Transazioni

Antonella Poggi

Dipartimento di informatica e Sistemistica
Università di Roma "La Sapienza"

Progetto di Applicazioni Software
Anno accademico 2008-2009

Questi lucidi sono stati prodotti sulla base del materiale preparato per il corso di progetto di Basi di Dati da A. Calì e D. Lembo.

Transazioni

- Una **transazione** è una unità elementare di lavoro svolta da un programma applicativo su un DBMS.
- Una transazione può comprendere una o più operazioni sui dati.
- L'esecuzione di una applicazione viene vista come una serie di transazioni, intervallate dall'esecuzione di operazioni non relative ai dati.

Transazioni

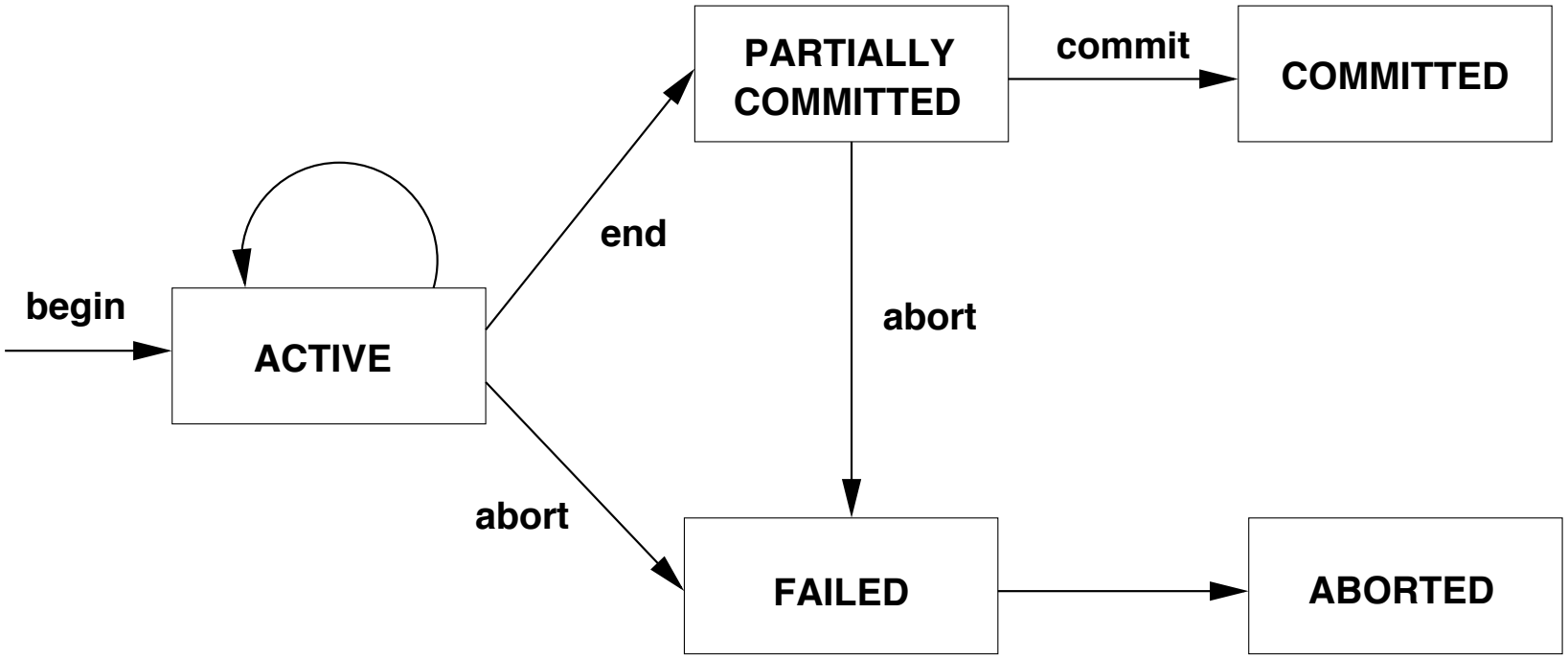
BOT (Begin Of Transaction): inizio della transazione

