In this lab class we will approach the following topics:

1. Query Tuning

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 logical reads required to fetch the required data**.

The bad news is that it is not magic, and 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 well their data can often influence the optimizer to choose certain indexes, in order to come up with the most efficient solution.

1.1. Rules of Thumb for Query Tuning

There are some very basic guidelines for writing efficient SQL code. These guidelines largely constitute nothing more than **writing queries in the proper way**. In fact, it might be quite surprising to learn that as you work with a relational database, one of the most common causes of performance problems can usually be tracked down to poorly coded queries. We will now discuss in general terms what, in SQL statements, is good for performance, and what is not.

- **Make careful use of the HAVING clause**. HAVING is intended to filter records from the result of a GROUP BY, and a common mistake is using it to filter records that can be more efficiently filtered using a WHERE clause.

- **Make careful use of the DISTINCT clause.** DISTINCT can cause an additional sort operation, and it should be avoided except when strictly necessary.

- Make careful use of **functions**. They **simply should not be used where you expect an SQL statement to use an index**. When using a function in a query, do not execute it against a table field if possible. Instead, apply it on the search value, for example: ***SELECT * FROM customer WHERE zip = TO_NUMBER('94002')***

- **Data type conversions** are often a problem and will likely conflict with existing indexes (e.g., an index is created over a VARCHAR atribute named *dateStr*, but a query accesses the atribute as if it were a date, for instance through a conversion function such as ***CONVERT(DATETIME, dateStr)***). Unless implicit data type conversion occurs, e.g. between number and string**, indexes will likely be ignored**.

- **Make careful use of the ORDER BY clause**. Avoid sorting the results when that is not strictly necessary.

- **Use UNION ALL instead of UNION**. When you use the UNION clause to concatenate the results from two or more SELECT statements, duplicate records are removed. This duplicate removal requires additional computing. If you are not concerned that your results may include duplicate records, use the UNION ALL clause, which concatenates the full results from the SELECT statements.

- **Avoid anti-comparisons**. Avoid instructions such as != or NOT, as they are looking for what is not in a table — the entire table must be read regardless.

- Use **IN to test against literal values** and **EXISTS to create a correlation between a calling query and a subquery**. IN will cause a subquery to be executed in its entirety before passing the result to the calling query. EXISTS will stop once a result is found.

- **Avoid using the OR or the IN operators**. Notice, for instance, that ***SELECT * FROM employees WHERE state IN ('CA', 'IL', 'KS')*** is the same as ***SELECT * FROM employees WHERE state = 'CA' OR state = 'IL' OR state = 'KS'***, and both are costly queries to execute. The query optimizer will always perform a table scan (or a clustered index scan on an indexed table) if the WHERE clause in the query contains an OR operator, and if any of the referenced columns in the OR clause does not have an index with the column as the search key. If you use many queries containing OR clauses or IN operators, you will want to ensure that each referenced column has an index. A query with one or more OR clauses, or using the IN operator, can sometimes be rewritten as a **series of queries that are combined with a UNION statement**, in order to boost the performance.

- For **optimizing joins**, the following rules of thumb apply:

- Use equality first, and only use range operators where equality does not apply.
- Avoid the use of negatives in the form of != or NOT.
- Avoid LIKE pattern matching.
- In the relations being joined, try to retrieve specific rows, and in small numbers, so that only a small number of rows is actually involved in the join operation(s).
- Filter large tables before applying a join operation, to reduce the number of rows that is joined. Also access tables from the most highly filtered, preferably the largest, downward. Notice that this is importante to reduce the number of rows that is involved in the join operation(s).
- Use indexes wherever possible except for very small tables.

Regarding joins, nested sub-queries can be difficult to tune but can often be a viable tool, and sometimes highly effective, for tuning ***mutable complex joins***, with three and sometimes many more tables in a single query. The term *mutable complex joins* refers to join queries involving more than two tables, that are *mutable* in the sense that different join orders can be considered, and that are complex in the sense that they involve other selections on the tables that are being joined. Using nested sub-queries, it might be easier to tune such queries, because one can tune each sub-query independently.

