrà scegliere una pagina da eliminare.

Politica d'ingresso: (di una o più pagine nel buffer) tipicamente è la Demanding Page cioè se richiesta dell'utente ma può anche essere la Prefetching cioè quello del caricamento in avanti.

Politica d'uscita: (di una o più pagine del buffer) REPLACEMENT ad esempio, cioè una pagina essa quando deve essere sostituita da un'altra ma se ne possono scambiare scagliare un'altra; da questa politica dipende quella di scrittura sul disco, tipicamente essa è asincrona ma provoca problemi in caso di mal funzionamenti perché se il sistema cade nel momento in cui una pagina modificata si trova sul buffer, tutte le modifiche vengono perse.

→ Naturalmente la scrittura buffer → disco riguarda solo le pagine modificate dalle applicazioni.

→ Le performance sono molto elevate perché le pagine a cui accediamo frequentemente in lettura si trovano già sul buffer senza la necessità di fare forse notevolmente un accesso al disco.

Tipicamente, come mostra il grafico, maggiore è la quantità di mem. disponibile e minore è il # di fault.



Differenze nella gestione interna della struttura del buffer tra sistemi a memoria virtuale e DB

### ① Spazio di indirizzamento

nei sistemi a VM ~ nell'ordine dei Mb - Gb con PageTable  
nei sistemi di DB ~ nell'ordine dei Tb con PageTable invertita

Con Page Table invertita, usata ormai anche nei sistemi a VM in quanto lo spazio di indirizzamento è ormai nell'ordine dei Gb, è ora facile la cui gestione è, in termini di tempo, più costosa delle normale PageTable ~~ma~~ anche se (cioè a causa del fatto che) occupa meno spazio di quest'ultima.

Esso mantiene un puntatore per ogni pagina in mem. reale (chiamato descrittori), esso quindi avrà più piccole & chi occorre mantiene un puntatore per ogni possibile pag. in mem. reale, mantiene solo i descrittori delle pagine effettivamente presenti in mem. reale. Il match fra la pagina cercata e quelle in mem si può fare con una tavola hash per velocizzarlo, tuttavia si impiega molto più tempo per capire se la pagina cercata è realmente presente nel buffer. (e cose del simile problema delle funzioni hash)

### ② Gestore del buffer

nei sistemi a VM ~ attivato per ogni riferimento alla memoria, praticamente continuamente

nei sistemi a DB ~ attivato quando si richiede una pagina del DB, meno frequentemente di quanto avviene ~~sia~~ nei sistemi a VM

La differente frequenza di utilizzo del gestore del buffer implica che, mentre nei sistemi a VM siamo obbligati ad usare delle politiche di replacement molto banali, quelle più sofisticate infatti farebbero decadere le prestazioni, nei sistemi a DB, proprio grazie alla bassa frequenza di utilizzo del gestore, possiamo usare delle politiche di replacement estremamente sofisticate tipo LRU (Software) ~ Least Recently Used, la quale sostituisce la pagina usata meno recentemente.

L'algoritmo LRU è un algoritmo a stack (non vuol dire che è implementato con uno stack!!) cioè gode della proprietà di inclusione la quale impedisce al numero di fault di aumentare al crescere della memo. del buffer, come ad esempio accade a FIFO. (Anomalia di Belady)

### Proprietà di inclusione

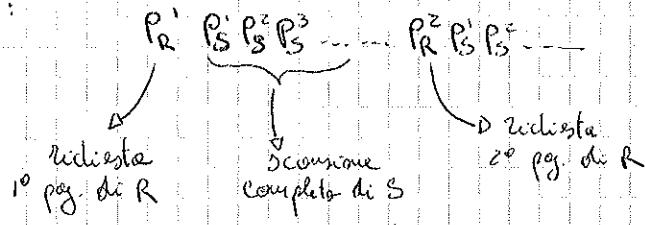
l'insieme delle pagine in memo. reale con  $S$  frames ( $P_s$ ) è un sottoinsieme delle pagine in memo. reale con  $B$  frames:  $P_s \subseteq P_B$   
il numero di fault quindi diminuirà (c'è una pagina in più) o sarà uguale.

### ③ Stringhe dei riferimenti: (sequenze delle richieste in memoria)

nei sistemi a VM non si conosce quel'è la stringa dei riferimenti; in questi sistemi si osserva un generico principio di località il quale consente che i riferimenti di un progr. in esecuzione tendono a localizzarsi in un certo limitato di pagine.

nei sistemi a DB si può prevedere o conoscere esattamente le stringhe dei riferimenti generati dall'applicazione. Questo è vero nei DB relazionali in quanto essendo non procedurali il sistema (e non l'utente) che crea ciò puono sulle basi di un mem. limitato di primitive; mentre non è vero nei DB gerarchici, reticolari (predecessori dei DB relazionali) o in alcuni sistemi DB su puri in quanto, in questi casi, è il programmatore a decidere come accedere ai dati.

Ad esempio per il nested-block conosciamo esattamente le stringhe dei riferimenti, essa è:



E' possibile calcolare il costo di esecuzione di un piano di caricamento del buffer utilizzando el' Hot Set Model : (caso Nested Block)

Supponiamo di avere

$$P_A = 5 \text{ pagine} \rightarrow \{1, 2, 3, 4, 5\}$$

$$P_B = 3 \text{ pagine} \rightarrow \{A, B, C\}$$

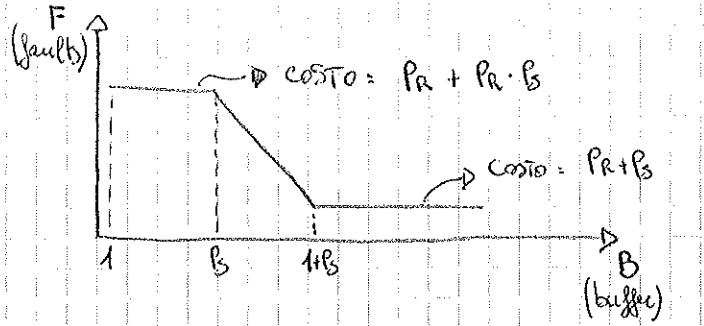
strutture di riferimenti: 1ABC 2ABC 3ABC --- (dove si riguarda nested block)

algoritmo di replacement: LRU

evoluzione dello stack LRU

		lettura					
		A	B	C	1	A	B
profondità	stack	1	A	B	C	Z	A
		1	A	B	C	Z	A
Passo	3		1	A	B	C	Z
4				1	A	B	C
5					1	A	B
6						1	A
7							1

- la profondità dello stack = n° frame nel buffer (3 frames di cui 1 libero)
- lo stack è ordinato in senso opposto al verso in cui stiamo leggendo i riferimenti
- Risparmio: se abbiamo un buffer di profondità 4 si può utilizzare tutto perché nel momento in cui riceve (A) al passo 6 di lettura questo si trova già nel buffer mentre, se abbiamo un buffer di profondità 3 quando riceve (A) al passo 6 di lettura, questo non è più presente e quindi do letto nuovamente (passo 6), questo lettura tuttavia spengerebbe fuori del buffer le pagine che ci servirebbe successivamente: B al passo 7 di lettura.
- Grafico Hot Set Model: (caso Nested Block)



Fissato il piano di esecuzione ed il metodo di replacement siamo in grado di determinare, usando questo Modello, il # di fault (cioè di Ifs praticamente) in funzione delle dimensioni del buffer  $\Rightarrow$  FAULT RATE

Il fault rate è caratterizzabile da un numero di intervalli stabili delimitati da punti caldi e punti freddi; i primi sono intorno l'allocazione ottimale (quindi sono quei punti che corrotteranno completamente il piano), i secondi invece sono gli estremi superiori degli intervalli individuati dai punti caldi e corrispondono ad un allocazione "irritile" di spazio nel buffer.

Il grafico mostra come:

- con un buffer di dimensione  $P_s$  (cioè 3 perché  $S$  è costituita da tre pagine) il # di fault è uguale a quello che ne avrebbe senza utilizzo di un buffer ( $B=1$ ), cioè non si riutilizza niente

• COLD POINT:  $P_s$

• HOT POINT,  $h_0 = 1$

• COSTO =  $Pr + Pr \cdot P_s$  (stima quadratica del nested-block)

$\rightarrow h_0 = 1$  è un punto caldo perché anche se allociamo più risorse (cioè meno di buffer), mantenendoci nell'intervalle  $[1, P_s]$  il # di fault non cambia, quindi allocare più di un frame avrebbe uno spreco.

- con un buffer di dimensione  $1+P_s$  (3 frame per contenere l'intera  $S$  poi un frame per scadrà  $R$ ) il # di fault diminuisce perché rischiamo di riutilizzare tutto.

• COLD POINT:  $+ \infty$

• HOT POINT,  $h_0 = 1+P_s$ ; è un HP per lo stesso motivo per cui lo è  $h_0$

• COSTO =  $Pr + P_s$ , costo lineare

- Come si nota dal grafico si passa da una situazione in cui

non si risulta niente ad una situazione in cui si riuscisse tutto, quindi di fatto LRU viene puro in disparte dal funzionamento del modello stesso. Da questa osservazione nasce il problema di chiedersi se

(A) Esistono degli altri algoritmi di replacement oltre ad LRU da poter utilizzare:

- In effetti nei sistemi a VM esiste un algoritmo ottimo chiamato OPT/Bo il quale tende a minimizzare il # di fault esaminando le stringhe dei raggruppamenti, tuttavia questo è un algoritmo che viene fatto a posteriori (che non si conosca in anticipo tale stringa) per valutare le prestazioni dell'algoritmo di replacement effettivamente usato: FIFO o Second Chance (ma non LRU la cui implementazione comporterebbe un eccessivo overhead mentre quella software è circondabile).
- Dopo che nei sistemi DB conosciamo o possiamo precedere esattamente la stringa dei raggruppamenti potremo scegliere di adottare l'algoritmo OPT/Bo anziché LRU, tuttavia si dimostra che il # di fault prodotto da Nested Block è questo caso > del # di fault prodotto da Nested Scan (il quale usa esplicitamente la memoria!) ; il motivo è che questo risultato è dovuto ad una diversa organizzazione dei raggruppamenti (la parte dell'algoritmo OPT/Bo) nel tentativo di ottenere un replacement ottimale, questo può comportare che le stringhe di raggruppamenti lati prodotte da Nested Block (riconosciuto secondo le tecniche di OPT/Bo) e Nested Scan risultino diverse, così come il # di fault complessivo.

(B) Esistono altri modelli alternativi al modello Hot Set:

esiste il Every Locality Set Model (Chu, Dewitt) il quale è descritto dall'Hot Set Model e caratterizza in maniera più fine i piani di esecuzione. La variazione introdotta riguarda il loop-revol (è una tecnica di riuso del buffer che corrisponde esattamente all'esempio visto precedentemente con il Nested Block del geogico); nel modello Hot Set è implicito che di volta in volta un frame venga utilizzato per scendere la rel. esterna e gli altri per scendere quella interna, con questo nuovo modello i due pattern di accesso (sequenziale e loop) vengono separati allocando un solo frame per la rel. esterna (il buffer costituito da un solo frame verrà ricavato secondo la tecnica semplice revol) ed altri M frame per la rel. interna (il buffer verrà ricavato secondo la tecnica loop-revol "puce")



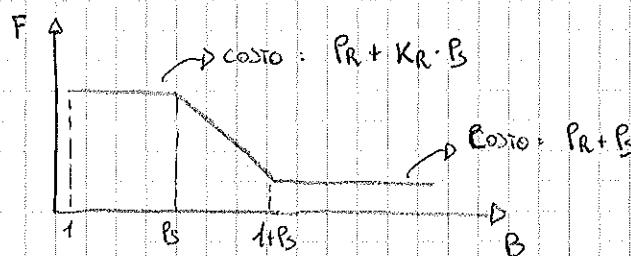
la politica di replacement (per la zola rel. interna) è MRC o Most Recently Used (per la rel. esterna la politica è obbligata ad avere un solo frame)

ESTERNA?

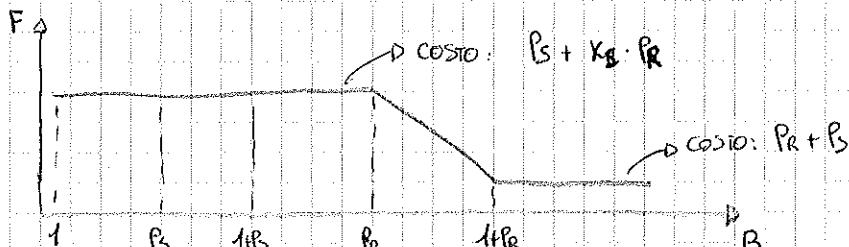
Il modello Hot Set Model permette di:

- calcolare costi "effettivi" di variazione della mem. Ausp. (come con il nested scans, anche se qui non usiamo esplicitamente la memoria) e quindi discriminare fra diverse posse di esecuzione

Ad esempio, Nested Loop a pagine



Se scambiamo fra loro le relazioni, supponendo che R riceva quelle più grande, otteniamo una valutazione diversa:



Notiamo che lo incremento del costo è ritardato finché occorre più spazio nel buffer per la relazione R (che deve entrare internamente + 1 frame per sovrapporre l'altra), tuttavia l'intervallo  $[1, Ps]$  costa  $Ps + Ks \cdot Pr$  cioè meno di quanto conta l'intervallo  $[1, Ps]$  nel precedente caso.

- determinare un'allocazione ottimale per le query prima della loro esecuzione.
- derivare politiche di precettione del thrashing a basso costo. Il thrashing ("agitarsi scompostamente") è un'attività di replacement utilizzata sui sistemi multitasking che condiziona i buffer e i quali si sottraggono le pagine di vicenda quando chiude in esecuzione. (thrashing esterno) Questo fenomeno, se portato all'estremo, può causare situazioni di collasso in cui l'unico effettivo che viene fatto è la pageazione. Esiste anche

il thrashing interno in cui un processo può uscire pagine e se stesso, ad esempio nel nested block con un solo hot point a  $hp=1$ .

Naturalmente quello che desideriamo è limitare il thrashing esterno ed esistono diversi modi per farlo:

(1) Desiderare processi attualmente schedulati in modo da diminuire il multi-processing; queste politiche non può essere applicata nei sistemi a DB a causa dell'elevata presenza di device che andrebbero monitorati.

(2) Utilizzo del Working Set Model (applicabile sia a sistemi VM che DB). Questo non è un modello di previsione (come l'Hot Set Model) ma è un modello che, basandosi sull'andare delle stringhe di riferimenti, stima il numero di pagine attive del processo, in un certo istante, osservando un principio di località;

MB Notiamo che per effettuare una stima occorre solo una visione del passato e non del futuro.

→ # pag. attive del processo = Working Set = località del processo

Mantenendo una finestra sui riferimenti a pagine singole effettuati in passato (cioè delle stringhe dei riferimenti <sup>non</sup> considerano quelli alle stesse pagine più volte ma solo una) e considerando i primi  $\tau$  di questi la stima del Working set al tempo  $t$  è

•  $WS(t+1) = W(t)$  per il principio di località

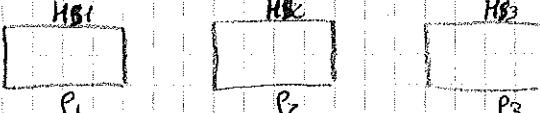
Giustificando che  $\sum_i WS_i \leq B$  non ci saranno problemi di risorse ma quando tale sommatoria supererà la dim. del buffer allora si inizierà a deschedulare.

→ Il problema principale di questo modello è stimare  $\tau$  se è troppo piccolo non si riesce a cogliere la località del processo, se è troppo grande il WS viene sovra stimato eccessivamente tanto da includere anche le meno località quindi questa sarebbe combattuta.

→ Risolti il problema della stima bisogna considerare il tipo di riuso che si cerca fare del buffer (dunque determinare il cambio delle località), non è detto infatti che un singolo  $\tau$  vada bene per tutti i possibili rafforzamenti dello stesso [vedi oltre x tecniche di riuso]

Nel caso dell'Hot Set Model il fenomeno del thrashing (esterno) si può prevenire e quindi evitare, basta avere:

$$\sum_i H_{Si} \leq B$$

Inoltre, considerando che solo LRU globale, il buffer viene spartito per i singoli processi.  Il buffer viene spartito per i singoli processi.  $H_{Si}$  è in uso una estera LRU ed è indipendente da ciascuno di essi.

Questo  $\Rightarrow$  comporta che le pagine conducano verso uno replicate in quanto il meccanismo LRU non è in grado di vedere le pagine di altri processi.

La possibilità del thrashing interno è così scattata.

Nel caso del Working Set Model sappiamo che ci possono trovare in situazione di thrashing che non sappiamo risolvere; inoltre esso è sempre applicabile mentre l'Hot Set Model lo è solo in quei sistemi in cui si conosce o si può prevedere la d.t.

Riferimenti facoltativi ad algoritmi di gestione del buffer con LRU globali dei riferimenti che tengono conto delle popolazioni delle pagine:

1 - O'Neil, Weikum  
"The LRU-K page replacement algorithm"  
ACM SIGMOD '93

2 - Johnson, Shasha  
"2A. A low overhead high performance"  
VLDB '94

## Tipi di Recupero del Buffer

- ① Simple Recup: un unico punto caldo,  $h_B = t$   $\Rightarrow$  scan sequenziale
- ② Loop Recup: quello studiato nell'esempio precedente del nested-block con il grafico; tipicamente è caratterizzato da M HP per JOIN e N relazioni; quindi possiamo avere che questo lineare e necessario che tutte le  $M+1$  relazioni stiano nel buffer + il frame libero per scendere l'ultima (quella più grande) e raggiungere minimizzare lo spazio occupato del buffer)

③ Index Rebal: [verrà trattato in seguito] struttura di accesso ad indici (come gli altri) le quali vengono accedute ripetutamente.

Vedi trattazione sui BTREE per garantire il minimo # di accessi

N.B. su ④

Nonostante il Simple Rebal sia molto bonito può comportare problemi di prestazioni nel caso di un LRU globale su un buffer: se c'è un processo veloce che fa una scansione del file, questo provocherà l'escita dal buffer di altre pagine del servizio (possibilmente per un rinvio indietro) ad altri processi (le pagine allocate per la scansione del file non saranno mai utilizzate!) perché in tretta di una semplice scansione !!)

Audizionare le stringhe di rigimenti per capire se stiamo facendo una scansione sequenziale o un loop molto grande è un compito non banale, tuttavia, l'ottimizzazione delle query, riesce a farlo e quindi bloccare solo un frame nel caso di uno scan.

### ③ - Metodi di JOIN basati sul partizionamento hash

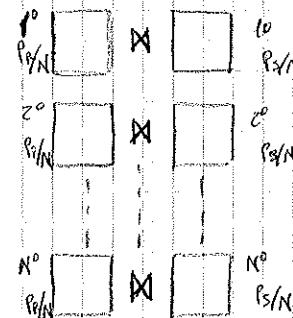
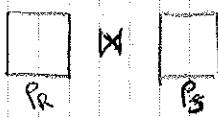
integrazione con articolo: "Fragmentation: a Technique for Efficient Query Processing" (INTEGRATO)

ACM TODS 11:2 1986  
Succo

Soltanto i paragrafi [3.2, 4.1, 6]

N.B.

- le prime oppure osserviamo che non ci siano duplicati nei valori degli attributi di JOIN
- la funzione hash distribuisce uniformemente i valori su N frammenti
- il partizionamento viene effettuato sempre nella relazione più piccola di ogni passo.



Il problema che risolve di JOIN tra R ed S viene ridotto in N problemi più piccoli ciascuno dei quali risolve un JOIN fra due frammenti di R ed S, opportunamente creati. Sotto certe condizioni, l'orme del risultato di tutti i JOIN sarà un costo lineare complessivo per l'operazione di JOIN nelle due relazioni iniziali.

Affinché ciò sia possibile gli N JOIN eseguiti sui frammenti delle relazioni debbono essere tra loro indipendenti ed avere un costo lineare. Per assicurare l'indipendenza bisogna partizionare sul attributo di JOIN in modo che i valori degli attributi di JOIN risultino disgiunti dai valori degli attributi di JOIN di relazioni che non fanno parte della giunzione corrente, ad esempio se abbiamo due relazioni da cui il JOIN scopia dati da entrambe cioè i match fra attributi di JOIN possono avvenire solo tra copie di frammenti che fanno parte della giunzione che si sta eseguendo.

Assumiamo che S sia la relazione più piccola, per ottenere un costo lineare il JOIN si deve soddisfare  $\frac{PS}{N} \leq B-1$  per ciascuno degli N frammenti infatti se ogni JOIN costa di tempo lineare di  $\frac{PS}{N} + \frac{PR}{N} = \frac{1}{N}(PS + PR)$ , il costo complessivo risulta  $\frac{1}{N}(PS + PR) = PS + PR$ .

se si dice  $\frac{P_5}{N} > B-1$ , cioè se un singolo frammento della partizione  $R$  deve rientra nel buffer, il partizionamento deve essere ripetuto ricorsivamente fino a quando lo stesso non riunisce tutti  $B-1$  frammenti.

### Def. di Partizionamento

- ① Per partizione  $i$ , il frammento della relazione  $R$  relativo alla partizione  $i$ -esima,  $R_i$ , ed il frammento della relazione  $S$  relativo sempre alla stessa partizione,  $S_i$ , hanno i valori dell'attrib. di join nello stesso insieme  $S_i$  (in bucket).
  - ②  $\forall i, j \ (i \neq j) \quad S_i \cap S_j = \emptyset$  (i frammenti di  $S$  sono disgiunti)
- Cioè implica che dato il valore di un attrib. di join esso si troverà solo in una partizione  $\Rightarrow$  i join possono essere eseguiti in parallelo eletti indipendenti.

Il costo totale dell'operazione di join tra  $R$  ed  $S$  sarà dato dal costo di partizionamento + il costo effettivo del join sui frammenti (che dovranno ridursi a  $P_R + P_S$ ) ; se forse vedere che in alcuni casi questo metodo è più o meno vantaggioso rispetto ad altri, tipo il merging scors.

### Singolo passo di partizionamento

Per partizione  $R_i$ , un frame del buffer viene utilizzato per leggere la pagina corrente di  $R_i$  e i restanti  $B-1$  per contenere i frammenti delle stesse (si creano  $B-1$  partizioni di  $R_i$ ). Per ogni tupla  $t \in R_i$  viene appena il frammento  $R_i[t]$  dove  $j$  è dato dalla funzione hash di partizionamento.

$$j = \text{hash}(t.a) \bmod (B-1) \quad \sim \text{la funzione hash quindi colonna dei frammenti assuniamo che:}$$

- la funzione hash usata per partizionare produce una distribuzione uniforme dei valori;
- i valori dell'attributo di join ( $R_i.a$ ) sono unici; questo implica (insieme alla precedente osservazione) che la distribuzione delle tuple nei vari frammenti di  $R_i$  è perfettamente uniforme.

**N.B!** quando si fa il join tra due frammenti, la funzione hash usata per partizionarli deve essere la stessa. Ma diversa per ogni diversa coppia di frammenti, altrimenti si ottiene sempre lo stesso risultato.

- ③ se due tuple hanno lo stesso valore nell'attrib. di join ( $t.a$ ) finiscono nello stesso frame

## Costi

- Il costo di una singola applicazione del partizionamento è di
  - $\geq P_R$  per partizione R ( $P_R$  per legge R e  $P_R$  per scrivere la partizione)
  - $\geq P_S$  per partizione S ( $P_S$  " " S "  $P_S$  " " "
- Ogni documento creato sono grande  $\frac{P}{B-1}$  frames, quindi i Join possono essere eseguiti in modo lineare se deve avere  $\frac{P}{B-1} \leq B-1$ , quindi se un grande documento risulta troppo grande, il partizionamento può essere applicato in modo ricorsivo sullo stesso sotto a quando non verrebbe in  $B-1$  frames:

# passi	# partizioni	dim media del documento creato da ogni partizione	dim max gestibile di PS relativamente a # di passi eseguiti	COSTO
1	$B-1$	$P_S / B-1$	$(B-1)^2$	$\geq P_R + \geq B + P_R + P_S = 3P_R + 3P_S$
2	$(B-1)^2$	$P_S / (B-1)^2$	$(B-1)^3$	
...	...	...	...	
K	$(B-1)^K$	$P_S / (B-1)^K$	$(B-1)^{K+1}$	

il processo termina quando  $\left\lceil \frac{P_S}{(B-1)^K} \right\rceil \leq B-1 \Rightarrow K = \left\lceil \log_{B-1} P_S \right\rceil - 1$   
 dove K è quindi il numero di passi richiesto per portare per una relazione di  $P_S$  pagine ed ottenere un insieme di documenti ciascuno dei quali sta in  $B-1$  frames.

- Il costo del join sarà quindi dato da: costo di PARTIZ. + costo del JOIN  
 cioè

$$\Rightarrow C_J = \geq P_R \left( \left\lceil \log_{B-1} P_S \right\rceil - 1 \right) + \geq P_S \left( \left\lceil \log_{B-1} B \right\rceil - 1 \right) + P_R + P_S$$

Mettiamo subito da

- il costo di partizionamento di una singola relazione:  $\geq P_R \left( \left\lceil \log_{B-1} P_S \right\rceil - 1 \right)$   
 è minore del costo che si dovrebbe pagare per ordinare con ~~mergesort~~  
 un external merge-sort:  $\geq P_R \left( \left\lceil \log_{B-1} P_R \right\rceil \right)$ ; stiamo assumendo che  $S < R$

- (A) - è differente dal merging-scans in cui il n° di passi è proporzionale alla relazione che stiamo ordinando, qui è fisso e dipende solo dalla relazione più piccola ( $P_S$  ad esempio)
- (B) - Mol è necessario che le triple siano sempre consecutive come nel merging-scans, è sufficiente che quella con lo stesso Id del join stiano in  $B-1$  frames per uscire da pagina con costo di join in più (una volta che tutti le triple siano in memoria si pagano costi aggiuntivi di  $I/O$  per fare il join); il partizionamento deve quindi far passare in meno del merging-scans (vedi pag 130 articolo)
- C) delle tabelle si nota come 1 singolo passo di granularizzazione è sufficiente per gestire relazioni  $S$  di  $P_S \leq (\beta-1)^2$  pagine, 2 passi di granularizzazione sono sufficienti per gestire relazioni  $S$  di  $P_S \leq (\beta-1)^3$  pagine e così via fino ad  $K$  passi necessari e sufficienti per gestire relazioni  $S$  di  $P_S \leq (\beta-1)^{K+1}$  pagine.  
 Notiamo quindi che il Número di passi di granularizzazione richiesti è abbastanza limitato, pertanto, quando ad esempio  $P_S$  supera di poco le  $(\beta-1)^2$  pagine, non conviene eseguire un altro passo di granular. (molti frame andrebbero sprecati) ma si può anticipare il join, ad esempio usando il nested ~~Map Reduce~~ [VEDI OLTRÉ]

Confronto con il Merging Scans (vedi grafico pag 119 articolo)

$$\cdot C_J = z_{PR}(\text{Fl}_S, P_S T-1) + z_{PS}(\text{Fl}_S, P_S T-1) + PR + PS \quad [\text{fragmentation}]$$

$$\cdot C_{MS} = z_{PR}(\text{Fl}_S, P_S T) + z_{PS}(\text{Fl}_S, P_S T) + PR + PS \quad [\text{merging-scans}]$$

- Se entrambe le relazioni sono ordinate il merging-scans è lavoro quindi sempre preferibile ad un partizionamento
- Se entrambe le relazioni ~~sono~~ da ordinare conviene sempre fare la granularizzazione per i motivi (A) e (B) elencati precedentemente
- Se dobbiamo ordinare quelle più piccole conviene usare il merging-scans
- Se dobbiamo ordinare quelle più grande conviene usare la granularizzazione

VEDI FORMULE SUL ARTICOLO.

## Esempio singolo passo di particionamento

R

1	5
7	10
0	9
3	2
6	8

X

S

1	11
7	9
2	4
5	12

due tuple per pagina

B = 3

hash function:

g: f.a AND z

Frammenti di R:

10	0
4	2
6	8

1	5
7	3

Frammenti di S:

1	11
2	4
12	-

7	9
5	-
-	-

Join A  
Join B

Join A:

il frammento di S sta interamente in memoria (è fatto da 2 pagine) quindi il costo del join è lineare

Join B:

il frammento di S non sta in memoria (è fatto da 3 pagine) quindi si dovrebbe frammentare ulteriormente (con una funzione hash diversa) altrimenti si otturerebbe di nuovo lo stesso frammento) ma poiché il frammento di R entra in memoria si può tenere questo nel buffer e quindi fare il join in tempo lineare senza eseguire ulteriori particionamenti.

## Problemi dell'overflow

Il metodo di base assume che le dimensioni di ciascun frammento sia la stessa; queste ipotesi è irrealistica nella maggior parte dei casi in quanto:

- i valori degli attributi di join sono spesso replicati in modo non uniforme.
- le funzioni hash utilizzate hanno una distribuzione non uniforme.

Mentre le funzioni hash possono essere scritte in modo che ci si aspetti di avere una minima deviazione dall'uniformità, i problemi maggiori derivano dalle distinzioni non uniformi degli attributi di join: si creano frammenti

di dimensioni molto diverse tra loro.

Le situazioni in cui ~~il~~ fragmento di S (ossia uno che ha la rel. più piccola) divenne di poco più grande di B-1 frame (cioè si classifica con over flow del bucket) vengono risolte utilizzando il nested scan.

**NB** ~~ma~~ è usato perdere quando le dimensioni (della distz. delle tuple nei fragmenti) è molto oltre ciò che si aspetta da un prodotto cartesiano (che è sempre una delle possibilità in cui può degenerare un join: idea dell'attrib. da join sempre grande) grande si applica il migliore metodo per risolverlo.

**NB** un altro indicatore di un possibile prodotto cartesiano è la presenza di bucket joins.

Oltre a fornire un modo per gestire gli over flow, ~~uno~~ scan può anche ridurre i costi dei join sui fragmenti, riducendo così lo spreco delle pagine. La modifica al metodo di base prevede un confronto dei costi tra applicare le fragmentazione classica ed anticipare alle K-1 estreme fare l'applicazione dell'**NB** ~~alla~~ anziché eseguire un altro passo di partizionamento ed un altro passo di join; se il confronto è a favore dell'applicazione dell'**NB** ~~alla~~ il join viene fatto alle K-1 estreme fare quindi il nested ~~join~~ e fondamentalmente per anticipare i join sia quando è conveniente, come nel caso precedente, sia quando è necessario, come nel caso dell'overflow, in situazioni di prodotto cartesiano.

### Algoritmo finale

**NB** il metodo esiste fondamentalmente di due fasi: partizionamento e join; poiché il processamento dei bucket (è indipendente) può essere eseguito in parallelo, l'algoritmo può essere reso completamente ricorsivo.

- ad ogni passo si considera una coppia di fragmenti in un bucket (i fragmenti iniziali sono R ed S) e si determina quelli più piccoli come base per il partizionamento:

① se  $i = 0 \rightarrow$  si termina, l'intero bucket viene scartato

② se  $i \leq B-1 \rightarrow$  si esegue un join con **NB** ~~semplice~~

③ confronta il costo dell'applicazione di ~~uno~~ **Nscans** con l'applicazione

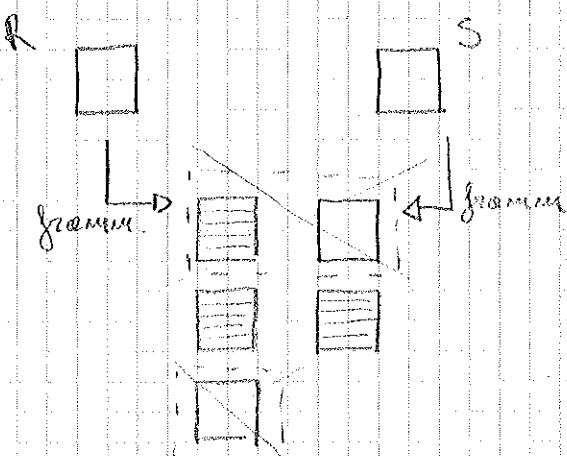
di fragmentazione su questa partizione; se il fragmento decide essere partizionato così effettua su questo una chiamata ricorsiva se ciascuno

fa la partizione e il bucket prodotto. Altrimenti

si applica il **NS scan** per eseguire il join. Questo divide l'algoritmo in due modi: si adatta meglio alle distribuzioni non uniformi e alle situazioni in cui  $P_A \approx P_B$ ; infatti le decisioni sono prese solo sulla base della situazione critica rispetto ad ogni passo.

