Appunti della lezione del **Mercoledì 20/05/2020**

Gabriele Frassi

A.A 2019-2020 - Secondo semestre

Indice

1 Organizzazione fisica e gestione della memoria

1.1 Memoria principale, memoria secondaria e gestione dei buffer

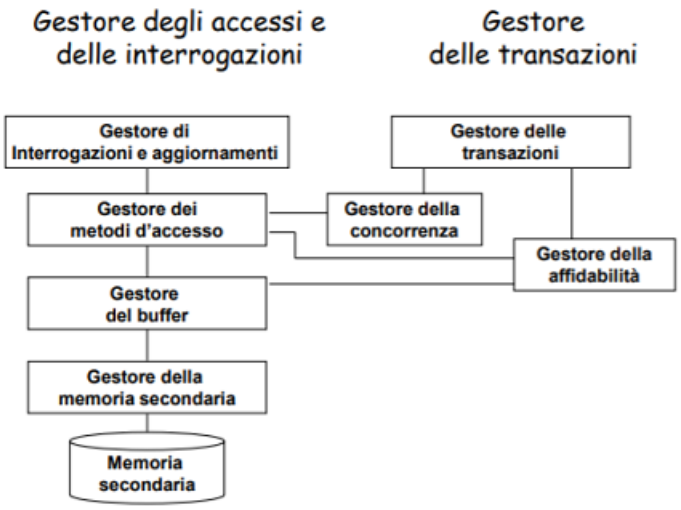
Il DBMS è una *scatola nera*. Non è strettamente necessario conoscere la sua struttura per poterne comprendere il funzionamento, ma in certe circostanze può essere utile.

Ricordiamo Un DBMS gestisce collezioni di dati grandi, persistenti, condivisi. Garantisce affidabilità e privatezza. Deve essere anche efficiente (ridurre i tempi e sfruttare al meglio le risorse) ed efficace (rendere produttiva l'attività di ogni singolo utente).

Grandezza e persistenza Il fatto che i dati siano grandi e persistenti richiede una gestione sofisticata della memoria secondaria. Gli utenti vedono il modello logico, ma le strutture che si celano dietro devono essere gestite efficientemente. Adotteremo delle strutture dati opportune.

Affidabilità La base di dati deve essere preservata, come già detto, anche in caso di malfunzionamento. L'affidabilità è impegnativa per via di aggiornamenti frequenti e per la gestione del buffer.

Condivisione Una base di dati è una risorsa condivisa fra più applicazione. Sono previsti meccanismi di autorizzazione e il cosiddetto controllo di concorrenza per gestire attività diverse e multi-utente su dati condivisi.



Tecnologie presenti

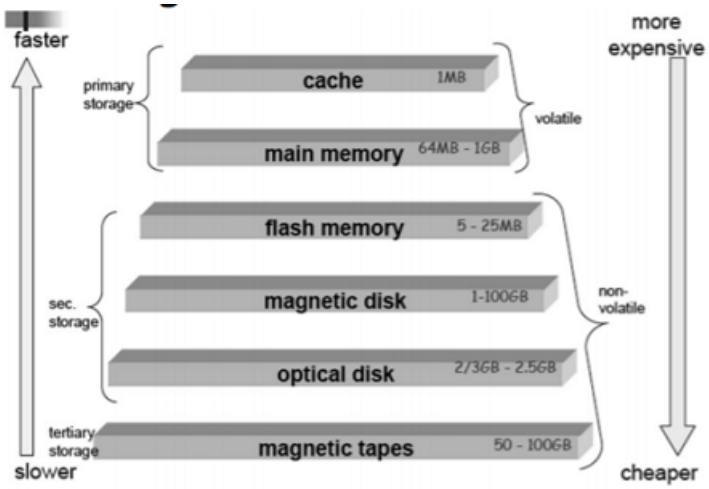
- Gestione della memoria secondaria e del buffer
- Organizzazione fisica dei dati
- Ottimizzazione delle interrogazioni (sfruttando il concetto di equivalenza e sfruttando dati a nostra disposizione per determinare la miglior forma di ottimizzazione)

File system e DBMS

- 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/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.

Ricordare Il programma non può entrare in contatto con la memoria secondaria ma solo con quella principale. Abbiamo il buffer che permette un'interazione fra memoria principale e secondaria, limitando il più possibile gli accessi alla secondaria.

Gerarchia di memoria Abbiamo una serie di memoria, ciascuna con tecnologie e costi diversi. Si distinguono le memoria primarie, volatili, dalle memorie secondarie e terziarie, permanenti. Ovviamente il costo di mantenimento di una memoria di tipo permanente è più elevato.



