Corso di Laurea
in
Ingegneria Informatica
"*Basi di dati*"
a.a. 2019-2020

Docente: Gigliola Vaglini
Docente laboratorio SQL: Francesco Pistolesi

1

1

# Lezione 9

# Organizzazione fisica e gestione della memoria

2

2

# Tecnologia delle BD

- Il DBMS è una "scatola nera"
- Perché aprirla?
  - capire come funziona può essere utile per un migliore utilizzo

3

3

## DataBase Management System — DBMS

Sistema per gestire collezioni di dati:

- **grandi**
- **persistenti**
- **condivise**, garantendo **affidabilità** e privatezza .

- In più 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).

4

4

# Le basi di dati sono grandi e persistenti

- La persistenza richiede la gestione della memoria secondaria
- La grandezza richiede che tale gestione sia sofisticata
- Gli utenti vedono il modello logico, ma le strutture logiche debbono essere gestite efficientemente in memoria secondaria:
  - servono strutture fisiche opportune

5

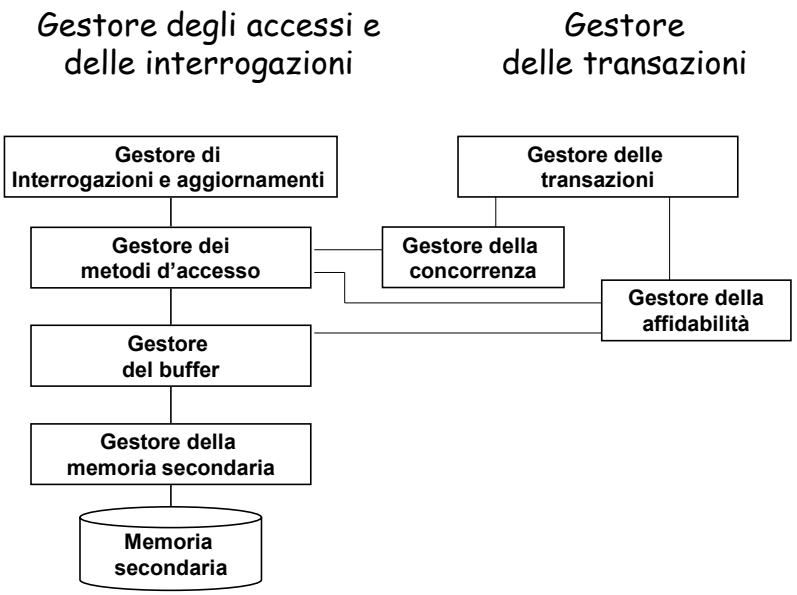
5

# Le basi di dati sono affidabili e condivise

- Le basi di dati debbono essere preservate anche in presenza di malfunzionamenti
- L'affidabilità è impegnativa per via degli aggiornamenti frequenti e della necessità di gestire il buffer
- Una base di dati è una risorsa **condivisa** fra le varie applicazioni
  - Attività diverse su dati in parte condivisi:
    - meccanismi di autorizzazione
  - Attività multi-utente su dati condivisi:
    - controllo della **concorrenza**

6

6




7

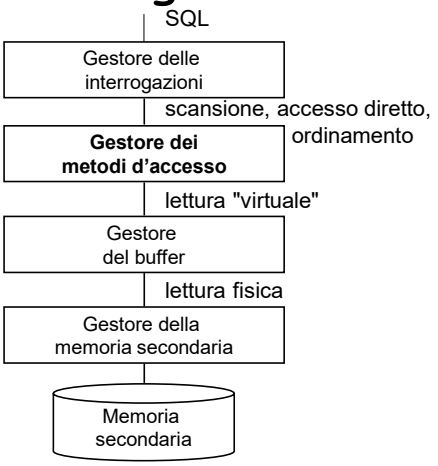
7

# Tecnologia delle basi di dati

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

8

# Gestore degli accessi e delle interrogazioni




9

# DBMS e file system

- Il file system è il componente del sistema operativo che gestisce la memoria secondaria
- I DBMS ne utilizzano le funzionalità per creare ed eliminare file e per leggere e scrivere singoli blocchi o sequenze di blocchi contigui.
- Il DBMS gestisce i 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.

10

10

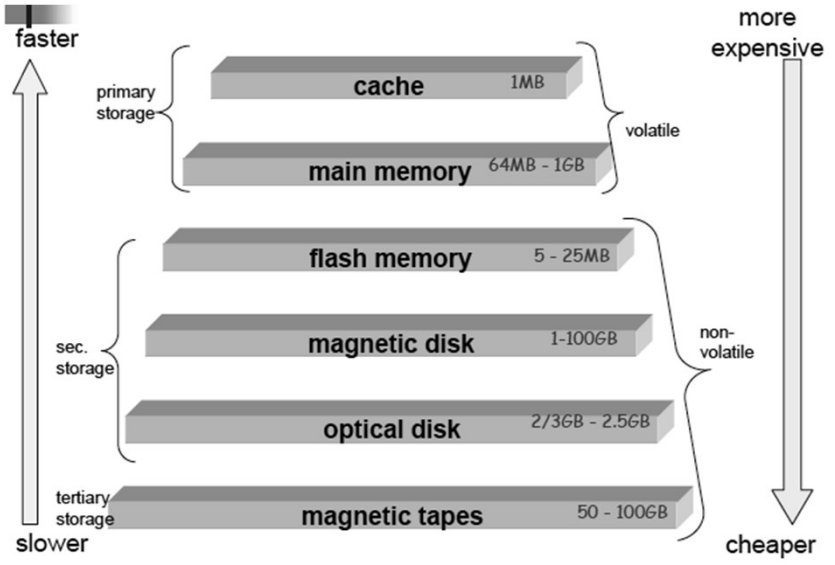
# Quando le basi di dati vengono interrogate …

