ORGANIZZAZIONE FISICA DEI DATI ED ESECUZIONE DELLE INTERROGAZIONI

P. Rullo
Corso di laurea in Informatica
Demacs - Unical

1 Premessa

Una base di dati è generalmente molto più grande della capacità della memoria centrale, per cui il DBMS deve frequentemente accedere alla memoria secondaria durante l'esecuzione delle interrogazioni e degli aggiornamenti. Siccome il tempo medio di accesso alla memoria secondaria è di diversi ordini di grandezza maggiore del tempo medio di accesso alla memoria principale, accessi ripetuti al disco diventano il collo di bottiglia dell'intero processo. Per tale motivo, la progettazione fisica si pone l'obiettivo di individuare una configurazione di **metodi di accesso** (**indici)** tali da **minimizzare il numero di accessi alla memoria secondaria**.

2 Nozioni introduttive

2.1 Memorizzazione di una relazione su disco

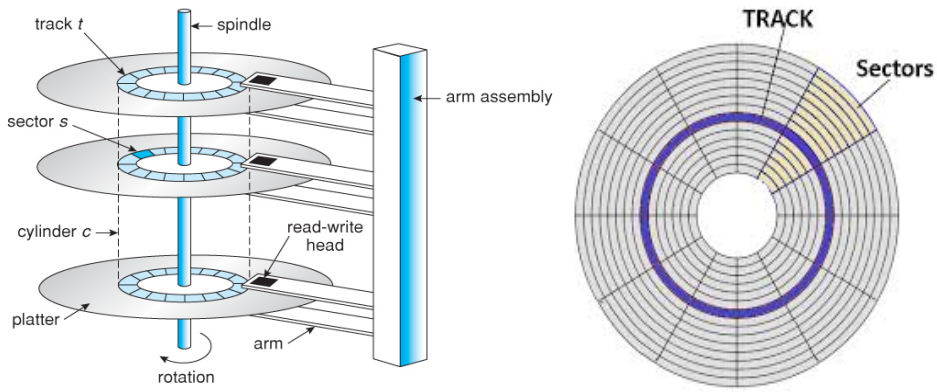
Le relazioni di una BD sono memorizzate come file nella memoria di massa (disco) dell'elaboratore. Un disco è organizzato in tracce e pagine (dette anche blocchi o settori). Una pagina è l'unità di trasferimento tra la memoria di massa e la memoria centrale, ed è una dimensione di qualche Kb. Una relazione viene memorizzata su un certo numero di pagine, in funzione del numero di tuple e della loro lunghezza. Ogni pagina contiene un numero intero di tuple.

Quando una tupla deve essere acceduta su disco, l'intera pagina che la contiene viene trasferita nel buffer di memoria centrale (vedi par. successivo). Trasferendo tutta la pagina si ha il vantaggio della *locality of reference* – se le tuple sono organizzate sulla base dell'ordine con cui vengono accedute allora è probabile che le altre tuple contenute nella pagina trasferita siano quelle a cui si richiede di accedere subito dopo.

Ci sono dischi a testina fissa (una testina per ogni traccia), e dischi a testina mobile. I parametri delle prestazioni sono:

- tempo di seek (posizionamento): 3-10ms
- tempo di latenza rotazionale: 2-5ms
- tempo di trasferimeto: dipende dalla velocità di trasferimento dal disco alla memoria centrale (qualche Giga al secondo) e dalla lunghezza di una pagina; ad esempio, se l pagina è di 4KB e la velocità di trasferimento è di un Giga/sec, allora il tempo di trasferimento è di 0,031msec.

Il tempo di accesso è pari alla somma dei primi due. Esso è dell'ordine di qualche decina di ms, migliaia di volte più elevato dei tempi di accesso alla RAM.

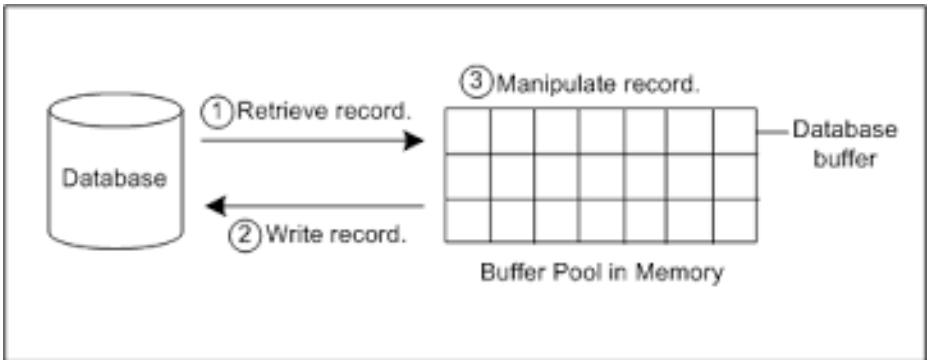


2.2 Buffer di memoria centrale

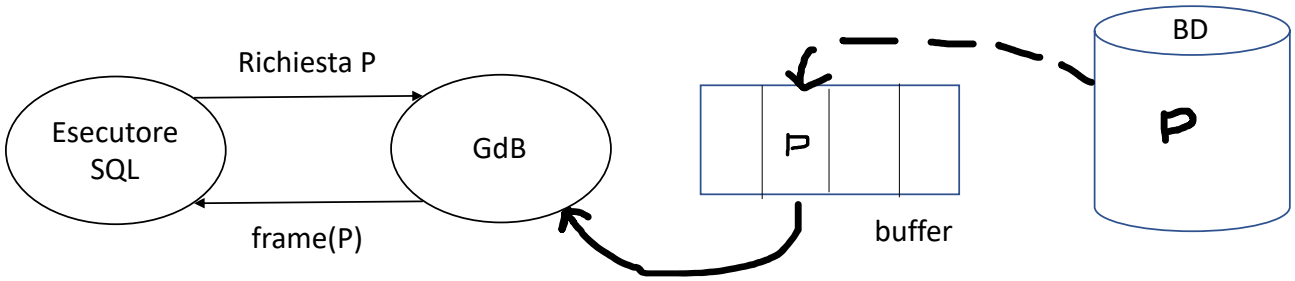
Il buffer è un'aria di memoria centrale in cui vengono trasferite le pagine della memoria secondaria.

Quando si cerca una tupla t in una pagina P di una relazione, la pagina viene preventivamente trasferita dal disco al buffer di memoria centrale – se non già presente in esso. Solo a questo punto viene effettuata la ricerca di t in P.

Il buffer è suddiviso in un certo numero di frame, ognuno della dimensione di una pagina (quindi un frame memorizza il contenuto di una pagina).



Il Gestore del buffer (GdB) si occupa del caricamento/scaricamento di pagine da e per la memoria secondaria. Durante l'esecuzione di una query, il modulo Esecutore di Query, invia al GdB le sue richieste di accesso alle pagine della BD.



2.3 Occupazione di memoria di una relazione

Data una relazione R formata da un numero di tuple *NT* di lunghezza *LT* e memorizzata su pagine di lunghezza *LP*, l'occupazione di memoria di R è (numero di pagine)

- Num tuple per pagina: **TP =** $\lfloor$ **LP/LT** $\rfloor$
- Occupazione memoria (numero di pagine): **N =** $\lceil$ **NT/TP** $\rceil$

Esempio 1: Si consideri una relazione R con NT=300.000, ognuna di lunghezza LT=100 byte, memorizzate su pagine di lunghezza LP=1024 byte. Le tuple sono indivisibili. Pertanto, il numero di tuple per pagina è 10, ed il numero di pagine necessario per memorizzare R è pari a **N=300.000/10 = 30.000**. ■

2.4 Come misurare il costo delle operazioni sulla BD

L'esecuzione di una operazione (di lettura o scrittura) sulla BD richiede il trasferimento nel buffer di memoria di un certo numero di pagine. Essendo i tempi di trasferimento da e per la memoria secondaria di diversi ordini di grandezza più elevati rispetto alle costanti di tempo della CPU ed ai tempi di accesso alla memoria centrale, il tempo complessivo richiesto per l'esecuzione di una query è dominato dal tempo richiesto dalle operazioni di accesso alla memoria secondaria.

Esempio 2. Si consideri la relazione *Cittadino* che memorizza i dati di tutti i cittadini italiani. Assumiamo che essa consista di 60 x 10^6 tuple (una per ogni cittadino) distribuite su un numero di pagine N=10^6. Il tempo richiesto per leggere i dati di tutti i cittadini coincide sostanzialmente con il tempo necessario per trasferire tutte le N pagine su cui la relazione è memorizzata. Assumendo che il tempo medio di trasferimento di una pagina in memoria sia di 10 msec, il tempo totale è nell'ordine di grandezza di 10^4 sec (circa 3 ore). ■

Il tempo di esecuzione di una operazione sulla BD è quindi proporzionale al numero di pagine trasferite, secondo una costante di proporzionalità che è il tempo medio di trasferimento di una pagina dalla memoria secondaria alla memoria centrale. Siccome quest'ultimo dipende dalle caratteristiche hw del sistema di calcolo, nonché da altri fattori, come il carico di lavoro, conviene esprimere il tempo di esecuzione di una operazione di accesso ai dati direttamente in termini di numero di accessi[1] alle pagine richiesti - che dipende solo dall'organizzazione fisica della BD, ed è invariante rispetto ad altri fattori.

3 Metodi di Acesso

Un metodo di accesso (o indice) è un meccanismo per accedere alle tuple di una relazione. Ogni relazione della BD è caratterizzata da

- un metodo di accesso primario (o struttura primaria), che definisce anche le modalità di memorizzazione delle tuple di una relazione
- un insieme di metodi di accesso secondari, usati solo l'accesso alle tuple.

Un metodo di accesso primario può essere sequenziale oppure un indice calcolato. In genere, gli indici tabellati vengono usati come metodi secondari.

