DECEMBER 2021

SQLServerGeeks MAGAZINE

The Monthly Dossier For SQL Professionals

SQL Server Deadlock Basics
by SQLMaestros
(Hands-On-Lab)

A Tip About Full Outer Join
by Erland Sommarskog

Why Missing Index Hints Are Missing
by Amit Bansal

Introducing SQL Server 2022

by Bob Ward

SQLServerGeeks

Magazine

We are thrilled to present to you, the sixth edition of the SQLServerGeeks Magazine, December 2021.

In this edition, we bring you fresh content from Bob Ward, Matt Luckham, Amit Bansal, Erland Sommarskog & SQLMaestros. A massive thanks to all our authors.

There's intriguing content on Introducing SQL Server 2022, Using JSON Objects with SQL Server, A Tip about Full Outer Join, Missing Index Hints & SQL Server Deadlock Basics.

In the August edition, we had started a new section - Hands-On-Lab. We got great feedback since then, so we are continuing with more HOL content. In this edition, we have a new HOL on "SQL Server Deadlock Basics." Hope you like it.

With great pride, we take this opportunity to share with you yet another successful execution of the signature event on Azure Data, Analytics, and AI - Data Platform Virtual Summit 2021.

DPS 2021 lived up to the monumental expectations placed upon it and most certainly delivered as promised.

Packed with 12 Training Classes, 25+ Gurukuls, 130+ General Sessions - including formats such as Deep-Dives, Short-Drives, Demos & Breakouts, DPS 2021 was a grand success and had a far-reaching impact on 7K data professionals from 94 countries.

You can now watch the session recordings here, and access the Training Class recordings here. (By the way, you can also access DPS 2020 Training Class recordings here).

We love to hear from you and make improvements to the magazine. Make sure to give us your feedback so we can continue to provide quality content, well-curated to your interests. If you are interested in writing an article for the magazine, do let us know. Write to us at Magazine@SQLServerGeeks.com

Help us spread the word. Ask your friends and colleagues to subscribe to the magazine.

From all of us at SQLServerGeeks, we wish you a pleasant read. Happy Learning.

Yours Sincerely
SQLServerGeeks Team

Got from a friend? Subscribe now to get your copy.

Our Social Channels

Website

Linkedin

Telegram

Youtube

Twitter

Facebook

CORPORATE ADDRESS

Bangalore Office:

686, 6 A Cross,
3rd Block Koramangala,
Bangalore – 560034

Kolkata Offices:

Office 1:
eDominer Systems Pvt. Ltd.
The Chambers
Office Unit 206 (Second Floor)
1865 Rajdanga Main Road
(Kasba)
Kolkata 700107

Office 2:
304, PS Continental,
83/2/1, Topsia Road (South),
Kolkata 700046

TABLE OF CONTENTS

USING JSON OBJECTS WITH SQL SERVER 2016 AND ABOVE

Matt Luckham | **luckhamm**

JSON has become the standard transmission protocol between front ends and your cloud API. Typically, the cloud API is responsible for consuming the JSON Object and then passing that data through to the database layer.

At my company, Care Control, we use SQL Server 2019 as our production database architecture. We made a decision that the database would be responsible for consuming data requests as JSON objects, and always return a JSON objects as a result.

This consistency made our Cloud API (we use .Net Core) more simplistic, with more a role as a pass through platform with the occasional responsibility for enriching the JSON object data with further data.

Simplified Login Process

We use Microsoft .Net Core to communicate with our production databases. I have extracted and simplified our login process for this example. The process describes a use case of a user "login in" to our system through the client web front end.

Step 1 – Define Login JSON Object

The first step is to define the object that the front end will produce to post to the login end point. For this example, the login object is as follows: -

```
{
  "LoginDetails": {
    "Username": "",
    "PasswordHash": ""
  }
}
```

This password hash field will hold a SHA 256 bit hashed password. Most front end libraries support 256 bit hash encryption. It is important that the hashing happens at the front end and we don't pass the actual password anywhere.

Never store the real password in the database!

Step 2 – Cloud API Enrichment

We pass the object to an open api. We use .Net Core with a Swagger (Open API) interface. For this call we want the Cloud API to enrich the JSON object with the calling IP Address. It is important that the cloud API adds this detail, rather than the front end to improve security: -

```
JSON = await JsonManager.ConstructLoginDetails(Convert.ToString(ParsedJSON.
CreateSession.Username),
                    Convert.ToString(ParsedJSON.CreateSession.PasswordHash),
                    Convert.ToString(session.OriginIPAddress));
```

This results in an object that now looks like this: -

```
{
  "LoginDetails": {
    "Username": "MLUCKHAM",
    "PasswordHash": "a7c96262c21db9a06fd49e307d694fd95f624569f9b35bb3ffacd880440f9787",
    "OriginIPAddress": "192.168.1.25"
  }
}
```

Step 3 – Call ValidateUser Stored Procedure

We then pass the object into the ValidateUser stored procedure to validate the login details. In our .Net Core Web API we use the sqlClient.SQLCommand object: -

This is our full C# code to call a generic stored procedure: -

```
public async Task<string> RunProcedure(Enums.ConnectionType Type,
string ProcName, SqlParameter[] parameters)
 {
     string connectionString;
     string result;

     // build the connection string for this calling type
     connectionString = GetConnectionString(Type);

     // create the new connection based off our connection string.
     // we're doing this within a using block for disposing reasons.
     using (var Conn = new SqlConnection(connectionString)) {

         // open SQL connection with the database.
         Conn.Open();

         // create the new database command.
         SqlCommand cmd = new SqlCommand(ProcName, Conn);

         // set the command type to procedure.
         cmd.CommandType = CommandType.StoredProcedure;

         // Implement the parameters here.
         foreach (SqlParameter p in parameters)
             cmd.Parameters.Add(p);

         // We use ExecureXMLReader as we are pulling back a single large blob
         using (var reader = cmd.ExecuteXmlReader())
         {
             if (reader.Read()) // Don't assume we have any rows.
```

```
            {
                result = reader.Value; // Handles nulls and empty strings.
            }
            else
            {
                result = "Error";
            }
        }

        // results
        Conn.Close();
    }

    return result;
}
```

Step 4 – Stored Procedure - ValidateUser

The stored procedure is the entry point to the database for the end point. In this example I have simplified the various operations that this procedure could perform, to focus on the substance of this article – validating the username and password.

This is our stored procedure. We will then break it down afterwards: -

```
CREATE PROCEDURE [Business].[ValidateUser]
(
	@dataObject VARCHAR(MAX)
)
AS
BEGIN

	DECLARE @ERROR VARCHAR(255) = '';
	DECLARE @RETURN INT = 0;
	DECLARE @ERROR VARCHAR(255) = '';
	DECLARE @RETURN INT = 0;
	DECLARE @UserName VARCHAR(50)
	DECLARE @PasswordHash VARCHAR(150)
	DECLARE @OriginIPAddress VARCHAR(20)
	DECLARE @UserID INT = -1
	DECLARE @TwoFactor VARCHAR(5) = 'FALSE'
	DECLARE @TrustIP VARCHAR(5) = 'FALSE'


	SET NOCOUNT ON;

	BEGIN TRY

		SELECT
			@UserName = d.Username,
			@PasswordHash = d.PasswordHash,
			@OriginIPAddress = d.OriginIPAddress
		FROM [Business].[tbfLoginDetails](@dataObject) as d

		IF TRIM(ISNULL(@UserName,'')) = ''
			RAISERROR('Username cannot be spaces or blank ',18,1)

		IF TRIM(ISNULL(@PasswordHash,'')) = ''
			RAISERROR('Password cannot be spaces or blank',18,1)


		IF EXISTS (SELECT * FROM [Business].[Users] as U
```

```
                                                    INNER JOIN
[Business].[UserStatus] as US ON US.StatusID = U.StatusID
                                                    WHERE Username = @UserName AND
US.StatusDescription <> 'Deactive' )
                BEGIN

                        SELECT @UserID = UserID FROM [Business].[Users] as U WHERE
Username = @UserName

                        IF NOT EXISTS (SELECT * FROM [Business].[Users] as U
                                        WHERE UserID = @UserID AND HashPassword =
@PasswordHash)
                                RAISERROR('Password incorrect',18,1)

                        -- Check if user has IP restriction
                        IF EXISTS (SELECT * FROM
[Business].[Users_IPAddress_Inclusion] as UI
                                        INNER JOIN [Business].[Users] as U ON
U.UserID = UI.UserID WHERE U.Username  = @UserName)
                                IF NOT EXISTS (SELECT * FROM
[Business].[Users_IPAddress_Inclusion] as UI
                                                INNER JOIN [Business].[Users] as U ON
U.UserID = UI.UserID
                                                WHERE U.Username  = @UserName AND
                                        IPAddress = @OriginIPAddress)
                                        RAISERROR('Invalid origin IP Address',18,1)

                        -- Call is valid - return object
                        SELECT Business.BuildLoginResponseObj(@UserName, @TrustIP,
@TwoFactor)
                END
                ELSE
                        RAISERROR('Username does not exist',18,1)

        END TRY

        BEGIN CATCH
                set @ERROR = 'ERROR:' + ERROR_MESSAGE()

                IF ISNULL(@UserID,0) > 0
                        INSERT INTO Business.Users_AccessLog (OriginIPAddress,
UserID, Result, AccessDate)
                        SELECT @OriginIPAddress, @UserID, @ERROR, GETDATE()

                SELECT [dbo].[PostAPIResult]('Failed', '', @ERROR)
        END CATCH
END
```

Stored Procedure Breakdown

```
        Entry Point
        CREATE PROCEDURE [Business].[ValidateUser]
        (
                @dataObject VARCHAR(MAX)
        )
        AS
        BEGIN

                DECLARE @ERROR VARCHAR(255) = '';
                DECLARE @RETURN INT = 0;
                DECLARE @ERROR VARCHAR(255) = '';
                DECLARE @RETURN INT = 0;
```

```
	DECLARE @UserName VARCHAR(50)
	DECLARE @PasswordHash VARCHAR(150)
	DECLARE @OriginIPAddress VARCHAR(20)
	DECLARE @UserID INT = -1
	DECLARE @TwoFactor VARCHAR(5) = 'FALSE'
	DECLARE @TrustIP VARCHAR(5) = 'FALSE'

	SET NOCOUNT ON;
```

The first few lines introduces the single parameter that is going to be passed – our data object. We declare a variable to hold our Error Value and a Return Value. We also declare other variables that we are going to use. Of course, we set NoCount to be On to give us that extra little performance boost.

Extract Data

The first part of the stored procedure is to extract the data from the data object. We create a table value function to provide a pseudo schema for our JSON Object. This is out TVF: -

```
CREATE FUNCTION [Business].[tbfLoginDetails](@JsonData VARCHAR(MAX)) --
Jsondata parametre
RETURNS @t TABLE (
		-- return temp table with these fields
			Username VARCHAR(50),
			PasswordHash VARCHAR(150),
			OriginIPAddress VARCHAR(20)

			)
AS
BEGIN

	INSERT INTO @t
	SELECT
		LoginDetails.Username,
		LoginDetails.PasswordHash,
		LoginDetails.OriginIPAddress

	FROM OPENJSON(@JsonData)
	WITH
		(LoginDetails NVARCHAR(MAX) AS JSON) as ObjJson
	CROSS APPLY OPENJSON (ObjJson.LoginDetails)
	WITH
		(Username VARCHAR(50),
		PasswordHash VARCHAR(150),
		OriginIPAddress VARCHAR(20)) as LoginDetails

    RETURN
END
```

We use his TVF in our stored procedure as below: -

```
BEGIN TRY

	SELECT
		@UserName = d.Username,
		@PasswordHash = d.PasswordHash,
		@OriginIPAddress = d.OriginIPAddress
	FROM [Business].[tbfLoginDetails](@dataObject) as d

	IF TRIM(ISNULL(@UserName,'')) = ''
		RAISERROR('Username cannot be spaces or blank ',18,1)
```

```
        IF TRIM(ISNULL(@PasswordHash,'')) = ''
            RAISERROR('Password cannot be spaces or blank',18,1)
```

We do some basic checking that the returned values are valid. Note the RAISERROR commands. This will jump code to our Catch Block as long as the Severity (second param) is greater than 10 and less than 20. We use 18 as a default.

Checking the Values

The next part of the Stored Procedure will check the login credentials: -

```
    IF EXISTS (SELECT * FROM [Business].[Users] as U
                                    INNER JOIN [Business].[UserStatus]
    as US ON US.StatusID = U.StatusID
                                    WHERE Username = @UserName AND
    US.StatusDescription <> 'Deactive' )
    BEGIN

        SELECT @UserID = UserID FROM [Business].[Users] as U WHERE
    Username = @UserName

        IF NOT EXISTS (SELECT * FROM [Business].[Users] as U
                                    WHERE UserID = @UserID AND
    HashPassword = @PasswordHash)
            RAISERROR('Password incorrect',18,1)
```

Here we are first checking that the Username matches and the user status is not deactive. If it passes this test we extract the UserID from the Users Table (will we use this later) and then verify the password hash. Remember, the password hash is a hexadecimal number so case sensitivity is not important.

