ARCHITETTURA DI UN DBMS

- Finora abbiamo visto modelli di DBMS ad alto livello (livello *logico)*: e' il livello corretto per gli *utenti* del DB

- 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

10	Edilizia Civile	1100	D1	7977		
7782	Neri	ingegnere	01-Giu-81	2,450.00	200.00	10
7839	Dare	ingegnere	17-Nov-81	2,600.00	300.00	10
7934	Milli	ingegnere	23-Gen-82	1,300.00	150.00	10
7977	Verdi	dirigente	10-Dic-80	3,000.00	?	10
20	Ricerche	2200	D1	7566		
7369	Rossi	ingegnere	17-Dic-80	1,600.00	500.00	20
7566	Rosi	dirigente	02-Apr-81	2,975.00	?	20
7788	Scotti	segretaria	09-Nov-81	800.00	?	20
7876	Adami	ingegnere	23-Set-81	1,100.00	500.00	20
7902	Fordi	segretaria	03-Dic-81	1,000.00	?	20
30	Edilizia Stradale	5100	D2	7698		
7499	Andrei	tecnico	20-Feb-81	800.00	?	30
7521	Bianchi	tecnico	20-Feb-81	800.00	100.00	30
7654	Martini	segretaria	28-Set-81	800.00	?	30
7698	Blacchi	dirigente	01-Mag-81	2,850.00	?	30
7844	Turni	tecnico	08-Set-81	1,500.00	?	30
7900	Gianni	ingegnere	03-Dic-81	1,950.00	?	30

– –

DEFINIZIONE DI CLUSTER E INDICI IN SQL

- La maggior parte dei DBMS relazionali fornisce varie primitive che permettono al progettista della base di dati di definire la configurazione fisica dei dati

- Queste primitive sono rese disponibili all'utente come comandi del linguaggio SQL

- I comandi piu' importanti sono il comando per la creazione di indici (**CREATE INDEX**), su una o piu' colonne di una relazione, e il comando per la creazione di cluster (**CREATE CLUSTER**)

- Un cluster permette di memorizzare fisicamente contigue le tuple di una o piu'relazioni che hanno lo stesso valore per una o piu'colonne, dette colonne del cluster

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 e’ organizzato in pagine, che hanno la stessa dimensione dei blocchi
- Quando una pagina della MS e’ presente nel buffer, il DBMS puo’ effettuare le sue operazioni di lettura e scrittura direttamente su di essa
- I tempi di accesso alla MS sono dell’ordine di millesimi di secondo mentre quelli di accesso alla MM sono dell’ordine di milionesimi di secondo
- Accedere alle pagine nel buffer invece che alle corrispondenti pagine su disco influenza notevolmente le prestazioni

GESTIONE DEL BUFFER

- 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 non sempre sono le piu′ adatte per i DBMS

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

  (iii) per motivi legati alla gestione del recovery in alcuni casi e' necessario forzare un blocco su disco (anche se il suo spazio non e' stato reclamato)

GESTIONE DEL BUFFER

- Un DBMS e' in grado di predire meglio di un SO il tipo dei futuri riferimenti

Esempio: operazione di join

`Impiegati` |x| **`Dipartimenti`**

(assumendo che le due relazioni siano in due file diversi)

- relazione **`Impiegati`**
  - una volta che una tupla della relazione e' stata usata non e' piu' necessaria
  - non appena tutte le tuple di un blocco sono state esaminate il blocco non serve piu' (strategia *toss-immediate*)

- relazione **`Dipartimenti`**
  - il blocco piu' recentemente acceduto sara' riferito di nuovo solo dopo che tutti gli altri blocchi saranno stati esaminati
  - la strategia migliore per il file **`Dipartimenti`** e' di rimuovere l'ultimo blocco esaminato (strategia *most recently used* - MRU)
  - e' pero' necessario eseguire il *pin* del blocco correntemente esaminato fino a che si siano esaminate tutte le tuple; quindi si puo' rendere il blocco unpinned

ESECUZIONE DI INTERROGAZIONI

- Abbiamo visto finora come organizzare i dati in un DB

- Le decisioni sulle strutture da allocare sono determinate durante la progettazione fisica del 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 interrogazioni complesse esistono piu' strategie possibili

- Anche se il costo di determinare la strategia ottima puo' essere alto, il vantaggio in termini di efficienza che se ne ricava e′ tale che in genere conviene eseguire l'ottimizzazione

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**
  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)

PERCHE' OTTIMIZZARE LE INTERROGAZIONI?

