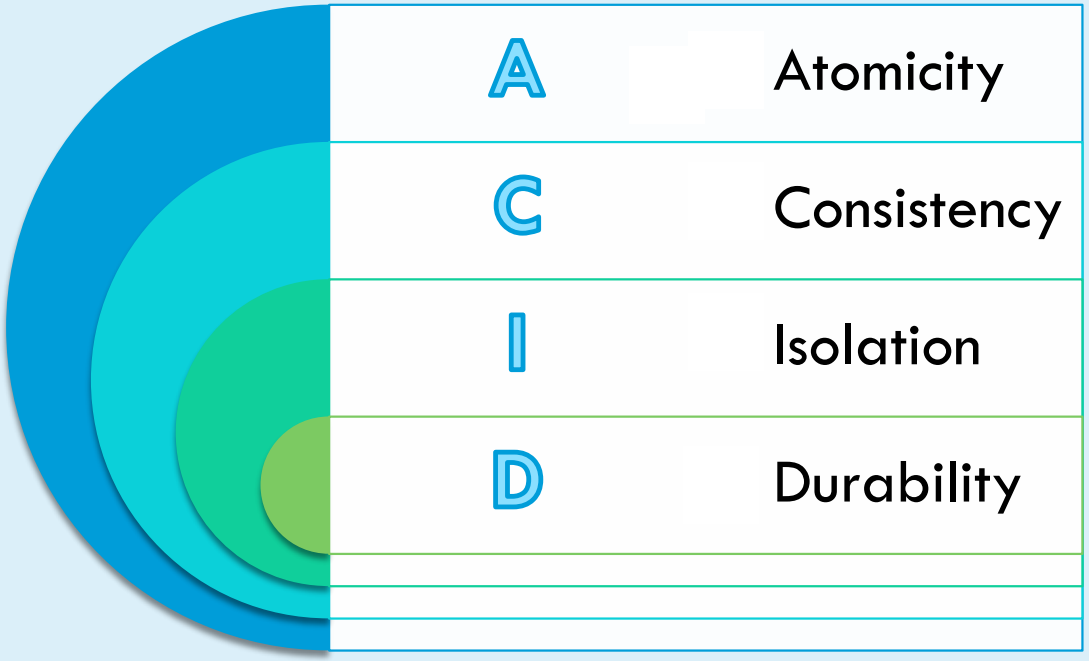


Gestione delle transazioni

Annalisa Franco

Università di Bologna

Concetto di transazione

- Una transazione è un'unità logica di elaborazione che corrisponde a un insieme di operazioni fisiche elementari (letture/scritture) sul DB
- Esempi:
  - Trasferimento di una somma da un conto corrente a un altro

```
UPDATE CC
SET    Saldo = Saldo - 50
WHERE  Conto = 'CA4112'
```

```
UPDATE CC
SET    Saldo = Saldo + 50
WHERE  Conto = 'CA5106'
```

  - Aggiornamento degli stipendi degli impiegati di una sede

```
UPDATE Imp
SET    Stipendio = 1.1*Stipendio
WHERE  Sede = 'S01'
```

- In entrambi i casi tutte le operazioni elementari devono essere eseguite

Proprietà ACID

- Un DBMS deve garantire che ogni transazione sia **ACID** ovvero goda delle seguenti proprietà:
  - **Atomicity** = una transazione è un'unità di elaborazione
    - o tutti gli effetti di una transazione sono registrati nel DB (committed) o nessuno (aborted);
  - **Consistency** = una transazione lascia il DB in uno stato consistente, cioè opera una trasformazione corretta dello stato del DB
    - Il DBMS garantisce che nessuno dei vincoli di integrità del DB venga violato
  - **Isolation** = una transazione esegue indipendentemente dalle altre
    - Se più transazioni eseguono in concorrenza, il DBMS garantisce che l'effetto netto è equivalente a quello di un'esecuzione seriale delle stesse
  - **Durability** = gli effetti di una transazione che ha terminato correttamente la sua esecuzione devono essere persistenti nel tempo
    - Il DBMS deve proteggere il DB a fronte di guasti

Proprietà ACID e moduli di un DBMS

Query manager

- analizza, autorizza, ottimizza ed esegue le richieste

Storage manager

- gestisce i dispositivi di storage

Transaction manager

- coordina l'esecuzione delle transazioni

DDL compiler

- genera parte dei controlli per garantire Consistency

Logging & Recovery Manager

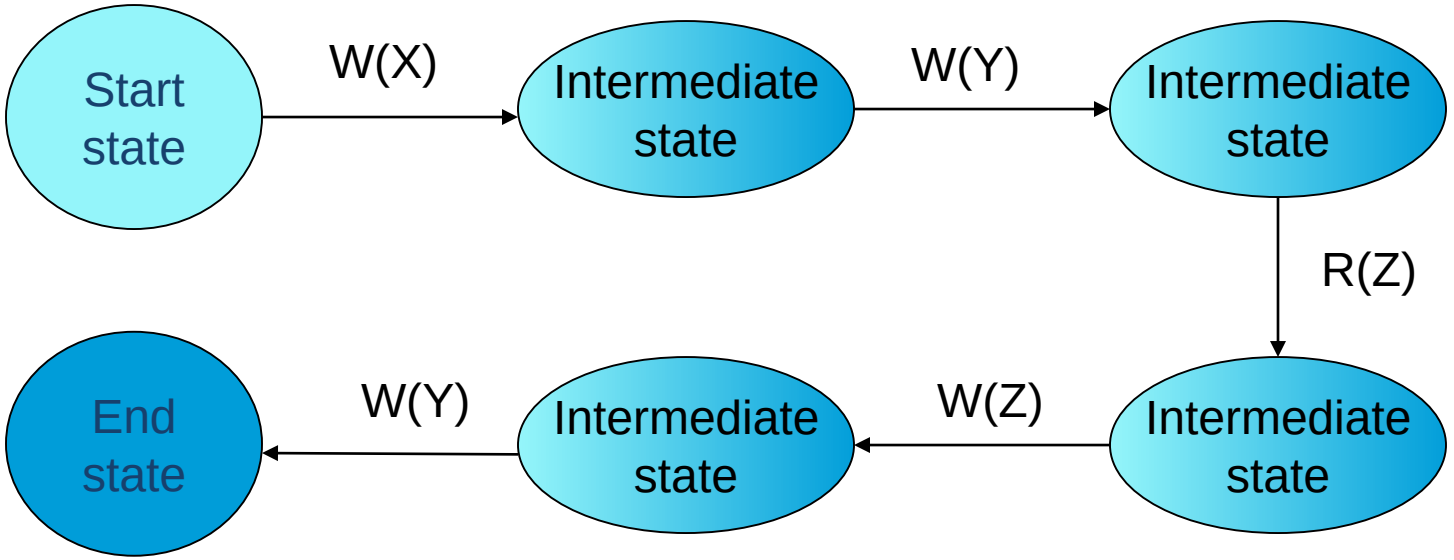
- garantisce Atomicity e Durability, protegge dai malfunzionamenti

