Database Systems(2)

Tutorial 6

DR. ALYAA HAMZA

ENG. ESRAA SHEHAB

1. Concurrency problems – theory and experimentation in SQL Server

In any relational database system, there is the concept of **transaction**.
A transaction is a set of logical operations that must be performed in a user session as a single piece of work. Let’s review the properties of a transaction:

Hence, a transaction must be **atomic** i.e. there is no halfway for it to complete: either all the logical operations occur or none of them occur.

A transaction must be **consistent** i.e. any data written using database system must be valid according to defined rules like primary key uniqueness or foreign key constraints.

Once the **consistency** is ensured, a transaction must be permanently stored to disk.

It guarantees the **durability** required for a transaction.

Finally, as multiple transactions could be running concurrently in a database system, we can find transactions that read from or writes to the same data object (row, table, index…).

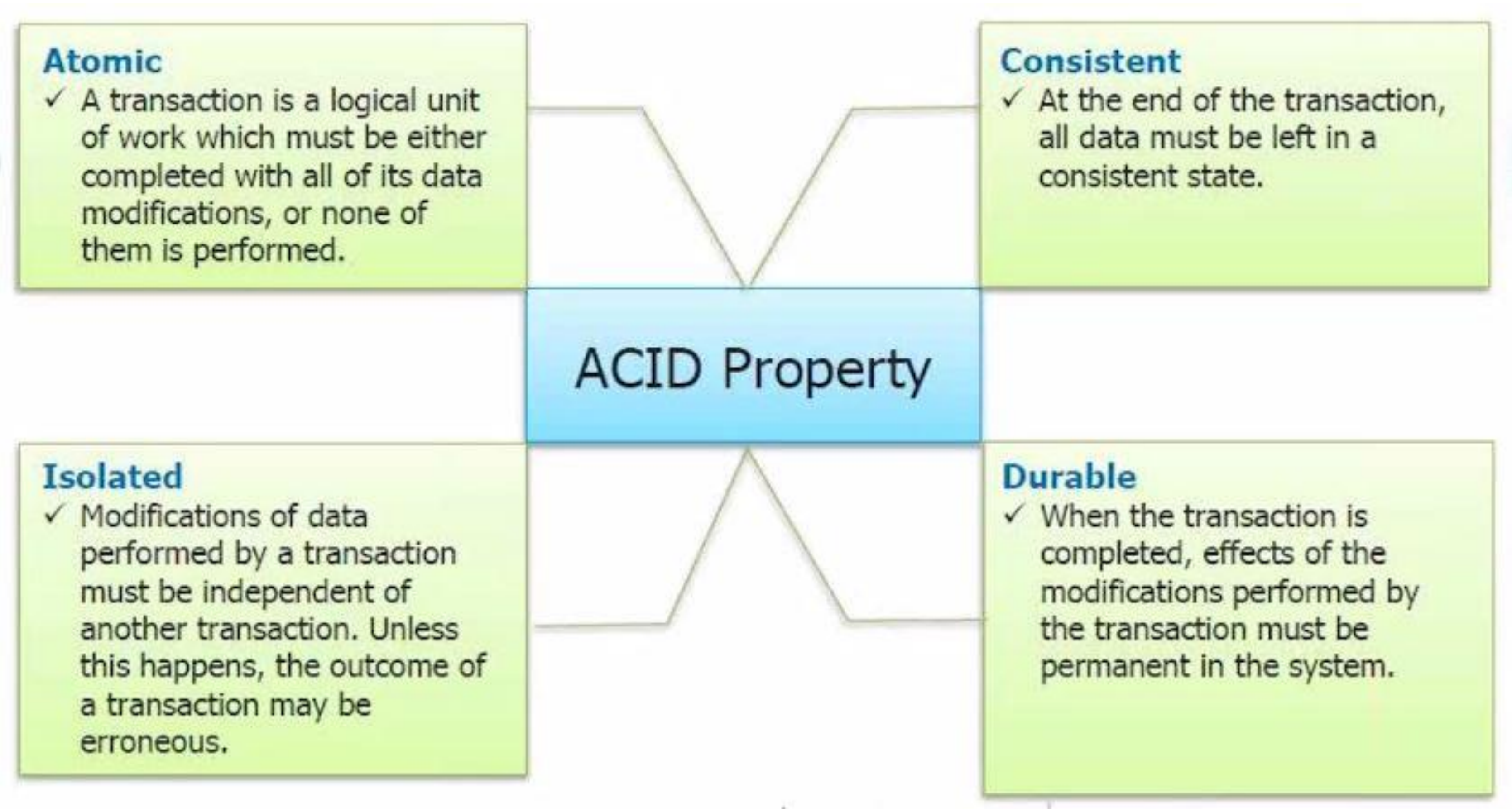
It introduces a set of problems to which different « transaction isolation (level) » tend to respond. A transaction isolation (level) defines how and when the database system will present changes made by any transaction to other user sessions.

