Modes of Transactions in SQL Server

February 17, 2021 by Esat Erkec

In this article, we are going to talk about the modes of transactions in SQL Server.

Introduction

A transaction is the smallest work unit that is executed in the database and transactions also meet the properties of the ACID (atomicity, consistency, isolation and durability). SQL Server can operate 3 different transactions modes and these are:

- Auto-commit transactions
- Implicit transactions
- Explicit transactions

In the following sections, we will tackle these transactions' features, similarities, and differences.

Auto-commit transactions in SQL Server

The auto-commit transaction mode is the default transaction mode of the SQL Server. In this mode, each SQL statement is evaluated as a transaction by the storage engine. In this context, if any SQL statement completes its execution successfully it is committed and the data modification will become permanent in the database. On the other hand, if any statement faces any error it will be rolled back. In this transaction mode, we don't try to manage transactions and all operations are managed by the SQL Server.

In the example below, after creating a table, we will insert one row into it.

```
CREATE TABLE Person
(PersonID  INT
PRIMARY KEY,
LastName  VARCHAR(255),
FirstName VARCHAR(255),
Address   VARCHAR(255),
City      VARCHAR(255),
Age       INT
);
```

When we execute the following query, SQL Server will automatically start a transaction and then it commits the transaction because this insert statement will not return any error.

```
INSERT INTO Person
VALUES
(1,
'Hayes',
'Corey',
'123  Wern Ddu Lane',
'LUSTLEIGH',
23
);
```

When we try to execute the following query, SQL Server rollbacks the data modification due to a duplicate primary key error.

```
INSERT INTO Person
VALUES
(1,
'Macdonald',
'Charlie',
'23  Peachfield Road',
'CEFN EINION',
45
);
```

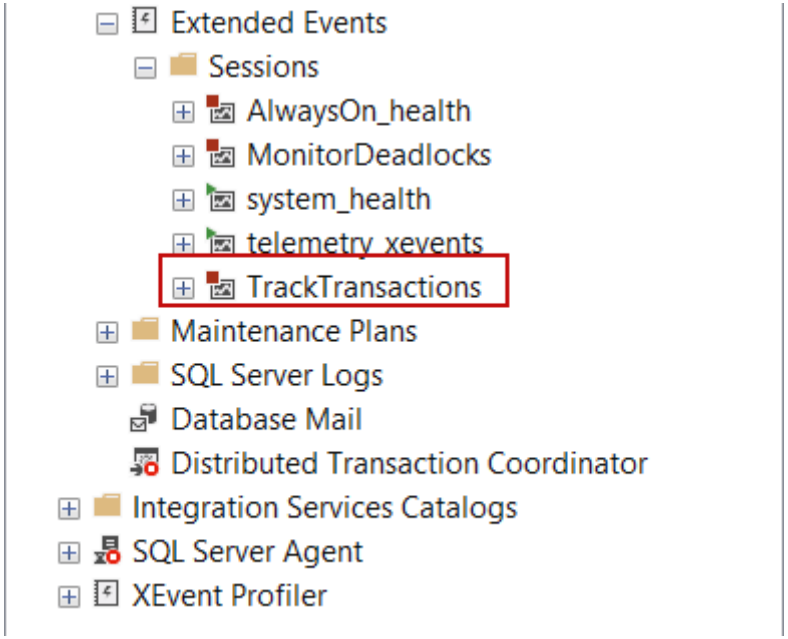
```
Messages
Msg 2627, Level 14, State 1, Line 4
Violation of PRIMARY KEY constraint 'PK__Person__AA2FFB859AB122F6'. Cannot insert duplicate key in object 'dbo.Person'. The duplicate key value is (1).
The statement has been terminated.

Completion time: 2020-12-26T15:50:53.8553187+03:00
```

Now, we will look at behind the scenes of auto-commit transactions in SQL Server with the help of the extended events. Extended Events helps to capture the activities of the SQL Server and it's a very beneficial feature to collect and monitor different events occurred behind the scene.

Through the following query, we will create an extended event that captures the committed and rollback transactions in SQL Server.

```
CREATE EVENT SESSION TrackTransactions
ON SERVER
ADD EVENT sqlserver.sql_transaction (
ACTION (sqlserver.session_id,sqlserver.database_id,sqlserver.sql_text)
WHERE
(transaction_state =1 OR transaction_state =2) AND
sqlserver.database_name = 'AdventureWorks2017'
)
ADD TARGET package0.ring_buffer;
GO
```

After creating the extended event, it will be shown under the Extended Events folder of the SQL Server Management Studio (SSMS).



