



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Gestione delle Transazioni

Basi di Dati

Corso di Laurea in Informatica per il Management

Alma Mater Studiorum - Università di Bologna

Prof. Marco Di Felice

Dipartimento di Informatica – Scienza e Ingegneria

marco.difelice3@unibo.it

Gestione delle Transazioni

Esempio. Gestione ordini su un sito di **ecommerce**.

(struttura del DB semplificata)

ITEM(Codice, Descrizione, Prezzo, Quantita)

ORDINE(Id, Data, Ordinate, ItemOrdinato)

```
SET NumItem=(SELECT COUNT(*) FROM ITEM WHERE
(Codice=CodiceScelto));
IF (NumItem > 0) THEN
    UPDATE ITEM SET Quantita=Quantita-1 WHERE
    (Codice=CodiceScelto);
    INSERT INTO ORDINE(Data,Ordinate,ItemOrdinato) VALUES
    (NOW(), NomeOrdinate, CodiceScelto);
END IF;
```

Gestione delle Transazioni

Esempio. Gestione ordini su un sito di **ecommerce**.

(struttura del DB semplificata)

ITEM(Codice, Descrizione, Prezzo, Quantita)

ORDINE(Id, Data, Ordinate, ItemOrdinato)

```
SET NumItem=(SELECT COUNT(*) FROM ITEM WHERE  
(Codice=CodiceScelto));
```

```
IF (NumItem > 0) THEN
```

```
    UPDATE ITEM SET Quantita=Quantita-1 WHERE (Codice=CodiceScelto));
```

```
    INSERT INTO ORDINE(Data,Ordinate,ItemOrdinato) VALUES      (NOW(),  
NomeOrdinate, CodiceScelto);
```

```
END IF;
```



Il sistema va in crash in questo punto!

Gestione delle Transazioni

Esempio. Gestione ordini su un sito di **ecommerce**.

(struttura del DB semplificata)

ITEM(Codice, Descrizione, Prezzo, Quantita)

ORDINE(Id, Data, Ordinate, ItemOrdinato)

```
SET NumItem=(SELECT COUNT(*) FROM ITEM WHERE (Codice=CodiceScelto));
```

```
IF (NumItem > 0) THEN
```

```
    UPDATE ITEM SET Quantita=Quantita-1 WHERE (Codice
```


Due ordini in contemporanea
eseguono la query

```
        INSERT INTO ORDINE(Data,Ordinate,ItemOrdinato) VALUES      (NOW(),  
NomeOrdinate, CodiceScelto);  
END IF;
```

Gestione delle Transazioni

- Le **transazioni** rappresentano **unità di lavoro** elementare (insiemi di istruzioni SQL) che **modificano** il contenuto di una base di dati.

```
start transaction
update SalariImpiegati
set conto=conto*1.2
where (CodiceImpiegato = 123)
commit work
```




Le transazioni
sono comprese
tra una start
transaction
ed una
commit/
rollback

Gestione delle Transazioni

- Le **transazioni** rappresentano **unità di lavoro** elementare (insiemi di istruzioni SQL) che **modificano** il contenuto di una base di dati.

```
start transaction
update SalariImpiegati
set conto=conto-10
where (CodiceImpiegato = 123)
if var > 0 then commit work;
else rollback work;
```



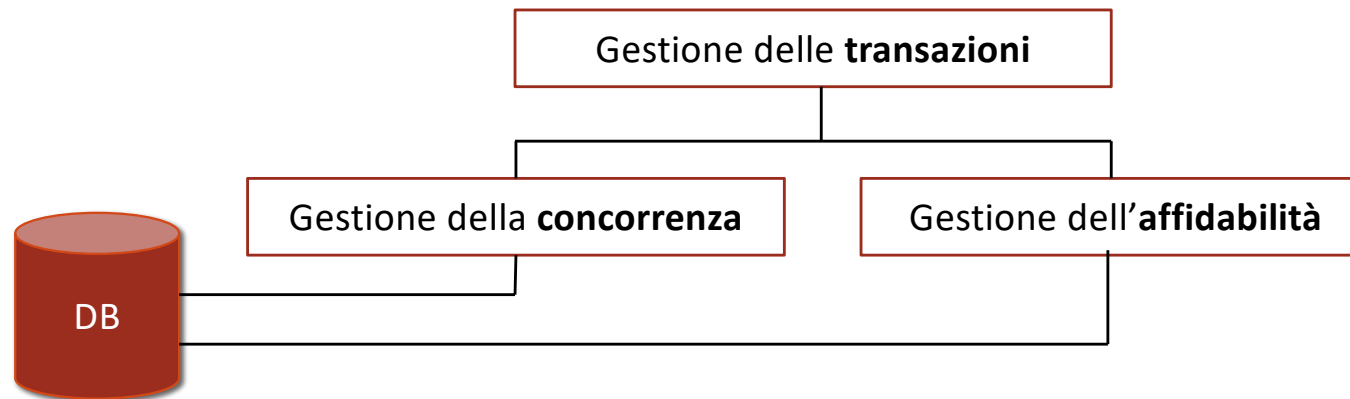
Le transazioni
sono comprese
tra una start
transaction
ed una
commit/
rollback

