SQL Server deadlock example

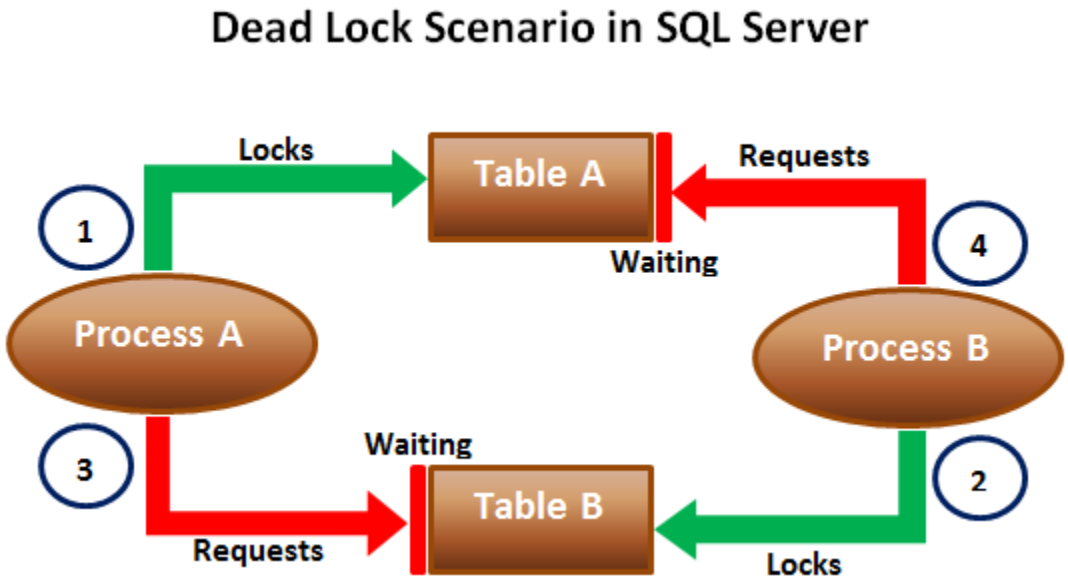
Suggested Videos

Part 75 - Snapshot isolation level in sql server
Part 76 - Read committed snapshot isolation level in sql server
Part 77 - Difference between snapshot isolation and read committed snapshot

In this video we will discuss a scenario when a deadlock can occur in SQL Server.

When can a deadlock occur

In a database, a deadlock occurs when two or more processes have a resource locked, and each process requests a lock on the resource that another process has already locked. Neither of the transactions here can move forward, as each one is waiting for the other to release the lock. The following diagram explains this.



When deadlocks occur, SQL Server will choose one of processes as the deadlock victim and rollback that process, so the other process can move forward. The transaction that is chosen as the deadlock victim will produce the following error.

Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 57) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

Let us look at this in action. We will use the following 2 tables for this example.

Table A	
Id	Name
1	Mark

Table B	
Id	Name
1	Mary

SQL script to create the tables and populate them with test data

```
Create table TableA
(
```

```
    Id int identity primary key,
    Name nvarchar(50)
)
Go

Insert into TableA values ('Mark')
Go

Create table TableB
(
    Id int identity primary key,
    Name nvarchar(50)
)
Go

Insert into TableB values ('Mary')

Go
```

The following 2 transactions will result in a dead lock. Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code.

```
-- Transaction 1
Begin Tran
Update TableA Set Name = 'Mark Transaction 1' where Id = 1

-- From Transaction 2 window execute the first update statement

Update TableB Set Name = 'Mary Transaction 1' where Id = 1

-- From Transaction 2 window execute the second update statement
Commit Transaction



-- Transaction 2
Begin Tran
Update TableB Set Name = 'Mark Transaction 2' where Id = 1

-- From Transaction 1 window execute the second update statement

Update TableA Set Name = 'Mary Transaction 2' where Id = 1

-- After a few seconds notice that one of the transactions complete
-- successfully while the other transaction is made the deadlock victim

Commit Transaction
```

Next Video : We will discuss the criteria SQL Server uses to choose a

deadlock victim
SQL Server deadlock victim selection

Suggested Videos

Part 76 - Read committed snapshot isolation level in sql server
Part 77 - Difference between snapshot isolation and read committed snapshot
Part 78 - SQL Server deadlock example

In this video we will discuss

1. How SQL Server detects deadlocks
2. What happens when a deadlock is detected
3. What is DEADLOCK_PRIORITY
4. What is the criteria that SQL Server uses to choose a deadlock victim when there is a deadlock

This is continuation to Part 78, please watch Part 78 before proceeding.

How SQL Server detects deadlocks

Lock monitor thread in SQL Server, runs every 5 seconds by default to detect if there are any deadlocks. If the lock monitor thread finds deadlocks, the deadlock detection interval will drop from 5 seconds to as low as 100 milliseconds depending on the frequency of deadlocks. If the lock monitor thread stops finding deadlocks, the Database Engine increases the intervals between searches to 5 seconds.

What happens when a deadlock is detected

When a deadlock is detected, the Database Engine ends the deadlock by choosing one of the threads as the deadlock victim. The deadlock victim's transaction is then rolled back and returns a 1205 error to the application. Rolling back the transaction of the deadlock victim releases all locks held by that transaction. This allows the other transactions to become unblocked and move forward.

What is DEADLOCK_PRIORITY

By default, SQL Server chooses a transaction as the deadlock victim that is least expensive to roll back. However, a user can specify the priority of sessions in a deadlock situation using the SET DEADLOCK_PRIORITY statement. The session with the lowest deadlock priority is chosen as the deadlock victim.

Example : SET DEADLOCK_PRIORITY NORMAL

DEADLOCK_PRIORITY

1. The default is Normal
2. Can be set to LOW, NORMAL, or HIGH
3. Can also be set to a integer value in the range of -10 to 10.

LOW : -5
NORMAL : 0
HIGH : 5

What is the deadlock victim selection criteria

1. If the DEADLOCK_PRIORITY is different, the session with the lowest priority is selected as the victim
2. If both the sessions have the same priority, the transaction that is least expensive to rollback is selected as the victim

3. If both the sessions have the same deadlock priority and the same cost, a victim is chosen randomly

SQL Script to setup the tables for the examples

```
Create table TableA
(
   Id int identity primary key,
   Name nvarchar(50)
)
Go

Insert into TableA values ('Mark')
Insert into TableA values ('Ben')
Insert into TableA values ('Todd')
Insert into TableA values ('Pam')
Insert into TableA values ('Sara')
Go

Create table TableB
(
   Id int identity primary key,
   Name nvarchar(50)
)
Go

Insert into TableB values ('Mary')
Go
```

Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code. We have not explicitly set DEADLOCK_PRIORITY, so both the sessions have the default DEADLOCK_PRIORITY which is NORMAL. So in this case SQL Server is going to choose Transaction 2 as the deadlock victim as it is the least expensive one to rollback.

```
-- Transaction 1
Begin Tran
Update TableA Set Name = Name + ' Transaction 1' where Id IN (1, 2, 3, 4, 5)
```

```
-- From Transaction 2 window execute the first update statement

Update TableB Set Name = Name + ' Transaction 1' where Id = 1

-- From Transaction 2 window execute the second update statement
Commit Transaction


-- Transaction 2
Begin Tran
Update TableB Set Name = Name + ' Transaction 2' where Id = 1

-- From Transaction 1 window execute the second update statement

Update TableA Set Name = Name + ' Transaction 2' where Id IN (1, 2, 3, 4, 5)

-- After a few seconds notice that this transaction will be chosen as the deadlock
-- victim as it is less expensive to rollback this transaction than Transaction 1
Commit Transaction
```

In the following example we have set DEADLOCK_PRIORITY of Transaction 2 to HIGH. Transaction 1 will be chosen as the deadlock victim, because it's DEADLOCK_PRIORITY (Normal) is lower than the DEADLOCK_PRIORITY of Transaction 2.

```
-- Transaction 1
Begin Tran
Update TableA Set Name = Name + ' Transaction 1' where Id IN (1, 2, 3, 4, 5)

-- From Transaction 2 window execute the first update statement

Update TableB Set Name = Name + ' Transaction 1' where Id = 1

-- From Transaction 2 window execute the second update statement
Commit Transaction


-- Transaction 2
SET DEADLOCK_PRIORITY HIGH
GO
Begin Tran
Update TableB Set Name = Name + ' Transaction 2' where Id = 1

-- From Transaction 1 window execute the second update statement

Update TableA Set Name = Name + ' Transaction 2' where Id IN (1, 2, 3, 4, 5)

-- After a few seconds notice that Transaction 2 will be chosen as the
-- deadlock victim as it's DEADLOCK_PRIORITY (Normal) is lower than the
-- DEADLOCK_PRIORITY this transaction (HIGH)
Commit Transaction
```

Logging deadlocks in sql server

Suggested Videos

Part 77 - Difference between snapshot isolation and read committed snapshot
Part 78 - SQL Server deadlock example
Part 79 - SQL Server deadlock victim selection

In this video we will discuss **how to write the deadlock information to the SQL Server error log**

When deadlocks occur, SQL Server chooses one of the transactions as the deadlock victim and rolls it back. There are several ways in SQL Server to track down the queries that are causing deadlocks. One of the options is to use SQL Server trace flag 1222 to write the deadlock information to the SQL Server error log.

Enable Trace flag : To enable trace flags use DBCC command. -1 parameter indicates that the trace flag must be set at the global level. If you omit -1 parameter the trace flag will be set only at the session level.

```
DBCC Traceon(1222, -1)
```

To check the status of the trace flag

```
DBCC TraceStatus(1222, -1)
```

To turn off the trace flag

```
DBCC Traceoff(1222, -1)
```

The following SQL code generates a dead lock. This is the same code we discussed in Part 78 of SQL Server Tutorial.

```
--SQL script to create the tables and populate them with test data
Create table TableA
(
   Id int identity primary key,
   Name nvarchar(50)
)
Go

Insert into TableA values ('Mark')
Go

Create table TableB
(
   Id int identity primary key,
   Name nvarchar(50)
)
Go

Insert into TableB values ('Mary')
Go
```

```
--SQL Script to create stored procedures
Create procedure spTransaction1
as
Begin
   Begin Tran
   Update TableA Set Name = 'Mark Transaction 1' where Id = 1
   Waitfor delay '00:00:05'
   Update TableB Set Name = 'Mary Transaction 1' where Id = 1
   Commit Transaction
End

Create procedure spTransaction2
as
Begin
   Begin Tran
   Update TableB Set Name = 'Mark Transaction 2' where Id = 1
   Waitfor delay '00:00:05'
   Update TableA Set Name = 'Mary Transaction 2' where Id = 1
   Commit Transaction
End
```

Open 2 instances of SQL Server Management studio. From the first window execute **spTransaction1** and from the second window execute **spTransaction2**.

After a few seconds notice that one of the transactions complete successfully while the other transaction is made the deadlock victim and rollback.

SQL Server deadlock analysis and prevention

Suggested Videos

Part 78 - SQL Server deadlock example
Part 79 - SQL Server deadlock victim selection
Part 80 - Logging deadlocks in sql server

In this video we will discuss **how to read and analyze sql server deadlock information captured in the error log**, so we can understand what's causing the deadlocks and take appropriate actions to prevent or minimize the occurrence of deadlocks. This is continuation to Part 80. Please watch Part 80 from SQL Server tutorial before proceeding.

The deadlock information in the error log has three sections

Section	Description
Deadlock Victim	Contains the ID of the process that was selected as the deadlock victim and killed by SQL Server.
Process List	Contains the list of the processes that participated in the deadlock.
Resource List	Contains the list of the resources (database objects) owned by the processes involved in the deadlock

Process List : The process list has lot of items. Here are some of them that are particularly useful in understanding what caused the deadlock.

Node	Description
loginname	The loginname associated with the process
isolationlevel	What isolation level is used
procname	The stored procedure name
Inputbuf	The code the process is executing when the deadlock occured

Resource List : Some of the items in the resource list that are particularly useful in understanding what caused the deadlock.

Node	Description
objectname	Fully qualified name of the resource involved in the deadlock
owner-list	Contains (owner id) the id of the owning process and the lock mode it has acquired on the resource. lock mode determines how the resource can be accessed by concurrent transactions. S for Shared lock, U for Update lock, X for Exclusive lock etc
waiter-list	Contains (waiter id) the id of the process that wants to acquire a lock on the resource and the lock mode it is requesting

To prevent the deadlock that we have in our case, we need to ensure that database objects (Table A & Table B) are accessed in the same order every time. SQL Server deadlock analysis and prevention

Suggested Videos

Part 78 - SQL Server deadlock example
Part 79 - SQL Server deadlock victim selection
Part 80 - Logging deadlocks in sql server

In this video we will discuss **how to read and analyze sql server deadlock information captured in the error log**, so we can understand what's causing the deadlocks and take appropriate actions to prevent or minimize the occurrence of deadlocks. This is continuation to Part 80. Please watch Part 80 from SQL Server tutorial before proceeding.

The deadlock information in the error log has three sections

Section	Description
Deadlock Victim	Contains the ID of the process that was selected as the deadlock victim and killed by SQL Server.
Process List	Contains the list of the processes that participated in the deadlock.
Resource List	Contains the list of the resources (database objects) owned by the processes involved in the deadlock

Process List : The process list has lot of items. Here are some of them that are particularly useful in understanding what caused the deadlock.

Node	Description
loginname	The loginname associated with the process
isolationlevel	What isolation level is used
procname	The stored procedure name
Inputbuf	The code the process is executing when the deadlock occured

Resource List : Some of the items in the resource list that are particularly useful in understanding what caused the deadlock.

Node	Description
objectname	Fully qualified name of the resource involved in the deadlock
owner-list	Contains (owner id) the id of the owning process and the lock mode it has acquired on the resource. lock mode determines how the resource can be accessed by concurrent transactions. S for Shared lock, U for Update lock, X for Exclusive lock etc
waiter-list	Contains (waiter id) the id of the process that wants to acquire a lock on the resource and the lock mode it is requesting

To prevent the deadlock that we have in our case, we need to ensure that database objects (Table A & Table B) are accessed in the same order every time.

Capturing deadlocks in sql profiler

Suggested Videos

Part 79 - SQL Server deadlock victim selection
Part 80 - Logging deadlocks in sql server
Part 81 - SQL Server deadlock analysis and prevention

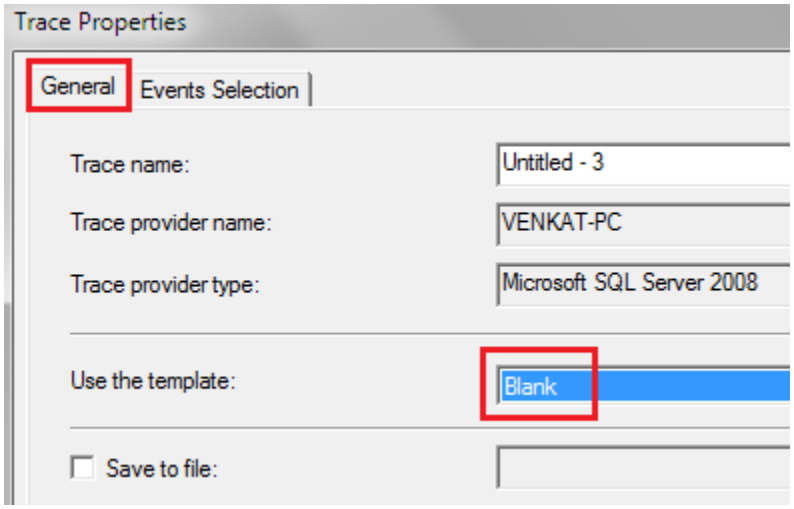
In this video we will discuss **how to capture deadlock graph using SQL profiler.** To capture deadlock graph, all you need to do is add Deadlock graph event to the trace in SQL profiler.

Here are the steps :

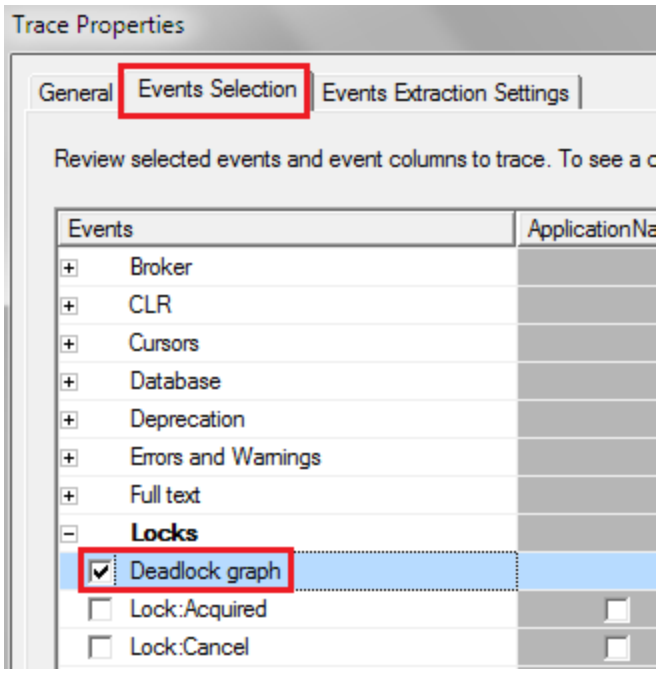
1. Open SQL Profiler

2. Click **File - New Trace**. Provide the credentials and connect to the server

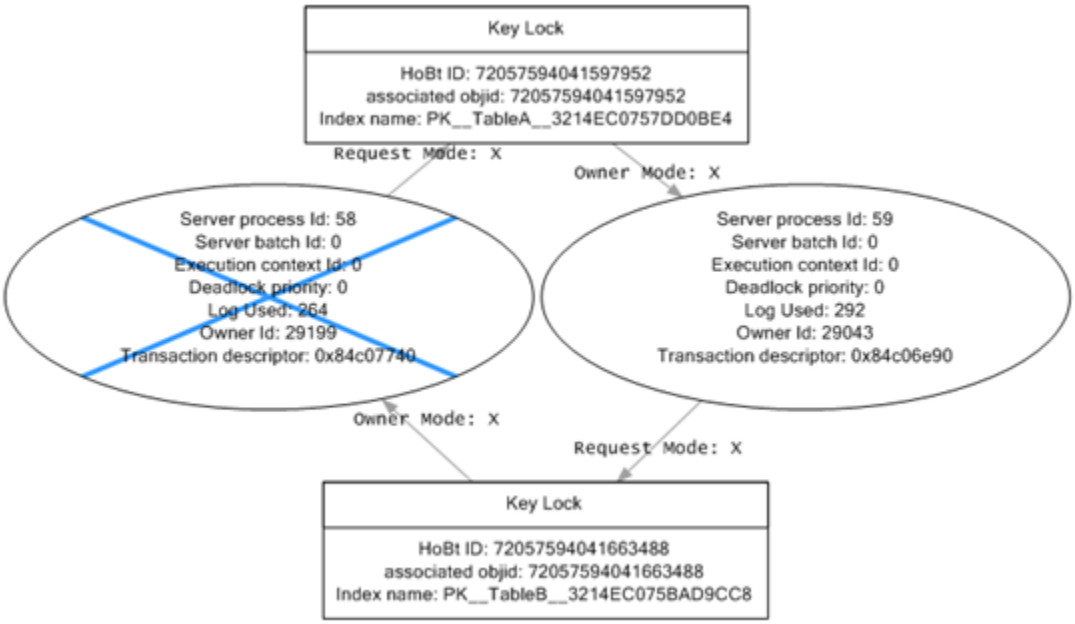
3. On the general tab, select **"Blank"** template from **"Use the template"** dropdownlist



4. On the **"Events Selection"** tab, expand **"Locks"** section and select **"Deadlock graph"** event



5. Finally click the **Run** button to start the trace
6. At this point execute the code that causes deadlock
7. The deadlock graph should be captured in the profiler as shown below.



The deadlock graph data is captured in XML format. If you want to extract this XML data to a physical file for later analysis, you can do so by following the steps below.
1. In SQL profiler, click on **"File - Export - Extract SQL Server Events - Extract Deadlock Events"**
2. Provide a name for the file
3. The extension for the deadlock xml file is **.xdl**

4. Finally choose if you want to export all events in a single file or each event in a separate file

The deadlock information in the XML file is similar to what we have captured using the trace flag 1222.

Analyzing the deadlock graph

1. The oval on the graph, with the blue cross, represents the transaction that was chosen as the deadlock victim by SQL Server.

2. The oval on the graph represents the transaction that completed successfully.

3. When you move the mouse pointer over the oval, you can see the SQL code that was running that caused the deadlock.

4. The oval symbols represent the process nodes

- **Server Process Id :** If you are using SQL Server Management Studio you can see the server process id on information bar at the bottom.
- **Deadlock Priority :** If you have not set DEADLOCK PRIORITY explicitly using SET DEADLOCK PRIORITY statement, then both the processes should have the same default deadlock priority NORMAL (0).
- **Log Used :** The transaction log space used. If a transaction has used a lot of log space then the cost to roll it back is also more. So the transaction that has used the least log space is killed and rolled back.

5. The rectangles represent the resource nodes.

- **HoBt ID :** Heap Or Binary Tree ID. Using this ID query **sys.partitions** view to find the database objects involved in the deadlock.

```
SELECT object_name([object_id])
FROM sys.partitions
WHERE hobt_id = 72057594041663488
```

6. The arrows represent types of locks each process has on each resource node.

SQL Server deadlock error handling

Suggested Videos

Part 80 - Logging deadlocks in sql server
Part 81 - SQL Server deadlock analysis and prevention
Part 82 - Capturing deadlocks in SQL Profiler

In this video we will discuss **how to catch deadlock error using try/catch in SQL Server**. Modify the stored procedure as shown below to catch the deadlock error. The code is commented and is self-explanatory.

```
Alter procedure spTransaction1
as
Begin
   Begin Tran
   Begin Try
```

```
        Update TableA Set Name = 'Mark Transaction 1' where Id = 1
        Waitfor delay '00:00:05'
        Update TableB Set Name = 'Mary Transaction 1' where Id = 1
        -- If both the update statements succeeded.
        -- No Deadlock occurred. So commit the transaction.
        Commit Transaction
        Select 'Transaction Successful'
    End Try
    Begin Catch
        -- Check if the error is deadlock error
        If(ERROR_NUMBER() = 1205)
        Begin
            Select 'Deadlock. Transaction failed. Please retry'
        End
        -- Rollback the transaction
        Rollback
    End Catch
End

Alter procedure spTransaction2
as
Begin
    Begin Tran
    Begin Try
        Update TableB Set Name = 'Mary Transaction 2' where Id = 1
        Waitfor delay '00:00:05'
        Update TableA Set Name = 'Mark Transaction 2' where Id = 1
        Commit Transaction
        Select 'Transaction Successful'
    End Try
    Begin Catch
        If(ERROR_NUMBER() = 1205)
        Begin
            Select 'Deadlock. Transaction failed. Please retry'
        End
        Rollback
    End Catch
End
```

After modifying the stored procedures, execute both the procedures from 2 different windows simultaneously. Notice that the deadlock error is handled by the catch block.

In our next video, we will discuss **how applications using ADO.NET can handle deadlock errors**.

Modify the stored procedure as shown below to catch the deadlock error. The code is commented and is self-explanatory.

```
Alter procedure spTransaction1
as
```

```
Begin
   Begin Tran
   Begin Try
      Update TableA Set Name = 'Mark Transaction 1' where Id = 1
      Waitfor delay '00:00:05'
      Update TableB Set Name = 'Mary Transaction 1' where Id = 1
      -- If both the update statements succeeded.
      -- No Deadlock occurred. So commit the transaction.
      Commit Transaction
      Select 'Transaction Successful'
   End Try
   Begin Catch
      -- Check if the error is deadlock error
      If(ERROR_NUMBER() = 1205)
      Begin
         Select 'Deadlock. Transaction failed. Please retry'
      End
      -- Rollback the transaction
      Rollback
   End Catch
End

Alter procedure spTransaction2
as
Begin
   Begin Tran
   Begin Try
      Update TableB Set Name = 'Mary Transaction 2' where Id = 1
      Waitfor delay '00:00:05'
      Update TableA Set Name = 'Mark Transaction 2' where Id = 1
      Commit Transaction
      Select 'Transaction Successful'
   End Try
   Begin Catch
      If(ERROR_NUMBER() = 1205)
      Begin
         Select 'Deadlock. Transaction failed. Please retry'
      End
      Rollback
   End Catch
End
```

After modifying the stored procedures, execute both the procedures from 2 different windows simultaneously. Notice that the deadlock error is handled by the catch block.

In our next video, we will discuss **how applications using ADO.NET can handle deadlock errors**.

To handle deadlock errors in ADO.NET

1. Catch the SqlException object
2. Check if the error is deadlock error using the Number property of the SqlException object

Stored Procedure 1 Code

```
Alter procedure spTransaction1
as
Begin
   Begin Tran
   Update TableA Set Name = 'Mark Transaction 1' where Id = 1
   Waitfor delay '00:00:05'
   Update TableB Set Name = 'Mary Transaction 1' where Id = 1
   Commit Transaction
End
```

Stored Procedure 2 Code

```
Alter procedure spTransaction2
as
Begin
   Begin Tran
   Update TableB Set Name = 'Mark Transaction 2' where Id = 1
   Waitfor delay '00:00:05'
   Update TableA Set Name = 'Mary Transaction 2' where Id = 1
   Commit Transaction
End
```

WebForm1.aspx HTML

```
<table>
   <tr>
      <td>
         <asp:Button ID="Button1" runat="server"
            Text="Update Table A and then Table B"
            OnClick="Button1_Click" />
      </td>
   </tr>
   <tr>
      <td>
         <asp:Label ID="Label1" runat="server"></asp:Label>
      </td>
   </tr>
</table>
```

WebForm1.aspx.cs code

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
   public partial class WebForm1 : System.Web.UI.Page
   {
```

```
        protected void Page_Load(object sender, EventArgs e)
        { }

        protected void Button1_Click(object sender, EventArgs e)
        {
            try
            {
                string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
                using (SqlConnection con = new SqlConnection(cs))
                {
                    SqlCommand cmd = new SqlCommand("spTransaction1", con);
                    cmd.CommandType = CommandType.StoredProcedure;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    Label1.Text = "Transaction successful";
                    Label1.ForeColor = System.Drawing.Color.Green;
                }
            }
            catch (SqlException ex)
            {
                if (ex.Number == 1205)
                {
                    Label1.Text = "Deadlock. Please retry";
                }
                else
                {
                    Label1.Text = ex.Message;
                }
                Label1.ForeColor = System.Drawing.Color.Red;
            }
        }
    }
}
```

WebForm2.aspx HTML

```
<table>
    <tr>
        <td>
            <asp:Button ID="Button1" runat="server"
                Text="Update Table B and then Table A"
                OnClick="Button1_Click" />
        </td>
    </tr>
    <tr>
        <td>
            <asp:Label ID="Label1" runat="server"></asp:Label>
        </td>
    </tr>
</table>
```

WebForm2.aspx.cs code

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }

        protected void Button1_Click(object sender, EventArgs e)
        {
            try
            {
                string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
                using (SqlConnection con = new SqlConnection(cs))
                {
                    SqlCommand cmd = new SqlCommand("spTransaction1", con);
                    cmd.CommandType = CommandType.StoredProcedure;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    Label1.Text = "Transaction successful";
                    Label1.ForeColor = System.Drawing.Color.Green;
                }
            }
            catch (SqlException ex)
            {
                if (ex.Number == 1205)
                {
                    Label1.Text = "Deadlock. Please retry";
                }
                else
                {
                    Label1.Text = ex.Message;
                }
                Label1.ForeColor = System.Drawing.Color.Red;
            }
        }
    }
}
```

Handling deadlocks in ado.net

Suggested Videos

Part 81 - SQL Server deadlock analysis and prevention
Part 82 - Capturing deadlocks in SQL profiler
Part 83 - SQL Server deadlock error handling

In this video we will discuss **how to handle deadlock errors in an ADO.NET application**. **To handle deadlock errors in ADO.NET**

1. Catch the SqlException object
2. Check if the error is deadlock error using the Number property of the SqlException object

Stored Procedure 1 Code

```
Alter procedure spTransaction1
as
Begin
   Begin Tran
   Update TableA Set Name = 'Mark Transaction 1' where Id = 1
   Waitfor delay '00:00:05'
   Update TableB Set Name = 'Mary Transaction 1' where Id = 1
   Commit Transaction
End
```

Stored Procedure 2 Code

```
Alter procedure spTransaction2
as
Begin
   Begin Tran
   Update TableB Set Name = 'Mark Transaction 2' where Id = 1
   Waitfor delay '00:00:05'
   Update TableA Set Name = 'Mary Transaction 2' where Id = 1
   Commit Transaction
End
```

WebForm1.aspx HTML

```
<table>
   <tr>
      <td>
         <asp:Button ID="Button1" runat="server"
            Text="Update Table A and then Table B"
            OnClick="Button1_Click" />
      </td>
   </tr>
   <tr>
      <td>
         <asp:Label ID="Label1" runat="server"></asp:Label>
      </td>
   </tr>
</table>
```

WebForm1.aspx.cs code

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
```

```
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }

        protected void Button1_Click(object sender, EventArgs e)
        {
            try
            {
                string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
                using (SqlConnection con = new SqlConnection(cs))
                {
                    SqlCommand cmd = new SqlCommand("spTransaction1", con);
                    cmd.CommandType = CommandType.StoredProcedure;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    Label1.Text = "Transaction successful";
                    Label1.ForeColor = System.Drawing.Color.Green;
                }
            }
            catch (SqlException ex)
            {
                if (ex.Number == 1205)
                {
                    Label1.Text = "Deadlock. Please retry";
                }
                else
                {
                    Label1.Text = ex.Message;
                }
                Label1.ForeColor = System.Drawing.Color.Red;
            }
        }
    }
}
```

WebForm2.aspx HTML

```
<table>
    <tr>
        <td>
            <asp:Button ID="Button1" runat="server"
                Text="Update Table B and then Table A"
                OnClick="Button1_Click" />
        </td>
    </tr>
    <tr>
        <td>
            <asp:Label ID="Label1" runat="server"></asp:Label>
        </td>
    </tr>
</table>
```

WebForm2.aspx.cs code

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }

        protected void Button1_Click(object sender, EventArgs e)
        {
            try
            {
                string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
                using (SqlConnection con = new SqlConnection(cs))
                {
                    SqlCommand cmd = new SqlCommand("spTransaction1", con);
                    cmd.CommandType = CommandType.StoredProcedure;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    Label1.Text = "Transaction successful";
                    Label1.ForeColor = System.Drawing.Color.Green;
                }
            }
            catch (SqlException ex)
            {
                if (ex.Number == 1205)
                {
                    Label1.Text = "Deadlock. Please retry";
                }
                else
                {
                    Label1.Text = ex.Message;
                }
                Label1.ForeColor = System.Drawing.Color.Red;
            }
        }
    }
}
```

Retry logic for deadlock exceptions

Suggested Videos

Part 82 - Capturing deadlocks in SQL profiler
Part 83 - SQL Server deadlock error handling
Part 84 - Handling deadlocks in ado.net

In this video we will discuss implementing **retry logic for deadlock exceptions**.

This is continuation to Part 84. Please watch Part 84, before proceeding.

When a transaction fails due to deadlock, we can write some logic so the system can resubmit the transaction. The deadlocks usually last for a very short duration. So upon resubmitting the transaction it may complete successfully. This is much better from user experience standpoint.

To achieve this we will be using the following technologies
C#
ASP.NET
SQL Server
jQuery AJAX

Result.cs

```
public class Result
{
    public int AttemptsLeft { get; set; }
    public string Message { get; set; }
    public bool Success { get; set; }
}
```

WebForm1.aspx HTML and jQuery code

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <script src="jquery-1.11.2.js"></script>
    <script type="text/javascript">
        $(document).ready(function () {
            var lblMessage = $('#Label1');
            var attemptsLeft;

            function updateData() {
                $.ajax({
                    url: 'WebForm1.aspx/CallStoredProcedure',
                    method: 'post',
                    contentType: 'application/json',
                    data: '{attemptsLeft:' + attemptsLeft + '}',
                    dataType: 'json',
                    success: function (data) {
                        lblMessage.text(data.d.Message);
                        attemptsLeft = data.d.AttemptsLeft;
                        if (data.d.Success) {
                            $('#btn').prop('disabled', false);
                            lblMessage.css('color','green');
                        }
                        else if(attemptsLeft > 0){
                            lblMessage.css('color', 'red');
                            updateData();
                        }
                        else {
                            lblMessage.css('color', 'red');
```

```
                    lblMessage.text('Deadlock Occurred. ZERO attempts left. Please try later');
                }
            },
            error: function (err) {
                lblMessage.css('color', 'red');
                lblMessage.text(err.responseText);
            }
        });
    }

    $('#btn').click(function () {
        $(this).prop('disabled', true);
        lblMessage.text('Updating....');
        attemptsLeft = 5;
        updateData();
    });
});
</script>
</head>
<body style="font-family: Arial">
<form id="form1" runat="server">
    <input id="btn" type="button"
        value="Update Table A and then Table B" />
    <br />
    <asp:Label ID="Label1" runat="server"></asp:Label>
</form>
</body>
</html>
```

WebForm1.aspx.cs code

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }

        [System.Web.Services.WebMethod]
        public static Result CallStoredProcedure(int attemptsLeft)
        {
            Result _result = new Result();
            if (attemptsLeft > 0)
            {
                try
                {
                    string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
```

```
                using (SqlConnection con = new SqlConnection(cs))
                {
                    SqlCommand cmd = new SqlCommand("spTransaction15", con);
                    cmd.CommandType = CommandType.StoredProcedure;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    _result.Message = "Transaction successful";
                    _result.AttemptsLeft = 0;
                    _result.Success = true;
                }
            }
            catch (SqlException ex)
            {
                if (ex.Number == 1205)
                {
                    _result.AttemptsLeft = attemptsLeft - 1;
                    _result.Message = "Deadlock occurred. Retrying. Attempts left : "
                        + _result.AttemptsLeft.ToString();
                }
                else
                {
                    throw;
                }
                _result.Success = false;
            }
        }
        return _result;
    }
  }
}
```

Copy and paste the above code in WebForm2.aspx and make the required changes as described in the video.

How to find blocking queries in sql server

Suggested Videos

Part 83 - SQL Server deadlock error handling
Part 84 - Handling deadlocks in ado.net
Part 85 - Retry logic for deadlock exceptions

In this video we will discuss, **how to find blocking queries in sql server**.

Blocking occurs if there are open transactions. Let us understand this with an example.

Execute the following 2 sql statements

```
Begin Tran
Update TableA set Name='Mark Transaction 1' where Id = 1
```

Now from a different window, execute any of the following commands. Notice that all the queries are blocked.

```
Select Count(*) from TableA
```

```
Delete from TableA where Id = 1
Truncate table TableA
Drop table TableA
```

This is because there is an open transaction. Once the open transaction completes, you will be able to execute the above queries.

So the obvious next question is - **How to identify all the active transactions**.

One way to do this is by using DBCC OpenTran. DBCC OpenTran will display only the oldest active transaction. It is not going to show you all the open transactions.

```
DBCC OpenTran
```

The following link has the SQL script that you can use to identify all the active transactions.
http://www.sqlskills.com/blogs/paul/script-open-transactions-with-text-and-plans

The beauty about this script is that it has a lot more useful information about the open transactions
Session Id
Login Name
Database Name
Transaction Begin Time
The actual query that is executed

You can now use this information and ask the respective developer to either commit or rollback the transactions that they have left open unintentionally.

For some reason if the person who initiated the transaction is not available, you also have the option to KILL the associated process. However, this may have unintended consequences, so use it with extreme caution.

There are 2 ways to kill the process are described below

Killing the process using SQL Server Activity Monitor :
1. Right Click on the Server Name in Object explorer and select **"Activity Monitor"**
2. In the **"Activity Monitor"** window expand Processes section
3. Right click on the associated **"Session ID"** and select **"Kill Process"** from the context menu

Killing the process using SQL command :

```
KILL Process_ID
```

What happens when you kill a session
All the work that the transaction has done will be rolled back. The database must be put back in the state it was in, before the transaction started.

SQL Server except operator

Suggested Videos

Part 84 - Handling deadlocks in ado.net
Part 85 - Retry logic for deadlock exceptions
Part 86 - How to find blocking queries in sql server

In this video we will discuss **SQL Server except operator with examples.**

EXCEPT operator returns unique rows from the left query that aren't in the right query's results.

- Introduced in SQL Server 2005
- The number and the order of the columns must be the same in all queries
- The data types must be same or compatible
- This is similar to minus operator in oracle

Let us understand this with an example. We will use the following 2 tables for this example.

Table A

Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male
4	John	Male
5	Sara	Female

Table B

Id	Name	Gender
4	John	Male
5	Sara	Female
6	Pam	Female
7	Rebeka	Female
8	Jordan	Male

SQL Script to create the tables

```
Create Table TableA
(
   Id int primary key,
   Name nvarchar(50),
   Gender nvarchar(10)
)
Go

Insert into TableA values (1, 'Mark', 'Male')
Insert into TableA values (2, 'Mary', 'Female')
Insert into TableA values (3, 'Steve', 'Male')
Insert into TableA values (4, 'John', 'Male')
Insert into TableA values (5, 'Sara', 'Female')
Go

Create Table TableB
(
   Id int primary key,
   Name nvarchar(50),
   Gender nvarchar(10)
)
```

```
Go

Insert into TableB values (4, 'John', 'Male')
Insert into TableB values (5, 'Sara', 'Female')
Insert into TableB values (6, 'Pam', 'Female')
Insert into TableB values (7, 'Rebeka', 'Female')
Insert into TableB values (8, 'Jordan', 'Male')
Go
```

Notice that the following query returns the unique rows from the left query that aren't in the right query's results.

```
Select Id, Name, Gender
From TableA
Except
Select Id, Name, Gender
From TableB
```

Result :

Result		
Id	**Name**	**Gender**
1	Mark	Male
2	Mary	Female
3	Steve	Male

To retrieve all of the rows from Table B that does not exist in Table A, reverse the two queries as shown below.

```
Select Id, Name, Gender
From TableB
Except
Select Id, Name, Gender
From TableA
```

Result :

Result		
Id	**Name**	**Gender**
6	Pam	Female
7	Rebeka	Female
8	Jordan	Male

You can also use Except operator on a single table. Let's use the following tblEmployees table for this example.

tblEmployees			
Id	Name	Gender	Salary
1	Mark	Male	52000
2	Mary	Female	55000
3	Steve	Male	45000
4	John	Male	40000
5	Sara	Female	48000
6	Pam	Female	60000
7	Tom	Male	58000
8	George	Male	65000
9	Tina	Female	67000
10	Ben	Male	80000

SQL script to create tblEmployees table

```
Create table tblEmployees
(
    Id int identity primary key,
    Name nvarchar(100),
    Gender nvarchar(10),
    Salary int
)
Go

Insert into tblEmployees values ('Mark', 'Male', 52000)
Insert into tblEmployees values ('Mary', 'Female', 55000)
Insert into tblEmployees values ('Steve', 'Male', 45000)
Insert into tblEmployees values ('John', 'Male', 40000)
Insert into tblEmployees values ('Sara', 'Female', 48000)
Insert into tblEmployees values ('Pam', 'Female', 60000)
Insert into tblEmployees values ('Tom', 'Male', 58000)
Insert into tblEmployees values ('George', 'Male', 65000)
Insert into tblEmployees values ('Tina', 'Female', 67000)
Insert into tblEmployees values ('Ben', 'Male', 80000)
Go
```

Result :

Result			
Id	Name	Gender	Salary
1	Mark	Male	52000
2	Mary	Female	55000
7	Tom	Male	58000

Order By clause should be used only once after the right query

```
Select Id, Name, Gender, Salary
From tblEmployees
Where Salary >= 50000
Except
Select Id, Name, Gender, Salary
From tblEmployees
Where Salary >= 60000
order By Name
```

Difference between except and not in sql server

Suggested Videos

Part 85 - Retry logic for deadlock exceptions
Part 86 - How to find blocking queries in sql server
Part 87 - SQL Server except operator

In this video we will discuss the **difference between EXCEPT and NOT IN operators in SQL Server**.

