I'll continue with the comprehensive SQL to SQL data migration implementation in C#:

metric.Stopwatch.Stop();

metric.EndTime = DateTime.Now;

metric.RowCount = rowCount;

metric.ElapsedTime = metric.Stopwatch.Elapsed;

metric.RowsPerSecond = rowCount / Math.Max(1, metric.ElapsedTime.TotalSeconds);

\_logger.LogInformation(

"Completed migration for table {TableName}: {RowCount} rows in {ElapsedTime}",

tableName, rowCount, metric.ElapsedTime);

}

}

public void UpdateTableProgress(string tableName, int currentRowCount)

{

if (\_tableMetrics.TryGetValue(tableName, out var metric))

{

var currentElapsed = metric.Stopwatch.Elapsed;

var rowsPerSecond = currentRowCount / Math.Max(1, currentElapsed.TotalSeconds);

\_logger.LogDebug(

"Table {TableName} progress: {CurrentRowCount} rows, {RowsPerSecond:F1} rows/sec",

tableName, currentRowCount, rowsPerSecond);

}

}

public class TableMetrics

{

public string TableName { get; set; }

public DateTime StartTime { get; set; }

public DateTime EndTime { get; set; }

public TimeSpan ElapsedTime { get; set; }

public int RowCount { get; set; }

public double RowsPerSecond { get; set; }

public Stopwatch Stopwatch { get; } = new Stopwatch();

}

}

public class DataValidationService

{

private readonly ILogger \_logger;

private readonly string \_sourceConnectionString;

private readonly string \_targetConnectionString;

public DataValidationService(IConfiguration config, ILogger logger)

{

\_logger = logger;

\_sourceConnectionString = config.GetConnectionString("Source");

\_targetConnectionString = config.GetConnectionString("Target");

}

public async Task ValidateTableDataAsync(TableMapping mapping)

{

\_logger.LogInformation("Starting data validation for table {SourceTable} to {TargetTable}",

mapping.SourceTable, mapping.TargetTable);

var validations = new List<ValidationResult>();

// Validate row counts

var rowCountValidation = await ValidateRowCountsAsync(mapping);

validations.Add(rowCountValidation);

// Validate column existence and data types

var columnValidation = await ValidateColumnsAsync(mapping);

validations.Add(columnValidation);

// Validate key data samples

if (mapping.IncrementalColumn != null)

{

var keyDataValidation = await ValidateKeyDataAsync(mapping);

validations.Add(keyDataValidation);

}

// Log validation results

foreach (var validation in validations)

{

if (validation.Success)

{

\_logger.LogInformation("Validation passed: {ValidationType} for {TableName}",

validation.ValidationType, mapping.SourceTable);

}

else

{

\_logger.LogError("Validation failed: {ValidationType} for {TableName}: {ErrorMessage}",

validation.ValidationType, mapping.SourceTable, validation.ErrorMessage);

}

}

}

private async Task<ValidationResult> ValidateRowCountsAsync(TableMapping mapping)

{

var result = new ValidationResult

{

ValidationType = "RowCount",

TableName = mapping.SourceTable

};

try

{

var sourceCount = await GetTableRowCountAsync(\_sourceConnectionString,

mapping.SourceSchema, mapping.SourceTable, mapping.CustomWhere);

var targetCount = await GetTableRowCountAsync(\_targetConnectionString,

mapping.TargetSchema, mapping.TargetTable, mapping.CustomWhere);

if (sourceCount == targetCount)

{

result.Success = true;

result.Details = $"Source: {sourceCount}, Target: {targetCount}";

}

else

{

result.Success = false;

result.ErrorMessage = $"Row count mismatch. Source: {sourceCount}, Target: {targetCount}";

}

}

catch (Exception ex)

{

result.Success = false;

result.ErrorMessage = $"Error validating row counts: {ex.Message}";

}

return result;

}

private async Task<ValidationResult> ValidateColumnsAsync(TableMapping mapping)