Concurrency Manager

- gestisce accessi concorrenti e garantisce Isolation

Modello delle transazioni

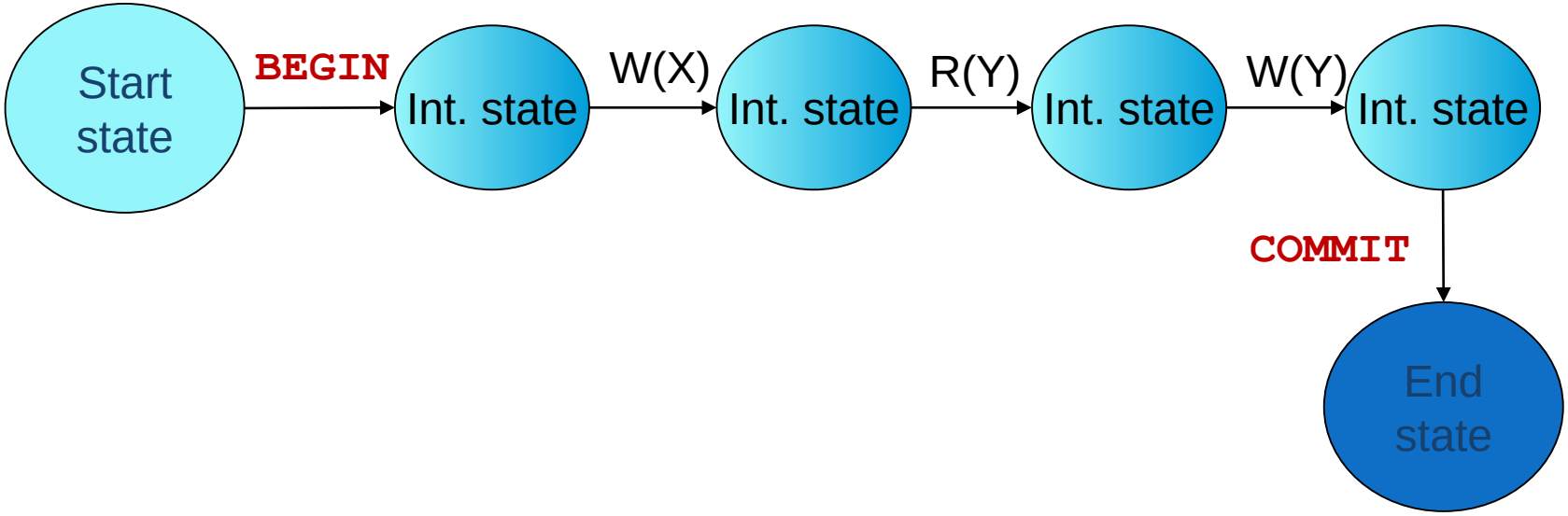
- Nel modello che consideriamo una transazione viene vista come una sequenza di operazioni elementari di lettura (R) e scrittura (W) di oggetti (tuple) del DB che, a partire da uno stato iniziale consistente del DB, porta il DB in un nuovo stato finale consistente



- In generale non è richiesto che siano consistenti gli stati intermedi in cui si trova il DB

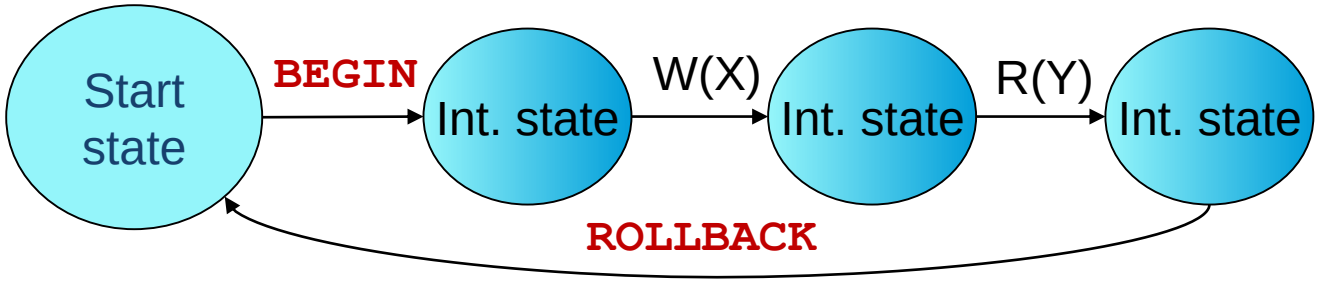
Esiti di una transazione (1)

- Nel modello considerato una transazione (il cui inizio viene indicato dalla parola chiave BEGIN, anche se in SQL è implicito) può avere solo 2 esiti:
  - **Terminare correttamente:** ciò avviene solo quando l'applicazione, dopo aver effettuato tutte le proprie operazioni, esegue una particolare istruzione SQL, detta COMMIT (o COMMIT WORK), che comunica "ufficialmente" al Transaction Manager il termine delle operazioni



Esiti di una transazione (2)

- **Terminare non correttamente** (anticipatamente); sono possibili 2 casi:
  - È la transazione che, per qualche motivo, decide che non ha senso continuare e quindi "abortisce" eseguendo l'istruzione SQL ROLLBACK (o ROLLBACK WORK)
  - È il sistema che non è in grado (ad es. per un guasto o per la violazione di un vincolo) di garantire la corretta prosecuzione della transazione, che viene quindi fatta abortire



- Se per qualche motivo la transazione non può terminare correttamente la sua esecuzione, il DBMS deve "disfare" (UNDO) le eventuali modifiche da essa apportate al DB.

Transazioni in MySQL

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

starts a consistent read

set the transaction access mode

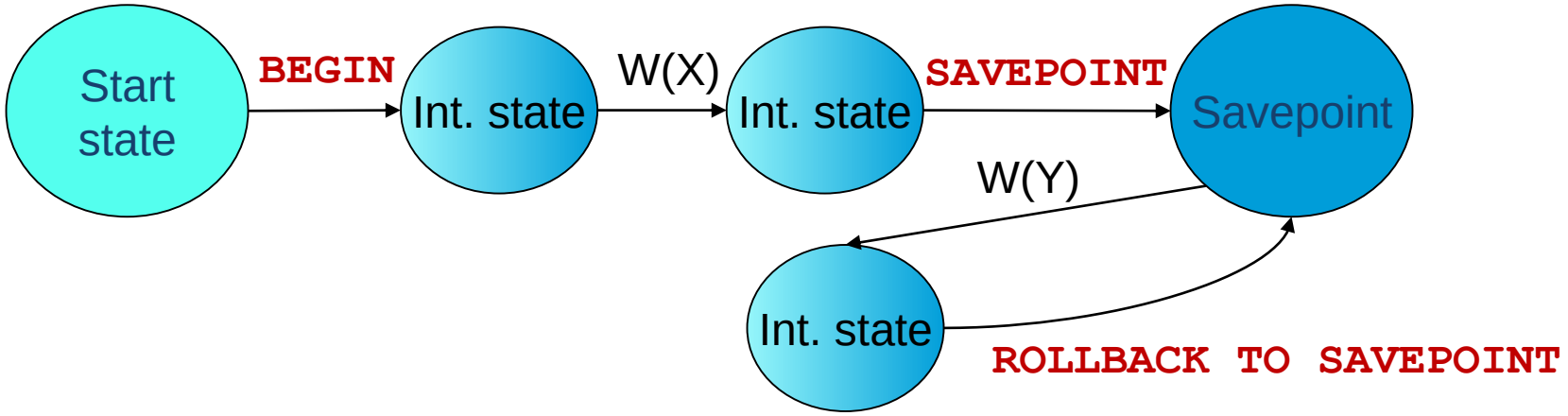
Esempio:

```
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary=@A WHERE type=1;
COMMIT;
```

https://dev.mysql.com/doc/refman/8.0/en/commit.html

Transazioni con Savepoint

- Il modello di transazioni usato dai DBMS è in realtà più articolato; in particolare è possibile definire dei cosiddetti "savepoint", che vengono utilizzati da una transazione per disfare solo parzialmente il lavoro svolto



In MySQL:

sets a named transaction savepoint with a name of identifier

rolls back a transaction to the named savepoint without terminating the transaction.

