ORGANIZZAZIONE FISICA DELLE BASI DI DATI

Vincenzo Calabrò

Organizzazione fisica dei dati

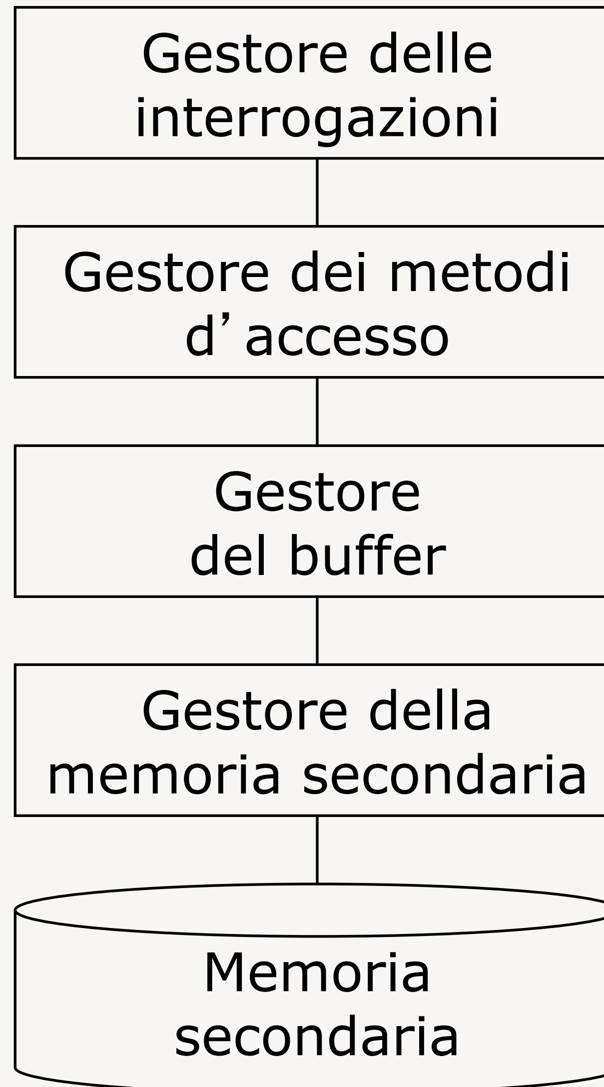
- livello logico indipendente dal livello fisico (indipendenza fisica)
- utenti (e operazioni SQL) fanno riferimento al livello logico
- DBMS traduce le definizioni e interrogazioni specificate sul livello logico in strutture e accessi a livello fisico

Moduli DBMS per accesso ai dati (1)

L'esecuzione delle interrogazioni coinvolge diversi moduli

- gestore (o ottimizzatore) delle interrogazioni: traduce le interrogazioni in operazioni su strutture fisiche (in modo ottimizzato)
- gestore dei metodi di accesso: traduce le operazioni su strutture fisiche in operazioni di lettura/scrittura su memoria secondaria
- gestore del buffer: media operazioni su memoria secondaria mantenendo temporaneamente dati in memoria principale
- gestore della memoria secondaria: esegue le scritture/letture su memoria secondaria

Moduli DBMS per accesso ai dati (2)



Memoria secondaria: vantaggi

- grandi dimensioni
  - la dimensione delle basi di dati è maggiore di quella della memoria principale
- garantisce persistenza
  - i dati in memoria principale sono volatili

Memoria secondaria: svantaggi

- non direttamente utilizzabile dai programmi
  - i dati devono essere trasferiti in memoria principale

- dati organizzati in blocchi di dimensione fissa
  - uniche operazioni possibili a livello di blocco
  - l'accesso a un bit richiede recupero intero blocco

- costo (tempo di accesso)
  - molto superiore alla memoria principale (10^{-3} vs 10^{-9})
  - dipende dall'ordine degli accessi
    - tempo di posizionamento della testina
    - tempo di latenza
    - tempo di trasferimento

Buffer

Zona di memoria centrale gestita dal DBMS in modo condiviso con tutte le applicazioni

- permette di evitare di ripetere accessi a memoria secondaria
- organizzato in **pagine** di dimensione pari ad un numero intero di blocchi
- gestore del buffer si occupa del caricamento e scaricamento delle pagine fra memoria principale e secondaria

Gestore del buffer

Informazioni di gestione, costituite da

- direttorio che descrive il contenuto del buffer e la corrispondenza fra pagine e blocchi fisici
- variabili di stato per ogni pagina
  - contatore: numero di applicazioni che utilizzano la pagina
  - bit di stato: indica se la pagina è stata modificata

Ottimizza accessi ai dati

- letture e scritture sono effettuate su pagine nel buffer
  - scritture su memoria secondaria sono differite

Gestore del buffer: primitive

- fix: per caricare una pagina nel buffer (quando non presente)
  - incrementa il contatore relativo all'utilizzo della pagina
- unfix: per terminare l'utilizzo di una pagina
  - decrementa il contatore
- setDirty: per indicare che la pagina è stata modificata
  - modifica bit di stato
- force: trasferisce in modo sincrono una pagina del buffer a memoria secondaria
- flush: trasferisce in modo asincrono pagine non valide (non utilizzate) a memoria secondaria

Primitiva fix

- se la pagina è nel buffer
  - restituisce l’indirizzo
- altrimenti (se pagine libere nel buffer)
  - legge la pagina da memoria secondaria e la carica nel buffer, restituendo poi l’indirizzo
- altrimenti (no pagine libere nel buffer)
  - seleziona pagina non libera nel buffer (vittima) e la riscrive in memoria secondaria (flush)
  - legge la pagina da caricare da memoria secondaria e la carica nel buffer, restituendo l’indirizzo

Primitiva fix: scelta della vittima

Due politiche possibili

- steal: permette al gestore del buffer di scegliere come vittima una pagina allocata a un'altra transazione
- no-steal: esclude questa possibilità
  - generalmente utilizzato dai DBMS commerciali

Gestore del buffer: politiche

Politiche di gestione simili a quelle usate dai sistemi operativi

- principio di località
  - dati referenziati di recente hanno maggiore probabilità di essere referenziati in futuro
- legge empirica
  - il 20% dei dati è tipicamente acceduto dall'80% delle applicazioni

