Session 5

Mohamed Emary

December 15, 2024

All the examples are on the ‘ITI‘ Database we have worked on before.

1 GROUP BY and HAVING

1.1 GROUP BY

In the `Instructor` table, to find the minimum salary between all the instructors, we can use:

```
SELECT
  MIN(Salary)
FROM
  Instructor;
```

But what if we want to find the minimum salary for each department? We can use the `GROUP BY` clause to group the rows based on the `Dept_name` and then apply the `MIN` function to each group.

```
SELECT
  MIN(Salary) "Min Salary Per Dept",
  Dept_Id
FROM
  Instructor
GROUP BY
  Dept_Id;
```

You may see a `NULL` value in the min salary column. This appears if there is one or more instructors in the department all having `NULL` salary. If there is at least one instructor with a salary, the `NULL` value will not appear (`NULL` is not a value and it only appears when there is no other value).

To avoid having `NULL` values in the result, we can use the `WHERE` clause to filter out the `NULL` values.

```
SELECT
  MIN(Salary) "Min Salary Per Dept",
  Dept_Id
```

```
FROM
  Instructor
WHERE
  Salary IS NOT NULL
GROUP BY
  Dept_Id;
```

The `GROUP BY` clause make the aggregation function (`MIN` in this case) apply to each group instead of the whole table.

Another way to apply the operation above is to use `PARTITION BY` in the `OVER` clause. This way, we can apply the aggregation function to each group without using the `GROUP BY` clause.

```
SELECT
  Dept_Id,
  MIN(Salary) OVER (
    PARTITION BY
      Dept_Id
  ) "Min Salary Per Dept"
FROM
  Instructor
WHERE
  Salary IS NOT NULL;
```

When using group by use it on a column that it's values are the same in multiple rows. Using it on a unique column like the primary key will be useless:

```
SELECT
  St_Id,
  COUNT(*)
FROM
  Student
GROUP BY
  St_Id;
```

You also can't group by `*`. For this to work you need to have at least two rows with the same values in all columns which shouldn't happen from the beginning since the primary key is unique. It's also not possible in code, you will get an error:

```
SELECT
  St_Id,
  COUNT(*)
FROM
  Student
GROUP BY
  * -- Error
```

To count the number of students in each department, the `COUNT` aggregate function would work on each group from the `GROUP BY` clause.

```
SELECT
  Dept_Id,
  COUNT(*) AS 'Number of Dep Students'
FROM
  Student
```

```sql
GROUP BY
  Dept_Id;

-- To ignore the NULL values in the Dept_Id column
SELECT
  Dept_Id,
  COUNT(*) AS 'Number of Dep Students'
FROM
  Student
WHERE
  Dept_Id IS NOT NULL
```

To count the number of students in the whole table:

```sql
SELECT
  -- Here the COUNT function will work on the whole table
  -- since there is no GROUP BY clause
  COUNT(*) AS 'Total Number of Students'
FROM
  Student;
```

Anything being selected next to the aggregate function and it's not an aggregate function should be in the `GROUP BY`:

```sql
SELECT
  St_Lname,
  COUNT(*)
FROM
  Student
GROUP BY
  St_Lname;

-- Another example
SELECT
  St_Fname,
  MAX(St_Age)
FROM
  Student
GROUP BY
  St_Fname;
```

Grouping by multiple columns have a similar idea to cross join:

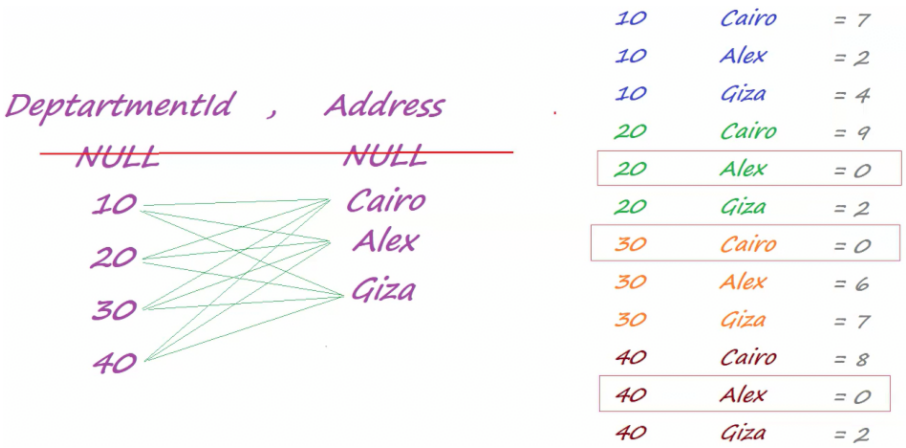

Figure 1: Grouping by multiple columns

```sql
SELECT
  Dept_Id,
  St_Address,
  COUNT(*) AS 'Number of Students'
FROM
  Student
WHERE
  Dept_Id IS NOT NULL
  AND St_Address IS NOT NULL
  -- Grouping here is done by address first (the second column)
  -- then by dept_id (the first column)
GROUP BY
  Dept_Id,
  St_Address;
```

1.2 HAVING

`SELECT` and `WHERE` work on table *record by record* while `COUNT` aggregate function works on the whole table, that is why you can't use `AND COUNT(*) > 2` in the `WHERE` clause in the statement below

If you want to get the number of students in each department that have more than 2 students:

```sql
-- This will not work
SELECT
  Dept_Id,
  COUNT(*) AS 'Number of Students'
FROM
  Student
WHERE
  Dept_Id IS NOT NULL
  AND COUNT(*) > 2
GROUP BY
  Dept_Id
```

To fix that you will need to use `HAVING` keyword, `HAVING` works on the groups created by the `GROUP BY` clause `HAVING` is mostly used with aggregate functions.

```sql
SELECT
  Dept_Id,
  COUNT(*) AS 'Number of Dep Students'
FROM
  Student
WHERE
  Dept_Id IS NOT NULL
GROUP BY
  Dept_Id
HAVING
  COUNT(*) > 2;
```

In general, we use `HAVING` without `GROUP BY` if we want to apply a condition on an aggregate function and we are not selecting that aggregate function in the `SELECT` clause **because we can't use aggregate functions in the `WHERE` clause**.

