JULY 2022

SQLServerGeeks MAGAZINE

The Bimonthly Dossier For SQL Professionals

A Tip About Isnull And Coalesce
by Erland Sommarskog

SQL Server-The Concept of Selectivity
by SQLMaestros
(Hands-On-Labs)

Auditing Azure SQL Database
by Sravani Saluru

Test SQL Server Backups And Offload Integrity Checks

by Josephine Bush

SQLServerGeeks

Magazine

July 2022 edition of the SQLServerGeeks Magazine is here. Previous editions combined, we have published almost 50+ articles by 25+ renowned authors. The magazine content has been covering the vast surface area of SQL Server and we sincerely hope it has helped in solving real-world challenges. We hope that the tenth edition of the SSG Magazine is as helpful to all the SQL Server enthusiasts.

By now you all are aware that the SSG Magazine is a bi-monthly publication (once in two months). We have another bi-monthly magazine for Data professionals - the DataPlatformGeeks (DPG) Magazine. Both the magazines combined, you are getting loads of free learning content each month. The SSG Magazine covers the SQL world and, the DPG Magazine covers the vast technology landscape of the data world. Download your DPG Magazine now!

For this release, a massive shoutout to all the authors - Erland Sommarskog (A Tip About Isnull and Coalesce), Josephine Bush (Test SQL Server Backups And Offload Integrity Checks), and Sravani Saluru (Auditing Azure SQL Database).

Another big news: Data Platform Virtual Summit 2022 registrations are happening in full swing. The largest online learning event for data professionals is free. DPS 2022 has five editions covering the entire globe. 12 Pre-Cons to choose from. Book your seat today.

Don't forget to give us your feedback so we can continue providing you the quality content, well-curated to your interests. If you're 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

COPYRIGHT STATEMENT

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

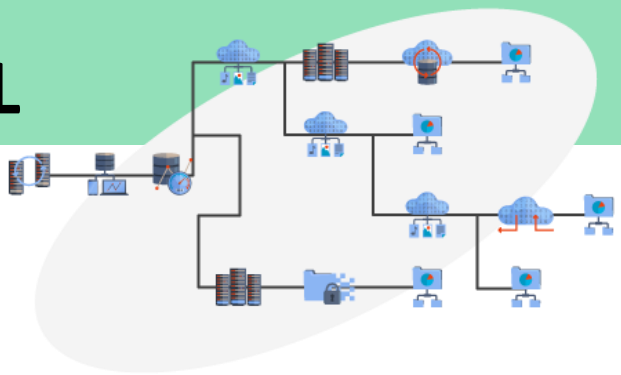
"Our heartiest congratulations to all the team members & authors of the SQLServerGeeks Magazine - we have completed one year. It has been a very humbling experience. Kudos to all the authors for their immense support & dedication. Special thanks to our advertisers - you keep us going. And last but not the least, you, our reader & subscriber - we do this for you and you have encouraged us so much along the way.

Thank you for your continuous feedback.

Thank you all once again - more SQL goodness to come."

TABLE OF CONTENTS

A TIP ABOUT ISNULL AND COALESCE



Erland Sommarskog

In this issue of the SQLServerGeeks Magazine, we shall take a look at ISNULL() and COALESCE(), which at first glance seem to do the same thing:

```
CREATE TABLE tbl (id    int         NOT NULL,
                  value varchar(20) NULL)
INSERT tbl(id, value)
      VALUES(1, NULL), (2, 'A value')
SELECT id, ISNULL(value, 'No value given') AS ISNULL,
     COALESCE(value, 'No value given') AS "COALESCE"
FROM   tbl
ORDER  BY id
```

This is the result set:

```
id          ISNULL              COALESCE
----------- ------------------- -------------------
1           No value given      No value given
2           A value             A value
```

That is, they both return the first argument as long as it is not a NULL value. If the first argument is NULL, you get back the second argument instead.

And, indeed, for many simple cases, the two are interchangeable. However, there are subtle differences that you should be aware of, and that is the topic of this article. One such subtle difference can be seen in the example: COALESCE is in quotes and ISNULL is not. I will come back to why it is so.

Let's start with ISNULL. ISNULL is a *function*, which takes exactly two arguments. The data type of the return value is the same as the data type of the first argument. This can sometimes bite you:

```
DECLARE @a varchar(5) = NULL
SELECT ISNULL(@a, 'MISSING!')
```

This returns *MISSI*. (Because the first argument is **varchar(5)** and only permits for five characters.)

Here is a mistake that beginners sometimes do. Rather seeing NULL, they want to see a blank string, which leads to:

```
DECLARE @i int, @d datetime2(3), @n numeric(12,2)
SELECT ISNULL(@i, '') AS int
SELECT ISNULL(@d, '') AS datetime2
SELECT ISNULL(@n, '') AS numeric
```

The output is:

```
int
-----------
0
datetime2
---------------------------
1900-01-01 00:00:00.000
numeric
---------------------------------------
Msg 8114, Level 16, State 5, Line 109
Error converting data type varchar to numeric.
```

For the first two examples, it happens to be the case that an empty string can be converted to **int** and **datetime**, so the conversion succeeds. But maybe the value was not what they wanted to see. On the other hand, it is not legal to convert the empty string to **numeric**. (Why are the rules not the same for **int** and **numeric**? I don't know, but inconsistency is a hallmark of SQL Server.

> **Note:** if you really want to get a blank string back in case of NULL, you need to convert the first argument to string. However, in most cases, it is better to return NULL and let the presentation layer to take care of the formatting.

Occasionally, you may have more than one alternate value. To deal with this situation with ISNULL, you need to nest two calls of ISNULL:

```
SELECT ISNULL(thiscol, ISNULL(thatcol, 0))
```

Also worth noticing is that ISNULL is a proprietary function and generally not supported by other database products than Microsoft SQL Server (and Sybase).

Let's now move to COALESCE, which is part of the ANSI standard and which you can expect to be available on most database engines. By the ANSI standard, COALESCE is a reserved keyword, and this is why I had to put it in quotes to use it as a column alias.

In difference to ISNULL, COALESCE is not limited to two arguments, but it can take any number of arguments, so if we have two alternate values, we can say:

```
SELECT COALESCE(thiscol, thatcol, 0)
```

So, in difference to ISNULL, there is no need to nest calls. In passing, while this is certainly is useful, I find that I only need this a few times per year. I'm not sure if I ever had used COALESCE with four or more values in the list.

While COALESCE may look like a function, it is not. COALESCE(expr1, expr2, ..., exprN) is a syntactic shortcut for a CASE expression:

```
CASE WHEN expr1 IS NOT NULL THEN expr1
     WHEN expr2 IS NOT NLUL THEN expr2
     ...
     WHEN exprN-1 IS NOT NULL THEN exprN-1
     ELSE exprN
END
```

This has some consequences that you need to be aware of. One is about data types. Since COALESCE is a CASE expression, the values in the list are on equal footing, and the first value has no precedence for determining the data type of the return value. Instead, if the list has a mix of data types, SQL Server will apply its *Data type precedence list*. That is, the return type will the type with the highest precedence.

This can work to our advantage. Consider:

```
DECLARE @a varchar(5) = NULL
SELECT COALESCE(@a, 'MISSING!')
```

This returns *MISSING!*. No truncation this time, since in the mix of **varchar(5)** and **varchar(9)**, the latter takes precedence.

But casual use of COALESCE can lead us into trouble. The examples with ISNULL above are no different with COALESCE:

```
DECLARE @i int, @d datetime2(3), @n numeric(12,2)
SELECT COALESCE(@i, '') AS int
SELECT COALESCE(@d, '') AS datetime2
SELECT COALESCE(@n, '') AS numeric
```

The result is the same as above. This is because strings are fairly low on the precedence list. An abridged version of the precedence list is binary – strings – numbers – dates. This means that strings that you supply to be used in place of NULL most of the time will be converted to the data type of the first expression in the list (assuming the normal case of two expressions).

If you get the idea of substituting a NULL value in a string column with some other data type, you can run into trouble with COALESCE. Say this, you have a table with time given in two columns. One column

is a string column giving the time in text, whereas the other uses the **time** data type. You want to use the string column if available, else you fall back to the **time** column.

```
DECLARE @t TABLE (id       int         NOT NULL PRIMARY KEY,
                  timetext varchar(20) NULL,
                  timeval  time(0)     NULL)

INSERT @t (id, timetext, timeval)
 VALUES (1, NULL, '09:43:23'),
        (2, NULL, '19:12:00'),
        (3, 'Noon', '12:00:00'),
        (4, 'Quarter past five', '17:15:00')
SELECT id, COALESCE(timetext, timeval) AS time
FROM   @t
```

This results in:

```
id          time
----------- ----------------
1           09:43:23
2           19:12:00
Msg 241, Level 16, State 1, Line 152
Conversion failed when converting date and/or time from character
string.
```

Because **time** has higher precedence than **varchar**, conversion goes in that direction, which leads to a conversion failure. Thus, if you want do something like this, you need to convert everything to strings:

```
SELECT id, COALESCE(timetext, convert(char(8), timeval, 108)) AS
time
FROM   @t
```

In this case, a simpler fix would be to use ISNULL instead.

The last trap with COALESCE is the most devilish. What do think about these two? Are they entirely equivalent:

```
CREATE TABLE tbl2(col int NOT NULL)
SELECT ISNULL((SELECT min(col) FROM tbl2), 0),
       COALESCE((SELECT min(col) FROM tbl2), 0)
```

It may seem so, but they are not. In one window run this:

```
DECLARE @i int = 1000
WHILE @i > 0
BEGIN
   INSERT tbl2(col) VALUES(88)
   WAITFOR DELAY '00:00:00.003'
   DELETE tbl2
   WAITFOR DELAY '00:00:00.003'
   SET @i -= 1
END
```

That is, the script repeatedly adds and deletes a row into the table with a short delay in between.

While the script above is running, open a second window to the script below. This script inserts the results of the query above into a temp table in a tight loop without delays and finally it returns a summary of the temp table.

```
DROP TABLE IF EXISTS #result
CREATE TABLE #result(id         int NOT NULL,
                     ISNULL     int NULL,
                     "COALESCE" int NULL)
DECLARE @i int = 200000
WHILE @i > 0
BEGIN
   INSERT #result(id, ISNULL, "COALESCE")
      SELECT @i, ISNULL((SELECT min(col) FROM tbl2), 0),
                 COALESCE((SELECT min(col) FROM tbl2), 0)
   SET @i -= 1
END
SELECT ISNULL, "COALESCE", COUNT(*)
FROM   #result
GROUP  BY ISNULL, "COALESCE"
ORDER  BY ISNULL, "COALESCE"
```

The exact result will vary from test to test, but here is one result I got:

```
ISNULL      COALESCE    count
----------- ----------- -----------
0           0           106409
0           88          35
88          NULL        22
88          0           42
88          88          93492
```

You may be puzzled to see that COALESCE has returned NULL a few times. How is that possible? Elementary, my dear Watson. Internally, `COALESCE((SELECT min(col) FROM tbl2), 0)` is rewritten as

```
CASE WHEN (SELECT min(col) FROM tbl2) IS NOT NULL
        THEN (SELECT min(col) FROM tbl2)
     ELSE 0
END
```

That is, the subquery is evaluated twice. When the WHEN condition is evaluated, there is a row in the table, which sends the CASE to the THEN branch. But on the second evaluation, the row that was there when the WHEN was evaluated has gone away, and whence the CASE – and, thus the COALESCE – evaluates to NULL.

