Basi di dati Vol.2

Capitolo 2

Gestione delle transazioni

30/03/2012

Le basi di dati sono affidabili

- Le basi di dati sono una risorsa per chi le possiede, e debbono essere conservate anche in presenza di malfunzionamenti

- Esempio:
  - un trasferimento di fondi da un conto corrente bancario ad un altro, con guasto del sistema a metà

- È necessario il supporto alla gestione di **transazioni**
  - atomiche (o tutto o niente)
  - definitive: dopo la conclusione, non si dimenticano

Le basi di dati vengono aggiornate …

- L'**affidabilità** è impegnativa per via degli aggiornamenti frequenti e della necessità di gestire il buffer

Le basi di dati sono condivise

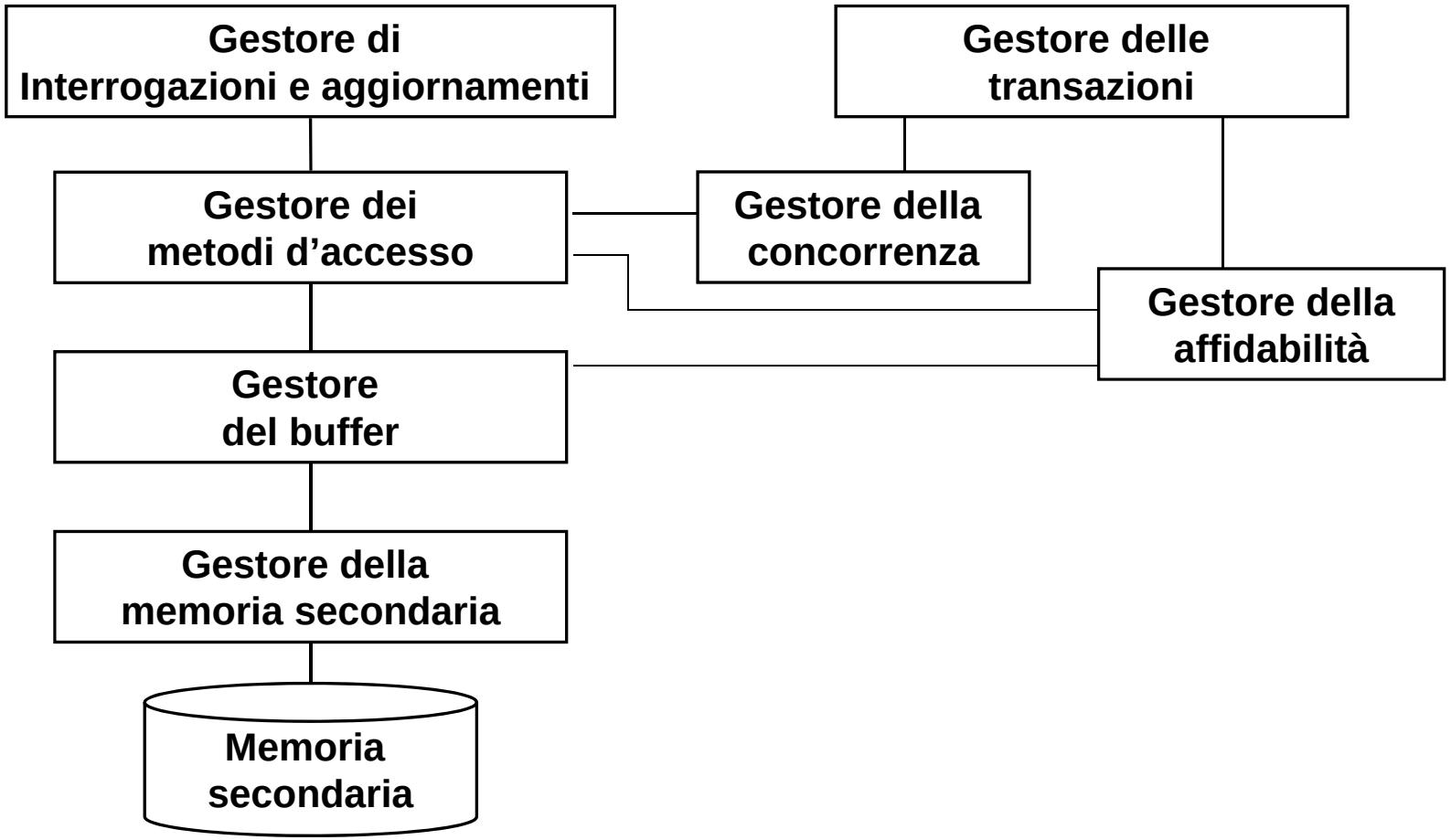
- Una base di dati è una risorsa **integrata**, **condivisa** fra le varie applicazioni
- conseguenze
  - Attività diverse su dati in parte condivisi:
    - meccanismi di autorizzazione
  - Attività multi-utente su dati condivisi:
    - controllo della **concorrenza**

Aggiornamenti su basi di dati condivise …

- Esempi:
  - due prelevamenti (quasi) contemporanei sullo stesso conto corrente
  - due prenotazioni (quasi) contemporanee sullo stesso posto
- Intuitivamente, le transazioni sono corrette se **seriali** (prima una e poi l'altra)
- Ma in molti sistemi reali l'efficienza sarebbe penalizzata troppo se le transazioni fossero seriali:
  - il **controllo della concorrenza** permette un ragionevole compromesso

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni



DEFINIZIONE DI TRANSAZIONE

- Transazione: parte di programma caratterizzata da un inizio (**begin-transaction**, non sempre esplicitata), una fine (**end-transaction,** quasi mai esplicitata) 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

  la transazione va eseguita per intero o per niente (dettagli tra poco)

Transazioni in JDBC

- Scelta della modalità delle transazioni: un metodo definito nell'interfaccia **`Connection`**:

  `setAutoCommit(boolean autoCommit)`

- **`con.setAutoCommit(true)`**
  - (default) "autocommit": ogni operazione è una transazione

- **`con.setAutoCommit(false)`**
  - gestione delle transazioni da programma

    `con.commit()`

    `con.rollback()`
  - non c'è **`start transaction`**

Transazioni in SQL

- Una transazione inizia al primo comando SQL dopo la "connessione" alla base di dati oppure alla conclusione di una precedente transazione (lo standard indica anche un comando start transaction, non obbligatorio, e quindi non previsto in molti sistemi)
- Conclusione di una transazione
  - commit [work]: le operazioni specificate a partire dall'inizio della transazione vengono eseguite sulla base di dati
  - rollback [work]: si rinuncia all'esecuzione delle operazioni specificate dopo l'inizio della transazione
- Molti sistemi prevedono una modalità autocommit, in cui ogni operazione forma una transazione

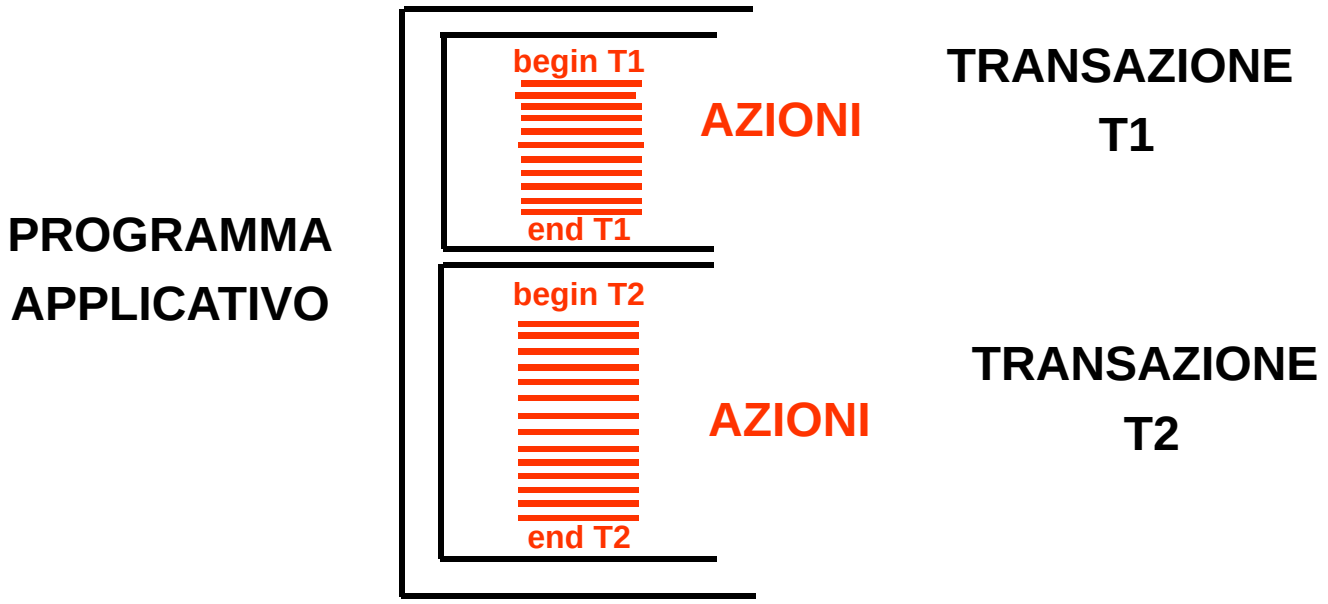
Una transazione

```
start transaction;                                    (opzionale)
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

