Intermediatte SQL

💻 Access the datasets used here: [https://github.com/AlexTheAnalyst](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbTVUa09MQ0hDYzJmY3RiQ1ZBeHVfUzBlaVlqQXxBQ3Jtc0trNEVOLTBjb3NsUjQ0Y1ZjaWEtRWg4eE9iX0lBbW1ydXh0ZlRNLU01QVVaMm5RM0gyTWtKWXNqYUFVSXJPRVhqbEdrXy1EREkyMFByelVjSVFZN0h2Rkt2cnZ5U0pmckR1TkIwemJTbURIWWsyQ0dVYw&q=https%3A%2F%2Fgithub.com%2FAlexTheAnalyst&v=PSNXoAs2FtQ)

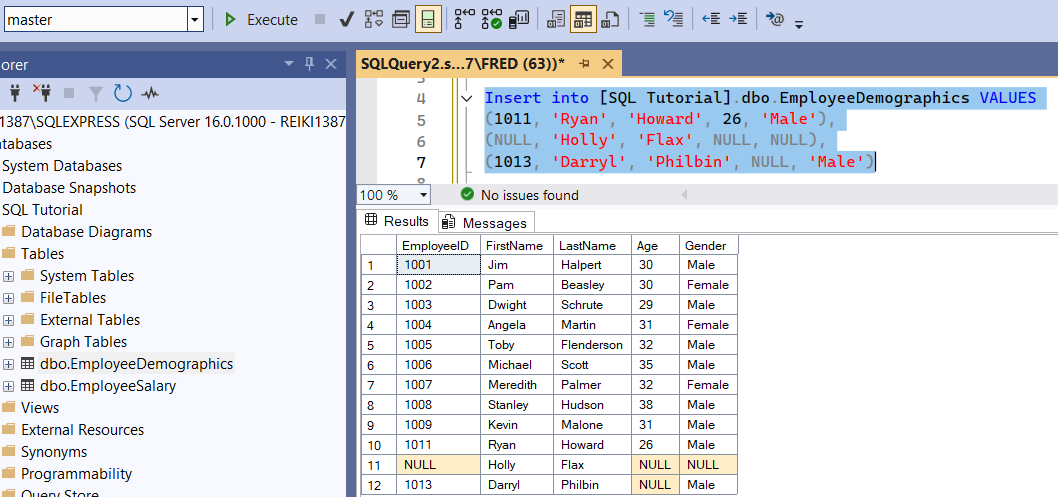


In SQL Server, .dbo refers to the **schema** that owns the object—in this case, a table. This is the **schema**, short for "database owner". It’s like a namespace or container that groups database objects (tables, views, procedures, etc.). Most of the time, when you create a table without specifying a schema, it defaults to dbo, especially if you're the database owner or using default permissions.

You can think of it like a file path:

C:\Users\YourName\Documents\file.txt

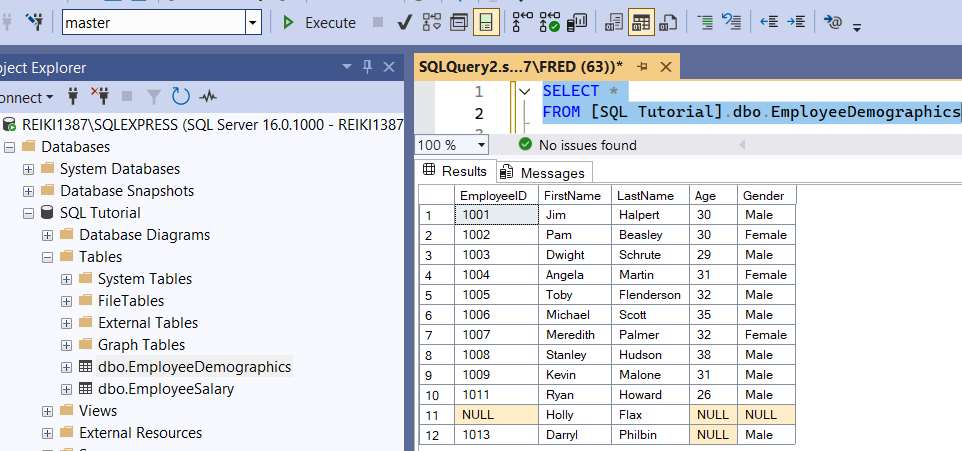
Where dbo is kind of like the Documents folder that holds your table.

Adding values to existing table from different database

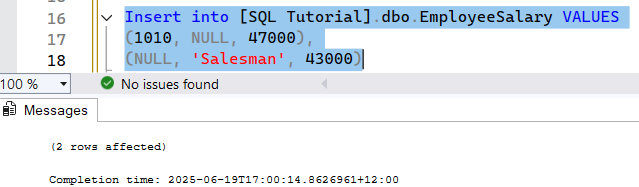
Deleting a row or a NULL row



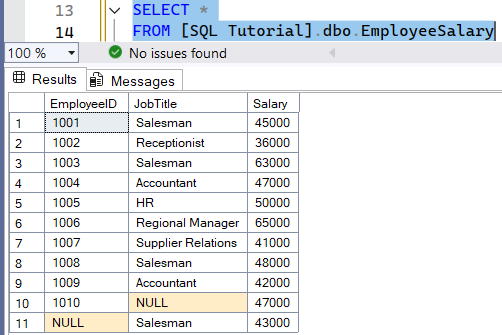
Displaying ALL data from table in another database



Inserting data into EmployeeSalary

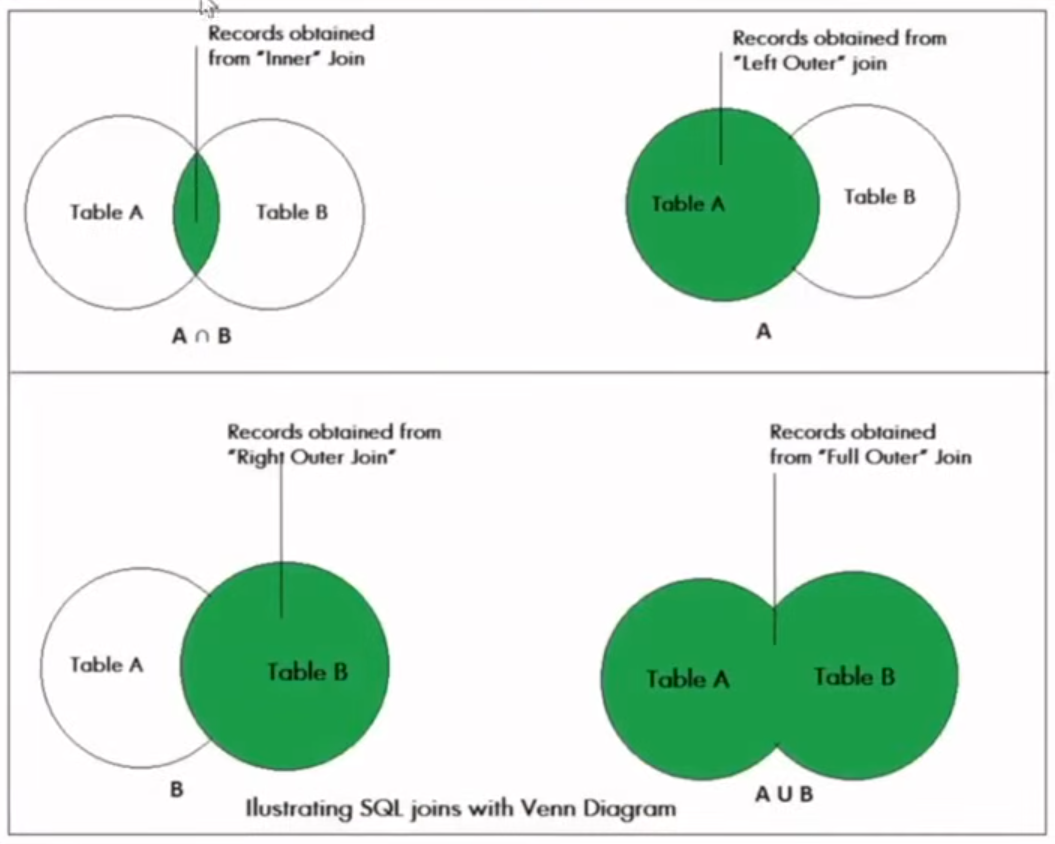


Displaying ALL data from table in another database

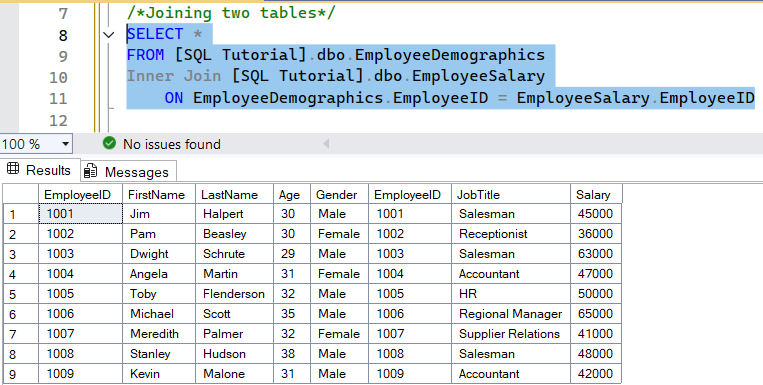


Inner Joins, Full/Left/Right Outer Joins

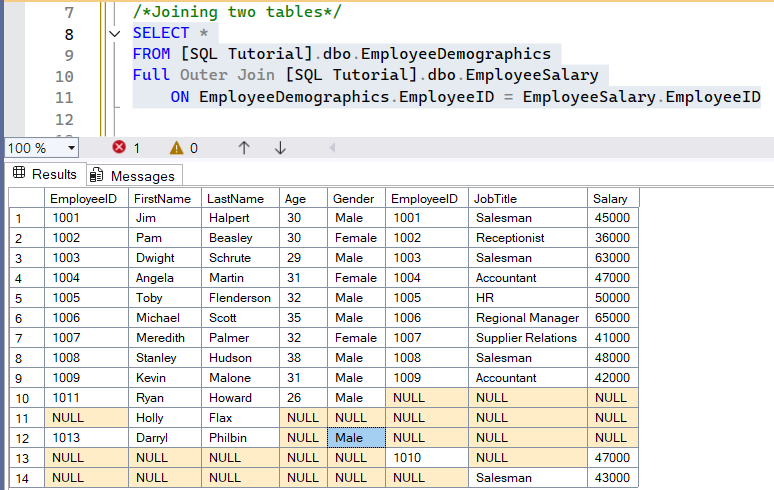
Joins- used to combine multiple tables into a single output using a common column



INNER JOIN/ JOIN- Joining two tables using this function will join based on the common data between the two tables. The common data used is EmployeeID. If there is no common emplyeeID on the two ttable it will not be include in the joined table



Full Outer Join – It fully joins ALL the rows in BOTH table even there is no common data. It puts NULL on the table that has no common data



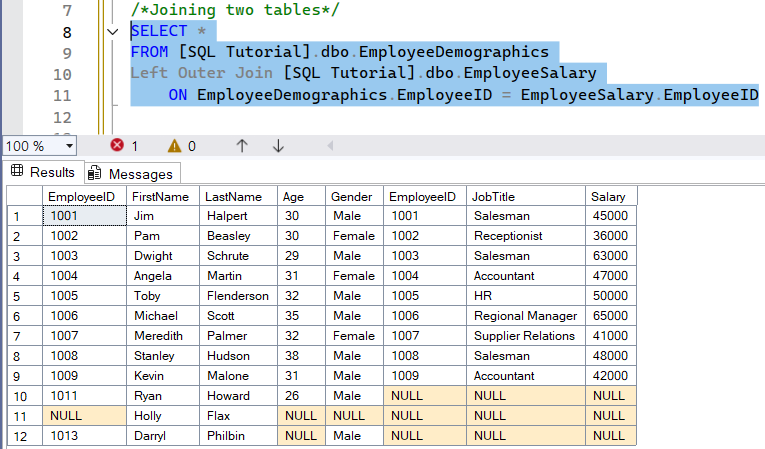
Because EmployeeDemographics is the first table (left) in the code, all the rows were listed first and the corresponding row to the right table is NULL(10-13). 

