Background to T-SQL Querying and Programming

Transactions and Concurrency

Transactions and Concurrency

Transactions

A transaction is a unit of work that might include multiple activities that query and modify data and that can also change data definition.

Define a transaction explicitly with a :

1. The start syntax BEGIN TRAN (or BEGIN TRANSACTION) statement.
2. The end syntax with either
   i. COMMIT TRAN statement to confirm it
   ii. ROLLBACK TRAN (or ROLLBACK TRANSACTION) statement if you do not want to confirm it (that is, if you want to undo its changes).

Transactions and Concurrency

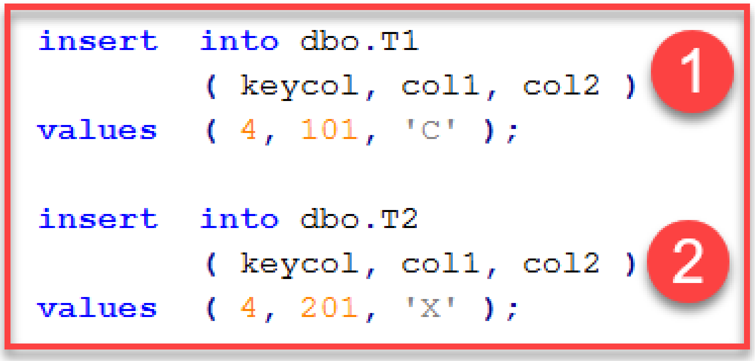
Transactions example

1. Explicitly starting a transaction (begin tran;)

```
begin tran;
```

2. Insert first row (1)

3. Insert second row (2)



```
insert  into dbo.T1
        ( keycol, col1, col2 )
values  ( 4, 101, 'C' );

insert  into dbo.T2
        ( keycol, col1, col2 )
values  ( 4, 201, 'X' );
```

```
commit TRAN;
```

4. Explicitly ending the transaction (commit tran;)

The explicit use of a transaction allows the insertion of the both rows consistently as a logical unit of work. If there were any issues then it could be rolled back.

Transactions and Concurrency

Transactions (acronym ACID)

Transactions have four properties

1. Atomicity
2. Consistency
3. Isolation
4. Durability

Transactions and Concurrency

Transactions (acronym ACID)

Atomicity

- A transaction is an atomic unit of work.
- Either all changes in the transaction take place or none do.

Consistency

- The term consistency refers to the state of the data that the RDBMS gives you access to as concurrent transactions modify and query it.

Transactions and Concurrency

Transactions (acronym ACID)

Isolation

- Isolation is a mechanism used to control access to data and ensure that transactions access data only if the data is in the level of consistency that those transactions expect.

Durability

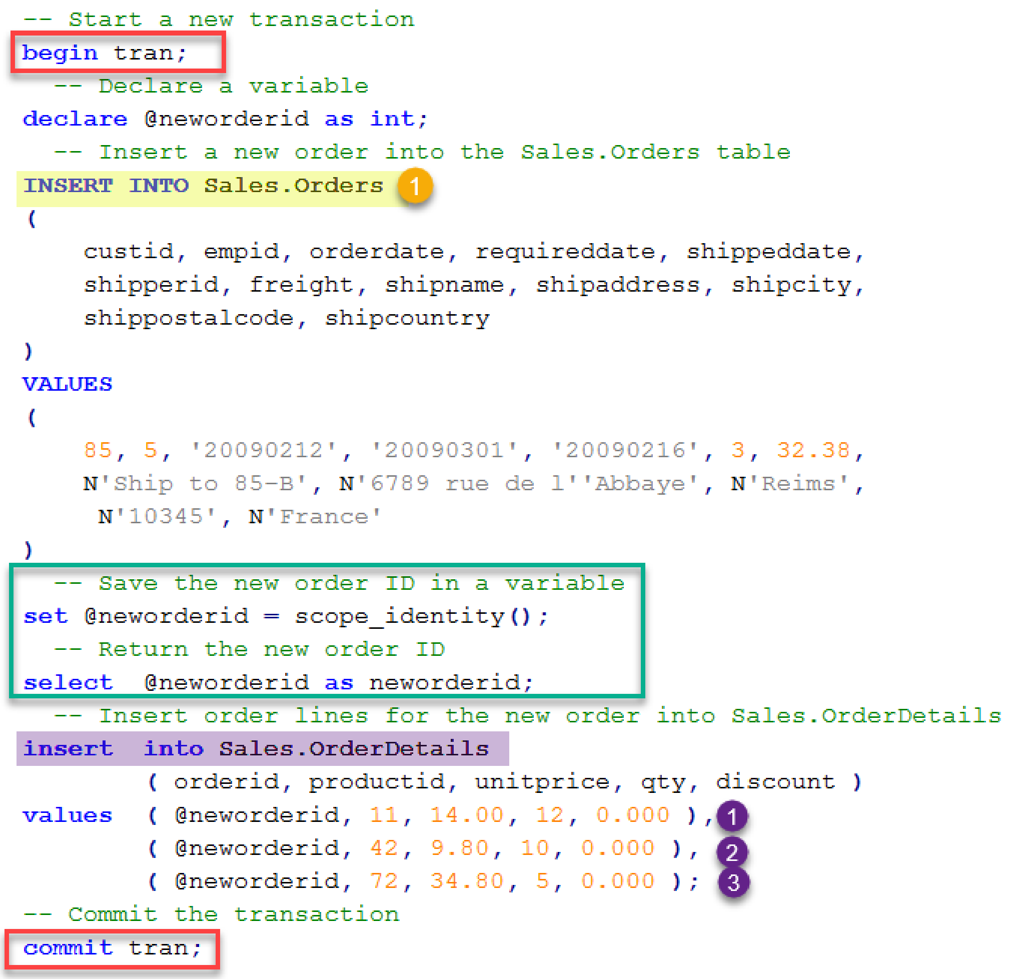
- Data changes are always written to the database's transaction log on disk before they are written to the data portion of the database on disk.
- After the commit instruction is recorded in the transaction log on disk, the transaction is considered durable even if the change hasn't yet made it to the data portion on disk.

