Transaction isolation levels

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER

Miriam Antona
Software Engineer

What is concurrency?

Concurrency: two or more transactions that read/change shared data at the same time.

Isolate our transaction from other transactions

Transaction isolation levels

- `READ COMMITTED` (default)
- `READ UNCOMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`
- `SNAPSHOT`

```
SET TRANSACTION ISOLATION LEVEL
    {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE | SNAPSHOT}
```

Knowing the current isolation level

```
SELECT CASE transaction_isolation_level
    WHEN 0 THEN 'UNSPECIFIED'
    WHEN 1 THEN 'READ UNCOMMITTED'
    WHEN 2 THEN 'READ COMMITTED'
    WHEN 3 THEN 'REPEATABLE READ '
    WHEN 4 THEN 'SERIALIZABLE'
    WHEN 5 THEN 'SNAPSHOT'
END AS transaction_isolation_level
FROM sys.dm_exec_sessions
WHERE session_id = @@SPID
```

```
| transaction_isolation_level |
|-----------------------------|
| READ COMMITTED              |
```

READ UNCOMMITTED

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

- Least restrictive isolation level
- Read rows modified by another transaction **which hasn't been committed or rolled back yet**

READ UNCOMMITTED

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

- Least restrictive isolation level
- Read rows modified by other transactions **without been committed/rolled back.**

	Dirty reads	Non-repeatable reads	Phantom reads
READ UNCOMMITTED	**yes**	**yes**	**yes**

Dirty reads

Original balance account 5 = $35,000

Transaction1

```
BEGIN TRAN
  UPDATE accounts
  SET current_balance = 30000
  WHERE account_id = 5;
```

```
ROLLBACK TRAN;
```

Transaction2

...

...

...

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT current_balance
FROM accounts WHERE account_id = 5;
```

```
| current_balance |
|-----------------|
| 30000,00        |
```

Non-repeatable reads

Transaction1

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN TRAN
    SELECT * FROM accounts WHERE account_id = 5;
```

```
| current_balance |
|-----------------|
| 35000,00        |
```

Transaction2

...

...

```
BEGIN TRAN
    UPDATE accounts
    SET current_balance = 30000 WHERE account_id = 5;
COMMIT TRAN
```

Non-repeatable reads

Transaction1

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN TRAN
    SELECT * FROM accounts WHERE account_id = 5;
```

```
| current_balance |
|-----------------|
| 35000,00        |
```

```
    SELECT * FROM accounts WHERE account_id = 5;
```

```
| current_balance |
|-----------------|
| 30000,00        |
```

Transaction2

...

...

```
BEGIN TRAN
    UPDATE accounts
    SET current_balance = 30000 WHERE account_id = 5;
COMMIT TRAN
```

Phantom reads

Transaction1

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN TRAN
SELECT * FROM accounts
    WHERE current_balance BETWEEN 45000 AND 50000
```

```
| account_number       | ... | current_balance |
|----------------------|-----|-----------------|
| 55555555552020202020 | ... | 50000,00        |
```

Transaction2

...

...

```
BEGIN TRAN
INSERT INTO accounts
    VALUES ('55555555553939393939', 1, 45000)
COMMIT TRAN
```

Phantom reads

Transaction1

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN TRAN
SELECT * FROM accounts
    WHERE current_balance BETWEEN 45000 AND 50000
```

```
| account_number       | ... | current_balance |
|----------------------|-----|-----------------|
| 55555555552020202020 | ... | 50000,00        |
```

```
SELECT * FROM accounts
    WHERE current_balance BETWEEN 45000 AND 50000
```

```
| account_number       |...| current_balance |
|----------------------|---|-----------------|
| 55555555553939393939 |...| 45000,00        | Phantom!
| 55555555552020202020 |...| 50000,00        |
```

Transaction2

...

...

```
BEGIN TRAN
INSERT INTO accounts
    VALUES ('55555555553939393939', 1, 45000)
COMMIT TRAN
```

READ UNCOMMITTED - summary

Pros:

- Can be faster, doesn't block other transactions.

Cons:

- Allows dirty reads, non-repeatable reads, and phantom reads.

When to use it?:

- Don't want to be blocked by other transactions but don't mind concurrency phenomena.
- You explicitly want to watch uncommitted data.

Let's practice!

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER

READ COMMITTED & REPEATABLE READ

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER

Miriam Antona
Software Engineer

datacamp

READ COMMITTED

- Default isolation level
- Can't read data modified by other transaction that hasn't committed or rolled back

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

READ COMMITTED - isolation level comparison

	Dirty reads	Non-repeatable reads	Phantom reads
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	**no**	**yes**	**yes**

READ COMMITTED - preventing dirty reads

Original balance account 5 = $35,000

Transaction1

```
BEGIN TRAN
  UPDATE accounts
  SET current_balance = 30000
  WHERE account_id = 5;
```

Transaction2

...

...

...

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT current_balance
FROM accounts WHERE account_id = 5;
```

Has to wait!