We can start it either manually or can start with the help of the below query:

```
ALTER EVENT SESSION TrackTransactions ON SERVER STATE=START;
```

Now, we will execute the following queries, the first of them will be executed successfully and the second one will return an error.

```
INSERT INTO Person
VALUES
(2,
'Townsend',
'Imogen ',
'100  Shannon Way',
'CHIPPENHAM',
20);
GO
INSERT INTO Person
VALUES
(2,
'Khan',
'Jacob',
'72  Ballifeary Road',
'BANCFFOSFELEN',
11
```

```
);
```

```
Messages

    (1 row affected)
    Msg 2627, Level 14, State 1, Line 3
    Violation of PRIMARY KEY constraint 'PK__Person__AA2FFB858C46DAFA'. Cannot insert duplicate key in object 'dbo.Person'. The duplicate key value is (2).
    The statement has been terminated.

    Completion time: 2020-12-26T16:52:48.9653707+03:00
```

The following query will return the collected details that have been captured by the extended event.

```
SELECT *
FROM(SELECT event.value('(event/@name)[1]', 'varchar(50)') AS event,
            DATEADD(hh, DATEDIFF(hh, GETUTCDATE(), CURRENT_TIMESTAMP), event.value('(event/@timestamp)[1]', 'datetime
2')) AS [timestamp],
            event.value('(event/action[@name="session_id"])[1]', 'int') AS session_id,
            event.value('(event/action[@name="database_id"])[1]', 'int') AS database_id,
            event.value('(event/data[@name="duration"])[1]', 'bigint') AS duration_microseconds,
            event.value('(event/data[@name="transaction_id"])[1]', 'bigint') AS transaction_id,
            event.value('(event/data[@name="transaction_state"]/text)[1]', 'nvarchar(max)') AS transaction_state,
            event.value('(event/data[@name="transaction_type"]/text)[1]', 'nvarchar(max)') AS transaction_type,
            event.value('(event/action[@name="sql_text"])[1]', 'nvarchar(max)') AS sql_text
    FROM
    (
        SELECT n.query('.') AS event
        FROM
            (
            SELECT CAST(target_data AS XML) AS target_data
            FROM sys.dm_xe_sessions AS s
                    JOIN sys.dm_xe_session_targets AS t ON s.address = t.event_session_address
            WHERE s.name = 'TrackTransactions' AND
                    t.target_name = 'ring_buffer'
        ) AS s
        CROSS APPLY target_data.nodes('RingBufferTarget/event') AS q(n)
    ) AS t) AS TMP_TBL
WHERE TMP_TBL.session_id <> @@SPID;
```

Results Messages

	event	timestamp	session_id	database_id	duration_microseconds	transaction_id	transaction_state	transaction_type	sql_text
1	sql_transaction	2020-12-26 17:04:11.2400000	81	13	242	338479	Commit	System	INSERT INTO Person VALUES(2,'Townsend','Imogen','100 Shannon Way','CHIPPENHAM',20)

2	sql_transaction	2020-12-26 17:04:11.2400000	81	13	431	338478	Commit	System	INSERT INTO Person VALUES(2,'Townsend','Imogen ','100 Shannon Way','CHIPPENHAM',20)
3	sql_transaction	2020-12-26 17:04:11.2800000	81	13	346	338484	Rollback	System	INSERT INTO Person VALUES(2,'Khan','Jacob','72 Ballifeary Road','BANCFFOSFELEN',11)

As we can see in the image, the **transaction_type** column shows the **System** value for our queries and it means that these transactions are operated by the SQL Server in the auto-commit transaction mode.

Implicit transaction mode in SQL Server

In the implicit transaction mode, SQL Server takes the responsibility for beginning the transactions implicitly but it waits for the commit or rollback commands from the user. In the implicit transaction mode, the database objects involved in the transaction will remain locked until the commit or rollback commands are executed. In order to use the implicit transaction mode, we need to set implicit transaction mode to ON. We can use the following syntax to enable or disable the implicit transaction mode.

```
SET IMPLICIT_TRANSACTIONS { ON | OFF }
```

The following keywords start a new transaction in the implicit transaction mode if it's enabled.

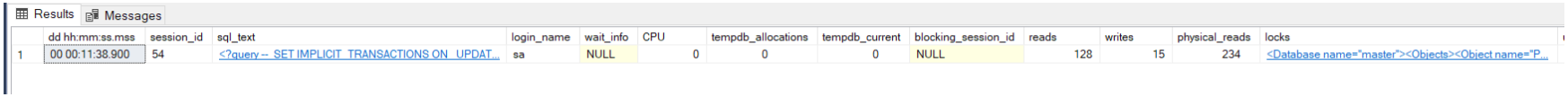
- ALTER TABLE
- BEGIN TRANSACTION
- CREATE
- DELETE
- DROP
- FETCH
- GRANT
- INSERT
- OPEN
- REVOKE
- SELECT
- TRUNCATE TABLE
- UPDATE

Now, let's update any row of the **Person** table to using the implicit transaction in SQL Server. At first, we will enable the implicit transaction and then update the row.

```
SET IMPLICIT_TRANSACTIONS ON
UPDATE
    Person
SET
    Lastname = 'Sawyer',
    Firstname = 'Tom'
WHERE
    PersonID = 2
```

We open another query window and execute the sp_WhoisActive procedure to monitor the locked objects.

```
EXEC sp_WhoisActive
@get_locks = 1;
```



Results Messages

	dd hh:mm:ss.mss	session_id	sql_text	login_name	wait_info	CPU	tempdb_allocations	tempdb_current	blocking_session_id	reads	writes	physical_reads	locks
1	00 00:11:38.900	54	<?query -- SET IMPLICIT_TRANSACTIONS ON UPDAT...	sa	NULL	0	0	0	NULL	128	15	234	<Database name="master"><Objects><Object name="P...

