In this lab class we will approach the following topics related to query optimization.

1.1 - Basics of Query Optimization in SQL Server

The SQL Server **query optimizer** takes a query and attempts to execute it in the most efficient way. SQL Server uses **cost-based optimization**, i.e. it tries to find the execution plan with the lowest possible cost, where cost means both the time the query will take to execute and the hardware resources that will be used. Basically, the query optimizer is looking to **minimize the number of reads** required to fetch the required data.

The bad news is that the optimizer does not always come up with the best solution. A database administrator should be aware of the factors that govern query optimization, what pitfalls there are, and how the query optimizer can be assisted in its job. Database administrators who know their data well can often influence the optimizer with hints on how to use the indexes for choosing the most efficient solution.

The query optimizer takes into account information such as the query itself, indexes and key distribution statistics, size of the table, and rows per page. It then calculates the logical read cost given a possible access path. For optimizing join orders, SQL Server uses the number of rows that are fetched by each join condition.

1.2 - Index and Column statistics in SQL Server

To obtain **information on the indexes present on a table**, as well as on their characteristics, SQL Server checks the **sys.*sysindexes*** and **sys.*sysobjects*** system views, establishing the indexes that are indeed present in a table by checking the rows from **sys.*sysindexes*** that have a value in the id column equal to the id in the **sys.*sysobjects*** system view. Other columns in the **sys.*sysindexes*** view are used by the query optimizer to determine on which columns is the index based.

Even when there are several indexes present, the query optimizer may choose to use no indexes, if a table scan is estimated to be a more efficient access mechanism. The optimizer may also choose to use one or more indexes.

The query optimizer uses **statistical information** available in **the distribution statistics** associated with the table attributes. By default, SQL Server maintains statistics on index and key columns of all tables. Distribution statistics are usually calculated when an index is created (except when the index is created on an empty table) and are held in **sys.stats** and **sys.stats_columns**. Statistics are automatically updated when certain thresholds within SQL Server are met, e.g. when the number of rows in the table increases or decreases by 10% of the number of rows the statistic was based on.

Besides key distribution statistics, SQL Server will **optionally maintain statistics on non-indexed columns, or columns in a composite index key other than the first**. If the database option **AUTO CREATE STATISTICS** is set to on, and if a column on which index statistics are not being maintained is referenced in a WHERE clause, statistics will be gathered if they help the query optimizer.

Now suppose we execute the following query over the *Sales.SalesOrderHeader* table, which contains about 30 000 records:

```
SELECT SalesOrderID
FROM Sales.SalesOrderHeader
WHERE TotalDue > 9999;
```

Will 30 000 rows be returned, or 1000 rows, or 25 rows, or 0 rows? If the table has a non-clustered index on the TotalDue column, then the query optimizer needs to estimate the number of rows to be returned so it can decide whether the non-clustered index should be considered interesting or whether a table scan is likely to be more efficient.

SQL Server maintains the following general information about tables and indexes:

- **Number of rows in the table/index (i.e., rows column in sys.sysindexes)**
- **Number of pages occupied by the table/index (i.e., dpages column in sys.sysindexes).**

SQL Server also collects the following statistics about tables and index keys (and optionally about other table columns):

- **Time when the statistics were collected**
- **Number of rows used to produce the histograms**
- **Density information,** i.e. the number of duplicates in a given column or combination of columns, calculated as 1/(number of distinct values)
- **Average key length**
- **Single-column histogram,** including the number of *steps* in the histogram
- **A string summary,** if the column contains character data. This string summary collects information about the frequency distribution of sub-strings, being useful for queries using the *LIKE* operator.

