Operatori relazionali

Tecnologie e Sistemi per la Gestione di Basi di Dati e Big Data M

Elaborazione di query mediante operatori

- Nei DBMS relazionali ogni interrogazione, comunque complessa, non viene elaborata come un blocco monolitico, bensì viene sempre risolta combinando in modo opportuno le azioni di un numero limitato di operatori

- Per risolvere al meglio un'interrogazione è quindi importante:
  1) Avere a disposizione un'implementazione efficiente di tali operatori
  2) Trovare il modo migliore di combinare tali operatori per produrre il risultato corretto

- Trattiamo qui il primo punto (il punto 2 verrà affrontato successivamente)
- Premessa importante è che, per ogni operatore, esistono diverse alternative per la sua realizzazione, e raramente esiste un algoritmo che è sempre migliore degli altri
  - L'efficienza dipende da diversi fattori, quali numero di tuple, pagine, buffer, presenza di indici, ecc.

Operatori logici vs. fisici

La terminologia comunemente adottata distingue tra:

- Operatori logici (es. join)
  - Sono un'estensione di quelli dell'algebra relazionale; svolgono una determinata funzione e producono un insieme di tuple con certe proprietà
- Operatori fisici (es. join nested-loops)
  - Sono implementazioni specifiche di un operatore logico; in funzione di vari fattori, è possibile associare ad ogni operatore fisico un costo di esecuzione

Vie di accesso (access path)

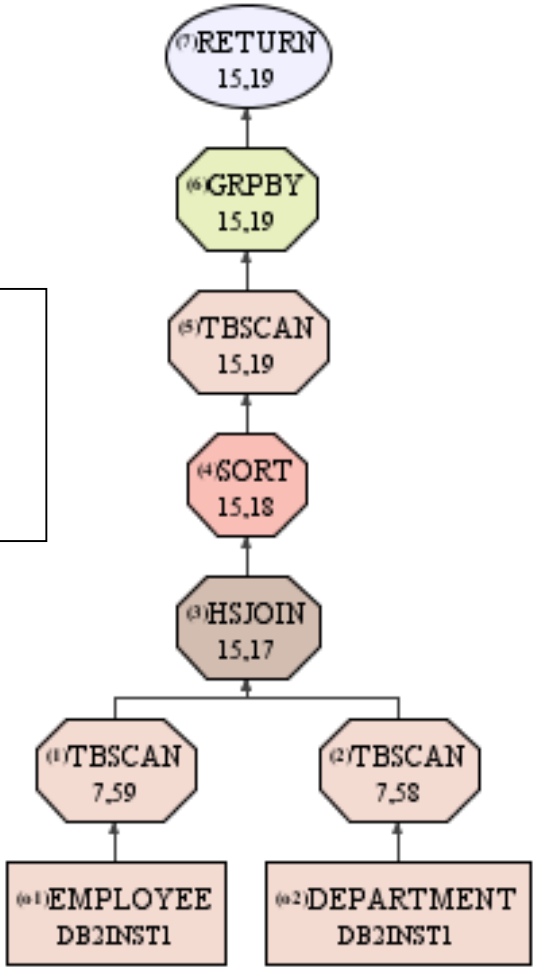
- I modi alternativi per recuperare i record da una relazione si dicono vie di accesso (anche: metodi, cammini)

- In generale, le vie di accesso possibili sono:
  - Scansione sequenziale
  - Accesso a uno o più indici con predicati di selezione

Piani di accesso (access plan)

- I modi alternativi per risolvere un'interrogazione si dicono piani di accesso
  - Simili agli alberi dell'algebra relazionale
- Ogni piano di accesso fa uso di:
  - metodi di accesso ai dati
  - operatori fisici

```
SELECT   E.Job,D.DeptName, COUNT(*)
FROM     Department D, Employee E
WHERE    E.WorkDept = D.DeptNo
GROUP BY E.Job, D.DeptName
```



IBM DB2

I numeri accanto ad ogni operatore sono i "timeron", unità di costo di DB2 che tiene conto sia dei costi di I/O che di quelli di CPU

Valutazione dei costi: una precisazione

- Per confrontare tra loro diverse alternative (operatori fisici, piani di accesso, ecc.) spesso è sufficiente considerare il numero di I/O eseguiti
  - Questo serve anche ad avere un'idea dell'ordine di grandezza dei tempi di esecuzione (secondi, minuti, ore, …)
- In alcuni casi il solo numero di I/O può non essere sufficiente a giustificare il perché di certe scelte; in questo caso conviene ragionare sul tipo di letture/scritture (random vs sequenziali)

- Nei DBMS la valutazione dei costi (stimati) di esecuzione considera anche i costi di CPU, al fine di meglio modellare il consumo di risorse

IBM DB2

The SQL optimizer does not estimate elapsed time but rather resource consumption. The optimizer does not model all factors that can affect elapsed time; it ignores those that do not affect the efficiency of the access plan. The elapsed time is affected by a number of run-time factors including: the system workload; the amount of resource contention; the amount of parallel processing and I/O; the cost of returning rows to the user; and the communication time between the client and server.

`IBM DB2 Administration Guide: Performance`

Operatori logici

- Ordinamento
- Selezione
- Proiezione
- Join
- Operatori insiemistici (unione, differenza)
- Group by
- Operatori aggregati (avg, sum, count)
- Operatori di modifica (update, delete, insert)

- In tutti i casi, analizzando i costi di esecuzione ignoreremo il costo di produzione del risultato (perché?)

Parametri di costo

- Al fine di poter ragionare sulla bontà delle diverse alternative consideriamo una serie di parametri fisici, tipicamente disponibili nei cataloghi del DBMS:
  - N(R) = numero di record di R
  - P(R) = numero di pagine di R
  - Len(R) = lunghezza (in byte) di un record di R
  - NK(R.A) = numero di valori distinti dell'attributo R.A
  - TP(R) = numero di tuple per pagina
    - Vale la relazione $P(R)=\lceil N/TP \rceil$
  - B = numero di pagine buffer a disposizione per l'operatore
  - L(IX) = numero di pagine foglia dell'indice IX
    - Si omettono R e IX se chiari dal contesto

Ordinamento (sort)

- Non è un operatore dell'algebra (perché?), ma è un'operazione molto importante:
  - Clausola ORDER BY
  - Utilizzo nel bulk-load di un indice
  - Eliminazione di record duplicati (clausola DISTINCT)
  - Usabile anche per eseguire join e GROUP BY
  - …

Ordinamento: varianti

- Nel caso base che consideriamo abbiamo in input un insieme di record e produciamo in output lo stesso insieme di record, ordinato secondo una dato criterio
  - Quindi le dimensioni (in byte) dell'input e dell'output coincidono

- E' evidente che vi possono essere diverse varianti, degne di nota, ad es.:
  - Se richiesto, si possono eliminare i duplicati durante l'esecuzione del sort
  - Se alcuni attributi in input non servono nell'output, si possono eliminare durante l'esecuzione del sort

```
SELECT DISTINCT LastName
FROM   Employee
ORDER BY LastName
```

Ordinamento: esempio

attributi nel risultato

attributi di ordinamento

Matricola	CodiceFiscale	Cognome	Nome	DataNascita
216635467	RSSNNA98A53A944V	Rossi	Anna	13/01/1998
160239654	VRDMRC99H21F839X	Verdoni	Marco	21/06/1999
214842132	VRDCRL99H20G125J	Verdi	Carlo	20/06/1999
200643121	RSSDRA98M10A944V	Rossi	Dario	10/08/1998

```
SELECT Cognome,Nome,Matricola
FROM   Studenti
ORDER BY Cognome,Nome
```

Matricola	Cognome	Nome
216635467	Rossi	Anna
200643121	Rossi	Dario
214842132	Verdi	Carlo
160239654	Verdoni	Marco

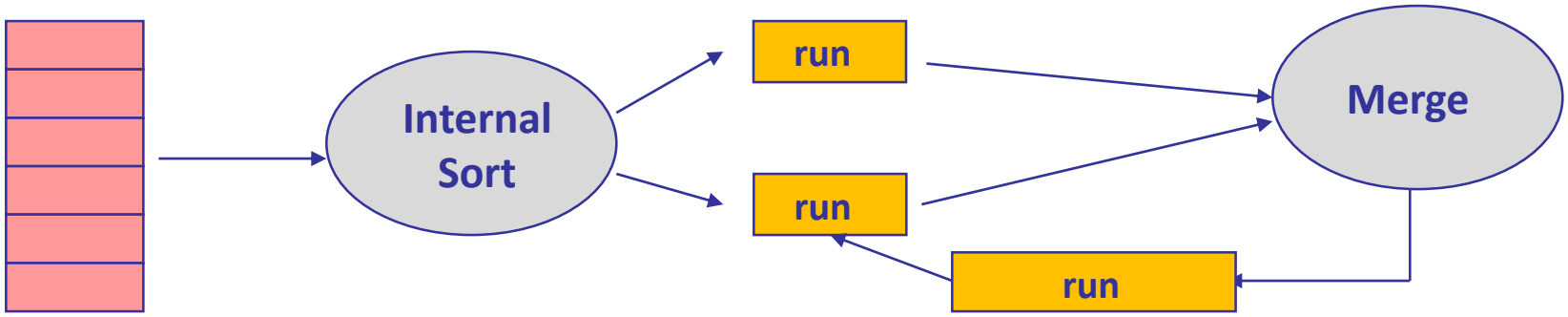
Algoritmi di sort "interni" (RAM)

- Bubble sort
- Insertion sort
- Shell sort
- Merge sort
- Heapsort
- Quicksort
- …

- Hanno prestazioni generalmente molto buone
  - Tipicamente O(N logN) o O(N^2)
- In genere richiedono che il dataset sia interamente contenuto in memoria
  - Perché non è opportuno caricare tutto il dataset in memoria virtuale?
- Il merge sort fa eccezione
  - È sufficiente che siano in memoria solo 2 elementi

Sort-merge "esterno"

- L'idea di base è che, siccome i dati non stanno in memoria, possiamo
  - Dividere i dati in sequenze (run) più piccole
  - Ordinare le sequenze una ad una
  - Fondere (merge) le sequenze un elemento per volta
- Di fatto, ogni sequenza ha inizialmente le dimensioni di una pagina (o meno), poi 2, 4,...
- Il primo passo di ordinamento utilizza un algoritmo di sort interno (es., quicksort)



Sort-merge: esempio

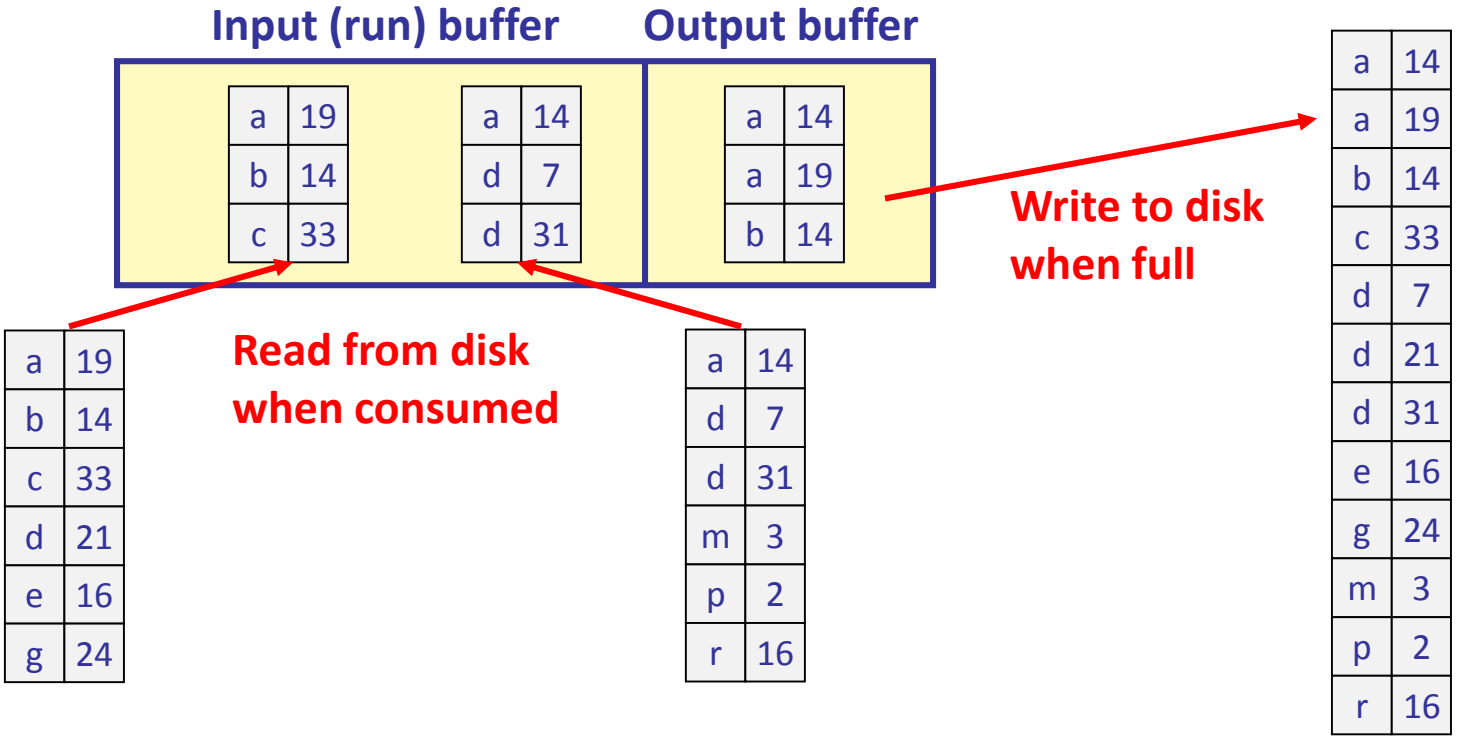
- Dati: 3, 4, 6, 2, 9, 4, 8, 7, 5, 6, 3, 1, 2
- 2 record per pagina
- Input: [3,4], [6,2], [9,4], [8,7], [5,6], [3,1], [2]

