ARCHITETTURA DI UN DBMS

- Finora abbiamo visto modelli di DBMS ad alto livello:
    livello *logico* :
    - e' il livello corretto per gli *utenti* del DB
    - l'utente vede il DB come una collezione di tabelle, o grafi (a seconda del modello dei dati)

- Tuttavia, un fattore importante nell'accettazione da parte dell'utente e' dato dalle prestazioni

- Le prestazioni del DBMS dipendono dall'efficienza delle strutture dati e dall'efficienza del sistema nell'operare su tali strutture dati

- Esistono varie strutture alternative per implementare un modello dei dati;
  la scelta dalla strutture piu' efficienti dipende dal tipo di accessi che si eseguono sui dati

- Normalmente un DBMS ha le proprie strategie di implementazione di un modello dei dati; tuttavia l'utente (esperto) puo' influenzare le scelte fatte dal sistema

ARCHITETTURA DI UN DBMS

Un DBMS consiste di un numero di **componenti funzionali** che includono:

- il **file system**: gestisce l'allocazione dello spazio su disco e le strutture dati usate per rappresentare le informazioni memorizzate su disco
- il **buffer manager** e' responsabile per il trasferimento delle informazioni tra disco e main memory
- il **query parser** traduce i comandi del DDL e del DML in un formato interno (parse tree)
- l'**optimizer** trasforma una richiesta utente in una equivalente ma piu' efficiente
- l'**authorization and integrity manager** controlla che i vincoli di integrita' (per esempio le chiavi) siano verificati e controlla che gli utenti abbiano i diritti di accesso ai dati
- il **recovery manager** assicura che il DB rimanga in uno stato consistente a fronte di cadute del sistema
- il **concurrency controller** assicura che interazioni concorrenti procedano senza conflitti

ARCHITETTURA DI UN DBMS

Un DBMS contiene inoltre alcune strutture dati che includono:

- i files con i dati (cioe' i files per memorizzare il DB stesso)

- i files dei dati di sistema;
  tali dati includono il dizionario dei dati, le autorizzazioni

- indici (esempio B-tree o tabelle hash)

- dati statistici: per esempio il numero di tuple in una relazione;
  tali dati sono usati dallo strategy selector per determinare la strategia ottima di esecuzione

ARCHITETTURA DI UN DBMS

SUPPORTI DI MEMORIZZAZIONE

- Cache

- Main memory (core memory): sebbene la main memory puo' contenere parecchi megabytes di dati, e' in generale non sufficiente per memorizzare un intero D B
  (tuttavia con l'avanzare della tecnologia si puo' prevedere lo sviluppo dei main-memory DB)
  inoltre il contenuto della MM e' perso se va via la corrente o si ha una caduta di sistema

- Direct-access storage (dischi): e' il mezzo principale per la memorizzazione dei dati in un DB

- Sequential-access storage (nastri): e' il mezzo usato principalmente per eseguire backup dei dati o per avere delle copie di archivio dei dati (cioe' copie ormai obsolete che comunque devono essere conservate)

SUPPORTI DI MEMORIZZAZIONE

Dischi

- Tempo di latenza del disco: il tempo necessario affinche' tutta una traccia passi sotto al braccio (arm) (e' un tempo molto breve)
- Tempo di seek: il tempo necessario per riposizionare il braccio sulle tracce (e' un tempo molto piu' alto rispetto al tempo di latenza)

SUPPORTI DI MEMORIZZAZIONE

Disk packs

- Sono dischi con piu' piatti

- Tutti i bracci del disco sono posizionati da una unita' chiamata *actuator*
- Ogni braccio a due testine: che permettono di leggere le due facce del piatto
- L'insieme di tracce su cui tutte le testine sono posizionate e' detto *cilindro* (un cilindro e' l'insieme delle tracce che sono accedibili in un tempo di latenza);

  quindi e' molto efficiente memorizzare dati sullo stesso cilindro

SUPPORTI DI MEMORIZZAZIONE

Disk packs

- I dati sono trasferiti tra il disco e la main memory in unita' chiamate blocchi;
  un blocco e' una sequenza di bytes contigui memorizzati in una stessa traccia di un singolo piatto

- La dimensione dei blocchi varia da 512bytes a parecchie migliaia (4K)

- Il tempo di trasferimento di un blocco e' il tempo impiegato dalla testina per trasferire un blocco nel buffer, una volta posizionata all'inizio del blocco

SUPPORTI DI MEMORIZZAZIONE

Dischi ottici

- Permettono (grazie alla tecnologia laser) di avere capacita' di memorizzazione molto alte

- Le cifre binarie sono memorizzate creando con un raggio laser dei fori (di diametro inferiore al micron) in uno strato metallico riflettente; i fori vengono poi "letti" da un raggio laser a potenza minore

- La capacita' e' molto maggiore rispetto ai dischi ottici (capacita' dell'ordine di Gbytes)

- Sono pero' molto lenti (100 a 500 ms) e non si possono modificare

- E' possibile ottenere la riscrittura con i dischi magneto-ottici. Un esempio e' l'unita' Canon da 5,25 pollici (usata nelle NeXT)

SUPPORTI DI MEMORIZZAZIONE

Dischi

Tabella di confronto:

	Dischi ottici	IBM 3380 (1981)
capacita'	1-4 Gbyte	2,5 Gbyte
numero piatti	1	15
distanza testina dal disco	1 mm	< 25 micron
tempo medio di accesso (ms)	100-500	16
velocita' di trasferimento (Kbyte/sec.)	500-2000	3000

ORGANIZZAZIONE DI FILES

- I files sono gestiti dal sistema operativo (SO)
  In tal senso il DBMS e' un *cliente* del SO

- Il DBMS deve determinare le strategie per mappare le strutture dati del modello in termini di records di files

- Normalmente la dimensione dei blocchi e' fissata dal SO (per esempio e' un parametro di generazione del SO); tuttavia la dimensione dei records po' variare

- Una soluzione e' di memorizzare il DB in vari files, in modo che ogni files abbia records di lunghezza uguale

- Questa soluzione puo' andare bene per il modello relazionale (in cui un file coincide con una relazione)

- Puo' non andare bene nel modello CODASYL in cui l'owner e il member di un set non e' detto che abbiano lunghezze uguali, ma devono essere memorizzate fisicamente vicini

ORGANIZZAZIONE DI FILES

Files con records a lunghezza fissa

Consideriamo la relazione
Deposit (branch-name: char(20);
account-number: integer;
customer-name: char(20);
balance: real;)

Per memorizzare questa relazione si puo' allocare un file, in cui ogni record abbia lunghezza 52

Questo tipo di organizzazione ha un problema in quanto e' difficile cancellare un record da questa struttura. Lo spazio occupato dal record deve essere occupato con un altro record, oppure e' necessario marcare i record cancellati

ORGANIZZAZIONE DI FILES

Files con records a lunghezza fissa

- Una soluzione e' di spostare il record successivo al record cancellato, poi spostare quello successivo e cosi' via

- Un'altra soluzione e' di spostare l'ultimo al posto di quello cancellato

- Poiche' le inserzioni tendono ad essere piu' numerose delle cancellazioni, e' preferibile lasciare libero lo spazio occupato dal record cancellato e aspettare la prossima inserzione

- Tuttavia non e' sufficiente marcare il record come cancellato, in quanto trovare spazio per una nuova inserzione e' piuttosto costoso

- Quindi e' necessario allocare delle strutture ausiliarie; normalmente l'insieme di tali strutture costituisce il *file header*

- In particolare il file header contiene il puntatore al primo record cancellato, questo record a sua volta punta al seguente record cancellato, e cosi' via

ORGANIZZAZIONE DI FILES

Files con records a lunghezza fissa

Esempio:

L'uso di puntatori richiede attenzione; se il record puntato viene spostato, il puntatore non e' piu' corretto (*dangling pointers*)

E' necessario evitare lo spostamento o la cancellazione di records puntati (tali records sono chiamati records *pinned*)

ORGANIZZAZIONE DI FILES

Files con records a lunghezza variabile

- La necessita' di record a lunghezza variabile deriva da:

  memorizzazione di tipi di record diversi nello stesso file (esempio: padre e figli di uno stesso set)

  memorizzazione di tipi di record con campi di lunghezza variabile

  memorizzazione di tipi di record con campi multivalore (per esempio, una tupla di O2 con un attributo di tipo set, o lista)

- Esempio:

```
add class deposit-list
   type tuple (branch-name: string,
         account-info: set(tuple
         (account-number: integer,
          customer-name: string,
         balance: real)));
```

ORGANIZZAZIONE DI FILES

Files con records a lunghezza variabile

Rappresentazione **byte-string**

- Questa rappresentazione consiste nell'aggiungere un simbolo speciale ($\perp$) *end-of-record* che indica la fine di ogni record

- Quindi ogni record consiste viene memorizzato come una sequenza di bytes consecutivi

ORGANIZZAZIONE DI FILES

Files con records a lunghezza variabile

Rappresentazione **byte-string**

Questo tipo di rappresentazione ha molti svantaggi tra cui i piu' gravi sono:

- Non e' facile riusare lo spazio occupato da un record cancellato. Esistono tecniche per gestire cancellazioni e inserzioni ma tendono a generare frammentazione

- Se un record aumenta di lunghezza deve essere spostato e questo e' costoso se il record e' pinned

ORGANIZZAZIONE DI FILES

Files con records a lunghezza variabile

Rappresentazione **fixed-length**

- **Reserved space**: se c'e' una lunghezza massima di record che non viene mai superata, si possono usare dei record di lunghezza fissa dimensionati al massimo

- Problema: spreco di spazio
  lentezza nell'accesso in quanto
  e' necessario accedere molti blocchi

ORGANIZZAZIONE DI FILES

Files con records a lunghezza variabile

Rappresentazione **fixed-length**

- **Pointers**: il record a lunghezza variabile e' rappresentato da una lista di record a lunghezza fissa, collegati tramite puntatori

- Problema: e' necessario includere *branch-name* in tutti i records

ORGANIZZAZIONE DI FILES

Files con records a lunghezza variabile

Rappresentazione **fixed-length**

Per risolvere il problema precedente si usano due tipi di blocchi nel file:

- **anchor block** contiene il primo record di una lista

- **overflow block** contiene i record che non sono i primi della lista

ORGANIZZAZIONE DI FILES

Organizzazione di records in blocchi

- Un file puo' essere visto come una collezione di records
- Tuttavia, poiche' i dati sono trasferiti in blocchi tra la MS e la MM, e' importante assegnare i records ai blocchi in modo tale che uno stesso blocco contenga records tra loro interrelati
- Se si riesce a memorizzare sullo stesso blocco records che sono spesso richiesti insieme (per esempio il figlio di un set dato l'owner) si risparmiano accessi a disco
- Se consideriamo l'esempio dell'organizzazione fixed-length per i records a lunghezza variabile, puo' essere utile memorizzare tutti i records, relativi allo stesso record a lunghezza variabile, nello stesso blocco
- Questo si puo' fare se il DB una volta caricato non cambia
- Se si hanno molte modifiche, si puo' determinare una situazione in cui un blocco finisce per avere records di liste diverse

ORGANIZZAZIONE DI FILES

Organizzazione di records in blocchi

Buckects

- Una soluzione e' usare un Bucket (un insieme di blocchi) per gruppi di records tra loro collegati
  (nel nostro esempio tutti i records relativi alla stessa filiale)

Esempio

Nell'esempio ogni bucket occupa un blocco

ORGANIZZAZIONE DI FILES

Organizzazione di records in blocchi

- E' possibile avere buckects che occupano piu' blocchi
- In tal caso, i blocchi di uno stesso buckect sono collegati tra loro (si alloca un certo spazio all'inizio di ogni blocco come *block header* e si usa per memorizzare il puntatore al primo blocco)

ORGANIZZAZIONE DI FILES

Organizzazione di records in blocchi

- Se un bucket cresce si allocano nuovi blocchi

- Se vengono cancellati molti records, un blocco del bucket puo' rendersi libero; i blocchi liberi sono collegati per poterli riusare in caso di nuove inserzioni nello stesso bucket

- E' preferibile non riusare i blocchi liberi di un bucket per memorizzare records di un altro bucket

- Il motivo e' che per maggiore efficienza i blocchi di uno stesso bucket sono memorizzati nello stesso cilindro

- Quindi se un blocco di un bucket si libera, e' preferibile mantenerlo disponibile per inserzioni nello stesso bucket

- Questa politica puo' causare un ampio numero di blocchi vuoti
  (notare che pero' nella maggioranza delle applicazioni di DB, la frequenza delle inserzioni e' maggiore della frequenza delle cancellazioni)

ORGANIZZAZIONE DI FILES

Organizzazione di records in blocchi

- Nel caso in cui un bucket causa overflow dal suo cilindro, e' necessario usare blocchi in altri cilindri

- Normalmente si cercano blocchi su cilindri vicini

- Se i bucket divengono troppo frammentati e' bene riorganizzare il DB

ORGANIZZAZIONE DI FILES

Gestione del buffer

- L'obiettivo principale delle strategie di memorizzazione e' di minimizzare gli accessi a disco
- Un altro modo e' di mantenere piu' blocchi possibile in MM
- Si usa un buffer che permette di tenere in MM copia di alcune pagine di disco
- Il buffer manager di un DBMS usa alcune politiche di gestione che sono piu' sofisticate delle politiche usate nei SO; in particolare

(i) le politiche di LRU devono essere estese per uso nei DBMS

(ii) in alcuni casi un blocco non puo' essere trasferito su disco (per motivi legati alla gestione del recovery); un blocco che non puo' essre trasferito e' detto *pinned*

(iii) in alcuni casi e' necessario forzare un blocco su disco (anche se il suo spazio non e' stato reclamato)

ORGANIZZAZIONE DI FILES

Gestione del buffer

- Un SO tipicamente usa le politiche LRU
- Questo tipo di politiche va bene quando non sia sa predire il pattern degli accessi; un DBMS e' tuttavia in grado di predire meglio il tipo dei futuri riferimenti
- Esempio:
  consideriamo una operazione di join
  borrow |x| customer (in cui le due relazioni siano in due files diversi)

ORGANIZZAZIONE DI FILES

Gestione del buffer

- Nell'esempio precedente una volta che una tupla della relazione borrow e' stata usata non e' piu' necessaria

- Quindi non appena tutte le tuple di un blocco sono state esaminate il blocco non serve piu'

- Il buffer manager deve pertanto liberare tale blocco (strategia *toss-immediate*)

- Consideriamo un blocco di customer:
  il blocco piu' recentemente acceduto sara' riferito di nuovo solo dopo che tutti gli altri blocchi saranno stati esaminati

- Pertanto la strategia migliore per il file customer e' di rimuovere l'ultimo blocco esaminato (strategia *most recently used* - MRU)

- Perche' la strategia MRU funzioni correttamente e' necessario eseguire il *pin* del blocco correntemente esaminato fino a che si siano esaminate tutte le tuple; quindi si puo' rendere il blocco unpinned

MAPPING DI RELAZIONI A FILES

- Una soluzione spesso adottata e' di memorizzare ogni relazione in un file separato

- Questa soluzione va bene per DBMS progettati per personal computers

- Nel caso di DBMS large-scale questa strategia di base deve essere estesa (molto spesso il DBMS deve poter allocare in modo opportuno i records ai blocchi per minimizzare le operazioni di I/O)

- Una strategia frequente e' di allocare per il DBMS un unico grosso file, in cui sono memorizzate tutte le relazioni;
  la gestione di questo file e' lasciata al DBMS

MAPPING DI RELAZIONI A FILES

Esempio - clustering

Consideriamo le relazioni

Consideriamo una query con un join

Select account-number, customer-name,
customer-city **from** deposit, customer
where deposit.customer-name =
customer.customer-name

Una strategia di memorizzazione efficiente per questo tipo di query e' basata sul clustering delle tuple che hanno lo stesso valore dell'attributo di join

MAPPING DI RELAZIONI A FILES

Esempio - clustering

- Il clustering puo' rendere l'esecuzione di altre query inefficiente

- Consideriamo la query:
  select * **from** customer

- Questa query richiede un accesso ad un numero maggiore di blocchi, rispetto ad una strategia in cui si usa un file separato per ogni relazione

- Per poter ritrovare le tuple della relazione customer puo' essere necessario collegarle con dei puntatori:

