Basi di dati

Architetture e linee di evoluzione

Capitolo 2

Gestione delle transazioni

Sistema di Gestione di Basi di Dati

- Un Sistema di Gestione di Basi di Dati (DataBase Management System - DBMS) è un Sistema che gestisce collezioni di dati:
  - grandi
  - persistenti
  - condivise

  garantendo
  - privatezza
  - affidabilità
  - efficienza
  - efficacia

Le Basi di Dati sono GRANDI

- Dimensioni (molto) maggiori della memoria centrale dei sistemi di calcolo utilizzati.
- Il limite deve essere solo quello fisico dei dispositivi.

Le Basi di Dati sono PERSISTENTI

- Le Basi di Dati hanno un tempo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano.

Le Basi di Dati sono CONDIVISE

- Ogni organizzazione (specie se grande) è divisa in settori o comunque svolge diverse attività.
- Ciascun settore/attività ha un (sotto)sistema informativo (non necessariamente disgiunto).
- Una base di dati è una risorsa integrata, condivisa fra applicazioni, quindi:
  - Attività diverse su dati condivisi:
    - meccanismi di autorizzazione
  - Accessi di più utenti ai dati condivisi:
    - controllo della concorrenza

I DBMS garantiscono PRIVATEZZA

- Si possono definire meccanismi di autorizzazione per l'accesso ai dati della Base di Dati.
- Esempio:
  - l'utente A è autorizzato a:
    - leggere tutti i dati degli impiegati, dei clienti e dei prodotti
    - modificare solo quelli dei prodotti
  - l'utente B è autorizzato a:
    - leggere i dati degli impiegati e dei clienti
    - modificare quelli dei clienti

I DBMS garantiscono AFFIDABILITÀ

- Affidabilità (per le basi di dati):
  - resistenza a malfunzionamenti hardware e software
- Una base di dati è una risorsa pregiata e quindi deve essere conservata a lungo termine.
- Tecnica fondamentale:
  - gestione delle transazioni

Transazione

- Insieme di operazioni da considerare:
  - indivisibile ("atomico")
  - corretto

  anche in presenza di:
  - concorrenza

  e con effetti:
  - Definitivi

➢ Concetto rilevante con riferimento alle operazioni che modificano il contenuto della base di dati

Definizione di transazione

- Un **sistema transazionale** è un sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione di transazioni
- Un **sistema transazionale** (**OLTP**) è in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti

- Transazione: parte di programma caratterizzata da un inizio (**begin-transaction,** `start transaction` in SQL), una fine (**end-transaction,** non esplicitata in SQL) e al cui interno deve essere eseguito una e una sola volta uno dei seguenti comandi
  - **commit work** per terminare correttamente
  - **rollback work** per abortire la transazione

Una transazione

Transazione in SQL che trasferisce 10 unità da un conto ad un altro.

```
start transaction;
update ContoCorrente
  set Saldo = Saldo + 10 where NumConto = 12202;
update ContoCorrente
  set Saldo = Saldo – 10 where NumConto = 42177;
commit work;
```


Una transazione con varie decisioni

Se il saldo del conto da cui si preleva risulta negativo, allora gli aggiornamenti vanno annullati (ROLLBACK)

```
start transaction;
update ContoCorrente
  set Saldo = Saldo + 10 where NumConto = 12202;
update ContoCorrente
  set Saldo = Saldo – 10 where NumConto = 42177;
select Saldo into A
  from ContoCorrente
  where NumConto = 42177;
if (A>=0)  then commit work
           else rollback work;
```


Il concetto di transazione

- Una unità di elaborazione che gode delle proprietà "ACIDE"
  - Atomicità
  - Consistenza
  - Isolamento
  - Durata (persistenza)

