Basi di dati Vol.2
Capitolo 1
Organizzazione fisica e gestione delle interrogazioni

27/04/2004

Tecnologia delle BD: perché studiarla?

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

DataBase Management System — DBMS

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

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

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

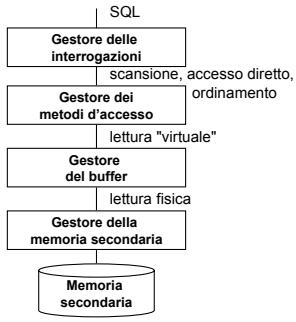
Le basi di dati sono grandi e persistenti

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

Gestore degli accessi e delle interrogazioni



Le basi di dati sono affidabili

- Le basi di dati sono una risorsa per chi le possiede, e debbono essere conservate anche in presenza di malfunzionamenti
- Esempio:
  - un trasferimento di fondi da un conto corrente bancario ad un altro, con guasto del sistema a metà
- Le **transazioni** debbono essere
  - atomiche (o tutto o niente)
  - definitive: dopo la conclusione, non si dimenticano

Le basi di dati vengono aggiornate …

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

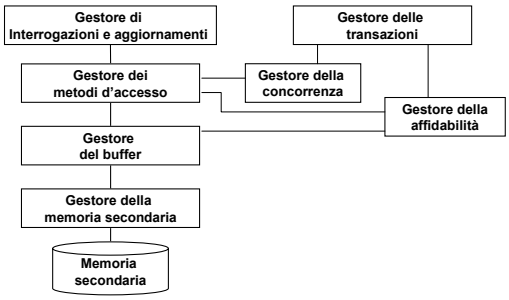
Le basi di dati sono condivise

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

Aggiornamenti su basi di dati condivise …

- Esempi:
  - due prelevamenti (quasi) contemporanei sullo stesso conto corrente
  - due prenotazioni (quasi) contemporanee sullo posto
- Intutuivamente, 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

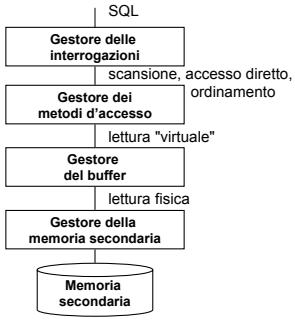
Gestore degli accessi e delle interrogazioni — Gestore delle transazioni



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



Memoria principale e secondaria

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

Memoria principale e secondaria, 2

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

Memoria principale e secondaria, 3

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

  in media non meno di 10 ms
- Il costo di un accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria centrale
- Perciò, nelle applicazioni "I/O bound" (cioè con molti accessi a memoria secondaria e relativamente poche operazioni) il costo dipende esclusivamente dal numero di accessi a memoria secondaria
- Inoltre, accessi a blocchi “vicini” costano meno (contiguità)

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

Scopo della gestione del buffer

- Ridurre il numero di accessi alla memoria secondaria
  - In caso di lettura, se la pagina è già presente nel buffer, non è necessario accedere alla memoria secondaria
  - In caso di scrittura, il gestore del buffer può decidere di differire la scrittura fisica (ammesso che ciò sia compatibile con la gestione dell’affidabilità – vedremo più avanti)
- La gestione dei buffer e la differenza di costi fra memoria principale e secondaria possono suggerire algoritmi innovativi.
- Esempio:
  - File di 1.000.000.000 di record di 100 byte ciascuno (100GB)
  - Blocchi di 10KB
  - Buffer disponibile di 100MB

  Come possiamo fare l’ordinamento?
  - Merge-sort “a più vie”

Dati gestiti dal buffer manager

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

Funzioni del buffer manager

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

Interfaccia offerta dal buffer manager

- ***fix***: richiesta di una pagina; richiede una lettura solo se la pagina non è nel buffer (incrementa il contatore associato alla pagina)
- ***setDirty***: comunica al buffer manager che la pagina è stata modificata
- ***unfix***: indica che la transazione ha concluso l'utilizzo della pagina (decrementa il contatore associato alla pagina)
- ***force***: trasferisce in modo sincrono una pagina in memoria secondaria (su richiesta del gestore dell'affidabilità, non del gestore degli accessi)