- Passo di Sort interno:
  [3,4], [2,6], [4,9], [7,8], [5,6], [1,3], [2]

- Passi di Merge:
  1. passo: [2,3,4,6], [4,7,8,9], [1,3,5,6], [2]
  2. passo: [2,3,4,4,6,7,8,9], [1,2,3,5,6]
  3. passo: [1,2,2,3,3,4,4,5,6,6,7,8,9]

Sort-merge: fusione e gestione dei buffer

- Sono usate solo B=3 pagine buffer:
  2 di input, 1 di output
- Si legge la prima pagina da ciascuna run e si può quindi determinare la prima pagina dell'output; quando tutti i record di una pagina di run sono stati consumati, si legge un'altra pagina della run



Sort-merge: costo

- Se il numero di pagine del file di input è $P = 2^k$
  - Il sort interno produce 2^k run di 1 pagina
  - Il primo passo di merge produce 2^{k-1} run di 2 pagine
  - Il secondo passo produce 2^{k-2} run di 4 pagine
  - Il k-esimo passo produce 1 run di 2^k pagine

- Numero di passi (sort+merge): $\lceil \log_2 P \rceil + 1$
- Costo totale: $P(\lceil \log_2 P \rceil + 1)$ letture, $P(\lceil \log_2 P \rceil + 1)$ scritture: $2P(\lceil \log_2 P \rceil + 1)$

NB1: Nell'ultimo passo di fusione si può evitare di scrivere su disco, fornendo direttamente il risultato in output

NB2: In questo caso, e nei seguenti, per semplicità assumiamo che ad ogni passo di fusione tutte le run vengano lette e scritte

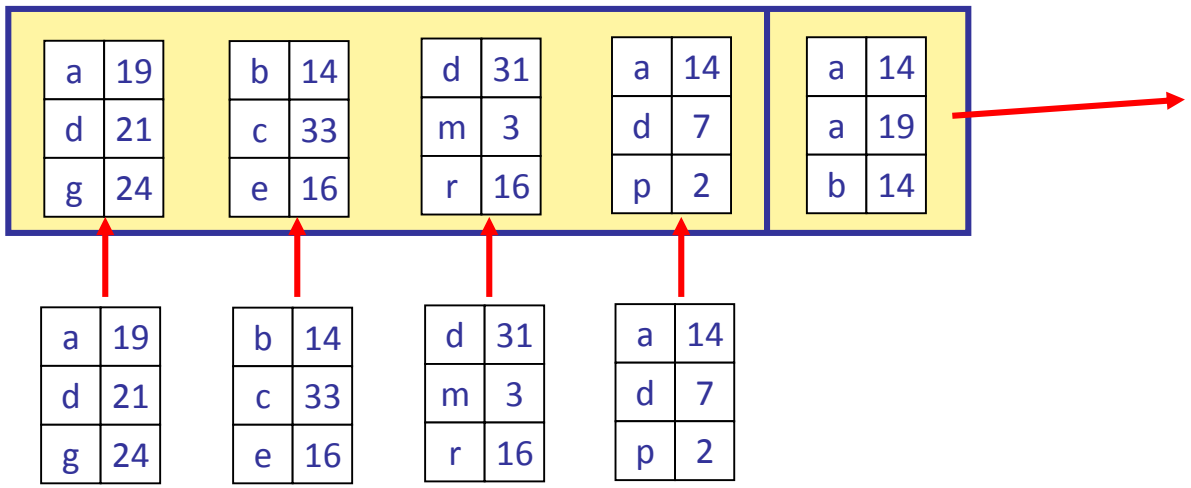
Sort-merge: esempio

- Nel caso base, con le formule date, si avrebbe, ad es.: P = 8192: 229376 I/O
- Se ogni I/O richiede 10 ms: circa 38 min!

- Un primo miglioramento si può ottenere se si hanno a disposizione B buffer e si ordinano inizialmente B pagine alla volta, anziché 1
- Il costo diventa $2P(\lceil \log_2 P/B \rceil + 1)$, in cui l'argomento del logaritmo rappresenta il numero iniziale di run
- Con B = 11: circa 30 minuti

Sort-merge a Z vie: idea

- Anche nel caso di più di 3 buffer a disposizione, notiamo che questi non verrebbero utilizzati dall'algoritmo descritto durante la fusione
- L'idea del sort-merge a Z vie è che, se si hanno a disposizione B=Z+1 pagine buffer, invece che fondere 2 pagine per volta se ne possono fondere Z
  - Una pagina serve sempre per l'output
- Aumentando il fan-in del passo di merge, si aumenta la base del logaritmo, e quindi si riducono i passi di fusione



Algoritmo sort-merge a Z vie

leggi il file B pagine alla volta, ordinando
    le pagine e scrivendole in un file ausiliario (run)

while (numero di run>1)

    while (ci sono ancora run da fondere)

        seleziona Z run dal passo precedente

        leggi le run in memoria una pagina per volta

        fondi le run, scrivendo sul buffer di output

        scrivi il buffer di output una pagina per volta

Sort-merge a Z vie: esempio

B=Z+1=11, P=8192

- Sort interno: produce $\lceil 8192/11 \rceil = 745$ run di 11 pagine, tranne l'ultima che è di 8
- Fusione (1): merge a 10 vie che produce $\lceil 745/10 \rceil = 75$ run di 110 pagine, tranne l'ultima di 52
- Fusione (2): merge a 10 vie che produce $\lceil 75/10 \rceil = 8$ run di 1100 pagine, l'ultima di 492
- Fusione (3): merge a 8 vie che produce le 8192 pagine ordinate

Sort-merge a Z vie: costo

- Nell'esempio il costo è dato dalla lettura e scrittura di 8192 pagine in 4 passi = 65536 I/O (circa 11 min)
- Costo: P letture e P scritture per ogni passo:

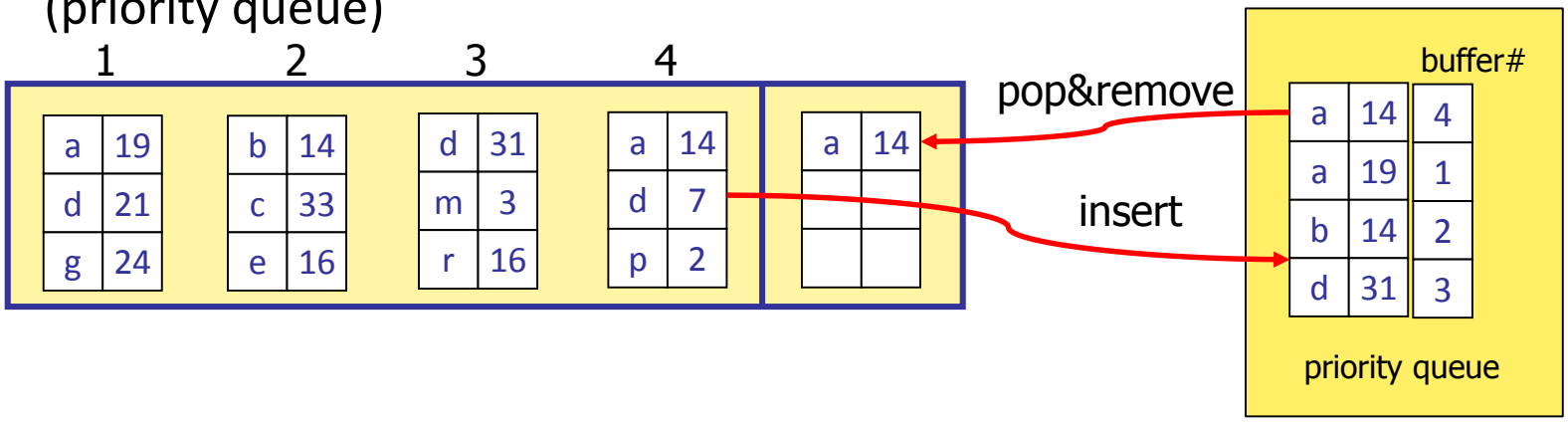
$$2P\left(\lceil \log_Z \lceil P/(Z+1) \rceil \rceil + 1\right) \approx 2P\lceil \log_Z P \rceil$$

P\Z	2	4	8	16	128	256
1K	10	5	4	3	2	2
10K	13	7	5	4	2	2
100K	17	9	6	5	3	3
1M	20	10	7	5	3	3
10M	23	12	8	6	4	3
100M	26	14	9	7	4	4
1G	30	15	10	8	5	4

Numero di passi
in funzione di P e Z

Sort-merge a Z vie: costi di CPU

- Se Z è grande, il merge a Z vie può essere costoso dal punto di vista della CPU
  - Ad es., se il numero di passi non diminuisce tra Z=128 e Z=256, conviene usare Z=128
- Per ridurre i costi di CPU, che essenzialmente dipendono dal dover determinare ogni volta il valore minimo tra Z elementi (con costo O(Z)), conviene inserire i valori minimi correnti di ogni run in una coda con priorità (priority queue)



- Se la coda viene implementata tramite un heap la scelta del valore minimo ha costo O(1), la sua rimozione dalla coda costo O(log Z) e l'inserimento ordinato in coda di un nuovo valore costo O(log Z)

Sort-merge a Z vie: tipi di letture

- Anche avendo molto spazio a disposizione in RAM (B grande), la soluzione migliore potrebbe non essere scegliere Z=B-1
  - Tutte le letture sono random!
- Distinguendo tra letture random e sequenziali si perviene a scelte più accurate e performanti, che spiegano anche perché nei DBMS valori troppo elevati di Z non vengono utilizzati
- Il modello che consideriamo assume che il costo di X letture sequenziali sia:

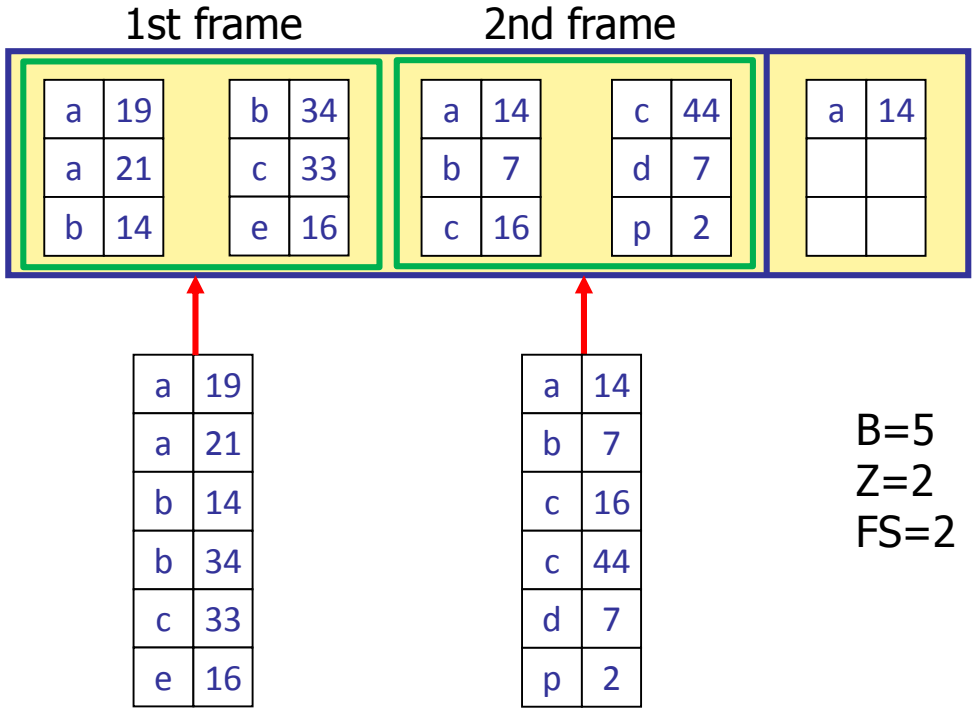
$$Ts + Tr + X \times Tt = \left(\frac{Ts + Tr}{Tt} + X\right) \times Tt = (C + X) \times Tt$$

in cui il parametro C ($C \approx 10 \div 50$) rappresenta l'overhead di una lettura random rispetto a una sequenziale:

- X letture sequenziali: (C + X) Tt
- X letture random: X (Ts + Tr + Tt) = X (C + 1) Tt

I/O random e sequenziali (1)