And row 13-14 of left table is NULL because it has no data similar to the data to the right table

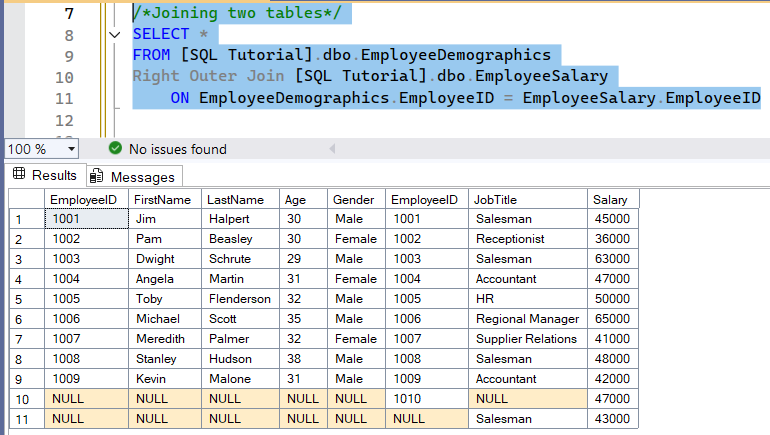


Left Outer Join- return *ALL the common table rows with the left table ONLY* of

“ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID” and give a NULL value to the right table that has nothing to match to the data value EmployeeID

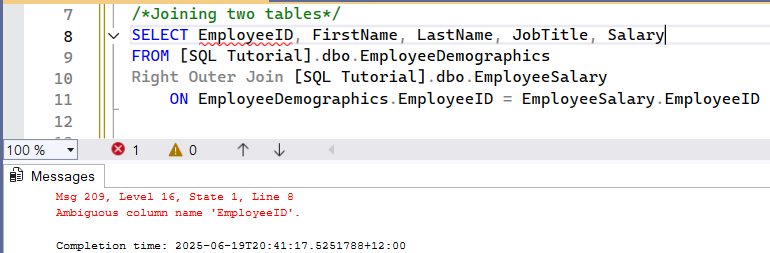


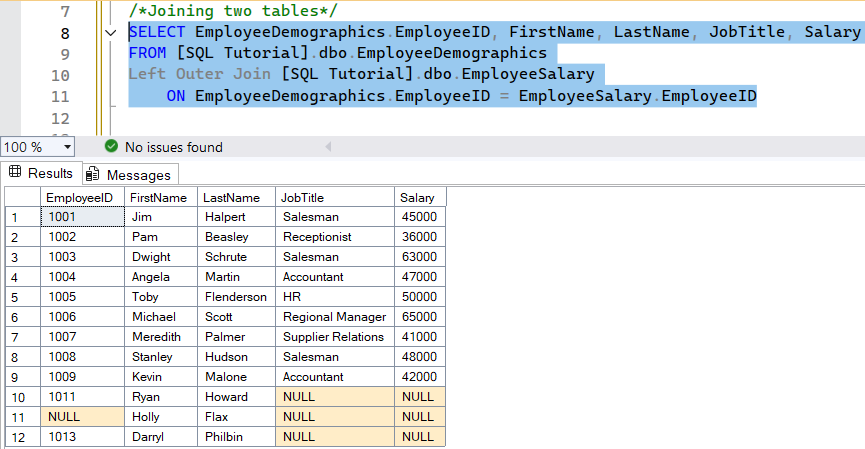
Right Outer Join – return *ALL the common table rows with the right table ONLY* . NULL will be put on the left table if there is no common data with the right table (EmployeeID)



NOTE: *When using joins, ‘SELECT \*’ is not really advisable. User should SELECT the specific column that need to be joined.*

SPECIAL CASE: This query gives error because there is an “EmployeeID” on both tables and the SELECT statement doesn’t know if it will select the LEFT or RIGHT table

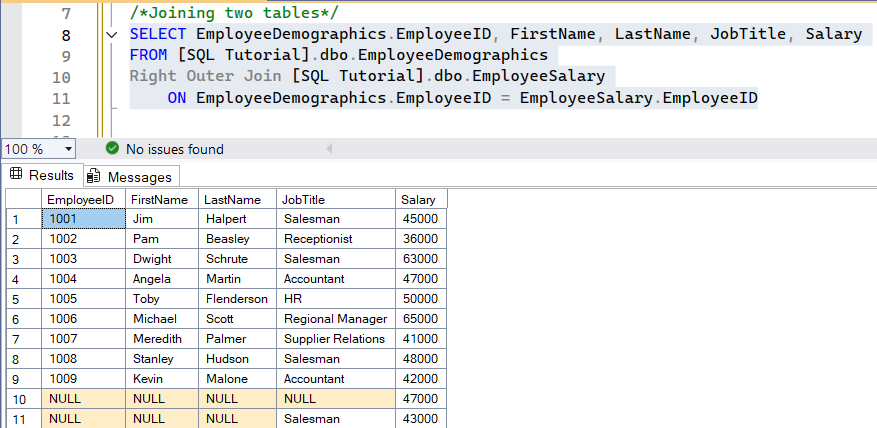


CASE 1: Here we use SELECT EmployeeID from the EmployeeDemographics and we are using ‘Left Outer Join”

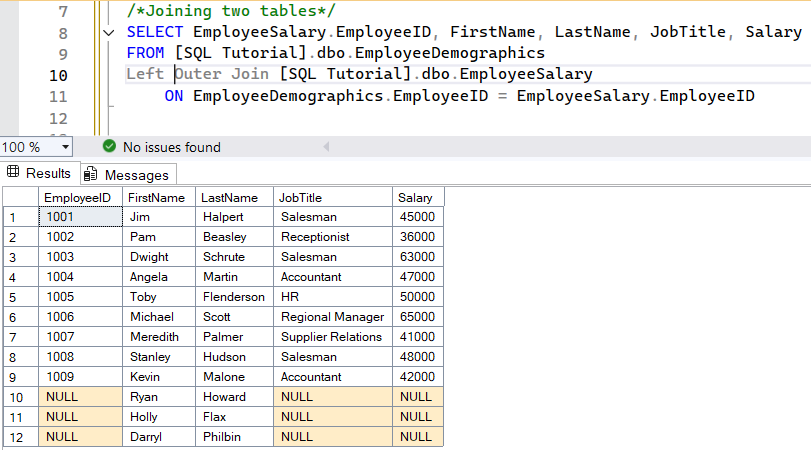
CASE 2: BELOW: The query “Right Outer Join” will determine the table length which is the right, Salary table length. So if the query is the “Right Outer Join” then all the column and rows data from the right will normally show

The ‘SELECT EmployeeDemographics.EmployeeID’ is what is showing in EmployeeID column.

If it has similar EmployeeID with the “Right Outer Join” then the left table (Demographic) will show its data (FirstName & LastName) and EmployeeID column



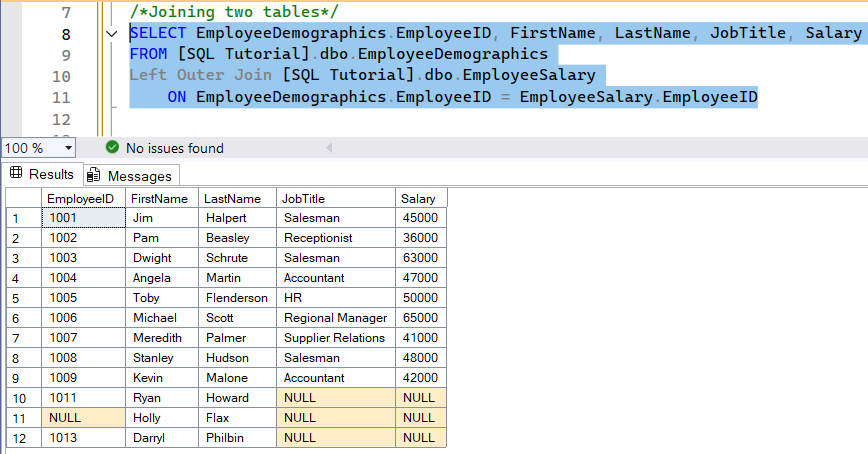
CASE 3: BELOW: The query “Left Outer Join” will determine the length based from the left table, Employee Demographic table length. So if the query is the “Left Outer Join” then all the column and rows data from the left will normally show BUT you are selecting EmployeeSalary.EmployeeID, that’s why the 10-13 EmployeeID is NULL because it is from the right table



CASE 4: BELOW: The query “Left Outer Join” will determine the table length which is the left, Demographic table length. So if the query is the “Left Outer Join” then all the column and rows data from the left will normally show

The ‘SELECT EmployeeDemographic.EmployeeID’ is what is showing in EmployeeID column. If it has similar EmployeeID with the Right table then the right table (Salary) will show its data.