MAPPING DI RELAZIONI A FILES

Dizionario dei dati

Oltre a memorizzare i dati, un DBMS deve avere strategie di memorizzazione dei cataloghi

Tra le informazioni che occorre memorizzare ci sono:
- i nomi delle relazioni
- i nomi degli attributi di ogni relazione
- i domini degli attributi
- i nomi delle views e la loro definizione
- vincoli di integrita' (per esempio dipendenze)

Inoltre molti DBMS memorizzano informazioni sugli utenti:
- i nomi degli utenti autorizzati a connettersi
- le passwords (encrypted)
- i gruppi utente

Infine il DBMS deve memorizzare statistiche sulle relazioni e sulle strategie di memorizzazione usate:
- cardinalita' di ogni relazione
- strategia di memorizzazione (clustered o non clustered)
- informazioni sugli indici (nome dell'indice, relazione su cui l'indice e' allocato, tipo dell'indice, attributi su cui l'indice e' allocato)

MAPPING DI RELAZIONI A FILES

Dizionario dei dati

- Molto spesso il dizionario dei dati in un sistema relazionale e' memorizzato come un insieme di relazioni

- Questo rende possibile sfruttare le funzionalita' del DBMS per eseguire un accesso veloce al dizionario:

- Possibile organizzazione:

  System-catalog = (relation-name, number-of-attributes)

  Attribute-scheme = (attribute-name, relation-name, domain-type, position)

  User-scheme = (user-name, encrypted-password, group)

  Index-scheme = (index-name, relation-name, index-type, index-attributes)

  View-scheme = (view-name, definition)

MAPPING DEL MODELLO RETICOLARE

- Il modello relazionale non prevede esplicitamente la nozione di puntatore; i puntatori sono usati solo a livello di implementazione
  (nota: il modello relazionale realizza i collegamenti tra le tuple per mezzo di valori)

- Il modello reticolare prevede, invece, *links* tra i dati a livello di modello; questi links sono rappresentati per mezzo di puntatori

MAPPING DEL MODELLO RETICOLARE

Implementazione con *struttura a ring*

Esempio **DBTG set** custAcct
owner is customer
member is account;

Statement **find first** <record type> **within** <set>
find next <record type> **within** <set>

(il risultato dipende dall'ordine dei records nel ring)

MAPPING DEL MODELLO RETICOLARE

Struttura a ring estesa

Per poter supportare efficientemente l'operazione **find owner** e' necessario estendere la struttura a ring con un puntatore al padre; ogni record che rappresenta un figlio ha due puntatori:

- al fratello (*next-member pointer*)
- al padre (*owner pointer*)

MAPPING DEL MODELLO RETICOLARE

Clustering

- E' possibile richiedere che i record di un set-type siano clusterizzati
  Una prima strategia e' ottenuta con la clausola
  placement clustered via CustAcct
  nel comando di definizione del record type
  Account

MAPPING DEL MODELLO RETICOLARE

Clustering

- Con la clausola precedente i padri di un set non sono memorizzati necessariamente vicino ai figli
  E' possibile richiedere questo tipo di clustering con la clausola
  placement clustered via CustAcct **near owner**
  nel comando di definizione del record type Account

STRUTTURE AUSILIARIE DI ACCESSO

- Spesso le interrogazioni accedono solo un piccolo sottoinsieme dei dati

- Per risolvere efficientemente le interrogazioni associative puo' essere utile allocare delle strutture ausiliarie che permettano di determinare direttamente i records che verificano una data query (i.e. senza scandire tutti i dati)

- I meccanismi piu' comunemente usati sono dai DBMS sono: <u>indici</u>, <u>funzioni hash</u>

- Ogni tecnica deve essere valutata in base a:
  - tempo di accesso
  - tempo di inserzione
  - tempo di cancellazione
  - occupazione di spazio

- Molto spesso e' preferibile aumentare l'occupazione di spazio se questo contribuisce a migliorare le prestazioni

- Si usa il termine *chiave di ricerca* per indicare un attributo o insiemi di attributi usati per la ricerca

(questo concetto di chiave e' diverso dalla chiave primaria)

FILES SEQUENZIALI DA INDICE

- Sono usati per applicazioni che richiedono sia:
  accessi sequenziali
  accesso random a singoli records

- Un file sequenziale da indice consiste di:
  un file sequenziale
  un file indice

- Un file sequenziale e' organizzato in modo tale da permettere una elaborazione efficiente di records ordinati in base ad una chiave di ricerca
  (per esempio: ordinamento alfabetico in base al valore dell'attributo Cognome della relazione Persone)

- Per rendere l'accesso veloce i records sono collegati per mezzo di puntatori in base all'ordinamento dettato dalla chiave di ricerca

- Inoltre per minimizzare il numero di blocchi acceduti, i records sono memorizzati (possibilmente) in base all'ordine della chiave di ricerca

FILE SEQUENZIALE

Esempio:

Operazioni:

- la cancellazione viene gestita usando le catene di puntatori
- l'inserzione viene eseguita in base alle seguenti regole:

  (1) si determina il record nel file che precede immediatamente il record da inserire

  (2) se c'e' un record libero nello stesso blocco del record determinato al passo (1), vi si inserisce il

nuovo record. Altrimenti is inserisce il nuovo record in un blocco di *overflow* .

In entrambi i casi e' necessario riaggiornare i puntatori

FILE SEQUENZIALE

Esempio:

File relativo alla relazione deposit dopo l'inserzione del record (North Town, 888, Adams, 800)

Dopo numerose inserzione, l'ordinamento dato dalla chiave di ricerca non coincide con l'ordinamento fisico, quindi puo' essere necessario riorganizzare il file

FILE INDICE

- Un file sequenziale e' utile quando si deve eseguire un accesso veloce a tutti i records di un file in base ad una data chiave di ricerca

- Un file sequenziale non consente un accesso random efficiente

- Si usano strutture ad indice per rendere piu' efficiente l'accesso random

- **Indice denso**:
  l'indice contiene un record per ogni valore della chiave di ricerca nel file. Il record contiene il valore della chiave di ricerca e un puntatore al record del file dei dati (chiamato *record primario*)

- **Indice sparso**:
  i records dell'indice sono creati solo per alcuni records primari. Per localizzare un record, si esegue una scansione fino a trovare il record con il piu' alto valore della chiave che sia minore o uguale al valore

cercato. Si accede il file dei dati e si comincia una ricerca sequenziale seguendo i puntatori fino a trovare il record cercato

FILE INDICE

Esempio:

indice denso

indice sparso

FILE INDICE

- Un indice denso consente una ricerca piu' veloce, ma impone maggiori costi di aggiornamento

- Un indice sparso e' meno efficiente ma impone minori costi di aggiornamento

- Poiche' molto spesso la strategia e' di minimizzare il numero di blocchi trasferiti, un compromesso spesso adottato consiste nell'avere una entrata nell'indice per ogni blocco

FILE INDICE

- Molto spesso un indice anche se sparso puo' essere di dimensioni notevoli
- Esempio:

  un file di 100,000 con 10 records per blocco richiede un indice con 10,000 entrate

  assumendo che un blocco contenga 100 entrate dell'indice, sono necessari 100 blocchi

- Se l'indice e' piccolo puo' essere tenuto in MM

- Molto spesso pero' e' necessario tenerlo su disco e quindi la scansione dell'indice puo' richiedere parecchi trasferimenti di blocchi

- Se l'indice occupa un numero *b* di blocchi e si usa una ricerca binaria e' necessario accedere un numero di blocchi pari a $1+\log_2(b)$
  (nel nostro esempio circa 7)

- Quindi e' necessario trattare l'indice come un file ed allocare un indice sparso sull'indice stesso

FILE INDICE

Indice sparso a due livelli:

Se l'indice outer e' mantenuto in MM, per eseguire una ricerca nell'esempio precedente e' necessario accedere un solo blocco di indice

FILE INDICE

Indice sparso a due livelli - operazioni

- Cancellazione:
  per cancellare un record occorre prima eseguirne la ricerca

  se il record cancellato e' l'ultimo con il proprio valore della chiave di ricerca, e' necessario cancellare l'entrata dall'indice nel caso di indici densi

  nel caso di indici sparsi, se nell'indice c'e' una entrata con la chiave di ricerca del record cancellato, tale entrata va sostituita con una entrata che ha la chiave immediatamente successiva in base all'ordinamento di ricerca

  se esiste gia' una tale entrata, allora l'entrata relativa al record cancellato va cancellata
  (significa che il record cancellato era l'ultimo del suo blocco)

FILE INDICE

Indice sparso a due livelli - operazioni

- Inserzione:

  si esegue una ricerca con valore per la chiave di ricerca uguale al valore v nel record da inserire;

  se l'indice e' denso e non esiste una entrata con valore v, allora si crea una tale entrata

  se l'indice e' sparso, non e' necessario creare una entrata a meno che si debba creare un nuovo blocco

INDICI SECONDARI

- In un file index-sequential normalmente si mantiene un solo indice

- Se si vogliono mantenere piu' indici, l'indice la cui chiave e' usata per mantenere l'ordinamento dei records nel file e' detto *indice primario*
- Gli altri indici sono detti *indici secondari*

- Gli indici secondari sono spesso organizzati in modo differente rispetto all'indice primario - esempio

- Nell'esempio tutti i puntatori ai records nel file dati per uno stesso valore della chiave secondaria sono memorizzati insieme (questo permette di eseguire parte di alcune queries usando i puntatori)

INDICI SECONDARI

- L'uso dell'indirezione permette inoltre di evitare di aggiungere puntatori ai records nel file dati

- Esempio:

- E' preferibile usare indici densi per gli indici secondari anziche' usare indici sparsi
  (nell'esempio e' necessario accedere tre records per trovare il record che ha nome Peterson:
  notare che il file dei dati non e' ordinato in base alle chiavi di ricerca secondarie, ma solo in base alla chiave di ricerca primaria)

INDICI SECONDARI

- L'uso di piu' indici secondari rende l'esecuzione delle queries piu' efficiente, ma rende piu' costosi gli aggiornamenti

- Quando si esegue l'inserzione o la cancellazione di un record e' necessario modificare tutti gli indici allocati sul file

- Gli indici secondari sono in genere di solito indici densi, mentre gli indici primari sono indici sparsi (notare che i records nel file dei dati sono ordinati in base al valore delle chiavi di ricerca dell'indice primario)

B-TREE

- Il problema principale con l'organizzazione sequenziale ad indice e' che la performance decresce con la crescita del file. Quindi e' necessario eseguire spesso delle riorganizzazioni

- Tali problemi si risolvono usando organizzazioni ad albero bilanciato (B-tree, B^*-tree, B^+-tree)

- Un B-tree ha le seguenti caratteristiche:
  a) ogni nodo ha al piu' m figli, dove m e' l'unico parametro dipendente dalla MS (piu' specificamente dalla dimensione del blocco)
  b) il costo dell'operazione di ricerca nel caso peggiore e' $(1 + \log_{\lceil m/2 \rceil} [(N + 1)/2])$, dove N e' la cardinalita' dell'indice (cioe' il numero di valori distinti delle chiavi di ricerca)
  c) l'utilizzo della MS e' almeno del 50%

B-TREE

Definizione

Un B-albero di ordine m ($m \geq 3$) e' un albero che soddisfa le seguenti proprieta':

- ogni nodo contiene al piu m-1 elementi;
- ogni nodo, tranne la radice, contiene almeno $\lceil m/2 \rceil - 1$ elementi;
- un nodo e' terminale, cioe' una foglia, oppure ha j +1 figli dove j e' il numero degli elementi in esso contenuto;
- tutti i cammini dalla radice ai nodi foglia hanno la stessa lunghezza (il B-tree e' un albero bilanciato)
- ogni nodo ha una struttura del tipo
  $[p_0 (k_1, r_1) p_1 (k_2, r_2) p_2 \dots\dots\dots (k_j, r_j) p_j]$
  dove:
    - j e' il numero degli elementi del nodo
    - le chiavi sono ordinate: $k_1 < \dots\dots < k_j$
    - nel nodo sono presenti j+1 riferimenti ai nodi figli (nei nodi terminali i riferimenti sono indefiniti)
    - per ogni nodo non terminale valgono le seguenti proprieta':

      $\forall y \in K(p_0)$, $y < k_1$

      $\forall y \in K(p_i)$, $k_i < y < k_{i+1}$, i=1,....., j-1

      $\forall y \in K(p_j)$, $y > k_j$

      dove $K(p_i)$ (i=0,.....,j) e' l'insieme delle chiavi memorizzate nel sottoalbero di radice p_i

B-TREE

Esempio:

Albero di altezza 3

L'altezza h di un albero e' definita come il numero di nodi che compaiono in un cammino dalla radice ad un nodo terminale

Una delle caratteristiche importanti dei B-tree e' la possibilita' di prevedere con sufficiente approssimazione l'altezza media dell'albero in funzione delle chiavi presenti: questo permette di stimare i costi di ricerca

B-TREE

Altezza media di un B-tree

b_{min}= numero minimo di nodi dell'albero
b_{max}= numero massimo di nodi dell'albero
N_{min}= numero minimo di chiavi dell'albero
N_{max}= numero massimo di chiavi dell'albero

si ottiene:

$b_{min} = 1 + 2 + 2\lceil m/2 \rceil + 2\lceil m/2 \rceil^2 + ... + 2\lceil m/2 \rceil^{h-2}$
$= 1 + [2(\lceil m/2 \rceil^{h-1} - 1)] / (\lceil m/2 \rceil - 1)$

$b_{max} = 1 + m + m^2 + ...+ m^{h-1} = (m^h - 1)/(m - 1)$

N_{min}= radice + (numero min. di chiavi per nodo)(b_{min} - 1) =
$2\lceil m/2 \rceil^{h-1} - 1$

N_{max}= (numero max. di chiavi per nodo)b_{max} =
m^h 1

Vale la relazione $2\lceil m/2 \rceil^{h-1} - 1 \leq N \leq m^h$ 1
e quindi
$\log_m (N+1) \leq h \leq 1 + \log_{\lceil m/2 \rceil} [(N + 1)/2]$
con $N \geq 1$ e h=0 per N=0

B-TREE

Esempi di valori per *h* in funzione di N e *m*
si assume che le chiavi siano di 10 bytes e i puntatori di 4 bytes

B-TREE

Operazioni - Ricerca

- Una volta portata la radice in memoria, si esegue la ricerca tra le chiavi contenute fino a determinare la presenza o l'assenza nel nodo della chiave cercata.