EOT (End of Transaction): fine della transazione

commit operazione che rende definitivi i cambiamenti

rollback operazione che annulla i cambiamenti eseguiti a partire dall'ultimo commit

Stati di una transazione



Stati di una transazione

ACTIVE, ABORTED, COMMITTED ovvî

PARTIALLY COMMITTED subito dopo l'esecuzione dell'ultimo statement della transazione

FAILED precede immediatamente ABORTED; la transazione va in FAILED quando non si può fare `commit` o quando un `abort` è esplicitamente invocato

SQL e la modalità auto-commit

- I DBMS mettono a disposizione una modalità di **auto-commit**, grazie alla quale ogni comando SQL sulla base dati è implicitamente seguito dal `commit`
- In tale modalità, di fatto, tutte le transazioni sono costituite da un solo comando SQL
- Per transazioni così semplici, utilizzare la modalità autocommit porta vantaggi in termini di praticità della scrittura del codice dell'applicazione e di efficienza
- Ad ogni modo, **nelle applicazioni è spesso necessario impostare transazioni che contengono più di un comando SQL**. In tutti questi casi il `commit` deve essere esplicitamente gestito dal programmatore.

Proprietà "acide" delle transazioni

1. **A**tomicity (atomicità)
2. **C**onsistency (consistenza)
3. **I**solation (isolamento)
4. **D**urability (persistenza)

Atomicità

L'**atomicità** garantisce che le operazioni di una transazione vengano eseguite im modo **atomico** (tutte o nessuna).

- È cruciale per l'integrità dei dati
- Se qualcosa va storto, il sistema deve essere in grado di annullare tutti i cambiamenti fatti a partire dall'inizio della transazione

Consistenza

La **consistenza** assicura che l'esecuzione della transazione porti la base di dati in uno stato *consistente*, i.e. che non violi i vincoli di integrità sulla base di dati.

In accordo con questa propriet'a, la **verifica** della violazione dei vincoli dovrebbe essere fatta **alla fine** della transazione (in modo **differito**).

Isolamento

L'**isolamento** garantisce che l'esecuzione di una transazione sia **indipendente** dalla esecuzione contemporanea di altre transazioni.

- Le transazioni concorrenti non si influenzano l'una con l'altra.

Persistenza

La **persistenza** garantisce che se la transazione va a buon fine, ovvero dopo che un `commit` è stato eseguito con successo, l'effetto della transazione sia registrato in maniera permanente sulla base di dati.

Schedule seriali

Dato un insieme di transazioni $T_1, T_2, , T_n$, una sequenza S di esecuzioni di azioni di tali transazioni che rispetta l'ordine delle azioni all'interno di una transazione (i.e. tale che se l'azione a occorre prima dell'azione b in T_i, allora a occorre prima di b anche in S) è chiamato *schedule* (su $T_1, T_2, , T_n$)

Uno schedule S è detto *seriale* se le azioni di ogni transazione in S avvengono prima di tutte le azioni di un'altra transazione in S, i.e., se in S le azioni si diverse transazioni non si alternano.

Serializzabilità

Uno schedule S è *serializzabile* se il risultato della sua esecuzione è lo stesso risultato prodotto dall'esecuzione di uno schedule seriale costituito dalle stesse transazioni. Cioè, uno schedule S su $T_1, T_2, , T_n$ è serializable se esiste uno schedule seriale su $T_1, T_2, , T_n$ che è "equivalente" ad S.

Due schedule S_1 e S_2 sono equivalenti se, per ogni stato iniziale, l'esecuzione di S_1 produce lo stesso risultato dell'esecuzione di S_2.