```
start transaction;                                    (opzionale)
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;
```

DIFFERENZA FRA APPLICAZIONE E TRANSAZIONE



Il concetto di transazione

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

Atomicità

- Una transazione è una unità atomica di elaborazione:
  - tutto o niente
- Non può lasciare la base di dati in uno stato intermedio
  - un guasto o un errore prima del commit debbono causare l'annullamento (UNDO) delle operazioni eventualmente svolte
  - 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 è corretto
  - anche lo stato finale è corretto
- Se lo stato finale non è corretto, la transazione deve fallire

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 potrebbe ottenere con una esecuzione sequenziale
  - esempio:
    - due prelevamenti quasi contestuali
- Conseguenza: i risultati intermedi di una transazione non dovrebbero essere visibili
  - altrimenti si potrebbe generare un "effetto domino"
  - o addirittura utilizzare e comunicare all'esterno dati sbagliati

Durabilità (Persistenza)

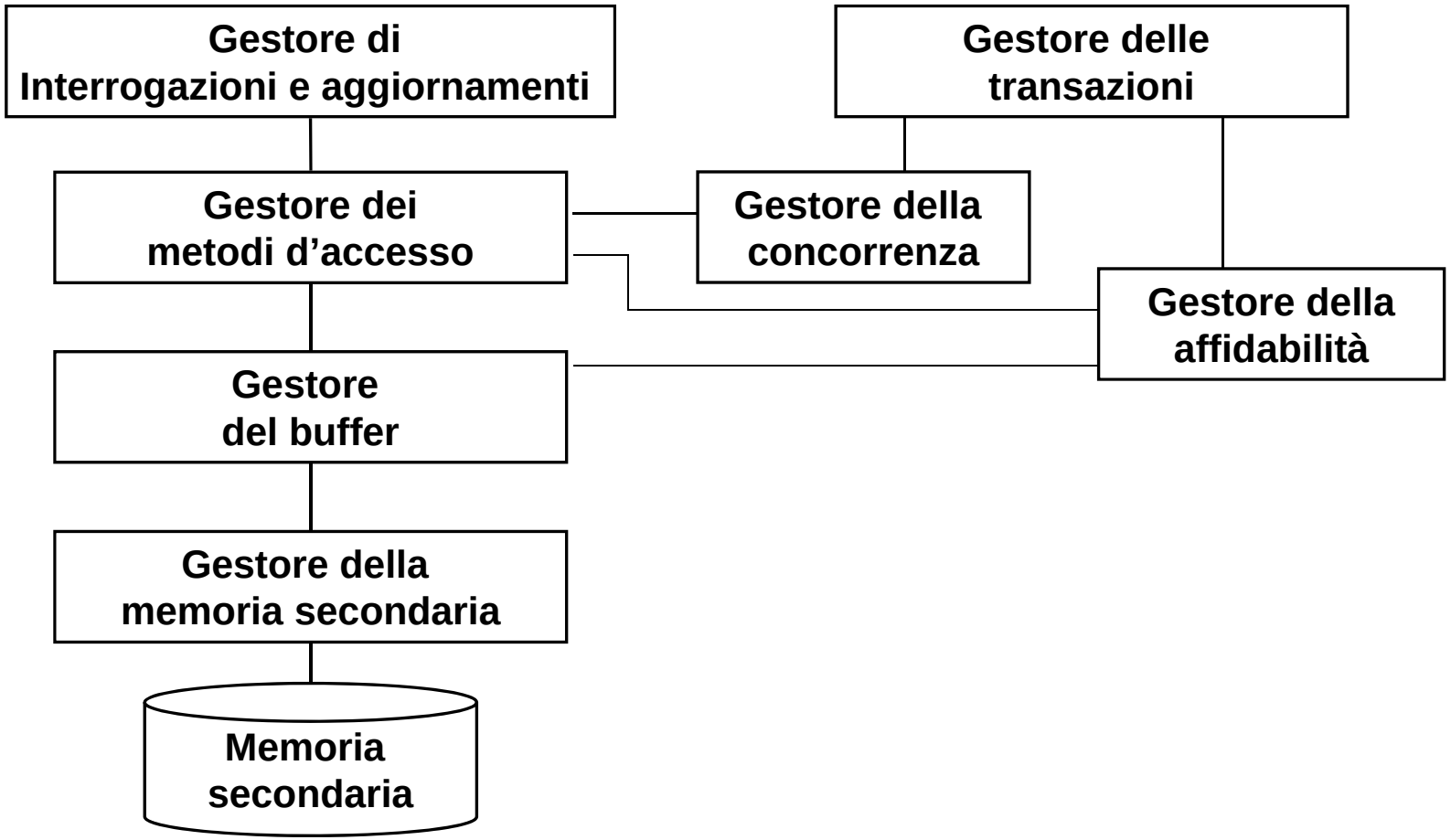
- Gli effetti di una transazione andata in commit non vanno perduti ("durano per sempre"), anche in presenza di guasti
  - "commit" significa "impegno"
  - attenzione al buffer

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 (con il supporto del compilatore del DDL)

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni



Persistenza delle memorie

- Memoria centrale: non è persistente
- Memoria di massa: è persistente ma può danneggiarsi
- Memoria stabile: memoria che non può danneggiarsi
  - astrazione “ideale,” non esiste, ma
  - perseguita attraverso la ridondanza:
    - dischi replicati
    - nastri
    - …

Guasti

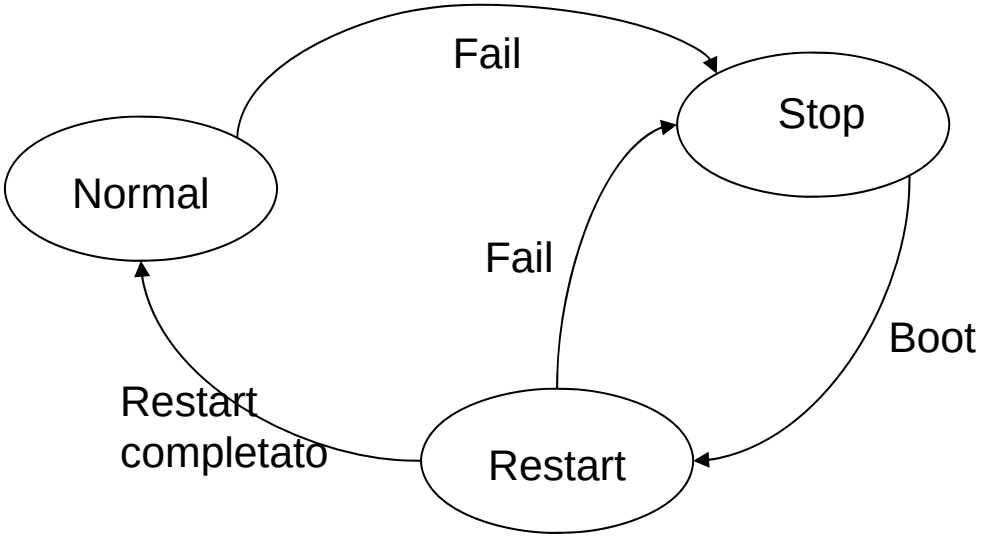
- **Guasti "soft":** errori di programma, crash di sistema, caduta di tensione
  - si perde la memoria centrale
  - non si perde la memoria secondaria

  warm restart (recovery), ripresa a caldo

- **Guasti "hard":** sui dispositivi di memoria secondaria
  - si perde anche la memoria secondaria
  - non si perde la memoria stabile

  cold restart, ripresa a freddo

Modello "fail-stop"

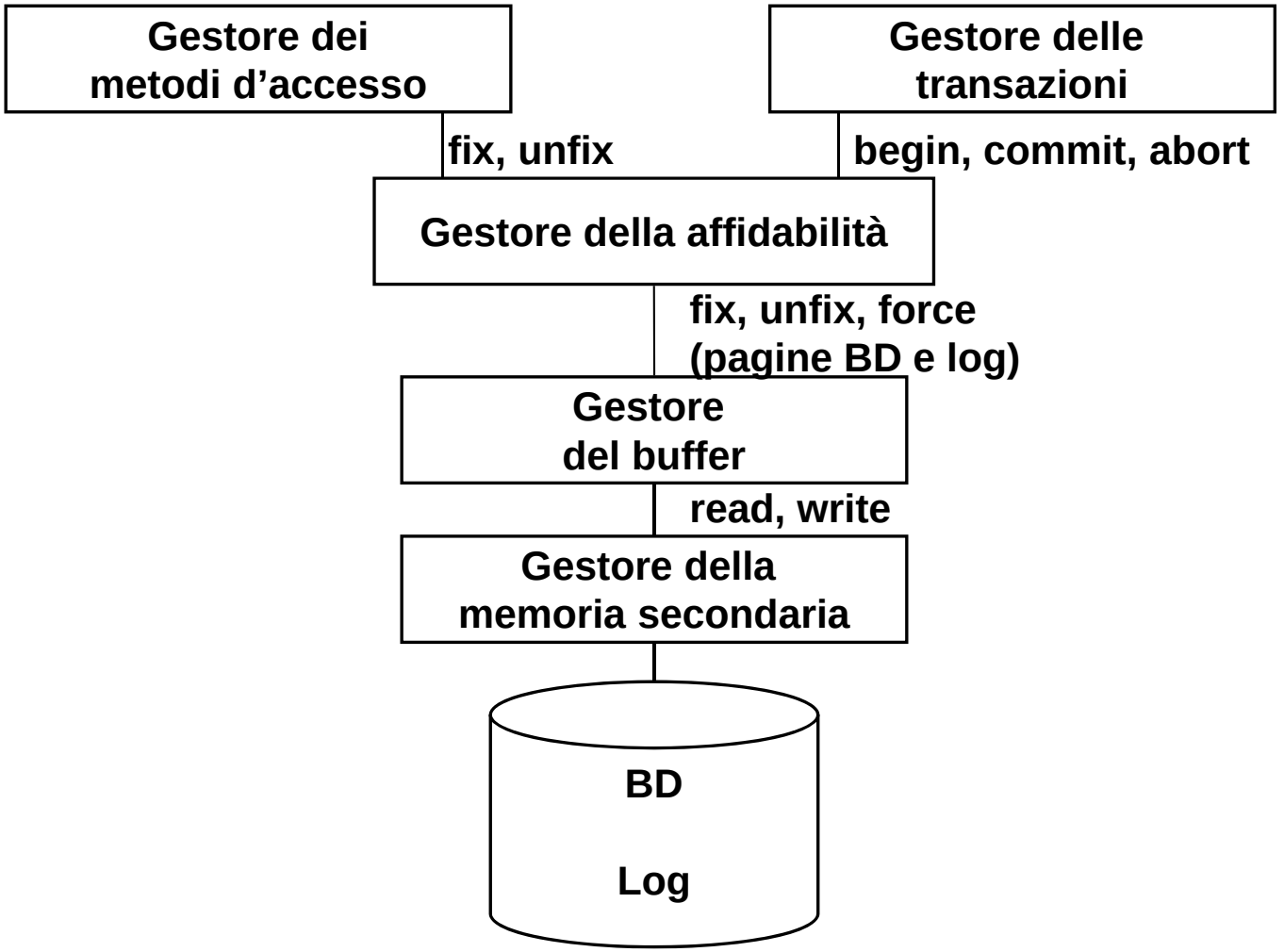


Gestore dell'affidabilità

- Assicura atomicità e durabilità
- 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*
- Usa il **log**:
  - Un archivio permanente che registra le operazioni svolte
  - Due metafore:
    - il filo di Arianna
    - i sassolini e le briciole di Hansel e Gretel
- e il **dump**
  - copia di riserva della base di dati

Architettura del controllore dell'affidabilità



Modello di riferimento

- Una transazione è costituita da una sequenza di operazioni di input-output su oggetti astratti x, y, z (ignoriamo la semantica delle applicazioni)

Il log

- Il log (o "giornale" o "diario di bordo") è un file sequenziale
  - riporta tutte le operazioni in ordine
  - scritto in memoria stabile, non sempre subito, ma sempre 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 (vediamo dopo)
    - dump
    - checkpoint

A che cosa serve il log?

- Introduce ridondanza, al fine di semplificare altre operazioni
- Permette di “ricostruire” le operazioni o anche di annullarle
- I sassolini di Hansel …

Esito di una transazione

- L'esito di una transazione è determinato irrevocabilmente quando viene scritto il record di **commit** nel log in modo sincrono, con una **force**
  - una guasto prima di tale istante può portare (se necessario) ad annullarne 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 (se non si è certi che sia stato ottenuto)
- record di abort possono essere scritti in modo asincrono

- Servono meccanismi per annullare e ricostruire

Undo e redo

- Undo di una azione su un oggetto O:
  - update, delete: copia il valore del before state (BS) nell'oggetto O
  - insert: eliminare O (se c'è)
- Redo di una azione su un oggetto O:
  - insert, update: copia il valore dell' after state (AS) nell'oggetto O
  - delete: elimina O (se c'è)

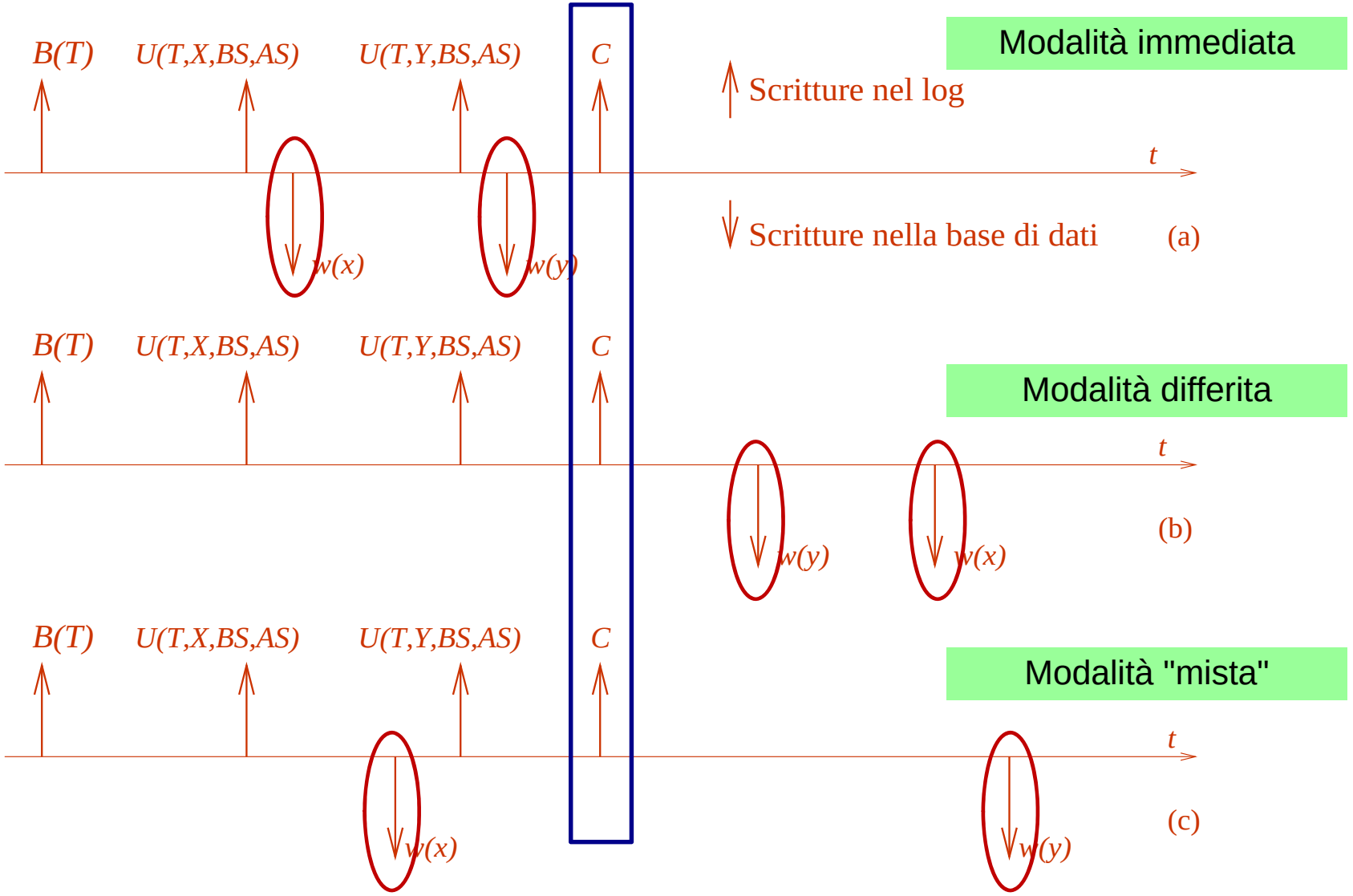
- Idempotenza di undo e redo (vedremo fra poco l'importanza)
  - undo(undo(A)) = undo(A)
  - redo(redo(A)) = redo(A)

Regole fondamentali per il log

- **Write-Ahead-Log**:
  - si scrive sul log (il BS) prima che nella base di dati
    - consente di disfare le azioni
- **Commit-Precedenza**:
  - si scrive sul log (l'AS) prima del commit (implementato garantendo che se un record del log è su disco allora ci sono tutte le precedenti)
    - consente di rifare le azioni

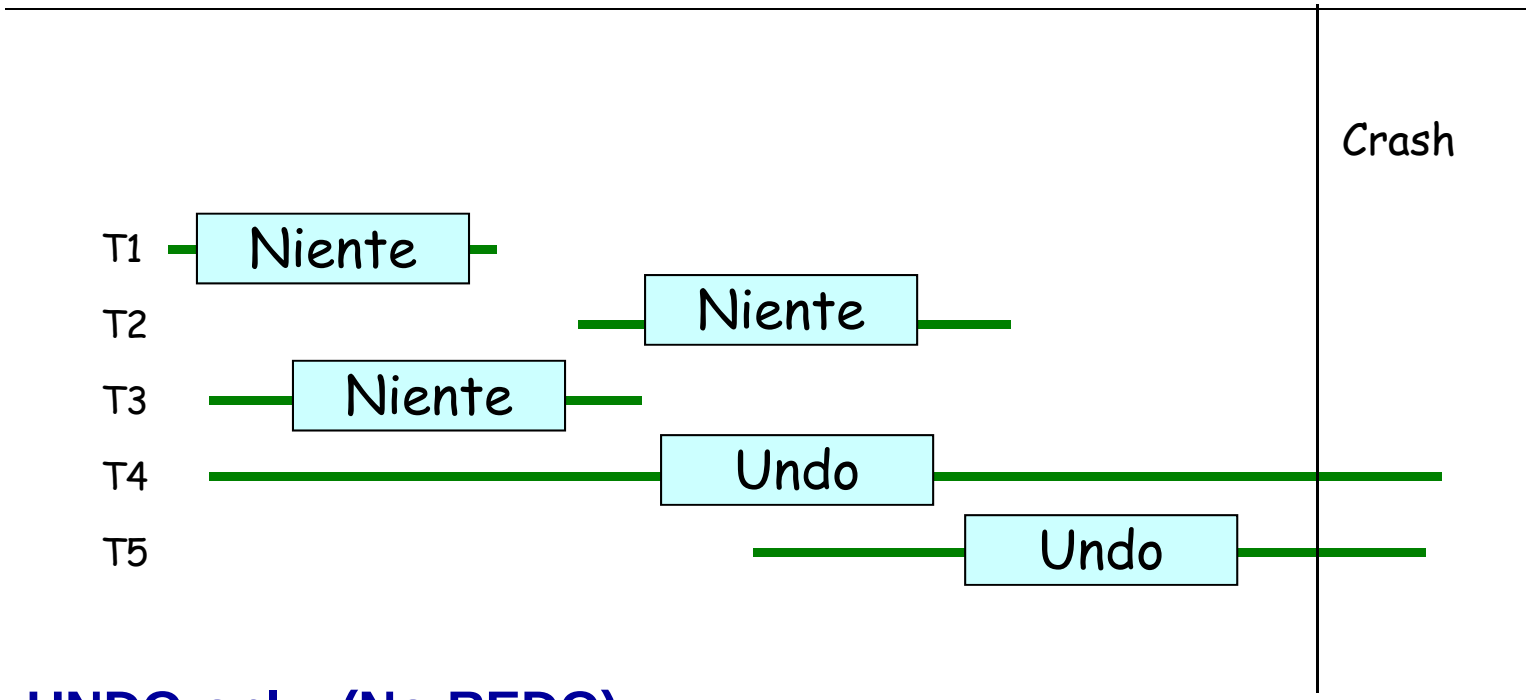
- Quando scriviamo nella base di dati?
  - Varie alternative

Scrittura nel log e nella base di dati



Modalità immediata

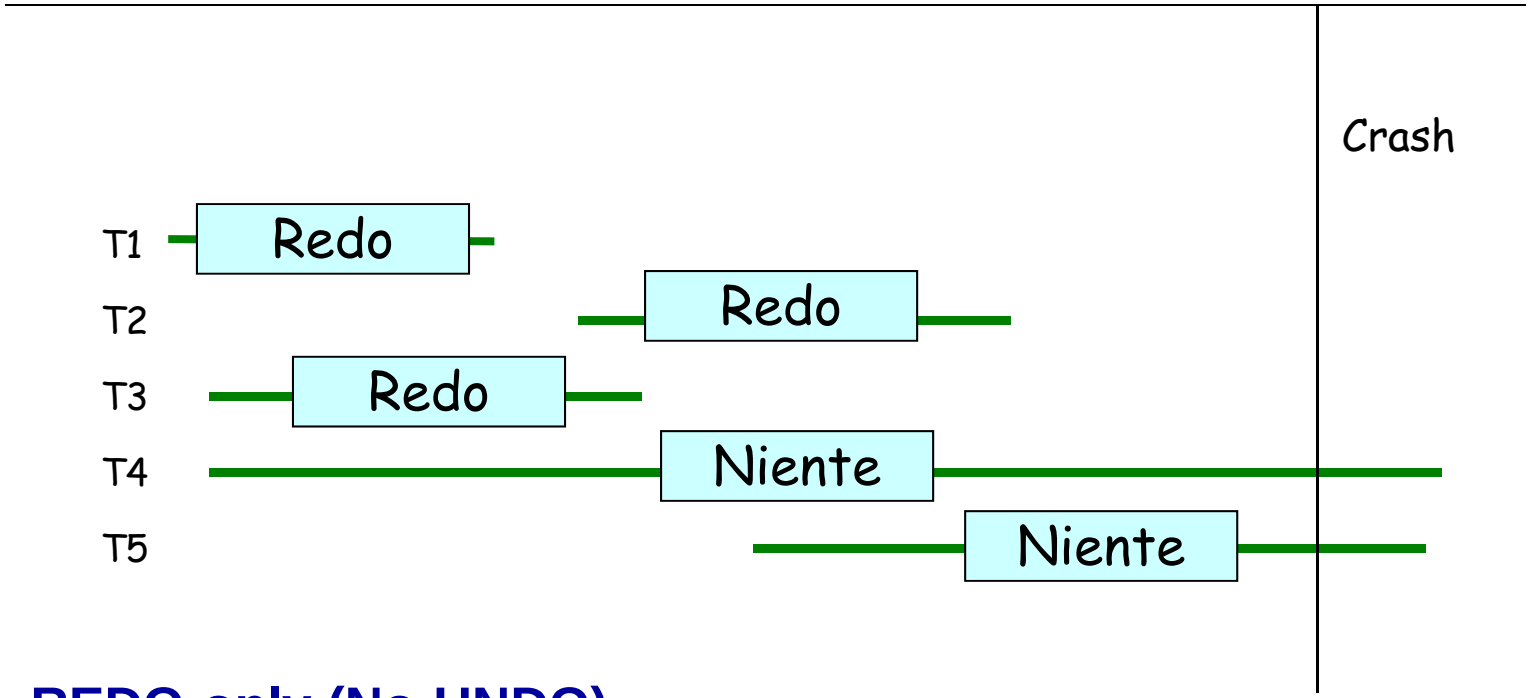
- Il DB contiene valori modificati per transazioni committed (tutte) e uncommitted



- **UNDO-only (No-REDO)**

Modalita’ differita

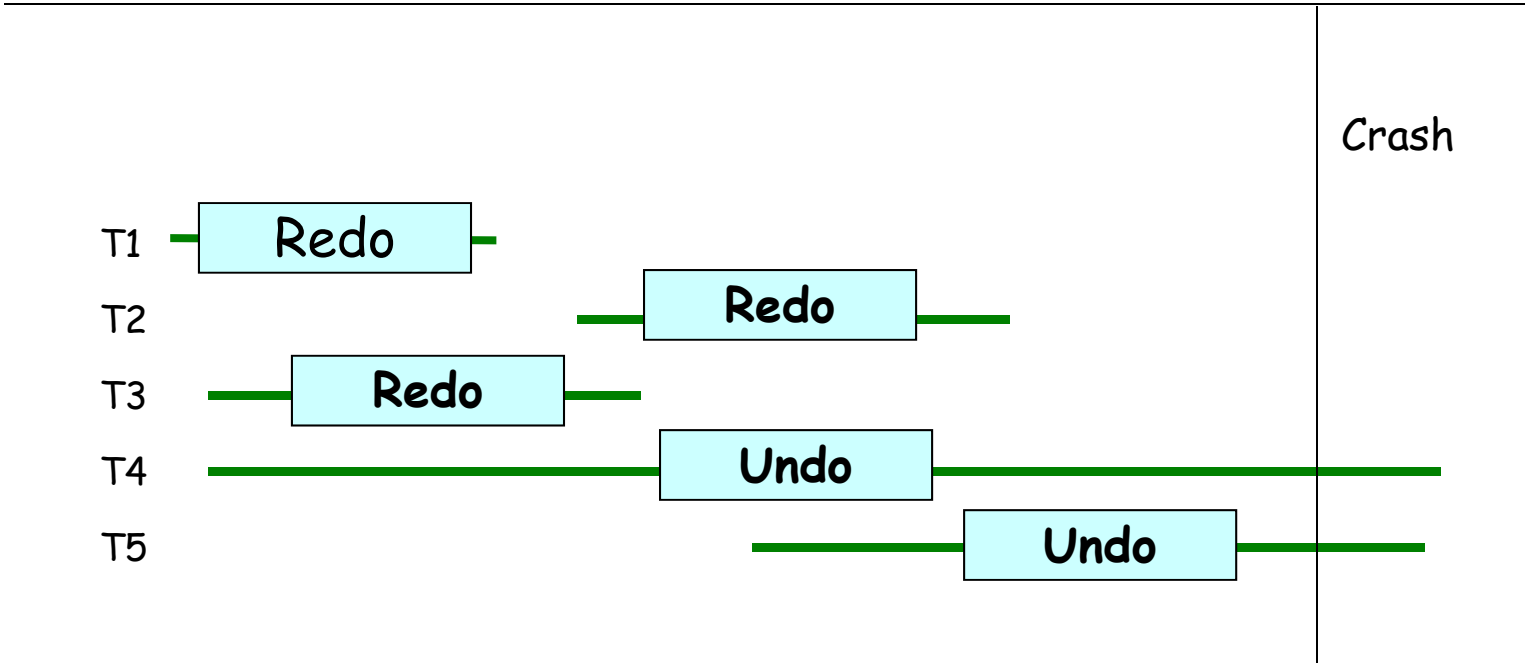
- Il DB non contiene valori modificati da transazioni uncommitted, ma non è sicuro contenga i nuovi valori per quelle committed



- **REDO-only (No-UNDO)**

Esiste una terza modalita': modalità mista

- La scrittura puo' avvenire in modalita' immediata e/o differita



- **REDO - UNDO**

Quale conviene?

- La terza, perché
  - anche se richiede operazioni diverse in caso di guasto,
  - permette al gestore del buffer di decidere quando scrivere

