8 October 2025

Quiet Performance Wins: Scheduled Job for SQL Index Maintenance in Optimizely

by Stanisław Szołkowski

Tags: database (1) db (1) episerver (8) indexes (1) jobs (3) maintenance (2) optimizely (8) performance (1) scheduled jobs (2)

As Optimizely CMS projects grow, it's not uncommon to introduce custom tables—whether for integrations, caching, or specialized business logic. But with great schema comes great responsibility: SQL Server indexes and statistics need love too.

While Optimizely handles its own data structures well, custom tables can quietly degrade performance if left unchecked. Optimizely Commerce includes a built-in job for index maintenance, but if your solution uses only CMS, this functionality will be missing. In this post, I'll show how to automate index and statistics maintenance using a scheduled job.

Why You Should Care

SQL Server relies heavily on up-to-date statistics and healthy indexes to optimize query execution. Fragmented indexes and stale stats can lead to slow queries, increased CPU usage, and unhappy editors.

If you're adding custom tables to your CMS database, especially ones that grow over time, you should consider regular maintenance. And what better way than a scheduled job that runs quietly in the background?

The Scheduled Job

Here's a simple implementation of a scheduled job that performs index and statistics maintenance on selected custom tables. You can trigger it manually or schedule it via Optimizely's job system.

View Fullscreen Copy

```
/// <summary>
/// Automated database index maintenance job that runs o
/// This job analyzes index fragmentation and performs m
/// </summary>
[ScheduledPlugIn(
    DisplayName = "Database Index Maintenance Scheduled 
    SortIndex = 20000)]
public sealed class DatabaseIndexMaintenanceScheduledJob
{
    private bool _stopRequested;
    private readonly IConfiguration _configuration;

    /// <summary>
    /// Constructor injecting configuration for database
    /// Sets IsStoppable to allow manual termination of 
    /// </summary>
    public DatabaseIndexMaintenanceScheduledJob(IConfigu
    {
        _configuration = configuration;

        // Allow administrators to stop the job if it's 
        IsStoppable = true;
    }

    /// <summary>
    /// Handles stop requests by setting a flag that's c
    /// This allows graceful cancellation between mainte
    /// </summary>
    public override void Stop()
    {
        _stopRequested = true;
    }

    /// <summary>
    /// Main entry point for the scheduled job execution
    /// Retrieves the database connection string and del
    /// </summary>
    public override string Execute()
    {
        // Get the connection string from configuration 
        var connectionString = _configuration.GetConnect

        // Validate connection string exists before proc
        var result = !string.IsNullOrEmpty(connectionStr
        ? ExecuteInternal(connectionString)
        : "Connection string is empty";

        return result;
    }

    /// <summary>
    /// Core maintenance logic that analyzes and optimiz
    /// Uses a three-phase approach:
    /// 1. Query all indexes and measure their fragmenta
    /// 2. Rebuild or reorganize indexes based on fragme
    /// 3. Update statistics for tables with fragmented 
    /// </summary>
    private string ExecuteInternal(string connectionStri
    {
        // StringBuilder accumulates log messages for th
        var log = new StringBuilder();
        try
        {
            // Establish database connection using 'usin
            using var conn = new SqlConnection(connectio
            conn.Open();

            log.AppendLine("Starting index maintenance..

            // Query SQL Server's Dynamic Management Vie
            // sys.dm_db_index_physical_stats provides f
            var indexQuery = @"
    SELECT OBJECT_SCHEMA_NAME(s.[object_id])
            OBJECT_NAME(s.[object_id]) AS Ta
            i.name AS IndexName,
            s.avg_fragmentation_in_percent A
    FROM sys.dm_db_index_physical_stats(DB_I
    JOIN sys.indexes i
        ON s.[object_id] = i.[object_id]
        AND s.index_id = i.index_id
    WHERE i.type_desc IN ('CLUSTERED', 'NONC
        AND s.page_count > 100;"; // Only an

using var cmd = new SqlCommand(indexQuery, c
using var reader = cmd.ExecuteReader();

// Phase 1: Collect all indexes and their fr
// Store in a list to avoid maintaining an o
var indexList = new List<(string Schema, str
while (reader.Read() && !_stopRequested)
{
    var schema = reader.GetString(0);      /
    var table = reader.GetString(1);       /
    var index = reader.GetString(2);       /
    var frag = reader.GetDouble(3);        /

    indexList.Add((schema, table, index, fra
}

// Close the reader before executing mainten
reader.Close();

// Phase 2: Perform index maintenance based
// Industry best practices: REBUILD > 30%, R
foreach (var (schema, table, index, frag) in
{
    // Check for stop request between each i
    if (_stopRequested) break;

    // Use pattern matching to determine the
    var sql = frag switch
    {
        // Severe fragmentation (>30%): REBU
        // Add optionally "WITH ONLINE = ON"
        > 30 => $"ALTER INDEX [{index}] ON [

        // Moderate fragmentation (5-30%): R
        // This is always an online operatio
        > 5 => $"ALTER INDEX [{index}] ON [{

        // Low fragmentation (<5%): No actio
        _ => null
    };

    // Execute the maintenance command if an
    if (sql != null)
    {
        log.AppendLine($"Maintaining index [
        using var alterCmd = new SqlCommand(
        // Set a generous timeout for long-r
        alterCmd.CommandTimeout = 180;
        alterCmd.ExecuteNonQuery();
    }
}

// Phase 3: Update statistics for tables tha
// Statistics help the query optimizer make
foreach (var (schema, table, _, frag) in ind
{
    // Check for stop request between each s
    if (_stopRequested) break;
```

```
                // Only update statistics for tables wit
                // SAMPLE 50 PERCENT balances accuracy w
                var sql = frag switch
                {
                    > 5 => $"UPDATE STATISTICS [{schema}]
                    _ => null
                };

                if (sql != null)
                {
                    log.AppendLine($"Updating statistics
                    using var statsCmd = new SqlCommand(
                    // Set a generous timeout for long-r
                    statsCmd.CommandTimeout = 180;
                    statsCmd.ExecuteNonQuery();
                    log.AppendLine("Statistics updated."
                }
            }

            conn.Close();
        }
        catch (Exception ex)
        {
            // Log any errors that occur during maintenan
            log.AppendLine($"Error: {ex.Message}");
        }

        // Return the accumulated log as the job executi
        return log.ToString();
    }
}
```