Esecuzione della fix

- Cerca la pagina nel buffer;
  - se c'è, restituisce l'indirizzo
  - altrimenti, cerca una pagina libera nel buffer (contatore a zero);
    - se la trova, legge il blocco di interesse dalla memoria secondaria e restituisce l'indirizzo della pagina
    - altrimenti, due alternative
      - "**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
      - "**no-steal**": pone l'operazione in attesa

Commenti

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

DBMS e file system

- Il file system è il componente del sistema operativo che gestisce la memoria secondaria
- I DBMS ne utilizzano le funzionalità, 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, 2

- 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 tuple di una relazione siano in file diversi.
- Spesso, ma non sempre, ogni blocco è dedicato a ennuple di un'unica relazione

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:

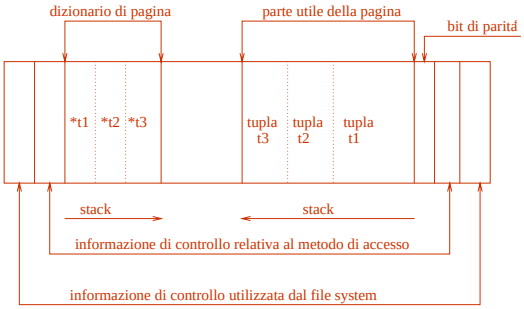
$$\lfloor L_B / L_R \rfloor$$

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

Organizzazione delle ennuple nelle pagine

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

Organizzazione delle ennuple nelle pagine



Strutture

- Sequenziali
- Calcolate ("Hash")
- Ad albero (di solito, indici)

Strutture sequenziali

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

Struttura seriale

- Chiamata anche:
  - "Entry sequenced"
  - file heap
  - file disordinato
- È molto diffusa nelle basi di dati relazionali, associata a indici secondari
- Gli inserimenti vengono effettuati
  - in coda (con riorganizzazioni periodiche)
  - al posto di record cancellati

Entry-sequenced sequential structure

- Optimal for the carrying out of sequential reading and writing operations
- Uses all the blocks available for files and all the spaces within the blocks
- Primitives:
  - Accessed with sequential *scan*
  - Data loading and insertion happen at the end of the file and in sequence
  - Deletes are normally implemented by leaving space unused
  - More problems are caused by the updates that increase the file size

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 happen 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 the key field
- Historically, ordered sequential structures were used on sequential devices (tapes) by batch processes. Data were located into the *main file,* modifications were collected in *differential files,* and the files were *periodically merged*. This has fallen out of use
- The main problems: insertions or updates which increase the physical space - they require reordering of the tuples already present
- 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 fino ad un certo punto (ad esempio, come troviamo la "metà del file"?
- Nelle basi di dati relazionali si utilizzano quasi solo in combinazione con indici (file ISAM o file ordinati con indice primario)

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) rispetto a quello strettamente necessario
    - poiché il numero di possibili chiavi è molto maggiore del numero di possibili indirizzi ("lo spazio delle chiavi è più grande dello spazio degli indirizzi"), la funzione non può essere iniettiva e quindi esiste la possibilità di collisioni (chiavi diverse che corrispondono allo stesso indirizzo)
    - le buone funzioni hash distribuiscono in modo causale e uniforme, riducendo le probabilità di collisione (che si riduce aumentando lo spazio ridondante)

Un esempio

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

M	M mod 50	M	M mod 50
60600	0	200268	18
66301	1	205619	19
205751	1	210522	22
205802	2	205724	24
200902	2	205977	27
116202	2	205478	28
200604	4	200430	30
66005	5	210533	33
116455	5	205887	37
200205	5	200138	38
201159	9	102338	38
205610	10	102690	40
201260	10	115541	41
102360	10	206092	42
205460	10	205693	43
205912	12	205845	45
205762	12	200296	46
200464	14	205796	46
205617	17	200498	48
205667	17	206049	49

