# SQL Index Manager – Free GUI Tool for Index Maintenance on SQL Server and Azure

**Sergii Syrovatchenko**
8 Nov 2021 GPL3

Open source index maintenance tool for SQL Server and Azure

This article assesses the design approach of my free index maintenance tool for SQL Server and Azure: SQL Index Manager. The article analyzes the approach of the RedGate SQL Index Manager (v1.1.9.1378) and DevArt - dbForge Index Manager for SQL Server (v1.10.38), and explores why my tool divides the scan into two parts. Initially, one large request determines the size of the partitions in advance by filtering those that are not included in the filtering range, and then, we get only those partitions that contain data to avoid unnecessary reads from empty indexes.

**Download SQL Index Manager v1.0.0.69 (latest)**

**Download source code - 16.3 MB**

**Download SQL Index Manager v1.0.0.68 - 16.2 MB**

**Download SQL Index Manager v1.0.0.67 - 16.2 MB**

## Introduction

I have been working as a SQL Server DBA for over 8 years, administering and optimizing servers' performance. In my free time, I wanted to do something useful for the Universe and for my colleagues. This is how we eventually got a **free index maintenance tool** for SQL Server and Azure.

## Idea

Every once in a while, people, while working on their priorities, can resemble a finger-type battery - a motivational charge only lasts for one flash and then everything fades away. Until recently, I was no exception in this life observation. I was frequently haunted by ideas to create something of my own, but priorities changed from one to another and nothing was completed.

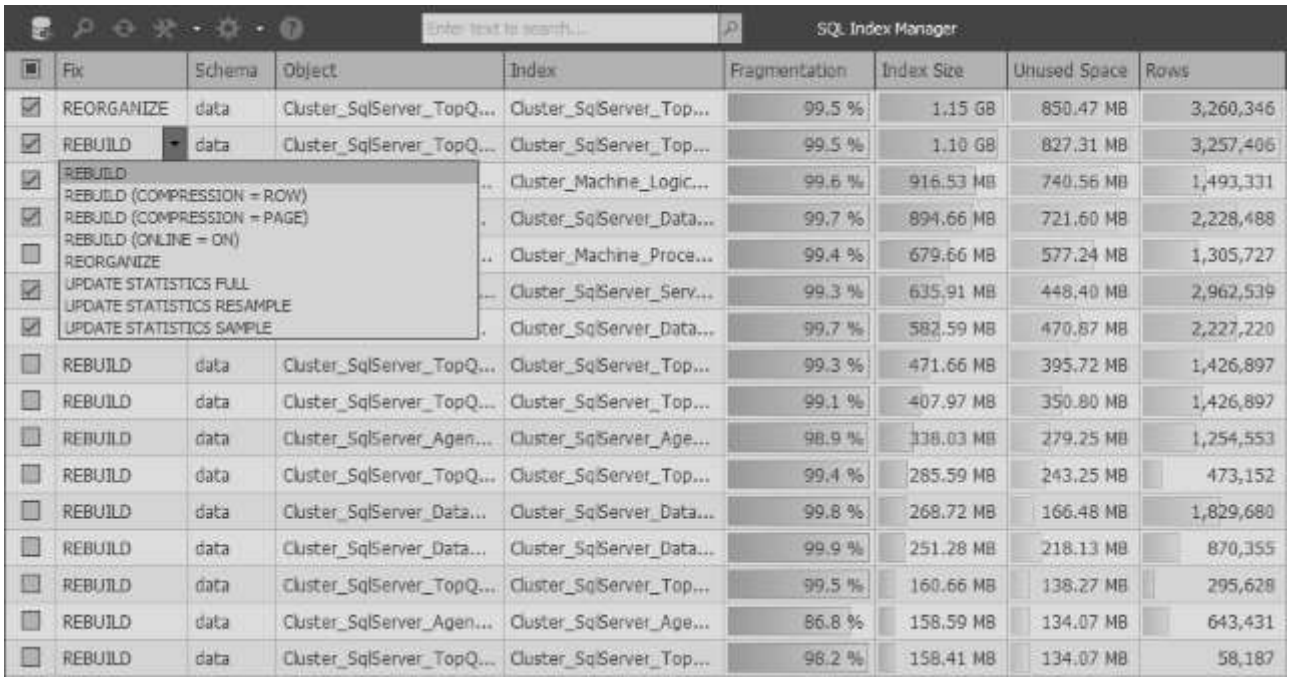
DevArt, which developed software for the development and administration of SQL Server, MySQL and Oracle databases, had a strong influence on my motivation and professional growth.