Transactions and Concurrency

Transactions and Concurrnency

The logical unit of work here is the a sales order with the details of the purchase.

Look at all of the moving parts to get this written to the database



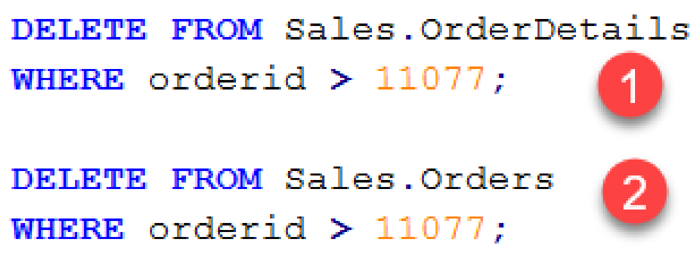
```
-- Start a new transaction
begin tran;
  -- Declare a variable
declare @neworderid as int;
  -- Insert a new order into the Sales.Orders table
INSERT INTO Sales.Orders
(
    custid, empid, orderdate, requireddate, shippeddate,
    shipperid, freight, shipname, shipaddress, shipcity,
    shippostalcode, shipcountry
)
VALUES
(
    85, 5, '20090212', '20090301', '20090216', 3, 32.38,
    N'Ship to 85-B', N'6789 rue de l''Abbaye', N'Reims',
     N'10345', N'France'
)
  -- Save the new order ID in a variable
set @neworderid = scope_identity();
  -- Return the new order ID
select  @neworderid as neworderid;
  -- Insert order lines for the new order into Sales.OrderDetails
insert  into Sales.OrderDetails
        ( orderid, productid, unitprice, qty, discount )
values  ( @neworderid, 11, 14.00, 12, 0.000 ),
        ( @neworderid, 42, 9.80, 10, 0.000 ),
        ( @neworderid, 72, 34.80, 5, 0.000 );
-- Commit the transaction
commit tran;
```

Transactions and Concurrency

Transactions

If you want to remove an orderid from the Sales.Order and Sales.OrderDetails tables.

- It is important to remember that there is a specific order for deletion due to foreign key relationships (referential integrity). It is always bottom up (child to parent)

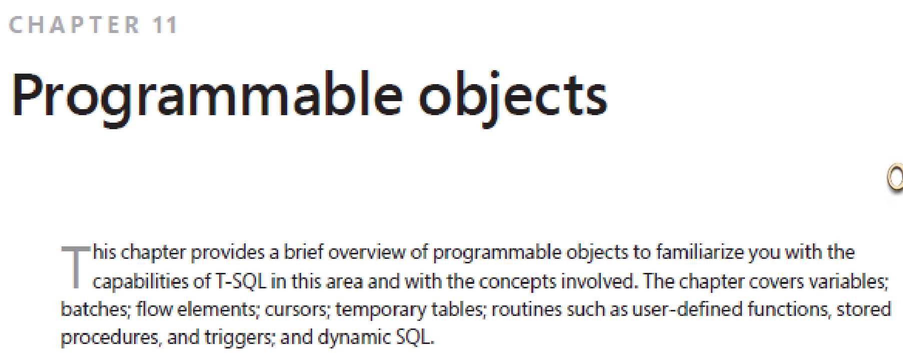


```
DELETE FROM Sales.OrderDetails
WHERE orderid > 11077;

DELETE FROM Sales.Orders
WHERE orderid > 11077;
```

