In this lab class we will approach the following topics:

1. Query Tuning

There are some 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. Through some examples, we will see how to write queries for better performance.

1.1. Avoiding DISTINCT

In most systems, DISTINCT will require a sort or other overhead, so it should be avoided. For example, consider the following query with three tables **R1**, **R2**, **R3** where **key** and **nonkey** indicate whether the column is the primary key or not:

```
SELECT DISTINCT R1.key
FROM R1, R2, R3
WHERE R1.nonkey = R2.key AND R2.nonkey = R3.key
```

There is no need for the keyword DISTINCT in this query because it won't produce any duplicates. To see this, consider the following:

- R1.key has no repeated values in R1 because it is the primary key. Duplicates could only arise by virtue of joining R1 with other tables.
- The join of R1 with R2 on the condition R1.nonkey = R2.key will produce at most one match, because R2.key is the primary key of R2.
- The join of R2 with R3 on the condition R2.nonkey = R3.key will produce at most one match, because R3.key is the primary key of R3.

In the query above, R1 is referred to as a **privileged table** (because its key appears in the SELECT), and R2 and R3 are referred to as **unprivileged tables** (because their keys do not appear in the SELECT). These unprivileged tables are being joined on equality conditions using their key fields. We say that R2 **reaches** R1 (because R2 is being joined with R1 via R2.key), and R3 **reaches** R2 (because R3 is being joined with R2 via R3.key). If R3 reaches R2, and R2 reaches R1, then by **transitivity** we conclude that R3 reaches R1, so **every unprivileged table reaches a privileged table**. In these circumstances, there will be no duplicates in the results, and therefore the use of DISTINCT is unnecessary.

1.2. Nested Queries

Many systems handle subqueries inefficiently, so it is preferable to rewrite those queries. For example, consider the following query:

```
SELECT R1.a
FROM R1
WHERE R1.b IN (SELECT R2.b
               FROM R2);
```

The inner query will retrieve all R2.b values from R2, when possibly only a subset of those values will match R1.b. A better approach would be to rewrite the query as:

```
SELECT R1.a
FROM R1, R2
WHERE R1.b = R2.b;
```

which will enable the system to use an index on either R1.b or R2.b, if such indexes exist.

Alternatively, we could write:

```
SELECT R1.a
FROM R1
WHERE EXISTS (SELECT *
              FROM R2
              WHERE R2.b = R1.b);
```

which is better than using IN, because IN will cause the inner query to be executed in its entirety, whereas EXISTS will stop as soon as the first result is found.

For nested queries that involve aggregates, such as:

```
SELECT R1.a
FROM R1
WHERE R1.b = (SELECT MAX(R2.b)
              FROM R2
              WHERE R2.c = R1.c);
```

it is often a good idea to first compute the aggregates and store them in a temporary table, and then execute a query over the temporary table. This avoids having to recompute the same aggregates over and over again. For example, the query above can be rewritten as:

```
SELECT MAX(R2.b) AS b, R2.c INTO Temp
FROM R2
GROUP BY R2.c;

SELECT R1.a
FROM R1, Temp
WHERE R1.b = Temp.b AND R1.c = Temp.c;
```

Note that the GROUP BY of the temporary table is done on the attribute that **correlates** the outer query with the inner query in the original nested query.

1.3. Temporary Tables

Temporary tables may help avoid ORDER BYs and scans when there are many queries with slightly different bind variables. For example, consider the following queries:

```
SELECT R1.a
FROM R1
WHERE R1.b >= 1000 AND R1.b < 2000
ORDER BY R1.c;

SELECT R1.a
FROM R1
WHERE R1.b >= 2000 AND R1.b < 3000
ORDER BY R1.c;

...

SELECT R1.a
FROM R1
WHERE R1.b >= 9000 AND R1.b < 10000
ORDER BY R1.c;
```

Each of these queries will scan through R1 and sort the results. A better approach would be to have a temporary table with all the records already sorted by R1.c:

```
SELECT R1.a, R1.b INTO Temp
FROM R1
WHERE R1.b >= 1000 AND R1.b < 10000
ORDER BY R1.c;
```

Then the queries could be performed without the ORDER BY:

```
SELECT Temp.a
FROM Temp
WHERE Temp.b >= 1000 AND Temp.b < 2000;

SELECT Temp.a
FROM Temp
WHERE Temp.b >= 2000 AND Temp.b < 3000;

...

SELECT Temp.a
FROM Temp
WHERE Temp.b >= 9000 AND Temp.b < 10000;
```

There are also some cases when temporary tables can hurt performance. For example, consider the following query:

```
SELECT R1.a, R1.b, R1.c INTO Temp
FROM R1
WHERE R1.b > 1000;

SELECT Temp.a
FROM Temp
WHERE Temp.c = 'xyz';
```

There are two selection conditions: b > 1000 and c = 'xyz'. If there is an index on R1.c, then in principle this index could be used to quickly locate the rows where R1.c = 'xyz'. However, the query is using Temp rather than R1, so the system will not have the opportunity to use the index. A more efficient solution would be:

```
SELECT R1.a
FROM R1
WHERE R1.b > 1000 AND R1.c = 'xyz';
```

1.4. Join on clustered indexes

When two are to be joined, it is a good idea to express the join condition based on clustering indexes. For example:

```
SELECT R1.b
FROM R1, R2
WHERE R1.a = R2.a;
```

If there is a clustered index on R1.a and another clustered index on R2.a, then the system can use a merge join, which will be faster than other join algorithms.

