DML stands for Data Manipulation Language and up till now we have introduced only one statement which pertains to this subset of SQL – the SELECT statement. As you should have understood by now the SELECT statement focuses on data retrieval, but there are other statements within DML which perform other important tasks. Other statements that pertain to the DML subset are: INSERT, UPDATE, DELETE, TRUNCATE and MERGE.

Adding Rows of data

In Microsoft SQL Server there are several methods that can be adopted to be able to insert data in an entity. Amongst the available T-SQL statements that we can use to insert data in an entity, one can find; INSERT VALUES, INSERT SELECT, INSERT EXEC, SELECT INTO and BULK INSERT.

INSERT VALUES statement

This statement is used to insert data in an entity which is based on specified values. This statement can be used to insert either a single row or multiple rows of data in a particular entity. Consider the below entity; we can use INSERT VALUES such that a new product is included in the entity. The new product will be Peaches, it will cost 0.70 and it will be supplied by company 10.

Pid	Product Name	Product Price	SupplierId
1	Orange	0.50	10
2	Apple	0.40	10
3	Banana	0.60	20

Pid	Product Name	Product Price	SupplierId
1	Orange	0.50	10
2	Apple	0.40	10
3	Banana	0.60	20
4	Peaches	0.70	10

In order to add row/s to a particular entity using the INSERT INTO statement the below syntax needs to be used:

```
INSERT INTO table [(column [, column . . . ])]
VALUES (value [, value . . .])
            [,(value [, value . . ]) . . . ];
```

What do the keywords in this INSERT VALUES syntax statement mean?

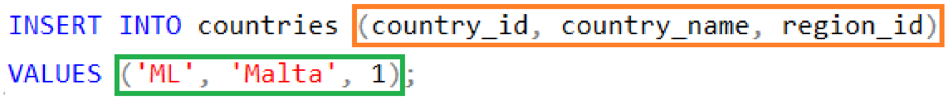
- **INSERT INTO table**– chooses the entity in which the data is to be inserted
- **[(column [, column . . .])]** – the names of the columns where the values need to be inserted. This is optional and can be left out (also known as target column names).
- **VALUES** – keyword which indicates that from this point onwards, the actual data to be inserted is to be listed
- **(value [, value . . .])** – the value keyword needs to be replaced by an actual value

NOTE:

- when inserting values which have a date or character/string data type these need to be enclosed in single quotes
- also remember that when inserting dates the format that is to be followed should be **yyyymmdd.**
- when inserting data for columns which have a numeric data type, you just need to include the actual numerical value – do not include any single quotes
- every new row to be inserted must include a value for ***each*** mandatory (NOT NULL) column in the entity

INSERT VALUES with target column names specified

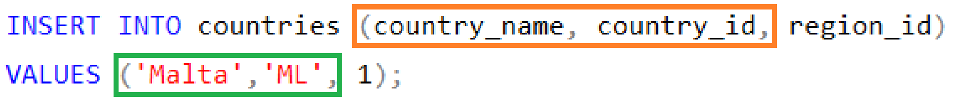
Although this option is not required, it is highly recommended that you use it as you will have control on the value-column relationships within the table. The order in which the target column names are written is not important as long as the equivalent values follow the order specified after the INSERT INTO keyword.



```
INSERT INTO countries (country_id, country_name, region_id)
VALUES ('ML', 'Malta', 1);
```

Figure 1 - example of a typical INSERT INTO statement

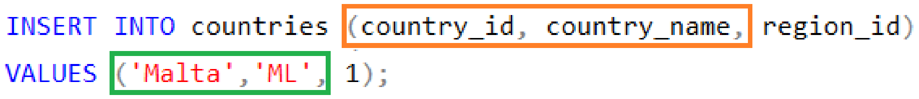
Figure 1 includes an example an INSERT INTO statement. The orange box include the columns that be inserted with data (OPTIONAL). The green box includes the actual data that will be inserted in the countries table.



```
INSERT INTO countries (country_name, country_id, region_id)
VALUES ('Malta','ML', 1);
```

Figure 2 - example of another INSERT INTO statement

Figure 2 will add the same data in the countries table, but it is written differently. Notice that the country_name and country_id have changed position. For this reason it was also important to change the position for the values 'Malta' and 'ML'.

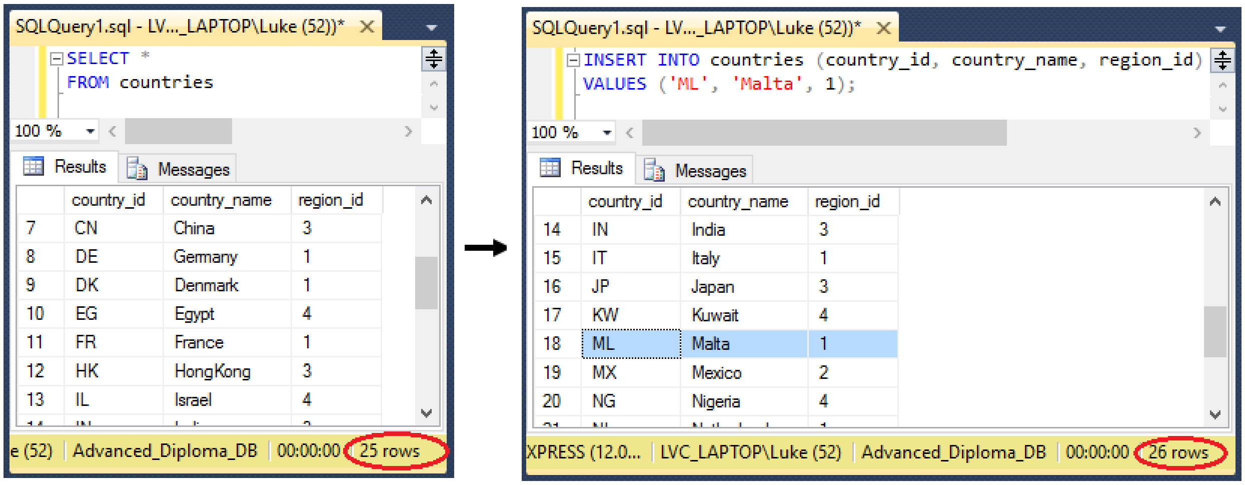


```
INSERT INTO countries (country_id, country_name, region_id)
VALUES ('Malta','ML', 1);
```

Figure 3 - erroneous INSERT INTO statement

NOTE: The statement in Figure 3, will execute normally without errors but the data inserted in the entity will be incorrect. In this case the value 'Malta' would be incorrectly placed in the country_id column and 'ML' in country_name. **Make sure that the value-column relationship is correct**

Example 1: Write a query that will insert a new row of data in the countries table. The new row should have the following data: country_id: ML, country_name: Malta, region_id: 1. The figure below shows that the row was inserted successfully as the number of rows increased by 2.


SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
100 %
Results
Messages

```
SELECT *
FROM countries
```

	country_id	country_name	region_id
7	CN	China	3
8	DE	Germany	1
9	DK	Denmark	1
10	EG	Egypt	4
11	FR	France	1
12	HK	HongKong	3
13	IL	Israel	4

e (52) | Advanced_Diploma_DB | 00:00:00 | 25 rows

SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
100 %
Results
Messages

```
INSERT INTO countries (country_id, country_name, region_id)
VALUES ('ML', 'Malta', 1);
```

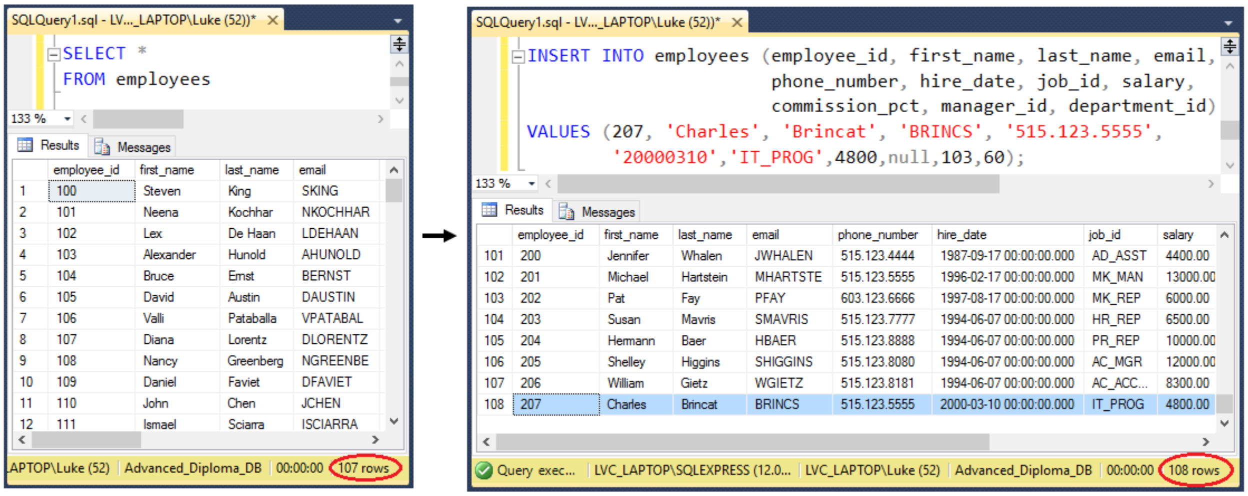
	country_id	country_name	region_id
14	IN	India	3
15	IT	Italy	1
16	JP	Japan	3
17	KW	Kuwait	4
18	ML	Malta	1
19	MX	Mexico	2
20	NG	Nigeria	4

XPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 26 rows

Figure 4 – Result of example 1: adding a new row using target column names

Example 2: Write a query that will input the following information in the employees table: employee_id: 207, first_name: Charles, last_name: Brincat, email: BRINCS, phone_number: 515.123.5555, hire_date: 10th March 2000, job_id: IT_PROG, salary: 4800, commission_pct: null, manager_id:103, department_id: 60

```sql
INSERT INTO employees (employee_id, first_name, last_name, email,
                       phone_number, hire_date, job_id, salary,
                       commission_pct, manager_id, department_id)
VALUES (207, 'Charles', 'Brincat', 'BRINCS', '515.123.5555',
        '20000310','IT_PROG',4800,null,103,60);
```



```sql
SELECT *
FROM employees
```

	employee_id	first_name	last_name	email
1	100	Steven	King	SKING
2	101	Neena	Kochhar	NKOCHHAR
3	102	Lex	De Haan	LDEHAAN
4	103	Alexander	Hunold	AHUNOLD
5	104	Bruce	Ernst	BERNST
6	105	David	Austin	DAUSTIN
7	106	Valli	Pataballa	VPATABAL
8	107	Diana	Lorentz	DLORENTZ
9	108	Nancy	Greenberg	NGREENBE
10	109	Daniel	Faviet	DFAVIET
11	110	John	Chen	JCHEN
12	111	Ismael	Sciarra	ISCIARRA

SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
133 %
Results
Messages
APTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 107 rows

```sql
INSERT INTO employees (employee_id, first_name, last_name, email,
                       phone_number, hire_date, job_id, salary,
                       commission_pct, manager_id, department_id)
VALUES (207, 'Charles', 'Brincat', 'BRINCS', '515.123.5555',
        '20000310','IT_PROG',4800,null,103,60);
```

	employee_id	first_name	last_name	email	phone_number	hire_date	job_id	salary
101	200	Jennifer	Whalen	JWHALEN	515.123.4444	1987-09-17 00:00:00.000	AD_ASST	4400.00
102	201	Michael	Hartstein	MHARTSTE	515.123.5555	1996-02-17 00:00:00.000	MK_MAN	13000.00
103	202	Pat	Fay	PFAY	603.123.6666	1997-08-17 00:00:00.000	MK_REP	6000.00
104	203	Susan	Mavris	SMAVRIS	515.123.7777	1994-06-07 00:00:00.000	HR_REP	6500.00
105	204	Hermann	Baer	HBAER	515.123.8888	1994-06-07 00:00:00.000	PR_REP	10000.00
106	205	Shelley	Higgins	SHIGGINS	515.123.8080	1994-06-07 00:00:00.000	AC_MGR	12000.00
107	206	William	Gietz	WGIETZ	515.123.8181	1994-06-07 00:00:00.000	AC_ACC...	8300.00
108	207	Charles	Brincat	BRINCS	515.123.5555	2000-03-10 00:00:00.000	IT_PROG	4800.00

SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
133 %
Results
Messages
Query exec... | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 108 rows