DMBS e file system

DBMS:

- utilizza SO per creare e cancellare file e per operazioni su blocchi
- crea una propria astrazione dei file
  - per garantire efficienza e transazionalità
- gestisce i blocchi dei file allocati come se fossero un unico grande spazio di memoria e vi costruisce le proprie strutture fisiche
- fornisce metodi di accesso (moduli software) per accedere alle diverse strutture

Separazione di compiti non semplice e netta

Organizzazione dei file

L'organizzazione (struttura) fisica regola la disposizione delle tuple all'interno del file:

- sequenziale
  - tuple disposte in modo sequenziale
- accesso calcolato (hash-based)
  - posizione delle tuple determinata da un algoritmo di calcolo
- a indici (index-based)
  - organizzazione ad albero

Organizzazione sequenziale delle tuple

- seriale (entry-sequenced/heap)
  - la sequenza delle tuple è indotta dal loro ordine di inserimento
- ad array
  - le tuple sono disposte in un array e la loro posizione dipende dal valore di uno o più campi indice
- sequenziale ordinata
  - la sequenza delle tuple dipende dal valore, in ciascuna tupla, di un campo di ordinamento (campo chiave)

Nota: chiave non è necessariamente la chiave primaria

Struttura sequenziale seriale (1)

La sequenza delle tuple è indotta dal loro ordine di inserimento

- accesso con scan sequenziale

Struttura sequenziale seriale (2)

Vantaggi

- ottimale per le operazioni di scrittura (avviene alla fine del file) e lettura sequenziali

Svantaggi

- le cancellazioni possono lasciare spazio non utilizzato
- le modifiche che creano aumento delle dimensioni della tupla non possono essere gestite ‘in loco’
- l’accesso a singoli dati richiede la scansione dell’intero file

Struttura sequenziale ad array (1)

Le tuple sono disposte in un array e la loro posizione dipende dal valore di uno o più campi indice

- array composto da n blocchi contigui, ognuno di m slot disponibili per le tuple (array di $n \cdot m$ posizioni)
- ogni tupla ha un indice numerico i ed è posizionata nell'i-esima cella dell'array

Struttura sequenziale ad array (2)

Vantaggi

- accesso diretto (sulla base della chiave)

Svantaggi

- possibile solo quando le tuple hanno lunghezza fissa
- le cancellazioni lasciano celle vuote
- si può applicare solo quando la chiave assume valori consecutivi

Raramente utilizzata nei DBMS a causa dei vincoli di applicabilità

Struttura sequenziale ordinata (1)

La sequenza delle tuple dipende dal valore, in ciascuna tupla, di un campo di ordinamento (campo chiave)

Vantaggi
- efficienti le operazioni basate sul valore della chiave

Svantaggi
- inserimenti e modifiche richiedono la riorganizzazione delle tuple già presenti

Struttura sequenziale ordinata (2)

Storicamente utilizzate su device sequenziali (nastri) da processi batch

- i dati erano posizionati in un file principale
- le modifiche raccolte in file differenziali
- i file periodicamente fusi (merge)

Merge periodico inaccettabile per i DBMS

Struttura sequenziale ordinata (3)

Per evitare riordino di tuple nei DBMS si può

- lasciare un certo numero di slot liberi al tempo del caricamento (permette riordinamento locale)
- inserire nuovi blocchi a seguito delle saturazioni
  - svantaggio: perde continuità dei blocchi
- integrare il file ordinato con un file di overflow
  - blocchi dei file di overflow collegati a formare una catena di overflow
  - ricerche sequenziali devono essere inframmezzate con analisi dei blocchi di overflow

Organizzazione dei file

L'organizzazione (struttura) fisica regola la disposizione delle tuple all'interno del file:

- sequenziale
  - tuple disposte in modo sequenziale
- accesso calcolato (hash-based)
  - posizione delle tuple determinata da un algoritmo di calcolo
- a indici (index-based)
  - organizzazione ad albero

Struttura con accesso calcolato (1)

Posizione delle tuple è determinata da un algoritmo di calcolo (hash-based)

- alloca al file *B* bucket (spesso adiacenti)
- la locazione fisica di una tupla dipende da un campo chiave (composto da un numero arbitrario di attributi di una data tabella)
- il metodo di accesso utilizza un algoritmo di hash che, applicato alla chiave, ritorna un valore fra *0* e *B–1*

Struttura con accesso calcolato (2)

- primitiva di interfaccia:
  `hash(fileid,Key):BucketId`
- l'implementazione consiste in due parti:
  - folding: trasforma il valore chiave in un intero positivo, distribuito uniformemente su un intervallo
  - hashing: trasforma l'intero positivo in un numero tra 0 e $B-1$
- funziona meglio quando il file è più grande del necessario
  - T il numero di tuple attese
  - F il numero medio di tuple per pagina
  - una buona scelta per B è $T/(0{,}8 \cdot F)$, utilizzando solo l'80% dello spazio disponibile

Struttura con accesso calcolato (3)

Numero di valori della chiave è maggiore del numero dei blocchi

- funzione hash non iniettiva
- si possono verificare collisioni:
  - l'algoritmo restituisce lo stesso blocco per due chiavi differenti.
- se ogni pagina può contenere un massimo di *F* tuple, la collisione è critica quando si oltrepassa il valore *F*

Struttura con accesso calcolato (4)

Collisioni risolte aggiungendo una catena di overflow partendo da ogni pagina

- aggiunge il costo di scandire la catena
- la lunghezza media della catena di overflow
  - aumenta con l'aumentare del coefficiente di riempimento del file
  - diminuisce con l'aumentare della dimensione del bucket

Struttura con accesso calcolato (5)

Vantaggi

- assicura un accesso associativo ed efficiente ai dati

Svantaggi

- collisioni
- le modifiche alla dimensione del file possono richiedere ristrutturazione
- ottimizza solo accessi basati sul campo chiave
- inefficiente per operazioni su intervalli

Gestione delle tuple nelle pagine (1)

Ogni pagina sequenziale o hash-based ha (assumendo un blocco per pagina):