Nell'accesso a memoria secondaria abbiano una testina che deve raggiungere il punto opportuno. Si considerano i seguenti tempi:

- tempo di posizionamento della testina (10-50ms)
- tempo di latenza (5-10ms)
- tempo di trasferimento (1-2ms)

In media non meno di 10ms. Si osserva che l'accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria principale. Segue che il costo delle operazioni dipende moltissimo dal numero di accessi in memoria secondaria.

Organizzazione delle memorie Ricordiamo che le memorie secondarie sono organizzate in blocchi di lunghezza fissa, mentre la memoria principale è organizzata in pagine. Nella memoria secondaria le uniche operazioni possibili sono quelle di lettura e scrittura dei dati di un blocco: in riferimento alla latenza (citata prima) si osserva che l'accesso a blocchi vicini è molto meno costoso (latenza a 0).

1.2 Buffer

Il buffer è un'area di memoria centrale gestita dal DBMS e condivisa fra le transazioni. Risulta organizzato in pagine di dimensioni pari o multiple di quelle dei blocchi di memoria secondaria (ricordiamoci che siamo in memoria principale). Il caricamento di una pagina del buffer equivale a un'operazione di lettura in memoria secondaria, mentre salvare una pagina equivale a un'operazione di scrittura.

Buffer manager Questo, oltre al buffer, si occupa di una directory che per ogni pagina mantiene il file fisico e il numero del blocco, e 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.

Si considera:

- l'alta probabilità di dover riutilizzare i dati attualmente in uso
- la legge di Pareto: l'80% delle operazioni utilizza sempre lo stesso 20% dei dati.

Il buffer si occupa di:

- di ricevere richieste di lettura e scrittura dalle transazioni
- di eseguire queste operazioni accedendo alla memoria secondaria solo quando indispensabile

Ricordiamo, a proposito di queste richieste, le primitive:

- fix: richiesta di pagina. Richiedo 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 (decremento il contatore associato alla pagina)
- force: trasferisco una pagina in memoria secondaria (su richiesta del gestore dell'affidabilità, non del gestore degli accessi)

Esecuzione della fix

- cerco la pagina nel buffer
- se la trovo restituisco l'indirizzo
- se non la trovo cerco una pagina libera nel buffer (contatore 0):
  - se la trovo inserisco i dati letti dalla memoria secondaria e ne restituisco l'indirizzo
  - se non la trovo ho due opzioni:
    - seleziono una pagina *vittima* occupata dal buffer, scrivo i dati di questa in memoria secondaria, effettuo le letture in memoria secondaria e restituisco l'indirizzo
    - pongo in attesa l'operazione

Scopo della gestione del buffer Vogliamo ridurre il numero di accessi alla memoria secondaria:

- in caso di lettura, se la pagina è già presente nel buffer non c'è bisogno di accedere alla memoria secondaria
- in caso di scrittura, il gestore del buffer può decidere di differire la scrittura fisica in modo da accorparla ad altre scritture.

Scrittura in memoria secondaria 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, cioè quando lo ritiene opportono. Si possono anticipare o posticipare scritture per coordinarle e/o sfruttare la disponibilità dei dispositivi.

1.3 Gestione delle tuple nelle pagine

Tuple e blocchi

- Il file system ha i suoi file: questi 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)
- 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.

Organizzazione delle tuple Ho varie alternative possibili. Solitamente inseriamo le tuple in modo sequenziale nei file. Inoltre:

- se la lunghezza delle tuple è fissa, la struttura può essere semplificata
- alcuni sistemi possono spezzare le tuple su più blocchi (tuple grandi, cosa difficile da gestire)

Fattore di blocco Attraverso il fattore di blocco stabilisco quanti record posso porre 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 per record *spanned* o non utilizzato.

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. Risolviamo così:

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

Operazioni sulla pagina

- Inserimento/Modifica di una tupla (la cosa può richiedere allocazione di ulteriore spazio)
- Cancellazione
- Accesso ad una tupla o ad un campo di una tupla

1.4 Strutture per l'organizzazione di file

Definizioni

- Le tuple organizzate all'interno dei blocchi dei file costituiscono le **strutture primarie**, cioè quelle che contengono propriamente i dati. Le principali sono
  - strutture sequenziali (seriali / ordinate / ad array)
  - strutture ad accesso calcolato (hash)
  - strutture non sequenziali ad albero
- Esistono anche blocchi contenenti **strutture secondarie**, che sono quelle che favoriscono l'accesso ai dati senza contenerli (in un certo senso sono dei puntatori ai dati).