3.1 Metodi di accesso primari

3.1.1 Organizzazione sequenziale

È una tecnica di **memorizzazione** che prevede che le tuple vengano memorizzate nel file

1. nell'ordine con cui sono inserite nel DB (la relazione è memorizzata come un *heap* **-** struttura disordinata o seriale), oppure
2. secondo l'ordine di uno o più attributi (struttura ordinata).

Per ottenere quindi una relazione ordinata (rispetto a un qualche attributo) è necessario scegliere una organizzazione primaria sequenziale.

3.1.2 Organizzazione ad accesso calcolato - funzione Hash

È una tecnica di **memorizzazione** e **ricerca**. In quanto tecnica di memorizzazione, costituisce un metodo di accesso primario.

Data una relazione R, memorizzata su N pagine, ed un attributo A di R, un **indice hash** h su A è una funzione

$$\mathbf{h: D_A \rightarrow I}$$

che associa ad ogni valore di A nel dominio D_A un valore nello lo *spazio degli indirizzi* I= [0, N-1]. Più precisamente:

- *Dominio D_A***:** è detto anche *spazio delle chiavi***,** cioè**,** l'insieme formato da tutti i valori che la chiave di indicizzazione può assumere. Per esempio, se la chiave è il CODICE_FISCALE, lo spazio delle chiavi è costituito da tutti i possibili codici (pari a 25^9 * 10^7) che si possono ottenere per una stringa del formato AAAAAANNNANNANNNA, in cui A è un carattere alfabetico e N rappresenta una cifra.
- *Codominio I***:** detto anche *spazio degli indirizzi*, cioè, l'insieme formato da tutti gli indirizzi di pagine disponibili per la memorizzazione delle tuple di R. Lo *spazio degli indirizzi* è quindi costituito dall'insieme {0,1, 2,…N-1}, se la relazione su cui memorizzare una data tupla consiste di N pagine.

[1] Si noti che il numero effettivo di trasferimenti è, in virtù della gestione del buffer, minore del numero di richieste di accesso alle pagine

Normalmente il numero di pagine presenti nella base di dati per memorizzare una relazione è molto minore della cardinalità dello spazio delle chiavi. Ne consegue che più tuple vengono in genere memorizzate nella stessa pagina (tale fenomeno è denominato "collisione" e le tuple che collidono vengono dette *sinonimi*). Quando una pagina non è più sufficiente per memorizzare tutti i sinonimi, si creano delle catene laterali (di *overflow*), che comportano un certo numero di accessi supplementari.

La funzione matematica che si usa per l'hash è la seguente

h(a) = a mod N

dove

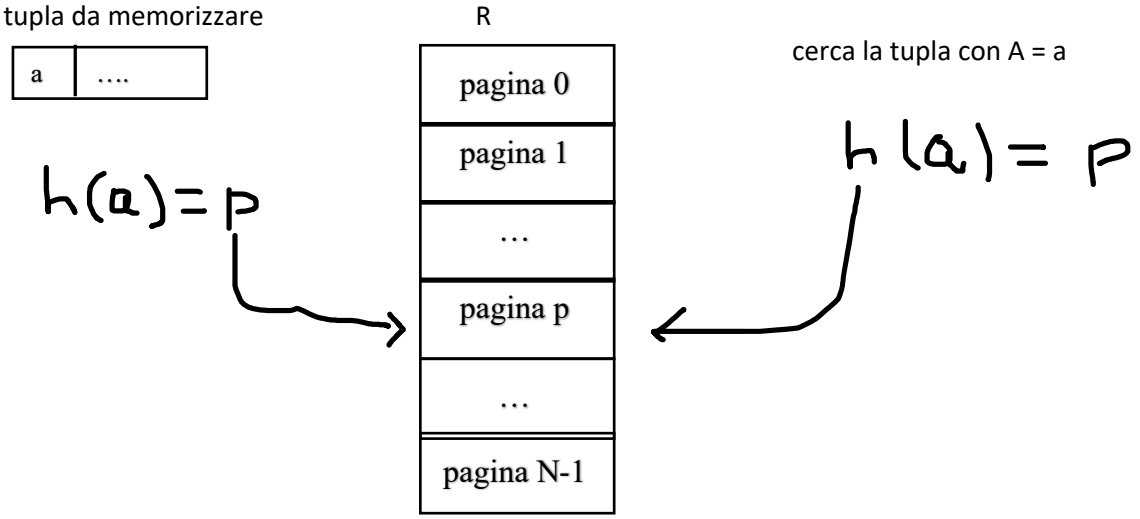
- a: codifica numerica della chiave di indicizzazione
- N: numero di pagine del file
- mod è la funzione *modulo* che restituisce il resto della divisione intera tra *a* e *N* – un valore quindi nell'intervallo [0, N-1].

Il valore h(a) è interpretato come il numero di pagina di memorizzazione/ricerca della tupla con A=a.

L'hash è detto *statico* se **N** è costante.

Un indice hash viene utilizzato sia (1) per determinare la pagina su cui una nuova tupla t deve essere memorizzata sia (2) per cercare successivamente t sulla base del valore della chiave A.

Esempio 3. Assumiamo che sull'attributo chiave A di R sia definita una funzione hash – scelta fatta dal progettista in fase di progettazione fisica.

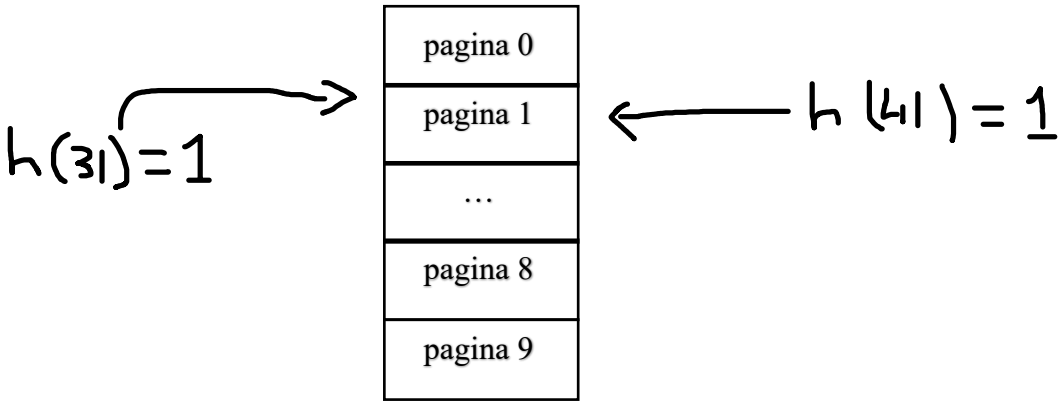


Quando la tupla t con A=a (assumiamo che A sia chiave primaria) viene inserita nella relazione R (attraverso il comando SQL INSERT(t)), al fine di individuare la pagina su cui memorizzarla, viene calcolata la funzione **h(a) = a mod N**. Il DBMS, quindi, memorizza la tupla t nella pagina con indirizzo h(a).

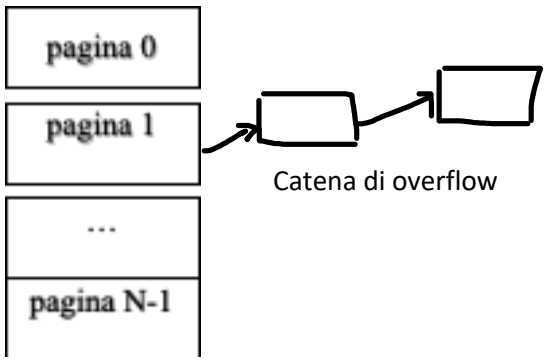
Quando successivamente si fa una ricerca $\sigma_{A=a}$ R, per trovare la tupla che soddisfa la condizione A=a, si calcola ancora la funzione **h(a)**, ottenendo in tal modo l'indirizzo della pagina in cui la tupla si trova (in quanto precedentemente memorizzata con la stessa funzione). ■

Esempio 4. Supponiamo che la relazione R sia memorizzata su N=10 pagine. Supponiamo, inoltre, che la chiave di indicizzazione A sia di tipo intero (ad es., la matricola di uno studente). Pertanto, la tupla con A=31 (lo studente con matricola 31) viene memorizzata nella pagina h(31) = 31 mod 10 = 1. Analogamente, la tupla con chiave A=41 viene memorizzata nella pagina 1. Quindi 31 e 41 sono sinonimi. ■

R



Se i sinonimi sono tanti da non poter essere memorizzati tutti nella stessa pagina, si creano delle catene di *overflow*, cioè, catene di pagine che contengono solo sinonimi (ad es., 11, 21, 51, …).



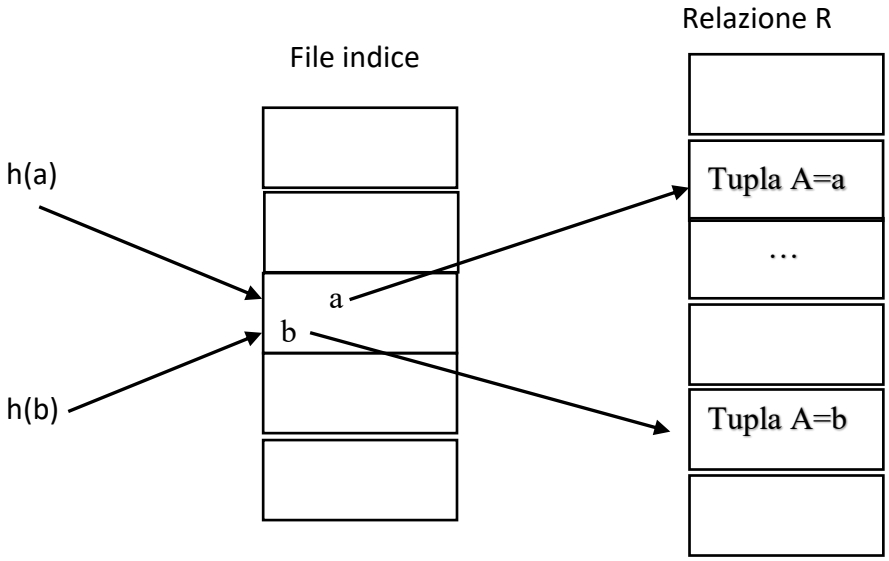
NOTA. Se l'attributo A di indicizzazione non è numerico, per calcolare h(a) si deve preliminarmente procedere ad una codifica numerica di *a*. Se *a* è la stringa $c_1c_2…c_n$ di *n* caratteri, allora la trasformazione può far uso del codice numerico ASCII $code(c_i)$ associato ad ogni singolo carattere, ed ottenere la codifica numerica di *a* attraverso, ad esempio, il prodotto $code(c_1) * code(c_2) * … * code(c_n)$. Se, ad esempio, la stringa è "*abc",* allora la sua codifica numerica è pari a 97*98*99=941094, essendo code(a)=97, code(b)=98 e code(c)=99. Pertanto, quando si deve memorizzare una tupla di R con A= "*abc"*, assumendo N=100, la pagina di memorizzazione è h(941.094) mod 100 = 94.

