

Tecnologia dei DB server (centralizzati e distribuiti)

Affidabilità, Lock Gerarchici

Transazioni “acide”

Controllo di Concorrenza

- Schedule Seriali
- VSR, CSR, 2PL, 2PL,
TS-Singolo, TS-Multi

Controllo di Affidabilità

- Log e dump del sistema
- Ripresa a caldo/freddo

Garantisce:

- I: Isolation

Garantisce:

- A: Atomicity
- D: Durability

C: Consistency è garantita a livello di compilazione delle query

Record del Log

Record di Transazione

Registrano le azioni delle transazioni in ordine

Begin (T)

Insert (T, O, A)

Delete (T, O, B)

Update (T, O, A, B)

Commit (T)

Abort (T)

Record di Sistema

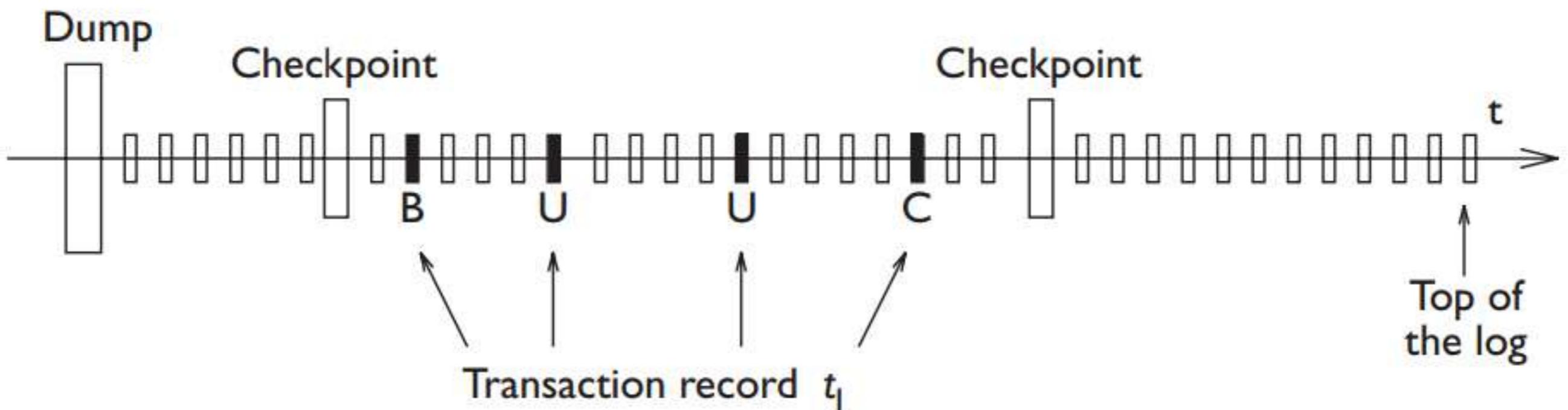
Registrano operazioni del controllore dell'affidabilità

Checkpoint (T1, T2, ...)

Dump ()

T = Transazione, O = Oggetto,
B = Before State, A = After State

Esempio di Log



Primitive per l'affidabilità

UNDO(Action)

- Update e Delete:
 - **O = B**
- Insert:
 - **Remove O**

REDO(Action)

- Update e Insert:
 - **O = A**
- Delete:
 - **Remove O**

Idempotenza:

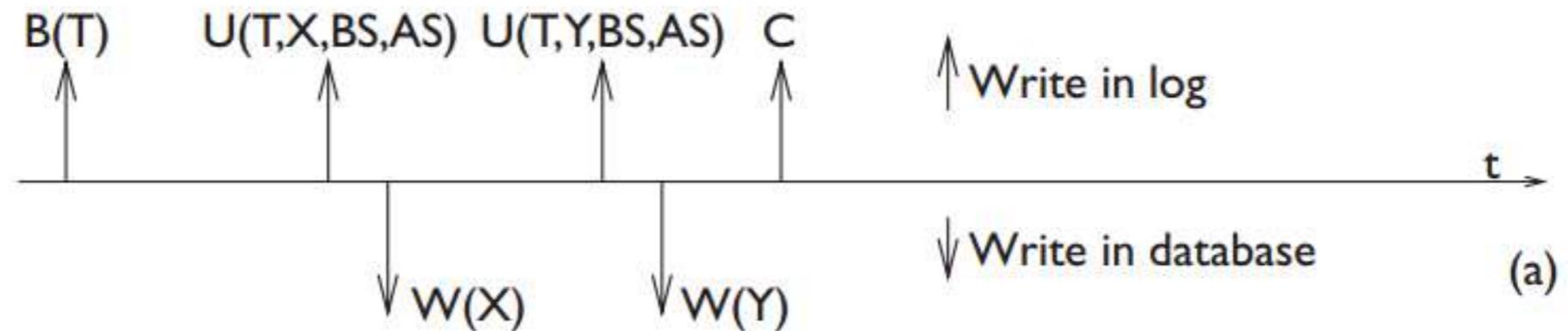
$$\text{UNDO}(\text{ UNDO}(\text{ Action })) = \text{UNDO}(\text{ Action })$$

