Basi di dati Vol.2

Capitolo 1

Organizzazione fisica e gestione delle interrogazioni

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?
  - ingegneri informatici debbono sapere che cosa utilizzano e saper realizzare DBMS … (anche se in Italia pochi o nessuno lo fanno)
  - capire come funziona può favorire un utilizzo efficace
  - ci sono sistemi che offrono solo alcuni servizi

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).

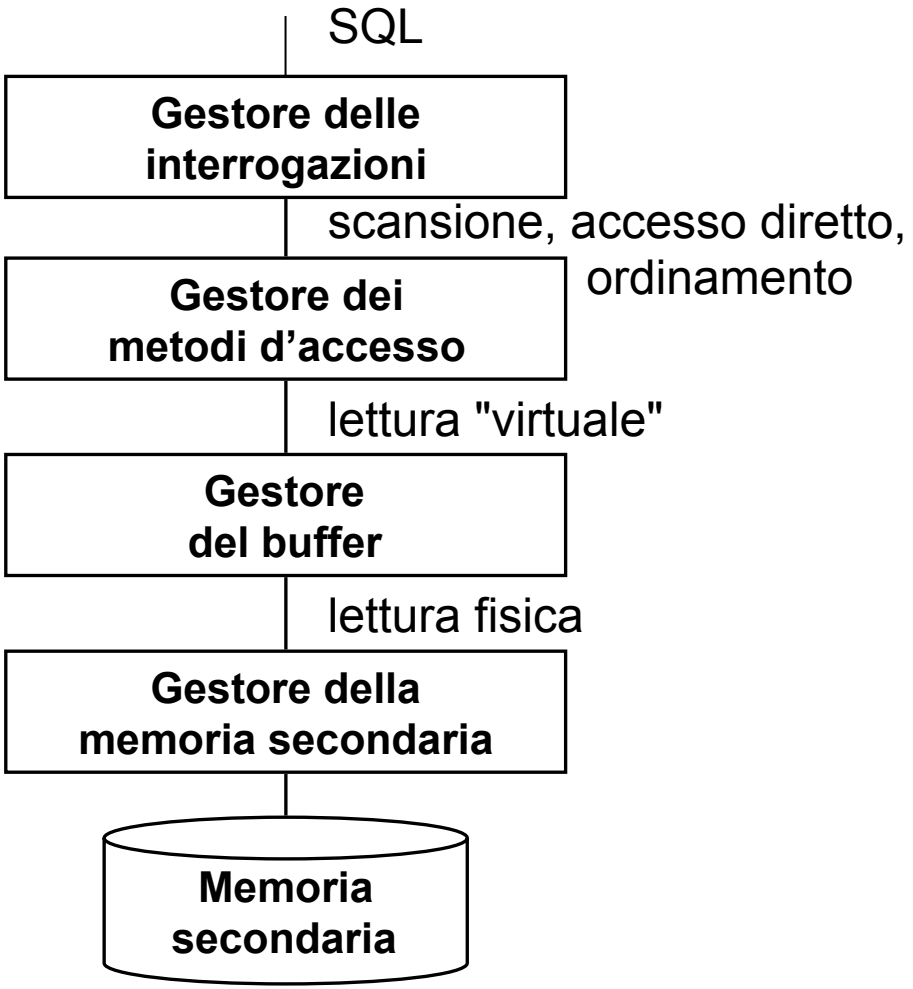
Le basi di dati sono grandi e persistenti

- 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 riscaricare)

Le basi di dati vengono interrogate …

- Gli utenti vedono il modello logico (relazionale)
- I dati sono in memoria secondaria
- Le strutture logiche non sarebbero 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: sono meglio due cursori nidificati oppure un solo cursore con una interrogazione che fa il 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 stesso posto

Due prelevamenti da un conto corrente

- Quanto c'è sul conto?
  - 1000
- Bene, allora prelevo 200
  - Ok, il nuovo saldo è 800

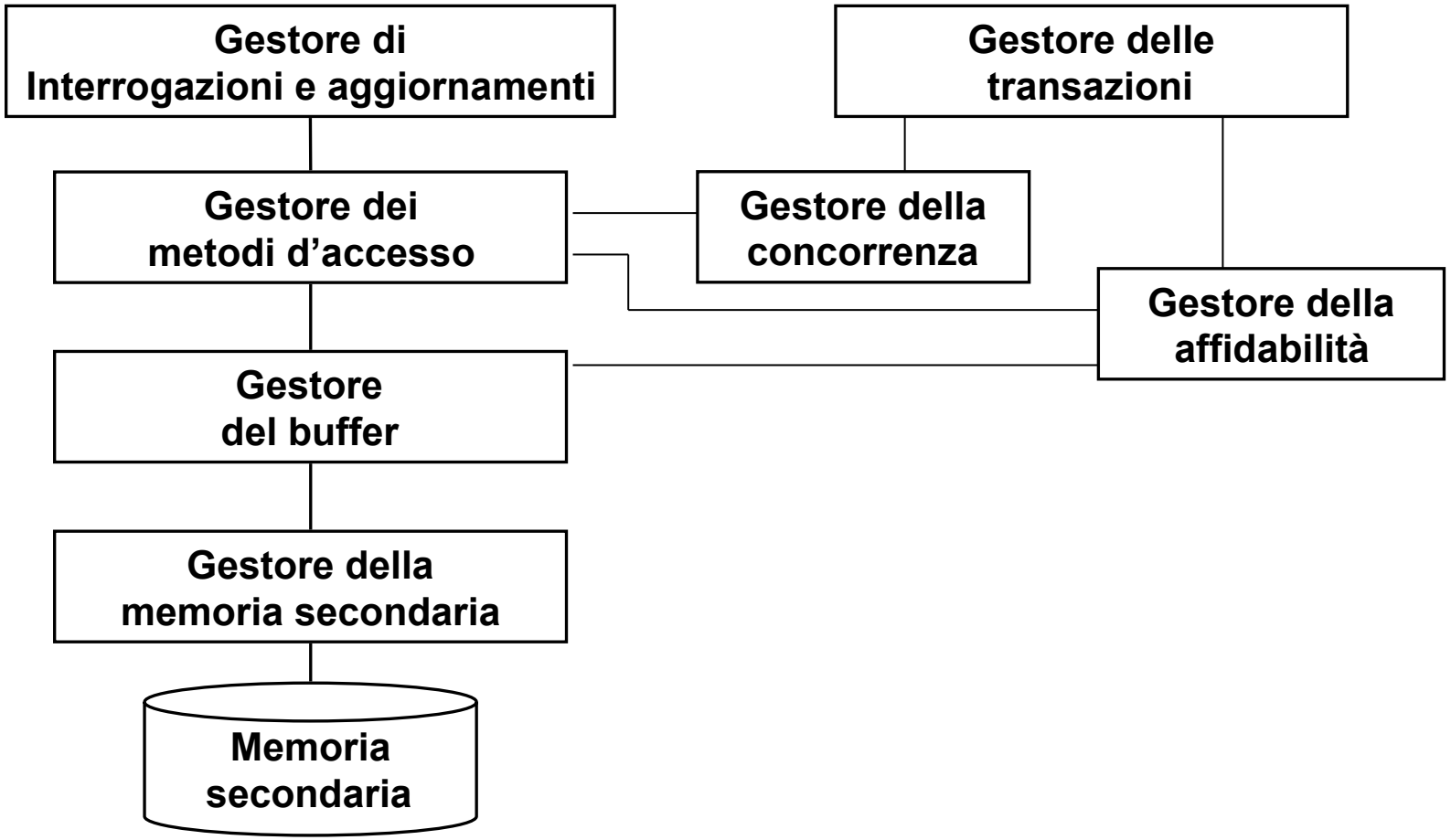
- Quanto c'è sul conto?
  - 1000
- Bene, allora prelevo 100
  - Ok, il nuovo saldo è 900

Aggiornamenti su basi di dati condivise …

- …
- 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

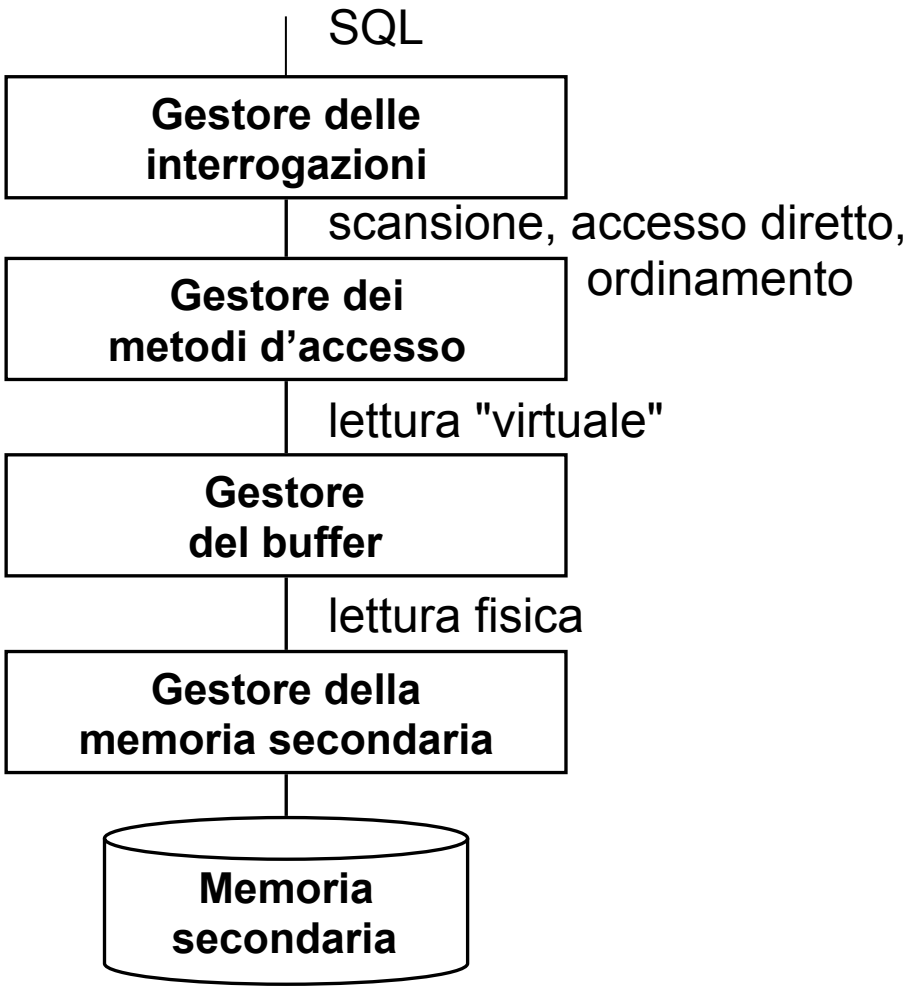


Tecnologia delle basi di dati, argomenti

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

- Architetture distribuite

Gestore degli accessi e delle interrogazioni



In concreto: SimpleDB, un DBMS didattico

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

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)
- Una **pagina** è un'area di memoria centrale della stessa dimensione di un blocco
- Le uniche operazioni sui dispositivi sono la lettura e la scrittura dei dati di un blocco (cioè di una stringa di byte) rispettivamente verso e da una pagina
- Spesso si usano i termini **blocco** e **pagina** come sinonimi
- Il sistema operativo assegna un numero ad ogni blocco del disco

Memoria principale e secondaria, 3

- Accesso a memoria secondaria (dati dal sito della Seagate e da Wikipedia, 2013), per dischi tradizionali (che ruotano)
  - tempo di **posizionamento della testina** (**seek time**): in media 4-15ms (a seconda del tipo di disco)
  - tempo di **latenza** (**rotational delay**): 2-8ms (conseguenza della velocità di rotazione, 4-15K giri al minuto)
  - tempo di **trasferimento** di un blocco: frazioni di ms (conseguenza della velocità di trasferimento, 100-600MB al secondo)
  - in totale, in media non meno di qualche ms

Memoria principale e secondaria, 4

- Commenti:
  - il costo di un accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria centrale
  - nelle applicazioni "I/O bound" (cioè con molti accessi a memoria secondaria e relativamente poche operazioni) il costo dipende (quasi) esclusivamente dal numero di accessi a memoria secondaria
  - la lettura di un bit o di un intero blocco ha lo stesso costo
  - accessi a blocchi “vicini” costano meno (contiguità), in particolare se il disco fa prefetching (legge e memorizza in una propria cache intere tracce)

Unità a stato solido

- Si stanno diffondendo e presentano caratteristiche diverse:
  - Non hanno parti meccaniche e permettono l'accesso diretto a costo uniforme
  - Quindi sono più veloci dei dischi, per l'accesso casuale
  - Costano ancora circa dieci volte di più dei dischi
  - Hanno prestazioni in scrittura che peggiorano nel tempo (perché il numero di operazioni di riscrittura è limitato e si usano tecniche che fanno scrivere in posizioni diverse)

File system

- Il file system è il componente del sistema operativo che gestisce la memoria secondaria, di solito a due livelli,
  - a livello basso, con primitive che agiscono sui blocchi
    - leggi un blocco (da disco) in una pagina (di memoria)
    - scrivi un blocco (su disco) da una pagina (di memoria)
    - alloca (cioè considera "occupati") uno o più blocchi contigui in una certa posizione
    - dealloca uno o più blocchi contigui
  - a livello più alto, con primitive sui file (intesi come sequenze di caratteri (o di oggetti) con un nome, una struttura e con una "posizione corrente")
    - posizionati su un certo carattere (*seek*)
    - leggi o scrivi in una certa posizione
    - …

DBMS e file system

- Quali funzionalità possono essere utilizzate dai DBMS?
  - a livello di blocchi
    - vantaggio: controllo completo su come i blocchi sono utilizzati e dove sono posizionati, con oggetti che si dividono su più dispositivi, se necessario o utile
    - svantaggio: i dischi debbono essere a completa disposizione del DBMS; inoltre l'amministrazione è molto più sofisticata
  - a livello di file (un file per ogni tabella, più o meno)
    - vantaggio: più facile da realizzare
    - svantaggi: il DBMS non vede i blocchi né la relativa allocazione, che invece ha impatto sulle prestazioni, e non vede il buffer e quindi non può sfruttarlo