To select the appropriate transaction **isolation** level, having a good understanding on common concurrency problems that can occur is mandatory.

This section divided into two parts:

1. The first one will explain the concurrency problems with a theoretical example
2. While the second will be more practical, and we will try to experience these problems on a SQL Server instance.

2. Transaction properties:



ACID Property

Atomic

- ✓ A transaction is a logical unit of work which must be either completed with all of its data modifications, or none of them is performed.

Consistent

- ✓ At the end of the transaction, all data must be left in a consistent state.

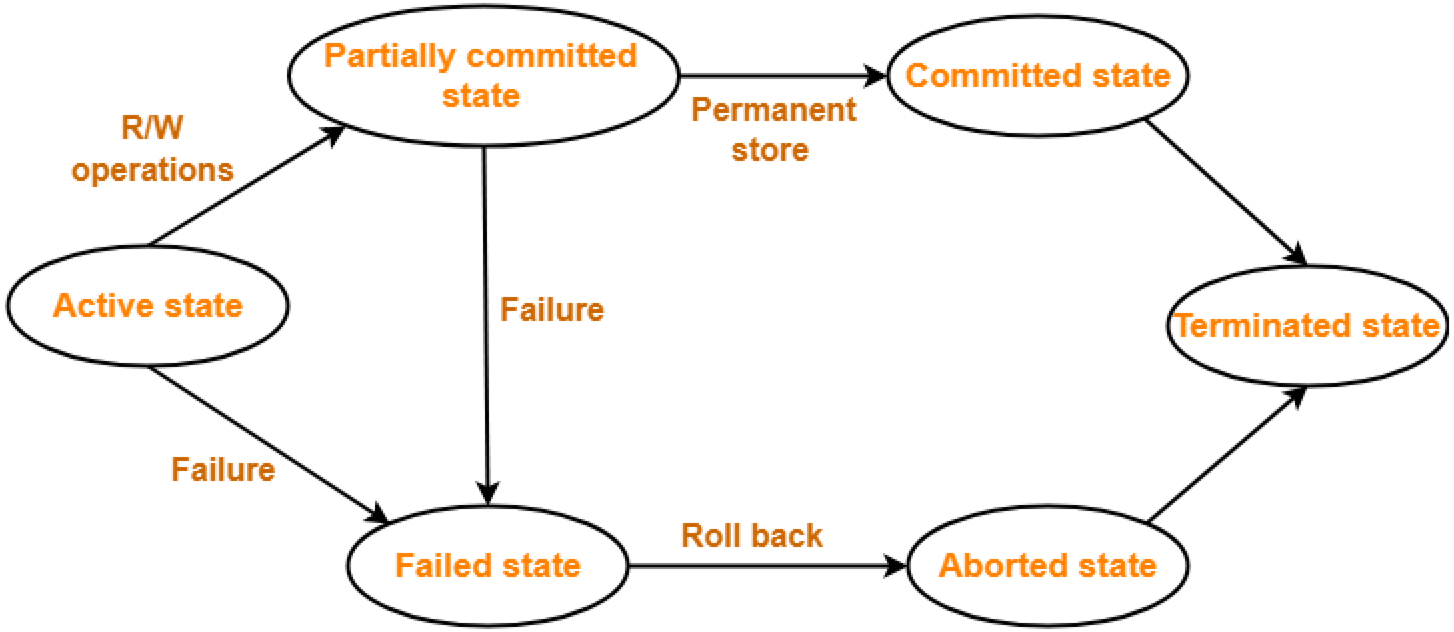
Isolated

- ✓ Modifications of data performed by a transaction must be independent of another transaction. Unless this happens, the outcome of a transaction may be erroneous.

Durable

- ✓ When the transaction is completed, effects of the modifications performed by the transaction must be permanent in the system.

3. Transaction states:



Transaction States in DBMS

1. Active State:

- This is the first state in the life cycle of a transaction.
- A transaction is called in an **active state** if its instructions are getting executed.
- All the changes made by the transaction now are stored in the buffer in main memory.