We will use the following 2 tables for this example.

Table A

Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male

Table B

Id	Name	Gender
2	Mary	Female
3	Steve	Male

The following query returns the rows from the left query that aren't in the right query's results.

```
Select Id, Name, Gender From TableA
Except
Select Id, Name, Gender From TableB
```

Result :

Result

Id	Name	Gender
1	Mark	Male

The same result can also be achieved using NOT IN operator.

```
Select Id, Name, Gender From TableA
Where Id NOT IN (Select Id from TableB)
```

So, what is the difference between EXCEPT and NOT IN operators

1. Except filters duplicates and returns only DISTINCT rows from the left query that aren't in the right query's results, where as NOT IN does not filter the duplicates.

Insert the following row into TableA
```
Insert into TableA values (1, 'Mark', 'Male')
```

Now execute the following EXCEPT query. Notice that we get only the DISTINCT rows
```
Select Id, Name, Gender From TableA
Except
Select Id, Name, Gender From TableB
```

Result:

Result		
Id	**Name**	**Gender**
1	Mark	Male

Now execute the following query. Notice that the duplicate rows are not filtered.
```
Select Id, Name, Gender From TableA
Where Id NOT IN (Select Id from TableB)
```

Result:

Result		
Id	**Name**	**Gender**
1	Mark	Male
1	Mark	Male

2. EXCEPT operator expects the same number of columns in both the queries, where as NOT IN, compares a single column from the outer query with a single column from the subquery.

In the following example, the number of columns are different.
```
Select Id, Name, Gender From TableA
Except
Select Id, Name From TableB
```

The above query would produce the following error.
```
Msg 205, Level 16, State 1, Line 1
All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists.
```

NOT IN, compares a single column from the outer query with a single column from subquery.

In the following example, the subquery returns multiple columns
```
Select Id, Name, Gender From TableA
Where Id NOT IN (Select Id, Name from TableB)
```

```
Msg 116, Level 16, State 1, Line 2
Only one expression can be specified in the select list when the subquery is not introduced with EXISTS.
```

Intersect operator in sql server

Suggested Videos

Part 86 - How to find blocking queries in sql server
Part 87 - SQL Server except operator
Part 88 - Difference between except and not in sql server

In this video we will discuss

1. Intersect operator in sql server
2. Difference between intersect and inner join

Intersect operator retrieves the common records from both the left and the right query of the Intersect operator.

- Introduced in SQL Server 2005
- The number and the order of the columns must be same in both the queries
- The data types must be same or at least compatible

Let us understand INTERSECT operator with an example.

We will use the following 2 tables for this example.

Table A

Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male

Table B

Id	Name	Gender
2	Mary	Female
3	Steve	Male

SQL Script to create the tables and populate with test data

```
Create Table TableA
(
    Id int,
    Name nvarchar(50),
    Gender nvarchar(10)
)
Go

Insert into TableA values (1, 'Mark', 'Male')
Insert into TableA values (2, 'Mary', 'Female')
Insert into TableA values (3, 'Steve', 'Male')
Go

Create Table TableB
(
    Id int,
    Name nvarchar(50),
    Gender nvarchar(10)
)
Go
```

```
Insert into TableB values (2, 'Mary', 'Female')
Insert into TableB values (3, 'Steve', 'Male')
Go
```

The following query retrieves the common records from both the left and the right query of the Intersect operator.

```
Select Id, Name, Gender from TableA
Intersect
Select Id, Name, Gender from TableB
```

Result :

Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male

We can also achieve the same thinkg using INNER join. The following INNER join query would produce the exact same result.

```
Select TableA.Id, TableA.Name, TableA.Gender
From TableA Inner Join TableB
On TableA.Id = TableB.Id
```

What is the difference between INTERSECT and INNER JOIN

1. INTERSECT filters duplicates and returns only DISTINCT rows that are common between the LEFT and Right Query, where as INNER JOIN does not filter the duplicates.

To understand this difference, insert the following row into TableA

```
Insert into TableA values (2, 'Mary', 'Female')
```

Now execute the following INTERSECT query. Notice that we get only the DISTINCT rows

```
Select Id, Name, Gender from TableA
Intersect
Select Id, Name, Gender from TableB
```

Result :

Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male

Now execute the following INNER JOIN query. Notice that the duplicate rows are not filtered.

```
Select TableA.Id, TableA.Name, TableA.Gender
From TableA Inner Join TableB
On TableA.Id = TableB.Id
```

Result :

Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male
2	Mary	Female

You can make the INNER JOIN behave like INTERSECT operator by using the DISTINCT operator

```
Select DISTINCT TableA.Id, TableA.Name, TableA.Gender
From TableA Inner Join TableB
On TableA.Id = TableB.Id
```

Result :

Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male

2. INNER JOIN treats two NULLS as two different values. So if you are joining two tables based on a nullable column and if both tables have NULLs in that joining column then, INNER JOIN will not include those rows in the result-set, where as INTERSECT treats two NULLs as a same value and it returns all matching rows.

To understand this difference, execute the following 2 insert statements

```
Insert into TableA values(NULL, 'Pam', 'Female')
Insert into TableB values(NULL, 'Pam', 'Female')
```

INTERSECT query

```
Select Id, Name, Gender from TableA
Intersect
Select Id, Name, Gender from TableB
```

Result :

Result		
Id	Name	Gender
NULL	Pam	Female
2	Mary	Female
3	Steve	Male

INNER JOIN query

```
Select TableA.Id, TableA.Name, TableA.Gender
From TableA Inner Join TableB
On TableA.Id = TableB.Id
```

Result :

Result		
Id	Name	Gender
2	Mary	Female
3	Steve	Male
2	Mary	Female

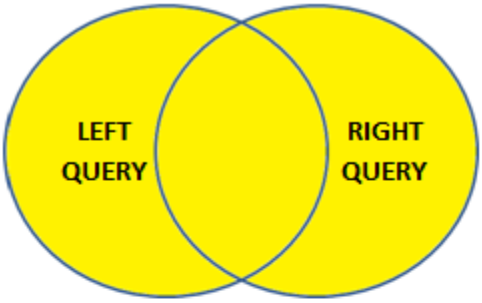
Difference between union intersect and except in sql server

Suggested Videos

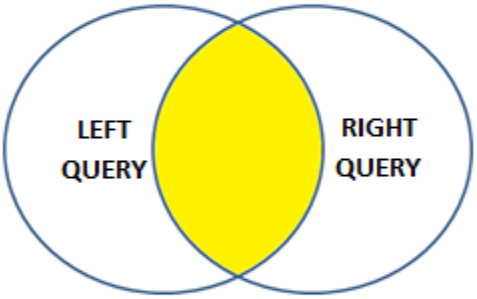
Part 87 - SQL Server except operator
Part 88 - Difference between except and not in sql server
Part 89 - Intersect operator in sql server

In this video we will discuss the **difference between union intersect and except in sql server with examples**.

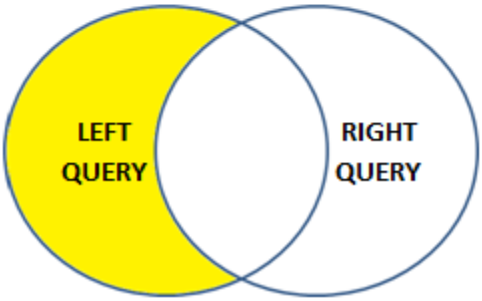
The following diagram explains the difference graphically



UNION operator returns all the unique rows from both the left and the right query. UNION ALL includes the duplicates as well



INTERSECT operator retrieves the common unique rows from both the left and the right query



EXCEPT operator returns unique rows from the left query that aren't in the right query's results

UNION operator returns all the unique rows from both the left and the right query. UNION ALL included the duplicates as well.

EXCEPT operator returns unique rows from the left query that aren't in the right query's results.

Let us understand these differences with examples. We will use the following 2 tables for the examples.

Table A

Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male
3	Steve	Male

Table B

Id	Name	Gender
2	Mary	Female
3	Steve	Male
4	John	Male

SQL Script to create the tables

```
Create Table TableA
(
    Id int,
    Name nvarchar(50),
    Gender nvarchar(10)
)
Go

Insert into TableA values (1, 'Mark', 'Male')
Insert into TableA values (2, 'Mary', 'Female')
Insert into TableA values (3, 'Steve', 'Male')
Insert into TableA values (3, 'Steve', 'Male')
Go

Create Table TableB
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10)
)
Go

Insert into TableB values (2, 'Mary', 'Female')
Insert into TableB values (3, 'Steve', 'Male')
Insert into TableB values (4, 'John', 'Male')
Go
```

UNION operator returns all the unique rows from both the queries. Notice the duplicates are removed.

```
Select Id, Name, Gender from TableA
UNION
Select Id, Name, Gender from TableB
```

Result :

UNION Result		
Id	**Name**	**Gender**
1	Mark	Male
2	Mary	Female
3	Steve	Male
4	John	Male

UNION ALL operator returns all the rows from both the queries, including the duplicates.

```
Select Id, Name, Gender from TableA
UNION ALL
Select Id, Name, Gender from TableB
```

Result :

UNION ALL Result		
Id	**Name**	**Gender**
1	Mark	Male
2	Mary	Female
3	Steve	Male
3	Steve	Male
2	Mary	Female
3	Steve	Male
4	John	Male

INTERSECT operator retrieves the common unique rows from both the left and the right query. Notice the duplicates are removed.

```
Select Id, Name, Gender from TableA
INTERSECT
Select Id, Name, Gender from TableB
```

Result :

INTERSECT Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male

EXCEPT operator returns unique rows from the left query that aren't in the right query's results.

```
Select Id, Name, Gender from TableA
EXCEPT
Select Id, Name, Gender from TableB
```

Result :

EXCEPT Result		
Id	**Name**	**Gender**
1	Mark	Male

If you wnat the rows that are present in Table B but not in Table A, reverse the queries.

```
Select Id, Name, Gender from TableB
EXCEPT
Select Id, Name, Gender from TableA
```

Result :

EXCEPT Result		
Id	Name	Gender
4	John	Male

For all these 3 operators to work the following 2 conditions must be met

- The number and the order of the columns must be same in both the queries
- The data types must be same or at least compatible

For example, if the number of columns are different, you will get the following error

Msg 205, Level 16, State 1, Line 1
All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists.

Cross apply and outer apply in sql server

Suggested Videos
Part 88 - Difference between except and not in sql server
Part 89 - Intersect operator in sql server
Part 90 - Difference between union intersect and except in sql server
In this video we will discuss **cross apply and outer apply in sql server** with examples.
We will use the following 2 tables for examples in this demo

Department Table	
Id	DepartmentName
1	IT
2	HR
3	Payroll
4	Administration
5	Sales

Employee Table				
Id	Name	Gender	Salary	DepartmentId
1	Mark	Male	50000	1
2	Mary	Female	60000	3
3	Steve	Male	45000	2
4	John	Male	56000	1
5	Sara	Female	39000	2

SQL Script to create the tables and populate with test data

```
Create table Department
(
   Id int primary key,
   DepartmentName nvarchar(50)
)
Go

Insert into Department values (1, 'IT')
Insert into Department values (2, 'HR')
Insert into Department values (3, 'Payroll')
Insert into Department values (4, 'Administration')
Insert into Department values (5, 'Sales')
```

```
Go

Create table Employee
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int,
    DepartmentId int foreign key references Department(Id)
)
Go

Insert into Employee values (1, 'Mark', 'Male', 50000, 1)
Insert into Employee values (2, 'Mary', 'Female', 60000, 3)
Insert into Employee values (4, 'John', 'Male', 56000, 1)
Insert into Employee values (5, 'Sara', 'Female', 39000, 2)
Go
```

We want to retrieve all the matching rows between **Department** and **Employee** tables.

DepartmentName	Name	Gender	Salary
IT	Mark	Male	50000
Payroll	Mary	Female	60000
HR	Steve	Male	45000
IT	John	Male	56000
HR	Sara	Female	39000

This can be very easily achieved using an Inner Join as shown below.

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
Inner Join Employee E
On D.Id = E.DepartmentId
```

Now if we want to retrieve all the matching rows between **Department** and **Employee** tables + the non-matching rows from the LEFT table (**Department**)

DepartmentName	Name	Gender	Salary
IT	Mark	Male	50000
IT	John	Male	56000
HR	Steve	Male	45000
HR	Sara	Female	39000
Payroll	Mary	Female	60000
Administration	NULL	NULL	NULL
Sales	NULL	NULL	NULL

This can be very easily achieved using a Left Join as shown below.

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
Left Join Employee E
On D.Id = E.DepartmentId
```

Now let's assume we do not have access to the Employee table. Instead we have access to the following Table Valued function, that returns all employees belonging to a department by Department Id.

```
Create function fn_GetEmployeesByDepartmentId(@DepartmentId int)
Returns Table
as
Return
(
   Select Id, Name, Gender, Salary, DepartmentId
   from Employee where DepartmentId = @DepartmentId
)
Go
```

The following query returns the employees of the department with Id =1.

```
Select * from fn_GetEmployeesByDepartmentId(1)
```

Now if you try to perform an Inner or Left join between **Department** table and **fn_GetEmployeesByDepartmentId**() function you will get an error.

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
Inner Join fn_GetEmployeesByDepartmentId(D.Id) E
On D.Id = E.DepartmentId
```

If you execute the above query you will get the following error

```
Msg 4104, Level 16, State 1, Line 3
The multi-part identifier "D.Id" could not be bound.
```

This is where we use **Cross Apply** and **Outer Apply** operators. **Cross Apply** is semantically equivalent to **Inner Join** and **Outer Apply** is semantically equivalent to **Left Outer Join.** Just like Inner Join, Cross Apply retrieves only the matching rows from the Department table and fn_GetEmployeesByDepartmentId() table valued function.

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
Cross Apply fn_GetEmployeesByDepartmentId(D.Id) E
```

Just like Left Outer Join, Outer Apply retrieves all matching rows from the Department table and fn_GetEmployeesByDepartmentId() table valued function + non-matching rows from the left table (Department)

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
```

```
from Department D
Outer Apply fn_GetEmployeesByDepartmentId(D.Id) E
```

How does Cross Apply and Outer Apply work

- The APPLY operator introduced in SQL Server 2005, is used to join a table to a table-valued function.
- The Table Valued Function on the right hand side of the APPLY operator gets called for each row from the left (also called outer table) table.
- Cross Apply returns only matching rows (semantically equivalent to Inner Join)
- Outer Apply returns matching + non-matching rows (semantically equivalent to Left Outer Join). The unmatched columns of the table valued function will be set to NULL.

- **DDL Triggers in sql server**
- **Suggested Videos**
  Part 89 - Intersect operator in sql server
  Part 90 - Difference between union intersect and except in sql server
  Part 91 - Cross apply and outer apply in sql server

In this video we will discuss **DDL Triggers in sql server**.
In SQL Server there are 4 types of triggers
1. DML Triggers - Data Manipulation Language. Discussed in Parts 43 to 47 of SQL Server Tutorial.
2. DDL Triggers - Data Definition Language
3. CLR triggers - Common Language Runtime
4. Logon triggers

What are DDL triggers
DDL triggers fire in response to DDL events - CREATE, ALTER, and DROP (Table, Function, Index, Stored Procedure etc...). For the list of all DDL events please visit https://msdn.microsoft.com/en-us/library/bb522542.aspx

Certain system stored procedures that perform DDL-like operations can also fire DDL triggers. Example - sp_rename system stored procedure

What is the use of DDL triggers

- If you want to execute some code in response to a specific DDL event
- To prevent certain changes to your database schema
- Audit the changes that the users are making to the database structure

Syntax for creating DDL trigger

```
CREATE TRIGGER [Trigger_Name]
ON [Scope (Server|Database)]
FOR [EventType1, EventType2, EventType3, ...],
AS
BEGIN
  -- Trigger Body
END
```

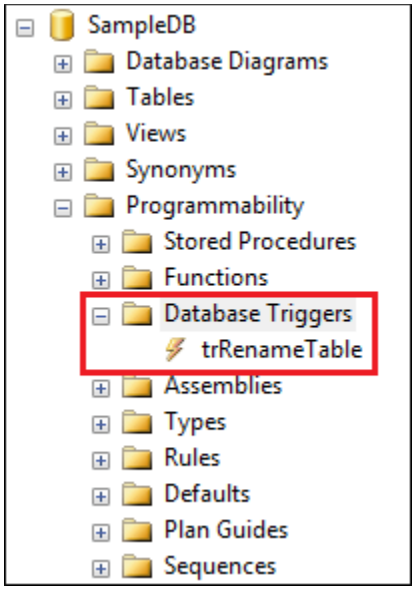
DDL triggers scope : DDL triggers can be created in a specific database or at the server level.

The following trigger will fire in response to CREATE_TABLE DDL event.

```
CREATE TRIGGER trMyFirstTrigger
ON Database
FOR CREATE_TABLE
AS
BEGIN
  Print 'New table created'
END
```

To check if the trigger has been created

1. In the Object Explorer window, expand the **SampleDB** database by clicking on the plus symbol.
2. Expand **Programmability** folder
3. Expand **Database Triggers** folder



Please note : If you can't find the trigger that you just created, make sure to refresh the Database Triggers folder.

When you execute the following code to create the table, the trigger will automatically fire and will print the message - New table created

```
Create Table Test (Id int)
```

The above trigger will be fired only for one DDL event CREATE_TABLE. If you want this trigger to be fired for multiple events, for example when you alter or drop a table, then separate the events using a comma as shown below.

```
ALTER TRIGGER trMyFirstTrigger
ON Database
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
```

```
   Print 'A table has just been created, modified or deleted'
END
```

Now if you create, alter or drop a table, the trigger will fire automatically and you will get the message - A table has just been created, modified or deleted.

The 2 DDL triggers above execute some code in response to DDL events
Now let us look at an example of how to prevent users from creating, altering or dropping tables. To do this modify the trigger as shown below.

```
ALTER TRIGGER trMyFirstTrigger
ON Database
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
   Rollback
   Print 'You cannot create, alter or drop a table'
END
```

To be able to create, alter or drop a table, you either have to disable or delete the trigger.

To disable trigger
1. Right click on the trigger in object explorer and select **"Disable"** from the context menu
2. You can also disable the trigger using the following T-SQL command

```
DISABLE TRIGGER trMyFirstTrigger ON DATABASE
```

To enable trigger
1. Right click on the trigger in object explorer and select "Enable" from the context menu
2. You can also enable the trigger using the following T-SQL command

```
ENABLE TRIGGER trMyFirstTrigger ON DATABASE
```

To drop trigger
1. Right click on the trigger in object explorer and select "Delete" from the context menu
2. You can also drop the trigger using the following T-SQL command

```
DROP TRIGGER trMyFirstTrigger ON DATABASE
```

Certain system stored procedures that perform DDL-like operations can also fire DDL triggers. The following trigger will be fired when ever you rename a database object using sp_rename system stored procedure.

```
CREATE TRIGGER trRenameTable
ON DATABASE
FOR RENAME
AS
BEGIN
    Print 'You just renamed something'
END
```

The following code changes the name of the TestTable to NewTestTable. When this code is executed, it will fire the trigger trRenameTable

```
sp_rename 'TestTable', 'NewTestTable'
```

The following code changes the name of the Id column in NewTestTable to NewId. When this code is executed, it will fire the trigger trRenameTable

```
sp_rename 'NewTestTable.Id' , 'NewId', 'column'
```

Server-scoped ddl triggers

Suggested Videos

Part 90 - Difference between union intersect and except in sql server
Part 91 - Cross apply and outer apply in sql server
Part 92 - DDL Triggers in sql server

In this video we will discuss **server-scoped ddl triggers**

The following trigger is a database scoped trigger. This will prevent users from creating, altering or dropping tables only from the database in which it is created.

```
CREATE TRIGGER tr_DatabaseScopeTrigger
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
   ROLLBACK
   Print 'You cannot create, alter or drop a table in the current database'
END
```

If you have another database on the server, they will be able to create, alter or drop tables in that database. If you want to prevent users from doing this you may create the trigger again in this database.

But, what if you have 100 different databases on your SQL Server, and you want to prevent users from creating, altering or dropping tables from all these 100 databases. Creating the same trigger for all the 100 different databases is not a good approach for 2 reasons.

1. It is tedious and error prone
2. Maintainability is a night mare. If for some reason you have to change the trigger, you will have to do it in 100 different databases, which again is tedious and error prone.

This is where server-scoped DDL triggers come in handy. When you create a server scoped DDL trigger, it will fire in response to the DDL events happening in all of the databases on that server.

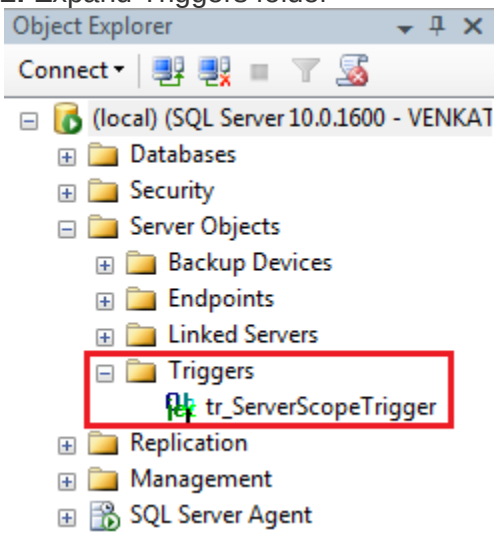
Creating a Server-scoped DDL trigger : Similar to creating a database scoped trigger, except that you will have to change the scope to ALL Server as shown below.

```
CREATE TRIGGER tr_ServerScopeTrigger
ON ALL SERVER
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
   ROLLBACK
   Print 'You cannot create, alter or drop a table in any database on the server'
END
```

Now if you try to create, alter or drop a table in any of the databases on the server, the trigger

will be fired.

Where can I find the Server-scoped DDL triggers
1. In the Object Explorer window, expand "Server Objects" folder
2. Expand Triggers folder



To disable Server-scoped ddl trigger
1. Right click on the trigger in object explorer and select "Disable" from the context menu
2. You can also disable the trigger using the following T-SQL command

```
DISABLE TRIGGER tr_ServerScopeTrigger ON ALL SERVER
```

To enable Server-scoped ddl trigger
1. Right click on the trigger in object explorer and select "Enable" from the context menu
2. You can also enable the trigger using the following T-SQL command

```
ENABLE TRIGGER tr_ServerScopeTrigger ON ALL SERVER
```

To drop Server-scoped ddl trigger
1. Right click on the trigger in object explorer and select "Delete" from the context menu
2. You can also drop the trigger using the following T-SQL command

```
DROP TRIGGER tr_ServerScopeTrigger ON ALL SERVER
```

sql server trigger execution order

Suggested Videos
Part 91 - Cross apply and outer apply in sql server
Part 92 - DDL Triggers in sql server
Part 93 - Server-scoped ddl triggers
In this video we will discuss **how to set the execution order of triggers** using **sp_settriggerorder** stored procedure. **Server scoped triggers will always fire before any of the database scoped triggers**.

This execution order cannot be changed.

In the example below, we have a database-scoped and a server-scoped trigger handling the same event (CREATE_TABLE). When you create a table, notice that server-scoped trigger is

always fired before the database-scoped trigger.

```
CREATE TRIGGER tr_DatabaseScopeTrigger
ON DATABASE
FOR CREATE_TABLE
AS
BEGIN
   Print 'Database Scope Trigger'
END
GO

CREATE TRIGGER tr_ServerScopeTrigger
ON ALL SERVER
FOR CREATE_TABLE
AS
BEGIN
   Print 'Server Scope Trigger'
END
GO
```

Using the **sp_settriggerorder** stored procedure, you can set the execution order of server-scoped or database-scoped triggers.

sp_settriggerorder stored procedure has 4 parameters

Parameter	Description
@triggername	Name of the trigger
@order	Value can be First, Last or None. When set to None, trigger is fired in random order
@stmttype	SQL statement that fires the trigger. Can be INSERT, UPDATE, DELETE or any DDL event
@namespace	Scope of the trigger. Value can be DATABASE, SERVER, or NULL

```
EXEC sp_settriggerorder
@triggername = 'tr_DatabaseScopeTrigger1',
@order = 'none',
@stmttype = 'CREATE_TABLE',
@namespace = 'DATABASE'
GO
```

If you have a database-scoped and a server-scoped trigger handling the same event, and if you have set the execution order at both the levels. Here is the execution order of the triggers.
1. The server-scope trigger marked First
2. Other server-scope triggers
3. The server-scope trigger marked Last
4. The database-scope trigger marked First
5. Other database-scope triggers
6. The database-scope trigger marked Last

Audit table changes in sql server

Suggested Videos
Part 92 - DDL Triggers in sql server
Part 93 - Server-scoped ddl triggers
Part 94 - SQL Server trigger execution order
In this video we will discuss, **how to audit table changes in SQL Server using a DDL trigger**.
Table to store the audit data

```
Create table TableChanges
(
   DatabaseName nvarchar(250),
   TableName nvarchar(250),
   EventType nvarchar(250),
   LoginName nvarchar(250),
   SQLCommand nvarchar(2500),
   AuditDateTime datetime
)
Go
```

The following trigger audits all table changes in all databases on a SQL Server

```
CREATE TRIGGER tr_AuditTableChanges
ON ALL SERVER
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
   DECLARE @EventData XML
   SELECT @EventData = EVENTDATA()

   INSERT INTO SampleDB.dbo.TableChanges
   (DatabaseName, TableName, EventType, LoginName,
    SQLCommand, AuditDateTime)
   VALUES
   (
       @EventData.value('(/EVENT_INSTANCE/DatabaseName)[1]', 'varchar(250)'),
       @EventData.value('(/EVENT_INSTANCE/ObjectName)[1]', 'varchar(250)'),
       @EventData.value('(/EVENT_INSTANCE/EventType)[1]', 'nvarchar(250)'),
       @EventData.value('(/EVENT_INSTANCE/LoginName)[1]', 'varchar(250)'),
       @EventData.value('(/EVENT_INSTANCE/TSQLCommand)[1]', 'nvarchar(2500)'),
       GetDate()
   )
END
```

In the above example we are using **EventData**() function which returns event data in XML format. The following XML is returned by the **EventData**() function when I created a table with name = **MyTable** in **SampleDB** database.

```
<EVENT_INSTANCE>
  <EventType>CREATE_TABLE</EventType>
  <PostTime>2015-09-11T16:12:49.417</PostTime>
  <SPID>58</SPID>
  <ServerName>VENKAT-PC</ServerName>
  <LoginName>VENKAT-PC\Tan</LoginName>
  <UserName>dbo</UserName>
```

```
  <DatabaseName>SampleDB</DatabaseName>
  <SchemaName>dbo</SchemaName>
  <ObjectName>MyTable</ObjectName>
  <ObjectType>TABLE</ObjectType>
  <TSQLCommand>
    <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON"
                ANSI_PADDING="ON" QUOTED_IDENTIFIER="ON"
                ENCRYPTED="FALSE" />
    <CommandText>
      Create Table MyTable
      (
        Id int,
        Name nvarchar(50),
        Gender nvarchar(50)
      )
    </CommandText>
  </TSQLCommand>
</EVENT_INSTANCE>
```

Logon triggers in sql server

Suggested Videos

Part 93 - Server-scoped ddl triggers
Part 94 - SQL Server trigger execution order
Part 95 - Audit table changes in sql server

In this video we will discuss **Logon triggers in SQL Server**.
As the name implies **Logon triggers fire in response to a LOGON event**. Logon triggers fire after the authentication phase of logging in finishes, but before the user session is actually established.

Logon triggers can be used for
1. Tracking login activity
2. Restricting logins to SQL Server
3. Limiting the number of sessions for a specific login

Logon trigger example : The following trigger limits the maximum number of open connections for a user to 3.

```
CREATE TRIGGER tr_LogonAuditTriggers
ON ALL SERVER
FOR LOGON
AS
BEGIN
   DECLARE @LoginName NVARCHAR(100)

   Set @LoginName = ORIGINAL_LOGIN()

   IF (SELECT COUNT(*) FROM sys.dm_exec_sessions
       WHERE is_user_process = 1
       AND original_login_name = @LoginName) > 3
   BEGIN
       Print 'Fourth connection of ' + @LoginName + ' blocked'
```

```
        ROLLBACK
    END
END
```

An attempt to make a fourth connection, will be blocked.

The trigger error message will be written to the error log. Execute the following command to read the error log.

```
Execute sp_readerrorlog
```

LogData	ProcessInfo	Text
13/09/2015	spid54	Fourth connection of VENKAT-PC\Tan blocked
13/09/2015	spid54	Error: 3609, Severity: 16, State: 2.
13/09/2015	spid54	The transaction ended in the trigger. The batc

Select into in sql server

Suggested Videos

Part 94 - SQL Server trigger execution order
Part 95 - Audit table changes in sql server
Part 96 - Logon triggers in sql server

In this video we will discuss the power and use of **SELECT INTO statement in SQL Server**.

We will be using the following 2 tables for the examples.

Departments Table	
DepartmentId	**DepartmentName**
1	IT
2	HR
3	Payroll

Employees Table				
Id	**Name**	**Gender**	**Salary**	**DeptId**
1	Mark	Male	50000	1
2	Sara	Female	65000	2
3	Mike	Male	48000	3
4	Pam	Female	70000	1
5	John	Male	55000	2

SQL Script to create Departments and Employees tables

```
Create table Departments
(
    DepartmentId int primary key,
```

```
    DepartmentName nvarchar(50)
)
Go

Insert into Departments values (1, 'IT')
Insert into Departments values (2, 'HR')
Insert into Departments values (3, 'Payroll')
Go

Create table Employees
(
    Id int primary key,
    Name nvarchar(100),
    Gender nvarchar(10),
    Salary int,
    DeptId int foreign key references Departments(DepartmentId)
)
Go

Insert into Employees values (1, 'Mark', 'Male', 50000, 1)
Insert into Employees values (2, 'Sara', 'Female', 65000, 2)
Insert into Employees values (3, 'Mike', 'Male', 48000, 3)
Insert into Employees values (4, 'Pam', 'Female', 70000, 1)
Insert into Employees values (5, 'John', 'Male', 55000, 2)
Go
```

The **SELECT INTO statement in SQL Server**, selects data from one table and inserts it into a new table.

SELECT INTO statement in SQL Server can do the following

1. Copy all rows and columns from an existing table into a new table. This is extremely useful when you want to make a backup copy of the existing table.

```
SELECT * INTO EmployeesBackup FROM Employees
```

2. Copy all rows and columns from an existing table into a new table in an external database.

```
SELECT * INTO HRDB.dbo.EmployeesBackup FROM Employees
```

3. Copy only selected columns into a new table

```
SELECT Id, Name, Gender INTO EmployeesBackup FROM Employees
```

4. Copy only selected rows into a new table

```
SELECT * INTO EmployeesBackup FROM Employees WHERE DeptId = 1
```

5. Copy columns from 2 or more table into a new table

```
SELECT * INTO EmployeesBackup
FROM Employees
INNER JOIN Departments
ON Employees.DeptId = Departments.DepartmentId
```

6. Create a new table whose columns and datatypes match with an existing table.

```
SELECT * INTO EmployeesBackup FROM Employees WHERE 1 <> 1
```

7. Copy all rows and columns from an existing table into a new table on a different SQL Server instance. For this, create a linked server and use the 4 part naming convention
```
SELECT * INTO TargetTable
FROM [SourceServer].[SourceDB].[dbo].[SourceTable]
```

Please note : You cannot use SELECT INTO statement to select data into an existing table. For this you will have to use INSERT INTO statement.

```
INSERT INTO ExistingTable (ColumnList)
Insert into Employees values (3, 'Mike', 'Male', 48000, 3)
Insert into Employees values (4, 'Pam', 'Female', 70000, 1)
Insert into Employees values (5, 'John', 'Male', 55000, 2)
Go
```

The **SELECT INTO statement in SQL Server**, selects data from one table and inserts it into a new table.

SELECT INTO statement in SQL Server can do the following
1. Copy all rows and columns from an existing table into a new table. This is extremely useful when you want to make a backup copy of the existing table.
```
SELECT * INTO EmployeesBackup FROM Employees
```

2. Copy all rows and columns from an existing table into a new table in an external database.
```
SELECT * INTO HRDB.dbo.EmployeesBackup FROM Employees
```

3. Copy only selected columns into a new table
```
SELECT Id, Name, Gender INTO EmployeesBackup FROM Employees
```

4. Copy only selected rows into a new table
```
SELECT * INTO EmployeesBackup FROM Employees WHERE DeptId = 1
```

5. Copy columns from 2 or more table into a new table
```
SELECT * INTO EmployeesBackup
FROM Employees
INNER JOIN Departments
ON Employees.DeptId = Departments.DepartmentId
```

6. Create a new table whose columns and datatypes match with an existing table.
```
SELECT * INTO EmployeesBackup FROM Employees WHERE 1 <> 1
```

7. Copy all rows and columns from an existing table into a new table on a different SQL Server instance. For this, create a linked server and use the 4 part naming convention
```
SELECT * INTO TargetTable
FROM [SourceServer].[SourceDB].[dbo].[SourceTable]
```

Please note : You cannot use SELECT INTO statement to select data into an existing table. For this you will have to use INSERT INTO statement.

```
INSERT INTO ExistingTable (ColumnList)
Insert into Employees values (3, 'Mike', 'Male', 48000, 3)
```

```
Insert into Employees values (4, 'Pam', 'Female', 70000, 1)
Insert into Employees values (5, 'John', 'Male', 55000, 2)
Go
```

Difference between where and having in sql server

Suggested Videos

Part 95 - Audit table changes in sql server
Part 96 - Logon triggers in sql server
Part 97 - Select into in sql server

In this video we will discuss the **difference between where and having** clauses in SQL Server. Let us understand the difference with an example. For the examples in this video we will use the following Sales table.

Sales	
Product	**SaleAmount**
iPhone	500
Laptop	800
iPhone	1000
Speakers	400
Laptop	600

SQL Script to create and populate Sales table with test data

```
Create table Sales
(
   Product nvarchar(50),
   SaleAmount int
)
Go

Insert into Sales values ('iPhone', 500)
Insert into Sales values ('Laptop', 800)
Insert into Sales values ('iPhone', 1000)
Insert into Sales values ('Speakers', 400)
Insert into Sales values ('Laptop', 600)
Go
```

To calculate total sales by product, we would write a GROUP BY query as shown below

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
```

The above query produces the following result

Product	TotalSales
iPhone	1500
Laptop	1400
Speakers	400

Now if we want to find only those **products where the total sales amount is greater than $1000**, we will use HAVING clause to filter products

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
HAVING SUM(SaleAmount) > 1000
```

Result :

Product	TotalSales
iPhone	1500
Laptop	1400

If we use WHERE clause instead of HAVING clause, we will get a syntax error. This is because the WHERE clause doesn't work with aggregate functions like sum, min, max, avg, etc.

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
WHERE SUM(SaleAmount) > 1000
```

So in short, the difference is **WHERE clause cannot be used with aggregates where as HAVING can.**

However, there are other differences as well that we need to keep in mind when using WHERE and HAVING clauses. WHERE clause filters rows before aggregate calculations are performed where as HAVING clause filters rows after aggregate calculations are performed. Let us understand this with an example.

Total sales of iPhone and Speakers can be calculated by using either WHERE or HAVING clause

Calculate Total sales of iPhone and Speakers using WHERE clause : In this example the WHERE clause retrieves only iPhone and Speaker products and then performs the sum.

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
WHERE Product in ('iPhone', 'Speakers')
GROUP BY Product
```

Result :

Product	TotalSales
iPhone	1500
Speakers	400

Calculate Total sales of iPhone and Speakers using HAVING clause : This example retrieves all rows from Sales table, performs the sum and then removes all products except iPhone and Speakers.

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
HAVING Product in ('iPhone', 'Speakers')
```

Result :

Product	TotalSales
iPhone	1500
Speakers	400

So from a performance standpoint, HAVING is slower than WHERE and should be avoided when possible.

Another difference is WHERE comes before GROUP BY and HAVING comes after GROUP BY.

Difference between WHERE and Having

1. WHERE clause cannot be used with aggregates where as HAVING can. This means WHERE clause is used for filtering individual rows where as HAVING clause is used to filter groups.

2. WHERE comes before GROUP BY. This means WHERE clause filters rows before aggregate calculations are performed. HAVING comes after GROUP BY. This means HAVING clause filters rows after aggregate calculations are performed. So from a performance standpoint, HAVING is slower than WHERE and should be avoided when possible.

3. WHERE and HAVING can be used together in a SELECT query. In this case WHERE clause is applied first to filter individual rows. The rows are then grouped and aggregate calculations are performed, and then the HAVING clause filters the groups.

Table valued parameters in SQL Server

Suggested Videos

Part 96 - Logon triggers in sql server
Part 97 - Select into in sql server
Part 98 - Difference between where and having in sql server

Table Valued Parameter is a new feature introduced in SQL SERVER 2008. Table Valued Parameter allows a table (i.e multiple rows of data) to be passed as a parameter to a stored procedure from T-SQL code or from an application. Prior to SQL SERVER 2008, it is not possible to pass a table variable as a parameter to a stored procedure.

Let us understand how to pass multiple rows to a stored procedure using Table Valued Parameter with an example. We want to insert multiple rows into the following Employees table.

At the moment this table does not have any rows.

Employees		
Id	Name	Gender

SQL Script to create the Employees table

```
Create Table Employees
(
   Id int primary key,
   Name nvarchar(50),
   Gender nvarchar(10)
)
Go
```

Step 1 : Create User-defined Table Type

```
CREATE TYPE EmpTableType AS TABLE
(
   Id INT PRIMARY KEY,
   Name NVARCHAR(50),
   Gender NVARCHAR(10)
)
Go
```

Step 2 : Use the User-defined Table Type as a parameter in the stored procedure. Table valued parameters must be passed as read-only to stored procedures, functions etc. This means you cannot perform DML operations like INSERT, UPDATE or DELETE on a table-valued parameter in the body of a function, stored procedure etc.

```
CREATE PROCEDURE spInsertEmployees
@EmpTableType EmpTableType READONLY
AS
BEGIN
   INSERT INTO Employees
   SELECT * FROM @EmpTableType
END
```

Step 3 : Declare a table variable, insert the data and then pass the table variable as a parameter to the stored procedure.

```
DECLARE @EmployeeTableType EmpTableType

INSERT INTO @EmployeeTableType VALUES (1, 'Mark', 'Male')
INSERT INTO @EmployeeTableType VALUES (2, 'Mary', 'Female')
INSERT INTO @EmployeeTableType VALUES (3, 'John', 'Male')
INSERT INTO @EmployeeTableType VALUES (4, 'Sara', 'Female')
INSERT INTO @EmployeeTableType VALUES (5, 'Rob', 'Male')

EXECUTE spInsertEmployees @EmployeeTableType
```

That's it. Now select the data from Employees table and notice that all the rows of the table variable are inserted into the Employees table.

Employees		
Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	John	Male
4	Sara	Female
5	Rob	Male

In our next video, we will discuss **how to pass table as a parameter to the stored procedure from an ADO.NET application**

Send datatable as parameter to stored procedure

Suggested Videos

Part 97 - Select into in sql server
Part 98 - Difference between where and having in sql server
Part 99 - Table valued parameters in SQL Server

In this video we will discuss **how to send datatable as parameter to stored procedure**. This is continuation to Part 99. Please watch Part 99 from SQL Server tutorial before proceeding.

In Part 99, we discussed creating a stored procedure that accepts a table as a parameter. In this video we will discuss **how to pass a datatable from a web application to the SQL Server stored procedure**.

Here is what we want to do.

1. Design a webform that looks as shown below. This form allows us to insert 5 employees at a time into the database table.

ID : Name : Gender :
ID : Name : Gender :
ID : Name : Gender :
ID : Name : Gender :
ID : Name : Gender :

Insert Employees

2. When "Insert Employees" button is clicked, retrieve the from data into a datatabe and then pass the datatable as a parameter to the stored procedure.

3. The stored procedure will then insert all the rows into the Employees table in the database.

Here are the steps to achieve this.

Step 1 : Create new asp.net web application project. Name it Demo.

