REPORTING AGGREGATE DATA USING GROUP FUNCTIONS

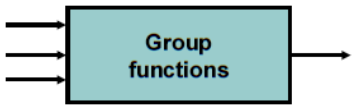
The previous topic covered some of the available single-row functions within T-SQL. As already mentioned in Topic 4, T-SQL has 4 different categories of built-in functions:

- *Scalar Functions*: also known as single row functions (covered in the previous topic)
- *Aggregate Functions*: also known as multiple row functions (covered in this topic)
- *Rowset Functions*: this category of functions will return an object can be used like table references in an SQL statement
- *Ranking Functions*: this type of function will return a value for each row in a partition which is related to the ranking of the row.

Once again, I would like to mention that we will be only covering the first two type of built-in functions in during the course of this subject. In both cases, there will not be ab exhaustive coverage of all the available functions.

What is an ***Aggregate Function***?

An aggregate function is sometimes also referred to as a *multiple-row function* or a *group function*. The aim of these type of functions is to operate on a set of arguments/rows and give one result per group. These type of functions differ from single row functions, as they operate on a number of rows and not on a single row.



Aggregate functions introduce new syntax to our default SELECT statement:

```
SELECT [column,] group_function(column)
FROM table
[WHERE condition/s]
[GROUP BY group_by_expression]
[HAVING group_condition]
[ORDER BY {COLUMN, EXPR, NUMERIC_POSITION} {ASC|DESC}];
```

NOTE:

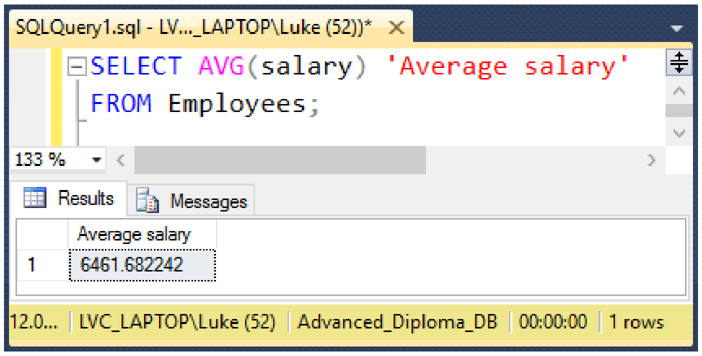
- All aggregate functions that are available in T-SQL, except for COUNT will ignore any null values. Null values are to be substituted using some single-row functions in order to be considered by multiple-row functions

- All aggregate functions are deterministic, therefore they will return the same value any time that they are called with the same specific inputs
- Two important keywords which are commonly used with aggregate functions are:
    - ALL: meaning that the aggregate function will be applied to all values
    - DISTINCT: meaning that the aggregate function will be applied only one unique instances of a value, regardless of the number of times that the value exists

The list of aggregate functions that we will be covering in this subject is not exhaustive, amongst which we can find:

- *AVERAGE Function*: [AVG([ALL | DISTINCT] expression)]
  This function will return the average value of n, ignoring any null values

Example 1: Write a query that will display the average salary of all the employees found in the employees table.



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT AVG(salary) 'Average salary'
FROM Employees;
```

133 %

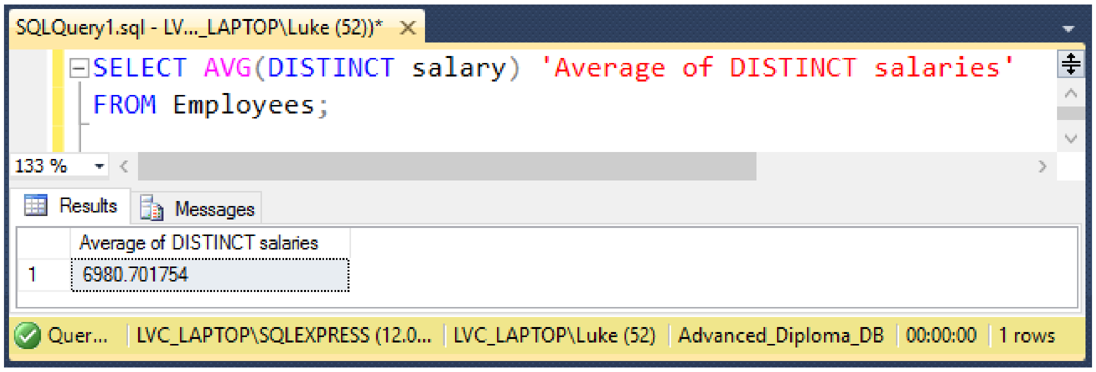
Results | Messages

	Average salary
1	6461.682242

12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows

Figure 1 – Result of example 1: average calculated of the salaries in the employees table.

Example 2: Write a query that will display the average of the different salaries that exist in the employees table:



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT AVG(DISTINCT salary) 'Average of DISTINCT salaries'
FROM Employees;
```

133 %

Results | Messages

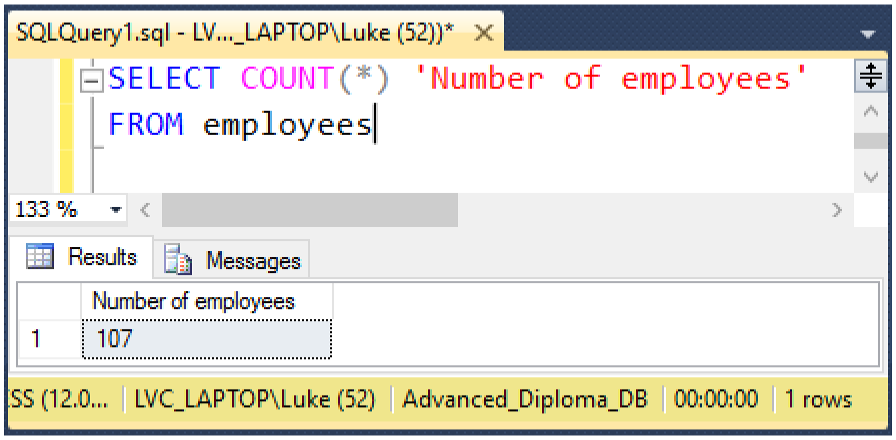
	Average of DISTINCT salaries
1	6980.701754

Quer... | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows

Figure 2 – Result of example 2: average of the distinct salaries in the employees table

- *COUNT Function*: [COUNT({ * | [ALL | DISTINCT] expression})]
  This will return the number of rows where the expression evaluates to something other than null. This function has three different formats:
  - COUNT(*) – this will return the number of rows that satisfy the SELECT statement, including duplicate rows and rows containing null values. If there is a WHERE clause the number of rows that satisfy this condition are returned.
  - COUNT(expression) – returns the number of non-null values that are in the column identified by the expression
  - COUNT(DISTINCT expression) – returns the number of unique, non-null values that are in the column identified by the expression

Example 3: Write a query that will display the number of employees in the employees table

