In the lab classes, we will be using **Microsoft SQL Server** to see how the theoretical concepts apply in a real-world database system. Students can refer to the following books for more information on managing and optimizing SQL Server databases:

1. K. Delaney et al., ***Microsoft SQL Server 2012 Internals***, Microsoft Press, 2013
2. G. Fritchey, ***SQL Server Query Performance Tuning***, Apress, 2014
3. K. England, G. Powell, ***Microsoft SQL Server 2005 Performance Optimization and Tuning***, Elsevier, 2007

Most of the materials presented in the labs follow directly from these books.

In this first lab class, we will address the following topics:

1.1 Microsoft SQL Server Databases and Database Files

In a database, **the logical schema** defines the way the data is organized into a set of **tables, attributes, and relationships between tables**. However, when the database is stored as a set of files on disk, its organization may be different. **The physical schema** is the **underlying file structure** of the database at the level of the operating system. If you have studied databases before, you may be familiar with logical schema design. In this course, we will start by looking at physical schema issues.

For an application developer, it is probably enough to consider an SQL Server instance as a collection of databases containing tables, indexes, triggers, stored procedures, views, and user data. For a database developer and administrator, it is useful to **look deeper at the physical storage structures used in SQL Server**. Understanding the physical organization of the database is crucial for optimizing performance.

An SQL Server instance will typically host many databases. The term "SQL Server" often refers to a computer containing an installation of the SQL Server software. An SQL Server database is a single database, created within a SQL Server installation. Usually, individual databases are backed up, restored, and checked for integrity. A **database can therefore be thought of as a unit of administration**.

A **database resides in operating system files**, called **database files**, stored on disk partitions managed by the operating system. The database files may be used to hold user and system tables (i.e., **data files**), or to track changes made to these tables (i.e., **transaction log files**). A database can contain a large number of files with size in the multiple terabyte (TB) range. A single data file, by itself, can have a size of many TB.

Each **file cannot be shared by more than one database**, even if those databases share the same SQL Server installation. Database clustering is the exception to this rule, allowing for multiple CPUs to share the same underlying data files. Besides that, **a file cannot be used to hold both data and transaction log** information. This means that a **database must consist of a minimum of two files**. In fact, there are three file types associated with an SQL Server database:

1. The **primary data file** is the starting point of the database and contains the **pointers to the other files in the database**. All databases have a single primary data file. The recommended extension for a primary data file is *.mdf*.

2. **Secondary data files** hold **data that does not fit into the primary data file**. Some databases do not have secondary data files, while others may have multiple secondary data files. The recommended extension for secondary data files is *.ndf*.

3. **Log files** hold all of the **log information used to recover the database**. There is at least one log file per database. The recommended extension for log files is *.ldf*.

The **primary data file** will hold the **system tables** and may hold **user tables**. For most users, placing all their database tables in this file, and placing the file on a suitable RAID configuration, will be sufficient. For some users, their user tables may be too large to be placed in a single file (e.g., the file is too large to be hold in a single individual storage device, i.e. in a single hard drive). In this case, multiple data files — one primary and multiple secondary files — may be used. As user tables are created and populated, SQL Server allocates space from each file to each table, such that tables are effectively spread across the available files and physical storage devices.

For very large databases with many data files, it is possible to use **filegroups**. The purpose of a filegroup is to **manage multiple data files together as a single collection**, which can hold certain tables, indexes, or text/image data that the database administrator decides to place there. This gives the database administrator considerable **flexibility and control over the placement of the database objects**. For example, if two database tables are very heavily accessed, they can be separated into two filegroups. Those two filegroups will consist of two sets of data files, residing on two sets of different physical storage devices.

Some rules govern the use of filegroups. **Transaction logs are never part of filegroups** — only data files are. Also, **each data file can only be part of one single filegroup**.

The **configuration of a database** is reflected in various **system tables/views** held in the master database (i.e., the main database used by SQL Server to hold information regarding the database management system) and the user database (i.e., in tables existing within the users's own database). Specifically, the **master** database contains a system view (**sys.databases**) that has one row for every database resident in the SQL Server instance. Also, **each database contains its own system tables/views**. For example, the view **sys.database_files** has one row representing each file (data or log) for the database, and the view **sys.filegroups** contains one row for every filegroup in the database.

The easiest way to create a database in **SQL Server** is to use the visual interface **SQL Server Management Studio**, which can also be used to generate a script for creating a database at a later time. These scripts can be manually changed, e.g. for tuning.