Gestione delle Transazioni

PROPRIETA' ACIDE DELLE TRANSAZIONI

- **Atomicità** → La transazione deve essere eseguita con la regola del “**tutto o niente**”.
- **Consistenza** → La transazione deve lasciare il DB in uno stato **consistente**, eventuali vincoli di integrità non devono essere violati.
- **Isolamento** → L'esecuzione di una transazione deve essere **indipendente** dalle altre.
- **Persistenza** → L'effetto di una transazione che ha fatto commit work non deve essere perso.

Gestione delle Transazioni



- **Gestore dell'affidabilità** → garantisce **atomicità** e **persistenza**
... COME? Usando *log* e *checkpoint*.
- **Gestore della concorrenza** → garantisce l'isolamento in caso di esecuzione *concorrente* di piu' transazioni.

Gestione delle Transazioni

Date un **insieme di transazioni** T_1, T_2, \dots, T_n , di cui ciascuna formata da un certo insieme di operazioni di scrittura (w_i) e lettura (r_i):

Es. $T_1 = r_1(x) \ r_1(y) \ r_1(z) \ w_1(y) \ \dots$

Si definisce **schedule** la sequenza di operazioni di lettura/scrittura di tutte le transazioni **così come eseguite sulla base di dati**:

$r_1(x) \ r_2(y) \ r_1(y) \ w_4(y) \ w_2(z) \ \dots$

Gestione delle Transazioni

Uno schedule **S** si dice **seriale** se **le azioni di ciascuna transazione appaiono in sequenza**, senza essere inframezzate da azioni di altre transazioni.

$$S = \{T_1, T_2, \dots T_n\}$$

Schedule seriale ottenibile se:

- (i) Le transazioni sono **eseguite uno alla volta** (scenario non realistico)
- (ii) Le transazioni sono **completamente indipendenti** l'una dall'altra (improbabile)

Gestione delle Transazioni

In un sistema reale, le **transazioni vengono eseguite in concorrenza** per ragioni di efficienza / scalabilità.

... Tuttavia, l'esecuzione concorrente determina un insieme di **problematiche** che devono essere gestite ...

T1= Read(x); x=x+1; Write(x); Commit Work

T2= Read(x); x=x+1; Write(x); Commit Work

Se x=3, al termine delle due transazioni x vale 5 (*esecuzione sequenziale*) ... cosa accade in caso di **esecuzione concorrente**?

Gestione delle Transazioni

Problema 1: **Perdita di Aggiornamento**

Transazione1 (T1)	Transazione2 (T2)
Read(x)	
x=x+1	
	Read(x)
	x=x+1
	Write(x)
	Commit work
Write(x) Commit work	

T1
scrive 4

T2
scrive 4

Gestione delle Transazioni

Problema 2: **Lettura sporca**

Transazione1 (T1)	Transazione2 (T2)
Read(x)	
x=x+1	
Write(x)	
	Read(x)
	Commit work
Rollback work	

T2 legge
4!