Step 2 : Include a connection string in the web.config file to your database.

```
<add name="DBCS"
     connectionString="server=.;database=SampleDB;integrated security=SSPI"/>
```

Step 3 : Copy and paste the following HTML in WebForm1.aspx

```
<asp:Button ID="btnFillDummyData" runat="server" Text="Fill Dummy Data"
   OnClick="btnFillDummyData_Click" />
<br /><br />
<table>
   <tr>
      <td>
         ID : <asp:TextBox ID="txtId1" runat="server"></asp:TextBox>
      </td>
      <td>
         Name : <asp:TextBox ID="txtName1" runat="server"></asp:TextBox>
      </td>
      <td>
         Gender : <asp:TextBox ID="txtGender1" runat="server"></asp:TextBox>
      </td>
   </tr>
   <tr>
      <td>
         ID : <asp:TextBox ID="txtId2" runat="server"></asp:TextBox>
      </td>
      <td>
         Name : <asp:TextBox ID="txtName2" runat="server"></asp:TextBox>
      </td>
      <td>
         Gender : <asp:TextBox ID="txtGender2" runat="server"></asp:TextBox>
      </td>
   </tr>
   <tr>
      <td>
         ID : <asp:TextBox ID="txtId3" runat="server"></asp:TextBox>
      </td>
      <td>
         Name : <asp:TextBox ID="txtName3" runat="server"></asp:TextBox>
      </td>
      <td>
         Gender : <asp:TextBox ID="txtGender3" runat="server"></asp:TextBox>
      </td>
   </tr>
   <tr>
      <td>
         ID : <asp:TextBox ID="txtId4" runat="server"></asp:TextBox>
      </td>
      <td>
         Name : <asp:TextBox ID="txtName4" runat="server"></asp:TextBox>
      </td>
```

```
        <td>
            Gender : <asp:TextBox ID="txtGender4" runat="server"></asp:TextBox>
        </td>
    </tr>
    <tr>
        <td>
            ID : <asp:TextBox ID="txtId5" runat="server"></asp:TextBox>
        </td>
        <td>
            Name : <asp:TextBox ID="txtName5" runat="server"></asp:TextBox>
        </td>
        <td>
            Gender : <asp:TextBox ID="txtGender5" runat="server"></asp:TextBox>
        </td>
    </tr>
</table>
<br />
<asp:Button ID="btnInsert" runat="server" Text="Insert Employees"
    OnClick="btnInsert_Click" />
```

Step 4 : Copy and paste the following code in the code-behind file

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }

        private DataTable GetEmployeeData()
        {
            DataTable dt = new DataTable();
            dt.Columns.Add("Id");
            dt.Columns.Add("Name");
            dt.Columns.Add("Gender");

            dt.Rows.Add(txtId1.Text, txtName1.Text, txtGender1.Text);
            dt.Rows.Add(txtId2.Text, txtName2.Text, txtGender2.Text);
       dt.Rows.Add(txtId4.Text, txtName4.Text, txtGender4.Text);
            dt.Rows.Add(txtId5.Text, txtName5.Text, txtGender5.Text);

            return dt;
        }

        protected void btnInsert_Click(object sender, EventArgs e)
        {
            string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
```

```
        using (SqlConnection con = new SqlConnection(cs))
        {
            SqlCommand cmd = new SqlCommand("spInsertEmployees", con);
            cmd.CommandType = CommandType.StoredProcedure;

            SqlParameter paramTVP = new SqlParameter()
            {
                ParameterName = "@EmpTableType",
                Value = GetEmployeeData()
            };
            cmd.Parameters.Add(paramTVP);

            con.Open();
            cmd.ExecuteNonQuery();
            con.Close();
        }
    }

    protected void btnFillDummyData_Click(object sender, EventArgs e)
    {
        txtId1.Text = "1";
        txtId2.Text = "2";
        txtId3.Text = "3";
        txtId4.Text = "4";
        txtId5.Text = "5";

        txtName1.Text = "John";
        txtName2.Text = "Mike";
        txtName3.Text = "Sara";
        txtName4.Text = "Pam";
        txtName5.Text = "Todd";

        txtGender1.Text = "Male";
        txtGender2.Text = "Male";
        txtGender3.Text = "Female";
        txtGender4.Text = "Female";
        txtGender5.Text = "Male";
    }
  }
}
```

Grouping Sets in SQL Server

Suggested Videos

Part 98 - Difference between where and having in sql server
Part 99 - Table valued parameters in SQL Server
Part 100 - Send datatable as parameter to stored procedure

Grouping sets is a new feature introduced in SQL Server 2008. Let us understand Grouping sets with an example.

We will be using the following **Employees table** for the examples in this video.

Employees Table				
Id	Name	Gender	Salary	Country
1	Mark	Male	5000	USA
2	John	Male	4500	India
3	Pam	Female	5500	USA
4	Sara	Female	4000	India
5	Todd	Male	3500	India
6	Mary	Female	5000	UK
7	Ben	Male	6500	UK
8	Elizabeth	Female	7000	USA
9	Tom	Male	5500	UK
10	Ron	Male	5000	USA

SQL Script to create and populate Employees table

```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int,
    Country nvarchar(10)
)
Go

Insert Into Employees Values (1, 'Mark', 'Male', 5000, 'USA')
Insert Into Employees Values (2, 'John', 'Male', 4500, 'India')
Insert Into Employees Values (4, 'Sara', 'Female', 4000, 'India')
Insert Into Employees Values (5, 'Todd', 'Male', 3500, 'India')
Insert Into Employees Values (6, 'Mary', 'Female', 5000, 'UK')
Insert Into Employees Values (7, 'Ben', 'Male', 6500, 'UK')
Insert Into Employees Values (8, 'Elizabeth', 'Female', 7000, 'USA')
Insert Into Employees Values (9, 'Tom', 'Male', 5500, 'UK')
Insert Into Employees Values (10, 'Ron', 'Male', 5000, 'USA')
Go
```

We want to calculate **Sum of Salary by Country and Gender**. The result should be as shown below.

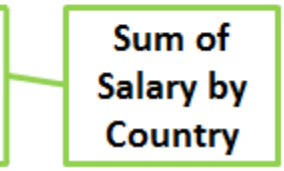
Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000

We can very easily achieve this using a Group By query as shown below

```
Select Country, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Country, Gender
```

Within the same result set we also want Sum of Salary just by Country. The Result should be as shown below. Notice that Gender column within the resultset is NULL as we are grouping only by Country column

Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000
India	NULL	12000
UK	NULL	17000
USA	NULL	22500

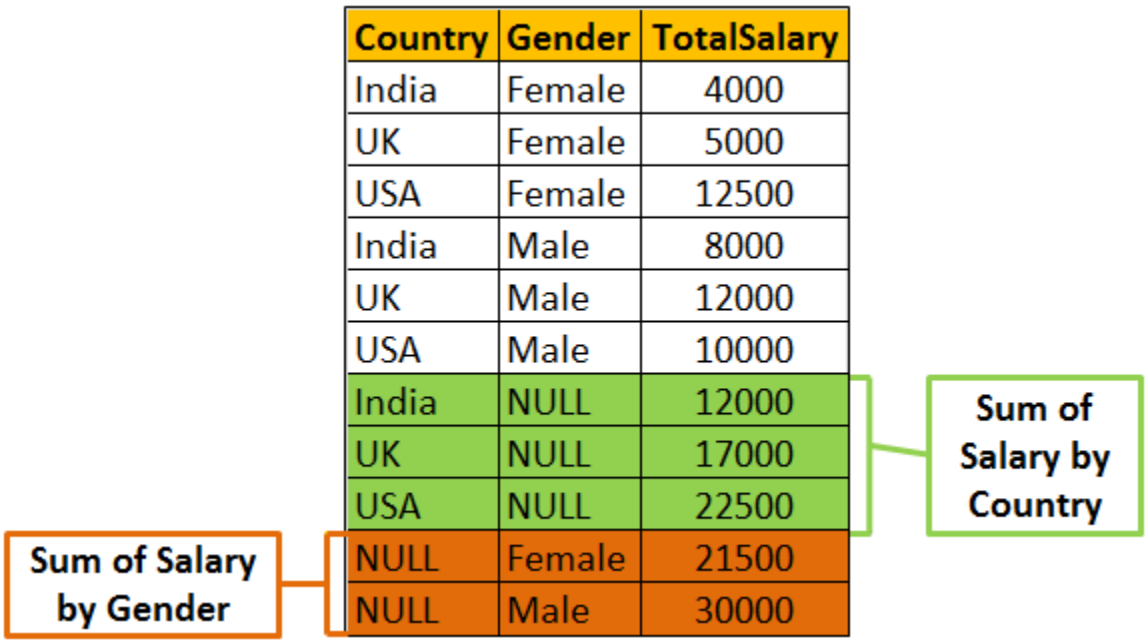


To achieve the above result we could combine 2 Group By queries using UNION ALL as shown below.

```
Select Country, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Country, Gender

UNION ALL

Select Country, NULL, Sum(Salary) as TotalSalary
From Employees
Group By Country
```

Within the same result set we also want Sum of Salary just by Gender. The Result should be as

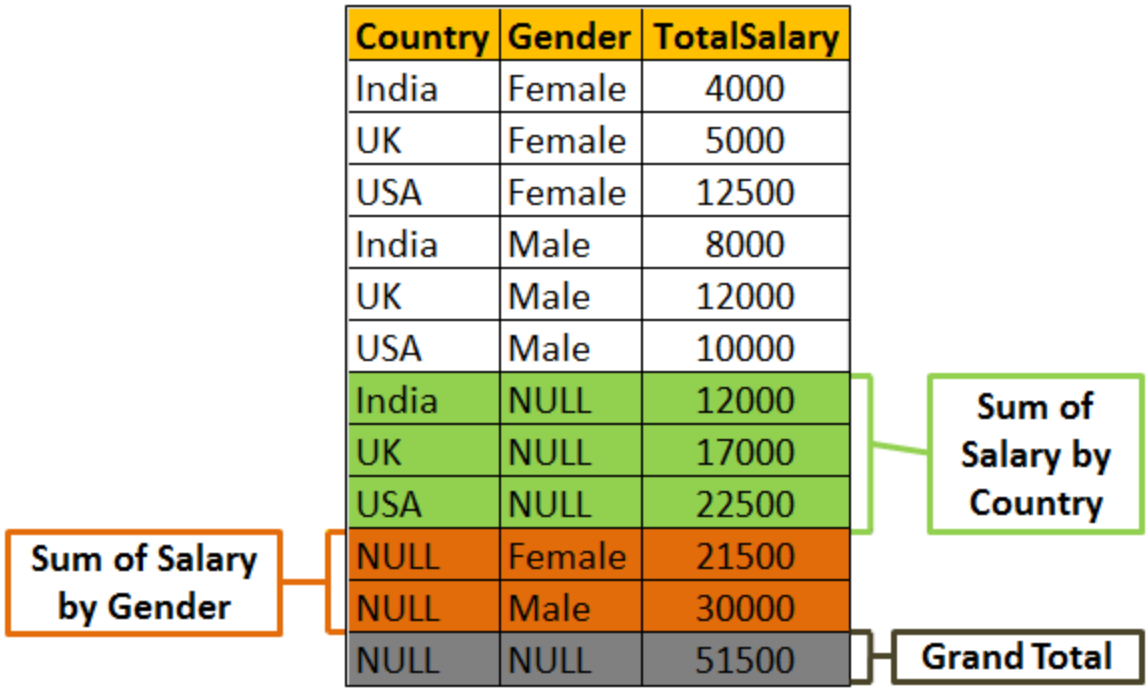
shown below. Notice that the Country column within the resultset is NULL as we are grouping only by Gender column.



We can achieve this by combining 3 Group By queries using UNION ALL as shown below

```
Select Country, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Country, Gender

UNION ALL

Select Country, NULL, Sum(Salary) as TotalSalary
From Employees
Group By Country

UNION ALL

Select NULL, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Gender
```

Finally we also want the grand total of Salary. In this case we are not grouping on any particular column. So both Country and Gender columns will be NULL in the resultset.



To achieve this we will have to combine the fourth query using UNION ALL as shown below.

```
Select Country, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Country, Gender

UNION ALL

Select Country, NULL, Sum(Salary) as TotalSalary
From Employees
Group By Country

UNION ALL

Select NULL, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Gender

UNION ALL

Select NULL, NULL, Sum(Salary) as TotalSalary
From Employees
```

There are 2 problems with the above approach.
1. The query is huge as we have combined different Group By queries using UNION ALL operator. This can grow even more if we start to add more groups
2. The Employees table has to be accessed 4 times, once for every query.

If we use **Grouping Sets** feature introduced in SQL Server 2008, the amount of T-SQL code that you have to write will be greatly reduced. The following Grouping Sets query produce the same result as the above UNION ALL query.

```
Select Country, Gender, Sum(Salary) TotalSalary
From Employees
Group BY
    GROUPING SETS
    (
        (Country, Gender), -- Sum of Salary by Country and Gender
        (Country),         -- Sum of Salary by Country
        (Gender) ,         -- Sum of Salary by Gender
        ()                 -- Grand Total
    )
```

Output of the above query

Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
NULL	Female	21500
India	Male	8000
UK	Male	12000
USA	Male	10000
NULL	Male	30000
NULL	NULL	51500
India	NULL	12000
UK	NULL	17000
USA	NULL	22500

The order of the rows in the result set is not the same as in the case of UNION ALL query. To control the order use order by as shown below.

```
Select Country, Gender, Sum(Salary) TotalSalary
From Employees
Group BY
    GROUPING SETS
    (
        (Country, Gender), -- Sum of Salary by Country and Gender
        (Country),         -- Sum of Salary by Country
        (Gender) ,         -- Sum of Salary by Gender
        ()                 -- Grand Total
    )
```

```
Order By Grouping(Country), Grouping(Gender), Gender
```

Output of the above query

Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000
India	NULL	12000
UK	NULL	17000
USA	NULL	22500
NULL	Female	21500
NULL	Male	30000
NULL	NULL	51500

Rollup in SQL Server

Suggested Videos

Part 99 - Table valued parameters in SQL Server
Part 100 - Send datatable as parameter to stored procedure
Part 101 - Grouping sets in SQL Server

ROLLUP in SQL Server is used to do aggregate operation on multiple levels in hierarchy. Let us understand Rollup in SQL Server with examples. We will use the following **Employees table** for the examples in this video.

Employees Table				
Id	Name	Gender	Salary	Country
1	Mark	Male	5000	USA
2	John	Male	4500	India
3	Pam	Female	5500	USA
4	Sara	Female	4000	India
5	Todd	Male	3500	India
6	Mary	Female	5000	UK
7	Ben	Male	6500	UK
8	Elizabeth	Female	7000	USA
9	Tom	Male	5500	UK
10	Ron	Male	5000	USA

Retrieve Salary by country along with grand total

Country	TotalSalary
India	12000
UK	17000
USA	22500
NULL	51500

There are several ways to achieve this. The easiest way is by using Rollup with Group By.

```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees

GROUP BY ROLLUP(Country)
```

The above query can also be rewritten as shown below

```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country WITH ROLLUP
```

We can also use UNION ALL operator along with GROUP BY

```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country

UNION ALL

SELECT NULL, SUM(Salary) AS TotalSalary
FROM Employees
```

We can also use Grouping Sets to achieve the same result

```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY GROUPING SETS
(
   (Country),
   ()
)
```

Let's look at another example.

Group Salary by Country and Gender. Also compute the Subtotal for Country level and Grand Total as shown below.

Country	Gender	TotalSalary
India	Female	4000
India	Male	8000
India	NULL	12000
UK	Female	5000
UK	Male	12000
UK	NULL	17000
USA	Female	12500
USA	Male	10000
USA	NULL	22500
NULL	NULL	51500

Using ROLLUP with GROUP BY

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY ROLLUP(Country, Gender)

--OR

SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country, Gender WITH ROLLUP
```

Using UNION ALL with GROUP BY

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country, Gender

UNION ALL

SELECT Country, NULL, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country

UNION ALL

SELECT NULL, NULL, SUM(Salary) AS TotalSalary
FROM Employees
```

Using GROUPING SETS

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY GROUPING SETS
(
    (Country, Gender),
```

```
    (Country),
    ()
)
```

Cube in SQL Server

Suggested Videos

Part 100 - Send datatable as parameter to stored procedure
Part 101 - Grouping sets in SQL Server
Part 102 - Rollup in SQL Server

Cube() in SQL Server produces the result set by generating all combinations of columns specified in GROUP BY CUBE().

Let us understand Cube() in SQL Server with examples. We will use the following **Employees table** for the examples in this video.

Employees Table

Id	Name	Gender	Salary	Country
1	Mark	Male	5000	USA
2	John	Male	4500	India
3	Pam	Female	5500	USA
4	Sara	Female	4000	India
5	Todd	Male	3500	India
6	Mary	Female	5000	UK
7	Ben	Male	6500	UK
8	Elizabeth	Female	7000	USA
9	Tom	Male	5500	UK
10	Ron	Male	5000	USA

Write a query to retrieve Sum of Salary grouped by all combinations of the following 2 columns as well as Grand Total.
Country,
Gender

The output of the query should be as shown below

Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
NULL	Female	21500
India	Male	8000
UK	Male	12000
USA	Male	10000
NULL	Male	30000
NULL	NULL	51500
India	NULL	12000
UK	NULL	17000
USA	NULL	22500

Using Cube with Group By

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Cube(Country, Gender)

--OR

SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country, Gender with Cube
```

The above query is equivalent to the following Grouping Sets query

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY
   GROUPING SETS
   (
       (Country, Gender),
       (Country),
       (Gender),
       ()
   )
```

The above query is equivalent to the following UNION ALL query. While the data in the result set is the same, the ordering is not. Use ORDER BY to control the ordering of rows in the result set.

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
```

```
GROUP BY Country, Gender

UNION ALL

SELECT Country, NULL, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country

UNION ALL

SELECT NULL, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Gender

UNION ALL

SELECT NULL, NULL, SUM(Salary) AS TotalSalary
FROM Employees
```

Difference between cube and rollup in SQL Server

Suggested Videos

Part 101 - Grouping sets in SQL Server
Part 102 - Rollup in SQL Server
Part 103 - Cube in SQL Server

In this video we will discuss the **difference between cube and rollup in SQL Server**. **CUBE generates a result set** that shows aggregates for all combinations of values in the selected columns, where as ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.

Let us understand this difference with an example. Consider the following **Sales** table.

Continent	Country	City	SaleAmount
Asia	India	Bangalore	1000
Asia	India	Chennai	2000
Asia	Japan	Tokyo	4000
Asia	Japan	Hiroshima	5000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	France	Paris	4000
Europe	France	Cannes	5000

SQL Script to create and populate Sales table

```
Create table Sales
(
    Id int primary key identity,
    Continent nvarchar(50),
    Country nvarchar(50),
```

```
    City nvarchar(50),
    SaleAmount int
)
Go

Insert into Sales values('Asia','India','Bangalore',1000)
Insert into Sales values('Asia','India','Chennai',2000)
Insert into Sales values('Asia','Japan','Tokyo',4000)
Insert into Sales values('Asia','Japan','Hiroshima',5000)
Insert into Sales values('Europe','United Kingdom','London',1000)
Insert into Sales values('Europe','United Kingdom','Manchester',2000)
Insert into Sales values('Europe','France','Paris',4000)
Insert into Sales values('Europe','France','Cannes',5000)
Go
```

ROLLUP(Continent, Country, City) produces Sum of Salary for the following hierarchy
Continent, Country, City
Continent, Country,
Continent
()

CUBE(Continent, Country, City) produces Sum of Salary for all the following column combinations
Continent, Country, City
Continent, Country,
Continent, City
Continent
Country, City
Country,
City
()

```
SELECT Continent, Country, City, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
```

Continent	Country	City	TotalSales
Asia	India	Bangalore	1000
Asia	India	Chennai	2000
Asia	India	NULL	3000
Asia	Japan	Hiroshima	5000
Asia	Japan	Tokyo	4000
Asia	Japan	NULL	9000
Asia	NULL	NULL	12000
Europe	France	Cannes	5000
Europe	France	Paris	4000
Europe	France	NULL	9000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	United Kingdom	NULL	3000
Europe	NULL	NULL	12000
NULL	NULL	NULL	24000

```
SELECT Continent, Country, City, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY CUBE(Continent, Country, City)
```

Continent	Country	City	TotalSales
Asia	India	Bangalore	1000
NULL	India	Bangalore	1000
NULL	NULL	Bangalore	1000
Europe	France	Cannes	5000
NULL	France	Cannes	5000
NULL	NULL	Cannes	5000
Asia	India	Chennai	2000
NULL	India	Chennai	2000
NULL	NULL	Chennai	2000
Asia	Japan	Hiroshima	5000
NULL	Japan	Hiroshima	5000
NULL	NULL	Hiroshima	5000
Europe	United Kingdom	London	1000
NULL	United Kingdom	London	1000
NULL	NULL	London	1000
Europe	United Kingdom	Manchester	2000
NULL	United Kingdom	Manchester	2000
NULL	NULL	Manchester	2000
Europe	France	Paris	4000
NULL	France	Paris	4000
NULL	NULL	Paris	4000
Asia	Japan	Tokyo	4000
NULL	Japan	Tokyo	4000
NULL	NULL	Tokyo	4000
NULL	NULL	NULL	24000
Asia	NULL	Bangalore	1000
Asia	NULL	Chennai	2000
Asia	NULL	Hiroshima	5000
Asia	NULL	Tokyo	4000
Asia	NULL	NULL	12000
Europe	NULL	Cannes	5000
Europe	NULL	London	1000
Europe	NULL	Manchester	2000
Europe	NULL	Paris	4000
Europe	NULL	NULL	12000
Europe	France	NULL	9000

You won't see any difference when you use ROLLUP and CUBE on a single column. Both the following queries produces the same output.

```
SELECT Continent, Sum(SaleAmount) AS TotalSales
FROM Sales
GROUP BY ROLLUP(Continent)

-- OR

SELECT Continent, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY CUBE(Continent)
```

Continent	TotalSales
Asia	12000
Europe	12000
NULL	24000

Grouping function in SQL Server

Suggested Videos

Part 102 - Rollup in SQL Server
Part 103 - Cube in SQL Server
Part 104 - Difference between cube and rollup in SQL Server

In this video we will discuss the use of **Grouping function in SQL Server**.

This is continuation to Part 104. Please watch Part 104 from SQL Server tutorial before proceeding. We will use the following Sales table for this example.

Continent	Country	City	SaleAmount
Asia	India	Bangalore	1000
Asia	India	Chennai	2000
Asia	Japan	Tokyo	4000
Asia	Japan	Hiroshima	5000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	France	Paris	4000
Europe	France	Cannes	5000

What is Grouping function

Grouping(Column) indicates whether the column in a GROUP BY list is aggregated or not. Grouping returns 1 for aggregated or 0 for not aggregated in the result set.

The following query returns 1 for aggregated or 0 for not aggregated in the result set

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
        GROUPING(Continent) AS GP_Continent,
        GROUPING(Country) AS GP_Country,
        GROUPING(City) AS GP_City
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
```

Result :

Continent	Country	City	TotalSales	GP_Continent	GP_Country	GP_City
Asia	India	Bangalore	1000	0	0	0
Asia	India	Chennai	2000	0	0	0
Asia	India	NULL	3000	0	0	1
Asia	Japan	Hiroshima	5000	0	0	0
Asia	Japan	Tokyo	4000	0	0	0
Asia	Japan	NULL	9000	0	0	1
Asia	NULL	NULL	12000	0	1	1
Europe	France	Cannes	5000	0	0	0
Europe	France	Paris	4000	0	0	0
Europe	France	NULL	9000	0	0	1
Europe	United Kingdom	London	1000	0	0	0
Europe	United Kingdom	Manchester	2000	0	0	0
Europe	United Kingdom	NULL	3000	0	0	1
Europe	NULL	NULL	12000	0	1	1
NULL	NULL	NULL	24000	1	1	1

What is the use of Grouping function in real world

When a column is aggregated in the result set, the column will have a NULL value. If you want to replace NULL with All then this GROUPING function is very handy.

```
SELECT
   CASE WHEN
       GROUPING(Continent) = 1 THEN 'All' ELSE ISNULL(Continent, 'Unknown')
   END AS Continent,
   CASE WHEN
       GROUPING(Country) = 1 THEN 'All' ELSE ISNULL(Country, 'Unknown')
   END AS Country,
   CASE
       WHEN GROUPING(City) = 1 THEN 'All' ELSE ISNULL(City, 'Unknown')
   END AS City,
   SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
```

Result :

Continent	Country	City	TotalSales
Asia	India	Bangalore	1000
Asia	India	Chennai	2000
Asia	India	All	3000
Asia	Japan	Hiroshima	5000
Asia	Japan	Tokyo	4000
Asia	Japan	All	9000
Asia	All	All	12000
Europe	France	Cannes	5000
Europe	France	Paris	4000
Europe	France	All	9000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	United Kingdom	All	3000
Europe	All	All	12000
All	All	All	24000

Can't I use ISNULL function instead as shown below

```
SELECT   ISNULL(Continent, 'All') AS Continent,
        ISNULL(Country, 'All') AS Country,
        ISNULL(City, 'All') AS City,
        SUM(SaleAmount) AS TotalSales
FROM Sales

GROUP BY ROLLUP(Continent, Country, City)
```

Well, you can, but only if your data does not contain NULL values. Let me explain what I mean.

At the moment the raw data in our Sales has no NULL values. Let's introduce a NULL value in the City column of the row where Id = 1

```
Update Sales Set City = NULL where Id = 1
```

Now execute the following query with ISNULL function

```
SELECT   ISNULL(Continent, 'All') AS Continent,
        ISNULL(Country, 'All') AS Country,
        ISNULL(City, 'All') AS City,
        SUM(SaleAmount) AS TotalSales
FROM Sales
```

GROUP BY ROLLUP(Continent, Country, City)

Result : Notice that the actuall NULL value in the raw data is also replaced with the word 'All', which is incorrect. Hence the need for Grouping function.

Continent	Country	City	TotalSales
Asia	India	All	1000
Asia	India	Chennai	2000
Asia	India	All	3000
Asia	Japan	Hiroshima	5000
Asia	Japan	Tokyo	4000
Asia	Japan	All	9000
Asia	All	All	12000
Europe	France	Cannes	5000
Europe	France	Paris	4000
Europe	France	All	9000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	United Kingdom	All	3000
Europe	All	All	12000
All	All	All	24000

Please note : Grouping function can be used with Rollup, Cube and Grouping Sets

GROUPING_ID function in SQL Server

Suggested Videos

Part 103 - Cube in SQL Server
Part 104 - Difference between cube and rollup in SQL Server
Part 105 - Grouping function in SQL Server

In this video we will discuss

1. GROUPING_ID function in SQL Server
2. Difference between GROUPING and GROUPING_ID functions
3. Use of GROUPING_ID function

GROUPING_ID function computes the level of grouping.

Difference between GROUPING and GROUPING_ID

Syntax : GROUPING function is used on single column, where as the column list for GROUPING_ID function must match with GROUP BY column list.

GROUPING(Col1)
GROUPING_ID(Col1, Col2, Col3,...)

GROUPING indicates whether the column in a GROUP BY list is aggregated or not. Grouping returns 1 for aggregated or 0 for not aggregated in the result set.

GROUPING_ID() function concatenates all the GOUPING() functions, perform the binary to decimal conversion, and returns the equivalent integer. In short
GROUPING_ID(A, B, C) = GROUPING(A) + GROUPING(B) + GROUPING(C)

Let us understand this with an example.

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
        CAST(GROUPING(Continent) AS NVARCHAR(1)) +
        CAST(GROUPING(Country) AS NVARCHAR(1)) +
        CAST(GROUPING(City) AS NVARCHAR(1)) AS Groupings,
        GROUPING_ID(Continent, Country, City) AS GPID
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
```

Query result :

#	Continent	Country	City	TotalSales	Groupings	GPID
1	Asia	India	Bangalore	1000	000	0
2	Asia	India	Chennai	2000	000	0
3	Asia	India	NULL	3000	001	1
4	Asia	Japan	Hiroshima	5000	000	0
5	Asia	Japan	Tokyo	4000	000	0
6	Asia	Japan	NULL	9000	001	1
7	Asia	NULL	NULL	12000	011	3
8	Europe	France	Cannes	5000	000	0
9	Europe	France	Paris	4000	000	0
10	Europe	France	NULL	9000	001	1
11	Europe	United Kingdom	London	1000	000	0
12	Europe	United Kingdom	Manchester	2000	000	0
13	Europe	United Kingdom	NULL	3000	001	1
14	Europe	NULL	NULL	12000	011	3
15	NULL	NULL	NULL	24000	111	7

Row Number 1 : Since the data is not aggregated by any column GROUPING(Continent), GROUPING(Country) and GROUPING(City) return 0 and as result we get a binar string with all ZEROS (000). When this converted to decimal we get 0 which is displayed in GPID column.

Row Number 7 : The data is aggregated for Country and City columns, so

GROUPING(Country) and GROUPING(City) return 1 where as GROUPING(Continent) return 0. As result we get a binar string (011). When this converted to decimal we get 10 which is displayed in GPID column.

Row Number 15 : This is the Grand total row. Notice in this row the data is aggregated by all the 3 columns. Hence all the 3 GROUPING functions return 1. So we get a binary string with all ONES (111). When this converted to decimal we get 7 which is displayed in GPID column.

Use of GROUPING_ID function : GROUPING_ID function is very handy if you want to sort and filter by level of grouping.

Sorting by level of grouping :

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
        GROUPING_ID(Continent, Country, City) AS GPID
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
ORDER BY GPID
```

Result :

Continent	Country	City	TotalSales	GPID
Asia	Japan	Hiroshima	5000	0
Asia	Japan	Tokyo	4000	0
Asia	India	Bangalore	1000	0
Asia	India	Chennai	2000	0
Europe	France	Cannes	5000	0
Europe	France	Paris	4000	0
Europe	United Kingdom	London	1000	0
Europe	United Kingdom	Manchester	2000	0
Europe	United Kingdom	NULL	3000	1
Europe	France	NULL	9000	1
Asia	India	NULL	3000	1
Asia	Japan	NULL	9000	1
Asia	NULL	NULL	12000	3
Europe	NULL	NULL	12000	3
NULL	NULL	NULL	24000	7

Filter by level of grouping : The following query retrieves only continent level aggregated data

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
        GROUPING_ID(Continent, Country, City) AS GPID
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
```

```
HAVING GROUPING_ID(Continent, Country, City) = 3
```

Result :

Continent	Country	City	TotalSales	GPID
Asia	NULL	NULL	12000	3
Europe	NULL	NULL	12000	3

Debugging sql server stored procedures

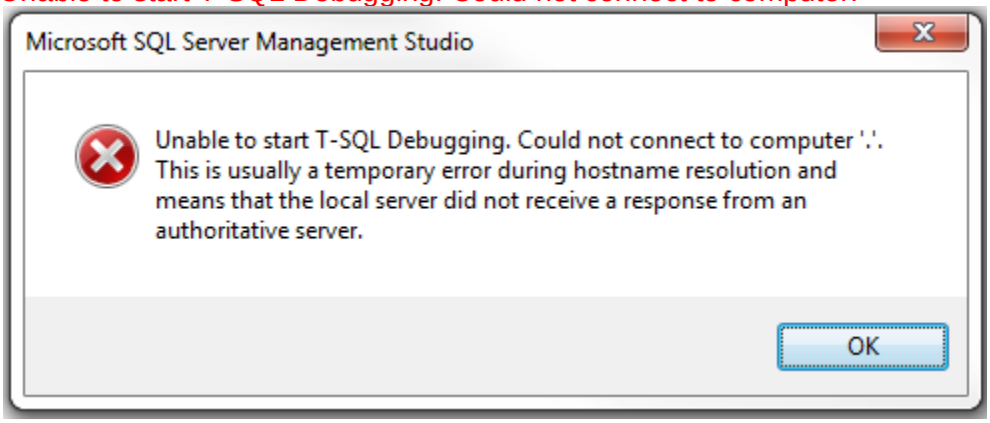
Suggested Videos

Part 104 - Difference between cube and rollup in SQL Server
Part 105 - Grouping function in SQL Server
Part 106 - Grouping Id function in SQL Server

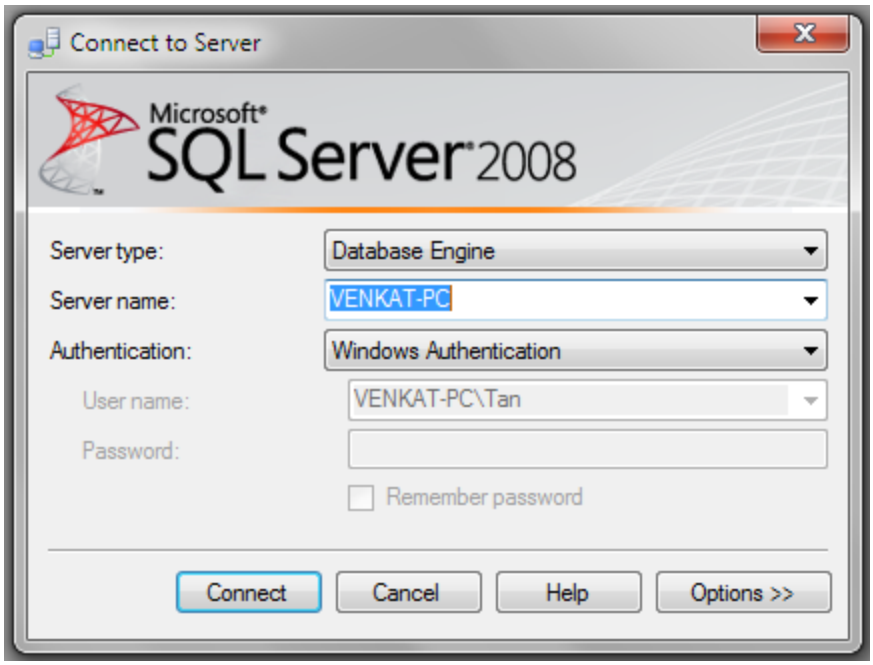
In this video we will discuss **how to debug stored procedures in SQL Server**.

Setting up the Debugger in SSMS : If you have connected to SQL Server using (local) or . (period), and when you start the debugger you will get the following error

Unable to start T-SQL Debugging. Could not connect to computer.



To fix this error, use the computer name to connect to the SQL Server instead of using (local) or .



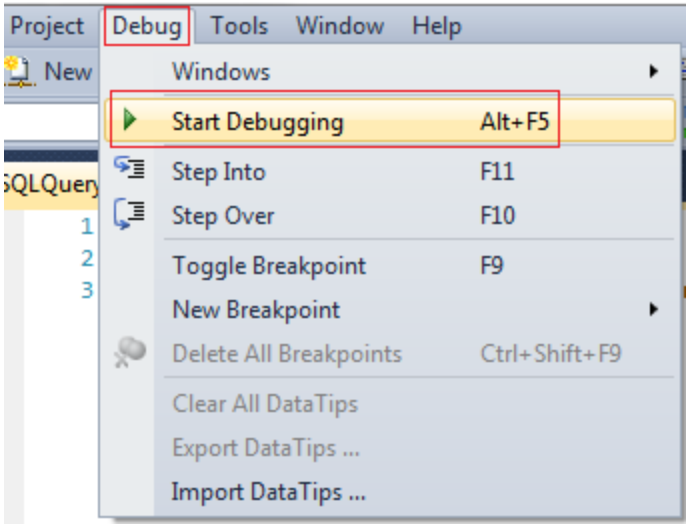
For the examples in this video we will be using the following stored procedure.