- Durante il sort interno si leggono e scrivono B pagine alla volta
- Il costo, usando Tt come unità di misura, si può stimare pari a circa 2*P/B (C + B), in cui 2*P/B rappresenta il numero di seek
  - La formula è approssimata se P non è multiplo di B
- Per la fusione, si organizzano i B-1 buffer dedicati alla lettura in Z "frame" di FS pagine ciascuno
  - Si leggeranno quindi FS pagine alla volta da ogni singola run



I/O random e sequenziali (2)

- Per ogni passo di fusione si hanno i seguenti costi (le write restano tutte random):
  - Letture: P/FS (C+FS)
  - Scritture: P (C+1)

- Complessivamente il costo, usando Tt come unità di misura, è

$$2\ P/B\ (C + B) + (P/FS\ (C+FS) + P(C+1)) \lceil \log_Z P/B \rceil$$

- Considerando solo la parte che varia con Z, e ricordando che FS = (B-1)/Z, si deve minimizzare: $(P*C*Z/(B-1) + P(C+2)) \lceil \log_Z P/B \rceil$

I/O random e sequenziali (3)

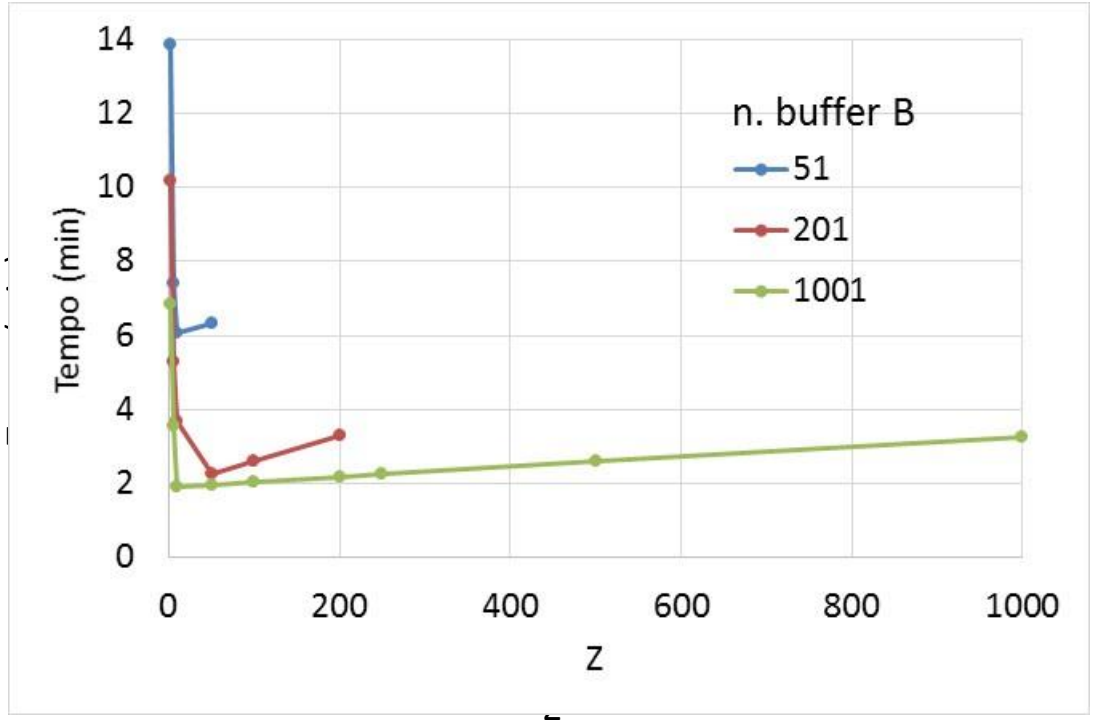
- Esempio: la tabella mostra i valori di $(P*C*Z/(B-1) + P(C+2)) \lceil \log_Z P/B \rceil$ per C=10 e P=50000

Z	B=101	B=501	B=1001
2	5490000	4214000	3606000
5	2500000	1815000	1807500
10	1950000	1220000	1210000
50	1700000	1300000	625000
100	2200000	700000	650000
250		850000	725000
500		1100000	850000
1000			1100000

- Quanto visto può ovviamente estendersi a considerare il caso in cui anche per le scritture si operi in maniera sequenziale

I/O random e sequenziali (4)

- Il grafico riporta i costi di esecuzione assumendo P = 8192, C = 10 e Tt = 1 msec



Considerazioni finali

- L'ordinamento di file è stato oggetto di molteplici studi condotti nell'arco di alcuni decenni. Quanto mostrato ignora volutamente importanti aspetti delle moderne architetture, che sfruttano tecniche di partizionamento e parallelismo, anche in configurazioni distribuite su più nodi di una rete.
- Sul sito web http://sortbenchmark.org/, sono disponibili vari benchmark per valutare gli algoritmi di external sort e sono reperibili notizie aggiornate sugli approcci più performanti. Uno dei vincitori del 2019 è:

Tencent Sort
100 TB in 134 Seconds
512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz,
512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD,
100Gb Mellanox ConnectX4-EN)

The sort application consists of three major phases:

1) *reading from storage and partitioning the data into non-overlapping ranges according to the sort keys,*
2) *distributing the ranges to the destination nodes (shuffle), and*
3) *at the destination nodes combining equivalent ranges from all the nodes into a sorted output file.*

Sorting con B^+-tree

- Se l'indice è clustered, le pagine del file sono ordinate (almeno logicamente)
  - Costo = L+P (foglie dell'albero+pagine del file)
  - Se l'indice è struttura di memorizzazione primaria, costo = L

- Se l'indice è unclustered, ogni record causa la lettura di una pagina
  - Costo = $L+N \approx N$ (numero di record)
  - Se però tutti i campi che interessano sono all'interno dell'indice, non è necessario accedere al file dati, quindi:
    Costo = L

Schema di riferimento

- **S**ommelier(sid smallint,
  snome char(40),
  livello int,
  età int)
  - Len(S)=50B, N(S)=40K, P(S)=500, TP(S)=80
- **V**ini(vid smallint,
  vnome char(20),
  cantina char(18))
  - Len(V)=40B, N(V)=10K , P(V)=100, TP(V)=100
- **R**ecensioni(sid smallint,
  vid smallint,
  anno smallint,
  rivista char(34))
  - Len(R)=40B, N(R)=100K , P(R)=1K, TP(R)=100

Selezione

- Supponiamo di avere una query del tipo:

```
SELECT *
FROM   Recensioni R
WHERE  R.rivista='Sapore DiVino'
```

- Qual è il metodo di accesso migliore per la sua risoluzione?

- La scelta dipende da diversi fattori, quali:
  - Quanti record compongono il risultato?
  - Esiste un indice in grado di gestire la query?
  - Che tipo di indice è?
  - …

Stima del numero di risultati

- Ovviamente, non possiamo sapere a priori il numero di risultati di una query, quindi dobbiamo effettuare una stima: E = f * N
  - N è il numero di record in input
  - E è il numero atteso (Expected) di record
  - f è detto fattore di selettività e dipende dal predicato usato nella selezione

    Attenzione! Una query si dice molto selettiva quando f è piccolo

- Se si suppongono uniformemente distribuiti i valori dell'attributo coinvolto nel predicato di selezione: f = EK/NK
  - EK è il numero di valori attesi nel risultato

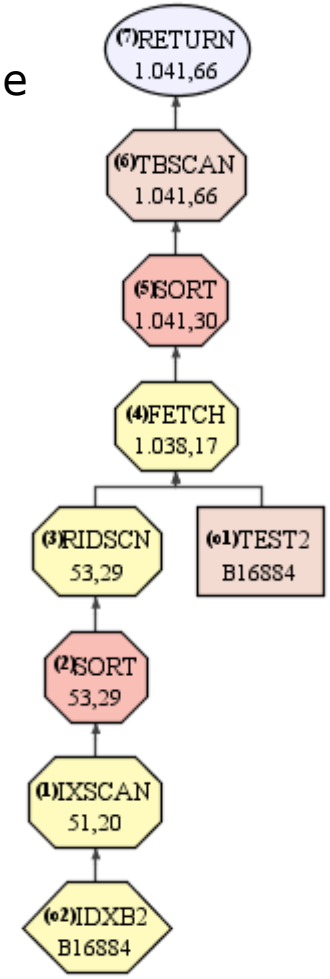
NB: Se la selezione opera su una relazione del DB, allora N è noto dai cataloghi di sistema. Dettagli su questi aspetti e su come si valuta la selettività nel caso generale si vedranno più avanti

Selezione: costo

- In assenza di indice è necessario leggere tutto il file dati: costo = P
- Indice B^+-tree: costo = h-1 + f * L + costo file dati
  - clustered: costo file dati = f * P
  - unclustered: costo file dati = EK * Φ(N/NK,P)
- Il costo dell'indice unclustered assume che per ogni valore di chiave che soddisfa il predicato si acceda al file dati e si reperiscano N/NK record
  - Quindi una stessa pagina può essere letta più volte
- Indice hash, predicato '=': costo = 1 + costo file dati

Selezione: alternativa con indice unclustered

- In caso di predicati che selezionano molti record, e più di un valore di chiave, può convenire ordinare i RID corrispondenti ai diversi valori prima di accedere al file dati:
  1. Scandisci l'indice e reperisci tutte le coppie (k,RID) che soddisfano il predicato
  2. Ordina le coppie per valore di RID crescente
  3. Scandisci il file dati ordinatamente, usando i RID
- Il costo di accesso ai dati si stima ora come $\Phi(f*N,P)$



IBM DB2

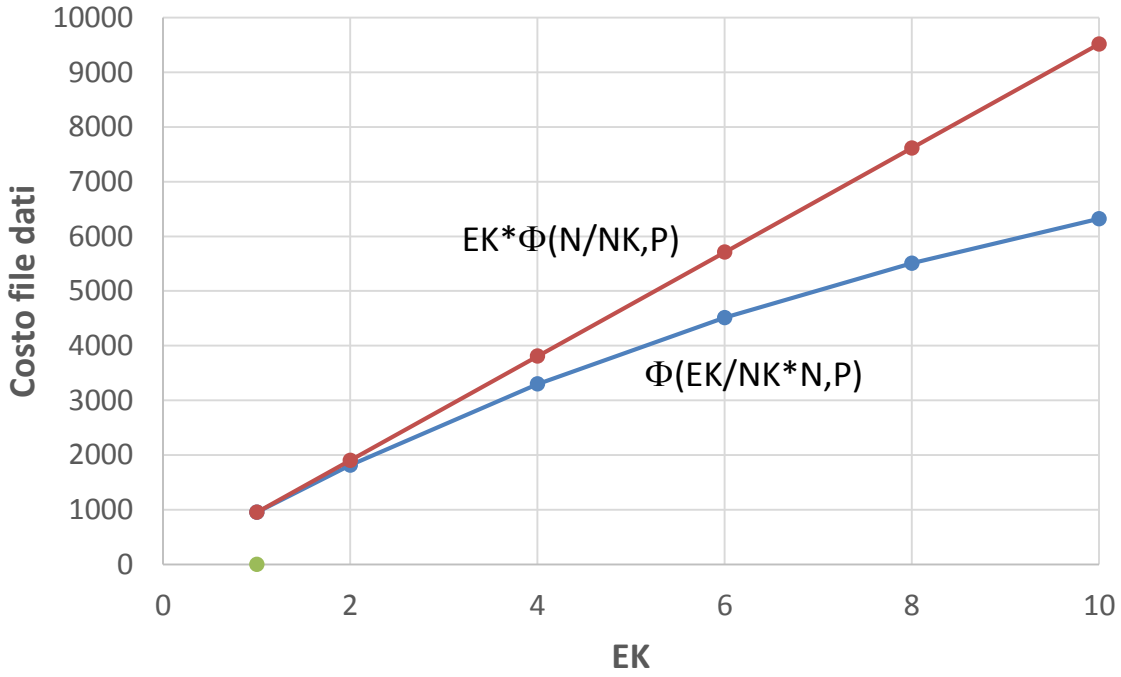
In DB2 questa strategia è realizzata dagli operatori fisici `IXSCAN(1)`,`SORT(2)`, `RIDSCN` e `FETCH(3)`

```
SELECT *
FROM B16884.TEST2
WHERE B < 50
ORDER BY A
```

Selezione: ordinare o no i RID?