Questo implica che

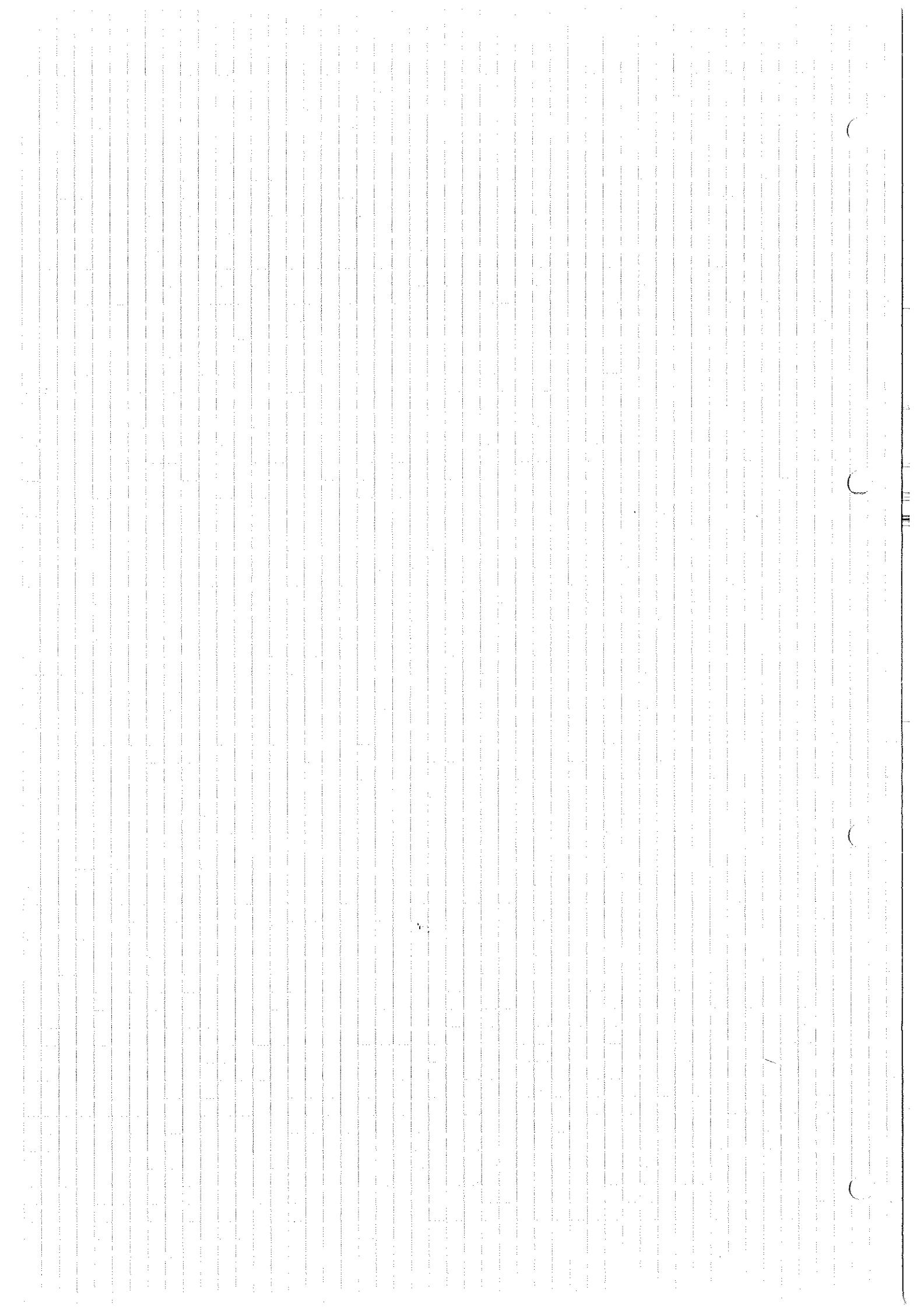
- ① l'overflow di un frammento può essere compensato dall'underflow dell'altro frammento nello stesso bucket.
- ② i bucket non devono più necessariamente essere partizionati.
- ③ i bucket vuoti possono essere elevati durante il partizionamento. Assumiamo che per ogni bucket il frammento più piccolo è partizionato prima di quello più grande, se questo produce un frammento vuoto il corrispondente frammento prodotto dal successivo partizionamento di quello più grande può essere controllato insieme a questo opposto prodotto:



(nella divisione di base)

Un problema di questo algoritmo è che, se ci estremi come il prodotto cartesiano, uno potrebbe mai terminare mai in quanto un frammento mai entra rebbi mai nel buffer e le chiamate ricorse si succedrebbero all'infinito. Per evitare questa situazione si può:

- ① stabilire un  $n^{\circ}$  massimo di passi di partizionamento
- ② osservare il comportamento corrente del partizionamento contro registrando quei frammenti che, a causa della loro eccedenza di dimensione, non devono più essere partizionati.
- ③ monitorare la presenza di bucket vuoti.



## Ottimizzazione delle interrogazioni (pg 279)

Il compito di ottimizzazione delle interrogazioni è affidato all'ottimizzatore il quale si occupa della scelta del piano di accesso per ottenere una query secondo uso degli operatori e delle strutture dati della macchina su cui.

Tipicamente l'ottimizzatore esplora in modo controllato lo spazio delle possibili soluzioni scegliendo quello di costo minimo (problema NP). La complessità del problema è dovuta a diversi fattori:

- una query può essere trasformata (in modi diversi) in un'altra, equivalente.
- gli operatori dell'algebra relazionale possono essere ridotti in più modi
- possono esistere delle strutture che agevolano l'accesso ai dati.

Es.  $\sigma_{R:a} \sigma_{S:b} (R \bowtie S)$

quello che si cerca di fare è anticipare le selezioni in modo che il join deve fatto su relazioni più piccole  $\Rightarrow$  esistono delle precise regole per farlo

### Regole di Trasformazione

#### 1. Commutatività e raggruppamento di selezioni

$$\sigma_x (\sigma_y (R)) = \sigma_y (\sigma_x (R)) \quad \text{commutatività}$$

$$\sigma_x (\sigma_y (R)) = \sigma_{x \cup y} (R) \quad \text{raggruppamento}$$

#### 2. Raggruppamento di proiezioni

$$\pi_z (\pi_y (R)) = \pi_z (R) \quad \text{se } z \subseteq y \quad \text{con } z, y = \text{liste di attrib.}$$

#### 3. Commutatività di proiezione e selezione

$$\sigma_x (\pi_y (R)) = \pi_y (\sigma_x (R)) \quad \text{se } x \subseteq y \quad \text{con } x, y = \text{liste di attrib.}$$

#### 4. Commutatività di proiezione e join

$$\pi_z (R \underset{x=y}{\bowtie} S) = \pi_z (\pi_{z \cup x} (R) \bowtie \pi_{z \cup y} (S))$$

Lod gli attrib. su cui fare join vanno mantenuti.

## 5. Commutatività di selezione e join

$$S_{xy}(R \times S) = S_x(R) \times S_y(S) \quad \text{se } x \in R, y \in S$$

## 6. Associatività del join

$$A \times B \times C = (A \times B) \times C = A \times (B \times C)$$

### Esempio

DB: Studente (MATRICOLA, NOME, PROVINCIA)

Esame (MATRSTUD, MATERIA, VOTO)

Query

```
select NOME
from Studente, Esame
where MATRSTUD = MATRICOLA and PROVINCIA = 'TO' and VOTO = 30
```

and MATERIA = 'ArchDB'

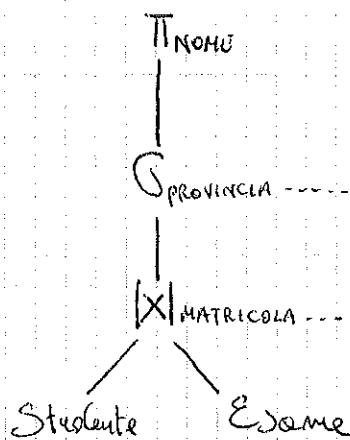
Cod.

$\Pi_{NOME} (G_{PROVINCIA = 'TO'} \wedge G_{MATERIA = 'ArchDB'} \wedge G_{VOTO = 30})$

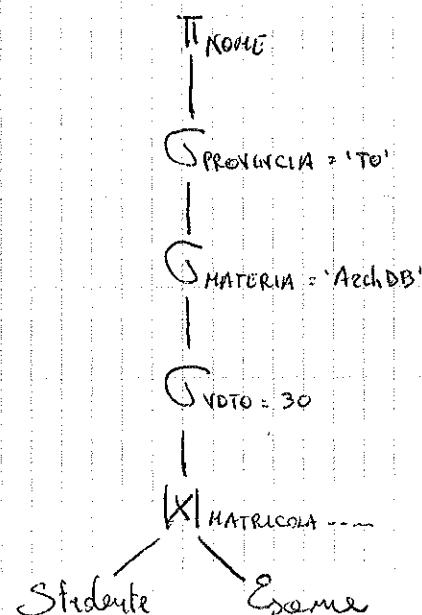
(Studente  $\times$  Esame)

~~MATRICOLA =  
MATRSTUD~~

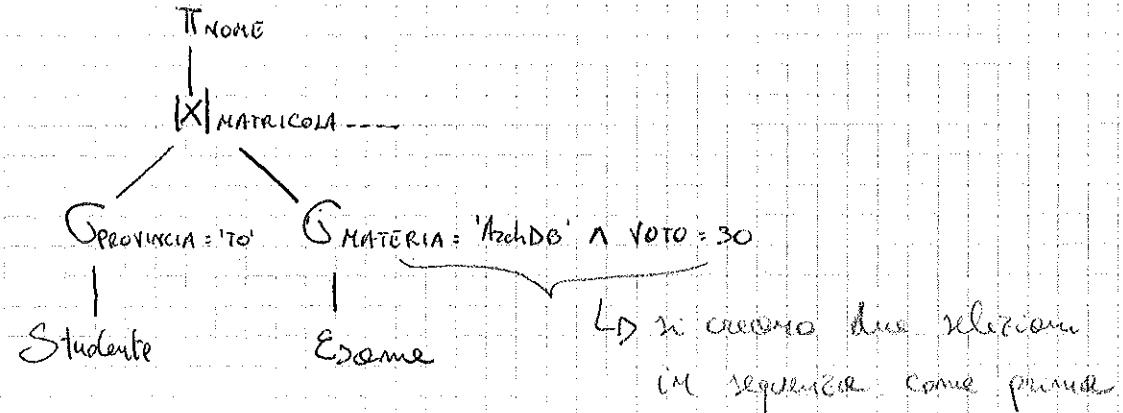
### Albero logico iniziale



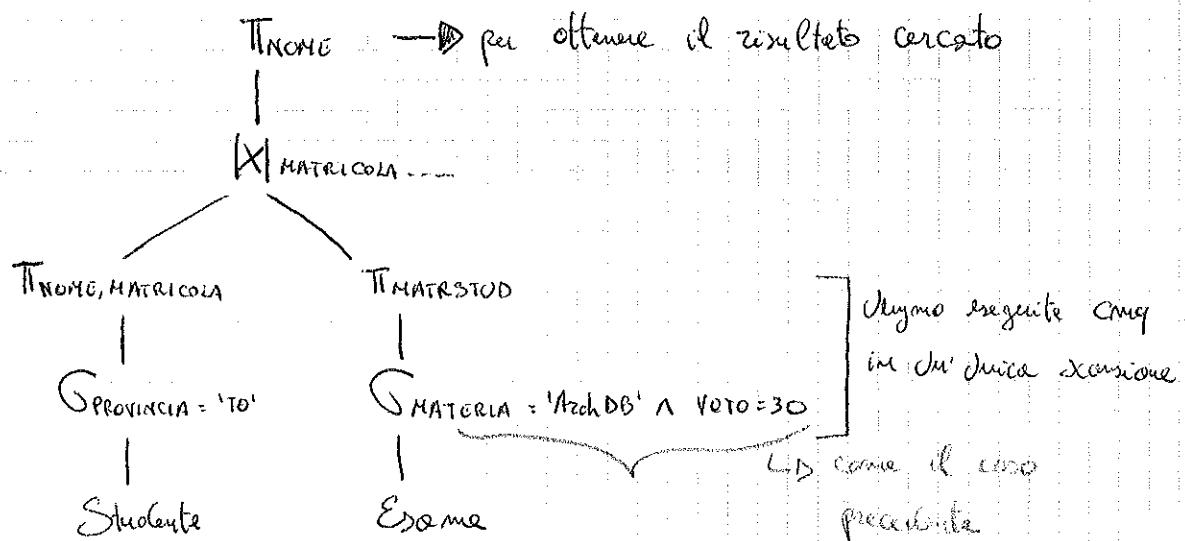
[1] Scindere le selezioni composte



[2] Anticipare le relazioni facendo migrazione nelle relazioni cui si riferiscono



[3] Anticipare le proiezioni



### Piani di accesso

Un piano di accesso è lo strategia di esecuzione dell'interrogazione di costo minimo, risultato della fase di ottimizzazione fisica.

Nel caso della nostra query abbiamo due alternative:

- Calcolare un temp per  $\text{T}$  e  $\text{S}$  obe: due zeri, applicare al join sui due temporanei creati, infine applicare la  $\text{T Nome}$ .
- Calcolare  $\text{T}$  e  $\text{S}$  delle relazioni esterne durante il join.

Se si calcola un solo temporaneo (che racchiude  $\text{T}$  e  $\text{S}$ ) delle relaz. esterna del join si esegue:

- scorsione della relazione esterna
- scrittura del temporaneo
- scorsione del temporaneo per fare il join

tuttavia si può fare di meglio se, mentre facciamo la scorsione per il join, calcoliamo in parallelo anche  $\pi$  e  $\sigma$ ; questo è possibile solo perché sulle rel. esterne non c'è alcun loop.

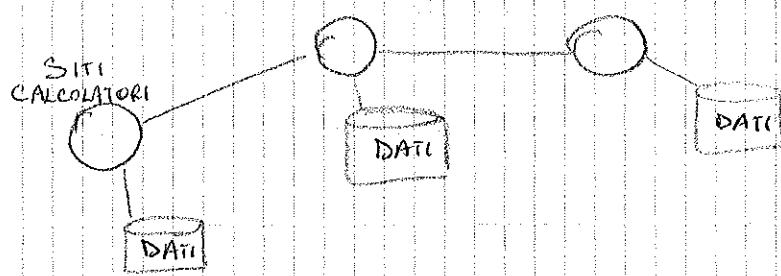
Osservando la relazione interna invece, poiché essa viene sconsigliata inoltre, conviene usare un temporaneo il quale è più piccolo della rel. originale (in teoria).

NB

Nel caso della grande relazione conviene pre-processare entrambi i ramî perché ad ogni passo non appiattiamo quale relazione verrà selezionata come più piccola.

Cenni su i DB distribuiti (per introdurre l'operazione di semi-join) pag 381

Un DBMS distribuito si compone di più DBMS dello stesso tipo cooperativo, che risiedono su siti diversi; ci saranno dei questi sistemi gestisce una parte dei dati complessi che possono essere replicati in altri siti. L'utente ha una percezione locale di una unica grande base dati.



Normalmente, ci saranno dei sistemi locali i cui grado di realizzazione autonomamente tutte le funzionalità di un DBMS completo, almeno fino a che le operazioni richieste coinvolgono solo dati locali; in caso contrario il sistema attiva una transazione distribuita la cui esecuzione richiede l'esecuzione coordinata di transazioni locali su più siti.

È compito dell'attivizzatore trasmettere gli strumenti verso i siti che contengono i dati necessari e materializzare il risultato sulla macchina dell'utente.

I costi principali sono:

- costo di start-up della trasmissione (costo fisso)
- costo di trasmissione (DIPENDE DALLA QUANTITÀ DI DATI DA TRASFERIRE)

VEDI PAG. 381  
SUCC. SINTA  
PRIMA  
 $t_{\pi}$  (costo di trasmissione delle proiez. su  $S$  degli attrib. di join) +  $t_{\rho}$  (costo di trasmissione della rel.  $R$  ridotta), naturalmente deve valere  $t_{\pi} + t_{\rho} \leq t_R$  altrimenti non il semi-join non conviene.

① Compito dell'ottimizzatore è di decomporre un'interrogazione globale (espressa cioè su relazioni distribuite) in tante sottointerrogazioni locali da eseguire nei nodi dove si trovano i dati corrispondenti (in parallelo naturalmente perché i nodi sono indipendenti), quindi deve trovare il modo di costituire il risultato finale e metterlo insieme sul nodo utente.

Criterio: trovare una sequenza ottimale di trasmissioni di relazioni in modo da trasmettere la rel. più piccola od un sottorel. con join, trasmettere il risultato od un altro sottorel. etc.

② Poiché il costo di trasmissione è una funzione lineare della quantità di caratteri trasmessi, l'obiettivo primario dell'ottimizzatore di interrogazioni distribuite è di minimizzazione tale quantità: semi-join,  $R \bowtie_S S$ , restringe  $R$  mantenendo solo le tuple di  $R$  che  $\in \Pi_R(S)$ , eliminando quindi tutte le tuple che non hanno un match nel join:

sotto di R

X
5
X
X

sotto di S

2
5



La trasmissione degli attributi del join,  $\Pi_R(S)$



trasmissione di R ridotta al sotto di S per fare il join localmente (od S)

$$R \bowtie_S S = R \bowtie_{\Pi_R(S)} S$$

NB

- quello della trasmissione di  $\Pi_R(S)$  è un costo aggiuntivo che bisogna pagare per ridurre  $R$ . VEDI COSTI PAGINA PRECEDENTE

- La semi-join è simmetrica

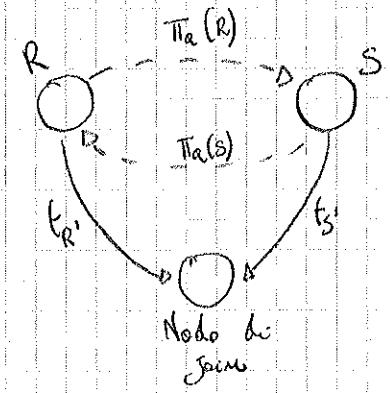


$t_R$

$t_S$

oppure

Nodo di join



- Le tecniche di semi-join sono applicabili anche se DB centralizzati

Esistono due metodi per calcolare il semi-join:

### ① - Metodo Esatto

- a. calcolo  $T = \Pi_a(S)$  eliminando i duplicati, nel sito delle rel. S
- b. trasmetto T al sito della rel. R
- c. calcolo  $T' = R \bowtie_a T$
- d. trasmetto  $T'$  al sito delle rel. S per calcolare il join originale  $R \bowtie_a S$

### ② - Metodo Probabilistico

→ eliminando i duplicati.

- (a). rappresento  $T = \Pi_a(S)$  con un Hash Bit Vector di dimensione N;  
 $\forall v \in T$  calcolo  $\text{hash}(v) \bmod N$  e setto il bit alla posizione restituita dalla funzione ad 1:  
$$\text{HBV}[\text{hash}(v) \bmod N] = 1 \quad \forall v \in T$$

- (b). trasmetto l'HBV al sito della rel. R
- (c).  $\forall$  valore h dell'attributo di join controllo il vettore alla posizione

$$\text{HBV}[\text{hash}(h) \bmod N]$$

se si trova 0  $\Rightarrow$  h non esiste in S quindi non farà parte del join  
se si trova 1  $\Rightarrow$  h potrebbe esistere in S, ma si ha l'errore bina  
certezza a causa delle possibili collisioni prodotte  
dalla funzione hash; gli errori vengono tollerati  
chi questo è un modo estremamente veloce per ridurre  
le R anche se questa viene ridotta meno di  
quanto effettivamente si potrebbe

T' sarà quindi costituita da R ~~minus~~ e conterrà le sole tuple i cui valori  
degli attributi di join hanno avuto un riscontro positivo (1) con l'HBV.  
I valori errati saranno eliminati durante la fase di join.

- (d). trasmetto  $T'$  al sito delle rel. S per calcolare il join originale  $R \bowtie_a S$

N.B. l'HBV rappresenta  $\Pi_a(S)$

## Esempio funzione hash in collisione

X										HSH
0	1	2	3		10					

questo bit a 1 rappresenta tutti i valori 10, 20, 30, ... tutti MOD 10, quindi quando si testa un valore dell'attributo quale è il bit corrispondente (dato dalla funzione hash MOD N) è a 1, non possiamo essere certi del valore rappresentato in S che ha causato la messa a 1 di questo bit.

$$\frac{M \text{ bit}}{N \text{ valori}} \Rightarrow \frac{M}{N} < 1 ; \text{ tanto più basso è questo rapporto, tanto più alta è la prob. di collisione.}$$

Questo metodo probabilistico viene anche chiamato dei Filtri di Bloom (permettono di eseguire un test probabilistico di appartenenza di un elemento ad un insieme).

## Applicazioni varie dei Filtri di Bloom (BF ~ Bloom Filter)

### (1)- Spelling Checker (ancora usato in DNAx)

- a. BF mantiene il dizionario
- b. il testo in ingresso viene suddiviso in parole e "filtrato", se otteniamo uno o siamo certi che le parole testata zapp. Un errore, altrimenti se otteniamo 1 non possiamo affermarlo con certezza

### (2)- Calcolo dei semi-join (come visto in precedenza)

### (3)- Calcolo dei valori distinti di un attributo

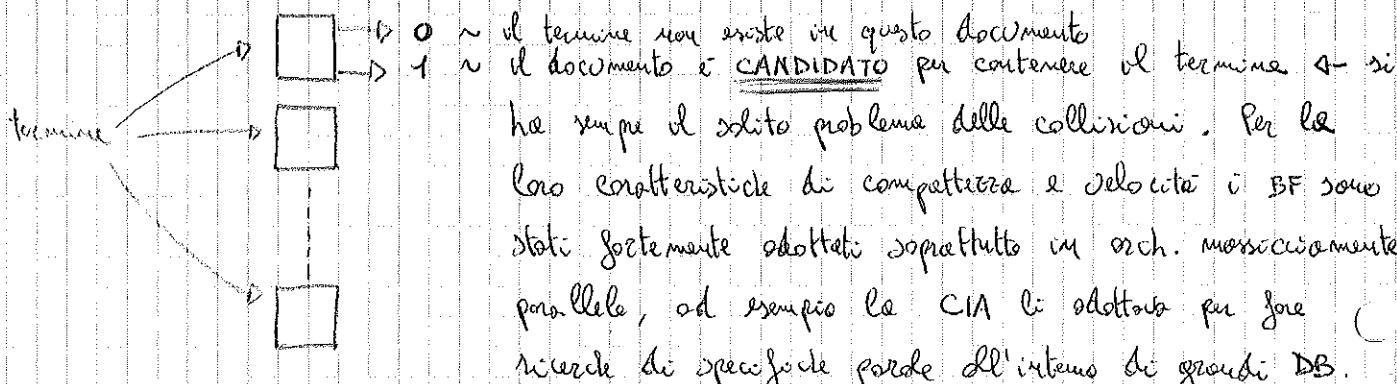
Il calcolo dei valori distinti di un attributo può essere fatto in molti modi diversi, più o meno efficienti:

- ① Usando degli indici.
- ② Usando una proiezione sull'attributo e contando i valori distinti.
- ③ Usando un BF inizialmente con tutti i bit a zero; nel momento del test di un valore, se otteniamo zero incrementiamo un contatore per i valori distinti e settiamo il corrispondente bit ad 1 (in modo che "lo stesso" non venga contato di nuovo), se otteniamo uno allora ignoriamo il valore testato. In questo modo è chiaro che otteniamo

Una soluzione dei valori distinti perché in caso di collisione ignoriamo il valore senza sapere se effettivamente esiste stato il bit si trova ad uno perché perde mente lo stesso era già apposato (e quindi conteggiato)

#### (4) - Text Search (chiamata anche Signature o Digest)

Si crea un BF per ogni documento



I documenti candidati vengono esaminati singolarmente.

#### (5) Caching delle pagine Web

Usato per ricercare una pagina Web: se il bit è 0 la pagina certamente non è in cache pertanto si deve fare un accesso al internet, se invece è 1 si controlla la cache nella speranza che il bit fosse stato posto ad 1 dalla presenza della pagina che ci interessa il che fa inoltre un collisione con essa.

Anche in questo caso è l'uso dei BF che la maggior parte delle pagine Web che visitiamo non sta in cache, così come le parole che contiene NON stanno in tutti i documenti che abbiamo, etc.

#### Considerazioni sui filtri di Bloom

##### • Vantaggi:

COMPATTEZZA  $\sim$  essi vengono memorizzati in RAM

VELOCITA'

##### • Svantaggi:

gli errori vengono omessi di fronte dei vantaggi ottenuti inoltre, la probabilità di errore può essere rese piccole (ma non nulla)

a piacere [VEDI OLTRÈ]

## Spiegazione dei filtri di Bloom

- Vettore ad  $M$  bit
- $K$  funzioni hash indipendenti

NB (verbale anche NB pagine successive)

- in linea di principio, come verrà dimostrato, converrebbe usare più funzioni hash possibile ma alcune più funzioni significa dare spazio più bit e quindi aumentare le probabilità di collisione; tuttavia al crescere di  $K$  le probabilità di collisione decresce in quanto le funzioni hash sono indipendenti

↳ ad esempio trovare che 10 funz. hash indip. producano un 1 in uno specifico indice è difficile (cmq possibile)

↓  
SOLUZ.

Tzade-off tra  $K$  ed  $M$  in particolare tra  $K$  e il rapporto  $\frac{M}{N}$   
visto in precedenza

### ② Inserimento $v \rightarrow BF$

- scegliamo settati  $K$  bit per  $v$ , ognuno con una funzione hash diversa
- $\forall i=1, 2, \dots, K$   $Hash_i(v) \rightarrow [0, M-1]$ 
  - ↓
  - ↳ mappare di una produce un valore da 0 a  $M-1$ , mod  $M$ ,
  - ||  $\forall v \in S$   $BF[Hash_i(v) \bmod M] = 1$  viene usato per settare il corrispondente bit  
con  $0 \leq i \leq K-1$  (come prima solo che ci sono  $K$  funzioni hash diverse che dunque non sovrapposono)

### ③ Test $v' \in BF$

- applico tutte le funzioni hash sul valore da testare, se esiste almeno uno dei  $K$  bit da controllo a zero,  $\exists b=0 \rightarrow v' \notin BF$  certamente.
- Se tutti i  $K$  bit da controllo sono ad 1 ce sono le solite due possibilità:
  - $v$  e  $v'$  sono lo stesso valore,  $v' \in BF$
  - esiste un valore di collisione con quello che cerchiamo

↓

- ||  $\exists i : BF[Hash_i(v) \bmod M] = 0 \rightarrow v' \notin S$
- ||  $\forall i : BF[Hash_i(v) \bmod M] = 1 \rightarrow ??$

## Considerazioni Probabilistiche (noi richieste per l'esame)

$M \sim$  dimensione del filtro

$N \sim$  elementi presenti nel filtro

$K \sim$  # di funz. hash utilizzate

- ①  $P(\text{specifico bit non resettato da 1 funzione hash}) = \frac{1}{M}$
- ②  $P(\text{"..."} \text{ "non resettato"} \text{ "..." "}) = \left(1 - \frac{1}{M}\right)$
- ③  $P(\text{"..."} \text{ "non resettato"} \text{ "..." "K"}) = \left(1 - \frac{1}{M}\right)^K$

→ dopo  $N$  registrazioni

- ④  $P(\text{specifico bit non resettato da } K \text{ funzioni hash}) = \left(1 - \frac{1}{M}\right)^{KN}$
  - ⑤  $P(\text{tutti i bit non resettati da } K \text{ funzioni hash}) = \left[1 - \left(1 - \frac{1}{M}\right)^{KN}\right]^K$
- quest'ultima probabilità rappresenta una ricontrazione di possibile errore, quindi vogliamo renderla minima:

$$\left[1 - \left(1 - \frac{1}{M}\right)^{KN}\right]^K \approx \left(1 - e^{-\frac{KN}{M}}\right)^K \Rightarrow \text{minimizzata direme:}$$

$$K = (\ln z)^{M/N}$$

$$P(\text{errore}) \approx \left(\frac{1}{z}\right)^K$$

→ Nel caso del semi-join il filtro viene dimensionato sulla cardinalità delle relaz. più piccole (che corrisponde ad  $N$ ):

- ①- si stabilisce la probabilità di errore che si è disposti a tollerare
- ②- dalle tabelle si ricava il rapporto  $M/N$  relativo a tale prob.
- ③- si determina, sempre dalle tabelle, qual'è il # ottimale di funz. hash da usare per ottenere la prob. di errore desiderata

## Esempio Trade prob. di errore

H/N	K	P(errore)	+ Prob. di errore	- Sicurezza del filtro
2	1	0,393		
8	6	0,021		
16	11	0,000454		
32	22	$2,1 \times 10^{-7}$		

NB

- Sul TRADE OFF tra K e H: per generazioni hash utilizziamo, minore è la probabilità di collisione in quanto tali generazioni sono indipendenti; tuttavia, all'aumentare del numero di generazioni usate, aumenta anche il numero di bit che si usano per rappresentare un valore quindi si riempie maggiormente il filtro, anche ridendo lo prob. di collisioni !!
- Trade per il Filtro di Bloom: [WWW.CS.WISC.EDU/~RCAZ/papers/](http://WWW.CS.WISC.EDU/~RCAZ/papers/)
- Sei TIPI DI ERROI del metodo probabilistico o Filtro di Bloom:

Errori Tollerabili: sono errori che non interverranno sul risultato finale prodotto ma ne determinano la solitudo, esempio: Spelling-Checker, Calcolo dei valori distinti di un attributo.

Errori Recuperabili: sono errori commessi in che prima fase ma che non influenzano il risultato se gli eliminati prima di produrlo, con esempio:

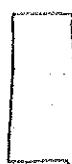
- Text Search: i documenti candidati possono essere erroneamente, essi essendo veramente esaminati regolarmente prima di affermare che quel preciso termine è presente nei doc esaminati
- Semi-jack: gli eventuali errori introdotti vengono eliminati durante il passo di jack finale
- Ricerca delle pagine Web nelle code: onde se una pagina viene segnalata nelle code e poi non ci si trova effettivamente, si fa un accesso al web per recuperarla.
- File differenziali ~ [vedi oltre]

## File Differenziali (ultima applicazione vista per i filtri di Bloom)

L'architettura a file differenziali può essere utile quando abbiamo cose che sono molto dinamiche con molti indici; di fatto questa architettura è stata proposta come architettura di base dei sistemi di DB.



file master molto grande ad accesso veloce ma con costo di aggiornamento molto elevato



file differenziale (sequenziale) che mantiene le modifiche fatte al file originale in modo da non dover pagare un costo eccessivo per modificare quest'ultimo.

- ② L'utente percepisce la presenza di un unico file aggiornato.
- ③ Il criterio di utilizzo è il seguente quando l'utente ricerca un determinato valore:
  - ④ se esiste un aggiornamento per lo stesso, lo si restituisce dal differenziale.
  - ⑤ se non esiste un aggiornamento si accede al master e lo si restituisce al li

N.B.

- ⑥ le ricerche dei valori vengono sempre effettuate per chiave (contenute nel differenziale); se la chiave corrispondente allo tuplo del valore cercato non è nel differenziale  $\rightarrow$  quel valore non è stato aggiornato e si deve accedere al master per recuperarlo

Problema:

- ⑦ per ogni record richiesto dall'utente si deve fare una scansione sequenziale del file differenziale e, se non lo si trova lì, un successivo accesso al file master: questo processo è molto lento e dipende direttamente dalla dimensione del file differenziale; anche se il file master viene esplorato in modo efficiente, si spende comunque molto tempo per recuperare un valore.