{

var result = new ValidationResult

{

ValidationType = "ColumnSchema",

TableName = mapping.SourceTable

};

try

{

var sourceColumns = await GetTableColumnsAsync(\_sourceConnectionString,

mapping.SourceSchema, mapping.SourceTable);

var targetColumns = await GetTableColumnsAsync(\_targetConnectionString,

mapping.TargetSchema, mapping.TargetTable);

var errors = new List<string>();

foreach (var colMapping in mapping.ColumnMappings)

{

// Validate source column exists

if (!sourceColumns.ContainsKey(colMapping.SourceColumn))

{

errors.Add($"Source column '{colMapping.SourceColumn}' does not exist");

continue;

}

// Validate target column exists

if (!targetColumns.ContainsKey(colMapping.TargetColumn))

{

errors.Add($"Target column '{colMapping.TargetColumn}' does not exist");

continue;

}

// Validate data types are compatible

var sourceType = sourceColumns[colMapping.SourceColumn];

var targetType = targetColumns[colMapping.TargetColumn];

if (!AreTypesCompatible(sourceType, targetType))

{

errors.Add($"Column type mismatch for '{colMapping.SourceColumn}'/'{colMapping.TargetColumn}': {sourceType} vs {targetType}");

}

}

if (errors.Count == 0)

{

result.Success = true;

result.Details = $"All {mapping.ColumnMappings.Count} columns validated successfully";

}

else

{

result.Success = false;

result.ErrorMessage = string.Join("; ", errors);

}

}

catch (Exception ex)

{

result.Success = false;

result.ErrorMessage = $"Error validating columns: {ex.Message}";

}

return result;

}

private async Task<ValidationResult> ValidateKeyDataAsync(TableMapping mapping)

{

var result = new ValidationResult

{

ValidationType = "KeyData",

TableName = mapping.SourceTable

};

try

{

// Choose a column to validate (preferably the key or incremental column)

var columnToValidate = mapping.IncrementalColumn;

// Get a small sample of data to validate

var sourceSample = await GetColumnSampleAsync(\_sourceConnectionString,

mapping.SourceSchema, mapping.SourceTable, columnToValidate);

var matchingCount = 0;

var mismatchDetails = new List<string>();

foreach (var (key, value) in sourceSample)

{

var targetValue = await GetValueByKeyAsync(\_targetConnectionString,

mapping.TargetSchema, mapping.TargetTable, columnToValidate, key);

if (AreValuesEqual(value, targetValue))

{

matchingCount++;

}

else

{

mismatchDetails.Add($"Key {key}: Source={value}, Target={targetValue}");

}

if (mismatchDetails.Count >= 5)

break; // Limit the number of mismatch details

}

if (mismatchDetails.Count == 0)

{

result.Success = true;

result.Details = $"All {sourceSample.Count} sample keys matched";

}

else

{

result.Success = false;

result.ErrorMessage = $"{mismatchDetails.Count} of {sourceSample.Count} keys mismatched: {string.Join("; ", mismatchDetails)}";

}

}

catch (Exception ex)

{

result.Success = false;

result.ErrorMessage = $"Error validating key data: {ex.Message}";

}

return result;

}

private async Task<int> GetTableRowCountAsync(string connectionString, string schema, string tableName, string whereClause = null)

{

await using var connection = new SqlConnection(connectionString);

await connection.OpenAsync();

var query = $"SELECT COUNT(\*) FROM [{schema}].[{tableName}]";

if (!string.IsNullOrEmpty(whereClause))

{

query += $" WHERE {whereClause}";

}

await using var command = new SqlCommand(query, connection);

return Convert.ToInt32(await command.ExecuteScalarAsync());

}

private async Task<Dictionary<string, string>> GetTableColumnsAsync(string connectionString, string schema, string tableName)

{

var columns = new Dictionary<string, string>();

await using var connection = new SqlConnection(connectionString);

await connection.OpenAsync();

var query = @"