- Il grafico mette a confronto i valori di:
  1. EK*Φ(N/NK,P): costo che si paga quando non si ordinano i RID
  2. Φ(EK/NK*N,P): costo nel caso di ordinamento dei RID

N = 10^5, P= 10^4, NK = 100



Selezione con condizione complessa

- Quanto visto vale se la condizione WHERE fa riferimento a un solo attributo

- In caso contrario:
  - Per prima cosa si riscrive la condizione in forma normale congiuntiva (CNF)
  - Quindi, occorre valutare se esiste un indice in grado di gestire la condizione in CNF

Condizioni risolubili (hash)

- Se abbiamo un indice su (R.sid, R.vid, R.rivista) possiamo usarlo per condizioni come:
  - rivista='Sapore DiVino' AND sid=3 AND vid=5

  ma non per:
  - rivista='Sapore DiVino' AND sid=3 o
  - anno=2005

- Se abbiamo un indice su (R.sid, R.vid) possiamo usarlo per condizioni come:
  - rivista='Sapore DiVino' AND sid=3 AND vid=5

  con un passo ulteriore di eliminazione in RAM dei record che non soddisfano rivista='Sapore DiVino'

Condizioni risolubili (B^+-tree)

- Se abbiamo un indice su (R.sid, R.vid, R.rivista) possiamo usarlo per condizioni come:
  - rivista='Sapore DiVino' AND sid=3 AND vid=5 o
  - sid=7 AND vid=12

  ma non per:
  - rivista='Sapore DiVino' AND vid=5 o
  - anno=2005

- Se abbiamo un indice su (R.sid, R.vid) possiamo usarlo per condizioni come
  - rivista='Sapore DiVino' AND sid=3 AND vid=5

  con un passo ulteriore di eliminazione dei risultati

Condizioni risolubili

- Un indice hash può risolvere una condizione congiuntiva solo se essa contiene un termine di uguaglianza per ogni attributo chiave dell'indice

- Un indice B^+-tree può risolvere una condizione congiuntiva solo se essa contiene un termine per ogni attributo di un **prefisso** della chiave dell'indice
  - In questo caso, non è necessario che il predicato sia di uguaglianza
- I termini non risolubili di una condizione sono detti residui

- Se abbiamo un indice su (sid, vid, rivista) possiamo usarlo per condizioni come sid=3 AND rivista='Sapore DiVino' ?
  - Sì, perché la condizione ha un predicato (sid=3) per un prefisso della chiave dell'indice
  - E rivista='Sapore DiVino' ?

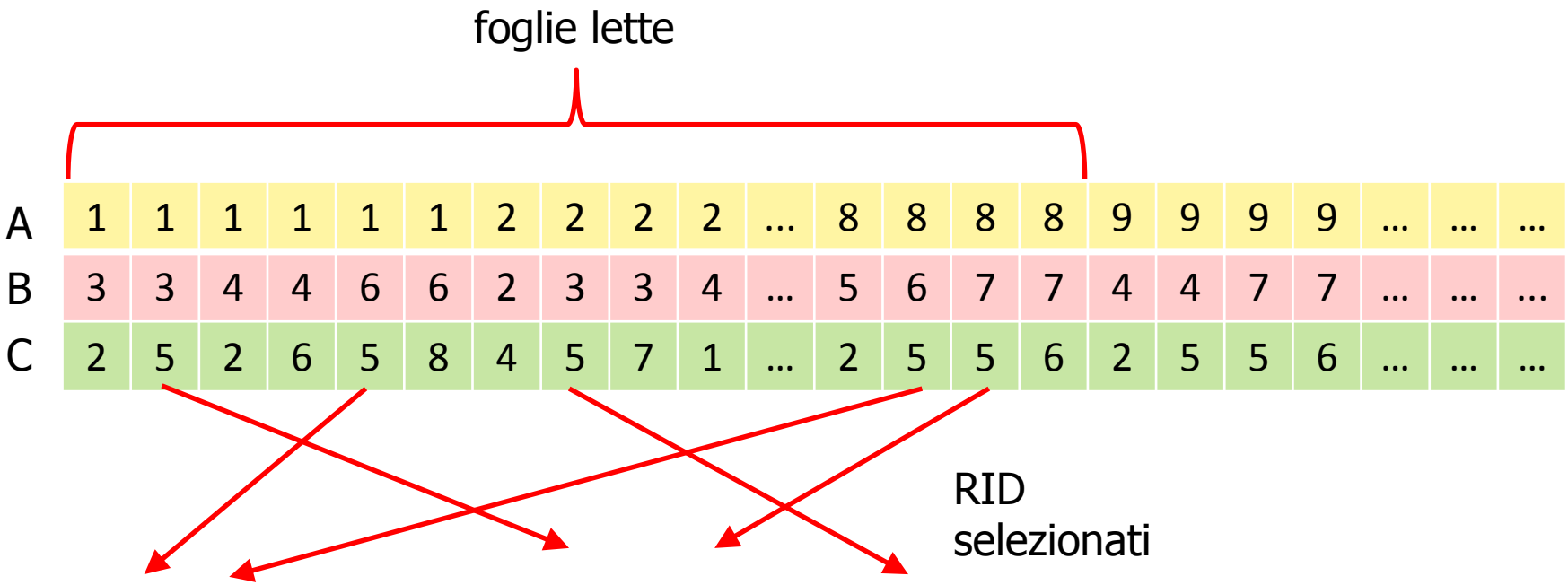
DB2: tipi di predicati

Ai fini della valutazione di un'interrogazione, DB2 distingue 4 tipi di predicato (in ordine di efficienza decrescente):

- Range-delimiting
  - Delimitano il range di foglie dell'indice cui accedere
- Index SARGable `(SARG = Search ARGument)`
  - Non delimitano il range di foglie, ma sono utilizzabili se si accede tramite indice
  - Es: indice su (sid, vid, rivista) ; sid=3 AND rivista='Sapore DiVino' : sid=3 è range-delimiting
    rivista='Sapore DiVino' è (solo) index SARGable
- Data SARGable (quelli che spesso vengono chiamati "residui")
  - Utilizzabili all'istante dell'accesso ai dati
- Residual
  - Es.: subquery correlate, ANY, ALL, …

Range-delimiting vs index SARGable

- La figura mostra il diverso effetto che hanno, sulle foglie di un indice (A,B,C), i predicati A < 9 e C = 5



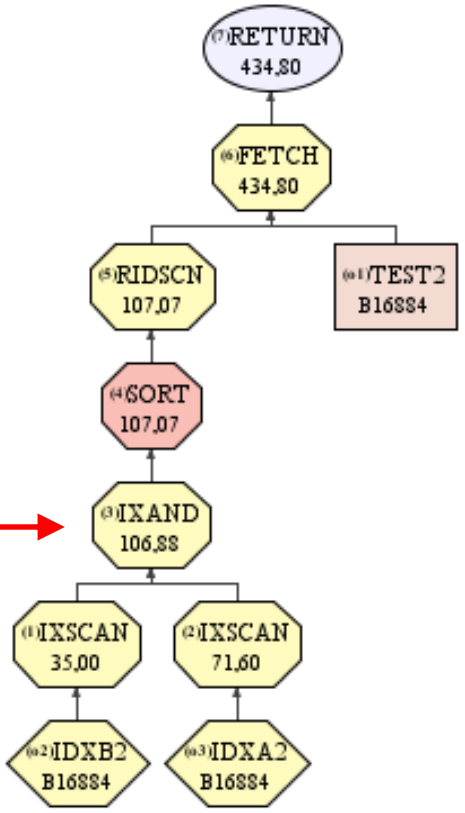
Effetto dei 4 tipi di predicati

- La tabella riassume gli effetti dei 4 tipi di predicati

	Range-delimiting	Index SARGable	Data SARGable	Residual
Riduzione Index I/O	Sì	NO	NO	NO
Riduzione Data I/O	Sì	Sì	NO	NO
Riduzione num. tuple	Sì	Sì	Sì	NO
Riduzione output finale	Sì	Sì	Sì	Sì

Selezione senza disgiunzioni

- Si usa il percorso di accesso più efficiente sui predicati risolubili e si valutano a posteriori i predicati residui
  - Costo = costo del percorso più efficiente
- Oppure, se esistono più predicati risolubili, si possono usare più indici facendo l'intersezione dei RID prima di accedere ai dati
- Il costo dati diminuisce all'aumentare del numero di indici utilizzati, per contro aumenta il costo di accesso agli indici



IBM DB2

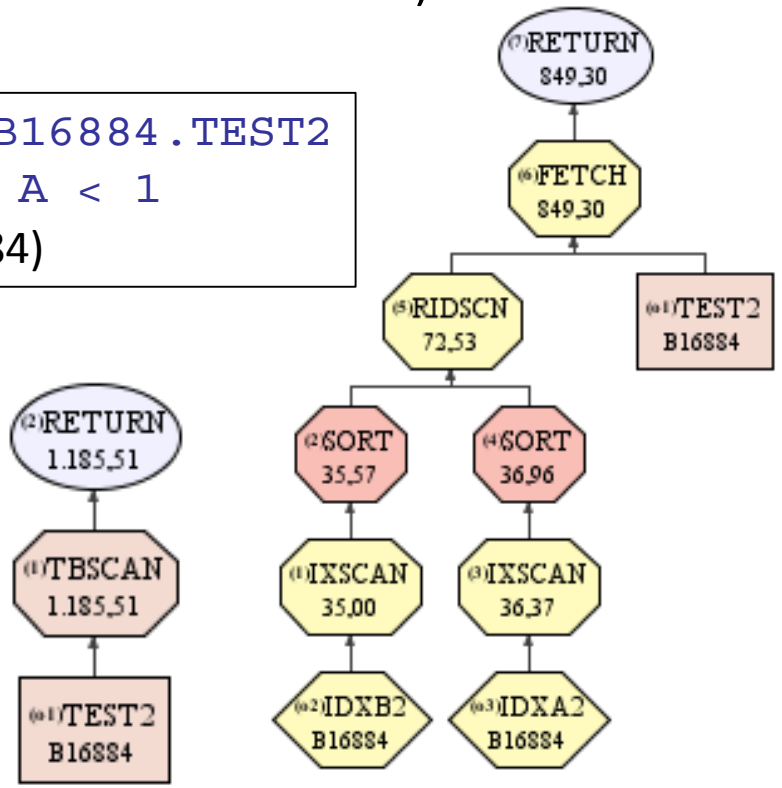
In DB2 questa strategia è realizzata dall'operatore fisico `IXAND`

```
SELECT * FROM B16884.TEST2
WHERE B < 2 AND A < 2
```

Selezione con disgiunzioni

- Se la condizione è una disgiunzione ed esiste anche una sola condizione non risolubile con indice, occorre scandire il file
- Se tutte le condizioni in OR sono risolubili con indice, si risolvono tutte e si prende l'unione (eventualmente facendo l'unione dei RID)…

```
SELECT * FROM B16884.TEST2
WHERE B < 2 OR A < 1
```
(4082 record su 16384)



- … se reputato conveniente

```
SELECT * FROM B16884.TEST2
WHERE B < 3 OR A < 3
```
(5000 record su 16384)

Esempio (i)

```
SELECT  *
FROM    Recensioni R
WHERE   R.sid=7
```

- N(R)=100K, P(R)=1K, NK(sid)=40K
- Fattore di selettività: f = 1/40K
- Indice B+-tree su sid: h = 2, L=⌈(40Kx2+100Kx4)/(4096x0.69)⌉ = 170
- Record nel risultato: ⌈100K/40K⌉ = 3

- Costo sequenziale = 1000

- Indice: numero di pagine lette del file: Φ(3, 1K) = 3

- Costo B^+-tree clustered = 1 + 1 + 1, unclustered = 1 + 1 + 3
- Costo hash clustered = 1 + 1, unclustered = 1 + 3

Proiezione

- Supponiamo di avere una query del tipo:

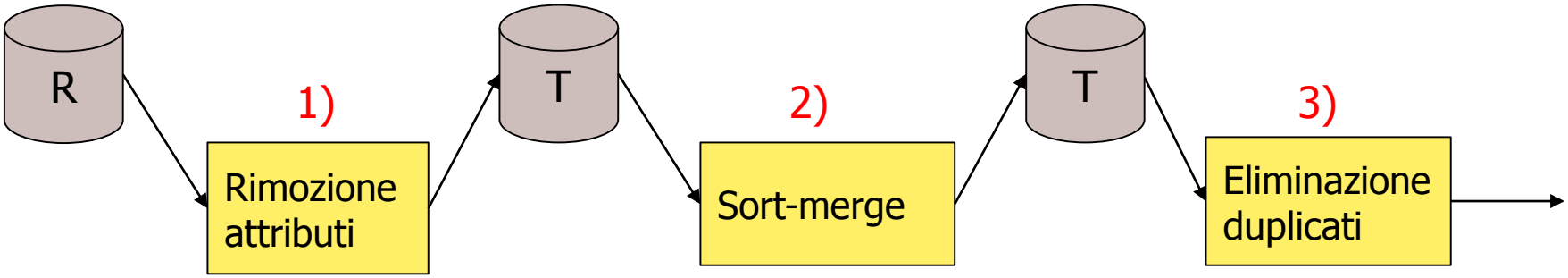
```
SELECT DISTINCT R.sid, R.vid
FROM   Recensioni R
```

- Quali sono le operazioni da compiere?
  - Cancellare gli attributi non richiesti (facile)
  - Eliminare i record duplicati (meno facile)

- Abbiamo 3 possibilità:
  - Usare il sorting
  - Usare l'hashing
  - Usare un indice

Proiezione basata su sorting

1) Si scandisce R e si producono i record con solo gli attributi richiesti (file T)
   - Costo = $P(R) + P(T)$
2) Si ordina il file T
   - Costo = $2P(T)\lceil \log_Z P(T) \rceil$
3) Si scandisce T (ordinata) eliminando i duplicati
   - Costo = $P(T)$

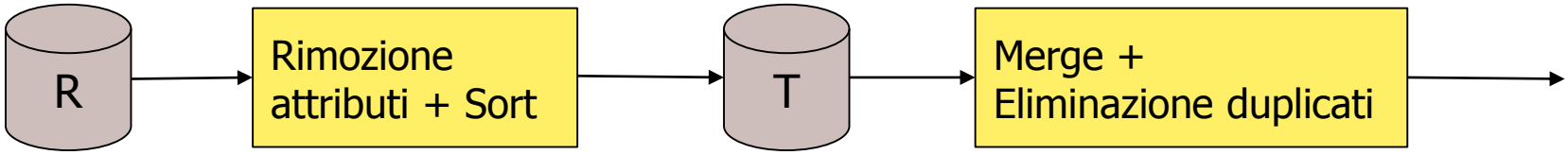


- È possibile diminuire il costo incorporando:
  - Il passo 1) nella fase di ordinamento di T
  - Il passo 3) durante il merge di T

