Unit 5: Transactions, Stored Procedures, and DML Triggers - Discussion

Briefly describe one of your favorite websites. Select just one thing the website allows you to do, and describe the steps you think would be performed by the stored procedure the website code would call when you do that action.

7/19/2014 5:22:01 PM

A stored procedure is basically a scripting/programming method to create reusable code. Rather than having to type and re-type SQL code to perform queries that you use regularly, a stored procedure can be created which saves the code. This saved code can easily be executed whenever you need to run it (Ben-Gan, Sarka, & Talmage, 2012).

One of my favorite websites is ubid.com. And, at the ubid.com website, there is a menu option that allows me to view my account information, and even change it for that matter. The premise of my stored procedure is that when I click the account option, it executes a stored procedure and returns *my* account information–making me, the customer, extremely satisfied.

Logically, my SQL code would start off like this (MSSQLTips, n.d.):

```
SELECT * FROM Customer.Account WHERE CustID=1
```

which would return my user account

CustID	First_Name	Last_Name	Address	ST
1	Eddie	Jackson	123 Kaplan	FL

Next, I would create the stored procedure. The SQL code would look like this:

```
CREATE PROCEDURE spAccount1 @myCustID varchar(10)
AS
SELECT * FROM Customer.Account WHERE CustID=@myCustID
GO
```

This would create the stored procedure named *spAccount1*.

Now, when I want to access just my account, I would type the following SQL code:

```
EXEC spAccount1
```

Notice how I started with just the SELECT statement. This approach verifies that my code is correct, and that what is being returned is actually what I want. Of course, this is only a simple example. Stored procedures can do much more, including having the ability to accept passed parameters.

For example, using ubid as inspiration, a stored procedure could be created to return customers account information for whatever state that was passed to it via a parameter.

The stored procedure code would look something like this:

```
CREATE PROCEDURE spCustomerState @City nvarchar(35)
AS
SELECT *
FROM Customer.Account
WHERE City = @City
GO
```

This stored procedure becomes dynamic…meaning that when you execute the stored procedure, you would specify which state you wanted to return account information for. For example, if I wanted to return all customers from the state of Florida, the code would look like this:

```
EXEC spCustomerState @City = 'Florida'
```

References

Ben-Gan, I., Sarka, D., & Talmage, R. (2012). *Querying Microsoft SQL Server 2012.* Sabastopol, CA: O'Reilly Media, Inc.

MSSQLTips (n.d.). Creating a simple stored procedure. Retrieved from http://www.mssqltips.com/sqlservertutorial/162/how-to-create-a-sql-server-stored-procedure-with-parameters/

7/20/2014 5:59:45 PM

You provided an excellent example of a stored procedure. In fact, it made me want to go back and re-evaluate my original stored procedure.

Originally, I had something like this (notice the CustID is statically set to 1):

CREATE PROCEDURE spAccount1
AS
SELECT * FROM Customer.Account WHERE CustID=1
GO

But then I realized it could be more dynamic if I created a variable, then I could pass that variable back to the procedure. The updated code would look more like this:

CREATE PROCEDURE spAccount1 @myCustID varchar(10)
AS
SELECT * FROM Customer.Account WHERE CustID=@myCustID
GO

I created another good example of a stored procedure (it's working code for the pubs database).

My code:

```
USE pubs;
GO
CREATE PROCEDURE spEmployees -- name of the stored procedure
       @LName nvarchar(25), -- variable name
       @FName nvarchar(25)  -- variable name
AS
        SET NOCOUNT ON; -- suppresses the number of rows message
        SELECT Emp_id, FName, LName -- column names
        FROM employee -- which table
        WHERE FName = @FName AND LName = @LName -- filter
        AND hire_date IS NOT NULL;
GO
```

Output

```
Emp_id      FName      LName
A-C71970F   Aria    Cruz
```

Reference

Microsoft. (2014). Create a stored procedure. Retrieved from http://msdn.microsoft.com/en-us/library/ms345415.aspx

7/21/2014 6:33:47 PM
That is a good beginning Tanya, and I also agree that W3Schools is an excellent reference and learning website. I was thinking we could extend your example into a procedure and explain some of the internal workings.

So, let's create a table using your syntax:

```
CREATE TABLE Students

(
    [ST_ID] [int] IDENTITY(1,1) NOT NULL,
    [FName] [nvarchar](50) NOT   NULL,
    [LName] [nvarchar](50) NOT NULL,
    [EMail] [nvarchar](35)   NULL
)
```

Next, let's insert some data into the table:

```
Insert into Students (FName, LName, EMail)
 Values('Eddie', 'Jackson', 'EJ@kaplan.com')

Insert into Students (FName, LName, EMail)
 Values('Tanya', 'Gallon', 'TG@kaplan.com')
```

Finally, we want to write a stored procedure that returns the student name when the student's ID is given (think library card):

```
CREATE PROCEDURE ReturnStudentName(

@ST_ID INT –This is the input parameter, the ID of the student

)
AS
BEGIN
SELECT FName+' '+LName FROM Students WHERE ST_ID=@ST_ID
END
```

And now we can execute our procedure by running this command:

Execute ReturnStudentName 1

<u>What are triggers?</u>

A trigger is a place to store business rules to fire off actions or code, when an insert, update, or delete is ran.

Can cause performance issues…so you need to know when to use them. Watch out for loops.

Used to enforce data integrity, tracking, and auditing.

