Capitolo 11 (Progettazione fisica)

Progettazione fisica

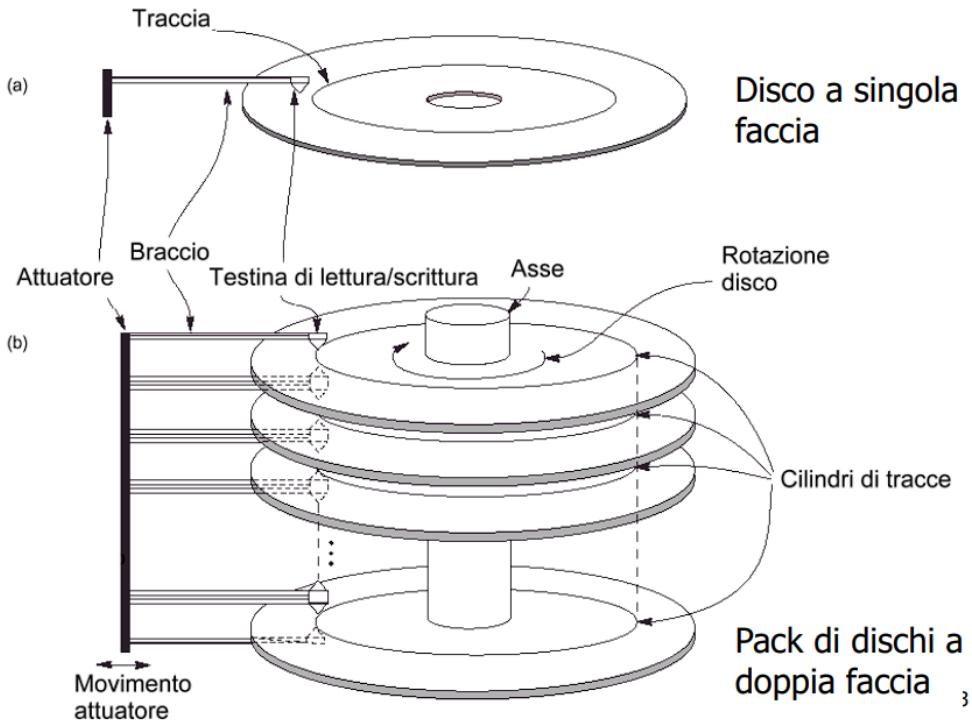
Memoria principale e secondaria

I programmi possono fare riferimento solo a dati in memoria principale

Le basi di dati debbono essere (sostanzialmente) in memoria secondaria per due motivi:

- Dimensioni
- Persistenza

I dati in memoria possono essere utilizzati solo se prima trasferiti in memoria principale



I dischi a singola faccia memorizzano i dati su una singola facciata, mentre quelli a doppia faccia su due

I dispositivi di memoria secondaria sono organizzati in **blocchi** di lunghezza (di solito) **fissa** (di ordine di grandezza: alcuni KB)

Le uniche operazioni sui dispositivi sono la lettura e la scrittura di una **pagina**, cioè dei dati di un blocco (ovvero una stringa di byte)

Per comodità consideriamo **blocco** e **pagina** sinonimi

Accesso a memoria secondaria (dati dal sito della Seagate e da Wikipedia, 2009):

- Tempo di **posizionamento della testina (seek time):** in media 2-15ms (a seconda del tipo di disco)
- Tempo di **latenza (rotational delay):** 2-6ms (conseguenza della velocità di rotazione, 4-15K giri al minuto)
- Tempo di **trasferimento** di un blocco: frazioni di ms (conseguenza della velocità di trasferimento, 100-300MB al secondo)

Tempo totale: non meno di qualche ms in media

Commenti:

- Il costo di accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria centrale
- Nelle applicazioni **I/O bound** (cioè con molti accessi a memoria secondaria e relativamente poche operazioni) il costo dipende esclusivamente dal numero di accessi a memoria secondaria
- Accessi a blocchi “vicini” costano meno **(contiguità)**

Strutture

Strutture primarie per l’organizzazione di file:

- Sequenziali (primarie)
  - Caratterizzate da una disposizione consecutiva delle tuple in memoria di massa (che può derivare dall’ordine di inserimento o da un’altra regola opportuna)
- Calcolate (”Hash”) (primarie e talvolta secondarie)

    - Collocano le tuple in posizioni determinate sulla base del risultato dell’esecuzione di un algoritmo
- Ad albero (di solito indici) (secondarie e primarie)

Strutture sequenziali

Esiste un ordinamento fra le ennuple, che può essere rilevante ai fini della gestione

- **Seriale:** ordinamento fisico ma non logico
- **Ordinata:** ordinamento fisico coerente con quello di un campo (attributo)

Struttura sequenziale seriale

Chiamata anche:

- Entry sequenced
- File heap
- File disordinato

In queste strutture un file è costituito da vari blocchi di memoria logicamente consecutivi, e le tuple vengono inserite nei blocchi rispettando una sequenza:

- In una organizzazione **disordinata (seriale)**, la sequenza delle tuple è indotta dal loro ordine di immissione

Il costo di una ricerca è **lineare** nel numero di blocchi del file e avviene tramite una scansione sequenziale del file

Molto diffusa nelle basi di dati relazionali, associata a indici secondari

Per le eliminazioni si procede di solito “marcando” il record come cancellato, ma senza alcuna riorganizzazione locale

Gli inserimento vengono effettuati:

- In coda (con riorganizzazioni periodiche)
- Al posto di record cancellati

La gestione è molto semplice ma spesso inefficiente

Struttura sequenziale ordinata

Nell'organizzazione **sequenziale ordinata** (detta anche clustered), la sequenza delle tuple dipende dal valore assunto in ciascuna tupla da un campo del file

Ogni ennupla ha una posizione basata sul valore di un campo "chiave" (o "pseudo-chiave") che permette di organizzare un ordinamento fisico (ad esempio ordine per matricola)

Storicamente, le strutture sequenziali ordinate venivano usate da processi batch su dispositivi sequenziali (nastri). I dati venivano riposti in un file principale, le modifiche venivano raccolte in file differenziali. Tali file venivano periodicamente fusi (merge). Questa pratica non è più attuale

Favoriscono le cosiddette **selezioni su intervallo (range query)**, come ad esempio la ricerca di tuple con un cognome che inizia con una certa sequenza di lettere

In modo analogo vengono favorite le operazioni aggregate, se l'ordinamento è sui campi di aggregazione (cioè quelli coinvolti dalla clausola `GROUP BY`)

Principali problemi: inserimenti o modifiche aumentano lo spazio fisico utilizzato - richiedono organizzazioni periodiche per mantenere l'ordinamento

Opzioni per evitare riordinamenti globali:

- Lasciare un certo numero di slot liberi durante il primo caricamento. Successivamente si applicano operazioni di riordinamento "locale"

- Integrare i file ordinati sequenzialmente con un overflow file, dove le nuove ennuple vengono inserite in blocchi collegati, formando una catena di overflow

Permettono ricerche binarie, ma solo fino ad un certo punto (difficoltà nel trovare la “metà” del file)

Nelle basi di dati relazionali si utilizzano quasi solo in combinazione con indici (file ISAM o file ordinati con indice primario)

Tavola hash

Obiettivo: accesso diretto ad un insieme di record sulla base del valore di un campo (detto **chiave**, che per semplicità supponiamo identificante, ma non è necessario)

Se i possibili valori della chiave sono in numero paragonabile al numero di record (e corrispondono ad un “tipo indice”) allora usiamo un array; ad esempio:

- Università con 1000 studenti e numeri di matricola compresi fra 1 e 1000 o poco più e file con tutti gli studenti

Se i possibili valori della chiave sono molti di più di quelli effettivamente utilizzati, non possiamo usare l’array (spreco); ad esempio:

- 40 studenti e numero di matricola di 6 cifre (un milione di possibili chiavi)

Volendo continuare ad usare qualcosa di simile ad un array, ma senza sprecare spazio, possiamo pensare di trasformare i valori della chiave in possibili indici di un array

Funzione hash:

- Associa ad ogni valore della chiave un “indirizzo” (indice), in uno spazio di dimensione paragonabile (leggermente superiore) rispetto a quello strettamente necessario
- Poiché il numero di possibili chiavi è molto maggiore del numero di possibili indirizzi (”lo spazio delle chiavi è più grande dello spazio degli indirizzi”), la funzione non può essere iniettiva e quindi esiste la possibilità di **collisioni** (chiavi diverse che corrispondono allo stesso indirizzo)
- Le buone funzioni hash distribuiscono in modo casuale e uniforme, riducendo la probabilità di collisione (che si riduce aumentando lo spazio ridondante), è comunque spesso necessaria una tecnica per la gestione delle collisioni