Figure 5 -Result of example2: adding a new employee

INSERT VALUES <u>without</u> target column names specified

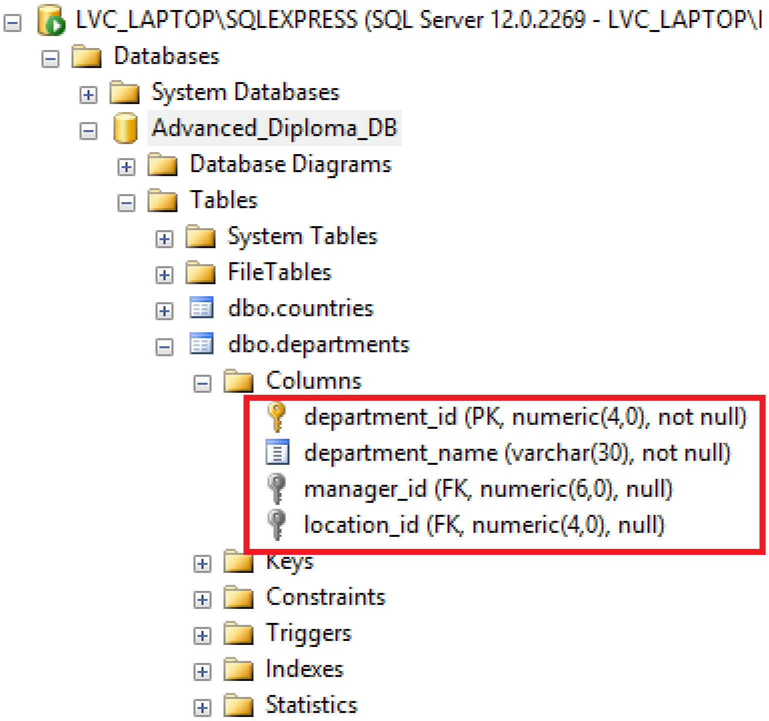
While inserting data in the database, it is possible to do without target names soon after the INSERT INTO table. As mentioned in the previous section, target columns are optional and for this reason they can be left out.

If this option is to be used when inserting data it is of *<u>utmost importance that the order of the columns is equivalent to that used while creating the entity</u>*. To determine the order of the columns used on creation there are two methods:

1. Open the Columns folder of the entity in Object Explorer
2. Use the *sp_columns* stored procedure

From example 3 and 4 in the next page, you can easily determine the order in which the columns have been created (department_id, department_name, manager_id, location_id). Also note that the sp_columns stored procedure includes a number of columns which give us a lot of information about each column in the table.

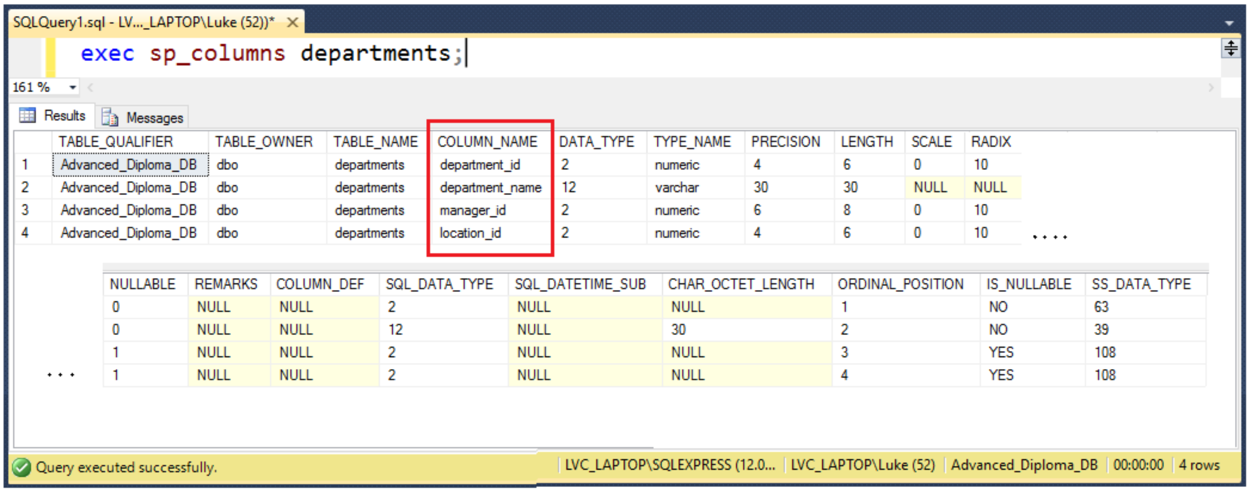
__Example 3__: Determine the order of the columns in the departments table using Object Explorer



```
⊟ LVC_LAPTOP\SQLEXPRESS (SQL Server 12.0.2269 - LVC_LAPTOP\l
  ⊟ Databases
    ⊞ System Databases
    ⊟ Advanced_Diploma_DB
      ⊞ Database Diagrams
      ⊟ Tables
        ⊞ System Tables
        ⊞ FileTables
        ⊞ dbo.countries
        ⊟ dbo.departments
          ⊟ Columns
              department_id (PK, numeric(4,0), not null)
              department_name (varchar(30), not null)
              manager_id (FK, numeric(6,0), null)
              location_id (FK, numeric(4,0), null)
          ⊞ Keys
          ⊞ Constraints
          ⊞ Triggers
          ⊞ Indexes
          ⊞ Statistics
```

Figure 6 - Screenshot showing the rows in the departments table (order they were created)

__Example 4__: Determine the order of the columns in the departments table using the *sp_columns* stored procedure



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

```sql
exec sp_columns departments;
```

	TABLE_QUALIFIER	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	DATA_TYPE	TYPE_NAME	PRECISION	LENGTH	SCALE	RADIX
1	Advanced_Diploma_DB	dbo	departments	department_id	2	numeric	4	6	0	10
2	Advanced_Diploma_DB	dbo	departments	department_name	12	varchar	30	30	NULL	NULL
3	Advanced_Diploma_DB	dbo	departments	manager_id	2	numeric	6	8	0	10
4	Advanced_Diploma_DB	dbo	departments	location_id	2	numeric	4	6	0	10

. . . .

NULLABLE	REMARKS	COLUMN_DEF	SQL_DATA_TYPE	SQL_DATETIME_SUB	CHAR_OCTET_LENGTH	ORDINAL_POSITION	IS_NULLABLE	SS_DATA_TYPE
0	NULL	NULL	2	NULL	NULL	1	NO	63
0	NULL	NULL	12	NULL	30	2	NO	39
1	NULL	NULL	2	NULL	NULL	3	YES	108
1	NULL	NULL	2	NULL	NULL	4	YES	108

. . .

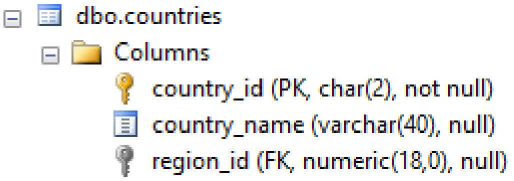
Query executed successfully. | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 4 rows

Figure 7 - Screenshot showing all the information related to the columns in the departments table.

Once that the order of the columns in a table is known, INSERT INTO statements can be written without the target columns. When using this method make sure that the values inserted correspond to the order of the columns.

Example 5: Write a query that will insert a new row of data in the countries table. The new row should have the following data: country_id: SP, country_name: Spain, region_id: 1. You are not to include the target columns in the INSERT INTO statement.

