

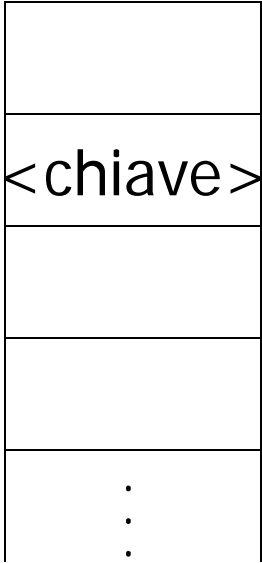
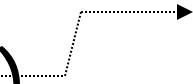
Hashing e indici multidimensionali

Leggere Cap 6 Riguzzi et al. Sistemi
Informativi

Lucidi derivati da quelli di Hector Garcia-Molina

Hashing

chiave \rightarrow $h(\text{chiave})$



Buckets
← (tipicamente 1
blocco di disco)

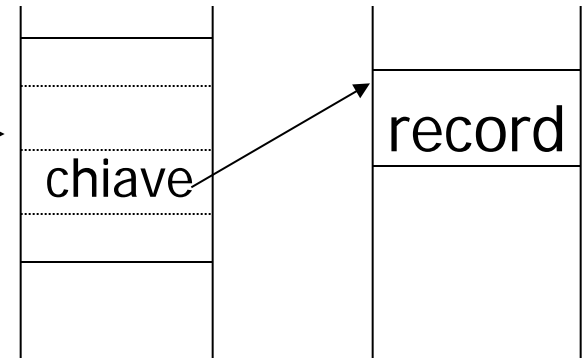
File hash

chiave \rightarrow $h(\text{chiave})$



Struttura hash su file non hash

(2) chiave \rightarrow $h(\text{chiave})$



Indice

La struttura hash diventa una struttura secondaria usata solo per l'accesso ai record

Esempio di funzione hash (1)

- Chiave: numero intero K
- Si abbiano b buckets
- h :
 - Calcola il resto della divisione intera di K per b
 - $h(K) = K \bmod b$

Esempio di funzione hash (2)

- Chiave = 'x₁ x₂ ... x_n' stringa di caratteri di n byte
- Si abbiano b buckets
- h :
 - Calcola somma = $x_1 + x_2 + \dots + x_n$
 - Calcola il resto della divisione intera di somma per b
 - $h(K) = \text{somma} \bmod b$

☒ Vi sono molte funzioni hash, queste sono solo esempi

Buona funzione 

Hash

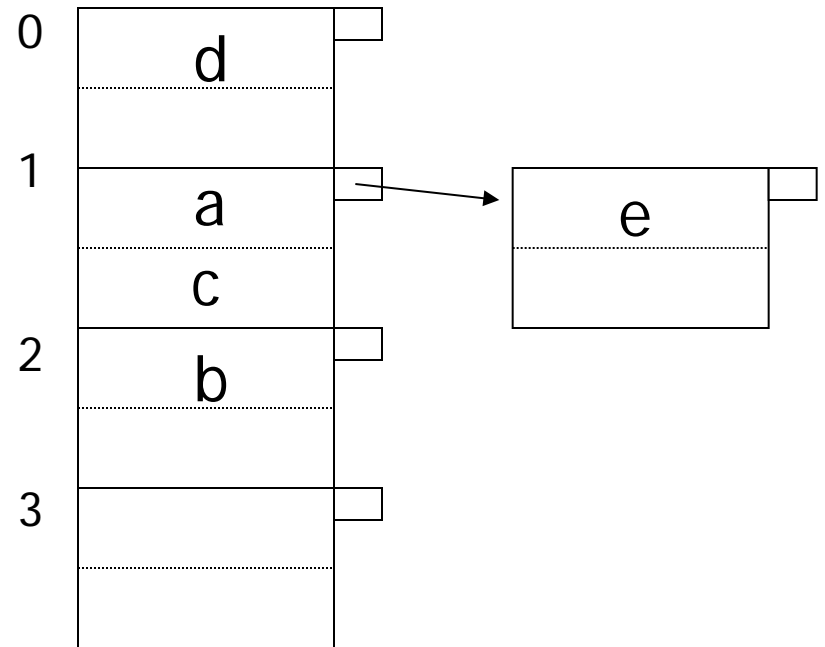
Il numero di chiavi atteso in ogni bucket è lo stesso per ogni bucket

In un bucket:

- Si tengono le chiavi ordinate?
- Si', se il tempo di CPU e' critico e gli inserimenti e le cancellazioni non sono troppo frequenti

Inserimenti

- Se c'è posto nel bucket, non c'è problema, altrimenti si crea un bucket di overflow
 - I bucket di overflow formano una catena



Cancellazioni

- Se si cancella un record da un bucket che aveva dei buckets di overflow, si possono spostare records da questi per ridurre la catena di overflow.

Esempio 2 records/bucket, 4 buckets

Inserisci:

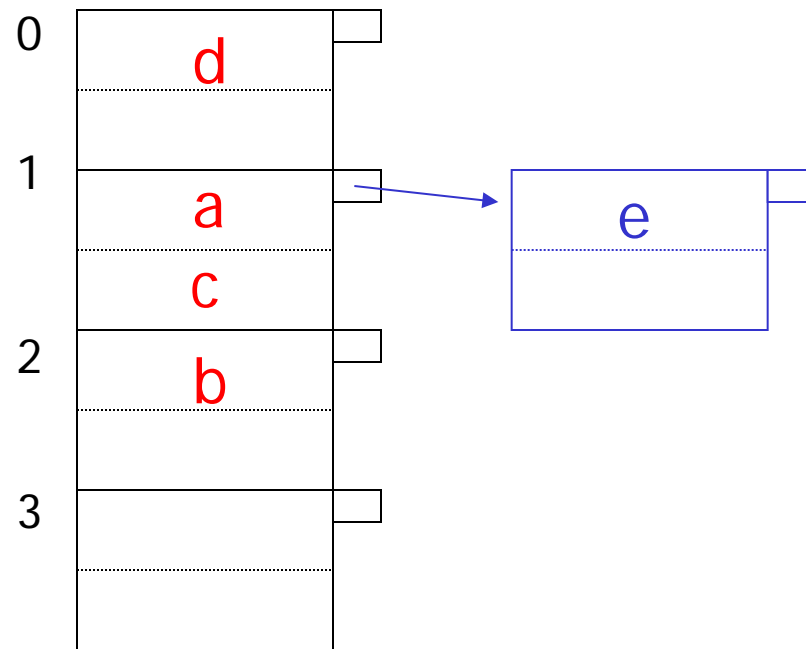
$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

$$h(d) = 0$$

$$h(e) = 1$$



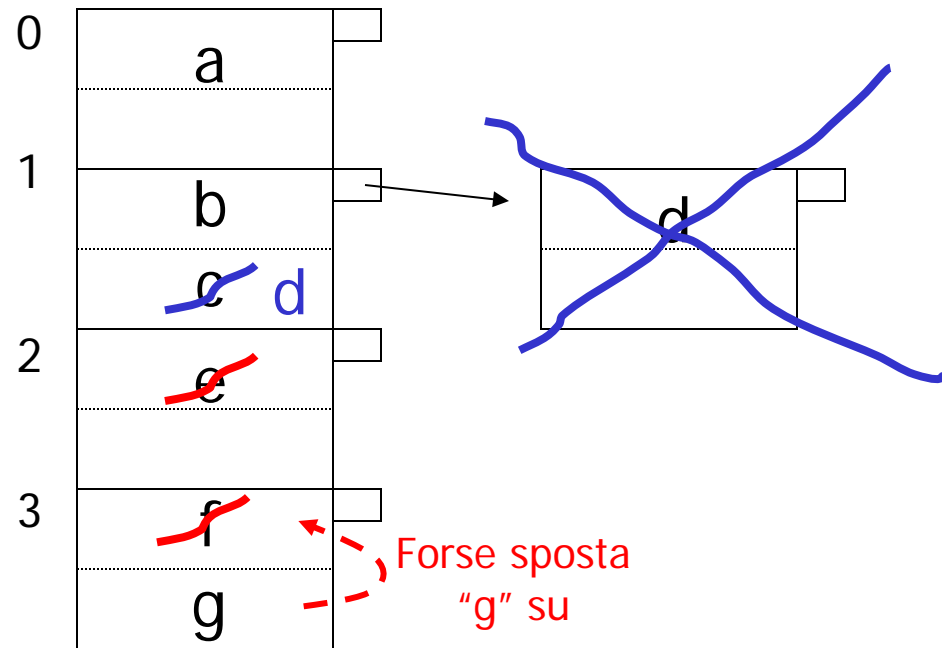
Esempio: cancellazione

Cancella:

e

f

c



Regola empirica:

- Cercare di tenere l'utilizzo tra il 50% e l'80%

$$\text{Utilizzo} = \frac{\# \text{ chiavi usate}}{\# \text{ massimo di chiavi nel file}}$$

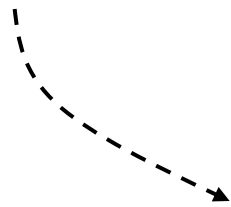
- Se $< 50\%$, si sta sprecando dello spazio
- Se $> 80\%$, l'overflow e' significativo e dipende da quanto e' buona la funzione hash e dal numero di chiavi per bucket

Costo dell'accesso

- Accedere ad un record costa 1 se il file non ha catene di overflow
 - Più economico dell'indice
- Costa di più se ci sono catene di overflow
 - Costa 1 per ogni blocco della catena
- Svantaggi dei file hash rispetto ai B+-trees: non consentono query di range

Come gestire la crescita?

