Architettura fisica dei DBMS

Daniele Riboni
Università degli Studi di Cagliari
Dip. di Matematica e Informatica

Slide tratte in parte dal materiale di supporto del libro: Atzeni, Ceri, Fraternali, Paraboschi, Torlone, Basi di dati, Quarta edizione, McGraw-Hill, 2014, e da
Hector Garcia-Molina, CS 245: Database System Principles Course @ Stanford Univ.

Materiale

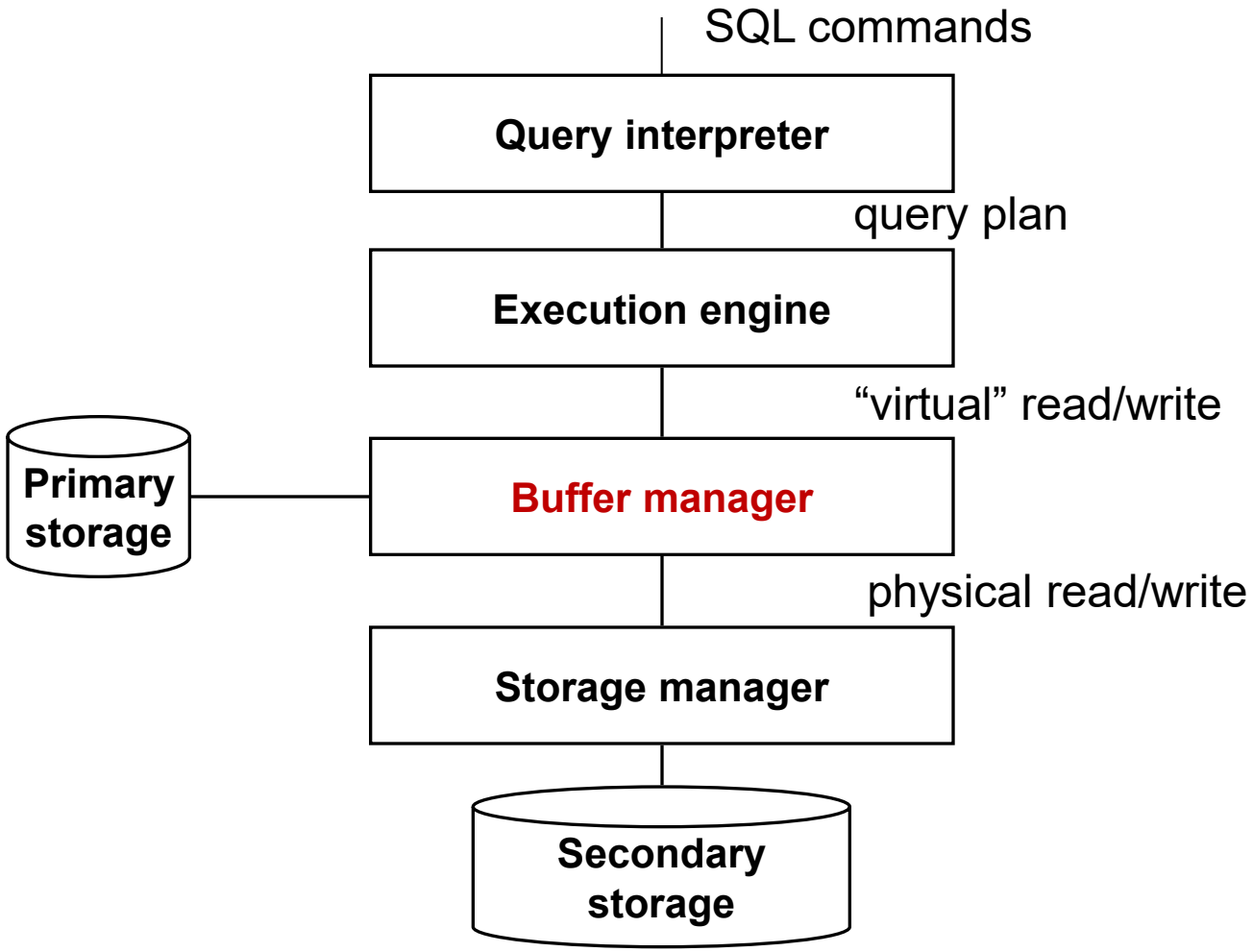
- Atzeni, Ceri, Fraternali, Paraboschi, Torlone. “Basi di dati”. Quinta edizione.McGraw-Hill, 2018
  - Capitolo 11

oppure

- Garcia-Molina, Ullman, Widom. “Database Systems: The Complete Book”. 2/E. Pearson, 2009
  - Sections 13.1 to 13.2, 13.5 to 13.8, 14.1 to 14.3, 15.1 to 15.4.1

Gestione della memoria

Architettura di alto livello del DBMS



Memoria principale e secondaria

- I programmi possono fare riferimento solo a dati in memoria principale
- Le basi di dati debbono essere scritte in memoria secondaria per due motivi:
  - Dimensioni
  - Persistenza
- I dati in memoria secondaria possono essere utilizzati solo se prima trasferiti in memoria principale

Memoria principale e secondaria

- I dispositivi di memoria secondaria sono organizzati in blocchi di lunghezza fissa (alcuni KB)

Le uniche operazioni sui dispositivi sono la lettura e la scrittura di un intero blocco

- Pagina: unità di suddivisione logica della RAM

- Per semplicità assumiamo:
  dimensione pagina = dimensione blocco

Tempi di accesso alla memoria secondaria

- Hard disk tradizionale (HDD):
  - tempo di posizionamento della testina, tempo di latenza, tempo di trasferimento...
- In media ~ 1 - 10 ms

- Memoria secondaria con unità a stato solido (SSD)
  - Più costosa, ma tempi di accesso circa 50 volte minori rispetto a HDD

- Accesso a memoria terziaria (tapes, optical jukebox)
- In media ~ 5 - 60 secondi

Memoria principale e secondaria

- Il costo di un accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria centrale
- Semplifichiamo: **il costo di una operazione del DBMS dipende esclusivamente dal numero di accessi a memoria secondaria** *(numero di blocchi che devono essere letti)*

- Accessi a blocchi "vicini" costano meno **(contiguità)**

Buffer manager

Buffer:

- area di memoria centrale, gestita dal DBMS (preallocata) e condivisa fra le transazioni
- "deposito temporaneo" delle porzioni di DB correntemente utilizzate

Scopo del buffer manager:

- Ridurre il numero di accessi alla memoria secondaria

Dati gestiti dal buffer manager

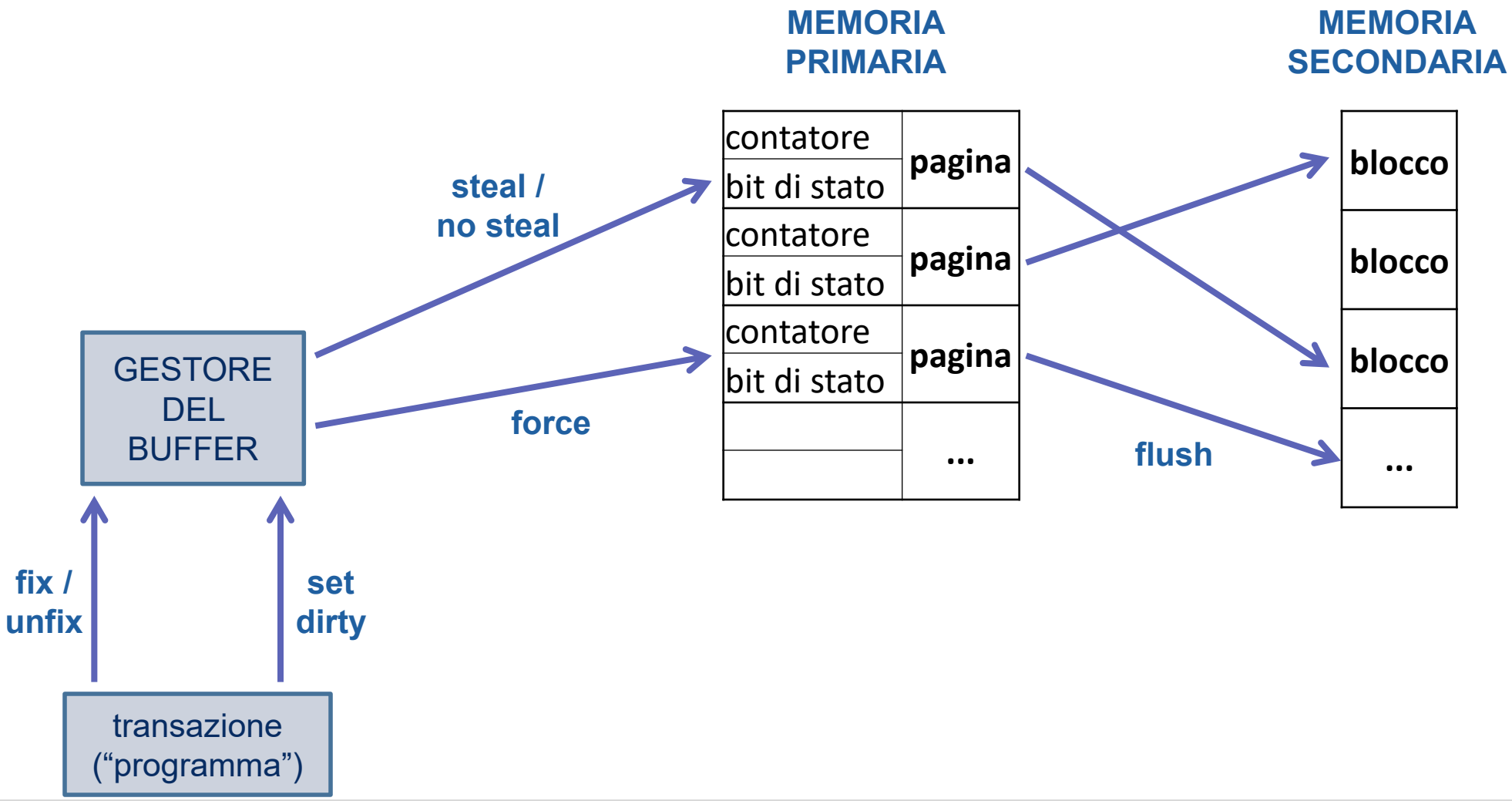
- Il contenuto del buffer

- Un direttorio che per ogni pagina mantiene
  - il file e il numero del blocco corrispondente in HD
  - due variabili di stato:
    - un contatore che indica quanti programmi utilizzano la pagina
    - un bit di stato che indica se la pagina è “sporca”, cioè se è stata modificata (la scrittura su memoria secondaria può essere differita)

Funzioni del buffer manager

- Riceve richieste di lettura e scrittura di blocchi (che contengono record)
- Utilizza il buffer quando possibile
- Esegue le richieste accedendo alla memoria secondaria solo quando indispensabile
- Esegue le primitive:
  - fix, unfix, setDirty, force

Gestione del buffer



Interfaccia offerta dal buffer manager

- fix: richiesta di una pagina
  - richiede una lettura in memoria secondaria solo se la pagina non è nel buffer
  - incrementa il contatore associato alla pagina

- setDirty: comunica al buffer manager che la pagina è stata modificata

Interfaccia offerta dal buffer manager

- unfix: indica che la transazione ha concluso l'utilizzo della pagina
- decrementa il contatore associato alla pagina

- force: trasferisce in modo sincrono una pagina in memoria secondaria

- flush: scrittura asincrona in memoria

Esecuzione della fix

- Cerca la pagina nel buffer
- Se c'è, restituisce l'indirizzo

- Altrimenti, cerca una pagina libera nel buffer (avente contatore a zero) e la sovrascrive
  - Se ce n'è più di una? Politica FIFO, oppure Least Recently Used, o altra politica...
  - La pagina sovrascritta viene prima salvata su disco se necessario

Esecuzione della fix

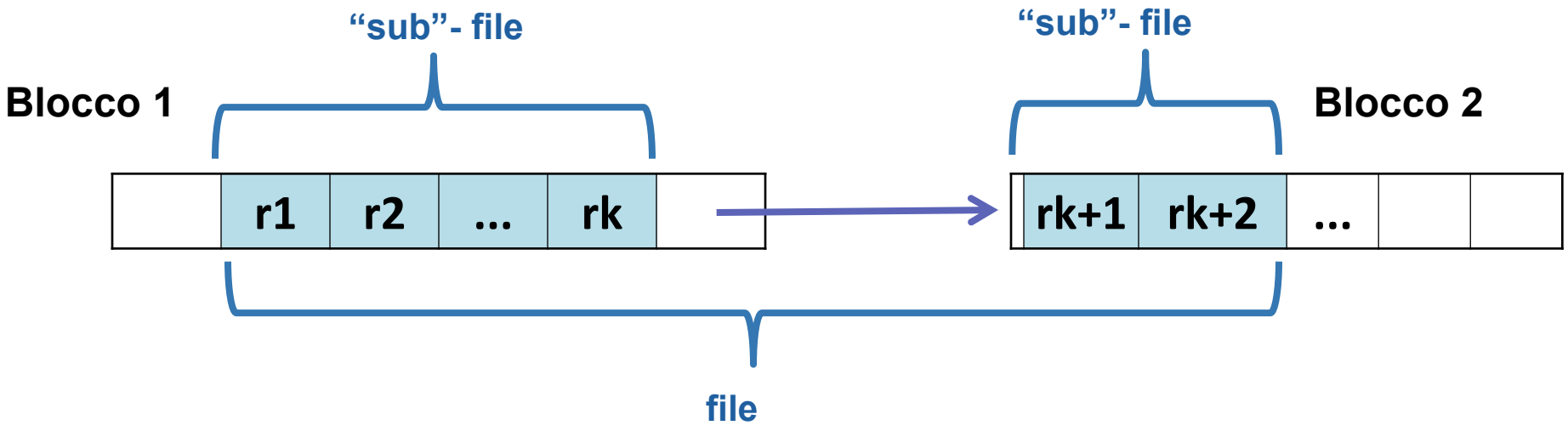
- E se non ci sono contatori a zero?

- Due alternative
  - “steal”: selezione di una "vittima“ (pagina occupata del buffer)
  - “no-steal”: l'operazione viene posta in attesa

DBMS e file system

- L'organizzazione interna dei file è gestita direttamente dal DBMS
  - struttura all'interno dei singoli blocchi
  - distribuzione dei record nei blocchi

Blocchi e file



rx = record

Serializzazione dei record

- Alternative:
  - Formato dei record fisso oppure auto-descrittivo
  - Lunghezza dei campi fissa oppure variabile

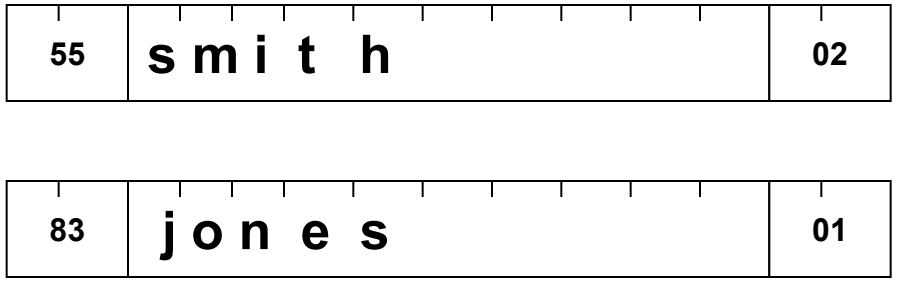
Formato fisso e lunghezza fissa

Employee record

(1) E#, 2 byte integer

(2) E.name, 10 char.

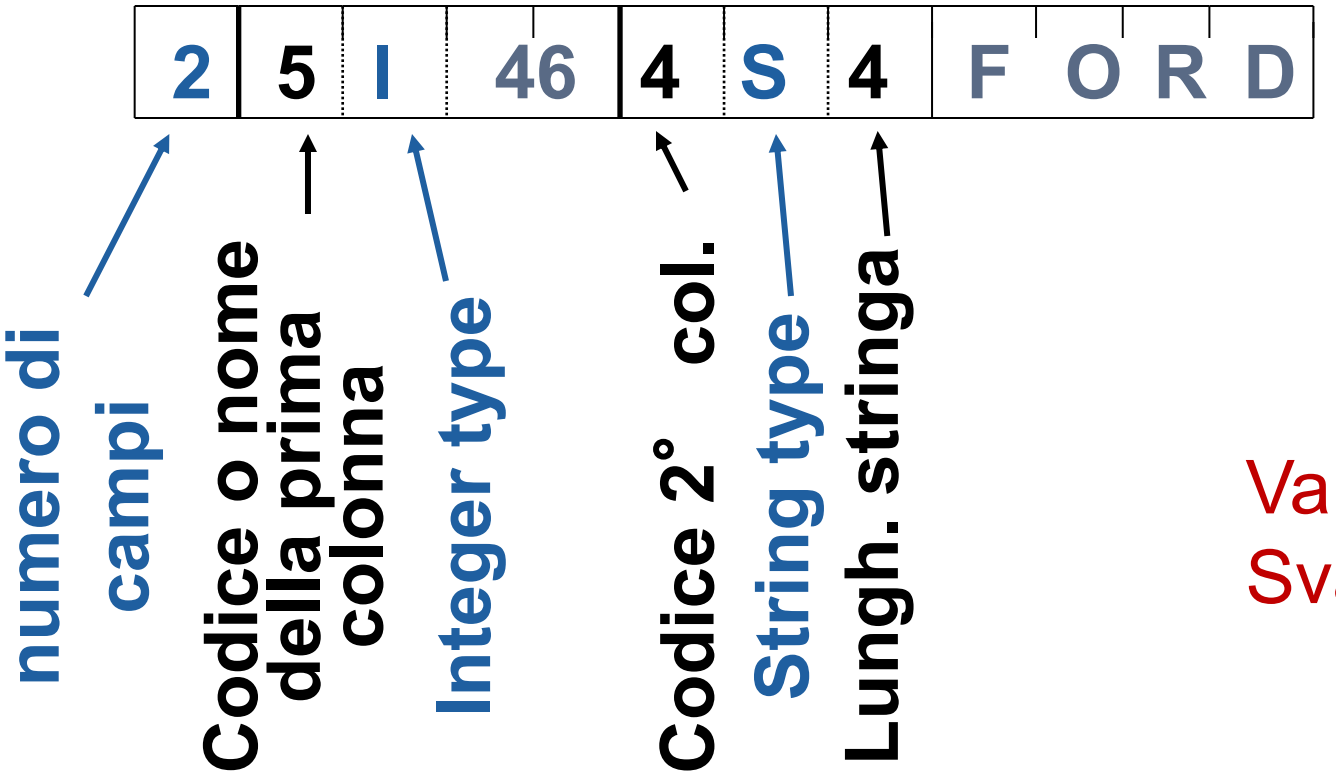
(3) Dept, 2 byte code



Schema (nel direttorio)

Record (in altri file)

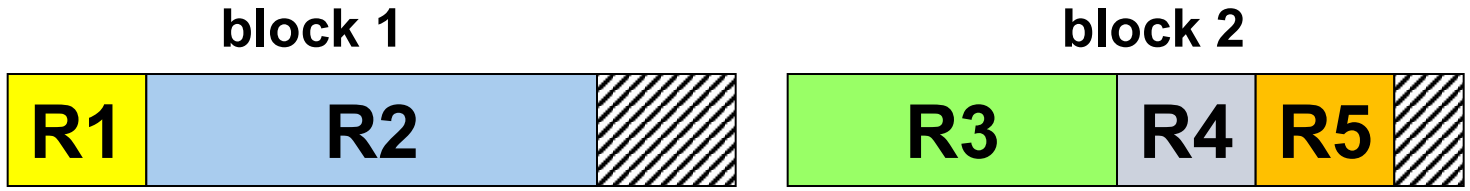
Formato auto-descrittivo e lunghezza variabile



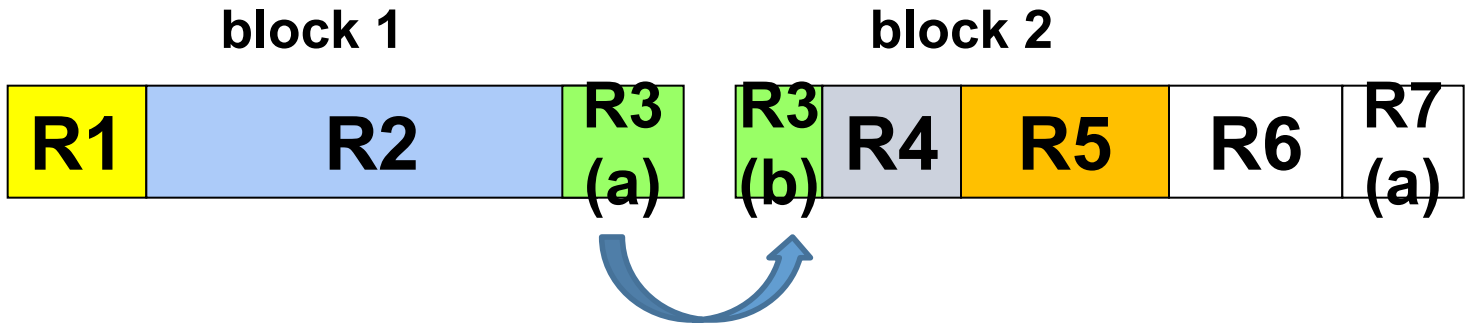
Vantaggi e Svantaggi?

Scrittura su disco: Unspanned vs Spanned

- Unspanned: ogni record sta in un solo blocco



- Spanned: record può essere diviso in più blocchi



Spanned vs. unspanned

- Unspanned molto più semplice da implementare e veloce, ma determina perdita di spazio

- Spanned necessario per record più grandi di un blocco

Row vs Column Store

- I formati visti in precedenza scrivono su disco in ordine di record (row store)...