- I programmi possono fare riferimento solo a dati in memoria principale, quindi i dati in memoria secondaria possono essere utilizzati solo se prima trasferiti in memoria principale (questo spiega i termini "principale" e "secondaria) serve un'interazione fra memoria principale e secondaria che limiti il più possibile gli accessi alla secondaria

11

11

# La gerarchia di memoria




12

12

## Prestazioni di una memoria

- Dato un indirizzo di accesso, le prestazioni di memoria si misurano in termini della somma tra il tempo che la testina impiega per raggiungere la traccia di interesse, la latenza (tempo per accedere al primo byte del blocco di interesse) e il tempo di trasferimento (tempo necessario a muovere tutti i dati)

13

13

## Memoria principale e secondaria

- 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 10 ms
- Il tempo 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

14

14

# Memoria principale e secondaria

- I dispositivi di memoria secondaria sono organizzati in **blocchi** di lunghezza (di solito) **fissa** (ordine di grandezza: alcuni KB)
- I record dei file sono memorizzati nei blocchi
- Le uniche operazioni sono la lettura e la scrittura dei dati di un blocco
- Accessi a blocchi "vicini" costano meno (contiguità): tempo di posizionamento + latenza=0
- La memoria principale è organizzata in pagine

15

15

# 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)
  - Se assumiamo che coincidano pagina e blocco, il caricamento di una pagina del buffer richiede una lettura in memoria secondaria, mentre salvare una pagina corrisponde ad una scrittura

16

16

## Dati gestiti dal buffer manager

- Il buffer
- Una directory 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 bit che indica se la pagina è "sporca", cioè se è stata modificata

17

17

## Il funzionamento del buffer manager

- 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

18

18

# Funzioni del buffer manager

- riceve richieste di lettura e scrittura (di pagine) dalle transazioni
- le esegue accedendo alla memoria secondaria solo quando indispensabile e utilizzando invece il buffer quando possibile

19



# Interfaccia

- esegue le primitive
  - ***fix***: richiesta di una pagina; richiede una lettura solo se la pagina non è nel buffer (incrementa il contatore associato alla pagina offerta dal buffer manager)
  - ***setDirty***: comunica 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)

20

# 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, vi inserisce i dati letti dalla memoria secondaria e ne restituisce l'indirizzo
    - altrimenti, due politiche alternative
      - "**steal**": selezione di una "vittima", pagina occupata del buffer; i dati della vittima sono scritti in memoria secondaria; vengono letti i dati di interesse dalla memoria secondaria e si restituisce l'indirizzo
      - "**no-steal**": l'operazione viene posta in attesa





# 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) in modo da accorparla ad altre scritture

# Quindi

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





# Tuple e blocchi

- I file sono logicamente organizzati in record
- I record sono mappati nei blocchi di memoria secondaria
- Le tuple di una relazione (record di file) stanno in blocchi contigui
- A volte in un blocco ci sono tuple di relazioni diverse ma correlate (i join sono favoriti)

# Blocchi e tuple

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

25



# Organizzazione delle tuple

- Ci sono varie alternative, anche legate ai metodi di accesso; in genere sono sistemate sequenzialmente nei file, inoltre:
  - se la lunghezza delle tuple è fissa, la struttura può essere semplificata
  - alcuni sistemi possono spezzare le tuple su più blocchi (necessario per tuple grandi)

26

# 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)
  - non utilizzato ("unspanned")

27



# Esercizio

- Calcolare il fattore di blocco e il numero di blocchi occupati da una relazione contenente T = 500000 tuple di lunghezza fissa pari a L = 100 byte in un sistema con blocchi di dimensione pari a B = 1 kilobyte.

28

## Soluzione

- $N_B = D_T/B$ $\quad D_T = T*L$
- $N_B = 50000000/1024$
- $F_B = B/L$
  $F_B = 1024/100$

29

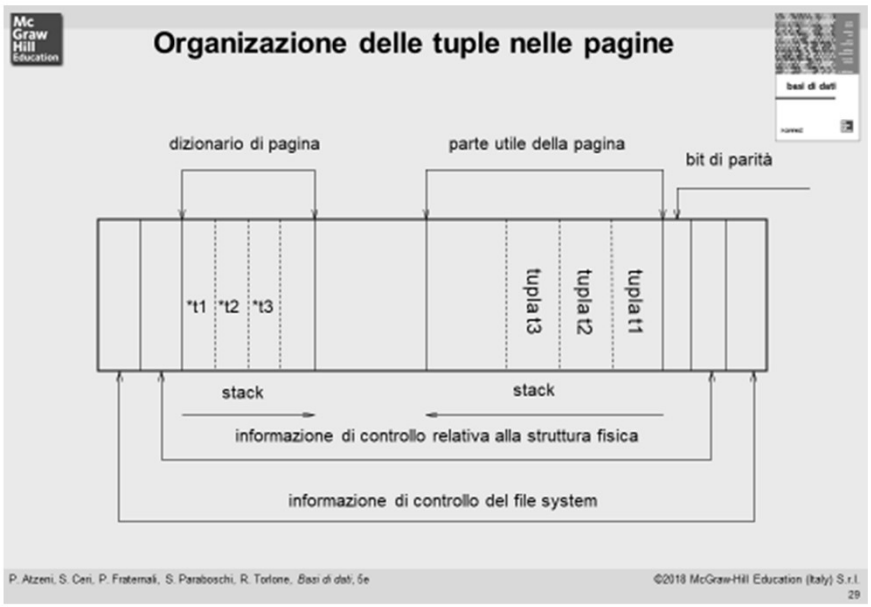
29

## Considerando la dimensione della pagina uguale a quella del blocco..

- Ad esempio nel buffer
- L'organizzazione delle tuple nella pagina è simile

30

30




31

31

# Operazioni sulla pagina

- Inserimento/modifica di una tupla
  - Può richiedere l'allocazione di nuovo spazio
- Cancellazione
- Accesso ad una tupla o ad un campo di una tupla