Atomicità

- Una transazione è una unità atomica di elaborazione (INDIVISIBILE)
- Non può lasciare la base di dati in uno stato intermedio (APPROCCIO "TUTTO O NIENTE")
  - un guasto o un errore prima del commit debbono causare l'annullamento (UNDO) delle operazioni svolte fino a quel momento
  - un guasto o errore dopo il commit non deve avere conseguenze; se necessario vanno ripetute (REDO) le operazioni
- Esito
  - Commit = caso "normale" e più frequente
  - Abort (o rollback)
    - richiesto dall'applicazione = suicidio
    - richiesto dal sistema (violazione dei vincoli, concorrenza, incertezza in caso di fallimento) = omicidio

Consistenza

- La transazione rispetta i vincoli di integrità
- Conseguenza:
  - se lo stato iniziale e' corretto
  - anche lo stato finale e' corretto

- Verifica dei vincoli
  - Immediata
    - fatta nel corso della transazione
    - gestione situazioni anomale e rimozione della cause di violazione dei vincoli (NO ABORT)
  - Differita
    - fatta alla conclusione della transazione dopo la richiesta di COMMIT
    - Se viola il vincolo il COMMIT non va a buon fine (ABORT)

Isolamento

- La transazione non risente degli effetti delle altre transazioni concorrenti
  - l'esecuzione concorrente di una collezione di transazioni deve produrre un risultato che si potrbbe ottenerre con una esecuzione sequenziale
- Conseguenza: una transazione non espone i suoi stati intermedi
  - Si evita l' "effetto domino"

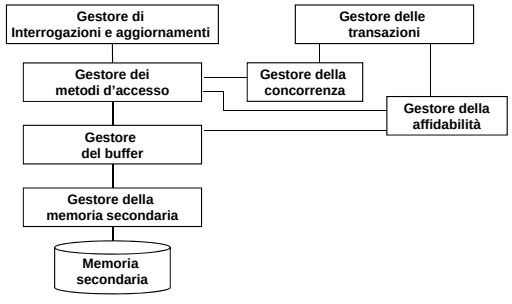
Durabilità (Persistenza)

- Gli effetti di una transazione andata in commit non vanno perduti ("durano per sempre"), anche in presenza di guasti
  - Commit significa impegno

Transazioni e moduli di DBMS

- Atomicità e durabilità
  - Gestore dell'affidabilità
- Isolamento:
  - Gestore della concorrenza
- Consistenza:
  - Gestore dell'integrità a tempo di esecuzione

Gestore degli accessi e delle interrogazioni — Gestore delle transazioni



Gestore delle transazioni, della affidabilità e della concorrenza

- Gestore delle transazioni
  - coordina tutte le attività connesse alle transazioni attraverso l'esecuzione delle istruzioni START TRANSACTION, COMMIT e ROLLBACK
- Gestore della affidabilità
  - garantisce atomicità e persistenza
  - interagisce col gestore dei metodi di accesso per tenere traccia delle operazioni richieste
  - interagisce col gestore del buffer per richiedere eventuali scritture fisiche
- Gestore della concorrenza
  - garantisce l'isolamento filtrando ed eventualmente ripianificando le operazioni di accesso richieste dal gestore degli accessi

Gestore dell'affidabilità

- Gestisce l'esecuzione dei comandi transazionali
  - `start transaction` (B, begin)
  - `commit work` (C)
  - `rollback work` (A, abort)

  e le operazioni di ripristino (recovery) dopo i guasti :
  - *warm restart* e *cold restart*
- Assicura atomicità e durabilità
- Usa il **log**:
  - Un archivio permanente che registra le operazioni svolte dal DBMS
  - Due metafore: il filo di Arianna e i sassolini e le briciole di Hansel e Gretel

Persistenza delle memorie

- **Memoria centrale**: non è persistente
- **Memoria di massa**: è persistente ma può danneggiarsi
- **Memoria stabile**: memoria che non può danneggiarsi (è una astrazione):
  - perseguita attraverso la ridondanza:
    - dischi replicati
    - nastri
    - …