Update is a delete followed by an insert

<u>How to use an Update trigger to backup a change (an audit trail)?</u>

```
CREATE TRIGGER uProductPricechange ON products
    FOR UPDATE -- or delete, insert
AS
        INSERT ProductPriceHistory -- where to put the inserted data
                    SELECT
                          NEWID(), p.ProdcuctID, d.price, i.price, etc.
                 -- the above are Column names from insert and deleted tables
                     FROM
                            Products p
                                   JOIN
                             inserted i ON p.ProductID = i.ProductID
                                   JOIN
                             deleted d ON p.ProductID = d.ProductID
GO
```

Dropping a trigger

```
IF OBJECT_ID ('udEmploymentAudit','TR') IS NOT NULL -- means, if it exists,
then delete it
     DROP TRIGGER udEmploymentAudit -- trigger name
GO
```

Update, Delete – creates a Trigger on Employee table to track who deleted and time of delete

```
CREATE TRIGGER dbo.udEmployeeAudit ON Employees -- name of trigger
and table
      FOR UPDATE, DELETE -- notice update and delete are there
AS

         INSERT EmployeeAuditTrail -- where to put the inserted data
                SELECT -- all the below are column names
                       e.EmployeeID, d.FirstName, d.MiddleName,
d.LastName,
                       d.Title, d.Hiredate, d.VacationHours, d.Salary,
                       GETDATE(), SYSTEM_USER -- Grab the current
timestamp and return system user
                FROM Employee e -- Employee table
                                JOIN -- joining the Employee and Eeleted tables
                           deleted d ON e.EmployeeID = d.EmployeeID
GO
```

What are ACID Properties?

Source:
http://blog.sqlauthority.com/2007/12/09/sql-server-acid-atomicity-consistency-isolation-durability/

ACID (an acronymn for Atomicity Consistency Isolation Durability) is a concept that Database Professionals generally look for when evaluating databases and application architectures. For a reliable database all this four attributes should be achieved.

Atomicity is an all-or-none proposition.

Consistency guarantees that a transaction never leaves your database in a half-finished state.

Isolation keeps transactions separated from each other until they’re finished.

Durability guarantees that the database will keep track of pending changes in such a way that the server can recover from an abnormal termination.

What is table truncating?

Source: http://www.tutorialspoint.com/sql/sql-truncate-table.htm

The SQL TRUNCATE TABLE command is used to delete complete data from an existing table.

You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.

Syntax:

The basic syntax of TRUNCATE TABLE is as follows:
TRUNCATE TABLE table_name;

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is the example to truncate:
SQL > TRUNCATE TABLE CUSTOMERS;

Now, CUSTOMERS table is truncated and following would be the output from SELECT statement:
SQL> SELECT * FROM CUSTOMERS;
Empty set (0.00 sec)

What is transaction, the 3 types, and transaction isolation types?

There are three different types of transactions that we can set in SQL Server. These transactions are:

1) Auto Commit Transactions
2) Implicit Transactions
3) Explicit Transactions

(1 of 3) Explicit Transactions

Source: http://technet.microsoft.com/en-us/library/ms175127(v=SQL.105).aspx

An explicit transaction is one in which you explicitly define both the start and end of the transaction.

DB-Library applications and Transact-SQL scripts use the BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK

TRANSACTION, or ROLLBACK WORK Transact-SQL statements to define explicit transactions.

BEGIN TRANSACTION
Marks the starting point of an explicit transaction for a connection.

COMMIT TRANSACTION or COMMIT WORK
Used to end a transaction successfully if no errors were encountered. All data modifications made in the transaction become a permanent part of the database. Resources held by the transaction are freed.

ROLLBACK TRANSACTION or ROLLBACK WORK
Used to erase a transaction in which errors are encountered. All data modified by the transaction is returned to the state it was in at the start of the transaction. Resources held by the transaction are freed.

You can also use explicit transactions in OLE DB. Call the ITransactionLocal::StartTransaction method to start a transaction. Call either the ITransaction::Commit or ITransaction::Abort method with fRetaining set to FALSE to end the transaction without automatically starting another transaction.

In ADO, use the BeginTrans method on a Connection object to start an explicit transaction. To end the transaction, call the Connection object's CommitTrans or RollbackTrans methods.

In the ADO.NET SqlClient managed provider, use the BeginTransaction method on a SqlConnection object to start an explicit transaction. To end the transaction, call the Commit() or Rollback() methods on the SqlTransaction object.

The ODBC API does not support explicit transactions, only autocommit and implicit transactions.

Explicit transaction mode lasts only for the duration of the transaction. When the transaction ends, the connection returns to the transaction mode it was in before the explicit transaction was started, either implicit or autocommit mode.

(2 of 3) Autocommit Transactions

Source: http://technet.microsoft.com/en-us/library/ms187878(v=SQL.105).aspx

Autocommit mode is the default transaction management mode of the SQL Server Database Engine. Every Transact-SQL statement is committed or rolled back when it completes. If a statement completes successfully, it is committed; if it encounters any error, it is rolled back. A connection to an instance of the Database Engine operates in autocommit mode whenever this default mode has not been overridden by either explicit or implicit transactions. Autocommit mode is also the default mode for ADO, OLE DB, ODBC, and DB-Library.

A connection to an instance of the Database Engine operates in autocommit mode until a BEGIN TRANSACTION statement starts an explicit transaction, or implicit transaction is set on. When the explicit transaction is committed or rolled back, or when implicit transaction mode is turned off, the connection returns to autocommit mode.

When ON, SET IMPLICIT_TRANSACTIONS sets the connection to implicit transaction mode. When OFF, it returns the connection to autocommit transaction mode.