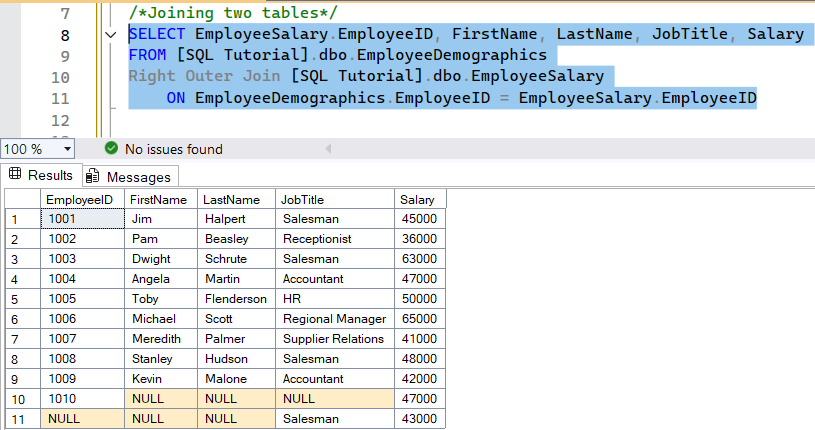
In this case, EmployeeID (Demographics) column has no common EmployeeID with the right table, so the right table 10-13 columns are NULL



CASE 5: The query “Right Outer Join” will determine the length from the right table, Salary table. So if the query is the “Right Outer Join” then all the column and rows data from the right will normally show

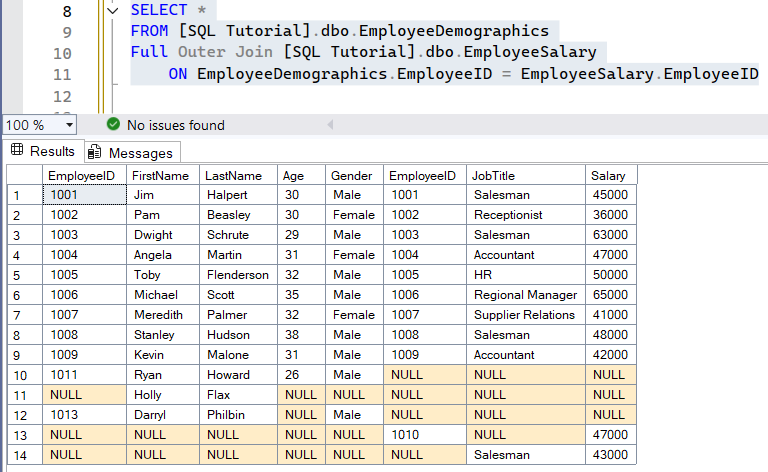
The ‘SELECT EmployeeSalary.EmployeeID’ is what is showing in EmployeeID column.

ALL THE NULL WAS NOT INCLUDED FROM THE RIGHT TABLE, so instead of 14 rows of EmployeeID it only become 11 rows.

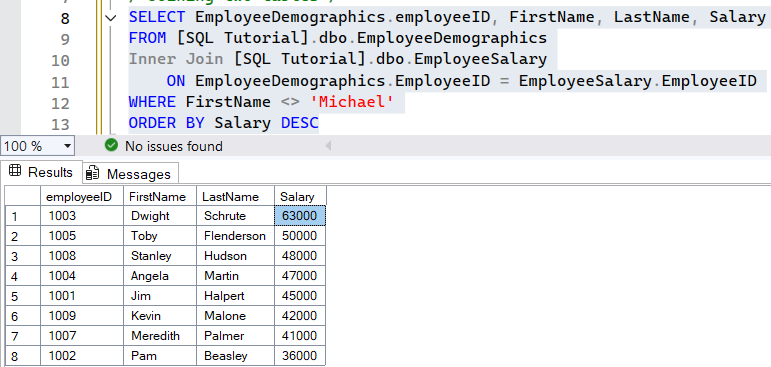


USE CASE A: Michael which is also an employee on the table wants to know who has the highest salary aside from him

1. Get a general overview first of all the columns

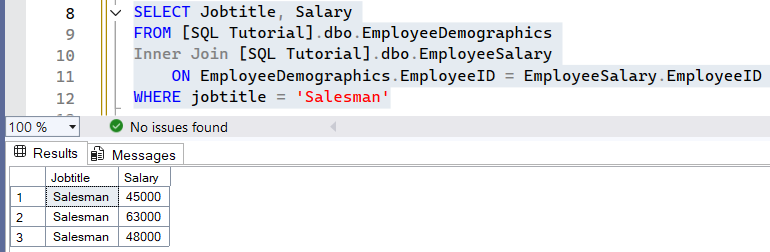


1. Know what data does user need. EmployeeId, first and lastname and Salary
2. Michael scott should not be included. Salary should be on descending order

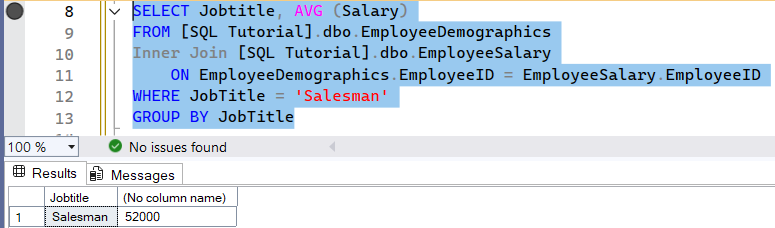


USE CASE B: Kevin Malone, an accountant and who is also an employee in the database, made a mistake at looking at the average salary of salesman. So we need to compute the average salary of salesman

1. User need to know all the salary of specific job title, Salesman.

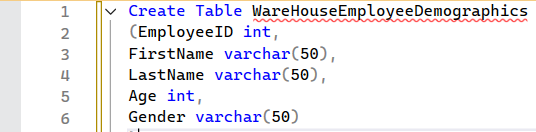


1. To compute the average salary of all the salesman in the company

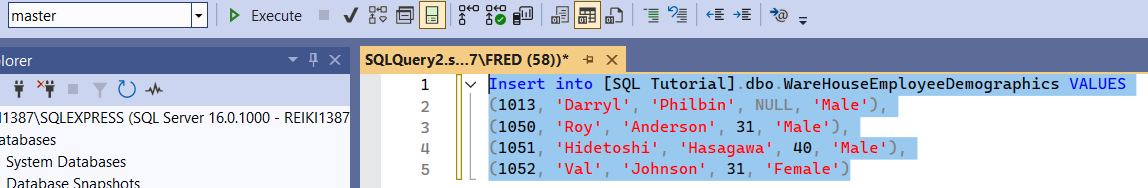


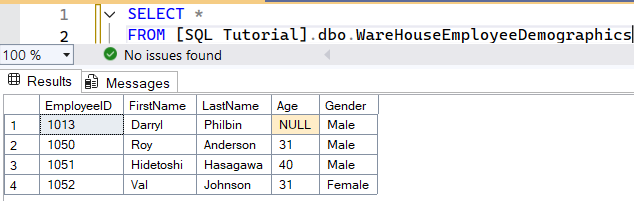
1:03:05 Union- like joins combine two tables to have one output. A joins combine two tables base on a common column and when selecting dat. The joins can only chose one table to be the primary joins. Full outer joins combines both table but the common column are both displayed. WHILE IN UNIONS you can select all the data in both the tables in put it in one output and not separated out

Creating empty tables that has 50 capacity



Inserting into a table

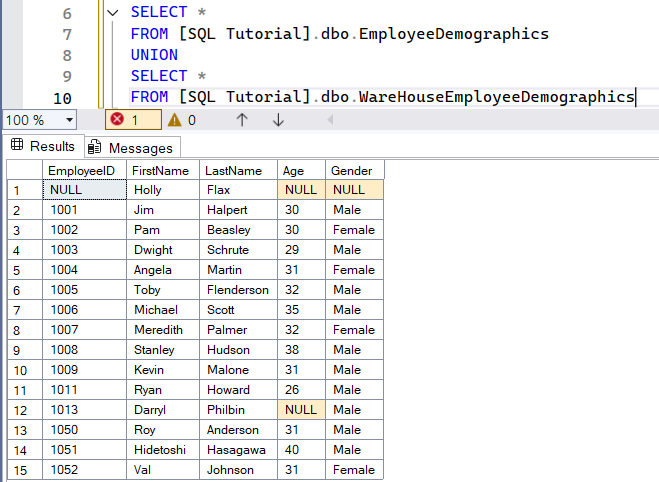




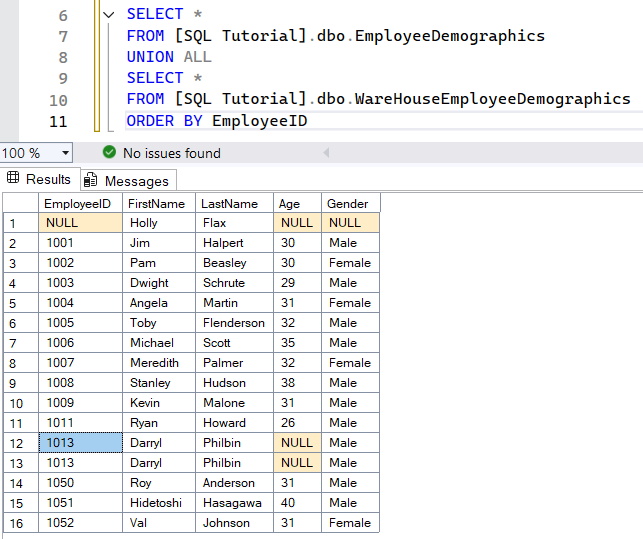
UNION sample data: The column names are the same on the two table. Using Full Outer join does not combine it.



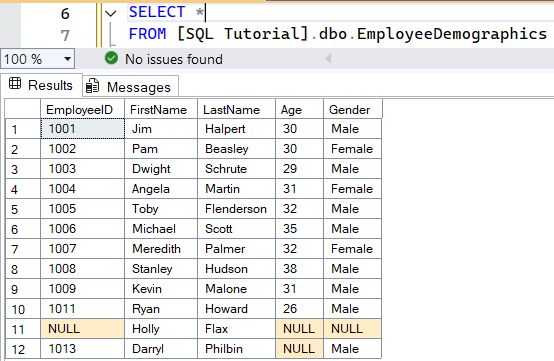
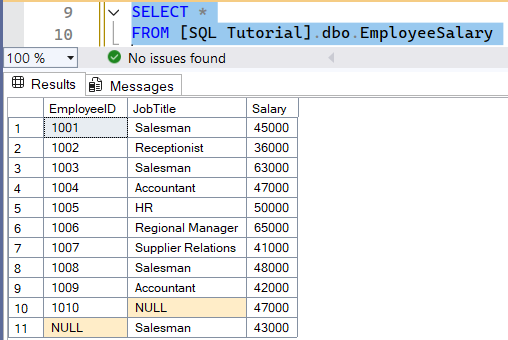
UNION take out and remove the duplicates. Observe that “1013 Daryl…” are exactly same data on the two tables. Also notice that Employee ID is automatically arrange in ASC order. The alphabetical data is always is always ranged before numerical data



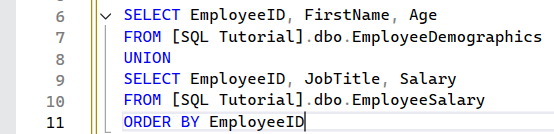
UNION ALL- combines all tables and does not remove duplicates



CASE 1: Combining two tables with DIFFERENT COLUMN NAMES using UNION

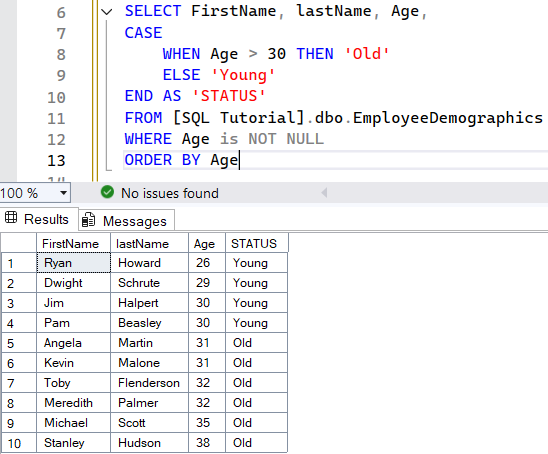
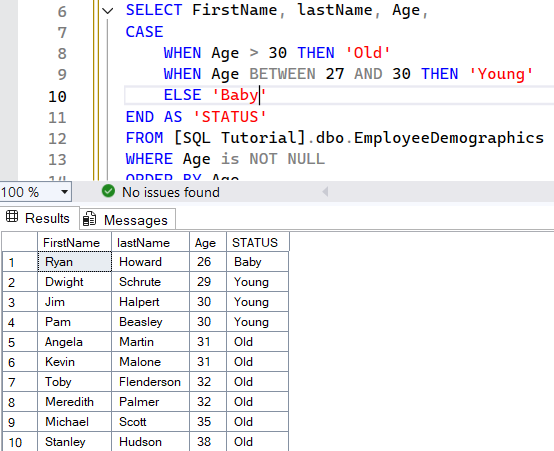
 

