

Transazioni in SQL

Nicola Vitacolonna
Corso di Basi di Dati
Università degli Studi di Udine

4 dicembre 2013

1 Introduzione

Informalmente, una **transazione** è una sequenza (arbitrariamente lunga) di operazioni che forma un'unità logica¹ di processamento di una base di dati, nel senso che dev'essere eseguita nella sua interezza per essere considerata corretta. In altre parole, quando si esegue una transazione, o tutte le operazioni che compongono la transazione sono eseguite, oppure non ne è eseguita nessuna. Ad esempio, una transazione può essere costituita da una singola istruzione **update**, che può causare la modificazione di varie tuple: in tal caso, o tutte le tuple sono aggiornate con successo oppure la base di dati rimane inalterata. Ciò è necessario a garantire la consistenza dei dati: una transazione non portata a termine completamente può lasciare la base di dati in uno stato in cui qualche vincolo d'integrità è violato. Atomicità e consistenza sono due delle quattro proprietà, considerate fondamentali, che le transazioni devono possedere:

Atomicità: la transazione è un'unità indivisibile, i cui effetti sono resi tutti visibili oppure la transazione non ha alcun effetto sulla base di dati.

Consistenza: ciascuna transazione deve portare la base di dati da uno stato consistente (in cui i vincoli d'integrità sono rispettati) a un altro stato consistente.

Isolamento: l'esecuzione di ciascuna transazione dev'essere indipendente dalla simultanea esecuzione di altre transazioni. In altre parole, le transazioni non devono interferire l'una con l'altra.

Persistenza: gli effetti di una transazione completata con successo devono essere registrati in modo permanente nella base di dati, anche a fronte di eventuali fallimenti del sistema.

Tali proprietà sono note con il nome di **proprietà acide** (**ACID**, acronimo inglese per *Atomicity, Consistency, Isolation, Durability*). Mette conto notare che la proprietà di consistenza non impone che *durante* l'esecuzione della transazione i vincoli d'integrità siano rispettati (anzi, talvolta è necessario che i vincoli siano temporaneamente violati). Tuttavia, i vincoli d'integrità devono essere soddisfatti prima e dopo l'esecuzione di ciascuna transazione. La proprietà d'isolamento ha come conseguenza che il risultato concorrente dell'esecuzione di più transazioni dev'essere "equivalente" al risultato ottenuto mediante una qualche esecuzione sequenziale delle medesime transazioni (secondo un'opportuna nozione di "equivalenza"). Infine, la persistenza impone che gli effetti di transazioni eseguite con esito positivo non siano perduti (i cambiamenti apportati da transazioni che falliscono per qualche ragione devono invece essere annullati).

Sebbene non sia possibile in generale garantire che tutte le transazioni siano sempre portate a termine con successo, è però possibile assicurarne le proprietà acide. Le operazioni fondamentali, a tale proposito, sono le operazioni di **commit** e **rollback** (o **abort**):

- un'operazione **commit** segnala il completamento con esito positivo di una transazione. La transazione è stata eseguita senza errori e tutti i cambiamenti apportati dalla transazione sono stati resi permanenti.

¹Dal punto di vista fisico, la transazione è tuttavia decomponibile.

- un'operazione **rollback** segnala il fallimento di una transazione. La base di dati potrebbe non essere in uno stato consistente e tutti i cambiamenti apportati dalla transazione devono essere disfatti.

Una transazione si dice **attiva** dal momento in cui la sua esecuzione ha inizio fino all'istante che precede l'operazione di commit/rollback. Fintantoché una transazione è attiva, tutti i cambiamenti che essa determina devono essere considerati provvisori e suscettibili di annullamento.

2 Esecuzione di transazioni in SQL

Lo standard SQL assume che l'inizio di una transazione sia implicito, vale a dire una qualunque istruzione SQL inizia una transazione automaticamente (a meno che non sia un'istruzione all'interno di un'altra transazione attiva). La fine di una transazione dev'essere invece denotata in modo esplicito mediante una delle due clausole **commit** o **rollback**. Una transazione può anche essere iniziata in modo esplicito con il comando **start transaction**. Ad esempio, la seguente è una transazione in SQL:

```
start transaction;
update accounts set balance = balance + 100.00 where acctnum = 12345;
update accounts set balance = balance - 100.00 where acctnum = 7534;
commit;
```

Il codice precedente garantisce che le due istruzioni di aggiornamento vadano entrambe a buon fine, o, in caso d'errore, che nessun record di **accounts** sia modificato.

