Part 1: How to solve the transactional issues of isolation levels

By at yazdani, 2012/07/06

Introduction

Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. Isolation levels are described in terms of which concurrency side-effects, such as dirty reads or phantom reads, are allowed. Choosing a proper transaction isolation level ensures you about data protection and integrity during handling a piece of data.

In this article, we discuss the levels of isolation in SQL Server and their impact on system performance. Code examples, in terms of practicality, a comparison and changes in SQL Server 2012 are also included.

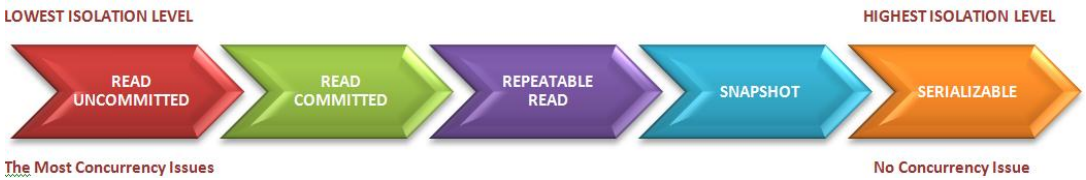
Database Transaction Isolation Levels

The isolation levels, supported by the SQL Server database engine, categorized as the following list:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE
- SNAPSHOT

It is important to know that at a time, only one of the isolation level options can be set, and it remains set for that connection until it is explicitly changed. All read operations performed within the transaction operate under the rules for the specified isolation level unless a table hint in the FROM clause of a statement specifies different locking for a table.

Degree of isolation levels



This figure shows different isolation levels from two aspects including degree of isolation level and amount of concurrency issues. As we are going through the arrows from left to right, the degree of isolation level increases (from the lowest level to the highest level). However, the amount of concurrency issues shows a reduction from left to right and finally at the highest level of isolation (SERIALIZABLE) we encounter no concurrency issue.

The important point in this figure is the relation between degree of isolation level and concurrency issue. As the isolation level becomes more restricted the concurrency issue decreases to zero from READ UNCOMMITTED through SERIALIZABLE.

On the other hand, a lower isolation level increases the ability of many users to access data at the same time, which consequently increases the number of impacts (such as dirty reads or lost updates) users might encounter. Conversely, a higher isolation level reduces the types of concurrency effects, but requires more system resources and increases the chance that one transaction will block another.

Therefore, the highest isolation level (SERIALIZABLE) guarantees that a transaction will retrieve exactly the same data every time it repeats a read operation, by performing a level of locking that is likely to impact other users in multi-user systems. The lowest isolation level (READ UNCOMMITTED), however, may retrieve data that has been modified but not committed yet by other transactions. Because there is no read locking, in this method, all the concurrency side-effects can happen. *As a general rule, choosing the appropriate isolation level depends on balancing the data integrity requirements of the application against the overhead of each isolation level.*

Concurrency side-effects

Now you have the general point about the degrees in transaction isolation levels and the related concurrency impacts. The thing you should be concerned about is the accurate understanding of impacts in each isolation level. To help you implementing the accurate isolation level based on data integrity or higher concurrency, the side-effects of each isolation levels are closely elaborated in next two sections.

- **Lost Update**: This generally occurs when more than one transaction tries to update a specific record at a time.
- **Non-Repeatable Read**: This deals with inconsistent data. Suppose you read one value from a table and started working on it but meanwhile another process modifies the value in the source resulting a false output in your transaction. This is called Non-repeatable read.

  Let's have a more practical example, suppose before withdrawing money from your account, you always perform a balance check and you find 90$ as a balance in your account. Then you perform withdraw operation and try to withdraw 60$ from your account but meanwhile the bank manager debits 50$ from your account as a penalty of minimum balance (100$), as a result you have only 40$ in your account now. So your transaction either fails as the demanded amount (60$) is not there in your account or it may show (-20$) (which is quite impossible as of banking constraints). More simply we can say Non-repeatable read take place if a transaction is able to read the same row several times and gets a different value for each time.
- **Dirty Read**: This is one of the types of a Non-repeatable read. This happens when a process tries to read a piece of data while another process is performing update operations on it and not completed yet.
- **Phantom Read**: Phantom means unexpected or unrealistic. It occurs when two identical queries are executed, and the set of rows returned by the second query differs from the first.

As a simple example, suppose your banking policy got changed and the minimum balance should be $150 instead of $100 for each account type. This is not a big deal for a database administrator. They will run an update statement for each account type where the minimum balance is less than $150 and update the value to $150. Unfortunately when the manager checked the database, he found one record with a minimum balance less than $150 in the same table. The DBA was surprised how is this possible as he performed the update statement on the whole table.

