Note: This lab assumes that you are using the provided virtual machine, or have otherwise installed SQL Server, SQL Server Management Studio, and the AdventureWorks database.

1. Open **File Explorer** and locate the following folder:
   C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\DATA

2. Inside this folder, locate the files for the **AdventureWorks** database. There should be at least one data file (***.mdf**) and a log file (***.ldf**)
   - Data files contain data and objects such as tables, indexes, stored procedures, and views.
   - Log files contain the information that is required to recover all transactions in the database.

   Take note of the size of these files.

3. Open **SQL Server Management Studio**.

4. In the **Connect to Server** window, connect to the database engine on the local machine with Windows authentication.

5. In **Object Explorer**, expand **Databases** and locate the **AdventureWorks** database.

6. Right-click the **AdventureWorks** database and select **Properties**.

7. In the **Database Properties** window, change to the **Files** page.

8. Take a moment to inspect the following details about the database files:
   - There are two files for different purposes (rows data and log, respectively).
   - The data file is in a filegroup called PRIMARY (more about filegroups later).
   - The size of each file should match what you have previously seen.
   - Each file can grow by a certain amount, up to a certain maximum size.
   - The path and file names should match what you have previously seen.

9. Use **File Explorer** to create the **C:\Temp** folder, if it doesn't already exist.

10. In **SQL Server Management Studio**, open a new query window (**New Query** in the toolbar).

11. Place the following code in the query window:

```
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%);
```

12. Before executing the code, note the following:
    - The database is called **ExampleDB** and it contains three data files and a log file.
    - By convention, the primary (master) data file has extension **.mdf**, other (secondary) data files have extension **.ndf**, and the log file has extension **.ldf**
    - There are three different filegroups: PRIMARY, SECONDARY_1 and SECONDARY_2.
    - The log file has an initial size of 5MB and a maximum size of 100MB.
    - The data files can have an unlimited maximum size. The data file on filegroup SECONDARY_1 has an initial size of 20MB, and the remaining files have an initial size of 30MB.
    - All files can grow at a rate of 15%, except for the data file in the first secondary filegroup, which grows by 2048KB, every time this is required.

13. Execute the CREATE DATABASE statement above.

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

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

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

17. Open another query window and write the following code:

```
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 (
    VALUE1 INT NOT NULL,
    VALUE2 INT NOT NULL,
    STR1 VARCHAR(50)
) ON ExampleDB_PartScheme1(VALUE1);
```

18. Before executing the code, note the following:
    - We are creating a new table called **ExampleTable**.
    - The table has two numeric columns (VALUE1 and VALUE2) and one string column (STR1).
    - The table is partitioned so that all tuples where VALUE1 < 10 are physically stored in a different filegroup from those tuples where VALUE1 >= 10.

19. Execute the statements above.

20. In **Object Explorer**, expand **ExampleDB** and then **Tables**. Check that the **ExampleTable** table has been created.
    *Note: Remember that, when creating a database table, if a schema is not specified, the default schema is **dbo**.*

21. Open another query window and write the following code:

```
USE ExampleDB;

INSERT INTO ExampleTable VALUES (8, 40, 'C');
INSERT INTO ExampleTable VALUES (8, 20, 'A');
INSERT INTO ExampleTable VALUES (9, 30, 'B');
INSERT INTO ExampleTable VALUES (9, 40, 'C');
INSERT INTO ExampleTable VALUES (10, 30, 'B');
INSERT INTO ExampleTable VALUES (10, 40, 'C');
INSERT INTO ExampleTable VALUES (11, 20, 'A');
INSERT INTO ExampleTable VALUES (11, 40, 'C');
INSERT INTO ExampleTable VALUES (12, 20, 'A');
```

22. Before executing the code, answer the following question:
    - Which records will end up in which data files?

23. Execute the statements above.

24. Open a new query window and write the following code:

```
SELECT fg.name, p.rows
FROM sys.partitions AS p,
     sys.destination_data_spaces AS dds,
     sys.filegroups AS fg
WHERE p.object_id = OBJECT_ID('ExampleTable')
  AND p.partition_number = dds.destination_id
  AND dds.data_space_id = fg.data_space_id;
```

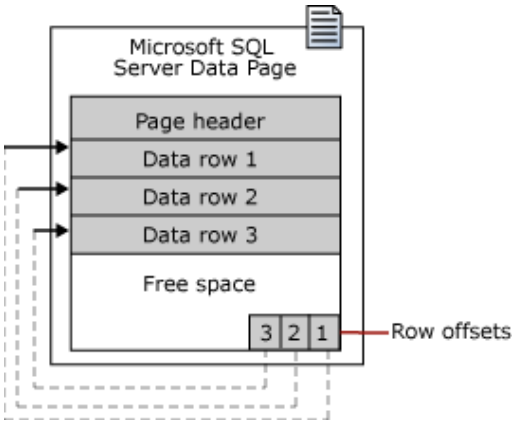
25. Before executing the code, note the following:
    - The system view **sys.partitions** returns a row for each partition of all tables in the database. (In this case, we want the partitions of **ExampleTable** only.)
    - The system view **sys.destination_data_spaces** returns a row for each data space destination of a partition scheme.
    - The system view **sys.filegroups** returns a row for each data space that is a filegroup.