$$\text{REDO}(\text{ REDO}(\text{ Action })) = \text{REDO}(\text{ Action })$$

Regole per la scrittura del log

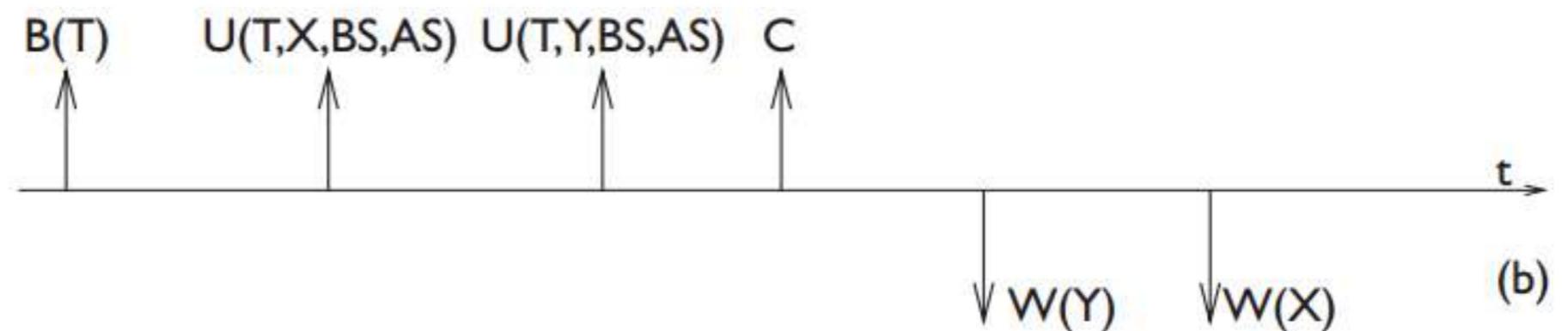
WAL

(Solo Undo)



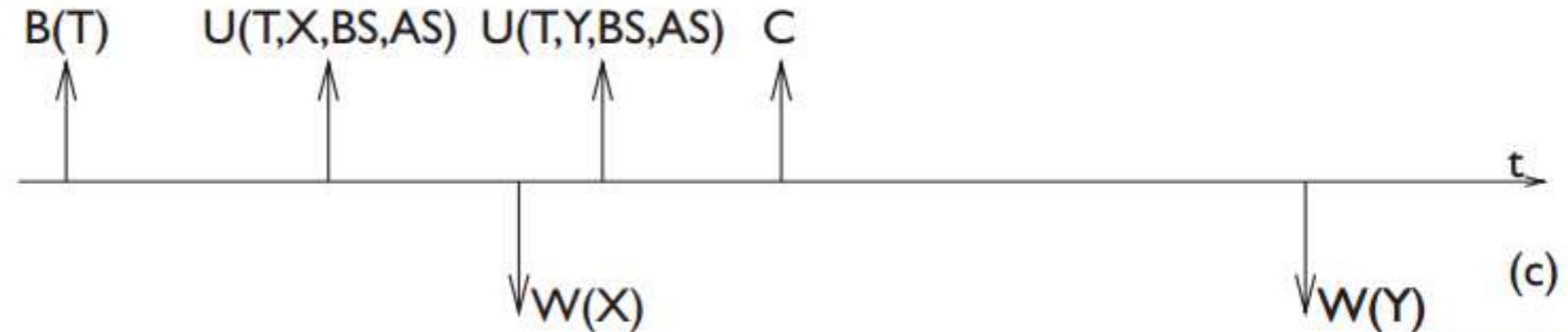
Commit-prec

(Solo Redo)



Misto

(Undo + Redo)



Gestione Guasti

- Guasti di Sistema:
 - Sistema inconsistente per bug, cali di tensione,..
 - Perdita memoria centrale, memoria di massa OK
 - Ripresa a Caldo
- Guasti di Dispositivo:
 - Perdita contenuto memoria di massa.
 - Memoria stabile OK
 - Ripresa a Freddo

Ripresa a Caldo

1. UNDO = Ultimo Checkpoint, REDO = {}

Ripresa a Caldo

1. UNDO = Ultimo Checkpoint, REDO = {}
2. Dall'ultimo Checkpoint:
 - Begin(T): UNDO += T
 - Commit(T): UNDO -= T, REDO += T

Ripresa a Caldo

1. UNDO = Ultimo Checkpoint, REDO = {}
2. Dall'ultimo Checkpoint:
 - Begin(T): UNDO += T
 - Commit(T): UNDO -= T, REDO += T
3. Disfacimento a ritroso azioni in UNDO

Ripresa a Caldo

1. UNDO = Ultimo Checkpoint, REDO = {}
2. Dall'ultimo Checkpoint:
 - Begin(T): UNDO += T
 - Commit(T): UNDO -= T, REDO += T
3. Disfacimento a ritroso azioni in UNDO
4. Ri-esecuzione in ordine azioni in REDO

F.1

Si abbia nel log di un sistema di gestione di basi di dati centralizzato la sequenza di record:

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),
ckpt(t1,t3,t4), b(t5), b(t6), u(t5,o5,b5,a5), a(t3),
ckpt(t1,t4,t5,t6), b(t7), a(t4), u(t7,o6,b6,a6),
u(t6,o3,b7,a7), b(t8), a(t7), **guasto**

Illustrare i passi da compiere per la **ripresa a caldo** del sistema.