With `HAVING` we always have a condition with aggregation that condition works on the groups created by the `GROUP BY` clause or on the whole table if there is no `GROUP BY` clause and we are using an aggregate function in the `SELECT` clause.

Some general rules:

1. Aggregate functions like `COUNT` work on the whole table.
2. `WHERE` works on the table record by record.
3. `HAVING` works on the groups created by the `GROUP BY` clause.

Example of using `HAVING` without `GROUP BY`, we want to get the sum of all instructors salaries if there is more than 10 instructors in the table:

```sql
SELECT
  SUM(Salary) AS 'Total Salaries'
FROM
  Instructor
HAVING
  COUNT(*) > 10;
```

If we want to get the sum of salaries for each department, we can use one of the two statements below.

The two statement below is similar to each other but the second one uses join. The performance in the second one is worse than the first since we are applying operations on two tables

Generally this is not a good case for using join. Join is used when we want to get data from two tables

```sql
SELECT
  Dept_Id,
  SUM(Salary) AS 'SumOfSalaries'
FROM
  Instructor
WHERE
  Dept_Id IS NOT NULL
GROUP BY
  Dept_Id;

```

```sql
-- Using join
SELECT
  I.Dept_Id,
  SUM(I.Salary) AS 'SumOfSalaries'
FROM
  Instructor I,
  Department D
WHERE
  I.Dept_Id = D.Dept_Id
GROUP BY
  I.Dept_Id;
```

Here each department have a different name, and to show the department name here we still need to group by `Dept_Name` to make the query work:

```sql
SELECT
  I.Dept_Id,
  D.Dept_Name,
  SUM(I.Salary) AS 'SumOfSalaries'
FROM
  Instructor I,
  Department D
WHERE
  I.Dept_Id = D.Dept_Id
GROUP BY
  I.Dept_Id,
  D.Dept_Name;
```

To select the students who act as supervisors and the number of students they supervise, we had to group by both `Supr.St_Fname`, `Supr.St_Id` since we are selecting both of them in the `SELECT` clause If we are selecting only one of them we can group by only that column.

> ***NOTE:*** If you group by only the `St_Fname` column, and we have two supervisors with the same `St_Fname`, the query will group them together as if they were the same person and show the total number of students they supervise together.

```sql
SELECT
  Supr.St_Fname 'Supervisor',
  Supr.St_Id 'Supervisor ID',
  COUNT(*) 'No of Students'
FROM
  Student Stud,
  Student Supr
WHERE
  Supr.St_Id = Stud.St_super
GROUP BY
  Supr.St_Id,
  Supr.St_Fname;
```

2 Subqueries

2.1 Introduction

Subqueries involve an inner query (the subquery) nested within an outer query. The subquery executes first, and its results are used by the outer query. While subqueries can be useful, they are generally **not recommended for performance reasons** except in some special cases where part of a query (usually contains an aggregate function) needs to be run separately and return output for the outer query. This is because using subqueries can negatively impact performance as it involves executing two queries instead of one. Sometimes you can do something with subqueries but there is a better way to do it.

The output of the inner query serves as input for the outer query.

Query execution phases

SQL Query Processing

2.2 Example 1: Students Older Than Average

Problem: Get students whose age is greater than the average age of all students.

2.2.1 Incorrect Attempt

Trying to directly use `AVG` in the `WHERE` clause will not work, because `AVG` is an aggregate function and works on the whole table, whereas the `WHERE` clause works record by record.

```sql
SELECT
  St_Id,
  St_Fname,
  St_Age
FROM
  Student
WHERE
  St_Age > AVG(St_Age); -- This will result in an error
```

2.2.2 Solution with Subquery:

```sql
SELECT
  St_Id,
  St_Fname,
  St_Age
FROM
  Student
WHERE
  St_Age > (
    SELECT
      AVG(St_Age)
    FROM
      Student
  );
```

2.3 Example 2: Student IDs with Total Student Count

Problem: Get each student's ID and the total number of students (e.g., 1/20, 2/20, 3/20,...).

2.3.1 Incorrect Attempt

Selecting `St_Id` alongside an aggregate function without grouping by `St_Id` will cause an error since it is not an aggregate function.

Note:

Anything selected next to an aggregate function that is not an aggregate function should be in the `GROUP BY` clause.

```sql
SELECT
  St_Id,
  COUNT(*)
FROM
  Student; -- This will result in an error
```

2.3.2 Solution with Subquery:

```sql
SELECT
  St_Id,
  (
    SELECT
      COUNT(*)
    FROM
      Student
  ) AS "Total Students"
FROM
  Student;
```

2.4 Example 3: Departments with Students

Problem: Get the names of departments that have students.

2.4.1 Using Join (Recommended)

This is the most efficient method.

As we knew before that the relation between student and department is one to many as each student belongs to one department and each department has many students

When to use `JOIN`:

`JOIN` is used when you want to select data from multiple tables that have a relation between them or when you want to get data from one table based on data from another table.

```sql
SELECT
  Dept_Name
```

```
FROM
  Student S
JOIN
  Department D
ON
  S.Dept_Id = D.Dept_Id;
```

Here we are selecting from only one column but we still needed to use `JOIN` because we need to get departments that have students

This might produce duplicate department names if multiple students belong to the same department and we can also select `S.St_Fname` to see the students names. To get only unique values we can use the `DISTINCT` keyword:

```
SELECT DISTINCT
  Dept_Name
FROM
  Student S
JOIN
  Department D
ON
  S.Dept_Id = D.Dept_Id;
```

2.4.2 Using Subquery (Not Recommended)

If you have no other choice but to use subqueries: use the `IN` operator to compare the column with the output of the subquery since the subquery returns an array of values and you can't use the `=` operator for that.

```
SELECT
  Dept_Name
FROM
  Department
WHERE
  Dept_Id IN (
    SELECT DISTINCT
      Dept_Id
    FROM
      Student
    WHERE
      Dept_Id IS NOT NULL
  );
```

SQL Server's query optimizer often transforms subqueries into joins internally for optimization. You can use SQL Server Profiler to see the performance difference between the two queries.

2.5 Example 4: Deleting Grades of Students in `Mansoura`

Problem: Delete the grades of students who live in `Mansoura`.