32

32

## Il file system usato dal DBMS

- I DBMS tramite i sistemi operativi gestiscono il file system per memorizzare il database
- Si occupano quindi di fare l'accesso alle strutture fisiche che contengono i dati

33



## Strutture primarie e secondarie

- Le tuple organizzate all'interno dei blocchi dei file costituiscono le
  - Strutture primarie, cioè quelle che contengono propriamente i dati
- Esistono anche blocchi contenenti
  - Strutture secondarie, che sono quelle che favoriscono l'accesso ai dati senza contenerli

34

## Strutture primarie

- Possono avere una organizzazione detta sequenziale
- L'organizzazione sequenziale può essere
  - **seriale**: ordinamento fisico ma non logico
  - **ordinata**: con ordinamento delle tuple coerente con quello di un campo
  - array: posizione individuate tramite indici

35

35

## Organizzazione seriale

- Chiamata anche:
  - "Entry sequenced"
  - file heap
  - file disordinato
- Gli inserimenti vengono effettuati
  - in coda (con riorganizzazioni periodiche)
  - al posto di record cancellati

36

36

## Strutture sequenziali ordinate

- Le strutture ordinate permettono ricerche binarie
- Il problema è mantenere l'ordinamento

37

37

## Con accesso calcolato

- I file hash permettono un accesso diretto molto efficiente
  - La tecnica si basa su quella utilizzata per le tavole hash in memoria centrale

38

38

# Remind: 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 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)

39

39

# Tavola hash

- senza sprecare spazio
  - **funzione hash:**
    - associa ad ogni valore della chiave un "indirizzo", in uno spazio di dimensione leggermente superiore rispetto a quello strettamente necessario
    - poiché il numero di possibili chiavi è molto maggiore del numero di possibili 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)

40

40

# Un esempio

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

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

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

41

41

# Risoluzione 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

42

42

# Hash su file

- L'idea è la stessa, ma si sfrutta l'organizzazione in blocchi e il fatto che l'accesso è al blocco
- In questo modo si "ammortizzano" le probabilità di collisione

43

43

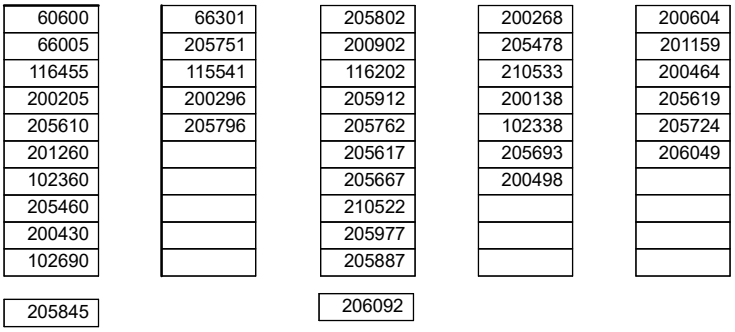
# Un esempio

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2
- file hash con fattore di blocco 10;
  5 blocchi con 10 posizioni ciascuno, la funzione hash restituisce un indirizzo tra 0 e 4:
  - due soli overflow!

44

44

# Un file hash




45

45

- Il record che confligge si memorizza nel record successivo
- Quando si tratterà di trovarlo si dovranno fare due accessi a blocchi
- In tutti gli altri casi si dovrà fare un solo accesso.

46

46

# Strutture ad albero (non sequenziali)

- Dette anche indici
- Strutture basate sull'uso di puntatori ma l'accesso è in base ai valori di uno o più campi
- Si utilizzano sia come strutture primarie che secondarie

47



# Strutture ad albero

- Indice:
  - struttura per l'accesso (efficiente) ai record sulla base dei valori di un campo (o di una "concatenazione di campi") detto chiave (o, meglio, pseudochiave, perché non è necessariamente identificante);
- 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
  - Ad es., l'indice analitico di un libro: lista di coppie (termine,pagina), ordinata alfabeticamente sui termini, posta in fondo al libro e separabile da esso

48

# Tipi di indice

- indice primario:
  - su un campo sul cui ordinamento è basata la memorizzazione
- indice secondario
  - su un campo con ordinamento diverso da quello dell'ordinamento di memorizzazione
- indice denso
  - contiene un record per ciascun record del file
- indice sparso
  - contiene un numero di record inferiore rispetto a quelli del file

49

49

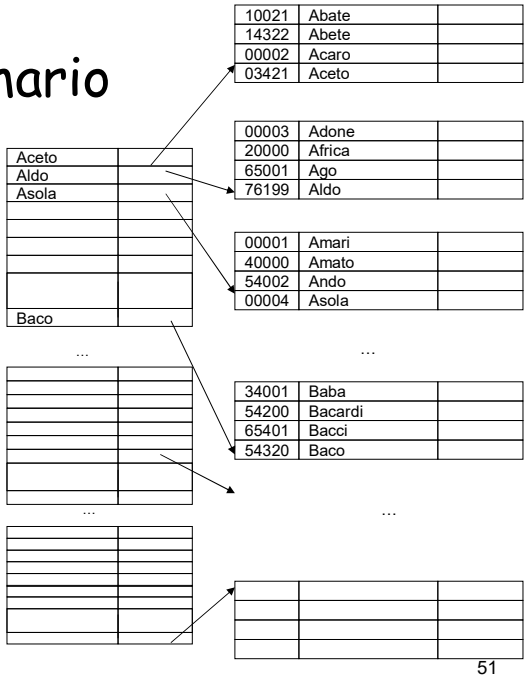
# Tipi di indice, commenti

- Un indice primario può essere sparso (non tutti i valori della chiave compaiono nell'indice)
  - Esempio, sempre rispetto ad un libro
    - indice generale (cap.1, sez.1)
- Gli indici secondari sono densi perché tutti i valori della chiave devono essere raggiungibili
- 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