Gestione delle Transazioni

Problema : **Lettture inconsistenti**

T1 legge 3!	Transazione1 (T1)	Transazione2 (T2)
	Read(x)	
		Read(x)
		x=x+1
		Write(x)
T1 legge 4!		Commit work
	Read(x) Commit work	

Gestione delle Transazioni

Problema 4: **Aggiornamento Fantasma**

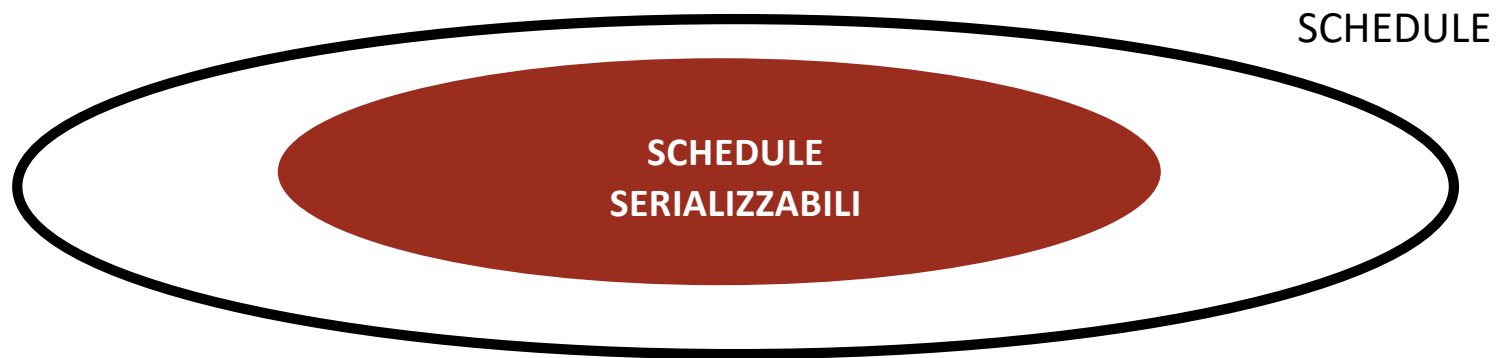
Vincolo:
 $x+y+z$
deve
essere =
a 1000

**Vincolo
violato!!**

Transazione1 (T1)	Transazione2 (T2)
Read(x)	
	Read(y)
Read(y)	
	$y=y-100$
	Read(z)
	$z=z+100$
	Write(y), Write(z)
	Commit work
Read(z) $s=x+y+z$; Commit work	

Gestione delle Transazioni

- Uno schedule **S** si dice **serializzabile** se produce lo stesso risultato di un qualunque scheduler seriale **S'** delle stesse transazioni.



Gestione delle Transazioni

D. Come implementare il **controllo della concorrenza**?

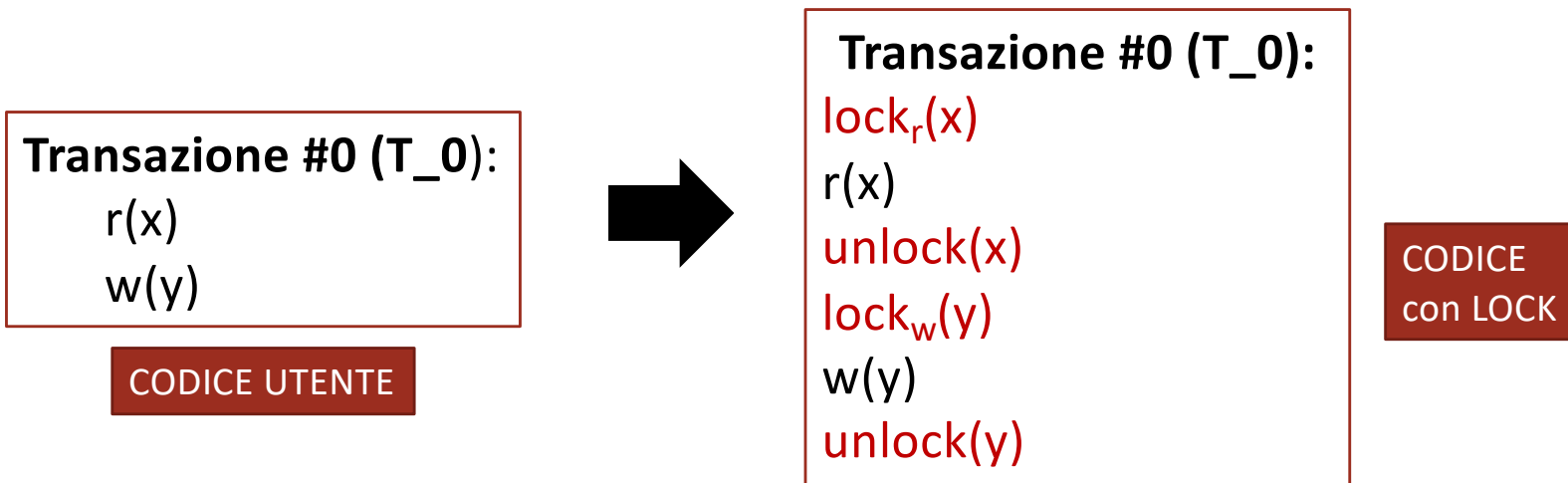
I DMBS commerciali usano il meccanismo dei **lock** → per poter effettuare una qualsiasi operazioni di lettura/scrittura su una risorsa (tabella o valore di una cella), **è necessario aver precedentemente acquisito il controllo (**lock**) sulla risorsa stessa.**