SELECT COLUMN\_NAME, DATA\_TYPE, CHARACTER\_MAXIMUM\_LENGTH

FROM INFORMATION\_SCHEMA.COLUMNS

WHERE TABLE\_SCHEMA = @Schema AND TABLE\_NAME = @TableName";

await using var command = new SqlCommand(query, connection);

command.Parameters.AddWithValue("@Schema", schema);

command.Parameters.AddWithValue("@TableName", tableName);

await using var reader = await command.ExecuteReaderAsync();

while (await reader.ReadAsync())

{

var columnName = reader.GetString(0);

var dataType = reader.GetString(1);

var maxLength = reader.IsDBNull(2) ? null : (int?)reader.GetInt32(2);

var typeStr = dataType;

if (maxLength.HasValue && maxLength.Value != -1)

{

typeStr += $"({maxLength})";

}

else if (maxLength.HasValue && maxLength.Value == -1)

{

typeStr += "(MAX)";

}

columns[columnName] = typeStr;

}

return columns;

}

private async Task<Dictionary<string, object>> GetColumnSampleAsync(string connectionString,

string schema, string tableName, string columnName, int sampleSize = 10)

{

var sample = new Dictionary<string, object>();

await using var connection = new SqlConnection(connectionString);

await connection.OpenAsync();

// Get a sample of rows

var query = $"SELECT TOP {sampleSize} {columnName} FROM [{schema}].[{tableName}] ORDER BY {columnName}";

await using var command = new SqlCommand(query, connection);

await using var reader = await command.ExecuteReaderAsync();

while (await reader.ReadAsync())

{

var value = reader.GetValue(0);

sample[value.ToString()] = value;

}

return sample;

}

private async Task<object> GetValueByKeyAsync(string connectionString,

string schema, string tableName, string columnName, string keyValue)

{

await using var connection = new SqlConnection(connectionString);

await connection.OpenAsync();

var query = $"SELECT TOP 1 {columnName} FROM [{schema}].[{tableName}] WHERE {columnName} = @KeyValue";

await using var command = new SqlCommand(query, connection);

command.Parameters.AddWithValue("@KeyValue", keyValue);

var result = await command.ExecuteScalarAsync();

return result ?? DBNull.Value;

}

private bool AreTypesCompatible(string sourceType, string targetType)

{

// Basic compatibility check - this could be expanded for more detailed checks

if (sourceType == targetType)

return true;

// Check numeric type compatibility

var numericTypes = new[] { "int", "bigint", "smallint", "tinyint", "decimal", "numeric", "float", "real", "money", "smallmoney" };

if (numericTypes.Any(t => sourceType.StartsWith(t)) && numericTypes.Any(t => targetType.StartsWith(t)))

return true;

// Check string type compatibility

var stringTypes = new[] { "char", "varchar", "nchar", "nvarchar", "text", "ntext" };

if (stringTypes.Any(t => sourceType.StartsWith(t)) && stringTypes.Any(t => targetType.StartsWith(t)))

return true;

// Check date type compatibility

var dateTypes = new[] { "datetime", "smalldatetime", "date", "time", "datetime2", "datetimeoffset" };

if (dateTypes.Any(t => sourceType.StartsWith(t)) && dateTypes.Any(t => targetType.StartsWith(t)))

return true;

return false;

}

private bool AreValuesEqual(object value1, object value2)

{

if (value1 == null && value2 == null)

return true;

if (value1 == null || value2 == null)

return false;

if (value1 is DBNull && value2 is DBNull)

return true;

if (value1 is DBNull || value2 is DBNull)

return false;

// Handle different numeric types

if (value1 is IConvertible && value2 is IConvertible)

{

try

{

var type = Type.GetTypeCode(value1.GetType());

switch (type)

{

case TypeCode.Int16:

case TypeCode.Int32:

case TypeCode.Int64:

case TypeCode.UInt16:

case TypeCode.UInt32:

case TypeCode.UInt64:

case TypeCode.Byte:

case TypeCode.SByte:

return Convert.ToInt64(value1) == Convert.ToInt64(value2);

case TypeCode.Decimal:

case TypeCode.Double:

case TypeCode.Single:

// Allow for small floating-point rounding differences

return Math.Abs(Convert.ToDouble(value1) - Convert.ToDouble(value2)) < 0.000001;

case TypeCode.DateTime:

var dt1 = Convert.ToDateTime(value1);

var dt2 = Convert.ToDateTime(value2);

return Math.Abs((dt1 - dt2).TotalSeconds) < 1; // Allow 1 second difference

}

}

catch

{

// If conversion fails, fall back to string comparison

}

}

// Default comparison

return value1.ToString() == value2.ToString();

}

}

public class ValidationResult

{

public string ValidationType { get; set; }

public string TableName { get; set; }

public bool Success { get; set; }

public string Details { get; set; }

public string ErrorMessage { get; set; }

}

public class StateCleanupService

{

private readonly ILogger \_logger;

private readonly string \_stateStorePath;

private readonly TimeSpan \_retentionPeriod;

public StateCleanupService(IConfiguration config, ILogger logger)

{

\_logger = logger;

\_stateStorePath = config.GetValue<string>("Settings:StateStorePath", "migrationState.json");

\_retentionPeriod = TimeSpan.FromDays(config.GetValue<int>("Settings:StateRetentionDays", 30));

}

public void ArchiveState()

{

try

{

if (!File.Exists(\_stateStorePath))

return;

var stateContent = File.ReadAllText(\_stateStorePath);

// Create archives directory if it doesn't exist

var archiveDir = Path.Combine(Path.GetDirectoryName(\_stateStorePath) ?? ".", "archives");

if (!Directory.Exists(archiveDir))

{

Directory.CreateDirectory(archiveDir);

}

// Archive the current state with timestamp

var timestamp = DateTime.Now.ToString("yyyyMMdd\_HHmmss");

var archivePath = Path.Combine(

archiveDir,

$"{Path.GetFileNameWithoutExtension(\_stateStorePath)}\_{timestamp}{Path.GetExtension(\_stateStorePath)}");

File.WriteAllText(archivePath, stateContent);

\_logger.LogInformation("Migration state archived to {ArchivePath}", archivePath);

// Clean up old archives

CleanupOldArchives(archiveDir);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error archiving migration state");

}

}

private void CleanupOldArchives(string archiveDir)

{

try

{

var cutoffDate = DateTime.Now - \_retentionPeriod;

var files = Directory.GetFiles(archiveDir, "\*.json");

foreach (var file in files)

{

var fileInfo = new FileInfo(file);

if (fileInfo.CreationTime < cutoffDate)

{

fileInfo.Delete();

\_logger.LogInformation("Deleted old state archive {FileName}", fileInfo.Name);

}

}

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error cleaning up old archives");

}

}

}

}

Let me add a utility class for command-line usage of the migration tool:

using System;

using System.CommandLine;

using System.CommandLine.Invocation;

using System.Threading.Tasks;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.Logging;

namespace SqlToSqlDataMigration

{

public class CommandLineInterface

{

public static async Task<int> Main(string[] args)

{

var rootCommand = new RootCommand("SQL to SQL Data Migration Tool")

{

new Option<string>(

new[] { "--config", "-c" },

description: "Path to the configuration file",

getDefaultValue: () => "appsettings.json"),

new Option<string>(

new[] { "--source-connection", "-s" },

"Source connection string (overrides config)"),

new Option<string>(

new[] { "--target-connection", "-t" },

"Target connection string (overrides config)"),

new Option<bool>(

new[] { "--validate", "-v" },

"Validate data after migration",

getDefaultValue: () => false),

new Option<bool>(

new[] { "--dry-run", "-d" },

"Perform a dry run without modifying target database",

getDefaultValue: () => false),

new Option<string>(

new[] { "--table", "-tb" },

"Migrate only specific table(s) (comma-separated)"),

new Option<bool>(

new[] { "--reset-state", "-r" },

"Reset migration state",

getDefaultValue: () => false),

new Option<LogLevel>(

new[] { "--log-level", "-l" },

"Log level (Trace, Debug, Information, Warning, Error, Critical)",

getDefaultValue: () => LogLevel.Information)