1.4.1 Strutture primarie sequenziali

Una struttura primaria sequenziale può essere

- **seriale** (detto anche *heap*, miscuglio): ordinamento fisico ma non logico. Gli inserimenti vengono effettuati:
  - in coda (gli elementi sono di fatto posti nell'ordine di inserimento)
  - al posto di record cancellati (che ho precedentemente sostituito con una *marca* che mi segnala la cancellazione)

  L'eliminazione, ma anche la sostituzione di una tupla con un'altra di dimensione minore, comportano uno spreco di spazio e un conseguente aumento del numero di blocchi utilizzati: segue la necessità di svolgere riorganizzazioni periodiche. Una ricerca è di tipo sequenziale: segue una complessità lineare e la necessità di scorrere, nel caso peggiore, tutto il file.
- **ordinata**: ordinamento fisico delle tuple coerente con quello di un campo detto chiave (per esempio, data una lista di studenti universitari, la matricola)
  - Sono possibili ricerche binarie

  Le riorganizzazioni periodiche sono necessarie anche in questo tipo di organizzazione. Si ha un'ulteriore problema: dobbiamo mantenere l'ordinamento a seguito di un qualunque cambiamento. Ciò rende le riorganizzazioni decisamente più importanti e complesse.
- **array**: le tuple hanno stessa dimensione, all'interno di uno o più blocchi contigui abbiamo una serie di posizioni identificate da indici.
  - Il caricamento iniziale delle informazioni avviene mediante incremento di un contatore e inserimento nelle varie posizioni
  - La cancellazione lascia spazi vuoti
  - Gli inserimenti possono essere effettuati solo in posizioni vuote o in fondo al file.

  Generalmente questa struttura non è quasi mai utilizzata nei DBMS.

1.4.2 Strutture primarie ad accesso calcolato

Questa struttura, detta anche *hash*, sfrutta le proprietà tipiche dell'organizzazione sequenziale ad array anche in circostanze in cui l'array non risulta immediatamente applicabile. L'array, normalmente, è ottimo, per organizzare insiemi di record che hanno come *campo chiave* valori consecutivi (esempio numero di matricole da 1 a 10000): questa cosa, in moltissime situazioni, non è possibile.

- i file hash permettono un accesso diretto molto efficiente.

- Ricordiamo che la 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)

Come risolvo le collisioni? Esistono varie tecniche:

- Occupare le posizioni successive
- Ricorrere a tabelle/catene di overflow (gestita in forma collegata)
- Adottare funzioni hash alternative

Osserviamo che:

- le collisioni sono quasi sempre presenti (tenendo conto che non possiamo avere funzioni iniettive nella maggior parte dei casi)
- le collisioni multiple hanno probabilità che decresce al crescere della molteplicità
- la molteplicità media delle collisioni è molto bassa

Hash su file Ricordiamo l'organizzazione di un file: esso è diviso in blocchi, e questi a loro volta presentano all'interno delle tuple. La funzione hash restituisce un indice rappresentante il blocco all'interno del quale è presente la tupla incriminata. Questo ci permette di ammortizzare le probabilità di collisione

- In caso di conflitto il record è memorizzato nello stesso blocco dove sono presenti altre tuple
- Quando lo spazio di un blocco è esaurito viene allocato un ulteriore blocco, collegamento al precedente, e il record viene posto al suo interno. Questa tecnica è detta *catena di overflow*.
- Segue che per una parte di tuple, associate a un certo indice, sarà necessario un solo accesso a blocco. Per questa tupla, appena aggiunta, invece, serviranno due accessi!

Esempio di esercizio

- Prendiamo la seguente tavola hash (con collisioni) e facciamo delle analisi relative all'*hash su file*

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

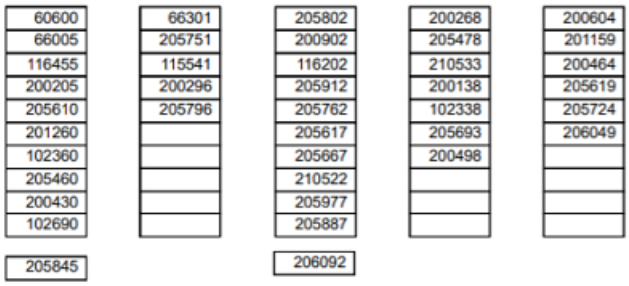
M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

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

- Consideriamo:
  - T il numero di tuple previsto per il file
  - F il fattore di blocco
  - f il fattore di riempimento (la frazione dello spazio fisico disponibile mediamente utilizzata)
- Svolgiamo il seguente calcolo per trovare il numero di blocchi utilizzato

$$B = \frac{T}{f \times F} = \frac{40}{\frac{4}{5} \times 10} = 5$$

- Abbiamo 5 blocchi con 10 posizioni ciascuno!
- Prendiamo la seguente figura, che contiene gli stessi dati della precedente tavola hash



Complessivamente abbiamo 30 chiavi: per raggiungere due chiavi è necessario l'accesso a due blocchi, uno solo per le rimanenti 28.