Transactions and Concurrency

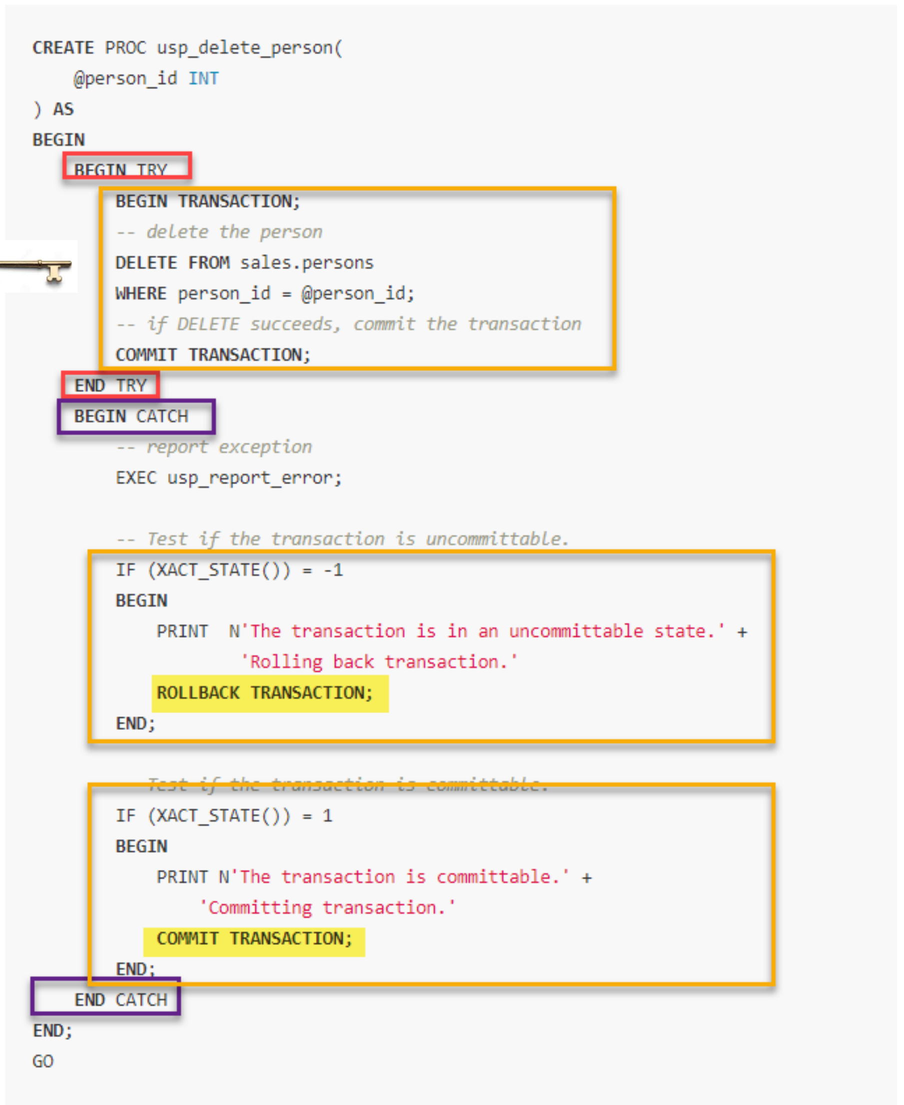
Transactions with Try Catch



CHAPTER 11

Programmable objects

This chapter provides a brief overview of programmable objects to familiarize you with the capabilities of T-SQL in this area and with the concepts involved. The chapter covers variables; batches; flow elements; cursors; temporary tables; routines such as user-defined functions, stored procedures, and triggers; and dynamic SQL.



```
CREATE PROC usp_delete_person(
    @person_id INT
) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;
        -- delete the person
        DELETE FROM sales.persons
        WHERE person_id = @person_id;
        -- if DELETE succeeds, commit the transaction
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        -- report exception
        EXEC usp_report_error;

        -- Test if the transaction is uncommittable.
        IF (XACT_STATE()) = -1
        BEGIN
            PRINT  N'The transaction is in an uncommittable state.' +
                    'Rolling back transaction.'
            ROLLBACK TRANSACTION;
        END;

        -- Test if the transaction is committable.
        IF (XACT_STATE()) = 1
        BEGIN
            PRINT N'The transaction is committable.' +
                'Committing transaction.'
            COMMIT TRANSACTION;
        END;
    END CATCH
END;
GO
```

Transactions and Concurrency

Locks and Blocking

SQL Server uses **locks** to enforce the isolation property of transactions.

