In this lab class, we will approach the following topics:

1.1 - Clustered and Non-Clustered Indexes

In SQL Server, database indexes are based on the B+Tree data structure.[1] There are two types of indexes, namely **clustered** that impose a physical ordering on the data, and **non-clustered,** which have no influence on the physical order of the data. On SQL server, the leaf nodes of a clustered index contain the data pages of the underlying table. The leaf nodes of a non-clustered index, on the other hand, do not store the data but instead pointers to the data rows (i.e. a Row ID, composed of a Page ID plus a row number).

SQL Server has also the particularity that **if a clustered index has been created on a table**, the non-clustered **index pointers** are no longer pointers to the data rows, instead **taking a route through the clustered index**. Thus, each leaf level entry of a non-clustered index consists of the non-clustered index key, plus the clustered index key. This approach avoids the work in adjusting non-clustered index entries when a data page splits because of insertion into a clustered index, causing data rows to migrate to new pages.

It is not uncommon to create an index that consists of more than one column. Such an index is known as a **composite index**. In SQL Server, an index can be created with **no more than 16 columns**. Also, the sum of the column sizes in the index cannot be greater than **900 bytes** (see the section on hash indexes for a technique that deals with this).

1.2 - The Effect of Indexes on Queries and Insertions

SQL Server indexes affect not only the performance of queries, but also the behavior of the database when rows are inserted. For non-clustered indexes, insertion and deletion may not change the index. For a clustered index, because the entire table is built as an index, any data

[1] http://sqlity.net/en/2445/b-plus-tree/

manipulation activity will likely change the index. Since insertions and deletions now have a cost *O(log(n))* instead of *O(1)*, indexes have a severe impact on these operations.

For queries, the choice of whether to use an existing index or not, and which one, is a decision that the query optimizer makes. The **more selective is the query**, i.e. the fewer rows returned, the **more likely that an index will be chosen** by the query optimizer. In an example where 90% of the rows in the table are returned, the query optimizer would probably choose a table scan instead of a non-clustered index.

Clustered indexes are often more efficient than their **non-clustered** counterparts. However, when a non-clustered index is used as a **covering index** (i.e., all the columns involved in the query belong to the index key) its usage is **normally more efficient** than using an equivalent clustered index, because the query can be answered based on the information that is present in the non-clustered index alone, without having to go to the actual table data.