1.4.3 Strutture ad albero (non sequenziali, dette *indici*)

Le strutture ad albero, come quelle *hash*, favoriscono l'accesso ai dati in base al valore di uno o più campi. Sono consentiti accessi

- puntuali
- corrispondenti a intervalli di valori

L'organizzazione ad albero, come vedremo, può essere utilizzata per realizzare sia strutture primarie che secondarie.

Cosa intendiamo con indice? Supponiamo di avere un file f e un campo chiave (che può avere al suo interno più attributi): un **indice secondario** consiste in un altro file dove ciascun record è logicamente composto da due campi:

- uno contenente il valore del campo chiave
- uno contenente l'indirizzo o gli indirizzi fisici dei record del file f che hanno quel valore di chiave

Un esempio per capire in cosa effettivamente consiste un indice secondario è l'indice analitico presenti in fondo a certi libri: ho un ordine alfabetico dei termini presenti e le pagine dove sono presenti tali termini. Un indice si dice **indice primario** se l'ordinamento è lo stesso della struttura dati o se contiene al suo interno i dati: in questo caso non garantisco solo un accesso in base a un campo chiave, ma contengo anche i record fisici necessari per memorizzare i dati. Un esempio per comprendere l'indice primario è l'indice generico presente in ogni libro: ho le sezioni e il numero delle pagine dove queste sono posizionate.

File Un file può avere un solo indice primario (questo non è presente solo quando l'organizzazione primaria è hash oppure sequenziale) e più indici secondari.

Puntatori in un indice I puntatori di un indice reindirizzano

- al blocco dove è presente la tupla, o
- alla tupla stessa.

il tempo per effettuare la ricerca di una tupla all'interno di un blocco è trascurabile. Questo ci permette di limitare, in certi casi, il numero di puntatori per blocco: potrei puntare alla tupla contenente valore della chiave minore o maggiore.

Indici densi e sparsi Consideriamo gli indici con puntatori a tuple. Essi saranno detti:

- **densi**, se tutte le tuple dei vari blocchi sono puntate da elementi dell'indice (solitamente quando adottiamo un'ordinamento diverso da quello fisico e quindi non è più possibile garantire la presenza di una tupla all'interno di un blocco).
- **sparsi**, se non tutte le tuple sono puntate (solitamente quando seguiamo l'ordinamento fisico, quindi puntando a una certa tupla di un blocco - rappresentante un valore di riferimento - sono certo che ciò che sto cercando sarà all'interno di quella tupla)

Gli indici secondari sono per forza densi, poichè basati su criteri di ordinamento diversi da quello fisico. L'indice primario può essere sparso.

Ricerche Il fatto di avere un ordinamento permette di svolgere ricerche binarie con complessità logaritmica. Sono possibili anche ricerche basate su intervalli e scansioni sequenziali ordinate (cose inefficienti sulle strutture hash).

Caratteristiche degli indici

- Accesso diretto ed efficiente sulla chiave
- Modifiche della chiave, inserimenti, eliminazioni inefficienti (come nei file ordinati). Possiamo adottare le seguenti tecniche per migliorare la situazione:
  - file o blocchi di overflow
  - marcatura per le eliminazioni
  - riempimento parziale
  - blocchi collegati (non contigui)
  - riorganizzazioni periodiche

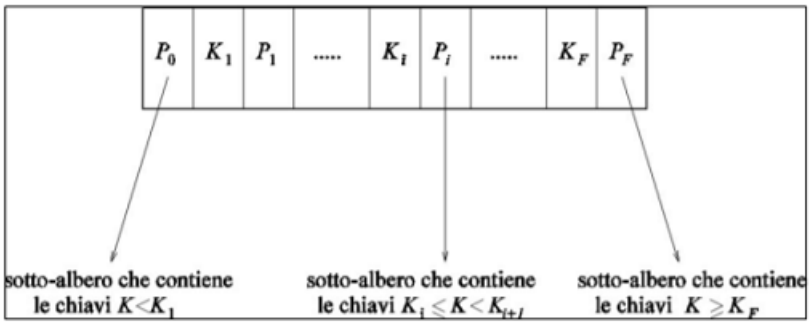
Segue una scarsa flessibilità in presenza di elevata dinamicità.

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
  - ogni nodo (al di fuori delle foglie) ha un numero di discendenti abbastanza elevato, dipendente dall'ampiezza della pagina.

- l'indice è ordinato, quindi l'indice sull'indice è sparso
- i record dell'indice sono piccoli

Sfruttando questi concetti andiamo a costruire degli alberi dove ciascun nodo consiste in un file, quindi in un indice!