Si ripercorre il log a ritroso fino all'ultimo checkpoint. Si inizializzano gli insiemi UNDO e REDO all'insieme di transazioni contenute nel record di checkpoint e a \emptyset rispettivamente

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), ckpt(t1,t3,t4), b(t5),
b(t6), u(t5,o5,b5,a5), a(t3), **ckpt(t1,t4,t5,t6)**, b(t7),
a(t4), u(t7,o6,b6,a6), u(t6,o3,b7,a7), b(t8), a(t7), guasto

UNDO={t1,t4,t5,t6} REDO={}

Si percorre poi il log da lì in avanti, inserendo in UNDO le transazioni di cui è presente un record di *begin* e spostando da UNDO a REDO quelle che hanno un record di *commit*

...

ckpt(t1,t4,t5,t6), b(t7), a(t4), u(t7,o6,b6,a6),
u(t6,o3,b7,a7), b(t8), a(t7), guasto

UNDO={t1,t4,t5,t6,**t7,t8**}

REDO={}

Si percorre ancora il log *a ritroso*, disfacendo le transazioni in UNDO (con azioni compensative), fino alla prima azione di t1, la più vecchia contenuta in UNDO e REDO

UNDO={ t1, t4, t5, t6, t7, t8} 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), ckpt(t1,t3,t4),
b(t5), b(t6), u(t5,o5,b5,a5), a(t3),
ckpt(t1,t4,t5,t6), b(t7), a(t4),
u(t7,o6,b6,a6), u(t6,o3,b7,a7), b(t8),
a(t7), guasto

Azioni:
o3=b7
o6=b6
o5=b5
o4=b4
o3=b3
delete(o1)

Infine si riapplicano le azioni delle transazioni in REDO, nell'ordine in cui sono registrate nel log (da percorrersi in avanti)

In questo caso REDO è vuoto

UNDO={t1,t4,t5,t6,t7,t8}

REDO={}

F.2 Individuare gli insiemi di Undo e Redo per il ripristino di un sistema caratterizzato dal seguente log:

Dump, b(t1), u(t1,o1,b1,a1), b(t2), b(t3),
u(t3,o3,a3,b3), i(t2,o2,a2), c(t2),
ckpt(t1,t3), c(t3), b(t4),
u(t4,o2,b4,a4), u(t4,o3,b5,a5), b(t5),
i(t5,o6,a6), a(t1), c(t4), u(t5,o7,b7,a7),
d(t5,o1,b8), guasto

Si ripercorre il log a ritroso fino all'*ultimo* checkpoint. Si inizializzano gli insiemi UNDO e REDO all'insieme di transazioni contenute nel record di checkpoint e a \emptyset rispettivamente

Dump, b(t1), u(t1,o1,b1,a1), b(t2), b(t3),
u(t3,o3,a3,b3), i(t2,o2,a2), c(t2), **ckpt(t1,t3)**,
c(t3), b(t4), u(t4,o2,b4,a4), u(t4,o3,b5,a5),
b(t5), i(t5,o6,a6), a(t1), c(t4), u(t5,o7,b7,a7),
d(t5,o1,b8), guasto

UNDO= {t1,t3}

REDO= {}

Si percorre il log da lì in avanti, aggiornando gli insiemi di UNDO e REDO.

I	L	M	N	O
ckpt(t1,t3), c(t3), b(t4), u(t4,o2,b4,a4), u(t4,o3,b5,a5),				
P	Q	R	S	T
b(t5), i(t5,o6,a6), a(t1), c(t4), u(t5,o7,b7,a7),				
d(t5,o1,b8), guasto				
U	V			

- I) UNDO={t1,t3} REDO={}
- L) UNDO={t1} REDO={t3}
- M) UNDO={t1,t4} REDO={t3}
- P) UNDO={t1,t4,t5} REDO={t3}
- S) UNDO={t1,t5} REDO={t3,t4}

Si annullano poi gli effetti delle azioni delle transazioni nell'insieme di UNDO:

- U) Re-insert (o1=b8)
- T) o7=b7
- Q) delete(o6)
- C) o1=b1

E si riapplicano le azioni delle transazioni in REDO:

- F) o3=a3
- N) o2=a4
- O) o3=a5

Ripresa a caldo sistemi distribuiti

- Nel log c'è anche l'azione $r(t)$ che indica lo stato di *ready* della transazione t . Il server locale è pronto ad effettuare il commit di t e sta aspettando una risposta dal transaction manager.
- Se la risposta c'è ($a(t)$ o $c(t)$) si ignora $r(t)$.
- Se la risposta non c'è si ipotizza.

G.1 Applicare il protocollo di ripresa a caldo dopo la caduta di un nodo assumendo un algoritmo di commit a due fasi, a fronte del seguente input (dove $r(t_i)$ indica la presenza di un record *ready*):

b(t1), b(t2), b(t3), i(t1,o1,a1), d(t2,o2,b2)
b(t4), r(t1), u(t4,o3,b3,a3), c(t1),
Ckpt(t2,t3,t4), b(t5), b(t6), u(t5,o5,b5,a5),
r(t5), b(t7), u(t7,o6,b6,a6), b(t8),
u(t6,o1,b7,a7), a(t7), r(t6), guasto

$b(t1), b(t2), b(t3), i(t1,o1,a1), d(t2,o2,b2)$ $b(t4), r(t1),$
 $u(t4,o3,b3,a3), c(t1), \text{Ckpt}(t2,t3,t4), b(t5), b(t6),$
 $u(t5,o5,b5,a5), r(t5), b(t7), u(t7,o6,b6,a6), b(t8),$
 $u(t6,o1,b7,a7), a(t7), r(t6), \text{guasto}$

Analizzando il log dall'ultimo checkpoint in avanti, identifichiamo due transazioni con un record di ready (**t5** e **t6**).