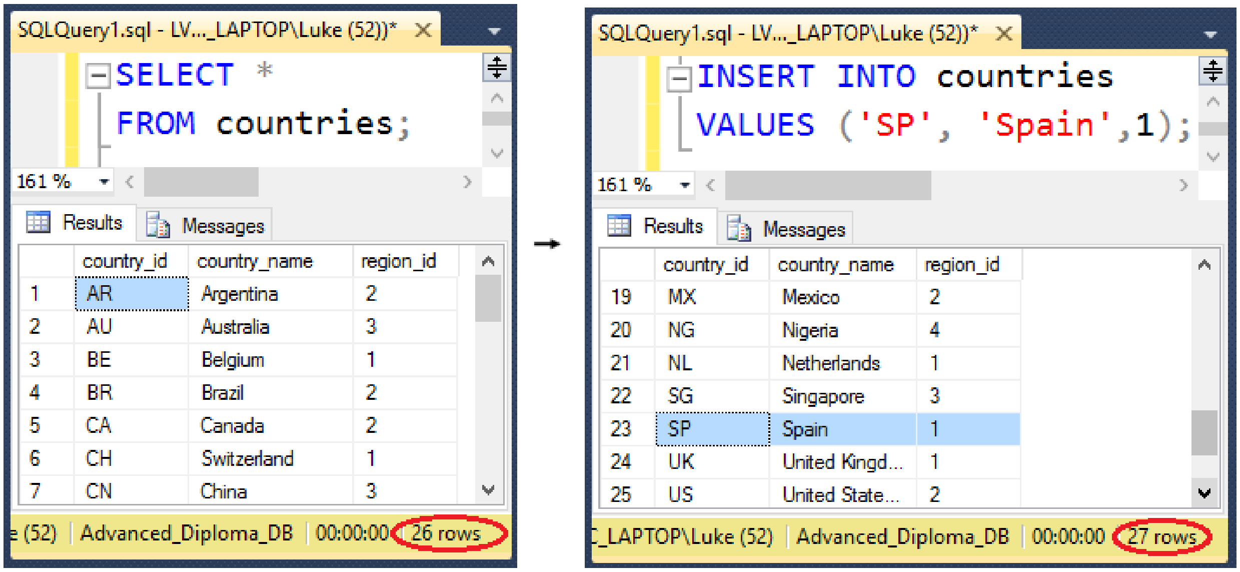
Given the fact that the target columns are not to be included the order of the columns needs to be determined. We used the Object explorer in this case. The order of the columns was country_id, country_name, region_id.



```
dbo.countries
  Columns
    country_id (PK, char(2), not null)
    country_name (varchar(40), null)
    region_id (FK, numeric(18,0), null)
```

Figure 8 - Order of the columns found in the countries table

Below is the answer after inserting the desired row in the countries table. You are to note that the values follow the order of the columns as shown in figure 8. Incorrectly order of values might result in data placed in incorrect columns or data type errors.


SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
161 %
Results
Messages
SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
161 %
Results
Messages

```sql
SELECT *
FROM countries;
```

	country_id	country_name	region_id
1	AR	Argentina	2
2	AU	Australia	3
3	BE	Belgium	1
4	BR	Brazil	2
5	CA	Canada	2
6	CH	Switzerland	1
7	CN	China	3

```
e (52) | Advanced_Diploma_DB | 00:00:00 | 26 rows
```

```sql
INSERT INTO countries
VALUES ('SP', 'Spain',1);
```

	country_id	country_name	region_id
19	MX	Mexico	2
20	NG	Nigeria	4
21	NL	Netherlands	1
22	SG	Singapore	3
23	SP	Spain	1
24	UK	United Kingd...	1
25	US	United State...	2

```
_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 27 rows
```

Figure 9 - Result of example 5: adding a new row without target columns being specified

INSERT VALUES for optional columns (which allow NULL values)

As you should have learnt during the first semester, a column/attribute in an entity can be either optional or mandatory. From figure 10, you can easily determine that in the Department entity, the department_id and department_name columns are mandatory while the location_id and manager_id columns are optional.

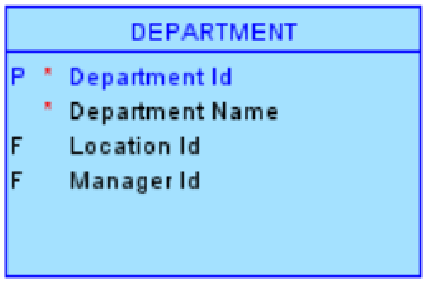
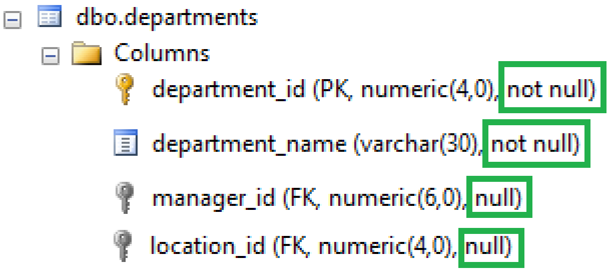


Figure 10 – representation of the Department entity using Information Engineering Notation

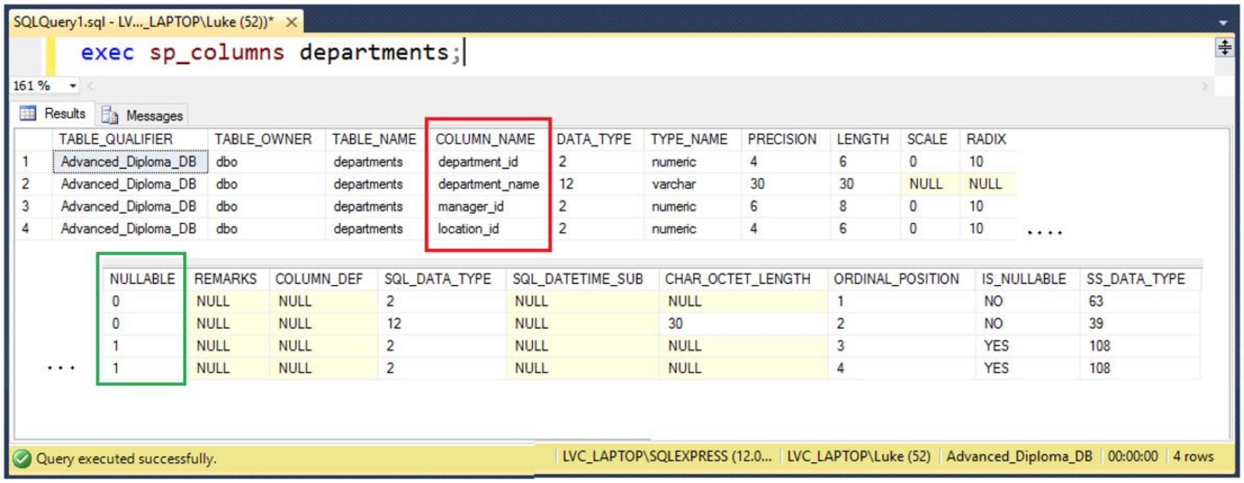
While creating the entity in the database, all mandatory attributes should be assigned a NOT NULL constraint. In simple words, a mandatory column should have a value for each row that exists in the entity and an optional column can be left empty for any row in the entity.