```
Create procedure spPrintEvenNumbers
@Target int
as
Begin
    Declare @StartNumber int
    Set @StartNumber = 1

    while(@StartNumber < @Target)
    Begin
        If(@StartNumber%2 = 0)
        Begin
            Print @StartNumber
        End
        Set @StartNumber = @StartNumber + 1
    End
    Print 'Finished printing even numbers till ' + RTRIM(@Target)
End
```

Connect to SQL Server using your computer name, and then execute the above code to create the stored procedure. At this point, open a New Query window. Copy and paste the following T-SQL code to execute the stored procedure.

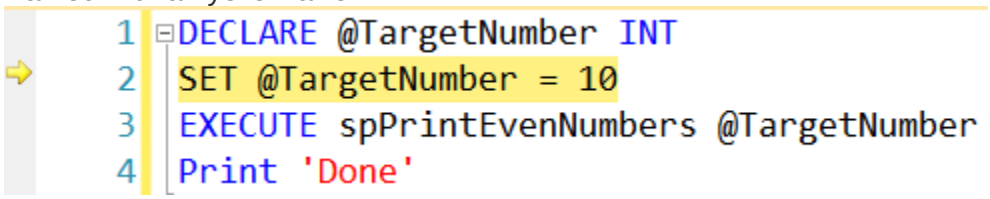
```
DECLARE @TargetNumber INT
SET @TargetNumber = 10
EXECUTE spPrintEvenNumbers @TargetNumber
Print 'Done'
```

Starting the Debugger in SSMS : There are 2 ways to start the debugger
1. In SSMS, click on the **Debug** Menu and select **Start Debugging**

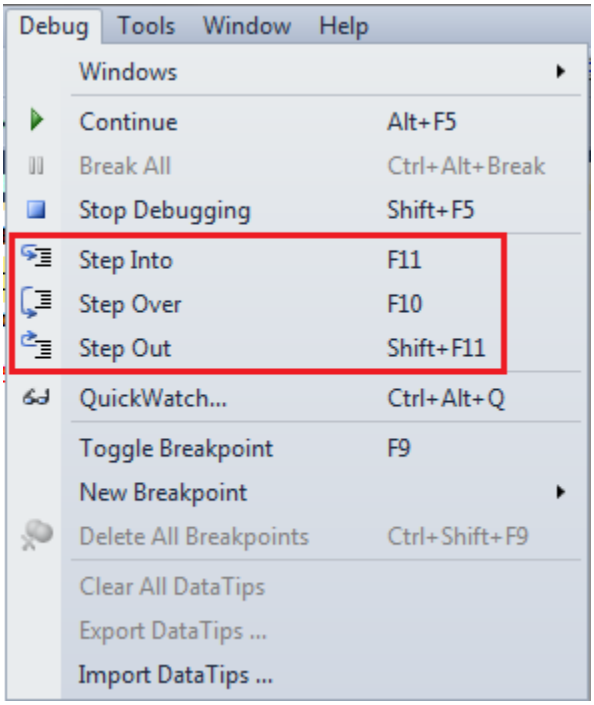


2. Use the keyboard shortcut **ALT + F5**

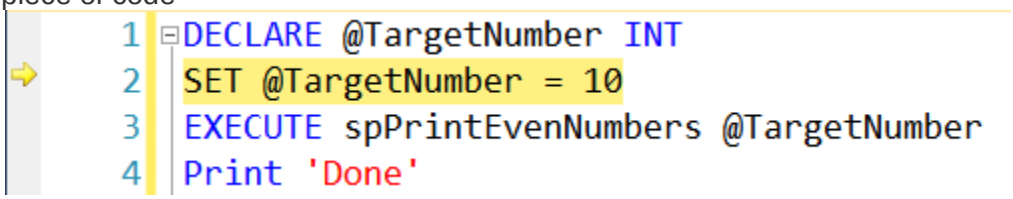
At this point you should have the debugger running. The line that is about to be executed is marked with an yellow arrow



Step Over, Step into and Step Out in SSMS : You can find the keyboard shortcuts in the Debug menu in SSMS.



Let us understand what Step Over, Step into and Step Out does when debugging the following piece of code



1. There is no difference when you STEP INTO (F11) or STEP OVER (F10) the code on LINE 2

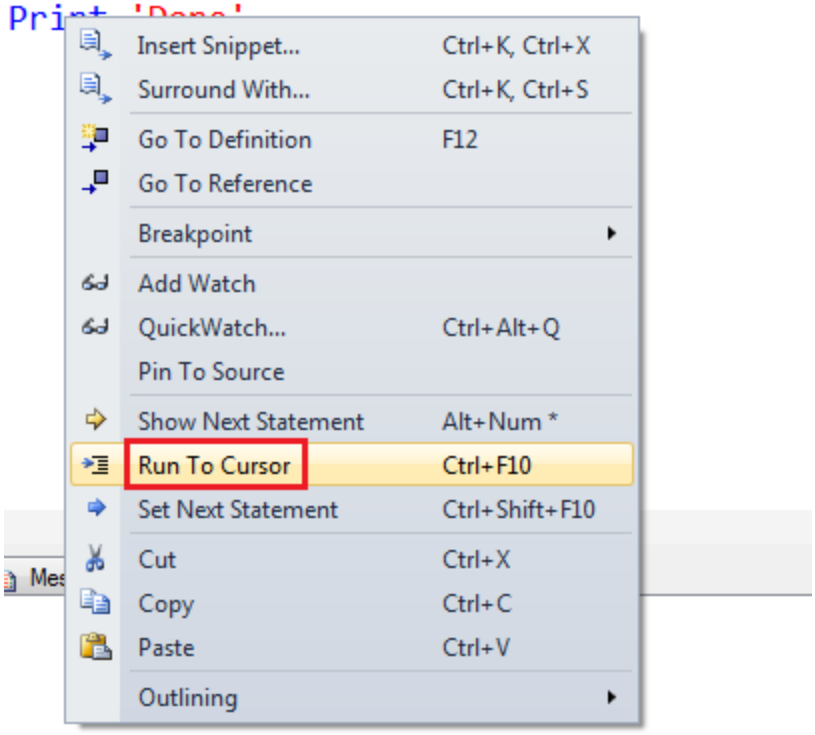
2. On LINE 3, we are calling a Stored Procedure. On this statement if we press F10 (STEP OVER), it won't give us the opportunity to debug the stored procedure code. To be able to debug the stored procedure code you will have to STEP INTO it by pressing F11.

3. If the debugger is in the stored procedure, and you don't want to debug line by line with in that stored procedure, you can STEP OUT of it by pressing SHIFT + F11. When you do this, the debugger completes the execution of the stored procedure and waits on the next line in the main query, i.e on LINE 4 in this example.

To stop debugging : There are 2 ways to stop debugging
1. In SSMS, click on the Debug Menu and select Stop Debugging
2. Use the keyboard shortcut SHIFT + F5

Show Next Statement shows the next statement that the debugger is about to execute.
Run to Cursor command executes all the statements in a batch up to the current cursor position

```
DECLARE @TargetNumber INT
SET @TargetNumber = 10
EXECUTE spPrintEvenNumbers @TargetNumber
Print 'Done'
```

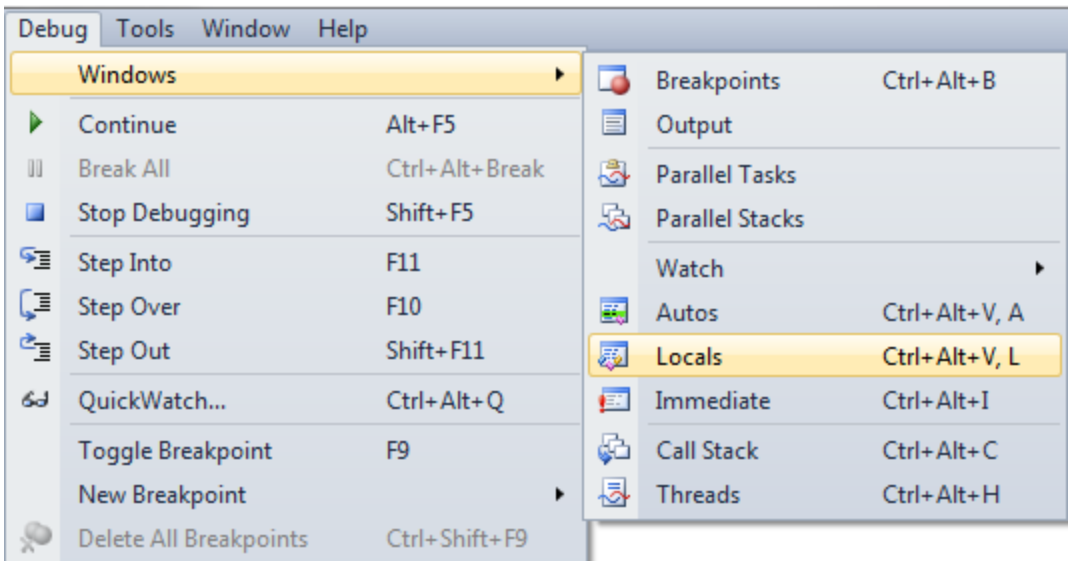


Locals Window in SSMS : Displays the current values of variables and parameters

Locals

Name	Value	Type
@Target	10	int
@StartNumber	1	int

If you cannot see the locals window or if you have closed it and if you want to open it, you can do so using the following menu option. Locals window is only available if you are in DEBUG mode.



Watch Window in SSMS : Just like Locals window, Watch window is used to watch the values of variables. You can add and remove variables from the watch window. To add a variable to the Watch Window, right click on the variable and select "Add Watch" option from the context menu.

Watch 1

Name	Value	Type
@StartNumber	2	int

Call Stack Window in SSMS : Allows you to navigate up and down the call stack to see what values your application is storing at different levels. It's an invaluable tool for determining why your code is doing what it's doing.

Call Stack

Name	Language
spPrintEvenNumbers(PC-LON-1124.SampleDB)(int @Target=10) Line 6	Transact-SQL
SQLQuery1.sql() Line 3	Transact-SQL

Immediate Window in SSMS : Very helpful during debugging to evaluate expressions, and print variable values. To clear immediate window type >cls and press enter.

Immediate Window

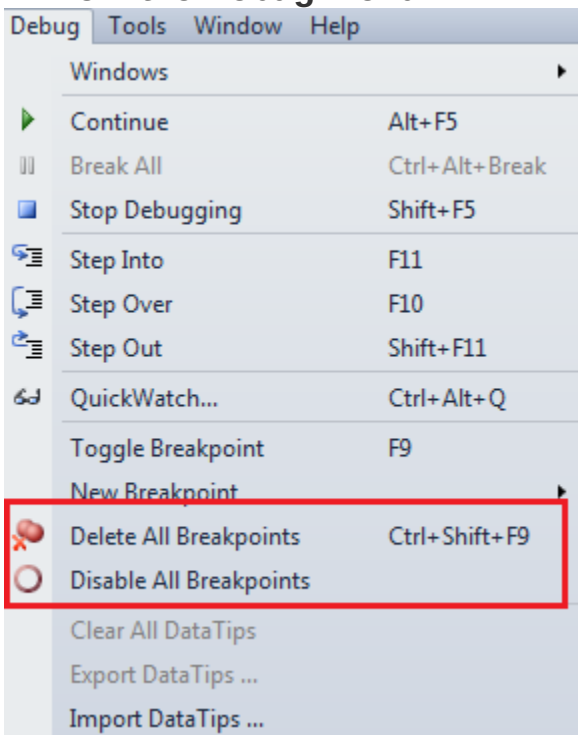
```
@StartNumber
4
@StartNumber * 50
200
```

Breakpoints in SSMS : There are 2 ways to set a breakpoint in SSMS.

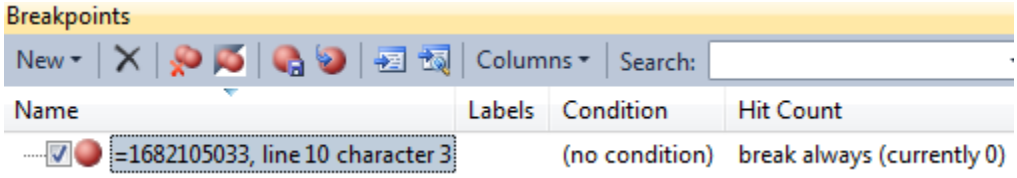
1. By clicking on the grey margin on the left hand side in SSMS (to remove click again)
2. By pressing F9 (to remove press F9 again)

Enable, Disable or Delete all breakpoints : There are 2 ways to Enable, Disable or Delete all breakpoints

1. From the Debug menu



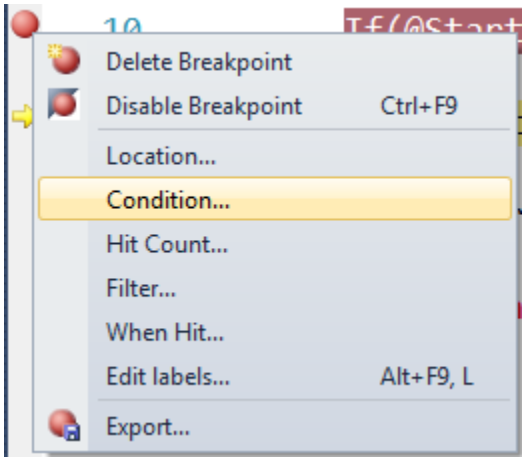
2. From the Breakpoints window. To view Breakpoints window select Debug => Windows => Breakpoints or use the keyboard shortcut ALT + CTRL + B



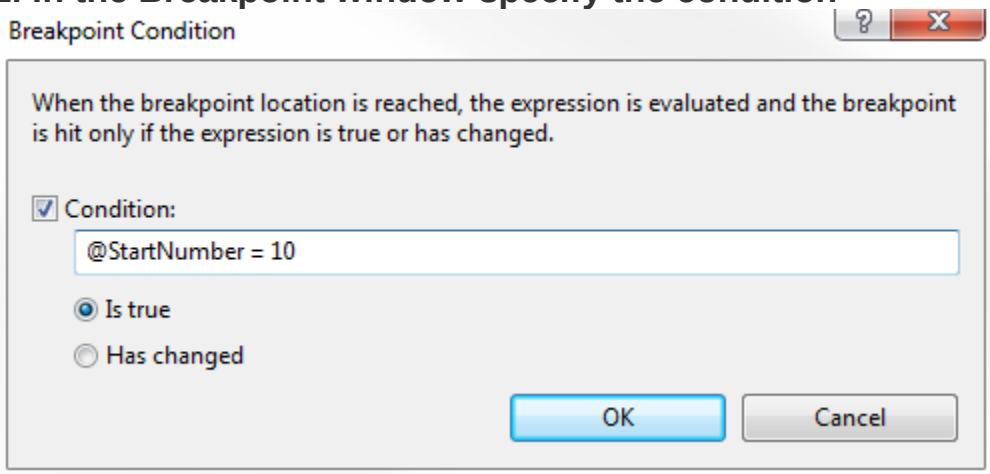
Conditional Breakpoint : Conditional Breakpoints are hit only when the specified condition is met. These are extremely useful when you have some kind of a loop and you want to break, only when the loop variable has a specific value (For example loop varible = 100).

How to set a conditional break point in SSMS :

1. Right click on the Breakpoint and select Condition from the context menu



2. In the Breakpoint window specify the condition



Over clause in SQL Server

Suggested Videos

Part 105 - Grouping function in SQL Server
Part 106 - Grouping_Id function in SQL Server
Part 107 - Debugging sql server stored procedures

In this video we will discuss the power and use of Over clause in SQL Server.
The **OVER** clause combined with **PARTITION BY** is used to break up data into partitions.
Syntax : function (...) OVER (PARTITION BY col1, Col2, ...)

The specified function operates for each partition.

For example :
COUNT(Gender) OVER (PARTITION BY Gender) will partition the data by **GENDER** i.e there will 2 partitions (Male and Female) and then the COUNT() function is applied over each partition.

Any of the following functions can be used. Please note this is not the complete list.
COUNT(), AVG(), SUM(), MIN(), MAX(), ROW_NUMBER(), RANK(), DENSE_RANK() etc.

Example : We will use the following **Employees table** for the examples in this video.

Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

SQl Script to create Employees table

```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go

Insert Into Employees Values (1, 'Mark', 'Male', 5000)
Insert Into Employees Values (2, 'John', 'Male', 4500)
Insert Into Employees Values (3, 'Pam', 'Female', 5500)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 3500)
Insert Into Employees Values (6, 'Mary', 'Female', 5000)
Insert Into Employees Values (7, 'Ben', 'Male', 6500)
Insert Into Employees Values (8, 'Jodi', 'Female', 7000)
Insert Into Employees Values (9, 'Tom', 'Male', 5500)
Insert Into Employees Values (10, 'Ron', 'Male', 5000)
Go
```

Write a query to retrieve total count of employees by Gender. Also in the result we want Average, Minimum and Maximum salary by Gender. The result of the query should be as shown below.

Gender	GenderTotal	AvgSal	MinSal	MaxSal
Female	4	5375	4000	7000
Male	6	5000	3500	6500

This can be very easily achieved using a simple **GROUP BY** query as show below.

```
SELECT Gender, COUNT(*) AS GenderTotal, AVG(Salary) AS AvgSal,
       MIN(Salary) AS MinSal, MAX(Salary) AS MaxSal
FROM Employees
GROUP BY Gender
```

What if we want **non-aggregated values** (like employee Name and Salary) in result set along with aggregated values

Name	Salary	Gender	GenderTotals	AvgSal	MinSal	MaxSal
Pam	5500	Female	4	5375	4000	7000
Sara	4000	Female	4	5375	4000	7000
Mary	5000	Female	4	5375	4000	7000
Jodi	7000	Female	4	5375	4000	7000
Tom	5500	Male	6	5000	3500	6500
Ron	5000	Male	6	5000	3500	6500
Ben	6500	Male	6	5000	3500	6500
Todd	3500	Male	6	5000	3500	6500
Mark	5000	Male	6	5000	3500	6500
John	4500	Male	6	5000	3500	6500

You cannot include **non-aggregated** columns in the **GROUP BY** query.

```
SELECT Name, Salary, Gender, COUNT(*) AS GenderTotal, AVG(Salary) AS AvgSal,
       MIN(Salary) AS MinSal, MAX(Salary) AS MaxSal
FROM Employees
GROUP BY Gender
```

The above query will result in the following error

Column 'Employees.Name' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause

One way to achieve this is by including the aggregations in a subquery and then **JOINING** it with the main query as shown in the example below. Look at the amount of T-SQL code we have to write.

```
SELECT Name, Salary, Employees.Gender, Genders.GenderTotals,
       Genders.AvgSal, Genders.MinSal, Genders.MaxSal
FROM Employees
INNER JOIN
(SELECT Gender, COUNT(*) AS GenderTotals,
        AVG(Salary) AS AvgSal,
       MIN(Salary) AS MinSal, MAX(Salary) AS MaxSal
FROM Employees
GROUP BY Gender) AS Genders
ON Genders.Gender = Employees.Gender
```

Better way of doing this is by using the **OVER** clause combined with **PARTITION BY**

```
SELECT Name, Salary, Gender,
    COUNT(Gender) OVER(PARTITION BY Gender) AS GenderTotals,
    AVG(Salary) OVER(PARTITION BY Gender) AS AvgSal,
    MIN(Salary) OVER(PARTITION BY Gender) AS MinSal,
    MAX(Salary) OVER(PARTITION BY Gender) AS MaxSal
FROM Employees
```

Row_Number function in SQL Server

Suggested Videos

Part 106 - Grouping_Id function in SQL Server
Part 107 - Debugging sql server stored procedures
Part 108 - Over clause in SQL Server

In this video we will discuss Row_Number function in SQL Server. This is continuation to Part 108. Please watch Part 108 from SQL Server tutorial before proceeding.

Row_Number function

- Introduced in SQL Server 2005
- Returns the sequential number of a row starting at 1
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, row number is reset to 1 when the partition changes

Syntax : `ROW_NUMBER() OVER (ORDER BY Col1, Col2)`

Row_Number function without PARTITION BY : In this example, data is not partitioned, so ROW_NUMBER will provide a consecutive numbering for all the rows in the table based on the order of rows imposed by the ORDER BY clause.

```
SELECT Name, Gender, Salary,
    ROW_NUMBER() OVER (ORDER BY Gender) AS RowNumber
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

Name	Gender	Salary	RowNumber
Pam	Female	5500	1
Sara	Female	4000	2
Mary	Female	5000	3
Jodi	Female	7000	4
Tom	Male	5500	5
Ron	Male	5000	6
Ben	Male	6500	7
Todd	Male	3500	8
Mark	Male	5000	9
John	Male	4500	10

Please note : If ORDER BY clause is not specified you will get the following error
The function 'ROW_NUMBER' must have an OVER clause with ORDER BY

Row_Number function with PARTITION BY : In this example, data is partitioned by Gender, so ROW_NUMBER will provide a consecutive numbering only for the rows with in a parttion. When the partition changes the row number is reset to 1.

```
SELECT Name, Gender, Salary,
      ROW_NUMBER() OVER (PARTITION BY Gender ORDER BY Gender) AS RowNumber
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

Name	Gender	Salary	RowNumber
Pam	Female	5500	1
Sara	Female	4000	2
Mary	Female	5000	3
Jodi	Female	7000	4
Tom	Male	5500	1
Ron	Male	5000	2
Ben	Male	6500	3
Todd	Male	3500	4
Mark	Male	5000	5
John	Male	4500	6

Use case for Row_Number function : Deleting all duplicate rows except one from a sql server table.

Discussed in detail in Part 4 of SQL Server Interview Questions and Answers video series.

Rank and Dense_Rank in SQL Server

Suggested Videos
Part 107 - Debugging sql server stored procedures
Part 108 - Over clause in SQL Server
Part 109 - Row Number function in SQL Server
In this video we will discuss **Rank and Dense_Rank functions in SQL Server**
Rank and Dense_Rank functions

- Introduced in SQL Server 2005
- Returns a rank starting at 1 based on the ordering of rows imposed by the ORDER BY clause
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, rank is reset to 1 when the partition changes

Difference between Rank and Dense_Rank functions
Rank function skips ranking(s) if there is a tie where as Dense_Rank will not.

For example : If you have 2 rows at rank 1 and you have 5 rows in total.
RANK() returns - 1, 1, 3, 4, 5
DENSE_RANK returns - 1, 1, 2, 3, 4

Syntax :
RANK() OVER (ORDER BY Col1, Col2, ...)
DENSE_RANK() OVER (ORDER BY Col1, Col2, ...)

Example : We will use the following **Employees** table for the examples in this video

Id	Name	Gender	Salary
1	Mark	Male	8000
2	John	Male	8000
3	Pam	Female	5000
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	6000
7	Ben	Male	6500
8	Jodi	Female	4500
9	Tom	Male	7000
10	Ron	Male	6800

SQl Script to create Employees table
```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go

Insert Into Employees Values (1, 'Mark', 'Male', 8000)
Insert Into Employees Values (2, 'John', 'Male', 8000)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 3500)
Insert Into Employees Values (6, 'Mary', 'Female', 6000)
Insert Into Employees Values (7, 'Ben', 'Male', 6500)
Insert Into Employees Values (8, 'Jodi', 'Female', 4500)
Insert Into Employees Values (9, 'Tom', 'Male', 7000)
Insert Into Employees Values (10, 'Ron', 'Male', 6800)
Go
```

RANK() and DENSE_RANK() functions without PARTITION BY clause : In this example, data is not partitioned, so RANK() function provides a consecutive numbering except when there is a tie. Rank 2 is skipped as there are 2 rows at rank 1. The third row gets rank 3.

DENSE_RANK() on the other hand will not skip ranks if there is a tie. The first 2 rows get rank 1. Third row gets rank 2.

```
SELECT Name, Salary, Gender,
RANK() OVER (ORDER BY Salary DESC) AS [Rank],
DENSE_RANK() OVER (ORDER BY Salary DESC) AS DenseRank
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	8000
2	John	Male	8000
3	Pam	Female	5000
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	6000
7	Ben	Male	6500
8	Jodi	Female	4500
9	Tom	Male	7000
10	Ron	Male	6800

Name	Salary	Gender	Rank	DenseRank
Mark	8000	Male	1	1
John	8000	Male	1	1
Tom	7000	Male	3	2
Ron	6800	Male	4	3
Ben	6500	Male	5	4
Mary	6000	Female	6	5
Pam	5000	Female	7	6
Jodi	4500	Female	8	7
Sara	4000	Female	9	8
Todd	3500	Male	10	9

RANK() and DENSE_RANK() functions with PARTITION BY clause : Notice when the partition changes from Female to Male Rank is reset to 1

```
SELECT Name, Salary, Gender,
RANK() OVER (PARTITION BY Gender ORDER BY Salary DESC) AS [Rank],
DENSE_RANK() OVER (PARTITION BY Gender ORDER BY Salary DESC)
AS DenseRank
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	8000
2	John	Male	8000
3	Pam	Female	5000
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	6000
7	Ben	Male	6500
8	Jodi	Female	4500
9	Tom	Male	7000
10	Ron	Male	6800

Name	Salary	Gender	Rank	DenseRank
Mary	6000	Female	1	1
Pam	5000	Female	2	2
Jodi	4500	Female	3	3
Sara	4000	Female	4	4
Mark	8000	Male	1	1
John	8000	Male	1	1
Tom	7000	Male	3	2
Ron	6800	Male	4	3
Ben	6500	Male	5	4
Todd	3500	Male	6	5

Use case for RANK and DENSE_RANK functions : Both these functions can be used to find Nth highest salary. However, which function to use depends on what you want to do when there is a tie. Let me explain with an example.

If there are 2 employees with the FIRST highest salary, there are 2 different business cases

- If your business case is, not to produce any result for the SECOND highest salary, then use RANK function
- If your business case is to return the next Salary after the tied rows as the SECOND highest Salary, then use DENSE_RANK function

Since we have 2 Employees with the FIRST highest salary. Rank() function will not return any rows for the SECOND highest Salary.

```
WITH Result AS
(
   SELECT Salary, RANK() OVER (ORDER BY Salary DESC) AS Salary_Rank
   FROM Employees
)
SELECT TOP 1 Salary FROM Result WHERE Salary_Rank = 2
```

Though we have 2 Employees with the FIRST highest salary. Dense_Rank() function returns, the next Salary after the tied rows as the SECOND highest Salary

```
WITH Result AS
(
   SELECT Salary, DENSE_RANK() OVER (ORDER BY Salary DESC) AS Salary_Rank
   FROM Employees
)
SELECT TOP 1 Salary FROM Result WHERE Salary_Rank = 2
```

You can also use RANK and DENSE_RANK functions to find the Nth highest Salary among Male or Female employee groups. The following query finds the 3rd highest salary amount paid among the Female employees group

```
WITH Result AS
(
   SELECT Salary, Gender,
        DENSE_RANK() OVER (PARTITION BY Gender ORDER BY Salary DESC)
        AS Salary_Rank
   FROM Employees
)
SELECT TOP 1 Salary FROM Result WHERE Salary_Rank = 3
AND Gender = 'Female'
```

Difference between rank dense_rank and row_number in SQL

Suggested Videos

Part 108 - Over clause in SQL Server
Part 109 - Row_Number function in SQL Server
Part 110 - Rank and Dense_Rank in SQL Server

In this video we will discuss the similarities and **difference between RANK, DENSE_RANK and ROW_NUMBER** functions in SQL Server.

Similarities between RANK, DENSE_RANK and ROW_NUMBER functions

- Returns an increasing integer value starting at 1 based on the ordering of rows imposed by the ORDER BY clause (if there are no ties)
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, the integer value is reset to 1 when the partition changes

We will use the following **Employees** table for the examples in this video

Id	Name	Gender	Salary
1	Mark	Male	6000
2	John	Male	8000
3	Pam	Female	4000
4	Sara	Female	5000
5	Todd	Male	3000

SQL Script to create the Employees table

```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go
```

```
Insert Into Employees Values (1, 'Mark', 'Male', 6000)
Insert Into Employees Values (2, 'John', 'Male', 8000)
Insert Into Employees Values (3, 'Pam', 'Female', 4000)
Insert Into Employees Values (4, 'Sara', 'Female', 5000)
Insert Into Employees Values (5, 'Todd', 'Male', 3000)
```

Notice that no two employees in the table have the same salary. So all the 3 functions RANK, DENSE_RANK and ROW_NUMBER produce the same increasing integer value when ordered by Salary column.

```
SELECT Name, Salary, Gender,
ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNumber,
RANK() OVER (ORDER BY Salary DESC) AS [Rank],
DENSE_RANK() OVER (ORDER BY Salary DESC) AS DenseRank
FROM Employees
```

Name	Salary	Gender	RowNumber	Rank	DenseRank
John	8000	Male	1	1	1
Mark	6000	Male	2	2	2
Sara	5000	Female	3	3	3
Pam	4000	Female	4	4	4
Todd	3000	Male	5	5	5

You will only see the difference when there ties (duplicate values in the column used in the ORDER BY clause).

Now let's include duplicate values for Salary column.

To do this
First delete existing data from the Employees table

```
DELETE FROM Employees
```

Insert new rows with duplicate valuse for Salary column

```
Insert Into Employees Values (1, 'Mark', 'Male', 8000)
Insert Into Employees Values (2, 'John', 'Male', 8000)
Insert Into Employees Values (3, 'Pam', 'Female', 8000)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 3500)
```

At this point data in the Employees table should be as shown below

Id	Name	Gender	Salary
1	Mark	Male	8000
2	John	Male	8000
3	Pam	Female	8000
4	Sara	Female	4000
5	Todd	Male	3500

Notice 3 employees have the same salary 8000. When you execute the following query you can clearly see the difference between RANK, DENSE_RANK and ROW_NUMBER functions.

```
SELECT Name, Salary, Gender,
ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNumber,
RANK() OVER (ORDER BY Salary DESC) AS [Rank],
DENSE_RANK() OVER (ORDER BY Salary DESC) AS DenseRank
FROM Employees
```

Name	Salary	Gender	RowNumber	Rank	DenseRank
Mark	8000	Male	1	1	1
John	8000	Male	2	1	1
Pam	8000	Female	3	1	1
Sara	4000	Female	4	4	2
Todd	3500	Male	5	5	3

Difference between RANK, DENSE_RANK and ROW_NUMBER functions

- **ROW_NUMBER :** Returns an increasing unique number for each row starting at 1, even if there are duplicates.
- **RANK :** Returns an increasing unique number for each row starting at 1. When there are duplicates, same rank is assigned to all the duplicate rows, but the next row after the duplicate rows will have the rank it would have been assigned if there had been no duplicates. So RANK function skips rankings if there are duplicates.
- **DENSE_RANK :** Returns an increasing unique number for each row starting at 1. When there are duplicates, same rank is assigned to all the duplicate rows but the DENSE_RANK function will not skip any ranks. This means the next row after the duplicate rows will have the next rank in the sequence.

- ## Calculate running total in SQL Server 2012
- **Suggested Videos**
  Part 109 - Row_Number function in SQL Server
  Part 110 - Rank and Dense_Rank in SQL Server
  Part 111 - Difference between rank dense_rank and row_number in SQL

In this video we will discuss how to calculate running total in SQL Server 2012 and later versions.

We will use the following **Employees table** for the examples in this video.

Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

SQL Script to create Employees table

```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go

Insert Into Employees Values (1, 'Mark', 'Male', 5000)
Insert Into Employees Values (2, 'John', 'Male', 4500)
Insert Into Employees Values (3, 'Pam', 'Female', 5500)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 3500)
Insert Into Employees Values (6, 'Mary', 'Female', 5000)
Insert Into Employees Values (7, 'Ben', 'Male', 6500)
Insert Into Employees Values (8, 'Jodi', 'Female', 7000)
Insert Into Employees Values (9, 'Tom', 'Male', 5500)
Insert Into Employees Values (10, 'Ron', 'Male', 5000)
Go
```

SQL Query to compute running total without partitions

```
SELECT Name, Gender, Salary,
      SUM(Salary) OVER (ORDER BY ID) AS RunningTotal
FROM Employees
```

Name	Gender	Salary	RunningTotal
Mark	Male	5000	5000
John	Male	4500	9500
Pam	Female	5500	15000
Sara	Female	4000	19000
Todd	Male	3500	22500
Mary	Female	5000	27500
Ben	Male	6500	34000
Jodi	Female	7000	41000
Tom	Male	5500	46500
Ron	Male	5000	51500

SQL Query to compute running total with partitions

```
SELECT Name, Gender, Salary,
	SUM(Salary) OVER (PARTITION BY Gender ORDER BY ID) AS RunningTotal
FROM Employees
```

Name	Gender	Salary	RunningTotal
Pam	Female	5500	5500
Sara	Female	4000	9500
Mary	Female	5000	14500
Jodi	Female	7000	21500
Mark	Male	5000	5000
John	Male	4500	9500
Todd	Male	3500	13000
Ben	Male	6500	19500
Tom	Male	5500	25000
Ron	Male	5000	30000

What happens if I use order by on Salary column

If you have duplicate values in the Salary column, all the duplicate values will be added to the running total at once. In the example below notice that we have 5000 repeated 3 times. So 15000 (i.e 5000 + 5000 + 5000) is added to the running total at once.

```
SELECT Name, Gender, Salary,
	SUM(Salary) OVER (ORDER BY Salary) AS RunningTotal
FROM Employees
```

Name	Gender	Salary	RunningTotal
Todd	Male	3500	3500
Sara	Female	4000	7500
John	Male	4500	12000
Mark	Male	5000	27000
Mary	Female	5000	27000
Ron	Male	5000	27000
Tom	Male	5500	38000
Pam	Female	5500	38000
Ben	Male	6500	44500
Jodi	Female	7000	51500

So when computing running total, it is better to use a column that has unique data in the ORDER BY clause.

NTILE function in SQL Server

Suggested Videos

Part 110 - Rank and Dense_Rank in SQL Server
Part 111 - Difference between rank dense_rank and row_number in SQL
Part 112 - Calculate running total in SQL Server 2012

In this video we will discuss **NTILE function in SQL Server**

Lead and Lag functions

- Introduced in SQL Server 2012
- Lead function is used to access subsequent row data along with current row data
- Lag function is used to access previous row data along with current row data
- ORDER BY clause is required
- PARTITION BY clause is optional

Syntax

```
LEAD(Column_Name, Offset, Default_Value) OVER (ORDER BY Col1, Col2, ...)
LAG(Column_Name, Offset, Default_Value) OVER (ORDER BY Col1, Col2, ...)
```

- **Offset -** Number of rows to lead or lag.
- **Default_Value -** The default value to return if the number of rows to lead or lag goes beyond first row or last row in a table or partition. If default value is not specified NULL is returned.

We will use the following **Employees table** for the examples in this video

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

SQL Script to create the Employees table

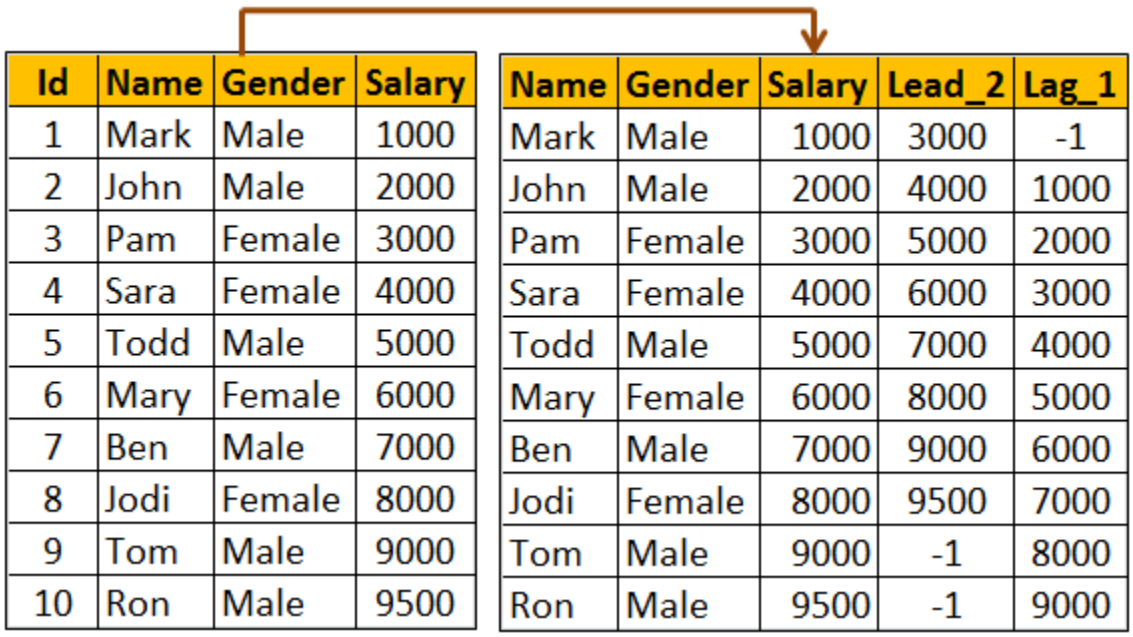
```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go

Insert Into Employees Values (1, 'Mark', 'Male', 1000)
Insert Into Employees Values (2, 'John', 'Male', 2000)
Insert Into Employees Values (3, 'Pam', 'Female', 3000)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 5000)
Insert Into Employees Values (6, 'Mary', 'Female', 6000)
Insert Into Employees Values (7, 'Ben', 'Male', 7000)
Insert Into Employees Values (8, 'Jodi', 'Female', 8000)
Insert Into Employees Values (9, 'Tom', 'Male', 9000)
Insert Into Employees Values (10, 'Ron', 'Male', 9500)
Go
```

Lead and Lag functions example WITHOUT partitions : This example Leads 2 rows and Lags 1 row from the current row.

- When you are on the first row, LEAD(Salary, 2, -1) allows you to move forward 2 rows and retrieve the salary from the 3rd row.
- When you are on the first row, LAG(Salary, 1, -1) allows us to move backward 1 row. Since there no rows beyond row 1, Lag function in this case returns the default value -1.

- When you are on the last row, LEAD(Salary, 2, -1) allows you to move forward 2 rows. Since there no rows beyond the last row 1, Lead function in this case returns the default value -1.
- When you are on the last row, LAG(Salary, 1, -1) allows us to move backward 1 row and retrieve the salary from the previous row.

```
SELECT Name, Gender, Salary,
    LEAD(Salary, 2, -1) OVER (ORDER BY Salary) AS Lead_2,
    LAG(Salary, 1, -1) OVER (ORDER BY Salary) AS Lag_1
FROM Employees
```



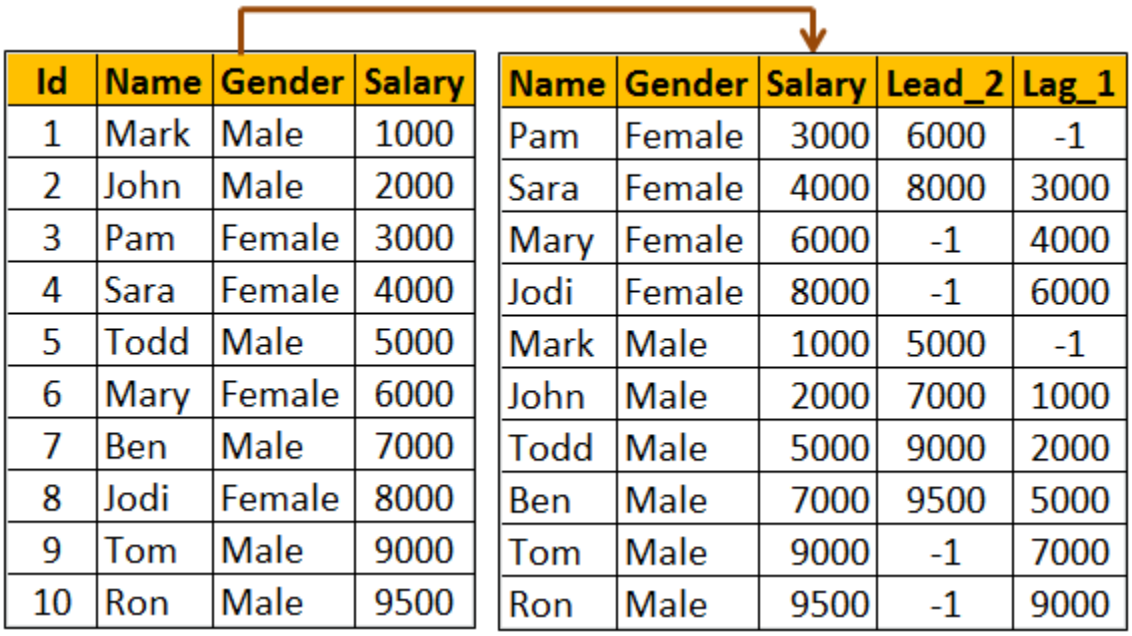
Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Lead_2	Lag_1
Mark	Male	1000	3000	-1
John	Male	2000	4000	1000
Pam	Female	3000	5000	2000
Sara	Female	4000	6000	3000
Todd	Male	5000	7000	4000
Mary	Female	6000	8000	5000
Ben	Male	7000	9000	6000
Jodi	Female	8000	9500	7000
Tom	Male	9000	-1	8000
Ron	Male	9500	-1	9000

Lead and Lag functions example WITH partitions : Notice that in this example, Lead and Lag functions return default value if the number of rows to lead or lag goes beyond first row or last row in the partition.

```
SELECT Name, Gender, Salary,
    LEAD(Salary, 2, -1) OVER (PARTITION By Gender ORDER BY Salary) AS Lead_2,
    LAG(Salary, 1, -1) OVER (PARTITION By Gender ORDER BY Salary) AS Lag_1
```

```
FROM Employees
```



Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Lead_2	Lag_1
Pam	Female	3000	6000	-1
Sara	Female	4000	8000	3000
Mary	Female	6000	-1	4000
Jodi	Female	8000	-1	6000
Mark	Male	1000	5000	-1
John	Male	2000	7000	1000
Todd	Male	5000	9000	2000
Ben	Male	7000	9500	5000
Tom	Male	9000	-1	7000
Ron	Male	9500	-1	9000

FIRST_VALUE function in SQL Server

Suggested Videos

Part 112 - Calculate running total in SQL Server 2012
Part 113 - NTILE function in SQL Server
Part 114 - Lead and Lag functions in SQL Server 2012

In this video we will discuss FIRST_VALUE function in SQL Server

FIRST_VALUE function

- Introduced in SQL Server 2012
- Retrieves the first value from the specified column
- ORDER BY clause is required
- PARTITION BY clause is optional

Syntax

```
:FIRST_VALUE(Column_Name) OVER (ORDER BY Col1, Col2, ...)
```

FIRST_VALUE function example WITHOUT partitions : In the following example, FIRST_VALUE function returns the name of the lowest paid employee from the entire table.

```
SELECT Name, Gender, Salary,
FIRST_VALUE(Name) OVER (ORDER BY Salary) AS FirstValue
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	FirstValue
Mark	Male	1000	Mark
John	Male	2000	Mark
Pam	Female	3000	Mark
Sara	Female	4000	Mark
Todd	Male	5000	Mark
Mary	Female	6000	Mark
Ben	Male	7000	Mark
Jodi	Female	8000	Mark
Tom	Male	9000	Mark
Ron	Male	9500	Mark

FIRST_VALUE function example WITH partitions : In the following example, FIRST_VALUE function returns the name of the lowest paid employee from the respective partition.

```
SELECT Name, Gender, Salary,
FIRST_VALUE(Name) OVER (PARTITION BY Gender ORDER BY Salary) AS FirstValue
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	FirstValue
Pam	Female	3000	Pam
Sara	Female	4000	Pam
Mary	Female	6000	Pam
Jodi	Female	8000	Pam
Mark	Male	1000	Mark
John	Male	2000	Mark
Todd	Male	5000	Mark
Ben	Male	7000	Mark
Tom	Male	9000	Mark
Ron	Male	9500	Mark

Window functions in SQL Server

Suggested Videos

Part 113 - NTILE function in SQL Server
Part 114 - Lead and Lag functions in SQL Server 2012
Part 115 - FIRST_VALUE function in SQL Server

In this video we will discuss **window functions in SQL Server**

In SQL Server we have different categories of window functions

- **Aggregate functions -** AVG, SUM, COUNT, MIN, MAX etc..
- **Ranking functions -** RANK, DENSE_RANK, ROW_NUMBER etc..
- **Analytic functions -** LEAD, LAG, FIRST_VALUE, LAST_VALUE etc...

OVER Clause defines the partitioning and ordering of a rows (i.e a window) for the above functions to operate on. Hence these functions are called window functions. The OVER clause accepts the following three arguments to define a window for these functions to operate on.

- **ORDER BY :** Defines the logical order of the rows
- **PARTITION BY :** Divides the query result set into partitions. The window function is applied to each partition separately.
- **ROWSor RANGE clause :** Further limits the rows within the partition by specifying start and end points within the partition.

The default for **ROWS** or **RANGE** clause is
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

Let us understand the use of **ROWS** or **RANGE** clause with an example.

Compute average salary and display it against every employee row as shown below.

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Average
Mark	Male	1000	5450
John	Male	2000	5450
Pam	Female	3000	5450
Sara	Female	4000	5450
Todd	Male	5000	5450
Mary	Female	6000	5450
Ben	Male	7000	5450
Jodi	Female	8000	5450
Tom	Male	9000	5450
Ron	Male	9500	5450

We might think the following query would do the job.

