

5. Strutture di indici per file

I file hanno un'organizzazione *primaria*, ossia possono essere organizzati su disco in maniera ordinata, non ordinata, oppure a hash. Per velocizzare le operazioni di reperimento delle informazioni, possiamo quindi aggiungere una struttura aggiuntiva chiamata **indice**, che fornisce un metodo alternativo per accedere ai record del file (*secondary access path*).

Concettualmente, una struttura di accesso secondaria è simile all'indice di un libro: tutti i termini principali sono listati in ordine alfabetico, e cercando un termine nell'indice otteniamo una lista di indirizzi (pagine), che ci permettono di localizzare velocemente il testo richiesto.

Gli indici, quindi, permettono di accelerare notevolmente le operazioni di ricerca in particolari condizioni, senza modificare il posizionamento fisico dei record. Di norma sono basati su un singolo file ordinato (indici a livello singolo), o su strutture dati ad albero (indici multilivello, B+-alberi).

Indici a livello singolo

In un file con più campi, in genere si definisce l'indice su un solo campo, detto indexing field. Un indice memorizza il valore del campo index, una lista di puntatori a tutti i disk block che contengono record con quel valore di campo. I valori nell'indice sono ordinati così da consentire l'esecuzione di una ricerca binaria. La ricerca è più efficiente poiché il file indice è più piccolo del file di dati.

Esistono più tipi di indici:

1. **Indice primario**: specificato su un campo chiave di ordinamento (ordering key field) di un file ordinato di record.
2. **Indice clustering**: specificato su un campo non chiave di ordinamento (ordering non-key field) di un file ordinato di record.
3. **Indice secondario**: specificato su un campo non di ordinamento (non-ordering field) di un file di record.

Primary Index

Un *primary index* è un file ordinato di record a lunghezza fissa con due campi. Ogni record, detto **index entry**, ha la seguente struttura:

- Il 1° campo, dello stesso tipo di dati della chiave primaria, contiene il valore della chiave primaria del primo record di un blocco (detto *record ancora*).
- Il 2° campo: di tipo puntatore a un disk block contiene l'indirizzo del blocco contenente il record.

I due valori dei campi dell'indice si indicano con $\langle K(i), P(i) \rangle$. Il numero di entry è pari al numero di blocchi del file dati ordinato. (vedi figura 1).

Gli indici possono essere densi o sparsi: è **denso** se contiene un'entry per ogni possibile valore del campo chiave, è **sparsa** se contiene meno entry rispetto al numero di valori del campo chiave. Gli indici primari sono sparsi: il file indice è molto più piccolo del file dati poiché il numero di entry è inferiore a quello dei record del file dati; inoltre la dimensione di un'entry è inferiore a quella di un record di dati (ha solo due campi).

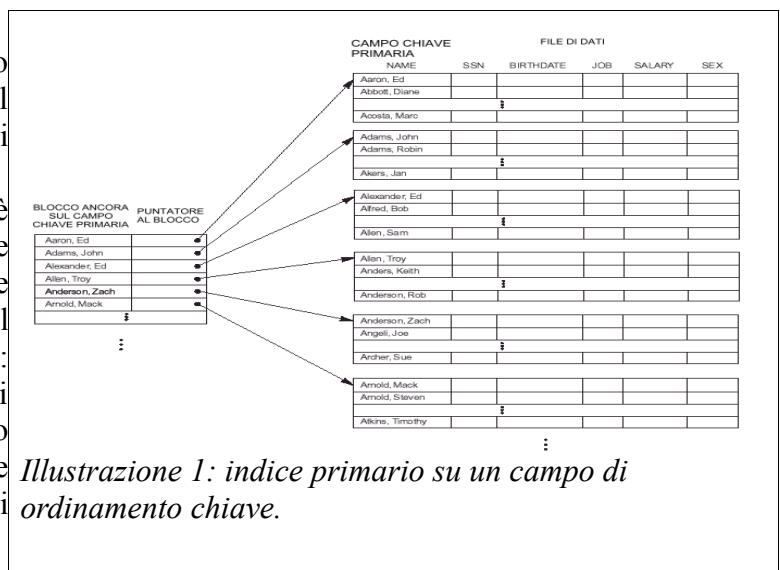


Illustrazione 1: indice primario su un campo di ordinamento chiave.

Retrieval con indici primari

Un record con valore chiave K è posizionato nel blocco di indirizzo $P(i)$, dove $K(i) < K < K(i+1)$: l'*i*-mo blocco contiene tutti i record che soddisfano tale formula, poiché i dati sono ordinati fisicamente sul campo chiave.

Per trovare un record con valore di chiave primaria K :

- Si effettua una ricerca binaria sul file indice per trovare l'entry i corrispondente
- Si recupera il blocco $P(i)$ del file di dati.

La ricerca binaria sul file indice è molto efficiente grazie alle sue ridotte dimensioni.