Given the fact that you will not always have access to the ERD or the CREATE TABLE code to determine if a column is mandatory or optional, you can use the two methods which were mentioned on page 4. As can be seen in figures 11 and 12, the mandatory fields are (department_id and department_name) and the optional fields are (manager_id and location_id). Note that when the *sp_columns* stored procedure is used to determine of a column is mandatory or not the NULLABLE column needs to be considered. If the value in this column is 0 then the column in mandatory and if the value is 1 then the column is optional (can be left empty)



```
⊟ dbo.departments
  ⊟ Columns
      department_id (PK, numeric(4,0), not null)
      department_name (varchar(30), not null)
      manager_id (FK, numeric(6,0), null)
      location_id (FK, numeric(4,0), null)
```

Figure 11 - mandatory or optional columns in the department table (using Object Explorer)



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

```sql
exec sp_columns departments;
```

161 %

Results | Messages

	TABLE_QUALIFIER	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	DATA_TYPE	TYPE_NAME	PRECISION	LENGTH	SCALE	RADIX
1	Advanced_Diploma_DB	dbo	departments	department_id	2	numeric	4	6	0	10
2	Advanced_Diploma_DB	dbo	departments	department_name	12	varchar	30	30	NULL	NULL
3	Advanced_Diploma_DB	dbo	departments	manager_id	2	numeric	6	8	0	10
4	Advanced_Diploma_DB	dbo	departments	location_id	2	numeric	4	6	0	10

NULLABLE	REMARKS	COLUMN_DEF	SQL_DATA_TYPE	SQL_DATETIME_SUB	CHAR_OCTET_LENGTH	ORDINAL_POSITION	IS_NULLABLE	SS_DATA_TYPE
0	NULL	NULL	2	NULL	NULL	1	NO	63
0	NULL	NULL	12	NULL	30	2	NO	39
1	NULL	NULL	2	NULL	NULL	3	YES	108
1	NULL	NULL	2	NULL	NULL	4	YES	108

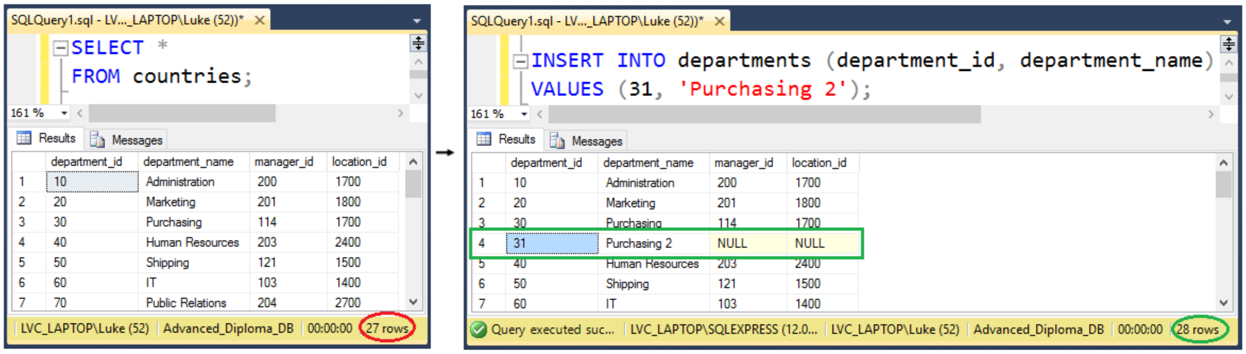
Query executed successfully. | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 4 rows

Figure 12 - mandatory or optional columns in department table (using sp_columns)

Once that you manage to determine which columns are optional, you can use either of these methods to add rows of data:

1. *Implicit Method*: remove the optional columns from the target columns and do not include a value for these columns
2. *Explicit Method*: the null keyword is to be specifically included in the values list

<u>*Example 6*</u>: Write a query that adds a new department (department_id = 31, department_name = Purchasing 2) to the departments table. Note that the manager_id and location_id should be left empty and the implicit method should be used.



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

```sql
SELECT *
FROM countries;
```

161 %

Results | Messages

	department_id	department_name	manager_id	location_id
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500
6	60	IT	103	1400
7	70	Public Relations	204	2700

LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 27 rows

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

```sql
INSERT INTO departments (department_id, department_name)
VALUES (31, 'Purchasing 2');
```