2. Partially Committed State:

- After the last instruction of transaction has executed, it enters a partially committed state.
- After entering this state, the transaction is partially committed.
- It is not considered fully committed because all the changes made by the transaction are still stored in the buffer in main memory.

3. Committed State:

- After all the changes made by the transaction have been successfully stored into the database, it enters into a committed state.
- Now, the transaction is considered to be fully committed.

NOTE:

- After a transaction has entered the committed state, it is not possible to roll back the transaction.
- In other words, it is not possible to undo the changes that has been made by the transaction.
- This is because the system is updated into a new consistent state.
- The only way to undo the changes is by carrying out another transaction called as **compensating transaction** that performs the reverse operations.

4. Failed State:

- When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it becomes impossible to continue the execution, it enters into a **failed state**.

5. Aborted State:

- After the transaction has failed and entered a failed state, all the changes made by it must be undone.
- To undo the changes made by the transaction, it becomes necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters an **aborted state**.

6. Terminated State:

- This is the last state in the life cycle of a transaction.
- After entering the committed state or aborted state, the transaction finally enters into a **terminated state** where its life cycle finally comes to an end.

Concurrency problems:

1. Lost update and dirty write:

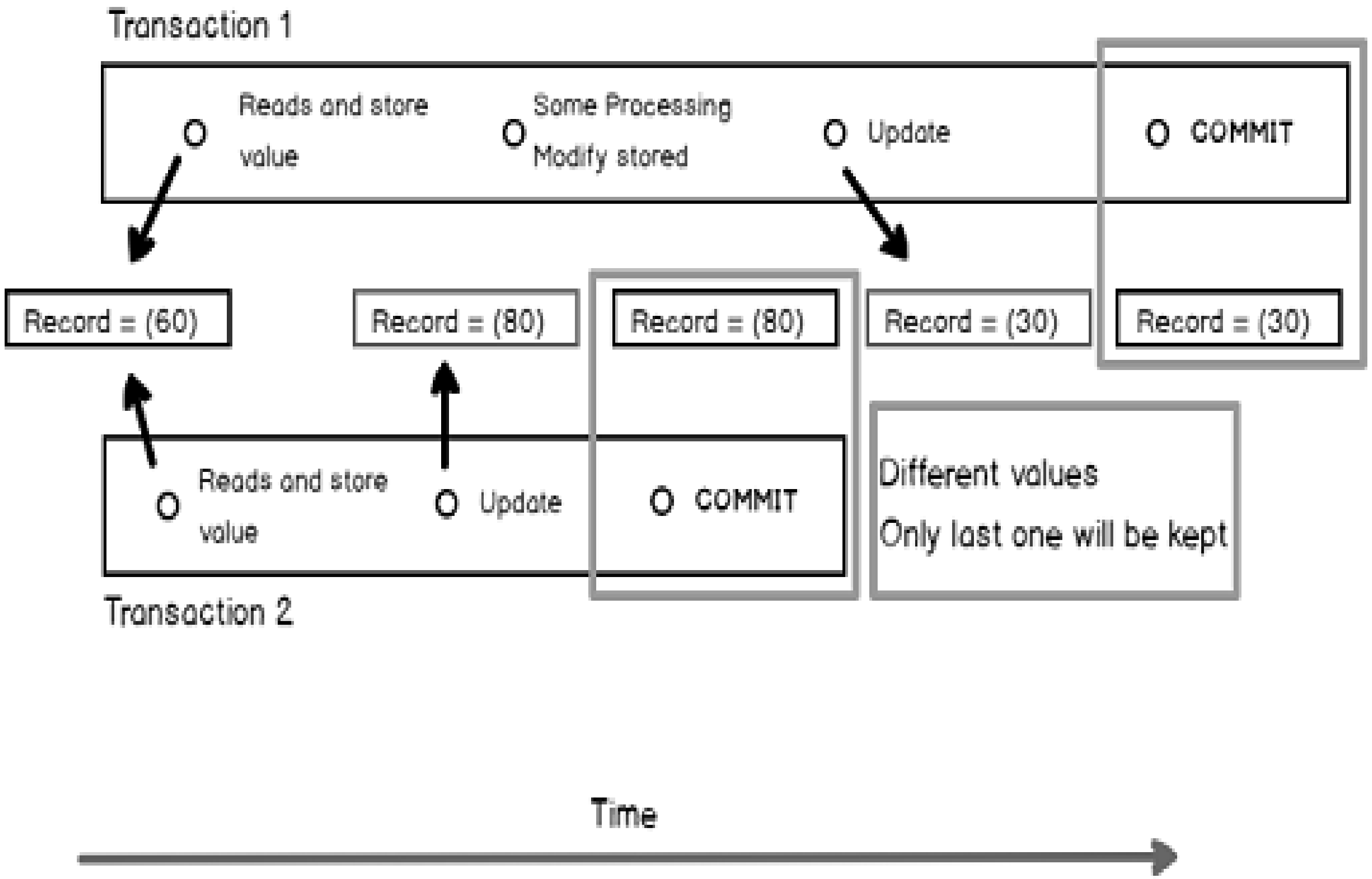
This phenomenon **happens when two transactions access the same record, and both updates this record.**

The following figure summarizes what could happen in a simple example.