- Consideriamo le relazioni

```
Studenti(MatrS,Nome,Ind,AltreInfo)
Esami(Corso,MatrS,Voto,Data)
```

- Supponiamo di voler trovare il nome degli studenti e la data degli esami per gli studenti che hanno sostenuto BD con 30

```
SELECT Nome,Data
FROM Studenti NATURAL JOIN Esami
WHERE Corso ='BD' AND Voto =30
```

- Consideriamo un database con 2.000 studenti e 20.000 esami, di cui 500 di BD e di questi solo 50 con 30 (consideriamo solo la scansione sequenziale delle relazioni)

- Se si fa il prodotto cartesiano delle due relazioni, si ottiene una relazione temporanea con 40.000.000 tuple, da queste si estraggono poi le 50 tuple desiderate (costo proporzionale a 120.000.000 accessi)

- Se si selezionano i 50 esami di BD con 30 e poi si fa il join di questa relazione temporanea con `Studenti` si ha un costo proporzionale a 120.050

TRASFORMAZIONI ALGEBRICHE

Basate su un'insieme di leggi di equivalenza tra espressioni dell'algebra, e su alcune regole per applicarle quali:

- Eseguire le operazioni di selezione il prima possibile

- Trasformare espressioni della forma $\sigma_{P1\ AND\ P2}(e)$ in
  $\sigma_{P1}(\sigma_{P2}(e))$

- In caso di join tra tre o piu' relazioni eseguire per primo il join che da' luogo alla relazione temporanea di dimensione minore

- Evitare i join che diventano prodotti cartesiani

- Anticipare le proiezioni in modo da eliminare gli attributi inutili dalle relazioni temporanee

  gli unici attributi da non eliminare sono quelli che
  - appaiono nel risultato della query
  - sono necessari in operazioni successive

SELEZIONE DELLA STRATEGIA

Profili delle relazioni

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

- Per selezionare la strategia si utilizzano stime del costo di esecuzione, basate su **profili** delle relazioni

- Quindi normalmente i DBMS 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 byte (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
  4. SA dimensione in byte di ciascun attributo A

- Da tali profili si ottengono i profili delle relazioni temporanee ottenute applicando operazioni algebriche alle relazioni di base
  (ad es. il prodotto cartesiano r x s contiene $n_r * n_s$ tuple di dimensione $S_r + S_s$)

SELEZIONE DELLA STRATEGIA

- Il profilo delle relazioni che coinvolgono selezioni e join viene stimato sotto l'ipotesi che ogni valore appaia con la stessa probabilita'

- Si stima cioe' che $\sigma_{A=a}(r)$ selezioni 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 Impiegati e l'attributo mansione, ci si puo' aspettare che ci siano piu' tecnici che dirigenti e quindi alcune mansioni appariranno con maggiore probabilita' di altre

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

- Un'altra assunzione fondamentale (anche se non sempre realistica) e' la non correlazione tra valori di attributi diversi (es. mansione-stipendio)

SELEZIONE DELLA STRATEGIA

- Nei sistemi si usano in genere modelli dei costi piu’ complicati, che tengono conto delle strategie di memorizzazione delle relazioni e della possibilita’ di usare indici nelle interrogazioni

- Ad esempio, in aggiunta alle statistiche viste si considerano

  - per ogni relazione r: NPAG(r), numero delle pagine di r

  - per ogni indice I: NLEAF(I), numero di foglie dell'indice I

- Le statistiche sono aggiornate in seguito al caricamento delle relazioni o alla creazione di un indice (ma l’utente puo’ richiedere esplicitamente il loro aggiornamento)