2. Index Tuning

In terms of tuning, the option that produces maximum gains with least impact on existing systems and processes is to examine your indexing strategy. However, the task of identifying the right indexes is not necessarily straightforward. It requires a sound knowledge of the sort of queries that will be run against the data, the distribution of that data, and the volume of data, as well as an understanding of what type of index will best suit your needs. Consider the following query:

Select A, COUNT(*) FROM T WHERE X < 10 GROUP BY A;

The following different physical design structures can reduce the execution cost of this query:

(i) A clustered index on X;
(ii) Table range partitioned on X;
(iii) A non-clustered index with key X)and including the additional atribute A;
(iv) A materialized view that matches the query, and so on.

These alternatives can have widely varying storage and update characteristics. Thus, in the

presence of storage constraints, or for a workload containing updates, making a global choice for a workload is difficult. For example, a clustered index on a table and horizontal partitioning of a table are both non-redundant structures (i.e., they incur negligible additional storage overhead) whereas non-clustered indexes and materialized views can be potentially storage intensive and involve higher update costs. However, non-clustered indexes and materialized views can often be much more beneficial than a clustered index or a horizontally partitioned table. Clearly, a physical design tool that can give an integrated physical design recommendation can greatly reduce/eliminate the need for a DBA to make ad-hoc decisions.

While understanding the basics is still essential, SQL Server does offer a helping hand in the form of some tools – in particular, the **Database Engine Tuning Advisor** – that can help to determine, tune and monitor your indexes. It can be used to get answers to the following questions:

- Which indexes are needed for specific queries?
- How to monitor index usage and its effectiveness?
- How to identify redundant indexes that could negatively impact performance?
- As the workload changes, how to identify missing indexes that could enhance performance for the new queries?

2.1. Using the Database Engine Tuning Advisor

Determining exactly the right indexes for your system can be quite a taxing process. For example, you have to consider:

- Which columns should be indexed, based on the knowledge on how the data is queried.
- Whether to choose a single-column index or a multiple column index.
- Whether to use a clustered index or a non-clustered index.
- Whether one could benefit from an index with included columns.
- How to utilize indexed (i.e., materialized) views.

Moreover, once you have determined the perfect set of indexes, your job is not finished. Your workload will change over time (i.e., new queries will be added, and older ones removed) and this might warrant revisiting existing indexes, analyzing their usage and making adjustments (i.e., modifying or dropping existing indexes and creating new ones). Maintenance of indexes is critical to ensure optimal performance in the long run.

The **Database Engine Tuning Advisor (DTA)** is a physical design tool providing an integrated console where DBAs can tune all **physical design features** supported by the server. The DTA takes into account all aspects of performance that the query optimizer

can model, including the impact of multiple processors, amount of memory on the server, and so on. It is important to note, however, that query optimizers typically do not model all the aspects of query execution (e.g., impact of indexes on locking behavior, impact of data layout etc.). Thus, DTA's estimated improvement can be different from the actual improvement in execution time.

Taking as input a workload to fine-tune, i.e., a set of SQL statements that execute against the database server, the DTA produces a **set of physical design recommendations**, consisting of **indexes, materialized views, and strategies for horizontal range partitioning of tables, indexes and views**. The basis of DTA's recommendations is a *what-if* analysis provided by the SQL Server query optimizer, which allows the computation of an estimated cost as if a given configuration (e.g., the existence of some indexes) was materialized in the database. Similarly to the actual evaluation of a given query plan, the query optimizer component can do an evaluation considering the *what-if* existence of a given physical design structure.

You can tune a single query or the entire workload to which your server is subjected. A workload can be obtained, for instance, by using **SQL Server Profiler**, i.e., a tool for logging events (e.g., queries) that execute on a server. In this case, the workload would be given to the DTA in the form of a trace file, obtained with the SQL Server Profiler. The Profiler tool is just used to collect the workload, whereas the DTA performs the actual analysis and the tuning suggestions.

Alternatively, a workload can be specified as an SQL file containing an organization or industry benchmark. In this case, a text file with the SQL for each query in the workload would be given to the DTA.