Checking the IP Address

The system supports restricting access to set IP Addresses. The next check verifies that the IP Address passed is valid: -

```
    -- Check if user has IP restriction
    IF EXISTS (SELECT * FROM [Business].[Users_IPAddress_Inclusion]
     as UI
                                    INNER JOIN [Business].[Users]
    as U ON U.UserID = UI.UserID WHERE U.Username  = @UserName)
        IF NOT EXISTS (SELECT * FROM [Business].[Users_IPAddress_Inclusion]
    as UI
                                    INNER JOIN [Business].[Users]
                                    as U ON U.UserID = UI.UserID WHERE
                                    U.Username  = @UserName AND
                                    IPAddress = @OriginIPAddress)
            RAISERROR('Invalid origin IP Address',18,1)
```

Here we first check if the User has an IP Address Inclusion. We then check if the IP Address passed is in that inclusion list. If the IP Address is not valid we raise an error.

```
    -- Call is valid - return object
        SELECT Business.BuildLoginResponseObj(@UserName, @TrustIP, @TwoFactor)
    END
    ELSE
        RAISERROR('Username does not exist',18,1)
```

The final part of procedure is to return a predefined login object based on a set of parameters. We return a standard object from any login. This is the JSON constructed: -

```
{
  "LoginResponse":{
    "UserID":28,
    "UserFirstName":"Matty",
    "UserSurname":"Luckham",
    "CurrentStatus":"Active",
    "StartView":"",
    "DefaultLanguage":"Eng",
    "TrustedIP":"FALSE",
    "TwoFactorRequired":"FALSE",
    "StaffID":1,
    "SystemSettings":[
      {
        "SettingName":"ForceMessageReadonSignOut",
        "SettingValue":"0"
      },
      {
        "SettingName":"BypassMessageSignIn",
        "SettingValue":"0"
      }
    ]
  }
}
```

This object is created using the following scalar function which looks like this: -

```
CREATE FUNCTION [Business].[BuildLoginResponseObj]
(
    @Username VARCHAR(50),
       @TrustedIP VARCHAR(5),
       @TwoFactorRequired VARCHAR(5)
)
RETURNS NVARCHAR(MAX)
AS
BEGIN

    DECLARE @ReturnJSON NVARCHAR(MAX)
       DECLARE @Trusted VARCHAR(3)

    DECLARE @TempTable TABLE (
       UserID INT,
       UserFirstName VARCHAR(50),
       UserSurname VARCHAR(50),
       CurrentStatus VARCHAR(20),
       StartView VARCHAR(50),
       DefaultLanguage VARCHAR(3),
       TrustedIP VARCHAR(5),
       TwoFactorRequired VARCHAR(5),
       StaffID INT)
```

```
	INSERT INTO @TempTable
			SELECT
				U.UserID,
				U.FirstName,
				U.Surname,
				US.StatusDescription,
				Business.GetApplicationView(U.UserID),
				U.DefaultLanguage,
				@TrustedIP,
				@TwoFactorRequired,
				U.StaffLink AS StaffID
			FROM	[Business].[Users] U
			LEFT OUTER JOIN [Business].[UserStatus] US ON U.STATUSID =
	US.STATUSID
			WHERE  ISNULL(U.USERNAME,'') = @UserName;

		-- Settings table for the settings array passed down at login.
		DECLARE @SystemSettingsTempTable TABLE (
			SettingName varchar(50),
			SettingValue varchar(50)
		)

		-- population of the message settings
		INSERT INTO @SystemSettingsTempTable
		SELECT ConstantName, ConstantValue
		FROM dbo.SystemConstants
		WHERE ConstantName = 'BypassMessageSignIn'
			OR ConstantName = 'ForceMessageReadonSignOut'

		SET @ReturnJSON = (
			SELECT JSON_QUERY(
				(SELECT
					UserID,
					UserFirstName,
					UserSurname,
					CurrentStatus,
					StartView,
					DefaultLanguage,
					TrustedIP,
					TwoFactorRequired,
					StaffID,
					(
						SELECT JSON_QUERY((
							SELECT SettingName, SettingValue
							FROM @SystemSettingsTempTable
						FOR JSON AUTO))
					) as SystemSettings
				FROM
					@TempTable
				FOR JSON AUTO, WITHOUT_ARRAY_WRAPPER)) as LoginResponse
			FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
		)

	    RETURN @ReturnJSON
	END
```

This object returns a standard login object based on the username that was passed. We return a standard JSON object with a child object array of some system settings.

Note the interesting notation: -

```
SELECT JSON_QUERY(
                    (SELECT
…..

FOR JSON AUTO, WITHOUT_ARRAY_WRAPPER)) as LoginResponse
              FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
```

We do this as if we want to create a child object, unless we use the JSON_QUERY function the inner JSON will become escaped. Notice we also want to remove any array wrappers.

The Catch Block

```
BEGIN CATCH
       set @ERROR = 'ERROR:' + ERROR_MESSAGE()

       IF ISNULL(@UserID,0) > 0
              INSERT INTO Business.Users_AccessLog (OriginIPAddress, UserID,
Result, AccessDate)
              SELECT @OriginIPAddress, @UserID, @ERROR, GETDATE()

       SELECT [dbo].[PostAPIResult]('Failed', '', @ERROR)
END CATCH
```

In our catch block we return any errors using a scalar function called PostAPIResult. We also do some logging in here. Our APIResult function looks like this: -

```
CREATE FUNCTION [dbo].[PostAPIResult]
(
    @Result NVARCHAR(50),
    @ExtraDetails NVARCHAR(100),
    @ErrorDetails NVARCHAR(100)
)
RETURNS NVARCHAR(MAX)
AS
BEGIN
    DECLARE @ReturnJSON NVARCHAR(MAX)
    DECLARE @TempTable TABLE (Result NVARCHAR(50), extraDetails NVARCHAR(100),
errorDetails NVARCHAR(100))

   INSERT INTO @TempTable (Result, extraDetails, errorDetails)
   VALUES (@Result, @ExtraDetails, @ErrorDetails)

    SET @ReturnJSON = (SELECT * FROM @TempTable FOR JSON AUTO,
without_array_wrapper);

    RETURN @ReturnJSON
END
```

That is how we validate a user with a username and hashed password!

A little more on Table Value Functions

The SQL Server Table Value Function allow you to represent more complex data in a table format. You can use a TVF to "process" a JSON Object and return it as a standard relational table format for further processing. Your TVF can present a schema for JSON Objects.

Simple JSON Object

If we take a simple JSON Object: -

```
{"ClientRecord": {
    "UniqueID": "1",
    "FirstName": "Matt",
    "LastName": "Luckham",
    "Contacts": [
      {
        "Title": "Mr",
        "FirstName": "Frank",
        "LastName": "Jones",
        "Relationship": "Son"
      },
      {
        "Title": "Mrs",
        "FirstName": "Sarah",
        "LastName": "Jones",
        "Relationship": "Daughter"
      }
    ]
  }
}
```

We can use a Table Value Function to "process" this object to then present the data back to other database operations.

```
CREATE FUNCTION [dbo].[tbfClientRecord](@JsonData VARCHAR(MAX))
RETURNS @t TABLE (

                    UniqueID INT,
                    FirstName VARCHAR(50),
                    Surname VARCHAR(50))
AS
BEGIN
       INSERT INTO @t
       SELECT
              ClientRecord.UniqueID,
              ClientRecord.FirstName,
              ClientRecord.LastName
       FROM OPENJSON(@JsonData)
       WITH
              (ClientRecord NVARCHAR(MAX) AS JSON) as ObjJson
       CROSS APPLY OPENJSON (ObjJson.ClientRecord)
       WITH
              (UniqueID INT,
              FirstName VARCHAR(50),
              LastName VARCHAR(50)) as ClientRecord
    RETURN
END
GO
```

If we run our object code into this TVF we get the following result: -

```
DECLARE @JSON Varchar(max) = '{
  "ClientRecord": {
    "UniqueID": "1",
    "FirstName": "Matt",
    "LastName": "Luckham",
    "Contacts": [
      {
        "Title": "Mr",
        "FirstName": "Frank",
        "LastName": "Jones",
        "Relationship": "Son"
      },
      {
        "Title": "Mrs",
        "FirstName": "Sarah",
        "LastName": "Jones",
        "Relationship": "Daughter"
      }
    ]
  }
}'
SELECT * FROM [dbo].[tbfClientRecord](@JSON)
If you run this code you get: -


UniqueID      FirstName     Surname
    1        Matt        Luckham
```

We can also add a TVF to extract the client contact array: -

```
CREATE FUNCTION [dbo].[tbfClientContacts](@JsonData VARCHAR(MAX))
RETURNS @t TABLE (

                    ContactID INT IDENTITY(1,1),
                    ClientID INT,
                    Title VARCHAR(10),
                    FirstName VARCHAR(50),
                    LastName VARCHAR(50),
                    Relationship VARCHAR(50)
                    )
AS
BEGIN
```

```
        INSERT INTO @t (ClientID, Title, FirstName, LastName, Relationship)

        SELECT
               ClientRecord.UniqueID,
               Contacts.Title,
               Contacts.FirstName,
               Contacts.LastName,
               Contacts.Relationship
        FROM OPENJSON(@JsonData)
        WITH
               (ClientRecord NVARCHAR(MAX) AS JSON) as ObjJson
        CROSS APPLY OPENJSON (ObjJson.ClientRecord)
        WITH
               (UniqueID INT,
               Contacts NVARCHAR(MAX) as JSON) as ClientRecord
        CROSS APPLY OPENJSON (ClientRecord.Contacts)
        WITH
               (      Title VARCHAR(10),
                      FirstName VARCHAR(50),
                      LastName VARCHAR(50),
                      Relationship VARCHAR(50)) as Contacts
    RETURN
END
GO
```

When we use this function like this: -

```
DECLARE @JSON Varchar(max) = '{
  "ClientRecord": {
    "UniqueID": "1",
    "FirstName": "Matt",
    "LastName": "Luckham",
    "Contacts": [
      {
        "Title": "Mr",
        "FirstName": "Frank",
        "LastName": "Jones",
        "Relationship": "Son"
      },
      {
        "Title": "Mrs",
        "FirstName": "Sarah",
        "LastName": "Jones",
        "Relationship": "Daughter"
      }
    ]
  }
}'
SELECT * FROM [dbo].[tbfClientContacts](@JSON)
```

And we run this function we get this data set: -

	ContactID	ClientID	Title	FirstName	LastName	Relationship
1	1	1	Mr	Frank	Jones	Son
2	2	1	Mrs	Sarah	Jones	Daughter

And of course with the two functions we can de-normalise the data and output the entire JSON object as a usable table:-

```
DECLARE @JSON Varchar(max) = '{
  "ClientRecord": {
    "UniqueID": "1",
    "FirstName": "Matt",
    "LastName": "Luckham",
    "Contacts": [
      {
        "Title": "Mr",
        "FirstName": "Frank",
        "LastName": "Jones",
        "Relationship": "Son"
      },
      {
        "Title": "Mrs",
        "FirstName": "Sarah",
        "LastName": "Jones",
        "Relationship": "Daughter"
      }
    ]
  }
}'
```

```
SELECT
       CR.UniqueID,
       CR.FirstName,
       CR.Surname,
       CC.ContactID,
       CC.Title,
       CC.FirstName,
       CC.LastName,
       CC.Relationship
FROM [dbo].[tbfClientRecord](@JSON) as CR
INNER JOIN [dbo].[tbfClientContacts](@JSON) as CC ON
CC.ClientID = CR.UniqueID
```

	UniqueID	FirstName	Surname	ContactID	Title	FirstName	LastName	Relationship
1	1	Matt	Luckham	1	Mr	Frank	Jones	Son
2	1	Matt	Luckham	2	Mrs	Sarah	Jones	Daughter

Hopefully you found this article useful in how you can use SQL Server to consume JSON Objects in a consistent way.

Questions? Comments? Talk to the author today. Matt Luckham on Linkedin

About Matt Luckham

A leader, entrepreneur, manager and IT professional, with over 20 years commercial experience.

LEARN MORE

Non-Tech World of Matt

When Matt is not writing software he enjoys playing golf.

Want to write for the magazine? Comments? Feedback? Reach out to us at magazine@sqlservergeeks.com

SQL SERVER TIPS & TRICKS

1

Use WITH COMPRESSION while taking database backup. The output file size will be relatively small. Useful to transfer the backups over the network. This may consume extra CPU cycles.

2

Need to delete a large volume of data at a time (let's say 100K)? Do it in smaller chunks (1K) in a loop to reduce the impact on the log file.

3

To perform statement-level recompile, use OPTION (RECOMPILE) hint along with the query.

4

To get the status (Enabled/Disabled) of all the trace flags for a session, use DBCC TRACESTATUS();

5

To perform statement-level recompile, use OPTION (RECOMPILE) hint along with the query.

6

Use the new GREATEST function (supports Azure SQL DB/MI at the moment) to find the maximum value from a list of expressions.