A **histogram** is a set of up to 200 values of a given column. All or a sample of the values in a given column are sorted; the ordered sequence is divided into up to 199 intervals (the steps) so that the most statistically significant information is captured. In general, these intervals are of non-equal size. The following values, or information sufficient to derive them, are stored with each step of the histogram:

- **RANGE_HI_KEY** - A key value showing the upper boundary of a histogram step
- **RANGE_ROWS** - Specifies how many rows are inside the range (they are smaller than this RANGE_HI_KEY, but bigger than the previous smaller RANGE_HI_KEY)
- **EQ_ROWS** - Specifies how many rows are exactly equal to RANGE_HI_KEY
- **DISTINCT_RANGE_ROWS** - Specifies how many distinct key values are inside this range (not including the previous key before RANGE_HI_KEY and RANGE_HI_KEY itself).
- **AVG_RANGE_ROWS** - Average number of rows per distinct value inside the range

We can use the command **`DBCC SHOW_STATISTICS('table_name', 'target')`** to investigate statistics objects, where target is an index name or a statistics collection name. For example, in the AdventureWorks database:

```
DBCC show_statistics('Person.Person', 'PK_Person_BusinessEntityID')
```

will show the histogram steps for the primary key of table **Person.Person**.[1]

The stored procedure **sp_helpstats** can be used to list all the statistics available for a given table:

```
EXEC sp_helpstats @objname = 'Person.Person', @results = 'ALL';
```

When a column is used in an equality predicate, the number of qualifying rows (i.e., the fraction of rows from the input set of the predicate that satisfy the predicate, commonly referred to as the ***cardinality*** or ***selectivity***) is estimated using the **density** derived from the histogram. The density is the average fraction of duplicate index key values in the index. Multiplying the total count of rows in the table by the index density, we can obtain the likely number of row hits by specifying a given value.

For example, if a table T has 100 000 rows, a query contains a selection predicate of the form T.a=10, and a histogram shows that the selectivity of T.a=10 is 10%, then the cardinality estimate for the fraction of rows of T that must be considered by the query is 10% * 100 000 = 10 000.

Histograms are also used to estimate the selectivity of non-equality selection predicates, joins, and other operators.

[1] For more information about this command, see the online documentation:
https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-show-statistics-transact-sql

In what regards index key statistics, SQL Server **only stores key values for the first column of a composite index**. It is **therefore better to choose the most selective column as the first column of an index** — that is, the column with the least number of duplicate values. Consider a three-column composite index on the columns region, state, city. There are normally few regions, more states, and many more cities, so the index should be created in the city, state and region column order.

If the key distribution statistics do not exist, the query optimizer applies a weighting to each operator. These weightings are very general estimates and can be inaccurate.

- For the **=** operator, assume 10%, which means that the query optimizer will assume that 10 percent of the rows in the table will be returned.
- For the **<** or **>** operators assume 33%, which means that the query optimizer will assume that 33 percent of the rows in the table will be returned.
- For the **BETWEEN** operator assume 12%, which means that the query optimizer will assume that 12 percent of the rows in the table will be returned.

If we have a unique index matching the search argument, then the query optimizer knows immediately the number of rows returned by the equality operator. Because of the unique index, the query optimizer knows that at most one row can be returned (of course, zero rows could be returned), so this figure is used rather than the 10 percent weighting.

1.3 - Hinting the SQL Server Query Optimizer

In most cases, the SQL Server Query Optimizer will correctly evaluate a query and run it as optimally as possible. But on occasion, the Query Optimizer will fail, producing a less than optimal execution plan. Query performance will suffer because of it. When you identify such a query, you can override the query optimizer using what is called an **optimizer hint**.

As a general note on hints, they are not always the most prudent option. **It is always best to attempt to get as much out of a query as possible, using just the query**. **Leave the hint for later.** The general trend in relational database technology is that hints are now more a forced rather than a suggested influence on an optimizer. Hints can nonetheless be useful to test assumptions, such as why a specific index is not used as one would expect.