If the join condition cannot be based on clustered indexes, then one should prefer, if possible, a join condition based on numerical equality rather than string equality.

1.5. HAVING vs. WHERE

Do not use HAVING when WHERE is enough. For example, consider the following query:

```
SELECT R1.a, AVG(R1.b)
FROM R1
GROUP BY R1.a
HAVING R1.a = 'xyz';
```

A better approach is to first find the relevant records and only then compute the average:

```
SELECT R1.a, AVG(R1.b)
FROM R1
WHERE R1.a = 'xyz'
GROUP BY R1.a;
```

1.6. OR vs. UNION

Some systems do not use indexes when different expressions are connected by the OR keyword. For example, consider the following query:

```
SELECT R1.a
FROM R1
WHERE R1.b = 1000 OR R1.c = 'xyz';
```

If there are indexes on R1.b and R1.c but the query plan does not use them, then consider using a union:

```
(SELECT R1.a
 FROM R1
 WHERE R1.b = 1000)
UNION
(SELECT R1.a
 FROM R1
 WHERE R1.c = 'xyz');
```

In addition, consider using UNION ALL instead of UNION, because UNION eliminates duplicates, which requires additional effort. If you can tolerate duplicates in the results, it is better to use UNION ALL, which will be faster than UNION.

2. Experiments and Exercises

We will use SQL Server Management Studio and the AdventureWorks database to check the impact of query tuning on the execution of several queries written in different ways.

2.1. Practical Experiments

2.1.1. - We want to build a phone book of all employees, with their name, phone number and type of phone (whether it's work, home or cell). For this purpose, we write the following query:

```
SELECT DISTINCT a.BusinessEntityID,
                a.FirstName,
                a.LastName,
                b.PhoneNumber,
                c.Name
FROM Person.Person AS a,
     Person.PersonPhone AS b,
     Person.PhoneNumberType AS c
WHERE a.BusinessEntityID = b.BusinessEntityID
  AND b.PhoneNumberTypeID = c.PhoneNumberTypeID;
```

a) Execute the query above to check that it's working.

b) Check the execution plan for this query. How is DISTINCT being implemented?

c) Hover the mouse over the root of the tree (SELECT node) and take note of the **Estimated Subtree Cost**.

d) Remove DISTINCT from the query and execute it again.

e) Check the execution plan for the query without DISTINCT. What is the difference to the previous plan?

f) Check the **Estimated Subtree Cost** for this new plan. Is there a performance gain?

2.1.2. – We want to find the names of all employees who work in the sales department. For this purpose, we start with the following query:

```
SELECT c.BusinessEntityID
FROM HumanResources.EmployeeDepartmentHistory AS c,
     HumanResources.Department AS d
WHERE c.DepartmentID = d.DepartmentID
  AND c.EndDate IS NULL
  AND d.Name = 'Sales';
```

a) Execute the query above to check that it's working. It identifies the employees who work in the sales department.

b) However, what we want is their names. So now execute the following nested query:

```
SELECT a.FirstName, a.LastName
FROM Person.Person AS a, HumanResources.Employee AS b
WHERE a.BusinessEntityID = b.BusinessEntityID
  AND b.BusinessEntityID IN (
          SELECT c.BusinessEntityID
          FROM HumanResources.EmployeeDepartmentHistory AS c,
               HumanResources.Department AS d
          WHERE c.DepartmentID = d.DepartmentID
            AND c.EndDate IS NULL
            AND d.Name = 'Sales');
```

The outer query gets the name of all employees. The inner query gets the employees from the sales department. The two queries are connected by IN, which checks if a given employee is in the set of employees from the sales department.

c) Check the execution plan for this query. Hover the mouse over the root of the tree and take note of the **Estimated Subtree Cost**.

d) The same query can be written by joining all the tables at once, as follows:

```
SELECT a.FirstName, a.LastName
FROM Person.Person AS a,
     HumanResources.Employee AS b,
     HumanResources.EmployeeDepartmentHistory AS c,
     HumanResources.Department AS d
WHERE a.BusinessEntityID = b.BusinessEntityID
  AND b.BusinessEntityID = c.BusinessEntityID
  AND c.DepartmentID = d.DepartmentID
  AND c.EndDate IS NULL
  AND d.Name = 'Sales';
```

