

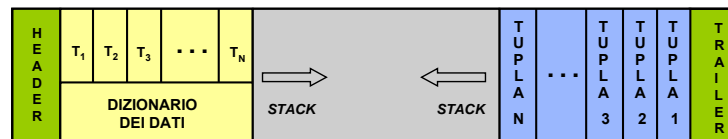
Parte VIII

Organizzazione fisica

Ottimizzazione e organizzazione fisica

- L'*Ottimizzatore* genera i piani esecutivi delle interrogazioni
- Un piano esecutivo rappresenta un'espressione algebrica, e specifica le modalità di esecuzione degli operatori
- Il *Gestore degli accessi* trasforma un piano esecutivo in una sequenza di accessi alle pagine della base di dati
- Sia l'ottimizzazione che l'accesso ai dati dipendono dalla *organizzazione fisica* (come le tabelle sono organizzate e memorizzate nei file):
 - Uso di metodi di accesso (indici)
 - Organizzazione dei dati all'interno delle pagine

Organizzazione della pagina



- Ogni pagina del DB contiene un certo numero di tuple, ed uno spazio libero per consentire inserimenti
- La pagina è composta da:
 - *Block header* e *block trailer* gestiti dal file system
 - *Dizionario di pagina* contiene puntatori ai dati all'interno della pagina (gestito a stack)
 - *Parte utile* contenente i dati (gestita a stack)

N.B. Non sempre è consentito avere tuple di lunghezza variabile e spezzare una tupla su più pagine

Primitive del gestore delle pagine

- *Inserzione e aggiornamento*, con eventuale riorganizzazione della pagina, svolta comunque in memoria centrale
- *Cancellazione*: tramite *invalidazione* della tupla
- *Accesso ad una tupla*
 - Associativo: tramite il valore della chiave
 - In base al suo offset
- *Accesso ad un campo di una tupla*, calcolato in base allo offset e alla lunghezza dei campi

N.B. È possibile avere nella stessa pagina tuple di relazioni diverse (clustering)

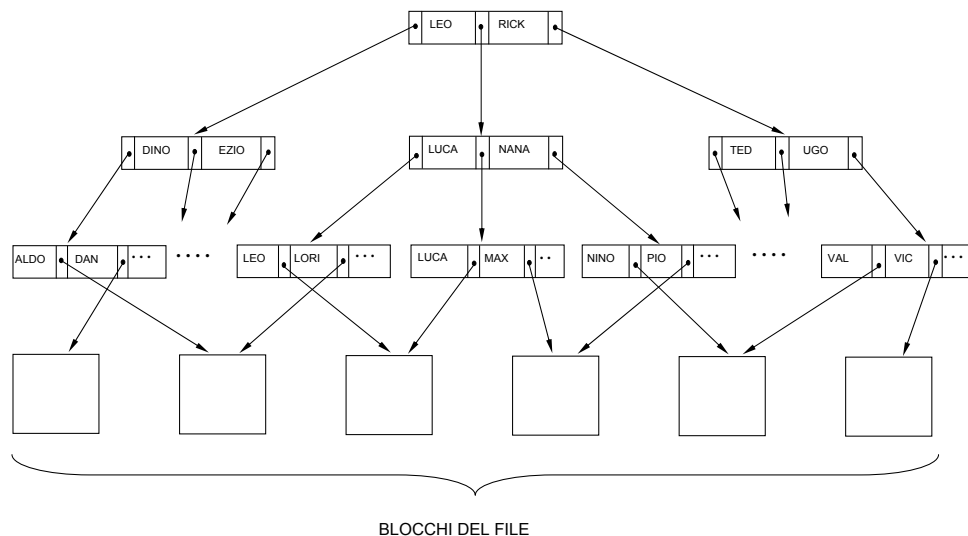
Strutture sequenziali

- Disposizione sequenziale delle tuple. Il file è costituito da blocchi consecutivi. Vari tipi di organizzazione
- *Entry-sequenced*: tuple in ordine di immissione
 - Utilizzata per *scan* sequenziali
 - Inserzioni e cancellazioni gestite lasciando i blocchi solo parzialmente pieni.
- *Sequenziale ad array*: tuple di dimensione fissa
 - File di n blocchi ciascuno con m tuple
 - Tuple accedute anche tramite il valore dell'indice

