

GESTIONE DELLE TRANSAZIONI (P.379 LIBRO) .

Simone Remoli.

Il controllo dell'affidabilità garantisce due proprietà fondamentali delle transazioni: l'atomicità e la persistenza. In pratica garantisce che le transazioni non vengano lasciate incomplete, con alcune operazioni eseguite e altre no, e che gli effetti di ciascuna transazione conclusa con un commit siano mantenuti in modo permanente.

Il controllore dell'affidabilità svolge il proprio compito attraverso il LOG, ossia un **archivio persistente** su cui registra le varie azioni svolte dal DBMS.

IL FILE LOG.

Il LOG è un file sequenziale di cui è responsabile il controllore dell'affidabilità, scritto in memoria e contenente l'informazione ridondante che permette di ricostruire il contenuto della base di dati a seguito di guasti.

Sul LOG vengono registrate le azioni svolte dalle varie transazioni.

Il LOG ha un blocco corrente (detto TOP) che è l'ultimo ad essere allocato al LOG stesso.

I record nel LOG vengono scritti **sequenzialmente** nel blocco corrente, e quando esso termina vengono scritti in un successivo blocco, allocato al LOG, che diventa il blocco corrente.

I record nel LOG sono di due tipi: **record di transazione** e **record di sistema**.

I **record di transazione** registrano le attività svolte da ciascuna transazione, nell'ordine in cui esse vengono effettuate.

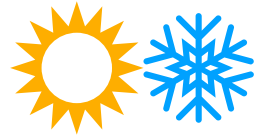
Quindi, ogni transazione inserisce nel LOG un record di **begin** = start transaction, vari record corrispondenti alle **azioni effettuate** (insert, delete, update) e un record di **commit** (o di **abort** = rollback).

I **record di sistema** indicano l'effettuazione di operazioni specifiche del controllore dell'affidabilità: **dump** e **checkpoint**.

UNDO VS REDO

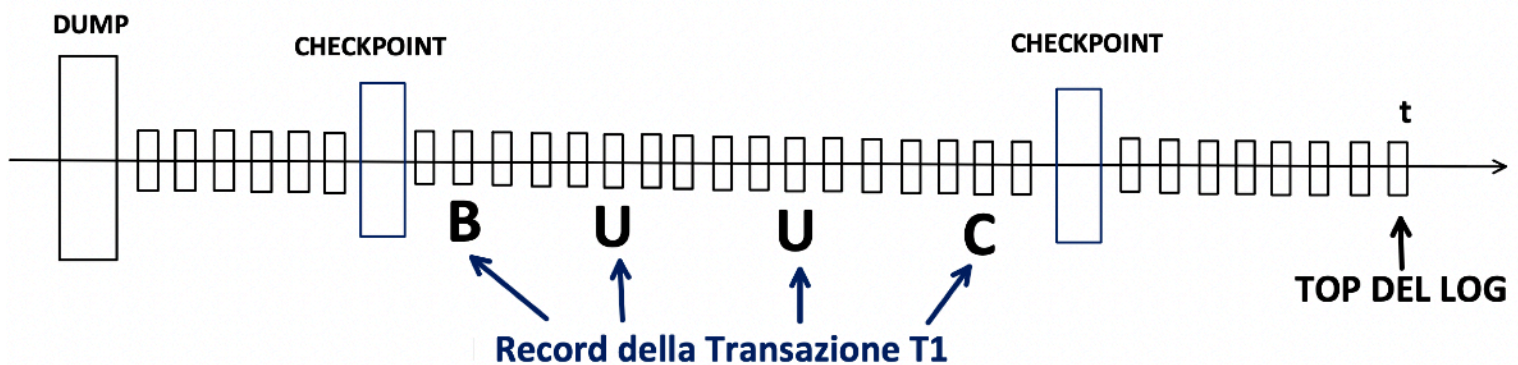
Come si vedrà, ogni azione di scrittura sulla base di dati viene protetta tramite un'azione sul LOG, in modo che sia possibile **“disfare”** (undo) le azioni a seguito di malfunzionamenti o guasti precedenti il commit, oppure **“rifare”** (redo) queste azioni qualora la buona riuscita sia incerta e le transazioni abbiano effettuato un commit.

RIPRESA A CALDO E RIPRESA A FREDDO



Il controllore dell'affidabilità è responsabile di realizzare le operazioni di ripristino dopo i malfunzionamenti, dette **ripresa a caldo** e **ripresa a freddo**.

ESEMPIO DI LOG



Questa figura mostra la sequenza di record presenti in un LOG.

Vengono registrati i record associati all'azione della transazione T1, in un LOG che viene scritto anche da altre transazioni.

La transazione T1 esegue due modifiche (U, Update) prima di andare a buon fine terminando con un commit.

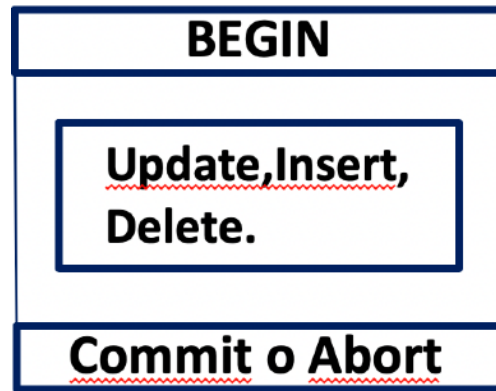
Questo LOG evidenzia la presenza di un record di dump e di vari record di checkpoint.

STRUTTURA DEI RECORD NEL LOG

I record di LOG che vengono scritti per descrivere le attività di una transazione T_i sono elencati di seguito:

1. I record di BEGIN, COMMIT e ABORT, che contengono, oltre all'indicazione del tipo di record, anche l'identificativo T della transazione.
2. I record di update, che contengono l'identificativo T della transazione, l'identificativo O dell'oggetto su cui avviene l'update, e poi due valori BS (Before State) e AS (After State) che descrivono rispettivamente il valore dell'oggetto O precedentemente alla modifica e successivamente alla modifica.
3. I record di insert e di delete sono analoghi a quelli di update, da cui si differenziano per l'assenza nei primi del BS e nei secondi dell'AS.

Quindi ricapitoliamo la struttura dei record nel LOG:



NOTAZIONE IMPORTANTE:

$B(T)$ = Record di Begin;

$A(T)$ = Record di Abort;

$C(T)$ = Record di Commit;

$U(T, O, BS, AS)$ = Record di Update;

$I(T, O, AS)$ = Record di Insert;

$D(T, O, BS)$ = Record di Delete;

Questi record consentono di disfare e rifare le rispettive azioni sulla base di dati, attraverso due operazioni di competenza chiamate **UNDO** e **REDO**.

Undo : Per disfare un'azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore BS; L'insert viene disfatto cancellando l'oggetto O.

Redo: Per rifare un'azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore AS; Il delete viene rifatto cancellando l'oggetto O.

Nota bene: per le primitive di Undo e Redo vale la proprietà di idempodenza, ossia vale la seguente espressione.

$$\text{Undo}(\text{Undo}(A)) = \text{Undo}(A)$$

$$\text{Redo}(\text{Redo}(A)) = \text{Redo}(A).$$

CHECKPOINT (frequenti) E DUMP(rare)

Il controllore dell'affidabilità predispone i dati necessari per eseguire i meccanismi di ripristino dai guasti realizzando azioni di **checkpoint** e di **dump**.

Per semplicità, facciamo riferimento alle azioni sulla base di dati come se fossero sempre operazioni di lettura o scrittura di un'intera pagina (fermo restando che la realtà è più complessa).

I record di LOG di sistema che abbiamo visto, potrebbero essere sufficienti per svolgere un'operazione di ripristino a seguito di un guasto. Questa azione di ripristino però sarebbe molto lunga, perché dovrebbe utilizzare l'intero LOG.

Per semplificarla sono stati inventati opportuni accorgimenti.

Definizione di Checkpoint: un Checkpoint è un'operazione che viene svolta **periodicamente** del gestore dell'affidabilità (in stretto coordinamento con il Buffer Manager), con l'obiettivo di registrare quali transazioni sono attive.

Come avviene questa operazione periodica?

1. Si sospende l'accettazione di operazioni di scrittura, commit o about, da parte di ogni transazione.
2. Si trasferiscono in memoria di massa tutte le pagine del buffer su cui sono state eseguite modifiche da parte di transazioni che hanno già effettuato il commit.
3. Si scrive nel LOG un record di checkpoint che contiene gli ID delle transazioni attive.
4. Si riprende l'accettazione delle operazioni sopra sospese.

Questo garantisce che gli effetti delle transazioni che hanno eseguito il commit siano registrati nella base di dati in modo permanente, mentre nel checkpoint sono elencate le transazioni che non hanno ancora espresso una scelta relativamente all'esito finale, quindi sostanzialmente transazioni in corso, che non hanno eseguito un commit o un abort.

Definizione di Dump: un Dump è una copia completa e consistente della base di dati, che viene normalmente effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo (ad esempio, di notte oppure durante il weekend). Dopo la conclusione dell'operazione di Dump viene scritto nel LOG un record di dump, che segnala appunto la presenza di una copia fatta in un determinato istante; dopodiché il sistema può tornare al suo funzionamento normale.

Da un punto di vista di notazione, useremo i simboli DUMP per denotare il record di dump e CK(T1,T2,.....,Tn) per denotare un record di checkpoint, ove T1,T2,.....,Tn denotano gli ID delle transazioni attive all'istante del checkpoint.