- una parte iniziale (block header) e una parte finale (block trailer) contenenti l'informazione di controllo usata dal file system
- una parte iniziale (page header) e una parte finale (page trailer) contenenti l'informazione di controllo del metodo di accesso

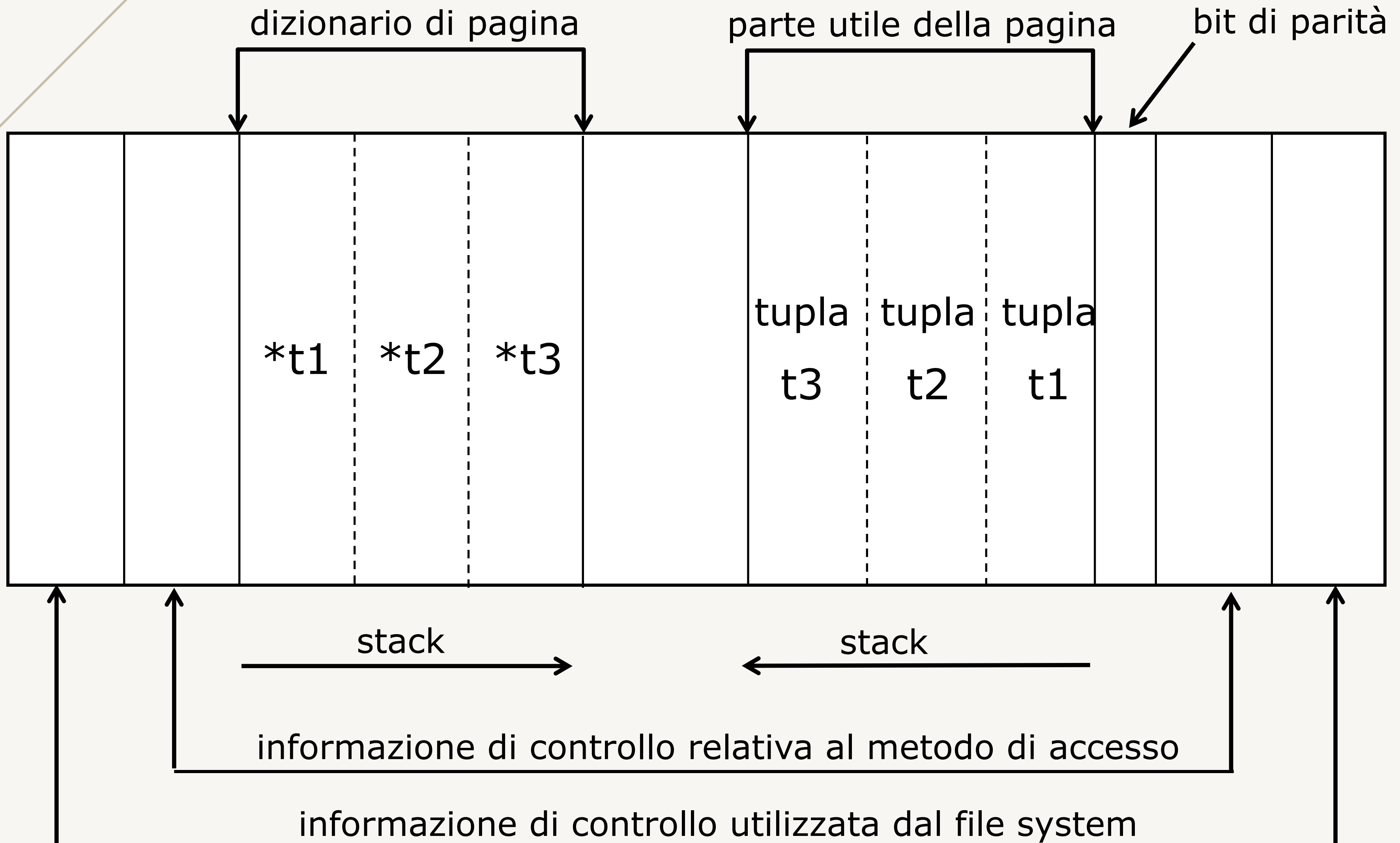
Gestione delle tuple nelle pagine (2)

Ogni pagina sequenziale o hash ha:

- un dizionario di pagina che contiene i puntatori a ogni dato elementare contenuto nella pagina
- una parte utile che contiene i dati; generalmente il dizionario e la parte utile crescono come stack contrapposti
- un bit di parità per verificare l'integrità dell'informazione contenuta nella pagina

Le strutture ad albero hanno una struttura diversa

Gestione delle tuple nelle pagine (3)



Gestione delle tuple nelle pagine (4)

Primitive:

- inserimento e aggiornamento di una tupla
  - può richiedere una riorganizzazione della pagina se non c'è spazio contiguo sufficiente per gestire i byte addizionali introdotti
- cancellazione di una tupla
  - spesso implementata marcando la tupla come 'invalida'
- accesso ad una specifica tupla
  - in base alla chiave (accesso associativo) o all'offset
- accesso ad un campo di una specifica tupla
  - identificato in base all'offset e alla lunghezza del campo stesso (dopo aver identificato la tupla)

Organizzazione dei file

L'organizzazione (struttura) fisica regola la disposizione delle tuple all'interno del file:

- sequenziale
  - tuple disposte in modo sequenziale
- accesso calcolato (hash-based)
  - posizione delle tuple determinata da un algoritmo di calcolo
- a indici (index-based)
  - organizzazione ad albero