3.2 Metodi di accesso secondari

I metodi secondari sono strutture solo di **ricerca** di tuple (non di memorizzazione). Su una relazione R, è possibile definire più metodi secondari di accesso ai dati per velocizzare l'esecuzione delle query. Questi metodi sono in genere strutture ad albero, B-tree o B+-tree (nel seguito ci limiteremo a considerare B+-tree). Anche un insice di tipo hash può essere utilizzato come indice secondario.

3.2.1 Indice hash

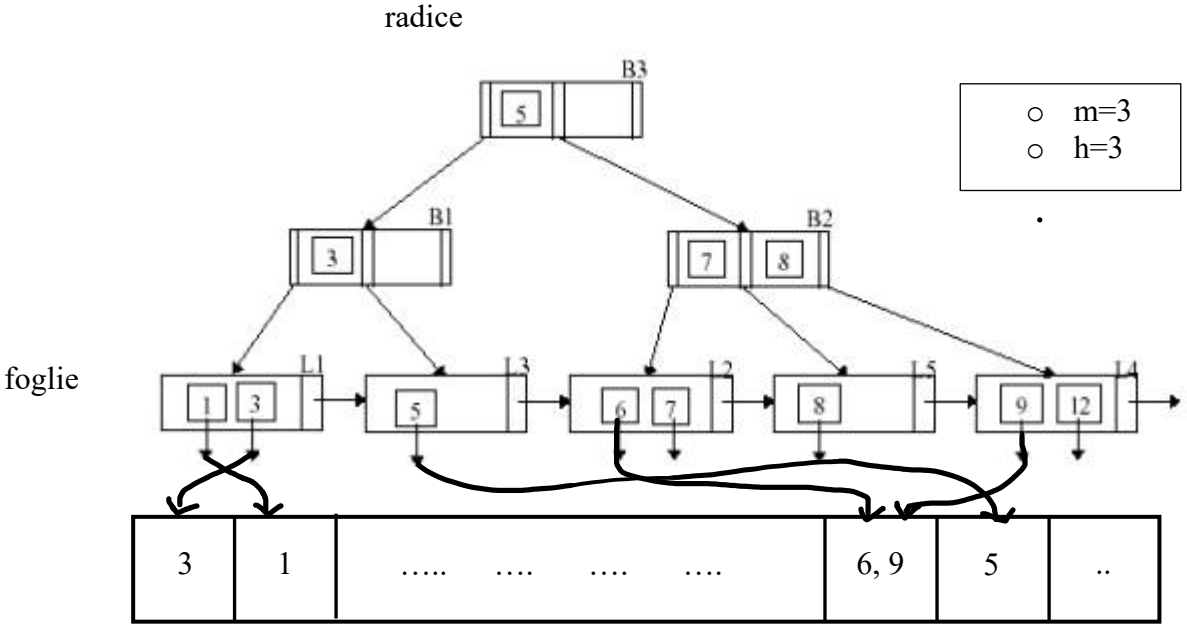
Per usare un indice hash come metodo di accesso secondario di una relazione R (cioè, non come criterio di memorizzazione, ma solo per la ricerca delle tuple di R), si procede come segue: h(a) è il numero di pagina di un *file indice* che contiene i puntatori alle pagine del file che contiene le tuple della relazione R. Il file indice può contenere catene di overflow (per eccesso di sinonimi).



3.2.2 B+-tree

Un B+-tree è un indice tabellato (a differenza dell'indice hash che è calcolato). Un indice tabellato è un insieme di coppie (k, p), dove *k* è un valore della chiave di indicizzazione e *p* è il puntatore alla pagina che contiene la tupla (o le tuple) con chiave k. Informalmente, un B+-tree è un indice a più livelli, con una struttura ad albero. Esso può essere definito su un qualsiasi attributo della relazione.

Esempio 5. La chiave primaria A di una relazione R assume i seguenti valori: 1, 3, 5, 6, 7, 8, 9, 12. Si può costruire su R un indice ad albero (B+-tree) come quello rappresentato in figura. Come si può vedere, esso ha una radice, un livello intermedio formato da 2 nodi, e 5 nodi foglie. Ad ogni valore *a* della chiave A contenuto nelle foglie, è associato un puntatore che indica il numero di pagina su cui la tupla con A=a è memorizzata. ■



Definizione di B+-tree

Un B+-tree di *ordine m*, costruito sull'attributo A, è un albero tale che

- ogni nodo N non foglia è del tipo $<p_1,V_1,p_2, V_2,\ldots, p_{k-1},V_{k-1},p_k)>$, dove
  - V_i è un valore della chiave A, p_{i-1} il puntatore al sottoalbero di sinistra e p_i puntatore al sottoalbero di destra. Il sottoalbero di sinistra contiene valori $X \leq V_i$, ed il sottoalbero di destra valori $X > Vi$ (oppure $X < V_i$ e $X \geq V_i$, rispettivamente)
  - Il numero k di nodi figli di N è tale che $\lceil \mathbf{m/2} \rceil \leq \mathbf{k} \leq \mathbf{m}$, se N non è la radice dell'albero, altrimenti $\mathbf{2} \leq \mathbf{k} \leq \mathbf{m}$
  - I valori della chiave sono ordinati, cioè, $V_1 < V_2 < \ldots < V_{k-1}$
- ogni nodo foglia è del tipo $<V_1,q_1, \ldots, V_{k-1},q_{k-1}>$, dove V_i è un valore della chiave A e q_i il puntatore alla pagina di R che contiene la tupla[2] in cui A=Vi, e k è tale che $\lfloor \mathbf{m/2} \rfloor \leq \mathbf{k} \leq \mathbf{m-1}$

Inoltre:
1. un B+-tree è bilanciato (le foglie sono tutte allo stesso livello)
2. ogni chiave appare in una foglia
3. un nodo foglia ha un puntatore alla foglia successiva ed uno a quella precedente.

L'*altezza* (o *profondità*) di un B+-tree è pari al numero di nodi che si incontrano in un qualsiasi percorso dalla radice ad una foglia.

In maniera più informale, possiamo sintetizzare così alcune proprietà di un B+-tree di ordine m:
4. Ogni nodo non foglia ha massimo m figli
5. La radice ha minimo 2 figli
6. Gli altri nodi non foglia hanno minimo $\lceil \mathbf{m/2} \rceil$ figli, cioè, sono pieni almeno a metà
7. Se un nodo non foglia ha **k** figli, ha k-1 valori della chiave
8. Un nodo foglia contiene minimo $\lfloor \mathbf{m/2} \rfloor$ **e** massimo m-1 chiavi - anche un nodo foglia è pieno almeno a metà.

Nella precedente figura, l'ordine è m=3 e la profondità h=3. Inoltre:
- ogni nodo intermedio (non radice e non foglia) ha almeno 1 valore della chiave e 2 puntatori, ed al più 2 valori della chiave e 3 puntatori.
- la radice ha 1 valore della chiave e 2 puntatori
- ogni nodo foglia ha minimo 1 e massimo 2 valori della chiave, con relativi puntatori al file.

I puntatori nelle foglie di un B+tree

Nella suddetta definizione di B+tree abbiamo visto che, ad ogni valore Vi di A presente in un nodo foglia, è associato un puntatore qi. Se A è chiave primaria, allora qi punta alla pagina che contiene l'unica tupla della relazione R per cui A=Vi. Quando A non è chiave primaria, ci sono due casi:

- la relazione è ordinata rispetto ad A. In tal caso, il puntatore qi punta alla prima tupla in cui A=a (le altre tuple seguono fisicamente la prima tupla, formando un cluster)
- la relazione non è ordinata rispetto ad A. Allora qi punta ad una pagina che contiene i puntatori a tutte le pagine che contengono le tuple in cui A=Vi.

Memorizzazione di un B+-tree

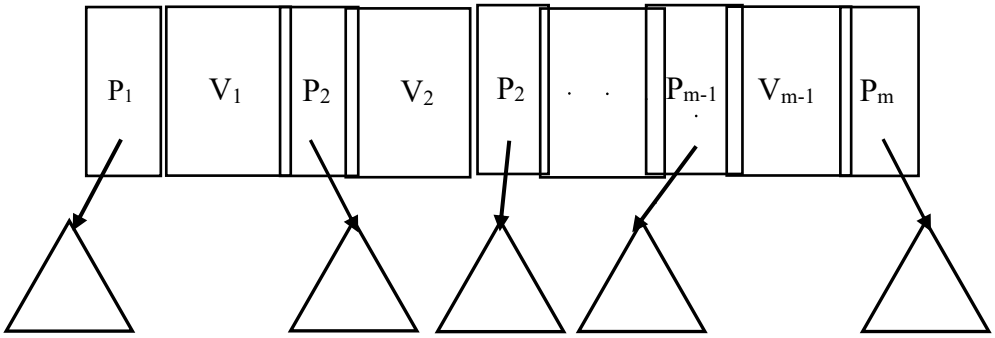
Ogni nodo di un B+-tree viene memorizzato in una pagina del disco. L'ordine dell'albero dipende quindi dalla dimensione delle pagine (in genere tra 1 e 64 K), dalle lunghezze delle chiavi e dei puntatori.