Proiezione basata su sorting: esempio

- Se supponiamo che ogni record proiettato abbia dimensione = 10B
- Poiché Len(Recensioni) = 40B, e P(Recensioni) = 1000, T avrà 250 pagine
  - Costo creazione T = 1000 + 250 = 1250
- Se supponiamo di avere 20 pagine nel buffer, T può essere ordinato in 2 passi di sort-merge (1 sort + 1 merge)
  - Costo ordinamento T = 2 * 2 * 250 = 1000
- Considerando che il costo di eliminazione duplicati è pari a 250:
  - Costo totale = 1250 + 1000 + 250 = 2500

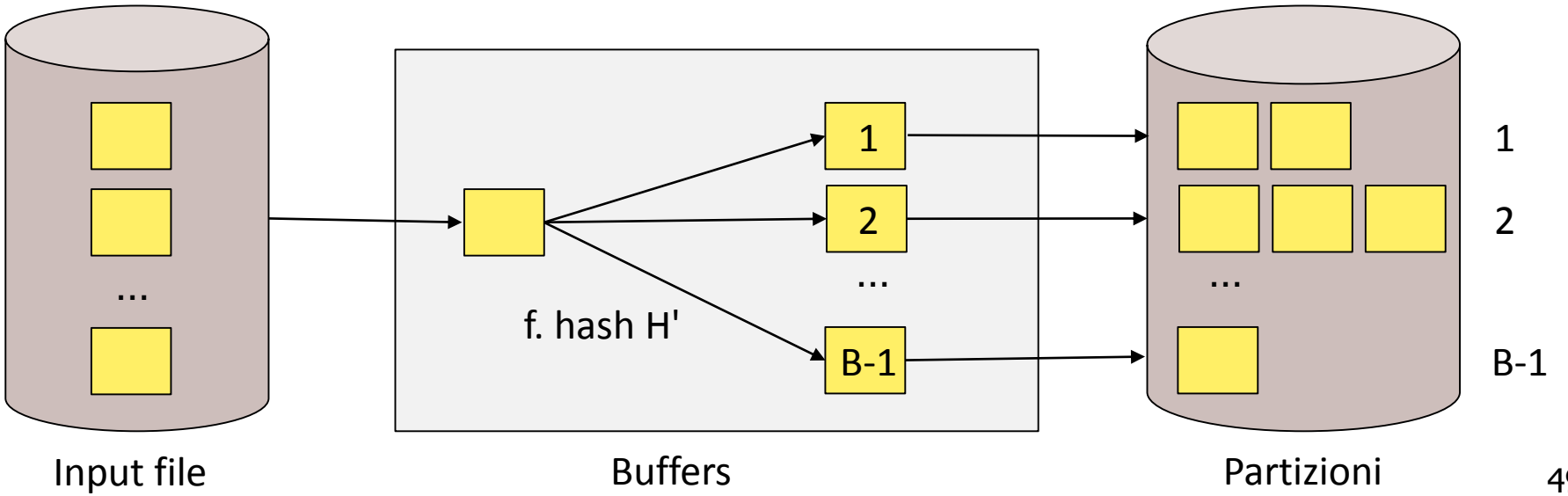
- Con l'ottimizzazione il costo si riduce a 1500:
  - Lettura Recensioni = 1000
  - Scrittura delle pagine di T ordinate con eliminazione degli attributi = 250
  - A questo punto ci sono 13 run (12 di 20 pagine + 1 di 10 pagine), che si fondono in un passo solo; costo = 250



Proiezione basata su hashing (1)

Se si ha un numero elevato B di pagine nel buffer si può usare un metodo hash a 2 fasi:

- Fase di partizionamento
  - Si leggono le pagine della relazione
  - Per ogni pagina letta, si eliminano gli attributi e si applica una funzione hash H' (a B-1 valori) agli attributi rimasti, distribuendo i record nelle B-1 pagine (un buffer serve per l'input)
  - Quando una pagina è piena la si scrive su disco



Proiezione basata su hashing (2)

- Fase di eliminazione dei duplicati
  - Si leggono, uno alla volta, i B-1 file generati
  - Per ogni pagina letta, si applica una nuova funzione hash H”
    (a B-1 valori), distribuendo i record nelle B-1 pagine ed eliminando i duplicati
  - I record rimasti sono il risultato

- Per file molto grandi (e/o bassi valori di B) si può ripetere il procedimento, ripartizionando usando una funzione hash H"', ecc.

- Nel caso di f. hash ideale, quanto grande può essere l'input per non dover ripartizionare?
  - Ogni partizione consiste di P/(B-1) pagine (si ignorano gli attributi eliminati)
  - Nel caso ideale, H'' le distribuisce uniformemente su B-1 buffer
  - Quindi P/(B-1) ≤ (B-1), ovvero $P \leq (B-1)^2$

Proiezione basata su hash: costo

- Costo fase di partizionamento = P(R) + P(T)
- Costo fase di eliminazione = P(T)
- Costo totale = P(R) + 2 P(T)
- Nell'esempio precedente, supponendo di avere abbastanza pagine nel buffer:
  - Costo partizionamento = 1000 + 250 = 1250
  - Costo eliminazione = 250
  - Costo totale = 1250 + 250 = 1500

Proiezione: sorting o hashing?

- La proiezione basata su sort-merge risulta preferibile quando vi sono molti duplicati o se la distribuzione dei valori (hash) è molto sbilanciata
- Inoltre con il sort i dati risultano ordinati
- In pratica:

DBMS	metodi usati
Informix	hashing
DB2	sorting
Oracle	sorting
Sybase	hashing & sorting
SQL Server	hashing & sorting

Proiezione basata su indice

- Per usare un indice, occorre che gli attributi mantenuti siano tutti contenuti nella chiave dell'indice
  - In questo caso si applicano le tecniche precedenti ai record nell'indice (senza accedere al file dati), eliminando i duplicati
  - Costo = costo delle tecniche precedenti per un numero di pagine pari a: L (B^+-tree) o P(H) (hash)

- Se l'indice è un B^+-tree e gli attributi sono un prefisso della chiave, i dati sono già ordinati
  - Basta scandire le foglie ed eliminare i duplicati al loro interno: Costo = L

Join

- Supponiamo di avere una query del tipo:

```
SELECT *
FROM   Recensioni R, Sommelier S
WHERE  R.sid=S.sid
```

- Qual è un modo efficiente per risolverla?
- Evidentemente quello di calcolare il prodotto Cartesiano e poi applicare il predicato di join non è molto furbo, in quanto il risultato intermedio contiene N(R)*N(S) record!

Join: considerazioni preliminari

- Dato il suo potenziale elevato costo di esecuzione, sono stati studiati molti algoritmi per eseguire il join efficientemente

- La varietà delle soluzioni (operatori fisici) dipende da diversi fattori, tra cui:
  - Indici esistenti
  - Ordinamento dei dati in input (e in output)
  - Quantità di buffer a disposizione
  - Possibilità di restituire o meno "subito" i primi risultati
  - …
- … questo sempre ignorando aspetti legati a parallelismo e distribuzione

- Non esiste un algoritmo di join “ottimo”…

Nested loops join

- L'algoritmo più semplice prevede il confronto di ogni record r di R con ogni record s di S:

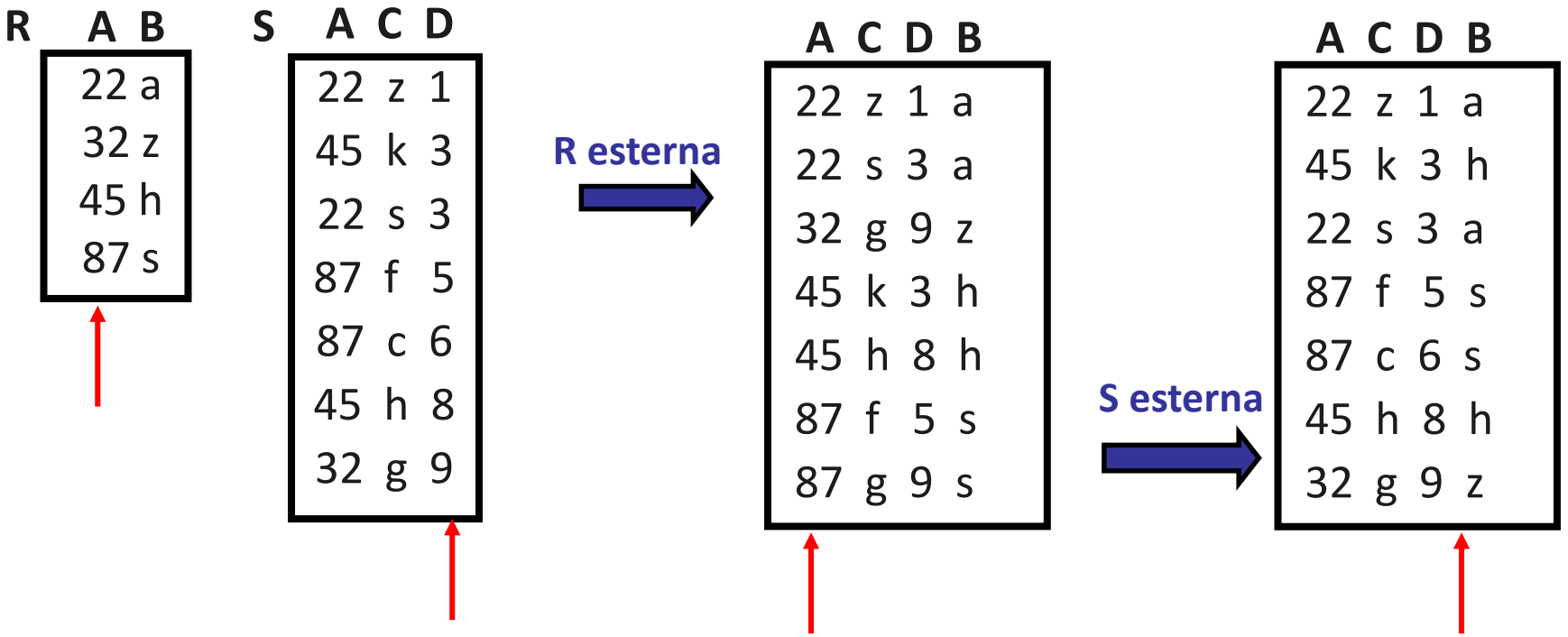
```
for each r in R
    for each s in S
        if JoinPred(r,s) add <r,s> to the result
/* JoinPred is the join predicate */
```

- Si scandisce la relazione "esterna" R e, per ogni record di R, si scandisce la relazione "interna" S
  - Costo = P(R) + N(R) * P(S) = P(R) + TP(R)*P(R)*P(S)
  - Nell'esempio: 1000 + 100000 * 500 ≈ 5 x 10^7
  - Supponendo 10ms per lettura = 140 ore!
  - Invertendo il ruolo di R e S: 500 + 40000 * 1000 ≈ 4 x 10^7
  - Costo = P(S) + N(S) * P(R) = P(S) + TP(S)*P(S)*P(R)

- Siccome il fattore dominante è il secondo, conviene avere come esterna la relazione avente record più "grandi" (deriva da N = TP * P), ma in pratica i miglioramenti sono non significativi

Nested loops join: proprietà

- Un'importante proprietà del join nested loops è che preserva l'ordine della relazione esterna



- Questa proprietà può tornare utile per l'esecuzione di altre operazioni, per evitare di eseguire un sort successivamente, …

Page nested loops join

- La versione "paginata" del nested loops rinuncia alla proprietà vista a fronte di una riduzione del numero di I/O di un fattore pari al numero di tuple per pagina, TP, della relazione esterna

```
for each page p_R in R
  for each page p_S in S
    add to the result all pairs <r,s> in p_R and p_s
    that satisfy PredJoin(r,s)
```

  - Costo = P(R) + P(R) * P(S)
  - Nell'esempio: $1000 + 1000 * 500 \approx 5 \times 10^5$
  - Supponendo 10ms per lettura = 1.4 ore!
  - Invertendo il ruolo di R e S: $500 + 500 * 1000 \approx 5 \times 10^5$

- Siccome il secondo fattore è uguale, conviene avere come esterna la relazione con meno pagine (anche ora i miglioramenti non sono significativi)

Block nested loops join

- Entrambi gli algoritmi visti non sfruttano la presenza di più pagine nel buffer (usano solo 2 pagine per l'input e 1 per l'output)

- Se abbiamo B pagine buffer, allo scopo di minimizzare il numero di I/O, possiamo usare:
  - B−2 pagine per la relazione esterna
  - 1 pagina per la relazione interna
  - 1 pagina per l'output

```
for each block of B-2 pages in R
    for each page p_s in S
          add to the result all pairs <r,s> in the
          buffers that satisfy PredJoin(r,s)
```