We can find out more details about the locked objects when we click the **locks** column.

```
<Database name="master">
  <Objects>
    <Object name="Person" schema_name="dbo">
      <Locks>
        <Lock resource_type="KEY" index_name="PK__Person__AA2FFB85AFA010D0" request_mode="X" request_status="GRANT" request_count="1" />
        <Lock resource_type="OBJECT" request_mode="IX" request_status="GRANT" request_count="1" />
        <Lock resource_type="PAGE" page_type="*" index_name="PK__Person__AA2FFB85AFA010D0" request_mode="IX" request_status="GRANT" request_count="1" />
      </Locks>
    </Object>
  </Objects>
</Database>
```

As the last step, we have to execute the **COMMIT TRAN** statement to commit the open transaction so the data changes will become permanent.

```
COMMIT TRAN
```

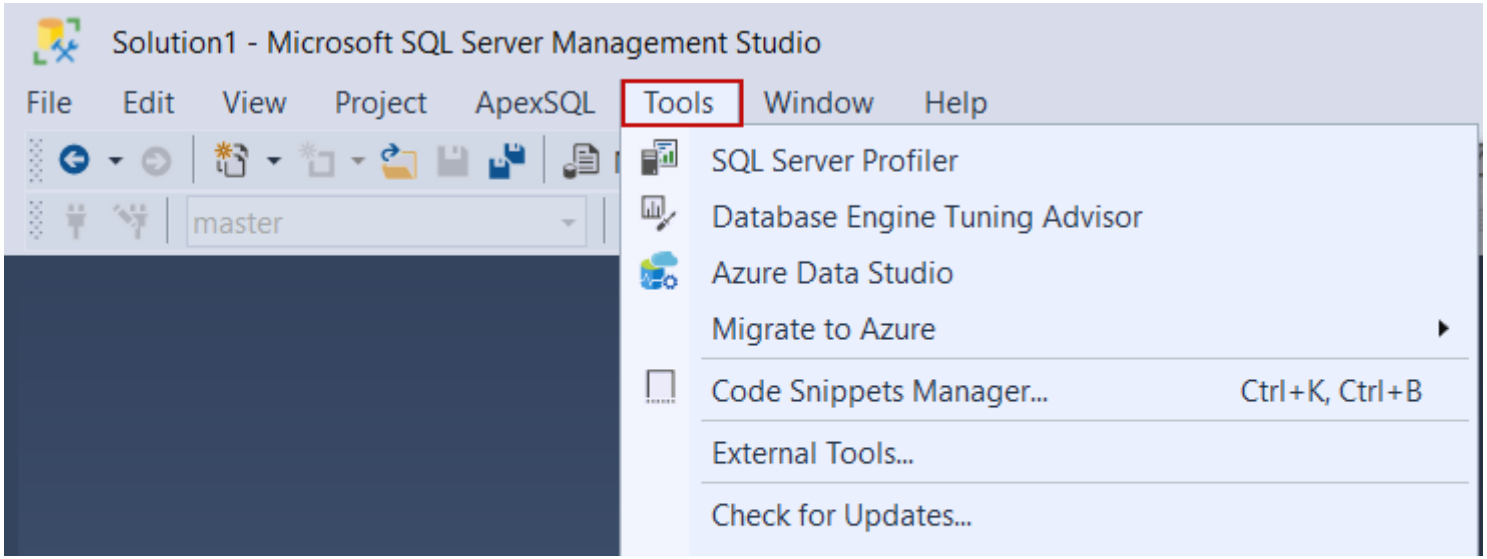
```
1  SET IMPLICIT_TRANSACTIONS ON
2  UPDATE
3    Person
4  SET
5    Lastname = 'Sawyer',
6    Firstname = 'Tom'
7  WHERE
8    PersonID = 2
9  COMMIT TRAN
```