More SQL Tips & Tricks

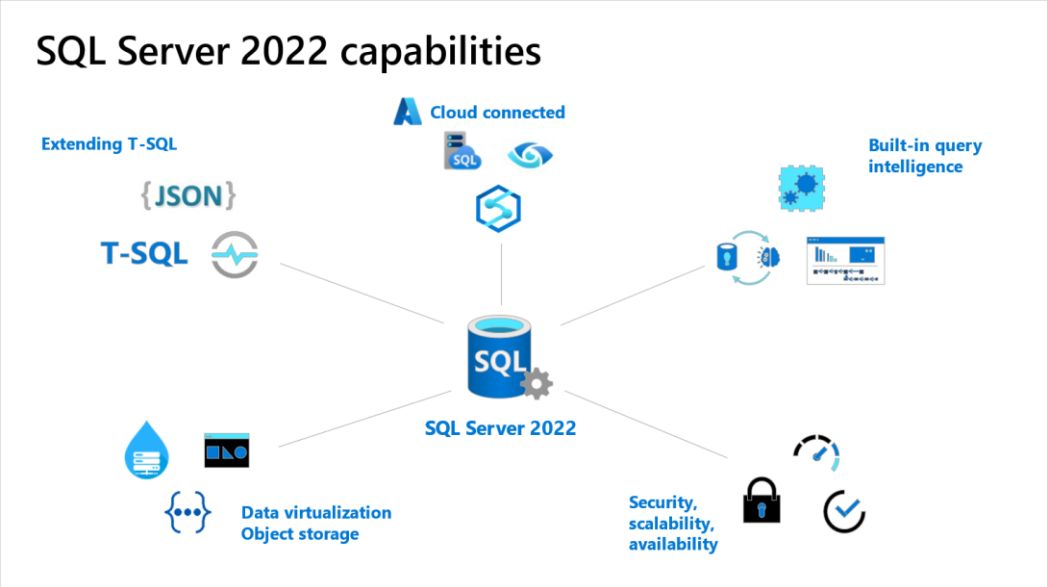
INTRODUCING SQL SERVER 2022

Bob Ward | @bobwardms

A few weeks ago we announced the next major release of SQL Server: SQL Server 2022. I wanted to give you more insights into what we announced, answer some of the common questions we have received so far including the private preview program and point you to the resources we have created so you can learn more.

SQL Server 2022 can be summarized as the most cloud-connected version of SQL Server to date with major innovations in built-in query intelligence, security, scalability, availability, data virtualization, and the T-SQL language.

Think of this release having new features in the following five areas:



I've nicknamed this visual with our team "the wheel of power". Let me explain more about the features in these areas and why these could solve challenges you face every day.

Cloud Connected

In SQL Server 2022 we will solve familiar challenges by connecting your SQL Server to Azure. For example, if you want a managed disaster recovery site, we will easily connect you with Azure SQL Managed Instance using the power of built-in Availability Group (AG) technology (and you don't have

to have an existing AG to do it). We have just shown you a glimpse of what this can do. By the time we release SQL Server 2022 this link will be a fully-functional bi-directional failover system.

Let's say you want to invest more in Azure Synapse Analytics but you are struggling to copy your data from SQL Server. The data is never really up to date and you have to build expensive ETL applications. SQL Server 2022 allows you to *link* SQL Server to Synapse. You can now select what tables in your SQL Server database you want synchronized to Synapse and we take care of the rest. We will synchronize your initial data set and then capture changes and feed them directly into SQL Pools. You now have the full data warehouse capabilities of Synapse on your data leaving you to focus your SQL Server on operational workloads and eliminate ETL jobs.

We have other cloud connected capabilities like integration with Azure Purview for centralized policy management, Azure Active Directory Authentication, and Microsoft Defender for SQL.

All of these cloud connected options are independent of each other. You choose what option you want when you need them.

Built-in Query Intelligence

We believe getting your application faster and reducing the need for expensive query tuning should be built into SQL Server. SQL Server 2022 builds on innovations in the previous releases to enhance the Query Store and Intelligent Query Processing (IQP).

Query Store will now be on by default in SQL Server 2022 for new databases. We believe we have made the right investments where most customers will benefit having this on by default. In addition, we will now allow Query Store to be enabled for read-only secondary replicas that are part of an Availability Group. Query Store data is kept on the primary replica but metadata tracks the source of the query (primary or secondary). Query Store hints are also part of SQL Server 2022 allowing you to shape query plans without changing the application and persist this for future execution.

Intelligent Query Processing (IQP) is all about fast and consistent performance with no code changes, all built into the query processor. Our customers have made significant use of these enhancements in Azure, SQL Server 2017, and SQL Server 2019. IQP has always been about solving problems we see customers face every day and SQL Server 2022 is no exception. For example, parameter sensitive plans have been a thorn in developers and SQL Server professionals for years. Now by moving to the latest database compatibility level with SQL Server 2022 (160), the query processor can cache multiple plans for the same stored procedure or parameterized statement. No longer will parameters that are sensitive to skewed data values cause unpredictable performance. We are also smart about how we determine how many plans can be cached so we don't bloat your plan cache. We achieve this functionality by creating variants of the query but the variants are hashed to the same query text. There are other *next-gen* IQP features like DOP and Cardinality Estimation (CE) feedback. These features will use the Query Store to take and store feedback about query execution to adjust query plans for consistent performance. IQP is a huge area of investment for SQL Server 2022. You can learn more with the resources I've listed at the end of this article.

Security, Scalability, and Availability

Many of you have heard me call innovations to the core engine the "meat and potatoes" of SQL Server. I borrowed that term from Conor Cunningham who always counseled me to always let our customers know about what we are doing with the core engine, because without it we don't have a product.

I'll call out one feature from each of these areas but there are many others that are part of SQL Server 2022:

Security

SQL Server Ledger solves the challenge of to have tamper evidence records of changes to data. Also in preview in Azure, we are bringing the power of blockchain to SQL Server and integrating this directly into the database. Ledger provides historical records of transaction changes but also uses crypto hashing algorithms to store information about these changes. It also includes capabilities to store a digest of hashed blocks in a separate trusted storage site. This allows a complete verification of changes to data at all levels. Hackers can try to gain access to the raw data stored in SQL Server tables but the ledger has verification mechanisms to detect any intruder. I've personally tried to hack the ledger myself using my knowledge of SQL Server internals but it appears to be "Bob Ward proof".

Scalability

We are also always looking for ways to make the engine more scalable for applications. One example in SQL Server 2022 is our continued effort to remove bottlenecks for tempdb. We have improved latch concurrency with more key system allocation pages. Combined with past innovations, we are close to having a *latch free worry* tempdb system for your applications.

Availability

Customers have used peer-to-peer transaction replication with SQL Server to implement a multi-write application across servers. The problem is that conflicts could result in a halt of replication and manual conflict detection used ID numbers which may not be a very logical answer. With SQL Server 2022, you have the choice now of configuring replication so that conflict detection is automatic and based on a *last-writer wins* concept using UTC datetimes synchronized across servers.

There is more and we look forward to keep giving you more details so you can see how we are innovating the core database engine.

Data Virtualization and Object Storage

We introduced the concept of data virtualization in SQL Server 2016 with Polybase. Data virtualization is the idea to keep "data where it lives". Use the power of T-SQL to access this data and bring back results, instead of moving the data. In SQL Server 2019, we introduced new Polybase capabilities with ODBC drivers to sources such as Oracle and MongoDB. In SQL Server 2022, we are extending Polybase to use a new technique to access data with REST APIs. You will now be able to build external tables or use the T-SQL OPENROWSET function to access files in Parquet, JSON, CSV or Delta formats with sources such as S3 compatible storage.

In addition, you can now use T-SQL to backup or restore a native SQL Server backup to any S3 compatible storage provider. We have found many customers shifting to common object storage systems and S3 is a popular protocol to access this storage.

Extending T-SQL

We still believe T-SQL is one of the best and most popular database languages in the industry. So instead of making you learn a new language, we are constantly extending the T-SQL language to support new capabilities. For SQL Server 2022, T-SQL extensions include new JSON functions, improve existing functions around string processing and others, and support for T-SQL functions to process time-series data as supported today with Azure SQL Edge.

You have questions

You may have heard that we announced SQL Server 2022 as a *private preview*. I've had several questions around what, how, and why. Let me share with you some of the most common questions and our answers.

Q: Why is it called a Private Preview and what does this mean?

A: A private preview means the release is not available to publicly download. Today, in order to try out this release you must register through our Early Adopter Program (EAP) at https://aka.ms/eapsignup. Not everyone who signs up will be accepted into the program. We built the program to work with customers who can directly and regularly interact with us and get feedback to make this a great release. This is not the first time we announced a major release of SQL Server with a private preview. It allows us to work with you to refine the product and gain directed feedback before we move to a public preview.

Q: When will SQL Server 2022 be available for public preview and General Availability?

A: I'll answer this question the same way I did for previous SQL Server releases. We won't miss the name (insert mild laughter here). Our plans are to release a public preview and general availability in calendar year 2022. We just are not ready to disclose the exact dates yet.

Q: Do I have to connect to Azure to use SQL Server 2022?

A: Definitely not. We are offering you the ability to connect to Azure with SQL Server 2022 on your terms for managed disaster recovery, near real-time analytics, and policy management. There are plenty of great capabilities just "built-into the box" without having to connect to Azure.

Q: What features will go into which editions and how will licensing and pricing work?

A: Like all past SQL Server releases, we don't make any announcements about pricing, licensing, or edition and features until the product goes to Generally Availability. We do ask for feedback on these topics during our preview program.

Q: Can I share information about SQL Server 2022 with my colleagues or customers?

A: Yes. Anything that Microsoft has released publicly can be shared with anyone. For example, we have PDF versions of our slides we have been presenting available at https://aka.ms/sqlserver2022decks. However, since the release is in private preview, only Microsoft will be showing demonstrations at this time.

Q: Where is the pre-release documentation for SQL Server 2022?

A: Members of the private preview program have access to GitHub repos for documentation. The official public pre-release of the documentation for SQL Server 2022 will be available to everyone when the product moves to public preview.

SQL Server 2022 resources

We are as excited as everyone else for SQL Server 2022. I'm sure you want to know more. Here are some resources I think you will find valuable.

https://aka.ms/sqlserver2022 - This is the main public website for SQL Server 2022

https://aka.ms/eapsignup - This is the site to sign-up for the private preview program

https://aka.ms/sqlmechanics22 - Enjoy this brief video as I share insights into SQL Server 2022 with the famous Microsoft Mechanics team.

https://aka.ms/sqlserver2022webinar - Sign-up for this on-demand webinar about SQL Server 2022

https://aka.ms/sqlserver2022decks - Get PDF versions of the decks presented by the Microsoft team on SQL Server 2022

http://aka.ms/dataexposed-sqlserver2022 - Keep up with the latest info on SQL Server 2022 in this series with Data Exposed.

Follow me at @bobwardms, my colleague @SQLPedro, or @AzureSQL for announcements on future public presentations myself and others from our team will be giving on SQL Server 2022. We all look forward to sharing more about SQL Server 2022 with all of you.

Questions? Comments? Talk to the author today. Bob Ward on Twitter.

About Bob Ward

Bob Ward is a Principal Architect for the Microsoft Azure Data team, which owns the development for all SQL Server versions. Bob has worked for Microsoft for 27+ years on every version of SQL Server shipped from OS/2 1.1 to SQL Server 2019 including Azure SQL.

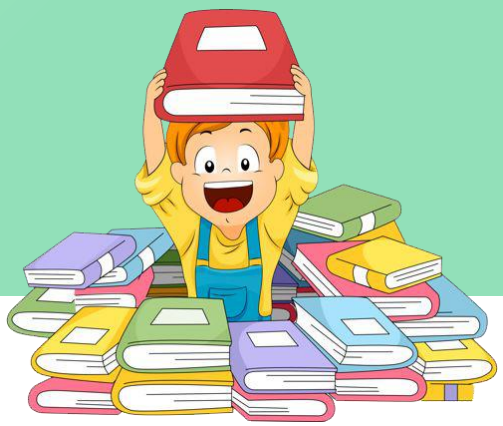
LEARN MORE

Non-Tech World of Bob Ward

During his free time, Bob loves to spend time with his wife Ginger and to cheer on his favorite college and pro sports teams.

Want to write for the magazine? Comments? Feedback? Reach out to us at magazine@sqlservergeeks.com

DPS 2021 HIGHLIGHTS



The DPS Team has released all DPS 2021 sessions for the worldwide data community. You can have free access and watch the sessions on-demand.

Tips & Tricks for creating Paginated Reports based on Power BI datasets **by Ferenc Csonka**

Building a Production Environment for Azure Arc enabled Data Services **by Chris ADKIN**

Azure DevOps and Database Deployment Automation **by Grant Fritchey**

No Code Custom Visuals in Charticulator **by Mike Carlo**

Using clustered columnstore indexes efficiently **by Thomas Grohser**

Govern Power BI and the rest of your data estate with Azure Purview **by Craig Bryden**

Now where did THAT estimate come from? **by Hugo Kornelis**

Bad Decisions Revealed: How our Emotional Brain Wins Over our Thinking Brain Every Single Time! **by Tammy Guns**

Help!! My SQL Server database is not performing well! **by Michelle Gutzait**

Parallelism in Microsoft SQL Server **by Torsten Strauss**

Building a Data Lake with Azure Synapse Analytics **by Warner Chaves**

All You Wanted to Know About Collations **by Erland Sommarskog**

Starting with Azure SQL Edge on Raspberry Pi **by Hasan Savran**

Getting Started with Extended Events **by Kathi Kellenberger**

Getting started with Query Store **by Deepthi Goguri**

Docker Deep Dive **by Andrew Pruski**

Quicker, faster and easier Power BI datasets using Power BI Premium Per User/Premium **by Gilbert Quevauvilliers**

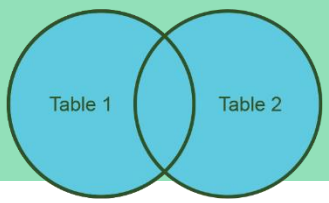
Serverless Azure data handling **by Benjamin Kettner**

SQL Assessment - Microsoft's Best Practices Checker **by Taiob Ali**

IoT Data: From Edge to Cloud - OPC/UA data Journey using Azure IoT Environment **by Jorge Maia**

Learn More

A TIP ABOUT FULL OUTER JOIN



Erland Sommarskog

You probably use both inner joins and left outer joins day in and day out, but the occasions when you need to use full outer joins are far less common. In a collection of some 5000 stored procedures I had lying around, I found that were around 14,000 inner joins, 4,280 left outer joins, but only 18 full outer joins.