Esempio:

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - 20 record senza collisione
- numero medio di accessi: (28x1+8x2+3x3+1x4)/40 = 1,425

M	M mod 50	M	M mod 50
60600	0	200268	18
66301	1	205619	19
205751	1	210522	22
205802	2	205724	24
200902	2	205977	27
116202	2	205478	28
200604	4	200430	30
66005	5	210533	33
116455	5	205887	37
200205	5	200138	38
201159	9	102338	38
205610	10	102690	40
201260	10	115541	41
102360	10	206092	42
205460	10	205693	43
205912	12	205845	45
205762	12	200296	46
200464	14	205796	46
205617	17	200498	48
205667	17	206049	49

Per 28 posizioni è necessario 1 solo accesso

Per 8 posizioni due accessi (sono i secondi)

Per 3 posizioni tre accessi (i terzi)

Per 1 posizioni quattro accessi (il quarto)

0	60600
1	66301
2	205802
4	200604
5	66005
9	201159
10	205610
12	205912
14	200464
17	205617
18	200268
19	205619

22	210522
24	205724
27	205977
28	205478
30	200430
33	210533
37	205887
38	102338

40	102690
41	115541
42	206092
43	205693
45	205845
46	205796
48	200498
49	206049

1	205751
2	200902
2	116202
5	116455
5	200205
10	201260
10	102360
10	205460
12	205762
17	205667
38	200138
46	200296

Collisioni

Varie tecniche per la gestione:

- Posizioni successive disponibili
- Tabella di overflow (gestita in forma collegata)
- Funzioni hash “alternative”

Nota:

- Le collisioni ci sono (quasi) sempre

- Le collisioni multiple hanno probabilità che decresce al crescere della molteplicità
- La molteplicità media delle collisioni è molto bassa

File hash

La memoria secondaria permette di memorizzare più record in un solo blocco

I blocchi vengono spesso riempiti solo in parte e si usa il termine **fattore di riempimento** per indicare la frazione dello spazio fisico disponibile mediamente utilizzata, nell'esempio dei 40 studenti, tale fattore è pari a 0.8

Inoltre, se nella tabella sono definiti:

- n, la dimensione del blocco
- m, la dimensione media del record
- $F = n/m$ si definisce **fattore di blocco** ed è il numero di record contenuti in un blocco
  - Quindi se T è il numero di tuple previsto per il file, F il fattore di blocco e f il fattore di riempimento, il file può prevedere un numero di blocchi B pari al numero intero immediatamente superiore a $T/(f \times F)$

La funzione hash, applicata alla chiave, restituisce un numero compreso tra 0 e $B - 1$

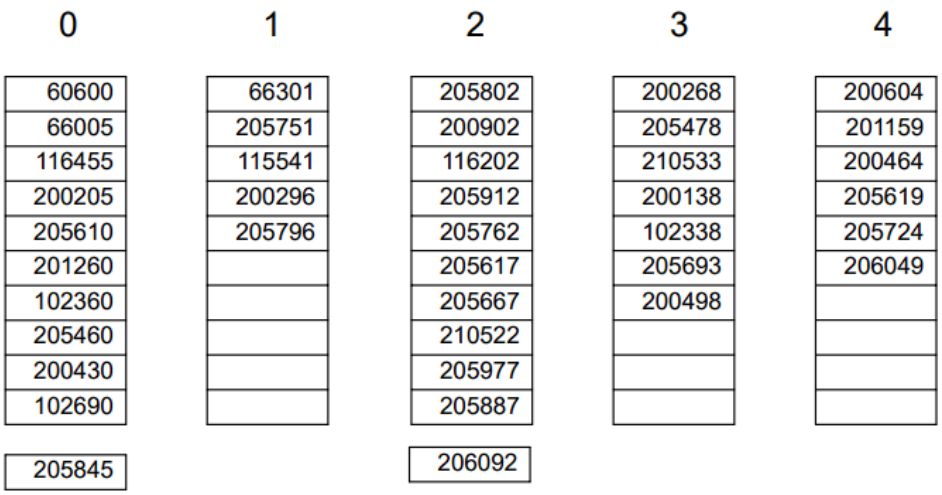
Se lo spazio relativo al blocco viene esaurito, viene allocato un ulteriore blocco, collegato al precedente, e il record viene disposto all'interno di esso, la stessa tecnica è usata anche in caso di completamento di spazio disponibile nel nuovo blocco (si formano così **catene di overflow**)

Queste strutture permettono un accesso diretto molto efficiente

La tecnica si basa su quella utilizzata per le tavole hash in memoria centrale

Nell'esempio, con fattore di blocco pari a 10, possiamo usare $mod(5)$ invece di $mod(50)$

Un file hash:



Nell'esempio:

- 40 record
- Tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - Numero medio di accessi: 1,425

- File hash con fattore di blocco 10; 5 blocchi con 10 posizioni ciascuno:
  - Due soli record in overflow
  - Numero medio di accessi: (42/40) = 1,05

Stima collisioni

Lunghezza media delle catene di overflow, al variare di:

- Numero di record esistenti: T
- Numero di blocchi: B
- Fattore di blocco: F
- Coefficiente di riempimento: $T/(F \times B)$

T/(FxB)	1	2	3	5	10	(F)
.5	0.5	0.177	0.087	0.031	0.005	
.6	0.75	0.293	0.158	0.066	0.015	
.7	1.167	0.494	0.286	0.136	0.042	
.8	2.0	0.903	0.554	0.289	0.110	
.9	4.495	2.146	1.377	0.777	0.345	

Le collisioni (overflow) sono di solito gestite con blocchi collegati

Risulta essere l’organizzazione più efficiente per l’accesso diretto basato su valori della chiave con condizioni di uguaglianza (accesso puntuale):

- Costo medio di poco superiore all’unità (il caso peggiore è molto costoso ma talmente improbabile da poter essere ignorato)

Non è efficiente per ricerche basate su intervalli (né per ricerche basate su altri attributi)

I file hash “degenerano” se si riduce lo spazio sovrabbondante: funzionano solo con file la cui dimensione non varia molto nel tempo

Indici di file

Indice:

- Struttura ausiliaria per l’accesso (efficiente) ai record di un file sulla base dei valori di un campo (o di una “concatenazione di campi”) detto chiave (o, meglio, pseudo-chiave, perché non è necessariamente identificante)

Idea fondamentale:

- L’indice analitico di un libro: lista di coppie (termine, pagina), ordinata alfabeticamente sui termini, posta in fondo al libro e separabile da esso

In prima approssimazione, dato un file f con un campo chiave k (anche non identificante), un **indice secondario** è un altro file, in cui ciascun record è logicamente composto di due campi:

- Uno contenete il valore della chiave k del file f
- L’altro contenete l’indirizzo o gli indirizzi fisici dei record di f che hanno quel valore di chiave
- L’indice secondario è ordinato in base al valore della chiave e consente quindi una rapida ricerca in base a tale valore

Quando invece l’indice *contiene al suo interno i dati*, oppure è *realizzato su un file ordinato sullo stesso campo su cui è definito l’indice stesso*, esso viene definito **indice primario**

- Si noti che un file la cui organizzazione primaria è hash oppure sequenziale non può avere un indice primario, poiché il criterio di memorizzazione dei dati è stabilito dalla funzione di hash o dalla particolare sequenza

Un indice primario, grazie all'ordinamento, può essere realizzato "puntando" a un solo record per ciascun blocco del file, in quanto gli altri record sono posti in modo *adiacente a quello puntato e possono essere poi reperiti tramite una breve scansione lineare*

- In questo caso l'indice si dice **sparso**, in quanto esistono valori della chiave che non sono presenti nell'indice
- Viceversa, un indice secondario deve per forza contenere riferimenti a tutti i valori della chiave, visto che record con valori consecutivi della chiave possono trovarsi in blocchi ben diversi; gli indici di questo tipo si dicono **densi**
  - Di conseguenza un indice secondario può essere solo **denso**

Tipi di indice (slide)

Indice primario:

- Su un campo sul cui ordinamento è basata la memorizzazione (detti anche indici di cluster, anche se talvolta si chiamano primari quelli su una chiave identificante e di cluster quelli su una pseudo-chiave non identificante)

Indice secondario:

- Su un campo con ordinamento diverso da quello di memorizzazione

Esempio, sempre rispetto ad un libro:

- Indice generale

- Indice analitico

I benefici legati alla presenza di indici secondari sono molto più sensibili

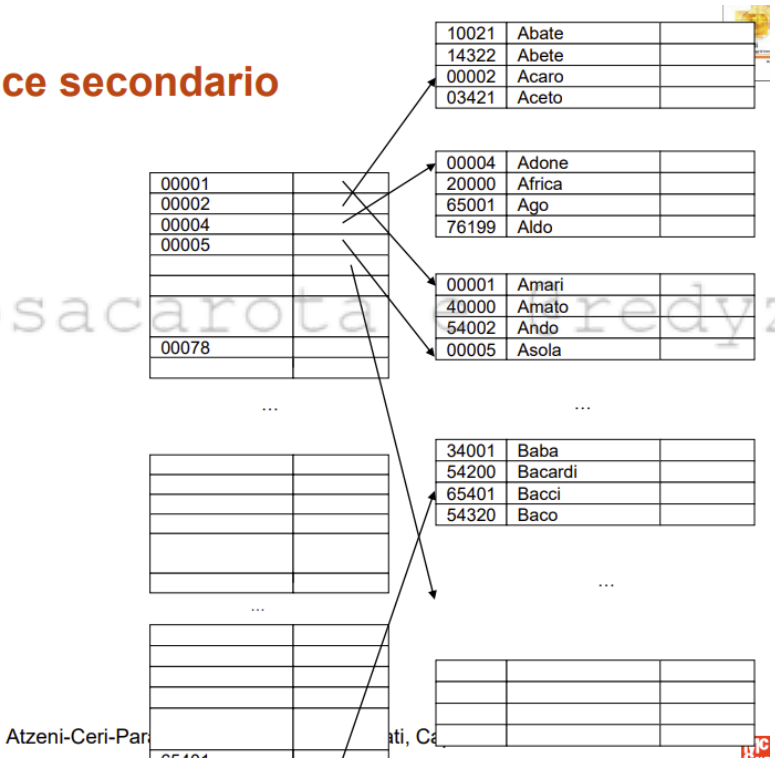
Ogni file può avere **al più un indice primario** e un **numero qualunque di indici secondari** (su campi diversi)

Esempio:

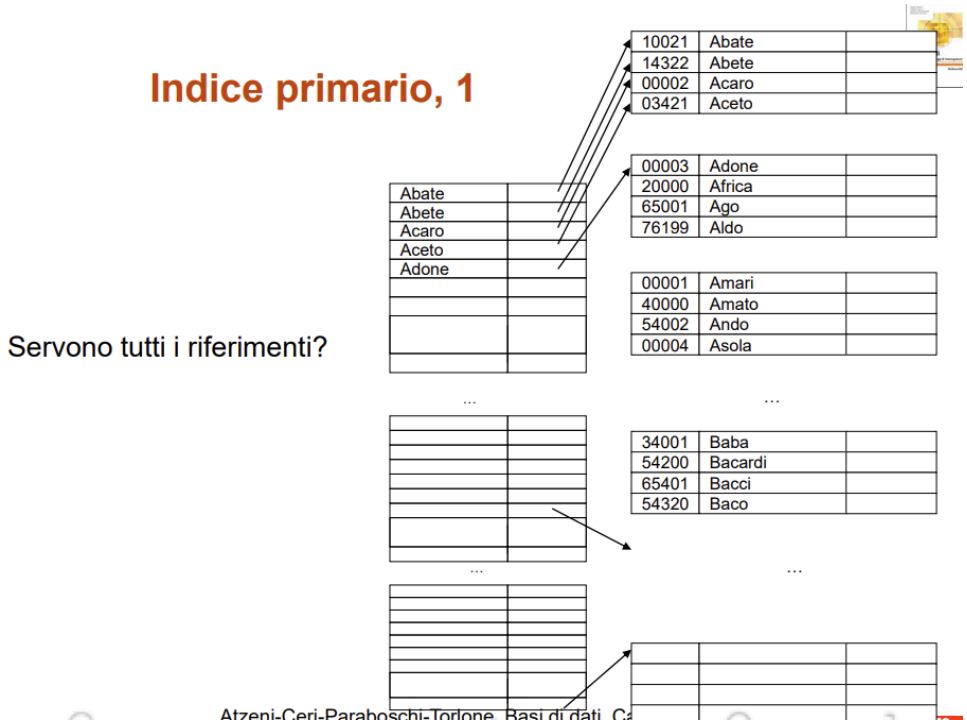
- Una guida turistica può avere l'indice dei luoghi e quello degli artisti

Un file hash **non può avere un indice primario**

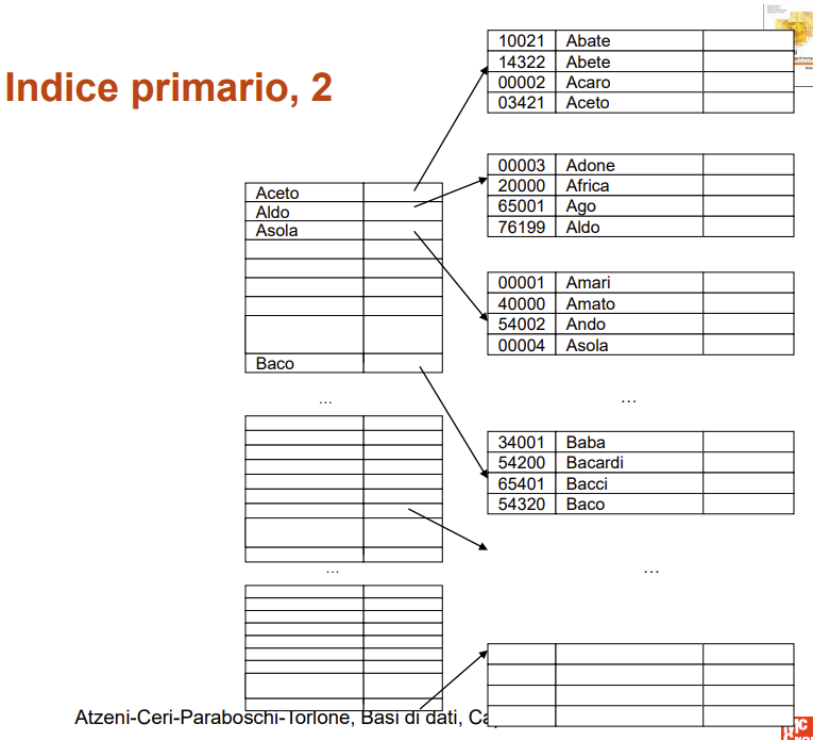
Indice secondario:



Indice primario **denso**:



Indice primario **sparso**:



Indice denso:

- Contiene tutti i valori della chiave (e quindi, per indici su campi identificanti, un riferimento per ciascun record del file)

Indice sparso:

- Contiene solo alcuni valori della chiave e quindi (anche per indici su campi identificanti) un numero di riferimenti inferiore rispetto ai record del file

Un indice primario:

- Di solito è sparso
- Se denso permette di eseguire operazioni sugli indirizzi, senza accedere ai record

Un indice secondario **deve essere denso**

Gli indici densi si possono usare come puntatori ai blocchi oppure puntatori ai record

- I puntatori ai blocchi sono più compatti
- I puntatori ai record permettono di
  - Semplificare alcune operazioni (effettuate solo sull'indice, senza accedere al file se non quando indispensabile)

Dimensioni dell'indice

L numero di record nel file

B dimensione dei blocchi

R lunghezza dei record (fissa)

K lunghezza del campo pseudo-chiave

P lunghezza degli indirizzi (ai blocchi)

N. di blocchi per il file (circa): $N_F = L/(B/R)$

N. di blocchi per un indice denso: $N_D = L/(B/(K+P))$

N. di blocchi per un indice sparso: $N_S = N_F/(B/(K+P))$

Esempio:

$L = 1.000.000$

$B = 4\ KB$

$R = 100\ B$

$K = 4\ B$

$P = 4\ B$

$N_F = L/(B/R) \approxeq 1.000.000/(4.000/100) = 25.000$

$N_D = L/(B/(K+P)) \approxeq 1.000.000/(4.000/8) = 2.000$

$N_S = N_F/(B/(K+P)) \approxeq 25.000/(4.000/8) = 50$

Caratteristiche degli indici

Accesso diretto (sulla chiave) efficiente, sia puntuale sia per intervalli

Scansione sequenziale ordinata efficiente:

- Tutti gli indici (in particolare quelli secondari) forniscono un **ordinamento logico** sui record del file; con numero di accessi pari al numero di record del file (a parte qualche beneficio dovuto alla bufferizzazione)

Modifiche della chiave, inserimenti, eliminazioni inefficienti (come nei file ordinati)

Tecniche per alleviare i problemi:

- File o blocchi di overflow
- Marcatura per le eliminazioni
- Riempimento parziale
- Blocchi collegati (non contigui)
- Riorganizzazioni periodiche
- …

Indici su campi non chiave

Ci sono (in generale) più record per un valore della (pseudo) chiave

- Primario sparso, possibili semplificazioni:

    - Puntatori solo a blocchi con valori “nuovi”
- Primario denso:
    - Una coppia con valore della chiave e riferimento per ogni record (quindi i valori della chiave si ripetono)
    - Valore della chiave seguito dalla lista di riferimenti ai record con quel valore
    - Valore della chiave seguito dal riferimento al primo record con quel valore (perde i benefici dell’indice primario denso legati alla possibilità di lavorare sui puntatori)
- Secondario denso:
    - Una coppia con valore della chiave e riferimento per ogni record (quindi i valori della chiave si ripetono)
    - Un livello (di “indirezione”) in più: per ogni valore della chiave l’indice contiene un record con riferimento al blocco di una struttura intermedia che contiene riferimenti ai record

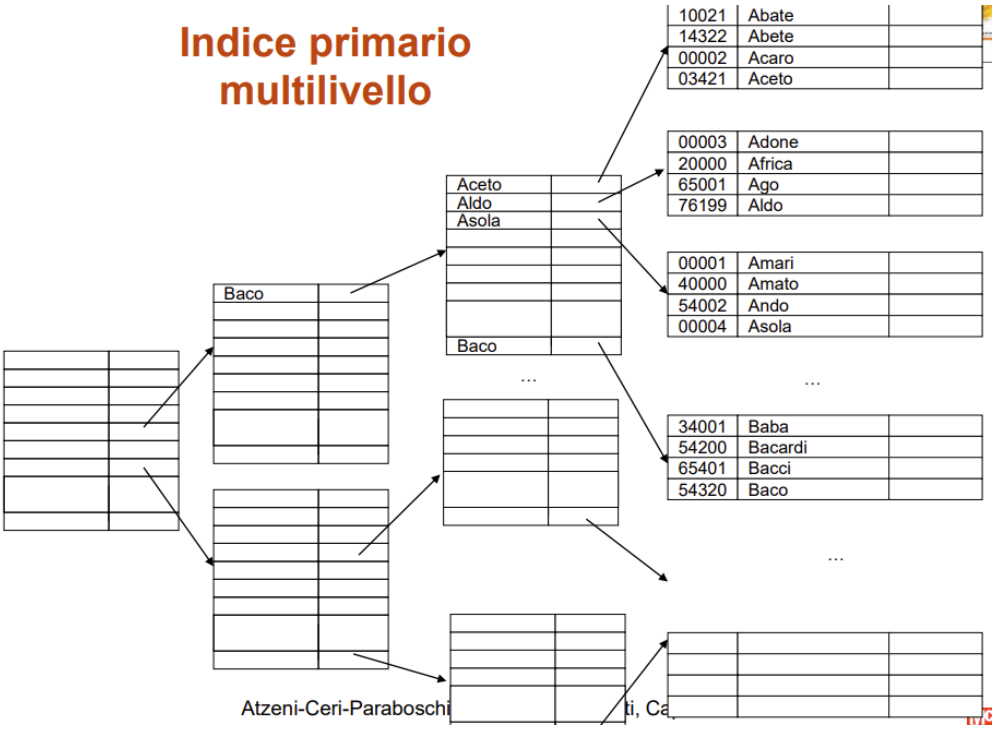
Indici multilivello

Gli indici sono file essi stessi e quindi ha senso costruire indici sugli indici, per evitare di fare ricerche fra blocchi diversi (che potrebbero richiedere scansioni sequenziali)

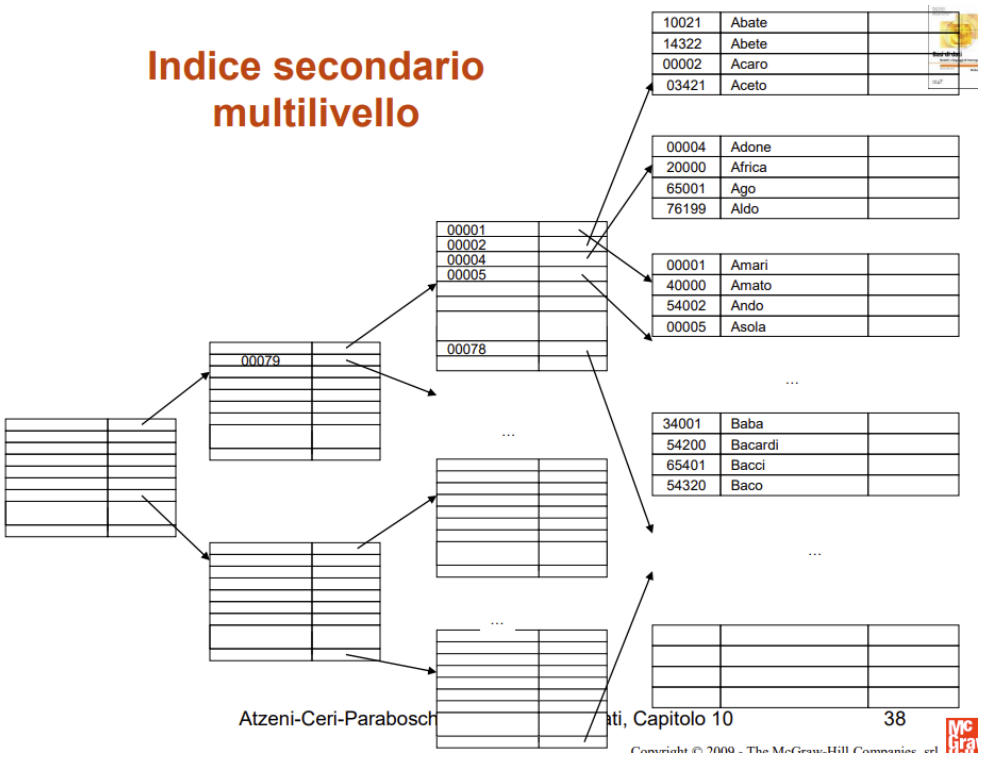
L’indice è ordinato e quindi l’indice sull’indice è primario (e sparso)

Il tutto a più livello, fino ad avere un livello con un solo blocco

Indice primario multilivello:



Indice secondario multilivello:



I livelli sono di solito abbastanza pochi, perché:

- L'indice è ordinato, quindi l'indice sull'indice è sparso
- I record dell'indice sono piccoli

N_j numero di blocchi al livello j dell'indice (circa):

- $N_j = N_{j-1}/(B/(K+P))$

Negli esempi numeri $(B/(K+P) = 4.000/8 = 500)$

- Denso: $N_1 = 2.000,\ N_2 = 4,\ N_3 = 1$
- Sparso: $N_1 = 50,\ N_2 = 1$

Indici, problemi

Tutte le strutture di indice viste finora sono basate su strutture ordinate e quindi sono poco flessibili in presenza di elevata dinamicità

Gli indici utilizzati dai DBMS sono più sofisticati:

- Indici dinamici multilivello: B-tree (intuitivamente: alberi di ricerca bilanciati)
  - Arriviamo ai B-tree per gradi
    - Alberi binari di ricerca
    - Alberi n-ari di ricerca
    - Alberi n-ari di ricerca bilanciati

Strutture ad albero dinamiche (libro)

Le strutture ad albero dinamiche (cioè efficienti anche in presenza di aggiornamenti), di tipo B(B-tree) oppure B+(B+-tree), sono le più frequentemente usate nei DBMS relazionali per la realizzazione di indici

Ogni albero è caratterizzato da un nodo radice, vari nodi intermedi e vari nodi foglia

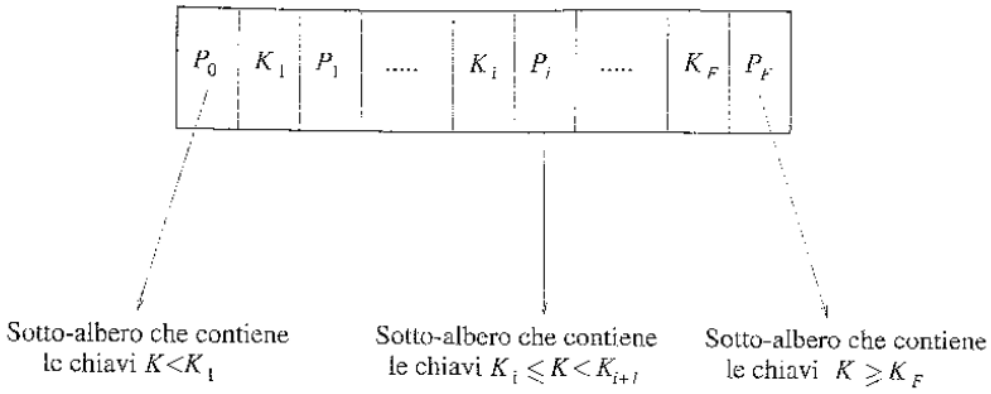
- Ogni nodo coincide con una pagina o blocco a livello di file system
- I legami tra nodi vengono stabiliti da puntatori che collegano fra loro le pagine
- Un requisito importante per il buon funzionamento di queste strutture dati è che gli alberi siano **bilanciati:**
  - Cioè che la lunghezza di un cammino che collega il nodo radice a un qualunque nodo foglia sia costante; in tal caso il tempo di accesso alle informazioni contenute

nell'albero è lo stesso per tutte le foglie ed è pari alla profondità dell'albero ($logn$)