Block nested loops join: costo

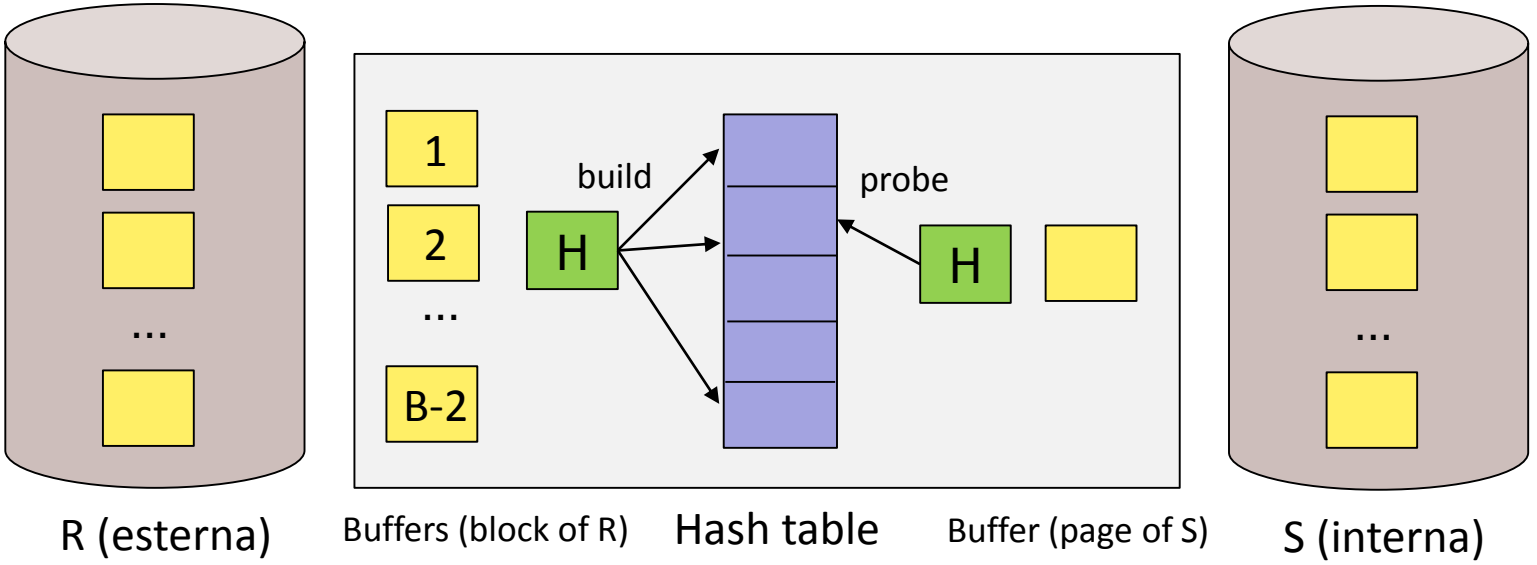
- La relazione interna S viene scandita P(R)/(B–2) volte
  - Costo = P(R) + P(R)/(B–2) * P(S)
  - Nell'esempio, supponendo B = 102: 1000 + 10*500 = 6000 (1 minuto)
  - Invertendo R e S: 500 + 5*1000 = 5500 (55 sec)

- È anche possibile usare più buffer per S
  - Come per il sort, si riducono le letture random, e quindi i tempi di latenza

- Provare a sviluppare un modello di costo che distingua tra I/O random e sequenziali, e quindi a ottimizzare l'allocazione dei buffer tra R e S

Block nested loops join: matching

- Un modo efficiente per trovare le coppie in join è costruire una hash table in memoria, in cui allocare i record del blocco di R
  - Usando la stessa funzione H si possono trovare più velocemente i match per ogni tupla di S
  - Ovviamente ciò è possibile solo per equi-join
- La figura mostra il caso in cui le B-2 pagine di R vengono lette sequenzialmente, e la hash table contiene i valori di join di R e puntatori ai record nei buffer con tali valori
- In alternativa, le B-2 pagine vengono lette 1 alla volta, e la hash table si costruisce allocando direttamente i record di R

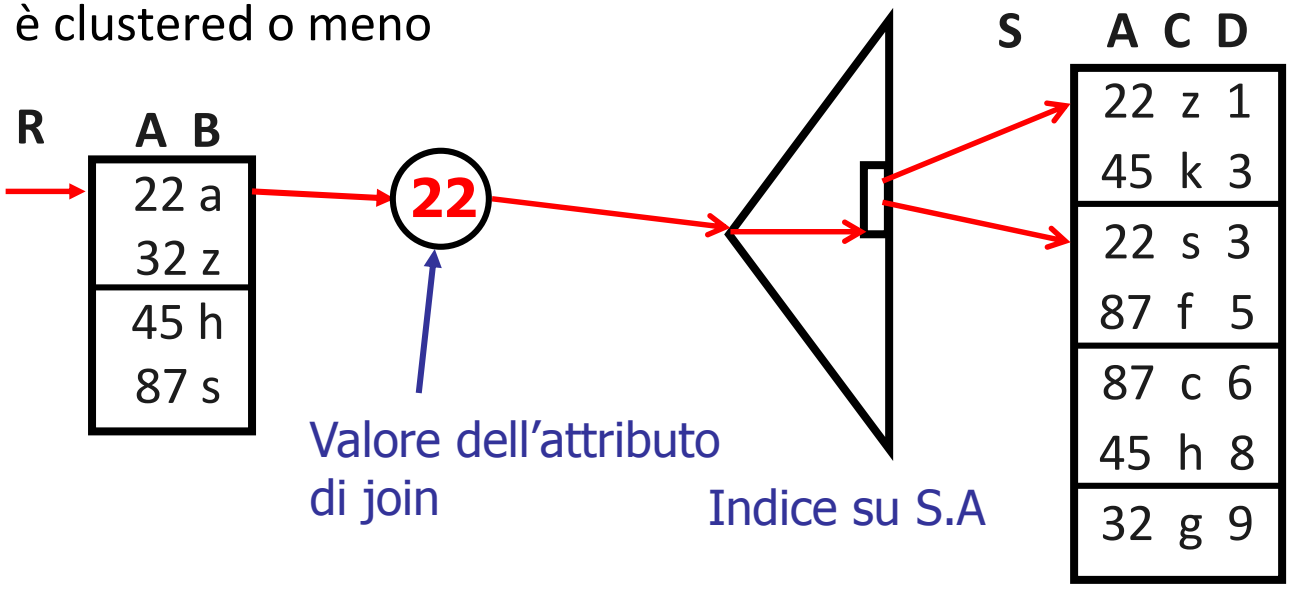


Index nested loops join

- Se S possiede un indice sull'attributo di join è possibile sfruttarlo
  - Non per la relazione esterna: perché?

```
for each r in R
    use the index on the join attribute of S to
    locate all s in S such that PredJoin(r,s),
    and add <r,s> to the result
```

- Costo = P(R) + N(R) * (costo indice + dati)
- Il costo per ogni record di R dipende dal tipo di indice (B⁺-tree/hash) e se l'indice è clustered o meno



Index nested loops join: esempio

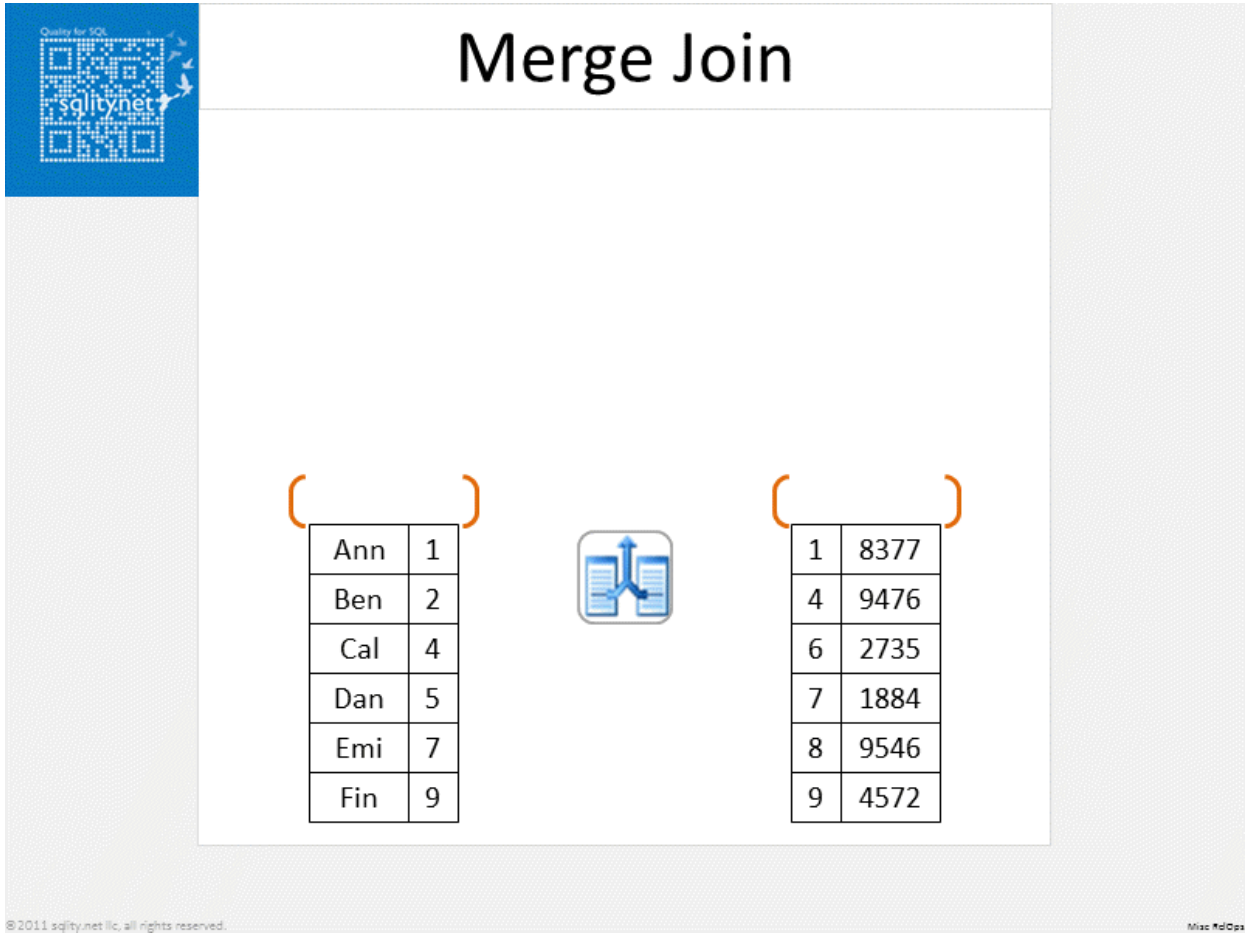
- Supponiamo di avere un indice hash su S.sid
  - Siccome sid è chiave di S, c'è un solo record di S per ogni record di R
  - Costo = 1000 + 100000*(1+1) = 200000 (33 min)

- Supponiamo di avere un indice hash su R.sid
  - Per ogni record di S esistono in media 100K/40K = 2.5 record di R
  - Se l'indice è clustered il costo per ogni record è 1:
    costo = 500 + 40000*(1+1) = 80000 (13 min)
  - Altrimenti il costo è 2.5 per ogni record:
    costo = 500 + 40000*(1+2.5) = 140000 (23 min)

- Conviene avere come esterna la relazione avente meno record

Merge-scan join (1)

- Entrambe le relazioni sono ordinate sugli attributi di join (eventualmente si ordinano, nel qual caso si chiama anche sort-merge join)
- In pratica è usato solo per equi-join



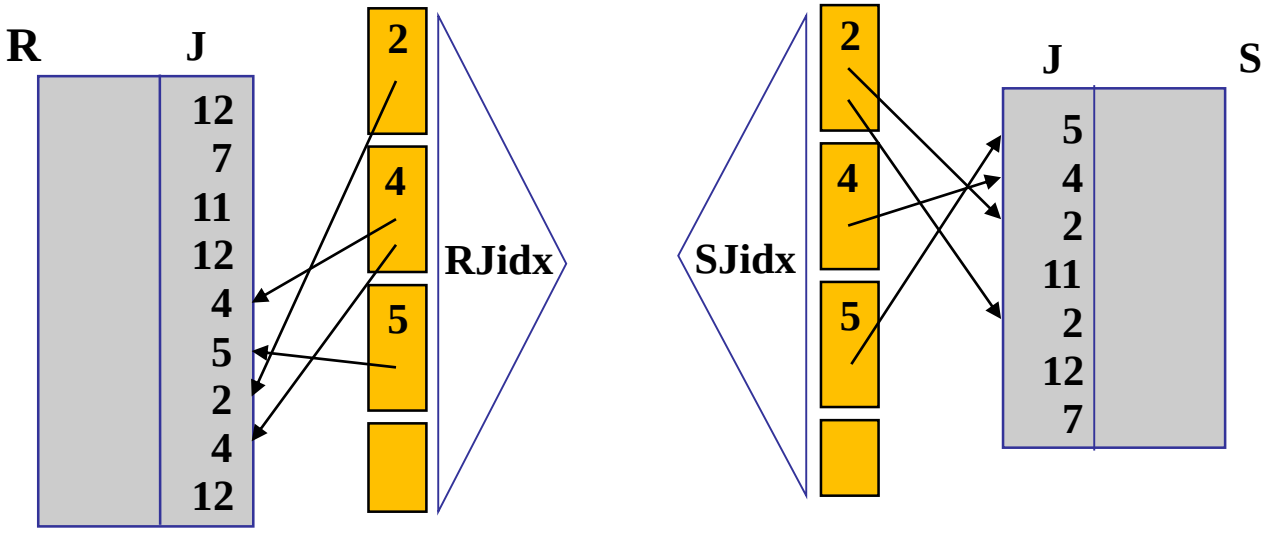
Merge-scan join (2)