Note: make sure that you run the demo in tempdb. If you run it in a database which has READ_COMMITTED_SNAPSHOT enabled, you will not see any NULL values, because the implementation of this mode. It is possible, though, to construct a demo that produces NULL also in a database that is in READ_COMMITTED_SNAPSHOT by using a user-defined function that does not inline.

Beside the surprise of getting NULL values, there is another issue lurking here: if the query is complex, using COALESCE instead of ISNULL adds extra load to the system because of the double execution.

To summarise: we have looked at the differences between ISNULL and COALESCE. We have seen that COALESCE is nice in the few cases we have more than one alternate value. Both ISNULL and COALESCE can surprise you when it comes to data types, COALESCE possible a little more than ISNULL. But the big thing is when your first expression is a subquery. In this case, you should always use ISNULL to avoid that the subquery is evaluated twice, which can have impact both for performance and correctness. If you from prefer to generalize this to decide that henceforth you will always use ISNULL and never use COALESCE, I will not come after you. After all, ISNULL is easier to spell.

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 likes travelling, listening to music and he plays bridge about once a week. In summertime he loves to go with his bicycle in the areas around Stockholm to one of the many lakes.

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

AB SQL SERVER NOTES

NOTE 01

Continuing my efforts on penning down some notes on SQL Server IO, here is the most critical (probably) weapon in your arsenal - PAGEIOLATCH wait type - identifying workloads that are waiting on page read IO completion.

READ MORE...

NOTE 02

What is an Index Range Scan? Let's say you have a clustered index on the OrderDate column. When you have a predicate that says WHERE OrderDate > 01/01/2022 ORDER BY OrderDate, the optimizer performs a range scan.

READ MORE...

NOTE 03

Did you know that you can get multiple missing index hints in your execution plan? When you switch over to the execution plan tab, you can see one missing index hint on the top (in green color).

READ MORE...

NOTE 04

When you are tuning queries, there are two fundamental oncepts that you need to be well aware of - Density & Selectivity. Today, let's alk about Selectivity. In simple words, it means, how selective your predicates are.

READ MORE...

LEARN MORE

TEST SQL SERVER BACKUPS AND OFFLOAD INTEGRITY CHECKS

Josephine Bush | @hellosqlkitty

In this article, you will learn about restoring your SQL Server backups to verify them. You will also learn how to offload integrity checks to your restore server. **You must test your SQL Server backups by restoring them.** It's not enough to do CHECKSUM when backing up your databases and VERIFYONLY when restoring.

CHECKSUM is an option available when backing up your database. CHECKSUM specifies that the backup operation verifies each page for checksum and torn page. If enabled and available, it generates a checksum for the entire backup. VERIFYONLY is an option available when restoring your database backup. It verifies the backup but does not restore it. It checks to see that the backup set is complete and the entire backup is readable.

These are helpful, yes, but they don't guarantee that you can restore your databases. The only way you can know if your database backup will restore is to actually restore it. **I do this on a separate restore server setup for this purpose alone** – to test backups by restoring them**. I also offload the integrity checks for large databases to this restore server.**

The specs on your restore server need to be based on how large your backups are. Mine are as follows:

- **CPU:** 4 cores
- **RAM:** 64GB
- **Disks:**
    - **Data/log drive:** 12TB – I have multiple, large databases. I don't want to be limited to restoring one at a time, so I have one large drive. I don't worry about separating data/log files for restoring, so they all go on one drive together. They will be dropped after restoring and doing an integrity check.
    - **System databases drive:** 10GB – I like keeping the system databases separate on all SQL Server installations
    - **TempDB drive:** 500GB – I made this somewhat large to accommodate multiple integrity checks running concurrently

Also, I use the most current version of SQL Server on my restore server. I don't match the version of SQL Server that the backup was taken on.

Restoring Your Databases

To begin with, I use Ola scripts to backup my databases. He has great scripts for this. You can get this script and more information about it here: https://ola.hallengren.com/sql-server-backup.html

Now that my backups are taken, I need to restore them to be absolutely sure they are restorable. **I use sp_DatabaseRestore to restore the backups.** This stored procedure assumes you are using Ola's backup scripts to backup your database. You could modify sp_DatabaseRestore to suit your own backup script/method, but why? Ola's backup script is great, so make the switch to it if you aren't already using it.

You can get a copy of sp_DatabaseRestore here: https://github.com/BrentOzarULTD/SQL-Server-First-Responder-Kit/blob/dev/sp_DatabaseRestore.sql

For more information about how to use sp_DatabaseRestore with examples, visit: https://www.brentozar.com/archive/2017/03/databaserestore-open-source-database-restore-stored-procedure/

The following script shows you **how I use sp_DatabaseRestore**:

```
/*****************************************************
DO NOT USE IN PRODUCTION unless you want to overwrite the prod DB with a restore backup
*****************************************************/

EXEC dbo.sp_DatabaseRestore
@Database = 'dbname',
@BackupPathFull = '\\serverpath \prod\servername\dbname\FULL\',
@BackupPathDiff = '\\serverpath \prod\servername\dbname\DIFF\',
@BackupPathLog = '\\serverpath \prod\servername\dbname\LOG\',
@RestoreDiff = 1,
@ContinueLogs = 0,
@RunRecovery = 1
```

Here is an explanation of each of the parameters from the previous script:

- **I always leave that giant warning at the top.** If you use this on a production system you can overwrite a database you didn't mean to. That's one of the nice things about having this script ready, though. You can use it in a production emergency if you need to restore from backup, but be very careful with it.
- **@Database** – Set this to what you want the restored database to be named
- **@BackupPathFull** – Set this to the path of your FULL backups. This path is the Ola backup path format.
- **@BackupPathDiff** – Set this to the path of your DIFF backups.
- **@BackupPathLog** - Set this to the path of your LOG backups.
- **@RestoreDiff** – Set to 1 because I want to restore a DIFF backup.
- **@ContinueLogs** – Set to 0 because I don't plan to restore more LOG backups later.
- **@RunRecovery** – Set to 1 because I want to bring the database online to run an integrity check.

Note: You don't have to restore diff and log backups if you don't want to. I like to restore everything I have up to that point with the restore script, but this is up to you. In my experience, the most likely

backup to be corrupted is the full backup, so I always restore those. Once you have the full backup restored, it's quite fast to restore the diff and log backups. That's why you might as well restore them before running recovery on the restored database. Plus, this way you can do an integrity check on the most recent data from your backups.

Integrity Check on Your Restored Database

It is very important to run a full integrity check at least once per week, also known as DBCC CHECKDB. A lot of my production databases are quite large and they may not have enough TempDB space to accommodate a full integrity check. This is why I offload these to my restore server.

For integrity checks, I also use Ola scripts. You can get the script and more information about the script here: https://ola.hallengren.com/sql-server-integrity-check.html

The following script shows you **how to run an integrity check**:

```
EXECUTE [dbo].[DatabaseIntegrityCheck]
@Databases = 'EWP',
@LogToTable = 'Y'
```

Here is an explanation of each of the parameters from the previous script:

- **@Databases** – Set this to the restored database name. You can also run this against multiple databases, but I like to do one at a time. This is because I put the restore script and the integrity check script in one job.
- **@ LogToTable** – Set to 'Y' for Yes. This will log the commands run to the dbo.CommandLog table. This table will be set up in the same database where you added the sp_DatabaseIntegrityCheck stored procedure.

Schedule Restores and Integrity Checks

To make sure my backups are restored regularly, I schedule an agent job on the restore server. I have one job per-database and three steps in each job:

- **Restore step** – Use the sp_DatabaseRestore script
- **Integrity check step** – Uses the sp_DatabaseIntegrityCheck script
- **Drop database step** – To free up room for the next database restores

I schedule this job making sure its start time is as close as possible to when the full backup finishes. This way if there is an issue with the full backup restore I'll know about it as soon as possible. Most of my full backups happen on the weekend and during off-hours. Then I have the restore job alert on failure so that I know at the earliest possible convenient time if all is not well. I don't check for failed restore alerts on the weekend, though, I save it until when I'm back in the office.

The following script gives you the agent job I would setup:

```
USE [msdb]
GO

        BEGIN TRANSACTION
        DECLARE @ReturnCode INT
        SELECT @ReturnCode = 0
        IF NOT EXISTS (SELECT name FROM msdb.dbo.syscategories WHERE
        name=N'[Uncategorized (Local)]' AND category_class=1)
```

```
        BEGIN
        EXEC @ReturnCode = msdb.dbo.sp_add_category @class=N'JOB', @type=N'LOCAL',
        @name=N'[Uncategorized (Local)]'
        IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback
        END

DECLARE @jobId BINARY(16)
EXEC @ReturnCode =  msdb.dbo.sp_add_job @job_name=N'dbname restore and integrity
check',

                @enabled=1,
                @notify_level_eventlog=0,
                @notify_level_email=2,
                @notify_level_netsend=0,
                @notify_level_page=0,
                @delete_level=0,
                @description=N'No description available.',
                @category_name=N'[Uncategorized (Local)]',
                @owner_login_name=N'sa',
                @notify_email_operator_name=N'sqlalerts', @job_id = @jobId OUTPUT

IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback

EXEC @ReturnCode = msdb.dbo.sp_add_jobstep @job_id=@jobId, @step_name=N'restore
dbname',

                @step_id=1,
                @cmdexec_success_code=0,
                @on_success_action=3,
                @on_success_step_id=0,
                @on_fail_action=2,
                @on_fail_step_id=0,
                @retry_attempts=0,
                @retry_interval=0,
                @os_run_priority=0, @subsystem=N'TSQL',
                @command=N'/**********************************************
                *****

                        DO NOT USE IN PRODUCTION. unless you want to restore
from backup

                        **********************************************
                *****/

                        EXEC dbo.sp_DatabaseRestore
                        @Database = ''dbname'',
                        @BackupPathFull = ''\\serverpath
                        \prod\servername\dbname\FULL\'',
                        @BackupPathDiff = ''\\serverpath
                        \prod\servername\dbname\DIFF\'',
```

```
                                        @BackupPathLog = ''\\serverpath
                                        \prod\servername\dbname\LOG\'',
                                        @RestoreDiff = 1,
                                        @ContinueLogs = 0,
                                        @RunRecovery = 1',

            @database_name=N'master',
            @flags=0

IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback

EXEC @ReturnCode = msdb.dbo.sp_add_jobstep @job_id=@jobId, @step_name=N'integrity
check dbname',

            @step_id=2,
            @cmdexec_success_code=0,
            @on_success_action=3,
            @on_success_step_id=0,
            @on_fail_action=2,
            @on_fail_step_id=0,
            @retry_attempts=0,
            @retry_interval=0,
            @os_run_priority=0,
            @subsystem=N'TSQL',
            @command=N'EXECUTE [dbo].[DatabaseIntegrityCheck]

                        @Databases = ''dbname'',
                        @LogToTable = ''Y''',
                        @database_name=N'master',

            @flags=0

IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback

EXEC @ReturnCode = msdb.dbo.sp_add_jobstep @job_id=@jobId, @step_name=N'drop
dbname after successful integrity check',

            @step_id=3,
            @cmdexec_success_code=0,
            @on_success_action=1,
            @on_success_step_id=0,
            @on_fail_action=2,
            @on_fail_step_id=0,
            @retry_attempts=0,
            @retry_interval=0,
            @os_run_priority=0, @subsystem=N'TSQL',
            @command=N'drop database dbname; ',
            @database_name=N'master',
            @flags=0
```

```
            IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback

    EXEC @ReturnCode = msdb.dbo.sp_update_job @job_id = @jobId, @start_step_id = 1

    IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback

    EXEC @ReturnCode = msdb.dbo.sp_add_jobschedule @job_id=@jobId, @name=N'restore-
    dbcc-schedule',

                @enabled=1,
                @freq_type=8,
                @freq_interval=1,
                @freq_subday_type=1,
                @freq_subday_interval=0,
                @freq_relative_interval=0,
                @freq_recurrence_factor=1,
                @active_start_date=20200804,
                @active_end_date=99991231,
                @active_start_time=101500,
                @active_end_time=235959,
                @schedule_uid=N'4556d8eb-807e-40a1-acba-bf0bd5c5075e'

    IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback

    EXEC @ReturnCode = msdb.dbo.sp_add_jobserver @job_id = @jobId, @server_name =
    N'(local)'

    IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback

    COMMIT TRANSACTION

    GOTO EndSave

    QuitWithRollback:

      IF (@@TRANCOUNT > 0) ROLLBACK TRANSACTION

    EndSave:

    GO
```