La struttura tipica di un nodo non foglia di un albero è formata da una sequenza di F valori ordinati di chiave

Ogni chiave $K_i, 1 \leq i \leq F$, è seguita da un puntatore P_i

K_1 è preceduta da un puntatore P_0



- Il puntatore P_0 indirizza al sotto-albero che permette di accedere ai record con chiavi minori di K_1
- Il puntatore P_F indirizza al sotto-albero che permette di accedere ai record con chiavi maggiori o uguali a K_F
- Ciascun puntatore intermedio P_i, $0 \leq i \leq F$, indirizza un sotto-albero che contiene chiavi comprese nell'intervallo $[K_i, K_{i+1})$

In sintesi ciascun nodo contiene F valori chiave e $F+1$ puntatori; il valore $F+1$ viene detto **fan-out** dell'albero

I vari nodi possono contenere meno di F valori e meno di $F+1$ puntatori poiché è necessario prevedere un riempimento parziale e flessibile

La ricerca inizia alla radice e prosegue passando per i nodi interni fino ad arrivare ai nodi foglia dell’albero

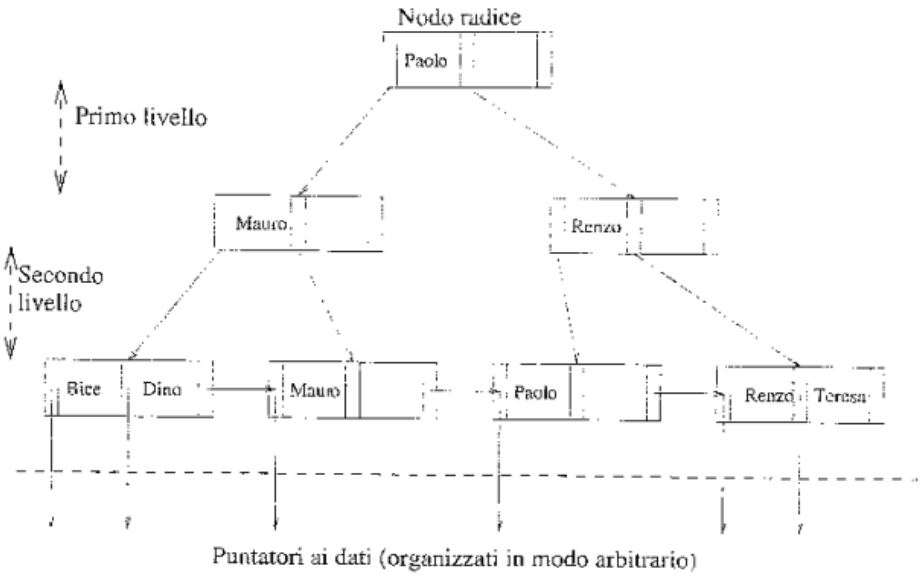
Gli inserimenti e le cancellazioni di tuple provocano anche aggiornamenti degli indici, che devono riflettere la situazione generata da una variazione dei valori del campo chiave

Alberi B e B+ (libro)

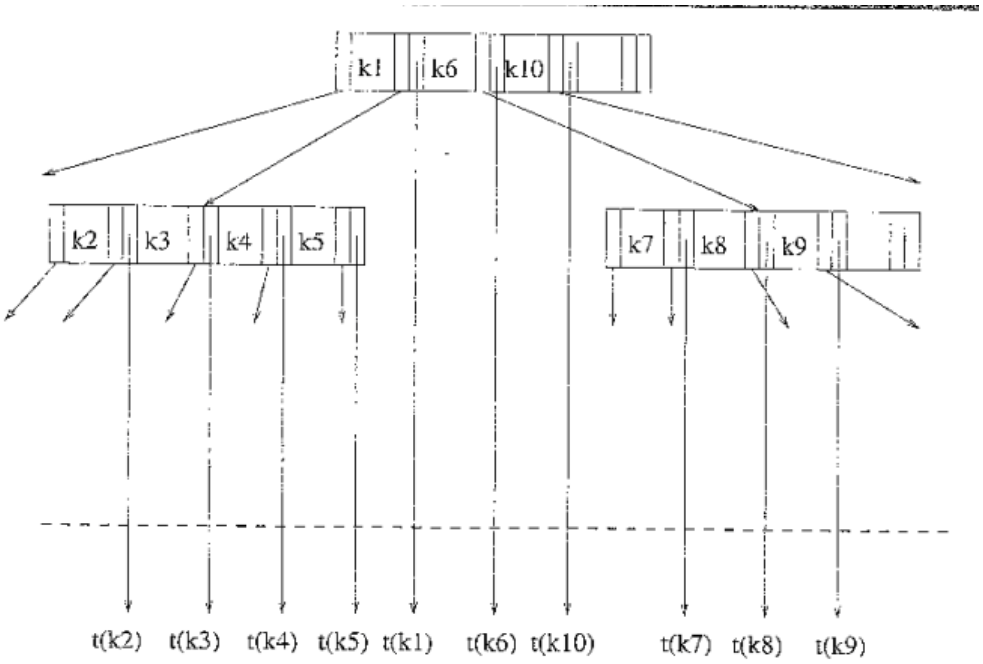
La distinzione degli alberi B e B+ è la seguente

- Negli alberi B+, i nodi foglia sono collegati da una catena che li connette in base all’ordine imposto dalla chiave. Tale catena consente di svolgere in modo efficiente anche interrogazioni il cui predicato di selezione definisce un intervallo di valori ammissibili, poiché i nodi foglia vengono scanditi in maniera sequenziale
- Nelle strutture B-tree non viene previsto di collegare sequenzialmente i nodi foglia
  - In tal caso si prevede di usare nei nodi intermedi due puntatori per ogni valore di chiave K_i; uno dei due puntatori viene utilizzato per puntare direttamente al blocco che contiene la tupla corrispondente a K_i, interrompendo la ricerca.
    L’altro serve per proseguire la ricerca nel sotto-albero che comprende i valori di chiave strettamente compresi fra K_i e K_{i+1}
  - Tale tecnica consente un risparmio di spazio, in quanto i valori di chiave sono presenti al più una volta nell’albero, mentre nei B+ tutti i valori sono presenti nelle foglie e alcuni vengono ripetuti nei livelli superiori

Esempio di struttura B+-tree:



Esempio di struttura B-tree:



Albero binario di ricerca

Albero binario etichettato in cui per ogni nodo il sottoalbero sinistro contiene solo etichette minori di quella del nodo e il sottoalbero destro etichette maggiori

Tempi di ricerca (e inserimento), pari alla profondità:

- Logaritmico nel caso “medio” (assumendo un ordine di inserimento casuale)i

Albero di ricerca di ordine P

Ogni nodo ha (fino a) P figli e (fino a) $P-1$ etichette, ordinate

Nell’i-esimo sottoalbero abbiamo tutte etichette maggiori della i-1-esima etichetta e minori della i-esima

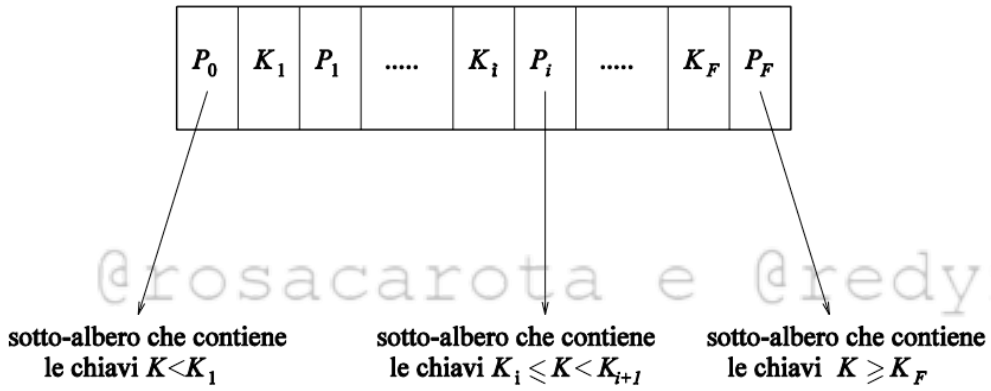
Ogni ricerca o modifica comporta la visita di un cammino radice foglia