- Se la chiave k non viene trovata, puo' essersi verificato uno dei seguenti casi:

1) $k_i < k < k_{i+1}$ con $1 \leq i \leq m$: si continua la ricerca nel nodo p_i
2) $k_m < k$: si continua la ricerca nel nodo p_m
3) $k < k_1$: si continua la ricerca nel nodo p_0

- Se il nodo e' foglia, e nessuno dei tre casi si e' verificato, cio' significa che la chiave non e' presente nell'albero
- I costi di ricerca sono $C_{rmin}=1$, $C_{rmax}=h$

- I B-tree permettono di reperire con facilita' il successore di un elemento. Nei casi piu' fortunati la pagina contenente il dato e' gia' in memoria

- Il caso peggiore e' quando si cerca il successore di un elemento che appare nella radice: in tal caso il successore appare in una foglia e il costo e' $C_{rmax}=h-1$

B-TREE

Operazioni - Inserimento

- L'inserimento di un nuovo elemento in un B-tree necessita prima di tutto di una ricerca per verificare se esso sia gia' presente

- In caso di insuccesso, la nuova chiave verra' inserita nell'ultimo nodo visitato, che sara' un nodo foglia: sia p l'indirizzo del nodo e k il valore della chiave da inserire

- Se la chiave deve essere inserita in un nodo con meno di m-1 elementi, essa viene inserita in posizione opportuna, e cosi' il processo di inserimento rimane all'interno del nodo

- Se in nodo e' gia' pieno, cioe' contiene gia' m-1 elementi, allora esso viene suddiviso in due nodi, ciascuno dei quali contiene m/2 chiavi: questa operazione e' denominata *suddivisione* (*split*)

- L'elemento centrale del nodo viene passato al livello superiore per essere inserito nel nodo che costituisce il padre dei due nuovi nodi

- In questo modo e' possibile innescare un procedimento ricorsivo

B-TREE

Operazioni - Inserimento

Esempio: si vuole inserire la chiave 70

1) si applica l'algoritmo di ricerca per controllare che la chiave 70 non sia gia' presente; si osserva che la pagina D in cui la chiave dovrebbe risiedere e' gia' piena (costo: tre letture)

2) si divide in due la pagina D, creando una nuova pagina E: entrambe sono mantenute per il momento in MM

3) si distribuiscono equamente le m+1 chiavi risultanti dall'aggiunta della chiave 70 alla pagina D tra le pagine D ed E, mentre la chiave di valore mediano viene innalzata di un livello per essere inserita nella pagina padre A (costo: due scritture per D ed E, una scrittura per A)

B-TREE

Operazioni - Inserimento

Costo:

- Se non si rende necessaria la suddivisione, l'algoritmo di inserimento richiede h letture (C_{rmin}=h) ed una riscrittura (C_{wmin}=1)

- Il caso peggiore e' quello in cui la uddivisione si ripercuote fino alla radice inclusa, aumentando di uno l'altezza dell'albero. I nodi letti sono C_{rmax}=h e quelli riscritti C_{wmax}=2h+1

- Il costo medio dell'operazione e' pero' inferiore a quello massimo, in quanto difficilmente il numero delle suddivisioni nsd raggiunge il valore massimo h; il numero medio delle suddivisioni verifica la seguente proprieta':

  $nsd/N < 1/(\lceil m/2 \rceil -1)$

  Per m=200, il numero medio di suddivisioni per inserimento e' 1/99

B-TREE

Operazioni - Cancellazione

- Se la chiave e' da cancellare e' contenuta in una foglia, la si elimina

- Altrimenti la si sostituisce con la chiave successiva (o precedente) nell'ordinamento, che si trova in un nodo foglia, e quest'ultima viene cancellata dal nodo di appartenenza

- L'operazione di cancellazione puo' abbassare il numero dei figli al di sotto del limite $\lceil m/2 \rceil$. In tal caso e' necessario completare l'operazione con una delle due seguenti:

Concatenazione. Il nodo si combina con un nodo fratello adiacente con $\lceil m/2 \rceil$ figli, operando in modo inverso al processo di divisione. Si supponga di partire da un nodo padre:

Q: [....., (k_{j-1}, r_{j-1}) p_{j-1}, (k_j, r_j) p_j,(k_{j+1}, r_{j+1}) p_{j+1},.....]

dove p_{j-1} e p_j puntano ai seguenti nodi Q1 e Q2

Q1: [p_0 , (k_1, r_1) p_1, ...,(k_e, r_e) p_e]

Q2: [p'_0 , (k_{e+1}, r_{e+1}) p_{e+1}, ...]

Il concatenamento dei due fratelli Q1 e Q2 modifica il nodo Q nel modo seguente:

Q: [....., (k_{j-1}, r_{j-1}) p_{j-1},(k_{j+1}, r_{j+1}) p_{j+1},.....]

B-TREE

Operazioni - Cancellazione

Concatenazione (continua):

il puntatore p_{j-1} punta ad un nodo Q' ottenuto concantenando Q1 e Q2

$[p_0\,,(k_1, r_1)\,p_1, \ldots\ldots\ldots\ldots, (k_e, r_e)\,p_e, (k_j, r_j)p'_0\,, (k_{e+1}, r_{e+1})\,p_{e+1}, \ldots\ldots\ldots\ldots]$

L'eliminazione della chiave k_j dal nodo Q (viene inserita nel nodo Q') puo' innescare a sua volta la concatenazione di Q con un nodo adiacente. Questo processo si puo' propagare fino alla radice

Bilanciamento. Se fra due nodi fratelli adiacenti non si puo' applicare la concatenazione, allora si distribuiscono fra di essi gli elementi in modo bilanciato. Il bilanciamento interessa anche il nodo padre, poiche' uno dei suoi elementi viene modificato, ma il numero di essi non cambia e pertanto il fenomeno non si propaga

B-TREE

Operazioni - Cancellazione

Bilanciamento (continua)

Supponendo di partire da un nodo padre Q

Q: $[........(k_f, r_f)\ p_f......]$ che punta a due nodi fratelli

Q1: $[p_0, (k_1, r_1)\ p_1, ...,(k_e, r_e)\ p_e]$

Q2: $[p'_0, (k'_1, r'_1)\ p'_1, ...,(k'_j, r'_j)\ p'_j]$

per bilanciare i due nodi si ridistribuiscono le chiavi in modo da averne $\lfloor (e + j)/2 \rfloor$ nel nodo sinistro, e $\lceil (e + j)/2 \rceil$ nel nodo destro;

nell'eseguire il bilanciamento e' necessario risistemare la chiave nel nodo padre

Costi:

1) la chiave appartiene ad un nodo terminale e non e' necessario ne' la concatenazione ne' il bilanciamento: C_{rmin}=h, C_{wmin}=1

2) la chiave appartiene ad un nodo non terminale e non e' necessario ne' la concatenazione ne' il bilanciamento: C_{rmin}=h, C_{wmin}=2

3) il caso peggiore e' quello in cui per tutti i nodi del cammino percorso, esclusi i primi due, e' richiesta una concatenazione e per il figlio della radice anche il bilaciamento: C_{rmax}=2h-1, C_{wmax}=h+1

B-TREE

Operazioni - Cancellazione

Esempio

B*-TREE
Definizione

- I B*-tree sono una variante dei B-tree ottenuta memorizzando tutte le coppie (k_i, r_i) nelle foglie e successivamente prendendo le chiavi piu' alte di ogni foglia ed organizzando con esse una struttura B-tree.

- Il B*-tree richiede una parziale duplicazione di chiavi perche' la ricerca di una chiave avviene nel B-tree fino ad individuare un foglia, nella quale si trovano le chiavi e i riferimenti ai dati, insieme ai puntatori che costituiscono un collegamento a lista bidirezionale fra i nodi-foglia, per agevolare ricerche per intervalli di chiavi o sequenziali

B*-TREE

Definizione

Un B*-tree di ordine *m* ($m \geq 3$) e' un albero che soddisfa le seguenti proprieta':

- le coppie (k_i, r_i) sono tutte e sole nei nodi terminali La struttura dei nodi terminali e' la stessa dei B-tree, ma sono eliminati tutti i puntatori inutili:
  $[(k_1, r_1)\ (k_2, r_2) \ldots\ldots\ldots (k_m, r_m)\ ]$;

- le chiavi piu' alte di ogni nodo terminale sono riportate nei nodi non terminali, che hanno la struttura:
  $[p_0 , k_1, p_1 , k_2, p_2 , k_j, p_j]$, dove $\lceil m/2 \rceil - 1 \leq j \leq m-1$;

- per ogni nodo non terminale valgono le seguenti proprieta':

  $\forall\ y \in K(p_0)\ ,\ y \leq k_1$
  $\forall\ y \in K(p_i)\ ,\ k_i < y \leq k_{i+1}\ \ ,\ i=1,\ldots..,\ j-1$
  $\forall\ y \in K(p_j)\ ,\ y > k_j$
  dove $K(p_i)$ (i=0,.....,j) e' l'insieme delle chiavi memorizzate nel sottoalbero di radice p_i

- Ogni nodo terminale ha un puntatore al nodo terminale precedente e successivo (cioe' tutte i nodi terminali sono collegati in una lista bidirezionale)

B*-TREE

Esempio

B+-TREE

- Un'altra variazione dei B-tree e' rappresentata dai B+-tree, che sono la struttura generalmente usata per organizzare indici

- Un B+-tree si ottiene dal B*-tree modificando la struttura dei nodi non terminali che assumono la forma:
  $[k_1, p_1, k_2, p_2, k_j, p_j]$

- Ogni k_i e' la chiave piu' alta presente nel sottoalbero puntato da p_i

- I nodi terminali sono collegati a lista come in un B*-tree

- Esempio:

B+-TREE

Confronto con B-tree

Confrontando un B+-tree con un B-tree, a parita' di dimensione dei nodi, si possono fare le seguenti osservazioni:

- la ricerca di una singola chiave e' piu' costosa in media in un B+-tree, in quanto si deve per forza raggiungere sempre la foglia per ottenere il puntatore ai dati, e dunque il costo medio *h* della ricerca coincide con il costo massimo del caso di un B-tree

- per operazioni che richiedono il reperimento dei records ordinate in base al valore della chiave o per intervalli di chiave i B+-tree sono da preferirsi, perche' il collegamento a lista delle foglie elimina la necessita' di accedere ai nodi ad altri livelli;

- il B-tree e' piu' conveniente in quanto ad occupazione di memoria, perche' le chiavi sono memorizzate una volta sola

B+-TREE

Ricerca per intervallo di chiavi

- Una delle ragioni che porta a preferire un'organizzazione basata su B+-tree ad un'organizzazione basata sul metodo hash e' la possibilita' di effettuare ricerche per valori della chiave in un intervallo (*range query*)

- Si vuole determinare tutte i records con valore della chiave k tale che $V1 < k < V2$

- L'intervallo non deve essere troppo ampio, altrimenti e' piu' conveniente eseguire una scansione sequenziale di tutto il file

- Sia N il numero dei records, memorizzati in b pagine, e Nf il numero delle foglie del B+-tree,
  $Nf=[(Lk + Lr)N/[dp \quad x\ (\ln 2)]$
  dove Lk e Lr sono la dimensione in bytes di un valore della chiave e di un riferimento (di solito Lr=4); dp e' la dimensione di un nodo; ln 2 = 0.69 e' il valore medio del coefficiente di riempimento di un nodo

- Il fattore di selettivita' FC stima la frazione dei dati che soddisfano la condizione
  $FC = (V2 - V1)/(kmax - kmin)$

B+-TREE

Ricerca per intervallo di chiavi

- Per recuperare una registrazione usando l'indice, si trova prima il riferimento ad essa in una foglia e poi si accede alla registrazione
- Supponendo che basti un accesso per recuperare una foglia (ipotesi molto spesso verificata), la ricerca per tutti i riferimenti comporta l'accesso ad un numero di foglie pari a FC x Nf
- Il numero di riferimenti trovati e':
  Nr = FC x N
- Essendo i records non ordinati sui valori della chiave dell'indice, il numero delle pagine dati da visitare per trovare gli Nr records e' pari a Nr
- Pertanto il costo complessivo della ricerca con l'uso di un indice denso e' quindi:
  FC x Nf + FC x N
- La ricerca dei records con una scansione del file dei dati ha invece un costo pari al numero delle pagine b, essendo i dati non ordinati rispetto ai valori della chiave
- Quindi l'indice e' vantaggioso se
  FC x (Nf + N) < b

Esempio:

N=500.000   dim_record = 200

lunghezza_chiave = 16, dim_pagina = 2400 bytes

conviene l'indice per FC < 0.12

FUNZIONI HASH

- Il metodo basato sugli indici ha lo svantaggio che e' necessario eseguire una scansione din una struttura dati per localizzare i dati
- Il meccanismo basato sulle funzioni hash evita questo determinando l'indirizzo di un dato tramite una funzione calcolata sul valore della chiave cercata
- Le funzioni hash sono usate principalmente per l'organizzazione primaria dei dati (cioe' nei casi in cui l'ordinamento dato dalla chiave coincide con l'ordinamento fisico dei dati)
- Sia V un alfabeto finito e V^L la cardinalita' dell'insieme delle chiavi di lunghezza L definibili su V. Se N sono i records da memorizzare, il rapporto (N/V^L) viene chiamato *densita' delle chiavi attive*.
- Il file dei dati e' suddiviso in un'area primaria e un'area per i trabocchi (overflow)
- Le M pagine della'are primaria, di capacita' c, sono numerate da zero ad M-1, e sono accedute in modo diretto; il rapporto d=N/(M x c), fra il numero delle chiavi attive e il massimo numero di chiavi memorizzabili e' detto *fattore di caricamento*

FUNZIONI HASH

- Le pagine dell'area primaria sono accedute direttamente

- Le pagine dei trabocchi sono accedute mediante riferimenti memorizzati nelle pagine dell'area primaria

- L'area dei trabocchi manca in alcune soluzioni, ma quando e' prevista e' gestita con due funzioni: la prima restituisce una pagina libera quando e' necessario memorizzare un nuovo trabocco; la seconda riporta nell'insieme delle pagine libere una pagina non piu' utlizzata

- Sull'insieme della chiavi K e' definita una funzione di trasformazione H che fa corrispondere ad ogni chiave l'indirizzo di una pagina primaria

- La funzione e' detta *uniforme* se gli indirizzi che produce come risultato sono uniformemente distribuiti nell'intervallo (0, M-1)
- Due chavi k_1 e k_2 sono dette *sinonimi* se $H(k_1)=H(k_1)$, cioe' se generano una *collisione*

FUNZIONI HASH

- Quando il numero delle chiavi assegnate dalla trasformazione ad una stessa pagina supera la sua capacita', si ha un trabocco ed occorre prevedere una strategia per memorizzare le chiavi in eccesso in pagine alternative

- Una trasformazione e' detta *perfetta* se per un certo insieme di chiavi attive, non produce trabocchi
  Per grandi insiemi di records dinamici non e' possibile trovare funzioni di trasformazioni perfette

- Una trasformazione perfetta puo' essere sempre definita disponendo di un'area primaria di capacita' complessiva pari alla cardinalita' dell'insieme delle possibili chiavi

- Ad esempio, se le chiavi sono N numeri naturali consecutivi $\{k_1, k_2, ..., k_N\}$, e i records hanno lunghezza uguale e costante, dividendo $(k-k_1)$ (dove k e' una chiave ricercata) per la capacita' delle pagine, il quoziente e' l'indirizzo associato alla chiave, mentre il resto e' la posizione all'interno della pagina

- In generale densita' delle chiavi attive e' bassa e pertanto questo metodo non e' applicabile

FUNZIONI HASH

Il progetto di un'organizzazione basata su funzioni hash richiede specificare i seguenti parametri:

- la funzione per la trasformazione della chiave

- il metodo per la gestione dei trabocchi

- il fattore di caricamento

- la capacita' delle pagine

TRASFORMAZIONE DELLA CHIAVE

- Una funzione di trasformazione e' un'applicazione suriettiva H dall'insieme delle chiavi all'insieme degli indirizzi possibili che verifichi le seguenti proprieta':
  - (1) distribuzione *uniforme* delle chiavi nello spazio degli indirizzi, cioe' ogni indirizzo deve essere generato con la stessa probabilita';
  - (2) distribuzione *casuale* della chiavi, cioe' eventuali correlazioni tra i valori delle chiavi (ad esempio A1, A2) non devono tradursi in correlazioni tra gli indirizzi generati

- Tali proprieta' dipendono dall'insieme delle chiavi su cui si opera e quindi non esiste una funzione universale ottima

- Se i valori delle chiavi sono stringhe alfanumeriche, prima di applicare la trasformazione, ad essi va associato un numero con una funzione Num(k), tale che $Num(k_1)$ ~=$Num(k_2)$

TRASFORMAZIONE DELLA CHIAVE
Esempi

- *Metodo della divisione*: E' la funzione piu' usata, per la sua semplicita' e per il buon comportamento nella maggioranza dei casi. E' definita come:

  $H(k) = k \bmod M_p$ con M_p numero primo piu' grande minore o uguale ad M, oppure numero non primo minore o uguale ad M con nessun fattore primo minore di 20

  Ad esempio: M=100, H(k)=k mod 97, e per rendere la funzione surgettiva si pone $M=M_p$

- *Metodo della somma.* La chiave e' divisa in un certo numero di parti, ognuna delle quali ha lo stesso numero di caratteri dell'indirizzo da generare, con la possibile eccezione dell'ultima parte; l'indirizzo e' ottenuto sommando le parti e ignorando il riporto

  Ad esempio: l'indirizzo e' di tre cifre, la chiave 356942781 deve essere partizionata in parti ognuna avente tre cifre: 356 942 781, l'indirizzo ottenuto e' 079

GESTIONE DEI TRABOCCHI

- Esistono due strategie per la gestione dei trabocchi:
  (1) metodi di indirizzamento aperto
  (*open addressing*)
  (2) metodi di concatenamento
  (*closed addressing, chained techniques*)

