Tecnologie delle Basi Di Dati M

Antonio Davide Calì

11 aprile 2014

WWW.ANTONIOCALI.COM
Anno Accademico 2013/2014
Docenti: Marco Patella, Paolo Ciaccia

Indice

Parte I
Operatori relazionali

1 Introduzione al query processing

Un sistema *dbms* commerciale deve saper gestire le query in maniera efficente ed efficace, quindi deve prima saper implementare la gestione delle query e successivamente effettuare un tuning, ovvero migliorarne le presetazioni.

Uno dei vantaggi dei *dbms* relazionali è che le interrogazioni sono composte da pochi operatori: un'implementazione efficiente di tali operatori permette quindi la risoluzione rapida delle query.

Esistono diverse alternative per la realizzazione dei vari operatori (cioè un operatore logico può essere implementato attraverso diversi operatori fisici) e raramente esiste un algortimo che è "sempre" migliore degli altri, infatti l'efficienza dipende da diversi fattori quali il numero di tuple, il numero di pagine, il buffer, la presenza di indici, ecc...

Gli *operatori relazionali* sono operatori che danno come risultato relazioni: si basano sull'algebra relazionale e si noti che non concidono perfettamente con gli operatori SQL, poichè, allo stesso modo in cui le relazioni non coincidono esattamente con le tabelle implementate, le tabelle relazionali non hanno ordinamento e possono presentare duplicati, cosa non possibile nelle relazioni poichè insiemi.

Notiamo che esiste una distinzione tra gli **operatori logici** dagli **operatori fisici** nei *dbms*:

- *Operatori Logici*: (es. JOIN) sono un'estensione di quelli dell'algebra relazionale, svogono 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 in modo da poter confrontare ogni volta le implementazioni per ottenere la miglior performance. Quando una query viene eseguita utilizzerà effettivamente un operatore fisico

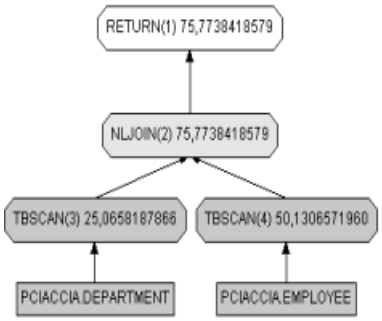
I modi alternativi per recuperare i record da una relazione si dicono *vie di accesso* (detti anche *metodi* o *cammini*). Si noti che sul disco abbiamo la presenza sia dei dati (i dati veri del database) sia degli indici e dunque in generale le vie di accesso possibili sono a *scansione sequenziale* o ad *accesso ad un indice con un predicato di selezione* gestibile dall'indice stesso. Il costo di una via di accesso dipende da alcuni fatori, come ad esempio il tempo o in base al numero di

operazioni di I/O (di nuovo considereremo trascurabile il costo di elaborazione in RAM, ma notiamo che non tutti i *dbms* effettivamente trascurano questo dato, si veda DB2 e il parametro TIMERON).

I modi alternativi per risolvere un'interrogazione si dicono *piani di accesso*: sono simigli agli alberi dell'algebra relazionale. Ogni piano di accesso fa uso di metodi di accesso ai dati e operatori fisici. Si veda la figura (Fig. 1) che risolve la query del codice sotto riportato che andremo in breve a commentare:

```
SELECT *
FROM Department, Employee
WHERE WorkDept = DeptNo
```

Figura 1:



Si noti che ogni nodo ha una sua rappresentazioni, ad esempio nelle foglie sono presenti i dati, al livello subito superiore invece sono presenti i *metodi di accesso*, salendo ancora di un livello troviamo un *operatore fisico* e infine la risoluzione della query (il numero presente nei nodi rappresenta il tempo di esecuzione in RAM cioè il parametro TIMERON di DB2).

2 Operatori relazionali

Di seguito una lista degli *operatori relazionali* di cui andremo a vedere il funzionamento:

1. Ordinamento
2. Selezione
3. Proiezione
4. Join
5. Operatori insiemistici (unione, differenza)
6. Group by

7. Operatori aggregati (avg, sum, count)

8. Operatori di modifica (update, delete, insert)

Assumiamo che il costo di produzione del risultato equivalga a zero, quindi lo ignoreremo: questo poichè in qualsiasi modo valutiamo la query, il risultato che essa produce è sempre lo stesso, quindi anche se cambiamo il piano di accesso (e quindi cambiamo il costo poichè ogni piano di accesso ha un costo diverso) il costo di produzione del risultato rimane invariato, dunque possiamo tranquillamente ignorarlo.

La **simbologia** che utilizzeremo è la seguente:

N(R) numero di record della relazione R

P(R) numero di pagine della relazione R

Len(R) lunghezza (in byte) di un record della relazione R

NK(R.A) numero di valori distinti dell'attributo A della relazione R

TP(R) numero di tuple per pagina: vale la seguente relazione $P(R) = \left\lceil \frac{N(R)}{TP(R)} \right\rceil$

B numero di pagine buffer

L(IX) numero di pagine foglia dell'indice IX

Ometteremo R e IX se chiari dal contesto.

Lo *schema di riferimento* su cui ci baseremo è il seguente.

`Sommelier (sid: integer, snome: string, val: integer, età: integer)`
`Vini (vid: integer, vnome: string, cantina: string)`
`Recensioni (sid: integer, vid: integer, anno: integer, rivista string)`

Gli attributi sottolineati sono gli attributi chiave della relazione associata. Indicheremo per semplicità con S la relazione Sommelier, V la relazione Vini e R la relazione Recensioni. I dati in nostro possesso sono

- Len(S)=50B - N(S)=40K - TP(S)=80 - P(S) = $\left\lceil \frac{N}{TP} \right\rceil = 500$
- Len(V)=40B - N(V)=10K - TP(V)=100 - P(V)=100
- Len(R)=40B - N(R)=100K - TP(R)=100 - P(R)=1K

Dove Len indica la lunghezza di un record in byte, N indica il numero di tuple nella relazione, TP il numero di tuple per pagina e infine P il numero di pagine per memorizzare tutti i record della relazione.

2.0.1 Algebra Relazione

Rivediamo, in sintesi, i tre operatori più importanti dell'algebra relazionale: *selezione*, *proiezione* e *join* (e alter join).

SELEZIONE Indicata con il simbolo σ, è un operatore *unario* che ha come dati di ingresso una *relazione R* e una *condizione booleana* (ricorda che una condizione booleana in SQL è un predicato che ha come possibili risultati *True False* o *Null*) e che restituisce come risultato tutte e sole le tuple della relazione che soddisfano la condizione booleana. Esempio $\sigma_{STUDENTI}(nome = '' \ pippo'')$

PROIEZIONE Indicata con il simbolo π, è un operatore *unario* che ha come dati di ingresso una *relazione R* e un'*insieme di attributi* della relazione stessa e che restituisce una nuova relazione, sottoinsieme della relazione di input R, in cui sono presenti solo le colonne degli attributi indicati. Si noti che se nell'algebra lineare questo potrebbe portare ad eliminare alcune tuple che nella nuova relazione potrebbero risultare duplicate, in SQL questo non è vero (si usa la cluasola DISTINCT per l'eliminazione dei duplicati). Esempio

$$\pi_{\sigma_{STUDENTI}(nome='' pippo'')}(matr, email)$$

In alcuni casi questi operatori possono commutare fra loro, in altri no. L'esempio appena mostrato non è commutativo, infatti risulta diverso da

$$\sigma_{\pi_{STUDENTI}(matr,email)}(nome = '' \ pippo'')$$

tanto che questo esempio non è neanche più valido (faccio una ricerca su un attributo non più esistente).

JOIN Indicato con il simbolo $><$, l'operatore di *join* esiste in molte varianti, il più semplice è il *join naturale*, cioè il join senza alcuna specifica, che applica un uguaglianza sugli attributi di ugual nome (è dunque un operatore binario). In caso tutti gli attributi abbiano nomi differenti, è possibile specificare esplicatamente gli attributi su cui fare join $><_{matr=matrs}$ e in questo caso si parla di *Theta-join*.

L'**alter-join**, il cui simbolo è $=><$ è diverso dal semplice join naturale, in quanto aggiunge anche tutte le tuple (chiamate *tuple dandling)* su cui non riesce a far *matching*. Le tuple dandling vengono completate, per gli attributi che non fanno parte dello schema ovvero per attributi non presenti in una relazione ma presenti nell'altra, con valori NULL.

Estensioni Si prenda come esempio la seguente query

```
SELECT CODC, COUNT(*)
FROM ESAMI
GROUP BY CODC
HAVING COUNT(*)>50
ORDER BY CONT(*) DESC
```

Essa raggruppa (GROUP BY) tutti gli esami per codice corso (CODC), e per ogni gruppo ne valuta l'*Having* (l'equivalente della clausola WHERE ma applicata ai gruppi): viene valutato per ogni codice corso quante tuple sono

presenti, e se sono superiori a 50 allora viene selezionato: di tutti i gruppi presi restituisco solo il codice corso e il numero di tuple ordinandolo (ORDER BY) in modo decrescente sul numero di tuple.

Di fatto abbiamo introdotto un gran numero di estensioni dell'algebra relazionale. Andiamo in velocità ad elencare le estensioni che più vengono usate:

- **Estensioni dell'algebra:** *GROUP BY, ORDER BY, HAVING*
- **Operatori insiemistici:** *UNION, INTERSECT, EXEPT*
- **Funzioni Aggregate:** *AVG, SUM, COUNT, MAX, MIN*

2.1 Ordinamento (sort)

Nonostante non sia un operatore vero e proprio (poichè l'algebra relazionale tratta insiemi e gli insiemi non sono ordinati), lo trattiamo poichè è un'operazione molto importante. Viene definito attraverso la *clausola* `ORDER BY.` Viene utilizzato nel *bulk-load* di un undice. Vedremo come elimina le copie di record (attraverso la clausola `DISTINCT`). Viene usato in diversi algoritmi di join e di group by.

Oltre al caso base che considereremo, esistono delle *varianti* all'operazione di ordinamento degne di nota: se richiesto, infatti, si possono eliminare i *duplicati* durante l'esecuzione del sort (codice di seguito) per una maggiore efficenza, oppure, se alcuni attributi in input non servono nell'output è possibile eliminarli durante l'esecuzione del sort stesso.

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

Vediamo subito un esempio (Fig. 2) di ordinamento che risolve la seguente query

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

Figura 2:

Matricola	CodiceFiscale	Cognome	Nome	DataNascita
216635467	RSSNNA78A53A944V	Rossi	Anna	13/01/1978
160239654	VRDMRC79H21F839X	Verdoni	Marco	21/06/1979
214842132	VRDCRL79H20G125J	Verdi	Carlo	20/06/1979
200643121	RSSDRA78M10A944V	Rossi	Dario	10/08/1978

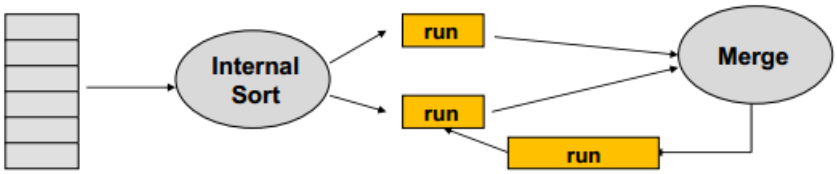
Matricola	Cognome	Nome
216635467	Rossi	Anna
200643121	Rossi	Dario
214842132	Verdi	Carlo
160239654	Verdoni	Marco

Bisogna innanzitutto distinguere tra gli algoritmi di ordinamento possibili in *RAM* e quelli invece in *memoria secondaria.*

Gli algoritmi di sort in RAM (Bubblie sort, Insertion sort, Shell sort, Merge sor, Heapsort, Quicksort, ...) hanno prestazioni generalmente molte buone, tipicamente $O(nlogn)$, $O(n^2)$ (ad esempio il *mergesort* ha sempre prestanzione $n{\cdot}logn$ mentre il *quicksort*, in media $n{\cdot}logn$, in alcune istanze ha complessità n^2). In genere richiedono che il dataset sia interamente contenuto in memoria (non è opportuno caricare tutto il data set in memoria virtuale poichè poi le pagine verrebbero allocate in modo casuale) eccetto per il *merge sort* che richiede la presenza di solo 2 elementi in memoria principale. Ecco dunque che ci viene possibile utilizzare l'algoritmo di *merge sort* in memoria secondaria: l'idea di base è che, siccome i dati non riescono a stare tutti in memoria, possiamo dividerli in sequenza, chiamate *run*, più piccole, ordinare le sequenze una ad una e infine fondere le sequenze un elemento per volta. Di fatto ogni sequenza ha le dimensioni massime di una pagina (e ad ogni passo la grandezza della sequenza avrà dimensioni massime di 2 pagine, poi di 4 pagine, etc...). Il primo passo di ordinamento utilizza un algoritmo di sort in RAM (ad esempio il quicksort).