Fine della premessa, cominciamo operativi.

ESECUZIONE DELLE TRANSAZIONI E SCRITTURA DEL LOG.

Durante il funzionamento normale delle transazioni, il controllore dell'affidabilità deve garantire che siano seguite due regole, che consentono di ripristinare la correttezza della base di dati a fronte di guasti.

REGOLA WAL (Write Ahead Log, cioè scrivi il Log per primo) : impone che la parte BS dei record di un LOG venga scritta nel LOG, prima di effettuare l'operazione sulla base di dati. In questo modo si possono disfare le scritture già effettuate in memoria di massa da parte di una transazione che non ha ancora effettuato un commit.
Quindi carica BS (L'aggiornamento precedente) perché comunque la transazione non ha fatto il commit e quindi non si dovrebbero vedere gli aggiornamenti nuovi.

REGOLA COMMIT-PRECEDENZA : impone che la parte AS dei record di un LOG venga scritta nel LOG prima di effettuare il commit. Questa regola permette di rifare le scritture già decise da una transazione che, ahimè, ha già effettuato il commit, ma le cui pagine non sono ancora state trascritte dal buffer manager alla memoria di massa.
Quindi carica AS (L'aggiornamento successivo) perché comunque la transazione ha fatto il commit e quindi si dovrebbero vedere gli aggiornamenti nuovi.

Prima del commit, un eventuale guasto comporta l'Undo delle azioni effettuate, ricostruendo lo stato iniziale della base di dati.

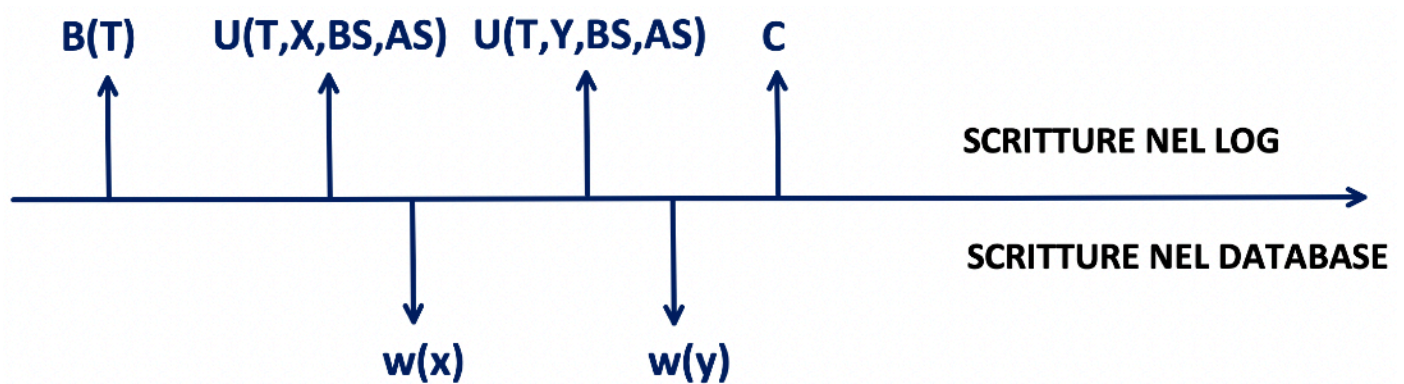
Dopo il commit, un eventuale guasto comporta il Redo delle azioni effettuate, in modo da ricostruire con certezza lo stato finale della transazione.

Undo : Per disfare un'azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore BS; L'insert viene disfatto cancellando l'oggetto O.

Redo: Per rifare un'azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore AS; Il delete viene rifatto cancellando l'oggetto O.

SCRITTURE NEL LOG VS SCRITTURE NEL DATABASE.

Consideriamo il seguente schema:



T è l'identificativo della transazione.

La transazione scrive il record $B(T)$: record di Begin.

Poi scrive il record di Log $U(T,X,BS,AS)$, relativo alla pagina X .

Dopo c'è la scrittura della pagina X sulla base di dati.

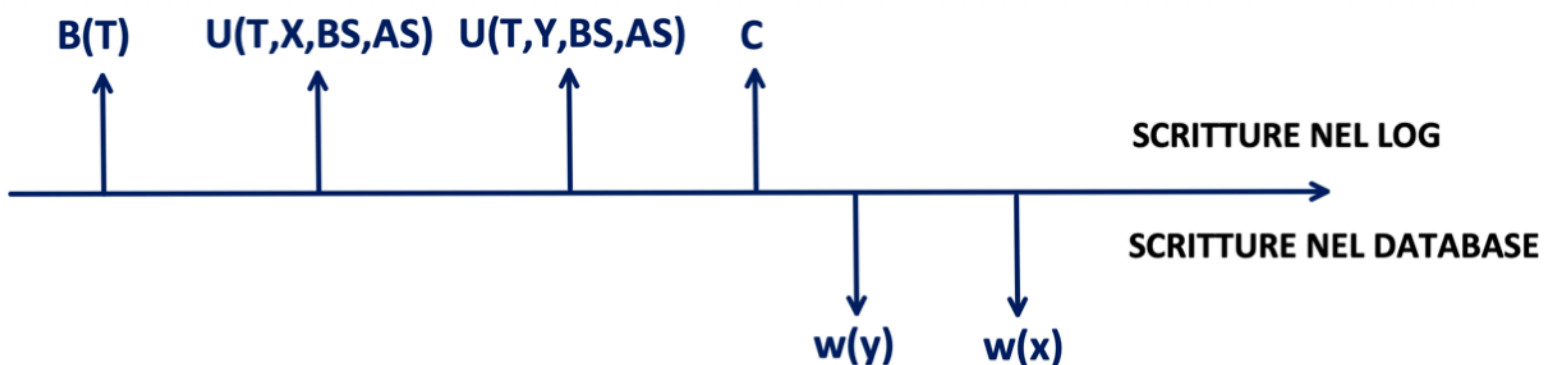
Poi scrive il record di Log $U(T,Y,BS,AS)$, relativo alla pagina Y .

Dopo c'è la scrittura della pagina Y sulla base di dati.

Tutte queste pagine sono scritte dal buffer manager PRIMA DEL COMMIT: in questo modo, al commit tutte le pagine della base di dati modificate dalla transazione sono certamente scritte in memoria di massa.

Quindi, questo schema non richiede operazioni di REDO.

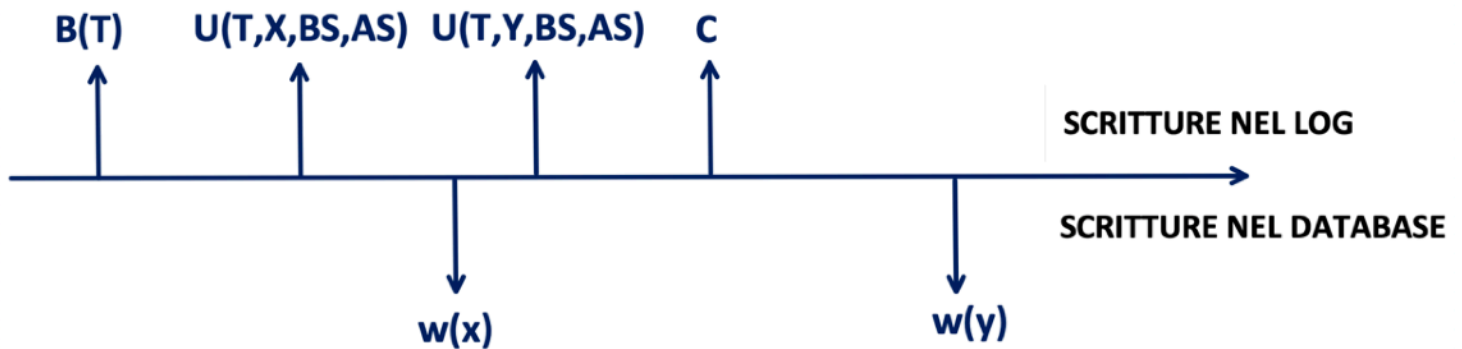
Consideriamo il seguente schema:



La scrittura dei record dei LOG precede quella delle azioni sulla Base di Dati. Queste ultime avvengono dopo la decisione di commit e la conseguente scrittura

del record di commit sul LOG;
Quindi, questo schema non richiede operazioni di UNDO.

Consideriamo il seguente schema:



Secondo questo schema, le scritture nella base di dati, una volta protette dalle opportune scritture sul LOG, possono avvenire in un qualunque momento rispetto alla scrittura del record di commit sul LOG.

Quindi, questo schema richiede operazioni sia UNDO che REDO.
Perché se si verifica un guasto devo fare REDO per vedere gli aggiornamenti $w(y)$, e devo fare UNDO per vedere gli aggiornamenti precedenti al commit $w(x)$.

Tutti e tre i protocolli rispettano le regola WAL e Commit-Precedenza, e scrivono il record di commit in modo sincrono: essi si differenziano solo per il momento in cui scrivono le pagine della base di dati.

GESTIONE DEI GUASTI E RIPRESE.

Quando il sistema individua un guasto, esso forza immediatamente un arresto completo delle transazioni e il successivo ripristino del corretto funzionamento del sistema operativo (boot).

Quindi, viene attivata una procedura di **ripresa a caldo** nel caso di guasti di sistema e di **ripresa a freddo** nel caso di guasto di dispositivo.

Al termine delle procedure di ripresa, il sistema diventa nuovamente utilizzabile dalle transazioni.