Before coming to them, little did I know about the specifics of creating my own product, but in the process, I gained a lot of knowledge about the internal structure of SQL Server. For more than a year, having been engaged in optimizing queries in their product lines, I gradually began to understand which functionality is more in demand on the market than the other one.

At a certain stage, the idea of making a new niche product arose in front of me, but due to certain circumstances, this idea did not succeed. At that time, basically I failed to find enough resources for a new project inside the company without affecting the core business.

Working at a brand-new place and trying to create a project on my own made me constantly compromise. The original idea to make a big product having all bells and whistles quickly grinded to a halt and gradually transformed into a different direction - to break the planned functionality into separate mini-tools and implement them independently from each other.

As a result, **SQL Index Manager** was born, a free index maintenance tool for SQL Server and Azure. The main idea was to take commercial alternatives from RedGate and Devart companies as a basis and try to improve its functionality in my own project.




# Implementation

Verbally, everything always sounds simple... just watch a couple of motivating videos, switch on "Rocky Balboa" mode and start making a cool product. But let's face the music, everything is not so rosy, because there are many pitfalls when working with the system table function `sys.dm_db_index_physical_stats` and at the same time, it is the only place from where you can get some up-to-date information about indexes fragmentation.

From the very first days of development, there was a great opportunity to make a dreary way among the standard schemes and copy the already debugged logic of the competing applications, while adding a little bit of ad-libbing. But after analyzing the queries for metadata, I wanted to do something more optimized, which, due to the bureaucracy of large companies, would never have appeared in their products.

When analyzing the RedGate SQL Index Manager (v1.1.9.1378 - $155 per user), you can see that the application uses a very simple approach: with the first query, we get a list of user tables and views, and after the second one, we return a list of all indexes within the selected database.

SQL

```
SELECT objects.name AS tableOrViewName
     , objects.object_id AS tableOrViewId
     , schemas.name AS schemaName
     , CAST(ISNULL(lobs.NumLobs, 0) AS BIT) AS ContainsLobs
     , o.is_memory_optimized
FROM sys.objects AS objects
JOIN sys.schemas AS schemas ON schemas.schema_id = objects.schema_id
LEFT JOIN (
    SELECT object_id
         , COUNT(*) AS NumLobs
    FROM sys.columns WITH (NOLOCK)
    WHERE system_type_id IN (34, 35, 99)
        OR max_length = -1
    GROUP BY object_id
) AS lobs ON objects.object_id = lobs.object_id
LEFT JOIN sys.tables AS o ON o.object_id = objects.object_id
WHERE objects.type = 'U'
    OR objects.type = 'V'

SELECT i.object_id AS tableOrViewId
     , i.name AS indexName
     , i.index_id AS indexId
     , i.allow_page_locks AS allowPageLocks
     , p.partition_number AS partitionNumber
     , CAST((c.numPartitions - 1) AS BIT) AS belongsToPartitionedIndex
FROM sys.indexes AS i
JOIN sys.partitions AS p ON p.index_id = i.index_id
                        AND p.object_id = i.object_id
JOIN (
    SELECT COUNT(*) AS numPartitions
         , object_id
         , index_id
    FROM sys.partitions
    GROUP BY object_id
           , index_id
) AS c ON c.index_id = i.index_id
      AND c.object_id = i.object_id
WHERE i.index_id > 0 -- ignore heaps
    AND i.is_disabled = 0
    AND i.is_hypothetical = 0
```

Next, in the `while` cycle for each index partition, a request is sent to determine its size and level of fragmentation. At the end of the scan, indexes that weigh less than the entry threshold are displayed on the client.