- Altra opzione: scrivere in ordine di colonna (column store)

Row Store

- Esempio: Relazione “Acquisto”
  - Acquisto(id, cust, prod, store, price, date, qty)

BLOCK1

id1	cust1	prod1	store1	price1	date1	qty1

BLOCK2

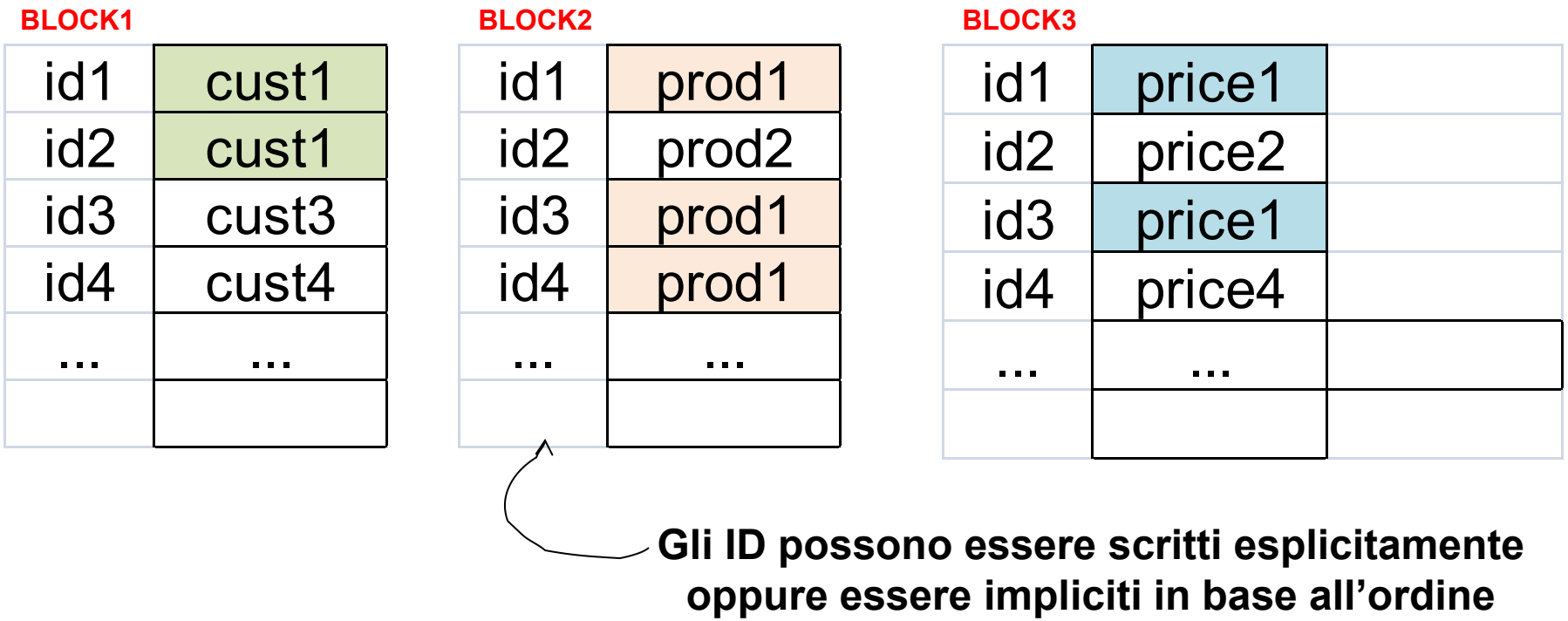
id2	cust1	prod2	store2	price2	date2	qty2

BLOCK3

id3	cust3	prod1	store1	price1	date3	qty3

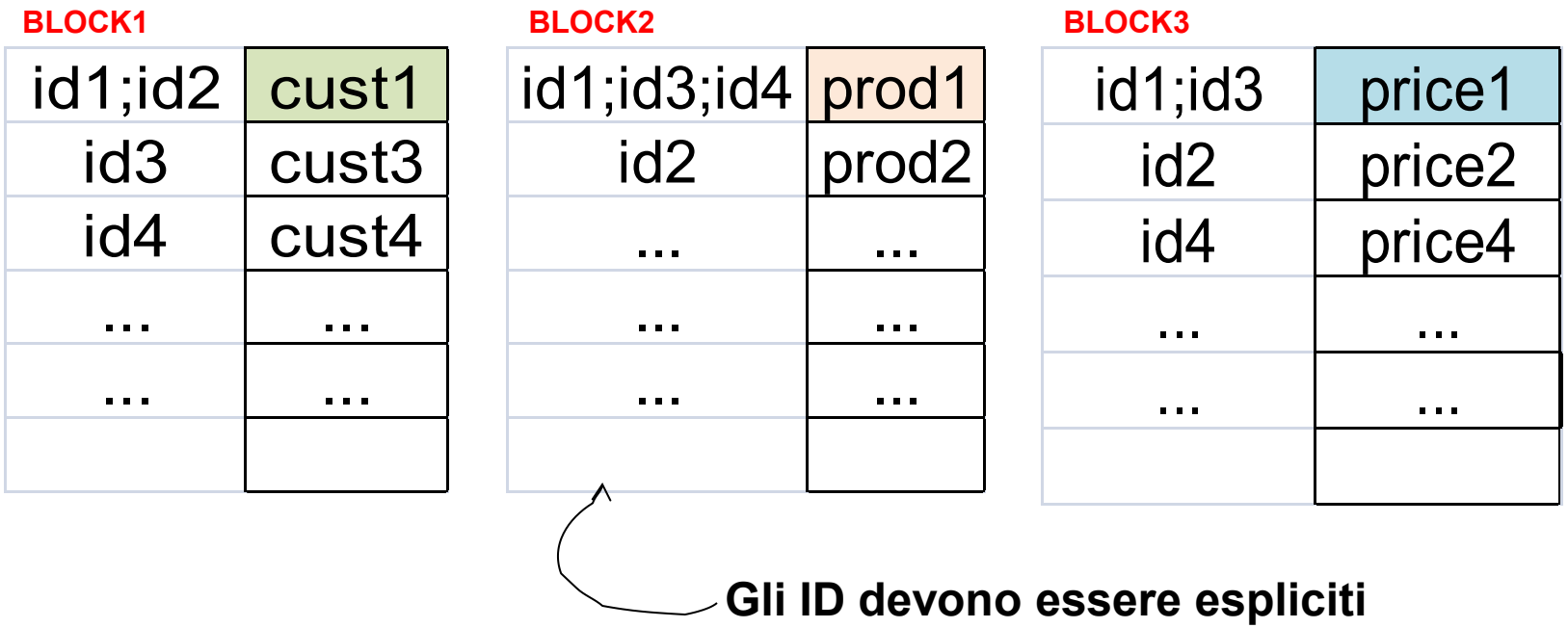
Column Store

- Esempio: Relazione “Acquisto”
  - Acquisto(id, cust, prod, store, price, date, qty)



Column Store - variante

- Esempio: Relazione “Acquisto”
  - Acquisto(id, cust, prod, store, price, date, qty)



Row vs Column Store

- Vantaggi del Column Store
  - Letture efficienti per applicazioni di “data mining” e statistiche
- Vantaggi del Row Store
  - Più efficienti per letture e scritture di molteplici campi dello stesso record
- Nel seguito assumiamo di usare Row Store

Strutture sequenziali

- Esiste un ordinamento fra le tuple che può essere rilevante ai fini della gestione

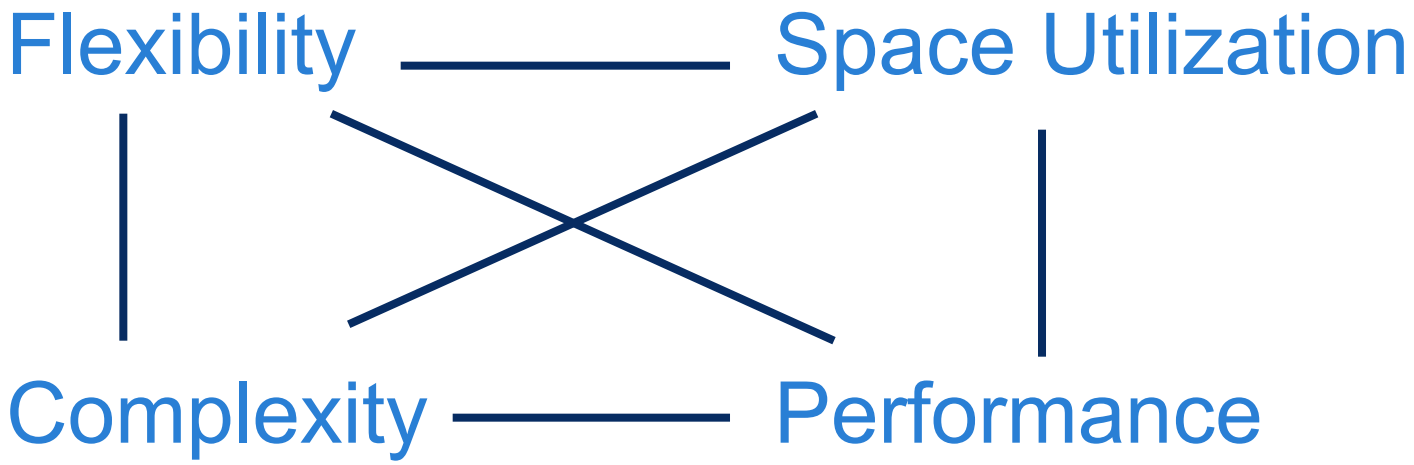
- Organizzazione disordinata:
  - Ordinamento fisico dei record ma non logico
  - Gli inserimenti vengono effettuati al posto di record cancellati, oppure in coda (con riorganizzazioni periodiche)

- Organizzazione ordinata:
  - L'ordinamento dei record è coerente con quello di un campo
  - Usata principalmente per memorizzare gli indici

Confronto

- Ci sono molti modi per organizzare i record in memoria

  Qual è il migliore?



Indici

Indici

- Indice:
  - struttura ausiliaria per l'accesso (efficiente) ai record
  - accesso sulla base dei valori di uno o più campi
  - i campi sono detti chiave (ma non sono necessariamente identificanti)

- 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

Tipi di indice

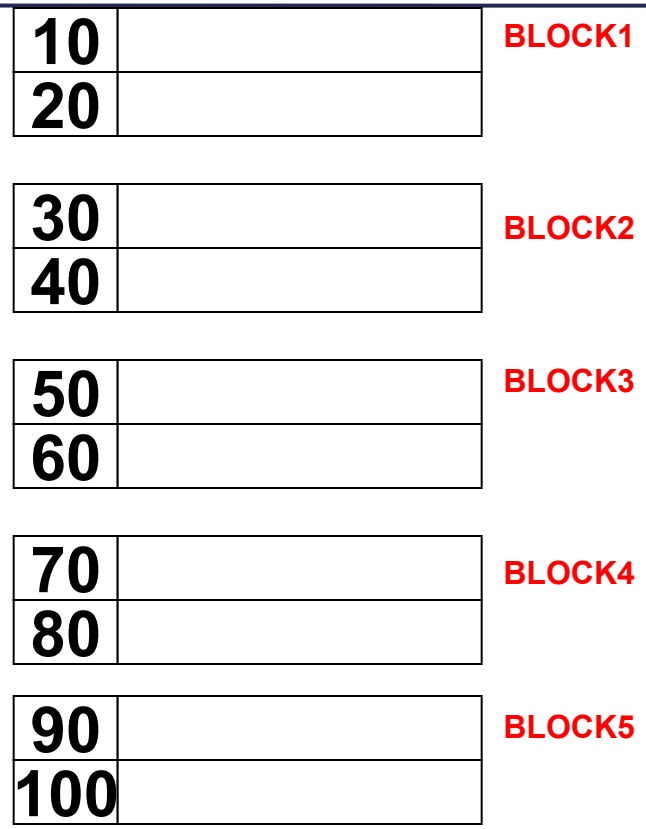
- Indice primario:
  - su un campo sul cui ordinamento è basata la memorizzazione
- Indice secondario
  - su una struttura non ordinata
  - su una struttura ordinata, ma su un campo che non determina l'ordinamento

Tipi di indice

- Indice denso:
  - contiene una entry per ciascun valore del campo chiave
- Indice sparso:
  - contiene un numero di entry inferiore rispetto al numero di valori diversi del campo chiave

Indice denso vs sparso

Scrittura ordinata



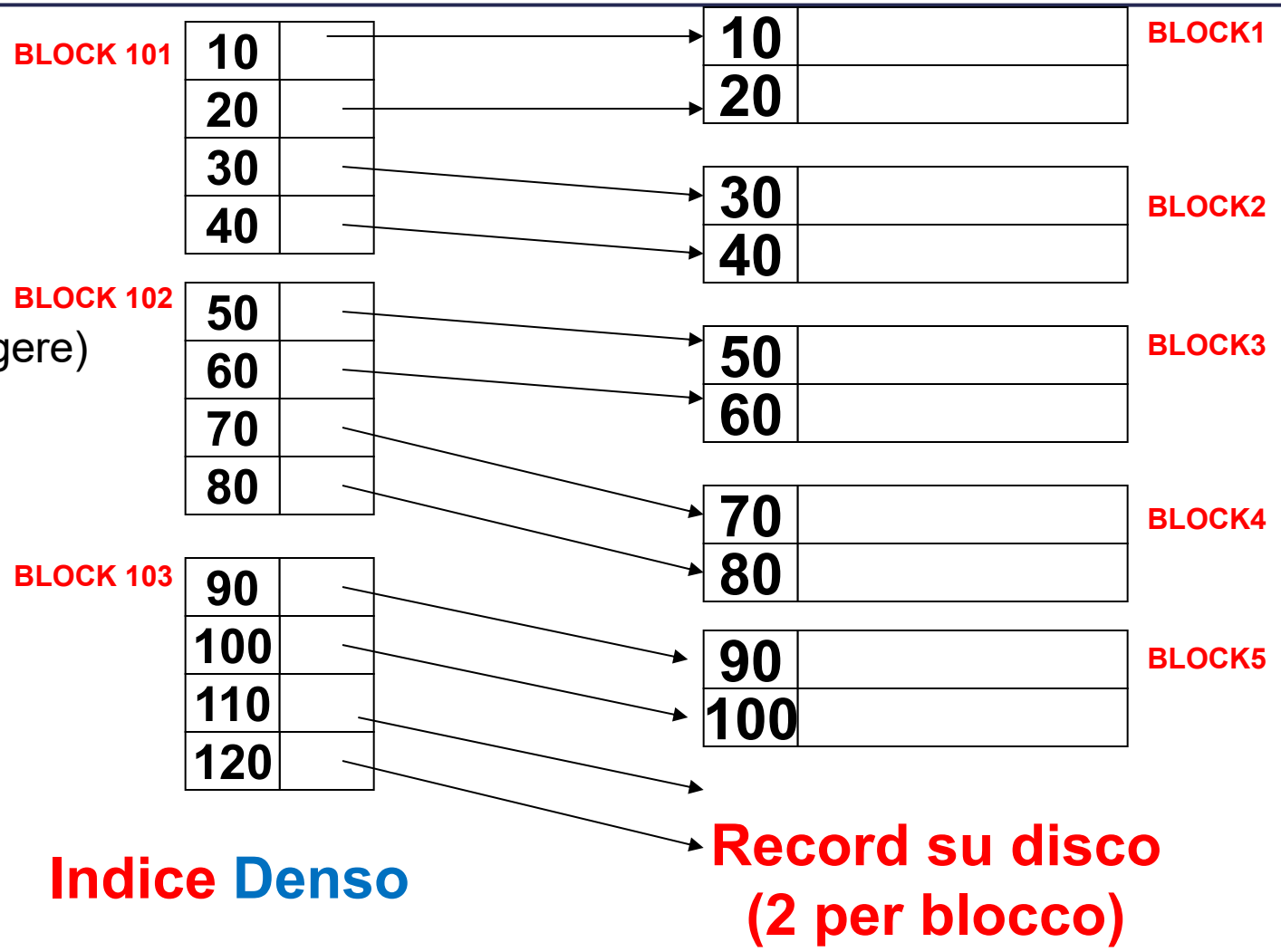
Record su disco (2 per blocco)

Indice denso vs sparso

Scrittura ordinata

(Index / No index)
Vantaggi?
(pensa in termini di numero di blocchi da leggere)

Posso fare meglio?



Indice denso vs sparso

Scrittura ordinata

Posso fare meglio?

Sparso vs. Denso: Tradeoff

- Sparso:

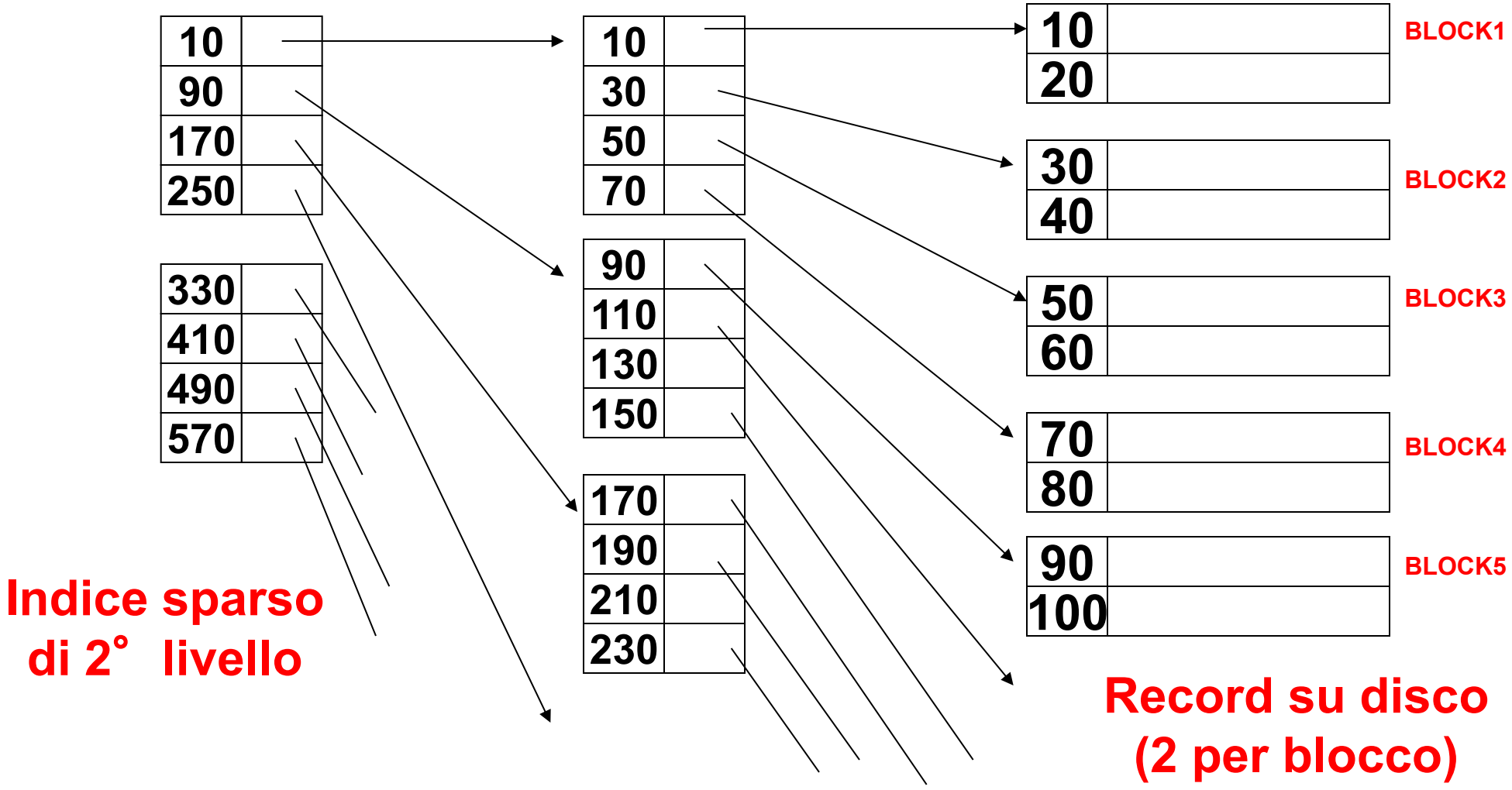
  Meno occupazione di spazio: posso tenere più parte dell'indice in RAM

- Denso:

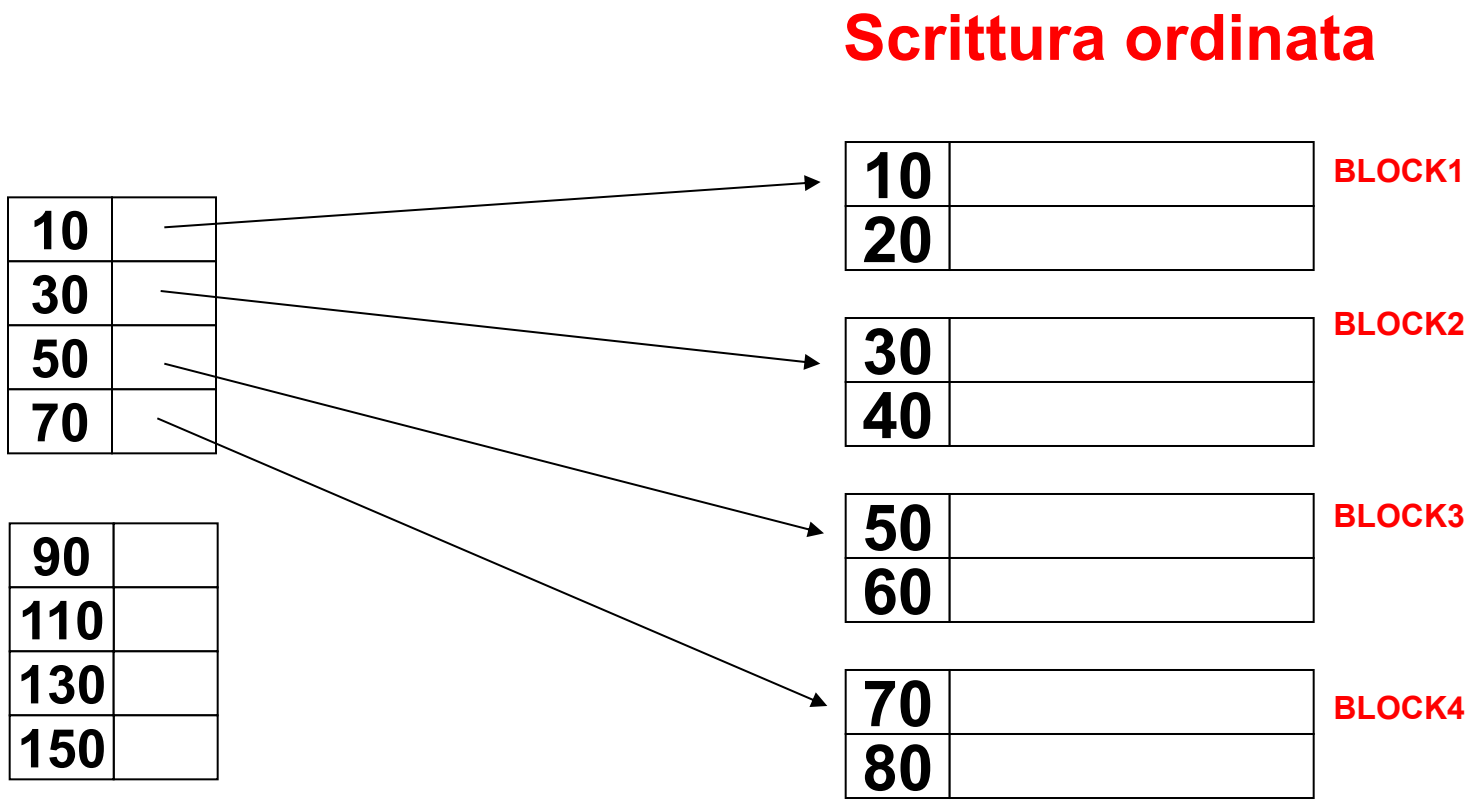
  Posso sapere se esiste un record con una certa chiave guardando solo l'indice

