Basi di dati

II

cap. 1- Organizzazione fisica e gestione delle interrogazioni

INTRODUZIONE ALLA TECNOLOGIA DELLE BASI DI DATI

Tecnologia delle BD: perché studiarla?

- I DBMS offrono i loro servizi in modo "trasparente":
  - per questo abbiamo potuto finora ignorare molti aspetti realizzativi
  - abbiamo considerato il DBMS come una "scatola nera"
- Perché aprirla?
  - capire come funziona può essere utile per un migliore utilizzo
  - alcuni servizi sono offerti separatamente

DataBase Management System — DBMS

Sistema (prodotto software) in grado di gestire collezioni di dati che siano (anche):

- **grandi** (di dimensioni (molto) maggiori della memoria centrale dei sistemi di calcolo utilizzati)
- **persistenti** (con un periodo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano)
- **condivise** (utilizzate da applicazioni diverse)

garantendo **affidabilità** (resistenza a malfunzionamenti hardware e software) e **privatezza** (con una disciplina e un controllo degli accessi). Come ogni prodotto informatico, un DBMS deve essere **efficiente** (utilizzando al meglio le risorse di spazio e tempo del sistema) ed **efficace** (rendendo produttive le attività dei suoi utilizzatori).

Le basi di dati sono grandi e persistenti

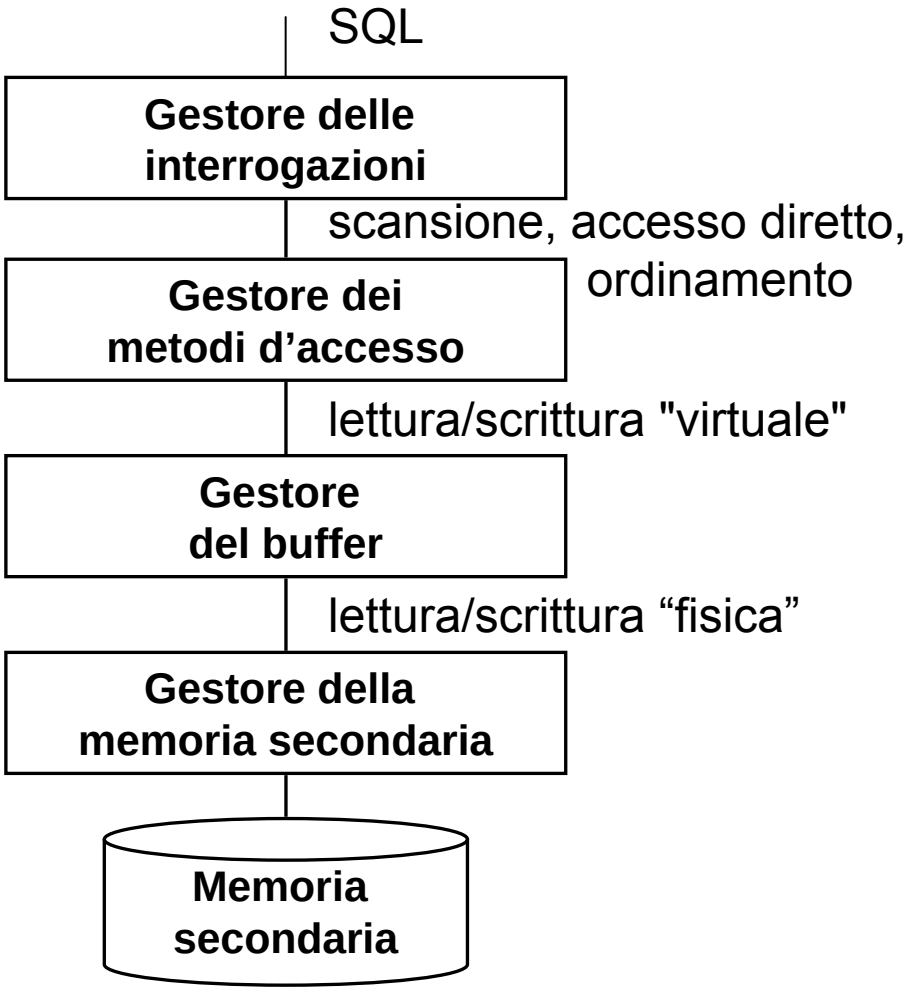
- La persistenza richiede una gestione in memoria secondaria.
- La grandezza richiede che tale gestione sia sofisticata:
  - non possiamo caricare tutto in memoria principale e poi scaricare in memoria secondaria.

Le basi di dati vengono interrogate …

- Gli utenti vedono il modello logico (relazionale)
- I dati sono in memoria secondaria
- Le strutture logiche non sarebbe efficienti in memoria secondaria:
  - servono strutture fisiche opportune
- La memoria secondaria è molto più lenta della memoria principale:
  - serve un'interazione fra memoria principale e secondaria che limiti il più possibile gli accessi alla secondaria

- Esempio: una interrogazione con un join

Gestore degli accessi e delle interrogazioni



Le basi di dati sono affidabili

- Le basi di dati sono una risorsa per chi le possiede, e debbono essere conservate anche in presenza di malfunzionamenti

- Esempio:
  - un trasferimento di fondi da un conto corrente bancario ad un altro, con guasto del sistema a metà

- Le transazioni debbono essere
  - atomiche (o tutto o niente)
  - definitive: dopo la conclusione, non si dimenticano

Le basi di dati vengono aggiornate …

- L'affidabilità è impegnativa per via degli aggiornamenti frequenti e della necessità di gestire il buffer

Le basi di dati sono condivise

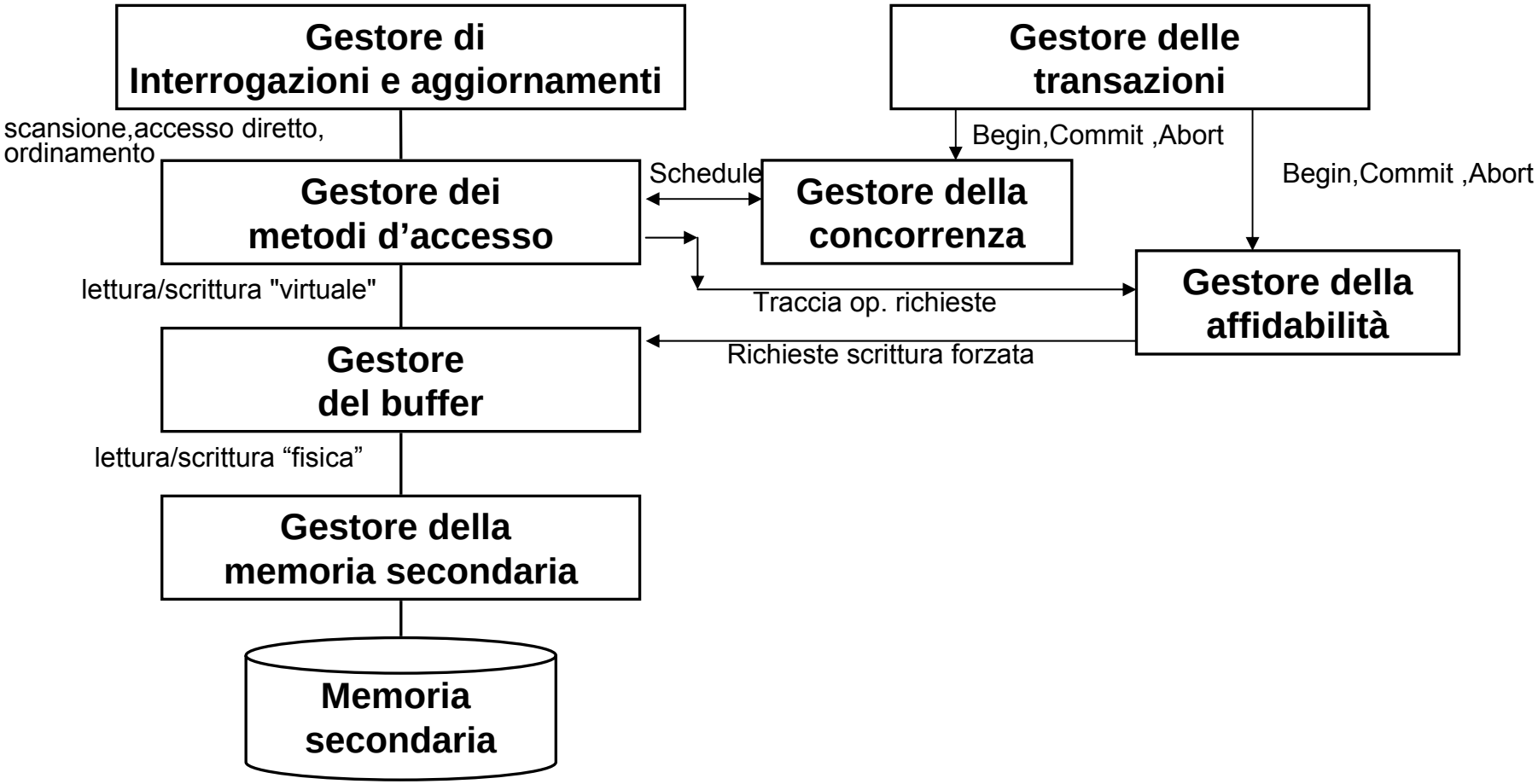
- Una base di dati è una risorsa **integrata**, **condivisa** fra le varie applicazioni
- conseguenze
  - Attività diverse su dati in parte condivisi:
    - meccanismi di **autorizzazione**
  - Attività multi - utente su dati condivisi:
    - controllo della **concorrenza**

Aggiornamenti su basi di dati condivise …

- Esempi:
  - due prelevamenti (quasi) contemporanei sullo stesso conto corrente
  - due prenotazioni (quasi) contemporanee sullo posto
- Intuitivamente, le transazioni sono corrette se **seriali** (prima una e poi l'altra)
- Ma in molti sistemi reali l'efficienza sarebbe penalizzata troppo se le transazioni fossero seriali:
  - il **controllo della concorrenza** permette un ragionevole compromesso

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni

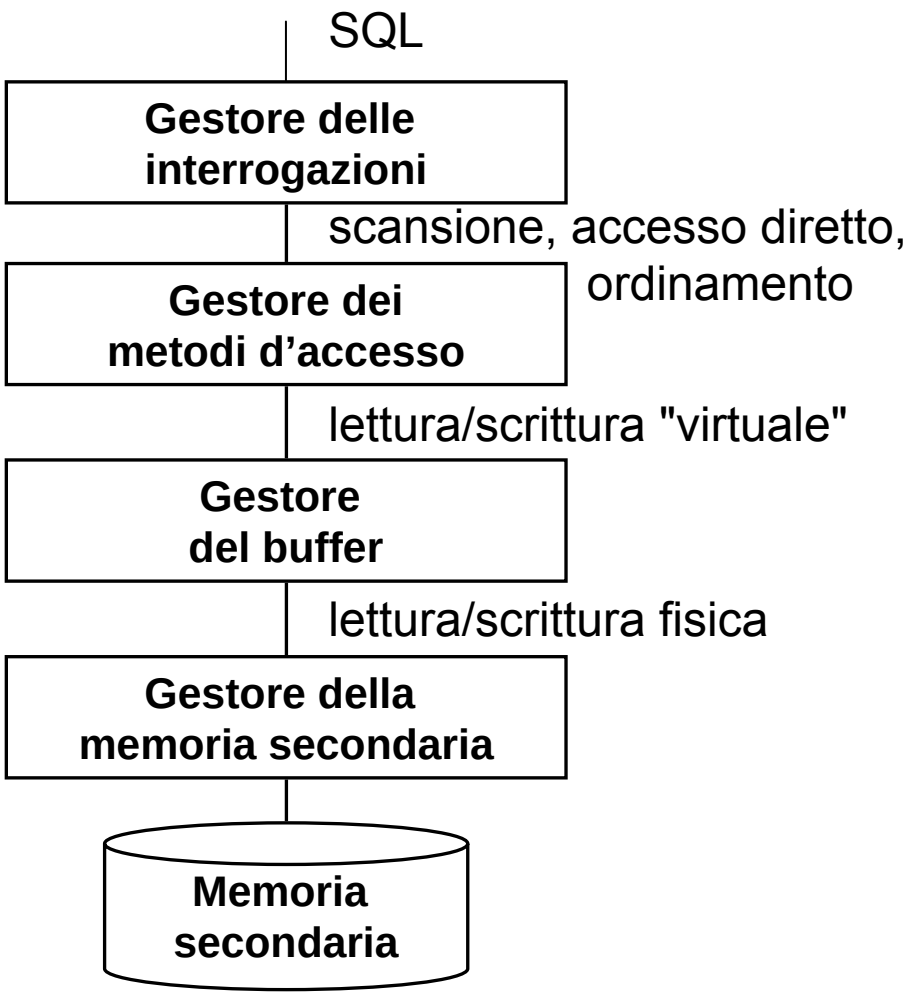


Tecnologia delle basi di dati, argomenti

- Gestione della memoria secondaria e del buffer
- Organizzazione fisica dei dati
- Gestione ("ottimizzazione") delle interrogazioni
- Controllo della affidabilità
- Controllo della concorrenza