50

50

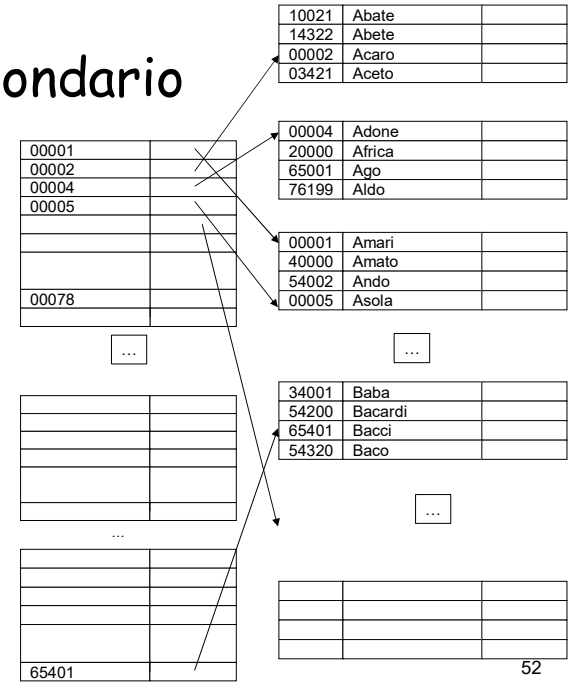
# Indice primario




51

51

# Indice secondario




52

52

# Caratteristiche degli indici

- Accesso diretto (sulla chiave) efficiente
- Scansione sequenziale ordinata efficiente
- 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