Esempio. Supponiamo di avere:

- File ordinato di $r = 30.000$ record
- Dimensione di un blocco $B = 1.024$ byte
- Ogni record è a lunghezza fissa $R=100$ byte
- Il bfr $(B/R)_{inf}$ è $(1024/100)_{inf} = 10$ record per blocco
- Il numero di blocchi b $(r/bfr)_{sup}$ è $(30000/10)_{sup} = 3000$ blocchi

Effettuando una ricerca binaria sul file dati dovremmo effettuare:

$(\log_2 b)_{sup} = (\log_2 3000)_{sup} = \mathbf{12}$ accessi a blocco.

Usando invece un *indice primario*:

- Campo chiave di ordinamento $V = 9$ byte
- Campo puntatore a blocco $P = 6$ byte
- dimensione di ogni entrata $R_i = (9+6) = 15$ byte
- Num. entrate $r_i = b$ (# blocchi file dati) = 3000
- bfr_i = $(B/R_i)_{inf} = (1024/15)_{inf} = 68$ entrate a blocco
- $b_i = (r_i/bfr_i)_{sup} = (3000/68)_{sup} = 45$ blocchi

Una ricerca binaria sul file indice richiede circa $(\log_2 b_i) = (\log_2 45) = 6$ accessi a blocchi del file indice + 1 accesso al blocco del file dati = **7 accessi a blocco**.

Inserimento e cancellazione

L'inserimento e la cancellazione presentano difficoltà (come per i file ordinati): Per inserire un nuovo record, bisogna:

- spostare records per fare spazio al nuovo record e
- cambiare alcune entry nel file indice.

Le possibili soluzioni sono 1. l'utilizzo di un file di overflow non ordinato (come per i sorted file), oppure 2. l'utilizzo di una lista a puntatori di record di overflow per ciascun blocco nel file dati.

Per cancellare un record, si usano i marcatori di cancellazione.

Indici di Clustering

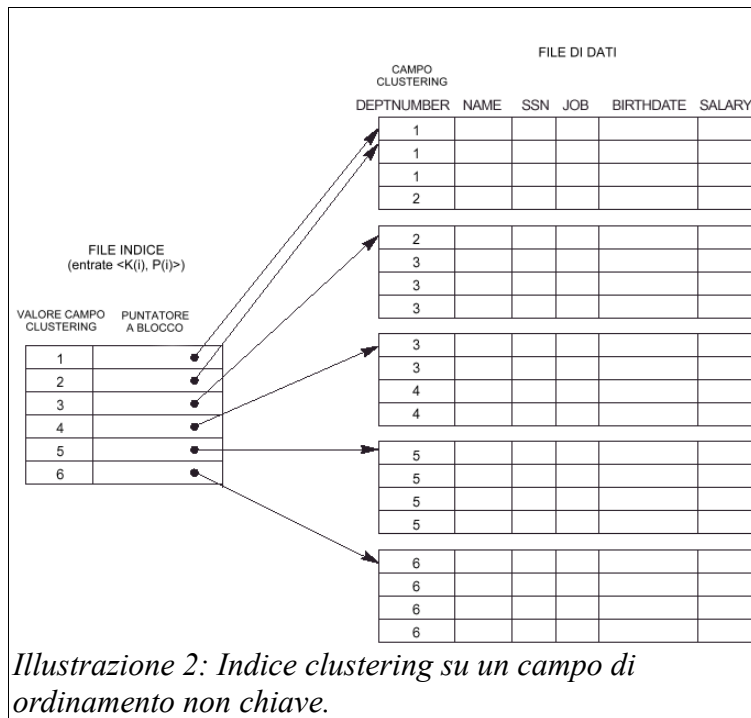
Gli **indici di clustering** si costruiscono su un file i cui record sono ordinati su un campo non chiave, detto *campo clustering*, che può avere ripetizioni di valori. L'indice di clustering è un file ordinato con due campi:

- 1° campo: stesso tipo di dati del campo clustering, contiene un valore del campo clustering.
- 2° campo: di tipo puntatore a un disk block, riferenzia il **primo blocco** del file dati che contiene un record con tale valore del campo clustering.

È un indice sparso: c'è una entry per ogni valore distinto del campo clustering.

Inserimento e cancellazione

L'inserimento e la cancellazione comportano delle difficoltà (i record sono ancora ordinati fisicamente). Per migliorare la situazione spesso si riserva un intero blocco per ogni valore del campo cluster.



Indici primari Vs. Hashing

Gli indici primari ricordano le directory usate per l'hashing: su entrambi si esegue la ricerca di un puntatore a un blocco dati che contiene il record desiderato, ma mentre una ricerca su indice usa i valori del campo di ricerca stesso, una ricerca su una directory hash usa il valore hash calcolato applicando la funzione hash al campo di ricerca.

