Gestione dei dati

Parte 5
Gestione dell'accesso ai file e valutazione delle query

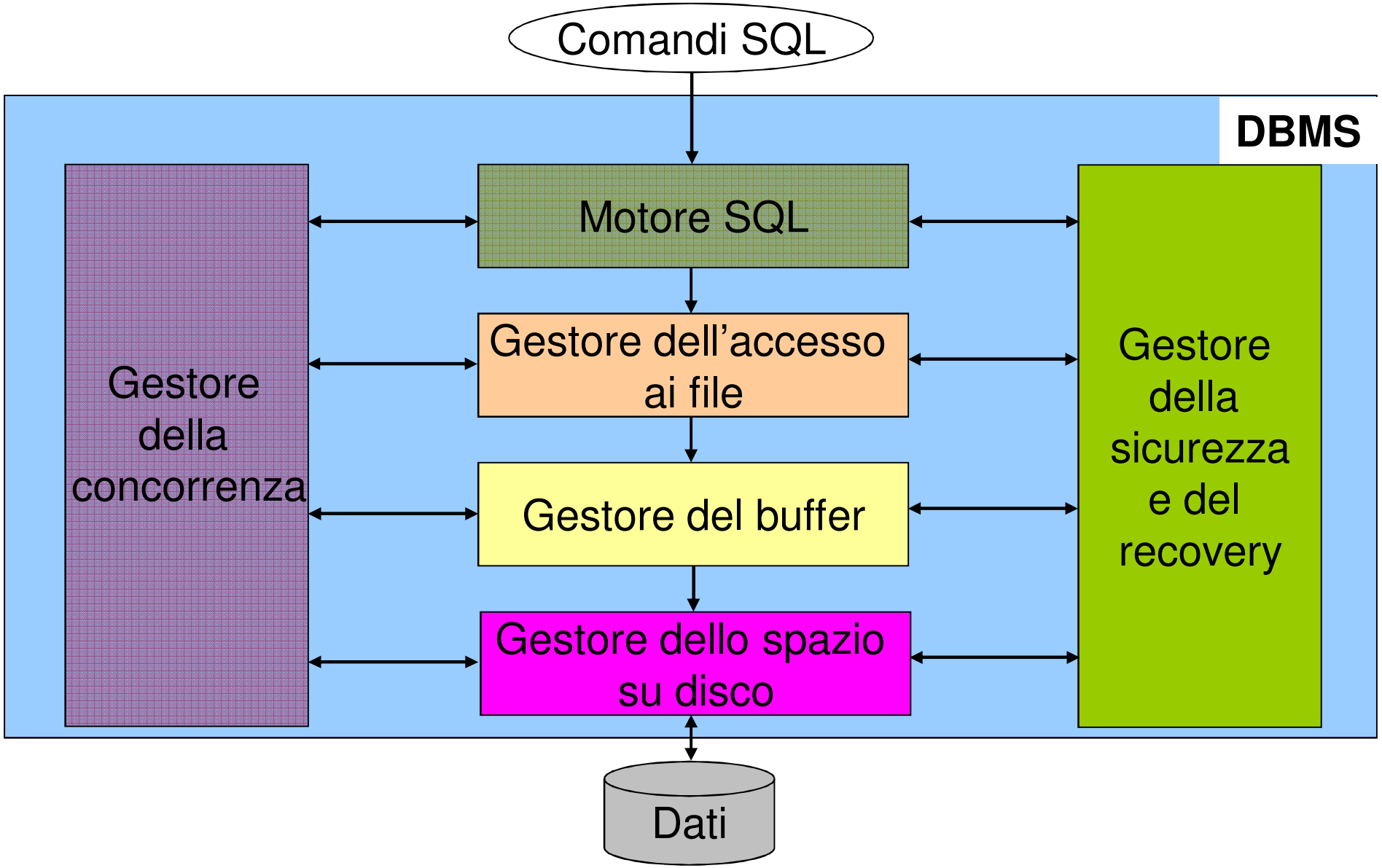
Maurizio Lenzerini, Riccardo Rosati

Facoltà di Ingegneria
Sapienza Università di Roma
Anno Accademico 2012/2013

http://www.dis.uniroma1.it/~rosati/gd/

SAPIENZA
UNIVERSITÀ DI ROMA

Architettura di un DBMS

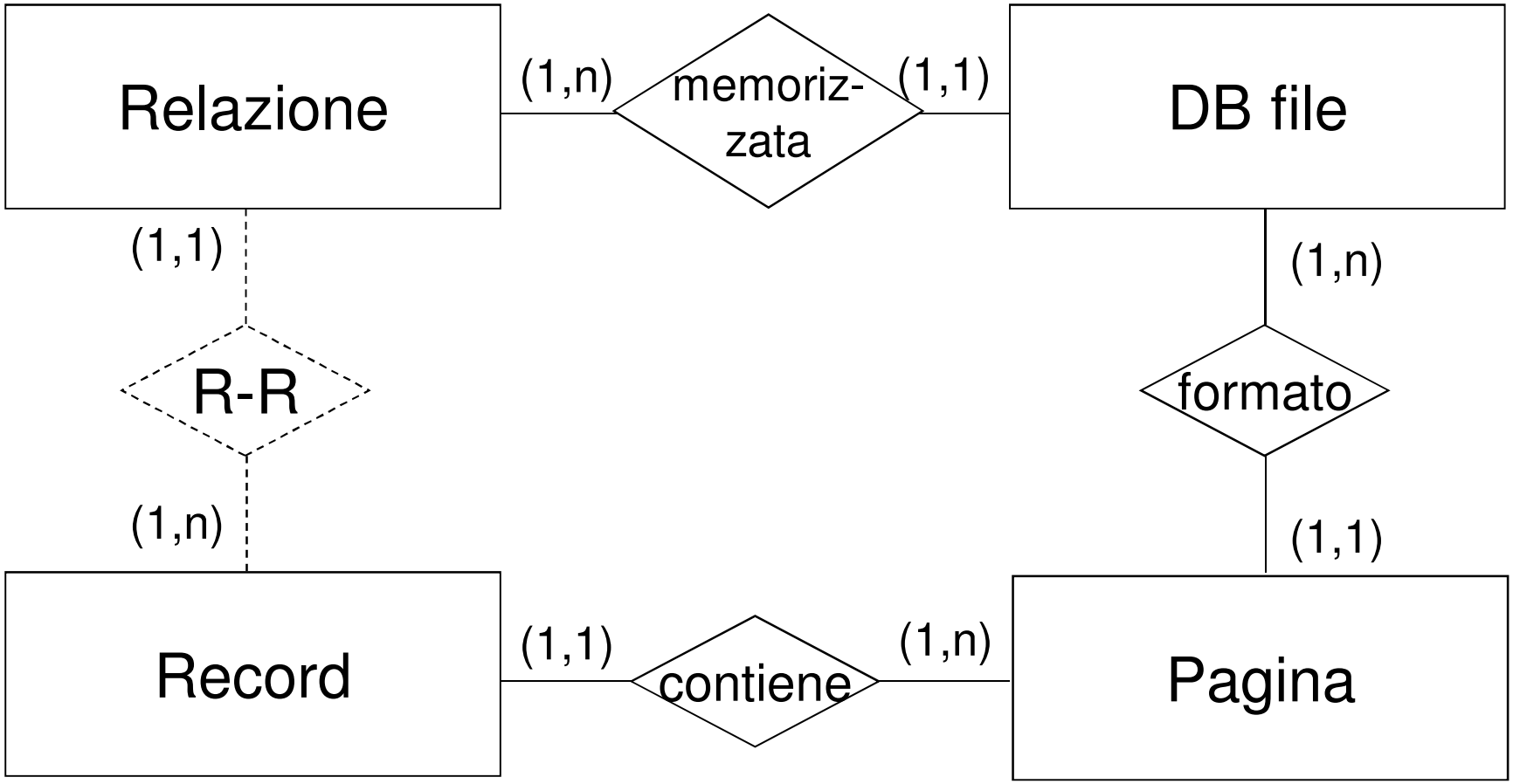


Sommario

1. Pagine e record
2. Le organizzazioni semplici di file
3. Le organizzazioni con indice
4. Valutazione di query

5.1
Pagine e record

Relazioni, file, pagine e record



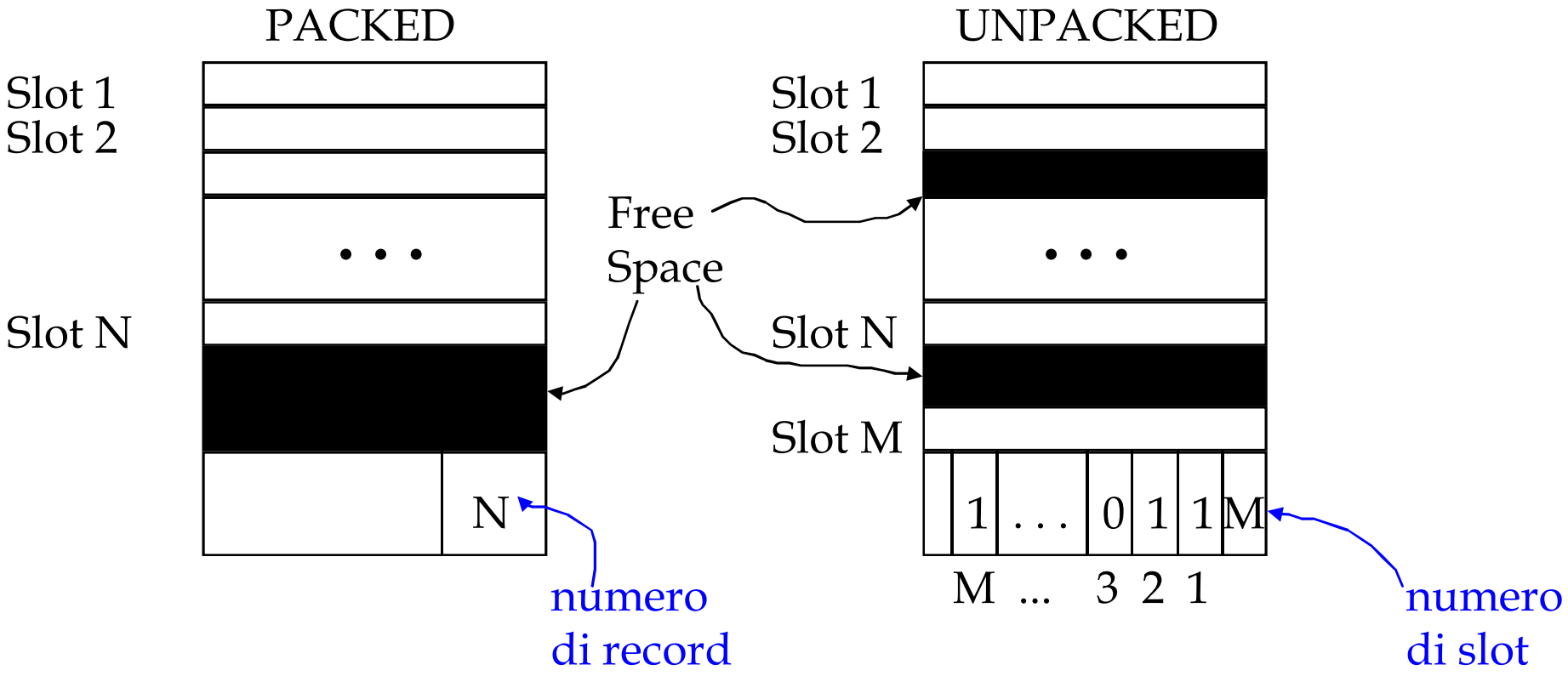
Nota: R-R è derivata

La pagina

- Usualmente ha la dimensione di un blocco
- Raccoglie un insieme di record (tipicamente di una relazione)
- Fisicamente, è una collezione di slot (ognuno per un record)
- Ogni record ha un identificatore (record id, o rid)

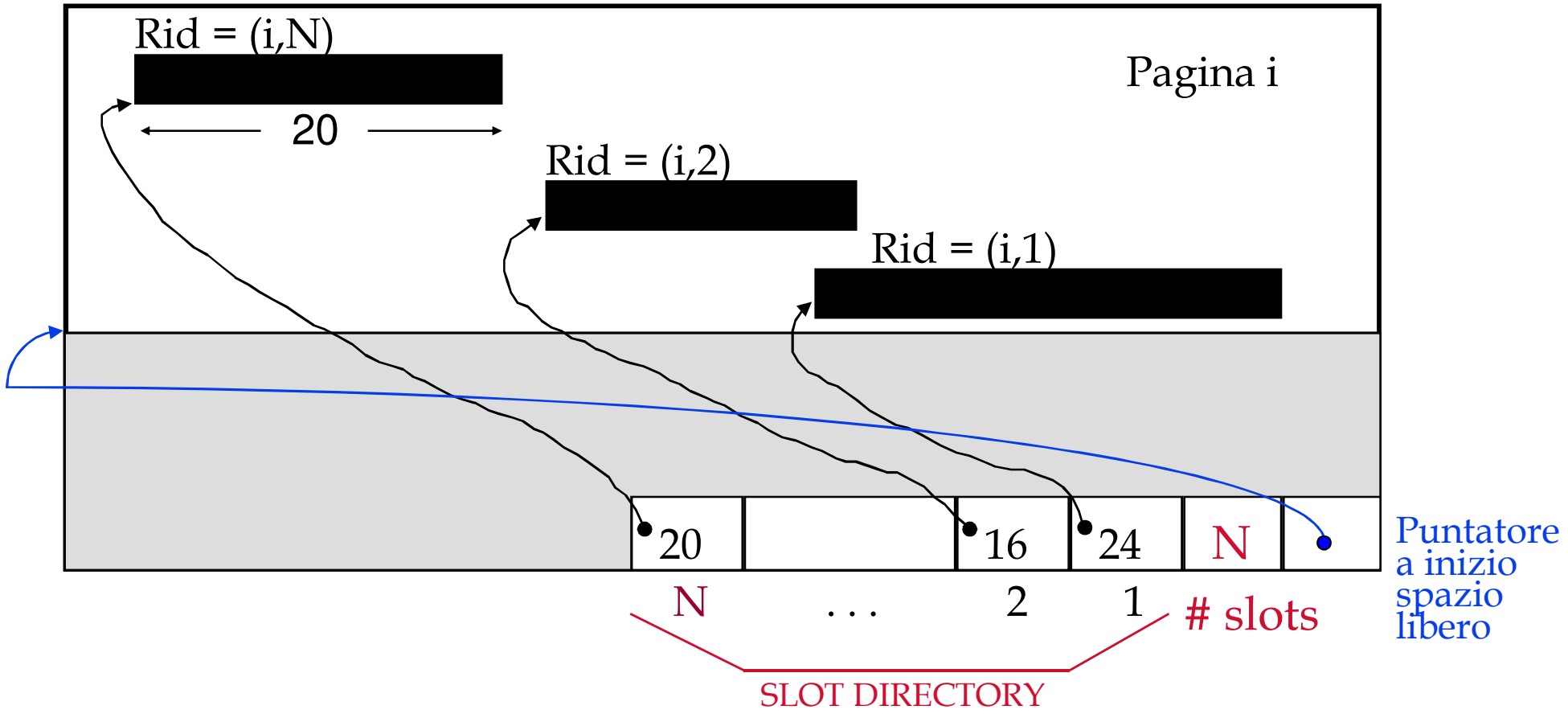
  rid = <page id, slot number>

Pagina con record a lunghezza fissa



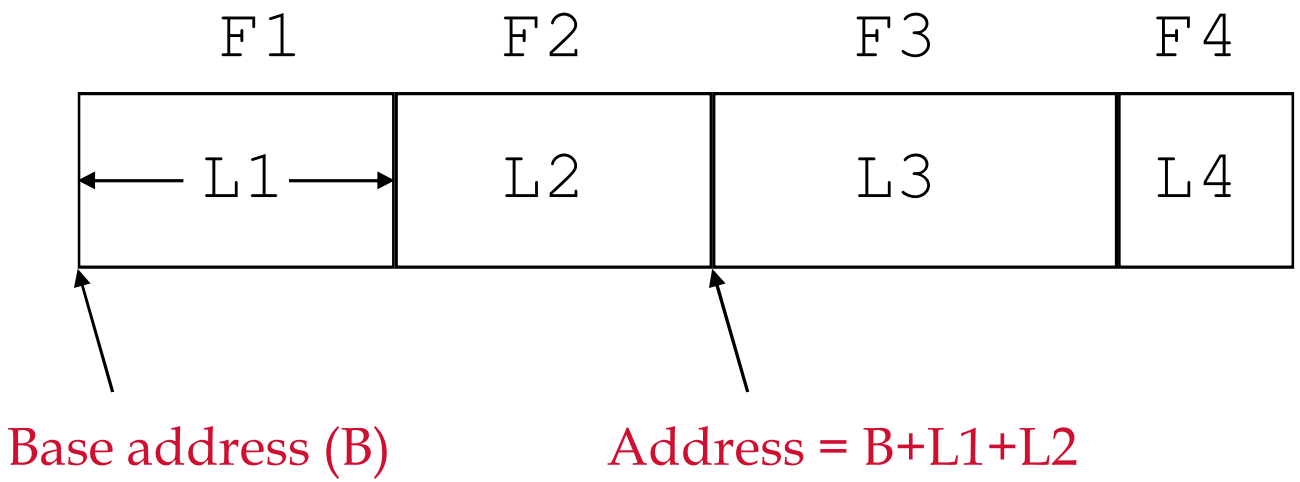
- Organizzazione packed: lo spostamento di record per la gestione dello spazio libero cambia il rid, e questo è un problema, se i record sono puntati da altri record in altre pagine
- Organizzazione unpacked: l'identificazione di un record richiede di analizzare l'array di bit, per vedere se lo slot è libero (0) o no (1)

Pagina con record a lunghezza variabile



Si possono spostare i record senza problemi, perché le loro informazioni sono nella directory. Cancellare un record significa mettere a -1 il valore del corrispondente slot nella directory, e spostare nello spazio libero la memoria occupata prima dal record.