Compile and Run-time Errors

In autocommit mode, it sometimes appears as if an instance of the Database Engine has rolled back an entire batch instead of just one SQL statement. This happens if the error encountered is a compile error, not a run-time error. A compile error prevents the Database Engine from building an execution plan, so nothing in the batch is executed. Although it appears that all of the statements before the one generating the error were rolled back, the error prevented anything in the batch from being executed. In the following example, none of the INSERT statements in the third batch are executed because of a compile error. It appears that the first two INSERT statements are rolled back when they are never executed.

```
USE AdventureWorks2008R2;
GO
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));
GO
INSERT INTO TestBatch VALUES (1, 'aaa');
INSERT INTO TestBatch VALUES (2, 'bbb');
INSERT INTO TestBatch VALUSE (3, 'ccc');   -- Syntax error.
GO
SELECT * FROM TestBatch;   -- Returns no rows.
GO
```

In the following example, the third INSERT statement generates a run-time duplicate primary key error. The first two INSERT statements are successful and committed, so they remain after the run-time error.

```
USE AdventureWorks2008R2;
GO
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));
GO
INSERT INTO TestBatch VALUES (1, 'aaa');
INSERT INTO TestBatch VALUES (2, 'bbb');
INSERT INTO TestBatch VALUES (1, 'ccc');   -- Duplicate key error.
GO
SELECT * FROM TestBatch;   -- Returns rows 1 and 2.
GO
```

The Database Engine uses deferred name resolution, in which object names are not resolved until execution time. In the following example, the first two INSERT statements are executed and committed, and those two rows remain in the TestBatch table after the third INSERT statement generates a run-time error by referring to a table that does not exist.

```
USE AdventureWorks2008R2;
GO
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));
GO
INSERT INTO TestBatch VALUES (1, 'aaa');
INSERT INTO TestBatch VALUES (2, 'bbb');
INSERT INTO TestBch VALUES (3, 'ccc');   -- Table name error.
GO
SELECT * FROM TestBatch;   -- Returns rows 1 and 2.
GO
```

(3 of 3) Implicit Transactions

Source: http://technet.microsoft.com/en-us/library/ms188317(v=sql.105).aspx

When a connection is operating in implicit transaction mode, the instance of the SQL Server Database Engine automatically starts a new transaction after the current transaction is committed or rolled back. You do nothing to delineate the start of a transaction; you only commit or roll back each transaction. Implicit transaction mode generates a continuous chain of transactions.

After implicit transaction mode has been set on for a connection, the instance of the Database Engine automatically starts a transaction when it first executes any of these statements:

```
ALTER TABLE
INSERT
CREATE
OPEN
DELETE
REVOKE
```

DROP
SELECT
FETCH
TRUNCATE TABLE
GRANT
UPDATE

The transaction remains in effect until you issue a COMMIT or ROLLBACK statement. After the first transaction is committed or rolled back, the instance of the Database Engine automatically starts a new transaction the next time any of these statements is executed by the connection. The instance keeps generating a chain of implicit transactions until implicit transaction mode is turned off.

Implicit transaction mode is set either using the Transact-SQL SET statement, or through database API functions and methods.

What is RAISERROR and THROW?
Raiserror is a way to throw an error in SQL

Generates an error message and initiates error processing for the session. RAISERROR can either reference a user-defined message stored in the sys.messages catalog view or build a message dynamically. The message is returned as a server error message to the calling application or to an associated CATCH block of a TRY…CATCH construct. New applications should use THROW instead.

Applies to: SQL Server (SQL Server 2008 through current version), Azure SQL Database.

Topic link icon Transact-SQL Syntax Conventions

Syntax

```
RAISERROR ( { msg_id | msg_str | @local_variable }
    { ,severity ,state }
    [ ,argument [ ,...n ] ] )
    [ WITH option [ ,...n ] ]
```

Example 1 of 2

```
DECLARE @DBID INT;
SET @DBID = DB_ID();

DECLARE @DBNAME NVARCHAR(128);
SET @DBNAME = DB_NAME();

RAISERROR
    (N'The current database ID is:%d, the database name is: %s.',
    10, -- Severity.
    1, -- State.
    @DBID, -- First substitution argument.
    @DBNAME); -- Second substitution argument.
GO
```

Example 2 of 2

```
BEGIN TRY
   -- RAISERROR with severity 11-19 will cause execution to
   -- jump to the CATCH block
   RAISERROR ('Error raised in TRY block.', -- Message text.
            16, -- Severity.
            1 -- State.
            );
END TRY
BEGIN CATCH
   DECLARE @ErrorMessage NVARCHAR(4000);
   DECLARE @ErrorSeverity INT;
   DECLARE @ErrorState INT;

   SELECT @ErrorMessage = ERROR_MESSAGE(),
        @ErrorSeverity = ERROR_SEVERITY(),
        @ErrorState = ERROR_STATE();

   -- Use RAISERROR inside the CATCH block to return
   -- error information about the original error that
   -- caused execution to jump to the CATCH block.
   RAISERROR (@ErrorMessage, -- Message text.
            @ErrorSeverity, -- Severity.
            @ErrorState -- State.
            );
END CATCH;
```

What are Stored Procedures?

Rather than have the Transact-**SQL** statements recompiled each time, you can create a temporary **stored procedure** that is compiled on the first execution, and then execute the precompiled plan multiple times. Heavy use of temporary **stored procedures**, however, can lead to contention on the system tables in tempdb.

Stored Procedures are a batch of SQL statements that can be executed in a couple of ways. Most major DBMs support stored procedures; however, not all do. You will need to verify with your particular DBMS help documentation for specifics. As I am most familiar with SQL Server I will use that as my samples.