ORGANIZZAZIONE FISICA DEI DATI

Gestore degli accessi e delle interrogazioni



MEMORIA PRINCIPALE E SECONDARIA

- I programmi possono fare riferimento solo a dati in memoria principale
- Le basi di dati debbono essere (sostanzialmente) in memoria secondaria per due motivi:
  - dimensioni
  - persistenza
- I dati in memoria secondaria possono essere utilizzati solo se prima trasferiti in memoria principale (questo spiega i termini "principale" e "secondaria")

Memoria principale e secondaria, 2

- I dispositivi di memoria secondaria sono organizzati in **blocchi** di lunghezza (di solito) **fissa** (ordine di grandezza: alcuni KB)
- Le uniche operazioni sui dispositivi solo la lettura e la scrittura di di una **pagina**, cioè dei dati di un blocco (cioè di una stringa di byte);
- per comodità consideriamo **blocco** e **pagina** sinonimi

Memoria principale e secondaria, 3

- Accesso a memoria secondaria:
  - tempo di posizionamento della testina (10-50ms)
  - tempo di latenza (5-10ms)
  - tempo di trasferimento (1-2ms)

  in media non meno di 16ms
- Il costo di un accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria centrale
- Perciò, nelle applicazioni "I/O bound" (cioè con molti accessi a memoria secondaria e relativamente poche operazioni) il costo dipende esclusivamente dal numero di accessi a memoria secondaria
- Inoltre, accessi a blocchi “vicini” costano meno (contiguità)

GESTIONE DEL BUFFER

- **Buffer**:
  - area di memoria centrale, gestita dal DBMS (preallocata) e condivisa fra le transazioni
  - organizzato in **pagine** di dimensioni pari o multiple di quelle dei blocchi di memoria secondaria (1KB-100KB)
  - è importantissimo per via della grande differenza di tempo di accesso fra memoria centrale e memoria secondaria

Scopo della gestione del buffer

- Ridurre il numero di accessi alla memoria secondaria
  - In caso di lettura, se la pagina è già presente nel buffer, non è necessario accedere alla memoria secondaria
  - In caso di scrittura, il gestore del buffer può decidere di differire la scrittura fisica (ammesso che ciò sia compatibile con la gestione dell'affidabilità – vedremo più avanti)
- La gestione dei buffer e la differenza di costi fra memoria principale e secondaria possono suggerire algoritmi innovativi:
  - Esempio:
    - File di 10.000.000 di record di 100 byte ciascuno (1GB)
    - Blocchi di 4KB
    - Buffer disponibile di 20M

    Come possiamo fare l'ordinamento?
      - Merge-sort "a più vie"

Dati gestiti dal buffer manager

- Il buffer
- Un direttorio che per ogni pagina mantiene (ad esempio)
  - il nome del file fisico e il numero del blocco
  - due variabili di stato:
    - un **contatore** che indica quanti programmi utilizzano la pagina
    - un **bit** che indica se la pagina è "sporca", cioè se è stata modificata

Funzioni del buffer manager

- Intuitivamente:
  - riceve richieste di lettura e scrittura (di pagine)
  - le esegue accedendo alla memoria secondaria solo quando indispensabile e utilizzando invece il buffer quando possibile
  - esegue le primitive
    - ***fix***, ***unfix***, ***setDirty***, ***force***.
- Le politiche sono simili a quelle relative alla gestione della memoria da parte dei sistemi operativi:
  - "località dei dati": è alta la probabilità di dover riutilizzare i dati attualmente in uso
  - "legge 80-20" l'80% delle operazioni utilizza sempre lo stesso 20% dei dati

Interfaccia offerta dal buffer manager

- ***fix****:* richiesta di una pagina; richiede una lettura 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
- ***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 (su richiesta del gestore dell'affidabilità, non del gestore degli accessi)

Esecuzione della fix

- Cerca la pagina nel buffer;
  - se c'è, restituisce l'indirizzo;
  - altrimenti, cerca una pagina libera nel buffer (contatore a zero);
    - se la trova, se “sporca” effettua una scrittura su memoria secondaria (flush) ed in ogni caso legge la pagina di interesse e restituisce l'indirizzo
    - altrimenti, due alternative
      - “**steal**”: selezione di una "vittima", pagina occupata del buffer; I dati della vittima sono scritti in memoria secondaria(flush); viene letta la pagina di interesse dalla memoria secondaria e si restituisce l'indirizzo
      - “**no-steal**”: l'operazione viene posta in attesa

Commenti

- Il buffer manager richiede scritture in due contesti diversi:
  - in modo **sincrono** quando è richiesto esplicitamente con una force
  - in modo **asincrono** (flush) quando lo ritiene opportuno (o necessario); in particolare, può decidere di anticipare o posticipare scritture per coordinarle e/o sfruttare la disponibilità dei dispositivi

ORGANIZZAZIONE DEI FILE

- I DBMS utilizzano le funzionalità del File System (*) in misura limitata :
  - *per creare ed eliminare file*
  - *per leggere e scrivere singoli blocchi o sequenze di blocchi contigui*.

- **L'organizzazione dei file**, sia in termini di distribuzione dei record nei blocchi sia relativamente alla struttura all'interno dei singoli blocchi **è gestita direttamente dal DBMS**.

(*) componente del Sistema Operativo che gestisce la memoria secondaria

Organizzazione dei file

- Il DBMS crea **file** di grandi dimensioni che utilizza per memorizzare diverse relazioni (al limite, l'intero database)
- Talvolta, vengono creati file *in tempi successivi (in fase di aggiornamento)*:
  - è possibile quindi che un file contenga i dati di più relazioni e che le varie tuple di una relazione siano in file diversi.
- Il DBMS gestisce i **blocchi dei file** allocati come se fossero un unico grande spazio di memoria secondaria e costruisce, in tale spazio, le strutture fisiche con cui implementa le relazioni.
  - spesso un blocco è dedicato a tuple di un'unica relazione
  - talora sono previste tuple di più relazioni nello stesso blocco.

Blocchi e record

- I blocchi (componenti "fisici" di un file) e i record (componenti "logici") hanno dimensioni in generale diverse:
  - la dimensione del blocco dipende dal file system
  - la dimensione del record (semplificando un po') dipende dalle esigenze dell'applicazione, e può anche variare nell'ambito di un file

Fattore di blocco

- numero di record in un blocco, fattore di blocco F:
  - L_R: dimensione di un record (per semplicità costante nel file: "record a lunghezza fissa")
  - L_B: dimensione di un blocco
  - se $L_B > L_R$, possiamo avere più record in un blocco:

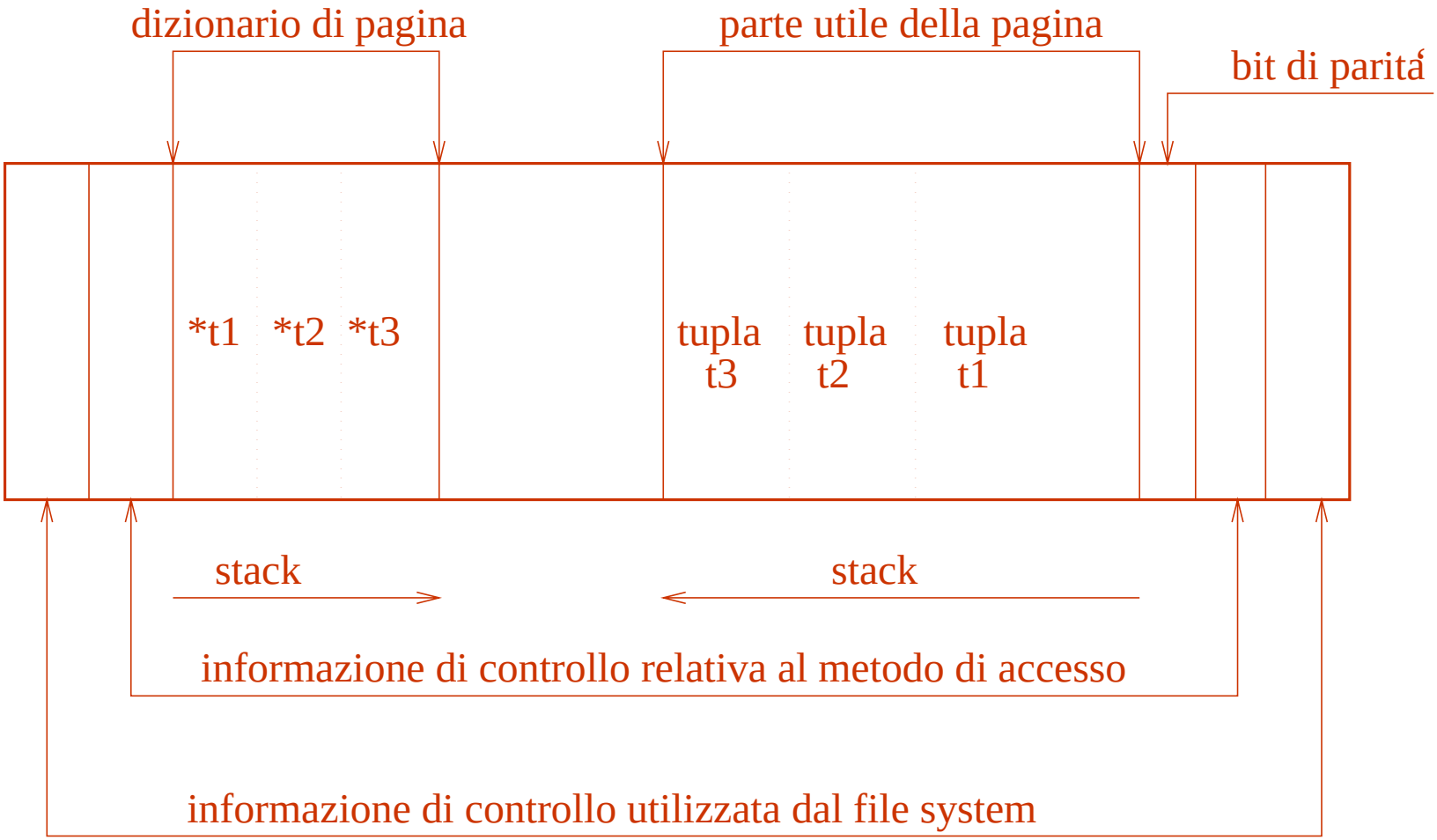
$$F = \lfloor L_B / L_R \rfloor$$

- lo spazio residuo può essere
  - utilizzato (record "spanned" o impaccati)
  - non utilizzato ("unspanned")

Organizzazione delle tuple nelle pagine

- Ci sono varie alternative, anche legate ai metodi di accesso; vediamo una possibilità nella figura a pagina successiva.
- Inoltre:
  - se la lunghezza delle tuple è fissa, la struttura può essere semplificata
  - alcuni sistemi possono spezzare le tuple su più pagine (necessario per tuple grandi)

Organizzazione delle tuple nelle pagine



Primitive del gestore delle pagine

- **Inserimento e aggiornamento**
  - con riorganizzazione della pagina
  - con allocazione di nuovi blocchi (*)
- **Cancellazione**
  - in generale con marcatura (ad es. “tupla non valida”)
- **Accesso ad una particolare tupla**
  - per via associativa o in base all’ “offset tupla” nel direttorio
- **Accesso ad un campo di una particolare tupla**
  - Identificato da indirizzo tupla + offset e lunghezza campo

(*) da parte del file system

Strutture primarie per l'organizzazione di file

- Per struttura primaria di un file s'intende il modo secondo il quale sono disposte le tuple nell'ambito del file.
- Tre sono le categorie principali.
  - sequenziali
  - ad accesso calcolato
  - ad albero
- Si noti che le strutture ad albero sono utilizzate soprattutto per implementare le cosiddette "**strutture secondarie**" al fine di velocizzare l'accesso alle informazioni contenute nella struttura primaria cioè nel file delle tuple.