Some important things to note about the restore job script above:

- **Setup the job with the script and then go in and change the schedule based on when your backup finishes running.** You will want to schedule it as close to the full backup completion as you can within reason. This may mean that no diff or log backups are restored. That's OK because the main thing is testing your full backup and getting a proper integrity check done on it.
- **Make sure to update all the job steps to correctly reference the restored database**
- **Make sure to update the paths to your backups**

If you do this for all your production database backups, you will be in very good shape. You can retake a backup if the previous one is corrupt before too much time passes. Plus, you can get an integrity check done on large databases without impacting your live database system.

Questions? Comments? Talk to the author today. Josephine Bush on Twitter.

About Josephine Bush

Josephine Bush has over 10 years of experience as a Database Administrator. Her experience is extensive and broad-based, including in financial, business, and energy data systems using SQL Server, MySQL, Oracle, and PostgreSQL.

LEARN MORE

Non-Tech World of Josephine Bush

During her free time, Josephine loves knitting, painting, and yoga.

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

SQL NUGGETS BY MICROSOFT

Released: SCOM Management Packs for SQL Server, RS, AS (7.0.38.0)

Hotfix: Microsoft Drivers 5.10.1 for PHP for SQL Server Released

Security Update for SQL Server 2016 SP2 GDR

SQL Server 2022 public preview is now available on Linux.

Hotfix: JDBC Driver 10.2.1 for SQL Server Released

Cumulative Update #16 for SQL Server 2019 RTM

AUDITING AZURE SQL DATABASE

Sravani Saluru | @sravanisaluru

Auditing is the most important feature for your databases in cloud platform, it's important to know how to configure, view and maintain auditing data for your SQL database. In this blog we will talk about how to configure audit for Azure SQL Database.

Like SQL Server, Azure SQL database also supports both server level auditing and database level auditing.

When server level auditing is enabled, it's enabled for all existing databases and new databases which will be created in future. So, it's important to understand when to enable server audit and database audit.

Enable Server audit when you must audit all databases for that logical server, enable database level audit when you want audit different action groups for a specific database or write to different target for a specific database.

If you are enabling both server and database level audit for a database, then you can choose predicate expression to filter the events to ensure you are not capturing duplicate data.

Server level and database level auditing can be enabled from portal and when you enable from portal the below audit action groups will be enabled by default.

Default Action groups for Azure SQL database:

BATCH_COMPLETED_GROUP This event is raised whenever any batch text, stored procedure, or transaction management operation completes executing. It is raised after the batch completes and will audit the entire batch or stored procedure text, as sent from the client, including the result.

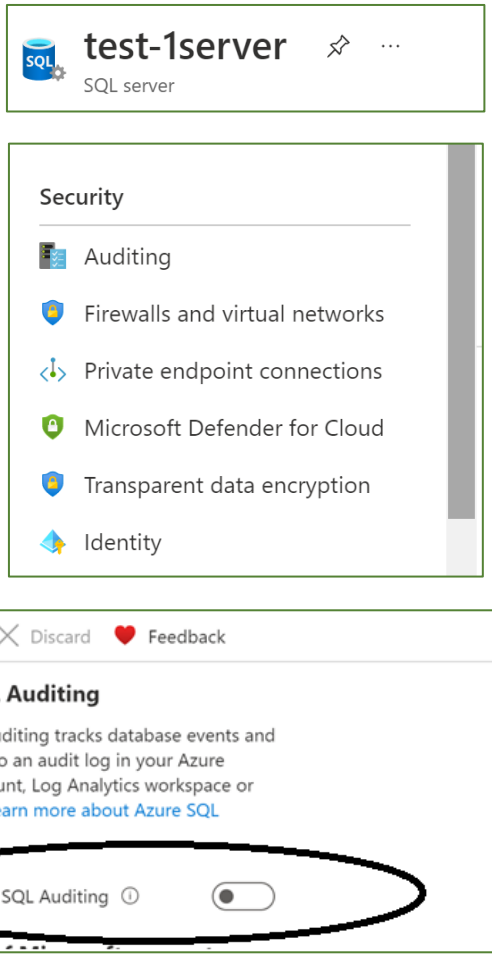
FAILED_DATABASE_AUTHENTICATION_GROUP Indicates that a principal tried to log on to a contained database and failed. Events in this class are raised by new connections or by connections that are reused from a connection pool

SUCCESSFUL_DATABASE_AUTHENTICATION_GROUP Indicates that a principal successfully logged in to a contained database.

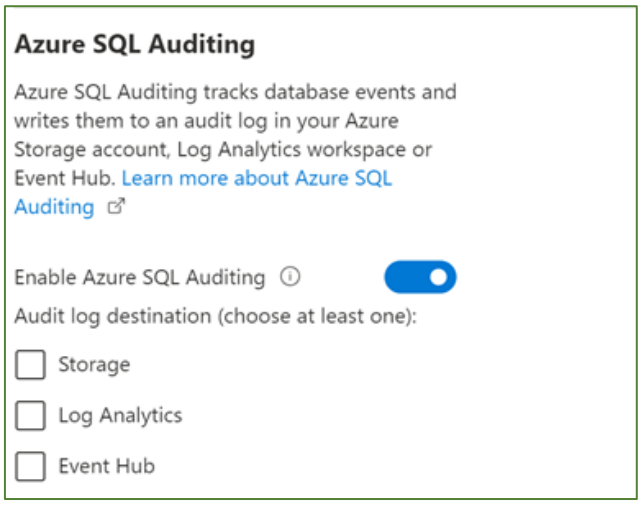
For more audit action groups please refer Audit action groups.

Server level auditing:

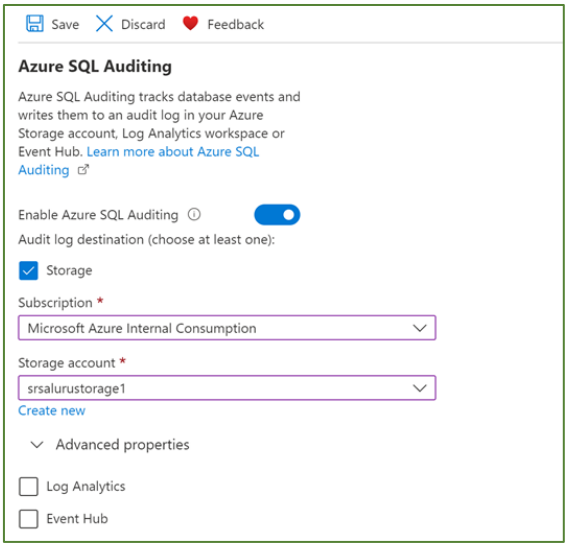
From Azure portal, go to the server blade and select Enable Azure SQL auditing. This will enable the server level audit for all databases



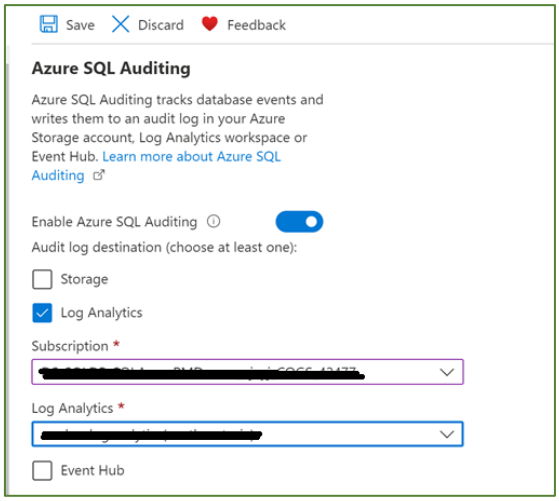
When you enable server level audit you can choose the log destination as Storage, log analytics or Event Hub.



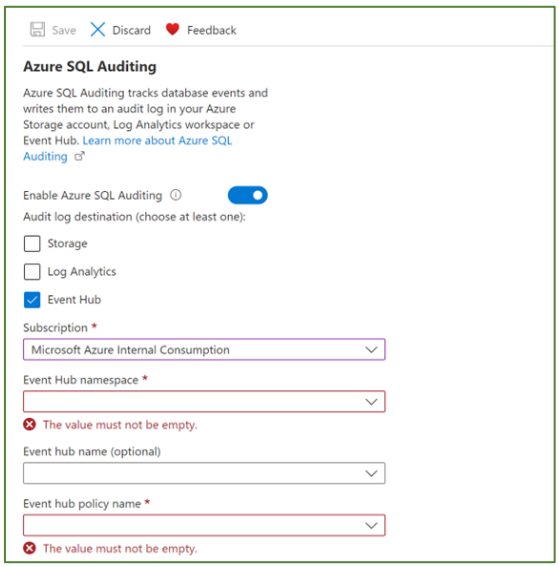
Enable Server audit to storage account



Enable auditing to Log analytics

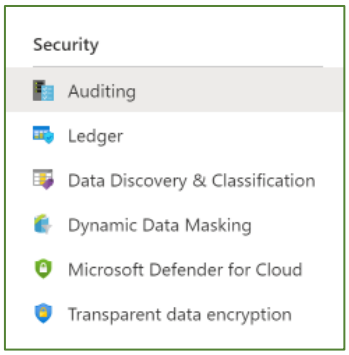


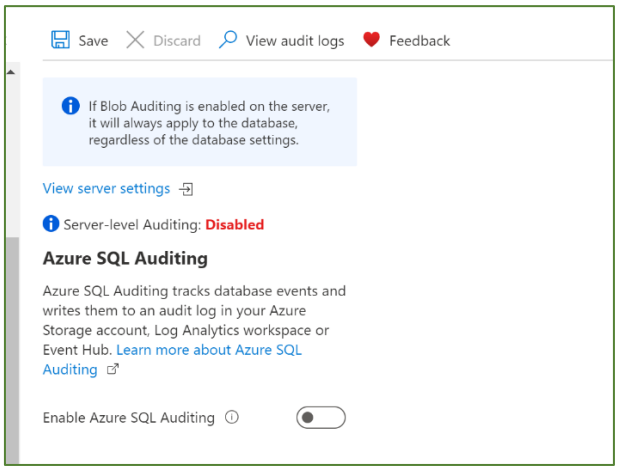
Enable auditing to Event hub



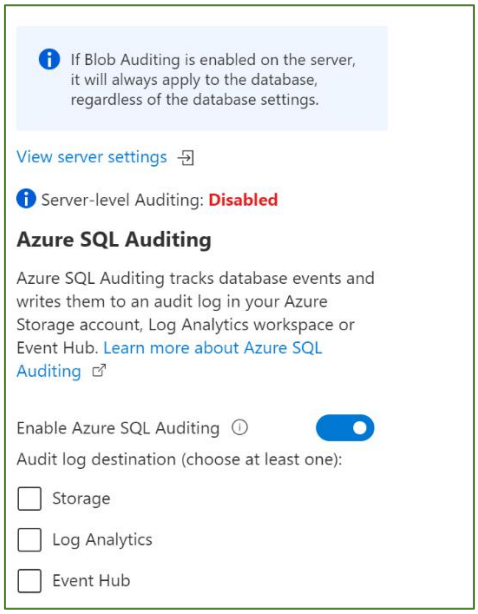
Database Level auditing:

To enable database level audit, In the azure portal go to database **security** blade -> **auditing** and select enable Azure SQL auditing. This will enable database level auditing. in the below screen shot you can see server level audit is disabled. but for your server if server level audit is enabled then you must check the reason why you still need database level audit.