Il log

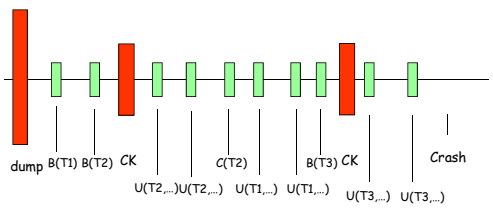
- Il log è un file sequenziale gestito dal controllore dell'affidabilità, scritto in memoria stabile
- "Diario di bordo": riporta tutte le operazioni in ordine temporale di esecuzione
- Record nel log
  - *operazioni delle transazioni*
  - *record di sistema*

Il log

Sia T l'identificativo di una transazione, O l'identificativo dell'oggetto della modifica, BS valore dell'oggetto prima della modifica e AS valore dell'oggetto dopo la modifica
Ogni transazione ha un begin, un commit/abort e più insert, delete e update

- *operazioni delle transazioni*
  - `begin`, B(T)
  - `insert`, I(T,O,AS)
  - `delete`, D(T,O,BS)
  - `update`, U(T,O,BS,AS)
  - `commit`, C(T), `abort`, A(T)
- *record di sistema*
  - `Dump` (rari)
  - `Checkpoint` (frequenti)

Struttura del log



Log, checkpoint e dump: a che cosa servono?

- Il log serve "a ricostruire" le operazioni
- Checkpoint e dump servono ad evitare che la ricostruzione debba partire dall'inizio dei tempi
  - si usano con riferimento a tipi di guasti diversi

Undo e redo

- Undo di una azione su un oggetto *O:*
  - `update`, `delete`: copiare il valore del before state (*BS*) nell'oggetto *O*
  - `insert`: eliminare *O*
- Redo di una azione su un oggetto *O:*
  `insert`, `update`: copiare il valore dell' after state (*AS*) nell'oggetto *O*
  - `delete`: eliminare *O*
- *Idempotenza* di *undo e redo:*
  - *undo*(*undo*(*A*)) = *undo*(*A*)
  - *redo*(*redo*(*A*)) = *redo*(*A*)

Checkpoint

- Operazione svolta periodicamente dal gestore della affidabilità
- Operazione che serve a "fare il punto" della situazione, semplificando le successive operazioni di ripristino:
  - ha lo scopo di registrare quali transazioni sono attive in un certo istante (e dualmente, di confermare che le altre o non sono iniziate o sono finite)
- Paragone (estremo):
  - la "chiusura dei conti" di fine anno di una amministrazione:
    - dal 25 novembre (ad esempio) non si accettano nuove richieste di "operazioni" e si concludono tutte quelle avviate prima di accettarne di nuove

Checkpoint (2)

- Varie modalità, vediamo la più semplice:
  - si sospende l'accettazione di richieste di ogni tipo (scrittura, inserimenti, …, commit, abort) da parte di qualunque transazione
  - si trasferiscono in memoria di massa tutte le pagine relative a transazioni andate in commit
  - si registrano sul log in modo sincrono gli identificatori delle transazioni in corso in un record di checkpoint
    CK(T1,T2,…,Tn) con T1,T2,Tn identificatori transazioni attive
  - si riprende l'accettazione delle operazioni
- Così siamo sicuri che
  - per tutte le transazioni che hanno effettuato il commit i dati sono in memoria di massa
  - le transazioni "a metà strada" sono elencate nel checkpoint

Dump

- Copia completa ("di riserva", backup) della base di dati
  - Solitamente prodotta mentre il sistema non è operativo
  - Salvato in memoria stabile, come backup
  - Un `dump` nel log indica il momento in cui il log è stato effettuato (e dettagli pratici, file, dispositivo, …)