Rollback

- Un semplice uso del log (non discusso sul libro, per lasciare spazio ad altri concetti, ma utile per capire meccanismi semplici):
  - annullamento di una transazione (che non sia arrivata al commit): rollback

Rollback di una transazione

- Scandire il log dalla fine fino al record di begin della transazione T da annullare
  - Per ogni record che si riferisce ad una azione della transazione T, eseguire l'undo
- Aggiungere in coda un record di rollback

- Note
  - Nel log non ci può essere il record di commit di T
  - Perché si procede a ritroso?
    - Le transazioni da annullare sono di solito nella parte finale
    - Se ci sono più azioni sullo stesso oggetto, alla fine si ottiene il valore iniziale

Recovery (ripristino, warm restart)

- Scopo fondamentale del gestore dell'affidabilità:
  - di garantire che il contenuto della base di dati sia corretto. Ad esempio, allo shut down, lo stato della base di dati non deve essere stato prodotto da transazioni lasciate a metà e tutte quelle andate in commit debbono essere state completate
- Poiché la conclusione non è sempre "controllabile" (la chiusura potrebbe essere stata un crash), si può immaginare che all'avvio venga sempre effettuata una verifica che controlla. Come? Un modo ingenuo e inefficiente:
  - Per ogni transazione
    - se è andata in commit (cioè se il commit è sul log), facciamo il redo di tutte le azioni (assicurandoci che i dati siano scritti su disco)
    - se è stata annullata (rollback) non facciamo niente
    - altrimenti, facciamo l'undo di tutte le azioni (su disco)

Recovery, algoritmo

0. Individuiamo le transazioni da disfare e rifare (insiemi UNDO e REDO): percorriamo il log all'indietro, se troviamo un commit la transazione è da inserire in REDO, se per una transazione troviamo azioni senza commit né rollback, allora va in UNDO
1. A ritroso, UNDO di tutte le azioni delle transazioni in UNDO
2. In avanti, REDO di tutte le azioni delle transazioni in REDO

- In effetti il passo 0 non è necessario (si può fare insieme a 1)

1. Per ogni record nel log (dalla fine all'inizio)
   - Se è commit, aggiungi la transazione a REDO
   - Se è rollback, aggiungi la transazione a RB
   - Se è un'azione e la transazione non è REDO né RB, disfa l'azione
2. Per ogni record del log (dall'inizio alla fine)
   - Se è un'azione di una transazione in REDO, allora rifai l'azione

Nota: apparenti incoerenze fra le due fasi, causate da transazioni diverse che operano sullo stesso dato sono evitate dal controllo di concorrenza, che vedremo dopo

Quanto "costa" il recovery?

- Richiede
  - la lettura di tutto il log (due volte)
  - la ripetizione e l'annullamento di moltissime operazioni
- Come si può rendere più efficiente?
  - evitando di andare troppo indietro, se siamo sicuri che non ci sono più buffer non salvati

Riduciamo i costi del recovery

- Periodicamente, chiudiamo le transazioni in corso e, per tutte quelle concluse, salviamo i dati su disco:
  - Checkpoint inattivo ("quiescent checkpoint")
- Paragone (estremo):
  - la "chiusura dei conti" di fine anno di una amministrazione:
    - da fine novembre (ad esempio) non si accettano nuove richieste di "operazioni" e si concludono tutte quelle avviate prima di accettarne di nuove (a metà gennaio)

Quiescent checkpoint

- Si interrompe l'accettazione di nuove transazioni
- Si aspetta la conclusione delle transazioni attive
- Si forza la scrittura su disco di tutte le pagine sporche nel buffer
- Si inserisce un record di checkpoint nel log e si forza la sua scrittura su disco
- Si riprende l'accettazione di nuove transazioni

Recovery, con quiescent checkpoint

1. Per ogni record nel log (**dall'ultimo record fino al primo record di checkpoint incontrato procedendo a ritroso**)
   - Se è commit, aggiungi la transazione a REDO
   - Se è rollback, aggiungi la transazione a RB
   - Se è un'azione e la transazione non è REDO né RB, disfa l'azione
2. Per ogni record del log (dal checkpoint alla fine, procedendo in avanti)
   - Se è un'azione di una transazione in REDO, allora rifai l'azione

Nota: apparenti incoerenze fra le due fasi, causate da transazioni diverse che operano sullo stesso dato, sono evitate dal controllo di concorrenza, che vedremo dopo

Quiescent checkpoint, commenti

- Il recovery non deve tornare fino all'inizio del log, ma si può fermare all'ultimo checkpoint (il primo incontrato partendo dalla fine)
- Il checkpoint va fatto abbastanza spesso, per ridurre la lunghezza delle operazioni di ripristino
  - in particolare, va fatto allo startup, per garantire che la base di dati sia corretta (non si sa come sia stata chiusa la sessione precedente)
  - facendolo anche allo shutdown, si semplifica il recovery allo startup
- Può comunque introdurre inefficienze, perché si interrompe l'accettazione di nuove transazioni:
  - ce ne sono altre versioni

Nonquiescent checkpoint

- "Fa 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)

Nonquiescent checkpoint

- Varie modalità, vediamo la più semplice:
  - si interrompe l'accettazione di richieste di ogni tipo (nuove transazioni, scrittura, inserimenti, …, commit, abort)
  - si rilevano le transazioni in corso
  - si forza la scrittura su disco di tutte le pagine sporche
  - si inserisce un record di checkpoint (con gli identificatori delle transazioni) nel log e si forza la sua scrittura su disco