This month we will look a little closer at this operator, because it's one you easily can go wrong with, if you are not careful. I will also make some observations that are applicable to the other join types. I will first cover some things about inner joins and left outer joins as this also helps to illustrates some of the issues you can run into with full outer joins.

As a starter, let's first repeat the different join types. Here are two very simple tables:

```
CREATE TABLE #a(aid    int           NOT NULL PRIMARY KEY,
                adata  nvarchar(20)  NOT NULL)
INSERT #a(aid, adata) VALUES (1, N'First'), (2, N'Second')
CREATE TABLE #b(bid    int           NOT NULL PRIMARY KEY,
                bdata  nvarchar(20)  NOT NULL)
INSERT #b(bid, bdata) VALUES (2, N'Second'),(3, N'Third')
```

Here is an inner join:

```
SELECT *
FROM   #a
INNER  JOIN #b ON #a.aid = #b.bid
```

We only get the rows with id = 2, as they are in both tables:

```
aid         adata               bid         bdata
----------- ------------------- ----------- -------------------
2           Second              2           Second
```

If we change INNER JOIN to LEFT OUTER JOIN, both rows on the left side are retained, and we get NULL where a row is missing on the right side:

```
aid         adata               bid         bdata
----------- ------------------- ----------- --------------------
1           First               NULL        NULL
2           Second              2           Second
```

And if we instead use RIGHT OUTER JOIN, both rows on the right side are retained, and we get NULL for the missing row on the left side:

```
aid         adata               bid         bdata
----------- ------------------- ----------- --------------------
2           Second              2           Second
NULL        NULL                3           Third
```

In the sample I mentioned above, there was only one single occurrence of a right outer join. A right outer join is the same as a left outer join except that it is written the other way around. I personally find right outer joins very confusing, and I suspect that I am not alone. I will not discuss right outer joins further in this article.

Finally, with FULL OUTER JOIN, all rows on both sides are retained, leaving NULLs on the side where there is no match:

```
aid         adata               bid         bdata
----------- ------------------- ----------- --------------------
1           First               NULL        NULL
2           Second              2           Second
NULL        NULL                3           Third
```

Let's now consider when we actually use these join types in practice and we will start with inner join and left outer join. Here are some tables for an order system, extremely simplified to only have the columns needed for the examples.

```
CREATE TABLE Customers(CustomerID    int          NOT NULL,
                       CustomerName  nvarchar(40) NOT NULL,
     CONSTRAINT pk_Customers PRIMARY KEY (CustomerID)
)
INSERT Customers (CustomerID, CustomerName)
   VALUES(1, N'Garima Kulkarni'), (2, N'Priya Pol'), (3, N'Manisha Mishra')
go
CREATE TABLE DiscountCodes (Code     char(6)       NOT NULL,
                            Discount decimal(5, 2) NOT NULL,
     CONSTRAINT pk_DiscountCodes PRIMARY KEY (Code)
)
INSERT DiscountCodes(Code, Discount) VALUES ('ABCDEF', 5)
go
CREATE TABLE Orders (OrderID      int     NOT NULL,
                     OrderDate    date    NOT NULL,
                     CustomerID   int     NOT NULL,
                     DiscountCode char(6) NULL,
   CONSTRAINT pk_Orders PRIMARY KEY (OrderID),
   CONSTRAINT fk_Orders_DiscountCodes
      FOREIGN KEY (DiscountCode) REFERENCES DiscountCodes(Code),
   CONSTRAINT fk_Orders_Customers
      FOREIGN KEY (CustomerID) REFERENCES Customers (CustomerID)
)
INSERT Orders(OrderID, OrderDate, CustomerID, DiscountCode)
   VALUES(11000, '2021-10-01', 1, 'ABCDEF'),
         (10000, '2020-08-23', 1, NULL),
         (10890, '2021-08-19', 2, NULL)
go
CREATE TABLE OrderDetails (OrderID    int NOT NULL,
                           ProductID  int NOT NULL,
```

```
   CONSTRAINT pk_OrderDetails PRIMARY KEY (OrderID, ProductID),
   CONSTRAINT fk_OrderDetails_Order
      FOREIGN KEY (OrderID) REFERENCES Orders(OrderID)
)
INSERT OrderDetails(OrderID, ProductID)
   VALUES (11000, 1), (11000, 2),
          (10000, 3),
          (10890, 1), (10890, 4), (10890, 5)
```

Here is a query that illustrates the use of inner and left outer joins.

```
SELECT     O.OrderID, O.OrderDate, C.CustomerName, OD.ProductID,
           O.DiscountCode, D.Discount
FROM       Orders O
INNER JOIN OrderDetails OD ON O.OrderID = OD.OrderID
INNER JOIN Customers C     ON O.CustomerID = C.CustomerID
LEFT  JOIN DiscountCodes D ON D.Code = O.DiscountCode
ORDER BY   O.OrderID, O.OrderDate
```

> **Note**: I have left out OUTER here from LEFT OUTER JOIN, as makes it easier to produce a nicely aligned query text that is easy to read.

There is an inner join from Orders to OrderDetails, because an order without details would be an anomaly. And we can make the join to Customers an inner join, since CustomerID is not nullable and there is also a foreign-key constraint. Thus, we know that there will always be a row in the Customers table to join to. With the DiscountCodes table it is different. This column is nullable, so we must use a left outer join to also include orders without a discount code. That is, the first join is from parent to child, and the other two joins are to lookup tables. You may not want to call these lookup tables "parents", but nevertheless there are foreign-key constraints in that direction, and from that sense, they are parents.

Above we used an inner join to go from parent to child, because we had reason to assume that there will always be children. But if that is not the case, we need to use a left outer join. For instance, say that we want to list all customers and their orders, keeping in mind that there may be customers who have yet to place their first order:

```
SELECT    C.CustomerName, O.OrderID, O.OrderDate
FROM      Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
ORDER BY  C.CustomerName, O.OrderID
```

This is the output:

```
CustomerName     OrderID  OrderDate
---------------- -------- ----------
Garima Kulkarni  10000    2020-08-23
Garima Kulkarni  11000    2021-10-01
Manisha Mishra   NULL     NULL
Priya Pol        10890    2021-08-19
```

There is a common mistake with left outer joins, which is worth looking at before we turn to full outer joins, because understanding this mistake will help us to understand why need to write full outer joins according to a certain pattern.

Say that in the list above, we only want to list orders from this year. If we mainly have been writing inner joins, this may seem natural to write:

```
SELECT    C.CustomerName, O.OrderID, O.OrderDate
FROM      Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
WHERE     O.OrderDate >= '2021-01-01'
ORDER BY  C.CustomerName, O.OrderID
```

However, the output is:

```
CustomerName     OrderID  OrderDate
---------------- -------- ----------
Garima Kulkarni  11000    2021-10-01
Priya Pol        10890    2021-08-19
```

As you can see, Manisha Mishra is missing. This happens because the left outer join is logically evaluated before the WHERE clause. So when the WHERE clause is evaluated, the OrderDate column is NULL for the row for Manisha Mishra. Thus, it is filtered out by the WHERE condition. Essentially, the WHERE clause changes the outer join to a plain inner join.

There is more than one way to change this. One is to extend the WHERE clause with the condition `OR O.OrderDate IS NULL`, but it is a little more verbose and may hamper readability in a more complex query. It may also be inefficient, because of limitations in the optimizer.

A better approach, both for readability and efficiency is to move the condition from the WHERE clause to the ON clause, so that the filter on OrderDate is evaluated as part of the join operation:

```
SELECT    C.CustomerName, O.OrderID, O.OrderDate
FROM      Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
                  AND O.OrderDate >= '2021-01-01'
ORDER BY  C.CustomerName, O.OrderID
```

This lists all three customers:

```
CustomerName       OrderID     OrderDate
------------------ ----------- ----------
Garima Kulkarni    11000       2021-10-01
Manisha Mishra     NULL        NULL
Priya Pol          10890       2021-08-19
```

After this lengthy introduction we now come to the main topic for this article, that is, full outer joins. We will however come back to these examples to apply the observations we will make when working with full outer joins.

In all the examples above, the joins were over a foreign-key relation and that is indeed very common. But in a full outer join there can be lone rows on both sides, something which normally cannot happen when there is a foreign-key constraint, since this enforces that there must be a row in the parent table. No, the tables in a full outer join are typically peers with the same set of keys. This is not exactly common.

Here is a prime example of tables being peers: We have two versions of the same query or stored procedure. We have made changes to the original to improve performance, and we want to ensure

that the result did not change. We save the result of the original version and the modified version into temp tables, and to compare them we run a full outer join like this:

```
SELECT ...
FROM      #ref r
FULL JOIN #new n ON r.keycol1 = n.keycol1
                AND r.keycol2 = b.keycol2
                AND ...
WHERE     r.col1 <> n.col1 OR
          r.col1 IS NULL AND n.col1 IS NOT NULL OR
          r.col1 IS NOT NULL AND n.col1 IS NULL OR
          r.col2 <> n.col2 OR
          ...
```

Because we have to check for the NULL values separately, the WHERE clause gets very tedious, but apart from that, the query does the job very well. (There is in fact a way to write this WHERE clause in a less tedious manner, but that is not a trick specific to full outer join, and this is something I will return to in the next issue of the SQLServerGeeks Magazine.) This initial example is straightforward, since we work with the tables in full. But if we only want to work with a subset of the tables, we need to be more careful.

A situation where full outer join often is needed is when we want to reconcile data from two different systems. Say that we have a need to mirror accounts from a couple of banks. We get transactions from the banks and we also send transactions to the banks when our clients buy or sell services or goods from us.

Every once in a while, we get files with the actual standings from the banks so that we can verify that our data is correct. Here is a script for these tables with sample data (as above, the tables are simplified to focus at what is essential):

```
CREATE TABLE OurData (CustomerID int           NOT NULL,
                      BankID     int           NOT NULL,
                      Amount     decimal(20,2) NOT NULL,
     CONSTRAINT pk_OurData PRIMARY KEY (CustomerID, BankID)
)
go
INSERT OurData(CustomerID, BankID, Amount)
  VALUES(1, 1, 1000.00), (2, 1, 1532.12), (3, 1, 1420.56),
        (1, 2,  536.00), (2, 2, 499.00)
go
CREATE TABLE BankData (CustomerID int           NOT NULL,
                       BankID     int           NOT NULL,
                       Amount     decimal(20,2) NOT NULL,
     CONSTRAINT pk_BankData PRIMARY KEY (CustomerID, BankID)
)
INSERT BankData(CustomerID, BankID, Amount)
  VALUES(1, 1, 1000.00), (2, 1,  532.12),              (4, 1, 1230.56),
        (1, 2,  759.00), (2, 2, 499.00)
go
```

When running a reconciliation, we want to do that for one bank at a time, since the reconciliation is only meaningful when we just have received a file. It's pointless to reconcile against data that is several days old.

A casual programmer who are in the auto-pilot mode for inner joins might try:

```
DECLARE @bank int = 1
SELECT    coalesce(A.CustomerID, B.CustomerID) AS CustomerID,
          A.Amount AS OurAmount, B.Amount AS BankAmount,
          coalesce(A.Amount, 0) - coalesce(B.Amount, 0) AS  DiffAmount
FROM      OurData A
FULL JOIN BankData B ON A.CustomerID = B.CustomerID
                    AND A.BankID     = B.BankID
WHERE     A.BankID = @bank
  AND     coalesce(A.Amount, 0) <> coalesce(B.Amount, 0)
```

But the result is not correct:

```
CustomerID  OurAmount   BankAmount   DiffAmount
----------- ----------- ------------ -----------
2           1532.12     532.12       1000.00
3           1420.56     NULL         1420.56
```

There should be a row for customer 4 as well. And if you recall the example with left outer join where we filtered on OrderDate in the WHERE clause, you understand why we get the wrong result: The condition filters out rows where A.BankID is NULL, and the WHERE clause effectively transforms the full outer join to a right outer join.

Before we look at how to address this, let me first highlight a few things in this query that are correct, and which are typical for full outer joins. In the SELECT list, we have this:

```
coalesce(A.CustomerID, B.CustomerID) AS CustomerID
```

Since the CustomerID can come from any side, we need to use `coalesce` (or `isnull`) to be sure to get a value. This quite typical for how we display key columns in full-outer-join queries. Occasionally, though, you may prefer to have two columns to make it easier to see from which table(s) the data is coming, but the key is you cannot only rely one table. Recall that full outer join is a symmetrical operation and that the tables are peers!

We also have this in the SELECT list:

```
A.Amount AS OurAmount, B.Amount AS BankAmount
```

For this reconciliation we need to see the amounts on both sides. This is also quite typical for full outer joins: for non-key columns that are in both tables, we want to display both values, since we often to compare them against each other.

Also note this condition in the WHERE clause:

```
AND coalesce(A.Amount, 0) <> coalesce(B.Amount, 0)
```

Since Amount will be NULL when there is no row on that side, we must account for this in the WHERE condition. Exactly how to do this, depends on the business rules. In the example above, where we compared the result of two queries that should produce exactly the same result, we had to add specific conditions to test for NULL. But for our reconciliation, the business rules tell us that the absence of a row equates to an amount of 0, so we can apply `coalesce` (or `isnull`) for the task. You can also see the same pattern in the SELECT list for the DiffAmount column.