Soluzione:

- ⑧ l'esecuzione può essere velocizzata usando un filtro di Bloom sul file differenziale (il filtro contiene tutte le chiavi contenute nell'ultimo)

Tecniche di indirizzaggio (classificate in classificanti e non classificanti)

B-TREE

B+TREE

composizione delle diverse

(B&H)

operazioni sul B-tree per i B+TREE (insert, update, delete, search, insert, compare, etc.)

Hashing (Linear Hashing, Hash, Hash conglomerati, Hash hash-table)

Intro & TUT

Esauribile Hashing

Indici per diverse considerazioni (potenza dei puntatori)

deg. & altezza sottotree

Formule per hash

Algoritmi per codificare

- Distribuzioni non uniformi
- Ottimizzazione delle query (sempre)
- Ulteriori più indici secondari (tasse e albero delle condizioni)
- Velocizzazione dei metodi di JOIN
- Ottimizzazione delle query (dettagli)
- Morfismo A\* e orientati
- Eredità e decomposizione
- Problemi di selezione del disegno logico
- Trasformazioni

Test

il record certamente non è nel dispositivo per cui  
accediamo subito al master

1 ~ solita problematica dei BF ma gli ERROI sono  
recuperabili perché se i record segnalati come presenti sul  
dispositivo in effetti non ci sono, accediamo al master  
per recuperarli

## 2° PARTE DEL CORSO

### Tecniche di indirizzamento

Permettono di risolvere operazioni come la selezione in modo molto più  
rapido degli scac regolari

Analizzeremo inizialmente gli Indici per chiave primaria / candidata che sono  
delle strutture per accedere in modo rapido ad un record dati per mezzo di  
una chiave (se primaria / candidata significa che essa non ha duplicati)

Esistono due famiglie di indici

(1) - Basati sulle strutture ad Albero ~ costo accesso  $\Theta(\log N)$

(2) - Basati su hashing ~ costo accesso  $\Theta(1)$

Esempi di (1) sono: B-TREES, B+TREES

Esempio di (2) è: EXTENDIBILE HASHING che permette di creare e  
mantenere un file binario gerarchico con n° di accessi pari a 2 per  
accedere ad un record.

N.B.: query basate su intervalli di valori non sono risolvibili con  
tecniche hash, che quindi risultano applicabili solo in sottosistemi di cui:

Def. di indice: è una coppia  $\{X_i, R_i\}$  dove

$X_i$ : chiave

$R_i$ : indirizzo di dove il record  $\vec{x}_i$  è memorizzato oppure il record  $\vec{x}_i$  è  
proprio.

In base a cosa contiene  $R_i$  distinguiamo in due architetture:

## (1) - Organizzazioni clusterizzate

K	record
---	--------

i record sono memorizzati fisicamente secondo l'ordine della chiave quindi si può leggere sequenzialmente un file ordinato con il minimo numero di accessi visto i record sono contigui

A	Record-A
B	Record-B
C	Record-C
D	Record-D

$X_i$        $R_i$

Una volta individuata la coppia chiave + valore si può leggere l'intero file sequenzialmente come un file normale.

- Vantaggio: costo ridotto

- Svantaggio: ci può essere un solo indice  $X_i$  e ci può essere un solo ordinamento. Quindi non si può avere più di un file clusterizzato per selezione.

## (2) - Organizzazioni moy-clusterizzate

K	record
---	--------

i record non sono memorizzati contiguentemente quindi per leggere un intero file si paga di più

A	•
B	•
C	•
D	•

$X_i$        $R_i$

- Vantaggio: si possono avere più indici

- Svantaggio: si paga di più la lettura  $X_i$  i puntatori puntano in memoria fisica ai record quindi si pensano delle più eccezioni allo stesso pagina onde non consecutivi; ciò comporta che la stessa può essere caricata e scarcata dalla memoria per volte.

Scopi: (indici clusterizzati)

- ④ Elocuire le operazioni di **SELEZIONE**
- ④ garantire il vincolo di chiavi delle chiavi (primarie / candidate) senza le necessità di fare una scan della relazione ogni volta che aggiungiamo una nuova tupla.
- ④ Nelle situazioni pratiche, tipicamente, si usa un solo indice clusterizzato e tutti gli altri indici che abbiamo avere (per garantire il vincolo di chiavi ad esempio) o vogliamo avere (come nel caso delle chiavi secundarie per cui un vincolo di chiavi non vale) sono invece non clusterizzati. Per quanto riguarda la struttura di questi indici è sempre preferibile adottare una strategia di albero (come B-TREE o BT+TREE) o come della ridotta applicabilità delle tecniche hash (di cui com'è vedremo un caso: EXTENDIBLE HASH) e prevedere l'utilizzo di queste ultime solo in specifici casi.
- ④ Per quanto riguarda la tipologia di alberi da usare, non avrebbe senso utilizzare alberi con un basso grado (n° di figli di un nodo); vogliamo invece adottare alberi bilanciati, dinamici (per poter rappresentare al fondo che un file su cui l'indice è costituito può crescere e/o decrescere dinamicamente) e con massimo grado  $\times k$ :
  - maggiore è il n° di figli, minore è l'altezza dell'albero, minore è il tempo da impiegiamo per andare dalla radice al record cercato.

Ritroviamo queste ed altre caratteristiche negli alberi B-TREE (e derivati) (BAYER, MC-CREIGHT 1972) [che IBM implementò nel suo VSAM e Virtual Sequential Access Method, evoluzione dell'ISAM che invece era un indice non bilanciato e non binarico, onde se insopportabile tramite catene di overflow]

Vedi articolo "The Ubiquitous B-TREE"  
COMER

ACM COMPUTING SURVEYS 11:2, 1979

NB

- riferendosi ai record dell'indice si parla di coppia chiave-registrazione  
ma fare riferimento al fatto che la registrazione può contenere un record  
dati (se l'indice è clusterizzato) o un ulteriore puntatore (se non lo è)

### Def. Informale di B-TREE

- Albero multivice perfettamente bilanciato (le foglie sono equidistanti dalla radice)
- Un modo rappresentare una pagina, cioè è equivalente ad una pagina di mezzi.
- Il foglio è m e dipende dalla dimensione della pagina e dalle coppie chiave-registrazione dell'indice stesso.
- Le pagine sono riempite almeno del 50% (del foglio) con l'indice (eccez. della radice che può essere riempita anche meno del 50% (questo significa che quando le pagine sono riempite del 50%, l'altro 50% è semplicemente speso per mantenere il bilanciamento dell'albero))
- Il costo di accesso al dato (costo di ricerca, CR) è

$$CR \leq h \leq 1 + \log_{\frac{m}{2}} \left( \frac{N+1}{2} \right) \quad \text{dove } N = \text{nº di elementi dell'indice (coppie)}$$

M = foglio

nel caso peggiore vale CR=h ≤ m

h = altezza dell'albero

quindi l'altezza dell'albero rapp. un limite superiore al costo di accesso,  
questo vale se l'albero è perfettamente bilanciato

Le operazioni più comuni su questi tipi di alberi sono

- Inserimento, cancellaz. e modifica
- Accesso per Chiave
- Scan sequenziale anche parziale\* (per risolvere query di range)

## Def. Formale di B-TREE

Un B-TREE di ordine  $m \geq 3$  soddisfa le seguenti proprietà:

(1) FAN OUT minimo = m

(2) FAN OUT minimo, eccetto la radice =  $\lceil \frac{m}{2} \rceil$

FANOUT minimo per la radice = 2; è l'unico nodo dell'albero che può essere occupato per meno del 50% dello spazio

→ (3) ogni nodo interno con  $J$  figli ha  $J-1$  chiavi (coppie chiave-oggetto dell'index) e  $J$  puntatori

(4) tutte le foglie stanno allo stesso livello, sono quindi equidistanti dalla radice.

(5) ogni nodo ha la seguente struttura:

$$[ P_0 (K_1, R_1) \quad P_1 (K_2, R_2) \quad \dots \quad P_{J-1} (K_J, R_J) \quad P_J ]$$

base

Le chiavi  $K$  sono memorizzate in ordine crescente cioè

$$K_1 < K_2 < \dots < K_{J-1} < K_J$$

→ E se il fanout minimo del nodo è  $m$  (# di puntatori  $P_j$  che possono partire da esso), il numero minimo di chiavi  $K$  che esso contiene è  $m-1$

→ i puntatori  $P_j$  sono definiti solo per i nodi interni ed hanno il seguente significato:

$P_0$  ~ punta ad un sottoalbero contenente un insieme di chiavi minori di  $K_1$

$P_1$  ~ punta ad un sottoalbero contenente un insieme di chiavi maggiori di  $K_1$  e minori di  $K_2$

$P_2$  ~ ...

$P_J$  ~ punta ad un sottoalbero contenente un insieme di chiavi maggiori di  $K_J$

## Esempi

- Se la chiave che stiamo cercando ( $y$ ) trova un match con una delle chiavi del nodo interno attuale: abbiamo trovato il record che stiamo cercando.
- Se  $y < k_i$  allora  $y \in K(P_{i-1})$  cioè all'interno di chiavi  $K$  appartenente al sottodisco puntato da  $P_{i-1}$ :

	11	30	---	
$P_{i-1}$	$k_i$	$P_i$	$K_{i+1}$	$P_{i+1}$
	2	7	---	

$y$

- Se  $k_i < y < k_{i+1}$  allora  $y \in K(P_i)$ :

	11	30	---	
$P_{i-1}$	$k_i$	$P_i$	$K_{i+1}$	$P_{i+1}$
	12	13	---	

$y$

- Se  $y > k_j$  cioè della chiave più grande memorizzata nel nodo, allora  $y \in K(P_j)$  in quanto  $P_j$  punta ad un sottodisco contenente un insieme di chiavi maggiore di tutte quelle mem. nel nodo e quindi superiore di  $k_j$ :

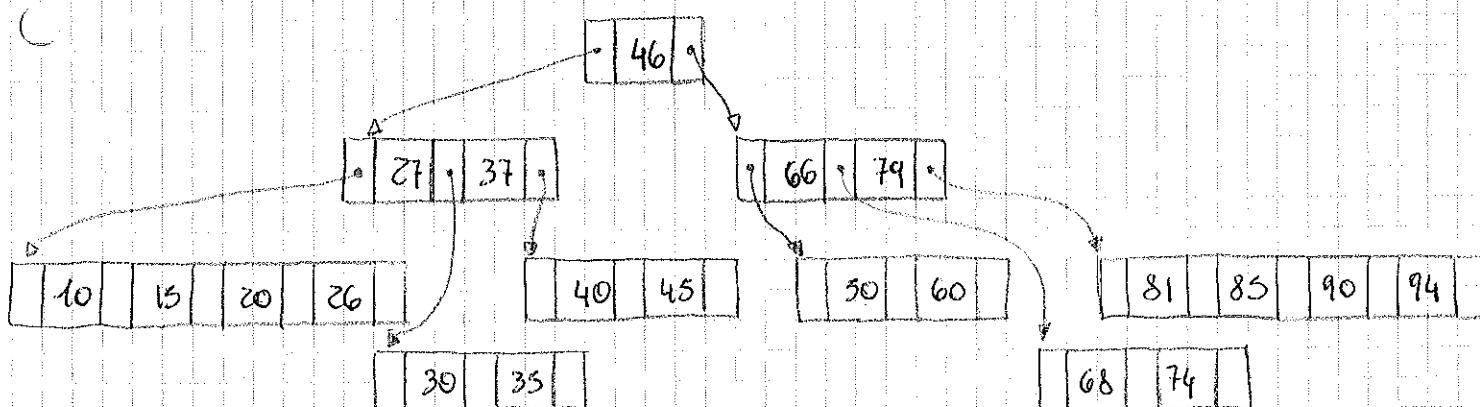
---	11	30	32	---
	$k_{j-1}$	$k_j$	$P_j$	
	35	41	---	

$y$

## Esempio di B-TREE di ORDINE 5 e altezza 2

ORDINE = 5 significa che un nodo può avere al più 5 puntatori (5 figli) e 4 chiavi e che, ad eccezione della radice, deve avere almeno  $\lceil 5/2 \rceil = 3$  puntatori (3 figli) e quindi almeno 2 chiavi.

Si noti come il n° delle chiavi può essere espresso in funzione dell'altezza dell'albero (come nell'articolo di CORM): ogni nodo in un B-TREE di altezza  $d$  contiene al più  $2d$  chiavi e  $2d+1$  puntatori, ed almeno  $d$  chiavi e  $d+1$  puntatori (ad eccez. della radice) ~ così il risultato di ogni nodo (ad eccez. eventualmente della radice) è sempre almeno di  $50\%$ .



## Esempio di ricerca di una chiave

$$y = 44$$

- si parte dalla radice:  $44 < 46 \rightarrow$  si segue il puntatore di sinistra
- $44 > 27 \rightarrow$  si procede alla chiave successiva
- $44 > 37 \rightarrow$  poiché questo nodo non contiene più chiavi si segue il puntatore alla destra dell'ultima chiave ( $P_3$ )
- $44 > 40 \rightarrow$  si procede alla chiave successiva
- $44 < 45 \rightarrow$  ricevendo questo un nodo foglie concludiamo che la chiave cercata non esiste

N.B.

• poiché i record dati non stanno solo al livello delle foglie ma a tutti i livelli, secondo cui chiave è possibile (se non sono sovrapposti) fermarsi anche prima di arrivare al loro livello, evitando così di attraversare tutto l'albero in profondità. Questo può solo se la chiave esiste altrimenti si arriverà sempre ad un nodo foglie.

④ A causa di questa organizzazione, la lettura regolare di un file che ha un indice di questo tipo richiede un attraversamento WORST dell'intero albero. Il vantaggio sarebbe quello di pagare meno i costi di accesso, ad esempio se cerchiamo una coppia  $\{x_i, k_i\}$  che sta nella radice, paghiamo un solo accesso. (vedi confronto con BTREE per valutare l'effetto vantaggio)

### Calcolo dell'altezza del B-TREE

$b = \text{n}^{\circ}$  di nodi nell'albero

$N = \text{n}^{\circ}$  di chiavi  $K$  (cioè di coppie  $\{x_i, k_i\}$  da inserire nell'albero)

$M = \text{ORDINE}$

$h = \text{altezza}$

①- Fissata l'altezza,  $b_{\max}$  è dato dalla situazione in cui ogni nodo ha il massimo numero:

$$b_{\max} = 1 + M + M^2 + \dots + M^{h-1} = \frac{M^h - 1}{M - 1}$$

radice      figli della  
radice       $\rightarrow M$  figli per ognuno dei figli della  
radice!!

②- Fissata l'altezza,  $b_{\min}$  è dato dalla situazione in cui ogni nodo ha il minimo numero:

$$\begin{aligned} b_{\min} &= 1 + 2 + 2 \left\lceil \frac{M}{2} \right\rceil + \dots + 2 \left\lceil \frac{M}{2} \right\rceil^{h-2} \\ &= 1 + 2 \left( \frac{\left\lceil \frac{M}{2} \right\rceil^{h-1} - 1}{\left\lceil \frac{M}{2} \right\rceil - 1} \right) \end{aligned}$$

③-  $N_{\min}$  =  $\text{n}^{\circ}$  minimo di chiavi che possiamo mettere nella radice  
+  $\text{n}^{\circ}$  minimo di chiavi che possiamo mettere in ogni nodo x il  $\text{n}^{\circ}$  minimo di nodi (privato di 1 xdi consideriamo la radice esplicitamente)

$$\begin{aligned} N_{\min} &= 1 + \left( \left\lceil \frac{M}{2} \right\rceil - 1 \right) \left( 2 \frac{\left\lceil \frac{M}{2} \right\rceil^{h-1} - 1}{\left\lceil \frac{M}{2} \right\rceil - 1} \right) = 1 + 2 \left( \left\lceil \frac{M}{2} \right\rceil^{h-1} - 1 \right) \\ &= 2 \left\lceil \frac{M}{2} \right\rceil^{h-1} - 1 \end{aligned}$$

2)  $N_{\text{Max}} = \text{NO MASSIMO di chiavi che possiamo mantenere per ogni nodo (inclusa la radice)} \times \text{NO MASSIMO di nodi}.$

$$N_{\text{Max}} = \frac{(m-1)}{m-1} \cdot m^{h-1}$$

quindi si ha che  $N_{\text{Min}} \leq N \leq N_{\text{Max}}$

cioè, sostituendo le espressioni per  $N_{\text{Min}}$  e  $N_{\text{Max}}$ ,  $\geq \lceil \frac{m}{2} \rceil^{h-1} - 1 \leq N \leq m^{h-1}$  il che esprime il  $\text{n}^{\circ}$  di coppie (chiave-registrazione) che possiamo mantenere in un albero di "ordine"  $m$  e altezza  $h$ .

Lower bound per  $h$ :

$$\begin{aligned} N &\leq N_{\text{Max}}, \\ N &\leq m^{h-1}, \\ h &\geq \log_m (N+1) \end{aligned}$$

Upper bound per  $h$ :

$$\begin{aligned} N_{\text{Min}} &\leq N, \\ \lceil \frac{m}{2} \rceil^{h-1} - 1 &\leq N, \\ \lceil \frac{m}{2} \rceil^{h-1} &\leq \frac{N+1}{2}, \\ h-1 &\leq \log_{\lceil \frac{m}{2} \rceil} \left( \frac{N+1}{2} \right), \\ h &\leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left( \frac{N+1}{2} \right) \end{aligned}$$

quindi l'altezza  $h$  dell'albero B-TREE di ORDINE  $m$  con  $N$  chiavi è:

$$\log_m (N+1) \leq h \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left( \frac{N+1}{2} \right)$$

questo ci dice con certezza che il costo di ricerca,  $C_R$ , di un elemento nell'albero è sempre:

$$C_R \leq h \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left( \frac{N+1}{2} \right)$$

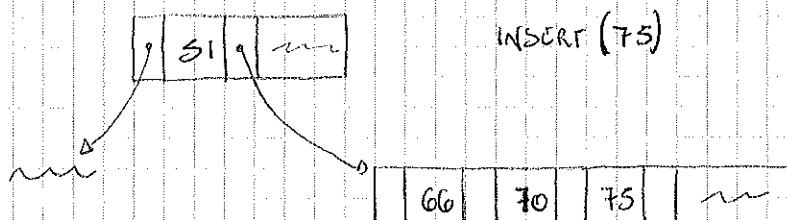
## Inserimento

l'inserimento di una nuova chiave richiede due passi:

- (1) rincorrere le chiavi stesse all'interno dell'albero per accettare che non esista già e per individuarne, nel caso non esista, la foglia in cui inserirla.
- (2)- inserimento e bilanciamento dell'albero a partire dalla foglia oppure modificata fino alla radice (se necessario).

Si possono presentare due casi:

- (1) Il nodo foglie ha sufficiente spazio per contenere un'altra chiave, la chiave viene inserita e la procedura termina:



- (2) Il nodo foglie contiene già tutti i chiavi, quindi un successivo inserimento causa un OVERFLOW:

Nelle n chiavi si seleziona l'elemento centrale; esso costituirà il separatore delle due sotto-nodi che conterranno rispettivamente tutte le chiavi più piccole del separatore e tutte le chiavi più grandi dello stesso.

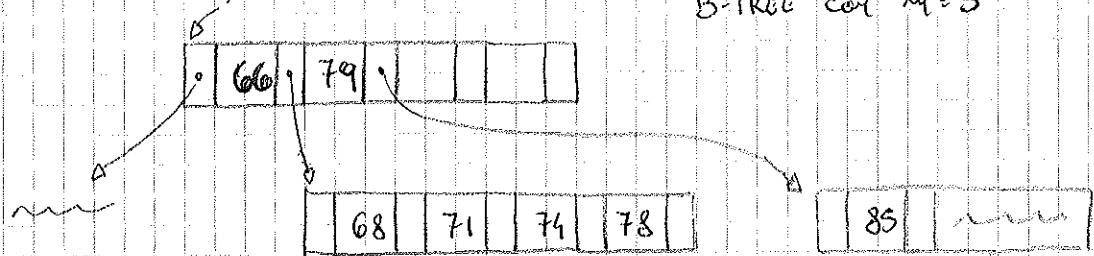
[NB]

- la foglia satira viene spaccata in due foglie contenute al minimo
- o il separatore verrà "inserito" (che vuol dire inserito) nel modo podre della foglia originale in modo da punti ad entrambe le foglie presenti.  
Se questo inserimento causa un ulteriore overflow al nodo padre si applica esattamente la stessa procedura oppure descritta.
- o Se la cotezza degli overflow e delle separazioni si propaga fino alla radice e anche queste si saturano se ne crea una nuova, contenente solo il valore di separazione; in questo caso l'albero aumenta di altezza di un livello.

[NB]- Una proprietà dei B-TREE è quella di crescere in altezza sempre e solo dalla radice.

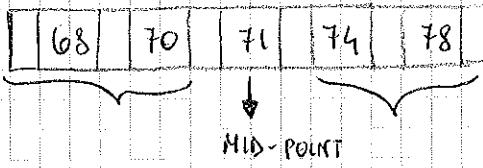
Esempio inserimento con un singolo overflow

B-TREE con  $m=5$

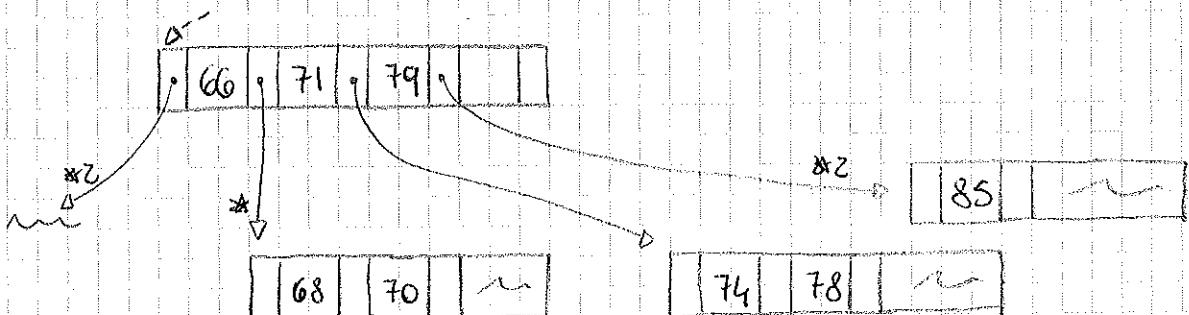


INSERT (70)

- La foglia in cui andrebbe inserito il 70 è satura
- le chiavi opportunamente ordinate, vengono separate in due gruppi:



- Vengono create due nuove foglie collegate al padre dell'elemento centrale individuato



\* questo puntatore may si cambia, ponte orice alla pagina originale,  
solo che essa si stava sovratata

\*2 questo puntatore non si aggiornato dell'operazione di inserimento

O Notiamo che l'altezza dell'albero may si cambia.

## Cancellazione (SPICHEZIONE TRATTA DALL'ARTICOLO)

- ① Anche la cancellazione richiede un primo passo di ricerca del nodo che contiene la chiave che vogliamo cancellare; ci sono due possibiltà:  
la chiave è menu. in questo modo foglia (in questo caso può essere corollata direttamente e la procedura termina) oppure è menu. in questo modo interno,  
in questo caso è necessario trovare una chiave "adiacente" a quella  
da cancellare, spostarla nella posizione di quest'ultima quindi cancellare  
la copia della chiave oppure spostata. \* POSSIBILITÀ DI UNDERFLOW
- ② Questo ha senso perché la chiave adiacente sarà sempre selezionata da una  
foglia adiacente al nodo in cui è menu. la chiave da eliminare in modo da  
poter eliminare la chiave senza preoccuparsi di ulteriori puntatori da sistemare.  
\* (come pure)

### Selezione della chiave adiacente

Dato la chiave da eliminare si individua "la più piccola chiave maggiore" di questa; essa sarà memorizzata nella foglia più a sinistra del sottoalbero di destra puntato dalla chiave da cancellare. (FIGURA 9, pag 126, articolo)

Equivalentemente si può selezionare "la più grande chiave minore" della chiave da cancellare: essa sarà memorizzata nella foglia più a destra del sottoalbero di sinistra puntato dalla chiave da eliminare.

### Eliminazione

Una volta sistemata la chiave adiacente al posto della chiave cancellata, bisogna eliminare la copia della chiave spostata rimasta nel suo nodo (foglie) originale; nel farlo bisogna stare attenti alla possibilità di un UNDERFLOW della foglia in quanto deve sempre soddisfare la condizione che tutti i nodi, ad eccez. della radice, siano riempiti almeno al 50%.

Vi sono due modi per gestire un underflow: concatenazione e ridistribuzione, entrambe queste operazioni agiscono su coppie adiacenti di foglie ma, mentre la ridistribuzione non influenza l'albero oltre il livello in cui si trova il nodo padre della foglia andata in underflow, la concatenazione può propagarsi fino alla radice.

• La scelta di una delle due operazioni dipende dal numero di chiavi presenti nel nodo adiacente a quello suddetto in Underflow; poiché entrambe le tecniche prendono in considerazione tutte le chiavi dei due nodi foglia per le chiavi che nel nodo potranno essere di queste, è il loro rapportore dobbiamo gestire un n° totale di chiavi dato da:

- una chiave come contributo del nodo potrebbe
- il nodo foglia in Underflow, poiché contiene meno di  $\frac{m-1}{2}$  elementi, contribuisce esattamente con  $(\frac{m-1}{2}) - 1$  chiavi
- il nodo foglia adiacente può avere un n° di chiavi compreso tra  $\frac{m-1}{2}$  e  $m-1$

### Ridistribuzione

- il nodo foglia adiacente a quello in Underflow contribuisce con un n° di chiavi  $> \frac{m-1}{2}$  (strettamente)
- in questo caso il n° totale di chiavi da gestire è  $> m-1$  (strettamente)

infatti se anche il nodo foglia adiacente contribuisce con  $\frac{m-1}{2} + 1$  chiavi avremo:

$$\# \text{chiavi} = \underbrace{\frac{m-1}{2} + 1}_{\text{nodo foglia adiacente}} + \underbrace{\frac{m-1}{2} - 1}_{\text{nodo foglia in Underflow}} + 1 = m > m-1 \quad (\text{Overflow})$$

nodo foglia adiacente      foglia in Underflow      nodo padre

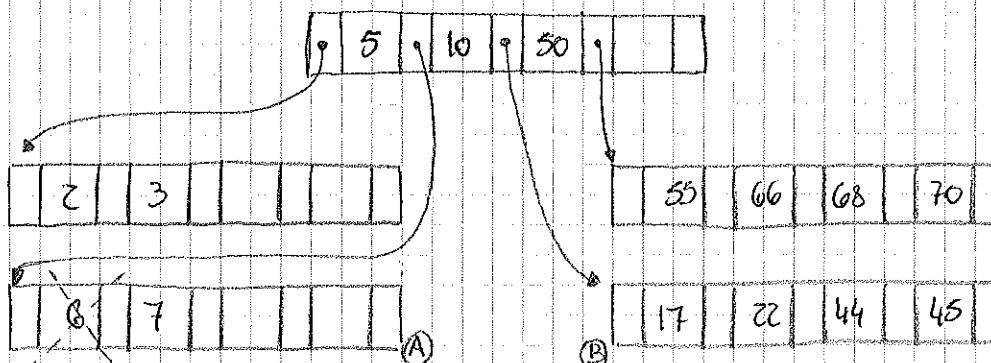
quello che si deve ridistribuire le ~~chiavi~~ chiavi tra le due foglie e il modo in modo che i nodi foglia siano conciati allo stesso modo (più o meno); tutte le chiavi da ridistribuire vengono suddivise e separate in due gruppi in modo simile a quanto avviene per l'inserimento solo che, in questo caso, per quanto riguarda il nodo padre andiamo di riplacarne o SOSTITUIRE le chiavi dello stesso valore e non di aggiungerne che nuove (la sua dimensione non cambia come invece può succedere nell'inserimento)

N.B.

in effetti basterebbe spostare una sola coppia chiave-registrazione per risolvere l'Underflow tuttavia visto che ci sono 3 nodi da riscrivere le 3 pagine (cioè i 3 nodi), conviene spostare più chiavi in modo da bilanciare le foglie adiacenti ed evitare così situazioni di sovraccarico del 50%.

## Esempio cancellazione con redistribuzione

B-TREE con  $M=5$



Cancella (6) ~ cancellazione di due chiavi in un nodo foglia

- La foglia (A) va in underflow perché con  $M=5$  il minimo n° di chiavi che un nodo non radice deve avere è  $\frac{M-1}{2} = 2$  ( $\lceil \frac{M}{2} \rceil = 3$  puntatori)

- La foglia adiacente più popolata è (B)

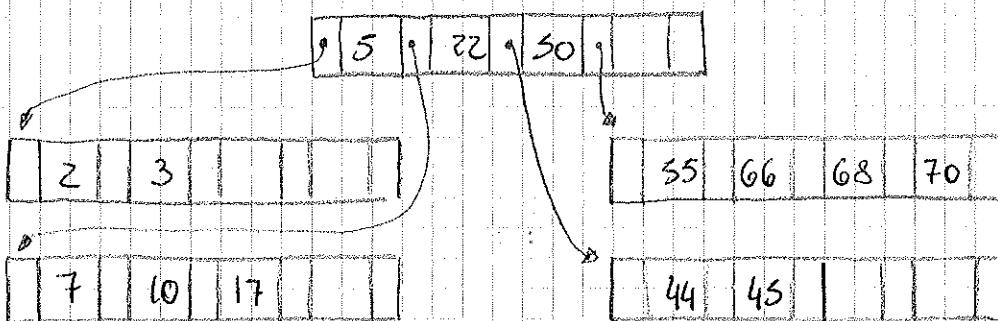
- Il n° tot. di chiavi da gestire è  $[7 \ 10 \ 17 \ 22 \ 44 \ 45] > M+1$  pertanto non si può creare un unico nodo (vedi concatenazione)

- Le chiavi vengono suddivise in:

Left: [7 10 17]

Center: [22] ~ prendere il posto occupato attualmente dalla chiave 10.

Right: [44 45]



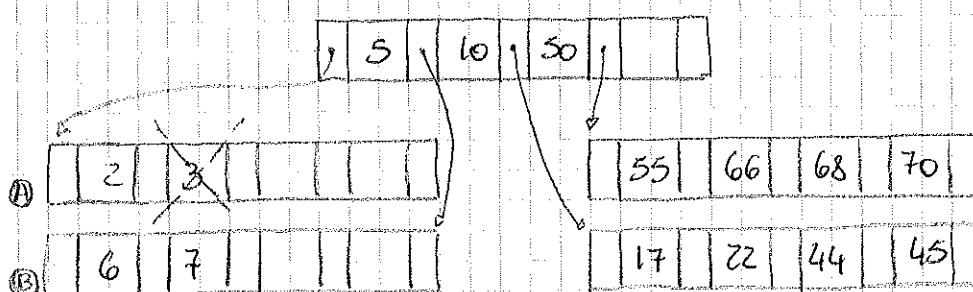
• Notiamo che la redistribuzione NON PUÒ influenzare l'albero oltre al livello che contiene il nodo padre delle due foglie da ristituire.