Esito di una transazione

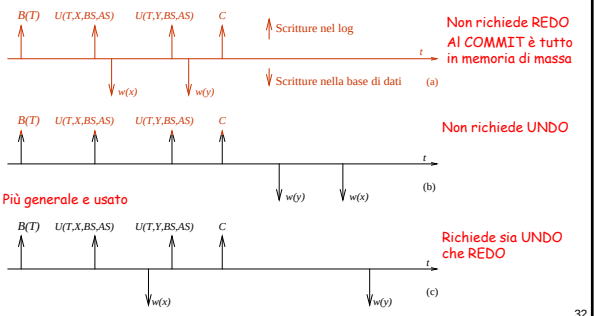
- L'esito di una transazione è determinato irrevocabilmente quando viene scritto il record di **commit** nel log
  - una guasto prima di tale istante porta ad un UNDO di tutte le azioni, per ricostruire lo stato originario della base di dati
  - un guasto successivo non deve avere conseguenze
  - lo stato finale della base di dati deve essere ricostruito, con REDO, se necessario

Regole fondamentali per il log

Il gestore della affidabilità deve garantire che siano seguite le seguenti regole, che definiscono i requisiti per ripristinare la correttezza della BD a fronte di guasti:

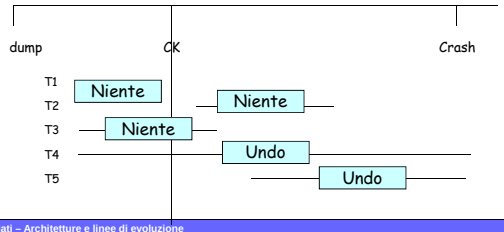
- **Write-Ahead-Log (WAL)**
  - si scrive il log (parte before) prima della BD
  - impone che la parte BEFORE STATE dei record venga scritta nel log prima di effettuare la corrispondente operazione sulla BD
    - consente di disfare le azioni
- **Commit-Precedenza (CP)**
  - si scrive il log (parte after) prima del commit
  - impone che la parte AFTER STATE dei record di log venga scritta prima del COMMIT
    - consente di rifare le azioni
- In realtà le componenti BEFORE e AFTER vengono scritte insieme
  - WAL: record di log scritti prima dei record nel DB
  - CP: record di log scritti prima di COMMIT

Scrittura nel log e nella base di dati



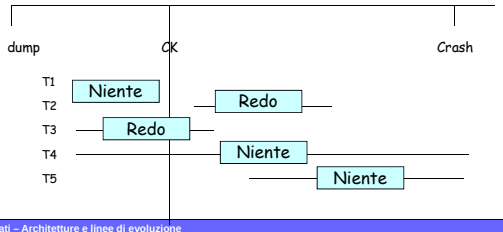
Modalità immediata

- Il DB contiene valori AS provenienti da transazioni uncommitted
- Richiede UNDO delle operazioni di transazioni uncommited al momento del guasto
- Non richiede REDO



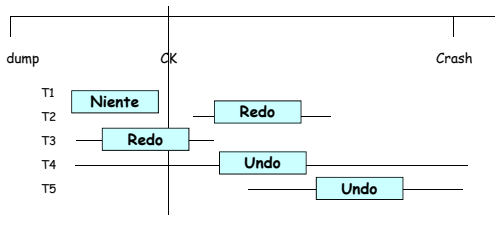
Modalità differita

- Il DB non contiene valori AS provenienti da transazioni uncommitted
- In caso di abort, non occorre fare niente
- Rende superflua la procedura di UNDO. Richiede REDO



Esiste una terza modalità: modalità mista

- La scrittura può avvenire in modalità sia immediata che differita
- Consente l'ottimizzazione delle operazioni di flush
- Richiede sia UNDO che REDO



Guasti

- **Guasto di sistema**
  - **Guasti "soft":** errori di programma, crash di sistema, caduta di tensione
    - si perde la memoria centrale e quindi tutti i buffer
    - non si perde la memoria secondaria e quindi BD e log

    warm restart, ripresa a caldo
- **Guasto del dispositivo**
  - **Guasti "hard":** sui dispositivi di memoria secondaria
    - si perde anche la memoria secondaria
    - non si perde la memoria stabile (e quindi il log)

    cold restart, ripresa a freddo

La perdita del LOG viene definita come evento CATASTROFICO per il quale non viene suggerito rimedio

Modello "fail-stop"

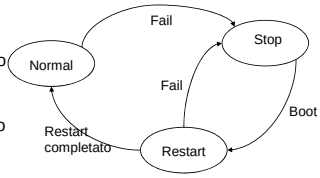
Modello ideale in cui ci si pone.