Lock Modes and Compatibility

As you focus on learning about transactions and concurrency, be aware of the two main lock modes:

1. **Exclusive** - Exclusive locks are called "exclusive" because you cannot obtain an exclusive lock on a resource if another transaction is holding any lock mode on the resource, and no lock mode can be obtained on a resource if another transaction is holding an exclusive lock on the resource
2. **Shared** - This lock mode is called "shared" because multiple transactions can hold shared locks on the same data resource simultaneously. Although you cannot change the lock mode and duration required when you are modifying data, you can control the way locking is handled when you are reading data by changing your isolation level.

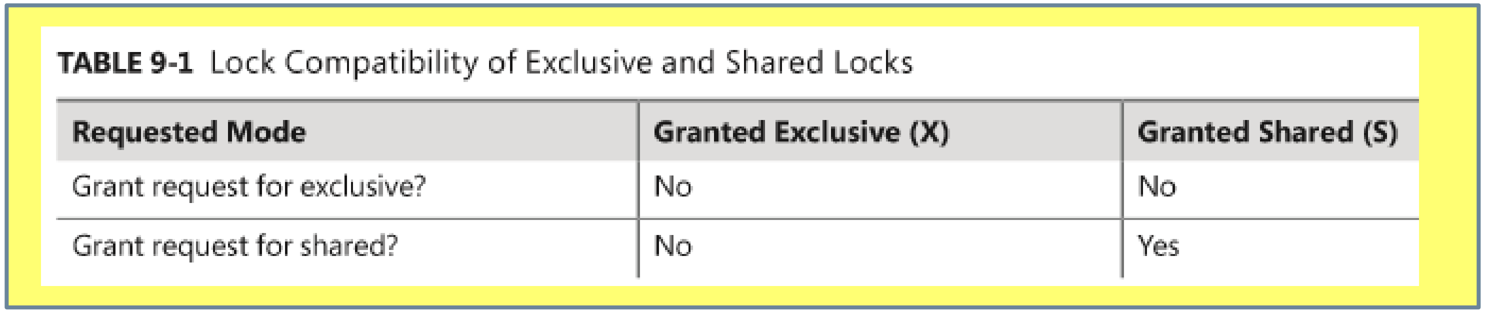


TABLE 9-1 Lock Compatibility of Exclusive and Shared Locks

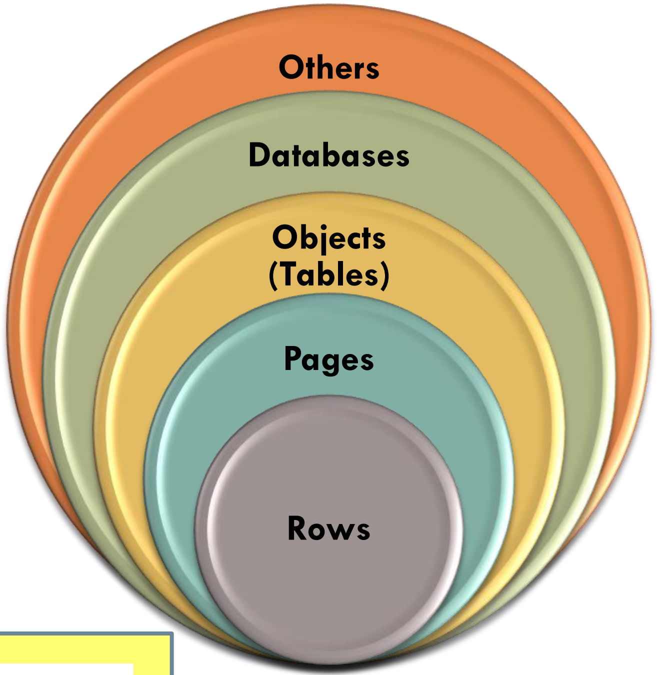
Requested Mode	Granted Exclusive (X)	Granted Shared (S)
Grant request for exclusive?	No	No
Grant request for shared?	No	Yes

Transactions and Concurrency

Lockable resource Types

SQL Server can lock different types of resources. The types of resources that can be locked include

1. RIDs or keys (row)
   acquire fine-grained locks (such as row or Pages)
2. objects (for example, tables)
   escalate the fine-grained locks to more coarse-grained locks (such as table locks)
3. Databases
4. others.