- **Lock in lettura** (*accesso condiviso*)
- **Lock in scrittura** (*mutua esclusione*)

Gestione delle Transazioni

Su ogni lock possono essere definite **due operazioni**:

- **Richiesta** del lock in lettura/scrittura.
- **Rilascio** del lock (**unlock**) acquisito in precedenza.



Gestione delle Transazioni

Lock Manager → componente del DBMS responsabile di **gestire i lock alle risorse del DB, e di rispondere alle richieste delle transazioni.**

STRUTTURE DATI del LOCK MANAGER

Per ciascun oggetto x del DBMS:

- **State(x)** → **stato** dell'oggetto (libero/r_locked/w_locked)
- **Active(x)** → **lista transazioni attive** sull'oggetto
- **Queued(x)** → **lista transazioni bloccate** sull'oggetto

Gestione delle Transazioni

Lock Manager → componente del DBMS responsabile di **gestire i lock alle risorse del DB, e di rispondere alle richieste delle transazioni.**

AZIONI DEL LOCK MANAGER

1. Riceve una richiesta (r_lock, w_lock, unlock) da una transazione T, su un oggetto x (oggetto=tabella, colonna, etc).
2. Controlla la **tabella stato/azione** (slide successiva).
3. Se la risposta è **OK**, **aggiorna lo stato della risorsa**, e concede il controllo della risorsa alla transazione T.
4. Se la risposta è **NO**, **inserisce la transazione T in una coda** associata ad x.

Gestione delle Transazioni

Su ogni lock possono essere definite **due operazioni**:

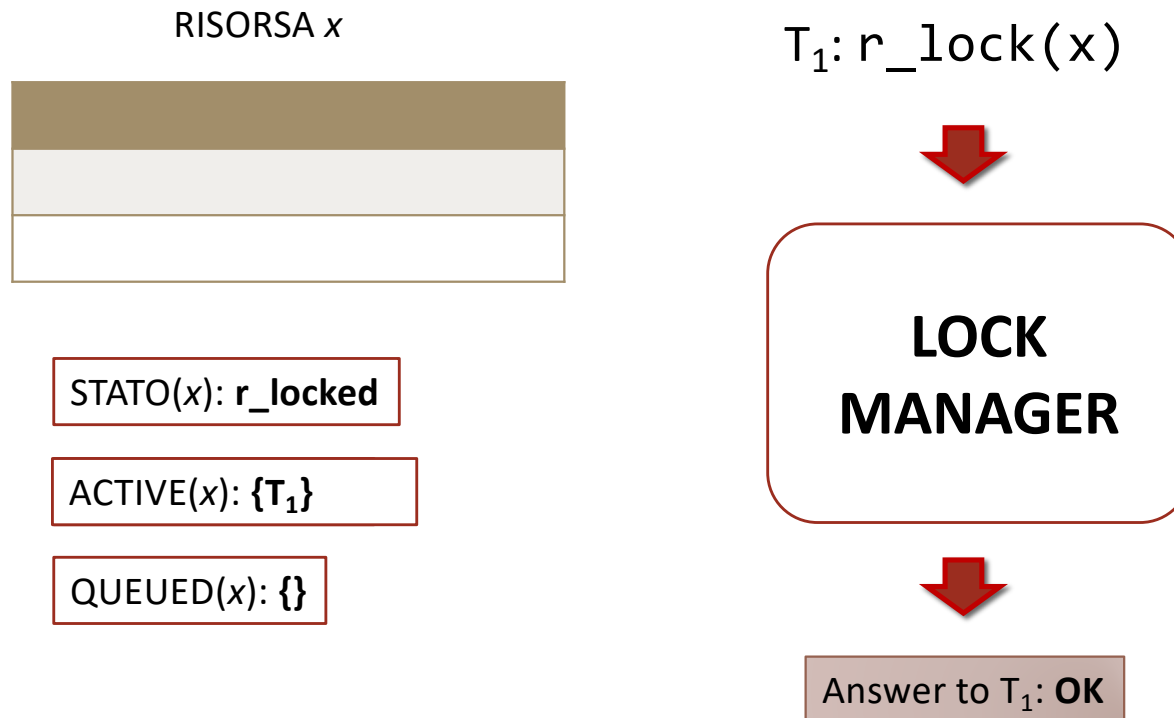
- **Richiesta** del lock in lettura/scrittura.
- **Rilascio** del lock (**unlock**) acquisito in precedenza.