SQL

```
EXEC sp_executesql N'
SELECT index_id, avg_fragmentation_in_percent, page_count
FROM sys.dm_db_index_physical_stats(@databaseId, @objectId, @indexId, @partitionNr, NULL)'
    , N'@databaseId int,@objectId int,@indexId int,@partitionNr int'
    , @databaseId = 7, @objectId = 2133582639, @indexId = 1, @partitionNr = 1

EXEC sp_executesql N'
```

```
SELECT index_id, avg_fragmentation_in_percent, page_count
FROM sys.dm_db_index_physical_stats(@databaseId, @objectId, @indexId, @partitionNr, NULL)'
    , N'@databaseId int,@objectId int,@indexId int,@partitionNr int'
    , @databaseId = 7, @objectId = 2133582639, @indexId = 2, @partitionNr = 1

EXEC sp_executesql N'
SELECT index_id, avg_fragmentation_in_percent, page_count
FROM sys.dm_db_index_physical_stats(@databaseId, @objectId, @indexId, @partitionNr, NULL)'
    , N'@databaseId int,@objectId int,@indexId int,@partitionNr int'
    , @databaseId = 7, @objectId = 2133582639, @indexId = 3, @partitionNr = 1
```

When analyzing the logic of this application, you may find various drawbacks. For example, before sending a request, no checks are made on whether the current partition contains any rows to exclude empty partitions from scanning.

But the problem is manifested even more sharply in another aspect - the number of requests to the server will be approximately equal to the total number of rows from `sys.partitions`. Given the fact that real databases can contain tens of thousands of partitions, this nuance can lead to a huge number of similar requests to the server. In a situation when the database located on remote server, the scanning time will be even longer due to the increased network delays in the execution of each request, even the simplest one.

Unlike RedGate, a similar product dbForge Index Manager for SQL Server developed by DevArt (v1.10.38 - $99 per user) receives information in one large query and then displays everything on the client:

SQL

```
SELECT SCHEMA_NAME(o.[schema_id]) AS [schema_name]
     , o.name AS parent_name
     , o.[type] AS parent_type
     , i.name
     , i.type_desc
     , s.avg_fragmentation_in_percent
     , s.page_count
     , p.partition_number
     , p.[rows]
     , ISNULL(lob.is_lob_legacy, 0) AS is_lob_legacy
     , ISNULL(lob.is_lob, 0) AS is_lob
     , CASE WHEN ds.[type] = 'PS' THEN 1 ELSE 0 END AS is_partitioned
FROM sys.dm_db_index_physical_stats(DB_ID(), NULL, NULL, NULL, NULL) s
JOIN sys.partitions p ON s.[object_id] = p.[object_id]
                     AND s.index_id = p.index_id
                     AND s.partition_number = p.partition_number
JOIN sys.indexes i ON i.[object_id] = s.[object_id]
                  AND i.index_id = s.index_id
LEFT JOIN (
    SELECT c.[object_id]
         , index_id = ISNULL(i.index_id, 1)
         , is_lob_legacy = MAX(CASE WHEN c.system_type_id IN (34, 35, 99) THEN 1 END)
         , is_lob = MAX(CASE WHEN c.max_length = -1 THEN 1 END)
    FROM sys.columns c
    LEFT JOIN sys.index_columns i ON c.[object_id] = i.[object_id]
                                 AND c.column_id = i.column_id
                                 AND i.index_id > 0
    WHERE c.system_type_id IN (34, 35, 99)
        OR c.max_length = -1
    GROUP BY c.[object_id], i.index_id
) lob ON lob.[object_id] = i.[object_id]
     AND lob.index_id = i.index_id
JOIN sys.objects o ON o.[object_id] = i.[object_id]
JOIN sys.data_spaces ds ON i.data_space_id = ds.data_space_id
WHERE i.[type] IN (1, 2)
    AND i.is_disabled = 0
    AND i.is_hypothetical = 0
    AND s.index_level = 0
    AND s.alloc_unit_type_desc = 'IN_ROW_DATA'
    AND o.[type] IN ('U', 'V')
```