Il recovery manager deve identificare il transaction manager relativo a ciascuna transazione e chiederne l'esito. **SUPPONIAMO** un **commit** per entrambe

Ora si procede come nel caso centralizzato

Si cerca l'ultimo checkpoint e si inizializzano gli insiemi UNDO e REDO, poi si percorre il log in avanti, aggiungendo in UNDO le transazioni che hanno un record di begin e spostando in REDO quelle che hanno un record di commit

b(t1), b(t2), b(t3), i(t1,o1,a1), d(t2,o2,b2) b(t4),
r(t1), u(t4,o3,b3,a3), c(t1), **Ckpt(t2,t3,t4)**, b(t5),
b(t6), u(t5,o5,b5,a5), r(t5), **b(t7)**, u(t7,o6,b6,a6),
b(t8), u(t6,o1,b7,a7), a(t7), r(t6), guasto

UNDO= {t2,t3,t4,t7,t8}

REDO= {t5,t6}

Si percorre il log a ritroso, disfacendo le transazioni in UNDO, fino alla prima azione della transazione più vecchia in UNDO e REDO

Infine si riapplicano le azioni di REDO nell'ordine in cui sono registrate nel log

In seguito all'applicazione delle operazioni di UNDO e REDO il database sarà nello stato

{o1:a7, o2:b2, o3:b3, o5:b5, o6:b6}

G.2

Si consideri il seguente log, trovato su di un nodo di una base di dati distribuita dopo un guasto, in cui $r(t_1)$ indica che t_1 è in stato ready e $lc(t_1)$, $la(t_1)$ indicano gli stati di local commit e di local abort.

A B C D E
b(t_1), b(t_2), u(t_1, o_1, b_1, a_1), u(t_2, o_2, b_2, a_2), b(t_3),
F G H I L M
c(t_2), ckpt(t_1, t_3), b(t_4), r(t_1), b(t_5), i(t_3, o_3, a_3),
N O P Q R
lc(t_1), d(t_4, o_4, b_4), d(t_5, o_1, b_5), r(t_5), u(t_4, o_6, b_6, a_6),
S T U V Z
a(t_3), b(t_6), u(t_6, o_7, b_7, a_7), c(t_6), r(t_4)

b(t1), b(t2), u(t1,o1,b1,a1), u(t2,o2,b2,a2), b(t3), c(t2), **ckpt(t1,t3)**, b(t4),
r(t1), b(t5), i(t3,o3,a3), lc(t1), d(t4,o4,b4),
d(t5,o1,b5), **r(t5)**, u(t4,o6,b6,a6), a(t3), b(t6),
u(t6,o7,b7,a7), c(t6), **r(t4)**

Analizzando il log dall'ultimo checkpoint in avanti,
identifichiamo due transazioni con un record di
ready (**t4** e **t5**) e prive di record di commit o abort.
Il recovery manager deve identificare il transaction
manager per ciascuna e chiedere l'esito.

SUPPONIAMO un **commit** per entrambe

Si cerca l'ultimo checkpoint e si inizializzano gli insiemi UNDO e REDO, poi si percorre il log in avanti, aggiungendo in UNDO chi ha record di begin e si sposta da UNDO a REDO chi ha record di commit

b(t1), b(t2), u(t1,o1,b1,a1), u(t2,o2,b2,a2),
b(t3), c(t2), **ckpt(t1,t3)**, **b(t4)**, r(t1), **b(t5)**,
i(t3,o3,a3), lc(t1), d(t4,o4,b4), d(t5,o1,b5),
r(t5), u(t4,o6,b6,a6), a(t3), **b(t6)**,
u(t6,o7,b7,a7), **c(t6)**, r(t4)

UNDO= {t3} REDO= {t1,t4,t5,t6}

Si percorre il log a rovescio, disfacendo le transazioni in UNDO, fino alla prima azione della transazione più vecchia in UNDO o REDO

UNDO= {**t3**}

REDO= {t1,t4,t5,t6}

b(t1), b(t2), u(t1,o1,b1,a1), u(t2,o2,b2,a2),
b(t3), c(t2), ckpt(t1,t3), b(t4), r(t1), b(t5),
i(t3,o3,a3), lc(t1), d(t4,o4,b4), d(t5,o1,b5),
r(t5), u(t4,o6,b6,a6), a(t3), b(t6),
u(t6,o7,b7,a7), c(t6), r(t4)

Infine si riapplicano le azioni di REDO
nell'ordine in cui sono registrate nel log

UNDO={t3}

REDO={**t1,t4,t5,t6**}

b(t1), b(t2), **u(t1,o1,b1,a1)**, u(t2,o2,b2,a2),
b(t3), c(t2), ckpt(t1,t3), **b(t4)**, r(t1), **b(t5)**,
i(t3,o3,a3), lc(t1), **d(t4,o4,b4)**, **d(t5,o1,b5)**,
r(t5), **u(t4,o6,b6,a6)**, a(t3), **b(t6)**,
u(t6,o7,b7,a7), c(t6), r(t4)

b(t1), b(t2), ~~u(t1,o1,b1,a1)~~, ~~u(t2,o2,b2,a2)~~,
b(t3), c(t2), ckpt(t1,t3), b(t4), r(t1), b(t5),
i(t3,o3,a3), lc(t1), d(t4,o4,b4), ~~d(t5,o1,b5)~~,
r(t5), ~~u(t4,o6,b6,a6)~~, a(t3), b(t6),
~~u(t6,o7,b7,a7)~~, c(t6), r(t4)

In seguito all'applicazione delle operazioni di UNDO e REDO il database sarà nello stato