For database level audit also, you can choose audit log destination as storage , Log analytics or Event hub.



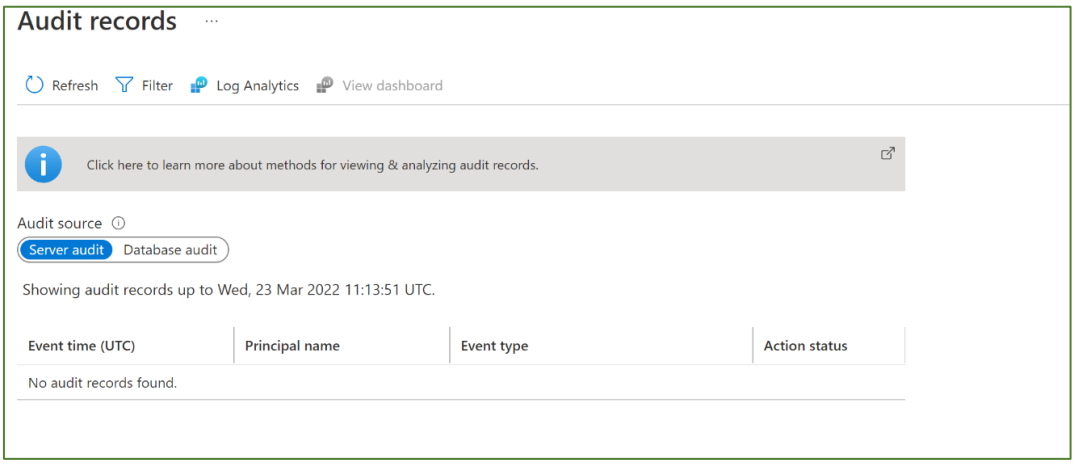
Auditing Microsoft support operations:

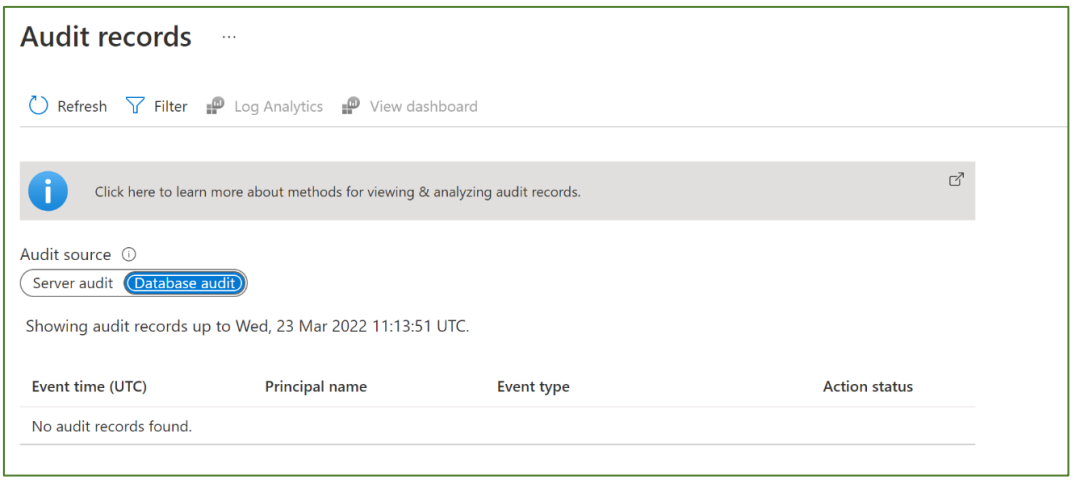
You can also audit microsoft support operations as we maintain your databases. Any action that's performed by devops can be audited and written to the audit log for your reference.

From portal , server auditing blade you can select enable auditing microsoft support operations and use different audit log destinations.

View Audit Logs:

Once auditing is enabled the server logs and database logs can be viewed from the portal.





In this blog we discussed enabling default server and database audit for Azure SQL database and view audit logs using azure portal. In the next blog we will discuss about using PowerShell cmdlets to modify, overwrite audit settings to use more precise action groups or filter actions /filter schemas based on your business needs.

Questions? Comments? Talk to the author today. Sravani Saluru on Twitter.

About Sravani Saluru

I am Sravani. I am working as a senior software engineer for Microsoft Mysql Engineering team. before joining the opensource team, I have 13 years of experience in Microsoft SQL Server. I worked as an Escalation Engineer in CSS Data & AI team and have been associated with Microsoft for past 9 years. I worked closely with product group, field engineers and our top customers on multiple critical situations. Problem solving is my cup of tea and I am a kepner Tregoe Program leader for problem solving and decision-making techniques. I am community a speaker and delivered multiple sessions in various community events in India and drive Technical sessions for Data & AI user groups for last couple of years. Performance tuning, Replication and High availability are my areas of expertise in Microsoft SQL Server.

LEARN MORE

Non-Tech World of Sravani Saluru

During her free time she loves to cook and explore new dishes most of time she is occupied with her 3 year old boy.

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

PIVOTING IN SQL SERVER

SQLMaestros Hands-On-Labs | @SQLMaestros

Estimated time to complete this lab

50 minutes

Objectives:

After completing this lab, we will learn:

- Manual pivoting
- Pivoting using **PIVOT** operator
- Pivot operator internals
- Dynamic pivoting
- Performance difference between pivoting using **PIVOT** operator and manual pivoting

Lab Setup Requirements

Before executing this lab:

- You must have SQL Server 2012 Standard/Developer/Enterprise/Evaluation edition or higher. Click here to download SQL Server evaluation edition
- You must have SQLMaestros sample database. Click here to download SQLMaestros sample database

Prerequisites

Before executing this lab:

- It is recommended that you have basic experience with SQL Server
- You have met the Lab Setup Requirements mentioned above

Lab Scenario

To understand and analyze the data in more efficient way, we may have to rotate rows into columns and columns into rows. The rotation of rows to columns is called pivoting and the reverse operation is called unpivoting. This lab explains pivoting and unpivoting operation using manual coding using **CASE** statement(s) and by using **PIVOT** and **UNPIVOT** operators. This lab is divided into four exercises. In the first exercise, we will learn the concept of pivoting. In the second exercise, we will learn the

pivoting and grouping in the presence of **IDENTITY** column and also will learn dynamic pivoting. In the third exercise, we will learn the performance differences between pivoting using **CASE** statement(s) and **PIVOT** operator. In the fourth and final exercise, we will learn to unpivot the pivoted data.

Tips to complete this lab successfully

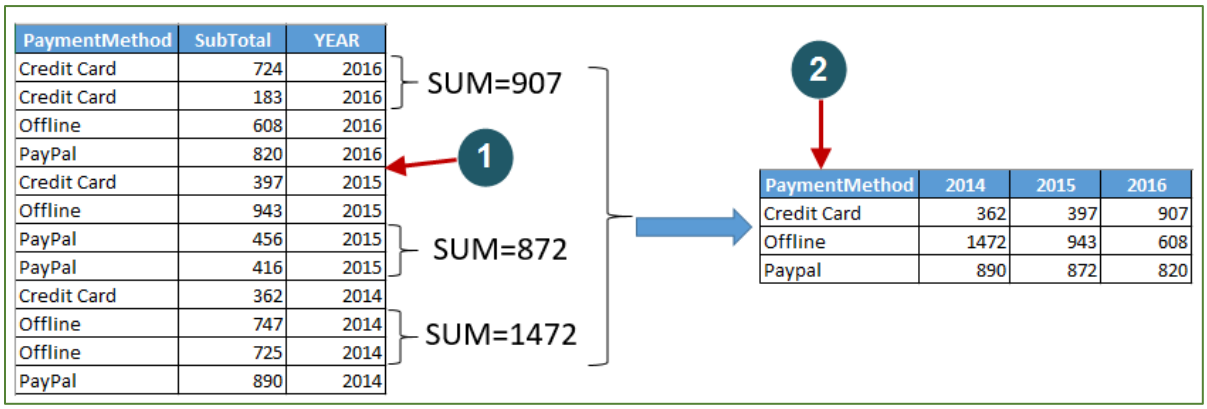
Following these tips will be helpful in completing the lab successfully in time

- All lab files are located in **Pivoting in SQL Server** folder
- The script(s) are divided into various sections marked with 'Begin', 'End' and 'Steps'. As per the instructions, execute the statements between particular sections only or for a particular step
- Read the instructions carefully and do not deviate from the flow of the lab
- Practice this lab only in your test machine/environment. Do not run this lab in your production environment

Exercise 1: Introduction to Pivoting

Overview:

Pivoting means transforming rows into columns. Pivot query assists us to produce interactive table that combines and compares large amount of data. Image 1 in the below snapshot, lists down the payment made under different payment methods each year. Observe that, in the year **2016** the amount paid through credit card method is **907** (724+183) and so on. If we want to summarize the subtotal per payment method per year, we need to sum up the subtotal per payment method per year. In order to visualize this easily, pivot the rows into columns based on YEAR column as shown by the image 2 in the below snapshot. It is easier to analyze lab subtotal per payment method per year using image 2.



Scenario

In this exercise, we will look at different components of extended events.

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_Introduction.sql**	1. Click **File** | **Open** | **File** or press (**Ctrl + O**). 2. In **Open File** dialogue box, navigate to **Pivoting in SQL Server/Scripts** folder. 3. Select **1_Introduction.sql** and click **Open.**

View the data	Execute the following statements(s) to review the data
	
	`--Step 1: Execute the following statement(s) to review the data in TransactionDetails table`
	`USE SQLMaestros;`
	`GO`
	`SELECT PaymentMethod,`
	`SubTotal,`
	`YEAR(OrderDate) AS [YEAR]`
	`FROM [SQLMaestros].[hol].[TransactionDetails]`
	`ORDER BY YEAR(OrderDate) DESC,`
	`PaymentMethod;`
	
	PaymentMethod | SubTotal | YEAR
	Credit Card | 378.8869 | 2016
	Credit Card | 385.3442 | 2016
	Credit Card | 195.5056 | 2016
	Credit Card | 442.138 | 2016
	Credit Card | 604.9409 | 2016
	
	Note: The above image is a trimmed snapshot of the original result set. The original result set contains more number of rows.
	