Experiment in combining using UNION. Observe that the column name output was taken from the first SELECT statement. This union does not give error because there are 3 columns on both tables that are combined and the DATA TYPE ARE THE SAME. You can see that even though there are common column (EmployeeID) it does not remove to consider it as duplicate. Be CAREFUL on selecting data using UNION, THE COLUMN NAMES SHOULD BE THE SAME

CASE STATEMENT- allows to specify conditions and specify what you want to return when that conditioned is met

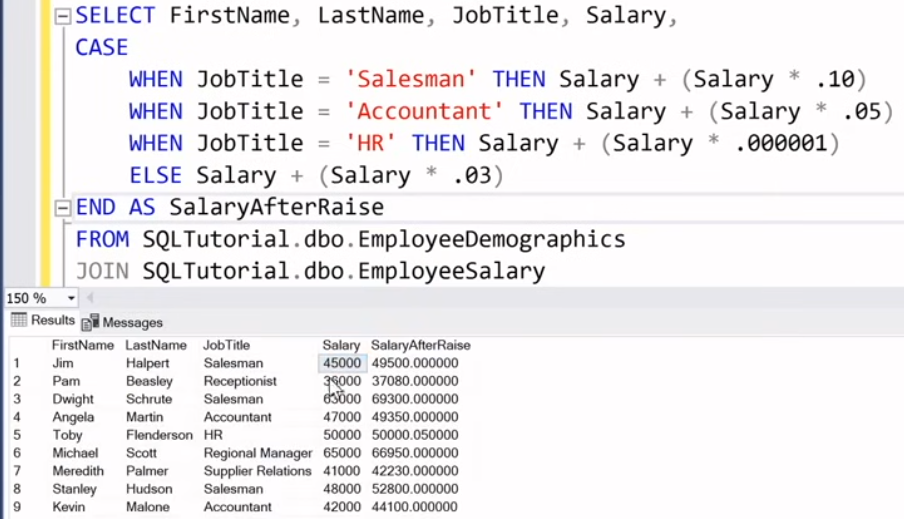
Case statement is part of the SELECT statement so comma is needed after ‘Age’

If there are multiple conditions that are satisfied, ONLY THE FIRST condition met will be executed



USE CASE: Increasing salary



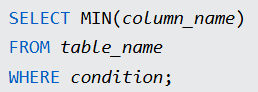
An aggregate function is a function that performs a calculation on a set of values, and returns a single value.

Aggregate functions are often used with the GROUP BY clause of the SELECT statement. The GROUP BY clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.

The most commonly used SQL aggregate functions are:

* MIN() - returns the smallest value within the selected column
* MAX() - returns the largest value within the selected column
* COUNT() - returns the number of rows in a set
* SUM() - returns the total sum of a numerical column
* AVG() - returns the average value of a numerical column

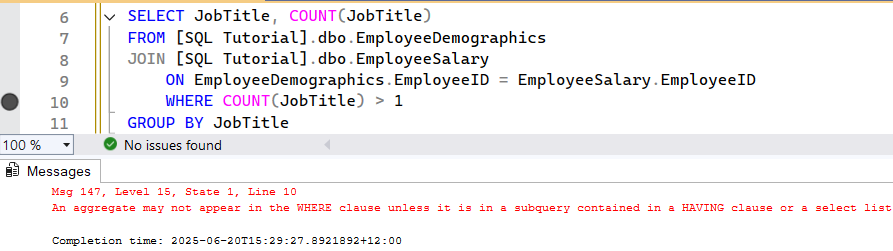
***Aggregate functions ignore null values (except for COUNT()).***

SQL Subquery: A Comprehensive Guide

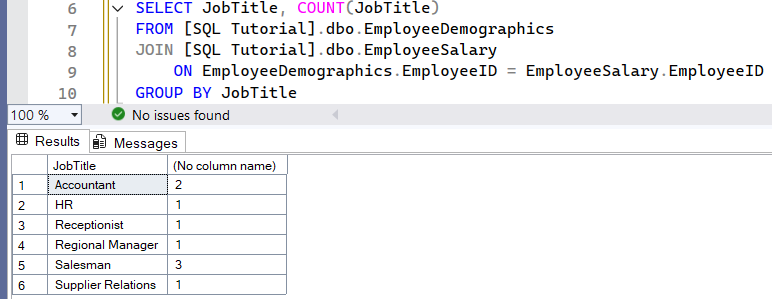
<https://www.datacamp.com/tutorial/sql-subquery>

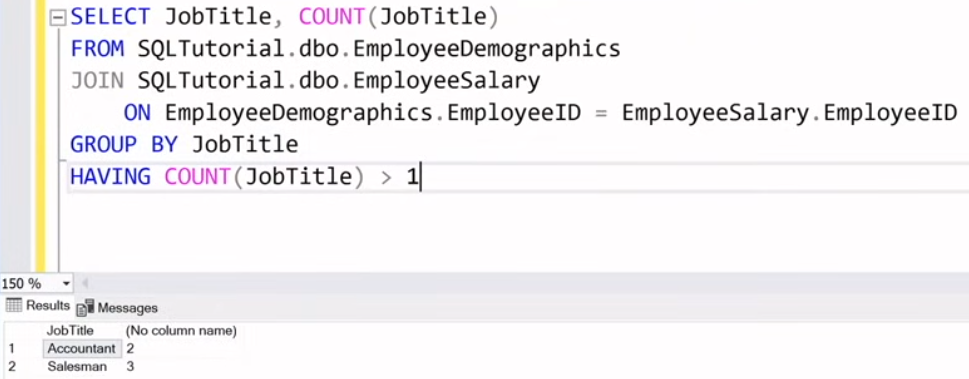
HAVING CLAUSE - Similar to WHERE but WHERE is only for non-aggregated values. The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.



*\*\*HAVING clause is completely dependent on GROUP BY and before ORDER BY. This means HAVING should come after GROUP BY*

Reason: The GROUP BY clause combines similar rows, producing a single result row for each group of rows that have the same values, for each column listed in the (SELECT) Projection clause. The HAVING clause sets conditions on those groups after you form them. WE CANNOT LOOK (having clause) AT THE AGGREGATED DATA BEFORE IT IS ACTUALLY AGREGATED IN THE GROUP BY CLAUSE



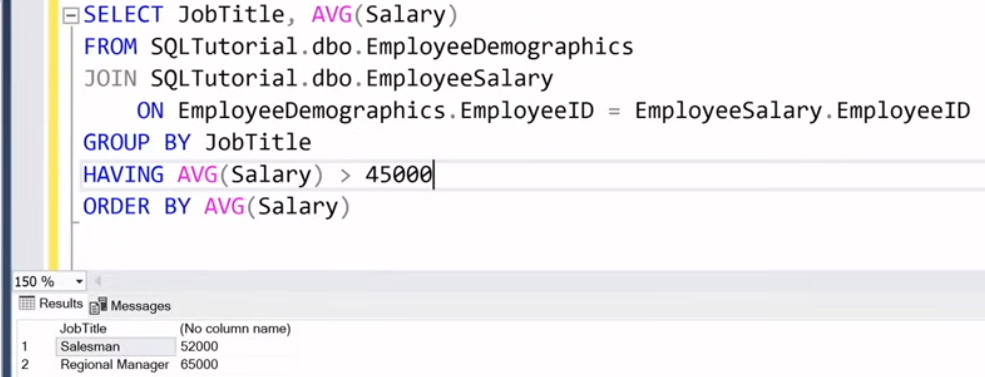


The Projection clause (sometimes called the Select clause) specifies a list of database objects or expressions to retrieve, and can set restrictions on qualifying rows. The select list is sometimes also called the projection list.

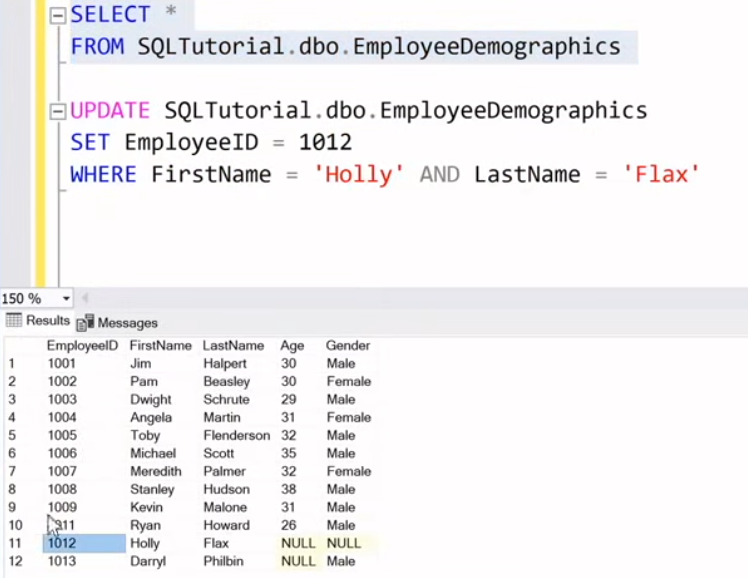
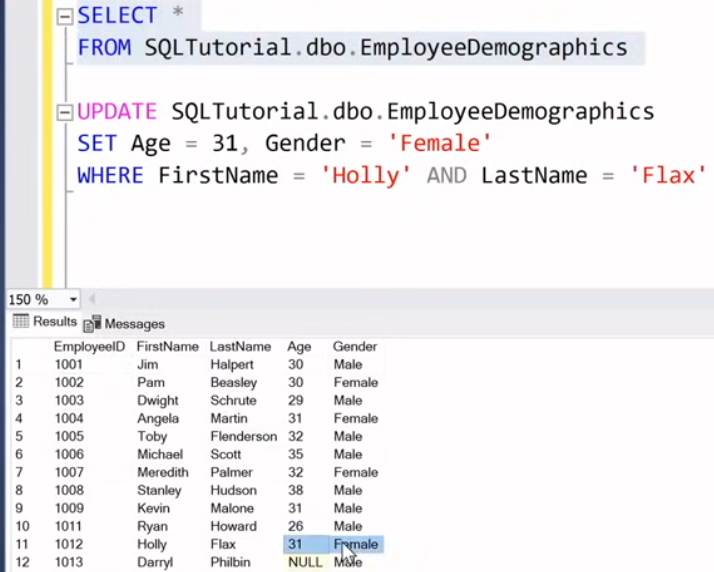
Difference between Selection and Projection: Selection retrieves complete rows of data based on certain conditions, whereas projection retrieves specific columns from these rows.