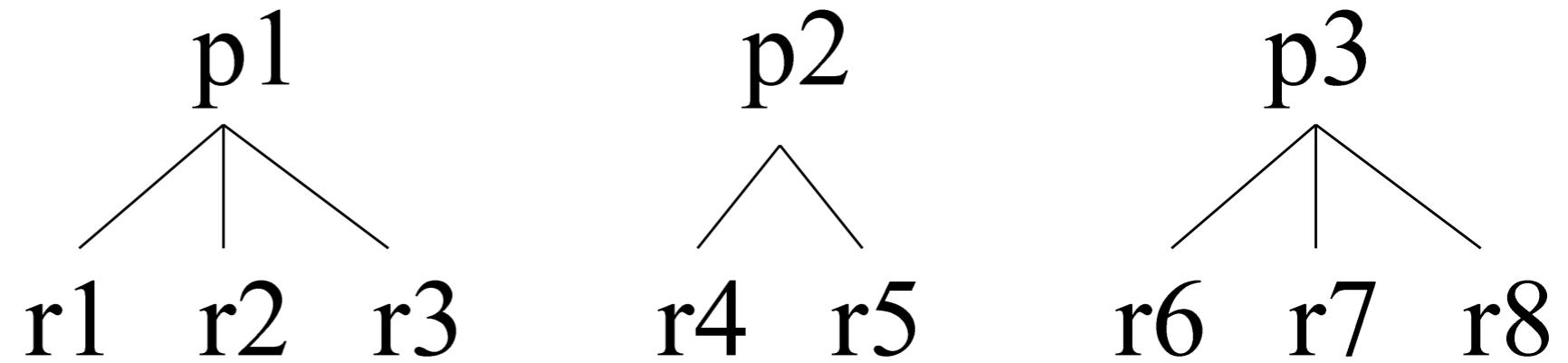
{o1:assente; o2:a2; o3:assente; o4:assente;
o6:a6; o7:a7}

Ripresa a freddo

1. Copia informazioni mancanti dal DUMP
2. Applicazione di tutte le operazioni nel log relative alla parte deteriorata
3. Si effettua una ripresa a caldo

I.1

Data la gerarchia
di risorse:



Si abbia il seguente **LOG**:

A B C D E
b(t1), b(t2), b(t3), u(t1,r1,b1,a1), u(t1,r4,b2,a2),

F G H I
u(t2,r6,b3,a3), u(t2,r5,b4,a4), c(t1), u(t2,r1,b5,a5),

L M N O
u(t3,r4,b6,a6), c(t2), u(t3,r1,b7,a7), c(t3)

Tale log può essere ottenuto da un sistema che segue il protocollo *2PL Strict* con locking gerarchico? (mostrare, anche in modo sintetico, le richieste di lock e unlock delle transazioni).

Occorre scandire il log *bloccando* le risorse subito prima delle azioni e *rilasciandole* solo dopo il commit (o abort).

In parallelo teniamo aggiornata una *tabella delle risorse* per verificare la compatibilità delle richieste con lo stato delle risorse

Conflitti nei lock gerarchici

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

A, B, C)

no richieste

D) $u(t_1, r1, b1, a1)$

$IXL(t_1, p1)$

$XL(t_1, r1)$

E) $u(t_1, r4, b2, a2)$

$IXL(t_1, p2)$

$XL(t_1, r4)$

F) $u(t_2, r6, b3, a3)$

$IXL(t_2, p3)$

$XL(t_2, r6)$

p1	r1	r2	r3	p2	r4	r5	p3	r6	r7	r8
-	-	-	-	-	-	-	-	-	-	-
iX_1										
iX_1	X_1									
iX_1				iX_1						
iX_1	X_1			iX_1	X_1					
iX_1				iX_1			iX_2			
iX_1	X_1			iX_1	X_1		iX_2	X_2		

G) $u(t_2, r5, b4, a4)$

$IXL(t_2, p2)$

$XL(t_2, r5)$

H) $c(t_1)$

$UXL(t_1, r1)$

$UXL(t_1, r4)$

$UIXL(t_1, p1)$

$UIXL(t_1, p2)$

I) $u(t_2, r1, b5, a5)$

$IXL(t_2, p1)$

$XL(t_2, r1)$

p1	r1	r2	r3	p2	r4	r5	p3	r6	r7	r8
\dot{ix}_1	x_1			$\dot{ix}_{1,2}$	x_1		\dot{ix}_2	x_2		
\dot{ix}_1	x_1			$\dot{ix}_{1,2}$	x_1	x_2	\dot{ix}_2	x_2		
\dot{ix}_1	x_1			$\dot{ix}_{1,2}$	x_1	x_2	\dot{ix}_2	x_2		
\dot{ix}_1				$\dot{ix}_{1,2}$	x_1	x_2	\dot{ix}_2	x_2		
\dot{ix}_1				$\dot{ix}_{1,2}$		x_2	\dot{ix}_2	x_2		
				$\dot{ix}_{1,2}$			\dot{ix}_2	x_2		

p1	r1	r2	r3	p2	r4	r5	p3	r6	r7	r8
L) $u(t_3, r4, b6, a6)$										
$IXL(t_3, p2)$	\dot{ix}_2	x_2		$\dot{ix}_{2,3}$		x_2	\dot{ix}_2	x_2		
$XL(t_3, r4)$	\dot{ix}_2	x_2		$\dot{ix}_{2,3}$	x_3	x_2	\dot{ix}_2	x_2		
M) $c(t_2)$										
$UXL(t_2, r1)$	\dot{ix}_2	x_2		$\dot{ix}_{2,3}$	x_3	x_2	\dot{ix}_2	x_2		
$UXL(t_2, r5)$	\dot{ix}_2			$\dot{ix}_{2,3}$	x_3	x_2	\dot{ix}_2	x_2		
$UXL(t_2, r6)$	\dot{ix}_2			$\dot{ix}_{2,3}$	x_3		\dot{ix}_2	x_2		
$UIXL(t_2, p1)$	\dot{ix}_2			$\dot{ix}_{2,3}$	x_3		\dot{ix}_2			
$UIXL(t_2, p2)$				$\dot{ix}_{2,3}$	x_3		\dot{ix}_2			
$UIXL(t_2, p3)$				\dot{ix}_3	x_3		\dot{ix}_2			