STATO DELLA RISORSA

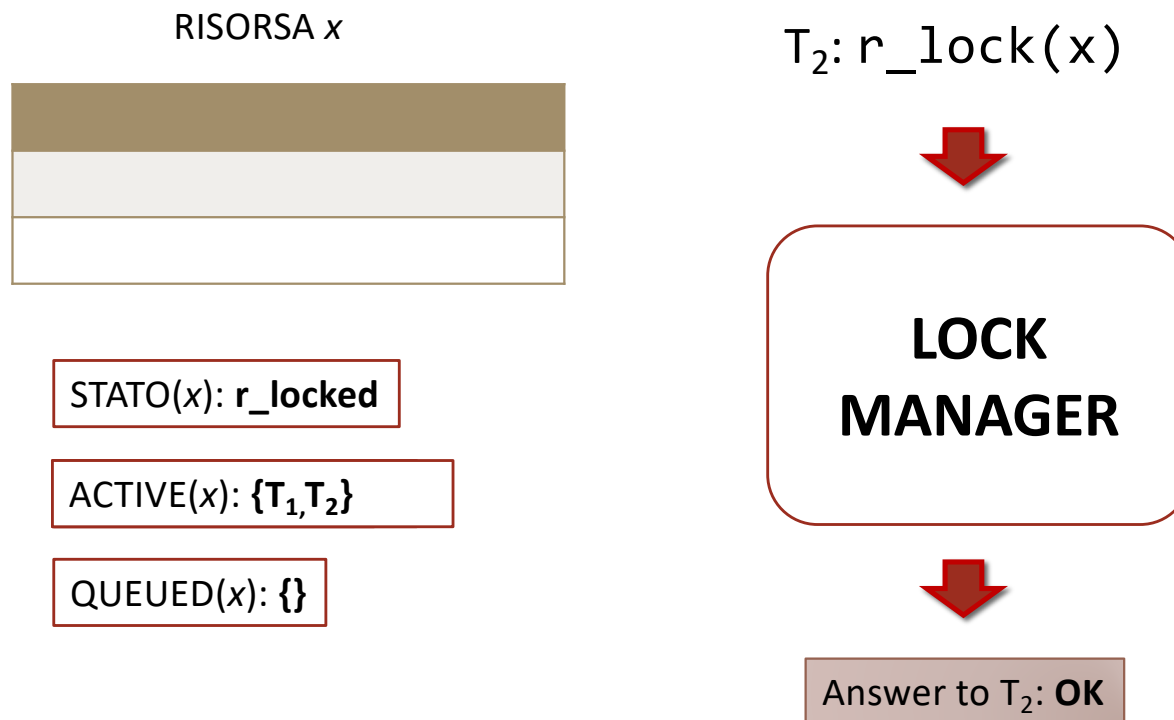
AZIONE

	Libero	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	Errore	OK /dipende	OK /libero