This is how the select statement would look:

```
SELECT
  *
```

```
FROM
  Stud_Course SC,
  Student S
WHERE
  SC.St_Id = S.St_Id
  AND S.St_Address = 'Mansoura';
```

2.5.1 Using Subquery

```
DELETE FROM Stud_Course
WHERE
  St_Id IN (
    SELECT
      St_Id
    FROM
      Student
    WHERE
      St_Address = 'Mansoura'
  );
```

2.5.2 Using Join

When using `DELETE` with `JOIN` we should add the table alias to specify from which table we want to delete the records after the `DELETE` keyword. If we add `SC` it will delete the records from the `Stud_Course` table (the student grades), and if we add `S` it will delete the records from the `Student` table (the student himself).

```
DELETE SC
FROM
  Stud_Course SC
JOIN
  Student S
ON
  SC.St_Id = S.St_Id
WHERE
  S.St_Address = 'Mansoura';
```

3 `TOP` Keyword

`TOP` is a SQL keyword (not a function) used to select the top n rows from a table. It accepts an expression specifying the number of rows to select. You typically use it after ordering the records to get the top records based on something.

After `TOP` we specify the columns we want to select, or we can just use `*` to select all columns.

3.1 Basic Usage

To select the first 2 students:

```
SELECT
  TOP (2) *
```

```
FROM
  Student;
```

Since data in the table is ordered by primary key by default, the query above will return the first two students based on the primary key.

To select the top 5 students' first names and ages:

```
SELECT
  TOP (5) St_Fname,
  St_Age
FROM
  Student;
```

Note:

`TOP` is a keyword and not a function. This means it can be used with column names without issue, unlike aggregate functions, which require a `GROUP BY` clause when used with other columns.

3.2 Selecting All Except the Last *n* Rows

To select all students except the last five, we get the count of all students and subtract 5 from it:

```
SELECT
  TOP (
    (
      SELECT
        COUNT(*)
      FROM
        Student
    ) - 5
  ) *
FROM
  Student;
```

3.3 Selecting the Last *n* Rows

To get the last 5 students, order by a column then use `TOP`:

Since data is ordered by the primary key by default, we can order by the primary key in descending order to get the last 5 students.

```
SELECT
  TOP (5) *
FROM
  Student
ORDER BY
  St_Id DESC;
```

3.4 Real-World Example: Top 3 Instructors by Salary

We knew in the last sessions that we can use the `MAX` aggregate function to get the maximum salary but it will return only the number

```
SELECT
  MAX(Salary) MaxSalary
FROM
  Instructor;
```

To get the names of the 3 instructors with the maximum salary using `TOP`:

```
SELECT
  TOP (3) Ins_Name,
  Salary
FROM
  Instructor
ORDER BY
  Salary DESC;
```

3.5 Find the Second Highest Salary

3.5.1 Without Using `TOP`

Using a subquery to get the second-highest salary:

```
SELECT
  MAX(Salary)
FROM
  Instructor
WHERE
  Salary != ( -- != is the same as <> in SQL
    SELECT
      MAX(Salary)
    FROM
      Instructor
  );
```

3.5.2 Using `TOP`

Another way to achieve the same result is to select the top two salaries, then select the lowest from that result set.

> Note:
>
> You must give the resulting table you get from the sub query an alias name or you will get an error (You can use `AS` or ignore it and write the alias name directly)

```
SELECT
  TOP (1) *
FROM
  (
    SELECT
      TOP (2) Ins_Name,
```

```
      Salary
    FROM
      Instructor
    ORDER BY
      Salary DESC
  ) AS TopTwoSal
ORDER BY
  Salary;
```

An alternative approach is to select the instructor with the highest salary from the table excluding the instructor(s) with the highest salary.

```
SELECT
  TOP (1) Ins_Name,
  Salary
FROM
  Instructor
WHERE
  Salary != (
    SELECT
      MAX(Salary)
    FROM
      Instructor
  )
ORDER BY
  Salary DESC;
```

3.6 TOP with WITH TIES

`WITH TIES` is used with `ORDER BY` to include records with the same value as the last record in the top set.

```
SELECT
  TOP (5)
WITH
  TIES St_Age
FROM
  Student
ORDER BY
  St_Age DESC;
```

4 Random Selection

4.1 NEWID() Function

`NEWID()` is a built-in function in SQL Server that generates a new Globally Unique Identifier (GUID). Each time it's run, it returns a different 32-character string divided into 5 groups separated by hyphens `-` e.g., `78ff8575-bc53-4f87-9079-94e225372658`.

```
SELECT
  NEWID();
```

4.2 Random Record Selection

With `NEWID()`, you can do the following:

1. You can use `NEWID()` to perform random selections
2. It can also be used as a primary key value in a table
3. Set a dynamic default value for a column

```
SELECT
  TOP (1) *
FROM
  Student
ORDER BY
  NEWID();
```

This will select a different student each time you run the query.

5 Ranking Functions

Ranking functions assign ranks to rows within a result set, they take no arguments and work on the table record by record.

The difference between the three functions is in how they handle rows with the same value. Choosing which function to use depends on the business requirements.

Ranking functions are not aggregate functions, so you can select other columns alongside them without using `GROUP BY`.

5.1 Types of Ranking Functions

1. `ROW_NUMBER()`: Assigns a unique sequential integer to each row, starting at 1.
2. `RANK()`: Assigns a rank to each row, with gaps for tied ranks.
3. `DENSE_RANK()`: Assigns a rank to each row, without gaps for tied ranks.

This image will help illustrate the difference between the three functions:

In the image we are sorting rows in a descending order based on the `salary` column, then we are using the three ranking functions to rank the rows based on the `salary`.