1.3 - Creating Indexes on SQL Server

Indexes can be created using a T-SQL **CREATE INDEX** statement,[2] or with SQL Server Management Studio. An index is also created when a **primary or unique key** constraint is added to a table. The T-SQL syntax for creating an index is as follows:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
ON table | view ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name ) | filegroup_name  | default }
    [ WITH (
         PAD_INDEX  = { ON | OFF }
       | FILLFACTOR = fillfactor
       | SORT_IN_TEMPDB = { ON | OFF }
       | IGNORE_DUP_KEY = { ON | OFF }
       | STATISTICS_NORECOMPUTE = { ON | OFF }
       | DROP_EXISTING = { ON | OFF }
       | ONLINE = { ON | OFF }
       | ALLOW_ROW_LOCKS = { ON | OFF }
       | ALLOW_PAGE_LOCKS = { ON | OFF }
       | MAXDOP = max_degree_of_parallelism
           [ ,...n ] )
           ]
           ]
      [ ; ]
```

We now describe some of the available options in the CREATE INDEX statement:

- The **UNIQUE** keyword ensures that **only one row has a particular key value**. In other words, the uniqueness of the key is enforced.

- The **INCLUDE** option allows **non-key fields** to be **added to a non-clustered index** at the leaf level. In this case, we have a **covering index**.

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

- The **FILLFACTOR** option, which takes a value from 0 to 100, is used to **reserve a percentage of free space on each leaf level** page of the index to accommodate future expansion and reduce the potential for page splits. An index created with a FILLFACTOR of 100 will have its **leaf pages** completely filled. This is useful if no data is to be entered into the table in the future. An index created with a FILLFACTOR below 100 will have its **leaf pages** filled to the FILLFACTOR percentage specified. As for **intermediate (non-leaf) levels** in the index, these will have enough space for at least another index entry. Over time, as rows are inserted into the table, the effectiveness of the FILLFACTOR value will vanish, and a planned rebuilding of critical indexes at periodic intervals should be considered if heavy inserts are made to the table (i.e., the fill factor **is implemented only when the index is created** and it is not maintained after the index is created as data is added, deleted, or updated in the table).

- The **PAD_INDEX** option is also relevant to reserving space. If PAD_INDEX is ON, then the FILLFACTOR setting is applied also to **intermediate (non-leaf) levels** in the index.

- The **IGNORE_DUP_KEY** specifies the response when attempting to insert a duplicate key value into a unique index. If the IGNORE_DUP_KEY option is ON, **rows containing duplicate key values are discarded**, but the **statement will succeed**, with a warning. However, if the IGNORE_DUP_KEY option is OFF, the statement as a whole will be aborted with an error.

- The **ON filegroup_name** option allows the database administrator to **create the index on a filegroup** different than that from the table itself. By using multiple filegroups, disk I/O operations performed on index entries and on data file entries can be spread across separate disk drives, for better performance.

- The **SORT_IN_TEMPDB** option can be used to place **the data from intermediate sort runs**, used while creating the index, into tempdb.[3] This can result in a performance improvement if tempdb is placed on another disk drive or on a RAID array. The default behavior, if this option is OFF, is to utilize space in the database in which the index is being created. This means that the disk heads are moving back and forth between the data pages and the temporary sort work area, which may degrade performance.

- The **ALLOW_ROW_LOCKS** and **ALLOW_PAGE_LOCKS** options are both defaulted to ON, allowing both individual record locking and page locking. Note that SQL Server can take locks at different levels of granularity, such as table, page, or row. These two options allow for considerable tuning flexibility:
    - If **ALLOW_PAGE_LOCKS = OFF**, the lock manager will not take page locks on that index. The manager will only user row or table locks;
    - If **ALLOW_ROW_LOCKS = OFF**, the lock manager will not take row locks on that index. The manager will only use page or table locks.

[3] https://docs.microsoft.com/en-us/sql/relational-databases/databases/tempdb-database

- If **ALLOW_PAGE_LOCKS = OFF** and **ALLOW_PAGE_LOCKS = OFF**, locks are assigned at a table level only.
- If **ALLOW_PAGE_LOCKS = ON** and **ALLOW_PAGE_LOCKS = ON**, SQL Server decides on which lock level to use, according to the amount of rows and memory available.

In general, OLTP (small transaction sized) systems require record locking, while some OLAP systems can work with only page or table locking, if large chunks of data are processed in each transaction.

- The **MAXDOP** option specifies a **maximum degree** (amount) of **parallelism** to allow for a particular index. The default value is 0, meaning that the parallel processing on all available CPUs will be used, depending on other server activities at the same time.

- Usually, table locks are applied for the duration of the create index operation. This prevents all user access to the underlying table for the duration of the operation. By switching **ONLINE** to **ON** (default is **OFF**), this enables the creation of the index without table locks, which allows queries or updates to the underlying table to proceed without interruption.

1.4 - Indexed Views and Indexes on Computed Values

One aspect of index creation that can be seen from the CREATE INDEX syntax is that SQL Server can **create indexes on views**. This can lead to significant improvements from a performance perspective. Unlike a non-indexed view, which does not physically hold data, an indexed view is physically stored (i.e. **the view is materialized**). Any modifications **to the table are reflected in the indexed view**, so they are best created on tables that are changed infrequently.

In SQL Server it is also possible to utilize **computed columns** in an index definition, provided that the definition of the computed column is deterministic (i.e., for a given input, it always produces the same output). Suppose we create the following table:

```
CREATE TABLE accounts (
    account_no      INT NOT NULL,
    customer_no     INT NOT NULL,
    branch_no       INT NOT NULL,
    balance         MONEY NOT NULL,
    account_notes   CHAR (400) NOT NULL,
    taxed_balance   AS (balance * 0.9)
);
```

We can create an index over the *taxed_balance* column with the command:

```
CREATE INDEX nci_taxed_balance ON accounts (taxed_balance);
```

1.5 - Hash Indexes in SQL Server

SQL Server does not support hash indexes natively (except for in-memory OLTP, which is out of the scope of this course). However, by using a combination of the CHECKSUM function and indexes on computed columns, it is possible to simulate hash indexes. This can be particularly beneficial for the queries over wide columns (e.g. text columns with more than 900 bytes, the size limit for B+Tree index keys).

A column with the hash value can be added to a table as follows:

```
alter table Names add NameHash as checksum(Name);
```

Then, we can build an index for the hash value as shown below:

```
create index IX_Names_NameHash on Names(NameHash);
```

Finally, a query that uses the hash index can be made as follows:

```
select customerId
from Names
where NameHash = checksum('Grover') and Name='Grover';
```

When executing the query, SQL Server recognizes that the hash column will eliminate the most number of rows, thus executing the *data seek* on the hash value first. Only then does it perform the more the expensive *Name* check. At that stage SQL Server should only need to compare a very small number of rows, normally only one if the *Name* exists.

2. Exercises and Examples to Execute

2.1 - Inspecting indexes with SQL Server Management Studio

In this exercise, we will inspect some of the tables and indexes in the AdventureWorks database.

1. Start up **SQL Server Management Studio** and connect to SQL Server.
2. In **Object Explorer**, expand **Databases**, then **AdventureWorks**, and finally **Tables**.
3. Right-click the table **Person.Person** and select the **Design** option.
4. At the top of the window, in the menu **Table Designer**, select **Indexes/Key**.
5. Identify the **clustered index** associated with the **primary key**.
6. Identify the **non-clustered** index associated with **LastName, FirstName, MiddleName**.

Do the same for the **Production.Product** table:

- Identify the **clustered index** associated with the **primary key**.
- Identify the **non-clustered indexes** associated with **ProductNumber** and **Name**.

Do the same for the **Sales.Store** table:

- Identify the **clustered index** associated with the **primary key**.
- Identify the **non-clustered index** associated with **SalesPersonID**.

Another way to inspect the indexes on a table is to use the **Query Window** and executing the **sp_helpindex** system stored procedure. For this purpose, open a new **Query Window** and execute the following statement:

```
EXEC sp_helpindex 'Person.Person';
```

Check that the information in the **Results** tab agrees with what you have seen before.

Execute a similar statement for the **Production.Product** table:

```
EXEC sp_helpindex 'Production.Product';
```

Finally, execute a similar statement for the **Sales.Store** table:

```
EXEC sp_helpindex 'Sales.Store';
```

A useful function that can be used to obtain information about the properties of an index is the INDEXPROPERTY function, which takes the following form:

```
INDEXPROPERTY(table_ID, index, property);
```

Among other things, the INDEXPROPERTY function allows us to retrieve information regarding an index depth, if the index is clustered or not, or if the index is unique or not. More information can be found in the online documentation.[4]

Some examples for the usage of the INDEXPROPERTY function are as follows:

```
SELECT INDEXPROPERTY(OBJECT_ID('Person.Person'),
                     'PK_Person_BusinessEntityID',
                     'IsClustered');
SELECT INDEXPROPERTY(OBJECT_ID('Person.Person'),
                     'PK_Person_BusinessEntityID',
                     'IndexDepth');
SELECT INDEXPROPERTY(OBJECT_ID('Person.Person'),
                     'PK_Person_BusinessEntityID',
                     'IsUnique');
```

The above instructions return a table with a single row and column, containing the value of the requested property.

[4] https://docs.microsoft.com/en-us/sql/t-sql/functions/indexproperty-transact-sql

2.2 - Optimizing queries with indexes

Open a new query window and try out the following set of instructions:

1. Use the following command to see the indexes on **Person.Address**:

```
EXEC sp_helpindex 'Person.Address';
```

Identify the **clustered index** associated with the **primary key**.

2. Execute the following commands:

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
```

The first command above instructs SQL Server to display information regarding the number of I/O operations, and the second one instructs SQL Server to display information regarding the amount of time involved in each operation.

3. Execute the following query:

```
SELECT * FROM Person.Address WHERE AddressID='1000';
```

The query results are shown in the **Results** tab.

Change to the **Messages** tab to see the number of I/O operations and the execution time.

4. With the same query on the query window:

```
SELECT * FROM Person.Address WHERE AddressID='1000';
```

Press the button **Display Estimated Execution Plan** to show the query execution plan. Check that the system is using an index (which index?) to answer the query.