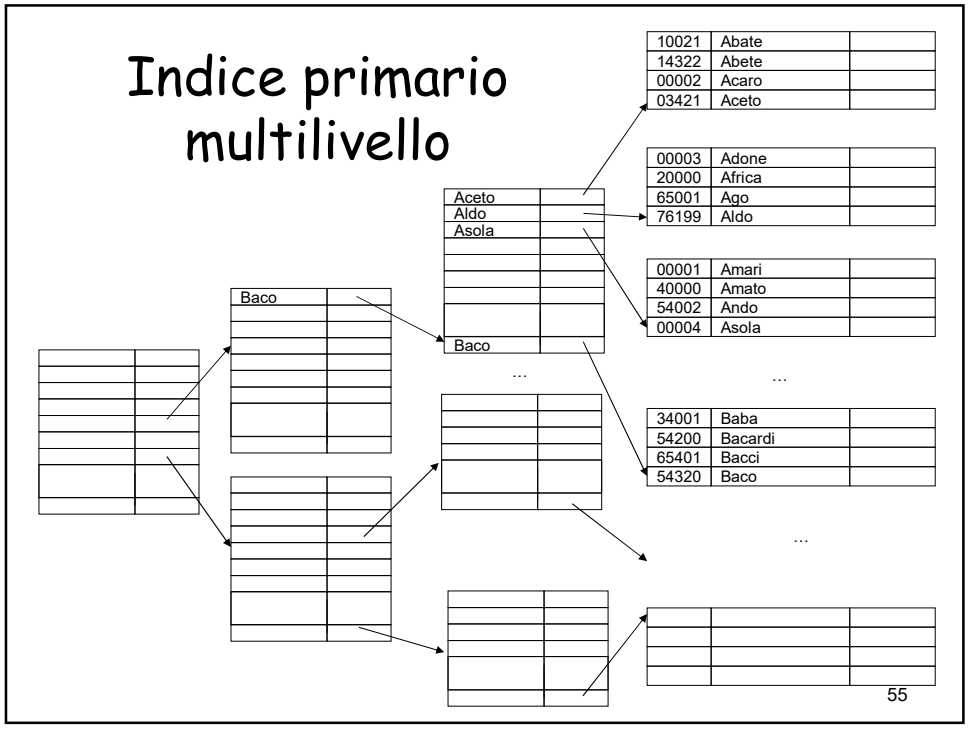
53



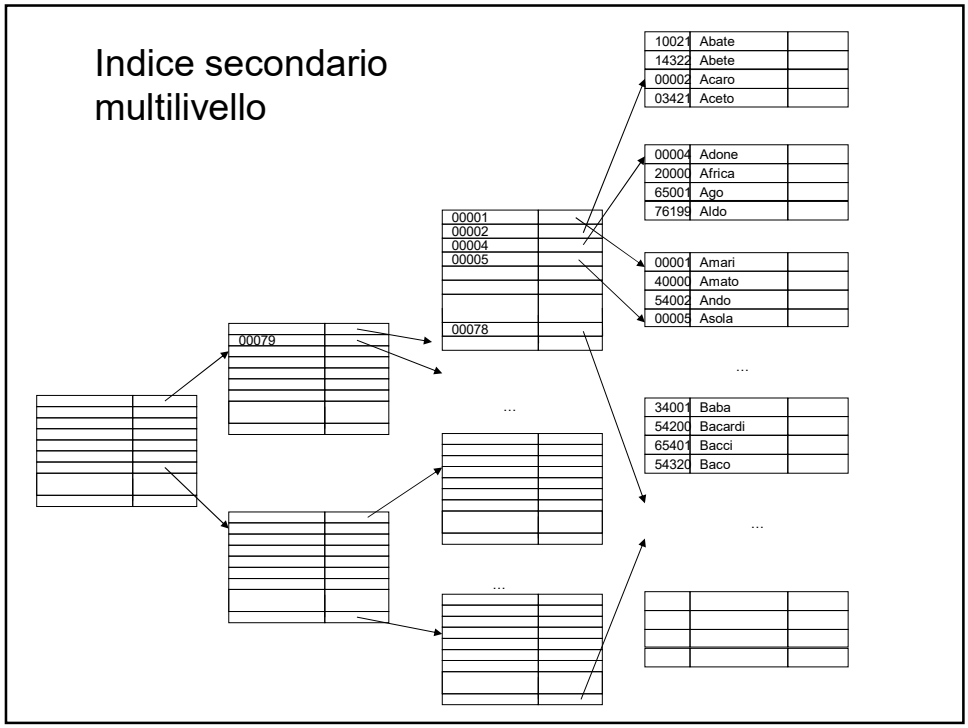
# Indici multilivello

- 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

54


Indice primario multilivello
10021 Abate
14322 Abete
00002 Acaro
03421 Aceto
00003 Adone
20000 Africa
65001 Ago
76199 Aldo
00001 Amari
40000 Amato
54002 Ando
00004 Asola
Aceto
Aldo
Asola
Baco
Baco
34001 Baba
54200 Bacardi
65401 Bacci
54320 Baco
55

55


Indice secondario multilivello
10021 Abate
14322 Abete
00002 Acaro
03421 Aceto
00004 Adone
20000 Africa
65001 Ago
76199 Aldo
00001 Amari
40000 Amato
54002 Ando
00005 Asola
34001 Baba
54200 Bacardi
65401 Bacci
54320 Baco
00001
00002
00004
00005
00078
00079

56

# Indici, problemi

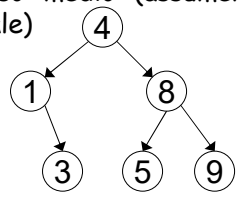
- Le strutture di indice basate su strutture ordinate sono poco flessibili in presenza di elevata dinamicità
- Gli indici utilizzati dai DBMS sono in generale
  - indici dinamici multilivello
  - Vengono memorizzati e gestiti come B-tree (intuitivamente: alberi di ricerca bilanciati)
    - Alberi binari di ricerca
    - Alberi n-ari di ricerca
    - Alberi n-ari di ricerca bilanciati

57

57

# 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)




58

58

# Albero di ricerca di ordine P

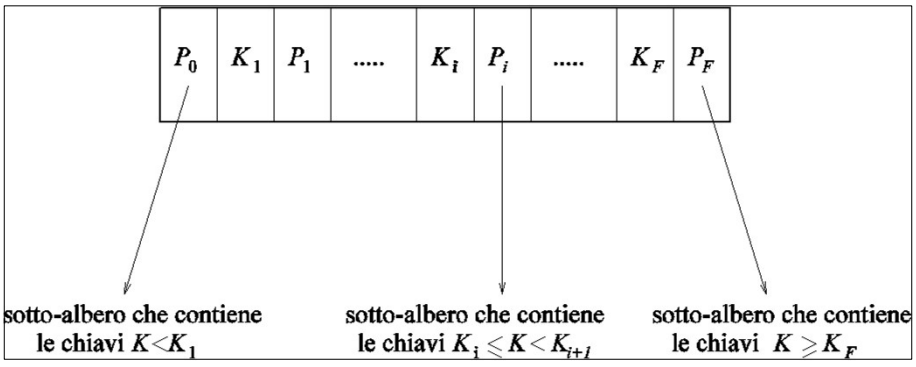
- Ogni nodo ha (fino a) P figli e (fino a) P-1 etichette, ordinate
- Nell'i-esimo sottoalbero abbiamo tutte etichette maggiori della (i-1)- esima etichetta e minori della (i+1)-esima
- Ogni ricerca o modifica comporta la visita di un cammino radice foglia
- In strutture fisiche, un nodo può corrispondere ad un blocco