Eid	Ename	Salary	Address	did	Row_Number() over (Order By Salary Desc)	Dense_Rank() over (Order By Salary Desc)	Rank() over (Order By Salary Desc)
1	ahmed	10000	cairo	10	1	1	1
2	ali	10000	cairo	10	2	1	1
3	eman	9000	cairo	10	3	2	3
4	khalid	9000	alex	10	4	2	3
5	sameh	8000	alex	10	5	3	5
6	yousef	8000	alex	10	6	3	5
7	alaa	7000	alex	20	7	4	7
8	mohamed	7000	alex	20	8	4	7
9	reem	6000	cairo	20	9	5	9
10	ola	6000	cairo	20	10	5	9

Figure 2: `ROW_NUMBER` Vs `DENSE_RANK` Vs `RANK`

5.2 Example of Ranking Functions

```sql
SELECT
  Ins_Id,
  Ins_Name,
  Salary,
  ROW_NUMBER() OVER (
    ORDER BY
      Salary DESC
  ) AS RN,
  DENSE_RANK() OVER (
    ORDER BY
      Salary DESC
  ) AS DR,
  RANK() OVER (
    ORDER BY
      Salary DESC
  ) AS R
FROM
  Instructor;
```

If ranking functions find two similar values they will order them based on the order of the rows in the table (the primary key) and we can use another column to order them so if we have two similar values we can order them based on that column.

```sql
SELECT
  ROW_NUMBER() OVER (
    ORDER BY
      Points DESC,
      NumberOfGoals DESC
  ) AS RN,
  DENSE_RANK() OVER (
    ORDER BY
      Points DESC,
      NumberOfGoals DESC
  ) AS DR,
  RANK() OVER (
    ORDER BY
      Points DESC,
      NumberOfGoals DESC
  ) AS R
FROM
  Instructor;
```

5.3 Example of TOP and Ranking Functions

To get the 2 oldest students in the `Student` table:

```sql
SELECT
  TOP 2 St_Age,
  St_Fname,
  St_Id
```

```
FROM
  Student
ORDER BY
  St_Age DESC;
```

6 Random Examples

6.1 Ranking with Subqueries and WITH Clause

6.1.1 Using ROW_NUMBER() in a Subquery

Here, instead of using TOP, we use ROW_NUMBER() in a subquery to get the students with the highest age. We then select the students with rank 1 and 2 from the subquery.

Important: We must give the table outputted by the subquery an alias name. We cannot apply the ROW_NUMBER() function directly in the WHERE clause without using a subquery.

```
SELECT
  *
FROM
  (
    SELECT
      St_Age,
      St_Fname,
      St_Id,
      ROW_NUMBER() OVER (
        ORDER BY
          St_Age DESC
      ) AS RN
    FROM
      Student
  ) AS Ages
WHERE
  Ages.RN IN (1, 2);
```

6.1.2 Using WITH ... AS Clause

Another way to write the query above is to use WITH ... AS, which defines a Common Table Expression (CTE).

```
WITH
  Ages AS (
    SELECT
      St_Age,
      St_Fname,
      St_Id,
      ROW_NUMBER() OVER (
        ORDER BY
          St_Age DESC
      ) AS RN
    FROM
      Student
```

```
  )
SELECT
  *
FROM
  Ages
WHERE
  Ages.RN IN (1, 2);
```

6.1.3 Comparison

Out of the three ways we wrote the query above (using subquery, using `WITH ... AS`, and using `TOP`), the best way is to use `TOP` as there is no subquery which makes it more performant. Subqueries increase the time it takes to execute a query as we do these steps two times one for the subquery and one for the main query.

> Query Execution Steps From MS Docs:
> The basic steps that SQL Server uses to process a single `SELECT` statement include the following:
>
> 1. The parser scans the `SELECT` statement and breaks it into logical units such as keywords, expressions, operators, and identifiers.
> 2. A query tree, sometimes referred to as a sequence tree, is built describing the logical steps needed to transform the source data into the format required by the result set.
> 3. The Query Optimizer analyzes different ways the source tables can be accessed. It then selects the series of steps that return the results fastest while using fewer resources. The query tree is updated to record this exact series of steps. The final, optimized version of the query tree is called the execution plan.
> 4. The relational engine starts executing the execution plan. As the steps that require data from the base tables are processed, the relational engine requests that the storage engine pass up data from the rowsets requested from the relational engine.
> 5. The relational engine processes the data returned from the storage engine into the format defined for the result set and returns the result set to the client.

6.2 Finding the n_{th} Youngest Student

6.2.1 Using `TOP`

To get the 5_{th} youngest student:

```
SELECT
  TOP 1 *
FROM
  (
    SELECT
      TOP 5 St_Fname,
      St_Age
    FROM
      Student
    WHERE
      St_Age IS NOT NULL
    ORDER BY
      St_Age
```

```
  ) AS YoungestFive
ORDER BY
  St_Age DESC;
```

6.2.2 Using `ROW_NUMBER()`

To get the 5_{th} youngest student using `ROW_NUMBER()`:

```
SELECT
  *
FROM
  (
    SELECT
      St_Fname,
      St_Age,
      ROW_NUMBER() OVER (
        ORDER BY
          St_Age
      ) AS RN
    FROM
      Student
    WHERE
      St_Age IS NOT NULL
  ) AS YoungestFive
WHERE
  YoungestFive.RN = 5;
```

> *Note:* If you have two queries and you want to know which one performs better you can use the benchmark tool in SQL Server Management Studio.

6.3 Youngest Student in Each Department

6.3.1 Using `GROUP BY`

To get the minimum age for each department using `GROUP BY`:

```
SELECT
  Dept_Id,
  MIN(St_Age) 'Min Age'
FROM
  Student
WHERE
  Dept_Id IS NOT NULL
GROUP BY
  Dept_Id;
```

Important Note:

When selecting an aggregate function, we are dealing with the table as groups (resulting from `GROUP BY`) or one group (the table itself if we are not using `GROUP BY` but selecting an aggregate function).

That is why if we want to get the sum of salaries when the number of instructors is greater than 2, we should use the `HAVING` clause instead of the `WHERE` clause. This is because our table is considered as one group (group of all instructors).

This is the reason you cannot select a column that is not in the `GROUP BY` clause when selecting an aggregate function. You are not selecting from the table but from the group, so you can only select group keys. If you want to select another column, you should add it to the `GROUP BY` clause.

6.4 Example: Total Salary Of All Instructors

6.4.1 Using `HAVING`

To get the total salary of all instructors if the number of instructors is greater than 10, using `HAVING`:

```
SELECT
  SUM(Salary) 'Total Salary'
FROM
  Instructor
HAVING
  COUNT(*) > 10;
```

6.5 Using `PARTITION BY`

`PARTITION BY` is used with aggregate functions when we want to get an aggregate value and still want to work on the table as a table not as group/s so we can use an aggregate function and still select columns from the table.