Formato dei record a lunghezza fissa

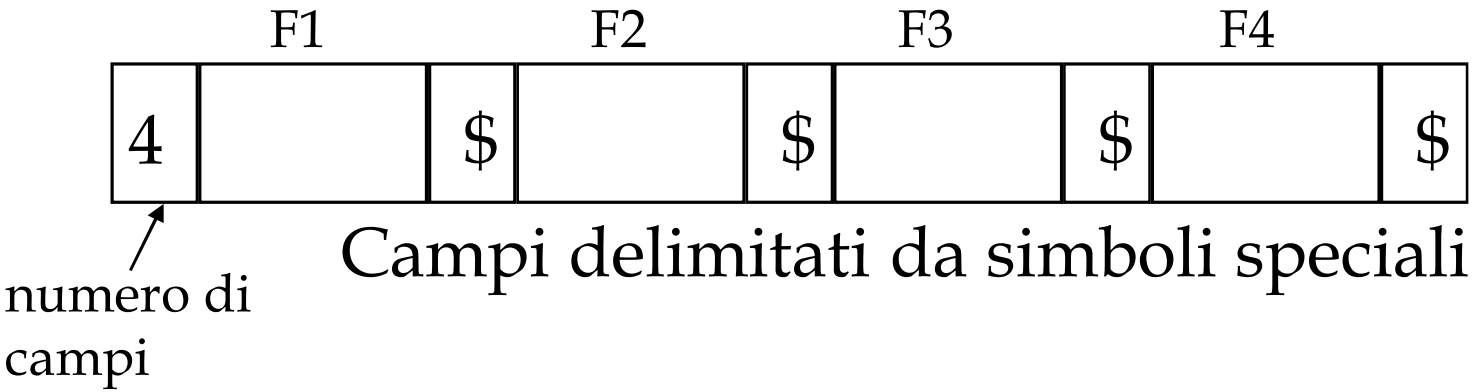


- Le informazioni sul tipo dei campi sono le stesse per tutti i record nel file, e sono memorizzati nel *system catalog*
- Trovare l'i-esimo campo richiede di scandire il record

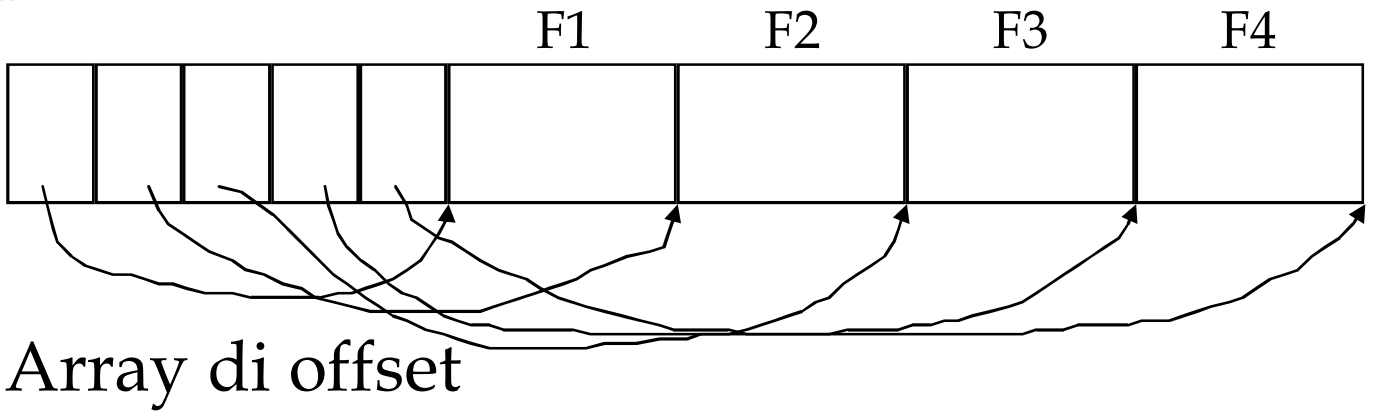
Formato dei record a lunghezza variabile

Due alternative (il numero dei campi è fisso):

(1)



(2)



La seconda alternativa consente l'accesso diretto ai campi ed efficienza nella memorizzazione dei null (puntatori contigui uguali)

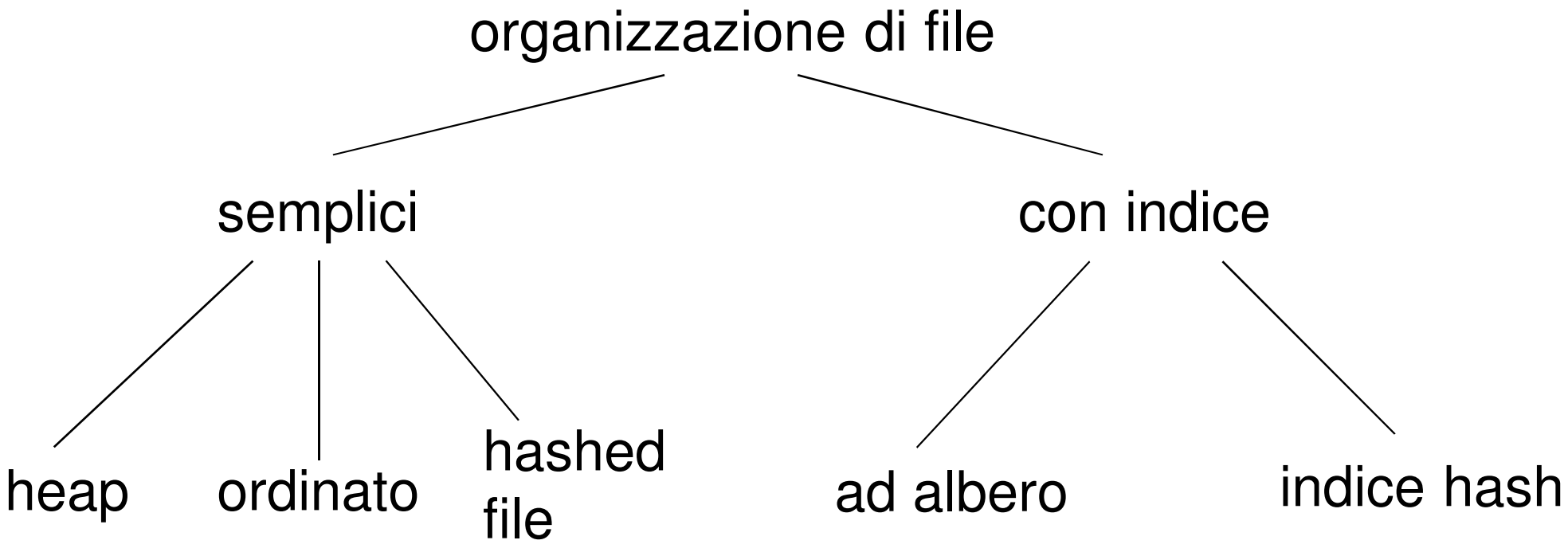
5.2
Le organizzazioni di file

File di record

- Un file è una collezione di pagine, ognuna contenente una collezione di record

- Su un file si devono poter eseguire operazioni di:
  - insert/delete/update record
  - leggere un record specificato dal suo rid
  - scandire tutti i record, eventualmente in base ad alcune condizioni che i record da trovare devono soddisfare

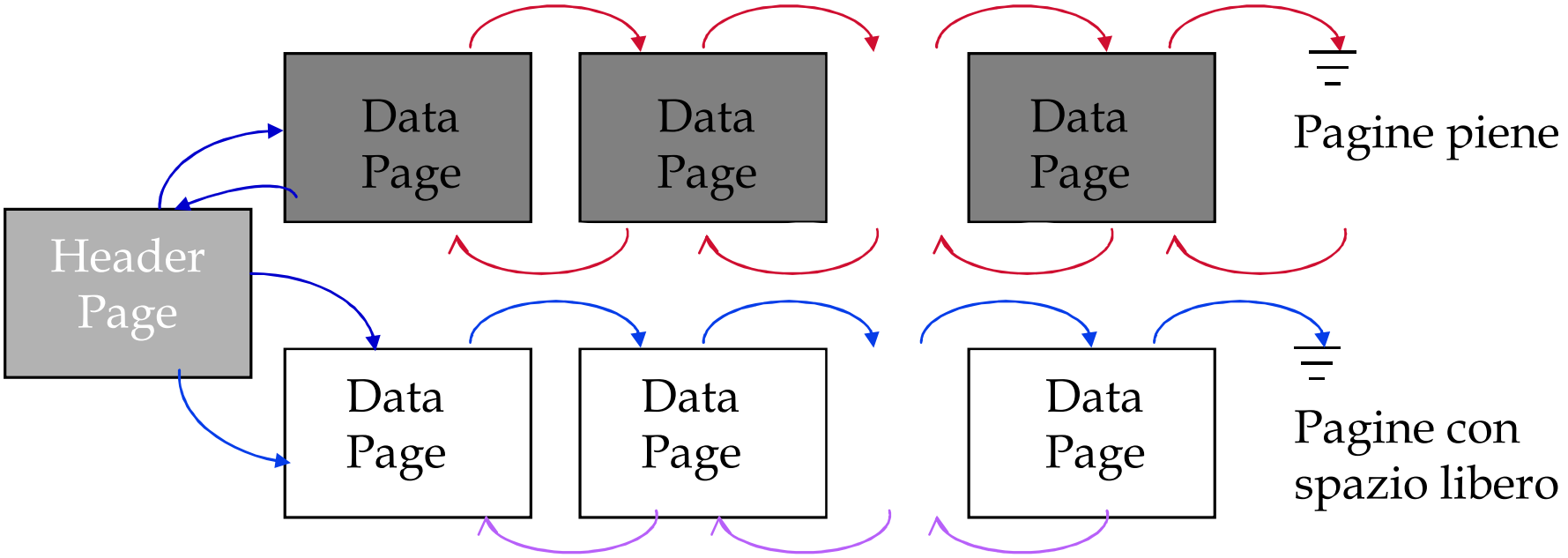
Organizzazioni di file



Heap File

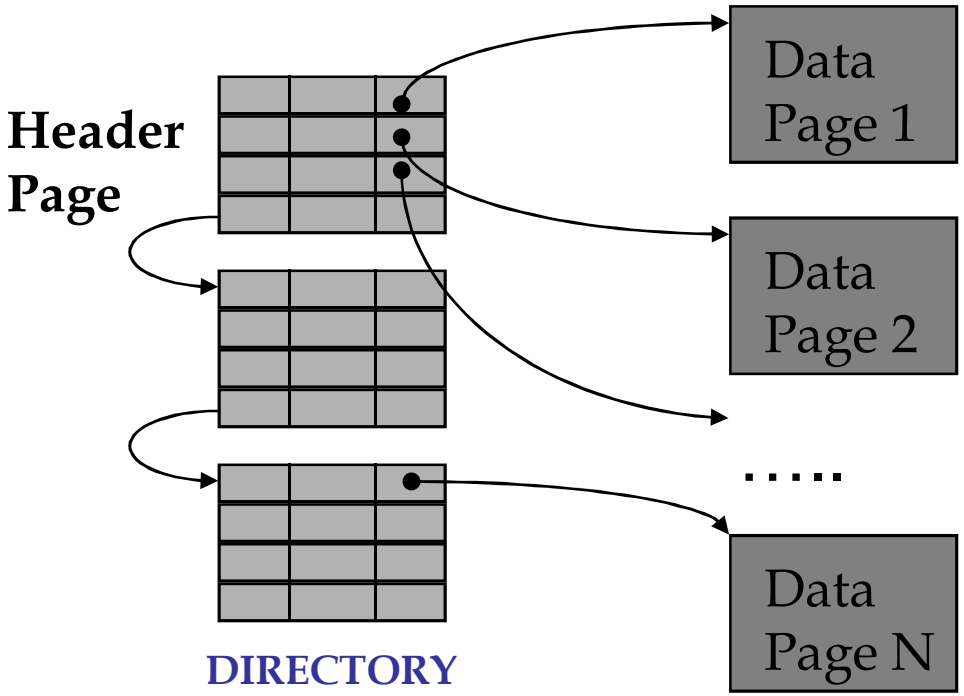
- Contiene record in pagine senza alcun ordine prestabilito
- Al crescere o al diminuire del file, le pagine vengono allocate e deallocate
- Per supportare le operazioni, dobbiamo:
  - tenere traccia della pagine
  - tenere traccia dello spazio libere nelle pagine
  - tenere traccia dei record nelle pagine

Heap File realizzato tramite lista



- Quando serve una pagina, si chiede al disk manager, e la si mette in testa alla lista di pagine con spazio libero
- Quando una pagina non serve più (ad esempio è vuota), si elimina dalla lista

Heap File realizzato con directory di pagine



- La directory è realizzata come una lista
- Ogni entry della directory si riferisce ad una pagina e dice se la pagina ha spazio libero (ad esempio con un contatore)
- Si cerca una pagina con spazio libero più efficientemente rispetto al caso precedente, perchè si esegue un numero di accessi alle pagine proporzionale alla dimensione della directory e non del file

File con record ordinati

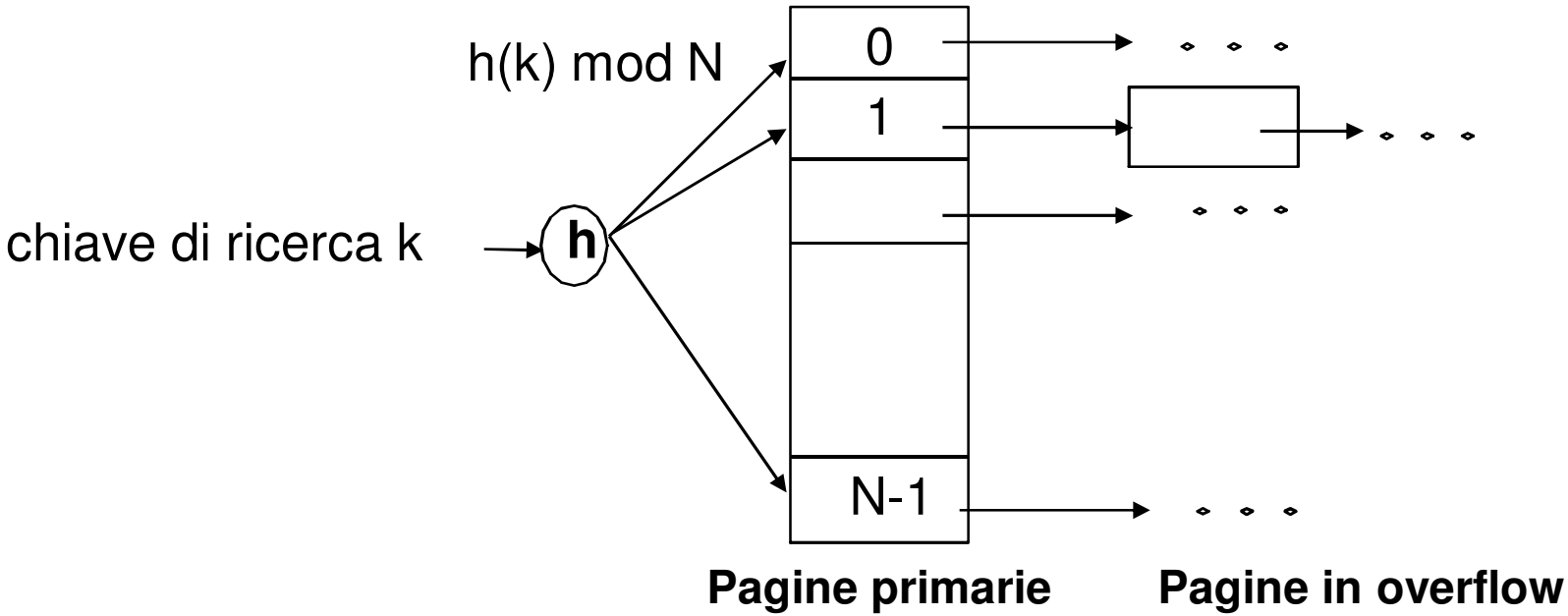
- L'ordinamento è su un insieme di campi, che formano la chiave di ricerca (search key)
- Possiamo pensare che le pagine siano organizzate come in un heap file, ma con pagine accessibili mediante una struttura sequenziale

Hashed file

- Le pagine del file sono organizzate in gruppi, chiamati *bucket*
- Un bucket consiste di:
  - una pagina, chiamata *pagina primaria*
  - eventualmente, ulteriori pagine (dette di *pagine di overflow*) collegate alla pagina primaria.
- Il bucket a cui un record appartiene può essere determinato applicando una funzione detta di *hash* alla chiave di ricerca

Hashed file

- Numero fissato N di pagine primarie (N = numero di bucket)
  - allocate sequenzialmente
  - mai de-allocated
  - nel bucket, pagine in overflow se necessario
- **h**(*k*) mod N = indirizzo del bucket a cui appartiene il record con chiave *k*
- **h** deve distribuire i valori in modo uniforme sull'intervallo 0..N-1



Modello di costo (tempo di esecuzione)

B: numero di pagine del file

R: numero di record per pagina

D: tempo di scrittura/lettura di una pagina su disco

- Tipicamente (oggi): 10 ms

C: tempo medio di elaborazione di un record (es. confronto di un campo con costante)

- Tipicamente (oggi): <100 ns

→ Le operazioni di I/O sono quindi dominanti!

Modello di costo (tempo di esecuzione)

- Modello che si concentra sui costi delle operazioni di ingresso-uscita (I/O)
  - C considerata costante
  - Per essere più accurati, occorrerebbe considerare anche:
    - Tempi di CPU
    - Costi delle trasmissioni via rete nelle basi di dati distribuite
- Costo delle operazioni di I/O basato sul numero di pagine lette/scritte su disco
  - Per essere più accurati, occorrerebbe considerare il fatto che tipicamente è possibile accedere ad un blocco di pagine contigue mediante una singola operazione di I/O

Le operazioni sui dati

1. Scansione di tutti i record di un file
   - Costo del caricamento delle pagine del file nel buffer
   - Costo CPU per localizzare ogni data record nella pagina
2. Selezione in base a predicati di uguaglianza
   - Caricamento delle pagine con data record di interesse
   - Localizzazione dei record di interesse nelle pagine caricate
   - Il record è unico se si lavora su una chiave della relazione, altrimenti non è unico
3. Selezione in base a predicati che vertono su di un intervallo di valori
   - Caricamento delle pagine con data record di interesse
   - Localizzazione dei record di interesse nelle pagine caricate

Le operazioni sui dati

4. Inserimento di un record
   - Costo dell'identificazione della pagina nel file
   - Caricamento della pagina
   - Modifica della pagina
   - Scrittura su disco della pagina modificata
   - Eventualmente, caricamento, modifica e scrittura di altre pagine
5. Cancellazione di un record
   - idem

Costo delle operazioni per heap file

1. Scansione:

   $$B(D + RC)$$

   – per ogni pagina (B) non vuota del file
     - caricamento (D)
     - per ogni record (R): elaborazione (C)

2. Selezione in base a criterio di uguaglianza:

   $$B(D + RC)$$

   → il data record può essere assente o può essercene più di uno

3. Selezione in base ad un intervallo di valori:

   $$B(D + RC)$$

Costo delle operazioni per heap file

4. Inserimento: D + C + D
   - caricamento della pagina (tipicamente l'ultima)
   - inserimento nella pagina
   - scrittura della pagina
5. Cancellazione
   - se il record è identificato dal rid: D + C + D (pagina identificata direttamente usando il rid)
   - se il record è specificato in base a criterio di uguaglianza o intervallo di selezione: B(D + RC) + XC + YD
     - X numero di record da cancellare
     - Y numero di pagine con almeno un record da cancellare

N.B. Sia nell'inserimento sia nella cancellazione, abbiamo considerato trascurabile il tempo necessario a trovare la pagina giusta ed a compattare le pagine in seguito a nuovo spazio libero

Costo delle operazioni per file ordinato

1. Scansione: $B(D + RC)$
2. Selezione basata su criterio di uguaglianza sulla search key:

   $D \log_2 B + C \log_2 R$ (caso peggiore)

   – ricerca binaria della pagina con (primo) record di interesse (se pagine memorizzate sequenzialmente)
     - $\log_2 B$ passi per localizzare la pagina
     - ad ogni passo, operazione di I/O + 2 confronti (che rispetto all'operazione di I/O si possono trascurare)

   – ricerca binaria del (primo) record di interesse: $C \log_2 R$
3. Selezione in base ad un intervallo di valori sulla search key:

   – ragionamento analogo

   – eventuale costo di caricamento di ulteriori pagine (se i record che soddisfano il criterio riempiono più pagine)

Costo delle operazioni per file ordinato

4. Inserimento:

$$D \log_2 B + C \log_2 R + C + 2B(D + RC)$$

   - caso peggiore: in prima posizione, nella prima pagina
   - costo di ricerca della corretta posizione nel file (cf. selezione in base a criterio di uguaglianza sulla search key):

$$D \log_2 B + C \log_2 R$$

   - inserimento del record: C
   - caricamento e scrittura di tutte le altre pagine: 2B(D + RC)

5. Cancellazione:
   - ragionamento analogo
   - se la condizione di cancellazione è basata su criterio di uguaglianza non sulla search key o appartenenza ad un intervallo di valori, il costo dipende anche dal numero di record da cancellare

Costo delle operazioni per hashed file

N.B.: si conta un quarto di pagine di overflow in più (cioè, si assume l'80% di occupazione), perchè si tende a sovradimensionare di un quarto lo spazio allocato al file

1. Scansione:

   $$1.25\ B(D + RC)$$

2. Selezione in base a criterio di uguaglianza sulla search key:

   $$(D + RC) \times 1.25$$

   → sulla search key, si assume di ottenere un accesso diretto con la funzione hash

3. Selezione in base ad un intervallo di valori: $1.25\ B(D + RC)$
4. Inserimento: $2D + RC$
5. Cancellazione: Costo della ricerca + $D + RC$

Confronto

Tipo di file	*Scansione*	*Ricerca con criterio di uguaglianza*	*Ricerca con intervallo di valori*	*Inserimento*	*Cancellazione*
Heap file	**BD**	**BD**	**BD**	**2D**	**Costo della ricerca + D**
File ordinato	**BD**	**D $\log_2$B**	**D $\log_2$B + num pagine interessate**	**Costo della ricerca + 2BD**	**Costo della ricerca + 2BD**
Hashed file	**1.25 BD**	**1.25 D**	**1.25 BD**	**2D**	**Costo della ricerca + D**

Nella tabella abbiamo contato solo il costo delle operazioni di I/O

5.3
Le organizzazioni con indice

La nozione di indice

Un *indice* è una struttura di dati ausiliaria che ha lo scopo di ottimizzare l'operazione di ricerca di un record in un file, sulla base del valore di uno o più campi, che costituiscono la chiave di ricerca (search key).