Le transazioni non si possono annidare, ossia non è possibile eseguire **start transaction** dopo aver dato un comando **start transaction** e prima di una **commit** o **rollback**. I DBMS sono inoltre frequentemente configurati in modo da eseguire una commit o rollback automaticamente dopo ciascuna istruzione SQL (tale modalità è spesso chiamata **autocommit mode**) a meno che l'inizio della transazione non sia specificato in modo esplicito come nell'esempio precedente.

Si noti che l'operazione di rollback può essere richiesta in modo esplicito dalla transazione (ad esempio, a valle di qualche condizione per cui la transazione determina che non può proseguire), ma può anche essere attivata implicitamente dal DBMS a fronte di eventi non previsti dalla transazione.

Esempio 2.1 *Si consideri la seguente sessione SQL in cui le istruzioni **rollback** e **commit** sono date in modo esplicito:*

```
create table R (A int primary key, B char);
start transaction;
insert into R(A,B) values (1,'a'), (2,'b'), (3,'c');
update R set B = 'z';
rollback;
start transaction;
insert into R(A,B) values (10,'x'), (20,'y');
commit;
table R; -- Sinonimo di 'select * from R;'
```

Il risultato della precedente sequenza di istruzioni è

```
a | b
----+----
10 | x
20 | y
```

Si noti come le operazioni della prima transazione (che è stata annullata) non compaiano nel risultato.

3 Transazioni e vincoli d'integrità

Normalmente i vincoli d'integrità sono verificati a valle di ciascuna istruzione SQL. In alcuni casi, tuttavia, può essere necessario posporre la verifica al termine della transazione, perché alcune operazioni della transazione possono causare una violazione temporanea di qualche vincolo. Si consideri, ad esempio, il caso di uno schema con vincoli d'integrità referenziale circolari:

```
R(X, Y)
VNN: {Y}
CE: Y → S

S(W, Z)
VNN: {Z}
CE: Z → R
```

A partire da un'istanza vuota, è necessario inserire una tupla in R e una tupla in S in modo atomico, altrimenti uno dei due vincoli d'integrità referenziale è violato. Ciò può essere fatto con una transazione in cui si specifica che la verifica di consistenza della base di dati va temporaneamente sospesa fino al momento della commit. Affinché la transazione sia corretta, è necessario inoltre che i vincoli d'integrità siano stati definiti "differibili". Lo schema sarà perciò implementato in SQL come segue:

```
create table R (X int primary key, Y int not null);
create table S (W int primary key, Z int not null references R deferrable);
alter table R add foreign key (Y) references S deferrable;
```

Si noti l'uso della parola chiave **deferrable**. Senza tale parola chiave, il sistema assume che i vincoli debbano essere sempre verificati dopo ciascuna istruzione SQL all'interno di una transazione, anche in presenza della clausola **set constraints... deferred** (vedi oltre).

Il primo inserimento nella base di dati può ora essere eseguito come segue:

```
start transaction;
set constraints all deferred; -- Posponi la verifica dei vincoli
insert into R(X,Y) values (1,10); -- Il vincolo di chiave esterna è violato
insert into S(W,Z) values (10,1); -- La consistenza è ripristinata
commit; -- Verifica che tutti i vincoli sono rispettati e registra i dati
```

In un caso come quello appena illustrato è tuttavia preferibile, laddove possibile, riprogettare la base di dati in modo da eliminare i riferimenti circolari.

4 Anomalie delle transazioni concorrenti

Una gestione non controllata della concorrenza può causare il verificarsi di diversi tipi di anomalie. Consideriamo i seguenti problemi:

Perdita d'aggiornamento: accade quando due transazioni concorrenti sono eseguite in modo tale che il valore di qualche dato viene modificato in modo non corretto. Uno schedule con perdita d'aggiornamento è, ad esempio, il seguente:

$$r_1(X) \ r_2(X) \ w_1(X) \ w_2(X)$$

Si supponga che la semantica di T_1 e T_2 sia "incrementare il valore di X di uno". Entrambe le transazioni T_1 e T_2 leggono lo stesso valore di X , diciamo n ; dopodiché T_1 scrive il valore $n + 1$ in X e subito dopo T_2 scrive anche essa $n + 1$ in X , cosicché l'effetto netto è che X è stata incrementata di uno e non di due, come sarebbe accaduto in una qualunque esecuzione seriale di T_1 e T_2 .

