Transazioni, concorrenza e deadlocks in SQL Server

Eliseo Foglietta
Microsoft Certified Trainer
e.foglietta@ns12.it

Microsoft Gold Partner

NS12

Agenda

- Transazioni implicite ed esplicite
- Concorrenza pessimistica
- Fenomeni indesiderati
- Livelli di isolamento
- Deadlocks
- Concorrenza ottimistica
- …

Concetto di transazione

- Definiamo **Transazione** una o più istruzioni T-SQL che sono raggruppate logicamente in un'unica unità di lavoro, che viene eseguita nella sua interezza (**commit**) oppure completamente annullata (**rollback**).

- Per essere una transazione, l'unità di lavoro deve godere di 4 proprietà, denominate proprietà **ACID:**

Atomicità, Consistenza, Isolamento,Durabilità

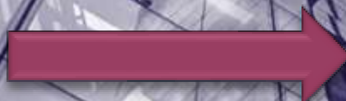
Transazione ACID

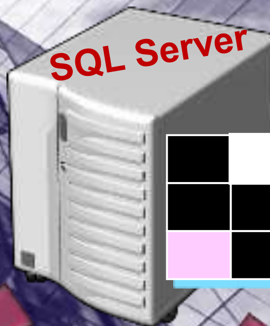
- **Atomicità**
  - Al termine di una transazione, tutte le modifiche effettuate sono applicate oppure nessuna.
- **Consistenza**
  - Dopo il completamento, una transazione deve lasciare tutti i dati e le relative strutture integre nonché conformi alle regole di business
- **Isolamento**
  - Definisce come le modifiche apportate da una transazione debbano essere isolate dalle altre transazioni simultanee
- **Durabilità.**
  - Le modifiche ai dati devono essere mantenute in caso di errore del sistema.
  - SQL Server è durabile grazie al Transaction Log

Ruolo del Transaction Log

1 L'applicazione invia istruzioni di modifica

2 Le pagine coinvolte vengono prima copiate In Buffer Cache e poi modificate lì in memoria





Buffer Cache

3 Le istruzioni di modifica vengono scritte nel Transaction Log

4 Solo quando avverrà Il Checkpoint le pagine saranno rese persistenti sui files dati

Auto commit in SQL Server

- In SQL Server, ogni singolo statement T-SQL è una transazione
- Una volta che lo statement è stato eseguito con successo, le modifiche divengono persistenti.
- Se al contrario durante la sua esecuzione avviene un errore, lo statement viene automaticamente annullato

- Demo

Transazioni implicite

- Si attivano tramite l'opzione:

```
SET IMPLICIT_TRANSACTIONS  ON
```

- SQL Server fa partire una transazione dopo le seguenti istruzioni:
  - ALTER TABLE, CREATE, DELETE, DROP, FETCH, GRANT, INSERT, OPEN, REVOKE, SELECT, TRUNCATE TABLE, UPDATE
- La transazione deve essere poi completata tramite COMMIT o ROLLBACK
- Valutare l'utilizzo di XACT_ABORT
  - In caso di errore, la transazione viene marcata come **Uncommitable**
  - Se non viene fatto esplicitamente, la transazione viene poi automaticamente annullata, inviando un messaggio al client.

- Demo

Transazioni esplicite

- La transazione viene avviata con il comando BEGIN TRANSACTION o BEGIN TRAN
- Viene completata con successo, rendendo tutte le modiche persistenti, tramite il comando COMMIT TRANSACTION o COMMIT TRAN o semplicemente COMMIT
- Viene completamente annullata tramite ROLLBACK TRANSACTION o ROLLBACK TRAN o semplicemente ROLLBACK

Demo

Savepoint

- E' possibile salvare uno stato intermedio di una transazione

```
SAVE TRANSACTION SavePointName
```

e decidere, in caso di errore o inconsistenza, di non annullare tutte le operazioni, ma tornare allo stato salvato

```
ROLLBACK TRANSACTION SavePointName
```

