Corso di Laurea in Ingegneria Informatica

Fondamenti di Informatica II

Modulo "*Basi di dati*"

a.a. 2017-2018

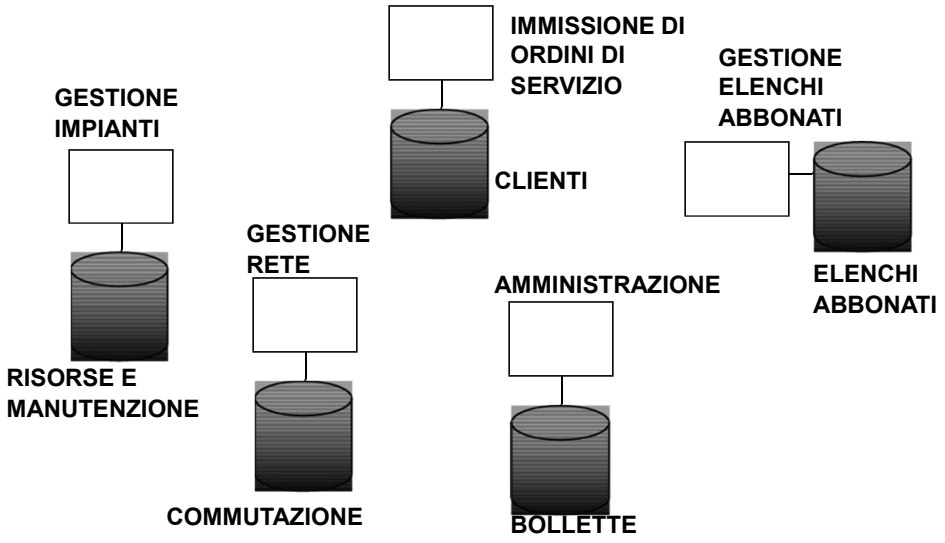
Docente: Gigliola Vaglini

Docente laboratorio: Francesco Pistolesi

Lezione 9

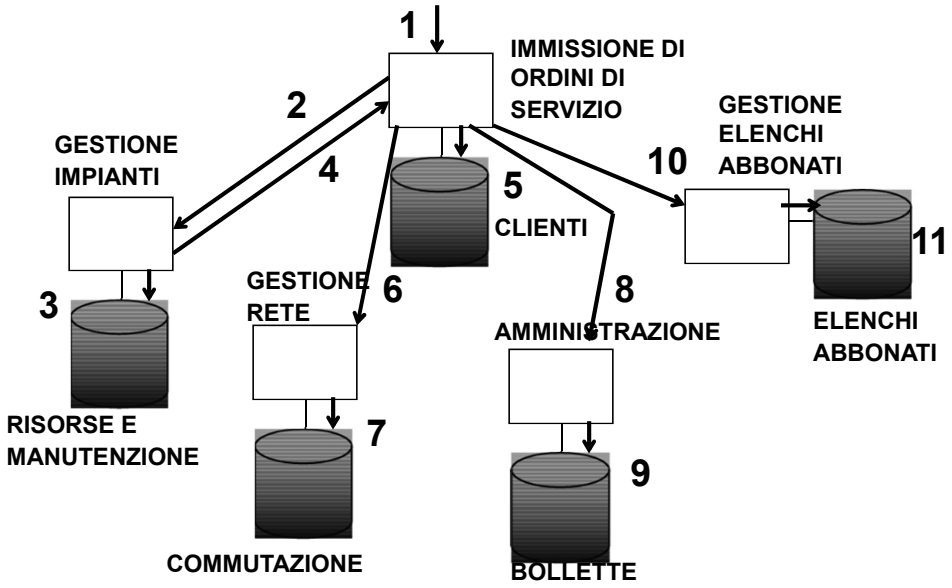
Gestione delle transazioni

Esempio di sistema informativo



3

Esempio di transazione



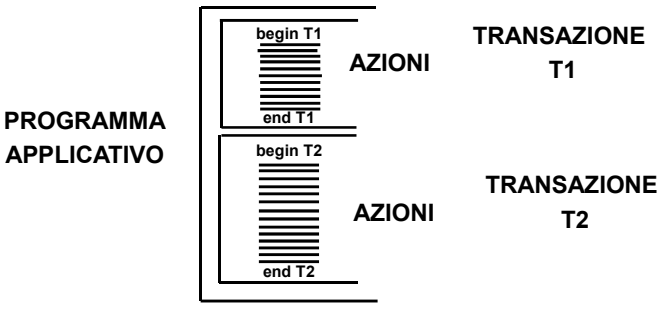
4

Definizione di transazione

- 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
- Un **sistema transazionale** è in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti

5

Applicazioni e transazioni



6

Una transazione

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

7

Una transazione con varie decisioni

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

8

Proprietà delle transazioni

- Proprietà "ACIDE"
  - Atomicità
  - Consistenza
  - Isolamento
  - Durata (persistenza)

9

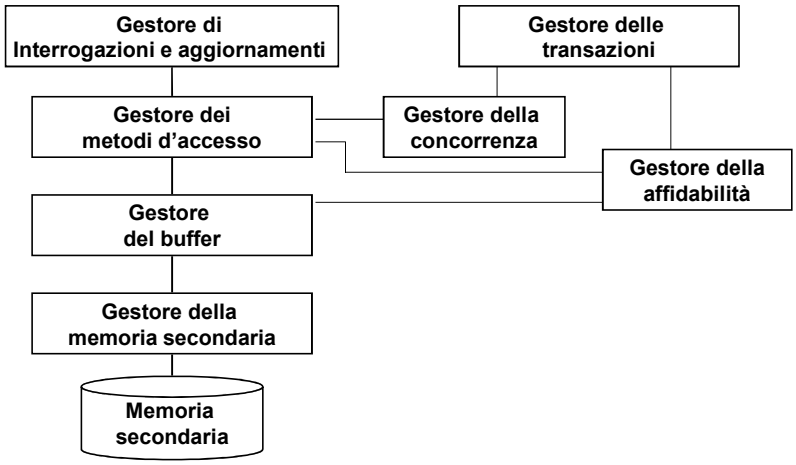
Transazioni e moduli di DBMS

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

10

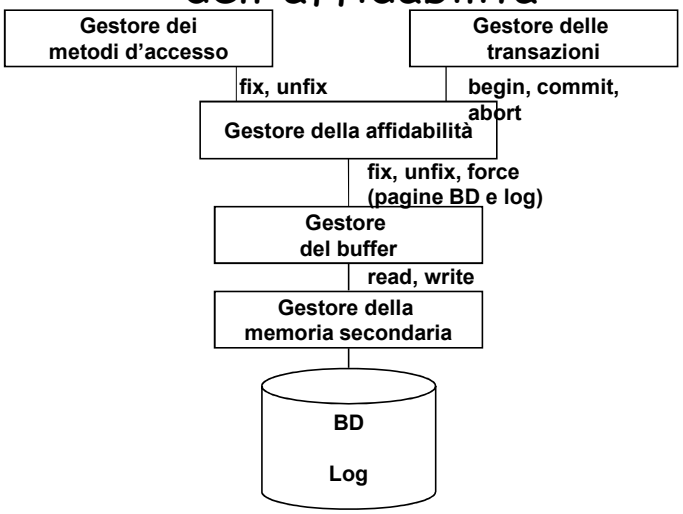
Gestore degli accessi e delle interrogazioni

Gestore delle transazioni



11

Architettura del controllore dell'affidabilità



12

Gestore dell'affidabilità

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

  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

13

Persistenza delle memorie

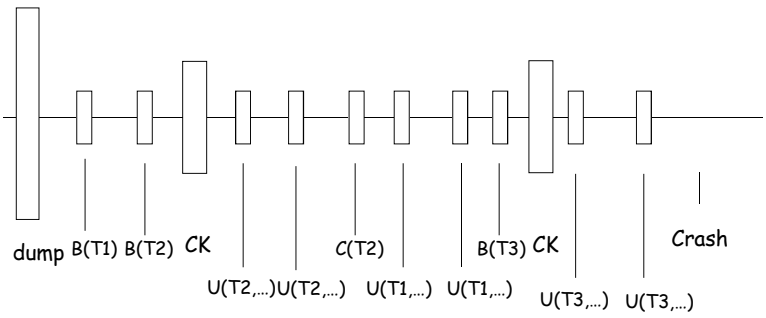
- **Memoria centrale**: non è persistente
- **Memoria di massa**: è persistente ma può danneggiarsi
- **Memoria stabile**: memoria che non può danneggiarsi (è una astrazione):
  - perseguita attraverso la ridondanza:
    - dischi replicati
    - nastri
    - ...