A fronte di un guasto, sia SOFT che HARD, il sistema forza subito un arresto completo delle transazioni e il successivo ripristino del corretto funzionamento.

Ripresa a caldo nel caso di guasto del sistema (SOFT)

Ripresa a freddo nel caso di guasto del dispositivo (HARD)

Al termine della procedura di ripresa il sistema è nuovamente utilizzabile.



Processo di restart

- Obiettivi della procedura di ripresa
- Classificare le transazioni in
  - completate (tutti i dati in memoria stabile)
  - in commit ma non necessariamente completate (può servire redo)
  - senza commit (vanno annullate, undo)

Ripresa a caldo

Garantisce atomicità e persistenza delle transazioni

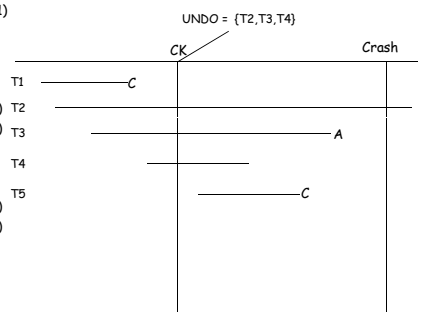
Quattro fasi:

- trovare l'ultimo checkpoint (ripercorrendo il log a ritroso)
- costruire gli insiemi *UNDO* (transazioni da disfare) e *REDO* (transazioni da rifare)
- ripercorrere il log all'indietro, fino alla più vecchia azione delle transazioni in *UNDO e REDO*, disfacendo tutte le azioni delle transazioni in *UNDO*
- ripercorrere il log in avanti, rifacendo tutte le azioni delle transazioni in *REDO*

Esempio di warm restart

LOG

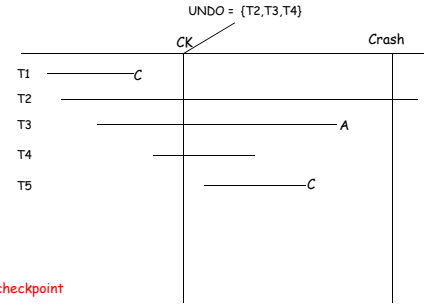
B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
C(T5)
I(T2,O6,A8)



1. Ricerca dell'ultimo checkpoint

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
C(T5)
I(T2,O6,A8)

Si accede al record di checkpoint



2. Costruzione degli insiemi UNDO e REDO

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
9. U(T4,O3,B4,A4)
CK(T2,T3,T4)
1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
10. U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4} Setup

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Si percorre il avanti il log e si aggiornano gli insiemi di UNDO e REDO

3. Fase UNDO

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
9. U(T4,O3,B4,A4)
CK(T2,T3,T4)
1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
10. U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}
1. C(T4) → UNDO = {T2, T3}. REDO = {T4}
2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4} Setup
3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}
4. D(O6)
5. O5 =B7
6. O3 = B5
7. O2 =B3
8. O1=B1

Undo

Si ripercorre indietro il log fino all'azione U(T2,O1,B1,A1) eseguendo le operazioni di UNDO
D(O6)
Re-Insert (O5=B7)
...