- Il funzionamento si generalizza al caso di join FK-PK (in generale 1-N)
- Ogni relazione viene scandita una sola volta
  - Costo = P(R) + P(S)
  - Nell'esempio, costo = 500 + 1000 = 1500 (15 sec)

- L'algoritmo si complica se ci sono valori duplicati per entrambi gli attributi di join (associazione M:N) o in caso di non equi-join
- È necessario modificare l'algoritmo per fare backtracking (eventualmente vanno rilette alcune pagine)
- Il caso peggiore è quando tutti i record di entrambe le relazioni hanno lo stesso valore degli attributi di join (prodotto Cartesiano!)
  - Costo = P(R) + N(R)*P(S)

Merge-scan join con indici

- Se è presente un indice sugli attributi di join non è necessario effettuare il sort della corrispondente relazione
  - Se gli indici sono unclustered, il costo può comunque risultare elevato, dell'ordine di N(R) + N(S)



Hash join

- Il vantaggio del merge-scan è che viene sostanzialmente ridotto il numero di confronti tra i record delle due relazioni
- Un altro modo per ottenere questo effetto, usabile solo per equi-join, è partizionare i record mediante una funzione hash H, la stessa per entrambe le relazioni, e quindi confrontare solo i record appartenenti a partizioni omologhe
  - Lo schema è quello visto per la proiezione basata su hashing
- La fase di matching (o *probing*) può essere realizzata con un'altra funzione hash H', seguendo lo schema descritto per il block nested loops join
  - Si costruisce una hash table per ogni partizione della relazione esterna R. Deve quindi essere $P(R) \leq (B-1)^2$
- Costo = $3\,P(R) + 3\,P(S)$
  - Nell'esempio: 4500 (45 sec)

Condizioni di join non di uguaglianza

- Se il predicato di join non è di uguaglianza:
  - Index nested loops join può usare solo un B^+-tree
  - Hash join non è applicabile
  - Merge-scan join è in linea di principio applicabile, ma si complica notevolmente
  - Gli altri algoritmi non sono influenzati

- Nel caso di più predicati di join:
  - Index nested loops join può sfruttare un indice su tali predicati nella relazione interna
  - Merge-scan join considera un ordine lessicografico sulla combinazione di attributi per decidere come avanzare
  - Hash join partiziona sulla stessa combinazione di valori

Considerazioni finali

- Nested loops è l'algoritmo più semplice da applicare, e non ha limitazioni per quanto riguarda i predicati di join. Ha però costi elevati a meno di non avere sufficienti buffer a disposizione (block nested loops) o un indice sugli attributi di join della relazione interna (index nested loops)

- Merge-scan è in pratica utilizzato solo per equi-join 1-1 o 1-N, ed è conveniente se gli input sono già ordinati. In caso contrario la convenienza diminuisce (a causa del sort), anche nel caso di accesso con indici unclustered

- Hash join è tipicamente utilizzato per equi-join su input di grandi dimensioni e/o quando lo spazio in memoria è sufficiente a garantire che la fase di matching possa essere svolta in un solo passo

Outer joins

```
SELECT   D.*, E.*
FROM     Department D LEFT JOIN Employee E
                              ON (E.WorkDept = D.DeptNo)
```

- Si ricorda che un left (outer) join produce in output anche le tuple (*dangling*) dell'operando sinistro che non hanno match…
  - Dipartimenti senza Impiegati, nell'esempio (D =▷◁ E)
- … e il right join fa lo stesso per l'operando destro, ricordando che D =▷◁ E equivale a E ▷◁= D, …
- … e il full join D =▷◁= E aggiunge le tuple dangling di entrambi gli input
- D =▷◁= E = D =▷◁ E union E =▷◁ D

Outer joins

- Si consideri il left join D =▷◁ E (analoghe considerazioni per il right join)
- Nested loops: si usa D come relazione esterna e si aggiunge all'output ogni tupla di D che non trova match in E
- Merge-scan: si usa D come relazione esterna e, quando non si trova un match per una tupla di D, la si aggiunge al risultato
- Hash join: si usa come relazione esterna E. Dopo aver partizionato E e D, per ogni partizione di E si costruisce una hash table e si fa il probing per tutti i record di D nella partizione omologa. Se il probing non trova match si aggiunge il record di D all'output
- Per full join D =▷◁= E:
  - Nested loops: non è applicabile
  - Merge-scan: aggiunge le tuple dangling di entrambi gli input
  - Hash join: per aggiungere anche le tuple dangling della relazione esterna, quando si costruisce la hash table si aggiunge un flag per tener traccia di quali tuple hanno trovato un match. Al termine si fa un passo finale sulla hash table per collezionare tutte le tuple dangling

Altri tipi di join: Semijoin e Antijoin

```
SELECT S.*        -- SEMIJOIN: basta trovare 1 match
FROM   STUDENTI S
WHERE  S.Matr IN
       ( SELECT E.Matr FROM ESAMI )

SELECT S.*        -- ANTIJOIN: ok se non ci sono match
FROM   STUDENTI S
WHERE  S.Matr NOT IN
       ( SELECT E.Matr FROM ESAMI )

SELECT S.*        -- ANTIJOIN: formulazione con LEFT JOIN
FROM   STUDENTI S LEFT JOIN ESAMI E ON (S.Matr = E.Matr)
WHERE E.Matr IS NULL
```

Operatori insiemistici

- Dal punto di vista dell'implementazione, intersezione e prodotto Cartesiano sono casi particolari di join
- Le tecniche per unione e differenza prevedono l'eliminazione di duplicati
  - Possono essere realizzate mediante sorting o hashing
- Sorting (Costo = costo sort + P(R) + P(S))
  - Si ordinano R ed S usando tutti gli attributi
  - Si leggono ordinatamente R e S in parallelo
    - eliminando i duplicati (unione)
    - eliminando da R i record presenti in S (differenza)
- Hashing (Costo = 3 P(R) + 3 P(S))
  - Si partizionano R ed S usando una funzione H
  - Si leggono le partizioni di R e S in parallelo
    - eliminando i duplicati con una tabella H' su S (unione)
    - eliminando i record di R tramite H' su S

Funzioni aggregate

- Supponiamo di avere una query del tipo:

```
SELECT AVG(età)
FROM   Sommelier S
```

- L'algoritmo scandisce tutti i record, mantenendo informazioni riassuntive su quelli già elaborati

Funzione	Informazione riassuntiva
SUM	totale dei valori esaminati
AVG	totale e numero dei valori
MIN	minimo dei valori
MAX	massimo dei valori
COUNT	numero di valori

Group by

- Supponiamo di avere una query del tipo:

```
SELECT   S.livello, MIN(età)
FROM     Sommelier S
GROUP BY S.livello
```

- Sostanzialmente, occorre partizionare i record e calcolare le funzioni di aggregazione

- Tre possibilità per partizionare:
  - Uso di sorting
  - Uso di indice sugli attributi di raggruppamento
  - Uso di hashing

Group by con sorting

- Algoritmi analoghi alla proiezione + calcolo delle funzioni
  - Costo = costo sort + P(S)
- Il costo può essere diminuito aggregando già durante la fase di sort, mantenendo per ogni valore di raggruppamento le necessarie informazioni riassuntive
- Idem nei passi di merge
- Come si procede nel caso di funzioni “olistiche” (es., la mediana)?

Group by con indice

- Se l'indice contiene sia gli attributi di raggruppamento che quelli usati nelle funzioni, non è necessario accedere al file dati
  - Costo indice = L
    - Può diminuire nel caso di clausola HAVING sul prefisso della chiave

- Altrimenti, se l'indice è un B^+-tree e gli attributi di raggruppamento formano un prefisso della chiave dell'indice, lo si può usare per accedere ordinatamente ai dati

Group by con hashing

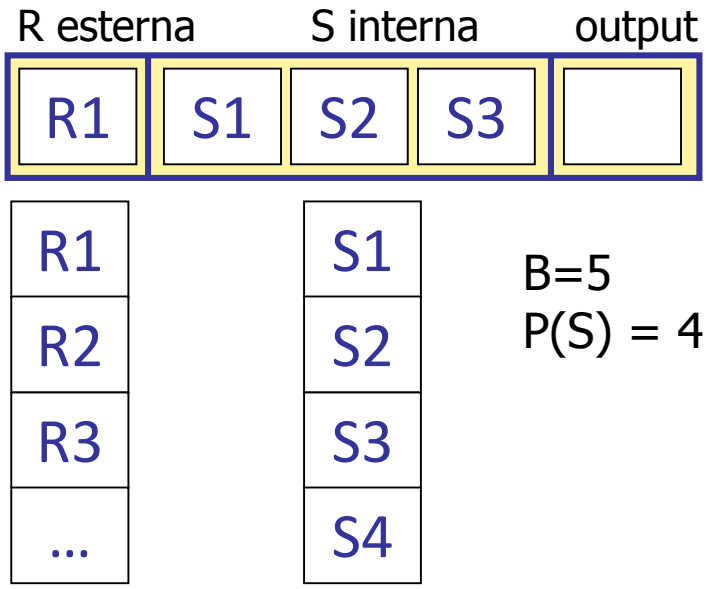
- Si costruisce una tabella hash in memoria sugli attributi di raggruppamento
  - La tabella conterrà anche le informazioni necessarie per il calcolo delle funzioni
- Al termine della scansione di S, si usano le informazioni per calcolare i valori
- È probabile che la tabella hash stia in memoria (*)
  - Costo = P(S)
  - Altrimenti, occorre partizionare la tabella usando una funzione hash

(*) Valgono le considerazioni fatte per la proiezione. Di fatto, il GROUP BY può essere visto come una "proiezione generalizzata", in cui per ogni combinazione di valori distinta (ovvero, per ogni gruppo) si calcolano anche delle statistiche

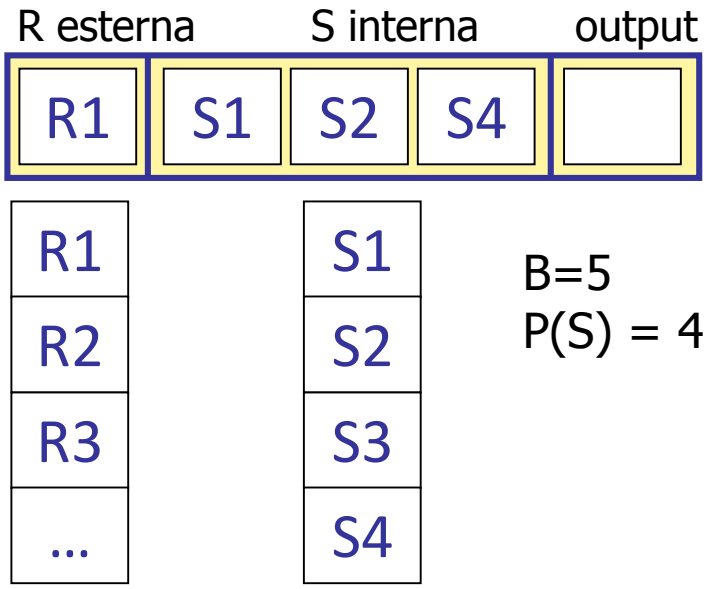
Uso del buffer

- Come abbiamo visto, l'uso del buffer pool è importante per diversi algoritmi, e per modellare "correttamente" i loro costi è importante considerare esplicitamente la dimensione del buffer pool
- Altre considerazioni da fare:
  - In caso di operazioni concorrenti la dimensione del buffer pool diminuisce (shared buffer pool)
  - Nell'accesso ai dati tramite indice, la probabilità di trovare una pagina dipende dalla dimensione del buffer pool e dalla politica di rimpiazzo
    - Se l'indice è unclustered, il buffer si riempie velocemente…
  - Se un'operazione presenta un pattern di accesso alle pagine che si ripete, si può aumentare tale probabilità scegliendo opportunamente la politica di rimpiazzo:
    - Ad esempio, per il nested loops join, se abbiamo B–2 pagine per la relazione interna, MRU è la politica migliore (MRU = Most Recently Used)
    - In generale, anche con meno buffer per l'interna MRU conviene rispetto a LRU (= Least Recently Used)

Page nested loops: LRU vs MRU



- Con LRU S4 rimpiazza S1
- Quando inizia la scansione per R2, S1 rimpiazza S2, S2 rimpiazza S3, ecc.
- Quindi la successiva pagina di S richiesta non è MAI nel buffer!



- Con MRU S4 ha rimpiazzato S3
- Quando inizia la scansione per R2, S1 e S2 sono in memoria, S3 rimpiazza S2, ecc.
- Per ogni scansione si legge da disco 1 sola pagina di S!