Strutture sequenziali

- Esiste un ordinamento fra le tuple, che può essere rilevante ai fini della gestione
  - **seriale***:* ordinamento fisico ma non logico
  - **array***:* posizioni individuate attraverso indici di array
  - **ordinata**: l'ordinamento delle tuple coerente con quello di un campo detto chiave

Struttura seriale

- Chiamata anche:
  - “entry sequenced"
  - file heap – cioè con record “ammucchiati” disordinatamente.
- È molto diffusa nelle basi di dati relazionali, associata a *indici secondari*
- Gli inserimenti vengono effettuati
  - in coda
  - al posto di record cancellati

Strutture ordinate su campo chiave

- Permettono ricerche binarie, ma solo fino ad un certo punto (ad esempio, come troviamo la "metà del file"?
- Nelle basi di dati relazionali si utilizzano quasi solo in combinazione con indici (*file **ISAM*** o *file ordinati con indice primario*)
- Inserimento nuove tuple:
  - prevedendo posizioni libere al caricamento iniziale al fine di consentire riordini locali;
  - a seguito di saturazioni, inserimento di **nuovi blocchi** legati a puntatore;
  - integrazione del file sequenziale con un **file di overflow** i cui blocchi sono legati da una catena a puntatori.

File hash

- Permettono un accesso diretto molto efficiente (da alcuni punti di vista)
- La tecnica si basa su quella utilizzata per le tavole hash in memoria centrale

Tavola hash (in memoria centrale)

- **Obiettivo**: accesso (quasi)diretto ad un insieme di record sulla base del valore di un campo (detto **chiave**, che per semplicità supponiamo identificante, ma non è necessario)
  - *se i possibili valori della chiave sono in numero paragonabile al numero di record* (e corrispondono ad un "tipo indice") allora usiamo un **array**; ad esempio: università con 1000 studenti e numeri di matricola compresi fra 1 e 1000 o poco più e file con tutti gli studenti
  - *se i possibili valori della chiave sono molti di più di quelli effettivamente utilizzati*, non possiamo usare l'array (spreco);
    - ad esempio: 40 studenti e numero di matricola di 6 cifre (un milione di possibili chiavi)

Tavola hash, 2

- Volendo continuare ad usare qualcosa di simile ad un array, ma senza sprecare spazio, possiamo pensare di trasformare i valori della chiave in possibili indici di un array,cioè **h=f(key), h** è detta **funzione hash :**
  - associa ad ogni valore della chiave key un "indirizzo“ h, in uno spazio di dimensione paragonabile (*leggermente superiore*) rispetto a quello strettamente necessario
  - poiché il numero di possibili chiavi è molto maggiore del numero di possibili indirizzi ("lo spazio delle chiavi è più grande dello spazio degli indirizzi"), la funzione non può essere iniettiva e quindi esiste la possibilità di collisioni (chiavi diverse che corrispondono allo stesso indirizzo)
  - le buone funzioni hash distribuiscono in modo causale e uniforme, riducendo le probabilità di collisione (che si riduce aumentando lo spazio ridondante)

Un esempio

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

Sequenza di arrivo

M	M mod 50	M	M mod 50
060600	0	200268	18
066301	1	205619	19
205751	1	210522	22
205802	2	205724	24
200902	2	205977	27
116202	2	205478	28
200604	4	200430	30
066005	5	210533	33
116455	5	205887	37
200205	5	200138	38
201159	9	102338	38
205610	10	102690	40
201260	10	115541	41
102360	10	206092	42
205460	10	205693	43
205912	12	205845	45
205762	12	200296	46
200464	14	205796	46
205617	17	200498	48
205667	17	206049	49

Tavola hash, gestione collisioni

- Varie tecniche:
  - posizioni successive disponibili (indirizzamento aperto)
  - tabella di overflow (gestita in forma collegata)
  - funzioni hash "alternative"
- Nota:
  - le collisioni ci sono (quasi) sempre;
  - le collisioni multiple hanno probabilità che decresce al crescere della molteplicità di collisione;
  - la molteplicità media delle collisioni è molto bassa.

File hash (su memoria secondaria)

- L'idea è la stessa della tavola hash, ma si basa sull‘ **organizzazione in blocchi (bucket).**
- In questo modo si "ammortizzano" le probabilità di collisione

Un esempio

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

  numero medio di accessi: 1,425
- file hash con **5 blocchi con 10 posizioni** ciascuno (fattore di blocco 10):
  - due soli overflow!

  numero medio di accessi: 1,05

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

Un file hash

Entra M -> calcolo M mod 5

0	1	2	3	4
60600	66301	205802	200268	200604
66005	205751	200902	205478	201159
116455	115541	116202	210533	200464
200205	200296	205912	200138	205619
205610	205796	205762	102338	205724
201260		205617	205693	206049
102360		205667	200498	
205460		210522		
200430		205977		
102690		205887		
205845		206092		

File hash, osservazioni

- È l'organizzazione più efficiente per l'accesso diretto basato su valori della chiave con condizioni di uguaglianza (accesso puntuale): costo medio di poco superiore all'unità (il caso peggiore è molto costoso ma talmente improbabile da poter essere ignorato)
- Le collisioni (overflow) sono di solito gestite con blocchi collegati
- Non è efficiente:
  - per ricerche basate su intervalli
  - per ricerche basate su attributi diversi da quello chiave
- I file hash "degenerano" **se si riduce lo spazio in eccesso**.
- funzionano solo con file la cui dimensione **non varia molto** nel tempo

Strutture ad albero

- Consentono l'accesso in base al valore di uno o più campi per:
  - accessi puntuali
  - accessi corrispondenti a campi di valori
- Possono essere utilizzate per realizzare:
  - strutture **primarie** cioè **contenenti i dati** (vedi *Isam*)
  - strutture **secondarie** cioè strutture dette **indici** che favoriscono l'accesso ai dati **senza contenere i dati stessi**.
- Le strutture primarie ad albero sono file costituiti da un insieme di nodi ordinati sul **campo** su cui è basata la memorizzazione: ciascun nodo contiene il valore del suddetto campo ed **i dati associati**.

INDICI DI FILE

- Indice:
  - struttura ausiliaria per l'accesso (efficiente) ai record di un file sulla base dei valori di un campo (o di una "concatenazione di campi") detto generalmente **chiave** (sarebbe meglio, **pseudochiave**, perché il campo – cioè l'attributo – non è necessariamente identificante);
- 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
- **Un indice I di un file f** è un altro file, con record a due campi: chiave e indirizzo (dei record di f o dei relativi blocchi), **ordinato secondo i valori della chiave**

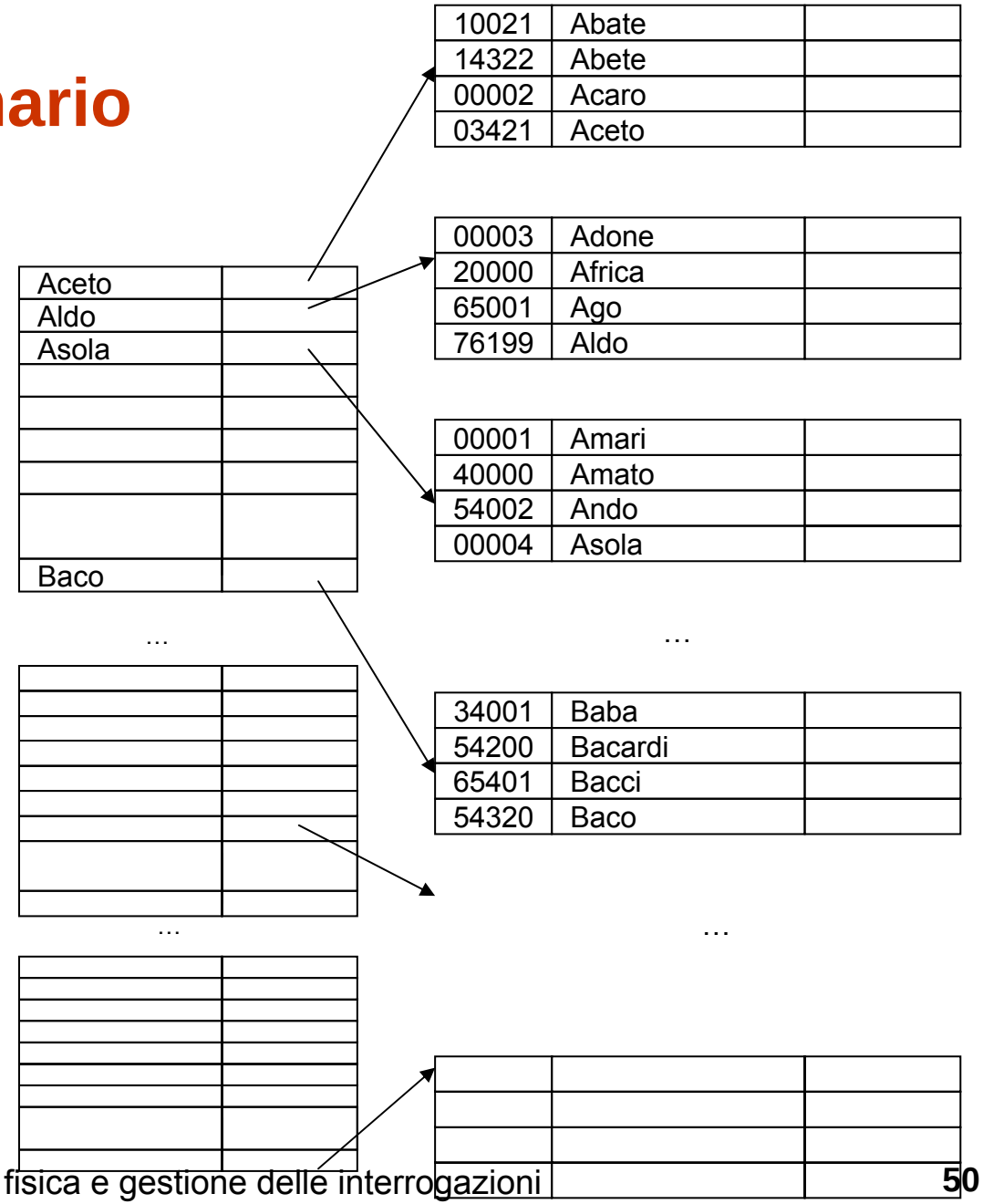
Tipi di indice

- **indice primario**:
  - *su un campo sul cui ordinamento è basata la memorizzazione* (detti anche indici di cluster)
    - talvolta si chiamano **primari** gli indici su una chiave identificante e di **cluster** quelli su una chiave non identificante
- **indice secondario**
  - *su un campo con ordinamento diverso da quello di memorizzazione – se ve ne sono.*
- **indice denso**:
  - contiene un record per ciascun record del file (primario o secondario)
- **indice sparso**:
  - contiene un numero di record inferiore rispetto a quelli del file (necessariamente primario)