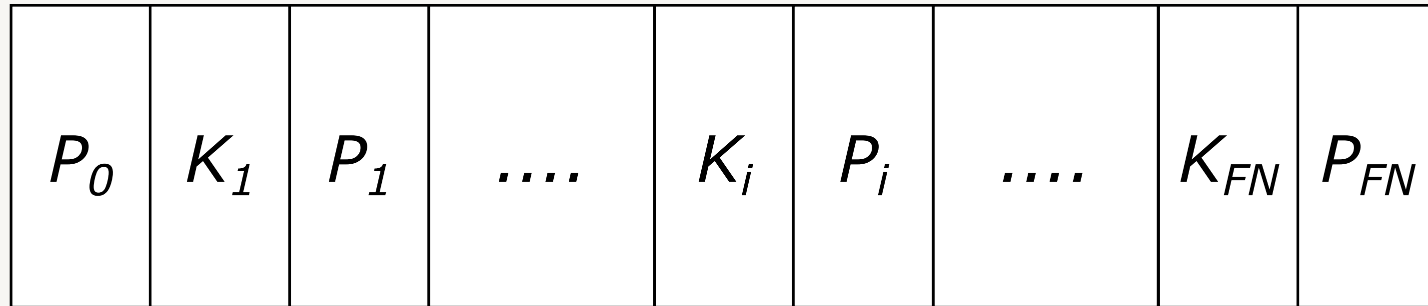
Struttura ad albero

- ogni albero ha:
  - un nodo radice
  - un insieme di nodi intermedi
  - un insieme di nodi foglia
- ogni nodo coincide con una pagina o blocco a livello file system o gestore del buffer
- le relazioni fra i nodi sono stabilite da puntatori
  - fan-out (*F*): numero di puntatori (discendenti) massimo ammesso per ogni nodo dell'albero

Alberi B

- ogni nodo *N* contiene:
  - $F_N \leq F\text{-}1$ valori (ordinati secondo la chiave)
  - $F_N + 1$ puntatori
- ogni valore con chiave K_j , $1 \leq j \leq F_N$, è seguita da un puntatore P_j ; K_1 è preceduta da un puntatore P_0
- puntatori in nodi non foglia:
  - P_0 indirizza il sottoalbero con chiavi < di K_1
  - P_{FN} indirizza il sottoalbero con chiavi > di K_{FN}
  - P_j , $0 < j < F_N$ indirizza il sottoalbero con chiavi *K:* $K_j < K < K_{j+1}$
- i nodi foglia sono tutti allo stesso livello e hanno puntatori a null

Alberi B: nodi



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

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

sotto-albero
che contiene
le chiavi
$K>K_{FN}$

Alberi B+

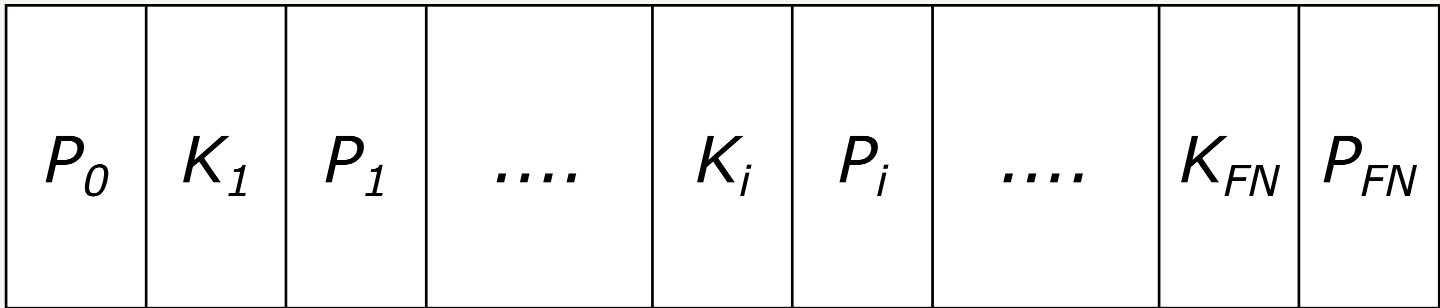
Come gli alberi B ma, per ogni nodo N:

- P_{FN} indirizza il sottoalbero con chiavi $\geq$ di K_{FN}
- P_j , $0 < j < F_N$ indirizza il sotto-albero con chiave K: $K_j \leq K < K_{j+1}$

  quindi le foglie contengono tutti i valori della chiave

- i nodi foglia sono fra loro collegati da una catena sulla base dell'ordine della chiave

Alberi B+: nodi



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

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

sotto-albero che contiene le chiavi $K \geq K_{FN}$

Alberi B e B+ (1)

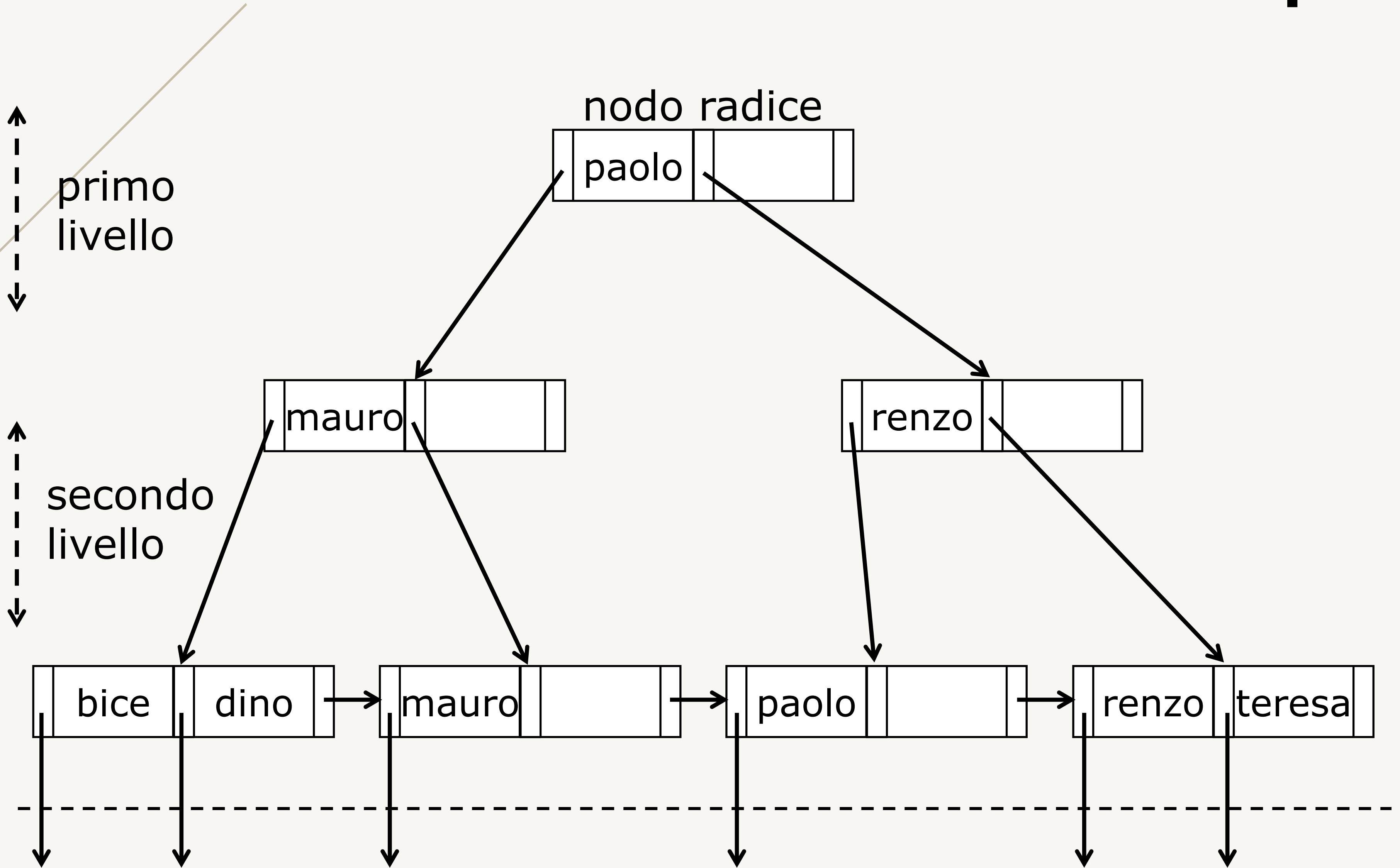
Valori dei nodi:

- chiave + tuple (indice primario)
  - insieme ad ogni chiave è memorizzata la tupla associata alla chiave (key-sequenced)
- chiave + puntatore (indice secondario)
  - insieme alla chiave è memorizzato un puntatore al blocco della base di dati che contiene la tupla associata alla chiave
  - le tuple possono essere allocate per mezzo di altri meccanismi primari (es., sequenziale o hash)

Alberi B e B+ (2)

- spesso utilizzati come indice secondario
  - indicizzano informazione già organizzata in un file (anche con altre strutture)
- forniscono un modo efficiente per accedere alle tuple in dipendenza di uno o più chiavi
- ogni file può avere 1 solo indice primario e più indici secondari

Alberi B+: esempio



puntatori ai dati (organizzati in modo arbitrario)

Alberi B+: operazioni

- ricerca di (tuple con) un valore di chiave
- inserimento di un valore di chiave
- cancellazione di un valore di chiave
- modifica di un valore di chiave
  - cancellazione seguita da inserimento

Alberi B+: ricerca

Ricerca di valore *V*:

- partendo dalla radice, ad ogni nodo non foglia *N*
  - se $V < K_1$ segui il puntatore P_0
  - se $V \geq K_{FN}$ segui il puntatore P_{FN}
  - altrimenti segui il puntatore P_j tale che $K_j \leq V < K_{j+1}$
- se arrivato ad un nodo foglia
  - l'indice è primario: leggi le tuple
  - l'indice è secondario: segui puntatore alle tuple

Alberi B+ ottimizzano l'accesso a intervalli (si trova il primo valore e poi si percorre la catena fra le foglie)

Alberi B+: inserimento (1)

Inserimento di valore *V*:

- partendo dalla radice, procedi cercando *V*
- arrivato ad un nodo foglia *N*
  - se non pieno: inserisci il valore nella foglia
  - se pieno: split and promotion

Alberi B+: inserimento (2)

Split and promotion

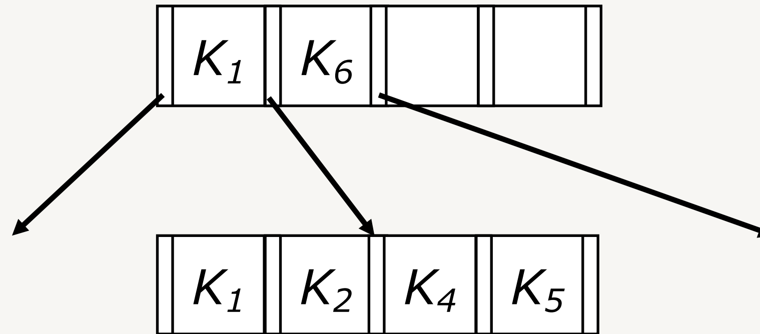
- supponi di inserire valore in *N* e metti i valori (incluso quello nuovo) in ordine
- sia K_i la chiave in posizione $\lceil (F+1)/2 \rceil$
- crea nodo *N*' e trasporta valori con $K > K_i$ ($\geq$ se *N* è foglia) in *N*'
- inserisci valore K_i seguito da puntatore a *N*' nel padre di *N*
- se padre di *N* è pieno: split and promotion

Split and promotion può propagarsi fino alla radice

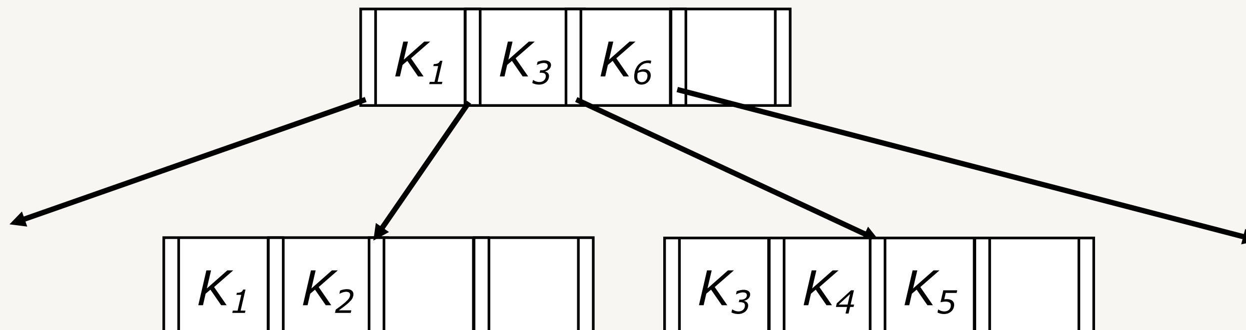
$\Rightarrow$ aumenta l'altezza dell'albero