4. Fase REDO

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
9. U(T4,O3,B4,A4)
CK(T2,T3,T4)
1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
10. U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}
1. C(T4) → UNDO = {T2, T3}. REDO = {T4}
2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4} Setup
3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}
4. D(O6)
5. O5 =B7
6. O3 = B5
7. O2 =B3
8. O1=B1
9. O3 = A4
10. O4 = A6

Undo

Redo

Vengono svolte le operazioni di REDO (A4=B5)

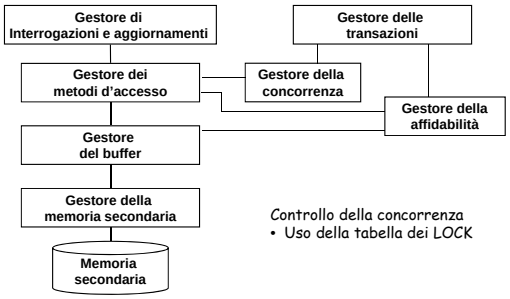
Ripresa a freddo

Risponde ad un guasto che provoca il deterioramento della BD

- Si ripristinano i dati a partire dal backup
- Si eseguono le operazioni registrate sul giornale fino all'istante del guasto
- Si esegue una ripresa a caldo

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni



Controllo della concorrenza
- Uso della tabella dei LOCK

Controllo di concorrenza

Un DBMS deve servire diverse applicazioni e rispondere alle richieste di diversi utenti

- La concorrenza è fondamentale: decine o centinaia di transazioni al secondo (tps), non possono essere seriali
- Esempi: banche, prenotazioni aeree

Modello di riferimento

- Operazioni di input-output su oggetti astratti *x*, *y*, *z*

Problema

- Anomalie causate dall'esecuzione concorrente, che quindi va governata

Anomalie

- Perdita di aggiornamento
- Lettura sporca
- Letture inconsistenti
- Aggiornamento fantasma
- Inserimento fantasma

Perdita di aggiornamento

- Due transazioni identiche:
  - t 1 : $r(x)$, $x = x + 1$, $w(x)$
  - t 2 : $r(x)$, $x = x + 1$, $w(x)$
- Inizialmente x=2; dopo un'esecuzione seriale x=4
- Un'esecuzione concorrente:

t_1	t_2
bot	
$r_1(x)$	
$x = x + 1$	
	bot
	$r_2(x)$
	$x = x + 1$
$w_1(x)$	
commit	
	$w_2(x)$
	commit

- Un aggiornamento viene perso: x=3
- Entrambe leggono 2 come valore iniziale

Lettura sporca

t_1	t_2
bot	
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	bot
	$r_2(x)$
abort	
	commit

- Aspetto critico: t_2 ha letto uno stato intermedio ("sporco") e lo può comunicare all'esterno
- L'unico modo per ripristinare la correttezza a seguito dell'ABORT di t_1 sarebbe imporre ricorsivamente l'ABORT di tutte le transazioni che avessero letto il dati modificati da t_1
  - Effetto domino: oneroso da gestire e spesso non praticabile

Letture inconsistenti

- t_1 legge due volte:

t_1	t_2
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

- t_1 legge due valori diversi per x !
- Sarebbe invece opportuno che una transazione che accede due volte alla BD trovasse lo stesso valore per ciascun dato letto

Aggiornamento fantasma

- Assumere ci sia un vincolo $y + z = 1000$;

t_1	t_2
bot	
$r_1(y)$	
	bot
	$r_2(y)$
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s = y + z$	
commit	

- s = 1100: il vincolo sembra non soddisfatto, t_1 vede un aggiornamento non coerente
- t_1 non viola il vincolo
- t_1 osserva uno stato che non soddisfa i vincoli

Inserimento fantasma

t_1	t_2
bot	
"legge gli stipendi degli impiegati del dip A e calcola la media"	
	bot
	"inserisce un impiegato in A"
	commit
"legge gli stipendi degli impiegati del dip A e calcola la media"	
commit	

Le due medie calcolate potranno essere differenti

Anomalie

• Perdita di aggiornamento	W-W
• Lettura sporca	R-W (o W-W) con abort
• Letture inconsistenti	R-W
• Aggiornamento fantasma	R-W
• Inserimento fantasma	R-W su dato "nuovo"

Teoria del controllo della concorrenza

- Modello formale di TRANSAZIONE come sequenza di operazioni di lettura e scrittura
  - $t_1 : r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$