Operatori di modifica

- Le operazioni possibili sono `INSERT`, `DELETE` e `UPDATE`
- Solo `UPDATE` e `DELETE` richiedono un percorso di accesso per recuperare i record interessati
- L'`UPDATE` richiede la modifica dei soli indici relativi agli attributi aggiornati

- Il costo è costituito da 3 componenti:
  - Costo di accesso ai dati (letture)
  - Costo di modifica dei dati (scritture)
  - Costo di modifica degli indici (letture e scritture)

- Vediamo inizialmente cosa succede quando non è presente nessun ordinamento dei dati (e quindi non c'è nemmeno un indice clustered)

Insert

- Il caso di `INSERT` di un singolo record comporta l'aggiunta in coda al file dati (o in una posizione libera) e l'inserimento di una nuova entry (k,RID) in tutti gli indici definiti sulla table
- Nel caso di NI inserimenti (ad es. NI record provenienti da un'istruzione `SELECT`), il costo di modifica dei dati si può stimare pari a $\lceil NI/TP \rceil$
- Il costo di modifica delle foglie di un indice (trascurando eventuali split) si può stimare pari a:
  1. 2*NI se si eseguono le modifiche in modo indipendente
  2. $2*\Phi(KI,L)$ se si ordinano gli NI record per valore di chiave (e KI è il numero di valori per quel dato indice)
     - Se ogni valore di chiave occupa più di una foglia la stima non è accurata, e conviene usare $2*KI*L/NK$, in cui L/NK è il numero di foglie che contengono uno stesso valore di chiave
  3. In alternativa, essendo KI un valore da stimare, si può usare 2*NI*L/N
     - Se il numero di record aumenta dell'1%, l'intuizione è che questa modifica interessa l'1% delle foglie

Delete

- Il costo di ricerca dei record da cancellare si stima come visto in precedenza
- Il costo di modifica dei dati dipende da come si accede ai dati:
  - Scansione ordinata (sequenziale, RID ordinati)
  - Costo = $\Phi(E,P)$, con E = f*N numero di record da cancellare
  - Scansione disordinata (indice unclustered)
  - Costo = $EK*\Phi(N/NK,EP)$, con EK=f*NK
- Il costo di modifica delle foglie di un indice (trascurando eventuali underflow) si può stimare pari a:
  - 2*f*N se si eseguono le cancellazioni in modo indipendente
  - Si può ridurre a $2*\Phi(KD,L)$ se si ordinano i record per valore di chiave (e KD è il numero di valori da cancellare per quel dato indice)
  - Oppure (come per l'`INSERT`) si usa 2*E*L/N = 2*f*L
- Se si aggiorna l'indice usato per accedere ai dati è KD=EK e il fattore 2 si elimina (perché le foglie da modificare sono già state lette)
  - Come per l'`INSERT` si può usare EK*L/NK per valori di chiave che occupano più di una foglia

Delete: esempio (1)

```
DELETE FROM Recensioni
WHERE  R.rivista IN ('Sapore DiVino','PerBacco')
```

- N=100K, P=1K, NK(rivista)=50, NK(sid)=40K
- Fattore di selettività: f = 2/50
- Record da cancellare: E=f*N = 100K*2/50 = 4000

- Supponiamo di avere indici unclustered su R.sid e R.rivista
  - Indice su rivista: L= 150
  - Indice su sid: L= 170

Delete: esempio (2)

- Scan sequenziale
  - Costo accesso ai dati = 1000
  - Costo modifica dati = Φ(4000,1000) = 982
  - Costo modifica indice su rivista = 2*(2/50)*150 = 12
    - Si eliminano 2 valori di chiave su 50
  - Costo modifica indice su sid = 2*(2/50)*170 = 14
    - Ordinando per sid. In alternativa bisognerebbe stimare quanti valori KD di sid distinti ci sono nei 4000 record da cancellare (?) e usare la formula 2* Φ(KD,170)
    - Con cancellazioni indipendenti sarebbe 2*f*N = 8000 (!)
  - Costo totale = 2008

- Accesso con indice su rivista
  - Costo accesso ai dati = 2*Φ(2000,1000) = 1730
  - Costo modifica dati = 2*Φ(2000,1000) = 1730
  - Costo modifica indice su rivista (solo scritture) = (2/50)*150 = 6
  - Costo modifica indice su sid = 2*(2/50)*170 = 14
  - Costo totale = 3480

Update: costo di accesso

- Il costo di ricerca dei record da modificare si stima come visto in precedenza
- La sindrome di Halloween è un fenomeno che può prevenire l'uso di un indice su un attributo soggetto a modifica

```
UPDATE Sommelier
SET    livello = livello + 1
WHERE  livello BETWEEN 3 AND 5
```

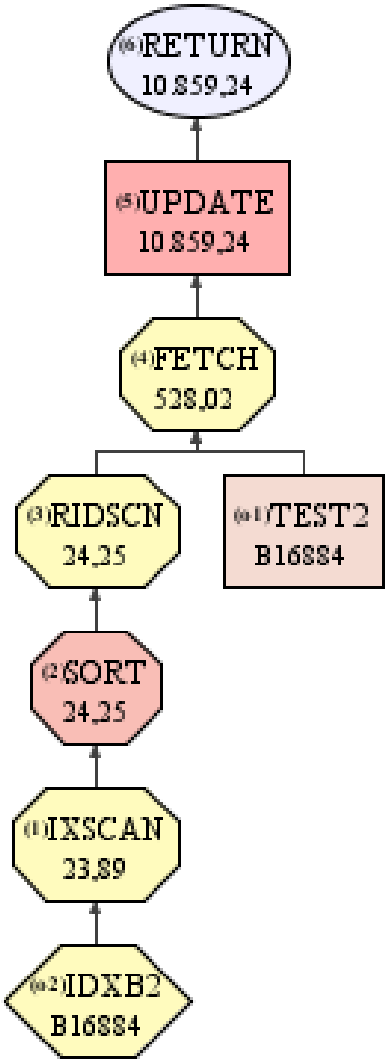
- Usando un indice su livello è possibile che alcuni record siano modificati più volte se il DBMS aggiorna l'indice valore per valore
  (3 diventa 4, che diventa 5, che diventa…)

Pat [Selinger] and Morton [Astrahan] discovered this problem on Halloween...
I remember they came into my office and said, '[Don] Chamberlin, look at this. We have to make sure that when the optimizer is making a plan for processing an update, it doesn't use an index that is based on the field that is being updated. How are we going to do that?' It happened to be on a Friday, and we said, 'Listen, we are not going to be able to solve this problem this afternoon. Let's just give it a name. We'll call it the Halloween Problem and we'll work on it next week.' And it turns out it has been called that ever since.

Sindrome di Halloween

- In DB2 il problema non si presenta (il risultato è corretto anche usando un indice soggetto a modifica):

```
UPDATE TEST2
SET    B = B + 1
WHERE  B >= 90
```



- Per evitare la sindrome di Halloween:
  - Si usa l'indice su B (`IDXB2`)
  - Si ordinano i RID per accedere ordinatamente ai dati
  - Prima si aggiornano tutti i record (1366 su 16384)
  - Dopo si aggiornano gli indici (su A e su B)

- Si noti il costo del `FETCH` (528.02-24.25) e quello dell'`UPDATE` (10859.24-528.02). Quest'ultimo include sia l'aggiornamento dei dati che quello degli indici

Update: costo di modifica dati e indici

- Come per il `DELETE`, ovvero:
  - Scansione ordinata (sequenziale, RID ordinati)
  - Costo = $\Phi(E,P)$, con E = f*N
  - Scansione disordinata (indice unclustered)
  - Costo = $EK * \Phi(N/NK,EP)$, con EK=f*NK

- La modifica di un indice prevede:
  - delete della vecchie entry (vedi `DELETE`)
  - re-insert delle nuove (vedi `INSERT`)

- Come per la modifica dei dati, i costi possono variare se la stessa foglia viene letta e riscritta al massimo una volta o no

Clustered data: DB2

- Nel caso di dati ordinati (clustered) su un attributo (la *clustering key*), quanto visto va esteso con considerazioni che variano in funzione dello specifico DBMS. A titolo di esempio:

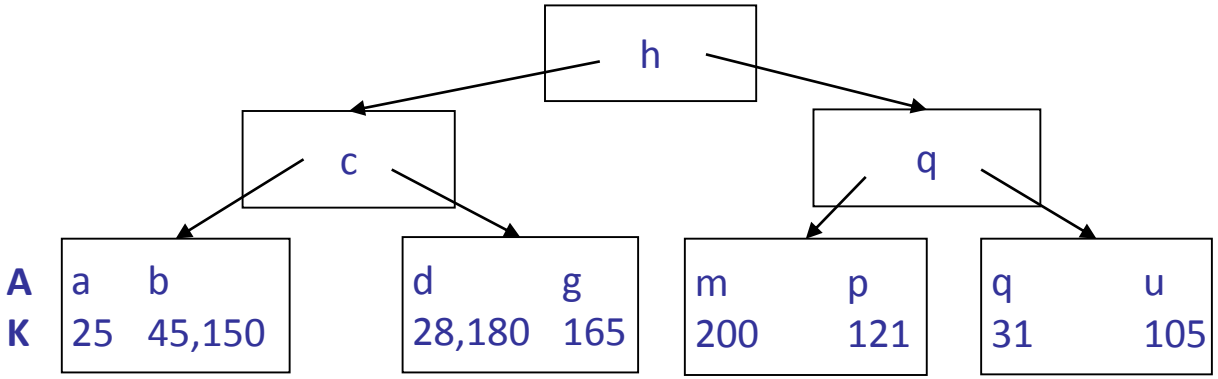
- In una table in cui è definito un indice clustered, DB2 *cerca* di preservare l'ordinamento a fronte di modifiche dei dati
  - Questo implica che in DB2 un indice può essere "più o meno" clustered
  - Può rendersi quindi necessaria una riorganizzazione dei dati quando il clustering è fortemente deteriorato (vedremo più avanti)
- Il vantaggio è che i RID, anche modificando i valori dell'attributo di ordinamento, non cambiano e quindi non ci sono effetti collaterali sugli altri indici

Clustered data: Oracle

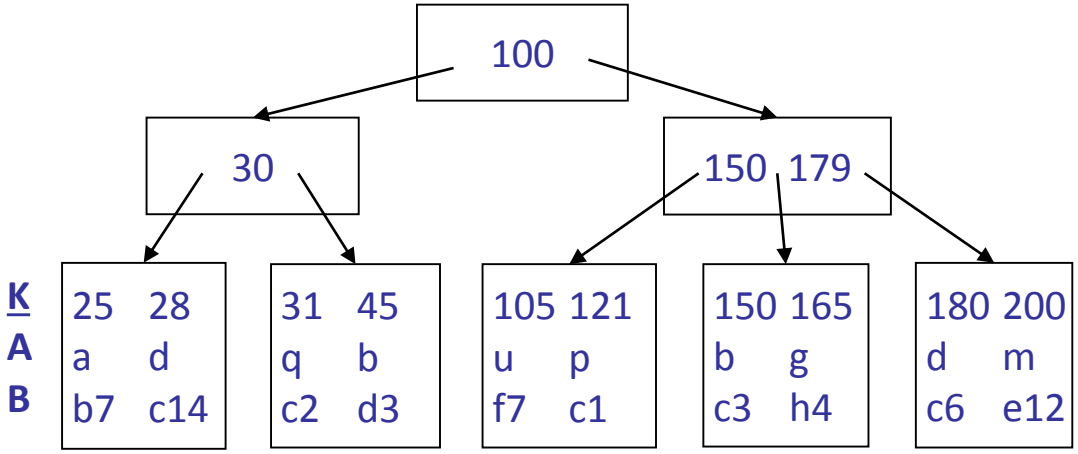
ORACLE®

- In Oracle il clustering dei dati equivale a definire una Index-Organized Table (IOT), ovvero un B^+-tree usato come organizzazione primaria dei dati (i record sono nelle foglie del B^+-tree), e la clustering key coincide con la primary key della relazione
- Per questo motivo, se una table è una IOT (anziché una normale heap table), negli altri indici (unclustered) non si usano RID "fisici", ma RID "logici" (i valori della primary key)
  - Altrimenti ogni modifica della IOT comporterebbe una modifica di RID negli altri indici (i dati si spostano fisicamente)
- L'uso di RID logici è penalizzante dal punto di vista delle prestazioni (introduce un livello di indirezione quando si accede con indice unclustered)
- Per questo motivo ogni RID logico ha anche una "physical guess" per accedere direttamente alla foglia che dovrebbe contenere il record referenziato

Clustered data: esempio di IOT e RID logici



Unclustered index on A



Index-Organized Table (on primary key K)

Clustered data: SQL Server

- SQL Server usa la stessa organizzazione di Oracle, ma permette di definire come clustering key qualsiasi combinazione di attributi (quindi non necessariamente la primary key)
- Per lo stesso motivo di Oracle, SQL Server usa RID logici negli indici unclustered

- Tuttavia, poiché la clustering key può non essere univoca, SQL Server ha bisogno di aggiungere a ogni valore duplicato un intero chiamato uniqueifier che permetta di individuare univocamente un record
- Questo uniqueifier è aggiunto a tutte le occorrenze di una clustering key (foglie degli indici unclustered, ma anche livelli intermedi dei B^+-tree)