26. Execute the query above.

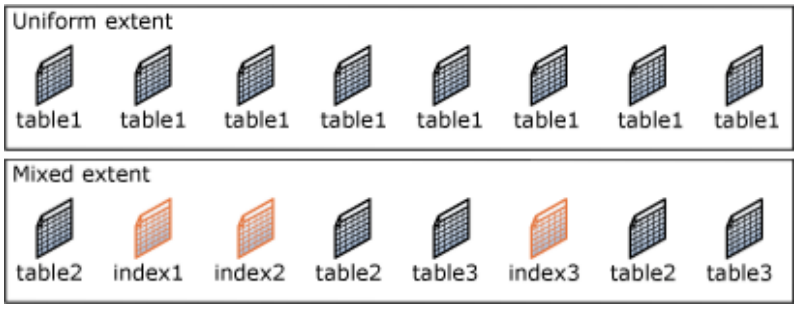
27. Confirm that the results agree with your answer to the question above.

28. We will now investigate the actual contents of a data file in SQL Server.
    For this purpose, it is useful to have these concepts in mind:
    - Inside a data file, the fundamental unit of storage is the **page**. The disk space allocated to a data file (**.mdf** or **.ndf**) in a database is logically divided into pages numbered contiguously from 0 to N. Disk I/O operations are performed at the page level. That is, SQL Server reads or writes whole data pages.
    - In SQL Server, the page size is 8 KB. This means SQL Server databases have 128 pages per megabyte. Each page begins with a 96-byte header that is used to store system information

about the page. This information includes the page number, page type, the amount of free space on the page, and an ID of the object that owns the page.



- Data rows are put on the page serially, starting immediately after the header. A row offset table starts at the end of the page, and each row offset table contains one entry for each row on the page. Each row offset entry records how far the first byte of the row is from the start of the page. Thus, the function of the row offset table is to help SQL Server locate rows on a page very quickly.
- When SQL Server needs to manage space (allocate new pages, or deallocate existing ones), it does so in groups of 8. A group of 8 pages is called an **extent**. An extent is 8 physically contiguous pages, or 64 KB. This means SQL Server databases have 16 extents per megabyte.
- SQL Server has two types of extents: **uniform** and **mixed**. Uniform extents are owned by a single object; all eight pages in the extent can only be used by the owning object. Mixed extents are shared by up to 8 objects; each of the eight pages in the extent can be owned by a different object.



- Log files (**.ldf**) do not contain pages; they contain a series of log records.

29. Open a new query window and write the following code:

```
SELECT partition_id, allocated_page_page_id
FROM sys.dm_db_database_page_allocations(db_id('ExampleDB'),
                                         object_id('ExampleTable'),
                                         NULL, NULL, 'DETAILED')
WHERE page_type_desc = 'DATA_PAGE';
```

30. Before executing the code, note the following:
    - The system function **sys.dm_db_database_page_allocations** provides information about the pages that belong to a particular database object (in this case, **ExampleTable**).
    - The type of pages that we are interest in is data pages (more on this later).

31. Execute the statement above, and take note of the **page IDs**.
    Note: In our case there are two partitions, and the page ID might happen to be the same in each of those partitions.

32. In the same query window, write the following code:

```
SELECT p.partition_number, df.file_id, df.physical_name
FROM sys.partitions AS p,
     sys.destination_data_spaces AS dds,
     sys.database_files AS df
WHERE p.object_id = OBJECT_ID('ExampleTable')
  AND p.partition_number = dds.destination_id
  AND dds.data_space_id = df.data_space_id;
```

33. Before executing the code, note the following:
    - The system view **sys.partitions** returns a row for each partition of all the tables and indexes in the database (in this case, we want the partitions of **ExampleTable** only).
    - The system view **sys.destination_data_spaces** returns a row for each data space destination of each partition scheme.
    - The system view **sys.database_files** indicates the data file that corresponds to each data space.

34. Execute the statement above (only the statement above, by highlighting the code and pressing **Execute**).

35. Take note of the **file IDs** that correspond to each partition.
    Note: In our case, each partition is in a different file, and the file ID identifies each of those physical files.

36. In the same query window, write the following commands:

```
DBCC TRACEON(3604);
DBCC PAGE('ExampleDB', 3, 8, 1);
```

37. Before executing these commands, note the following:
    - DBCC (database console commands) are special SQL Server commands used for database administration, maintenance and troubleshooting.
    - The first command above configures a trace flag to redirect the output of DBCC commands to the results window.
    - The second command allows us to inspect the actual contents of a given data page. The first parameter is the **database**, the second is the **file ID**, the third is the **page ID** and the last is a print option that can be changed from 0 to 3 to provide more detailed information.
    - In this case, our **file ID** is 3 and our **page ID** is 8. You should replace these values with the **file ID** and the **page ID** that you have obtained earlier for partition 1.