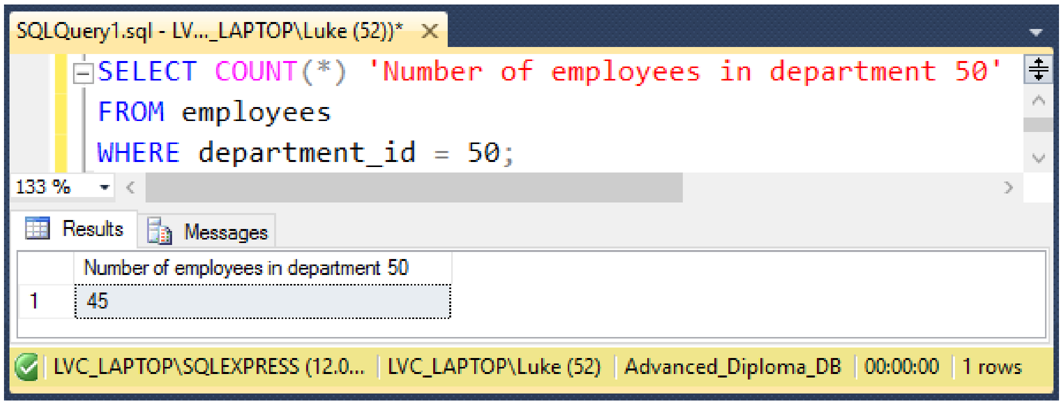


```
SELECT COUNT(*) 'Number of employees'
FROM employees
```

```
   Number of employees
1  107
```

Figure 3 - Result of example 3: number of employees in the employees table

Example 4: Write a query that will return the number of employees in department 50

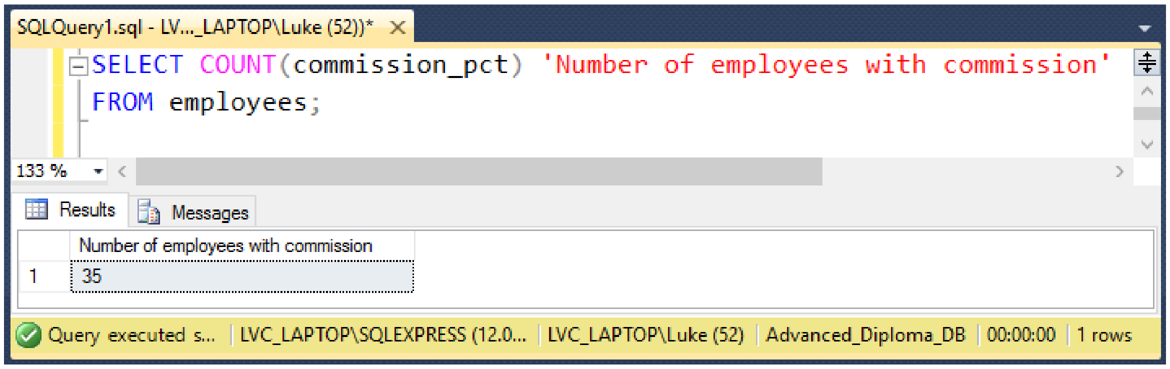


```
SELECT COUNT(*) 'Number of employees in department 50'
FROM employees
WHERE department_id = 50;
```

```
   Number of employees in department 50
1  45
```

Figure 4 - Result of example 4: number of employees in department 50

Example 5: Write a query that will display the number of people who have a commission (their commission value is not null)



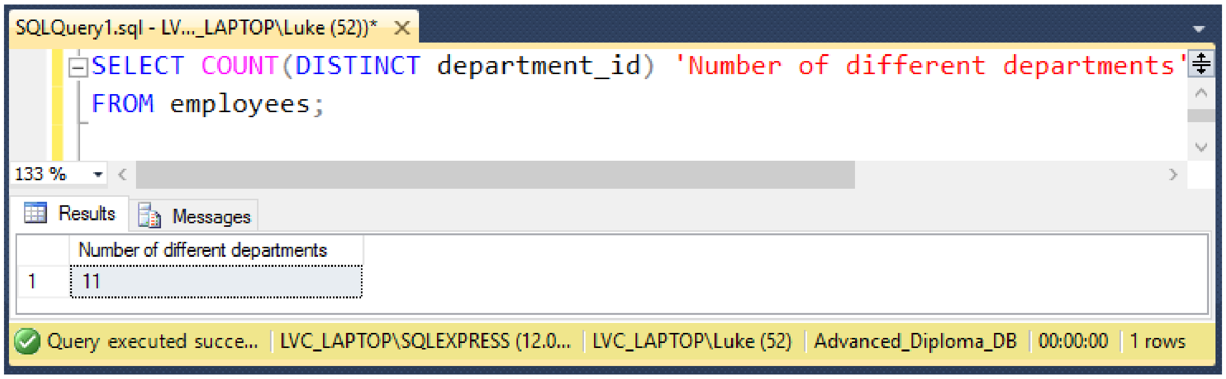
```
SELECT COUNT(commission_pct) 'Number of employees with commission'
FROM employees;
```

	Number of employees with commission
1	35

Query executed s... | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows

Figure 5 - Result of example 5: number of employees with a commission

Example 6: Write a query that will display the number of distinct department numbers in the employees table



```
SELECT COUNT(DISTINCT department_id) 'Number of different departments'
FROM employees;
```

	Number of different departments
1	11

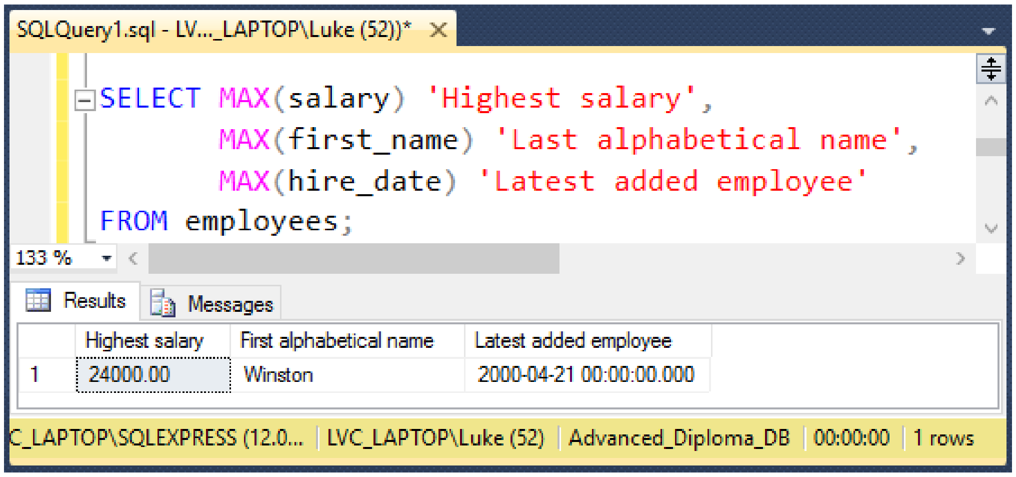
Query executed succe... | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows

Figure 6 - Result of example 6: the different department numbers in the employees table

- *COUNT_BIG Function*: [COUNT_BIG({ * | [ALL | DISTINCT] expression})]
  This function is the equivalent of the COUNT function, with the only difference being the data type that is returned. COUNT_BIG always returns a *bigint* data type, while COUNT always returns an *int* data type.