Tavola hash, 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

File hash

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

Un esempio

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

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

  numero medio di accessi: 1,05

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

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

Un file hash

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

File hash, osservazioni

- È l'organizzazione più efficiente per l'accesso diretto basato su valori della chiave con condizioni di uguaglianza (accesso puntuale): costo medio di poco superiore all'unità (il caso peggiore è molto costoso ma talmente improbabile da poter essere ignorato)
- Le collisioni (overflow) sono di solito gestite con blocchi collegati
- Non è efficiente per ricerche basate su intervalli (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

Collisions

- Occur when the same block number is returned by the algorithm starting from two different keys. If each page can contain a maximum of *F* tuples, a collision is critical when the value of *F* is exceeded
- Solved by adding an overflow chain starting from that page. They give the additional cost of scanning the chain
- The average length of the overflow chain is tabled as a function of the ratio T/(*F* x B) and of the average number *F* of tuples per page:

	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	
T/(*F* x*B*)						

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

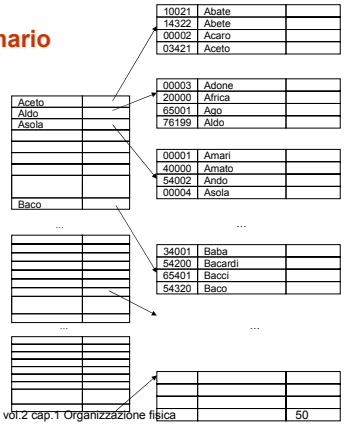
Tipi di indice

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

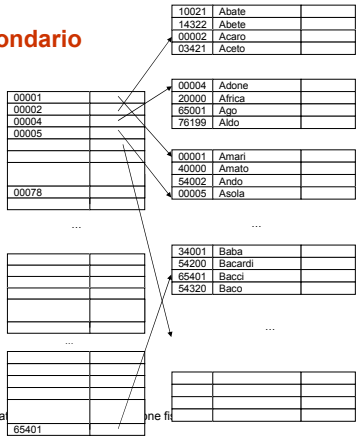
Tipi di indice, commenti

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

Indice primario



Indice secondario



Dimensioni dell'indice

- L numero di record nel file
- B dimensione dei blocchi
- R lunghezza dei record (fissa)
- K lunghezza del campo chiave
- 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: $N_S = N_F / (B/(K+P))$

Caratteristiche degli indici

- Accesso diretto (sulla chiave) efficiente, sia puntuale sia per intervalli
- 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

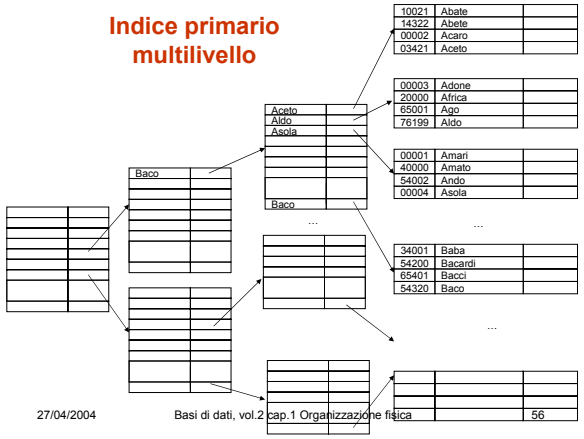
Indici secondari, due osservazioni

- 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)
- 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)