READ COMMITTED - preventing dirty reads

Original balance account 5 = $35,000

Transaction1

```
BEGIN TRAN
  UPDATE accounts
  SET current_balance = 30000
  WHERE account_id = 5;
```

```
COMMIT TRAN;
```

Transaction2

...

...

...

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT current_balance
FROM accounts WHERE account_id = 5;
```

```
| current_balance |
|-----------------|
| 30000,00        |
```

READ COMMITTED - selecting without waiting

Transaction1

```
BEGIN TRAN
  SELECT current_balance
  FROM accounts WHERE account_id = 5;
```

Transaction2

...

...

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT current_balance
FROM accounts WHERE account_id = 5;
```

```
| current_balance |
|-----------------|
| 35000,00        |
```

READ COMMITTED - summary

Pros:

- Prevents dirty reads

Cons:

- Allows non-repeatable and phantom reads
- You can be blocked by another transaction

When to use it?:

- You want to ensure that you only read committed data, not non-repeatable and phantom reads

REPEATABLE READ

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

- Can't read uncommitted data from other transactions
- If some data is read, other transactions cannot modify that data until REPEATABLE READ transaction finishes

REPEATABLE READ - isolation level comparison

	Dirty reads	Non-repeatable reads	Phantom reads
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	**no**	**no**	**yes**

REPEATABLE READ - preventing non-repeatable reads

Transaction1

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
    SELECT current_balance FROM accounts
    WHERE account_id = 5;
```

```
| current_balance |
|-----------------|
| 35000,00        |
```

Transaction2

...

...

```
UPDATE accounts
SET current_balance = 30000
WHERE account_id = 5;
```

Has to wait!

REPEATABLE READ - preventing non-repeatable reads

Transaction1

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
    SELECT current_balance FROM accounts
    WHERE account_id = 5;
```

```
    SELECT current_balance FROM accounts
    WHERE account_id = 5;
```

```
| current_balance |
|-----------------|
| 35000,00        |
```

```
COMMIT TRAN
```

Transaction2

...

...

```
UPDATE accounts
SET current_balance = 30000
WHERE account_id = 5;
```

Has to wait!

REPEATABLE READ - preventing non-repeatable reads

Transaction1

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
    SELECT current_balance FROM accounts
    WHERE account_id = 5;
```

```
    SELECT current_balance FROM accounts
    WHERE account_id = 5;
```

```
COMMIT TRAN
```

Transaction2

...

...

```
UPDATE accounts
SET current_balance = 30000
WHERE account_id = 5;
```

```
(1 rows affected)
```

REPEATABLE READ - summary

Pros:

- Prevents other transactions from modifying the data you are reading, non-repeatable reads
- Prevents dirty reads

Cons:

- Allows phantom reads
- You can be blocked by a `REPEATABLE READ` transaction.

When to use it?:

- Only want to read committed data and don't want other transactions to modify what you are reading. You don't care if phantom reads occur

Let's practice!

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER

SERIALIZABLE isolation level

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER

Miriam Antona
Software Engineer

datacamp

SERIALIZABLE

- Most restrictive isolation level

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

Isolation level comparison

	Dirty reads	Non-repeatable reads	Phantom reads
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	no	no	yes
SERIALIZABLE	**no**	**no**	**no**

Locking records with SERIALIZABLE

- Query with `WHERE` clause based on an index range -> Locks only that records
- Query not based on an index range -> Locks the complete table

SERIALIZABLE - query based on an index range

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRAN
    SELECT * FROM customers
    WHERE customer_id BETWEEN 1 AND 3;
```

```
| customer_id | first_name | last_name | ... | phone     |
|-------------|------------|-----------|-----|-----------|
| 1           | Dylan      | Smith     | ... | 555888999 |
```

Locked record

SERIALIZABLE - query based on an index range

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRAN
    SELECT * FROM customers
    WHERE customer_id BETWEEN 1 AND 3;
```

```
| customer_id | first_name | last_name | ... | phone     |
|-------------|------------|-----------|-----|-----------|
| 1           | Dylan      | Smith     | ... | 555888999 |
```

Locked record

Transaction 2

...

...