  - si riprende l'accettazione di nuove richieste

Recovery, con nonquiescent checkpoint

1. Per ogni record nel log (dall'ultimo record fino al primo record di checkpoint incontrato procedendo a ritroso)
   - Se è commit, aggiungi la transazione a REDO
   - Se è rollback, aggiungi la transazione a RB
   - Se è un'azione e la transazione non è REDO né RB, disfa l'azione

1 bis. Per ogni record nel log (dal checkpoint fino allo start della più vecchia transazione elencata nel checkpoint, ancora a ritroso),
   - Se la transazione è elencata nel checkpoint, procedere come al punto 1

2. Per ogni record del log (dal punto in cui è arrivato 1bis alla fine, procedendo in avanti; se il checkpoint salva tutti i buffer sporchi, allora basta ripartire dal checkpoint)
   - Se è un'azione di una transazione in REDO, allora rifai l'azione

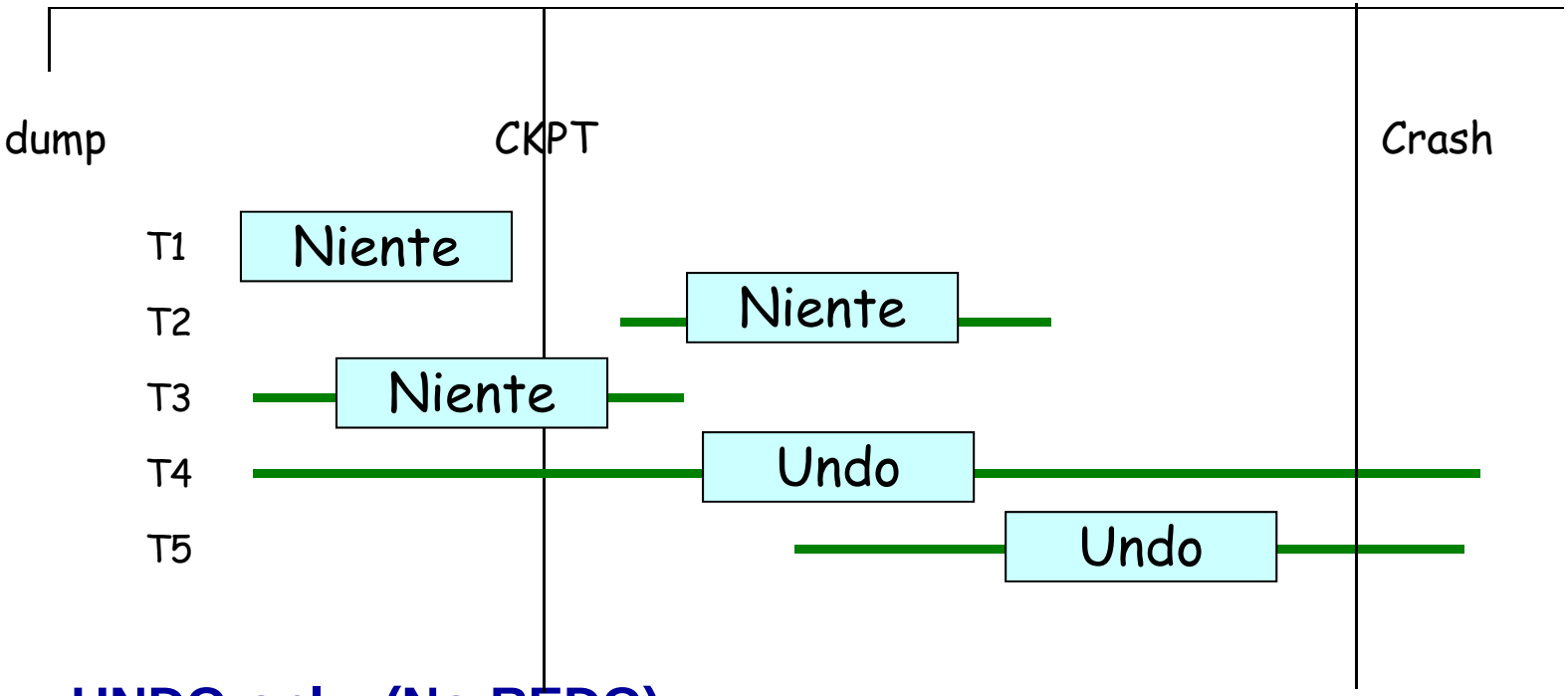
Nota: apparenti incoerenze fra le due fasi, causate da transazioni diverse che operano sullo stesso dato sono evitate dal controllo di concorrenza, che vedremo dopo

Checkpoint e scrittura nella base di dati

- Come prima, solo si ignorano tutte le transazioni completate prima del checkpoint

Modalità immediata

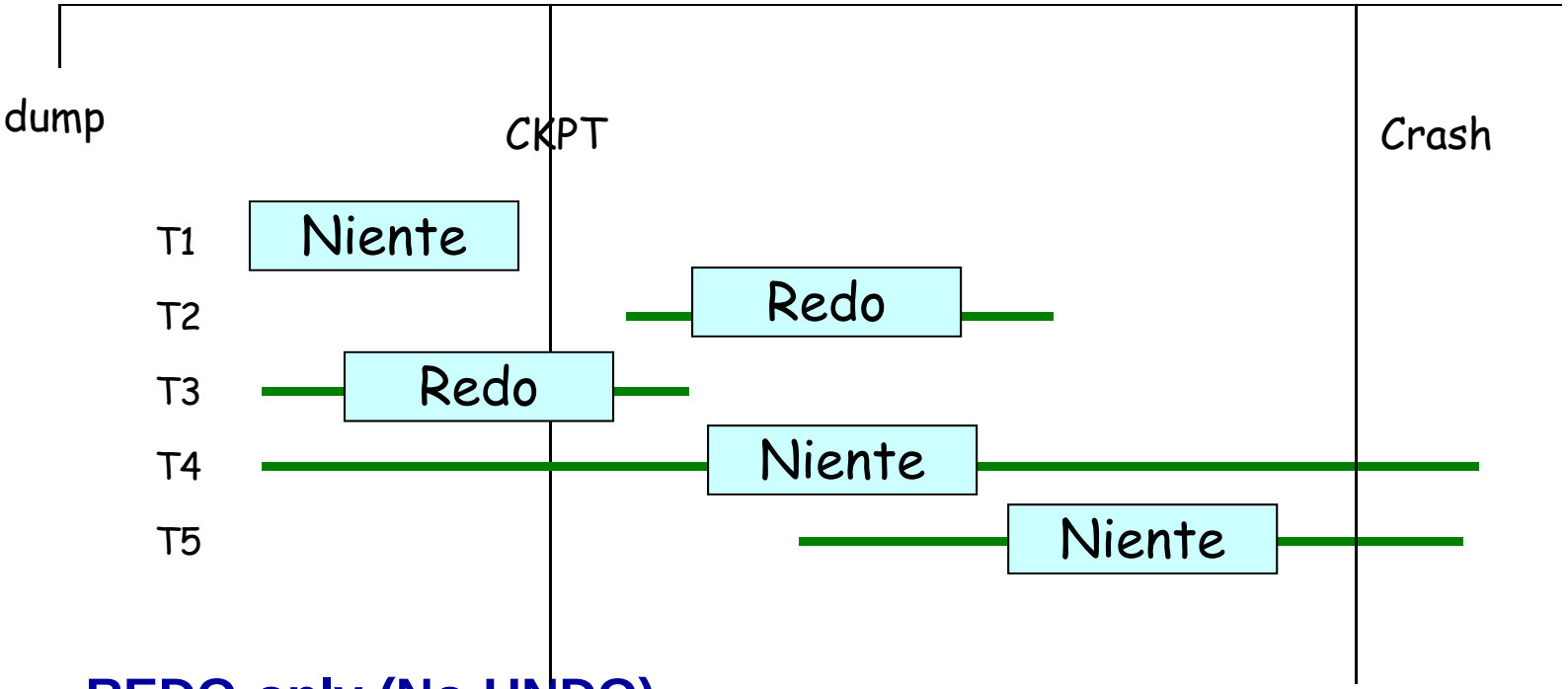
- Il DB contiene valori modificati per transazioni committed (tutte) e uncommitted



- **UNDO-only (No-REDO)**

Modalita’ differita

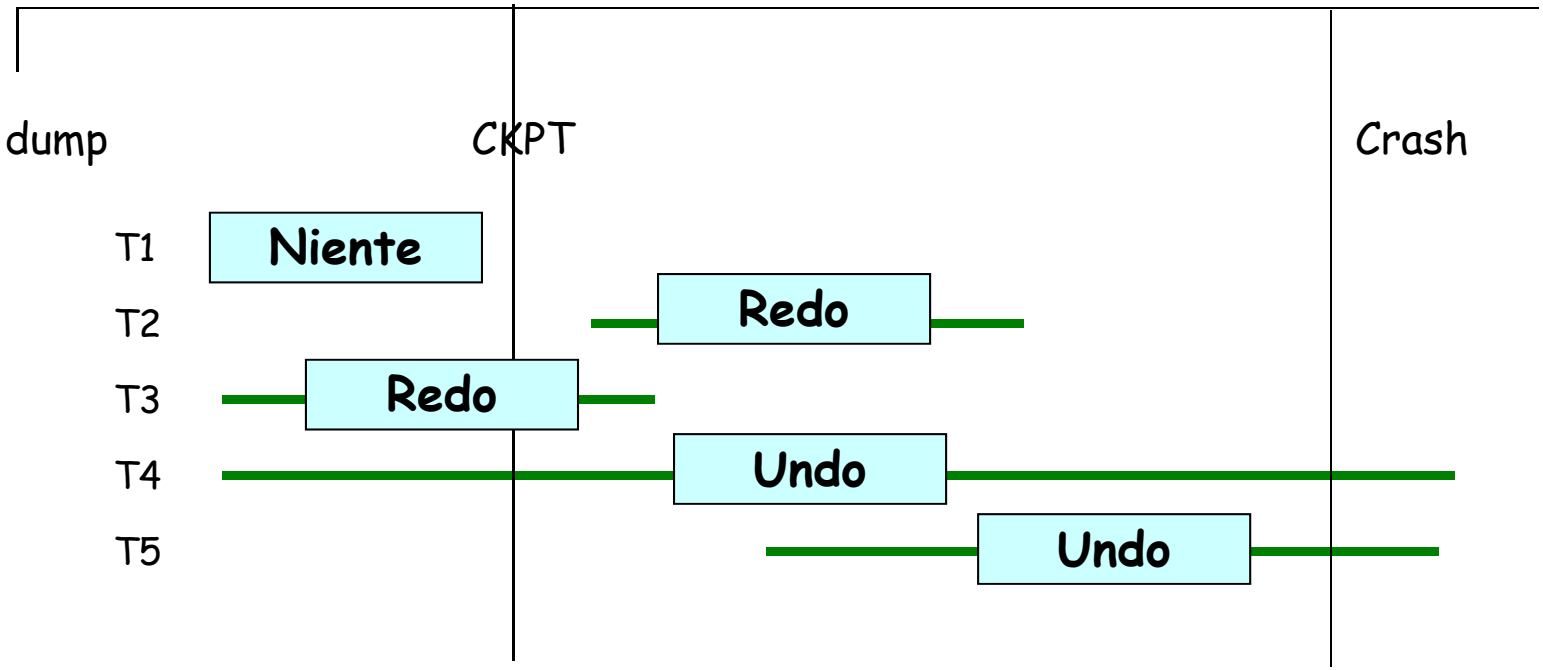
- Il DB non contiene valori modificati da transazioni uncommitted, ma non è sicuro contenga i nuovi valori per quelle committed



- **REDO-only (No-UNDO)**

Modalità mista

- La scrittura puo' avvenire in modalita' sia immediata che differita



- **REDO - UNDO**

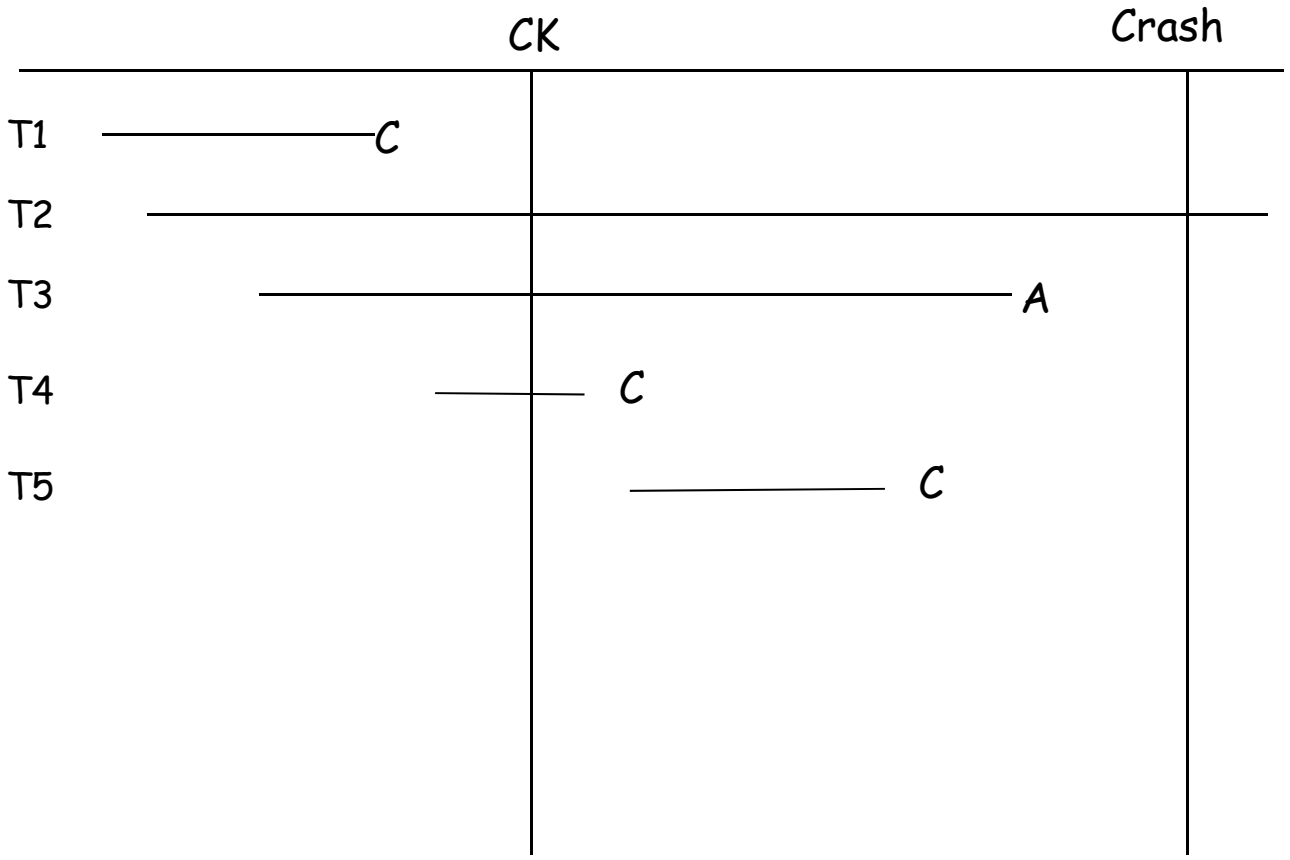
Ripresa a caldo (come sul libro)

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*

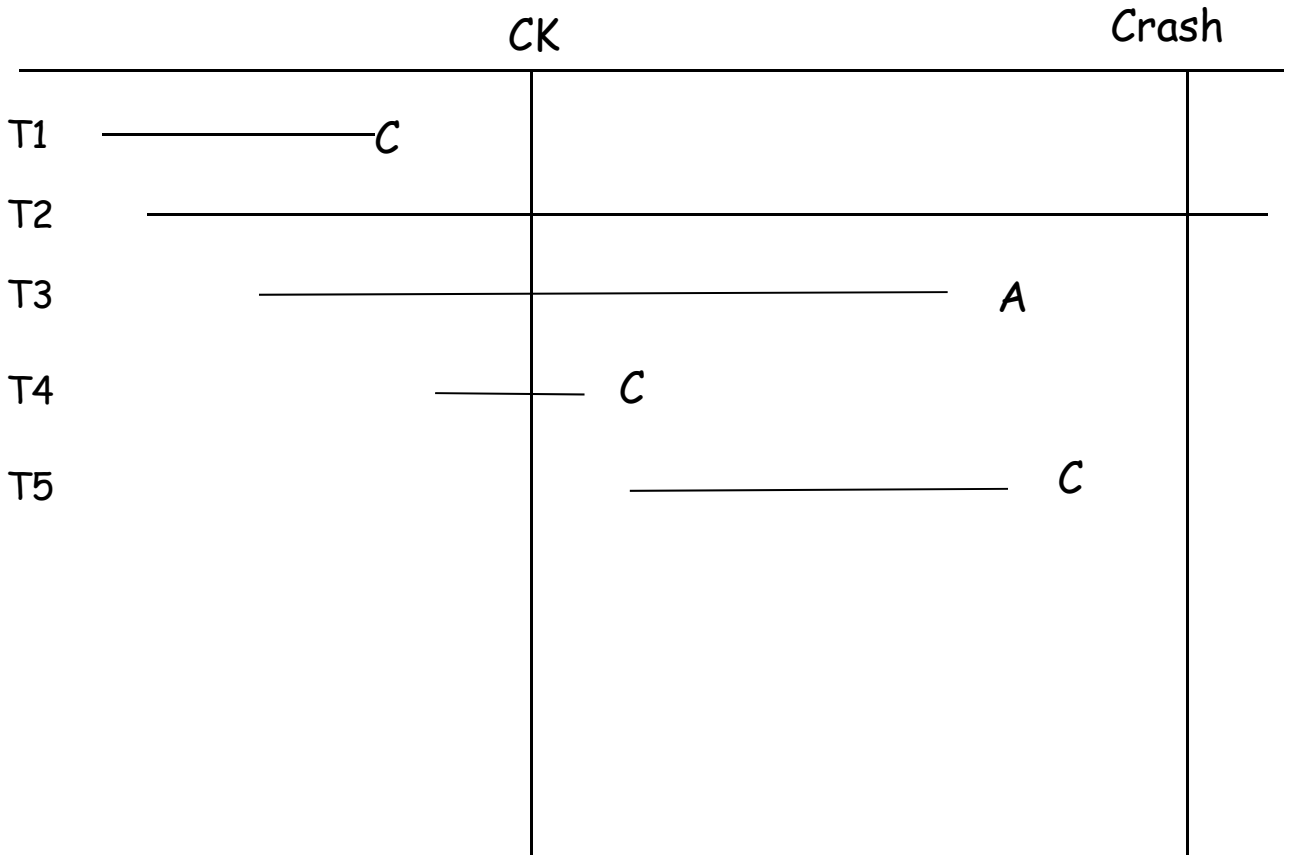
Esempio di warm restart

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)



2. Costruzione degli insiemi UNDO e REDO

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)

1. C(T4)
2. B(T5)
   U(T3,O3,B5,A5)
   U(T5,O4,B6,A6)
   D(T3,O5,B7)
   A(T3)
3. C(T5)
   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}
3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Setup

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)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
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 Undo

7. O2 =B3

8. O1=B1

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

Undo

7. O2 =B3

8. O1=B1

---

9. O3 = A4

Redo

10. O4 = A6

Guasti

- **Guasti "soft":** errori di programma, crash di sistema, caduta di tensione
  - si perde la memoria centrale
  - non si perde la memoria secondaria

  warm restart (recovery), ripresa a caldo

- **Guasti "hard":** sui dispositivi di memoria secondaria
  - si perde anche la memoria secondaria
  - non si perde la memoria stabile

  cold restart, ripresa a freddo

Dump

- Copia completa (“di riserva,” backup) della base di dati
  - Solitamente prodotta mentre il sistema non è operativo
  - Salvato in memoria stabile
  - Nel log viene inserito un record di dump che indica il momento in cui il dump è stato effettuato (e dettagli pratici, file, dispositivo, …)
  - Il dump viene eseguito molto più di rado rispetto al checkpoint

Ripresa a freddo

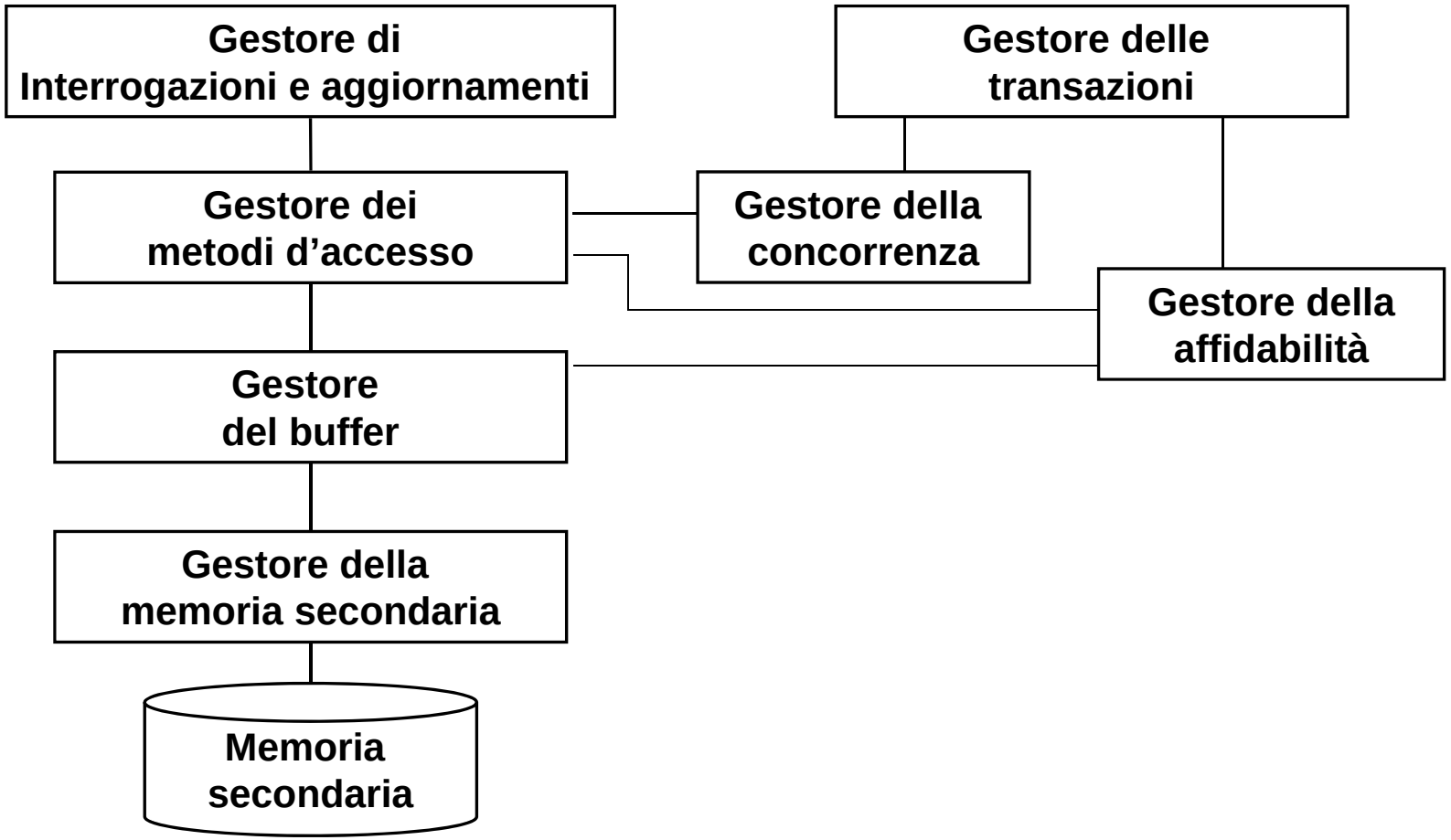
- Si ripristinano i dati a partire dal backup
- Si eseguono le operazioni registrate sul log fino all'istante del guasto
- Si esegue una ripresa a caldo

Backup e recovery

- Le tecniche per la gestione dell'affidabilità permettono anche
  - Ripristino di versioni salvate (dump)
    - In DB2
      - "version recovery"
  - Ripristino dello stato ad un certo istante
    - In DB2
      - "roll forward recovery"

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni



Controllo di concorrenza

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

Problema

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

Due prelevamenti da un conto corrente

- Quanto c'è sul conto?
  - 1000

- Quanto c'è sul conto?
  - 1000

- Bene, allora prelevo 200
  - Ok, il nuovo saldo è 800

- Bene, allora prelevo 100
  - Ok, il nuovo saldo è 900

Perdita di aggiornamento (lost update)

  - t_1 : r(x), $x = x$ -200 , w(x)
  - t_2 : r(x), $x = x$ -100, w(x)
- Inizialmente x=1000; dopo un'esecuzione seriale x=700
- Un'esecuzione concorrente:

t_1	t_2
$r_1(x)$	
$x = x$ -200	
	$r_2(x)$
	$x = x - 100$
$w_1(x)$	
`commit`	
	$w_2(x)$
	`commit`

- Un aggiornamento viene perso: alla fine x=900
- Le azioni delle transazioni si alternano (Paperina, Paperino, Paperina, …) in modo in accettabile
- Se eseguissimo in modo seriale prima Paperina e poi Paperino, allora ok

Lettura prima della conferma (commit)

1000 sul conto

- Ecco un assegno da 10.000
  - Bene, il nuovo saldo è 11.000

- Quanto c'è sul conto?
  - 11.000
- Abbiamo 11.000 !?!

  - Ma l'assegno è falso!
  - Annulliamo l'operazione
  - Il saldo resta 1000

- Paperina ha letto un dato sbagliato!

Lettura sporca (dirty read)

t_1	t_2
$r_1(x)$	
$x = x + 11000$	
$w_1(x)$	
	$r_2(x)$
`abort`	

- t_2 ha letto uno stato intermedio ("sporco") e lo può comunicare all'esterno

Lettura, scrittura, lettura

1000 sul conto A
1000 sul conto B

- Quanto c'è sul conto A?
  - 1000

- Sposta 500 da A a B
  - Nuovo saldo A: 500
  - Nuovo saldo B: 1500

- Quanto c'è sul conto B?