Dipendenza da transazioni non committed: accade quando una transazione legge dati scritti da una transazione concorrente attiva. Se quest'ultima transazione fallisce, i dati letti dalla prima non sono più validi. Un esempio di tale **lettura sporca** è il seguente:

$$w_1(X) \ r_2(X) \ abort_1$$

dove $abort_1$ denota l'evento di rollback della transazione T_1 . Le anomalie in tal caso derivano dal fatto che una transazione fa dipendere le proprie operazioni da transazioni di cui non è ancora noto l'esito. Nell'esempio qui sopra, T_2 legge un valore scritto da T_1 , ma T_1 successivamente fallisce, cosicché il valore letto da T_2 non è più valido.

Analisi inconsistente (aggiornamento fantasma e lettura inconsistente): si consideri una transazione che calcola il valore di una funzione aggregata. Durante il calcolo un'altra transazione modifica alcuni dei record coinvolti nel calcolo, cosicché il valore finale calcolato dalla funzione non è corretto. Tale anomalia prende il nome di **aggiornamento fantasma**. Si consideri, ad esempio, una transazione che calcoli la somma dei saldi di alcuni conti correnti mentre, concorrentemente, un'altra transazione trasferisce una somma da un conto a un altro. Se $X = 40$, $Y = 50$ e $Z = 30$ sono tre saldi, le operazioni potrebbero essere eseguite in questo modo:

T_1	T_2
$(somma \leftarrow 0)$	
$r_1(X)$	
$(somma \leftarrow 40)$	
$r_1(Y)$	
$(somma \leftarrow 90)$	
	$r_2(Z)$
	$(Z \leftarrow Z - 10)$
	$w_2(Z)$
	$r_2(X)$
	$(X \leftarrow X + 10)$
	$w_2(X)$
	$commit_2$
$r_1(Z)$	
$(somma \leftarrow 110)$	
$commit_1$	

La somma dei tre saldi è 120, ma la transazione T_1 calcola il valore 110 a causa dell'aggiornamento operato concorrentemente da T_2 . Si noti che, in questo caso, non vi è alcuna dipendenza da transazioni non committed, perché T_1 legge da T_2 solo dopo che T_2 va a buon fine.

Una **lettura inconsistente** (o **non ripetibile**) può avvenire quando una transazione legge due volte lo stesso dato X in momenti successivi e, tra le due letture, un'altra transazione (che va a buon fine) modifica X , cosicché la prima transazione legge due valori diversi per lo stesso dato X .

Si noti che in un sistema in cui le letture inconsistenti sono impediti anche il problema dell'aggiornamento fantasma è evitato.

Osservazione 4.1 *Gli schedule serializzabili non sono soggetti a perdite d'aggiornamento, perché in un'esecuzione seriale di due transazioni una delle due deve necessariamente leggere il valore scritto dall'altra, e dunque uno schedule con perdita d'aggiornamento, in cui è possibile che due transazioni leggano lo stesso valore, non può essere serializzabile. In particolare, se si adotta il protocollo 2PL, lo schedule:*

$$r_1(X) \ r_2(X) \ w_1(X) \ w_2(X)$$

dà luogo a uno stallo. Se si usa un protocollo basato sui timestamp (con o senza multi-versioni), la transazione T_1 è annullata prima di eseguire $w_1(X)$, perché a quel punto $TS(1) < RTM(X) = TS(2)$.²

²La notazione $TS(i)$ denota il *timestamp* della transazione i . Per un oggetto X , $RTM(X)$ è il timestamp più grande tra i timestamp delle transazioni che hanno letto X .