In strutture fisiche, un nodo corrisponde di solito ad un blocco di record e quindi ogni nodo intermedio ha molti figli (un “fan-out” molto grande, pari al fattore di blocco dell’indice)

All’interno di un nodo, la ricerca è sequenziale (ma in memoria centrale)

La struttura è ancora (potenzialmente) rigida

Nodi in un albero di ricerca di ordine $F+1$:



B-Tree

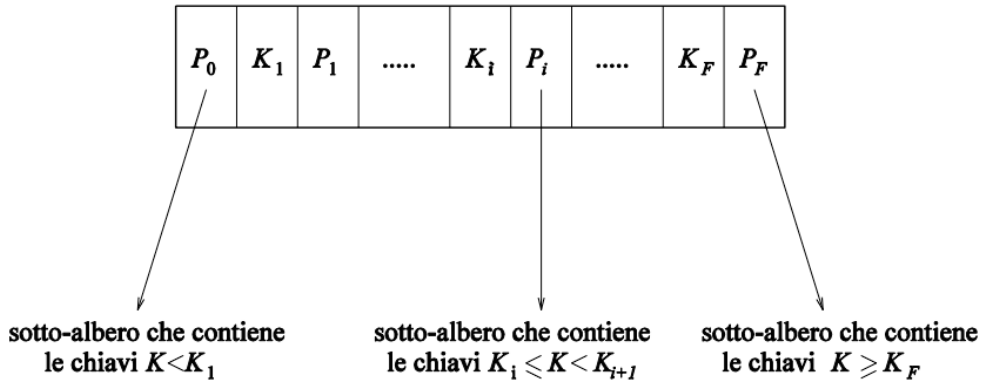
Albero di ricerca in cui ogni nodo corrisponde ad un blocco

Viene mantenuto perfettamente bilanciato (tutte le foglie sono allo stesso livello), grazie a:

- Riempimento parziale (mediamente 70%)

- Riorganizzazioni (locali) in caso di sbilanciamento

Organizzazione dei nodi del B-tree:



Split e merge

Inserimenti ed eliminazioni sono precedute da una ricerca fino ad una foglia

Per gli inserimenti, se c'è posto nella foglia, ok, altrimenti il nodo va suddiviso, con necessità di un puntatore in più per il nodo genitore; se non c'è posto, si sale ancora, eventualmente fino alla radice. Il riempimento rimane sempre superiore al 50%

Dualmente, le eliminazioni possono portare a riduzioni di nodi

Modifiche del campo chiave vanno trattato come eliminazioni seguite da inserimenti

Split e merge:

I nodi, in questo caso, possono anche ripetersi

Esempio 1

Calcolo dell'ordine p di un B-Tree su disco:

- Campo di ricerca di $V = 9\ byte$
- Dimensione blocchi su disco $B = 512\ byte$
- Puntatore a record di $Pr = 7\ byte$
- Puntatore a blocco di $P = 6\ byte$

Ogni nodo del B-Tree, contenuto in un blocco del disco, può avere al più p puntatori ad albero, $p - 1$ puntatori a record e $p - 1$ valori del campo chiave di ricerca

Quindi deve essere:

- $(p * P) + ((p - 1) * (Pr + V)) \leq B \rightarrow (p * 6) + ((p - 1) * (7 + 9)) \leq 512 \rightarrow (22 * p) \leq 528 \rightarrow p \leq 24$

Scegliamo $p = 23$, poiché in ogni nodo ci potrebbe essere la necessità di memorizzare informazioni addizionali, quali il numero di entrate q on un puntatore al nodo padre

Esempio 2

Calcolo del numero di blocchi in un B-Tree:

- Supponiamo di costruire un B-Tree sul campo di ricerca dell'esempio precedente, non ordinato e chiave
- Supponiamo che ogni nodo sia pieno per il 69%
- Ogni nodo avrà in media $p * 0,69 = 23 * 0,69$ = circa 16 puntatori (fo medio) e quindi 15 valori del campo chiave di

ricerca. Partendo dalla radice vediamo quanti valori e puntatori esistono in media a ogni livello:

- Radice: 1 nodo, 15 entry, 16 ptr
- Livello 1: 16 nodi, 240(16*15) entry, 256(16*16) ptr
- Livello 2: 256 nodi, 3840(256*15) entry, 4096 ptr
- Livello 3: 4096 nodi, 61440(4096*15) entry (4096*16) ptr
- Per ogni livello:
  - # entrate livello i-mo = [# puntatori livello (i-1)-mo] * 15

Per le date dimensioni di blocco, puntatore e campo chiave di ricerca, un B-Tree a 3 livelli contiene fino a 3840 + 240 + 15 = 4096 entrate

Un B-Tree a 4 livelli contiene fino a 4095 + 61440 = 65535 entrate

B-Tree e B+Tree

B+ Tree:

- Le chiavi compaiono tutte nelle foglie (e quindi quelle nei nodi intermedi sono comunque ripetute nelle foglie)
- Le foglie sono collegate in una lista
- Ottimi per le ricerche su intervalli
- Molto usati nei DBMS

B-Tree:

- Le chiavi che compaiono nei nodi intermedi non sono ripetute nelle foglie

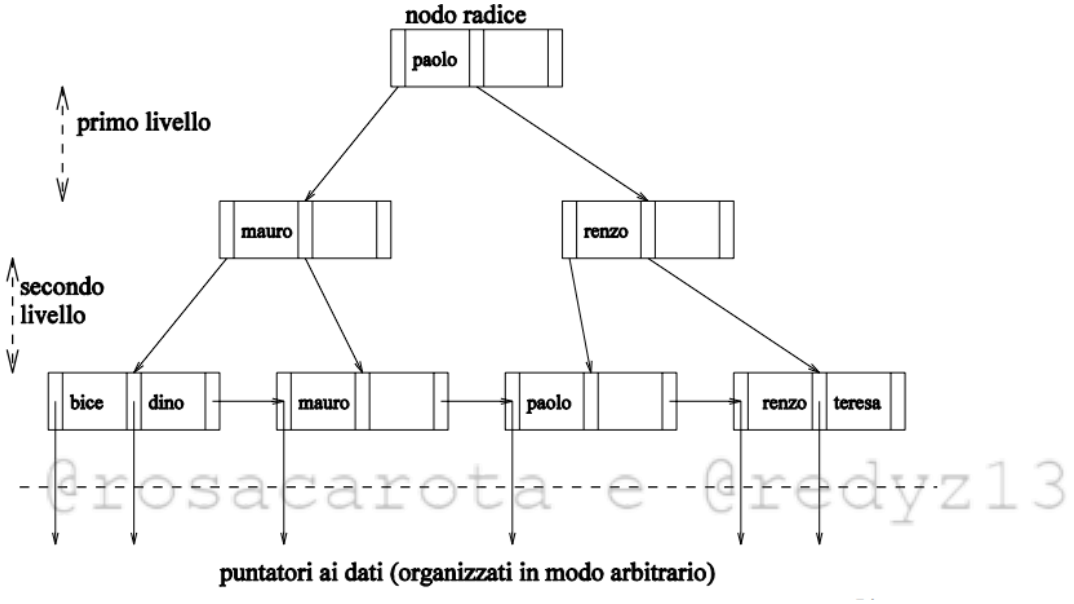
In un B+ Tree:

- Primario, le ennuple possono essere contenute nelle foglie
- Secondario, le foglie contengono puntatori alle ennuple

In un B-Tree:

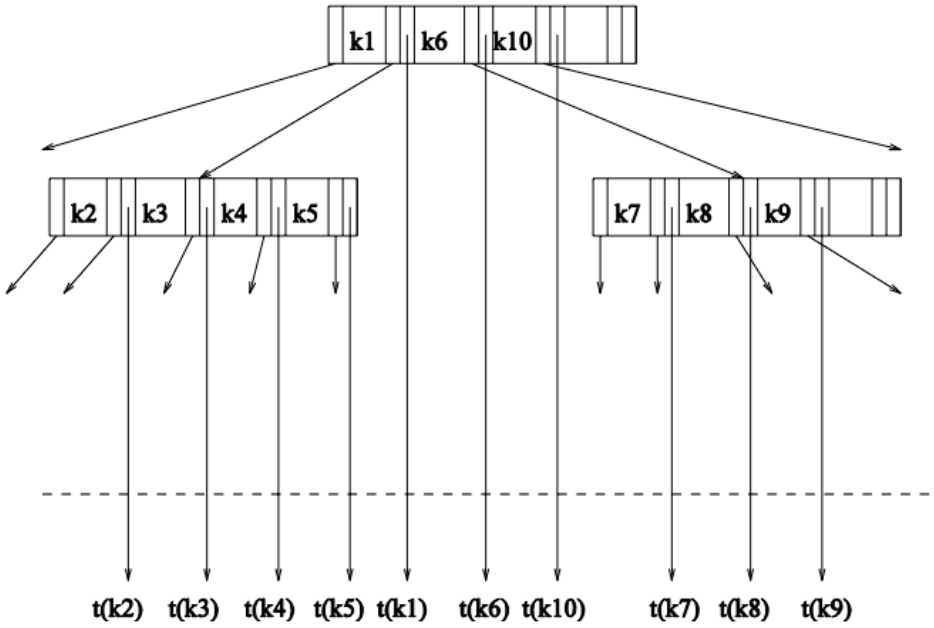
- Anche i nodi intermedi contengono ennuple (se primari) o puntatori (se secondari)