<u>Nota importante: granularita' dell'ottimizzazione</u>
L'ottimizzazione e' eseguita separatamente per ogni interrogazione pertanto e' bene formulare interrogazioni complesse e non scomporle in tante interrogazioni piu' semplici perche' cio' fa perdere i vantaggi dell'ottimizzazione

SELEZIONE DELLA STRATEGIA

Interrogazione su una singola relazione

Consideriamo il caso di voler utilizzare al piu' un indice per relazione (spesso e' cosi')

- *predicato di ricerca* (o *predicato risolubile*) e' 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 (es. C+D=1.000 o D <> 20 non sono predicati di ricerca)

- *fattore booleano* e' un predicato, che se falso, rende falsa tutta l'interrogazione; tutte le tuple del risultato quindi verificano i fattori booleani

Passi:

- per ogni predicato di ricerca fattore booleano si valuta il costo di accesso con l'impiego dell'indice, si sceglie quello con costo minimo
- si controlla la condizione sui dati

Nota: a volte puo' risultare piu' efficiente la scansione sequenziale

SELEZIONE DELLA STRATEGIA
Ordinamento delle tuple

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

- La query puo’ richiedere che le tuple siano in un dato ordine (es. clausole ORDER BY, GROUP BY)
  Pertanto, prima di restituire le tuple all'utente, e' necessario ordinarle se non lo sono gia'

- In questo caso si devono 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' necessario calcolare anche il costo di ordinamento della relazione risultato

SELEZIONE DELLA STRATEGIA
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 (*clusterizzati* vs *non-clusterizzati*)
  - il costo di costruire un indice temporaneo allo scopo di eseguire il join

- Diverse strategie possibili (iterazione semplice, iterazione orientata ai blocchi, merge-join, uso di indici, three-way join)

TRANSAZIONI

- Per mantenere le informazioni consistenti e’ necessario controllare opportunamente le sequenze di accessi e aggiornamenti ai dati

- Tali sequenze sono dette **transazioni**

- Ogni transazione e’eseguita o completamente (cioe’ effettua il *commit*), oppure per nulla (cioe’ effettua l'*abort*) se si verifica un qualche errore (hardware o software) durante l'esecuzione

- Necessita’ di garantire che le transazioni eseguite *concorrentemente* si comportino come se fossero state eseguite in sequenza

- Necessita’ di tecniche per r*ipristinare* uno stato corretto della base di dati a fronte di malfunzionamenti di sistema.

TRANSAZIONI: CONCETTI DI BASE

- Gli utenti interagiscono con la base di dati attraverso programmi applicativi ai quali viene dato il nome di transazioni

- Una transazione e′ un insieme parzialmente ordinato di operazioni di lettura e scrittura; essa costituisce l'effetto dell'esecuzione di programmi che effettuano le funzioni desiderate dagli utenti

- L'insieme di operazioni che costituiscono una transazione deve soddisfare alcune proprieta′, note come proprieta′ **ACID**:

    - Atomicita′
    - Consistenza
    - Isolamento
    - Durabilita′

TRANSAZIONI

Atomicita’

- e’ detta anche proprieta’ tutto-o-niente

- tutte le operazioni di una transazione devono essere trattate come una singola unita’: o vengono eseguite tutte, oppure non ne viene eseguita alcuna

- l'atomicita’ delle transazioni e’ assicurata dal sottosistema di *ripristino* (*recovery*)

Consistenza

- una transazione deve agire sulla base di dati in modo corretto

- se viene eseguita su una base di dati in assenza di altre transazioni, la transazione trasforma la base di dati da uno stato consistente (cioe’ che riflette lo stato reale del mondo che la base di dati deve modellare) ad un altro stato ancora consistente

- l'esecuzione di un insieme di transazioni corrette e concorrenti deve a sua volta mantenere consistente la base di dati