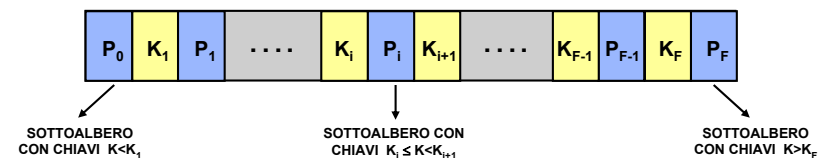
Accesso calcolato: indici hash

- Accesso *associativo* ai dati: si calcola la posizione della tupla a partire dal valore della chiave
- Al file vengono allocati B blocchi
- Per inserire e per accedere ad una tupla di chiave k si utilizza una funzione *hash*
- $h(k)$ con codominio $[0 \dots B]$ calcola il blocco in cui la tupla di chiave k è contenuta
- Si sovradimensiona il file per evitare il trabocco dei blocchi a causa delle *collisioni*
- Metodo più sofisticati: *hashing dinamico*

Indici B-tree



Indici B-tree: struttura



- Sono alberi m -ri bilanciati di ricerca i cui nodi sono blocchi allocati su memoria di massa
- Ciascun nodo contiene una sequenza ordinata di F chiavi K_1, K_2, \dots, K_F e $F + 1$ puntatori a sottoalberi:
- $F + 1$ è detto *fan-out* dell'albero
- La ricerca ha profondità logaritmica (in base $F + 1$): parte dalla radice e confronta ad ogni passo la chiave k da ricercare con quelle del nodo corrente.

B-tree: organizzazione delle foglie

A) *Key-sequenced* : tuple sono allocate nei nodi foglia

- Utilizzata per gli *indici primari*
- Indici definiti come **unique** sulla primary key

B) *Indiretta*: i nodi foglia contengono i puntatori ai blocchi contenenti le tuple

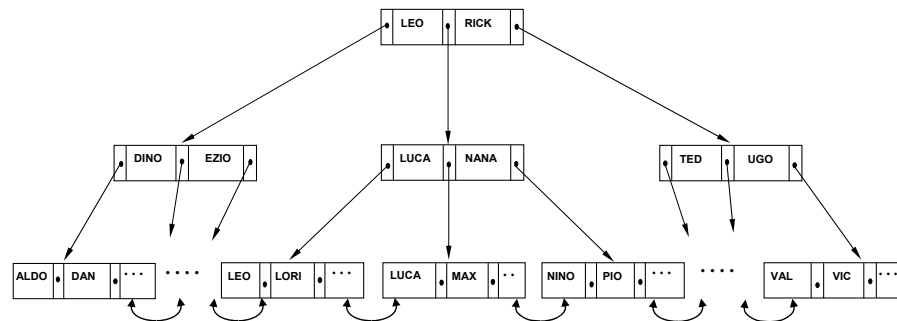
- Utilizzata per *indici secondari*
- Tuple allocate in base ad un indice primario

N.B. Su strutture sequenziali ordinate è possibile costruire *indici sparsi* cioè indici sulle prime tuple di ciascun blocco

B-tree: inserzioni e cancellazioni

- Cancellazioni ed inserzioni sulle tabelle provocano sempre anche aggiornamenti dell'indice.
- Se in inserzione il nodo del B-tree non ha più spazio libero si effettua uno *split* del blocco
- A seguito di una cancellazione, si può dover effettuare il *merge* di due blocchi
- Inserzioni e cancellazioni possono anche provocare una variazione nel numero di livelli (costo di accesso)
- Per mantenere l'albero bilanciato, si mantiene il livello di riempimento dei nodi fra $F/2$ e F

B+tree



- Nei B+tree le foglie sono legate in una catena che le collega in base all'ordinamento della chiave
- Ciò è vantaggioso nel caso delle *range queries*, in cui il predicato di selezione è su di un intervallo