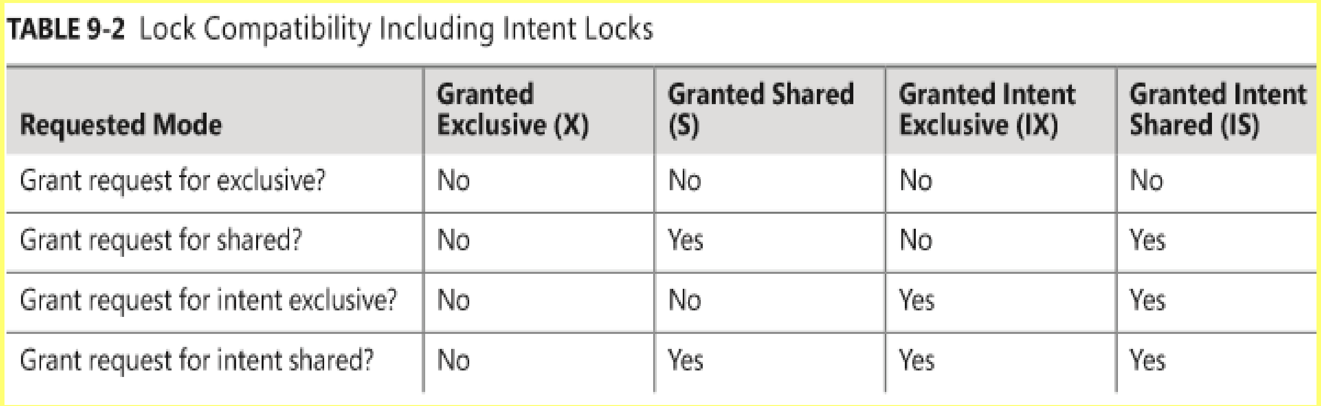


TABLE 9-2 Lock Compatibility Including Intent Locks

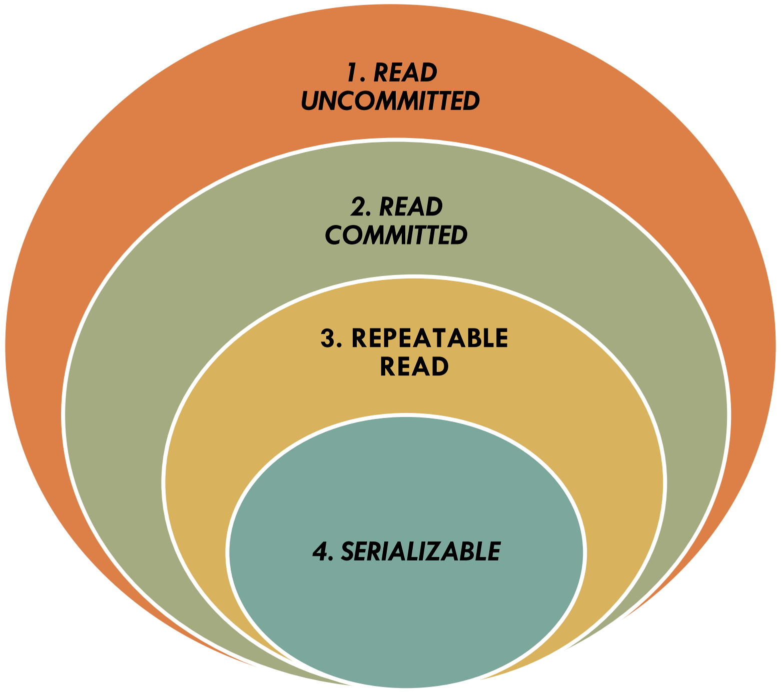
Requested Mode	Granted Exclusive (X)	Granted Shared (S)	Granted Intent Exclusive (IX)	Granted Intent Shared (IS)
Grant request for exclusive?	No	No	No	No
Grant request for shared?	No	Yes	No	Yes
Grant request for intent exclusive?	No	No	Yes	Yes
Grant request for intent shared?	No	Yes	Yes	Yes

