

# Tranzacții. Controlul concurenței în SQL Server

Seminar 3



# Tranzacții în SQL Server

- SQL Server utilizează tranzacții pentru compunerea mai multor operații într-o singură unitate de lucru
- Acțiunile fiecărui utilizator sunt procesate utilizând o tranzacție diferită
- Pentru a maximiza *throughput*-ul, tranzacțiile ar trebui să se execute în paralel
- Proprietățile ACID ale tranzacțiilor au ca efect final serializabilitatea
- Dacă o tranzacție este efectuată cu succes, toate modificările realizate în timpul tranzacției sunt comise și devin permanente în baza de date
- Dacă apar erori în timpul unei tranzacții și aceasta trebuie anulată (canceled/rolled back), atunci toate modificările realizate în timpul tranzacției sunt șterse

# Tranzacții în SQL Server

Mecanisme pentru invocarea tranzacțiilor:

- Fiecare comandă este o tranzacție dacă nu se specifică altceva (tranzacții autocommit)
- BEGIN TRAN, ROLLBACK TRAN, COMMIT TRAN sunt cel mai des utilizate (tranzacții explicite)
  - BEGIN TRAN marchează punctul de început al unei tranzacții explicite
  - ROLLBACK TRAN întoarce o tranzacție implicită sau explicită până la începutul tranzacției sau până la un *savepoint* din interiorul tranzacției
  - COMMIT TRAN marchează finalul unei tranzacții implicite sau explicite efectuate cu succes
- SET IMPLICIT\_TRANSACTIONS ON (tranzacții implicite)
  - Se marchează doar finalul tranzacției (tranzacții înlanțuite)
  - În acest caz, o nouă tranzacție este începută în mod implicit atunci când tranzacția anterioară este finalizată (fiecare tranzacție este finalizată explicit cu COMMIT sau ROLLBACK)



# Tranzacții în SQL Server

- SET XACT\_ABORT ON
  - Orice erori SQL vor duce la roll back-ul tranzacției
- @@TRANCOUNT returnează numărul de instrucțiuni BEGIN TRANSACTION care au avut loc pe conexiunea curentă
  - Instrucțiunile BEGIN TRANSACTION incrementează @@TRANCOUNT cu 1
  - Instrucțiunile COMMIT TRANSACTION decrementează @@TRANCOUNT CU 1
  - ROLLBACK TRANSACTION setează @@TRANCOUNT pe 0, excepție ROLLBACK TRANSACTION *savepoint\_name*, care nu afectează @@TRANCOUNT
- SAVE TRANSACTION *savepoint\_name*
  - Setează un *savepoint* într-o tranzacție

# Tipuri de tranzacții

- SQL Server poate utiliza tranzacții locale sau distribuite
- Imbricarea tranzacțiilor este permisă (dar nu tranzacțiile imbricate)
- Savepoints cu nume - rollback-ul unei porțiuni dintr-o tranzacție
- Un *savepoint* definește o locație la care o tranzacție se poate întoarce dacă o parte a tranzacției este anulată condiționat



# Probleme de concurență

Izolarea tranzacțiilor în SQL Server rezolvă patru probleme majore de concurență:

- Lost updates
  - Când doi scriitori modifică aceleași date
- Dirty reads
  - Când un cititor citește date necomise
- Unrepeatable reads
  - Când o înregistrare existentă se schimbă în cadrul unei tranzacții
- Phantom reads
  - Când sunt adăugate noi înregistrări și apar în cadrul unei tranzacții

# Probleme de concurență

- SQL Server asigură izolarea prin blocări
- Blocările pentru scriere sunt blocări exclusive
- Blocările pentru citire permit existența altor cititori
- O tranzacție well-formed obține tipul corect de blocare înainte de a utiliza datele
- O tranzacție two-phased menține blocările până când se obțin toate
- Nivelurile de izolare determină durata blocărilor (cât timp sunt menținute acestea)



# Blocarea în SQL Server

- Blocările sunt gestionate de obicei dinamic de **Lock Manager**, nu prin intermediul aplicațiilor
- Blocările sunt menținute pe resurse SQL Server, cum ar fi înregistrări citite sau modificate în timpul unei tranzații, pentru a preveni utilizarea concurentă a resurselor de către diferite tranzații
- Granularitatea blocărilor:
  - Row/Key
  - Page
  - Extent (grup contiguu de 8 pagini)
  - Table
  - Database