The occurrence of the phantom read is very rare as it needs proper circumstances and timing as in the above example. Someone may have inserted one new record with the minimum balance less than $150 at the very same time when the DBA executed the UPDATE statement. As it is a new record, it didn't interfere with the UPDATE transaction and executed successfully.

After familiarizing ourselves with the possible impacts of isolation levels, we are going to examine how each isolation level deals with the mentioned side-effects. In examples here, in order to explain the behavior of mentioned isolation levels, two users simulate concurrency.

READ UNCOMMITTED

Transactions running at the READ UNCOMMITTED level do not issue shared locks to prevent other transactions from modifying data read by the current transaction. This isolation level is also not blocked by exclusive locks that would prevent the current transaction from reading rows that have been modified but not committed by other transactions.

The impact of this method is that the ability to read uncommitted modifications of data, called dirty reads. It is like setting NOLOCK on all tables in all SELECT statements in a transaction and is known as the least restrictive isolation level. This is a useful case when you need a higher concurrency level, at the cost of side-effects like dirty reads.

The following example simulates two users are working with "Person.Contact" table, concurrently. The second user applies the isolation level defined by "SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED" command, and tries to access the same piece of data being changed by the first user. Consequently, user2 returns the value that is discarded by the user1's transaction.

User1:

```
Use AdventureWorks;
go

Begin TRAN
Update Person.Contact
  SET Phone = '500 555-0132'
  Where ContactID = 5
-- Consider a CPU intensive operation here before committing the transaction
-- For achieving time concurrency in this example
Waitfor Delay '00:00:10'
RollBack
```

User2:

```
Use AdventureWorks
go

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
Select phone
 From Person.Contact
 Where ContactID = 5
```

Result for User2:

Such adverse effects as dirty reads, lost updates, phantom reads, and non-repeatable reads are the cost of the higher levels of concurrency achieving by this level of isolation level.

READ COMMITTED

READ COMMITTED is the default isolation level of SQL Server and specifies statements cannot read data that has been modified but not yet committed. Although it prevents dirty reads, other side-effects might occur. In other words, data can be changed by other transactions between individual statements within the current transaction, resulting in lost updates, phantom reads, or non-repeatable reads.

In the following code snippet, the second user uses the default isolation level (READ COMMITTED).

User1:

```
Use AdventureWorks;
go

Begin TRAN
Update Person.Contact
SET Phone = '500 555-0132'
Where ContactID = 5
-- Consider a CPU intensive operation here before Committing the transaction
-- For achieving time concurrency in this example:
Waitfor Delay '00:00:10'
RollBack
```

User2:

```
Use AdventureWorks
go
```

```
Select phone
 From Person.Contact
 Where ContactID = 5
```

Result for User2

As noted, READ COMMITTED isolation level does not allow uncommitted date to be read. Therefore, the second user should be blocked until the commitment of the first user's transaction. As a result, the dirty read impact does not happen and the valid phone is returned. However, another problem, the lost update, might occur. It is the most possible impact of concurrency in data modification. It happens while two concurrent transactions are trying to update a piece of data and one of them overwrites and discards the second transaction changes.

In this example, both users compete to change the phone filled in "Person.Contact" Table. The users completeo their respective updates and at the end, user2 overwrites the value modified by user1.

User1:

```
Use AdventureWorks;
go

Begin TRAN
Declare @phone nvarchar(25)
Select @phone = phone From Person.Contact Where ContactID = 5
-- Consider a CPU intensive operation here before Committing the transaction
-- For achieving time concurrency in this example:
Waitfor Delay '00:00:10'
IF @phone is null
Update Person.Contact SET Phone = '500 555-0131' Where ContactID = 5
COMMIT TRAN
```

User2:

```
Use AdventureWorks;
go

Begin TRAN
Declare @phone nvarchar(25)
Select @phone = phone From Person.Contact Where ContactID = 5
-- Consider a CPU intensive operation here before Committing the transaction
-- For achieving time concurrency in this example:
Waitfor Delay '00:00:10'
```

```
IF @phone is null
Update Person.Contact SET Phone = '500 555-0132' Where ContactID = 5
COMMIT TRAN
```

The result after commitment of second user's transaction

The result, after commitment of both users' transactions, shows the lost update made by user1. This is called "lost update" impact which should be addressed by a higher isolation level like repeatable read.

Conclusion

To avoid unpredictable result set in running concurrent queries, it is important to manage data modification based on your desired degree of isolation level. In the next part, we will address other concurrency issues.

References

- http://msdn.microsoft.com/en-us/library/ms189122.aspx
- http://www.sqllion.com/2009/07/transaction-isolation-levels-in-sql-server/
- http://msdn.microsoft.com/en-us/library/ms173763.aspx