Run this query and check that it produces the same results.

e) Check the execution plan, and take note of the **Estimated Subtree Cost**.

f) To try to make things faster, we could create a materialized view with the employees from the sales department:

```
CREATE VIEW SalesEmployees WITH SCHEMABINDING AS
SELECT c.BusinessEntityID
FROM HumanResources.EmployeeDepartmentHistory AS c,
     HumanResources.Department AS d
WHERE c.DepartmentID = d.DepartmentID
  AND c.EndDate IS NULL
  AND d.Name = 'Sales';
```

Run the statement above to create the view.

g) In SQL Server, we materialize a view by creating an index over it (hence the designation of *indexed view*). Execute the following command to create such index:

```
CREATE UNIQUE CLUSTERED INDEX SalesEmployeesIdx
ON SalesEmployees(BusinessEntityID);
```

h) Now that the view is materialized, we rewrite the nested query to use the view:

```
SELECT a.FirstName, a.LastName
FROM Person.Person AS a, HumanResources.Employee AS b
WHERE a.BusinessEntityID = b.BusinessEntityID
  AND b.BusinessEntityID IN (SELECT BusinessEntityID
                             FROM SalesEmployees);
```

Execute the query above and check that it gives the same results as before.

i) Check the execution plan and take note of the **Estimated Subtree Cost**.

j) In conclusion, what is the fastest way to get the names of employees from the sales department?

i. nested query over base tables
ii. nested query with materialized view
iii. query with joins over the base tables

2.1.3. – We want to identify all the sales orders coming from France. These include:

- Sales orders placed in France (by French customers or other customers visiting France)
- Sales orders placed by French customers (in France or in other countries).

a) To obtain the sales orders placed in France, run the following query:

```
SELECT a.SalesOrderID
FROM Sales.SalesOrderHeader AS a,
     Sales.SalesTerritory AS b
WHERE a.TerritoryID = b.TerritoryID
  AND b.Name = 'France';
```

b) Check the execution plan and confirm that it is using an index (*index seek*) on Name.

c) To obtain the sales orders placed by French customers, run the following query:

```
SELECT a.SalesOrderID
FROM Sales.SalesOrderHeader AS a,
     Sales.Customer AS c,
     Sales.SalesTerritory AS d
WHERE a.CustomerID = c.CustomerID
  AND c.TerritoryID = d.TerritoryID
  AND d.Name = 'France';
```

d) Again, check the execution plan and confirm that it is using an index (*index seek*) on Name.

e) The two queries can be combined into a single query as follows:

```
SELECT a.SalesOrderID
FROM Sales.SalesOrderHeader AS a,
     Sales.SalesTerritory AS b,
     Sales.Customer AS c,
     Sales.SalesTerritory AS d
WHERE a.TerritoryID = b.TerritoryID
  AND a.CustomerID = c.CustomerID
  AND c.TerritoryID = d.TerritoryID
  AND (b.Name = 'France' OR d.Name = 'France');
```

Note that this query uses an OR, which some systems find difficult to optimize.

f) Check the execution plan for this query and confirm that the system is not using the index on Name. Take note of the **Estimated Subtree Cost** for this execution plan.

g) A better way to combine the queries is through a UNION:

```
(SELECT a.SalesOrderID
 FROM Sales.SalesOrderHeader AS a,
      Sales.SalesTerritory AS b
 WHERE a.TerritoryID = b.TerritoryID
   AND b.Name = 'France')
UNION
(SELECT a.SalesOrderID
 FROM Sales.SalesOrderHeader AS a,
      Sales.Customer AS c,
      Sales.SalesTerritory AS d
 WHERE a.CustomerID = c.CustomerID
   AND c.TerritoryID = d.TerritoryID
   AND d.Name = 'France');
```

h) Check the execution plan for this UNION and confirm that the system is using the index (twice) on Name. Take note of the **Estimated Subtree Cost** for this execution plan and compare it to the previous plan.

i) Finally, the query can be made even faster by replacing UNION with UNION ALL. (What is the effect of this change?) Check the new execution plan and the **Estimated Subtree Cost**.

2.2. Exercise

Consider again the following query:

```
SELECT DISTINCT a.BusinessEntityID,
                a.FirstName,
                a.LastName,
                b.PhoneNumber,
                c.Name
FROM Person.Person AS a,
     Person.PersonPhone AS b,
     Person.PhoneNumberType AS c
WHERE a.BusinessEntityID = b.BusinessEntityID
  AND b.PhoneNumberTypeID = c.PhoneNumberTypeID;
```

a) Find the primary keys for these tables in the database.

b) Is DISTINCT really necessary in this query? Use the concepts of **privileged** and **unprivileged** tables to find the answer.

c) If DISTINCT is necessary, give an example of how duplicates could occur; if DISTINCT is unnecessary, give an example of how duplicates could not occur.