- Nei metodi del primo tipo, un trabocco e' memorizzato in un'altra pagina dell'area primaria, il cui indirizzo e' determinato con una opportuna procedura, detta *legge di scansione*

- Nei metodi di concatenamento, invece, oltre all'utilizzo di una legge di scansione, si impiega una struttura a lista che collega i trabocchi di una pagina. Inoltre i trabocchi possono essere memorizzati in altre pagine dell'area primaria, oppure possono avvalersi di un'area distinta, detta area *secondaria* o *separata*

GESTIONE DEI TRABOCCHI

Metodi di indirizzamento aperto

- Le tecniche di indirizzamento aperto (studiate inizialmente per applicazioni in MM) si distinguono per la legge di scansione adottata per visitare l'area primaria, a partire dalla pagina di indirizzo H(k), alla ricerca di una pagina non satura in fase di inserimento di un record, oppure alla ricerca della pagina contenente la chiave cercata, in fase di ricerca

- Il metodo piu' semplice di procedere prende il nome di *scansione lineare* (*linear probing, open overflow, consecutive spill method*) e consiste nell'incrementare l'indirizzo iniziale H(k) di una quantita' costante s, detta *passo*, secondo la legge:

  $H_i(k) = (H_0(k) + s \times i) \bmod M, i=0,1,2...$

  Se il valore di s non ha divisori in comune con M, i primi M valori $H_i(k)$ sono tutti i possibili indirizzi delle pagine dell'area primaria (immaginando che l'area sia circolare)

- Questo modo di procedere comporta un fenomeno indesiderato, detto di *agglomerazione primaria*: invece di distribuirsi uniformemente su tutta l'area primaria, i records tendono ad addensarsi in certe pagine

GESTIONE DEI TRABOCCHI

Metodi di indirizzamento aperto

- L'agglomerazione primaria e' dovuta al fatto che la sequenza di indirizzi $H_i(k)$ e' generata con una legge di scansione lineare, pertanto ad una chiave $k_1$puo' essere associato come indirizzo iniziale $H_0(k_1)$ quello generato al passo i con la legge di scansione di un'altra chiave $H_i(k_2)$, aumentando la probabilita' di trabocco della pagina

  Ad esempio le chiavi 1234 e 245 con la funzione
  $H_i(k) = (H_0(k) + 3 \times i) \bmod 31$
  generano le sequenze
  (25, 28, 0, 3, 6, 9, 12,......) e
  (28, 0, 3, 6, 9, 12, 15,....)
  pertanto un trabocco dalla pagina 25 aumenta la probabilita' che trabocchi la pagina 28, poi la pagina 0, e cosi' via

  Supponiamo che le pagine 25, 28, e 0 siano sature; mentre gli indirizzi $H_0(k)$ sono ugualmente probabili, gli indirizzi delle pagine non sature non hanno la stessa probabilita' di essere generati.
  Infatti una chiave $\underline{k}$ viene memorizzata nella pagina 3, se $H_0(\underline{k})$ appartiene a {25, 28, 0, 3}, ma viene memorizzata nella pagina 6 solo se $H_0(\underline{k})=6$

GESTIONE DEI TRABOCCHI

Metodi di indirizzamento aperto

- Per evitare l'agglomerazione primaria, e' necessario rendere variabile il passo di scansione ricorrendo ad una funzione non lineare come legge di scansione

- Ad esempio usando una scansione quadratica
  $H_i(k) = (H_0(k) + a * i + b * i^2) \text{ Mod } M$
  con a=3 e b=5, le sequenze di indirizzi generati per le chiavi 1234 e 245 diventano:
  (25, 2, 20, 17, 24, 6,......) e
  (28, 5, 23, 20, 27, 13,.....)

- Questo accorgimento non elimina l'agglomerazione *secondaria*, dovuta a chiavi che vengono associate allo stesso indirizzo iniziale

- Una soluzione del problema puo' essere ottenuta con una legge di scansione *casuale*: ogni chiave ha la stessa probabilita' di essere assegnata ad una qualsiasi delle pagine non sature, indipendentemente dal numero delle pagine sature
  $H_1(k) = H_0(k)$
  $H_i(k) = (H_{i-1}(k) + s_i) \text{ Mod } M \quad (i > 1)$
  dove s_i e' una sequenza di numeri casuali diversi nell'intervallo [1, M-1] generata prendendo la chiave come seme

GESTIONE DEI TRABOCCHI

Metodi di indirizzamento aperto

- Sebbene la trasformazione casuale sia semplice da realizzare, di solito non viene presa in considerazione per applicazioni su files che occupano piu' cilindri, perche' la natura casuale degli indirizzi generati dalla legge di scansione comporta ritardi nei tempi di risposta dovuti agli spostamenti delle testine di lettura

- Per questa ragione, quando si usa il metodo di indirizzamento aperto, si preferisce adottare una legge di scansione lineare con passo unitario nonostante i fenomeni di agglomerazione

GESTIONE DEI TRABOCCHI

Metodi di concatenamento in area primaria

Catene confluenti (coaleshed chaining).

- Un trabocco da una pagina primaria i viene memorizzato nella prima pagina non piena i+h, e si attiva un riferimento da i a i+h

- Tutte i records per cui $H(k_i)$ = i+h vengono memorizzate nella pagina i+h finche' questa non diviene satura

- Quando la pagina i+h da' luogo da un trabocco, si procede in modo analogo

- Esempio:
  c=2 e una sequenza di chiavi tali che:
  $H(k_1) = H(k_2) = H(k_3) = i$
  $H(k_4) = i+h$, $H(k_5) = i$, $H(k_6) = i + h$
  si ha la fusione delle catene

GESTIONE DEI TRABOCCHI

Metodi di concatenamento in area primaria

Catene distinte(separate chaining).

- Per evitare la fusione delle catene, tutti records che collidono vengono collegati a lista

- Mentre nel caso delle catene confluenti, le catene collegano le pagine, in questo caso collegano i records

- Quando la trasformazione associa un record ad una pagina satura, ma occupata da trabocchi, uno di essi si memorizza altrove

- Questo metodo migliora ulteriormente le prestazioni, ma complica la gestione dei trabocchi

GESTIONE DEI TRABOCCHI

Metodi di concatenamento in area separata

- I trabocchi vengono memorizzati in un'area di memoria, distinta da quella primaria, che puo' essere unica o distribuita per gruppi di pagine consecutive dell'area primaria

- L'area separata e' in generale impaginata ed ogni pagina puo' essere dedicata a trabocchi provenienti dalla stessa pagina dell'area primaria, oppure a trabocchi provenienti da pagine diverse

- Nel primo caso si collegano a lista le pagine con trabocchi provenienti dalla stessa pagina, mentre nel secondo caso si collegano a lista i trabocchi

- La capacita' delle pagine dell'area separata non e' necessariamente uguale a quella delle pagine dell'area primaria, ma nella letteratura non vengono date regole da seguire

GESTIONE DEI TRABOCCHI

Fattore di caricamento

- Oltre che dalla trasformazione, il fenomeno del trabocco dipende anche dal fattore di caricamento d: al diminuire d si riducono i trabocchi, ma si aumenta lo spazio di memoria occupato dal file

- Al fine di trovare il giusto compromesso, si sviluppano alcune considerazioni di tipo statistico per analizzare l'effetto della capacita' delle pagine e del fattore di caricamento sotto le seguenti ipotesi:

  - la trasformazione e' uniforme, pertanto ogni indirizzo ha probabilita' p=1/M di essere generato ad ogni operazione di indirizzamento e la probabilita' q=(1 -1/M) di non essere generato;
  - l'archivio e' statico, altrimenti i risultati valgono solo nella fase di caricamento;
  - i trabocchi sono memorizzati in area separata

- Poiche' gli indirizzi sono generati indipendentemente, applicando la funzione H ad N chiavi, la probabilita' che un indirizzo si presenti ripetuto x volte e' $p^x q^{N-x}$

- Il numero dei possibili eventi, cioe' di sequenze di N indirizzi x dei quali uguali e' (N su x)

GESTIONE DEI TRABOCCHI

Fattore di caricamento

- Pertanto la probabilita' che fra gli N indirizzi generati se ne presenti uno ripetuto x volte, indipendentemente dall'ordine in cui compare e' data dalla distribuzione binomiale

$$P(x) = \binom{N}{x} p^x q^{N-x}$$

- Questa distribuzione puo' essere usata anche per stimare il numero di indirizzi ai quali verranno assegnati un certo numero di records

- Infatti moltiplicando M per la probabilita' che un certo indirizzo sia generato x volte, si ottiene il numero medio di pagine con x registrazioni, dato da $M*P(x)$

- Nell'ipotesi che N ed M siano grandi, sempre valida per i databases, la distribuzione binomiale e' approssimata dalla distribuzione di Poisson:

  $P(x) = (m^x e^{-m})/x!$ $\quad$ $(m= N/M)$

GESTIONE DEI TRABOCCHI

Fattore di caricamento

La tabella I contiene i valori della distribuzione di Poisson al variare di m=N/M

GESTIONE DEI TRABOCCHI

Fattore di caricamento

- Il numero totale dei trabocchi e':
  Nt = N - (registrazioni in pagine senza trabocchi +
  registrazioni in pagine con trabocchi)=
  = N (

- dove Pc e' la probabilita' che un indirizzo venga generato piu' di c volte e':

- Nt/N =

(ricordando che d*c=N/M   abbiamo che M/N=1/d*c)

Nt/N =

La seguente tabella riporta i valori di 100*Nt/N

GESTIONE DEI TRABOCCHI

Fattore di caricamento

- Quindi per dimensionare il file si procede come segue:

  - siano N le registrazioni da memorizzare
  - sia la capacita' delle pagine fissata
  - si sceglie un fattore di caricamento (l'esperienza suggerisce un valore tra 0.75 e 0.85)
  - si determina il numero delle pagine necessarie
  - dalla tabella II si valuta la percentuale dei trabocchi. Se e' ritenuta eccessiva, bisogna ridurre d. Se l'obiettivo e' di ridurre il numero delle pagine, a scapito del fattore di caricamento, e' necessario aumentare d (la tabella tabula 100(Nt/N))

d	0.5	0.7	0.9	1.0
c				
1	21.31	28.08	34.06	36.79
5	2.48	7.11	13.78	17.55
10	0.44	2.88	8.59	12.51
50	0.00	0.05	2.04	5.63

DIMENSIONE DELLA PAGINA

- La capacita' della pagina e' un fattore molto importante per le prestazioni di un'organizzazione hash

- Supponiamo di dover memorizzare 750 records in 1000 pagine con c=1, oppure in 500 pagine con c=2

- In entrambi i casi il fattore di caricamento e' d=0.75
  Nel primo caso si trova che i trabocchi sono il 29.6%
  Nel secondo caso, pur aumentando, le collisioni perche' si e' ridotto M, i trabocchi diventano il 18.7% con una riduzione del 37%

- Perche' i trabocchi degradano sensibilmente le prestazioni, si puo' pensare di aumentare la capacita' della pagina

- Ad esempio, mantenendo il fattore di caricamento d=0.75, ma ponendo c=10, le prestazioni migliorano

- Da uno studio effettuato da Severance (1976) risulta che per organizzazioni su memoria secondaria, e' preferibile non usare l'organizzazione hash se la capacita' delle pagine e' inferiore a 10

DIMENSIONE DELLA PAGINA

Organizzazione con bucket

- La dimensione della pagina, pero', dipende dal sistema operativo
- Si puo' usare un'organizzazione con buckets:
  - il file dei dati e' suddiviso in un numero B di bucket (dove ogni bucket consiste di uno o piu' pagine)
  - viene allocato un *bucket directory* che contiene un numero B di puntatori uno per ogni bucket
- La trasformazione della chiave determina il numero del bucket
- Si consulta il bucket directory per determinare il primo blocco del bucket che viene poi acceduto per cercare il record; se nel primo blocco il record non e' trovato si passa al successivo e cosi' via

HASH DINAMICO

- L'organizzazione hash ha lo svantaggio di richiedere un'allocazione statica della memoria, che deve essere stimata al momento della progettazione iniziale. Nel caso di files dinamici, il costo delle operazioni aumenta rapidamente

- Sono state proposte tecniche dinamiche che si propongono di aumentare o diminuire l'allocazione di spazio a seconda delle dimensioni del file, senza richiedere riorganizzazioni del file

- Gli approcci possono essere suddivisi in due categorie:
  (1) quelli che fanno uso di strutture ausiliarie
  virtual hashing, extensible hashing, dynamic hashing
  (2) quelli che agiscono esclusivamente sull'area primaria
  linear hashing, spiral hashing

- In entrambi i casi, la funzione di trasformazione della chiave viene modificata opportunamente quando l'organizzazione si ristruttura, in modo da consentire sempre il recupero di ogni record con un numero medio di accessi costante e molto basso (vicino a 1)

VIRTUAL HASHING (Litwin 1978)

Il metodo funziona nel modo seguente:

a) inizialmente si alloca per l'area dati un certo numero M di pagine contigue di capacita' c. M puo' essere piccolo (inizialmente l'autore fece gli esperimenti con M=7)
b) si introduce un vettore binario B con tanti elementi quante sono le pagine dell'area dati. Quando una pagina dell'area dati viene utilizzata, il corrispondente elemento di B viene posto ad 1
c) si utilizza una funzione di trasformazione H_0 che applicata ad una chiave k produce un indirizzo compreso tra 0 e M-1. Se inserendo un record con chiave k' nella pagina di indirizzo m si genera un trabocco, allora:

  - si raddoppia l'area primaria, mentenendo contigue le pagine
  - si sostituisce H_0 con H_1 la quale produce indirizzi compresi tra 0 e 2M-1
  - si applica H_1 a k' e a tutte e sole le chiavi dei records nella pagina che ha dato luogo al trabocco, per stabilire in quali pagine memorizzarle

VIRTUAL HASHING (Litwin 1978)

Questo richiede l'uso di una serie di funzioni di trasformazione H_0, H_1,.....,H_r; H_r in generale restituisce un indirizzo compreso tra 0 e $2^r M$ -1

L'indice della funzione H indica il numero di raddoppi subiti dall'area dati; le funzioni H devono soddisfare le seguenti proprieta':

$H_{j+1}(k) = H_j(k)$ oppure $H_{j+1}(k) = H_j(k) + 2^j M$

con j=r, r-1,.....,0

In altre parole, applicando H_{j+1} alle chiavi k con $H_j(k)$ =m, i nuovi indirizzi generati possono essere soltanto m, oppure m+M, cioe' quello corrispondente alla seconda meta' dell'area raddoppiata

La funzione adottata da Litwin e':$H_r(k) = k \bmod 2^r M$

Se k e una chiave e r e' il numero di raddoppi subiti dall'area dati, l'indirizzo della pagina che contiene k viene calcolato con la seguente funzione:

```
rec RicercaPagina (r: int, k: int) :int:=
    if r<0 then print "chiave non esistente"
    else if   B(Hr(k))=1 then Hr(k)
              else RicercaPagina (r-1, k))
```

VIRTUAL HASHING (Litwin 1978)

Gestione dei trabocchi

- Quando si ha il primo trabocco da una pagina $p<M$ e l'area dati e' quella iniziale, si raddoppia l'area dei dati e si distribuiscono i records della pagina p fra le pagine p e $p+M$

- Se l'area dati e' stata gia' raddoppiata una volta per effetto di un trabocco da un'altra pagina, basta distribuire i records della pagina p fra le pagine p e $p+M$, con la pagina $p+M$ gia' esistente, ma non ancora utilizzata

- Il procedimento si complica se ci sono stati gia' r raddoppi dall'area dati perche' occorre stabilire in quali delle possibili (2r - 1) pagine occorre distribuire i records della pagina p

VIRTUAL HASHING (Litwin 1978)
Esempio

Si supponga di inserire la chiave 3820 nel seguente file, con M=7, c=3

Applicando la funzione $H_r(k)=H_0(3820)$ si ottiene m=5; poiche' B(5)=1, la nuova chiave va memorizzata nella pagina di indirizzo 5.