In figura (Fig. 3) viene mostrato lo schema di principio

Figura 3:

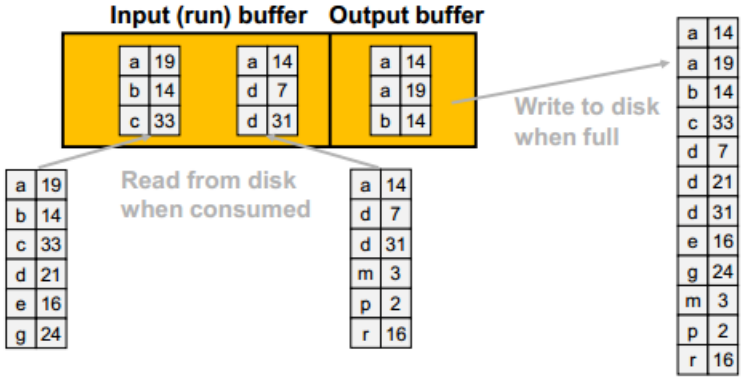


Facciamo un rapido esempio per ricordare il funzionamento del merge sort e applichiamolo ad un caso studio:
Ho 2 record per pagina.
L'input assegnato è: 3, 4, 6, 2, 9, 4, 8, 7, 5, 6, 3, 1, 2

1. passo: [3,4], [6,2], [9,4], [8,7], [5,6], [3,1], [2]
2. passo: [3,4], [2,6], [4,9], [7,8], [5,6], [1,3], [2]
3. passo: [2,3,4,6], [4,7,8,9], [1,3,5,6], [2]
4. passo: [2,3,4,4,6,7,8,9], [1,2,3,5,6]
5. passo: [1,2,2,3,3,4,4,5,6,6,7,8,9]

Analizziamo il costo del *Mergesort*: sono usate solo 3 buffer, 2 per l'input e una per l'output: quando il buffer di output è pieno lo devo scrivere su disco. 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. Si veda in figura (Fig. 4) come il Merge-sort gestisce il buffer.

Figura 4:



Se il numero di pagine del file di input è 2^k allora:

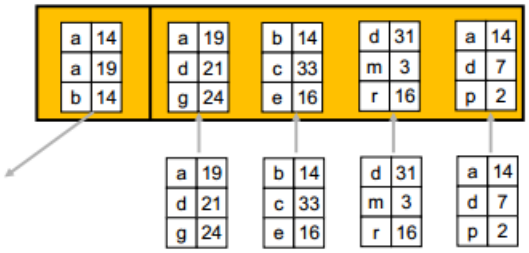
- il primo passo produce 2^k *run* di una pagina
- il secondo passo produce 2^{k-1} *run* di 2 pagine
- il terzo passo produce 2^{k-2} *run* di 4 pagine
- il k-esimo passo (l'ultimo) produce 1 *run* di 2^k pagine

Il numero totale di passi è $\lceil \log_2 P \rceil + 1$ (dove il $+1$ è dovuto al primo passo di *sort* prima evidenziato) e dunque il costo è pari a $P \cdot (\lceil \log_2 P \rceil + 1)$ letture e $P \cdot (\lceil \log_2 P \rceil + 1)$ scritture (dove il $+1$ in entrambi i casi è dovuto per leggere e scrivere i primi run per il primo passo di sort). Notiamo che l'ultimo passo di scrittura è non necessario, infatti l'ultimo passo implica avere il risultato e non è necessario scriverlo su disco, ma semplicemente restituirlo a chi aveva demandato l'operazione di ordinamento.

Facciamo un esempio. Dato il numero di pagine P uguale a $P = 8192$, applicando semplicemente le formule avrei un costo di $8192 \cdot (\log_2 8192 + 1)\, letture + 8192 \cdot (\log_2 8192 + 1)\, scritture = 229376$ operazioni di I/O: se ogni operazione I/O richiede 10ms l'ordinamento con merge-sort richiederebbe circa 38 minuti (impensabile). Un primo miglioramento si potrebbe ottenere ordinando B *pagine* alla volta invece che una, diminuendo il costo a $2 \cdot P \cdot (\lceil \log_2 \frac{P}{B} \rceil + 1)$ operazioni. Applicando la nuova formula all'esempio precedente e imponendo $B = 11$, l'operazione di ordinamento richiederebbe 30 minuti invece che 38 (ma è ancora troppo!).

Guardiamo ora un altro algoritmo di ordinamento, il *sort-merge a Z vie*. Notiamo che se anche il costo del *mergesort* è $O(P \cdot \log P)$ il buffer non viene utilizzato al meglio durante l'operazione di fusione: infatti l'operazione di fusione effettuata solo su due run alla volta è il fattore che più incide sul peggioramento delle prestazioni. Ecco allora l'idea base del sort-merge: supponenedo di avere B=Z+1 pagine nel buffer a disposizione, invece che fondere 2 pagine alla volta se ne possono fondere Z alla volta (serve sempre una pagina in più per l'output). Aumentando il *fan-in* del passo di *merge* si aumenta la base del logaritmo. Si veda la figura (Fig. 5) per capire meglio l'idea di base.

Figura 5:



Vediamo come funziona l'algoritmo sort-merge a Z vie in pseudo codice.

```
leggi il file B pagine alla volta
ordina le pagine
scrivi le pagine 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
                scrivi sul buffer di output una pagina alla volta
```

e vediamolo dunque applicato ad un esempio.
B=Z+1=11 - P=8192

1. passo: produce $\lceil \frac{8192}{11} \rceil = 745$ *run* di 11 pagine ciascuna (i primi 744 run hanno 11 pagine$\rightarrow 744 \cdot 11 = 8184$), tranne l'ultimo di 8 pagine (per arrivare a 8192)

2. passo: *merge* a Z=10 vie che produce $\lceil \frac{745}{10} \rceil = 75$ run di 110 pagine (i primi 74 run hanno 110 pagine$\rightarrow 74 \cdot 110 = 8140$), tranne l'ultimo di 52 pagine (per arrivare a 8192)

3. passo: *merge a* 10 vie che produce $\lceil \frac{75}{10} \rceil = 8$ run di cui di 1100 pagine (i primi 7 run hanno 1100 pagine$\rightarrow 1100 \cdot 7 = 7700$) tranne l'ultimo di 492 pagine (per arrivare a 8192)

4. passo: *merge a* 8 vie (essendo Z=10 ma avendo solo 8 run so che è l'ultimo passo) che produce le 8192 pagine ordinate

Analizziamo il costo del *sortmerge a Z vie*: nell'esempio appena descritto il costo è dato dalla *lettura* delle 8192 pagine in 4 passi $\rightarrow 8192 \cdot 4 = 32768$ operazioni I/O più il costo dovuto alla *scrittura* delle 8192 pagine in 4 passi $\rightarrow 8192 \cdot 4 = 32768$ operazioni I/O per un totale di $32768 + 32768 = 65536$ operazioni di I/O (circa 11 minuti). Più in generale il *numero di passi* è dato da $\left\lceil \log_Z \left\lceil \frac{P}{Z+1} \right\rceil \right\rceil + 1 \approx \lceil \log_Z P \rceil$ e il costo è dato da P *letture* più P *scritture* per ogni singolo passo (quindi il costo è dato da $2 \cdot P \cdot numero\,di\,passi$). Si veda la figura (Fig. 6) che descrive il numero di passi da fare in funzione del numero di pagine P e del numero di vie Z.

Figura 6:

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

Alcune considerazioni: se Z risulta essere troppo grande, il *merge a Z vie* può essere costoso dal punto di vista della CPU e quindi, ad esempio, siccome il numero di passi tra Z=128 e Z=256 non varia è preferibile usare Z=128 per diminuire il costo computazionale dato alla CPU. Altre ottimizzazioni sono possibili per evitare i tempi morti della CPU in attesa di una lettura, ad esempio il *double buffering* in cui si leggono 2 pagine per ogni *run* cioè terminata di leggere la prima si legge immediatamente la successiva.

Anche avendo molto spazio a disposizione in RAM (quindi un B grande) la soluzione migliore di merge potrebbe non essere scegliere Z=B-1: distinguendo tra le letture random e le letture sequenziali si perviene a scelte più accurate infatti finora abbiamo supposto che tutte le letture effettuate da disco siano letture "random" quindi con costo 1, ma di fatto se considero letture sequenziali il costo diminuisce. Denominato con Ts il tempo di seek, Tr il tempo di latenza e Tt il tempo di trasferimento di una pagina, il modello considera che il costo di X letture di pagine sequenziali (quindi ho X pagine) sia

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

dove $C = \frac{Ts+Tr}{Tt} \approx 10 \div 50$ e si nota che la lettura sequenziale di pagine è data da $C + X$. Nel caso di lettura Random il costo di X pagine è invece dato da

$$X \cdot (Ts + Tr + Tt) = X \cdot (\frac{Ts + Tr + Tt}{Tt}) \cdot Tt = X \cdot (C + 1) \cdot Tt = (X \cdot C + X) \cdot Tt$$

quindi pago X volte il costo di seek.

Durante il *sort interno* si leggono e si scrivono B pagine alla volta (quindi sostituisco B a X). Il costo, usando Tt come unità di misura, si può stimare a circa $2 \cdot \frac{P}{B} \cdot (C + B)$ in cui $2 \cdot \frac{P}{B}$ rappresenta il numero di seek (si noti che la formula è approssimata se P non è multiplo di B). Si vede che ho ridotto il numero di seek di un fattore pari a B.

Se voglio sfruttare le letture sequenziali, anziche utilizzare Z buffer per ogni run, utilizzo Z frame per ciascun buffer e dunque per la fuzione si organizzano i B-1 buffer dedicati alla lettura in Z "frame" di FS pagine ciascuno (il che vuol dire che si leggeranno FS pagine alla volta).

Tentiamo di capire meglio: prendo un run e carico FS pagine alla volta (invece che una pagina alla volta) mettendole nel frame, ed essendo letture sequenziali non ho impatto nel costo. La run non la leggo una una pagina alla volta ma FS alla volta, riducendo di FS il tempo di seek.

Per ogni passo di fusioni si hanno i seguenti costi (le operazioni di scrittura rimangono tutte random): Lettura ha costo $\frac{P}{FS} \cdot (C + FS)$ e la scrittura ha ancora costo $P \cdot (C + 1)$, complessivamente il costo, utilizzando Tt come unità di misura, diventa

$$2 \cdot \frac{P}{B} \cdot (C + B) + \left(\frac{P}{FS} \cdot (C + FS) + P \cdot (C + 1) \right) \cdot \left\lceil \log_Z \frac{P}{B} \right\rceil$$

Considerando solo la parte che varia con Z (cioè le letture in fase di fusione) e ricordando che $FS = \frac{B-1}{Z}$ si deve minimizzare

$$\left(P \cdot C \cdot \frac{Z}{B-1} + P \cdot (C + 2) \right) \left\lceil \log_Z \frac{P}{B} \right\rceil$$

Esempio. $C = 10$ e $P = 50000$ al variare di B si ottengono i seguenti risultati (Fig. 7)

Figura 7:

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

Morale, anche avendo un modello più preciso ci si accorge che non ha senso per i *dbms* aumentare troppo Z, poichè, si veda la tabella, aumentando anche Z non si migliora il costo.

Quanto visto può ovviamente estendersi a considerare il caso in cui anche per le scritture si operi in maniera sequenziale. Si possono anche considerare gli effetti della cache.

