

Strutture ausiliarie di accesso / 24-03

I record possono essere organizzati

→ casualmente

→ rispetto ad attributi

→ rispetto un hash: valori simili o uguali su alcune proprietà vengono memorizzati vicini

Questo però non impedisce l'accesso a blocchi inutili (= rallentamenti)
Attraverso strutture ausiliarie posso ovviare a questi problemi

Ad esempio posso avere una struttura (relativa a una o più colonne) con puntatori a tuple

→ delle INDICE: devono comunque scandire il file ma presto è più sintetico e si focalizza sulle informazioni di mio interesse (struttura più compatta)

Esempio

Indice con chiave dataNol, ordinato su dataNol

Recuperare tutti i noleggi effettuati dopo il '15-Mar-2006'

File ordinato rispetto a dataNol

Indice con chiave di ricerca dataNol

Recuperare tutti i noleggi del video con collocazione 1122

01-Mar-2006
01-Mar-2006
02-Mar-2006
02-Mar-2006
...
15-Mar-2006
15-Mar-2006
15-Mar-2006
...

1111
1115
...
1122
1122
...

File ordinato su dataNol

colloc	dataNol	codCli	dataRest
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1116	21-Mar-2006	6610	?
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?

File non ordinato rispetto a colloc
Indice con chiave di ricerca colloc

Indice con chiave

dataNol

ordinato su dataNol

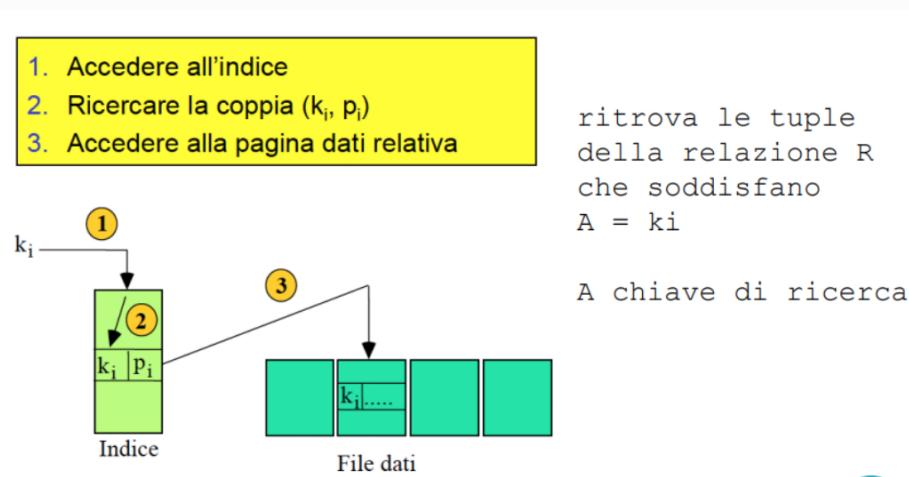
Il file puo' anche non essere ordinato ma accedo comunque a una struttura piu' sintetica del file completo

Un indice e' un **MULTI-INSIERE** d: coppie (k_i, r_i)

$\rightarrow k_i$ e' un valore x lo chiave di ricerca su cui l'indice e' costituito
(es. un valore di telefono)

$\rightarrow r_i$ e' puntatore al record con chiave di ricerca k_i

Il vantaggio e' che l'indice occupa un minore spazio del file di dati



Si assume che le coppie siano ordinate per r_i

\rightarrow le interrogazioni non vengono rallentate dall'uso di indici (lettura ok)

\rightarrow ma a seguito di aggiornamenti deve aggiornare anche i file

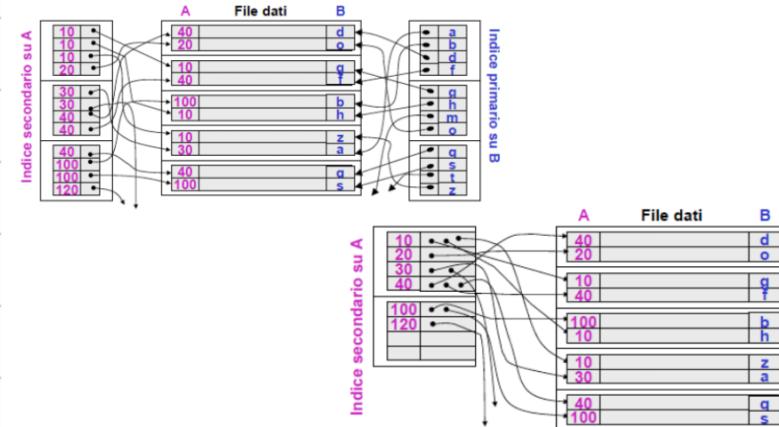
di indice (costoso: deve cancellare un puntatore in ogni indice
alla rimozione di una tupla) (modifica costosa)