Questa pagina e' satura percio' occorre raddoppiare l'area dati

I record della pagina di indirizzo 5 sono divise tra tale pagina e una nuov pagina di indirizzo m'=12, mediante la funzione H_1. Il vettore B viene raddoppiato e i suoi valori aggiornati

VIRTUAL HASHING (Litwin 1978)
Esempio (continua)

D'ora in avanti si applica la funzione H_1.
Supponiamo di dover inserire la chiave 3343, ad essa va applicata la trasformazione $H_1(3343)=11$

VIRTUAL HASHING (Litwin 1978)

Esempio (continua)

Il bit a zero indica pero' che questa pagina non e' stata ancora utilizzata: per questa ragione si deve usare la trasformazione $H_0(3343)=4$

Essendo la pagina 4 satura e' necessario ricorrere alla procedura per la risoluzione dei trabocchi, che pero' in questo caso non richiede il raddoppio dell'area primaria, in quanto la pagina 11 gia' esiste nell'archivio.

E' sufficiente attivare questa pagina ponendo a 1 il bit nel vettore B e trasformare con H_1 tutte le chiavi della pagina 4 (7830, 1075, 6647) piu' la nuova chiave 3343 dividendole tra le due pagine 4 e 11

EXTENDIBLE HASHING (Fagin et Al. 1979)

- L'extendible hashing garantisce il reperimento di ogni record in al piu' due accessi alla memoria secondaria

- Rispetto al virtual hashing impiega un indice piu' complesso ma non raddoppia l'area dati

- L'espansione dell'area dati avviene aggiungendo una nuova pagina ogni volta che si tenta di inserire un record in una pagina satura

- Per mantenere traccia delle evoluzioni dell'area dati si usa un indice che si estende e si contrae dinamicamente e che contiene i riferimenti alle pagine dell'area dati

- La funzione di trasformazione associa ad ogni chiave una stringa binaria di opportuna lunghezza (ad es. 32 bits) denominata *pseudochiave*.

- Le pseudochiavi non vengono usate per accedere direttamente ad una pagina dell'area dati, ma con i primi p caratteri si accede ad un indice B di dimensione 2^p

EXTENDIBLE HASHING (Fagin et Al. 1979)

- L'elemento i-simo di tale indice e' un riferimento alla pagina dell'area dati contenente tutte e sole le registrazioni con pseudochiavi con lo stesso prefisso di lunghezza p' con $1 \leq p' \leq p$

- Ricerca: si estraggono i primi p bits dalla pseudochiave; si accede l'entrata nell'indice che corrisponde alla stringa di p bits; dall'entrata si determina l'indirizzo della pagina del file che contiene il record cercato

- Gestione dei trabocchi:
  Supponiamo che inizialmente la struttura si costituita da un indice con un elemento e da un'area dati primaria di una pagina (p=p'=0)

EXTENDIBLE HASHING (Fagin et Al. 1979)

Gestione dei trabocchi (continua):
Quando si deve inserire un record in una pagina satura a cui e' associato un certo valore p', si procede come segue:

1) viene allocata una nuova pagina: tanto alla vecchia quanto alla nuova viene associato il valore p'+1

2) i records si suddividono fra le due pagine in base al valore dei primi p'+1 bit della loro pseudochiave e si aggiorna B facendo in modo che un elemento di esso contenga un riferimento alla nuova pagina

3) se p'=p, prima di procedere all'aggiornamento di B, si raddoppia la sua dimensione, si modificano opportunamente i riferimenti e si incrementa p di uno

EXTENDIBLE HASHING (Fagin et Al. 1979)
Esempio

Informazioni sulle filiali
Funzione hash sull'attributo branch-name

Supponiamo che ogni pagina contenga solo due records

Inizialmente le strutture sono vuote; lo zero all'inizio della pagina indica che zero bits della funzione hash h(k) sono necessari per determinare la posizione corretta per il record di chiave k

EXTENDIBLE HASHING (Fagin et Al. 1979)
Esempio

- Supponiamo di inserire inizialmente i records per le filiali Perryridge e Round Hill

- Supponiamo di inserire il record per la filiale Downtown; si determina che la pagina e' piena;
  p' diventa 1;
  si alloca un'altra pagina e si dividono i records in modo da allocare quelli la cui chiave comincia con 1 nella nuova pagina, mentre quelli la cui chiave comincia con 0 restano nella vecchia pagina

EXTENDIBLE HASHING (Fagin et Al. 1979)

Esempio

- Supponiamo adesso di inserire il record con chiave Redwood

- Poiche' il primo bit di h(Redwood) e' 1, tale record va inserito nella pagina puntata dall'elemento dell'indice che corrisponde alla stringa di bit '1'

- Tale pagina e' piena e quindi e' necessario allocarne un'altra e incrementare il numero di bits usati dalla funzione hash. Tale numero va portato a 2

- Questo causa che l'indice deve essere raddoppiato portando il numero di entrate a 4

- Notare che poiche' la pagina puntata dell'entrata '0' non e' stata raddoppiata, le entrate '00' e '01' puntano alla stessa pagina del file

EXTENDIBLE HASHING (Fagin et Al. 1979)
Esempio

- Continuando si ottiene la seguente struttura

EXTENDIBLE HASHING (Fagin et Al. 1979)

Vantaggi dell'extensible hashing:

- non richiede il raddoppio delle pagine (come nel caso del virtual hashing)

- assicura l'accesso in al piu' due operazioni di I/O

- molto spesso l'indice e' piccolo e quindi puo' stare in memoria

- per esempio:
  se la dimensione delle pagine e' 4k, le chiavi sono lunghe 7 byte e i riferimenti alle pagine occupano 3 byte, B occupera' solo tre pagine dopo un milione di inserzioni

Svantaggio principale:

- Complessita' di implementazione

CONFRONTO TRA INDICI E FUNZIONI HASH

- L'uso di una tecnica piuttosto che di un'altra dipende spesso dal tipo di query

- Se la maggior parte delle queries ha la forma:
  select A1,A2,......An **from** R **where** Ai=C
  la tecnica hash e' preferibile

- Infatti:
  la scansione di un indice ha un costo proporzionale al logaritmo del numero di valori in R per Ai
  in una struttura hash il tempo di ricerca e' indipendente dalla dimensione del DB
  (notare che il costo del caso peggiore nelle funzioni hash puo' essere d'altra parte molto alto e non si riesce a stimare in modo significativo)

- Gli indici sono preferibili se le queries usano condizioni di range
  select A1,A2,......An **from** R **where** C1 $\leq$Ai $\leq$C2
  (Infatti e' difficile determinare funzioni hash che mantengono l'ordine)

- Quindi quasi tutti i sistemi usano tecniche di indici perche' e' difficile sapere a priori il tipo di interrogazioni

STRUTTURE DI ACCESSO PER APPLICAZIONI AVANZATE

- Metodi per l'accesso a testi e documenti

  In questo caso le interrogazioni non sono esatte come nei casi visti finora

  In generale si cerca di recuperare a seguito di una richiesta i documenti *probabilmente* rilevanti rispetto alla richiesta

  Organizzazioni piu' comunemente seguite:
  signature files, liste invertite sulle parole

- Metodi per applicazioni spaziali e geografiche

  Esempio: determinare tutte le citta' contenute in una data area geografica

  Organizzazioni piu' comunemente seguite:
  quad tree, grid files, R-trees

ESECUZIONE DI INTERROGAZIONI

- Abbiamo visto finora come organizzare i dati in un D B

- Normalmente le decisioni sulle strutture da allocare sono determinate durante la progettazione fisica della DB

- La modifica di tali strutture in seguito puo' essere costosa

- Quindi quando una query e' presentata al sistema occorre determinare il modo piu' efficiente per eseguirla usando le strutture disponibili

- Per queries complesse esistono piu' strategie possibili

- Anche se il costo di determinare la strategia ottima puo' essere alto, e' tuttavia molto spesso utile eseguire l'ottimizzazione

- Le tecniche di ottimizzazione sono state sviluppate principalmente per il modello relazionale
  (tecniche per gli OODBMSs costituiscono un'area di ricerca aperta)

PASSI NELL'ESECUZIONE DI UNA INTERROGAZIONE

- Parsing
  Viene controllata la correttezza sintattica della query e ne viene generata una rappresentazione interna (parse tree)

- Trasformazioni algebriche
  Questo e' il primo passo di ottimizzazione
  La query viene trasformata in una query equivalente ma piu' efficiente da eseguire (ci si basa sulle proprieta' dell'algebra relazionale)

- Selezione della strategia
  Si determina in modo preciso come la query sara' eseguita (per esempio si determina che indici si useranno)
  La scelta della strategia e' fatta principalmente in base al numero di accessi a disco

- Esecuzione della strategia scelta

E' possibile eseguire alcuni dei passi a tempo di compilazione del programma (DB2 e System R usano questa strategia) o a tempo di esecuzione (Oracle usa questa strategia)

EQUIVALENZA DI ESPRESSIONI

Selezione

- Relazioni esempio:
  customer (c-name, street, c-city)
  deposit (b-name, account-number, c-name, balance)
  branch(b-name, assets, b-city)

- Supponiamo di voler determinare assets e b-name per le filiali che hanno clienti residenti a Port Chester

- Questa interrogazione e' espressa in RA come segue
  $\Pi_{\text{b-name, assets}} (\sigma_{\text{c-city="Port Chester"}}$
  $(\text{customer} \;|x|\; \text{deposit} \;|x|\; \text{branch}))$ (Q1)

- Questa espressione genera una relazione come join naturale delle tre relazioni

- Questa relazione intermedia puo' risultare troppo grande per risiedere in MM e quindi puo' essere necessario memorizzarla su disco

EQUIVALENZA DI ESPRESSIONI
Selezione

- E' possibile eseguire Q1 piu' efficientemente se si riesce a ridurre la dimensione della relazione intermedia generata come join naturale

- Una espressione equivalente ma piu' efficiente e' la seguente:
  $\Pi_{\text{b-name, assets}}$ (($\sigma_{\text{c-city="Port Chester"}}$ (customer))
  |x| deposit |x| branch)

- L'espressione precedente suggerisce una prima regola per trasformare queries

  Eseguire le operazioni di selezione (σ) *il piu' presto possibile* (R1)

EQUIVALENZA DI ESPRESSIONI
Selezione

- Consideriamo una query uguale alla precedente con in piu' la restrizione che i clienti abbiano un conto con un bilancio maggiore di 1000

- Questa interrogazione e' espressa in RA come segue

  $\Pi_{\text{b-name, assets}}$ (σ c-city="Port Chester" AND balance >1000 (customer |x| deposit |x| branch)) (Q2)

- Non e' possibile applicare la selezione

  c-city="Port Chester" AND balance >1000

  direttamente alla relazione customer poiche' il predicato coinvolge attributi di customer e deposit

- Osserviamo che:

  (customer |x| deposit |x| branch) ≡
  ((customer |x| deposit) |x| branch)

- Pertanto Q1 puo' essere riscritta come segue

  $\Pi_{\text{b-name, assets}}$ ((σ c-city="Port Chester" AND balance >1000 (customer |x| deposit)) |x| branch)

EQUIVALENZA DI ESPRESSIONI
Selezione

- Consideriamo la sotto-query
  $\sigma_{\text{c-city="Port Chester" AND balance >1000}}$
  (customer |x| deposit) (E1)

- Questa sottoquery puo' essere riscritta come segue
  $\sigma_{\text{c-city="Port Chester"}}$ ($\sigma_{\text{balance >1000}}$
  (customer |x| deposit)) (E2)

- L'espressione E2 permette di applicare la regola R1

- Pertanto la sottoquery puo' essere riscritta come segue:
  ($\sigma_{\text{c-city="Port Chester"}}$ (customer)) |x|
  ($\sigma_{\text{balance >1000}}$(deposit))

- Le trasformazioni precedenti sono basate sulla seguente regola
  Trasforma espressioni della forma $\sigma_{P1 \text{ AND } P2}(e)$
  in espressioni della forma $\sigma_{P1}(\sigma_{P2}(e))$
  dove P1 e P2 sono predicati ed e e' una espressione algebrica (R2)

- R2 e' basata sulle seguenti equivalenze tra RA espr.
  $\sigma_{P1}(\sigma_{P2}(e)) \equiv \sigma_{P2}(\sigma_{P1}(e)) \equiv \sigma_{P1 \text{ AND } P2}(e))$

EQUIVALENZA DI ESPRESSIONI
Join naturale

- In generale trasformare le interrogazioni in modo che le selezioni siano applicate il prima possibile permette di ridurre la dimensione dei risultati intermedi

- Un altro modo per ridurre tale dimensione e' di determinare un ordine ottimale nell'esecuzione delle operazioni di join naturale

- Questa strategia e' basata sulla proprieta' del join naturale di essere una operazione associativa:
  $(r1 \ |x| \ r2) \ |x| \ r3 \equiv r1 \ |x| \ (r2 \ |x| \ r3)$

- Sebbene le due precedenti espressioni sono equivalenti, il costo di calcolarle puo' essere diverso

- Consideriamo nuovamente la query
  $\Pi_{b\text{-}name,\ assets}$ (($\sigma_{c\text{-}city=\text{"Port Chester"}}$ (customer))
  |x| deposit |x| branch)

- Una possibilita' e' di eseguire il join naturale
  deposit |x| branch e di eseguire il join naturale del risultato con il risultato della sottoquery
  ($\sigma_{c\text{-}city=\text{"Port Chester"}}$ (customer))

EQUIVALENZA DI ESPRESSIONI
Join naturale

- T1 = deposit |x| branch
  (σ $_{\text{c-city="Port Chester"}}$ (customer)) |x| T1

- E' da notare che pero' la relazione temporanea T1 e' molto probabilmente di dimensioni elevate in quanto contiene una tupla per ogni deposito

- Invece la relazione che si otterebbe dalla sottoquery
  σ $_{\text{c-city="Port Chester"}}$ (customer)
  e' molto probabilmente piccola in quanto per una banca con molte filiali e' improbabile che tutti i clienti abbiano la residenza nello stesso posto

- Quindi se calcoliamo
  (σ $_{\text{c-city="Port Chester"}}$ (customer))|x| deposit
  otteniami una tupla per ogni conto intestato ad un cliente che abita a Port Chester

- Pertanto la relazione temporanea che si deve memorizzare e' molto piu' piccola

- $\Pi_{\text{b-name, assets}}$
  (((σ $_{\text{c-city="Port Chester"}}$ (customer)) |x| deposit)
  |x| branch) (Q3)

EQUIVALENZA DI ESPRESSIONI
Join naturale

- Un'altra proprieta' del join e' di essere commutativo
  r1 |x| r2 ≡ r2 |x| r1

- Pertanto nel caso di Q3 potremmo considerare una strategia alternativa
  $\Pi_{\text{b-name, assets}}$
  $(((\sigma_{\text{c-city="Port Chester"}}$ (customer)) |x| branch)
  |x| deposit)

- Questa strategia non e' molto buona in quanto customer e branch non hanno attributi a comune e quindi il join natturale diventa un prodotto Cartesiano

- Se customer ha c tuple e branch ne ha b si ottiene un totale di b*c tuple

- Quindi si otterrebbe una relazione temporanea di dimensione elevata

EQUIVALENZA DI ESPRESSIONI
Proiezione

- L'operazione di proiezione ha la proprieta' di ridurre la dimensione della relazione: quindi e' conveniente applicare le proiezioni il prima possibile

- Consideriamo la query
  $\Pi_{\text{b-name, assets}}$
  $(((\sigma_{\text{c-city="Port Chester"}}(\text{customer}))\ |x|\ \text{deposit})$
  $|x|\ \text{branch})$ (Q3)

- Quando si calcola la sottoquery
  $((\sigma_{\text{c-city="Port Chester"}}(\text{customer}))\ |x|\ \text{deposit})$
  otteniamo una relazione il cui schema e'
  (c-name, c-city, b-name, account-number, balance)

- E' possibile eliminare alcuni attributi da questo schema. Gli unici attributi da non eliminare sono quelli che
  - appaiono nel risultato della query
  - sono necessari in operazioni successive

- Pertanto Q3 puo' essere riscritta come segue:
  $\Pi_{\text{b-name, assets}}((\Pi_{\text{b-name}}$
  $((\sigma_{\text{c-city="Port Chester"}}(\text{customer}))\ |x|\ \text{deposit}))$
  $|x|\ \text{branch})$

EQUIVALENZA DI ESPRESSIONI

Altre operazioni

- Le equivalenze viste sono le piu' utili in quanto coinvolgono operazioni usate molto spesso

- Altre equivalenze utili sono:

  - $\sigma_P(e1 \cup e2) \equiv \sigma_P(e1) \cup \sigma_P(e1)$
  - $\sigma_P(e1 - e2) \equiv \sigma_P(e1) - \sigma_P(e1)$
  - $(e1 \cup e2) \cup e3 \equiv e1 \cup (e2 \cup e3)$
  - $e1 \cup e2 \equiv e2 \cup e1$

EQUIVALENZA DI ESPRESSIONI
Sommario delle leggi di equivalenza

1. Leggi commutative per il join e il prodotto Cartesiano

   $e1 \ |x|_F \ e2 \equiv r2 \ |x|_F \ e1$

   $e1 \ |x| \ e2 \equiv e2 \ |x| \ e1$

   $e1 \ x \ e2 \equiv e2 \ x \ e1$

2. Leggi associative per il join e il prodotto Cartesiano

   $(e1 \ |x|_{F1} \ e2) \ |x|_{F2} \ e3 \equiv e1 \ |x|_{F1} \ (e2 \ |x|_{F2} \ e3)$

   $(e1 \ |x| \ e2) \ |x| \ e3 \equiv e1 \ |x| \ (e2 \ |x| \ e3)$

   $(e1 \ x \ e2) \ x \ e3 \equiv e1 \ x \ (e2 \ x \ e3)$

3. Cascata di proiezioni

   $\Pi_{A1,....,An}(\Pi_{B1,....,Bm}(e)) \equiv \Pi_{A1,....,An}(e)$

   Notare che {A1,.....,An} $\subseteq$ {B1,.....,Bm} affinche' la cascata sia legale

4. Cascata di selezioni

   $\sigma_{P1}(\sigma_{P2}(e)) \equiv \sigma_{P1 \ AND \ P2}(e))$

   inoltre poiche' P1 AND P2 = P2 AND P1
   segue che la selezione gode della proprieta' commutativa

   $\sigma_{P1}(\sigma_{P2}(e)) \equiv \sigma_{P2}(\sigma_{P1}(e))$

EQUIVALENZA DI ESPRESSIONI
Sommario delle leggi di equivalenza

5. Commutazione di selezione e proiezione

Se una selezione con predicato P coinvolge solo gli attributi A1,......,An, allora

$$\Pi_{A1,....,An}(\sigma_P(e)) \equiv \sigma_P(\Pi_{A1,....,An}(e))$$

Piu' in generale, se il predicato P coinvolge anche gli attributi B1,.....,Bm che non sono tra gli attributi A1,......,An allora

$$\Pi_{A1,....,An}(\sigma_P(e)) \equiv$$
$$\Pi_{A1,....,An}(\sigma_P(\Pi_{A1,....,An,B1,....,Bm}(e)))$$

6. Commutazione di selezione e prodotto Cartesiano

Se una selezione con predicato P coinvolge solo gli attributi di e1, allora

$$\sigma_P(e1 \times e2) \equiv \sigma_P(e1) \times e2$$

Come corollario, se P=P1 AND P2 dove P1 coinvolge solo gli attributi di e1 e P2 quelli di e2, usando le regole (1), (4), e (6) si ottiene

$$\sigma_P(e1 \times e2) \equiv \sigma_{P1}(e1) \times \sigma_{P2}(e2)$$

Inoltre se P1 coinvolge solo attributi di e1, mentre P2 coinvolge attributi di e1 e di e2

$$\sigma_P(e1 \times e2) \equiv \sigma_{P2}(\sigma_{P1}(e1) \times e2)$$

EQUIVALENZA DI ESPRESSIONI
Sommario delle leggi di equivalenza

7. Commutazione di selezione e unione
   Se e = e1 U e2 possiamo assumere che gli attributi di e1 ed e2 hanno gli stessi nomi degli attributi di e, o almeno, che esiste una corrispondenza che associa ad ogni attributo di e rispettivamente un unico attributo di e1 ed un unico attributo di e2

   $$\sigma_P(e1 \cup e2) \equiv \sigma_P(e1) \cup \sigma_P(e2)$$

   Se gli attributi di e1 e/o di e2 sono diversi P deve essere modificato usando il nome appropriato

8. Commutazione di selezione e differenza

   $$\sigma_P(e1 - e2) \equiv \sigma_P(e1) - \sigma_P(e2)$$

9. Commutazione di proiezione e prodotto Cartesiano
   Sia A1,....,An una lista di attributi di cui gli attributi B1,.....,Bm siano attributi di e1, e i rimanenti C1,.....,Ck siano attributi di e2

   $$\Pi_{A1,....,An}(e1 \times e2) \equiv \Pi_{B1,....,Bm}(e1) \times \Pi_{C1,....,Ck}(e2)$$

10. Commutazione di proiezione e unione

   $$\Pi_{A1,....,An}(e1 \cup e2) \equiv \Pi_{A1,....,An}(e1) \cup \Pi_{A1,....An}(e2)$$

STIMA DEL COSTO DI ESECUZIONE

- La strategia scelta dipende dalla dimensione di ogni relazione e dalla distribuzione dei valori nelle varie colonne

- Normalmente i DBMSs mantengono statistiche per ogni relazione memorizzata:
  1. n_r numero di tuple nella relazione r
  2. S_r dimensione di una tupla della relazione r in bytes (per tuple a lunghezza fissa, altrimenti si usano valori medi)
  3. V(A, r) il numero di valori distinti che appaiono nella relazione r per l'attributo A

- Le prime due statistiche permettono di stimare il costo di un prodotto Cartesiano
  - r x s conterra' un numero di tuple pari a $n_r * n_s$
  - una tupla di r x s ha dimensione pari a
    $S_r + S_s$
  - pertanto la dimensione di r x s e' pari a
    $n_r * n_s *(S_r + S_s)$

STIMA DEL COSTO DI ESECUZIONE

- La terza statistica permette di stimare quante tuple verificano un predicato di selezione della forma:
  <attribute-name> = <value>

- Per eseguire una stima corretta e' necessario determinare la frequenza di ogni valore per una data colonna

- Sotto l'ipotesi che ogni valore appare con la stessa probabilita', allora si stima che $\sigma_{A=a}(r)$ ha una seleziona un numero di tuple pari a $n_r/V(A, r)$
  (La quantita' $1/V(A,r)$ e' detta *fattore di selettivita' del predicato*)

- In alcune situazioni non e' realistico assumere l'equiprobabilita' dei valori

- Per esempio se consideriamo la relazione deposit e l'attributo b-name, ci si puo' aspettare che le filiali piu' grandi abbiano piu' depositi e quindi alcuni nomi di filiale appariranno con maggiore probabilita' di altri

- Tuttavia l'ipotesi di equiprobabilita' e' una buona approssimazione in molto casi

STIMA DEL COSTO DI ESECUZIONE
Join Naturale

Siano r1 e r2 due relazioni di schema rispettivamente R1 e R2

- se $R1 \cap R2 = \emptyset$, allora r1 |x| r2 e' lo stesso di r1 x r2 e si puo' usare la stessa formula di stima usata per il prodotto Cartesiano

- se $R1 \cap R2$ e' una chiave per R1, allora una tupla di r2 e' connessa tramite il join con al piu' una tupla di r1; quindi $n_{(r1\ |x|\ r2)} \leq n_{r2}$

- se $R1 \cap R2$ non e' una chiave per nessuno dei due schemi, si deve usare la terza statistica. Consideriamo una tupla t di r1, e supponiamo che $R1 \cap R2 = \{A\}$
  Il numero di tuple di r2 che hanno t[A] come valore per A sono $n_{r2}/V(A, r2)$
  Pertanto la tupla t produce $n_{r2}/V(A, r2)$ tuple di r1 |x| r2
  Se consideriamo tutte le tuple in r1, allora otteniamo $n_{r1} * n_{r2}/V(A, r2)$ (i)

- Scambiando i ruoli

STIMA DEL COSTO DI ESECUZIONE
Join Naturale

- Scambiando i ruoli nel calcolo della stima otteniamo
  $n_{r1} * n_{r2} / V(A, r1)$ (ii)

  Espressioni (i) ed (ii) sono uguali se $V(A, r2) = V(A, r1)$

  Se $V(A, r2) \sim= V(A, r1)$ sono differenti allora esiste qualche tupla *dangling* che non partecipa al join

  In questo caso il minore dei due valori costituisce una stima migliore

- Esempio (1):

r1	A	B	r2	A	C	
	a1	b1		a1	c1	$n_{r1} = 4$
	a1	b2		a2	c2	$V(A,r1)=2$
	a2	b3		a1	c3	$n_{r2} = 6$
	a2	b4		a1	c6	$V(A,r2)=2$
				a2	c5	
				a2	c7	

$n_{r1} * n_{r2} / V(A, r2) = (4*6)/2 = 12$

STIMA DEL COSTO DI ESECUZIONE
Join Naturale

- Esempio (2):

r1	A	B	r2	A	C	
	a1	b1		a1	c1	n_{r1}= 6
	a1	b2		a2	c2	V(A,r1)=3
	a2	b3		a1	c3	
	a2	b4		a1	c6	n_{r2}= 6
	a3	b2		a2	c5	V(A,r2)=2
	a3	b3		a2	c7	

$n_{r1} * n_{r2} / V(A, r2) = (6*6)/2=18$
$n_{r1} * n_{r2} / V(A, r1) = (6*6)/3=12$

- Le stime precedenti non vanno molto bene se le due relazioni hanno pochi valori in comune per l'attibuto in comune; in tal caso la stima della dimensione del join risulta troppo alta

- Nelle pratica, d'altra parte, questi casi non si verificano molto spesso

- Nel caso in cui si dovessero avere molte tuple dangling, occorre applicare dei fattori di correzione

STIMA DEL COSTO DI ESECUZIONE
Strategie di join

- Abbiamo visto come stimare la dimensione del risultato di un'operazione di join

- Il costo dell'esecuzione di un join e' influenzato da vari fattori:
  - l'ordine fisico delle tuple in una relazione
  - la presenza e il tipo di indici (*clustering* vs *non-clustering*)
  - il costo di costruire un indice temporaneo allo scopo di eseguire il join

- Consideriamo l'espressione

  deposit |x| customer

  e supponiamo che non ci siano indici; inoltre
  - $n_{deposit} = 10{,}000$
  - $n_{customer} = 200$

- Strategie possibili
  - iterazione semplice
  - iterazione orientata ai blocchi
  - merge-join
  - uso di indici
  - three-way join

STIMA DEL COSTO DI ESECUZIONE
Iterazione semplice

- Si accede una tupla di deposit (*outer relation*) e si confronta con ogni tupla di customer (*inner relation*)

- **for each** tuple d **in** deposit **do**
  begin
      for each tuple c **in** customer **do**
          begin
          test pair (d,c) to see if a tuple should be added to the result
          end
  end

- Questa strategia richiede leggere ogni tupla di deposit una sola volta; questo puo' richiedere un massimo di 10,000 accessi

- Se le tuple sono clusterizzate allora il costo diminuisce sensibilmente;
  supponiamo di avere 20 tuple di deposit per blocco, il numero totale di accessi per la relazione deposit e' 10,000/20=500

STIMA DEL COSTO DI ESECUZIONE
Iterazione semplice

- Per quanto riguarda le tuple di customer queste vengono accedute per ogni tupla di deposit; quindi ogni tupla di customer viene acceduta 10,000 volte

- Poiche' le tuple di customer sono 200, il totale degli accessi e' 10,000*200 = 2,000,000

- Se le tuple di customer sono clusterizzate, il totale degli accessi si riduce

- Supponiamo di avere 20 tuple di customer per blocco; la scansione dell'intera relazione richiede solo l'accesso di 10 blocchi

- Quindi il costo della scansione della relazione customer per tutte le tuple di deposit e' 10,000*10=100,000

- Il costo totale dell'esecuzione del join nel caso in cui entrambe le relazioni siano clusterizzate e' 500 + 100,000 = 100,500

STIMA DEL COSTO DI ESECUZIONE
Iterazione orientata ai blocchi

- E' possibile migliorare la strategia precedente se si elaborano le relazioni sulla base dei blocchi e non delle tuple

- E' utile principalmente quando le tuple di una stessa relazione sono clusterizzate

- **for each** block Bd **of** deposit **do**

```
begin
      for each block Bc of customer do
            begin
              for each tuple b in Bd do
                begin
                  for each tuple c in Bc do
                        begin
            test pair (b,c) to see if a tuple should be
            added to the result
                        end
                end
            end
end
```

STIMA DEL COSTO DI ESECUZIONE
Iterazione orientata ai blocchi

- Questa strategia esegue il join esaminando un intero blocco di tuple di deposit alla volta

- Quindi il costo dell'accesso di deposit e' lo stesso della strategia di iterazione semplice (i.e. 500)

- Questa strategia implica che dobbiamo comunque scandire la relazione customer piu' volte (ad un costo di 10 a scansione)

- Tuttavia rispetto alla strategia di iterazione semplice, dobbiamo scandire la relazione customer tante volte quanti sono i *blocchi* di deposit e non quante sono le tuple di deposit

- Quindi il numero di scansioni della relazione customer e' 500; quindi il costo totale di accesso a customer e' 500*10=5000 accessi

- Il costo totale di questa strategia e' quindi
  500 + 5000 = 5500

STIMA DEL COSTO DI ESECUZIONE
Iterazione orientata ai blocchi

- La scelta di deposit come outer relation e customer come inner e' stata arbitraria

- Se avessimo usato customer come outer relation e deposit come inner il costo finale della strategia sarebbe stato 5010

- Un vantaggio nell'usare la relazione piu' piccola come inner e' che se la relazione e' piccola abbastanza puo' risiedere tutto il tempo in MM

- Se ad esempio customer fosse abbastanza piccola di risiedere in memoria, la strategia richiederebbe solo 500 accessi per leggere deposit e 10 per leggere customer, per un totale di 510.

STIMA DEL COSTO DI ESECUZIONE
Merge-Join

- Nel caso in cui nessuna delle due relazioni sia piccola abbastanza da poter risiedere in MM e' possibile eseguire efficientemente il join se entrambe le relazioni sono ordinate in base al valore dell'attributo di join

- Relazioni customer e deposit ordinate in base all'attributo customer-name

- L'operazione di merge-join richiede di associare inizialmente un puntatore ad ogni relazione

- I puntatori inizialmente puntano alla prima tupla di ogni relazione

- Poiche' le tuple sono ordinate in base all'attributo di join ogni tupla viene letta esattamente una volta

- Nel caso delle relazioni customer e deposit il costo totale sarebbe 510 accessi
  (nell'ipotesi che le tuple di ogni relazione sono clusterizzate)

STIMA DEL COSTO DI ESECUZIONE
Merge-Join

- L'algoritmo non richiede che la relazione entri tutta in MM; e' sufficiente che tutte le tuple con lo stesso valore dell'attributo di join stiano in MM

- Questo e' possibile anche per relazioni di ampie dimensioni

- Lo svantaggio principale di questo metodo e' che richiede che le relazioni siano ordinate

- Tuttavia, poiche' e' molto efficiente puo' convenire ordinare le relazioni prima di eseguire il join

STIMA DEL COSTO DI ESECUZIONE
Merge-Join: algoritmo

```
pd := address of first tuple of deposit;
pc := address of first tuple of customer;
while (pc ~= null) do
    begin
        tc :=tuple to which pc points;
        Sc:= {tc};
        set pc to point to  next tuple of customer;
        done := false;
        while (not done) do
            begin
                tc ':=tuple to which pc points;
                if tc [customer-name] =tc '[customer-name]
                then begin
                        Sc:= Sc U {tc'};
                        set pc  to point to next tuple of customer;
                end
                else done:=true;
            end
        td:=tuple to which pd points;
        while (td [customer-name] < tc [customer-name]) do
            begin
                set pd to point to  next tuple of deposit;
                td:=tuple to which pd points;
            end
        while (td [customer-name] = tc [customer-name]) do
            begin
                for each t in Sc do
                    begin
                        compute t |x| td and add this to the result;
                    end
                set pd to next tuple of deposit;
                td:=tuple to which pd points;

            end
end.
```