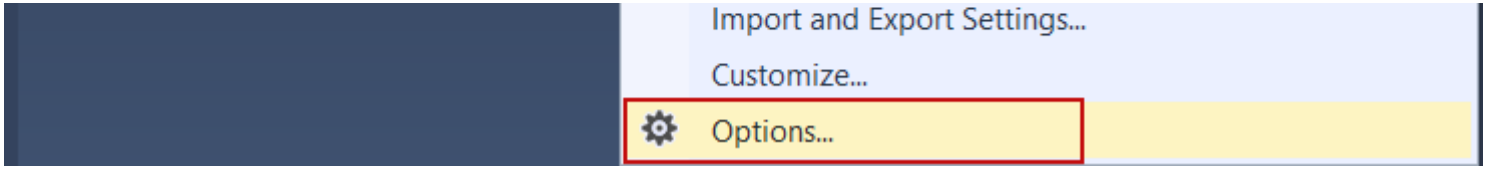
150 %

Messages

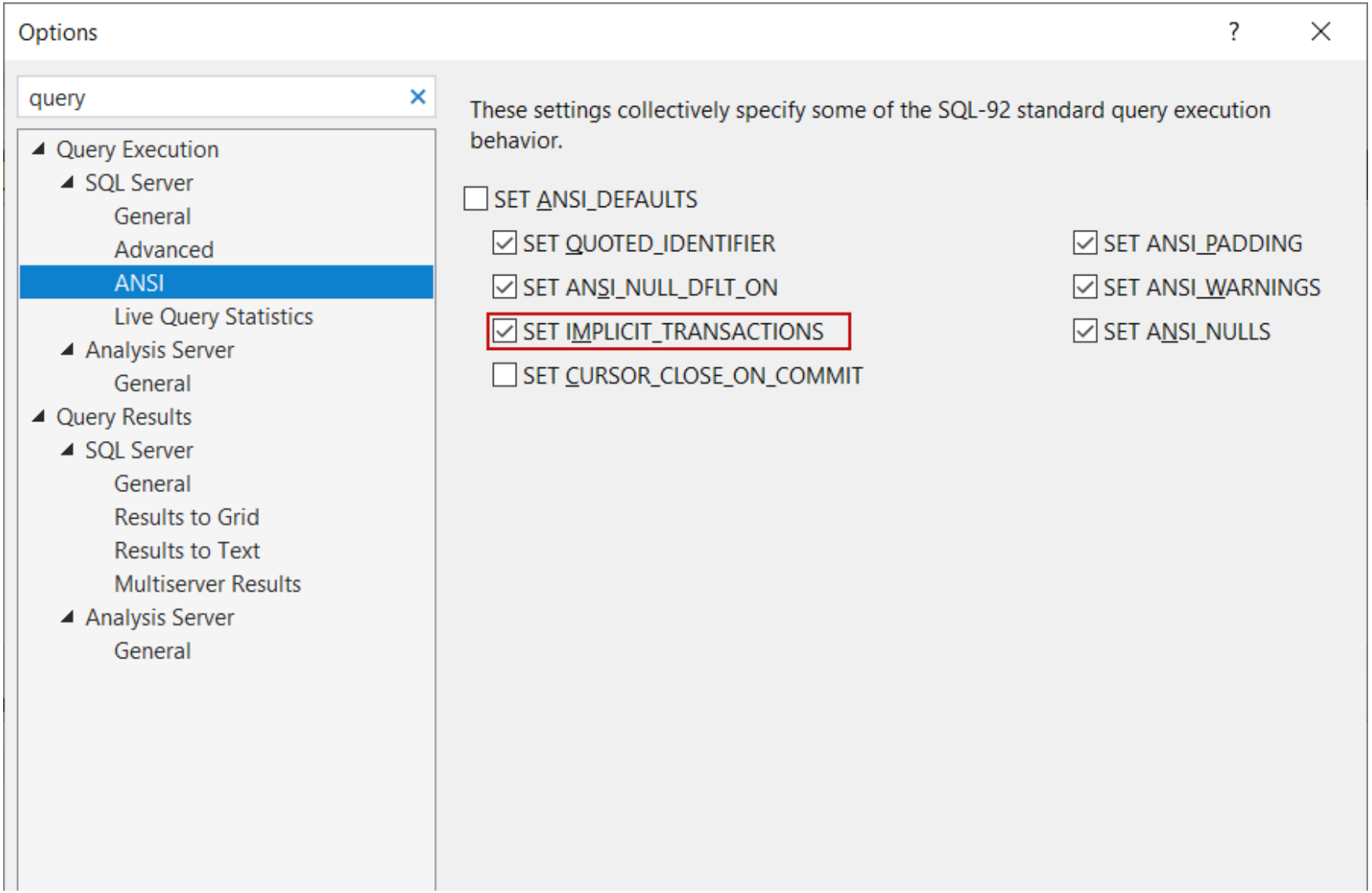
```
Commands completed successfully.

Completion time: 2020-12-27T16:57:39.3151393+03:00
```

The implicit transaction is a connection-level setting and we can set this setting when connecting to the SQL Server. On the SQL Server Management Studio, we can determine the implicit transaction as a default transactions mode for our connections. At first, we click the **Tools** menu and select the **Options**.



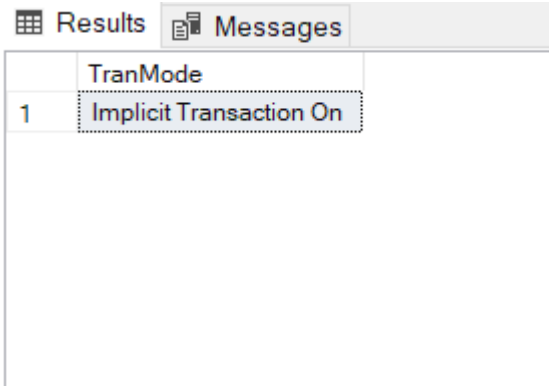


Find the **ANSI** option which is placed under the **Query Execution** tab and check the **SET IMPLICIT TRANSACTIONS** option.