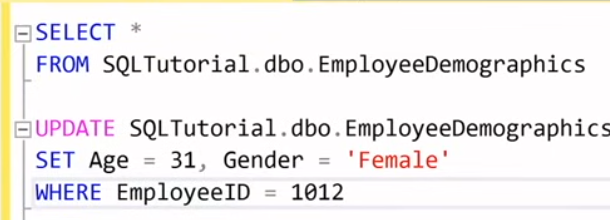
Syntax: Selection is accomplished using the WHERE clause in SQL, while projection is achieved by specifying the desired columns in the SELECT statement.

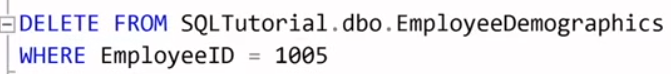
Querying job titles that have over 45000



UPDATING AND DELETING DATA IN A TABLE



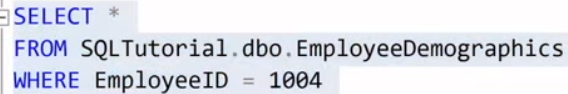
DELETE- removes entire row. It cannot be reversed once deleted

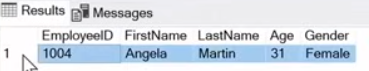


This line will delete everything from a table in a database. If you remove the “.dbo.EmployeeDemographic” and remain only the “DELET FROM SQLTutorial”, this will delete the whole database



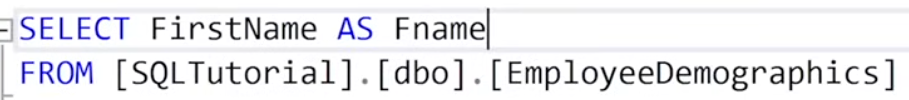
To be safe, use select statement first

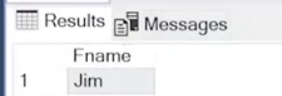




ALIASING

The same result





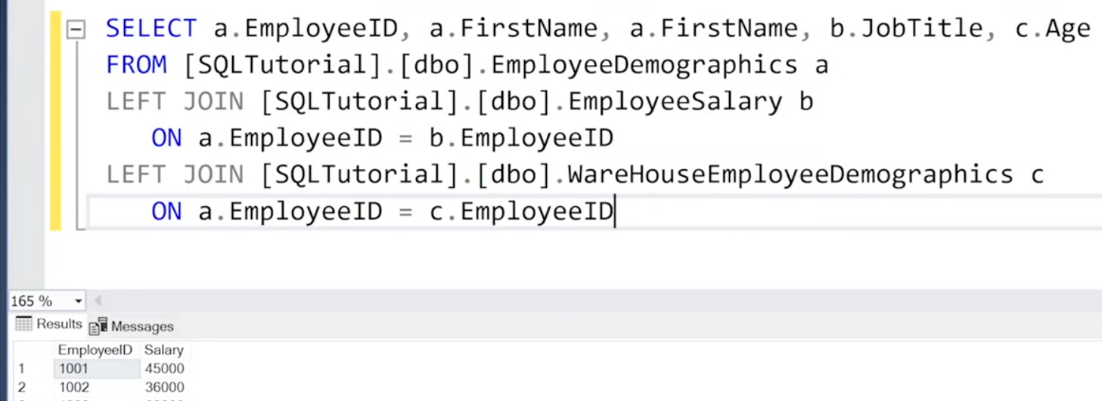
SELECTING AND CONCATENATING

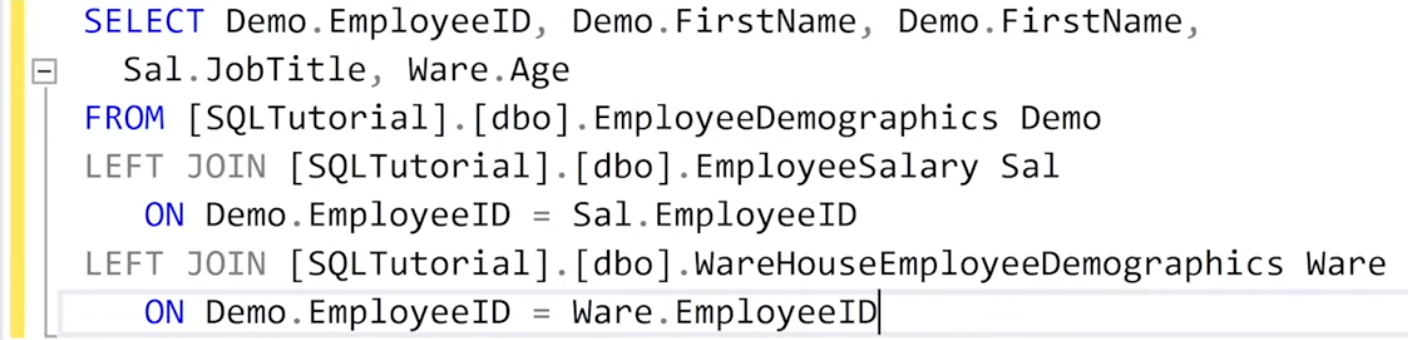
NOTE: If you are selecting a table from a database that has been aliased you need to use the Alias to SELECT the database table even though the aliasing is after the SELECT clause



BAD ALIASING



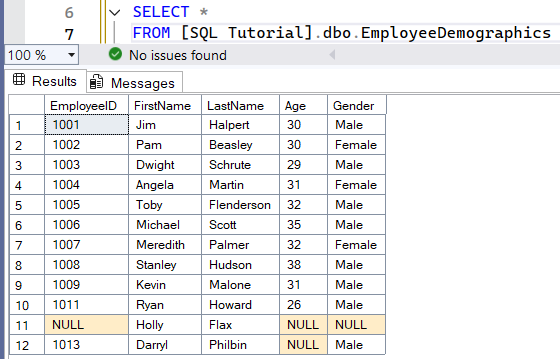
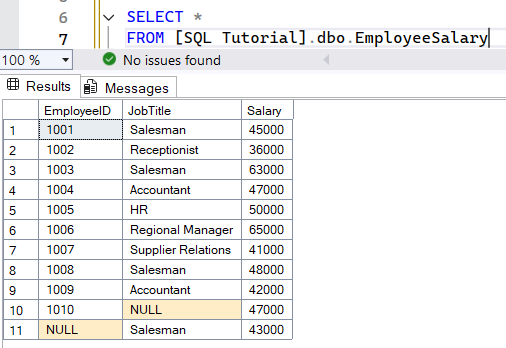
BETTER



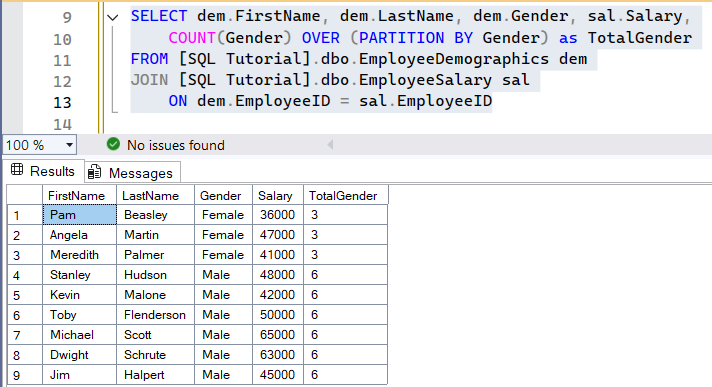
PARTITION BY- where it reduces the number of rows returned in output. Is used to compare to GROUP BY . The PARTITION BY clause in MySQL is used to divide result sets into partitions to perform computations on each partition independently. It is often used in conjunction with window functions to provide a more granular level of analysis.

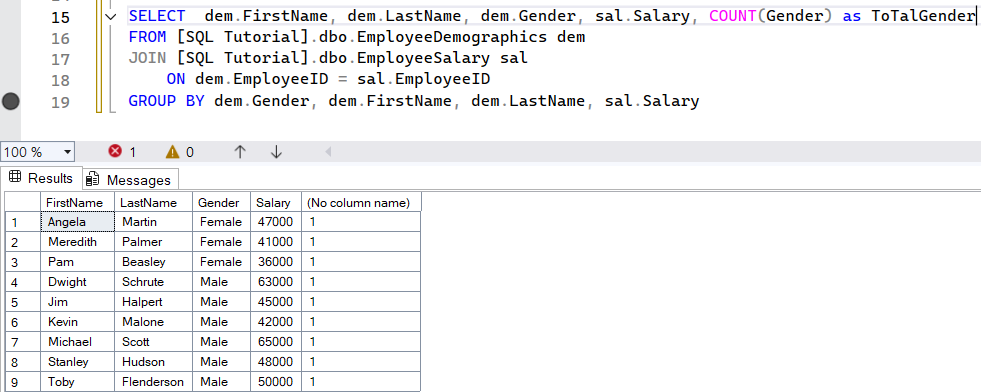
<https://www.datacamp.com/doc/mysql/mysql-partition-by>

1. Checking database

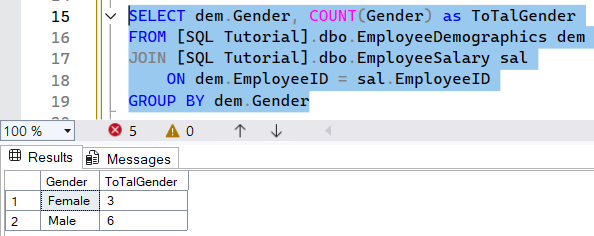
1. COMPARISON: Partition By (Shows the aggregate function output. Does not need to use what is selected in SELECT clause) vs Group BY (Does not show the aggregate function output. Need to use in group by what was selected in SELECT clause)





### Use Case Here: Adding Group-Level Insight to Each Row

This is useful when you want **summary statistics alongside detailed data**, without grouping and losing rows like you would with a GROUP BY.



### Why is this useful?

* You can show each employee and also include how many others share their gender.
* It's helpful for reporting, dashboards, or analytics where you want row-level data with context.

So this is a good use case for PARTITION BY: **showing group-level aggregates (like counts, averages, ranks) in every row, without hiding the details**.