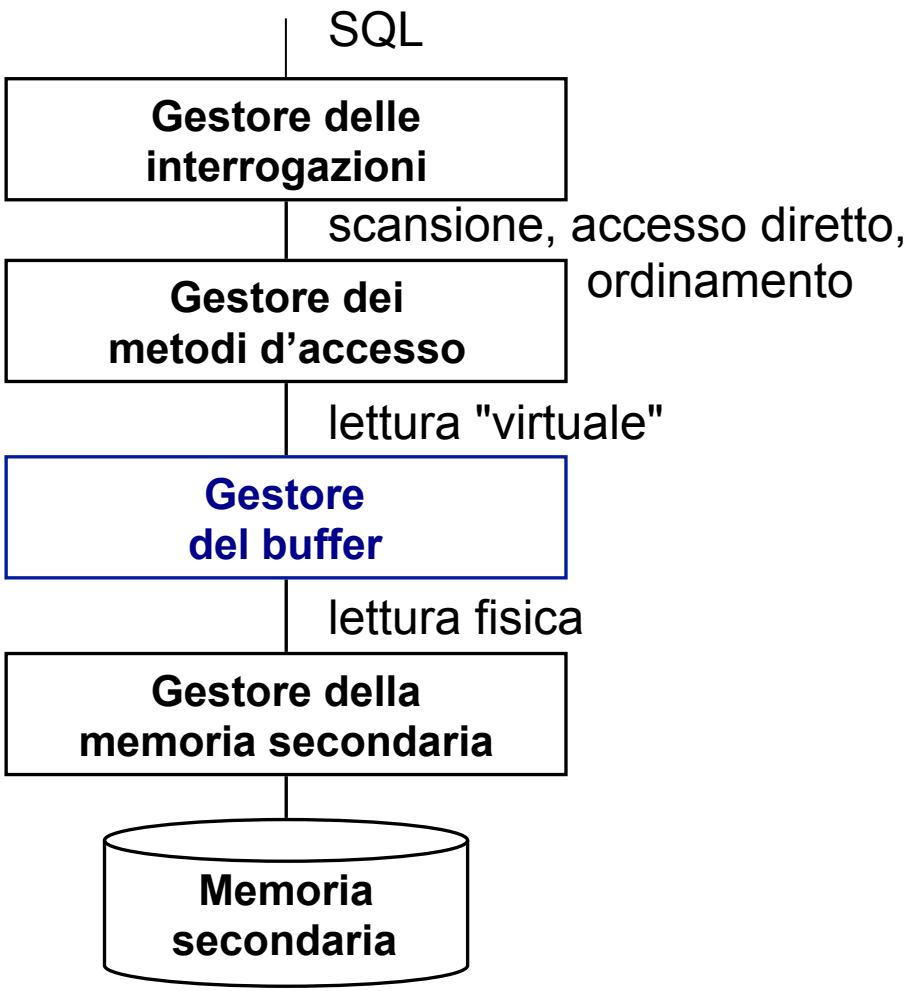
DBMS e file system, 2

- Soluzione intermedia (frequente):
  - I DBMS utilizzano le funzionalità del file system, ma in misura limitata, per creare ed eliminare file e 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.

DBMS e file system, 3

- Di solito:
  - 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.
  - Il DBMS crea file di grandi dimensioni che utilizza per memorizzare diverse relazioni (al limite, l'intera base di dati)
- Talvolta, vengono creati file in tempi successivi:
  - è possibile che un file contenga i dati di più relazioni e che le varie ennuple di una relazione siano in file diversi.
- Spesso, ma non sempre, ogni blocco è dedicato a ennuple di un'unica relazione

Gestore degli accessi e delle interrogazioni



In SimpleDB

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

Gestione dei buffer ("buffer management")

- **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

- NB, esiste spesso anche un buffer del disco (chiamato anche cache del disco) che è chiaramente una cosa diversa

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.

Buffer e memoria virtuale

- Potrebbe un DBMS usare la memoria virtuale?
  - è potenzialmente enorme, ad esempio quanto la base di dati
  - potrebbe associare ad ogni blocco su disco una pagina di memoria virtuale, lasciando al sistema operativo la responsabilità delle letture e scritture attraverso le tipiche operazioni di swap
- Dov'è il problema?

Buffer e memoria virtuale, 2

- Il DBMS deve conoscere l'uso delle pagine (in particolare se sono in memoria centrale o secondaria)
  - La memoria centrale è volatile e il DBMS deve garantire l'affidabilità e quindi deve poter scaricare su disco i blocchi da preservare (il DBMS sa quali sono, il sistema operativo no)
  - Il sistema operativo non sa quali pagine sono in uso (nel senso che si vuole continuare a utilizzarle) e quindi potrebbe fare scelte di swap che penalizzano l'efficienza
- Quindi:
  - Il buffer manager del DBMS controlla direttamente ("fisicamente") la memoria centrale

Dati gestiti dal buffer manager

- Il buffer
- Un direttorio che per ogni pagina mantiene (ad esempio)
  - il file fisico e il numero del blocco
  - due variabili di stato:
    - un contatore che indica quanti programmi utilizzano la pagina
    - un booleano 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; princìpi
  - "località dei dati":
    - è alta la probabilità di dover riutilizzare i dati attualmente in uso
    - "regola 80-20:" l'80% delle operazioni utilizza lo stesso 20% dei dati

Interfaccia offerta dal buffer manager (una possibilità, non l'unica)

- ***fix*** o ***pin****:* 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 e non ancora salvata in memoria secondaria (a seconda delle implementazioni può non essere necessaria; o essere gestita direttamente dalle operazioni di modifica)
- ***unfix*** o ***unpin****:* indica che la transazione ha concluso l'utilizzo della pagina (decrementa il contatore associato alla pagina)
- ***force*** o ***flush***: 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 (in tutti i casi positivi incrementa il contatore della pagina restituita)
  - se c'è, restituisce l'indirizzo
  - altrimenti, cerca una pagina libera nel buffer (contatore a zero);
    - se la trova,
      - se è sporca la salva
      - legge il blocco di interesse dalla memoria secondaria e restituisce l'indirizzo della pagina
    - se non la trova, due alternative
      - "**no-steal**": pone l'operazione in attesa
      - "**steal**": seleziona una "vittima", pagina occupata del buffer; scrive i dati della vittima in memoria secondaria (se "dirty"); legge il blocco di interesse dalla memoria secondaria e restituisce l'indirizzo

Commenti

- Il buffer manager richiede scritture in due contesti diversi:
  - in modo **sincrono** quando è richiesto esplicitamente con una force/flush (in seguito, cfr. gestione dell'affidabilità)
  - in modo **asincrono** quando lo ritiene opportuno (o necessario); ad esempio per liberare una pagina; inoltre, può decidere di anticipare o posticipare scritture per coordinarle e/o sfruttare la disponibilità dei dispositivi

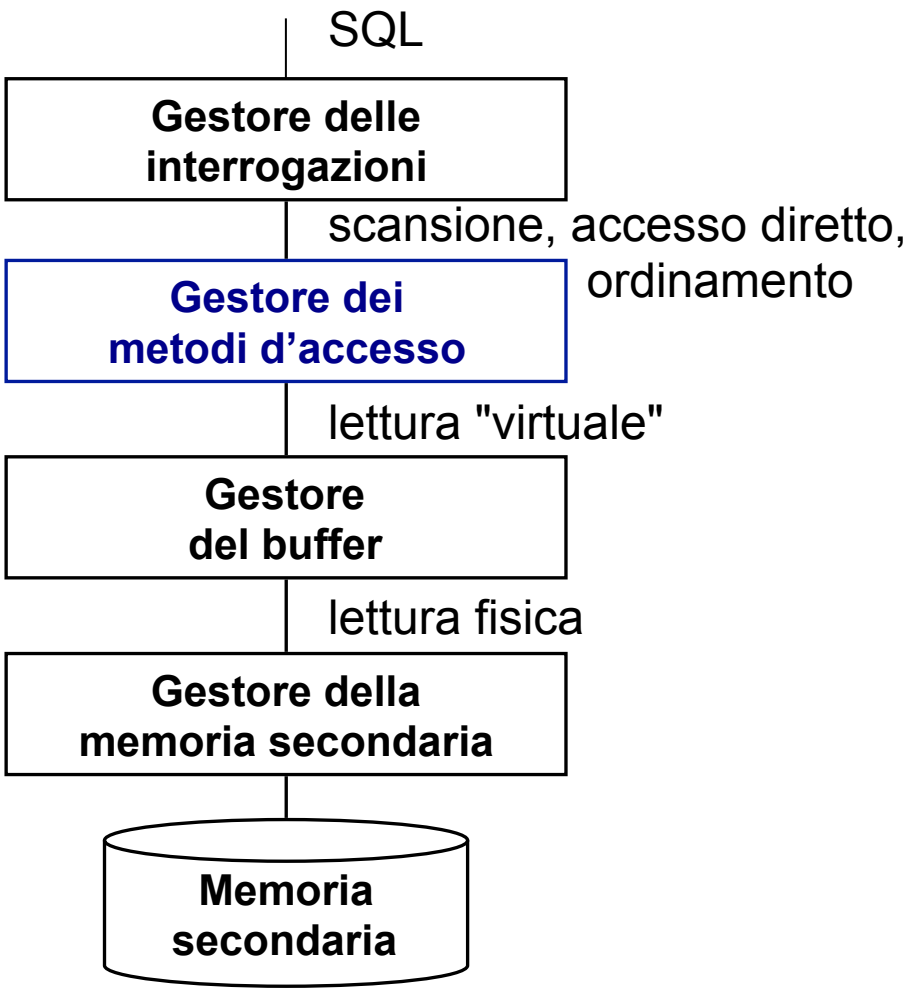
Strategia di rimpiazzo (Buffer replacement strategy)

- Quale che sia la politica (steal o no-steal), si pone l'esigenza di scegliere la pagina libera da associare al blocco (l'ideale sarebbe mantenere nel buffer le pagine che, pur libere, potrebbero essere riutilizzate)
- Strategie (supponiamo politica no-steal):
  - naif (cerca la prima pagina libera)
  - FIFO (utilizza la pagina libera che è stata caricata da più tempo)
  - LRU (utilizza la pagina libera che è stata usata meno di recente)
  - clock (fa una scansione, come nel caso naif, ma non dall'inizio, bensì dalla pagina successiva a quella del rimpiazzo precedente)

Esercizio

- Prova parziale del 28/03/2012, domanda 4

Gestore degli accessi e delle interrogazioni



In SimpleDB

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

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
  - 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:

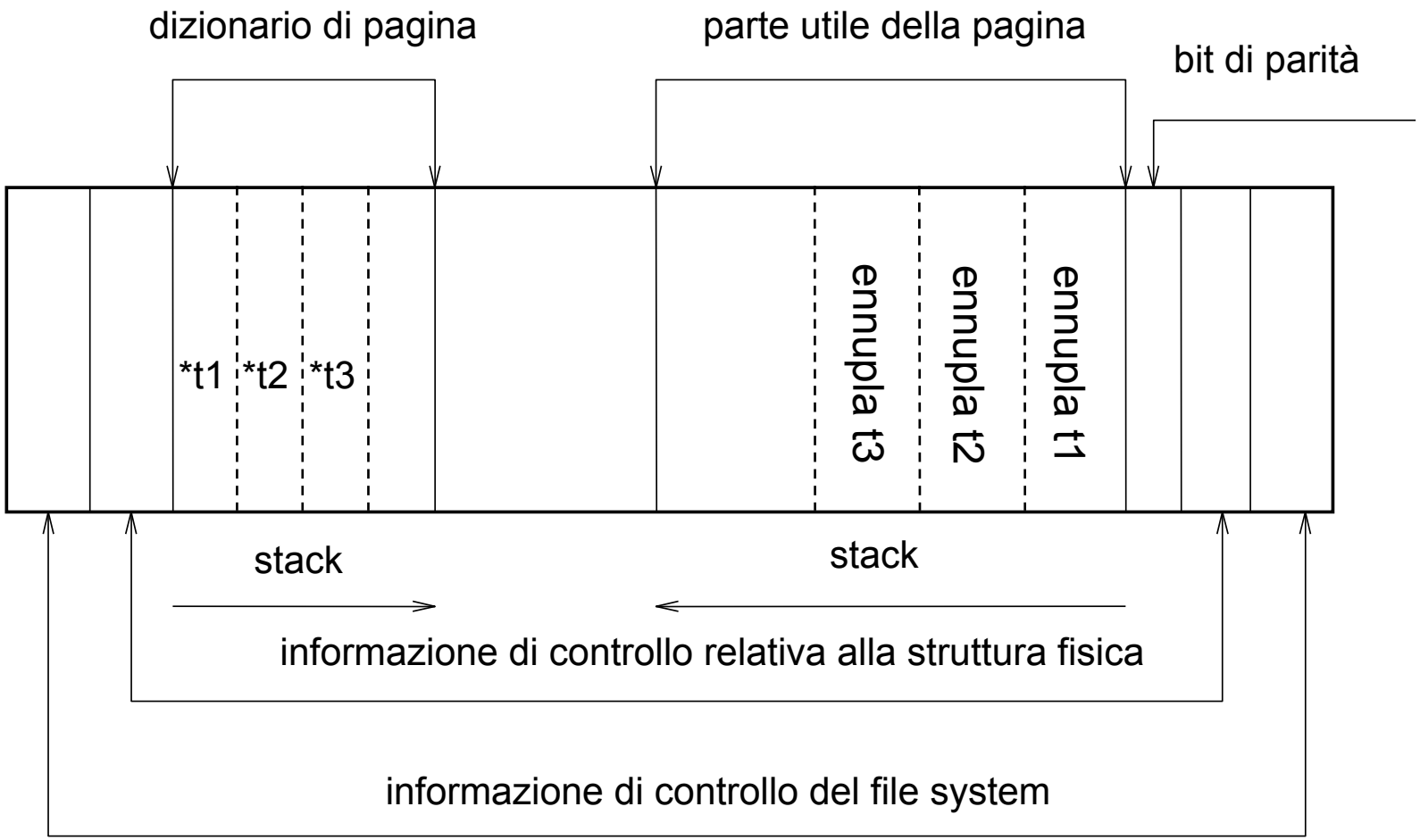
$$\lfloor L_B / L_R \rfloor$$

- lo spazio residuo può essere
  - utilizzato (record "spanned" o impaccati, di solito solo alcuni sono spezzati)
  - non utilizzato (record "unspanned")

Organizzazione delle ennuple nelle pagine

- Ci sono varie alternative, anche legate alle specifiche strutture fisiche; vediamo una possibilità (nelle esercitazioni pratiche vedremo probabilmente altri casi)
- Inoltre:
  - se la lunghezza delle ennuple è fissa, la struttura può essere semplificata
  - le ennuple (come visto, caso "spanned") possono essere su più pagine (necessario per ennuple grandi)

Organizzazione delle ennuple nelle pagine



Strutture fisiche

- Modalità di organizzazione
  - dei record in un file (o delle ennuple di una tabella):
    - strutture primarie
  - di ulteriori elementi che permettono un accesso efficiente ai record di un file
    - strutture secondarie

Tipi di strutture fisiche

- Sequenziali
  - primarie
- Calcolate ("Hash")
  - primarie (e talvolta secondarie)
- Ad albero (di solito, indici)
  - secondarie o primarie

Strutture sequenziali

- Esiste un ordinamento fra le ennuple, che può essere rilevante ai fini della gestione
  - **seriale***:* ordinamento fisico ma non logico
  - **array***:* posizioni individuate attraverso valori (numerici progessivi)
  - **ordinata**: ordinamento fisico coerente con quello di un campo