RIPRESA A CALDO 🥵

La ripresa a caldo è articolata in quattro fasi successive.

1. Si accede all'ultimo blocco del LOG, che era corrente all'istante del guasto, e si ripercorre all'indietro il Log fino all'ultimo (cioè più recente) record di checkpoint.

2. Si decidono le transazioni da rifare o disfare. Si costruiscono due insiemi, detti UNDO e REDO, contenenti identificativi di transazioni. Si analizza l'insieme di UNDO con le transazioni attive al checkpoint e l'insieme di REDO con l'insieme vuoto. Si percorre poi il LOG in avanti, aggiungendo all'insieme di UNDO tutte le transazioni di cui è presente un record di begin, e spostando dall'insieme di UNDO all'insieme di REDO tutti gli identificativi delle transazioni di cui è presente il record di commit. Al termine di questa fase, gli insiemi di UNDO e REDO contengono rispettivamente tutti gli identificativi delle transazioni da disfare e di quelle da rifare.

3. Si ripercorre all'indietro il LOG disfacendo le transazioni nel set di UNDO, risalendo fino alla prima azione della transazione più "vecchia" nei due insiemi di UNDO e REDO; Si noti che questa azione potrebbe precedere il record di checkpoint nel LOG.

4. Infine, nella quarta fase si applicano le azioni di REDO nell'ordine in cui sono registrate nel LOG. In questo modo, viene replicato esattamente il comportamento delle transazioni originali.

ESEMPIO DI APPLICAZIONE - TIPICO ESERCIZIO D'ESAME.

Si supponga che nel LOG vengano registrate le azioni:

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).

Successivamente si verifica un guasto.

Illustrare dettagliatamente i passi da compiere per effettuare la **ripresa a caldo**.

Soluzione.

1. Si accede al record di checkpoint CK(T2,T3,T4), e si costruiscono gli insiemi UNDO e REDO.

$$\text{UNDO} = \{T2, T3, T4\}.$$

$$\text{REDO} = \{\}.$$

Questa è l'inizializzazione.

2. Ora, si percorre in avanti il record di LOG e vediamo quali sono i record di commit e di begin.

I record sono: C(T4), B(T5), C(T5).

C(T4): Undo = {T2,T3} Redo = {T4}.

B(T5): Undo = {T2,T3,T5} Redo = {T4}.

C(T5): Undo = {T2,T3} Redo = {T4, T5}.

3. Ora si ripercorre all'indietro il LOG fino alla prima azione della transazione più vecchia nei due insiemi UNDO e REDO.

T2 è la transazione più vecchia dei due insiemi, quindi si prende la prima azione (che non sia begin) di questa transazione, ossia U(T2,O1,B1,A1).

Quindi ripercorriamo all'indietro il LOG disfacendo le transazioni nel set di UNDO.

Ora in Undo ci sono solo T2 e T3: UNDO = {T2,T3}.

Vengono generate le seguenti azioni di UNDO:

(Si aggiorna con BEFORE STATE)

I(T2,O6,A8) ERA INSERT, quindi delete (O6).

D(T3,O5,B7) era DELETE, quindi insert (O5 = B7).

U(T3,O3,B5,A5) ERA UPDATE, quindi (O3=B5).

U(T3,O2,B3,A3) era UPDATE, quindi (O2=B3).

U(T2,O1,B1,A1) era UPDATE, quindi (O1=B1).

4. Infine, vengono svolte le azioni di REDO:

L'insieme è inizializzato in questo stato: Redo = {T4,T5}.

Applichiamo lo stesso procedimento nell'ordine in cui le transazioni sono registrate nel LOG.

(Si aggiorna con AFTER STATE)

$U(T4, O3, B4, A4)$ quindi $O3=A4$
 $U(T5, O4, B6, A6)$ quindi $O4=A6$.

Attenzione a disfare e rifare le azioni (UNDO) VS (REDO) :

Undo : Per disfare un'azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore BS; L'insert viene disfatto cancellando l'oggetto O.

Redo: Per rifare un'azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore AS; Il delete viene rifatto cancellando l'oggetto O.

Fine della ripresa a caldo.

PROVA D'ESAME 23 FEBBRAIO 2024 – ESERCIZIO TRANSAZIONI

Si consideri il seguente log di un sistema di gestione di basi di dati:

Dump, B(T1), B(T2), B(T3), I(T1, O1, A1), D(T2, O2, B2), B(T4), U(T4, O3, B3, A3),
U(T1, O4, B4, A4), C(T2), CK(T1, T3, T4), B(T5), B(T6), U(T5, O5, B5, A5), A(T3), CK(...),
B(T7), A(T4), U(T7, O6, B6, A6), U(T6, O3, B7, A7), B(T8), A(T7).

Successivamente, si verifica un guasto.

1. Individuare le transazioni attive al secondo checkpoint
2. Illustrare dettagliatamente i passi da compiere per effettuare la ripresa a caldo.

Soluzione.

1. Per rispondere al primo quesito ci si basa esclusivamente sui checkpoint.
Esiste un checkpoint che è presente prima del secondo, ossia CK(T1, T3, T4)! Questo indica che, al momento, le transazioni attive sono T1, T3, T4.

Ma la T3 va in ABORT.

Nel frattempo vanno in begin T5 e T6.

Quindi, le transazioni attive al secondo checkpoint sono:

T1,T4,T5,T6.

Possiamo riscrivere il Log nel seguente modo:

Dump, B(T1), B(T2), B(T3), I(T1,O1,A1), D(T2,O2,B2), B(T4), U(T4,O3,B3,A3),
U(T1,O4,B4,A4), C(T2), CK(T1,T3, T4), B(T5), B(T6), U(T5,O5,B5,A5), A(T3),
CK(T1,T4,T5,T6), B(T7), A(T4), U(T7,O6,B6,A6), U(T6,O3,B7,A7), B(T8), A(T7).

2. Andiamo nel primo checkpoint che incontriamo partendo dalla fine.

Dump, B(T1), B(T2), B(T3), I(T1,O1,A1), D(T2,O2,B2), B(T4), U(T4,O3,B3,A3),
U(T1,O4,B4,A4), C(T2), CK(T1,T3, T4), B(T5), B(T6), U(T5,O5,B5,A5), A(T3),
CK(T1,T4,T5,T6), B(T7), A(T4), U(T7,O6,B6,A6), U(T6,O3,B7,A7), B(T8), A(T7).

E inizializzo l'insieme di UNDO con le transazioni attive nel checkpoint:

Undo = {T1,T4,T5,T6}.

Redo = {}.

Ora percorro il LOG in avanti e aggiorno gli insiemi di UNDO e REDO.

Dump, B(T1), B(T2), B(T3), I(T1,O1,A1), D(T2,O2,B2), B(T4), U(T4,O3,B3,A3),
U(T1,O4,B4,A4), C(T2), CK(T1,T3, T4), B(T5), B(T6), U(T5,O5,B5,A5), A(T3),
CK(T1,T4,T5,T6), **B(T7)**, A(T4), U(T7,O6,B6,A6), U(T6,O3,B7,A7), B(T8), A(T7).

Undo = {T1,T4,T5,T6, T7}.

Redo = {}.

Dump, B(T1), B(T2), B(T3), I(T1,O1,A1), D(T2,O2,B2), B(T4), U(T4,O3,B3,A3),
U(T1,O4,B4,A4), C(T2), CK(T1,T3, T4), B(T5), B(T6), U(T5,O5,B5,A5), A(T3),
CK(T1,T4,T5,T6), B(T7), A(T4), U(T7,O6,B6,A6), U(T6,O3,B7,A7), **B(T8)**, A(T7).

Undo = {T1,T4,T5,T6, T7,T8}.

Redo = {}.

3. Ora ripercorriamo all'indietro il LOG fino alla transazione più vecchia nei due insiemi di UNDO e REDO.

Le azioni di UNDO che vengono eseguite sono:

Dump, B(T1), B(T2), B(T3), **I(T1,O1,A1)**, D(T2,O2,B2), B(T4), U(T4,O3,B3,A3),
U(T1,O4,B4,A4), C(T2), CK(T1,T3,T4), B(T5), B(T6), U(T5,O5,B5,A5), A(T3),
CK(T1,T4,T5,T6), B(T7), A(T4), U(T7,O6,B6,A6), U(T6,O3,B7,A7), B(T8), A(T7).

L'insieme UNDO = {T1,T4,T5,T6,T7,T8}

O3 = B7

O6 = B6

O5 = B5

O4 = B4

O3 = B3

Delete O1.

Fine dell'esercizio! (Risoluzione identica al Pellegrini)

RIPRESA A FREDDO.

Durante una prima fase, si accede al dump e si ricopia selettivamente la parte deteriorata dalla base di dati. Si accede anche al più recente record di dump del LOG. Dopodiché si ripercorre in avanti il LOG, applicando relativamente alla parte deteriorata della base di dati sia le azioni sulla base di dati sia le azioni di commit o abort, riportandosi così nella situazione precedente al guasto.

Infine, si parte con una ripresa a caldo.

FINE PRIMA PARTE CONCORRENZA.

CONTROLLO DI CONCORRENZA.

In un DBMS, è indispensabile che le transazioni vengano eseguite concorrentemente: è impensabile infatti una loro esecuzione seriale, in cui cioè le transazioni vengono eseguite una alla volta.