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

removes the named savepoint

Esecuzione seriale di transazioni

- Un DBMS, dovendo fornire supporto per l'esecuzione di diverse transazioni che accedono a dati condivisi, potrebbe eseguire le transazioni in sequenza ("**serial execution**")
- Ad esempio, due transazioni T1 e T2 potrebbero essere eseguite in questo modo, in cui si evidenzia la successione temporale ("**schedule**") delle operazioni elementari sul DB:

T1	T2
R(X)	
W(X)	
Commit	
	R(Y)
	W(Y)
	Commit

Esecuzione concorrente

- In alternativa, il DBMS può eseguire più transazioni in concorrenza, alternando l'esecuzione di operazioni di una transazione con quella di operazioni di altre transazioni ("**interleaved execution**")
- Eseguire più transazioni concorrentemente è necessario per garantire buone prestazioni:
  - Si sfrutta il fatto che, mentre una transazione è in attesa del completamento di una operazione di I/O, un'altra può utilizzare la CPU, il che porta ad aumentare lo "throughput" (num. transazioni elaborate nell'unità di tempo) del sistema
  - Con una transazione "breve" e una "lunga", l'esecuzione concorrente porta a ridurre il tempo medio di risposta del sistema

T1	T2
R(X)	
	R(Y)
	W(Y)
	Commit
W(X)	
Commit	

Riduzione del tempo di risposta

- T1 è “lunga”, T2 è “breve”; per semplicità ogni riga della tabella è un’unità di tempo

time	T1	T2
1	R(X1)	
2	W(X1)	
…		
999	R(X500)	
1000	W(X500)	
1001	Commit	
1002		R(Y)
1003		W(Y)
1004		Commit

T2 richiede di iniziare a time = 2

time	T1	T2
1	R(X1)	
2		R(Y)
3		W(Y)
4		Commit
5	W(X1)	
…		
1002	R(X500)	
1003	W(X500)	
1004	Commit	

Tempo medio di risposta = (1001 + (1004-1))/2 = **1002**

Tempo medio di risposta = (1004 + 3)/2 = **503.5**

Isolation: gestire la concorrenza

- Il Transaction Manager deve garantire che transazioni che eseguono in concorrenza non interferiscano tra loro. Se ciò non avviene, si possono avere 4 tipi base di problemi, esemplificati dai seguenti scenari:
  - **Lost Update:** due persone, in due agenzie diverse, comprano entrambe l'ultimo biglietto per il concerto degli U2 a Roma (!?)
  - **Dirty Read:** nel programma dei concerti degli U2 figura una tappa a Bologna il 15/07/02, ma quando provate a comprare un biglietto per quella data vi viene detto che in realtà non è ancora stata fissata (!?)
  - **Unrepeatable Read:** per il concerto degli U2 (finalmente la data è stata fissata!) vedete che il prezzo è di 40€, riflettete per 5 minuti, ma il prezzo nel frattempo è salito a 50€ (!?)
  - **Phantom Row:** volete comprare i biglietti di tutte e due le tappe degli U2 in Italia, ma quando comprate i biglietti scoprite che le tappe sono diventate 3 (!?)

Lost Update

- Il seguente schedule mostra un caso tipico di lost update, in cui per comodità si evidenziano anche le operazioni che modificano il valore del dato X e si mostra come varia il valore di X nel DB

Questo aggiornamento viene perso!

T1	X	T2
R(X)	1	
X=X-1	1	
	1	R(X)
	1	X=X-1
W(X)	0	
Commit	0	
	0	W(X)
	0	Commit

- Il problema nasce perché T2 legge il valore di X prima che T1 (che lo ha già letto) lo modifichi ("entrambe vendono l'ultimo biglietto ")

Dirty Read

- In questo caso il problema nasce dal fatto che una transazione legge un dato "che non esiste":

T1	X	T2
R(X)	0	
X=X+1	0	
W(X)	1	
	1	R(X)
Rollback	0	
	0	…
	0	Commit

Questa lettura è "sporca"!

- Quanto svolto da T2 si basa su un valore di X "intermedio", e quindi non stabile ("la data definitiva non è il 15/07/02").
- Le conseguenze sono impredicibili e dipendono dalle azioni svolte da T2.

Unrepeatable Read

- Ora il problema sorge in quanto una transazione legge due volte un dato e trova valori diversi ("il prezzo nel frattempo è aumentato"):

T1	X	T2
R(X)	0	
	0	R(X)
	1	X=X+1
	1	W(X)
	1	Commit
R(X)	1	
Commit	1	

Le 2 letture sono tra loro inconsistenti!

- Anche in questo caso si possono avere gravi conseguenze
- Lo stesso problema si presenta per transazioni di "analisi"

  Ad esempio T1 somma l'importo di 2 conti correnti mentre T2 esegue un trasferimento di fondi dall'uno all'altro (T1 potrebbe quindi riportare un totale errato)

Riepilogando…

- **Lost Update**

  (dipendenza write → write)

  perdita di modifiche da parte di una transazione a causa di un aggiornamento effettuato da un'altra transazione

- **Dirty Read** (Uncommited Dependency)

  (dipendenza write → read)

  dipendenza di una transazione da un'altra non ancora completata con successo e lettura di dati inconsistenti

- **Unrepeatable Read** (Inconsistent analysis)

  (dipendenza read → write)

  analisi inconsistente di dati da parte di una transazione causata da aggiornamenti prodotti da un'altra

Più precisamente….

- **Lost Update** (dipendenza write → write)

  Una transazione T_i legge un dato O con versione v, denotato [O, v], ne produce una nuova versione [O, v+1], e un'altra transazione T_i legge la vecchia versione [O, v] e produce la versione [O, v+2].

- **Dirty Read** (dipendenza write → read)

  T_i legge [O, v], produce una nuova versione [O, v+1] che viene letta da un'altra transazione T_i, ma [O, v+1] non permane al termine dell'esecuzione di T_i.

- **Unrepeatable Read** (dipendenza read → write)

  T_i legge [O, v] e T_i produce una nuova versione [O, v+1], che viene letta nuovamente da T_i dopo il commit di T_i.

Phantom Row

- Questo caso si può presentare quando vengono inserite o cancellate tuple che un'altra transazione dovrebbe logicamente considerare
  - Nell'esempio la tupla t4 è un "phantom", in quanto T1 "non la vede"

T1:

```
UPDATE Prog
SET    Sede = 'Firenze'
WHERE  Sede = 'Bologna'
```

T2:

```
INSERT INTO Prog
VALUES ('P03','Bologna')
```

Progetti

CodProg	Citta	
P01	Milano	t1
P01	Bologna	t2
P02	Bologna	t3
P03	Bologna	t4

T1 "non vede" questa tupla!

T1	T2
R(t2)	
R(t3)	
…	
W(t2)	
W(t3)	
	Insert(t4)
…	
Commit	
	Commit

- Il problema del del Phantom Row è un caso particolare di Dirty Read
  - L'anomalia dipende dall'esecuzione di un'operazione sui dati basata su modifiche effettuate da una transazione non ancora conclusa

Dirty Read e Cascading Abort

- Quando una transazione T fallisce, il Recovery Manager deve eliminare gli effetti da essa prodotti che riguardano, in generale, i dati alterati da T e da transazioni che hanno eseguito "dirty read", leggendo dati scritti da T. In quest'ultimo caso si verifica il fenomeno detto cascading abort, ovvero una catena di fallimenti delle transazioni alterate da T e delle transazioni eventualmente alterate da queste, e così via.

T1	time	T2
read(x) x=1	t1	
write(x) x:=2	t2	
	t3	read(x) x=2
	t4	read(y) y=1
	t5	write(y) y:=3
	t6	
rollback	t7	

Poiché T1 fallisce, il dato x viene ripristinato al valore 1 e, avendo T2 eseguito una "dirty read", anche le sue azioni devono essere annullate, ripristinando y al valore 1.

Cascading Abort

- Il problema nasce quando il cascading abort coinvolge anche transazioni che hanno eseguito commit.

T1	time	T2
read(x) x=1	t1	
write(x) x:=2	t2	
	t3	read(x) x=2
	t4	read(y) y=1
	t5	write(y) y:=3
	t6	commit
rollback	t7	

- Il Recovery Manager avrebbe due possibilità, entrambe non corrette:
  - lasciare permanenti gli effetti di T2, violando la semantica di read
  - annullare gli effetti prodotti da T2, violando la semantica di commit

Recoverability

- Per ovviare ai problemi dovuti a cascading abort indotto da dirty read, si introduce il requisito di recoverability (ripristinabilità).
- Si dice che una transazione T è "recoverable" se le viene impedito di eseguire commit prima che tutte le transazioni, che scrivono valori *letti* da T, eseguano commit o abortiscano. Più precisamente:

  T_i legge x da una transazione attiva T_i se:

  1. T_i legge x dopo che T_i l'ha modificato;
  2. T_i non fallisce prima che T_i legga x;
  3. ogni transazione che modifica x fra il tempo che intercorre tra la modifica di x da parte di T_i e la lettura di x da parte di T_i, fallisce prima che T_i legga x.

  Una transazione T_i legge da T_i se T_i legge qualche dato scritto da T_i.

- Un'esecuzione concorrente di transazioni è detta recoverable se, per ogni T che esegue commit, quest'ultimo segue il commit di ogni transazione da cui T legge.