	Observation: The above query retrieves the transaction details of subscribers, who have subscribed for Hands-On-Labs. Observe that the **PaymentMethod** column contains different payment methods such as Credit Card, PayPal, and Offline use to make payment when purchasing the Hands-On-Labs subscription(scroll down through result set in SSMS to observe all payment methods). **SubTotal** column contains the transaction amount and the **YEAR** column contains transaction year details.

Aggregate the data	Execute the following statement(s) to aggregate the data to find the total labs sales amount per year per payment method
	
	`-- Step 2: Execute the following statement(s) to write query to aggregate the data as per the business requirement`
	`SELECT PaymentMethod ,`
	`SUM(SubTotal) SubTotal,`
	`YEAR(OrderDate) AS [YEAR]`
	`FROM [SQLMaestros].[hol].[TransactionDetails]`
	`GROUP BY PaymentMethod,`
	`YEAR(OrderDate)`
	`ORDER BY YEAR(OrderDate) DESC,`
	`PaymentMethod;`

PaymentMethod	SubTotal	YEAR
Credit Card	153556.1265	2016
Offline	155777.1503	2016
Paypal	157826.982	2016
Credit Card	185267.131	2015
Offline	179381.2192	2015
Paypal	170344.6133	2015

Note: The above image is a trimmed snapshot of the original result set. The original result set contains more number of rows.

Explanation: The above query retrieved the total labs sales details per year per payment method. Observe that, in the year **2016**, the total sales through **Credit Card** method is **153556.1265**, through **Offline** is **155777.1503** and through the **Paypal** is **157826.982**. Scroll down through the result set in SSMS to observe lab sales details for remaining years. To analyze the output efficiently, the result set should be pivoted, which will be explained in the later steps.

Pivoting without **PIVOT** operator

Execute the following statement(s) to perform pivot without using **PIVOT** operator

```
-- Step 3: Execute the following statement(s) to perform Pivot
without using the pivot operator
SELECT TransactionID,PaymentMethod,OrderDate,
CASE
       YEAR(OrderDate) WHEN 2016 THEN SubTotal ELSE 0 END AS
'2016',
       CASE
       YEAR(OrderDate) WHEN 2015 THEN SubTotal ELSE 0 END AS
'2015',
       CASE
       YEAR(OrderDate) WHEN 2014 THEN SubTotal ELSE 0 END AS
'2014',
       CASE
       YEAR(OrderDate) WHEN 2013 THEN SubTotal ELSE 0 END AS
'2013',
       CASE
       YEAR(OrderDate) WHEN 2012 THEN SubTotal ELSE 0 END AS
'2012',
       CASE
       YEAR(OrderDate) WHEN 2011 THEN SubTotal ELSE 0 END AS
'2011',
       CASE
       YEAR(OrderDate) WHEN 2010 THEN SubTotal ELSE 0 END AS
'2010',
       CASE
       YEAR(OrderDate) WHEN 2009 THEN SubTotal ELSE 0 END AS
'2009'
FROM hol.TransactionDetails
ORDER BY PaymentMethod
```

TransactionID	PaymentMethod	OrderDate	2016	2015	2014	2013	2012	2011	2010	2009
1	Credit Card	2009-01-01 00:00:00.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	746.3452
2	Credit Card	2009-01-02 00:00:00.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	607.5474
8	Credit Card	2009-01-08 00:00:00.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	819.6584
10	Credit Card	2009-01-10 00:00:00.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	396.8175
13	Credit Card	2009-01-13 00:00:00.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	943.2763
23	Credit Card	2009-01-23 00:00:00.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	415.8763

Note: The above image is a trimmed snapshot of the original result set. The original result set contains more number of rows.

Explanation: The above query pivoted the data using **CASE** statement(s). Observe that, the sales amount of few records is **0.00** as there is no order placed on that particular **OrderDate**. Scroll down through the result set in SSMS to observe the lab sales details for all the years. To retrieve the aggregated sales details per year per payment method, we need to sum up the data for each year by performing grouping on **PaymentMethod** column.

Perform aggregation

Execute the following statement(s) to sum up the individual case statement(s) to get aggregated data

```
--Step 4: Execute the statement(s) to Sum up the individual
case statement to get the aggregated data
SELECT  PaymentMethod,
SUM(CASE
      YEAR(OrderDate) WHEN 2016 THEN SubTotal ELSE 0 END) AS
'2016',
SUM(CASE
      YEAR(OrderDate) WHEN 2015 THEN SubTotal ELSE 0 END) AS
'2015',
SUM(CASE
      YEAR(OrderDate) WHEN 2014 THEN SubTotal ELSE 0 END) AS
'2014',
SUM(CASE
      YEAR(OrderDate) WHEN 2013 THEN SubTotal ELSE 0 END) AS
'2013',
SUM(CASE
      YEAR(OrderDate) WHEN 2012 THEN SubTotal ELSE 0 END) AS
'2012',
SUM(CASE
      YEAR(OrderDate) WHEN 2011 THEN SubTotal ELSE 0 END) AS
'2011',
SUM(CASE
      YEAR(OrderDate) WHEN 2010 THEN SubTotal ELSE 0 END) AS
'2010',
SUM(CASE
      YEAR(OrderDate) WHEN 2009 THEN SubTotal ELSE 0 END) AS
'2009'
FROM    hol.TransactionDetails
GROUP BY PaymentMethod
GO
```

PaymentMethod	2016	2015	2014	2013	2012	2011	2010	2009
Offline	155777.1503	179381.2192	164466.1686	172816.6914	231785.4597	247249.0987	224551.6442	219711.3608
Credit Card	153556.1265	185267.131	202949.4373	192290.1197	220553.8572	250976.0652	271399.4337	259257.2316
Paypal	157826.982	170344.6133	184943.0984	172473.0546	247092.5496	239686.5805	232642.3448	250432.4497

	Observation: The above query retrieved the lab sales details per payment method per year by rotating rows into columns using **CASE** statement(s). This result set is easy to analyze when compared to the result set we got at **step 2**.

Perform pivoting using **PIVOT** operator	Execute the following statement(s) to perform pivoting using **PIVOT** operator
	
	`-- Step 5: Execute the following statement(s) to perform pivoting using PIVOT operator`
	`SELECT PaymentMethod, [2016], [2015], [2014], [2013], [2012], [2011], [2010], [2009]`
	`FROM ( SELECT PaymentMethod, SubTotal, YEAR(OrderDate) AS [YEAR] FROM [SQLMaestros].[hol].[TransactionDetails] ) AS sq PIVOT( SUM(SubTotal) FOR [YEAR] IN ( [2016], [2015], [2014], [2013], [2012], [2011], [2010], [2009] ) ) AS Pvt;`
	
	PaymentMethod | 2016 | 2015 | 2014 | 2013 | 2012 | 2011 | 2010 | 2009
	Offline | 155777.1503 | 179381.2192 | 164466.1686 | 172816.6914 | 231785.4597 | 247249.0987 | 224551.6442 | 219711.3608
	Credit Card | 153556.1265 | 185267.131 | 202949.4373 | 192290.1197 | 220553.8572 | 250976.0652 | 271399.4337 | 259257.2316
	Paypal | 157826.982 | 170344.6133 | 184943.0984 | 172473.0546 | 247092.5496 | 239686.5805 | 232642.3448 | 250432.4497
	
	Explanation: The above query also retrieved the labs sales details per payment method per year using **PIVOT** operator. The syntax for the **PIVOT** operator is as follows:
	
	`SELECT <non-pivoted column>,`
	`<pivoted columns>`
	`FROM`
	`(<SELECT query that produces the data>)`
	`AS <alias for the source query>`
	`PIVOT`
	`(`
	`<aggregation function>(<column being aggregated>)`
	`FOR`
	`[<column that contains the values that will become column headers>]`
	`IN (<pivoted columns>)`
	`) AS <alias for the pivot table>`
	`<optional ORDER BY clause>;`

	Note: The above image represents annotated syntax of **PIVOT** operator, in which purpose of each keyword\step explained properly.
Dynamic pivoting	In cases where we do not know all the values we need to pivot on, dynamic pivoting is used. Execute the following statement(s) altogether to perform dynamic pivot operation

```
-- Step 6: Execute the following statement(s) to perform
dynamic pivoting
-- Query 1: Declare required variables
DECLARE @DynamicPivotQuery AS NVARCHAR(MAX);
DECLARE @ColumnName AS NVARCHAR(MAX);
-- Query 2: To Find distinct years and select them to
@ColumnName variable
SELECT    @ColumnName = ISNULL(@ColumnName + ',', '') +
QUOTENAME([YEAR])
FROM    ( SELECT DISTINCT
                  YEAR(OrderDate) AS YEAR
          FROM    [SQLMaestros].[hol].[TransactionDetails]
        ) AS Pvtsource;

-- Query 3: Dynamic pivot query
SET  @DynamicPivotQuery  =  N'SELECT  PaymentMethod,  '  +
@ColumnName + '
    FROM ( SELECT    PaymentMethod,
                  SubTotal,
                  YEAR(OrderDate) AS [YEAR]
          FROM    [SQLMaestros].[hol].[TransactionDetails]
        ) AS sq
    PIVOT( SUM([Subtotal]) FOR [YEAR] IN (' + @ColumnName +
')) AS pvt';

-- Query 4: Execute the Dynamic Pivot Query
EXEC sp_executesql @DynamicPivotQuery;
```

PaymentMethod	2010	2013	2016	2014	2011	2012	2009	2015
Offline	209849.4753	195602.5965	207141.4283	187000.4577	187787.8103	204225.3347	211318.9065	192812.7836
Credit Card	229192.4148	231779.9614	184728.8356	212467.0785	234201.928	239557.4171	195616.5736	208705.1932
Paypal	206771.8093	243385.758	200358.5406	213371.3324	179199.2783	180558.5099	203239.0167	228557.4277

Explanation: The above query retrieved the lab sales details per payment method per year. **Query1** in the above query declares the required variables to work with. **Query2** will select distinct order years into **@ColumnName** variable. The @ColumnName variable contains the row values which are to be converted to columns when pivoting the data. **Query3** selects the actual pivot query into **@DynamicPivotQuery** variable. **Query4** will execute **@DynamicPivotQuery** variable which contains dynamic string. Observe that **@ColumnName** is used at pivoted columns instead of hardcoding the year values.

Close all the query windows	Close all the query windows ( ) and if **SSMS** asks to save changes, click **NO**

Exercise 2: Pivot Operator Internals

Scenario

In this exercise, we will look into internals of pivot operator and understand how the pivot is implemented by pivot operator.