An example of a command for creating a database using the T-SQL CREATE DATABASE statement is:[1]

```
CREATE DATABASE BankingExampleDB ON
PRIMARY (
  NAME = BankingData,
  FILENAME = 'd:\data\BankingData.mdf',
  SIZE = 3MB, MAXSIZE = 100MB, FILEGROWTH = 1MB )
LOG ON (
  NAME = 'BankingLog',
  FILENAME = 'e:\data\BankingLog.ldf',
  SIZE = 1MB, MAXSIZE = 100MB, FILEGROWTH = 10% )
```

Let's look in detail into some of the options that are presented in the T-SQL command shown in the example above:

- The **ON** keyword introduces a list of **one or more data file definitions**, while the **LOG ON** keyword introduces a list containing **one or more transaction log file definitions**.

- The **NAME** option specifies a **logical name for the file** and the **FILENAME** option specifies its **physical storage location**.

- The **PRIMARY** keyword identifies the list of files following it as the **files that belong to the primary filegroup**. The first file defined in the primary filegroup becomes the primary file, which is the file containing the database system tables. The PRIMARY keyword can be omitted, in which case the first file specified in the CREATE DATABASE statement is the primary file. In addition to this keyword, the **FILEGROUP keyword** can be used to specify secondary filegroups.

[1] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-database-sql-server-transact-sql

- The **SIZE** keyword specifies the **initial size of the file**. In T-SQL, the units are, by default, megabytes, although this can be specified explicitly by using the suffix MB. If desired, the file size can be specified in Kilobytes (KB), Gigabytes (GB) or Terabytes (TB).

- When a data file or transaction log file is full, it can automatically grow. In T-SQL, by default the file will be allowed to grow, unless the **FILEGROWTH** keyword is set to 0. When a file grows, the **size of the growth increment** is controlled by the **FILEGROWTH** parameter. This growth increment can be specified as a fixed value, such as 10MB, or as a percentage of the size of the file at the time the increment takes place. If not specified, the default FILEGROWTH value depends on the SQL Server version; for SQL Server 2017, it is 64MB.

- The file may be allowed to grow until it takes up all the available space in the physical storage device on which it resides, at which point an error will be returned when it tries to grow again. Alternatively, a limit can be set using the **MAXSIZE** keyword. Size can be specified as a numeric value or as the keyword **UNLIMITED** — this is the default.

Every time a file extends (i.e. grows), the applications using the database during the file extension operation may experience performance degradation. Also, extending a file multiple times may result in fragmented disk space, particularly when file growth additions are not all of the same size. It is advisable, therefore, to try to **create the file with an initial size estimated to be close to the size that will be required by the file**.

Various properties of a database can also be modified after it has been created. Students should refer to the documentation on the ALTER DATABASE T-SQL command for more information.[2]

Dropping a database can be performed as follows:

```
USE master;
ALTER DATABASE TestDB SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
DROP DATABASE TestDB;
```

In the previous set of instructions, the first instruction changes to a database other than the one that is going to be dropped, the ALTER DATABASE ensures that no other user is accessing the database, by stopping any existing connections and rolling back their transactions. Finally, the third instruction drops the database.

1.2 - Inside SQL Server Database Files

A **database file** is structured as a collection of **pages**, each with **8 KB in size**. Pages are always this size and cannot be adjusted. The 8 KB page is the fundamental unit of storage and it is also a **unit of I/O and locking** (there are other units of I/O and locking). Each page has a fixed 96-byte **page header**, which contains information such as the page number, pointers to the previous and next page, or the ID of the object to which the page belongs. After the header come the **data rows**, each containing columns of data (columns can have a fixed or a variable length).

[2] https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-database-transact-sql

Tables and indexes typically occupy multiple pages. The way that **pages are allocated to tables and indexes is through extents**. An **extent** is a structure that contains **eight pages (64 KB)**. Extents are of two types — **uniform** and **mixed**. A **uniform extent** devotes its **eight pages to one object**, e.g. a particular table in the database. A **mixed extent** allows its pages to be used by **up to eight different objects**.

The reason SQL Server uses **mixed extents** is to ensure that a whole 8 page (64 KB) extent is not used for a small table (i.e. to **save space**). Instead, single pages are allocated to the table one at a time, as the number of rows grows. When eight pages have been allocated and more are needed, uniform extents are used to allocate eight pages more.