# Blocarea în SQL Server

- Blocările pot fi atribuite la mai mult de un nivel → ierarhie de blocări înrudite
- Lock escalation > 5000 blocări pe obiect
  - Este procesul de conversie a mai multor blocări cu granulație mai fină (fine-grain locks) în mai puține blocări cu granulație mai grosieră (coarse-grain locks), pentru a reduce supraîncărcarea sistemului
  - Lock escalation se declanșează atunci când nu este dezactivat la nivel de tabel (folosind opțiunea ALTER TABLE SET LOCK\_ESCALATION) și când una din următoarele condiții există:
    - O singură instrucțiune Transact-SQL obține cel puțin 5000 de blocări pe un singur index sau tabel nepartiționat
    - O singură instrucțiune Transact-SQL obține cel puțin 5000 de blocări pe o singură partiție a unui tabel partiționat și opțiunea ALTER TABLE SET LOCK\_ESCALATION este setată pe AUTO
    - Numărul de blocări pe o instanță Database Engine depășește capacitatea memoriei sau limitele configurației

# Blocarea în SQL Server

- Durata blocărilor (precizată prin niveluri de izolare):
  - Până la finalizarea operației
  - Până la finalul tranzacției
- Tipuri de blocări:
  - Shared (S) – pentru citirea datelor
  - Update (U) – blocare S -> se anticipează X
  - Exclusive (X) – blocare pentru modificarea datelor ce este incompatibilă cu alte tipuri de blocări (excepție hint-ul NOLOCK și nivelul de izolare READ UNCOMMITTED)
  - Intent (IX, IS, SIX) – intenție de blocare (îmbunătățire a performanței)
  - Schema (Sch-M, Sch-S) – modificare schemă, stabilitate schemă (Sch-M și Sch-S nu sunt compatibile)
  - Bulk Update (BU) – blochează tot tabelul în timpul unui bulk insert
  - Key range – blochează mai multe înregistrări pe baza unei condiții



# Blocarea în SQL Server

	Shared Lock (S)	Update Lock (U)	Exclusive Lock (X)
Shared Lock (S)	Da	Da	Nu
Update Lock (U)	Da	Nu	Nu
Exclusive Lock (X)	Nu	Nu	Nu

# Blocări speciale

## Blocări schema

- Nu blochează DML, doar DDL
- Folosite atunci când se execută o operație dependentă de schema unui tabel
- Tipuri
  - **Schema modification**
    - Se folosește în timpul execuției unei operații DDL
    - Previne accesul concurent la tabel
    - Folosită de unele operații DML (cum ar fi table truncation) pentru a preveni accesul operațiilor concurente la tabelele afectate



# Blocări speciale

## – Schema stability

- Folosită în timpul compilării și execuției interogărilor
- Nu blochează alte blocări tranzacționale, inclusiv blocări exclusive (X)
- Blochează operațiile concurente DDL și DML care obțin blocări Schema modification

## Blocări Bulk Update

- Blocare explicită (TABLOCK)
- Automat în timpul SELECT INTO
- Folosită atunci când are loc o copiere de tipul bulk a datelor într-un tabel și este specificat hint-ul TABLOCK

## Blocări Application

- Aplicațiile pot utiliza lock manager-ul intern al SQL Server (blocare resurse aplicație)

# Niveluri de izolare în SQL Server

- READ UNCOMMITTED: fără blocare la citire
- READ COMMITTED: menține blocările pe durata execuției instrucțiunii (default) (elimină dirty reads)
- REPEATABLE READ: menține blocările pe durata tranzacției (elimină unrepeatable reads)
- SERIALIZABLE: menține blocările și blocările key range pe durata întregii tranzacții (elimină phantom reads)
- SNAPSHOT: lucrează pe un snapshot al datelor
- Sintaxă SQL:

```
SET TRANSACTION ISOLATION LEVEL <isolation_level>
```