Indice denso vs sparso

Scrittura ordinata



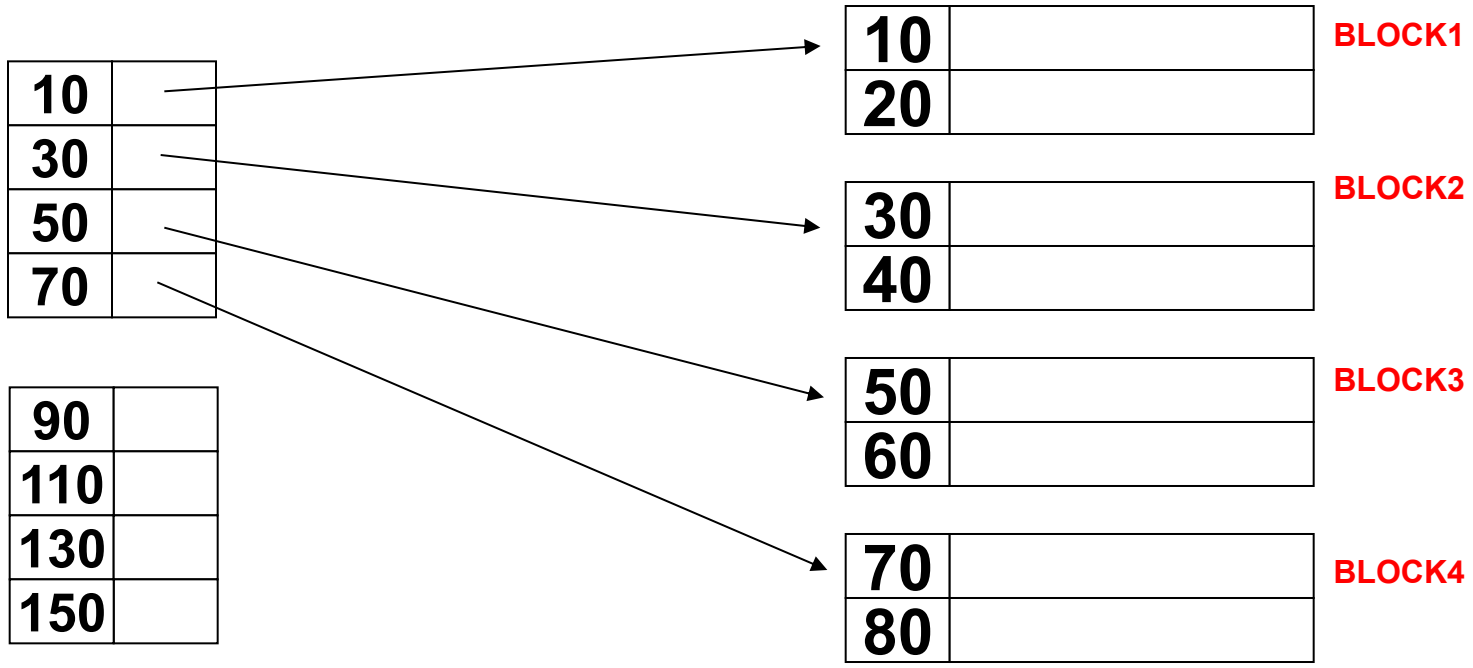
Cancellazione da indice sparso



Cancellazione da indice sparso

– delete record 40

Scrittura ordinata

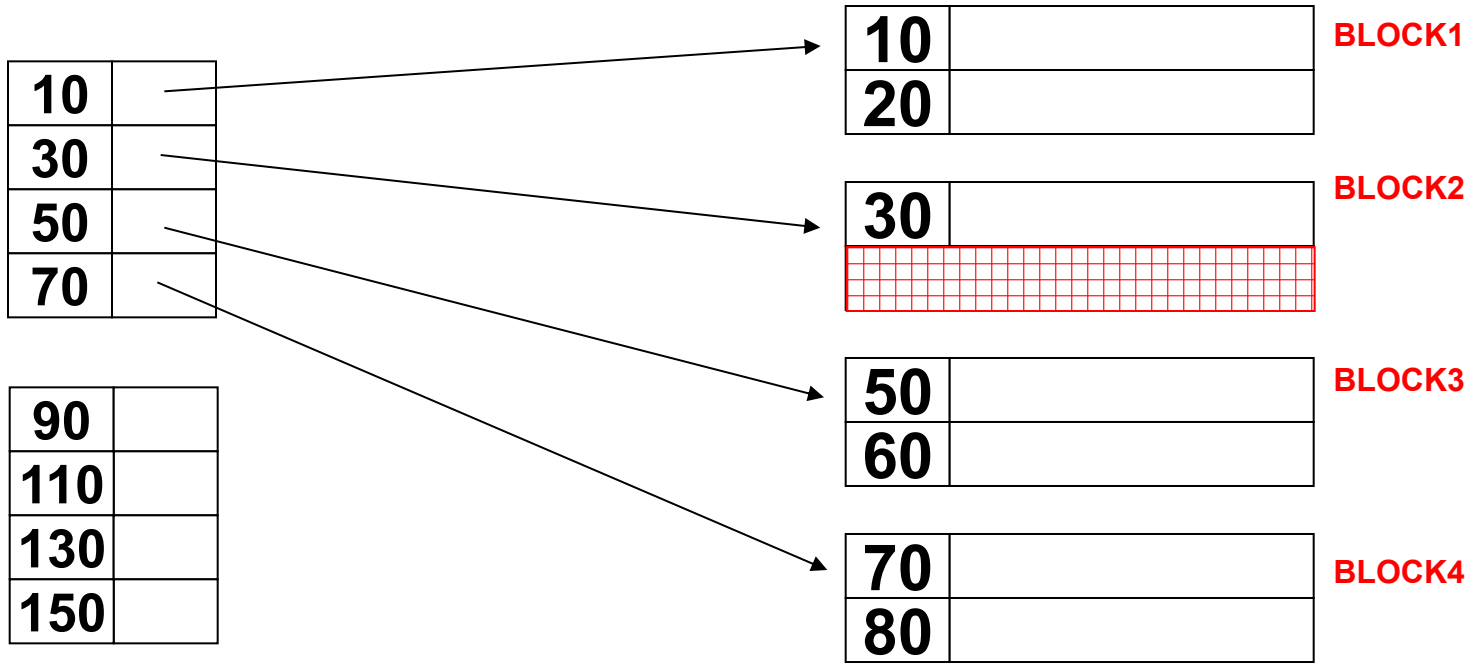


Record su disco (2 per blocco)

Cancellazione da indice sparso

– delete record 40

Scrittura ordinata

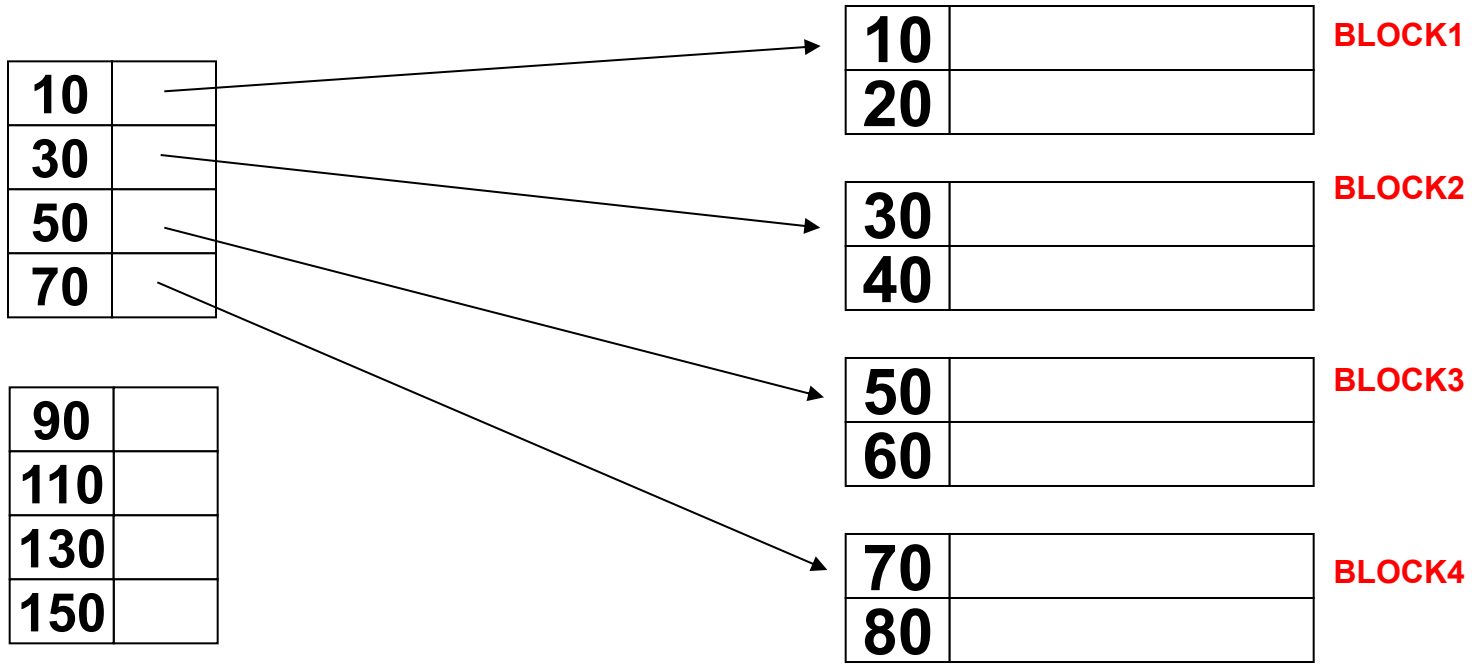


Record su disco (2 per blocco)

Cancellazione da indice sparso

– delete record 30

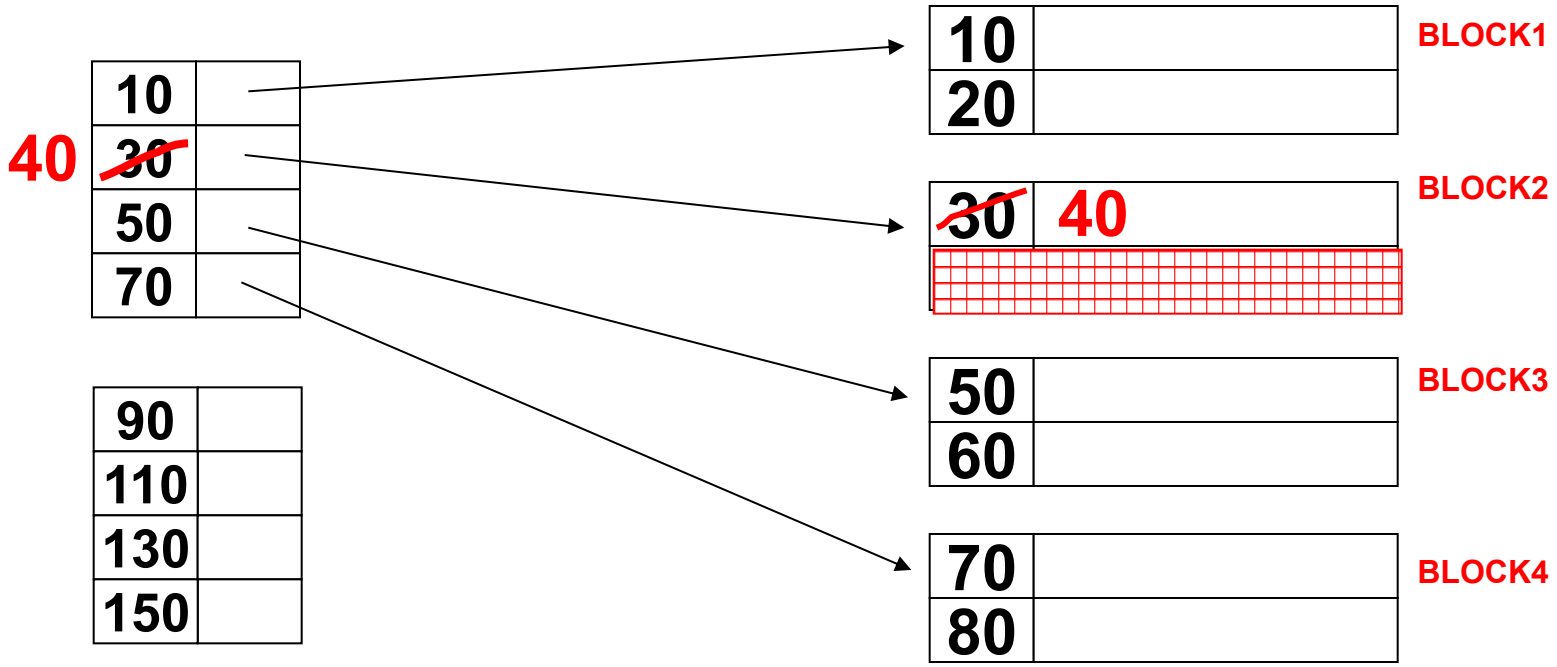
Scrittura ordinata



Record su disco (2 per blocco)

Cancellazione da indice sparso

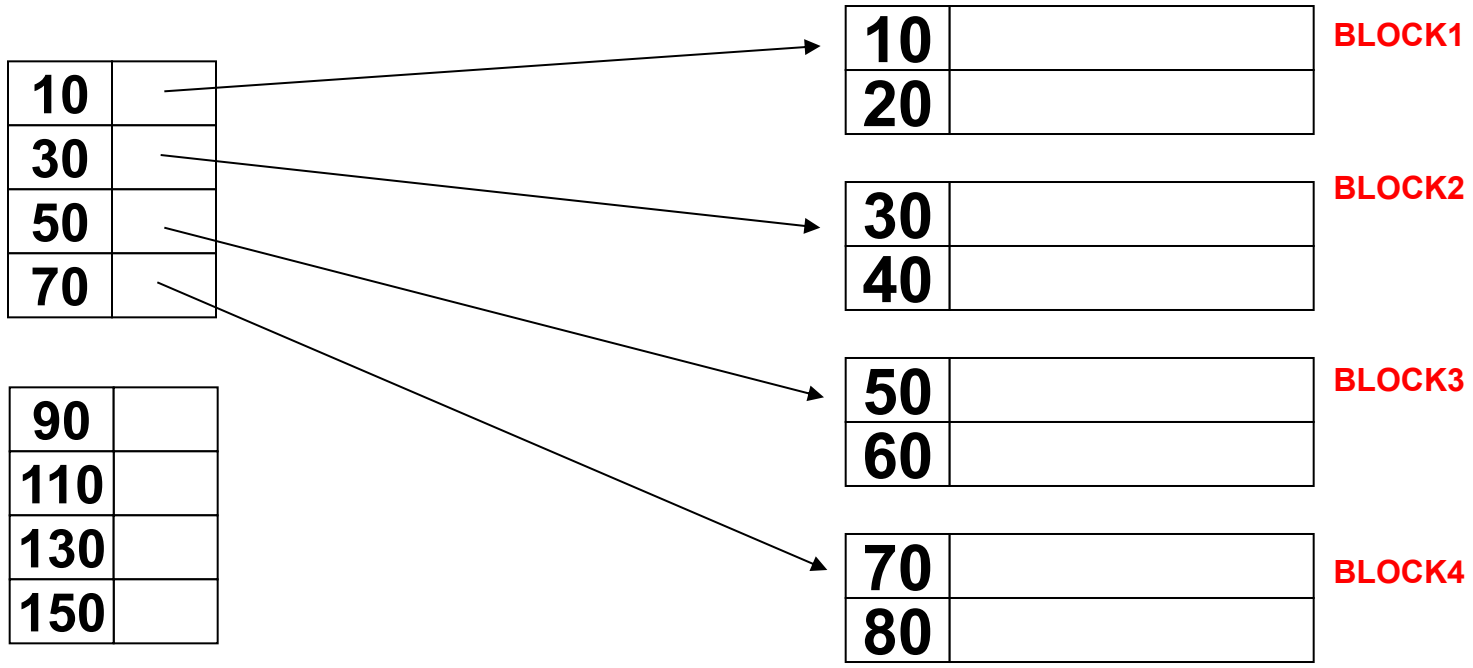
– delete record 30



Cancellazione da indice sparso

– **delete records 30 & 40**

Scrittura ordinata

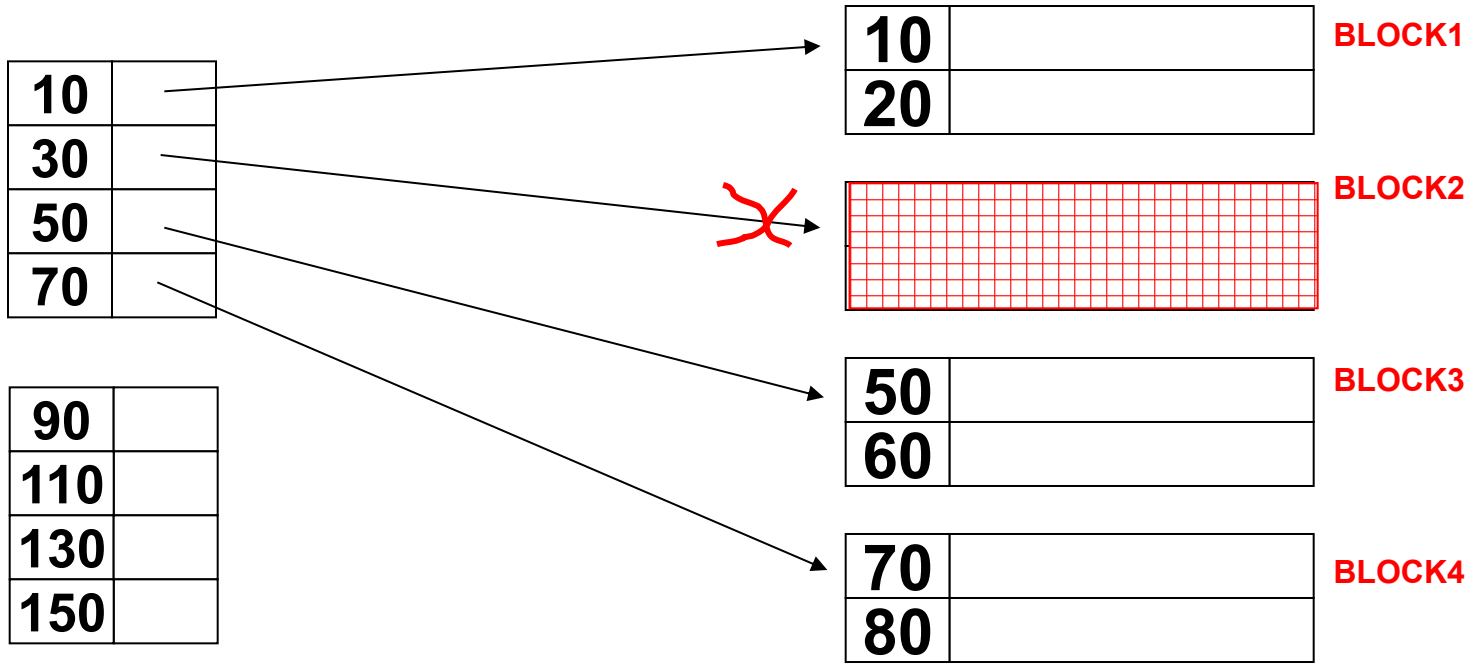


Record su disco
(2 per blocco)

Cancellazione da indice sparso

– **delete records 30 & 40**

Scrittura ordinata

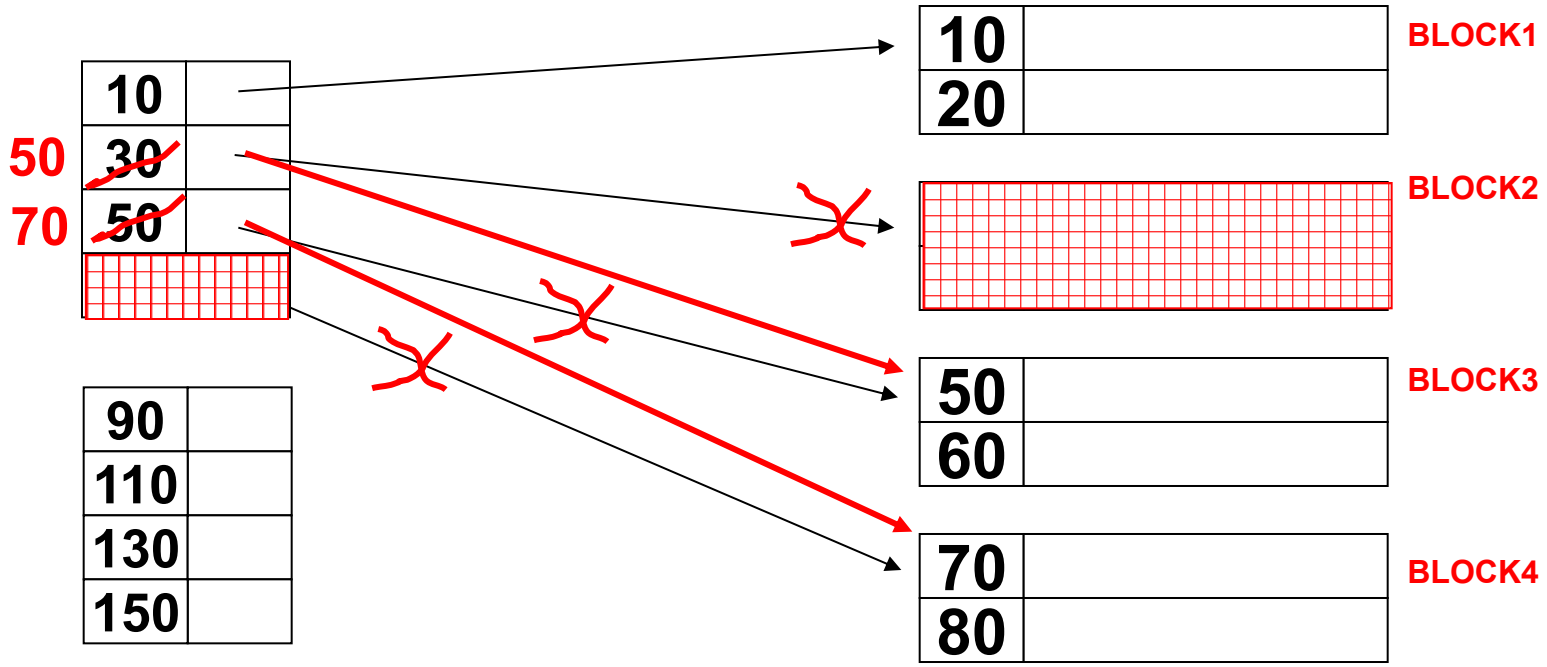


Record su disco
(2 per blocco)

Cancellazione da indice sparso

– **delete records 30 & 40**

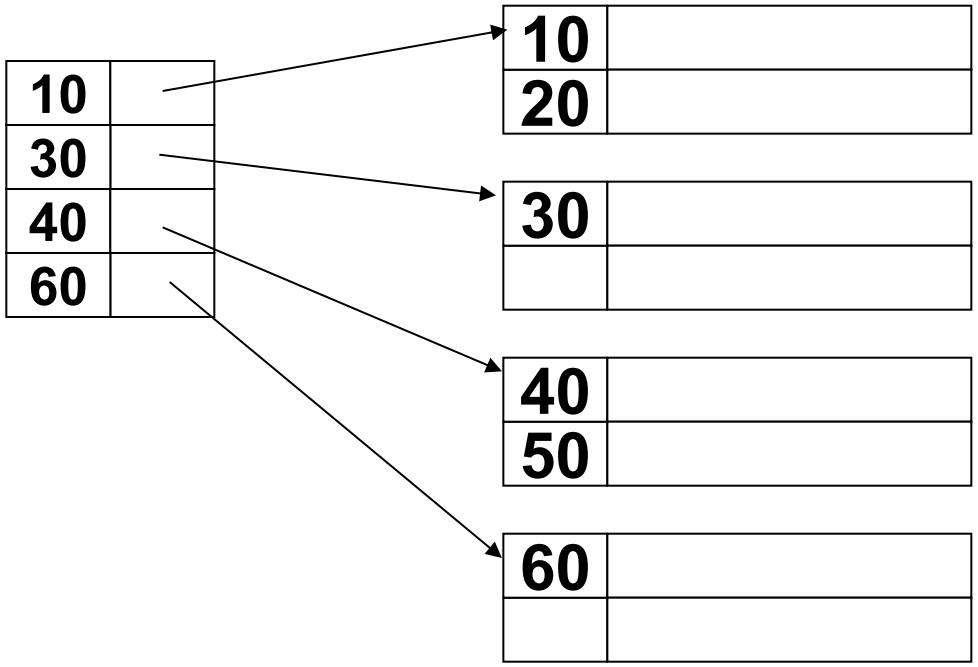
Scrittura ordinata



Record su disco (2 per blocco)