Gli indici si distinguono in primari e secondari

Primario: la chiave di ricerca è CHIAVE della relazione, non ha stessa valori di v_i per più coppie

Secondario: continuo di primario

Questa divisione è utile per poter associare una lista di puntatori e record differenti che condividono stesso valore v_i :
→ indice più compatto, minor dimensione occupata



Vi sono diversi tipi di indici, a seconda di come organizziamo le coppie

- Indici ORDINATI (detti ad albero): le coppie (u_i, v_i) sono inserite esplicitamente, salvate in un file e ordinate su u_i .
- Indici HASH: le coppie non vengono salvate in un file ma vengono calcolate tramite una funzione di hash. In tale che $h(u_i) = v_i$

Indici ordinati

Se l'indice e' piccolo puo' essere tenuto in memoria principale

Spesso puo' esser tenuto sul disco, con molti trasferimenti di blocchi.

→ si utilizza una struttura multilivello x accedere piu' velocemente alle coppie dell'indice

L'indice assume una struttura ad albero in cui ogni nodo e' un blocco del disco. Il file che memorizza gli indici costituisce l'organizzazione secondaria dei dati.

L'albero e' bilanciato (segue BST) :

→ bilanciamento : indice bilanciato rispetto ai blocchi

→ occupazione minima di memoria

→ efficienza delle operazioni di aggiornamento

La struttura piu' comune e' β^+ -tree

→ un nodo = un blocco

→ Ricerca, inserimento, cancellazione al peggio costo h , solitamente $\log \alpha$ al numero di valori distinti N delle chiavi di ricerca $\Theta(\log N)$

→ un parametro m si dice pronto in modo e' se

ciò determina le dimensioni dei blocchi e il numero massimo $m-1$ di elementi memorizzabili.

→ garantisce almeno il 50% dei blocchi occupati

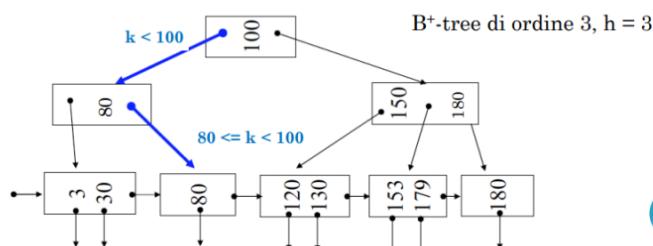
Le copie sono memorizzate in nodi (blocki) foglie

→ lo ricerca deve individuare la foglia

I nodi interni memorizzano il cammino

B⁺-TREE - DEFINIZIONE FORMALE

- Un B⁺-albero di ordine m ($m \geq 3$) è un albero bilanciato che soddisfa le seguenti proprietà:
 - ogni nodo contiene al più $m-1$ elementi
 - ogni nodo, tranne la radice, contiene almeno $\lceil m/2 \rceil - 1$ elementi, la radice può contenere anche un solo elemento
 - ogni nodo non foglia contenente j elementi ha $j+1$ figli



Note: occedo solo 3 blocki
e non (ollo peggio), nel caso
80 fosse l'ultimo di una lista
8 blocki

Le operazioni di inserimento e cancellazione prendono un'operazione di ricerca, e pertanto a livello delle foglie. Cost. lineare h (oltre) se devo ripristinare i vincoli di memorizzazione secondo m.

Un albero alternativo è il B-tree, sempre un livello, i puntatori puo' passare contenuti anche nei nodi interni.

Inoltre un indice ed un file è detto clusterizzato se il file di dati è ordinato rispetto alle chiavi di ricerca, altrimenti detto non clusterizzato.

Un file di dati ordinato presuppone l'esistenza di un indice
ed albero clusterizzato, quindi (perché ordinato secondo un
solo criterio) puo' esistere al massimo un indice clusterizzato
x file di dati.