Solo la concorrenza delle transazioni consente un uso efficiente del DBMS, massimizzando il numero di transazioni servite per secondo e minimizzando i tempi di risposta.

D'ora in poi daremo una descrizione astratta della base di dati.

Useremo gli oggetti x, y e z e le azioni $r(x)$ e $w(x)$ denoteranno la lettura e la scrittura della pagina in cui il dato x è memorizzato nel DBMS.

Anomalie delle transazioni concorrenti.

L'esecuzione concorrente di varie transazioni può causare anomalie.

PERDITA DI AGGIORNAMENTO.

Supponiamo di avere due transazioni identiche che operano sullo stesso oggetto della base di dati:

$T1: r1(x), x = x + 1, w1(x)$

$T2: r2(x), x = x + 1, w2(x)$

Dove " $ri(x)$ " denota la lettura del generico oggetto x da parte della transazione " Ti " e " $wi(x)$ " la scrittura dello stesso oggetto.

L'incremento di un oggetto x è effettuato per opera di un programma applicativo, per esempio un update in SQL.

Se le due transazioni vengono eseguite in sequenza con un valore iniziale di $x = 2$, allora il valore finale dell'oggetto x sarà pari a 4.

Analizziamo ora una possibile esecuzione concorrente delle due transazioni:

Quindi, sostanzialmente, viene letto un dato che rappresenta uno stato intermedio nell'evoluzione di una transazione.

LETTURE INCONSISTENTI.

Supponiamo invece che la transazione T1 svolga solamente operazioni di lettura, ma che ripeta la lettura del dato X in istanti successivi, come descritto nella seguente esecuzione:

| T1 | T2 |
|---------------|------------------|
| r1(x) | r2(x) |
| | x = x + 1 |
| | w2(x) |
| | <u>commit</u> |
| r1(x) | |
| <u>commit</u> | |

In questo caso, x assume il valore 2 dopo la prima operazione di lettura e il valore 3 dopo la seconda operazione di lettura.

Però, è opportuno, normalmente, che una transazione che accede due volte alla base di dati trovi esattamente lo stesso valore per ciascun dato letto, e non risenta dell'effetto di altre transazioni.

AGGIORNAMENTO FANTASMA.

Si consideri una base di dati con 3 oggetti x,y e z che soddisfano un vincolo di integrità, tali cioè che $x + y + z = 1000$, ed eseguiamo le seguenti transazioni.

| T1 | T2 |
|----------------------|--------------------|
| r1(x) | r2(y) |
| r1(y) | |
| | y = y - 100 |
| | r2(z) |
| | z = z + 100 |
| | w2(y) |
| | w2(z) |
| | <u>commit</u> |
| r1(z) | |
| s = x + y + z | |
| <u>commit</u> | |

La transazione T2 non altera la somma dei valori e quindi non viola il vincolo di integrità;
Però la variabile “s” della transazione T1, che *dovrebbe contenere la somma di x,y e z,
contiene in effetti al termine dell’esecuzione il valore 1100.
La transazione T1 osserva solo in parte gli effetti della transazione T2, e quindi osserva
uno stato che non soddisfa i vincoli di integrità.

TEORIA DEL CONTROLLO DI CONCORRENZA - IMPORTANTE PER ESAME

Definiamo una transazione come una sequenza di azioni di lettura o scrittura.

Per esempio, una transazione T1 è rappresentata da:

$T1 : r1(x) \ r1(y) \ w1(x) \ w1(y).$

Questo è un oggetto sintattico di cui si conoscono solo le azioni di ingresso/uscita.
Dato che le transazioni avvengono in modo concorrente, le operazioni di ingresso/uscita
vengono richieste da varie transazioni in istanti successivi.

Uno schedule è la sequenza di operazioni ingresso/uscita presentate da transazioni concorrenti.

Uno schedule S1 - esempio.

$S1 : r1(x) \ r2(z) \ w1(x) \ w2(z) \ \dots$

Dove $r1(x)$ rappresenta la lettura dell’oggetto x effettuata dalla transazione t1, e $w2(z)$ la
scrittura dell’oggetto z effettuata dalla transazione t2. Le operazioni compaiono nello
schedule seguendo l’ordine temporale con cui sono eseguite sulla base di dati.
Lo scheduler ha il compito di intercettare le operazioni compiute sulla base di dati dalle
transazioni, decidendo per ciascuna se rifiutarla o accettarla.

Uno schedule **SERIALE** è uno schedule in cui le azioni di ciascuna transazione compaiono
in sequenza, senza essere inframmezzate da istruzioni di altre transazioni. Lo schedule S2 è
seriale:

$S2: r0(x) \ r0(y) \ w0(x) \ r1(y) \ r1(x) \ w1(y) \ r2(x) \ r2(y) \ r2(z) \ w2(z)$

Teorema: L'esecuzione di uno schedule "Si" è corretta quando produce lo stesso risultato prodotto da un qualunque schedule seriale "Sj" delle stesse transazioni, in quel caso diremo che "Si" è SERIALIZZABILE.

Però, affermare questo è da coglioni: è vera questa cosa, quello sì, però stiamo assumendo che ciascuna transazione sia corretta, e che l'esecuzione di una sequenza di transazioni seriali pure.

I due metodi utilizzati in pratica per garantire la serializzabilità (accettano solo schedule serializzabili) sono il **locking a due fasi (2PL)** e il **controllo di concorrenza basato su timestamp**.

VIEW-EQUIVALENZA.

Un'operazione di lettura "ri(x)" legge-da una scrittura "wj(x)" quando "wj(x)" precede "ri(x)" e non vi è alcuna scrittura "wk(x)" compresa tra le due operazioni.

Esempio: w2(x) r1(x), r1(x) legge-da una scrittura w2(x).

Due schedule vengono detti View-Equivalenti se possiedono la stessa relazione "legge-da" e le stesse scritture finali.

Due schedule sono view-equivalenti se:

1. Ogni **operazione di lettura $r_i(x)$** legge dalla **stessa scrittura $w_j(x)$** in entrambi gli schedule.
2. L'**ultima scrittura su ogni variabile** (es: x, z, ecc.) è fatta dalla **stessa transazione**.
3. **Ogni operazione deve esistere in entrambi gli schedule** affinché possano essere confrontati: **View-equivalenza richiede che entrambi gli schedule abbiano le stesse operazioni** (letture e scritture), **per poterle confrontare**.
4. I due schedule devono avere **lo stesso insieme di transazioni**.

Quando si parla di **view-equivalenza**, è fondamentale che tutte le **letture e scritture si riferiscano allo stesso oggetto x (o y, z, ecc.)** tra le diverse transazioni.

Uno schedule viene detto View-Serializzabile se esiste uno schedule seriale view-equivalente ad esso.

VSR = L'insieme degli schedule View-Serializzabili.

Si considerino gli schedule seguenti:

$S3 : w0(x) \ r2(x) \ r1(x) \ w2(x) \ w2(z)$

$S4 : w0(x) \ r1(x) \ r2(x) \ w2(x) \ w2(z)$

$S5 : w0(x) \ r1(x) \ w1(x) \ r2(x) \ w1(z)$

$S6 : w0(x) \ r1(x) \ w1(x) \ w1(z) \ r2(x)$

$S3$ è View-Equivalente allo schedule seriale $S4$ → Quindi è View-Serializzabile (VSR).

Ogni lettura ($r1(x)$, $r2(x)$) legge dalla stessa scrittura $w0(x)$ in entrambi. E inoltre, l'ultima scrittura su x e z è fatta dalla stessa transazione ($T2$) in entrambi.

$S3 \in \text{VSR}$, perché esiste almeno uno schedule seriale view-equivalente, ovvero $S4$.

Notiamo che $S5$ non è invece view-equivalente a $S4$, ma è view-equivalente allo schedule seriale $S6$, e quindi risulta anche esso VSR.

Questo perché:

| Lettura | In S5 | In S6 | Uguale ? |
|------------|------------|------------|----------|
| $r1(x)$ | da $w0(x)$ | da $w0(x)$ | ✓ |
| $r2(x)$ | da $w1(x)$ | da $w1(x)$ | ✓ |
| Ultimo x | $w1(x)$ | $w1(x)$ | ✓ |
| Ultimo z | $w1(z)$ | $w1(z)$ | ✓ |

$R1$ legge da $W0$, $R2$ da $W1$ in $S5$.

$R1$ legge da $W0$, $R2$ da $W1$ in $S6$.

Si noti che tra $W1(x)$ e $R2(x)$ nello schedule $S6$, esiste un elemento di mezzo, ossia $W1(z)$: questo elemento non lede il problema in quanto rappresenta una scrittura di un elemento diverso, poichè $Z \neq X$.

Si considerino gli schedule seguenti:

$S7 : r1(x) \ r2(x) \ w2(x) \ w1(x)$

$S8 : r1(x) \ r2(x) \ w2(x) \ r1(x)$

S9 : $r1(x) \ r1(y) \ r2(z) \ r2(y) \ w2(y) \ w2(z) \ r1(z)$

E, come giustamente si riesce ad evincere, questi schedule non sono VSR.

In S7, $r1(x) \rightarrow$ legge da...?????? Non c'è scrittura su x prima, quindi legge dal valore iniziale.

$r2(x) \rightarrow$ idem: legge dal valore iniziale.

Si considerino gli schedule:

S1: $r1(x) \ w1(x)$

S2: $w1(x)$

Hanno la stessa scrittura finale ma non sono equivalenti.