In this example, we have 2 concurrent transactions that access a record with a **(60)** modifiable value. This record is identified either by its **rowId** or by a primary key column that won't be presented here for simplicity.

The first transaction reads this record, does some processing then updates this record and finally commits its work. The second transaction reads the record then updates it immediately and commits. Both transactions do not update this record to the same value. This leads to a loss for the update statement performed by second transaction.

As **Transaction 1** overwrites a value that **Transaction 2** already modified. We could have said that **Transaction 1** did a **« dirty write »** if **Transaction 2** didn't commit its work.



Transaction 1

Reads and store value

Some Processing Modify stored

Update

COMMIT

Record = (60)

Record = (80)

Record = (80)

Record = (30)

Record = (30)

Reads and store value

Update

COMMIT

Transaction 2

Different values

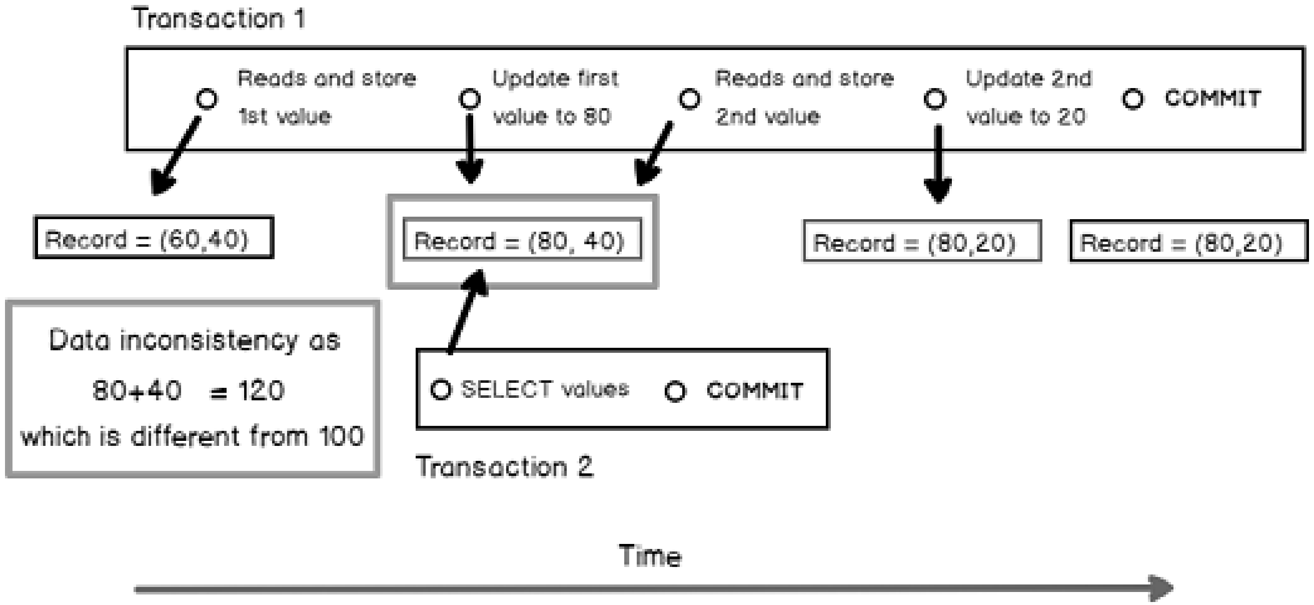
Only last one will be kept

Time

2. Dirty read

A dirty read happens when a transaction accesses a data that has been modified by another transaction, but this change has not been committed or rolled back yet. Following figure shows a case of dirty read.

In this example, record has two columns with a starting value of (60,40). In this context, let’s say we have an applicative constraint that says that the sum of those values must always be 100.