B-tree: costo di accesso

- Per accedere ad una tupla di chiave data, è necessario un accesso per ogni livello del B-tree
- Il costo di accesso dipende quindi dalla profondità dello albero, e quindi dal fattore di ramificazione (*fan-out*) $F+1$
- Per calcolare la dimensione, ed il numero di livelli di un B-tree consideriamo i seguenti parametri
 - **N**: numero di tuple nella tabella indicizzata
 - **B**: dimensione di una pagina o blocco
 - **k**: lunghezza di una chiave
 - **t**: lunghezza di una tupla
 - **p**: lunghezza di un puntatore
 - **u**: coefficiente medio di riempimento dei nodi

Dimensioni e profondità dell'albero

- Numero medio di tuple per foglia

– Key-sequenced $C = \lfloor B/(k+t) \rfloor \times u$

– Indiretto $C = \lfloor B/(k+p) \rfloor \times u$

- Fan-out medio $F = \lfloor B/(k+p) \rfloor \times u$

- Profondità media dell'albero $H = \lceil 1 + \log_F (N/C) \rceil$

- Dimensione in pagine dell'albero

$$S = \sum_{i=0}^H F^{i+1}$$

Piani esecutivi

- L'ottimizzatore produce un piano esecutivo rappresentabile con un albero, secondo il quale la query viene poi valutata
- Le foglie sono le tabelle ed i nodi interni specificano le modalità con cui gli accessi alle tabelle e le operazioni relazionali sono effettuate
- Le operazioni elementari supportate dai DBMS relazionali tipicamente sono:
 - *Scansione sequenziale*
 - *Accesso tramite indice*
 - *Join (con varie modalità)*
 - *Ordinamento*

Scansione sequenziale

- Le tuple della tabella vengono accedute *sequenzialmente*
- Vengono letti e bufferizzati più blocchi contigui (per ridurre i tempi di accesso)
- Contestualmente alla scansione possono essere eseguite varie operazioni:
 - Selezione: filtraggio su una condizione sulla tupla
 - Proiezione su una lista di attributi;
 - Ordinamento su una lista di attributi;
 - Aggiornamento: inserzioni, cancellazioni e modifiche sulle tuple man mano che vengono accedute

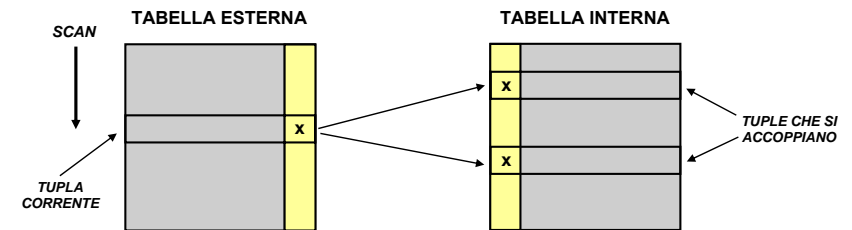
Ordinamento

- *Sort-merge a più vie*, con parallelismo che dipende dal numero m di pagine nel buffer dedicate all'operazione
- Data una tabella R_i di B_i blocchi i singoli blocchi vengono ordinati a m per volta
- Successivamente si effettuano più passate, ciascuna delle quali consta in un merge a $m-1$ vie che produce $m-1$ file, che costituiscono l'input della passata successiva
- In tutto le passate sono $\log_{m-1} B_i$, e ciascuna comporta una lettura ed una scrittura per ogni blocco di R_i con un numero di accessi complessivo di $2 B_i \log_{m-1} B_i$

Accesso tramite indice

- Un indice su di un attributo A_i può essere sfruttato per valutare predicati del tipo $A_i = V$ o del tipo $V_1 \leq A_i \leq V_2$, che si dicono *valutabili* tramite l'indice
- Se c'è una *congiunzione* di più predicati si utilizza l'indice corrispondente al predicato che si ritiene più selettivo, e poi si applicano gli altri predicati alle tuple così accedute
- Se c'è una *disgiunzione* o sono tutti valutabili oppure tanto vale fare un'unica scansione sequenziale
- Nel caso di predicati poco selettivi può convenire una scansione sequenziale