Osservazione 4.2 *Gli schedule serializzabili non hanno problemi di dipendenze da transazioni non committed. Poiché in un'esecuzione seriale delle transazioni ciascuna transazione legge solo da transazioni committed, si ha che uno schedule con dipendenze da transazioni che falliscono non è serializzabile. In particolare, si può dimostrare che gli schedule generati in accordo al protocollo 2PL stretto sono immuni dal problema delle letture sporche.*³

Osservazione 4.3 *Gli schedule serializzabili non portano ad analisi inconsistenti. Nell'esempio dei saldi di conto corrente di pag. 4, l'uso del protocollo 2PL porta a uno stallo. Il metodo basato sui timestamp causa l'annullamento di T_1 al momento di eseguire $r_2(Z)$, perché $TS(2) > WTM(Z)$.⁴ Se si adotta il protocollo MVCC (Multi-Version Concurrency Control, o metodo dei timestamp con multi-versioni), T_1 calcola il valore corretto della somma, perché "vede" la versione di Z precedente agli aggiornamenti operati da T_2 . In nessuno dei tre casi viene calcolato un risultato sbagliato.*

I problemi fin qui presentati riguardano tutti situazioni in cui le transazioni manipolano dati già presenti nella base di dati. Nella pratica, tuttavia, alcune anomalie possono presentarsi a causa di interrogazioni che sono eseguite concorrentemente a operazioni d'inserimento:⁵

Inserimenti fantasma: una transazione reperisce l'insieme di record che soddisfa una data condizione C ed esegue un'operazione O (ad esempio, il calcolo di una funzione aggregata) su tale insieme; successivamente, esegue di nuovo la stessa operazione, ma quando richiede l'insieme di record che soddisfa C , tale insieme risulta essere diverso dal precedente a causa di un'operazione di inserimento che nel frattempo è stata eseguita da un'altra transazione. Allora, il risultato dell'operazione O nei due casi può essere diverso.

Esempio 4.4 *Si consideri una tabella contenente i saldi di alcuni conti correnti. Una transazione T_1 legge i saldi e ne calcola la somma, che risulta essere pari a 120. A seguire, T_2 aggiunge un ulteriore saldo uguale a 200 ed effettua una commit. Ora, T_1 ricalcola la somma e determina che il risultato è diventato 320. Si noti che lo schedule precedente obbedisce al protocollo 2PL stretto (in particolare, non ci sono conflitti lettura-scrittura o scrittura-lettura). Per poter evitare tale anomalia, perciò, è necessario aggiungere ulteriori vincoli al protocollo 2PL stretto.*

Il problema dei fantasmi deriva dal fatto che il protocollo si limita a bloccare le risorse esistenti, mentre bisognerebbe in qualche modo bloccare ciò che il *predicato* della condizione di selezione dei conti correnti specifica. Ad esempio, se i tre conti correnti sono selezionati da T_1 con l'interrogazione "ottieni i saldi dei conti correnti del cliente 'Silvio'", allora, fintantoché T_1 è attiva, a nessun'altra transazione dev'essere consentito inserire nuovi conti correnti per il cliente 'Silvio'. Per evitare gli inserimenti fantasma, è pertanto necessario che i lock possano essere definiti anche con riferimento a condizioni di selezione, impedendo non solo l'accesso ai dati coinvolti, ma anche la scrittura di nuovi dati che soddisfano un certo predicato. Tali lock prendono il nome di **lock di predicato** e la loro descrizione dettagliata esula dagli scopi delle presenti note.

5 Livelli d'isolamento nello standard SQL

Lo standard SQL specifica quattro **livelli d'isolamento**, che definiscono diversi "gradi d'interferenza" tra transazioni sulla base di quali, tra le seguenti tre anomalie, devono essere proibite: letture sporche, letture inconsistenti e fantasmi. L'idea è che in certe applicazioni è accettabile rinunciare alla serializzabilità e preferire un aumento del grado di concorrenza del sistema. In tal caso, tipicamente i DBMS consentono all'utente di specificare i lock in modo esplicito nei casi in cui il sistema non è in grado di garantire una corretta esecuzione concorrente delle transazioni. I quattro livelli d'isolamento e i corrispondenti requisiti sono riassunti nella seguente tabella:

³Si noti, tuttavia, che un problema come quello della lettura sporca non può essere trattato in un formalismo in cui le nozioni di serializzabilità sono riferite a commit-proiezioni, ossia a schedule che contengono soltanto transazioni committed.