Un B+ Tree:



Nei B+ Tree le foglie formano una lista a puntatori (nella parte terminale destra) con il puntatore alla prossima foglia

Un B-Tree:



Esempio 1:

Calcolo dell'ordine p di un B+ Tree su disco:

- Campo di ricerca di $V = 9\ byte$
- Dimensione blocchi su disco $B = 512\ byte$
- Puntatore a record di $Pr = 7\ byte$
- Puntatore a blocco di $P = 6\ byte$

Un nodo interno di un B+ Tree, contenuto in un singolo blocco, può avere fino a p puntatori ad albero e $p - 1$ valori del campo di ricerca. Quindi deve essere:

- $(p * P) + ((p - 1) * V) \leq B \rightarrow (p * 6) + ((p - 1) * 9) \leq 512 \rightarrow (15 * p) \leq 521$

Il massimo intero che soddisfa la disuguaglianza è $p = 34$, che è maggiore di 23 (ordine del B-Tree corrispondente). Quindi ne risulta un fan-out maggiore e più entrante in ogni nodo interno

L'ordine P_{leaf} dei nodi foglia è:

- $(P_{leaf} * (Pr + V)) + P$ (per memorizzare P_{next}) $\leq B \rightarrow$
  $(P_{leaf} * (7 + 9)) + 6 \leq 512 \rightarrow$
  $(P_{leaf} * 16) \leq 506$

Quindi ogni nodo foglia può avere fino a $P_{leaf} = 31$ combinazioni valore chiave/data pointer, supponendo che i data pointer siano puntatori a record

Esempio 1:

Calcolo del numero di blocchi e di livelli in un B+-Tree

- Supponiamo che ogni nodo del B+-Tree sia pieno per il 69%
- Ogni nodo interno avrà in media $p * 0.69 = 34 * 0.69 =$ circa 23 puntatori (fan-out medio) e quindi 22 valori del campo di ricerca. Ogni nodo foglia avrà in media $P_{leaf} * 0.69 = 31 * 0.69 =$ circa 21 puntatori a record dati

Partendo dalla radice vediamo quanti valori e puntatori esistono in media a ogni livello:

- Radice: 1 nodo, 22 entry, 23 ptr
- Livello 1: 23 nodi, 506 (22*23) entry, 529 (23*23) ptr
- Livello 2: 529 nodi, 11638 (22*529), 12167 ptr
- Livello foglie: 12167 nodi, 255507 (21*12167) puntatori a record

Quindi un B+-Tree di quattro livelli ha fino a 255.507 puntatori a record, contro i 65.535 calcolati per il B-Tree corrispondente

Inserimento in un B+-Tree

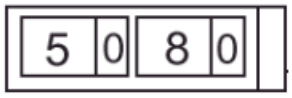
Inizialmente la radice è l'unico nodo dell'albero: essendo quindi un nodo foglia conterrà anche i puntatori ai dati

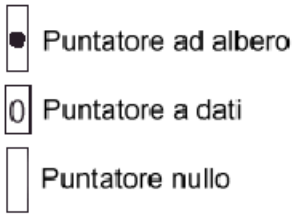
Se si cerca di inserire un'entry in un nodo foglia pieno, il nodo va in overflow e deve essere scisso:

- Le prime $j = \lceil (p_{leaf} + 1)/2 \rceil$ entry sono mantenute nel nodo, mentre le rimanenti sono spostate in un nuovo nodo foglia
  - Il j-mo valore di ricerca (quello mediano) è replicato nel nodo padre
  - Nel padre viene creato un puntatore al nuovo nodo
- Se il nodo padre (interno) è pieno si ha un altro overflow
  - Il nodo viene diviso, creando un nuovo nodo interno
  - Le entrate nel nodo interno fino a P_j (con $j = \lfloor (p + 1)/2 \rfloor$) sono mantenute nel nodo originario, mentre il j-mo valore di ricerca è spostato al padre, non replicato
  - Il nuovo nodo interno conterrà le entrare dalla P_{j+1} alla fine delle entrate del nodo originario
  - Tale scissione si può propagare verso l'alto fino a creare un nuovo livello per il B+-Tree

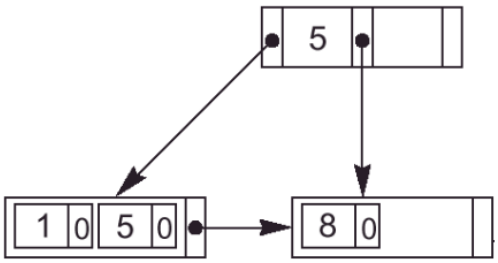
Esempio di inserimento

Vogliamo inserire dei record in un B+-Tree di ordine $p = 3$ e $P_{leaf} = 2$, nella sequenza $8, 5, 1, 7, 3, 12$





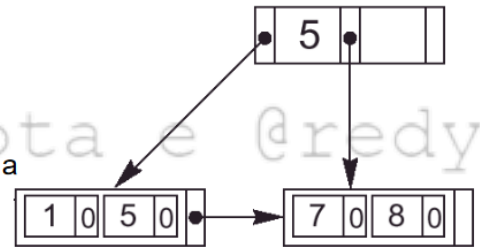
L'inserimento dei valori 8 e 5 non provoca overflow: sono entrambi inseriti nella radice

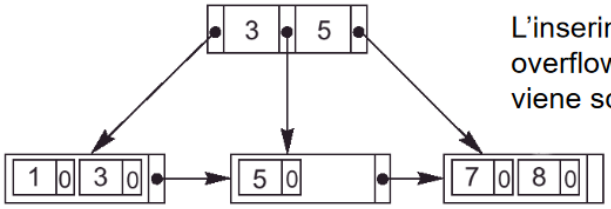


L'inserimento di 1 provoca overflow. Il nodo è scisso ed il valore mediano è ripetuto in un nuovo nodo radice

L'inserimento di 7 non provoca overflow.

Si noti che tutti i valori sono a livello foglia, perché i data pointer sono tutti a quel livello. Alcuni sono replicati nei nodi interni per guidare la ricerca.

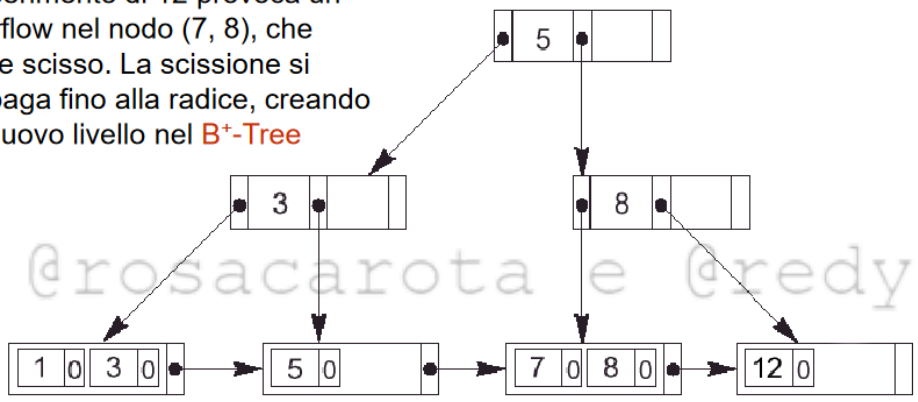




L'inserimento di 3 provoca un overflow nel nodo (1, 5), che viene scisso.