After changing this setting in SSMS, the new connection's default transaction modes will be the implicit transaction. The following query helps to find out the connections transaction mode.

```
SELECT IIF(@@OPTIONS&2 = 0, 'Implicit Transaction Off', 'Implicit Transaction On') AS TranMode;
```



Explicit transaction mode in SQL Server

In the explicit transaction mode, we have to define the starting and ending points of the transactions. It means that all transactions must start with the BEGIN TRANSACTION statement and end with either COMMIT TRANSACTION or ROLLBACK TRANSACTION statements. We can use explicit transactions in SQL Server in its simplest form as below. After committing the transaction the data modification will be persisted in the database.

```
SET IMPLICIT_TRANSACTIONS OFF
BEGIN TRAN;
INSERT INTO Person
```

```
VALUES
(3,
'Bunny',
'Bugs',
'742 Evergreen Terrace',
'Springfield',
54
);
COMMIT TRAN;

SELECT * FROM Person
```

Results Messages

	PersonID	LastName	FirstName	Address	City	Age
1	3	Bunny	Bugs	742 Evergreen Terrace	Springfield	54

When we rollback a transaction the data modifications will be undone.

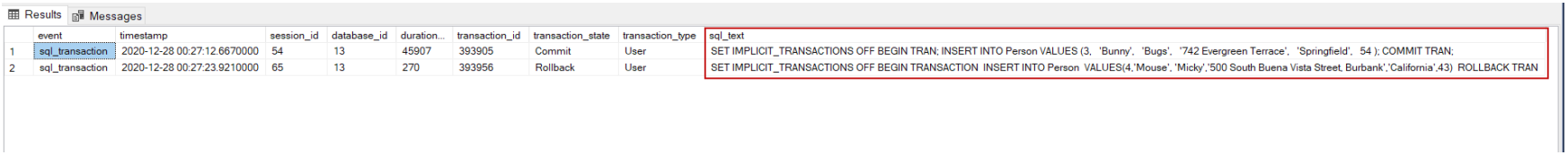
```
SET IMPLICIT_TRANSACTIONS OFF
BEGIN TRANSACTION
INSERT INTO Person
VALUES(4,'Mouse', 'Micky','500 South Buena Vista Street, Burbank','California',43)

ROLLBACK TRAN

SELECT * FROM Person WHERE PersonID=4
```

Results Messages

PersonID	LastName	FirstName	Address	City	Age

When we retrieve data from the extended events, we can see the above insert operations which were executed by us. These insert statements transaction_type will show as User. It means these transactions are managed by the session user instead of the SQL Server

transaction_type will show as **User**. It means these transactions are managed by the session user instead of the SQL Server.



	event	timestamp	session_id	database_id	duration...	transaction_id	transaction_state	transaction_type	sql_text
1	sql_transaction	2020-12-28 00:27:12.6670000	54	13	45907	393905	Commit	User	SET IMPLICIT_TRANSACTIONS OFF BEGIN TRAN; INSERT INTO Person VALUES (3, 'Bunny', 'Bugs', '742 Evergreen Terrace', 'Springfield', 54); COMMIT TRAN;
2	sql_transaction	2020-12-28 00:27:23.9210000	65	13	270	393956	Rollback	User	SET IMPLICIT_TRANSACTIONS OFF BEGIN TRANSACTION INSERT INTO Person VALUES(4,'Mouse', 'Micky','500 South Buena Vista Street, Burbank','California',43) ROLLBACK TRAN

Tip: Generally, it would be a logical option to use explicit transactions with TRY-CATCH blocks.

```
SET IMPLICIT_TRANSACTIONS OFF
BEGIN TRY
BEGIN TRANSACTION
INSERT INTO Person
VALUES(4,'Mouse', 'Micky','500 South Buena Vista Street, Burbank','California',43)

COMMIT TRANSACTION
END TRY
BEGIN CATCH
IF(@@TRANCOUNT > 0)
        ROLLBACK TRAN

END CATCH
```

Differences between the auto-commit and explicit transactions in SQL Server

In this section, we will observe the differences of the auto-commit transaction mode against the explicit transaction mode. At first, we briefly take a glance at the log buffer flush mechanism of the SQL Server. SQL Server writes all modifications into the log buffer and this buffered data is sent into the log file when the following conditions are met:

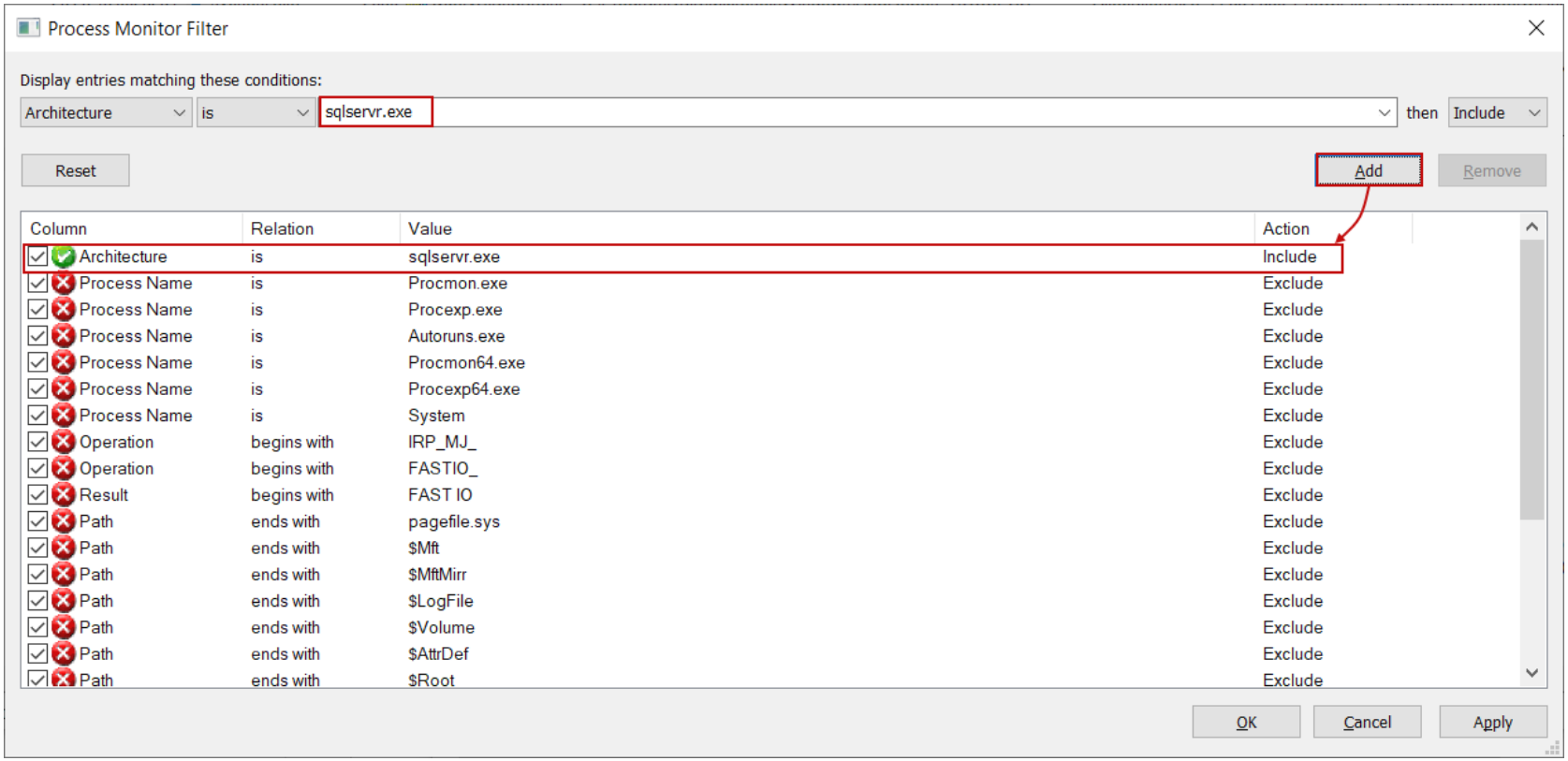
- When a transaction is committed
- The log buffer size reaches 60 KB
- sys.sp_flush_log procedure is executed
- CHECKPOINT process is completed

In order to monitor the difference between two transaction modes in SQL Server, we will use a tool, Process Monitor. Process Monitor is a tool that helps to monitor all activities of processes in the windows based operating systems. At first, we will create the following table and insert this table with 100.000 records.

```
CREATE TABLE InsertSomeRecord
(RowNumber INT, RowNumberString VARCHAR(10))
```

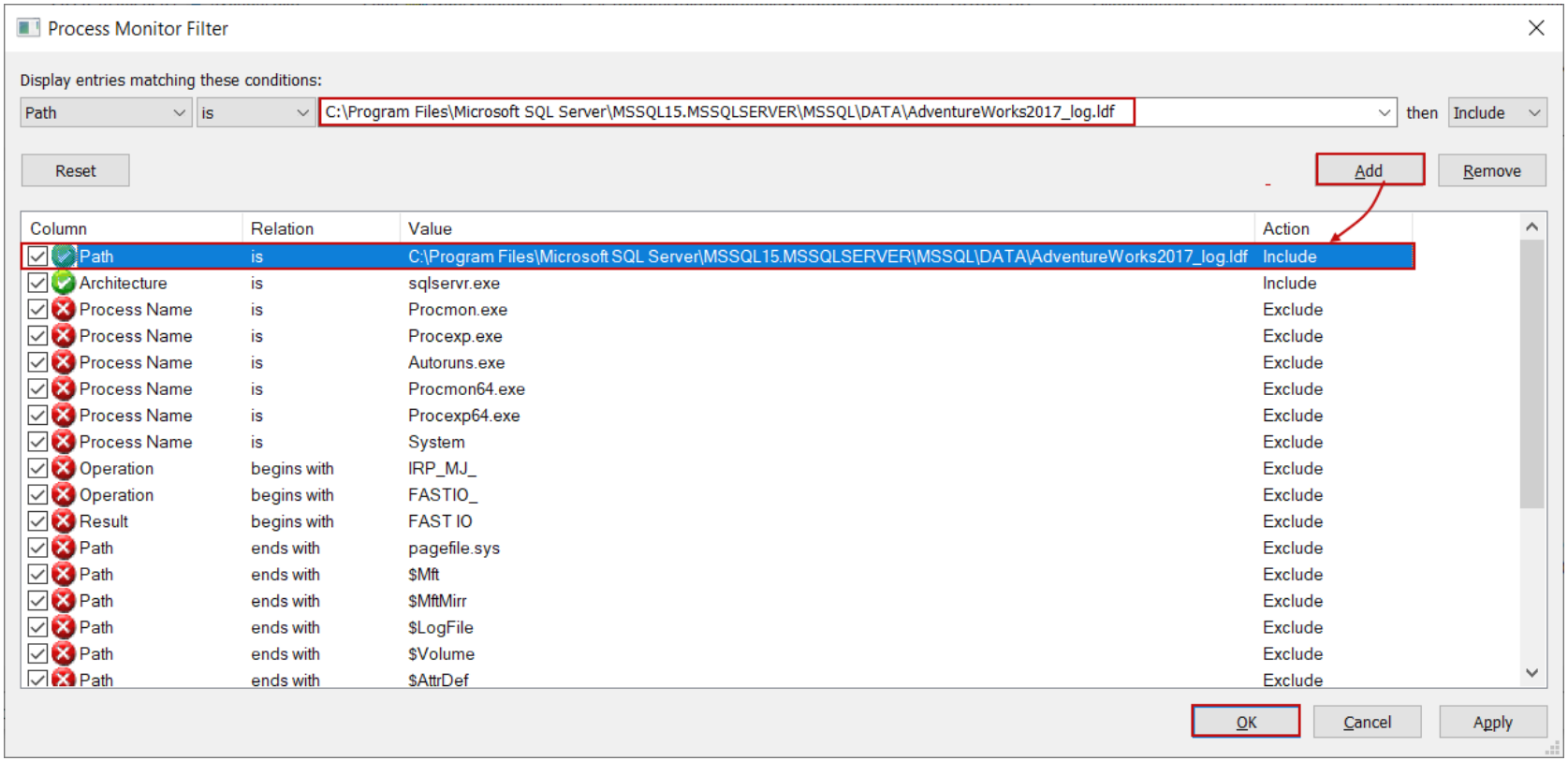
After creating the table, we will launch the Process Monitor and filter the SQL Server engine process and the log file path that the test is performed.