## Grade de izolare

Denumire	Haos	Read Uncommitted	Read Committed	Repeatable Read	Serializable
Lost Updates	Da	Nu	Nu	Nu	Nu
Dirty Reads	Da	Da	Nu	Nu	Nu
Unrepeatable Reads	Da	Da	Da	Nu	Nu
Phantom Reads	Da	Da	Da	Da	Nu

## Blocare Key Range

- Blochează seturi de înregistrări controlate de un predicat
- Exemplu:

```
SELECT nume, cantitate FROM Produse  
  
WHERE cantitate BETWEEN 2500 AND 5000;
```

- Este necesară blocarea datelor care nu există
- Dacă utilizarea “cantitate BETWEEN 2500 AND 5000” nu returnează nicio înregistrare prima dată, nu ar trebui să returneze înregistrări nici la scanările ulterioare (nu se pot insera înregistrări noi dacă au valoarea specificată pentru cantitate cuprinsă între 2500 și 5000)



# Blocări Transaction Workspace

Indică o conexiune

- Fiecare conexiune la o bază de date obține blocare *Shared\_Transaction\_Workspace*
- Excepții
  - conexiunile la master și tempdb

Utilizate pentru prevenirea:

- DROP
- RESTORE

# Deadlocks

- Un deadlock are loc atunci când două sau mai multe tranzații se blochează reciproc permanent din cauza faptului că fiecare tranzație deține o blocare pe o resursă pentru care cealaltă tranzație încearcă să obțină o blocare
- Exemplu:
  - Tranzacția T1 are o blocare pe o resursă R1 și a cerut o blocare pe resursa R2
  - Tranzacția T2 are o blocare pe o resursă R2 și a cerut o blocare pe resursa R1
  - Apare o stare de deadlock deoarece nicio tranzație nu poate continua decât în momentul în care o resursă este disponibilă și nicio resursă nu poate fi eliberată până când o tranzație nu își continuă execuția
- SQL Server recunoaște un deadlock
- Implicit, tranzația mai nouă este terminată
  - Eroarea 1205 – ar trebui detectată și gestionată corespunzător



# Deadlocks

- SET LOCK TIMEOUT
  - se menționează cât timp (în milisecunde) o tranzacție așteaptă ca un obiect blocat să fie eliberat (0 = terminare imediată)
- Un utilizator poate specifica prioritatea sesiunilor într-o situație de deadlock folosind instrucțiunea SET DEADLOCK PRIORITY (în mod implicit este setată pe NORMAL)
- DEADLOCK PRIORITY poate fi setată pe:
  - Low
  - Normal
  - High
  - O valoare numerică cuprinsă între -10 și 10
- Dacă două sesiuni au diferite valori setate pentru DEADLOCK PRIORITY , sesiunea cu valoarea cea mai scăzută este aleasă ca victimă a deadlock-ului

# Reducerea situațiilor de deadlock

- Tranzacții scurte și într-un singur batch
- Colectarea și verificarea datelor input de la utilizatori înainte de deschiderea unei tranzacții
- Accesarea resurselor în aceeași ordine în tranzacții
- Utilizarea unui nivel de izolare mai scăzut sau a unui nivel de izolare row versioning
- Accesarea celei mai mici cantități posibile de date în tranzacții



## Tranzacții - Exemplu

- În SQL Server, vom crea o bază de date numită "SGBDIR"
- După crearea bazei de date, vom crea trei tabele noi:
- Primul tabel se va numi *Presents*:

```
CREATE TABLE Presents  
(  
    present_id INT PRIMARY KEY IDENTITY,  
    description VARCHAR(100),  
    owner VARCHAR(100),  
    price REAL  
);
```

# Tranzacții - Exemplu

- Al doilea tabel se va numi *Movies*:

```
CREATE TABLE Movies  
(  
    movie_id INT PRIMARY KEY IDENTITY,  
    title VARCHAR(100),  
    runtime INT  
);
```



# Tranzacții - Exemplu

- Al treilea tabel se va numi *Actors*:

```
CREATE TABLE Actors  
(  
    actor_id INT PRIMARY KEY IDENTITY,  
    firstname VARCHAR(100),  
    lastname VARCHAR(100)  
);
```