Indici Secondari

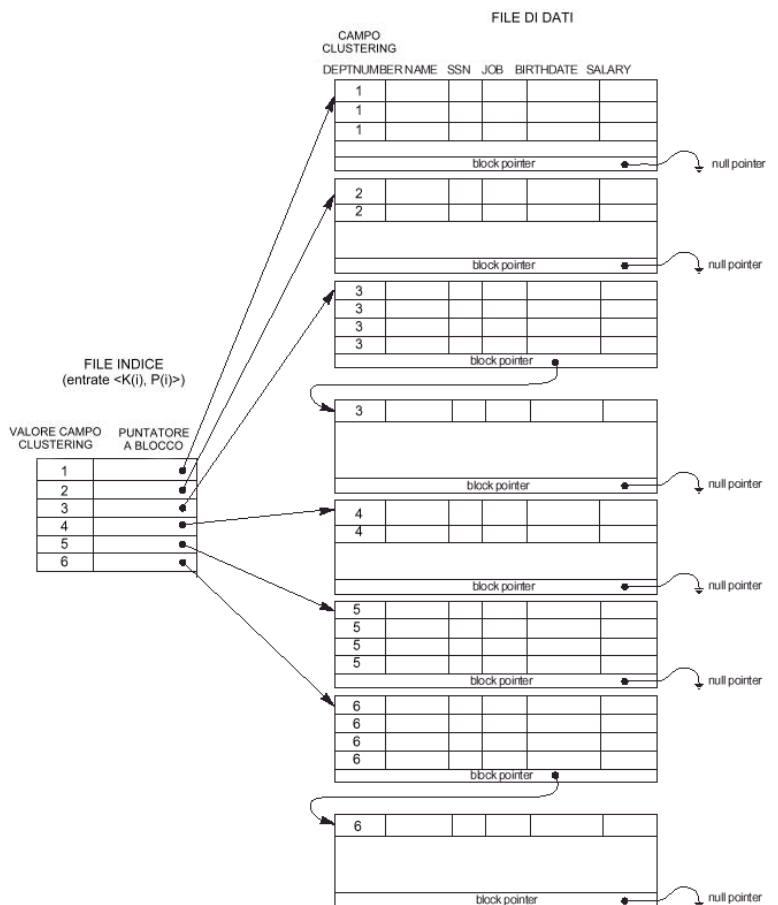
A differenza dell'indice primario, è possibile avere più indici secondari associati allo stesso file: In questo modo si velocizzano ricerche effettuate su campi non di ordinamento, creando nuove strutture di accesso.

Un indice secondario è un file ordinato con due campi:

- 1° campo (di indicizzazione): stesso tipo di dati di un campo non-ordering.
- 2° campo: puntatore a un disk block o a un record.

Esistono due tipi di indici secondari, in base al campo di indicizzazione:

1. Il campo di indicizzazione è un campo chiave, detto chiave secondaria, contenente un valore distinto per ogni record nel file dati.
2. Il campo di indicizzazione è un campo non chiave, in cui più record nel file



Caso 1

Nel primo caso, l'indice è denso: c'è un'entrata per ogni record nel file dati e Le block anchor non si possono usare perché i record del file dati non sono ordinati fisicamente rispetto al campo chiave secondaria.

Le entrate dell'indice $\langle K(i), P(i) \rangle$ sono ordinate sul valore $K(i)$, quindi si può eseguire una ricerca binaria sull'indice.; $P(i)$ è un puntatore al blocco.

Se non esistesse l'indice il tempo di ricerca sarebbe lineare poiché il file non è ordinato su quel campo.

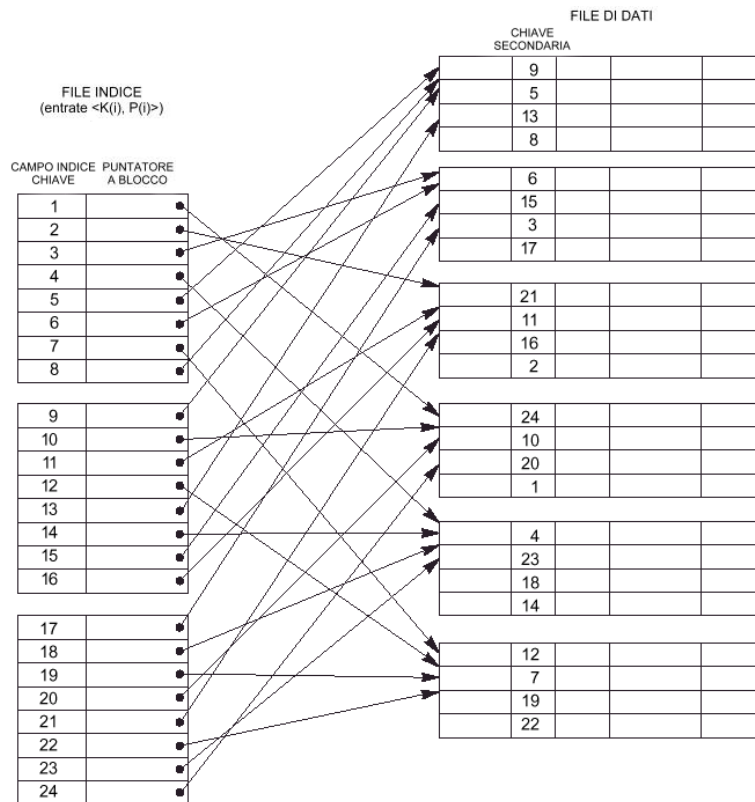


Illustrazione 4: Un indice secondario denso, su un campo non di ordinamento

Caso 2

Il file indice secondario è creato su un campo non chiave (oltre che non-ordering), quindi più record nel file dati possono avere lo stesso valore del campo di indicizzazione.