```
SELECT Name, Gender, Salary,
      AVG(Salary) OVER(ORDER BY Salary) AS Average
FROM Employees
```

As you can see from the result below, the above query does not produce the overall salary average. It produces the average of the current row and the rows preceeding the current row. This is because, the default value of ROWS or RANGE clause (RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) is applied.

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Average
Mark	Male	1000	1000
John	Male	2000	1500
Pam	Female	3000	2000
Sara	Female	4000	2500
Todd	Male	5000	3000
Mary	Female	6000	3500
Ben	Male	7000	4000
Jodi	Female	8000	4500
Tom	Male	9000	5000
Ron	Male	9500	5450

To fix this, provide an explicit value for ROWS or RANGE clause as shown below. ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING tells the window function to operate on the set of rows starting from the first row in the partition to the last row in the partition.

```
SELECT Name, Gender, Salary,
      AVG(Salary) OVER(ORDER BY Salary ROWS BETWEEN
      UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS Average
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Average
Mark	Male	1000	5450
John	Male	2000	5450
Pam	Female	3000	5450
Sara	Female	4000	5450
Todd	Male	5000	5450
Mary	Female	6000	5450
Ben	Male	7000	5450
Jodi	Female	8000	5450
Tom	Male	9000	5450
Ron	Male	9500	5450

The same result can also be achieved by using RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

What is the difference between ROWS and RANGE

We will discuss this in a later video

The following query can be used if you want to compute the average salary of
1. The current row
2. One row PRECEDING the current row and
3. One row FOLLOWING the current row

```
SELECT Name, Gender, Salary,
    AVG(Salary) OVER(ORDER BY Salary
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS Average
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Average
Mark	Male	1000	1500
John	Male	2000	2000
Pam	Female	3000	3000
Sara	Female	4000	4000
Todd	Male	5000	5000
Mary	Female	6000	6000
Ben	Male	7000	7000
Jodi	Female	8000	8000
Tom	Male	9000	8833
Ron	Male	9500	9250

Difference between rows and range

Suggested Videos

Part 114 - Lead and Lag functions in SQL Server 2012
Part 115 - FIRST_VALUE function in SQL Server
Part 116 - Window functions in SQL Server

In this video we will discuss the **difference between rows and range in SQL Server**. This is continuation to Part 116. Please watch Part 116 from SQL Server tutorial before proceeding.

Let us understand the difference with an example. We will use the following **Employees** table in this demo.

Id	Name	Salary
1	Mark	1000
2	John	2000
3	Pam	3000
4	Sara	4000
5	Todd	3000

SQL Script to create the Employees table

```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Salary int
)
Go

Insert Into Employees Values (1, 'Mark', 1000)
Insert Into Employees Values (2, 'John', 2000)
Insert Into Employees Values (3, 'Pam', 3000)
Insert Into Employees Values (4, 'Sara', 4000)
Insert Into Employees Values (5, 'Todd', 5000)
Go
```

Calculate the running total of Salary and display it against every employee row

Id	Name	Salary
1	Mark	1000
2	John	2000
3	Pam	3000
4	Sara	4000
5	Todd	3000

→

Name	Salary	RunningTotal
Mark	1000	1000
John	2000	3000
Pam	3000	6000
Sara	4000	10000
Todd	5000	15000

The following query calculates the running total. We have not specified an explicit value for ROWS or RANGE clause.

```
SELECT Name, Salary,
      SUM(Salary) OVER(ORDER BY Salary) AS RunningTotal
FROM Employees
```

So the above query is using the default value which is
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

This means the above query can be re-written using an explicit value for ROWS or RANGE clause as shown below.

```
SELECT Name, Salary,
      SUM(Salary) OVER(ORDER BY Salary
      RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningTotal
FROM Employees
```

We can also achieve the same result, by replacing RANGE with ROWS

```
SELECT Name, Salary,
      SUM(Salary) OVER(ORDER BY Salary
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningTotal
FROM Employees
```

What is the difference between ROWS and RANGE
To understand the difference we need some duplicate values for the Salary column in the Employees table.

Execute the following UPDATE script to introduce duplicate values in the Salary column

```
Update Employees set Salary = 1000 where Id = 2
Update Employees set Salary = 3000 where Id = 4
Go
```

Now execute the following query. Notice that we get the running total as expected.

```
SELECT Name, Salary,
    SUM(Salary) OVER(ORDER BY Salary
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningTotal
FROM Employees
```

Id	Name	Salary
1	Mark	1000
2	John	1000
3	Pam	3000
4	Sara	3000
5	Todd	3000

Name	Salary	RunningTotal
Mark	1000	1000
John	1000	2000
Pam	3000	5000
Sara	3000	8000
Todd	5000	13000

The following query uses RANGE instead of ROWS

```
SELECT Name, Salary,
    SUM(Salary) OVER(ORDER BY Salary
    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningTotal
FROM Employees
```

You get the following result when you execute the above query. Notice we don't get the running total as expected.

Id	Name	Salary
1	Mark	1000
2	John	1000
3	Pam	3000
4	Sara	3000
5	Todd	3000

Name	Salary	RunningTotal
Mark	1000	2000
John	1000	2000
Pam	3000	8000
Sara	3000	8000
Todd	5000	13000

So, the main difference between ROWS and RANGE is in the way duplicate rows are treated. ROWS treat duplicates as distinct values, where as RANGE treats them as a single entity.

All together side by side. The following query shows how running total changes

1. When no value is specified for ROWS or RANGE clause

2. When RANGE clause is used explicitly with it's default value
3. When ROWS clause is used instead of RANGE clause

```
SELECT Name, Salary,
      SUM(Salary) OVER(ORDER BY Salary) AS [Default],
      SUM(Salary) OVER(ORDER BY Salary
      RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS [Range],
      SUM(Salary) OVER(ORDER BY Salary
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS [Rows]
FROM Employees
```

Id	Name	Salary
1	Mark	1000
2	John	1000
3	Pam	3000
4	Sara	3000
5	Todd	3000

Name	Salary	Default	Range	Rows
Mark	1000	2000	2000	1000
John	1000	2000	2000	2000
Pam	3000	8000	8000	5000
Sara	3000	8000	8000	8000
Todd	5000	13000	13000	13000

LAST_VALUE function in SQL Server

Suggested Videos

Part 115 - FIRST_VALUE function in SQL Server
Part 116 - Window functions in SQL Server
Part 117 - Difference between rows and range

In this video we will discuss LAST_VALUE function in SQL Server.

LAST_VALUE function

- Introduced in SQL Server 2012
- Retrieves the last value from the specified column
- ORDER BY clause is required
- PARTITION BY clause is optional
- ROWS or RANGE clause is optional, but for it to work correctly you may have to explicitly specify a value

Syntax : LAST_VALUE(Column_Name) OVER (ORDER BY Col1, Col2, ...)

LAST_VALUE function not working as expected : In the following example, LAST_VALUE function does not return the name of the highest paid employee. This is because we have not specified an explicit value for ROWS or RANGE clause. As a result it is using it's default value RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

```
SELECT Name, Gender, Salary,
   LAST_VALUE(Name) OVER (ORDER BY Salary) AS LastValue
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	LastValue
Mark	Male	1000	Mark
John	Male	2000	John
Pam	Female	3000	Pam
Sara	Female	4000	Sara
Todd	Male	5000	Todd
Mary	Female	6000	Mary
Ben	Male	7000	Ben
Jodi	Female	8000	Jodi
Tom	Male	9000	Tom
Ron	Male	9500	Ron

LAST_VALUE function working as expected : In the following example, LAST_VALUE function returns the name of the highest paid employee as expected. Notice we have set an explicit value for ROWS or RANGE clause
to ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

This tells the LAST_VALUE function that it's window starts at the first row and ends at the last row in the result set.

```
SELECT Name, Gender, Salary,
   LAST_VALUE(Name) OVER (ORDER BY Salary ROWS BETWEEN UNBOUNDED PRECE
DING AND UNBOUNDED FOLLOWING) AS LastValue
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	LastValue
Mark	Male	1000	Ron
John	Male	2000	Ron
Pam	Female	3000	Ron
Sara	Female	4000	Ron
Todd	Male	5000	Ron
Mary	Female	6000	Ron
Ben	Male	7000	Ron
Jodi	Female	8000	Ron
Tom	Male	9000	Ron
Ron	Male	9500	Ron

LAST_VALUE function example with partitions : In the following example, LAST_VALUE

function returns the name of the highest paid employee from the respective partition.

```
SELECT Name, Gender, Salary,
   LAST_VALUE(Name) OVER (PARTITION BY Gender ORDER BY Salary
   ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS Last
Value
FROM Employees
```

Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	LastValue
Pam	Female	3000	Jodi
Sara	Female	4000	Jodi
Mary	Female	6000	Jodi
Jodi	Female	8000	Jodi
Mark	Male	1000	Ron
John	Male	2000	Ron
Todd	Male	5000	Ron
Ben	Male	7000	Ron
Tom	Male	9000	Ron
Ron	Male	9500	Ron

UNPIVOT in SQL Server

Suggested Videos

Part 116 - Window functions in SQL Server
Part 117 - Difference between rows and range
Part 118 - LAST_VALUE function in SQL Server

In this video we will discuss UNPIVOT operator in SQL Server.

PIVOT operator turns ROWS into COLUMNS, where as UNPIVOT turns COLUMNS into ROWS.

We discussed PIVOT operator in Part 54 of SQL Server tutorial. Please watch Part 54 before proceeding.

Let us understand UNPIVOT with an example. We will use the following tblProductSales table in this demo.

SalesAgent	India	US	UK
David	960	520	360
John	970	540	800
David	350	470	1340

SQL Script to create tblProductSales table

```
Create Table tblProductSales
(
 SalesAgent nvarchar(50),
 India int,
 US int,
 UK int
)
Go

Insert into tblProductSales values ('David', 960, 520, 360)
Insert into tblProductSales values ('John', 970, 540, 800)
Go
```

Write a query to turn COLUMNS into ROWS. The result of the query should be as shown below.

SalesAgent	India	US	UK
David	960	520	360
John	970	540	800

SalesAgent	Country	SalesAmount
David	India	960
David	US	520
David	UK	360
John	India	970
John	US	540
John	UK	800

```
SELECT SalesAgent, Country, SalesAmount
FROM tblProductSales
UNPIVOT
(
    SalesAmount
    FOR Country IN (India, US ,UK)
) AS UnpivotExample
```

Reverse PIVOT table in SQL Server

Suggested Videos

Part 117 - Difference between rows and range
Part 118 - LAST_VALUE function in SQL Server
Part 119 - UNPIVOT in SQL Server

In this video we will discuss if it's always possible to reverse what PIVOT operator has done using UNPIVOT operator.

Is it always possible to reverse what PIVOT operator has done using UNPIVOT operator.
No, not always. If the PIVOT operator has not aggregated the data, you can get your original data back using the UNPIVOT operator but not if the data is aggregated.

Let us understand this with an example. We will use the following table **tblProductSales** for the

examples in this video.

SalesAgent	Country	SalesAmount
David	India	960
David	US	520
John	India	970
John	US	540

SQL Script to create tblProductSales table

```
Create Table tblProductSales
(
    SalesAgent nvarchar(10),
    Country nvarchar(10),
    SalesAmount int
)
Go

Insert into tblProductSales values('David','India',960)
Insert into tblProductSales values('David','US',520)
Insert into tblProductSales values('John','India',970)
Insert into tblProductSales values('John','US',540)
Go
```

Let's now use the PIVOT operator to turn ROWS into COLUMNS

```
SELECT SalesAgent, India, US
FROM tblProductSales
PIVOT
(
    SUM(SalesAmount)
    FOR Country IN (India, US)
) AS PivotTable
```

The above query produces the following output

SalesAgent	Country	SalesAmount
David	India	960
David	US	520
John	India	970
John	US	540

→

SalesAgent	India	US
David	960	520
John	970	540

Now let's use the UNPIVOT operator to reverse what PIVOT operator has done

```
SELECT SalesAgent, Country, SalesAmount
FROM
(SELECT SalesAgent, India, US
FROM tblProductSales
```

```
PIVOT
(
    SUM(SalesAmount)
    FOR Country IN (India, US)
) AS PivotTable) P
UNPIVOT
(
    SalesAmount
    FOR Country IN (India, US)
) AS UnpivotTable
```

The above query reverses what PIVOT operator has done, and we get the original data back as shown below. We are able to get the original data back, because the SUM aggregate function that we used with the PIVOT operator did not perform any aggregation.

SalesAgent	India	US
David	960	520
John	970	540

→

SalesAgent	Country	SalesAmount
David	India	960
David	US	520
John	India	970
John	US	540

Now execute the following **INSERT** statement to insert a new row into **tblProductSales** table.

```
Insert into tblProductSales values('David','India',100)
```

With this new row in the table, if you execute the following **PIVOT** query data will be aggregated

```
SELECT SalesAgent, India, US
FROM tblProductSales
PIVOT
(
    SUM(SalesAmount)
    FOR Country IN (India, US)
) AS PivotTable
```

The following is the result of the above query

SalesAgent	Country	SalesAmount
David	India	960
David	US	520
John	India	970
John	US	540
David	India	100

→

SalesAgent	India	US
David	1060	520
John	970	540

Now if we use UNPIVOT opertaor with the above query, we wouldn't get our orginial data back as the PIVOT operator has already aggrgated the data, and there is no way for SQL Server to know how to undo the aggregations.

```
SELECT SalesAgent, Country, SalesAmount
FROM
(SELECT SalesAgent, India, US
FROM tblProductSales
PIVOT
(
    SUM(SalesAmount)
    FOR Country IN (India, US)
) AS PivotTable) P
UNPIVOT
(
    SalesAmount
    FOR Country IN (India, US)

) AS UnpivotTable
```

Notice that for SalesAgent - David and Country - India we get only one row. In the original table we had 2 rows for the same combination.

SalesAgent	India	US
David	1060	520
John	970	540

SalesAgent	Country	SalesAmount
David	India	1060
David	US	520
John	India	970
John	US	540

Choose function in SQL Server

Suggested Videos

Part 118 - LAST_VALUE function in SQL Server
Part 119 - UNPIVOT in SQL Server
Part 120 - Reverse PIVOT table in SQL Server

In this video we will discuss **Choose function in SQL Server**

Choose function

- Introduced in SQL Server 2012
- Returns the item at the specified index from the list of available values
- The index position starts at 1 and NOT 0 (ZERO)

Syntax : CHOOSE(index, val_1, val_2, ...)

Example : Returns the item at index position 2

```
SELECT CHOOSE(2, 'India','US', 'UK') AS Country
```

Output :

Country
US

Example : Using CHOOSE() function with table data. We will use the following **Employees** table for this example.

Id	Name	DateOfBirth
1	Mark	1980-01-11
2	John	1981-12-12
3	Amy	1979-11-21
4	Ben	1978-05-14
5	Sara	1970-03-17
6	David	1978-04-05

SQL Script to create Employees table

```
Create table Employees
(
    Id int primary key identity,
    Name nvarchar(10),
    DateOfBirth date
)
Go

Insert into Employees values ('Mark', '01/11/1980')
Insert into Employees values ('John', '12/12/1981')
Insert into Employees values ('Amy', '11/21/1979')
Insert into Employees values ('Ben', '05/14/1978')
Insert into Employees values ('Sara', '03/17/1970')
Insert into Employees values ('David', '04/05/1978')
Go
```

We want to display Month name along with employee Name and Date of Birth.

Name	DateOfBirth	MONTH
Mark	1980-01-11	JAN
John	1981-12-12	DEC
Amy	1979-11-21	NOV
Ben	1978-05-14	MAY
Sara	1970-03-17	MAR
David	1978-04-05	APR

Using CASE statement in SQL Server

```
SELECT Name, DateOfBirth,
      CASE DATEPART(MM, DateOfBirth)
         WHEN 1 THEN 'JAN'
         WHEN 2 THEN 'FEB'
         WHEN 3 THEN 'MAR'
```

```
            WHEN 4 THEN 'APR'
            WHEN 5 THEN 'MAY'
            WHEN 6 THEN 'JUN'
            WHEN 7 THEN 'JUL'
            WHEN 8 THEN 'AUG'
            WHEN 9 THEN 'SEP'
            WHEN 10 THEN 'OCT'
            WHEN 11 THEN 'NOV'
            WHEN 12 THEN 'DEC'
        END
    AS [MONTH]
FROM Employees
```

Using CHOOSE function in SQL Server : The amount of code we have to write is lot less than using CASE statement.

```
SELECT Name, DateOfBirth,CHOOSE(DATEPART(MM, DateOfBirth),
    'JAN', 'FEB', 'MAR', 'APR', 'MAY', 'JUN', 'JUL', 'AUG',
    'SEP', 'OCT', 'NOV', 'DEC') AS [MONTH]
FROM Employees
```

IIF function in SQL Server

Suggested Videos

Part 119 - UNPIVOT in SQL Server
Part 120 - Reverse PIVOT table in SQL Server
Part 121 - Choose function in SQL Server

In this video we will discuss **IIF function in SQL Server**.

IIF function

- Introduced in SQL Server 2012
- Returns one of two the values, depending on whether the Boolean expression evaluates to true or false
- IIF is a shorthand way for writing a CASE expression

Syntax : IIF (boolean_expression, true_value, false_value)

Example : Returns Male as the boolean expression evaluates to TRUE

```
DECLARE @Gender INT
SET @Gender = 1
SELECT IIF( @Gender = 1, 'Male', 'Femlae') AS Gender
```

Output :

Gender
Male

Example : Using IIF() function with table data. We will use the following **Employees** table for this example.

Id	Name	GenderId
1	Mark	1
2	John	1
3	Amy	2
4	Ben	1
5	Sara	2
6	David	1

SQL Script to create Employees table

```
Create table Employees
(
    Id int primary key identity,
    Name nvarchar(10),
    GenderId int
)
Go

Insert into Employees values ('Mark', 1)
Insert into Employees values ('John', 1)
Insert into Employees values ('Amy', 2)
Insert into Employees values ('Ben', 1)
Insert into Employees values ('Sara', 2)
Insert into Employees values ('David', 1)
Go
```

Write a query to display **Gender** along with **employee Name** and **GenderId**. We can achieve this either by using **CASE** or **IIF**.

Id	Name	GenderId
1	Mark	1
2	John	1
3	Amy	2
4	Ben	1
5	Sara	2
6	David	1

Name	GenderId	Gender
Mark	1	Male
John	1	Male
Amy	2	Female
Ben	1	Male
Sara	2	Female
David	1	Male

Using CASE statement

```
SELECT Name, GenderId,
      CASE WHEN GenderId = 1
                THEN 'Male'
                ELSE 'Female'
           END AS Gender
FROM Employees
```

Using IIF function

```
SELECT Name, GenderId, IIF(GenderId = 1, 'Male', 'Female') AS Gender
FROM Employees
```

TRY_PARSE function in SQL Server 2012

Suggested Videos

Part 120 - Reverse PIVOT table in SQL Server
Part 121 - Choose function in SQL Server
Part 122 - IIF function in SQL Server

In this video we will discuss

- TRY_PARSE function
- Difference between PARSE and TRY_PARSE functions

TRY_PARSE function

- Introduced in SQL Server 2012
- Converts a string to Date/Time or Numeric type
- Returns NULL if the provided string cannot be converted to the specified data type
- Requires .NET Framework Common Language Runtime (CLR)

Syntax : TRY_PARSE (string_value AS data_type)

Example : Convert string to INT. As the string can be converted to INT, the result will be 99 as expected.

```
SELECT TRY_PARSE('99' AS INT) AS Result
```

Output :

Result
99

Example : Convert string to INT. The string cannot be converted to INT, so TRY_PARSE returns NULL

```
SELECT TRY_PARSE('ABC' AS INT) AS Result
```

Output :

Result
NULL

Use **CASE** statement or **IIF** function to provide a meaningful error message instead of NULL when the conversion fails.

Example : Using CASE statement to provide a meaningful error message when the conversion fails.

```
SELECT
```

```
CASE WHEN TRY_PARSE('ABC' AS INT) IS NULL
        THEN 'Conversion Failed'
        ELSE 'Conversion Successful'
END AS Result
```

Output : As the conversion fails, you will now get a message 'Conversion Failed' instead of NULL

Result
Conversion Failed

Example : Using IIF function to provide a meaningful error message when the conversion fails.

```
SELECT IIF(TRY_PARSE('ABC' AS INT) IS NULL, 'Conversion Failed',
            'Conversion Successful') AS Result
```

What is the difference between PARSE and TRY_PARSE
PARSE will result in an error if the conversion fails, where as TRY_PARSE will return NULL instead of an error.

Since ABC cannot be converted to INT, PARSE will return an error
```
SELECT PARSE('ABC' AS INT) AS Result
```

Since ABC cannot be converted to INT, TRY_PARSE will return NULL instead of an error
```
SELECT TRY_PARSE('ABC' AS INT) AS Result
```

Example : Using TRY_PARSE() function with table data. We will use the following Employees table for this example.

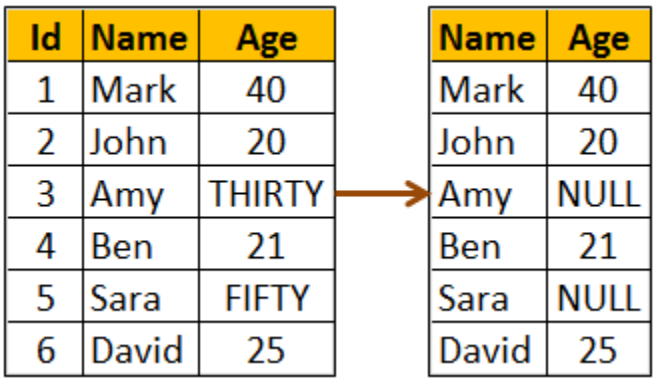
Id	Name	Age
1	Mark	40
2	John	20
3	Amy	THIRTY
4	Ben	21
5	Sara	FIFTY
6	David	25

SQL Script to create Employees table
```
Create table Employees
(
    Id int primary key identity,
    Name nvarchar(10),
    Age nvarchar(10)
)
Go

Insert into Employees values ('Mark', '40')
```

```
Insert into Employees values ('John', '20')
Insert into Employees values ('Amy', 'THIRTY')
Insert into Employees values ('Ben', '21')
Insert into Employees values ('Sara', 'FIFTY')
Insert into Employees values ('David', '25')
Go
```

The data type of Age column is nvarchar. So string values like (THIRTY, FIFTY) are also stored. Now, we want to write a query to convert the values in Age column to int and return along with the Employee name. Notice TRY_PARSE function returns NULL for the rows where age cannot be converted to INT.

```
SELECT Name, TRY_PARSE(Age AS INT) AS Age
FROM Employees
```



Id	Name	Age
1	Mark	40
2	John	20
3	Amy	THIRTY
4	Ben	21
5	Sara	FIFTY
6	David	25

Name	Age
Mark	40
John	20
Amy	NULL
Ben	21
Sara	NULL
David	25

If you use PARSE instead of TRY_PARSE, the query fails with an error.

```
SELECT Name, PARSE(Age AS INT) AS Age
FROM Employees
```

The above query returns the following error
Error converting string value 'THIRTY' into data type int using culture

TRY_CONVERT function in SQL Server 2012

Suggested Videos
Part 121 - Choose function in SQL Server
Part 122 - IIF function in SQL Server
Part 123 - TRY PARSE function in SQL Server 2012
In this video we will discuss

- TRY_CONVERT function
- Difference between CONVERT and TRY_CONVERT functions
- Difference between TRY_PARSE and TRY_CONVERT functions

TRY_CONVERT function

- Introduced in SQL Server 2012

- Converts a value to the specified data type
- Returns NULL if the provided value cannot be converted to the specified data type
- If you request a conversion that is explicitly not permitted, then TRY_CONVERT fails with an error

Syntax : TRY_CONVERT (data_type, value, [style])

Style parameter is optional. The range of acceptable values is determined by the target data_type. For the list of all possible values for style parameter, please visit the following MSDN article
https://msdn.microsoft.com/en-us/library/ms187928.aspx

Example : Convert string to INT. As the string can be converted to INT, the result will be 99 as expected.

```
SELECT TRY_CONVERT(INT, '99') AS Result
```

Output :

Result
99

Example : Convert string to INT. The string cannot be converted to INT, so TRY_CONVERT returns NULL

```
SELECT TRY_CONVERT(INT, 'ABC') AS Result
```

Output :

Result
NULL

Example : Converting an integer to XML is not explicitly permitted. so in this case TRY_CONVERT fails with an error

```
SELECT TRY_CONVERT(XML, 10) AS Result
```

If you want to provide a meaningful error message instead of NULL when the conversion fails, you can do so using CASE statement or IIF function.

Example : Using CASE statement to provide a meaningful error message when the conversion fails.

```
SELECT
CASE WHEN TRY_CONVERT(INT, 'ABC') IS NULL
        THEN 'Conversion Failed'
        ELSE 'Conversion Successful'
END AS Result
```

Output : As the conversion fails, you will now get a message 'Conversion Failed' instead of

NULL

Result
Conversion Failed

Example : Using IIF function to provide a meaningful error message when the conversion fails.

```
SELECT IIF(TRY_CONVERT(INT, 'ABC') IS NULL, 'Conversion Failed',
            'Conversion Successful') AS Result
```

What is the difference between CONVERT and TRY_CONVERT
CONVERT will result in an error if the conversion fails, where as TRY_CONVERT will return NULL instead of an error.

Since ABC cannot be converted to INT, CONVERT will return an error

```
SELECT CONVERT(INT, 'ABC') AS Result
```

Since ABC cannot be converted to INT, TRY_CONVERT will return NULL instead of an error

```
SELECT TRY_CONVERT(INT, 'ABC') AS Result
```

Example : Using TRY_CONVERT() function with table data. We will use the following Employees table for this example.

Id	Name	Age
1	Mark	40
2	John	20
3	Amy	THIRTY
4	Ben	21
5	Sara	FIFTY
6	David	25

SQL Script to create Employees table

```
Create table Employees
(
    Id int primary key identity,
    Name nvarchar(10),
    Age nvarchar(10)
)
Go

Insert into Employees values ('Mark', '40')
Insert into Employees values ('John', '20')
Insert into Employees values ('Amy', 'THIRTY')
Insert into Employees values ('Ben', '21')
Insert into Employees values ('Sara', 'FIFTY')
Insert into Employees values ('David', '25')
```

```
Go
```

The data type of Age column is nvarchar. So string values like (THIRTY, FIFTY) are also stored. Now, we want to write a query to convert the values in Age column to int and return along with the Employee name. Notice TRY_CONVERT function returns NULL for the rows where age cannot be converted to INT.

```
SELECT Name, TRY_CONVERT(INT, Age) AS Age
FROM Employees
```

Id	Name	Age
1	Mark	40
2	John	20
3	Amy	THIRTY
4	Ben	21
5	Sara	FIFTY
6	David	25

→

Name	Age
Mark	40
John	20
Amy	NULL
Ben	21
Sara	NULL
David	25

If you use CONVERT instead of TRY_CONVERT, the query fails with an error.

```
SELECT NAME, CONVERT(INT, Age) AS Age
FROM Employees
```

The above query returns the following error
Conversion failed when converting the nvarchar value 'THIRTY' to data type int.

Difference between TRY_PARSE and TRY_CONVERT functions
TRY_PARSE can only be used for converting from string to date/time or number data types where as TRY_CONVERT can be used for any general type conversions.

For example, you can use TRY_CONVERT to convert a string to XML data type, where as you can do the same using TRY_PARSE

Converting a string to XML data type using TRY_CONVERT

```
SELECT TRY_CONVERT(XML, '<root><child/></root>') AS [XML]
```

The above query produces the following

XML
<root><child /></root>

Converting a string to XML data type using TRY_PARSE

```
SELECT TRY_PARSE('<root><child/></root>' AS XML) AS [XML]
```

The above query will result in the following error
Invalid data type xml in function TRY_PARSE

Another difference is TRY_PARSE relies on the presence of .the .NET Framework Common Language Runtime (CLR) where as TRY_CONVERT does not.

EOMONTH function in SQL Server 2012

Suggested Videos
Part 122 - IIF function in SQL Server
Part 123 - TRY_PARSE function in SQL Server 2012
Part 124 - TRY_CONVERT function in SQL Server 2012
In this video we will discuss **EOMONTH function in SQL Server 2012**

EOMONTH function

- Introduced in SQL Server 2012
- Returns the last day of the month of the specified date

Syntax : EOMONTH (start_date [, month_to_add])

start_date : The date for which to return the last day of the month
month_to_add : Optional. Number of months to add to the start_date. EOMONTH adds the specified number of months to start_date, and then returns the last day of the month for the resulting date.

Example : Returns last day of the month November

```
SELECT EOMONTH('11/20/2015') AS LastDay
```

Output :

LastDay
30/11/2015

Example : Returns last day of the month of February from a NON-LEAP year

```
SELECT EOMONTH('2/20/2015') AS LastDay
```

Output :

LastDay
28/02/2015

Example : Returns last day of the month of February from a LEAP year

```
SELECT EOMONTH('2/20/2016') AS LastDay
```

Output :

LastDay
29/02/2016

month_to_add optional parameter can be used to add or subtract a specified number of months from the start_date, and then return the last day of the month from the resulting date.

The following example adds 2 months to the start_date and returns the last day of the month from the resulting date

```
SELECT EOMONTH('3/20/2016', 2) AS LastDay
```

Output :

LastDay
31/05/2016

The following example subtracts 1 month from the start_date and returns the last day of the month from the resulting date

```
SELECT EOMONTH('3/20/2016', -1) AS LastDay
```

Output :

LastDay
29/02/2016

Using **EOMONTH** function with table data. We will use the following **Employees** table for this example.

Id	Name	DateOfBirth
1	Mark	11/01/1980
2	John	12/12/1981
3	Amy	21/11/1979
4	Ben	14/05/1978
5	Sara	17/03/1970
6	David	05/04/1978

SQL Script to create Employees table

```
Create table Employees
(
   Id int primary key identity,
   Name nvarchar(10),
   DateOfBirth date
)
Go

Insert into Employees values ('Mark', '01/11/1980')
Insert into Employees values ('John', '12/12/1981')
Insert into Employees values ('Amy', '11/21/1979')
Insert into Employees values ('Ben', '05/14/1978')
Insert into Employees values ('Sara', '03/17/1970')
Insert into Employees values ('David', '04/05/1978')
Go
```

The following example returns the last day of the month from the DateOfBirth of every

employee.

```
SELECT Name, DateOfBirth, EOMONTH(DateOfBirth) AS LastDay
FROM Employees
```

Id	Name	DateOfBirth
1	Mark	11/01/1980
2	John	12/12/1981
3	Amy	21/11/1979
4	Ben	14/05/1978
5	Sara	17/03/1970
6	David	05/04/1978

Name	DateOfBirth	LastDay
Mark	11/01/1980	31/01/1980
John	12/12/1981	31/12/1981
Amy	21/11/1979	30/11/1979
Ben	14/05/1978	31/05/1978
Sara	17/03/1970	31/03/1970
David	05/04/1978	30/04/1978

If you want just the last day instead of the full date, you can use **DATEPART** function

```
SELECT Name, DateOfBirth, DATEPART(DD,EOMONTH(DateOfBirth)) AS LastDay
FROM Employees
```

Id	Name	DateOfBirth
1	Mark	11/01/1980
2	John	12/12/1981
3	Amy	21/11/1979
4	Ben	14/05/1978
5	Sara	17/03/1970
6	David	05/04/1978

Name	DateOfBirth	LastDay
Mark	11/01/1980	31
John	12/12/1981	31
Amy	21/11/1979	30
Ben	14/05/1978	31
Sara	17/03/1970	31
David	05/04/1978	30

DATEFROMPARTS function in SQL Server

Suggested Videos

Part 123 - TRY_PARSE function in SQL Server 2012
Part 124 - TRY_CONVERT function in SQL Server 2012
Part 125 - EOMONTH function in SQL Server 2012

In this video we will discuss **DATEFROMPARTS function in SQL Server**

DATEFROMPARTS function

- Introduced in SQL Server 2012
- Returns a date value for the specified year, month, and day
- The data type of all the 3 parameters (year, month, and day) is integer
- If invalid argument values are specified, the function returns an error
- If any of the arguments are NULL, the function returns null

Syntax : DATEFROMPARTS (year, month, day)

Example : All the function arguments have valid values, so DATEFROMPARTS returns the expected date

```
SELECT DATEFROMPARTS ( 2015, 10, 25) AS [Date]
```

Output :

Date
25/10/2015

Example : Invalid value specified for month parameter, so the function returns an error

```
SELECT DATEFROMPARTS ( 2015, 15, 25) AS [Date]
```

Output : Cannot construct data type date, some of the arguments have values which are not valid.

Example : NULL specified for month parameter, so the function returns NULL.

```
SELECT DATEFROMPARTS ( 2015, NULL, 25) AS [Date]
```

Output :

Date
NULL

Other new date and time functions introduced in SQL Server 2012

- EOMONTH (Discussed in Part 125 of SQL Server tutorial)
- **DATETIMEFROMPARTS :** Returns DateTime
- **Syntax**

: DATETIMEFROMPARTS (year, month, day, hour, minute, seconds, milliseconds)

- **SMALLDATETIMEFROMPARTS :** Returns SmallDateTime
- **Syntax :** SMALLDATETIMEFROMPARTS (year, month, day, hour, minute)
- **We will discuss the following functions in a later video**
- TIMEFROMPARTS
- DATETIME2FROMPARTS
- DATETIMEOFFSETFROMPARTS

In our next video we will discuss the **difference between DateTime and SmallDateTime**.

Difference between DateTime and SmallDateTime in SQL Server

Suggested Videos

Part 124 - TRY_CONVERT function in SQL Server 2012
Part 125 - EOMONTH function in SQL Server 2012
Part 126 - DATEFROMPARTS function in SQL Server

In this video we will discuss the **difference between DateTime and SmallDateTime in SQL Server**

The following **table summarizes the differences**

Attribute	SmallDateTime	DateTime
Date Range	January 1, 1900, through June 6, 2079	January 1, 1753, through December 31, 9999
Time Range	00:00:00 through 23:59:59	00:00:00 through 23:59:59.997
Accuracy	1 Minute	3.33 Milli-seconds
Size	4 Bytes	8 Bytes
Default value	1900-01-01 00:00:00	1900-01-01 00:00:00

The range for SmallDateTime is **January 1, 1900**, through **June 6, 2079**. A value outside of this range, is not allowed.

The following 2 queries have values outside of the range of SmallDateTime data type.

```
Insert into Employees ([SmallDateTime]) values ('01/01/1899')
Insert into Employees ([SmallDateTime]) values ('07/06/2079')
```

When executed, the above queries fail with the following error

The conversion of a varchar data type to a smalldatetime data type resulted in an out-of-range value

The range for DateTime is **January 1, 1753**, through **December 31, 9999**. A value outside of this range, is not allowed.

The following query has a value outside of the range of DateTime data type.

```
Insert into Employees ([DateTime]) values ('01/01/1752')
```

When executed, the above query fails with the following error

The conversion of a varchar data type to a datetime data type resulted in an out-of-range value.

DateTime2FromParts function in SQL Server 2012

Suggested Videos

Part 125 - EOMONTH function in SQL Server 2012
Part 126 - DATEFROMPARTS function in SQL Server
Part 127 - Difference between DateTime and SmallDateTime in SQL Server

In this video we will discuss **DateTime2FromParts function in SQL Server 2012**.

DateTime2FromParts function

- Introduced in SQL Server 2012
- Returns DateTime2
- The data type of all the parameters is integer
- If invalid argument values are specified, the function returns an error
- If any of the required arguments are NULL, the function returns null
- If the precision argument is null, the function returns an error

Syntax : DATETIME2FROMPARTS (year, month, day, hour, minute, seconds, fractions, precision)

Example : All the function arguments have valid values, so DATETIME2FROMPARTS returns DATETIME2 value as expected.

```
SELECT DATETIME2FROMPARTS ( 2015, 11, 15, 20, 55, 55, 0, 0 ) AS [DateTime2]
```

Output :

DateTime2
2015-11-15 20:55:55

Example : Invalid value specified for month parameter, so the function returns an error

```
SELECT DATETIME2FROMPARTS ( 2015, 15, 15, 20, 55, 55, 0, 0 ) AS [DateTime2]
```

Output : Cannot construct data type datetime2, some of the arguments have values which are not valid.

Example : If any of the required arguments are NULL, the function returns null. NULL specified for month parameter, so the function returns NULL.

```
SELECT DATETIME2FROMPARTS ( 2015, NULL, 15, 20, 55, 55, 0, 0 ) AS [DateTime2]
```

Output :

DateTime2
NULL

Example : If the precision argument is null, the function returns an error

```
SELECT DATETIME2FROMPARTS ( 2015, 15, 15, 20, 55, 55, 0, NULL ) AS [DateTime2]
```

Output : Scale argument is not valid. Valid expressions for data type datetime2 scale argument are integer constants and integer constant expressions.

TIMEFROMPARTS : Returns time value

Syntax : TIMEFROMPARTS (hour, minute, seconds, fractions, precision)

Next video : We will discuss the **difference between DateTime and DateTime2 in SQL Server**

Difference between DateTime and DateTime2 in SQL Server

Suggested Videos

Part 126 - DATEFROMPARTS function in SQL Server
Part 127 - Difference between DateTime and SmallDateTime in SQL Server
Part 128 - DateTime2FromParts function in SQL Server 2012

In this video we will discuss the **difference between DateTime and DateTime2 in SQL Server**

Differences between DateTime and DateTime2

Attribute	DateTime	DateTime2

Date Range	January 1, 1753, through December 31, 9999	January 1, 0001, through December 31, 9999
Time Range	00:00:00 through 23:59:59.997	00:00:00 through 23:59:59.9999999
Accuracy	3.33 Milli-seconds	100 nanoseconds
Size	8 Bytes	6 to 8 Bytes (Depends on the precision)
Default Value	1900-01-01 00:00:00	1900-01-01 00:00:00

DATETIME2 has a bigger date range than DATETIME. Also, DATETIME2 is more accurate than DATETIME. So I would recommend using DATETIME2 over DATETIME when possible. I think the only reason for using DATETIME over DATETIME2 is for backward compatibility.

DateTime2 Syntax : DATETIME2 [(fractional seconds precision)]

With DateTime2

- Optional fractional seconds precision can be specified
- The precision scale is from 0 to 7 digits
- The default precision is 7 digits
- For precision 1 and 2, storage size is 6 bytes
- For precision 3 and 4, storage size is 7 bytes
- For precision 5, 6 and 7, storage size is 8 bytes

The following script creates a table variable with 7 DATETIME2 columns with different precision start from 1 through 7

```
DECLARE @TempTable TABLE
(
   DateTime2Precision1 DATETIME2(1),
   DateTime2Precision2 DATETIME2(2),
   DateTime2Precision3 DATETIME2(3),
   DateTime2Precision4 DATETIME2(4),
   DateTime2Precision5 DATETIME2(5),
   DateTime2Precision6 DATETIME2(6),
   DateTime2Precision7 DATETIME2(7)
)
```

Insert DateTime value into each column

```
INSERT INTO @TempTable VALUES
(
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567'
)
```

The following query retrieves the prcision, the datetime value, and the storage size.