Indici e alberi binari

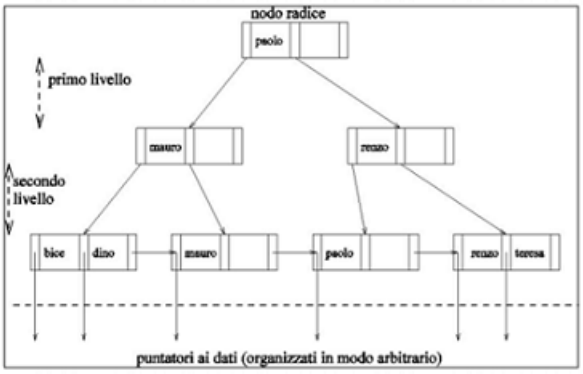
- Gli indici utilizzati dai DBMS sono in generale
  - indici dinamici multilivello
  - Vengono memorizzati e gestiti come B-tree (intuitivamente: alberi di ricerca bilanciati): Alberi binari di ricerca, Alberi n-ari di ricerca, Alberi n-ari di ricerca bilanciati

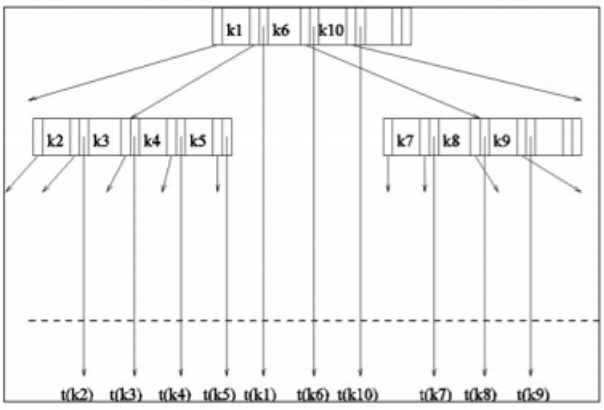
Si ricorre ad alberi generici di ordine P (dove ciascun nodo presenta fino a P figli e $P-1$ etichette). Questi alberi saranno tenuti bilanciati nel tempo grazie a:

- Riempimento parziale (mediamente 70%)
- Riorganizzazioni (locali) in caso di sbilanciamento

Si distingue una versione B-tree da una B+-tree:

- nel B+-tree a le foglie costituiscono una lista e sono collegate fra di loro
- nel B-tree possiamo avere, in nodi intermedi, puntatori diretti ai dati.





Entrambi gli alberi sono molto efficienti: la forma preferita è la B+-tree in quanto permette di svolgere ricerche basate su un intervallo di valori. Con le foglie collegate posso effettuare selezioni basate su un intervallo di valori: individuo il primo valore dell'intervallo e poi effettuo una scansione sequenziale per trovare i valori maggiori rispetto a questo.

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`

Diapositive Vedere attorno alla diapositiva 66 le strutture fisiche presenti in alcuni DBMS

2 Esecuzione e ottimizzazione delle interrogazioni

Riprendiamo un discorso iniziato con l'algebra relazionale. Abbiamo visto il *query processor*, un modulo del DBMS nel quale avviene un processo di ottimizzazione basato su equivalenze (e sul catalogo, che contiene una serie di informazioni utili per l'esecuzione[1]). Le interrogazioni sono espresse ad alto livello (ricordare il concetto di indipendenza dei dati): insiemi di tuple, poca proceduralità.

Fase finale di ottimizzazione Abbiamo lasciato in sospeso l'ultima delle tre fasi del *query processor*. In questa abbiamo un'**ottimizzazione in base ai costi**: consideriamo il numero di trasferimenti in memoria da fare e stimiamo le dimensioni dei risultati intermedi. Nell'ottimizzazione dobbiamo tener conto anche

- delle operazioni da eseguire (vedere paragrafo dopo)
- i dettagli del metodo (per esempio quale tipo di JOIN eseguire)
- ordine delle operazioni (ho un join di tre relazioni, da dove mi conviene partire?)

Implementazione degli operatori dell'AR Le interrogazioni sono svolte mediante un linguaggio ad alto livello, cioè lontano dalla macchina. I DBMS implementano gli operatori dell'algebra relazionale per mezzo di operazioni di livello abbastanza basso. Abbiamo

- **Operatori fondamentali**: accesso diretto e scansione
- **A livello più alto**: ordinamento
- **A livello ancora più alto**: JOIN, operazione più costosa (che coinvolge, ricordiamoci, il prodotto cartesiano).

2.1 Accesso diretto

Fare *accesso diretto* significa ottenere, dato il valore di un campo, l'indirizzo del blocco in cui il record si trova. Ciò può essere fatto solo con strutture hash o indici. I tipi di accessi sono:

- puntuali, del tipo $A_i = V$
- su intervallo, del tipo $V_1 \leq A_i \leq V_2$

Il primo tipo di accesso è efficiente in entrambe le strutture (in particolare in quelle hash), il secondo è efficiente solo negli accessi basati su indice.