Possibili implementazioni:

1. Includere più entry con lo stesso valore $k(i)$, una per ogni record (indice denso).
2. Avere record di lunghezza variabile per le entry, con un campo puntatore: si crea una lista a puntatori $\langle P(i,1), \dots, P(i,k) \rangle$ nell'entrata per $k(i)$, con un puntatore a ciascun blocco con valore di indexing field $k(i)$.
3. Si crea un nuovo livello di indizione per gestire puntatori multipli:
 - o Le entrate restano a lunghezza fissa;
 - o Il secondo campo $P(i)$ è un puntatore ad un blocco di puntatori a record.
 - o Ogni puntatore nel secondo blocco referencia un record nel file dati con valore del campo di indicizzazione $k(i)$.
 - o Se i record con valore $k(i)$ sono tanti da riempire il secondo blocco, si crea una lista a puntatori di blocchi.

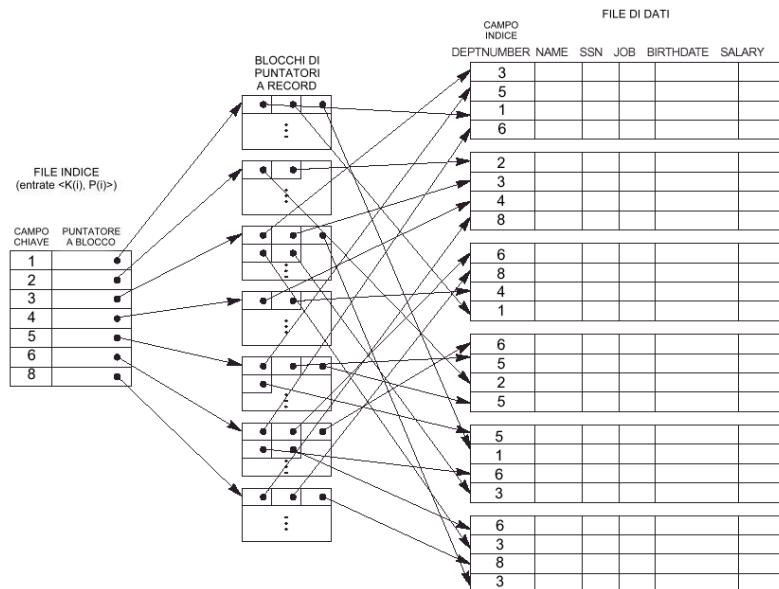


Illustrazione 5: Un indice secondario su un campo non chiave, implementato con un livello di in direzione.

Il retrieval via indice richiede un accesso a un blocco in più, a causa del livello extra. Il retrieval su condizioni di selezione complesse può essere gestito direttamente con i puntatori senza dover operare sul file. L'inserimento di nuovi record è immediato.

Osservazione: Un indice secondario fornisce un ordinamento logico sui record rispetto al campo di indicizzazione.

Riepilogo

	Campo Ordering	Campo Non Ordering
Campo Chiave	Indice Primario	Indice Secondario (chiave)
Campo Non Chiave	Indice clustering	Indice Secondario (non chiave)

	Numero di entry di 1° livello	Denso o non denso	Ancoraggio dei blocchi del file dati
Primario	Numero dei blocchi del file dati	Non denso	Si
Clustering	Numero di valori distinti nel campo clustering	Non denso	Si/No: Si se ogni valore distinto del campo ordering inizia un nuovo blocco; no altrimenti.
Secondario (chiave)	Numero dei record del file dati	Denso	No
Secondario (non chiave)	Num. Record del file dati (caso 1) oppure num. Valori distinti del campo indexing (casi 2 e 3)	Denso o non denso	No

Indici Multilivello

L'idea di base degli indici multilivello è di ridurre la dimensione dell'indice per velocizzare la ricerca binaria. Per fare ciò, si creano vari livelli di indici, ognuno con poche entry.

Definiamo fan-out (fo) di un indice multilivello il blocking factor bfr_i dell'indice: Approssimativamente una ricerca impiegherà $\log_{fo} b_i$ accessi a blocco (per un indice con b_i entry).

La struttura di un indice multilivello ha al primo livello (o livello di base) un file indice ordinato con un valore distinto per ogni $k(i)$. Al secondo livello c'è un indice primario sul primo livello. Utilizzando le block anchor, il numero di entry di 2° livello è uguale al numero di blocchi del 1° livello.

È possibile continuare questo processo, creando un indice per il secondo livello, etc. È necessario creare un nuovo livello solo se il precedente occupa più di un blocco su disco. L'indice di livello superiore è detto **top index**.