- Un qualsiasi sottoinsieme dei campi di una relazione può essere la chiave di ricerca per l'indice
- Una chiave di ricerca non si deve confondere con la chiave della relazione (insieme minimale di campi che identifica univocamente un record in una relazione)

Data entry, index entry and data record

Una organizzazione di file con indice per una relazione R prevede:

- File di indice (index file), che contiene
  - Data entry, ciascuno dei quali contiene un valore della search key k, e serve a localizzare i data record nel file di dati, in particolare quelli che hanno un legame con il valore k della search key
  - Index entry, che eventualmente servono alla gestione dell'indice
- File di dati (data file), che contiene i data record, cioè i record della relazione R

Proprietà degli indici

1. Come è organizzato il file di indice
   - basato su albero
   - basato su funzione hash
2. Come è fatto un data entry
3. Clustered/unclustered
4. Denso/sparso
5. Primario/secondario
6. Chiave semplice/chiave composta

Possibili realizzazioni dei data entry

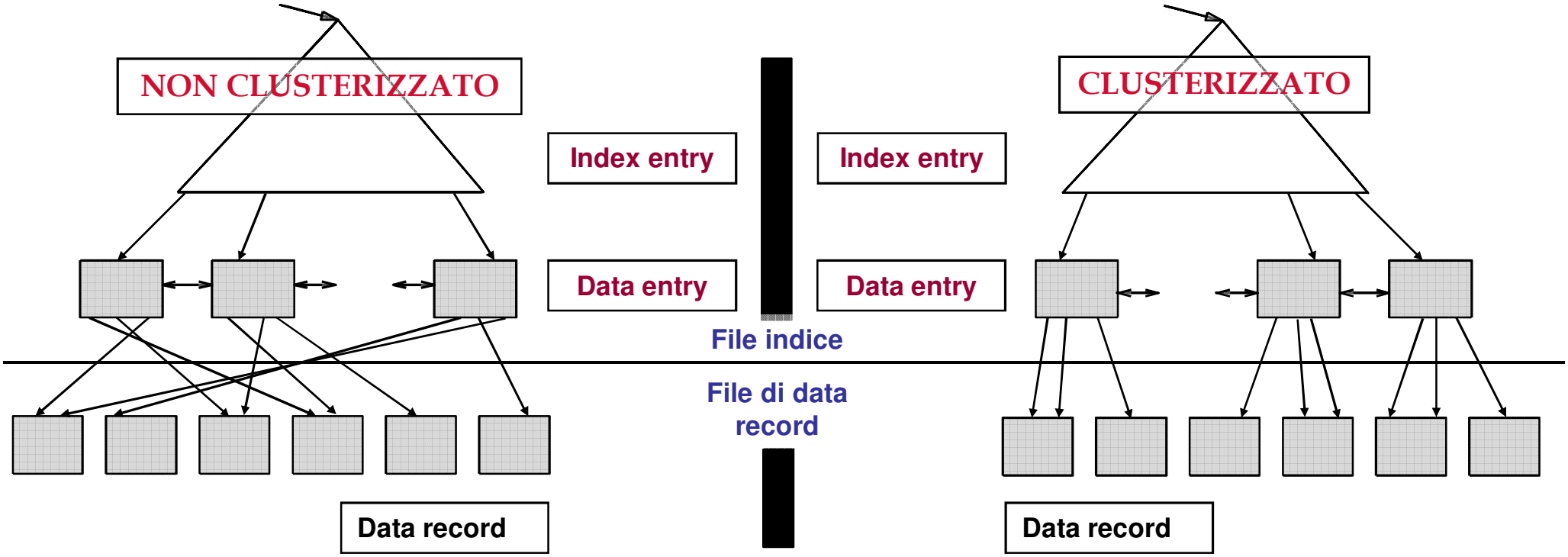
Tre sono le principali alternative per memorizzare un data entry con valore k per la search key (un tale data entry si indica con **k***):

1. **k*** è un data record (con chiave di ricerca pari a **k**)
   → è un caso degenere, perché non prevede in realtà i data entry (l'oganizzazione semplice hashed file può essere considerato un esempio)
2. **k*** è una coppia **(k,rid)**, dove **rid** è l'identificatore di un data record con chiave di ricerca pari a **k**
   → l'index file è indipendente dal file di dati
3. **k*** è una coppia **(k, rid-lista)**, dove **rid-lista** è una lista di identificatori di data record con chiave di ricerca pari a **k**
   → l'index file è indipendente dal file di dati
   → migliore utilizzazione dello spazio ma i data entry hanno lunghezza variabile

Se vogliamo usare più indici sullo stesso data file, al massimo uno di questi userà il metodo 1.

Clustered/unclustered

Un indice è *clustered* quando i suoi data entry sono organizzati secondo un ordine analogo (o identico) all'ordine dei data record nel file di dati che permette di accedere. Altrimenti si dice *unclustered*.



Clustered/unclustered

- Un indice i cui data entry sono memorizzati come nell'alternativa 1 è clustered per definizione
- Per le altre alternative, un indice è clustered solo se i data record sono ordinati nel data file in base al campo chiave di ricerca
- Se l'indice non è clustered, scade la sua efficacia per la ricerca basata su un intervallo di valori
- Ci può essere un unico indice clustered per data file, perché l'ordinamento nel data file può essere coerente solo con la chiave di ricerca di un indice

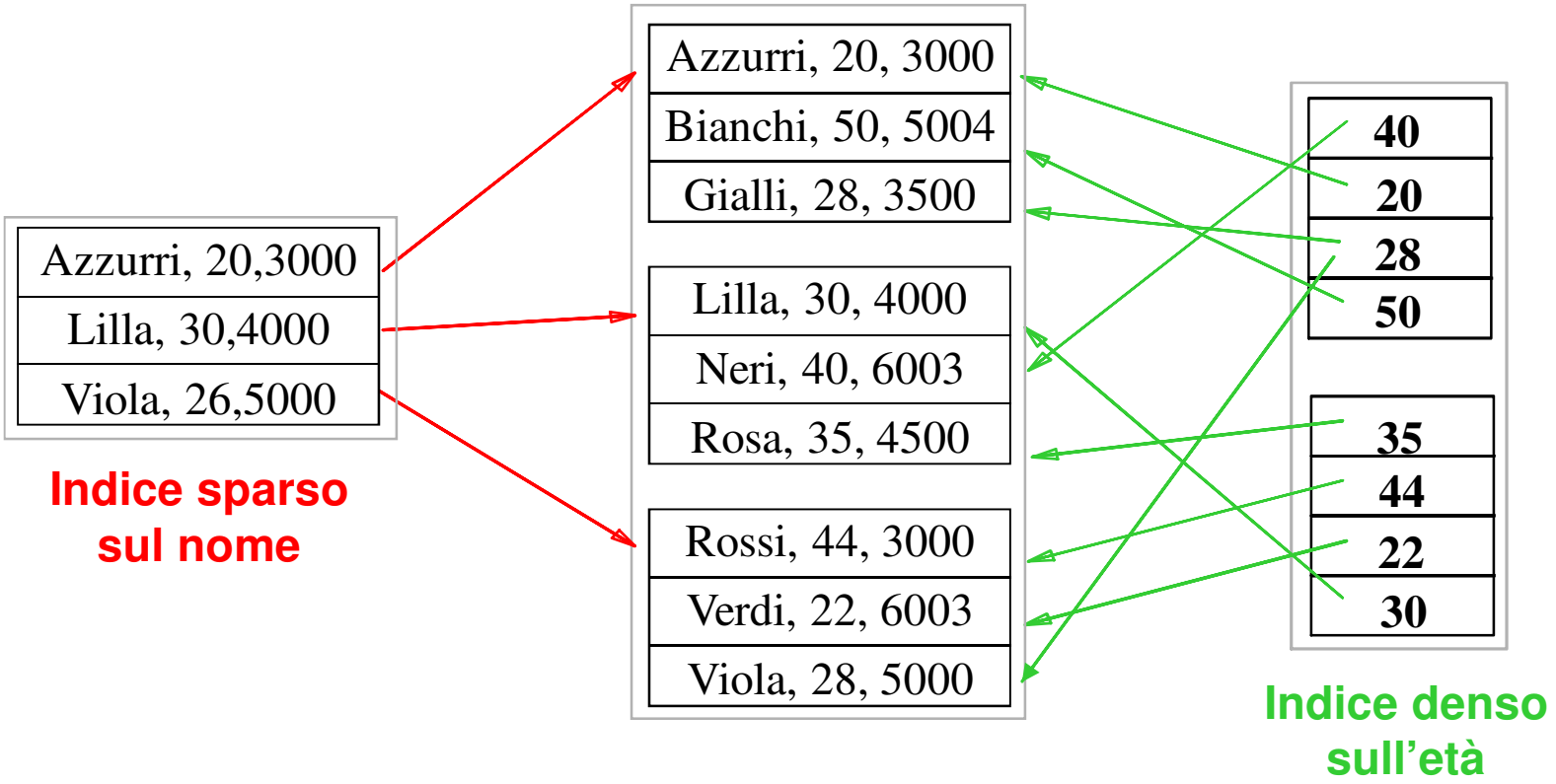
Indici primari e secondari

- Dicesi *indice primario* un indice su un insieme di campi che include la chiave primaria di una relazione. Altrimenti un indice è detto *indice secondario*

- Diconsi *duplicate* due data entry che hanno lo stesso valore per la chiave di ricerca.
  → Un indice primario non può contenere duplicati
  → Un indice secondario in genere contiene duplicati
  → Non esistono duplicati in un indice secondario, quando la chiave di ricerca per tale indice contiene una chiave (non primaria): si dice che l'indice secondario è *unique*

Indici sparsi e indici densi

Si dice che un indice è *denso* se ogni valore della chiave di ricerca presente sul data file ha almeno un corrispondente data entry. Altrimenti si dice che l'indice è *sparso*.

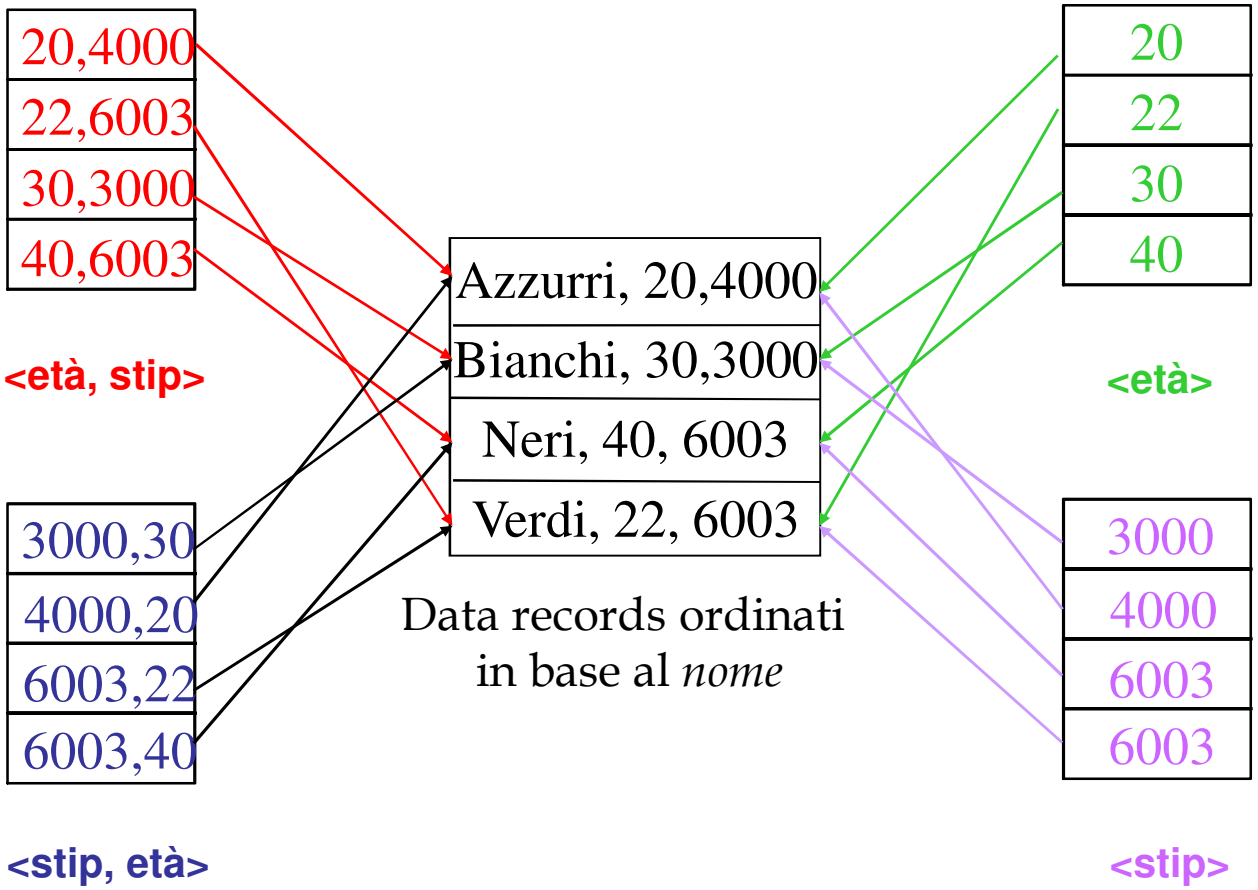


- Un indice organizzato secondo l'alternativa 1 è denso per definizione
- Un indice sparso è più piccolo di un indice denso
- Un indice sparso è per forza clusterizzato, e quindi si ha al massimo un indice sparso per data file

Chiave semplice e chiave composta

- Dipende da quanti campi formano la chiave di ricerca:
  - chiave di ricerca *semplice* se un solo campo, altrimenti *composta*
- Se chiave composta, allora una query in base a criterio di uguaglianza (equality query) è una query in cui il valore di ogni campo della chiave è fissato, mentre una query che fissa solo alcuni campi e lascia libero il valore per i campi non specificati è una query in cui la ricerca è basata su un intervallo di valori (range query)
- Un indice composto supporta un maggior numero di query ed, inoltre, dato che l'indice contiene più campi, attraverso l'accesso al solo indice si riesce ad estrarre più informazione - ad esempio si può in certi casi addirittura evitare l'accesso al file dati (*index-only evaluation*).
- Un indice composto necessita più di frequente di essere aggiornato (a seguito di ogni operazione che modifica uno qualsiasi dei suoi campi).

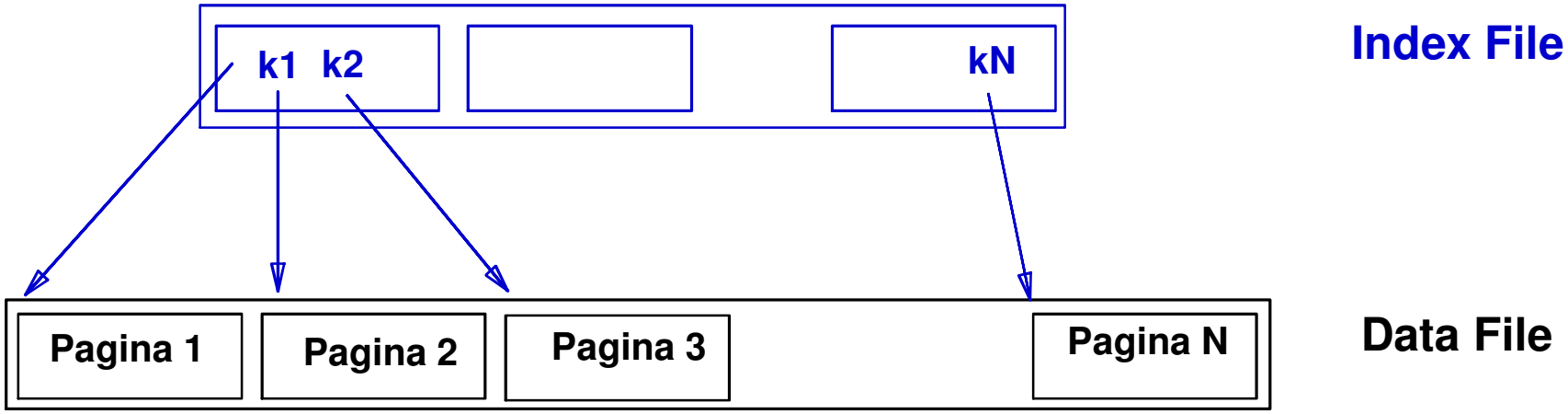
Chiave semplice e chiave composta



Organizzazioni con indice

Idea di base degli indici ad albero

- Trova gli studenti con media *> 27*
  - Se i dati sono ordinati per media, ci si può affidare alla ricerca binaria
  - Ma il costo della ricerca binaria può diventare alto all'aumentare della dimensione del file
- Semplice idea: crea un file indice sul file ordinato



Adesso si può eseguire la ricerca binaria su un file più piccolo (index entries piccole, un'entry per pagina del data file)

Indici ad albero: caratteristiche generali

- I data entry sono organizzati ad albero in diverse pagine in base al valore del campo chiave di ricerca

- Le ricerche sono dirette verso la pagina corretta dei data entry, mediante l'utilizzo di un albero, cioè una struttura dati gerarchica di ricerca
  - Ogni nodo coincide con una pagina
  - Le pagine con i data entry costituiscono le foglie dell'albero
  - Tutte le ricerche cominciano dalla radice e finiscono su una foglia: importante usare alberi con altezza minima
  - I legami fra nodi vengono stabiliti da puntatori che collegano fra loro le pagine

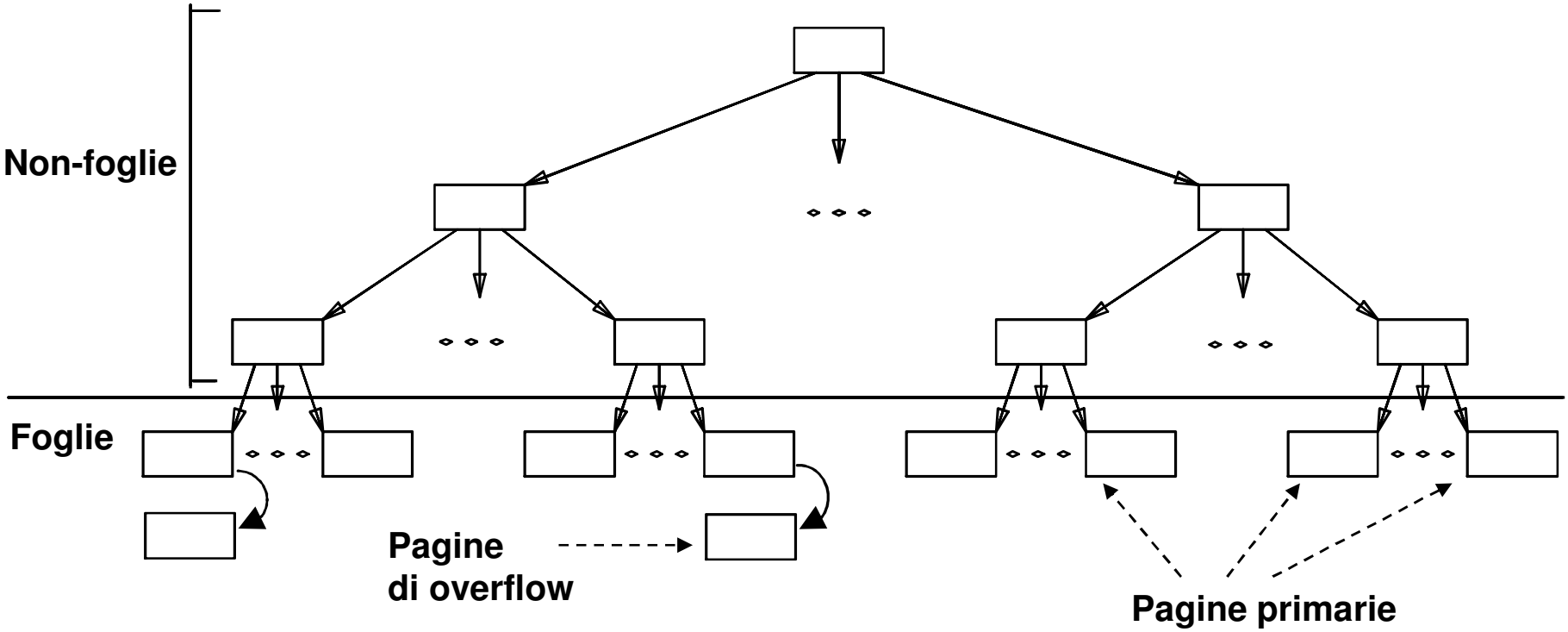
Indici ad albero: caratteristiche generali

La struttura tipica di ciascun nodo intermedio di un albero (inclusa la radice) è la seguente:

- Sequenza di m + 1 puntatori P separati da diversi valori K ordinati della chiave di ricerca
- Il puntatore P_{i-1} alla sinistra del valore K_i ($1 \leq i \leq m$) punta al sottoalbero che contiene solo data entry con valori di chiave strettamente inferiori a K_i
- Il puntatore P_i alla destra del valore K_i punta al sottoalbero che contiene solo data entry con valori di chiave superiori o uguali a K_i

ISAM

Il nome deriva da Indexed Sequential Access Method



Le foglie contengono i data entries e possono anche essere scandite sequenzialmente

Commenti su ISAM

- *Creazione del file*: foglie allocate sequenzialmente
- *Ricerca*: Iniziamo dalla radice, e confrontiamo le chiavi fino arrivare ad una foglia. Il costo è