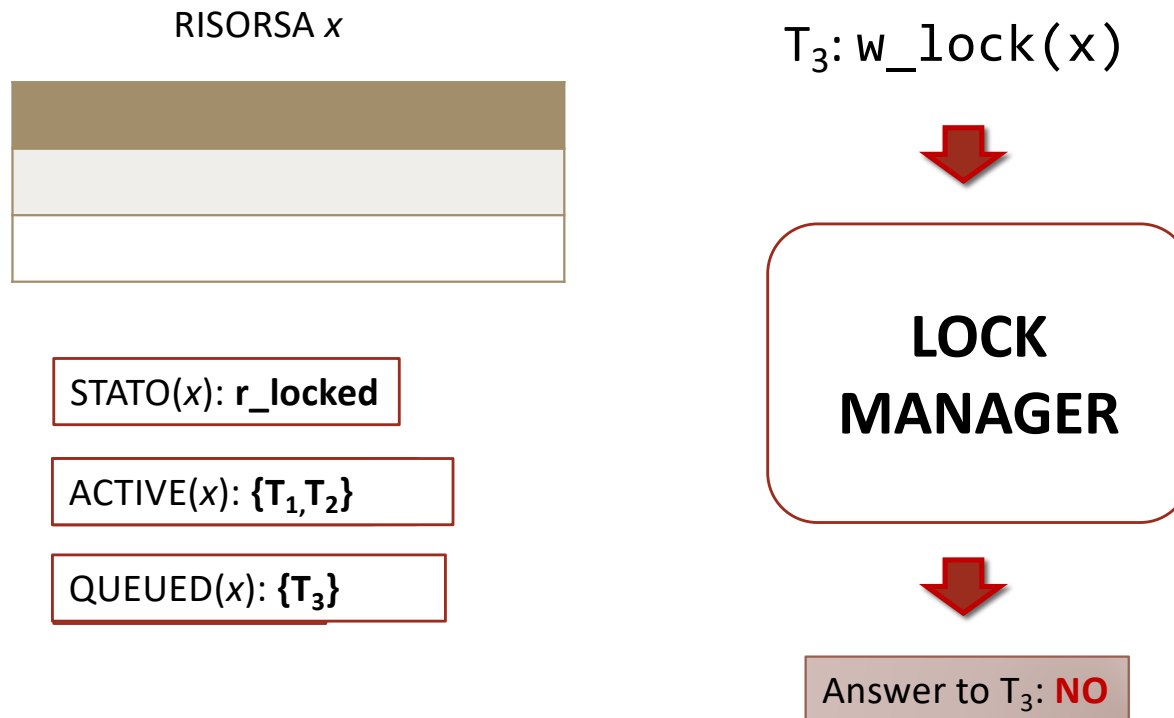
Gestione delle Transazioni



Gestione delle Transazioni



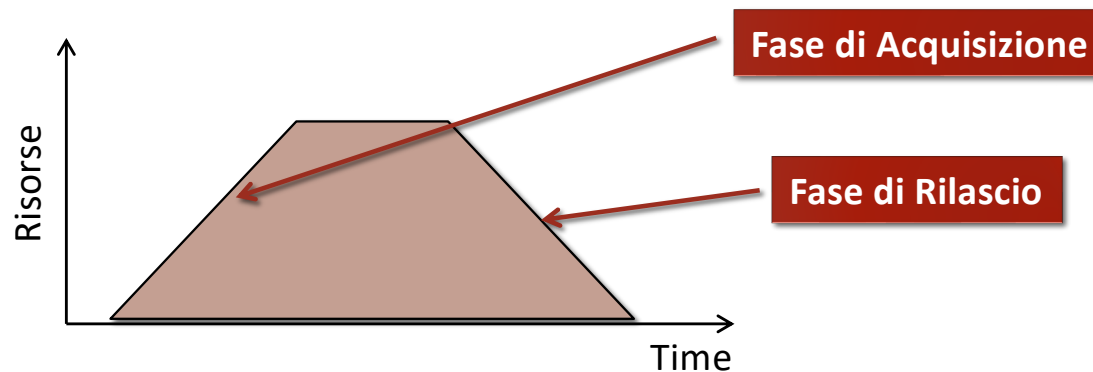
Gestione delle Transazioni



Gestione delle Transazioni

Two Phase Lock (2PL) → Una transazione, dopo aver rilasciato un lock, non può acquisirne un altro.

- In pratica, una transazione **acquisisce prima tutti i lock delle risorse di cui necessita ...**



Gestione delle Transazioni

TRANSAZIONI

$T_1 = r(x), w(y), \text{Commit}$
 $T_2 = w(x), \text{Commit}$

SCHEDULE



D. E' uno schedule **2PL**?

R. NO!

T ₁	T ₂
r_lock(x)	
r(x)	
unlock(x)	
	w_lock(x)
	W(x)
	Unlock(x)
	Commit
w_lock(y)	
w(y)	
unlock(y)	
Commit	

Gestione delle Transazioni

TRANSAZIONI

$T_1 = r(x), w(y), \text{Commit}$
 $T_2 = r(y), \text{Commit}$

SCHEDULE



D. E' uno schedule **2PL**?

R. SI!

T ₁	T ₂
r_lock(x)	
r(x)	
w_lock(y)	
	r_lock(y)
w(y)	
unlock(y)	
unlock(x)	
	r(y)
	unlock(y)
Commit	
	Commit

Gestione delle Transazioni

Two Phase Lock (2PL) → Una transazione, dopo aver rilasciato un lock, non può acquisirne un altro.