Lo schema multilivello può essere usato su qualsiasi tipo di indice, purché l'indice del primo livello abbia valori distinti per $k(i)$ ed entrate a lunghezza fissa.

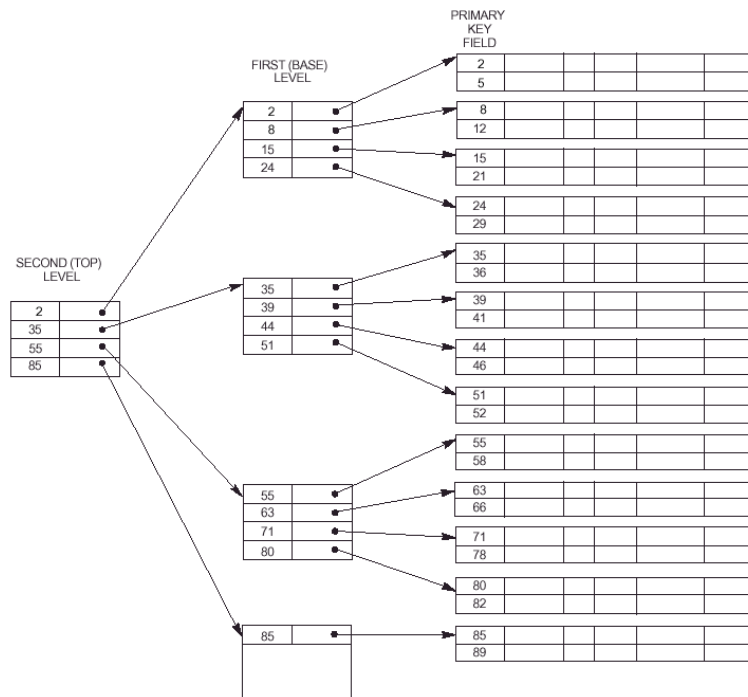


Illustrazione 6: un indice a due livelli

I livelli hanno lo stesso blocking factor del primo livello, poiché tutte le entry hanno la stessa dimensione. Se il primo livello ha r_1 entrate e blocking factor $bfr_1=fo$, allora richiederà $(r_1/fo)_{sup}$ blocchi, che è anche il numero di entry del secondo livello. Ciascun livello riduce il numero di entrate del livello precedente di un fattore fo , quindi vale la formula:

$$(r_1/(fo)^t) \geq 1 \rightarrow t = (\log_{fo} r_1)_{sup}$$

Esempio. Supponiamo di voler convertire l'indice denso secondario dell'esempio visto in precedenza, costruito su un file di $r = 30.000$ record a lunghezza fissa:

- $bfr_1 (= fo) = 68$
- Numero blocchi 1° livello $b_1 = 442 (=30.000/68)$
- Numero blocchi 2° livello $b_2 = (b_1 / fo)_{sup} = 7$
- Numero blocchi 3° livello $b_3 = (b_2 / fo)_{sup} = 1$

Quindi il livello top è il terzo ($t=3$) ed il numero di accessi a blocco è $t + 1 = 4$ contro i 10 accessi

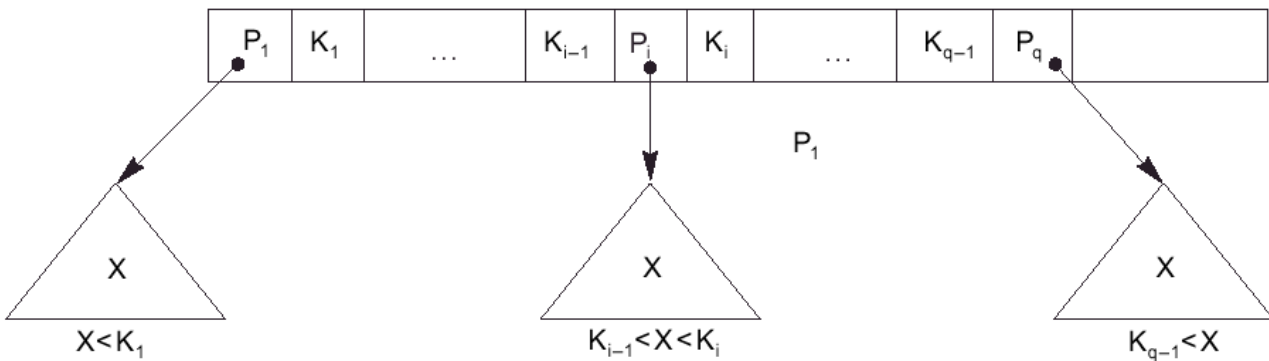
dell'indice secondario (+1 per l'accesso al file dati).

Indici multilivello dinamici con B-Tree e B⁺-Tree

Un albero di ricerca è un particolare tipo di struttura dati ad albero, utilizzato per recuperare un record dato il valore di uno dei suoi campi. Gli indici multilivello sono una variante degli alberi di ricerca: il valore del campo indice in ogni nodo guida verso il prossimo nodo, finché non si raggiunge il blocco del data file contenente il record cercato.