  - 1500
- Ehi, abbiamo 2500!

Aggiornamento fantasma (ghost update), lettura inconsistente (con la base di dati)

- Supponiamo che y=1000 e z= 1000
- quindi $y + z = 2000$

t_1	t_2
r_1(y)	
	$r_2(y)$
	$y = y - 500$
	$r_2(z)$
	$z = z + 500$
	$w_2(y)$
	$w_2(z)$
	`commit`
$r_1(z)$	
$s = y + z$	

- s = 2500: ma la somma $y + z$ non è mai stata “davvero” 2500:
  - t_1 vede uno stato non esistente (o non coerente)

Letture inconsistenti (inconsistent read), caso ancora più semplice

- t_1 legge due volte x :

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

- t_1 legge due valori diversi per x !
- Esempio: "quanti posti disponibili" ?

Inserimento fantasma (phantom)

- Esempio: sistema di prenotazione

t_1	t_2
"conta i posti disponibili"	
	"aggiungi nuovi posti" (aereo più grande)
	`commit`
"conta i posti disponibili"	

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"**

Livelli di isolamento in SQL:1999 (e JDBC)

- Le transazioni possono essere definite **`read-only`** (non possono scrivere)
- 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 dovrebbe essere sempre evitata, ma non è così: conviene usare sempre **`serializable`** per le transazioni che scrivono

Anomalie

Perdita di aggiornamento	W-W
• `read uncommitted`	
Lettura sporca	R-W (o W-W) con abort
• `read committed`	
Letture inconsistenti	R-W
Aggiornamento fantasma	R-W
• `repeatable read`	
Inserimento fantasma	R-W su dato "nuovo"
• `serializable`	

Livelli di isolamento, perché?

- La gestione del controllo della concorrenza è costosa, se non serve, possiamo rinunciarci
- Nota bene: per le letture, non per le scritture

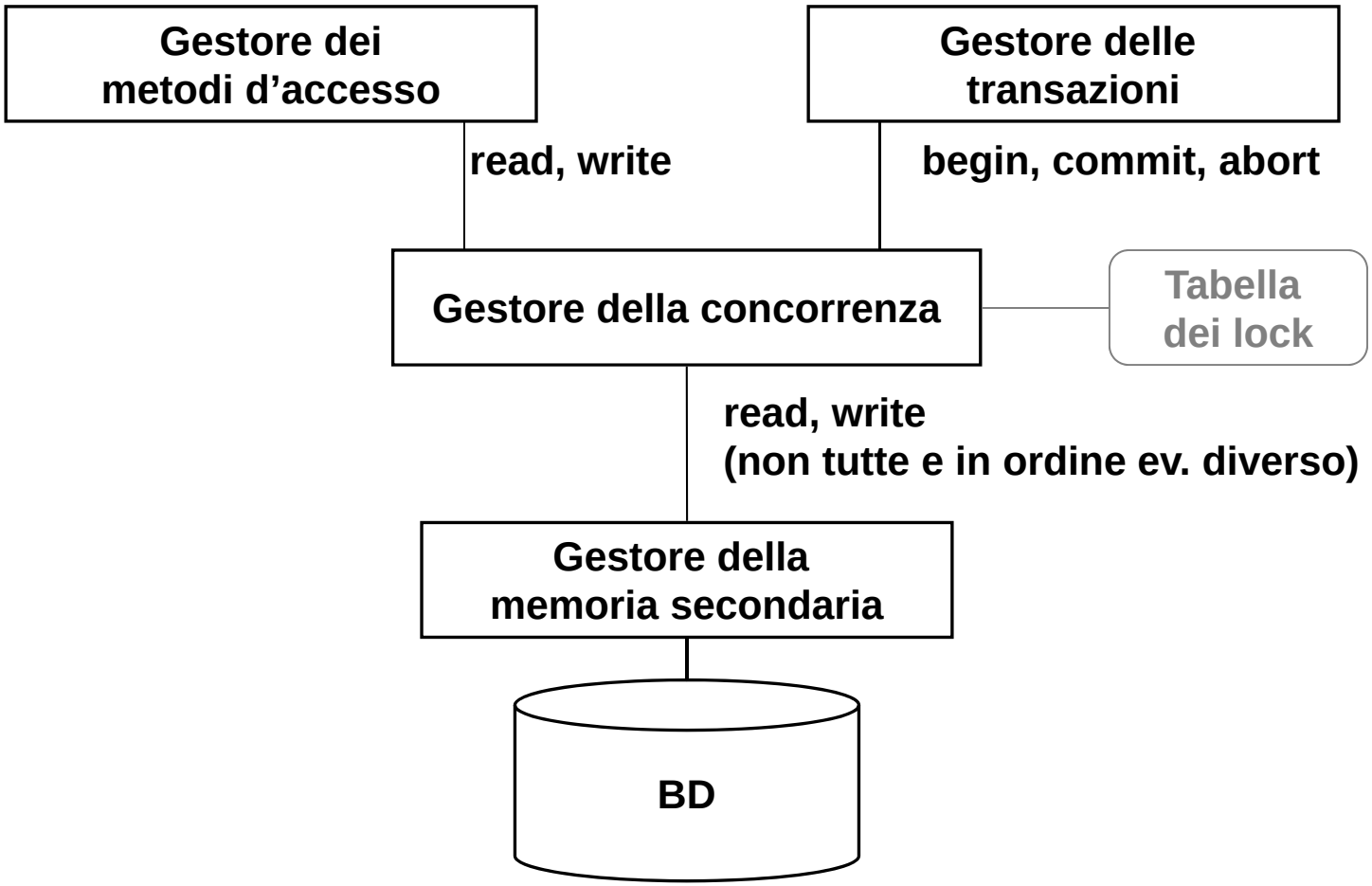
Anomalie

Perdita di aggiornamento	**W-W**
• `read uncommitted`	
Lettura sporca	**R-W (o W-W) con abort**
• `read committed`	
Letture inconsistenti	**R-W**
Aggiornamento fantasma	**R-W**
• `repeatable read`	
Inserimento fantasma	**R-W su dato "nuovo"**
• `serializable`	

Livelli di isolamento, esempi

- Vedi compito d'esame del 15/06/2005 domanda 4

Gestore della concorrenza (ignorando buffer e affidabilità)



“Teoria” del controllo di concorrenza

- Transazione: sequenza di letture e scritture (senza ulteriore semantica)
- Notazione:
  - ogni transazione ha un identificatore univoco
- Esempio:
  - $t_1 : r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$

Schedule

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

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

- Ipotesi semplificativa preliminare (non accettabile in pratica, la rimuoveremo):
  - ignoriamo le transazioni che vanno in abort (e quindi non scriviamo commit e abort)
  - quindi, consideriamo la **commit-proiezione** degli schedule reali, in quanto ignoriamo le transazioni che vanno in abort, rimuovendo tutte le loro azioni dallo schedule

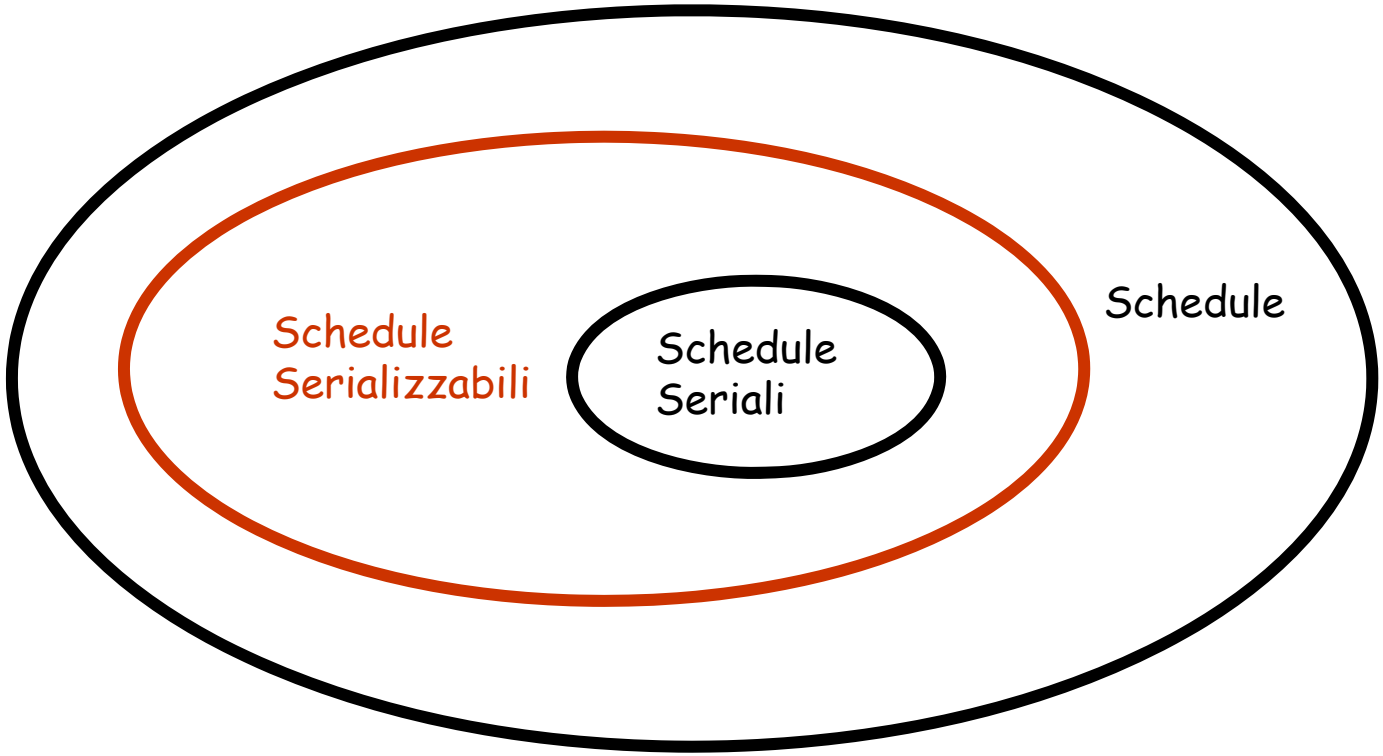
Controllo di concorrenza

- *Obiettivo*: evitare le anomalie
- *Schedule seriale:*
  - le transazioni sono separate, una alla volta

    $S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$
  - uno schedule seriale non presenta anomalie
- *Schedule serializzabile:*
  - produce lo “stesso risultato” di uno schedule seriale sulle stesse transazioni e quindi è accettabile
    - richiede una nozione di equivalenza fra schedule
- *Controllore di concorrenza (Scheduler)*:
  - un sistema che accetta o rifiuta (o riordina) le operazioni richieste dalle transazioni (mantenendo l'ordine delle azioni in ciascuna transazione)
  - deve ammettere solo schedule serializzabili

Idea base

- Individuare classi di schedule serializzabili per i quali la 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 $w_j(x)$ e $r_i(x)$ 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*** ($S_i \approx_V S_j$):
  - le operazioni di ciascuna transazione mantengono l'ordine
  - stessa relazione *legge-da*
  - 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_2 : $r_2(x)$ $w_0(x)$ $r_1(x)$ $w_2(x)$ $w_2(z)$

S_3 : $w_0(x)$ $r_2(x)$ $r_1(x)$ $w_2(x)$ $w_2(z)$

- S_2 e S_3 non sono view-equivalenti

S_4 : $w_0(x)$ $r_1(x)$ $r_2(x)$ $w_2(x)$ $w_2(z)$

- S_4 è seriale
- S_3 è view-equivalente a S_4 (e quindi view-serializzabile)
- S_2 non è view-serializzabile

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_6 è seriale
- S_5 è view-equivalente a S_6

View serializzabilità: esempi (2)

- perdita di aggiornamento

  $S_7 : r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$
- letture inconsistenti

  $S_8 : r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$
- aggiornamento fantasma

  $S_9 : r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$

- S_7, $S_{8,}$ S_9 non sono view-serializzabili.

View serializzabilità

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

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 nei due schedule nel medesimo ordine
- Uno schedule è *conflict-serializable* se è conflict-equivalente ad un qualche schedule seriale
- L'insieme degli schedule conflict-serializzabili è indicato con **CSR**

CSR e VSR

- Ogni schedule conflict-serializzabile è view-serializzabile, ma non necessariamente viceversa
- Controesempio per la non necessità:

  $r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$

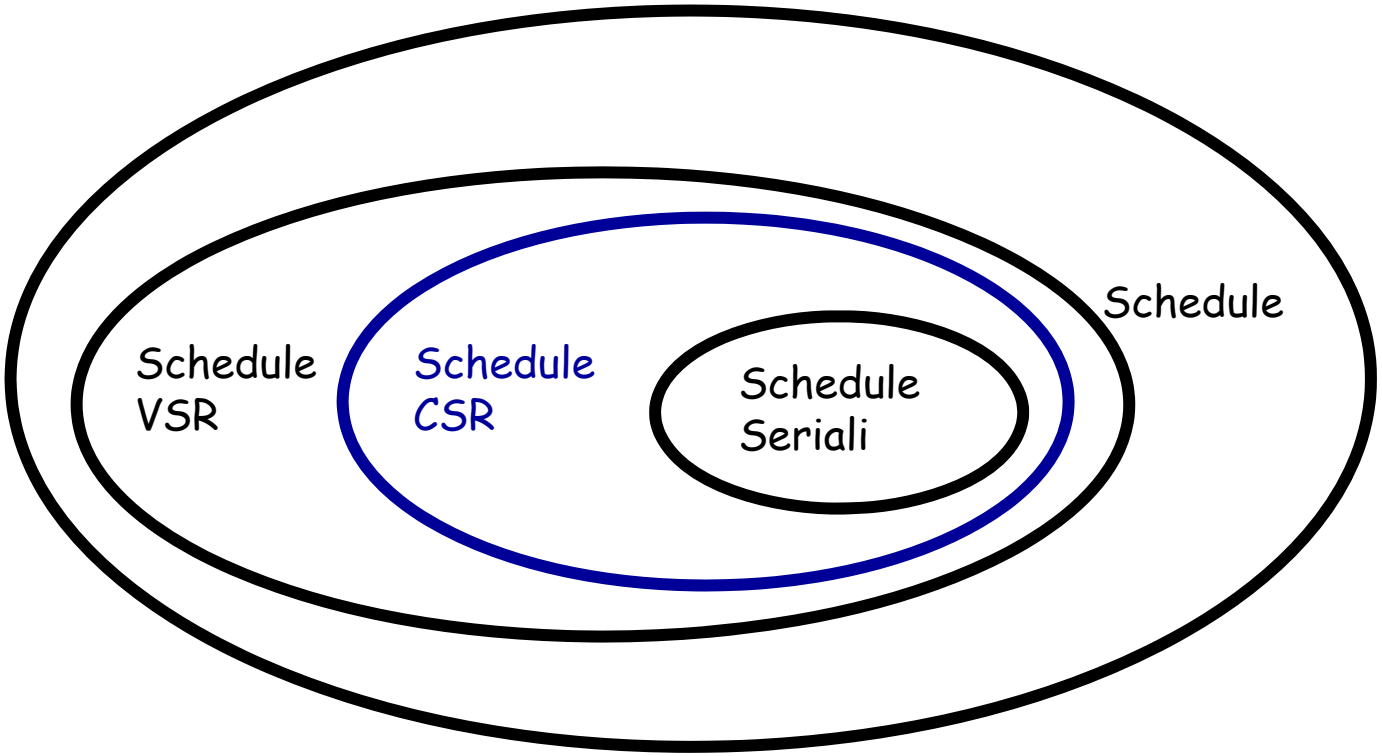
  - view-serializzabile:
    - view-equivalente a $r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$