Transactions and Concurrency

Isolation Levels

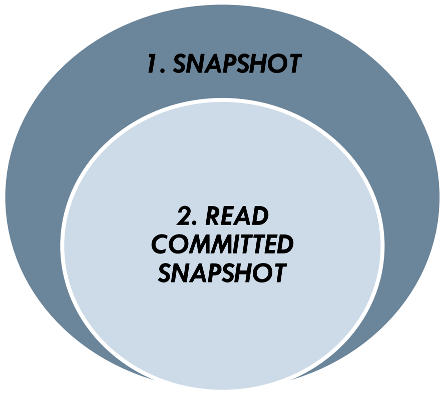
Four traditional isolation levels based on pessimistic concurrency control (locking):

1. *READ UNCOMMITTED*
2. ***READ COMMITTED* (the default in on-premises SQL Server instances)**
3. *REPEATABLE READ*
4. *SERIALIZABLE.*



Two isolation levels based on optimistic concurrency control (row versioning):

1. *SNAPSHOT*
2. *READ COMMITTED SNAPSHOT* (the default in SQL Database).



Transactions and Concurrency

Isolation Levels

You can set **the isolation level of the whole session** by using the following command.

SET TRANSACTION ISOLATION LEVEL <isolation name>;

You can use a table hint to set **the isolation level of a query**.

SELECT ... FROM <table> WITH (<isolationname>);

Transactions and Concurrency

1. READ UNCOMMITTED Isolation solves “Dirty Reads”

Dirty Reads Video

- ✓ READ UNCOMMITTED is the lowest available isolation level.
- ✓ In this isolation level, a reader doesn’t ask for a shared lock.
- ✓ A reader that doesn’t ask for a shared lock can never be in conflict with a writer that is holding an exclusive lock.
- ✓ **This means that the reader can read uncommitted changes (also known as dirty reads).** It also means that the reader won’t interfere with a writer that asks for an exclusive lock.
- ✓ In other words, **a writer can change data** while a reader that is running under the READ UNCOMMITTED isolation level reads data.

Transactions and Concurrency

2. READ COMMITTED Isolation Level solves "No Dirty Reads"

- ✓ READ COMMITTED isolation level, you don't get dirty reads. Instead, **you can only read committed changes.**
- ✓ In terms of the duration of locks, in the READ COMMITTED isolation level, a reader only holds the shared lock until it is done with the resource.
- ✓ It doesn't keep the lock until the end of the transaction; in fact, it doesn't even keep the lock until the end of the statement. This means that in between two reads of the same data resource in the same transaction, no lock is held on the resource.
- ✓ Therefore, another transaction can modify the resource in between those two reads, and the reader might get different values in each read.
- ✓ This phenomenon is called non-repeatable reads or inconsistent analysis.

Transactions and Concurrency

3. REPEATABLE READ Isolation solves "lost update"

- ✓ Another phenomenon prevented by REPEATABLE READ but not by lower isolation levels is called a lost update ("**Hidden Update**").
- ✓ A lost update happens when two transactions read a value, make calculations based on what they read, and then update the value.
- ✓ Because in isolation levels lower than REPEATABLE READ no lock is held on the resource after the read, both transactions can update the value, and whichever transaction updates the value last "wins," overwriting the other transaction's update.
- ✓ In REPEATABLE READ, both sides keep their shared locks after the first read, so neither can acquire an exclusive lock later in order to update. The situation results in a deadlock, and the update conflict is prevented. I'll provide more details on deadlocks later in this chapter, in the "Deadlocks" section