Recoverability: esempio

- Al tempo t6 vi è una richiesta di commit di T2 che viene ritardata, in quanto T2 legge da T1 che è ancora attiva.
- Il commit di T1 rende possibile quello di T2 all'istante t8.

T1	time	T2
read(x) x=1	t1	
write(x) x:=2	t2	
	t3	read(x) x=2
	t4	read(y) y=1
	t5	write(y) y:=3
	t6	
commit	t7	
	t8	commit

Avoiding Cascading Abort

- Il requisito di recoverability evita di far fallire transazioni che hanno eseguito commit, ma non rimuove il problema del cascading abort per transazioni attive, la cui gestione richiede meccanismi sofisticati e inoltre implica la possibilità di far abortire in modo non controllato parecchie transazioni a causa dell'abort di altre.
- Per evitare il cascading abort è sufficiente impedire che una transazione esegua dirty read. Ciò implica ritardare ogni read[x] fino a che tutte le transazioni che hanno operato write[x] sono state completate con successo o sono state fatte abortire.

T1	time	T2
read(x) x=1	t1	
write(x) x:=2	t2	
	t3	read(y) y=1
commit	t4	
	t5	read(x) x=2
	t6	write(y) y:=3
	t7	commit

Come garantire Isolation

- Una comune tecnica usata dai DBMS per evitare i problemi visti consiste nell'uso di lock
  - I lock ("blocchi") sono un meccanismo comunemente usato dai sistemi operativi per disciplinare l'accesso a risorse condivise
  - Per eseguire un'operazione è prima necessario "acquisire" un lock sulla risorsa interessata (ad es. una tupla, o una relazione, o un indice, o…)
  - La richiesta di lock è implicita, e quindi non è visibile a livello SQL

    …ma anche in SQL si possono imporre scelte…

- Nei DBMS vi sono vari tipi di lock; quelli di base sono:
  - S (**Shared**): un lock condiviso è necessario per leggere
  - X (**eXclusive**): un lock esclusivo è necessario per scrivere/modificare

Compatibilità dei lock

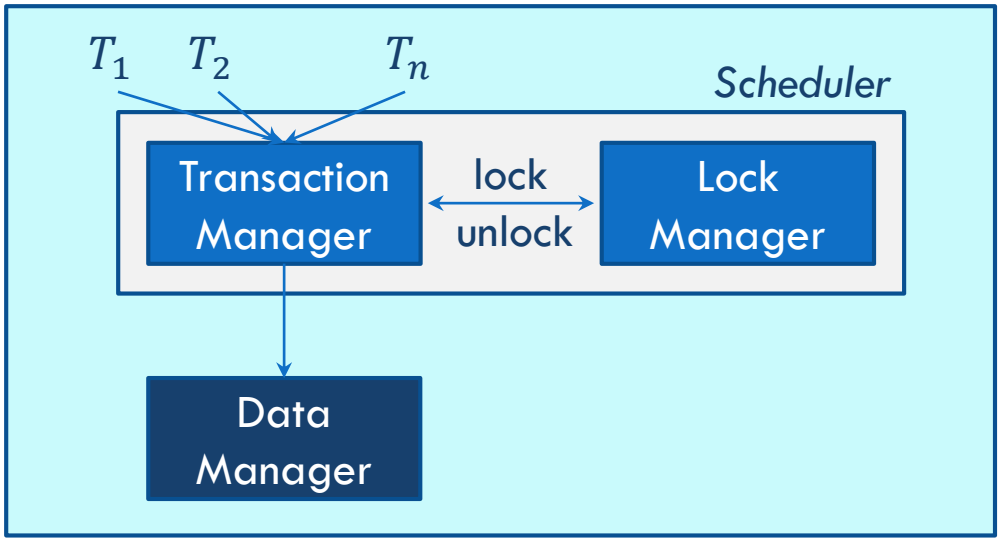
- Il Lock Manager è un modulo del DBMS che si occupa di tener traccia delle risorse correntemente in uso, delle transazioni che le stanno usando (e in che modo) e delle transazioni che ne hanno fatto richiesta.
- Quando una transazione T vuole operare su un dato Y, viene inviata la richiesta di acquisizione del lock corrispondente al Lock Manager
- Il lock viene accordato a T in funzione della seguente tabella di compatibilità

T richiede su Y un lock di tipo	Su Y un'altra transazione detiene un lock di tipo: S	Su Y un'altra transazione detiene un lock di tipo: X
S	OK	NO
X	NO	NO

- Quando T ha terminato di usare Y, può rilasciare il lock (unlock(Y))

Lock Manager

- Lo Scheduler della maggior parte dei sistemi reali è di fatto realizzato estendendo il Transaction Manager (TM) con un Lock Manager (LM).
- Quando TM riceve un richiesta di read o write da una transazione invia l'appropriata richiesta di lock a LM. Solo quando LM conferma l'attribuzione del lock, TM inoltra la richiesta al Data Manager (DM).



- LM gestisce una lock table, rispondendo a richieste del tipo:
  - LOCK (transaction_id, data_item, lock_mode)
  - UNLOCK (transaction_id, data_item)

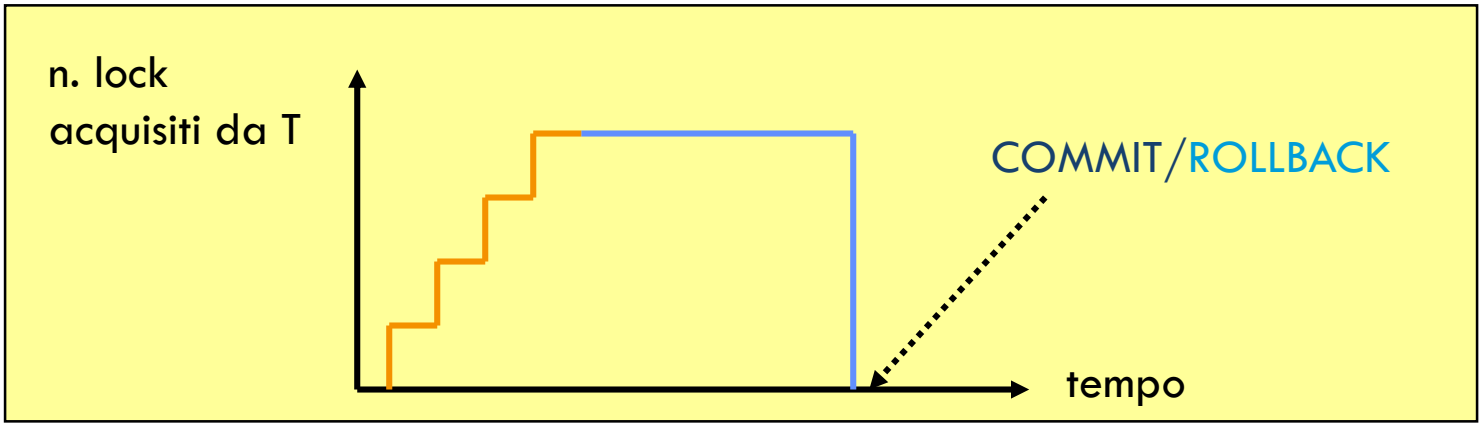
Strict 2-phase locking (Strict 2PL)

- Il two phase locking (2PL) è il protocollo più usato per lo scheduler nei DBMS centralizzati.
- In particolare, viene usata una variante, detta Strict 2PL, che opera in base alle seguenti regole:
  - A seguito di una richiesta di operazione su un certo dato da parte della transazione T, viene assegnato il lock corrispondente solo se non vi sono lock incompatibili sul dato stesso detenuti da altre transazioni.
  - Un lock acquisito dalla transazione T non può essere rilasciato almeno fino alla conferma dell'avvenuta esecuzione da parte di DM dell'operazione corrispondente.
  - Lo scheduler rilascia i lock acquisiti da T in una volta sola, al termine della transazione; più precisamente, quando il Data Manager conferma l'avvenuta esecuzione di COMMIT o ABORT.