# Tranzacții - Exemplu

- După aceea, vom insera niște înregistrări noi în tabele:

```
INSERT INTO Presents (description, owner, price) VALUES  
('pony', 'Jack', 10000), ('candle', 'Jane', 2.5),  
('chocolate', 'James', 3);
```

Rezultat:

	present_id	description	owner	price
1	1	pony	Jack	10000
2	2	candle	Jane	2.5
3	3	chocolate	James	3



## Tranzacții - Exemplu

```
INSERT INTO Movies (title, runtime) VALUES ('IT', 135),  
('Black Panther', 134), ('Red Sparrow', 140);
```

Rezultat:

	movie_id	title	runtime
1	1	IT	135
2	2	Black Panther	134
3	3	Red Sparrow	140

## Tranzacții - Exemplu

```
INSERT INTO Actors (firstname, lastname) VALUES  
('Bill', 'Skarsgard'), ('Chadwick', 'Boseman'),  
('Jennifer', 'Lawrence');
```

Rezultat:

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jennifer	Lawrence



# Tranzacții - Exemplu

- Exemplu de tranzacție explicită:

```
BEGIN TRAN;
```

```
UPDATE Presents SET description='horse' WHERE  
owner='Jack';
```

```
DELETE FROM Presents WHERE price=3;
```

```
COMMIT TRAN;
```

Rezultat:

	present_id	description	owner	price
1	1	horse	Jack	10000
2	2	candle	Jane	2.5

# Tranzacții - Exemplu

- Exemplu de tranzacție implicită:

```
SET IMPLICIT_TRANSACTIONS ON;
```

```
INSERT INTO Presents (description, owner, price) VALUES  
('car', 'Sam', 40000);
```

```
UPDATE Presents SET price=4 WHERE owner='Jane';
```

```
COMMIT TRAN;
```

Rezultat:

	present_id	description	owner	price
1	1	horse	Jack	10000
2	2	candle	Jane	4
3	4	car	Sam	40000



# Tranzacții - Exemplu

- Exemplu de imbricare a tranzacțiilor: (SET IMPLICIT\_TRANSACTIONS OFF)

```
BEGIN TRAN;
```

```
UPDATE Presents SET description='bike' WHERE owner='Jane';
```

```
BEGIN TRAN;
```

```
INSERT INTO Presents (description, owner, price) VALUES  
( 'necklace', 'Sharon', 50);
```

```
COMMIT TRAN;
```

```
UPDATE Presents SET price=100 WHERE owner='Jane';
```

```
ROLLBACK TRAN;
```

# Tranzacții - Exemplu

- Rezultat:

	present_id	description	owner	price
1	1	horse	Jack	10000
2	2	candle	Jane	4
3	4	car	Sam	40000

- După cum putem observa, ambele tranzacții au primit un rollback deoarece tranzacția exterioară a primit un rollback
- Dacă tranzacția cea mai exterioară primește un rollback, atunci toate tranzacțiile interioare primesc un rollback, indiferent dacă au fost sau nu comise individual



# Tranzacții - Exemplu

- Exemplu de tranzacție autocommit:

```
SELECT * FROM Presents;
```

Rezultat:

	present_id	description	owner	price
1	1	horse	Jack	10000
2	2	candle	Jane	4
3	4	car	Sam	40000

- Când IMPLICIT\_TRANSACTIONS este setat pe OFF, fiecare instrucțiune Transact-SQL este mărginită de către o instrucțiune BEGIN TRANSACTION invizibilă și de către o instrucțiune COMMIT TRANSACTION invizibilă

# Tranzacții - Exemplu

- Exemplu de tranzacție cu savepoint:

```
BEGIN TRAN;
```

```
INSERT INTO Movies (title, runtime) VALUES ('Frozen', 109);
```

```
SAVE TRAN savepoint;
```

```
INSERT INTO Actors (firstname, lastname) VALUES  
('Kristen', 'Bell');
```

```
ROLLBACK TRAN savepoint;
```

```
COMMIT TRAN;
```



# Tranzacții - Exemplu

Rezultat:

	movie_id	title	runtime
1	1	IT	135
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jennifer	Lawrence

- Din cauza faptului că tranzacția face un rollback până la savepoint și este apoi comisă, modificarea făcută de către prima instrucțiune INSERT a fost salvată