- *MAXIMUM Function*: [MAX([ALL | DISTINCT] expression)]
  This function returns the maximum value in an expression. Note that this function can be applied to different data types, including strings, numbers and even dates

Example 7: Write a query that will display the highest salary, largest name in alphabetical order and the date of the last employed employee.



```
SELECT MAX(salary) 'Highest salary',
       MAX(first_name) 'Last alphabetical name',
       MAX(hire_date) 'Latest added employee'
FROM employees;
```

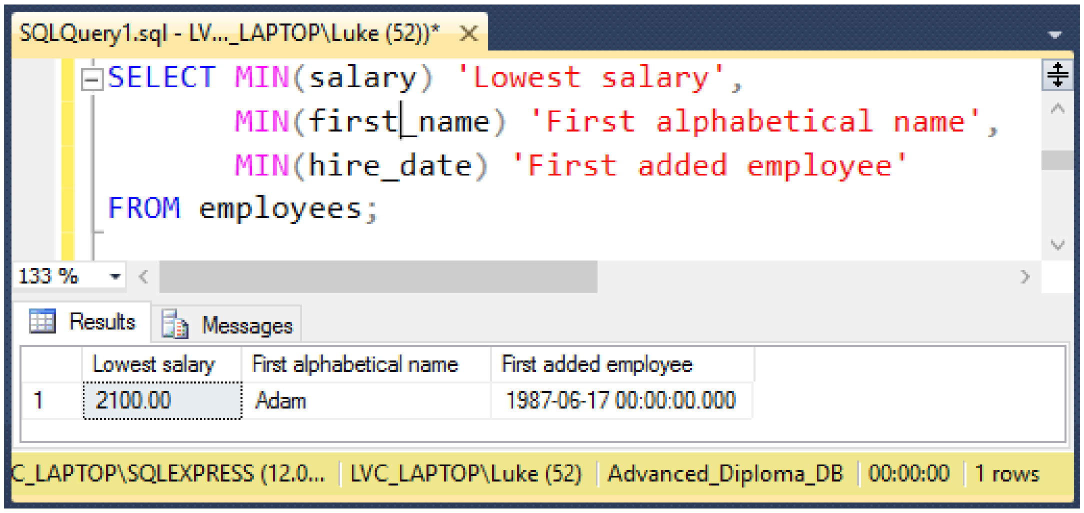
	Highest salary	First alphabetical name	Latest added employee
1	24000.00	Winston	2000-04-21 00:00:00.000

```
C_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows
```

Figure 7 - Result of example 7: the maximum values for the salary, name and hire_date columns

- *MINIMUM Function*: [MIN([ALL | DISTINCT] expression)]
  This function returns the minimum value in an expression and ignores any null values. As in the case of MAX function, this function can be applied to different data types.

Example 8: Write a query that will display the lowest salary, smallest name in alphabetical order and the date that the first employee was hired in.



```
SELECT MIN(salary) 'Lowest salary',
       MIN(first_name) 'First alphabetical name',
       MIN(hire_date) 'First added employee'
FROM employees;
```

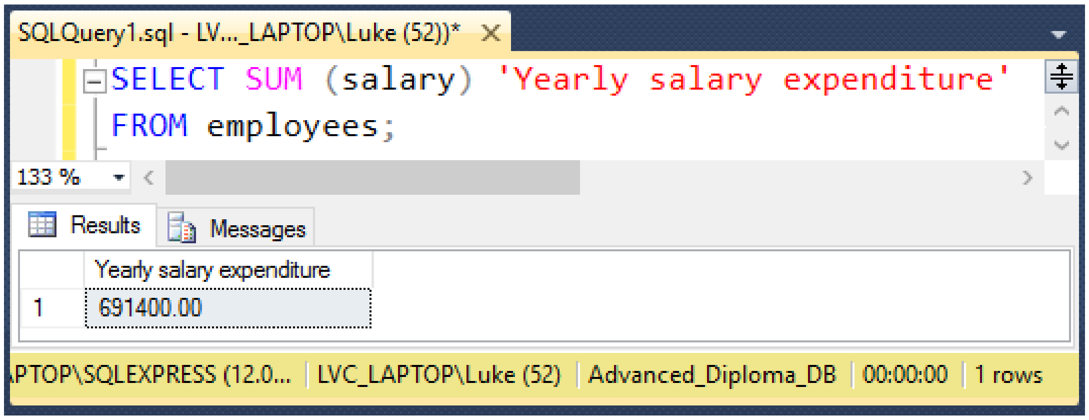
	Lowest salary	First alphabetical name	First added employee
1	2100.00	Adam	1987-06-17 00:00:00.000

```
C_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows
```

Figure 8 - Result of example 8: the minimum values for the salary, name and hire_date columns

- *SUM Function*: [SUM([ALL | DISTINCT] expression)]
  The SUM function will return the sum of all the values in the expression, whilst ignoring all null values

Example 9: Write a query that will display the total salary of all the employees within the company



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT SUM (salary) 'Yearly salary expenditure'
FROM employees;
```

133 %

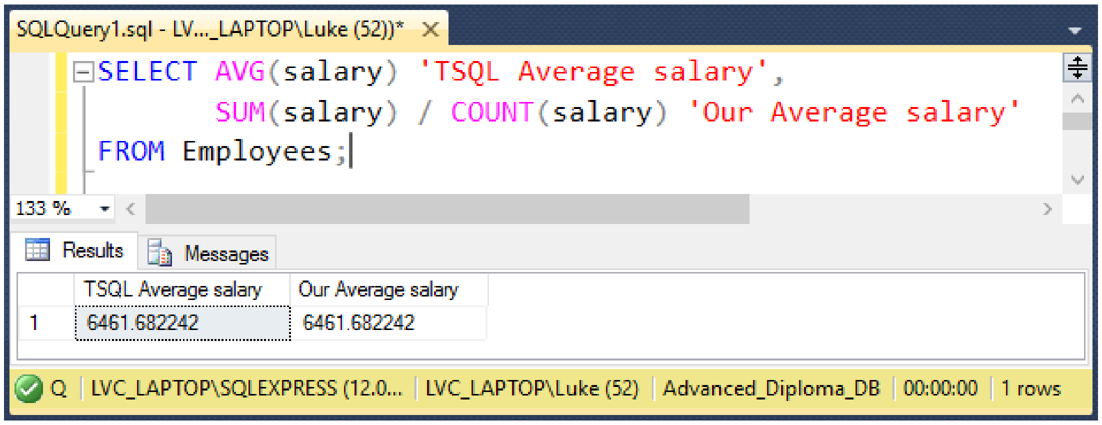
Results | Messages

	Yearly salary expenditure
1	691400.00

PTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows

Figure 9 - Result of example 9: the yearly salary of all employees

Example 10: Write a query that will display two columns 'TSQL Average salary' and 'Our Average salary'. The first column should make use of the Average function while the other column should be calculated using any combination of function which give the same result



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT AVG(salary) 'TSQL Average salary',
       SUM(salary) / COUNT(salary) 'Our Average salary'
FROM Employees;
```