161 %

Results | Messages

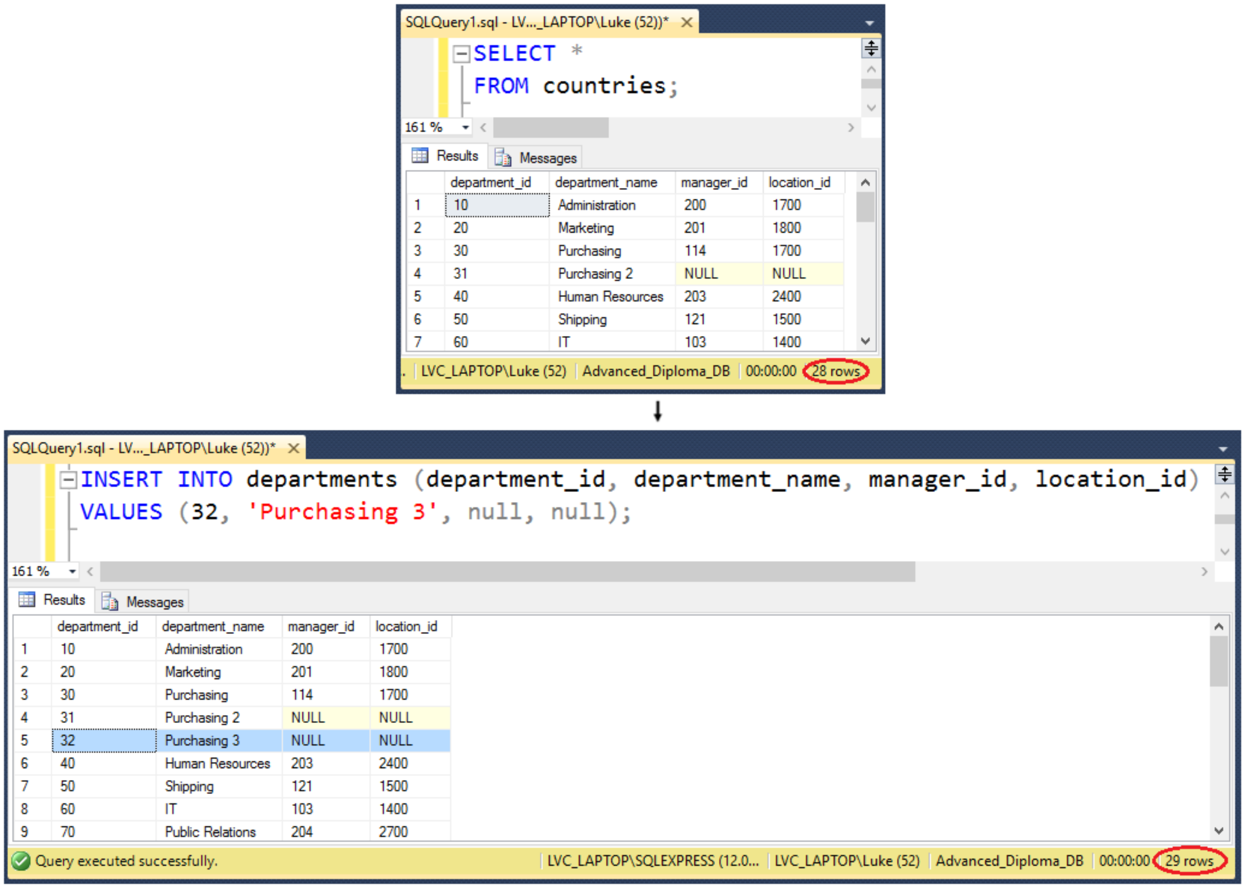
	department_id	department_name	manager_id	location_id
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	31	Purchasing 2	NULL	NULL
5	40	Human Resources	203	2400
6	50	Shipping	121	1500
7	60	IT	103	1400

Query executed suc... | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 28 rows

Figure 13 - Result of example 6: Implicit method to add new row

NOTE: since we have used the *Implicit method*, the manager_id and location_id are not included in the target columns as they are optional columns

Example 7: Write a query that adds a new department (department_id = 32, department_name = Purchasing 3) to the departments table. Note that the manager_id and location_id should be left empty and the explicit method should be used.



SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
161 %
Results
Messages

```sql
SELECT *
FROM countries;
```

	department_id	department_name	manager_id	location_id
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	31	Purchasing 2	NULL	NULL
5	40	Human Resources	203	2400
6	50	Shipping	121	1500
7	60	IT	103	1400

LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 28 rows

SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
161 %
Results
Messages

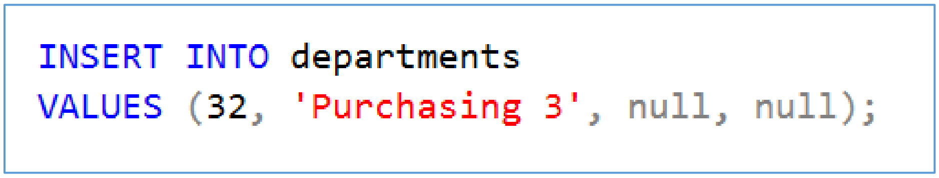
```sql
INSERT INTO departments (department_id, department_name, manager_id, location_id)
VALUES (32, 'Purchasing 3', null, null);
```

	department_id	department_name	manager_id	location_id
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	31	Purchasing 2	NULL	NULL
5	32	Purchasing 3	NULL	NULL
6	40	Human Resources	203	2400
7	50	Shipping	121	1500
8	60	IT	103	1400
9	70	Public Relations	204	2700

Query executed successfully. | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 29 rows

Figure 14 - Result of example 7: Explicit method to add a new row

NOTE: since we have used the *Explicit method*, in the VALUES section we had to include two consecutive 'null' one for manager_id and the other for location_id. Also note that the INSERT statement could be changed to the one in figure 15 (remember that the target columns are optional and can be left out, provided that the actual values are in the column order which was used during table creation).



```sql
INSERT INTO departments
VALUES (32, 'Purchasing 3', null, null);
```

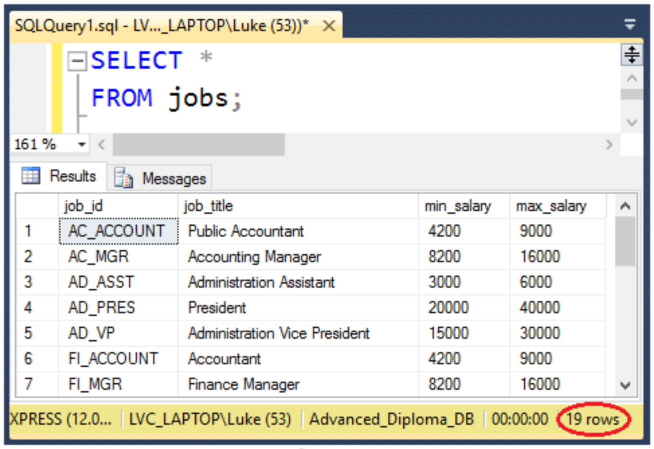
Figure 15 - The equivalent of the code in the INSERT VALUES used in figure 14

INSERT VALUES and multiple rows

The INSERT VALUES statement can be used to add more than one row at a time using just a single statement. In simple words this means that with one statement you can add multiple rows in the table. Each row of data in the statement is to be separated using a comma.

A very important point to know is that even though multiple rows can be added the statement is still considered as **one** transaction. This means that if a row in the statement fails to be added to the table, the entire statement will fail.