- La transazione rimane comunque in piedi fino a quando non si effettua il COMMIT o ROLLBACK completo

- Demo

Transazioni annidate

- Si parla di transazioni annidate quando all'interno di una transazione ne viene avviata un'altra, prima che questa sia stata completata.
- Il livello di annidamento è determinabile tramite la variabile di sistema @@TRANCOUNT

```
BEGIN transaction
--Esterna
    BEGIN transaction
    --Interna
    select @@trancount --2
```

- Nelle transazioni annidate, Il COMMIT ed il ROLLBACK si comportano diversamente:
  - Se si invoca il COMMIT, viene completata solo la transazione più interna e @@trancount decrementato di 1
  - Se si invoca Il ROLLBACK, vengono invece annullate tutte le transazioni e @@trancount viene posto a 0
    - Utilizzare i savepoints per un maggior controllo del rollback

Concorrenza

- Capacità di un sistema di consentire a più utenti di accedere o modificare dati condivisi contemporaneamente.

  - Maggiore è il numero di utenti attivi in grado di lavorare su dati condivisi, maggiore è il **livello di concorrenza** offerto dal sistema
  - Maggiore è il livello di concorrenza, maggiore è la probabilità di **conflitto**, ossia che almeno un utente tenti di modificare il dato condiviso mentre gli altri lo stanno leggendo/modificando.

  - La gestione dei conflitti influenza il livello di concorrenza se consideriamo come priorità l'affidabilità del dato

Problemi di concorrenza

- Una gestione non ottimale dei conflitti può causare i seguenti fenomeni indesiderati:
- **Dirty read**
  - Si verifica quando una transazione legge un dato appena modificato da un'altra transazione che però ancora non si è completata : questa potrebbe essere poi annullata ed il dato sarebbe non confermato, quindi utilizzato impropriamente
- **Non-repeatable Read**
  - Una transazione si trova a leggere valori diversi del dato in tempi successivi perché nel mezzo un'altra transazione lo ha modificato(es: prezzo prodotto)
- **Phantom Read**
  - Una transazione legge un insieme di dati ma prima che questi possano essere rielaborati, un'altra transazione effettua modifiche o inserimenti che alterano il subset letto precedentemente, rendendolo inconsistente.
- **Lost Update**
  - Due transazioni leggono lo stesso dato pressoché in simultanea, ma poi lo aggiornano in tempi differenti: soltanto l'ultimo verrà salvato sovrascrivendo l'altro come se non fosse mai avvenuto(es: contatore)
- **Double Read**
  - Causati da modifiche al valore di un indice mentre un'altra transazione è in attesa di leggere i dati

Problemi di concorrenza

- Dirty read.

```
BEGIN TRANSACTION --T1

update webinar.dbo.prodotti set prezzo*=.5
where idprodotto=1
```

```
BEGIN TRANSACTION --T2

select prezzo from webinar.dbo.prodotti
where idprodotto=1
……….
```

```
ROLLBACK TRANSACTION
```

Problemi di concorrenza

- Non repeatable read.

```
BEGIN TRANSACTION --T1

declare @prezzo int
select @prezzo=prezzo
from webinar.dbo.prodotti where idprodotto=1
--1^ lettura:--prezzo viene mostrato maschera di acquisto
select @prezzo


---elaborazione..
 WAITFOR DELAY '00:00:10’



 --chiamata a pagina di riepilogo
select @prezzo=prezzo
from webinar.dbo.prodotti where idprodotto=1

--2^ Lettura:è cambiato
select @prezzo

COMMIT TRANSACTION
```

```
BEGIN TRANSACTION --T2

 update webinar.dbo.prodotti
 set prezzo+=1
 where idprodotto=1

COMMIT TRANSACTION
```