- Ogni schedule che rispetta il 2PL e' anche **serializzabile** (perchè ?).
- Ogni schedule che rispetta il 2PL non può incorrere in **configurazioni erronee** dovute a: aggiornamento fantasma, lettura inconsistente, perdita di aggiornamento ... **che accade in caso di lettura sporca?**

Gestione delle Transazioni

TRANSAZIONI

$T_1 = r(x), w(y), \text{Abort}$
 $T_2 = r(y), \text{Commit}$

SCHEDULE

Lettura sporca!

T_1	T_2
$r_lock(x)$	
$r(x)$	
$w_lock(y)$	
	$r_lock(y)$
$w(y)$	
$unlock(y)$	
$unlock(x)$	
	$r(y)$
	$unlock(y)$
Abort	
	Commit

Gestione delle Transazioni

Strict Two Phase Lock (2PL) → I lock di una transazione sono rilasciati solo dopo aver effettuato le operazioni di commit/abort.

- Variante strict del 2PL, utilizzato in alcuni DBMS commerciali.
- Uno schedule che rispetta lo S2PL **eredita tutte le proprietà del 2PL**, ed inoltre **NON** presenta anomalie causate da problemi di **lettura sporca**.

Gestione delle Transazioni

TRANSAZIONI

$T_1 = r(x), w(y), \text{Abort}$
 $T_2 = r(y), \text{Commit}$

SCHEDULE



D. E' uno schedule **S2PL**?

R. SI!

T_1	T_2
$r_lock(x)$	
$r(x)$	
$w_lock(y)$	
	$r_lock(y)$
$w(y)$	
Abort	
unlock(x)	
unlock(y)	
	$r(y)$
	Commit
	unlock(y)

Gestione delle Transazioni

PROBLEMA: I protocolli 2PL e S2PL possono generare schedule con situazioni di **deadlock**.

TRANSAZIONI

$T_1 = r(x), w(y), \text{Commit}$
 $T_2 = r(y), w(x), \text{Commit}$

SCHEDULE



T_1	T_2
$r_lock(x)$	
	$r_lock(y)$
$r(x)$	
	$r(y)$
$w_lock(y)$	
	$w_lock(x)$

Gestione delle Transazioni

Per gestire le situazioni di **deadlock** causate dal Lock Manager, si possono usare **tre tecniche**:

1. **Uso dei timeout** → ogni operazione di una transazione ha un **timeout** entro il quale deve essere completata, pena **annullamento (abort)** della transazione stessa.

T_1 : `r_lock(x, 4000)`, `r(x)`, `w_lock(y, 2000)`, `w(y)`,
`commit`, `unlock(x)`, `unlock(y)`

Gestione delle Transazioni

Per gestire le situazioni di **deadlock** causate dal Lock Manager, si possono usare **tre tecniche**:

2. Deadlock avoidance → prevenire le configurazioni che potrebbero portare ad un deadlock ... COME?

- Lock/Unlock di tutte le risorse allo stesso tempo.
- Utilizzo di **time-stamp** o di **classi di priorità** tra transazioni (problema: puo' determinare **starvation!**)

Gestione delle Transazioni

Per gestire le situazioni di **deadlock** causate dal Lock Manager, si possono usare **tre tecniche**:

3. Deadlock detection → utilizzare **algoritmi per identificare eventuali situazioni di deadlock**, e prevedere meccanismi di recovery dal deadlock

- **Grafo delle richieste/risorse** utilizzato per identificare la presenza di cicli (corrispondenti a deadlock)
- In caso di ciclo, si fa abort delle transazioni coinvolte nel ciclo in modo da eliminare la mutua dipendenza ...

Gestione delle Transazioni

Un metodo alternativo al 2PL per la gestione della concorrenza in un DBMS prevede l'utilizzo dei **time-stamp delle transazioni** (metodo **TS**).