Inserimento in indice sparso

– insert record 25



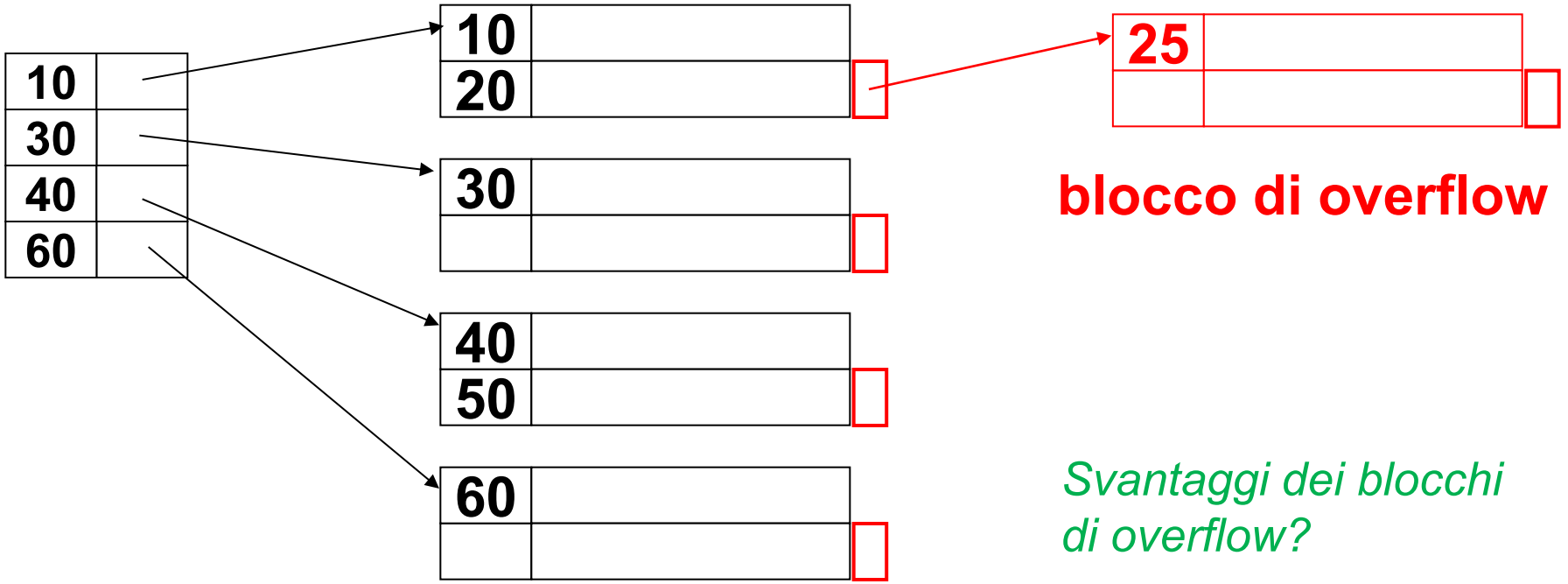
Cosa devo fare?

Cosa succede se non c'e' spazio dopo il 30?

Scrittura ordinata

Inserimento in indice sparso

– insert record 25



blocco di overflow

Svantaggi dei blocchi di overflow?

Scrittura ordinata

Indici primari

Vantaggi:

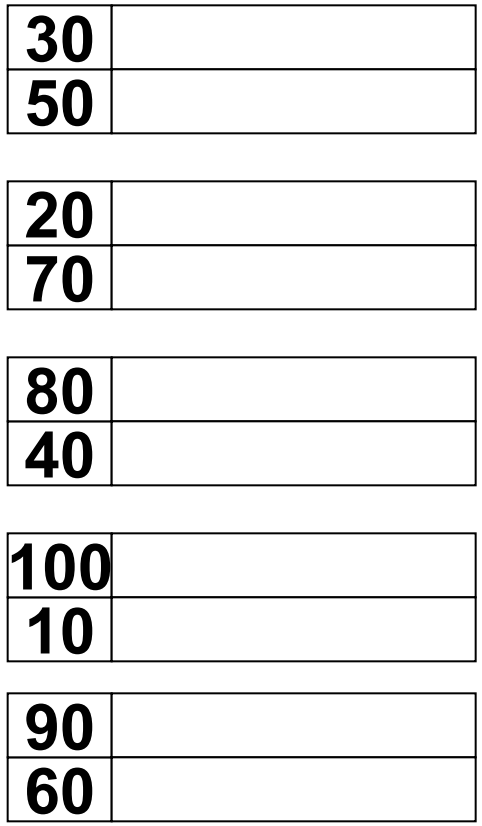
- Semplici da implementare
- Efficienti per scansioni (= letture di sequenze contigue di record)

Svantaggi:

- Modifiche costose
- Difficile mantenere l'ordine

Indici secondari

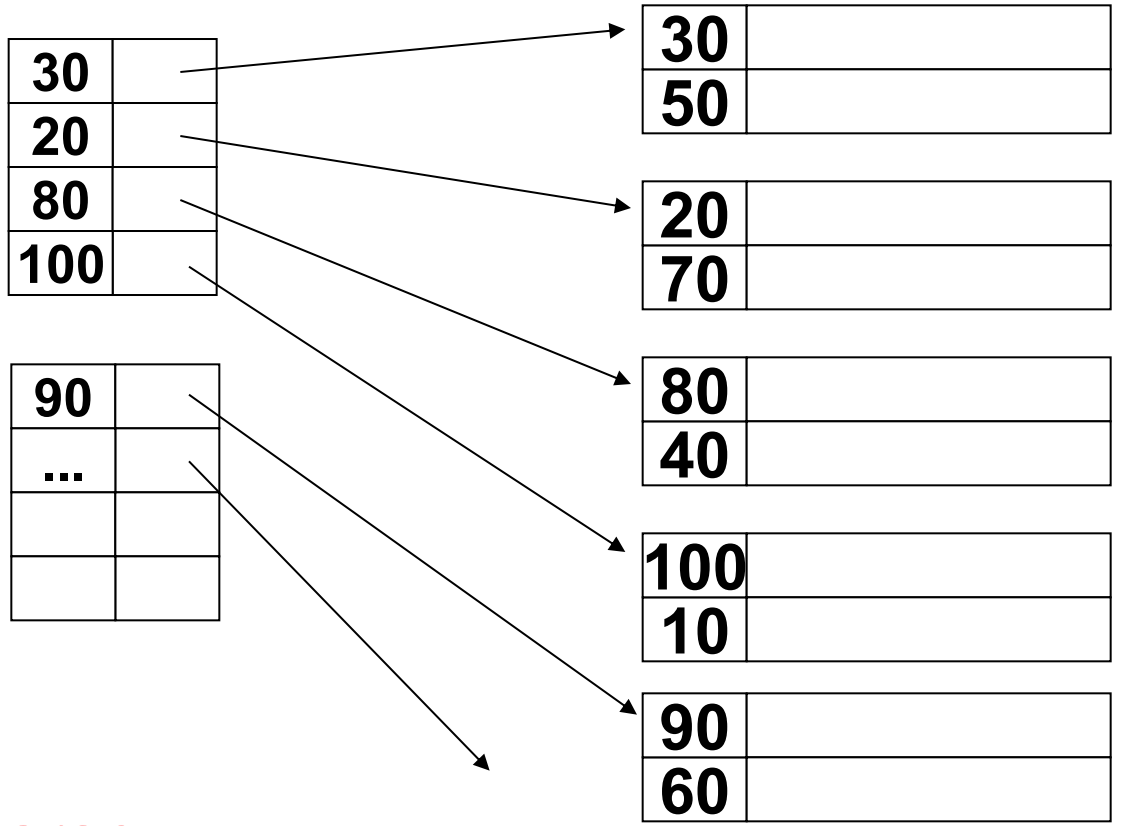
Come indicizzare una struttura disordinata ?



Scrittura disordinata

Indici secondari

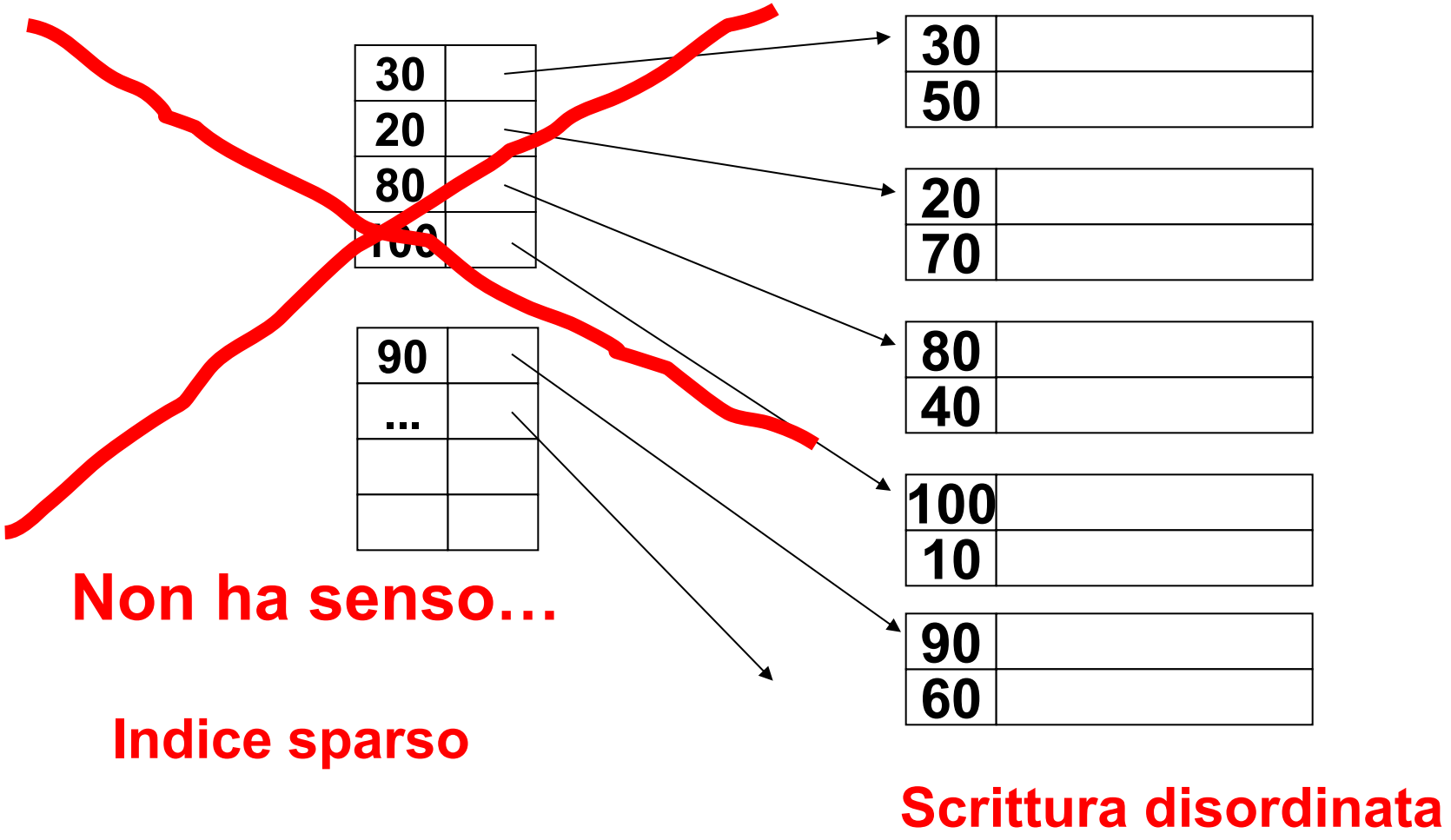
Buona scelta?



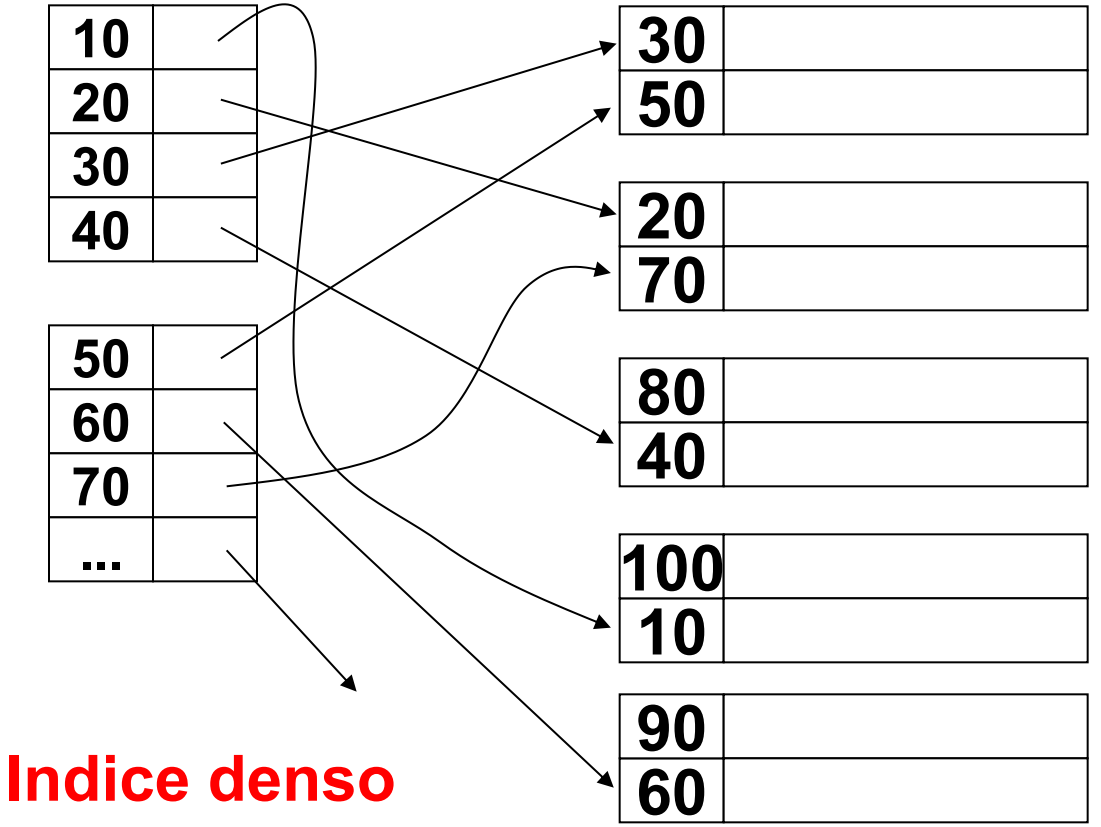
Indice sparso

Scrittura disordinata

Indici secondari



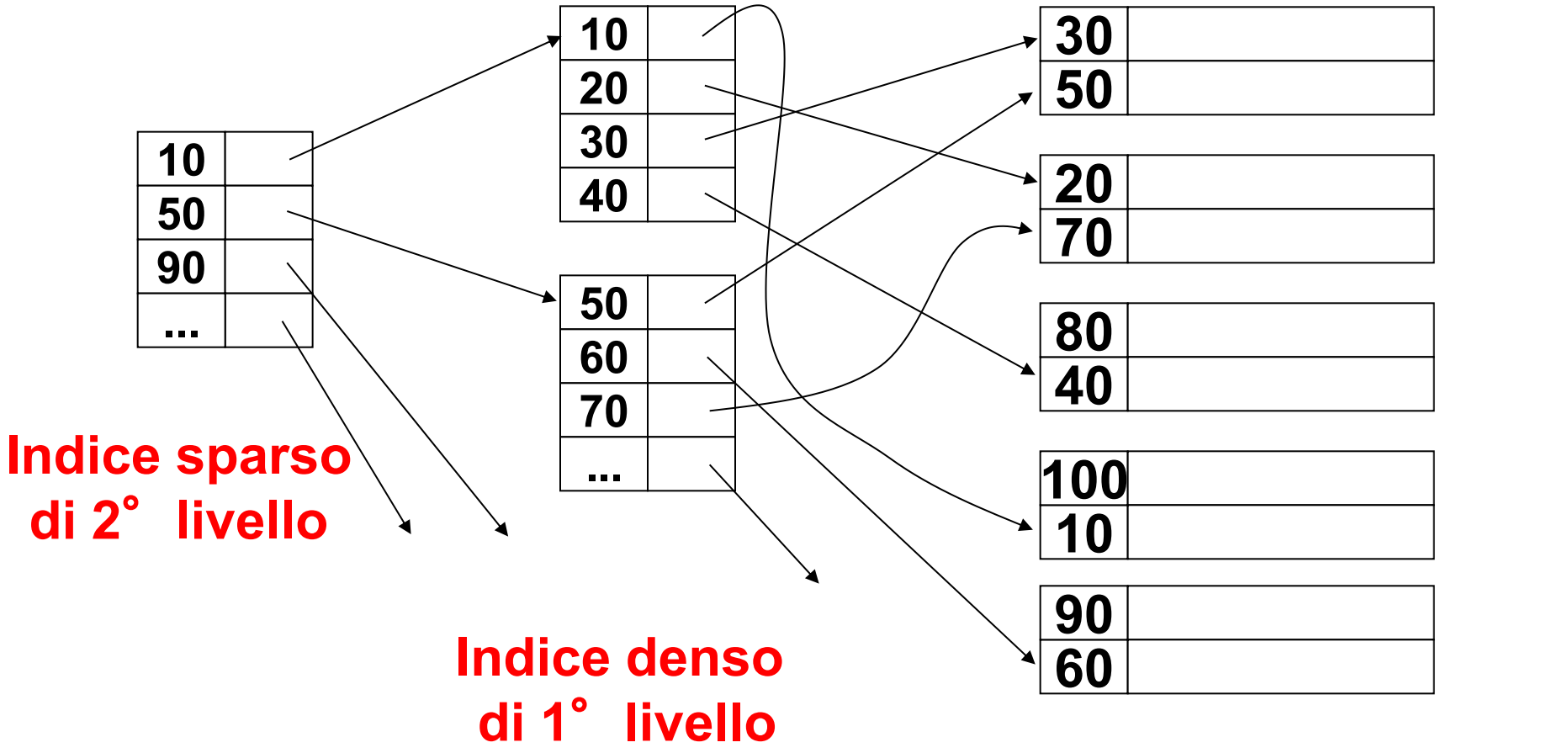
Indici secondari



Indice denso

Scrittura disordinata

Indici secondari



Caratteristiche degli indici

- Accesso diretto (sulla chiave) efficiente, sia puntuale sia per intervalli
  - E per le ricerche su altri campi (non chiave)?
- Modifiche della chiave, inserimenti, eliminazioni inefficienti (come in tutte le strutture ordinate)
- Tecniche per alleviare i problemi:
  - blocchi di overflow
  - marcatura per le eliminazioni
  - riempimento parziale
  - riorganizzazioni periodiche

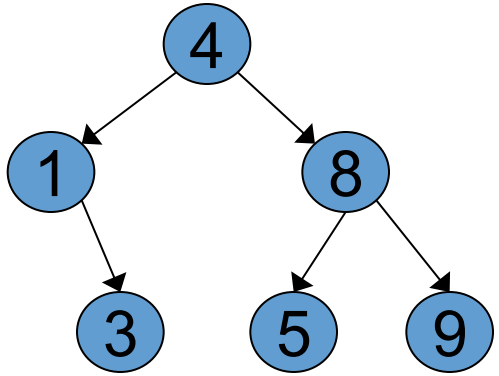
Indici, problemi

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

- Nei DBMS si usano indici ad-hoc, più sofisticati
  - Indici dinamici multilivello: B-tree (alberi di ricerca bilanciati)

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



- Tempo di ricerca e inserimento pari alla profondità:
  - logaritmico nel caso medio (assumendo un ordine di inserimento casuale)

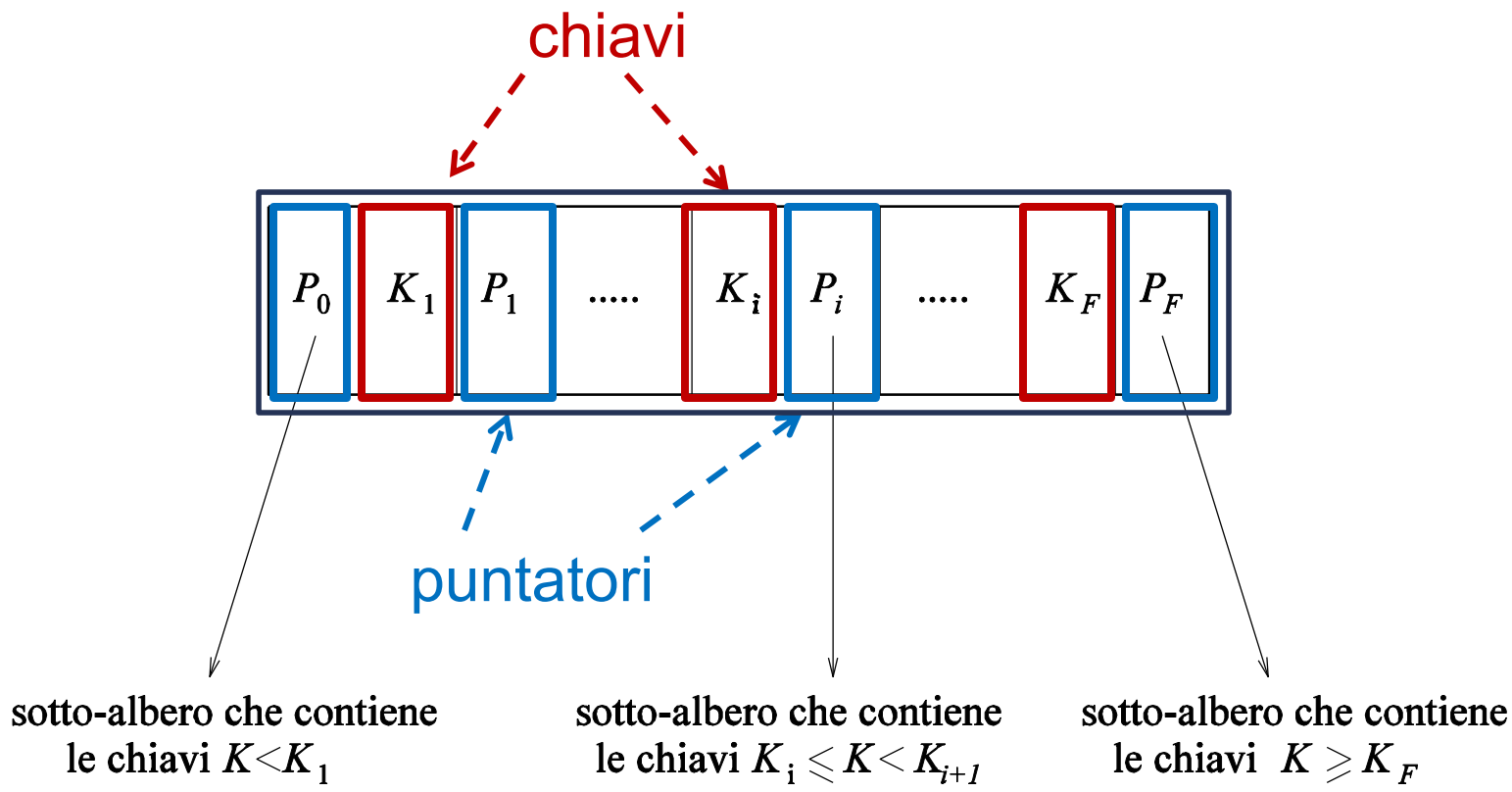
Albero di ricerca di ordine P

- 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 può corrispondere ad un blocco

