In this lab class we will approach the following topics related to performance monitoring:

1. **Performance monitoring and performance indicators**
2. **Isolating slow running queries using the SQL Profiler**
3. **Saving and analyzing the trace as a database table**
4. **Generating a trace for tuning**
5. **Reorganizing and rebuilding indexes**
6. **Using of the query governor configuration option**
7. **Check files auto growth frequency**

1. Performance monitoring and indicators

Performance monitoring should check that the performance-influencing database parameters are correctly set and if they are not, it should point to where the problems are. Monitoring can be made by extracting and analyzing the relevant performance indicators (counters, gauges and details of the internal DBMS activities).

One way to measure performance indicators in SQL Server is to use the **Performance Monitor** available on Windows. Open the application and click **Performance > Monitoring Tools > Performance Monitor**. The tool will be monitoring your system using its default counters (e.g. **%Processor Time**). We can add more counters by right clicking on any area on the right side and selecting **Add Counters**. Under **Available Counters**, you will find several counters related to SQL Server, such as:

- **SQLServer:Access Methods**: Full scans and index searches per second
- **SQLServer:Buffer Manager**: Page writes and reads per second
- **SQLServer:Databases**: Transactions per second
- **SQLServer:Locks**: Number of deadlocks and timeouts per second
- **SQLServer:Transactions**: Longest running transaction time
- **SQLServer:Wait Statistics**: Lock waits
- etc.

Alternatively, commands such as **SET STATISTICS TIME ON**, **SET STATISTICS IO ON**, **DBCC showcontig**, stored procedures such as **master.dbo.sp_lock**, or system views such as **master.dbo.sysperfinfo** can provide detailed results, or results on a per-query basis.

- **Profiling query executions**: **SET STATISTICS TIME ON** displays the number of milliseconds required to parse, compile, and execute each statement. **SET STATISTICS IO ON** causes SQL Server to display information regarding the disk activity generated by T-SQL statements.

- **Disk subsystem**: **DBCC showcontig** shows information related to disk fragmentation. The **scan density** and **avg page density** values are the most important information to look at. Values of 90% and above are OK. The higher the numbers, the lower the amount of fragmentation.

- **Lock monitor**: the **sp_lock** stored procedure uses information from the **syslockinfo** view to display information on all granted, converting, and waiting lock requests. Executing it returns the fields *spid*, *dbid*, *objid*, *indid*, *type*, *resource*, *mode*, and *status*. The *spid* is the process identification number, which identifies your connection to SQL Server. To find out which user is associated with that *spid*, execute the stored procedure **sp_who** and pass the *spid* as a parameter to the procedure. The *dbid* is the identifier of the database the lock is occurring in; you can find it in the **sysdatabases** view in the master database. The *objid* field indicates what object is being locked. To view this object, you can query the **sysobjects** view in the master database for that specific *objid*. The *type* field is the type of lock (e.g. TABLE, PAGE or ROW), mode indicates the lock requester's lock mode, and *status* indicates the lock request status.

2. Isolating a slow running query using SQL Profiler

Queries that take a long time to run are obvious candidates for optimization. In this exercise, we will use the **SQL Server Profiler** to isolate a poorly performing query within a query batch. We will start by defining a trace to capture query performance information:

a) Start **SQL Server Profiler** from Microsoft SQL Server Tools.

b) Click the **New Trace** button and connect to the Database Engine.

c) In the **General** tab:
   - In **Trace name** write **Trace1**
   - In the **Use the template** drop-down, choose **Standard (Default)**
   - Check the **Save to File** box and save the file as: **C:\Temp\Trace1.trc**

d) Change to the **Events Selection** tab. We are going to change the type of events that SQL Profiler will capture:
   - In the **Security Audit** category, uncheck **Audit Login** and **Audit Logout**.
   - In the **Stored Procedures** category, uncheck **RPC:Completed**.
   - Click the **Show all events** checkbox.
   - Expand the **Performance** category, and select **Showplan All** and **Showplan XML**.
   - Uncheck **Show all events** to see only the events that have been selected for this trace.

e) Still in the **Events Selection** tab:
   - Click the **Show all columns** checkbox.
   - Right-click the **DatabaseName** column heading and select **Edit Column Filter**.
   - Expand the option **Like** and type in **AdventureWorks2017**. Click **OK**. This will filter out all activity except for the selected events in the AdventureWorks2017 database.

f) Click **Run**. The trace begins.

Now, to isolate a poorly performing query in a batch:

g) Start **SQL Server Management Studio** and connect to the Database Engine.

h) From the menu bar, select **File > Open > File** and open the **lab9.sql** file from the previous lab.

i) Click **Execute** to execute the entire script.

j) Once the script completes, go back to **SQL Profiler** and stop the trace by clicking the **Stop Selected Trace** button in the toolbar.

k) Scroll to the right to find the **Duration** column. Only some events have a duration.

l) Find the event with the greatest value in the **Duration** column.

m) Select that row, and you should see the query in the lower pane.

n) Click the preceding row (**Showplan XML**) and you should see the execution plan.