## Concatenazione

- il nodo foglia adiacente a quello in Underflow contiene con un n° di chiavi  $\frac{m-1}{z}$  (esattamente)
- in questo caso il n° totale di chiavi da gestire è:

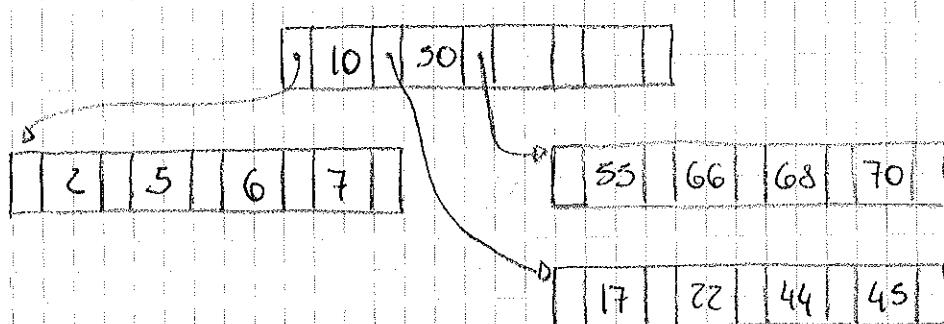
$$\# \text{ chiavi} = \frac{m-1}{z} + \underbrace{\frac{m-1}{z}}_{\text{foglia adiacente foglia in Underflow nodo padre}} - 1 + t = m-1 \quad (\text{condizione limite})$$

foglia adiacente foglia in Underflow nodo padre  
 quello che si fa è copiare tutte le chiavi di una foglia, nell'altro aggiungendo anche la chiave separatore del nodo padre; la foglia le cui chiavi sono state copiate nell'altro, viene eliminata:



delete(3) ~ cancellazione di una chiave in un nodo foglia

- poiché  $m=5$  la foglia ④ fa un underflow (come caso precedente)
- la foglia adiacente è ⑤
- il n° tot. di chiavi da gestire è  $[2 \ 5 \ 6 \ 7] = m-1$  pertanto questi valori possono essere inseriti in un nodo supe andare in overflow:



**NB**

la concatenazione produce sempre una foglia sottile.

Una complicazione di questa tecnica è che essa riduce effettivamente di 1 la dimensione del nodo padre della foglia concatenata, quindi anche questo nodo può avere un underflow (sarebbe in eccesso in questo esempio se il nodo padre avesse avuto solo le chiavi [5 10])

- Se anche un nodo padre da un underflow esso deve trattato esattamente allo stesso modo e la procedura deve ripetuta per livelli successivi fino a quando o non si raggiunge più underflow oppure la radice stessa da un underflow.
- Il minimo valore di riempimento per la radice è di 2 puntatori (1 chiave) quindi per poter andare un underflow essa deve contenere originariamente solo tale chiave che andiamo a rimuovere:
  - se la radice era anche un foglio: l'intero albero è stato eliminato!
  - se la radice non era un foglio allora originariamente aveva solo 2 sottosbari e questi, per carenze di cancellazioni e concatenazioni, sono arrivati al punto di doverli ricombinare tra loro, insieme alla chiave che nel nodo padre (la radice) li separava; ma se i due figli della radice vengono ricombinati da un solo nodo allora si può considerare questo come nuova radice ed eliminare quella vecchia (la quale contiene ormai solo due puntatori allo stesso nodo) ~ l'albero decrese di un livello.

NB

Se cercchiamo un B-TREE con un file ordinato, il caricamento ottenuto è minimo ad eccez. eventualmente dell'ultimo nodo:



i nostri inserimenti non andranno mai sulle pagine decchie ma sempre su quelle nere che verranno separate man mano.

Per evitare queste situazioni si effettua una ridistribuzione delle pagine in modo da non sovrapporre le sezioni dei nodi, in questo modo si ottiene un caricamento massimo tra una eventualmente sull'ultimo nodo.

Le situazioni di caricamento massimo non sono sempre positive infatti, nel momento in cui tutte le pag. sono (incominciano ad essere) piene, la probabilità di fare uno split èe coinvolge tutti i livelli dell'albero divenendo sempre maggiore

- Situazione OTTIMALE per la ricerca : l'altezza dell'albero è minima
- Situazione PEGGIORE per l'inserimento : frequenti split delle pagine → dovuti alle paglie

Conclusione sui costi di ricerca con un B-TREE

- costo migliore :  $\mathcal{O}(1)$  la chiave cercata è nella radice
- costo peggiore :  $C_R = h$  la chiave cercata è in un nodo foglia
- costo medio:

$$h = \log_{\frac{0,69}{\ln 2}} (N+1) \quad C_R \leq \log_{\frac{0,69}{\ln 2}} (N+1)$$

Varianti del B-TREE:

#### B\*-TREE

è un B-TREE in cui ogni nodo è pieno per  $2/3$  almeno ; la sola operazione di inserimento implica un bilanciamento fra 3 pagine (piene tutte per almeno  $2/3$  alla fine) il che fa aumentare le percentuali di spazio utilizzato per la memorizzazione (almeno il 66%) e di conseguenza velocizzare la ricerca di un elemento , in quanto l'albero è più basso di un B-TREE normale.

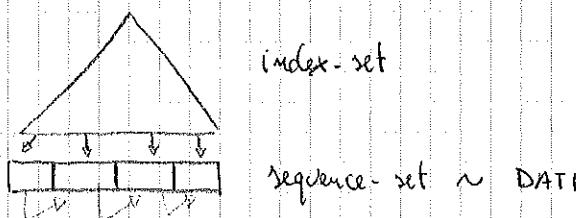
#### B<sup>+</sup>-TREE

- In un B<sup>+</sup>-TREE tutte le coppie  $(K_i, R_i)$  risiedono nelle foglie, essendo queste linkate tra di loro (singolarmente o doppicemente) , ci si riferisce ad esse con il termine sequence-set (in quanto possono essere viste come un normale file sequenziale ordinato.)
- Il livello superiore è fino alla radice è costituito esclusivamente da "chiavi di discriminazione" (vedi OLTRE) le quali formano la struttura dell'indice e NON rappresentano record dati ; ci si riferisce a questo insieme di chiavi con il termine index-set
- Il # di chiavi è uguale al # di puntatori : manca un puntatore rispetto al B-TREE tradizionale , in particolare se durante la ricerca discriminante con il  $<$  (cioè se  $y < K_i$  , punta al sottoalbero sinistro di  $K_i$ ) allora manca il puntatore per puntare al sottoalbero destro di

$K_j$ , nel caso in cui  $y > K_j$  dà  $K_j$  e l'ultima chiave del nodo; se invece durante la ricerca si incontra con  $y >$  (cioè se  $y > K_i$ , punto al sotto di  $K_i$ ) allora manterrà il puntatore per puntare al sottosbarco destro di  $K_i$ , nel caso in cui  $y < K_0$  dà  $K_0$  e le prime chiavi del nodo.

Questi due casi vengono risolti introducendo due chiavi fittizie ( $\infty$  e  $\phi$ ) che com puntatori validi i quali permetteranno l'indirizzamento verso i sottosbarci corretti.

- Poiché il B<sup>+</sup>TREE permette una netta differenza tra il livello indice ed il livello dati il costo di ricerca di una chiave sarà sempre  $C_{R_B} = h$  anche se, come si dimostrerà più avanti, non necessariamente l'altezza del B<sup>+</sup>TREE è la stessa altezza del corrispondente B-TREE.



Dallo studio del B-TREE, sappiamo che il costo di ricerca è  $C_{R_B} \leq h$  con la "possibilità" cioè di pagare meno di  $h$ , nel caso la chiave cercata si trovi in qualche livello intermedio tra le radice e le foglie; per sapere quale dei due metodi è più conveniente calcoliamo la probabilità di effettuare un n° di accessi uguali ad  $h$  (caso peggiore) con un B-TREE pieno al massimo:

$$P(\text{accessi} = h) = \frac{\frac{m^h - 1}{m^h}}{\frac{m^h - 1}{m^h - 1}} = \frac{(m-1) m^{h-1}}{m^h - 1}$$

# di nodi al livello foglie

# di nodi totali (tmax)

prob. di trovare od una foglia

Osservando  $m^h \gg 0$  in modo da ignorare la costante  $1$ , otteniamo

$$= \frac{(m-1) m^{h-1}}{m^h} = 1 - \frac{m^{h-1}}{m^h} = 1 - \frac{1}{m}$$

Come si può notare quindi, se il genot è massimo (cioè molto alto) allora la prob. di trovare sempre od una foglia è molto alta;

poiché quindi il B-TREE non offre grandi vantaggi rispetto al B<sup>+</sup>-TREE  
(con il quale siamo sicuri di pagare sempre h) in termini di costo di ricerca:

la probabilità, con un B-TREE, di fermarsi prima di che segno, durante la ricerca di una chiave, è inversamente proporzionale al fogliot dell'albero  $\rightarrow 1 - \left(1 - \frac{1}{m}\right)^h = \frac{1}{m}$

conviene usare i B<sup>+</sup>-TREE (infatti in pratica sono quelli attualmente usati) poiché non dobbiamo a riorganizzare i puntatori a tutti i livelli dell'albero ad ogni operazione di modifica e perché permettono la lettura sequenziale di un file con una semplice scansione del sequence-set, piuttosto che un altissimo numero di accessi all'interno dell'intero albero.

### Correlazione

- index-set: contiene coppie  $(K_i, P_i)$  dove  $K_i$  è una chiave usata solo come funzione di discriminazione, pertanto non è necessario che lo stesso si trovi più volte nel sequence-set (almeno una volta ci sarà stata se adesso queste si trovano nell'index-set non potrebbe non esserci più riconoscere la sua presenza nell'indice).
- sequence-set: contiene coppie  $(K_i, P_i)$  dove  $K_i$  è una chiave che rappresenta il nostro record dati.

il disaccoppiamento fra struttura dell'indice e i dati comporta che una correlazione può esistere solo a livello delle foglie che negli altri livelli non ci sono dati da correlare! Questo comporta che le chiavi possono essere eliminate direttamente e, fin tanto che le relative foglie rimane sìma sol 50%, niente lo necessita di modificare l'indice, anche se una copia delle chiavi cancellata si era propagata in esso. La discriminazione di successive ricercate viene effettuata correttamente anche se le chiavi  $K_i$  di confronto non si trovano più nel sequence-set.

VEDI Figura 14 pag 130 articolo Comer.

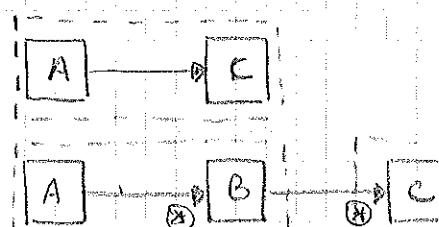
Se si utilizza un ordered-flow le tecniche di redistribuzione e cancellazione possono essere applicate anche ai B<sup>+</sup>-TREE, tuttavia potrebbe essere necessario un passo di riallineamento delle chiavi nell'index-set prima.

## Inserimento

Le procedure di inserimento e' quasi del tutto analogo a quelle dei B-TREE.  
 Solo che, in questo caso, quando una foglie va in overflow e si deve spezzare in due, anziché ~~una~~ promuovere al livello superiore la chiave in posizione centrale viene promossa una copia della chiave che ha causato l'overflow e quelle originale viene mantenuta nella foglia di destra.

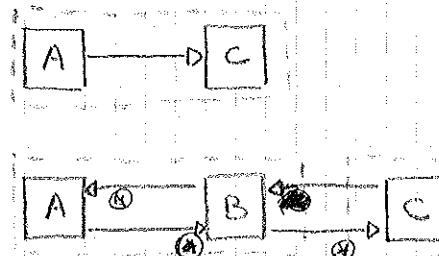
[NB] per il sequence-set nel caso di split delle pagine

(a) singolo link (insert di B)



l'aggiornamento dei puntatori ② non ha costi aggiuntivi che essi vengono modificati durante la riscrittura delle pagine

(b) link doppio (insert di B)



l'aggiornamento dei puntatori ③ non ha costi aggiuntivi come prima ma per aggiornare il puntatore all'indietro di C si deve fare un ulteriore accesso per un totale di 2 ogni volta che una pagina viene spezzata, contro l'unico accesso richiesto dall'orch. a singolo link (solo per il puntatore)

[NB]

- l'unico vantaggio nell'aver un sequence-set doppia mente linkato sta nella possibilità di leggere sequenzialmente da file anche un ordine inverso.

- n° di accessi nei due casi:

link doppio: 1 scrittura per la pagina che contiene A,B + 1

- link singolo: 1 scrittura per la pagina che contiene A,B + 1 scrittura per la nuova pagina (quella di C) + 1 lettura per la pagina che contiene A,B + 1 scrittura + 1 scrittura per la nuova pagina (per impostare il link per la nuova pagina  $\Rightarrow$  TOT = 2 accessi all'indietro)  $\Rightarrow$  TOT 4 accessi

Confronto fra le altezze, e quindi il n° di accessi nel caso peggiore, degli alberi B-TREE e B<sup>t</sup>-TREE

B-Tree, con caricamento massimo

- ① dim. pagina (nodo) = 2048 byte
- ② dim. chiave = 4 byte
- ③ dim. puntatore = 4 byte
- ④ dim. registrazione = 1014 byte
- ⑤ N = 10<sup>6</sup>

→ Notiamo che scegliendo volutamente la dimensione delle registrazioni (1014) in funzione degli altri parametri, in questo modo il fanout massimo che ci potranno essere sarà il più piccolo possibile, cioè 3 (chi nei nodi non soglia manteniamo delle registrazioni che occupano spazio a scopo dei puntatori che potremmo avere in più uscenti dagli stessi nodi)

Questo viene fatto esclusivamente per evidenziare le caratteristiche del B<sup>t</sup>-TREE che, non mantenendo record dati nell'index-set, massimizza sempre il fanout  $\sim$  tutto lo spazio e i puntatori sosterizionalmente. Verrà poi mostrato un esempio più "equo"

$$\begin{array}{c} \text{B-TREE} \quad M = 3 \\ \diagdown \quad \diagup \\ \begin{array}{l} 1018 \sim \text{due coppie chiave (4)} + \text{registrazione (1014)} \\ 2036 \sim \text{una'altra coppia} \\ 12 \sim 3 \text{ puntatori per 4 byte ciascuno} \end{array} \\ \hline \text{TOT} 2048 \end{array}$$

$$\log_{\frac{M}{2}}(N+1) = h_{\max} = \log_3 10^6 = \frac{\ln 10^6}{\ln 3} = 13 \text{ livelli}$$

$$1 + \log_{\frac{M}{2}} \left( \frac{N+1}{2} \right) = h_{\max} = \log_2 5 \times 10^5 + 1 = 20 \text{ livelli}$$

B<sup>+</sup>-Tree, con conciamento massimo

front  $m = 256 \leftarrow \frac{2048}{8}$  coppia (chiave, puntatore)

dato che nell'index-set non vengono inseriti dati, il front è maggiore di l'altezza dell'albero sono minore

→ l'ultimo livello dell'index-set ha un puntatore per ogni pagina del sequence-set quindi possiamo calcolare l'altezza dell'albero di funzione di esse (stiamo approssimando al n° di record che si trovano nell'intero index-set, al n° di record all'interno del sequence-set)

poiché ipotizziamo l'albero pieno, per ogni pagina non possiamo aderire più di due record (coppie), quindi si avranno

$\frac{10^6}{2}$  coppie  
 $\frac{10^6}{2}$  coppie per pagina  
=  $5 \times 10^5$  pagine totali nel sequence-set; questo implica che, per quanto detto prima, il numero di chiavi nell'index-set sono anch'essi di  $5 \times 10^5$

$$h_{\max} = \log_{256} 5 \times 10^5 = 3 + 1 = 4 \text{ livelli}$$

$$h_{\max} = 1 + \log_{128} 10^6 = 4 + 1 = 5 \text{ livelli}$$

sequenze-set concato minima: 1  
record per pagina:  $\frac{10^6}{1} = 10^6$  pagine cioè  $10^6$  chiavi nell'index-set

il n° di record dell'index-set non è semplicemente N ma sempre ricavato dal n° di record presenti nel sequence-set

Altro esempio ~ supponiamo sempre di avere un albero pieno

dim pagina = 2048 byte

dim chiave = 4 byte = dimensione puntuatore

dim registrazione = 1014 byte

$$N = 10^6$$

$\uparrow$  3 puntatori  $\times$  4 byte  
ciascuno

B-TREE

$m = 3 \rightarrow 3$  puntatori, 2 figli di  $(1014 + 4)$  byte ciascuno =  $12 + 2036 = 2048$

$$\log_{3^m}(N+1) = h_{\min} = \log_3^{10^6} = \frac{\ln 10^6}{\ln 3} = 13 \text{ livelli}$$

$$1 + \log_{\frac{N+1}{2}} \left( \frac{N+1}{2} \right) = h_{\max} = 1 + \log_2 5 \times 10^3 = 20 \text{ livelli}$$

B<sup>+</sup>-TREE

$2 \text{ record/pagina} \Rightarrow 10^6 / 2 = 5 \times 10^5 \text{ pagine totali nel sequenze-set}$

$$M = \frac{2048}{8} = 256$$

$\hookrightarrow 4 \text{ (chiave)} + 4 \text{ (puntatori)}$

$$h_{\min} = \log_{256} 5 \times 10^5 = 3 + 1 = 4 \rightarrow$$
 livello del sequenze-set

$$h_{\max} = 1 + \log_{128} 10^6 = 4 + 1 = 5$$

se combiniamo la dim. delle registrazioni, prendendole di 4 byte

B<sup>+</sup>-TREE

$$\frac{10^6}{256} = 3907 \text{ pagine totali nel sequenze-set}$$

$$\hookrightarrow m = \frac{2048}{8}$$

$$h_{\min} = \log_{256} 3907 = 2 + 1 = 3$$

Vantaggi del B<sup>+</sup>-TREE

- ottimizza l'utilizzo delle strutture indice massimizzando il fanout
- la lettura di un file sequenziale può essere fatta accedendo direttamente al sequenze-set inoltre, se questo è appiattito, la lettura può avvenire ugualmente in sequenza.
- non dobbiamo organizzare tutti i puntatori ai livelli dell'albero ad ogni op. di modifica

Il getto che nell'index-set le chiavi condiviso usato solo per discriminare  
se si che esse non possono comprendere per aumentare ulteriormente il  
fornito:

- Comprensione Frontale (senza perdita): gettonazione i prefissi comuni  
fra due chiavi consecutive

Esempio

ANNA  
ANNA LIBERA  
ANNA LISA  
ANNA MARIA



O ANNA  
LIBERA ~ 4 elen. in comune con la  
chiave precedente

Sintesi:

- \* le operazioni di modifica sono più complesse: se ad esempio cancelliamo la chiave ANNA LIBERA bisogna modificare il<sup>o</sup> contatore della lunghezza del prefisso comune sulla chiave ANNA LISA
- \* le operazioni di ricerca sono più lente xclsi si deve decomprimere ogni chiave per trovare quello che stiamo cercando
- \* queste op. vengono dette a "costo zero" xclsi in mem. centrale

- Comprensione in Coda ~ ROR (con perdita): minimo discriminatore fra  
due elementi

Esempio

Laura  
LIBRA

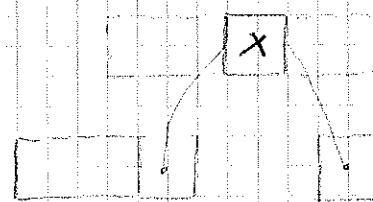
=> è sufficiente il prefisso "LI" per discriminare  
Laura da Lisa

questo tipo di comprensione, usata nei B+TREE, admette  
ulteriormente il fornito ma può essere usata solo nell'index-set su  
quanto le info in esso contenute ci servono solo per discriminare e  
non rappresentano record dati. (il cause della perdita di info, non  
possiamo usarla nel sequence-set o nei B-TREE tradizionali)

Con le tecniche di compressione al formato discreto varabile: non c'è più scelta a livello di albero ma dipendentemente dalle chiavi e dai prefissi che esse comprendono;

[N.B.]

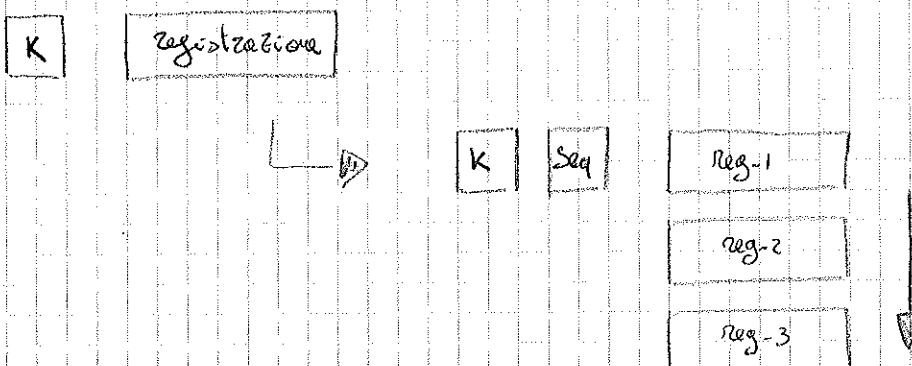
- ② Nelle foglie i prefissi comuni sono di più perché i record tendono ad ordinarsi per valore di chiave.
- ③ Le chiavi precedenti e successive sono scelte tenendo presente la struttura dell'albero.



il discriminatore "X" è l'istruzione delle chiavi per grandi e per piccole dei sottoalberi di sinistra e destra rispettivamente; esse rappresentano la chiave precedente e successiva di "X" nonostante il tronco su di diverso livello dell'albero.

Con direzione single dagli indici ha al albero

Se il campo registrazione di una coppia risulta troppo grande per essere mantenuto in memoria, esso si può frammentare



- ④ Il campo "Seq" contiene i numeri di sequenza dei due blocchi per poter ricostruire il campo unico
- ⑤ Notiamo che in questo modo accedere ad un record implica via Cittadella segnificare di saltare o posizionarsi dal 1° blocco, fino a che non compare la chiave.

- Questo tecnica veniva usata in passato per memorizzare gli oggetti BLOB ~~soltanto queste rimanevano~~ all'interno del DB, ogni tali oggetti vengono memorizzati nel file system con un puntatore, all'interno del DB, che punta ad essi.

## Buffe Management per i B<sup>+</sup> TREE

articolo non obbligatorio

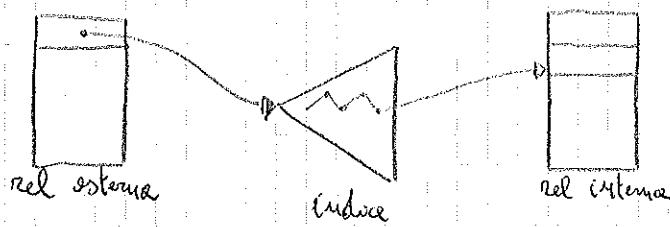
"Index Access With a Finite Buffer"

13th VLDB 1987

- Lo scopo è quello di trarre una politica di gestione del buffer "ottimale", ovvero che garantisca il minimo di accessi minimi ad un B<sup>+</sup> TREE.

### Possibilità di riuso del buffer (index reuse)

- indice comdiviso, ad esempio per un'applicazione tipo "conti correnti" in cui si usa ripetutamente l'indice per accedere alle info di uno specifico conto.
- indice per join, data una tupla della rel. esterna si usa l'indice per trovare tutte le altre tuple sulla rel. interna:

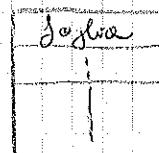
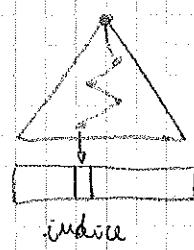


In questo caso supponiamo che il join abbia fatto sotto ad una chiave primaria cioè senza ripetizioni dell'elenco dell'attributo di join.

- Intuitivamente notiamo che il livello più ricavato (in entrambi i tipi di ricavato) è quello della radice; per quanto riguarda il riuso degli altri livelli, tanto più si accede all'indice, tanto maggiore è la probabilità che questo accada;

Vedi 2° foglio dell'articolo consigliato ~ viene mostrato come per poter ricavare il livello delle foglie sono necessari almeno 5000 accessi all'indice altrimenti, mantenere anche tale livello in memoria, è invitabile.

La politica di replacement proposta inizialmente per mini-mizzare gli accessi allo mem. secondaria anche per struttura ad indice è stata LRU ma, a causa del pesante riuso delle pagine dell'indice (in particolare della radice) e della tendenza di LRU a memorizzare comuni di altrove verso il modo verticale, facendo sì che le pagine sostituite siano in realtà quelle che ormai non più, queste tecniche non vengono applicate ed al suo posto si utilizza il modello IRM ~ Independent Reference Model il quale garantisce una replacement ottimale utilizzando una versione dell'LRU detta OLRU ~ Optimal LRU



stack di altrovengimento LRU

→ Verrebbe scartata da LRU  
stack di altrovengimento che usata meno di  
recente.

- Q) trattandosi di uno stack le pagine vengono men. in ordine inverso rispetto l'ordine in cui accediamo alle stesse quindi alla fine la radice, che è sempre la prima pagina a cui accediamo quando oriamo l'indice (quindi la pagina più riusata), si viene a trovare al fondo dello stack e quindi scartata.

### Independent Reference Model (IRM)

Note le probabilità di accesso per tutte le pagine si può implementare una politica di replacement ottimale suddividendo il buffer in:

- ① B+ frames per mantenere in modo "permanente" le B+ pagine più probabili a esse vengono sostituite se cambia la loro prob. di accesso, diminuendo minore della prob. di accesso ad una pagina di altrettanto. Non si tratta tra questi B+ frames ma non sono soggette al replacement classico  $\Rightarrow$  chiamate LRC; Optimal LRU la quale risulta di ~~soltanto~~ 30-50% migliore dell'BRU tradizionale.
- ② I frame per mantenere, una per volta, tutte le altre pagine

## Replacement

① - le p pagine vengono ordinate secondo i loro decessenti di  $\{P_i\}$  (probabilità di accesso alla pagina  $i$ , che assumiamo di conoscere)

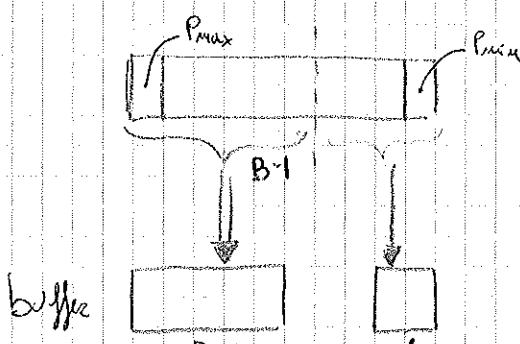
② - l'insieme delle pagine ordinate viene partitionato in due insiemi

$H \cong$  contiene le prime  $B-1$  pagine più probabili

$$H = \{ P_{i_1} \mid 1 \leq i \leq B-1 \}$$

$L \cong$  le restanti pagine

③ - bloccano  $B-1$  frames all'insieme  $H$  e 1 frame all'insieme  $L$



La politica di replacement è, anche solo intuitivamente, ottimale perché manteniamo in memoria sempre le pag. La cui probabilità di utilizzo è massima

Nei sistemi a memoria virtuale le probabilità di accesso alle pagine non si conoscono, ma nel nostro caso, possiamo stimarle:

① assumiamo una distribuz. uniforme delle prob. di accesso alle foglie

②  $B^T$ -TREE di ordine  $m$

③ le probabilità mostrate avrebbero normalizzate rispetto l'altezza dell'albero

LIVELLO

0 (radice)

1

2

..

L

PROB. DI ACCESSO

1

$1/m$

$1/m^2$

..

$1/m^{L-1}$

# FRAMES

1

2

$m+2$

..

$M+L$

COSTO (in termini di Scolt)

0

$h-1$

$h-2$

..

$h-L$

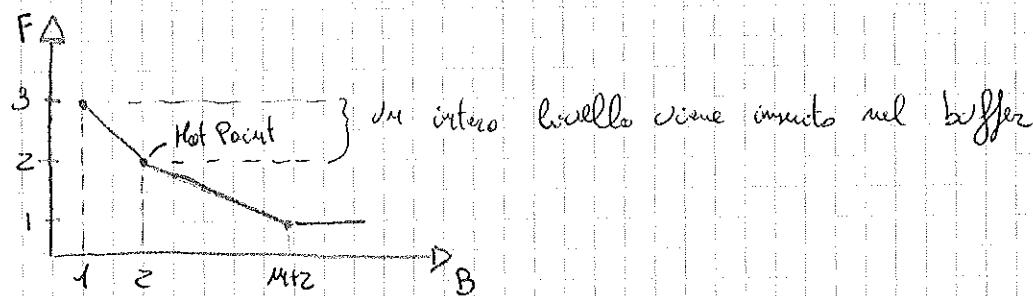
## Osservazioni

- con 2 frames: si mantiene il blocco della zodice nel buffer e si usa l'altro frame per i restanti blocchi
- con  $M+2$  frames: 1 frame per la zodice  
1 frame per il livello successivo  
1 frame per scorrere i restanti blocchi  
il costo di accesso è chiaramente  $h-2$  poiché i primi due livelli stanno nel buffer
- Per diminuire il costo di accesso di 1 è necessario un buffer sempre più ampio

$1 \text{ M} \quad M^2 \quad M^3 \dots$

qui si il grosso del intaglio di c'è all'accessione come quello del stree che procede per livelli, anzitutto, si ottiene nei primi livelli dell'albero.

Bisogna considerare anche che, come si vede dalla tabella riportata precedentemente, essendo la crescita delle pagine nello stesso livello esponenziale, la probabilità di ricevere alle stesse decresce all'aumentare dei livelli.  $\Rightarrow$  reale vantaggio solo nei primi livelli dell'albero.



- L'Re può essere utilizzato se non ne ha sufficiente spazio per mantenere un intero livello in memoria.

# Tecniche di indirizzamento basate su Hashing

Integrale con

"Dynamic Hashing Schemes"

INTEGRATO

ACM Computing surveys

20.2 1988 fino a pag 95 2° paragrafo incluso

Lo scopo dell'hashing è quello di effettuare una trasformazione chiave - indirizzo  
in modo che la coppia chiave-registrazione venga poi memorizzata all'interno  
restante della funzione stessa.

L'hashing dovrebbe assicurare una distribuzione (pseudo-casuale) uniforme degli indirizzi  
generati, il suo limitante sta nel fatto che i file creati sono ordinati per  
prezzo-chiave (cioè per indirizzo) e questo rende impossibile eseguire letture sequenziali  
dei file o query di range.

## Metodi per il calcolo della funzione hash

- ①  $H(K) = K \text{ MOD } M$  dove  $K$  è una chiave numerica di cui vogliamo  
calcolare l'hash ed  $M$  è un numero primo molto grande
- ② Metodo dei quadrati controllati: quadrato del seme + shift a dx per prendere i  
bit controllati della chiave
- ③ Tecniche di SOMMA-XOR-QUADRATO utilizzate sia su chiavi non numeriche  
in modo da ottenere una serie di cui calcolare successivamente il Modulo  $M$ ,  
sia su chiavi numeriche, per randomizzarle ulteriormente prima di calcolare  
il modulo

## Funzionamento dell'hashing

$K \rightarrow H(x)$



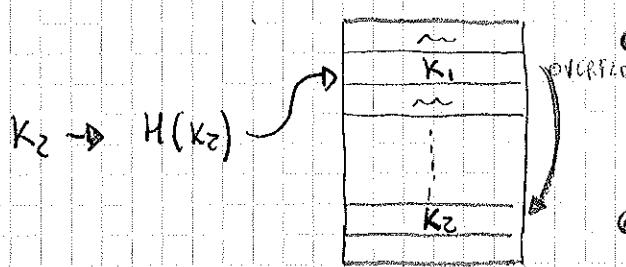
file con  $M$  slot

→ se stiamo facendo un inserimento, inseriamo la coppia  $(K, H(x))$   
→ se stiamo facendo una ricerca bisogna controllare che  
in questa posizione ci sia effettivamente la chiave  
che stiamo cercando; l'hashing infatti soffre del  
problema di overflow degli indirizzi (collissioni  
di più chiavi nello stesso bucket); il  
costo effettivo di accesso non è quindi  $\Theta(1)$   
ma dipende da come trattiamo l'overflow:

## Linear Probing

- metodo di risoluzione dell'overflow che utilizza solo un'area primaria
- Chiamiamo Home Address un indirizzo, restituito dalla funzione hash, al quale si trova memorizzata una certa chiave

Quando si definisce un overflow su l'Home Address si esegue una scansione  $M \times M$  del file (quando si cerca alla fine, si ricomincia dall'inizio) in modo da cercare uno slot libero in cui inserire la chiave che ha causato l'overflow:



Questo schema non è molto datogross \*che tecnicamente, ogni accesso ad uno slot può stare su due pagine distinte.