5. Delete the primary key index from the **Person.Address** table by executing the following commands:

```
ALTER TABLE Sales.SalesOrderHeader
DROP CONSTRAINT FK_SalesOrderHeader_Address_ShipToAddressID;

ALTER TABLE Sales.SalesOrderHeader
DROP CONSTRAINT FK_SalesOrderHeader_Address_BillToAddressID;

ALTER TABLE Person.BusinessEntityAddress
DROP CONSTRAINT FK_BusinessEntityAddress_Address_AddressID;

ALTER TABLE Person.Address DROP CONSTRAINT PK_Address_AddressID;
```

6. Now check the execution plan for the same query again:

```
SELECT * FROM Person.Address WHERE AddressID='1000';
```

The system is no longer using an index, and instead is doing a full **table scan**.

8. Execute the following command to re-create the primary key index:

```
ALTER TABLE Person.Address
ADD CONSTRAINT PK_Address_AddressID PRIMARY KEY(AddressID);
```

In the **Results** tab, check the number of I/O operations and the execution time.

9. Check the execution plan for the same query again:

```
SELECT * FROM Person.Address WHERE AddressID='1000';
```

The system is using the primary key index that has been just re-created.

10. Check the execution plan a different query:

```
SELECT ModifiedDate FROM Person.Address
WHERE ModifiedDate = '2014-01-01';
```

The system is going through all the records in the table by scanning the clustered index associated with the primary key.

11. Create a new index for ModifiedDate on Person.Address:

```
CREATE INDEX date_idx ON Person.Address(ModifiedDate);
```

12. Check the execution plan again:

```
SELECT ModifiedDate FROM Person.Address
WHERE ModifiedDate = '2014-01-01';
```

The system now using the non-clustered index that we have just created. This is a **covering index** for this query, i.e. the query can be answered based on the index alone.

13. Modify the query to select all columns and check the execution plan:

```
SELECT * FROM Person.Address
WHERE ModifiedDate = '2014-01-01';
```

The system is now using two indexes. It uses the **non-clustered index** to locate the records with the desired ModifiedDate, and then uses the **clustered index** to retrieve all the columns for those records.

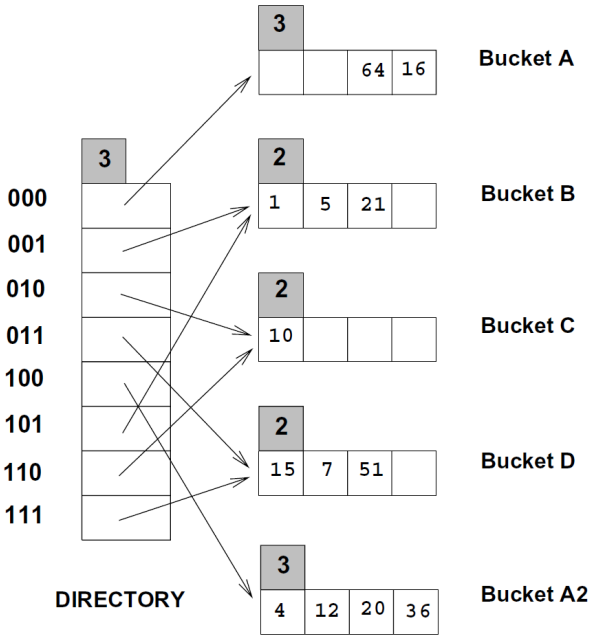
2.3 - B+-Tree Indices

a) Consider a B+Tree with 3 pointers in each internal node, and in which values are sorted alphabetically. Show the B+Tree data structure resulting from the sequential insertion of each of the following values: ***A, C, L, R, X, Y, Z***.

b) Consider the final result from the previous question. Show the B+Tree data structure resulting from the sequential deletion of each of the following values: ***R, X, Y, Z, A, C, L***.

2.4 - Extendable Hashing

Consider the Extendable Hashing index from the figure below, where the least significant bits of the value resulting from hash function ***h(x)=x mod 100*** is used. Answer the following questions about this index:



a) Show the index after inserting an entry with value 68.

b) Show the index after inserting entries with values 17 and 69 into the original structure.

c) Show the index after inserting a second entry with value 36 into the original structure.