Transaction

In general, a Transaction is a single unit of work consists of multiple related tasks that should succeed or fail as one atomic unit. To make the concept of the transaction more familiar and why it should go all or none, imagine one of the most critical transaction examples in our daily life, which is withdrawing money from the ATM.

All of us are familiar with the steps of withdrawing money from the ATM, but it deserves listing the common ones below:

1. Insert your card
2. Select the language
3. Enter your PIN
4. Select the transaction type, which is withdraw in our example here
5. Enter the amount
6. Pull the card
7. Take your money

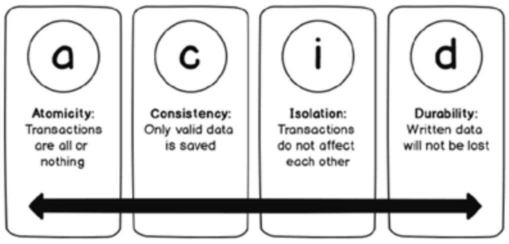
The previous tasks should run as one unit to keep your bank account in consistent state. This means that, if you specify the amount to withdraw from your bank account, it should be guaranteed that the money is getting out of the ATM before confirming the deduction from your account. If any task from the previous tasks fails, the overall withdraw transaction should be cancelled. For example, if any technical or mechanical error occurred after specifying the amount and deducting it from your account, that prevents the money from getting out the ATM, the operation should be canceled and the amount will be returned to your account.

In SQL Server, the Transaction concept is highly required to maintain the integrity of data in a database, especially when executing multiple related tasks sequentially on different tables, databases or servers, or accessing the same records by more than one session concurrently. In all these cases, the transaction should work as one unit of failure or success.

In the ATM withdraw example, the SQL Server Engine will commit the overall transaction, including the amount deduction from the bank account, when all the steps are performed successfully, and the money received by the user. To handle the concurrency, the SQL Server Engine holds a lock on the bank account during that transaction, to guarantee that no other transaction will be performed on the same account at the same time, that may result with dirty read or incorrect operation result, and release the lock when the overall transaction is committed or rolled back. Due to this lock that prevents other transactions from accessing the same records, it is better always better to narrow the scope of the transaction, so that the records will be locked for short period of time, in order not to affect the overall performance of the system.

ACID

After understanding the concept of the SQL Server Transaction, we can describe the Transaction using the four ACID properties. ACID is the acronym for Atomicity, Consistency, Isolation and Durability.


a
Atomicity: Transactions are all or nothing
c
Consistency: Only valid data is saved
i
Isolation: Transactions do not affect each other
d
Durability: Written data will not be lost

Atomicity means that the transaction will succeed, as one unit, if all the separate tasks succeed with no issue. On the other hand, the failure of any single task within this transaction leads to the overall transaction failure and rollback. In other words, Atomicity guarantees that the transaction is all or none.

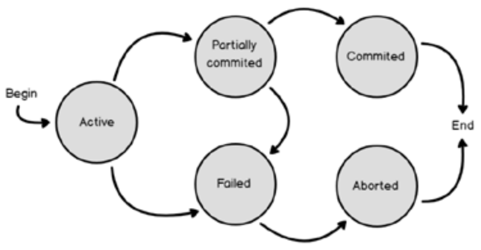
Consistency guarantees that the transaction will not affect the database consistency and will leave it in a valid state by complying with all database rules, such as foreign keys and constraints, defined on the columns.

Isolation means that, each transaction has it is own boundary, that is separated from the other concurrently executing transactions, and will not be affected by these transactions' operations.

Durability means that the result of the committed transaction that is written permanently to the database will not be lost even in the case of any abnormal system failure or termination, as there should be mechanism of recovering this data, that we will see later in the next articles of this series.