•  $K_2$  viene memorizzato ad un home address diverso da  $H(K_2)$  questo implica due problemi:

- 1 - in fase di inserimento,  $K_2$  occupa uno slot che appartenrebbe ad una chiave  $K_4$  il cui  $H(K_4)$  sarebbe l'home address al quale si trova mem.  $K_2 \rightarrow$  ~~overflow~~ overflow in cascata.
- 2 - in fase di ricerca, se  $K_2$  non viene trovata all'home address  $H(K_2)$  non vuol dire che essa non ci sia del tutto; potrebbe essere mem. in uno slot diverso, quindi in questo caso si dovrebbe fare una scansione sequenziale del file: tempo di accesso  $\Theta(M)$

### Conseguenze derivanti da ①

- Maggiore che si procede negli inserimenti la velocità di accesso al file degenera perché la probabilità di collisione aumenta.

\*

- Se ~~il # delle chiavi da inserire (N)~~ è  $M-1$  siamo nel caso peggiore perché per ~~ogni~~\*3 chiavi da inserire\*3 si deve scendere l'intero file che ha tutti gli slot occupati tranne uno

\*3 Che detta un overflow

\* il numero delle posizioni occupate nel file

\*2 Uno

- se il file ha tutti gli slot occupati, questo metodo non terminerebbe mai ciclando all'infinito durante la ricerca di uno spazio libero.

### Limitazioni del Linear Probing (Ulteriori)

- ① Non si conosce a priori il tempo di risposta massimo.
- ② La dimensione del file hash è fissata ad  $M$  questo significa che, se il file deve crescere si deve fare un rehash con un file ( $M'$ ) più grande mentre se il file deve decrescere si deve fare lo stesso con un file ( $M''$ ) più piccolo per evitare di sprecare spazio.

rehash: consiste nel calcolare un nuovo hash per una chiave, utilizzando una funzione hash diversa dall'a precedente

### Soluzione al problema delle collisioni degli overflow

- ③ Se si vuole utilizzare una tecnica di tipo Linear Probing, eventualmente procedendo alla possibilità di fare un rehash in caso di collisioni, tipicamente si sceglie di utilizzare un  $M$  sovrastimato (accettando di sprecare spazio) visto che le operazioni di trasformazione degli  $\#$  sono molto costose e danno limitate il più possibile
- ④ Si possono usare alcune variazioni della tecnica Linear Probing oppure tecniche di hash binarie le quali cioè permettono al file di crescere in funzione del  $\#$  di chiavi da inserire, garantendo inoltre un accesso costante. (Vedi, per quest'ultimo caso, l'extensible hashing)

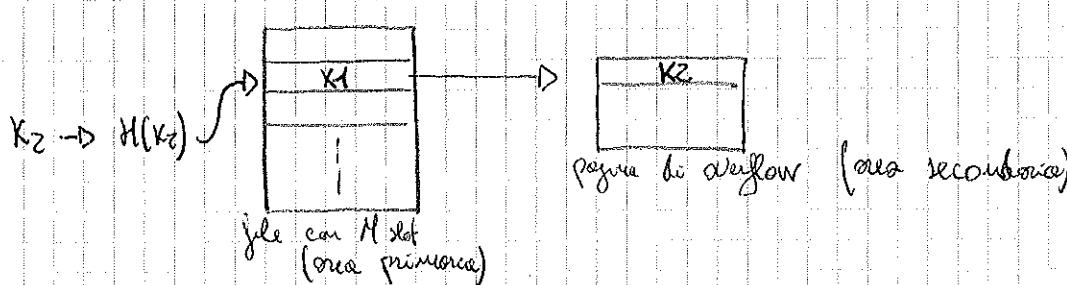
### Variazione 1 del Linear Probing: Liste Conguenti

- ⑤ Vengono utilizzate solo aree primarie organizzate a pagine: su ciascuna pagina si collocano dei puntatori che puntano ad un'altra pagina la quale contiene quindi tutte le chiavi da ordinare in collisione con le quelle menzionate prima

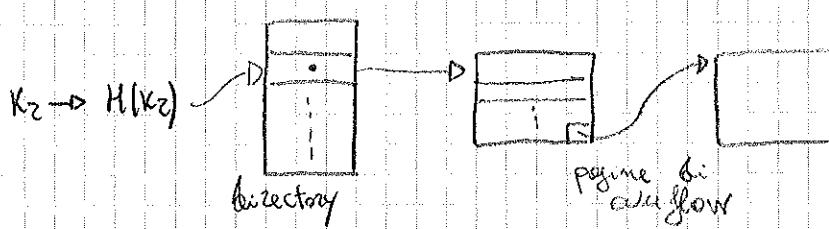
- Vantaggio: può leggere il file senza seguire le catene di overflow
- Svantaggio: il file non cresce dinamicamente visto che i puntatori puntano sempre all'interno dell'area primaria

## Variante Z del Linear Probing: Liste Distinte

- Utilizzo di area primaria e secondaria la quale permette al file di crescere e decrescere. Per ogni slot del file in area primaria si mantiene, quando necessario, una pagina di overflow in area secondaria la quale contiene tutte le chiavi in collisione con quello nullo slot:



- Una degenerazione di questo schema consiste nell'area l'area primaria senza record ma solo con un puntatore alle pagine/e di overflow per ogni slot: il file diventa quindi due directory ad  $M$  entry:



In media la cughetta di ogni coda sarà di  $N/M$  poiché la directory mai contiene chiavi ma solo puntatori, dove  $N$  è il n° di chiavi da inserire.

## Extendible Hashing

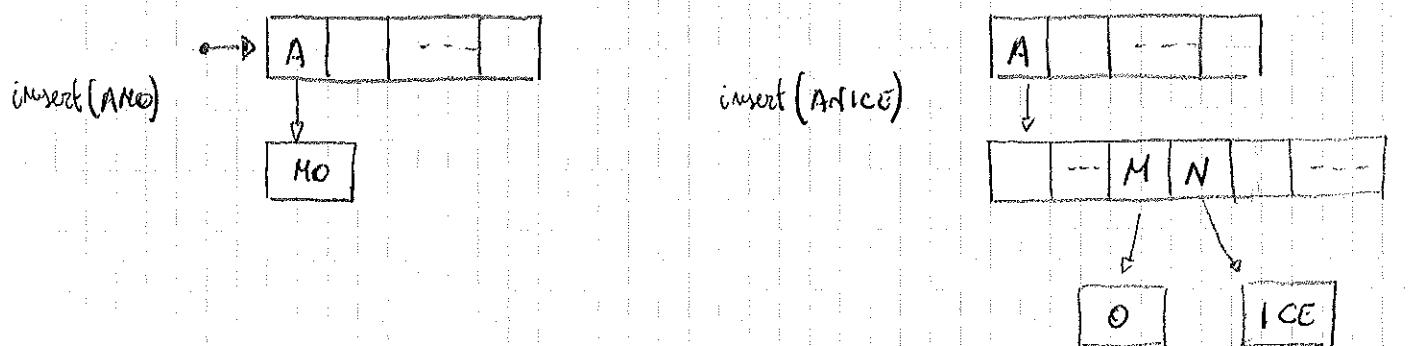
- Proposto per superare i limiti del Linear Probing, permettendo al file di crescere dinamicamente
- Basato sul concetto di Trie (termine derivato da "trieedol")

## Introduzione alle strutture dati Trie

È un albero di grado  $\geq 2$  in cui il branching ad ogni livello è basato solo su una parte delle chiavi di ricerca senza la necessità di effettuare un confronto con chiavi memorizzate come nel B-TREE ad esempio.

### Esempio

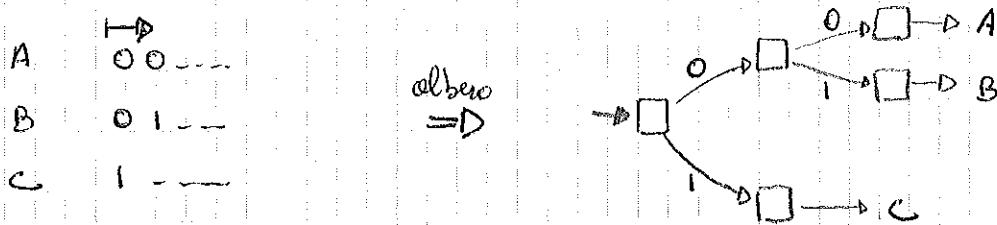
alfabeto di dimensione  $N \sim$  per ogni nodo abbiamo  $N+1$  puntatori se le chiavi sono di lunghezza variabile (il puntatore più vicino all'elenco ha fine delle stesse), altrimenti se le chiavi sono a lunghezza fissa abbiamo  $N$  puntatori.



→ Un albero di questo tipo è particolarmente obsoletato\* ed ha un basso fonsort, per questo nella pratica non viene utilizzato; si usa invece la sua idea di base: discriminazione con parte delle chiavi e senza confronto con chiavi mem.

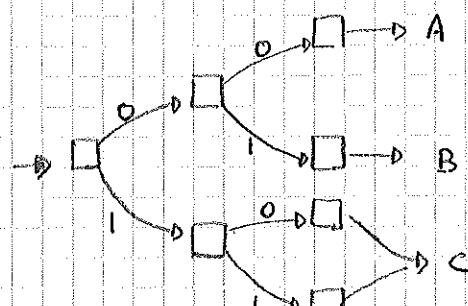
## Trie binario o elicottere

- alfabeto di dimensione 2 (la lunghezza dell'alfabeto corrisponde al max fonsort)
- risulta dunque esso un albero obsoletato con fonsort basso (in effetti il minimo possibile) ma n'è vero successivamente che discende da questa struttura.



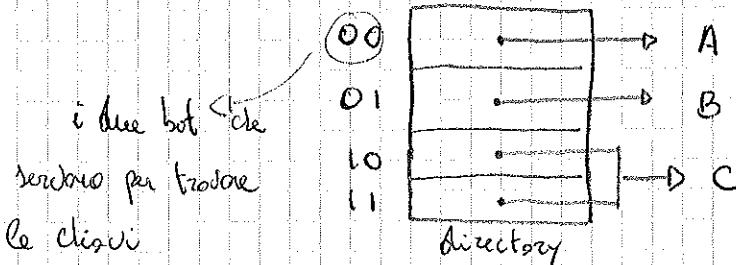
\* questo implica che non si può stabilire una limite superiore per il tempo di accesso alle chiavi.

Variante: bilanciamento dell'albero (tutte le chiavi allo stesso livello)



come si può notare l'albero diventa un albero binario completo bilanciato quindi; tuttavia pagine al tempo per scrivere alla chiave C

Il vantaggio di questa struttura è che permette di eseguire due linearizzazioni



Come si vede la struttura del Tree si compone del tutto.

In queste strutture a directory si ha un puntatore per ogni possibile configurazione degli "i" bit più significativi (da sx a dx) delle chiavi, in questo esempio abbiamo considerato i bit quindi 4 possibili configurazioni ma, in generale, gli "i" bit considerati varieranno a seconda delle chiavi da memorizzare per garantire sempre una corretta individuazione di tutte le pagine; questo implica che il file a directory dovrà espondersi a/contrarsi a secondo dei casi.

[NB]

Nel seguito gli "i" bit per la considerazione per accedere alla directory dovranno quelli meno significativi (da dx a sx), cioè si creeranno dei suffissi comuni delle chiavi anziché altri prefissi; questa scelta è giustificata dal fatto che, così facendo, l'implementazione di tecniche quali la duplicazione della directory (vedi oltre) risulta più semplice.

## Costruzione della funzione hash

Per avere un indice che combini dinamicamente, abbiamo bisogno di una funzione hash il cui range di valori sia in grado di combinare anch'esso dinamicamente; creeremo quindi una funzione hash  $H(K)$  nell'intervallo  $0 \div 2^m - 1$  con  $m >> 0$ , dove

①  $K$  è la chiave di cui vogliamo calcolare l'hash.

②  $m$  è il numero di bit della chiave; più è grande questo valore più la probabilità di non riuscire a risolvere un conflitto diminuisce.

→ All'interno dell'Extendible Hashing ~~tree~~ la funzione hash definita precedentemente ha, di per sé, un valore marginale; serve infatti solo per generare, a partire dalle chiavi, delle configurazioni binarie il più possibile uniformi, riducendo il più possibile lo sbilanciamento dell'albero.

③ Nei casi esemplificati supponiamo sempre che ogni pagina contenga solo un record, nei caso reali quando cerchiamo una ~~pagina~~ chiave ci viene restituita la pagina che l'avrebbe contenuta ma poiché la pagina ci viene ritornata in memoria principale, cercare al suo interno la chiave che vogliamo non comporta costi aggiuntivi.

## Operazioni di modifica della directory

④ Consideriamo le chiavi con le rispettive codifiche binarie restituite dalla funzione hash:

A =	..... 000
B =	.... 010
C =	... 001
D =	.. 110

⑤ Definiamo profondità locale il n° di bit usati per indirizzare una specifica pagina, PL nel seguito.

⑥ Definiamo profondità globale il n° di bit usati per accedere ad una entry della directory, PG nel seguito.

⑦ Dalle precedenti definizioni segue che:

una directory ha esattamente  $2^{PG}$  puntatori (entry).

una pagina è puntata esattamente da  $2^{PG - PL}$  puntatori

## Inserimento

Se le pagine su cui dobbiamo inserire il nuovo record non è sature, esso viene memorizzato e lo procedere terminare, se invece tale pagina è satura (nel nostro caso è sufficiente che essa contenga già un record ma in genere ne potrebbe contenere b più di svariati) allora distinguiamo due casi:

- PL < PG ci sono almeno due puntatori che puntano alla pagina satura (essi sono esattamente  $\geq PG - PL$ ), quindi questa può essere splitata e i dati della stessa ridistribuiti tenendo conto dei bit discriminanti ~ nei nostri esempi non ci sarà mai niente da ridistribuire ~ chi ipotizzaremo sempre che una pagina contiene un solo record; lo split quindi creerà una pagina vuota ed una pagina contenente il record originale tuttavia, nella pratica, la probabilità di avere realmente una pagina vuota a fronte di uno split è molto bassa (vedi OLTRE)

Dopo aver eseguito lo split si tenta di nuovo l'inserimento del nuovo record

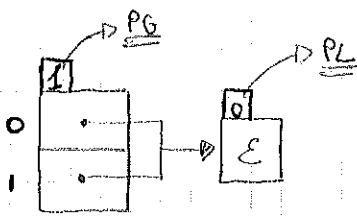
- PL = PG significa che, per la discriminazione, stiamo usando già tutti i bit a disposizione a livello di directory; l'unico alternativo è duplicare la directory facendo admettere la sua profondità globale di 1, questo ci riporta nuovamente nella condizione su cui  $PL \leq PG$  dandoci cioè la possibilità di eseguire un virtuale successivo split con ridistribuzione (solo nella pratica) delle chiavi

Dopo aver eseguito la duplicazione dello directory si tenta di nuovo l'inserimento del record.

[NB]

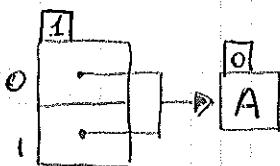
la duplicazione della directory viene fatta solo se la profondità locale della pagina su cui si deve inserire il nuovo record è uguale alla profondità globale e questa è satura

## Esempio



### → insert (A)

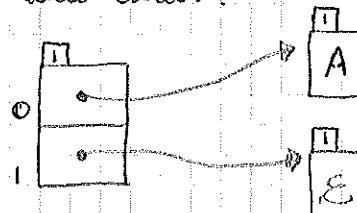
la directory sta discriminando secondo il primo bit meno significativo delle chiavi ma, dato che la pagina è attualmente vuota, qualcosa va. La configurazione di tale bit per la chiave A, essa durerà come mem. in queste pagine (di sotto non c'è nessuna discriminazione in questa fase)



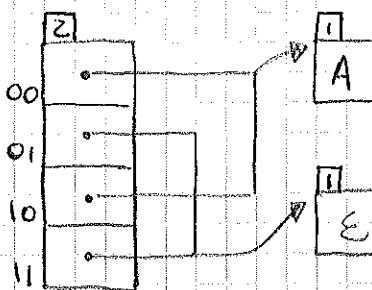
### → insert (B)

la directory discrimina ancora con il primo bit meno significativo delle chiavi (che per B è uno o zero) ma la pagina puntata dall'entry individuata da tale bit è occupata, questo comporta una collisione e, nel nostro caso particolare in cui una pagina contiene solo un record, un conseguente overflow.

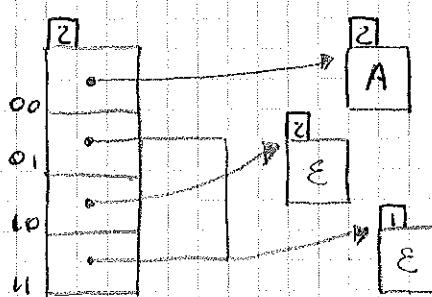
Poiché  $PL < PG$  sappiamo che ci sono almeno due puntatori che puntano alla pagina dotata (ce n'è  $2^{PG-PL} = 2^{1-0} = 2$  esattamente) il che comporta la possibilità di fare uno split della pagina con redistribuzione (solo nella pratica) delle chiavi:



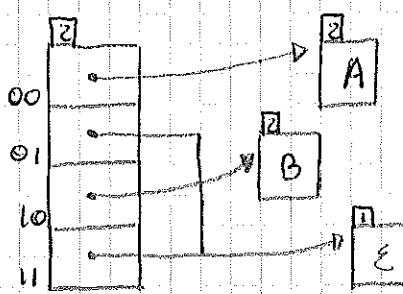
il modo tentativo di inserimento di B porta ancora ad un overflow in quanto la directory continua a discriminare su un singolo bit; in questo caso però abbiamo che PL della pagina in overflow è uguale a PG quindi si può duplicare la directory aumentando il n° di bit discriminatori:



il successivo tentativo di inserimento di B viene effettuato discriminando gli due bit meno significativi della chiave cle, nel caso di B, sono 10. Ancora una volta la pagina in cui dovrebbe essere inserita B è satura ma poiché  $PL < PG$  si può eseguire uno split con redistribuzione delle chiavi (quest'ultima non eseguita nel nostro caso) :



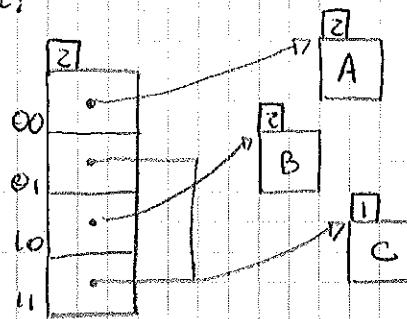
a questo punto l'inserimento di B può essere fatto senza ulteriori problemi poiché la pagina indirizzata dall'entry 10 ha abbastanza spazio (e' vuota nel nostro caso) per contenere un nuovo record ; notiamo che nonostante sia  $PL = PG$  nessuna duplicazione viene eseguita in quanto non vi è dedicato nessun overflow.



→ insert (c)

la directory sta discriminando records i primi due bit meno significativi della chiave cle, nel caso di c, sono 01 ; poiché la pagina indirizzata dall'entry individuata da tali bit è vuota, il record può essere inserito direttamente. La profondità totale di 1 seguente cle ci permette di sufficienze solo uno dei due bit per indirettore

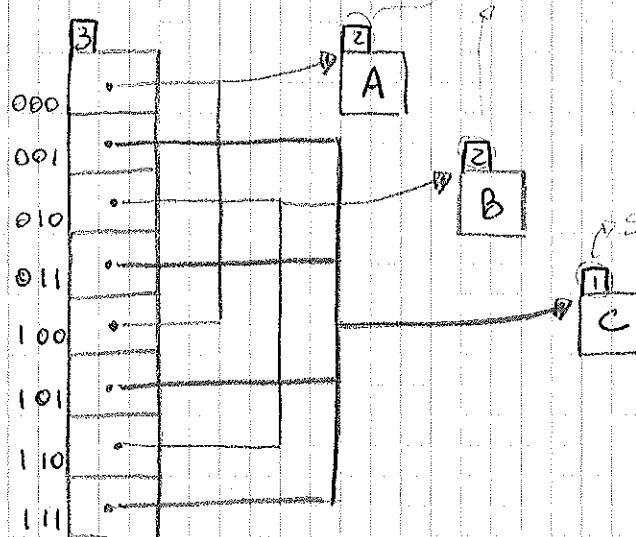
la pagina:



→ insert (d)

i due bit meno significativi dello indice sono 10 ma la pagina indirizzata da essi è saturata; dato che  $PL = PG$  si rende necessaria una riduzione

duplicazione

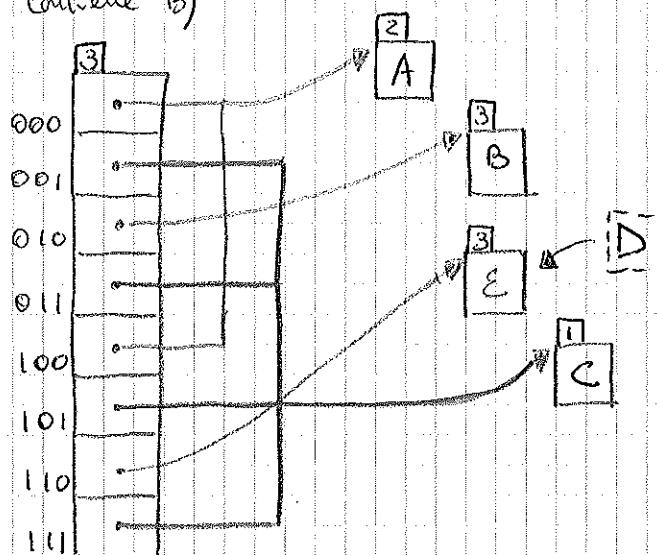


l'entry di pagina è fermo a 2 bit non scindibili, blabla

riducendo alla fine il 1 bit meno significativo, blabla

discriminato sui 3 bit 110

il successivo tentativo di inserimento del record provoca ancora un overflow ma questa volta  $PL < PG$  quindi si può effettuare una split (della pagina che contiene B)



a questo punto l'inserimento del record D potrebbe essere completato senza problemi perché la pagina indirizzata da 110 ha sufficiente spazio per contenervlo.

## Operazioni

- Quando accediamo ad una chia e paghiamo esattamente 2 accessi: uno per trovare il puntatore sullo directory ed uno per seguire il puntatore ed arrivare alla pagina dove le chiavi dovrebbe essere memorizzata. (☞) la scorsa le indicizzazione proposta non dipende dalla dimensione delle directory.
- Nei casi pratici la probabilità di avere una pagina vuota (e quindi spazio) a fronte di un'operazione di split è bassa:

# di possibili casi in cui tutti i record vadano a finire su una stessa pagina:  $2^B$  (o tutti i bit sono a zero o ad 1)

# di tutte le possibili configurazioni di bit, avendo la possibilità di mett.  $B$  record per pagina:  $2^B$

$$\rightarrow P = \frac{2}{2^B} = \frac{1}{2^B}$$

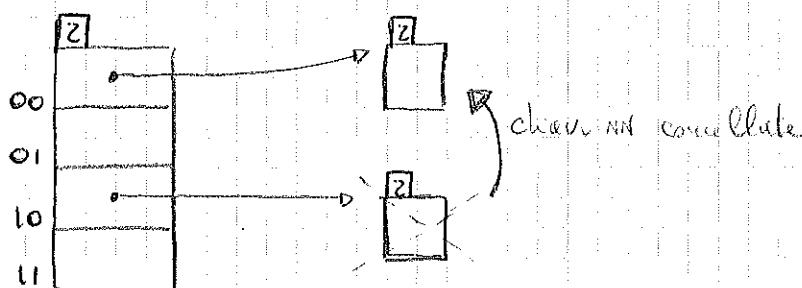
quando  $B$  è abbastanza grande questa prob è molto piccola.

- Dalle tabella 1 pog 100 dell'articolo si nota come al n° totale di puntatori e quindi la dimensione dello directory diminuisca all'aumentare di  $b$ ; dove  $b$  è il fattore di bloccaggio, se noi abbiamo appena 1, della pagina avremo il n° di record per pag, mentre  $M$  è il # di record nel file.

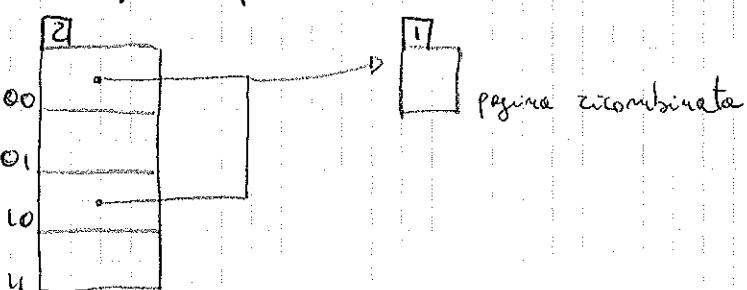
## Cancellazione

- L'idea è quella di ricombinare le pagine soggette ad una cancellazione con le loro pagine SIBLING; la ricombinazione è l'operazione inversa dello split e può essere effettuata se il # tot. di record nelle pagine ricombinate non supera le dimensioni delle pagine stesse.
- Il # di SIBLING di una pagina dipende dalla sua profondità locale:

- PG = PL ~ esiste una sola pagina SIBLING, essa è individuata dall'entry nella directory che ha i PG-1 bit meno significativi uguali a quelli dell'entry che indirizzerà la pagina da cancellare ed il restante bit a zero o al 1



per cancellare le pagine puntate da 10 si deve verificare se essa può essere ricombinata con la sua unica pagina SIBLING che è quella puntata da 00



- PG = PL + 1 ~ i PG-2 bit meno significativi sono uguali a quelli che indirizzano la pagina da cancellare ed i restanti due bit individuano i 4 SIBLING esistenti per la pagina beta. (VEDI USANDO ESEMPIO ROTATIVO ALL'INSERIMENTO, PAG. PRECEDENTE)

- PG = PL + 2 ~ i PG-3 bit meno significativi sono uguali a quelli che indicizzano la pagina da eliminare ed i restanti tre bit individuano gli 8 SIBLING esistenti per la pagina beta

○ e così via.

**[NB]** Una directory può essere bimette quando TUTTE le pag. da esse puntate hanno  $PL < PG$ ; semplicemente si elimina la 2<sup>a</sup> metà della directory

riguardo bibliografico non obbligatorio su queste tecniche;

"Extendible Hashing - A fast access method for dynamic files"

FAGIN, STRONG et al.

ACM TODS 4:3 1979

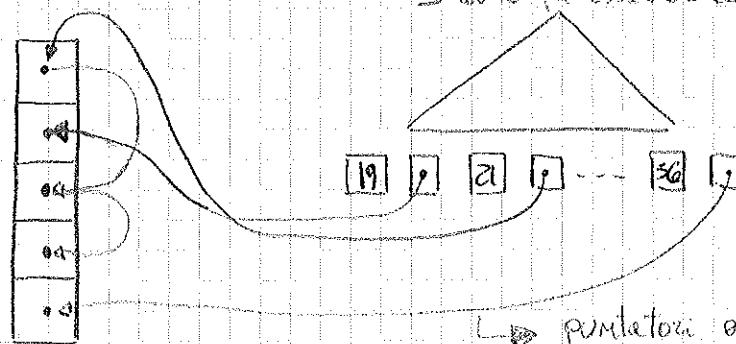
## Organizzazioni di indici per chiavi secundarie

- questi tipi di indici servono solo per velocizzare le operazioni perché tipicamente nelle chiavi secundarie non c'è il circolo di chiavi ~~alla~~ ~~chiave~~ e quindi possiamo recuperare più facilmente gruppi di tuple con un particolare valore di chiave (secundaria)
- essi sono non clusterizzati. Cioè la coppia  $[k_i, r_i]$  non mantiene il record dati in  $r_i$  ma un puntatore ad esso.

### Esempio

CIA	Dati
1	21
2	19
3	21
4	21
5	56

File Dati (records)



Indice su chiavi Secundarie

→ puntatori alle tuple delle liste

### Svantaggi

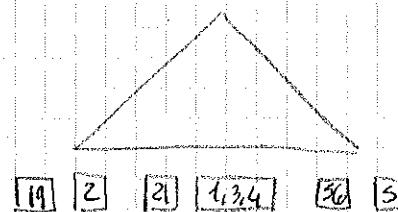
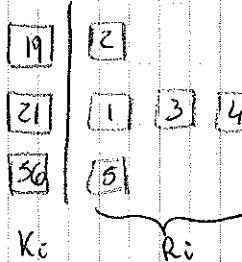
- ① Per creare o eliminare un indice si deve re-processare l'intero file dati in modo da aggiornare o cancellare un campo che mantiene tutti i puntatori che legano i valori uguali della chiave su cui l'indice è definito
- ② Poiché i record sono bloccati a pagina non c'è modo di controllare come essi dirigono acceduti, cioè è possibile che si acceda più volte alla stessa pagina in modo non consecutivo danneggiando così ogni accesso
- ③ Supponiamo di aver 2000000 righe query del tipo: "tutti gli impiegati

che hanno  $\text{et}\ddot{\text{e}} = 30$  e  $\text{stipendio} = 2000$ " e supponiamo anche di avere due indici su chiavi secundarie: uno definito su  $\text{et}\ddot{\text{e}}$  e l'altro su  $\text{stipendio}$ . Quello che succede è che non possiamo dare entrambi gli indici contemporaneamente, ma si deve sceglierne uno  $\Rightarrow$  se si sceglie di processare l'indice  $\text{et}\ddot{\text{e}}$  si ricava un certo numero di record sui quali è già possibile applicare l'altra condizione sullo stipendio. Quindi si può dare al punto un indice ormai se ne abbiamo definiti 1.

### Organizz. di Liste Invertite

- ① queste organizz. è stata proposta per la prima volta nell'info. retrieval.
- ② si crea una lista di questo tipo facendo su di essa la registrazione di diversi liste di riferimento a record su cui compone il codice di chiave  $K_i$ :  $[K_i, \{P_0, P_1, \dots, P_n\}]$

Esempio n. relativo al precedente campo  $\text{et}\ddot{\text{e}}$ :



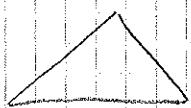
### Vantaggi

- ① "indipendenza" dai file dati: quando si crea o cancella un indice non si deve modificare il file dati
- ② i puntatori possono essere facilmente ordinati in modo che accensi ripetuti allo stesso peggio siano fisicamente contigui per accedere allo stesso slot due volte
- ③ si possono usare più indici in quanto l'esecuzione delle operazioni viene fatta sullo stesso record ma utilizzando delle semplici operazioni binaristiche:

- ④ per fare un AND  $\Rightarrow$  intersezione di liste } le diverse liste relative agli indici introdotti
- ⑤ per fare un OR  $\Rightarrow$  unione di liste
- ⑥ per fare una ~~NOT~~  $\Rightarrow$  differenza tra liste

## Problema con gli indici non clusterizzati: concetto di puntatore

indice  
clusterizzato



indice non clusterizzato



indice non clusterizzato

→ Se abbiamo un indice clusterizzato ed uno o più indici non clusterizzati, come nel caso mostrato, poiché il file è dinamico se accade che split di pagina ad esempio, un puntatore nell'indice secondario che puntava alla pagina prima dello split va opportunamente gestito  operazione non banale perché essa comporterebbe di dare comandi a tutti i puntatori di tutti gli indici secondari.

### Soluzioni:

- (1) Nel file clusterizzato per gestire i dati, si potrebbe usare ad esempio una pila così il vantaggio che i puntatori resterebbero sempre gli stessi ma con lo svantaggio di non poter accedere all'indice con costo minimo secondo l'ordine di elenco
- (2) Usare dei puntatori logici: cioè si usano degli indirizzi fisici logici associati ai record ed una tavola di mapping ID-Logico  $\rightarrow$  Fisico del record:

è logico (nell'indice secondario)

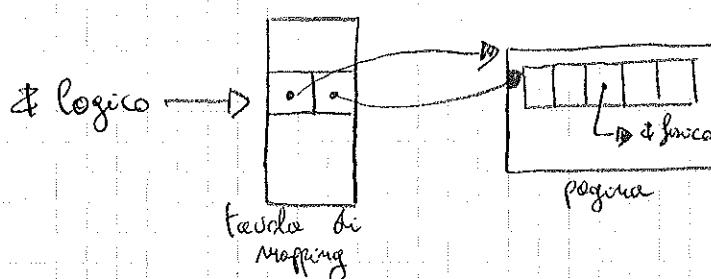
↓  
tavola di mapping

↓  
è fisico del record

in questo modo i puntatori dell'indice secondario rimangono sempre gli stessi inoltre, quando i record dell'indice primario si spostano

si devono aggiornare solo le entry della tavola di indirezione corrispondenti ai record spostati e non tutti i puntatori di tutti gli indici secundari come nel caso ~~precedente~~ in cui non si usa nessuna strategia di soluzione a questo problema.