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

59

59

# Organizzazione dei nodi del B-tree




60

60

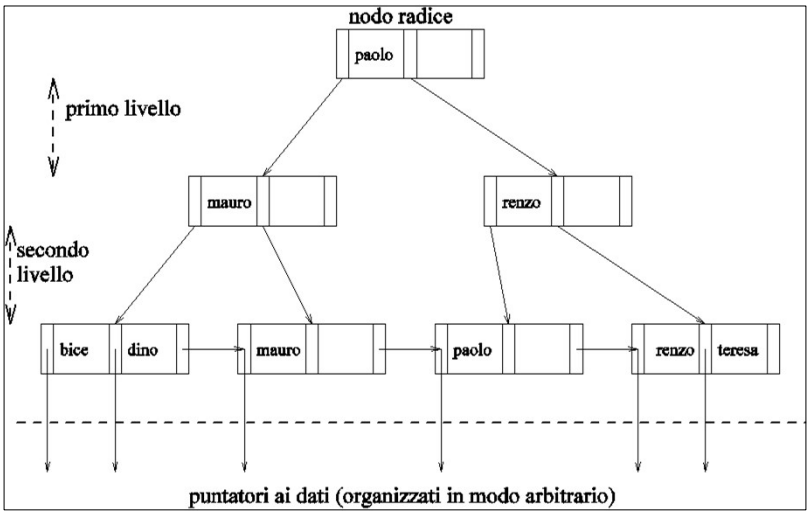
# B tree e B+ tree

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

61

61

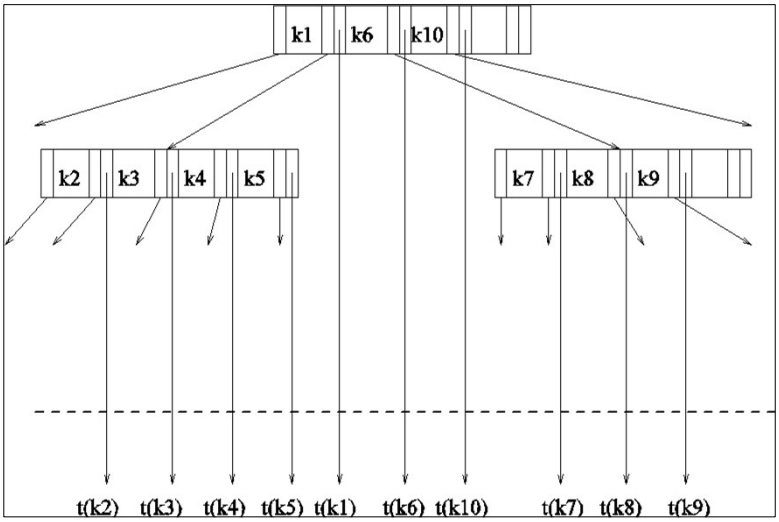
# Un B+ tree




62

62

# Un B-tree




63

63

# Definizione degli indici SQL

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

64

64

# Strutture fisiche nei DBMS relazionali

- Struttura primaria:
  - disordinata (heap, "unclustered")
  - ordinata ("clustered")
  - hash ("clustered")
- Indici (densi/sparsi, semplici/composti):
  - Organizzazione sequenziale (statica), di solito su struttura ordinata
  - B-tree (dinamico)

65

65

# 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)

66

66

# Strutture fisiche in alcuni DBMS

- 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

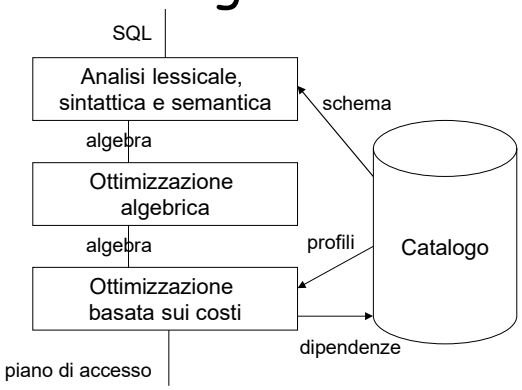
67



# 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




69

# 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"

70

# "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
- dopo l'ottimizzazione algebrica (si ottengono tutte le espressioni minimali equivalenti) va fatta quella in base ai costi
- In questa fase finale dell'ottimizzazione, occorre valutare il numero di trasferimenti in memoria da fare e stimare le dimensioni dei risultati intermedi





# Esecuzione delle operazioni

- I DBMS implementano gli operatori dell'algebra relazionale (o meglio, loro combinazioni) per mezzo di operazioni di livello abbastanza basso
- Operatori fondamentali:
  - accesso diretto
  - scansione
- A livello più alto:
  - ordinamento
- Ancora più alto
  - Join, l'operazione più costosa

# Ottimizzazione basata sui costi

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

73



# Accesso diretto

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

# 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

75

# 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

76

## 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?

77

## Ricerca (scansione)

- L'operatore di selezione esprime la ricerca su una relazione; è possibile implementarlo tramite un algoritmo di ricerca completa la cui complessità media, se la relazione ha n tuple, è lineare e uguale a n/2.
- Tutti i blocchi vanno trasferiti dalla memoria secondaria al buffer

78

78

## Altri metodi

- Se le tuple sono ordinate e la selezione è fatta sull'attributo su cui la relazione è ordinata e i blocchi sono memorizzati contigui, si può ottenere un numero medio di trasferimenti logaritmico $\log_2 n$
- Si possono usare gli indici

79

79

## Ordinamento

- Ci sono query che richiedono un risultato ordinato
- L'ordinamento serve per implementare efficientemente alcune operazioni come il join, ad esempio
- se la relazione sta tutta in memoria si può usare il quicksort altrimenti il mergesort

80

80

# Oltre l'ordine ci sono i metodi di join

- Nested loop
- Merge scan
- Hash-based
- Ognuno ha un costo in termini di accessi al disco

81

81

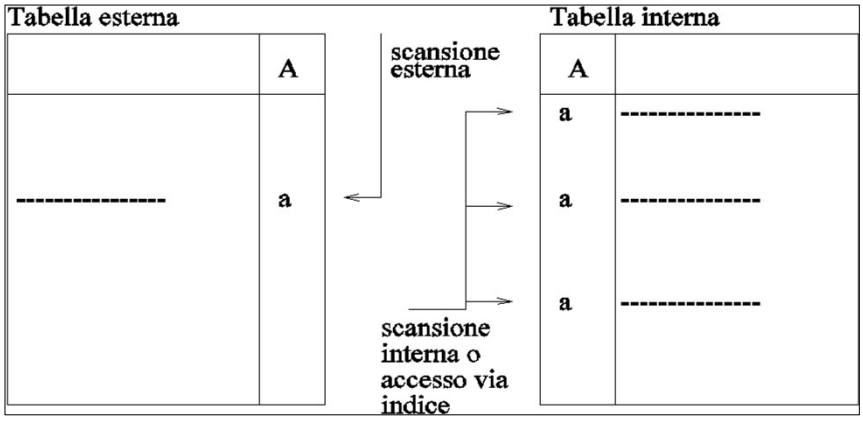
# nested loop

- per ogni tupla nella tabella esterna si esaminano tutte le tuple di quella interna per verificare la condizione di join.
- date R ed S, si hanno due possibilità R esterna o R interna
- il costo, in termini di trasferimenti in memoria, dipende dal numero di accessi e dal fatto che la tabella interna sia piccola