Si considerino gli schedule:

S1: $w1(x) \ r1(x) \ w2(x)$

S2: $r2(x) \ w1(x) \ r1(x)$

S1 e S2 non sono view-equivalenti perché: S2 ha una lettura ($r2(x)$) che **non è presente** in S1

Le **scritture finali su x** sono fatte da transazioni diverse (T2 in S1, T1 in S2)

Si considerino gli schedule:

S3 : $w0(x) \ r2(x) \ r1(x) \ w2(x) \ w2(z) \ r2(x)$

S4 : $w0(x) \ r1(x) \ r2(x) \ w2(x) \ w2(z)$

S3 e S4 **NON** sono view-equivalenti, perché:

In S3, c'è una lettura aggiuntiva $r2(x)$ che legge da una scrittura diversa ($w2(x)$). Anche se le scritture finali sono uguali, le letture devono coincidere sia come presenza che come origine.

Si considerino gli schedule:

$$S3 : w0(x) \ r2(x) \ r1(x) \ w2(x) \ w2(z) \ r2(x)$$

$$S4 : w0(x) \ r1(x) \ r2(x) \ w2(x) \ w2(z) \ w2(y) \ r2(y)$$

In $S3$, $r2(x)$ legge due volte: una da $w0(x)$ e una da $w2(x)$.

In $S4$, $r2(x)$ legge una volta sola, da $w0(x)$.

Inoltre, $S4$ ha lettura $r2(y)$, che non esiste in $S3$.

CONFLICT-EQUIVALENZA.

Una nozione di equivalenza più facilmente utilizzabile si basa sulla definizione di conflitto.

Date due azioni “ a_i ” e “ a_j ”, con $i \neq j$, si dice che “ a_i ” è in conflitto con “ a_j ” se esse operano sullo stesso oggetto e almeno una di esse è una scrittura.

Si dice che lo schedule “ S_i ” è **Conflict-equivalente** allo schedule “ S_j ” se i due schedule presentano le stesse operazioni e ogni coppia di operazioni in conflitto è nello stesso ordine nei due schedule.

Uno schedule viene detto **Conflict-Serializable** se esiste uno schedule seriale a

esso Conflict-Equivalente: **CSR**.

Siano dati i seguenti schedule:

$$S10 : w0(x) \ r1(x) \ w0(z) \ r1(z) \ r2(x) \ r3(z) \ w3(z) \ w1(x)$$

$$S11 : w0(x) \ w0(z) \ r2(x) \ r1(x) \ r1(z) \ w1(x) \ r3(z) \ w3(z)$$

Lo schedule $S10$ è Conflict-Equivalente a uno schedule seriale $S11$.

Questi due schedule intanto presentano le stesse operazioni (vedi le disposizioni dei colori).

Poi evidenziamo le operazioni in conflitto:

| SCHEDULE | OPERAZIONE1 | OPERAZIONE2 |
|----------|-------------|-------------|
| S10 | w0(x) | r1(x) |
| | w0(x) | r2(x) |
| | w0(x) | w1(x) |
| | w0(z) | r1(z) |
| | w0(z) | r3(z) |
| | w0(z) | w3(z) |
| | r1(z) | w3(z) |
| | r2(x) | w1(x) |
| S11 | w0(x) | w1(x) |
| | w0(x) | r1(x) |
| | w0(x) | r2(x) |
| | w0(z) | r1(z) |
| | w0(z) | r3(z) |
| | w0(z) | w3(z) |
| | r2(x) | w1(x) |
| | r1(z) | w3(z) |

Ogni coppia di operazioni in conflitto è nello stesso ordine nei due schedule.

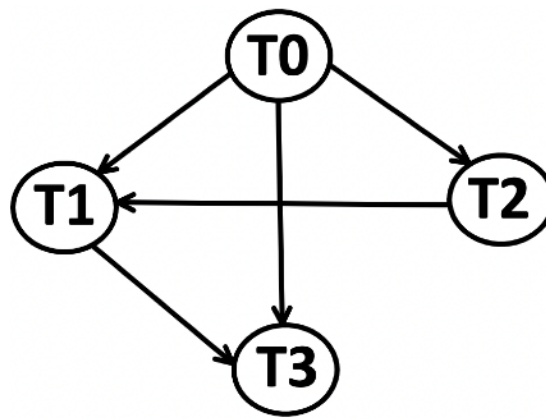
GRAFO ORIENTATO DEI CONFLITTI.

Teorema: È possibile determinare se uno schedule è Conflict-Serializable tramite il grafo dei conflitti. Uno schedule è in CSR se e solo se il suo grafo dei conflitti è aciclico.

Come si costruisce questo grafo?

Il grafo è costruito facendo corrispondere un nodo a ogni transazione.

Si traccia quindi un arco orientato da T_i a T_j se esiste almeno un conflitto tra un'azione "ai" e un'azione "aj", e si ha che "ai" precede "aj";

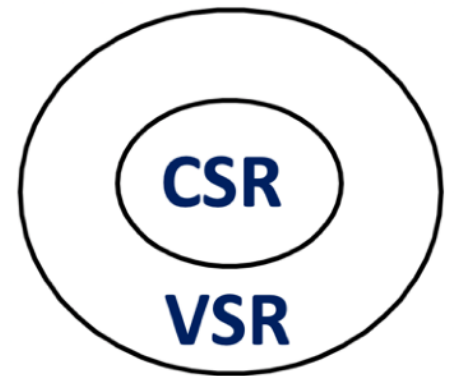


Questo grafo è aciclico: grafo in cui non esiste alcun cammino diretto che parte da un nodo e ritorna a quel nodo stesso.

E questo grafo rappresenta entrambi gli schedule.

LO SPAZIO DI CSR E VSR.

È possibile dimostrare che la classe degli schedule **CSR** è strettamente inclusa in quella degli schedule **VSR**: esistono cioè schedule che appartengono a **VSR** ma non a **CSR**, mentre tutti gli schedule **CSR** appartengono a **VSR**.



Quindi la Conflict-Serializzabilità è condizione sufficiente, ma non necessaria, per la View-Serializzabilità.

Conclusione: S10 è conflict-equivalente a S11, e poiché S11 è seriale (le transazioni sono ordinate senza interleaving), allora: S10 è conflict-serializzabile ed è conflict-equivalente a S11.

(Inoltre il grafo è aciclico, quindi S10 è CSR - Conferma tutto).

LOCKING A DUE FASI : 2PL.

Il protocollo di locking a due fasi (2PL) è un protocollo di controllo della concorrenza utilizzato nei sistemi di gestione delle transazioni per garantire la serializzabilità degli schedule.

Teorema: L'esecuzione di uno schedule "Si" è corretta quando produce lo stesso risultato prodotto da un qualunque schedule seriale "Sj" delle stesse transazioni, in quel caso diremo che "Si" è SERIALIZZABILE.

Come funziona?

Tutte le operazioni di lettura e scrittura devono essere protette tramite l'esecuzione di opportune primitive: **r_lock**, **w_lock** e **unlock**.

Nell'esecuzione di operazioni di lettura e scrittura si devono rispettare i seguenti vincoli:

1. Ogni operazione di **lettura** deve essere preceduta da un "**r_lock**" e seguita da un "**unlock**";
2. Ogni operazione di **scrittura** deve essere preceduta da un "**w_lock**" e seguita da un "**unlock**";

Nota: Una transazione, dopo aver rilasciato un lock, non può acquisirne altri.

Lo scheduler *riceve una sequenza di richieste di esecuzione* di queste **primitive** da parte delle transazioni, e ne determina la correttezza con una semplice ispezione di una struttura dati.

Ogni richiesta di lock che perviene allo scheduler (ossia il lock manager) è caratterizzata solo dall'identificativo della transazione che fa la richiesta, e dalla risorsa per la quale la richiesta viene effettuata.

RICAPITOLIAMO:

$r_lock(x) \rightarrow$ richiesta di lock in lettura su x.
 $w_lock(x) \rightarrow$ richiesta di lock in scrittura su x.
 $unlock(x) \rightarrow$ rilascio del lock su x.

Ogni operazione di lettura/scrittura va **protetta** da lock:

| Operazione | Deve essere preceduta da | E seguita da |
|------------|--------------------------|--------------|
| $r(x)$ | $r_lock(x)$ | $unlock(x)$ |
| $w(x)$ | $w_lock(x)$ | $unlock(x)$ |

Esempio:

T1: $r_lock(x) \rightarrow r(x) \rightarrow unlock(x)$

T2: $w_lock(x) \rightarrow w(x) \rightarrow unlock(x)$

Lo scheduler concede a T1 il $r_lock(x)$ perché nessuno sta scrivendo.
T2 chiede un $w_lock(x)$, ma **viene bloccata** finché T1 non fa $unlock(x)$.

Dopo $unlock(x)$, T2 ottiene il lock

Esempio:

T3: $r_lock(x) \rightarrow r(x) \rightarrow unlock(x) \rightarrow w_lock(y)$

Errore!: T3 non può più richiedere $w_lock(y)$ dopo aver fatto $unlock(x)$,
violazione del 2PL.

FASE CRESCENTE - FASE CALANTE.

. 1. Fase di crescita (growing phase)

- La transazione può **acquisire lock** (su oggetti che vuole leggere o scrivere)
- **Non può rilasciare** alcun lock in questa fase

Esempio: T1 prende lock su x, poi su y

2. Fase di decrescita (shrinking phase)

- La transazione **rilascia i lock**
- Da questo momento in poi **non può più acquisire nuovi lock**

Esempio: T1 rilascia lock su x \rightarrow ora può solo rilasciare, non prenderne di nuovi

Esempio:

T1: lock(x) r(x) lock(y) w(y) unlock(x) unlock(y)

Prima prende tutti i lock, poi li rilascia → rispetta 2PL.