# Probleme de concurență - Exemplu

- Exemplu dirty reads:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
UPDATE Movies SET runtime=200 WHERE title='IT';
```

```
WAITFOR DELAY '00:00:07';
```

```
ROLLBACK TRAN;
```



## Probleme de concurență - Exemplu

- În a doua fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
  
BEGIN TRAN;  
  
SELECT * FROM Movies;  
  
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

## Probleme de concurență - Exemplu

- În a doua fereastră de interogare putem vedea următorul rezultat:

	movie_id	title	runtime
1	1	IT	200
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

- Deoarece nivelul de izolare a tranzacției a fost setat pe READ UNCOMMITTED în a doua fereastră de interogare, a doua tranzacție a citit date necomise (dirty reads)



# Probleme de concurență - Exemplu

- După execuția celor două tranzacții, vom executa următoarea interogare:

```
SELECT * FROM Movies;
```

- Deoarece prima tranzacție face un rollback, modificarea făcută de către aceasta nu a fost salvată:

	movie_id	title	runtime
1	1	IT	135
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

- Deci cea de-a doua tranzacție a citit date care nu au fost persistate pe disc

# Probleme de concurență - Exemplu

- Exemplu unrepeatable reads:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
SELECT * FROM Movies;
```

```
WAITFOR DELAY '00:00:06';
```

```
SELECT * FROM Movies;
```

```
COMMIT TRAN;
```



# Probleme de concurență - Exemplu

- În a doua fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
WAITFOR DELAY '00:00:03';
```

```
UPDATE Movies SET runtime=200 WHERE title='IT';
```

```
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

# Probleme de concurență - Exemplu

- În prima fereastră de interogare putem vedea următorul rezultat:

	movie_id	title	runtime
1	1	IT	135
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

	movie_id	title	runtime
1	1	IT	200
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

- După cum putem observa, aceeași interogare executată de două ori în aceeași tranzacție a returnat două valori diferite pentru aceeași înregistrare (unrepeatable reads)



# Probleme de concurență - Exemplu

- Exemplu phantom reads:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
SELECT * FROM Actors WHERE actor_id BETWEEN 1 AND 100;
```

```
WAITFOR DELAY '00:00:06';
```

```
SELECT * FROM Actors WHERE actor_id BETWEEN 1 AND 100;
```

```
COMMIT TRAN;
```

## Probleme de concurență - Exemplu

- În a doua fereastră de interogare punem următoarea tranzacție:

-- !!!Asigurați-vă că IDENTITY generează o valoare cuprinsă în intervalul specificat (actor\_id trebuie să aibă o valoare cuprinsă între 1 și 100)

```
BEGIN TRAN;
```

```
WAITFOR DELAY '00:00:03';
```

```
INSERT INTO Actors (firstname, lastname) VALUES  
( 'Alexander', 'Skarsgard' );
```

```
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)



## Probleme de concurență - Exemplu

- În prima fereastră de interogare putem vedea următorul rezultat:

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jennifer	Lawrence

---

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jennifer	Lawrence
4	5	Alexander	Skarsgard

- În aceeași tranzacție, o interogare care specifică un interval de valori în clauza WHERE a fost executată de două ori și numărul de înregistrări incluse în cel de-al doilea result set este mai mare decât numărul de înregistrări incluse în primul result set (phantom reads)

# Probleme de concurență - Exemplu

- Exemplu deadlock:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN TRAN;
```

```
UPDATE Movies SET runtime=135 WHERE title='IT';
```

```
WAITFOR DELAY '00:00:05';
```

```
UPDATE Actors SET firstname='Jen' WHERE  
lastname='Lawrence';
```

```
COMMIT TRAN;
```



## Probleme de concurență - Exemplu

- În a doua fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN TRAN;
```

```
UPDATE Actors SET firstname='Jenny' WHERE  
lastname='Lawrence';
```

```
WAITFOR DELAY '00:00:05';
```