Per esempio, date $T_1(x = x + x; x = x + 2)$ e $T_2(x = x * *2; x = x + 2)$, due possibili schedule seriali su di esse sono:

Sequenza 1: $x = x + x; x = x + 2; x = x * *2; x = x + 2$

Sequenza 2: $x = x * *2; x = x + 2; x = x + x; x = x + 2$

Schedule seriale – Esempio

Tempo	**T1**	**T2**	X	Y
t1		beginTrans	100	100
t2		read(X,t)	100	100
t3		t := t+100	100	100
t4		write(X,t)	200	100
t5		read(Y,v)	200	100
t6		v := v*3	200	100
t7		write(Y,v)	200	300
t8		commit	200	300
t9	beginTrans		200	300
t10	read(X,s)		200	300
t11	s := s-10		200	300
t12	write(X,s)		190	300
t13	commit		190	300

Schedule serializzabile – Esempio

Tempo	T1	T2	X	Y
t1		beginTrans	100	100
t2		read(X,t)	100	100
t3		t := t+100	100	100
t4		write(X,t)	200	100
t5	beginTrans		200	100
t6	read(X,s)		200	100
t7	s := s-10		200	100
t8	write(X,s)		190	100
t9	commit		190	100
t10		read(Y,v)	190	100
t11		v := v*3	190	100
t12		write(Y,v)	190	300
t13		commit	190	300

Conflitti nelle transazioni concorrenti

Due azioni sullo stesso oggetto sono in conflitto se almeno una di queste è una operazione di scrittura.

Due transazioni concorrenti T1 e T2 sono in conflitto se presentano azioni in conflitto sullo stesso oggetto.

Individuiamo tre tipi di conflitti (assumendo che sia T1 a subire il conflitto)

- **scrittura-lettura** o Write-Read (WR): T1 legge un oggetto precedentemente scritto da T2 (che non è ancora conclusa);

Anomalie nelle transazioni concorrenti (cont.)

- **lettura-scrittura** o Read-Write (RW): T2 scrive un oggetto precedentemente letto da T1 (che potrebbe leggere di nuovo questo oggetto);
- **scrittura-scrittura** o Write-Write (WW): T2 scrive un oggetto precedentemente scritto da T1 (per cui sovrascrive il cambiamento di T1).

Anomalie nelle transazioni concorrenti

Le anomalie nelle transazioni concorrenti possono essere caratterizzate come segue:

1. **dirty read** (WR)
2. **unrepeatable read** (RW)
3. **lost update** (WW)
4. **phantom read**
5. **inconsistent analysis**

Dirty Read – Esempio

Tempo	**T1**	**T2**	X
t1		beginTrans	100
t2		read(X,t)	100
t3		t := t+100	100
t4	beginTrans	write(X,t)	200
t5	read(X,s)	···	200
t6	s := s-10	rollback	100
t7	write(X,s)		190
t8	commit		190

Dirty Read

- Si può verificare quando una transazione può leggere le modifiche effettuate da un'altra transazione non ancora completata (committed)
- Di fatto, una transazione legge un dato **intermedio** e non persistente a tutti gli effetti
- Nell'esempio, T1 legge un valore modificato da T2, che poi non ha buon fine (fa `rollback`). La modifica eseguita da T1 è pertanto inconsistente.

Unrepeatable Read – Esempio

Tempo	T1	T2	X
t1	beginTrans		100
t2	read(X,t)	beginTrans	100
t3		read(X,s)	100
t4		s := s+100	100
t5		write (X,s)	200
t6		commit	200
t7	read(X,v)		200
t8	commit		200

Unrepeatable read

- Si può verificare quando una transazione può scrivere su un oggetto letto da un'altra transazione non ancora completata (committed)
- può quindi accadere che:
  - T1 legge due valori diversi per lo stesso dato in momenti diversi (e T2 scrive un oggetto precedentemente letto da T1) – *unrepetable read*

Phantom read – Esempio

Tempo	**T1**	**T2**	X
t1		beginTrans	{A,B,C}
t2	beginTrans	read(X,t)	{A,B,C}
t3	read(X,s)	v:=count(|t|)	{A,B,C}
t4	insert(D,s)		{A,B,C}
t5	write(X,s)		{A,B,C,D}
t6	commit	commit	{A,B,C,D}

Phatom read

- Si può verificare quando quando si modifica un insieme di tuple che è stato letto da un'altra transazione, e.g. quando una transazione aggiunge delle tuple in una tabella che soddisfano la clausola WHERE di uno statement eseguito da un'altra transazione

Lost Update – Esempio

Tempo	**T1**	**T2**	X
t1		beginTrans	100
t2	beginTrans	read(X,t)	100
t3	read(X,s)	t := t+100	100
t4	s := s-10	write (X,t)	200
t5	write(X,s)	commit	90
t6	commit		90