Strict 2PL e isolation

- Si può dimostrare che se:
  - Una transazione prima acquisisce tutti i lock necessari
  - Rilascia i lock solo al termine dell'esecuzione (COMMIT o ROLLBACK)

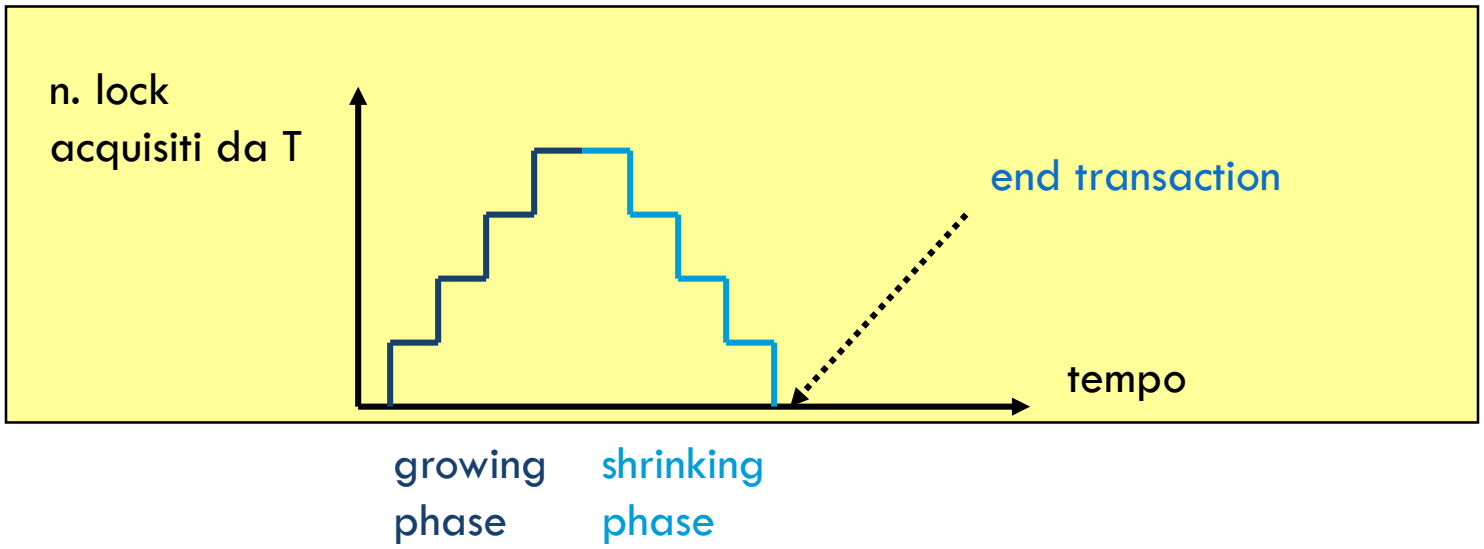
  allora è garantito isolamento completo delle transazioni.



- Come effetto collaterale si possono verificare deadlock, ossia situazioni di stallo, che vengono risolte facendo abortire una transazione

2-phase locking (2PL)

- La versione base del protocollo 2PL (da cui il nome) richiede solo che, dopo il rilascio di un lock, una transazione non ne possa acquisire altri. Da ciò deriva che una transazione può essere suddivisa in due fasi: durante la prima (**growing phase**), la transazione può solo aumentare il numero dei lock posseduti; al rilascio del primo lock inizia la seconda fase (**shrinking phase**), durante la quale la transazione può solo rilasciare lock e non acquisirne di nuovi.



2-phase locking e isolation

- 2PL garantisce isolamento delle transazioni solo in assenza di malfunzionamenti, infatti permette letture sporche.

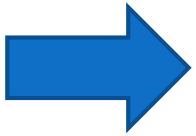
  Esempio: T1 esegue una dirty read, e l'esecuzione non è ripristinabile.

T1	time	T2
	t1	Xlock(x)
	t2	write(x)
	t3	unlock(x)
Slock(x)	t4	
read(x)	t5	
unlock(x)	t6	
commit	t7	
	t8	rollback

Strict 2PL: no Lost Update

- Lo schedule visto precedentemente si modifica come segue:

T1	X	T2
R(X)	1	
X=X-1	1	
	1	R(X)
	1	X=X-1
W(X)	0	
Commit	0	
	0	W(X)
	0	Commit

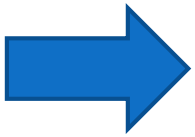


T1	X	T2
S-lock(X)	1	
R(X)	1	
X=X-1	1	
	1	S-lock(X)
	1	R(X)
	1	X=X-1
X-lock(X)	1	
wait	1	X-lock(X)
wait	1	wait

- Né T1 né T2 riescono ad acquisire il lock per poter modificare X (restano in attesa, "wait"); si verifica quindi un deadlock. Se il sistema decide di far abortire, ad esempio, T2, allora T1 può proseguire.

Strict 2PL: no Dirty Read

T1	X	T2
R(X)	0	
X=X+1	0	
W(X)	1	
	1	R(X)
Rollback	0	
	0	…
	0	Commit

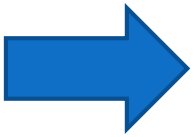


T1	X	T2
S-lock(X)	0	
R(X)	0	
X=X+1	0	
X-lock(X)	0	
W(X)	1	
	1	S-lock(X)
	0	wait
Rollback	0	wait
Unlock(X)	0	wait
	0	R(X)

- In questo caso l'esecuzione corretta richiede che T2 aspetti la terminazione di T1 prima di poter leggere il valore di X

Strict 2PL: repeatable Read

T1	X	T2
R(X)	0	
	0	R(X)
	1	X=X+1
	1	W(X)
	1	Commit
R(X)	1	
Commit	1	



T1	X	T2
S-lock(X)	0	
R(X)	0	
	0	S-lock(X)
	0	R(X)
	0	X=X+1
	0	X-lock(X)
	0	wait
R(X)	0	wait
Commit	0	wait
Unlock(X)	0	wait
	1	W(X)

- Anche in questo caso T2 viene messa in attesa, e T1 ha quindi la garanzia di poter leggere sempre lo stesso valore di X

Assenza di Phantom Row

- Questo è, tra quelli visti, il problema più difficile da risolvere. Le soluzioni adottabili sono varie, e differiscono in complessità e livello di concorrenza che permettono:
  - Si può acquisire un S-lock su tutta la table, e poi richiedere gli X-lock per le tuple che si vogliono modificare
  - Si introduce un nuovo tipo di lock, detto "predicate lock", che riguarda tutte le tuple che soddisfano un predicato

    (Sede = 'Bologna' nell'esempio)
  - Se esiste un indice su Sede, si pone un lock sulla foglia che contiene 'Bologna'
- Nei DBMS la situazione è in realtà più complessa, sia per i tipi di lock presenti, sia per la "granularità" con cui i lock possono essere richiesti e acquisiti (a livello di attributo, tupla, pagina, relazione, …)

Controllo della concorrenza in SQL

- Benché il problema di gestire transazioni concorrenti sia a carico del DBMS, con conseguente semplificazione del codice delle applicazioni, SQL mette a disposizione due meccanismi di base per influenzare il modo con cui una transazione viene eseguita
  - Come visto, la richiesta di lock è implicita; tuttavia, poiché acquisire i lock e gestirli richiede tempo e spazio, se una transazione sa di dover elaborare molte tuple di una relazione può richiedere esplicitamente di porre un lock (SHARED o EXCLUSIVE **MODE**) sull'intera relazione

```
LOCK TABLES
    tbl_name [[AS] alias] lock_type
    [, tbl_name [[AS] alias] lock_type] ...

lock_type: {
    READ [LOCAL]
  | [LOW_PRIORITY] WRITE
}

UNLOCK TABLES
```

If you lock a table explicitly with LOCK TABLES, any tables related by a foreign key constraint are opened and locked implicitly.
For foreign key checks, a shared read-only lock (LOCK TABLES READ) is taken on related tables.
For cascading updates, a shared-nothing write lock (LOCK TABLES WRITE) is taken on related tables that are involved in the operation.