- Overflow e riorganizzazione
- Hashing dinamico



- Estendibile
- Lineare

Hashing estendibile: tre idee

(a) Usa i di b bits generati dalla funzione

hash $\leftarrow b \rightarrow$

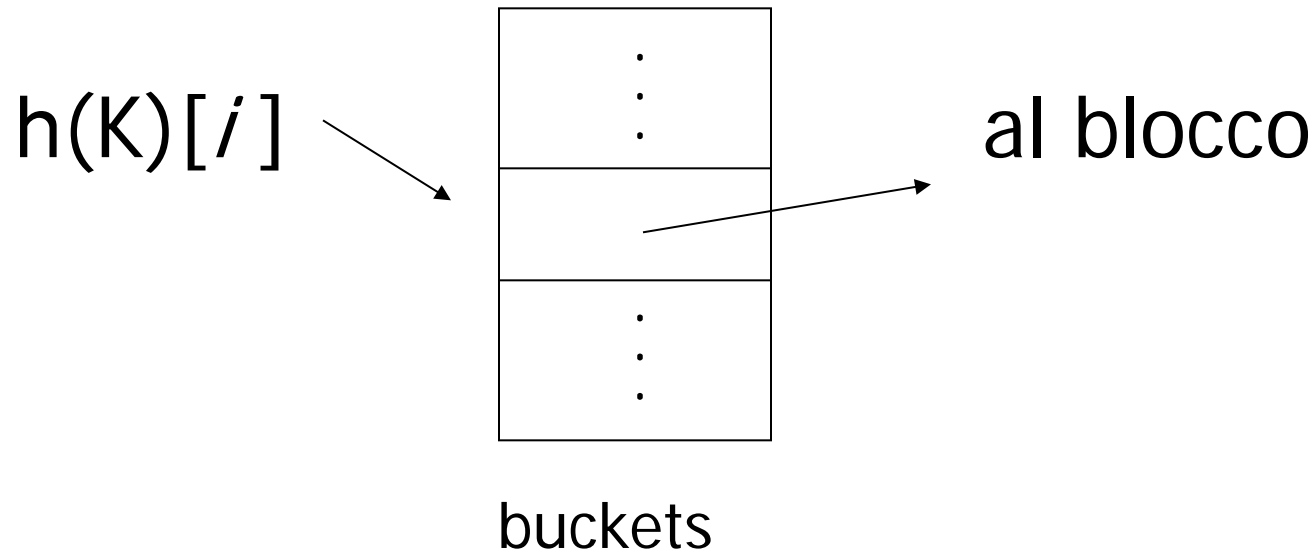
$h(K) \rightarrow$ 00110101



usa $i \rightarrow i$ cresce nel tempo

$h(K)[i]$

(b) Usa un direttorio (aggiunge un livello di indirizzazione)

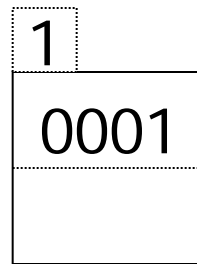


La dimensione del direttorio è sempre pari a 2^i buckets

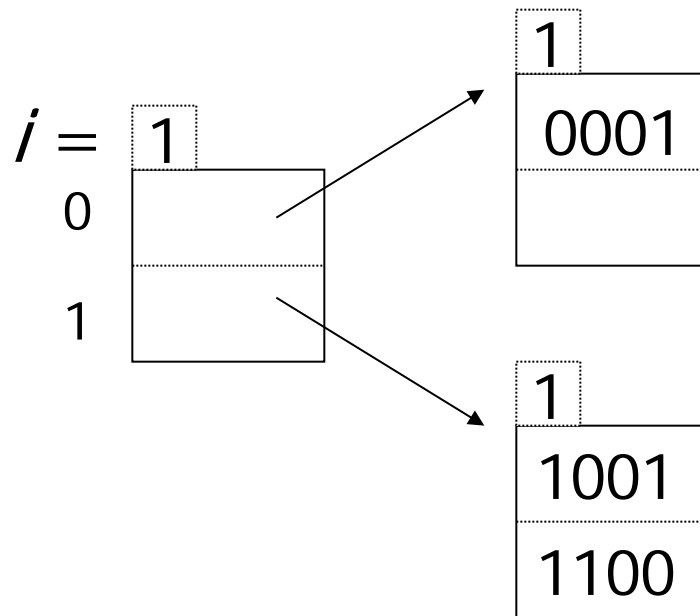
Ogni bucket (entry) contiene solo un puntatore ad un blocco

(c) Ciascun blocco è associato a un numero $j \leq i$ che indica quanti bit della funzione hash sono utilizzati per determinare l'appartenenza dei record al blocco

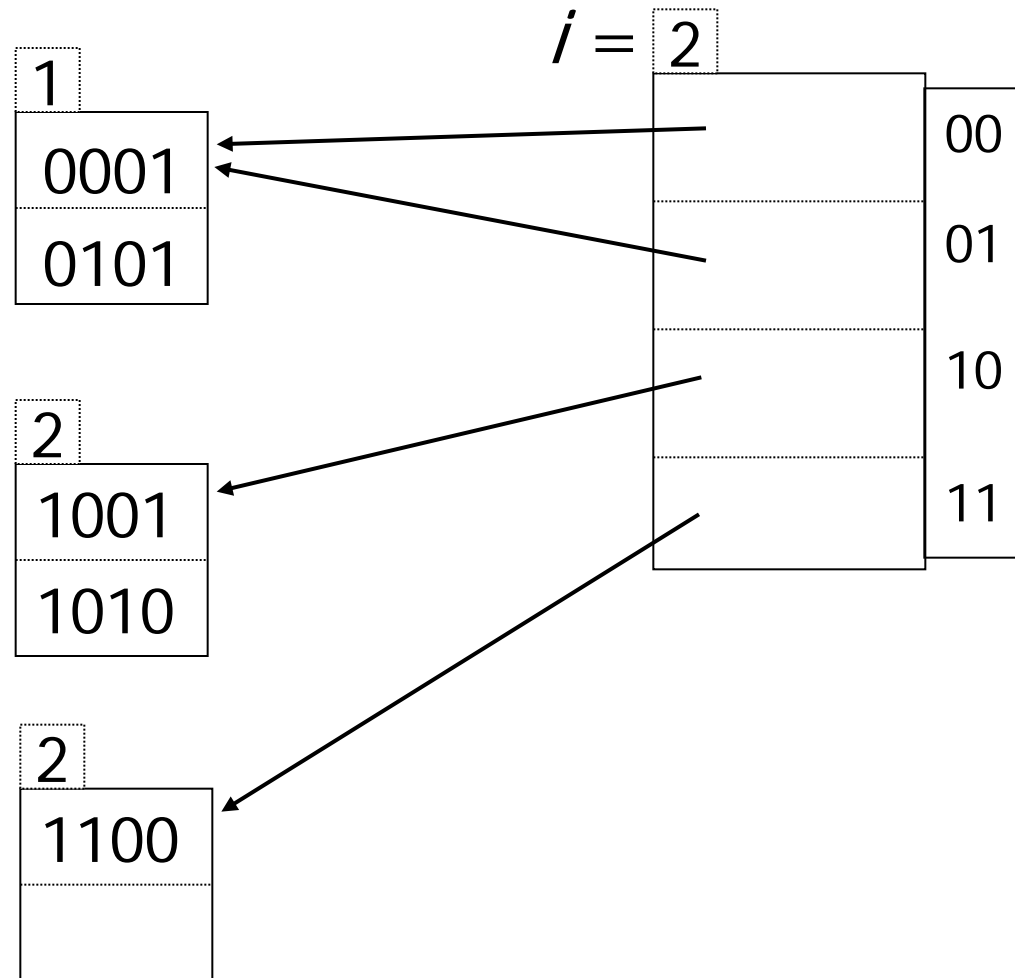
- j Viene memorizzato nell'header del blocco