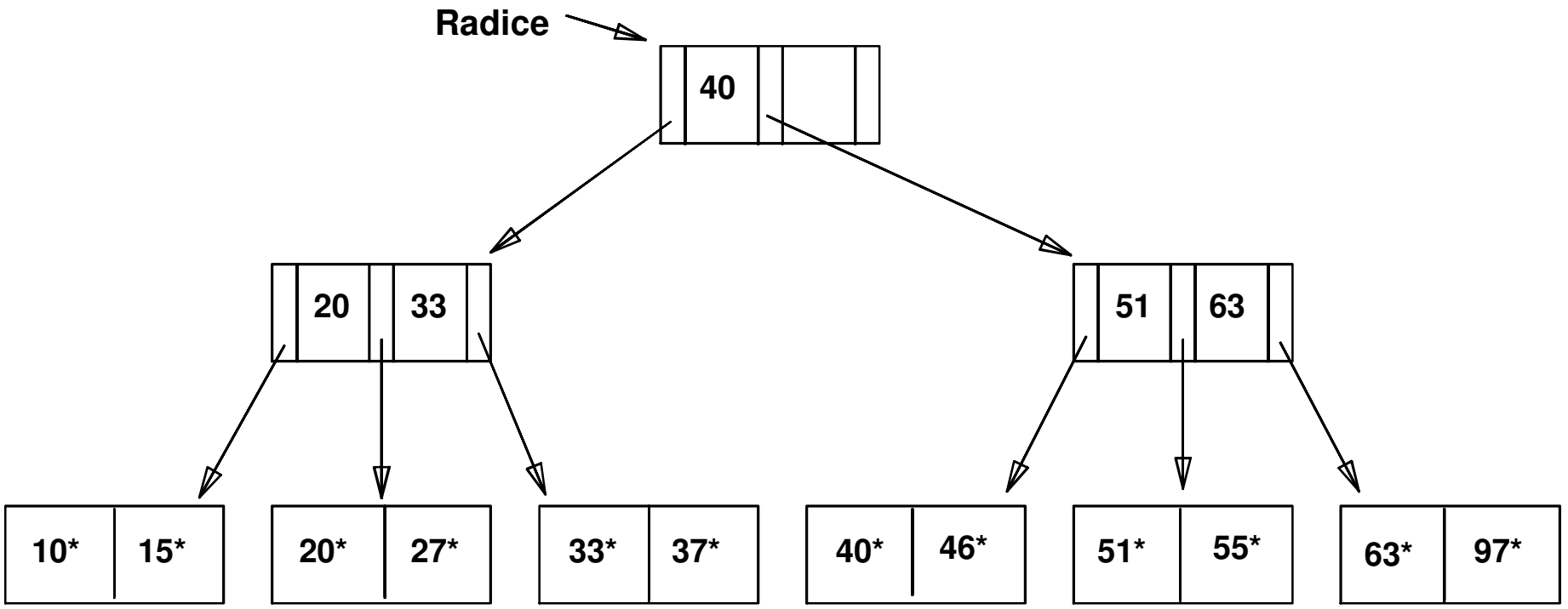
$$\log_F N$$

  dove F è il numero di index entries diviso il numero di pagine con data entries (ovvero il fan-out dell'albero, ovvero il numero dei figli di un nodo), ed N è il numero di foglie (che tipicamente è B/F)
- *Inserimento*: Trova la foglia ed inserisci lì, allocando una pagina di overflow, se necessaria (inserisci poi nel data file)
- *Cancellazione*: Trova e rimuovi dalle foglie; se la pagina è di overflow ed è vuota, allora deallocala (elimina poi dal data file)

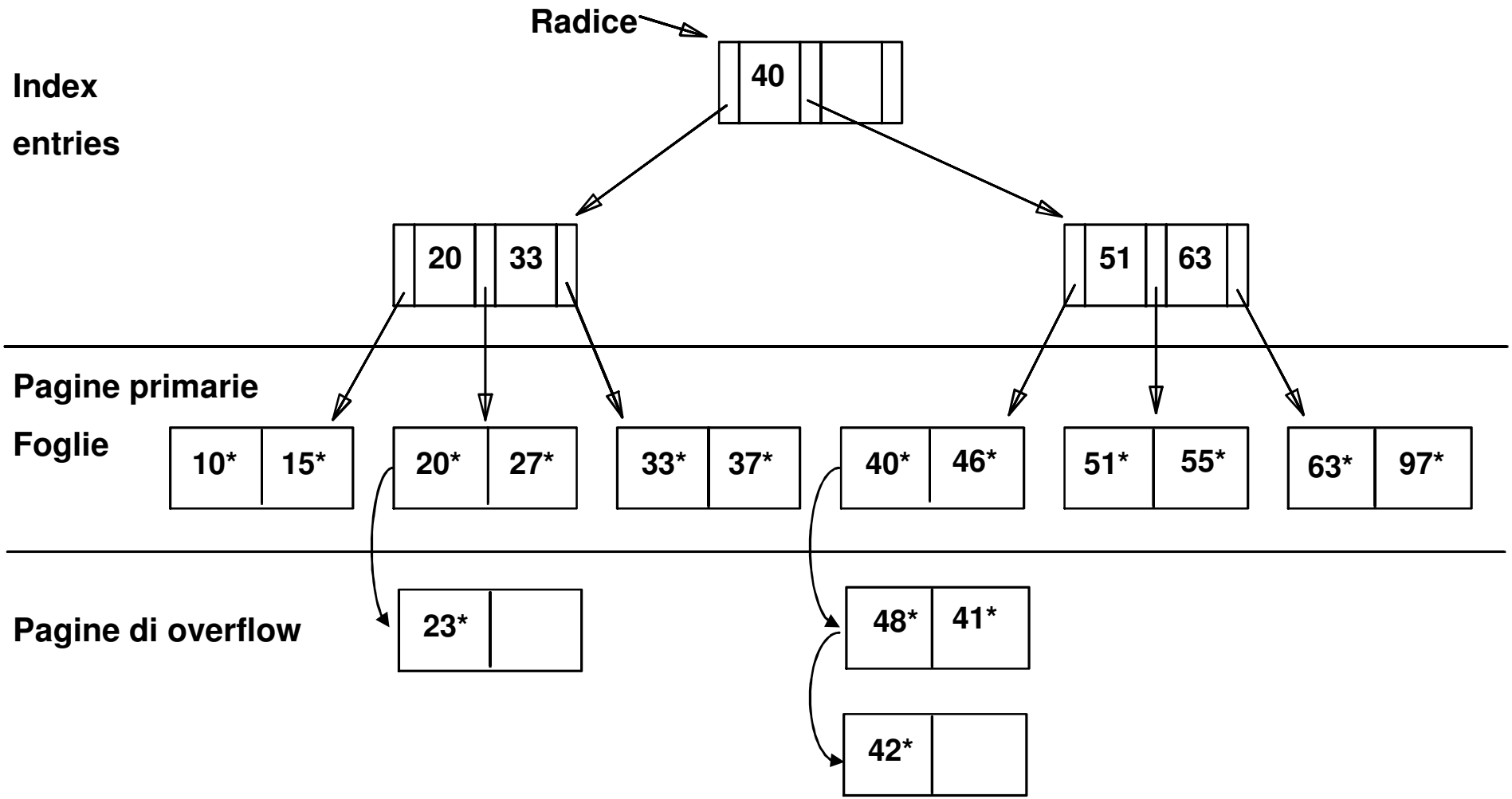
Struttura statica: *insert/delete coinvolgono solo le foglie*

Esempio di ISAM

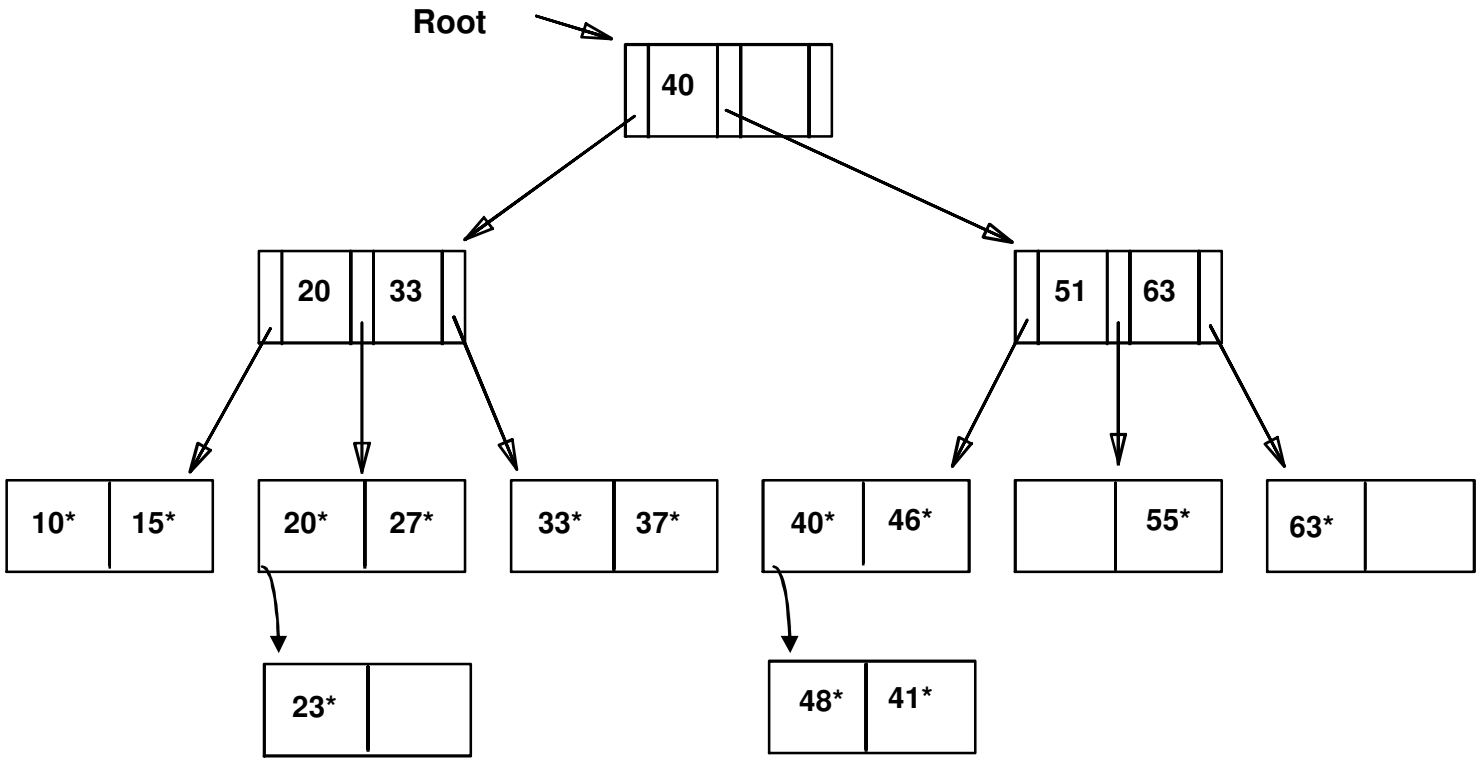
Assumiamo qui che ogni nodo contenga 2 entries:



Inserimento di 23*, 48*, 41*, 42* ...



Cancellazione di 42*, 51*, 97*



Si noti che 51* appare negli index entries, ma non nelle foglie

Indice B+-tree

- Alberi *bilanciati* (in altezza), in cui la lunghezza del cammino dalla radice ad una foglia è la stessa per tutte le foglie
- Superano i problemi di ISAM con le operazioni di insert/delete
- Chiamiamo *rank* dell'albero il numero **d** pari alla metà del numero massimo di tuple che possono essere contenute in un nodo
- Ogni nodo del B+-tree deve contenere un numero di tuple *m* tale che **d** <= *m* <= 2**d** (ogni nodo è pertanto occupato almeno al 50%)
- Foglie (data entry) collegate da lista in base all'ordine sulla chiave

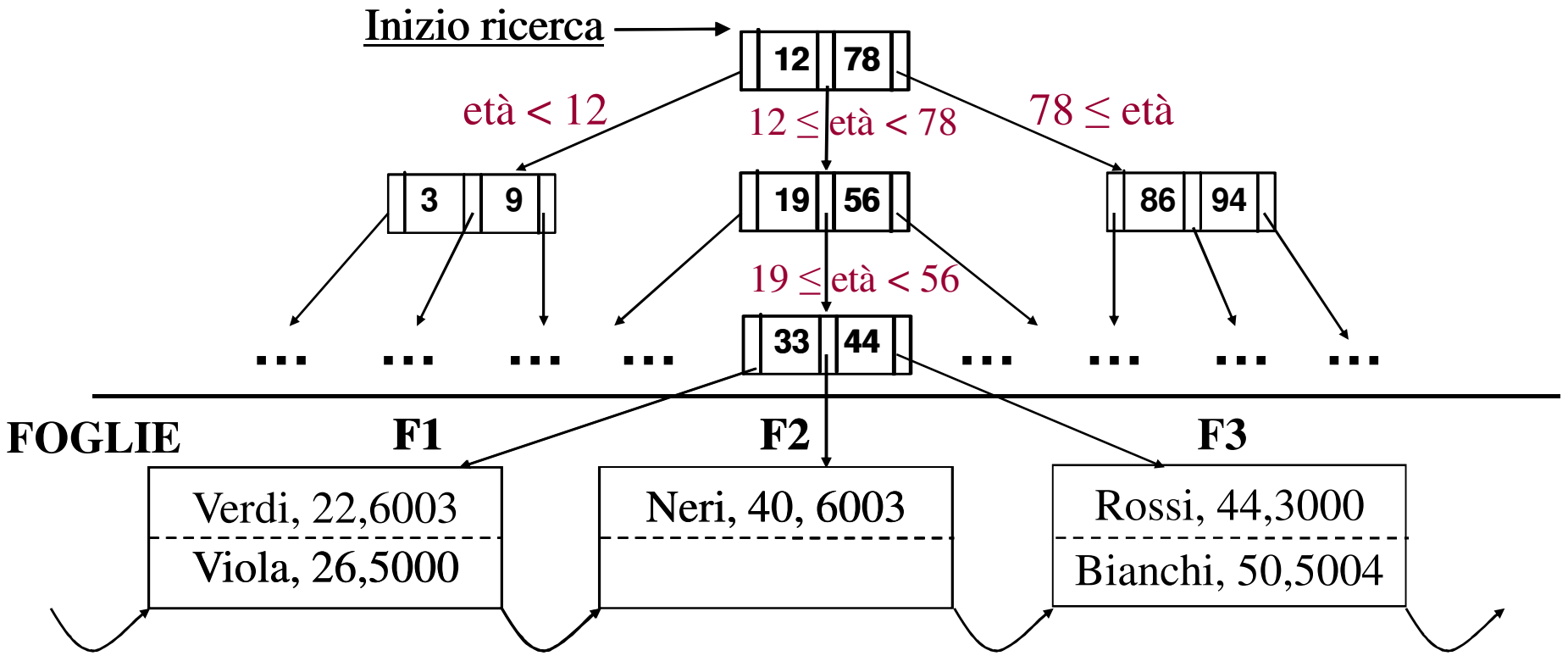
  → Tale collegamento è utile per le interrogazioni in cui il predicato definisce un intervallo di valori ammissibili:
  - si accede al primo valore dell'intervallo (con una normale ricerca)
  - si scandiscono uno dopo l'altro i nodi foglia dell'albero fino a trovare valori di chiave maggiori dell'estremo superiore dell'intervallo

Ricerca con indice B$^+$-tree: esempio

Si cercano i data entry con 24 < età ≤ 44

→Ricerca diretta verso foglia con primo data entry di interesse

→Tutte le foglie sono collegate da una lista: si recuperano così quei data entry in F2 e F3 che soddisfano il criterio di ricerca



Ricerca con indice B$^+$-tree: osservazioni

- Il numero di operazioni di ingresso-uscita che occorrono durante una ricerca è al più pari alla lunghezza del cammino dalla radice alle foglie (F è il fan-out dell'albero cioè il numero medio di figli per i vari nodi):

  $\log_F N$, con N numero totale di foglie

- Si fa in modo che ogni nodo abbia un numero di discendenti grande (che dipende dalla dimensione della pagina):
  - → tipicamente il fan-out è almeno pari a 100, e noi assumeremo, se non specificato diversamente, il fan-out proprio uguale a 100; con un fan-out del genere, con 1.000.000 di pagine, il costo della ricerca è 3
  - → alberi con numero limitato di livelli
  - → maggioranza delle pagine occupata da nodi foglia

Ricerca con indice B^+-tree: osservazioni

- Gli alberi di tipo B^+-tree costituiscono la tecnica ideale per accedere a dati compresi in un intervallo di valori
- Sono anche buoni per accedere a dati in base a predicati di uguaglianza
- Come detto, il numero medio di figli per i nodi intermedi è detto *fan-out* dell'albero
- Con buona approssimazione, dato un albero con altezza h e fan-out F, si ha che il numero di foglie è pari a F^h

Inserimento in un B+-tree

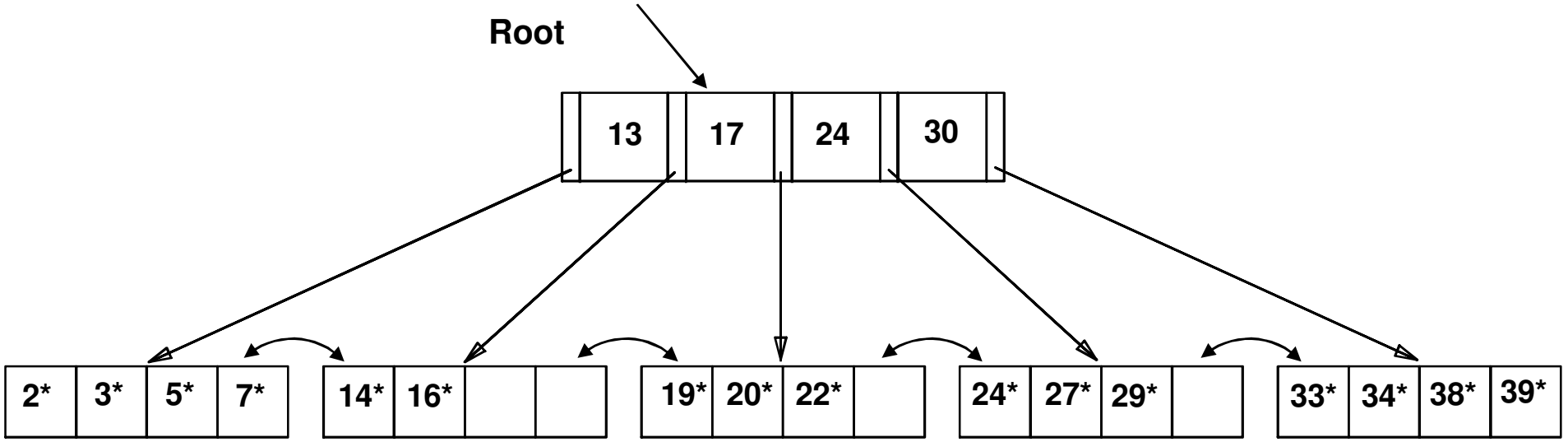
Per mantenere il bilanciamento, quando si vuole effettuare un inserimento in una foglia L piena:

- tuple di L sono suddivise equamente tra L ed un nuovo nodo L2
- si inserisce nel nodo padre un data entry che punta al nodo L2 e che ha valore di chiave pari al valore minimo tra quelli che compaiono in L2
- in maniera ricorsiva, se necessario si risale alla radice

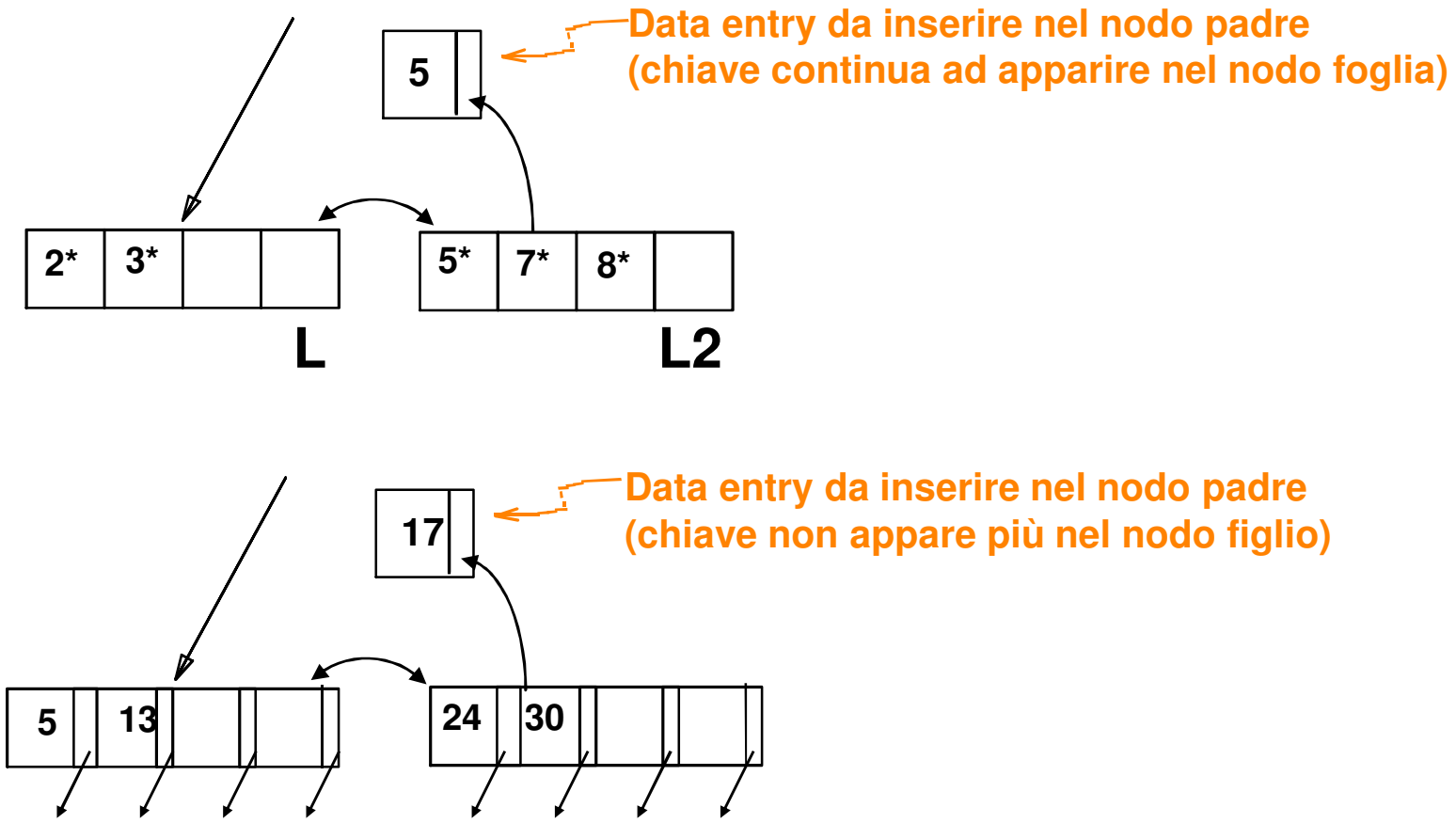
→nel suddividere i data entry di un nodo intermedio, si sposta la chiave media dal nodo al padre (non si copia il valore della chiave come nel caso di un nodo foglia)