  - è inoltre possibile scegliere esplicitamente il "livello di isolamento" a cui si vuole eseguire la transazione

Livelli di isolamento in SQL

- Scegliere di operare a un livello di isolamento in cui si possono presentare dei problemi ha il vantaggio di aumentare il grado di concorrenza raggiungibile, e quindi di migliorare le prestazioni
- Lo standard SQL definisce 4 livelli di isolamento:

Isolation Level	Phantom	Unrepeatable Read	Dirty Read	Lost Update
Serializable	NO	NO	NO	NO
Repeatable Read	YES	NO	NO	NO
Read Committed	YES	YES	NO	NO
Read Uncommitted	YES	YES	YES	NO

Transazioni in MySQL

- In MySQL, la transazione inizia con l'istruzione **BEGIN WORK** e termina con un comando **COMMIT** o **ROLLBACK**. I comandi SQL tra i comandi di inizio e fine costituiscono l'insieme di comandi della transazione.
- È possibile controllare il comportamento di una transazione impostando la variabile di sessione **AUTOCOMMIT**.
  - Se AUTOCOMMIT = 1 (default), ogni comando SQL è considerato una transazione completa e viene eseguito automaticamente il COMMIT al termine;
  - Se AUTOCOMMIT = 0 (si può impostare con il comando **SET AUTOCOMMIT = 0**), tutti i comandi successivi operano come all'interno di una transazione e non viene eseguito nessun commit finché non viene eseguito esplicitamente il comando COMMIT.
- Per poter utilizzare le transazioni è necessario che le tabelle del database siano create in modo particolare (impostando **TYPE = InnoDB** in fase di creazione della tabella).

Locking in MySQL: Shared vs. Exclusive

- InnoDB implementa lo standard row-level locking sulla base di due diversi tipi di lock: shared (S) o exclusive (X).
  - Uno shared lock (S) permette a una transazione che lo detiene di leggere una riga.
  - Un exclusive lock (X) permette a una transazione che lo detiene di aggiornare o cancellare una riga.
- Se una transazione T1 possiede uno shared lock (S) su una riga *r*, tutte le successive richieste da parte di una generica transazione T2 sono gestite come segue:
  - Una richiesta di un S lock da parte di T2 viene soddisfatta immediatamente (T1 e T2 avranno contemporaneamente un S lock su *r*).
  - Una richiesta di X lock da parte di T2 non può essere soddisfatta immediatamente.
- Se una transazione T1 possiede un X lock su una riga *r*, qualsiasi richiesta da parte di una generica transazione T2 non può essere soddisfatta immediatamente. Qualsiasi transazione T2 dovrà attendere che T1 rilasci il lock (X) sulla riga *r*.

Locking in MySQL: Intention Locks (1)

- InnoDB supporta una politica di locking a diversi livelli di granularità attraverso l'utilizzo di Intention Locks, lock a livello di tabella che indicano quale tipo di lock (S o X) una transazione richiederà per una riga *r* di una specifica tabella. Esistono due tipi di intention lock:
  - Intention Shared Lock (IS) - una transazione intende richiedere uno Shared Lock su singole righe di una tabella.
  - Intention Exclusive Lock (IX) - una transazione intende richiedere uno Shared Lock su singole righe di una tabella.

`SELECT ... FOR SHARE` imposta un IS lock

`SELECT ... FOR UPDATE` imposta un IX lock

Locking in MySQL: Intention Locks (2)

- Il protocollo per gli intention lock è il seguente:
  - Affinché una transazione possa acquisire un S lock su una riga di una tabella, deve avere prima acquisito un IS lock (o un IX lock) sulla tabella stessa.
  - Affinché una transazione possa acquisire un X lock su una riga di una tabella, deve avere prima acquisito un IX lock sulla tabella stessa.
  - Gli intention lock non bloccano nulla se non le richieste di lock a livello di intera tabelle (es. LOCK TABLES ... WRITE).
  - I lock a livello di tabella sono gestiti secondo la seguente tabella:

	X	IX	S	IS
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	Compatible	Conflict	Compatible
S	Conflict	Conflict	Compatible	Compatible
IS	Conflict	Compatible	Compatible	Compatible

https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html

Livelli di isolamento in MySQL (1)

- InnoDB supporta tutti e quattro i livelli di isolation:
  - READ_UNCOMMITTED,
  - READ_COMMITTED,
  - REPEATABLE_READ
  - SERIALIZABLE.
- Il livello di default è REPEATABLE_READ. All'interno della stessa transazione sono garantite letture consecutive consistenti con la prima lettura effettuata. Se una stessa interrogazione viene eseguita più volte, il risultato prodotto sarò sempre il medesimo.

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic: {
    ISOLATION LEVEL level
  | access_mode
}