Transactions and Concurrency

4. SERIALIZABLE Isolation solves "phantom reads"

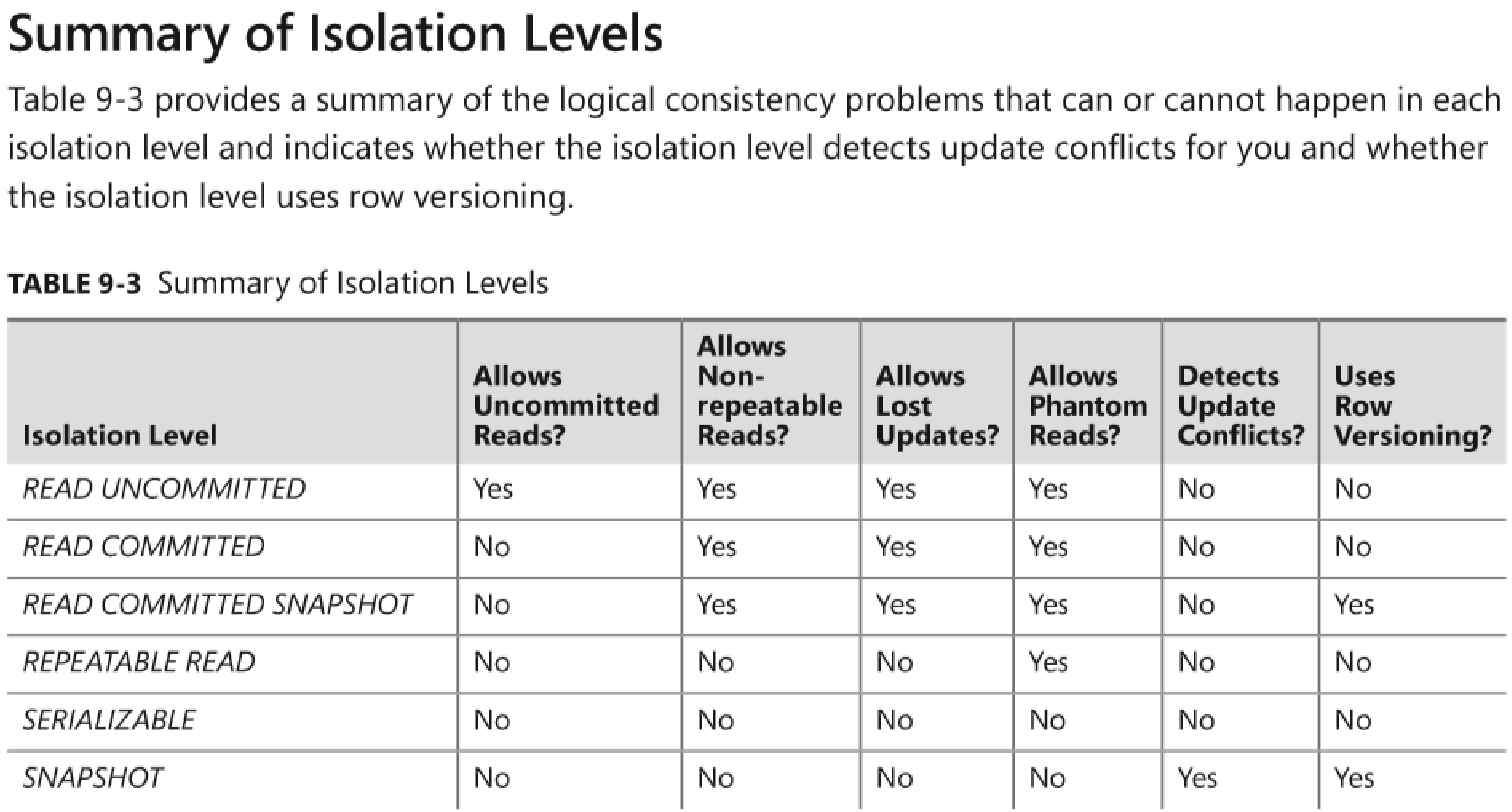
The SERIALIZABLE isolation level behaves similarly to REPEATABLE READ:

- ✓ Namely, it requires a reader to obtain a shared lock to be able to read, and keeps the lock until the end of the transaction.

✓ SERIALIZABLE isolation level adds another facet—logically,

- ✓ This isolation level causes a reader to lock the whole range of keys that qualify for the query's filter.
- ✓ This means that the reader locks not only the existing rows that qualify for the query's filter, but also future ones.
- ✓ Or, more accurately, it blocks attempts made by other transactions to add rows that qualify for the reader's query filter.

Transactions and Concurrency



Summary of Isolation Levels

Table 9-3 provides a summary of the logical consistency problems that can or cannot happen in each isolation level and indicates whether the isolation level detects update conflicts for you and whether the isolation level uses row versioning.

TABLE 9-3 Summary of Isolation Levels

Isolation Level	Allows Uncommitted Reads?	Allows Non-repeatable Reads?	Allows Lost Updates?	Allows Phantom Reads?	Detects Update Conflicts?	Uses Row Versioning?
READ UNCOMMITTED	Yes	Yes	Yes	Yes	No	No
READ COMMITTED	No	Yes	Yes	Yes	No	No
READ COMMITTED SNAPSHOT	No	Yes	Yes	Yes	No	Yes
REPEATABLE READ	No	No	No	Yes	No	No
SERIALIZABLE	No	No	No	No	No	No
SNAPSHOT	No	No	No	No	Yes	Yes

Transactions and Concurrency

Blocking v/s Deadlocking ("**Deadly Embrace**")

Blocking : Occurs if a transaction tries to acquire an incompatible lock on a resource that another transaction has already locked. The blocked transaction remains blocked until the blocking transaction releases the lock.