Inserimento in un B+-tree: esempio

Inserimento di un data record con valore della chiave di ricerca pari a 8

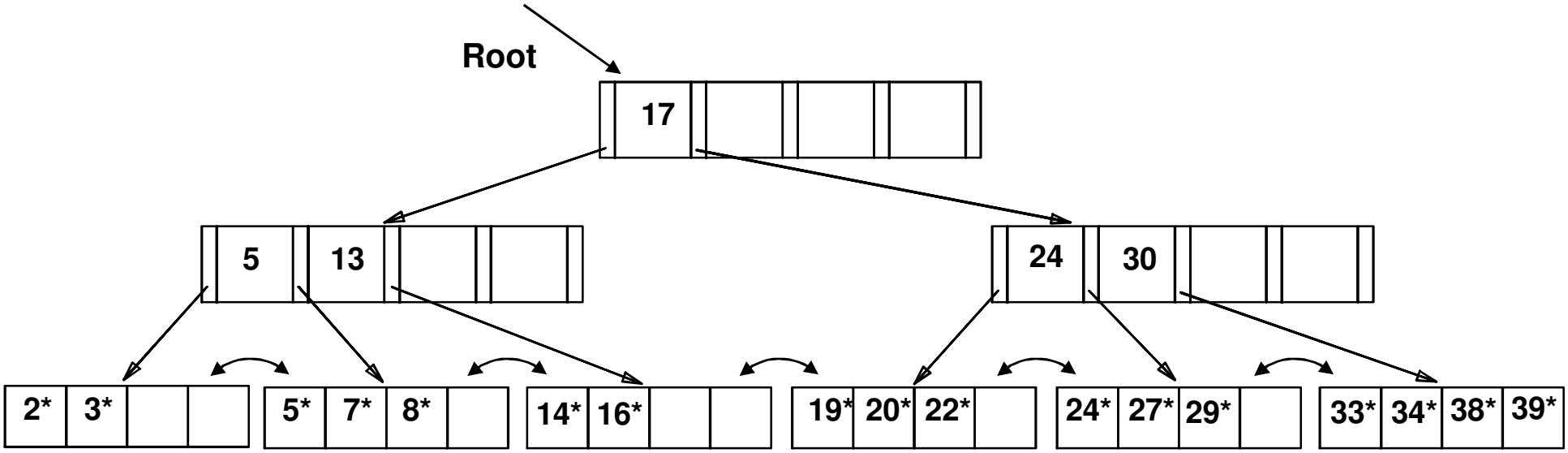


Inserimento in un B+-tree: esempio



Ogni nodo mantiene un numero di data entry superiore al rank **d** (=2) dell'albero

Inserimento in un B+-tree: esempio



→L'altezza dell'albero è aumentata

L'albero aumenta sempre in larghezza, tranne quando la radice è piena, caso in cui aumenta in altezza

Tecniche analoghe si possono applicare per la cancellazione

Indice ad albero B+-tree clustered

Premesse:

- Assumiamo l'alternativa (1): studi empirici hanno dimostrato che in un indice ad albero siffatto, le pagine delle foglie sono mediamente riempite per il 67%, e questo significa che il numero di pagine con data entry è circa 1.5B, dove B è il numero di pagine necessarie per i data record nel data file

  → numero di pagine fisiche per le foglie diventa B'=1.5B
- F è il numero di fan-out della struttura ad albero

1. Scansione: $1.5B(D+RC)$
2. Selezione basata su criterio di uguaglianza:

$$D\log_F(1.5B)+C\log_2 R$$

- localizzazione della (prima) pagina con data record di interesse
- localizzazione del (primo) data record con ricerca binaria nella pagina

N.B. Nella pratica, la radice è sempre nel buffer in quanto è acceduta di frequente → si risparmia un'operazione di I/O

Indice ad albero B+-tree clustered

3. Selezione in base ad un intervallo di valori:
   - come per selezione in base a criterio di uguaglianza
   - eventualmente, con ulteriori operazioni di I/O se data record di interesse comprese in altre foglie (collegate)
4. Inserimento: $D\log_F(1.5B)+C\log_2 R+C+D$
   - costo di ricerca + inserimento + scrittura
   - ignoriamo costi aggiuntivi che si presentano quando l'inserimento deve avvenire in una pagina piena (casi rari), che sono comunque costi logaritmici
5. Cancellazione
   - come per l'inserimento (se un solo data record da cancellare)
   - eventualmente ulteriori operazioni di cancellazione e di I/O se più data record da cancellare comprese in altre foglie
   - ignoriamo altri costi per mantenere bilanciato l'albero, che comunque sono costi logaritmici

Indice ad albero B+-tree unclustered

Premesse:

- Assumiamo l'alternativa (2), e supponiamo che un data entry di una foglia dell'indice abbia una dimensione pari ad un decimo di un data record
- numero di foglie dell'indice: 0.1(1.5B)=0.15B

→ numero di data entry in una pagina di indice (ricordiamo che una pagina con data entry è occupata al 67%):
10(0.67R)=6.7R

- denotiamo con F il fan-out dell'albero

1. Scansione: 0.15B(D+6.7RC) + BR(D+C)
   - scansione di tutti i data entry dell'indice: 0.15B(D+6.7RC)
   - ogni data entry in una foglia dell'indice può puntare ad una pagina differente del data file: BR(D+C)

   → costo molto più alto della scansione diretta del file: per questa operazione è preferibile IGNORARE l'indice

Indice ad albero B+-tree unclustered

2. Selezione in base a criterio di uguaglianza:

$$D\log_F(0.15B)+C\log_2(6.7R)+XD$$

   - localizzazione della (prima) pagina dell'indice con data entry di interesse: $D\log_F(0.15B)$
   - ricerca binaria del (primo) data entry: $C\log_2(6.7R)$
   - X: numero di data record che soddisfano il criterio
     → può richiedere un'operazione di I/O per ognuno
3. Selezione basata su di un intervallo di valori:
   - ragionamento analogo

N.B.:

→ Il costo dipende dal numero di data record selezionati (può diventare alto)

→ Se l'intervallo contiene un elevato numero di record, è preferibile non utilizzare l'indice, ma ordinare il file ed effettuare l'operazione direttamente sul file

Indice ad albero B+-tree unclustered

4. Inserimento:

$$2D+C+ D\log_F(0.15B)+C\log_2(6.7R)+C+D$$

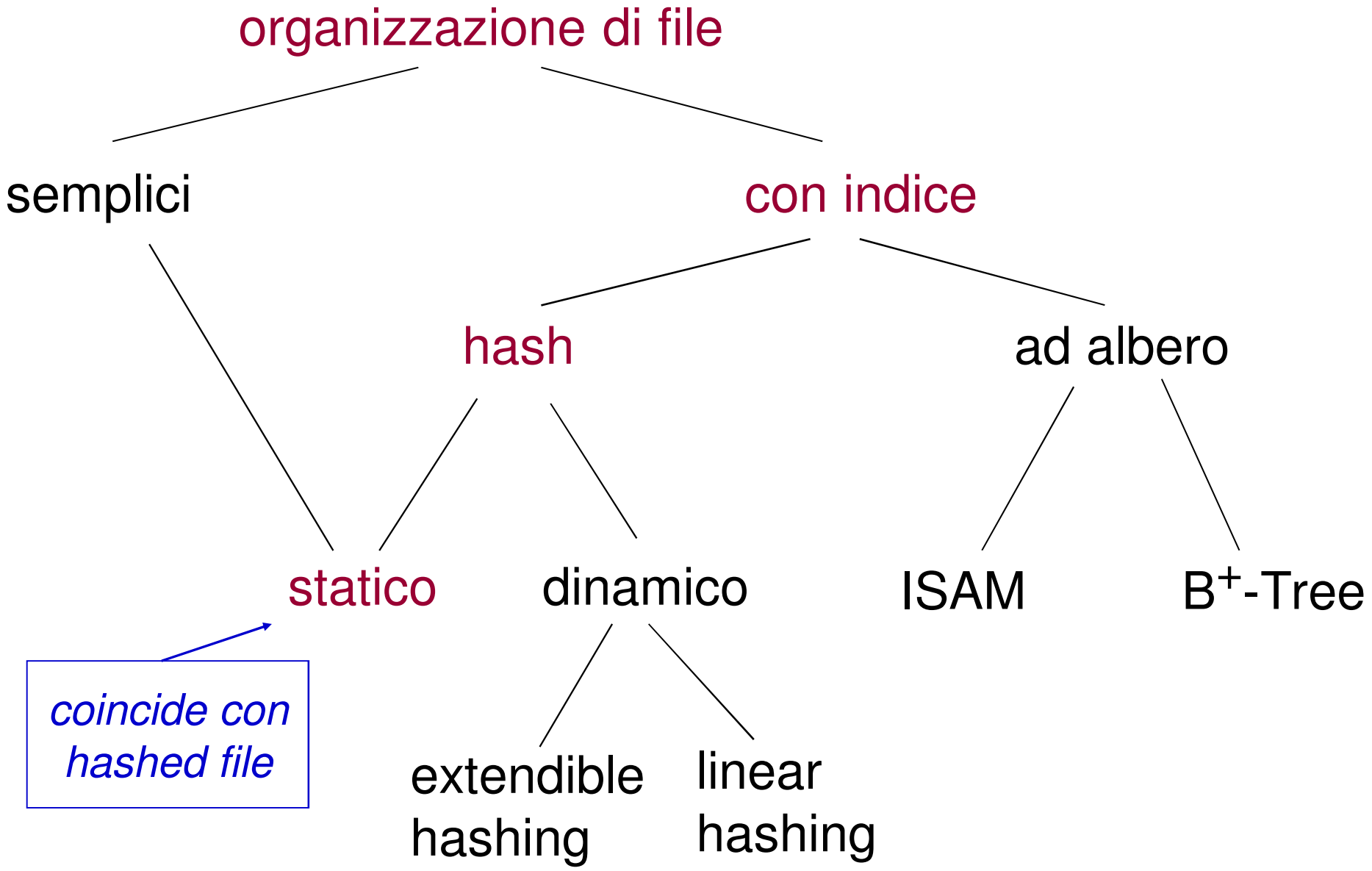
   - inserimento nel file (non ordinato) del data record: 2D+C
   - ricerca della posizione corretta nell'indice per inserire il data entry: $D\log_F(0.15B)+C\log_2(6.7R)$
   - modifica e scrittura della corrispondente pagina dell'indice: C+D

5. Cancellazione (caso di un solo data record):

$$D\log_F(0.15B)+C\log_2(6.7R)+D+2(C+D)$$

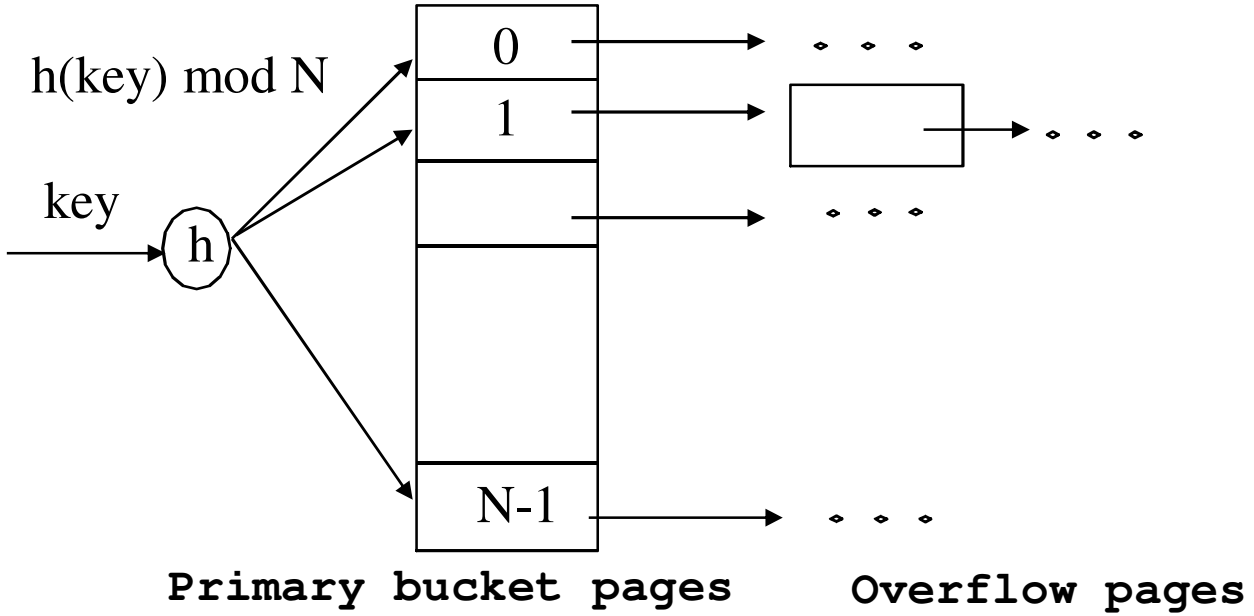
   - ricerca del data entry: $D\log_F(0.15B)+C\log_2(6.7R)$
   - caricamento della pagina con data record da cancellare: D
   - modifica+scrittura delle pagine modificate dell'indice e del file: 2(C+D)

Organizzazioni con indice



Static Hashing (Hashed file)

- Numero di pagine primarie fisse, allocate sequenzialmente, mai deallocate. Si allocano pagine di overflow se necessario
- **h**(*k*) mod N = bucket per i data entry con search key *k* (N = numero di bucket)



Static Hashing (Cont.)

- I bucket contengono *data entries*
- La funzione hash deve distribuire i valori nell'intervallo 0 ... N-1.
  - Usualmente, **h**(*key*) = (a**key* + b), con a,b constanti
  - Sono note tecniche per scegliere buone funzioni **h**
- Lunghe catene di overflow degradano le prestazioni
  - *Extendible* e *Linear Hashing*: tecniche dinamiche per affrontare questo problema

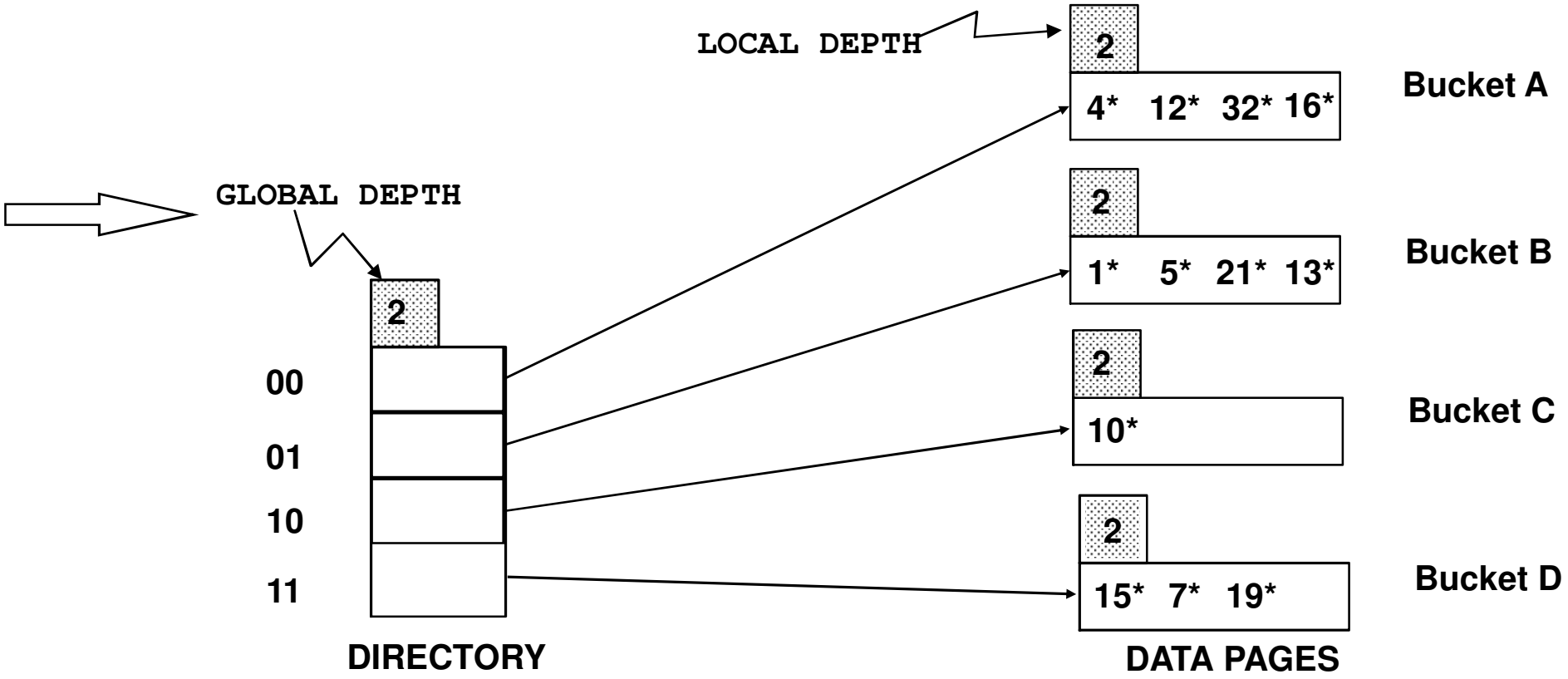
Organizzazioni con indice

Extendible Hashing

- Quando un bucket (primary page) diventa pieno, perché non riorganizzare il file raddoppiando il numero di bucket?
  - Leggere e scrivere tutte le pagine costa troppo
  - *Idea*: usare una *directory di puntatori a bucket*, raddoppiare il numero di bucket raddoppiando la sola *directory*, e dividere solo il bucket diventato pieno
  - La directory è molto più piccola del file, e raddoppiarla costa meno che raddoppiare tutti i bucket
  - Ovviamente, quando si raddoppia la directory, si deve aggiustare la funzione hash