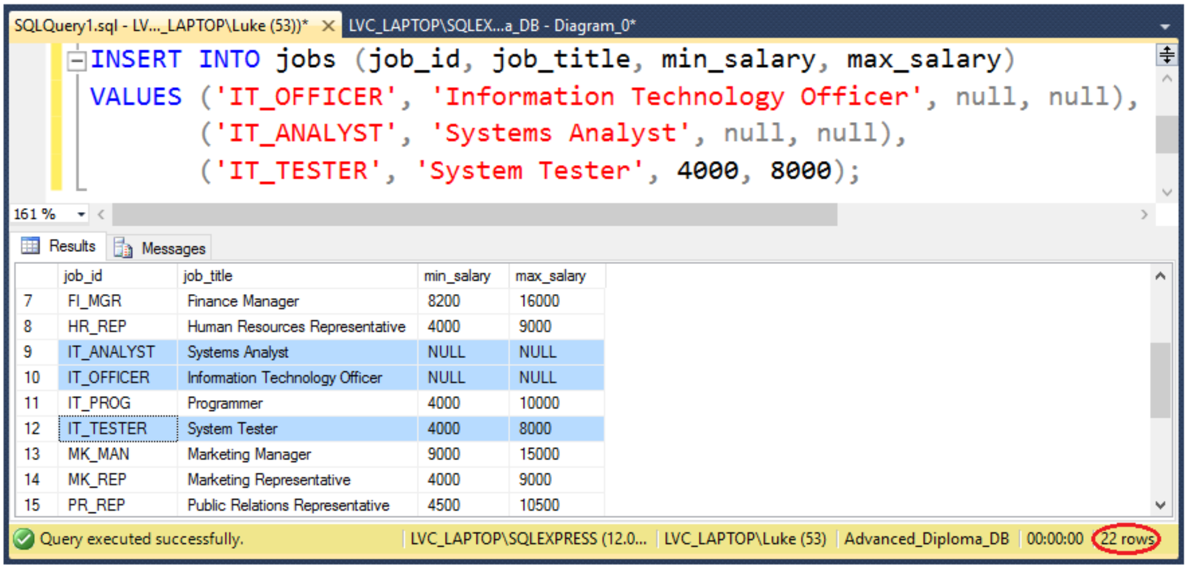
<u>Example 8</u>: Write a query that will add three job titles (Information Technology Officer, Systems Analyst and System Tester) to the jobs table.



```sql
SELECT *
FROM jobs;
```

	job_id	job_title	min_salary	max_salary
1	AC_ACCOUNT	Public Accountant	4200	9000
2	AC_MGR	Accounting Manager	8200	16000
3	AD_ASST	Administration Assistant	3000	6000
4	AD_PRES	President	20000	40000
5	AD_VP	Administration Vice President	15000	30000
6	FI_ACCOUNT	Accountant	4200	9000
7	FI_MGR	Finance Manager	8200	16000

XPRESS (12.0... | LVC_LAPTOP\Luke (53) | Advanced_Diploma_DB | 00:00:00 | 19 rows


SQLQuery1.sql - LV..._LAPTOP\Luke (53))*
LVC_LAPTOP\SQLEX...a_DB - Diagram_0*
161 %
Results
Messages

```sql
INSERT INTO jobs (job_id, job_title, min_salary, max_salary)
VALUES ('IT_OFFICER', 'Information Technology Officer', null, null),
       ('IT_ANALYST', 'Systems Analyst', null, null),
       ('IT_TESTER', 'System Tester', 4000, 8000);
```

	job_id	job_title	min_salary	max_salary
7	FI_MGR	Finance Manager	8200	16000
8	HR_REP	Human Resources Representative	4000	9000
9	IT_ANALYST	Systems Analyst	NULL	NULL
10	IT_OFFICER	Information Technology Officer	NULL	NULL
11	IT_PROG	Programmer	4000	10000
12	IT_TESTER	System Tester	4000	8000
13	MK_MAN	Marketing Manager	9000	15000
14	MK_REP	Marketing Representative	4000	9000
15	PR_REP	Public Relations Representative	4500	10500

Query executed successfully. | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (53) | Advanced_Diploma_DB | 00:00:00 | 22 rows

Figure 16 - Result of example 8: three rows added using INSERT VALUES with multiple columns

NOTE:

- In the previous screenshot, explicit method is used in the case of null values (the case with the first two rows inserted).
- Given that all the rows have all the four values, the targeted columns could have been left out.

While using INSERT VALUES one must keep in mind the following common mistakes:

- Columns which are mandatory (NOT NULL) are left without a value
- Columns which have a unique constraint have attempts of duplicate data
- Values which attempt to violate a FOREIGN KEY constraint
- Values which attempt to violate a CHECK constraint
- Values which have an incorrect data type (Data type mismatch)
- Values which are too wide for the column they are intended for.

INSERT SELECT statement

The INSERT SELECT statement is commonly considered as the statement that can be used to copy data from one existing table to another existing table. If the target table does not exist this command will not function.

```
INSERT INTO table [(column [, column . . . ])]
        SELECT {column|expression [,column|expression . .]}
        FROM table
        [WHERE search_condition]
```

It is very important to keep in mind that the number of columns returned by the SELECT statement cannot exceed the number of columns in the target table. Also all the mandatory columns (columns who have NOT NULL and PRIMARY KEY constraints) in the target table should have an entry in the SELECT statement as these columns should always have a value.

Example 9: Write the required statements such that you create a new entity named *employees_names* with three columns – emp_id, name, surname. You are then to copy the name and surname of all the employees to the newly created table.

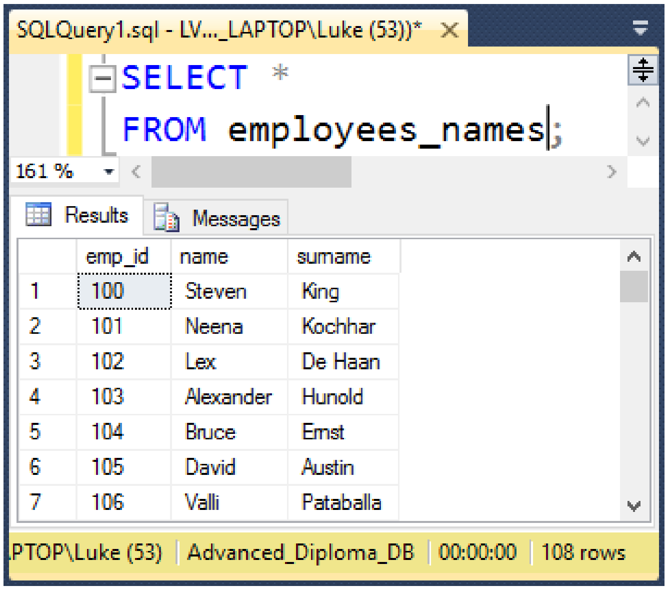
The below statement will create the employees_names table with the three columns stated above

```
CREATE TABLE employees_names
(
    emp_id INTEGER PRIMARY KEY ,
    name VARCHAR(50) CONSTRAINT en_name_nn NOT NULL,
    surname VARCHAR(50)  CONSTRAINT en_surname_nn NOT NULL
);
```

The below statement copies the name and surname from the employees table to the newly created table employees_names

```
INSERT INTO employees_names
    SELECT employee_id,first_name, last_name
    FROM employees;
```

The below result screenshot, displays all the columns and rows in the newly created table.



```
SELECT *
FROM employees_names;
```

	emp_id	name	surname
1	100	Steven	King
2	101	Neena	Kochhar
3	102	Lex	De Haan
4	103	Alexander	Hunold
5	104	Bruce	Ernst
6	105	David	Austin
7	106	Valli	Pataballa

PTOP\Luke (53) | Advanced_Diploma_DB | 00:00:00 | 108 rows

Figure 17 -Result of example 9: rows from employees table successfully inserted in employees_names

Example 10: Write the necessary statements such that a new table (employees_restricted) is created. This table should have 3 columns – *eid* (PRIMARY KEY), *name* (mandatory) and *salary*. You are to include a statement that copies the employee_id, full name and salary from the employees table to the employees_restricted table.