Alberi B+: inserimento (3)

situazione iniziale

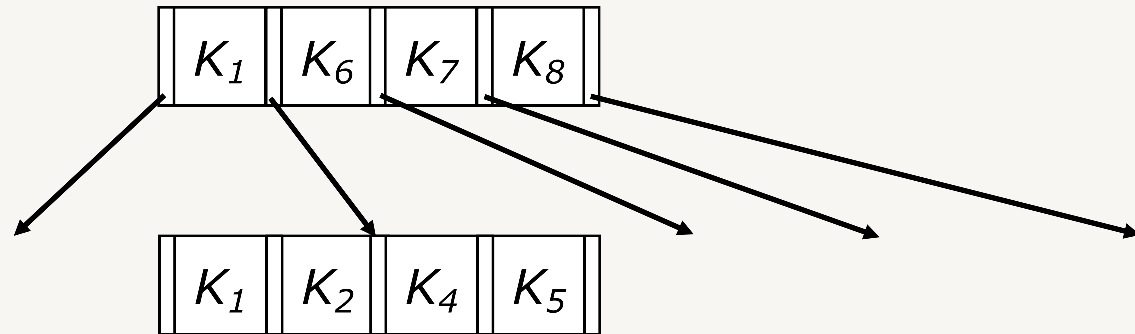


inserimento K_3: split

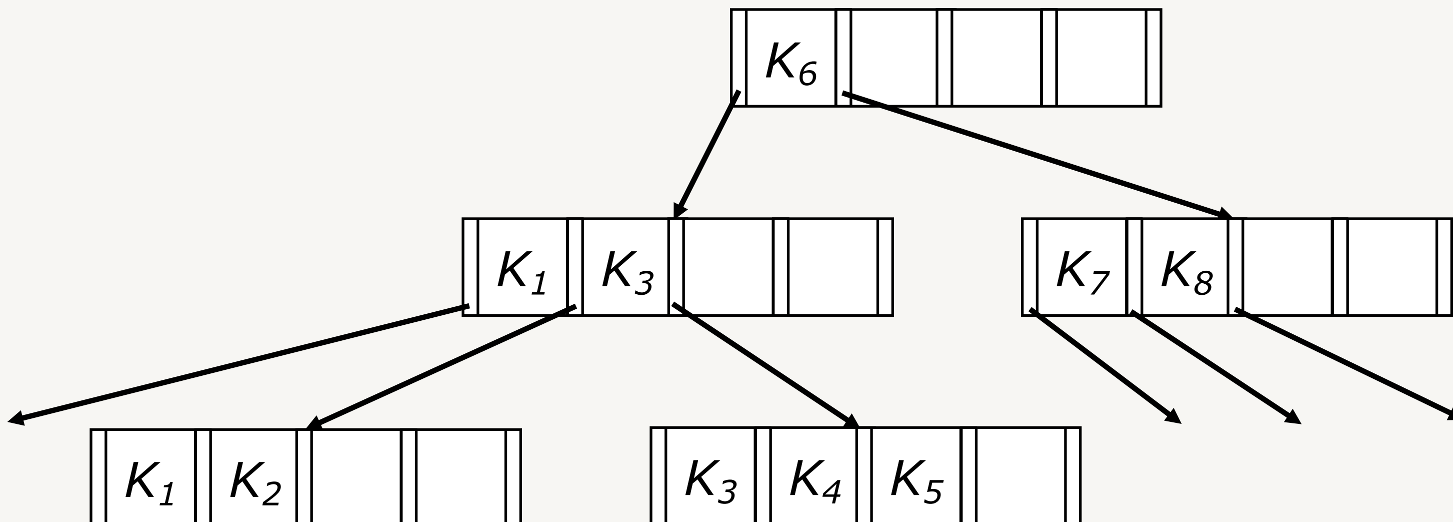


Alberi B+: inserimento (4)

situazione iniziale



inserimento K_3: split



Alberi B+: cancellazione (1)

Cancellazione di valore V:

- cerca V fino a nodo foglia N
- cancella V
  - se V era primo valore (K_1) sostituisci il nuovo K_1 (ex K_2) a V nel suo ascendente
  - se N troppo vuoto (es., meno di $\lfloor F/2 \rfloor$ valori)
    - se possibile bilancia con nodo adiacente
    - se non possibile: merge

Alberi B+: cancellazione (2)

Merge

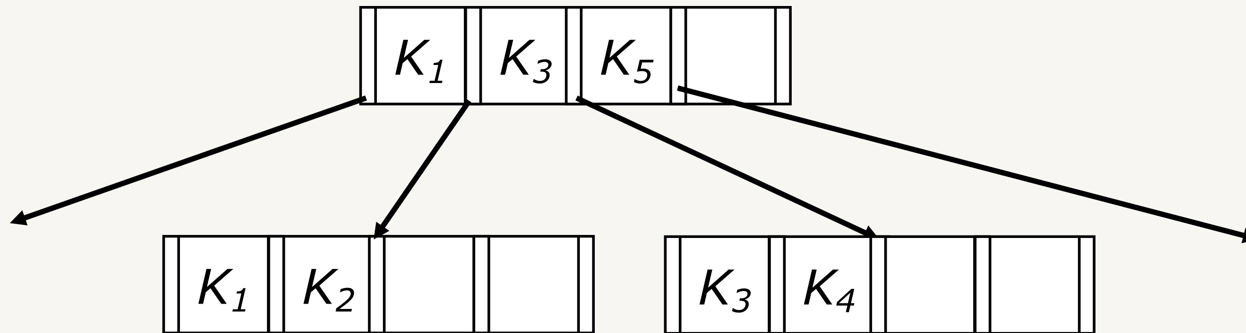
- unisci *N* a un nodo adiacente *N'* a creare unico nodo
  - con i valori di *N* e *N' ed il valore che li separava nel padre*
- cancella valore (e puntatore alla sua destra) che separava *N* e *N'* nel padre
- se padre troppo vuoto: balance o merge

