Concurrency problems – theory and experimentation in SQL Server

July 24, 2018 by Jefferson Elias

Introduction

Intended audience

This document is intended for application developers and database administrators who are willing to get an overview of comm concurrency problems to which transaction isolation levels respond in the particular case of Microsoft SQL Server.

Typographical Conventions

Convention	Meaning
`Stylized Consolas Font`	Used for blocks of code, commands and script examples. Text should be interpreted exactly as presented.
Consolas Font	Used for inline code, commands or examples. Text should be interpreted exactly as presented.
<italic font in brackets>	Italic texts set in angle brackets denote a variable requiring substitution for a real value.
Italic font	Used to denote the title of a book, article, or other publication.
Note	Additional information or caveats.
About screen capture	• Selections and arrows default color is dark red (Red 192, Green 0 and Blue 0)

Context

In any relational database system, there is the concept of transaction. A transaction is a set of logical operations that have to 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 has to 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.

Last but not least, 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.

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

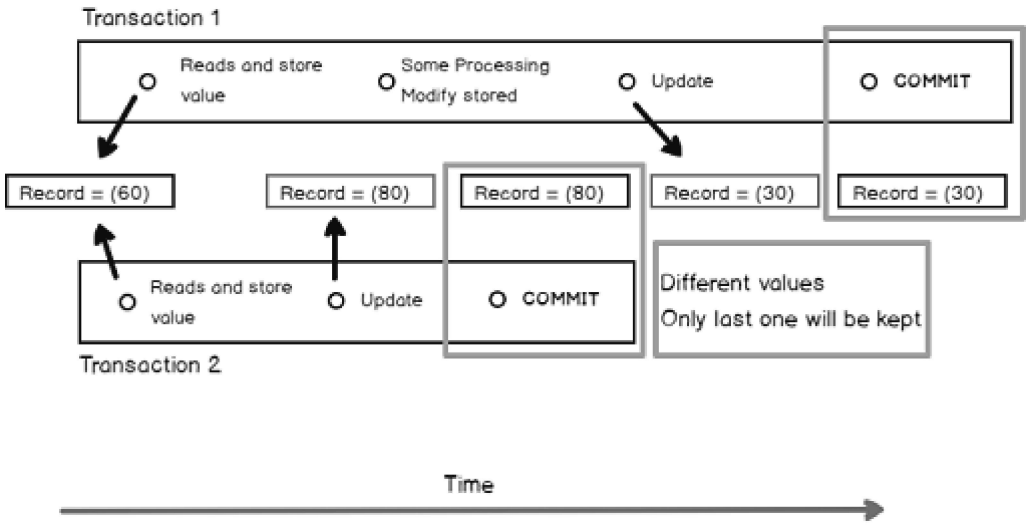
This article is the first one of a series about transaction isolation level. It is divided into two parts. The first one will explain the concurrency problems with a theoretical example while the second will be more practical and we will try to experience these problems on a SQL Server instance.

Concurrency problems

Before diving into transaction levels details, it's important to get used to typical concurrency problems and how we call them.

Lost update and dirty write

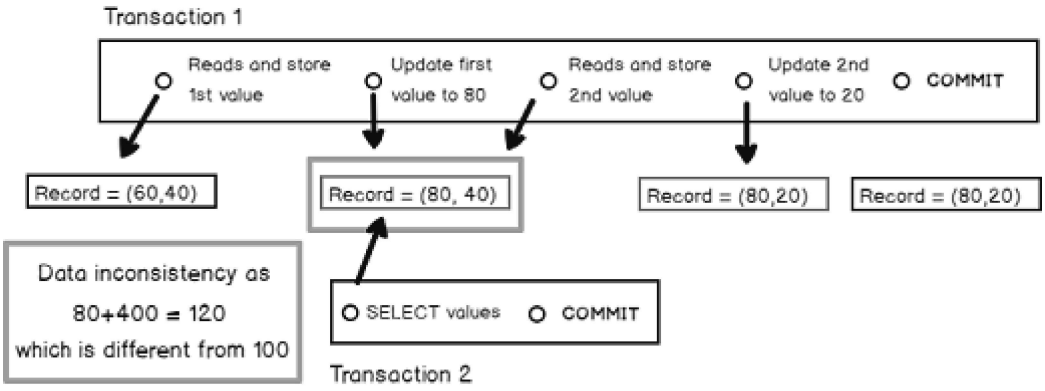
The first transaction reads this record, does some processing then updates this record and finally commits its work. The second transaction reads the record then updates it immediately and commits. Both transactions do not update this record to the same value. This leads to a loss for the update statement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



