

Transazioni / 16-05

Spesso bisogna gestire più blocchi sui inseriti allo stesso
logico applicativo.

Ad es. in un trasferimento di denaro, tolgo da un conto
(una tuple) e li aggiunge in un altro.

Cosa succede se il sistema esegue solo la rimozione?

Ho perso soldi.

Come ottengo garanzia? Con la transazione

- unità logica di elaborazione lettura / scrittura
- azione su tuple con una serie di operazioni fisiche elementari sul DB
- garantisce buone proprietà

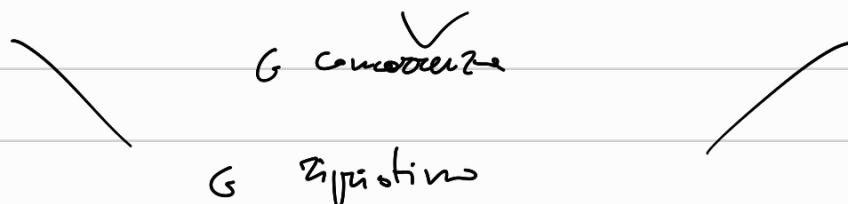
Ogni op. di base cambia lo stato del DB.

L'esecuzione delle transazioni è mediata dal Transaction

Manager (costituito dal gestore della concorrenza e ripristino)

Vengono assicurate delle proprietà: ACID

Atomicità, Causalità, Isolamento, Durabilità (persistenza)



- Attenzione: se l'esecuzione va a buon fine e allora lo stato
del DB viene modificato, altrimenti, anche se un solo
questo, non cambia nulla nel DB (abort)
 - viene eseguita con **rollback** che ritorna
ell'istante del DB precedente alle operazioni

- consistenza: particolare da cui stato del DB consistente
 (= soddisfa i vincoli di integrità) origina in un stato consistente
 (es. l'importo prelevato è lo stesso che viene versato)
 → somma dei soldi è costante. Se così non fosse × dimenticato, giochi sulla borsa
- isolamento: poiché esistono un accesso concorrente (da più applicazioni) sul DB, si garantisce da ogni transazione, se pure in modo, e' vista come unica nell DB, ovvero una transazione non potra' leggere risultati intermedi di altre opp.
- durabilità: i riavvisti, in caso l'app venga interrotta, deve rimanere permanente anche in caso di guasti futuri

Tipi di transazioni:

tipo semplice, transazioni brevi, vengono dette **FAT**
 e' tale:
 → A comandi SQL (implicitamente)
 → esplicitamente con BEGIN [transazione] → START transaction

Due esiti:

- terminazione corretta
- terminazione non corretta

Termino correttamente se dopo lo suo esecuzione
viene eseguito infine commit [work], ovvero comunica
"ufficialmente" la terminazione delle transazioni



Se notiamo degli errori durante l'operazione viene effettuata una terminazione anticipata con ROLLBACK [work] (esplicito) (il DB fa un Undo)

→ La terminazione anticipata è implicita in caso di guasti / violazioni, e vice fatto un rollback estremistico con Undo ritornando allo stato esistente prima delle transazioni

Per quanto dàth sulla consistenza, lo stato iniziale e finale delle transazioni deve sempre soddisfare i vincoli

→ anche quelle intermedie devono soddisfare? Dipende, in generale possono non essere consistenti, ma possono farlo il controllo a scende dalla necessità

(per lo p.m. di isolamento solo le trans. nelle quali gli sloti intermedi e nessun altro, x questo possono essere inconsistenti)

Ma due modelli di controllo x decidere quando verificare i vincoli:

- valutazione immediata: effettuata dopo ogni comando
(controlli intermedi, come detto sopra)

- valutazione diffusa: effettuata sullo stato finale (al termine delle transazioni)

In caso di violazione, in entrambi le modalitè, la trans. viene abortita.

Con le clausole NOT DEFERRABLE dopo qualsiasi vincolo, posso forzare la validazione immediata, senza not e' differibile ma potenzialmente posso decidere:

INITIALLY IMMEDIATE, inizialmente verificato in modo immediato, ma comunque differibile

se all'interno della transazione

SET CONSTRAINT ALL DEFERRABLE automaticamente viene eseguita una validazione differibile

Allora di default e' immediato ma potenzialmente differibile una ogni transazione puo' decidere per se'

In generale le transazioni iniziano al primo comando SQL e terminano implicitamente (esclusivamente) con AUTO COMMIT