```
INSERT INTO customers (customer_id, first_name, ...)
VALUES (2, 'Phantom', 'Ph', 'phanton@mail.com', 555666222);
```

Has to wait!

SERIALIZABLE - query based on an index range

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRAN
    SELECT * FROM customers
    WHERE customer_id BETWEEN 1 AND 3;
```

```
    SELECT * FROM customers
    WHERE customer_id BETWEEN 1 AND 3;
```

```
| customer_id | first_name | last_name | ... | phone     |
|-------------|------------|-----------|-----|-----------|
| 1           | Dylan      | Smith     | ... | 555888999 |
```

Transaction 2

...

...

```
INSERT INTO customers (customer_id, first_name, ...)
VALUES (2, 'Phantom', 'Ph', 'phanton@mail.com', 555666222);
```

Has to wait!

SERIALIZABLE - query based on an index range

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRAN
    SELECT * FROM customers
    WHERE customer_id BETWEEN 1 AND 3;
```

```
    SELECT * FROM customers
    WHERE customer_id BETWEEN 1 AND 3;
```

```
COMMIT TRAN
```

Transaction 2

...

...

```
INSERT INTO customers (customer_id, first_name, ...)
VALUES (2, 'Phantom', 'Ph', 'phanton@mail.com', 555666222);
```

Finally executed!

SERIALIZABLE - query based on an index range

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRAN
    SELECT * FROM customers
    WHERE customer_id BETWEEN 1 AND 3;
```

```
| customer_id | first_name | last_name | ... | phone     |
|-------------|------------|-----------|-----|-----------|
| 1           | Dylan      | Smith     | ... | 555888999 |
```

Transaction 2

...

...

```
INSERT INTO customers (customer_id, first_name, ...)
VALUES (200, 'Phantom', 'Ph', 'phanton@mail.com', 555666222);
```

Instantly inserted!

SERIALIZABLE - query not based on an index range

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRAN
    SELECT * FROM customers;
```

```
| customer_id | first_name | last_name | ... | phone     |
|-------------|------------|-----------|-----|-----------|
| 1           | Dylan      | Smith     | ... | 555888999 |
...
| 10          | Carol      | York      | ... | 555148988 |
```

Locks the complete table

Transaction 2

...

...

```
INSERT INTO customers
VALUES (100, 'Phantom', 'Ph', 'phanton@mail.com', 555666222);
```

Has to wait!

SERIALIZABLE - query not based on an index range

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRAN
    SELECT * FROM customers;
```

```
    SELECT * FROM customers;
```

```
| customer_id | first_name | last_name | ... | phone     |
|-------------|------------|-----------| ... |-----------|
| 1           | Dylan      | Smith     | ... | 555888999 |
...
| 10          | Carol      | York      | ... | 555148988 |
```

Transaction 2

...

...

```
INSERT INTO customers
VALUES (100, 'Phantom', 'Ph', 'phanton@mail.com', 555666222);
```

Has to wait!

SERIALIZABLE - query not based on an index range

Transaction 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRAN
    SELECT * FROM customers;
```

```
    SELECT * FROM customers;
```

```
COMMIT TRAN
```

Transaction 2

...

...

```
INSERT INTO customers
VALUES (100, 'Phantom', 'Ph', 'phanton@mail.com', 555666222);
```

Finally executed!

SERIALIZABLE - summary

Pros:

- Good data consistency: Prevents dirty, non-repeatable and phantom reads

Cons:

- You can be blocked by a `SERIALIZABLE` transaction

When to use it?:

- When data consistency is a must

Let's practice!

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER

SNAPSHOT

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER

Miriam Antona
Software Engineer

datacamp

SNAPSHOT

- Every modification is stored in the `tempDB` database
- Only see committed changes that occurred before the start of the SNAPSHOT transaction and own changes
- Can't see any changes made by other transactions after the start of the SNAPSHOT transaction
- Readings don't block writings and writings don't block readings
- Can have update conflicts

```
ALTER DATABASE myDatabaseName SET ALLOW_SNAPSHOT_ISOLATION ON;
```

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

SNAPSHOT - isolation level comparison

	Dirty reads	Non-repeatable reads	Phantom reads
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	no	no	yes
SERIALIZABLE	no	no	no
SNAPSHOT	**no**	**no**	**no**

SNAPSHOT - example

Transaction1

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT

BEGIN TRAN
    SELECT * FROM accounts;
```

```
|account_id|account_number      |...| current_balance|
|----------|--------------------|---|----------------|
|1         |55555555551234567890|...| 25000,00       |
|2         |55555555559876543210|...| 200,00         |
|...       |...                 |...| ...            |
|15        |55555555551234567890|...| 25000,00       |
```

Transaction2

...