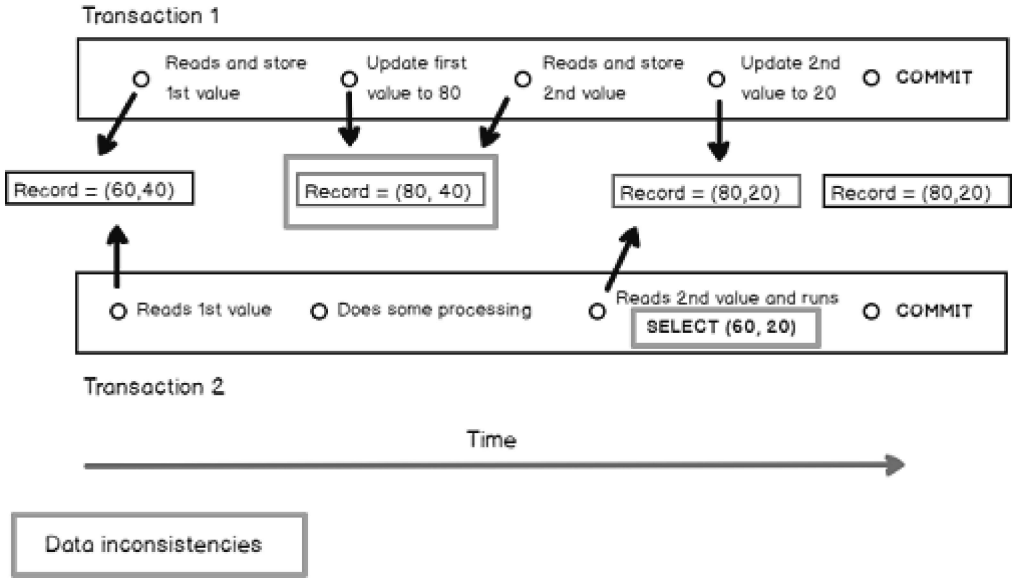
As **Transaction 1** overwrites a value that **Transaction 2** already modified. We could have said that **Transaction 1** did a « dirty write » if **Transaction 2** didn't commit its work.

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.



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.

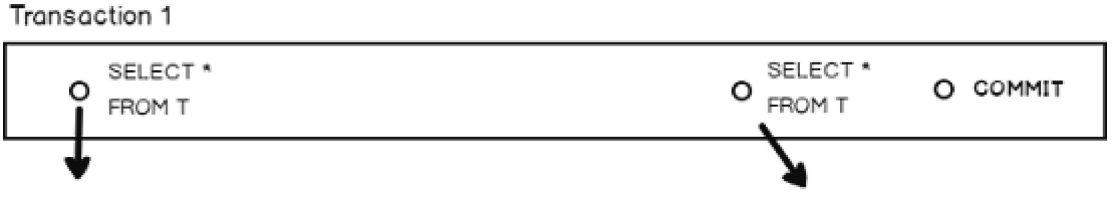


Phantom reads

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:



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.

This will only work in SERIALIZABLE isolation level. We won't discuss it anymore in this article.

Experimentation on SQL Server

The experimentation scripts presented here are all designed using the AdventureWorks2012 database.

If no precision is made about a transaction isolation level in the experimentation detailed explanation, the results presented are those returned using SQL Server's default transaction isolation level, which is READ COMMITTED.