Pages that are used to **hold table rows and index entries** are known as **data pages** and **index pages**, respectively. If the table contains columns of the data type TEXT or IMAGE, then these columns are usually implemented as structures of **Text/Image pages** (unless the TEXT/IMAGE data is very small and stored in the row). There are other types of pages, in both primary or secondary database files, used to manage space in a file:

- Global Allocation Map (GAM) pages;
- Shared Global Allocation Map (SGAM) pages;
- Index Allocation Map (IAM) pages;
- Page Free Space (PFS) pages;

GAM pages hold information concerning **which extents are currently allocated** — that is, are not free. A single GAM page can manage 64,000 extents, which equates to nearly 4 GB of space. If more than 64,000 extents are present in the file, additional GAM pages are used. A GAM page uses a single bit to represent each extent out of the 64,000 range. If the bit is set (1), the extent is free; if it is not set (0), it is allocated.

SGAM pages hold information on **which extents are currently being used as mixed extents** and have one or more unused pages — that is, have space that can still be allocated to objects. A single SGAM page can manage 64,000 extents. If more than 64,000 extents are present in the file, additional SGAM pages are used. An SGAM page uses a single bit to represent each extent. If the bit is set (1), the extent is a mixed extent and has one or more unused pages; if it is not set (0), the mixed extent is completely allocated.

IAM pages hold information concerning the **allocation of GAM pages**, tracking what space within a specific GAM page belongs to a single object. Each **table or index has at least one IAM page** and if the object is spread across more than one file it will have an IAM page for each. A single IAM page can manage 512,000 pages and the IAM pages for a file or index are chained together. The first IAM page in the chain holds eight slots, which can contain pointers to the eight pages that may be allocated from mixed extents. All IAM pages also contain a bitmap with each bit presenting an extent in the range of extents held by the IAM. If the bit is set (1), the extent represented by that bit is allocated to the table or index; if it is not set (0), the extent is not allocated to the table or index.

PFS pages hold information concerning **which other pages have free space**. Each PFS page keeps track of the other pages in the database file. A PFS page uses a single byte to represent each page. This byte keeps track of whether the page is allocated, whether it is empty, and, if it is not empty, how full the page is. A single PFS page can keep track of 8,000 contiguous pages. Additional PFS pages are created as needed.

In a database file, the first page (0) contains a file header. The second page (1) is the first PFS page (the next will only be found after another 8,000 pages). The third page (2) is the first GAM, and the fourth page (3) is the first SGAM. IAM pages are located in arbitrary positions throughout the file, and the same for data and index pages.

1.3 - Partitioning Logical Objects into Physical Parts

Partitioning involves the **physical splitting of large objects**, e.g. tables and indexes, into separate physical parts (i.e., partitions). Partitioning results in two primary benefits:

- Operations can be performed on individual physical partitions, thus reducing I/O requirements. Queries can be executed on a single partition, and single partitions can be added or removed. The rest of the table (all other partitions) remains unaffected.

- Operations on the multiple partitions can be executed in parallel, for faster processing.

Both factors above make partitioning a tuning aspect. Partition tuning is essentially related to the underlying structures, indexing techniques, and the way in which partitions are constructed. SQL Server allows for table partitioning and index partitioning, i.e. users can create a table as a partitioned table, defining specifically where each physical chunk of the table or index resides. For more information, students should refer to the documentation for the T-SQL CREATE TABLE[3] and CREATE INDEX[4] commands.

SQL Server has two important commands for defining database partitions:

- **Command for defining the range partition function**: Data will be partitioned based on the values of a table column, putting a variable number of rows in each partition. The allocation of a table row to a partition is based on the value of the partitioning column. An example of this column is the incremental identity column, which can be partitioned in different ranges.[5]

```
CREATE PARTITION FUNCTION partition_function_name ( input_parameter_type )
AS RANGE [ LEFT | RIGHT ]
FOR VALUES ( [ boundary_value [ ,...n ] ] )
```

[3] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql
[4] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql
[5] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-partition-function-transact-sql

- **Placing each range into a physical disk space area**: This is done by using a partitioning function, in tandem with filegroups, through the CREATE PARTITION SCHEME statement.[6]

```
CREATE PARTITION SCHEME partition_scheme_name
AS PARTITION partition_function_name
[ ALL ] TO ( { file_group_name | [ PRIMARY ] } [ ,...n ] )
```

Different ranges can be on different partitions, different partitions can be on different filegroups, and different filegroups can be on different disks to improve performance.