[1] Esempi di profili:

- cardinalità di ciascuna relazione
- dimensioni delle tuple e dei valori degli attributi
- numero di valori distinti degli attributi
- valore minimo e massimo di ciascun attributo

Interrogazioni Nelle interrogazioni consideriamo predicati congiuntivi e disgiuntivi. Se si ha un solo predicato predicato valutabile conviene usare indice o hash. In caso contrario:

- **predicati congiuntivi**[2]: prima scelgo il più selettivo (quello che mi leva più tuple), dopo verifico gli altri scorrendo quanto salvato in memoria principale (cioè le tuple che rispettano il primo predicato). Nel caso hash pongo come argomento della funzione il cognome e verifico l'altra condizione.
- **predicati disgiuntivi**[3]: se tutti i predicati sono valutabili utilizzo gli indici per ogni elemento. Sono convenienti, rispetto a una scansione, solo se molto selettivi. Bisogna fare attenzione anche ai duplicati. Il caso nel footnote è impossibile in hash.

2.2 Scansione

La scansione consiste in un'operazione di ricerca: è implementabile tramite un algoritmo di ricerca la cui complessità media, date n tuple, è lineare. **I blocchi vengono trasferiti dalla memoria secondaria al buffer**.

Metodo alternativo Un metodo alternativo è il ricorso agli indici. Inoltre, se

- le tuple sono ordinate
- la selezione è fatta sull'attributo su cui la relazione è ordinata
- i blocchi sono memorizzati in modo contiguo

allora possiamo ottenere un numero medio di trasferimenti logaritmico $\log_2 n$.

2.3 Ordinamento

La procedura di ordinamento è necessaria sia per ottenere risultati ordinati che per una corretta realizzazione delle proiezioni, con eliminazione dei duplicati. Risulta utile anche per implementare in modo efficiente operazioni come il JOIN, o il raggruppamento. Possiamo ottenere l'ordinamento con:

- **quickSort**, se la relazione può essere posta tutta nel buffer;
- **mergeSort**, se la relazione non può essere posta tutta nel buffer e quindi dobbiamo lavorare su più frammenti.

2.4 JOIN

Il JOIN è l'operazione a più alto livello. Si hanno vari metodi:

- Nested-loop (*quello che si fa ad occhio* - cit)
- Merge scan
- Hash-based

2.4.1 Nested-loop

Algoritmo Abbiamo una tabella, una *interna* e una *esterna*. Per ogni tupla della tabella esterna scandisco la tabella interna individuando le tuple che fanno JOIN.

[2]Cognome = 'Rossi' AND Nome = 'Maria'
[3]Cognome = 'Rossi' OR Nome = 'Maria'

Costo

- Prendiamo le seguenti tabelle:
  - Tabella R con 10.000 record che occupano 400 blocchi
  - Tabella S con 5.000 record che occupano 100 blocchi.
- Consideriamo R esterna
- Prendiamo il caso peggiore: nel buffer si può inserire solo un record di S alla volta, quindi devo effettuare innumerevoli accessi. Ottengo il seguente numero di trasferimenti

  $$N = 10000 * 100 + 400$$

  Per ogni tupla della tabella R scandisco tutti i blocchi della tabella S, portandoli ripetutamente in memoria. La tabella R rimane nel buffer, quindi la porto in memoria principale soltanto una volta.
- Nel caso migliore, cioè quando tutta la tabella S entra nel buffer, ottengo

  $$N = 400 + 100$$

2.4.2 Merge scan

Questa tecnica si basa sull'ordinamento delle tabelle in base agli attributi di JOIN. L'algoritmo risulta molto efficiente quando le tabelle sono già ordinate o se sono presenti indici adeguati.

- Fondamentalmente abbiamo una scansione in parallelo delle due tabelle, come nella funzione *merge* dell'algoritmo *mergeSort*.
- Quando i valori dei due attributi coincidono ottengo una tupla appartenente al risultato. Il meccanismo fa sì che il risultato sia ordinato.

Il costo consiste nei trasferimenti in memoria ($N = br + bs$, ogni blocco è letto una volta soltanto assumendo che tutte le tuple per un dato valore di JOIN stiano insieme nel buffer) e nell'ordinamento (questo se le tabelle non sono già ordinate). L'algoritmo è utilizzato per il JOIN naturale o l'Equi-JOIN.

2.4.3 Hash-JOIN

- Utilizzo una funzione h di hash sugli stessi attributi per memorizzare una copia di ciascuna delle due tabelle (difetto del metodo è un utilizzo maggiore di memoria) in memoria centrale
- La funzione h comporta una partizione delle tabelle R ed S coinvolte: dati i valori del dominio di tali attributi ottengo come risultato lo stesso indice.
- L'indice identificherà una partizione di R ed una di S: a quel punto mi basta effettuare dei semplici JOIN tra queste partizioni.

Questa tecnica è utile solo per JOIN-naturale ed Equi-JOIN, quest ultimo svolto in modo più efficiente se confrontato con il nested-loop.