⁴La notazione $WTM(Z)$ denota il timestamp della transazione che ha eseguito la scrittura più recente di Z .

⁵Si tenga presente che un inserimento fantasma è un problema diverso dall'*aggiornamento fantasma* precedentemente discusso.

Livello d'isolamento	Lecture sporche	Lecture inconsistenti	Fantasma
Read Uncommitted	Possibile	Possibile	Possibile
Read Committed	Non possibile	Possibile	Possibile
Repeatable Read	Non possibile	Non possibile	Possibile
Serializable	Non possibile	Non possibile	Non possibile

Il livello *Serializable* garantisce la serializzabilità. Può essere implementato mediante protocollo 2PL stretto o MVCC con l'aggiunta dei lock di predicato. Negli altri livelli d'isolamento, la serializzabilità può essere violata nei modi specificati: se il livello è *Repeatable Read* allora possono verificarsi inserimenti fantasma; se il livello è *Read Committed* allora sono possibili anche letture inconsistenti; infine, se il livello è *Read Uncommitted* nemmeno le letture sporche sono proibite (a questo livello le transazioni devono essere *read-only* per garantire la ripristinabilità). Le uniche anomalie che non si verificano nel livello *Read Uncommitted* (e dunque nemmeno nei livelli superiori) sono le perdite d'aggiornamento.

6 Esempi di transazioni in PostgreSQL

È interessante valutare il comportamento di sistemi reali rispetto alla gestione della concorrenza. Nel seguito si descrive il risultato dell'esecuzione concorrente di transazioni in PostgreSQL 9.1 o superiore.⁶ I livelli d'isolamento delle transazioni sono descritti nel §13.2 (**Transaction Isolation**) del manuale di PostgreSQL. Poiché il codice SQL usato è standard, tali esempi possono essere provati, probabilmente senza variazioni, anche con altri DBMS.

PostgreSQL adotta una tecnica chiamata “*snapshot isolation*”,⁷ che è implementata mediante MVCC e i cui dettagli esulano dalle presenti note. PostgreSQL implementa anche il locking dei predicati, che consente di ottenere la serializzabilità. La caratteristica principale dell'approccio di PostgreSQL è che le letture non bloccano mai le scritture e le scritture non bloccano mai le letture. Solo scritture concorrenti possono causare l'annullamento di transazioni.

Sebbene il livello d'isolamento predefinito previsto dallo standard sia *Serializable*, il livello d'isolamento di default in PostgreSQL è *Read Committed*. Il livello d'isolamento può essere modificato con l'istruzione SQL standard:⁸

```
set transaction isolation level
{serializable|repeatable read|read committed|read uncommitted};
```

PostgreSQL in realtà implementa solo tre dei quattro livelli d'isolamento: se si richiede il livello *Read Uncommitted* si ottiene un comportamento più restrittivo equivalente a quello del livello *Read Committed*.⁹ PostgreSQL offre inoltre la possibilità di specificare lock in modo esplicito per gestire le situazioni più critiche nei livelli d'isolamento inferiori a *Serializable*.

Osservazione 6.1 *Si presti attenzione al fatto che il comando `set transaction` dev'essere eseguito dopo `start transaction`, e ha effetto solo per la transazione corrente. Si consulti il manuale del sistema per ulteriori opzioni.*

Negli esempi seguenti, assumeremo che le transazioni siano eseguite sulla seguente base di dati:

```
create table account (name varchar(5) primary key, balance int not null);
insert into account(name, balance)
values ('Xeno', 40), ('Yuri', 50), ('Zoe', '30');
```

Il modo più semplice per provare gli esempi è aprire due sessioni di `psql` in due diverse finestre di terminale.

Esempio 6.2 (Conflitti scrittura-scrittura) *Si consideri la seguente coppia di transazioni che tentano di inserire record con lo stesso valore della chiave:*

⁶Versioni precedenti possono esibire comportamenti diversi da quelli riportati in queste note.

⁷Michael J. Cahill, Uwe Röhm e Alan D. Fekete, *Serializable isolation for snapshot databases*, 2008.

⁸Il livello di default può anche essere impostato nel file `postgresql.conf`.

⁹Ciò è comunque consistente con le specifiche dello standard SQL.