Esempio 6. Si supponga che l'attributo di indicizzazione sia lungo 9 byte e che il puntatore alle pagine sia lungo 6 byte. Per definizione di B+-tree di ordine m, ogni nodo può contenere al massimo *(m-1)* chiavi e *m* puntatori. Pertanto, nell'ipotesi che una pagina sia lunga 2K byte, vale la relazione

$$(m-1)*9 + m*6 \leq 2048$$

da cui ricaviamo m= 137 quale valore massimo di m. ■

[2] La tupla con A=Vi è unica solo se l'attributo di indicizzazione A è chiave primaria



Altezza di un B+-tree.

Altezza minima. Tutti i nodi sono pieni, cioè, contengono *m-1* chiavi. In tal caso

- $\mathbf{h_{min}\ (m, V) = \lceil \log_m V \rceil}$

dove *m* è l'ordine del B+-tree e V il numero di valori della chiave di indicizzazione.

Altezza Massima. Tutti i nodi contengono il numero minimo di chiavi $\lceil \mathbf{m/2} \rceil \mathbf{-1}$ (e $\lceil \mathbf{m/2} \rceil$ puntatori), tranne la radice che contiene una chiave e 2 puntatori. In tal caso

- $\boldsymbol{h_{max}(m,V) = \lceil\, log_{m/2} \frac{V}{2} \,\rceil + 1}$

Pertanto, per indicizzare un attributo con V valori diversi attraverso un albero di ordine *m*, la profondità **h(m,V)** dell'albero sarà $\mathbf{h_{min}\ (m,V) \leq h(m,V) \leq h_{max}\ (m,V).}$

Esempio 7. Si vuole indicizzare con un B+-tree una relazione su un attributo che può assumere V=1.000.000 valori diversi. Assumendo m=137, le altezze minima e massima del B+-tree sono

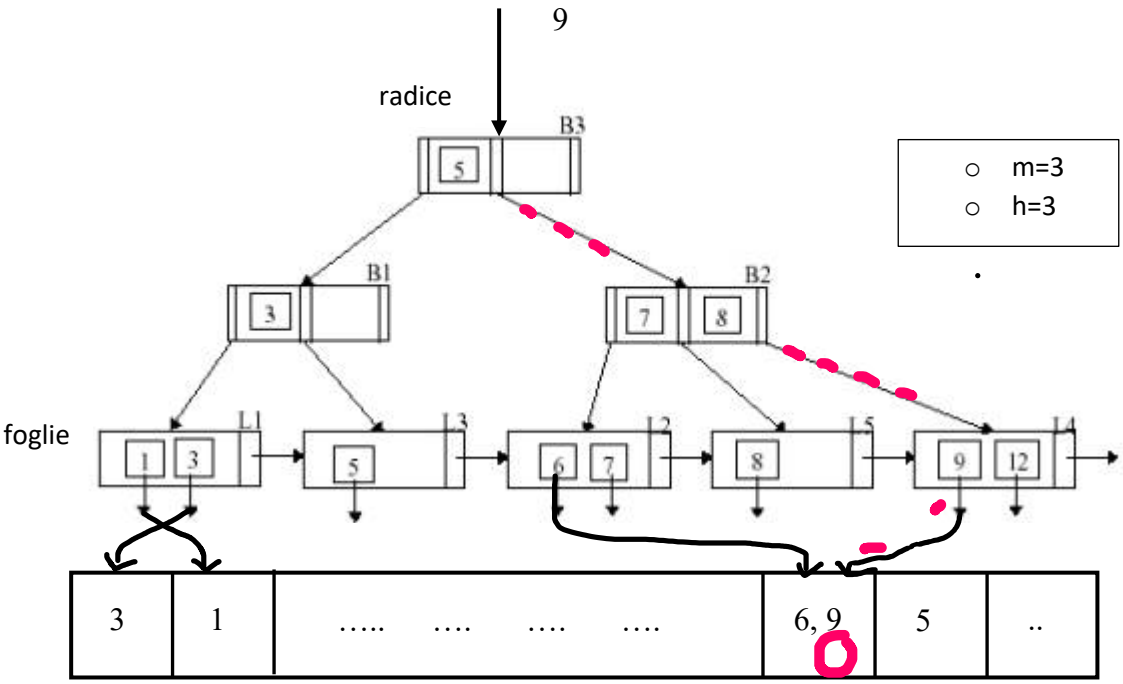
- $h_{min}\ (m, V) = \lceil \log_m V \rceil = \lceil \log_{137} 1.000.000 \rceil = 3$
- $h_{max}\ (m,V) = \lceil \log_{69} 500.000 \rceil + 1 = 5$

Ricerca tramite B+-tree

La ricerca di una tupla inizia partendo dal nodo radice, seguendo poi i puntatori verso i livelli inferiori, sulla base del valore *a* cercato, fino ad arrivare ad una foglia dove, in corrispondenza di *a,* si trova il puntatore alla pagina che contiene la tupla con A=a. Una schematizzazione del processo di ricerca è riportata di seguito.

- Sia N il nodo corrente all'interno del quale si cerca il valore *a* della chiave di ricerca; siano $V_1 ... V_k$ (con k < m, dove m è l'ordine dell'albero) i valori della chiave A contenuti in N
- Se N è un nodo foglia, allora
  - Se $a=V_i$, per qualche i, restituisci il puntatore associato a V_i – cioè, il numero della pagina in cui si trova la tupla che soddisfa la condizione A=a
  - Se *a* non è in N allora restituisci *null* – cioè, non esiste una tupla che soddisfa la condizione A=a
- Se N non è foglia, ci sono tre possibilità:
  - Se $a \leq V_1$, cerca ricorsivamente *a* nel sottoalbero sinistro di V_1
  - Se $a > V_k$, cerca ricorsivamente *a* nel sottoalbero destro di V_k
  - Se $V_i < a \leq V_{i+1}$, per qualche i tale che $1 < i < k$, allora ricorsivamente cerca *a* nel sottoalbero che si trova tra V_i e V_{i+1}.

Nella seguente figura è rappresentata la ricerca del valore 9 tramite B+-tree.

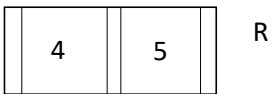


Creazione di un B+tree

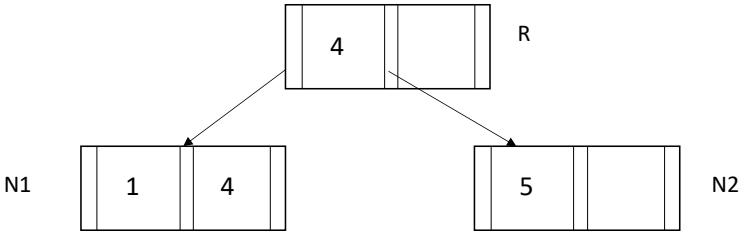
La costruzione di un B+tree si basa su un algoritmo ricorsivo che, a grandi linee, opera come segue. Quando un nuovo valore V della chiave deve essere inserito nel B+tree, viene individuata la foglia F dove V deve essere memorizzata. Se F non è piena, allora V viene effettivamente memorizzato in F. Altrimenti, F viene suddivisa in due nuovi nodi, all'interno dei quali i valori (V1, …, Vp} in F, insieme al nuovo valore V, vengono opportunamente distribuiti. Inoltre, l'elemento mediano Q di {V, V1, …, Vp} viene fatto migrare verso l'alto, cioè, verso il nodo padre di F. Questa migrazione può, a sua volta, provocare ricorsivamente la suddivisione del nodo padre di F (se tale nodo è a sua volta pieno e quindi non può accogliere Q). Illustriamo l'algoritmo con un esempio.

Esempio. Si consideri la creazione di un B+tree di ordine m=3 che memorizza la seguente sequenza di valori: 5, 4, 1, 9, 7, 12, 3, 15.

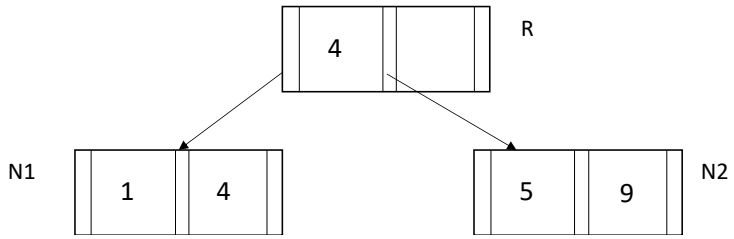
1. Si crea il nodo radice R, e le prime due chiavi (4 e 5) vengono inserite in esso, in ordine crescente



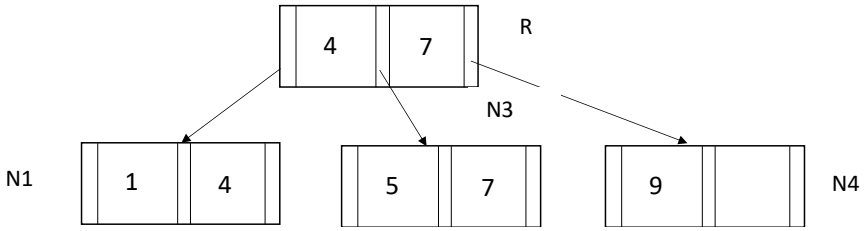
2. La terza chiave, il cui valore è 1, non trova posto nel nodo R. Siccome 4 è l'elemento mediano nell'insieme S={1, 4, 5}, allora R (che, oltre ad essere radice, è anche un nodo foglia) viene suddiviso in due nodi N1 = <1,4> e N2 = <5>, che diventano i sottoalberi, rispettivamente sinistro e destro, di una nuova radice contenente l'elemento mediano 4. In particolare, N1 contiene le chiavi in S minori o uguali a 4, ed N2 quelle maggiori di 4