Esempio



Esempio



Inserimento di K

- Si calcola $h(K)$, si prendono i primi i bit, si va nel direttorio, si va nel blocco dati B
- Se c'è posto in B, si mette il record lì. Se non c'è posto, sia j il numero di bit associato a B
 - Se $j < i$
 - Si divide B in due blocchi B e B'
 - Si distribuiscono i record di B in B e B' sulla base del $(j+1)$ -esimo bit
 - Si associa ai blocchi B e B' il numero $j+1$
 - Si aggiustano i puntatori nel direttorio in modo che le entry che puntavano a B ora puntino a B o a B' a seconda del valore del $(j+1)$ -esimo bit

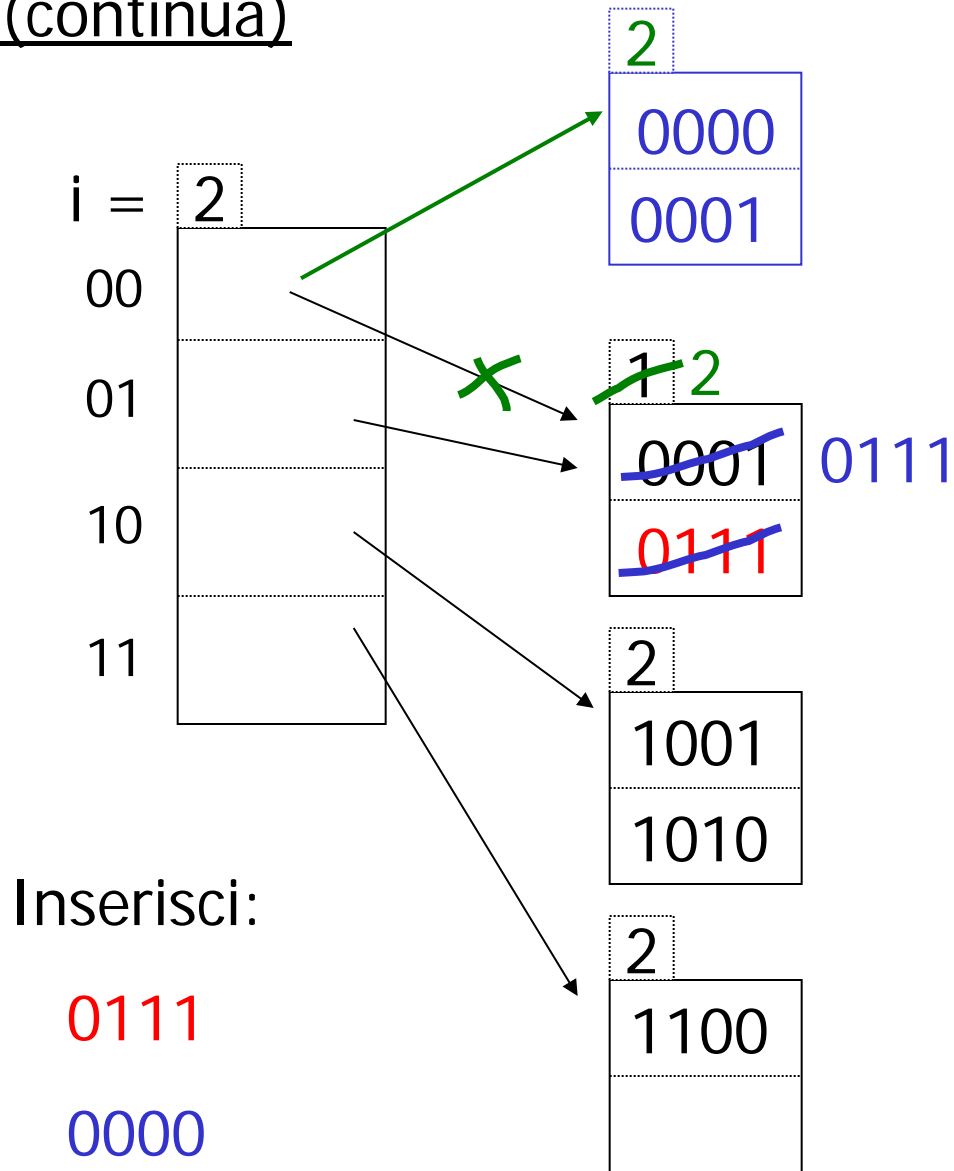
Se necessario, si ripete il processo per $j+1$

Inserimento di K

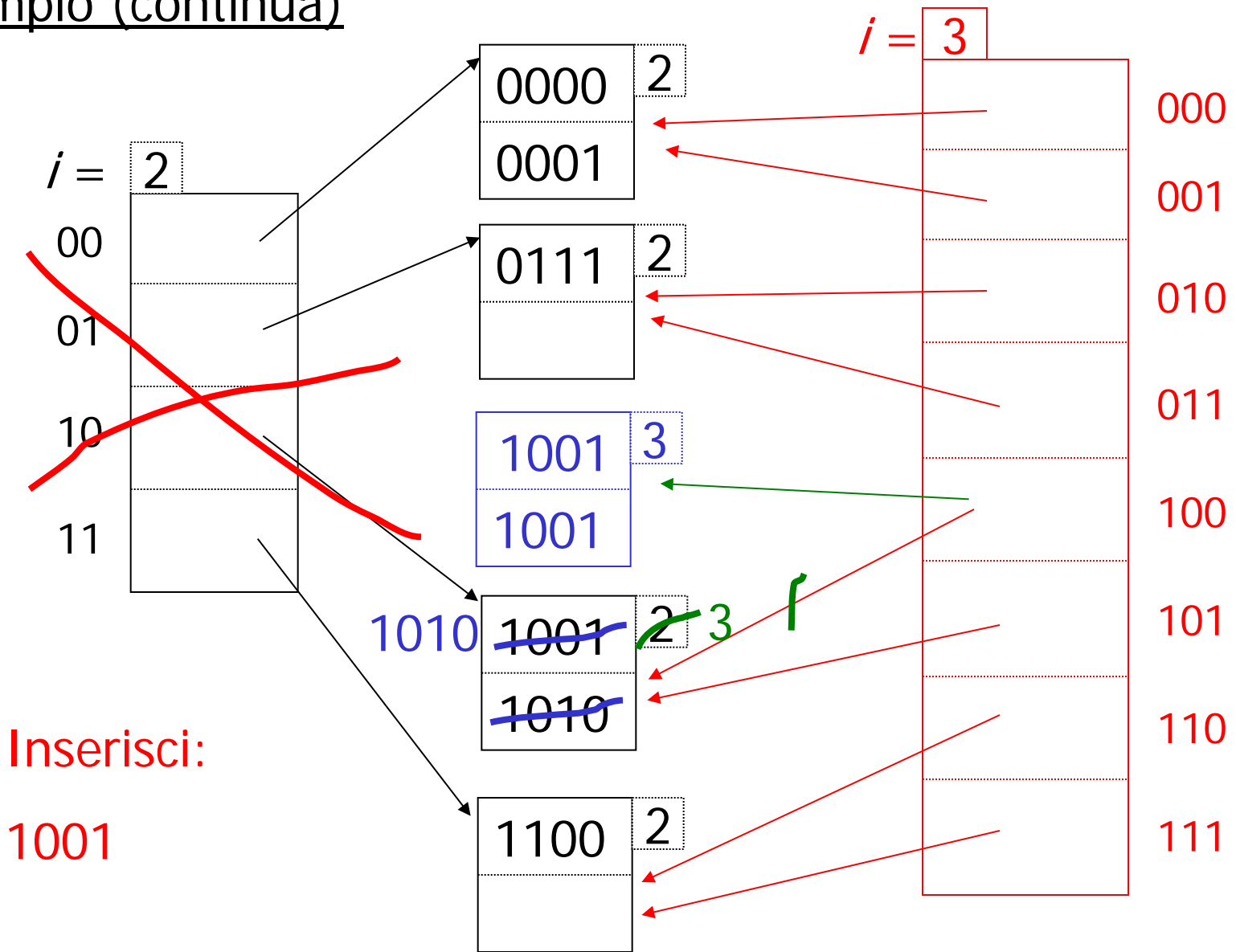
– Se $j=i$

- Si incrementa i di 1
- Si raddoppia la lunghezza del direttorio, che diventa di 2^i entries
- Sia w una sequenza di bit associata ad una delle entry del precedente direttorio
- Nel nuovo direttorio le entries associate a $w0$ e a $w1$ puntano allo stesso blocco a cui puntava w prima
- Dato che ora $j < i$, si ricade nel caso precedente