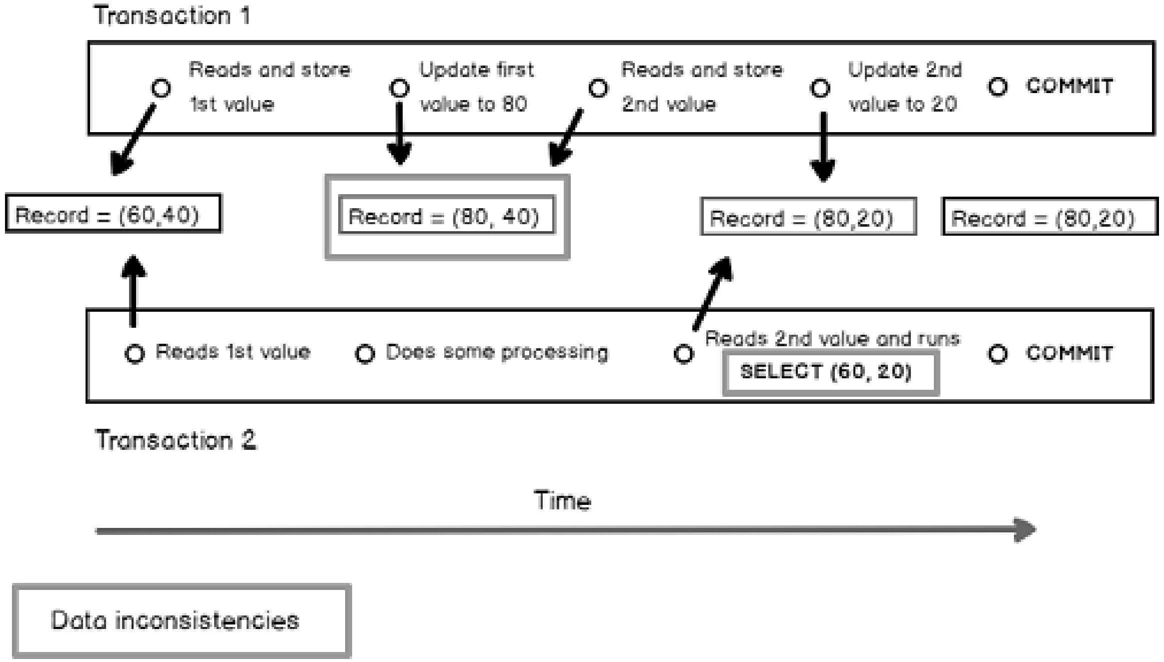


3. Non-repeatable read or fuzzy read

The situation of non-repeatable read is almost the **same as dirty read except that both values are modified**.

As in previous sub-sections, we will review a graphical representation of a non-repeatable read situation. To do so, we will keep two concurrent transactions accessing two columns of the same record.

One of them reads and modifies each value, one at a time, then commits while the other reads the first value, does some processing, reads the second value then commits. Keeping our constraint from previous example (the sum of both values equals 100), the presented situation leads the second transaction to manipulate inconsistent data and maybe to present it to an end-user.