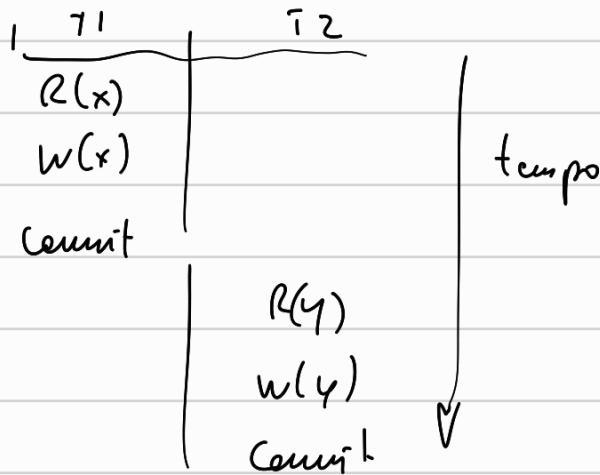
→ se TRUE: ogni comando costituisce una transazione e se falso (COMMIT ed ogni comando SQL) controllo intervento,

→ se FALSE: tutti i comandi in un'unica transazione (COMMIT alla fine)

Controllo della concorrenza

Si eseguono più transazioni eseguite insieme, in sequenza

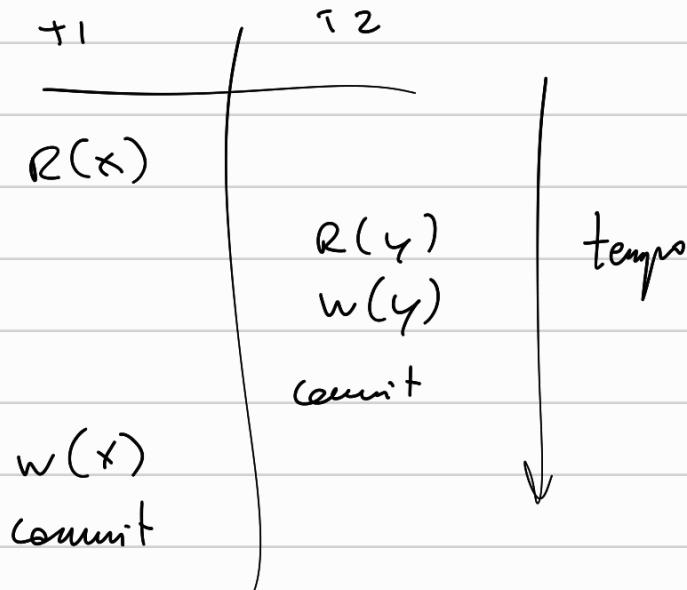
→ serial execution



schedule

ci prende uno schema
-tempo-

• esecuzione concorrente → interleaved execution



mentre viene eseguito T1
e ho un op. di T2 e quindi:
T1 ottiene, T2 pro' sfruttare
nel frattempo lo CPU,
conseguendo il n° di op/tempo
(throughput), in particolare
se T2 è più breve di T1

- T1 è "lunga", T2 è "breve", 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

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

T2 richiede a time = 2 di iniziare

il sistema vede un ergo x lotto (relativa a una linea)

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 =
 $(1004 + 3)/2 = 503.5$

interleaved

→ migliora le prestazioni del sistema

→ ha già ricevuto risposta di T2.

OBIETTIVO è ottimizzare il tempo medio.

Le operazioni J/O delle transazioni coinvolte sono interate alle solo letture e scritture su disco (no modifiche in memoria a catena)

Il controllo di correttezza garantisce che le transazioni in correttezza non interferiscono fra loro (isolamento). Se l'isolamento non viene garantito possono avere le anomalie:

• **lost update**: si compra l'ultimo biglietto da parte di 2 clienti di 2 agenzie diverse.

→ se non isolati entrambi comprano lo stesso biglietto

• **dirty read**: viene fornita una data del concerto, se comprato il biglietto mi viene detto che la data non è fissata

→ se durante l'inserimento viene chiesto la lettura e non isolato

poco dopo viene fatto un rollback dell'insertion
(l'acquisto del biglietto per lo stesso letto non più
presente nell'elenco)

→ la lettura è "sparsa" perché viene letto uno stato non stabile,
intermedio che non è detto sia persistente allo fine

- **unrepeatable read**: il prezzo del biglietto è 40€ e dopo
5 minuti, l'acquisto costa 50€
(leggo due volte, nelle stesse transazioni)

→ se non isolato, per il modello proposto (a transazione legge
uno stesso valore due volte ma deve assumere un
valore diverso, se fra le due letture ha un aggiornamento
da un'altra transazione)

- **phantom row**: voglio comprare i biglietti × le ^{nuove} 2 tuple disponibili,
ma l'acquisto ora ve ne sono 3

→ se non isolato, similmente al precedente → si applica alle tuple
(più di una), qui viene aggiunta una tuple dopo la lettura
dalla prima transazione. Alla seconda lettura viene visualizzata
una riga (phantom row) in più.