STIMA DEL COSTO DI ESECUZIONE
Merge-Join: Esempio

D

N	B	
a1	b1	i1
a1	b2	i2
a2	b4	i3
a2	b5	i4
a5	b7	i5

C

N	D	
a1	c1	i6
a1	c3	i7
a3	c5	i8
a4	c7	i9
a5	c9	i10

(i1,......,i10 sono indirizzi)

pd=i1 pc= i6

tc=<a1,c1> Sc={<a1,c1>}

pc=i7 tc'=<a1,c3>

poiche' tc'[N]=tc[N] Sc={<a1,c1>, <a1,c3>}

pc=i8

pd=i1 td=<a1,b1>

td[N]<tc[N] ? NO

td[N]=tc[N] ? SI

Si esegue il join tra la tupla td corrente e tutte le tuple in Sc

R={<a1,b1,c1>, <a1,b1,c3>}

STIMA DEL COSTO DI ESECUZIONE
Merge-Join: Esempio

pd=i2 td=<a1,b2>

td[N]=tc[N] ? SI

R = {<a1,b1,c1>, <a1,b1,c3>, <a1,b2,c1>, <a1,b2,c3>}

pd=i3 td=<a2,b4>

td[N]=tc[N] ? NO

tc=<a3,c5> Sc={<a3,c5>}

pc=i9 tc'=<a4,c7>

tc[N]=tc'[N]? NO

td[N] < tc[N] ? SI (td[N]=a2 tc[N]=a3)

pd=i4 td=<a2,b5>

td[N] <tc[N]? SI (td[N]=a2 tc[N]=a3)

pd=i5 td=<a5,b7>

td[N]<tc[N]? NO (td[N]=a5 tc[N]=a3)

td[N]=tc[N]? NO

pc=i9 tc=<a4,c7> pc=10

td[N]<tc[N]? NO (td[N]=a5 tc[N]=a4)

td[N]=tc[N]? NO

pc=10 tc=<a5,c9> td[N]=tc[N]? SI

<a5,b7,c9> viene aggiunto al risultato

STIMA DEL COSTO DI ESECUZIONE

Uso di indici

- Molto spesso l'attributo di join e' la chiave di ricerca di un indice di una delle relazioni

- Esempio: supponiamo che esista un indice sull'attributo c-name della relazione customer

- La strategia di iterazione semplice e' molto piu' efficiente in tal caso

-
```
for each tuple d of  deposit do
    begin
        for each tuple c of  customer do
            begin
             test pair (b,c) to see if  a tuple should be
            added to the result
            end
    end
```

- Data una tupla d di deposit non e' piu' necessario scandire l'intera relazione customer, ma e' sufficiente eseguire una ricerca sull'indice con il valore di c-name dato da d[c-name]

STIMA DEL COSTO DI ESECUZIONE
Uso di indici

- Senza l'uso dell'indice la strategia di iterazione semplice richiede un costo di 2 milioni di accessi per la relazione customer

- Usando l'indice e senza fare alcuna ipotesi sulla memorizzazione fisica della relazione, il join puo' essere calcolato con un numero significativamente minore di accessi

- Si devono eseguire comunque 10,000 accessi per leggere la relazione deposit

- Se $n_{customer}$ = 200 (come nel caso precedente) e ogni blocco dell'indice contine 20 puntatori, la scansione dell'indice ha un costo di 2 accessi

- Quindi e' necessario eseguire 3 accessi per ogni tuple di deposit, invece di 200; il costo totale della strategia e' di 40,000 accessi

- Sebbene il costo di 40,000 puo' sembrare alto e' da notare che nel caso della iterazione semplice si ha un costo minore solo nel caso in cui le tuple siano clusterizzate; in caso contrario puo' essere utile costruire un indice prima di eseguire il join

STIMA DEL COSTO DI ESECUZIONE
Three-way join

- Consideriamo branch |x| deposit |x| customer
  - $n_{deposit}$ = 10,000
  - $n_{customer}$ = 200
  - n_{branch}= 50

- E' necessario scegliere non solo il tipo di strategia di join (iterazione, merge-join, index) da eseguire ma anche in che ordine eseguire il join

- Strategia 1.
  - Si esegue (deposit |x| customer) usando una delle strategie viste; poiche' la c-name e' chiave di customer, il risultato di questo join e' di 10,000 tuple
  - Se si costruisce un indice sull'attributo b-name di branch si puo' calcolare
    branch |x| (deposit |x| customer)
    considerando ogni tupla t di
    (deposit |x| customer) e poi eseguendo una ricerca sull'indice con il valore t[b-name]
  - Poiche' b-name e' chiave per branch-name e' necessario esaminare una sola tupla per ognuna delle 10,000 tuple della relazione intermedia ottenuta come (deposit |x| customer)

STIMA DEL COSTO DI ESECUZIONE
Three-way join

- Strategia 2.
  Si calcola il join senza costruire alcun indice. Questo richiede esaminare 50*10000*200 possibilita', con un costo totale di 100,000,000

- Strategia 3.
  Invece di eseguire due joins in cascata, si eseguono contemporaneamente.
  - Questa tecnica richiede di costruire prima due indici
    sull'attributo b-name per branch
    sull'attributo c-name per customer
  - Quindi si considera ogni tupla t in deposit; per ogni t, si determinano le tuple in customer e le tuple in branch
  - Pertanto ogni tupla di deposit e' esaminata una volta soltanto

Questa strategia puo' essere conveniente, specialmente quando gli indici sulle relazioni sono gia' esistenti e non occorre costruirli appositamente

OTTIMIZZAZIONE - System R

Ipotesi:

Tipo delle interrogazioni

- Le interrogazioni hanno la forma seguente

  SELECT ListaDiAttributi
  FROM ListaDiRelazioni
  WHERE Condizione

  dove:

  - ListaDiAttributi indica gli attributi che interessano nel risultato (* sta per tutti gli attributi)

  - ListaDiRelazioni indica le relazioni coinvolte (per semplicita' consideriamo il caso di due relazioni)

  - Condizione e' una congiunzione di disgiunzioni di condizioni semplici del tipo

    Attributo Op Valore con $Op \in \{>, \geq, <, \leq, =, \neq\}$
    Attributo **isin** [$Valore_1$, $Valore_2$,.....,$Valore_N$]
    Attributo **within** ($Valore_1$, $Valore_2$)
    Attributo = Attributo′ con gli attributi appartenenti a relazioni diverse
    (tale predicato e' detto *predicato di join*)

OTTIMIZZAZIONE - System R

Tipo delle interrogazioni (continua)

- Si definisce *predicato di ricerca* (o *predicato risolubile*) un predicato per il quale:
  - esiste un indice utilizzabile per trovare le tuple che soddisfano il predicato
  - l'indice e' utile per ridurre il costo della verifica della condizione

  predicati del tipo
  C+D=1.000 oppure D ≠ 20 con C e D attributi con indici non sono predicati di ricerca in quanto gli indici non aiutano a limitare il numero di tuple da visitare

  Pertanto i predicati risolubili sono del tipo:

  Attributo ConIndiceOp Valore
  (con *Op* diverso da ≠)

  Attributo **isin** [$Valore_1$, $Valore_2$,.....,$Valore_N$]

  Attributo **within** ($Valore_1$, $Valore_2$)

  Predicato di join in cui almeno un attributo abbia un indice
- Si definisce *fattore booleano* un predicato, che se falso, rende falsa tutta l'interrogazione; tutte le tuple del risultato quindi verificano i fattori booleani.

OTTIMIZZAZIONE - System R

Tipo delle interrogazioni (continua)

- Esempio
  Relazione Impiegati(Codice, Nome, Lavoro, Salario, Obiettivo, Citta') con indici su tutti gli attributi

  Interrogazione
  SELECT Nome, Salario
  FROM Impiegati
  WHERE Lavoro = "Programmatore" AND
  (Citta'="Bologna" OR Salario >2000000)

  Il fattore booleano e' uno

Granularita' dell'ottimizzazione

L'ottimizzazione e' eseguita separatamente per ogni interrogazione, come accade in tutti i sistemi commerciali; pertanto e' bene formulare interrogazioni complesse e non scomporle in tante interrogazioni piu' semplici perche' cio' fa perdere i vantaggi dell'ottimizzazione.

OTTIMIZZAZIONE - System R

<u>Organizzazione fisica dei dati</u>

Nel recupero dei dati soddisfacenti un'interrogazione si utilizza la scansione sequenziale o si utilizzano indici allocati su singoli attributi

Gli indici per chiave primaria sono densi organizati come B+-tree

Gli indici per chiavi secondarie sono indici densi organizzati come B+-tree; le foglie contengono i valori degli attributi seguiti ciascuno dagli identificatori interni (TID) delle tuple che hanno quel valore per gli attributi

Gli indici (sia primari che secondari) si suddividono in *clustered* e *non-clustered*

Nel primo caso le tuple sono fisicamente ordinate in base ai valori dell'attributo dell'indice

Nel secondo caso le tuple non sono fisicamente ordinate in base ai valori dell'attributo dell'indice

Una pagina puo' contenere tuple di relazioni diverse

OTTIMIZZAZIONE - System R

Modello dei costi

Il costo di esecuzione di un piano di accesso e' espresso come una combinazione lineare del numero di accessi a MS, indicato con NPAG, e il numero di record NREC da esaminare in MM per generare il risultato finale

CA = NPAG + H*NREC

dove H e' il coefficiente del costo di CPU

(1/H indica il numero di confronti di tuple che sono considerati equivalenti al costo di un accesso da una pagina di disco)

Nel caso di scansione sequenziale NPAG=NPAG(R)

Nel caso di accesso con indici, NPAG e' costituito da due termini

NPAG=CI+CD

dove CI e' costo di scansione degli indici

e CD il costo dovuto all'accesso alle pagine contenenti i dati

OTTIMIZZAZIONE - System R

<u>Statistiche</u>

L'ottimizzatore del system R assume le seguenti ipotesi:

- uniformita' della distribuzione dei valori
- non correlazione tra valori di attributi diversi

Per ogni relazione R

- CARD(R): numero dei records in R
- NPAG(R): numero delle pagine
- NA(R): numero degli attributi della relazione R
- CARD(A): numero di valori distinti dell'attributo A

Per ogni indice I

- NLEAF(I): il numero di foglie dell'indice il cui valore e' stimato come segue

$$NLEAF(I) = [CARD(R) * L_{TID}+CARD(A)*L(A)]/ [P*p_f]$$

dove:

- R e A sono rispettivamente la relazione e l'attributo su cui e' allocato l'indice
- L_{TID} e' la lunghezza di un TID
- L(A) e' la lunghezza di un valore dell'attributo
- p_f e' il fattore di riempimento di un nodo dell'indice
- P e' la dimensione delle pagine

OTTIMIZZAZIONE - System R

Statistiche (continua)

Nel System R queste statistiche sono aggiornate in seguito al caricamento delle relazioni o alla creazione di un indice

E' possibile aggiornare i valori usando il comando Update Statistics on Relazione

Le statistiche non vengono aggiornate dopo ogni singola operazione di inserzione, cancellazione, o modifica in quanto si appesantirebbe il costo delle operazioni di aggiornamento

Algoritmo di ottimizzazione
L'algoritmo usa al piu' un indice per relazione

Interrogazione su una singola relazione- passi
- selezione di un predicato di ricerca
- controllo della condizione sui dati

Il predicato di ricerca scelto deve essere un fattore booleano

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

Nel corso di questo passo occorre:
- analizzare la condizione per ricavarne tutti i fattori booleani
- individuare i predicati di ricerca fra i fattori booleani
- valutare il costo di accesso con l'impiego dell'indice, per ogni predicato di ricerca
- scegliere il predicato di ricerca piu' conveniente, cioe' quello con costo minimo

Per eseguire questo passo, la condizione viene rappresentata come un albero in cui

- i nodi terminali sono associati alle condizioni semplici

- gli altri nodi agli operatori AND e OR

- i nodi associati ai predicati sono marcati con l'etichetta R se si tratta di predicati di ricerca

- altrimenti sono marcati con N

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

(a) Si costruisce l'insieme LP dei predicati risolubili applicando il seguente algoritmo ricorsivo a partire dalla radice dell'albero

1) se il nodo e' un predicato di tipo R, allora LP=[predicato]

2) se il nodo e' un predicato di tipo N, allora LP=[]

3) se il nodo e' AND si applica l'algoritmo ricorsivamente ai sottoalberi sinistro e destro, producendo due insiemi che uniti danno l'insieme LP associato al nodo

4) se il nodo e' OR si procede come per il nodo AND

Alla fine del passo (a) alla radice dell'albero e' associato l'insieme dei predicati per i quali esiste un indice utile per l'ottimizzazione della query

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

(b) Si eliminano dall'insieme LP tutti i predicati che hanno un antenato OR, producendo l'insieme LPA dei predicati che sono anche fattori booleani

A questo punto si possono verificare tre casi

- LPA e' vuoto: non esistono indici da usare per limitare il numero di tuple da visitare. Si marca la radice dell'albero con l'etichetta N (non risolto)

- LPA contiene un solo predicato: esiste allora un indice per fare la restrizione. Si marca la radice con S (semi-risolto), oppure con R (risolto) se tutta la condizione coincide con il predicato

- LPA contiene piu' predicati: occorre allora scegliere il predicato piu' conveniente, stimando per ognuno di essi il costo di accesso CA=CI+CD. Una volta selezionato il predicato di ricerca con costo minimo, la radice viene marcata con S.

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

Per stimare il costo di accesso associato ad un predicato di ricerca, si stima il fattore della selettivita' dell condizione (indicato con F(condizione)): cioe' la stima della frazione di tuple che soddisfano la condizione, rispetto al numero totale di tuple presenti nella relazione

Assumendo l'uniformita' di distribuzione dei valori di ogni attributo, il fattore di selettivita' si stima come segue:

- F(A=Valore) = 1/CARD(A)
  (se per qualche motivo CARD(A) non e' noto, il sistema assume F=1/10)
- F(A>Valore) = (max(A) - Valore)/(max(A)-min(A))
  F(A<Valore) = (Valore-min(A))/(max(A)-min(A))
  se si conoscono i valori di max e min dell'attributo e l'attributo e' numerico. Se tali informazioni non sono note, o l'attributo non e' di numerico, si stima F=1/3
  Non c'e' alcuna significativita' in questo numero a parte il fatto che e' maggiore di 1/10 e minore di 1/2 (i progettisti del System R assumono che poche queries che hanno predicati di selezione sono soddisfatte da piu' della meta' delle tuple)

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

- F(A **isin** [$Valore_1$, $Valore_2$,.....,$Valore_N$])=
  = N * F(A=Valore)
  F(A **within** ($Valore_1$, $Valore_2$))=
  = ($Valore_2$ -$Valore_1$)/(max(A)-min(A))
  se si conoscono i valori di max e min dell'attributo sono noti e l'attributo e' numerico. Se tali informazioni non sono note, o l'attributo non e' numerico si considera F=1/4
- F(C_1 AND C_2) = F(C_1) *F(C_2)
  F(C_1 OR C_2) = F(C_1) +F(C_2) - F(C_1) *F(C_2)

Per come e' stato definito, e' chiaro che tanto piu' il fattore di selettivita' e' minore di uno, tanto piu' e' selettivo il predicato a cui si riferisce

Una volta calcolato il fattore di selettivita' del predicato, occorre calcolare il costo di accesso all'indice e il costo di accesso ai dati

Indicato con C_A un predicato di ricerca, e dato F(C_A) fattore di selettivita' di tale predicato, il costo di accesso all'indice I allocato su A, e' dato da:
CI = F(C_A) * NLEAF(I)
(quindi F(C_A) da' anche la stima della frazione di foglie accedute)

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

- La stima del costo di accesso ai dati e' piu' complessa e dipende dalla proprieta' dell'indice di essere clustering oppure no

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

- Stima del costo di accesso ai dati:
  - clustered index
    $CD = F(C_A) * NPAG$

  - non-clustered index
    In questo caso poiche' la lista di riferimenti e' ordinata, non si puo' verificare il caso che per un valore dell'attributo A una pagina venga visitata piu' volte

    Pero' si puo' verificare che una stessa pagina venga visitata piu' volte per valori dell'attributo compresi in un certo intervallo (piu' liste da visitare). Pertanto

    CD = N.Liste di TIDs * N.Pagine da visitare per ogni lista

    N.Pagine da visitare per ogni lista =
    Φ(NTK(A), NPAG, CARD(R))