3. Saving and analyzing the trace as a database table

a) In **SQL Server Profiler**, click the **Properties** button in the toolbar.

b) Select the **Save to table** checkbox, and connect to the Database Engine.

c) In the **Destination Table** dialog:
- Accept the default database (**master**)
- Accept the default schema (**dbo**)
- Accept the default table (**Trace1**)
- Click **OK**

d) Click **Run** to begin the trace.

e) Switch to **SQL Server Management Studio**, and again **Execute** the entire script lab09.sql.

f) Once the script completes, stop the trace in **SQL Profiler**.

g) In **SQL Server Management Studio**, expand **Databases > System Databases > master > Tables**.

h) Locate the **dbo.Trace1** table. If needed, right-click **Tables** and select **Refresh**.

i) Right-click the **dbo.Trace1** table, and **Select Top 1000 Rows** to see the table contents.

j) Right-click the **master** database, and select **New Query**.

k) Execute the following query to sort the trace records according to duration:

```
SELECT *
FROM dbo.Trace1
WHERE EventClass = 12
ORDER BY Duration DESC;
```

l) Compare the value of **Duration** shown in **SQL Server Profiler** to the value of **Duration** in the **dbo.Trace1** table. What do you conclude? [1]

4. Generating a trace for tuning

a) In **SQL Server Profiler**, Click the **New Trace** button and connect to the Database Engine.

b) In the **General** tab:
- In **Trace name** write **Trace2**
- In the **Use the template** drop-down, choose **Tuning**
- Select the **Save to table** checkbox, and connect to the Database Engine.

c) In the **Destination Table** dialog:
- Accept the default database (**master**)
- Accept the default schema (**dbo**)
- Accept the default table (**Trace2**)
- Click **OK**

o) In the **Events Selection** tab:
- In the **Stored Procedures** category, uncheck **RPC:Completed** and **SP:StmtCompleted**.
- (Leave only the **TSQL** category with **SQL:BatchCompleted** checked.)
- Click the **Show all columns** checkbox.
- Right-click the **DatabaseName** column heading and select **Edit Column Filter**.
- Expand the option **Like** and type in **AdventureWorks2017**. Click **OK**.

d) Click **Run** to begin the trace.

e) Switch to **SQL Server Management Studio**, and **Execute** the script lab09.sql.

f) Once the script completes, stop the trace in **SQL Profiler**.

g) Open the **Database Engine Tuning Advisor** and connect to the Database Engine.

[1] In SQL Server Profiler, in the menu Tools > Options, there is an option for showing Duration in microseconds.

h) In **Workload**, select **Table** and click the browse button on the right side of the text box.

i) In the **Select Workload Table** dialog, choose the **master** database, the **dbo** schema, and the **Trace2** table. Press **OK**.

j) In **Database for workload analysis**, select **AdventureWorks2017**.

k) Also, check the **AdventureWorks2017** database under **Select databases and tables to tune**.

l) Click **Start Analysis** on the toolbar.

m) Once the analysis is complete, switch to the **Reports** tab and select the **Statement cost report**.

n) Locate the query that you have previously identified as having the longest duration, and check the **Performance Improvement** that can be expected with the DTA recommendations.

o) Switch to the report **Statement-index relations report (recommended)**. Locate the same query and check if DTA is recommending new indexes for that query (with prefix **_dta_index**).

5. Reorganizing and rebuilding indexes

Regarding indexes, SQL Server automatically maintains indexes whenever insert, update, or delete operations are made to the underlying data. Over time, these modifications can cause the information in the index to become scattered in the database (i.e., **fragmented**).

The first step in deciding which defragmentation method to use is to analyze the index to determine the degree of fragmentation. By using the system function **sys.dm_db_index_physical_stats**, you can detect fragmentation in a specific index, all indexes on a table or indexed view, all indexes in a database, or all indexes in all databases.

For this purpose, try the following query in SQL Server Management Studio:

```
USE AdventureWorks2017;
SELECT indexstats.index_id,
       dbschemas.name as 'Schema',
       dbtables.name as 'Table',i.name as 'index',
       avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats(DB_ID(N'AdventureWorks2017'), NULL, NULL,
NULL, NULL) AS indexstats
      JOIN sys.indexes AS i ON indexstats.object_id = i.object_id AND
      indexstats.index_id = i.index_id
      JOIN sys.tables dbtables on dbtables.object_id = indexstats.object_id
      JOIN sys.schemas dbschemas on dbtables.schema_id = dbschemas.schema_id
ORDER BY indexstats.avg_fragmentation_in_percent DESC;
```

After the degree of fragmentation is known, the following table shows the best method to correct the fragmentation: [2]

avg_fragmentation_in_percent value	Corrective statement
> 5% and < = 30%	ALTER INDEX REORGANIZE
> 30%	ALTER INDEX REBUILD WITH (ONLINE = ON)

6. Using of the query governor configuration option

Regarding duration, the **query governor configuration** option can prevent long-running queries from executing, thus preventing system resources from being consumed. By default, the query governor configuration option allows all queries to execute, no matter how long they take. However, the query governor can be set to the **maximum number of seconds that all queries for all connections, or just the queries for a specific connection, are allowed to execute**. Because the query governor is based on estimated query cost, rather than actual elapsed time, it does not have any run-time overhead. It also **stops long-running queries before they start**, rather than running them until some predefined limit is hit.

The instruction below illustrates the use of the query governor option on a per connection basis. If you use **sp_configure** stored procedure to change the value of query governor cost limit, the changed value is server-wide. The value refers to the estimated elapsed time, in seconds, required to execute a query.

SET QUERY_GOVERNOR_COST_LIMIT value

If you specify a nonzero, nonnegative value, the query governor disallows execution of any query that has an estimated cost exceeding that value. Specifying 0 (the default) for this option turns off the query governor. In this case, all queries are allowed to run.

7. Check files auto growth frequency

Finally, one type of operation that can decrease the performance of databases is the automatic growth of files (remember the **FILEGROWTH** parameter from Lab 1). A good practice to follow is to check regularly if the current configurations for auto growth are appropriate. One way to do this is to check if the database files are growing too often.

In **SQL Server Profiler**, in the **Events Selection**, under the **Database** category, you can find the events **Data File Auto Grow** and **Log File Auto Grow** that can be used to monitor file growth.

[2] https://docs.microsoft.com/en-us/sql/relational-databases/indexes/reorganize-and-rebuild-indexes