L'ultimo ordinamento che controlliamo è il *sorting con B^+-tree.* Se l'indice è *clustered* (ordinato come il file/relazione) allora le pagine del file sono ordinate (almeno logicamente), dunque il costo è dato da $Costo = L + P$ dove L corrisponde al numero delle foglie dell'albero, mentre P al numero di pagine del file, se l'indice oltre ad essere clustered è una struttura di memorizzazione primaria (cioè memorizza i record e non i PID/RID) allora il costo è dato solo

da $Costo = L$. A differenza se l'indice è *unclustered* ogni record causa la lettura di una pagina e dunque il costo diventa $Costo = L + N \approx N$ dove N equivale al numero dei record; se però tutti i campi che ci interessano sono all'interno dell'indice allora il costo si abbatte e diventa $Costo = L$.

Si noti un ulteriore osservazione. Prendiamo come esempio un indice costruito sull'attributo COGNOME della relazione STUDENTI, che però risulta essere *unclustered.* La query che vogliamo risolvere è

```
SELECT cognome
FROM studenti
ORDER by cognome
```

Nonostante l'indice risulta *unclestered* il fatto che come output io voglia solo l'ordinamento sull'attributo COGNOME, il costo è semplicemente $Costo = L$ poichè l'indice risulterà ordinato per conto suo (anche se unclustered con la relazione) e non dovendo fornire dati ulteriori se non il cognome stesso il costo si abbatte, nel risultato non dovrà neppure restituire i RID, perchè tanto non ha senso accedere ai dati poichè le informazioni su come ordinare l'attributo cognome sono reperibili dall'indice stesso (creato appunto sull'attributo cognome). Se volessi anche mantenere i *duplicati* nella richiesta, allora posso sempre utilizzare esclusivamente l'indice, in quanto nell'indice ricordiamo sono salvate informazioni <chiave,RID> ma in caso di duplicati il RID è una lista di RID: di conseguenza nel risultato di ordinamento genererà tante "chiavi" dello stesso tipo quanto è lunga la lista di RID.

Se ora volessi risolvere questa query

```
SELECT cognome, matr
FROM studenti
ORDER BY cognome
```

l'indice prima descritto non basterebbe (poichè non ho l'informazione su matr che devo reperire sui dati), e si ricadrebbe nuovamente nel caso di costo pari a $Costo = L + N$. Ecco che, in molti casi, i sistemi *dbms* includono negli indici non solo la chiave per la ricerca, ma aggiungono valori di uno o più attributi reputati significativi evitando così di accedere ai dati aumentando dunque la performance.

Notiamo infine che non è possibile fare un sort con un indice hash, dato che l'indice hash sparpaglia i valori di chiave senza rispettare alcun ordine. L'utilizzo dell'indice B+tree per il sorting è un metodo alternativo all'utilizzo del merge sort a Z vie: quest'ultimo infatti (merge sort) richiede un passo importante nel "design" della struttura, in quanto bisogna sapere come allocare i buffer, se massimizzare il numero Z o tentare di raggiungere una buona soluzione introducendo le letture sequenziali (che richiedono di diminuire Z) ed ecco perchè l'alternativa col B+tree potrebbe riternersi parecchio valida.

2.2 Selezione

Si supponga di avere una query del tipo

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

Quale risulta il *percorso d'accesso* migliore per la sua risoluzione? La scelta dipende da molti fattori come ad esempio quanti record compongono il risultato o se esiste un indice in grado di gestire la query.

Poichè non è possibile sapere il numero di tuple che comporranno il risultato della nostra query, non possiamo far altro che effettuare una stima su questo numero. Da ora in poi chiameremo questa stima con E e risulterà equivalente a $E = f \cdot N$ dove f è detto *fattore di selettività* della query (si noti che più f risulta piccolo più la query è selettiva) mentre N è il numero di record della relazione sulla quale stiamo svolgendo la selezione. Si ricordi che N è un numero noto grazie alle statistiche presenti sui catologhi. Bisogna dunque valutare il *fattore di selettività f*. Se si suppongono i valori dell'attributo coinvolto uniformemente distribuiti nel predicato di selezione (che risulta essere un approccio semplicistico, che infatti, come vedremo più avanti potrà essere migliorato) allora f risulta essere $f = \frac{E_k}{N_k}$ dove E_k è il numero di valori attesi nel risultato (e nuovamente N_k è il numero di valori distinti presenti nell'attributo coinvolto nella selezione). Il valore E_k è un valore atteso, e varia a seconda del predicato che si sta applicando, ad esempio, se il predicato è di uguaglianza, cioè sto facendo una selezione per uguaglianza, posso immaginare che il valore atteso sia uno, cioè che solo una tupla abbia effettivamente il valore che sto cercando. Il valore di selettività f deriva da la supposizione di distribuzione uniforme, quindi posso immaginare che ogni valore dell'attributo comparirà con frequenza $\frac{N}{N_k}$, moltiplicando questo valore per il valore atteso E_k otterremo la stima E$\rightarrow E = E_k \cdot \frac{N}{N_k} = f \cdot N$ in cui appunto $f = \frac{E_k}{N_k}$.

Notiamo che il fattore di selettività cambia a seconda del *predicato* utilizzato come si vede nella seguente tabella (Tabella 1)

Tabella 1:

Predicato	**Fattore di selettività f**
$=$	$f = \frac{1}{N_k}$
IN set	$f = \frac{\lvert set \rvert}{N_k}$
$< v$	$f = \frac{(v-min)}{(max-min)}$
between	$f = \frac{(high-low)}{(max-min)}$
attributo non numerico e $< v$	$f = \frac{1}{2}$
attributo non numerico e *between*	$f = \frac{1}{3}$
P AND Q	$f = f_P \cdot f_Q$
not P	$f = 1 - f_P$
P or Q	$f = f_P + f_Q - f_P \cdot f_Q$

Notiamo che per gli attributi non numerici, si utilizzano i loro valori e il fattore di selettività non è dipendente da parametri. In caso della congiunzione $P\ AND\ Q$ o della disgiunzione $P\ OR\ Q$ stiamo effettuando un ulteriore semplificazione, infatti stiamo supponendo che i due predicati siano indipendenti l'un dall'altro (non siano dunque correllati), cosa che effettivamente non è vera, ma se lasciassimo cadere questa ipotesi, il modello risulterebbe eccessivamente complicato da analizzare poichè dovrebbe controllare le correlazioni di tutte le possibile coppie di attributi.

Analizziamo dunque i *costi* della Selezione:

- Se non è presente alcun *indice* e il *file* è *disordinato* allora il costo è uguale al numero delle pagine $Costo = P$
- Se non è presente alcun *indice* e il *file* è *ordinato* allora il costo è dovuto a una ricerca binaria $Costo = \log_2 P + f \cdot P$ in cui $\log_2 P$ è dovuto a una ricerca binaria all'interno del file, più effettivamente la percentuale di pagine $(f \cdot P)$ che immaginiamo risolvino la selezione
- Indice attraverso B+tree allora il costo è dato da $Costo = h + f.L + costo\,file\,dati$ dove h è l'altezza del B+tree e
  - Se indice *clustered* allora $costo\,file\,dati = f \cdot P$
  - Se indice *unclustered* allora $costo\,file\,dati = E_K \cdot \phi(\frac{N}{N_k}, P)$ dove E_K è il numero dei valori attesi mentre la funzione ϕ è la stima di Cardennas o Yao di quante pagine bisogna accedere per la lettura di un record. Si noti che stiamo supponendo di non ordinare le RID e di conseguenza bisogna effettuare per E_K volte l'accesso alle pagine, poichè è possibile che in momenti successivi si acceda alla stessa pagina. Se invece, una volta reperiti i RID li ordinassimo, l'accesso ai dati risulterebbe essere $costo\,file\,dati = \Phi(f \cdot N, P)$ questo perchè saremmo sicuri di accedere solo una volta ad ogni pagina, di contro però mi richiede l'ordinamento dei RID e il gioco potrebbe non valere la candela
  - Se l'interrogazione della selezione avviene attraverso il predicato IN occorre sempre ripartire dalla radice per ogni valore, dunque è come se l'indice risultasse essere *unclustered*
  - Se l'interrogazione fosse di disugaglianza (quindi non di intervallo e non di uguaglianza ma un semplice $>$ o $<$) potremmo evitare anche il costo dell'altezza dell'albero, poichè semplicemente potremmo accedere al livello delle foglie e scandire (a seconda dei casi dall'inizio verso la fine o viceversa) queste ultime per reperire il risultato
- Indice attraverso struttura *hash* e con *predicato di uguaglianza* ('=') allora il costo è dato da $Costo = 1 + costo\,file\,dati$

Notiamo che il costo di selezione finora descritto riguardava una condizione semplice, cioè se la condizione WHERE fa riferimento ad un solo attributo. Cosa succede in caso contrario?

Supponiamo che il mio predicato sia sì fatto $[P1\,OR\,(P2\,AND\,P3)]\,AND\,P4$: una tupla deve essere scelta se soddisfa il predicato P4 e o P1 oppure P2 e P3. Tentiamo di capire come un indice potrebbe essere di aiuto o meno. Se avessi un solo indice che riesce a risolvere P2 sarebbe utile? Sfortunatamente no, perchè anche se di tutte le tuple che soddisfano P2 possiamo poi selezionare solo quelle che soddisfano anche P3, questo non ci assicura che non stiamo dimenticando tuple che soddisfino P1, il che richiede quindi l'accesso ai dati in modo sequenziale. Il problema, come si vede, è dovuto alla presenza della disgiunzione OR.

Bisogna attuare alcune tecniche: in primo luogo bisogna riscrivere la condizione in *forma normale congiuntiva* (CNF o nota come prodotto di somme, in cui ogni AND è una somma e ogni OR è un prodotto) che trasforma, nell'esempio precedente, il predicato in $[(P1\,OR\,P2)\,AND\,(P1\,OR\,P3)]\,AND\,P4$, quindi occorre valutare se esiste un indice in grado di gestire la condizione siffata (cioè in CNF). Nuovamente bisogna distinguere il caso di indici *hash* o indici B+tree.

Introduciamo prima degli esempi per poi spiegare in maniera più generale ciò che stiamo descrivendo.

Supponiamo di avere un indice sulla relazione R (Recensioni) applicato agli attributi **(R.sid, R.vid, R.rivista)**.
In caso di *indice hash* possiamo utilizzare l'indice stesso per condizioni come
`rivista='Sapore DiVino' AND sid=3 AND vid=5`
ma non per
`rivista='Sapore DiVino' AND sid=3 OR anno=2005`
poichè per utilizzare l'indice hash ho bisogno di avere effettivamente tutti i dati necessari che sono stati usati per la costruzione dell'indice, quindi R.sid R.vid e R.rivista, e in questo ultimo caso, mancando l'attributo R.vid non risulta possibile accedere ai dati tramite indice.
In caso di *indice b+tree* applicato agli stessi attributi, esso risolvere query con condizioni del tipo
`rivista='Sapore DiVino' AND sid=3 AND vid=5 OR sid=7 AND vid=12`
ma non riesce con condizioni del tipo
`rivista='Sapore DiVino' AND sid=3 o anno=2005`
poichè non risulta possibile utilizzare l'indice B+tree in mancanza del "primo" attributo (R.sid) su cui l'indice è stato costruito.

Proviamo ora a cambiare l'indice e ad applicarlo sempre alla relazione R (Riviste) ma sugli attributi **(R.sid, R.vid)**, in questo caso sia che l'indice sia hash sia che sia costrutito con un B+tree esso risolve query con condizioni del tipo
`rivista='Sapore DiVino' AND sid=3 AND vid=5`
con però un ulteriore passo necessario per eliminare dei risultati (poichè attraverso l'indice il risultato ha un insieme più grande di tuple, infatti non è possibile eliminare attraverso l'indice quelle che hanno sid diverso da 3, dunque è necessario in un passo successivo ignorarle).

In generale un *indice hash* può risolvere una condizione congiuntiva solo se essa contiene *un termine di uguaglianza per ogni attributo chiave dell'indice* (cioè se il predicato è formato da almeno gli stessi attributi con cui ho costruito

l'indice e su questi attributi venga richiesta l'uguaglianza), mentre un *indice B+tree* è più flessibile poichè può risolvere una condizione congiuntiva solo se essa contiene *un termine per ogni attributo nel prefisso della chiave dell'indice* (ad esempio se l'indice è costruito su R.sid,R.vid,R.rivista lo posso utilizzare per risolvere predicati solo su R.sid o anche solo su R.sid,R.vid) ed in questo caso non è necessario che il predicato risulti essere d'uguaglianza. I termini non risolubili di una condizione sono detti **termini residui**.