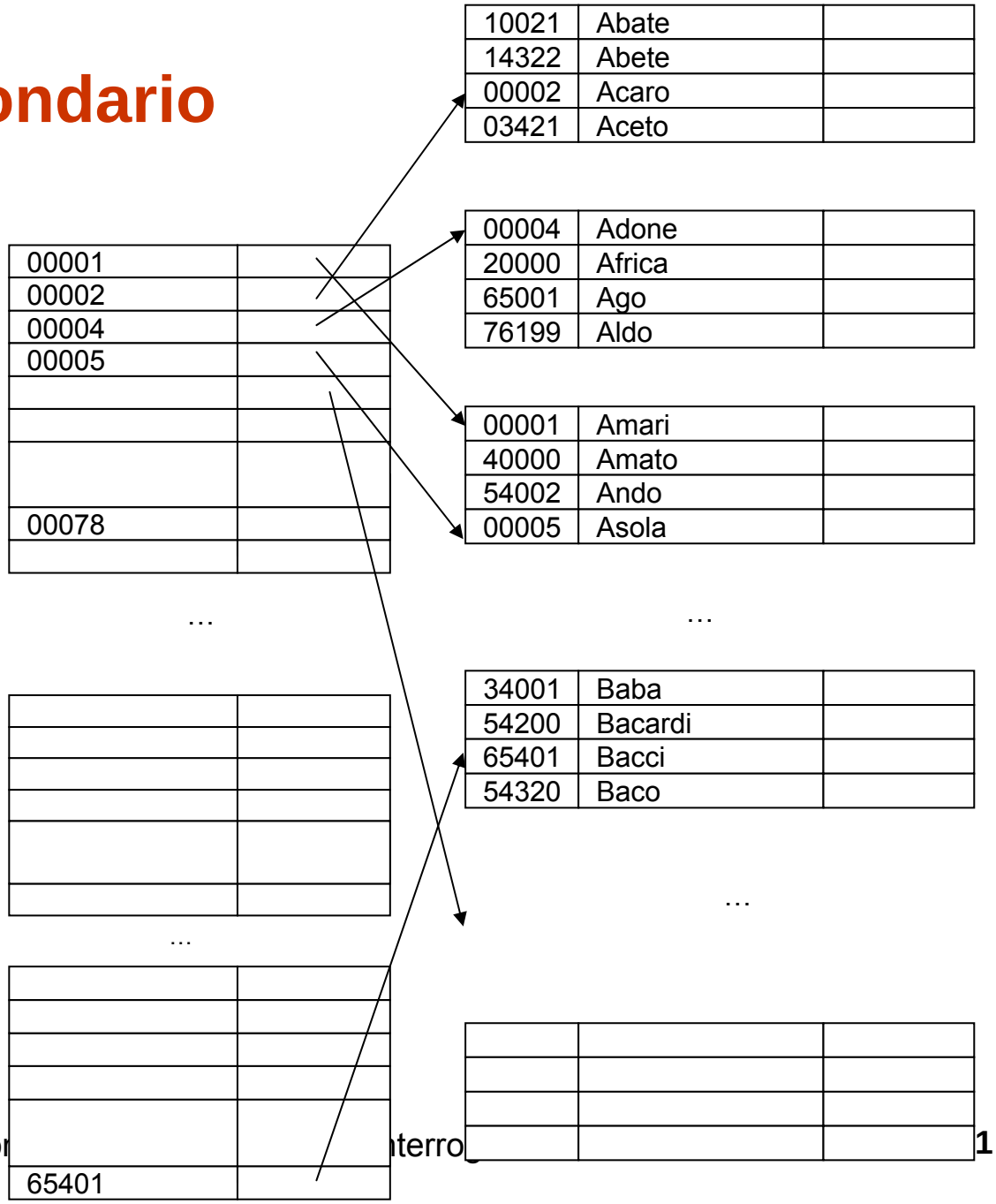
Tipi di indice, commenti

- Pertanto un indice primario **può** essere sparso, uno secondario **deve** essere denso
  - Esempio, sempre rispetto ad un libro:
    - indice generale
    - indice analitico
- Ogni file *può avere **al più un** indice primario* e ***un numero qualunque** di indici secondari* (su campi diversi).
  - Esempio:
    - una guida turistica può avere l'indice dei luoghi e quello degli artisti
- **Un file hash non può avere un indice primario**

Indice primario



Indice secondario



Dimensioni dell'indice

- NR_F numero di record nel file
- L_B lunghezza del blocco
- L_R lunghezza del record
- L_K lunghezza del campo chiave
- L_P lunghezza dell' indirizzo (ai blocchi)
- F fattore di blocco del file
- F_i fattore di blocco dell'indice

N. blocchi per il file (circa): $NB_F = NR_F / \lfloor L_B / L_R \rfloor = NR_F / F$

- Numero di record del file diviso il fattore di blocco del file

N. blocchi per un indice denso: $NB_{ID} = NR_F / \lfloor L_B / (L_K + L_P) \rfloor = NR_F / F_i$

- Numero di record del file diviso il fattore di blocco dell'indice

N. blocchi per un indice sparso: $NB_{IS} = NB_F / \lfloor L_B / (L_K + L_P) \rfloor = NB_F / F_i$

- Numero di blocchi del file diviso il fattore di blocco dell'indice

Caratteristiche degli indici

- Accesso diretto - sulla (pseudo)chiave - efficiente, sia puntuale sia per intervalli
- Scansione sequenziale ordinata efficiente:
  - tutti gli indici (in particolare quelli secondari) forniscono un **ordinamento logico** sui record del file; con numero di accessi pari al numero di record del file - a parte qualche beneficio dovuto alla bufferizzazione.
- Modifiche della (pseudo)chiave, inserimenti, eliminazioni inefficienti (come nei file ordinati)
  - tecniche per alleviare i problemi:
    - file o blocchi di overflow
    - marcatura per le eliminazioni
    - riempimento parziale
    - blocchi collegati (non contigui)
    - riorganizzazioni periodiche

Indici secondari, due osservazioni

- Si possono usare **puntatori ai blocchi oppure puntatori ai record:**
  - i puntatori ai **blocchi** sono più compatti
  - i puntatori ai **record** permettono di semplificare alcune operazioni - ad. es. ricerca dei record che soddisfano una relazione su attributi "indicizzati" - che possono essere *effettuate solo sull'indice* senza pertanto accedere al file.

Indici multilivello

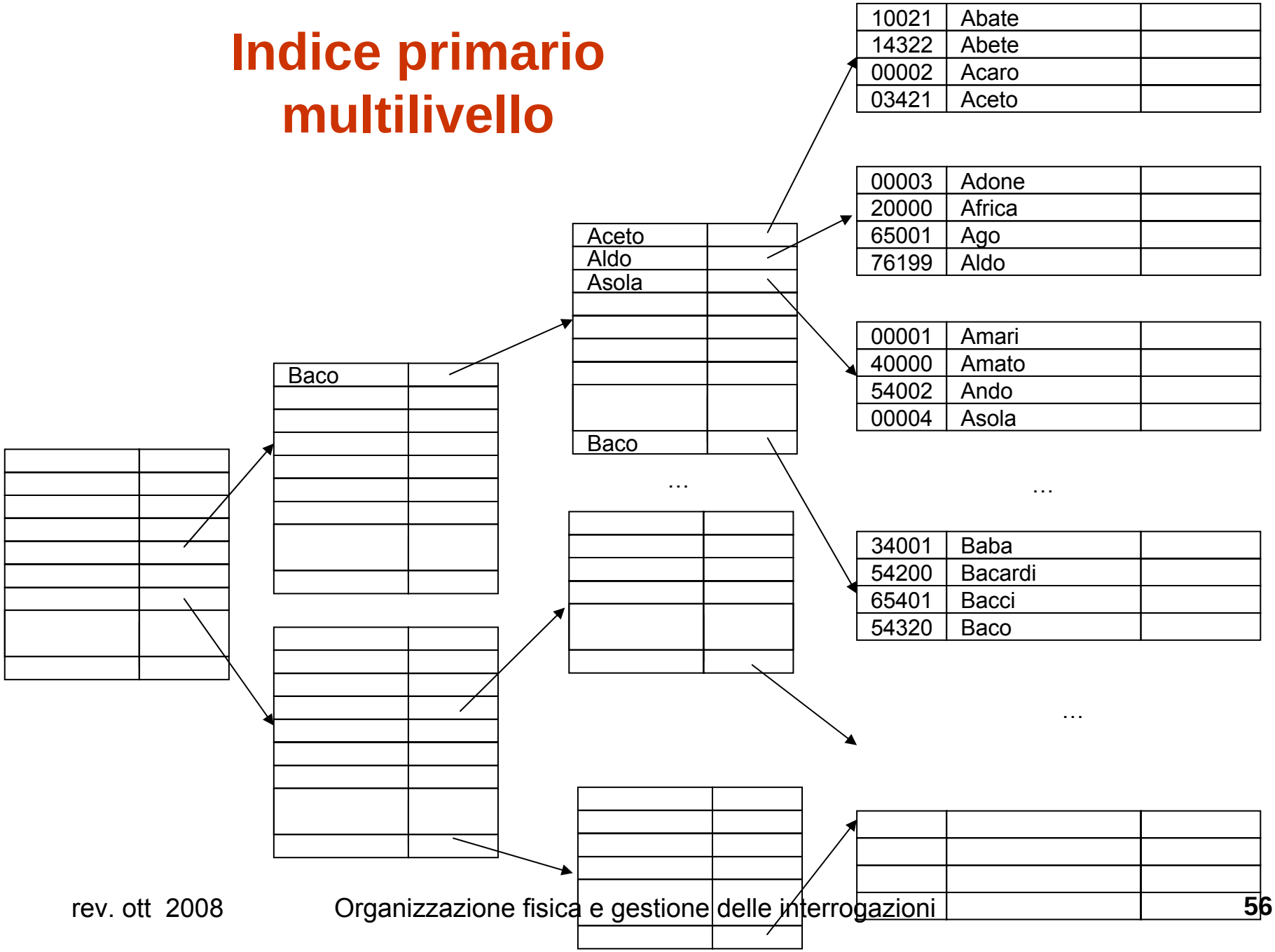
- Gli indici sono file essi stessi e quindi ha senso costruire indici sugli indici, per evitare di fare ricerche fra blocchi diversi
- Possono esistere più livelli fino ad avere il livello più alto con un solo blocco; i livelli sono di solito abbastanza pochi, perché
  - l'indice è ordinato, quindi **l'indice sull'indice è sparso**
  - i record dell'indice sono piccoli

- N_j numero di blocchi al livello j dell'indice (circa):