- Click the Ctrl + L key combination in order to open the filtering options
- Select the **Process Name** in the combobox and then type **sqlservr.exe** to capture the only SQL Server process



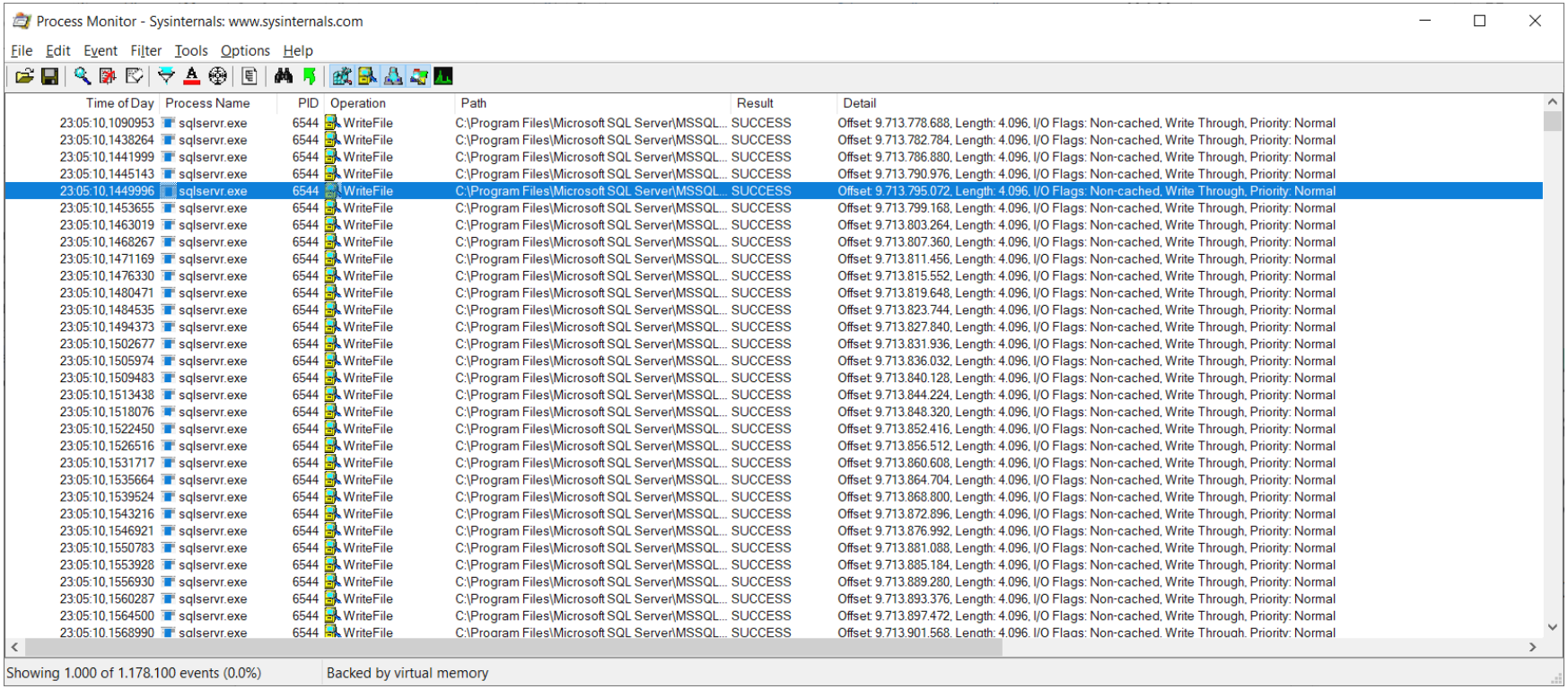
- Select the **Path** in the combobox and then type the log file path that we want to capture activities

- Click the **OK** button



As a first step, we will execute the following query which will insert 1000 rows in auto-commit transaction mode.