Esenpio (continua)



Esempio (continua)



Hashing esetendibile: cancellazione

- Due possibilita':
 - Nessuna fusione dei blocchi
 - Fusione dei blocchi e riduzione del direttorio se possibile

(Procedura di insert alla rovescio)

Riassunto

Hashing estendibile

- ⊕ Puo' gestire file crescenti
 - con meno spazio sprecato
 - senza riorganizzazioni complete

- ⊖ Indirezione (non e' un problema se il direttorio sta in memoria centrale)

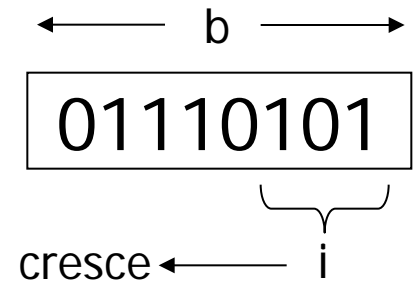
- ⊖ Il direttorio raddoppia di dimensione

(da un momento all'altro puo' non stare in memoria centrale)

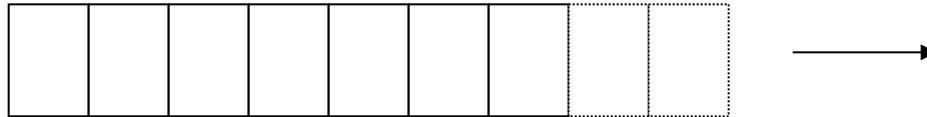
Hashing lineare

- Un altro schema di hashing dinamico

(a) Usa gli i bit della funzione hash di ordine piu' basso ($h(K)[i]$)



(b) Il file cresce linearmente



(c) Si tiene traccia del numero di blocchi allocati n e del numero di record nel file r

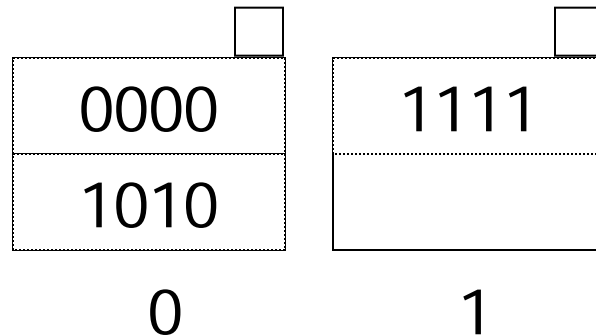
Hashing lineare

- (d) i è pari a $\lceil \log_2 n \rceil$
- (e) Si fissa una soglia per r/n
- (f) Solo quando r/n supera la soglia si incrementa n (e, se necessario, anche i)
- (g) Sono consentiti blocchi di overflow

Esempio

2 record per blocco, $b=4$

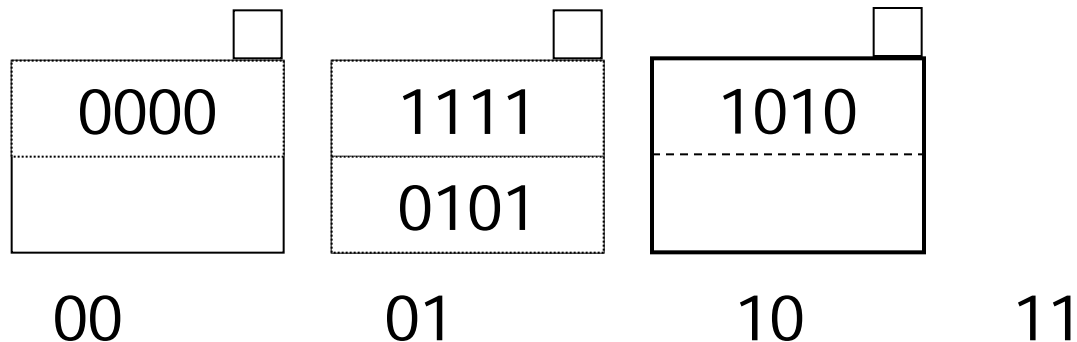
$i=1$, $n=2$, $r=3$, $r/n=1,5$, soglia di $r/n=1,7$



Esempio

2 record per blocco

$i=2$, $n=3$, $r=4$, $r/n=1,33$, soglia di $r/n=1,7$



Hashing lineare

- Inserimento di k :
 - Se $h(k)[i] < n$, allora
 - mettilo al bucket $h(k)[i]$, eventualmente aggiungendo un bucket di overflow
 - altrimenti, guarda al bucket $h(k)[i] - 2^{i-1}$ (metti a 0 il primo bit di $h(k)[i]$)
 - Aumenta r , se $r/n >$ soglia allora
 - Se $n=2^i$ allora incrementa i
 - sia $1a_2..a_i$ la rappr. binaria di n
 - Alloca un blocco per la posizione n
 - Prendi i record del blocco $0a_2..a_i$ che hanno bit i -esimo da destra a 1 e mettili nel blocco $n=1a_2..a_i$
 - Incrementa n