Let's now get back to the broken filtering on the BankID. As with the case of the left outer join, we can solve the issue by changing the WHERE clause to have an OR condition, in this case `(A.BankID = @bank`

`OR B.BankID = @bank`). This certainly treats the tables as peers, but it can become bulky if the conditions are more complex, and even more so if the conditions are different for the two tables. (Just because the tables are peers, they need not to be identical.) This gets even more apparent if we need to filter against other tables. Also, as with left outer join, the optimizer may not be able to find the best plan when we use OR.

For the left outer join, the simple fix was to move the condition from the WHERE clause to the ON clause, but if we try this with full outer join, the result is not usable. For instance, this query:

```
DECLARE @bank int = 1
SELECT    coalesce(A.CustomerID, B.CustomerID) AS CustomerID,
          A.Amount AS OurAmount, B.Amount AS BankAmount,
          coalesce(A.Amount, 0) - coalesce(B.Amount, 0) AS  DiffAmount
FROM      OurData A
FULL JOIN BankData B ON A.CustomerID = B.CustomerID
                     AND A.BankID     = B.BankID
                     AND A.BankID     = @bank
                     AND B.BankID     = @bank
WHERE    coalesce(A.Amount, 0) <> coalesce(B.Amount, 0)
```

Produces this nonsense:

```
CustomerID  OurAmount  BankAmount  DiffAmount
----------- ---------- ----------- -----------
1           NULL       759.00      -759.00
1           536.00     NULL        536.00
2           1532.12    532.12      1000.00
2           NULL       499.00      -499.00
2           499.00     NULL        499.00
3           1420.56    NULL        1420.56
4           NULL       1230.56     -1230.56
```

I leave it as an exercise to the reader to figure out what is happening here.

No, to find a good solution, we need to think a little more of the logic of what we want to do. What we really want to do is to first extract the rows for the bank in question from both tables, and once we have done this, we want to compare these extractions. Thus, we must write the query so that logically the bank filter is applied before the full outer join. Here is a query to do this:

```
DECLARE @bank int = 1
SELECT     coalesce(A.CustomerID, B.CustomerID) AS CustomerID,
           A.Amount AS OurAmount, B.Amount AS BankAmount,
           coalesce(A.Amount, 0) - coalesce(B.Amount, 0) AS  DiffAmount
FROM      (SELECT CustomerID, Amount FROM OurData  WHERE BankID = @bank) A
FULL JOIN (SELECT CustomerID, Amount FROM BankData WHERE BankID = @bank) B
     ON    A.CustomerID = B.CustomerID
WHERE     coalesce(A.Amount, 0) <> coalesce(B.Amount, 0)
```

That is, as table sources for the full outer join, we use two derived tables that filter out the bank. Now we get the correct result:

```
CustomerID  OurAmount  BankAmount  DiffAmount
----------- ---------- ----------- -----------
2           1532.12    532.12      1000.00
3           1420.56    NULL        1420.56
4           NULL       1230.56     -1230.56
```

Some readers may prefer to use common table expressions (CTEs) instead:

```
DECLARE @bank int = 1
; WITH A AS (
    SELECT CustomerID, Amount FROM OurData WHERE BankID = @bank
), B AS (
    SELECT CustomerID, Amount FROM BankData WHERE BankID = @bank
)
SELECT     coalesce(A.CustomerID, B.CustomerID) AS CustomerID,
           A.Amount AS OurAmount, B.Amount AS BankAmount,
           coalesce(A.Amount, 0) - coalesce(B.Amount, 0) AS  DiffAmount
FROM       A
FULL  JOIN B ON A.CustomerID = B.CustomerID
WHERE      coalesce(A.Amount, 0) <> coalesce(B.Amount, 0)
```

Which you use is completely a matter of taste. The queries are equivalent and will always produce the same query plan in SQL Server.

Normally, when we work with inner joins and the left side of left outer joins, we typically put initial filtering in the WHERE clause. For instance, let's take the initial query on orders, but add the condition that we only want see orders from this year. Most people would write this as:

```
SELECT     O.OrderID, O.OrderDate, C.CustomerName, OD.ProductID,
           O.DiscountCode, D.Discount
FROM       Orders O
INNER JOIN OrderDetails OD ON O.OrderID = OD.OrderID
INNER JOIN Customers C     ON O.CustomerID = C.CustomerID
LEFT  JOIN DiscountCodes D ON D.Code = O.DiscountCode
WHERE      O.OrderDate >= '2021-01-01'
ORDER BY   O.OrderID, O.OrderDate
```

But taking the above in regard, you could argue that it would be better to write it this way:

```
SELECT    O.OrderID, O.OrderDate, C.CustomerName, OD.ProductID,
          O.DiscountCode, D.Discount
FROM      (SELECT * FROM Orders WHERE OrderDate > '2021-01-01') O
JOIN      OrderDetails OD ON O.OrderID = OD.OrderID
JOIN      Customers C ON O.CustomerID = C.CustomerID
LEFT JOIN DiscountCodes D ON D.Code = O.DiscountCode
ORDER BY  O.OrderID, O.OrderDate
```

That is, we first filter out rows from the Orders want to work on, before we join to the rest of the tables.

Or take the query where we listed customers and their orders for this year. We can apply this technique to this query as well:

```
SELECT    C.CustomerName, O.OrderID, O.OrderDate
FROM      Customers C
LEFT JOIN (SELECT * FROM Orders WHERE OrderDate >= '2021-01-01') O
     ON   C.CustomerID = O.CustomerID
ORDER BY  C.CustomerName, O.OrderID
```

This is somewhat more verbose that sticking the condition in the ON clause, but by separating the filter condition from the join condition, it makes our intentions clearer.

If we were to apply this model across the board, we would only use the WHERE clause to filter on conditions on non-key columns involving two or more tables in the join. One example of this is the comparison of the Amount columns in the full outer join. However, out of habit most people use the WHERE clause for this initial filtering, and I will need to confess that I normally do this myself too. As long it is a matter of inner joins, or outer tables in left outer joins, it does not matter. The logical result will be the same, and it also highly likely that the optimizer will get it right.

But when we work with full outer joins, or for that matter with the inner tables of left outer joins, we really need to think about what we are doing and make sure we apply them in right order:

1. First extract the rows from the source tables we want to work on. This includes filtering that we need to do against other tables.
2. Perform the join operation.
3. Apply any filter on the result of the join.

Let's now take this one step further and look at a three-table full outer join. This is even less common, and in the sample I mentioned in the beginning, there were no case of a three-way full outer join. However, I know that one of these procedures originally had exactly this, because I wrote it myself many years ago. And I still remember that it was absolutely not a walk in the park. It took me quite some time to get this right, and this experience is the inspiration for this article.

Let's say that in our reconciliation problem that there is a transactions table. The effect of these transactions is reflected in OurData, but unsent transactions are not reflected in BankData, so we need to account for these transactions. And while unlikely, there could be a customer that is only in the transactions table. Then again, the purpose of a reconciliation is to find things that are out of order, so we need to account for this case. Here is a script for this table:

```
CREATE TABLE BankTransactions (TransactionID int           NOT NULL,
                               CustomerID    int           NOT NULL,
                               BankID        int           NOT NULL,
                               IsSent        bit           NOT NULL,
                               Amount        decimal(20,2) NOT NULL,
    CONSTRAINT pk_BankTransactions PRIMARY KEY (TransactionID)
)
INSERT BankTransactions (TransactionID, CustomerID, BankID, IsSent, Amount)
   VALUES(980, 1, 1, 1, 1000.00),
         (981, 2, 1, 0,  100.00),
         (982, 2, 1, 0,  900.00),
         (983, 3, 1, 0,  700.00),
         (984, 4, 1, 0, -1230.56),
         (985, 5, 1, 0, -1900.00),
         (900, 1, 2, 0, 200)
```

As we have learnt, to work with the BankTransactions table we need to filter it on Bank and also on the IsSent column, before we can join it to the other two tables. As it happens this comes naturally, since we need to aggregate the data per customer before we can join. So this leads is to:

```
DECLARE @bank int = 1
; WITH A AS (
    SELECT CustomerID, Amount FROM OurData WHERE BankID = @bank
), B AS (
    SELECT CustomerID, Amount FROM BankData WHERE BankID = @bank
), C AS (
   SELECT CustomerID, SUM(Amount) AS Amount
   FROM   BankTransactions
```

```
      WHERE  BankID = @bank
        AND  IsSent = 0
      GROUP  BY CustomerID
   )
   SELECT    coalesce(A.CustomerID, B.CustomerID, C.CustomerID) AS CustomerID,
             A.Amount AS OurAmount, B.Amount AS BankAmount,
   C.Amount AS TransAmount,
             coalesce(A.Amount, 0) - coalesce(B.Amount, 0) AS DiffAmount,
             coalesce(A.Amount, 0) -
                (coalesce(B.Amount, 0) + coalesce(C.Amount, 0)) AS  AdjDiffAmount
   FROM      A
   FULL JOIN B ON A.CustomerID = B.CustomerID
   FULL JOIN C ON C.CustomerID = ?
```

Let's first look at the first element in the SELECT list. There are now three sources the CustomerID may come from, and we need to account for all of them. (And since `coalesce` accepts any number of arguments, it makes it a winner over `isnull` here.)

But what should we put in place of the question mark? Had this been a plain matter of two inner joins, the answer would be that it does not matter if we take A.CustomerID or B.CustomerID, as they are equivalent. But if you pick, say, A.CustomerID, you will find that you get two rows with CustomerID = 4, one with the data from B and with the data from C. And if you instead try B.CustomerID, you get two rows for CustomerID = 3.

No, again, in a full outer join, the tables are peers and we need to handle them symmetrically, for instance any of these two:

```
   FULL JOIN C ON C.CustomerID = A.CustomerID OR
                  C.CustomerID = B.CustomerID
   FULL JOIN C ON C.CustomerID = coalesce(A.CustomerID, B.CustomerID)
```

Yes, I have discouraged the use of OR earlier in this article, but there is not really any better option here.

This is the output with any of these conditions:

```
CustomerID  OurAmount  BankAmount  TransAmount  DiffAmount  AdjDiffAmount
----------- ---------- ----------- ------------ ---------- --------------
1           1000.00    1000.00     NULL         0.00        0.00
2           1532.12    532.12      1000.00      1000.00     0.00
4           NULL       1230.56     -1230.56     -1230.56    0.00
3           1420.56    NULL        700.00       1420.56     720.56
5           NULL       NULL        -1900.00     0.00        1900.00
```

Some readers may think that these queries can also be written with help of the UNION ALL operator. That is likely to be true, although I have not tried this myself, but I leave it as an exercise to the reader. It may also be that the UNION ALL queries are more efficient that the queries with full outer join, not the least in the example with the three-table join. However, not all queries with full outer join can easily be rewritten with UNION ALL. For this article, I wanted to use very simple examples to focus on my main points: you must identify the logical order you need for the operations in the query, and write your query accordingly. Filter source tables before joining them. And in the join conditions, the SELECT list and the WHERE clause handle table symmetrically. They are peers.

Finally, I should point out that there is one operation in SQL Server which includes a full outer join, but where you may not have thought of it, and that is the MERGE statement, particularly in the case where

you have a branch with WHEN NOT MATCHED BY SOURCE. But at least, with MERGE there is no WHERE clause that can lead us astray, but it is more apparent that we need to apply the filtering first.

Questions? Comments? Talk to the author today. Erland Sommarskog.

About Erland Sommarskog

Erland Sommarskog is an independent consultant based in Stockholm, working with SQL Server since 1991. He was first awarded SQL Server MVP in 2001, and has been re-awarded every year since.

LEARN MORE

Non-Tech World of Erland Sommarskog

Erland plays bridge (when there is not a pandemic), enjoys travelling (when there is not a pandemic), and in the far too short Swedish summer he goes hiking and biking in the forests and around the lakes around Stockholm.

Want to write for the magazine? Comments? Feedback? Reach out to us at magazine@sqlservergeeks.com

DPS 2021 - A GRAND SUCCESS

With great pride, we take this opportunity to share with you, yet another successful execution of the signature event on Azure Data, Analytics, and AI - Data Platform Virtual Summit 2021.

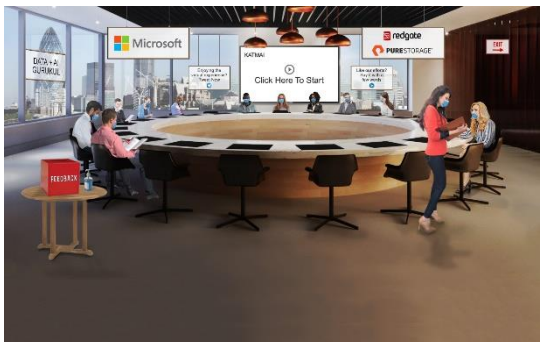
DPS 2021 lived up to the monumental expectations placed upon it and most certainly delivered as promised.

Packed with 12 Training Classes, 25+ Gurukuls, 130+ General Sessions - including formats such as Deep-Dives, Short-Dives, Demos & Breakouts, the overall event facilitated learning and networking for 7K data professionals from 94 countries. Special thanks to Microsoft, Redgate, PURE Storage, CDATA, MLPA & SQLMaestros for sponsoring the event.

DPS 2021 kicked off on the 8th of September, bringing you two consecutive days of Pre-Cons (training classes), delivered by seasoned trainers from the industry. Following up Pre-Cons, the Summit spanned from 13th to the 16th of September, offering curated content across several tracks including Architecture, Data Administration, Data Development, Advanced Analytics, Data Science, and the newly introduced Industry Solutions.