// Performance Considerations

/// During Execution

REBUILD operations:

- High CPU usage (50-80% spike for 1-5 minutes)
- Locks the table (unless `ONLINE = ON` on Enterprise Edition)
- Should be run during maintenance windows

REORGANIZE operations:

- Minimal CPU impact (10-20%)
- Online operation (no blocking)
- Safe to run during business hours

/// After Execution

Typical improvements on databases with 30%+ fragmentation:

- Query performance: 15-40% faster
- CPU usage: 10-15% reduction
- Page I/O: 20-30% reduction

Note: Benefits are most noticeable on:

- Tables with > 1M rows
- Queries with table/index scans
- Reports and analytics queries

// What Gets Maintained?

This job analyzes **all indexes** across your entire database, including:

Table Type	Examples	Maintained?
Your custom tables	`CustomOrderCache`, `IntegrationLog`	Yes
Optimizely CMS core	`tblContent`, `tblContentProperty`, `tblWorkContent`	Yes
Commerce tables	`OrderGroup`, `Shipment`, `LineItem`	Yes (if installed)
ASP.NET Identity	`AspNetUsers`, `AspNetRoles`	Yes

/// Is This Safe?

Yes, generally. Index maintenance operations are safe on all tables. However:

- **REBUILD operations** may briefly lock tables on Standard Edition
- Large Optimizely tables (like `tblContentProperty`) may take several minutes
- First run might take 10-20 minutes on established sites

/// Should You Filter?

For production safety, consider filtering to custom tables only if:

- You have a very large CMS database (50GB+)
- You're on SQL Server Standard Edition (no ONLINE rebuilds)
- You want to minimize maintenance window impact

// Understanding Statistics Updates

`UPDATE STATISTICS` ensures the query optimizer has accurate data about:

- Row counts
- Data distribution
- Index selectivity

The `SAMPLE 50 PERCENT` option:

- Faster than FULLSCAN
- Accurate enough for most scenarios
- Use `WITH FULLSCAN` for critical tables if needed

// Notes

UPDATE STATISTICS ensures the query planner has fresh data to work with.

You can extend the job to log execution time or errors using Optimizely's logging framework.

Provided job targets ALL indexes in the database, including Optimizely's core tables. Logic can be adjusted to target only whitelisted tables by changing index query:

```
SELECT OBJECT_SCHEMA_NAME(s.[object_id]) AS SchemaNa
       OBJECT_NAME(s.[object_id]) AS TableName,
       i.name AS IndexName,
       s.avg_fragmentation_in_percent AS Frag
FROM sys.dm_db_index_physical_stats(DB_ID(), NULL, N
JOIN sys.indexes i ON s.[object_id] = i.[object_id]
WHERE i.type_desc IN ('CLUSTERED', 'NONCLUSTERED')
```

```
        AND s.page_count > 100
        AND OBJECT_NAME(s.[object_id]) IN (XXXX)
```

Query can be also adjusted to filter by schema or prefix.

// Troubleshooting

Job times out:

- Increase the SQL command timeout
- Consider running REORGANIZE on specific indexes rather than ALL

Permission errors:

- Ensure the app pool identity has `db_ddladmin` role
- Check Azure SQL firewall rules if using DXP

High CPU during execution:

- Move to off-peak hours
- Add `WITH (ONLINE = ON)` option if using Enterprise edition

// Summary

This kind of job is especially useful in environments where custom tables are updated frequently but not covered by Optimizely's internal maintenance routines. It's a small addition that can yield big performance wins.

If you're deploying to DXP, make sure the job is safe to run in production and doesn't interfere with other scheduled tasks. It is usually good to run this job weekly during the low traffic hours.

Read more articles | subscribe via RSS