};

rootCommand.Handler = CommandHandler.Create<string, string, string, bool, bool, string, bool, LogLevel>(

async (config, sourceConnection, targetConnection, validate, dryRun, table, resetState, logLevel) =>

{

// Load configuration

var configuration = new ConfigurationBuilder()

.AddJsonFile(config, optional: true)

.AddEnvironmentVariables()

.Build();

// Override connection strings if provided

if (!string.IsNullOrEmpty(sourceConnection))

{

configuration["ConnectionStrings:Source"] = sourceConnection;

}

if (!string.IsNullOrEmpty(targetConnection))

{

configuration["ConnectionStrings:Target"] = targetConnection;

}

// Configure logging

using var loggerFactory = LoggerFactory.Create(builder =>

{

builder

.SetMinimumLevel(logLevel)

.AddConsole()

.AddFile(configuration.GetSection("Logging:FilePath").Value ?? "logs/migration.log");

});

var logger = loggerFactory.CreateLogger<CommandLineInterface>();

try

{

// Reset migration state if requested

if (resetState)

{

var stateStorePath = configuration.GetValue<string>("Settings:StateStorePath", "migrationState.json");

if (System.IO.File.Exists(stateStorePath))

{

// Archive current state

var stateCleanupService = new StateCleanupService(configuration, logger);

stateCleanupService.ArchiveState();

// Delete current state

System.IO.File.Delete(stateStorePath);

logger.LogInformation("Migration state has been reset");

}

}

// Configure migration service

var migrationService = new DataMigrationService(configuration, logger);

// Apply table filter if provided

if (!string.IsNullOrEmpty(table))

{

migrationService.SetTableFilter(table.Split(',', StringSplitOptions.RemoveEmptyEntries));

}

// Set dry run mode if requested

if (dryRun)

{

migrationService.EnableDryRun();

logger.LogInformation("Running in dry-run mode - no data will be modified");

}

// Run migration

await migrationService.RunMigrationAsync();

// Validate data if requested

if (validate)

{

logger.LogInformation("Running data validation checks...");

var validationService = new DataValidationService(configuration, logger);

await validationService.ValidateAsync(migrationService.GetProcessedTables());

}

logger.LogInformation("Migration completed successfully");

return 0;

}

catch (Exception ex)

{

logger.LogCritical(ex, "Migration failed");

return 1;

}

});

return await rootCommand.InvokeAsync(args);

}

}

}

Finally, let's add a README.md file to explain how to use this tool:

# **SQL to SQL Incremental Data Migration Tool**

A robust, comprehensive, and customizable C# application for incremental data migration between SQL Server databases.

## **Features**

* **Configuration-based mapping** - Define table and column mappings in JSON configuration
* **Incremental loading** - Track timestamps or IDs to only migrate new or changed data
* **Performance optimization** - Batch processing and SqlBulkCopy for efficient data transfer
* **Error handling and logging** - Comprehensive logging and retry policies
* **Data validation** - Compare source and target data to ensure integrity
* **Scheduling** - Run migrations at specified intervals
* **Transaction management** - Ensure data consistency
* **Monitoring** - Track performance metrics for each migration
* **Dry run mode** - Test migrations without modifying the target database

## **Getting Started**

### **Prerequisites**

* .NET 6.0 or higher
* SQL Server (source and target)
* Access rights to both databases

### **Installation**

1. Clone the repository

Build the solution:  
 dotnet build

1. Create a configuration file (see Configuration section below)

### **Usage**

#### **As a scheduled service**

dotnet run

This will start the migration service with the default configuration from appsettings.json.