Using `PARTITION BY` is like running the aggregate function on a separate thread and running the `SELECT` statement on another thread.

To get the maximum salary for each department, along with instructor details:

```
SELECT
  Ins_Id,
  Ins_Name,
  Dept_Id,
  MAX(Salary) OVER (
    PARTITION BY
      Dept_Id
    ORDER BY
      Salary DESC
  ) AS 'Max Dep Salary'
FROM
  Instructor
WHERE
  Salary IS NOT NULL;
```

6.5.1 `GROUP BY` Vs `PARTITION BY`

To show the difference between the two, when using `GROUP BY` we will only get the max salary for each department, but when using `PARTITION BY` we are getting all instructors with the max salary for their department.

6.5.2 Using `GROUP BY`

```
SELECT
  Dept_Id,
  MAX(Salary) 'Max Dep Salary'
FROM
  Instructor
WHERE
  Salary IS NOT NULL
GROUP BY
  Dept_Id;
```

6.5.3 Using `PARTITION BY`

```
SELECT
  *
FROM
  (
    SELECT
      Dept_Id,
      Ins_Name,
      ROW_NUMBER() OVER (
        PARTITION BY
          dept_id
        ORDER BY
          Salary DESC
      ) AS SalaryRank
    FROM
      Instructor
    WHERE
      Salary IS NOT NULL
  ) AS InstructorSalaries
WHERE
  SalaryRank = 1;
```

6.6 Youngest Student in Each Department

Back to youngest student in each department example. We can use either `PARTITION BY` or `TOP` to get the youngest student in each department.

6.6.1 Using `PARTITION BY`

```
SELECT
  *
FROM
  (
```

```
    SELECT
      St_Fname,
      Dept_Id,
      St_Age,
      ROW_NUMBER() OVER (
        PARTITION BY
          Dept_Id
        ORDER BY
          St_Age
      ) AS AgeRank
    FROM
      Student
    WHERE
      St_Age IS NOT NULL
      AND Dept_Id IS NOT NULL
      -- ORDER BY -- we can't use order by here
      --   Dept_Id
  ) AS AgeRankTable
WHERE
  AgeRank = 1;
```

We can also use `TOP` for the query above.

7 `NTILE` Function

`NTILE` is a window function used to divide rows into a specified number of groups (tiles) based on an ordering. It is often used for pagination or grouping data.

7.1 Basic Usage

To divide instructors into 3 tiles based on salary:

Since we have 15 rows (instructors), we will have 5 rows in each tile.

```
SELECT
  Ins_Id,
  Ins_Name,
  Dept_Id,
  NTILE(3) OVER (
    ORDER BY
      Salary
  )
FROM
  Instructor;
```

If the number of rows is not divisible by the number of tiles, the earlier tiles will have more rows, and the tile with the least number of rows will always be the last tile.

To divide instructors into 4 tiles based on salary:

Each tile will have 4 rows except for the last tile, which will have 3 rows.

```sql
SELECT
  Ins_Id,
  Ins_Name,
  Dept_Id,
  NTILE(4) OVER (
    ORDER BY
      Salary
  ) SalaryTile
FROM
  Instructor;
```

7.2 Selecting Data from Specific Tiles

To select the highest paid instructors (those in the first tile when divided into 3 tiles):

```sql
SELECT
  *
FROM
  (
    SELECT
      Ins_Id,
      Ins_Name,
      Dept_Id,
      NTILE(3) OVER (
        ORDER BY
          salary DESC
      ) salaryTile
    FROM
      Instructor
  ) AS HighestPaidInstructors
WHERE
  salaryTile = 1;
```

7.3 Use Cases

`NTILE` can be used for:

- Pagination (dividing results into pages as in amazon results).
- Selecting top, middle, or bottom groups of data based on a certain criteria. For example dividing products into high, medium, and low price categories.

For each of the cases above we can use:

- A subquery with a `WHERE` condition to select the desired tile.
- `TOP` and select:
  - Top 1000 if we want the first 1000 products
  - Sort the products in descending order and select top 1000 if we want the last 1000 products
  - If we want to get for example from 3000 to 4000 products (tile number 4) we can use top 4000 and then reverse the order and select top 1000.

> One of the good resources that you can use to study SQL is JavaTPoint and SQLServerTutorial. You can also refer to the official documentation of SQL Server.

8 OFFSET and FETCH

- `OFFSET` is used to skip a specified number of rows from the beginning of a result set.
- `FETCH` is used after `OFFSET` to select a specified number of rows from the result set.

8.1 Example

To select distinct `Top_Id` values from the `Course` table, skipping the first row, and fetching the next 2 rows:

```sql
SELECT DISTINCT
  Top_Id
FROM
  Course
ORDER BY
  Top_Id
OFFSET
  1 ROWS
FETCH NEXT
  2 ROWS ONLY;
```

9 SQL Query Execution Order

The SQL query execution order is as follows:

1. `FROM`: Specifies the tables involved.
2. `JOIN`: Combines rows from multiple tables.
3. `ON`: Specifies join conditions.
4. `WHERE`: Filters rows based on a condition.
5. `GROUP BY`: Groups rows based on a column.
6. `HAVING`: Filters groups based on a condition.
7. `SELECT`: Selects columns to be returned.
8. `ORDER BY`: Sorts the result set.
9. `TOP`: Limits the number of rows returned.

If there is a subquery, the subquery is executed before the outer query with the same order, and its output is used in the outer query.

Important Note:

> When reading or analyzing SQL queries, especially in interviews, follow the execution order listed above instead of the written order.
>
> Knowing the query execution order helps with tracing and optimizing queries.

9.1 Example of Execution Order

The `WHERE` clause is executed before the `SELECT` clause. Therefore, you cannot use an alias defined in the `SELECT` clause in the `WHERE` clause as it will result in an error:

```sql
-- This query will result in an error
SELECT
    St_Id,
    CONCAT_WS(' ', St_Fname, St_Lname) AS FullName,
    St_Age
FROM
    Student
WHERE
    FullName = 'Ahmed Hassan'; -- Error, `FullName` is not yet defined
```

The same alias can be used in the `ORDER BY` clause because `ORDER BY` is executed after the `SELECT` clause.

`ORDER BY` is executed after `SELECT` because we need to select the data first before ordering it.