Tasks	Detailed Steps
Open **2_PivotingAndGrouping.sql**	1. Click **File** | **Open** | **File** or press (**Ctrl + O**) 2. In **Open File** dialogue box, navigate to **Pivoting in SQL Server/Scripts** folder 3. Select **2_PivotingAndGrouping.sql** and click **Open**

Create a table	Execute the following statement(s) to create a **tpivot** table
	
	`--Step 1: Create a table`
	`USE SQLMaestros;`
	`GO`
	`IF ( OBJECT_ID('tpivot') ) IS NOT NULL`
	`  DROP TABLE tpivot;`
	`GO`
	`CREATE TABLE tpivot`
	`  (`
	`  ID INT,`
	`  [key] NVARCHAR(MAX),`
	`  [Value] SQL_VARIANT`
	`  );`
	`GO`

Insert records	Execute the following statement(s) to insert the records into **tpivot** table
	
	`-- Step 2: Insert the records into tpivot table`
	`INSERT INTO tpivot`
	`VALUES ( 1, 'key1', 'ABC' );`
	`INSERT INTO dbo.tpivot`
	`VALUES ( 1, 'key2', '2016-10-01' );`
	`INSERT INTO dbo.tpivot`
	`VALUES ( 1, 'key3', 123 );`
	`GO`
	`INSERT INTO tpivot`
	`VALUES ( 2, 'key1', '123' );`
	`INSERT INTO dbo.tpivot`
	`VALUES ( 2, 'key2', 'ABC' );`
	`INSERT INTO dbo.tpivot`
	`VALUES ( 2, 'key3', '2016-10-01' );`

	GO

View the records	Execute the following statement(s) to view the records
	
	`-- Step 3: View the records in tpivot table`
	`SELECT * FROM dbo.tpivot;`
	
	ID | key | Value
	1 | key1 | ABC
	1 | key2 | 2016-10-01
	1 | key3 | 123
	2 | key1 | 123
	2 | key2 | ABC
	2 | key3 | 2016-10-01
	
	Observation: The above query retrieves the data present in **tpivot** table. Observe that, **Key1** in **key** column has **Value** field as **ABC** and **123** with ID **1** and **2** respectively. To view the values representing each **key** in a better way, we need to pivot the data.

Manual pivoting	Execute the following statement(s) to perform manual pivoting on **tpivot** table
	
	`-- Step 4: Execute the following statement(s) to perform manual pivot`
	`SELECT ID ,`
	`CASE WHEN [key] = 'key1' THEN Value`
	`END AS [key1],`
	`CASE WHEN [key] = 'key2' THEN Value`
	`END AS [Key2],`
	`CASE WHEN [key] = 'key3' THEN Value`
	`END AS [Key3]`
	`FROM dbo.tpivot;`
	
	ID | key1 | Key2 | Key3
	1 | ABC | NULL | NULL
	1 | NULL | 2016-10-01 | NULL
	1 | NULL | NULL | 123
	2 | 123 | NULL | NULL
	2 | NULL | ABC | NULL
	2 | NULL | NULL | 2016-10-01
	
	Observation: The above query pivoted the data. The NULL values in the above output represents the corresponding key value are not present. For example, **Key1** has only one value with ID **1** i.e., **ABC**, so remaining values are NULL. In the next step, we will learn to eliminate NULLs using aggregations to get better output.

Manual pivoting using **GROUP BY**	Execute the following statement(s) to perform manual pivoting using **GROUP BY** and aggregations
	
	`-- Step 5: Execute the statement(s) to perform manual pivoting using group by and aggregations`
	`SELECT ID ,`
	`MAX(CASE WHEN [key] = 'key1' THEN Value END) AS [key1],`
	`MAX(CASE WHEN [key] = 'key2' THEN Value END) AS [Key2],`
	`MAX(CASE WHEN [key] = 'key3' THEN Value END) AS [Key3]`
	`FROM dbo.tpivot`
	`GROUP BY ID;`
	
	ID | key1 | Key2 | Key3
	1 | ABC | 2016-10-01 | 123
	2 | 123 | ABC | 2016-10-01
	
	Explanation: The above query eliminates the NULL values using **MAX** aggregate function and performs grouping by **ID** column. **MAX** aggregate function will give the maximum value present in the column. For example, **Key1** column for **ID 1** has **ABC** and two NULLs present in it (observe the output of **Step 4**). **MAX** aggregate function retrieved only **ABC** as other values are NULL.

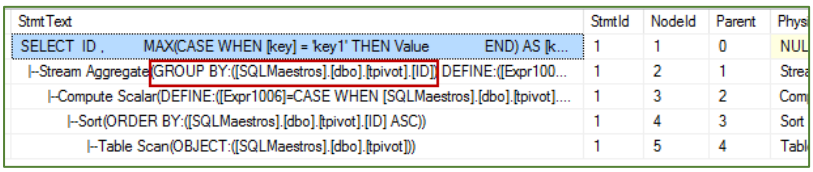
Pivoting using **PIVOT** operator	Execute the following statement(s) to perform pivoting on **tpivot** table using **PIVOT** operator
	
	`-- Step 6: Execute the following statement(s) to perform pivoting using pivot operator`
	`SELECT ID ,`
	`[key1],`
	`[key2],`
	`[key3]`
	`FROM dbo.tpivot PIVOT( MAX([Value]) FOR [key] IN ( [key1], [key2], [key3] ) ) AS pvt;`
	
	ID | key1 | Key2 | Key3
	1 | ABC | 2016-10-01 | 123
	2 | 123 | ABC | 2016-10-01
	
	Explanation: The above query performed the pivoting using **PIVOT** operator to retrieve the same output as in **Step 5**.

Add **IDENTITY** column	Execute the following statement(s) to **ALTER** the table to add **IDENTITY** column

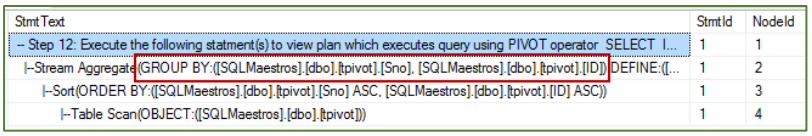
	-- Step 7: Execute the following statement(s) to add identity column to tpivot table ALTER TABLE dbo.tpivot ADD Sno INT IDENTITY;
Execute Manual pivoting query	Execute the following statement(s) to perform manual pivoting -- Step 8: Execute the following statement(s) to perform manual pivoting SELECT ID , MAX(CASE WHEN [key] = 'key1' THEN Value END) AS [key1], MAX(CASE WHEN [key] = 'key2' THEN Value END) AS [Key2], MAX(CASE WHEN [key] = 'key3' THEN Value END) AS [Key3] FROM dbo.tpivot GROUP BY ID; ID key1 Key2 Key3 1 ABC 2016-10-01 123 2 123 ABC 2016-10-01 **Observation**: Even after adding the new **IDENTITY** column, the manual pivot query gives the same result set as shown in **step 5**.
Execute the query using **PIVOT** operator	Execute the following statement(s) to perform pivoting using the **PIVOT** operator -- Step 9: Execute the following statement(s) to perform pivoting using PIVOT operator SELECT ID , [key1], [key2], [key3] FROM dbo.tpivot PIVOT(MAX([Value]) FOR [key] IN ([key1], [key2], [key3])) AS pvt; ID key1 key2 key3 1 ABC NULL NULL 1 NULL 2016-10-01 NULL 1 NULL NULL 123 2 123 NULL NULL 2 NULL ABC NULL 2 NULL NULL 2016-10-01 **Observation:** After adding the new column with **IDENTITY** property, the output of the **PIVOT** query has changed. This is because of internal grouping

	that **PIVOT** operator performs. In the next few steps, we will observe the grouping of the manual pivoting query and the query using **PIVOT** operator.

Turn ON **SHOWPLAN_ALL**	Execute the following statement(s) to turn ON **SHOWPLAN_ALL**
	
	`-- Step 10: Execute the following statement(s) to turn SHOWPLAN_ALL ON`
	`SET SHOWPLAN_ALL ON`
	
	Note: **SHOWPLAN_ALL** gives the details about, how the TSQL statements are executed and provides estimates of the resource requirements of the statements. For more information click on following MSDN link
	
	https://msdn.microsoft.com/en-in/library/ms187735.aspx

Execute manual pivoting query	Execute the following statement(s) to perform manual pivoting
	
	`-- Step 11: Execute the following statement(s) to view plan which executes manual pivoting query`
	`SELECT ID ,`
	`MAX(CASE WHEN [key] = 'key1' THEN Value END) AS [key1],`
	`MAX(CASE WHEN [key] = 'key2' THEN Value END) AS [Key2],`
	`MAX(CASE WHEN [key] = 'key3' THEN Value END) AS [Key3]`
	`FROM dbo.tpivot`
	`GROUP BY ID;`
	
	 
	
	Observation: Observe that, the manual pivoting query groups the columns only by **ID** column. This query is not using **Sno** column (IDENTITY column) in the grouping. Therefore, the output is same with and without the **IDENTITY** column as we mentioned the grouping column explicitly in the query.

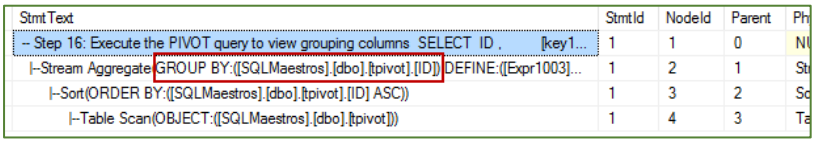
Execute PIVOT query	Execute the following statement(s) to perform pivoting using **PIVOT** query
	
	`-- Step 12: Execute the following statement(s) to view plan which executes query using PIVOT operator`
	`SELECT ID ,`
	`[key1],`
	`[key2],`
	`[key3]`
	`FROM dbo.tpivot PIVOT( MAX([Value]) FOR [key] IN ( [key1], [key2],`

	[key3])) AS pvt;  **Observation:** The above query is using **PIVOT** operator to perform pivoting. Observe that, the **PIVOT** operator is grouping the columns first by **Sno** column and then by **ID** column. The pivot operator performs an implicit group by on all the columns except the pivot column. As **Sno** is an **IDENTITY** column and we are grouping by **Sno** column first, NULL values are displayed in the output of **step 9**. A subquery is used to overcome this problem, which will be our next step.

Turn OFF SHOWPLAN_ALL	Execute the following statement(s) to turn OFF **SHOWPLAN_ALL**
	`-- Step 13: Turn OFF ShowPlan_ALL`
	`SET SHOWPLAN_ALL OFF`

Execute PIVOT query	Execute the following statement(s) to perform pivoting using PIVOT operator using subquery
	`-- Step 14: Execute the following statement(s) to fix the PIVOT query using subquery`
	`SELECT ID , [key1], [key2], [key3] FROM ( SELECT ID , [key], Value FROM dbo.tpivot ) AS sq --Subquery PIVOT( MAX([Value]) FOR [key] IN ( [key1], [key2], [key3] ) ) AS pvt;`
	 
	Explanation: The subquery is used in above query instead of directly using the table as shown in **Step 12**. In the subquery, we mentioned **ID**, **Key** and **Value** columns and not mentioned the **Sno** column. The query retrieved output as we expected because of the **PIVOT** operator groups the records on **ID** column only.