level: {
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
}

access_mode: {
     READ WRITE
   | READ ONLY
}
```

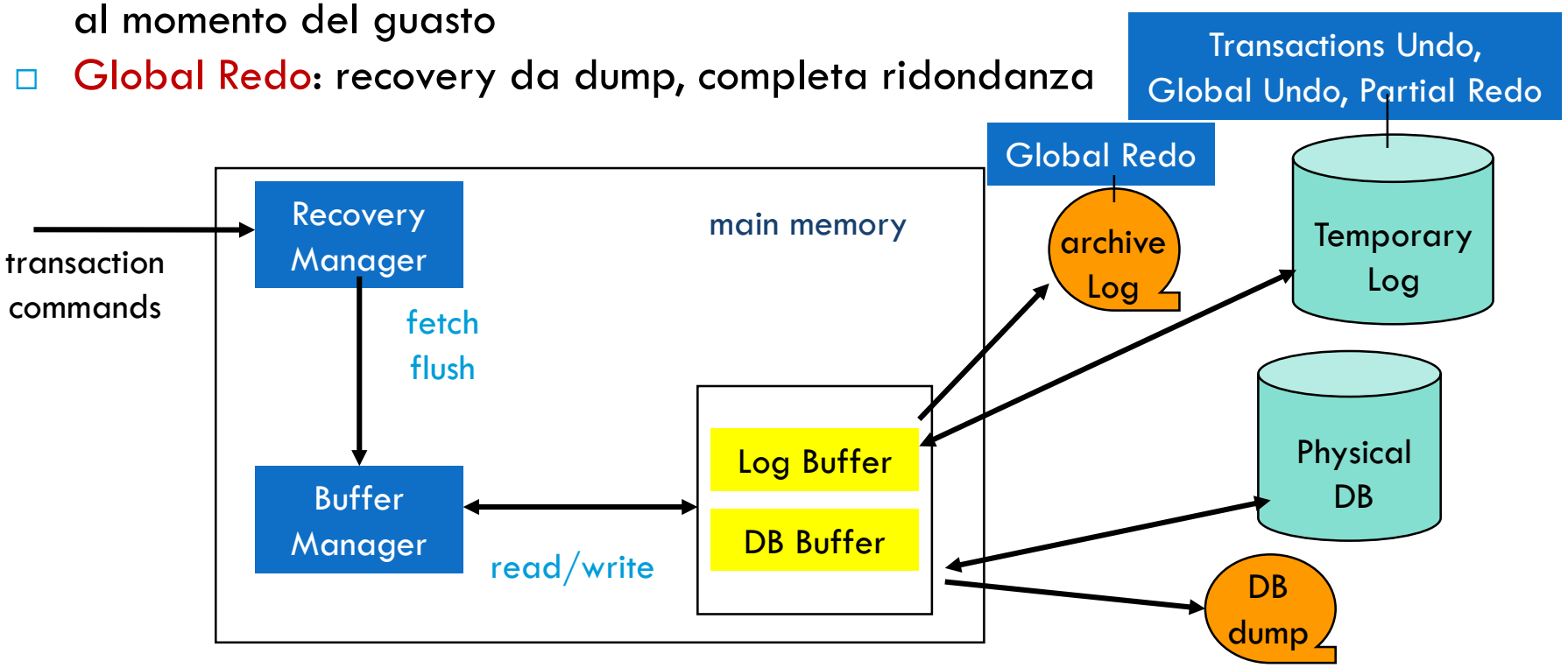
https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html

Atomicity e Durability: convivere con i guasti

- L'"altra faccia" della gestione delle transazioni riguarda il trattamento dei "guasti" (failure), ovvero di tutti quegli eventi anomali che possono pregiudicare il corretto funzionamento delle transazioni.
- I tipi di malfunzionamenti sono essenzialmente 3:
  - Transaction failure: è il caso in cui una transazione abortisce; l'interruzione può essere causata da una situazione prevista dal programma (esempio: violazione vincoli di integrità, tentativo di accesso a dati protetti) o rilevata dal gestore (deadlock). L'interruzione non comporta perdita di dati in memoria temporanea o permanente.
  - System failure: interruzione di transazioni attive derivante da un'anomalia hardware o software dell'unità centrale o di una periferica (un esempio tipico è l'interruzione dell'alimentazione elettrica). Si assume che il contenuto della memoria permanente sopravviva, mentre si considera perso il contenuto della memoria temporanea.
  - Media failure: in questo caso il contenuto (persistente) della base di dati viene danneggiato.

Azioni di ripristino

- Transaction Undo: in seguito ad abort di una transazione
- Global Undo: ripristino da un system failure, interessa transazioni non ancora completate al momento del guasto
- Partial Redo: ripristino da un system failure, interessa transazioni completate al momento del guasto
- Global Redo: recovery da dump, completa ridondanza



Atomicity e Durability

- Per far fronte ai malfunzionamenti, un DBMS fa uso di diversi strumenti, in particolare:
  - DataBase Dump: copia di archivio del DB (o parte di esso)
  - Log file ("giornale"): file in cui vengono registrate le operazioni di modifica eseguite dalle transazioni
    - Se una pagina P del DB viene modificata da T, il log contiene un record del tipo
      (LSN, T, PID, before(P), after(P), prevLSN) dove:
      - LSN = Log Sequence Number (n. progressivo del record)
      - T = identificatore della transazione
      - PID = identificatore della pagina modificata
      - before(P) = è la cosiddetta before image di P, ovvero il contenuto di P prima della modifica
      - after(P) = è l'after image di P, ossia il contenuto di P dopo la modifica
      - prevLSN = LSN del precedente record del Log relativo a T

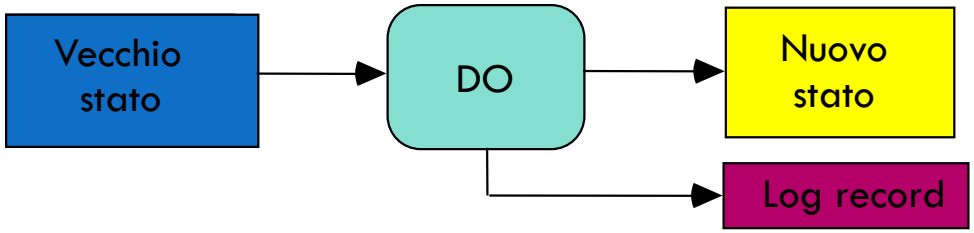
Esempio di Log

- Il Log contiene anche record che specificano l'inizio (BEGIN) di una transazione e la sua terminazione (COMMIT o ABORT)

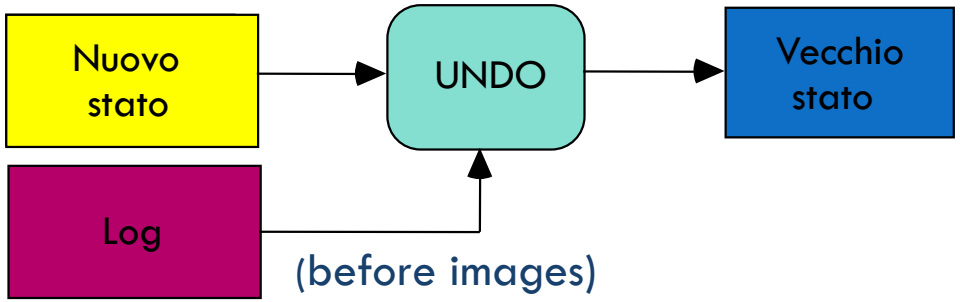
LSN	T	PID	before(P)	after(P)	prevLSN
…					
235	T1	BEGIN			-
236	T2	BEGIN			-
237	T1	P15	(abc, 10)	(abc, 20)	235
238	T2	P18	(def, 13)	(ghf, 13)	236
239	T1	COMMIT			237
240	T2	P19	(def, 15)	(ghf, 15)	238
241	T3	BEGIN			-
242	T2	P19	(ghf, 15)	(ghf, 17)	240
243	T3	P15	(abc, 20)	(abc, 30)	241
244	T2	ABORT			242
245	T3	COMMIT			243
…					

Transaction Abort

- Ogni azione di aggiornamento comporta l'aggiunta di una entrata nel log file. Non è necessario appendere record per azioni di pura lettura

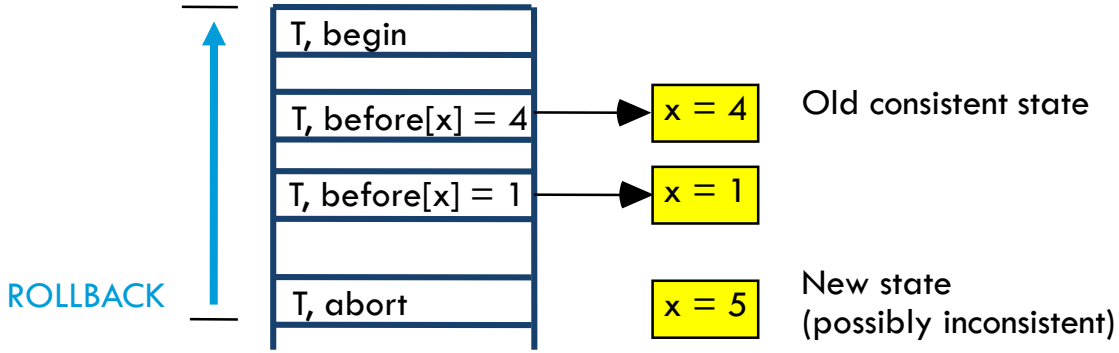


- Il rollback di una transazione T abortita viene gestito prima scrivendo un abort record sul Log, e quindi "disfacendo" (UNDO action) le azioni di T che hanno modificato la base di dati, secondo il seguente schema:



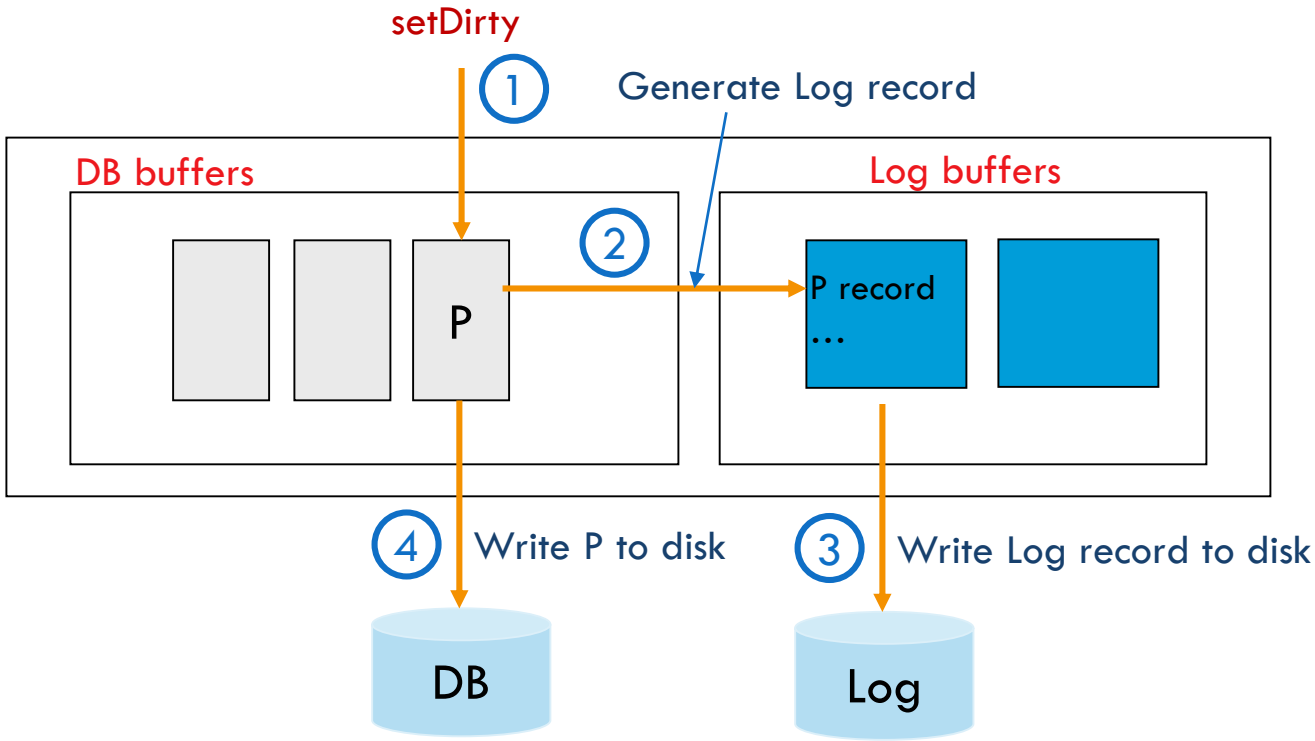
Write Ahead Log protocol

- WAL Protocol: se una transazione deve essere disfatta, è indispensabile che le before image siano registrate nel Log prima che le relative pagine del DB siano effettivamente modificate, ovvero:
  - Si scrive prima sul Log e poi sul DB
  - Forzare la scrittura del log record prima che la corrispondente pagina dati modificata venga registrata su disco garantisce il rispetto della proprietà "atomicity".
  - Forzare la scrittura di tutti i log record di una transazione prima del commit garantisce il rispetto della proprietà "durability".
- In caso di fallimento di una transazione T la lettura del log avviene a ritroso, e termina quando si incontra il begin record di T.



Implementazione protocollo WAL

