

## EXECUTIVE SUMMARY

During development of a new mobile application integrating with a CMMS platform, initial stored procedures for paginated data loading presented critical performance and maintainability issues:

### **Problem #1 – Performance Crisis**

Inconsistent and unacceptable execution times averaging 27 seconds per page load, creating a poor user experience and system bottlenecks.

### **Problem #2 – Maintenance Nightmare**

The original approach required building individual stored procedures for each data list, resulting in 350+ procedures across modules (Assets, Work Orders, Parts, Repair Centers, Departments, Classifications, etc.).

As Database Administrator and Lead SQL Developer, I was tasked with architecting a comprehensive solution.

## SOLUTION ARCHITECTURE

### **Strategic Approach**

Instead of managing hundreds of individual procedures, I designed a single, universal stored procedure with parameterized inputs to handle all CMMS modules. This approach prioritized:

- Development efficiency and consistency
- Simplified maintenance with schema changes
- Centralized optimization and performance tuning
- Reduced database storage footprint

## Technical Evolution

My optimization progressed through two iterations

1. **Initial Optimization:** Recursive CTE implementation reduced execution time to ~250ms—acceptable, but not optimal.
2. **Final Architecture:** Three-layered CTE design with zero recursion achieved consistent 17ms average execution times across all table sizes.

## Performance Results

- **Before:** 27,000ms average execution time
- **After:** 17ms average execution time
- **Improvement:** 99.94% performance gain
- **Consistency:** Execution time independent of table size
- **Maintainability:** 350+ procedures reduced to 1 universal solution

## Technical Implementation Notes

### Dynamic SQL Approach

The parameterized design necessitated dynamically generated SQL. Two common concerns addressed:

### Query Plan Recompilation

While dynamic SQL requires new query plans per execution rather than cached plans, the compilation overhead proved negligible compared to the massive performance gains from tailored query execution.

### SQL Injection Security

This implementation maintains security through controlled parameter handling and exclusive use by internal applications with no direct user input exposure.

The solution demonstrates how strategic architectural decisions can transform both performance and maintainability in enterprise database systems.

## STORED PROCEDURE

```
CREATE PROCEDURE [dbo].[GetPagedData]
    @Tablename varchar ( 50 ),
    @TablePK varchar ( 50 ),
    @Page int,
    @MaxRows int,
    @OrderString varchar ( max ),
    @SearchString varchar ( max ),
    @Fields varchar ( max ),
    @Join varchar ( max ),
    @IncludeRowVersion bit

AS
SET NOCOUNT ON

/*
Purpose: Retrieve paginated, filtered, and ordered data from specified table(s).
Designed for: All mobile app page loads requiring a paginated list.
Author: Jamie Whitbeck

Design notes:
- Initial CTE selects only primary keys, applies row numbering, and filters duplicates to optimize page calculations.
- Data fields are joined later, only for the selected page rows, to improve performance.
- NOLOCK is used intentionally for high-concurrency, read-only operations where dirty reads are acceptable.
- All inputs are controlled by internal mobile app calls – zero risk of SQL injection.
- Dynamic SQL enables flexibility, generating a new query plan per call:
    - The cost of recompilation is negligible in this context.
    - Recompilation often results in less I/O than executing with a stale plan, improving overall performance.
*/

DECLARE @SqlString nvarchar ( max )

-- Build initial string with only the primary key, row ordering, and a duplicate check from the primary table
SET @SqlString = '
WITH CTE_AllRecords AS (
    SELECT
        ' + @TableName + '.' + @TablePK + '
        ,RowNum = ROW_NUMBER ( ) OVER (ORDER BY ' + @OrderString + ')
```

```

        ,DuplicateCheck = ROW_NUMBER () OVER (PARTITION BY ' + @TableName + '.' + @TablePK + ' ORDER BY ' + @TableName + '.' +
@TablePK + ')
        FROM ' + @TableName + ' WITH(NoLock) '

-- If there are other tables being joined, add the JOIN statement to the string
-- This first join operates only on primary keys and does not returning data fields yet, optimizing performance
IF IsNull(@Join, '') != ''
BEGIN
    SET @SqlString = @SqlString + ' ' + @Join
END

-- If there are conditions in the WHERE clause, add it to the string
IF IsNull(@SearchString, '') != ''
BEGIN
    SET @SqlString = @SqlString + ' WHERE ' + @SearchString
END

-- Build the layered CTE structure
-- Add count of pages for application to determine if to display next page button and final page number for jump to page
functionality
-- Add row numbers for start and end on the dataset being returned
-- Add the data fields to be returned
SET @SqlString = @SqlString + '
)
,CTE_RowNums AS (
    SELECT
        AR.' + @TablePK + '
        ,RowNum = ROW_NUMBER () OVER (ORDER BY RowNum)
    FROM CTE_AllRecords AR
    WHERE DuplicateCheck = 1
)
,CTE_PageCount AS (
    SELECT
        RecordCount
        ,[PageCount] = CAST(RecordCount AS decimal(18,2))/CAST(' + CAST(@MaxRows as varchar(max)) + ' AS decimal(18,2))
        ,StartRow = ((' + CAST(@Page as varchar(max)) + ' - 1) * ' + CAST(@MaxRows as varchar(max)) + ') + 1
        ,EndRow = ((( ' + CAST(@Page as varchar(max)) + ' - 1) * ' + CAST(@MaxRows as varchar(max)) + ') + ' + CAST(@MaxRows as
varchar(max)) + ')
        FROM (SELECT RecordCount = COUNT(' + @TablePK + ') FROM CTE_RowNums) AS A
)