On the other hand, the `WHERE` clause is executed before the `SELECT` clause because we need to filter the rows first before selecting the columns.

```sql
SELECT
  St_Id,
  CONCAT_WS(' ', St_Fname, St_Lname) AS FullName,
  St_Age
FROM
  Student
ORDER BY
  FullName;
```

10 Union Family Operators

Union family operators consist of:

- `UNION`
- `UNION ALL`
- `INTERSECT`
- `EXCEPT`

They combine the results of two or more `SELECT` statements into a single result set. They can reduce requests to the database as data can be fetched in a single request instead of multiple requests.

For example, you have two databases each one in a different location, and you want to get the data from both databases, you can use the `UNION` operator to get the data from both databases in one request.

10.1 `UNION`

`UNION` combines the results of two or more `SELECT` statements and ignores duplicate rows (only returns one copy of the duplicate rows).

In the image below `Osama` appeared only once with data returned from the first table:

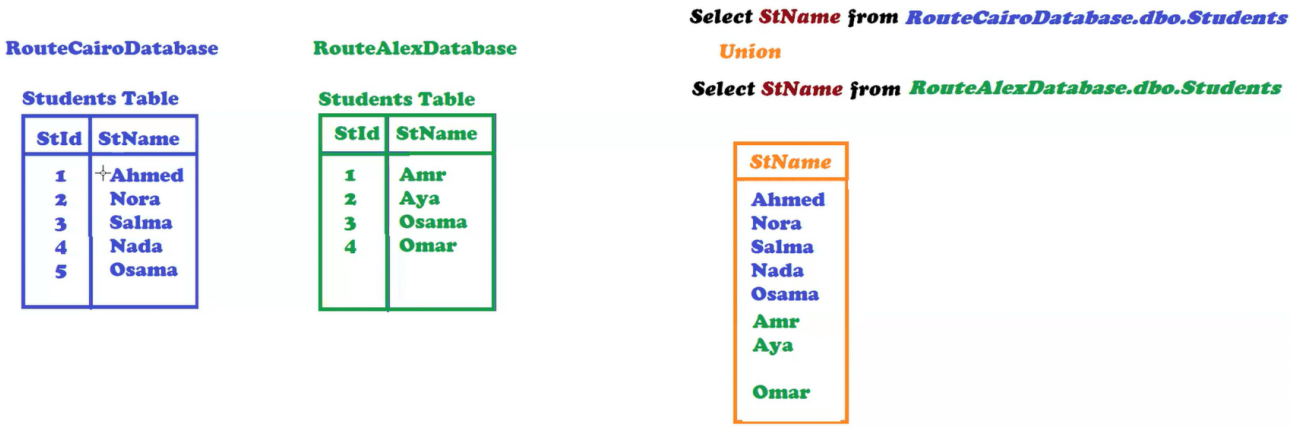

Figure 3: `UNION` Operator

10.2 `UNION ALL`

`UNION ALL` combines the results of two or more `SELECT` statements and does not ignore duplicate rows.

> **Note:**
>
> Duplicates are determined based on the selected columns and not all columns.
>
> For example if we select only `St_FName` in both queries, `UNION` will check for duplication in `St_FName` only and not in all column values in that row in both tables.
>
> If we select `St_LName` and `St_FName` in both queries, `UNION` will check for duplication in both `St_LName` and `St_FName` and if it sees a row that's for example has the same `St_LName` but different `St_FName` it will not remove it from result set.

Here `Osama` appeared twice:

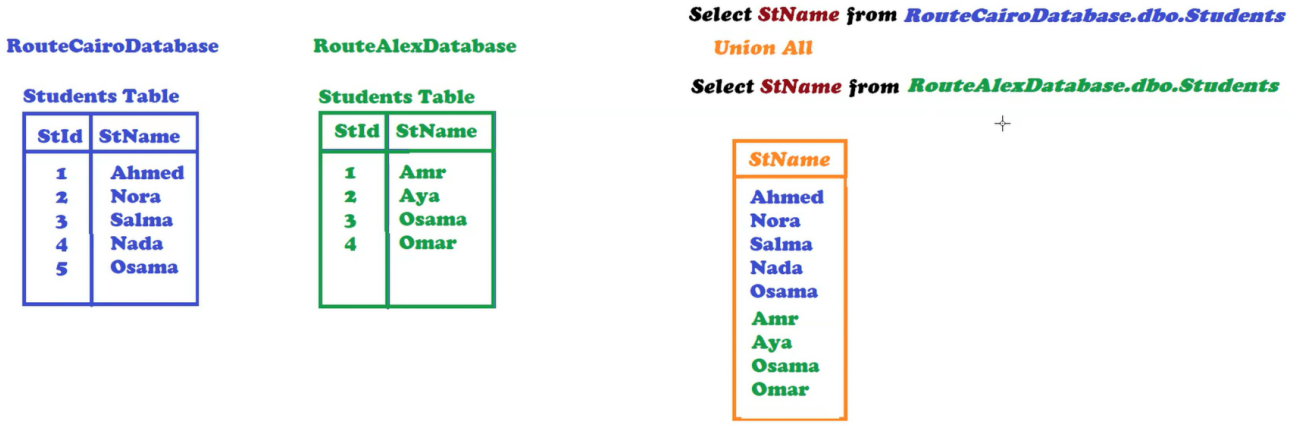

Figure 4: `UNION ALL` Operator

10.3 `INTERSECT`

`INTERSECT` returns only the common rows between two `SELECT` statements. The result of `INTERSECT` is what is what `UNION` has ignored (the duplicates).