QUESTO È IL 2PL CLASSICO: Per avere la garanzia che le transazioni seguano uno schedule serializzabile è necessario porre una restrizione sull'ordinamento delle richieste di lock, che prende il nome di 2PL, ossia una transazione dopo aver rilasciato un lock, non può acquisirne altri.

Teorema: Ogni schedule che rispetti i requisiti del protocollo di 2PL risulta uno schedule serializzabile rispetto alla Conflict-Equivalenza, ovvero la classe 2PL è contenuta nella classe CSR.

Teorema: Si può dimostrare che le classi 2PL e CSR non sono equivalenti, 2PL è inclusa strettamente in CSR.

Per dimostrarlo basta mostrare uno schedule non in 2PL ma in CSR, come il seguente:

S12 : r1(x) w1(x) r2(x) w2(x) r3(y) w1(y)

In questo caso la transazione T1 deve cedere un lock sulla risorsa X e successivamente richiedere un lock sulla risorsa Y, risultando pertanto non a due fasi.

Però, lo schedule è CSR rispetto alla sequenza T3, T1, T2.

Proof.

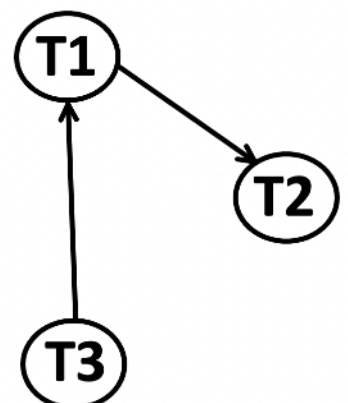
Cioè: dobbiamo verificare se è possibile riordinare le transazioni senza cambiare l'ordine dei conflitti, in modo che lo schedule sia equivalente a quello seriale T3 → T1 → T2.

Vediamo i conflitti tra transazioni diverse.

Su X:

- w1(x) vs r2(x)
w1(x) vs w2(x)

Su Y:



- $r3(y) \text{ vs } w1(y) \rightarrow T3 \rightarrow T1$

Il grafo è lineare e coerente con l'ordine $T3 \rightarrow T1 \rightarrow T2$

Quindi lo schedule è conflict-serializzabile rispetto a quella sequenza.

In conclusione, possiamo dire che la classe 2PL è strettamente contenuta nella classe CSR.

PREVENZIONE DI UN AGGIORNAMENTO FANTASMA TRAMITE L'USO DEI LOCKING A DUE FASI (2PL).

Riprendiamo l'esempio della lettura fantasma.

| T1 | T2 |
|---|---|
| $r1(x)$ $r1(y)$ $r1(z)$ $s = x + y + z$ <u>commit</u> | $r2(y)$ $y = y - 100$ $r2(z)$ $z = z + 100$ $w2(y)$ $w2(z)$ <u>commit</u> |

FASI:

$r1(x)$

$r1(y) \leftarrow$ Legge y **prima** che T2 la modifichi e scriva la nuova y
 > T2 cambia y e z e fa commit

$r1(z) \leftarrow$ Legge z **dopo** che T2 ha scritto il nuovo valore.

T1 ha letto:

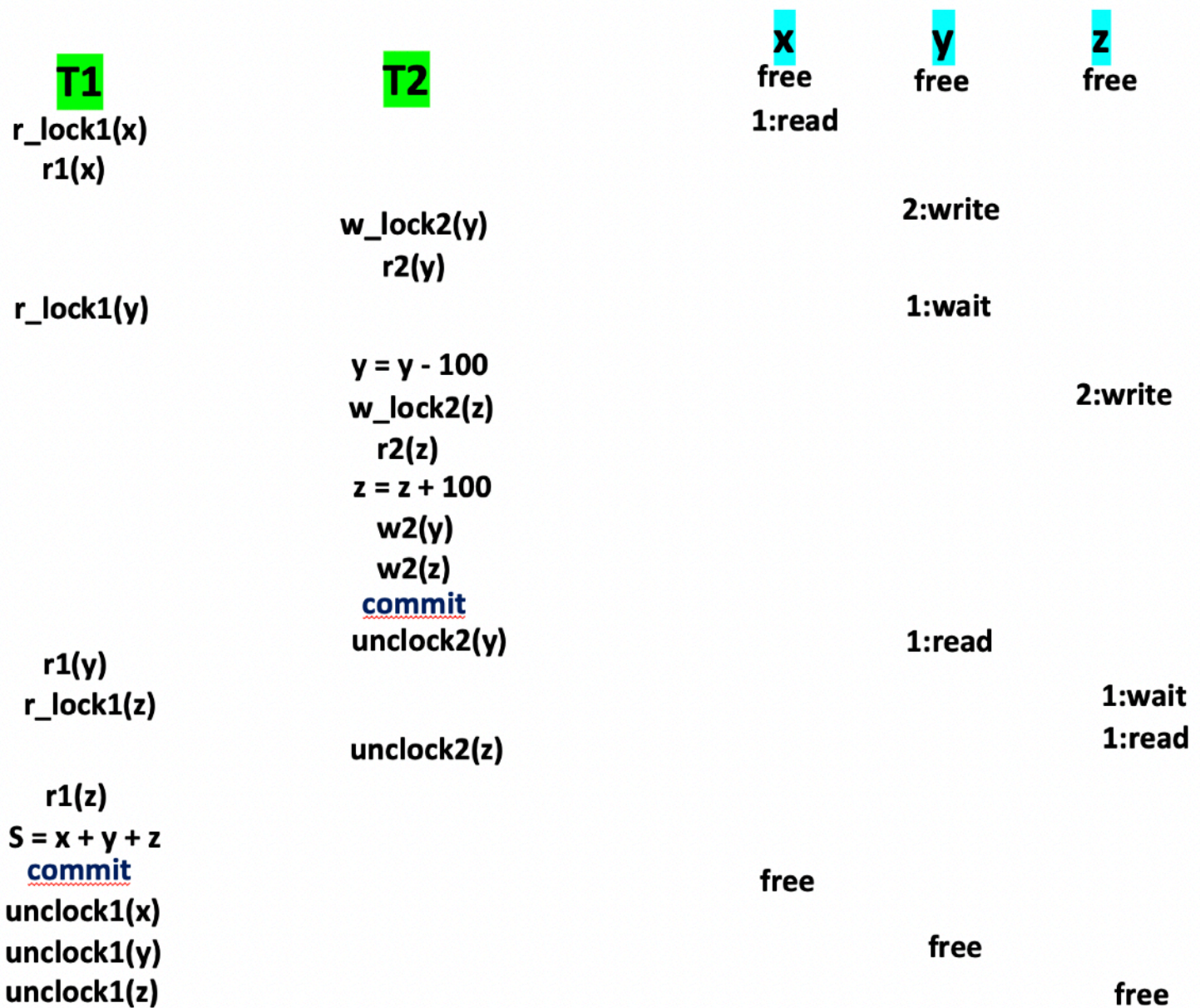
x : corretto

y : il valore vecchio

z : il valore nuovo ($z + 100$)

Quindi la somma che calcola T1 è falsata:

Legge y vecchio + z nuovo \rightarrow somma = 1100



Prendiamo l'esempio dell'aggiornamento fantasma e mostriamo che il 2PL risolve il problema.

Per ogni risorsa, viene indicato il suo stato libero come "FREE", bloccato in lettura dalla i-esima transazione "i:read", bloccato in scrittura dalla i-esima transazione "i:write", posta in stato di attesa "i:wait".

Ho:

- **T1** che vuole leggere **x**, **y**, **z** e calcolare $S = x + y + z$
- **T2** che modifica **y** e **z** mantenendo la somma invariata

Senza controllo: T1 potrebbe leggere **y** **prima** della modifica, e **z** **dopo**, ottenendo somma = **1100**.

IL 2PL RISOLVE QUESTO PROBLEMA.

ESECUZIONE PUNTO PUNTO DEL 2PL SULL'ESEMPIO:

Le transazioni eseguono:

T1:

1. `r_lock1(x)` → blocca `x` in lettura (stato `1:read`)
2. `r1(x)` → legge `x`
3. `r_lock1(y)` → **vuole bloccare `y` in lettura**, ma:

T2 (nel frattempo):


1. `w_lock2(y)` → blocca `y` in scrittura (stato `2:write`)
2. `r2(y)` → legge `y`
3. modifica `y = y - 100`
4. `w_lock2(z)` → blocca `z` in scrittura (stato `2:write`)
5. `r2(z)` → legge `z`
6. modifica `z = z + 100`
7. `w2(y), w2(z)` → MODIFICA `y` e `z`.
8. `commit`
9. `unlock2(y)` → libera `y`
10. `unlock2(z)` → libera `z`

Intervento del 2PL

Torniamo a **T1** che era in attesa:

- Quando T1 prova a fare `r_lock1(y)`, trova che `y` è bloccato in scrittura da T2 → **stato `1:wait`**
- Quindi **T1 si blocca** e aspetta il rilascio, quindi quando riparte t1 legge un valore modificato di Y, cosa che prima non succedeva (OCCHIO EH).
- Solo **dopo che T2 ha rilasciato tutto (fine fase crescente + calante)**, T1 può continuare

Ora T1 fa:

- `r_lock1(y)` → concesso ASSOLUTAMENTE
- `r1(y)` → legge **valore aggiornato**
- `r_lock1(z)` → concesso
- `r1(z)` → legge **valore aggiornato**
- Calcola `s = x + y + z = 1000` 
- `commit`
- `unlock(x, y, z)`

COSA DIMOSTRA QUESTO

1. **T1 ha aspettato** di acquisire tutti i lock → **nessun valore parziale**
2. Ha letto `x, y, z` in uno **stato coerente**
3. **Il vincolo $x + y + z = 1000$ è rispettato**
4. Grazie a 2PL, **non è stato possibile** leggere `y` vecchio e `z` nuovo → **niente aggiornamento fantasma**

In pratica, con questo vincolo i lock vengono rilasciati solo al termine della transazione, dopo che ciascun dato è stato portato nel suo stato finale.