Transaction states

- The states of the transaction can be summarized as follows:
- The running transaction is referred to as the Active transaction
- The transaction that completes its execution successfully without any error is referred to as a Committed transaction
- The transaction that does not complete it is execution successfully is referred to as an Aborted transaction
- The transaction that is not fully committed yet is referred to as a Partially Committed transaction
- If the transaction does not complete its execution, it is referred to as a Failed transaction, that is Aborted without being committed
- If the Partially Committed transaction completes its execution successfully, it will be Committed, otherwise it will be Failed then Aborted



Commits

In SQL Server, each single T-SQL statement is considered as an Auto-Commit transaction, that will be committed when the statement is completed successfully and rolled back if the statement is failed. If the T-SQL query uses the BEGIN TRANSACTION statement to control the start the transaction, COMMIT TRANSACTION statement to complete the transaction successfully, and ROLLBACK TRANSACTION statement to abort the transaction, the auto-commit transaction mode will be overridden to become an Explicit Transaction.

Implicit vs explicit transactions

Assume that you want to perform a database change that starts but is not completed unless you explicitly indicate that. To achieve that, you can enforce the opened database connection to work in the Implicit Transaction mode by using the SET IMPLICIT TRANSACTIONS ON|OFF statements. In this case, the command specified after the SET statement will start executing and remain, and locking the database resource, until you explicitly issue a COMMIT statement to complete the transaction successfully or a ROLLBACK statement to abort the transaction and revert all changes.

When the Multiple Active Result Sets (MARS) option is enabled, the implicit or explicit transaction that has multiple batches running at the same time will be running under the Batch-Scoped Transaction mode. In this case, the batch-scoped transaction that is not committed or rolled back after completing the batch will be rolled back automatically by the SQL Server Engine.

Local vs distributed transactions

SQL Server supports both the Local transactions that are processing data from the local database server and the Distributed transactions that are processing data from more than one database server. In distributed transactions, the transaction states, COMMIT and ROLLBACK coordination among the different servers is performed using a transaction manager component called the Microsoft Distributed Transaction Coordinator, also known as MSDTC. To commit the distributed transaction, all participants must complete their related task successfully. If even a single participant fails for any reason, the overall transaction fails, and any changes to data within the scope of the transaction will be rolled back.

Transaction best practices

Using the SQL Server transaction helps maintaining the database integrity and consistency. On the other hand, a badly written transaction may affect the overall performance of your system by locking the database resources for long time. To overcome this issue, it is better to consider the following points when writing a transaction:

- Narrow the scope of the transaction
- Retrieve the data from the tables before opening the transaction if possible
- Access the least amount of data inside the transaction body
- Do not ask for user input inside the body of the transaction
- Use a suitable mode of transactions

General Remarks

BEGIN TRANSACTION (Transact-SQL)

Syntax

```
--Applies to SQL Server and Azure SQL Database

BEGIN { TRAN | TRANSACTION }
    [ { transaction_name | @tran_name_variable }
      [ WITH MARK [ 'description' ] ]
    ]
[ ; ]
```

transaction_name

Is the name assigned to the transaction. transaction_name must conform to the rules for identifiers, but identifiers longer than 32 characters are not allowed. Use transaction names only on the outermost pair of nested BEGIN...COMMIT or BEGIN...ROLLBACK statements. transaction_name is always case sensitive, even when the instance of SQL Server is not case sensitive.

@tran_name_variable

Is the name of a user-defined variable containing a valid transaction name. The variable must be declared with a char, varchar, nchar, or nvarchar data type. If more than 32 characters are passed to the variable, only the first 32 characters will be used; the remaining characters will be truncated.

WITH MARK ['description']

Specifies that the transaction is marked in the log. description is a string that describes the mark. A description longer than 128 characters is truncated to 128 characters before being stored in the msdb.dbo.logmarkhistory table.

If WITH MARK is used, a transaction name must be specified. WITH MARK allows for restoring a transaction log to a named mark.

Marks the starting point of an explicit, local transaction. Explicit transactions start with the BEGIN TRANSACTION statement and end with the COMMIT or ROLLBACK statement.

BEGIN TRANSACTION increments @@TRANCOUNT by 1.