- La responsabilità di garantire il rispetto del protocollo WAL è del Buffer Manager, che gestisce, oltre ai buffer del DB, anche quelli del Log.
- In figura viene riportato l'ordine in cui si succedono le varie operazioni relative alla modifica di una pagina P.

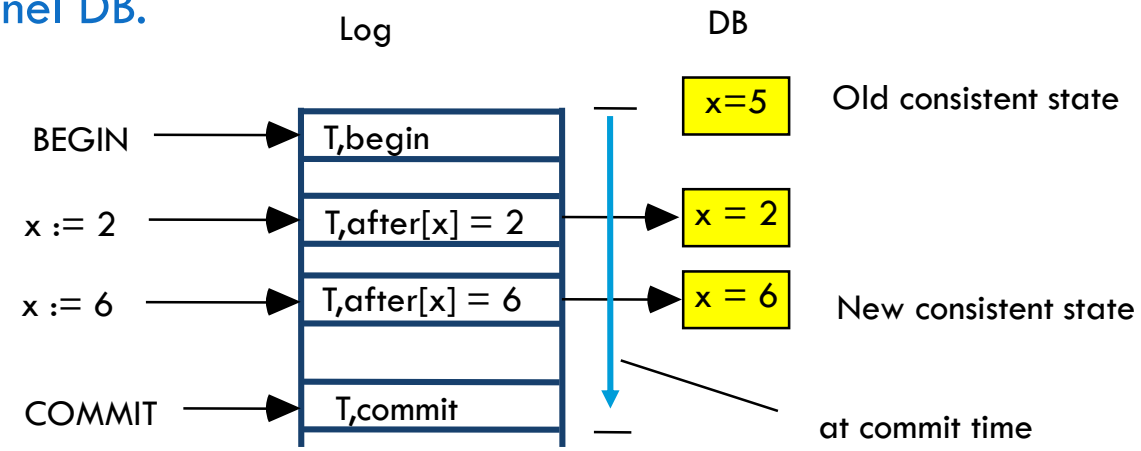


Gestione dei buffer

- Quando una transazione T modifica una pagina P, il Buffer Manager ha 2 possibilità:
  - Politica NO-STEAL: Mantenere la pagina P nel buffer, e attendere che T abbia eseguito **COMMIT** prima di scriverla su disco
  - Politica STEAL: Scrivere P quando "più conviene" (per liberare il buffer o per ottimizzare le prestazioni di I/O), eventualmente anche prima della terminazione di T
- E' evidente che nel primo caso non ci sarà mai bisogno di eseguire l'"UNDO" di transazioni che abortiscono, ma si rischia di esaurire lo spazio a disposizione in memoria centrale (in quanto una transazione non può "rubare" i buffer ad altre transazioni ancora in esecuzione).

STEAL / NO STEAL

- STEAL: implica che il Buffer Manager può scrivere le pagine modificate da T nel DB prima del commit di T
- NO STEAL: implica che il Recovery Manager impedisce al Buffer Manager la scrittura sul DB delle pagine modificate da T fino al commit di T, e pertanto:
  - la nuova versione di una pagina modificata (after image) non viene memorizzata subito nel DB, ma solo sul Log. L'esecuzione del commit avviene scrivendo il commit record sul Log, e quindi copiando le after image nel DB.



  - La politica NO STEAL migliora le prestazioni in caso di frequenti Transaction e System Failure, ma aumenta la complessità del commit.

Transaction failure

- Con la politica STEAL, se una transazione T abortisce è possibile che alcune pagine da essa modificate siano già state scritte su disco
- Per annullare (UNDO) queste modifiche si scandisce il Log a ritroso (usando i prevLSN) e si ripristinano nel DB le before image delle pagine modificate da T

LSN	T	PID	before(P)	after(P)	prevLSN
…					
236	T2	BEGIN			-
237	T1	P15	(abc, 10)	(abc, 20)	235
238	T2	P18	(def, 13)	(ghf, 13)	236
239	T1	COMMIT			237
240	T2	P19	(def, 15)	(ghf, 15)	238
241	T3	BEGIN			-
242	T2	P19	(ghf, 15)	(ghf, 17)	240
243	T3	P15	(abc, 20)	(abc, 30)	241
244	T2	ROLLBACK			242

System failure

- La terminazione effettiva di una transazione T_i che richiede di eseguire commit si ha quando sul Log viene scritto il commit record relativo. All'atto di questa operazione la transazione può definitivamente considerarsi conclusa, e i suoi lock essere rilasciati.
- In caso di fallimento di sistema, il Recovery Manager attiva la procedura di restart che provvede a riportare il DB in uno stato consistente, facendo l'Undo di tutte le transazioni attive all'atto del guasto, ovvero quelle che non hanno il commit record nel Log:

  If (T_i,commit) non è nel Log then Undo(T_i)

- La necessità di dover eseguire Undo per transazioni non terminate con successo deriva dal fatto che tali transazioni possono modificare il DB prima di eseguire commit (politica di update immediato o a 1 fase o STEAL).
- In alcuni sistemi (es.: Ingres), si evita sempre l'Undo di transazioni (NoUndo), adottando la politica NO STEAL di modifica del DB, nota anche come update ritardato o a 2 fasi.

Checkpoint

- La procedura di restart è quella che si occupa di riportare il DB in uno stato consistente a fronte di system failure; per ridurre i tempi di restart, periodicamente si può eseguire un "checkpoint", ovvero una scrittura forzata su disco delle pagine modificate
  - L'esecuzione del checkpoint viene registrata scrivendo sul Log un record CKP (checkpoint)
- In questo modo se T ha eseguito COMMIT prima del checkpoint si è sicuri che T non deve essere rifatta

T1 e T2 non devono essere rifatte!

LSN	T	PID	before(P)	after(P)	prevLSN
237	T3	P15	…	…	…
238	T2	P18	…	…	…
239	T1	P17	…	…	…
240	T1	COMMIT			…
241	T2	COMMIT			…
242		CKP			
243	T3	P19	…	…	…
244	T3	COMMIT			…

Media failure

- Nel caso di media failure si ha un ripristino che usa una copia archiviata del DB (DataBase Dump)
- Facendo uso del Log, si rifanno quindi tutte le transazioni che hanno eseguito COMMIT e si disfano tutte quelle per cui il record di COMMIT non si trova nel Log
- È evidente che se il Log dovesse corrompersi tutte le operazioni di ripristino non potrebbero funzionare correttamente
- Per tale motivo è comune mantenere il Log almeno in duplice copia

Domande?