*In SQL, the OVER keyword is used with* ***window functions****. It tells the database to apply a function (like ROW\_NUMBER(), RANK(), COUNT(), AVG(), etc.)* ***across a "window" of rows****, rather than collapsing rows like GROUP BY does.*

### “OVER” Basic Idea:

The function runs over a set of rows defined by the OVER() clause, but the original rows are **not grouped or removed**, you still get every row in the result

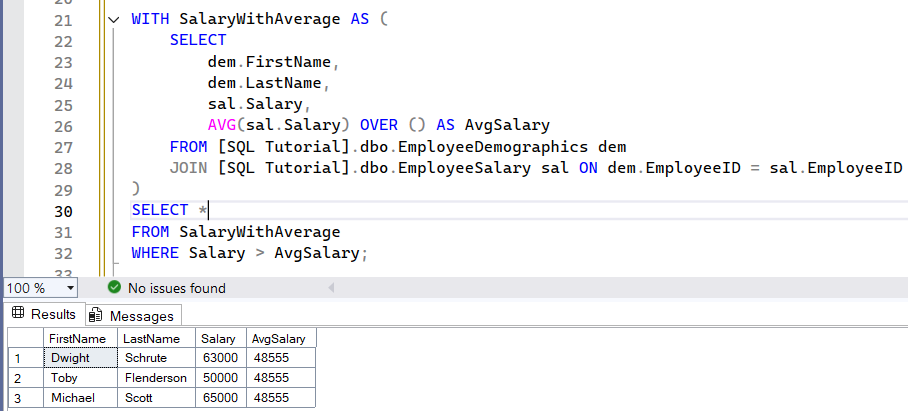
ADVANCE SQL

CTEs (also called as “WITH” query) 1:34:50

A CTE in SQL stands for Common Table Expression. It’s like a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. **Only exist within the query. If you make another CTE or query, the previously made CTE is useless**

Think of it as writing a subquery but giving it a name, which can make your SQL much easier to read and manage—especially if you're repeating logic or building something step-by-step

**CTE (Common Table Expression)** exists **only in memory** and **only for the duration of the query** that uses it. It’s a **temporary, logical construct** and not something that gets stored in the database or written to disk.



1. A semicolon (;) ends the SQL statement.
2. When you see OVER() with **empty parentheses**, like in:

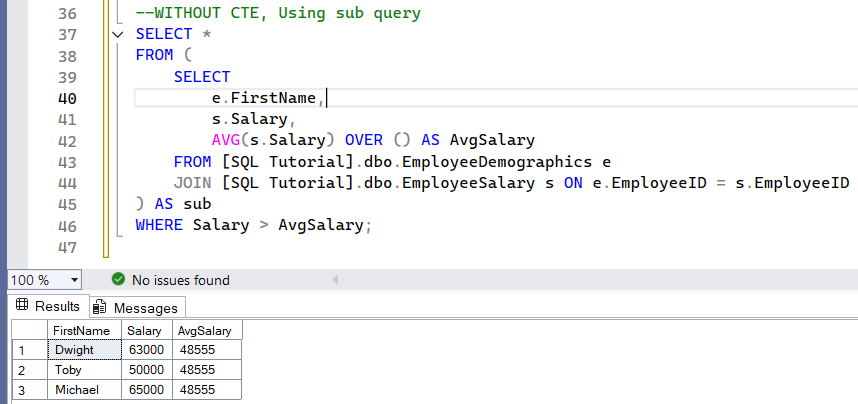
*AVG(s.Salary) OVER () AS AvgSalary*

…it means you're calculating the **average over the entire result set**—no partitioning, no ordering, just one big window that includes all rows.

### In short:

* OVER() with **nothing inside** = "apply this function to **all rows**"
* It’s a clean, readable way to include full-table aggregates **inline with your data**

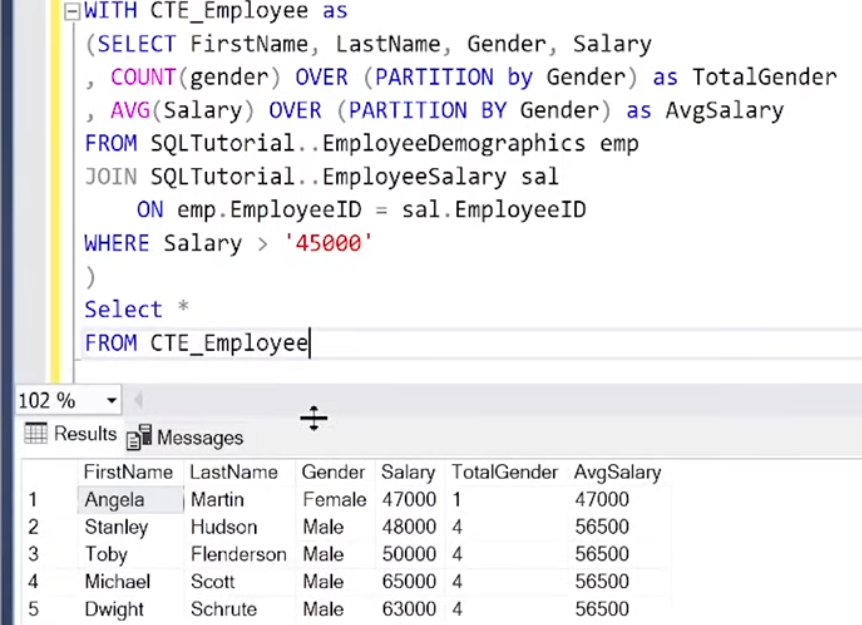
**Comparing without CTE, using sub query**



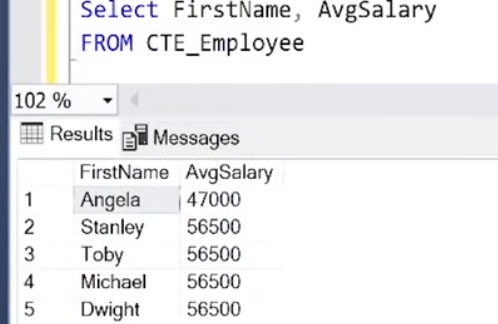
Another Example

Note: CTE is like a temporary place where we can query on the data inside it. CTE only exist in memory, so executing only the query (Select \* FROM CTE\_EMployee) will give error. You need to select all

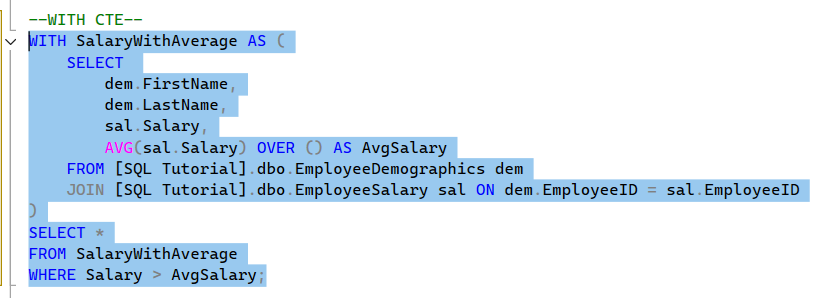
In this example, the CTE\_Employee contains data like FirstName, LastName, Gender, Salary, COUNT(gender), AVG(Salary). So the query (Select \* FROM CTE\_EMployee) is querying from the CTE\_Employee



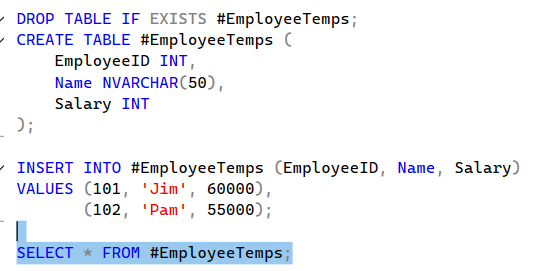
Below: User can also query to only specific columns without re-writing the whole SELECT clause



NOTE: If there are many code line in the query and you want only the CTE code to be executed, you need to select all the query code line and press Execute. There will be no errors even though how many times you execute the CTE code.



BELOW: WHILE in TEMP TABLES, you do not need to run the whole temp table codes. You can execute any query without including the creation of temp tables



**TEMP TABLES**

**Temporary tables** (or **temp tables**) in SQL are like regular tables, but they **only exist for a short period of time**—usually just **for the duration of your session** or until you manually drop them.

They’re useful when you need to **store and work with intermediate results** in a complex query or process.

### ✅ Types of Temp Tables

1. **Local Temp Table**:
   * Created with # (e.g., #MyTempTable)
   * Only visible in your current session/connection
   * Automatically deleted when your session ends
2. **Global Temp Table**:
   * Created with ## (e.g., ##GlobalTemp)
   * Visible to all sessions
   * Deleted when the last session using it ends

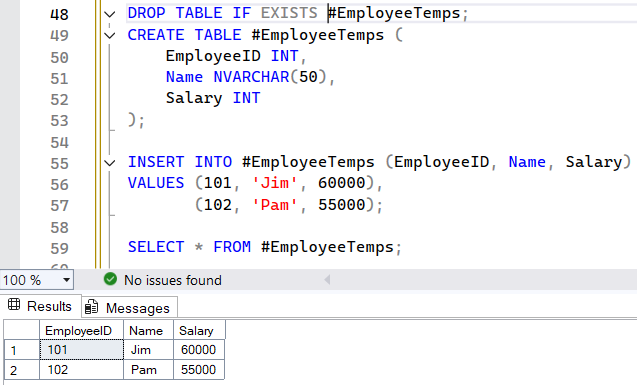
### ✅ When to Use Temp Tables

* When you need to **break a complex query into steps**
* When you want to **reuse a result set multiple times**
* When you're **debugging** and want to inspect intermediate data
* When **performance** matters and you'd rather avoid recalculating expensive joins or filters repeatedly

### 🟡 Notes

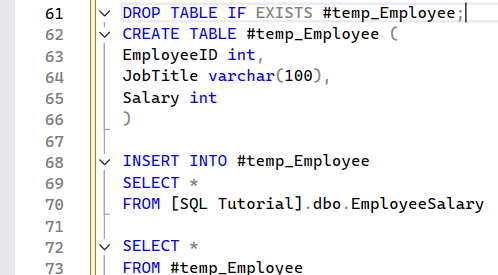
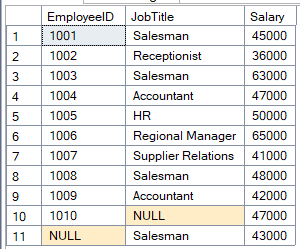
* Temp tables are stored in the **tempdb** system database (Databases -> System Database -> tempdb). **tempdb itself is a permanent system database**. It exists **as long as the SQL Server instance is running**, not just for the duration of your query. Automatically recreated **fresh every time SQL Server restarts**.
* If you're in SQL Server Management Studio (SSMS) and close the window, the temp table is gone—but tempdb is still there.
* You can create indexes on them.
* They persist across multiple statements, unlike CTEs (which only exist within one query).

Creating a temp table is like creating a normal table, the only difference is the ‘#’ and the naming convention that includes ‘temp’. “DROP TABLE..’ line means delete the table and it is important because it allows you to re-create the temp table many times. If you do not use this line, if you execute again after first time creating temp table, there will be an error

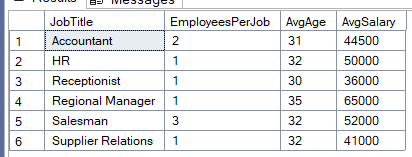


ADDITIONAL: The create table code lines does not need to be executed every time just like in CTE. You can just select the query and execute

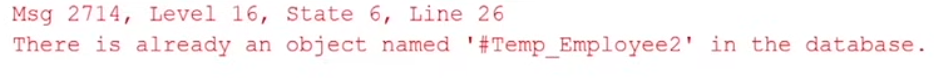
BIG USE CASE: Inserting data from other existing table to temp tables

ANOTHER USE CASE: like in CTE, you can store procedures in the temp table so that there would not be a need to run that code every time.