133 %

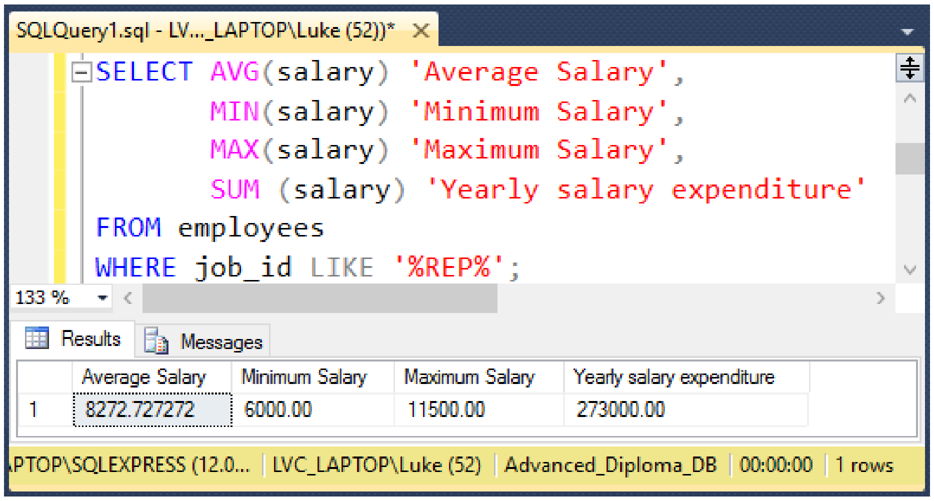
Results | Messages

	TSQL Average salary	Our Average salary
1	6461.682242	6461.682242

Q | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows

Figure 10 - Result of example 10: Average salary calculated using different methods

Example 11: Write a query that will return the average salary, minimum salary, maximum salary and sum of all the salaries of all the people who have REP in their job_id



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT AVG(salary) 'Average Salary',
       MIN(salary) 'Minimum Salary',
       MAX(salary) 'Maximum Salary',
       SUM (salary) 'Yearly salary expenditure'
FROM employees
WHERE job_id LIKE '%REP%';
```

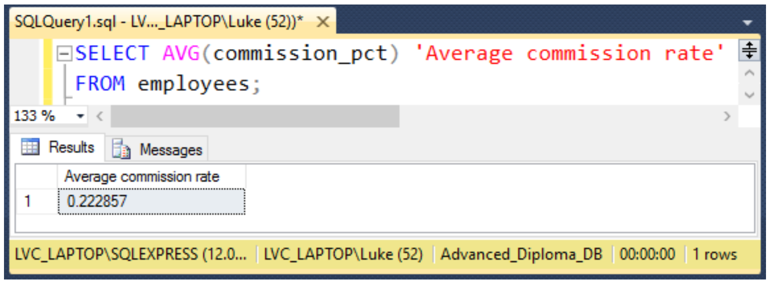
	Average Salary	Minimum Salary	Maximum Salary	Yearly salary expenditure
1	8272.727272	6000.00	11500.00	273000.00

PTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows

Figure 11 - Result of Example 11: average, minimum, maximum and sum of salaries

NOTE: While working with such functions it is very important to handle NULL values correctly. Values which are set as NULL are normally disregarded by aggregate functions, therefore it is very important that you handle them beforehand.

Example 12: In this example you are to try and determine the difference between the two statements

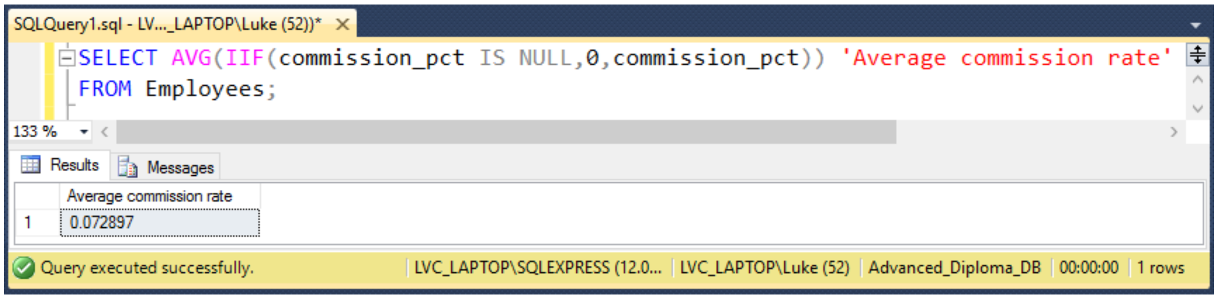


SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT AVG(commission_pct) 'Average commission rate'
FROM employees;
```

	Average commission rate
1	0.222857

LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT AVG(IIF(commission_pct IS NULL,0,commission_pct)) 'Average commission rate'
FROM Employees;
```

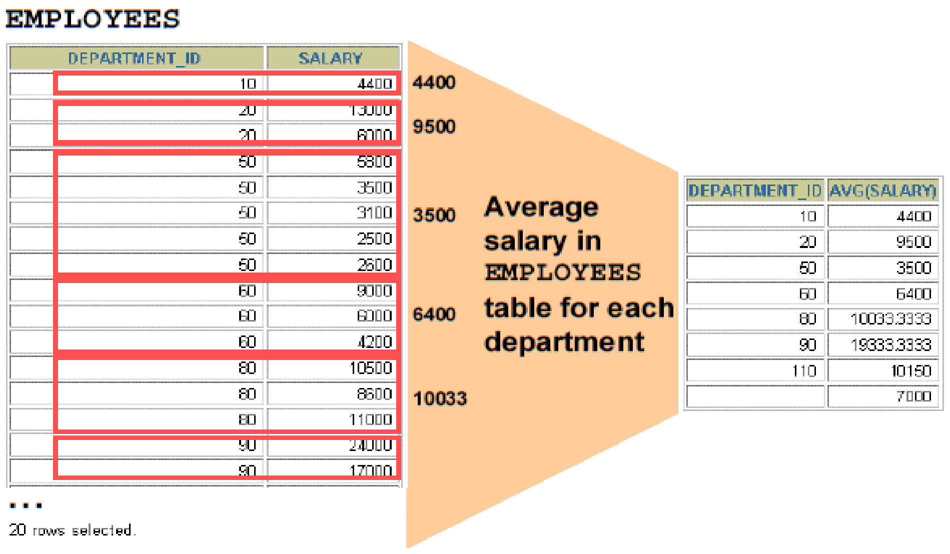
	Average commission rate
1	0.072897

Query executed successfully. | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows

Figure 12 -Result of example 12: average commission rate

The GROUP BY clause (Single columns)

Up till now we have applied the aggregate functions on the whole table as one single large group of information. In certain situations we will be required to divide the table into smaller groups and this requires the use of the GROUP BY clause. This clause can be used to return summary information for each particular group in the table.

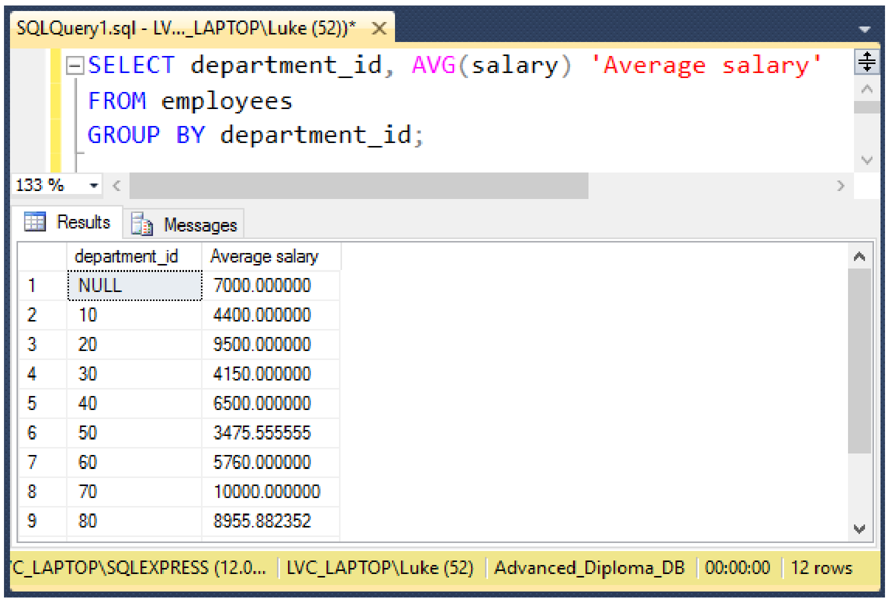

EMPLOYEES

DEPARTMENT_ID	SALARY
10	4400
20	13000
20	6000
50	5800
50	3500
50	3100
50	2500
50	2600
60	9000
60	6000
60	4200
80	10500
80	8600
80	11000
90	24000
90	17000

...
20 rows selected.

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

The above query clearly pin points an example where such a clause can be applied. As it is being demonstrated, the query requires the average salary per department and for this reason the GROUP BY clause needs to be used.