The following set of instructions exemplifies the creation of a database with two filegroups, and a partitioned table inside that database:

```
CREATE DATABASE TestDB ON
PRIMARY (
     NAME = 'TestDB_Part1',
     FILENAME = 'C:\Data\Primary\TestDB_Part1.mdf',
     SIZE = 2, MAXSIZE=100 ),
FILEGROUP TestDB_Part2 (
     NAME = 'TestDB_Part2',
     FILENAME = 'C:\Data\Secondary\TestDB_Part2.ndf',
     SIZE = 2, MAXSIZE = 100 )
LOG ON (
     NAME = 'TestDB_Log',
     FILENAME = 'C:\Data\Log\TestDB_Log.ldf',
     SIZE = 1, MAXSIZE = 100 );

USE TestDB;

CREATE PARTITION FUNCTION TestDB_PartitionRange (INT)
AS RANGE LEFT FOR VALUES (10);

CREATE PARTITION SCHEME TestDB_PartitionScheme AS
PARTITION TestDB_PartitionRange TO ([PRIMARY], TestDB_Part2);

CREATE TABLE TestTable (ID INT NOT NULL, Date DATETIME)
ON TestDB_PartitionScheme (ID);
```

[6] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-partition-scheme-transact-sql

2. Exercises

2.1. Creating Databases with SQL Server Management Studio

SQL Server Management Studio is a GUI tool used to manage SQL Server, including multiple databases, tables, indexes, fields, and data types.

We will start by using the visual interface of SQL Server Management Studio to create a sample database:

a) Start up **SQL Server Management Studio** and connect to the default server.
b) Right-click **Databases** in the **Object Explorer**, then click **New Database**.
c) Enter the name of the database: **ExampleDB**
d) Leave everything else set to the default settings.
e) Click OK.

   Various properties can be set for each data and log file, corresponding to the options of the CREATE DATABASE T-SQL statement. Various properties of a database can also be modified after it has been created. These include increasing and reducing the size of data and transaction log files, adding and removing database and transaction log files, creating filegroups, and changing the DEFAULT filegroup.

f) Delete the database by right-clicking on **ExampleDB** and selecting **Delete**.

2.2. Creating Databases with T-SQL

a) Create the C:\Temp directory, which we will use to store the files for a new database that we are about to create.

b) Write a T-SQL command for creating a database with the following characteristics:

   *The database should be called **ExampleDB**, and it should contain one log file and three different data files, in three distinct filegroups (i.e., one data file in the primary filegroup, one in a secondary filegroup, and another in a second secondary filegroup).*

   The log file should have an initial size of 5MB and a maximum size of 100MB. The data files should have an unlimited maximum size. The data file on the first secondary filegroup should have an initial size of 20MB, and the remaining files should have an initial size of 30MB.

   All files should grow at a rate of 15%, except for the data file in the first secondary filegroup, which should grow by 2048KB, every time this is required.

(solution on next page)

```
CREATE DATABASE ExampleDB ON
PRIMARY (
       NAME = ExampleDB_File1,
       FILENAME= 'C:\Temp\ExampleDB_File1.mdf',
       SIZE = 30MB,
       FILEGROWTH = 15%),
FILEGROUP SECONDARY_1 (
       NAME = ExampleDB_File2,
       FILENAME= 'C:\Temp\ExampleDB_File2.ndf',
       SIZE = 20MB,
       FILEGROWTH = 2048KB),
FILEGROUP SECONDARY_2 (
       NAME = ExampleDB_File3,
       FILENAME= 'C:\Temp\ExampleDB_File3.ndf',
       SIZE = 30MB,
       FILEGROWTH = 15%)
LOG ON (
       NAME = ExampleDB_Log,
       FILENAME = 'C:\Temp\ExampleDB_Log.ldf',
       SIZE = 5MB,
       MAXSIZE = 100MB,
       FILEGROWTH = 15%);
```

c) Execute the CREATE DATABASE statement above.

d) In **Object Explorer**, right-click **Databases** and select **Refresh**. Check that the ExampleDB database has been created.

e) Check that the corresponding files have been created in C:\Temp. Also, check that the initial file sizes agree with the specification.

f) In **Object Explorer**, right-click **ExampleDB** and select properties. In **Files**, check that the file properties agree with the specification.

g) Write a T-SQL command for creating a table in the ExampleDB database, considering the following characteristics:

*The table should be called **ExampleTable**, and it should have a numeric attribute named ID and a numeric attribute named VALUE1. Together, these two attributes should identify the records of the table. The table should also have a numeric attribute named VALUE2, and an alphanumeric attribute named STR1.*

The table should be partitioned so that all tuples where VALUE1 is less than 10 are physically stored in a different filegroup from those tuples where VALUE1 is greater or equal to 10.