N) $u(t_3, r1, b7, a7)$

$IXL(t_3, p1)$

$XL(t_3, r1)$

O) $c(t_3)$

$UXL(t_3, r1)$

$UXL(t_3, r4)$

$UIXL(t_3, p1)$

$UIXL(t_3, p2)$

p1	r1	r2	r3	p2	r4	r5	p3	r6	r7	r8
\dot{ix}_3				\dot{ix}_3	x_3					
\dot{ix}_3	x_3			\dot{ix}_3	x_3					
				\dot{ix}_3	x_3					
				\dot{ix}_3	x_3					
				\dot{ix}_3	x_3					
				\dot{ix}_3	x_3					
				\dot{ix}_3	x_3					
				\dot{ix}_3	x_3					
				\dot{ix}_3	x_3					

NON si sono verificati conflitti: il log è compatibile

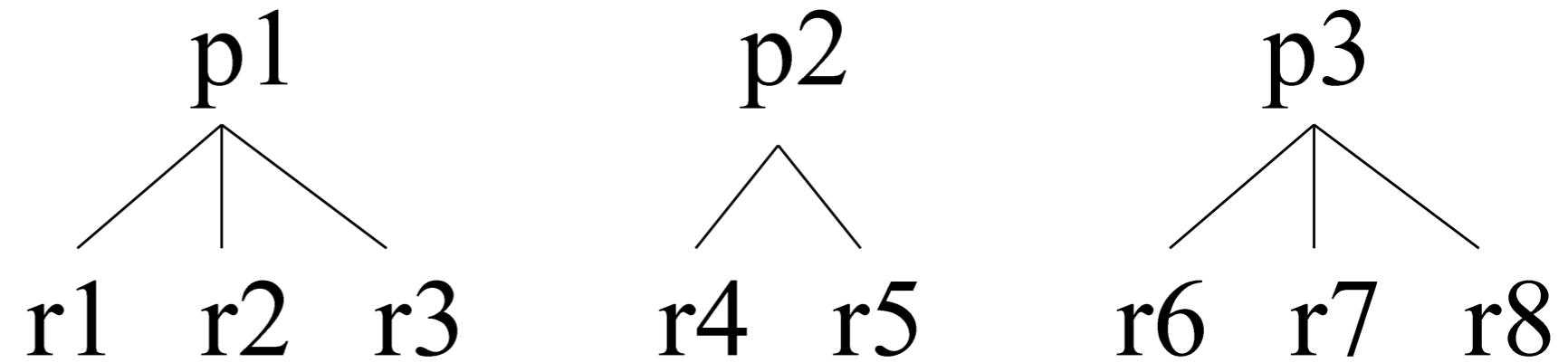
Si ha la garanzia che le transazioni hanno eseguito in modo serializzabile (ovvero, il log è equivalente allo schedule)?

NON si dispone di informazioni sufficienti per rispondere a questa domanda – il log, infatti, *non* fornisce informazioni rispetto ai momenti in cui sono avvenute le *lettura*.

Inoltre, le operazioni di lettura e scrittura (dei dati) possono essere asincrone, ed arbitrariamente lontane (ma successive – cfr. WAL e Com-Prec) rispetto alla scrittura dei record del log.

I.2

Con la stessa gerarchia:



Ma il seguente **LOG**:

A B C D
b(t₁), b(t₂), u(t₁,r4,b1,a1), u(t₁,r2,b2,a2),

E F G H I
u(t₂,r6,b3,a3), u(t₂,r3,b4,a4), u(t₂,r2,b5,a5), c(t₁), c(t₂)

A, B)

no richieste

C) $u(t_1, r4, b1, a1)$

$IXL(t_1, p2)$

$XL(t_1, r4)$

D) $u(t_1, r2, b2, a2)$

$IXL(t_1, p1)$

$XL(t_1, r2)$

E) $u(t_2, r6, b3, a3)$

$IXL(t_2, p3)$

$XL(t_2, r6)$

p1	r1	r2	r3	p2	r4	r5	p3	r6	r7	r8
-	-	-	-	-	-	-	-	-	-	-
ix_1				ix_1	x_1					
ix_1			x_1	ix_1	x_1	ix_1	x_1			
ix_1		x_1		ix_1	x_1	ix_1	x_1	ix_2	x_2	

p1	r1	r2	r3	p2	r4	r5	p3	r6	r7	r8
F) u(t ₂ ,r3,b4,a4)										
IXL(t ₂ ,p1)	ix _{1,2}	x ₁		ix ₁	x ₁		ix ₂	x ₂		
XL(t ₂ ,r3)	ix _{1,2}	x ₁	x ₂	ix ₁	x ₁		ix ₂	x ₂		
G) u(t ₂ ,r2,b5,a5)										
XL(t ₂ ,r2)	ix _{1,2}	x ₁	x ₂	ix ₁	x ₁		ix ₂	x ₂		

