**Aggregate Functions to Validate Data for SQL Server Replication and ETL Processes**

By: [Tim Smith](https://www.mssqltips.com/sqlserverauthor/92/tim-smith/)

FROM: <https://www.mssqltips.com/sqlservertip/4204/aggregate-functions-to-validate-data-for-sql-server-replication-and-etl-processes/>

**Problem**

We've experienced SQL Server replication problems in the past when we thought it was running normally because our reports indicated that source and destination row counts were similar or identical, but when we looked at the data, we discovered that they were different. Can we create a another check where we perform an aggregate function on a numeric column when we suspect that the row counts might not be accurate?

**Solution**

We can use an aggregate function on a numeric column to double verify, but let's first look at our row count report because, depending on how we're running it, we may be guaranteeing some problems.

In the tip [Using PowerShell for SQL Server Replication Tracking](https://www.mssqltips.com/sqlservertip/4145/using-powershell-for-sql-server-replication-tracking/), I use .NET's RowCount property for a reason: it's fast and the measurement of the source and destination will be similar in time. A few thousand tables can be checked within two minutes, unless we see network issues where it struggles to connect to either the source or destination server (and this would be a red flag for replication too - if that script struggles to connect to a server, replication will as well).

Imagine I used a hard count, where I actually performed a **COUNT(\*)**, locking the table - not only could I possibly delay replication in my check, the timing of one may finish before the other - creating the possibility of false alerts. No perfect way exists for monitoring replication flow, but one must be aware of the advantages and disadvantages of every approach, as their approach may create a problem of latency (I've seen many replication reports from developers that do this and when they complain about latency, I point out that bad code guarantees latency).

I highly advise against using aggregates for regular analysis of data involved in replication, unless you know specific down times where these will not interrupt any replication or reporting process. Even then, you should be very careful about how you will approach it and be willing to accept the reality that you may see some delays in your data.  For instance, suppose I take the sum of two columns and the destination is different by 100, and we know replication is still running. Is that a cause for concern? This entire approach could open a statistical nightmare because in any given moment with replication, it could be off by 100.

You will have to not only know the average and deviation, but you'd need to know these values for different points of a load. In an example where I measure a sum after the Hang Seng closes - so in theory, replication should have finished, I could just as easily measure the RowCount property instead of performing an expensive sum. The bottom line: this approach will - from a statistical standpoint - possibly create false alerts, and you'll unintentionally be training people to ignore them, which is far worse than relying on mismatches or the command and transaction backlog (both of which together are precise and can be used with averages and deviations overall).

For ETL flows, this script will help since most ETL flows occur at specific times and should finish at specific times; in these cases, the below script makes sense to use. Those cautions being written, we can use an aggregate function - like **SUM** - to perform a check if we suspect there may be an issue with a particular table.

Function Get-CheckSumColumn {

Param(

[ValidateLength(7,15)][string]$readserver

, [ValidateLength(50,4000)][string]$readquery

, [ValidateRange(0,1)][int]$options

)

Process

{

$scon = New-Object System.Data.SqlClient.SqlConnection

$scon.ConnectionString = "Data Source=$readserver;Initial Catalog=master;Integrated Security=true;"

$cmd = New-Object System.Data.SqlClient.SqlCommand

$cmd.Connection = $scon

$cmd.CommandText = $readquery

$cmd.CommandTimeout = 10

try

{

$scon.Open()

$sqlread = $cmd.ExecuteReader()

while ($sqlread.Read())

{

if ($options -eq 0)

{

$checksum\_column = $sqlread["COLUMN\_NAME"]

}

elseif ($options -eq 1)

{

$checksum\_column = $sqlread["CheckSumValue"]

}

}

}

catch [Exception]

{

Write-Warning "Get-CheckSumColumn ($options)"

Write-Warning $\_.Exception.Message

}

finally

{

$cmd.Dispose()

$scon.Dispose()

}

return $checksum\_column

}

}

Function Get-CheckSumQueries {

Param(

[ValidateLength(3,75)][string]$readserver

, [ValidateLength(3,75)][string]$distributiondb

, [ValidateLength(3,100)][string]$article

, [ValidateLength(3,100)][string]$source\_database

, [ValidateLength(3,100)][string]$destination\_server

)

Process

{

$checksumquery = "

SELECT DISTINCT

DENSE\_RANK() OVER (ORDER BY (dist.publisher + ' ' + s.publisher\_db + ' ' + a.article + ' ' + ss.srvname + ' ' + s.subscriber\_db + ' ' + a.destination\_object)) ID

, dist.publisher SourceServer

, s.publisher\_db SourceDatabase

, ss.srvname DestinationServer

, s.subscriber\_db DestinationDatabase

, 'SELECT TOP 1 COLUMN\_NAME FROM ' + s.publisher\_db + '.INFORMATION\_SCHEMA.COLUMNS WHERE TABLE\_NAME = ''' + a.article + ''' AND DATA\_TYPE IN (''numeric'',''int'',''decimal'',''float'')' AS CheckSumQuerySource

, 'SELECT TOP 1 COLUMN\_NAME FROM ' + s.subscriber\_db + '.INFORMATION\_SCHEMA.COLUMNS WHERE TABLE\_NAME = ''' + a.article + ''' AND DATA\_TYPE IN (''numeric'',''int'',''decimal'',''float'')' AS CheckSumQueryDestination