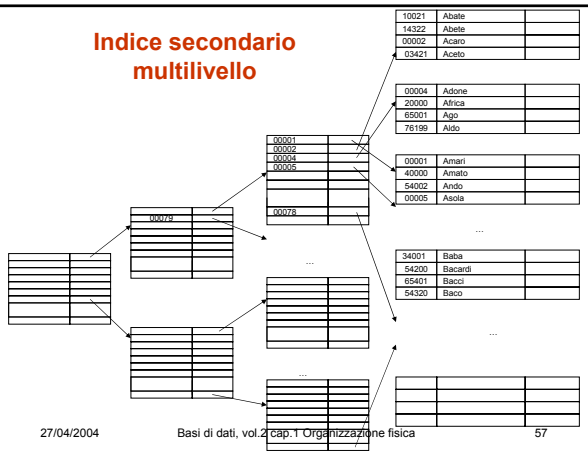
Indici multilivello

- Gli indici sono file essi stessi e quindi ha senso costruire indici sugli indici, per evitare di fare ricerche fra blocchi diversi
- Possono esistere più livelli fino ad avere il livello più alto con un solo blocco; i livelli sono di solito abbastanza pochi, perché
  - l'indice è ordinato, quindi l'indice sull'indice è sparso
  - i record dell'indice sono piccoli
- N_j numero di blocchi al livello j dell'indice (circa):
  - $N_j = N_{j-1} / (B/(K+P))$

Indice primario multilivello



Indice secondario multilivello

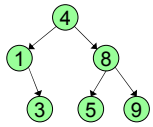


Indici, problemi

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

Albero binario di ricerca

- Albero binario etichettato in cui per ogni nodo il sottoalbero sinistro contiene solo etichette minori di quella del nodo e il sottoalbero destro etichette maggiori
- tempo di ricerca (e inserimento), pari alla profondità:
  - logaritmico nel caso "medio" (assumendo un ordine di inserimento casuale)



Albero di ricerca di ordine P

- Ogni nodo ha (fino a) P figli e (fino a) P-1 etichette, ordinate
- Nell'i-esimo sottoalbero abbiamo tutte etichette maggiori della (i-1)- esima etichetta e minori della i-esima
- Ogni ricerca o modifica comporta la visita di un cammino radice foglia
- In strutture fisiche, un nodo può corrispondere ad un blocco
- La struttura è ancora (potenzialmente) rigida
- Un B-tree è un albero di ricerca che viene mantenuto bilanciato, grazie a:
  - Riempimento parziale (mediamente 70%)
  - Riorganizzazioni (locali) in caso di sbilanciamento

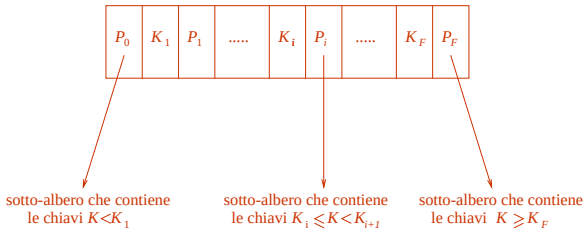
Tree structures

- The most frequently used in relational DBMSs
- Gives associative access (based on a value of a *key*, consisting of one or more attributes) without constraints on the physical location of the tuples
- Note: the primary key of the relational model and the key of tree structures are different concepts

Tree structure organization

- Each tree has:
  - a root node
  - a number of intermediate nodes
  - a number of leaf nodes
- The links between the nodes are established by pointers
- Each node coincides with a page or block at the file system and buffer manager levels. In general, each node has a large number of descendants (fan out), and therefore the majority of pages are leaf nodes
- In a *balanced tree*, the lengths of the paths from the root node to the leaf nodes are all equal. In this case, the access times to the information contained in the tree are almost constant (and optimal)

Organizzazione dei nodi del B-tree



Node contents

- Each intermediate node contains F keys (in lexicographic order) and $F + 1$ pointers
- Each key K_j , $1 \leq j \leq F$, is followed by a pointer P_j; K_1 is preceded by a pointer P_0
- Each pointer addresses a sub-tree:
  - P_0 addresses the sub-tree with the keys less than K_1
  - P_F addresses the sub-tree with keys greater than or equal to K_F
  - P_j, $0 < j < F$, addresses the sub-tree with keys included in the interval $K_j \leq K < K_{j+1}$
- The value $F + 1$ is called the *fan-out* of the tree