Se l'indice e' clusterato i puntatori al record "non si intrecciano"
 → anche se non c'è primarietà sufficiente associare un solo
 puntatore al record di due v_i , perché sicuramente, se
 esistenti, tuple di uguali valori v_i si troveranno in posizioni
 consecutive

L': dire pu' essere su

- singolo attributo (classe di ricerca costituita da un solo attributo)
- multi attributo (" " " de più attributi)
 - (es. (`codice`, `dataN°1`))

Lo importa l'ordine con cui voglio ordinare le tuple

→ sviluppo più prece, ma aggiornamenti più frequenti e di maggior

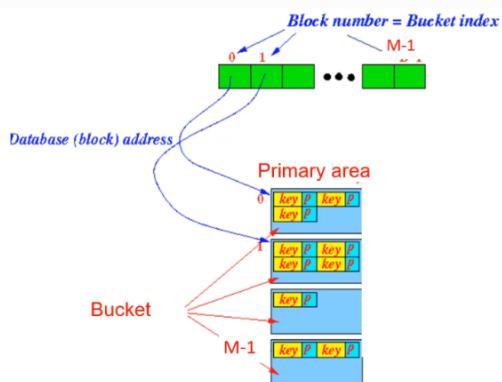
L'unico indice, più attivo:

↳ besteht aus einem Objekt

Judici hash

Lo sviluppo degli indici ed elenco e memorizzare gli indici
 (= occupa memoria), l'host risolve coltando gli indici
 (solvendo solo le 'associazioni' fra i riferimenti)

L'idea è memorizzare dati in ^mblock (block vicini) dove un valore di funzione di hash corrisponde a un indirizzo di memoria.



FILE DEI DATI
 e l'organizzazione primaria
 ad essere strutturata
 come file hash
 (organizzato secondo una f di hash)

Le funzioni hash non mantengono ordine, dunque non supporta
 una ricerca per intervallo

Quando un bucket è pieno, si prende un **overflow pointer**
 che punta a una nuova allocazione di blocchi che occupa una posizione
 non necessariamente vicina al primo, questo perché può
 essere "distante", quindi muoversi "tutto" sul disco (blocchi non contigui)

L'hash permette di evitare di occedere a file, più efficiente; se
 non ha overflow ha un accesso costante ai bucket tramite un
 index (come un array)

Se ha overflow è difficile determinare la complessità, le
 prestazioni non sono determinabili

↳ ciò dipende dal fattore di cercamento (valore t_{av} o ϵ)
 di un bucket

I sistemi DDBS permettono di evitare alle funzioni di hash
 in modo limitato

Le funzioni di hash deve seguire delle proprietà:

- distribuzione uniforme: ogni indirizzo deve essere generato con la stessa probabilità
- distribuzione casuale; no correlazione fra valori delle chiavi e valori di indirizzi generati

Soltamente basate sulla divisione:

$H(u) = u \bmod M$, quindi la chiave viene divisa rispetto ad M , se ne prende il resto che è sicuramente compreso fra 0 e $M-1$ (inclusi)

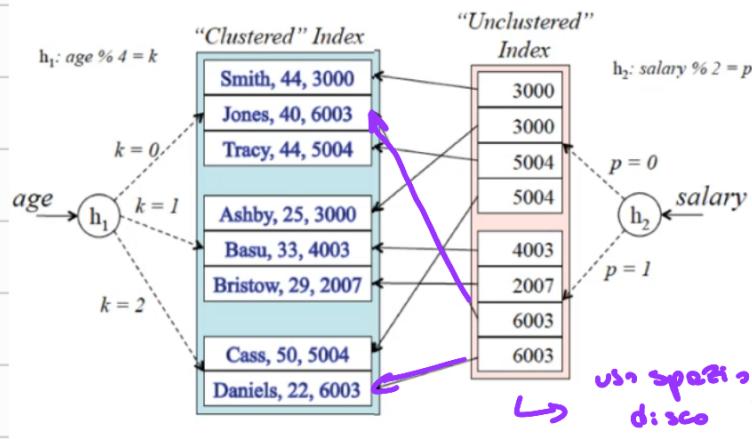
Anche gli indici hash possono essere clusterizzati: i record che condividono lo stesso valore per H : sono memorizzati in posizioni adiacenti nel file di dati, altrimenti detto non clusterizzato
↳ stesso bucket

In file dati tipo hash è sempre associato a un indice hash clusterizzato

(↳ l'organizzazione primaria dei dati corrisponde all'area primaria + record alle overflown).

→ l'organizzazione ecco qui è unica nello tabella (ogni record ha un solo criterio) quindi al più ha un solo indice clusterizzato
file di dati

In un non clusterizzato i record con stesso valore H possono essere memorizzati in bucket diversi, ovvero fare riferimento attraverso a un indice multivello alla struttura in memoria primaria, attraverso puntatori



- = osserviamo che fanno riferimento a bucket diversi

- Indice ed albero più flessibili perché supporta ricerca > intervalli
- Indice hash permette minore numero di accessi ai blocchi (in assenza di overflow) che l'albero invece ha vincolato sempre ad un'altezza dell'albero (perde: record stanno nelle foglie)