Esempio $b=4$ bits, 2 chiavi/bucket,
 $i = 1, r=3, n=2$, soglia di $r/n=1,7$

Regola

Se $h(k)[i] < n$, allora

guarda al bucket $h(k)[i]$

altrimenti, guarda al bucket $h(k)[i] - 2^{i-1}$

inserisci 0101

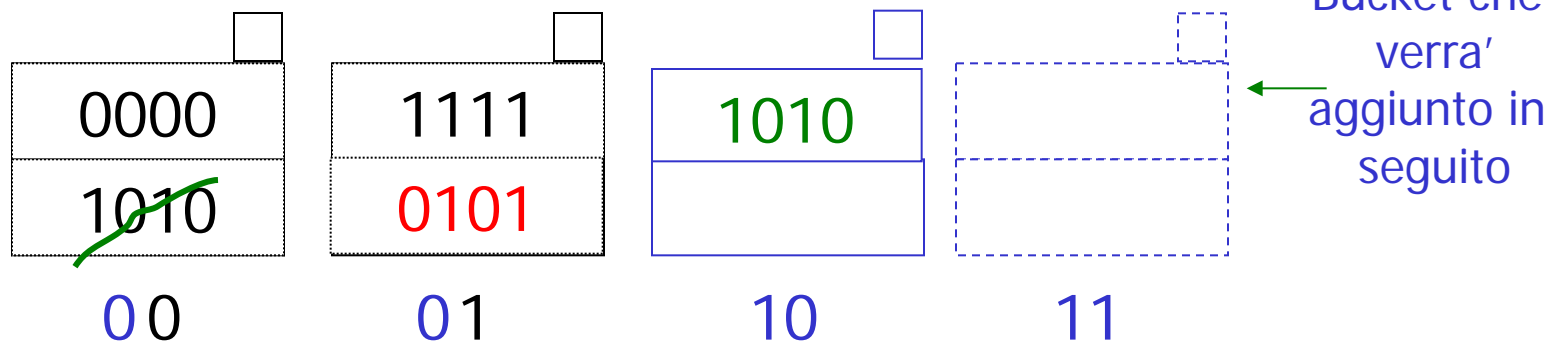
$h(k)[1]=1 < 2$

$r/n=2 > 1.7$ allora: $n=2^i$ allora: incrementa $i \Rightarrow i=2$

Alloca un blocco nella posizione $2=10_2$

Dividi i record tra i blocchi 00 e 10

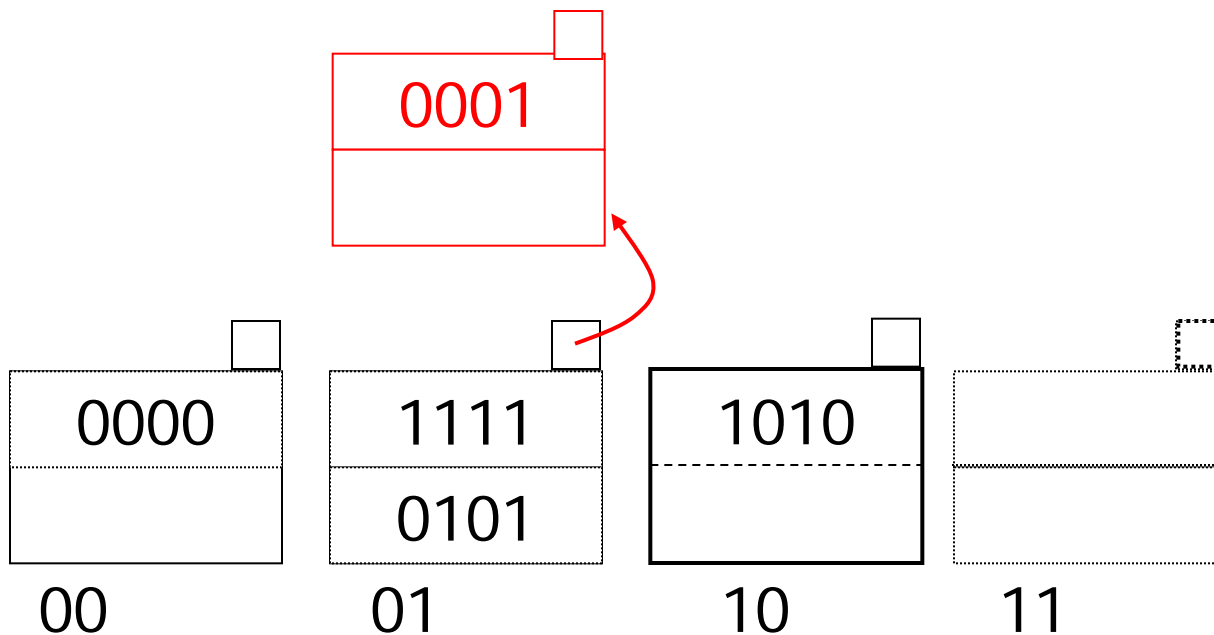
Incrementa n : $n \rightarrow 3$ (numero di blocchi)



Esempio $i = 2, r = 4, n = 3, \text{ soglia } r/n = 1,7$

- inserisci 0001
- $h(k)[2] = 01$
- $01 < 3 \Rightarrow$ bucket 01
- Bucket 01 pieno, si aggiunge un blocco di overflow

$r/n = 5/3 = 1,66 < 1,7 \Rightarrow$ non si incrementa n ,



Esempio $i = 2, r = 5, n = 3, \text{ soglia } r/n = 1,7$

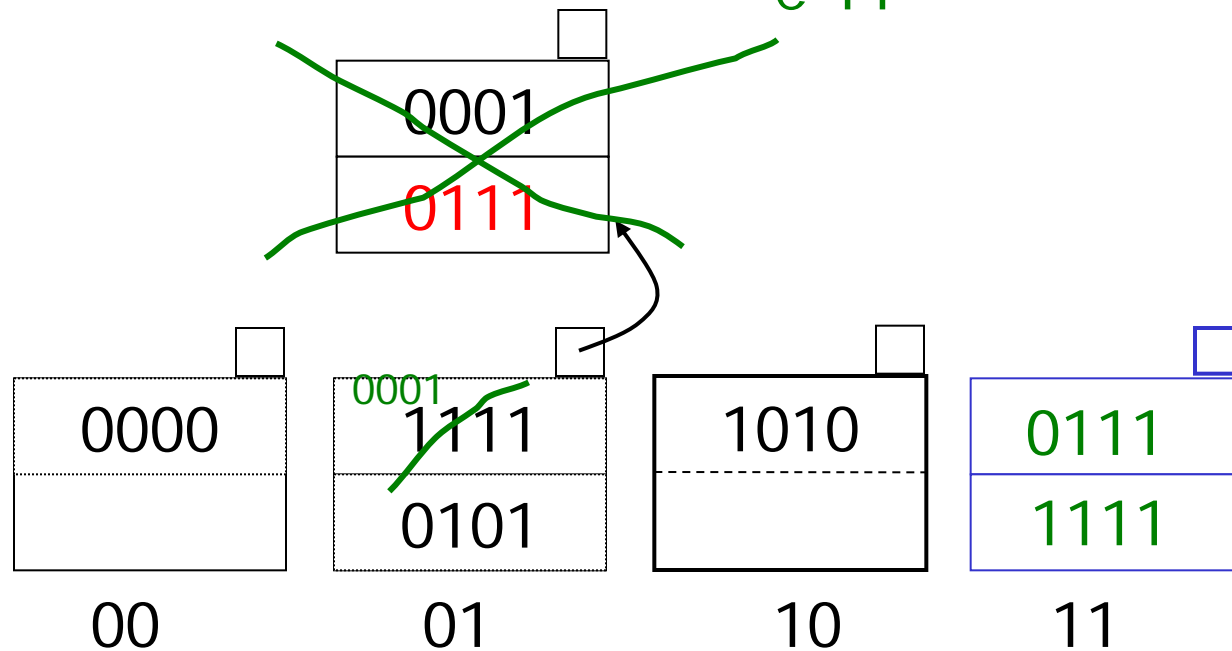
inserisci 0111

$r/n = 6/3 = 2 > 1,7$ allora

$n < 4 \Rightarrow$ non incrementare i

Alloca un nuovo blocco in
posizione 11

Dividi i record di 01 in 01
e 11



Esempio: al prossimo inserimento si supererà la soglia di 1,7 per r/n ($7/4=1,75$) quindi bisognerà aggiungere un blocco. n varrà 5 e i dovrà essere incrementato di 1

Riassunto

Hashing Lineare

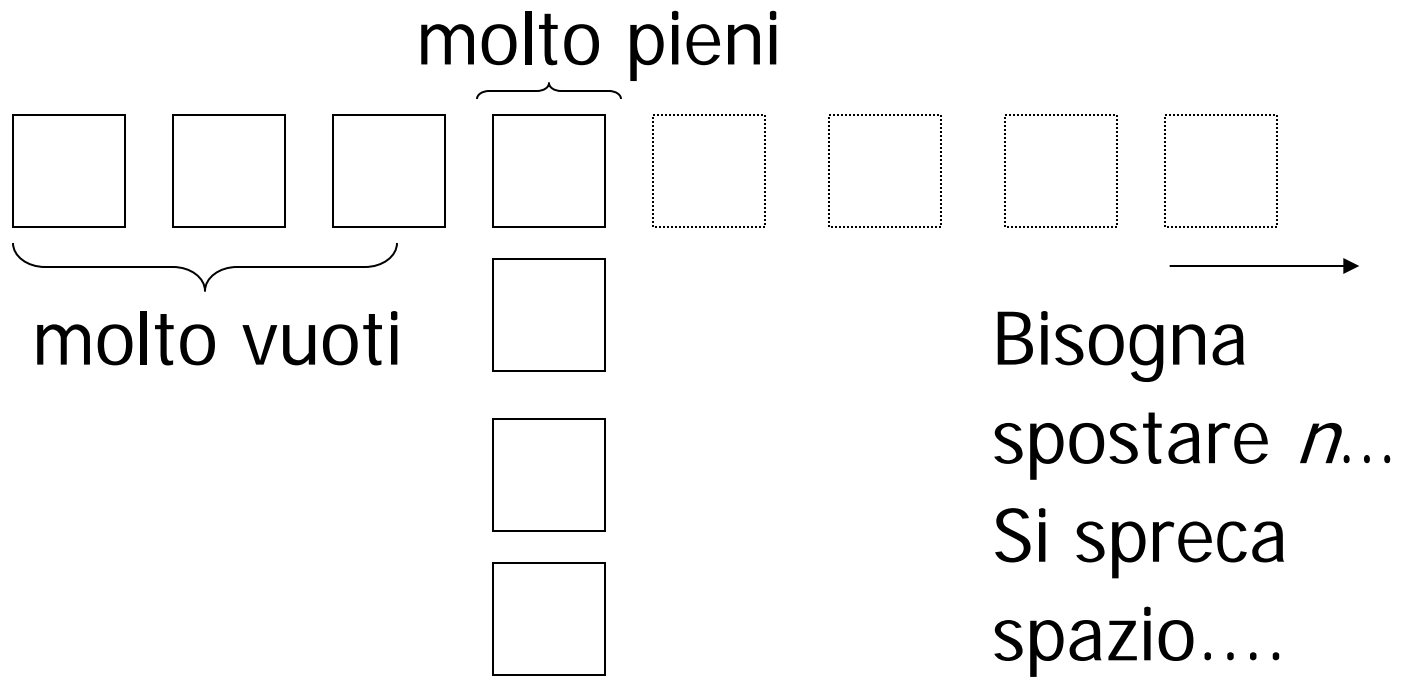
- ⊕ Puo' gestire file che crescono
 - con meno spazio sprecato
 - senza riorganizzazione completa