```
SELECT 'Precision - 1' AS [Precision],
	DateTime2Precision1 AS DateValue,
	DATALENGTH(DateTime2Precision1) AS StorageSize
FROM @TempTable

UNION ALL

SELECT 'Precision - 2',
	DateTime2Precision2,
	DATALENGTH(DateTime2Precision2) AS StorageSize
FROM @TempTable

UNION ALL

SELECT 'Precision - 3',
	DateTime2Precision3,
	DATALENGTH(DateTime2Precision3)
FROM @TempTable

UNION ALL

SELECT 'Precision - 4',
	DateTime2Precision4,
	DATALENGTH(DateTime2Precision4)
FROM @TempTable

UNION ALL

SELECT 'Precision - 5',
	DateTime2Precision5,
	DATALENGTH(DateTime2Precision5)
FROM @TempTable

UNION ALL

SELECT 'Precision - 6',
	DateTime2Precision6,
	DATALENGTH(DateTime2Precision6)
FROM @TempTable

UNION ALL
SELECT 'Precision - 7',
	DateTime2Precision7,
	DATALENGTH(DateTime2Precision7) AS StorageSize
FROM @TempTable
```

Notice as the precision increases the storage size also increases

Precision	DateValue	StorageSize
Precision - 0	2015-10-20 15:09:12.0000000	6
Precision - 1	2015-10-20 15:09:12.1000000	6
Precision - 2	2015-10-20 15:09:12.1200000	6
Precision - 3	2015-10-20 15:09:12.1230000	7
Precision - 4	2015-10-20 15:09:12.1235000	7
Precision - 5	2015-10-20 15:09:12.1234600	8
Precision - 6	2015-10-20 15:09:12.1234570	8
Precision - 7	2015-10-20 15:09:12.1234567	8

Offset fetch next in SQL Server 2012

Suggested Videos

Part 127 - Difference between DateTime and SmallDateTime in SQL Server
Part 128 - DateTime2FromParts function in SQL Server 2012
Part 129 - Difference between DateTime and DateTime2 in SQL Server
In this video we will discuss **OFFSET FETCH Clause in SQL Server 2012**

One of the common tasks for a SQL developer is to come up with a stored procedure that can return a page of results from the result set. With SQL Server 2012 OFFSET FETCH Clause it is very easy to implement paging.

Let's understand this with an example. We will use the following **tblProducts** table for the examples in this video. The table has got 100 rows. In the image I have shown just 10 rows.

Id	Name	Description	Price
1	Product - 1	Product Description - 1	10
2	Product - 2	Product Description - 2	20
3	Product - 3	Product Description - 3	30
4	Product - 4	Product Description - 4	40
5	Product - 5	Product Description - 5	50
6	Product - 6	Product Description - 6	60
7	Product - 7	Product Description - 7	70
8	Product - 8	Product Description - 8	80
9	Product - 9	Product Description - 9	90
10	Product - 10	Product Description - 10	100

SQL Script to create tblProducts table

```
Create table tblProducts
```

```
(
    Id int primary key identity,
    Name nvarchar(25),
    [Description] nvarchar(50),
    Price int
)
Go
```

SQL Script to populate tblProducts table with 100 rows

```
Declare @Start int
Set @Start = 1

Declare @Name varchar(25)
Declare @Description varchar(50)

While(@Start <= 100)
Begin
    Set @Name = 'Product - ' + LTRIM(@Start)
    Set @Description = 'Product Description - ' + LTRIM(@Start)
    Insert into tblProducts values (@Name, @Description, @Start * 10)
    Set @Start = @Start + 1
End
```

OFFSET FETCH Clause

- Introduced in SQL Server 2012
- Returns a page of results from the result set
- ORDER BY clause is required

OFFSET FETCH Syntax :

```
SELECT * FROM Table_Name
ORDER BY Column_List
OFFSET Rows_To_Skip ROWS
FETCH NEXT Rows_To_Fetch ROWS ONLY
```

The following SQL query

1. Sorts the table data by Id column
2. Skips the first 10 rows and
3. Fetches the next 10 rows

```
SELECT * FROM tblProducts
ORDER BY Id
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY
```

Result :

Id	Name	Description	Price
11	Product - 11	Product Description - 11	110
12	Product - 12	Product Description - 12	120
13	Product - 13	Product Description - 13	130
14	Product - 14	Product Description - 14	140
15	Product - 15	Product Description - 15	150
16	Product - 16	Product Description - 16	160
17	Product - 17	Product Description - 17	170
18	Product - 18	Product Description - 18	180
19	Product - 19	Product Description - 19	190
20	Product - 20	Product Description - 20	200

From the front-end application, we would typically send the **PAGE NUMBER** and the **PAGE SIZE** to get a page of rows. The following stored procedure accepts PAGE NUMBER and the PAGE SIZE as parameters and returns the correct set of rows.

```
CREATE PROCEDURE spGetRowsByPageNumberAndSize
@PageNumber INT,
@PageSize INT
AS
BEGIN
   SELECT * FROM tblProducts
   ORDER BY Id
   OFFSET (@PageNumber - 1) * @PageSize ROWS
   FETCH NEXT @PageSize ROWS ONLY
END
```

With PageNumber = 3 and PageSize = 10, the stored procedure returns the correct set of rows

```
EXECUTE spGetRowsByPageNumberAndSize 3, 10
```

Id	Name	Description	Price
21	Product - 21	Product Description - 21	210
22	Product - 22	Product Description - 22	220
23	Product - 23	Product Description - 23	230
24	Product - 24	Product Description - 24	240
25	Product - 25	Product Description - 25	250
26	Product - 26	Product Description - 26	260
27	Product - 27	Product Description - 27	270
28	Product - 28	Product Description - 28	280
29	Product - 29	Product Description - 29	290
30	Product - 30	Product Description - 30	300

Identifying object dependencies in SQL Server

Suggested Videos

Part 128 - DateTime2FromParts function in SQL Server 2012
Part 129 - Difference between DateTime and DateTime2 in SQL Server
Part 130 - Offset fetch next in SQL Server 2012

In this video we will discuss **how to identify object dependencies in SQL Server using SQL Server Management Studio**.

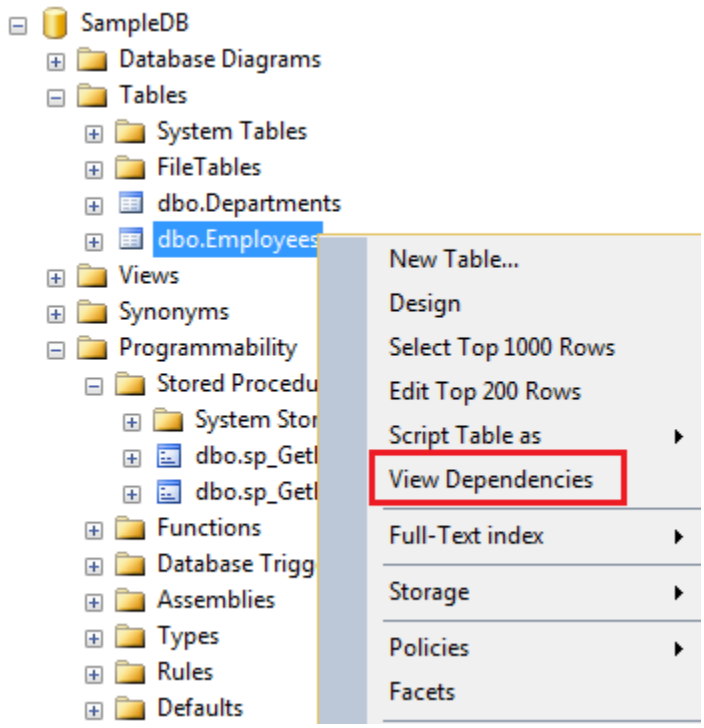
The following SQL Script creates 2 tables, 2 stored procedures and a view

```
Create table Departments
(
   Id int primary key identity,
   Name nvarchar(50)
)
Go

Create table Employees
(
   Id int primary key identity,
   Name nvarchar(50),
   Gender nvarchar(10),
   DeptId int foreign key references Departments(Id)
)
Go

Create procedure sp_GetEmployees
as
Begin
   Select * from Employees
End
Go
```

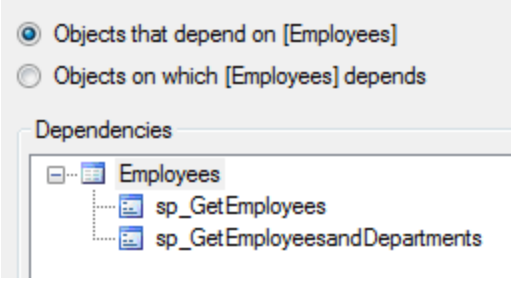
```
Create procedure sp_GetEmployeesandDepartments
as
Begin
   Select Employees.Name as EmployeeName,
            Departments.Name as DepartmentName
   from Employees
   join Departments
   on Employees.DeptId = Departments.Id
End
Go

Create view VwDepartments
as
Select * from Departments
Go
```

How to find dependencies using SQL Server Management Studio
Use View Dependencies option in SQL Server Management studio to find the object dependencies

For example : To find the dependencies on the Employees table, right click on it and select View Dependencies from the context menu



In the **Object Dependencies** window, depending on the radio button you select, you can find the objects that depend on **Employees** table and the objects on which **Employees** table depends on.



Identifying object dependencies is important especially when you intend to modify or delete an object upon which other objects depend. Otherwise you may risk breaking the functionality.

For example, there are 2 stored procedures (sp_GetEmployees and sp_GetEmployeesandDepartments) that depend on the Employees table. If we are not aware of these dependencies and if we delete the Employees table, both stored procedures will fail with the following error.

Msg 208, Level 16, State 1, Procedure sp_GetEmployees, Line 4
Invalid object name 'Employees'.

There are other ways for finding object dependencies in SQL Server which we will discuss in our upcoming videos.

sys.dm_sql_referencing_entities in SQL Server

Suggested Videos

Part 129 - Difference between DateTime and DateTime2 in SQL Server
Part 130 - Offset fetch next in SQL Server 2012
Part 131 - Identifying object dependencies in SQL Server
In this video we will discuss

- How to find object dependencies using the following dynamic management functions
- sys.dm_sql_referencing_entities
- sys.dm_sql_referenced_entities
- Difference between
- Referencing entity and Referenced entity
- Schema-bound dependency and Non-schema-bound dependency
- This is continuation to Part 131, in which we discussed how to find object dependencies using SQL Server Management Studio. Please watch Part 131 from SQL Server tutorial before proceeding.

  The following example returns all the objects that depend on Employees table.

```
Select * from sys.dm_sql_referencing_entities('dbo.Employees','Object')
```

  Difference between referencing entity and referenced entity

  A dependency is created between two objects when one object appears by name inside a SQL statement stored in another object. The object which is appearing inside the SQL expression is known as referenced entity and the object which has the SQL expression is known as a referencing entity.

To get the REFERENCING ENTITIES use
SYS.DM_SQL_REFERENCING_ENTITIES dynamic management function

To get the REFERENCED ENTITIES use
SYS.DM_SQL_REFERENCED_ENTITIES dynamic management function

Now, let us say we have a stored procedure and we want to find the all objects that this stored procedure depends on. This can be very achieved using another dynamic management function, sys.dm_sql_referenced_entities.

The following query returns all the referenced entities of the stored procedure sp_GetEmployeesandDepartments

- Select * from
- sys.dm_sql_referenced_entities('dbo.sp_GetEmployeesandDepartments','Object')
-

Please note : For both these dynamic management functions to work we need to specify the schema name as well. Without the schema name you may not get any results.

Difference between Schema-bound dependency and Non-schema-bound dependency
Schema-bound dependency : Schema-bound dependency prevents referenced objects from being dropped or modified as long as the referencing object exists

Example : A view created with SCHEMABINDING, or a table created with foreign key constraint.

Non-schema-bound dependency : A non-schema-bound dependency doesn't prevent the referenced object from being dropped or modified.

-

- ## sp_depends in SQL Server
- **Suggested Videos**
  Part 130 - Offset fetch next in SQL Server 2012
  Part 131 - Identifying object dependencies in SQL Server
  Part 132 - sys.dm_sql_referencing_entities in SQL Server

There are several ways to find object dependencies in SQL Server
1. View Dependencies feature in SQL Server Management Studio - Part 131
2. SQL Server dynamic management functions - Part 132
sys.dm_sql_referencing_entities
sys.dm_sql_referenced_entities
3. sp_depends system stored procedure - This video

sp_depends
A system stored procedure that returns object dependencies
For example,

- If you specify a table name as the argument, then the views and procedures that depend on the specified table are displayed

- If you specify a view or a procedure name as the argument, then the tables and views on which the specified view or procedure depends are displayed.

Syntax : Execute sp_depends 'ObjectName'

The following SQL Script creates a table and a stored procedure

```
Create table Employees
(
   Id int primary key identity,
   Name nvarchar(50),
   Gender nvarchar(10)
)
Go

Create procedure sp_GetEmployees
as
Begin
   Select * from Employees
End
Go
```

Returns the stored procedure that depends on table Employees

```
sp_depends 'Employees'
```

Ouptut :

name	type
dbo.sp_GetEmployees	stored procedure

Returns the name of the table and the respective column names on which the stored procedure sp_GetEmployees depends

```
sp_depends 'sp_GetEmployees'
```

Output :

name	type	updated	selected	column
dbo.Employees	user table	no	yes	Id
dbo.Employees	user table	no	yes	Name
dbo.Employees	user table	no	yes	Gender

Sometime sp_depends does not report dependencies correctly. For example, at the moment we have Employees table and a stored procedure sp_GetEmployees.

Now drop the table Employees

```
Drop table Employees
```

and then recreate the table again

```
Create table Employees
(
```

```
    Id int primary key identity,
    Name nvarchar(50),
    Gender nvarchar(10)
)
Go
```

Now execute the following, to find the objects that depend on Employees table

```
sp_depends 'Employees'
```

- We know that stored procedure **sp_GetEmployees** still depends on **Employees** table. But sp_depends does not report this dependency, as the Employees table is dropped and recreated.
  Object does not reference any object, and no objects reference it.

  sp_depends is on the deprecation path. This might be removed from the future versions of SQL server.
- **Sequence object in SQL Server 2012**
- **Suggested Videos**
  Part 131 - Identifying object dependencies in SQL Server
  Part 132 - sys.dm_sql_referencing_entities in SQL Server
  Part 133 - sp_ depends in SQL Server
- In this video we will discuss **sequence object in SQL Server**.

Sequence object

- Introduced in SQL Server 2012
- Generates sequence of numeric values in an ascending or descending order

Syntax :

```
CREATE SEQUENCE [schema_name . ] sequence_name
    [ AS [ built_in_integer_type | user-defined_integer_type ] ]
    [ START WITH <constant> ]
    [ INCREMENT BY <constant> ]
    [ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]
    [ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]
    [ CYCLE | { NO CYCLE } ]
    [ { CACHE [ <constant> ] } | { NO CACHE } ]
    [ ; ]
```

Property	Description
DataType	Built-in integer type (tinyint , smallint, int, bigint, decimal etc...) or user-defined integer type. Default bigint.
START WITH	The first value returned by the sequence object
INCREMENT BY	The value to increment or decrement by. The value will be decremented if a negative value is specified.
MINVALUE	Minimum value for the sequence object
MAXVALUE	Maximum value for the sequence object
CYCLE	Specifies whether the sequence object should restart when the max value (for incrementing sequence object) or min value (for decrementing sequence object) is reached. Default is NO CYCLE, which throws an error when minimum or

	maximum value is exceeded.
CACHE	Cache sequence values for performance. Default value is CACHE.

- **Creating an Incrementing Sequence :** The following code create a Sequence object that starts with 1 and increments by 1

```
CREATE SEQUENCE [dbo].[SequenceObject]
AS INT
START WITH 1
INCREMENT BY 1
```

Generating the Next Sequence Value : Now we have a sequence object created. To generate the sequence value use NEXT VALUE FOR clause

```
SELECT NEXT VALUE FOR [dbo].[SequenceObject]
```

Output : 1

Every time you execute the above query the sequence value will be incremented by 1. I executed the above query 5 times, so the current sequence value is 5.

Retrieving the current sequence value : If you want to see what the current Sequence value before generating the next, use **sys.sequences**

```
SELECT * FROM sys.sequences WHERE name = 'SequenceObject'
```

Alter the Sequence object to reset the sequence value :

```
ALTER SEQUENCE [SequenceObject] RESTART WITH 1
```

Select the next sequence value to make sure the value starts from 1

```
SELECT NEXT VALUE FOR [dbo].[SequenceObject]
```

Using sequence value in an INSERT query :

```
CREATE TABLE Employees
(
   Id INT PRIMARY KEY,
   Name NVARCHAR(50),
   Gender NVARCHAR(10)
)

-- Generate and insert Sequence values
INSERT INTO Employees VALUES
(NEXT VALUE for [dbo].[SequenceObject], 'Ben', 'Male')
INSERT INTO Employees VALUES
(NEXT VALUE for [dbo].[SequenceObject], 'Sara', 'Female')

-- Select the data from the table
SELECT * FROM Employees
```

-

Id	Name	Gender
2	Ben	Male
3	Sara	Female

Creating the decrementing Sequence : The following code create a Sequence object that starts with 100 and decrements by 1

```
CREATE SEQUENCE [dbo].[SequenceObject]
AS INT
START WITH 100
INCREMENT BY -1
```

Specifying MIN and MAX values for the sequence : Use the MINVALUE and MAXVALUE arguments to specify the MIN and MAX values respectively.

Step 1 : Create the Sequence object

```
CREATE SEQUENCE [dbo].[SequenceObject]
  START WITH 100
  INCREMENT BY 10
  MINVALUE 100
  MAXVALUE 150
```

Step 2 : Retrieve the next sequence value. The sequence value starts at 100. Every time we call NEXT VALUE, the value will be incremented by 10.

```
SELECT NEXT VALUE FOR [dbo].[SequenceObject]
```

If you call NEXT VALUE, when the value reaches 150 (MAXVALUE), you will get the following error

- ## Difference between sequence and identity in SQL Server
- **Suggested Videos**
  Part 132 - sys.dm_sql_referencing_entities in SQL Server
  Part 133 - sp_depends in SQL Server
  Part 134 - Sequence object in SQL Server 2012
- In this video we will discuss the **difference between SEQUENCE and IDENTITY in SQL Server**
-

This is continuation to Part 134. Please watch Part 134 from SQL Server tutorial before proceeding.

Sequence object is similar to the Identity property, in the sense that it generates sequence of numeric values in an ascending order just like the identity property. However there are several differences between the 2 which we will discuss in this video.

Identity property is a table column property meaning it is tied to the table, where as the sequence is a user-defined database object and is not tied to any specific table meaning it's value can be shared by multiple tables.

Example : Identity property tied to the Id column of the Employees table.

```
CREATE TABLE Employees
(
    Id INT PRIMARY KEY IDENTITY(1,1),
    Name NVARCHAR(50),
    Gender NVARCHAR(10)
)
```

Example : Sequence object not tied to any specific table

```
CREATE SEQUENCE [dbo].[SequenceObject]
AS INT
START WITH 1
INCREMENT BY 1
```

This means the above sequence object can be used with any table.

Example : Sharing sequence object value with multiple tables.

Step 1 : Create Customers and Users tables

```
CREATE TABLE Customers
(
    Id INT PRIMARY KEY,
    Name NVARCHAR(50),
    Gender NVARCHAR(10)
)
GO

CREATE TABLE Users
(
    Id INT PRIMARY KEY,
    Name NVARCHAR(50),
    Gender NVARCHAR(10)
)
GO
```

Step 2 : Insert 2 rows into Customers table and 3 rows into Users table. Notice the same sequence object is generating the ID values for both the tables.

```
INSERT INTO Customers VALUES
  (NEXT VALUE for [dbo].[SequenceObject], 'Ben', 'Male')
INSERT INTO Customers VALUES
  (NEXT VALUE for [dbo].[SequenceObject], 'Sara', 'Female')

INSERT INTO Users VALUES
  (NEXT VALUE for [dbo].[SequenceObject], 'Tom', 'Male')
INSERT INTO Users VALUES
  (NEXT VALUE for [dbo].[SequenceObject], 'Pam', 'Female')
```

- INSERT INTO Users VALUES
- (NEXT VALUE for [dbo].[SequenceObject], 'David', 'Male')
- GO
-

Step 3 : Query the tables

- SELECT * FROM Customers
- SELECT * FROM Users
- GO
-

Output : Notice the same sequence object has generated the values for ID columns in both the tables

Customers		
Id	Name	Gender
1	Ben	Male
2	Sara	Female

Users		
Id	Name	Gender
3	Tom	Male
4	Pam	Female
5	David	Male

To generate the next identity value, a row has to be inserted into the table, where as with sequence object there is no need to insert a row into the table to generate the next sequence value. You can use NEXT VALUE FOR clause to generate the next sequence value.

Example : Generating Identity value by inserting a row into the table

```
INSERT INTO Employees VALUES ('Todd', 'Male')
```

Example : Generating the next sequence value using NEXT VALUE FOR clause.

```
SELECT NEXT VALUE FOR [dbo].[SequenceObject]
```

Maximum value for the identity property cannot be specified. The maximum value will be the maximum value of the correspoding column data type. With the sequence object you can use the MAXVALUE option to specify the maximum value. If the MAXVALUE option is not specified for the sequence object, then the maximum value will be the maximum value of it's data type.

Example : Specifying maximum value for the sequence object using the MAXVALUE option

- CREATE SEQUENCE [dbo].[SequenceObject]
- START WITH 1
- INCREMENT BY 1
- MAXVALUE 5
-

CYCLE option of the Sequence object can be used to specify whether the sequence should restart automatically when the max value (for incrementing sequence object) or

min value (for decrementing sequence object) is reached, where as with the Identity property we don't have any such option to automatically restart the identity values.

Example : Specifying the CYCLE option of the Sequence object, so the sequence will restart automatically when the max value is exceeded

- CREATE SEQUENCE [dbo].[SequenceObject]
- START WITH 1
- INCREMENT BY 1
- MINVALUE 1
- MAXVALUE 5
- CYCLE
- **Guid in SQL Server**
- **Suggested Videos**
  Part 133 - sp_depends in SQL Server
  Part 134 - Sequence object in SQL Server 2012
  Part 135 - Difference between sequence and identity in SQL Server

  In this video we will discuss
  1. What is a GUID in SQL Server
  2. When to use GUID
  3. Advantages and disadvantages of using a GUID

  What is in SQL Server
  The GUID data type is a 16 byte binary data type that is globally unique. GUID stands for Global Unique Identifier. The terms GUID and UNIQUEIDENTIFIER are used interchangeably.
-

  To declare a GUID variable, we use the keyword UNIQUEIDENTIFIER

- Declare @ID UNIQUEIDENTIFIER
- SELECT @ID = NEWID()
- SELECT @ID as MYGUID

How to create a GUID in sql server
To create a GUID in SQL Server use NEWID() function

For example, SELECT NEWID(), creates a GUID that is guaranteed to be unique across tables, databases, and servers. Every time you execute SELECT NEWID() query, you get a GUID that is unique.

Example GUID : 0BB83607-00D7-4B2C-8695-32AD3812B6F4

When to use GUID data type : Let us understand when to use a GUID in SQL Server with an example.

1. Let us say our company does business in 2 countries - USA and India.

2. USA customers are stored in a table called USACustomers in a database called USADB.

Create Database USADB

```
Go

Use USADB
Go

Create Table USACustomers
(
    ID int primary key identity,
    Name nvarchar(50)
)
Go

Insert Into USACustomers Values ('Tom')
Insert Into USACustomers Values ('Mike')

Select * From USADB.dbo.USACustomers
```

The above query produces the following data

USADB.dbo.USACustomers	
ID	**Name**
1	Tom
2	Mike

3. India customers are stored in a table called IndiaCustomers in a database called IndiaDB.

```
Create Database USADB
Go

Use USADB
Go

Create Table USACustomers
(
    ID int primary key identity,
    Name nvarchar(50)
)
Go

Insert Into USACustomers Values ('Tom')
Insert Into USACustomers Values ('Mike')

Select * From USADB.dbo.USACustomers
```

- The above query produces the following data

IndiaDB.dbo.IndiaCustomers	
ID	**Name**
1	John
2	Ben

In both the tables, the ID column data type is integer. It is also the primary key column which ensures the ID column across every row is unique in that table. We also have turned on the identity property,

4. Now, we want to load the customers from both countries (India & USA) in to a single existing table **Customers**.

- First let's create the table. Use the following SQL script to create the table. ID column is the primary key of the table.

```
Create Table Customers
(
    ID int primary key,
    Name nvarchar(50)
)

Go
```

- Now execute the following script which selects the data from IndiaCustomers and USACustomers tables and inserts into Customers table

```
Insert Into Customers
Select * from IndiaDB.dbo.IndiaCustomers
Union All
Select * from USADB.dbo.USACustomers
```

We get the following error. This is because in both the tables, Identity column data type is integer. Integer is great for identity as long as you only want to maintain the uniqueness across just that one table. However, between IndiaCustomers and USACustomers tables, the ID coulumn values are not unique. So when we load the data into Customers table, we get "Violation of PRIMARY KEY constraint" error.

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

A GUID on the other hand is unique across tables, databases, and servers. A GUID is guaranteed to be globally unique. Let us see if we can solve the above problem using a GUID.

Create **USACustomers1** table and populate it with data. Notice ID column datatype is

uniqueidentifier. To auto-generate the GUID values we are using a Default constraint.

```
Use USADB
Go

Create Table USACustomers1
(
    ID uniqueidentifier primary key default NEWID(),
    Name nvarchar(50)
)
Go

Insert Into USACustomers1 Values (Default, 'Tom')
Insert Into USACustomers1 Values (Default, 'Mike')
```

Next, create **IndiaCustomers1** table and populate it with data.

```
Use IndiaDB
Go

Create Table IndiaCustomers1
(
    ID uniqueidentifier primary key default NEWID(),
    Name nvarchar(50)
)
Go

Insert Into IndiaCustomers1 Values (Default, 'John')
Insert Into IndiaCustomers1 Values (Default, 'Ben')
```

Select data from both the tables (USACustomers1 & IndiaCustomers1). Notice the ID column values. They are unique across both the tables.

```
Select * From IndiaDB.dbo.IndiaCustomers1
UNION ALL
Select * From USADB.dbo.USACustomers1
```

ID	Name
F2BD596B-55A8-4A1B-8BD1-6D42F7E3A1A0	Ben
FE90F0E9-7C5F-4540-BA57-80FD7B7EF6E9	John
BD51334C-AFE1-4DE4-BEB9-B5BDA2CDCC8A	Tom
D483AF60-8F5D-4BE6-8E24-E648C0744A48	Mike

Now, we want to load the customers from **USACustomers1** and **IndiaCustomers1** tables in to a single existing table called Customers1. Let us first create Customers1 table. The ID column in Customers1 table is uniqueidentifier.

```
Create Table Customers1
(
```

```
    ID uniqueidentifier primary key,
    Name nvarchar(50)
)
Go
```

Finally, execute the following insert script. Notice the script executes successfully without any errors and the data is loaded into Customers1 table.

```
Insert Into Customers1
Select * from IndiaDB.dbo.IndiaCustomers1
Union All
Select * from USADB.dbo.USACustomers1
```

- The main advantage of using a GUID is that it is unique across tables, databases and servers. It is extremely useful if you're consolidating records from multiple SQL Servers into a single table.

  The main disadvantage of using a GUID as a key is that it is 16 bytes in size. It is one of the largest datatypes in SQL Server. An integer on the other hand is 4 bytes,

  An Index built on a GUID is larger and slower than an index built on integer column. In addition a GUID is hard to read compared to int.
- **How to check GUID is null or empty in SQL Server**
- **Suggested Videos**
  Part 134 - Sequence object in SQL Server 2012
  Part 135 - Difference between sequence and identity in SQL Server
  Part 136 - Guid in SQL Server
- In this video we will discuss **how to check if a GUID is null or empty**

  How to check if a GUID is NULL : Checking if a GUID is null is straight forward in SQL Server. Just use IS NULL keywords as shown below.

```
Declare @MyGuid Uniqueidentifier

If(@MyGuid IS NULL)
Begin
    Print 'Guid is null'
End
Else
Begin
    Print 'Guid is not null'
End
```

- In the above example, since @MyGuid is just declared and not initialised, it prints the message "Guid is null"

  Now let's say, if a GUID variable is NULL, then we want to initialise that GUID variable with a new GUID value. If it's not NULL, then we want to retain it's value. One way to do this is by using an IF condition as shown below.

```
Declare @MyGuid UniqueIdentifier
```

```
If(@MyGuid IS NULL)
Begin
    Set @MyGuid = NEWID()
End

Select @MyGuid
```

We can achieve exactly the same thing by using ISNULL() function. The advantage of using ISNULL() function is that, it reduces the amount of code we have to write.

```
Declare @MyGuid UniqueIdentifier
Select ISNULL(@MyGuid, NewID())
```

How to check if a GUID is EMPTY : Before understanding how to check if a GUID is empty, let's understand what is an empty GUID. An empty GUID is a GUID with all ZEROS as shown below.
00000000-0000-0000-0000-000000000000

How to create this empty GUID. Do we have to type all the ZERO's and Hyphens. The answe is NO. We do not have to type them manually. Instead use one of the following SELECT query's to create an empty GUID. I prefer to use the second SELECT statement as it has only one CAST

```
SELECT CAST(CAST(0 AS BINARY) AS UNIQUEIDENTIFIER)
OR
SELECT CAST(0x0 AS UNIQUEIDENTIFIER)
```

To check if a GUID is an empty GUID, you have 2 options
Option 1: You can compare it to an Empty GUID value as shown below

```
Declare @MyGuid UniqueIdentifier
Set @MyGuid = '00000000-0000-0000-0000-000000000000'

If(@MyGuid = '00000000-0000-0000-0000-000000000000')
Begin
    Print 'Guid is Empty'
End
Else
Begin
    Print 'Guid is not Empty'
End
```

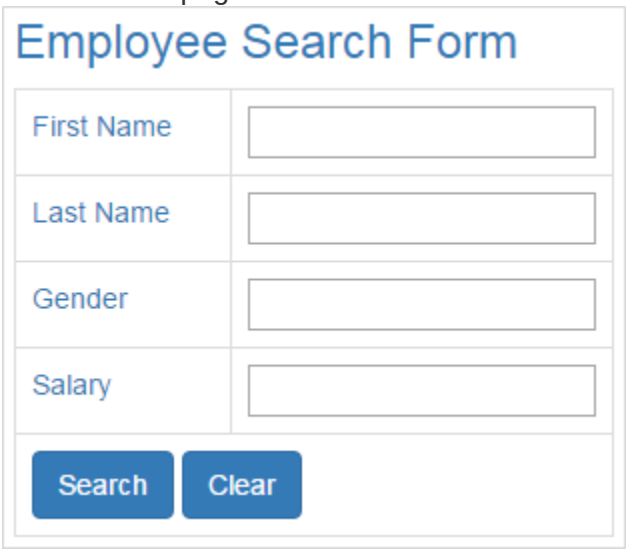
Option 2: You can also compare it to a return value of the CAST method

```
Declare @MyGuid UniqueIdentifier
Set @MyGuid = '00000000-0000-0000-0000-000000000000'

If(@MyGuid = Cast(0x0 as Uniqueidentifier))
Begin
```

- Print 'Guid is Empty'
- End
- Else
- Begin
- Print 'Guid is not Empty'
- End
- ## Dynamic SQL in SQL Server
- **Suggested Videos**
  Part 135 - Difference between sequence and identity in SQL Server
  Part 136 - Guid in SQL Server
  Part 137 - How to check guid is null or empty in SQL Server

  In this video we will discuss
  1. What is Dynamic SQL
  2. Simple example of using Dynamic SQL

  What is Dynamic SQL
  Dynamic SQL is a SQL built from strings at runtime.
- **Simple example of using Dynamic SQL :** Let's say we want to implement **"Employee Search"** web page as shown below.



Depending on the serach fields the end user provides, we want to search the following Employees table.

ID	FirstName	LastName	Gender	Salary
2	Steve	Pound	Male	45000
3	Ben	Hoskins	Male	70000
4	Philip	Hastings	Male	45000
5	Mary	Lambeth	Female	30000
6	Valarie	Vikings	Female	35000
7	John	Stanmore	Male	80000

Here is the SQL Script to create Employees table and populate it with data

```
Create table Employees
(
    ID int primary key identity,
    FirstName nvarchar(50),
    LastName nvarchar(50),
    Gender nvarchar(50),
    Salary int
)
Go

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000)
Insert into Employees values ('Steve', 'Pound', 'Male', 45000)
Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000)
Insert into Employees values ('Philip', 'Hastings', 'Male', 45000)
Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000)
Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000)
Insert into Employees values ('John', 'Stanmore', 'Male', 80000)
Go
```

One way to achieve this is by implementing a stored procedure as shown below that this page would call.

```
Create Procedure spSearchEmployees
@FirstName nvarchar(100),
@LastName nvarchar(100),
@Gender nvarchar(50),
@Salary int
As
Begin

    Select * from Employees where
    (FirstName = @FirstName OR @FirstName IS NULL) AND
    (LastName  = @LastName  OR @LastName  IS NULL) AND
    (Gender    = @Gender    OR @Gender    IS NULL) AND
    (Salary    = @Salary    OR @Salary    IS NULL)
End
Go
```

The stored procedure in this case is not very complicated as we have only 4 search filters. What if there are 20 or more such filters. This stored procedure can get complex. To make things worse what if we want to specify conditions like AND, OR etc between these search filters. The stored procedure can get extremely large, complicated and difficult to maintain. One way to reduce the complexity is by using dynamic SQL as show below. Depending on for which search filters the user has provided the values on the "Search Page", we build the WHERE clause dynamically at runtime, which can reduce complexity.

However, you might hear arguments that dynamic sql is bad both in-terms of security and performance. This is true if the dynamic sql is not properly implemented. From a security standpoint, it may open doors for SQL injection attack and from a performance standpoint, the

cached query plans may not be reused. If properly implemented, we will not have these problems with dynamic sql. In our upcoming videos, we will discuss good and bad dynamic sql implementations.

For now let's implement a simple example that makes use of dynamic sql. In the example below we are assuming the user has supplied values only for FirstName and LastName search fields. To execute the dynamicl sql we are using system stored procedure sp_executesql.

sp_executesql takes two pre-defined parameters and any number of user-defined parameters.

@statement - The is the first parameter which is mandatory, and contains the SQL statements to execute

@params - This is the second parameter and is optional. This is used to declare parameters specified in @statement

The rest of the parameters are the parameters that you declared in @params, and you pass them as you pass parameters to a stored procedure

```
Declare @sql nvarchar(1000)
Declare @params nvarchar(1000)

Set @sql = 'Select * from Employees where FirstName=@FirstName and
LastName=@LastName'
Set @params = '@FirstName nvarchar(100), @LastName nvarchar(100)'

Execute sp_executesql @sql, @params, @FirstName='Ben',@LastName='Hoskins'
```

This is just the introduction to dynamic SQL. If a few things are unclear at the moment, don't worry. In our upcoming videos we will discuss the following

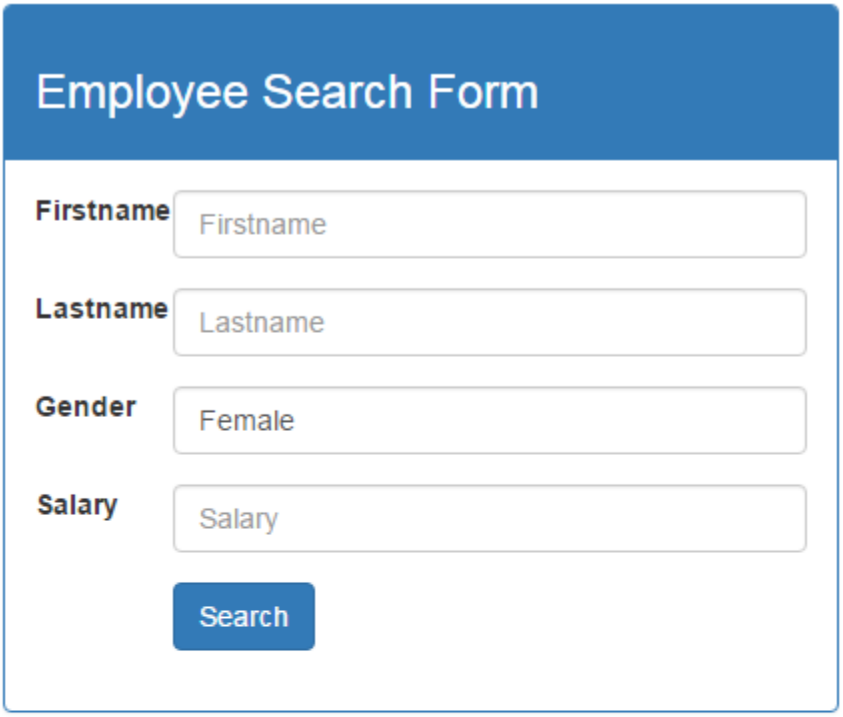
1. Implementing a real world "Search Web Page" with and without dynamic SQL
2. Performance and Security implications of dynamic sql. Along the way we will also discuss good and bad dynamic sql implementations.
3. Different options available for executing dynamic sql and their implications
4. Using dynamic sql in stored procedures and it's implications

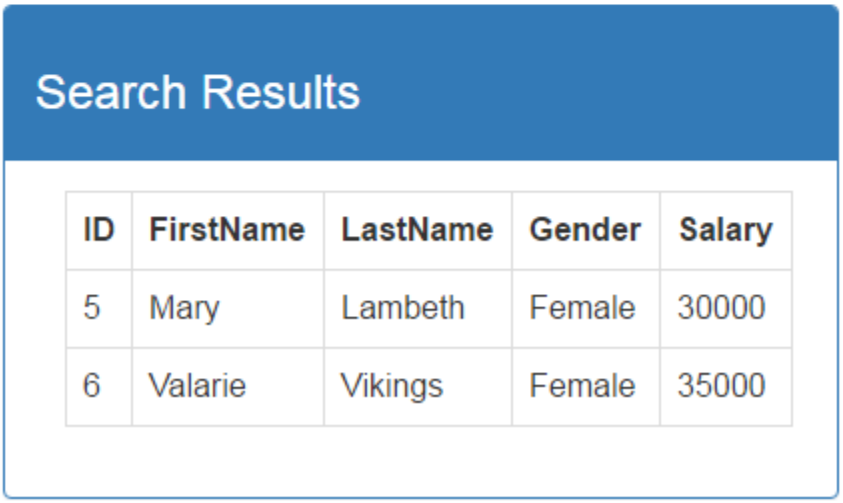
- **Once we discuss all the above, you will understand**
  1. The flexibility dynamic sql provides
  2. Advantages and disadvantages of dynamic sql
  3. When and when not to use dynamic sql
- **Implement search web page using ASP.NET and Stored Procedure**
- **Suggested Videos**
  Part 136 - Guid in SQL Server
  Part 137 - How to check guid is null or empty in SQL Server
  Part 138 - Dynamic SQL in SQL Server

  In this video we will discuss **implementing a search web page using ASP.NET and Stored Procedure**. This is continuation to Part 138. Please watch Part 138 from SQL

Server Tutorial before proceeding.

The search page looks as shown below.





- **Step 1 :** Modify the "spSearchEmployees" stored procedure to include NULL as the default value for the parameters. The advantage of specifying default value for the parameters is that the ASP.NET page need not pass those parameters when calling the stored procedures if the user did not specify any values for the corresponding search fields on the Search Page.

- Alter Procedure spSearchEmployees
- @FirstName nvarchar(100) = NULL,
- @LastName nvarchar(100) = NULL,
- @Gender nvarchar(50) = NULL,
- @Salary int = NULL

```
As
Begin

    Select * from Employees where
    (FirstName = @FirstName OR @FirstName IS NULL) AND
    (LastName  = @LastName  OR @LastName  IS NULL) AND
    (Gender    = @Gender    OR @Gender    IS NULL) AND
    (Salary    = @Salary    OR @Salary    IS NULL)
End
Go
```

Step 2 : Create a new empty ASP.NET Web Forms application. Name it "DynamicSQLDemo".

Step 3 : Add the connection string to your database in web.config

```
<add name="connectionStr"
     connectionString="server=.;database=EmployeeDB;integrated security=true"/>
```

Step 4 : Add a WebForm to the project. Name it **"SearchPageWithoutDynamicSQL.aspx"**

Step 5 : Copy and paste the following HTML on the ASPX page. Notice we are using Bootstrap to style the page. If you are new to Bootstrap, please check out our Bootstrap tutorial for beginners playlist.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Employee Search</title>
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
        type="text/css" />
</head>
<body style="padding-top: 10px">
    <div class="col-xs-8 col-xs-offset-2">
        <form id="form1" runat="server" class="form-horizontal">
            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3>Employee Search Form</h3>
                </div>
                <div class="panel-body">
                    <div class="form-group">
                        <label for="inputFirstname" class="control-label col-xs-2">
                            Firstname
                        </label>
                        <div class="col-xs-10">
                            <input type="text" runat="server" class="form-control"
                                id="inputFirstname" placeholder="Firstname" />
                        </div>
                    </div>

                    <div class="form-group">
```

```
                    <label for="inputLastname" class="control-label col-xs-2">
                        Lastname
                    </label>
                    <div class="col-xs-10">
                        <input type="text" runat="server" class="form-control"
                            id="inputLastname" placeholder="Lastname" />
                    </div>
                </div>

                <div class="form-group">
                    <label for="inputGender" class="control-label col-xs-2">
                        Gender
                    </label>
                    <div class="col-xs-10">
                        <input type="text" runat="server" class="form-control"
                            id="inputGender" placeholder="Gender" />
                    </div>
                </div>

                <div class="form-group">
                    <label for="inputSalary" class="control-label col-xs-2">
                        Salary
                    </label>
                    <div class="col-xs-10">
                        <input type="number" runat="server" class="form-control"
                            id="inputSalary" placeholder="Salary" />
                    </div>
                </div>
                <div class="form-group">
                    <div class="col-xs-10 col-xs-offset-2">
                        <asp:Button ID="btnSearch" runat="server" Text="Search"
                            CssClass="btn btn-primary" OnClick="btnSearch_Click" />
                    </div>
                </div>
            </div>
        </div>

        <div class="panel panel-primary">
            <div class="panel-heading">
                <h3>Search Results</h3>
            </div>
            <div class="panel-body">
                <div class="col-xs-10">
                    <asp:GridView CssClass="table table-bordered"
                        ID="gvSearchResults" runat="server">
                    </asp:GridView>
                </div>
            </div>
        </div>
    </form>
</div>
```

```
</body>
</html>
```

Step 6 : Copy and paste the following code in the code-behind page. Notice we are using the stored procedure "spSearchEmployees". We are not using any dynamic SQL in this example. In our next video, we will discuss implementing the same "Search Page" using dynamic sql and understand the difference between using dynamic sql and stored procedure.