Only `Osama` appeared here because he is the only common row between the two tables:

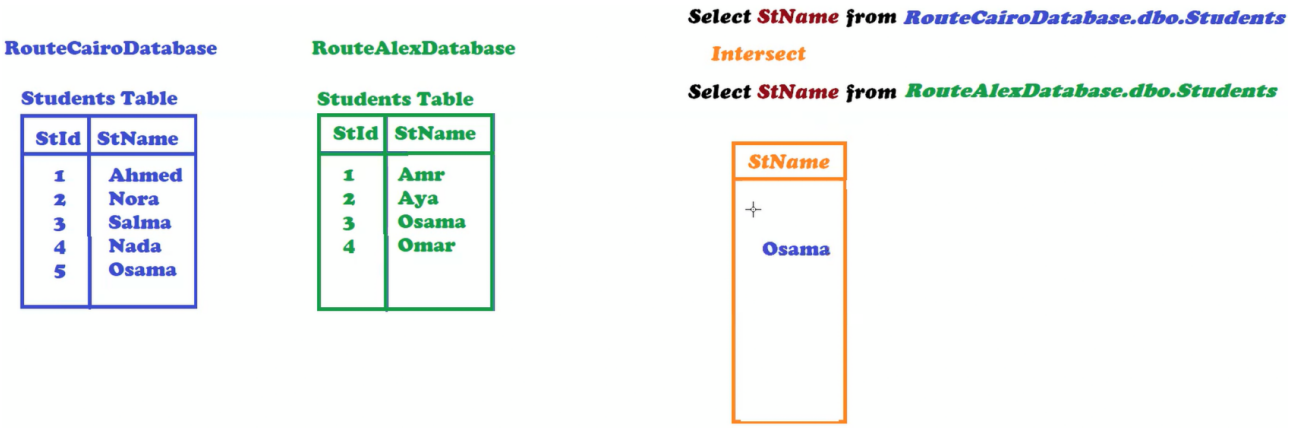

Figure 5: `INTERSECT` Operator

10.4 `EXCEPT`

`EXCEPT` returns the rows that are in the first `SELECT` statement but not in the second `SELECT` statement.

In the image below `Ahmed`, `Nora`, `Salma`, and `Nada` appeared because they are in the first table but not in the second table:

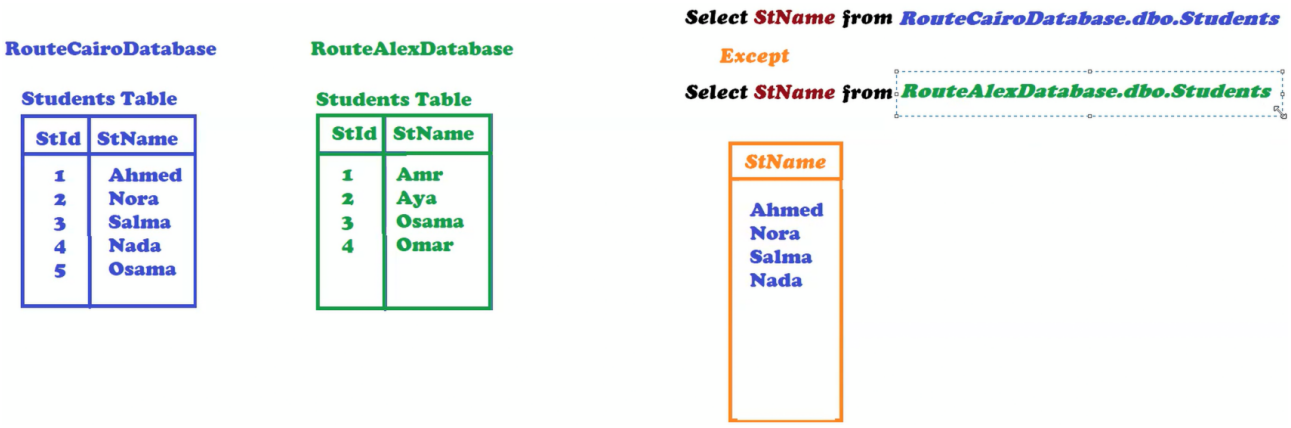

Figure 6: `EXCEPT` Operator

10.5 Importance of Understanding Union Family Operators

Understanding these operators is important when working with technologies like LINQ (Language Integrated Query) in C#, LINQ has functions that are converted to SQL queries and executed on the database.

The advantage of LINQ is that it provides a unified syntax to work with any database, and it will handle the differences between the databases SQL syntax. You just write LINQ code and it will be converted to the correct SQL syntax for the database you are using.

Note:

Why to learn SQL if we are going to use LINQ?

- LINQ queries are converted to SQL queries and executed on the database, so you

need to know SQL to write efficient LINQ queries.

- To write reports, you need to know SQL because the data in the reports is fetched using SQL queries.
- The first 30 min of most interviews are about SQL, so you need to know SQL to pass the interview.

10.6 Example of Using Union Family Operators

We can have two arrays of employees and we want to get the common employees between the two arrays based on their SSN, Name which are a subset of the properties the employee object has.

We can use `INTERSECT` to get the common employees between the two arrays based on this subset of properties.

Csharp 11 update came with `IntersectBy` function that helps us to intersect two collections based on a key selector, but we still need to learn the older `Intersect` function as you may see it in existing code.

10.7 Misuse of `UNION`

We may see an misuse of `UNION` in some cases, for example if we have a table of students and we want to get students who live in either `Cairo` or `Giza`, we can use the `IN` operator, but some people may use `UNION` to get the students who live in `Cairo` and then get the students who live in `Giza` and then use `UNION` to combine the two results, which is not efficient.

`UNION` family operators should be used when you have to combine results from two or more tables with no relation, and we want to get the data from both tables in one result set (for example two different tables in different databases or different schemas).

Incorrect:

Getting students in Cairo or Alex using `UNION` is not efficient, use `IN` instead

```
SELECT
  *
FROM
  Student
WHERE
  St_Address = 'Cairo'
UNION
SELECT
  *
FROM
  Student
WHERE
  St_Address = 'Alex'
```

Correct:

Getting students in Cairo or Alex using the `IN` operator is more efficient:

```sql
SELECT
  *
FROM
  Student
WHERE
  St_Address IN ('Cairo', 'Alex')
```

10.8 When to Use

Use `UNION` family operators when:

- Combining data from two or more tables with no relation.
- Combining data from two or more tables that have different columns.
- Combining data from different databases or schemas.

When using union family operators:

- the number of columns selected in the two select statements should be the same
- The data type of the columns should be the same.
- The order of the columns should be the same.

If you don't follow the two rules above you will get this error:

```
All queries combined using a UNION, INTERSECT or EXCEPT operator must
have an equal number of expressions in their target lists.
```

10.9 Example

```sql
SELECT
  Ins_Name,
  Ins_Degree
FROM
  Instructor
UNION ALL
SELECT
  St_Fname
FROM
  Student;
```

11 Database Hierarchy and Schemas

Figure 7: SQL Server Hierarchy

SQL Server has a hierarchy:

1. SQL Server service (DB Engine).
2. SQL Server instance has multiple databases.
3. A database consists of schemas.
4. A schema consists of database objects (tables, views, functions, stored procedures, etc.).
5. Tables consist of columns and rows.