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre> start transaction; insert into account values ('Bud', 90); commit; </pre>	<pre> start transaction; insert into account values ('Bud', 110); commit; </pre>

L'istruzione `insert` di T_2 pone T_2 in attesa. Se T_1 effettua una `commit`, allora T_2 riceve un errore di chiave duplicata; se invece T_1 esegue una `rollback`, allora T_2 va a buon fine. Tale comportamento è (fortunatamente!) indipendente dal livello di isolamento.

Esempio 6.3 (Lettura inconsistente) Si consideri la seguente esecuzione concorrente (in cui per chiarezza è stato impostato in modo esplicito il livello di isolamento anche se coincide con il livello di default di PostgreSQL):

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre> start transaction read only; set transaction isolation level read committed; select * from account where name = 'Zoe'; select * from account where name = 'Zoe'; commit; </pre>	<pre> start transaction; set transaction isolation level read committed; update account set balance = 90 where name = 'Zoe'; commit; </pre>

La transazione T_1 in questo caso ottiene due valori diversi del record: la prima `select` produce il risultato di sinistra, mentre la seconda produce il risultato di destra:

<pre> name balance -----+----- Zoe 50 </pre>	<pre> name balance -----+----- Zoe 90 </pre>
--	--

La lettura diventa consistente se si imposta un livello d'isolamento più alto (`repeatable read` o `serializable`).

Esempio 6.4 (Aggiornamento fantasma) L'esempio dei saldi di conto corrente di pag. 4 può essere implementato come segue (per semplicità, non calcoliamo le somme in modo esplicito):

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre> start transaction read only; set transaction isolation level read committed; select balance from account where name = 'Xeno'; -- 40 select balance from account where name = 'Yuri'; -- 50 select balance from account where name = 'Zoe'; -- 20 commit; </pre>	<pre> start transaction; set transaction isolation level read committed; update account set balance = balance - 10 where name = 'Zoe'; update account set balance = balance + 10 where name = 'Xeno'; commit; </pre>

La transazione T_1 reperisce i valori 40, 50 e 20, la cui somma è 110 (mentre la somma dei saldi è 120). Se il livello d'isolamento è impostato a **repeatable read** o **serializable**, l'ultima **select** di T_1 restituisce, correttamente, il valore 30.

Esempio 6.5 (Inserimento fantasma) Verifichiamo il comportamento di PostgreSQL rispetto al problema degli inserimenti fantasma. Consideriamo le seguenti due sessioni concorrenti:

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre> start transaction read only; select sum(balance) from account; select sum(balance) from account; select sum(balance) from account; commit; </pre>	<pre> start transaction; insert into account values ('Joe', 60); commit; </pre>

Poiché Read Committed è il livello d'isolamento di default di PostgreSQL, due **select** successive all'interno di una transazione possono leggere dati diversi, se tra le due istruzioni un'altra transazione effettua cambiamenti e va a buon fine, come accade nell'esempio considerato tra la prima e la terza **select** di T_1 . Infatti, la terza **select** produce il valore 180 (invece la prima e la seconda producono il valore 120). Se si eseguono le transazioni nel livello Serializable, allora T_1 vede sempre consistentemente lo stesso valore. Il livello interessante in questo caso è Repeatable Read, che è più restrittivo rispetto a quanto richiesto dallo standard: gli inserimenti fantasma non sono possibili nemmeno in questo livello (il comportamento rispetto a questo esempio è dunque identico a quello del livello Serializable).

Esempio 6.6 Dall'esempio 6.5 si può ricavare l'errata conclusione che, poiché il livello Repeatable Read di PostgreSQL è immune dagli inserimenti fantasma, tale livello sia equivalente a Serializable. Il seguente esempio, tratto dal §13.2.3 (*Serializable Isolation Level*) del manuale di

PostgreSQL 9.1, mostra che non è così (e che la classificazione delle anomalie proposta dallo standard SQL forse non è completamente adeguata). Si consideri la seguente tabella:¹⁰

```
create table mytab (class int, value int);
insert into mytab(class, value) values (1,10), (1,20), (2,100), (2,200);
```