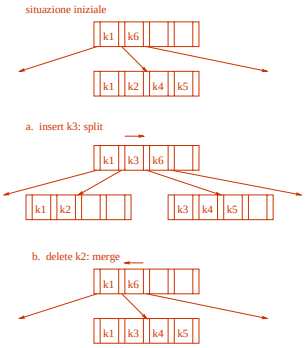
Search technique

- At each intermediate node:
  - if $V < K_1$ follow the pointer P_0
  - if $V \geq K_F$ follow the pointer P_F
  - otherwise, follow the pointer P_j such that $K_j \leq V < K_{j+1}$
- The leaf nodes of the tree can be organized in two ways:
  - In *key-sequence*d trees the tuple is contained in the leaves
  - In *indirect trees* leaf nodes contain pointers to the tuples, that can be allocated by means of any other 'primary' mechanism (for example, entry-sequenced, hash, or key-sequenced)
- In some cases, the index structure is sparse (not complete). They locate a key value close to the value being sought, then a sequential search is carried out

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 trattae come eliminazioni seguite da inserimenti

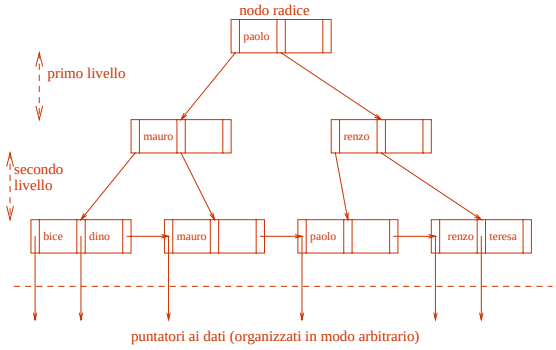
Split and merge operations



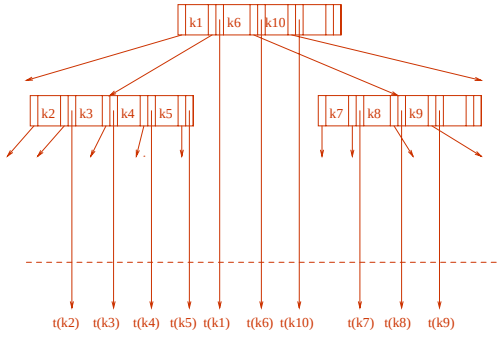
B tree e B+ tree

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

Un B+ tree



Un B-tree



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

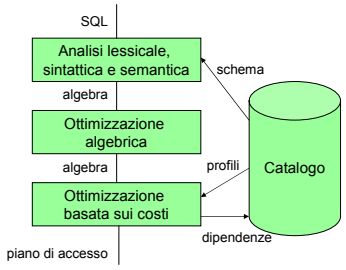
Query processing

- The query processor receives a query written in SQL, produces an access program is obtained in 'object' or 'internal' format, which uses the data structures provided by the system. Steps:
  - Lexical, syntactic and semantic analysis, using the data dictionary
  - Translation into an internal, algebraic form
  - Algebraic optimization (execution of all the algebraic transformation that are always convenient, such as the 'push' of selections)
  - Cost-based optimization
  - Code generation using the physical data access methods provided by the DBMS

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



"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 le dimensioni dei risultati intermedi

Ottimizzazione algebrica

- Il termine ottimizzazione è improprio (anche se efficace) perché il processo utilizza euristiche
- Si basa sulla nozione di equivalenza:
  - Due espressioni sono equivalenti se producono lo stesso risultato qualunque sia l'istanza attuale della base di dati
- I DBMS cercano di eseguire espressioni equivalenti a quelle date, ma meno "costose"
- Euristica fondamentale:
  - selezioni e proiezioni il più presto possibile (per ridurre le dimensioni dei risultati intermedi):
    - "push selections down"
    - "push projections down"

"Push selections"