Sample 1 of 2

To create a stored procedure the syntax is fairly simple:

```
CREATE PROCEDURE <owner>.<procedure name>

       <Param> <datatype>

AS

       <Body>
```

So for example:

```
CREATE PROCEDURE Users_GetUserInfo

     @login nvarchar(30)=null

AS

     SELECT * from [Users]
     WHERE ISNULL(@login,login)=login
```

A benefit of stored procedures is that you can centralize data access logic into a single place that is then easy for DBA's to optimize. Stored procedures also have a security benefit in that you can grant execute rights to a stored procedure but the user will not need to have read/write permissions on the underlying tables. This is a good first step against SQL Injection.

Stored procedures do come with downsides, basically the maintenance associated with your basic CRUD operation. Let's say for each table you have an Insert, Update, Delete and at least one Select based on the Primary key that means each table will have 4 procedures. Now take a decent size database of 400 tables, and you have 1600 procedures! And that's assuming you don't have duplicates which you probably will.

This is where using an ORM or some other method to auto generate your basic CRUD operations has a ton of merit.

Sample 2 of 2

Stored Procedure is a Set precompiled SQL statements that used to perform a special Task

Example:- if I have Employee Table

Employee ID	Name	Age	Mobile
001	Sidheswar	25	9938885469
002	Pritish	32	9178542436

First, I am retrieving the Employee table

```
Create Procedure Employee details
As
Begin
Select * from Employee
End
```

To run the Procedure in SQL Server

```
Execute     Employee details
```

--- (Employee details is a user defined name, give a name as you want)

Then Second, I am inserting value to Employee Table

```
Create Procedure employee_insert (@EmployeeID     int,@Name Varchar (30), @Age
int, @Mobile int)
As
Begin
Insert In to Employee Values (@EmployeeID, @Name, @Age, @Mobile )
End
```

To Run the parameterized Procedure in SQL Server

```
Execute employee_insert 003,’xyz’,27,1234567890
```

-- Parameter Size must be Same with declared Column Size

What is autocommit and explicit transaction mode?

Source:
http://www.mstecharticles.com/2013/03/sql-2005-understanding-auto-commit.html

1. Introduction

In SQL Server we call a complete set of action as ***transaction***. The complete set of action together forms a meaningful change to the database. Say for Example, you are transferring money from one bank account to another one. This involves the below specified actions:

1) Create a withdraw entry in the First back account
2) Reduce the Balance in your First Bank account
3) Create a Deposit entry in the second bank entry
4) Increment the Balance in the Second bank account.

Consider first bank account as Savings Account and second bank account as Current Account within the same bank. Now we say the bank website will consider all these four actions together as ***Money Transfer Transaction.*** Even a single action fails the entire transaction get cancelled to avoid un-matched records for the account holder.

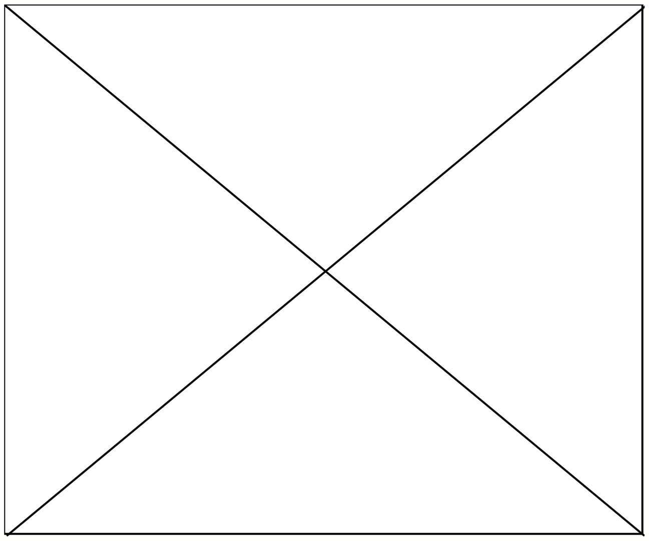
In this article, we will explore how to use transactions in SQL server.

There are three different types of transactions that we can set in SQL Server. These transactions are:
1) Auto Commit Transactions
2) Implicit Transactions
3) Explicit Transactions

2. Auto Commit Transactions

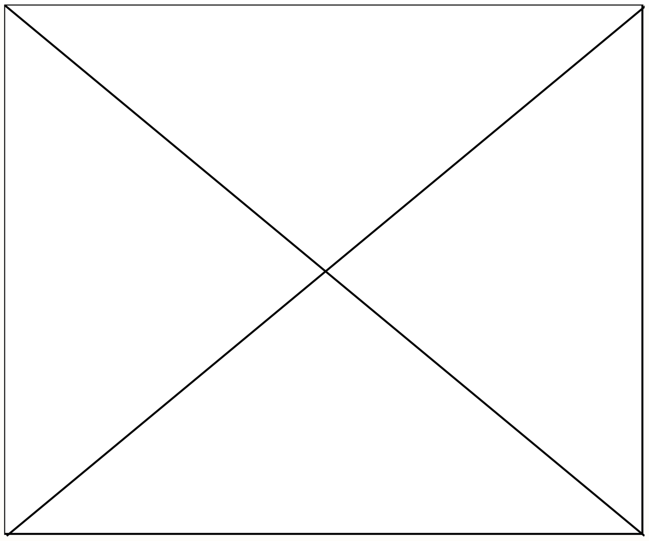
In auto commit transaction mode, SQL Server immediately commits the change after executing the statement. That’s why we call this as Auto commit transaction. Auto commit transaction is the ***default transaction*** mode in SQL Server.

Have a look at the video 1.



In the video you can observe that we executed three update statements by highlighting each update statement one by one. The second update statement is made wrong intentionally. After executing all three statements, the queries result shows first and third update persisted to the database. Here, after executing each statement, the change is committed to the database automatically. That why we call this as ***Auto Commit Transaction,*** which is the default transaction mode in SQL server 2005.

Now, have look at the video 2.