Struttura seriale

- Chiamata anche:
  - "entry sequenced"
  - file heap ("mucchio")
  - file disordinato
- Molto diffusa nelle basi di dati relazionali, associata a indici secondari
- Gli inserimenti vengono effettuati (varianti)
  - in coda (con riorganizzazioni periodiche)
  - al posto di record cancellati
- La gestione è molto semplice, ma spesso inefficiente

Array sequential structure

- Possible only when the tuples are of fixed length
- Made of *n* of adjacent blocks, each block with *m* of available slots for tuples
- Each tuple has a numeric index *i* and is placed in the *i*-th position of the array
- Primitives:
  - Accessed via `read-ind` (at a given index value).
  - Data loading happens at the end of the file (indices are obtained simply by increasing a counter)
  - Deletions create free slots
  - Updates are done on place

Ordered sequential structure

- Each tuple has a position based on the value of a “key” (or “pseudo-key”) field
- The main problems: insertions or updates which increase the physical space - they require reorganizations
- Options to avoid global reorderings:
  - Leaving a certain number of slots free at the time of first loading. This is followed by ‘local reordering’ operations
  - Integrating the sequentially ordered files with an *overflow file*, where new tuples are inserted into blocks linked to form an *overflow chain*

Strutture ordinate

- Permettono ricerche binarie ma …
  - … solo in alcuni casi (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)

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

- Obiettivo: accesso 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:
  - **funzione hash:**
    - associa ad ogni valore della chiave un "indirizzo", in uno spazio di dimensione paragonabile (leggermente superiore) 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 casuale e uniforme, riducendo le probabilità di collisione (che si riduce aumentando lo spazio ridondante)
    - esempio (utile didatticamente, ma non efficace in generale): funzione mod (resto della divisione fra interi)

Un esempio semplicissimo (alla lavagna)

- 8 record con chiavi
  - 240772
  - 240810
  - 449726
  - 447004
  - 453900
  - 281425
  - 281267
  - 405154
- tavola hash con 10 posizioni
- funzione hash
  - K mod 10

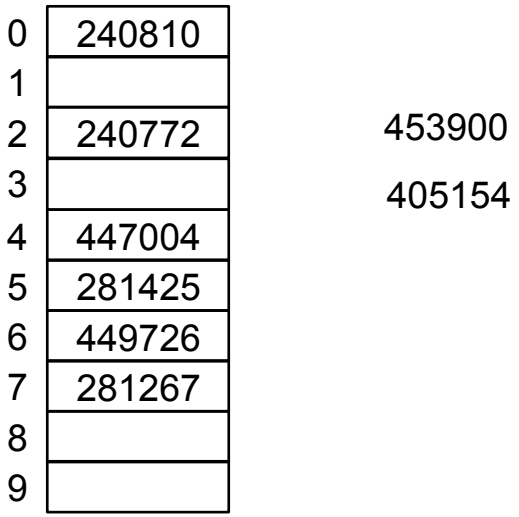


Tavola hash, un esempio semplice

- 40 record
- tavola hash con 50 posizioni:

0	60600
1	66301
2	205802
4	200604
5	66005
9	201159
10	205610
12	205912
14	200464
17	205617
18	200268
19	205619

22	210522
24	205724
27	205977
28	205478
30	200430
33	210533
37	205887
38	102338

40	102690
41	115541
42	206092
43	205693
45	205845
46	205796
48	200498
49	206049

1	205751
2	200902
2	116202
5	116455
5	200205
10	201260
10	102360
10	205460
12	205762
17	205667
38	200138
46	200296

Le chiavi e le collisioni

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

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

Tavola hash, gestione delle collisioni

- Varie tecniche:
  - posizioni successive disponibili
  - 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
  - la molteplicità media delle collisioni è molto bassa

L'esempio semplice, "costo"

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - 20 record senza collisione
- numero medio di accessi:
  (28x1+8x2+3x3+1x4)/40 = 1,425

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

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

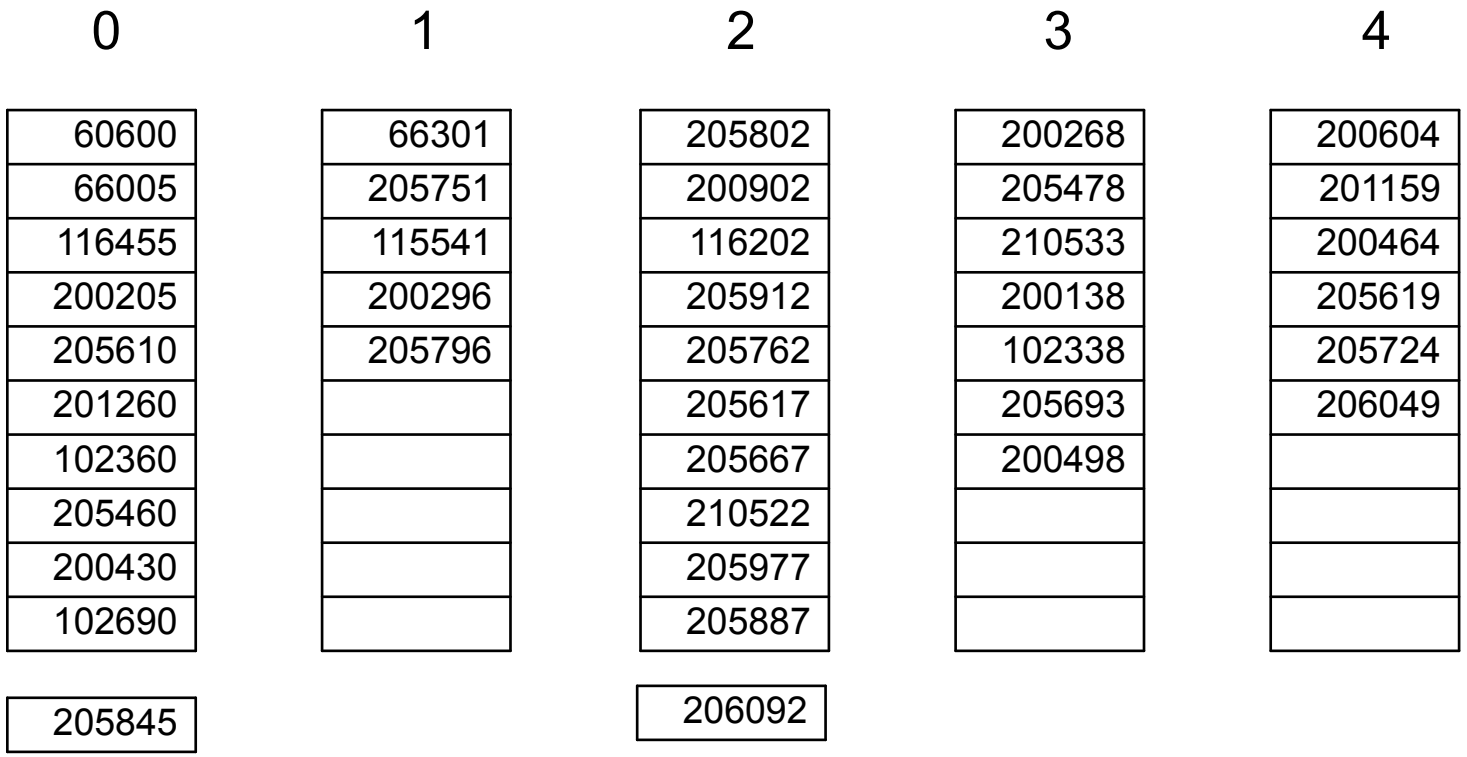
File hash

- L'idea è la stessa della tavola hash, ma si basa sull'organizzazione in blocchi:
  - ogni blocco contiene più record
  - lo spazio degli indirizzi è più piccolo
    - nell'esempio, con fattore di blocco pari a 10, possiamo usare "mod 5" invece di "mod 50"
- Le collisioni (overflow) sono di solito gestite con blocchi collegati

L'esempio semplicissimo

- Fattore di blocco 5, bastano due blocchi (alla lavagna)

File hash per l'esempio semplice



Nell'esempio semplice

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
  - in totale, 12 record in overflow

  numero medio di accessi: 1,425
- file hash con fattore di blocco 10; 5 blocchi con 10 posizioni ciascuno:
  - due soli record in overflow

  numero medio di accessi: (42/40) = 1,05
- perché?

Collisioni, stima

- Lunghezza media delle catene di overflow, al variare di
  - Numero di record esistenti: T
  - Numero di blocchi: B
  - Fattore di blocco: F
  - Cefficiente di riempimento: T/(FxB)

T/(FxB)	1	2	3	5	10	(F)
.5	0.5	0.177	0.087	0.031	0.005	
.6	0.75	0.293	0.158	0.066	0.015	
.7	1.167	0.494	0.286	0.136	0.042	
.8	2.0	0.903	0.554	0.289	0.110	
.9	4.495	2.146	1.377	0.777	0.345	

File hash, osservazioni

- È l'organizzazione più efficiente per l'accesso diretto basato su valori della chiave con condizioni di uguaglianza (accesso puntuale):
  - costo medio di poco superiore all'unità (il caso peggiore è molto costoso ma talmente improbabile da poter essere ignorato)
- Non è efficiente per ricerche basate su intervalli (né per ricerche basate su altri attributi)
- I file hash "degenerano" se si riduce lo spazio sovrabbondante: funzionano solo con file la cui dimensione non varia molto nel tempo

Strutture primarie e secondarie

- Le strutture seriali (sequenziali, ordinate, etc.) sono strutture primarie
- Le strutture hash, come viste finora sono pure primarie
- Vediamo ora strutture che nascono come secondarie (anche se loro varianti possono essere primarie)

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 chiave (o, meglio, pseudochiave, perché 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

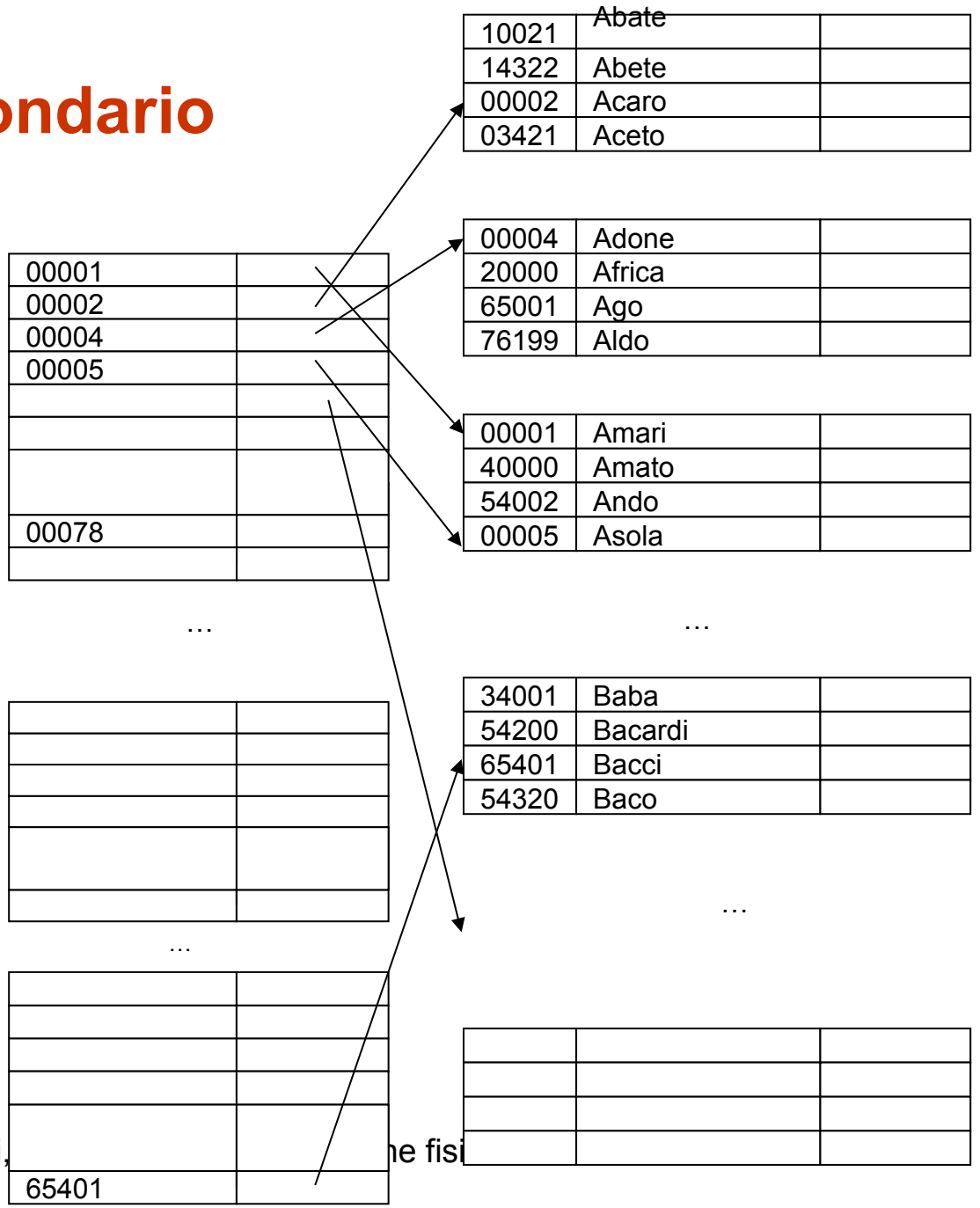
Tipi di indice

- indice primario:
  - su un campo sul cui ordinamento è basata la memorizzazione (detti anche indici di cluster, anche se talvolta si chiamano primari quelli su una chiave identificante e di cluster quelli su una pseudochiave non identificante)
- indice secondario
  - su un campo con ordinamento diverso da quello di memorizzazione