LOBBY

Multiple delivery formats were set up on the virtual platform including the popular Breakouts and the innovative ones like Deep-Dives, Short-Drives & Demo-Only sessions. Gurukuls were a huge hit that provided a networking opportunity with the Speakers via open discussions. At the Gurukuls, delegates discussed their project challenges & possible solutions with Microsoft Product Teams and industry experts. Overall, DPS 2021 was one of the most interactive and versatile events of its kind.



DATA + AI GURUKUL

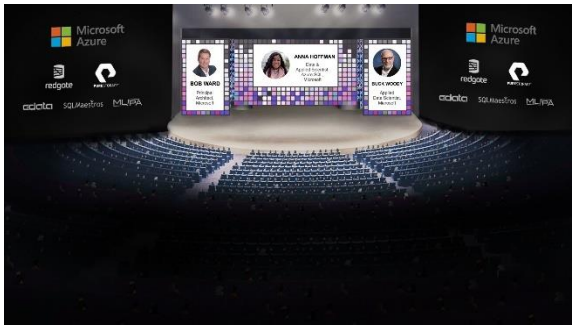
This year's keynote - SQL Edge to Cloud – featured Microsoft Veterans - Bob Ward, Anna Hoffman, and Buck Woody. With in-depth knowledge and decades of experience in the world of SQL, they provided insightful wisdom on how to get ahead of the curve by understanding the immense possibilities with the modern SQL data platform and the right "flavor" of SQL suited to your specific needs.

DPS 2021 entered its final stage with the Post-Cons on the 20th and 21st of September. Post-Cons, just like the Pre-Cons offered deep-dive content in form of 8-hour virtual classes by the world's best educators.

DPS 2021 also featured WIT & DEI panel discussions.

A massive shout-out to the Speakers, Sponsors, delegates & the DPS Team for making the event possible.

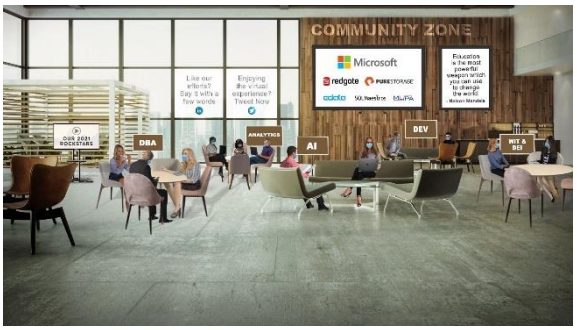
Make sure to check out the testimonials.
You can now watch the session recordings here. And access the Training Class recordings here. (By the way, you can also access DPS 2020 Training Class recordings here).



KEYNOTE ROOM

MICROSOFT BOOTH

EXHIBITOR HALL



COMMUNITY ZONE

Access Summit Recordings

Access Training Class Recordings

WHY MISSING INDEX HINTS ARE MISSING

Amit Bansal | @A_Bansal

Let's understand a few reasons why SQL Server (sometimes) does not show missing index hints.

We are using **AdventureWorks2016** database for the demo.

Let's make a copy of **Sales.Customer** table as **Sales.CustomerDup**

```
SELECT *
INTO sales.customerdup
FROM sales.customer
```

Before jumping into the actual query, have a look at **AccountNumber** column which is of **VARCHAR** data type.

```
SP_help 'sales.customerdup'
```

Name	Owner	Type	Created_datetime
customerdup	dbo	user table	2021-02-15 16:15:51.553

Column_name	Type	Computed	Length	Prec	Scale	Nullab
CustomerID	int	no	4	10	0	no
PersonID	int	no	4	10	0	yes
StoreID	int	no	4	10	0	yes
TerritoryID	int	no	4	10	0	yes
AccountNumber	varchar	no	10			no
rowguid	uniqueidentifier	no	16			no
ModifiedDate	datetime	no	8			no

Note that this table is a heap, we just created it. There are no indexes on the table.

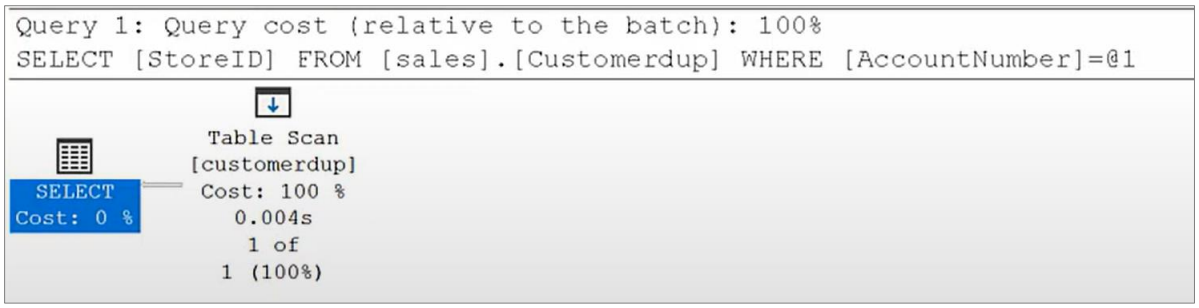
```
SP_helpindex 'sales.customerdup'
```

```
The object 'sales.customerdup' does not have any indexes, or

Completion time: 2021-02-15T17:36:50.9835066+05:30
```

Let's execute a simple query with the Actual Execution Plan turned ON (Ctrl + M). Observe the execution plan. We can see a table scan there.

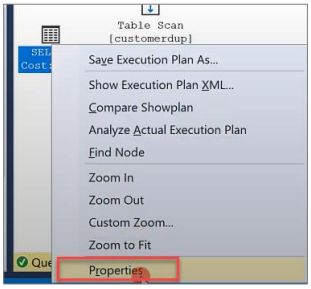
```
SELECT StoreID FROM Sales.CustomerDup
WHERE AccountNumber= N'AW00029594'
```



I was expecting that since the table is a heap and there is a predicate on **AccountNumber,** SQL Server will hint me to create an index on **AccountNumber** column. But there are no missing index hints here.

Going by the Microsoft documentation on missing index hints, SQL Server will not show missing index hints for trivial execution plans.

What does that mean? Right-click on the **SELECT** operator from the execution plan and click on properties, and observe the **Optimization Level** being **Trivial**.

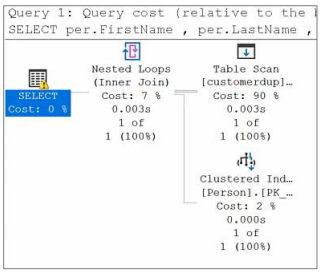


There are multiple phases in SQL Server query optimization and one such phase is called Trivial Plan Evaluation. For all the queries that fall under the Trivial phase, SQL Server will not show up missing index recommendations. To put it in simple words, the query is too simple for SQL Server to get into the headache of evaluating missing index hints.

Let's look at another example. Here, the prefix 'N' in the WHERE clause plays an important role. We already know **AccountNumber** column is of VARCHAR data type. Now NVARCHAR data (due to prefix 'N') is being compared with VARCHAR, so SQL Server will perform an implicit data type conversion. The moment we try to join two or more tables, SQL Server will perform a full optimization.

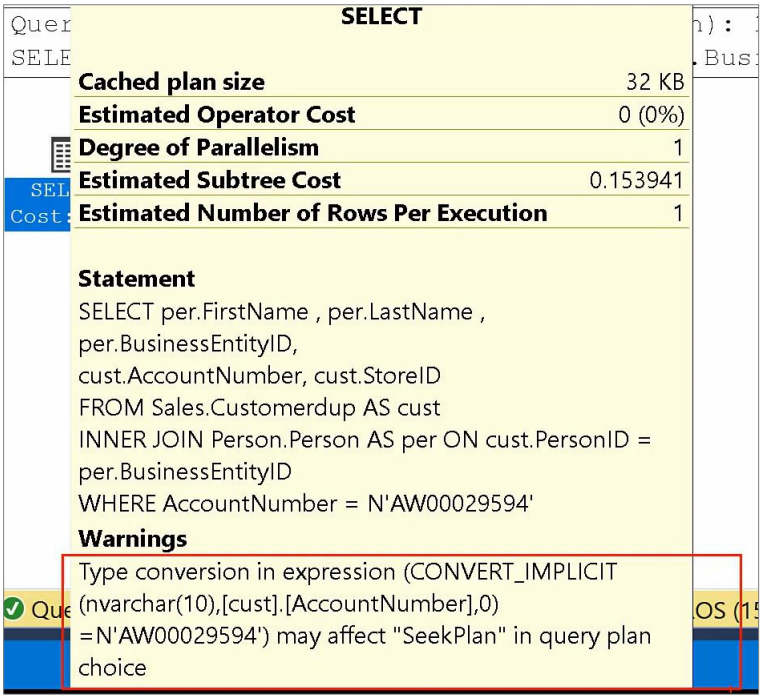
```
SELECT Per.FirstName, Per.LastName, Per.BusinessEntityID,
Cust.AccountNumber, cust.StoreID
FROM Sales.CustomerDup as Cust
INNER JOIN Person.Person AS Per ON Cust.PersonID =
      Per.BusinessEntityID
WHERE AccountNumber = N'AW00029594'
```

Once we execute the query, you will observe that there is still no missing index recommendation despite the Optimization Level being FULL.



We can observe that the optimization level is FULL.

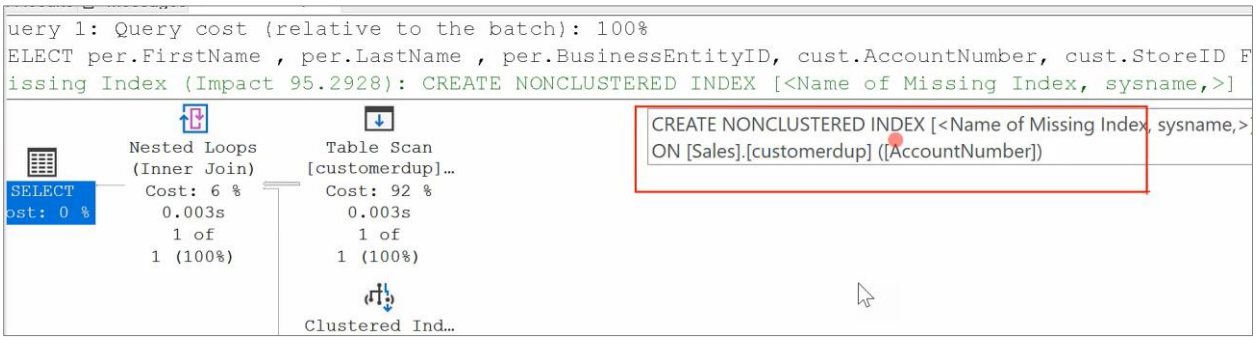
Going by the previous conclusion, when the optimization level is FULL, we should get missing index recommendations. But that does not happen as SQL Server is performing implicit conversion. If we hover over the SELECT operator in the execution plan (also observe the warning symbol on the SELECT operator), the warning clearly says that there is type conversion happening which may affect the plan choice.



Let's execute the query again, this time without the prefix 'N'.

```
SELECT Per.FirstName, Per.LastName, Per.BusinessEntityID,
Cust.AccountNumber, cust.StoreID
FROM Sales.CustomerDup as Cust
INNER JOIN Person.Person AS Per ON Cust.PersonID =
      Per.BusinessEntityID
WHERE AccountNumber = 'AW00029594'
```

Now there will be no implicit conversion. No more warnings, optimization level is FULL and SQL Server comes up with the missing index recommendation, and as expected, the recommendation is on the **AccountNumber** column.



There are the other cases too when SQL Server does not show up missing index hints, example, SARGABILITY.

Questions? Comments? Talk to the author today. Amit Bansal on Twitter.

About Amit Bansal

Amit R S Bansal is a SQL Server Specialist at SQLMaestros (brand of eDominer Systems). He leads the SQL and BI practice with a much focused team providing consulting, training and content development services to more than 160+ SQL customers globally.

LEARN MORE

Non-Tech World of Amit Bansal

During his free time, Amit loves playing racquet sports like Tennis, Badminton, etc.

Want to write for the magazine? Comments? Feedback? Reach out to us at magazine@sqlservergeeks.com

SQL NUGGETS BY MICROSOFT

Cumulative Update #14 for SQL Server 2019 RTM

Released: General Availability of Microsoft.Data.SqlClient 4.0

Private Preview: controlling access to Azure SQL at scale with policies in Purview

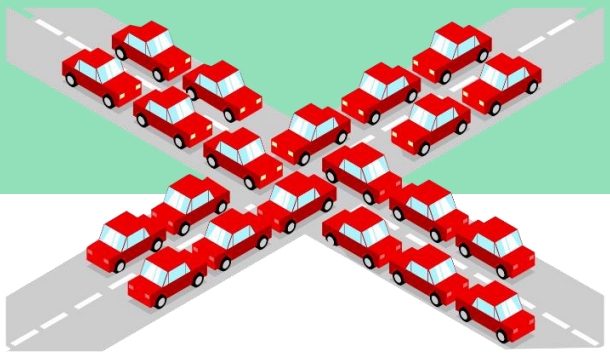
Azure SQL News Update: November 2021

Azure SQL Migration extension - November 2021 updates

Announcing the new premium-series hardware for SQL Managed Instance

Managed Instance link – connecting SQL Server to Azure reimagined

SQL SERVER DEADLOCK BASICS



SQLMaestros Hands-On-Labs | @SQLMaestros

A deadlock occurs when two or more sessions permanently block each other with each session having a lock on a resource, the other session is trying to lock. This lab is divided into 4 exercises explaining different dead lock scenarios. In the first exercise we will learn about deadlocks, second exercise will talk about deadlock detection, third exercise will explain about deadlock graph and the final exercise will talk about deadlock XML.In this exercise, we will learn the concept of NULL

Exercise 1: Introduction to Deadlocks

Scenario

In this exercise, we will look at a simple deadlock scenario in SQL Server.