Problemi di concorrenza

Phantom read.

```
BEGIN TRANSACTION

declare @nIniziale int
select @nIniziale= Count(*) from northwind.dbo.categories
where categoryname between 'a' and ‘g'
select @nIniziale

--effettuo varie opearazioni, ad esempio inserimenti...
insert into  northwind.dbo.categories (categoryname) values('Cereali’)

WAITFOR DELAY '00:00:10’;
```

```
BEGIN TRANSACTION
insert into  northwind.dbo.categories (categoryname)
values(‘Frutta')
--aggiorno altro record non letto da T1 che però
 andrebbe così nel range di ricerca
update  northwind.dbo.categories set CategoryName='Banane’
where categoryid=6
COMMIT TRANSACTION
```

```
declare @nFinale int
select @nFinale= Count(*) from  northwind.dbo.categories
where categoryname between 'a' and 'e’
--calcolo numero nuovi record per differenza
select 'n. inserimenti:', (@nFinale-@nIniziale)

COMMIT TRANSACTION
```

Problemi di concorrenza

Lost Update

```
BEGIN TRANSACTION --T1

declare @contatore int
select @contatore=visite from webinar.dbo.VisiteSiti
where idsito=1
select 'visite lette :', @contatore

--simulazione ritardo
 WAITFOR DELAY '00:00:15;

set @contatore=@contatore+1
update webinar.dbo.VisiteSiti set visite=@contatore
where idsito=1

COMMIT TRANSACTION
```

```
BEGIN TRANSACTION --T2

declare @contatore int
select @contatore=visite from webinar.dbo.VisiteSiti
where idsito=1
set @contatore=@contatore+1
update webinar.dbo.VisiteSiti set visite=@contatore
where idsito=1

COMMIT TRANSACTION
```

Problemi di concorrenza

Double Read

```
--T1
BEGIN TRANSACTION

--prezzo prodotto1: 1
--prezzo prodotto2: 2

update webinar.dbo.prodottidoubleread set prezzo=15
where nome='prodotto2’
```

```
--T2
select nome,prezzo from
Webinar.dbo.prodottidoubleread
where prezzo<=1 or prezzo >=10
```

```
update webinar.dbo.prodottidoubleread set prezzo=20
where nome='prodotto1'

COMMIT TRANSACTION
```

nome	prezzo
prodotto1	1.00
prodotto2	15.00
prodotto1	20.00

Livelli di Isolamento

- Read Uncommitted
  - Alta concorrenza, Bassa consistenza
- Read Committed
  - Buona Concorrenza, Buona consistenza
- Repeatable Read
  - Discreta Concorrenza, Alta consistenza
- Serializable
  - Bassa Concorrenza, Massima Consistenza

Isolation level	Dirty read	Nonrepeatable read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable	No	No	No

Concorrenza Pessimistica

- Chiamata così perché presuppone che qualcosa possa andare storto e cerca di impedire che ciò accada.
- Modalità di gestione ottimale quando:
  - Si stima che i conflitti possano avvenire frequentemente
  - I tempi di utilizzo dei dati condivisi è breve
- E' basata sui lock: solo un utente alla volta può accedere al dato
  - Conviene se il costo di mantenere i dati bloccati è inferiore a quello di dover annullare un'operazione a causa di un conflitto
  - Modello utilizzato di default da SQL Server

Principali tipi di Lock

- Shared lock (S)
  - Si acquisisce quando si legge un dato e ne garantisce l'integrità.
  - Impedisce la modifica da parte delle altre transazioni fino al rilascio
  - Può essere acquisito da più transazioni
- Exclusive lock (X)
  - Si acquisisce quando si modifica un dato
  - Impedisce letture e scritture di altre transazioni
  - Solo una transazione (la prima) può acquisire un lock esclusivo
  - Viene mantenuto fino al termine della transazione
- Update lock (U)
  - Si acquisisce durante una fase di modifica ed è una combinazione di shared ed exclusive lock.
  - Un Update lock inizia con la ricerca delle righe da modificare e si trasforma in exclusive lock quando la modifica inizia.
  - Solo una transazione (la prima) può acquisire un update lock
  - Le altre transazioni possono acquisire solo uno shared lock.
- Intent locks (IX,IS,SIX) (Lock preventivi)
  - Lock creati da SQL Server a livello superiore per permettere alla transazione corrente di poter poi acquisire i propri lock (S e X) senza interferenza da parte di altre transazioni a più alti livelli