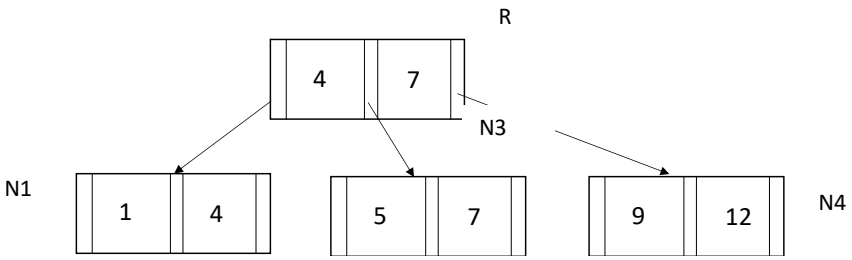
3. Successivamente, la chiave 9 trova posto nella foglia di destra N2 del suddetto albero – ciò perché 9 > 4 e, quindi, deve essere memorizzto nel sottoalbero di destra



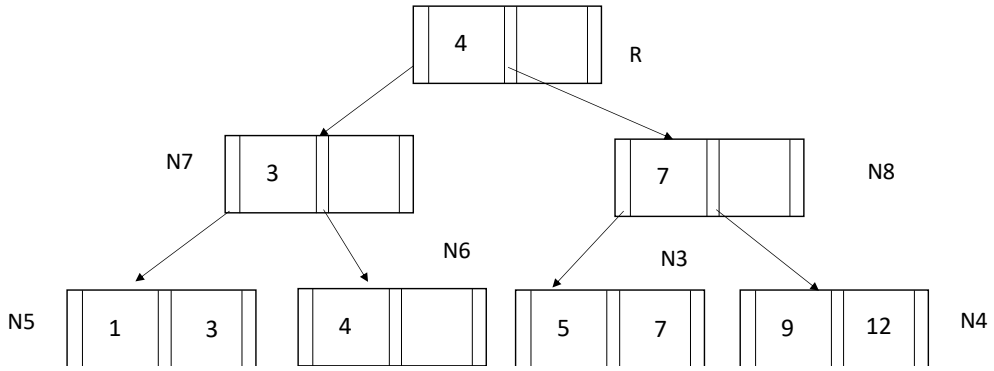
4. La quinta chiave, il cui valore è 7, dovrebbe andare anch'essa nella foglia di destra N2, che tuttavia è già completa. Siccome 7 è l'elemento mediano di S = {5, 7, 9}, allora il nodo N2 viene suddiviso in due nodi N3=<5,7> e N4=<9>, e la chiave 7 viene fatta risalire nel nodo padre di N2, cioè, la radice R dell'albero



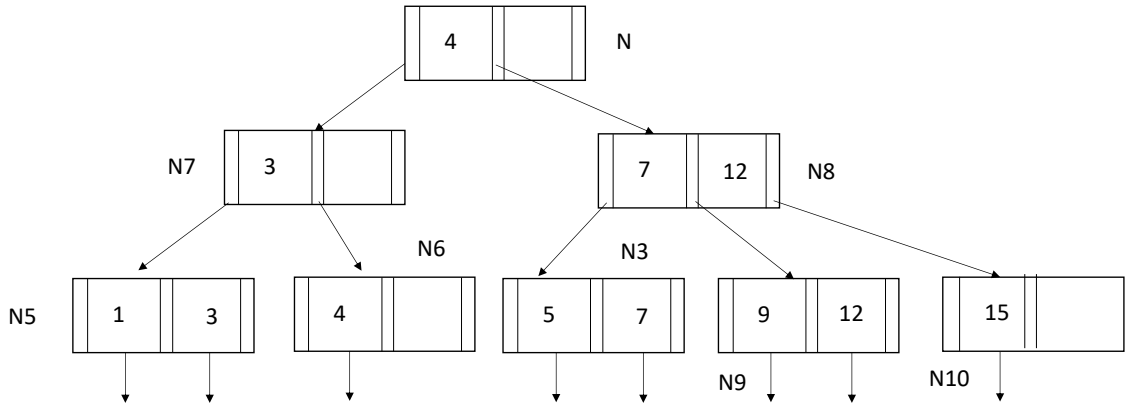
5. Successivamente, la chiave 12 viene inserita nel nodo N4 – infatti 12 > 7



6. La settima chiave, il cui valore è 3, dovrebbe andare nella foglia di sinistra N1=<1, 4> del suddetto albero, che tuttavia è già completa. Siccome 3 è l'elemento mediano di S = {1, 3, 4}, il nodo N1 viene suddiviso in due nodi N5=<1,3> e N6=<4>, e la chiave 3 viene fatta risalire nella radice R dell'albero. Questa è però anch'essa completa, per cui viene ricorsivamente prodotta un'altra suddivisione. In particolare, essendo 4 l'elemanto di separazione tra 3, 4 e 7, la radice R viene a sua volta suddivisa in due nodi (non foglia), N7=<3> e N8=<7>, ed il 4 viene fatto risalire creando una nuova radice



7. L'ultima chiave, il cui valore è 15, dovrebbe andare nella foglia di sinistra N4 del suddetto albero, che tuttavia è già completa. Siccome 12 è l'elemento mediano di S = {9, 12, 15}, allora la foglia N4 viene suddivisa in due nodi N9=<9,12> e N10=<15>, e la chiave 12 viene fatta risalire nel nodo padre N8



Si noti che: (1) l'albero risultante è bilanciato, (2) tutte le chiavi sono contenute, in ordine crescente, nelle foglie del B+tree, (3) ogni nodo foglia contiene al più m-1 puntatori alle pagine della relazione indicizzata – uno per ogni valore della chiave.

4 Implementazione e costi dei principali operatori algebrici

Ad ogni operatore algebrico viene associato un costo, che dipende dalla sua implementazione. Il costo è espresso in termini di numero di accessi alle pagine della BD richiesti – vedi par. 2.4.

Le equazioni di costo riportate nel seguito di questo paragrafo sono delle approssimazioni che, fra l'altro, non tengono conto del costo di creazione del risultato.

4.1 Operatore di Selezione

L'operatore $\sigma_F(R)$ seleziona le tuple di R che soddisfano la condizione F. Nel seguito ci limiteremo a considerare condizioni semplici del tipo **A=a**, dove A è un attributo di R.

4.1.1 Ricerca sequenziale

Per accedere alle tuple che soddisfano F è sempre possibile fare una scansione sequenziale (o lineare) delle pagine di R (ciò diventa necessario in mancanza di indici su A).

Caso 1. L'attributo A è chiave primaria – quindi esiste una unica tupla che soddista la condizione di ricerca A=a. Una schematizzazione del processo di ricerca è riportata di seguito.

Ricerca di una tupla

- accedi alla prima pagina P di R (tramite il Gestore del Buffer)
- ricerca in P (nel buffer) la tupla t che soddisfa F
- while t not trovata
  - accedi alla pagina successiva P di R (tramite il Gestore del Buffer)
  - ricerca in P (nel buffer) la tupla t che soddisfa F
- return t

E' facile intuire che, per la ricerca della tupla t, bisogna accedere mediamente a metà delle N pagine di R. Il *costo medio* della ricerca è quindi

- **Costo medio =** $\lceil \mathbf{N/2} \rceil$ richieste di accesso alle pagine di R.

Esempio 8: Si consideri la relazione *Cittadino* che memorizza tutti i cittadini italiani. Supponiamo che il numero di tuple sia pari a 60 x 10^6 distribuite su un numero di pagine N=10^6. La selezione $\sigma_{CF=xyz}$ **(***Cittadino***)** di un cittadino sulla base del codice fiscale costa in media 500.000 trasferimenti di pagine. (Se il tempo medio necessario per accedere ad una pagina e trasferirla in memoria centrale è mediamente di 10msec, allora il tempo totale è di 5.000sec - circa 1,5 ore). ■

Caso 2. L'attributo A non è chiave primaria - esiste quindi un insieme di tuple che soddisfano la condizione A=a.

Se la relazione è *ordinata* rispetto ad A (quindi la struttura primaria è di tipo sequenziale), tutte le tuple con A=a sono memorizzate in sequenza (formano un cluster). Quindi, una volta trovata la prima tupla che soddisfa la condizione di ricerca, è necessario proseguire nella scansione per reperire tutte le altre, accedendo alle M pagine successive. Si tratta quindi di accedere (1) nel caso migliore (quello in cui la prima tupla che soddisfa la condizione di ricerca si trova nella prima pagina) alle prime M pagine, e (2) nel caso peggiore (quello in cui le tuple che soddisfano la condizione di ricerca si trovano nelle ultime M pagine) a tutte le N pagine della relazione. Nel caso medio si deve quindi accedere a $\lceil \mathbf{(M+N)/2} \rceil$ pagine. Si noti che, se il numero di tuple in cui A=a è relativamente piccolo, allora $M \ll N$, ed il costo medio può essere approssimato a $\lceil \mathbf{N/2} \rceil$ richieste di accesso alle pagine di R**.**

Esempio 9. Si consideri ancora la relazione *Cittadino*, e assumiamo che sia ordinata rispetto all'attributo *provincia di residenza*. Quindi le tuple dei cittadini di una stessa provincia sono memorizzate l'una di seguito all'altra (formando un cluster). Considerato che le province sono 100, mediamente ogni cluster contiene 60x10^4 tuple (cioè, un centesimo dei cittadini), occupando un numero di pagine M pari ad 1/100 di N, cioè, M=10^4 pagine. Siccome $M \ll N$, il costo medio della ricerca $\sigma_{prov=x}$ **(Cittadino)** si può approssimare a 500.000 pagine**.** ■

Se la relazione R *non è ordinata* rispetto ad A, le tuple in cui A=a sono distribuite in maniera casuale nelle pagine della relazione. Per reperirle tutte è quindi necessario scorrere tutta la relazione, ed il costo è uguale a N trasferimenti di pagine **Costo = N.**

Esempio 10. Se la relazione Cittadino di cui all'esempio 3 non è ordinata rispetto alla provincia, il costo della ricerca $\sigma_{prov=x}$ **(Cittadino)** è pari a 1.000.000 pagine, in quanto è necessario visitare tutte le pagine della relazione.