  - non conflict-serializzabile

- Sufficienza: vediamo

CSR implica VSR

- CSR: esiste schedule seriale conflict-equivalente
- VSR: esiste schedule seriale view-equivalente
- Per dimostrare che CSR implica VSR è sufficiente dimostrare che la conflict-equivalenza $\approx_C$ implica la view-equivalenza $\approx_V$, cioè che se due schedule sono $\approx_C$ allora sono $\approx_V$
- Supponiamo $S_1 \approx_C S_2$ e dimostriamo che $S_1 \approx_V S_2$.
  - stesse scritture finali:
    - se così non fosse, ci sarebbero due scritture (su uno stesso oggetto) in ordine diverso e poiché due scritture (su uno stesso oggetto) sono in conflitto i due schedule non sarebbero $\approx_C$
  - stessa relazione “legge-da”:
    - se così non fosse, ci sarebbero scritture (su uno stesso oggetto) in ordine diverso o coppie lettura-scrittura (su …) in ordine diverso e quindi, come sopra, sarebbe violata la $\approx_C$

CSR e VSR



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

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.

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
- Lemma
  - **Due schedule conflict-equivalenti hanno lo stesso grafo dei conflitti**
- Teorema
  - **Uno schedule è in CSR se e solo se il grafo è aciclico**

Grafo dei conflitti e conflict-equivalenza

Lemma **Due schedule conflict-equivalenti hanno lo stesso grafo dei conflitti**

- **$S_1 \approx_C S_2$;** dimostriamo che **S_1e S_2 hanno lo stesso grafo.**
  - Per assurdo: supponiamo che S_1 e S_2 siano equivalenti e non abbiano lo stesso grafo.
    Allora nel grafo di uno dei due schedule c'è un arco non presente nell'altro; supponiamo che S_1 abbia un tale arco; esso è relativo a due azioni che in S_1 sono in conflitto;
    se l'arco non compare nel grafo di S_2, allora le due azioni sono in S_2 ordine diverso e quindi S_1 e S_2 non sono $\approx_C$: una contraddizione

CSR e aciclicità del grafo dei conflitti

Teorema **S è in CSR se e solo se il grafo di S è aciclico**

- **Se S è in CSR allora il grafo di S è aciclico**

  Se S è in CSR allora è $\approx_C$ ad uno schedule seriale S_0.

  Per il lemma, S e S_0 hanno lo stesso grafo

  Il grafo di S_0, che è seriale, è aciclico, perché ci possono essere conflitti fra azioni a_i e a_j solo se i<j (mentre un ciclo richiede almeno un arco (i,j) con i>j).

  Quindi i il grafo di S è aciclico.

- **Se il grafo di S è aciclico allora S è in CSR**

  Se il grafo 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.

  Quindi esiste uno schedule seriale equivalente ad S: S è in CSR

Controllo della concorrenza in pratica

- la conflict-serializabilità
  - è verificabile più rapidamente della view-serializzabilità (con opportune strutture dati, complessità lineare),
- però
  - sarebbe davvero 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
  - si basa sull'ipotesi di commit-proiezione

- è inutilizzabile in pratica

Controllo della concorrenza in pratica, 2

- In pratica, si utilizzano tecniche che
  - garantiscono la conflict-serializzabilità senza dover costruire il grafo
  - non richiedono l'ipotesi della commit-proiezione

- Le tecniche utilizzate
  - 2PL (two-phase locking)
  - timestamp (implementazione "multiversion")

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 upgrade*)
- Il *lock manager* riceve queste richieste dalle transazioni e le accoglie o rifiuta, sulla base della tavola dei conflitti

Gestione dei 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	-	OK / dipende	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 (spesso con un “time-out”)
- Il lock manager gestisce una tabella dei lock, per ricordare la situazione

I lock da soli non bastano

- La perdita di aggiornamento, con i lock

t_1	t_2
$rlock_1$ (x)	
$r_1(x)$	
$x = x - 1000$	
$unlock_1$ (x)	
	$rlock_2$ (x)
	$r_2(x)$
	$x = x - 1000$
	$unlock_2$ (x)
$wrlock_1$ (x)	
$w_1(x)$	
$unlock_1$ (x)	
`commit`	
	$wlock_2$ (x)
	$w_2(x)$
	$unlock_2$ (x)
	`commit`

- Non cambia niente, il problema resta

Locking a due fasi (2PL)

- Oltre alla protezione con lock di tutte le letture e scritture
  - vincolo sulle richieste e i rilasci dei lock:
    - **una transazione, dopo aver rilasciato un lock, non può acquisirne altri**
- Garantisce "a priori" la conflict-serializzabilità, vediamo

2PL e CSR

- **Ogni schedule 2PL e' anche conflict serializzabile, ma non necessariamente viceversa**
- Controesempio per la non necessità:

$$r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$$

  - Viola 2PL
  - Conflict-serializzabile

- Sufficienza: vediamo

2PL implica CSR

- S schedule 2PL
- Consideriamo per ciascuna transazione l'istante in cui ha tutti i lock e sta per rilasciare il primo
- Ordiniamo le transazioni in accordo con questo valore temporale e consideriamo lo schedule seriale corrispondente
- Vogliamo dimostrare che tale schedule è equivalente ad S:
  - allo scopo, consideriamo un conflitto fra un'azione di t_i e un'azione di t_j con i<j; è possibile che compaiano in ordine invertito in S? no, perché in tal caso t_j dovrebbe aver rilasciato la risorsa in questione prima della sua acquisizione da parte di t_i

Concorrenza e fallimento di transazioni

- Rimuoviamo l'ipotesi di "commit-proiezione"
- Le transazioni possono fallire
- Conseguenze negative: rischio di:
  - letture sporche:
    - se T_i ha letto un dato scritto da T_k e T_k fallisce, ma nel frattempo T_i è andata in commit, allora abbiamo l'anomalia