FROM MSArticles a

INNER JOIN (

SELECT DISTINCT job\_id

, publisher

, publication\_id

, publisher\_db

FROM MSreplication\_monitordata

) AS dist ON dist.publication\_id = a.publication\_id

AND dist.publisher\_db = a.publisher\_db

INNER JOIN MSpublications p ON a.publication\_id = p.publication\_id

AND p.publisher\_db = a.publisher\_db

AND a.publisher\_id = p.publisher\_id

INNER JOIN MSsubscriptions s ON s.publisher\_db = dist.publisher\_db

INNER JOIN master..sysservers ss ON s.subscriber\_id = ss.srvid

INNER JOIN master..sysservers srv ON srv.srvid = s.publisher\_id

INNER JOIN MSdistribution\_agents da ON da.publisher\_id = s.publisher\_id

AND da.subscriber\_id = s.subscriber\_id

AND da.publisher\_db = dist.publisher\_db

AND da.job\_id = dist.job\_id

WHERE p.publication\_type = 0

AND a.article = '$article'

AND s.publisher\_db = '$source\_database'

AND ss.srvname = '$destination\_server'

"

$scon = New-Object System.Data.SqlClient.SqlConnection

$scon.ConnectionString = "Data Source=$readserver;Initial Catalog=$distributiondb;Integrated Security=true;Connection Timeout=0;"

$cmd = New-Object System.Data.SqlClient.SqlCommand

$cmd.Connection = $scon

$cmd.CommandText = $checksumquery

$cmd.CommandTimeout = 0

try

{

$scon.Open()

$sqlread = $cmd.ExecuteReader()

while ($sqlread.Read())

{

$checksum\_source = $sqlread["CheckSumQuerySource"]

$checksum\_destination = $sqlread["CheckSumQueryDestination"]

$checksum\_sourceserver = $sqlread["SourceServer"]

$checksum\_destinationserver = $sqlread["DestinationServer"]

$checksum\_sourcedatabase = $sqlread["SourceDatabase"]

$checksum\_destinationdatabase = $sqlread["DestinationDatabase"]

$checksum\_sret = Get-CheckSumColumn -readserver $checksum\_sourceserver -readquery $checksum\_source -options 0

$checksum\_dret = Get-CheckSumColumn -readserver $checksum\_destinationserver -readquery $checksum\_destination -options 0

$sourcecheck = "SELECT SUM($checksum\_sret) AS CheckSumValue FROM $checksum\_sourcedatabase.dbo.$article WITH(NOLOCK)"

$destinationcheck = "SELECT SUM($checksum\_dret) AS CheckSumValue FROM $checksum\_destinationdatabase.dbo.$article WITH(NOLOCK)"

$checksum\_svals = Get-CheckSumColumn -readserver $checksum\_sourceserver -readquery $sourcecheck -options 1

$checksum\_dvals = Get-CheckSumColumn -readserver $checksum\_destinationserver -readquery $destinationcheck -options 1

Write-Host $checksum\_svals

Write-Host $checksum\_dvals

}

}

catch [Exception]

{

Write-Warning "Get-CheckSumQueries"

Write-Warning $\_.Exception.Message

}

finally

{

$cmd.Dispose()

$scon.Dispose()

}

}

}

Get-CheckSumQueries -readserver "" -distributiondb "" -article "" -source\_database "" -destination\_server ""

A couple of notes on the above script:

1. This script assumes that the source and destination tables are identical; I also use a T-SQL query to get the source and destination information by using the table name, the destination server and the source database.
2. I limit the timeout to get both the **COLUMN\_NAME** and the **CheckSumValue** by ten seconds. If it can't return it that quickly, I don't want it to continue. The key here is that this isn't important, this is just a check.
3. I scale distribution databases by publisher, so I try to avoid using the *distribution* database for anything. Every distribution database - in my opinion - needs to be named *distribution\_publisher*, such as *distribution\_hangseng*, *distribution\_bitcoin*, *distribution\_shanghai*, etc. For this reason, this script requests which distribution database a developer is using.
4. The **Get-CheckSumColumn** returns both the column name and the value, using an if statement on **$options**. Keep this in mind if the script hits the catch block and errors out - the error outputs the option.
5. The script returns one value from the **INFORMATION\_SCHEMA**, looking at the numeric columns of either int, numeric, decimal, or float. If you use other data types, such as smallint, or tinyint, you may also want to include them.

Finally, anyone familiar with the law of large numbers knows that at any given moment, this could provide false information for replication too because it's statistically possible that both columns could equal each other, yet have different data relative to the data set. This is another reason to use this only in those *certain* cases. For ETL applications and time loads, this will help provided that you use columns (you may want to specify which ones instead of pulling anyone in particular) where you know the data will be different after a new load.

**Next Steps**

* Define a scenario before you would use this approach to validate data flow, noting that the above script can be used in ETL processes as well.
* When using this, cross check your results with what you see for row miscounts and transaction and command backlogs.
* Check out all of the [SQL Server Replication tips](https://www.mssqltips.com/sql-server-tip-category/7/replication/).

Last Update: 4/13/2016