Gestione delle transazioni

Le **transazioni** rappresentano **unità di lavoro** elementari che **modificano** il contenuto di una base di dati. Si osservino degli esempi possibili:

start transaction

update SalariImpiegati

*set conto = conto * 1.2*

where (CodiceImpiegato = 123)

commit work;

Come si può vedere, le transazioni sono comprese tra una start transaction ed una commit/rollback, come osservabile nell'esempio:

start transaction

update SalariImpiegati

*set conto = conto * 1.2*

where (CodiceImpiegato = 123)

if var > 0 then ***commit work;***

else ***rollback work;***

Le transazioni possiedono delle proprietà, dette **ACID**:

- **Atomicità**: la transazione deve essere eseguita con la regola del *tutto o niente*
- **Consistenza**: la transazione deve lasciare la base di dati 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

La gestione delle transazioni può essere vista come una suddivisione di due gestioni differenti:

- **Gestione dell'affidabilità**: garantisce *atomicità* e *persistenza*, utilizzando **log** e **checkpoint**
- **Gestione della concorrenza**: garantisce l'*isolamento* in caso di esecuzione **concorrente** di più transazioni

Si osservino ora dei concetti teorici inerenti alle transazioni.
Dato un **insieme di transazioni** T_1, T_2, ..., T_n di cui ciascuna formata da un certo insieme di operazioni di scrittura (w_i) e lettura (r_i), si definisce **schedule** la sequenza di operazioni di lettura/scrittura di tutte le transazioni **cosi come eseguite sulla base di dati**.

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

1. le transazioni sono **eseguite uno alla volta**, scenario non realistico
2. le transazioni sono **completamente indipendenti** l'una dall'altra, impossibile

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

In un sistema reale, le **transazioni vengono eseguite in concorrenza** per ragioni di efficienza e scalabilità. Tuttavia, l'esecuzione concorrente determina un insieme di **problematiche** che devono essere gestite. Alcuni problemi possono essere:

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

In questo caso, l'esecuzione della seconda transazione invalida il risultato della prima transazione

2. **Lettura sporca**

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

T2 legge 4!

Nel caso di rollback in una transazione, le transazioni che utilizzano quei dati compromessi si trovano in uno stato di errore

3. **Letture inconsistenti**

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

In questo caso, viene effettuato un read precedente alla seconda transazione, che invalida il risultato richiesto

4. **Aggiornamento fantasma**

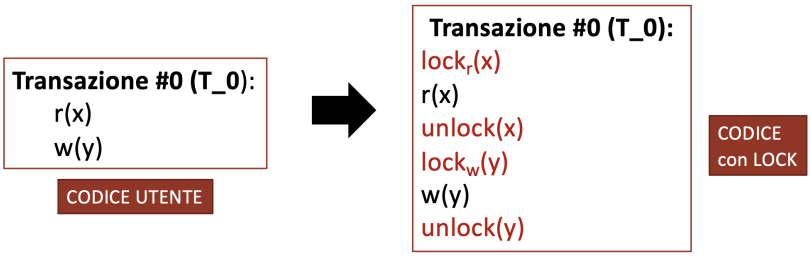
	Transazione1 (T1)	Transazione2 (T2)
Vincolo: x+y+z deve essere = a 1000	Read(x)	
		Read(y)
	Read(y)	
		y=y-100
		Read(z)
		z=z+100
		Write(y), Write(z)
		Commit work
Vincolo violato!!	Read(z) s=x+y+z; Commit work	

In questo caso, la prima transazione non considera i valori modificati dalla seconda transazione, ottenendo un risultato errato

Bisogna quindi stabilire un metodo per il **controllo della concorrenza**. I DBMS commerciali usano il meccanismo dei **lock**. Per poter effettuare una qualsiasi operazione di lettura/scrittura su una risorsa, tabella o valore di una cella, **è necessario aver precedenetemente acquisito il controllo, o lock, sulla risorsa stessa**.
Come visto in precedenza in altri corsi, il **lock in lettura** consente un accesso condiviso alla risorsa, mentre il **lock in scrittura** segue il concetto di mutua esclusione.
Su ogni lock possono essere definite **due operazioni**:

- **Richiesta** del lock in lettura/scrittura

- **Rilascio** del lock acquisito in precedenza



Il **lock manager** è un componenete del DBMS responsabile di **gestire i lock alle risorse del DB, e di rispondere alle richieste delle transazioni**. Per ciascun oggetto x del DBMS, possiamo accedere a tre strutture dati:

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

Davanti ad una richiesta, il lock manager esegue diversi step:

1. Riceve una richiesta (r_lock / w_lock / unlock) da una transazione T, su un oggetto x, che può essere una tabella, una colonna, etc
2. Controlla la **tabella stato/azione**

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

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

Two Phase Lock (2PL)

In un **Two Phase Lock**, 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**.
Ogni schedule che rispetta il 2PL è anche **serializzabile**. Inoltre, ogni schedule che rispetta il 2PL non può incorrere in **configurazioni erronee** dovuta ad aggiornamenti fantasma, letture inconsistenti o perdite di aggiornamento; è ancora presente il problema della lettura sporca.

Strict Two Phase Lock (S2PL)

In un **Strict Two Phase Lock**, **i lock di una transazione sono rilasciati solo dopo aver effettuato le operazioni di commit/rollback**.
Uno schedule che rispetta lo S2PL **eredita tutte le proprietà del 2PL**, ed inoltre **non** presenta anomalie causate da problemi di **lettura sporca**.

I due protocolli presentano un problema che li accomuna, potendo generare schedule con situazioni di **deadlock**. 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 (rollback) della transazione stessa**
2. **Deadlock avoidance**: prevenire le configurazioni che potrebbero portare ad un deadlock. Si può prevenire in due modi: attraverso l'utilizzo di **lock/unclock** di tutte le risorse allo stesso tempo; o attraverso l'utilizzo di **time stamp o di classi di priorità** tra transazioni (può provocare **starvation**)
3. **Deadlock detection**: utilizzare **algoritmi per identificare eventuali situazioni di deadlock**, e prevedere meccanismi di recovery dal deadlock. Si possono identificare attraverso l'utilizzo di **grafi delle richieste/risorse**, utilizzato per identificare la presenza di cicli. In caso di ciclo, si fa rollback delle transazioni coinvolte nel ciclo in modo da eliminare la mutua dipendenza

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

Si osservi ora la sintassi generale in MySQL:
Per **iniziare una transazione** e completarla:

SET AUTOCOMMIT = 0

START TRANSACTION

... (Statements SQL)

COMMIT/ROLLBACK

Per **configurare il livello di isolamento** di esecuzione:

SET TRANSACTION ISOLATION LEVEL

REPEATABLE READ | READ COMMITTED |

READ UNCOMMITTED | SERIALIZABLE

Nota Bene: le transazioni sono **utilizzabili solo** su tabelle di tipo **INNODB**.

E' possibile inoltre **gestire manualmente le operazioni di lock** su tabelle, anche se sconsigliato su tabelle di tipo INNODB:

LOCK TABLES

tabella READ | WRITE