Adesso voglio far correttamente notare un grandissimo e immenso risultato.
Prima di chiudere questa sezione del 2PL voglio parlare dei lock di predicato.

Il livello di isolamento serializable applica il 2PL stretto e i lock di predicato.

I lock di predicato sono degli specifici lock che impediscono l'accesso (e la scrittura) su dati che soddisfano il predicato, ed è per questo che nel livello di isolamento serializable si impedisce la modifica di valori che possono entrare a far parte di una specifica condizione. Questo accade anche su un singolo elemento, se una transazione sta leggendo da un elemento A, arriva una seconda transazione che prova a modificare l'elemento B, comunque quella riga è bloccata e la modifica viene completamente impedita.

Invece nel livello REPEATABLE READ quella modifica non viene bloccata, anzi, viene

eseguita, ma la differenza è che la transazione che leggeva da A vedrà la modifica solo alla fine della transazione stessa: quindi dentro la sua transazione avrà a disposizione sempre lo stesso valore.

Su REPEATABLE READ la modifica parte - Su SERIALIZABLE non parte proprio.

Vedi P.398, è ottimo come approfondimento. (Questo non credo sia importante ai fini dell'esame, però un ingegnere informatico dovrebbe conoscerlo come cultura personale).

FINE 2PL.

(Se tutto è chiaro, siamo sulla buona strada).

CONTROLLO DI CONCORRENZA BASATO SUI TIMESTAMP.

Illustriamo infine un altro metodo per il controllo di concorrenza assai semplice da realizzare: questo metodo utilizza i **TIMESTAMP**.

Definizione di TIMESTAMP: i TIMESTAMP sono identificatori associati a ogni evento temporale.

Il timestamp viene generato leggendo il valore dell'orologio di sistema al momento in cui è avvenuto l'evento.

IL METODO TS.

Il controllo di concorrenza mediante timestamp (metodo TS) avviene nel seguente modo:

1. A ogni transazione si associa un timestamp che rappresenta il momento di inizio della transazione.
2. Si accetta uno schedule solo se esso riflette l'ordinamento seriale delle transazioni in base al valore del timestamp di ciascuna transazione.

Questo metodo di controllo di concorrenza impone che le transazioni risultino serializzate in base all'ordine in cui esse acquisiscono il loro timestamp.

Ad ogni oggetto X vengono associati due indicatori, $WTM(X)$ e $RTM(X)$, che sono rispettivamente i timestamp della transazione che ha eseguito l'ultima scrittura, e della transazione con T più grande che ha letto X .

Allo scheduler arrivano le richieste di accesso agli oggetti del tipo $R_t(x)$ o $W_t(x)$, dove t rappresenta il timestamp della transazione che esegue la lettura o la scrittura.

Lo scheduler non fa altro che permettere o no l'operazione, secondo la seguente politica:

- $R_t(x)$: se $t < WTM(X)$ **la transazione viene uccisa**, altrimenti la richiesta viene accettata; in tal caso $RTM(X)$ viene aggiornato e posto pari al massimo tra $RTM(X)$ e T .
- $W_t(x)$: se $t < WTM(X)$ o $t < RTM(X)$ **la transazione viene uccisa**, altrimenti la richiesta viene accettata; in tal caso $WTM(X)$ viene aggiornato e posto pari a t .

ESEMPIO PER CHI NON STA CAPENDO.

Si supponga che $RTM(X) = 7$ e $WTM(X) = 5$.

$RTM(X) = 7 \rightarrow$ l'oggetto x è stato letto fino ad ora da transazioni con
timestamp ≤ 7 ;

$WTM(X) = 5 \rightarrow$ l'ultima scrittura su x è avvenuta da una transazione con
timestamp 5 ;

Nel seguito, descriviamo la risposta dello scheduler alle richieste di lettura e scrittura ricevute:




| Richieste | Risposte | Nuovi Valori |
|-----------|----------|---------------|
| $r6(x)$ | Ok | |
| $r7(x)$ | Ok | |
| $r9(x)$ | Ok | $RTM(X) = 9$ |
| $w8(x)$ | No | T8 UCCISA |
| $w11(x)$ | Ok | $WTM(X) = 11$ |
| $r10(x)$ | No | T10 UCCISA |

Il metodo TS comporta l'uccisione di un gran numero di transazioni.




| Richiesta | Risposta | Motivazione |
|-----------|--------------|---|
| $r6(x)$ | Ok | $6 \geq WTM(5) \rightarrow ok (t = 6 = \text{timestamp})$ |
| $r7(x)$ | Ok | $7 \geq WTM(5) \rightarrow ok$ |
| $r9(x)$ | Ok | $9 \geq WTM(5) \rightarrow ok \rightarrow \text{aggiorna } RTM(X) = 9$ |
| $w8(x)$ | ✗ T8 UCCISA | $TS=8 < RTM=9 \rightarrow$ non può scrivere perché qualcuno più "recente" ha già letto |
| $w11(x)$ | Ok | $TS=11 > RTM=9$ e $> WTM=5 \rightarrow$ può scrivere \rightarrow aggiorna $WTM(X) = 11$ |
| $r10(x)$ | ✗ T10 UCCISA | $TS=10 < WTM=11 \rightarrow$ sta leggendo un dato sovrascritto da una transazione più "nuova" |

Applichiamo il procedimento.




Arriva $r6(x)$:

- $T = 6$
- $6 < 5?$  \rightarrow  lettura accettata
- Aggiorna: $RTM(X) = \max(7, 6) = 7 \rightarrow$  **rimane 7**

Arriva $r7(x)$:

- $T = 7$
- $7 < 5?$  \rightarrow  lettura accettata
- Aggiorna: $RTM(X) = \max(7, 7) = 7 \rightarrow$  **rimane 7**

Arriva $r9(x)$:

- $T = 9$
- $9 < 5?$  \rightarrow  lettura accettata
- Aggiorna: $RTM(X) = \max(7, 9) = 9 \rightarrow$  **diventa 9**

Quindi, adesso abbiamo i valori così modificati:

$TM(X) = 9$ (aggiornato da $r9(x)$).

$WTM(X) = 5$ (non è cambiato da inizio).

Arriva $w8(x)$:

- $T = 8$

- È una **scrittura**, quindi:

- Devo verificare:

$$\blacksquare \quad 8 < WTM(X) ? \rightarrow 8 < 5 \rightarrow \text{NO}$$

$$\blacksquare \quad 8 < RTM(X) ? \rightarrow 8 < 9 \rightarrow \text{SÌ} \rightarrow \text{T8 UCCISA}$$

T8 è più vecchia di una transazione (T9) che ha già letto x.

Se T8 scrivesse ora, T9 avrebbe letto un valore che non rappresenta più uno stato coerente, cioè uno "sporco futuro". Per garantire l'ordine temporale, T8 viene abortita (uccisa).

Arriva w11(x):

- T = 11
- Verifica:

$$\circ \quad 11 < WTM(X) ? \rightarrow 11 < 5 \rightarrow \text{NO}$$

$$\circ \quad 11 < RTM(X) ? \rightarrow 11 < 9 \rightarrow \text{NO}$$

⇒  **Scrittura consentita**

Viene aggiornata: WTM(X) = 11

⇒ **T11 scrive x**, aggiornando il timestamp di scrittura.

Quindi, adesso abbiamo i valori così modificati:

TM(X) = 9 and WTM(X) = 11.

Arriva r10(x):

- T = 10
- È una lettura, quindi:

$$\circ \quad 10 < WTM(X) ? \rightarrow 10 < 11 \rightarrow \text{SÌ} \rightarrow \text{T10 UCCISA}$$

T10 prova a leggere x, ma nel frattempo T11 (più giovane) ha già scritto su x.

Per non leggere un dato che proviene da una transazione futura, T10 viene abortita.

Vale questa uguaglianza: timestamp più piccoli = transazioni più vecchie.

Ricapitoliamo al volo e disbrigo pratiche finali.

Il controllo della concorrenza basato su timestamp è una tecnica ottimistica (senza lock), che garantisce serializzabilità dei dati senza bisogno di bloccare le risorse.

In pratica:

- Ogni transazione riceve un **timestamp univoco** all'inizio;
- Il sistema garantisce che l'esecuzione **rispetti l'ordine temporale dei timestamp**;
- Se una transazione cerca di leggere o scrivere un dato **fuori ordine**, viene **abortita**;

“Ogni transazione non può leggere o scrivere un dato scritto da una transazione con timestamp superiore, e non può scrivere un dato che è già stato letto da una transazione con timestamp superiore”

Fine timestamp.