The main problem with the veil of similar requests in a competing product was eliminated, but the drawbacks of this implementation are that no additional parameters are passed to the `sys.dm_db_index_physical_stats` function that can restrict scanning of blatantly unnecessary indexes. In fact, this leads to obtaining information on all indexes in the system and unnecessary disk loads at the scanning stage.

It is important to mention that the data obtained from `sys.dm_db_index_physical_stats` is not permanently cached in the buffer pool, so minimizing physical reads when getting information about index fragmentation was one of the priority tasks during development of my application.

After a number of experiments, I managed to combine both approaches by dividing the scan into two parts. Initially, one large request determines the size of the partitions in advance by filtering those that are not included in the filtering range:

SQL

```
INSERT INTO #AllocationUnits (ContainerID, ReservedPages, UsedPages)
SELECT [container_id]
     , SUM([total_pages])
     , SUM([used_pages])
FROM sys.allocation_units WITH(NOLOCK)
GROUP BY [container_id]
HAVING SUM([total_pages]) BETWEEN @MinIndexSize AND @MaxIndexSize
```

Next, we get only those partitions that contain data to avoid unnecessary reads from empty indexes.

SQL

```
SELECT [object_id]
     , [index_id]
     , [partition_id]
     , [partition_number]
     , [rows]
     , [data_compression]
INTO #Partitions
FROM sys.partitions WITH(NOLOCK)
WHERE [object_id] > 255
    AND [rows] > 0
    AND [object_id] NOT IN (SELECT * FROM #ExcludeList)
```

Depending on the settings, only the types of indexes that the user wants to analyze are obtained (work with heaps, cluster/non-clustered indexes and columnstores is supported).

SQL

```
INSERT INTO #Indexes
SELECT ObjectID         = i.[object_id]
     , IndexID          = i.index_id
     , IndexName        = i.[name]
     , PagesCount       = a.ReservedPages
     , UnusedPagesCount = a.ReservedPages - a.UsedPages
     , PartitionNumber  = p.[partition_number]
     , RowsCount        = ISNULL(p.[rows], 0)
     , IndexType        = i.[type]
     , IsAllowPageLocks = i.[allow_page_locks]
     , DataSpaceID      = i.[data_space_id]
     , DataCompression  = p.[data_compression]
     , IsUnique         = i.[is_unique]
     , IsPK             = i.[is_primary_key]
     , FillFactorValue  = i.[fill_factor]
     , IsFiltered       = i.[has_filter]
FROM #AllocationUnits a
JOIN #Partitions p ON a.ContainerID = p.[partition_id]
JOIN sys.indexes i WITH(NOLOCK) ON i.[object_id] = p.[object_id]

                                AND p.[index_id] = i.[index_id]
```

```
WHERE i.[type] IN (0, 1, 2, 5, 6)
    AND i.[object_id] > 255
```

Afterwards, we add a little bit of magic, and... for all small indices, we determine the level of fragmentation by repeatedly calling the function `sys.dm_db_index_physical_stats` with full indication of all parameters.