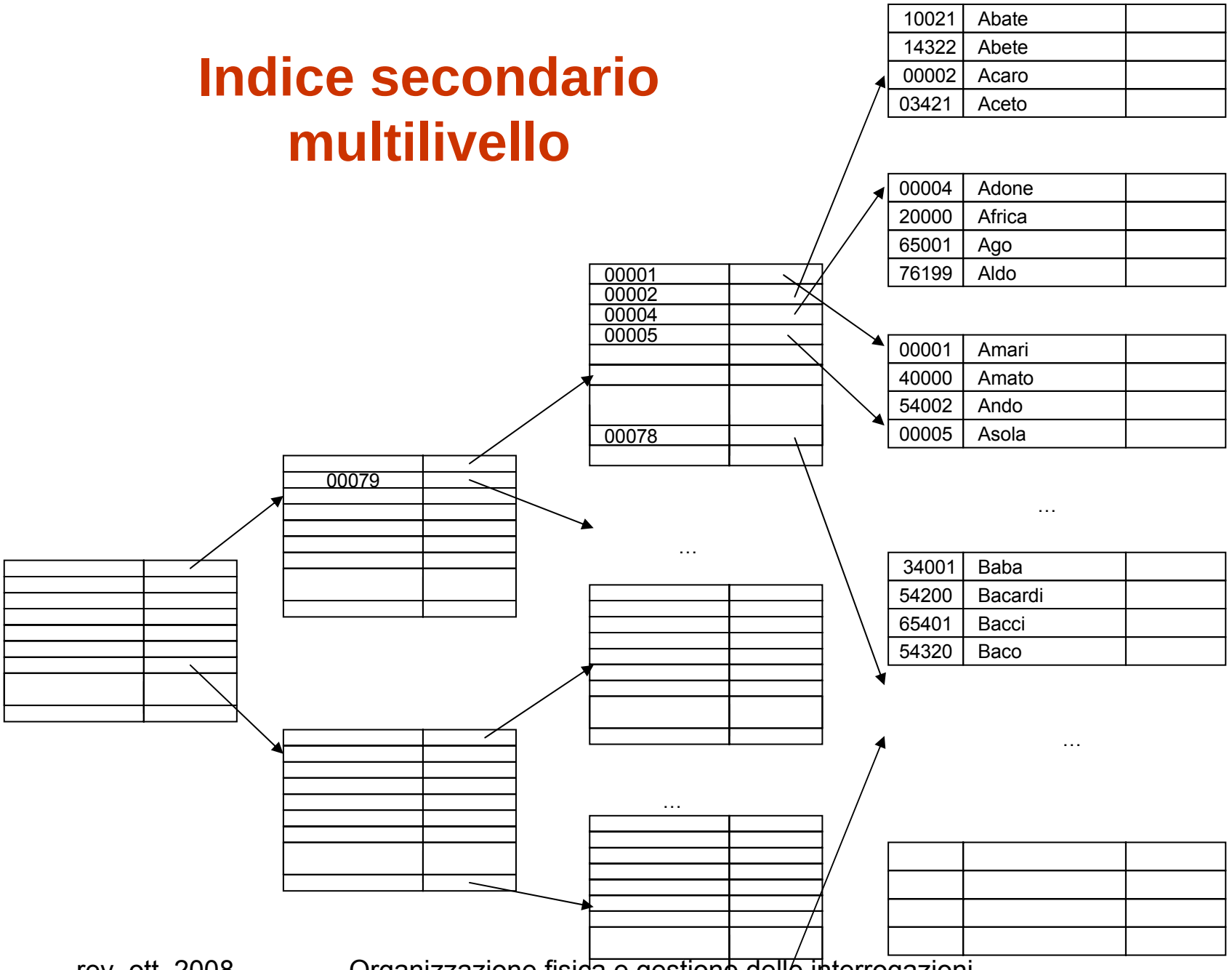
$$NB_j = NB_{j-1} / \lfloor L_B/(L_K+L_P) \rfloor = NB_{j-1} / F_i$$

Il numero di blocchi al livello j è pari al numero di blocchi al livello j-1 (indice sparso) diviso il numero di record di indice per blocco

Indice primario multilivello



Indice secondario multilivello

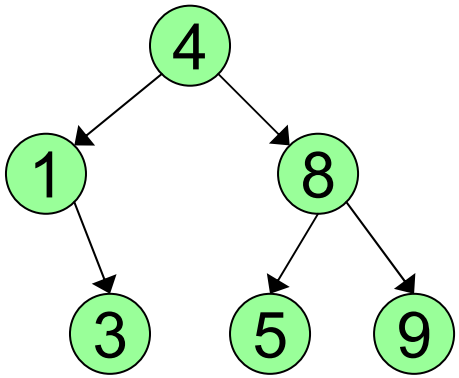


Indici, problemi

- Tutte le strutture di indice viste finora sono basate su strutture ordinate e quindi sono poco flessibili in presenza di elevata dinamicità.
- Gli indici utilizzati dai DBMS sono più sofisticati:
  - indici dinamici multilivello: B-tree (intuitivamente: alberi di ricerca bilanciati)
    - Arriviamo ai B-tree per gradi
      - Alberi binari di ricerca
      - Alberi n-ari di ricerca
      - Alberi n-ari 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)

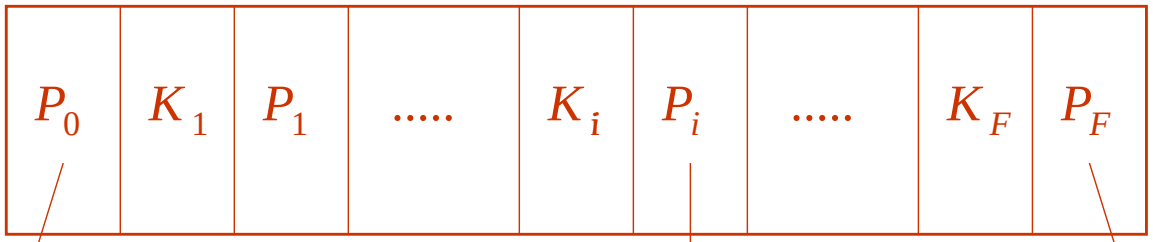


Albero di ricerca di ordine B

- Ogni nodo ha al massimo F=B-1 valori ordinati di chiave e B puntatori . Ogni chiave K_i è seguita da un puntatore p_i ; K_1 è preceduto da un puntatore $p_{0:}$
  - p_0 indirizza al sottalbero con chiavi $K<K_1$
  - p_F indirizza al sottalbero con chiavi $K\geq K_F$
  - p_i indirizza al sottalbero con chiavi € [K_i, K_{i+1}**)**
- Ogni ricerca o modifica dell'albero comporta la visita di un cammino radice foglia.
- In strutture fisiche, un nodo può corrispondere ad un blocco.
- 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

$K_1 < K_2 < ... < K_F$



sotto-albero che contiene le chiavi $K < K_1$

sotto-albero che contiene le chiavi $K_i \leqslant K < K_{i+1}$

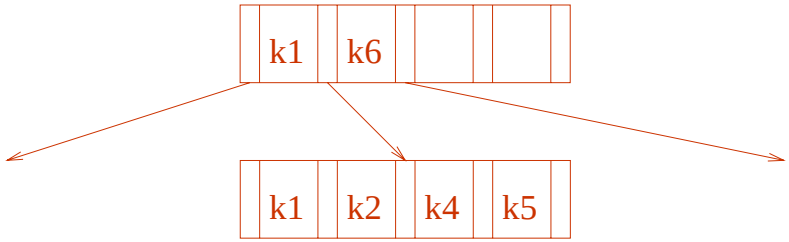
sotto-albero che contiene le chiavi $K \geqslant K_F$

Split e merge

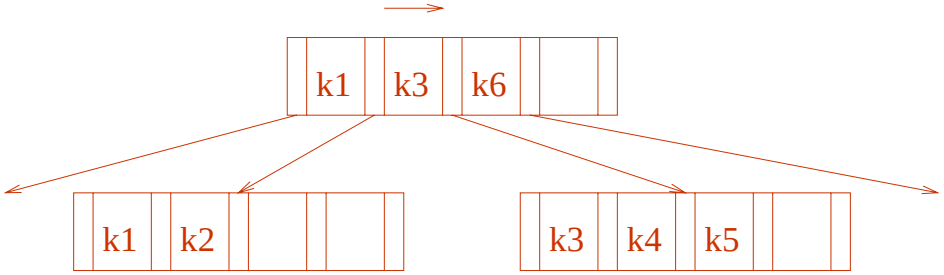
- **Inserimenti ed eliminazioni sono precedute da una ricerca fino ad una foglia**
- Per gli inserimenti, se c'è posto nella foglia, ok, altrimenti il nodo va suddiviso, con necessità di un puntatore in più per il nodo genitore; se non c'è posto, si sale ancora, eventualmente fino alla radice. Il riempimento rimane sempre superiore al 50%
- Dualmente, le eliminazioni possono portare a riduzioni di nodi
- Modifiche del campo chiave vanno trattate come eliminazioni seguite da inserimenti