- Ad ogni transazione si associa un **timestamp** che rappresenta il momento di inizio della transazione.
- Ogni transazione **non può leggere o scrivere un dato scritto da una transazione con timestamp maggiore.**
- Ogni transazione **non può scrivere su un dato già letto da una transazione con timestamp maggiore.**

Gestione delle Transazioni

Un metodo alternativo al 2PL per la gestione della concorrenza in un DBMS prevede l'utilizzo dei **time-stamp delle transazioni** (metodo **TS**).

- Ad ogni oggetto x si associano **due indicatori**:
 - ✧ $WTM(x) \rightarrow$ timestamp della transazione che ha fatto l'ultima scrittura su x .
 - ✧ $RTM(x) \rightarrow$ timestamp dell'ultima transazione (ultima=con t piu' alto) che ha letto x .

Gestione delle Transazioni

Lo **scheduler di sistema** verifica se un'eventuale azione ($r_t(x)$ o $w_t(x)$) eseguita da una transazione T con timestamp t può essere eseguita o meno:

- $r_t(x)$ → Se $t < WTM(x)$ allora la transazione viene uccisa. Se $t \geq WTM(x)$, la richiesta viene eseguita, ed $RTM(x)$ viene aggiornato al massimo tra il valore precedente di $RTM(x)$ e t stesso.

Gestione delle Transazioni

Lo **scheduler di sistema** verifica se un'eventuale azione ($r_t(x)$ o $w_t(x)$) eseguita da una transazione T con timestamp t può essere eseguita o meno:

- $w_t(x)$ → Se $t < WTM(x)$ oppure $t < RTM(x)$ allora la transazione viene uccisa. Altrimenti, la richiesta viene accettata, e $WTM(x)$ viene posto uguale a t .

Gestione delle Transazioni

ESEMPIO: $RTM(x)=6$, $WTM(x)=3$

$T_5: r_5(x) \rightarrow \text{OK}, RTM(x)=6$

$T_9: w_9(x) \rightarrow \text{OK}, WTM(x)=9$

$T_6: w_6(x) \rightarrow \text{NO}, T_6$ uccisa

$T_8: r_8(x) \rightarrow \text{NO}, T_8$ uccisa

$T_{10}: r_{10}(x) \rightarrow \text{OK}, RTM(x)=10$

Gestione delle Transazioni

In SQL-3, ed in molti DBMS commerciali (DB2, MySQL, PostgreSQL, Oracle, etc) sono definiti quattro **livelli di isolamento** tra transazioni:

Livello	Descrizione
read uncommitted	(read only) La transazione non emette lock in lettura, e non rispetta lock esclusivi da altre transazioni.
read committed	Richiede lock condivisi per effettuare le letture.
repeteable read	Applica il protocollo S2PL anche in lettura.
serializable	Applica il protocollo S2PL con lock di predicato.

- **S2PL** utilizzato per le operazioni di scrittura, da tutti i livelli.

Gestione delle Transazioni

MySQL offre quattro livelli di **isolamento**:

- **READ UNCOMMITTED** → sono visibili gli aggiornamenti non consolidati fatti da altri.
- **READ COMMITTED** → aggiornamenti visibili solo se consolidati (ossia solo dopo COMMIT).
- **REPEATABLE READ** → tutte le letture di un dato operate da una transazione leggono sempre lo stesso valore (comportamento di **default**).
- **SERIALIZABLE** → lettura di un dato blocca gli aggiornamenti fino al termine della transazione stessa che ha letto il dato (lock applicato ad ogni SELECT).

Gestione delle Transazioni

SINTASSI MySQL

- Iniziare una transazione e completarla:

```
SET AUTOCOMMIT =0;  
START TRANSACTION  
... (Statements SQL)  
COMMIT/ROLLBACK
```

- Configurare livello di isolamento di esecuzione:

```
SET TRANSACTION ISOLATION LEVEL  
    REPEATABLE READ | READ COMMITTED |  
    READ UNCOMMITTED | SERIALIZABLE
```

Gestione delle Transazioni

SINTASSI MySQL

- Le transazioni sono **utilizzabili solo** su tabelle di tipo **InnoDB** (ACID-compliant).
- E' possibile **gestire manualmente le operazioni di lock** su tabelle (non consigliabile su tabelle di tipo InnoDB):

LOCK TABLES

```
tabella { READ | WRITE }
```