BEGIN TRANSACTION represents a point at which the data referenced by a connection is logically and physically consistent. If errors are encountered, all data modifications made after the BEGIN TRANSACTION can be rolled back to return the data to this known state of consistency. Each transaction lasts until either it completes without errors and COMMIT TRANSACTION is issued to make the modifications a permanent part of the database, or errors are encountered and all modifications are erased with a ROLLBACK TRANSACTION statement.

BEGIN TRANSACTION starts a local transaction for the connection issuing the statement. Depending on the current transaction isolation level settings, many resources acquired to support the Transact-SQL statements issued by the connection are locked by the transaction until it is completed with either a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement. Transactions left outstanding for long periods of time can prevent other users from accessing these locked resources, and also can prevent log truncation.

Although BEGIN TRANSACTION starts a local transaction, it is not recorded in the transaction log until the application subsequently performs an action that must be recorded in the log, such as executing an INSERT, UPDATE, or DELETE statement. An application can perform actions such as acquiring locks to protect the transaction isolation level of SELECT statements, but nothing is recorded in the log until the application performs a modification action.

Naming multiple transactions in a series of nested transactions with a transaction name has little effect on the transaction. Only the first (outermost) transaction name is registered with the system. A rollback to any other name (other than a valid savepoint name) generates an error. None of the statements executed before the rollback is, in fact, rolled back at the time this error occurs. The statements are rolled back only when the outer transaction is rolled back.

The local transaction started by the BEGIN TRANSACTION statement is escalated to a distributed transaction if the following actions are performed before the statement is committed or rolled back:

- An INSERT, DELETE, or UPDATE statement that references a remote table on a linked server is executed. The INSERT, UPDATE, or DELETE statement fails if the OLE DB provider used to access the linked server does not support the ITransactionJoin interface.
- A call is made to a remote stored procedure when the REMOTE_PROC_TRANSACTIONS option is set to ON.

The local copy of SQL Server becomes the transaction controller and uses Microsoft Distributed Transaction Coordinator (MS DTC) to manage the distributed transaction.

A transaction can be explicitly executed as a distributed transaction by using BEGIN DISTRIBUTED TRANSACTION.

When SET IMPLICIT_TRANSACTIONS is set to ON, a BEGIN TRANSACTION statement creates two nested transactions. For more information see, SET IMPLICIT_TRANSACTIONS (Transact-SQL)

Marked Transactions

The WITH MARK option causes the transaction name to be placed in the transaction log. When restoring a database to an earlier state, the marked transaction can be used in place of a date and time.

Additionally, transaction log marks are necessary if you need to recover a set of related databases to a logically consistent state. Marks can be placed in the transaction logs of the related databases by a distributed transaction. Recovering the set of related databases to these marks results in a set of databases that are transactionally consistent. Placement of marks in related databases requires special procedures.

The mark is placed in the transaction log only if the database is updated by the marked transaction. Transactions that do not modify data are not marked.

BEGIN TRAN new_name WITH MARK can be nested within an already existing transaction that is not marked. Upon doing so, new_name becomes the mark name for the transaction, despite the name that the transaction may already have been given. In the following example, M2 is the name of the mark.

```sql
BEGIN TRAN T1;
UPDATE table1 ...;
BEGIN TRAN M2 WITH MARK;
UPDATE table2 ...;
SELECT * from table1;
COMMIT TRAN M2;
UPDATE table3 ...;
COMMIT TRAN T1;


--Examples
--A. Using an explicit transaction
use AdventureWorks2012
BEGIN TRANSACTION;
DELETE FROM HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
COMMIT;

--B. Rolling back a transaction
/*
The following example shows the effect of rolling back a transaction.
In this example, the ROLLBACK statement will roll back the
INSERT statement, but the created table will still exist.
*/
CREATE TABLE ValueTable (id INT);
BEGIN TRANSACTION;
       INSERT INTO ValueTable VALUES(1);
       INSERT INTO ValueTable VALUES(2);
ROLLBACK;

--C. Naming a transaction
--The following example shows how to name a transaction.
DECLARE @TranName VARCHAR(20);
SELECT @TranName = 'MyTransaction';
```

```
BEGIN TRANSACTION @TranName;
USE AdventureWorks2012;
DELETE FROM AdventureWorks2012.HumanResources.JobCandidate
    WHERE JobCandidateID = 13;

COMMIT TRANSACTION @TranName;
GO
--D. Marking a transaction
/*
The following example shows how to mark a transaction.
The transaction CandidateDelete is marked.
*/
BEGIN TRANSACTION CandidateDelete
    WITH MARK N'Deleting a Job Candidate';
GO
USE AdventureWorks2012;
GO
DELETE FROM AdventureWorks2012.HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
GO
COMMIT TRANSACTION CandidateDelete;
GO
```