Albero di ricerca di ordine P

- La struttura è ancora (potenzialmente) rigida
- Un B-tree è un albero di ricerca che viene mantenuto bilanciato grazie a:
  - Riempimento parziale (mediamente 70%)
  - Riorganizzazioni (locali) in caso di sbilanciamento

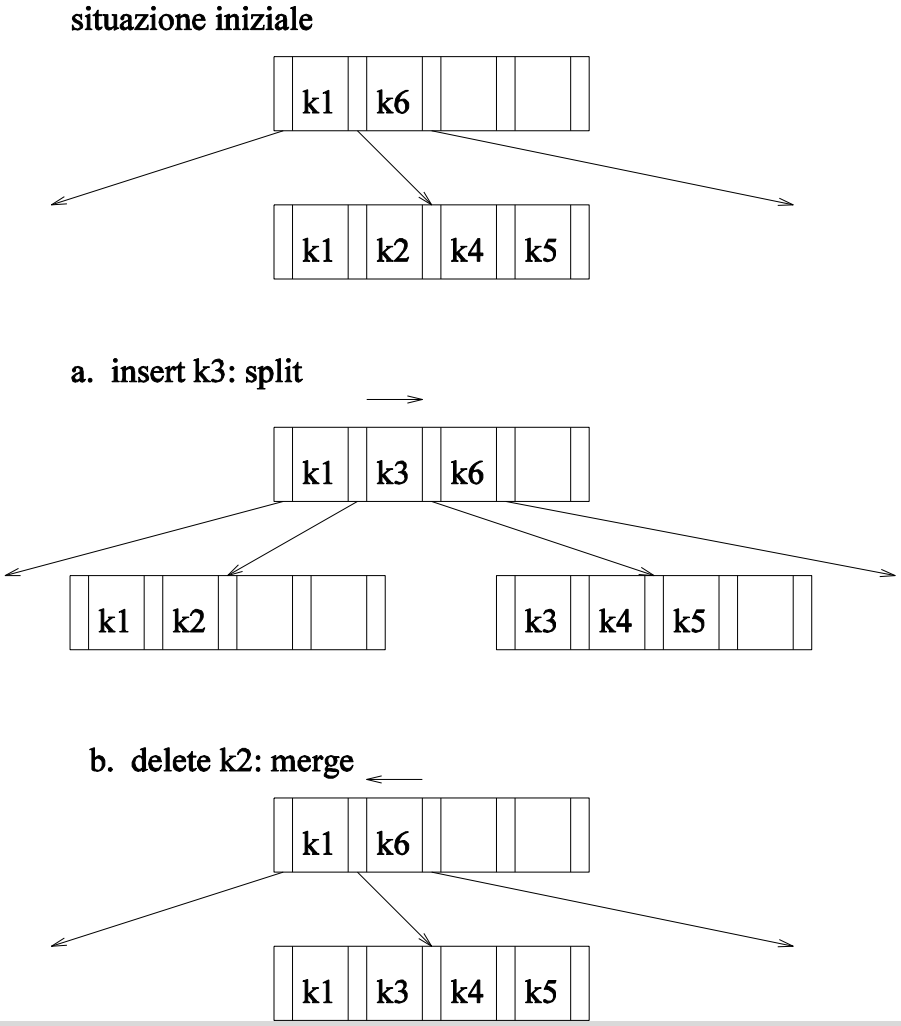
Organizzazione dei nodi del B-tree



Split e merge

- Inserimenti ed eliminazioni sono precedute da una ricerca fino ad una foglia
  - Se c'è posto nella foglia, si inserisce lì
  - Altrimenti il nodo va suddiviso, con un puntatore in più per il genitore
  - Se non c'è posto, si sale ancora
  - Il riempimento rimane sempre superiore al 50%

Split e merge



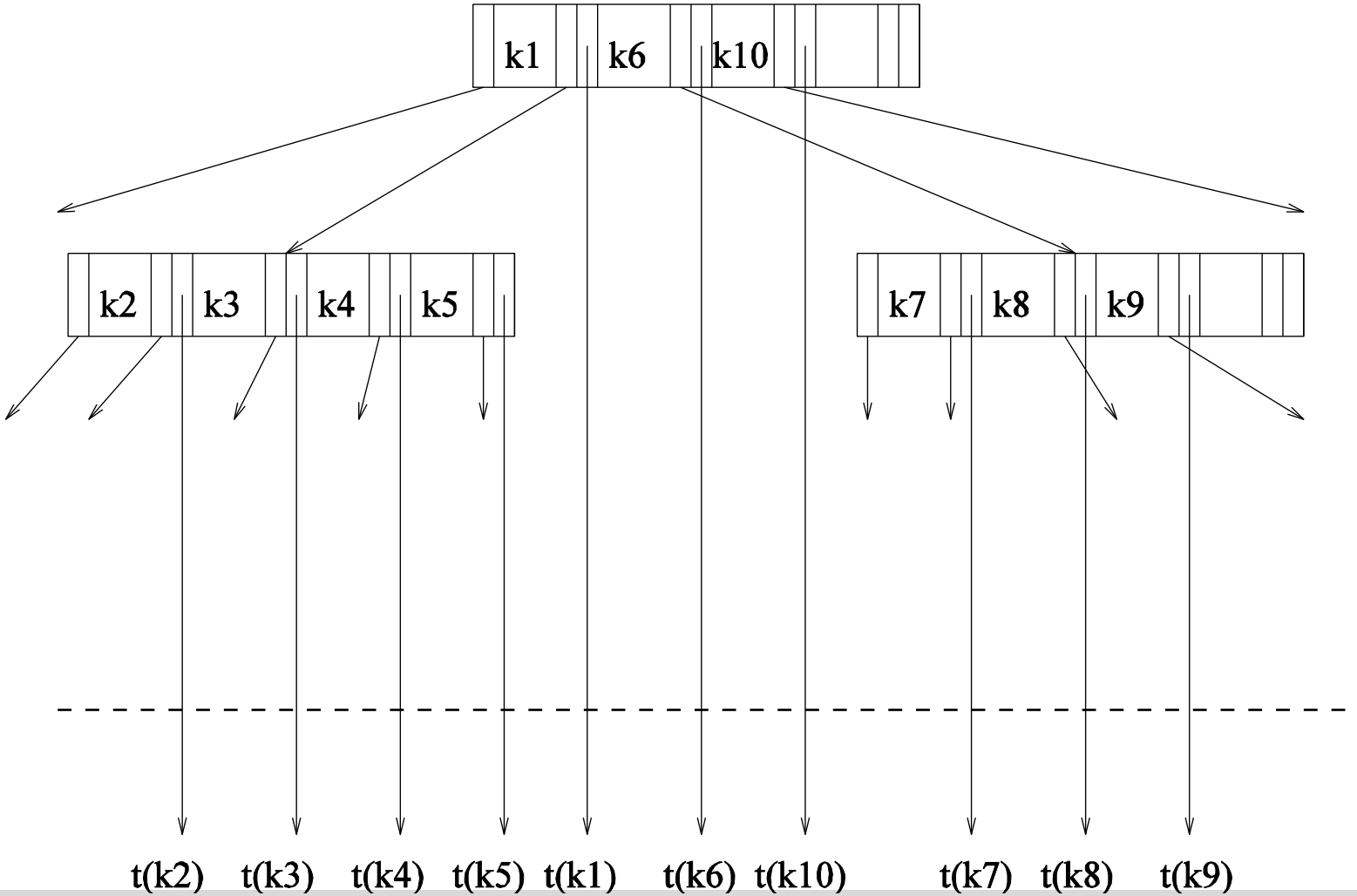
Split e merge

- Esempio
  - https://www.youtube.com/watch?v=coRJrclYbF4

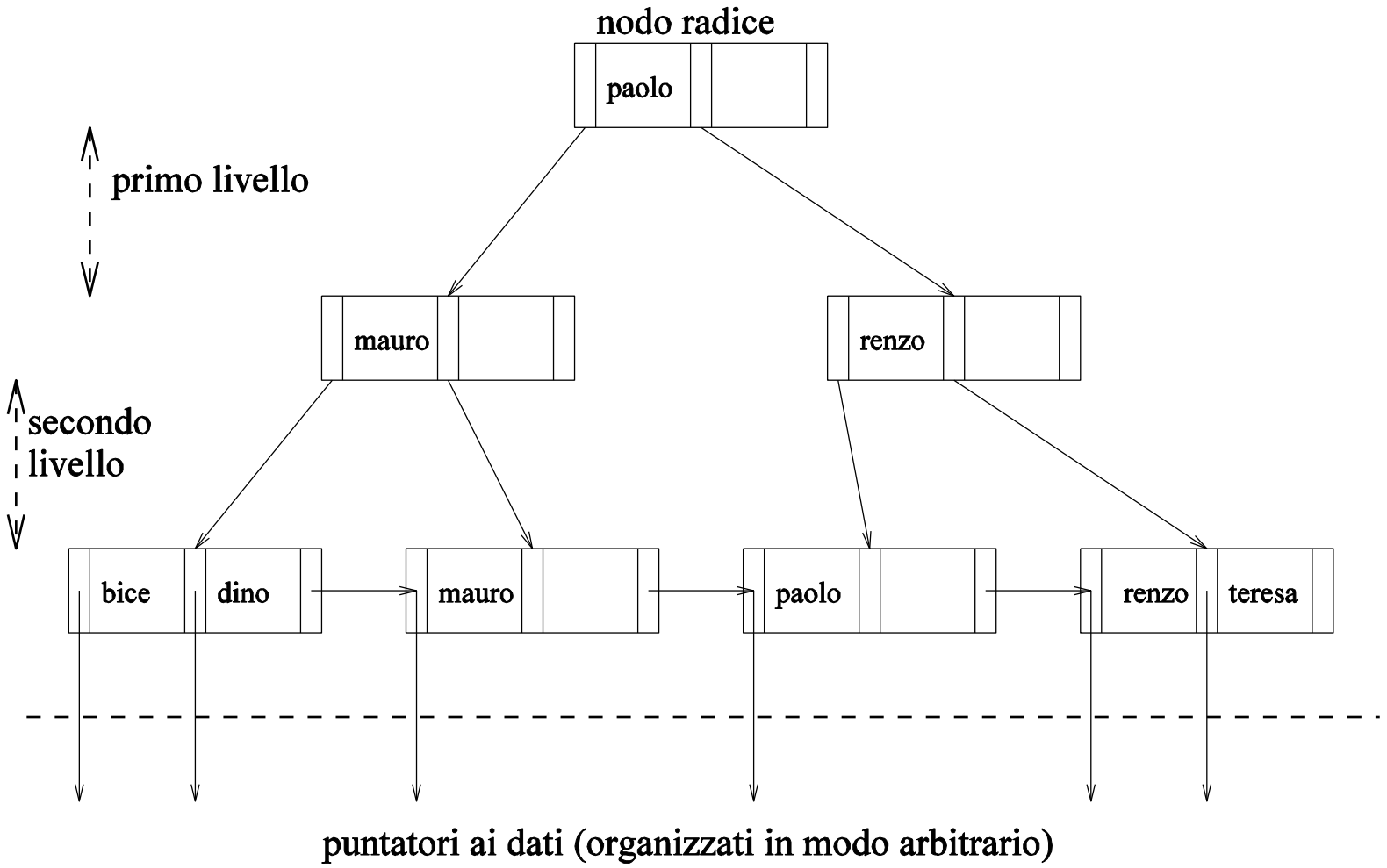
B tree e B+ tree

- B tree:
  - I nodi intermedi possono avere puntatori direttamente ai dati
- B+ tree:
  - le foglie sono collegate in una lista
  - ottimi per le ricerche su intervalli
  - molto usati nei DBMS

Un B-tree



Un B+ tree



Efficienza dei B-Tree

Molto efficienti quando il numero delle chiavi per blocco è grande

- Splitting e merging dei blocchi sono poco frequenti
  - E in genere limitati a piccole porzioni dell'albero
- Ricerca
  - Numero di letture = profondità dell'albero + 1 (per leggere il record su disco)
  - Profondità tipica: 3 o 4 (eccetto DB molto grandi)

Esercizio

- Assumiamo che:
  - Un blocco possa contenere 10 record oppure 99 chiavi e 100 puntatori
  - In media, un nodo del B-tree è pieno per il 70% full: avrà quindi 69 chiavi e 70 puntatori
  - Inizialmente la RAM è vuota
  - La chiave di indicizzazione è la chiave primaria della tabella
- Determinare:
  - (i) numero totale di blocchi necessari per memorizzare 1.000.000 di record della tabella e il corrispondente indice B-tree
  - (ii) il numero medio di letture necessarie per recuperare un record data la sua chiave
- Caso a) I dati sono memorizzati in modo ordinato in base alla chiave, con 10 record per blocco. Il B-tree è denso.

Soluzione caso a)

- Servono 100.000 blocchi per i dati. Essendoci in media 70 puntatori per blocco nelle foglie, allora ci servono 1.000.000/70 = 14.286 blocchi per le foglie del B-tree. Al livello superiore, serve 1/70imo dei blocchi delle foglie (204 blocchi), e al terzo livello ne serve 1/70esimo di quelle (3 blocchi). Il quarto livello è il blocco della radice. In totale servono 100.000 + 14.286 + 205 + 3 + 1 = 114.495 blocchi.

- Poichè il B-tree ha 4 livelli, servono 5 letture per ottenere il record cercato.

Hashing, processing, e ottimizzazione delle query

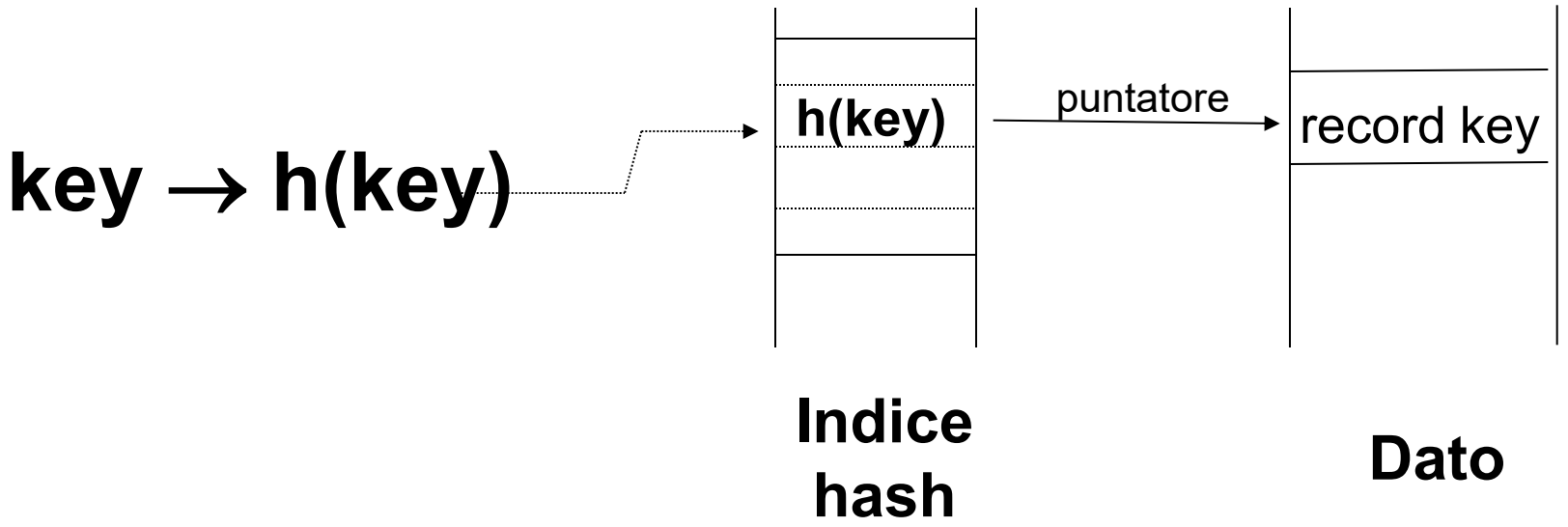
Strutture ad accesso calcolato (hashing)

- Obiettivo: accesso diretto ad un insieme di record sulla base del valore di un campo

- Il campo è detto impropriamente “chiave”
  - Anche se non è detto sia identificante

- Funzione hash
  - associa ad ogni valore della chiave un "indirizzo", in uno spazio di dimensione leggermente superiore al necessario

Hashing



Funzione hash

- Funzione **h : C → I**
- C = insieme delle possibili chiavi
- I: insieme degli indirizzi

- Cardinalità(I) << Cardinalità(C)

- Una buona funzione hash deve distribuire i valori delle chiavi in I in modo uniforme

Collisioni

- Funzione hash:
  - il numero di possibili chiavi è molto maggiore del numero di possibili indirizzi

- La funzione hash non può essere iniettiva
  → esiste la possibilità di **collisioni**
  (chiavi diverse che corrispondono allo stesso indirizzo)

Esempio di collisioni

- Funzione hash h:

  h(k) = k mod 50

- 40 record

- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

Collisioni

- Come ridurre la probabilità di collisione?
  - Usando una “buona” funzione hash (che distribuisca in modo causale e uniforme)
  - Aumentando lo spazio ridondante
- Come affrontare le collisioni?
  - Posizioni successive disponibili
  - Tabella di overflow

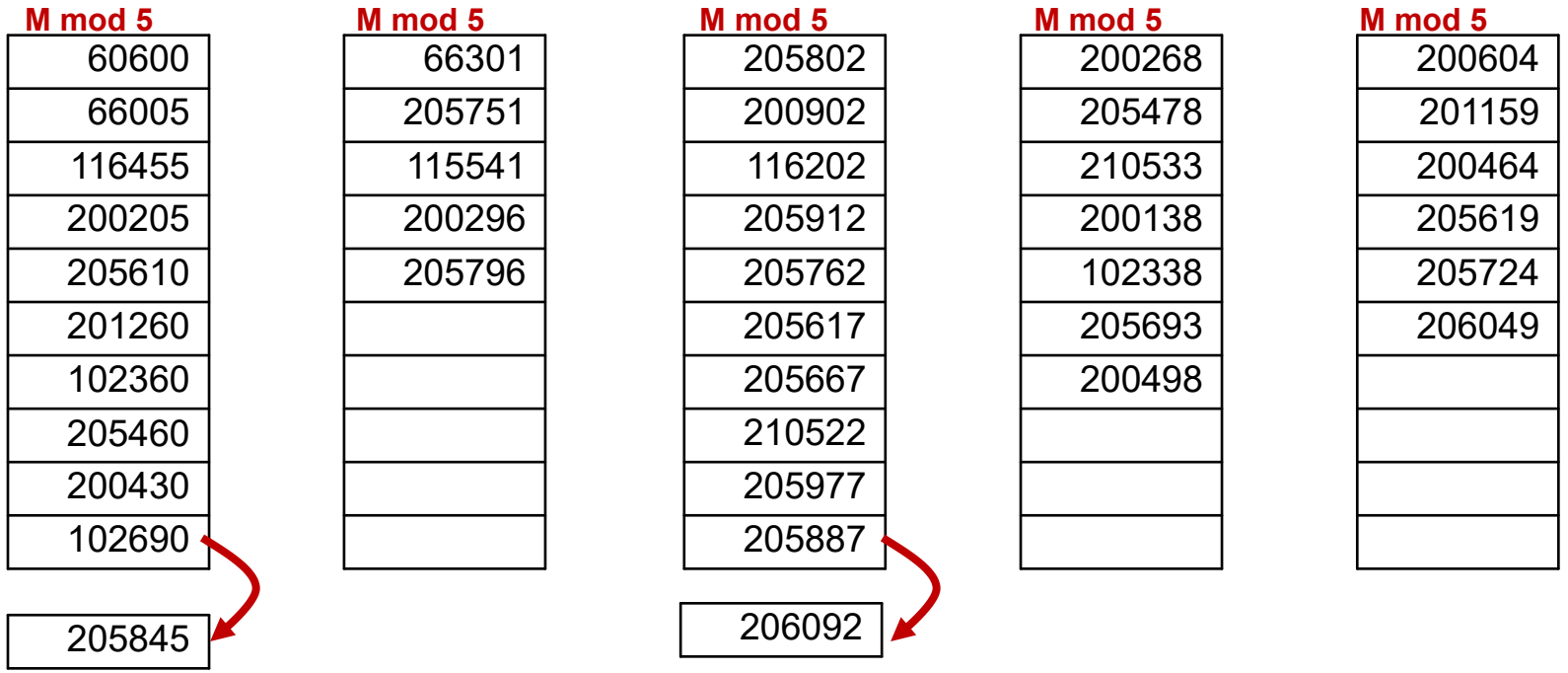
Un esempio

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - Se 1 blocco = 1 record: numero medio di accessi: 1,425

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

File hash con fattore di blocco 10; 5 blocchi con 10 posizioni ciascuno



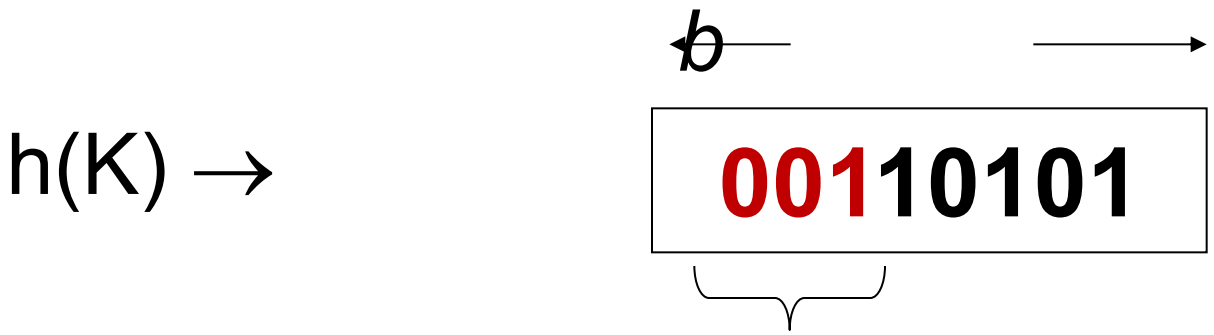
40 record - tavola hash con 50 posizioni:
file hash con fattore di blocco 10
5 blocchi con 10 posizioni ciascuno:
Due soli overflow - numero medio di accessi: 1,05

Come gestire la crescita di dimensione dei dati?