- Assumiamo A attributo di R_2

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

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

Rappresentazione interna delle interrogazioni

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

Alberi per la rappresentazione di interrogazioni

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

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
- Anticipare il più possibile le proiezioni (anche introducendone di nuove)

Esempio

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

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

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

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

Esempio, continua

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

- diventa qualcosa del tipo

$$PROJ_{AE}\ (SEL_{B>100}\ (R1)\ JOIN_{C=D}\ R2)\ JOIN_{F=G}\ SEL_{I>2}(SEL_{H=7}(R3)))$$

- oppure

$$PROJ_{AE}(PROJ_{AEF}((PROJ_{AC}(SEL_{B>100}\ (R1)))\ JOIN_{C=D}\ R2)\ JOIN_{F=G}\ PROJ_{G}\ (SEL_{I>2}(SEL_{H=7}(R3))))$$

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 simple predicate (of type: $A_i = v$)
  - Insertions, deletions, and modifications of the tuples currently accessed during the scan
- Primitives:
  `open`, `next`, `read`, `modify`, `insert`, `delete`, `close`

Sort operation

- Various methods for ordering the data contained in the main memory, typically represented by means of a record array
- DBMSs typically cannot load all data in the buffer; thus, they separately order and then merge data sets, using the available buffer space

Accesso diretto

- Può essere eseguito solo se le strutture fisiche lo permettono
  - strutture hash
  - indici (vediamo in dettaglio)

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

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

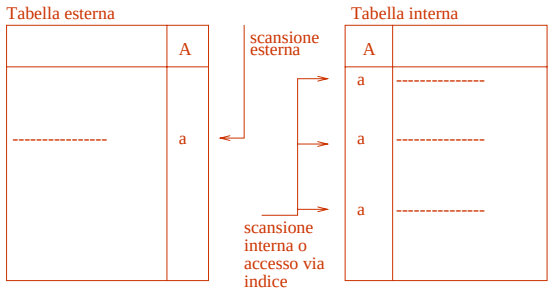
Indici e hash su più campi

- 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?

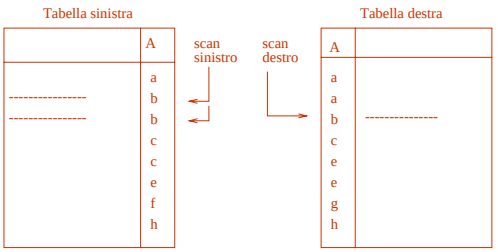
Join

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

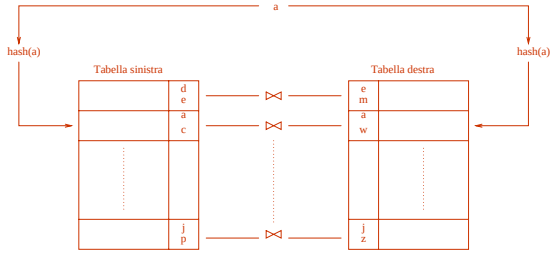
Nested-loop



Merge-scan



Hash join



Ottimizzazione basata sui costi

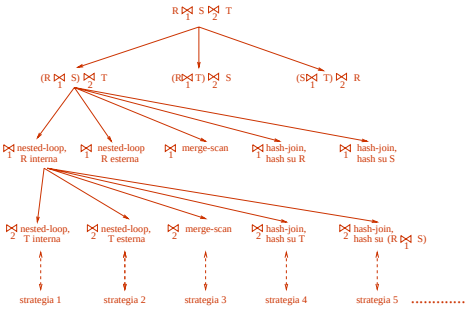
- Un problema articolato, con scelte relative a:
  - operazioni da eseguire (es.: scansione o accesso diretto?)
  - ordine delle operazioni (es. join di tre relazioni; ordine?)
  - i dettagli del metodo (es.: quale metodo di join)
- Architetture parallele e distribuite aprono ulteriori gradi di libertà

Il processo di ottimizzazione

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

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

Un albero di decisione



Progettazione fisica

- 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)

Progettazione logica 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`

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*

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)

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)

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