Lost Update

- Si può verificare quando una transazione può scrivere su un oggetto già scritto da un'altra transazione non ancora completata (committed)
- gli effetti della transazione T2 sono annullati, giacché T1 sovrascrive l'aggiornamento effettuato da T2
- Come nei casi precedenti, se T1 e T2 fossero eseguite in modo sequenziale, il risultato sarebbe differente

Inconsistent analysis – Esempio

Si abbiano tre dati X,Y,Z con un vincolo di integrità: X+Y+Z=100.

Tempo	**T1**	**T2**	X	Y	Z	t+s+v
t1	beginTrans		20	30	50	
t2	read(X,t)	beginTrans	20	30	50	
t3	read(Y,s)	read(Y,s')	20	30	50	
t4		s' := s'-10	20	30	50	
t5		read(Z,v')	20	30	50	
t6		v' := v'+10	20	30	50	
t7		write(Y,s')	20	20	50	
t8		write(Z,v')	20	20	60	
t9	read(Z,v)	commit	20	20	60	110
t10	commit		20	20	60	110

Inconsistent analysis – Esempio

- Si può verificare quando una transazione può scrivere su un oggetto legato ad un altro oggetto da un vincolo di integrità e letto da un'altra transazione non ancora completata (committed)
- la transazione T1 osserva **solo in parte** gli effetti di T2, e pertanto il valore s+t+v(=110) calcolato da T1 è inconsistente ($\neq$ 100), sebbene T2 e T1 (considerate da sole) preservino l'integrità – *inconsistent analysis*

Lock

- Per evitare anomalie nelle transazioni concorrenti, i DBMS usano i **lock**, che sono meccanismi che bloccano l'accesso da parte di altre transazioni, ai dati ai quali una transazione accede (o che modifica). I lock possono essere a livello di riga o di tabella.
- In genere i lock sono di due tipi: **condivisi** (possono essere imposti da due diverse transazioni contemporaneamente) o **esclusivi** (se una transazione ha questo tipo di lock su di un oggetto, nessun'altra transazione può avere un lock - di qualsiasi tipo - sull'oggetto in questione).

Protocollo 2 Phase Locking (2PL)

1. Una transazione T che vuole leggere (risp. modificare) un oggetto deve prima richiedere un lock condiviso (risp. esclusivo). T resterà sospesa fino a che il DBMS non sia in grado di garantire il lock richiesto
2. Le richieste di lock da parte delle transazioni concorrenti precedono le operazioni di rilascio dei lock.

Tale protocollo permette solo l'esecuzione di transazioni "sicure" (cioè senza anomalie). Suoi opportuni "rilassamenti" fanno convivere l'esecuzione di transazioni con alcune anomalie, a seconda del **livello di isolamento** specificato.

Transazioni in SQL / 1

L'SQL permette di impostare due caratteristiche di una transazione:

- il **modo di accesso**, che può essere `READ ONLY` oppure `READ WRITE`
  → transazioni con modalità di accesso `READ ONLY` non potranno eseguire `INSERT`, `UPDATE`, `DELETE`, `CREATE`, etc.; come tali, potranno richiedere solo lock condivisi (aumentando la concorrenza)

  ```
  SET TRANSACTION READ ONLY
  ```

Transazioni in SQL / 2

- il **livello di isolamento**, che determina in quale misura la transazione è esposta alla azione di altre transazioni concorrenti

  Impostare un livello di isolamento con SQL

  ```
  SET TRANSACTION ISOLATION LEVEL <livello>
  ```

Livelli di isolamento in SQL / 1

Lo standard SQL-92 definisce 4 livelli di isolamento, basandosi sulla definzione classica di serializzabilità, e su 3 tipi di anomalie: dirty read, non-repeatable read e *phantom*.

Phantom – Esempio

1. La transazione T1 legge un insieme di dati che soddisfano alcune *condizioni di ricerca* `<search condition>`.
2. La transazione T2 crea un insieme di dati che soddisfano le condizioni `<search condition>` e fa commit.
3. T1 ripete lo stesso tipo di lettura effettuato in precedenza, ovvero un insieme di dati che soddisfano `<search condition>` $\rightarrow$ T1 ottiene un insieme di dati diverso rispetto alla prima lettura.