Merge può propagarsi fino alla radice

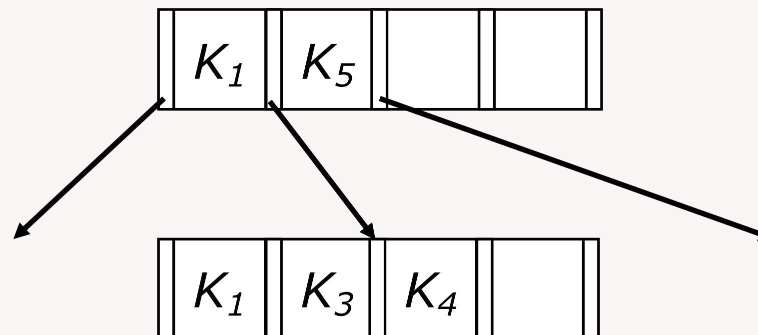
$\Rightarrow$ diminuisce l'altezza dell'albero

Alberi B+: cancellazione (3)

situazione iniziale

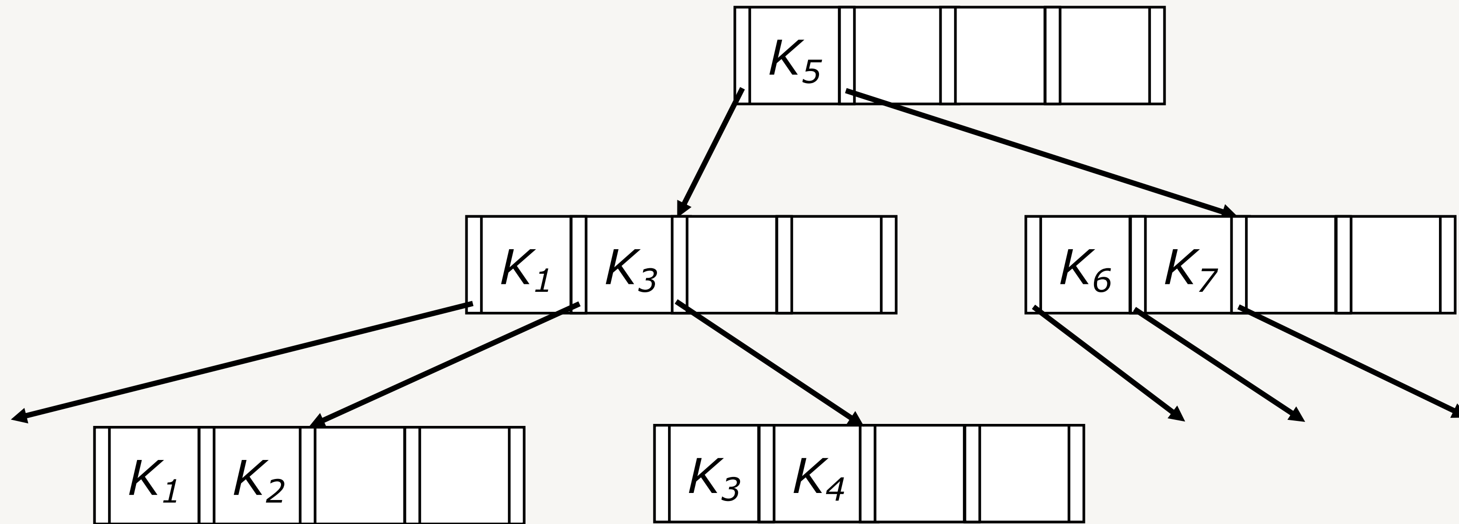


cancellazione K_2: merge

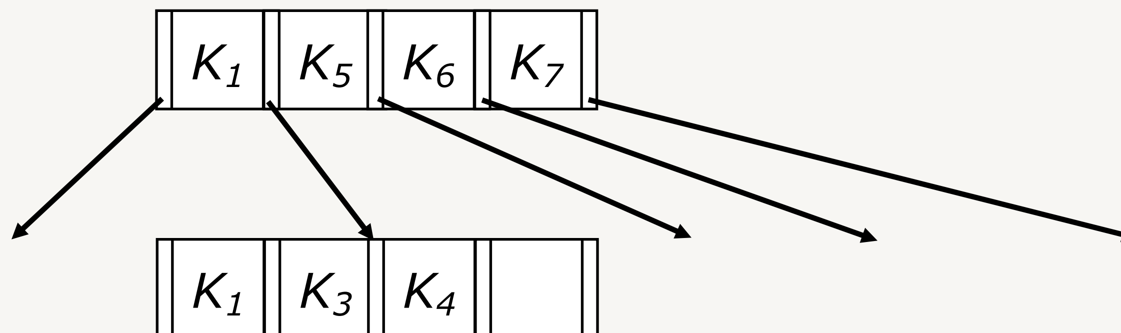


Alberi B+: cancellazione (4)

situazione iniziale



cancellazione K_2: merge



Alberi B+: vantaggi

- forniscono accesso associativo (basato sul valore di una chiave) senza restrizioni sulla posizione fisica delle tuple
- efficienti anche nelle interrogazioni di intervallo
- l' utilizzo attento di split e merge consente di mantenere l'albero ben bilanciato
  - se necessario esplicito ri-bilanciamento

È la struttura utilizzata più frequentemente da DBMS relazionali

Progettazione fisica (1)

Ingresso

- schema logico
- caratteristiche del DBMS scelto
- previsioni sul carico applicativo
- altre informazioni
  - es., dimensioni iniziali dei file, possibilità di espansione

Uscita

- schema fisico della base di dati
  - strutture fisiche di accesso con relativi parametri
  - particolare importanza ha l'individuazione degli indici

Progettazione fisica (2)

Indici:

- definiti su attributi si traducono in speciali strutture di accesso che garantiscono accesso diretto

- ottimizzano accesso agli attributi su cui sono definiti

Progettazione fisica (3)

- la progettazione fisica deve identificare gli indici da definire su ogni relazione
  - indice primario: uno con struttura key-sequenced, che fornisce ricerca sequenziale
    - generalmente unique (es., la chiave primaria della tabella)
  - indici secondari: molti con alberi indiretti
    - su attributi spesso coinvolti in predicati delle interrogazioni
    - su attributi per i quali è spesso richiesto ordinamento in uscita

Progettazione fisica (4)