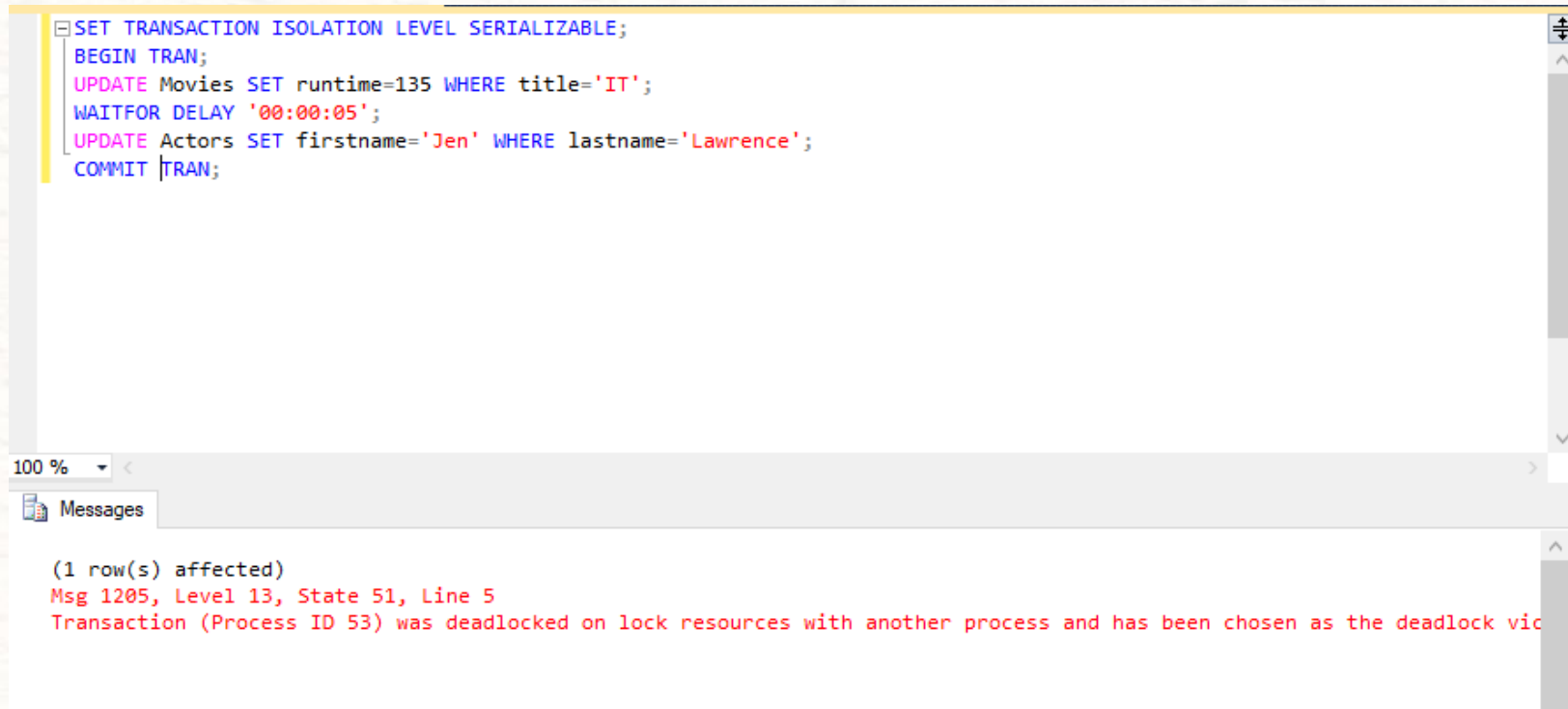
```
UPDATE Movies SET runtime=140 WHERE title='IT';
```

```
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

# Probleme de concurență - Exemplu

- A avut loc un deadlock, iar prima tranzacție a fost aleasă drept victimă a deadlock-ului și face un rollback:



```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
BEGIN TRAN;  
UPDATE Movies SET runtime=135 WHERE title='IT';  
WAITFOR DELAY '00:00:05';  
UPDATE Actors SET firstname='Jen' WHERE lastname='Lawrence';  
COMMIT TRAN;
```

100 %

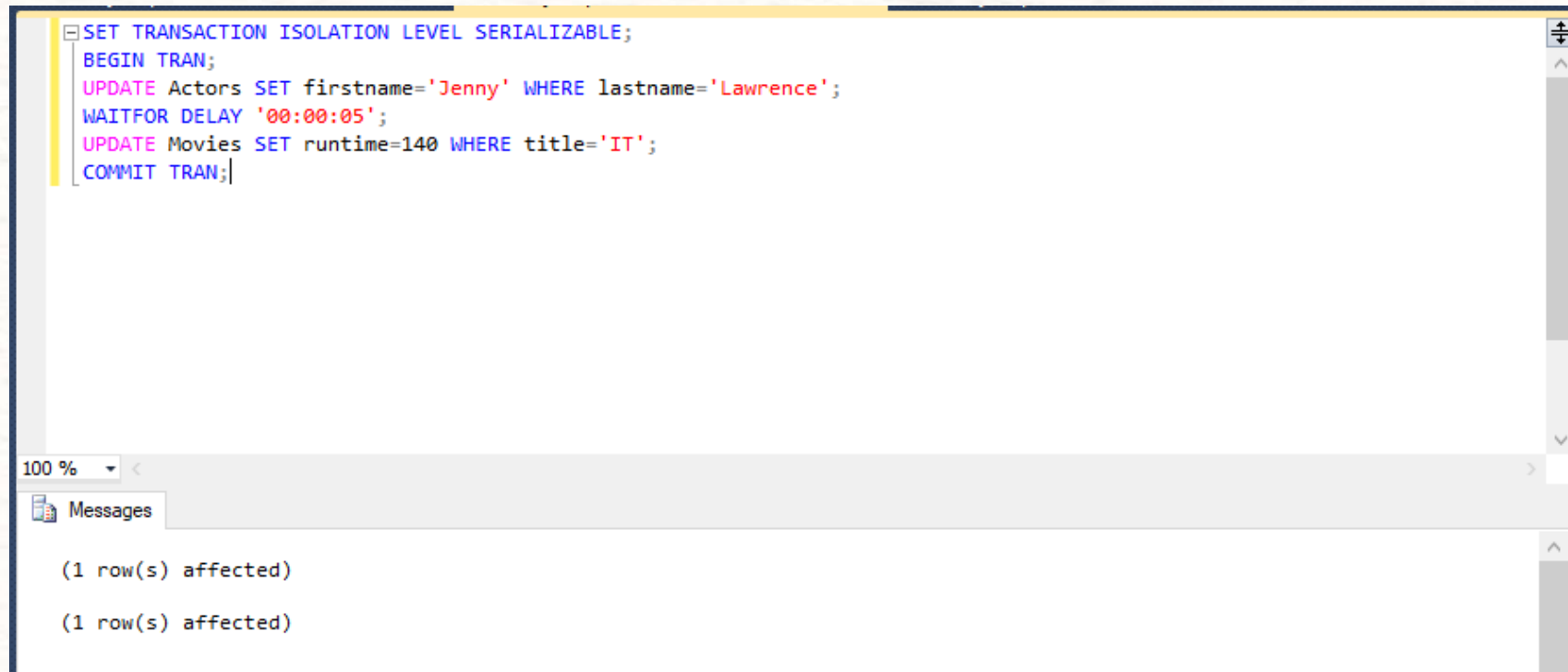
Messages

(1 row(s) affected)  
Msg 1205, Level 13, State 51, Line 5  
Transaction (Process ID 53) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. ROLLBACK TRANSACTION



# Probleme de concurență - Exemplu

- Deoarece prima tranzacție a fost aleasă drept victimă a deadlock-ului, cea de-a doua tranzacție a fost executată cu succes:



```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
BEGIN TRAN;  
UPDATE Actors SET firstname='Jenny' WHERE lastname='Lawrence';  
WAITFOR DELAY '00:00:05';  
UPDATE Movies SET runtime=140 WHERE title='IT';  
COMMIT TRAN;
```

100 %

Messages

(1 row(s) affected)

(1 row(s) affected)

# Probleme de concurență - Exemplu

- Dacă executăm următoarele interogări, putem vedea rezultatul final:

```
SELECT * FROM Movies;
```

```
SELECT * FROM Actors;
```

	movie_id	title	runtime
1	1	IT	140
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jenny	Lawrence
4	5	Alexander	Skarsgard