Schemas are used to divide the database into logical related groups, for example, we can have a schema for the HR department, a schema for the IT department, a schema for the finance department, etc.

The default schema in SQL Server is `dbo` which stands for database owner.

Based on the hierarchy above the actual select statement should be like this:

```
SELECT * FROM ServerName.DatabaseName.SchemaName.TableName;
```

> To get the server name we can run:
> `SELECT @@SERVERNAME;`

But since the server name and the database name are already set in the connection string, we can use the following select statement:

```
SELECT * FROM SchemaName.TableName;
```

If the table you are selecting from is in the default schema `dbo`, you can also ignore the schema name in the select statement:

```
SELECT * FROM TableName;
```

Schemas solve the following problems:

1. You can't create database objects (Table, View, Index, Trigger, Stored Procedure, Rule) with the same name.
2. There is no logical meaning (grouping related objects together).
3. Schemas help manage permissions and security.

11.1 Managing Schemas

11.1.1 Creating a Schema

```
1 CREATE SCHEMA SchemaName;
```

You can't put `CREATE SCHEMA` in the same batch with existing queries, so you should put it in a separate batch.

> From MS Docs
> ***Rules for Using Batches:***
> - `CREATE DEFAULT`, `CREATE FUNCTION`, `CREATE PROCEDURE`, `CREATE RULE`, `CREATE SCHEMA`, `CREATE TRIGGER`, and `CREATE VIEW` statements cannot be combined with other statements in a batch. The `CREATE` statement must start the batch. All other statements that follow in that batch will be interpreted as part of the definition of the first `CREATE` statement.

You can use GO to separate between batches (if you are using command line have to use GO to execute the batch):

```
... -- Some existing queries
GO; -- End of the batch above
CREATE SCHEMA SchemaName;
GO; -- End of the current batch
```

11.1.2 Creating a Table in a Schema

```
CREATE TABLE SchemaName.TableName (
    Column1 DataType,
    Column2 DataType,
    ...
);
```

If you remove the schema name from the query, the table will be created in the default schema dbo.

11.1.3 Dropping a Schema

You will have to remove all objects in the schema first before dropping a schema.

```
DROP SCHEMA SchemaName;
```

11.1.4 Transferring a Table Between Schemas

```
ALTER SCHEMA
  NewSchemaName TRANSFER OldSchemaName.TableName;
```

The schema you are transferring data to can't have a table with the same name.

After transferring the table from its old to new schema and trying to query the table from the new schema, you may get an error that the table does not exist, you can solve this by refreshing your connection to the database.

11.2 Examples

Example selecting Student table from the dbo schema in the ITI database in my EndeavourOS server:

```
SELECT
  *
FROM
  EndeavourOS.ITI.dbo.Student;
```

12 DDL: SELECT INTO and TRUNCATE

DDL (Data Definition Language) includes statements like CREATE, ALTER, DROP, SELECT INTO, and TRUNCATE.

So far we have talked about CREATE, ALTER, DROP. Now lets talk about SELECT INTO and TRUNCATE.

12.1 SELECT INTO

`SELECT INTO` is used to **create a new table** based on the result of a `SELECT` statement.

Each table consists of structure which is (Columns, Keys, Constraints, Indexes, Triggers, etc) and data which is the rows.

`SELECT INTO` copies data and only (columns, constraints) from a source table to a new table.

12.1.1 Basic Usage

```
SELECT
 * INTO table_name
FROM
[db_name].[dbo].[table_name];
```

This will create a new table named `table_name` in the current database, and copy the data and structure (columns and constraints) from the table `[db_name].[dbo].[table_name]`.

12.1.2 Copying Specific Columns

You can specify which columns to copy to the new table:

```
SELECT
  column1,
  column2
INTO
  table_name
FROM
  [db_name].[dbo].[table_name];
```

12.1.3 Filtering Rows

You can also use a `WHERE` clause to filter the rows you want to copy:

```
SELECT
  *
INTO
  table_name
FROM
  [db_name].[dbo].[table_name]
WHERE
  condition;
```

12.1.4 Copying Only Table Structure

To copy only the table structure (columns, constraints) without copying the data, use a `WHERE` clause that is **always `false`**:

```
SELECT
  *
INTO
  table_name
FROM
  [db_name].[dbo].[table_name]
```

```
7 WHERE
8   1 = 0;
```

12.2 TRUNCATE

TRUNCATE is used to remove all rows from a table. It is faster than DELETE because it does not log the deleted rows. However, you cannot restore the data after using TRUNCATE.

12.2.1 How It Works

TRUNCATE is considered a **DDL statement**. DELETE is a **DML statement**.

Note:

TRUNCATE is a DDL command bacaue it drops the table and recreates it, while DELETE is a DML command because it only deletes the rows.

12.2.2 Restrictions

If a table has a foreign key constraint, you cannot use TRUNCATE and must use DELETE instead.

12.2.3 Recovering Data

As we know database has two main files, the .mdf file which contains the data and the .ldf file which contains the transaction log.

You can recover the removed data from DELETE because DELETE logs the deleted rows into the transaction log (.ldf file). TRUNCATE does not log the deleted rows so you can't recover them.

13 INSERT INTO with SELECT

INSERT INTO with SELECT is used to insert data from one table into an existing table.

So far we have learned two types of INSERT:

1. Simple INSERT
2. Row Constructor INSERT

INSERT INTO with SELECT is a third type of INSERT.

13.1 Requirements

1. The number of columns in the source and destination tables must be equal.
2. The data types of the corresponding columns must be compatible.
3. The columns must be in the same order in both tables.

13.2 Inserting Specific Columns

```
1 INSERT INTO
2   table_name (column1, column2, ...)
3 SELECT
4   column1,
5   column2,
```

```
  ...
FROM
  source_table_name
WHERE
  condition;
```

13.3 Inserting All Columns

To insert all columns, you can remove the column list and use *:

```
INSERT INTO
  table_name
SELECT
  *
FROM
  source_table_name
WHERE
  condition;
```