Se la selezione risulta essere priva di *disgiunzioni* (non sono presenti OR) è possibile operare per più vie: si può utilizzare il percorso di accesso più efficiente sui predicati risolubili (cioè fra gli indici disponibili tali che risolvano uno dei predicati presenti fra gli AND, prendo quello che ha fattore di selettività f più piccolo, così da riuscire a "tagliare" di più) e si valutano a posteriori i predicati residui: dunque il costo equivale a $Costo = costo\,del\,percorso\,più\,efficiente$; oppure, se esistono più predicati risolubili, si può pensare di usare più indici portando il costo a $Costo = somma\,dei\,costi\,degli\,indici + costo\,dati$ in cui il costo dati equivale a $Costo\,dati = \phi(f_1 \cdot f_2 \cdot \ldots \cdot f_Q \cdot N,\, P)$ in cui ϕ rappresenta la solita stima di Cardennas o Yao di quante pagine bisogna accedere per la lettura di un record e in cui il numero di risultati (NR) della query è dato da $NR = f_1 \cdot f_2 \cdot \ldots \cdot f_Q \cdot N$ dove ogni f_i equivale al fattore di selettività del predicato i-esimo ed N è il numero dei record della relazione. Si noti che il costo dati diminuisce all'aumentare del numero di indici.

Vediamo ora cosa succede se la selezione contiene delle *disgiunzioni*. Se la condizione è una disgiunzione ed esiste anche una sola condizione non risolubile con l'indice (cioè ad esempio una condizione in OR non ha un indice associato) occorre necessariamente scandire il file (poichè essendo in OR con altre condizioni, non ho la certezza di star "ignorando" tuple che sono in realtà necessarie e dunque la scansione del file diventa necessaria). Se invece esiste almeno una condizione in AND (cioè fa parte di un gruppo in cui è presente un AND) risolubile con indice si usa il percorso d'accesso più efficiente. Infine se tutte le condizioni in OR sono risolubili con indice, si risolvono tutte attraverso l'indice e se ne prende l'unione (eventualmente facendo l'unione delle RID così da migliorare la performance, poichè così facendo prima si conoscono tutti i RID e solo successivamente si accede ai dati).

Facciamo ora alcuni esemepi di query di selezione e vediamone le caratteristiche

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

- $N(R) = 100K$, $P(R) = \lceil \frac{N}{TP} \rceil = 1K$, $NK(sid) = 40K$
- Fattore di selettività f $f = \frac{1}{40k}$

- Indice su attributo *sid*: $h = 2$, $L = \left\lceil \frac{(40k \cdot 2 + 100k \cdot 4)}{4096 \cdot 0.69} \right\rceil = 170$ dove il prodotto per due è dovuto alla grandezza in byte dei record e invece il prodotto per quattro è dovuto alla grandezza del puntatore
- Numero di record nel risultato: $\lceil \frac{100k}{40k} \rceil = 3$
- Costo sequenziale= 1000
- Costo su file ordinato= $\log_2 1000 = 11$
- Numero di pagine lette del file: $\Phi(3,\ 1k) = 3$
- Costo con indice B+tree clustere= $2 + 1 + 1$
- Costo con indice B+tree unclustered= $2 + 1 + 3$
- Costo con indice hash clustered= $1 + 1$
- Costo con indice hash unclustered= $1 + 3$

```
SELECT *
FROM Recensioni as R
WHERE R.anno>2000 AND R.rivista="Sapori DiVino"
```

- $N(R) = 100k$, $P(R) = \lceil \frac{N}{TP} \rceil = 1k$, $N(S) = 40k$
- Fattore di selettività f: $f = \frac{1}{2} \cdot \frac{1}{50} = \frac{1}{100}$
- Indice su attributo *anno*: $h = 2$, $L = \left\lceil \frac{(20 \cdot 2 + 100k \cdot 4}{4096 \cdot 0.69} \right\rceil = 142$
- Indice su attributo *rivista*: $h = 2$, $L = \left\lceil \frac{(1k \cdot 22 + 100k \cdot 4}{4096 \cdot 0.69} \right\rceil = 143$
- Numero di record nel risultato= $\lceil \frac{100k}{100} \rceil = 1k$
- Costo sequenziale= 1000
- Costo indice B+tree clustered su anno= $2 + \frac{142}{2} + \frac{1000}{2} = 573$
- Costo indice B+tree unclustered su rivista= $2 + \frac{142}{50} + \Phi(2k,\ 1k) = 870$
- Costo indice B+tree clustered su rivista= $2 + \frac{142}{50} + \frac{1000}{50} = 25$
- Costo indice B+tree unclustered su anno= $2 + \frac{142}{2} + \Phi(50k,\ 1k) = 1073$
- Costo indice B+tree clustered su (anno,rivista)= $2 + \frac{143}{2} + \frac{1000}{100} = 84$
- Costo indice B+tree clustere su (rivista,anno)= $2 + \frac{143}{50} + \frac{1000}{100} = 15$

2.3 Proiezione

Supponiamo di avere la seguente query

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

Quali sono le operazioni da compiere? In primo luogo bisogno eliminare gli attributi della relazione che non sono richiesti (questa operazione risulta facile), successivamente bisogna eliminare i record duplicati che possono apparire (operazione meno facile). Abbiamo 3 possibilità per operare: *sorting*, *hashing*, o utilizzo di un *indice*. Dobbiamo ora stimare la dimensione delle tuple su cui bisognerà a lavorare: si nota subito che se la proiezione è su un solo attributo allora la stima E è data da $E = NK$ (NK è il numero di valori distinti per l'attributo), mentre se almeno uno degli attributi della proiezione è un attributo *chiave* allora la stima diventa $E = N$ (con N numero di tuple della relazione, poichè grazie all'attributo chiave non potranno esserci duplicati). In tutti gli altri i casi la stima equivale a $E = min\{N, \prod_i NK_i\}$, cioè si prende il numero dei valori distinti di ogni attributo della proiezione e se ne fa il prodotto e si prende come stima il valore minimo tra il numero totale delle tuple della relazione o il prodotto appena effettuato.

Proiezione con Sorting La proiezione attraverso *sorting* richiede di scandire tutta la relazione R producendo nuovi record contenti solo gli attributi richiesti nella proiezione, scrivendoli su un nuovo file T. Il costo di questa operazione è $Costo = P(R) + P(T) = O(P(R))$, cioè devo leggere P(R) pagine (numero di pagine della relazione R) e scriverne P(T) (numero di pagine necessarie per contenere T), e ovviamente essendo $P(T) \leq P(R)$ il costo è O(P(R)). A questo punto si ordina il file T appena scritto usando una combinazione lessicografica di tutti gli attributi, il costo dell'ordinamento è $Costo = O(P(T) \cdot \log P(T)) = O(P(R) \cdot \log P(R))$. Effettuato l'ordinamento si scandisce il file T ordinato eliminando dunque i duplicati$\rightarrow$ $Costo = P(T) = O(P(R))$ e l'algoritmo di proiezione termina restituendo il file T ordinato e privo di duplicati. È possibile diminuire il costo incorporando il primo passo (l'eliminazione degli attributi) all'atto dell'ordinamento di T (dell'ordinamento della prima run di T per essere precisi) oppure incorporando il terzo passo (l'eliminazione dei duplicati) durante l'esecuzione del *merge* del file T. Facciamo un esempio: se supponiamo che ogni record proiettato (cioè il record dopo aver effettuato la proiezione) abbia dimensione $Len = 10B$ e il file T abbia $P(T) = 250$ pagine, allora il costo della creazione del file T è dato da $T = 1000 + 250 = 1250$. Se supponiamo di avere 20 pagine nel buffer, T può essere ordinato in 2 passi di *sort-merge* e dunque i costi diventano: $costo\, ordinamento\, T = 2 \cdot 2 \cdot 250 = 1000$, $costo\, eliminazione\, duplicati = 250$ (dovuta alla letture di tutte le pagine di T), $costo\, totale = 1250 + 1000 + 250 = 2500$. Con le ottimizzazioni prima descritte il costo arriva a 1750.

Proiezione con Hashing È possibile utilizzare una tecnica completamente diversa per la proiezione basata sull'hashing. Essa, però, richiede un numero elevato B di pagine nel buffer a disposizione. È strutturata in due fasi: la prima è la fase di *partizionamento*, la seconda (successiva) è la fase di *eliminazione dei duplicati*.

Nella *fase di partizionamento* vengono lette tutte le pagine della relazione R a cui stiamo applicando la proiezione: per ogni pagina letta si eliminano gli attributi che non si vogliono e si applica una funzione hash H_1 (a valori in [1, B-1] poichè il valore 0 si suppone essere il buffer da cui si prende in input la relazione) agli attributi rimasti (cioè quelli della proiezione) distribuendo i record nelle B-1 pagine. Quando una pagina risulta piena viene scritta su disco. Si noti che quando si distribuisce un record in un determinato buffer, si legge se quel record è già presente o meno nel buffer, in caso risulti essere duplicato lo si ignora. Il problema è proprio dovuto al fatto che se una pagina è piena viene scritta su disco perdendo dunque la possibilità di controllare se un record, che andrà a finire nel buffer appena svuotato poichè scritto nel disco, era già presente o meno: ecco allora che si rende necessaria la fase di *eliminazione dei duplicati*. In questa fase si leggono i B-1 file generati (possono essere anche di meno se ad esempio un buffer non è mai andato in *overflow*) durante la fase di partizionamento. Partendo dal primo file (che corrisponderà al primo buffer andato in overflow) si legge ogni pagina di cui è composto: si applica dunque una nuova funzione hash H_2 (sempre a valori in [1, B-1]) diversa da H_1 e si ridistribuiscono nuovamente i record nei vari buffer eliminando come prima spiegato i record duplicati: se di nuovo un buffer risulta pieno, esso verrà scritto nel disco e al passo successivo bisognerà nuovamente leggere i file generati con una nuova funzione H_3 diversa dalle precedenti. Il procedimento continua iterativamente finchè non si leggono tutti i file e finchè, attraverso una funzione H_i la ridistribuzione dei record non causa alcuna scrittura su disco (cioè nessun buffer in overflow). I record rimasti alla fine del procedimento costituiscono il risultato.

I costi della proiezione basata su hash risultano i seguenti:
$costo\,fase\,di\,partizionamento = P(R) + P(T)$, $costo\,fase\,di\,eliminazione = P(T)$, $costo\,totale = P(R) + 2 \cdot P(T)$. Applicando all'esempio precedente, viene fuori $costo\,partizionamento = 1000 + 250 = 1250$, $costo\,eliminazione = 250$, $costo\,totale = 1250 + 250 = 1000$.

La proiezione basata su *sorting* risulta preferibile quando vi sono molti valori duplicati o se la distribuzione dei valori è molto sbilanciata (il che causerebbe una pessima performance della proiezione su hashing poichè i valori andrebbero a sbilanciarsi tutti sullo stesso buffer). Inoltre con la proiezione attraverso il sorting i dati risultano anche ordinati. Si guardi la tabella (Fig. 8) per vedere come alcuni *dbms* implementano la proiezione.

Figura 8:

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

Si nota come alcuni *dbms* di punta, quali DB2 e ORACLE, utilizzino solo il sorting per effettuare la proiezione.

Proiezione con Indice Un'ultima possibiltà è data dalla proiezione con indice: per usare un indice occorre che gli attributi che si vogliono mantenere (gli attributi di proiezione) siano tutti contenuti nella chiave su cui è fatto l'indice: in questo caso si applicano le tecniche precedenti (hashing e sorting) ai record dell'indice senza dover accedere al file dati eliminando i record duplicati (semplicemente ignorando la lista di RID associati ad ogni fhicave). Il costo è dunque dato a seconda della tecnica che viene implementata (hashing o sorting) moltiplicata per un numero di volte parti al numero di pagine usate dall'indice, quindi o pari a L, numero delle foglie, nel caso di B+tree o pari a P(H) in caso di hash (cioè al numero delle pagine dell'indice hash). Si noti che se l'indice è un B+tree e gli attributi di proiezione sono un prefisso della chiave dell'indice, i dati sono già ordinati (cioè le informazio che ci servono sono già comunque tutte presenti nell'indice e non è richiesto accedere ai dati), dunque basta scandire le foglie ed eliminare i duplicati al loro interno abbattendo il costo che risulta essere $Costo = L$ (numero delle foglie).