The script is same. We have same three update statements with a same spell mistake in the second update statement. However, this time we update all three update statements at once, by selecting all three and hitting the executing button. We call this executing the statements as a ***batch***. Since the second update statement in the batch is failed, none of the rows updated is recorded to the database. In this case, a Rollback is performed because the batch failure (2nd update).

The ***Go*** statement actually forms the batch of SQL statement. We can write the script for video 1 and video 2 (The way the update statements are executed) as shown below:

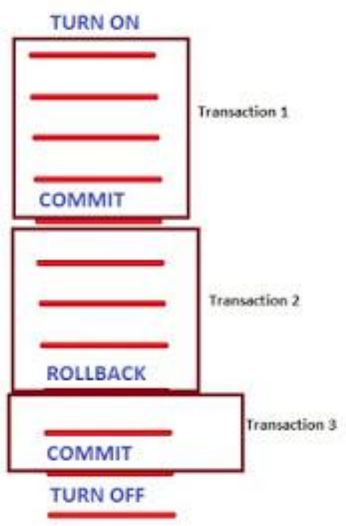
```
--Example 1
--Batch 1
Update Authors Set Au_LName = 'Black' where Au_Id = '172-32-1176';
Go
--Batch 2
Update Authors Ser Au_LName = 'Voyer' where Au_Id = '213-46-8915';
Go
--Batch 3
Update Authors Set Au_LName = 'Peterson' where Au_Id = '238-95-7766';
Go

--Batch 4
Select Au_Id, Au_Fname + ',' + Au_Lname as Name,
Phone, City from Authors where Au_id in ('172-32-1176',
        '213-46-8915','238-95-7766');
Go

--Batch 5
Update Authors Set Au_LName = 'White' where Au_Id = '172-32-1176';
Update Authors Set Au_LName = 'Green' where Au_Id = '213-46-8915';
Update Authors Set Au_LName = 'Carson' where Au_Id = '238-95-7766';
Go
```

```
--Example 2
--Treat All three as a Single Batch.
--Batch 1
Update Authors Set Au_LName = 'Black' where Au_Id = '172-32-1176';
Update Authors Ser Au_LName = 'Voyer' where Au_Id = '213-46-8915';
Update Authors Set Au_LName = 'Peterson' where Au_Id = '238-95-7766';
Go

--Batch 2
Select Au_Id, Au_Fname + ',' + Au_Lname as Name,
Phone, City from Authors where Au_id in ('172-32-1176',
        '213-46-8915','238-95-7766');
Go

--Batch 3
Update Authors Set Au_LName = 'White' where Au_Id = '172-32-1176';
Update Authors Set Au_LName = 'Green' where Au_Id = '213-46-8915';
Update Authors Set Au_LName = 'Carson' where Au_Id = '238-95-7766';
Go
```

From the above statements, I can say that SQL server performs a Commit or Rollback after the Go statement is encountered. Hence, in the Auto commit transaction we don't have to control the commit and rollback as it is taken care by the SQL Server. Also note that the roll back will not be applied when SQL Server encounters a Runtime error. Imagine that in example 2, instead of the misspelled keyword set, we have the correct Set keyword. But, we violate any of the constraint (Say Primary key constraint) in the second statement. In this case, SQL server commits first and third statement.

3. Implicit Transactions

In this transaction Mode, we can control the ***Rollback*** and the ***commit*** operation. A new transaction automatically begins after the commit/Rollback. We can turn ON and Turn OFF this transaction mode based on the need.



In the above picture, two commit and one rollback statement are executed after turning on the Implicit Transaction. As already told after a commit or rollback a new transaction gets started and that is how you see three transactions in the above depiction. Once you turn OFF the implicit transaction, the transaction will be set back to the default Auto Commit transaction.

Use the below SQL statements to Turn-ON or Turn-OFF the implicit transaction:

```
Set Implicit_Transactions On;
Set Implicit_Transactions Off;
```

Now Have a look at the below given example:

```
--Example 3
Set Implicit_Transactions On;
--Transaction 1
Update Authors Set Au_LName = 'Black' where Au_Id = '172-32-1176';
commit;

--Transaction 2
Update Authors Set Au_LName = 'Voyer' where Au_Id = '213-46-8915';
Rollback;

--Transaction 3
Update Authors Set Au_LName = 'Peterson' where Au_Id = '238-95-7766';
commit;

--Select the Authors, to see the effect of Update statement
Select Au_Id, Au_Fname + ',' + Au_Lname as Name,
Phone, City from Authors where Au_id in ('172-32-1176',
       '213-46-8915','238-95-7766');


--Revert back to Original
Update Authors Set Au_LName = 'White' where Au_Id = '172-32-1176';
Update Authors Set Au_LName = 'Green' where Au_Id = '213-46-8915';
Update Authors Set Au_LName = 'Carson' where Au_Id = '238-95-7766';
Go

Set Implicit_Transactions Off;
```

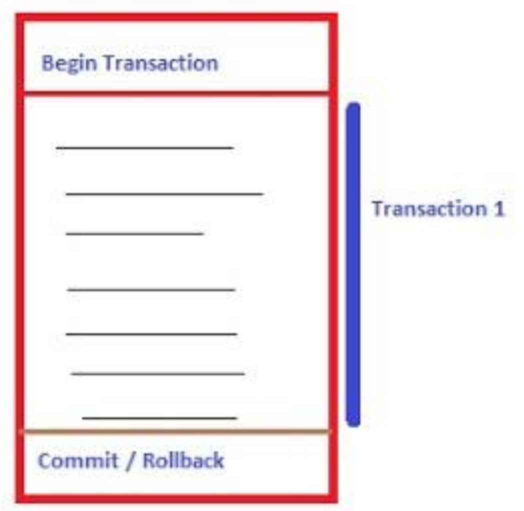
In this example, three update statements are executed followed by either a Rollback or a commit. Note that, here we have the Freedom of specifying the where we want to do a commit or a roll back which we cannot do in the default Auto Commit Transaction mode. Also be aware that even though we can specify where we want to end the transaction, it is not possible to control the starting of the transaction.