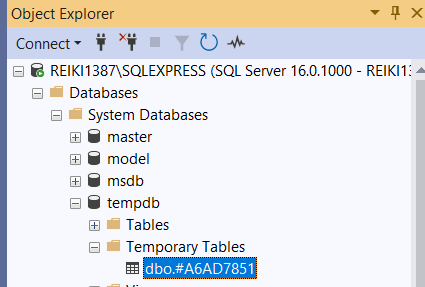
 

DO NOT forget the “Drop table..” line to prevent error like below when executing the second time



If you create a **temporary table** (like #TempTable) and it’s active in your current session, you’ll see it appear

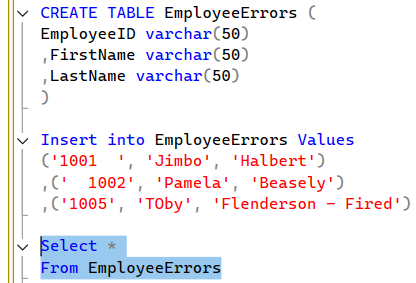
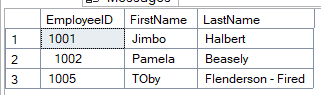
under:



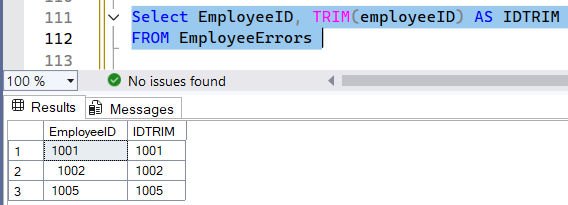
But this only happens **while the session is open**—once it ends, the temp table disappears.

String Functions – TRIM, LTRIM, RTRIM, Replace, Substring, Upper, Lower

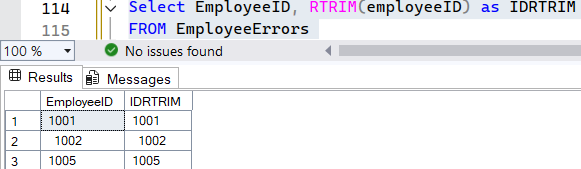
These datas have many errors in it

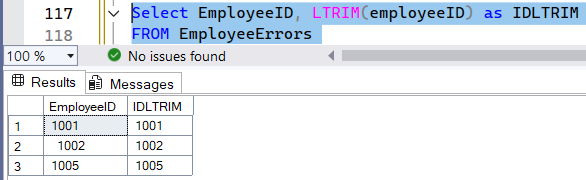
TRIM- Takes out spaces to the left and right of the datas in selected column



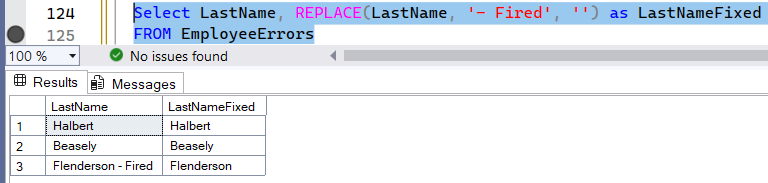
RTRIM- takes out space ONLY AT THE RIGHT



LTRIM- takes out spaces ONLY AT THE LEFT



REPLACE – argument(column, string to replace, what to replace with)



SUBSTRING

In SQL, SUBSTRING is a **string function** that lets you extract part of a text (string) based on a **starting position** and a **length**.

### ✅ Syntax:

SUBSTRING(expression, start\_position, length)

* expression: the string you want to extract from
* start\_position: where to begin (1-based index)
* length: how many characters to take. **THIS IS LENGTH NOT INDEX**

IMPORTANT : **string indexing in SQL starts at 1**, **not** at 0.

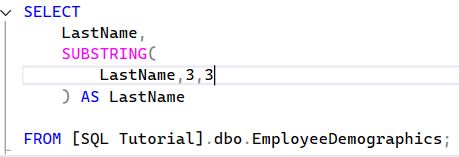
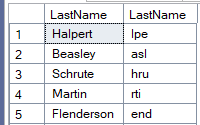
### 🔍 Example:

*SELECT SUBSTRING('Michael Scott', 1, 7) AS FirstName;*

*-- Result: 'Michael'*

This grabs 7 characters starting at position 1 from 'Michael Scott'.

The last argument is NOT INDEX but LENGTH. Start from index 3 then take 3 letters including the index

### 🧠 Real Example with a Table:

Say you have a column like this:

| **FullName** |
| --- |
| Michael Scott |
| Pam Beesly |
| Dwight Schrute |

If you wanted to extract the first name:

*SELECT*

*FullName,*

*SUBSTRING(FullName, 1, CHARINDEX(' ', FullName) - 1) AS FirstName*

*FROM Employees;*

This:

1. Finds the position of the space using CHARINDEX(' ', FullName)
2. 'Michael Scott' → space is at position **8, returns 8**
3. **SUBSTRING(FullName, 1, 8 - 1) -> SUBSTRING(FullName, 1, 7)**
4. Starts at position 1 (the first character, 'M')
5. Takes **7 characters** → 'Michael'

### Summary:

* SUBSTRING is like slicing a string: pick where to start and how long to go
* Often used with functions like CHARINDEX or LEN to dynamically extract names, codes, or IDs from a string
  1. CHARINDEX is a **string function** in SQL that tells you the **position of a substring** inside another string.

### ✅ Syntax:

CHARINDEX(substring, string)

* substring: what you're looking for
* string: where you're searching

It returns the **starting position** (1-based index) of the first occurrence of the substring. If it’s not found, it returns 0.

### 🔍 Example:

SELECT CHARINDEX('a', 'Michael Scott') AS Position;

-- Returns: 5 (the first 'a' is at position 5)

SELECT CHARINDEX('Scott', 'Michael Scott') AS Position;

-- Returns: 9

SELECT CHARINDEX('x', 'Michael') AS Position;

-- Returns: 0 (not found)

### Summary:

CHARINDEX helps you **find positions inside strings**, often combined with SUBSTRING to split names, emails, IDs, etc. It’s SQL’s version of "find the first occurrence of this text."

* 1. LEN in SQL is a function that returns the **number of characters** in a string—basically, the string’s length

### ✅ Syntax:

LEN(expression)

* expression: the string you want to measure

### 🔍 Example:

SELECT LEN('Michael Scott') AS Length;

-- Returns: 13

SELECT LEN(' ') AS Length;

-- Returns: 0 (because it's just a space and SQL Server trims trailing spaces)

### 🧠 Important Note (SQL Server-specific):

In SQL Server, LEN() **ignores trailing spaces**, but **not** leading ones.

SELECT LEN('Hello ') -- Returns: 5 (trailing spaces are ignored)

SELECT LEN(' Hello') -- Returns: 8 (leading spaces are counted)

### ✅ Real-World Example: Get Last Name

You can use LEN to help extract the **last name** from a full name:

SELECT

FullName,

SUBSTRING(

FullName,

CHARINDEX(' ', FullName) + 1,

LEN(FullName)

) AS LastName

FROM Employees;

* CHARINDEX(' ', FullName) + 1 → start just after the space, returns
* LEN(FullName) → returns total length of sting. extract the rest of the string

### Summary:

* LEN() tells you how many characters are in a string.
* It’s useful for string manipulation, validations, or trimming logic.
* Just remember: **trailing spaces are ignored** in SQL Server.

FUZZY MATCHING

**Fuzzy matching** in SQL refers to comparing text values that are **similar but not exactly the same**, to find matches that might have **typos, different formatting, or slight variations**.

It's helpful when you're dealing with **messy data**, like trying to match:

* "Jon" and "John"
* "Acme Inc" and "Acme Incorporated"
* "123 Elm St." and "123 Elm Street"

### 🧠 Why can't you use “=” sign ?

SQL's regular equality operator (=) only returns true if the strings are exactly the same. Fuzzy matching helps you **find close matches**, not just identical ones.

### ✅ Common Approaches for Fuzzy Matching in SQL Server:

#### 1. SOUNDEX()

* Encodes words by how they **sound**, not how they’re spelled.

*SELECT \**

*FROM Customers*

*WHERE SOUNDEX(Name) = SOUNDEX('Jon');*

* Matches names like "Jon", "John", "Juan" (if they sound similar)

#### 2. DIFFERENCE()

* Compares the **SOUNDEX** of two strings and gives a score from 0 to 4  
  (4 = strong match)

SELECT DIFFERENCE('Jon', 'John'); -- Returns: 4

You can use it in a WHERE clause:

*SELECT \**

*FROM Customers*

*WHERE DIFFERENCE(Name, 'John') >= 3;*

#### 3. **Levenshtein Distance** (Custom function)

* Counts the **number of changes** (inserts, deletes, substitutions) needed to turn one string into another.
* Not built-in, but you can write or install a **user-defined function** (UDF) for it.

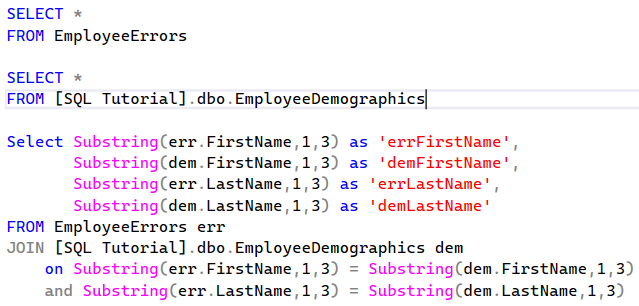
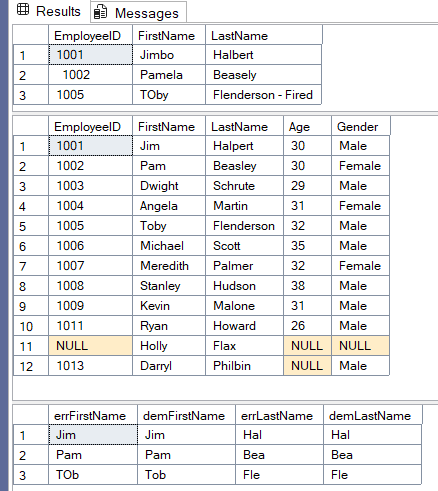
### ✅ When to Use Fuzzy Matching

* Cleaning or deduplicating names or addresses
* Joining data across systems with inconsistent naming
* Matching user input to known values (e.g., search, forms)

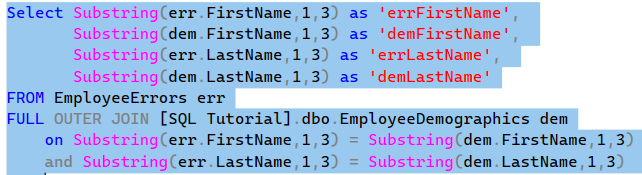
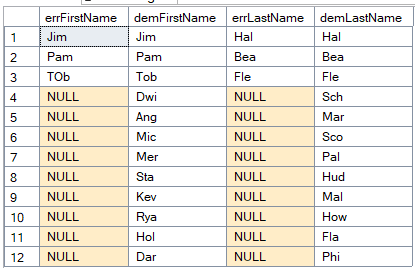
### ⚠️ A Few Things to Watch Out For