Tasks	Detailed Steps
Launch SQL Server Management Studio	1. Click **Start** | **All Programs** | **SQL Server 2012** | **SQL Server Management Studio** 2. In the **Connect to Server** dialog box, click **Connect**
Open 1_CyclicDeadlock.sql	1. Click **File** | **Open** | **File** or press (Ctrl + O) 2. In **Open File** dialogue box, navigate to **SQL Server Deadlock Basics\Scripts** folder 3. Select **1_CyclicDeadlock.sql** and click **Open**
Execute the statement(s) in the Setup section	The setup section performs the following: • **SQLMaestros** database is created • **SQLMaestros** schema is created

	• Two tables are created **(Table1** and **Table2)** with 5 records each
	
	Note: We will use these tables to simulate the deadlock scenario
	
	In **1_ CyclicDeadlock.sql**, review and execute the statement(s)in section '**Begin: Setup**' and '**End: Setup**'
	
	`----------`
	`-- Begin: Setup`
	`----------`
	`-- Create a database.`
	`USE master;`
	`GO`
	
	`IF EXISTS(SELECT * FROM sys.databases WHERE name='SQLMaestros')`
	`BEGIN`
	`ALTER DATABASE [SQLMaestros] SET SINGLE_USER WITH ROLLBACK IMMEDIATE;`
	`DROP DATABASE [SQLMaestros];`
	`END`
	`GO`
	`CREATE DATABASE SQLMaestros;`
	`GO`
	
	`USE SQLMaestros;`
	`GO`
	`SET NOCOUNT ON;`
	`GO`
	
	`-- Create a schema named SQLMaestros`
	`CREATE SCHEMA [SQLMaestros] AUTHORIZATION [dbo];`
	`GO`
	
	`-- Create a table Table1 in SQLMaestros database.`
	`CREATE TABLE [SQLMaestros].[Table1](`
	`Column1 INT IDENTITY,`
	`Column2 INT);`
	`GO`
	`-- Create a table Table2 in SQLMaestros database.`
	`CREATE TABLE [SQLMaestros].[Table2](`
	`Column1 INT IDENTITY,`
	`Column2 INT,`
	`);`
	`GO`
	`-- Populate Table1 and Table2 with 5 records each.`
	`INSERT INTO [SQLMaestros].[Table1] VALUES(100)`
	`GO 5`
	`INSERT INTO [SQLMaestros].[Table2] VALUES(200)`
	`GO 5`
	`----------`
	`-- End: Setup`
	`----------`

Begin the first transaction	**Execute** the following statement(s) in a new query window (first transaction). Press Ctrl+N to open a new query window or click on **New Query** on the standard toolbar in SSMS

```
-- Step 1: Open a new query window(first transaction) and
execute the below query
USE SQLMaestros
GO

BEGIN TRAN
UPDATE SQLMaestros.Table1
SET Column2=20
WHERE Column2=100
```

Explanation: In the above query, we start a new explicit transaction and execute an update statement within the context of the transaction. The transaction will acquire exclusive (X) locks on the rows matching the filter criteria. Note that we have neither committed nor rolled back the transaction, which means, the transaction is still open. This is important to simulate the deadlock.

View the lock details

Execute the following statement(s)in the original window to view the lock details of the first transaction

```
-- Step 2: View lock details of the first transaction
SELECT resource_type,
DB_NAME(resource_database_id) AS DatabaseName,
resource_description,request_mode,
request_type,
request_status
FROM
sys.dm_tran_locks
WHERE DB_NAME(resource_database_id) = 'SQLMaestros'
```

Results | Messages

	resource_type	DatabaseName	resource_description	request_mode	request_type	request_status
1	DATABASE	SQLMaestros		S	LOCK	GRANT
2	DATABASE	SQLMaestros		S	LOCK	GRANT
3	DATABASE	SQLMaestros		S	LOCK	GRANT
4	PAGE	SQLMaestros	1:228	IX	LOCK	GRANT
5	RID	SQLMaestros	1:228:0	X	LOCK	GRANT
6	RID	SQLMaestros	1:228:1	X	LOCK	GRANT
7	RID	SQLMaestros	1:228:2	X	LOCK	GRANT
8	RID	SQLMaestros	1:228:3	X	LOCK	GRANT
9	RID	SQLMaestros	1:228:4	X	LOCK	GRANT
10	OBJECT	SQLMaestros		IX	LOCK	GRANT

Observation: The first transaction acquires exclusive locks on all the rows of Table1. Note the **X** in request_mode column which has **GRANT** status in request_status column.

Begin the second transaction

Execute the following statement(s) in a new query window (second transaction)
Press Ctrl+N to open a new query window or click on **New Query** on the standard toolbar in SSMS

```
-- Step 3: Open another query window(second transaction)
and execute the below query
```

```
USE SQLMaestros
GO

BEGIN TRAN
UPDATE SQLMaestros.Table2
SET Column2=10
WHERE Column2=200

UPDATE SQLMaestros.Table1
SET Column2=20
WHERE Column2=100
```

Observation: The second explicit transaction fires two UPDATE statements. The first one exclusively locks all the rows of Table2 similar to the first transaction which still holds the locks on Table1. The second UPDATE statement attempts to lock all the rows of Table1 but as we know that rows in Table1 are already locked by the first transaction, the second transaction has to wait. Also, note that this transaction is also open just like the first transaction

View the lock details

Execute the following statement(s) to view the lock details of the second transaction

```
-- Step 4: View the lock details again
SELECT resource_type,
DB_NAME(resource_database_id) AS DatabaseName,
resource_description,request_mode,
request_type,
request_status
FROM
sys.dm_tran_locks
WHERE DB_NAME(resource_database_id) = 'SQLMaestros'
ORDER BY request_status DESC
```

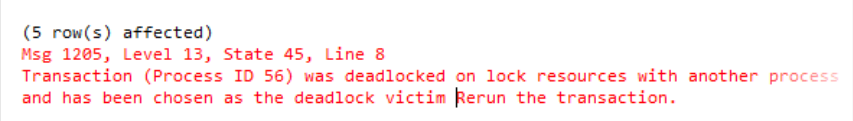
Results | Messages

	resource_type	DatabaseName	resource_description	request_mode	request_type	request_status
1	RID	SQLMaestros	1:228:0	U	LOCK	WAIT
2	RID	SQLMaestros	1:230:0	X	LOCK	GRANT
3	RID	SQLMaestros	1:228:1	X	LOCK	GRANT
4	RID	SQLMaestros	1:230:1	X	LOCK	GRANT
5	RID	SQLMaestros	1:228:2	X	LOCK	GRANT
6	RID	SQLMaestros	1:230:2	X	LOCK	GRANT
7	RID	SQLMaestros	1:228:3	X	LOCK	GRANT
8	RID	SQLMaestros	1:230:3	X	LOCK	GRANT
9	RID	SQLMaestros	1:230:4	X	LOCK	GRANT
10	RID	SQLMaestros	1:228:4	X	LOCK	GRANT

Observation: We can see that the second transaction has taken a few **X** locks and is now waiting to acquire **U**-lock.

Execute a UPDATE statement in the first transaction

Execute the following statement(s) in the query window of the first transaction

	1. Copy/paste the below code in the query window of the first transaction.
	2. Highlight/select it
	3. And execute. Make sure that only the above query is highlighted/selected in SSMS before you **EXECUTE** or press **F5**
	4. After execution, flip between the first and the second transaction windows to see which one has been chosen as the deadlock victim
	
	`-- Step 5: execute below query in the first transaction.`
	`-- Copy/paste the below code in the query window of the first transaction, highlight/select it and execute`
	`-- One of the transactions will terminate with a deadlock error`
	`UPDATE SQLMaestros.Table2`
	`SET Column2=10`
	`WHERE Column2=200`
	`ROLLBACK TRAN`
	
	Note: Make sure that only the above query is highlighted/selected in SSMS before you **EXECUTE** or press **F5**
	
	 
	`(5 row(s) affected)`
	`Msg 1205, Level 13, State 45, Line 8`
	`Transaction (Process ID 56) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.`
	
	Explanation: One of the transactions has been terminated with the above error. It could be the first one or the second one. The one terminated is the deadlock victim. A cyclic deadlock phenomenon was created where each transaction held locks on their respective rows and tried to access the others in a cyclic fashion resulting in a deadlock. The victim transaction is automatically rolled back. In the next step, we will manually roll back the other transaction.

Rollback the non-victim transaction	**Execute** the following statement(s) in the query window of the transaction which was not rolled back.
	
	1. Copy/paste the below code in the query window of the transaction which was not rolled back
	2. Highlight/select it
	3. Execute. Make sure that only the above query is highlighted/selected in SSMS before you **EXECUTE** or press **F5**
	
	`-- Step 6: Rollback the non-victim transaction`
	`-- Copy/paste the below code in the query window of the transaction which was not the victim, highlight/select it and execute`

	ROLLBACK TRAN

Summary

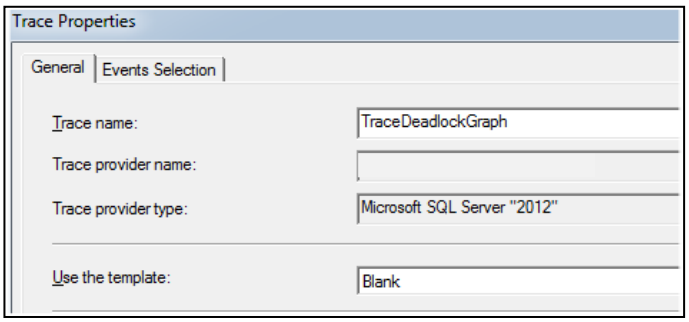
In this exercise, we have learned:

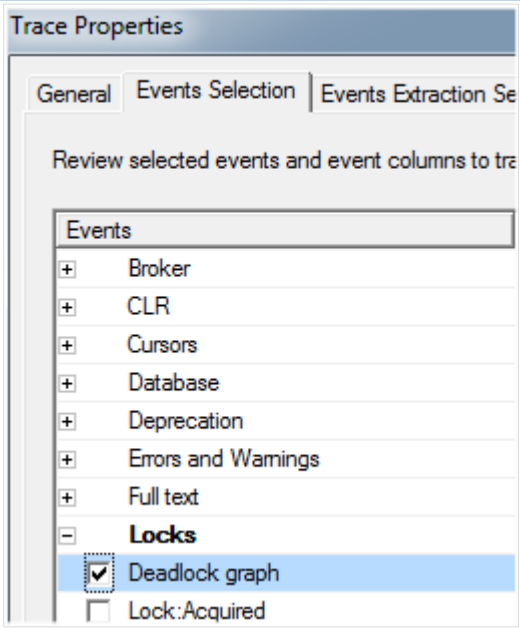
- What is a deadlock
- How to create a cyclic deadlock scenario

Exercise 2: Deadlock Detection

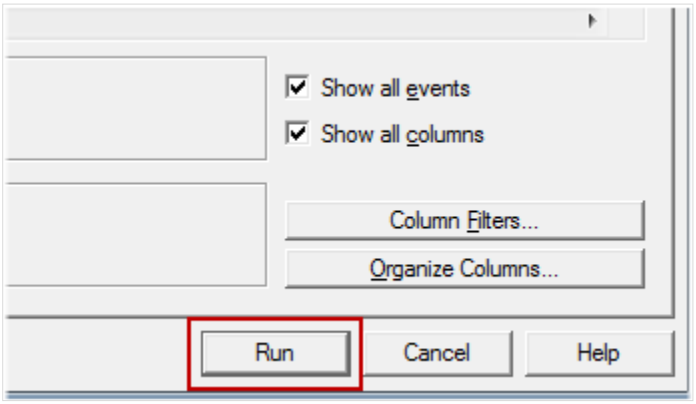
Scenario

In this exercise, we will learn to detect deadlock using SQL Server Profiler and System Health Extended Event.

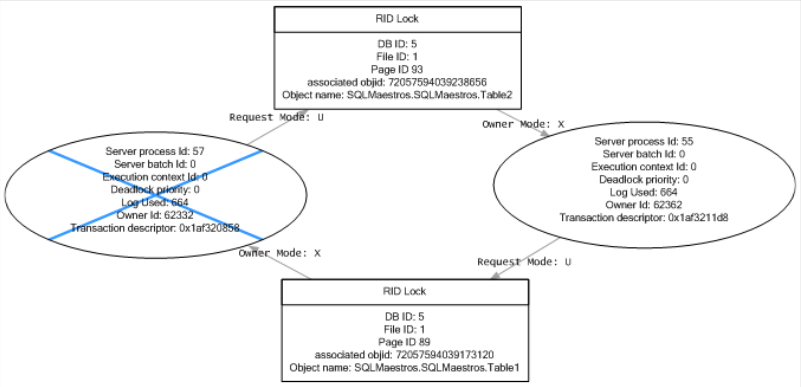
Tasks	Detailed Steps
Setup **SQL Server Profiler** to record the deadlock graph	1. Click **Start** | **All Programs** | **SQL Server 2012** | **SQL Server Management Studio| Performance Tools | SQL Server Profiler** **2.** Click **File** | **New Trace** | **Connect** OR **Ctrl + N** | **Connect** 3. **General** | **Trace Name: TraceDeadlockGraph** | **Use The Template: Blank**  4. **Events Selection** | **Events : Locks : Deadlock Graph**



5. Click **Run** to start the profiler.
6. Follow the instructions in **Exercise 1** to simulate a deadlock scenario which will get in profiler trace.
7. The deadlock is recorded in SQL Profiler as shown below.