SQL

```
INSERT INTO #Fragmentation (ObjectID, IndexID, PartitionNumber, Fragmentation)
SELECT i.ObjectID
     , i.IndexID
     , i.PartitionNumber
     , r.[avg_fragmentation_in_percent]
FROM #Indexes i
CROSS APPLY sys.dm_db_index_physical_stats_
    (@DBID, i.ObjectID, i.IndexID, i.PartitionNumber, 'LIMITED') r
WHERE i.PagesCount <= @PreDescribeSize
    AND r.[index_level] = 0
    AND r.[alloc_unit_type_desc] = 'IN_ROW_DATA'
    AND i.IndexType IN (0, 1, 2)
```

Next, we return all possible information to the client by filtering out the extra data:

SQL

```
SELECT i.ObjectID
     , i.IndexID
     , i.IndexName
     , ObjectName       = o.[name]
     , SchemaName       = s.[name]
     , i.PagesCount
     , i.UnusedPagesCount
     , i.PartitionNumber
     , i.RowsCount
     , i.IndexType
     , i.IsAllowPageLocks
     , u.TotalWrites
     , u.TotalReads
     , u.TotalSeeks
     , u.TotalScans
     , u.TotalLookups
     , u.LastUsage
     , i.DataCompression
     , f.Fragmentation
     , IndexStats       = STATS_DATE(i.ObjectID, i.IndexID)
     , IsLobLegacy      = ISNULL(lob.IsLobLegacy, 0)
     , IsLob            = ISNULL(lob.IsLob, 0)
     , IsSparse         = CAST(CASE WHEN p.ObjectID IS NULL THEN 0 ELSE 1 END AS BIT)
     , IsPartitioned    = CAST(CASE WHEN dds.[data_space_id] _
                         IS NOT NULL THEN 1 ELSE 0 END AS BIT)
     , FileGroupName    = fg.[name]
     , i.IsUnique
     , i.IsPK
     , i.FillFactorValue
     , i.IsFiltered
     , a.IndexColumns
     , a.IncludedColumns
FROM #Indexes i
JOIN sys.objects o WITH(NOLOCK) ON o.[object_id] = i.ObjectID
JOIN sys.schemas s WITH(NOLOCK) ON s.[schema_id] = o.[schema_id]
LEFT JOIN #AggColumns a ON a.ObjectID = i.ObjectID
                       AND a.IndexID = i.IndexID
LEFT JOIN #Sparse p ON p.ObjectID = i.ObjectID
LEFT JOIN #Fragmentation f ON f.ObjectID = i.ObjectID
```

```
                        AND f.IndexID = i.IndexID
                        AND f.PartitionNumber = i.PartitionNumber
LEFT JOIN (
    SELECT ObjectID      = [object_id]
         , IndexID       = [index_id]
         , TotalWrites   = NULLIF([user_updates], 0)
         , TotalReads    = NULLIF([user_seeks] + [user_scans] + [user_lookups], 0)
         , TotalSeeks    = NULLIF([user_seeks], 0)
         , TotalScans    = NULLIF([user_scans], 0)
         , TotalLookups  = NULLIF([user_lookups], 0)
         , LastUsage     = (
                               SELECT MAX(dt)
                               FROM (
                                   VALUES ([last_user_seek])
                                        , ([last_user_scan])
                                        , ([last_user_lookup])
                                        , ([last_user_update])
                               ) t(dt)
                           )
    FROM sys.dm_db_index_usage_stats WITH(NOLOCK)
    WHERE [database_id] = @DBID
) u ON i.ObjectID = u.ObjectID
   AND i.IndexID = u.IndexID
LEFT JOIN #Lob lob ON lob.ObjectID = i.ObjectID
                  AND lob.IndexID = i.IndexID
LEFT JOIN sys.destination_data_spaces dds WITH(NOLOCK) _
            ON i.DataSpaceID = dds.[partition_scheme_id]
            AND i.PartitionNumber = dds.[destination_id]
JOIN sys.filegroups fg WITH(NOLOCK) _
            ON ISNULL(dds.[data_space_id], i.DataSpaceID) = fg.[data_space_id]
WHERE o.[type] IN ('V', 'U')
    AND (
            f.Fragmentation >= @Fragmentation
        OR
            i.PagesCount > @PreDescribeSize
        OR
            i.IndexType IN (5, 6)
    )
```

After that, point requests determine the level of fragmentation for large indexes.