- Notiamo che la tavola di mapping introduce una indirezione tra i puntatori nell'indice non clusterizzato ed ~~tra~~ i record nell'indice clusterizzato, vale a dire che ci paga un accesso in più (costo di accesso alla tavola) ogni volta che desideriamo recuperare un record.
- Per ridurre il n° di aggiornamenti della tavola si può montare due doppie indirezioni con cui puntatore alla pagina che contiene i record ed un altro che punta ad un'altra tavola di indirezione menu. direttamente all'interno della pagina e che ha il compito di restituire gli & fixci dei record menu. nella stessa: l'aggiornamento dei record all'interno della stessa pagina avviene ora localmente alla stessa senza ne modificare la tavola di indirezione principale, né essere accessi aggrediti



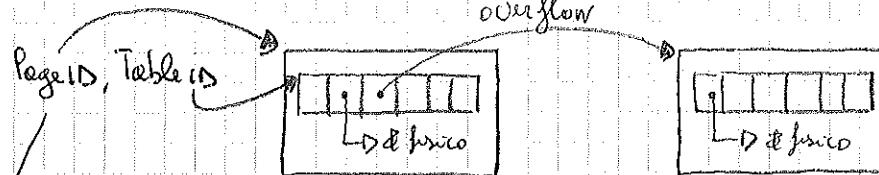
Una limitazione di questo schema consiste nell'usare la chiave primaria come puntatore logico anziché una tavola di indirezione, questo solo se la chiave ha un valore numerico (come tavola di indirezione viene usato lo stesso indice clusterizzato)

### ③ Usare dei puntatori fixci: file differenziati

Poiché le modifiche ai record vengono fatte solo sul differenziale il problema del puntamento indice-record → indice.primario non esiste più: i record nel file master non cambiano mai

### ④ Usare dei puntatori fixci: Catene di overflow

Queste tecniche consentono di mantenere dei puntatori fixci ma, nel momento in cui le tuple si spostano si ricongiungono delle catene di overflow:



e l'overflow della pagina in cui si trova il record che stiamo cercando.

- Come si vede quindi, viene mantenuta una tavola di indirizzamento all'interno delle pagine, essa restituirà o l'overflow di un record menù, nella stessa oppure un overflow di un'altra pagina in cui il record è stato spostato.
- Questa tecnica, adottata in SYSTEM R, ha senso solo per file molto grandi in cui le operazioni di scrittura sono molto rare.

- ① La scelta di mantenere o meno un indice secondario dipende da molti fattori come ad esempio il # di record per pagina, il # totale di pagine, etc. ma dipende anche dalla distribuzione dei colori su cui l'indice deve agire, ad esempio: se abbiamo un indice sul sesso ed il 99% degli impiegati sono uomini mentre solo l'1% sono donne, allora conviene mantenere l'indice per le donne e quello degli uomini comporterebbe comunque l'accesso alla quasi totalità delle pagine; se la distribuzione fosse stretta del 50% e 50% allora l'indice sarebbe utile per entrambi i gruppi.
- ② In generale quindi, se il processing di un indice secondario ha una performance pari a quella degli scorri sequenziali, cioè comporta l'accesso a tutte le pagine del file, non conviene mantenerne l'indice perché, rispetto agli scorri sequenziali, paghiamo in più il costo di mantenimento dello stesso.
- ③ Dato un file dati di  $N$  records in  $P$  pagine, egnate contenente  $B = N/p$  record (B è il fattore di bloccaggio = # di record per pagina) selezioniamo casualmente  $K$  records: quante pagine distinte tocchiamo? Questo calcolo ci può aiutare a capire se conviene mantenere l'indice oppure no:

N.B.: ci interessano solo le pagine distinte che essendo esse forse inoltre contigue occidono più volte allo stesso pagina, piuttosto che costare nulla.

## Formulo di Yeo (non richiesta all'esame)

$$\text{Se } K \leq N-B \quad \phi(P, B, K) = P \cdot \left(1 - \frac{\frac{K}{B}}{N-i+1} \cdot \frac{N-B-i+1}{N-i+1}\right)$$

$$\text{Se } K > N-B \quad \phi(P, B, K) = P$$

## Approx di Cordenas (richieste)

- ignora il fattore di bloccaggio quindi introduce un errore
- presuppone una selezione dei record con rimpiazzamento, permettendo cioè che si possa accedere ad un record più volte

$$\phi(P, K) = P \cdot \left(1 - \left(1 - \frac{1}{P}\right)^K\right)$$

### Dimo

- probabilità che una data pagina contenga uno dei record referenziati dalla lista invertita:  $\frac{1}{P}$
  - probabilità che una data pagina non contenga una specifica registrazione:  $\left(1 - \frac{1}{P}\right)$
  - Supponendo tutte le registrazioni indipendenti, la probabilità che la pagina non contenga nessuna delle  $K$  registrazioni:  $\left(1 - \frac{1}{P}\right)^K$
- Quindi il valore  $1 - \left(1 - \frac{1}{P}\right)^K$  esprime la probabilità che una specifica pag. contenga almeno una delle  $K$  registrazioni; se poi molti plichiamo questo valore per il # tot. di pagine  $P$ , otteniamo il # di pag. totali alle quali dobbiamo accedere.

- il costo totale dell'operazione è: (VALO ANKI PER APPROX SDD)



B tree ~ index set

processamento  
delle liste

$h +$  lunghezza lista invertita ( $K$ ) in pagine +  $\phi(P, K)$   $\rightarrow$  # di pag. su cui andranno  
aggiornati i link

$K$  può essere espresso come  $K = \frac{LR}{NOd}$  ~ Lunghezza media (in elementi) della lista di  $K$   
 $LR$  # di valori Distinti di una classe  $K$

$\rightarrow$  # di elementi nella  
liste di  $K$

se  $K \geq 2P$

$$\phi(p, K) = P \quad (\text{caso peggiore}) \quad \left. \begin{array}{l} \phi(p, K) \text{ è la lunghezza} \\ \text{media delle liste create} \\ \text{che, per una clausa, è} \\ \text{in pagina} \end{array} \right\}$$

se  $\frac{1}{2}P < K < 2P$

$$\phi(p, K) = \frac{1}{3}(K + P)$$

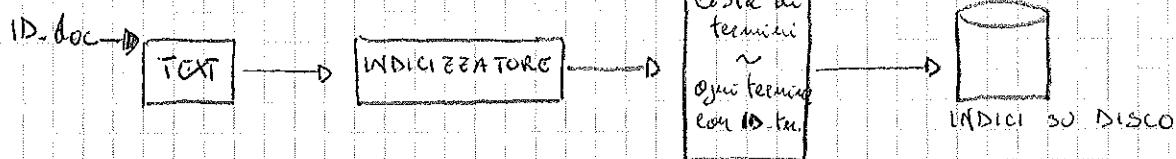
se  $K \leq \frac{1}{2}P$

$$\phi(p, K) = K$$

→ Motiviamo che nel momento in cui la lunghezza media delle liste inserite è tanto grande da superare il n° di pagine nel file, l'accesso attraverso l'indice non clusterizzato è inutile e dannoso rispetto l'esecuzione di uno scan regolare completo; è dannoso che paghiamo  $P + il$  raggiungimento della lista,  $h + il$  suo processamento (VEDI STIMA PRECEDENTE)

### Applicazioni nei motori di ricerca (Text Retrieval)

Schema generale:



intuisci:

- ① di ogni termine c'è una lista di documenti che lo contengono [termine, {ID-doc-0, ID-doc-1, ..., ID-doc-n}] per ridurre la dimensione degli indici ed admettere l'estensione delle risposte si usano delle particolari tecniche come lo STOPPING: si considera il "teme" della parola e non la parola *in se*, così delle parole AMO si memorizza solo ANI per restituire nella risposta anche le parole AMO, AMERO, AMIRO, etc.
- ② Naturalmente bisogna considerare sempre il contesto *sia* se stiamo cercando la parola ANO, utile come strumento per la pesca, e ci viene restituita una lista di riferimenti a parole quali ANGRO, etc. L'utente *non* resta sottoposto del motore di ricerca.

### Lista dei termini

- ① ogni termine ha un ID-termine
- ② le liste rapp. *ogni* riferimento del documento (si eliminano ad escluso tutti gli articoli o le stopwords)

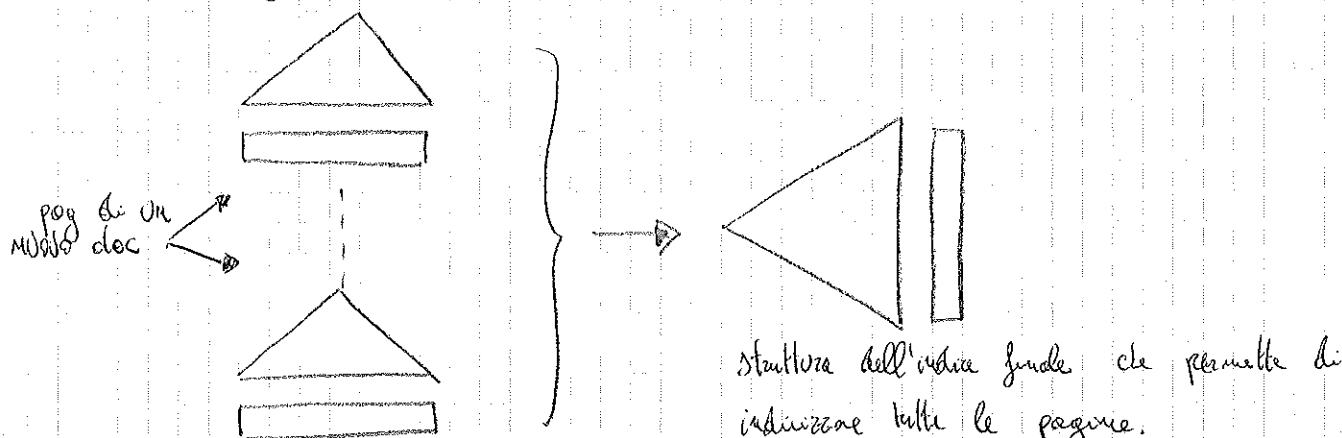
riperimento bibliografico sui motori di ricerca: WITTEN, MOFFAT, BOYD  
"Managing Gigabytes"

### Varianti introdotte dai sistemi moderni (Google)

- per migliorare la relevantza delle risposte si fa un peso ai documenti nella base di molti fattori, ad esempio la frequenza dei termini, il # di pagine che riportano lo stesso termine, etc.; in questo modo i primi riferimenti restituiti sono quelli che più si addicono alle nostre richieste
- interrogazioni per prossimità e adiacenze (in google è la ricerca tra "termini") : le parole tra "debbono essere fisicamente contigue nel documento".

### Sull'information Retrieval

- l'indicizzazione di un documento è molto costosa visto per ogni parola si deve individuare la corretta lista invertita ed il suo interno le posizioni esatte in cui inserire l'id del documento
- Una soluzione a questo problema, applicabile se l'indice non è stato ancora creato, è quella di mantenere in memoria dei piccoli alberi di indirizzamento per contenere le pag. del documento da aggiungere e quindi creare l'indice con la stessa struttura di memoria

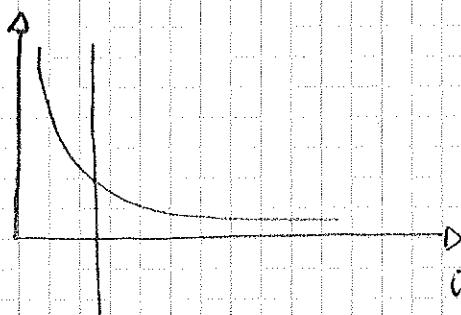


creare l'albero di indicizzazione in questo modo ha un costo di molto inferiore rispetto l'inserimento partiale di ogni termine. Nel caso un indice esiste già si possono creare delle strutture di alberi in memoria, per mantenere le modifiche all'indice, esse saranno periodicamente fatte con quelli globali in mem. secondaria.

## Distribuzioni non uniformi

### Leye di Zipf

- esse descrive tutta una serie di leggi che si trodano in natura, ad esempio
- la distribuzione delle zucche ~ pochi hanno molto e molti hanno poco
  - la distribuzione della popolazione a poche megalopoli, molti paesi.
  - la distribuzione delle parole nei testi
  - etc.



$$P_i = \frac{C}{i} \quad i=1 \dots N$$

C è il fattore di normalizzazione, vale  
Molto armonico di ordine 1:

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N} = \sum_k \frac{1}{k}$$

$$\approx \ln N + \gamma \quad \text{dove } H_N \text{ è l'N-esimo}$$

costante di Euler  $\approx 0,577$

### Leye 80/20 (viene verificata empiricamente nel caso delle distribuz. di accesso ai dati)

- l'80% delle transazioni accede al 20% dei dati; è una legge vicinica per cui il 64% delle transazioni, ad esempio, accede al 4% dei dati  $\Rightarrow$  se vissiamo a mantenere un memoria il 4% dei dati caldi, vedremo il 64% delle transazioni (buonissime per la gestione di accesso)

- Esiste una generalizzazione di questa legge in grado di modellare tutte le distribuz. che vengono dalla uniforme alla Zipf

$$P_i = \frac{C}{i^{(1-\Theta)}} \quad , \quad C = \frac{1}{H_N^{(1-\Theta)}} \quad \text{dove } H_N \text{ è l'N-esimo Molto}$$

armonico di ordine  $1-\Theta$   $\Rightarrow H_N^{(1-\Theta)} = \sum_i \frac{1}{i^{(1-\Theta)}}$

Distrib. Uniforme

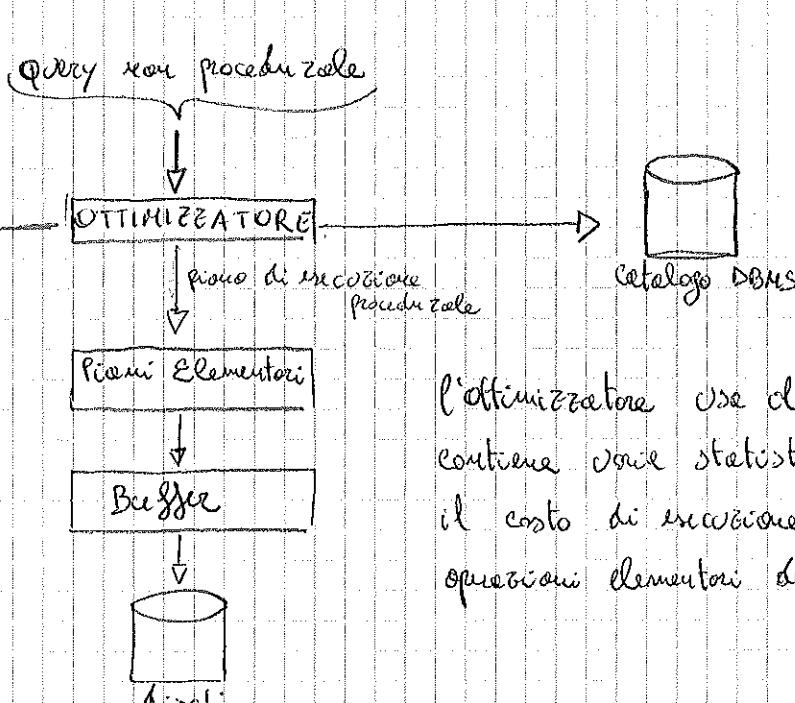
(tetra)

$$\left\{ \begin{array}{l} 1 \\ \log \frac{8}{\log 2} \\ \frac{1}{2} \end{array} \right.$$

80/20

ZIPF

## Ottimizzazione delle query



l'ottimizzatore usa il catalogo del DBMS, il quale contiene varie statistiche, per riuscire a stimare il costo di esecuzione della query utilizzando operazioni elementari dell'algebra.

### Statistiche contenute nel catalogo

#### ① Per ogni relazione $R$

- $|R|$
- $P_R$
- $N_a(R)$  = # attrib. di  $R$
- $L(R)$  = dim. in byte di una tupla

#### ② Per ogni attributo $a \in R$

- $NDV =$  # di valori distinti di  $a$ ; può essere mantenuto con un indice
    - o stimato con un filtro di Bloom
  - $L(a)$  = dim. in byte di  $a$
  - $\text{Min}(a)$
  - $\text{Max}(a)$
- } valore min e max di  $a$ , nel caso in cui esso sia numerico

#### ③ Per ogni indice definito su $R$

- $|PI| =$  # pag o livello soglia
- $h =$  altezza dell'albero
- $NDV =$  # di valori distinti (può essere diverso dall'NDV di un attributo)

## Selezione

- assumiamo l'indipendenza degli attributi ed una distribuzione uniforme dei valori (introducendo un errore nella stima della correttezza della selezione)

$$G_{A=0}(R) = \frac{1}{NDV_A} |R|$$

- se  $NDV$  è ignoto, si usa una stima
- il rapporto  $1/NDV_A$  è chiamato selettività della selezione

- Fattore di selettività,  $g_s$ :

$$g_s(A > 0) = \frac{\text{MAX}(A) + 0}{\text{MAX}(A) - \text{MIN}(A)}$$

per selezioni di tipo  $A > 0$

$$g_s(A < 0) = \frac{0 - \text{MIN}(A)}{\text{MAX}(A) - \text{MIN}(A)}$$

per selezioni di tipo  $A < 0$

$$g_s(v_1 \leq A \leq v_2) = \frac{v_2 - v_1}{\text{MAX}(A) - \text{MIN}(A)}$$

per selezioni di tipo  $v_1 \leq A \leq v_2$

Se gli attributi su cui fare la selezione non sono numerici si utilizzano delle esistenze.

## Utilizzo dei più indici secondari per l'esecuzione dei predicitivi complessi

$$G_{A \wedge (B \cap C)}(R)$$

- ① se abbiamo due indici, uno su  $A$  ed uno su  $B$ :

$$A \wedge B \Rightarrow R = \text{lista\_inv}(A) \wedge \text{lista\_inv}(B)$$

$$A \vee B \Rightarrow R = \text{lista\_inv}(A) \cup \text{lista\_inv}(B)$$

- ② se abbiamo solo un indice, ad esempio su  $A$ :

$$A \wedge B \Rightarrow R = \text{lista\_inv}(A), \text{ accesso ai records e lista\_inv di } B \text{ sui record di } A; \text{ questo vale perché } A \wedge B \subseteq A \wedge A \wedge B \subseteq B$$

C'indice su  $A$  ci porta quindi in una situazione di gravi risolti che dobbiamo contro il residuo in  $B$ .

$A \vee B \Rightarrow$  l'indice su A non può essere usato perché  $A \vee B \subseteq$  chiuse.

Ci ritroviamo in una situazione di Non-risolto.

→ Per risolvere tutti i casi, si creano delle tabelle con il seguente significato:

○ R(l<sub>ox</sub>) è la lista incertita relativa alla selezione alla destra del predicato (AND oppure OR) da risolvere [situazione RISOLTA]

○ S(l<sub>ox</sub>) è una lista incertita di candidati relativi alla selezione alla destra del predicato (AND oppure OR) da risolvere; su tale lista si dovrà uno poi unificare delle condizioni residue [situazione SEMI-RISOLTA]

○ N è una situazione di Non-risolto in cui cioè non è possibile usare gli indici [situazione NON-RISOLTA]

AND  
P<sub>Sx</sub>      P<sub>Dx</sub>

sinistra ↓

(AND)	R(l <sub>ox</sub> )	S(l <sub>ox</sub> )	N
R(l <sub>Sx</sub> )	R(l <sub>Sx</sub> ∧ l <sub>Dx</sub> )	S(l <sub>Sx</sub> ∧ l <sub>Dx</sub> )	S(l <sub>Sx</sub> )
S(l <sub>Sx</sub> )	S(l <sub>Sx</sub> ∧ l <sub>Dx</sub> )	S(l <sub>Sx</sub> ∧ l <sub>Dx</sub> )	S(l <sub>Sx</sub> )
N	S(l <sub>ox</sub> )	S(l <sub>ox</sub> )	N

destra →

↳ nessuno dei due operandi ha un indice

OR  
P<sub>Sx</sub>      P<sub>Dx</sub>

sinistra ↓

(OR)	R(l <sub>ox</sub> )	S(l <sub>ox</sub> )	N
R(l <sub>Sx</sub> )	R(l <sub>Sx</sub> ∨ l <sub>Dx</sub> )	S(l <sub>Dx</sub> ∨ l <sub>Sx</sub> )	N
S(l <sub>Sx</sub> )	S(l <sub>Sx</sub> ∨ l <sub>Dx</sub> )	S(l <sub>Dx</sub> ∨ l <sub>Sx</sub> )	N
N	N	N	N

destra →

→ Non si può usare l'indice se un operando per fare l'OR

↳ nessuno dei due operandi ha un indice

## Esempio

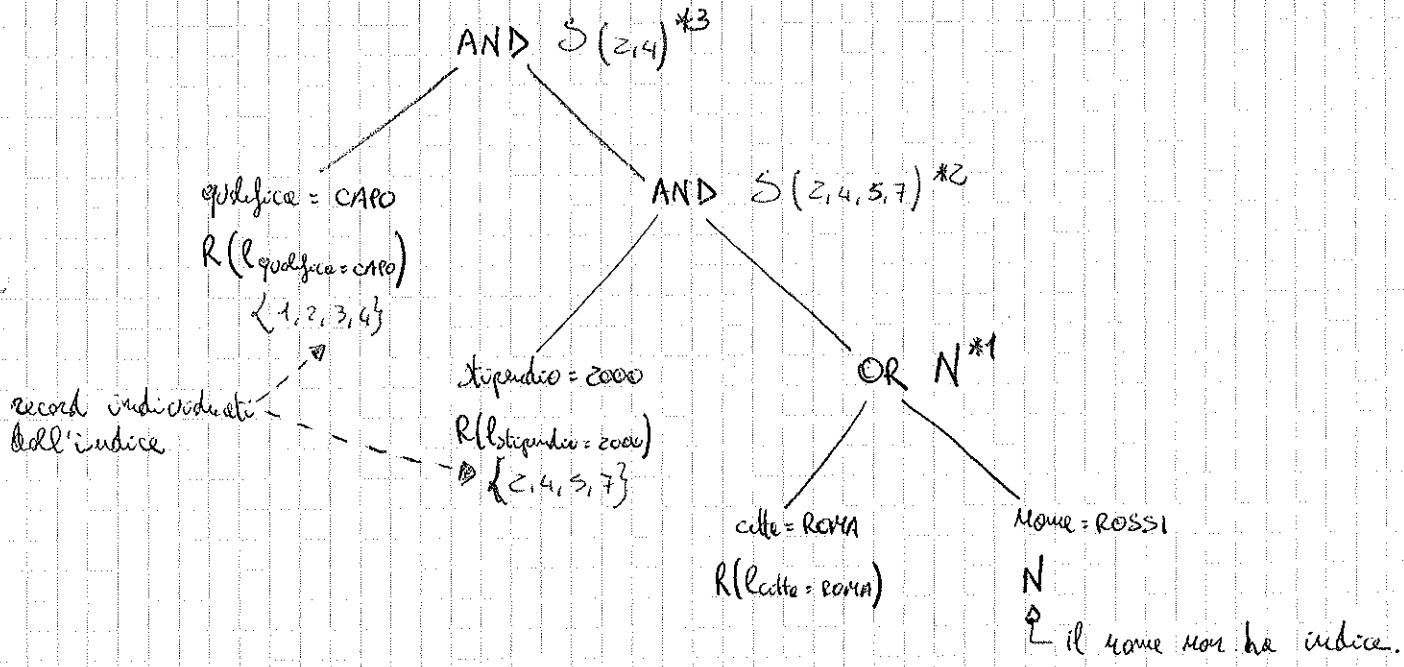
```

SELECT *
FROM IMPIEGATI
WHERE qualifica = CAPO AND stipendio = 2000 AND (città = ROMA OR nome = ROSSI)

```

con indici definiti su tutti gli attributi ad eccezione del nome.

## Albero delle condizioni



- (\*) ~ delle tabella per l'OR tra R(lsx) ed N a destra. L'indice su città non può essere usato; notiamo inoltre che, se l'interrogazione fosse stata solo questo pezzo avremmo dovuto risolverla sequenzialmente ma, dato che ci sono altri predicati questo pezzo verrà risolto successivamente (come residuo) sugli interi relazioni IMPIEGATI ma solo su quei parti di esse.
  - (\*) ~ delle tabella per l'AND tra R(lsx) ed N nella destra.
  - (\*) ~ delle tabella per l'AND tra R(lsx) ed S(lsx), la relazione divise in tipo SEMI-RISOLTA sull'interazione delle due liste invertite; poiché non ci sono altri predicati da calcolare, nella lista risultato dell'interazione si dovranno avere a controllare il predicato residuo che riguarda le due condizioni, sulla città oppure sul nome, ~~ma anche~~ valida.
- Notiamo che i residui vengono controllati su ogni sottoinsieme della relazione da partenza.

## Esempio

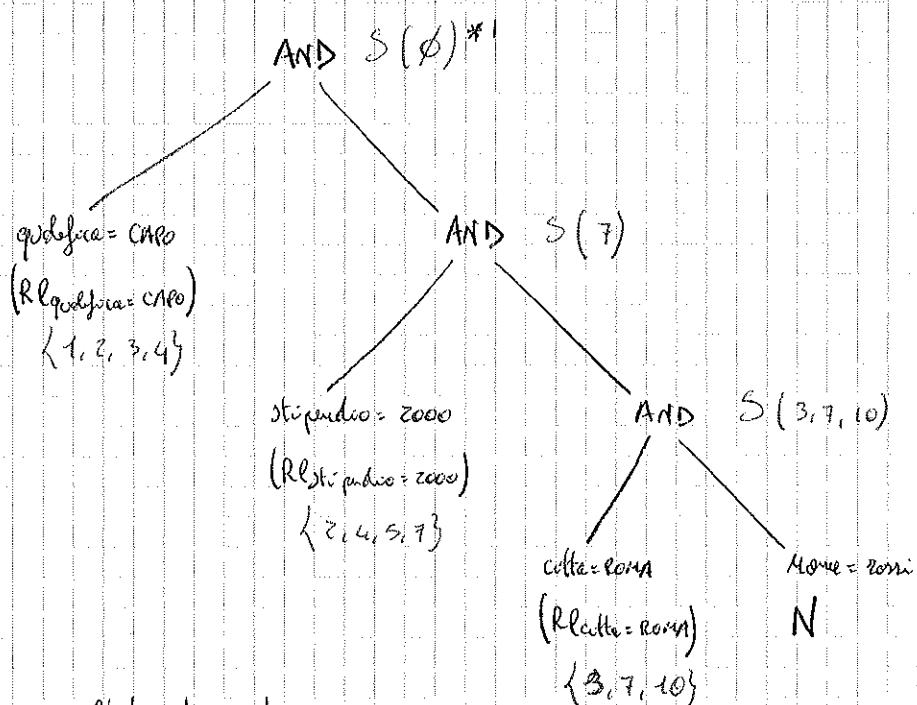
SELECT \*

FROM impiegati

WHERE qualifica = CAPO AND stipendio = 2000 AND (citta = romma AND nome = loris)

con indici definiti come nel caso precedente.

Albero delle combinazioni



(\*) → nessun risultato trovato

## Risultati delle operazioni

- ① gli indici secondari May sono sempre utili infatti ci possono essere casi in cui essi non possono essere usati. (problema dell'OR); inoltre i risultati sono sufficientemente selezionati, cioè piccolo, May concerne continuare a processare le altre liste nel tentativo di rispondere ulteriori accensi (riducendo ulteriormente le liste), concerne invece fermarsi e processare tutto il resto come residuo. Anche se questo caso quando ci possono essere degli indici secondari che May danno uno più usati.
- ② se un indice ad un certo livello produce una lista veritiera abbastanza piccola, May concerne continuare a processare le altre liste nel tentativo di rispondere ulteriori accensi (riducendo ulteriormente le liste), concerne invece fermarsi e processare tutto il resto come residuo.

## Join

$$RRIS = \frac{1}{NDR.e} \cdot \frac{1}{NDVs.e} \cdot |R| \times |S| \cdot NDR_{join}$$

→ misura (NDVs.e, NDR.e)

ci possono essere degli errori di stima nei risultati

## Velocizzazione dei metodi di join

### Metodi di join

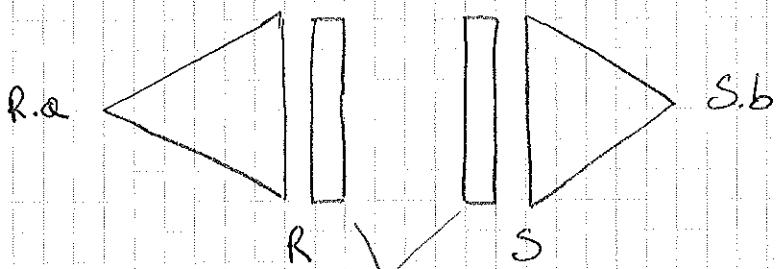
- basati su scan sequenziali (NL, NB, NS)
- basati sull'ordinamento (MS)
- basati sul partitionamento hash (hashjoin)

→ si possono usare degli indici per accelerare l'esecuzione

### Ad esempio per il NL

per ogni tupla di R si fa una selezione delle tuple di S (quelle con lo stesso valore dell'attributo da join), si può usare un indice che ci rimanda direttamente alle tuple di S con un particolare valore di join (ponendo h per attraversare l'albero + uno scan sequenziale della lista)

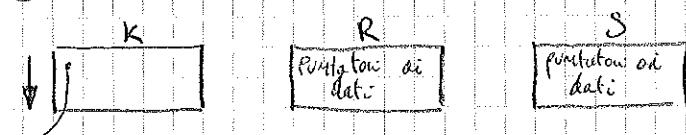
se abbiamo due indici nella stessa chiave (uno di R ed uno di S), si possono usare delle tecniche di merge:



i sequence-set sono ordinati secondo l'ordine di chiave, si può applicare un MS ad esempio.

### Indice di join

- è un indice su due relazioni che possono essere messe in join.
- è forte di un valore di chiave abbinato a liste invertite (una per relazione) e debole cioè due puntatori, uno ad R, che punta a una volta a tutte le tuple che contengono quello specifico valore di join ed uno ad S, che punta a sua volta a tutte le tuple che contengono quello specifico valore di join.



→ valori dell'attributo di join da comparare in R oppure S

[NB]

bisogna verificare se effettivamente esore l'indice di ogni ottimizzazione l'esecuzione rispetto ad una scan sequenziale; il join infatti viene precomputato sotto forma di lista invertita, quindi è possibile che si debbano toccare emg tutte le pag. per ottenere i dati.