8. **Right Click | Deadlock graph | Extract Event Data | Save DeadlockGraph as .xdl file**



Observation: The deadlock graph is thus recorded and can be saved in a **.xdl/.xml** format for later analysis.

Open 2_DeadlockDetailsFromSystemHealth.sql

1. Click **File** | **Open** | **File** or press (Ctrl + O)
2. In **Open File** dialogue box, navigate to **SQL Server Deadlock Basics\Scripts** folder
3. Select **2_DeadlockDetailsFromSystemHealth.sql** and click **Open**

Get Deadlock Details from system health extended event

Execute the following statement(s) to **view** deadlock XML from system health extended event

```
-- Step 1: Execute below query to check what all
information system health extended event captures
SELECT * FROM sys.dm_xe_session_eventse
INNER JOIN sys.dm_xe_sessionss
ON e.event_session_address=s.ADDRESS
```

	event_session_address	event_name
1	0x00000001B48B0041	spinlock_backoff_warning
2	0x00000001B48B0041	resource_monitor_ring_buffer_recorded
3	0x00000001B48B0041	latch_suspend_warning
4	0x00000001B48B0041	stack_trace
5	0x00000001B48B0041	bad_memory_detected
6	0x00000001B48B0041	bad_memory_fixed
7	0x00000001BA3D6141	xml_deadlock_report
8	0x00000001BA3D6141	wait_info_external

Observation: The system health extended event is a default extended event much like the default trace. It captures details required to troubleshoot SQL Server issues.

Get Deadlock Details from system health extended event	**Execute** the following statement(s) to **view** deadlock XML being recorded by System Health Extended Event
	
	`-- Step 2: Execute below query to deadlock XML`
	`SELECT`
	`  xed.value('@timestamp', 'datetime') as Deadlock_Creation_Date,`
	`  xed.query('.') AS Deadlock_XML`
	`FROM`
	`(`
	`  SELECT CAST([target_data] AS XML) AS Target_Data`
	`  FROM sys.dm_xe_session_targets AS xt`
	`  INNER JOIN sys.dm_xe_sessions AS xs`
	`  ON xs.ADDRESS = xt.event_session_address`
	`  WHERE xs.name = N'system_health'`
	`  AND xt.target_name = N'ring_buffer'`
	`) AS XML_Data`
	
	`CROSS APPLY Target_Data.nodes('RingBufferTarget/event[@name="xml_deadlock_report"]') AS XEventData(xed)`
	`ORDER BY Deadlock_Creation_Date DESC`
	
	 
	
	Observation: The system health extended lists all the deadlocks occurred in **XML** format. The **XML** can be later analyzed to troubleshoot deadlocks.

Save Deadlock XML as deadlock Graph	1. **Click** on XML under Deadlock_XML Column. The XML will open in a new query window 2. Remove Event, data and value elements from XML file 3. Click **File** | **Save As** | **Deadlock_XML.xml** 4. Click **File** | **Save As** | **Deadlock_Graph.xdl**

	Deadlock_Creation_Date	Deadlock_XML
1	2014-03-07 23:06:45.183	<event name="xml_deadlock_report" package="sqlse...
2	2014-03-07 23:03:57.620	<event name="xml_deadlock_report" package="sqlse...
3	2014-03-07 23:02:55.117	<event name="xml_deadlock_report" package="sqlse...
4	2014-03-07 23:01:57.610	<event name="xml_deadlock_report" package="sqlse...
5	2014-03-07 22:18:39.880	<event name="xml_deadlock_report" package="sqlse...

Summary

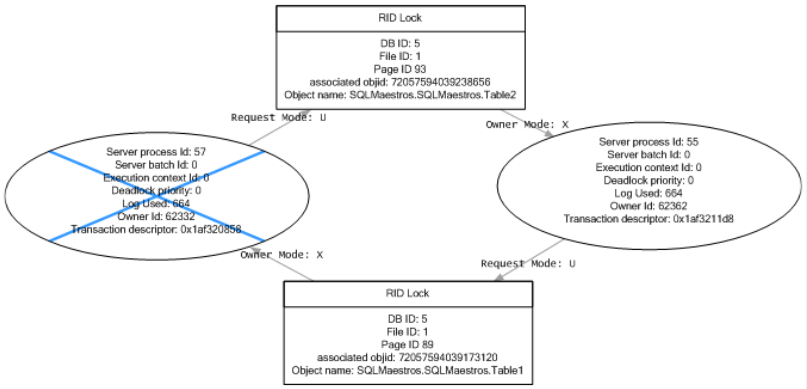
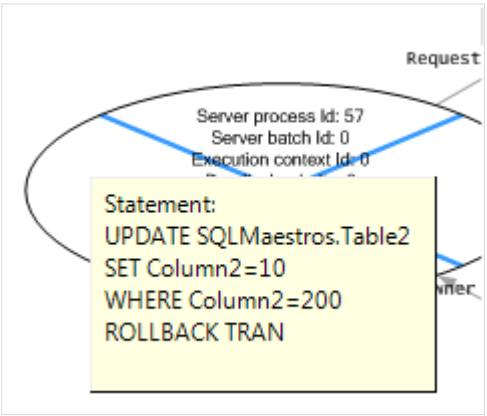
In this exercise, you have learned:

- Setup profiler to capture deadlock graph
- Get deadlock XML from system health extended event
- Save deadlock XML as deadlock graph (.xdl)

Exercise 3: Understanding Deadlock Graph

Scenario

In this exercise, we will look at deadlock graph recorded in Exercise 2

Tasks	Detailed Steps
Open **DeadlockGraph.xdl**	1. Click **File** | **Open** | **File** or press (Ctrl + O) 2. In **Open File** dialogue box, navigate to **SQL Server Deadlock Basics\Scripts** folder 3. Select **DeadlockGraph.xdl** and click **Open**
Understanding Deadlock Graph	 Take the mouse pointer over an oval to get the statement involved in the deadlock.  **Explanation:** The session id 57 owns an exclusive lock on Table2 (owner mode) and is requesting update lock (request mode) on Table1. The session 55 owns an exclusive lock on Table1 (owner

	mode) and is requesting update lock on Table2. This results in a deadlock situation and session id 57 is chosen as a deadlock victim and is forcefully terminated by SQL Server.

Summary

In this exercise, we have learned:

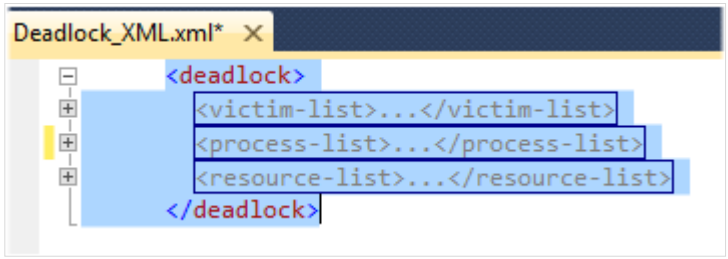
- Different components of a deadlock graph
- To infer queries participating in a deadlock

Exercise 4: Understanding Deadlock XML

Scenario

In this exercise, we will look at deadlock XML recorded in Exercise 2

Tasks	Detailed Steps
Open **DeadlockGraph.xml**	1. Click **File** | **Open** | **File** or press (Ctrl + O) 2. In **Open File** dialogue box, navigate to **SQL Server Deadlock Basics\Scripts** folder 3. Select **Deadlock_XML.xml** and click **Open**

Understanding Deadlock XML	XML consists of 3 main elements
	 
	
	1. **victim-list:** The **Victim-List** Lists all the process ids chosen as the deadlock victim
	`<victim-list>`
	`<victimProcessid="process1af094188" />`
	`</victim-list>`
	2. **Process-list:** The process-list lists all the process/sessions participating in a deadlock. It also lists down queries running under each session during deadlock.
	`<process-list>`
	`<processid="process1af094188"taskpriority="0"logused="664"`

```
                              waitresource="RID:
5:1:89:0"waittime="5185"ownerId="91564"

      transactionname="user_transaction"lasttranstarte
d="2014-03-08T04:36:39.910"

      XDES="0x1b1810d28"lockMode="U"schedulerid="3"kpi
d="3232"status="suspended"

      spid="56"sbid="0"ecid="0"priority="0"trancount="
2"
                              lastbatchstarted="2014-03-
08T04:36:39.910"
                              lastbatchcompleted="2014-03-
08T04:35:39.657"
                              lastattention="1900-01-
01T00:00:00.657"
                              clientapp="Microsoft SQL
Server Management Studio - Query"
                              hostname="AHMAD-
PC"hostpid="5956"loginname="Ahmad-PC\Ahmad"
                              isolationlevel="read
committed (2)"
                              xactid="91564"currentdb="5"
                              lockTimeout="4294967295"

      clientoption1="671090784"clientoption2="390200">
<inputbuf>
-- Step 2: Open another query window(Connection 2) and
execute below query
BEGIN TRAN

UPDATE SQLMaestros.Table2
SET Column2=10
WHERE Column2=200

UPDATE SQLMaestros.Table1
SET Column2=20
WHERE Column2=100

ROLLBACK TRAN
</inputbuf>
```

3. **Resource-list:** The **Resource-list** lists type of resources (Key/page/Object) involved in a deadlock. It also lists all the processes owning/waiting on a particular resource.

```
<resource-list>
<ridlockfileid="1"
                              pageid="89"
                              dbid="5"

      objectname="SQLMaestros.SQLMaestros.Table1"
                              id="lock1b6ac8a80"
                              mode="X"

      associatedObjectId="72057594039173120">
<owner-list>
<ownerid="process1b1634928"mode="X" />
```

	</owner-list> <waiter-list> <waiterid="process1af094188"mode="U"requestType="wait" /> </waiter-list> </ridlock>
Cleanup	**Execute** the following **script** in **Cleanup** section -- CLEANUP USE[master] GO ALTER DATABASE[SQLMaestros]SET SINGLE_USER WITH ROLLBACK IMMEDIATE; GO DROP DATABASE[SQLMaestros]; GO

Summary

In this exercise, we have learned:

- Understanding deadlock XML
- What are victim-list, process-list, and resource-list

Questions? Comments? Talk to the author today. SQLMaestros on Twitter.

About SQLMaestros Hands-On-Labs

SQLMaestros Hands-On-Labs are packaged in multiple volumes based on roles (DBA, DEV & BIA). Each lab document consists of multiple exercises and each exercise consists of multiple tasks.

LEARN MORE

Want to write for the magazine? Comments? Feedback? Reach out to us at magazine@sqlservergeeks.com

DPS 2020

FREE CONTENT

The DPS Team has released all DPS 2020 sessions for the worldwide data community. You can have free access and watch the sessions on-demand.

Sessions highlights for this month

How Data Engineers and Data Scientist can get started with Azure Synapse Analytics, Azure Data Factory and Azure Machine Learning **by Wee Hyong Tok & Nellie Gustafsson**

Azure Data Studio Above and Beyond **by Warwick Rudd**

Global Analytics with Azure Cosmos Db and Synapse Analytics **by Warner Chaves**

Inside Computer Vision by Varsha Parthasarathy

Advanced insides into System Versioned Temporal Tables **by Uwe Ricken**

Microsoft SQL Server - In-memory OLTP Design Principle **by Torsten Strauss**

Building Enterprise grade bot in no time **by Sudhir Rawat**

Adopting a DevOps Process for your Database **by Steve Jones**

Blogging for the Tech Professional **by Steve Jones**

How to Remove Bias and Make Machine Learning Models Fair & Discrimination-Free at the Example of Credit Risk Data **by Sray Agarwal**

Designing for high performance in Power BI **by Siva Harinath** & **Philip Seamark**

Machine Learning - Best Practices and Vulnerabilities **by Sebastiano Galazzo**

Accelerating Enterprise Analytics Solutions with Azure Synapse Analytics **by Saveen** Reddy

Understanding NULLs in SQL Server **by Satya Ramesh**

Azure Arc Enabled SQL Server **by Sasha Nosov**

Inside OCR and Form Recognizer **by Sanjeev Jagtap**

Blast to The Future: Accelerating Legacy SSIS Migrations into Azure Data Factory **by Sandy Winarko**

Responsible ML – A Peek into Model Interpretability **by Sandeep Alur**

Storing and Searching Files in SQL Server FileTables **by Sam Nasr**

Notebooks, PowerShell and Excel Automation **by Rob Sewell**

Learn More

DPS 2021 Training Class | Recordings

8-hours video course (recorded class) available for $149.5 only

Execution plans in depth
Hugo Kornelis

Time to Learn Dax
Markus Ehrenmuller-Jensen

PowerShell for the DBA - Let's Do It!
Ben Miller

Azure Synapse Analytics - From data ingestion to visualization
Wolfgang Strasser

Data Engineering Using ADF
Abhishek Narain
Sunil Sabat
Linda Wang

SQL Managed Instance
Niko Neugebauer
Vitor Tomaz
Vitor Pombeiro
Nevena Nikolic

10 Rules A Database Developer Must Know
Uwe Ricken

Demystifying Design Principles and Building Impactful Reports in Power BI
Reid Havens

Migration & Modernization To Azure
Ajay Jagannathan
Raj Pochiraju
Rajesh Setlem
Alexandra Ciortea
Mohamed Kabiruddin

The Future of Deployment for Modern Data Platform Applications
Ben Weissman &
Anthony Nocentino

Big Data & Analytics With SQL Server 2019 Big Data Cluster
Niels Berglund

SQL Server Performance Tuning – Fast Track
Amit Bansal

End of December Edition