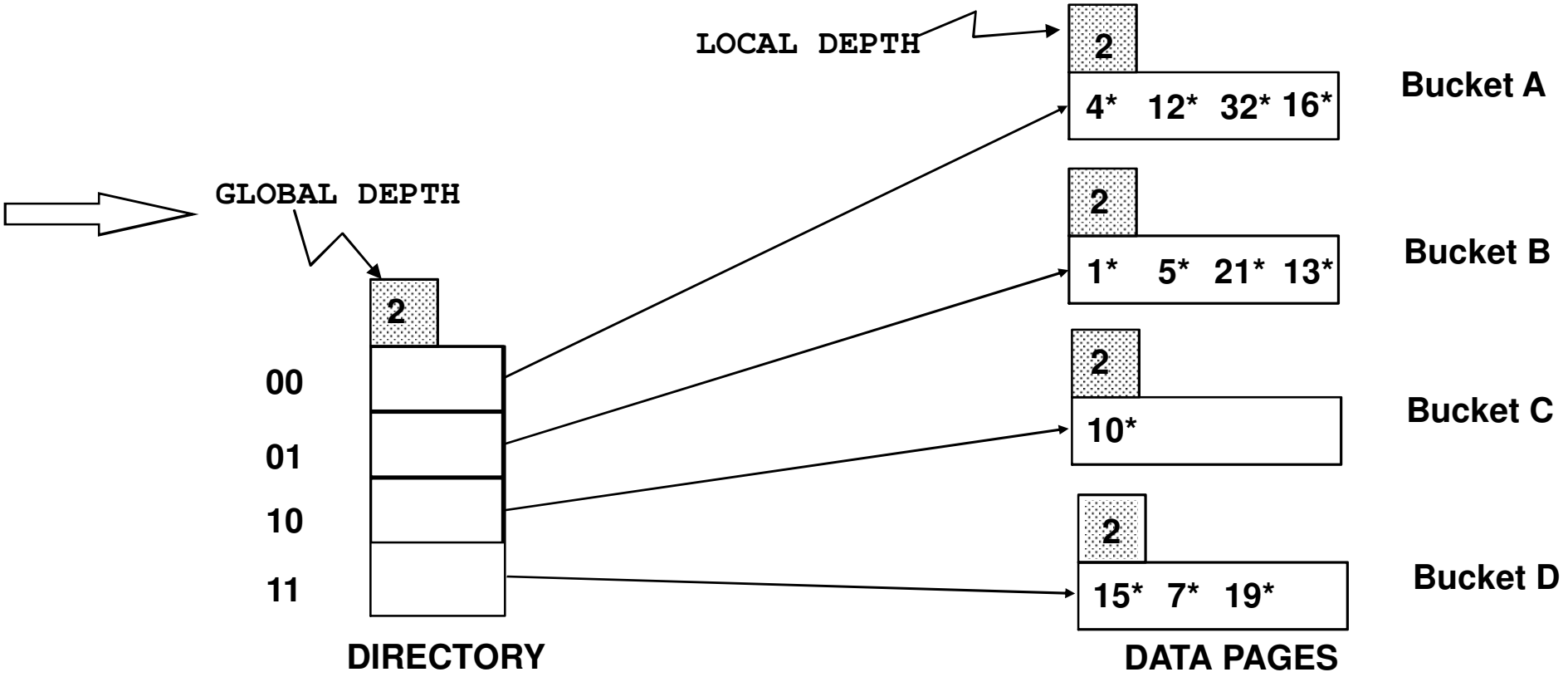
Esempio

- La directory è un array di 4 elementi
- Per trovare il bucket per *k*, considera gli ultimi g bit di **h**(*k*), dove g è la *global depth*
- Nella figura, denotiamo *k* con **h**(*k*)*
- Se **h**(*k*) = 5 = binary 101, allora 5* è nel bucket puntato da 01

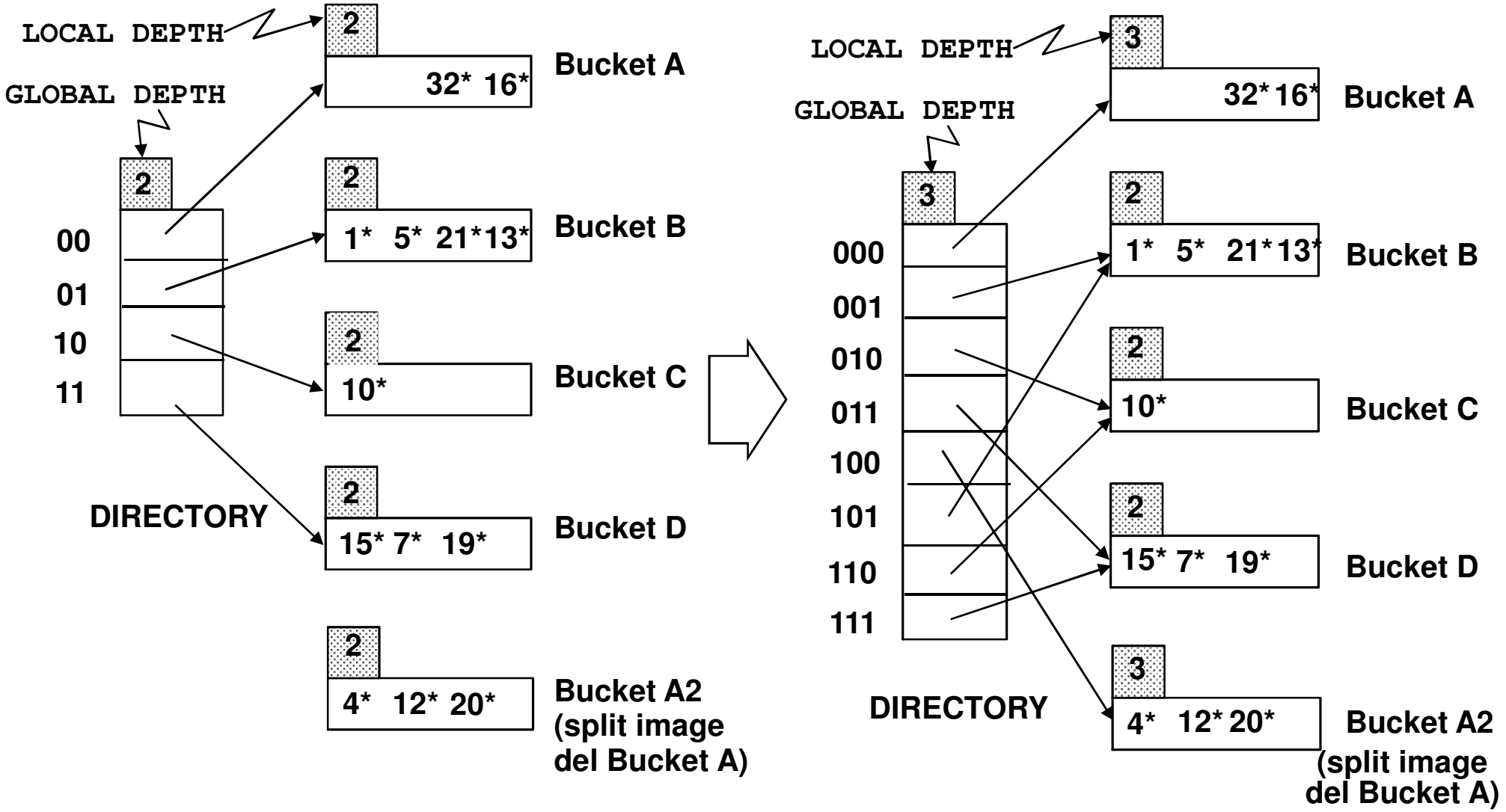


Inserimento di un data entry

- **Insert**: Se il bucket è pieno, dividilo (ridistribuendo i record in accordo con il valore restituito dalla funzione di hash)
- Se necessario, raddoppia la directory. Dividere un bucket non sempre richiede il raddoppio della directory: la decisione si basa sul confronto tra *global depth* e *local depth* del bucket diviso
- **Esempio**: insert h(r)=20*= (101)00



Insert h(r)=20* (raddoppio della directory)

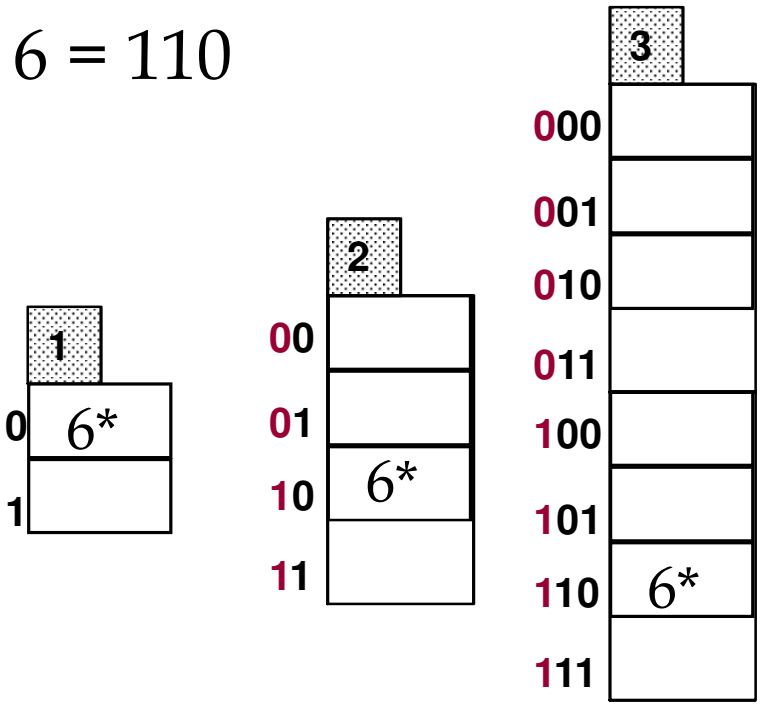


Osservazioni

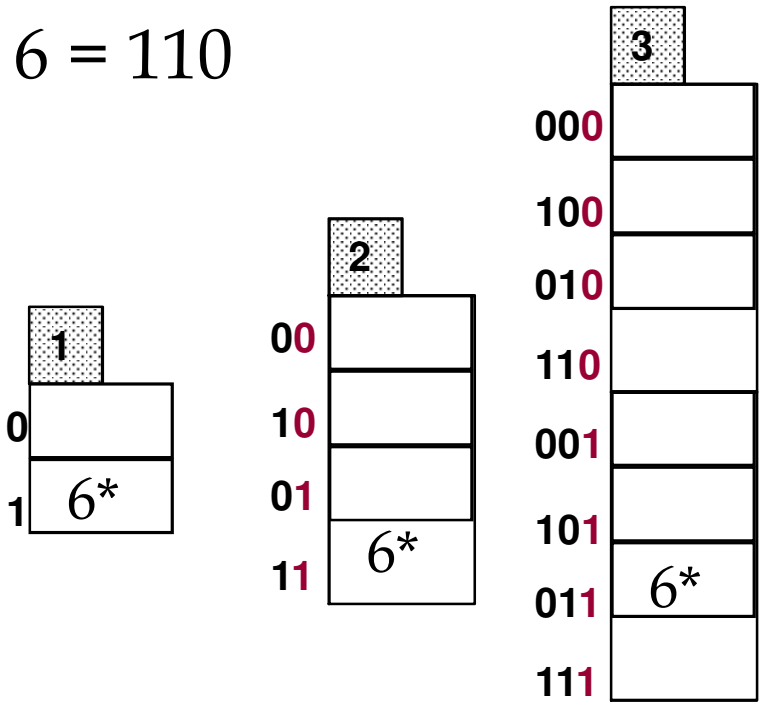
- 20 = binary 10100. Gli ultimi 2 bits (00) ci dicono che *r* appartiene ad A o ad A2. Abbiamo bisogno di un ulteriore bit per decidere quale.
  - *Global depth della directory*: Massimo numero di bit necessari per decidere a quale bucket appartiene un entry
  - *Local depth di un bucket*: numero di bit usati per determinare se un entry appartiene ad un bucket.
- Quando si divide un bucket e si raddoppia la directory?
  - Prima dell'inserimento, se vale la condizione *local depth del bucket* = *global depth,* allora Insert causerebbe *local depth* > *global depth*; ne segue che la directory deve essere raddoppiata, e questo lo si fa *copiandola ed aggiustando* poi il puntatore alla pagina corrispondente allo split image

Raddoppio della directory

Si usano i bit meno significativi perché questo consente il raddoppio della directory mediante la copia



Bit meno Significativi



Bit più significativi

Commenti su Extendible Hashing

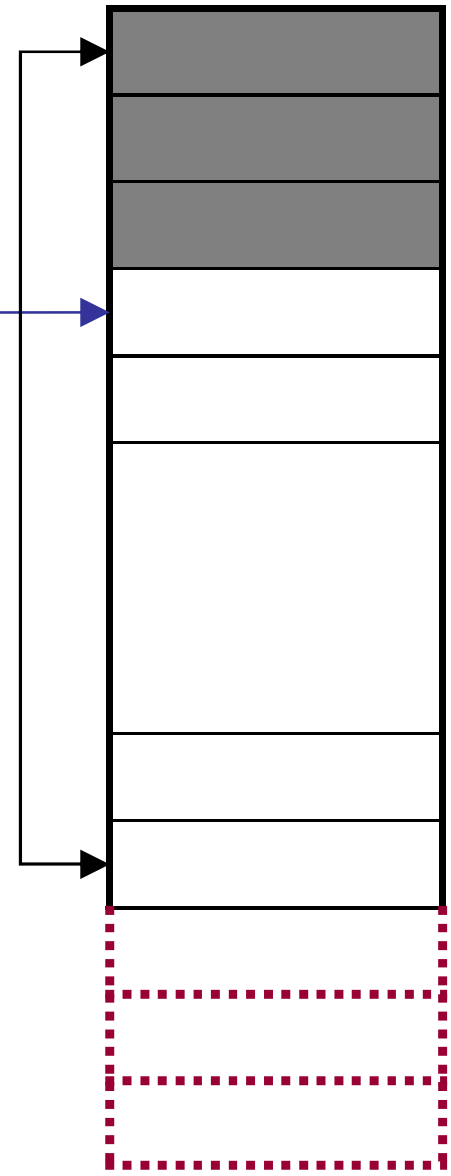
- Se la directory entra nel buffer, la ricerca con condizioni di uguaglianza si effettua con un unico accesso a pagine, altrimenti con due.
  - Esempio. File di 100MB, 100 bytes/rec, pagina di dimensione 4KB
    - 1.000.000 di records (data entries)
    - 40 records per pagina
    - 25,000 elementi della directory

  → la directory potrebbe entrare nel buffer!
  - La directory cresce a scatti, e se la distribuzione dei valori della funzione hash non è uniforme, allora la directory può diventare molto grande
  - Le collisioni possono causare pagine di overflow!
- **Delete**: Se la cancellazione di un data entry rende un bucket vuoto, allora esso può essere fuso con la sua split image. Se poi ogni elemento della directory punta allo stesso bucket a cui punta la sua split image della directory, allora possiamo dimezzare la directory

Linear Hashing

- Si tenta di evitare l'uso della directory (in modo da risparmiare un accesso)
- Le pagine primarie dei bucket (il cui numero iniziale indichiamo con N) sono memorizzate sequenzialmente
- Si organizza l'accesso in tornate. La variabile LEVEL (inizialmente uguale a 0) indica a quale "tornata" siamo arrivati
  - I bucket allocati all'inizio del round vengono sdoppiati dal primo all'ultimo, in maniera tale da ottenere alla fine del round un numero doppio di bucket
  - Si mantiene un riferimento (NEXT) al bucket che si deve sdoppiare per primo, con la regola che i bucket da 0 a NEXT – 1 sono stati già sdoppiati
- Si ha una certa flessibilità nello scegliere quando lo sdoppiamento deve avvenire:
  - quando una qualunque pagina di overflow viene allocata (caso dei nostri esempi)
  - quando una certa condizione sullo spazio utilizzato è soddisfatta

Situazione dei bucket alla tornata LEVEL



Bucket sdoppiati
nella tornata corrente

NEXT:

prossimo bucket
a essere sdoppiato

Bucket presenti all'inizio della tornata corrente (range di h_{LEVEL})

"Bucket immagini" creati dallo sdoppiamento di altri bucket nella tornata corrente

Sdoppiamento di bucket

- In sostanza si usa una famiglia di funzioni hash
  - h_0,
  - h_1,
  - h_2,
  - ….

  tali che se h_i va in M bucket, allora h_{i+1} va in 2M
- Per gestire lo sdoppiamento, quando si cerca un valore k, si applica la funzione hash h_{LEVEL} e se questa va a finire su un bucket T:
  - se il bucket non e' stato sdoppiato (T≥NEXT), allora si cerca il data entry k* in T
  - altrimenti, si usa la funzione hash $h_{LEVEL+1}$ per decidere se andare su T oppure sulla sua immagine sdoppiata

Famiglia di funzioni hash

- La famiglia si definisce tipicamente così:

$$h_i(v) = h(v) \bmod 2^i N$$

dove h è la funzione hash principale ed N è il numero di bucket iniziali

- Se si sceglie N come potenza di 2, allora il calcolo per $h_i(v)$ può consistere nel restituire gli ultimi d_i bit di h(v), dove d_0 è il numero di bit necessario per rappresentare N (cioè log N), e $d_i = d_{i-1} + 1$

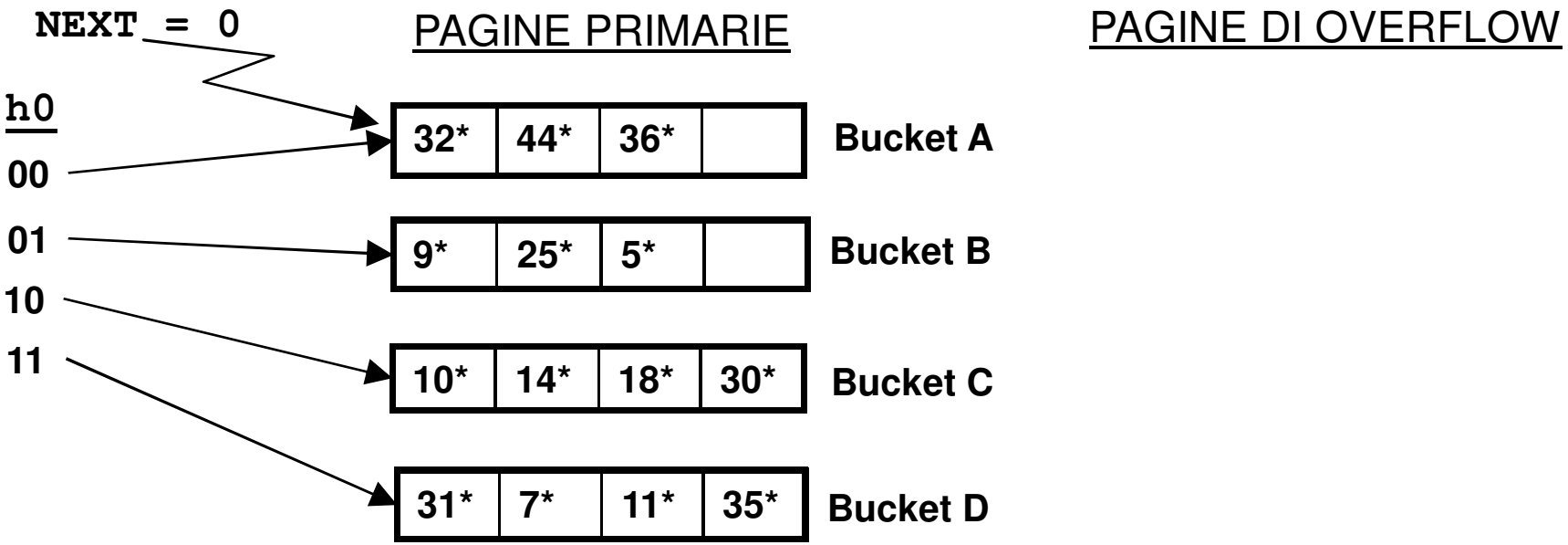
Esempio

Fissiamo N = 32. Si hanno i seguenti valori per i vari parametri in gioco:

- $d_0 = 5$
  - $h_0(v) = h(v) \bmod 32$
- $d_1 = 6$
  - $h_1(v) = h(v) \bmod 64$
- $d_2 = 7$
  - $h_2(v) = h(v) \bmod 128$
- ……

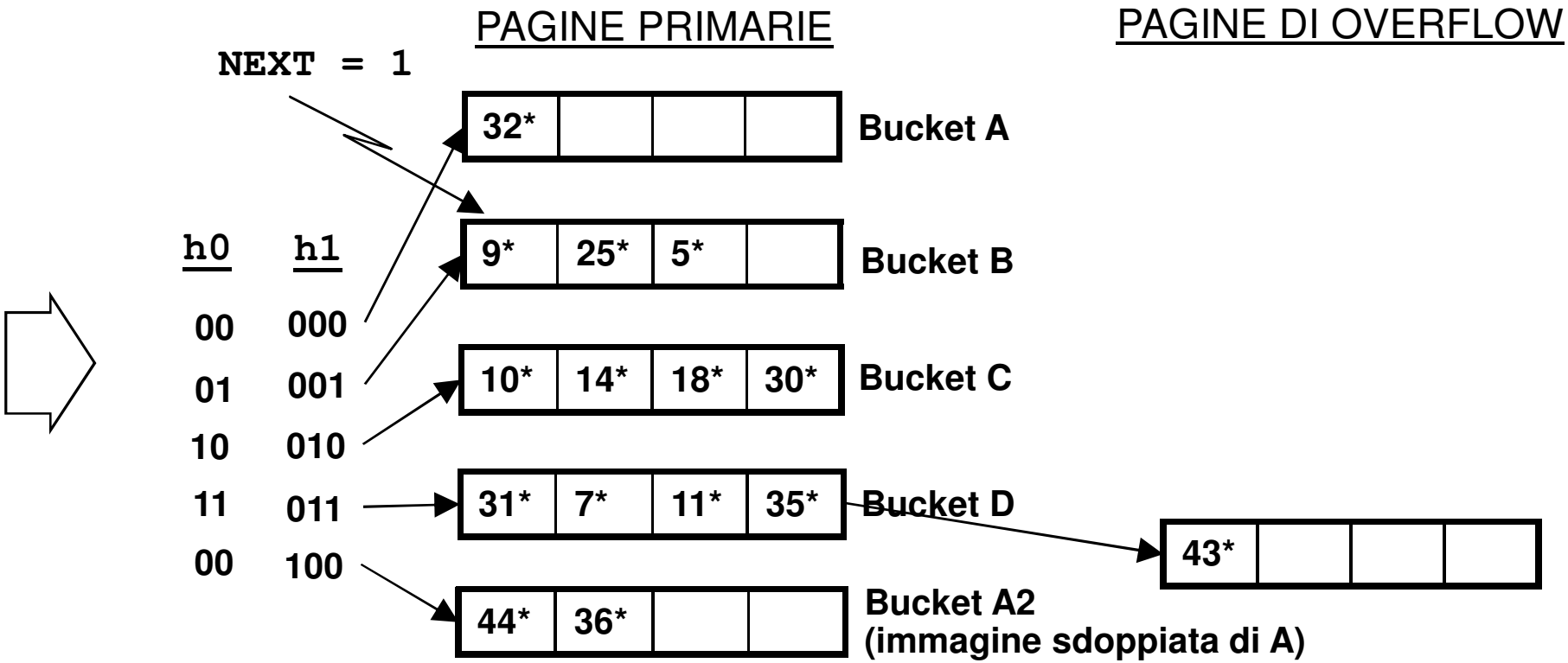
Insert h(r)=43*

- Ogni bucket contiene 4 entry
- Inizialmente, LEVEL = 0, N = 4, NEXT=0
- $h_0(r)$ = 43* mod N = (1010)11
- Poiché si va a finire in un bucket pieno, si alloca una pagina di overflow → si sdoppia il bucket NEXT…