- ⊕ nessuna indirezione come nell'hashing estendibile

- ⊖ puo' ancora avere catene di overflow

Esempio: CASO CATTIVO

Dato che il bucket che si aggiunge non ha relazione con il record che si aggiunge puo' succedere che:



File hash, osservazioni

- È 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

Possiamo definire anche indici su piu' di un attributo, ad esempio:

```
CREATE INDEX foo ON R(A,B,C)
```

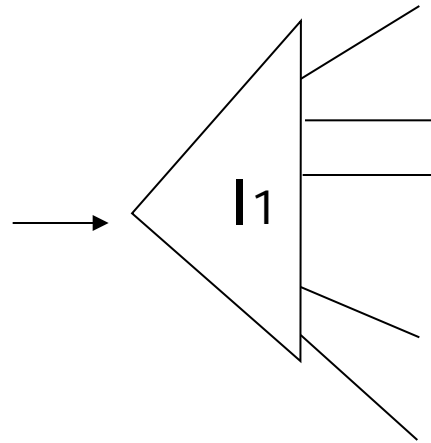

Indice Multichiave

Motivazione: trova i record per i quali

DEPT = "Toy" AND SAL > 50k

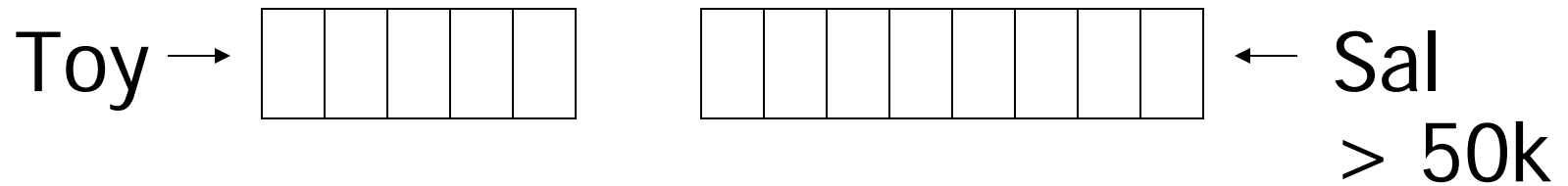
Strategia I:

- Usa un indice, ad esempio Dept.
- Ottieni tutti i record con Dept = "Toy" e verifica il loro salario



Strategia II:

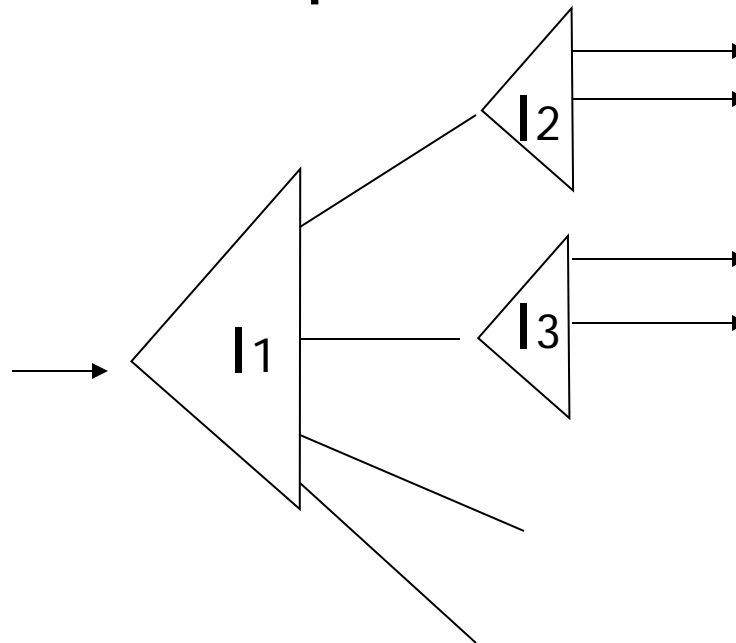
- Usa 2 indici; interseca i puntatori



Strategia III:

- Indice su chiave multipla

Una possibilita':



Esempio

Art	
Sales	
Toy	

10k	
15k	
17k	
21k	

12k	
15k	
15k	
19k	

Esempio di Record

Name=Joe
DEPT=Sales
SAL=15k

Indice su
Dept

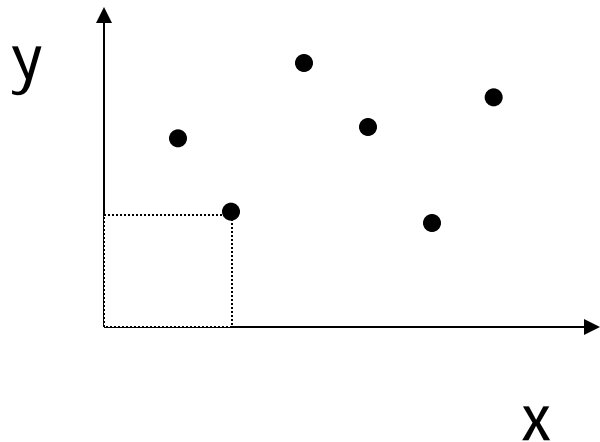
Indice su
Sal

Per quali query quest'indice e' buono?

- Dept = "Sales" AND SAL=20k
- Dept = "Sales" AND SAL \geq 20k
- Dept = "Sales"
- SAL = 20k

Dati multidimensionali:

- Dati geografici



DATI:

$\langle X_1, Y_1, \text{Attributi} \rangle$

$\langle X_2, Y_2, \text{Attributi} \rangle$

⋮

- Dati qualunque, ogni attributo e' una dimensione

Esempi di query su dati multidimensionali

- Match parziale: dati valori per una o piu' dimensioni trovare tutti i punti con quei valori
- Query di range: dati range per una o piu' dimensione trovare tutti i punti in quei range. Se sono rappresentate forme, trovare le forme che sono parzialmente o completamente incluse nei range
- Nearest neighbor query: dato un punto, trovare il punto piu' vicino

Esempi di query su dati multidimensionali

- Query “dove sono”: dato un punto vogliamo sapere in quale forma, se ce n'è una, è collocato

Esempi

- Sia $\text{Points}(x,y)$ una tabella che contiene tutti i punti
- Query di match parziale, tutti i punti con $x=5$:
`SELECT * FROM Points WHERE x=5`
- Query di range, tutti i punti con x in $[5,10]$:
`SELECT * FROM Points WHERE x >= 5 AND x <= 10`

Esempi

- Nearest neighbor query: punto più vicino a (10.0,20.0)

```
SELECT * FROM Points p WHERE NOT EXISTS(  
    SELECT * FROM Points q WHERE  
        (q.x-10.0)*(q.x-10.0) + (q.y-20.0)*(q.y-20.0) <  
        (p.x-10.0)*(p.x-10.0) + (p.y-20.0)*(p.y-20.0)  
);
```

Esempi

- Query “dove sono”: trovare i rettangoli che contengono (10.0,20.0)

- Sia abbia una tabella

Rectangles(id,xll,yll,xur,yur)

- Query

```
SELECT id FROM Rectangles WHERE  
  xll <= 10.0 AND yll <= 20.0 AND  
  xur >= 10.0 AND yur >= 20.0;
```

Trova il punto più vicino usando indici

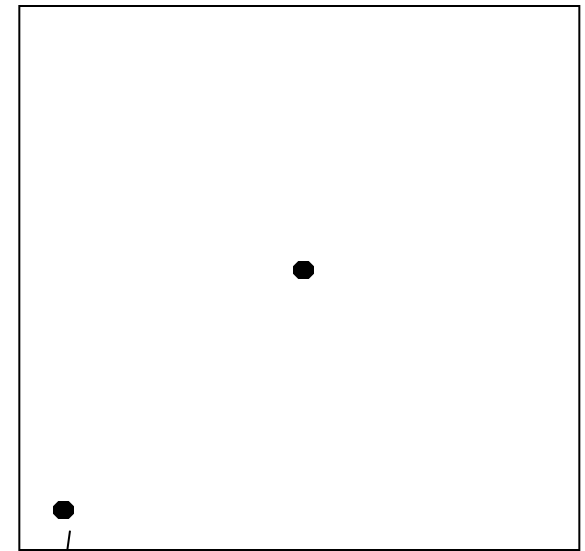
- Dato un punto (X, Y) , si costruiscono due range: $[X-d, X+d]$ e $[Y-d, Y+d]$ con d scelto dall'utente
- Si esegue una query per restituire tutti i punti nel range usando i B+-trees su x e su y
- Si prende il punto più vicino tra quelli restituiti

Trova il punto più vicino usando indici

- Problemi:
 1. Non ci sono punti nel range
 2. Il punto più vicino nel range potrebbe non essere il più vicino complessivamente
- Problema 1.: si prova con un d più grande

Problema 2.