SQL

```
EXEC sp_executesql N'
DECLARE @DBID INT = DB_ID()
SELECT [avg_fragmentation_in_percent]
FROM sys.dm_db_index_physical_stats(@DBID, @ObjectID, @IndexID, @PartitionNumber, ''LIMITED'')
WHERE [index_level] = 0
    AND [alloc_unit_type_desc] = ''IN_ROW_DATA'''
    , N'@ObjectID int,@IndexID int,@PartitionNumber int'
    , @ObjectId = 1044198770, @IndexId = 1, @PartitionNumber = 1

EXEC sp_executesql N'
DECLARE @DBID INT = DB_ID()
SELECT [avg_fragmentation_in_percent]
FROM sys.dm_db_index_physical_stats(@DBID, @ObjectID, @IndexID, @PartitionNumber, ''LIMITED'')
WHERE [index_level] = 0
    AND [alloc_unit_type_desc] = ''IN_ROW_DATA'''
    , N'@ObjectID int,@IndexID int,@PartitionNumber int'
    , @ObjectId = 1552724584, @IndexId = 0, @PartitionNumber = 1
```

Due to such kind of approach, when generating requests, I managed to solve problems with scanning performance that were encountered in competitors' applications. This could have been the end of it, but in the process of development, a variety of new ideas gradually emerged which made it possible to expand the scope of application of my product.

Initially, the support for working with `WAIT_AT_LOW_PRIORITY` was implemented, and then it became possible to use `DATA_COMPRESSION` and `FILL_FACTOR` for rebuilding indexes.

The application has been "sprinkled" with previously unplanned functionality like maintenance of columnstores:

SQL

```
SELECT *
FROM (
    SELECT IndexID          = [index_id]
         , PartitionNumber  = [partition_number]
         , PagesCount       = SUM([size_in_bytes]) / 8192
         , UnusedPagesCount = ISNULL(SUM(CASE WHEN [state] = 1 _
                              THEN [size_in_bytes] END), 0) / 8192
         , Fragmentation    = CAST(ISNULL(SUM(CASE WHEN [state] = 1 _
                              THEN [size_in_bytes] END), 0)
                            * 100. / SUM([size_in_bytes]) AS FLOAT)
    FROM sys.fn_column_store_row_groups(@ObjectID)
    GROUP BY [index_id]
           , [partition_number]
) t
WHERE Fragmentation >= @Fragmentation
    AND PagesCount BETWEEN @MinIndexSize AND @MaxIndexSize
```

Or the ability to create nonclustered indexes based on information from `dm_db_missing_index`:

SQL

```
SELECT ObjectID     = d.[object_id]
     , UserImpact   = gs.[avg_user_impact]
     , TotalReads   = gs.[user_seeks] + gs.[user_scans]
```

```
    , TotalSeeks   = gs.[user_seeks]
    , TotalScans   = gs.[user_scans]
    , LastUsage    = ISNULL(gs.[last_user_scan], gs.[last_user_seek])
    , IndexColumns =
              CASE
                  WHEN d.[equality_columns] IS NOT NULL
                              _AND d.[inequality_columns] IS NOT NULL
                      THEN d.[equality_columns] + ', ' + d.[inequality_columns]
                  WHEN d.[equality_columns] IS NOT NULL AND d.[inequality_columns] IS NULL
                      THEN d.[equality_columns]
                  ELSE d.[inequality_columns]
              END
    , IncludedColumns = d.[included_columns]
FROM sys.dm_db_missing_index_groups g WITH(NOLOCK)
JOIN sys.dm_db_missing_index_group_stats gs WITH(NOLOCK) _
                    ON gs.[group_handle] = g.[index_group_handle]
JOIN sys.dm_db_missing_index_details d WITH(NOLOCK) _
                    ON g.[index_handle] = d.[index_handle]
WHERE d.[database_id] = DB_ID()
```

# Results and Plans

The crucial thing is that development plans do not end there, because I am craving to further develop this application. The next step is to add functionality for finding duplicate (**done**) or unused indexes (**done**), as well as implement full support for maintaining statistics (**done**) within SQL Server.