Join nested-loop

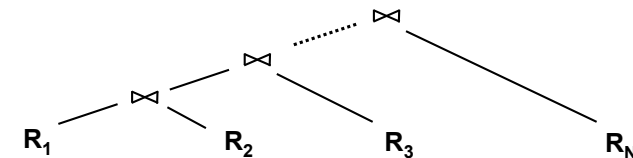


- Si ha una *tabella esterna* R_E e una *tabella interna* R_I
- Si effettua una scansione sulla tabella esterna, e per ogni tupla si cercano le tuple della tabella interna che possono essere accoppiate con essa
- Accesso alla tabella interna con indice, se possibile

Costo del join nested-loop

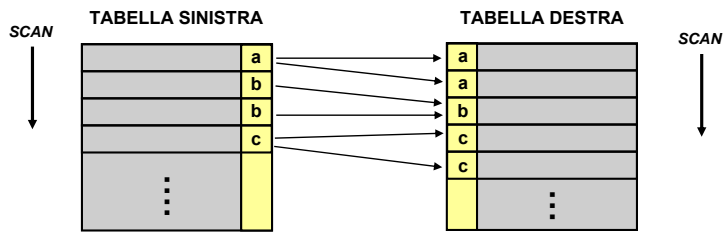
- Ha costo lineare o quadratico a seconda che ci sia o meno un indice su R_1 su almeno uno degli attributi di join
- A) Indice su R_1 :**
- Si apre uno scan su R_E
 - Per ogni tupla di R_E si cercano tramite l'indice le tuple di R_I che si possono accoppiare con essa
- B) Nessun indice sugli attributi di join:** occorre fare una doppia scansione. Se R_E e R_I hanno rispettivamente B_E e B_I blocchi
- Si dedicano m pagine nel buffer al join,
 - Vengono letti $m - 2$ blocchi di R_E per volta
 - Per essi si effettua una intera scansione di R_I
 - Si hanno in totale $\lceil B_E / (m - 2) \rceil B_I$ accessi

Esecuzione di join in cascata



- Si può effettuare una esecuzione in pipeline
- Sono necessari indici sugli attributi di join su $R_2 \dots R_n$
- Si effettua il join nested-loop tra R_1 e R_2 , e poi si passa ciascun blocco del risultato intermedio così prodotto, per effettuarne il join con indice con R_3 , e così via
- Eventuali condizioni di selezione e proiezione sugli operandi vengono applicate durante l'esecuzione del join
- L'ordine in cui i join vengono eseguiti può influire sul costo

Join merge-scan



- Occorre che entrambe le tabelle siano ordinate in base agli attributi di join
- Nel caso si può effettuare un passo preliminare di sort
- Si aprono in parallelo due scansioni sulle due tabelle e si confrontano le tuple, come se si facesse un *merge*
- Il costo è lineare