After the first Update statement, we asked for a commit, so the update of author's last name as ***Black*** is committed to the database and new transaction is started after that. The second Transaction comes to an End when the rollback statement is executed. The update of author's last name (as ***Voyer***) kept in the memory is discarded after the rollback and one more new Transaction is started. Note that at the end of the script we turned off the implicit transaction. Once the implicit transaction is turned off, the mode changes to default ***Auto Commit*** transaction mode.

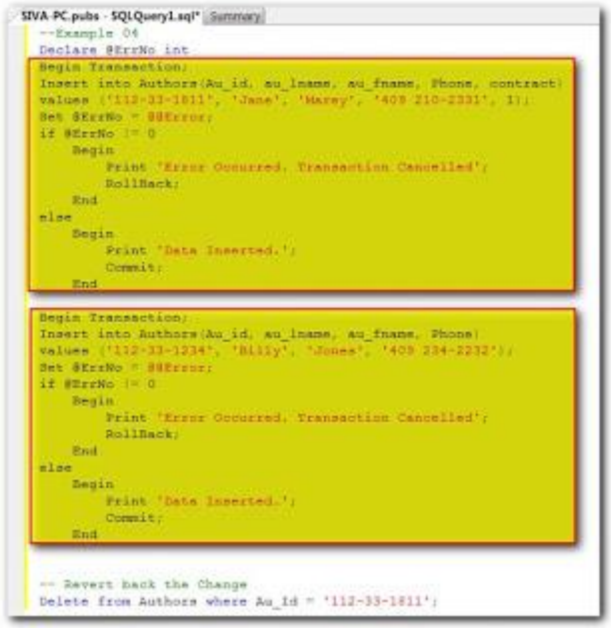
OK. What is the Final result is executing this script? The first and last update statement committed to the database and the second update is discarded.

4. Explicit Transactions

In explicit transaction, we control the starting and ending of the transaction. Look at the below Pic:

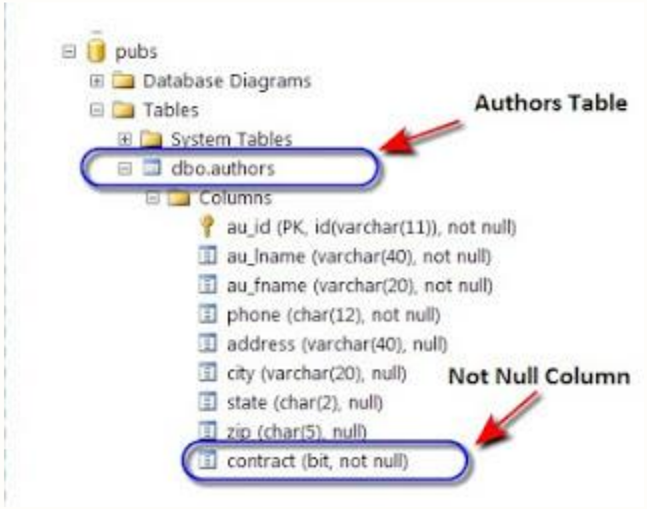


As we control the starting and ending of the transaction, this transaction type is widely used while making the compiled unit of code Say for Ex. Stored Procedures. Have a look at the below specified code:



```
--Example 04
Declare @ErrNo int
Begin Transaction;
Insert into Authors(Au_id, au_lname, au_fname, Phone, contract)
values ('112-33-1811', 'Jane', 'Marey', '408 210-2331', 1);
Set @ErrNo = @@Error;
if @ErrNo != 0
    Begin
        Print 'Error Occurred. Transaction Cancelled';
        RollBack;
    End
else
    Begin
        Print 'Data Inserted.';
        Commit;
    End

Begin Transaction;
Insert into Authors(Au_id, au_lname, au_fname, Phone)
values ('112-33-1234', 'Billy', 'Jones', '408 234-2232');
Set @ErrNo = @@Error;
if @ErrNo != 0
    Begin
        Print 'Error Occurred. Transaction Cancelled';
        RollBack;
    End
else
    Begin
        Print 'Data Inserted.';
        Commit;
    End


-- Revert back the Change
Delete from Authors where Au_Id = '112-33-1811';
```

Here we started two transactions and note that the transaction will end when there is a commit or rollback. In this example the first transaction will get succeeded and the "Data Inserted message” will appear in the message area while executing the script. The second Transaction fails as Not Null column Contract is skipped from the insert (Note skipping a column means we like to leave the column as null) statement.



As this is a constraint violation and this happens at runtime the @@Error in our script returns a Non-Zero Error number. When the statement execution succeeds without any problem, we do get zero in the @@Error environment variable. The remaining portion of the script is understandable. If you need complete example, you can take that from below:

```
--Example 04
Declare @ErrNo int
Begin Transaction;
Insert into Authors(Au_id, au_lname, au_fname, Phone, contract)
values ('112-33-1811', 'Jane', 'Marey', '409 210-2331', 1);
Set @ErrNo = @@Error;
if @ErrNo != 0
        Begin
            Print 'Error Occurred. Transaction Cancelled';
            RollBack;
        End
else
        Begin
            Print 'Data Inserted.';
            Commit;
        End

Begin Transaction;
Insert into Authors(Au_id, au_lname, au_fname, Phone)
values ('112-33-1234', 'Billy', 'Jones', '409 234-2232');
Set @ErrNo = @@Error;
if @ErrNo != 0
        Begin
```

```
            Print 'Error Occurred. Transaction Cancelled';
            RollBack;
      End
else
      Begin
            Print 'Data Inserted.';
            Commit;
      End


-- Revert back the Change
Delete from Authors where Au_Id = '112-33-1811';
```

What is Isolation?