The DTA can also take as input workloads referring to either a single or to a set of databases, as many applications use more than one database simultaneously.

Based on the options that you select, you can use the DTA to make recommendations for several Physical Design Structures (PDS), including:

- Clustered indexes
- Non-clustered indexes
- Indexes with included columns (to avoid bookmark lookups)
- Indexed views
- Partitions

The first step is to collect a workload for DTA to analyze. You can do this in one of two ways:

- **Using the Management Studio –** If you need to optimize the performance of a single query, you can use Management Studio to provide directly an input to DTA. Type the query in Management Studio, highlight it and then right click on it to choose Analyze in Database Engine Tuning Advisor.

- **Using the Profiler –** If you want to determine the optimum index set for the entire workload, corresponding to the actual queries that are being executed against an SQL Server instance, you should collect a profiler trace with the **TUNING template** (i.e., one of the possible options for the trace file that is generated by the SQL Server Profiler, and that contains all the information that is required by the Database Engine Tuning Advisor).

To fully exploit the effectiveness of DTA, you should always use a representative profiler trace. For instance, the indexes and partitioning considered by the DTA are limited only to interesting column groups (i.e., those columns that appear in a large fraction of the queries in the workload that have the highest cost), in order to improve scalability with little impact on quality. If the profiler trace is not representative of a true workload, important queries will likely be missing.

You should make sure that you subject your server to all the queries that will typically be run against the data, while you are collecting the trace. This could lead to a huge trace file, but that is normal. If you simply collect a profiler trace over a 5-10 minute period, you can be pretty sure it will not be truly representative of all the queries executed against your database.

In the SQL Profiler, the *TUNING* template captures only minimal events, so there should not be any significant performance impact on your server. A technique for workload compression is also employed, partitioning workloads with basis on a signature of each query (i.e., two queries have the same signature if they are identical in all aspects except for the constants referenced in the query).

3. Experiments and Exercises

3.1. A Practical Exercise Using the Database Engine Tuning Advisor

As materials for this class, we have provided a workload (***Queries4Workload.sql***) against the *AdventureWorks2012 database*. We recommend that you use this workload, to get a hands-on perspective of the DTA. Alternatively to providing an SQL script with the workload, you can use the SQL Profiler Tool to gather a system trace, and provide this trace as input to the DTA.

Assuming that the given workload is representative of the queries that would be executed

against the database, you can use it as an input to the DTA, which will then generate recommendations. You can perform one of the following two types of analysis.

A. Keep my existing Physical Design Structures and tell me what else I am missing

This type of analysis is common and is useful if you have previously established the set of indexes that you deem to be most useful for your given workload, and are seeking further recommendations. To conduct this analysis:

1. **Initiate a new session in the DTA, by launching the corresponding tool from the Windows menu with the SQL Server Performance Tools, or by selecting this tool from the tools menu within the SQL Server Management Studio.**
2. **Choose the provided workload as the input to this session.**
3. **In the "*Select databases and tables to tune*" section, select AdventureWorks2012.**
4. **In the "*Database for workload analysis*" dropdown, use AdventureWorks2012 or Tempdb.**
5. **At the "*Tuning Options*" tab, select the following options:**
   (1) Physical Design Structures to use in database -> Indexes and Indexed views
   (2) Physical Design Structures to keep in database -> Keep all existing PDS
6. **Uncheck the checkbox for limit tuning time.**
7. **Hit START ANALYSIS and DTA will start consuming your workload.**

Once DTA finishes consuming the workload, it will list its recommendations under the recommendations tab.

B. Ignore my existing Physical Design Structures and tell me what query optimizer needs

In the previous scenario, DTA makes recommendations for any missing indexes. However, this does not necessarily mean your existing indexes are optimal for the query optimizer. You may also consider conducting an analysis whereby DTA ignores all existing physical design structures and recommends what it deems the best possible set for the given workload. This way, you can validate your assumptions about what indexes are required.

To conduct this analysis, follow steps 1 to 6 of the previous scenario, except that at step 5(2), choose ***Do not keep any existing PDS***.