Individuazione degli indici

- spesso fatta in modo empirico con una fase di regolazione (tuning) che può introdurre nuovi indici per migliorare le prestazioni

- se le prestazioni non sono soddisfacenti, possono essere aggiunti o tolti degli indici

- utile controllare come gli indici sono utilizzati dalle query utilizzando il comando `show plan`

Definizione degli indici in SQL

- i comandi nei sistemi relazionali per la creazione e la cancellazione di indici non sono parte dello standard SQL ma la loro sintassi è abbastanza simile in tutti i DBMS
- sintassi:
  - `create [unique] index` *Indexname* `on` *Tablename(Attributelist)*
  - `drop index` *Indexname*

Esempi:

- `create index CoNoIdx on Impiegato(Cognome,Nome)`
- `drop index CoNoIdx`

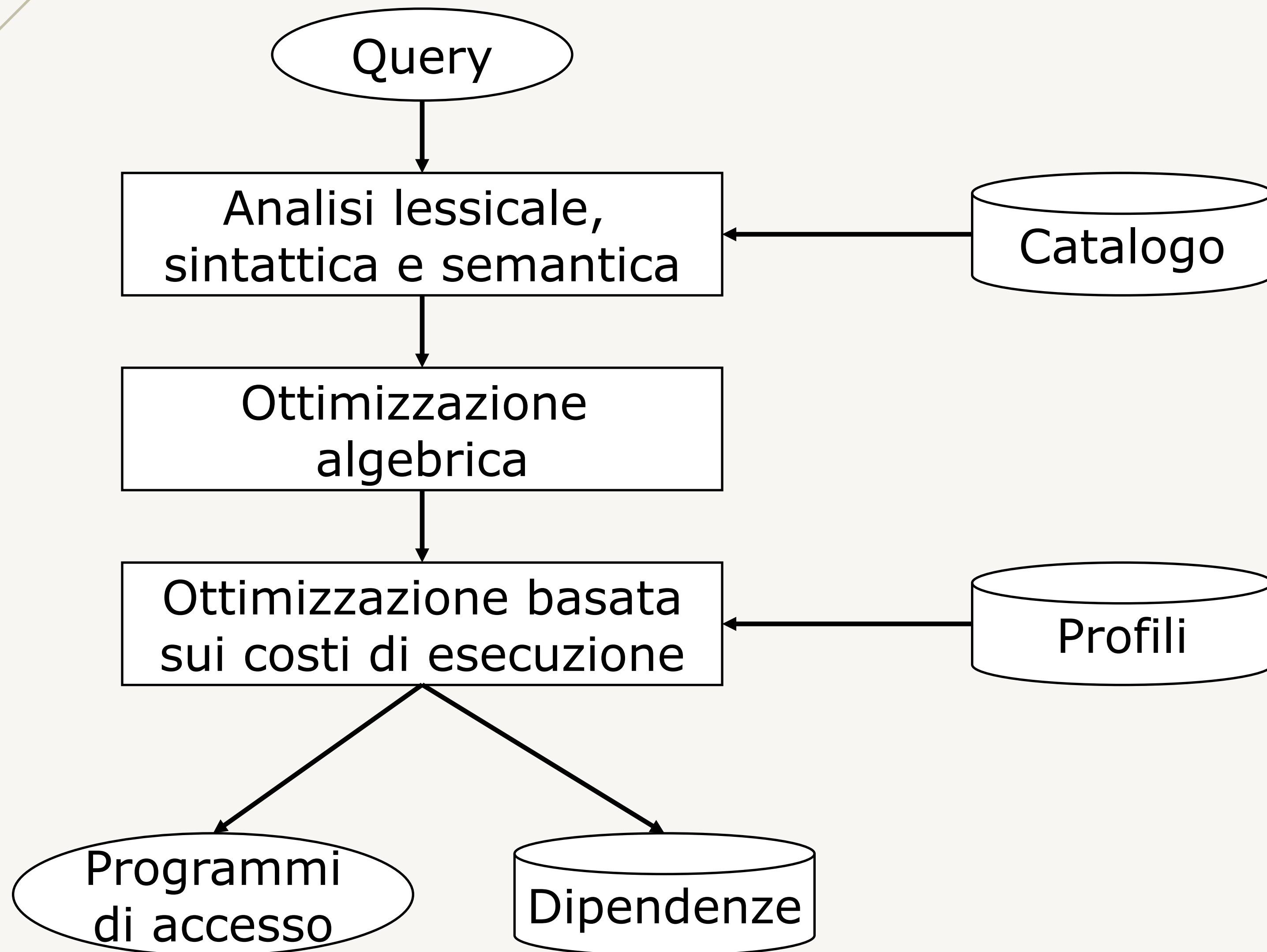
Gestione delle interrogazioni

Input: query SQL

- analisi lessicale, sintattica e semantica
- traduzione in forma algebrica interna e ottimizzazione algebrica
  - es., push down selection
- ottimizzazione basata sui costi di esecuzione
  - dipende dai metodi di accesso e dai profili delle tabelle
- traduzione delle operazioni in termini dei metodi di accesso forniti dal DBMS

Output: programma di accesso in formato interno

Compilazione di una interrogazione



Profili delle tabelle

- contengono informazioni quantitative sulla tabelle
  - cardinalità (numero di tuple) di ogni tabella *T*
  - dimensione in byte di ogni tupla in *T*
  - dimensione in byte di ogni attributo A_j in *T*
  - numero di valori distinti di ogni attributo A_j in *T*
  - valore minimo e massimo di ogni attributo A_j in *T*
- calcolati periodicamente attivando appropriate primitive di sistema (es., `update statistics`)
- sono memorizzati nel dizionario dei dati

Compilazione delle interrogazioni

Due approcci

- compile-and-store: l'interrogazione è compilata una volta e riutilizzata più volte
  - il codice interno è memorizzato nella base di dati, insieme con una indicazione delle dipendenze del codice sulle specifiche versioni delle tabelle e degli indici della base di dati
  - quando ci sono modifiche la compilazione è invalidata e ripetuta
- compile-and-go: esecuzione immediata, l'interrogazione compilata non è memorizzata

LinkedIn vincenzocalabro

www.vincenzocalabro.it