Per $i = 1, 2$, sia T_i la transazione che calcola la somma dei valori dei record di classe i e inserisce il risultato in un nuovo record di classe $1 + (i \bmod 2)$. Allora, T_1 e T_2 potrebbero essere eseguite come segue:

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre>start transaction; set transaction isolation level repeatable read; select sum(value) from mytab where class = 1; -- 30 insert into mytab values (2,30);</pre>	<pre>start transaction; set transaction isolation level repeatable read; select sum(value) from mytab where class = 2; -- 300 insert into mytab values (1,300); commit;</pre>
<pre>commit;</pre>	

In PostgreSQL tale schedule ha successo nel livello Repeatable Read e produce la tabella:

class	value
1	10
1	20
2	100
2	200
2	30
1	300

Si noti che non si riscontra nessuna delle anomalie della Sezione 5. Tuttavia, nessuna esecuzione seriale di T_1 e T_2 avrebbe potuto produrre tale risultato. Un'esecuzione seriale di T_1 e T_2 produce una delle due seguenti tabelle:

class	value	class	value
1	10	1	10
1	20	1	20
2	100	2	100
2	200	2	200
2	30	1	300
1	330	2	330

Nel livello Serializable viene correttamente rilevato che lo schedule non è serializzabile:

```
ERROR: could not serialize access due to read/write dependencies among
transactions
```

¹⁰Per mantenere l'esempio il più semplice possibile, la chiave primaria non è specificata. È chiaro che in una base di dati reale ciò non sarebbe accettabile.

Un altro esempio con effetti analoghi, che è lasciato per esercizio analizzare, è il seguente:

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre>start transaction; update account set balance = 30 where balance = 50; commit;</pre>	<pre>start transaction; update account set balance = 50 where balance = 30; commit;</pre>

Per ulteriori esempi, si veda <https://wiki.postgresql.org/wiki/SSI>.

Esempio 6.7 (Aggiornamento concorrente) Si consideri la seguente esecuzione concorrente di due transazioni:

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre>start transaction; set transaction isolation level repeatable read; update account set balance = balance - 10 where name = 'Zoe'; commit;</pre>	<pre>start transaction; set transaction isolation level repeatable read; update account set balance = balance + 10 where name = 'Zoe'; commit;</pre>

L'aggiornamento concorrente di un record è bloccato da PostgreSQL nel livello Repeatable Read e nel livello Serializable, e risulta nel seguente messaggio d'errore:

```
ERROR: could not serialize access due to concurrent update
```

Sebbene tale risultato possa sorprendere (il protocollo 2PL stretto in tale situazione consentirebbe a entrambe le transazioni di effettuare la commit), è consistente con il modello usato da PostgreSQL (si legga il §13.2.2 del manuale). Nel livello Read Committed entrambe le istruzioni **update** hanno successo.

7 Esempi di transazioni in MySQL

Nota: per installare MySQL nella macchina virtuale del corso, usare il comando

```
sudo apt-get install mysql-server
```

Il collegamento al DBMS server può poi essere effettuato con il comando:

```
mysql -u root -p
```

specificando la password che è stata inserita in fase d'installazione.

L'implementazione di MySQL è un misto di MVCC e 2PL stretto. Il livello d'isolamento di default in MySQL (con InnoDB) è *Repeatable Read*.

Osservazione 7.1 *MySQL non implementa il locking dei predicati, ma usa un'approssimazione che garantisce comunque la serializzabilità nel livello Serializable.¹¹ MySQL offre inoltre la possibilità di specificare lock in modo esplicito per gestire le situazioni più critiche nei livelli d'isolamento inferiori a Serializable.*

La base di dati degli esempi precedenti può essere implementata in MySQL come segue:

```
create table account (name varchar(5) primary key, balance int not null)
engine="InnoDB";
insert into account(name, balance)
values ('Xeno', 40), ('Yuri', 50), ('Zoe', '30');
```

Osservazione 7.2 *MySQL può usare diverse strutture di memorizzazione dei dati e quella di default fino alla versione 5.1 (ISAM) non è transazionale. Gli esempi che seguono assumono che il sistema faccia uso di **InnoDB**, che offre il supporto per la gestione di transazioni concorrenti.*

Osservazione 7.3 *Si presti attenzione al fatto che in MySQL il comando **set transaction** dev'essere dato prima di **start transaction**, e ha effetto solo per la transazione successiva. Si consulti il manuale del sistema per ulteriori opzioni.*

Il modo più semplice per provare gli esempi precedentemente discussi è aprire due sessioni del client da linea di comando `mysql` in due diverse finestre di terminale. I risultati che si ottengono con MySQL 5.1 o superiore sono analoghi a quelli di PostgreSQL, tranne nel caso dell'Esempio 6.7, in cui le transazioni sono entrambe portate a termine con successo nel livello *Serializable*, in accordo al protocollo 2PL stretto.