- Usare overflows e riorganizzazioni periodiche
- Hashing dinamico

Dynamic (extensible) hashing

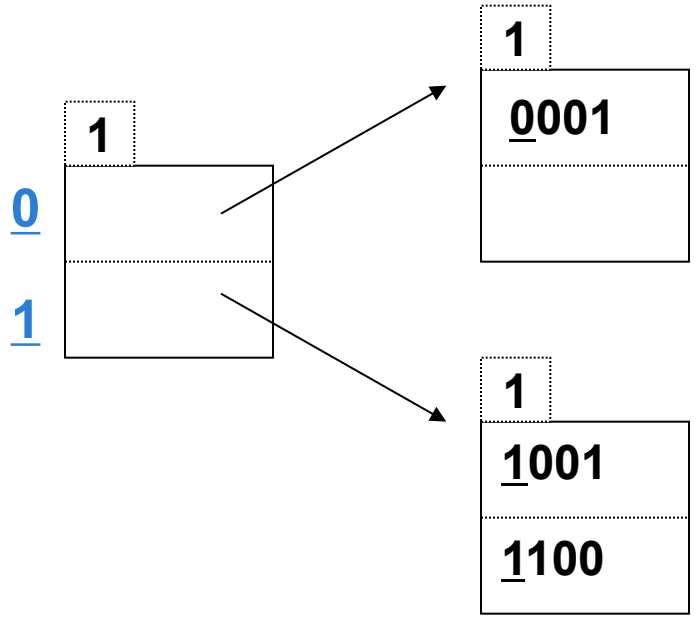
Invece di usare tutti i *b* bit di outpud della funzione di hash,
usiamo solo i primi *i*



la dimensione di *i* cresce col tempo…

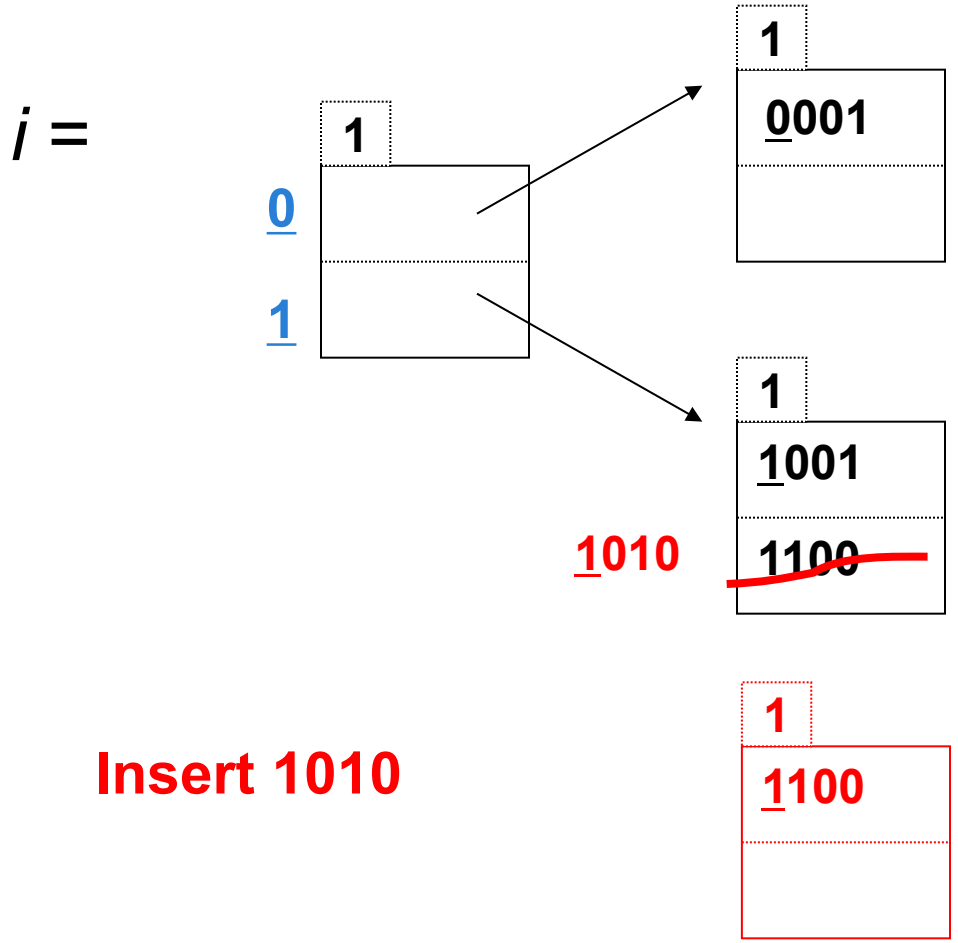
Esampio: h(k) restituisce 4 bit; 2 chiavi per blocco

i =

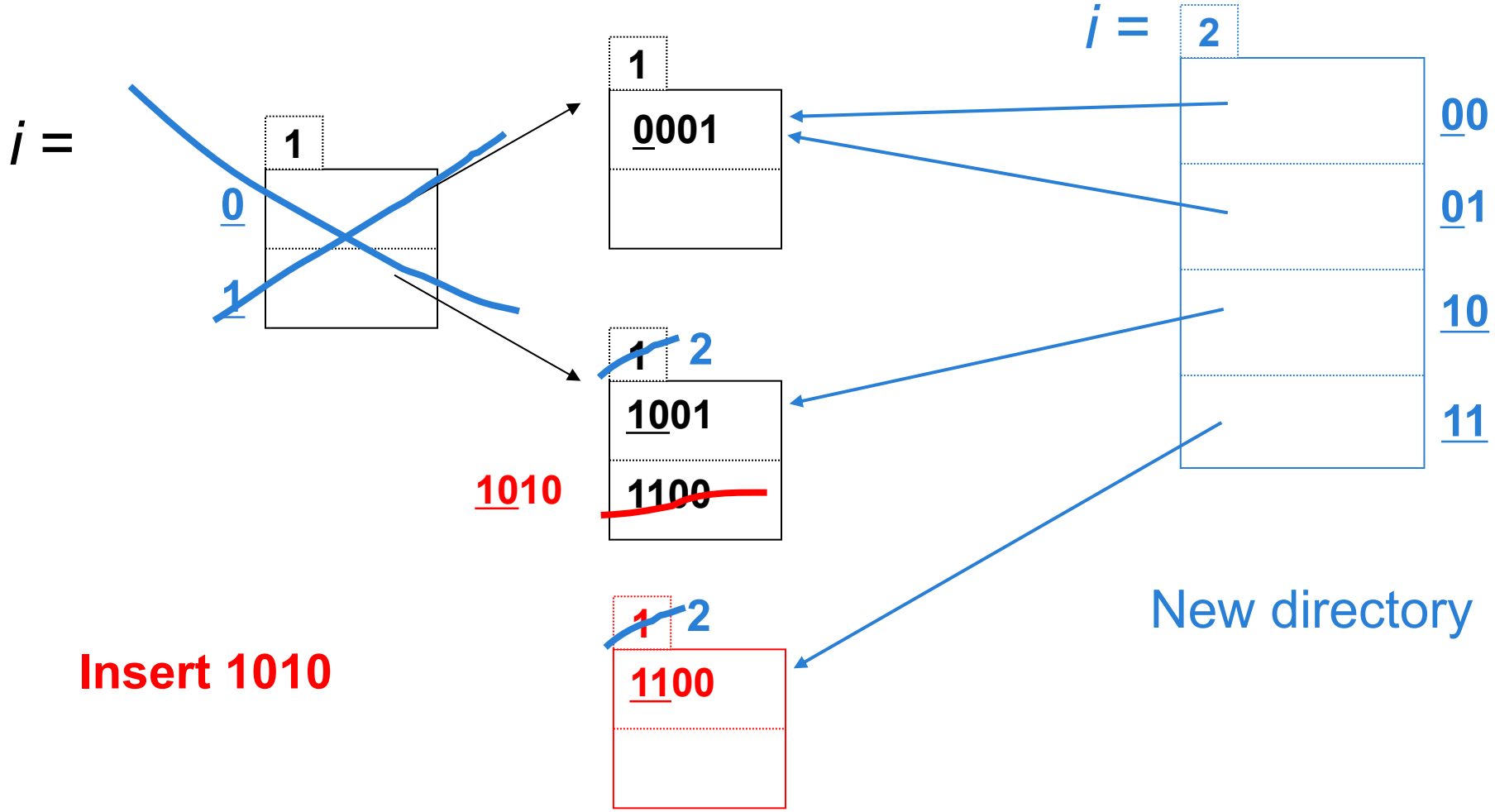


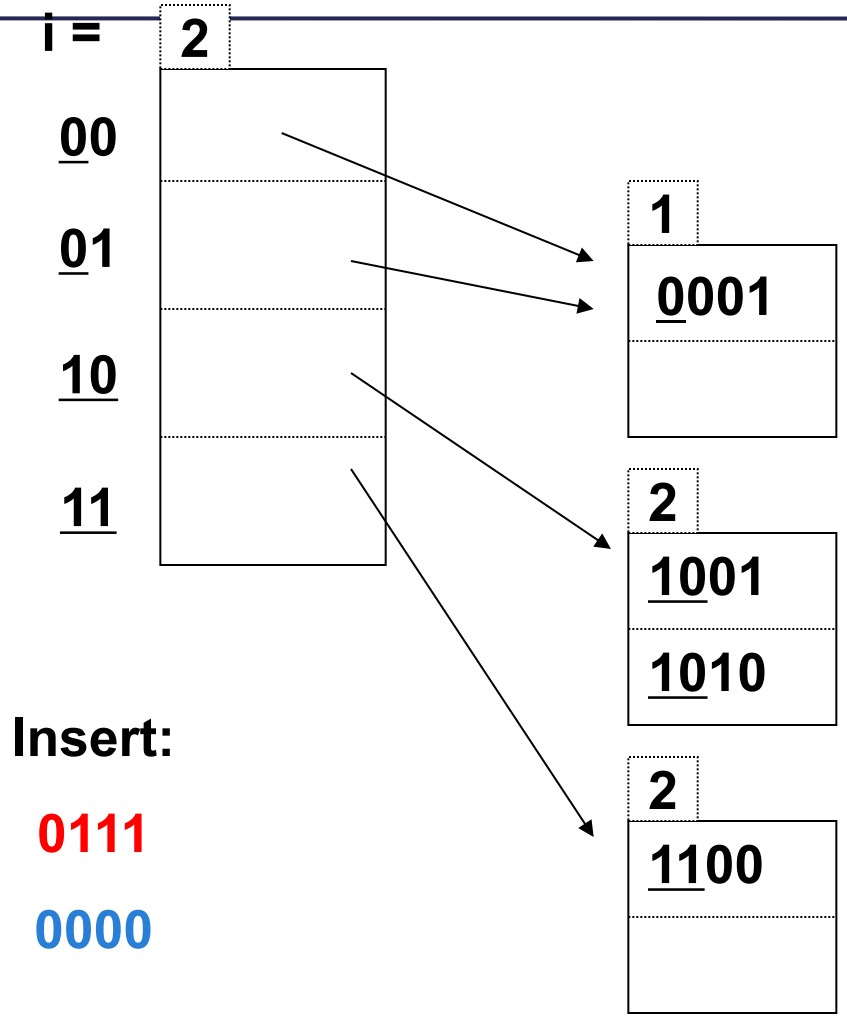
Insert 1010

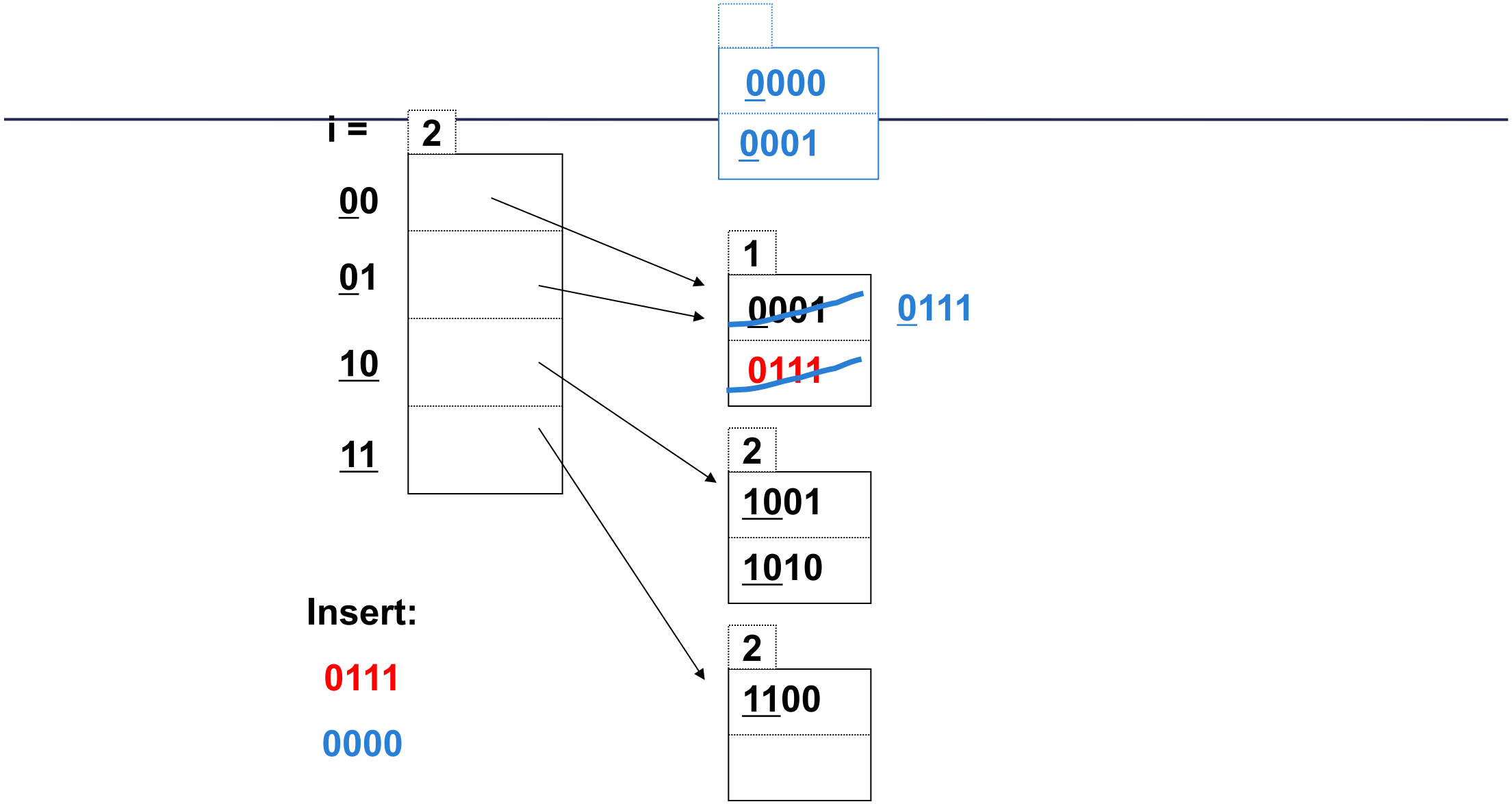
Esampio: h(k) restituisce 4 bit; 2 chiavi per blocco

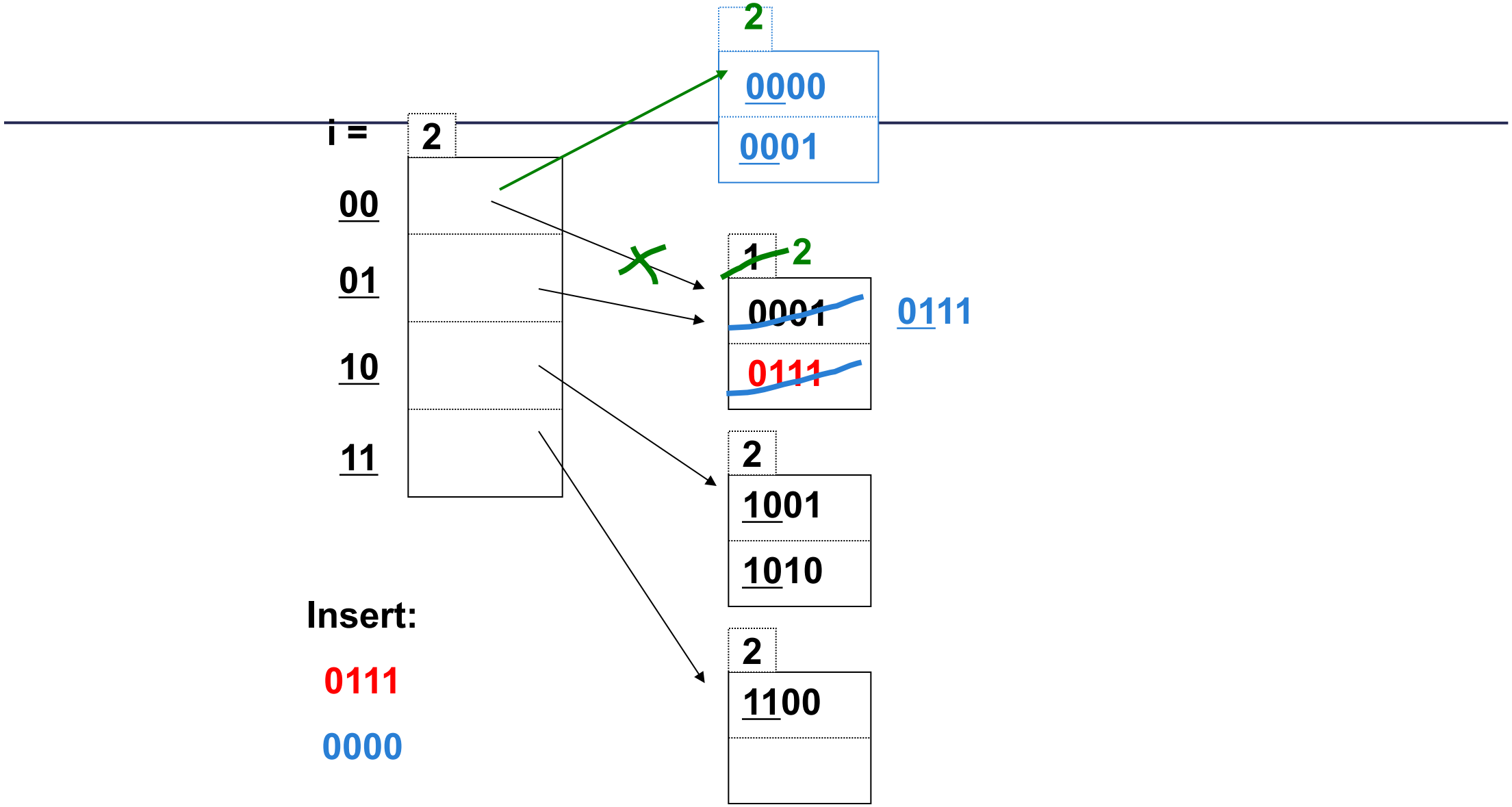


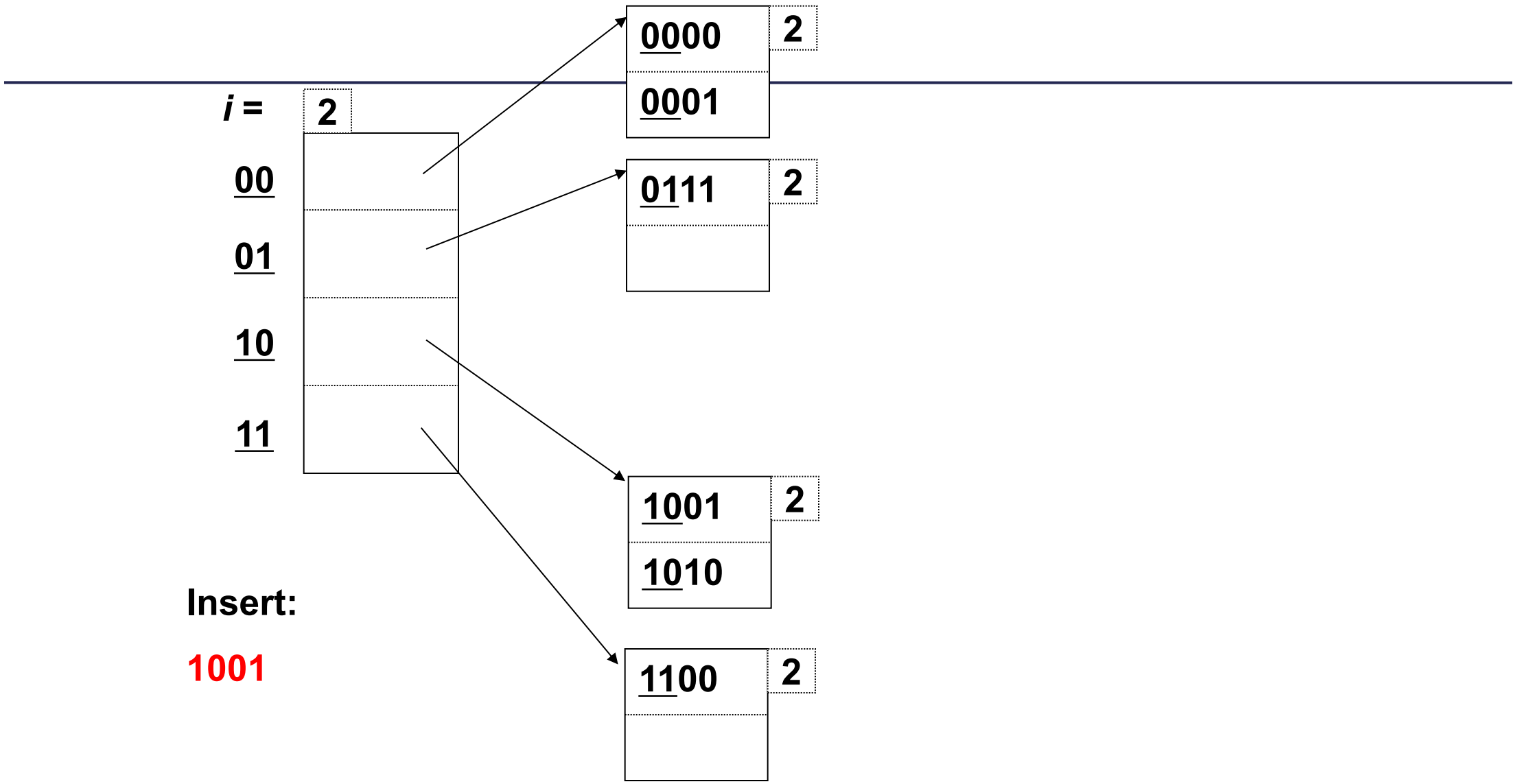
Esampio: h(k) restituisce 4 bit; 2 chiavi per blocco