Costi della ricerca sequenziale

Attributo di ricerca	relazione ordinata su A	relazione non ordinata su A
A chiave primaria	$\lceil N/2 \rceil$	$\lceil N/2 \rceil$
A non chiave primaria	$\lceil N/2 \rceil$ se M << N	N

4.1.2 Ricerca binaria

L'algoritmo di ricerca binaria di una tupla richiede che la relazione sia ordinata rispetto all'attributo A di ricerca. Esso funziona come la ricerca binaria di un elemento in un array. Assumiamo che A sia chiave primaria (quindi esiste una unica tupla con A=a).

Ricerca binaria

- Accedi (tramite il Gestore del Buffer) alla pagina mediana P di R. Se t è in P, allora la ricerca termina.
- Altrimenti
  - se *a* è minore di un qualsiasi valore di A presente nelle tuple in P, esegui la ricerca binaria nelle pagine che precedono P
  - se *a* è maggiore di un qualsiasi valore di A presente nelle tuple in P, esegui la ricerca binaria nelle pagine successive a P

Il costo è, nel caso peggiore, **Costo = $\lceil \log(N) \rceil$.**

Esempio 11. La ricerca binaria sulla relazione di cui all'esempio 1 costa $\log_2(30.000) = 15$ accessi a pagine (nel caso peggiore), contro i 15.000 in media necessari nel caso della ricerca lineare (vedi Esempio 2).

La ricerca binaria di un cittadino (vedi esempio 3) richiede, nel caso peggiore, 20 accessi, anziché i 500.000 mediamente richiesti da una scansione sequenziale. ■

4.1.3 Ricerca tramite accesso calcolato – hash

Assumiamo che sull'attributo di ricerca A sia definito un indice *primario* di tipo hash. In tal caso, come già accennato, la ricerca delle tuple che soddisfano la condizione A=a può essere effettuata semplicemente accedendo alla pagina della relazione di indirizzo h(a), ed eventualmente scandendo la relativa catena di overflow. Nel caso migliore[3], il costo di accesso è **Costo=1**.

Se sull'attributo di ricerca A è definito un indice *secondario* di tipo hash, allora il costo della ricerca è pari, nel caso migliore a 2 accessi alle pagine, uno al file indice ed uno alla pagina della relazione che contiene la tuple cercata. Quindi, **Costo = 2**.

4.1.4 Ricerca tramite B+Tree

Assumiamo che sull'attributo di ricerca A sia definito un B+-tree di profondità h. Nel seguito ci limiteremo a considerare **il caso in cui A è chiave primaria**.

Come già descritto, per selezionare la tupla con A=a, si deve accedere alla radice, e da questa seguire i puntatori per raggiungere il nodo foglia in cui è memorizzato il valore *a* della chiave di ricerca. Siccome ogni nodo del B+-tree è memorizzato su una pagina, per raggiungere il nodo foglia dalla radice è necessario trasferire h pagine nel buffer di memoria, dove h è la profondità dell'albero. Dal nodo foglia è poi necessario accedere, tramite il puntatore associato al valore *a*, alla pagina di R che contiene la tupla con A=a. Il costo della ricerca è pertanto **Costo = h+1.**

Esempio 12. La ricerca di una tupla in una relazione con 1.000.000 tuple, attraverso un albero di ordine m=137, costa $h_{min}+1=4$ accessi nel caso migliore, e $h_{max}+1 = 6$ nel peggiore (vedi Esempio 7). Se si facesse una ricerca binaria, nell'ipotesi che la relazione sia memorizzata su 100.000 pagine (e le tuple siano ordinate), il costo sarebbe pari a 17 pagine (caso peggiore), mentre una ricerca sequenziale costerebbe mediamente 50.000 pagine. ■

4.2 Operatore di Join

Un altro operatore fondamentale dell'algebra relazionale è il join. Ricordiamo che si tratta di un operatore commutativo, cioè, $R \bowtie S \equiv S \bowtie R$.

[3] Il caso migliore è quello in cui non è necessario scandire la catena di overflow

Esistono vari algoritmi per implementare il join, fra cui il Nested Loop, l'index Join, l'Hash Join e il SortMerge. Di seguito riportiamo una descrizione dei primi due metodi.

4.2.1 Nested Loop

Il Nested Loop è una tecnica che prevede che, per ogni pagina della relazione *interna*, vengano trasferite nel buffer tutte le pagine della relazione *esterna*.

Algoritmo NestedLoop **(**$R \bowtie S$, R è relazione interna)
for each pagina P di R {
    trasferisci P nel buffer di memoria centrale (se non è già nel buffer)
    for each pagina Q di S {
        trasferisci Q nel buffer di memoria centrale (se non è già nel buffer)
        fai il join delle tuple in P con quelle in Q}
        }

Considerato che, nel caso peggiore, bisogna trasferire in memoria ogni pagina di R e, per ognuna di esse, tuttele pagine di S, il costo del join effettuato con il nested loop (quando R è relazione interna) è il seguente

- **Costo (**$R \bowtie S$**) = NPag(R)+NPag(R)*NPag(S)**

Ovviamenet, se S è relazione interna, abbiamo

- **Costo (**$S \bowtie R$**) = NPag(S)+NPag(R)*NPag(S)**

Confrontando le due equazioni, si evince facilmente che conviene usare come relazione interna quella più piccola (cioè, memorizzata su un numero minore di pagine).

Esempio 13. La relazione R è memorizzata su 1.000 pagine, e la relazione S su 10.000 pagine. Il costo del join con il Nested Loop tra R e S è il seguente:

1. *R esterna*: Costo = NPag(R) + NPag(R)*NPag(S) = 10.001.000 pag
2. *S esterna*: Costo = NPag(S) + NPag(R)*NPag(S) = 10.010.000 pag

Si noti che il costo è leggermente inferiore nel caso di R relazione interna (essendo R più piccola di S). Vale la pena osservare che, assumendo un tempo medio di trasferimento di una pagina pari a 10msec, l'esecuzione del join tra R e S con il Nested Loop richiede all'incirca 100.000 sec (27 ore). ■

4.2.2 Index Join

Si tratta di una versione efficiente del Nested Loop, possibile nel caso in cui sugli attributi di join sono definiti degli indici (hash o B+-tree). Supponiamo che il join sia $R \bowtie S$ e che valgano le seguenti assunzioni:

- la condizione di join è R.A = S.B
- l'attributo B è chiave primaria di S
- su B è definita una funzione hash oppure un B+-tree.

Questa è una situazione molto frequente, in quanto molti dei join richiesti nella pratica operativa sono equijoin fatti su attributi chiave (chiave primaria in join con una chiave secondaria).

In tal caso, il join può essere eseguito secondo il seguente schema:

- esegui una scansione sequenziale di R (relazione interna)
- per ogni tupla t di R, usa il valore t.A come input all'indice su B per trovare la tupla t' di B tale che t'.B=t.A.

Algoritmo Index join
for each page P of R
    trasferisci P nel buffer di memoria centrale
    for each tuple t in P
        Q = index_lookup(t.A) //Q è la pagina che contiene la tupla t'$\in$S s.t. t'.B=t.A
        trasferisci Q nel buffer di memoria centrale
        cerca t' in Q e fai il join tra t e t' //crea una nuova tupla

end

Il precedente algoritmo richiede la scansione di ogni pagina P di R (*for each page P of R*) e, per ogni tupla in P (*for each tuple t in P*), un accesso ad una pagina di S (effettuato tramite indice). Pertanto, il costo del join è il seguente

Costo= NPag(R) + Ntuple(R)*CostoAccessoTupla(S)

dove

- NPag(R): numero pagine di R
- NTuple(R): numero tuple di R
- CostoAccessoTupla(S): costo di accesso ad una tupla di S tramite indice

Caso 1 - l'indice è di tipo hash. CostoAccessoTupla(S) = 1 (caso ideale), e pertanto

Costo= NPag(R) + Ntuple(R)

Caso 2 - l'indice è un B+-tree. CostoAccessoTupla(S) = h+1, e pertanto

Costo = NPag(R) + NT(R) * (h+1)

Esempio 14. La relazione R ha 10.000 tuple memorizzate su 1000 pagine. La relazione S è memorizzata su 10.000 pagine ed ha un indice sulla chiave primaria B. I costi del join tra R e S sono i seguenti:

- Index join(hash): Costo= = NPag(R) + Ntuple(R) = 1.000 + 10.000 = 11.000 pag
- Index join (B+-tree): Costo = NPag(R) + NT(R)*(h+1) = 1.000 + 4*10.000 = 41.000 pag

È utile confrontare I suddetti costi con quelli del Nested Loop dell'Esercizo 13 (il costo si riduce di 3 ordine di grandezza). ■

4.2.3 Hash join

Si tratta di un algoritmo di join che usa l'hash per incidiczzare le tuple in una tabella in memoria centrale. Siano R e S le due relazioni operando, e sia |R| < |S|. Indichiamo con T una tabella hash in memoria centrale.

Il primo passo nell'algoritmo *hashJoin* è il riempimento di T usando le tuple della relazione più piccola R. Una volta costruita la tabella T, si scandisce l'altra relazione S per trovare corrispondenze nelle sue righe attraverso la tabella hash.