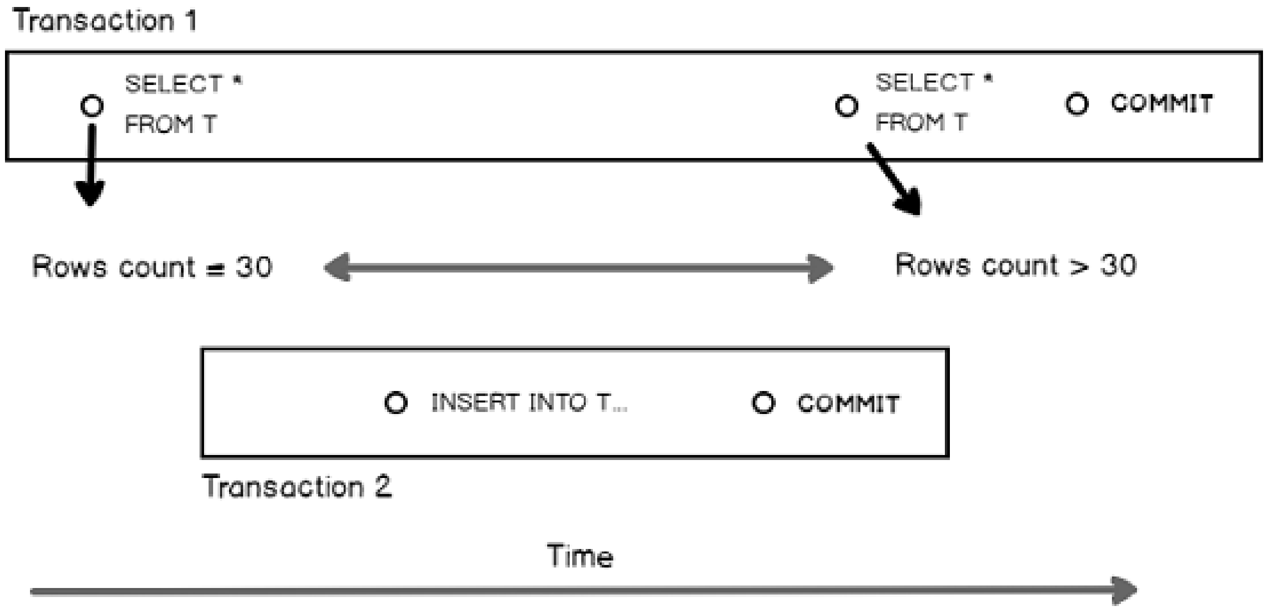


4. Phantom reads

Phantom reads are a **variation of non-repeatable reads in the context of row sets.** Here is an example that illustrates this:

Let’s say we have a transaction **Transaction 1** that performs twice a **SELECT** query against a table **T**, once at its beginning and once just before its end. Let’s assume another transaction **Transaction 2** starts after the first one, inserts a new row to the table **T** and commits before the second time **Transaction 1** will run its **SELECT** query. The result sets that will be returned by the two occurrences of the SELECT query will differ.

Here is a diagram that summarizes the situation:



5. Locking reads

This is not really a concurrency problem, but more likely a “design pattern”. In short, the principle is to read a value from a given record and update this record based on the returned value inside the same transaction, with the insurance that no other session will modify the value that has just been read.

It’s the concept of **SELECT FOR UPDATE** in Oracle or **SELECT … FROM <table> WITH (UPDLOCK)** in SQL Server.

Experimentation on SQL Server

1. Lost update

Following queries depict the following scenario.

We are in a bank system. Your bank account has an initial balance of 1500 (currency does not matter). You will find below the T-SQL statement to set up the test.

```
CREATE TABLE BankAccounts(
    AccountIdINT IDENTITY(1,1),
    BalanceAmount   INT
);

insert into BankAccounts (BalanceAmount)
VALUES (1500);
```

```
-- Session 1: Employer
DECLARE @CustomerBalanceINT ;
DECLARE @BalanceDifferenceINT ;

SET @BalanceDifference = 1600 ;

BEGIN TRANSACTION ;
-- Getting back current balance value
SELECT @CustomerBalance = BalanceAmount
FROM BankAccounts
WHERE AccountId = 1;
PRINT 'Read Balance value: ' +
CONVERT(VARCHAR(32),@CustomerBalance);

-- adding salary amount
SET @CustomerBalance = @CustomerBalance +
@BalanceDifference ;
```

```
-- Slowing down transaction to let tester the
time
-- to run query for other session
PRINT 'New Balance value: ' +
CONVERT(VARCHAR(32),@CustomerBalance);


-- updating in table
UPDATE BankAccounts
SET BalanceAmount = @CustomerBalance
WHERE AccountId = 1 ;

-- display results for user
SELECT BalanceAmount as BalanceAmountSession1
FROM BankAccounts
WHERE AccountId = 1 ;
COMMIT ;
```

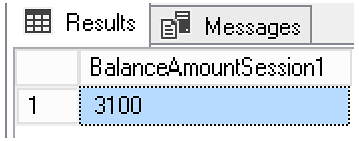
At the same time, as you’ve returned an article to your favorite web reseller, he’s also trying to add 40 to your bank account. Following code will be run:

```
-- Session 2: Web reseller

DECLARE @CustomerBalanceINT ;
DECLARE @BalanceDifferenceINT ;

SET @BalanceDifference = 40 ;

BEGIN TRANSACTION ;
-- Getting back current balance value
SELECT @CustomerBalance = BalanceAmount
FROM BankAccounts
WHERE AccountId = 1 ;

PRINT 'Read Balance value: ' +
CONVERT(VARCHAR(32),@CustomerBalance);
```

```
-- adding salary amount
SET @CustomerBalance = @CustomerBalance +
@BalanceDifference ;

PRINT 'New Balance value: ' +
CONVERT(VARCHAR(32),@CustomerBalance);

-- updating in table
UPDATE BankAccounts
SET BalanceAmount = @CustomerBalance
WHERE AccountId = 1 ;

-- display results for user
SELECT BalanceAmount as BalanceAmountSession2
FROM BankAccounts
WHERE AccountId = 1 ;
COMMIT ;
```