Schedule

- Sequenza di operazioni di input/output di transazioni concorrenti
- Le operazioni compaiono nello schedule seguendo l'ordine temporale in cui sono eseguite sulla BD
- Esempio:

  $S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$

  $r_1(x)$ lettura dell'oggetto x effettuata dalla transazione t_1

- Ipotesi semplificativa (che rinuoveremo in futuro, in quanto non accettabile in pratica):
  - consideriamo la **commit-proiezione** e ignoriamo le transazioni che vanno in abort, rimuovendo tutte le loro azioni dallo schedule

Controllo di concorrenza

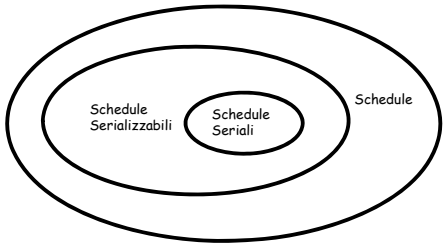
- *Obiettivo*: evitare le anomalie
- *Scheduler*: un sistema che accetta o rifiuta (o riordina) le operazioni richieste dalle transazioni
- *Schedule seriale:* le transazioni sono separate, una alla volta

  $S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$

- *Schedule serializzabile:* produce lo stesso risultato di uno schedule seriale sulle stesse transazioni
  - Richiede una nozione di equivalenza fra schedule

Idea base

- Individuare classi di schedule serializzabili che siano sottoclassi degli schedule possibili, siano serializzabili e la cui proprietà di serializzabilità sia verificabile a costo basso



View-Serializzabilità

- Definizioni prelilminari:
  - $r_i(x)$ ***legge-da*** $w_j(x)$ in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è $w_k(x)$ fra $r_i(x)$ e $w_j(x)$ in S
  - $w_i(x)$ in uno schedule S è ***scrittura finale*** *se* è l'ultima scrittura dell'oggetto x in S
- Schedule ***view-equivalenti*** ($S_i \approx_V S_j$): hanno la stessa relazione *legge-da* e le stesse scritture finali
- Uno schedule è ***view-serializzabile*** se è view-equivalente ad un qualche schedule seriale
- L'insieme degli schedule view-serializzabili è indicato con **VSR**

View serializzabilità: esempi

- $S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  $S_4 : w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$
  $S_5 : w_0(x)\ r_1(x)\ w_1(x)\ r_2(x)\ w_1(z)$
  $S_6 : w_0(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_2(x)$
  - S_3 è view-equivalente allo schedule seriale S_4 (e quindi è view-serializzabile)
  - S_5 non è view-equivalente a S_4, ma è view-equivalente allo schedule seriale S_6, e quindi è view-serializzabile
- $S_7 : r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$ (perdita di aggiornamento)
  $S_8 : r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$ (letture inconsistenti)
  $S_9 : r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$ (aggiornamento fantasma)
  - S_7, S_8, S_9 non view-serializzabili

View serializzabilità

- Complessità:
  - la verifica della view-equivalenza di due dati schedule:
    - polinomiale
  - decidere sulla View serializzabilità di uno schedule:
    - problema NP-completo
- Non è utilizzabile in pratica

- Si preferisce definire una condizione di equivalenza più ristretta che non copra tutti i casi di equivalenza tra schedule, ma sia utilizzabile nella pratica (avendo complessità inferiore)

Conflict-serializzabilità

- Si basa sul concetto di conflitto
- Definizione preliminare:
  - Un'azione a_i è in *conflitto* con a_j ($i \neq j$), se operano sullo stesso oggetto e almeno una di esse è una scrittura. Due casi:
    - conflitto *read-write* (*rw* o *wr*)
    - conflitto *write-write* (*ww*).
- *Schedule conflict-equivalenti* ($S_i \approx_C S_j$): includono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi
- Uno schedule è *conflict-serializable* se esiste uno schedule seriale a esso *conflict-equivalente*
- L'insieme degli schedule conflict-serializzabili è indicato con **CSR**