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(
    AccountId       INT IDENTITY(1,1),
    BalanceAmount   INT
);

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

Your employer attempts to pay your (tiny) salary, let's say 1600. This will consist of your first ses-

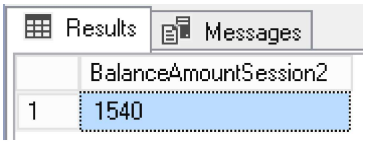
```
    FROM BankAccounts
    WHERE AccountId = 1 ;
    PRINT 'Read Balance value: ' + CONVERT(VARCHAR(32),@CustomerBalance);

    -- 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);

    WAITFOR DELAY '00:00:10.000';

    -- 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 @CustomerBalance    INT ;
DECLARE @BalanceDifference  INT ;

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
```

```
Read Balance value: 1500
New Balance value: 3100
```

Session 2:




```
Read Balance value: 1500
New Balance value: 1540
```

Unfortunately, we lost the money from web reseller…

Now, let's clean up our test.

```
DROP TABLE BankAccounts ;
```

Dirty read

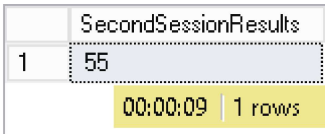
To illustrate dirty reads, we will update data in **Person.Person** table: we will update all records so that all rows where **FirstName** column value is "Aaron" will bear the same value for it's their corresponding **LastName** column. This value will be "Hotchner" but won't persist: we will rollback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

```
    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 :



So, we can say that, by default, SQL Server protects you from dirty reads. Let's just change session isolation level to **READ UNCOMMITTED** and check that SQL Server will present « dirty » data to session 2 in that mode...

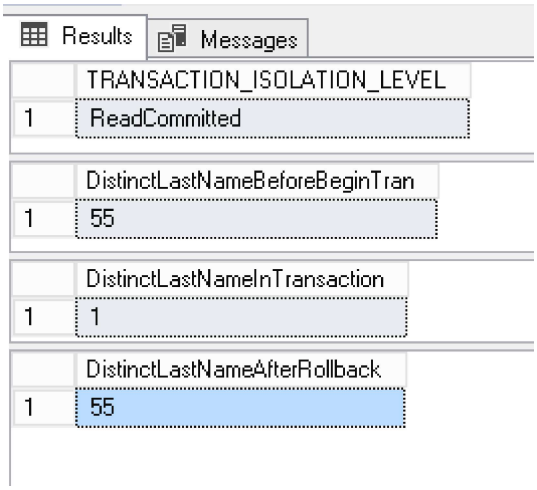
In order to change session's transaction isolation level, run following query:

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

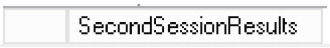
Then, run the code for second session from above again.

Here are the results for both sessions:

Session 1



Session 2



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;
```

We can revert our changes with following query:

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

If we are willing to prevent this behavior to happen, we could have heard about the **REPEATABLE READ** isolation level. Let's try it out!

For the first session, we just need to change the first command from:

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

to

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

and execute it. The code for second session can be used as is.

With this simple change, we have a lock that is held by first session. This leads the second session to wait for the first one to complete before actually modify rows.

Here is a screenshot that proves it:

Session 1

	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	NULL
4	Kim	NULL	Abercrombie	NULL

And, obviously, we do not forget to revert our changes to the query shown above.

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 ('Obv');
```

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

As we can see, duet o 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.

Summary

So far, we've seen that the concurrency of transactions implies some basic issues that transaction isolation level can eventually prevent them to happen.

In the next article, you will get more details about each transaction isolation level, included what common concurrency problem(s) they solve...

References

- A Critique of ANSI SQL Isolation Levels
- Isolation Levels in the Database Engine
- Difference between Non- repeatable read and Phantom read

See more

Check out ApexSQL Plan, to view SQL execution plans, including comparing plans, stored procedure performance profiling, missing index details, lazy profiling, wait times, plan execution history and more

Jefferson Elias

Living in Belgium, I obtained a master degree in Computer Sciences in 2011 at the University of Liege.

I'm one of the rare guys out there who started to work as a DBA immediately after his graduation. So, I work at the university hospital of Liege since 2011. Initially involved in Oracle Database administration (which are still under my charge), I had the opportunity to learn and manage SQL Server instances in 2013. Since 2013, I've learned a lot about SQL Server in administration and development.

I like the job of DBA because you need to have a general knowledge in every field of IT. That's the reason why I won't stop learning (and share) the products of my learnings.

View all posts by Jefferson Elias

Related Posts:

1. **Filtered indexes: Performance analysis and hidden costs**
2. **Isolation levels behavior in SQL Server Always On Availability Groups**
3. **Snapshot Isolation in SQL Server**
4. **SQL Server Transaction Log Interview Questions**
5. **Top SQL Server Books**

Concurrency
168 Views