Costo del metodo

- Le relazioni R ed S, per essere partizionate, devono essere portate in memoria centrale. Intanto individuiamo i seguenti trasferimenti

  $$2(br + bs)$$

  Due volte poichè dobbiamo svolgere un'operazione di lettura e una di scrittura

- Per ogni tupla t_r, in ogni partizione di R, si considera la tupla t_s nella partizione corrispondente secondo la funzione hash, quindi si legge ciascuna partizione una volta. Seguono ulteriori trasferimenti
$$(br + bs)$$
- Il costo complessivo dell'hash-JOIN sarà
$$3(br + bs)$$

2.5 Misura del costo di una query

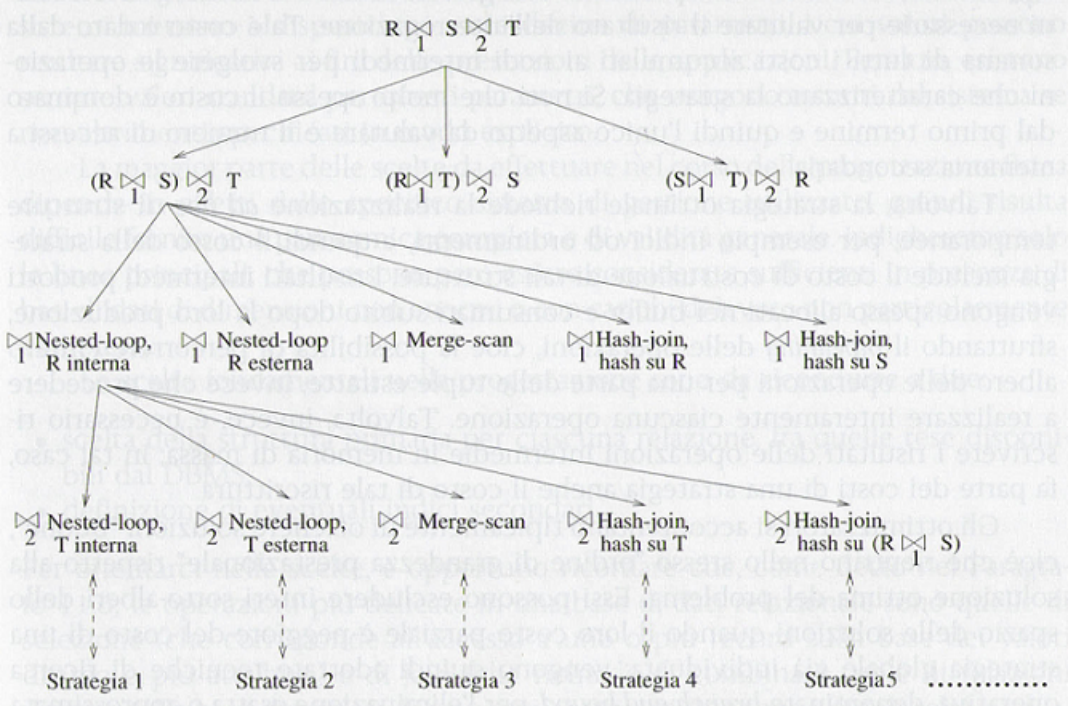
Molti fattori contribuiscono al costo di una query, cioè il tempo necessario per avere la risposta:

- L'accesso al disco, il tempo predominante calcolabile considerando:
  - il numero di scansioni
  - il numero di lettura
  - il numero di scritture
- il tempo di CPU
- il tempo di rete

Poniamo per semplicità il fattore t unico come tempo per accedere a un blocco.

- Dobbiamo considerare, oltre al numero di trasferimenti, la memoria utilizzata per memorizzare i risultati intermedi.
- Abbiamo ribadito come l'hash-JOIN necessiti di tabelle intermedie.
- Operazioni con più operandi devono essere eseguite uno step alla volta.
  - **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.
  - **Ordine dei JOIN**
    * L'ordine in cui si fanno i JOIN dipende dalla dimensione dei risultati intermedi
    * Date le relazioni $R1, R2, R3$, sappiamo che vale la proprietà associativa
    $$(R1 \bowtie R2) \bowtie R3 = R1 \bowtie (R2 \bowtie R3)$$
    * Se la dimensione di $R2 \bowtie R3$ è molto grande e quella di $R1 \bowtie R2$ è piccola, conviene eseguire i JOIN nel seguente ordine
    $$(R1 \bowtie R2) \bowtie R3$$
    in modo da dover memorizzare una relazione intermedia più piccola

Processo di ottimizzazione Si costruisce un albero di decisione con le varie alternative, si valuta il costo di ciascun piano e si sceglie il piano di costo minore. L'ottimizzatore sceglie di solito una soluzione *buona* non necessariamente una soluzione *ottima*.