14

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
- Record nel log
  - *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`
    - `checkpoint`

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 (vedi avanti)

17

Scritture nel log

- I record nel log sono di due tipi
  - Record di sistema: checkpoint e dump vengono scritti dal controllore dell'affidabilità
  - Record di transazione: attività svolte dalle transazione nell'ordine in cui sono svolte (begin, commit, rollback, insert, delete, update)

18

Checkpoint

- 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, transazioni "a metà strada"
  - e, dualmente, di confermare che le altre o non sono iniziate o sono finite; infatti per tutte le transazioni che hanno effettuato il commit i dati sono in memoria di massa

19

Descrizione dell'operazione Ceckpoint

- Si sospende l'accettazione delle operazioni di write, commit, abort da parte delle transazioni
- Si forza (force) la scrittura in memoria di massa delle pagine del buffer modificate da transazioni che hanno fatto commit
- Si forza (force) la scrittura nel log di un record contenente gli identificatori delle transazioni attive
- Si riprende ad accettare le operazioni da parte delle transazioni

20

Dump

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

21

Record di transazione

- Begin,commit,rollback: identificativo transazione (T)
- Update: T, O, BS (before state), AS (after state)
- Insert: T, AS
- Delete: T, BS

22

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
- record di **`abort (rollback)`** possono essere scritti in modo asincrono

23

Quando il controllore dell'affidabilità può consentire la modifica del log da parte delle transazioni

- **Regola Write-Ahead-Log**:
  - si scrive la parte BS dei record del log prima di effettuare la corrispondente operazione sul database
    - consente di disfare le azioni di transazioni senza commit avendo in memoria stabile un valore corretto
- **Regola Commit-Precedenza**:
  - si scrive la parte AS dei record di log prima del commit
    - consente di rifare le azioni di transazioni che hanno già fatto commit

24

In pratica

- il record di log viene scritto contemporaneamente in tutte le sue componenti
- Il *commit* si considera effettuato quando il corrispondente record di log è scritto
  - prima di questa scrittura il guasto causa l'*undo* di tutte le operazioni
  - dopo il guasto causa il *redo* di tutte le operazioni

25

I protocolli per la scrittura

- nel più usato la scrittura del log avviene prima di quella nella base di dati
- la scrittura nella base di dati può avvenire in qualunque momento, anche prima del *commit*

26

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*)

27

Guasti

- **Guasti "soft":** errori di programma, crash di sistema, caduta di tensione
  - si perde la memoria centrale e quindi anche il buffer
  - non si perde la memoria secondaria, cioè la base di dati e il log

  warm restart, ripresa a caldo
- **Guasti "hard":** dei dispositivi di memoria secondaria
  - si perde anche la memoria secondaria, i.e. la base di dati
  - non si perde la memoria stabile (e quindi il log)

  cold restart, ripresa a freddo
- **La perdita del log è considerato un evento catastrofico e quindi non è definita alcuna strategia di recupero.**

28

Processo di restart

- Obiettivo: classificare le transazioni in
  - completate (tutti i dati in memoria stabile)
  - in commit ma non necessariamente completate (vanno rifatte, redo)
  - senza commit (vanno annullate, undo)

29

Modello fail-stop

1. L'individuazione di un guasto forza l'arresto completo delle transazioni
2. Il sistema operativo viene riavviato
3. Viene avviata una procedura di restart
4. Al termine del restart il buffer è vuoto, ma le transazioni possono ripartire

30

Ripresa a caldo

Quattro fasi:

- trovare l'ultimo checkpoint (ripercorrendo il log a ritroso)
- costruire gli insiemi *UNDO* (transazioni da disfare) e *REDO* (transazioni da rifare)
  - UNDO riguarda le transazioni attive ma non committed, REDO le transazioni che sono committed prima del guasto
- 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*

31

Ripresa a freddo

- Si ripristina la parte di dati deteriorata a partire dal backup e ci si riporta al record di dump più recente nel log
- Si eseguono le operazioni registrate sul giornale sulla parte deteriorata fino all'istante del guasto
- Si esegue una ripresa a caldo