2.4 Join

Supponiamo di avere la seguente query

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

Quale è un modo efficiente per risolverla? Evidentemente calcolare il prodotto cartesiano e successivamente applicare la selezione è il metodo meno furbo possibile. La scelta dipende, più o meno, dagli stessi fattori visti per la selezione.

Esistono molti algoritmi di join, cioè modi diversi in cui viene implementanto l'operatore logico di join, ma, essenzialmente, appartengono tutti a 3 grandi famiglie: il confronto *nested loop* in cui si confronta "tutto su tutto", il confronto grazie all'ordinamento (sorting), e il confronto che si basa su tecniche di hash (possibile solo in caso di equi-join).

L'algoritmo più semplice per risolvere un join prevede il confronto di ogni record r di R con ogni record s di S, ma il costo ad esso associato risulterebbe $Costo = O(N(R) \cdot N(S))$, costo decisamente proibitivo. Ci poniamo dunque alcune domande: come viene influenzata l'esecuzione del join dai percorsi di accesso esistenti (cioè in presenza di indici, possiamo utilizzarli, e sotto che condizioni)? L'ordinamento dei dati influisce sulle prestazioni? C'è un modo migliore per allocare le pagine del buffer? Come si determinano gli indici utili all'esecuzione di un join? Esiste un ordinamento "furbo" per eseguire il join fra n relazioni?
Precisiamo da subito che non esiste un algoritmo "ottimo" (così come non esisteva per le precedenti operazioni). Di seguito descriveremo alcuni algoritmi che vengono utilizzati per implementare il join. (Si noti che in tutti i prossimi algoritmi utilizzeremo il segno di == per indicare il soddisfacimento del predicato di join)

2.4.1 Nested Loops Join

```
for each r in R
  for each s in S
    if (r==s) add <r,s> to the result
```

Per ogni tupla della relazione R, chiamata relazione *esterna*, si scandisce la relazione S, chiamata relazione *interna*, e per ogni tupla di S si confronta se il predicato di join viene verificato, in caso affermativo viene aggiunta la tupla al risultato. È evidente che sto confrontando ogni tupla di R con ogni tupla di S. Il costo è dato da $Costo = P(R) + N(R) \cdot P(S)$ e applicato al nostro esempio su Riviste e Sommelier abbiamo $costo = 1000 + 100000 \cdot 500 \approx 5 \cdot 10^7$, supponendo ogni lettura costi $10ms$ il join chiederebbe 140 ore (TROPPO!). Il termine P(R) nel costo è dovuto all'accesso alla relazione esterna, cioè, ovviamente, il numero delle pagine della relazione (poichè se ho più record in una pagina non occorre ricaricare la pagina), mentre il termine $N(R) \cdot P(S)$ indica che per ogni tupla di R (dunque N(R)) devo scandire tutta la relazione S (dunque P(S) per lo stesso motivo indicato su P(R)). Si noti come la relazione esterna R viene scandita una sola volta. Invertendo il ruolo di R e S otterremmo un costo inferiore $costo = 500 + 40000 \cdot 1000 \approx 4 \cdot 10^7$. Siccome il fattore dominante è il secondo ($N(R) \cdot P(S)$) conviene avere come relazione *esterna* la relazione con record "più grandi". Questa ultima affermazione è derivabile dai seguenti passaggi. $N(S) = TP(S) \cdot P(S)$ (numero di tuple di S è dato dal numero di tuple per pagina per il numero di pagine), analogamente $N(R) = TP(R) \cdot P(R)$. Se confrontiamo il fattore dominante nel costo $N(R) \cdot P(S)$ con $N(S) \cdot P(R)$ che sono i casi in cui le due relazioni si scambiano i ruoli fra relazione interna ed esterna, e sostituendo, otteniamo $TP(R) \cdot P(R) \cdot P(S)$ nel primo caso e $TP(S) \cdot P(S) \cdot P(R)$: da cui si nota che maggiore è il valore TP maggiore risulterà il costo, di conseguenza per avere un TP più piccolo occorre avere tuple "più grandi".

2.4.2 Page Nested Loops Join

```
for each page pr in R
  for each page ps in S
    for each r in pr
      for each s in ps
        if (r==s) add <r,s> to the result
```

In questo caso si scandiscono i record attraverso le pagine, cioè se nella prima pagina della relazione esterna ho 4 tuple, non devo per ogni tupla scandire sempre da capo la relazione interna, ma bensì per tutte le 4 tuple della prima pagina esterna confronto ogni pagina della relazione interna, una volta fatto, proseguo con la seconda pagina della relazione esterna e così via. Il costo di esecuzione si abbatte diventando $costo = P(R) + P(R) \cdot P(S)$ evitando dunque N(R) che era il fattore più incisivo del prodotto nel *nested loops join*. Applicato all'esempio si ottiene $costo = 1000 + 1000 \cdot 500 \approx 5 \cdot 10^5$, supponendo ancora una volta che una lettura costi $10ms$, l'esecuzione del join richiede 1.4 ore (ancora troppo!). Invertendo il ruolo di R e S si ottiene $costo = 500 + 500 \cdot 1000 \approx 5 \cdot 10^5$ dove si nota che il prodotto è rimasto invariato, di conseguenza conviene avere come relazione *esterna* la relazione conil minor numero di pagine.

2.4.3 Block Nested Loops Join

```
for each blocco in B–2 pagine in R
   for each page ps in S
      for each coppia di tuple <r,s> nel buffer
          if (r==s) add <r,s> to the result
```

Entrambi gli algoritmi visti in precedenza non sfruttano la presenza di più pagine nel buffer (ne usano solo due). Se invece abbiamo B pagine buffer le possiamo suddividere in questa maniera:

- B-2 pagine per la relazione esterna
- 1 pagina per la relazione interna
- 1 pagina per il risultato

In realtà altro non è che una generalizzazione del *Page Nested Loops Join*, in quanto ora prima di passare alla "seconda pagina" della relazione interna, ogni pagina della relazione esterna presente nei B-2 buffer analizza la prima pagina: di conseguenza la relazione *interna* S viene dunque scandita $\frac{P(R)}{B-2}$ volte. Il costo risultato è dunque dato da $costo = P(R) + \frac{P(R)}{B-2} \cdot P(S)$. Applichiamo ancora una volta all'esempio, supponendo $B = 102$: otteniamo $costo = 1000 + \frac{1000}{100} \cdot 500 = 6000$ →costo di lettura $10ms$ allora l'operazione di join richiede 1 minuto. Invertendo R e S otteniamo $costo = 500 + \frac{5000}{100} \cdot 1000 = 5500$ (55 secondi). Un modo efficiente per trovare le coppie in join è di costruire una tabella hash nel buffer, si riduce però la capacità del buffer per leggere R. Si potrebbe decidere in realtà di allocare più pagine alla relazione interna S e non solo una: controllando

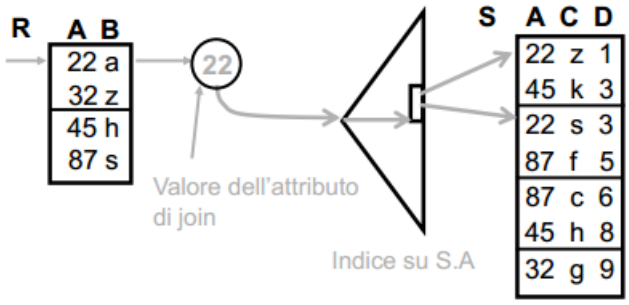
i tempi di accesso attraverso le letture sequenziali e letture random, si ottiene che il miglior costo lo si ha quando il numero di buffer allocati alla relazione interna R è uguale al numero di buffer allocati alla relazione esterna S (si noti che se per il calcolo del costo utilizziamo solo le operazioni di I/O qualsiasi distribuzione dei B buffer tra le due relazioni, il costo stesso non varia).

2.4.4 Index Nested Loops Join

```
for each r in R
    for each s in S where (r==s)
        add <r,s> to the result
```

Se la relazione *interna* S possiede un indice sull'attributo di join è possibile sfruttarlo, mentre è di minore utilità se l'indice è presente sulla relazione esterna (infatti, come vedremo, in questo caso miglioreremo l'accesso, finora costato P(R), alla relazione esterna). Il costo in questo caso diventa $costo = P(R) + N(R) \cdot (costo\,indice + costo\,dati)$. Il costo per ogni record di R dipende dal tipo di indice usato (B+tree/hash) e se l'indice risulta essere *clustered* o meno. Si veda la figura (Fig. 9) per una maggiore chiarezza sul funzionamento

Figura 9:

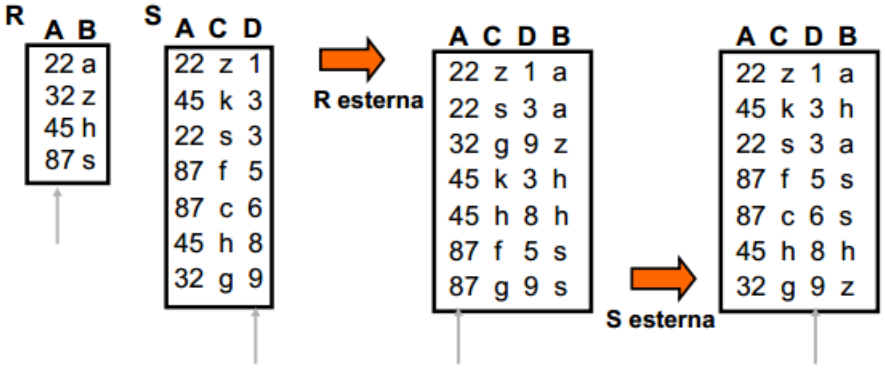


Vediamo un esempio. Supponiamo di avere un indice hash su S.sid. Siccome *sid* è attributo chiave di S c'è un solo record di S per ogni record di R→ $costo = 1000 + 100000 \cdot (1 + 1) = 200000 \backsim 33min$ (si veda che nonostante l'accesso ai dati sia veloce, il fatto di doverlo ripetere per un numero di volte pari al numero di record della relazione esterna non garantisce comunque buone performance).

Supponiamo invee di avere un indice hash su R.sid. Per ogni record di R esistono in media $\frac{100k}{40k} = 2.5$ record di S. Se l'indice è *clustered* il costo per ogni record è 1 e dunque $costo = 500 + 40000 \cdot (1 + 1) = 80000 \backsim (13min)$, altrimenti se l'indice è *unclustered* il costo per ogni record è 2.5→ $costo = 500 + 40000 \cdot (1 + 2.5) = 140000 \backsim (23min)$ (si noti che se anche l'indice risulta unclustered il costo è minore rispetto al caso prima discusso). A conseguenza di ciò conviene dunque avere come relazione *esterna* la relazione avente meno record (ricorda che il fattore dominante è causato dal numero di volte che bisogna ripetere l'accesso all'indice dovuto al numero di record).

Un'importante proprietà del *nested join loops* è che preserva l'ordine della relazione esterna: si veda la figura (Fig. 10) per una miglior comprensione

Figura 10:



Pertanto la scelta della relazione esterna può dipende anche da altri fattori, come poichè ad esempio: la proprietà, effettivamente, non incide in alcun modo sui costi della risoluzione del join, ma potrebbe essere di notevole utilità alla risoluzione "completa" della query (ad esempio se dopo il join viene richiesto un ordinamento, potrebbe convenire risolvere il join utilizzando come relazione esterna quella che causa un costo maggiore, ma ritrovandoci i valori già ordinati e dunque pronti per essere restituiti).

Si voglia notare che la logica generale dei *nested loops* (almeno per ciò che riguarda i confronti fra tuple e non fra pagine) ricade nella seguente formulazione: Costo Accesso alla relazione Esterna + Tuple Residue della relazione Esterna*Costo accesso alla relazione Interna. A seconda delle situazioni, ognuno di questi singoli valori può modificarsi. **NB** per tuple residue si intendono le tuple che hanno il diritto a partecipare all'operatore di join.