38. Execute the commands above (only the commands above, by highlighting the code and pressing **Execute**).

39. Check that there are 4 records in this page and these are the ones ending with ‘C’, ‘A’, ‘B’, ‘C’.

40. In the same query window, write the following code:

```
DBCC PAGE('ExampleDB', 4, 8, 1);
```

41. Before executing this command, note the following:
    - In this case, our **file ID** is 4 and our **page ID** is 8. You should replace these values with the **file ID** and the **page ID** that you have obtained earlier for partition 2.

42. Execute the command above (only the commands above, by highlighting the code and pressing **Execute**).

43. Check that there are 5 records in this page and these are the ones ending with 'B', 'C', 'A', 'C', 'A'.

44. Open a new query window and write the following code:

```
USE ExampleDB;

SET STATISTICS IO ON;

SELECT * FROM ExampleTable;
```

45. Before executing this code, note the following:
    - The command **SET STATISTICS IO ON** turns on statistics about the amount of disk activity generated by queries.
    - The query retrieves all records from **ExampleTable**.

46. Execute the statements above, and the contents of **ExampleTable** will appear in the **Results** tab.

47. Switch to the **Messages** tab, and note the following:
    - The **scan count** is 2 because there are two partitions to retrieve data from (which requires two seek/scan operations).
    - The number of **logical reads** is 2 because there are two data pages to retrieve (one data page in each partition; it could be larger if there were more data pages in each partition).
    - The number of **physical reads** is 0 because the data pages did not have to be read from disk (they were already in memory).

48. In the toolbar, press the button **Include Actual Execution Plan** (the button will remain pressed).
    Note: If you cannot find it in the toolbar, the same option is available in the Query menu.

49. Highlight the query (only the query) and press **Execute**:

```
SELECT * FROM ExampleTable;
```

50. Switch to the **Execution plan** tab, and check that the system is performing a **Table Scan**.

51. Hover the mouse (or click) over the **Table Scan**, and a large tooltip will appear.

52. Check the number of rows, the partition count, and the object that the system is operating on.

53. Imagine that you insert many more record in this table. Then:
    - As the table grows, more pages are allocated in the data file. As previously explained, new pages are allocated in groups of 8, called extents.

- As the file grows, SQL Server needs to know which extents contain pages of a given object. For this purpose, SQL Server uses a special type of page, called IAM page (for Index Allocation Map, but not to be confused with database indexes).
- Internally, an IAM page contains a bitmap where each bit refers to an extent in the file, and the bit value (1 or 0) indicates whether the extent has been allocated to the object or not.

54. Open a new query window and write the following code:

```
SELECT partition_id, allocated_page_page_id
FROM sys.dm_db_database_page_allocations(db_id('ExampleDB'),
                                         object_id('ExampleTable'),
                                         NULL, NULL, 'DETAILED')
WHERE page_type_desc = 'IAM_PAGE';
```

55. Before executing the code, note the following:
    - The system function **sys.dm_db_database_page_allocations** provides information about the pages that belong to a particular database object (in this case, **ExampleTable**).
    - The type of page that we are interest now is IAM pages.

56. Execute the statement above, and you will notice that each partition has its own IAM page.
    Note: An IAM page can cover about 4 GB of data, so the table would have to grow considerably before another IAM page needs to be created.

57. Take note of the **page IDs** for those IAM pages.
    *Note: The **file IDs** for those partitions will be the same as before.*

58. In the same query window, write the following commands:

```
DBCC TRACEON(3604);
DBCC PAGE('ExampleDB', 3, 16, 1);
```

59. Before executing these commands, note the following:
    - In this case, our **file ID** is 3 and our **page ID** is 16. You may have to replace these values with the correct **file ID** and **page ID** that you have obtained earlier.

60. Execute the commands above, and note that the IAM page contains two records:
    - The first record (shorter) is an IAM header with specific information about this IAM page, such as a sequence number (for IAM pages), the starting extent of the range of extents mapped by this IAM page, and single page allocations in mixed extents, if any.
    - The second record (longer) is the actual bitmap that indicates which extents have been allocated to the object in the same partition where this IAM page is located. Given the small number of records that we are working with, you will notice that most of this bitmap is filled with zeros, i.e. very few extents have been allocated to this table.

61. Confirm that the scenario is very similar for the IAM page in the other partition.

62. In **Object Explorer**, right-click the **ExampleDB** database, and select **Delete**.

63. In the **Delete Object** window, check the option **Close existing connections** and press **OK**.

64. The **ExampleDB** database has been dropped. You can close **SQL Server Management Studio**.