Turn ON **SHOWPLAN_ALL**	Execute the following statement(s) to turn ON **SHOWPLAN_ALL**
	
	`-- Step 15: Turn ON SHOWPLAN_ALL`
	`SET SHOWPLAN_ALL ON`

Observe the grouping columns	Execute the following statement(s) to observe the grouping columns
	
	`-- Step 16: Execute the PIVOT query to view grouping columns`
	`SELECT ID ,`
	`[key1],`
	`[key2],`
	`[key3]`
	`FROM ( SELECT ID,`
	`[key],`
	`Value`
	`FROM dbo.tpivot`
	`) AS sq PIVOT( MAX([Value]) FOR [key] IN ( [key1], [key2], [key3] ) ) AS pvt;`
	
	 
	
	Observation: As we used the subquery in the **PIVOT** query, the **PIVOT** operator grouping the data only based on **ID** column.

Turn OFF **SHOWPLAN_ALL**	Execute the following statement(s) to turn OFF **SHOWPLAN_ALL**
	
	`-- Step 17: Turn OFF SHOWPLAN_ALL`
	`SET SHOWPLAN_ALL OFF`

Close all the query windows	Close all the query windows ( ) and if **SSMS** asks to save changes, click **NO**

Exercise 3: Performance comparison between manual pivoting and PIVOT operator

Scenario

In this exercise, we will learn the performance difference between manual pivot and PIVOT operator.

Tasks	Detailed Steps
Open **3_Performance.sql**	1. Click **File** | **Open** | **File** or press (**Ctrl + O**) 2. In **Open File** dialogue box, navigate to **Pivoting in SQL Server/Scripts** folder

	3. Select **3_Performance.sql** and click **Open**

Setup	Execute the following setup section. This setup section will
	• **CREATE** kvpair table
	• **INSERT** 10000 records into kvpair table
	
	`--------------`
	`--Begin:Setup`
	`--------------`
	`USE SQLMaestros`
	`GO`
	`--Create kvpair table`
	`CREATE TABLE [dbo].kvpair(`
	`[Sno] [BIGINT] IDENTITY(1,1) NOT NULL,`
	`[ID] [INT] NULL,`
	`[key] [NVARCHAR](MAX) NULL,`
	`[Value] [NVARCHAR](MAX) NULL,`
	`)`
	`GO`
	`--Insert records into kvpair table using bulk insert`
	`BULK INSERT SQLMaestros.dbo.kvpair`
	`FROM 'C:\Pivoting in SQL Server\Data\kvpair.csv' WITH (`
	`FIELDTERMINATOR = ','`
	`,ROWTERMINATOR = '\n',`
	`FIRSTROW = 2`
	`)`
	`--------------`
	`--End:Setup`
	`--------------`

View the records	Execute the following statement(s) to view the data in **kvpair** table
	
	`-- Step 1: Execute the following statement(s) to view the data in table`
	`USE SQLMaestros;`
	`SELECT *`
	`FROM kvpair;`
	
	Sno | ID | key | Value
	1 | 1 | Marine | Emnipar
	2 | 2 | Chemicals | Frotinax Holdings
	3 | 1 | Marine | Upwerower International
	4 | 2 | Marine | Tupbanover Direct Inc
	5 | 1 | Chemicals | Cipjubantor Holdings Inc
	
	Note: The above image is a trimmed snapshot of the original result set. The original result set contains more number of records.

	Observation: The **kvpair** table contains four columns, **Sno** is an IDENTITY column, **ID** values contains the ID of each record. The **key** column contains the department details and **Value** column contains the value for the respective department.

Drop buffers	Execute the following statement(s) to drop buffers
	
	`--Step 2: Execute the following statement(s)`
	`DBCC DROPCLEANBUFFERS`
	`GO`
	
	Note: Do not execute the command in the production environment.

Execute the statement(s)	Execute the following statement(s) to set **STATISTICS TIME** and **STATISTICS IO ON**
	
	`--Step 3: Execute the following statement(s)`
	`SET STATISTICS TIME ON`
	`SET STATISTICS IO ON`

Execute manual pivoting query	Execute the following statement(s) to perform manual pivoting
	
	`--Step 4: Execute the following statement(s) to perform manual pivoting`
	`SELECT ID ,`
	`MAX(CASE WHEN [key] = 'Construction' THEN Value END) AS [Construction],`
	`MAX(CASE WHEN [key] = 'Chemicals' THEN Value END) AS [Chemicals],`
	`MAX(CASE WHEN [key] = 'Marine' THEN Value END) AS [Marine],`
	`MAX(CASE WHEN [key] = 'Railways' THEN Value END) AS [Railways],`
	`MAX(CASE WHEN [key] = 'Water' THEN Value END) AS [Water],`
	`MAX(CASE WHEN [key] = 'Power' THEN Value END) AS [Power]`
	`FROM dbo.kvpair`
	`GROUP BY ID;`
	`GO`
	
	ID | Construction | Chemicals | Marine | Railways | Water | Power
	1 | Zeezapegazz WorldWide Company | Zeezapefex | Zeezapower | Zeewerpex Holdings | Zeewerpex International Company | Zeevenentor Direct
	2 | Zeevenplax Direct | Zeewerupicator WorldWide | Zeezapazz Direct | Zeewerar | Zeezapor | Zeezapover International
	
	`Table 'Worktable'. Scan count 0, logical reads 0,`
	`Table 'Workfile'. Scan count 0, logical reads 0,`
	`Table 'kvpair'. Scan count 1, logical reads 101`
	
	`SQL Server Execution Times:`
	`CPU time = 1375 ms, elapsed time = 1494 ms.`

	Observation: To perform the pivoting operation on **kvpair** table, manual pivoting took **1.5** seconds and did **101** logical reads.

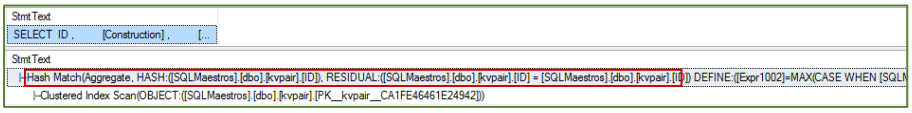
Drop buffers	Execute the following statement(s) to drop buffers
	
	`--Step 5: Execute the following statement(s)`
	`DBCC DROPCLEANBUFFERS`
	`GO`

Pivoting using PIVOT operator	Execute the following statement(s) to perform pivoting using **PIVOT** operator
	
	`--Step 6: Execute the following statement(s) to perform pivoting using PIVOT operator`
	`SELECT ID ,`
	`[Construction],`
	`[Chemicals],`
	`[Marine],`
	`[Railways],`
	`[Water],`
	`[Power]`
	`FROM ( SELECT ID,`
	`[key],`
	`Value`
	`FROM dbo.kvpair`
	`) AS sq PIVOT( MAX([Value]) FOR [key] IN ( [Construction], [Chemicals],`
	`[Marine], [Railways],`
	`[Water], [Power] ) ) AS pvt`
	
	ID | Construction | Chemicals | Marine | Railways | Water | Power
	1 | Zeezapegazz WorldWide Company | Zeezapefex | Zeezapower | Zeewerpex Holdings | Zeewerpex International Company | Zeevenentor Direct
	2 | Zeevenplax Direct | Zeewerupicator WorldWide | Zeezapazz Direct | Zeewerar | Zeezapor | Zeezapover International
	
	`Table 'Worktable'. Scan count 0, logical reads 0,`
	`Table 'Workfile'. Scan count 0, logical reads 0,`
	`Table 'kvpair'. Scan count 1, logical reads 101,`
	
	`SQL Server Execution Times:`
	`CPU time = 1390 ms, elapsed time = 1470 ms.`
	
	Observation: To perform the pivoting operation on **kvpair** table using **PIVOT** operator, the query took **1.4** seconds' time and did **101** logical reads. Therefore, there is no performance difference between manual pivoting and pivoting using **PIVOT** operator because **PIVOT** operator internally uses the manual pivoting (CASE) query to implement pivoting. This will be shown in the later step(s).

Execute the statement(s)	Execute the following statement(s) to turn OFF **STATISTICS TIME** and **STATISTICS IO**
	
	`-- Step 7: Execute the following statement(s)`
	`SET STATISTICS TIME OFF`
	`SET STATISTICS IO OFF`

Clean the buffers	Execute the following statement(s) to clean the buffers
	
	`--Step 8: Execute the following statement(s)`
	`DBCC DROPCLEANBUFFERS`
	`GO`

Turn **ON SHOWPLAN_TEXT**	Execute the following statement(s) to turn **ON SHOWPLAN_TEXT**
	
	`-- Step 9: Execute the following statement(s) to turn on SHOWPLAN_TEXT`
	`SET SHOWPLAN_TEXT ON`
	
	Note: **SHOWPLAN_TEXT** when turned **ON** gives us detailed information about how statements are executed.

Execute the **PIVOT** query	Execute the following statement(s) to view the text plan
	
	`-- Step 10: Execute the following statement(s) to perform pivoting using PIVOT operator`
	`SELECT ID,`
	`[Construction],`
	`[Chemicals],`
	`[Marine],`
	`[Railways],`
	`[Water],`
	`[Power]`
	`FROM ( SELECT ID ,`
	`[key],`
	`Value`
	`FROM dbo.kvpair`
	`) AS sq PIVOT( MAX([Value]) FOR [key] IN ( [Construction], [Chemicals],`
	`[Marine], [Railways],`
	`[Water], [Power])) AS pvt`
	
	 

	Note: Copy the highlighted portion of the code to notepad for the better view. The code will look like `|--Hash Match(Aggregate, HASH:([SQLMaestros].[dbo].[kvpair].[ID]), RESIDUAL:([SQLMaestros].[dbo].[kvpair].[ID] = [SQLMaestros].[dbo].[kvpair].[ID]) DEFINE:([Expr1002]=MAX(CASE WHEN [SQLMaestros].[dbo].[kvpair].[key]=N'Construction' THEN [SQLMaestros].[dbo].[kvpair].[Value] ELSE NULL END), [Expr1003]=MAX(CASE WHEN [SQLMaestros].[dbo].[kvpair].[key]=N'Chemicals' THEN [SQLMaestros].[dbo].[kvpair].[Value] ELSE NULL END), [Expr1004]=MAX(CASE WHEN [SQLMaestros].[dbo].[kvpair].[key]=N'Marine' THEN [SQLMaestros].[dbo].[kvpair].[Value] ELSE NULL END), [Expr1005]=MAX(CASE WHEN [SQLMaestros].[dbo].[kvpair].[key]=N'Railways' THEN [SQLMaestros].[dbo].[kvpair].[Value] ELSE NULL END), [Expr1006]=MAX(CASE WHEN [SQLMaestros].[dbo].[kvpair].[key]=N'Water' THEN [SQLMaestros].[dbo].[kvpair].[Value] ELSE NULL END), [Expr1007]=MAX(CASE WHEN [SQLMaestros].[dbo].[kvpair].[key]=N'Power' THEN [SQLMaestros].[dbo].[kvpair].[Value] ELSE NULL END)))` **Observation**: Internally **PIVOT** operator writes the same code what we have written using manual pivoting.