Source: wikipedia

In database systems, isolation determines how transaction integrity is visible to other users and systems. For example, when a user is creating a Purchase Order and has created the header, but not the PO lines, is the header available for other systems/users, carrying out concurrent operations (such as a report on Purchase Orders), to see?

A lower isolation level increases the ability of many users to access data at the same time, but increases the number of concurrency effects (such as dirty reads or lost updates) users might encounter. Conversely, a higher isolation level reduces the types of concurrency effects that users may encounter, but requires more system resources and increases the chances that one transaction will block another.[1]

It is typically defined at database level as a property that defines how/when the changes made by one operation become visible to other, but on older systems may be implemented systemically, for example through the use of temporary tables. In two-tier systems, a TP manager is required to maintain isolation. In n-tier systems (such as multiple websites attempting to book the last seat on a flight) a combination of stored procedures and transaction management is required to commit the booking and confirm to the customer.[2]

Isolation is one of the ACID (Atomicity, Consistency, Isolation, Durability) properties.

Isolation Levels

Of the four ACID properties in a DBMS (Database Management System), the isolation property is the one most often relaxed. When attempting to maintain the highest level of isolation, a DBMS usually acquires locks on data or implements multiversion concurrency control, which may result in a loss of concurrency. This requires adding logic for the application to function correctly.

Most DBMSs offer a number of *transaction isolation levels*, which control the degree of locking that occurs when selecting data. For many database applications, the majority of database transactions can be constructed to avoid requiring high isolation levels (e.g. SERIALIZABLE level), thus reducing the locking overhead for the system. The programmer must carefully analyze database access code to ensure that any relaxation of isolation does not cause software bugs that are difficult to find. Conversely, if higher isolation levels are used, the possibility of deadlock is increased, which also requires careful analysis and programming techniques to avoid.

The isolation levels defined by the ANSI/ISO SQL standard are listed as follows.

Serializable[edit]

This is the *highest* isolation level.

With a lock-based concurrency control DBMS implementation, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction. Also *range-locks* must be acquired when a SELECT query uses a ranged *WHERE* clause, especially to avoid the *phantom reads* phenomenon (see below).

When using non-lock based concurrency control, no locks are acquired; however, if the system detects a *write collision* among several concurrent transactions, only one of them is allowed to commit. See *snapshot isolation* for more details on this topic.

Repeatable reads[edit]

In this isolation level, a lock-based concurrency control DBMS implementation keeps read and write locks (acquired on selected data) until the end of the transaction. However, *range-locks* are not managed, so ***phantom reads*** can occur.

Read committed[edit]

In this isolation level, a lock-based concurrency control DBMS implementation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed (so the *non-repeatable reads* phenomenon can occur in this isolation level, as discussed below). As in the previous level, *range-locks* are not managed.

Putting it in simpler words, read committed is an isolation level that guarantees that any data read is committed at the moment it is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. It makes no promise whatsoever that if the transaction re-issues the read, it will find the same data; data is free to change after it is read.

Read uncommitted[edit]

This is the *lowest* isolation level. In this level, ***dirty reads*** are allowed, so one transaction may see *not-yet-committed* changes made by other transactions.

Since each isolation level is stronger than those below, in that no higher isolation level allows an action forbidden by a lower one, the standard permits a DBMS to run a transaction at an isolation level stronger than that requested (e.g., a "Read committed" transaction may actually be performed at a "Repeatable read" isolation level).

What is READ COMMITED?

When data is read inside a transaction, any changes that have been made by that transaction are visible - within that transaction only (although READ UNCOMMITTED changes this). So above, even though you've started a second, nested, transaction, you're still in scope of the first transaction and can thus read changed data and get 'the changed values'.

Another transaction, on a separate SPID for example, would block if it was using READ COMMITTED and attempted to read this data.

Source: Wikipedia

In this isolation level, a lock-based concurrency control DBMS implementation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed (so the *non-repeatable reads* phenomenon can occur in this isolation level, as discussed below). As in the previous level, *range-locks* are not managed.

Putting it in simpler words, read committed is an isolation level that guarantees that any data read is committed at the moment it is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. It makes no promise whatsoever that if the transaction re-issues the read, it will find the same data; data is free to change after it is read.

What is SERIALIZED?

Source: Wikipedia

This is the *highest* isolation level.

With a lock-based concurrency control DBMS implementation, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction. Also *range-locks* must be acquired when a SELECT query uses a ranged *WHERE* clause, especially to avoid the *phantom reads* phenomenon (see below).

When using non-lock based concurrency control, no locks are acquired; however, if the system detects a *write collision* among several concurrent transactions, only one of them is allowed to commit. See *snapshot isolation* for more details on this topic.

Transactions, Stored Procedures, and DML Triggers

This unit will cover transactions and their ACID properties and isolation levels, stored procedures and their uses, and DML triggers. The common theme in this unit is maintaining data consistency.

Announcement

In Unit 4, we studied creating views and inline table-valued functions, and modifying data using INSERT, UPDATE, DELETE, and TRUNCATE TABLE

.

In Unit 5, you will learn to

- Describe the tradeoff between transaction isolation and concurrency.
- Create a stored procedure to meet a business need.
- Create a DML trigger to maintain data consistency.