#### **Command-line interface**

dotnet run -- --config=myconfig.json --validate

Command-line options:

* --config, -c: Path to the configuration file (default: appsettings.json)
* --source-connection, -s: Source connection string (overrides config)
* --target-connection, -t: Target connection string (overrides config)
* --validate, -v: Validate data after migration
* --dry-run, -d: Perform a dry run without modifying target database
* --table, -tb: Migrate only specific table(s) (comma-separated)
* --reset-state, -r: Reset migration state
* --log-level, -l: Set log level (Trace, Debug, Information, Warning, Error, Critical)

## **Configuration**

The application uses a JSON configuration file with the following structure:

{

"ConnectionStrings": {

"Source": "Server=source-server;Database=SourceDB;User Id=username;Password=password;",

"Target": "Server=target-server;Database=TargetDB;User Id=username;Password=password;"

},

"Settings": {

"BatchSize": 5000,

"CommandTimeout": 600,

"EnableTransaction": true,

"StateStorePath": "migrationState.json",

"ScheduleIntervalMinutes": 10,

"StateRetentionDays": 30,

"LogLevel": "Information"

},

"TableMappings": [

{

"SourceSchema": "common",

"SourceTable": "tbl\_Daily\_actions\_players",

"TargetSchema": "common",

"TargetTable": "tbl\_Daily\_actions\_players",

"Enabled": true,

"FailOnError": true,

"IncrementalType": "DateTime",

"IncrementalColumn": "LastUpdated",

"IncrementalStartValue": "2024-01-01 00:00:00",

"CustomWhere": "IsTest = 0",

"OrderBy": "LastUpdated, PlayerID",

"ColumnMappings": [

{

"SourceColumn": "PlayerID",

"TargetColumn": "PlayerID",

"DataType": "bigint",

"AllowNull": false

},

// Additional column mappings...

],

"BulkCopyOptions": {

"KeepIdentity": true,

"KeepNulls": true,

"TableLock": true,

"Timeout": 600

}

},

// Additional table mappings...

]

}

### **Table Mapping Configuration**

Each table mapping can have the following properties:

* SourceSchema: Source database schema (default: "dbo")
* SourceTable: Source table name
* TargetSchema: Target database schema (default: "dbo")
* TargetTable: Target table name
* Enabled: Whether this table mapping is enabled (default: true)
* FailOnError: Whether to fail the entire migration if an error occurs with this table (default: true)
* IncrementalType: Type of incremental loading ("None", "DateTime", "Int", "BigInt")
* IncrementalColumn: Column to track for incremental loads
* IncrementalCompareOperator: Operator for incremental comparison (default: ">")
* IncrementalStartValue: Initial value for incremental tracking
* CustomWhere: Additional WHERE clause for the query
* OrderBy: Custom ORDER BY clause
* TopN: Limit the number of rows to migrate (0 for no limit)
* ColumnMappings: Array of column mappings
* BulkCopyOptions: Options for SqlBulkCopy

### **Column Mapping Configuration**

Each column mapping can have the following properties:

* SourceColumn: Column name in the source table
* TargetColumn: Column name in the target table
* DataType: Data type of the column
* AllowNull: Whether the column allows NULL values (default: true)
* DefaultValue: Default value if NULL and AllowNull is false
* Transformation: Transformation to apply to the data (e.g., "ToUpper", "Trim")
* TransformationFormat: Format string for the transformation (e.g., date format)

## **State Management**

The application maintains state between migrations by storing the last successful incremental value for each table. This state is stored in a JSON file (configurable via Settings:StateStorePath).

## **Logging**

Logs are written to both the console and a file. The log level and file path are configurable:

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning"

},

"FilePath": "logs/migration.log"

}

## **Error Handling and Retries**

The application includes built-in retry policies for transient SQL errors:

* Regular retry policy: Handles most SQL exceptions with exponential backoff
* Deadlock retry policy: Specifically handles deadlock errors with linear backoff

## **Data Validation**

When validation is enabled, the application performs the following checks:

* Row count comparison between source and target
* Column schema validation
* Sample data comparison

## **Performance Tuning**

For optimal performance, consider adjusting the following settings:

* BatchSize: Number of rows to process in each batch
* CommandTimeout: SQL command timeout in seconds
* BulkCopyOptions: Various options for SqlBulkCopy

## **License**

This project is licensed under the MIT License - see the LICENSE file for details.