Tipi di indice, commenti

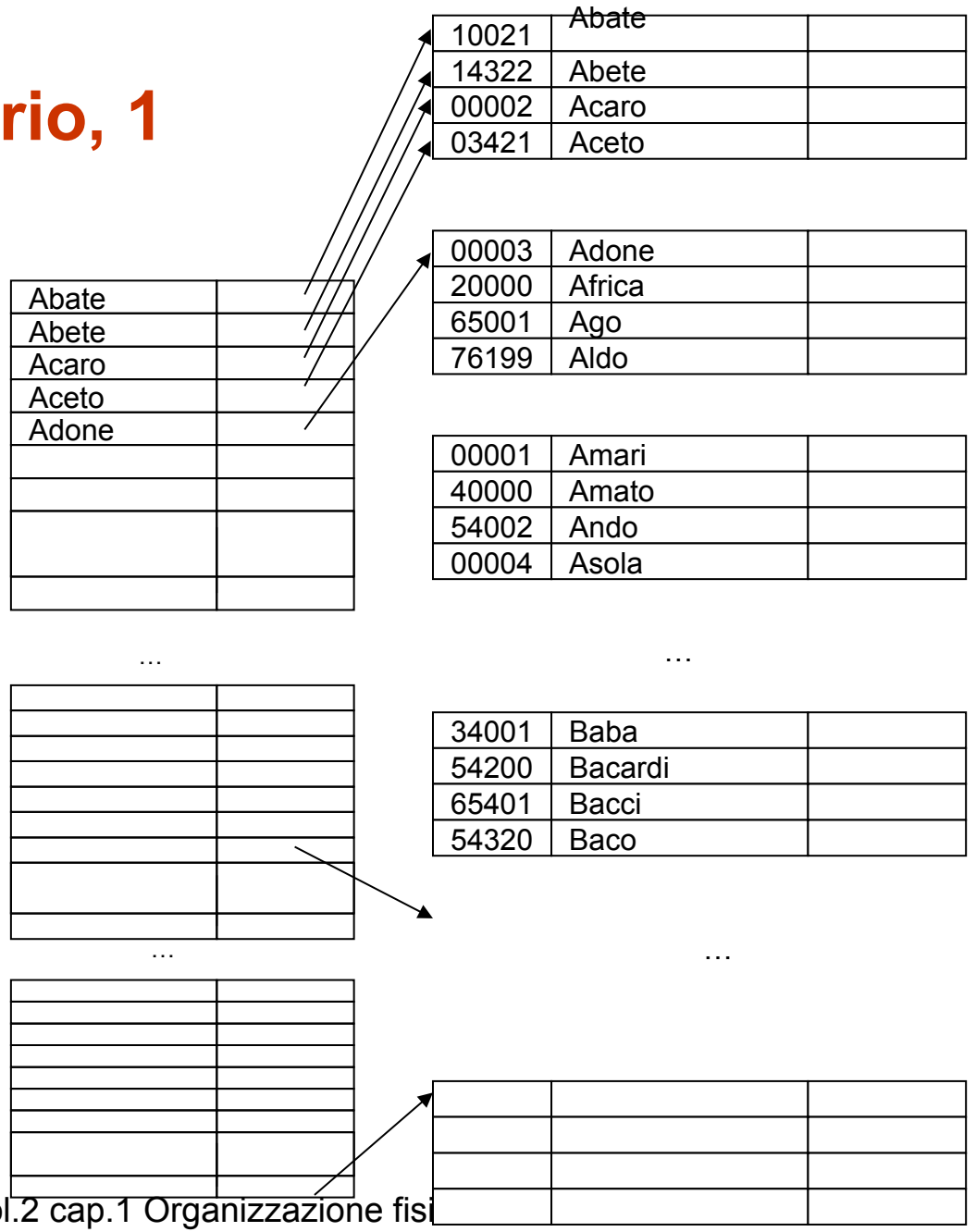
- Esempio, sempre rispetto ad un libro
  - indice generale
  - indice analitico
- Ogni file può avere al più un indice primario e un numero qualunque di indici secondari (su campi diversi). Esempio:
  - una guida turistica può avere l'indice dei luoghi e quello degli artisti
- Un file hash non può avere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

Indice secondario

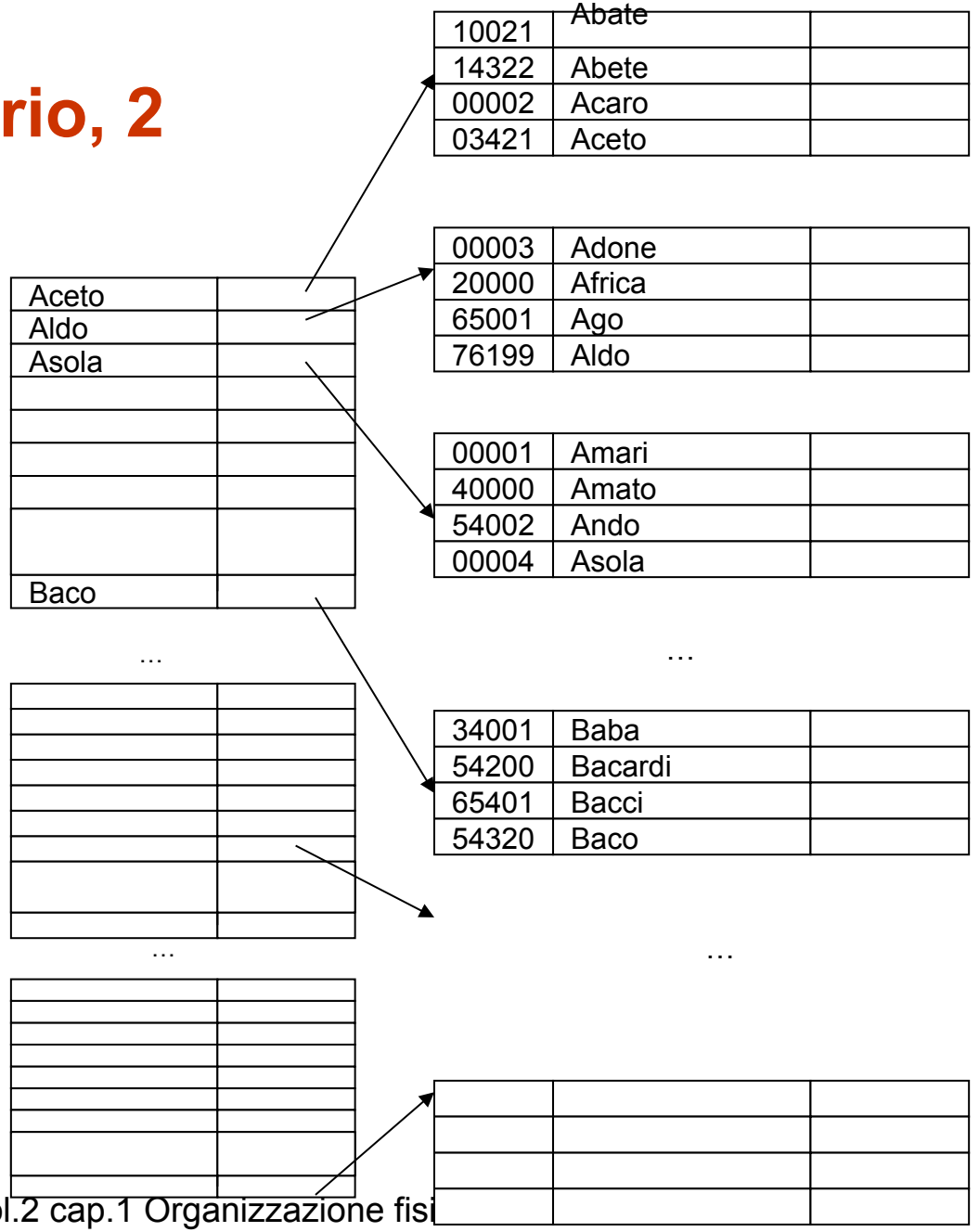


Indice primario, 1

Servono tutti i riferimenti?



Indice primario, 2



Tipi di indice, ancora

- indice denso:
  - contiene tutti i valori della chiave (e quindi, per indici su campi identificanti, un riferimento per ciascun record del file)
- indice sparso:
  - contiene solo alcuni valori della chiave e quindi (anche per indici su campi identificanti) un numero di riferimenti inferiore rispetto ai record del file

- Un indice secondario
  - deve essere denso
- Un indice primario
  - può essere sparso (e di solito lo è)
  - può essere denso per permettere operazioni sugli indirizzi, senza accedere ai record

Indici densi, un’osservazione

- Si possono usare, come detto, puntatori ai blocchi oppure puntatori ai record
  - I puntatori ai blocchi sono più compatti
  - I puntatori ai record permettono di
    - semplificare alcune operazioni (effettuate solo sull'indice, senza accedere al file se non quando indispensabile)

Dimensioni dell'indice

- L numero di record nel file
- B dimensione dei blocchi
- R lunghezza dei record (fissa)
- K lunghezza del campo pseudochiave
- P lunghezza degli indirizzi (ai blocchi)

N. di blocchi per il file (circa): $N_F = L / (B/R)$

N. di blocchi per un indice denso: $N_D = L / (B/(K+P))$

N. di blocchi per un indice sparso: $N_S = N_F / (B/(K+P))$

Dimensioni dell'indice, esempio

- L numero di record nel file 1.000.000
- B dimensione dei blocchi 4KB
- R lunghezza dei record (fissa per semplicità) 100B
- K lunghezza del campo chiave 4B
- P lunghezza degli indirizzi (ai blocchi) 4B

$N_F = L / (B/R)$ =~ 1.000.000/(4.000/100) = 25.000

$N_D = L / (B/(K+P))$ =~ 1.000.000/(4.000/8) = 2.000

$N_S = N_F / (B/(K+P))$ =~ 25.000/(4.000/8) = 50

Caratteristiche degli indici

- Accesso diretto (sulla 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 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
    - … (vedremo più avanti)

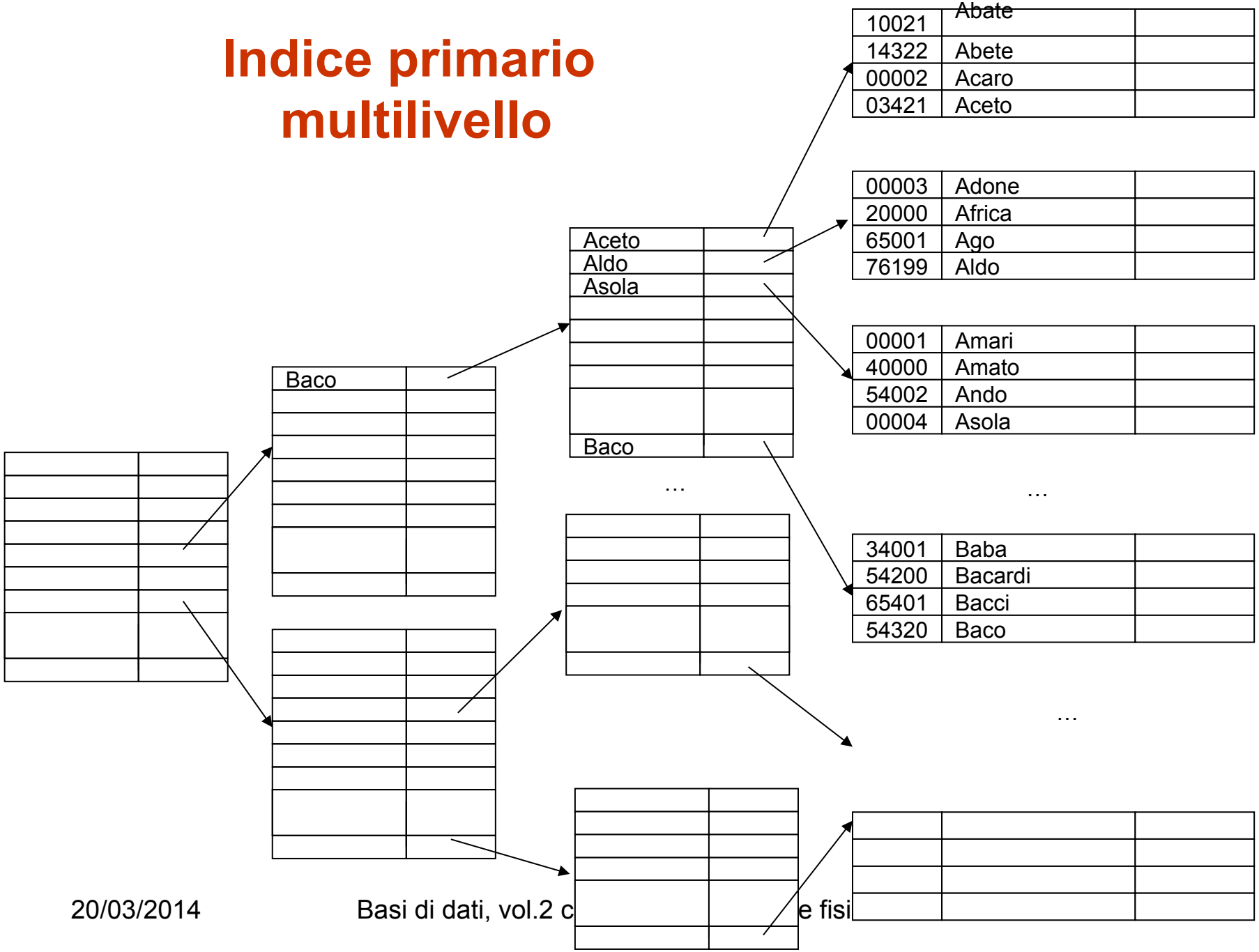
Indici su campi non chiave

- Ci sono (in generale) più record per un valore della (pseudo)chiave
  - primario sparso, possibili semplificazioni:
    - puntatori solo a blocchi con valori “nuovi”
  - primario denso:
    - per ogni record, una coppia con valore della chiave e riferimento (quindi i valori della chiave si ripetono)
    - valore della chiave una sola volta, seguito dalla lista di riferimenti ai record con quel valore
    - valore della chiave, seguito dal riferimento al primo record con quel valore (perde i benefici dell’indice primario denso legati alla possibilità di lavorare sui puntatori)
  - secondario (denso):
    - una coppia con valore della chiave e riferimento per ogni record (quindi i valori della chiave si ripetono)
    - un livello (di “indirezione”) in più: per ogni valore della chiave l’indice contiene un record con riferimento al blocco di una struttura intermedia che contiene riferimenti ai record