Immaginiamo di avere la seguente query

```
SELECT *
FROM R, S
WHERE R.J=S.J AND R.A=5
```

a seconda di alcuni fattori, come la presenza di indici, possiamo imbatterci in diversi scenari:

1. **Nessun indice**: nel caso non vi sia alcun indice, il costo di accesso alla relazione esterna (R) è l'accesso sequenziale, dunque risulta essere P(R). Poichè non vi è alcun indice neanche sulla relazione interna (S) allora l'accesso alla relazione interna risulterà ancora sequenziale e dunque ancora P(S). La presenza però del secondo predicato *R.A=5 locale* (cioè

applicato a una sola relazione) in AND con il predicato di join, mi diminuisce il numero di tuple residue, esso infatti risulterà non più essere N(R), ma bensì $f \cdot N(R)$ dove f è il fattore di selettività del predicato. $Costo = P(R) + f \cdot N(R) \cdot P(S)$

2. **Indice su S.J**: nuovamente il costo di accesso alla relazione esterna risulta essere P(R) poichè non vi è alcun indice, allo stesso modo il numero di tuple residue è dato da $f \cdot N(R)$. La presenza però dell'indice su S.J mi diminuisce il costo di accesso alla relazione interna portandolo ad essere (Costo di Indice + Costo ai Dati). $Costo = P(R) + f \cdot N(R) \cdot (Costo\,indice + Costo\,dati)$.

3. **Indice su R.A**: in questo caso a modificarsi è il costo di accesso alla relazione esterna, in quanto è presente un indice su R.A (che si noti essere il predicato locale): esso si modifica in $\Phi(\frac{N(R)}{NK(R)}, P(R))$ cioè il numero di pagine da accedere (in probabilità) per risolvere il mio predicato su R.A (è il modello di *cardennas*). Il costo alla relazione interna tornerà ad essere P(S) non essendoci più un indice, mentre il numero di tuple residue rimane invariato. Si noti che il numero di tuple residue $f \cdot N(R)$ può differire da Φ: si prenda ad esempio la stessa query con però in aggiunta un nuovo predicato in AND R.Q>7, il modello di cardennas ϕ rimarrà invariato, ma il numero di tuple residue risulterà minore poichè $f_1 \cdot f_2 \cdot N(R)$ dove f_1 e f_2 sono i due fattori di selettività dei predicati locali. $Costo = \Phi(\frac{N(R)}{NK(R)}, P(R)) + f \cdot N(R) \cdot P(S)$.

4. **Indice su R.A e S.J**: è semplicemente la combinazione dei casi 2 e 3. $Costo = \Phi(\frac{N(R)}{NK(R)}, P(R)) + f \cdot N(R) \cdot (Costo\,indice + Costo\,dati)$.

2.4.5 Merge-scan Join

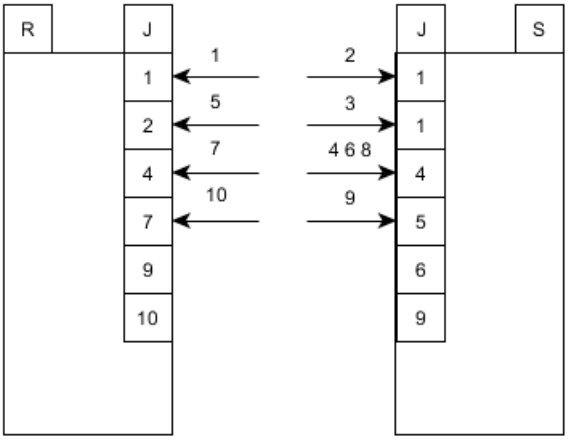
```
Se non già ordinate: sort(R), sort(S)
r = next(R)
s = next(S)
while(!EOF(R) && !EOF(S))
    if (r==s)
       add <r,s> to the result
       s = next(S)
    if (r<s) r = next(R)
    else s = next(S)
```

Richiede che entrambe le relazioni siano ordinate sugli attributi di join, in caso contrario si ordinano (in questo caso l'algoritmo si chiama *sort-merge join*). Le relazioni dunque vengono esaminate un record per volta per costruire il risultato. Salvo casi particolari ogni relazione viene scansionata una sola volta e dunque il costo diventa $costo = P(R) + P(S)$. L'algoritmo partendo dalla relazione esterna legge il primo record r_1 e lo confronta col primo record della

relazione interna s_1: se essi "matchano" li aggiungo nel risultato, se invece s_1 risulta essere maggiore di r_1 sono sicuro che non vi sarà alcun record della relazione interna che potrà "matchare" con r_1 (poichè sono ordinate), dunque vado avanti e faccio lo stesso controllo con r_2 però partendo non più da s_1 ma bensì da s_2 (poichè sono certo che per s_1 non vi era alcun record di R che poteva soddisfare il join) e così via.

Si veda la figura (Fig. 11) in cui R e S identificano le relazioni, la colonna J identifica i valori ordinati dell'attributo di join e i numeri sopra le frecce identificano i passi compiuti dall'algoritmo.

Figura 11:



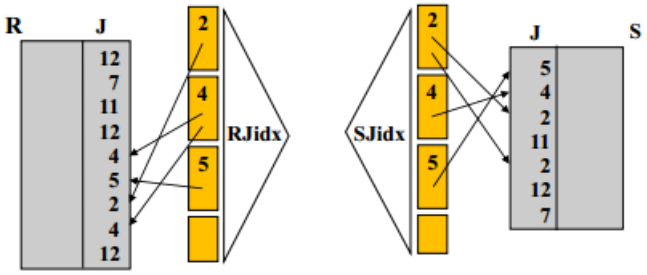
Nell'esempio $costo = 500 + 1000 = 1500\,(15sec)$. In presenza di predicati locali, il costo può diminuire in quanto le relazioni risultano essere "più piccole" poichè prima di effettuare il join vengono applicati i predicati locali che, si spera, riescano a tagliare parte dei record sul quale poi effettuare il join (cioè le tuple residue).

In caso di valori duplicati su entrambe le relazioni, ovvero quando l'associazione è molti a molti M:N, l'algoritmo si complica. Risulta essere necessario modificare l'algoritmo per fare *back-tracking* (eventualmente vanno rilette alcune pagine) poichè non è più vero che se s_1 risulta maggiore di r_1 allora s_1 non potrà più matchare con alcun valore di R (potrebbe infatti soddisfare il predicato di join con r_2) e dunque viene richiesta la presenza di un *place-holder* per sapere da che punto della relazione interna ripartire. Il caso peggiore si ha quando tutti i record di entrambe le relazioni hanno lo stesso valore degli attributi di join portando il costo a $costo = N(R) \cdot P(S)$.

In linea di principio non è necessario che i dati siano realmente ordinati, basta infatti che siano ordinati "logicamente" come può accadere su indici (Fig. 12): se sono presenti indici sugli attributi di join non è necessario effettuare il

sort, se però gli indici sono *unclustered* il costo può risultare nel caso peggiore comunque elevato $costo = N(R) + N(S)$.

Figura 12:



2.4.6 (Simple) Hash Join

```
for each r in R add r to buffer page using H(r)
for each s in S add s to buffer page using H(s)
for each partition H' of H
        read H'(R) and H'(S)
                for each matching record <r,s>
                        add <r,s> to the result
```

Il vantaggio del *merge-scan* è che viene sostanzialmente ridotto il numero di confronti tra i record delle due relazioni, ma di contro richiede che le due relazioni siano ordinate sugli attributi di join. Una buona alternativa è utilizzare l'*hash join:* essa richiede di avere B buffer disponibili e una funzione hash H a valori in [1,B-1] (serve sempre un buffer per il risultato). L'idea è dunque quella di partizionare i record di entrambe le relazioni attraverso una stessa funzione hash, confrontando in seguito solo i record appartenenti a partizioni omologhe. L'hash-join si basa proprio sull'impossibilità che due tuple appartenenti a partizioni diverse possono verificare il predicato di join, cioè se $H(C.j) \neq H(S.j) \Rightarrow C.j \neq S.j$ dove C e S sono due relazioni e j è l'attributo sul quale si effettua il join. Si noti come in una partizione vengano messe sia tuple di C sia tuple di S e si noti infine che per partizione si intende il buffer i-esimo che la funzione hash restituisce più eventuali pagine salvate su disco in caso di buffer pieno. Ovviamente dato che la funzione hash presenta collisioni, bisogna controllare in ogni partizione se i record soddisfano il predicato di join (cioè non è garantito che se due tuple sono nella stessa partizione allora sicuramente faranno parte del risultato).

La fase di matching (o chiamata anche *probing*) dell'algortimo può essere realizzata utilizzando un'altra funzione hash H_1. Il costo dell'algortimo è dato da $costo = 3 \cdot P(R) + 3 \cdot P(S)$. Nell'esempio risulta $costo = 3 \cdot 1000 + 3 \cdot 500 = 4500\,(45sec)$. Il fattore 3 è dovuto alle operazioni di lettura dei record delle

relazioni, alla scrittura nelle partizioni e al confront "match" delle tuple in ogni partizione.

2.4.7 Confronti

Facciamo un breve riepilogo, confrontando alcuni degli algoritmi finora visti.

Hash Join vs Block Nested Loops Join Nel *block nested loops join* ogni pagina di R è confrontata con tutte le pagine di S. Invece nell'*hash join* ogni pagina di R è confrontata con un'unica pagina di S.

Hash Join vs Merge-scan Join Le prestazioni di *hash join* peggiorano con distribuzioni non uniformi dei dati (diventano pessime se le relazioni sono composte da uno stesso identico valore, poichè tutte le tuple finiranno nello stesso partizionamento). In genere, però, l'hash join richiede meno utilizzo di memoria. Il vantaggio maggiore di usare il *merge-scan* è che i risultati risultano essere già ordinati.

Condizioni Generali In presenza di più predicati di join, l'*index nested loops join* può sfruttare un indice su tali predicati nella relazione *interna*. Il *merge-scan join* invece considera un ordine lessicografico sulla combinazione di attributi per decidere se $r <_J s$. Infine l'*hash join* partiziona i record sulla stessa combinazione di attributi.

Se il predicato di join non è di uguaglianza (non *equi-join*) allora l'*index nested loops join* può usare solo un B+tree. L'*hash join* e il *merge-join* non risultano applicabili, mentre tutti gli altri algoritmi non sono influenzati.

2.5 Operatori Insiemistici

Dal punto di vista dell'implementazione, l'*intersezione* e il *prodotto cartesiano* sono casi particolari di join, infatti l'intersezione altro non è che un un join su tutti gli attributi, il prodotto cartesiano invece è un join senza alcuna condizione di join.

Le tecinche per *unione* e *differenza* invece sono un attimo diverse, entrambe devono essere discusse perchè possono prevedere la presenza o meno di duplicati, ma in generale entrambe utilizzano tecniche di *partizionamento* (sorting o hashing). In generale la stima dei risultati è pari a N(R)+N(S) per l'unione e N(R) per la differenza (che è ciò che accade nel caso peggiore).

L'unione con duplicati altro non è che un operazione di *append,* in cui semplicemente viene concatenata la relazione R alla relazione S (operazione SQL: UNION ALL).

Capiamo cosa si intende per *differenza* con duplicati: supponiamo di voler fare la differenza tra R e S (R-S) dove R è formata dai seguenti valori (a,b,a,a,b,c) mentre S ha i valori (a,b,b,a,d). La differenza senza duplicati semplicemente "ignora" i duplicati e dunque R-S produce solo (c), in caso invece si vogliano

mantenere i duplicati R-S produce (a,c) poichè ora oltre a controllare se l'elemento nella relazione R è presente nella relazione S e in caso eliminarlo, bisogna contarne anche le occorrenze e in questo caso l'elemento apparirà tante volte quante occorrenze in più sono presenti su R rispetto a S (operatore SQL: EXCEPT [no duplicati] - EXCEPT ALL [con duplicati]).

Vediamo come utilizzare il *sorting* per l'unione e la differenza.
Per implementare l'unione si ordinano R ed S usando tutti gli attributi (ordinamento lessicografico) e successivamente si leggono ordinatamente R e S in parallelo eliminando i duplicati. Per implementare la differenza ancora una volta si ordinano R e S usando tutti gli attributi, successivamente si leggono ordinatamente R e S in parallelo eliminando da R i record presenti in S (in caso di presenza di duplicati si elimina un record alla volta, in caso non si vogliano i duplicati si elimina proprio il valore). $Costo = costo\,sort + P(R) + P(S)$.