Livelli di isolamento in SQL / 2

La definizione dei livelli di isolamento è tale da ammettere diverse implementazioni (non solo il locking), ed è definita secondo la seguente matrice:

Livello	**Dirty Read**	**Unrepeatable Read**	**Phantom Read**
READ UNCOMMITTED	SI	SI	SI
READ COMMITTED	NO	SI	SI
REPEATABLE READ	NO	NO	SI
SERIALIZABLE	NO	NO	NO

Lo standard specifica inoltre che il livello di isolamento SERIALIZABLE garantisce che NESSUNA anomalia possa occorrere.

Read uncommitted

È il livello di isolamento più basso.

- T può leggere i cambiamenti fatti da una transazione in corso
- T non può effettuare cambiamenti (richiede la modalità `READ ONLY`)

Evita solo le anomalie di tipo lost-update (perchè la modalità è `READ ONLY`)

Read committed

1. Una transazione T leggerà solo i cambiamenti apportati da transazioni concluse con successo.
2. Nessun valore **scritto** da T verrà cambiato da un'altra transazione.

Evita le anomalie dirty read (grazie a 1), lost update (grazie a 2), ma non:

- le unrepeatable read (poichè gli oggetti solo letti da una transazione T potranno essere scritti da altre transazioni non ancora concluse)
- le phantom read
- le inconsistent analysis

Repeatable read

1. Una transazione T leggerà solo i cambiamenti apportati da transazioni concluse con successo.
2. Nessun valore **scritto** da T verrà cambiato da un'altra transazione.
3. Nessun valore **letto** da T verrà cambiato da un'altra transazione.

Evita le anomalie dirty read (grazie a 1), lost update (grazie a 2), le unrepeatable read (grazie a 3), ma non le phantom read e neanche le inconsistent analysis (poichè altre transazioni potranno modificare valori non letti da T ma legati da qualche vincolo a valori letti da T).

Serializable

Quale che sia l'implementazione del controllo di concorrenza adottata nel DBMS, questo livello garantisce che tutte le operazioni delle transazioni siano eseguite come se ogni transazione occorresse una dopo l'altra, in maniera isolata.

Livelli di isolamento e 2PL / 1

SERIALIZABLE la transazione è gestita secondo il 2PL

REPEATABLE READ vengono acquisiti blocchi condivisi per tutti i dati letti da ogni istruzione della transazione e tali blocchi vengono mantenuti attivi fino al completamento della transazione

→ impedisce ad altre transazioni di modificare qualsiasi riga letta dalla transazione corrente

→ altre transazioni possono inserire nuove righe, se tali righe corrispondono alle condizioni di ricerca delle istruzioni eseguite dalla transazione corrente

Livelli di isolamento e 2PL / 2

READ COMMITTED prima di scrivere gli oggetti la transazione ottiene lock esclusivi che conserva fino a quando termina; prima di leggere gli oggetti ottiene lock condivisi che però rilascia immediatamente dopo

READ UNCOMMITTED Non fa alcuna richiesta di lock

Livelli di isolamento supportati da MySQL

1. `READ UNCOMMITTED`
2. `READ COMMITTED`
3. `REPEATABLE READ`
4. `SERIALIZABLE`

Il livello `REPEATABLE READ` è il default.

Leggere ed Impostare il livello di isolamento tramite l'interprete dei comandi MySQL

Impostare un livello di isolamento

```
SET TRANSACTION ISOLATION LEVEL <livello>
```

Leggere il livello di isolamento della sessione corrente

```
mysql> SELECT @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set (0.03 sec)
```

Gestire l'autocommit tramite l'interprete dei comandi MySQL

Impostare lo stato dell'autocommit

```
SET AUTOCOMMIT {1 | 0}
```

Leggere lo stato dell'autocommit

```
mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
|            1 |
+--------------+
1 row in set (0.03 sec)
```

autocommit=1, e quindi attivo, è lo stato di default.