- il sottosistema di *controllo della concorrenza* (*concurrency control*) sincronizzale transazioni concorrenti in modo da assicurare esecuzioni concorrenti libere da interferenze

TRANSAZIONI

Isolamento

- ogni transazione deve sempre osservare una base di dati consistente, cioe', non puo' leggere risultati intermedi di altre transazioni

- la proprieta' di isolamento e' assicurata dal sottosistema di controllo della concorrenza che isola gli effetti di una transazione fino alla sua terminazione

Durabilita' (persistenza)

- i risultati di una transazione terminata con successo devono essere resi permanenti nella base di dati nonostante possibili malfunzionamenti del sistema

- la persistenza e' assicurata dal sottosistema di ripristino

- tale sottosistema uo' inoltre fornire misure addizionali, quali back-up su supporti diversi e journaling elle transazioni, per garantire la durabilita' anche a fronte di guasti ai dispositivi i memorizzazione

TRANSAZIONI: PROPRIETA’ ACID

Le proprieta’ ACID vengono assicurate utilizzando due insiemi distinti di algoritmi o protocolli, che assicurano:

- **l'atomicita’ dell'esecuzione**
  - mantenere la consistenza globale della base di dati e quindi assicurare la proprieta’ di consistenza delle transazioni (anche concorrenti)
  - protocolli di *controllo della concorrenza*
- **l'atomicita’ del fallimento**
  - assicura l'atomicita’, l'isolamento e la persistenza
  - protocolli di *ripristino*

TRANSAZIONI FLAT

- Facciamo riferimento al modello di transazioni piu' semplice (*transazioni flat*), che prevede un solo livello di controllo a cui appartengono tutte le transazioni eseguite (e' il modello usato nei DBMS commerciali)

- Tutte le istruzioni eseguite devono essere contenute tra le istruzioni **`BeginWork`** e **`CommitWork`**

    - l'istruzione **`BeginWork`** dichiara l'inizio di una transazione flat

    - l'istruzione **`CommitWork`** e' invocata per indicare che il sistema ha raggiunto un nuovo stato consistente

- La transazione puo' terminare la propria esecuzione con successo (*commit*) e rendere definitivi i cambiamenti prodotti sulla base di dati dalle istruzioni eseguite tra **`BeginWork`** e **`CommitWork`**, oppure sara' disfatta (cioe' i suoi effetti saranno annullati) e tutti gli aggiornamenti eseguiti andranno persi (*abort*)

- In questo caso, si dice che viene eseguito il *rollback* della transazione

TRANSAZIONI FLAT

- i vari DBMS forniscono specifiche istruzioni SQL per supportare l'uso di transazioni flat

- tali istruzioni sono invocate all'interno di programmi applicativi e consentono al programmatore di identificare l'inizio e la fine di una transazione

- l'istruzione

  ```
  exec sql connect :user_name identified
  by :user_pws
  ```

  (cioe’ l'istruzione di connessione) avvia la transazione, cioe’ implementa la primitiva **`BeginWork`**

- l'istruzione **`exec sql commit work`** esegue il commit di una transazione

- l'istruzione **`exec sql rollback`** esegue il rollback di una transazione

CONTROLLO DELLA CONCORRENZA

Scopo:
garantire l'integrita'della base di dati in presenza di accessi concorrenti da parte di piu' utenti

necessita' di sincronizzare le transazioni eseguite concorrentemente

Esempio di problema:

- base di dati che organizza le informazioni sui conti dei clienti di una banca

- il sig. Rossi e' titolare di due conti: un conto corrente (intestato anche alla sig.ra Rossi) e un libretto di risparmio, i cui saldi sono Lit. 100.000 e Lit. 1.000.000

- con la transazione T1 il sig. Rossi trasferisce Lit. 150.000 dal libretto di risparmio al conto corrente

- contemporaneamente con la transazione T2 la sig.ra Rossi deposita Lit.500.000 sul conto corrente