Lock escalation

- I lock possono avvenire a vari livelli, i principali sono:
  - **RID** (heap) **/ KEY** (indice) (identificativi usati per lock livello row)
  - **Pagina** (blocco di 8K che può contenere 1 o più righe, unità fisica fondamentale per i files dati in SQL Server)
  - **Extent** (blocco logico di 8 pagine utilizzato per velocizzare ricerche ed inserimenti)
  - **Tabella** (dati ed indici)

- Se i lock ad un livello risultano numerosi, il costo di gestione potrebbe superare quello di un lock a livello superiore, quindi in questi casi SQL Server effettua una escalation
- Tipica è l'escalation da lock di livello Riga a lock di livello Tabella, quando i lock sulle righe sono troppi(>5000)
  - L'escalation crea però lock più pesanti ed in alcuni casi inutili, ad esempio in questo caso sulle righe che non sarebbero dovute essere bloccate

```
ALTER TABLE ... SET (LOCK_ESCALATION = DISABLE)
```

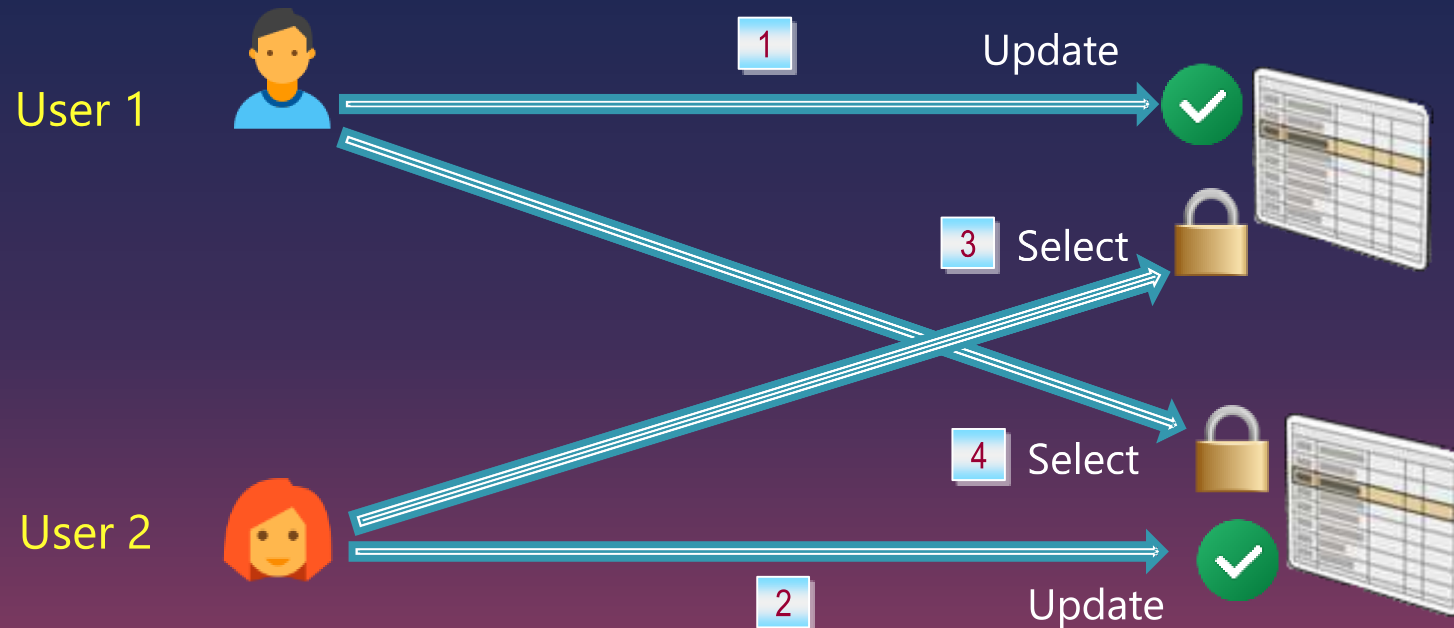
Override dei livelli di isolamento: Lock hints