### Dettagli nell'ottimizzazione delle interrogazioni

Fasi

- ① analisi strutturale delle query
- ② trasformazione (vedi regole di trasformazione dell'algebra relazionale)
- ③ ottimizzazione (creazione di un piano di costo minimo)
- ④ esecuzione del piano

le fasi 3 e 4 non sono necessariamente disgiunte, si potrebbe infatti scegliere di adottare un'ottimizzazione interattiva

→ Analisi

- ① controllo correttezza lessicale & semantica sintattica
- ② controllo correttezza semantica (esistenza degli attributi, entità esistenti necessarie per eseguire la query, etc.)
- ③ ~~normalizzazione~~ delle condizioni (ad esempio  $\varphi \wedge \text{TRUE} \Rightarrow \varphi$  oppure  $\varphi \wedge \text{FALSE} \Rightarrow \text{FALSE}$ , etc.)      ~~# SIMPLIFICAZIONE~~
- ④ normalizzazione delle condizioni (forma normale congruente)
- ⑤ eliminazione predicati contraddittori (ad esempio  $A > 20 \wedge A < 18$ )
- ⑥ eliminazione del NOT su condizione semplice (ad esempio  $\neg A > 20 \Rightarrow A \leq 20$ )
- ⑦ generazione dell'albero logico creato per rapp. l'algebra relazionale

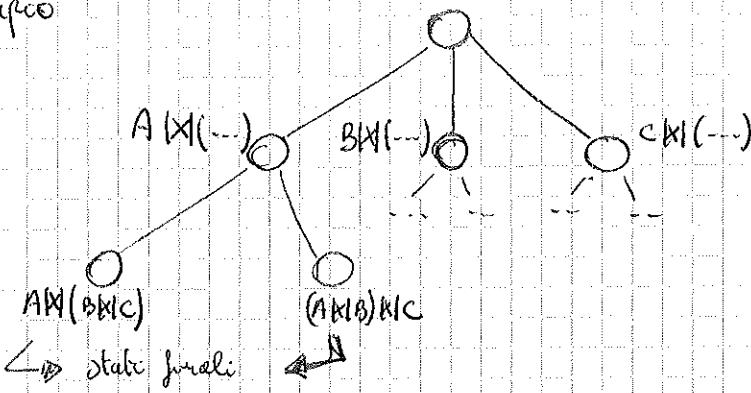


OTTIMIZZATO RU

### Ottimizzazione Esauriente

- proposta inizialmente in SYSTEM-R
- se una ricerca va a finire per trovare il miglior percorso di ottimizzazione, c'è una tecnica che genera tutti gli stessi finché quando sceglie il risultato migliore  $\Rightarrow$  costo eccezionale

Esempio



con  $N$  selezioni di posizione si hanno  $\frac{(N-1)!}{(N-1)!}$  permutazioni di joint

ad esempio con 5 selezioni si hanno 1600 permutazioni, con 8 selezioni, 18 milioni di permutazioni.

Un'alternativa all'utilizzo di un'ottimizzazione esauriente consiste nell'utilizzare delle heuristiche; gli obiettivi generali di un'heuristica sono:

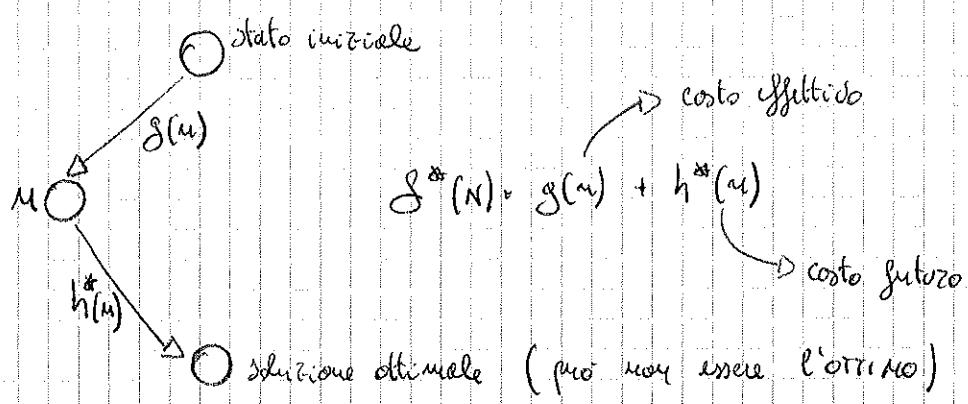
- ① dato un nodo, determinare i suoi successori
- ② dato un nodo, determinare quale dei suoi successori esplorare

### Possibili heuristiche

- ① Greedy → ad ogni passo scelgo il percorso di costo minimo; non si può garantire l'ottimo globale ma solo locale
- ② Alberi bilanciati a  $\infty$
- ③ Algoritmo A\* → è un algoritmo generale di ottimizzazione che applica tecniche heuristiche ed esauriente
- ④ etc.

### Algoritmo A\*

- ① Ad ogni passo si determina il costo di una soluzione che posso per ogni nodo,  $f^*(n)$ ; tale costo è dato dal costo della sequenza di passi (di operazioni) che ci hanno portato al nodo corrente + il costo di due sequenze ottimali (caso di costo minimo) che ci porterà ad un risultato finale ottimale di operazioni
- ② è un algoritmo ricerca dell'ottimo in modo "esauriente" ma più intelligentemente di un normale BFS → le considera anche delle heuristiche come la condizione di ammissibilità: tanto più  $f^*(n)$  è vicino al costo reale, tanto più alta è la priorità.



- si costituisce una lista di nodi da espandere ordinate per  $g^*(N)$  crescente e si seleziona per l'espansione il nodo che ha il minimo valore di  $g^*(N)$

### ④ CONDIZIONE DI AMMISSIONE (è un'heuristica \*)

se  $h^*(u) \leq h(u)$  Vu il nodo finale cui giungiamo è OTTIMO  
 $\hookrightarrow$  costo futuro ≤ costo effettivo

- ⑤ questo algoritmo tende ad avere una ricerca in ampiezza (buone penetrazioni) infatti non meno che le stime  $h^*(u)$  diventa sempre più approssimata, minori costi di espansione sempre più bassi per l'espansione.

#### CASI ESTREMI

- $h^*(u) = h(u)$  → penetrazione massima, ricerca molto calore ma profonda-
- $h^*(u) = 0$  → penetrazione minima, l'unica cosa che accade è che il costo effettivo sarà > di 0 !

### ⑥ Variante

Definire delle tolleranze ammissibili  $\Rightarrow$  trovare soluzioni che ~~non~~ devono dall'ottimo ma con un costo vicino ragionevole

- ① si calcola una soluzione greedy  $\rightarrow$  costo massimo
- ② si esplorano i vari nodi e ci si ferma quando un  $g^*(N)$  trovato è nella tolleranza della soluz. greedy trovata precedentemente.

si può usare il metodo TAO → "Truly Adaptive Optimization", vedi articolo (non obbligatorio) "TAO: The basic Ideas" G.M. SACCÀ, DGAS 2006, LNCS 40:80

- ⑧ ~~esistono~~ si eliminano sottoinsiemi di soluzioni che potrebbero contenere l'ottimo.  
 (impone la possibilità di

Il metodo TAO consiste in un raffinamento successivo di una soluzione heuristica fino al momento in cui la soluzione trovata non rientra nelle tolleranze di stabilità.

NB le tolleranze determinano la qualità del risultato

cioè: tutte le volte che esploriamo un nodo, calcoliamo una soluzione heuristica da quel nodo ed una soluz. finale, essa sarà il costo minimo accettabile per tutto il sottoalbero del nodo. Ci si ferma quando

$$\frac{HC}{g^*(u)} \leq 1 + \Theta \quad \text{per il nodo } u \text{ con } g^*(u) \text{ minimo}$$

Migliore heuristica calcolata ~~finora~~ fino a questo momento

- se  $\Theta = 0$   $\Rightarrow$  non c'è tolleranza  $\Rightarrow$  cogliamo una soluzione ottima
- se  $\Theta = \infty$   $\Rightarrow$  c'è una tolleranza  $\infty$   $\Rightarrow$  la prima soluz. trovata ci va bene.

## Heuristic Decomposition

"A Strategy for query processing"

Wong, Younghi

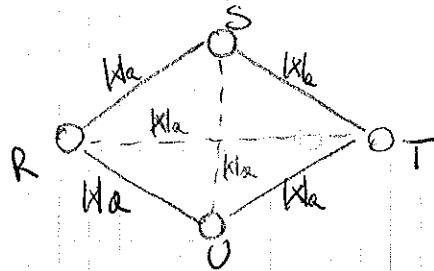
ACM TODS vol 1:3 1976 (non obbligatorio)

Questa heuristica è basata su Nested Loops; si parte da una rappresentazione come grafo della query in cui i nodi sono zappi. delle relazioni e gli archi, tra due nodi, zappi. Un zappo tra le corrispondenti relazioni, se uno specifico attributo:

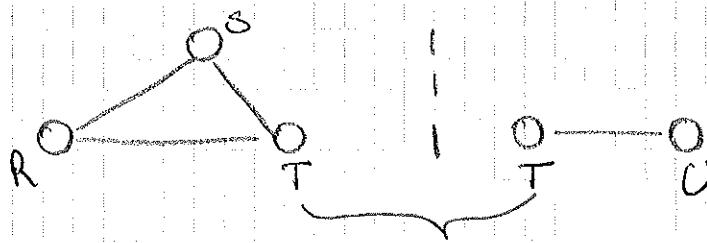


→ si usa questo tipo di zappi. x decidere il fatto che serve di operazione molte volte convenienti, come ad esempio tra S ed T in cui, lo stesso nome di un arco indica la necessità di fare fare un prodotto cartesiano per poter mettere le relazioni.

Un altro vantaggio dell'utilizzo di questi zappi, consiste nella possibilità di completare i join tra relazioni non direttamente collegate ma che condividono gli stessi attributi su quali tra loro e altre relazioni intermedie:



Dopo aver creato il grafo, esso viene decomposto in componenti connesse in modo da separare le singole ottimizzazioni:



→ bisogna stabilire il sequencing a livello di componenti, cioè decidere quale eseguire prima:

~~①~~  $R \bowtie S \bowtie T \rightarrow T'$  ed essere  $T'$  il punto delle  $T$  in  $R \bowtie S \bowtie T$

②  $R \bowtie S \bowtie T \rightarrow T'$  ed essere  $T'$  il punto delle  $T$  in  $T' \bowtie U$

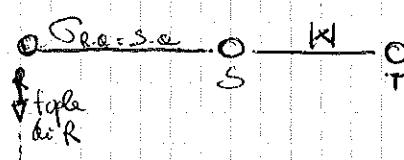
Ottenute le singole componenti, si posso all'ottimizzazione di ciascuna di esse

#### Criterio: TOPIC SUBSTITUTION

consiste nel selezionare la relazione con cardinalità minima di una componente e considerarla non più come un'variabile ma come un'insieme di query di cui i predicati delle stesse funzionano da predicati di selezione ogni sola e sceglie la relaz. più esterna di un nested-loop;

ad esempio per la componente  $R \bowtie S \bowtie T$

approcciamo di scegliere  $R$  come rel esterna: per ogni tupla di  $R$ , bisogna fare di joiin tra  $S$  e  $T$ :



→ notiamo che scegliendo  $R$  come esterna si impone che tutti i joiin siano del tipo  $R \bowtie \dots$ , eliminando tutti i possibili punti d'esecuzione con  $R$  interna.

Dopo le tuple substitution restano delle componenti non completamente connesse quindi l'algoritmo può essere applicato ricorsivamente. Notiamo che esso è un algoritmo edistico perché non si prendono in considerazione tutte le soluzioni, ad esempio nel caso precedente non si può considerare  $(S \bowtie T) \bowtie U$  perché non c'è modo di ottimizzare il join fra S,T ed U in quanto il join S  $\bowtie$  T fa parte di una componente connessa che include anche R e non può prescindere da essa.

L'ottimizzazione può avvenire più soff.: si può eseguire un passo di query e ricalcolare il piano utilizzando i valori non pur stimati ma redatti e prodotti dall'esecuzione precedente.

Problematiche relative al disegno fisico

→ Scelte trasparenti alle applicazioni su cui si può operare:

### 1. INDICI

- ① non usare indici se il % di pag delle rel. è troppo basso o se la seleattività > 20% (se tramite un indice non clusterizzato accediamo a più del 20% delle tuple, stiamo praticamente toccando tutte le pagine dell'indice quindi l'indice non serve)
  - ↳ DELLA REL.
- ② non usare indici se le modifiche sono frequenti, in particolare definire pur di 4 indici nello stesso rel. SOLO se le modifiche sono molto rare
- ③ considerare le strategie di sollecitazione, cioè soltanto se l'indice deve effettivamente usato oppure no (problema dell'OK e della dominanza di un indice sull'altro)
- ④ considerare la distribuzione dei dati evitando di usarli in distribuzioni non uniformi
- ⑤ utilizzarli per istruzioni di ordinamento o join anche se non hanno una seleattività adatta.

### 2. STRUTTURA (B<sup>+</sup>-TREE, Hashing)

#### 3. BUFFER

- ① aumentare e diminuire la dim. del buffer
- ② mettere il buffer in mem. secondaria
- ③ non eseguire transazioni di report in periodi critici (problema del double reversal con lock globale senza destinazioni)

- ④ "inviatore" le relazioni in memoria (esse non vengono salvate secondo le tecniche tradizionali, l'accesso è molto veloce ed a basso costo)

## 14. OTTIMIZZAZIONE

### 1. PARALLELISMO

### 2. CAMBIO DBMS

Scelte non trasparenti le quali cioè richiedono una ristrutturazione dello schema logico del DB e quindi impattano sulle apppl. che debbono usarlo; si considerano due aspetti: (in conflitto tra loro)

- ① TEMPO: tutti e solo i dati manipolati insieme devono stare fisicamente insieme e ad esempio, men. i risultati di una query in una sola relazione (solo materializzata) (aumento della ridondanza)

- ② SPAZIO: ogni dato è men. in un unico posto (minimizzazione della ridondanza) e quello che faccio le tecniche di normalizzazione nei db, esse infatti tendono a minimizzare lo spazio

## Minimizzazione del tempo ( $\rightarrow$ aumento dello spazio occupato, ridondanza)

esse si ottiene con tecniche di partizionamento delle relazioni:

- ① VERTICALE: la relazione su N attrb. viene operata in un numero M di relazioni più piccole, ciascuna contenente un sottoinsieme degli attributi della prima + la chiave primaria.

CASO ESTREMO: "db column-oriented"; ogni sotto tab. contiene un solo attributo + la chiave primaria

Nell'ottica dei DB distribuiti ogni sotto tab. sta su un altro diverso ed è costituita solo da attributi di cui quel unto ha bisogno per lavorare

- ② ORIZZONTALE: la relazione di partenza viene spezzata in sottosezioni di tuple; ogni sotto tab. ha gli stessi attrb. di quelli originali ma contiene dati diversi in base al fatto che questi debbono essere osati

La minimizzazione del tempo, per ottenere un processing più veloce, implica quindi la presenza di campi pre-calcolati (ad esempio il campo "STIPENDIO MAX")

di un impiegato a fronte di un dipartimento) che invece dovrebbero esistere e l'elemento, in genere, delle sovrapposizioni; ad esempio

ID	---	ID dip
Impiegato		

ID	Nome	---
Dipartimento		

se volemmo sapere qual'è il nome del dipartimento cui appartiene un impiegato dovremmo fare un join fra le due relazioni; se invece replicassimo nella rel. Impiegato il nome del dipartimento, queste info. arrivano immediatamente disponibile (sulla base necessaria di fare un join)

### denormalizzazione



ID	---	ID dip	Nome dip	ID	Nome	---
Impiegato						

ID	Nome	---
Dipartimento		

ultimo caso di modifica dell'info. replicata, bisogna modificare in tutte le rel. in cui essa è presente. (mentre nel primo caso è sufficiente una sola modifica)

### Minimizzazione dello spazio (non necessaria mente implica un conseguente aumento dei tempi)

essa può essere ottenuta tramite:

#### ① codifica di "entità sostanziale"

è esattamente l'operazione inversa della denormalizzazione che ci riporta, nell'esempio precedente, al caso in cui il nome del dip. è messo solo nella rel. Dipartimento; notiamo che, se la rel. Dipartimento è abbastanza piccola da stare in memoria, otterremo un ottimale vantaggio (anche del punto di vista dei tempi che si riducono!!)

#### ② compressione

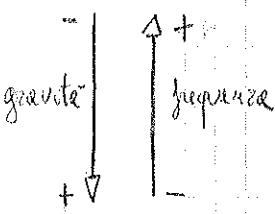
permette di evitare i campi ologrammici e lunghezza fisse, altrimenti per memorizzare "ROMA" e "NOME DI UNA CITTÀ LUNGHISSIMO" pagheremmo lo stesso costo.

## Trasazioni

Vediamo le proprietà ACID:

- ① Atomicità ~ se la transazione termina correttamente (istruzione di tutto) oppure viene abortita e tutte le modifiche fatte fino a quel punto hanno effetto (istruzione NIENTE)
- ② Coerenza ~ assumendo che la transazione eseguita isolatamente faccia pensare al DB da uno stato consistente ad un altro, anche se eseguite concorrentemente ovverrà lo stesso.
- ③ Isolamento ~ gli effetti delle transazioni sono visibili solo dopo il termine della stessa.
- ④ Durability (persistenza) ~ se T termina normalmente (istruzione TODO) i suoi effetti sono persistiti anche innanzi a malfunzionamenti

## Malfunzionamenti



1. ABORT ~ terminazione prematura della transazione; può essere esplicito: da parte del programmatore che rilascia una istruzione anomala, ad esempio un prelievo da un conto con saldo nullo.  
implicito: da parte del sistema a causa di un deadlock oppure delle mancanze di autorizzazioni, etc.
2. FALLIMENTI ~ interruzioni del funzionamento del sistema di cui la menu, permanentemente sopravvive ma quelle temp. no, ad esempio crash del S.O., mancanza di corrente, etc.
3. DISASTRI ~ la menu. permanente non si provvede, ad esempio perdita di un disco o dell'intero sistema

## Gestione dei malfunzionamenti

- ☒ per i DISASTRI, poiché le cose sono fisiche, l'unica cosa che si può fare è replicare su più siti fisici, distanti ormai geograficamente, il contenuto dei dischi
- ☒ per ABORT e FALLIMENTI esistono diverse tecniche:
  1. Copia di Backup (DUMP) dei dati (vole anche per i disastri come snapshot sopra) ~ in caso di malfunzionamento si perdono solo le modifiche effettuate dopo il DUMP

2. LOG LIBRO-GIORNALE il quale mantiene, in mem. stabile, alcune info. se cosa fa una certa transazione; la prevenzione dei disastri avviene tramite una tecnica di multiplexing del log, cioè esso viene riportato su più collocazioni fisiche distinte come per i dischi.

Info morteante dal log:

- ① ID transazione
- ② Operazioni
  - ↙ connir ~ terminazione normale della transazione
  - ABORT
  - MODIFICA
- ☒ Before Image (BIM) ~ serve per disfare le transazioni
- ☒ After Image (AIM) ~ serve per recuperare transazioni andate a finire (sempre necessaria per il recupero da DISASTRI)

Il problema con l'utilizzo di un log è che, per far fronte ad un malfunzionamento, non sappiamo quanto indietro bisogna andare nello stesso per recuperare le info. di cui abbiamo bisogno → Vogliamo sfruttare efficacemente il log per evitare di riportare ogni volta dall'ultimo DUMP ~ Cold Start (vedi OLTRE)

per ottimizzare l'utilizzo del log si usa l'operazione di checkpoint la quale serve per sincronizzare la mem. secondaria; questa op. viene eseguita tipicamente ogni 3 minuti (mentre un DUMP può essere "scelto" anche di due settimane) e garantisce che dal momento in cui viene eseguita al momento dell'ultimo DUMP il DB è consistente

Esistono due tipi di checkpoint

COMMIT CONSISTENT: nel momento in cui si fa un commit consistent tutte le nuove transazioni vengono messe in attesa, per quelle già attive si attende la loro terminazione, si scorre il buffer sulla mem. stabile e si scrive la morca di checkpoint sul log.

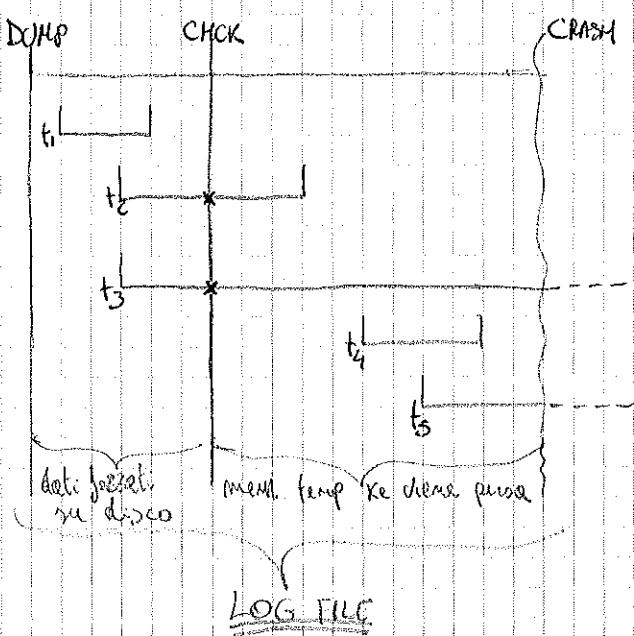
Tutte le modifiche da apporre al DB vengono effettuate una volta. Questo tipo di checkpoint avviene praticamente a sistema fermo.

BUFFER CONSISTENT: nel momento in cui si fa un buffer consistent tutte le nuove transazioni vengono messe in attesa, si scorre il buffer sulla mem. stabile e si scrive sul log la morca di checkpoint + l'insieme delle transazioni attive nel momento in cui abbiamo fatto il checkpoint.

⊕ VANTAGGIO: coste di meno di un commit consistent perché non si deve aspettare lo fine delle transaz. attive.

⊖ SVANTAGGIO: le transazioni attive, riportate sul log, potrebbero non arrivare a finire fine lasciando il DB in uno stato non perfettamente allineato.

Ripresa da fallimenti ~ WARM START (con checkpoint buffer consistent)



intuitivamente: si desidera bisognare tutte le transazioni che non sono andate a finire fine ( $t_3$  e  $t_5$ ) e rigore "tutte" quelle che voleva sono andate a finire fine  $\rightarrow$  se non avessimo fatto il checkpoint avremmo dovuto rigore tutte le transaz. a partire dalla morca dell'ultimo DUMP ( $t_1, t_2, t_4$ )

utilizzando il checkpoint

⊕  $t_1$  non si deve rigore perché ~~non è stata eseguita~~ essa è stata forzata in mem. secondaria nel momento del checkpoint

⊖  $t_2$  e  $t_3$  sono in una situazione simile ma processando il log in avanti (operazione di roll-forward, vom oltre)

troviamo il commit per te quindi la prima parte di questo log dove viene eseguita xdo essa e successivamente (il checkpoint) andata a buon fine : si riforma solo il pezzo mancante di questa eccezione l'alter image ; per quanto riguarda t<sub>2</sub>, anche la sua prima parte è stata fatta in memoria stabile ma essendo questa non andata a buon fine (nel log del momento del checkpoint al crash non c'è un commit per t<sub>2</sub>) si dovranno disfare tutte le modifiche già apportate (sono le before image).

O t<sub>2</sub> e t<sub>3</sub> devono essere riportate correttamente.

Nella stessa situazione illustrata precedentemente, se abbiamo avuto due checkpoints correnti consecutivi : avremo dovuto effettuare la memorizzazione di t<sub>1</sub> e t<sub>2</sub>, quindi lavorare sulla mem. secondo t<sub>1</sub>, t<sub>2</sub>, t<sub>3</sub> ; mentre t<sub>2</sub> e t<sub>3</sub> nebbiano stesse misse in allineamento.

Come si componete il commit corrente nel tempo in cui accade un crash nell'intervallo di tempo in cui si sta eseguendo la transazione, già attesa nel momento del checkpoint, che sarà ? (caso di t<sub>2</sub>)  
(se le transazioni vengono eseguite una dopo l'altra, si parla solo t<sub>2</sub>)

### Algoritmo

pero di roll back ~ procedimento di indietro del log per creare una lista di transazioni committute , Lc ;

se n' trova nel log :

- commit t<sub>i</sub> , Lc += t<sub>i</sub> (caso di t<sub>1</sub> e t<sub>2</sub>)
- record di modifica per t<sub>i</sub> & Lc , disjo trasmite BMS (caso di t<sub>3</sub>)
- checkpoint {Lat} , La = Lat - Lc ~ lista transaz. da abortire  
Lat inizio delle transaz. attive al momento del checkpoint
- start t<sub>i</sub> , La -= t<sub>i</sub>
- Lc =  $\emptyset$  , fine

nel nostro esempio

$$Lat = \{2, 3\}$$

$$Lc = \{4, 2\}$$

$$La = \{3\}$$

passo di roll forward o processamento avanti del log (a partire dal checkpoint)

se si trova nel log:

- a) record di modifica su  $t_i$  e  $t_{i+1}$ , infine  $t_i$  diventa l'after image
- b) commit  $t_i$ ,  $L_c = t_i$
- c)  $L_c = \emptyset$ , fine

### Ripresa dei fallimenti a CARD START

④ si riporta dell'ultimo DUMP

⑤ bisogna determinare quali transazioni sono da rigore, altro verso l'ADM della memoria del DUMP ci dà dati, mentre non è necessario rigore quelle che non sono andate a finire fine richiedendo queste iniziate dopo il DUMP, riportando da esso non hanno avuto alcun effetto sulla mem. secondaria.

Possi

- a) riportare del DUMP
- b) roll back: costruire lista  $L_c$
- c) roll forward: rig.  $t_i \in L_c$

### Politiche di recupero da abort / fallimenti

#### El Politica di modifica

- ⑥ POSTICIPATA: scrittura delle modifiche solo dopo il commit  
(NON DISFARO) quindi non abbiamo bisogno di rigore le transaz.  
(NO BIM) in caso di problemi.
- ⑦ ANTICIPATA: si ha la possibilità di scrivere le modifiche sulla mem. recordaria prima del commit (ad esempio è il bugger manager a decidere di effettuare una scrittura indip. dalla transazione). In questo caso siamo costretti a disfare le transazioni perché non abbiamo la garanzia che la mem. recuperata non sia stata modificata

## ■ record di commit

① **POSTICIPATO** = l'indicazione che ti è andata a buon fine viene scritta, nel log, solo dopo che le modifiche apportate da ti sì sono state vere persistenti. Non ci necessario eseguire ti ciò cosa di problemi perché le sue modifiche saranno già presenti nel DB.

② **ANTICIPATO** = l'indicazione che ti è andata a buon fine viene scritta, nel log, quando questo è vero ma indipendentemente dall'allineamento delle modifiche apportate da ti con il DB. Queste politiche viene scelte da chi è il gestore del buffer cioè, tipicamente, deciderà quando scrivere su disco le pagine; quindi è possibile che le pag. modificate da ti siano scritte sul buffer ma non allineate con quelle del disco, nonostante ti abbia committato.

Sulla base di queste due politiche si possono definire tutte le possibili comituti:

- ① DISFARE / RIFARE ~ tipicamente usata nei sistemi moderni che supporta al meglio il buffer
- ② DISFARE / NON RIFARE
- ③ NON DISFARE / RIFARE
- ④ Non DISFARE / NON RIFARE ~ (vedi OLTRE per dettagli su questa politica)

Per evitare di perdere le modifiche di cosa di abort/fallimenti:

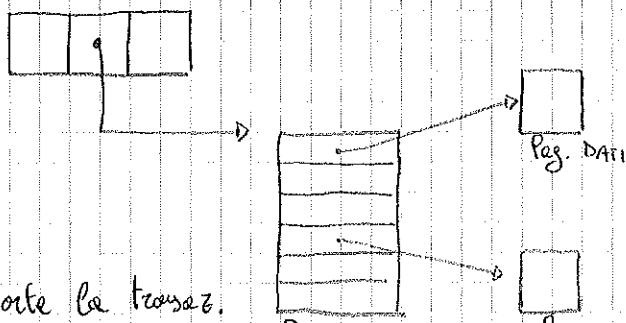
■ Modifica anticipata => protocollo WAL ~ Write Ahead Log, si modifica il log prima di scrivere le modifiche sulla mem. recenderà

■ Commit anticipato => protocollo commit-rule, tutte le modifiche sono riportate nel log prima che ti termini

## Politica NON DISFARE / NON RIFARE

Con questa politica il commit avviene di operazione atomica che se posso il DB ha uno stato in cui non è influenzato dalle operazioni di tu, ad uno stato in cui avviene **TUTTO** le modifiche apportate da tu sono assimilate dallo stesso.

tecniche delle pagina ombra (Loiret, 1977)



Quando porta la transaz. essa fa una copia delle page table nel suo spazio privato; se deve modificare una pagina dati, moy cambia quelle originali ma crea una copia delle pag. nel suo spazio e modifica quella; la page table copia si detta d'acqua da quella originale (cambiano i puntatori quando si creano e modificano nuove pagine). Quando si avvia al commit:

- ① si eseguono una serie di flush del buffer per forzare le pag. in mem. secondaria
- ② si esegue un flush delle page table per forzare anche questa su disco
- ③ si cambia il puntatore che puntava allo page table originale a questo è il momento, per il DB, in cui si posse da una situazione in cui moy si conoscono le modifiche, al momento in cui si conoscono tutte.

Questa tecnica moy viene usata nella pratica più realizzante nel caso in cui, su grandi DB, le transaz. accedano ad un minimo committo di pagine.

## Schedule o storia S di N transazioni

Operazioni:  $R_i(x)$ ,  $W_i(x)$ ,  $C_i$ ,  $Q_i$

dove, ad esempio,  $R_i(x)$  vuole indicare che la transazione  $i$  legge l'oggetto  $x$ , (per cui degl.  $i$  cosa si intende per oggetto VEDI OLTRÉ) ci è  $C_i$  stesso per commit della transazione  $i$  ed abort della transazione  $i$ , rispettivamente.

Esempio

$T_1$ :

$R_1(x)$

$x \leftarrow x + 10$

$W_1(x)$

$C_1$

$\Rightarrow$  Schedule :  $R_1(x) W_1(x) C_1$

$\hookrightarrow$  Le op. di modifica non avranno effetto sulla schedule il cui compito è quello di "tracciare" le transazioni attive mostrando una sequenza di operazioni che le stesse eseguono.

$\rightarrow$  In generale, secondo  $N$  transazioni, sono possibili N! di questi schedules.

Obbligatorio bisogno di dare proprietà di isolamento, infatti consideriamo la schedule:

$R_1(x) W_1(x) R_2(x) W_2(x) C_2 Q_1$

Legge il dato  $x$   
modificato da  $T_1$

$\hookrightarrow$  Le modifiche fatte da  $T_1$  vengono annullate, implica quindi  $Q_2$

produce uno stato inconsistente perché  $T_2$  si prende da uno stato prodotto da  $T_1$  le quali non avranno un buon fine  $\rightarrow T_2$  dovrà essere disfatta

### Soluzione (al problema dell'isolamento)

Una storia  $S$  è recoverabile (recuperabile) se nessuna  $T \in S$  committerà finché tutte le  $T' \in S$  che hanno modificato un obj  $x$  che  $T$  legge non hanno committuto  $\sim$  (Non bisogna disfare il commit)  $\rightarrow$  (Una transazione che committerà, non può essere disfatta)

In questo modo  $T_2$  dovrà aspettare prima di fare il commit: se  $T_1$  abbandona  $\rightarrow T_2$  deve abbandonare, se  $T_1$  committerà  $\rightarrow T_2$  può comittere

$\circ$  può verificarsi che abort by consensus:

se  $T_1$  fa abort anche tutte le altre transaz.  
dovranno farlo



Lettura dei dati modificati dalla  $T_i$  precedente

## Selezione (al problema dell'abort in cascata)

Una storia  $S$  è NON CASCADING ROLL BACK cioè immagine dell'abort in cascata se ogni  $T \in S$  pre cassa solo valori modificati da transazioni che hanno (effettuato il commit) terminato (commit o abort).  
 quindi la schedule precedente diventerebbe:  $R_1(x) W_1(x) \oplus_1 R_2(x) W_2(x) \oplus_2 \dots$   $x$  è ad uno stato costante del DB.

Def: Schedule restitutiva (strict) (SOTTO INSIEME  $\{I\}$ )  
 Una storia  $S$  è strict se nessuna  $T \in S$  pre modifica un  $obj_x$  finché tutte le TCS che hanno modificato lo stesso oggetto non hanno (committato) terminato.  
 OLTRE AD EXITARE UN ABORT IN CASCATA COME LE PROCEDIMENTI, PROMOTORI IL RIPARATIVO TRAMITE LA BIM; COSA CHE INVECE POTREBBE non essere possibile per le sole  $S$  NON strict esempio, se ignoriamo queste condizioni, potremmo avere: CASCODING ROLL BACK, VEDI ESEMPIO.