REGOLA DI Robert H. Thomas

Una variante del metodo TS prevede l'uso della cosiddetta “regola di Thomas”, la quale specifica un comportamento leggermente diverso per le operazioni di scrittura.

La regola è la seguente.

Per un'operazione di scrittura $W_t(x)$:

1. Se $t < RTM(x) \rightarrow \text{✗}$ la transazione viene abortita.
2. Altrimenti, se $t < WTM(x) \rightarrow \text{✓}$ la scrittura viene scartata (ignorata), ma la transazione continua.
3. Altrimenti $\rightarrow \text{✓}$ la scrittura viene accettata, e $WTM(x)$ viene aggiornato a t .

In sostanza, se la tua scrittura è vecchia e **non serve più**, scartala pure ma **non uccidere la transazione**.

La Regola di Thomas appare nel contesto della timestamp ordering già nel 1977, in un importante articolo accademico >

A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases

ROBERT H. THOMAS
Bolt Beranek and Newman, Inc.

È possibile visitare il documento del 1977 a questo [link](#).

L'idea che c'è dietro è che si possa evitare di eseguire una scrittura su un oggetto che è già stato modificato da una transazione più giovane e che non è stato ancora letto da una transazione più giovane.

MULTIVERSIONI (MVCC).

Un'altra variante del metodo è l'uso delle **multiversioni**.

Questa regola consiste nel mantenere diverse copie degli oggetti della base di dati, per ogni transazione che modifica la base di dati.

Ogni volta che una transazione scrive un oggetto, la vecchia copia non viene persa, ma una nuova N-esima copia viene creata, con un corrispondente WTM N (X).

Si ha invece un solo RTM(X) globale.

WTM_n(X) = timestamp della versione più nuova di X!

Quindi, in un generico istante sono attive $N \geq 1$ copie di ciascun oggetto X.

Con questo metodo, le richieste di lettura non vengono mai rifiutate, ma vengono dirette alla versione dei dati corretta rispetto al timestamp della transazione richiedente.

Le copie vengono rilasciate quando sono divenute inutili, in quanto non esistono più transazioni in lettura interessate al loro valore.

In sintesi:

MVCC è una variante del timestamp ordering in cui:

- Ogni scrittura crea una **nuova versione** del dato
- Ogni lettura accede alla **versione più recente visibile** rispetto al timestamp della transazione.
- Nessuna lettura viene mai **rifiutata**, perché ogni transazione legge solo ciò che "esisteva" al momento in cui è iniziata.

Le regole di comportamento diventano:

- $R_t(x)$: una lettura è sempre accettata. Si legge un X_k siffatto.
Se $t > WTM_N(X)$, allora $k=N$;
Altrimenti si prende "i" in modo che sia : $WTM_i(X) < t < WTM_{i+1}(X)$;
- $W_t(x)$: se $t < RTM(X)$ si rifiuta la richiesta, altrimenti si aggiunge una nuova versione del dato (N cresce di 1) con $WTM_N(X) = t$.

ESEMPIO CON LETTURE.

Supponiamo di avere 3 versioni dell'oggetto X:

| Versione | WTM _i (X) |
|----------------|----------------------|
| X ₁ | 5 |
| X ₂ | 10 |
| X ₃ | 20 |

$N = 3$ (perché ci sono 3 versioni).

$WTM_3(X) = 20 \rightarrow$ la scrittura più recente.

Facciamo 3 richieste di lettura $R_t(x)$ con timestamp diversi:

Primo caso: $r_{15}(x)$ (T15 legge x).

Timestamp $t = 15$

Confrontiamo:

- $15 > WTM_3(x)? \rightarrow 15 > 20$ **✗** NO
- Cerchiamo i tale che $WTM_i(x) < 15 < WTM_{i+1}(x)$: $10 < 15 < 20$ **✓** $\rightarrow i = 2$.

Succede che **T15 legge x_2** , la versione con $WTM_2(x) = 10$.

Secondo caso: $r_{25}(x)$ (T25 legge x).

Timestamp $t = 25$

Confrontiamo:

$25 > 20 = WTM_3(x)$ **✓**, allora $k = 20$.

Leggo x_3 , **leggo x_n** , quindi $i = N = 3$.

Terzo Caso: $r_8(x)$ (T8 legge x).

- Timestamp $t = 8$
- $8 > 20?$ **✗** NO
- Cerco i tale che $WTM_i(x) < 8 < WTM_{i+1}(x)$

Verifico:

- $5 < 8 < 10 \checkmark \rightarrow i = 1$

✓ T8 legge x_1 , la versione con $WTM_1(x) = 5$.

Riepilogo finale:

| Transazione | t | Versione letta | Perché |
|-------------|----|----------------|----------------------------------|
| T8 | 8 | x_1 | $5 < 8 < 10 \rightarrow i = 1$ |
| T15 | 15 | x_2 | $10 < 15 < 20 \rightarrow i = 2$ |
| T25 | 25 | x_3 | $25 > 20 \rightarrow i = N = 3$ |

ESEMPIO CON SCRITTURE

| Versione | $WTM_i(x)$ |
|----------|------------|
| x_1 | 5 |
| x_2 | 10 |
| x_3 | 20 |

Valori aggiuntivi:

- $RTM(x) = 18$
Significa che la **lettura più recente** è stata fatta da **T18**.

Caso 1: $w15(x) \rightarrow T15$ vuole scrivere

- $t = 15, RTM(x) = 18$
- $15 < 18 \rightarrow \text{✗ T15 viene rifiutata / abortita}$

Motivo: T15 è **più vecchia di una lettura già avvenuta**:
non può modificare dati letti da transazioni più nuove.

Caso 2: $w25(x) \rightarrow T25$ vuole scrivere

- $t = 25, RTM(x) = 18$
- $25 > 18 \rightarrow \text{✓ OK}$

Crea nuova versione: x_4 con $WTM_4(x) = 25$.
N = 4 ora.

| Versione | $WTM_i(x)$ |
|----------|------------|
| x_1 | 5 |
| x_2 | 10 |
| x_3 | 20 |
| x_4 | 25 |

Caso 3: $w_{10}(x) \rightarrow T_{10}$ vuole scrivere

- $t = 10, RTM(x) = 18$
- $10 < 18 \rightarrow \text{❌ rifiutata}$

T_{10} è **troppo vecchia**, non può scrivere: sarebbe come modificare dati già letti da transazioni più nuove.

Caso 4: $w_{30}(x) \rightarrow T_{30}$ vuole scrivere

- $t = 30 > 18 \rightarrow \text{✅ OK}$

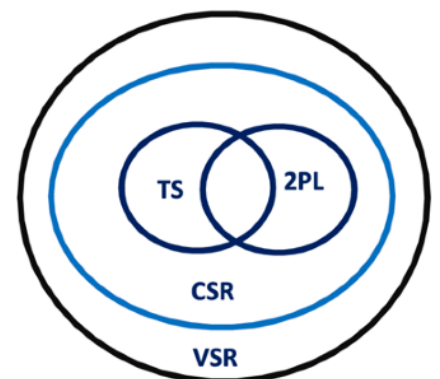
Crea nuova versione: x_5 , con $WTM_5(x) = 30$.
N = 5

| Versione | $WTM_i(x)$ |
|----------|------------|
| x_1 | 5 |
| x_2 | 10 |
| x_3 | 20 |
| x_4 | 25 |
| x_5 | 30 |

CONFRONTO FINALE FRA VSR, CSR, 2PL E TS.

Si osserva che la classe VSR è la classe più generale: essa include strettamente al suo interno CSR, la quale a sua volta include sia la classe 2PL sia la classe TS.

2PL e TS a loro volta presentano un'intersezione non nulla, ma nessuna presenta una relazione di inclusione con l'altra. Quest'ultima può essere verificata facilmente, costruendo schedule che sono in TS ma non in 2PL, oppure in 2PL ma non in TS, o infine in 2PL e in TS.



Esistono schedule che sono in TS ma non in 2PL:

S13 : r1(x) w2(x) r3(x) r1(y) w2(y) r1(v) w3(v) r4(v) w4(y) w5(y)

Questo schedule è in CSR poiché il suo grafo dei conflitti mostra l'assenza di ciclicità.

L'ordinamento seriale delle transazioni Conflict-Equivalenti allo schedule di partenza è T1,T2,T3,T4,T5.

Lo schedule NON risulta essere in 2PL in quanto T2 prima rilascia X affinché venga letto da T3, e poi acquisisce Y, rilasciato da T1.

E siccome una transazione, dopo aver rilasciato un lock, non può acquisirne altri, questo schedule non è in 2PL.

Lo schedule è in TS, poiché presso ogni oggetto le transazioni operano nell'ordine indotto dai timestamp.

Esempio schedule sia in TS sia in 2PL:

S14 : r1(x) w1(x) r2(x) w2(x)

T1:

lock1(x) // fase crescente

r1(x)

w1(x)

unlock1(x) // fase calante

T2:

lock2(x) // fase crescente

r2(x)

w2(x)

unlock2(x) // fase calante

Quindi sì, è in 2PL, e in TS.

Esempio di schedule che non appartiene a TS ma in 2PL.

S15 : r2(x) w2(x) r1(x) w1(x).

lock1(x) // fase crescente

r1(x)

w1(x)

unlock1(x) // fase calante

Ok 2PL.

Ma non in TS poichè T2 ha un timestamp più nuovo ma accede a **x** prima di T1 → **violazione**.

FINE.