Insert h(r)=43* (si sdoppia il bucket NEXT)

- I data entries nel bucket NEXT vengono ridistribuiti tra il bucket e la sua immagine sdoppiata, in accordo con la funzione di hash $h_{LEVEL+1}$
- Diversamente dall'extendible hashing, quando un inserimento causa uno sdoppiamento, allora il dato non viene necessariamente inserito all'interno del bucket sdoppiato (cf. esempio sotto)



Linear Hashing: osservazioni (1)

- Durante la tornata LEVEL, solo le funzioni di hash h_{LEVEL} e $h_{LEVEL+1}$ sono in uso
- L'immagine di un bucket di numero b è il bucket b + N_{LEVEL}, dove N_{LEVEL} è il numero di bucket quando la tornata è LEVEL, ed è definito come N * 2^{LEVEL} (N è il numero di bucket iniziali)
- Se la funzione hash ci dà un numero tra NEXT ed N_{LEVEL}, allora sappiamo che il bucket non è sdoppiato
- Se invece il risultato è tra 0 e NEXT-1, allora sappiamo che è sdoppiato, e dobbiamo ricorrere alla nuova funzione hash (guardando l'ulteriore bit) per trovare il bucket giusto

Linear Hashing: osservazioni (2)

- Esempio:
  - Il valore $h_0(18)=10_2$ ci dà un numero tra NEXT ed N_{LEVEL}, perciò il bucket giusto è 2 ($=10_2$)
  - Il valore $h_0(32)=00_2$ ci dà un numero tra 0 e NEXT-1; $h_1(32) = 000_2$, perciò il bucket giusto è 0
  - Il valore $h_0(44)=00_2$ ci dà un numero tra 0 e NEXT-1; $h_1(44)=100_2$, perciò il bucket giusto è 4
- Ovviamente, ad un certo punto tutti i bucket saranno sdoppiati: si passa ad una nuova tornata, ovvero si aumenta LEVEL di 1, si pone NEXT = 0, ed è come se il codominio della funzione hash fosse raddoppiato (questa situazione è analoga al raddoppio della directory nell'extendible hashing)
- **Delete**: duale della insert (se l'ultimo bucket "primario" rimane vuoto, allora NEXT viene decrementato,…)

Extendible and Linear Hashing

- Supponiamo di avere una directory anche nel Linear Hashing
  - inizialmente (bucket presenti da 0…N-1), quando viene allocata una pagina di overflow, si sdoppia il bucket puntato da 0 e si aggiunge nella directory un elemento N
  - in principio, si puo' immaginare che l'intera directory sia stata sdoppiata, ma poiche' l'elemento i punta allo stesso data entry a cui punta l'elemento N+i allora e' possibile evitare la copia degli altri elementi della directory (che non sarebbero comunque puntati da nessuno)

    → alla fine della tornata la dimensione della directory e' stata raddoppiata
- Idea: allocando i bucket primari sequenzialmente (in modo che il bucket i possa essere localizzato calcolando un semplice "salto" di i pagine), l'utilizzo della directory e' evitato nel Linear Hashing
- Extendible Hashing vs Linear Hashing:
  - Extendible: poiche' sdoppia il bucket piu' appropriato (quello pieno) puo' portare a meno sdoppiamenti
  - Linear: permette di tenere in media meno bucket allocati "vuoti" per funzioni di hash con distribuzioni uniformi di valori + richiede un accesso in meno per la directory

Confronto delle prestazioni

- Confronto sul costo di semplici operazioni su diverse organizzazioni di file per una collezione di data record

- In caso di file con struttura di accesso, il costo delle operazioni di selezione viene calcolato assumendo che l'indice sia composto e che le operazioni vertano su almeno il primo campo della chiave di ricerca composta (altrimenti costo uguale al caso di file non ordinato)

Tabella riassuntiva di confronto

Tipo di file	Scansione	Ricerca con criterio di uguaglianza	Ricerca con intervallo di valori	Inserimento	Cancellazione
Heap file	BD	BD	BD	2D	Costo della ricerca + D
File ordinato	BD	$D\log_2 B$	$D(\log_2 B$ + num pagine interessate)	Ricerca + 2BD	Ricerca + 2BD
Indice ad albero clustered	1.5BD	$D\log_F(1.5B)$	$D(\log_F(1.5B)$ +num pagine interessate)	Ricerca + D	Ricerca + D
Indice ad albero unclustered	BD* (R+0.15)	$D(\log_F(0.15B)$ + num record di interesse)	$D(\log_F(0.15B)$ + num record di interesse)	$D(3+ \log_F(0.15B))$	Ricerca + 2D
Indice hash statico unclustered	1.25BD	1.25D	1.25BD	2D	Ricerca + D

Osservazioni

- Heap file:
  - efficiente in termini di occupazione di spazio
  - scansione e inserimento veloci
  - ricerche e cancellazione lente

- File ordinato:
  - efficiente in termini di occupazione di spazio
  - inserimento e cancellazione lente
  - ricerche più veloci rispetto al caso dei file disordinati

Osservazioni

- Indice ad albero clustered:
  - piccolo overhead in termini di spazio di occupazione
  - inserimento e cancellazione efficienti
  - ricerche piuttosto veloci
  - è la migliore organizzazione per ricerche con intervallo di valori

- Indice hash statico:
  - ricerca con criterio di uguaglianza, inserimento e cancellazione veloci
  - è la migliore organizzazione per ricerche con uguaglianza
  - scansione e ricerca con intervallo di valori lente

→ **Nessuna organizzazione di file è uniformemente superiore in tutte le situazioni**

5.4
Valutazione di query

Valutazione degli operatori relazionali

Il nostro scopo è analizzare come il motore SQL calcola il risultato di query.

In realtà, assumeremo che il motore SQL nel valutare le query effettui una traduzione in algebra relazionale, ed analizzeremo quindi i metodi per valutare epressioni dell'algebra costituite da un solo operatore.

Gli operatori su cui ci concentreremo sono: Selezione, Proiezione e Join

Molti dei concetti che analizzeremo si basano sulla nozione di access path

Access path

L'access path è il metodo per accedere ai record che consentono di realizzare una operazione dell'algebra relazionale

1. File scan
2. Indice + matching selection condition

Un indice si dice conforme a

attr op value

se

- l'indice è ad albero, la search key è attr, e op è <, <=, =, <>, >=, oppure >
- l'indice è hash, la search key è attr, e op è =

L'access path più selettivo è quello che consente di accedere a meno pagine

Access path

Prefisso di una search key: segmento iniziale non vuoto di attributi per una search key composta da una sequenza di attributi

Esempio: per una search key <a,b,c>, si ha che <a> e <a,b> sono prefissi, mentre <a,c> e <b,c> non lo sono

Un indice si dice conforme a

(attr op value) and (attr op value) and …

se

- l'indice è ad albero, ed esiste un prefisso della search key tale che c'è un termine (attr op value) nella congiunzione per ogni attributo in tale prefisso; tali termini si dicono i termini primari della congiunzione
- l'indice è hash, e c'è un termine (attr = value) nella congiunzione per ogni attributo della search key

Esempi

- Un indice hash con search key <r,b,s> è conforme alla condizione (r = 'J' and b = 5 and s = 3), ma non alla condizione (r = 'J' and b = 7)
- Un indice ad albero con search key <r,b,s> è conforme alla condizione (r > 'J' and b = 7 and s = 5), e alla condizione (r = 'H' and s = 9), ma non alla condizione (b = 8 and s = 10)
- Un indice qualunque con search key <b,s> è conforme alla condizione (d > 1000 and b = 5 and s = 7): dopo aver trovato gli entry per le condizioni sui termini primari b = 5 e s = 7, si selezionano solo quelli che soddisfano l'ulteriore condizione d > 1000
- Un indice qualunque con search key <b,s> ed un indice ad albero con search key <d> sono entrambi conformi alla condizione (d > 1000 and b = 5 and s = 7):
  - se usiamo il primo, dopo aver trovato le entry selezioniamo ulteriormente quelle che verificano l'ulteriore condizione d>1000;
  - se viceversa usiamo il secondo, dopo aver trovato le entry selezioniamo ulteriormente quelle che verificano le ulteriori condizioni b=5 e s=7

Ordinamento in memoria secondaria

Nelle considerazioni che seguono faremo riferimento al problema di ordinare i record di un file (secondo una chiave di ordinamento) in memoria secondaria. Esistono diversi algoritmi di ordinamento in memoria secondaria (external sorting).

Noi faremo riferimento all'algoritmo "merge-sort", la cui complessità rispetto al numero di accessi a pagine assumeremo che sia $O(N \log_H N)$, dove

- N è il numero di pagine del file,
- si è assunto che il buffer contenga H slot liberi da dedicare all'algoritmo

Selezione

Analizziamo il caso di una query composta da:

```
select *
from R
where <attr op valore>
```

distinguendo vari casi, a seconda del metodi di rappresentazione della relazione R.

Assumeremo che R sia costituita da M pagine, e, quando vorremo riferirci ad un esempio concreto, assumeremo che M = 1000 e che in una pagina entrino 10 record.

Selezione

Caso 1) Relazione R rappresentata con heap file

- Access path: scan
- Il costo in termini di accessi è O(M)
- Nell'esempio, il costo è circa 1000

Caso 2) Relazione rappresentata con file ordinato

- Assumiamo che la condizione "where" sia relativa alla chiave di ordinamento (altrimenti il file è come se non fosse ordinato)
- Access path: binary search (in realtà il file ordinato nella pratica viene rappresentato con un B+-tree, e quindi l'access path sarebbe con indice B+-tree)
- Con M pagine, il costo è $O(\log_2 M)$ + numero di pagine con record di interesse
- Se M = 1000, $\log_2 M$ è circa 10

Selezione

Caso 3) Relazione rappresentata con indice B+-tree

- Assumiamo che la condizione “where” sia relativa alla search key (altrimenti l’indice non è conforme, ed è come se non ci fosse)
- Access path: usa l’indice
- L’indice si usa così:
  - Cerca il primo record (con costo tra 2 e 4)
  - Ulteriore eventuale costo dipende da
    - quanti sono i record di interesse successivi
    - se l’indice è clustered o no
- Si noti che la valutazione di 2-4 come costo di accesso al primo record è realistica. Infatti, tale intervallo corrisponde (con un fan-out di 100) ad un intervallo di 10.000 – 100.000.000 pagine nelle foglie (corrispondente ad un intervallo di 100.000 – 1.000.000.000 di record nel data file, se in ogni pagina delle foglie entrano 10 data entries), che diventa 1-3 se la radice è nel buffer.

Selezione

Caso 3) Relazione rappresentata con indice B+-tree

- Se l'indice è clustered, allora ci si aspetta pochi altri accessi oltre a quello al primo record
- Se l'indice è unclustered, allora ogni altro data entry può in generale puntare a pagine diverse.
  - Possiamo fare meglio se ordiniamo i rids nella pagina dell'indice sulla base della componente page-id. Questo infatti assicura che quando portiamo una pagina nel buffer, tutti i record di interesse che soddisfano la condizione sono acceduti uno dopo l'altro, ed il numero di accessi è uguale al numero di pagine che contengono i record di interesse

Se assumiamo che il 10% dei record della relazione soddisfi la condizione di selezione (ovvero 10.000 tuple per 100 pagine sulle 1000 totali), allora con indice clustered si ha un costo di 2-4 più 100, mentre con indice non clustered, abbiamo 2-4 più 10.000 pagine (si noti che scan darebbe 1000!). Se l'indice è unclustered, ma ordiniamo i rids, allora le cose migliorano, ma è probabile che i record di interesse si trovino comunque su più di 100 pagine (per cui le prestazioni dell'indice degradano rispetto al caso clustered).

Selezione

Caso 4) Relazione rappresentata con indice hash

- Assumiamo che la condizione “where” sia relativa alla search key (altrimenti l’indice non è conforme, ed è come se non ci fosse)
- Access path: usa l’indice se conforme, e se la condizione è di uguaglianza
- Se la search key è chiave della relazione, il costo è 1-2 accessi
- Si noti che il costo si valuta in 1-2 accessi perché si considera accettabile una situazione in cui siamo forzati ad effettuare al massimo un accesso ad una pagina di overflow
- Se la search key non è chiave della relazione, si deve aggiungere il costo di reperire gli ulteriori record di interesse oltre al primo, ma comunque (essendo la condizione di uguaglianza) tali record si trovano nel bucket corrente, per cui è ancora realistico un costo pari a 1-2 accessi

Proiezione

Il problema nella proiezione è l'eliminazione dei duplicati. Sono popolari due algoritmi:

1. Sort-based projection
2. Hash-based projection

Noi analizzeremo solo il primo, che ha la seguente struttura:

1. Fai lo scan della relazione R su cui stai operando la proiezione, producendo su un file temporaneo i record con solo gli attributi voluti
2. Ordina il file temporaneo (indichiamo con T la sua dimensione in pagine) utilizzando come chiave di ordinamento tutti gli attributi rimasti
3. Fai lo scan del file temporaneo ordinato, eliminando i record contigui uguali, e scrivendo il risultato in un nuovo file

Costo di (1): $O(M) + O(T)$ dove $T = O(M)$
Costo di (2): $O(T \log_H T)$ (H= numero slot liberi del buffer per il mergesort)
Costo di 3): $O(T)$
Riassumendo: il costo totale è $O(M \log_H M)$

Proiezione con “index-only scan”

Se dobbiamo calcolare la proiezione su A1,…,An della relazione R ed abbiamo un indice (in particolare B+-tree) la cui search key comprende tutti gli attributi A1,…,An, allora possiamo operare la cosiddetta “index-only scan”

Essa consiste nel fare lo scan delle foglie dell’indice, risparmiando notevolmente rispetto alla scan del data file

Se poi A1,…,An sono un prefisso della search key, allora durante lo scan possiamo anche più efficientemente (basandoci in particolare su entry adiacenti):

- eliminare gli attributi non voluti dal risultato
- eliminare i duplicati

Join

Ci riferiremo all'equi join, effettuato sulle relazioni R ed S. Quando ci riferiremo ad un esempio, assumeremo

- R con M pagine e p_R tuple per pagina
- S con N pagine e p_S tuple per pagina

e considereremo i seguenti valori: M=1000, N=500, p_R=100, p_S=10, ed assumeremo 10 ms per accesso. Inoltre, quando valuteremo il costo, ignoreremo il costo dell'operazione di scrittura del risultato.

Analizzeremo i seguenti algoritmi:

- Nested loop
- Block nested loop
- Index nested loop
- Sort merge join

Nested loop (versione naive)

Si sceglie una delle due relazioni (diciamo R) e la si considera come "relazione esterna (outer relation)", mentre l'altra si usa come "relazione interna (inner relation)".

Fai lo scan dell'outer relation R, e:
per ogni pagina P di R:
per ogni record E in P:
fai lo scan di S, cercando i record in join con E

Il costo è $M + (p_R \times M \times N)$, ed è proibitivo. Infatti, con i valori indicati prima, si ha che il costo del join di R e S effettuato con l'algoritmo "nested loop naive" è

$$(1000 + (100 \times 1000 \times 500)) \times 10\text{ms} = (1000 + 5 \times 10^7) \times 10\text{ms}$$

che corrisponde a circa 139 ore.

Nested loop

Come prima, si scelgono l'outer relation e la inner relation.

fai lo scan dell'outer relation R, e:
per ogni pagina P di R:
fai lo scan di S, cercando i record in join con i record in P

Il costo è $M + (M \times N)$. Con i valori indicati prima, si ha che il costo del join di R e S effettuato con l'algoritmo nested loop è

$$(1000 + (1000 \times 500)) \times 10\text{ms} = 5.010.000\text{ms}$$

che corrisponde a circa 1,5 ore.

Conviene scegliere la relazione più piccola come outer relation. Nel nostro esempio, se scegliamo S come outer relation, il costo migliora, ma di poco (diventa 5.005.000ms).

Block nested loop

Supponiamo che l'outer relation stia nel buffer, e che nel buffer ci siano anche 2 pagine extra. Possiamo portarci tutta l'outer relation nel buffer, ed usare una delle extra pagine del buffer per leggere la inner relation (una pagina alla volta) ed usare l'altra pagina per scrivere il risultato (una pagina alla volta).

Il costo è M + N, che è ovviamente ottimale. Nel nostro esempio, il costo diventa 1500 × 10ms, che corrisponde a 15 secondi (1500*10ms=15 secondi).

Block nested loop

Nella pratica, sarà impossibile che l'outer relation stia tutta nel buffer, ma sarà probabile che si possano usare G pagine del buffer (di cui 2 come pagine extra, come detto prima).

L'algoritmo diventa: porta G-2 pagine dell'outer relation nel buffer, fai lo scan della inner relation portando in una pagina extra del buffer una pagina alla volta, e per ognuna di tali pagine calcola le tuple in join con quelle dell'outer relation che si trovano nelle G pagine del buffer, scrivendo le tuple del risultato nell'altra pagina extra.

Se l'outer relation è R, il costo è $M + (N \times M/(G-2))$. Ovviamente, affinchè l'algoritmo sia efficace, B deve essere sufficientemente grande. Nel nostro esempio, assumendo G=102, e scegliendo come outer relation la più piccola, si ha un costo di $(500 + (1000 \times 500/100)) \times 10ms = 55000ms$, che corrisponde a circa un minuto.

Index nested loop

L'algoritmo di index nested loop si può utilizzare se c'è un indice sulla inner relation, e tale indice è conforme alla condizione di equi join

Se la inner relation è S, allora l'algoritmo è così:

```
fai lo scan di R, e per ogni pagina P di R:
    per ogni tupla t nella pagina P:
        usa l'indice su S per trovare le tuple di S che sono in join con t
```

Il costo dipende ovviamente dal tipo di indice (B+-tree o hash) e da quante tuple della inner relation sono in join con una tupla della outer relation. In particolare, se l'attributo di equi join è chiave per la inner relation S, allora al massimo una tupla della inner relation è in join con una tupla della outer relation

Index nested loop

Assumendo di avere un indice hash su S (inner relation) e che l'attributo di equi join sia chiave per S, abbiamo 100.000 tuple per R, e contando 1,2 il costo per il singolo accesso con l'indice hash, il costo è 1000 + 100.000 × 1,2, ovvero 121.000, che corrisponde a circa 20 minuti.

Assumendo invece che l'indice su S (inner relation) sia B+-tree clustered, che l'attributo di equi join non sia chiave per S, che per ogni valore dell'attributo di equi join di R si abbiano in media 2 tuple in S i cui corrispondenti index entries si trovino nella stessa pagina, ed assumendo che 3 sia la profondità dell'indice ad albero, il costo è (1000 + 100.000 × 3) × 10ms = 3.010.000ms, che corrisponde a circa 50 minuti.

Se invece l'indice B+-tree non è clustered, allora il costo deve tenere conto dell'accesso in più per ogni utilizzo dell'indice, e diventa (1000 + 100.000 × 4) × 10ms = 4.010.000ms (circa 1 ora e 7 minuti).

Sort merge join

L'algoritmo di sort merge join è così:

1 - ordina (con il merge sort) R sull'attributo di equi join
2 - ordina (con il merge sort) S sull'attributo di equi join
3 - fai lo scan delle pagine di R (outer relation), e per ogni tupla di ogni pagina, cerca le tuple in join nella pagina corrente di S (inner relation)

Costo dei passi 1 e 2:

Come detto in precedenza, assumiamo che l'ordinamento in memoria secondaria sia effettuato mediante algoritmo merge-sort, la cui complessità rispetto al numero di accessi a pagine è $O(P \log_H P)$, dove P è il numero di pagine del file, e H è il numero di slot liberi che il buffer ha da dedicare all'algoritmo.