$S: W_1(x, 5) W_2(x, 8) \oplus_1$

considerando un altro valore di  $x=9$ , quello quale dovrebbe essere  $x=8$ ; poiché per  $T_1$  abbia eseguito, se la riparazione tramite BIM si particolare ad uno stato precedente la modifica di  $T_1$  in maniera secondaria (Before Image) cioè ad uno stato di cui  $x=9$ , perdendo la modifica fatta da  $T_2$ .

se avesse considerato la condizione di strict la schedule sarebbe:

$W_1(x, 5) \oplus_1 W_2(x, 8)$

La condizione strict comporta il completo isolamento delle transazioni in scrittura.

## Operazioni in Conflitto

Due operazioni sono in conflitto se svolgono, contemporaneamente, le seguenti proprietà:

- ① appartengono a due transazioni diverse
- ② lavorano sullo stesso item
- ③ almeno una di queste opaz. è di scrittura

Esempio:  $R_1(x) W_2(x)$ ,  $R_2(x) W_1(x)$ ,  $W_1(x) W_2(x)$ ,  $W_2(x) W_1(x)$

mentre non sono in conflitto ad esempio le seguenti:

$R_1(x) R_2(x) \sim$  poiché sono op. di lettura

$W_1(x) W_2(y) \sim$  poiché lavorano su item diversi

$R_1(x) W_1(x) \sim$  poiché si tratta di op. della stessa transaz.

Def: Schedule Completa S (non è diverso da quello degli esempi visti finora)

④ essa raccolge un insieme di transazioni fino al loro completamento.

⑤ vediamo le seguenti proprietà:

A - tutte le op in S sono tutte e solo le op. che compiono in  $T_i \cdot Y_i$ , con lo stesso ordine di opposizione cioè se  $T_i$  fa  $b_i(x)$  e poi  $N_i(x)$  deve fare lo stesso anche nella storia, pur con commit o un abort come ultima op.

B - ogni coppia di operazioni di scrittura in S è ordinata come in  $T_i \cdot Y_i$

C - per ogni coppia di operaz. il conflitto, vale accade prima dell'altra

⑥ le schedule quindi definisce un ordinamento parziale:

→ op. in conflitto → in ordine obbligatoriamente

→ op. non in conflitto → nessun ordine obbligatorio

Def: Equivalenza per conflitti

due schedule S ed S' sono equivalenti per conflitti se l'ordine di qualsiasi coppia di op. in conflitto è lo stesso nelle due schedule.

cioè se esistono opaz. in conflitto tra  $T_i$  e  $T_j$  o l'operazione su i precede quella su j:  $O_{Pi} < O_{Pj}$  in S, S' oppure le regole:  $O_{Pi} > O_{Pj}$  in S, S'

Serializzabilità

Un'esecuzione risale è certamente consistente ma non prevede concorrenza; tuttavia ci basta dimostrare che una storia S (che contiene transazioni di concorrenza su più obj) sia equivalente ad una delle sue possibili esecuzioni seriali per affermare che S è consistente

→ S è serializzabile per conflitti se è equivalente per conflitti ad una qualsiasi schedula serial S'; ricordiamo che in N transazioni vi sono  $(N!)$  possibili schedule, quindi S sia serializzabile è sufficiente che esse sia equivalenti ad una qualsiasi di tali  $N!$  schedule

per definizione una schedula risale S' ha  $T_i < T_j \forall i, j$  oppure  $T_i > T_j \forall i, j$  cioè per ogni coppia di transazioni  $T_i, T_j$  si ha:  $O_{Pi} < O_{Pj}$  oppure  $O_{Pi} > O_{Pj}$

Per dimostrare se S è equivalente ad S' si può usare un grafico di precedenza

chiamato anche grafo di realizzabilità:

① NODI = transazioni

② ARCHI: Esiste tra  $T_i$  e  $T_j$  se esiste una coppia di op. in conflitto (o S dove  $Op_i < Op_j$ )

All' esempio per un orco  $T_i \rightarrow T_j$  esistono 3 possibilità:

$$W_i(x) < R_j(x)$$

$$R_i(x) < W_j(x)$$

$$W_i(x) < W_j(x)$$

nella storia  $\pi$  Jemo prima le op in i e poi in j

→ la configurazione necessaria e sufficiente per la storia  $S$  sia realizzabile è che il grafo non deve avere cicli; se siamo in una situazione del tipo



nella storia  $\pi$   $T_i$  dovrebbe precedere  $T_j$  e  $T_j$  dovrebbe precedere  $T_i$ , il che non è possibile → storia non realizzabile

### Esempio

$$T_1: R_1(x) \quad W_1(x) \quad R_1(y) \quad W_1(y)$$

$$T_2: R_2(z) \quad R_2(y) \quad W_2(y) \quad R_2(x) \quad W_2(x)$$

$$T_3: R_3(y) \quad R_3(z) \quad W_3(y) \quad W_3(z)$$

possibile scambiare delle  $N!$  →  $S$ : ricordata dalla sequenza temporale di operazioni mostrata di seguito:

$T_1$

$T_2$

$T_3$

$$R_2(z)$$

$$R_2(y)$$

$$W_2(y)$$

$$R_1(x)$$

$$W_1(x)$$

$$R_1(y)$$

$$W_1(y)$$

$$R_2(x)$$

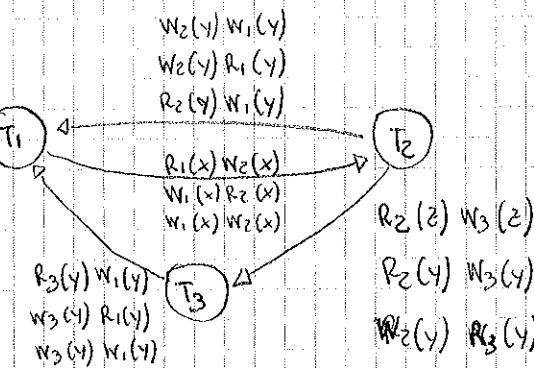
$$W_2(x)$$



$$\Rightarrow S: R_2(z) \quad R_2(y) \quad W_2(y) \quad R_3(y) \quad R_3(z) \quad R_1(x) \\ W_1(x) \quad W_3(y) \quad W_3(z) \quad R_2(x) \quad R_1(y) \quad W_1(y) \\ W_2(x)$$

→ per ogni operazione in  $S$  si cercheranno tutte le op. in conflitto con essa e le si riporteranno sul grafo di realizzabilità

## grafo di scelte eseguibili



~ coppia or conflitto per le proprietà ordine

la schedula S non è realizzabile!

NB nessuna delle  $N!$  possibili

scadenze: in caso di scadenza molto lunghe le si può  
ridurre per obj

$$Z: R_2(z) R_3(z) W_3(z)$$

$$X: \dots$$

$$Y: \dots$$

$$W_1(x)$$

$$S: R_1(x) R_2(x) \cancel{R_1(y)} W_2(x) W_1(y)$$

## Esempio

$$R_1(x)$$

$$X = X - N$$

$$R_2(x)$$

$$X = X + M$$

$$W_1(x)$$

$$R_1(y)$$

$$Y = Y + N$$

$$W_1(y)$$

$$W_2(x)$$

queste op. non dicono nes s'è considerazione della scadenza le sole  
è costituita solo da che sono le op. di W, R

## grafo di scelte eseguibili



la scadenza non è realizzabile; se ordiniamo a guardare l'esecuzione, la  
scrittura  $W_1(x)$  viene ricoperta da  $W_2(x)$   $\rightarrow$  l'esempio si conclude con  
 $X = X + M - N$

$X = X + M$  mentre dovrebbe concludersi con

## Metodi pratici per gestire la concorrenza

- [1] Metodi basati su locking (tipo 2PL)
- [2] Metodi basati su Time-Stamp Ordering (TSO)
- [3] Metodi basati su locking ottimistico

↳ se DB molto grande aveva problemi di concorrenza e molto raro quindi la si controlla a posteriori, abortendo quelle transazioni che hanno dato problemi.

I lock possono essere di due tipi

- ① BINARI ESCLUSIVI ~ limitano fortemente la concorrenza
- ② DI TIPO MULTIPLO ~ permettono accessi condivisi  $\text{W} \& R$  e per questo chiamati shared, S o R nel seguito.

③ concetto di base dell'utilizzo dei lock:

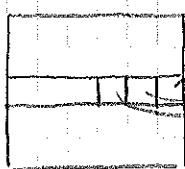
$T(x)$ :  
S lock x

shared  
-----  
Unlock x

④ Una tipica impl. di questo semplice protocollo avviene tramite una hash table, chiamata LOCK TABLE, in mem. c.le la quale mantiene una entry per ogni lock attivo. Se l'obj per cui si richiede un lock già esiste (conflict) sulla hash table) possono presentarsi due casi:

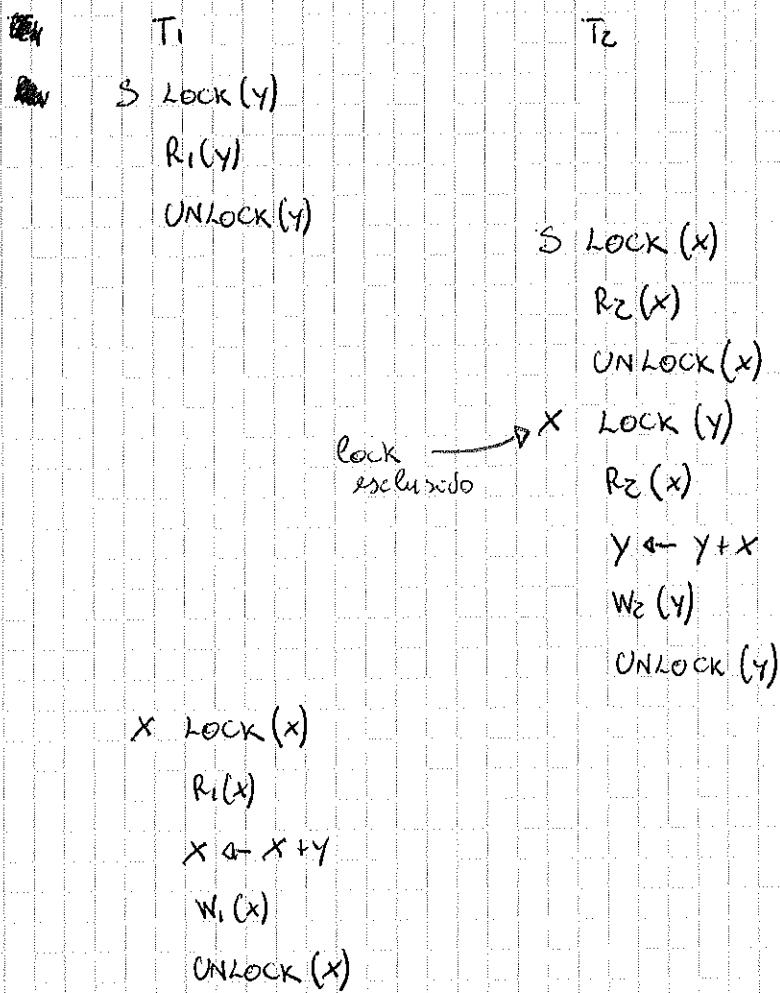
(i) si sta richiedendo un lock incompatible con quello attualmente attivo per l'obj  $\rightarrow T$  viene messa in attesa.

(ii) si sta richiedendo un lock compatibile con quello attualmente attivo (un lock shared),  $\rightarrow T$  viene aggiunta ad una lista di transazioni che hanno il lock shared su quell'obj.

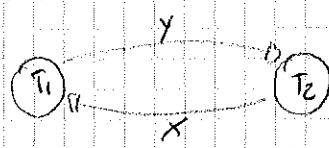


- ↳ tipo di lock
- ↳ Costa di attesa
- ↳ Costa di  $T_i$  che ha il lock shared

→ questo semplice meccanismo da solo non è sufficiente per gestire la concorrenza, ad esempio:

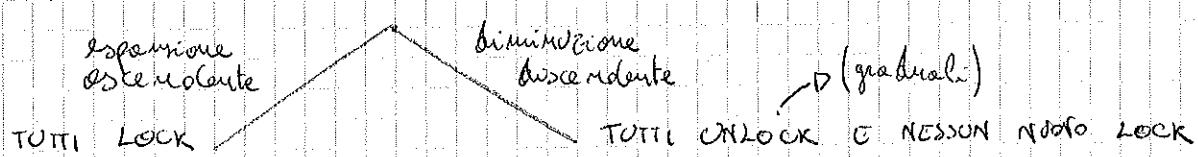


$$S = R_1(y) \quad R_2(x) \quad R_2(x) \quad W_2(y) \quad R_1(x) \quad W_1(x)$$



le schedule non sono serializzabili

Allora il protocollo che usa lock di questo tipo ma che garantisce la serializzabilità delle schedule è il protocollo 2-Phase Locking (ZPL): esso è costituito da due fasi, la prima in cui si richiedono tutte le op di lock e la seconda in cui tutti questi vengono rilasciati con il circolo di non poterne chiedere di altri.

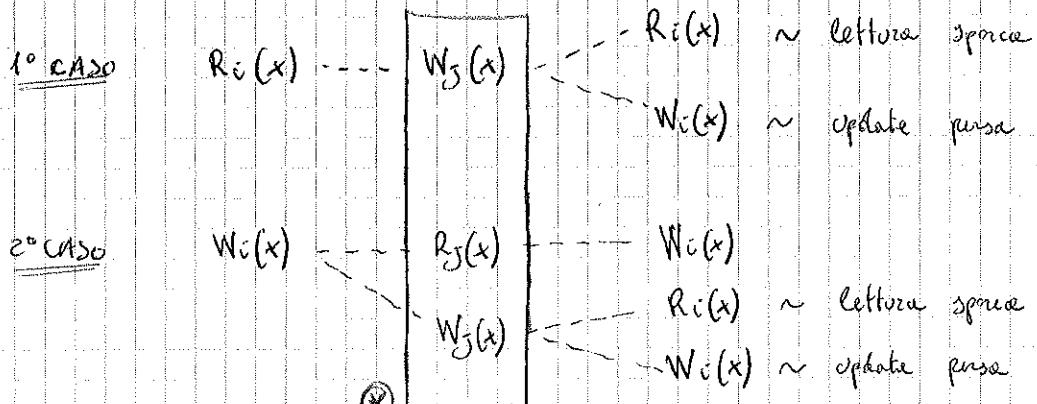


questo protocollo garantisce la probabilità di schedule serializzabili (non struct) ma non tutte quelle possibili (è estremamente restrittivo),

vedi oltre

Dimostrazione intuitiva del fatto che ZPL produce solo schedules serializzabili:

Sappiamo che affinché una storia  $S$  sia serializzabile debba esistere dei codi nel grafico di serializzabilità del tipo:



se NON serializzabile debbo esistere e quindi delle sequenze del

-  $R_i(x) \sim$  lettura spracca

-  $W_i(x) \sim$  update pusa

-  $W_i(x) \sim$  lettura spracca

-  $R_i(x) \sim$  update pusa

- \*) E come se  $T_3$  si inserisse nella  $T_1$ ; nel protocollo di base sappiamo che questo è possibile (vedi esempio precedente) ma nel ZPL no sarà:

1° caso  $\Rightarrow$   $T_1$  chiede un lock shared il quale verrà montato fino a che  $T_3$  non eseguirà l'ultima operazione su  $X$ , quindi  $T_3$ , chiedendo un lock esclusivo per scrivere  $X$ , verrà messo in attesa.

2° caso  $\Rightarrow$   $T_1$  chiede un lock esclusivo e quello chiesto da  $T_3$  (shared o esclusivo) viene sospeso ( $T_3$  è messo in attesa).

Controesempio (ZPL non produce TUTTE le possibili schedules serializzabili)

$R_2(x) W_3(x) C_3 W_1(y) C_1 R_2(y) W_C(z) C_C$

questa schedule non è produttibile da ZPL

perché il lock shared su  $x$  chiesto da  $T_2$

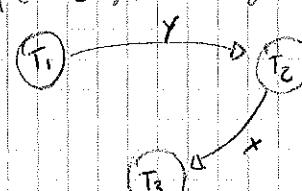
non deve rilasciato fino al momento in cui non accediamo per l'ultima

volta con  $T_2$  ad un dato, cioè al punto (\*)

Non prodotte da ZPL

questo significa che  $W_3(x)$  dovrebbe comporre lì, nonostante questo la storia è serializzabile.

Acquisendo l'ultimo lock, la  $T_2$  finisce la sua esecuzione ed inizia quella degli unlock (rilascio graduale)



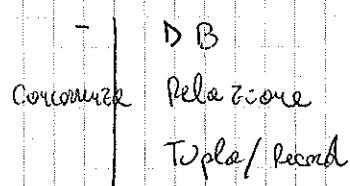
→ il punto è che ZPL lavora non su storie complete ma max mano che la  $T_1$  esegue e quindi non hanno il grado di vedere il futuro.

## Tipi di ZPL

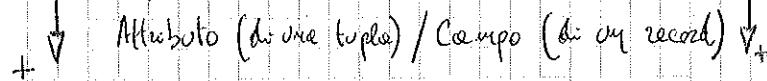
- BASIC: quello appena descritto. Gestisce la concorrenza ma non produce schedules strict
- CONSERVATIVE: tutti i lock sono acquisiti automaticamente prima dell'inizio (in modo atomico)
  - VANTAGGIO: Non si possono avere deadlocks, in quanto non ci possono essere attese circolari in una risorsa  $\rightarrow$  o dunque tutti i lock sono messi.
  - Svantaggio: il livello di granularità richiesto è piuttosto alto (da tabella su su, VEDI OLTRE per def di granularità), non produce schedules strict.
- STRICT: la fase di ch-lock è eseguita automaticamente al termine della transazione. Gestisce la concorrenza e tutti i lock hanno le proprietà delle schedules strict:
  - Ora Ti vengono chiamati NON CASCADING, RECOGNIZABLE, recuperabile BLM contemporaneamente al termine perché tutte le gradi di quello che la transazione ha visto fino alla fine delle stesse
- STRICT + CONSERVATIVE: produce schedules strict e nessun deadlock

## Oggetti e granularità

All'interno del DB possono essere presenti diverse granularità di "oggetti" sui quali si può richiedere un lock:



overhead (lock che si debba mantenere)



$\Rightarrow$  il livello di granularità scelto dipende dal carico di Transazioni che devono essere eseguite e il livello di tale granularità può essere cambiato effettuando delle scadenze di blocco in funzione delle op. da eseguire.

Problema: Una transazione a livello di "Relazione" ad esempio non può modificare dei dati che già con un'altra transazione a livello più

basso lo sta già facendo; per risolvere questo problema si considerano altri tipi di lock oltre S e X:

- ① IS n' intenzione di leggere a livello più basso del corrente
  - ② IX n' " " scrivere " " " " "
  - ③ SIX n' lettura di tutto il livello corrente, intenzione di scrivere a livello più basso.
- "intenzione"  $\Rightarrow$  il lock effettivo di quel tipo verrà messo a livello più basso.

Si costruisce ora tavola di incompatibilità dei lock

	S	X	IS	IX	SIX
S	V	V			
X					
IS	V	V	V*	V*	
IX		V*	V*		
SIX	O	V*			

\* eventuali incompatibilità verranno risolte al livello più basso

$\Rightarrow$  No perché la scrittura avviene più in basso.

## Deadlock

Cos'è un deadlock:

- Mutua esclusione
  - Hold & Wait: una transaz. ha ottenuto una risorsa e ne attende altre
- ② Attesa circolare: una transaz. attende da un'altra una risorsa; se il ciclo si chiude abbiamo un deadlock



attesa circolare

Esistono diverse tecniche per gestire i deadlock, esse sono suddividibili in:

- ② Tecniche che rimuovono => se esiste un deadlock si chiude una transazione e la chiude in modo riportivo "sperando" che il ciclo non si ripresenti nuovamente

queste tecniche montano un grafo delle attese:   
in cui i nodi rapp. le transazioni e gli archi, quando entrano, il fatto che una transaz. Ti sta aspettando una risorsa della transazione Tj (se il verso dell'arco va da Ti a Tj). Se il grafo punta 1 o più nodi, sappiamo che esistono 1 o più deadlock; per risolverli si fa che abort di Ti e ciclo: fisicamente Ti è la transazione che ha fatto meno lavoro o quello più giovane (che si suppone abbia fatto poco lavoro ma non è detto). Per evitare ridurre la probabilità che facendo riportare Ti si ripresenti subito un ciclo, si fa attendere la transazione abortita per un certo periodo anziché farla partire subito.

- ② Tecniche che prevenono => richiedono una gradi monitorate molto alta (selezione in re)

metodi:

- ② Z PL conservative
- ② ordinare gli obj in cui permettiamo i lock in difficile da implementare in gerarchie fiscate

- ② Tecniche che evitano => non siamo in grado di prevedere a priori che il deadlock non si può evitare ma possiamo monitorare le transaz. in modo che se arriviamo in uno stato "sospetto", del quale ci può apparire verificarsi un deadlock, allora si applicano dei metodi per evitarlo:

- 1- NO WAITING: se la transazione non ottiene subito il lock fa un abort.
  - 2- CAUTIONS WAITING: (Attese cieca) se  $T_i$  è in attesa su un lock ottenuto da  $T_j$  allora fa un abort solo se  $T_j$  è in attesa.
  - 3- WAIT & DIE: (basata sul Transaction Timestamp ~~timestamp~~)
- ```

    IF TS(Ti) < TS(Tj) → Ti è più vecchia di Tj
    Ti WAITS → Time Stamp di Tj
    ELSE PREEMPT(Ti)
        ↳ abort
    ↳ la transazione più vecchia attende quella più giovane: Ti waits IF TS(Ti) < TS(Tj)
    altrimenti, se c'è la più giovane e deve attendere TS(Ti) > TS(Tj), allora
    Ti fa un abort
  
```
- [N.B.]**: situazione in cui  $T_i$  chiede un lock( $x$ ) e l'obj  $x$  è bloccato da  $T_j$

#### Considerazioni

- ①  $T_i$  può solo aspettare su transazioni più giovani e non può vecchie, quindi non si può verificare un cicle di attesa (**deadlock**)
 

```

        Ti → Tj → ...
        X
        ↳ Non è possibile che una transazione più giovane ( $T_j$ ), dovrebbe
        aspettar su una più vecchia ( $T_i$ )
      
```
- ② Nel momento in cui  $T_i$  fa un abort, riporta con lo stesso timestamp per consentire l'aging delle transazioni: se una transazione giovane viene abortita più volte (via else), ad ogni riportanza sarà sempre più vecchia (quindi altre transazioni saranno partite con un TS più elevato) fino a diventare abbastanza per attendere un lock concesso da due  $T_j$  giovani (via if), impedendo del lock e provocare l'uccisione di tutte le transazioni più giovani che vogliono impadronirsi di esse
 

```

        Ti → ...
        X
        ↳ Una transazione  $T_i$  può essere abortita molte volte, causando situazione di
        starvation se le  $T_j$  che hanno i lock sono molto lunghe (ma comunque
        finite)
      
```

4 - WOUND & Wait: (Ferrisi & Aspelta) ~ basata anch'essa sull'utilizzo dei TS;

[NB]: la situazione è ancora quella in cui  $T_i$  chiede un lock ( $x$ ) e l'obj  $x$  è bloccato da  $T_j$

{ IF  $TS(T_i) < TS(T_j)$

PREEMPT ( $T_j$ )

ELSE  $T_i$  WAITS

↳ La transazione più vecchia manda l'occisione di quella più giovane che ha il lock attualmente: PREEMPT ( $T_j$ ) IF  $TS(T_i) < TS(T_j)$   
altrimenti la più giovane aspetta: ELSE  $T_i$  WAITS

### Considerazioni

- ① Non si può ~~una~~ definire un ciclo di attesa e coesce dell'avviamento imposto dalle stesse (come prima)
- ② Una  $T_j$  che ripete dopo un abort ha lo stesso TS (come prima)
- ③ Una transazione può essere abortita solo quando ha un lock ed una transaz. più vecchia ne richiede uno sullo stesso obj; quando  $T_j$  ripete dopo il 1° abort solo una transazione più vecchia può avere il lock quando  $T_j$  dovrà eseguire un wait  $\rightarrow$  non può essere starvation ??

→ Esiste un altro metodo che utilizza i TS (ma in modo diverso da quanto visto con i metodi 3 e 4) e rientra nella categoria dei metodi che preengono al deadlock; esso non è molto usato nella pratica, gli si preferisce il CPL consentibile con si tratta del protocollo Time stamp Ordering (TSO)

idee generale: vediamo i TS per ordinare l'esecuzione delle transazioni secondo una specifica schedula seriale equivalente (quella ~~che~~ con lo stesso ordine dei TS), se non è possibile costituire una tale schedula si uccide una transaz. e la si fa ripartire con un timestamp più alto

Schedula iniziale:  $T_3 T_1 T_4 T_2 \Rightarrow$  vogliamo ottenere  $T_1 T_2 T_3 T_4$  cioè la schedula seriale equivalente; se questa non può essere ottenuta si pro: occidere  $T_1$  e farla ripartire con un TS più alto  $\rightarrow T_3 T_4 T_2 T_1$ , occidere  $T_2$  e farla ripartire con un TS più alto  $\rightarrow T_3 T_4 T_5 T_6$  ~~se si dovrà evitare deadlock~~

TSO Basic

definiamo Oggetto  $x$   $\xrightarrow{\text{RTS}(x)}$   $\xrightarrow{\text{WTS}(x)}$  } timestamps massimi per le op di R e W  
in un obj  $x$

T: WRITE( $x$ )

{ IF RTS( $x$ ) > TS( $T$ ) || WTS( $x$ ) > TS( $T$ )  
abort( $T$ )

ELSE Write( $x$ ), WTS( $x$ ) = TS( $T$ )

$\Rightarrow$  se una transazione più giovane di T ha letto ( $RTS(x) > TS(T)$ ) oppure  
scritto ( $WTS(x) > TS(T)$ ) il dato  $x$  che T vuole modificare, T viene  
abortita perché sta cercando di scrivere un dato che nel futuro ~~non c'è~~ è  
già stato letto o scritto. Altrimenti se nessuno nel futuro ha letto R o W  
esempio, schedule ...R<sub>3</sub>( $x$ ) W<sub>1</sub>( $x$ ) ... di  $x$ , T scrive  $x$  e aggiorna WTS( $x$ )

dove R<sub>3</sub>( $x$ ) ha impostato l' $RTS(x) = 3$  (vedi oltre, per op. di lettura delle Tc)

T<sub>1</sub> deve essere abortita perché esiste una transazione più giovane, T<sub>3</sub>, che ha  
già letto il dato che vogliamo modificare: T<sub>1</sub> è nel passato e non può  
eseguire modifiche su dati già utilizzati (R o W) in futuro  $\rightarrow$  T<sub>1</sub> viene  
fatta ripartire con un TS più alto, come cioè proiettato nel futuro:

...R<sub>3</sub>( $x$ ) W<sub>4</sub>( $x$ ) ...

T: READ( $x$ )

{ IF WTS( $x$ ) > TS( $T$ )

abort( $T$ )

ELSE read( $x$ ), RTS( $x$ ) = MAX (RTS( $x$ ), TS( $T$ ))

$\Rightarrow$  se una transazione più giovane di T ha scritto (nel futuro quindi) un  
dato che T vuole ora, cioè nel passato, leggere, T viene abortita perché  
la sua lettura scrivebbe inconsistente.

Altrimenti T legge il dato (se nessuno nel futuro lo ha modificato)

e, poiché le op. di lettura possono essere eseguite senza problemi nel passato  
e nel futuro, RTS( $x$ ) viene ~~sempre~~ tenuto aggiornato sempre con la

transazione più giovane che ha letto  $x$  (scrivere per l'esecuzione di WRITE)

## Svantaggi rispetto al ZPL

- (1) Si devono mantenere  $RTS(x)$  e  $WTS(x)$  per tutti gli "obj" del DB, inoltre anche le operazioni di lettura comportano una scrittura, quella dell' $RTS(x)$  oppure no.
- (2) molti abort ~ nel ZPL le transaz. singole messe in attesa e non abortite

## Vantaggi (in generale)

- (1) le schedule protette da TSO sono serializzabili anche se non le produce tutte, ovvero ci possono essere delle schedule serializzabili non protette da TSO:  

S:  $W_3(x) W_2(x) W_1(x)$  è serializzabile ma non TSO poiché la schedula che ~~converge~~ convergono avrebbe  $T_1, T_2, T_3$  mentre TSO abortirebbe le transazioni 2 e 1 e le farebbe riaprire come 4 e 5 producendo quindi la schedula  $T_3, T_4, T_5$

## Esempio di schedule TSO ma non ZPL

$R_2(x) W_3(x) C_3 \quad W_1(y) C_1 \quad R_2(y) W_2(z) C_2$

|   |         | 0 | 2 | 2 | 2 | 2 |  |
|---|---------|---|---|---|---|---|--|
| X | RTS = 0 | 0 | 2 | 2 | 2 | 2 |  |
|   | WTS = 0 | 0 | 3 | 3 | 3 | 3 |  |
| Y | RTS = 0 | 0 | 0 | 0 | 2 | 2 |  |
|   | WTS = 0 | 0 | 0 | 0 | 1 | 1 |  |
| Z | RTS = 0 | 0 | 0 | 0 | 0 | 0 |  |
|   | WTS = 0 | 0 | 0 | 0 | 0 | 2 |  |

poiché non si definiscono abort la schedula è TSO ma non ZPL poiché il lock shared su  $R_2(x)$  viene rilasciato solo dopo un eseguito  $W_2(z)$  (ultima op. della  $T_2$ ) quindi  $W_3(x)$  dovrebbe comparire in quel punto.

## Esempio di schedule ZPL ma non TSO

$R_2(x) W_2(y) C_2 \quad R_1(y) W_1(x) C_1$

|   |         | 0 | 2 | 2 |  |
|---|---------|---|---|---|--|
| X | RTS = 0 | 0 | 0 | 0 |  |
|   | WTS = 0 | 0 | 0 | 0 |  |
| Y | RTS = 0 | 0 | 0 | 0 |  |

abort ~ xclli una transaz. nel futuro ( $T_2$ ) ha scelto il dato che la transazione  $T_1$  (nul posato) crede leggere.  $T_2$  viene letta esistente come  $T_3$

## Thorios Write Rule

E' una regola che permette di abortire meno transazioni possibile

T : WRITE(x)

IF  $RTS(x) > TS(T)$     II     $WTS(x) > TS(T)$

↓  
abort  
↓  
Aborts nel futuro già scritto

↓  
Aborts nel futuro già letto

Se una transazione nel futuro ha fatto una scrittura cieca di X, allora anziché abortire T, semplicemente non facciamo nella pila sapendo già che la modifica fatta adesso avrebbe cresciuto; la regola quindi diventa

IF  $RTS(x) > TS(T)$

abort

caso non ancora IF  $WTS(x) > TS(T)$  non fare niente  
ELSE write(x),  $WTS(x) = TS(T)$

→ si finisce nelle vie else \* punti  $TS(T) > RTS(x)$  indipendentemente dal fatto che sia  $WTS(x) \geq TS(T)$ , nel caso poi in cui sia  $WTS(x) > TS(T)$  allora non si fa niente, altrimenti si effettua la scrittura

## TSO STRICT

se per le op. di READ di un WRITE si richiede che:

IF  $WTS(x) < TS(T) \rightarrow$  qualcuno nel passato ha scritto il dato che vogliamo

THEN \* leggere o modificare

~~Espresso del 20/03/2007~~

\* aspetta che T termini, dove T' è tale che  $TS(T') = WTS(x)$ ; cioè bisogna aspettare che la transazione che un passato ha scritto x, termini.

① No deadlock perché l'attesa e sempre sulla Ti più vecchia

② produce schedules strict. È immagine dei rollback e cascata

[FINE CORSO : ~ 22/03/07]