Verifica di conflict-serializzabilità

- Per mezzo del **grafo dei conflitti**:
  - un nodo per ogni transazione t_i
  - un arco (orientato) da t_i a t_j se c'è almeno un conflitto fra un'azione a_i e un'azione a_j tale che a_i precede a_j
- Teorema
  - **Uno schedule è in CSR se e solo se il grafo è aciclico**

CSR e aciclicità del grafo dei conflitti

- Se uno schedule S è CSR allora è $\approx_C$ ad uno schedule seriale. Supponiamo le transazioni nello schedule seriale ordinate secondo il TID: $t_1, t_2, \ldots, t_n$. Poiché lo schedule seriale ha tutti i conflitti nello stesso ordine dello schedule S, nel grafo di S ci possono essere solo archi (i,j) con i<j e quindi il grafo non può avere cicli, perché un ciclo richiede almeno un arco (i,j) con i>j.
- Se il grafo di S è aciclico, allora esiste fra i nodi un "ordinamento topologico" (cioè una numerazione dei nodi tale che il grafo contiene solo archi (i,j) con i<j). Lo schedule seriale le cui transazioni sono ordinate secondo l'ordinamento topologico è equivalente a S, perché per tutti i conflitti (i,j) si ha sempre i<j.

Controllo della concorrenza in pratica

- Anche la conflict-serializzabilità, pur più rapidamente verificabile (l'algoritmo, con opportune strutture dati richiede tempo lineare), è inutilizzabile in pratica
- La tecnica sarebbe efficiente se potessimo conoscere il grafo dall'inizio, ma così non è: uno scheduler deve operare "incrementalmente", cioè ad ogni richiesta di operazione decidere se eseguirla subito oppure fare qualcos'altro; non è praticabile mantenere il grafo, aggiornarlo e verificarne l'aciclicità ad ogni richiesta di operazione
- Inoltre, la tecnica si basa sull'ipotesi di commit-proiezione
- In pratica, si utilizzano tecniche che
  - garantiscono la conflict-serializzabilità senza dover costruire il grafo
  - non richiedono l'ipotesi della commit-proiezione

Lock

- Il meccanismo principale di controllo della concorrenza è basato sul locking
- Principio:
  - Tutte le letture sono precedute da *r_lock* (lock condiviso) e seguite da *unlock*
  - Tutte le scritture sono precedute da *w_lock* (lock esclusivo) e seguite da *unlock*
- Quando una transazione prima legge e poi scrive un oggetto, può:
  - richiedere subito un lock esclusivo
  - chiedere prima un lock condiviso e poi uno esclusivo (*lock escalation*)
- Il *lock manager* riceve queste richieste dalle transazioni e le accoglie o rifiuta, sulla base della tavola dei conflitti

Gestione dei lock

- Politica per concedere i lock: basata sulla tavola dei conflitti

Richiesta	Stato della risorsa		
	free	*r_locked*	*w_locked*
r_lock	OK / r_locked	OK / r_locked	NO/ w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / depends	OK / free

(OK/NO: esito della richiesta – secondo valore: stato assunto dalla risorsa dopo l'esecuzione della primitiva)

  - Un contatore tiene conto del numero di "lettori"; la risorsa è rilasciata quando il contatore scende a zero (depends)
- Se la risorsa non è concessa, la transazione richiedente è posta in attesa (eventualmente in coda), fino a quando la risorsa non diventa disponibile
- Il lock manager gestisce una tabella dei lock, per ricordare la situazione

Locking a due fasi

- Usato da quasi tutti i sistemi
- Garantisce "a priori" la conflict-serializzabilità a-priori
- Basata su due regole:
  - "proteggere" tutte le letture e scritture con lock
  - un vincolo sulle richieste e i rilasci dei lock:
    - **una transazione, dopo aver rilasciato un lock, non può acquisirne altri**
- Come conseguenza di questo principio si possono distinguere due fasi nell'esecuzione della transazione:
  - Crescente: si acquisiscono i lock
  - Calante: si rilasciano i lock