Pertanto:

- il costo del passo 1 è $M \log_H M$
- il costo del passo 2 è $N \log_H N$

Sort merge join

Costo del passo 3:

Nel caso peggiore (tutte le tuple di R sono in join con tutte le tuple di S) l'algoritmo costa M × N. Nei casi favorevoli il costo è nettamente inferiore. Ad esempio, quando l'attributo di equi join è chiave per la inner relation, allora ogni pagina di S viene analizzata una sola volta, ed il costo è M + N (ottimo). Mediamente, anche se una pagina di S viene richiesta più volte, è possibile che sia ancora nel buffer, e la valutazione M + N per il costo dell'algoritmo è ragionevole.

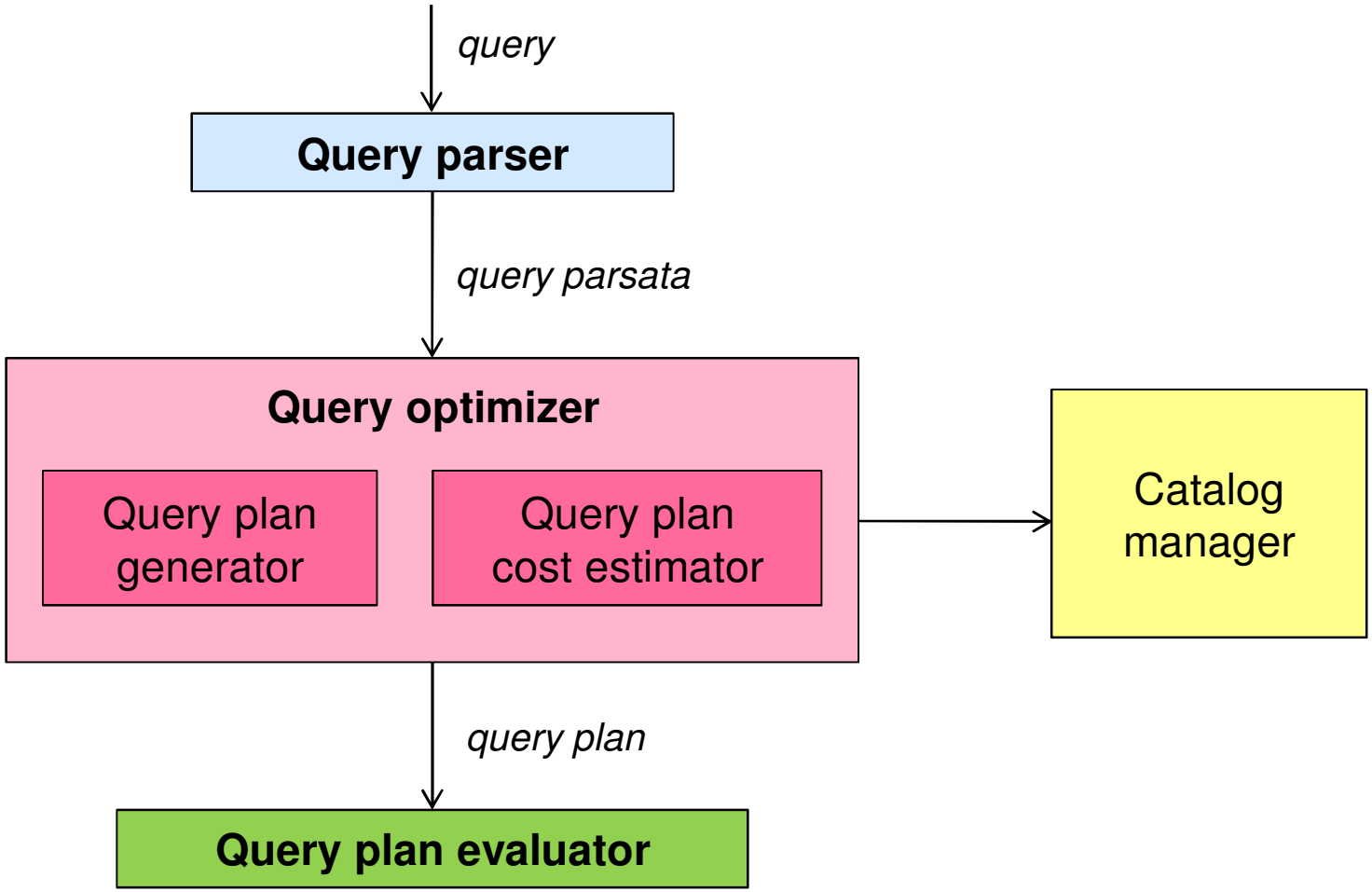
Nel nostro caso, supponendo che il buffer abbia 10 slot (pagine) disponibili per il merge sort, si ha che:

- il costo di ordinare R (passo 1) è 1000 $\log_{10}1000 = 3000$
- il costo di ordinare S (passo 2) è 500 $\log_{10}500 = 1500$
- il costo del passo 3 è 1000+500= 1500

Il costo totale è pertanto 6000 × 10ms, che corrisponde a un minuto.

Valutazione di query in un DBMS

Struttura del motore SQL:

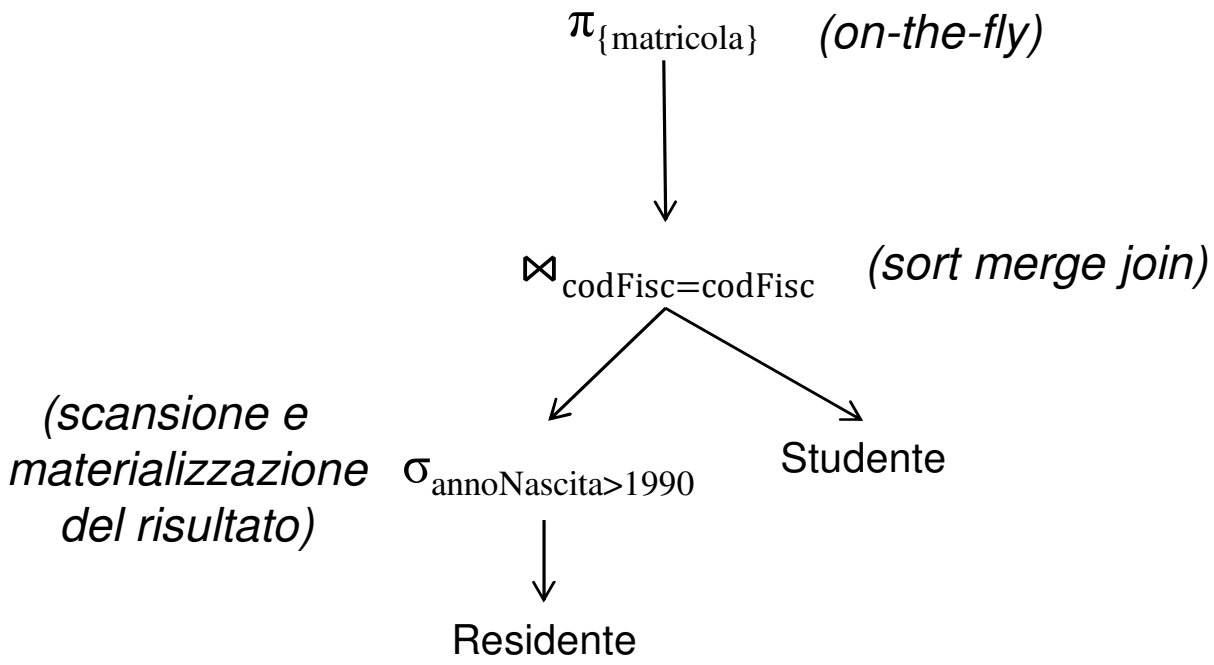


Query optimizer

- Il **query optimizer** è il modulo che si occupa di definire il piano di valutazione della query SQL (query evaluation plan)
- Un **query evaluation plan** (o semplicemente **query plan**) è un albero i cui nodi sono operatori dell'algebra relazionale (select, project, join, ecc.) annotati da una descrizione dei metodi di accesso alle relazioni e della modalità di esecuzione di tale operazione
- Il query optimizer è costituito da:
  - **Query plan generator**: è un modulo che genera un insieme di possibili query plan, ognuno dei quali corrisponde ad un possibile piano di esecuzione della query SQL
  - **Query plan cost estimator**: per ogni query plan generato dal query plan generator, calcola una stima del costo di esecuzione di tale piano.
- Il query optimizer sceglie, tra tutti i query plan generati, quello con costo stimato minimo

Query plan

Esempio di query plan:



Query plan

- Esempio: data la seguente query Q:

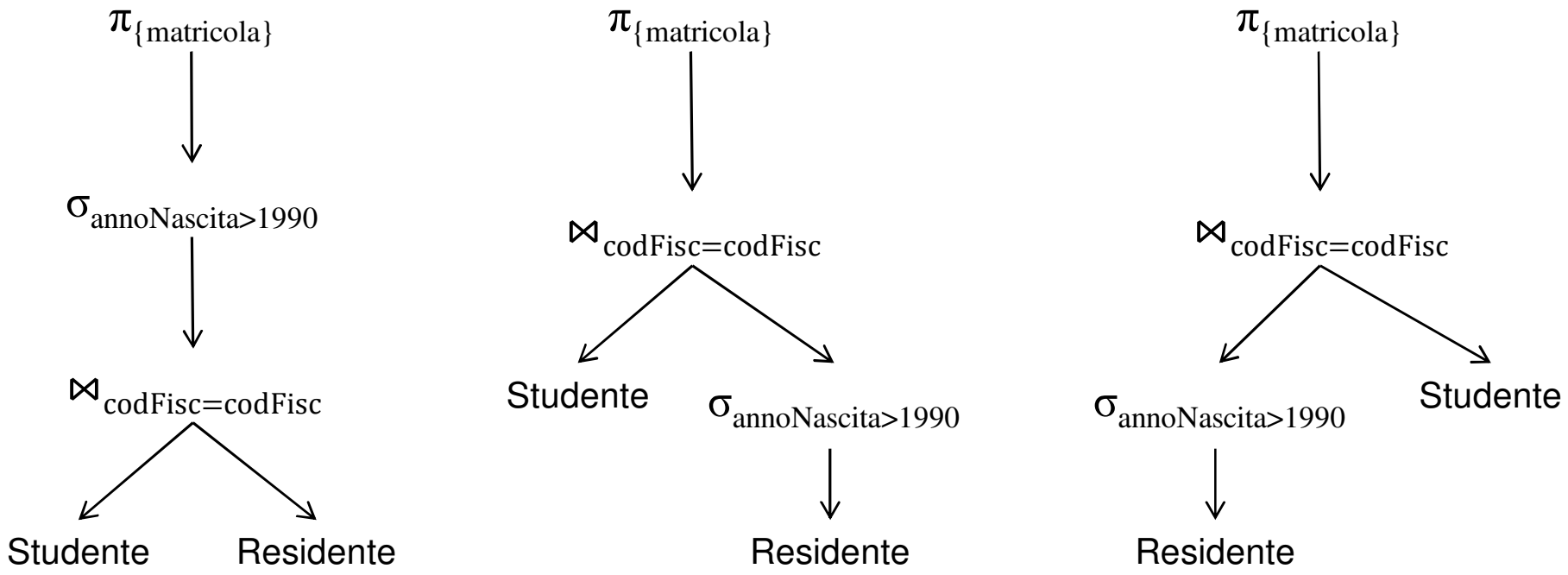
```
SELECT S.matricola
FROM Studente S, Residente R
WHERE S.codFisc = R.codFisc AND R.annoNascita>1990
```

Q può essere espressa in algebra relazionale mediante l'espressione

$$\pi_{\{matricola\}}(\sigma_{annoNascita>1990}(Studente \bowtie_{codFisc=codFisc} Residente))$$

Query plan

- ogni query plan senza specifica delle modalità di esecuzione corrisponde ad una espressione dell'algebra relazionale equivalente a Q
- Ad esempio, da $\pi_{\{matricola\}}(\sigma_{annoNascita>1990}(Studente \bowtie_{codFisc=codFisc} Residente))$, i seguenti alberi:



corrispondono a espressioni equivalenti a quella iniziale

Partizionamento della query SQL

- Il query parser partiziona la query SQL in **blocchi**
- Ogni blocco è una query di tipo SELECT FROM WHERE (con eventualmente clausole HAVING e GROUP BY) senza query annidate
- Pertanto un blocco corrisponde essenzialmente ad una espressione dell'algebra relazionale che usa soltanto gli operatori di selezione (σ), proiezione (π), join ($\bowtie$) e prodotto cartesiano ($\times$)
- Ogni blocco è processato indipendentemente dal successivo query optimizer

Query plan generator

- Il query plan generator riceve in input un blocco, ovvero una espressione dell'algebra relazionale di tipo select-project-join, e genera un insieme di query plan per tale espressione
- Per svolgere questo compito:
  - calcola **trasformazioni equivalenti** dell'espressione
  - utilizza le informazioni (fornite dal catalog manager) sui **metodi d'accesso** delle relazioni (ovvero sulla organizzazione fisica dei file che memorizzano le relazioni)
- Il query plan generator **non** genera tutti i possibili query plan, perché il loro numero è molto grande anche per query relativamente semplici
- La generazione dei query plan è limitata da regole che permettono di generare solo certi tipi di query plan
- Esempio: tutti i join e i prodotti cartesiani devono avere come figlio destro (inner relation) una relazione primitiva

Query plan cost estimator

- È il modulo che assegna un **costo** ad ogni query plan generato dal query plan generator
- Il costo è assegnato sulla base di un modello di costo che considera un insieme di parametri che influenzano il costo effettivo dell'esecuzione della query
- Per calcolare il costo, il query plan cost estimator fa uso delle meta-informazioni fornite dal catalog manager, in particolare:
  - dimensione di ogni relazione (numero di record, numero di pagine)
  - tipo e dimensione (numero di byte) degli attributi
  - dimensione degli indici (numero di pagine)
  - per ogni indice, minimo e massimo valore della chiave di ricerca attualmente presente
  - ecc.

Query plan evaluator

- Il query optimizer seleziona, tra tutti i query plan generati dal query plan generator, quello di costo minimo secondo la valutazione del query plan cost estimator
- Il query plan evaluator esegue tale piano, ovvero esegue l'albero corrispondente al query plan
- Vengono eseguite dapprima le operazioni più interne (sulle foglie dell'albero) e da sinistra verso destra, e si risale man mano fino alla radice (ultima operazione)
- In tale esecuzione viene ovviamente fatto uso dei metodi d'accesso e delle modalità di esecuzione delle operazioni stabiliti dalle annotazioni presenti nel query plan

Equivalenze in algebra relazionale

- Nella generazione dei query plan, il query plan generator fa uso di **equivalenze tra espressioni dell'algebra relazionale** per generare nuovi alberi a partire da quello corrispondente all'espressione iniziale
- Tali equivalenze sono molto importanti ai fini dell'ottimizzazione delle query, ovvero la ricerca di un query plan con costo minimo
- Questa fase del query plan generator è indipendente dai metodi d'accesso delle relazioni e le modalità di esecuzione degli operatori

Equivalenze in algebra relazionale

Esempi:

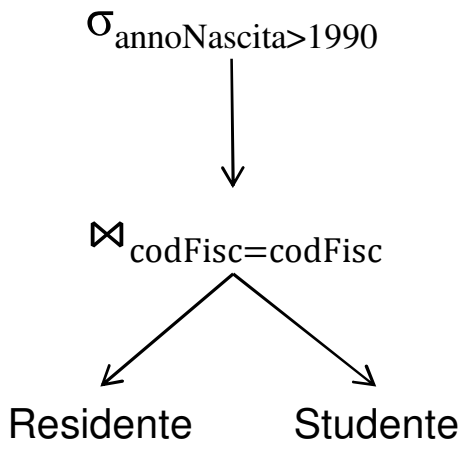
- $R \bowtie_C S \equiv S \bowtie_C R$
- $(R \bowtie_{C1} S) \bowtie_{C2} T \equiv R \bowtie_{C1} (S \bowtie_{C2} T)$
- $R \bowtie_C S \equiv \sigma_C (R \times S)$
- $\sigma_{C1} (\sigma_{C2} (R)) \equiv \sigma_{C2} (\sigma_{C1} (R))$
- $\pi_A(\sigma_C (R)) \equiv \sigma_C(\pi_A(R))$ se la condizione C coinvolge solo attributi di A
- $\pi_A(\sigma_C (R \times S)) \equiv \sigma_C (\pi_A(R) \times \pi_A(S))$ se la condizione C coinvolge solo attributi di A
- $\sigma_{C1 \wedge C2 \wedge C3} (R \bowtie S)) \equiv \sigma_{C3}(\sigma_{C1}(R) \bowtie \sigma_{C2}(S))$ dove
  - $C1$ sono le condizioni che coinvolgono solo attributi di R
  - $C2$ sono le condizioni che coinvolgono solo attributi di S
  - $C3$ sono le condizioni che coinvolgono sia attributi di R che attributi di S

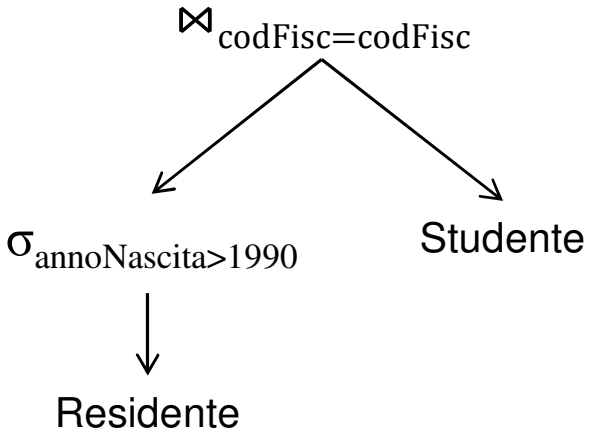
Ottimizzazioni basate su equivalenze

- Euristiche principali:
  - **Push-down selections**: spingere il più possibile in basso (ovvero eseguire il più presto possibile) le selezioni nel query plan
  - **Push-down projections**: spingere il più possibile in basso (ovvero eseguire il più presto possibile) le proiezioni nel query plan
- Entrambe queste euristiche si basano sul fatto che in genere il costo delle operazioni di selezione e proiezione è minore del costo del join
- Ad esempio, se si effettua una selezione ed un join allora è molto probabile che, eseguendo prima la selezione, si riduca almeno una delle due relazioni su cui si effettuerà il join, con notevoli vantaggi in termini di costo

Push-down selection

- Esempio:





- Il secondo query plan è intuitivamente più efficiente del primo, perché il join viene fatto solo dopo aver selezionato solo I record di Residente con anno di nascita maggiore di 1990. Pertanto la relazione intermedia su cui viene eseguito il join potrebbe avere una dimensione molto minore della relazione Residente

Limiti delle euristiche

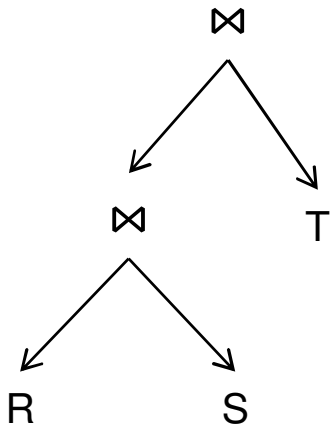
- Non sempre queste euristiche producono soluzioni ottimali
- Vanno infatti valutate le conseguenze delle precedenze tra gli operatori rispetto ai possibili metodi d'accesso delle relazioni e alle possibili modalità di esecuzione dei singoli operatori
- Se nell'esempio precedente avessimo un indice su Residente con chiave di ricerca codFisc, con il primo albero tale indice potrebbe essere usato, mentre con il secondo albero no
- In molte situazioni questo potrebbe implicare che l'esecuzione del primo albero è più efficiente dell'esecuzione del secondo albero

Generazione dei risultati intermedi

- I risultati delle operazioni intermedie (corrispondenti ai nodi intermedi nell'albero del query plan) possono essere gestiti in due modi:
  - **con materializzazione del risultato**: viene generata (nella base di dati) una tabella temporanea che memorizza il risultato dell'operazioni
  - **senza materializzazione del risultato**: il risultato non viene memorizzato in memoria secondaria, ma, man mano che viene generato, viene passato un po' alla volta (pipelining) all'operazione successiva (nodo predecessore nell'albero)
- La materializzazione implica una esecuzione più lenta dell'operazione
- D'altra parte però non tutte le operazioni possono essere eseguite senza materializzare il risultato
- Se ad esempio si vuole che il risultato sia ordinato, è necessario materializzarlo

Generazione dei risultati intermedi

- Esempio:



- Se l'unica modalità di esecuzione a disposizione per l'operazione di join è il sort merge join, il risultato intermedio del join R ⋈ S DEVE essere materializzato, perché l'algoritmo di sort merge join esegue l'ordinamento delle relazioni su cui viene eseguita l'operazione