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace DynamicSQLDemo
{
    public partial class SearchPageWithoutDynamicSQL : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {}

        protected void btnSearch_Click(object sender, EventArgs e)
        {
            string connectionStr = ConfigurationManager
                .ConnectionStrings["connectionStr"].ConnectionString;
            using(SqlConnection con = new SqlConnection(connectionStr))
            {
                SqlCommand cmd = new SqlCommand();
                cmd.Connection = con;
                cmd.CommandText = "spSearchEmployees";
                cmd.CommandType = CommandType.StoredProcedure;

                if(inputFirstname.Value.Trim() != "")
                {
                    SqlParameter param = new SqlParameter
                        ("@FirstName", inputFirstname.Value);
                    cmd.Parameters.Add(param);
                }

                if (inputLastname.Value.Trim() != "")
                {
                    SqlParameter param = new SqlParameter
                        ("@LastName", inputLastname.Value);
                    cmd.Parameters.Add(param);
                }

                if (inputGender.Value.Trim() != "")
                {
                    SqlParameter param = new SqlParameter
                        ("@Gender", inputGender.Value);
                    cmd.Parameters.Add(param);
```

```
            }

            if (inputSalary.Value.Trim() != "")
            {
                SqlParameter param = new SqlParameter
                    ("@Salary", inputSalary.Value);
                cmd.Parameters.Add(param);
            }

            con.Open();
            SqlDataReader rdr = cmd.ExecuteReader();
            gvSearchResults.DataSource = rdr;
            gvSearchResults.DataBind();
        }
    }
  }
}
```

- **Implement search web page using ASP.NET and Dynamic SQL**
- **Suggested Videos**
  Part 137 - How to check guid is null or empty in SQL Server
  Part 138 - Dynamic SQL in SQL Server
  Part 139 - Implement search web page using ASP.NET and Stored Procedure

  In this video we will discuss implementing a search web page using ASP.NET and Dynamic SQL. This is continuation to Part 139. Please watch Part 139 from SQL Server Tutorial before proceeding.
- **Step 1 :** Add a WebForm to the web project. Name it "SearchPageWithDynamicSQL.aspx"

Step 2 : Copy and paste the following HTML on the ASPX page. Notice we are using Bootstrap to style the page. If you are new to Bootstrap, please check out our Bootstrap tutorial for beginners playlist.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Employee Search</title>
  <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    type="text/css" />
</head>
<body style="padding-top: 10px">
  <div class="col-xs-8 col-xs-offset-2">
    <form id="form1" runat="server" class="form-horizontal">
      <div class="panel panel-primary">
        <div class="panel-heading">
          <h3>Employee Search Form</h3>
        </div>
        <div class="panel-body">
          <div class="form-group">
            <label for="inputFirstname" class="control-label col-xs-2">
```

```
                Firstname
            </label>
            <div class="col-xs-10">
                <input type="text" runat="server" class="form-control"
                    id="inputFirstname" placeholder="Firstname" />
            </div>
        </div>

        <div class="form-group">
            <label for="inputLastname" class="control-label col-xs-2">
                Lastname
            </label>
            <div class="col-xs-10">
                <input type="text" runat="server" class="form-control"
                    id="inputLastname" placeholder="Lastname" />
            </div>
        </div>

        <div class="form-group">
            <label for="inputGender" class="control-label col-xs-2">
                Gender
            </label>
            <div class="col-xs-10">
                <input type="text" runat="server" class="form-control"
                    id="inputGender" placeholder="Gender" />
            </div>
        </div>

        <div class="form-group">
            <label for="inputSalary" class="control-label col-xs-2">
                Salary
            </label>
            <div class="col-xs-10">
                <input type="number" runat="server" class="form-control"
                    id="inputSalary" placeholder="Salary" />
            </div>
        </div>
        <div class="form-group">
            <div class="col-xs-10 col-xs-offset-2">
                <asp:Button ID="btnSearch" runat="server" Text="Search"
                    CssClass="btn btn-primary" OnClick="btnSearch_Click" />
            </div>
        </div>
    </div>
</div>

<div class="panel panel-primary">
    <div class="panel-heading">
        <h3>Search Results</h3>
    </div>
    <div class="panel-body">
```

```
                <div class="col-xs-10">
                    <asp:GridView CssClass="table table-bordered"
                        ID="gvSearchResults" runat="server">
                    </asp:GridView>
                </div>
            </div>
        </div>
    </form>
  </div>
</body>
</html>
```

Step 3 : Copy and paste the following code in the code-behind page. Notice we are using dynamic sql instead of the stored procedure "spSearchEmployees".

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using System.Text;

namespace DynamicSQLDemo
{
  public partial class SearchPageWithDynamicSQL : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    { }

    protected void btnSearch_Click(object sender, EventArgs e)
    {
      string strConnection = ConfigurationManager
        .ConnectionStrings["connectionStr"].ConnectionString;

      using (SqlConnection con = new SqlConnection(strConnection))
      {
        SqlCommand cmd = new SqlCommand();
        cmd.Connection = con;

        StringBuilder sbCommand = new
          StringBuilder("Select * from Employees where 1 = 1");

        if (inputFirstname.Value.Trim() != "")
        {
          sbCommand.Append(" AND FirstName=@FirstName");
          SqlParameter param = new
            SqlParameter("@FirstName", inputFirstname.Value);
          cmd.Parameters.Add(param);
        }

        if (inputLastname.Value.Trim() != "")
        {
```

```
            sbCommand.Append(" AND LastName=@LastName");
            SqlParameter param = new
                SqlParameter("@LastName", inputLastname.Value);
            cmd.Parameters.Add(param);
        }

        if (inputGender.Value.Trim() != "")
        {
            sbCommand.Append(" AND Gender=@Gender");
            SqlParameter param = new
                SqlParameter("@Gender", inputGender.Value);
            cmd.Parameters.Add(param);
        }

        if (inputSalary.Value.Trim() != "")
        {
            sbCommand.Append(" AND Salary=@Salary");
            SqlParameter param = new
                SqlParameter("@Salary", inputSalary.Value);
            cmd.Parameters.Add(param);
        }

        cmd.CommandText = sbCommand.ToString();
        cmd.CommandType = CommandType.Text;

        con.Open();
        SqlDataReader rdr = cmd.ExecuteReader();
        gvSearchResults.DataSource = rdr;
        gvSearchResults.DataBind();
    }
  }
 }
}
```

- At this point, run the application and SQL profiler. To run SQL profiler
1. Open SQL Server Management Studio
2. Click on "Tools" and select "SQL Server Profiler"
3. Click the "Connect" button to connect to local SQl Server instance
- 4. Leave the "Defaults" on "Trace Properties" window and click on "Run" button
5. We now have the SQL Profiler running and in action

On the "Search Page" set "Gender" filter to Male and click the "Search" button. Notice we get all the Male employees as expected. Also in the SQL Server profiler you can see the Dynamic SQL statement is executed using system stored procedure sp_executesql.

```
exec sp_executesql N'Select * from Employees where 1 = 1 AND
Gender=@Gender',N'@Gender nvarchar(4)',@Gender=N'Male'
```

In our next video, we will discuss the differences between using Dynamic SQL and Stored Procedures

Prevent sql injection with dynamic sql

- **Suggested Videos**
  Part 138 - Dynamic SQL in SQL Server
  Part 139 - Implement search web page using ASP.NET and Stored Procedure
  Part 140 - Implement search web page using ASP.NET and Dynamic SQL

  In this video we will discuss, how to prevent SQL injection when using dynamic SQL. This is continuation to Part 140. Please watch Part 140 from SQL Server Tutorial before proceeding.

  In Part 140, we have implemented "Search Page" using dynamic SQL. Since we have used parameters to build our dynamic SQL statements, it is not prone to SQL Injection attack. This is an example of good dynamic SQL implementation.

  I have seen lot of software developers, not just the beginners but even experienced developers, buidling their dynamic sql queries by concatenating strings instead of using parameters without realizing that they are opening the doors for SQL Injection.
- Here is an example of bad dynamic SQL that is prone to SQL Injection

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using System.Text;

namespace DynamicSQLDemo
{
    public partial class SearchPageWithDynamicSQL : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }

        protected void btnSearch_Click(object sender, EventArgs e)
        {
            string strConnection = ConfigurationManager
                .ConnectionStrings["connectionStr"].ConnectionString;

            using (SqlConnection con = new SqlConnection(strConnection))
            {
                SqlCommand cmd = new SqlCommand();
                cmd.Connection = con;

                StringBuilder sbCommand = new
                    StringBuilder("Select * from Employees where 1 = 1");

                if (inputFirstname.Value.Trim() != "")
                {
                    sbCommand.Append(" AND FirstName = '" +
                        inputFirstname.Value + "'");
                }
```

```
                if (inputLastname.Value.Trim() != "")
                {
                    sbCommand.Append(" AND LastName = '" +
                        inputLastname.Value + "'");
                }

                if (inputGender.Value.Trim() != "")
                {
                    sbCommand.Append(" AND Gender = '" +
                        inputGender.Value + "'");
                }

                if (inputSalary.Value.Trim() != "")
                {
                    sbCommand.Append(" AND Salary = " + inputSalary.Value);
                }

                cmd.CommandText = sbCommand.ToString();
                cmd.CommandType = CommandType.Text;

                con.Open();
                SqlDataReader rdr = cmd.ExecuteReader();
                gvSearchResults.DataSource = rdr;
                gvSearchResults.DataBind();
            }
        }
    }
}
```

Since we are concatenating the user input values to build the dynamic sql statement, the end user can very easily inject sql. Imagine, what happens for example, if the user enters the following in the "Firstname" textbox.

```
' Drop database SalesDB --
```

With the above SQL injected into the "Firstname" textbox, if you click the "Search" button, the following is the query which is sent to SQL server. This will drop SalesDB database.

```
Select * from Employees where 1 = 1 AND FirstName = '' Drop database SalesDB --'
```

On the other hand, if you use parameters to build your dynamic SQL statements, SQL Injection is not an issue. The following input in the "Firstname" textbox, would not drop the SalesDB database.

```
' Drop database SalesDB --
```

The text the user has provided in the "Firstname" textbox is treated as the value for @Firstname parameter. The following is the query that is generated and executed.

```
exec sp_executesql N'Select * from Employees where 1 = 1 AND
FirstName=@FirstName',N'@FirstName nvarchar(26)',@FirstName=N''' Drop database
SalesDB --'
```

We don't have this problem of sql injection if we are using stored procedures. "SearchPageWithoutDynamicSQL.aspx" is using the stored procedure "spSearchEmployees" instead of dynamic SQL. The same input in the "Firstname" texbox on this page, would generate the following. Notice, whatever text we typed in the "Firstname" textbox is treated as the value for @FirstName parameter.

```
exec spSearchEmployees @FirstName=N''' Drop database SalesDB --'
```

An important point to keep in mind here is that if you have dynamic SQL in your stored procedure, and you are concatenating strings in that stored procedure to build your dynamic sql statements instead of using parameters, it is still prone to SQL injection. If this is not clear at the moment don't worry, we will discuss an example of this in out next video.

So in summary, while dynamic sql provides great flexibility when implementing complicated logic with lot of permutations and combinations, if not properly implemented it may open doors for sql injection. Always use parameters to build dynamic sql statements, instead of concatenating user input values.

Another benefit of using parameters to build dynamic sql statements is that it allows cached query plans to be reused, which greatly increases the performance. We will discuss an example of this in our upcoming videos.

Dynamic SQL in Stored Procedure

Suggested Videos

Part 139 - Implement search web page using ASP.NET and Stored Procedure
Part 140 - Implement search web page using ASP.NET and Dynamic SQL
Part 141 - Prevent sql injection with dynamic sql

In this video we will discuss, using dynamic sql in a stored procedure and it's implications from sql injection perspective. We will discuss performance implications of using dynamic sql in a stored procedure in a later video.
Consider the following stored procedure "spSearchEmployees". We implemented this procedure in Part 139 of SQL Server tutorial. This stored procedure does not have any dynamic sql in it. It is all static sql and is immune to sql injection.

```
Create Procedure spSearchEmployees
@FirstName nvarchar(100) = NULL,
@LastName nvarchar(100) = NULL,
@Gender nvarchar(50) = NULL,
```

```
@Salary int = NULL
As
Begin

    Select * from Employees where
    (FirstName = @FirstName OR @FirstName IS NULL) AND
    (LastName  = @LastName  OR @LastName  IS NULL) AND
    (Gender    = @Gender    OR @Gender    IS NULL) AND
    (Salary    = @Salary    OR @Salary    IS NULL)
End
Go
```

Whether you are creating your dynamic sql queries in a client application like ASP.NET web application or in a stored procedure, you should never ever concatenate user input values. Instead you should be using parameters.

Notice in the following example, we are creating dynamic sql queries by concatenating parameter values, instead of using parameterized queries. This stored procedure is prone to SQL injection. Let's prove this by creating a "Search Page" that calls this procedure.

```
Create Procedure spSearchEmployeesBadDynamicSQL
@FirstName nvarchar(100) = NULL,
@LastName nvarchar(100) = NULL,
@Gender nvarchar(50) = NULL,
@Salary int = NULL
As
Begin

    Declare @sql nvarchar(max)

    Set @sql = 'Select * from Employees where 1 = 1'

    if(@FirstName is not null)
        Set @sql = @sql + ' and FirstName=''' + @FirstName + ''''
    if(@LastName is not null)
        Set @sql = @sql + ' and LastName=''' + @LastName + ''''
    if(@Gender is not null)
        Set @sql = @sql + ' and Gender=''' + @Gender + ''''
    if(@Salary is not null)
        Set @sql = @sql + ' and Salary=''' + @Salary + ''''

    Execute sp_executesql @sql
End
Go
```

Add a Web Page to the project that we have been working with in our previous video. Name it "DynamicSQLInStoredProcedure.aspx". Copy and paste the following HTML on the page.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Employee Search</title>
```

```
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
        type="text/css" />
</head>
<body style="padding-top: 10px">
    <div class="col-xs-8 col-xs-offset-2">
        <form id="form1" runat="server" class="form-horizontal">
            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3>Employee Search Form</h3>
                </div>
                <div class="panel-body">
                    <div class="form-group">
                        <label for="inputFirstname" class="control-label col-xs-2">
                            Firstname
                        </label>
                        <div class="col-xs-10">
                            <input type="text" runat="server" class="form-control"
                                id="inputFirstname" placeholder="Firstname" />
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="inputLastname" class="control-label col-xs-2">
                            Lastname
                        </label>
                        <div class="col-xs-10">
                            <input type="text" runat="server" class="form-control"
                                id="inputLastname" placeholder="Lastname" />
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="inputGender" class="control-label col-xs-2">
                            Gender
                        </label>
                        <div class="col-xs-10">
                            <input type="text" runat="server" class="form-control"
                                id="inputGender" placeholder="Gender" />
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="inputSalary" class="control-label col-xs-2">
                            Salary
                        </label>
                        <div class="col-xs-10">
                            <input type="number" runat="server" class="form-control"
                                id="inputSalary" placeholder="Salary" />
                        </div>
                    </div>
```

```
                <div class="form-group">
                    <div class="col-xs-10 col-xs-offset-2">
                        <asp:Button ID="btnSearch" runat="server" Text="Search"
                            CssClass="btn btn-primary" OnClick="btnSearch_Click" />
                    </div>
                </div>
            </div>

            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3>Search Results</h3>
                </div>
                <div class="panel-body">
                    <div class="col-xs-10">
                        <asp:GridView CssClass="table table-bordered"
                            ID="gvSearchResults" runat="server">
                        </asp:GridView>
                    </div>
                </div>
            </div>
        </form>
    </div>
</body>
</html>
```

Copy and paste the following code in the code-behind page.

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace DynamicSQLDemo
{
    public partial class DynamicSQLInStoredProcedure : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {}

        protected void btnSearch_Click(object sender, EventArgs e)
        {
            string connectionStr = ConfigurationManager
                .ConnectionStrings["connectionStr"].ConnectionString;
            using (SqlConnection con = new SqlConnection(connectionStr))
            {
                SqlCommand cmd = new SqlCommand();
                cmd.Connection = con;
                cmd.CommandText = "spSearchEmployeesGoodDynamicSQL";
                cmd.CommandType = CommandType.StoredProcedure;
```

```
            if (inputFirstname.Value.Trim() != "")
            {
                SqlParameter param = new SqlParameter("@FirstName",
                    inputFirstname.Value);
                cmd.Parameters.Add(param);
            }

            if (inputLastname.Value.Trim() != "")
            {
                SqlParameter param = new SqlParameter("@LastName",
                    inputLastname.Value);
                cmd.Parameters.Add(param);
            }

            if (inputGender.Value.Trim() != "")
            {
                SqlParameter param = new SqlParameter("@Gender",
                    inputGender.Value);
                cmd.Parameters.Add(param);
            }

            if (inputSalary.Value.Trim() != "")
            {
                SqlParameter param = new SqlParameter("@Salary",
                    inputSalary.Value);
                cmd.Parameters.Add(param);
            }

            con.Open();
            SqlDataReader rdr = cmd.ExecuteReader();
            gvSearchResults.DataSource = rdr;
            gvSearchResults.DataBind();
        }
    }
  }
}
```

At this point, run the application and type the following text in the "Firsname" text and click "Search" button. Notice "SalesDB" database is dropped. Our application is prone to SQL injection as we have implemented dynamic sql in our stored procedure by concatenating strings instead of using parameters.

```
' Drop database SalesDB --
```

In the following stored procedure we have implemented dynamic sql by using parameters, so this is not prone to sql injecttion. This is an example for good dynamic sql implementation.

```
Create Procedure spSearchEmployeesGoodDynamicSQL
@FirstName nvarchar(100) = NULL,
@LastName nvarchar(100) = NULL,
@Gender nvarchar(50) = NULL,
@Salary int = NULL
```

```
As
Begin

    Declare @sql nvarchar(max)
    Declare @sqlParams nvarchar(max)

    Set @sql = 'Select * from Employees where 1 = 1'

    if(@FirstName is not null)
        Set @sql = @sql + ' and FirstName=@FN'
    if(@LastName is not null)
        Set @sql = @sql + ' and LastName=@LN'
    if(@Gender is not null)
        Set @sql = @sql + ' and Gender=@Gen'
    if(@Salary is not null)
        Set @sql = @sql + ' and Salary=@Sal'

    Execute sp_executesql @sql,
    N'@FN nvarchar(50), @LN nvarchar(50), @Gen nvarchar(50), @sal int',
    @FN=@FirstName, @LN=@LastName, @Gen=@Gender, @Sal=@Salary
End
Go
```

On the code-behind page, use stored procedure spSearchEmployeesGoodDynamicSQL instead of spSearchEmployeesBadDynamicSQL. We do not have to change any other code. At this point run the application one more time and type the following text in the "Firstname" textbox and click the "Search" button.

```
' Drop database SalesDB --
```

Notice "SalesDB" database is not dropped, So in this case our application is not susceptible to SQL injection attack.

Summary : Whether you are creating dynamic sql in a client application (like a web application) or in a stored procedure always use parameters instead of concatenating strings. Using parameters to create dynamic sql statements prevents sql injection.

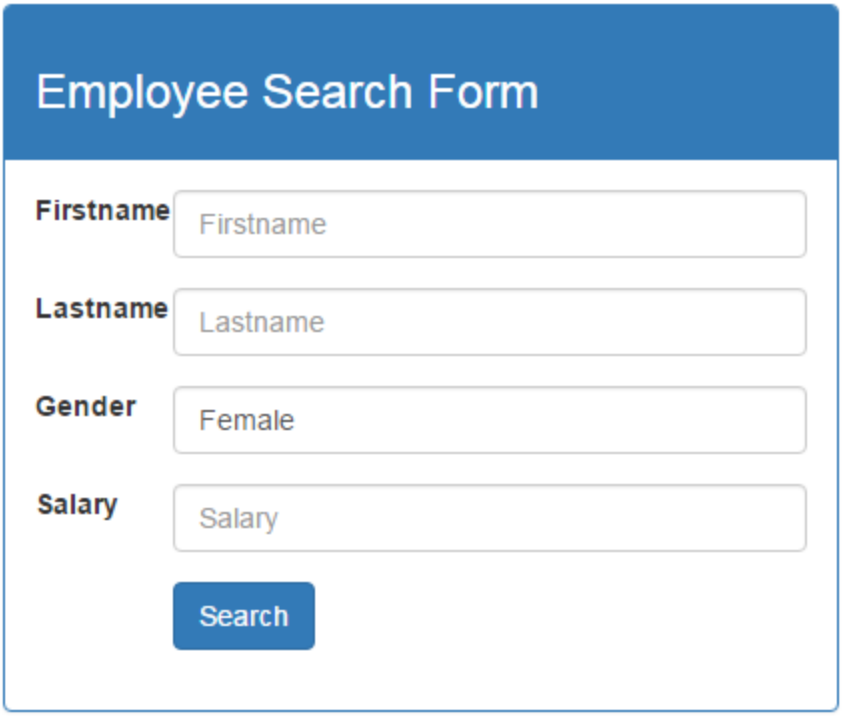
Implement search web page using ASP.NET and Stored Procedure

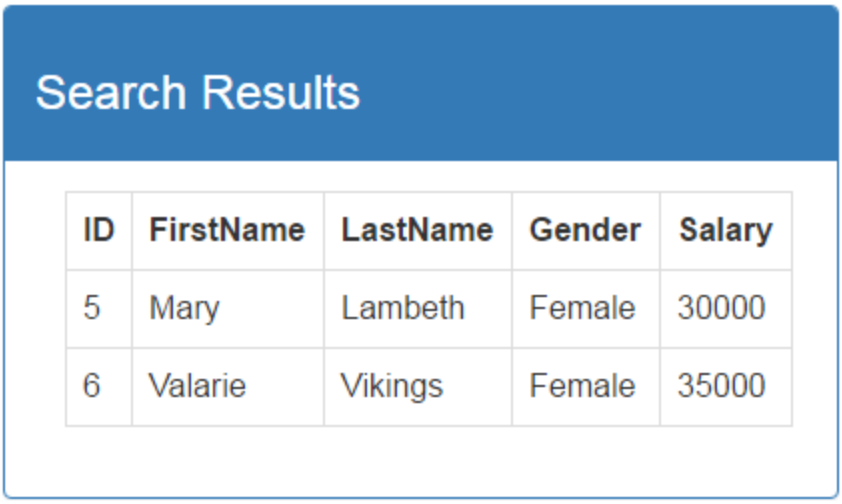
Suggested Videos

Part 136 - Guid in SQL Server
Part 137 - How to check guid is null or empty in SQL Server
Part 138 - Dynamic SQL in SQL Server

In this video we will discuss **implementing a search web page using ASP.NET and Stored Procedure**. This is continuation to Part 138. Please watch Part 138 from SQL Server Tutorial before proceeding.

The search page looks as shown below.





Step 1 : Modify the "spSearchEmployees" stored procedure to include NULL as the default value for the parameters. The advantage of specifying default value for the parameters is that the ASP.NET page need not pass those parameters when calling the stored procedures if the user did not specify any values for the corresponding search fields on the Search Page.

```
Alter Procedure spSearchEmployees
@FirstName nvarchar(100) = NULL,
@LastName nvarchar(100) = NULL,
@Gender nvarchar(50) = NULL,
@Salary int = NULL
As
```

```
Begin

    Select * from Employees where
    (FirstName = @FirstName OR @FirstName IS NULL) AND
    (LastName  = @LastName  OR @LastName  IS NULL) AND
    (Gender    = @Gender    OR @Gender    IS NULL) AND
    (Salary    = @Salary    OR @Salary    IS NULL)
End
Go
```

Step 2 : Create a new empty ASP.NET Web Forms application. Name it "DynamicSQLDemo".

Step 3 : Add the connection string to your database in web.config

```
<add name="connectionStr"
     connectionString="server=.;database=EmployeeDB;integrated security=true"/>
```

Step 4 : Add a WebForm to the project. Name it **"SearchPageWithoutDynamicSQL.aspx"**

Step 5 : Copy and paste the following HTML on the ASPX page. Notice we are using Bootstrap to style the page. If you are new to Bootstrap, please check out our Bootstrap tutorial for beginners playlist.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Employee Search</title>
   <link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
      type="text/css" />
</head>
<body style="padding-top: 10px">
   <div class="col-xs-8 col-xs-offset-2">
      <form id="form1" runat="server" class="form-horizontal">
         <div class="panel panel-primary">
            <div class="panel-heading">
               <h3>Employee Search Form</h3>
            </div>
            <div class="panel-body">
               <div class="form-group">
                  <label for="inputFirstname" class="control-label col-xs-2">
                     Firstname
                  </label>
                  <div class="col-xs-10">
                     <input type="text" runat="server" class="form-control"
                        id="inputFirstname" placeholder="Firstname" />
                  </div>
               </div>

               <div class="form-group">
                  <label for="inputLastname" class="control-label col-xs-2">
                     Lastname
                  </label>
                  <div class="col-xs-10">
```

```
                    <input type="text" runat="server" class="form-control"
                        id="inputLastname" placeholder="Lastname" />
                </div>
            </div>

            <div class="form-group">
                <label for="inputGender" class="control-label col-xs-2">
                    Gender
                </label>
                <div class="col-xs-10">
                    <input type="text" runat="server" class="form-control"
                        id="inputGender" placeholder="Gender" />
                </div>
            </div>

            <div class="form-group">
                <label for="inputSalary" class="control-label col-xs-2">
                    Salary
                </label>
                <div class="col-xs-10">
                    <input type="number" runat="server" class="form-control"
                        id="inputSalary" placeholder="Salary" />
                </div>
            </div>
            <div class="form-group">
                <div class="col-xs-10 col-xs-offset-2">
                    <asp:Button ID="btnSearch" runat="server" Text="Search"
                        CssClass="btn btn-primary" OnClick="btnSearch_Click" />
                </div>
            </div>
        </div>
    </div>

    <div class="panel panel-primary">
        <div class="panel-heading">
            <h3>Search Results</h3>
        </div>
        <div class="panel-body">
            <div class="col-xs-10">
                <asp:GridView CssClass="table table-bordered"
                    ID="gvSearchResults" runat="server">
                </asp:GridView>
            </div>
        </div>
    </div>
  </form>
 </div>
</body>
</html>
```

Step 6 : Copy and paste the following code in the code-behind page. Notice we are using the

stored procedure "spSearchEmployees". We are not using any dynamic SQL in this example. In our next video, we will discuss implementing the same "Search Page" using dynamic sql and understand the difference between using dynamic sql and stored procedure.

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace DynamicSQLDemo
{
    public partial class SearchPageWithoutDynamicSQL : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {}

        protected void btnSearch_Click(object sender, EventArgs e)
        {
            string connectionStr = ConfigurationManager
                .ConnectionStrings["connectionStr"].ConnectionString;
            using(SqlConnection con = new SqlConnection(connectionStr))
            {
                SqlCommand cmd = new SqlCommand();
                cmd.Connection = con;
                cmd.CommandText = "spSearchEmployees";
                cmd.CommandType = CommandType.StoredProcedure;

                if(inputFirstname.Value.Trim() != "")
                {
                    SqlParameter param = new SqlParameter
                        ("@FirstName", inputFirstname.Value);
                    cmd.Parameters.Add(param);
                }

                if (inputLastname.Value.Trim() != "")
                {
                    SqlParameter param = new SqlParameter
                        ("@LastName", inputLastname.Value);
                    cmd.Parameters.Add(param);
                }

                if (inputGender.Value.Trim() != "")
                {
                    SqlParameter param = new SqlParameter
                        ("@Gender", inputGender.Value);
                    cmd.Parameters.Add(param);
                }

                if (inputSalary.Value.Trim() != "")
                {
                    SqlParameter param = new SqlParameter
```

```
                ("@Salary", inputSalary.Value);
            cmd.Parameters.Add(param);
        }

        con.Open();
        SqlDataReader rdr = cmd.ExecuteReader();
        gvSearchResults.DataSource = rdr;
        gvSearchResults.DataBind();
    }
  }
}
```

Implement search web page using ASP.NET and Dynamic SQL

Suggested Videos

Part 137 - How to check guid is null or empty in SQL Server
Part 138 - Dynamic SQL in SQL Server
Part 139 - Implement search web page using ASP.NET and Stored Procedure

In this video we will discuss implementing a search web page using ASP.NET and Dynamic SQL. This is continuation to Part 139. Please watch Part 139 from SQL Server Tutorial before proceeding.

Step 1 : Add a WebForm to the web project. Name it "SearchPageWithDynamicSQL.aspx"

Step 2 : Copy and paste the following HTML on the ASPX page. Notice we are using Bootstrap to style the page. If you are new to Bootstrap, please check out our Bootstrap tutorial for beginners playlist.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Employee Search</title>
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
        type="text/css" />
</head>
<body style="padding-top: 10px">
    <div class="col-xs-8 col-xs-offset-2">
        <form id="form1" runat="server" class="form-horizontal">
            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3>Employee Search Form</h3>
                </div>
                <div class="panel-body">
                    <div class="form-group">
                        <label for="inputFirstname" class="control-label col-xs-2">
                            Firstname
                        </label>
                        <div class="col-xs-10">
                            <input type="text" runat="server" class="form-control"
                                id="inputFirstname" placeholder="Firstname" />
                        </div>
```

```
        </div>

        <div class="form-group">
            <label for="inputLastname" class="control-label col-xs-2">
                Lastname
            </label>
            <div class="col-xs-10">
                <input type="text" runat="server" class="form-control"
                    id="inputLastname" placeholder="Lastname" />
            </div>
        </div>

        <div class="form-group">
            <label for="inputGender" class="control-label col-xs-2">
                Gender
            </label>
            <div class="col-xs-10">
                <input type="text" runat="server" class="form-control"
                    id="inputGender" placeholder="Gender" />
            </div>
        </div>

        <div class="form-group">
            <label for="inputSalary" class="control-label col-xs-2">
                Salary
            </label>
            <div class="col-xs-10">
                <input type="number" runat="server" class="form-control"
                    id="inputSalary" placeholder="Salary" />
            </div>
        </div>
        <div class="form-group">
            <div class="col-xs-10 col-xs-offset-2">
                <asp:Button ID="btnSearch" runat="server" Text="Search"
                    CssClass="btn btn-primary" OnClick="btnSearch_Click" />
            </div>
        </div>
    </div>
</div>

<div class="panel panel-primary">
    <div class="panel-heading">
        <h3>Search Results</h3>
    </div>
    <div class="panel-body">
        <div class="col-xs-10">
            <asp:GridView CssClass="table table-bordered"
                ID="gvSearchResults" runat="server">
            </asp:GridView>
        </div>
    </div>
```

```
        </div>
      </form>
   </div>
</body>
</html>
```

Step 3 : Copy and paste the following code in the code-behind page. Notice we are using dynamic sql instead of the stored procedure "spSearchEmployees".

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using System.Text;

namespace DynamicSQLDemo
{
   public partial class SearchPageWithDynamicSQL : System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      { }

      protected void btnSearch_Click(object sender, EventArgs e)
      {
         string strConnection = ConfigurationManager
            .ConnectionStrings["connectionStr"].ConnectionString;

         using (SqlConnection con = new SqlConnection(strConnection))
         {
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = con;

            StringBuilder sbCommand = new
               StringBuilder("Select * from Employees where 1 = 1");

            if (inputFirstname.Value.Trim() != "")
            {
               sbCommand.Append(" AND FirstName=@FirstName");
               SqlParameter param = new
                  SqlParameter("@FirstName", inputFirstname.Value);
               cmd.Parameters.Add(param);
            }

            if (inputLastname.Value.Trim() != "")
            {
               sbCommand.Append(" AND LastName=@LastName");
               SqlParameter param = new
                  SqlParameter("@LastName", inputLastname.Value);
               cmd.Parameters.Add(param);
            }
```

```
            if (inputGender.Value.Trim() != "")
            {
                sbCommand.Append(" AND Gender=@Gender");
                SqlParameter param = new
                    SqlParameter("@Gender", inputGender.Value);
                cmd.Parameters.Add(param);
            }

            if (inputSalary.Value.Trim() != "")
            {
                sbCommand.Append(" AND Salary=@Salary");
                SqlParameter param = new
                    SqlParameter("@Salary", inputSalary.Value);
                cmd.Parameters.Add(param);
            }

            cmd.CommandText = sbCommand.ToString();
            cmd.CommandType = CommandType.Text;

            con.Open();
            SqlDataReader rdr = cmd.ExecuteReader();
            gvSearchResults.DataSource = rdr;
            gvSearchResults.DataBind();
        }
    }
  }
}
```

At this point, run the application and SQL profiler. To run SQL profiler
1. Open SQL Server Management Studio
2. Click on "Tools" and select "SQL Server Profiler"
3. Click the "Connect" button to connect to local SQl Server instance

5. We now have the SQL Profiler running and in action

On the "Search Page" set "Gender" filter to Male and click the "Search" button. Notice we get all the Male employees as expected. Also in the SQL Server profiler you can see the Dynamic SQL statement is executed using system stored procedure sp_executesql.

```
exec sp_executesql N'Select * from Employees where 1 = 1 AND
Gender=@Gender',N'@Gender nvarchar(4)',@Gender=N'Male'
```

In our next video, we will discuss the differences between using Dynamic SQL and Stored Procedures

Prevent sql injection with dynamic sql

Suggested Videos

Part 138 - Dynamic SQL in SQL Server
Part 139 - Implement search web page using ASP.NET and Stored Procedure
Part 140 - Implement search web page using ASP.NET and Dynamic SQL

In this video we will discuss, how to prevent SQL injection when using dynamic SQL. This is continuation to Part 140. Please watch Part 140 from SQL Server Tutorial before proceeding.

In Part 140, we have implemented "Search Page" using dynamic SQL. Since we have used parameters to build our dynamic SQL statements, it is not prone to SQL Injection attack. This is an example of good dynamic SQL implementation.
I have seen lot of software developers, not just the beginners but even experienced developers, buidling their dynamic sql queries by concatenating strings instead of using parameters without realizing that they are opening the doors for SQL Injection.

Here is an example of bad dynamic SQL that is prone to SQL Injection

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using System.Text;

namespace DynamicSQLDemo
{
    public partial class SearchPageWithDynamicSQL : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }

        protected void btnSearch_Click(object sender, EventArgs e)
        {
            string strConnection = ConfigurationManager
                .ConnectionStrings["connectionStr"].ConnectionString;

            using (SqlConnection con = new SqlConnection(strConnection))
            {
                SqlCommand cmd = new SqlCommand();
                cmd.Connection = con;

                StringBuilder sbCommand = new
                    StringBuilder("Select * from Employees where 1 = 1");

                if (inputFirstname.Value.Trim() != "")
                {
                    sbCommand.Append(" AND FirstName = '" +
                        inputFirstname.Value + "'");
                }


                if (inputLastname.Value.Trim() != "")
                {
                    sbCommand.Append(" AND LastName = '" +
                        inputLastname.Value + "'");
                }
```

```
            if (inputGender.Value.Trim() != "")
            {
                sbCommand.Append(" AND Gender = '" +
                    inputGender.Value + "'");
            }

            if (inputSalary.Value.Trim() != "")
            {
                sbCommand.Append(" AND Salary = " + inputSalary.Value);
            }

            cmd.CommandText = sbCommand.ToString();
            cmd.CommandType = CommandType.Text;

            con.Open();
            SqlDataReader rdr = cmd.ExecuteReader();
            gvSearchResults.DataSource = rdr;
            gvSearchResults.DataBind();
        }
    }
  }
}
```

Since we are concatenating the user input values to build the dynamic sql statement, the end user can very easily inject sql. Imagine, what happens for example, if the user enters the following in the "Firstname" textbox.

```
' Drop database SalesDB --
```

With the above SQL injected into the "Firstname" textbox, if you click the "Search" button, the following is the query which is sent to SQL server. This will drop SalesDB database.

```
Select * from Employees where 1 = 1 AND FirstName = '' Drop database SalesDB --'
```

On the other hand, if you use parameters to build your dynamic SQL statements, SQL Injection is not an issue. The following input in the "Firstname" textbox, would not drop the SalesDB database.

```
' Drop database SalesDB --
```

The text the user has provided in the "Firstname" textbox is treated as the value for @Firstname parameter. The following is the query that is generated and executed.

```
exec sp_executesql N'Select * from Employees where 1 = 1 AND
FirstName=@FirstName',N'@FirstName nvarchar(26)',@FirstName=N''' Drop database
SalesDB --'
```

We don't have this problem of sql injection if we are using stored procedures. "SearchPageWithoutDynamicSQL.aspx" is using the stored procedure "spSearchEmployees" instead of dynamic SQL. The same input in the "Firstname" texbox on this page, would generate the following. Notice, whatever text we typed in the "Firstname" textbox is treated as

the value for @FirstName parameter.

```
exec spSearchEmployees @FirstName=N'" Drop database SalesDB --'
```

An important point to keep in mind here is that if you have dynamic SQL in your stored procedure, and you are concatenating strings in that stored procedure to build your dynamic sql statements instead of using parameters, it is still prone to SQL injection. If this is not clear at the moment don't worry, we will discuss an example of this in out next video.

So in summary, while dynamic sql provides great flexibility when implementing complicated logic with lot of permutations and combinations, if not properly implemented it may open doors for sql injection. Always use parameters to build dynamic sql statements, instead of concatenating user input values.

Another benefit of using parameters to build dynamic sql statements is that it allows cached query plans to be reused, which greatly increases the performance. We will discuss an example of this in our upcoming videos.

Dynamic SQL in Stored Procedure

Suggested Videos

Part 139 - Implement search web page using ASP.NET and Stored Procedure
Part 140 - Implement search web page using ASP.NET and Dynamic SQL
Part 141 - Prevent sql injection with dynamic sql

In this video we will discuss, using dynamic sql in a stored procedure and it's implications from sql injection perspective. We will discuss performance implications of using dynamic sql in a stored procedure in a later video.

Consider the following stored procedure "spSearchEmployees". We implemented this procedure in Part 139 of SQL Server tutorial. This stored procedure does not have any dynamic sql in it. It is all static sql and is immune to sql injection.