(solution on next page)

```
USE ExampleDB;

CREATE PARTITION FUNCTION ExampleDB_Range1(INT)
AS RANGE RIGHT FOR VALUES (10);

CREATE PARTITION SCHEME ExampleDB_PartScheme1
AS PARTITION ExampleDB_Range1 TO
(SECONDARY_1, SECONDARY_2);

CREATE TABLE ExampleTable (
       ID INT NOT NULL,
       VALUE1 INT NOT NULL,
       VALUE2 INT NOT NULL,
       STR1 VARCHAR(50),
       PRIMARY KEY(ID,VALUE1)
) ON ExampleDB_PartScheme1(VALUE1);
```

h) Execute the SQL statements above.

i) In **Object Explorer**, expand **ExampleDB** and then **Tables**. Check that the table has been created.

j) Consider the following INSERT statements. Which records will end up in which data files?

```
USE ExampleDB;
INSERT INTO ExampleTable VALUES (4,  8, 40, 'C');
INSERT INTO ExampleTable VALUES (5,  8, 20, 'A');
INSERT INTO ExampleTable VALUES (5,  9, 30, 'B');
INSERT INTO ExampleTable VALUES (6,  9, 40, 'C');
INSERT INTO ExampleTable VALUES (6, 10, 30, 'B');
INSERT INTO ExampleTable VALUES (7, 10, 40, 'C');
INSERT INTO ExampleTable VALUES (7, 11, 20, 'A');
INSERT INTO ExampleTable VALUES (8, 11, 40, 'C');
INSERT INTO ExampleTable VALUES (8, 12, 20, 'A');
```

k) Open a new query window and execute the following statements to drop the ExampleDB database:

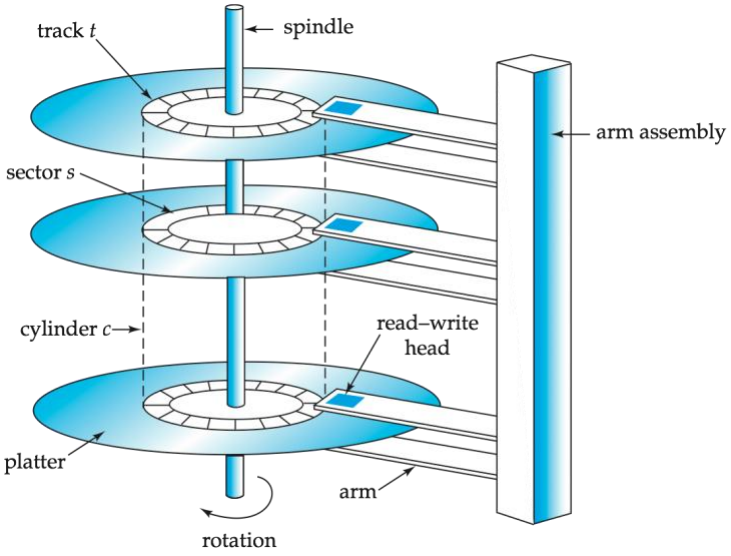
```
USE master;
ALTER DATABASE ExampleDB SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
DROP DATABASE ExampleDB;
```

l) In **Object Explorer**, right-click **Databases** and select **Refresh**. Check that the ExampleDB database has been dropped.

m) Check that the database files have been removed from C:\Temp.

2.3. Accessing Data on Hard Disks

Consider a hard disk with the following specifications:



- The disk has **4 platters**, and **2 surfaces** each platter.
- The disk has **192 cylinders**, with **64 inner cylinders** and **128 outer cylinders** (the density will be different in these two types of cylinders).
- In the disk, **1 block** equals a total of **4 KiB** (i.e., 4 x 2^{10} bytes).
- The disk does one full revolution in **512 µs**.
- The **average seek time** (i.e., the time it takes the head assembly on the actuator arm to travel to the track of the disk where the data is located) is **8000 µs.**
- The total usable **capacity** is **500 GiB** (i.e., 500 x 2^{30} bytes).

The **outer cylinders** have double the density than the **inner ones**, that is, an **outer cylinder** has twice the number of blocks than an **inner cylinder**.

Hint: To compute your answers, you can work with powers of 2, and you can leave your answer as a power of 2, whenever appropriate. For example, if the answer is 500 x 2^{35} divided by 2^5, just leave your answer as 500 x 2^{30}.

a) What is the total number of blocks on the hard disk?

b) How many of the blocks are on the inner cylinders of the disk?

c) How many blocks are on the outer cylinders?

Note: your answers for (b) and (c) should add up to the total in (a).

d) On the inner cylinders of the disk, how many blocks are on each track?

e) Once the head arrives at the beginning of an inner block, how much time does it take to read a block off the disk (ignoring the data transfer rate)?

f) What is the expected time to read a block that resides on an inner cylinder of the disk? Your answer should include the three types of delays that are involved.