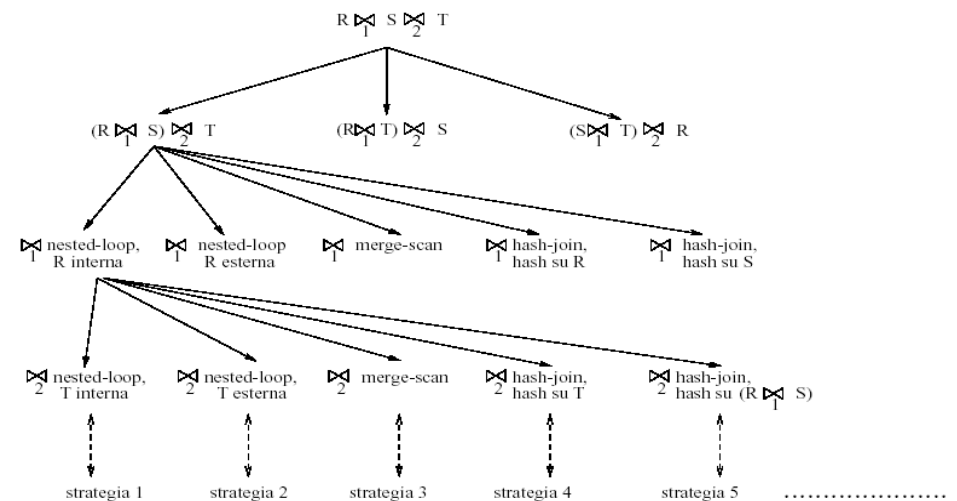
Ottimizzazione basata sui costi

- Nel selezionare il migliore piano esecutivo per una data query l'ottimizzatore opera scelte su più dimensioni, fra loro interconnesse:
 - Tipo di accesso ai dati: scan sequenziale, accesso con indici
 - Ordinamento degli operatori: il costo dipende anche dalla dimensione dei risultati intermedi
 - Metodo: quando più di un metodo è disponibile per la stessa operazione
 - Metodi di join
 - Scegliere a che livello effettuare un ordinamento e con che metodo effettuarlo

Albero delle alternative

- L'ottimizzatore costruisce un *albero di scelta* in cui ogni nodo corrisponde a fissare una particolare opzione
- A ciascuna *foglia* dell'albero corrisponde un *piano esecutivo* definito dal cammino tra la radice e la foglia
- L'ottimizzatore tipicamente effettua una visita parziale dell'albero, usando formule di costo approssimate
- Si cerca di scartare il più presto possibile interi sottoalberi sottoalberi, e si usano strategie di *branch and bound*, sia esatte che approssimate
- Nel caso di query compilate fuori linea e poi eseguite molte volte, ha senso investire molto tempo nella ottimizzazione

Albero delle alternative (esempio)



Progettazione fisica

- È la fase finale della progettazione: a partire dallo schema logico, dalle previsioni sul carico applicativo e dalle caratteristiche e dalla configurazione del sistema prescelto, produce in uscita lo *schema fisico*
- Occorre stabilire:
 - Per ciascuna tabella come e dove viene memorizzata
 - Dimensioni iniziali del file, vincoli di allocazione etc
 - Su quali attributi definire indici e di che tipo
 - Parametri relativi alla gestione del buffer, del controllo della concorrenza e dell'affidabilità

N.B. Il DBMS offre comunque accettabili valori di default

Scelta degli indici

- Occorre analizzare il carico e individuare gli attributi coinvolti in operazioni di join e di selezioni
- Si predispongono indici sugli attributi di join, in modo da rendere possibili piani esecutivi vantaggiosi
- Gli attributi chiave sono spesso coinvolti in join e selezioni, e pertanto su di essi tipicamente vengono predisposti indici.
- Considerare anche gli aspetti negativi:
 - L'occupazione di un indice può essere dello stesso ordine, o addirittura superare quella della tabella
 - Gli aggiornamenti comportano operazioni costose sugli indici, oltre che sulla tabella

Tuning delle applicazioni: analisi

- Fase molto delicata di cui necessitano tutte le applicazioni con requisiti o problemi di prestazioni.
- In sede di analisi è possibile stabilire:
 - Tramite il *monitor* del DBMS il piano esecutivo scelto dall'ottimizzatore per ciascuna interrogazione (albero esecutivo, metodi di accesso alle tavole, metodi di join)
 - Tramite il *monitor* del DBMS gli accessi *logici* e *fisici* effettuati in lettura e scrittura alle tabelle e agli indici da ciascuna interrogazione
 - Tramite il *monitor* del sistema operativo gli accessi effettivamente (al netto degli effetti della *buffer cache*) ai vari dischi, le loro utilizzazioni e i valori medi dei tempi di accesso e di attesa

Tuning delle applicazioni: sintesi

- In base ai risultati dell'analisi si possono stabilire azioni di vario tipo per migliorare le prestazioni
 - Aggiunta o ridefinizione di indici, per rendere possibili diversi piani esecutivi e/o metodi di join
 - Riaggiustamento dei parametri relativi al buffer, allo scopo di migliorare il rapporto tra I/O logico e I/O fisico;
 - Riallocazione dei file fra le unità a disco, per bilanciare le utilizzazioni
 - Eventuali ristrutturazioni dello schema logico;
- N.B.** Spesso il problema è di *forzare* l'ottimizzatore ad adottare i piani esecutivi che si sta cercando di rendere possibili