CONTROLLO DELLA CONCORRENZA

Esempio

T1	T2
Read(Lr)	
Lr = Lr – 150000	
	Read(Cc)
Write(Lr)	
	Cc = Cc + 500000
Read(Cc)	
	Write(Cc)
	Commit
Cc = Cc + 150000	
Write(Cc)	
Commit	

La somma depositata da T2 e’ persa (*lost update*)
e non si ottiene l'effetto voluto sulla base di dati

- l'esecuzione concorrente di piu’ transazioni genera un'alternanza di computazioni da parte delle varie transazioni, detta *interleaving*

- l'interleaving tra le transazioni T1 e T2 nell'esempio produce uno stato della base di dati scorretto; si sarebbe ottenuto uno stato corretto se ciascuna transazione fosse stata eseguita da sola o se le due transazioni fossero state eseguite l'una dopo l'altra, consecutivamente

LOCK

- **idea base:** ritardare l'esecuzione di operazioni in conflitto imponendo che le transazioni pongano dei blocchi (*lock*) sui dati per poter effettuare operazioni di lettura e scrittura

- due operazioni si dicono in conflitto se operano sullo stesso dato e almeno una delle due e’ un'operazione di scrittura

- una transazione puo’ accedere ad un dato solo se ha un lock su quel dato

- due tipi di lock:

  - **shared** se una transazione T ha ottenuto un lock con modalita’ shared (condivisa) sul dato Q, allora T puo’ leggere questo dato ma non puo’ modificarlo

  - **exclusive** se una transazione T ha ottenuto un lock con modalita’ exclusive (esclusiva) sul dato Q, allora T puo’ sia leggere che modificare Q

- richiesta di lock (esclusivo, condiviso) sul dato Q: **`Ls(Q)`**, **`Lx(Q)`**

- rilascio di lock sul dato Q: **`Un(Q)`**

LOCK

matrice di compatibilita’ dei lock:

	Shared	**Exclusive**
Shared	S	N
Exclusive	N	N

se Ti richiede un lock di modalita’ A sul dato Q su cui Tj (Ti ≠Tj) possiede un lock di modalita’ B, a Ti puo’ essere concesso un lock su Q se il lock di modalita’ A e’ **compatibile** con il lock di modalita’ B.

ogni transazione deve effettuare richieste di lock e unlock in due fasi distinte:

- *fase di acquisizione* in cui una transazione puo’ ottenere dei lock, ma non puo’rilasciarne

- *fase di rilascio* in cui una transazione puo’ rilasciare dei lock, ma non puo’ ottenerne di nuovi

inizialmente, una transazione si trova nella fase di acquisizione e acquisisce i lock necessari

non appena la transazione rilascia un lock, inizia
la fase di rilascio e non puo’ piu’ essere richiesto alcun lock

PROTOCOLLO TWO-PHASE LOCKING

- il protocollo di two-phase locking visto assicura che le esecuzioni concorrenti di transazioni corrette non producano stati inconsistenti, ma puo’ provocare **deadlock**

- per aumentare il grado di parallelismo viene consentito l'uso di **conversioni**:

  uno shared lock puo’ essere promosso a exclusive lock e un exclusive lock puo’ essere regredito ad uno shared lock

- la promozione di lock puo’ avvenire solo durante la fase di acquisizione, la regressione solo durante la fase di rilascio

- esistono anche altri meccanismi di controllo della concorrenza, ma quello basato sui lock e’ il piu’ utilizzato

GESTIONE DEL RIPRISTINO

Tre principali tipi di malfunzionamenti:

- **malfunzionamenti del disco:** le informazioni residenti su disco vengono perse (rottura della testina, errori durante il trasferimento dei dati)

- **malfunzionamenti di alimentazione:** le informazioni memorizzate in memoria centrale e nei registri vengono perse

- **errori nel software:** si possono generare risultati scorretti e il sistema potrebbe essere in uno stato inconsistente (errori logici ed errori di sistema)

Il **sottosistema di recovery** (ripristino) deve identificare i malfunzionamenti e ripristinare la base di dati allo stato (consistente) precedente il malfunzionamento