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

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

Abbiamo le seguenti tabelle

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

E i seguenti dati:

- Numero di tuple in F: $N(F) = 30000$
- Numero di tuple in A: $N(A) = 2000$
- Numero di tuple in I: $N(I) = 600000$
- Numero di film distinti nelle interpretazioni: $N(Fid, I) = 30000$
- Numero di attori distinti nelle interpretazioni: $N(Aid, I) = 2000$
- Numero di nazionalità distinte tra gli attori: $N(Nazionalita, A) = 4$
- Numero di categorie distinte tra i film: $N(Categoria, F) = 5$
- Numero di anni distinti tra i film: $N(Anno, F) = 20$

Procediamo

1. Per prima cosa eseguiamo le ottimizzazioni sempre valide. Applicando *push selections down* e *push projections down* otteniamo

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

2. Prendiamo $\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X}(F))$
   - $\sigma_{(C=D \vee C=B)}(F)$
     - Vogliamo stabilire che gli oggetti appartengono alla categoria C o alla categoria D
     - Calcoliamo il numero di elementi appartenenti a una categoria

$$\frac{N(F)}{N(Categoria, F)} = \frac{30000}{5} = 6000$$

     - Poichè mi interessano due categorie moltiplico per 2: $6000 * 2 = 12000$
   - $\sigma_{A=X}(F)$
     - Vogliamo anche che siano selezionati solo i film realizzati nell'anno X
     - Per questo calcoliamo il numero di film per anno

$$\frac{N(F)}{N(Anno, F)} = \frac{30000}{20} = 1500$$

   - Attraverso i calcoli precedenti individuiamo che l'ordine più conveniente è il seguente

$$\sigma_{(C=D \vee C=B)}(\sigma_{A=X}(F))$$

   - Il numero finale di tuple si ottiene risvolgendo i calcoli: non pongo 30000 tuple ma 1500

$$\frac{1500}{5} * 2 = 300 * 2 = \boxed{600}$$

   - La proiezione $\pi_{T,FId}(\dots)$ non altera il numero di tuple visto che contiene la chiave. Il numero di tuple trovato prima (300) è confermato!

3. $\pi_{AId}(\sigma_{N=Y}(Ac))$
   - Consideriamo la selezione: vogliamo soltanto gli attori di nazionalità Y
   - Calcoliamo il numero di attori per nazionalità

$$\frac{N(Ac)}{N(Nazionalita, Ac)} = \frac{2000}{4} = \boxed{500}$$

   - La proiezione non altera il numero di tuple poichè si proietta la chiave di Ac

4. $\pi_{FId,AId}(I))$
   - Il numero di tuple relative all'interpretazione è $N(I) = \boxed{600000}$
   - La proiezione contiene la chiave, quindi il numero di tuple non viene alterato.

5. Abbiamo determinato l'ordine delle selezioni ed eventali riduzioni di tuple a causa delle proiezioni

6. Adesso dobbiamo determinare l'ordine dei JOIN.
   - $\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X}(F)) \bowtie \pi_{AId}(\sigma_{N=Y}(Ac))$
     - Non avendo attributi comuni il JOIN naturale risulta in un prodotto cartesiano. Segue il seguente numero di tuple

$$500 * 600 = 300000$$

- $\pi_{AId}(\sigma_{N=Y}(Ac)) \bowtie \pi_{FId,AId}(I)$
  - $\min(500 * 600000/2000, 600000 * 1) = 150000$
- $\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X}(F)) \bowtie \pi_{FId,AId}(I)$
  - $\min(600 * 600000/30000, 600000 * 1) = 6000$

7. Attraverso questi risultati finali sappiamo a quale JOIN dare priorità, cioè quello che mi restituisce il minor numero di tuple! Il JOIN in questione è

$$\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X}(F)) \bowtie \pi_{FId,AId}(I)$$

3 Progettazione fisica: fase finale

La progettazione fisica consiste nella fase finale del processo di progettazione di una base di dati.

- **Input**: Schema logico e informazioni sul carico applicativo
- **Output**: Schema fisico, costituito dalle definizioni delle relazioni con le relative strutture fisiche (e molti parametri, spesso legati allo specifico DBMS)

Dettagli

- La caratteristica comune dei DBMS relazionali è la disponibilità degli indici: la progettazione logica, spesso, coincide con la scelta degli indici.
- Le chiavi primarie delle relazioni sono di solito coinvolte in selezioni e JOIN: molti sistemi prevedono di definire indici sulle chiavi primarie.
- Altri indici vengono definiti con riferimento ad altre selezioni o JOIN importanti
- Se le prestazioni sono insoddisfacenti si pongono nuovi indici o si eliminano indici già esistenti. Attraverso il comando

  ```
  show plan
  ```

  possiamo caricare la lista degli indici creati.