Turn off **SHOWPLAN_TEXT**	Execute the following statement(s) to turn **OFF SHOWPLAN_TEXT**
	
	`-- Step 11: Execute the following`
	`SET SHOWPLAN_TEXT OFF`

Cleanup	Execute following cleanup section
	
	`--cleanup`
	`DROP TABLE dbo.kvpair`
	`GO`

Close all the query windows	Close all the query windows ( ) and if **SSMS** asks to save changes, click **NO**

Exercise 4: Unpivoting the data

Scenario

In this exercise, we will learn to unpivot the pivoted data using manual method and using **UNPIVOT** operator

Tasks	Detailed Steps
Open **4_Unpivot.sql**	1. Click **File** | **Open** | **File** or press (**Ctrl + O**) 2. In **Open File** dialogue box, navigate to **Pivoting in SQL Server/Scripts** folder 3. Select **4_Unpivot.sql** and click **Open**

Setup	Execute the following setup section. This setup section will perform • Creates **Orders** table • Creates **INDEX** on **Orders** table • Inserts **11** records into the **Orders** table • Performs pivoting on **Orders** table and move the pivoted data to **PvtCustOrders** table

```
----------------
-- Begin: Setup
----------------
-- Execute the following statement(s) to create Orders table
USE SQLMaestros;
GO
SET NOCOUNT ON
IF OBJECT_ID('dbo.Orders') IS NOT NULL
    DROP TABLE dbo.Orders;
GO
CREATE TABLE dbo.Orders
    (
      orderid INT NOT NULL
                  PRIMARY KEY NONCLUSTERED,
      orderdate DATETIME NOT NULL,
      empid INT NOT NULL,
      custid VARCHAR(5) NOT NULL,
      qty INT NOT NULL
    );

-- Execute the following statement(s) to create index on Orders
table
CREATE UNIQUE CLUSTERED INDEX idx_orderdate_orderid
ON dbo.Orders(orderdate, orderid);

-- Execute the following statement(s) to insert records into
Orders table
INSERT  INTO dbo.Orders ( orderid ,orderdate ,empid ,custid
,qty    )
VALUES  ( 30001 ,'20020802' ,3 ,'A' ,10);
INSERT  INTO dbo.Orders( orderid , orderdate ,empid ,custid
,qty )
VALUES  ( 10001 ,'20021224' ,1 ,'A' ,12);
INSERT  INTO dbo.Orders( orderid ,orderdate ,empid ,custid
,qty)
VALUES  ( 10005 , '20021224' ,1 ,'B' ,20);
INSERT  INTO dbo.Orders( orderid , orderdate ,empid ,custid
,qty )
VALUES  ( 40001 ,'20030109' , 4 ,'A' ,40);
INSERT  INTO dbo.Orders  ( orderid ,orderdate ,empid , custid
, qty)
VALUES  ( 10006 ,'20030118' , 1 ,'C' ,14);
INSERT  INTO dbo.Orders( orderid ,orderdate , empid ,custid
,qty)
VALUES  ( 20001 , '20030212' ,2 , 'B' ,  12);
INSERT  INTO dbo.Orders( orderid ,orderdate ,empid ,custid ,
qty)
VALUES  ( 40005 ,'20040212' ,4 ,'A' ,10);
```

```
INSERT  INTO dbo.Orders( orderid ,orderdate , empid , custid
,qty)
VALUES  ( 20002 ,'20040216' ,2 ,'C' ,20);
INSERT   INTO dbo.Orders( orderid ,orderdate ,empid ,custid
,qty)
VALUES  ( 30003 ,'20040418' ,3 ,'B' ,15);
INSERT   INTO dbo.Orders( orderid ,orderdate ,empid ,custid
,qty)
VALUES  ( 30004 ,'20020418' ,3 ,'C' ,22);
INSERT  INTO dbo.Orders( orderid , orderdate , empid ,custid
,qty)
VALUES  ( 30007 ,'20020907' ,3 ,'D' ,30);

-- Execute the following statement(s) to pivot the data in
Orders table and move the records to PvtCustOrders table
IF OBJECT_ID('dbo.PvtCustOrders') IS NOT NULL
    DROP TABLE dbo.PvtCustOrders;
SELECT  custid ,
        ISNULL([2002], 0) AS [2002] ,
        ISNULL([2003], 0) AS [2003] ,
        ISNULL([2004], 0) AS [2004]
INTO    dbo.PvtCustOrders
FROM    ( SELECT    custid ,
                    YEAR(orderdate) AS orderyear,
                    qty
          FROM      dbo.Orders
        ) AS D PIVOT( SUM(qty) FOR orderyear IN ( [2002],
[2003], [2004] ) ) AS P;

----------------
-- End: Setup
----------------
```

View the records

Execute the following statement(s) to view the records

```
-- Step 1: Execute the following statement(s) to view the
records from PvtCustOrders table
SELECT  *
FROM    dbo.PvtCustOrders;
```

custid	2002	2003	2004
A	22	40	10
B	20	12	15
C	22	14	20
D	30	0	0

Observation: PvtCustOrders table contains the pivoted data.

Manual Unpivot

UNPIVOT operation represents rotating columns to rows.

Execute the following statement(s) to perform manual unpivot

```
--Step 2: Execute the following statement(s) to perform manual
unpivot
SELECT  custid ,
        orderyear ,
        qty
FROM    ( SELECT    custid ,
                    orderyear ,
                    CASE orderyear
                      WHEN 2002 THEN [2002]
                      WHEN 2003 THEN [2003]
                      WHEN 2004 THEN [2004]
                    END AS qty
          FROM      dbo.PvtCustOrders ,
                    ( SELECT    2002 AS orderyear
                      UNION ALL
                      SELECT    2003
                      UNION ALL
                      SELECT    2004
                    ) AS OrderYears
        ) AS D
WHERE   qty IS NOT NULL;
```

custid	orderyear	qty
A	2002	22
A	2003	40
A	2004	10
B	2002	20
B	2003	12
B	2004	15
C	2002	22
C	2003	14
C	2004	20
D	2002	30
D	2003	0
D	2004	0

Explanation: The above query performs unpivoting operation using **CASE** statement(s).

Unpivoting using **UNPIVOT** operator

UNPIVOT operation rotates columns to rows.

Execute the following statement(s) to perform unpivoting using **UNPIVOT** operator

```
-- Step 3: Execute the following statement(s) to perform
Unpivot using UNPIVOT operator
SELECT  custid ,
        orderyear,
        qty
FROM      dbo.PvtCustOrders UNPIVOT(  qty  FOR orderyear  IN  (
[2002], [2003],

[2004]) ) AS U;
GO
```

custid	orderyear	qty
A	2002	22
A	2003	40
A	2004	10
B	2002	20
B	2003	12
B	2004	15
C	2002	22
C	2003	14
C	2004	20
D	2002	30
D	2003	0
D	2004	0

Explanation: The above query performs unpivoting using **UNPIVOT** operator.

Compare the outputs

Execute the following statement(s) to compare the outputs

```
-- Step 4: Execute the following statement(s) to compare the
output before pivoting and after unpivoting the pivoted data
SELECT  custid ,
        YEAR(orderdate) AS orderyear ,
        qty
FROM    dbo.Orders
ORDER BY custid;
--Before pivoting
GO
SELECT  custid ,
        orderyear ,
        qty
FROM      dbo.PvtCustOrders UNPIVOT(  qty  FOR  orderyear  IN  (
[2002], [2003],

[2004] ) ) AS U;
GO  -- After unpivoting the pivoted data
```

Before Pivoting

custid	orderyear	qty
A	2002	10
A	2002	12
A	2003	40
A	2004	10
B	2004	15
B	2003	12
B	2002	20
C	2002	22
C	2003	14
C	2004	20
D	2002	30

After unpivoting the pivoted data

custid	orderyear	qty
A	2002	22
A	2003	40
A	2004	10
B	2002	20
B	2003	12
B	2004	15
C	2002	22
C	2003	14
C	2004	20
D	2002	30
D	2003	0
D	2004	0

	Observation: Unpivoting may not give the exact data that is there before pivoting the data. Which means **UNPIVOT** is not exact reverse operation of **PIVOT**. **PIVOT** operation performs aggregation on given column and will merge all the rows into one single row. As the values are merged into one row, UNPIVOT cannot reproduce exact table valued function which there earlier before pivoting. Observe the data mismatches between the two images above. For example, **Custid D** repeats two more times after unpivoting.

cleanup	Execute the following cleanup section
	
	`--Cleanup`
	`DROP TABLE dbo.Orders`
	`GO`
	`DROP TABLE dbo.PvtCustOrders`
	`GO`

Close all the query windows	Close all the query windows ( ) and if **SSMS** asks to save changes, click **NO**

Exercise 5: T-SQL Challenge

Here is a TSQL challenge for you related to **Dynamic pivoting**

Problem Statement: "Write a query to get an output looks like following using dynamic pivoting"

Table(s) used: **tpivot** table in **SQLMaestros** database

Final output:

ID	key1	key2	key3
1	ABC	2016-10-01	123
2	123	ABC	2016-10-01

Solution: Please try to solve the above problem by yourself. If unable to solve then refer to the **5_T-SQL Challenge.sql** script which is kept in Pivoting in **SQL Server**Scripts folder.

Summary

In this article, we have learned:

- Pivoting the data without using PIVOT operator
- Pivoting the data using PIVOT operator
- Dynamic pivoting
- Manual pivoting
- Pivoting using PIVOT operator

- Grouping of PIVOT operator in the presence of IDENTITY column
- The performance difference between manual pivoting and pivoting using PIVOT operator
- How to unpivot the pivoted data using manual method and using UNPIVOT operator
- How to solve the grouping problem using subquery

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

SQL SERVER TIPS AND TRICKS

TIPS 01

Want to open MS Docs from SSMS? Either use keyboard shortcut Ctrl + Alt + R or Menu > View > Other Windows > Web Browser

TIPS 02

Use SET NOCOUNT ON in Stored Procedures to reduce network traffic and to boost performance.

TIPS 03

Using INDEXED VIEW in a subquery? Use along with NOEXPAND hint to avoid expanding the Indexed View. Increases performance.

TIPS 04

Traversing through the XML execution plan requires XQuery skills. Sys.dm_exec_text_query_plan DMF gives the text version of the XML execution plan. A simple LIKE search can help find what is needed

TIPS 05

Wish to see the progress of Backup/Restore? Use WITH STATS N along with BACKUP/RESTORE command. N is the percentage interval.

TIPS 06

Want to add line numbers in SSMS? Menu > Tools > Options > Text Editor > All Languages > General. Check the 'Line numbers' box.

LEARN MORE

DPS 2022
PRE - CON

8-hours Virtual Classroom Training. Instructor-Led. LIVE Attendance + Class Recordings.

- Analytics At Scale With Power BI And Azure Synapse
- Advanced Analytics With Transact-SQl
- Data Factory - ETL In The Cloud
- Masters Passport To Extended Events
- Data Modelling In Power BI – From Zero To Hero
- Query Store Deep Dive
- Cracking Power BI Performance Tuning In A Day!
- Spark For Data Engineers For DPVS 2022
- End-To-End Analytics With Power BI!
- The Performance Tuning Magic Show
- SQL Server Performance Tuning And Troubleshooting
- PowerShell DBA Dream DBATools Workshop
- Synapse Pipelines Best Practices

LEARN MORE

SQLMaestros

FREE VIDEO COURSES

Query Tuning Soup to Nuts
(2 Hours)

T-SQL Querying Crash Course
(3 hours)

SUBSCRIBE

@SQLMaestros

www.SQLMaestros.com