  - rollback a cascata ("effetto domino"):
    - se T_i ha letto un dato scritto da T_k e T_k fallisce, allora anche T_i deve fallire

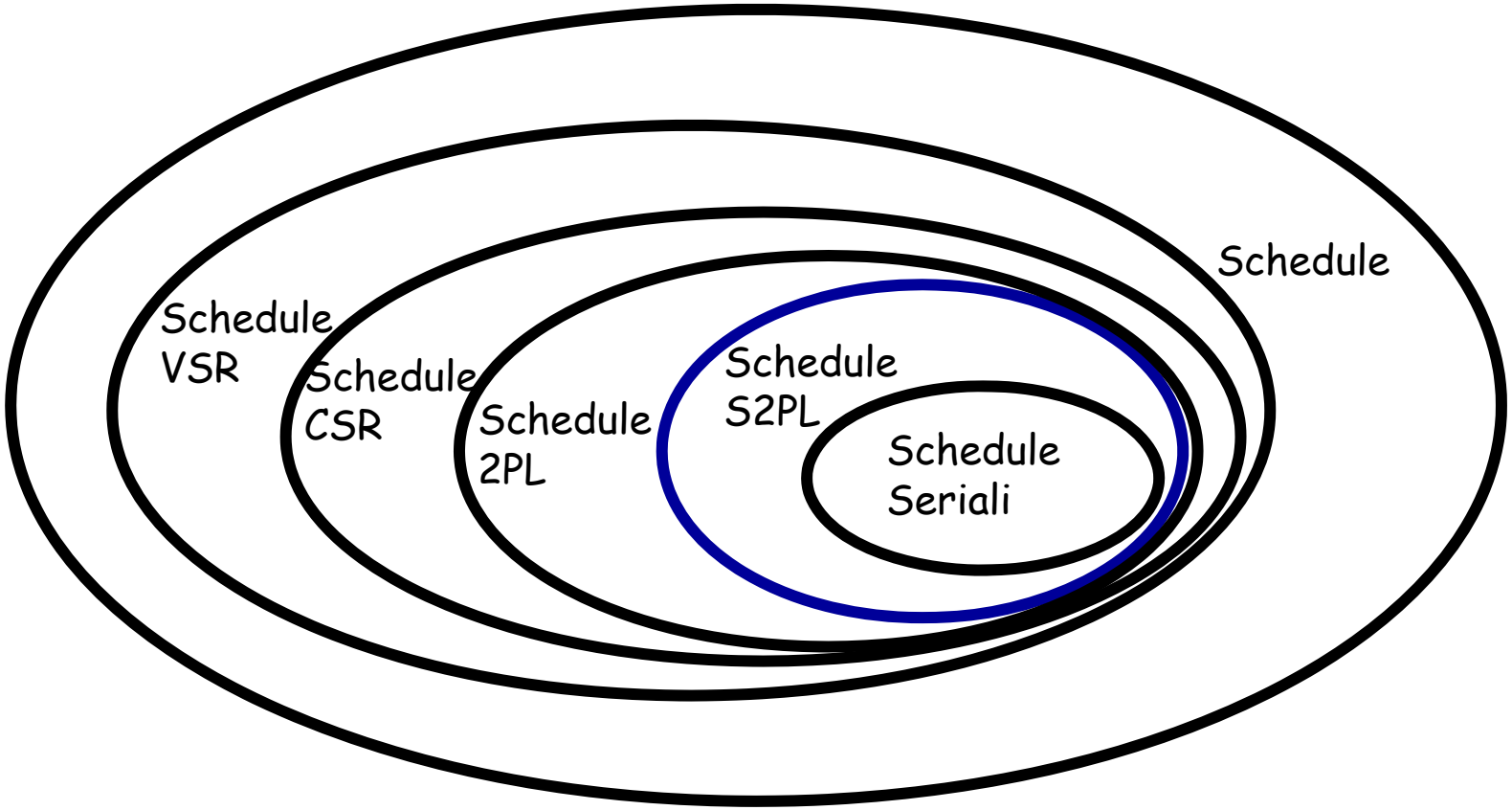
Evitiamo effetto domino e letture sporche

- letture sporche:
  - una transazione non può andare in commit e non può comunicare all'esterno ciò che ha letto, finché non sono andate in commit tutte le transazioni da cui ha letto;
- rollback a cascata ("effetto domino")
  - una transazione non deve poter leggere dati scritti da transazioni che non sono ancora andate in commit

Locking a due fasi stretto (S2PL)

- 2PL con una condizione aggiuntiva:
  - **i lock possono essere rilasciati solo dopo il commit o l'abort**
- Evita sia le letture sporche sia l’effetto domino

CSR, VSR e 2PL


Schedule
Schedule VSR
Schedule CSR
Schedule 2PL
Schedule S2PL
Schedule Seriali

Lock management

- Interface:
  - `r_lock(T, x, errcode, timeout)`
  - `w_lock(T, x, errcode, timeout)`
  - `unlock(T, x)`

    `T`: transaction identifier
    `X`: data element
    `timeout`: max wait in queue
- If timeout expires, `errcode` signals an error, typically the transaction rolls back and restarts

Hierarchical locking

- In many real systems locks can be specified at different granularity, e.g. tables, fragments, tuples, fields. These are organized in a hierarchy (possibly a directed acyclic graph)
- 5 locking modes:
  - 2 are shared and exclusive, renamed as:
    - XL: exclusive lock
    - SL: shared lock
  - 3 are new:
    - ISL: intention shared lock
    - IXL: intention exclusive lock
    - SIXL: shared intention-exclusive lock
- The choice of lock granularity is left to application designers;
  - too coarse: many resources are blocked
  - too fine: many locks are requested

Hierarchical locking protocol

- Locks are requested from the root to descendents in a hierarchy
- Locks are released starting at the node locked and moving up the tree
- In order to request an SL or ISL on a node, a transaction must already hold an ISL or IXL lock on the parent node
- In order to request an IXL, XL, or SIXL on a node, a transaction must already hold an SIXL or IXL lock on the parent node
- The new conflict table is shown in the next slide

Conflicts in hierarchical locking

Request \ Resource state	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Stallo (deadlock)

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

    $r\_lock_1(x)$, $r\_lock_2(y)$, $r_1(x)$, $r_2(y)$, $w\_lock_1(y)$, $w\_lock_2(x)$

Risoluzione dello stallo

- Uno stallo corrisponde ad un ciclo nel grafo delle attese (nodo=transazione, arco=attesa)
- Tre tecniche
  1. Timeout (problema: scelta dell'intervallo, con trade-off)
  2. Rilevamento dello stallo
  3. Prevenzione dello stallo
- Rilevamento: ricerca di cicli nel grafo delle attese
- Prevenzione: uccisione di transazioni "sospette" (può esagerare)

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 (o almeno dovrebbe esserlo)

Anomalie

Perdita di aggiornamento	**W-W**
• `read uncommitted`	
Lettura sporca	**R-W (o W-W) con abort**
• `read committed`	
Letture inconsistenti	**R-W**
Aggiornamento fantasma	**R-W**
• `repeatable read`	
Inserimento fantasma	**R-W su dato "nuovo"**
• `serializable`	

Livelli di isolamento: implementazione

- Sulle scritture si ha sempre il 2PL stretto (e nonostante ciò che dicono i manuali non sempre si evita la perdita di aggiornamento; le transazioni in scrittura debbono avere il livello massimo di isolamento)
- **`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 stretto anche in lettura, con lock sui dati
- **`serializable`**:
  - 2PL stretto con lock di predicato

Lock di predicato

- Caso peggiore:
  - sull'intera relazione
- Se siamo fortunati:
  - sull'indice

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 variabili RTM(x) e WTM(x) per ogni oggetto
- Lo scheduler riceve richieste di letture e scritture (con indicato il timestamp della transazione):
  - $r_t(x)$:
    - se $t <$ 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 t
  - $w_t(x)$:
    - se $t <$ WTM(x) o $t <$ RTM(x) allora la richiesta è respinta e la transazione viene uccisa,
    - altrimenti, la richiesta viene accolta e WTM(x) è posto uguale a t

- Per funzionare anche senza ipotesi di commit-proiezione, deve "bufferizzare" le scritture fino al commit (con attese)

Esempio

RTM(x) = 7

WTM(x) = 4

Richiesta	Risposta	Nuovo valore
$r_6(x)$	ok	
$r_8(x)$	ok	RTM(x) = 8
$r_9(x)$	ok	RTM(x) = 9
$w_8(x)$	no, t_8 uccisa	
$w_{11}(x)$	ok	WTM(x) = 11
$r_{10}(x)$	no, t_{10} uccisa	

2PL vs TS

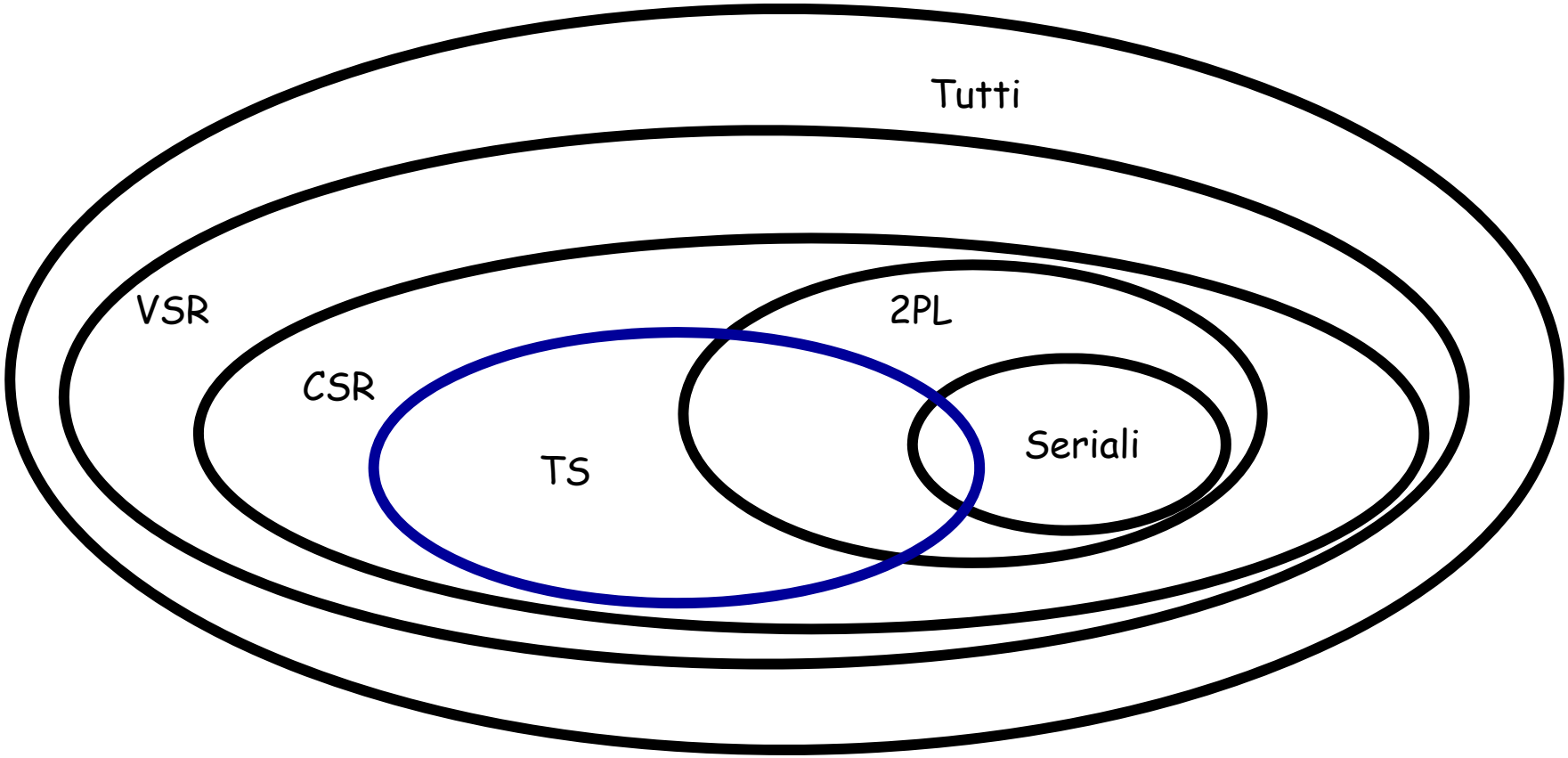
- Sono incomparabili
  - Schedule in TS ma non in 2PL

    $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_0(y)\ w_1(y)$
  - Schedule in 2PL ma non in TS

    $r_2(x)\ w_2(x) r_1(x)\ w_1(x)$
  - Schedule in TS e in 2PL

    $r_1(x)\ r_2(y)\ w_2(y)\ w_1(x)\ r_2(x)\ w_2(x)$

CSR, VSR, 2PL e TS



2PL vs TS

- In 2PL le transazioni sono poste in attesa
- In TS uccise e rilanciate
- Per rimuovere la commit proiezione, attesa per il commit in entrambi i casi
- 2PL può causare deadlock
- Le ripartenze sono di solito più costose delle attese:
  - originariamente quasi tuitti i sistemi preferivano il 2PL
  - ora si stanno diffondendo quelli con timestamp e multiversioni (vediamo fra poco) e quelli che usano 2PL per le transazioni che scrivono e multiversioni per le read-only