Contrary to how this might sound, the DTA will not actually drop or delete any existing physical design structures. This is the biggest advantage of using DTA, as it means you can use the tool to perform *what-if* analysis without actually introducing any changes to the underlying schema.

After consuming the workload, DTA presents, under the recommendations tab, a set of tuning recommendations. A good idea is to focus on the following sections:

- **Recommendation –** this is the action that you need to take. Possible values include Create or Drop.
- **Target of Recommendation –** this is the proposed name of the physical design structure to be created. The naming convention is typical of DTA and generally starts with _dta*. However, it is recommended that you change this name based on the naming convention in your database.
- **Definition –** this is the list of columns that this new physical design structure will include. If you click on the hyperlink, it will open up a new window with the T-SQL script to implement this recommendation.
- **Estimated Improvements –** this is the estimated percentage improvement that you can expect in your workload performance, if you implement all the recommendations made by DTA.
- **Space used by recommendation (MB) –** under the Tuning Summary section of the Reports tab, you can find out the extra space in MB that you would need, if you decide to implement these recommendations.

The reports tab features several in-built analysis reports. There are 15 built-in reports, but the following three are the most important.

- **Current Index Usage Report -** Start with this report to see how your existing indexes are being used by the queries running against your server. Each index that has been used by a query is listed here. Each referenced index has a Percent Usage value which indicates the percentage of statements in your workload that referenced this index. If an index is not listed here, it means that it has not been used by any query in your workload. If you are certain that all the queries that run against your server have been captured by your profiler trace, then you can use this report to identify indexes that are not required and possibly delete them.

- **Recommended Index Usage Report -** Look at this report to identify how index usage will change if the recommended indexes are implemented. If you compare these two reports, you will see that the index usage of some of the current indexes has fallen while some new indexes have been included with a higher usage percentage, indicating a different execution plan for your workload and improved performance.

- **Statement Cost Report -** This report lists individual statements in your workload and the estimated performance improvement for each one. Using this report, you can

identify your poorly performing queries and see the sort of improvement you can expect if you implement the recommendations made by DTA. You will find that some statements don't have any improvements (Percent improvement = 0). This is because either the statement was not tuned for some reason or it already has all the indexes that it needs to perform optimally.

3.2 Implementing the Database Engine Tuning Advisor Recommendations

By now, we have collected a workload using Profiler, consumed it using the DTA, and got a set of recommendations to improve performance. You then have the choice to either:

- **Save recommendations –** you can save the recommendations in an SQL script by navigating to **ACTIONS -> SAVE RECOMMENDATIONS**. You can then manually run the script in Management Studio to create all the recommended physical design structures.
- **Apply recommendations using the DTA –** if you are happy with the set of recommendations, then simply navigate to **ACTIONS -> APPLY RECOMMENDATIONS**. You can also schedule a later time to apply these recommendations, for instance during off-peak hours so that interference with other operations is minimal.

Performing *what-if* analysis is a very useful feature of the DTA. You may not want to apply all the recommendations that the DTA provided. However, since the Estimated Improvement value can only be achieved if you apply all of these recommendations together, you are not really sure what kind of impact it will have if you only choose to apply a sub-set of these recommendations.

To do a *what-if* analysis, deselect the recommendations that you do not want to apply. Now, go to **ACTIONS -> EVALUATE RECOMMENDATIONS**. This will launch another session with the same options as the earlier one. However, when you click on **START ANALYSIS**, the DTA will provide data on estimated performance improvements, based on just this sub-set of the recommendations. Again, the key thing to remember is that the DTA performs this *what-if* analysis without actually implementing anything in the database.

3.3. Exercise

Consider the following normalized relation where the primary key is ID:

Employees(ID, name, salary, department, contract_year)

Consider as well the following four queries equally important and frequent:

a) What is the average number of employees per department?
b) Which are the IDs of the employees with the highest salary?

c) What is the total amount of salaries paid by each department?
d) How many employees were hired in the current year?

Which indices would you create over the relation? For each index, indicate the type (hash or B+tree) and indicate if the index is clustered or non-clustered. Justify.