```
SELECT department_id, AVG(salary) 'Average salary'
FROM employees
GROUP BY department_id;
```

	department_id	Average salary
1	NULL	7000.000000
2	10	4400.000000
3	20	9500.000000
4	30	4150.000000
5	40	6500.000000
6	50	3475.555555
7	60	5760.000000
8	70	10000.000000
9	80	8955.882352

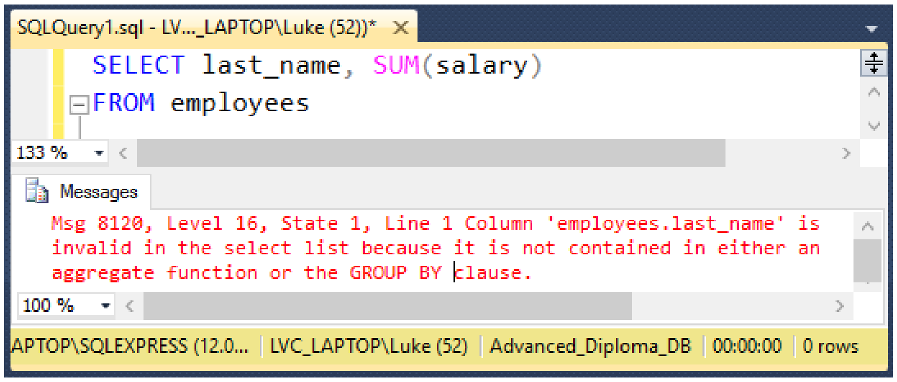
C_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 12 rows

Figure 13 - Average salary per department

NOTE:

- In which order is a query with a GROUP BY clause evaluated?
    - The SELECT clause specifies the columns to be retrieved (In the example of Figure 13, department number and average of all salaries in the particular department)
    - The FROM clause specifies the tables that the database must access (In the example of Figure 13 – the employees table)
    - The WHERE clause specifies the rows to be retrieved
    - The GROUP BY clause specifies how the rows should be grouped (In the example of Figure 13, it should be via the department number)

- If a group function is included in the SELECT clause, you cannot select individual expressions as well, unless the individual expression appears in the GROUP BY clause. If you fail to include the individual expression in the GROUP BY clause SSMS will return an error as shown below.



```sql
SELECT last_name, SUM(salary)
FROM employees
```

```
Msg 8120, Level 16, State 1, Line 1 Column 'employees.last_name' is
invalid in the select list because it is not contained in either an
aggregate function or the GROUP BY clause.
```

SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
133 %
Messages
100 %
APTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 0 rows

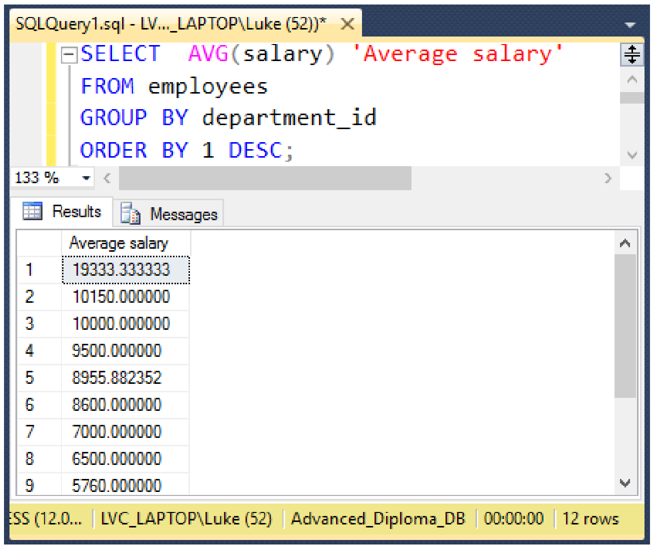
Figure 14 - Error in statement as last_name should be included in GROUP BY clause

- When the WHERE clause is used, rows can be excluded before the actual dividing of the rows takes place. Note that the WHERE is evaluated before the GROUP BY clause

- If the expressions in the SELECT clause are given an alias, this cannot be used in the GROUP BY clause. The GROUP BY clause needs to include the column name and not the alias.

- The expression which is listed in the GROUP BY clause, does not necessarily need to be included in the SELECT clause. Although this is possible the resultant result will not be so clear if this approach is taken.

Example 13: Write a query that will display the average salary of each department but do not include the department number in the final result. The highest average salary should be displayed first



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```sql
SELECT  AVG(salary) 'Average salary'
FROM employees
GROUP BY department_id
ORDER BY 1 DESC;
```

133 %

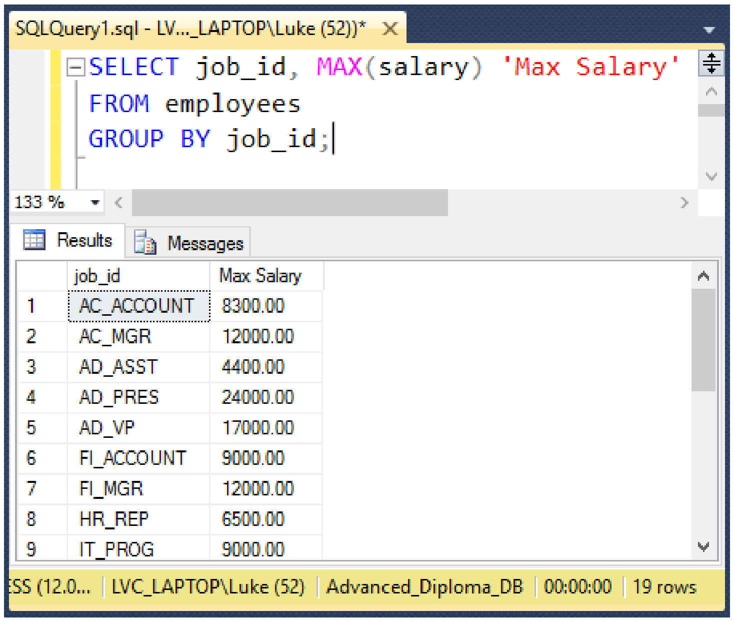
Results | Messages

	Average salary
1	19333.333333
2	10150.000000
3	10000.000000
4	9500.000000
5	8955.882352
6	8600.000000
7	7000.000000
8	6500.000000
9	5760.000000

ESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 12 rows

Figure 15 - Result of example 13: average salary of each department

Example 14: Write a query that will display the maximum salary for each job title which is available in the employees table



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```sql
SELECT job_id, MAX(salary) 'Max Salary'
FROM employees
GROUP BY job_id;
```

133 %

Results | Messages

	job_id	Max Salary
1	AC_ACCOUNT	8300.00
2	AC_MGR	12000.00
3	AD_ASST	4400.00
4	AD_PRES	24000.00
5	AD_VP	17000.00
6	FI_ACCOUNT	9000.00
7	FI_MGR	12000.00
8	HR_REP	6500.00
9	IT_PROG	9000.00

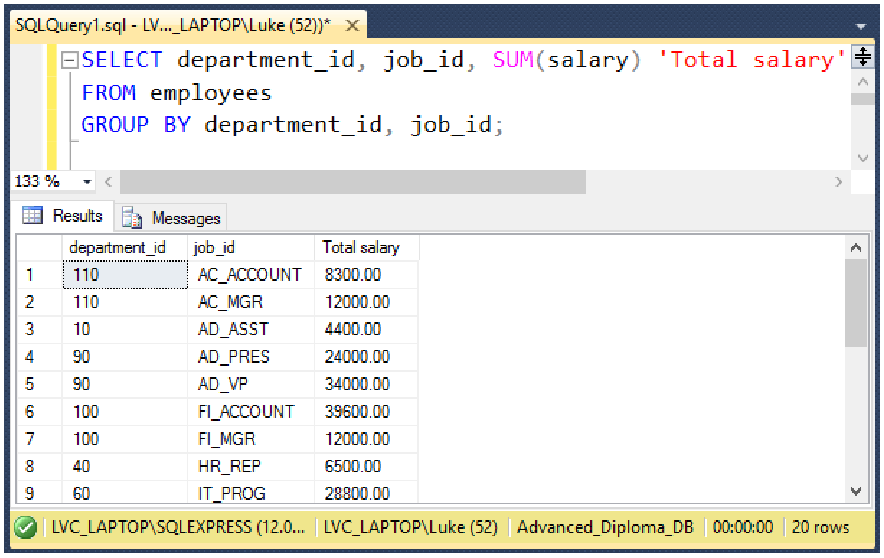
ESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 19 rows

Figure 16 - Result of example 14: The maximum salary of each job title

The GROUP BY clause (Multiple columns)

In certain situations grouping results by using one single expression is not enough. There will be instances where summary results for groups and sub groups will be required. This involves the use of multiple columns in the GROUP BY clause. The default sort order is determined by the order in which the columns are placed in the ORDER BY.

Example 15: Write a query that will display the total salary for each job_id in each department



```
SELECT department_id, job_id, SUM(salary) 'Total salary'
FROM employees
GROUP BY department_id, job_id;
```

	department_id	job_id	Total salary
1	110	AC_ACCOUNT	8300.00
2	110	AC_MGR	12000.00
3	10	AD_ASST	4400.00
4	90	AD_PRES	24000.00
5	90	AD_VP	34000.00
6	100	FI_ACCOUNT	39600.00
7	100	FI_MGR	12000.00
8	40	HR_REP	6500.00
9	60	IT_PROG	28800.00

SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
133 %
Results
Messages
LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 20 rows

Figure 17 - Result of example 15: the total salary for each job_id in each department

How is the above query evaluated?

- SELECT clause specifies the columns to be retrieved (department number, job id, sum of all salaries in the group that you specified in the GROUP BY clause)
- FROM clause specifies the tables that the database must access (Employees)
- GROUP BY clause specifies how the rows should be grouped (first by department number, second by job id in the department number groups)

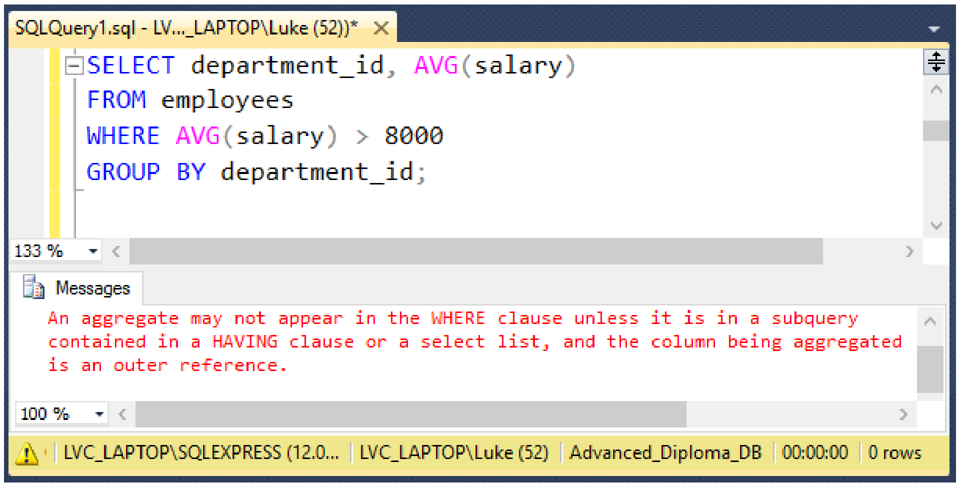
The HAVING clause

Another important clause that is to be introduced in this topic is the HAVING clause. This clause is the equivalent of the WHERE clause, but it is to be used to specify which groups are to be displayed instead of which rows. When an SQL statement includes the HAVING clause, SSMS evaluated the query as follows:

- Rows are grouped together
- The group function is applied to the group
- The groups that match the HAVING condition are displayed

The use of the HAVING is very important as SSMS does not allow the GROUP functions to be included in the WHERE clause. Any conditions that include a group function are to be listed in the HAVING clause.

Example16: In the below screenshots, the first one is incorrect as the condition AVG(salary)>8000 cannot be placed in the WHERE clause. The second screenshot shows the correct statement which includes the HAVING clause. This statement will display all the departments whose average salary is greater than 8000



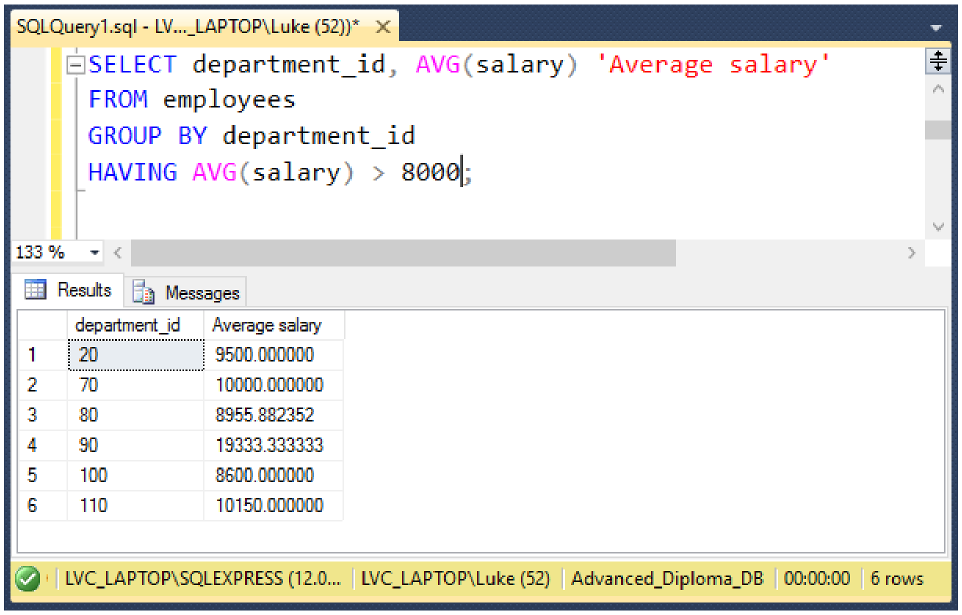
SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT department_id, AVG(salary)
FROM employees
WHERE AVG(salary) > 8000
GROUP BY department_id;
```