We will create a sample table through the following query and will populate some sample data.

An Implicit Transaction in SQL Server

In order to define an implicit transaction, we need to enable the IMPLICIT_TRANSACTIONS option. The following query illustrates an example of an implicit transaction.

@@TRANCOUNT function returns the number of BEGIN TRANSACTION statements in the current session and we can use this function to count the open local transaction numbers in the examples

```
SET IMPLICIT_TRANSACTIONS ON
UPDATE
    Person
SET
    Lastname = 'Sawyer',
    Firstname = 'Tom'
WHERE
    PersonID = 2
SELECT
    IIF(@@OPTIONS & 2 = 2,
    'Implicit Transaction Mode ON',
    'Implicit Transaction Mode OFF'
    ) AS 'Transaction Mode'
SELECT
    @@TRANCOUNT AS OpenTransactions
COMMIT TRAN
SELECT

    @@TRANCOUNT AS OpenTransactions
```

An Explicit Transaction in SQL Server

In order to define an explicit transaction, we start to use the BEGIN TRANSACTION command because this statement identifies the starting point of the explicit transaction. It has the following syntax:

```
BEGIN TRANSACTION [ {transaction_name | @tran_name_variable }

    [WITH MARK ['description']]]
```

transaction_name option is used to assign a specific name to transactions

@trans_var option is a user-defined variable that is used to hold the transaction name

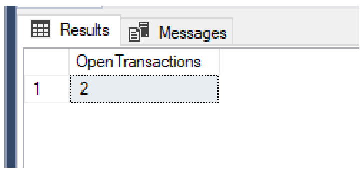
WITH MARK option enable to mark a particular transaction in the log file

After defining an explicit transaction through the BEGIN TRANSACTION command, the related resources acquired a lock depending on the isolation level of the transaction. For this reason as possible to use the shortest transaction will help to reduce lock issues. The following statement starts a transaction and then it will change the name of a particular row in the Person table.

```
BEGIN TRAN

UPDATE Person
SET    Lastname = 'Lucky',
        Firstname = 'Luke'
WHERE  PersonID = 1

SELECT @@TRANCOUNT AS OpenTransactions
```

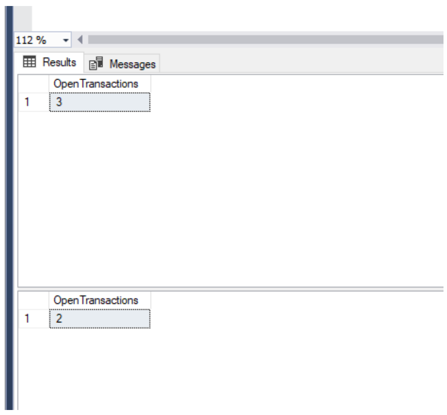


```
Results  Messages
   OpenTransactions
1  2
```

As we stated in the previous section COMMIT TRAN statement applies the data changes to the database and the changed data will become permanent. Now let's complete the open transaction with a COMMIT TRAN statement.

```
BEGIN TRAN
UPDATE Person
SET    Lastname = 'Lucky',
        Firstname = 'Luke'
WHERE  PersonID = 1
SELECT @@TRANCOUNT AS OpenTransactions
COMMIT TRAN

SELECT @@TRANCOUNT AS OpenTransactions
```