Split e merge

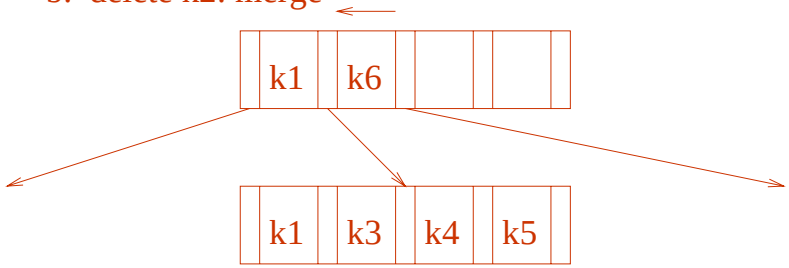
situazione iniziale



a. insert k3: split



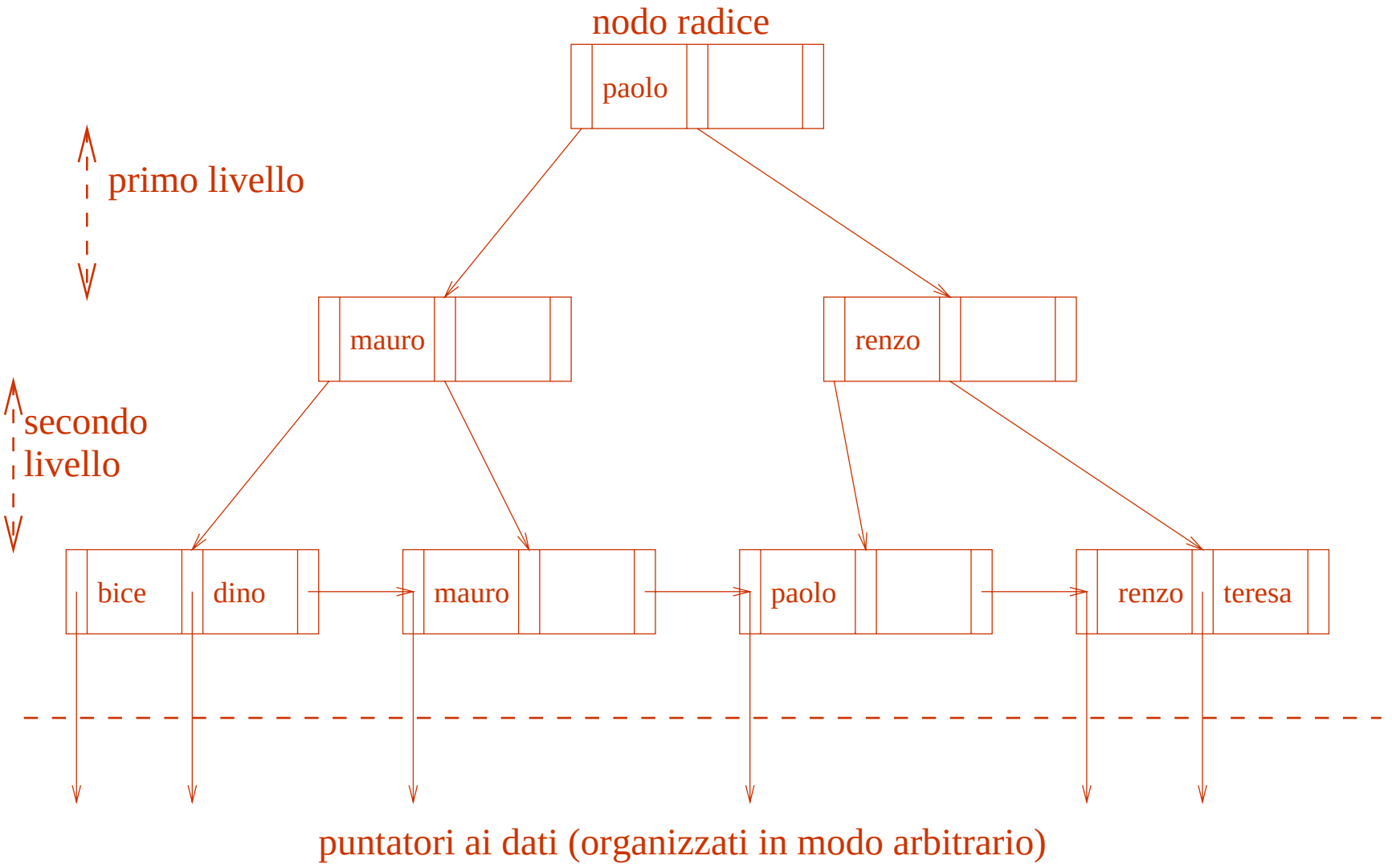
b. delete k2: merge



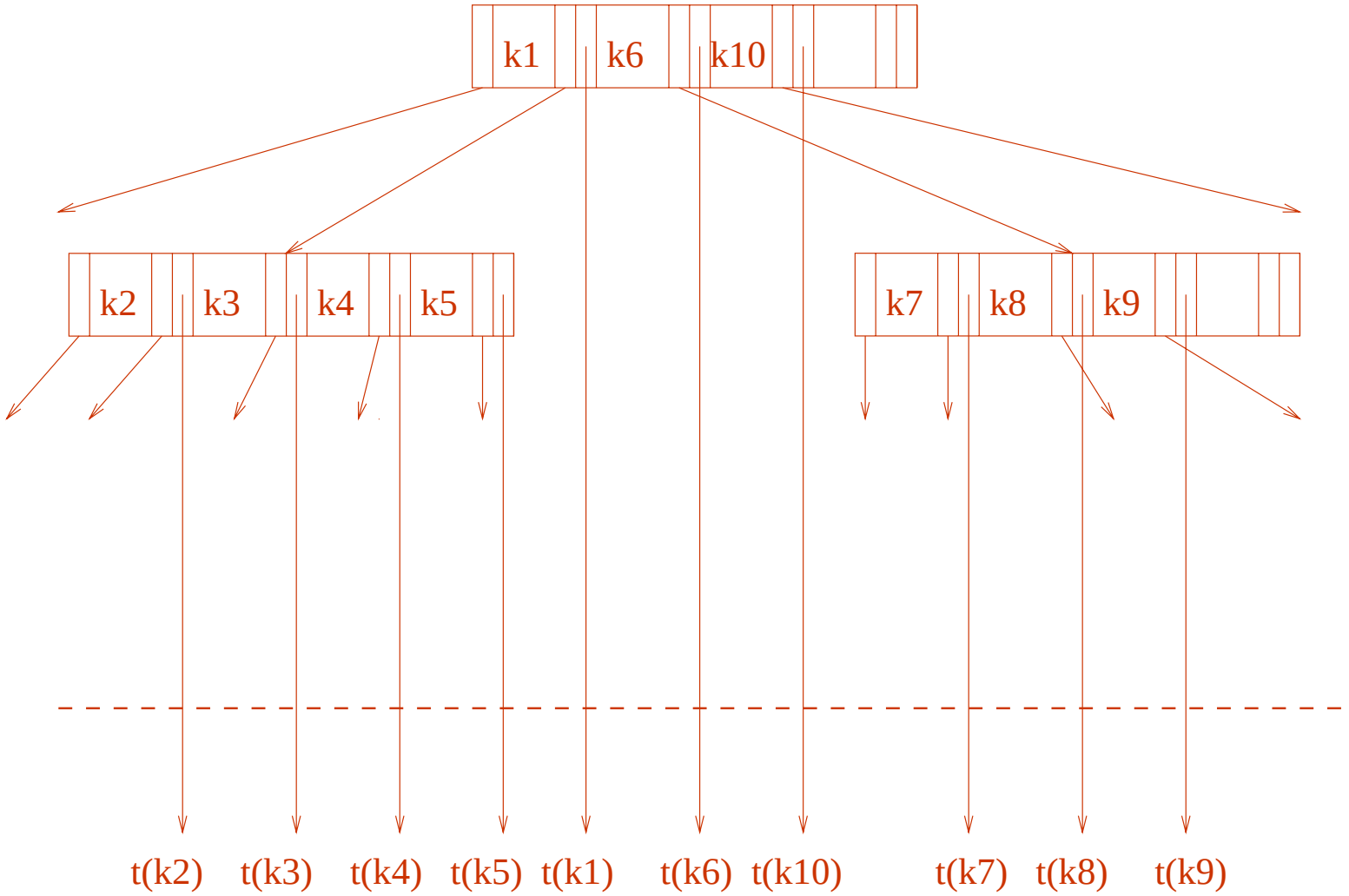
B+- tree e B-tree

- B+ tree:
  - I nodi foglia sono collegati in una lista
  - ottimi per le ricerche su intervalli
  - molto usati nei DBMS
- B tree:
  - Ciascun nodo intermedio ha un ulteriore puntatore per puntare direttamente ai dati e in genere non è previsto di collegare i nodi foglia. In tal caso:
    - p_0 indirizza al sottalbero con chiavi $K<K_1$
    - p_F indirizza al sottalbero con chiavi $K>K_F$
    - p_i indirizza al sottalbero con chiavi € (K_i, K_{i+1}**)**

Un B+ tree


nodo radice
paolo
primo livello
mauro
renzo
secondo livello
bice
dino
mauro
paolo
renzo
teresa
puntatori ai dati (organizzati in modo arbitrario)

Un B-tree


k1
k6
k10
k2
k3
k4
k5
k7
k8
k9
t(k2)
t(k3)
t(k4)
t(k5)
t(k1)
t(k6)
t(k10)
t(k7)
t(k8)
t(k9)

STRUTTURE FISICHE ED INDICI NEI DBMS RELAZIONALI

Strutture fisiche e sistemi reali

- Molti sistemi reali prevedono
  - *una struttura primaria disordinata* su cui è solo possibile definire indici secondari;
  - *un indice sulla chiave primaria* creato automaticamente dal DBMS e chiamato impropriamente “primario”;
  - *la tecnica cluster* che consiste nel memorizzare in modo “logicamente contiguo” le tuple con gli stessi valori di un certo campo –, mediante tecniche hash, indici o addirittura ordinamento fisico.
  - *cluster di più relazioni* che consentono di realizzare “join preparati” implementabili solo con una scansione.
  - *indici ISAM* (statici) di solito su struttura ordinata
  - *indici B+* che vengono chiamati nei manuali impropriamente B-tree.
  - *indici “hash”* che realizzano strutture secondarie basate sull’accesso calcolato per identificare blocchi contenenti *puntatori* ai record che contengono il valore della chiave cercata.

Definizione degli indici SQL

- Non è standard, ma presente in forma simile nei vari DBMS
  - **`create`** [***`unique`***] **`index`** *NomeIndice* **`on`** *NomeTabella* **(***ListaAttributi***)**
    - *Lista attributi* determina la sequenza di ordinamento
    - *Unique* specifica che lista attributi è una superchiave
  - **`drop index`** *NomeIndice :*
    - l'indice *non è più usato*
    - l'indice non è più utile in termini spazio-temporali

Strutture fisiche in alcuni DBMS

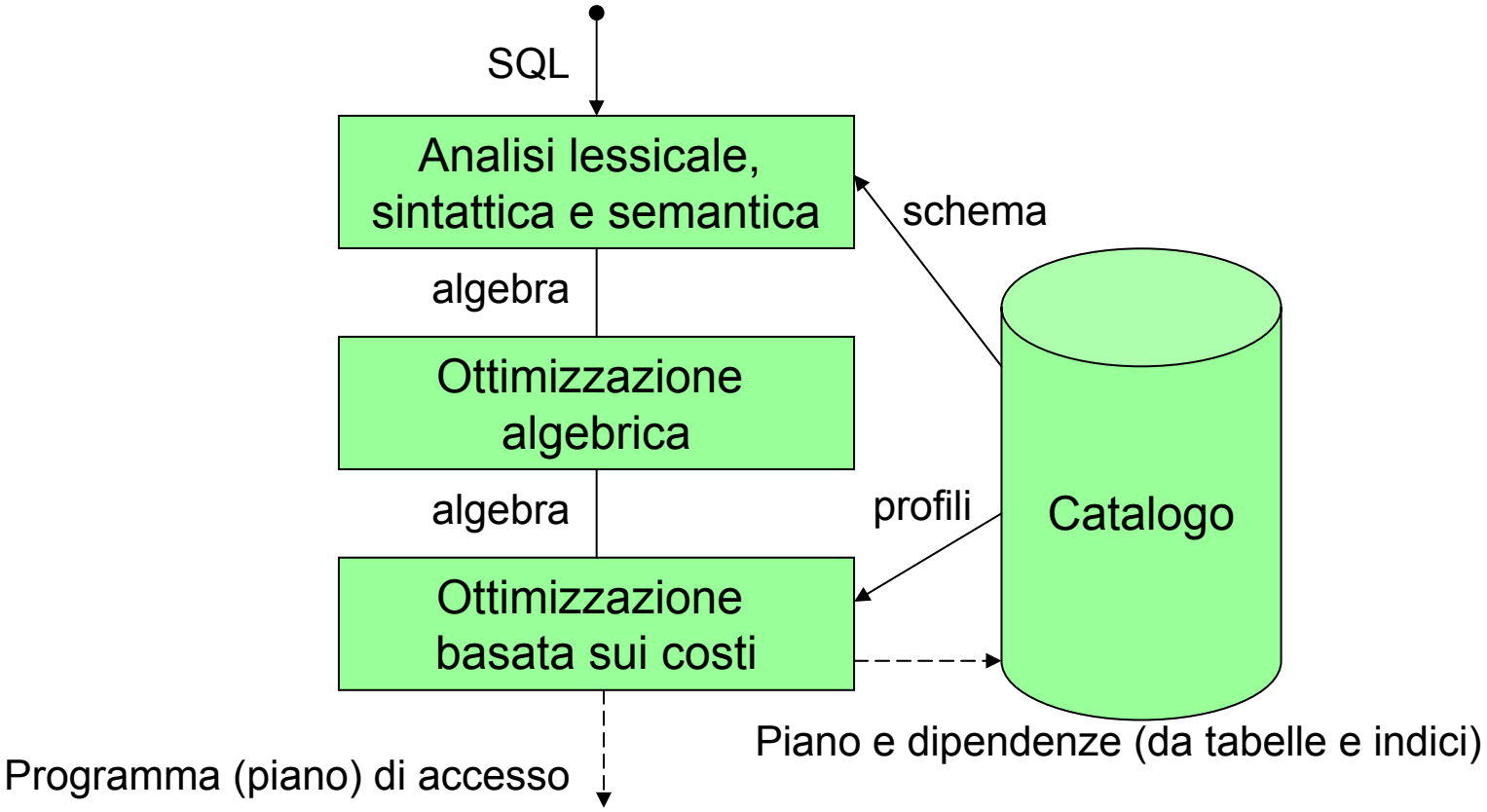
- **Oracle**:
  - struttura primaria
    - file heap
    - "hash cluster" (cioè struttura hash)
    - cluster (anche plurirelazionali) anche ordinati (con B-tree denso)
  - indici secondari di vario tipo (B-tree, bit-map, funzioni)
- **DB2**:
  - primaria: heap o ordinata con B-tree denso
  - indice sulla chiave primaria (automaticamente)
  - indici secondari B-tree densi
- **SQL Server**:
  - primaria: heap o ordinata con indice B-tree sparso
  - indici secondari B-tree densi

GESTIONE DELLE INTERROGAZIONI

Esecuzione e ottimizzazione delle interrogazioni

- ***Query processor*** (o ***Ottimizzatore***): un modulo del DBMS
- Più importante nei sistemi attuali che in quelli "vecchi" (gerarchici e reticolari):
  - le interrogazioni sono espresse ad alto livello (ricordare il concetto di **indipendenza dei dati**):
    - insiemi di tuple
    - poca proceduralità
  - l'ottimizzatore sceglie la strategia realizzativa (di solito fra diverse alternative), a partire dall'istruzione SQL

Il processo di esecuzione delle interrogazioni



"Profili" delle relazioni

- Informazioni quantitative:
  - cardinalità di ciascuna relazione :***card(T)***
  - dimensioni delle tuple : ***SIZE(T)***
  - dimensioni degli attributi: ***size (Aj,T)***
  - numero di valori distinti degli attributi: ***val(Ai,T)***
  - valore minimo e massimo di ciascun attributo: ***min(Ai,T), max(Ai,T).***
- Sono memorizzate nel "catalogo" e aggiornate con comandi del tipo "`update statistics`"
- Utilizzate nella fase finale dell'ottimizzazione, per stimare le dimensioni dei risultati intermedi.