La tecnica alternativa è utilizzare l'*hashing* per l'unione e la differenza.
Per implementare l'unione si partizionano i record di R ed S usando una funzione hash H su tutti gli attributi. Successivamente si legge ogni partizione di R e S in parallelo, eliminando i duplicati utilizzando una tabella hash con funzione H' su S (serve per rendere veloce il matching). Per implementare la differenza si partizionano i record di R ed S utilizzando una funzione hash H su tutti gli attributi. Successivamente si legge ogni partizione di R e S in parallelo eliminano i record di S tramite uan funzione hash H' su S. $Costo = 3 \cdot P(R) + 3 \cdot P(S)$ dove il 3 è giustificato dalla lettura, la scrittura e il confronto della funzione.

2.6 Funzioni Aggregate

Supponiamo di avere la seguente query

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

L'algoritmo utilizzato dalle funzioni aggregate è relativamente semplice: scandisce tutti i record mantenendo informazioni riassiuntive sui record già elaborati e visti (e dunque ad ogni nuovo record letto l'informazione viene aggiornata). La tabella (Tabella 2) riassume le informazioni necessarie per ogni funzione.

Tabella 2:

Funzione	**Informazione riassuntiva**
SUM	totale dei valori esaminati
AVG	totale dei valori esaminati e il numero dei valori
MIN	minimo tra i valori esaminati
MAX	massimo tra i valori esaminati
COUNT	numero dei valori esaminati

2.7 Group By

Le funzioni aggregate risultano più interessanti se applicate a dei raggruppamenti. Supponiamo di avere la seguente query

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

che raggruppa i sommelier per il valore *val* restituendo anche l'età minima di ogni gruppo creato. Sostanzialmente l'algoritmo opera per partizione: occorre cioè partizionare i record e applicare ad ogni partizione le funzioni di aggregazione.

Esistono tre possibilità per il partizionamento: l'uso del *sorting*, l'uso dell'*indice* sugli attributi di grouping e l'uso dell'*hashing*.

Si ricordi che esiste una forte relazione fra la *proiezione* e la funzione di raggruppamento GROUP BY, si esamini infatti la seguente query

```
SELECT S.val
FROM Sommelier as S
GROUP BY S.val
```

essa equivale a un SELECT DISTINCT (cioè per eliminazione dei duplicati) poichè viene richiesto di raggruppare per i valori che vogliono essere proiettati: di conseguenza gli algortimi di partizionamento utilizzati dal GROUP BY saranno molto simili a quelli utilizzati nella proiezione. Si noti infine che il costo del GROUP BY con funzioni aggregate è dato da $costo = costo\,partizionamento + costo\,funzione\,aggregata$.

2.7.1 GROUP BY con sorting

Gli algoritmi utilizzati sono analghi a quelli della proiezione (scansione relazione scrivendo in un file solo gli attributi necessari, si ordina il file e si eliminano i duplicati) più l'algortimo associato al costo della funzione aggregata: dunque il costo risulta essere $costo = costo\,sort + P(S)$. Il costo può essere diminuito se si aggregano le tuple durante la fase di *sort* mantenendo per ogni valore di raggruppamento (e dunque per ogni gruppo) le informazioni necessarie riassuntive. Lo stesso può accadere nei passi di *merge*.

2.7.2 GROUP BY con indice

Se l'indice contiene sia gli attributi di raggruppamento che quelli usati nelle funzioni allora non è neppure necessario accedere al file (poichè le informazioni si possono reperire tutte dall'indice) e dunque il costo è dato $Costo = L$ (con $L = numero\,foglie$). Se inoltre è presente una clausola HAVING formata da attributi che fanno parte del prefisso della chiave dell'indice allora il costo può diminuire ulteriormente poichè non è necessario accedere a tutte le foglie.

Se l'indice invece è un B+tree e gli attributi di raggruppamento formano un prefisso della chiave dell'indice, allora si può usare l'indice stesso per accedere ordinatamente ai dati.

In tutti gli altri casi l'indice non è utilizzabile.

2.7.3 GROUP BY con hashing

Viene creata una tabella hash (in memoria) utilizzando gli attributi di raggruppamento (appunto partizioniamo): la tabella dovrà contenere anche le informazioni necessarie per il calcolo delle funzioni aggregate. Al termine della scansione di S (e dunque del partizionamento) si usano le informazioni salvate nella tabella per calcolare i valori delle funzioni. È molto probabile che la tabella hash creata riesca a stare completamente in memoria portando il costo a $costo = P(S)$; in caso contrario occorre partizionare la tabella creata con una nuova funzione hash e ripetere il procedimento. Si noti che il calcolo della funzione aggreggata può essere ottenuto "ricorsivamente" utilizzando valori di sottogruppi, cioè se voglio sapere la somma fra tutti i valori e i valori sono stati divisi in due sottogruppi (e in ogni gruppo viene mantenuta l'informazione della somma dei valori appartenente al quel gruppo) allora ottengo la somma totale come la somma dei valori dei due sottogruppi.

2.8 Buffer

Dedichiamo una sezione all'utilizzo fatto dei *buffer*. Come abbiamo visto l'uso del buffer pool è importante per diversi algoritmi. Notiamo dapprima che diversi operatori utilizzano il buffer pool, ma è fallace pensare che sul *dbms* venga eseguita una sola query alla volta: in realtà di query in esecuzione c'è ne sono molte, di conseguenza la memoria fisica a disposizione deve essere condivisa fra esse il che porta a una rivalutazione delle prestazioni (poichè in alcuni casi esse dipendono fortemente dal numer di buffer disponibili). In caso dunque di operazioni concorrenti la dimensione del buffer pool diminuisce (si parla di *shared buffer pool*).

Come abbiamo visto nell'accesso ai dati tramite indice, la probabilità di trovare una pagina dipende dalla dimensione del buffer pool e dalla politica di rimpiazzo (si noti che se l'indice è *unclustered* il buffer pool si riempe molto velocemente poichè vengono richieste pagine quasi ad ogni accesso all'indice). Molte operazioni, in particolare il join, hanno *pattern d'accesso* predicibili: ad esempio il *nested loop* scandisce ripetutamente la relazione interna. Potendo gestire la politica di rimpiazzo, esiste una tecnica che mi rende il page fault minimo?

Notiamo che normalmente i sistemi operativi utilizzano la tecnica *LRU* (Least Recently Used) in cui viene rimpiazzata la pagina che non viene utilizzata da più tempo. Provando ad applicare LRU ad uno schema di nested loop cosa accade? Immaginiamo che la mia relazione interna sia composta da 4 valori (A,B,C,D) ma che il buffer pool possa contenerne solo 3. Al primo ciclo caricherò i primi tre valori (A,B,C) e alla richiesta di poter leggere D (page fault)

bisognerà rimpiazzare la pagina che non viene utilizzata da più tempo, cioè A. La nuova configurazione risulterà (D,B,C). Essendo ora la scansione della relazione interna terminata, bisognerà ricominciare ricontrollarla da capo (poichè è il funzionamento dei nested loop) e dunque la pagina necessaria da leggere sarà A (page fault), viene rimpiazzata B trovandoci (D,A,C); viene ora richiesta B (page fault) e viene rimpiazzata C (D,A,B)... Come si nota, in un caso del genere, ogni richiesta provoca un *page fault*: è come se sia disponibile un solo buffer anziche 3 (poichè ogni pagina deve essere ricaricata a causa del page fault).

Proviamo ad applicare come politica di rimpiazzo la *MRU* (Most Recently Used), politica che rimpiazza la pagina utilizzata più recentemente. Abbiamo ancora la nostra relazione interna composta da 4 valori (A,B,C,D) e il buffer pool di capacità massima 3. Al primo ciclo carichiamo i primi tre valori (A,B,C) con richiesta successiva di leggere D: la richiesta provoca un page fault ma questa volta ad essere rimpiazzata sarà C (l'ultima pagina usata) portando alla nuova configurazione (A,B,D). A questo punto la relazione interna deve essere scandita di nuovo, dunque bisogna rileggere A: la lettura va a segno poichè presente nel buffer pool. Segue la lettura B (a segno), segue C (page fault): viene dunque rimpiazzata B poichè è stata l'ultima pagina ad essere stata utilizzata (A,C,D); la lettura prosegue con D (a segno), segua A (a segno), segue B (page fault)... Come si nota, con il rimpiazzo MRU la situazione è nettamente migliorata, portando ad avere un solo *page fault* ogni ciclo (con LRU invece ogni lettura provocava un page fault).

Facciamo delle ultime considerazioni: per il *simple nested loops* se abbiamo B-2 pagine per la relazione interna allora come abbiamo visto MRU è la politica migliore. Viceversa, per il *block neste loops* la politica di rimpiazzo è irrilevante (viene data solo 1 pagina per la relazione interna S). Infine per l'*index nested loops* conviene ordinare i record della relazione esterna per i valori degli attributi di join sfruttando dunque la località dell'indice su S.

2.9 Operatori di modifica

Le operazioni possibili sono *update*, *delete* e *insert*. Solo l'update e il delete richiedono un percorso di accesso per recuperare i record interessati (poichè includono la clausola WHERE): inoltre se per il *delete* devo solo eliminare dati (che comporterà comunque la modifica di indici poichè bisognerà eliminare dei RID da alcune chiavi) per l'*update* (con clausola SET) dopo aver eliminato i dati bisognerà riaggiungere quelli nuovi aggiornati. Il costo è costituito da 3 componenti: il costo di accesso ai dati, il costo di modifica dei dati e il costo di modifica degli indici

2.9.1 Update

Nell'update il *costo di accesso* equivale al costo di ricerca dei record da modificare (si stima come visto in precedenza). Non è possibile utilizzare alcun indice sugli attributi che vengono modificati (viene chiamata *sindrome di Halloween*). Si veda la seguente query

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

usando un indice su *val* è possibile che alcuni record siano modificati più volte: infatti le operazione di un update sono eseguite sequenzialmente e dunque quando aggiorno un dato devo aggiornarne anche l'indice, ma potrebbe accadere (se utilizzo l'indice stesso per accedere ai dati) che il dato appena modificato venga "riacceduto" poichè ricompare nuovamente nelle condizioni che soddisfa la query (ad esempio se ho il valore 3 che devo modificare a 4, una volta effettuata la modifica nei dati, modifico anche l'indice, ma se accedessi attraverso l'indice a tutte le tuple che sono comprese tra 3 e 5 mi ricapiterebbe di riguardare il 4 appena inserito e rimodificarlo, cosa ovviamente errata). Notiamo che anche gli indici sono entità soggette a transizioni, dunque operazioni di rollback e *undo* devono poter essere eseguite (gli indici insomma sono ripristinabili).

Il *costo di modifica dei dati* dipende dall'eventuale ordinamento dei dati stessi (si noti che il costo di modifica equivale a leggere il dato, cancellarlo o modificarlo e riscriverlo col nuovo valore). Una scansione ordinata (che sia sequenziale, con indice clustered o per RID ordinate) ha un costo di $costo = \Phi(E_N, E_P)$ dove $E_N = f \cdot N$ (f è il fattore di selettività della clausola WHERE) ed $E_P = f_1 \cdot P$ risultano essere i valori attesi di tuple e pagine da modificare (con f_1 fattore di selettività sull'attributo di ordinamento, cioè il modello di cardennas non viene applicato a tutte le pagine P ma a E_P pagine poichè grazie all'ordinamento potrebbe essere necessario accedere ad un numero inferiore di pagine). Ricordiamo ancora una volta che il modello di cardennas restituisce quante pagine probabilmente bisognerà accedere per leggere i record richiesti.

In caso di scansione disordinata (con indici unclustered) il costo peggiora divenendo $costo = E_K \cdot \Phi(\frac{E_N}{E_K}, E_P)$ con $E_N = f \cdot N$, $E_K = f \cdot K$ ed $E_P = f_1 \cdot P$ (con f_1 fattore di selettività dell'attributo di ordinamento) con E_K uguale alla stima di valori (diversi) di chiave: in questo caso il prodotto per E_K è dovuto alla possibilità di dover riaccedere più volte alla stessa pagina per valori diversi di chiave.