- E' possibile specificare il tipo di lock direttamente nelle istruzioni SQL, ecco I più utilizzati:
- **TABLOCK**.
  - Lock S, blocca sutte le modifiche sulla tabella
- **TABLOCKX**
  - Lock S e X, blocca tutte le letture e scritture delle altre transazioni sulla tabella
- **UPDLOCK**
  - Permette di acquisire un UPDATE LOCK
- **HOLDLOCK**
  - Equivale a SERIALIZABLE
- **NOLOCK**
  - Equivale a READ UNCOMMITED, utilizzare solo nelle SELECT

- Se non usati in una transazione implicita o esplicita, i lock durano solo il tempo di esecuzione dell'istruzione

```
--Con una semplice SELECT blocca in maniera esclusiva (S e X)
 tutta la tabella, quindi anche letture su altre righe
BEGIN TRAN
select * from production.product with (tablockx) where productid=1
```

Deadlocks



- SQL Server effettua periodicamente un controllo dei deadlocks correnti:
  - Viene eseguito ogni 5 secondi
  - Appena intercettato un deadlock, viene scelta la transazione «vittima» di cui viene effettuato il Rollback e restituito l'errore 1205 al client
  - A parità di priorità, viene scelta come vittima la transazione che costa meno annullare (fa riferimento il transaction log)
  - A parità di priorità e di costo, la vittima è casuale.

Demo

Deadlock priority

- Tutte le transazioni sono impostate di default allo stesso livello.
- E' possible cambiare la priorità di una transazione tramite:

```
SET DEADLOCK_PRIORITY HIGH; -- { HIGH | NORMAL | LOW }
```

  - In questo caso,viene scelta come vittima la transazione con priorità minore
- Per un maggior controllo è possibile impostare un valore numerico da -10 a +10

```
SET DEADLOCK_PRIORITY 10; -- [-10,-9,...,9, 10]
```

- In caso di valori misti, considerare le seguenti corrispondenze: **LOW** (-5), **NORMAL** (0), **HIGHT** (+5)

Demo

Monitorare i deadlocks

- SQL Server Error Log
  - EXEC **sp_altermessage 1205**, 'WITH_LOG', 'true'
  - EXEC **xp_ReadErrorLog** 0, 1, N'deadlock'
  - Difficile investigazione, poche informazioni
- Alert su errore **1205**
  - Per essere avvisati in tempo reale del verificarsi di deadlocks
  - Permette di raccogliere poche informazioni
  - Sconsigliato se i deadlocks sono frequenti
- Vista dinamica **sys.dm_os_performance_counters**
  - Per contare numero deadlocks e calcolare medie (giornaliere, mensili..)
- SQL Profiler
  - Ricco di informazioni (deadlock graph) e permette salvataggio su file
  - Impegnativo per le risorse, da utilizzare per indagini mirate e limitate nel tempo
- Extended event
  - Ricco di informazioni (deadlock graph) e permette salvataggio su file
  - Soluzione ottimale se si vuole tenerne traccia costante con minimo impegno del Sistema

Demo

Best practice

- Usare transazioni solo se necessario.
- Creare transazioni rapide per minimizzare i tempi di lock, lavorando sulle ottimizzazioni delle query coinvolte
- Utilizzare un livello di isolamento più basso possible, prevedendo quali fenomeni indesiderano possono realmente verificarsi.
- Evitare interazione con l'input durante una transazione, i tempi di completamento non sarebbero prevedibili.
- Evitare se possible i Triggers, essi introducono transazioni. Se è necessario utlizzarli, prestare grande attenzione al Rollback, esso annullerebbe l'intero contesto transazionale.