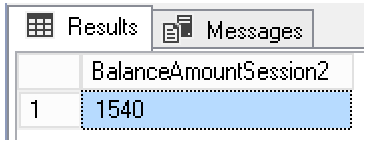
Here are the results we will get from final SELECT in each query:

Session 1:



	BalanceAmountSession1
1	3100

Session 2:



	BalanceAmountSession2
1	1540

Unfortunately, we lost the money from web reseller…

2. Dirty read

To illustrate dirty reads, we will update data in **Person.Person** table: we will update all records so that all rows where **FirstName** column value is “Aaron” will bear the same value for it’s their corresponding **LastName** column. This value will be “Hotchner” but won’t persist we will rollback the transaction.

Here is the script for the first session:

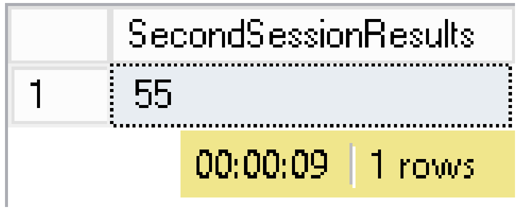
```
SELECT
    COUNT(DISTINCT LastName)
DistinctLastNameBeforeBeginTran
FROM Person.Person
WHERE FirstName = 'Aaron';

BEGIN TRANSACTION;

UPDATE Person.Person
SET LastName = 'Hotchner'
WHERE FirstName = 'Aaron'
;
```

```
SELECT
    COUNT(DISTINCT LastName)
DistinctLastNameInTransaction
FROM Person.Person
WHERE FirstName = 'Aaron';

WAITFOR DELAY '00:00:10.000';

ROLLBACK TRANSACTION;

SELECT
    COUNT(DISTINCT LastName)
DistinctLastNameAfterRollback
FROM Person.Person
WHERE FirstName = 'Aaron';
```

While the first session is in running its **WAITFOR DELAY** instruction, we can run following query in a second session:

```
SELECT
   COUNT(DISTINCT LastName) SecondSessionResults
FROM Person.Person
WHERE FirstName = 'Aaron';
```

With SQL Server’s default isolation level, the second session will be waiting for the first session to complete :



	SecondSessionResults
1	55

```
00:00:09 | 1 rows
```

So, we can say that, by default, SQL Server protects you from dirty reads.

3. Non-repeatable reads

If you remind the explanation above about non-repeatable reads, this problem occurs when two consecutive reads of a given column value in a particular table row led to two different values, meaning that values returned by a query are time-dependent even inside the same transaction.

Once again, we will use two sessions for this experiment. The first session will run a query returning five first rows from Person. Person table, wait for some time then rerun the exact same query.

Here is the code for the first session:

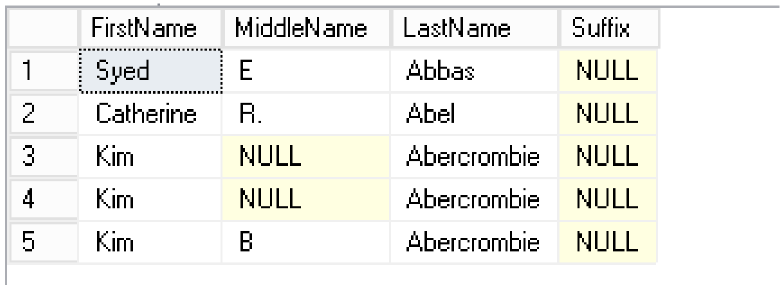
```
-- ensure we use SQL Server default isolation level
SET TRANSACTION ISOLATION LEVEL READ
COMMITTED;

BEGIN TRANSACTION;
-- Query 1 - first run
SELECT TOP 5
    FirstName,
    MiddleName,
    LastName,
    Suffix
FROM Person.Person
ORDER BY LastName;
```

```
-- let some time for session 2
WAITFOR DELAY '00:00:10.000';

-- Query 1 - second run
SELECT TOP 5
    FirstName,
    MiddleName,
    LastName,
    Suffix
FROM Person.Person
ORDER BY LastName
;

COMMIT TRANSACTION;
```