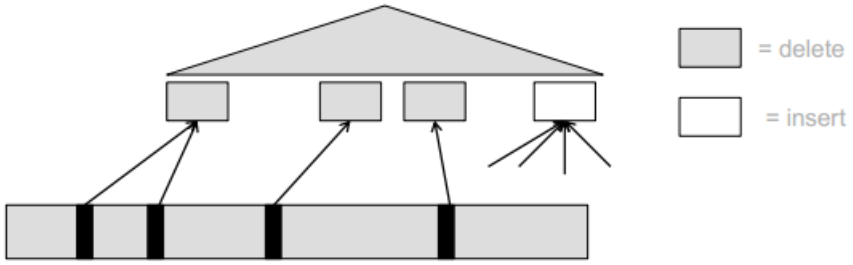
Infine manca da analizzare il *costo di modifica degli indici*. La modifica di un indice prevede il **delete** della vecchia entry e il **re-insert** delle nuove (potenzialmente è cambiato anche il RID). Esistono due casi:

- **Visita ordinata delle foglie**: possibile se sto modificando un indice clustered successivo a una modifica sequenziale dei file dati (di conseguenza avendo l'indice ordinato come il file dati accederò in maniera ordinata all'indice poichè sto leggendo allo stesso modo il file dati) oppure, solo se il predicato è di uguaglianza, se ho delle RID ordinate in un indice *unclustered*. Questa seconda opzione capita poichè anche se ho un indice *unclustered* (e quindi non ordinato come i dati) se voglio accedere a un determinato valore di chiave (ad esempio 5) e mantengo le RID di tutti i record che hanno quel valore ordinate, allora sono sicuro che visiterò ogni pagina che contiene almeno una RID una sola volta.

- **Visita disordinata delle foglie**: Accade quando si modificano gli indici *unclustered* o l'indice *clustered* con più liste di RID (cioè, in questo secondo caso, quando il predicato non è di uguaglianza).

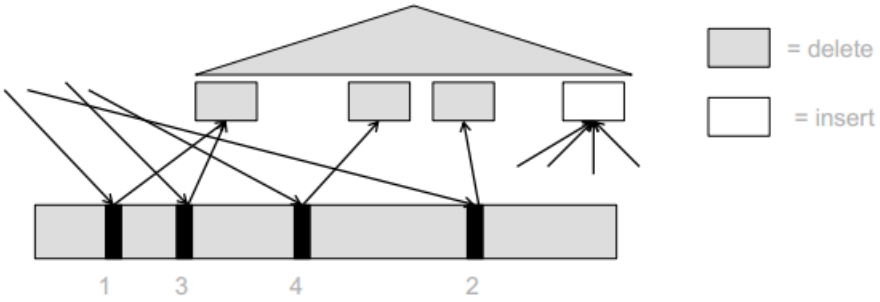
Visita ordinata delle foglie L'aggiornamento con la visita ordinata delle foglie (Fig. 13) ha i seguenti costi: $Costo\,cancellazione = 2 \cdot \Phi(E_N, f_1 \cdot L)$ (corrisponde a due volte il numero di foglie a cui probabilmente dovrò accedere per modifica le E_N entry, ancora una volta f_1 è il fattore di selettività dell'attributo di ordinamento che potrebbe permettermi di accedere solo a una parte delle foglie e non a tutte) e $costo\,inserimento = 2 \cdot E_N$ (o costo di inserimento uguale a 2 in caso di modifiche locali come *SET attr=value* poichè se ho una modifica del tipo val=5, tutte le tuple che ho modificato andranno a finire nella stessa foglia e quindi devo accedere solo a quella foglia): questo caso risulta essere pessimistico, cioè si pensa che ogni valore che deve essere modificato nell'indice si trovi in una foglia diversa. Il prodotto per due, in entrambi i casi, è dovuto alle due operazioni: la prima di lettura della foglia e la seconda di aggiornamento.

Figura 13:



Visita disordinata delle foglie Con la visita disordinata delle foglie (Fig. 14) il costo di cancellazione peggiora, mentre resta invariato il costo di inserimento: $costo\,cancellazione = 2 \cdot E_K \cdot \Phi(\frac{E_N}{E_K}, f_1 \cdot L)$ mentre il $costo\,inserimento = 2 \cdot E_N$ (o ancora 2 in caso di modifiche locali come *SET attr=value*). La cancellazione è ancora una stima pessimistica poichè suppongo che ogni record che mi arriva da cancellare (arriva in maniera disordinata) si trovi in una foglia diversa da quella caricata in memoria, e dunque il modello di *Cardennas* viene moltiplicato per il numero di valori di chiave che supponiamo avere nella nostra query di update.

Figura 14:



Di seguito mostriamo alcuni esempi.

```
UPDATE spareparts
SET shelf=5, price=price*1.15
WHERE shelf=10 AND sid=25
```

Con indici *unclustered* sia su *shelf* che su *price* non è possibile utilizzarli per l'accesso ai dati (poichè come detto non si possono usare indici sugli attributi di modifica SET). $E_N = 50$ (stima di numero di record da modificare) con RID in un'unica foglia.

Modificare l'indice su *shelf* ha $costo\,delete = 2$ e $costo\,insert = 2$ (poichè abbiamo la clausola locale WHERE shelf=10 che mi garantisce di avere le cancellazioni in una sola foglia e il resinserimento, granzie al SET shelf=5, in una sola foglia anch'essa). La modifica dell'indice su *price* ha invece $costo\,delete = 2 \cdot 50$ e $costo\,insert = 2 \cdot 50$ (poichè si suppone che ogni tupla che abbia shelf=10 and sid=25 abbia un price diverso e quindi si trovi in una foglia diversa, e dunque l'aggiornamento vada ad attualizzarsi in foglie differenti pari al numero di record che dobbiamo modificare).

```
UPDATE spareparts
SET shelf=55
WHERE pid BETWEEN 125 AND 140 AND ...
```

Con indice *clustered* su *shelf* e indice *unclustered* su *pid*. $L(shelf) = 10$ (numero di foglie dell'indice shelf), $L(pid) = 10$, $P = 100$ (numero di pagine) all'interno di un file di 400 pagine contenente anche altre relazioni. $E_N = 200$ (nuermo di tuple da modificare) e $E_K = 10$ (numero di valori di chiave). Percorsi di accesso possibili sono la scansione sequenziale (ordinata) del file o l'utilizzo dell'indice su pid (non possiamo utilizzare l'indice su shelf per il solito motivo di non poter utilizzare indici che hanno attributi di modifica).

In caso di scansione sequenziale ho i seguenti costi: $costo\,accesso = 400$ (sono obbligato a leggere tutto il file), $costo\,modifica\,dati = \Phi(200, 100) = 87$ (Car-

dennas restituisce quante pagine dovrò leggere fra le mie 100 che contengono tutte le tuple per trovare le mie 200 tuple da aggiornare) e $costo\,modifica\,indice\,shelf = 2 \cdot \Phi(200, 10) + 2 = 22$ (Cardennas restituisce quante foglie dell'indice shelf dovrò modificare fra le 10 di cui è composto l'indice).

In caso di accesso tramite indice su pid con i seguenti parametri $h = 2$ (altezza del b+tree), $L = 10$ (numero di foglie), $f_1 = 0.2 = \frac{1}{5}$ (fattore selettività su attributo di ordinamento), $E'_K = 10$ e $E'_N = 320$ ho i seguenti costi: $costo\,accesso = 2 + \frac{10}{5} (= 10 \cdot 0.2) + 10 \cdot \Phi(\frac{320}{10}, 100) = 2 + 2 + 10 \cdot 27.6 = 280$, $costo\,modifica\,dati = 10 \cdot \Phi(\frac{200}{10}, 100) = 180$ e $costo\,modifica\,indice\,shelf = 2 \cdot 10 \cdot \Phi(\frac{200}{10}, 10) + 2 = 178$

2.9.2 Delete

Il costo di ricerca dei record da cancellare si stima come visto in precedenza: per l'accesso ai dati ora è possibile utilizzare qualsiasi indice (che sia clustered o unclustered) in quanto non dovendo apportare modifiche non potrà capitare il problema della *sindrome di Halloween*. Il costo di modifica dei dati è uguale a quello visto nell'*update*. Il costo di modifica degli indici tiene conto solo del costo di cancellazione (e ovviamente non di quello di re-inserimento dato che non c'è nulla da inserire) e nel caso di accesso con indice da modificare, il costo si dimezza poichè mentre accedo ai dati attravero l'indice posso cancellare le RID dall'indice stesso così da evitare di farlo ad un passo successivo.

2.9.3 Insert

Il caso interessante di inserimento riguarda l'inserimento di risultati provenienti da un'istruzione SELECT (cioè di interesse è quando ho inserimento di tuple risultanti da un'altra query). I costi si differenziano a seconda dell'ordinamento della relazione e dei dati:

- Relazione ordinata, dati ordinati
- Relazione ordinata, dati disordinati
- Relazone disordinata, dati disordinati

Il costo di accesso ai dati è pari al costo per trovare le pagine che dovranno contenere i nuovi dati: se esiste un indice di ordinamento lo si usa per trovare la pagina in cui inserire i dati; se non esiste alcun indice ma la relazione è ordinata si opera una ricerca binaria altrimenti se non vi è alcun indice e la relazione è disordinata il costo è nullo poichè si aggiunge in coda.

Bisogna ora controllare il costo di modifica di dati e modifica degli indici. Si supponga di inserire n record con k valori di chiave e c sia la capacità di ogni pagina.

- **Relazione ordinata, dati ordinati**

- $costo\,modifica\,dati = 2 \cdot \min\{k \cdot \lceil \frac{n}{k \cdot c} \rceil, P\}$
- $costo\,modifica\,indice = 2 \cdot \lceil \Phi(k, L) \rceil$

Il costo di modifica dei dati prende o il numero di pagine che ogni record di valore k occuppa oppure il numero totale delle pagine dei dati. Mentre per il costo di modifica dell'indice il modello di Cardennas restituisce il numero di foglie a cui bisognerà (probabilmente) accedere per modificare tutti i k valori.

- **Relazione ordinata, dati disordinati**

  - $costo\,modifica\,dati = 2 \cdot n$
  - $costo\,modifica\,indice = 2 \cdot n$

  In questo caso ogni record, sia per quanto riguarda i dati sia per quanto riguarda gli indici (anche se si tratta di entry) viene visto come indipendente, quindi il costo è proporzionale al numero di record n da modificare. (Il prodotto per due è dovuto alla lettura della foglia in primo luogo più la scrittura della stessa).

- **Relazione disordinata, dati disordinati**

  - $costo\,modifica\,dati = \lceil \frac{n}{c} \rceil$
  - $costo\,modifica\,indice = 2 \cdot n$

  La modifica dell'indice ha pari costo (trattiamo sempre dati disordinati) invece la modifica dei dati altro non è che mettere tutto in coda controllando il numero di pagine $\lceil \frac{n}{c} \rceil$ che andrò ad aggiungere.

Esempio Viene data la relazione Personale(cod, nome, anno_assunzione) con $N = 50k$ (numero tuple), $P = 1000$ (numero pagine), $L(anno\_assunzione) = 100$ (numero foglie dell'indice su anno_assunzione), $K(anno\_assunzione) = 20$ (numero di valori distinti di anno_assunzione). Si voglia ora calcolare il costo di esecuzione della seguente query

```
SELECT nome
FROM Personale
WHERE anno_assunzione >2005
```

nei seguenti casi: nessun indice, indice clustered su anno_assunzione e indice unclustere su anno_assunzione.

Si noti che il fattore di selettività del predicato $anno_assunzione > 2005$è dato da $f = \frac{(2010-2005)}{20} = \frac{1}{4}$ (si è supposto che il valore massimo di anno assunzione sia 2010).

Nel caso di *nessun indice* il costo è uguale al numero di pagine $costo = P = 1000$ (poichè la ricerca risulta essere sequenziale).

Nel caso di *indice clustered* invece il costo diventa $costo = f \cdot L + f \cdot P = \frac{100}{4} + \frac{1000}{4} = 275$,

Infine nell'ultimo caso di *indice unclustered* il costo è pari a $costo = f \cdot L + \Phi(f \cdot N, P) = \frac{100}{4} + \Phi(50k \cdot \frac{1}{4}, 1000) = 25 + 1000 = 1025$.