Si noti che un valore che compare nel nodo interno, compare anche come valore più a destra del sottoalbero referenziato dal puntatore alla sinistra di tale valore

L'inserimento di 12 provoca un overflow nel nodo (7, 8), che viene scisso. La scissione si propaga fino alla radice, creando un nuovo livello nel B^+-Tree



Cancellazione in un B+-Tree

Un'entry è cancellata sempre a livello foglia

Se essa ricorre anche in un nodo interno, allora è rimossa anche da quello e al suo posto si inserisce il valore immediatamente alla sinistra del valore rimosso nel nodo foglia

La cancellazione di valori può causare l'**underflow** di un nodo, riducendo il numero di entry in un nodo foglia per meno del minimo consentito

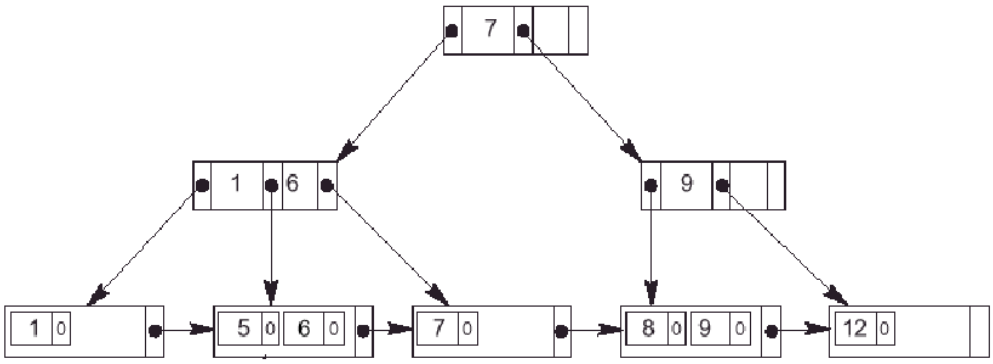
- In tal caso si cerca un fratello del nodo foglia in modo da ridistribuire le entrate, cosicché entrambi i nodi siano pieni almeno per metà
- Se ciò non è possibile si effettua il merge del nodo con i suoi fratelli, riducendo di uno il numero di nodi foglia

Approccio comune:

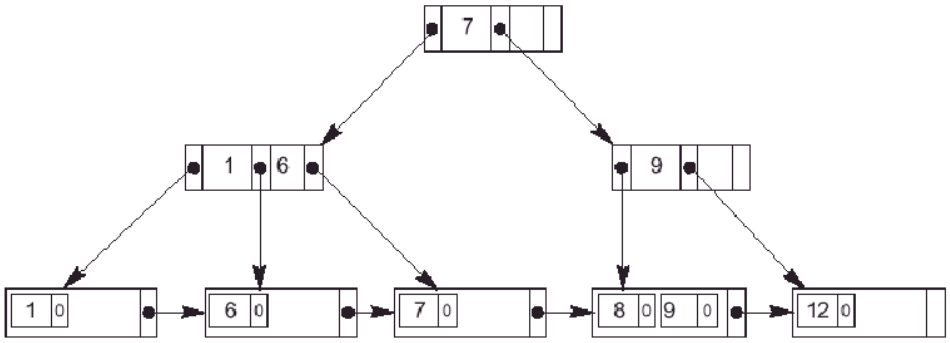
- Si tenta di ridistribuire le entrate con il fratello sinistro
- Se non è possibile, si tenta di ridistribuire con il fratello destro
- Altrimenti si fondono i tre nodi in due nodi foglia. L'underflow in questo caso si può trasmettere ai nodi interni, poiché sono necessari un puntatore ad albero ed un valore di ricerca in meno. Se la propagazione arriva alla radice, si riduce il numero di livelli dell'albero

Esempio di cancellazione

Dato il seguente B+-Tree, vogliamo cancellare i record $5, 12$ e 9

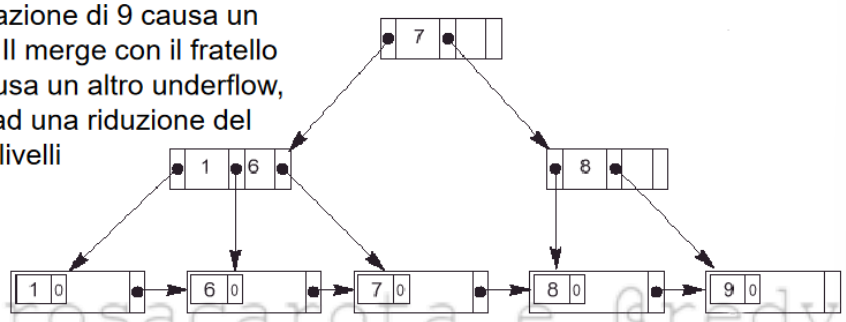


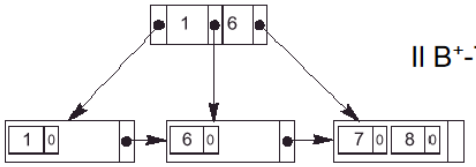
La cancellazione di 5 non pone alcun problema



La cancellazione di 12 viene risolta con una ridistribuzione

La cancellazione di 9 causa un underflow. Il merge con il fratello sinistro causa un altro underflow, che porta ad una riduzione del numero di livelli





Il B⁺-Tree risultante

70

Progettazione fisica

La fase finale del processo di progettazione di basi di dati

- Input
  - Lo schema logico e informazioni sul carico applicativo
- Output
  - Schema fisico, costituito dalle definizioni delle relazioni con le relative strutture fisiche (e molti parametri, spesso legati allo specifico DBMS)

Strutture fisiche nei DBMS relazionali

Tutti i sistemi in generale prevedono una struttura base che è disordinata (e quindi la struttura seriale) su cui è possibile definire indici secondari

Peraltro, quasi tutti i sistemi creano un indice per la chiave primaria, perché in questo modo risulta semplice verificare il rispetto del vincolo

Alcuni sistemi utilizzano il termine *indice primario* per fare riferimento a un indice definito sulla chiave primaria; si tratta quindi di una terminologia diversa da quella qui adottata secondo cui un indice è primario se realizzato sul campo su cui il file è ordinato

Molti sistemi prevedono la possibilità di memorizzare in modo contiguo le tuple di una tabella con g li stessi valori su un certo campo

- Questa tecnica è detta **cluster** e può essere sia associata a una funzione hash o a un indice, in altri sistemi può trattarsi di un vero e proprio ordinamento fisico

Tutti i sistemi permettono di realizzare indici e praticamente tutti prevedono realizzazioni basate su B-tree (o meglio, su B+-tree, anche se la documentazione parla spesso di B-tree)

Struttura primaria:

- Disordinata (heap, unclustered)
- Ordinata (clustered), anche su una pseudochiave
- Hash (clustered), anche su una pseudochiave, senza ordinamento
- Clustering di più relazioni

Indici (densi/sparsi, semplici/compositi):

- ISAM (statico), di solito su struttura ordinata
- B-tree (dinamico)
- Indici hash (secondario, poco dinamico)

Strutture fisiche in alcuni DBMS

Oracle:

- Struttura primaria:
  - File heap
  - Hash cluster (cioè struttura hash)
  - Cluster (anche plurirelazionali) anche ordinati (con B-tree denso)
- Indici secondari di vario tipo (B-tree, bit-map, funzioni)

DB2:

- Primaria: heap o ordinata con B-tree denso
- Indice sulla chiave primaria (automaticamente)
- Indici secondari B-tree densi

SQL Server:

- Primaria: heap o ordinata con indice B-tree sparso
- Indici secondari B-tree densi

Ingres (anni fa):

- File heap, hash, ISAM (ciascuno anche compresso)
- Indici secondari

Informix (per DOS, 1994):

- File heap
- Indici secondari (e primari [cluster] ma non mantenuti)

Definizione degli indici SQL

I comandi definiti di seguito non fanno parte dello standard SQL-3 del linguaggio perché non si è raggiunto un accodo all'interno del comitato di standardizzazione, tale sintassi è comunque molto diffusa

Creazione di un indice:

```
CREATE [UNIQUE] INDEX IndexName ON TableName(AttributeList)
```


- L'ordine della lista di attributi è significativa, infatti, le chiavi dell'indice vengono ordinate prima in base ai valori del primo attributo della lista, poi del secondo, e così via

Cancellazione indici:

```
DROP INDEX IndexName
```

Progettazione fisica nel modello relazionale

La caratteristica comune dei DBMS relazionali è la disponibilità degli indici:

- La progettazione fisica spesso coincide con la scelta degli indici (oltre ai parametri strettamente dipendenti dal DBMS)

Le chiavi (primarie) delle relazioni sono di solito coinvolte in selezioni e join: molti sistemi prevedono (oppure suggeriscono) di definire indici sulle chiavi primarie

Altri indici vengono definiti con riferimento ad altre selezioni o join “importanti”

Se le prestazioni sono insoddisfacenti, si “tara” il sistema aggiungendo o eliminando indici

È utile verificare se e come gli indici sono utilizzati con il comando SQL `SHOW PLAN` oppure `EXPLAIN`