In other words, you will be learning about isolation (one process not stepping on another during a data modification), concurrency (the ability of SQL Server to support the activity of thousands of users at the same time), creating reusable code in the form of a stored procedure, and using a trigger – a business rule enforcement mechanism that can be used when table constraints do not suffice.

The **Unit 5 Seminar** will attempt to cover all of: transactions, isolation levels, locking and blocking, stored procedures, and DML triggers.

```
Create procedure sp_add_customer (@custid integer, @cust_fname varchar(50),
@cust_lname varchar(50))
as
Insert customer (custid, cust_fname, cust_lname) values (@custid, @cust_fname,
@cust_lname)
```

After as use the word Begin and after insert use the word End

```
Create procedure sp_add_customer (@custid integer, @cust_fname varchar(50),
@cust_lname varchar(50))
as
BEGIN
insert customer (custid, cust_fname, cust_lname) values (@custid, @cust_fname,
@cust_lname)
END
```

Outcomes

After completing this unit, you should be able to:

- Describe the tradeoff between transaction isolation and concurrency.
- Create a stored procedure to meet a business need.
- Create a DML trigger to maintain data consistency.

Course outcome(s) practiced in this unit:

IT526-2: Demonstrate the modification of data.

IT526-4: Describe principles of data isolation and consistency.

What do you have to do in this unit?

- Complete assigned Reading.
- Participate in Seminar or complete alternative assignment.
- Participate in Discussion.
- Complete unit Assignment.

Reading

Read Chapter 12 Lesson 1, pages 411–435. This lesson describes the ACID properties of a transaction, the three transaction types, and the transaction isolation levels.

Read the "Locking, Blocking, and Isolation Levels" document in Doc Sharing.

Read Chapter 12 Lesson 2, about RAISERROR and THROW, pages 435–439. Understanding the use of RAISERROR and THROW will help you understand the code in Chapter 13.

Read Chapter 13 Lessons 1 and 2, pages 469–500. These two lessons describe the coding and purposes of stored procedures and DML triggers.

Attending live Seminars is important to your academic success, and attendance is highly recommended. The Seminar allows you to review the important concepts presented in each unit, discuss work issues in your lives that pertain to these concepts, ask your instructor questions, and allow you to come together in real time with your fellow classmates. There will be a graded Seminar in Units 1, 2, 3, 5, and 6 in this course. You must either attend the live Seminar or you must complete the Seminar alternative assignment in order to earn points for this part of the class.

ELITE
ESJ

Option 1

Transactions and Routines

The Unit 5 Seminar will cover:

- Transactions
- Isolation levels
- Locking and blocking
- Writing stored procedures
- DML triggers

Option 2- Alternative Assignment:

You will benefit most from attending the graded Seminar as an active participant. However, if you are unable to attend you have the opportunity to make up the points by completing the alternative assignment.

Write a 200-word summary of the Seminar. Include an example of when a DML trigger would be used. Your paper should be in APA format and cite all references used. Submit to the Seminar Dropbox.

ID: IT526-05-09-

Assignment:

Course outcomes addressed in this activity:

IT526-2: Demonstrate the modification of data.

IT526-4: Describe principles of data isolation and consistency.

Part 1. Questions **(50 points)**

Each answer should be about 100 to 300 words.

a. Compare and contrast autocommit mode with explicit transaction mode. (10)
b. Compare and contrast a DML trigger with a stored procedure. (15)
c. Compare and contrast a DML trigger with either a foreign key or a check constraint. Explain why use of a constraint (if possible) is preferred to use of a trigger. (15)

d. Explain why READ COMMITTED isolation allows more concurrency than SERIALIZABLE. (10)

Part 2. Create routines **(50 points)**

a. You want to make sure that rows in the Sales.Orders table (TSQL2012 database) are archived when deleted. You have created the table Sales.OrdersArchive that has the same columns / data types as Sales.Orders, plus one additional column, Archived, of type datetime, to store the date and time the row is written to the archive table. Archived has a default value of CURRENT_TIMESTAMP.
Because of the foreign key constraint in the Sales.OrderDetails table on the orderid column, you know you cannot delete a row from Orders without first deleting all rows with the same orderid value from OrderDetails. You create a table Sales.OrderDetailsArchive that has the same columns / data types as Sales.OrderDetails, plus the Archived column of type datetime with default value CURRENT_TIMESTAMP.
To solve the deletion problem, create an INSTEAD OF trigger, Sales.tr_ArchiveOrders, that watches the Sales.Orders table for a DELETE and instead does the following:

1. Copies all relevant rows from Sales.Orders to Sales.OrdersArchive
2. Copies all relevant rows from Sales.OrderDetails to Sales.OrderDetailsArchive
3. Deletes those rows from Sales.OrderDetails
4. Finally, deletes the relevant rows from Sales.Orders

If you want to create the two Archive tables to test your trigger, see the scripts in the "Unit5.Create Archive Tables.sql" file in Doc Sharing. (25)

b. In the ***pubs*** database, create a stored procedure that will INSERT an employee. You can develop the procedure in the same way as the procedure in Exercise 2 starting on page 486 is developed. Your finished procedure should have all relevant parameter testing. Only the finished CREATE PROC code should be given in your Assignment document. Also show a call to the procedure that would insert an employee. (25)

Review the grading rubric below before beginning this activity.

100-point Assignment grading rubric

Project Requirements / criterion	**Points Possible**	**Points earned by student**
1.Questions:		
a and d are 10 points each	0–50	
b and c are 15 points each		

2.Create routines:	
a. CREATE TRIGGER – 25 points	0–50
b. CREATE PROCEDURE – 25 points	
Total (Sum of all points)	
Points deducted for spelling, grammar, and/or APA errors.	
Adjusted total points	