Algoritmo hashJoin

for each tupla t in R

- sia t.a il valore dell'attributo di join; aggiungi t alla tabella T: T[hash(a)] = t
- if tabella T è piena
  - Scandisci S e, per ogni tuple t' in S con attributo di join t'.b, cerca in T la tupla t=T[hash(t'.b)]; se t.a=t'.b, fai il join di t e t'
  - Svuota T

Esegui scansione finale di S e fai il join con le tuple in T

end

Il costo dell'hashJoin dipende dal rapporto K tra la dimensione di R e quella della tabella T (K indica quante volte la tabella T viene caricata con le tuple di R e, di conseguenza, quante volte la relazione S viene scandita):

Costo = NP(R) + K · NP(S).

Esempio 15. La relazione R ha 10.000 tuple memorizzate su 1.000 pagine, mentre la relazione S è memorizzata su 10.000 pagine. Assumiamo che la tabella hash T in memoria centrale possa contenere 4.000 tuple di R. Pertanto, K=3 (infatti, T viene caricata 2 volte con 4.000 tuple di R, ed 1 volta con le rimanenti 2.000 tuple). Il costo dell'hashJoin è il seguente:

costo = 1.000 + 3 x 10.000 = 31.000 pag.

4.2.4 Confronto delle prestazioni

Nei precedenti paragrafi abbiamo analizzato vari metodi di join e formulato, per ognuno di essi, abbiamo proposto una equazione di costo. Nella seguente tabella confrontiamo le prestazioni dei diversi metodi usando i risultati degli esempi 13-15.

$R \bowtie S$; NP(R)=1.000, NT(R)=10.000, NP(S) = 10.000			1 trasferimento = 10 ms
Metodo	**Alternativa**	**Costo (#trasferimenti di pagine)**	**Stima tempi**
Nested Loop	R interna	10.001.000	97.000 sec (circa 27 ore)
Nested Loop	S interna	10.010.000	97.000 sec (circa 27 ore)
Index join	hash	11.000	108 sec (circa 0,03 ore)
Index join	B+tree	41.000	396 sec (circa 0,11 ore)
HashJoin		31.000	324 sec (circa 0,09 ore)

5 Cenni sulla progettazione fisica di una BD

Scopo principale della progettazione fisica di una BD è l'individuazione dei metodi di accesso alle relazioni al fine di rendere efficienti le operazioni più costose e frequenti. Più precisamente, per ogni relazione, si tratta di scegliere

1. la struttura primaria – sequenziale o hash
2. un insieme di indici secondari.

Nell'effettuare tali scelte, il progettista deve tenere conto dell'insieme delle query e delle relative frequenze. Infatti, la scelta di un indice può avvantaggiare determinate operazioni, a scapito di altre. Si tratta quindi di trovare il giusto compromesso tra vantaggi e svantaggi. Un problema di ottimizzazione estremamente difficile dal punto di vista computazionale.

A titolo esemplificativo, si consideri un contesto applicativo nel quale vi sono prevalentemente due tipi di accesso alle tuple di R:

- accessi diretti del tipo "seleziona la tupla in cui A=a", dove A è chiave primaria
- accessi per intervallo del tipo "seleziona tutte le tuple per cui vale la condizione $b1 \leq B \leq b2$".

Nel primo caso, è evidente che la definizione di una funzione hash su A (assumiamo come metodo di accesso primario) risulterebbe molto vantaggiosa, in quanto consentirebbe il reperimento di una specifica tupla di R con un singolo accesso (nel caso ideale).

Nel secondo caso, sarebbe invece conveniente ordinare le tuple rispetto all'attributo B (con una struttura primaria di tipo sequenziale). Ciò perchè le selezioni per intervallo $\sigma_{b1 \leq B \leq b2}(R)$ traggono vantaggio dall'ordinamento, in quanto tutte le tuple che soddisfano la condizione di selezione si trovano disposte in maniera contigua (formano un *cluster*), limitando in tal modo lo spazio di ricerca.

Esempio 16. Si consideri la relazione Studente, e si assuma che i principali tipi di interrogazione siano del tipo:

a) seleziona lo studente cha ha una data matricola
b) seleziona gli studenti con età compresa in un certo intervallo.

Caso 1 - Sull'attributo matricola viene definito un metodo primario di tipo hash. Pertanto, trovare un singolo studente attraverso la sua matricola costa un singolo accesso in memoria secondaria (nel caso migliore), mentre per trovare tutti gli studenti con età compresa, ad esempio, tra 18 e 23 anni, è richiesta una scansione di tutte le pagine della relazione.

Caso 2 - Viene definito un metodo di accesso primario sequenziale (che garantisce l'ordinamento su età). In tal caso, la ricerca di un singolo studente richiede una scansione sequenziale, mentre risulta avvantaggiata la seleziona degli studenti che si trovano in un certo intervallo di età (per via dell'ordinamento).

Per il progettista, si tratta quindi di decidere se conviene favorire gli accessi diretti (tramite funzione hash), o gli accessi per intervallo. Ciò dipende evidentemente dalla frequenza con cui i diversi tipi di query si presentano. Si possono tuttavia trovare soluzioni di compromesso. Ad esempio, usando una struttura primaria di tipo sequenziale, per favorire le query intervallo, e definendo un indice secondario, hash o B+-tree, su matricola, per favorire gli accessi diretti. ■

Il problema della progettazione diventa ancora più complesso nel momento in cui si considerano anche le operazioni di modifica della BD. Infatti, mantenere un file ordinato, così come mantenere un indice tabellato (come il B+-tree) aggiornato, a seguito di operazioni di inserimento, cancellazione o modifica, ha un costo tutt'altro che trascurabile. Pertanto, quando si opera la scelta, ad esempio, di un indice tabellato, bisogna tenere conto non solo dei vantaggi per le operazioni di ricerca, ma anche dei costi degli aggiornamenti. Anche in questo caso, cioè, si tratta di fare un bilanco di tipo costi/benefici.

In conclusione, nella definizione degli indici è necessario trovare il giusto punto di equilibrio tra effetti positivi ed effetti negativi. Per tale ragione, il problema della scelta degli indici è un task complesso (complessità esponenziale), che non può essere risolto in maniera esatta attraverso l'analisi di tutte le combinazioni possibili di indici.

6 ESECUZIONE DI INTERROGAZIONI

Il **Gestore delle Interrogazioni** (GdI) ha il compito di compilare ed eseguire in maniera efficiente le interrogazioni SQL. L'esecuzione efficiente richiede la preparazione di un piano di esecuzione 'ottimo' basato sulla organizzazione fisica della BD.

I passi canonici per l'esecuzione di una query SQL Q sono i seguenti:

- Compilazione - generazione del piano di esecuzione
  - Traduzione di Q in una forma algebrica
  - Ottimizzazione logica
  - Ottimizzazione fisica e generazione di un piano di esecuzione
- Esecuzione bottom up del piano di esecuzione.

Ottimizzazione logica. L'ottimizzazione logica consiste nella trasformazione di una espressione algebrica in una equivalente, ma più 'conveniente'. Ad esempio, facendo l'anticipazione della selezione rispetto al join, come nella seguente espressione

$$\sigma_F\,(R \bowtie_{A=B} S) \equiv (\sigma_F\,R) \bowtie_{A=B} S$$

dove F è un predicato che contiene solo attributi della relazione R. Il vantaggio della espressione a destra, rispetto a quella originaria, è che, riducendo la lunghezza di R attraverso la selezione, il costo del join viene ridotto.

Indichiamo con C1 e C2 i costi delle espressioni, rispettivamente, a sinistra e a destra nella suddetta equivalenza:

$$C1 = costo\big(\sigma_F\,(R \bowtie_{A=B} S)\big) = C'_{\bowtie} + C'_{\sigma}$$
$$C2 = costo((\sigma_F\,R) \bowtie_{A=B} S) = C''_{\sigma} + C''_{\bowtie}$$

dove

- $C'_{\bowtie}$ è il costo del calcolo della relazione $R' = R \bowtie_{A=B} S$
- C'_{σ} è il costo della selezione $\sigma_F\,(R')$
- C''_{σ} è il costo del calcolo della relazione $R'' = \sigma_F\,(R)$
- $C''_{\bowtie}$ è il costo del join $R'' \bowtie_{A=B} S$

Siccome $|R''| \leq |R|$, allora $C''_{\bowtie} \leq C'_{\bowtie}$. Pertanto, affinchè C2>C1, è necessario che $C''_{\sigma} > C'_{\sigma}$. Ovviamente, il costo delle due selezioni dipende dalle dimensioni delle relazioni R e R', nonchè dai relativi metodi di accesso.

Per semplicità, assumiamo che le selezioni vengano fatte per scansione sequenziale. Quindi

- $C'_{\sigma} = NP(R')$
- $C''_{\sigma} = NP(R)$

Assumiamo, inoltre, che il join venga eseguito attraverso il Nested Loop. Ne consegue che

$$C1 = C'_{\sigma} + C'_{\bowtie} = NP(R') + NP(R) + NP(R) \cdot NP(S)$$

$$C2 = C''_{\sigma} + C''_{\bowtie} = NP(R) + NP(R'') + NP(R'') \cdot NP(S)$$

É evidente che se |R'|>|R| (cioè, dimensione della relazione risultato del join è maggiore della dimensione di R) allora, essendo |R''|≤ |R|, la condizione C2 < C1 è sempre verificata. In particolare, se $R \bowtie_{A=B} S$ coincide con il prodotto cartesiano tra R e S, allora NP(R') = NP(R)*NP(S) e, quindi, C2 < C1. Altrimenti, la condizione C2 < C1 dipende dai valori di NP(R') e NP(R''). Tuttavia, si può notare che, sia in C1 che in C2, i termini dominanti sono i prodotti $NP(R) \cdot NP(S)$ e $NP(R'') \cdot NP(S)$, per cui, affinchè C2 possa essere maggiore di C1, è necessario che NP(R'') sia molto simile a NP(R), cioè, la selezione abbia un effetto molto limitato.