32

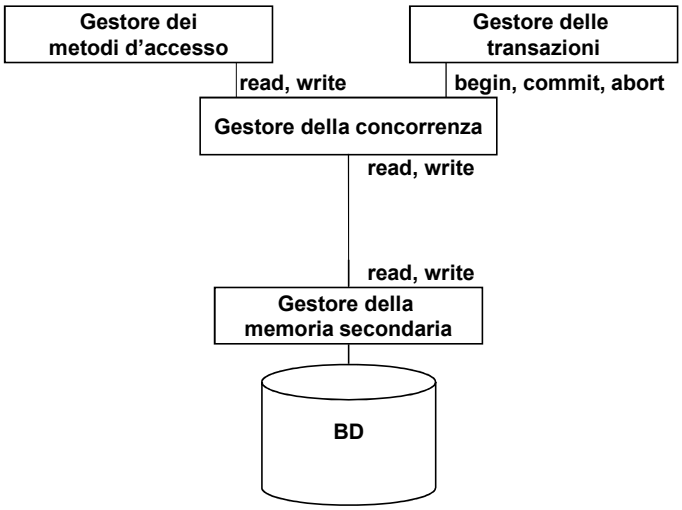
Controllo di concorrenza

- La concorrenza è fondamentale: decine o centinaia di transazioni al secondo, non possono essere seriali

Problema

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

Architettura del controllore della concorrenza



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

- Un aggiornamento viene perso: x=3

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

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 !

37

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	

- $s = 1100$: t_1 vede un aggiornamento non completo

38

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	

39

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"

40

Schedule

- Sequenza di operazioni di input/output di transazioni concorrenti
- Esempio:

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

41

Controllo di concorrenza

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

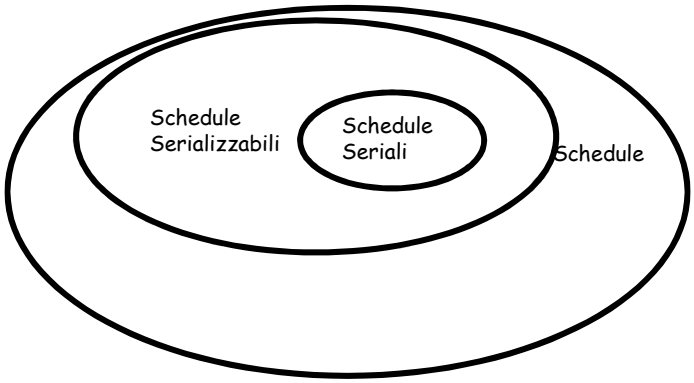
$$S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$$

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

42

Idea base

- Individuare classi di schedule serializzabili la cui proprietà di serializzabilità sia verificabile a costo basso



43

View-Serializzabilità

- Schedule ***view-equivalenti*** ($S_i \approx_V S_j$): hanno la stessa relazione *legge-da* e le stesse scritture finali su ogni oggetto.
- Uno schedule è ***view-serializzabile*** se è view-equivalente ad un qualche schedule seriale
- L'insieme degli schedule view-serializzabili è indicato con **VSR**

- Esiste la relazione legge-da tra $r_i(x)$ e $w_j(x)$ in S se $w_j(x)$ precede $r_i(x)$ in S e non c'è nessun $w_k(x)$ ($k \neq j$) tra di loro.
- $w_i(x)$ in S è ***scrittura finale*** se è l'ultima scrittura sull'oggetto x in S

44

View serializzabilità: esempi

- S_1 : $w_{01}(x)\ r_{21}(x)\ r_{11}(x)\ w_{22}(x)\ w_{23}(z)$
  S_2 : $w_{01}(x)\ r_{11}(x)\ r_{21}(x)\ w_{22}(x)\ w_{23}(z)$
  - S_1 è view-equivalente allo schedule seriale S_2 (e quindi è view-serializzabile)

- S_3 : $r_{11}(x)\ r_{21}(x)\ w_{12}(x)\ w_{22}(x)$ (perdita di aggiornamento)
  S_4 : $r_{11}(x)\ r_{21}(x)\ w_{22}(x)\ r_{12}(x)$ (letture inconsistenti)
  S_5 : $r_{11}(x)\ r_{12}(y)\ r_{21}(z)\ r_{22}(y)\ w_{23}(y)\ w_{24}(z)\ r_{13}(z)$
  (aggiornamento fantasma)
  - S_3, S_4, S_5 non view-serializzabili, non view-equivalenti a nessun schedule seriale

45

View serializzabilità: verifica

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

46

Conflict-serializzabilità

- 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 è conflict-equivalente ad un qualche schedule seriale
- L'insieme degli schedule conflict-serializzabili è indicato con **CSR**