Join hints can be used to force the query optimizer to create a query plan that adopts a **particular join technique** when joining two tables. The join type is specified as part of the JOIN clause, as follows:

```
SELECT *
FROM Sales.Customer INNER HASH JOIN Sales.SalesTerritory
     ON Sales.Customer.TerritoryID = Sales.SalesTerritory.TerritoryID
WHERE CustomerID > 10000;
```

The **type of join** may also be specified as a **query hint**, in which case the join type will be applied to all the joins in the query, as follows:

```
SELECT *
FROM Sales.Customer INNER JOIN Sales.SalesTerritory
     ON Sales.Customer.TerritoryID = Sales.SalesTerritory.TerritoryID
     INNER JOIN Person.Person
     ON Person.Person.BusinessEntityID = Sales.Customer.PersonID
WHERE Sales.Customer.CustomerID > 10000 OPTION(HASH JOIN);
```

Another practical hint is an option that can force the query optimizer to **change the join order** to that specified by the query syntax, as follows:

```
SELECT *
FROM Sales.Customer INNER JOIN Sales.SalesTerritory
     ON Sales.Customer.TerritoryID = Sales.SalesTerritory.TerritoryID
     INNER JOIN Person.Person
     ON Person.Person.BusinessEntityID = Sales.Customer.PersonID
WHERE Sales.Customer.CustomerID > 10000 OPTION(FORCE ORDER);
```

Table hints can be used to **dictate the access method** to use when retrieving data from a table. This can be a table scan, a single index, or multiple indexes. An example of the syntax used is as follows:

```
SELECT *
FROM Sales.Customer WITH (INDEX(AK_Customer_AccountNumber))
WHERE AccountNumber BETWEEN 'AW00000100' AND 'AW00000200'
      AND TerritoryID BETWEEN 5 and 10;
```

The above example forces the query optimizer to adopt a plan that uses the index *AK_Customer_AccountNumber* to access the *Sales.Customer table*.

Now suppose a table scan is to be forced. The following syntax **forces the query optimizer to adopt a plan that uses a table scan** if there is no clustered index present on the table or that uses a clustered index scan if there is:

```
SELECT *
FROM Sales.Customer WITH (INDEX(0))
WHERE AccountNumber BETWEEN 'AW00000100' AND 'AW00000200'
      AND TerritoryID BETWEEN 5 AND 10;
```

If there is a clustered index present on the table, a **clustered index scan or seek can be forced**, as shown in the following example:

```
SELECT *
FROM Sales.Customer WITH (INDEX(1))
WHERE CustomerID BETWEEN 100 and 200
      AND TerritoryID BETWEEN 5 and 10;
```

To force a query plan to **deliver the first rows quickly**, perhaps at the expense of the whole query, the **FAST** query hint can be used, as in the following example:

```
SELECT *
FROM Sales.Customer INNER JOIN Sales.SalesTerritory
     ON Sales.Customer.TerritoryID = Sales.SalesTerritory.TerritoryID
     INNER JOIN Person.Person
     ON Person.Person.BusinessEntityID = Sales.Customer.PersonID
WHERE Sales.Customer.CustomerID > 10000 OPTION(FAST 10);
```

The query hint above will force the query optimizer to create a plan that will be optimized to return the first ten rows.

The following query hint forces the query optimizer to **use hashing when calculating the aggregate**:

```
SELECT Type, COUNT(SpecialOfferID)
FROM Sales.SpecialOffer
GROUP BY Type OPTION (HASH GROUP);
```

If an **ordering (sorting) rather than a hashing** technique should be used, then this can be forced as follows:

```
SELECT Type, COUNT(SpecialOfferID)
FROM Sales.SpecialOffer
GROUP BY Type OPTION (ORDER GROUP);
```

The following example will force the use of a **concatenation to perform the union**, and thus a subsequent operation will be needed to eliminate the duplicate rows (in SQL Server, the concatenation operator simply copies rows from the first table to the output, and then repeats this operation for each additional table, being commonly used to implement the UNION ALL SQL operation):

```
SELECT CustomerID from Sales.Customer
UNION
SELECT BusinessEntityID from Sales.SalesPerson
OPTION(CONCAT UNION);
```