Esempio 17. A titolo di esempio, consideriamo i seguenti due casi, in cui la relazione R è memorizzata su 10.000 pagine, e la relazione S su 1.000 pagine.

Caso 1. Assumiamo NP(R'') = 0.6*NP(R)=6.000 (cioè, viene selezionato il 60% delle tuple di R). Pertanto,

$$C1 = NP(R') + 1.000 + 1.000 * 10.000 = NP(R') + 10.001.000$$

$$C2 = 10.000 + 6.000 + 6.000 * 1.000 = 6.016.000$$

É facile vedere che, con questi numeri, C1 è sempre maggiore di C2, a prescindere dalla dimensione della relazione R' (risultato del join).

Caso 2. Assumiamo NP(R'') = 0.995*NP(R)=9995 (solo 5 tuple vengono selezionate su 10.000). Pertanto,

$$C1 = NP(R') + 10.000 + 1.000 * 10.000 = NP(R') + 10.010.000$$

$$C2 = 10.000 + 9.995 + 9.995 * 1.000 = 10.014.995$$

In tal caso, C2 > C1 se NP(R') < 4.995 pagine. ■

L'ottimizzazione logica si basa su un insieme di regole di equivalenza dell'algebra relaziona, tra cui:

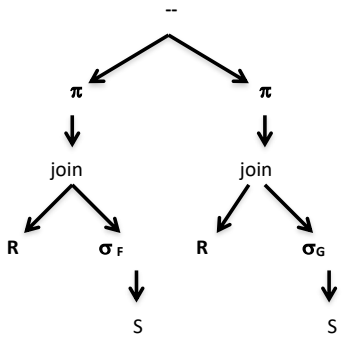
1. Inglobamento della selezione nel prodotto cartesiano $\sigma_F(R \bowtie S) \equiv (R \bowtie_F S)$
2. Distributività della selezione rispetto all'unione e alla differenza
$$\sigma_F(R \cup S) \equiv \sigma_F R \cup \sigma_F S$$
$$\sigma_F(R - S) \equiv \sigma_F R - \sigma_F S$$
3. Corrispondenza tra operatori insiemistici e operatori logici, ad esempio
$$\sigma_{F1} R \cup \sigma_{F2} R \equiv \sigma_{F1 \vee F2} R$$
$$\sigma_{F1} R \cap \sigma_{F2} R \equiv \sigma_{F1 \wedge F2} R$$

Ottimizzazione fisica. Riceve in ingresso una espressione algebrica (ottimizzata), e produce un piano ottimo di esecuzione. Questa fase dipende dai metodi di accesso (primari e secondari) definiti sulle relazioni della BD nella fase di progettazione fisica. In generale, quando esistono più alternative per eseguire una data operazione algebrica, il sistema deve scegliere, sulla base di una stima dei costi, quella più efficiente.

Ad esempio, si consideri la seguente espressione

$$\pi_A(R \bowtie \sigma_F S) - \pi_A(R \bowtie \sigma_G S)$$

di cui è riportata una rappresentazione ad albero nella figura seguente. Come si può notare, i nodi foglia rappresentano relazioni, mentre i nodi interni rappresentano operatori algebrici.



Nella fase di ottimizzazione fisica, il GdI associa ad ogni nodo operatore (nodo interno) dell'albero una scelta implementativa. A tal fine, esso valuta le diverse ipotesi di implementazione (che dipendono dall'organizzazione fisica dei dati) utilizzando le equazioni di costo. Quindi procede alla scelta dell'ipotesi migliore (quella a costo più basso, cioé, che richiede il minore numero di trasferimenti di pagine). Con riferimento al suddetto esempio, l'ottimizzatore deve decidere come eseguire le due operazioni di selezione, i due join, così come l'operatore di differenza. Le scelte effettuate saranno quelle cui corrispondono i costi minimi stimati.

Il risultato di questa fase è il cosiddetto *piano ottimo di esecuzione*, cioè, un albero come quello sopra riportato in cui ad ogni operatore (nodo interno) è associato un metodo di implementazione (il 'migliore' secondo la stima dei costi). In sintesi, per ogni nodo operatore:

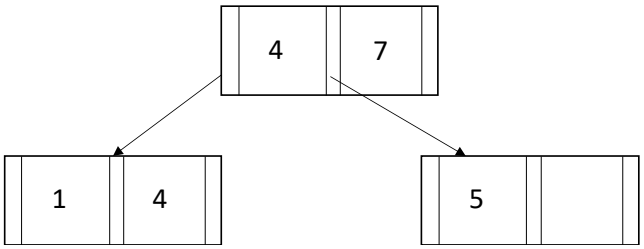
- Si considerano le diverse alternative implementative
- Per ognuna di esse si valuta il costo e si sceglie quella a costo minimo

Inoltre, si stima la lunghezza della relazione risultato, da utilizzare come input al passo successivo.

La valutazione della query avviene secondo un approccio bottom up (dalle foglie alla radice).

7 Esercizi

1. Si consideri una relazione R con 100.000 tuple, ognuna di lunghezza 200 byte, memorizzate su pagine di lunghezza 1024 byte. Le tuple sono indivisibili, cioè, ogni pagina può contenerne un numero intero. Assumendo che il tempo medio di trasferimento nel buffer di memoria di una pagina sia di 20ms, qual è il tempo medio richiesto per trovare una tupla di R attraverso una scansione sequenziale?
2. La relazione Studente ha un metodo di accesso primario sequenziale di tipo cronologico – quindi l'i-esimo studente iscritto si trova nella i-esima posizione. Il sistema genera automaticamente i valori di un attributo A che indicano la posizione di ogni studente. Assumendo che ogni tupla sia lunga 100 byte, e che le pagine siano di 4Kb, qual è il costo della ricerca $\sigma_{A=500}$ del 500° studente iscritto?
3. Una relazione R ha un indice primario hash sulla chiave K, che è un attributo di tipo stringa di 3 caratteri. Calcolare su quale pagina viene memorizzata la tuple con K = 'oca' assumendo che (1) la relazione sia memorizzata su N=100 pagine, e (2) il valore numerico di K viene calcolato come prodotto dei codici ASCII delle lettere che formano la stringa.
4. Dire qual è la profondità minima di un B+-tree di ordine m=60 che indicizza una relazione su un attributo che ammette 1.000.000 valori diversi.
5. Dire quante pagine sono necessarie per memorizzare un B+tree di ordine m=100 e profondità h=4, nell'ipotesi che tutti i nodi siano pieni.
6. Dire come viene trasformato il seguente B+tree quando viene inserita la chiave 3



7. Sulla chiave primaria K della relazione R è definito un B+-tree. La lunghezza di K è pari a 50 byte, il numero di tuple di R è pari a 300.000, la lunghezza di una tupla è pari a 400 byte, e la lunghezza delle pagine di memorizzazione di R e dei nodi del B+-tree è pari a 2048 byte. (1) Calcolare la profondità minima del B+tree nell'ipotesi che la lunghezza dei puntatori sia pari a 10 byte; (2) Calcolare il costo della ricerca di una tupla di R sulla base del valore di K nei seguenti 2 casi: (a) utilizzo del B+tree di cui sopra; (b) scansione sequenziale (costo medio).
8. Siano date le relazioni R(K1, A1, … An) e S(K2, B1, … Bm), la prima con un numero di tuple NT(R) = 200.000, memorizzate su 5.000 pagine, e la seconda con NT(S)=500.000 memorizzate su 10.000 pagine. Si assuma che su K2 sia definito un B+-tree di profondità 3. Calcolare il costo delle seguenti operazioni algebriche:
   (1) $\sigma_{K2=a}(S)$, (2) $\sigma_{K1=a}(R)$, e (3) *R join S,* con la condizione An=K2.
9. Sull'attributo A della relazione R è definito un B+tree. Dire qual è la migliore strategia per l'implementazione delle seguenti selezioni: $\sigma_{A=a \text{ AND } B=b}(R)$ e $\sigma_{A=a \text{ OR } B=b}(R)$.
10. La relazione R(K,A,B,C) ha NT(R)=10.000 tuple memorizzate su NP(R) = 2.000 pagine, mentre la relazione S(H,E,F) consiste di NT(S) = 5.000 tuple memorizzate su NP(S) = 1.000 pagine. Si vuole calcolare il costo del join $R \bowtie_{H=C} S$ nei seguenti casi:
    a. Nested Loop con R relazione interna
    b. Index join quando sulla chiave primaria H di S è definito (i) un B+-tree di profondità h=3, e (ii) una funzione hash.
11. Siano date le relazioni R(K, A1, … An) e S(K, A1, … An), la prima con un numero di tuple NT(R) = 100.000, memorizzate su 5000 pagine. Calcolare i costi delle operazioni algebriche *R union S* e *R minus S*, nell'ipotesi che sull'attributo K di S sia definita una funzione hash e che l'algoritmo sfrutti tale funzione. Si assuma che tutti i sinonimi (secondo la funzione hash) siano memorizzati su una stessa pagina (cioè, non ci sono catene di overflow).
12. Una relazione R ha un indice B+tree definito su un attributo A non chiave primaria. La relazione R ha NT=10.000 tuple e non è ordinata rispetto ad A. Dire qual è il costo di ricerca, tramite il B+tree, delle tuple che soddisfano la condizione A=a, nell'ipotesi che A assuma 100 valori diversi, tutti ugualmente probabili.
13. Una relazione R ha un B+tree come indice (secondario) sulla chiave primaria K. Dire come il B+tree può essere utilizzato per fare ricerche del tipo: seleziona le tuple tali che $k1 \leq K \leq k2$.