Poiché in MySQL i livelli *Read Uncommitted* e *Read Committed* sono distinti, è possibile sperimentare letture sporche, come illustra l'esempio seguente.

Esempio 7.4 *Si consideri la seguente esecuzione concorrente:*

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre>start transaction; update account set balance = 42 where name = 'Zoe';</pre>	<pre>set transaction isolation level read uncommitted; start transaction read only; select balance from account where name = 'Zoe'; -- 42 commit;</pre>
<pre>rollback;</pre>	

La transazione T_2 legge il valore aggiornato da T_1 , che successivamente esegue una *rollback*. Quest'anomalia non si presenta nei livelli superiori a *Read Uncommitted*.

¹¹Intuitivamente, l'implementazione di MySQL è un'approssimazione del lock dei predicati nel senso che in alcuni casi blocca più risorse di quanto strettamente necessario.

Esercizi

Esercizio: Si consideri la seguente esecuzione concorrente di due transazioni:

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre>start transaction; set transaction isolation level serializable; select * from account where name = 'Zoe'; update account set balance = balance - 10 where name = 'Zoe'; commit;</pre>	<pre>start transaction; set transaction isolation level serializable; update account set balance = balance + 10 where name = 'Zoe'; commit;</pre>

Qual è il risultato in PostgreSQL? Qual è il risultato in MySQL? Il risultato di MySQL è consistente con le regole del protocollo 2PL stretto? In che modo differisce dall'Esempio 6.7?

Esercizio (conflitto scrittura-lettura): Si consideri la seguente esecuzione concorrente:

Sessione 1 (T_1)	Sessione 2 (T_2)
<pre>start transaction; update account set balance = 0 where name = 'Zoe'; commit;</pre>	<pre>start transaction read only; select * from account where name = 'Zoe'; commit;</pre>

Si spieghino i risultati e le eventuali differenze che si ottengono in PostgreSQL e in MySQL nei vari livelli d'isolamento.

Esercizio: Si consideri un sistema di *bug tracking*, la cui base di dati contiene le seguenti definizioni:¹²

```
create domain dom_pri as varchar(6)
  check (value = 'alta' or value = 'bassa');

create domain dom_stato as varchar(10)
  check (value = 'attivo' or value = 'in ferie');

create table Sviluppatore (
  nome varchar(50) primary key,
  stato dom_stato not null
);
```

¹²Questo esercizio è ispirato a [un blog post](#).

```

create table Ticket (
  descrizione varchar(200) primary key,
  priorità dom_pri not null,
  responsabile varchar(50) not null references Sviluppatore
);

```

Un'azienda di software che fa uso di tale sistema ha stabilito la seguente regola aziendale: “i ticket ad alta priorità non devono essere mai assegnati a personale in ferie”. Per forzare il rispetto di tale regola, quando uno sviluppatore *S* chiede di andare in ferie il suo stato è aggiornato con l'istruzione:

```

update Sviluppatore set stato = 'in ferie'
  where nome = 'S'
  and not exists (
    select * from Ticket
    where responsabile = Sviluppatore.nome
    and priorità = 'alta'
  );

```

In altre parole, lo stato può essere modificato solo se lo sviluppatore non è responsabile di alcun ticket ad alta priorità. Similmente, la priorità di un ticket *T* può essere aumentata solo se il responsabile del ticket non è in ferie:

```

update Ticket set priorità = 'alta'
  where descrizione = 'T'
  and 'in ferie' <> (select stato from Sviluppatore
    where nome = Ticket.responsabile);

```

Si determini il minimo livello d'isolamento di PostgreSQL necessario a garantire il rispetto della regola aziendale, assumendo che inizialmente la base di dati sia in uno stato consistente.