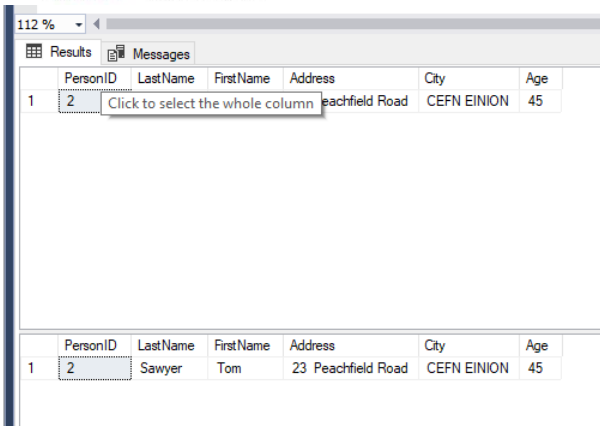
```
112 %
Results  Messages
   OpenTransactions
1  3

   OpenTransactions
1  2
```

On the other hand, the ROLLBACK TRANSACTION statement helps in undoing all data modifications that are applied by the transaction. In the following example, we will change a particular row but this data modification will not persist.

```
BEGIN TRAN
UPDATE Person
SET    Lastname = 'Donald',
        Firstname = 'Duck'  WHERE PersonID=2
SELECT * FROM Person WHERE PersonID=2
ROLLBACK TRAN

SELECT * FROM Person WHERE PersonID=2
```


112 %

Results | Messages

	PersonID	LastName	FirstName	Address	City	Age
1	2	Click to select the whole column		eachfield Road	CEFN EINION	45

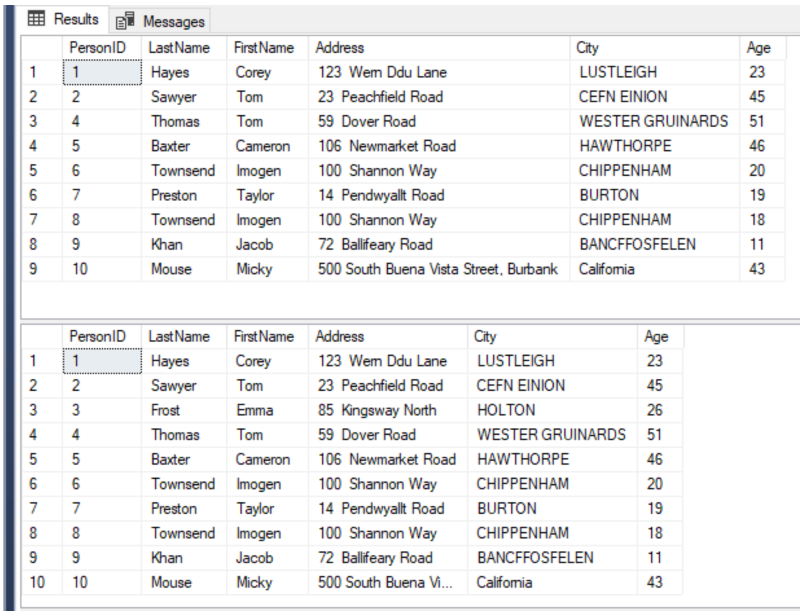
	PersonID	LastName	FirstName	Address	City	Age
1	2	Sawyer	Tom	23 Peachfield Road	CEFN EINION	45

Save Points in Transactions

Savepoints can be used to rollback any particular part of the transaction rather than the entire transaction. So that we can only rollback any portion of the transaction where between after the save point and before the rollback command. To define a save point in a transaction we use the SAVE TRANSACTION syntax and then we add a name to the save point. Now, let's illustrates an example of savepoint usage. When we execute the following query, only the insert statement will be committed and the delete statement will be rolled back.

```
BEGIN TRANSACTION
INSERT INTO Person
VALUES('Mouse', 'Micky','500 South Buena Vista Street, Burbank','California',43)
SAVE TRANSACTION InsertStatement
DELETE Person WHERE PersonID=3
SELECT * FROM Person
ROLLBACK TRANSACTION InsertStatement
COMMIT

SELECT * FROM Person
```