```

```
SELECT DISTINCT
    RN.RowNum
    ,PC.[PageCount]
    ,PC.RecordCount
    ,PC.StartRow
    ,PC.EndRow
    , ' + @Fields

-- Format the string to remove bad chars
IF RIGHT(RTrim(@SqlString),1) = ','
BEGIN
    SET @SqlString = LEFT(RTrim(@SqlString),(LEN(RTrim(@SqlString))-1))
END

-- Continue building string by adding the FROM, JOIN, WHERE, and ORDER BY clauses
SET @SqlString = @SqlString + '
    FROM CTE_PageCount PC
    INNER JOIN (SELECT * FROM CTE_RowNums) RN
        ON RN.RowNum BETWEEN PC.StartRow AND PC.EndRow
    INNER JOIN ' + @TableName + ' WITH(NoLock)
        ON RN.' + @TablePK + ' = ' + @TableName + '.' + @TablePK
-- Join the tables again, this time to return the actual data fields requested
IF IsNull(@Join,'') != ''
BEGIN
    SET @SqlString = @SqlString + ' ' + @Join
END

IF IsNull(@SearchString,'') != ''
BEGIN
    SET @SqlString = @SqlString + ' WHERE ' + @SearchString
END

SET @SqlString = @SqlString + ' ORDER BY RN.RowNum ASC'

SET NOCOUNT OFF

-- Execute the dynamic SQL
EXEC sp_executesql @SqlString
```

## SAMPLE EXECUTION CALL

-- Page 3, return Asset ID and AssetName for assets assigned to the primary repair center, ordered by AssetName, 20 items/page

```
EXEC GetPagedData 'Asset', 'AssetPK', 1, 20, 'AssetName ASC', 'RepairCenter.RepairCenterPK = 1', 'Asset.AssetID, Asset.AssetName',
'INNER JOIN RepairCenter ON Asset.RepairCenterPK = RepairCenter.RepairCenterPK', 0
```