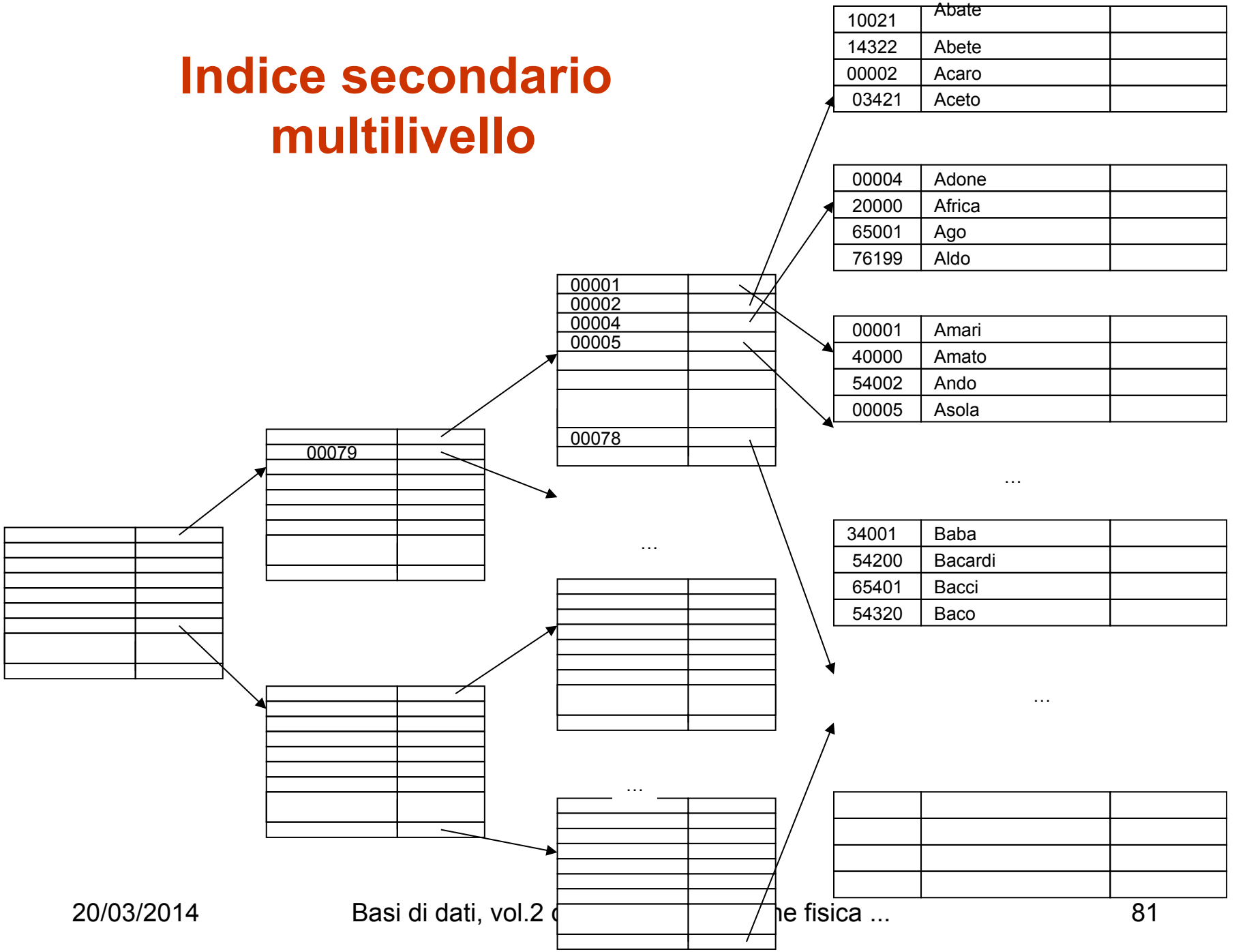
Indici multilivello

- Gli indici sono file essi stessi e quindi ha senso costruire indici sugli indici, per evitare di fare ricerche fra blocchi diversi (che potrebbero richiedere scansioni sequenziali)
- L'indice è ordinato e quindi l'indice sull'indice è primario (e sparso)
- Il tutto a più livelli, fino ad avere un livello con un solo blocco

Indice primario multilivello



Indice secondario multilivello



Indici multilivello

- 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):
  - $N_j = N_{j-1} / (B/(K+P))$

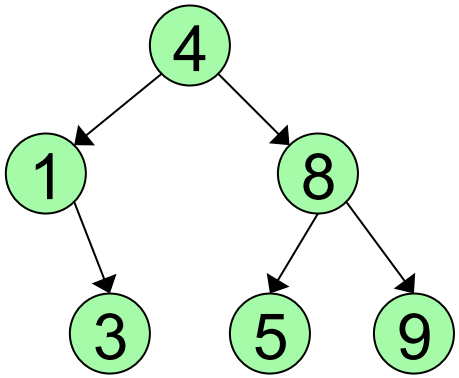
- Negli esempi numerici (B/(K+P) = 4.000/8=500)
  - Denso: $N_1 = 2.000$, $N_2 = 4$, $N_3 = 1$
  - Sparso: $N_1 = 50$, $N_2 = 1$

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)

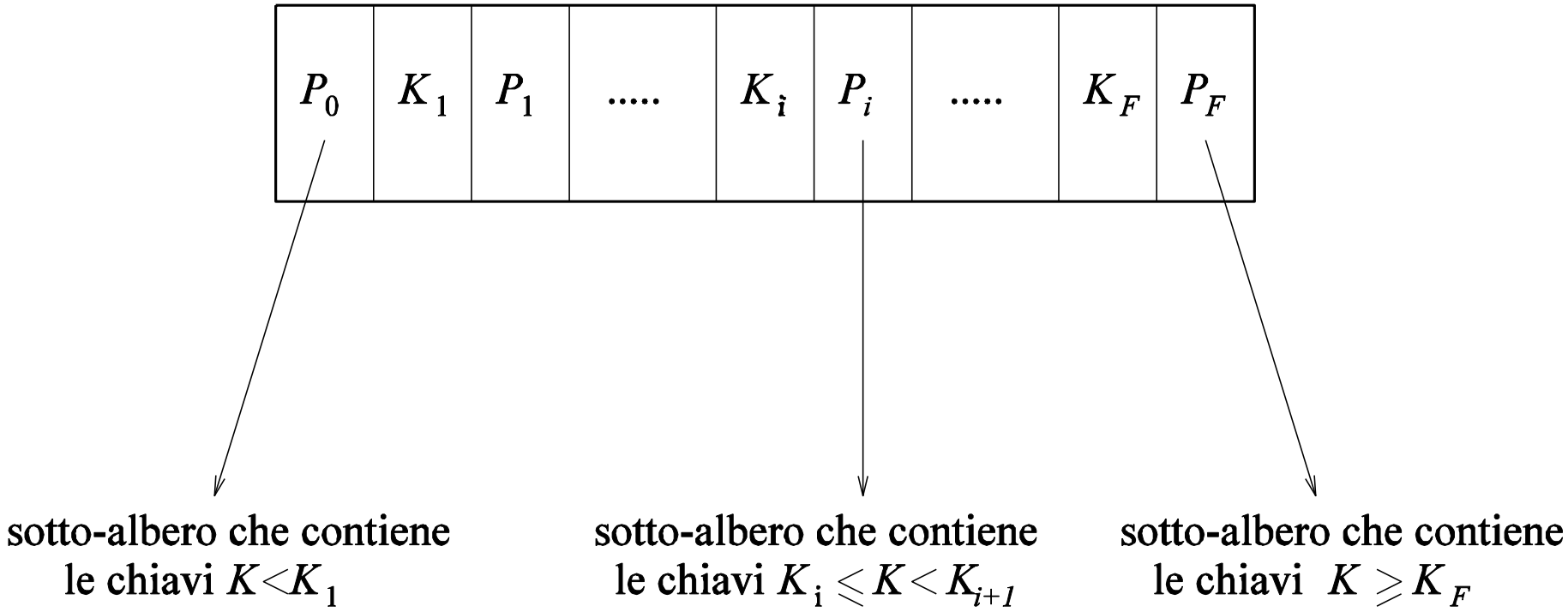


Albero di ricerca di ordine P

- Ogni nodo ha (fino a) P figli e (fino a) P-1 etichette, ordinate
- Nell'i-esimo sottoalbero abbiamo tutte etichette maggiori della (i-1)- esima etichetta e minori della i-esima
- Ogni ricerca o modifica comporta la visita di un cammino radice foglia
- In strutture fisiche, un nodo corrisponde di solito ad un blocco e quindi ogni nodo intermedio ha molti figli (un “fan-out” molto grande, pari al fattore di blocco dell’indice)
- All’interno di un nodo, la ricerca è sequenziale (ma in memoria centrale!)

- La struttura è ancora (potenzialmente) rigida

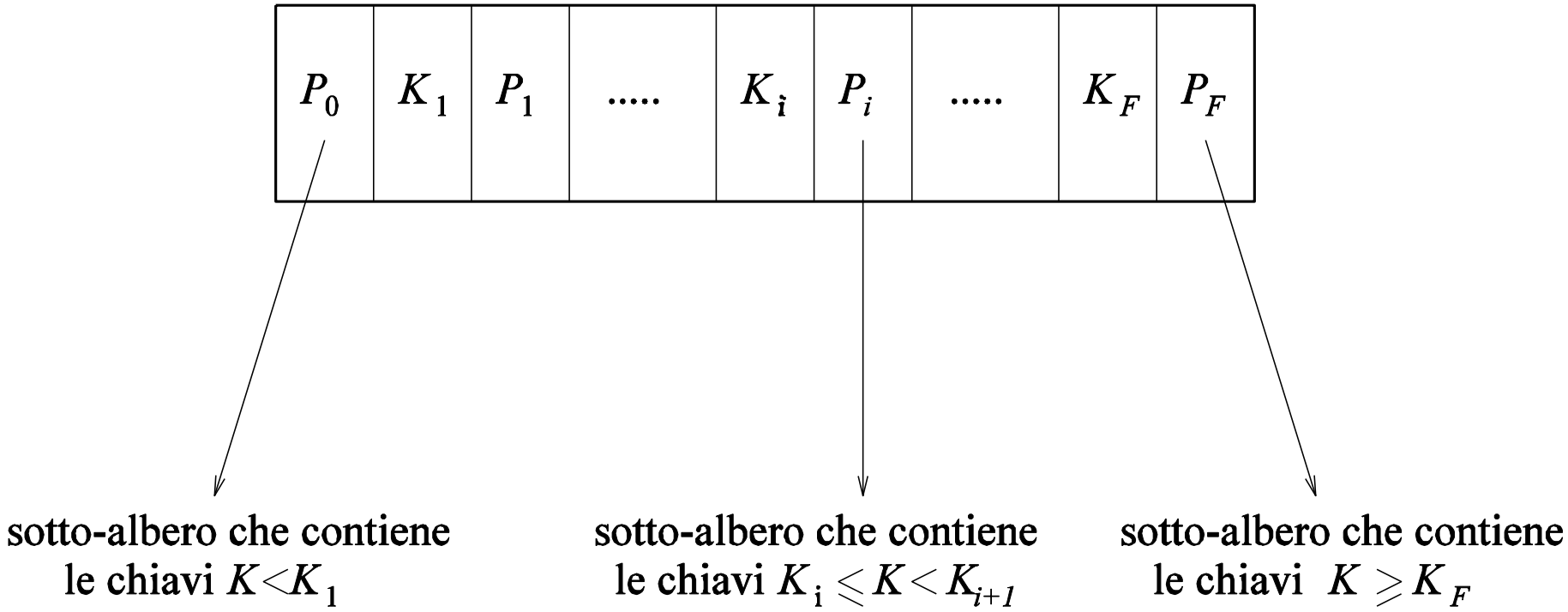
Nodi in un albero di ricerca di ordine F+1



B-tree

- Albero di ricerca in cui ogni nodo corrisponde ad un blocco,
  - viene mantenuto perfettamente bilanciato (tutte le foglie sono allo stesso livello), grazie a:
    - riempimento parziale (mediamente 70%)
    - riorganizzazioni (locali) in caso di sbilanciamento

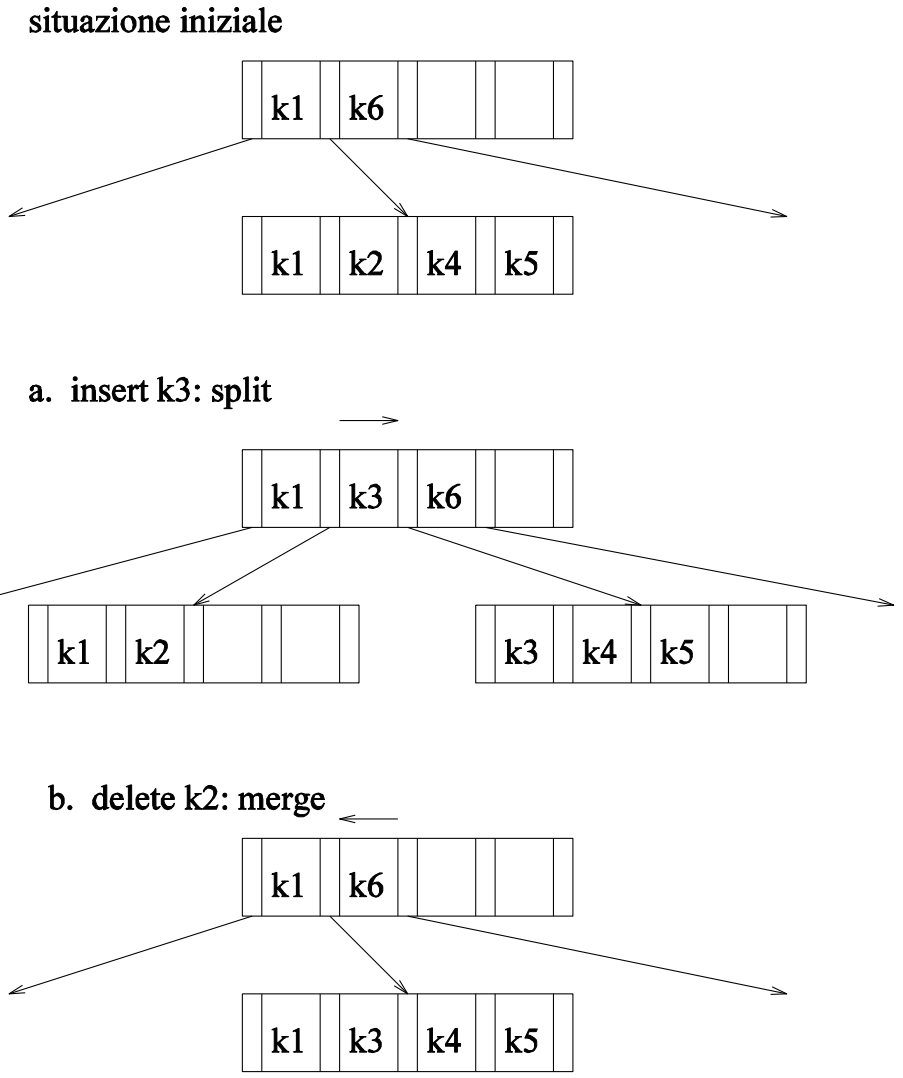
Organizzazione dei nodi del B-tree



Split e merge

- Inserimenti ed eliminazioni sono precedute da una ricerca fino ad una foglia
- 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
- Vedi applet