    (Φ e' la funzione di Yao)

$$\Phi = NPAG * \left[1 - \prod_{i=1}^{NTK(A)} \frac{CARD(R) - (CARD(R)/NPAG) - i + 1}{CARD(R) - i + 1}\right]$$

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

- Stima del costo di accesso ai dati (continua):

CD= $F(C_A)$ * CARD(A) * $\Phi($NTK(A), NPAG, CARD(R))

dove NTK(A) e' il numero medio di TID associati ad un valore dell'attributo; e' ottenuto come:

CARD(R)/CARD(A)

Pertanto, il costo globale di accesso e' dato dalle seguenti espressioni:

- CA = $F(C_A)$* [NLEAF(I) + NPAG(R)]
  se l'indice e' clusterizzato

- CA = $F(C_A)$* [NLEAF(I) +
  CARD(A) * $\Phi($NTK(A), NPAG, CARD(R))]
  se l'indice e' non-clusterizzato

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

Osservazione

- Se si usa un indice per accedere alle tuple della relazione, le tuple vengono reperite e restituite nell'ordine dei valori dell'attributo su cui e' costruito l'indice

- Spesso pero' e' richiesto che le tuple siano in un ordine diverso; per esempio l'utente ha specificato una clausola ORDER BY o GROUP BY

- Pertanto, prima di restituire le tuple all'utente, e' necessario ordinarle se non lo sono gia'

- In questo caso e' opportuno distinguere tra vie di accesso (indici o scansione sequenziale) che producono l'ordinamento richiesto e vie di accesso che non lo producono

- Nel secondo caso e' quindi necessario calcolare anche il costo di ordinamento della relazione risultato

OTTIMIZZAZIONE - System R
Selezione di un predicato di ricerca

<u>Stima del risultato</u>

- La dimensione di tale relazione e' calcolata in base a:
  - la dimensione di una tupla risultato (si ottiene facendo la somma delle dimensioni degli attributi presenti nella *ListaDiAttributi*)

  - il numero delle tuple risultanti CRES
    $CRES = CARD(R) * F_1 * F_2 * \ldots\ldots * F_n$
    dove F_1, F_2, ..., F_n sono le selettivita' di tutti i congiunti della interrogazione che si riesce a stimare

- In base alla stima della dimensione della relazione intermedia, e' possibile stimare il numero di pagine per contenere il risultato e quindi il costo di ordinamento

- Pertanto un indice clusterizzato e' di solito da preferire ad uno non clusterizzato, a meno che il predicato risolubile con l'indice non clusterizzato e' molto selettivo oppure le tuple sono richieste ordinate in base ai valori dell'attributo su cui e' allocato l'indice non clusterizzato

OTTIMIZZAZIONE - System R
Controllo della condizione sui dati

- Una volta calcolati i costi di ciascun predicato e selezionato quello con costo minimo, si esegue il passo successivo nell'esecuzione dell'interrogazione

- Il procedimento di controllo dipende da come e' marcata la radice dell'albero

- Se l'etichetta e' N, allora si procede con la scansione sequenziale della relazione e si estraggono le tuple che verificano la condizione

- Se l'etichetta e' S, si usa l'indice associato al predicato (scelto al passo precedente) solo se il suo costo e' inferiore a quello della scansione sequenziale, stimato pari a NPAG (numero delle pagine cha hanno tuple della relazione); in caso contrario si procede con la scansione sequenziale

- Se l'etichetta e' R, si procede come al caso precedente, la differenza e' che, se si usa l'indice, i riferimenti sono solo quelli delle tuple che soddisfano l'interrogazione

OTTIMIZZAZIONE - System R
Controllo della condizione sui dati

<u>Conclusione</u>

Per quanto riguarda il costo del metodo si hanno due stime:

- CA=NPAG se l'etichetta della radice e' N
- se l'etichetta della radice e' S o R, supponendo di scartare la scansione sequenziale, e trascurando il costo di ordinamento esistono due sottocasi:

  - $CA = F(C_A)^*$ [NLEAF(I) + NPAG(R)]
    se l'indice selezionato e' clusterizzato

  - $CA = F(C_A)^*$ [NLEAF(I) + CARD(A) * Φ(NTK(A), NPAG, CARD(R))]
    se l'indice selezionato e' non-clusterizzato

OTTIMIZZAZIONE - System R
Esempio

Si consideri la relazione
Impiegato(Codice, Nome, Qualifica, Stipendio, Progetto, Citta')

supponendo che:

- la relazione sia ordinata su Stipendio, e sia costituita da 2.000 tuple, contenute in 100 pagine, ciascuna delle quali contiene 20 tuple
  CARD(Impiegati)=2000
  NPAG=100

- esista un indice I clusterizzato su Stipendio
  CARD(Stipendio)=200
  NLEAF(I') = 9
  Min(Stipendio) = 1.200.000
  Max(Stipendio)=2.700.000

- esista un indice I' non clusterizzato su Qualifica
  CARD(Qualifica)=40
  NLEAF(I') = 7

OTTIMIZZAZIONE - System R
Esempio

Si vuole selezionare il Nome e lo Stipendio dei programmatori che guadagnano tra 1.500.000 e 2.000.000, e cha lavorano al progetto "Banche" o "Comuni"

- Questa interrogazione e' formulata come segue
  SELECT Nome, Stipendio
  FROM Impiegati
  WHERE Qualifica = "Programmatore" AND
  Stipendio within (1.500.000, 2.000.000) AND
  Progetto isin ["Banche"; "Comuni"]

Albero della condizione

AND LP=[Qualifica, Stipendio]

R Qualifica="Programmatore"

AND LP=[Stipendio]

R Stipendio within (1.500.000, 2.000.000)

N Progetto isin["Banche", "Comuni"]

OTTIMIZZAZIONE - System R
Esempio

- Si ha che LPA=[Qualifica, Stipendio]

- Strategie alternative
  a) scansione sequenziale
  b) accesso con indice I (su Stipendio)
  c) accesso con indice I' (su Qualifica)

- I fattori di selettivita' sono:
  F(Qualifica = "Programmatore") = 1/40=0.025
  F(Stipendio within (1.500.000, 2.000.000)) = 0.33

  NTK(Qualifica) = CARD(Impiegati)/
  /CARD(Qualifica) = 2.000/40=50

- Quindi i costi per le tre alternative sono:
  a) $CA_{sequenziale}= 100$
  b) $CA_{Qualifica}= 0.025*[7 + 40 *\Phi(50,100, 2000)]=40$
  c) $CA_{Stipendio}= 0.33 *(9 + 100) = 36$

- Viene quindi preferito l'indice su Stipendio

- In questo caso si nota come un predicato poco selettivo possa dar luogo ad un costo di scansione inferiore attraverso indice inferiore a quello prodotto usando non indice non clusterizzato

OTTIMIZZAZIONE - System R
Esempio

- Supponiamo di avere anche un indice I'' non clusterizzato su Progetto
  CARD(Progetto) = 300 e NLEAF(I'') = 16
  si ha
  F(Progetto isin ["Banche", "Comuni"])=2/300=0.0066

  NTK(Progetto)=2000/300 = 7 da cui
  $CA_{Progetto}$= 0.0066 * [16 + 300 *Φ(7,100, 2000)] =
  1 + 14=15

- In questo caso l'indice su Progetto e' da preferire a quello su Stipendio

OTTIMIZZAZIONE - System R
Joins

- Il System R usa come metodi di join sia il nested-loop (iterazione) che il merge-join

- Inoltre vengono considerate tutte le possibili permutazioni quando si deve eseguire il join di piu' di due relazioni

- Per restringere lo spazio della ricerca non vengono considerate permutazioni che implicano dei prodotti cartesiani: in altra parole si cerca di eseguirli il piu' tardi possibile

  Per esempio:
  date R1(a,b) R2 (d,e,f), R3(g,h) relazioni e una query della forma
  select R1.a, R2.f, R3.h **from** R1, R2, R3 **where**
  R1.b=R2.d AND R2.e=R3.g

  esiste un join tra R1 e R2 e un join tra R2 e R3
  il primo join e' su un attributo di R2 diverso dall'attributo su cui e' definito il secondo join

  Le permutazioni che non vengono considerate sono:
  (R1-R3)-R2 (R3-R1)-R2

OTTIMIZZAZIONE - System R
Joins

- Per determinare la strategia ottima per il join di n relazioni, viene generato un albero delle possibili strategie

- L'approccio seguito consiste nel determinare una soluzione ottima per ogni sottoinsieme dei join della query, e poi determinando il modo migliore per eseguire il join della relazione composita ottenuta con una ulteriore relazione

- Questo approccio parte dal considerare i join di due relazioni ed ogni passo incrementa il numero di relazioni

- Una soluzione consiste di:
  - una lista ordinata di relazioni su cui si esegue il join
  - il metodo di join usato
    (nested loop, merge join)
  - un piano indicante come ogni relazione deve essere acceduta
    (indice, scansione sequenziale)
  - indicazione sulla necessita' di ordinamento delle relazioni prima dell'esecuzione del join

OTTIMIZZAZIONE - System R
Joins

- Un aspetto importante riguarda se le relazioni devono essere ordinate. Si definiscono ordinamenti *interessanti*, gli ordinamenti eseguiti su attributi che compaiono in clausole di Group by e di Order by, e sugli attributi di join

- Infatti in alcuni casi, puo' essere preferibile scegliere un indice non clustered rispetto ad un indice clustered, se il primo ritrova le tuple secondo un ordinamento interessante

- L'albero di ricerca e' costruito tramite iterazione sul numero di relazioni "joined"

- Al primo passo, si determina il modo piu' efficiente per accedere ogni singola relazione per ogni ordinamento interessante e per il caso non ordinato

- Al secondo passo, si determina il modo migliore per eseguire il join di ogni singola relazione con ogni altra singola relazione (alcune combinazioni non sono valutate in base all'euristica di ritardare l'esecuzione del prodotto cartesiano)

OTTIMIZZAZIONE - System R
Joins

- Il secondo passo produce soluzioni per eseguire i joins di coppie di relazioni

- Al terzo passo si determina il modo migliore per eseguire i joins di tre relazioni

- Il terzo passo viene eseguito considerando ogni relazione composita, ottenuta al passo precedente, e determinando il modo migliore per collegarla con ogni altra relazione (soggetta alla euristica)

- Dopo che tutte le soluzioni complete sono determinate (il join di tutte le relazioni e' eseguito), l'ottimizzatore sceglie la soluzione meno costosa che restituisce le tuple nell'ordine richiesto (se e' stato specificato un ordine)

- Notare che se esiste una soluzione che da' l'ordinamento richiesto, non deve essere eseguito alcuno ordinamento extra per le clausole di Order by e Group by, a meno che la soluzione ordinata sia piu' costosa della soluzione piu' economica non ordinata con il costo addizionale dell'esecuzione dell'ordinamento

OTTIMIZZAZIONE - System R
Joins

- Il numero di soluzioni da memorizzare e' al piu' 2**n (il numero di possibili sottoinsiemi di n relazioni) moltiplicato il numero di ordinamenti interessanti

- Il tempo necessario per generare l'albero e' proporzionale a tale numero

- In realta' questo numero e' spesso ridotto usando l'euristica sul prodotto cartesiano

- L'esperienza indica che il tempo di CPU (su 370/168) necessario per eseguire l'ottimizzazione e' dell'ordine di poche decine di secondi

- Formule di stima della dimensione delle relazioni intermedie
  Il System R usa delle formule piu' semplici rispetto a quelle viste precedentemente

OTTIMIZZAZIONE - System R
Joins - esempio

OTTIMIZZAZIONE - System R
Joins - esempio

Indici:
relazione EMP: indice su DNO, indice su JOB
relazione DEPT: indice su DNO
relazione JOB: indice su JOB

Passo 1

Si determina il modo piu' efficiente per eseguire l'accesso ad ogni singola relazione

Relazione EMP - tre possibili strategie
a) indice su DNO
b) indice su JOB
c) scansione sequenziale
Gli ordinamenti interessanti sono DNO, JOB

Supponiamo che l'indice su JOB sia la strategia di accesso piu' efficiente

Questo permette di eliminare la scansione sequenziale; la strategia (a) viene comunque mantenuta perche' rappresenta un ordinamento interessante

OTTIMIZZAZIONE - System R
Joins - esempio

Passo 1

Relazione DEPT- due possibili strategie
a) indice su DNO
b) scansione sequenziale

Supponiamo che l'accesso basato sull'indice su DNO sia la strategia piu' efficiente

Questo permette di eliminare la scansione sequenziale

Relazione JOB - due possibili strategie
a) indice su JOB
b) scansione sequenziale

Supponiamo che la scansione sequenziale sia piu' efficiente

In questo caso la strategia (a) viene mantenuta in quanto da' un ordinamento efficiente

OTTIMIZZAZIONE - System R
Joins - esempio

Sommario del passo 1

OTTIMIZZAZIONE - System R
Joins - esempio

Risultati del passo 1

(nella figura C<set of relations> (path) e' usato per rappresentare il costo di eseguire la scansione della relazione in base al cammino di accesso indicato da path;

Ni indica le cardinalita' delle relazioni intermedie)

OTTIMIZZAZIONE - System R
Joins - esempio

<u>Passo 2</u>

Si determinano soluzioni per tutte le possibili coppie di relazioni

Si esegue il join di una seconda relazione con ogni singola relazione nell'albero di ricerca ottenuto al passo 1

Si considerano prima tutte le strategie basate sul nested loop

In questo esempio si assume che e' piu' economico eseguire il join EMP-JOB usando l'indice su JOB

(infatti data una tupla di EMP possiamo trovare subito la tupla di JOB che ha lo stesso valore per l'attributo JOB)

OTTIMIZZAZIONE - System R
Joins - esempio

<u>Passo 2</u>

Albero di ricerca strategie nested-loop

OTTIMIZZAZIONE - System R
Joins - esempio

<u>Passo 2</u>

- Si determinano tutte le soluzioni basate sul merge-join

- Poiche' esiste una cammino di accesso che ordina EMP in base al DNO e DEPT in base al DNO e' possibile eseguire un merge join tra queste relazioni, senza eseguire alcun ordinamento

- Notare che un'altra possibilita' e' di accedere EMP in base al indice sull'attributo JOB, eseguire l'ordinamento su DNO, e quindi eseguire il merge

- Per eseguire il merge di JOB con EMP, consideriamo solo l'indice sulla colonna JOB di EMP poiche' e' la strategia di accesso piu' efficiente, indipendentemente dall'ordine

- Usando l'indice su JOB per la relazione JOB e' possibile eseguire il merge senza alcun ordinamento preliminare

- Un'alternativa e' di eseguire una scansione sequenziale di JOB e poi eseguirne l'ordinamento (quindi non si usa l'indice sulla relazione JOB)

OTTIMIZZAZIONE - System R
Joins - esempio

Passo 2

Albero di ricerca strategie merge-join

OTTIMIZZAZIONE - System R
Joins - esempio

<u>Passo 2</u>

Una volta calcolati i costi di tutte le strategie sia nel caso nested-loop che nel caso merge-join, viene fatto un confronto per determinare per ogni coppia di relazioni la *soluzione piu' economica equivalente* (due soluzioni sono equivalenti se hanno le stesse tabelle e l'ordine delle tuple nel risultato e' lo stesso)

Risultato del pruning

OTTIMIZZAZIONE - System R
Joins - esempio

<u>Passo 3</u>

Si determina la soluzione per eseguire il join di tre relazioni

Le soluzioni possibili sono determinate eseguendo il join di una terza relazione con le relazioni intermedie (ottenute come join di due relazioni alla fine del passo 2)

Come al passo 2, si determinano prima le strategie nested-loop e poi quelle merge-join

Supponiamo che C5U e' un costo minore di C5D e C5J (se non e' cosi' la strategia basata sulla scansione sequenziale puo' essere eliminata)

Supponiamo inoltre che per eseguire il join di JOB con il join ottenuto da EMP e DEPT sia preferibile usare l'indice su JOB per la relazione JOB

OTTIMIZZAZIONE - System R
Joins - esempio

<u>Passo 3</u>

Albero di ricerca strategie nested-loop

OTTIMIZZAZIONE - System R
Joins - esempio

<u>Passo 3</u>

Albero di ricerca strategie merge-join

Dopo che tutte le soluzioni sono state valutate, si sceglie la piu' economica