82

82

# Nested-loop




83

# Il costo per nested loop join

- Supponiamo la tabella R abbia 10.000 record che occupano complessivamente 400 blocchi
- La tabella S contiene 5.000 record che occupano 100 blocchi
- Se R è esterna e nel buffer si può mettere solo un record alla volta, il numero di trasferimenti sarà N=10000*100+400
- Se invece nel buffer entra tutta S, N=400+100

84

84

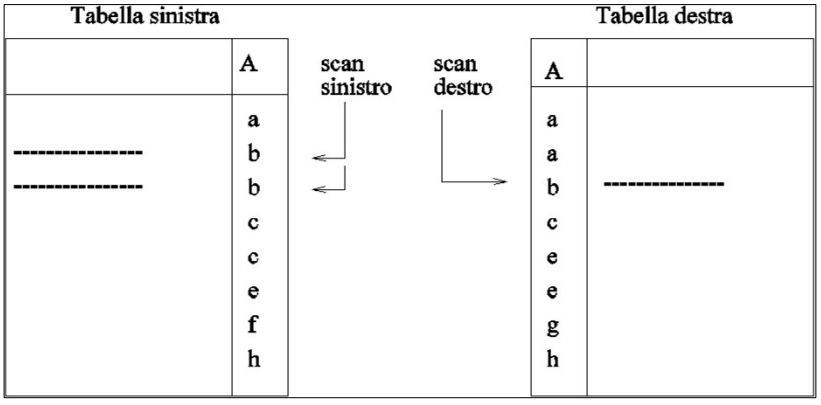
# merge scan

- si ordinano le tabelle in base agli attributi di join
- si trova l'elemento della seconda tabella da cui partire rispetto al primo elemento della prima tabella e poi si continua da lì
- il costo è l'ordinamento

85

85

# Merge-scan




86

# Cont.

- Si usa per il join naturale o l'equi-join
- Ogni blocco deve essere letto una sola volta (assumendo che tutte le tuple per un dato valore di join stiano insieme nel buffer (duplicati))
- Il costo del metodo merge scan è:
  - *br + bs* trasferimenti in memoria
  - + il costo dell'ordinamento se le relazioni sono non ordinate

87

87

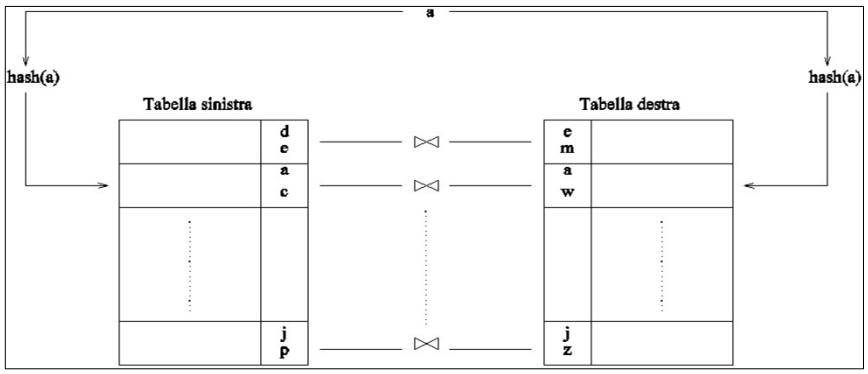
# hash

- Utile solo per join naturale e equi-join
  la funzione hash h sull'atributo di join è usata per partizionare le tuple di entrambe le relazioni
- Le partizioni sono inserite in tabelle aggiuntive, h produce partizioni di S tali che ognuna sta in memoria, R è partizionata di conseguenza
- serve memoria ed è migliore del nested loop nel caso di equijoin

88

88

# Hash join




89

# Costo del hash join

- Le relazioni R e S vengono partizionate sulla base del valore della funzione hash, questo richiede la lettura e la scrittura completa delle due relazioni, ovvero
  - 2(*br* +*bs*) trasferimenti di blocco
- Per ogni tupla $t_r$ in ogni partizione di R, si considera la tupla ts nella partizione corrispondente secondo la funzione hash, quindi si legge ciascuna partizione una volta
  - *(br +bs) trasferimenti di blocco*
- Costo complessivo
  - 3(*br* +*bs* trasferimenti di blocco

90

90

# Misura del costo di una query

- Per concludere, molti fattori contribuiscono al costo di una query, cioè il tempo necessario per avere la risposta:
  - Gli accessi al disco, il tempo di CPU o il tempo di rete
  - L'accesso al disco è il tempo predominante ed è anche facilmente calcolabile considerando
    - Numero di scansioni
    - Numero di letture
    - Numero di scritture
- Per semplicità consideriamo un fattore *t* unico come tempo per accedere un blocco

91

91

# Costo di una query

- Oltre al numero di trasferimenti bisogna considerare anche la memoria che serve per memorizzare i risultati intermedi
- Ad esempio l'hash join necessita di tabelle intermedie
- Operazioni a più operandi devono essere eseguite un passo alla volta

92

92

# Congiunzione di condizioni

- Una congiunzione di operazioni di selezione può essere scomposta in una sequenza di selezioni semplici
  - $\sigma_{\theta 1 \ni \theta 2}(R) = \sigma_{\theta 1}(\sigma_{\theta 2}(R))$
- Le operazioni di selezione sono commutative

  $\sigma_{\theta 1}(\sigma_{\theta 2}(R)) = \sigma_{\theta 2}(\sigma_{\theta 1}(R))$
- l'ordine viene scelto in base alla dimensione del risultato intermedio

93

93

# Ordine dei join

- l'ordine in cui si fanno i join dipende dalla dimensione dei risultati intermedi
- Date le relazioni *R1*, *R2*, R3, per la associatività
- (R1 ⋈ R2) ⋈ R3 = R1 ⋈(R2 ⋈ R3 )
- se *R2* ⋈ *R3* ha una dimensione molto grande, mentre *R1* ⋈ *R2* è piccolo, possiamo scegliere di eseguire (R1 ⋈ R2) ⋈ R3 in modo da dover memorizzare una relazione intermedia più piccola.