Split and merge operations



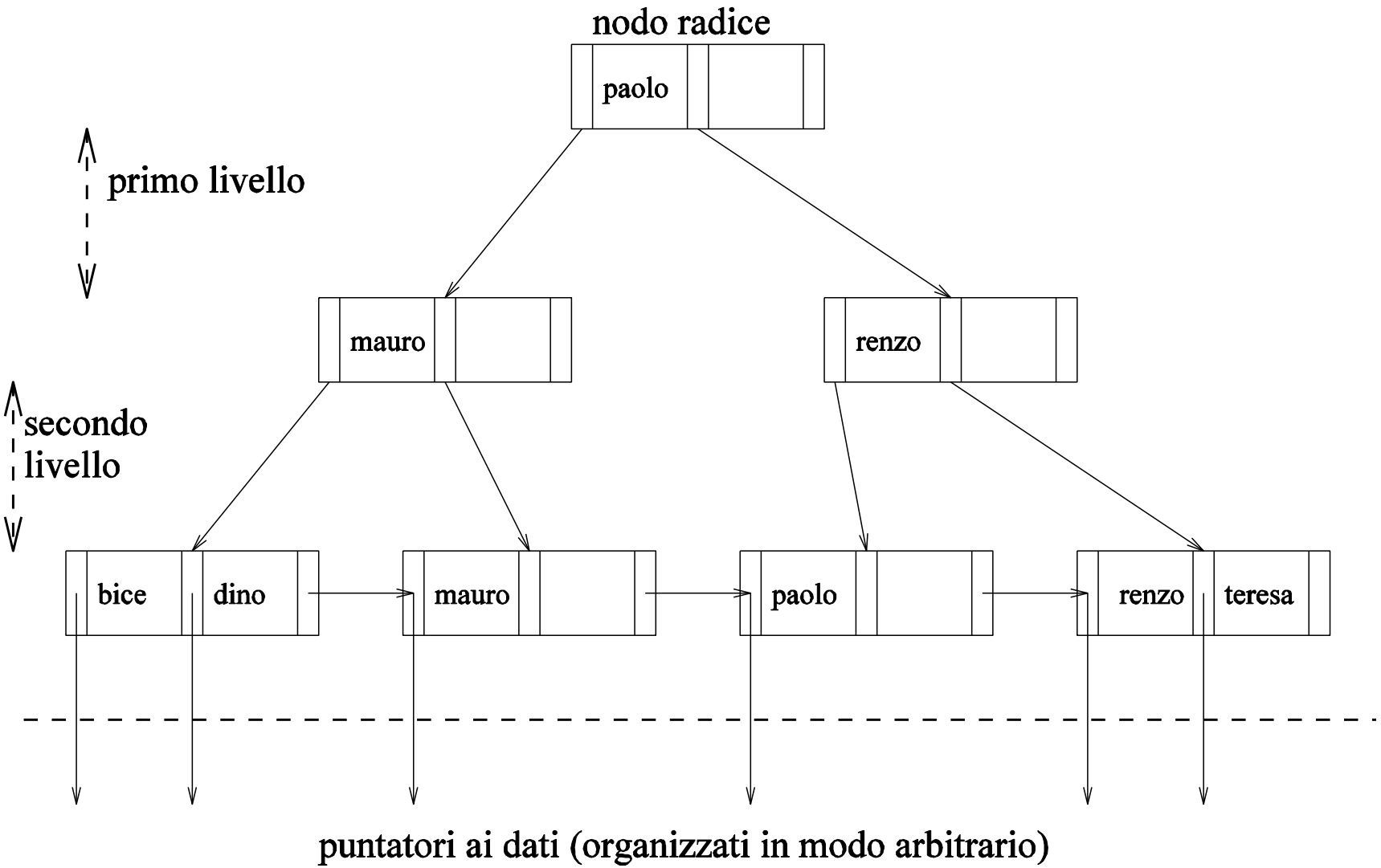
B tree e B+ tree

- B+ tree:
  - le chiavi compaiono tutte nelle foglie (e quindi quelle nei nodi intermedi sono comunque ripetute nelle foglie)
  - le foglie sono collegate in una lista
  - ottimi per le ricerche su intervalli
  - molto usati nei DBMS
- B tree:
  - Le chiavi che compaiono nei nodi intermedi non sono ripetute nelle foglie

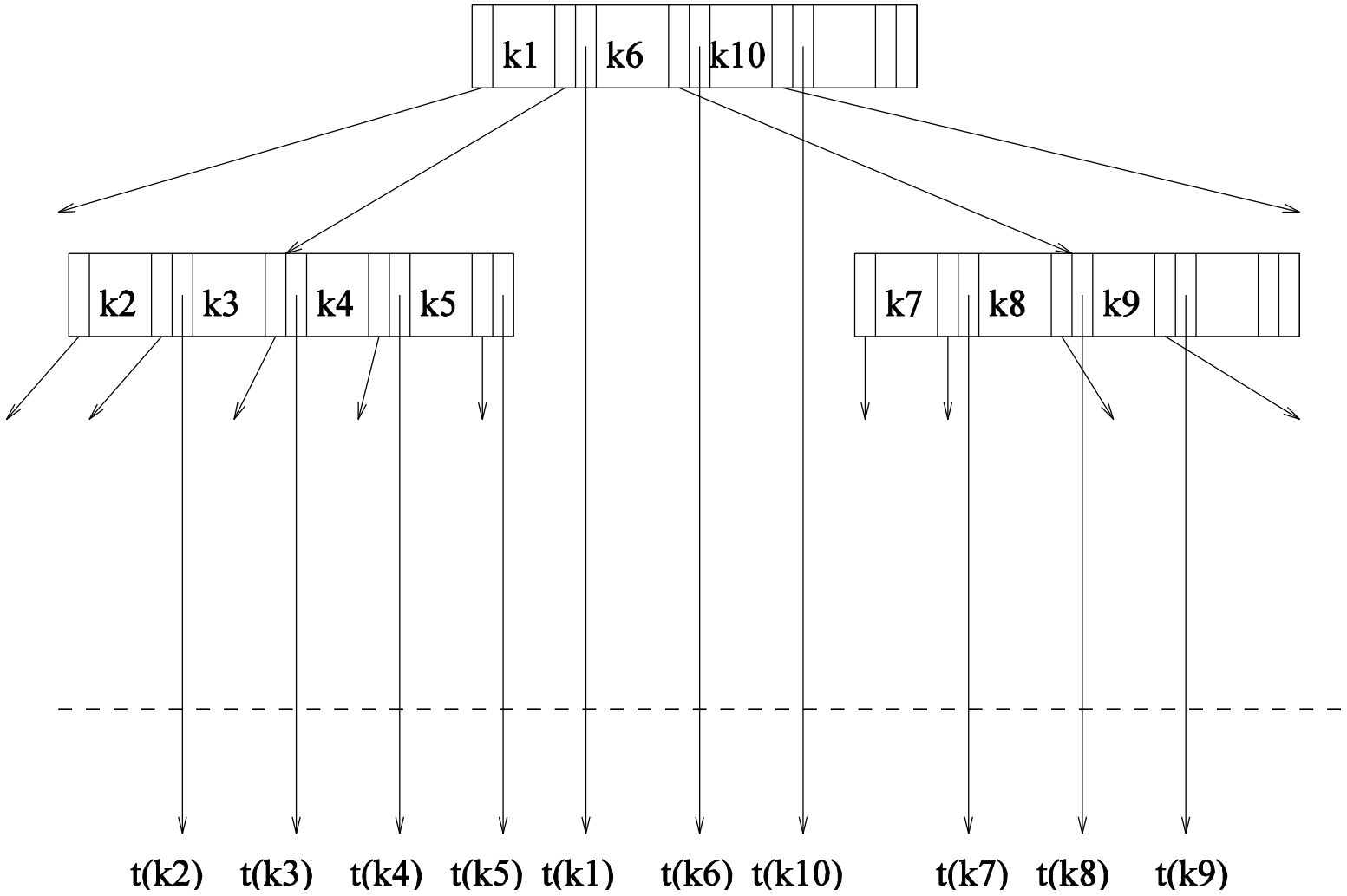
B tree e B+ tree, primari e secondari

- In un B+-tree
  - primario, le ennuple possono essere contenute nelle foglie
  - secondario, le foglie contengono puntatori alle ennuple
- In un B-tree
  - anche i nodi intermedi contengono ennuple (se primari) o puntatori (se secondari)

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)

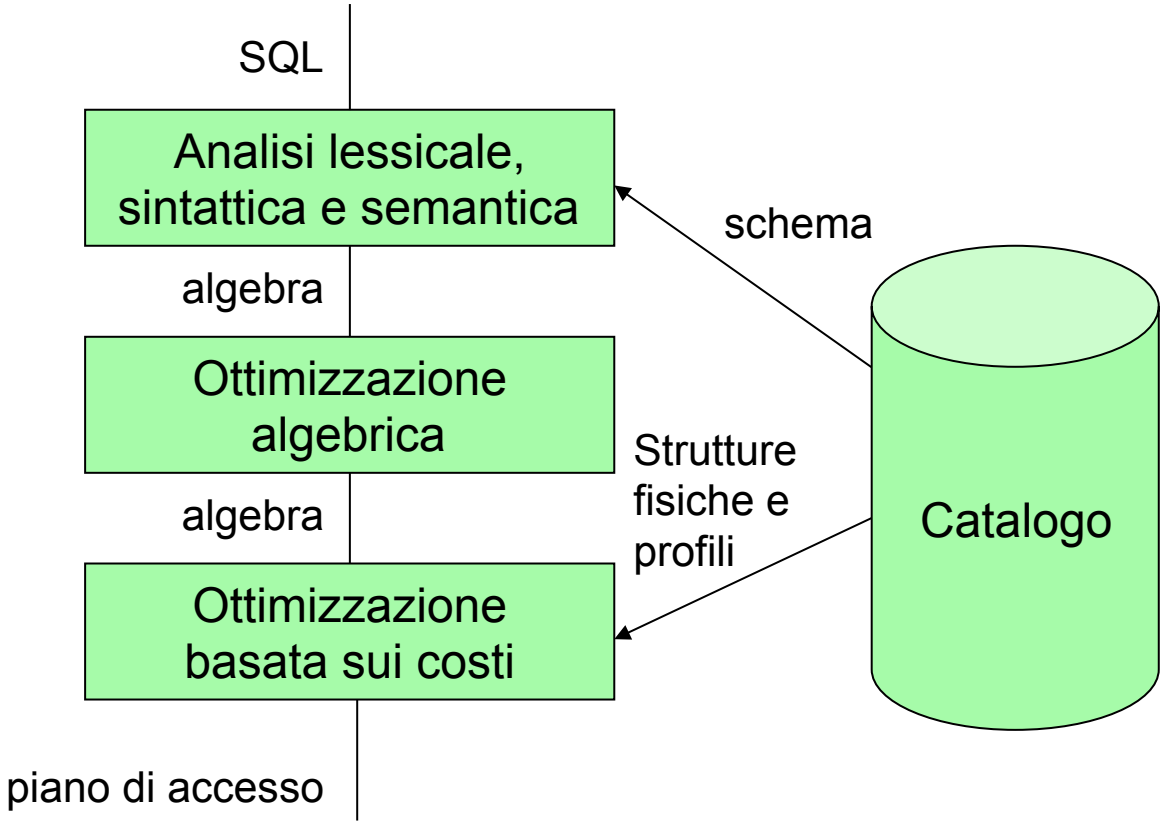
Indici hash

- Strutture secondarie costituite da un file hash con record contenenti
  - pseudochiavi
  - puntatori ai record
- Costo della ricerca:
  - poco più di due accessi: uno (di solito, salvo overflow) all'indice e l'altro al file

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 ennuple
    - poca proceduralità
  - l'ottimizzatore sceglie la strategia realizzativa (di solito fra diverse alternative), a partire dall'istruzione SQL

Il processo di esecuzione delle interrogazioni



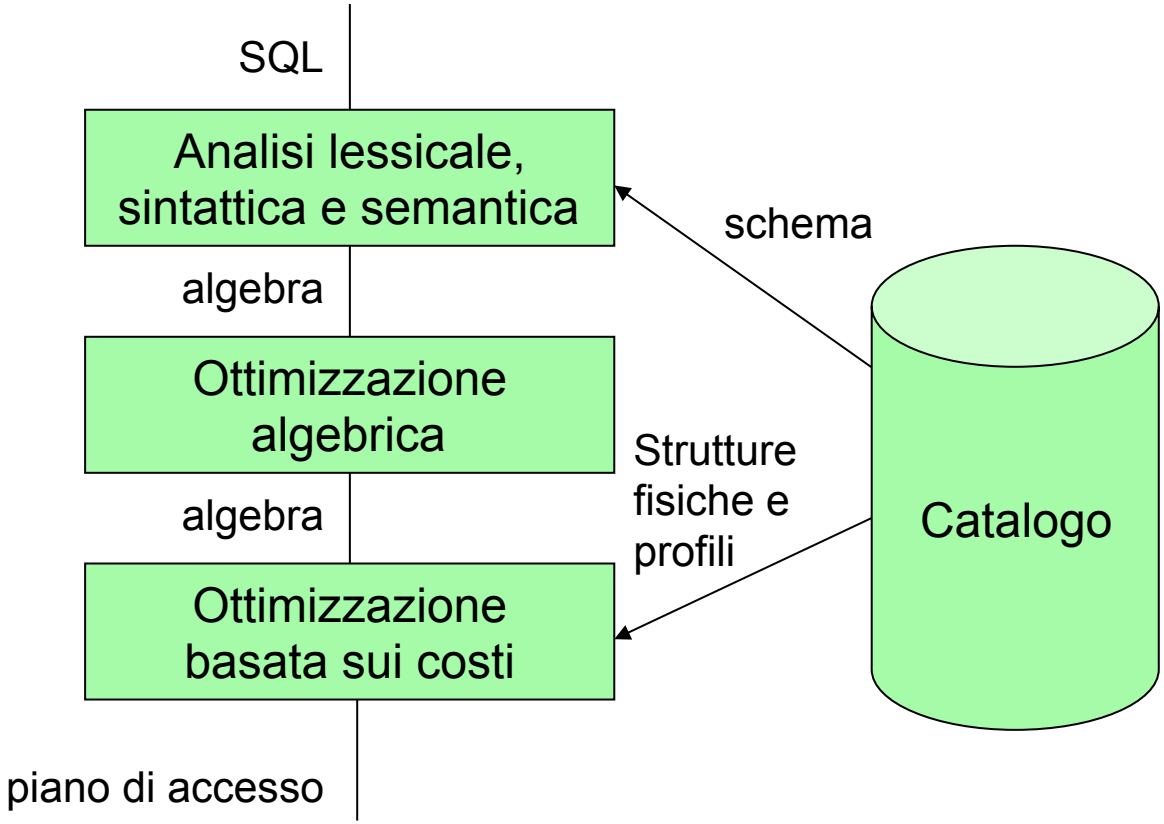
"Profili" delle relazioni

- Informazioni quantitative:
  - cardinalità di ciascuna relazione
  - dimensioni delle tuple
  - dimensioni dei valori
  - numero di valori distinti degli attributi
  - valore minimo e massimo di ciascun attributo
- Sono memorizzate nel "catalogo" e aggiornate con comandi del tipo `update statistics`
- Utilizzate nella fase finale dell'ottimizzazione, per stimare I costi delle seingole operazioni e le dimensioni dei risultati intermedi

Approaches to query compilation

- *Compile and store*: the query is compiled once and carried out many times
  - The internal code is stored in the database, together with an indication of the dependencies of the code on the particular versions of tables and indexes of the database
  - On changes, the compilation of the query is invalidated and repeated
- *Compile and go*: immediate execution, no storage