Ottimizzazione algebrica

- Il termine ottimizzazione è improprio (anche se efficace) 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"
- Euristica fondamentale:
  - selezioni e proiezioni il più presto possibile (per ridurre le dimensioni dei risultati intermedi):
    - "push selections down"
    - "push projections down"

"Push selections"

- Assumiamo A attributo di R_2

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

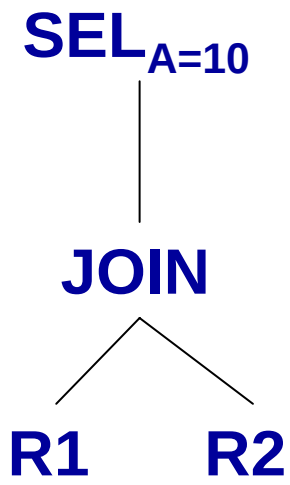
- Riduce in modo significativo la dimensione del risultato intermedio (e quindi il costo dell'operazione)

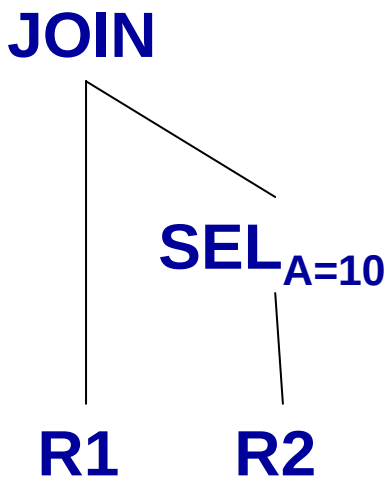
Rappresentazione interna delle interrogazioni

- Alberi:
  - foglie: dati (relazioni, file)
  - nodi intermedi: operatori (operatori algebrici, poi effettivi operatori di accesso)

Alberi per la rappresentazione di interrogazioni

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





Una procedura euristica di ottimizzazione

1. Decomporre le selezioni congiuntive (“and”) in successive selezioni atomiche
2. Anticipare il più possibile le selezioni
3. In una sequenza di selezioni, anticipare le più selettive
4. Combinare prodotti cartesiani e selezioni per formare join
5. Anticipare il più possibile le proiezioni (anche introducendone di nuove)

Esempio

R1(ABC), R2(DEF), R3(GHI)

```
SELECT   A , E
FROM     R1, R2, R3
WHERE    C=D AND B>100 AND F=G AND H=7 AND I>2
```

- prodotto cartesiano (FROM)
- selezione (WHERE)
- proiezione (SELECT)

$$\mathbf{PROJ}_{AE}\ (\mathbf{SEL}_{C=D\ AND\ B>100\ AND\ F=G\ AND\ H=7\ AND\ I>2}\ (\mathbf{(R1\ JOIN\ R2)\ JOIN\ R3)})$$

Esempio, continua

$$\text{PROJ}_{AE}\ (\text{SEL}_{C=D\ \text{AND}\ B>100\ \text{AND}\ F=G\ \text{AND}\ H=7\ \text{AND}\ I>2}\ ((R1\ \text{JOIN}\ R2)\ \text{JOIN}\ R3))$$

- diventa qualcosa del tipo

$$\text{PROJ}_{AE}\ (\text{SEL}_{B>100}\ (R1)\ \text{JOIN}_{C=D}\ R2)\ \text{JOIN}_{F=G}\ \text{SEL}_{I>2}(\text{SEL}_{H=7}(R3)))$$

- oppure

$$\text{PROJ}_{AE}(\ \text{PROJ}_{AEF}((\text{PROJ}_{AC}(\text{SEL}_{B>100}\ (R1)))\ \text{JOIN}_{C=D}\ R2)\ \text{JOIN}_{F=G}\ \text{PROJ}_{G}\ (\text{SEL}_{I>2}(\text{SEL}_{H=7}(R3))))$$

Esecuzione delle operazioni

- I DBMS implementano gli operatori dell'algebra relazionale (o meglio, loro combinazioni) per mezzo di operazioni di livello abbastanza basso, che però possono implementare vari operatori "in un colpo solo"
- Operatori fondamentali:
  - scansione
  - accesso diretto
- A livello più alto:
  - ordinamento
- Ancora più alto
  - join

Scansione

- Realizza varie operazioni (pseudo)algebriche anche one-shot:
  - proiezione con duplicati,
  - selezione su un predicato,
  - “aggiornamenti” durante la scansione.
- Primitive della scansione:
  - open
  - next
  - read
  - modify, delete; insert
  - close

Accesso diretto

- Può essere eseguito solo se:
  - il predicato dell'interrogazione è valutabile tramite indici o strutture hash:
    - $A_i = V$ oppure $V_1 \leq A_i \leq V_2$

  e
  - le strutture fisiche (primarie e/o secondarie) lo permettono

Accesso diretto basato su indice

- Efficace per interrogazioni (sulla "chiave dell'indice)
  - "puntuali" ($A_i = v$)
  - su intervallo ($v_1 \leq A_i \leq v_2$)
- Per predicati congiuntivi
  - si sceglie il più selettivo per l'accesso diretto e si verifica poi sugli altri dopo la lettura (e quindi in memoria centrale)
- Per predicati disgiuntivi:
  - servono indici su tutti, ma conviene usarli se molto selettivi e facendo attenzione ai duplicati

Accesso diretto basato su hash

- Efficace per interrogazioni (sulla "chiave dell'indice)
  - "puntuali" ($A_i = v$)
  - NON su intervallo ($v_1 \leq A_i \leq v_2$)
- Per predicati congiuntivi e disgiuntivi, vale lo stesso discorso fatto per gli indici

Ordinamento

Tengono conto:

- delle caratteristiche della memoria secondaria, con costi valutati rispetto al numero di blocchi acceduti.
- della disponibilità del buffer;

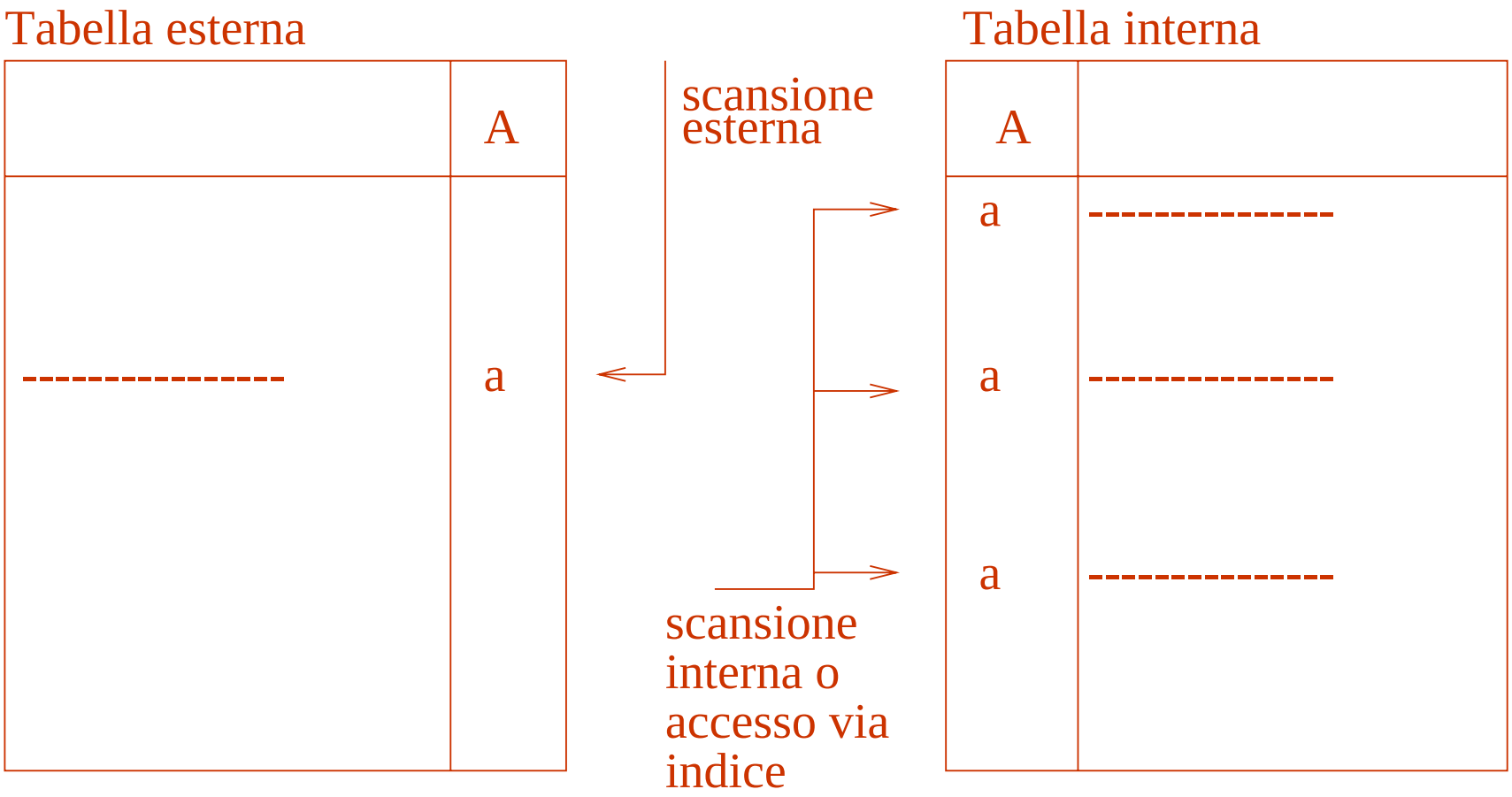
(ad es. ordinamento con tecniche di “sort – merge “)

Join

- L'operazione più costosa
- Vari metodi; i più noti:
  - *nested-loo*p, *merge-scan* and *hash-based*

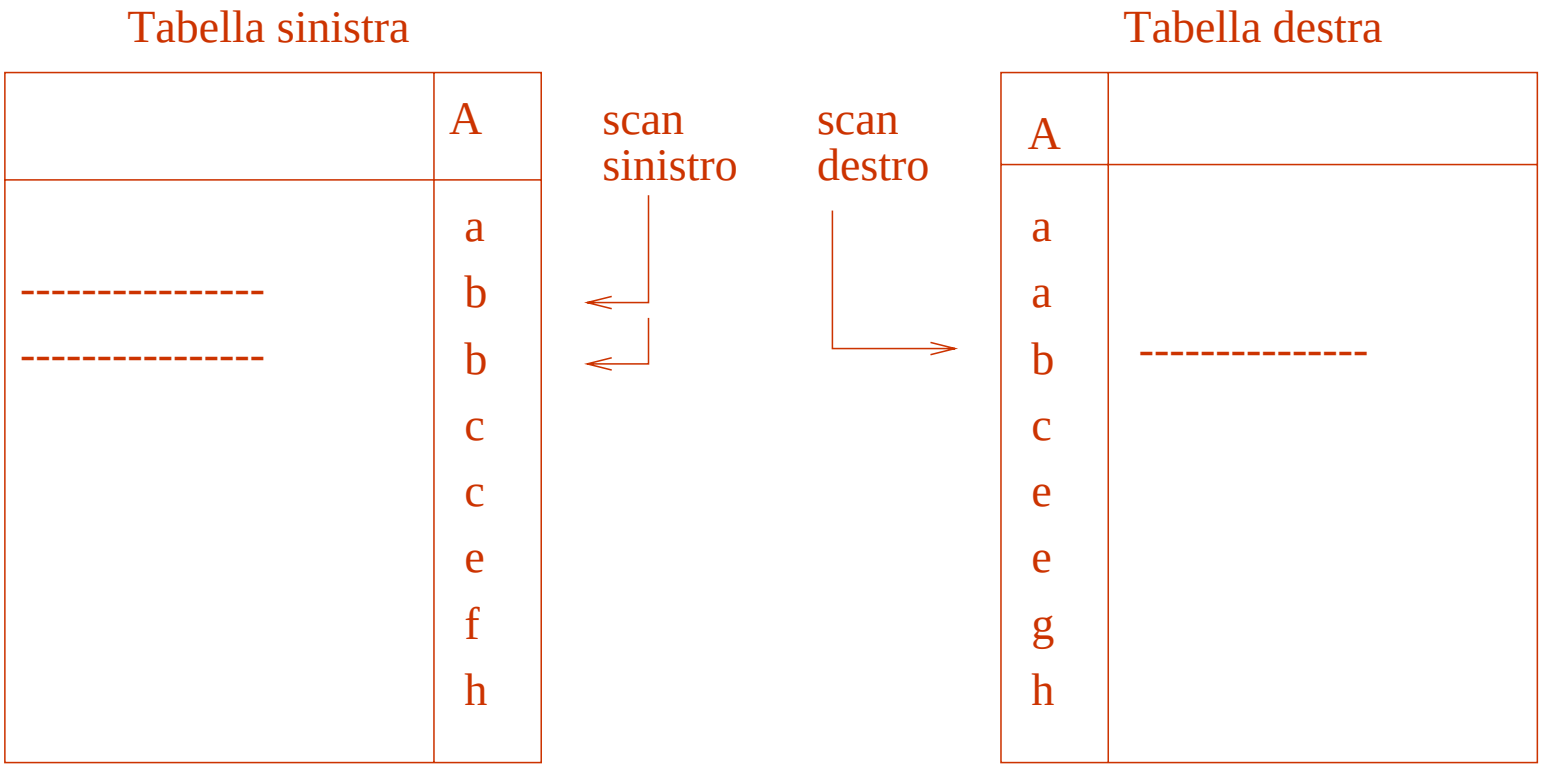
  Essi si basano sull'uso combinato di:
    - scansione
    - accesso via indice o via hash
    - ordinamento
- Il costo delle tecniche può essere valutato solo conoscendo:
  - condizione iniziale: presenza di indici, possibilità di caricamento nel buffer
  - condizione finale: ad. es. risultato ordinato.