Results | Messages

	PersonID	LastName	FirstName	Address	City	Age
1	1	Hayes	Corey	123 Wern Ddu Lane	LUSTLEIGH	23
2	2	Sawyer	Tom	23 Peachfield Road	CEFN EINION	45
3	4	Thomas	Tom	59 Dover Road	WESTER GRUINARDS	51
4	5	Baxter	Cameron	106 Newmarket Road	HAWTHORPE	46
5	6	Townsend	Imogen	100 Shannon Way	CHIPPENHAM	20
6	7	Preston	Taylor	14 Pendwyallt Road	BURTON	19
7	8	Townsend	Imogen	100 Shannon Way	CHIPPENHAM	18
8	9	Khan	Jacob	72 Ballifeary Road	BANCFFOSFELEN	11
9	10	Mouse	Micky	500 South Buena Vista Street, Burbank	California	43

	PersonID	LastName	FirstName	Address	City	Age
1	1	Hayes	Corey	123 Wern Ddu Lane	LUSTLEIGH	23
2	2	Sawyer	Tom	23 Peachfield Road	CEFN EINION	45
3	3	Frost	Emma	85 Kingsway North	HOLTON	26
4	4	Thomas	Tom	59 Dover Road	WESTER GRUINARDS	51
5	5	Baxter	Cameron	106 Newmarket Road	HAWTHORPE	46
6	6	Townsend	Imogen	100 Shannon Way	CHIPPENHAM	20
7	7	Preston	Taylor	14 Pendwyallt Road	BURTON	19
8	8	Townsend	Imogen	100 Shannon Way	CHIPPENHAM	18
9	9	Khan	Jacob	72 Ballifeary Road	BANCFFOSFELEN	11
10	10	Mouse	Micky	500 South Buena Vi...	California	43

Auto Rollback transactions in SQL Server

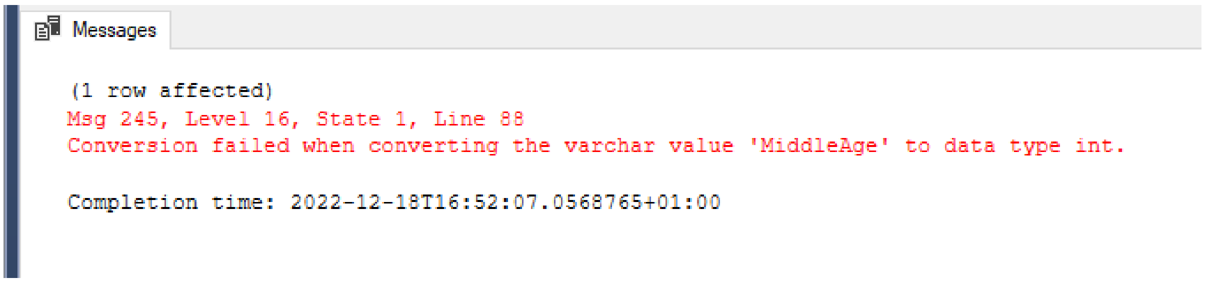
Generally, the transactions include more than one query. In this manner, if one of the SQL statements returns an error all modifications are erased, and the remaining statements are not executed. This process is called Auto Rollback Transaction in SQL. Now let's explain this principle with a very simple example.

```
BEGIN TRAN
INSERT INTO Person
VALUES('Bunny', 'Bugs','742 Evergreen Terrace','Springfield',54)

UPDATE Person SET Age='MiddleAge' WHERE PersonID=7
```

```
SELECT * FROM Person

COMMIT TRAN
```



```
(1 row affected)
Msg 245, Level 16, State 1, Line 88
Conversion failed when converting the varchar value 'MiddleAge' to data type int.

Completion time: 2022-12-18T16:52:07.0568765+01:00
```

As we can see from the above image, there was an error that occurred in the update statement due to the data type conversion issue. In this case, the inserted data is erased and the select statement did not execute.