...

```
BEGIN TRAN
    INSERT INTO accounts
    VALUES (11111111111111111111, 1, 25000);

    UPDATE accounts
    SET current_balance = 30000 WHERE account_id = 1;

    SELECT * FROM accounts;
COMMIT TRAN
```

It is not blocked!

SNAPSHOT - example

Transaction1

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT

BEGIN TRAN
    SELECT * FROM accounts;
```

```
    SELECT * FROM accounts;
```

```
|account_id|account_number      |...| current_balance|
|----------|--------------------|---|----------------|
|1         |55555555551234567890|...| 25000,00       |
|2         |55555555559876543210|...| 200,00         |
|...       |...                 |...| ...            |
|15        |55555555551234567890|...| 25000,00       |
```

Transaction2

...

...

```
BEGIN TRAN
    INSERT INTO accounts
    VALUES (11111111111111111111, 1, 25000);

    UPDATE accounts
    SET current_balance = 30000 WHERE account_id = 1;

    SELECT * FROM accounts;
COMMIT TRAN
```

It is not blocked!

SNAPSHOT - summary

Pros:

- Good data consistency: Prevents dirty, non-repeatable and phantom reads without blocking

Cons:

- `tempDB` increases

When to use it?:

- When data consistency is a must and don't want blocks

READ COMMITTED SNAPSHOT

- Changes the behavior of READ COMMITTED

```
ALTER DATABASE myDatabaseName SET READ_COMMITTED_SNAPSHOT {ON|OFF};
```

- OFF by default
- To use ON:

```
ALTER DATABASE myDatabaseName SET ALLOW_SNAPSHOT_ISOLATION ON;
```

- Set to ON, makes every READ COMMITTED statement can only see committed changes that occurred before the start of that statement
- Can't have update conflicts

READ COMMITTED SNAPSHOT - example

Transaction1

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
  UPDATE accounts
  SET current_balance = 30000
  WHERE account_id = 1;
```

```
COMMIT TRAN
```

Transaction2

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
```

```
    SELECT current_balance FROM accounts
    WHERE account_id = 1;
```

```
| current_balance |
|-----------------|
| 35000,00        |
```

READ COMMITTED SNAPSHOT - example

Transaction1

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
  UPDATE accounts
  SET current_balance = 30000
  WHERE account_id = 1;
```

```
COMMIT TRAN
```

Transaction2

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
```

```
    SELECT current_balance FROM accounts
    WHERE account_id = 1;
```

```
    SELECT current_balance FROM accounts
    WHERE account_id = 1;
```

```
| current_balance |
|-----------------|
| 30000,00        |
```

WITH (NOLOCK)

- Used to read uncommitted data
- `READ UNCOMMITTED` applies to the entire connection / `WITH (NOLOCK)` applies to a specific table
- Use under any isolation level when you just want to read uncommitted data from specific tables

WITH (NOLOCK) - example

Original balance account 5 = $35,000

Transaction1

```
BEGIN TRAN
  UPDATE accounts
  SET current_balance = 30000
  WHERE account_id = 5;
```

Transaction2

...

...

...

```
SELECT current_balance
FROM accounts WITH (NOLOCK)
WHERE account_id = 5;
```

```
| current_balance |
|-----------------|
| 30000,00        |
```

Let's practice!

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER

Congratulations!

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER

Miriam Antona
Software Engineer

datacamp

Chapters 1 and 2 - Error handling

- `TRY...CATCH` construct
- Error anatomy
- Uncatchable errors by a `CATCH` block
- Error functions: `ERROR_NUMBER()`, `ERROR_SEVERITY()`, `ERROR_STATE()`, `ERROR_LINE()`, `ERROR_PROCEDURE()`, `ERROR_MESSAGE()`
- `RAISERROR`
- `THROW`

Chapter 3 - Transactions

- What is a transaction?
- Transaction statements:
  - `BEGIN TRAN`
  - `COMMIT TRAN`
  - `ROLLBACK TRAN`
- `@@TRANCOUNT`
- Savepoints
- `XACT_ABORT`
- `XACT_STATE`

Chapter 4 - transaction isolation levels

- What is concurrency?
- Isolation levels:
  - `READ UNCOMMITTED`
  - `READ COMMITTED`
  - `REPEATABLE READ`
  - `SERIALIZABLE`
  - `SNAPSHOT`
- Concurrency phenomena: dirty reads, non-repeatable reads, phantom reads

Thank you!

TRANSACTIONS AND ERROR HANDLING IN SQL SERVER