Ridurre lock e deadlocks: Concorrenza Ottimistica

- Il nome deriva dal presupposto ottimistico che i conflitti tra le transazioni si verifichino raramente o comunque non in maniera elevata
- Non è basata sui lock, che quindi vengono minimizzati e con essi anche i deadlocks
- Utilizza il controllo delle versioni delle righe, mantenendole su **tempddb**, per capire se è avvenuto un conflitto
  - Maggiore impegno di tempdb, valutare potenziamento del sottosistema di I/O
- Genera una eccezione ed annulla la transazione se rileva che il conflitto ha causato una inconsistenza del dato
- SQL Server supporta 2 modalità di concorrenza ottimistica:
  - **Read Committed Snapshot**
  - **Snapshot Isolation**

Modalità Concorrenza Ottimistica

- **Read Committed Snapshot**
  - Si abilita con `ALTER DATABASE <<database>> SET READ_COMMITTED_SNAPSHOT ON`
  - Qualsiasi query eseguita con il livello di isolamento Read Committed non acquisisce lock e le letture effettuano automaticamente una scansione dello snapshot alla ricerca della versione corretta, in genere l'ultima committata
- **Snapshot Isolation**
  - Si abilita con `ALTER DATABASE <<database>> SET ALLOW_SNAPSHOT_ISOLATION ON`
  - Viene abiitato un nuovo ISOLATION LEVEL: **SNAPSHOT**
  - Nessun fenomeno indesiderato può più manifestarsi: il livello SHAPSHOT si comporta come l'equivalente pessimistico SERIALIZABLE minimizzando i lock e generando eccezioni in caso di conflitto

Isolation level	Dirty read	Nonrepeatable read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Snapshot	No	No	No
Serializable	No	No	No

Demo

Benefici nell'uso del row versioning

- Letture effettuate su uno shapshot consistente del database (last committed)
- Letture senza lock anche se è in corso una modifica in un'altra transazione
- Minor numero di lock, quindi di blocchi ed attese
- Riduzione casi Lock Escalation
- Riduzione Deadlocks

Costi del row versioning

- 14 bytes aggiuntivi per ogni riga
- Maggior uso di CPU e memoria
- Gli aggiornamenti sono più lenti a causa della creazione/mantenimento delle versioni (Demo)
- Le letture possono rallentare se devono ricostruire troppe versioni delle righe
- Spazio aggiuntivo e maggior impegno di tempdb (Demo)
- Ad ogni Update e Delete, anche se non facenti parte di transazioni che ne avrebbero bisogno, viene comunque creato uno snapshot e mantenuto il row versioning

Conclusioni finali

- Iniziare utilizzando un approccio basato su concorrenza pessimistica: valutare un eventuale cambio di strategia in caso di deadlock frequenti o attese eccessive dovute ai lock
- Se si decide di utilizzare la concorrenza ottimistica, abilitare in primis solamente Read committed Snapshot, monitorando ancora i deadlock e valutando variazioni significative nei tempi di esecuzione delle query più pesanti
  - Nel caso potenziare sottosistema I/O
- Passare infine a snapshot solo se il numero di lock e deadlock è ancora non accettabile

Grazie

ns12@ns12.it www.ns12.it

Via di Settebagni 390, 00139 Roma

Via Don Tosatto 127, 30174 Venezia

Corso Novara 10 , 80143 Napoli

Viale Angelo Masini 12/14 ,- 40126 Bologna