CLASSIFICAZIONE DELLE MEMORIE

- **memoria volatile** le informazioni contenute vengono perse in caso di cadute di sistema

  esempi: memoria principale e cache

- **memoria non volatile** le informazioni contenute sopravvivono a cadute di sistema, possono pero' essere perse a causa di altri malfunzionamenti

  esempi: disco e nastri magnetici

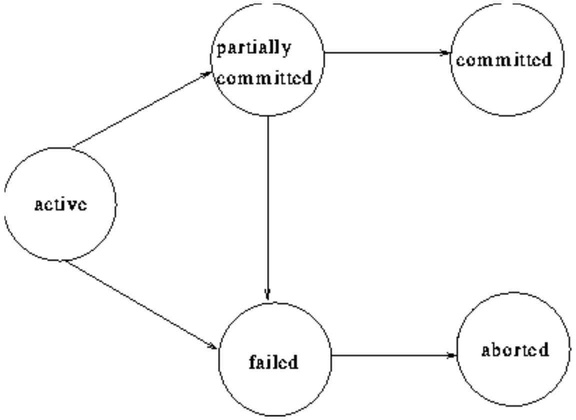
- **memoria stabile** le informazioni contenute non possono essere perse (astrazione teorica)

  se ne implementano approssimazioni, duplicando le informazioni in diverse memorie non volatili con probabilita' di fallimento indipendenti

MODELLO ASTRATTO PER L'ESECUZIONE DI TRANSAZIONI

una transazione e′ sempre in uno dei seguenti stati:

- **active:** lo stato iniziale
- **partially committed:** lo stato raggiunto dopo che e′ stata eseguita l'ultima istruzione
- **failed:** lo stato raggiunto dopo aver determinato che l'esecuzione non puo′ procedere normalmente
- **aborted:** lo stato raggiunto dopo che la transazione ha subito un rollback e la base di dati e′ stata ripristinata allo stato precedente l'inizio della transazione
- **committed:** dopo il completamento con successo



MODELLO ASTRATTO PER L'ESECUZIONE DI TRANSAZIONI

e’ importante che una transazione non effettui *scritture esterne osservabili* (cioe’ scritture che non possono essere “cancellate”, ad es. su terminale o stampante) prima di entrare nello stato di commit

dopo il rollback di una transazione, il sistema ha due possibilita’:

rieseguire la transazione ha senso solo se la transazione e’ stata abortita a seguito di errori software o hardware non dipendenti dalla logica interna della transazione

eliminare la transazione se si verificano degli errori interni che possono essere corretti solo riscrivendo il programma applicativo

GESTIONE DEL RIPRISTINO: ESEMPIO

T transazione che trasferisce Lit. 100000 dal conto A al conto B

A = Lit. 1000000, B = Lit. 15000000

dopo la modifica di A e prima della modifica di B, si verifica una caduta di sistema e i contenuti della memoria vengono persi

- se si riesegue T: stato (inconsistente) in cui A = Lit. 800000 e B = Lit. 15.100000

- se non si riesegue T : stato corrente (inconsistente) in cui A = Lit. 900000 e B = Lit. 15000000

in entrambi i casi lo stato risultante e' inconsistente

il problema e' causato dal fatto di avere modificato la base di dati prima di avere la certezza che la transazione abbia terminato con successo

MECCANISMI DI RECOVERY CON LOG

- durante l'esecuzione di una transazione tutte le operazioni di scrittura sono registrate in un particolare file gestito dal sistema, detto **file di log**

- concettualmente, il log puo' essere pensato come un file sequenziale, nell'implementazione effettiva possono essere usati piu' file fisici

- ad ogni record inserito nel log viene attribuito un identificatore unico (LSN, log sequence number o numero di sequenza di log) che in genere e' l'indirizzo logico del record

- la versione non volatile del log e' memorizzata su memoria stabile

- i record di log sono memorizzati in un primo momento solo nei buffer dei file di log nella memoria volatile