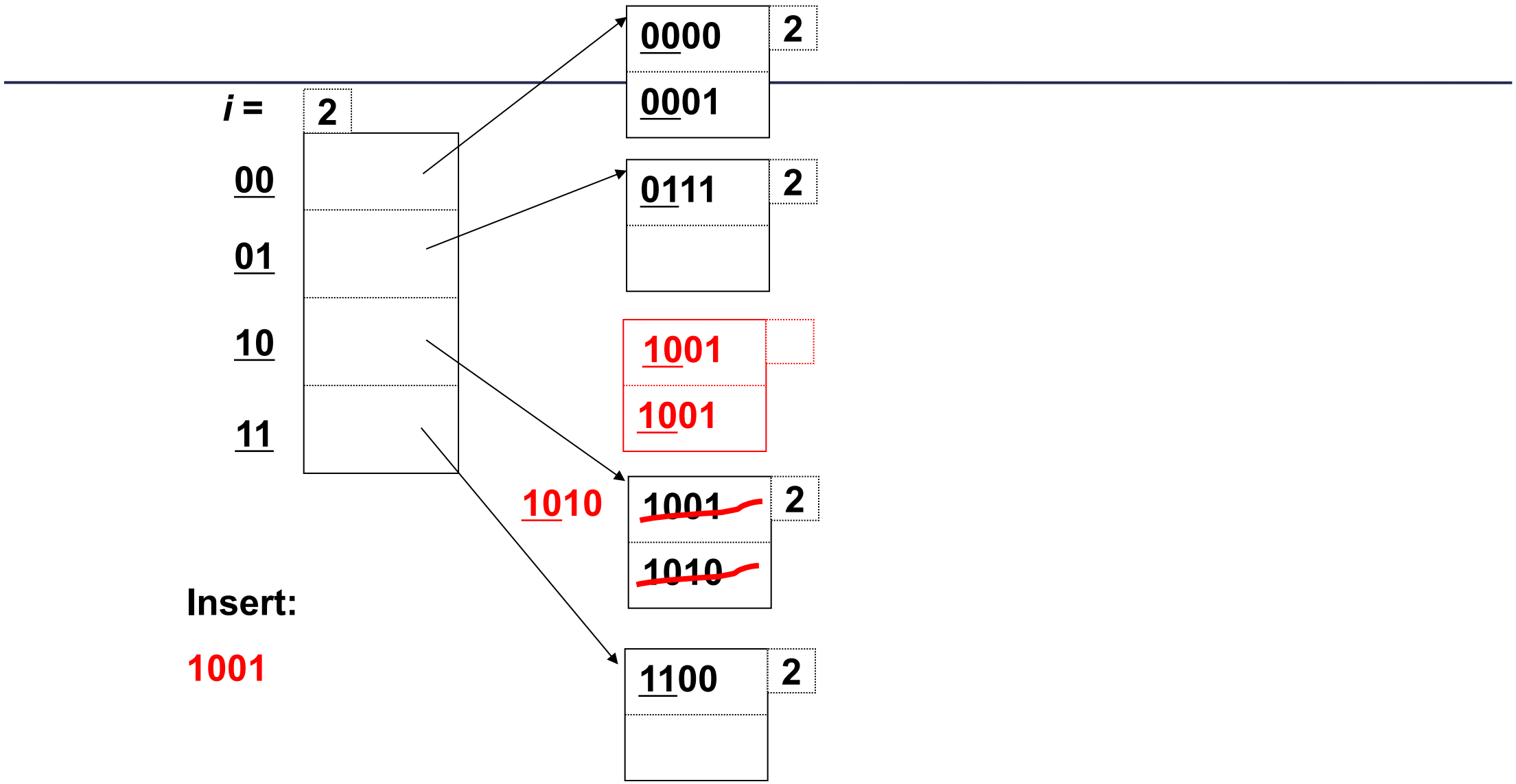


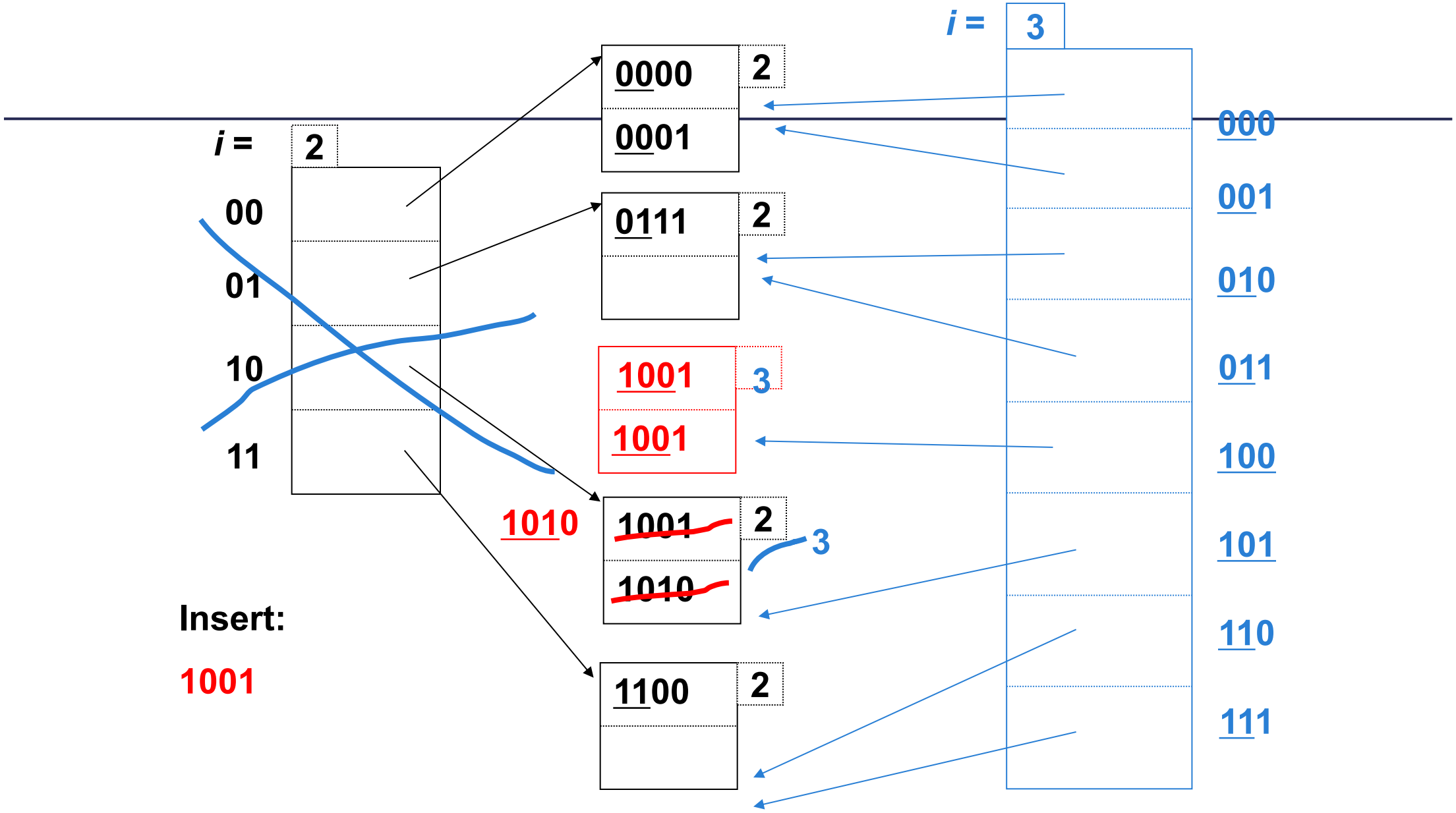

i = 2
00
01
10
11
1
0001
2
1001
1010
2
1100
Insert:
0111
0000


0000
0001
i =
2
00
01
10
11
1
0001
0111
0111
2
1001
1010
2
1100
Insert:
0111
0000


2
0000
0001
i =
2
00
01
10
11
1
2
0001
0111
0111
2
1001
1010
2
1100
Insert:
0111
0000


i =
2
00
01
10
11
0000
0001
2
0111
2
1001
1010
2
1100
2
Insert:
1001


i =
2
00
01
10
11
0000
2
0001
0111
2
1001
1001
1010
1001
2
1010
1100
2
Insert:
1001


i = 3
i = 2
00
01
10
11
0000
0001
2
0111
2
1001
1001
3
1001
1010
2
3
1100
2
000
001
010
011
100
101
110
111
1010
Insert:
1001

File hash, osservazioni

- È l'organizzazione più efficiente per l'accesso diretto basato su valori della chiave con condizioni di uguaglianza
- Non è efficiente per ricerche basate su intervalli
- Costo medio di poco superiore all'unità
- Il caso peggiore è molto costoso ma improbabile

Indice hash

- Considera la tabella Persona(ID, Nome, …)
- Si crea un indice hash (non dinamico) su ID, usando una funzione H con codominio a 2 byte
- Un blocco ha dimensione 4.096 byte
- Il sistema usa puntatori a 4 byte
- Di quanti blocchi avrò bisogno per memorizzare l'indice hash di Persona.ID? Motivare la risposta

Indice hash - soluzione

- L'hash di ID occupa 2 byte, il puntatore ne occupa 4, quindi la coppia (H(ID),puntatore) occupa 6 byte
- H ha codominio a 2 byte, quindi i valori possibili di H(ID) sono 2^16 = 65.536
- In totale l'indice occupa 65.536 * 6 = 393.216 byte
- Occorrono quindi 393.216 / 4.096 = 96 blocchi per memorizzare l'indice hash

Indice hash

- Consideriamo l'indice hash precedente.
- Supponiamo che un blocco possa contenere in media 10 record di Persona.

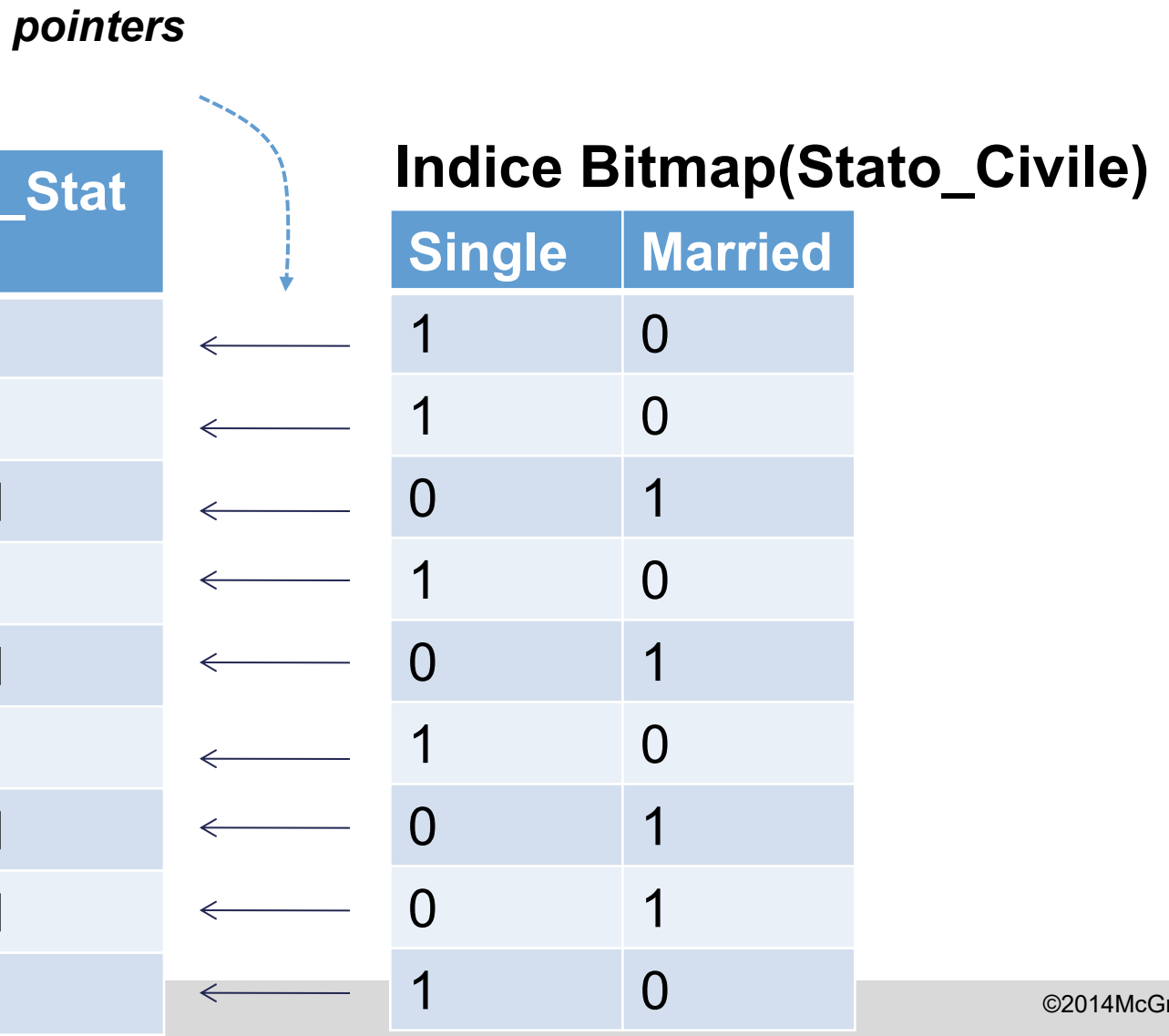
- L'indice che usiamo è una buona scelta se Persona contiene 100.000 record? E se contiene 1.000.000 di record?

Indice Bitmap

Adatto per colonne con un basso numero di possibili valori diversi

- Stato civile: Single / Sposato
- Range d'età: 0-17 / 18-34 / 35-49 / 50-65 / 65+
- Regione: Sardegna / Sicilia / Lombardia / ...

Indice Bitmap



Indice Bitmap

Numero di colonne = numero di valori differenti

Tabella

Person_ID	Age_Range
1001	18-34
1002	65+
1003	65+
1004	50-65
1005	35-49
1006	35-49
1007	65+
1008	35-49
1009	50-65

Indice Bitmap (Age_Range)

18-34	35-49	50-65	65+
1	0	0	0
0	0	0	1
0	0	0	1
0	0	1	0
0	1	0	0
0	1	0	0
0	0	0	1
0	1	0	0
0	0	1	0

Indice Bitmap

"Trova tutti gli uomini anziani non sposati"

Tabella

Person_ID	Age_Range
1001	18-34
1002	65+
1003	65+
1004	50-65
1005	35-49
1006	35-49
1007	65+
1008	35-49
1009	50-65

Indice Bitmap (varie colonne)

18-34	65+	Married	Male
1	0	0	1
0	1	0	1
0	1	1	0
0	0	0	1
0	0	1	0
0	0	0	1
0	1	1	0
0	0	1	0
0	0	0	1

Indice Bitmap

- Efficiente per query su molti campi, ognuno con pochi valori possibili

- Non adatto per dati molto dinamici (insert/update/delete frequenti)

  perchè una operazione su un dato può "bloccare" migliaia di altre operazioni su altri dati della stessa tabella *(scritture concorrenti sull'indice bitmap)*

Indice bitmap

- Considera la tabella Persona(ID, …, Regione)
- Il campo Regione ha 20 valori possibili
- Si crea un indice bitmap su Regione

- Un blocco ha dimensione 4.096 byte
- Il sistema usa puntatori a 4 byte
- Persona contiene 100.000 record

- Di quanti blocchi avrò bisogno per memorizzare l'indice bitmap di Persona.Regione?

Indice bitmap - soluzione

- Per ogni record, l'indice memorizzerà un puntatore al record (4 byte = 32 bit), e il valore corrispondente di Regione
- Regione ha 21 valori possibili: 20 + *NULL*. Servono quindi 5 bit (2^5 = 32) per memorizzare il valore di Regione del record
- In totale, per ogni record, servono 37 bit, quindi 100.000 * 37 bit = 3.700.000 bit = 462.500 byte
- Data la dimensione dei blocchi, servono 462.500 / 4.096 = 113 blocchi

Definizione degli indici in SQL

- Non è standard, ma presente nei vari DBMS
  - create [unique] index IndexName
    on TableName(AttributeList)
  - drop index IndexName
  - *unique* specifica che AttributeList è una superchiave

- Esempio:
  - create index NomeIndice
    on Impiegato(Cognome, Citta)

Linee guida per gli indici

- Creare un indice se si vuole *frequentemente* recuperare *meno del 15%* delle righe di una *grande* tabella
- Indicizzare le colonne usate per i join

  (le chiavi primarie e i campi "unique" hanno automaticamente indici, ma può essere utile creare indici per le chiavi esterne)
- Indicizzare le colonne i cui valori sono relativamente univoci (eccetto che per gli indici bitmap)

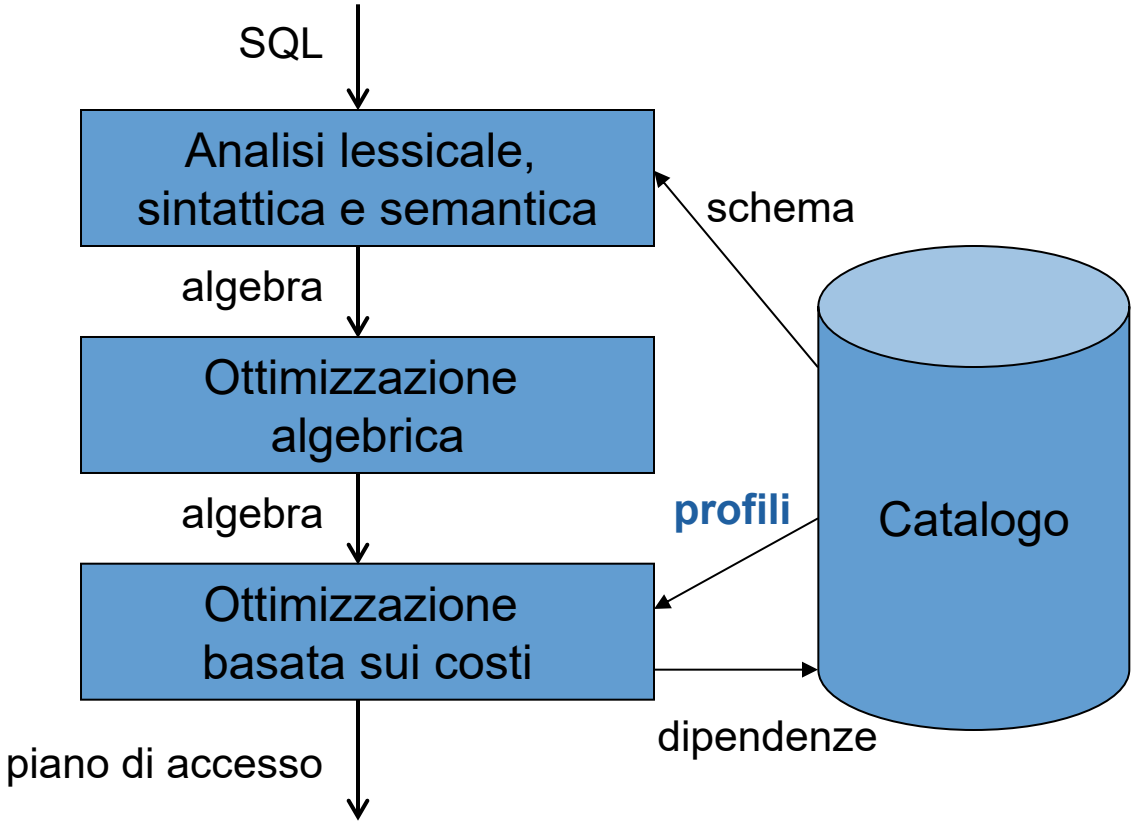
Esecuzione e ottimizzazione delle query

- Query processor (o ottimizzatore): un modulo del DBMS

- Le query SQL sono espresse ad alto livello
  - SQL è (di base) un linguaggio dichiarativo
  - L'algebra relazionale offre un approccio procedurale

- L'ottimizzatore sceglie la strategia realizzativa a partire dall'istruzione SQL

Esecuzione delle query



Ottimizzazione algebrica

- Il termine ottimizzazione è improprio perché il processo utilizza euristiche

- Si basa sulla nozione di equivalenza:
  - Due espressioni sono equivalenti se producono lo stesso risultato qualunque sia l'istanza attuale della base di dati

- I DBMS cercano di eseguire espressioni equivalenti a quelle date, ma meno "costose”

Ottimizzazione algebrica

- Euristica fondamentale:

  selezioni e proiezioni il più presto possibile

  (per ridurre le dimensioni dei risultati intermedi):

- "push selections down"
- "push projections down"

"Push selections down"

- Assumiamo A attributo di R2
- SEL A=10 (R1 JOIN R2)

  ⇔

  R1 JOIN (SEL A=10 (R2))
- La seconda forma riduce in modo significativo la dimensione del risultato intermedio

  (e quindi il costo dell'operazione)

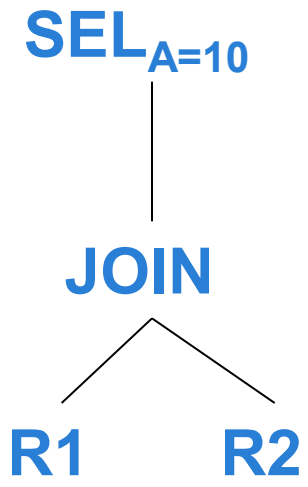
Rappresentazione interna delle query

- Tramite alberi:
  - foglie: dati
  - nodi intermedi: operatori

    (operatori algebrici, poi effettivi
    operatori di accesso)

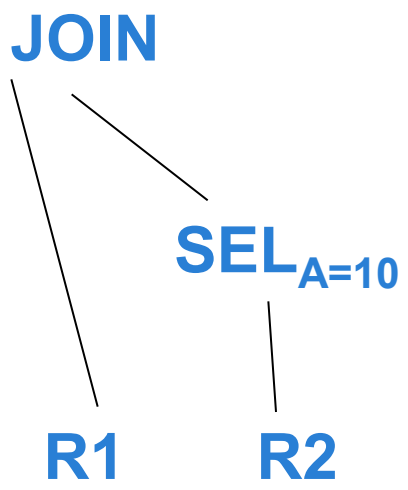
Alberi per la rappresentazione di query

$SEL_{A=10}(R_1 \text{ JOIN } R_2)$



Più efficiente

$R_1 \text{ JOIN } (SEL_{A=10}(R_2))$



Esecuzione delle operazioni

- I DBMS implementano gli operatori dell'algebra per mezzo di operazioni di basso livello

  che possono implementare vari operatori "in un colpo solo“

- Operatori fondamentali:

  Scansione e accesso diretto

- A livello più alto:

  Ordinamento e join

Accesso diretto e scansione

- Accesso diretto possibile solo se uso:
  - Indici (B-tree...)
  - Strutture hash
- Scansione efficiente se uso:
  - B+ tree

Join

- R1 JOIN R2

- L'operazione più costosa (e molto frequente)