Messages

```
An aggregate may not appear in the WHERE clause unless it is in a subquery
contained in a HAVING clause or a select list, and the column being aggregated
is an outer reference.
```

LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 0 rows



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT department_id, AVG(salary) 'Average salary'
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 8000;
```

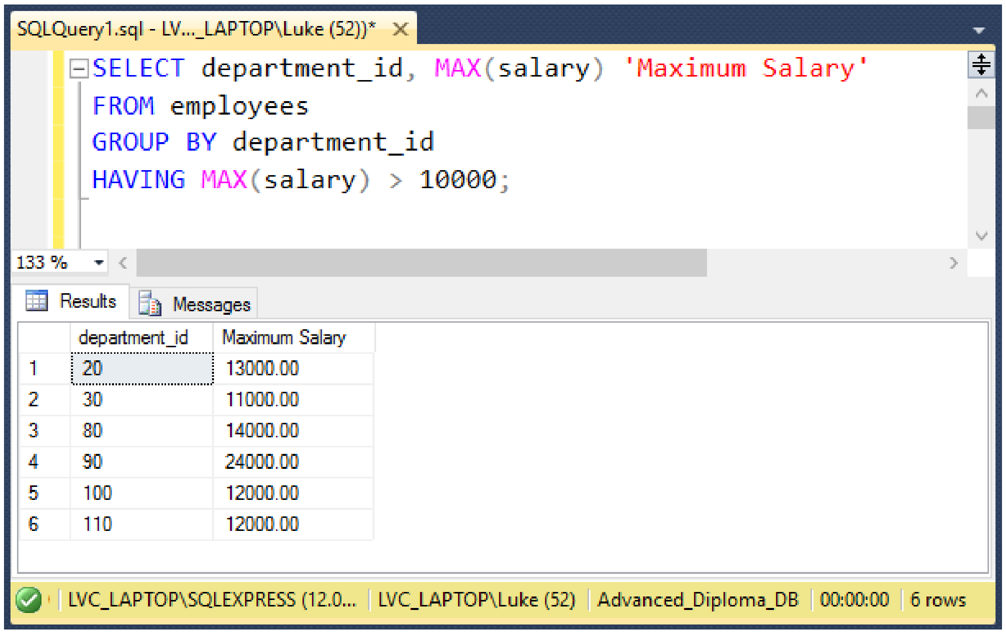
Results | Messages

	department_id	Average salary
1	20	9500.000000
2	70	10000.000000
3	80	8955.882352
4	90	19333.333333
5	100	8600.000000
6	110	10150.000000

LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 6 rows

Figure 18 - result of example 16: the second screenshot is correct as it uses the HAVING clause

__Example 17:__ Write a query that displays the department number and maximum salaries for those departments with a maximum salary that is greater than 1000.



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT department_id, MAX(salary) 'Maximum Salary'
FROM employees
GROUP BY department_id
HAVING MAX(salary) > 10000;
```

133 %

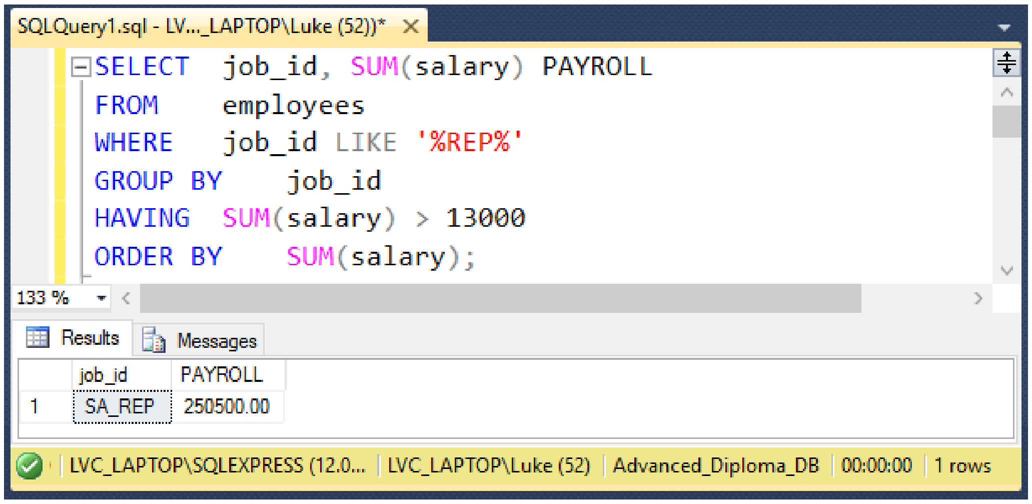
Results | Messages

	department_id	Maximum Salary
1	20	13000.00
2	30	11000.00
3	80	14000.00
4	90	24000.00
5	100	12000.00
6	110	12000.00

LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 6 rows

Figure 19 - Result of example 17: departments with a maximum salary exceeding 10000

__Example 18:__ Write a query that displays the total expenditure for each job title who has REP included in it and which exceeds 13000



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

```
SELECT  job_id, SUM(salary) PAYROLL
FROM    employees
WHERE   job_id LIKE '%REP%'
GROUP BY    job_id
HAVING  SUM(salary) > 13000
ORDER BY    SUM(salary);
```