## DYNAMIC SQL GENERATED BY SAMPLE EXECUTION CALL

```
WITH CTE_AllRecords AS (
    SELECT
        Asset.AssetPK
        ,RowNum = ROW_NUMBER () OVER (ORDER BY AssetName ASC)
        ,DuplicateCheck = ROW_NUMBER () OVER (PARTITION BY Asset.AssetPK ORDER BY Asset.AssetPK)
    FROM Asset WITH(NOLOCK) INNER JOIN RepairCenter ON Asset.RepairCenterPK = RepairCenter.RepairCenterPK
    WHERE RepairCenter.RepairCenterPK = 1
)
,CTE_RowNums AS (
    SELECT
        AR.AssetPK
        ,RowNum = ROW_NUMBER () OVER (ORDER BY RowNum)
    FROM CTE_AllRecords AR
    WHERE DuplicateCheck = 1
)
,CTE_PageCount AS (
    SELECT
        RecordCount
        ,[PageCount] = CAST(RecordCount AS decimal(18,2))/CAST(20 AS decimal(18,2))
        ,StartRow = ((3 - 1) * 20) + 1
        ,EndRow = (((3 - 1) * 20) + 20)
    FROM (SELECT RecordCount = COUNT(AssetPK) FROM CTE_RowNums) AS A
```

```
)  
  
SELECT DISTINCT  
    RN.RowNum  
    ,PC.[PageCount]  
    ,PC.RecordCount  
    ,PC.StartRow  
    ,PC.EndRow  
    , Asset.AssetId, Asset.AssetName  
FROM CTE_PageCount PC  
INNER JOIN (SELECT * FROM CTE_RowNums) RN  
    ON RN.RowNum BETWEEN PC.StartRow AND PC.EndRow  
INNER JOIN Asset WITH(NOLOCK)  
    ON RN.AssetId = Asset.AssetId INNER JOIN RepairCenter ON Asset.RepairCenterPK = RepairCenter.RepairCenterPK  
WHERE RepairCenter.RepairCenterPK = 1  
ORDER BY RN.RowNum ASC
```

# SAMPLE RESULTSET FROM A DATABASE IN MY DEV ENVIRONMENT

	RowNum	PageCount	RecordCount	StartRow	EndRow	AssetID	AssetName
1	41	49.850000000000000000000000	997	41	60	WWA000005-1	Analyzer Chlorine Well 13
2	42	49.850000000000000000000000	997	41	60	WWA000009-1	Analyzer Chlorine Well 2
3	43	49.850000000000000000000000	997	41	60	WWA000021-1	Analyzer Chlorine Well 4
4	44	49.850000000000000000000000	997	41	60	WWA000007-1	Analyzer Chlorine Well 9
5	45	49.850000000000000000000000	997	41	60	WWA000001-2	Analyzer Fluoride Well 1
6	46	49.850000000000000000000000	997	41	60	WWA000002-2	Analyzer Fluoride Well 10
7	47	49.850000000000000000000000	997	41	60	WWA000010-3	Analyzer Fluoride Well 12
8	48	49.850000000000000000000000	997	41	60	WWA000005-2	Analyzer Fluoride Well 13
9	49	49.850000000000000000000000	997	41	60	WWA000009-2	Analyzer Fluoride Well 2
10	50	49.850000000000000000000000	997	41	60	WWA000021-2	Analyzer Fluoride Well 4
11	51	49.850000000000000000000000	997	41	60	WWA000007-7	Analyzer Fluroide Regional
12	52	49.850000000000000000000000	997	41	60	WWA000007-2	Analyzer Fluroide Well 9
13	53	49.850000000000000000000000	997	41	60	WWA000007-3	Analyzer PH
14	54	49.850000000000000000000000	997	41	60	WWA000002-3	Analyzer PH
15	55	49.850000000000000000000000	997	41	60	WWA000001-3	Analyzer PH
16	56	49.850000000000000000000000	997	41	60	WWA000009-3	Analyzer PH
17	57	49.850000000000000000000000	997	41	60	WWA000007-8	Analyzer PH Regional
18	58	49.850000000000000000000000	997	41	60	WWA000021-5	Analyzer Turbidity Backwash Well 4
19	59	49.850000000000000000000000	997	41	60	WWA000021-4	Analyzer Turbidity Finished Well 4
20	60	49.850000000000000000000000	997	41	60	WWA000007-4	Analyzer Turbidity Regional



## EXECUTION STATISTICS

### Notes:

*This database contains approximately 130,000 assets, which would be considered a moderately high count for this platform.*

*Execution performed after server restart to ensure no cached data was being accessed.*

### SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

### SQL Server parse and compile time:

CPU time = 0 ms, elapsed time = 6 ms.

(20 rows affected)

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

Table 'Asset'. Scan count 3, logical reads 30, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

Table 'RepairCenter'. Scan count 0, logical reads 6, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

### SQL Server Execution Times:

CPU time = 16 ms, elapsed time = 1 ms.

### SQL Server Execution Times:

CPU time = 16 ms, elapsed time = 8 ms.

### SQL Server Execution Times:

CPU time = 16 ms, elapsed time = 8 ms.