* **Fuzzy logic can be slow on large datasets—**consider indexing or limiting the scope
* Not always accurate—review matches manually if quality is critical
* For better control, consider **SQL + Python** or **SQL + Full-Text Search** for more advanced matching

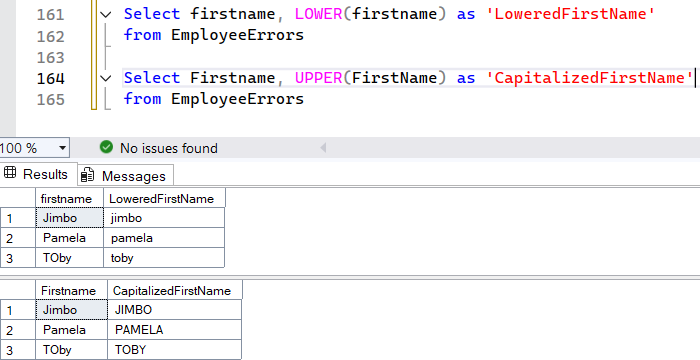
Example: Matching people in two tables using substrings fuzzy matching



Looking at the actual joining of table, only 3 people matches with the two table. NULL was indicated on the rows that do not have match

UPPER and LOWER



**STORED PROCEDURES**

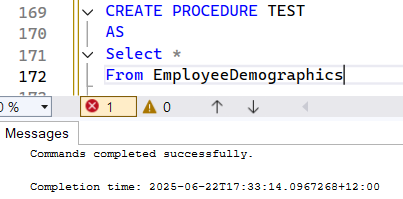
A **stored procedure** in SQL is a **pre-written, saved block of code**—basically a script stored in the database that you can **reuse** and **run on demand**. Decreases network traffic. Improve performance

Think of it like a **function or method** in programming, but for your database.

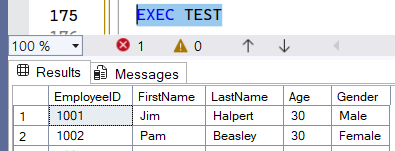
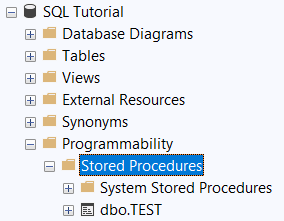
### What a stored procedure can do:

* Run SELECT, INSERT, UPDATE, DELETE
* Use variables, parameters, loops, if/else
* Call other procedures
* Handle transactions (BEGIN TRAN, ROLLBACK, COMMIT)
* Return data or just perform actions

Simple Example: We stored simple select statement that displays data from a table



It will be stored in the database- programmability- stored procedures. Sometimes needs refresh. To run what is in the procedure just use ‘EXEC’ command and the name of the procedure



### Creating new procedure.

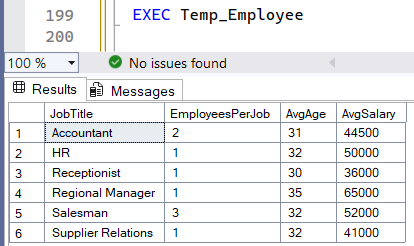
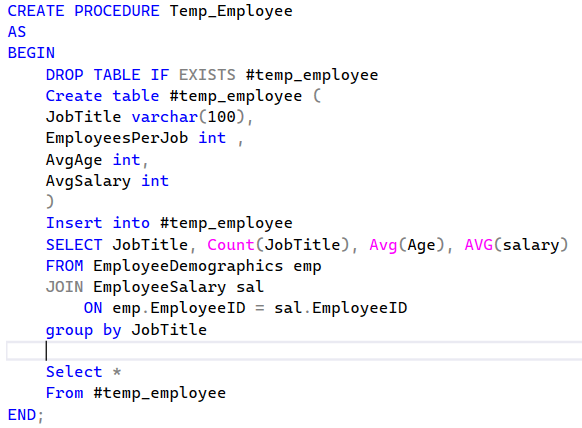
### You can include both the CREATE TABLE and the INSERT into the temp table **inside** a stored procedure.

### But here's the catch:

If you're using a **local temp table** (#temp\_employee), it only exists **in the same session** that created it. So:

* If you create the temp table **inside a stored procedure**, and that procedure ends, the temp table is **gone** unless you use it in the **same session** immediately.
* You can't create a temp table in one session and use it in another (unless it's a **global temp table**, like ##temp\_employee).

1. Always use BEGIN ... END inside procedures to avoid syntax errors, especially when running multiple statements.
2. You may also want to add a SELECT \* FROM #temp\_employee at the end if you want to return the data.

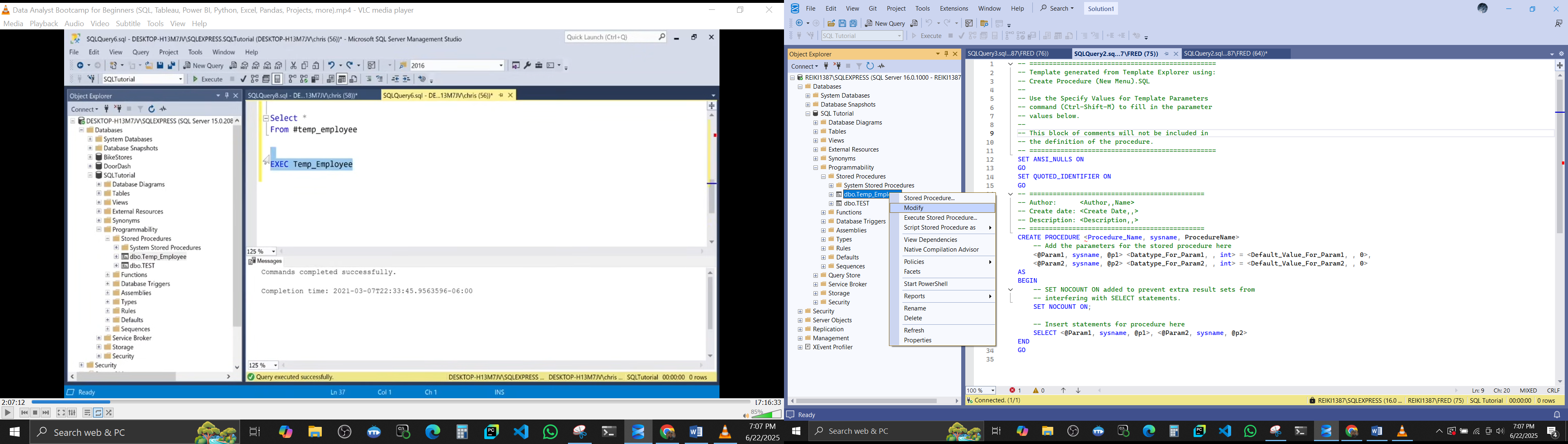


If you're using a **local temp table** (#temp\_employee), it only exists **in the same session** that created it. So:

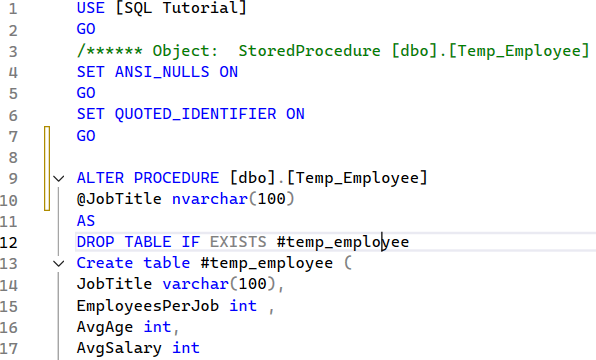
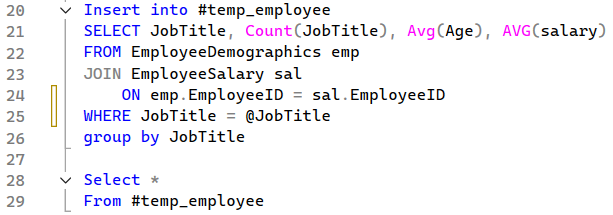
* If you create the temp table **inside a stored procedure**, and that procedure ends, the temp table is **gone** unless you use it in the **same session** immediately.
* You can't create a temp table in one session and use it in another (unless it's a **global temp table**, like ##temp\_employee).

MODIFYING STORED PROCEDURES to add parameters

1. Right click and choose modify



1. Adding parameters (Line 10 and Line 25 are added). You can add as many parameter as you can and can put it anywhere in the stored procedure

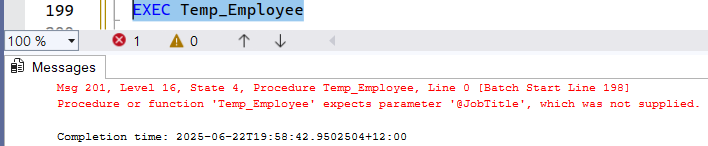
### What are parameters in a stored procedure?

A **parameter** is like a variable that you pass into a stored procedure when you execute it. It lets you make the procedure behave differently depending on what you pass in.

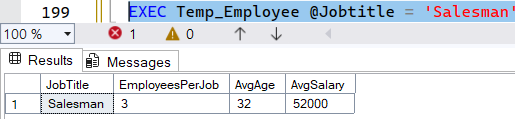
Instead of hardcoding values inside the procedure (like a fixed job title), you use a **placeholder** (parameter) that you can swap dynamically.

BELOW NOTE: After putting parameter, execute it so that the changes will take effect

Now when you try to execute the stored procedure again it will have an error because now the procedure need a certain parameter



You need to specify the parameter in the execution



CODE EXPLANATIONS

### 🔹 SET ANSI\_NULLS ON

This controls **how SQL Server treats NULL values in comparisons**.

* When ANSI\_NULLS is **ON** (recommended and default):

*WHERE column = NULL -- returns nothing*

*WHERE column IS NULL -- correct*

* When ANSI\_NULLS is **OFF**:

*WHERE column = NULL -- this would actually work (non-standard)*

**Why this matters:**  
SQL Server follows the ANSI SQL standard by default, which says NULL is not equal to anything—not even another NULL. Keeping this ON is best practice.

### 🔹 SET QUOTED\_IDENTIFIER ON

This controls how SQL Server treats **quoted strings and identifiers**.

* When it's **ON**:
  + Strings must be in single quotes 'string'
  + Double quotes "TableName" are treated as object names (useful if the name has spaces or reserved words)

*SELECT \* FROM "Employee Table" -- valid if QUOTED\_IDENTIFIER is ON*

* When **OFF**:
  + Double quotes are treated like single quotes (not standard)

This setting helps enforce compatibility with ANSI SQL and tools like SSMS or EF.

### 🔹 nvarchar(100)

This is a data type that stands for:

* \*\*n\*\* = **national** (i.e., Unicode support)
* \*\*varchar\*\* = variable-length string
* (100) = max 100 characters

So nvarchar(100) means:

A Unicode string that can hold up to 100 characters.

Use nvarchar if you need to support international characters or Unicode (like emojis, Chinese, Arabic, etc). Otherwise, varchar works fine for plain English text.