N.B. t₁ non ha ancora effettuato il commit

	p1	r1	r2	r3	p2	r4	r5	p3	r6	r7	r8
F) $u(t_2, r3, b4, a4)$											
$IXL(t_2, p1)$	$ix_{1,2}$		x_1		ix_1	x_1		ix_2	x_2		
$XL(t_2, r3)$	$ix_{1,2}$		x_1	x_2	ix_1	x_1		ix_2	x_2		
G) $u(t_2, r2, b5, a5)$											
$XL(t_2, r2)$	$ix_{1,2}$		x_1	x_2	ix_1	x_1		ix_2	x_2		

t_1 non ha ancora effettuato il commit, quindi se t_2 ha ottenuto il lock il log *non può derivare* da un sistema 2PL Strict – non è compatibile.

	p1	r1	r2	r3	p2	r4	r5	p3	r6	r7	r8
F) $u(t_2, r3, b4, a4)$											
$IXL(t_2, p1)$	$ix_{1,2}$		x_1		ix_1	x_1		ix_2	x_2		
$XL(t_2, r3)$	$ix_{1,2}$		x_1	x_2	ix_1	x_1		ix_2	x_2		
G) $u(t_2, r2, b5, a5)$											
$XL(t_2, r2)$	$ix_{1,2}$		x_1	x_2	ix_1	x_1		ix_2	x_2		

t_1 non ha ancora effettuato il commit, quindi se t_2 ha ottenuto il lock il log *non può derivare* da un sistema 2PL Strict – **non è compatibile**.

Tuttavia a t_1 manca **solo** il commit, quindi potrebbe aver rilasciato $r2$ subito prima di G): il log è quindi compatibile con un sistema 2PL NON Strict

Another idea for 2PL / 2PL strict

- We can use the same approach to check if a schedule is 2PL or 2PL strict.
- Let's write all the actions and the state of the resources in a table.
- First we check if the schedule is 2PL strict.
- If it is, we know that it's also 2PL, otherwise we try to anticipate the unlocks.

Z.1 Determine if the following schedule is compatible with 2PL Strict, 2PL or non-2PL.

$r1(x) \; w1(x) \; r2(z) \; r1(y) \; w1(y)$
 $r2(x) \; w2(x) \; w2(z)$

We SUPPOSE that the commits are performed immediately after the transaction last operation

$r1(x) \; w1(x) \; r2(z) \; r1(y) \; w1(y) \; \textcolor{red}{c1}$
 $r2(x) \; w2(x) \; w2(z) \; \textcolor{red}{c2}$

	X	Y	Z	notes
1	$r1(x)$	$\textcolor{blue}{S_1}$		
2	$w1(x)$	$\textcolor{blue}{X_1}$		
3	$r2(z)$	X_1	$\textcolor{blue}{S_2}$	
4	$r1(y)$	X_1	$\textcolor{blue}{S_1}$	S_2
5	$w1(y)$	X_1	$\textcolor{blue}{X_1}$	S_2
6	$c1$	X_1	X_1	S_2
7	$r2(x)$	$\textcolor{blue}{S_2}$		S_2
8	$w2(x)$	$\textcolor{blue}{X_2}$		S_2
9	$w2(z)$	X_2		$\textcolor{blue}{X_2}$
10	$c2$	X_2		X_2

No incompatibilities, the schedule is 2PL Strict compatible (and also 2PL).

Z.2 Determine if the following schedule is compatible with 2PL Strict, 2PL or non-2PL.

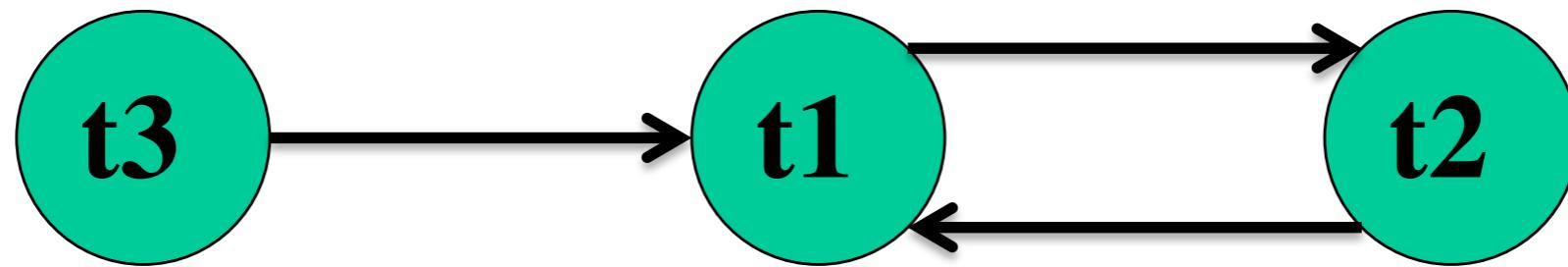
$r1(x) \ w1(x) \ w3(x) \ r2(y) \ r3(y) \ w3(y)$
 $w1(y) \ r2(x)$

We SUPPOSE that the commits are performed immediately after the transaction last operation

$r1(x) \ w1(x) \ w3(x) \ r2(y) \ r3(y) \ w3(y)$
 $c3 \ w1(y) \ c1 \ r2(x) \ c2$

	X	Y	notes
1	$rl(x)$	S_1	
2	$wl(x)$	X_1	
3	$w3(x)$		3 has to wait t1 for X
4	$r2(y)$	X_1	S_2
5	$r3(y)$		waiting
6	$w3(y)$		waiting
7	$c3$		waiting, can't commit
8	$wl(y)$		1 has to wait t2 for Y
9	cl		waiting, can't commit
10	$r2(x)$		2 has to wait t1 for X
11	$c2$		waiting, can't commit

There is a DEADLOCK between t1 and t2.