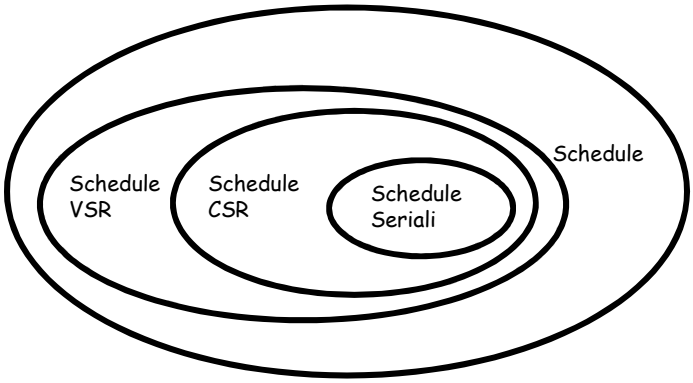
47

VSR e CSR

- Ogni schedule conflict-serializable è anche view-serializable
  - CSR implica VSR

48

CSR e VSR



49

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**

50

Grafo dei conflitti

- S= r1(x) w2(x)r3(x)r1(y)w2(y)r1(v)w3(v)r4(v)w4(y)w5(y)

x	y	v
r1	r1	r1
w2	w2	w3
r3	w4	r4
	w5	

51

Conflict-serializabilità: verifica

- La conflict-serializabilità è più rapidamente verificabile (l'algoritmo, con opportune strutture dati richiede tempo lineare), ma necessita della costruzione del grafo dei conflitti ad ogni richiesta di scrittura
- Quindi non è praticabile mantenere il grafo, aggiornarlo e verificarne l'aciclicità ad ogni richiesta di operazione

52

In pratica

- In pratica, si utilizzano tecniche che
  - garantiscono la conflict-serializzabilità senza dover costruire il grafo

53

Lock

- 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

54

Transazioni ben formate rispetto al locking

- Ogni read è preceduta da un *r_lock* e seguita da un *unlock*
- Ogni write è preceduta da un *w_lock* e seguita da un *unlock*

55

Gestione dei lock

- Basata sulla tavola dei conflitti (politica per la gestione 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

  - Un contatore tiene conto del numero di "lettori"; la risorsa è rilasciata quando il contatore scende a zero
- 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 gestore della concorrenza (o lock manager o scheduler) gestisce una tabella dei lock (lock già concessi), per ricordare la situazione

56

Locking a due fasi

- Usato da quasi tutti i sistemi
- Garantisce "a priori" la conflict-serializzabilità
- 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 finchè tutti quelli che ha acquisito non sono stati rilasciati**

57

2PL e CSR

- Ogni schedule 2PL e' anche conflict serializzabile, ma non è vero il viceversa
  - 2PL implica CSR

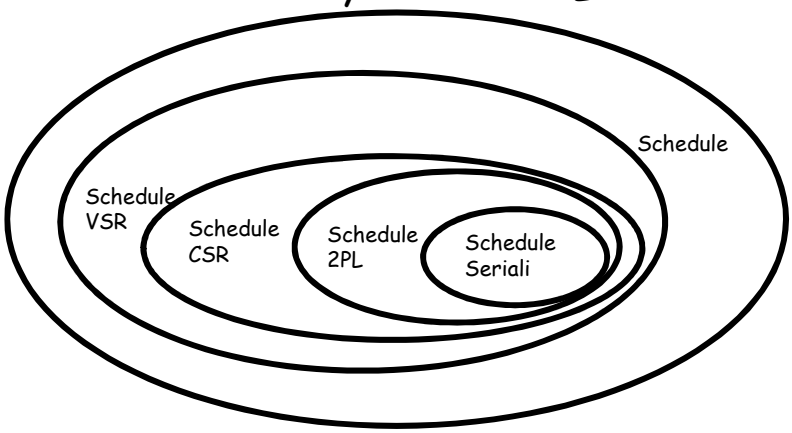
58

Locking a due fasi stretto

- Condizione aggiuntiva:
  - **I lock possono essere rilasciati solo dopo il commit o abort**
- elimina il rischio di letture sporche

59

CSR, VSR e 2PL



60

Controllo di concorrenza basato su timestamp

- Tecnica alternativa al 2pL
- **Timestamp**:
  - identificatore che definisce un ordinamento totale sugli eventi di un sistema
- Ogni transazione ha un timestamp che rappresenta l'istante di inizio della transazione
- Uno schedule è accettato solo se riflette l'ordinamento seriale delle transazioni indotto dai timestamp