Dato un albero di ricerca di *ordine p*:

- ogni nodo contiene al più $p - 1$ valori di ricerca e p puntatori, nell'ordine $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$, con $q \leq p$
- ogni P_i è un puntatore ad un nodo figlio (o un puntatore nullo).
- ogni K_i è un valore di ricerca da qualche insieme ordinato (si suppone che tutti i valori di ricerca siano distinti).



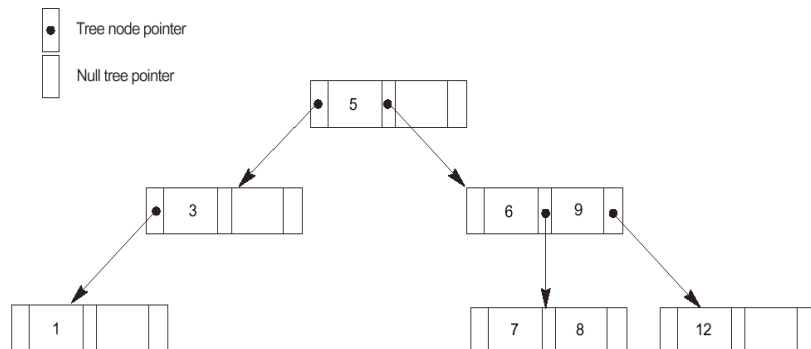
In un albero di ricerca, devono valere sempre due vincoli:

1. All'interno di ogni nodo, $K_1 < K_2 < \dots < K_{q-1}$
2. Per ogni valore X nel sottoalbero puntato da P si deve avere $K_{i-1} < X < K_i$ per $1 < i < q$
 - $X < K_i$ (per $i = 1$)
 - $K_{i-1} < X$ (per $i = q$)

Come vengono usati gli alberi di ricerca per recuperare un record memorizzato in un file su disco? I valori nell'albero sono i valori di un campo del record, detto campo di ricerca. Ad ogni valore nell'albero è associato un puntatore o al record nel file dati con tale valore o al disk block contenente il record.

Un albero di ricerca può essere esso stesso memorizzato su disco, assegnando ogni nodo ad un blocco su disco.

Esempio. Un albero di ricerca di ordine $p = 3$:



Gli algoritmi per inserire e cancellare record dall'albero (mantenendo sempre i due vincoli) non garantiscono che l'albero rimanga bilanciato: mantenere l'albero di ricerca bilanciato è fondamentale per limitare il numero di accessi a blocco, poiché l'altezza di un albero bilanciato è sempre logaritmica sul numero di nodi.

La cancellazione negli alberi di ricerca potrebbe lasciare dei nodi quasi vuoti, causando uno spreco di spazio ed un aumento del numero di livelli.

B-Tree

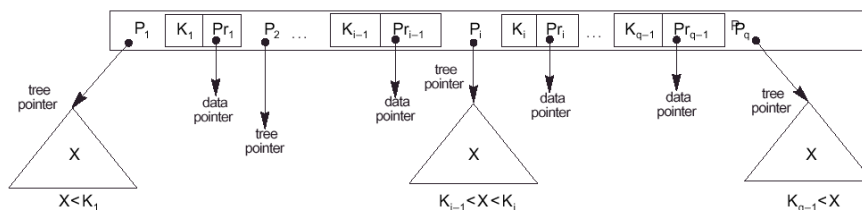
I B-Tree ed i B⁺-Tree sono alberi di ricerca bilanciati, progettati per ottimizzare operazioni su dischi magnetici o qualunque altro tipo di memoria secondaria ad accesso diretto. Alcuni vincoli posti sulla struttura assicurano che:

1. L'albero è sempre bilanciato.
2. Lo spreco di spazio nei nodi è abbastanza limitato.

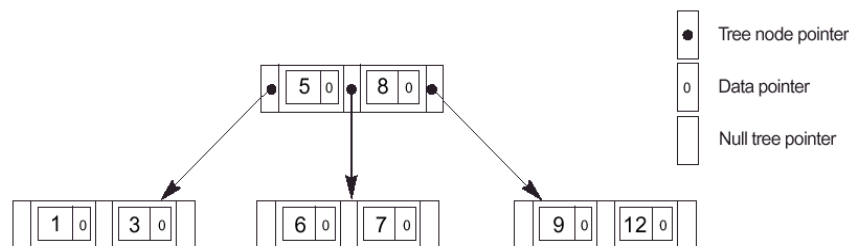
Dato un **B-Tree di ordine p**,

1. Ogni nodo interno ha la forma $\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, P_{q-1}, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$, con $q \leq p$
 - ogni P_i è un puntatore ad un albero (un altro nodo nel B-Tree).
 - ogni Pr_i è un data pointer (puntatore al record con valore K_i del campo chiave di ricerca).
2. All'interno di ogni nodo deve valere che $K_1 < K_2 < \dots < K_{q-1}$.
3. Per ogni valore X del campo chiave di ricerca nel sottoalbero puntato da P_i vale:
 - $K_{i-1} < X < K_i$ (per $1 < i < q$),
 - $X < K_i$ (per $i = 1$),
 - $K_{i-1} < X$ (per $i = q$)
4. Ogni nodo ha al più p puntatori ad albero.
5. Ogni nodo, tranne la radice ed i nodi foglia, ha almeno $(p/2)_{sup}$ puntatori ad albero. Il nodo radice ne ha almeno 2, a meno che non sia l'unico nodo nell'albero.
6. Un nodo con q ($q < p$) puntatori ad albero ha $q-1$ valori del campo chiave di ricerca (e quindi $q-1$ data pointer).
7. Tutti i nodi foglia sono allo stesso livello. I nodi foglia hanno la stessa struttura dei nodi interni, tranne per i puntatori ad albero, che sono nulli.

Esempio di nodo di B-Tree.



Esempio di B-Tree.



Un B-Tree di ordine $p = 3$.

Da osservare l'unicità del valore di ricerca poiché assumiamo che è una struttura di accesso sul campo chiave. Se il campo fosse non chiave si dovrebbe far puntare Pr ad una lista a puntatori aggiungendo un livello extra.

Costruzione di un B-Tree

Inizialmente un B-Tree ha un unico nodo radice (che è anche foglia).

Se la radice contiene p-1 valori di ricerca e si tenta di inserire un'altra entrata nell'albero, il nodo viene scisso in due nodi a livello 1. Nella radice resta solo il valore mediano mentre gli altri valori vengono distribuiti equamente tra i due nuovi nodi.

Se un nodo non radice è pieno e vi si inserisce una nuova entrata, quel nodo è scisso in due nodi allo stesso livello e l'entrata mediana è spostata al nodo padre insieme a due puntatori ai nodi risultanti dalla scissione.

Se anche il nodo padre è pieno, viene scisso anch'esso. La scissione si può propagare fino alla radice, aggiungendo un nuovo livello ogni volta che la radice viene scissa.

Se la cancellazione di un valore fa sì che un nodo diventi pieno per meno della metà, esso è combinato con suoi vicini. Ciò può propagarsi fino alla radice, riducendo il numero di livelli dell'albero.

Osservazione: È stato mostrato tramite analisi e simulazione che, dopo numerosi inserimenti e cancellazioni casuali su un B-Tree, quando il numero di valori nell'albero si stabilizza, i nodi sono pieni circa al 69%: la scissione e la combinazione dei nodi accadranno solo raramente, rendendo inserimento e cancellazione molto efficienti.

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:

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

Scegliamo p = 23, poiché in ogni nodo ci potrebbe essere la necessità di memorizzare informazioni addizionali, quali il numero di entrate q o 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-ordering e chiave. Supponiamo che ogni nodo sia pieno per il 69%. Ogni nodo avrà in media $p \cdot 0.69 = 23 \cdot 0.69 =$ 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	-

Per ogni livello:

$$\#entrate\ livello\ i-mo = [\#puntatori\ livello\ (i-1)-mo] \cdot 15$$

Per le date dimensioni di blocco, puntatore e campo chiave di ricerca, un B-tree a 3 livelli contiene fino a $3840 + 240 + 15 = 4095$ entrate. Un B-tree a 4 livelli contiene fino a $4095 + 61440 = 65535$ entrate.

B⁺-Tree

I B⁺-Tree sono usati nella maggior parte delle implementazioni di indici multilivello dinamici.

I puntatori ai dati sono memorizzati solo nei nodi foglia, che hanno quindi una struttura diversa da quella dei nodi interni:

- Per un campo di ricerca chiave, i nodi foglia hanno un'entrata per ogni valore del campo, insieme ad un puntatore al record (o al blocco che lo contiene).
- Per un campo non chiave, il puntatore referencia un blocco che contiene puntatori ai record del file dati, creando un ulteriore livello di indirezione.

I nodi foglia sono collegati da puntatori, così da fornire un accesso ordinato ai record sul campo chiave. I nodi foglia sono simili al primo livello di un indice multi-livello.

I nodi interni corrispondono agli altri livelli di un indice multi-livello. Alcuni valori del campo di ricerca dei nodi foglia sono ripetuti in nodi interni del B⁺-Tree per guidare la ricerca.

Confronto tra B-Tree e B⁺Tree.