Il processo di esecuzione delle interrogazioni



Da SQL all'algebra

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

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
```

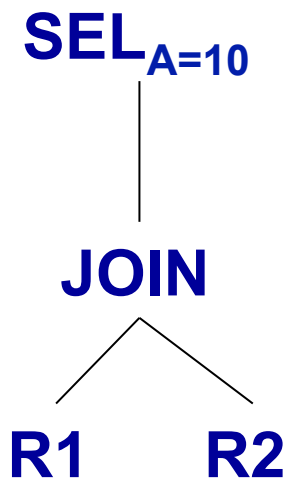
$$\mathbf{PROJ}_{AE}\ (\mathbf{SEL}_{C=D\ AND\ B>100\ AND\ F=G\ AND\ H=7\ AND\ I>2}\ (\mathbf{(R1\ JOIN\ R2)\ JOIN\ R3))}$$

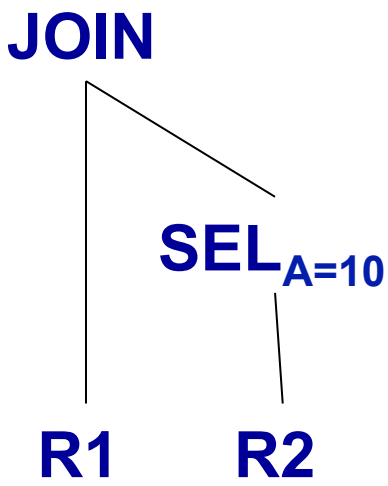
Rappresentazione ad albero

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

Alberi per la rappresentazione di interrogazioni

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





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)

Una procedura euristica di ottimizzazione

- Decomporre le selezioni congiuntive in successive selezioni atomiche
- Anticipare il più possibile le selezioni
- In una sequenza di selezioni, anticipare le più selettive
- Combinare prodotti cartesiani e selezioni per formare join (eventualmente riordinando gli operandi)
- Anticipare il più possibile le proiezioni (anche introducendone di nuove); peraltro questo serve se i risultati intermedi vengono materializzati (vediamo la distinzione fra materializzazione e pipelining più avanti)

Esempio

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

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

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

Esempio, continua

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

- 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

Scan operation

- Performs a sequential access to all the tuples of a table, at the same time executing various operations of an algebraic or extra-algebraic nature:
  - Projection of a set of attributes (no duplicate elimination)
  - Selection on a local predicate (of type: $A_i = v \ldots$)
  - Insertions, deletions, and modifications of the tuples currently accessed during the scan
- Primitives:

  `open`, `next`, `read`, `modify`, `insert`, `delete`, `close`

Accesso diretto

- Può essere eseguito solo se le strutture fisiche lo permettono
  - indici
  - strutture hash

Accesso diretto basato su indice

- Efficiente per interrogazioni (sulla "chiave" dell'indice)
  - "puntuali" ($A_i = v$)
  - su intervallo ($v_1 \leq A_i \leq v_2$)
  - purché l'indice sia selettivo
- 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)
  - Oppure intersezioni sui riferimenti
- Per predicati disgiuntivi:
  - servono indici su tutti, ma conviene usarli solo se molto selettivi e facendo attenzione ai duplicati

Accesso diretto basato su hash

- Efficiente 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

Indici e hash su più campi

- Indice su cognome e nome
  - funziona per accesso diretto su cognome?
  - funziona per accesso diretto su nome?
- Hash su cognome e nome
  - funziona per accesso diretto su cognome?
  - funziona per accesso diretto su nome?

Ordinamento

- Importante, per
  - Produrre risultati ordinati
  - Preparare aggregazioni
  - Preparare i join
  - Eliminazione duplicati
- Utilizza significativamente i buffer

Ordinamento, con buffer

- Esempio:
  - File di 1.000.000.000 di record di 100 byte ciascuno (100GB)
  - Blocchi di 10KB
  - Buffer disponibile di 100MB

  Come possiamo procedere?
  - Merge-sort …

Tradizionale ordinamento di file

- Merge-sort "esterno" (con memoria secondaria e "poca" memoria principale), file di N blocchi:
  - approssimativamente, $\log_2 N$ passi di merge, ognuno dei quali ha un costo pari a $2 \times N$ (si legge e scrive l'intero file); costo complessivo:

$$2 \times N \times \log_2 N$$

Merge sort, con buffer grandi

- Il merge-sort "esterno" richiede:
  - $\log_2 N$ passi di merge,
  - ognuno di costo pari a $2 \times N$
- Costo complessivo: $2 \times N \times \log_2 N$
- Se abbiamo molta memoria, possiamo migliorare riducendo il secondo termine (cioè il numero di passi di merge) e non il primo (il costo del merge), che non è riducibile:
  - inizialmente, invece di ordinare singoli blocchi, ordiniamo porzioni di file che entrano in memoria
  - poi, invece di fondere due porzioni, ne fondiamo (come estremo, forse non praticabile) tante quanti sono le pagine del buffer (o quasi); in pratica, questo porta quasi sempre a un solo passo di merge o al massimo a due

Merge sort, quanti passi di merge?

- Con P pagine di buffer, possiamo
  - ordinare P blocchi
  - fondere (merge) P porzioni ordinate (dette run)
- Quindi, con P buffer possiamo ordinare con un solo passo di merge (preceduto da ordinamenti dei run di P blocchi) un file di P^2 blocchi:
  1. ordinamento di P run ognuno di P blocchi
  2. fusione di P run
- Sempre con P buffer possiamo ordinare con due passi di merge un file di P^3 blocchi
  1. ordinamento di P run ognuno di P blocchi
  2. fusione di P run (di P blocchi ciascuno) alla volta a formare run di P^2 blocchi
  3. fusione di P run (di P^2 blocchi ciascuno) alla volta a formare il file ordinato di P^3 blocchi

Merge sort, quanti passi di merge? (2)

- Con P buffer, possiamo
  - ordinare direttamente un file di P blocchi (una passata)
  - con un primo ordinamento e un merge (due passate), ordinare un file di P^2 blocchi
  - con un primo ordinamento e due merge (tre passate), ordinare un file di P^3 blocchi
  - …
  - con i passate, possiamo ordinare un file di P^i blocchi
- Dato un file di N blocchi, lo possiamo ordinare in i passate se
  - $P^i \geq N$ cioè se P è ≥ della radice i-esima di N

  Quindi il numero di passate necessario è il più piccolo i per cui risulta P ≥ della radice i-esima di N e il numero di buffer da utilizzare è proprio la radice i-esima di N (più precisamente, la sua parte intera superiore)

Esercizio

- Compito d'esame del 26/02/2013, esercizio 5
  - 16 blocchi, 8 buffer
    - una passata non basta (8 è minore di 16)
    - due bastano (8 è maggiore di 4) e si possono usare 4 buffer

Join

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

Nested-loop

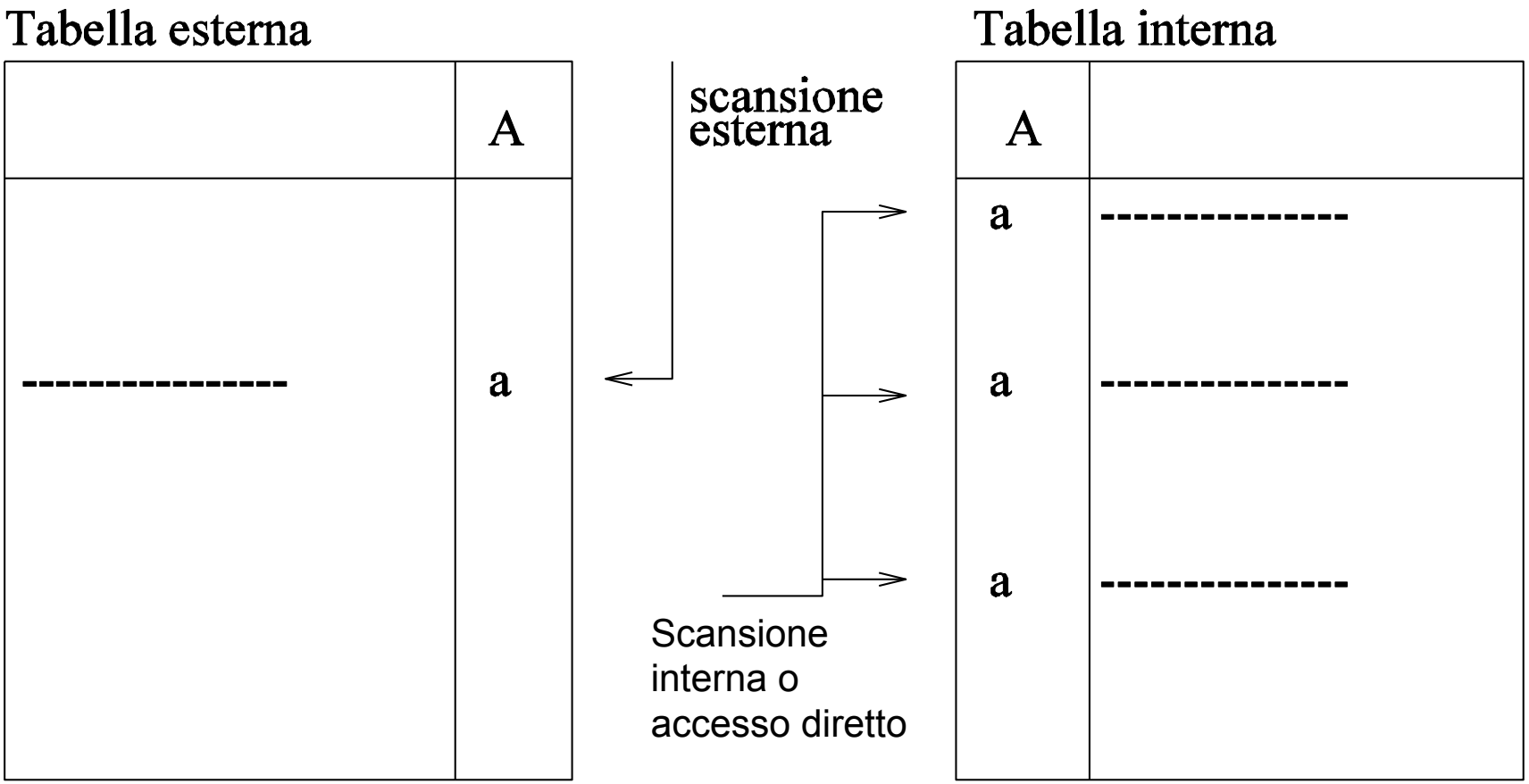


Tabella esterna
A
a
Tabella interna
A
a
a
a
scansione esterna
Scansione interna o accesso diretto

Nested-loop, costi (con buffer!)

- Join, con nested loop
  - R1 1000 blocchi
  - R2 500 blocchi
  - 101 pagine a disposizione nel buffer

Nested-loop, costi (2)

- Join, con nested loop, senza indici;
  - relazioni R_1 e R_2 di N_1 e N_2 blocchi
  - l'algoritmo base richiede la scansione di R_1 e, per ciascun blocco di essa, la scansione di R_2; quindi il costo (numero di accessi a memoria secondaria) può essere stimato pari a:

$$N_1 + N_1 \times N_2$$

  - avendo a disposizione più pagine di buffer, si possono usare per caricare più blocchi di R_1, riducendo di conseguenza il numero di scansioni di R_2 (le ennuple di R_2 durante la scansione possono essere confrontate con quelle in tutti i blocchi di R_1 nel buffer); con B pagine di buffer dedicate a blocchi di R_1 il costo diventa

$$N_1 + (N_1/B \times N_2)$$

  - Nell'esempio, si passa da circa 500.000 accessi a circa 6.000

Nested-loop, costi (3)

- Join, con nested loop, con indice
  - relazioni R_1 e R_2 di L_1 e L_2 ennuple e N_1 e N_2 blocchi e indice su R_2 di profondità I_2
  - l'algoritmo base richiede la scansione di R_1 e, per ciascun record di essa, l'accesso diretto a R_2; il costo può essere stimato pari a:

$$N_1 + L_1 \times (1 + I_2)$$

  - avendo a disposizione più pagine di buffer, si può pensare che esse contengano i livelli più alti dell'indice, ad esempio due o tre

Merge-scan

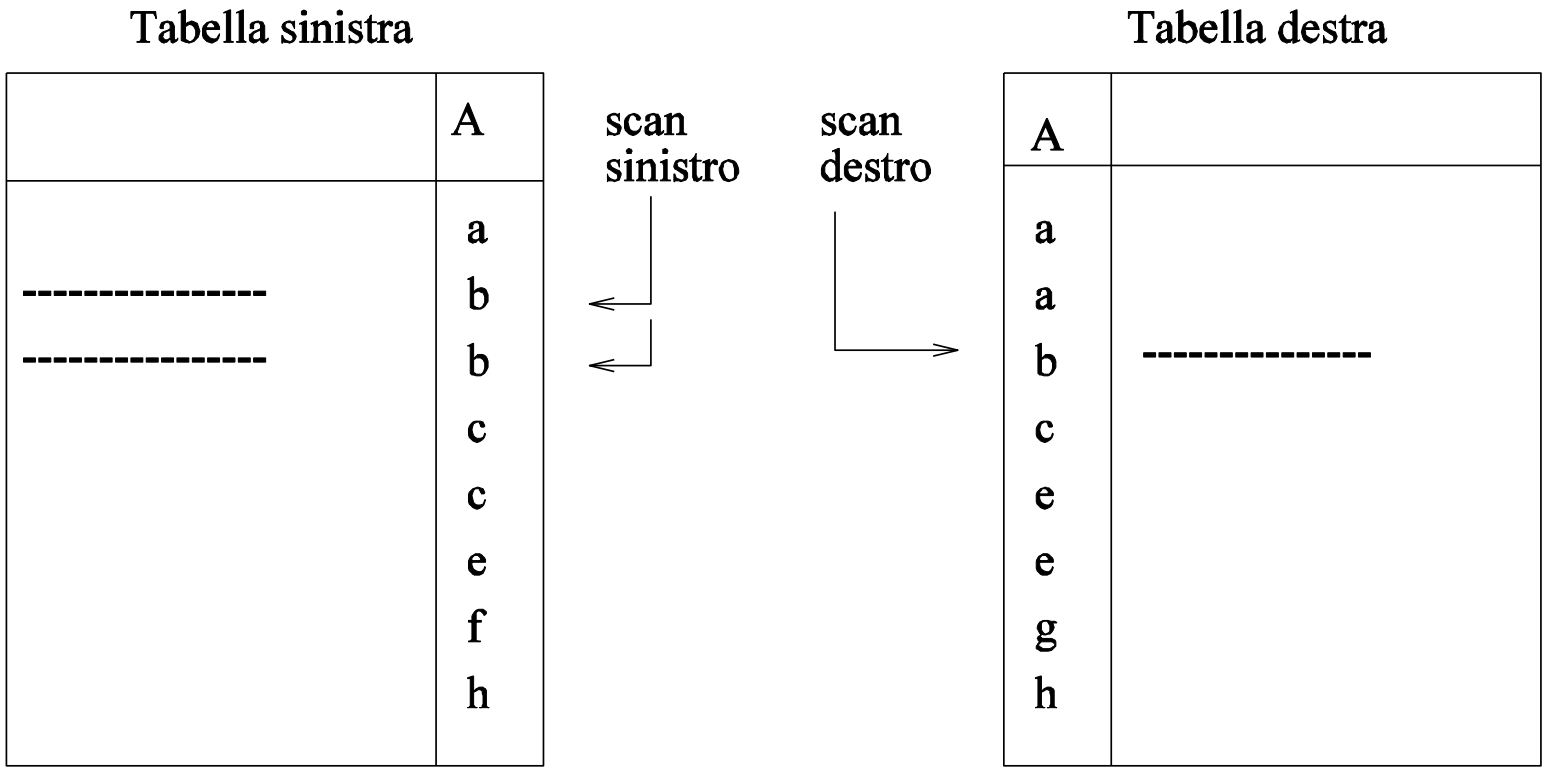
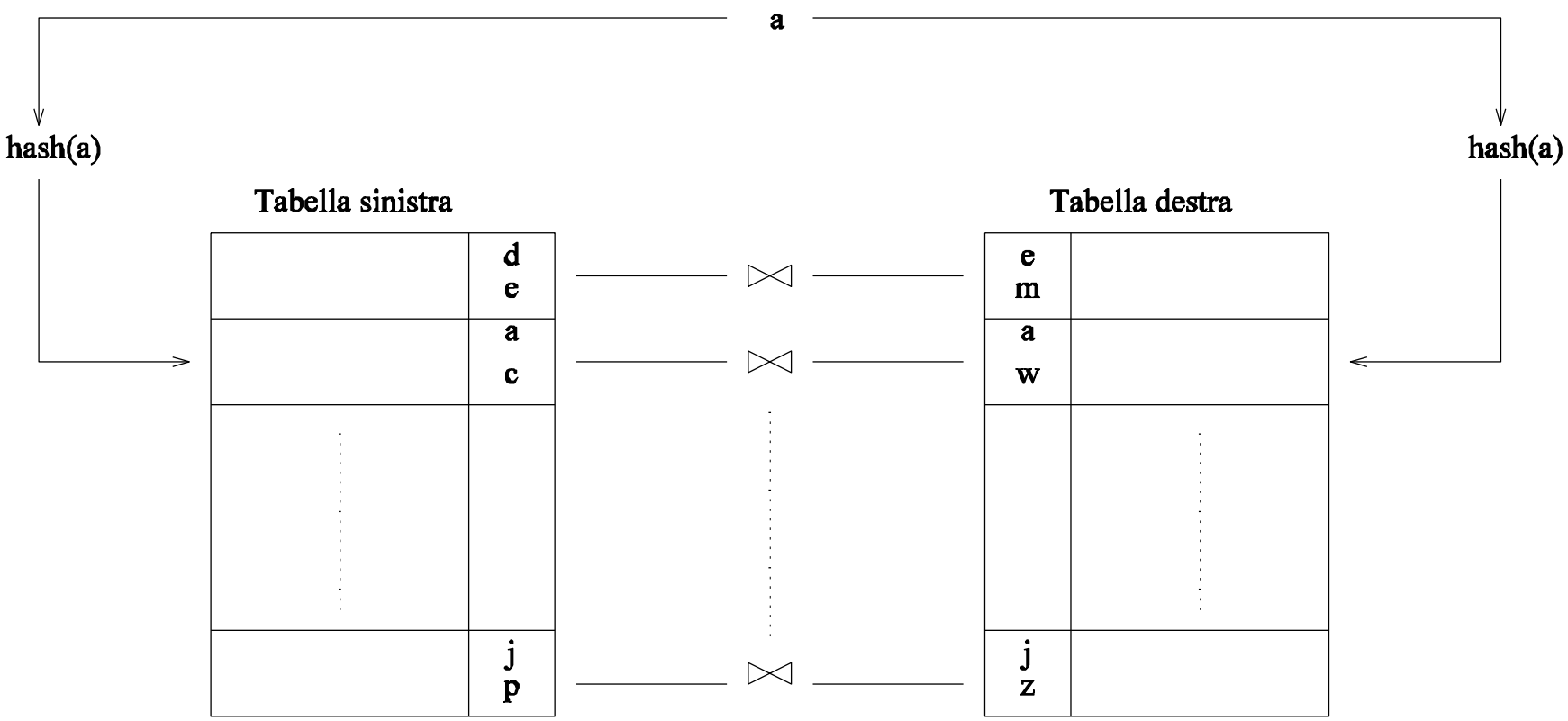


Tabella sinistra
A
a
b
b
c
c
e
f
h
scan sinistro
scan destro
Tabella destra
A
a
a
b
c
e
e
g
h

Merge-scan, costi

- Approssimativamente (N1 e N2 numero di blocchi dei due file):
  - Se i file sono fisicamente ordinati:
    - N1 + N2
  - Se i file sono disordinati, ma ci sono indici:
    - L1 + L2 (se tutti i record partecipano; se invece il join è selettivo il costo diminuisce) più il costo della scansione delle foglie degli indici

Hash join



- Stessa funzione hash applicata ai campi di join delle due relazioni
- Spesso, una viene effettivamente memorizzata e l'altra no

Hash join, costi ed esercizio

- Compito d'esame del 25/09/2012, esercizio 7

Hash-join, costi

- Con un approccio semplice:
  - Ciascun file viene letto sequenzialmente (costo Bi) e poi i record vengono memorizzati secondo la funzione hash (costo Li); poi si rilegge il tutto per "accoppiare"
    - B1 + L1 + B2 + L2 + B1 + B2
- Primo miglioramento:
  - non è necessario memorizzare il secondo file (si valuta la funzione hash e si cerca nella struttura temporanea del primo):
    - B1 + L1 + B2 + L2
- Se le pseudochiavi sono brevi rispetto ai record e il join è selettivo, si può migliorare costruendo strutture temporanee (molto più piccole) che sono indici hash e accedendo ai file solo quando i record partecipano al join

Hash-join, costi in realtà

- Sfruttando i buffer, si può usare una funzione hash con numero di valori diversi paragonabile al numero di pagine di buffer P a disposizione e così si può fare tutto in due passate di lettura più una scrittura:
  - Ciascun file viene letto e riorganizzato in P liste di blocchi (costo Bi per la lettura e Bi per la riscrittura, numero di blocchi e non di record perché si scrivono blocchi completi) poi, per ciascun valore della funzione hash si confrontano le liste omologhe (costo B1 + B2 per la lettura).
  - L'algoritmo non richiede altri accessi se le singole partizioni di uno dei file entrano in memoria, cioè se min(B1,B2)/P < P-2 cioè se P^2 > min(B1,B2). In tal caso il costo è 3(B1+B2) + il numero di blocchi del risultato (se va scritto)

Confronto costi

- Risultato piccolo (molta selettività):
  - spesso conviene nested loop con indice
- Operandi di dimensione paragonabile:
  - Mergejoin (in teoria anche HashJoin, ma non è garantita la distribuzione uniforme dei valori)
- Operandi grandi, ma di dimensioni diverse:
  - Hashjoin (le porzioni della tabella più piccola possono entrare in memoria)

Pipelining vs materializzazione

- Due alternative per le interrogazioni nei sottoalberi:
  - **pipelining**: le ennuple sono "utilizzate" dal nodo superiore a mano a mano che vengono prodotte (anzi, vengono prodotte a richiesta):
    - vantaggio: non dovendo salvare i risultati intermedi, riduce i costi di I/O
    - svantaggio: se i risultati intermedi vengono riutilizzati più volte, è necessario ricalcolarli (ad esempio, il ciclo interno di un nested loop)
  - **materializzazione**: l'intero risultato intermedio viene prodotto e memorizzato, prima di essere utilizzato

Ottimizzazione basata sui costi

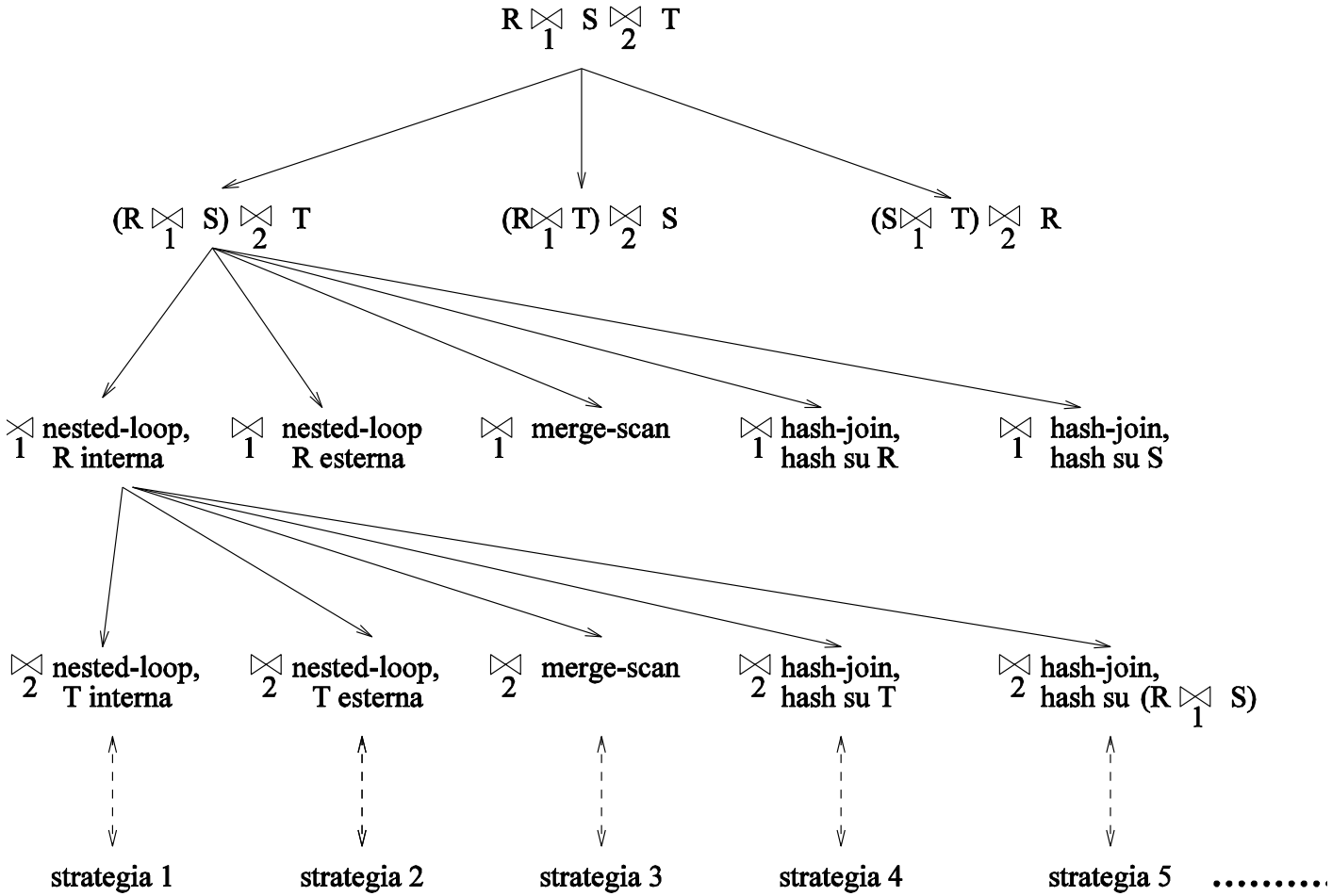
- Un problema articolato, con scelte relative a:
  - operazioni da eseguire (es.: scansione o accesso diretto?)
  - ordine delle operazioni (es. join di tre relazioni; ordine?)
  - i dettagli del metodo (es.: quale metodo di join)
- Architetture parallele e distribuite aprono ulteriori gradi di libertà

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 necessariamente quella "ottima"

Un albero di decisione



Esempio reale (anche se piccolo) con Postgres