133 %

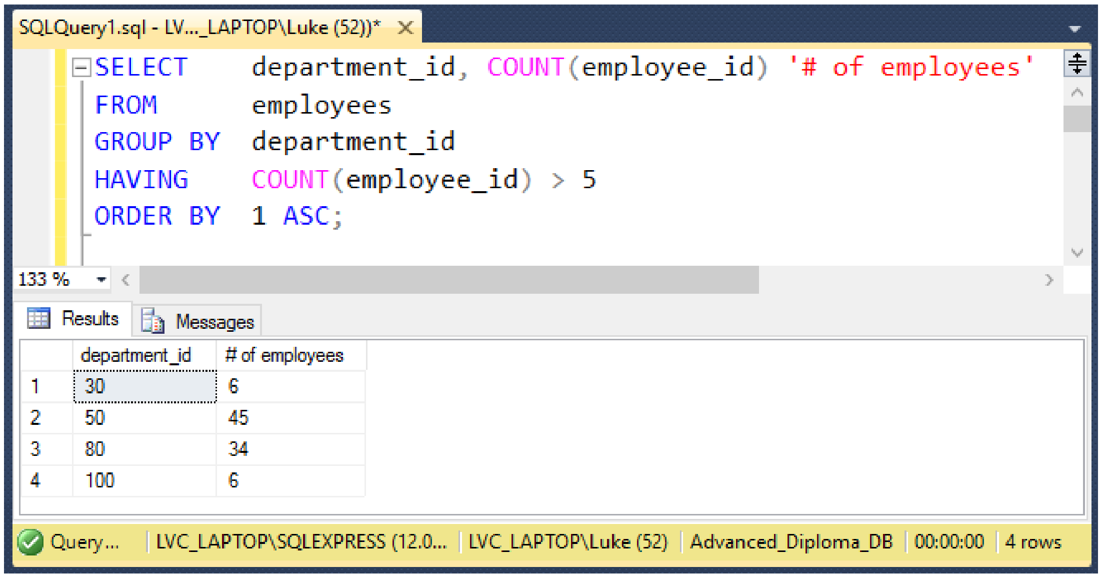
Results | Messages

	job_id	PAYROLL
1	SA_REP	250500.00

LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 1 rows

Figure 20 - Result of example 18: Total salary of job title with REP in it which exceeds 13000

__Example 19:__ Write a query that will list all the departments who have more than 5 employees in them. You are to list the department with the smallest value first



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

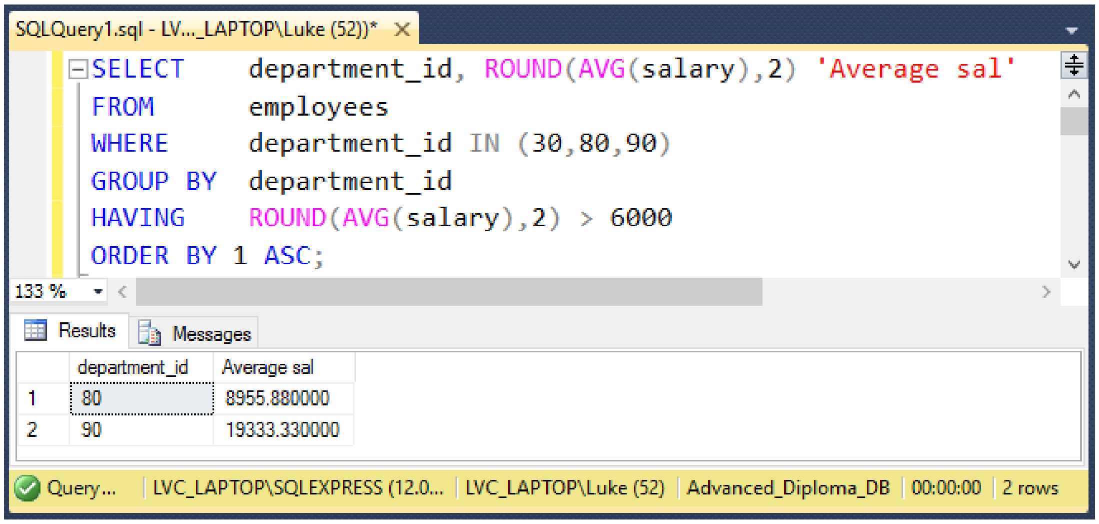
```
SELECT    department_id, COUNT(employee_id) '# of employees'
FROM      employees
GROUP BY  department_id
HAVING    COUNT(employee_id) > 5
ORDER BY  1 ASC;
```

	department_id	# of employees
1	30	6
2	50	45
3	80	34
4	100	6

Query... | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 4 rows

Figure 21 - result of example 19: Display all the departments and number of employees

__Example 20:__ Write a query that will display the average salary (rounded to 2 decimal places) in departments 30, 80 and 90 and whose average salary is larger than 6000



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*

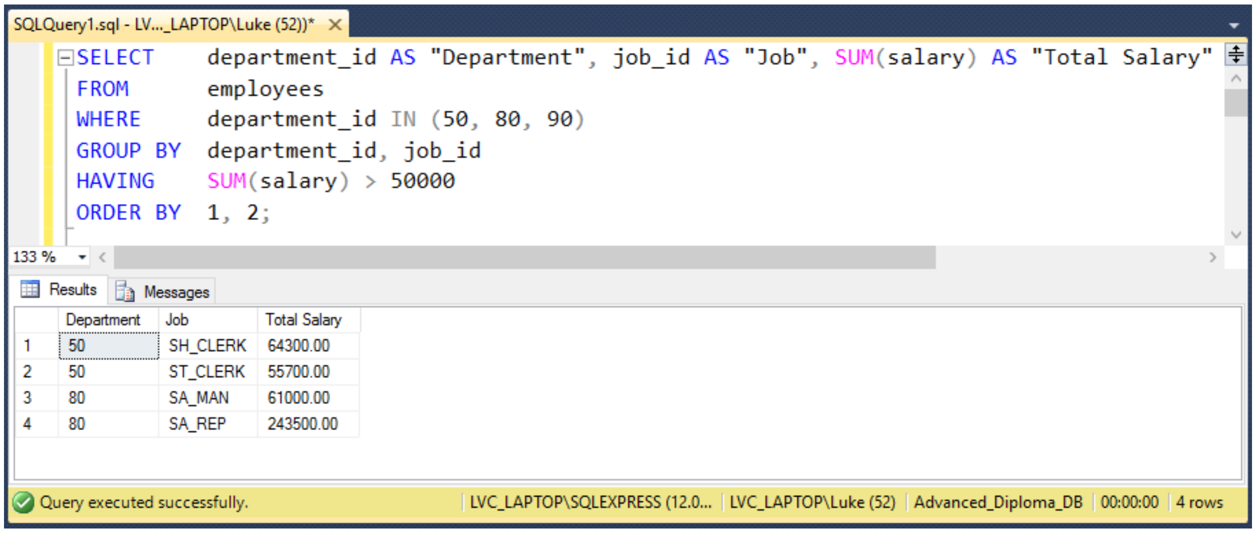
```
SELECT    department_id, ROUND(AVG(salary),2) 'Average sal'
FROM      employees
WHERE     department_id IN (30,80,90)
GROUP BY  department_id
HAVING    ROUND(AVG(salary),2) > 6000
ORDER BY 1 ASC;
```

	department_id	Average sal
1	80	8955.880000
2	90	19333.330000

Query... | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 2 rows

Figure 22 - Result of example 20: Average salary for departments 30, 80 and 90 exceeding 6000

Example 21: Write a query that will display the total salary for each department and each job_id. The departments that should be considered are 50, 80 and 90 and only sub queries with a total sum exceeding 50000 should be displayed



```sql
SELECT     department_id AS "Department", job_id AS "Job", SUM(salary) AS "Total Salary"
FROM       employees
WHERE      department_id IN (50, 80, 90)
GROUP BY   department_id, job_id
HAVING     SUM(salary) > 50000
ORDER BY   1, 2;
```

	Department	Job	Total Salary
1	50	SH_CLERK	64300.00
2	50	ST_CLERK	55700.00
3	80	SA_MAN	61000.00
4	80	SA_REP	243500.00

```
Query executed successfully. | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 4 rows
```