There are a lot of paid solutions on the market now. I want to believe that due to free positioning, more optimized queries and the availability of various useful gismos for someone, this product will definitely become useful in everyday tasks.

The latest version of the application can be downloaded on **GitHub**. The sources are in the same place.

I am looking forward to any feedback.

Thanks for reading! :)

# History

- 15th July, 2019: Initial version
- 27th July, 2019: 1.0.0.46 - Display duplicate and unused indexes
- 3rd August, 2019: 1.0.0.47 - Bugfix and new functionality in scan engine
- 31st August, 2019: 1.0.0.51 - Statistics maintenance, new options and bugfix
- 9th September, 2019: 1.0.0.52 - Fix issue with Azure
- 3rd November, 2019: 1.0.0.53 - Cannot insert duplicate key when scanning Azure SQL Database
- 7th December, 2019: 1.0.0.54 - Small improvements in GUI
- 15th December, 2019: 1.0.0.55 - Auto cell filter and several improvements in GUI
- 29th December, 2019: 1.0.0.56 - Show all indexes, change fix action for several rows, new options, improvements in query cancellation
- 7th January, 2020: 1.0.0.57 - Realtime stats, refactoring and bugfix
- 21st January, 2020: 1.0.0.58 - Improvements in database dialog, show drive disk space, bugfix in UI
- 2nd February, 2020: 1.0.0.59 - Async refresh databases, bugfix in script generation and early query cancellation
- 13th February, 2020: 1.0.0.60 - Bugfix during columnstore processing
- 29th March, 2020: 1.0.0.61 - New columns with index usage stats and improvements in UI
- 11th May, 2020: 1.0.0.62 - Bugfix during columnstore filtering, improvements for UI and unused index functionality

- 31st May, 2020: 1.0.0.63 - Move heaps to clustered CCL indexes, remove flicks when manually refresh database list, new options: "Ignore heaps with compression", "Ignore tables with rows count < 1000", bugfix and other small improvements in UI
- 7th June, 2020: 1.0.0.64 - Remove unused DevExpress libs, remove export functionality, resource optimization, small improvements in UI
- 29th June, 2020: 1.0.0.65 - Scrolling to maintenance index, improvements in sorting
- 27th April, 2021: 1.0.0.66 - Fixed issue with ignore `NO_RECOMPUTE` option during index maintenance, added new columns "No Recompute" & "Stats Sampled", display date-time columns in local time zone, possibility to exclude rows from grid via popup
- 27th June, 2021: 1.0.0.67 - Fixed '`STATISTICS_NORECOMPUTE`' is not a recognized `ALTER INDEX REBUILD PARTITION` option, fix issue with empty list of databases if any database metadata is corrupted, improvements in grid context menu, added possibility to truncate tables, improved auto-scroll, remove delay functionality
- 28th June, 2021: 1.0.0.68 - Additional tooltips for status bar, changes in `ErrorBox`
- 8th November, 2021: 1.0.0.69 - Ignore `UPDATE STATISTICS` for already up-to-date stats via options, update `StatsSampled`/`RowsSampled` columns during maintenance, bugfix during generation of truncate table/partition statement for columnstore indexes, improvements during searching into Server ComboBox control, remove save password functionality

# License



# About the Author

**Sergii Syrovatchenko**

Database Administrator Teamwork Commerce

Ukraine

SQL Server DBA/DB Developer with 10+ years of experience in SQL Server 2005-2019, Azure/GCP. Worked on high-load OLTP/DW projects and develops system tools for SQL Server. In depth understanding of SQL Server Engine and experience in working with big databases. Domain knowledge of ERP/CRM, crawlers, gambling and retail sales. Blogger, mentor and speaker at local SQL Server events.

# Comments and Discussions

**61 messages** have been posted for this article Visit **https://www.codeproject.com/Articles/5162340/SQL-Index-Manager-Free-GUI-Tool-for-Index-Maintena** to post and view comments on this article, or click **here** to get a print view with messages.