```
create table r1 ( A numeric not null primary key,
                  B numeric,
                  C numeric);
create table r2 ( D numeric not null primary key,
                  E numeric,
                  F numeric);
create table r3 ( G numeric not null primary key,
                  H numeric,
                  I numeric);
```

6-10.000 insert per ciascuna, pochi valori su B, E, H molti su C,F,I

```
create index i12 on r1(B)
create index i13 on r1(C)
vacuum analyze
```

In DB2: runstats on table atzeni.r1 for indexes all

Esempio reale , segue

set search_path to *nomeschema* (proveindici)

```
select * from r1 where C>2 AND C<8

select * from r1 where C>2 AND C<30

SELECT    A , E
FROM      R1, R3, R2
WHERE     C=D AND B<40 AND F=G AND A=7 AND I>2

SELECT    A , E
FROM      R1, R3, R2
WHERE     C=D AND B<40 AND F=G AND H=7 AND I>2
```

Progettazione fisica

- La fase finale del processo di progettazione di basi di dati
- input
  - lo schema logico e informazioni sul carico applicativo
- output
  - schema fisico, costituito dalle definizione delle relazioni con le relative strutture fisiche (e molti parametri, spesso legati allo specifico DBMS)

Strutture fisiche nei DBMS relazionali

- Struttura primaria:
  - disordinata (heap, "unclustered")
  - ordinata ("clustered"), anche su una pseudochiave
  - hash ("clustered"), anche su una pseudochiave, senza ordinamento
  - clustering di più relazioni
- Indici (densi/sparsi, semplici/composti):
  - ISAM (statico), di solito su struttura ordinata
  - B-tree (dinamico)
  - Indici hash (secondario, poco dinamico)

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

Strutture fisiche in alcuni DBMS, 2

- Ingres (anni fa):
  - file heap, hash, ISAM (ciascuno anche compresso)
  - indici secondari
- Informix (per DOS, 1994):
  - file heap
  - indici secondari (e primari [cluster] ma non mantenuti)

Definizione degli indici SQL

- Non è standard, ma presente in forma simile nei vari DBMS
  - `create` [`unique`] `index` *IndexName* `on` *TableName*(*AttributeLis*t)
  - `drop index` *IndexName*

Progettazione fisica nel modello relazionale

- La caratteristica comune dei DBMS relazionali è la disponibilità degli indici:
  - la progettazione fisica spesso coincide con la scelta degli indici (oltre ai parametri strettamente dipendenti dal DBMS)
- Le chiavi (primarie) delle relazioni sono di solito coinvolte in selezioni e join: molti sistemi prevedono (oppure suggeriscono) di definire indici sulle chiavi primarie
- Altri indici vengono definiti con riferimento ad altre selezioni o join "importanti"
- Se le prestazioni sono insoddisfacenti, si "tara" il sistema aggiungendo o eliminando indici
- È utile verificare se e come gli indici sono utilizzati con il comando SQL `show plan` oppure `explain`

Progettazione fisica: euristiche suggerite da Informix

- Non creare indici su relazioni piccole (<200 ennuple)
- non creare indici su campi con pochi valori (se proprio servono, che siano primari)
- creare indici su campi con selezioni
- per i join: creare indici sulla relazione più grande

Scelta della struttura secondo Shasha

D. Shasha. Database Tuning: a principled approach. Prentice-Hall, 1992

D. Shasha, P. Bonnet Database Tuning: Principles, Experiments, and Troubleshooting Techniques Morgan Kaufmann 2002