94

94

## Ordine dei join cont.

- $\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”}}(branch)) \bowtie (account \bowtie depositor))$
  - *account* $\bowtie$ *depositor* è probabilmente una relazione con molte tuple, ma solo pochi di tutti i clienti di banca hanno un conto in una filiale di Brooklyn, è quindi meglio calcolare prima
  - $\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie account$

95

95

## Il processo di ottimizzazione

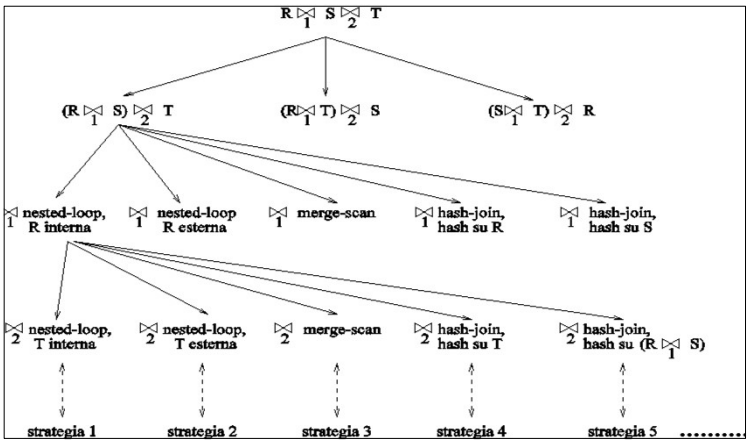
- Si costruisce un albero di decisione con le varie alternative ("**piani di esecuzione**")
- Si valuta il costo di ciascun piano
- Si sceglie il piano di costo minore

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

96

96

# Un albero di decisione




97

97

# Esempio

- Calcoliamo il piano di esecuzione migliore per la seguente interrogazione dal punto di vista della dimensione dei risultati intermedi

$\pi_T (\sigma_{(C=D \vee C=B) \wedge A=X \wedge N=Y} (F \rhd\lhd Ac \rhd\lhd I)$

F(Fid, Titolo, Anno, Categoria,..)
Ac(Aid,Nazionalità,..)
I(Fid, Aid,..)
•

98

98

- N(F)=30000
- N(*A*)=2000
- N(I)=600000

- N(Fid, I)=30000
- N(*A*id,I)=2000
- N(Nazionalità, *A*)=4
- N(*C*ategoria, F)=5
- N(*A*, F)=20

99

99

# Ottimizzazione algebrica

- $\pi_T (\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X} (F)) \rhd\lhd \pi_{AId}(\sigma_{N=Y} (Ac)) \rhd\lhd \pi_{Fid,AId} (I))$

100

100

## Dimensione delle varie relazioni

- R1=$\sigma_{(C=D \vee C=B)}(F)$
  - R11=$\sigma_{(C=D)}(F)$ R12=$\sigma_{(C=B)}(F)$
  - $n_{R11}$=N(F)/N(Categoria, F)=30000/5=6000= $n_{R12}$
  - $n_{R1}$=6000*2=12000
- R2= $\sigma_{A=X}(F)$
  - $n_{R2}$=N(F)/N(A, F)=30000/20=1500

101

101

## Cont.

- L'ordine più conveniente è
  - R3= $\sigma_{(C=D \vee C=B)}(\sigma_{A=X}(F))$
- Il numero finale delle tuple è
  - $n_{R3}$= $n_{R2}$ /N(Categoria, R2)*2=1500/5*2=300*2
- R4= $\pi_{T,FId}$(R3)
  - $n_{R4}$= $n_{R3}$=600

102

102

# Cont.

- R5= ($\sigma_{N=X}(Ac)$)
  - $n_{R5}$=N(Ac)/N(Nazionalità, Ac)=2000/4=500
- R6= $\pi_{AId}$(R5)
  - $n_{R6}$= $n_{R5}$=500
- R7= $\pi_{Fid,AId}$ (I)
  - $n_{R7}$=N(I)=600000

103

103

# Cont.

- $\pi_T$ (R4 $\triangleright\triangleleft$ R6 $\triangleright\triangleleft$ R7)
- R4 $\triangleright\triangleleft$ R6
  - 500*600=300000 tuple
- R6 $\triangleright\triangleleft$ R7
  - min(500*600000/2000,600000*1)= min(150000,600000)=150000
- R4 $\triangleright\triangleleft$ R7
  - min(300*600000/30000,600000*1)= min(6000,600000)=6000

104

104

# Cont.

- Quindi l'ordine sarà
- $\pi_T$ ((R4 $\triangleright\triangleleft$ R7) $\triangleright\triangleleft$ R6 )

105

105

# 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)

106

# Progettazione fisica nel modello relazionale

- La caratteristica comune dei DBMS relazionali è la disponibilità degli indici:
  - la progettazione logica 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`

107