Multiversion concurrency control

- Main idea: writes generate new copies, reads make access to the correct copy
- Writes generate new copies each with a WTM. At any time, $N \geq 1$ copies of each object x are active, with $WTM_N(x)$. There is only one global $RTM(x)$
- Mechanism:
  - $r_t(x)$: is always accepted. x_k selected for reading such that: if $t > WTM_N(x)$, then k = N, otherwise k is taken such that $WTM_k(x) < t < WTM_{k+1}(x)$
  - $w_t(x)$: if $t < RTM(x)$ the request is refused, otherwise a new version of the item of data is added *(*N increased by one) with $WTM_N(x) = t$
- Old copies are discarded when there are no read transactions interested in their values

Controllo di concorrenza in DBMS reali

- DB2
  - sostanzialmente 2PL stretto, anche se ci sono molti dettagli che tralasciamo;
- PostgesSQL:
  - Multiversione, con la condizione "a serializable transaction cannot modify or lock rows changed by other transactions after the serializable transaction began"
- In entrambi, diversi livelli di isolamento anche per le transazioni che scrivono (cosa che non si dovrebbe fare)

Controllo di concorrenza in Postgres: multiversion

- Read Committed (Read Uncommitted non è implementato, si comnporta nello stesso modo), default:
  - Letture su dati in commit al momento della lettura (SELECT vede solo dati andati in commit prima dell'inizio della SELECT stessa, a parte gli aggiornamenti della transazione in cui è inserita)
  - Scritture con 2PL stretto (richiede lock, e li tiene fino al commit; e rispetta i lock altrui)
- Serializable (e Repeatable Read); in effetti i phantom si possono presentare, tralasciamo:
  - Letture multiversion (su dati in commit all'inizio della transazione, a parte … come sopra)
  - Scritture con 2PL stretto e rispetto delle versioni (una transazione T non può modificare dati che siano stati modificati da un'altra transazione T' dopo l'avvio di T)

Esercizi proposti

- Verificare il comportamento dei DBMS (almeno uno) rispetto ai livelli di isolamento e alle anomalie
- Ad esempio, verificare che cosa succede, con diversi livelli di isolamento, per gli esempio seguenti

Esempio 1 (perdita di aggiornamento)

Start Transaction	
Read(x)	
	Start Transaction
	Read(x)
x=x+100	
Write(x)	
Commit	
	x=x+1
	Write(x)
	Commit

Esempio 1 bis (piccola variante)

Start Transaction	
Read(x)	
	Start Transaction
	Read(x)
x=x+100	
Write(x)	
	x=x+1
	Write(x)
Commit	
	Commit

Esempio 2, lettura inconsistente

Start Transaction	
Read(x)	
	Start Transaction
	Read(x)
	x=x+20
	Write(x)
	Commit
Read(x)	
Commit	

Casi di studio per il tuning di basi di dati

dal testo:

D. Shasha.
Database Tuning: a principled approach.
Prentice-Hall, 1992

Caso 1

- Abbiamo una base di dati con dieci relazioni su due dischi: cinque e cinque; su un disco c'è anche il log.
- Compriamo un nuovo disco; che cosa ci mettiamo?

Possibile risposta

- Il log

Caso 2

- Il tempo di risposta è variabile, in particolare ci sono rallentamenti quando vengono eseguite operazioni DDL

Possibile motivazione

- Le operazioni DDL richiedono lock in scrittura sul catalogo

Caso 3

- Transazioni così organizzate sono insoddisfacenti:
  - attribuzione di un nuovo numero di pratica
  - richiesta di informazioni interattive
  - effettuazione dell'inserimento

Possibile soluzione

- transazione troppo lunga; vanno riordinati i passi, e nella transazione ci devono essere solo il primo e il terzo

Caso 4

- Una transazione così organizzata eseguita a fine mese, di sera, è inefficiente:

  per ogni conto stampare tutte le info ...

Possibile soluzione

- essendo di sola lettura, di sera (tempo morto) potrebbe essere senza lock (Read Uncommitted)

Casi 5 e 6

- Nell'ambito di una transazione, si calcola lo stipendio medio per ciascun dipartimento. Contemporaneamente si fanno modifiche su singoli stipendi.
- Le prestazioni sono insoddisfacenti.

Possibili soluzioni

- si può eseguire in un tempo morto (senza aggiornamenti) senza lock (Read Uncommitted)
- se sono tollerate (leggere) inconsistenze, si può procedere senza lock (Read Uncommitted o Read Committed)
- si può fare una copia e lavorare su di essa (dati non attuali)
- se nessuna delle alternative è praticabile (non ci sono tempi morti e si vogliono dati attuali e consistenti) si può provare con Repeatable Read (non c'è rischio di "phantom")

Caso 7

- Un'applicazione prevede:
  - migliaia di inserimenti ogni ora
  - centinaia di migliaia di piccoli aggiornamenti ogni ora
- Gli inserimenti arrivano in transazioni grandi ogni mezz'ora e durano 5 minuti. In queste fasi le prestazioni sono inaccettabili (tempo di risposta 30 sec, rispetto a mezzo secondo)

Possibili soluzioni

- spezzare le transazioni con gli inserimenti

Caso 8

- Un'applicazione che prevede un'istruzione SQL all'interno di un ciclo è lenta (e usa molto tempo di CPU)

Possibile soluzione

- usare bene il cursore (facciamo fare i cicli e soprattutto i join all'SQL)

Caso 10

- Una società di servizi emette tutte le bollette a fine mese, con un programma che ha bisogno di tutta la notte, impedendo così l'esecuzione di altri programmi batch che sarebbero necessari

Possibili soluzioni

- è proprio necessario che le bollette siano emesse tutte insieme?
- se è proprio necessario, magari facciamolo durante il week-end (tempo morto più lungo)