- prima di cominciare il commit, tutti i record di log fino ad un certo punto, identificato da un LSN, vengono scritti su memoria stabile (il log viene *forzato* fino a quel LSN)

- tramite il log, il sistema puo' gestire qualsiasi malfunzionamento che non implichi la perdita di informazioni contenute in memoria non volatile

LOG INCREMENTALE CON MODIFICHE IMMEDIATE

- gli aggiornamenti sono effettuati sulla base di dati e un log incrementale tiene traccia di tutti i cambiamenti; tali informazioni sono usate per riportare la base di dati allo stato consistente precedente in caso di malfunzionamento

durante l'esecuzione di una transazione Ti:

- prima che Ti cominci la propria esecuzione viene scritto nel log il record <Ti start>

- ogni operazione di scrittura **write[x]** e’ preceduta dalla scrittura un record di log della forma:
  <Ti, x, vecchio valore, nuovo valore >

- quando Ti entra in stato partially committed, viene scritto nel log il record <Ti commit>

non si puo’ aggiornare effettivamente la base di dati prima che il corrispondente record di log sia scritto in memoria stabile

LOG INCREMENTALE CON MODIFICHE IMMEDIATE

due procedure di ripristino:

- **undo(Ti):** ripristina i valori di tutti i dati aggiornati da Ti ai vecchi valori

- **redo(Ti):** pone il valore di tutti i dati aggiornati da Ti ai nuovi valori

entrambe devono essere idempotenti

a seguito di un malfunzionamento:

- Ti e' disfatta se il log contiene <Ti start>, ma non <Ti commit>

- Ti e' rieseguita se il log contiene sia <Ti start> che <Ti commit>

LOG INCREMENTALE CON MODIFICHE IMMEDIATE – ESEMPIO

T1	T2
Read(Lr)	Read(Cc2)
Lr = Lr –1500000	Cc2 = Cc2 – 200000
Write(Lr)	Write(Cc2)
Read(Cc1)	commit
Cc1 = Cc1 + 150000	
Write(Cc1)	
Commit	

Esecuzione sequenziale T1; T2

Stato iniziale: Lr = 500.000, Cc1 = 600.000, Cc2 = 800.000

contenuto del log al termine dell'esecuzione delle due transazioni:
< T_1, start >
< T_1, Lr,350000 >
< T_1, Cc1, 750000 >
< T_1, commit >
< T_2, start >
< T_2, Cc2, 600000 >
< T_2, commit >

LOG INCREMENTALE CON MODIFICHE IMMEDIATE – ESEMPIO

se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione Write Cc1, lo stato del log al momento del crash e' il seguente:

< T_1, start >
< T_1, Lr,500000,350000 >
< T_1, Cc1, 600000,750000 >

si esegue l'azione di undo(T1), il nuovo stato dei dati e':
Lr = 500000 Cc1 = 600000 Cc2 = 800000

se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione Write Cc2, lo stato del log al momento del crash e' il seguente:

< T_1, start >
< T_1, Lr,500000,350000 >
< T_1, Cc1, 600000,750000 >
< T_1, commit >
< T_2, start >
< T_2, Cc2, 800000,600000 >

viene effettuato redo(T1) e undo(T2), lo stato della base di dati diventa:

Lr = 350000 Cc1 = 750000 Cc2 = 800000

LIVELLI DI ISOLAMENTO

grado 0 *chaos*: non consente che piu′ transazioni aggiornino simultaneamente un dato

grado 1 *browse:* non consente la sovrascrittura di dati aggiornati da transazioni che non hanno effettuato il commit

grado 2 *cursor stability*: 1 + impone alle transazioni di leggere solo i dati che sono stati aggiornati da transazioni che hanno effettuato il commit

grado 3 *repeatable reads:* una transazione che vuole leggere/scrivere un dato deve attendere il completamento di tutte le altre transazioni che hanno precedentemente letto/scritto quel dato

fornisce un isolamento completo