- **Scadute concorrenti** ↗
 → più giovani concorrenti
 → migliori prestazioni

• schedule seriale / no anomolie (isolamento totale)
pecciori prestazioni

Si adotta un compromesso: **schedule semi-distribuite**

→ schedule non seriale (garanzie prestazionali)

→ non cause anomalie (garanzie di correttezza)

Per ottenere tali schedule devono soddisfare le proprie per cui produce lo stesso risultato / stato di uno schedule seriale.

Cioè se usa interleaving

↳ a seconda dell'ord delle
trans eseguite ne ha più d'una

Protocolli di locking

Si utilizzano dei **lock** per creare tali schedule, utilizzato anche dai SO.
Per eseguire un'operazione c'è prima necessario "acquisire" un lock
sulla risorsa interessata (es una -tuple)
Le richieste è implicita, non visibile al livello SEL

Sono di vario tipo, ad es:

- S-lock : condiviso, usato x leggere (più per risorse)

- X-lock : esclusivo, usato x scrivere/modificare (unico x risorsa)

Il lock manager è una componente del diversoritier manager
e utilizza come motivo di compatibilità:

τ_2 ha un lock di tipo

	S	X
{	OK	NO
}	NO	NO

es. se τ_2 ha lock esclusivo, tu non puoi leggere

Tuttavia
un lock di tipo

ha conflitto se qualcuno ha lock di scrittura / modifica sullo stesso dato

Al termine di utilizzo delle risorse viene rilasciato il lock

I protocolli specificano le strategie di acquisizione e rilascio del lock

→ il protocollo che garantisce al meglio è il 2-phase locking (STRONG)

nella prima fase si acquisiscono tutti i lock necessari vengono rilasciati alla fine della transazione (COMMIT/ROLLBACK)

→ non si possono generare anomalie (isolamento tot. della transazione)

→ garantisce serializzabilità

effetti collaterali: deadlock, situazioni di stallo in cui una transazione è in attesa di un lock da un'altro che attende a sua volta

→ risolte solo con (abortimento di una delle due transazioni, dopo un po' di tempo)

Se viene chiesto un lock esclusivo ma ne sono già di condizioni, attende perché ha conflitto (non termina).

unlock (X) rilascia tutti i lock (S e X) presenti su X

Gestione delle anomalie

ASSENZA DI LOST UPDATE

T1	X	T2
R(X)	1	
X=X+1	1	
	1	
W(X)	0	
Commit	0	
	0	
	W(X)	
	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	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 abortire, ad esempio, T2, allora T1 può proseguire

ASSENZA DI DIRTY READ

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

ASSENZA DI UNREPEATABLE 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

Per le phantom row si risolve con vari meccanismi,
il più semplice: il lock non viene chiesto sulle singole tuple,
ma sull'intera tabella e sui suoi indici
(bloccando quindi le op di modifica)

2-PHASE-LOCKING

CONTRO:

- garantisce serializzabilità // isolamento
- assenza di anomalie // let.
- sisteme rigido (molti wait)
- molte op in conflitto da bloccare

Compromesso: permette di far presentare solo alcune anomalie
(priorità al lost update, poche informazioni!)

isolamento totale → parziale

il livello ol. isolamento viene scelto dalla transazione

Un livello di isolamento corrisp. a un protocollo di lock me
lo quale di acquisire / rilasciare lock con regole diverse
di 2 - ORDER - LOCKING STANDARDS. Vi sono 4 livelli nell. standard,
fra cui l'utens. puo scegliere:
"LIVELLO SERIALIZZABILE"

- Lo standard definisce 4 livelli di isolamento

Livello di isolamento	Lost update	Dirty read	Unrepeatable read	Phantom problem
READ UNCOMMITTED	NO	YES	YES	YES
READ COMMITTED	NO	NO	YES	YES
REPEATABLE READ	NO	NO	NO	YES
SERIALIZABLE	NO	NO	NO	NO

- Nessun livello puo generare anomalie di lost update

2

- lock read committed villoso: lock condivisi oppure possibile, mentre gli esclusivi vengono salvaguardati fino alla terminazione della trans.
- lock phantom row come sopra ma i condivisi: alla terminazione
- lock read committed come sopra ma i condivisi non vengono ridistesi (ma x rilassamento, rischio repeated read)

il livello di isolamento viene scelto dopo la definizione della clausione
 → serializable è quello da impiegare più tempo
 → read uncommitted .. meno tempo
 read committed è giusto compromesso onde se posso perdere info
 in lettura

I sistemi utilizzano un meccanismo di implementazione detto **snapshot isolation** (concorrenza multiversione) (utilizzato da PostgreSQL) (lo standard SQL non dice come implementare i livelli, che si limita a prohibire)

- risolve la gestione dei lock in lettura
- limita le anomie create dalle read uncommitted
 - ↳ ogni lettura viene fatta su una versione esistente all'inizio della transazione
- le scritture non cambiano approccio
- non garantisce serializzabilità

PostgreSQL implementa

- serializable
- repeatable reads
- read committed = read uncommitted