The following example will force the use of a **hash operation to perform the union**, thus eliminating the duplicate rows at once:

```
SELECT CustomerID from Sales.Customer
UNION
SELECT BusinessEntityID from Sales.SalesPerson
OPTION(HASH UNION);
```

The following example will force the use of **merge operation to perform the union**, which also eliminates the duplicate rows at once. Normally, the merge operation would exploit the sorted order of the inputs in a manner similar to a merge join:

```
SELECT CustomerID from Sales.Customer
UNION
SELECT BusinessEntityID from Sales.SalesPerson
OPTION(MERGE UNION);
```

2 – Experiments and Exercises

2.1 - A benchmark experiment for SQL Server Indexes

Through the following benchmark, students will be able to **see how clustered/non-clustered indexes influence the performance of SQL Server queries**. The command **DBCC DROPCLEANBUFFERS** should be executed after every operation, clearing data from the cache and this way ensuring a fair testing.

a) Start by inserting some values into a new table:

```
SELECT SalesOrderID AS OrderNo,
       ProductID AS ItemCode,
       OrderQty AS Qty,
       UnitPrice as Price,
       'Processing order...' as Status
       INTO dbo.MyOrderDetails
FROM Sales.SalesOrderDetail;
```

b) Turn on the run statistics with the following commands:

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

c) Now, try a simple query without any indexes (cleaning the cache first):

```
DBCC DROPCLEANBUFFERS;
SELECT OrderNo, ItemCode, Qty, Price
FROM dbo.MyOrderDetails
WHERE (ItemCode = 709) AND (OrderNo BETWEEN 20000 AND 60000);
```

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

d) Now, create a non-clustered index for the search attributes:

```
CREATE NONCLUSTERED INDEX IX_Order_Details_ItemCode
ON dbo.MyOrderDetails(ItemCode);
CREATE NONCLUSTERED INDEX IX_Order_Details_OrderNo
ON dbo.MyOrderDetails(OrderNo);
```

e) After creating the indexes, try the same query as in c). Check the number of I/O operations and the execution time of the query.

f) Next, create a non-clustered composite index:

```
DROP INDEX IX_Order_Details_ItemCode ON dbo.MyOrderDetails;
DROP INDEX IX_Order_Details_OrderNo ON dbo.MyOrderDetails;
CREATE NONCLUSTERED INDEX IX_Order_Details_Coverindex
ON dbo.MyOrderDetails(OrderNo, ItemCode, Qty, Price);
```

g) Try the same query as in c). Check the number of I/O operations and the execution time of the query.

h) Finally, create a clustered index:

```
DROP INDEX IX_Order_Details_Coverindex ON dbo.MyOrderDetails;
CREATE CLUSTERED INDEX IX_Order_Details_Clustered
ON dbo.MyOrderDetails(OrderNo, ItemCode);
```

i) Try the same query as in c). Check the number of I/O operations and the execution time of the query.

2.2. Exercises

2.2.1. For each of the following pairs of expressions, give instances of relations that show the expressions are not equivalent:

a) $\pi_A(R - S)$ and $\pi_A(R) - \pi_A(S)$

b) $\sigma_{B<4}({}_A G_{\max(B)}(R))$ and ${}_A G_{\max(B)}(\sigma_{B<4}(R))$

c) In the case of b), if both occurrences of *max* were replaced by *min* would the expressions be equivalent? Justify.

d) Consider three relations in the form: R(A,B), S(A,C), and T(A,D).
Show that ((R ⟕ S) ⟕ T) and (R ⟕ (S ⟕ T)) are not equivalent. In other words, the natural left outer join is not associative.

2.2.2. Consider two relations R(A,B) and S(A,C). The primary key of R is A, and the primary key of S is C. Consider also that:
- R contains 20 000 tuples, in 300 blocks.
- S contains 10 000 tuples, in 100 blocks.

a) Estimate, and justify, the maximum size in tuples of the relation resulting from R⋈S.

b) Suppose that the primary key of R is B, and the primary key of S is C. In this case, what would be the maximum size of R⋈S ? Justify.