- Se d' è la distanza del punto più vicino nel range e $d' > d$, si riprova la query con d'
- Se c'è un punto con distanza $< d'$, con la seconda query lo si trova



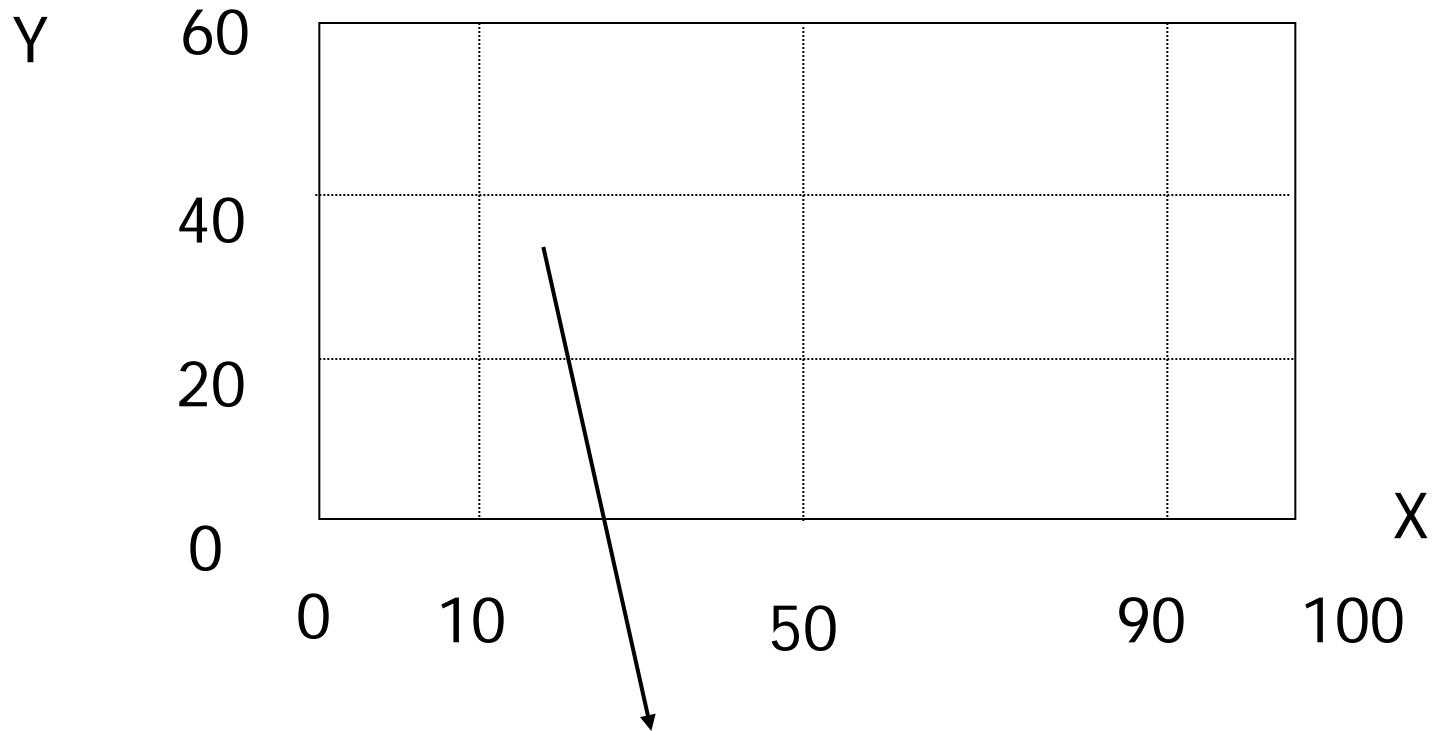
Punto più vicino
nel range

Punto più vicino
in assoluto

Strutture di tipo hash

- Grid
- Hash partizionato

Grid file

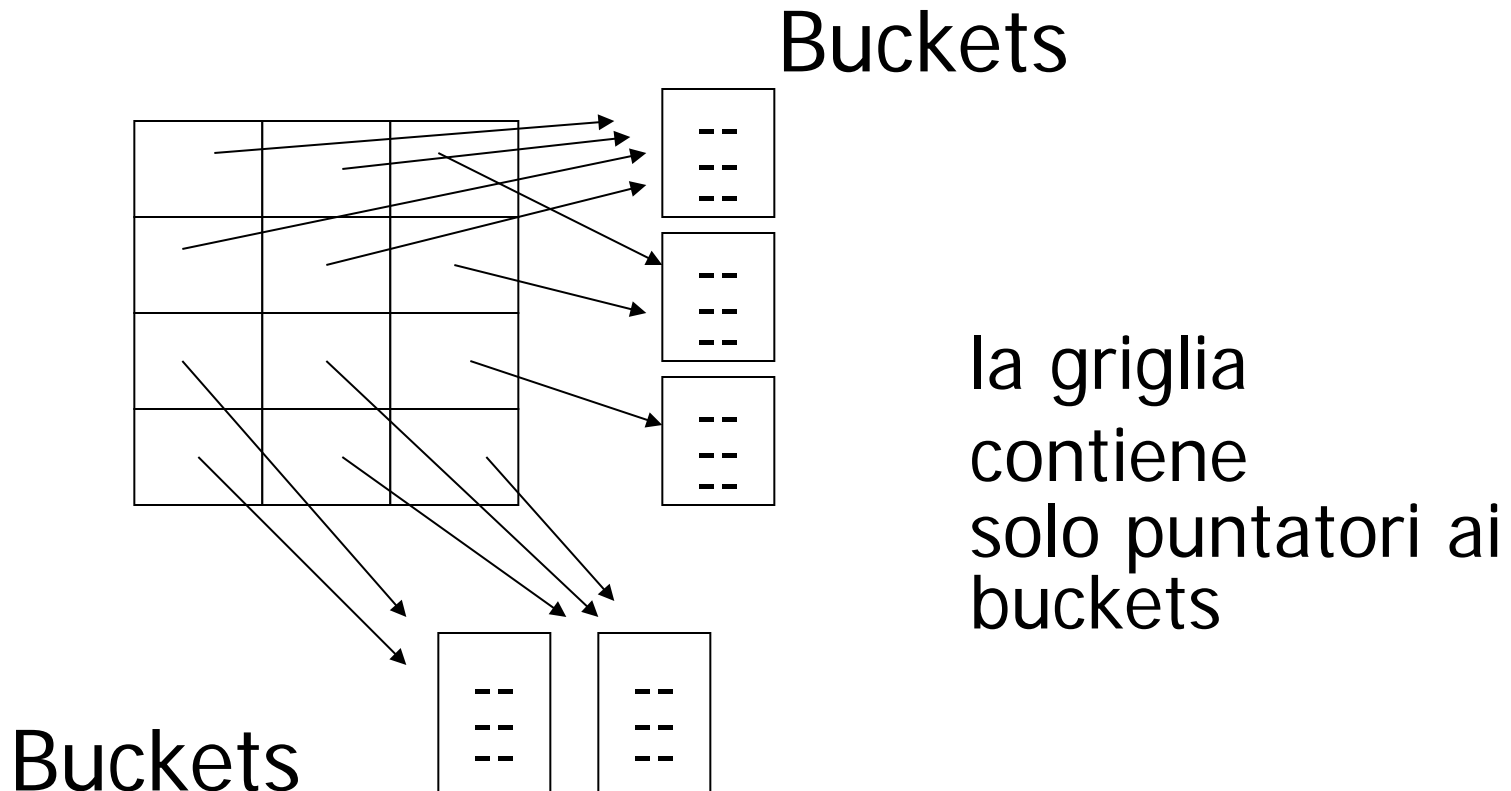


Bucket contenente i record con chiave $10 \leq X < 50$ e $20 \leq Y < 40$

Utilizzo

- Si riesce a rispondere facilmente a query del tipo
 - $X = V_i \wedge Y = W_j$
 - $X = V_i$
 - $Y = W_j$
- E anche del tipo
 - $X \geq V_i \wedge Y < W_j$

Memorizzazione su disco



Con l'indirezione

- La griglia puo' essere regolare senza spreco di spazio
- Abbiamo il prezzo dell'indirezione

Grid files

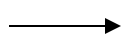
- ⊕ Buoni per ricerche su piu' chiavi
- ⊖ Necessita' di spazio
- ⊖ I vari range devono dividere i record in modo uniforme

Funzione hash partizionate

Idea:

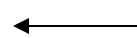
010110 1110010

Key1



h1

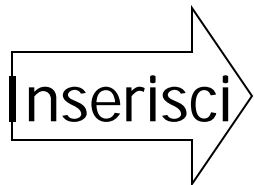
h2



Key2

Es:

h1(toy)	=0	000	
h1(sales)	=1	001	<Fred>
h1(art)	=1	010	
.		011	
h2(10k)	=01	100	
h2(20k)	=11	101	<Joe><Sally>
h2(30k)	=01	110	
h2(40k)	=00	111	
:			



<Fred,toy,10k> , <Joe,sales,10k>
<Sally,art,30k>

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe><Jan>
h1(art)	=1	010	<Mary>
.		011	
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom><Bill>
h2(40k)	=00	111	<Andy>
:			

- Trova Imp. Con Dept. = Sales \wedge Sal=40k

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe> <Jan>
h1(art)	=1	010	<Mary>
.		011	
.		100	<Sally>
h2(10k)	=01	101	
h2(20k)	=11	110	<Tom> <Bill>
h2(30k)	=01	111	<Andy>
h2(40k)	=00		
:			

- Trova Imp. con Sal=30k

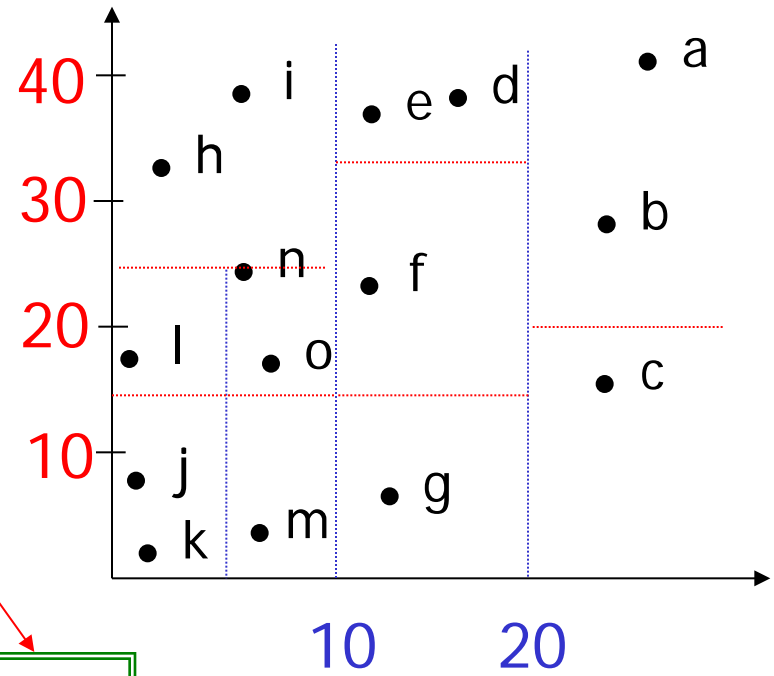
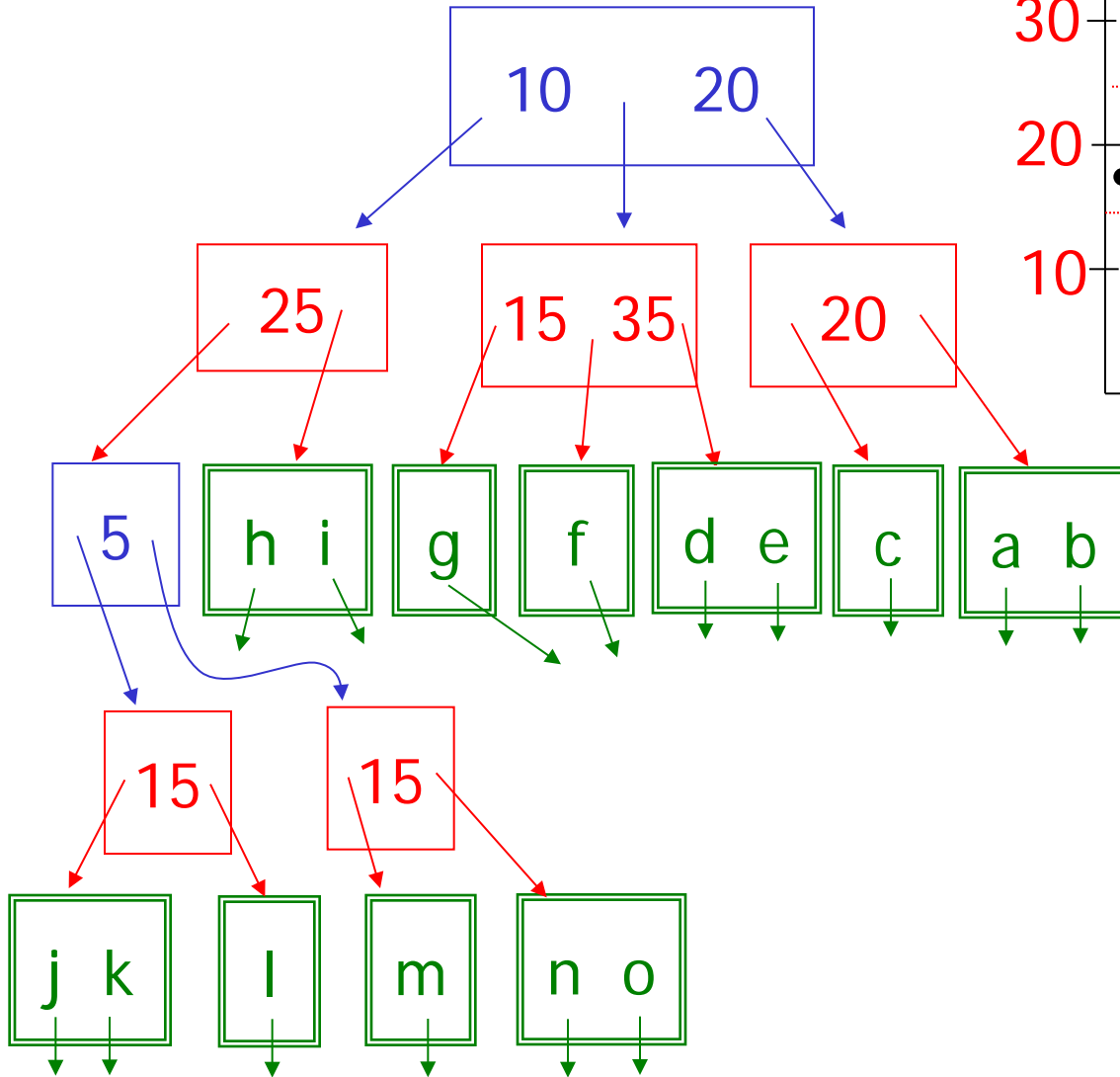
Guarda qui

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe><Jan>
h1(art)	=1	010	<Mary>
.		011	
.			
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom><Bill>
h2(40k)	=00	111	<Andy>
:			

- Find Emp. with Dept. = Sales

Guarda qui

Kd-tree



Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
 - Given a number n it must be easy to retrieve record n
 - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
 - E.g. gender, country, state, ...
 - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
 - Bitmap has as many bits as records
 - In a bitmap for value v , the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

Bitmap Indices (Cont.)

record number	<i>name</i>	<i>gender</i>	<i>address</i>	<i>income-level</i>
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for <i>gender</i>		Bitmaps for <i>income-level</i>											
m	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	1	0	0	1	0	L1	<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	1	0	1	0	0
1	0	0	1	0									
1	0	1	0	0									
f	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	0	1	1	0	1	L2	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	0	0	0
0	1	1	0	1									
0	1	0	0	0									
		L3	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	0	1					
0	0	0	0	1									
		L4	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	0	1	0					
0	0	0	1	0									
		L5	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0					
0	0	0	0	0									

Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
 - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
 - Intersection (and)
 - Union (or)
 - Complementation (not)

Bitmap Indices (Cont.)

- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
 - E.g. $100110 \text{ AND } 110011 = 100010$
 $100110 \text{ OR } 110011 = 110111$
 $\text{NOT } 100110 = 011001$
 - Males with income level L1: $10010 \text{ AND } 10100 = 10000$
 - Can then retrieve required tuples.
 - Counting number of matching tuples is even faster

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
 - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
 - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
 - Existence bitmap to note if there is a valid record at a record location
 - Needed for complementation
 - $\text{not}(A=v)$: *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
 - To correctly handle SQL null semantics for $\text{NOT}(A=v)$:
 - intersect above result with *(NOT bitmap-A-Null)*