Nested-loop (nidificata)



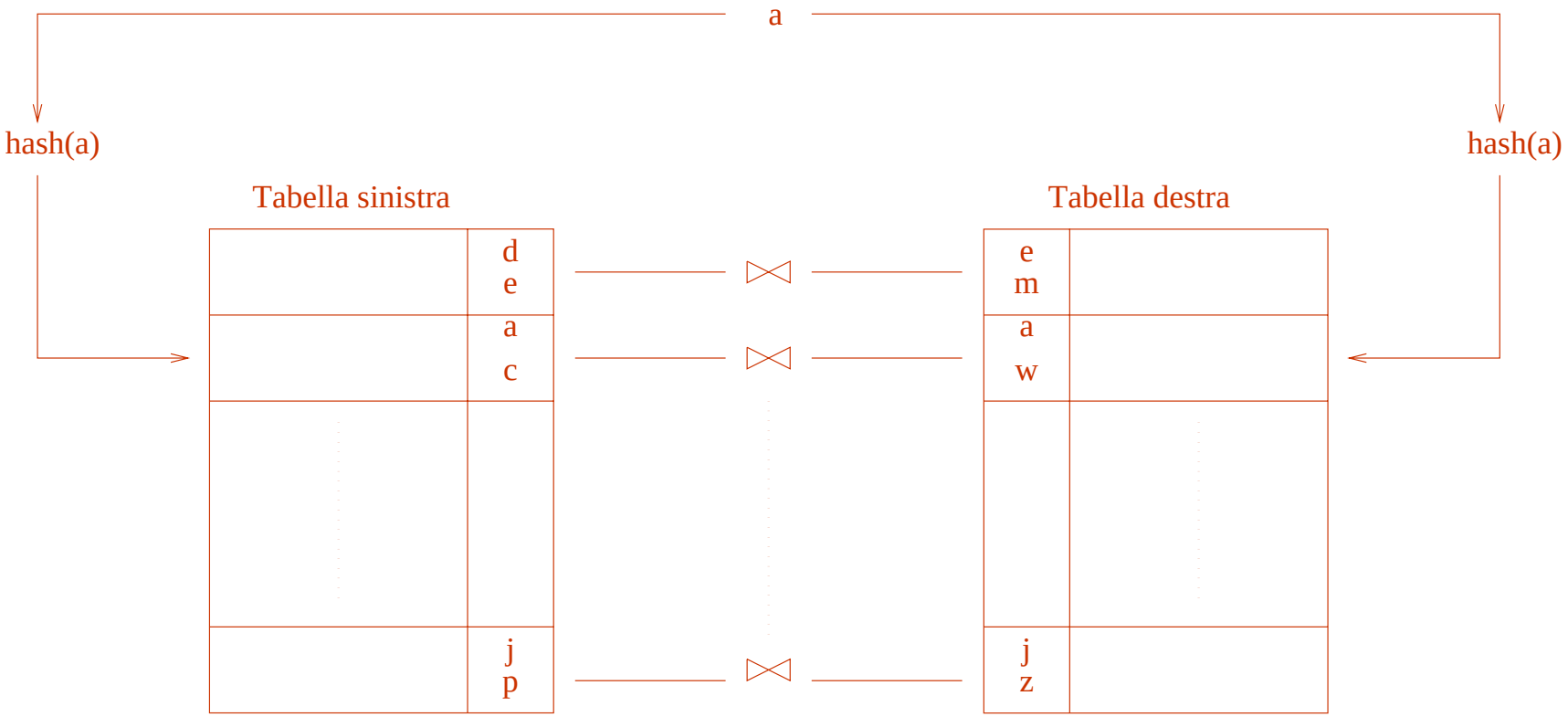
Merge-scan

- Le tabelle sono ordinate in base agli attributi di join o vengono su di esse definiti indici adeguati: lo scan è senza nidificazione.



Hash join

- Viene utilizzata una funzione hash “h” per generare copia delle due tabelle (o di loro parte);
- Si generano B partizioni (con ugual valore dell’indice) su cui effettuare un join “semplice”.



Ottimizzazione basata sui costi

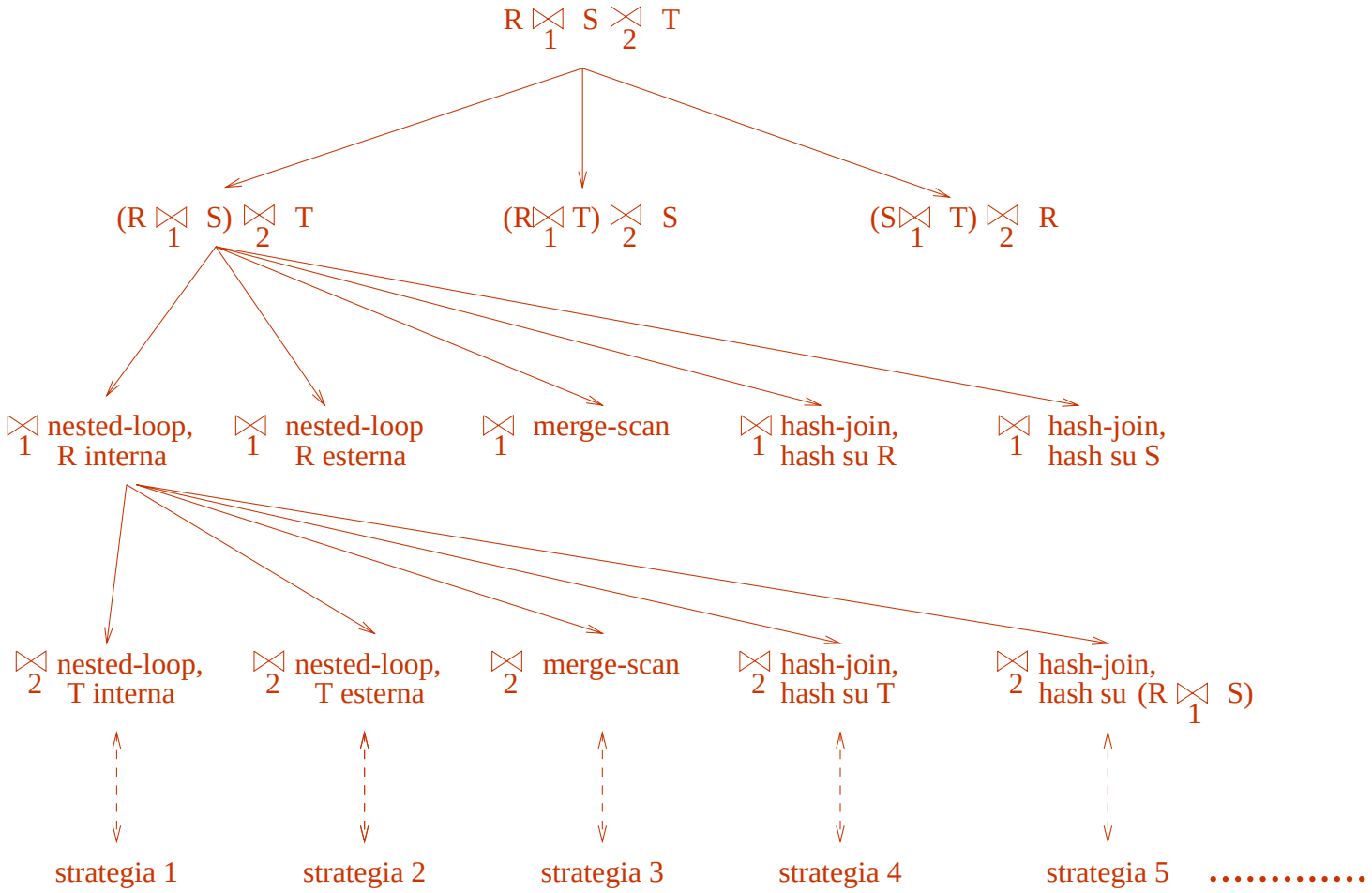
- Un problema articolato, con scelte relative a:
  - operazioni da eseguire (es.: scansione o accesso diretto?)
  - ordine delle operazioni (es. join di tre relazioni; ordine?)
  - i dettagli del metodo (es.: quale metodo di join)
- Architetture parallele e distribuite aprono ulteriori gradi di libertà

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

- L'ottimizzatore trova di solito una "buona" soluzione, non necessarimante l"ottimo"

Un albero di decisione



PROGETTAZIONE FISICA DI UNA BASE DI DATI RELAZIONALE

Progettazione fisica (1)

- La fase finale del processo di progettazione di basi di dati
- **input**
  - lo schema logico e informazioni sul carico applicativo
- **output**
  - schema fisico, costituito dalla effettiva definizione delle relazioni, dalle strutture fisiche realizzate con i relativi parametri.

Progettazione fisica (2)

- **Scelta dei valori dei parametri**:
  - Dimensioni iniziali dei file e possibilità di espansione
  - Contiguità di allocazione
  - Quantità e dimensione dei buffer ...

  esistono valori di default per tutti i parametri.
- **Le scelte fondamentali relative alle strutture fisiche** :
  - Scelta della struttura primaria di una relazione tra quelle disponibili al DBMS
  - Scelta di strutture a indici secondarie.

  Le operazioni da esaminare per tali scelte sono:
    - *selezione* e *join*

  che vengono effettuate con maggiore efficienza con l'uso di un indice.

Progettazione fisica (3)

- Le chiavi (primarie) delle relazioni sono di solito coinvolte in operazioni di selezione e join – ricorda ad esempio il funzionamento del metodo di nested loop per l'effettuazione del join – e molti sistemi, pertanto, prevedono (oppure suggeriscono) di definire *indici sulle chiavi primarie*.
- *Altri indici* vengono definiti con riferimento ad altre selezioni o join "importanti".

- **E' importante sperimentare sul campo il comportamento dell'applicazione:**
  - è spesso utile verificare se e come gli indici sono utilizzati con il comando SQL `show plan;`
  - se le prestazioni sono insoddisfacenti, si "tara" ("tuning") il sistema aggiungendo o eliminando indici.