The schedule is not compatible with 2PL Strict.

We can remove the strictness and check if it's 2PL.
(A transaction can unlock resources before the commit, but cannot acquire other locks after the first unlock).

	X	Y	notes
1	$rl(x)$	S_1	
2	$wl(x)$	X_1	
3	$w3(x)$		
4	$r2(y)$		
5	$r3(y)$		In order to let $t3$ lock X, $t1$ should unlock it before.
6	$w3(y)$		
7	$c3$		
8	$wl(y)$		
9	$c1$		
10	$r2(x)$		Because of the 2PL rule (after the first unlock the transaction cannot lock any other resource), we should try to anticipate all the locks for $t1$ before the first unlock.
11	$c2$		

	X	Y	notes
1	$r1(x)$	S_1	
2	$w1(x)$	X_1	
3	$w3(x)$		
4	$r2(y)$		
5	$r3(y)$		
6	$w3(y)$		
7	$c3$		
8	$w1(y)$		The only other action for $t1$ is $w1(y)$. We should move its lock upwards, to be able to unlock X.
9	$c1$		
10	$r2(x)$		
11	$c2$		

	X	Y	notes
1	$r1(x)$	S_1	
2	$w1(x)$	X_1	
		X_1	
3	$w3(x)$	X_3	
4	$r2(y)$		
5	$r3(y)$		
6	$w3(y)$		
7	$c3$		
8	$w1(y)$		
9	$c1$		
10	$r2(x)$		
11	$c2$		

A green L-shaped bracket on the left side of the table indicates a sequence of actions from row 1 to row 11.

A green arrow points to row 2.

Text annotations:

- Row 4: Now it's possible for t3 to lock X.
- Row 7: But t1 cannot unlock Y until $w1(y)$ (instant 8).
- Row 10: This would conflict with the actions $r2(y)$, $r3(y)$ and $w3(y)$ (instants 4, 5, 6). So this solution is *not feasible*.

	X	Y	notes
1	$r1(x)$	S_1	
2	$w1(x)$	X_1	
		X_1	
3	$w3(x)$	X_3	
4	$r2(y)$		
5	$r3(y)$		
6	$w3(y)$		
7	$c3$		
8	$w1(y)$		
9	$c1$		
10	$r2(x)$		
11	$c2$		

A green L-shaped bracket on the left side groups rows 1 through 4, and a green arrow points from this group to row 2.

From this example we can derive that:

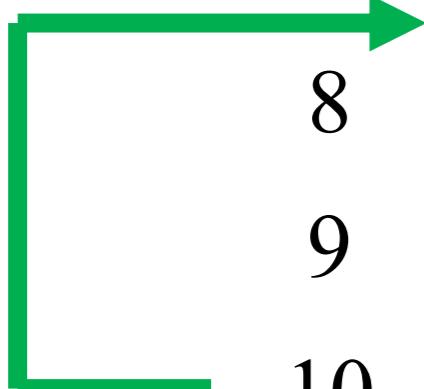
We can anticipate the locks/unlocks of the action A (instant M) to the instant N , only if A is not in conflict with any other action X between N and M .

Otherwise we would generate more conflicts → more waiting states.

	X	Y	notes
1	$r1(x)$	S_1	
2	$w1(x)$	X_1	
3	$w3(x)$		
4	$r2(y)$		
5	$r3(y)$		
6	$w3(y)$		
7	$c3$		
8	$w1(y)$		We could anticipate the lock of Y and unlock of X for $t1$ before 7, but not before 6 (it would generate a conflict).
9	$c1$		
10	$r2(x)$		
11	$c2$		This doesn't solve the waiting state, so $w3(x)$ has to wait.

	X	Y	notes
1	$r1(x)$	S_1	
2	$w1(x)$	X_1	
3	$w3(x)$		3 has to wait t1 for X
4	$r2(y)$	X_1	S_2
5	$r3(y)$		waiting
6	$w3(y)$		waiting
7	$c3$		waiting, can't commit
8	$w1(y)$		
9	$c1$		
10	$r2(x)$		t1 needs an exclusive lock on Y, t2 does not need it anymore, but it should lock X before unlocking Y.
11	$c2$		

	X	Y	notes
1	$r1(x)$	S_1	
2	$w1(x)$	X_1	
3	$w3(x)$		3 has to wait t1 for X
4	$r2(y)$	X_1	S_2
5	$r3(y)$		waiting
6	$w3(y)$		waiting
7	$c3$		waiting, can't commit
8	$w1(y)$	Unfortunately, the resource X is locked by t1, so it cannot be locked by t2.	
9	$c1$		
10	$r2(x)$	And t1 cannot release it otherwise it loses the ability to lock Y.	
11	$c2$		



	X	Y	notes
1	$r1(x)$	S_1	
2	$w1(x)$	X_1	
3	$w3(x)$		3 has to wait t1 for X
4	$r2(y)$	X_1	S_2
5	$r3(y)$		waiting
6	$w3(y)$		waiting
7	$c3$		waiting, can't commit
8	$w1(y)$		1 has to wait t2 for Y
9	$c1$		waiting, can't commit
10	$r2(x)$		2 has to wait t1 for X
11	$c2$		waiting, can't commit