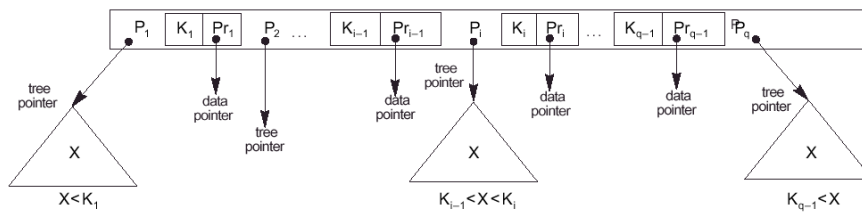


Illustrazione 7: nodo interno di un B-Tree

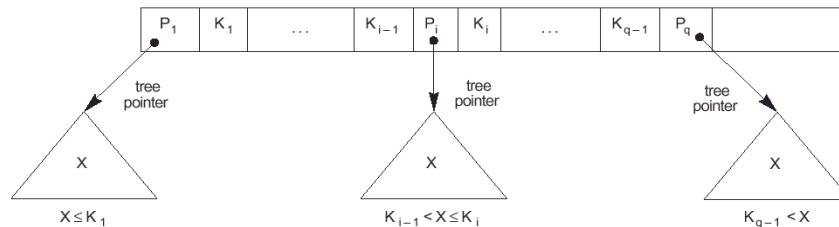


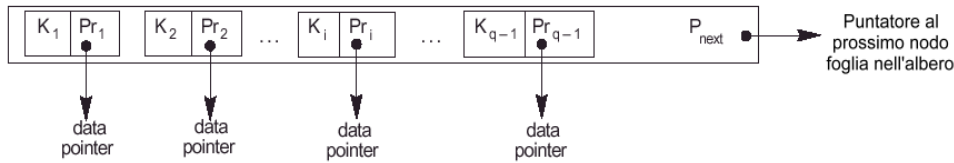
Illustrazione 8: struttura di un nodo di un B+tree

Struttura dei nodi interni di un B⁺Tree

1. Ogni nodo interno ha la forma $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$, con $q \leq p$ ogni P_i è un puntatore ad albero.
2. In ogni nodo interno si ha $K_1 < K_2 < \dots < K_{q-1}$.
3. Per ogni valore X del campo di ricerca nel sottoalbero puntato da P_i vale che:
 $K_{i-1} < X \leq K_i$ (per $1 < i < q$),
 $X \leq K_i$ (per $i = 1$),
 $K_{i-1} < X$ (per $i = q$).
4. Ogni nodo interno ha al più p puntatori ad albero.
5. Ogni nodo interno, tranne la radice, ha almeno $(p/2)_{sup}$ puntatori ad albero. Il nodo radice ne ha almeno 2 se è un nodo interno.
6. Un nodo interno con q puntatori, $q \leq p$, ha $q-1$ valori del campo di ricerca.

Struttura di un nodo foglia di un B+Tree

- Ogni nodo foglia ha la forma $\langle \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_{next} \rangle$, con $q \leq p$:



- Ogni Pr_i è un data pointer.
 - P_{next} è un puntatore al prossimo nodo foglia del B⁺-Tree.
- In ogni nodo foglia si ha $K_1 < K_2 < \dots < K_{q-1}$, con $q \leq p$.
 - Ogni Pr_i è un data pointer, che riferenzia:
 - Il record con valore K_i nel campo di ricerca, oppure
 - il blocco del file contenente tale record, oppure
 - un blocco di puntatori a record che riferenziano i record con valore del campo di ricerca K_i , se il campo di ricerca è non chiave.
 - Ogni nodo foglia ha almeno $(p/2)_{sup}$ valori.
 - Tutti i nodi foglia sono allo stesso livello.

Partendo dalla foglia più a sinistra, è possibile visitare tutti i nodi foglia come una lista a puntatori, usando P_{next} . Ciò fornisce un accesso ai record di dati ordinato sul campo di indicizzazione.

Se il campo di ricerca non è chiave, è necessario un ulteriore livello di indizione, simile a quello della struttura ad indice secondario con campo non chiave.

Un nodo interno di un B⁺-Tree contiene più entrate di un nodo interno di un B-Tree, poiché nel B⁺-Tree tali nodi hanno solo valori di ricerca e puntatori ad albero. Di conseguenza, a parità di dimensione di blocco, l'ordine p sarà maggiore per un B⁺-Tree che per un B-Tree.

Poiché le strutture di nodi interni e di nodi foglia sono diverse, l'ordine p può essere diverso nei due tipi di nodo: P_{leaf} indica il numero massimo di data pointers in un nodo foglia.

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:

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

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 entrate in ogni nodo interno.

L'ordine P_{leaf} dei nodi foglia è:

$$(P_{leaf} * (Pr + V)) + P \text{ (per } P_{next}) \leq B \rightarrow$$

$$(P_{leaf} * (7 + 9)) + 6 \leq 512 \rightarrow$$

$$(P_{leaf} * 16) \leq 506$$

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

Esempio 2. 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 (fo 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) entry	12167 ptr
livello foglie:	12167 nodi	255507 (21 * 12167) puntatori a record	

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

Inserimento

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'entrata in un nodo foglia pieno, il nodo va in overflow e deve essere scisso:

- le prime $j = (p_{leaf} + 1)/2_{sup}$ entrate 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 $j = ((p+1)/2)_{inf}$) sono mantenute nel nodo originario, mentre il j-mo valore di ricerca è spostato al padre, non replicato.
- Il nuovo nodo interno conterrà le entrate 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.

Per un esempio di inserimento e cancellazione in un B+Tree, vedasi da slide 71 della lez. 5.