```
Create Procedure spSearchEmployees
@FirstName nvarchar(100) = NULL,
@LastName nvarchar(100) = NULL,
@Gender nvarchar(50) = NULL,
@Salary int = NULL
As
Begin

    Select * from Employees where
    (FirstName = @FirstName OR @FirstName IS NULL) AND
    (LastName  = @LastName  OR @LastName  IS NULL) AND
    (Gender    = @Gender    OR @Gender    IS NULL) AND
    (Salary    = @Salary    OR @Salary    IS NULL)
End
Go
```

Whether you are creating your dynamic sql queries in a client application like ASP.NET web application or in a stored procedure, you should never ever concatenate user input values. Instead you should be using parameters.

Notice in the following example, we are creating dynamic sql queries by concatenating parameter values, instead of using parameterized queries. This stored procedure is prone to SQL injection. Let's prove this by creating a "Search Page" that calls this procedure.

```
Create Procedure spSearchEmployeesBadDynamicSQL
@FirstName nvarchar(100) = NULL,
@LastName nvarchar(100) = NULL,
@Gender nvarchar(50) = NULL,
@Salary int = NULL
As
Begin

    Declare @sql nvarchar(max)

    Set @sql = 'Select * from Employees where 1 = 1'

    if(@FirstName is not null)
        Set @sql = @sql + ' and FirstName=''' + @FirstName + ''''
    if(@LastName is not null)
        Set @sql = @sql + ' and LastName=''' + @LastName + ''''
    if(@Gender is not null)
        Set @sql = @sql + ' and Gender=''' + @Gender + ''''
    if(@Salary is not null)
        Set @sql = @sql + ' and Salary=''' + @Salary + ''''

    Execute sp_executesql @sql
End
Go
```

Add a Web Page to the project that we have been working with in our previous video. Name it "DynamicSQLInStoredProcedure.aspx". Copy and paste the following HTML on the page.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Employee Search</title>
   <link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
      type="text/css" />
</head>
<body style="padding-top: 10px">
   <div class="col-xs-8 col-xs-offset-2">
      <form id="form1" runat="server" class="form-horizontal">
         <div class="panel panel-primary">
            <div class="panel-heading">
               <h3>Employee Search Form</h3>
            </div>
            <div class="panel-body">
```

```
        <div class="form-group">
            <label for="inputFirstname" class="control-label col-xs-2">
                Firstname
            </label>
            <div class="col-xs-10">
                <input type="text" runat="server" class="form-control"
                    id="inputFirstname" placeholder="Firstname" />
            </div>
        </div>

        <div class="form-group">
            <label for="inputLastname" class="control-label col-xs-2">
                Lastname
            </label>
            <div class="col-xs-10">
                <input type="text" runat="server" class="form-control"
                    id="inputLastname" placeholder="Lastname" />
            </div>
        </div>

        <div class="form-group">
            <label for="inputGender" class="control-label col-xs-2">
                Gender
            </label>
            <div class="col-xs-10">
                <input type="text" runat="server" class="form-control"
                    id="inputGender" placeholder="Gender" />
            </div>
        </div>

        <div class="form-group">
            <label for="inputSalary" class="control-label col-xs-2">
                Salary
            </label>
            <div class="col-xs-10">
                <input type="number" runat="server" class="form-control"
                    id="inputSalary" placeholder="Salary" />
            </div>
        </div>
        <div class="form-group">
            <div class="col-xs-10 col-xs-offset-2">
                <asp:Button ID="btnSearch" runat="server" Text="Search"
                    CssClass="btn btn-primary" OnClick="btnSearch_Click" />
            </div>
        </div>
    </div>
</div>

<div class="panel panel-primary">
    <div class="panel-heading">
        <h3>Search Results</h3>
```

```
            </div>
            <div class="panel-body">
               <div class="col-xs-10">
                  <asp:GridView CssClass="table table-bordered"
                     ID="gvSearchResults" runat="server">
                  </asp:GridView>
               </div>
            </div>
         </div>
      </form>
   </div>
</body>
</html>
```

Copy and paste the following code in the code-behind page.

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace DynamicSQLDemo
{
   public partial class DynamicSQLInStoredProcedure : System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      {}

      protected void btnSearch_Click(object sender, EventArgs e)
      {
         string connectionStr = ConfigurationManager
            .ConnectionStrings["connectionStr"].ConnectionString;
         using (SqlConnection con = new SqlConnection(connectionStr))
         {
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = con;
            cmd.CommandText = "spSearchEmployeesGoodDynamicSQL";
            cmd.CommandType = CommandType.StoredProcedure;

            if (inputFirstname.Value.Trim() != "")
            {
               SqlParameter param = new SqlParameter("@FirstName",
                  inputFirstname.Value);
               cmd.Parameters.Add(param);
            }

            if (inputLastname.Value.Trim() != "")
            {
               SqlParameter param = new SqlParameter("@LastName",
                  inputLastname.Value);
               cmd.Parameters.Add(param);
```

```
            }

            if (inputGender.Value.Trim() != "")
            {
                SqlParameter param = new SqlParameter("@Gender",
                    inputGender.Value);
                cmd.Parameters.Add(param);
            }

            if (inputSalary.Value.Trim() != "")
            {
                SqlParameter param = new SqlParameter("@Salary",
                    inputSalary.Value);
                cmd.Parameters.Add(param);
            }

            con.Open();
            SqlDataReader rdr = cmd.ExecuteReader();
            gvSearchResults.DataSource = rdr;
            gvSearchResults.DataBind();
        }
    }
  }
}
```

At this point, run the application and type the following text in the "Firsname" text and click "Search" button. Notice "SalesDB" database is dropped. Our application is prone to SQL injection as we have implemented dynamic sql in our stored procedure by concatenating strings instead of using parameters.

```
' Drop database SalesDB --
```

In the following stored procedure we have implemented dynamic sql by using parameters, so this is not prone to sql injecttion. This is an example for good dynamic sql implementation.

```
Create Procedure spSearchEmployeesGoodDynamicSQL
@FirstName nvarchar(100) = NULL,
@LastName nvarchar(100) = NULL,
@Gender nvarchar(50) = NULL,
@Salary int = NULL
As
Begin

    Declare @sql nvarchar(max)
    Declare @sqlParams nvarchar(max)

    Set @sql = 'Select * from Employees where 1 = 1'

    if(@FirstName is not null)
        Set @sql = @sql + ' and FirstName=@FN'
    if(@LastName is not null)
        Set @sql = @sql + ' and LastName=@LN'
```

```
    if(@Gender is not null)
        Set @sql = @sql + ' and Gender=@Gen'
    if(@Salary is not null)
        Set @sql = @sql + ' and Salary=@Sal'

    Execute sp_executesql @sql,
    N'@FN nvarchar(50), @LN nvarchar(50), @Gen nvarchar(50), @sal int',
    @FN=@FirstName, @LN=@LastName, @Gen=@Gender, @Sal=@Salary
End
Go
```

On the code-behind page, use stored procedure spSearchEmployeesGoodDynamicSQL instead of spSearchEmployeesBadDynamicSQL. We do not have to change any other code. At this point run the application one more time and type the following text in the "Firstname" textbox and click the "Search" button.
' Drop database SalesDB --

Notice "SalesDB" database is not dropped, So in this case our application is not susceptible to SQL injection attack.

Summary : Whether you are creating dynamic sql in a client application (like a web application) or in a stored procedure always use parameters instead of concatenating strings. Using parameters to create dynamic sql statements prevents sql injection.

Sql server query plan cache

Suggested Videos

Part 140 - Implement search web page using ASP.NET and Dynamic SQL
Part 141 - Prevent sql injection with dynamic sql
Part 142 - Dynamic SQL in Stored Procedure

In this video we will discuss

1. What happens when a query is issued to SQL Server
2. How to check what is in SQL Server plan cache
3. Things to consider to promote query plan reusability

What happens when a query is issued to SQL Server

In SQl Server, every query requires a query plan before it is executed. When you run a query the first time, the query gets compiled and a query plan is generated. This query plan is then saved in sql server query plan cache.

Next time when we run the same query, the cached query plan is reused. This means sql server does not have to create the plan again for that same query. So reusing a query plan can increase the performance.

How long the query plan stays in the plan cache depends on how often the plan is reused besides other factors. The more often the plan is reused the longer it stays in the plan cache.

How to check what is in SQL Server plan cache

To see what is in SQL Server plan cache we will make use of the following 3 dynamic management views and functions provided by sql server

1. sys.dm_exec_cached_plans
2. sys.dm_exec_sql_text
3. sys.dm_exec_query_plan

Please note we discussed CROSS APPLY in detail in Part 91 of SQL Server tutorial.

Use the following query to see what is in the plan cache

```
SELECT cp.usecounts, cp.cacheobjtype, cp.objtype, st.text, qp.query_plan
FROM sys.dm_exec_cached_plans AS cp
CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
CROSS APPLY sys.dm_exec_query_plan(plan_handle) AS qp
ORDER BY cp.usecounts DESC
```

As you can see we have sorted the result set by **usecounts** column in descending order, so we can see the most frequently reused query plans on the top. The output of the above query from my computer is shown below.

usecounts	objtype	text	query_plan
8	Adhoc	Select * From...	<ShowPlanX...
1	Prepared	(@1 varchar(...	<ShowPlanX...
1	Adhoc	SELECT cp....	<ShowPlanX...

The following table explains what each column in the resultset contains

Column	Description
usecounts	Number of times the plan is reused
objtype	Specifies the type of object
text	Text of the SQL query
query_plan	Query execution plan in XML format

To remove all elements from the plan cache use the following command

```
DBCC FREEPROCCACHE
```

In older versions of SQL Server up to SQL Server 6.5 only stored procedure plans are cached. The query plans for Adhoc sql statements or dynamic sql statements are not cached, so they get compiled every time. With SQL Server 7, and later versions the query plans for Adhoc sql statements and dynamic sql statements are also cached.

Things to consider to promote query plan reusability

For example, when we execute the following query the first time. The query is compiled, a plan is created and put in the cache.

```
Select * From Employees Where FirstName = 'Mark'
```

When we execute the same query again, it looks up the plan cache, and if a plan is available, it reuses the existing plan instead of creating the plan again which can improve the performance

of the query. However, one important thing to keep in mind is that, the cache lookup is by a hash value computed from the query text. If the query text changes even slightly, sql server will not be able to reuse the existing plan.

For example, even if you include an extra space somewhere in the query or you change the case, the query text hash will not match, and sql server will not be able find the plan in cache and ends up compiling the query again and creating a new plan.

Another example : If you want the same query to find an employee whose FirstName is Steve instead of Mark. You would issue the following query

```
Select * From Employees Where FirstName = 'Steve'
```

Even in this case, since the query text has changed the hash will not match, and sql server will not be able find the plan in cache and ends up compiling the query again and creating a new plan.

This is why, it is very important to use parameterised queries for sql server to be able to reuse cached query plans. With parameterised queries, sql server will not treat parameter values as part of the query text. So when you change the parameters values, sql server can still reuse the cached query plan.

The following query uses parameters. So even if you change parameter values, the same query plan is reused.

```
Declare @FirstName nvarchar(50)
Set @FirstName = 'Steve'
Execute sp_executesql N'Select * from Employees where FirstName=@FN', N'@FN
nvarchar(50)', @FirstName
```

One important thing to keep in mind is that, when you have dynamic sql in a stored procedure, the query plan for the stored procedure does not include the dynamic SQL. The block of dynamic SQL has a query plan of its own.

Summary: Never ever concatenate user input values with strings to build dynamic sql statements. Always use parameterised queries which not only promotes cached query plans reuse but also prevent sql injection attacks.

exec vs sp_executesql in sql server

Suggested Videos

Part 141 - Prevent sql injection with dynamic sql
Part 142 - Dynamic SQL in Stored Procedure
Part 143 - Sql server query plan cache

In this video we will discuss the **difference between exec and sp_executesql**. This is continuation to Part 143. Please watch Part 143 from SQL Server tutorial before proceeding.

In SQL Servere we have 2 options to execute dynamic sql

1. Exec/Execute
2. sp_executesql

We discussed sp_executesql in detail in Part 138 of SQL Server tutorial. Please check out that video if you are new to sp_executesql.

If you do a quick search on the internet for the difference between exec and sp_executesql, you will see that many articles on the web states using exec over sp_executesql will have the following 2 problems
1. It open doors for sql injection attacks
2. Cached query plans may not be reused and leads to poor performance

This is generally true, but if you use QUOTENAME() function you can avoid sql injection attacks and with sql server auto-parameterisation capability the cached query plans can be reused so performance is also not an issue. Let's understand these with examples.

What is exec() in SQL Server
Exec() or Execute() function is used to execute dynamic sql and has only one parameter i.e the dynamic sql statement you want to execute.

As you can see in the example below, we are concatenating strings to build dynamic sql statements which open doors for sql injection.

```
Declare @FN nvarchar(50)
Set @FN = 'John'
Declare @sql nvarchar(max)
Set @sql = 'Select * from Employees where FirstName = ''' +  @FN + ''''
Exec(@sql)
```

If we set @FN parameter to something like below, it drops SalesDB database

```
Declare @FN nvarchar(50)
Set @FN = ''' Drop Database SalesDB --'''
Declare @sql nvarchar(max)
Set @sql = 'Select * from Employees where FirstName = ''' + @FN + ''''
Exec(@sql)
```

However, we can prevent SQL injection using the QUOTENAME() function as shown below.

```
Declare @FN nvarchar(50)
Set @FN = ''' Drop Database SalesDB --'''
Declare @sql nvarchar(max)
Set @sql = 'Select * from Employees where FirstName = ' + QUOTENAME(@FN,'''')
--Print @sql
Exec(@sql)
```

Notice with the quotename function we are using a single quote as a delimiter. With the use of this function if there is a single quote in the user input it is doubled.

For example, if we set @FN='John', notice the string 'John' is wrapped in single quotes

```
Declare @FN nvarchar(50)
Set @FN = 'John'
Declare @sql nvarchar(max)
Set @sql = 'Select * from Employees where FirstName = ' + QUOTENAME(@FN,'''')
Print @sql
```

When the above query is executed the following is the query printed

```
Select * from Employees where FirstName = 'John'
```

Along the same lines, if we try to inject sql, QUOTENAME() function wraps all that input in another pair of single quotes treating it as a value for the FirstName column and prevents SQL injection.

With sql server auto-parameterisation capability the cached query plans can be reused. SQL Server can detect parameter values and create parameterised queries on its own, even if you don't explicitly declare them. However, there are exceptions to this. Auto-parameterisation comes in 2 flavours - Simple and Forced. We will discuss auto-parameterisation in detail in a later video.

Execute the following DBCC command to remove all entries from the plan cache

```
DBCC FREEPROCCACHE
```

Execute the following query. Notice we have set @FN='Mary'

```
Declare @FN nvarchar(50)
Set @FN = 'Mary'
Declare @sql nvarchar(max)
Set @sql = 'Select * from Employees where FirstName = ' + QUOTENAME(@FN,'''')
Exec(@sql)
```

Execute the following query to retrieve what we have in the query plan cache

```
SELECT cp.usecounts, cp.cacheobjtype, cp.objtype, st.text, qp.query_plan
FROM sys.dm_exec_cached_plans AS cp
CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
CROSS APPLY sys.dm_exec_query_plan(plan_handle) AS qp
ORDER BY cp.usecounts DESC
```

Notice in the 3rd row, we have an auto-parameterised query and at the moment usecounts is 1.

usecounts	objtype	text
1	Adhoc	SELECT cp.usecounts, cp.objtype, st.text FROM sys.dm_exec_cached_pl...
1	Adhoc	Select * from Employees where FirstName = 'Mary'
1	Prepared	(@1 varchar(8000))SELECT * FROM [Employees] WHERE [FirstName]=@1

Now set @FN='Mark' and execute the same query. After the query is completed, retrieve the entries from the plan cache. Notice the usecounts for the auto-parameterised query is 2, suggesting that the same query plan is reused.

usecounts	objtype	text
2	Prepared	(@1 varchar(8000))SELECT * FROM [Employees] WHERE [FirstName]=@1
1	Adhoc	SELECT cp.usecounts, cp.objtype, st.text FROM sys.dm_exec_cached_p...
1	Adhoc	Select * from Employees where FirstName = 'Mark'
1	Adhoc	Select * from Employees where FirstName = 'Mary'

Along the same lines, if you change @FN='John' and execute the query, you will see that the usecounts is now 3 for the auto-parameterised query.

Summary

- If you use QUOTENAME() function, you can prevent sql injection while using Exec()
- Cached query plan reusability is also not an issue while using Exec(), as SQL server automatically parameterize queries.
- I personally prefer using sp_executesql over exec() as we can explicitly parameterise queries instead of relying on sql server auto-parameterisation feature or QUOTENAME() function. I use Exec() only in throw away scripts rather than in production code.

Dynamic sql table name variable

Suggested Videos

Part 142 - Dynamic SQL in Stored Procedure
Part 143 - Sql server query plan cache
Part 144 - exec vs sp_executesql in sql server

In this video we will discuss **how to pass table name dynamically for stored procedure in sql server**. This is one of the sql questions that is very commonly asked. Here is what we want to do.

I have a web page with a textbox as shown below. When I enter a table name in the textbox and when I click "Load Data" button, we want to retrieve data from that respective table and display it on the page.

For the purpose of this demo, we will use the following 2 tables.

Employees				
ID	FirstName	LastName	Gender	Salary
1	Mark	Hastings	Male	60000
2	Steve	Pound	Male	45000
3	Ben	Hoskins	Male	70000
4	Philip	Hastings	Male	45000
5	Mary	Lambeth	Female	30000
6	Valarie	Vikings	Female	35000
7	John	Stanmore	Male	80000

Countries	
Id	CountryName
1	USA
2	India
3	UK
4	Australia
5	Canada

SQL Script to create the required tables

```
Create table Countries
(
    Id int identity primary key,
    CountryName nvarchar(50)
)
Go

Insert into Countries values ('USA')
Insert into Countries values ('India')
Insert into Countries values ('UK')
Insert into Countries values ('Australia')
Insert into Countries values ('Canada')
Go

Create table Employees
(
    ID int primary key identity,
    FirstName nvarchar(50),
    LastName nvarchar(50),
    Gender nvarchar(50),
    Salary int
)
Go
```

```
Insert into Employees values ('Mark', 'Hastings', 'Male', 60000)
Insert into Employees values ('Steve', 'Pound', 'Male', 45000)
```

For the purpose of this demo, we will use the following 2 tables.

Employees				
ID	**FirstName**	**LastName**	**Gender**	**Salary**
1	Mark	Hastings	Male	60000
2	Steve	Pound	Male	45000
3	Ben	Hoskins	Male	70000
4	Philip	Hastings	Male	45000
5	Mary	Lambeth	Female	30000
6	Valarie	Vikings	Female	35000
7	John	Stanmore	Male	80000

Countries	
Id	**CountryName**
1	USA
2	India
3	UK
4	Australia
5	Canada

SQL Script to create the required tables

```
Create table Countries
(
    Id int identity primary key,
    CountryName nvarchar(50)
)
Go

Insert into Countries values ('USA')
Insert into Countries values ('India')
Insert into Countries values ('UK')
Insert into Countries values ('Australia')
Insert into Countries values ('Canada')
Go

Create table Employees
(
    ID int primary key identity,
    FirstName nvarchar(50),
    LastName nvarchar(50),
    Gender nvarchar(50),
```

```
    Salary int
)
Go

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000)
Insert into Employees values ('Steve', 'Pound', 'Male', 45000)
```

For the purpose of this demo, we will use the following 2 tables.

Employees				
ID	**FirstName**	**LastName**	**Gender**	**Salary**
1	Mark	Hastings	Male	60000
2	Steve	Pound	Male	45000
3	Ben	Hoskins	Male	70000
4	Philip	Hastings	Male	45000
5	Mary	Lambeth	Female	30000
6	Valarie	Vikings	Female	35000
7	John	Stanmore	Male	80000

Countries	
Id	**CountryName**
1	USA
2	India
3	UK
4	Australia
5	Canada

SQL Script to create the required tables

```
Create table Countries
(
    Id int identity primary key,
    CountryName nvarchar(50)
)
Go

Insert into Countries values ('USA')
Insert into Countries values ('India')
Insert into Countries values ('UK')
Insert into Countries values ('Australia')
Insert into Countries values ('Canada')
Go

Create table Employees
(
```

```
    ID int primary key identity,
    FirstName nvarchar(50),
    LastName nvarchar(50),
    Gender nvarchar(50),
    Salary int
)
Go

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000)
Insert into Employees values ('Steve', 'Pound', 'Male', 45000)
Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000)

Insert into Employees values ('Philip', 'Hastings', 'Male', 45000)
Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000)
Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000)
Insert into Employees values ('John', 'Stanmore', 'Male', 80000)
Go
```

Create the following stored procedure. Notice we are passing table name as a parameter to the stored procedure. In the body of the stored procedure we are concatenating strings to build our dynamic sql statement. In our previous videos we discussed that this open doors for SQL injection.

```
Create procedure spDynamicTableName
@TableName nvarchar(100)
As
Begin
    Declare @sql nvarchar(max)
    Set @sql = 'Select * from ' + @TableName
    Execute sp_executesql @sql
End
```

So the obvious question that comes to our mind is, why are we not creating parameterised sql statement instead. The answers is we can't. SQL Server does not allow table names and column names to be passed as parameters. Notice in the example below, we are creating a parameterised query with @TabName as a parameter. When we execute the following code, the procedure gets created successfully.

```
Create procedure spDynamicTableName1
@TableName nvarchar(100)
As
Begin
    Declare @sql nvarchar(max)
    Set @sql = 'Select * from @TabName'
    Execute sp_executesql @sql, N'@TabName nvarchar(100)',
    @TabName = @TableName
End
```

But when we try to execute it we get an error - Must declare the table variable "@TabName"

```
Execute spDynamicTableName1 N'Countries'
```

Add a Web Page to the project that we have been working with in our previous video. Name it "DynamicTableName.aspx". Copy and paste the following HTML on the page.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Employee Search</title>
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
        type="text/css" />
</head>
<body style="padding-top: 10px">
    <div class="col-xs-8 col-xs-offset-2">
        <form id="form1" runat="server" class="form-horizontal">
            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3>Table Lookup</h3>
                </div>
                <div class="panel-body">
                    <div class="form-group">
                        <label for="inputTableName" class="control-label col-xs-4">
                            Table Name
                        </label>
                        <div class="col-xs-8">
                            <input type="text" runat="server" class="form-control"
                                id="inputTableName" placeholder="Please enter table name" />
                        </div>
                    </div>
                    <div class="form-group">
                        <div class="col-xs-10 col-xs-offset-2">
                            <asp:Button ID="btnLoadData" runat="server" Text="Load Data"
                                CssClass="btn btn-primary" OnClick="btnLoadData_Click" />
  <asp:Label ID="lblError" runat="server" CssClass="text-danger">
                            </asp:Label>
                        </div>
                    </div>
                </div>
            </div>

            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3>Table Data</h3>
                </div>
                <div class="panel-body">
                    <div class="col-xs-10">
                        <asp:GridView CssClass="table table-bordered"
                            ID="gvTableData" runat="server">
                        </asp:GridView>
                    </div>
                </div>
            </div>
        </form>
```

```
    </div>
</body>
</html>
```

Copy and paste the following code in the code-behind page.

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace DynamicSQLDemo
{
    public partial class DynamicTableName : System.Web.UI.Page
    {
        protected void btnLoadData_Click(object sender, EventArgs e)
        {
            try
            {
                if (inputTableName.Value.Trim() != "")
                {
                    string strConnection = ConfigurationManager
                    .ConnectionStrings["connectionStr"].ConnectionString;

                    using (SqlConnection con = new SqlConnection(strConnection))
                    {
                        SqlCommand cmd = new SqlCommand();
                        cmd.Connection = con;
                        cmd.CommandText = "spDynamicTableName";
                        cmd.CommandType = CommandType.StoredProcedure;

                        SqlParameter param = new
                            SqlParameter("@TableName", inputTableName.Value);
                        param.SqlDbType = SqlDbType.NVarChar;
                        param.Size = 100;
                        cmd.Parameters.Add(param);

                        con.Open();
                        SqlDataReader rdr = cmd.ExecuteReader();
                        gvTableData.DataSource = rdr;
                        gvTableData.DataBind();
                    }
                }
                lblError.Text = "";
            }
            catch (Exception ex)
            {
                lblError.Text = ex.Message;
            }
        }
    }
```

}

At this point, run the application and type the following text in the "Table Name" textbox and click "Load Data" button. Notice "SalesDB" database is dropped. Our application is prone to SQL injection as we have implemented dynamic sql in our stored procedure by concatenating strings instead of using parameters.
Employees; Drop database SalesDB

One way to prevent SQL injection in this case is by using SQL Server built-in function - QUOTENAME(). We will discuss QUOTENAME() function in detail in our next video. For now understand that by default, this function wraps that string that is passed to it in a pair of brackets.
SELECT QUOTENAME('Employees') returns [Employees]

Modify the stored procedure to use QUOTENAME() function as shown below.

```
Alter procedure spDynamicTableName
@TableName nvarchar(100)
As
Begin
    Declare @sql nvarchar(max)
    Set @sql = 'Select * from ' + QUOTENAME(@TableName)
    Execute sp_executesql @sql
End
```

At this point, type the following text in the "Table Name" textbox and click "Load Data" button. Notice you will see a message - Invalid object name 'Employees; Drop database SalesDB'. Also "SalesDB" database is not dropped.
Employees; Drop database SalesDB

The entire text in "Table Name" textbox is wrapped in a pair of brackets by the QUOTENAME function and is treated as table name. Since we do have a table with the specified name, we get the error - Invalid object name.

Quotename function in SQL Server

Suggested Videos
Part 143 - Sql server query plan cache
Part 144 - exec vs sp_executesql in sql server
Part 145 - Dynamic sql table name variable

In this video we will discuss **Quotename function in SQL Server**. This is continuation to Part 145. Please watch Part 145 from SQL tutorial before proceeding.
This function is very useful when you want to quote object names. Let us understand the use of this function with an example.

We will use the following table for the examples in this demo

ID	FirstName	LastName	Gender
1	Mark	Hastings	Male
2	Steve	Pound	Male
3	Ben	Hoskins	Male
4	Philip	Hastings	Male
5	Mary	Lambeth	Female
6	Valarie	Vikings	Female
7	John	Stanmore	Male

SQL Script to create and populate the table with test data

```
Create table [USA Customers]
(
    ID int primary key identity,
    FirstName nvarchar(50),
    LastName nvarchar(50),
    Gender nvarchar(50)
)
Go

Insert into [USA Customers] values ('Mark', 'Hastings', 'Male')
Insert into [USA Customers] values ('Steve', 'Pound', 'Male')
Insert into [USA Customers] values ('Ben', 'Hoskins', 'Male')
Insert into [USA Customers] values ('Philip', 'Hastings', 'Male')
Insert into [USA Customers] values ('Mary', 'Lambeth', 'Female')
Insert into [USA Customers] values ('Valarie', 'Vikings', 'Female')
Insert into [USA Customers] values ('John', 'Stanmore', 'Male')
Go
```

Let us say, we are using dynamic SQL to build our SELECT query as shown below

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = 'USA Customers'
Set @sql = 'Select * from ' + @tableName
Execute sp_executesql @sql
```

When we execute the above script, we get the following error

```
Msg 208, Level 16, State 1, Line 1
Invalid object name 'USA'.
```

The query that our dynamic sql generates and executes is as shown below. To see the generate SQL statement, use Print @sql.

```
Select * from USA Customers
```

Since there is a space in the table name, it has to be wrapped in brackes as shown below

```
Select * from [USA Customers]
```

One way to fix this is by including the brackets in @tableName variable as shown below

```
Set @tableName = '[USA Customers]'
```

The other way to fix this is by including the brackets in @sql variable as shown below

```
Set @sql = 'Select * from [' + @tableName +']'
```

While both of the above methods give the result we want, it is extremely dangerous because it open doors for sql injection.

If we set the brackets in @tableName variable, sql can be injected as shown below and SalesDB database is dropped

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = '[USA Customers] Drop Database SalesDB'
Set @sql = 'Select * from ' + @tableName
Execute sp_executesql @sql
```

If we set the brackets in @sql variable, sql can be injected as shown below and SalesDB database is dropped

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = 'USA Customers] Drop Database SalesDB --'
Set @sql = 'Select * from [' + @tableName +']'
Execute sp_executesql @sql
```

So, the right way to do this is by using QUOTENAME() function as shown below.

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = 'USA Customers Drop Database SalesDB --'
Set @sql = 'Select * from ' + QUOTENAME(@tableName)
Execute sp_executesql @sql
```

When we execute the above script we get the following error. SalesDB database is not dropped. The reason we get this error is because we do not have a table with name - [USA Customers Drop Database SalesDB --]. To see the sql statement use PRINT @sql.

```
Invalid object name 'USA Customers Drop Database SalesDB --'.
```

If we set @tableName = 'USA Customers', the query executes successfully, without the threat of SQL injection.

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = 'USA Customers'
Set @sql = 'Select * from ' + QUOTENAME(@tableName)
Execute sp_executesql @sql
```

If you want to use sql server schema name "dbo" along with the table name, then you should

not use QUOTENAME function as shown below.

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = 'dbo.USA Customers'
Set @sql = 'Select * from ' + QUOTENAME(@tableName)
Execute sp_executesql @sql
```

The above query produces the following error
Invalid object name 'dbo.USA Customers'

Instead use QUOTENAME function as shown below

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = 'USA Customers'
Set @sql = 'Select * from ' + QUOTENAME('dbo') + '.' + QUOTENAME(@tableName)
Execute sp_executesql @sql
```

QUOTENAME() function

- Takes two parameters - the first is a string, and the second is a delimiter that you want SQL server to use to wrap the string in.
- The delimiter can be a left or right bracket ([]), a single quotation mark ('), or a double quotation mark (")
- The default for the second parameter is []

QUOTENAME() function examples

SELECT QUOTENAME('USA Customers','"") returns "USA Customers"

SELECT QUOTENAME('USA Customers','') returns 'USA Customers'

If we set @tableName = 'USA Customers', the query executes successfully, without the threat of SQL injection.

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = 'USA Customers'
Set @sql = 'Select * from ' + QUOTENAME(@tableName)
Execute sp_executesql @sql
```

If you want to use sql server schema name "dbo" along with the table name, then you should not use QUOTENAME function as shown below.

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = 'dbo.USA Customers'
Set @sql = 'Select * from ' + QUOTENAME(@tableName)
Execute sp_executesql @sql
```

The above query produces the following error
Invalid object name 'dbo.USA Customers'

Instead use QUOTENAME function as shown below

```
Declare @sql nvarchar(max)
Declare @tableName nvarchar(50)
Set @tableName = 'USA Customers'
Set @sql = 'Select * from ' + QUOTENAME('dbo') + '.' + QUOTENAME(@tableName)
Execute sp_executesql @sql
```

QUOTENAME() function

- Takes two parameters - the first is a string, and the second is a delimiter that you want SQL server to use to wrap the string in.
- The delimiter can be a left or right bracket ([]), a single quotation mark ('), or a double quotation mark (")
- The default for the second parameter is []

QUOTENAME() function examples

SELECT QUOTENAME('USA Customers','"") returns "USA Customers"

SELECT QUOTENAME('USA Customers','') returns 'USA Customers'

Dynamic SQL vs Stored Procedure

Suggested Videos
Part 144 - exec vs sp_executesql in sql server
Part 145 - Dynamic sql table name variable
Part 146 - Quotename function in SQL Server

In this video we will discuss the advantages and disadvantages of Dynamic SQL and Stored Procedures based on the following aspects

1. Separating database logic from business logic
2. Network traffic
3. SQL Injection Attacks
4. Cached query plans reuse
5. Maintenance
6. Implementing flexible logic

Separating database logic from business logic : Stored procedures allow us to keep database logic separate from business logic. The benefit of keeping database logic separate from the business logic is that, if there is an issue with the business logic you know you only have to check the application code. On the other hand if the issue is with the database logic, you will only have to check and modify the stored procedure.

Another added benefit here is that, if you change the stored procedure there is no need to compile your application code and deploy it. Just modify the stored procedure and you are done. You will loose this benefit if you are composing your dynamic sql statements in client

code, as you will have to change the application code if there is a bug. Changing the application code requires compilation, build and deployment.

Network traffic : **Stored procedures reduce network traffic as only the procedure name and a few parameters need to be sent over the network. With dynamic SQL, you will have to send your entire sql statement over the network. If the query is a complex one, with 50 to 60 lines, imagine the increased network traffic between the client application and the database server.**

SQL Injection Attacks : **Stored procedures prevent SQL injection attacks. In general, dynamic SQL open doors for SQL injection attacks if not careful. However, even with dynamic SQL, we can prevent SQL injection attacks by using parameterised queries. In some cases where you need to pass a table name or a column name as a parameter, it is not possible to use parameterised queries with dynamic sql. In such cases use QUOTENAME() function to prevent SQL injection attacks.**

Cached query plans reuse : **Stored procedures provide increased performance as cached query plans reusability increases. Even with dynamic SQL, if we use parameterised queries, cached query plan reusability increases, which in turn increases the performance. If you are not using parameterised queries, SQL Server auto-parameterisation feature can automatically detect parameter values and create parameterised queries which promotes query plan reusability and in turn performance.**

One important thing to keep in mind is that, from a performance standpoint OLTP queries benefit from cached query plan reuse. However, with OLAP systems as your data drifts and optimizer choices change, OLAP queries benefit from unique plans, so query plan reuse may not be desirable in this case for performance.

Maintenance : **With static SQL in a stored procedure, a syntax error is reported immediately so ease of writing is definitely one of the benefits of using a stored procedure. On the other hand if you have dynamic SQL in the stored procedure, and if there is a syntax error you wouldn't know it until you run it.**

Stored procedures with static SQL are also easy to maintain as you can use sp_depends procedure to check the dependencies on other SQL objects. For example, let's say you have a database with lot of tables, and you want to know if a certain table is referenced, because you are considering changing or dropping it. In this case using sp_depends TableName will let us know if it is referenced anywhere, so we can make changes without breaking anything. On the other hand if you are using dynamic sql in a stored procedure or sending it from a client, you loose this benefit.

Implementing flexible logic : **Sometimes, with stored procedures it is hard to implement flexible logic compared with dynamic SQL. For example, we want to implement a "Search" stored procedure with 20 or more filters. This stored procedure can get complex. To make things worse what if we want to specify conditions like AND, OR etc between these search filters. The stored procedure can get extremely large, complicated and difficult to maintain. One way to reduce the complexity is by using dynamic SQL. Depending on for which search filters the user has provided the values on the "Search Page", we build the WHERE clause dynamically at runtime, which can reduce complexity.**

Dynamic sql output parameter

Suggested Videos
Part 145 - Dynamic sql table name variable
Part 146 - Quotename function in SQL Server
Part 147 - Dynamic SQL vs Stored Procedure

In this video we will discuss, **how to use output parameters with dynamic sql**. Let us understand this with an example.

We will use the following **Employees** table in this demo.

ID	FirstName	LastName	Gender	Salary
1	Mark	Hastings	Male	60000
2	Steve	Pound	Male	45000
3	Ben	Hoskins	Male	70000
4	Philip	Hastings	Male	45000
5	Mary	Lambeth	Female	30000
6	Valarie	Vikings	Female	35000
7	John	Stanmore	Male	80000

SQL script to create Employees table

```
Create table Employees
(
    ID int primary key identity,
    FirstName nvarchar(50),
    LastName nvarchar(50),
    Gender nvarchar(50),
    Salary int
)
Go
Insert into Employees values ('Mark', 'Hastings', 'Male', 60000)
Insert into Employees values ('Steve', 'Pound', 'Male', 45000)
Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000)
Insert into Employees values ('Philip', 'Hastings', 'Male', 45000)
Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000)
Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000)
Insert into Employees values ('John', 'Stanmore', 'Male', 80000)
Go
```

We want to write a dynamic sql statement that returns total number of male of female employees. If the gender value is specified as "Male", then the query should return total male employees. Along the same lines, if the the value for gender is "Female", then we should get total number of female employees.

The following dynamic sql, will give us what we want. In this case, the query returns total number of "Male" employees. If you want the total number of female employees, simply set @gender='Female'.

```
Declare @sql nvarchar(max)
```

```
Declare @gender nvarchar(10)
Set @gender = 'Male'
Set @sql = 'Select Count(*) from Employees where Gender=@gender'
Execute sp_executesql @sql, N'@gender nvarchar(10)', @gender
```

At the moment we are not using output parameters. If you want the count of employees to be returned using an OUTPUT parameter, then we have to do a slight modification to the query as shown below. The key here is to use the OUTPUT keyword in your dynamic sql. This is very similar to using OUTPUT parameters with a stored procedure.

```
Declare @sql nvarchar(max)
Declare @gender nvarchar(10)
Declare @count int
Set @gender = 'Male'
Set @sql = 'Select @count = Count(*) from Employees where Gender=@gender'
Execute sp_executesql @sql, N'@gender nvarchar(10), @count int OUTPUT',
                        @gender, @count OUTPUT
Select @count
```

The OUTPUT parameter returns NULL, if you forget to use OUTPUT keyword.. The following query returns NULL, as we removed the OUTPUT keyword from @count parameter

```
Declare @sql nvarchar(max)
Declare @gender nvarchar(10)
Declare @count int
Set @gender = 'Male'
Set @sql = 'Select @count = Count(*) from Employees where Gender=@gender'
Execute sp_executesql @sql, N'@gender nvarchar(10), @count int OUTPUT',
                        @gender, @count
Select @count
```

Temp tables in dynamic sql

Suggested Videos

Part 146 - Quotename function in SQL Server
Part 147 - Dynamic SQL vs Stored Procedure
Part 148 - Dynamic sql output parameter

In this video we will discuss the **implications of creating temp tables in dynamic sql**

Temp tables created by dynamic SQL are not accessible from the calling procedure. They are dropped when the dynamic SQL block in the stored procedure completes execution.

Let us understand this with an example. Notice in the example below, all the following 3 operations are in the block of dynamic sql code.
1. Creating the Temp Table
2. Populating the Temp Table
3. Select query on the Temp Table

```
Create procedure spTempTableInDynamicSQL
as
Begin
    Declare @sql nvarchar(max)
```

```
    Set @sql = 'Create Table #Test(Id int)
                insert into #Test values (101)
                Select * from #Test'
    Execute sp_executesql @sql
End
```

So when we execute the above procedure we are able to access data from the Temp Table.

```
Execute spTempTableInDynamicSQL
```

Now, let's move the SELECT statement outside of the dynamic sql code block as shown below and ALTER the stored procedure.

```
Alter procedure spTempTableInDynamicSQL
as
Begin
    Declare @sql nvarchar(max)
    Set @sql = 'Create Table #Test(Id int)
                insert into #Test values (101)'
    Execute sp_executesql @sql
    Select * from #Test
End
```

At this point, execute the stored procedure. Notice, we get the error - Invalid object name '#Test'. This is because temp tables created by dynamic SQL are not accessible from the calling procedure. They are dropped when the dynamic SQL block in the stored procedure completes execution.

```
Execute spTempTableInDynamicSQL
```

On the other hand, dynamic SQL block can access temp tables created by the calling stored procedure. Let's prove this by modifying the stored procedure as shown below.

```
Alter procedure spTempTableInDynamicSQL
as
Begin
    Create Table #Test(Id int)
    insert into #Test values (101)
    Declare @sql nvarchar(max)
    Set @sql = 'Select * from #Test'
    Execute sp_executesql @sql
End
```

At this point, execute the stored procedure. Notice that we are able to access the temp table, which proves that dynamic SQL block can access temp tables created by the calling stored procedure.

```
Execute spTempTableInDynamicSQL
```

Summary

- Temp tables created by dynamic SQL are not accessible from the calling procedure.

- They are dropped when the dynamic SQL block in the stored procedure completes execution.
- On the other hand, dynamic SQL block can access temp tables created by the calling stored procedu