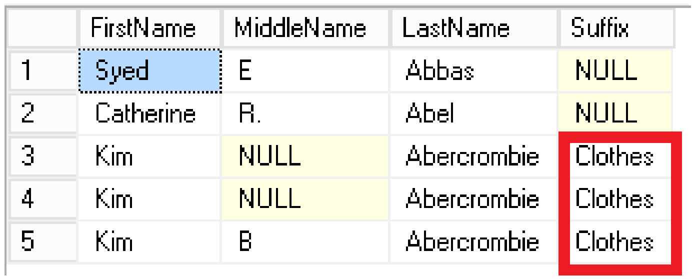
While the first session is waiting, run following code that will update all records that have **FirstName** column value set to “Kim” and **LastName** column value set to “Abercrombie”.

```
-- ensure we use SQL Server default isolation level
SET TRANSACTION ISOLATION LEVEL READ
COMMITTED;

BEGIN TRANSACTION;

UPDATE Person.Person
SET
   Suffix = 'Clothes'
WHERE
   LastName = 'Abercrombie'
AND FirstName = 'Kim';

COMMIT TRANSACTION;
```

Here are the result sets returned in the first session:



	FirstName	MiddleName	LastName	Suffix
1	Syed	E	Abbas	NULL
2	Catherine	R.	Abel	NULL
3	Kim	NULL	Abercrombie	NULL
4	Kim	NULL	Abercrombie	NULL
5	Kim	B	Abercrombie	NULL



	FirstName	MiddleName	LastName	Suffix
1	Syed	E	Abbas	NULL
2	Catherine	R.	Abel	NULL
3	Kim	NULL	Abercrombie	Clothes
4	Kim	NULL	Abercrombie	Clothes
5	Kim	B	Abercrombie	Clothes

We can revert our changes with following query:

```
UPDATE Person.Person
SET
   Suffix = NULL
WHERE
   LastName = 'Abercrombie'
AND FirstName = 'Kim';
```

4. Phantom reads

For this experiment, we will create a table called dbo.Employee and let it empty:

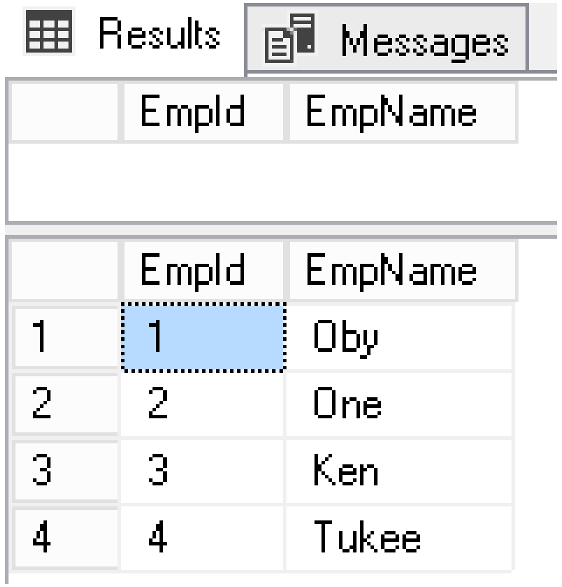
```
IF (OBJECT_ID('dbo.Employee') IS NOT NULL)
BEGIN
  DROP TABLE [dbo].[Employee];
END;

CREATE TABLE [dbo].[Employee] (
  EmpId      int IDENTITY(1,1) NOT NULL,
  EmpName    nvarchar(32)      NOT NULL,
  CONSTRAINT pk_EmpId
    PRIMARY KEY CLUSTERED (EmpId)
);
```

In first session, we will run a SELECT query against this tablespace in time by 10 seconds:

```
-- ensure we use SQL Server default isolation level
SET TRANSACTION ISOLATION LEVEL READ
COMMITTED;

BEGIN TRANSACTION;

-- Query 1 - first run
SELECT *
FROM dbo.Employee
;

-- let some time for session 2
WAITFOR DELAY '00:00:10.000';

-- Query 1 - second run
SELECT *
FROM dbo.Employee
;
COMMIT TRANSACTION;
```

In second session, we will insert some rows in dbo.Employee table.

```
-- ensure we use SQL Server default isolation level
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION;

INSERT INTO [dbo].[Employee] ([EmpName]) VALUES ('Oby');
INSERT INTO [dbo].[Employee] ([EmpName]) VALUES ('One');
INSERT INTO [dbo].[Employee] ([EmpName]) VALUES ('Ken');
INSERT INTO [dbo].[Employee] ([EmpName]) VALUES ('Tukee');

COMMIT TRANSACTION;
```



	EmpId	EmpName

	EmpId	EmpName
1	1	Oby
2	2	One
3	3	Ken
4	4	Tukee

And here are the results sets returned in first session:

As we can see, concurrency (and transaction isolation level), it’s possible to get different results sets for the same query, from time to time even in the same transaction.