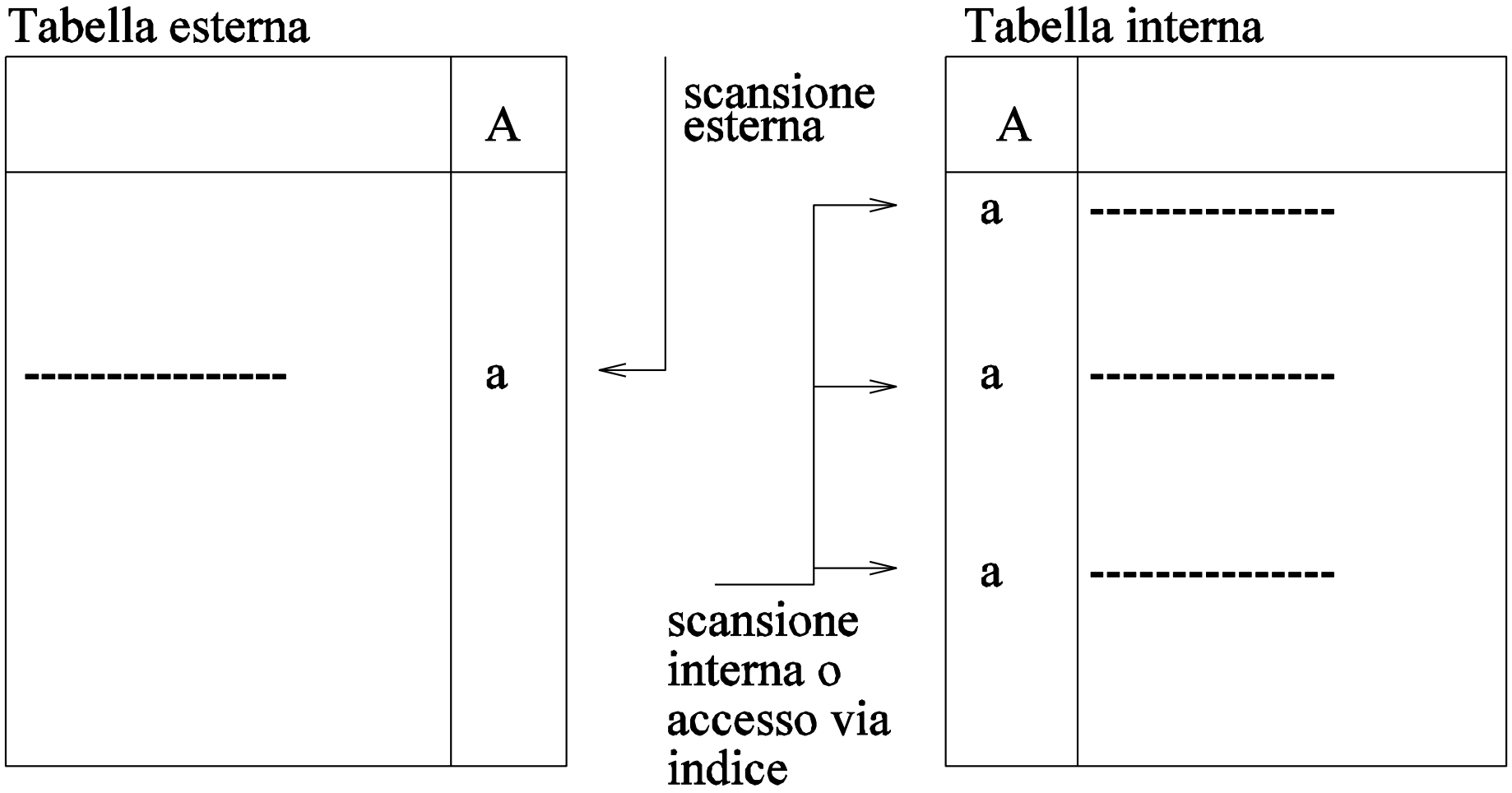
- Vari metodi; i più noti:
  1. Nested-loop (chiamato anche “iteration join”)
  2. Merge-join (chiamato anche “merge scan”)
  3. Hash-join

1) Nested-loop

- (concettualmente) R1 JOIN R2 ON R1.C = R2.C:

```
for each r ∈ R1 do
    for each s ∈ R2 do
        if r.C = s.C then output <r,s> pair
```

1) Nested-loop



2) Merge-join

- (concettual.) R1 JOIN R2 ON R1.C = R2.C:

  (1) Ordinare R1 e R2 su C, o indicizzarli con B+ tree

  (2) i ← 1; j ← 1;

  while (i ≤ #T(R1)) ∧ (j ≤ #T(R2)) do

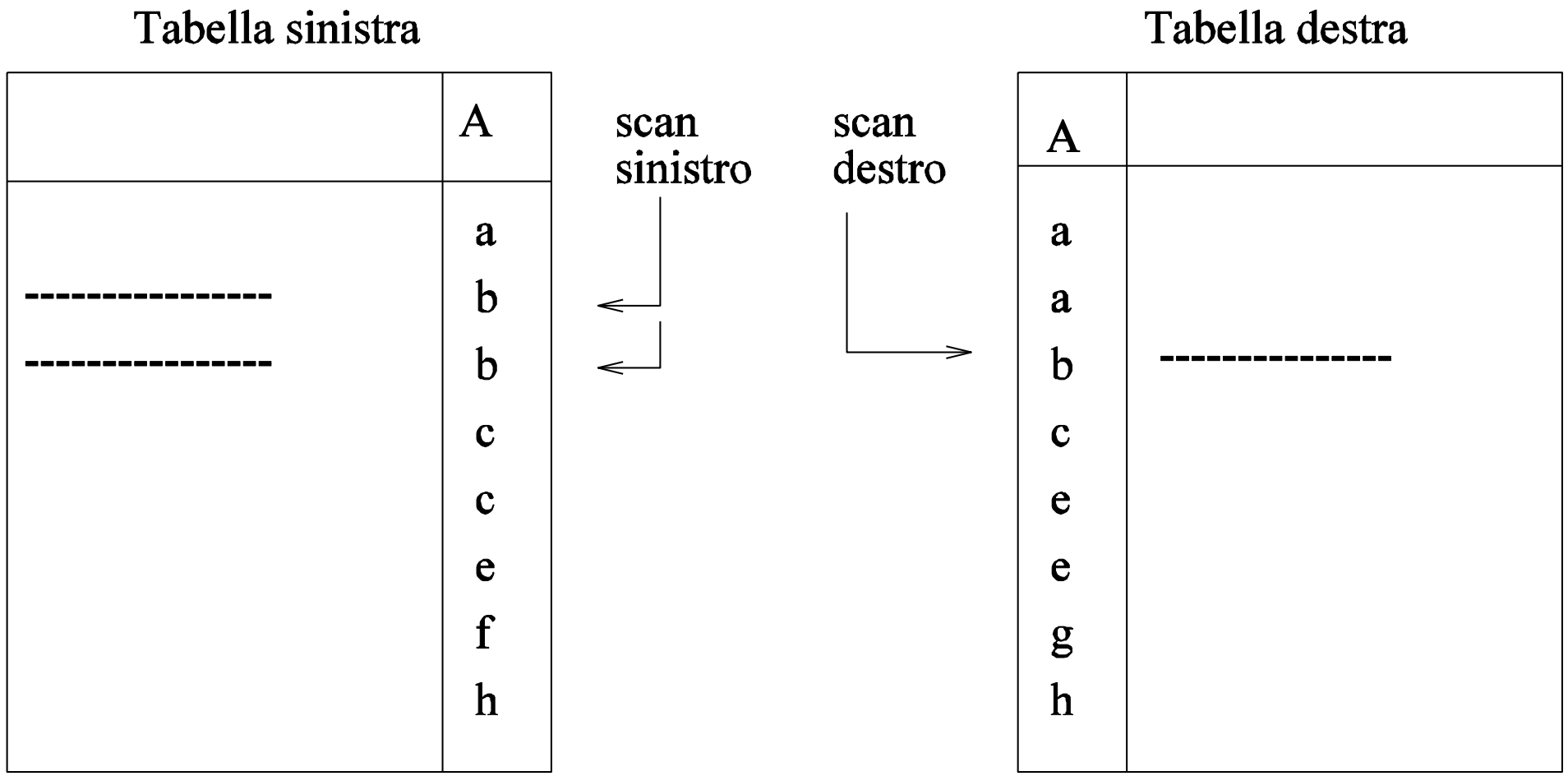
  if R1{ i }.C = R2{ j }.C then {

  outputTuples(R1{ i }.C) ; i ← i+1 ; j ← j+1 }

  else if R1{ i }.C > R2{ j }.C then { j ← j+1 }

  else if R1{ i }.C < R2{ j }.C then { i ← i+1 }

2) Merge-join



3) Hash-join

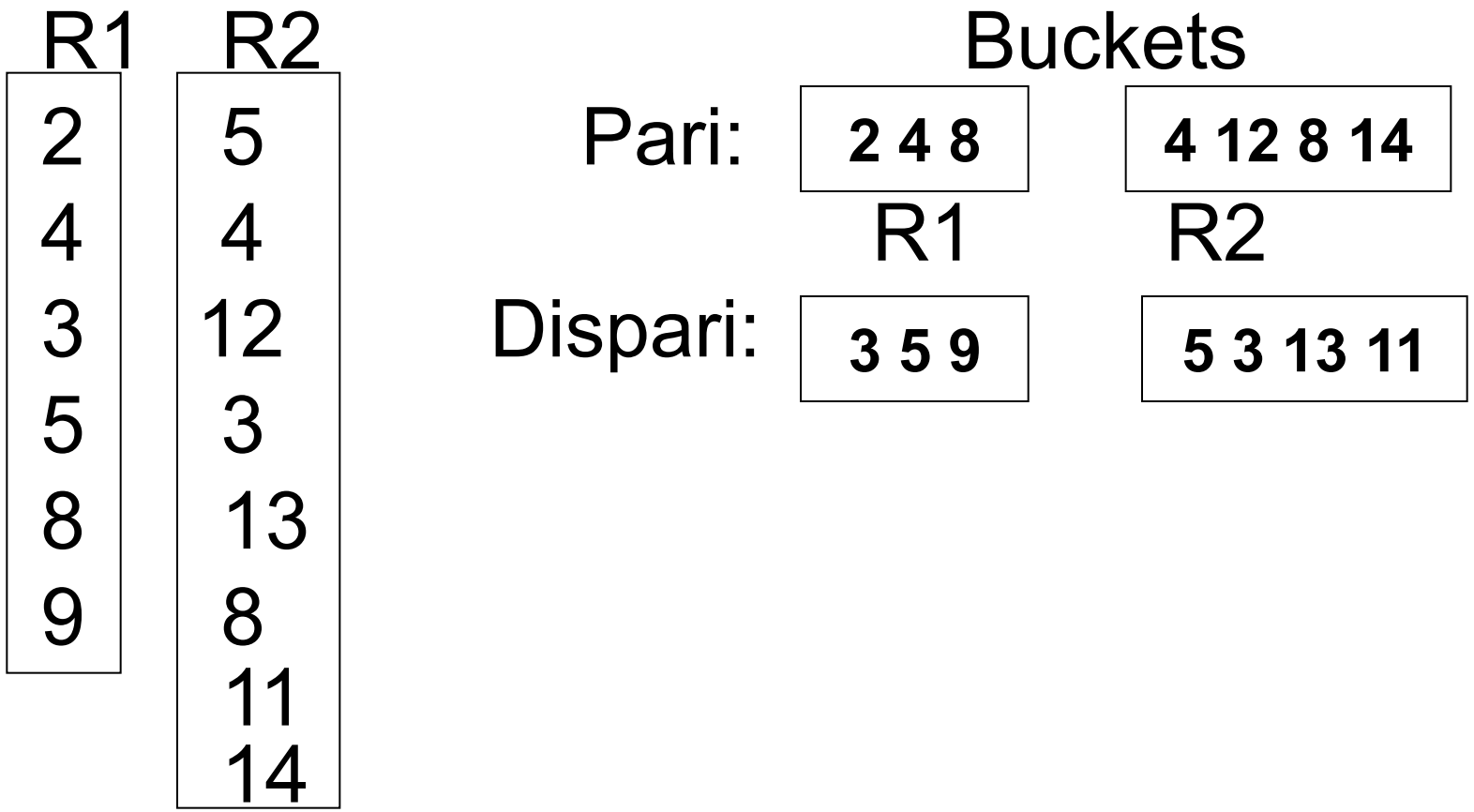
- Hash function h, range 0 $\rightarrow$ k
  - Buckets per R1: G*0*, G*1*, ... G*k*
  - Buckets per R2: H*0*, H*1*, ... H*k*

- Algoritmo per R1 JOIN R2 ON R1.C = R2.C:
  - (1) Fare Hash delle tuple di R1 in G buckets (in base al valore di C)
  - (2) Fare Hash delle tuple di R2 in H buckets (in base al valore di C)
  - (3) For *i* = 0 to k do:

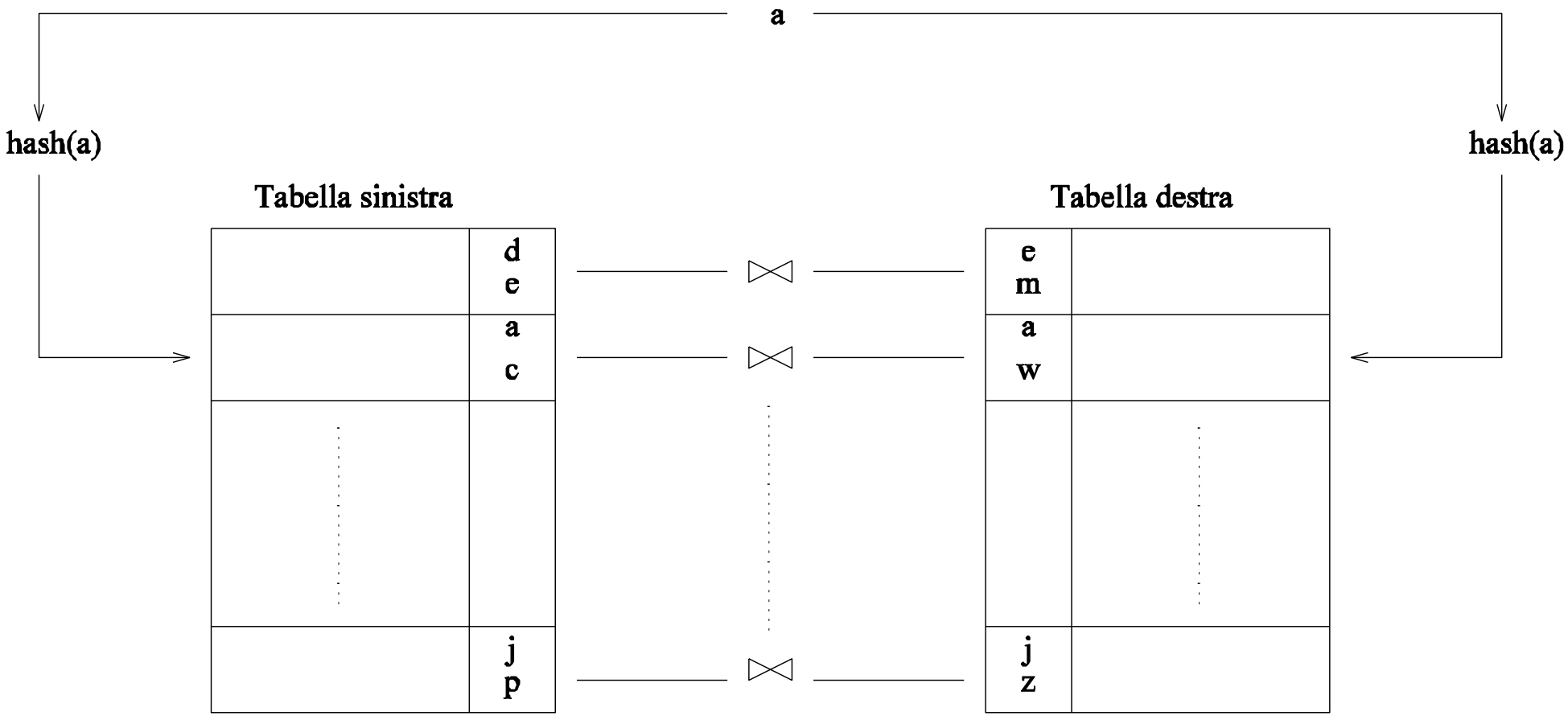
    confrontare le tuple nei bucket G*i* e H*i*

3) Hash-join

Simple example: hash even/odd



3) Hash-join

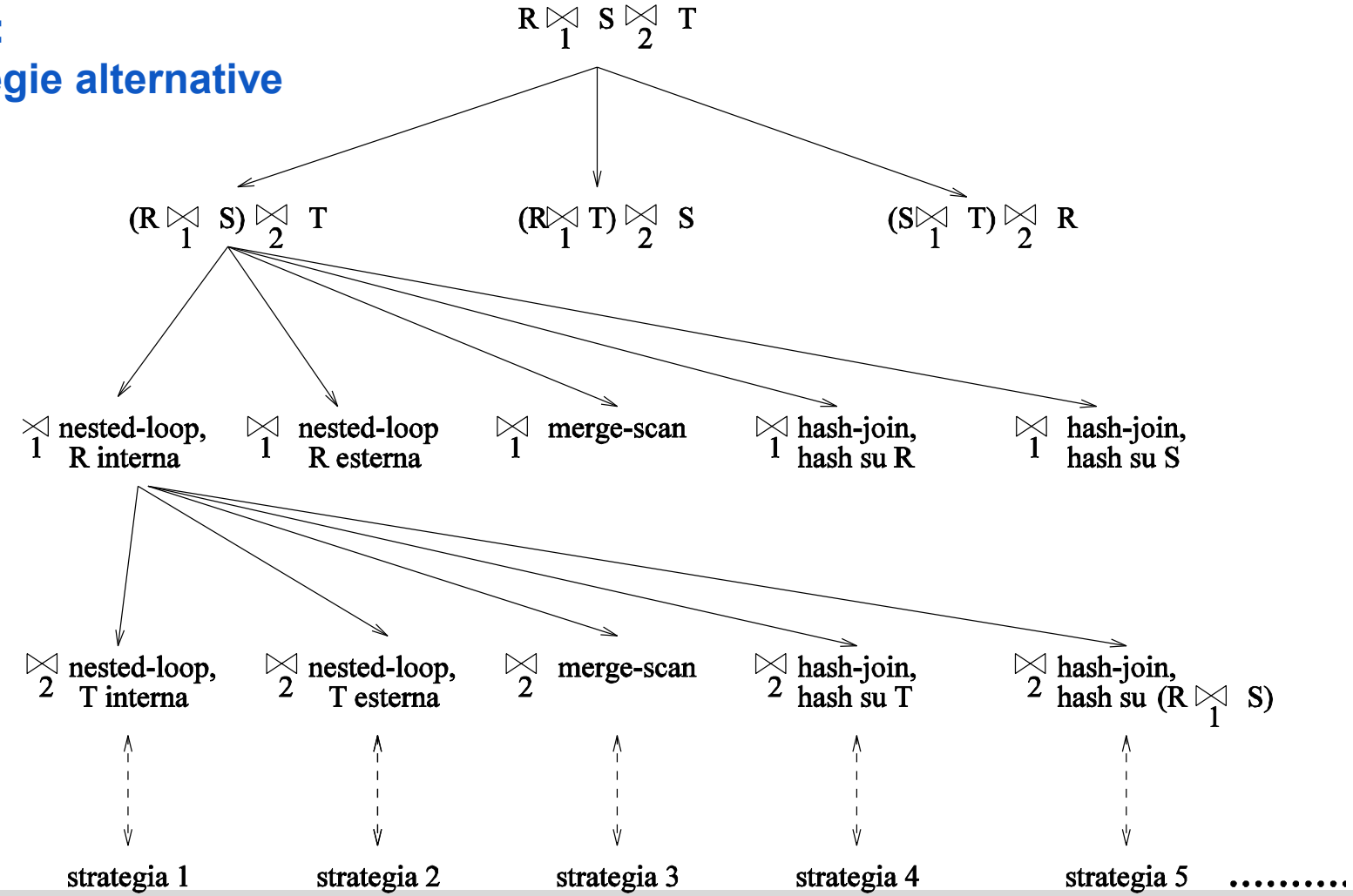


Il processo di ottimizzazione

- Si costruisce un albero di decisione con le varie alternative ("piani di esecuzione")
- Si valuta il costo di ciascun piano
- Si sceglie il piano di costo minore
- Il cosidetto ”ottimizzatore” trova di solito una "buona" soluzione, non necessariamente l’ottimo
- L’ottimizzatore non dev’essere troppo complesso o si perde il vantaggio dell’ottimizzazione!

Un albero di decisione

In totale:
75 strategie alternative



Esercizio 1(a) Nested loop R1 ⋈ R2

- Relazioni contigue; nessun indice

- Supporre:
  - #Records(R1) = 10.000
  - #Records(R2) = 5.000
  - Space(R1 record) =
  - Space(R2 record) = 1/10 block
  - RAM space = 101 blocks

R1 contiene 10.000 record; R2 ne contiene 5.000. Ogni record (di R1 o R2) occupa 1/10 dello spazio di un blocco. In RAM sono disponibili 101 blocchi.

Esercizio 1(a) Nested loop R1 ⋈ R2

- Relazioni contigue; nessun indice

#Tuples(R1) = 10.000
#Tuples(R2) = 5.000
Space(R1 tuple) = Space(R2 tuple) = 1/10 block
RAM space = 101 blocks

Qual è il costo del join?

For each R1 block:

- [Read R1 tuple + Read R2 tuples]

Totale = 1.000 [1+500] = 501.000 IOs

Possiamo fare di meglio?

Usiamo la memoria!

(1) Porzione di R1: Leggo 1.000 valori di R1,
usando 100 blocchi di RAM

(2) Leggo tutte le tuple di R2 (usando un blocco di RAM alla volta) + join con la porzione di R1

(3) Ripeto da (1) per tutte le porzioni di R1

Costo: per ogni porzione di R1:

Leggo la porz. di R1: 100 letture +

Leggo R2: 500 letture =

600 letture

Ripeto per tutte le 10 porzioni:

Totale = 10 x 600 = 6.000 letture

Possiamo fare meglio?

- Invertire l'ordine del join: R2 ⋈ R1

Totale = 5 x (100 + 1.000) =

= 5 x 1.100 = 5.500 letture

Esampio 1(b) Nested loop R2⋈R1

- Relazioni non contigue (nel caso peggiore, ogni record sta in un blocco diverso)

Costo

Per ogni porzione di R2:

Leggo la porzione: 1.000 letture +

Leggo R1: 10.000 letture =

11.000 letture

Totale = 5 porz. x 11.000 = 55.000 lett.

Esampio 1(c) Nested loop R2⋈R1

- Relazioni non contigue

Supponiamo di avere un indice hash su R1; l'indice occupa 99 blocchi di RAM – qual è il costo se ogni tupla di R2 ha esattamente 1 match con una tupla di R1 (caso a)? E se non c'è nessun match (caso b)?

Caso (a) costo: 5.000 x (1 + 1) = 10.000 letture

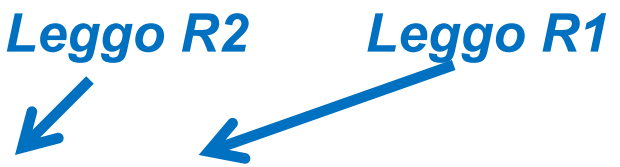
Caso (b) costo: 5.000 x (1 + 0) = 5.000 letture

Costo lettura indice hash: 99 letture

Esempio 1(d) Nested loop R2⋈R1

- Relazioni contigue

Supponiamo di avere un indice hash su R1; l'indice occupa 99 blocchi di RAM – qual è il costo se ogni tupla di R2 ha esattamente 1 match con una tupla di R1 (caso a)? E se non c'è nessun match (caso b)?



Caso (a) costo: 500 x (1 + 10) = 5.500 letture

Caso (b) costo: 500 x (1 + 0) = 500 letture

Costo lettura indice hash: 99 letture