Example: Blocking Scenario in SQL Server

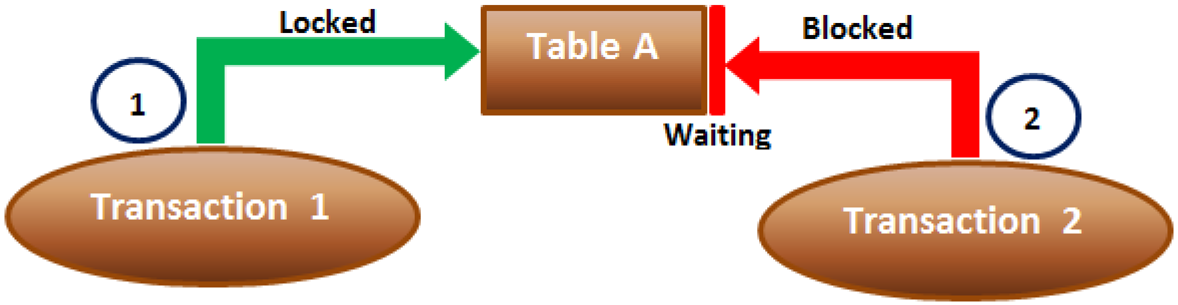


Table A	
Id	**Name**
1	Mark

Table B	
Id	**Name**
1	Mary

```
--Transaction 1
Begin Tran
Update TableA set
Name='Mark Transaction 1' where Id = 1
Waitfor Delay '00:00:10'
Commit Transaction
```

```
--Transaction 2
Begin Tran
Update TableA set
Name='Mark Transaction 2' where Id = 1
Commit Transaction
```

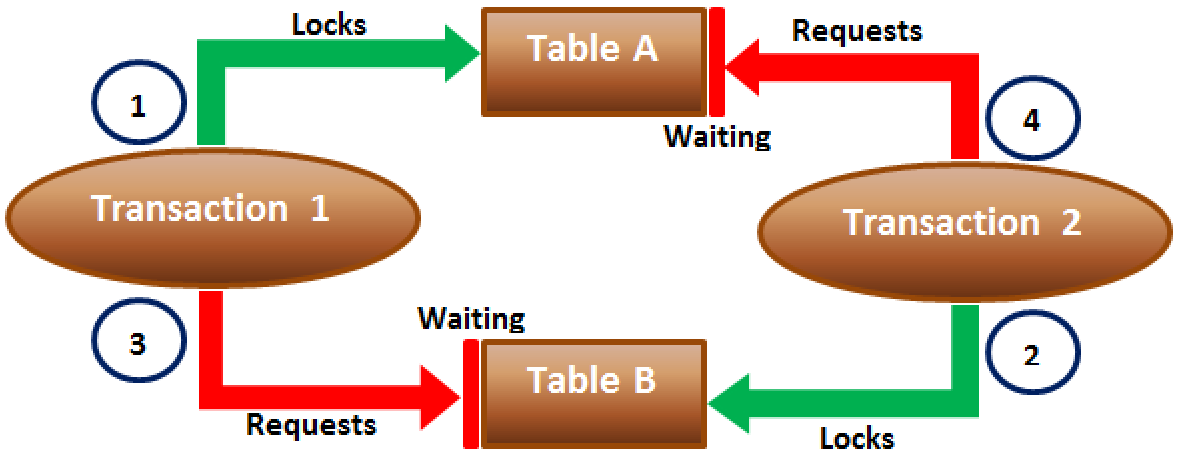
Transactions and Concurrency

Blocking v/s Deadlocking("**Deadly Embrace**")

Deadlock : Occurs when two or more transactions have a resource locked, and each transaction requests a lock on the resource that another transaction has already locked. Neither of the transactions here can move forward, as each one is waiting for the other to release the lock.

In this case, SQL Server intervenes and ends the deadlock by cancelling one of the transactions, so the other transaction can move forward.

Dead Lock Scenario in SQL Server



Transactions and Concurrency

Deadlock ("Deadly Embrace") Example

```
-- Transaction 1
Begin Tran
Update TableA Set Name = 'Mark Transaction 1' where Id = 1

-- From Transaction 2 window execute the first update statement

Update TableB Set Name = 'Mary Transaction 1' where Id = 1

-- From Transaction 2 window execute the second update statement
Commit Transaction


-- Transaction 2
Begin Tran
Update TableB Set Name = 'Mark Transaction 2' where Id = 1

-- From Transaction 1 window execute the second update statement

Update TableA Set Name = 'Mary Transaction 2' where Id = 1

-- After a few seconds notice that one of the transactions complete
-- successfully while the other transaction is made the deadlock victim
Commit Transaction
```