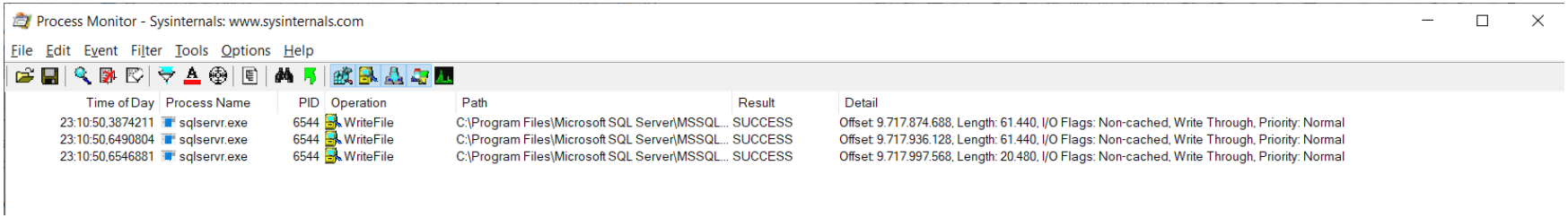
```
DECLARE @Counter INT= 1;
WHILE @Counter <= 1000
    BEGIN
        INSERT INTO InsertSomeRecord
        (RowNumber,
        RowNumberString
        )
        VALUES
        (@Counter,
        'RowNumber=' + CAST(@Counter AS NVARCHAR(7))
        );
        SET @Counter = @Counter + 1;
    END;
```



As we have seen that in the auto-commit transaction mode the log flush process has occurred 1000 times in random sized chunks. For the explicit transaction mode, we will clear the captured process and execute the following query:

```
BEGIN TRAN

DECLARE @Counter INT= 1;
WHILE @Counter <= 1000
    BEGIN
        INSERT INTO InsertSomeRecord
        (RowNumber,
        RowNumberString
        )
        VALUES
        (@Counter,
        'RowNumber=' + CAST(@Counter AS NVARCHAR(7))
        );
        SET @Counter = @Counter + 1;
```

```
    END;
    COMMIT TRAN
```



In the explicit transaction mode, the log buffer size has reached the maximum size and then it flushes into the log file.

Result of the Benchmark: In this benchmark, we have seen how explicit transactions and auto-commit transactions affect the log file activity differently. According to our application and business requirements, we can decide which transaction mode is suitable for us. However, the main point in this comparison, we manage all inserts in one transaction for the explicit mode. In this way, we have decreased the log file activity. When we format the same query like the below, we can not see the same effect on log activity.

```
DECLARE @Counter INT= 1;
WHILE @Counter <= 1000
    BEGIN
    BEGIN TRAN
        INSERT INTO InsertSomeRecord
        (RowNumber,
        RowNumberString
        )
        VALUES
        (@Counter,
        'RowNumber=' + CAST(@Counter AS NVARCHAR(7))
        );
        SET @Counter = @Counter + 1;
            COMMIT TRAN
    END;
```

Conclusion

In this article, we have learned the modes of transactions in SQL Server. Understanding transaction modes are very important because they directly affect data integrity and different transaction modes have different characteristics

direct data integrity and different transaction modes have different characteristics.

See more

Interested in a SQL log reader to view and analyze the SQL Server transaction log? Consider ApexSQL Log, a powerful tool that renders transaction log information into a searchable, sortable information that can be used for forensic auditing, replication, continuous auditing, disaster recovery and more

Esat Erkec

Esat Erkec is a SQL Server professional who began his career 8+ years ago as a Software Developer. He is a SQL Server Microsoft Certified Solutions Expert.

Most of his career has been focused on SQL Server Database Administration and Development. His current interests are in database administration and Business Intelligence. You can find him on LinkedIn.

View all posts by Esat Erkec

Related Posts:

1. **How to rollback using explicit SQL Server transactions**
2. **Transactions in SQL Server for beginners**
3. **Top SQL Server Books**
4. **Term Extraction Transformation in SSIS**
5. **SQL Server Transaction Log Interview Questions**

Locking, Monitoring, Transaction log

14,557 Views

ALSO ON **SQL SHACK**

How to use iterations and conditions …

3 months ago • 4 comments

In this article, we will show how to use the Iterations and Conditions activities …

How to prepare for the Exam DP-201: …

6 months ago • 2 comments

In this article, we will discuss how to prepare yourself for an important …

Three ways you can add tSQLt to your …

5 months ago • 1 comment

This article will show three different methods to integrate tSQLt with SQL …

Identify unused SQL databases

4 months ago • 1 comment

This article will explain two approaches to identify unused SQL databases, …

Set up a local serverless …

7 months ago • 1 comment

This article describes hov develop serverless applications using the …