Dettagli

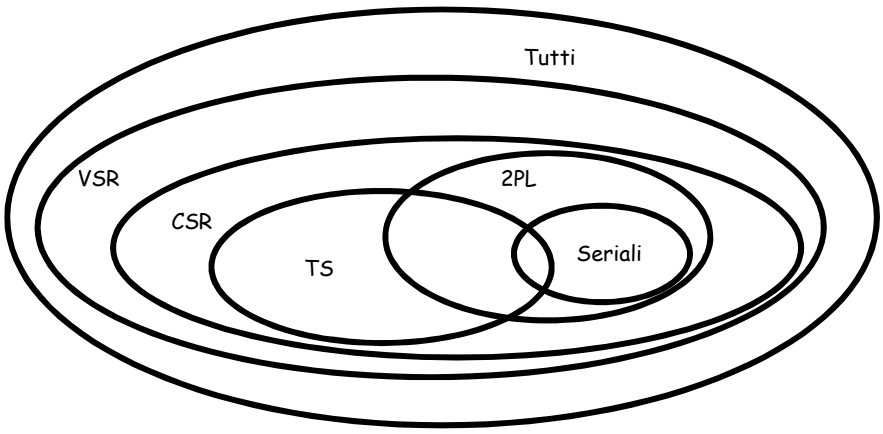
- Lo scheduler ha due contatori RTM(x) e WTM(x) per ogni oggetto
- Lo scheduler riceve richieste di letture e scritture (con indicato il timestamp della transazione):
  - *read*(x,*ts*):
    - se *ts* < WTM(x) allora la richiesta è respinta e la transazione viene uccisa;
    - altrimenti, la richiesta viene accolta e RTM(x) è posto uguale al maggiore fra RTM(x) e *ts*
  - *write*(x,*ts*):
    - se *ts* < WTM(x) o *ts* < RTM(x) allora la richiesta è respinta e la transazione viene uccisa,
    - altrimenti, la richiesta viene accolta e WTM(x) è posto uguale a *ts*
- Vengono uccise molte transazioni

2PL vs TS

- Sono incomparabili

63

CSR, VSR, 2PL e TS



64

2PL vs TS

- In 2PL le transazioni sono poste in attesa, in TS uccise e rilanciate
  - Le ripartenze sono di solito più costose delle attese:
  - conviene il 2PL
- 2PL può causare deadlock

65

Stallo (deadlock)

- Attese incrociate: due transazioni detengono ciascuna una risorsa e aspetttano la risorsa detenuta dall'altra
- Esempio:
  - t_1: *read*(*x*), *write*(*y*)
  - t_2: *read*(*y*), *write*(*x*)
  - Schedule:

    $r\_lock_1(x)$, $r\_lock_2(y)$, $read_1(x)$, $read_2(y)$
    $w\_lock_1(y)$, $w\_lock_2(x)$

66

Risoluzione dello stallo

- Uno stallo corrisponde ad un ciclo nel grafo delle attese
- Tre tecniche di risoluzione

1. Timeout (problema: scelta dell'intervallo, con trade-off)
2. Rilevamento dello stallo
   - ricerca di cicli nel grafo delle attese
3. Prevenzione dello stallo
   - Prevenzione: uccisione di transazioni "sospette"

Livelli di isolamento in SQL:1999 (e JDBC)

- Le transazioni possono essere definite **`read-only`** (non possono richiedere lock esclusivi)
- Il livello di isolamento può essere scelto per ogni transazione
  - **`read uncommitted`** permette letture sporche, letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`read committed`** evita letture sporche ma permette letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`repeatable read`** evita tutte le anomalie esclusi gli inserimenti fantasma
  - **`serializable`** evita tutte le anomalie
- Nota:
  - la perdita di aggiornamento è sempre evitata

Livelli di isolamento: implementazione

- Sulle scritture si ha sempre il 2PL stretto (e quindi si evita la perdita di aggiornamento)
- **`read uncommitted`**:
  - nessun lock in lettura (e non rispetta i lock altrui)
- **`read committed`**:
  - lock in lettura (e rispetta quelli altrui), ma senza 2PL
- **`repeatable read`**:
  - 2PL anche in lettura
- **`serializable`**:
  - 2PL

Esempio

- Dire se i seguenti due schedule sono view-equivalenti o conflict-equivalenti o nessuna delle due cose.
-

S1=w2(x) r2(x) w1(x) r1(x) w2(y) r2(y) w1(x) w2(z)

S2= w1(x) r1(x) w2(x) r2(x) w1(x) w2(y) r2(y) w2(z)

-