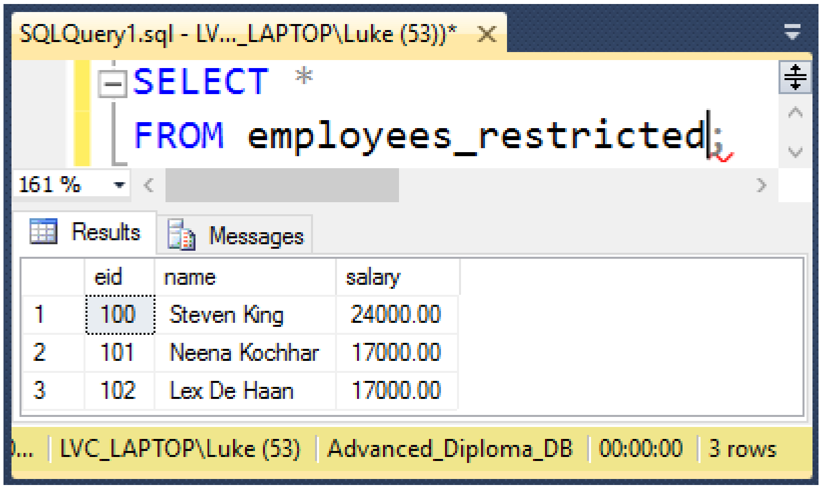
The below code is used to create the target table which will contain the data that is to be copied

```
CREATE TABLE employees_restricted
(
    eid INTEGER PRIMARY KEY ,
    name VARCHAR(50) CONSTRAINT en_name_nn NOT NULL,
    salary NUMERIC(8,2)
);
```

The below INSERT SELECT will take the employee_id, the concatenated name and surname, and the salary of all the employees who earn more than 16999 and place a copy in the employees_restricted entity.

```
INSERT INTO employees_restricted (eid, name, salary)
    SELECT employee_id,first_name +' '+ last_name, salary
    FROM employees
    WHERE salary >= 17000;
```

Figure 18 below shows the three employees which satisfy the condition and which have been copied to the employees_restricted entity



```
SELECT *
FROM employees_restricted;
```

	eid	name	salary
1	100	Steven King	24000.00
2	101	Neena Kochhar	17000.00
3	102	Lex De Haan	17000.00

... LVC_LAPTOP\Luke (53) Advanced_Diploma_DB 00:00:00 3 rows

Figure 18 - result of example 10 - employees which were copied from the original table to employees_restricted

SELECT INTO statement

This statement is a bit different from the previous one as it does not require the target table to be created before the actual copying of data takes place. This statement involves a query (found in the SELECT part) and a target entity (created by the statement itself). The syntax for the SELECT INTO statement is the below:

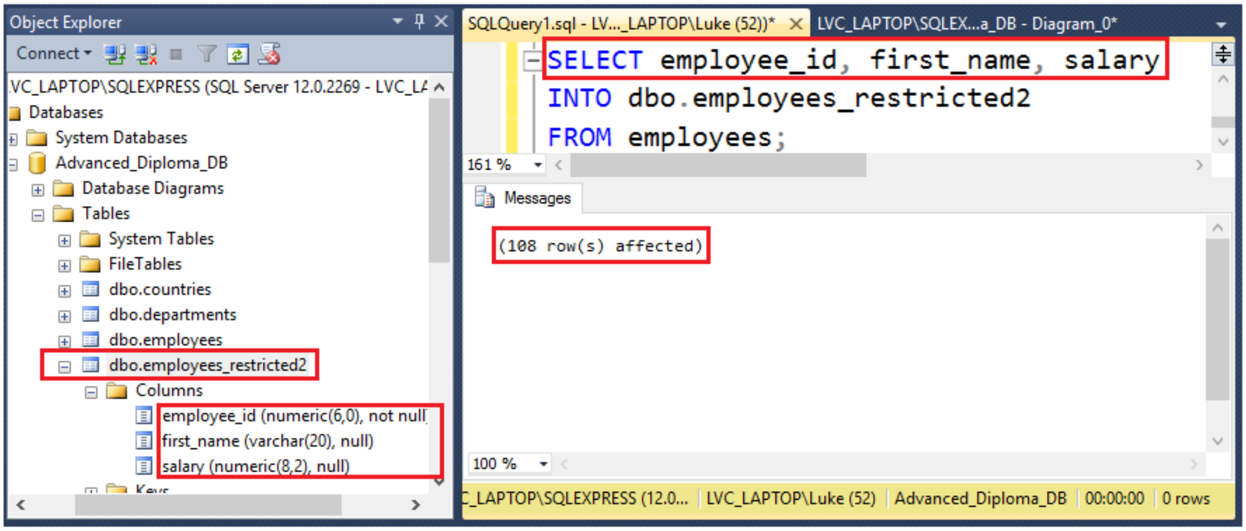
```
SELECT {column [,column . .]}
INTO table
FROM table
[WHERE search_condition]
```

The SELECT INTO statement automatically creates the target entity. The generated target entity will have the same definition of the source database and aspects such as column names, data types, nullability, and IDENTITY property are copied. Although the mentioned aspects are copied, other aspects such as indexes, triggers, permissions and others are not (these need to be specifically applied through scripting from the source entity). Apart from the structure of the entity itself, this statement will also copy the data which satisfies the conditions placed in the SELECT part.

One of the problems of this statement is that you will have no control on the definition of the target entity. Although some things such as IDENTITY and NOT NULL can be modified from within the statement itself, not all the copied aspects can be modified from within the statement. For the scope of this subject we will not attempt to modify the target table from within the SELECT INTO statement itself.

Example 11: Write a query that will copy all the data found in the employee number, name and salary columns within the employees table, into a new table which does not even exist in the database. The new table in which the extracted data is to be placed should be named employees_restricted2.

Figure 19 shows the code that is used to copy all the 108 rows extracted from the employees table to the newly generated table employees_restricted2 table. Notice that after executing the statement and refreshing the tables folder in the Object Explorer, the new table (employees_restricted2) is available and the 3 columns selected in the SELECT part are included.

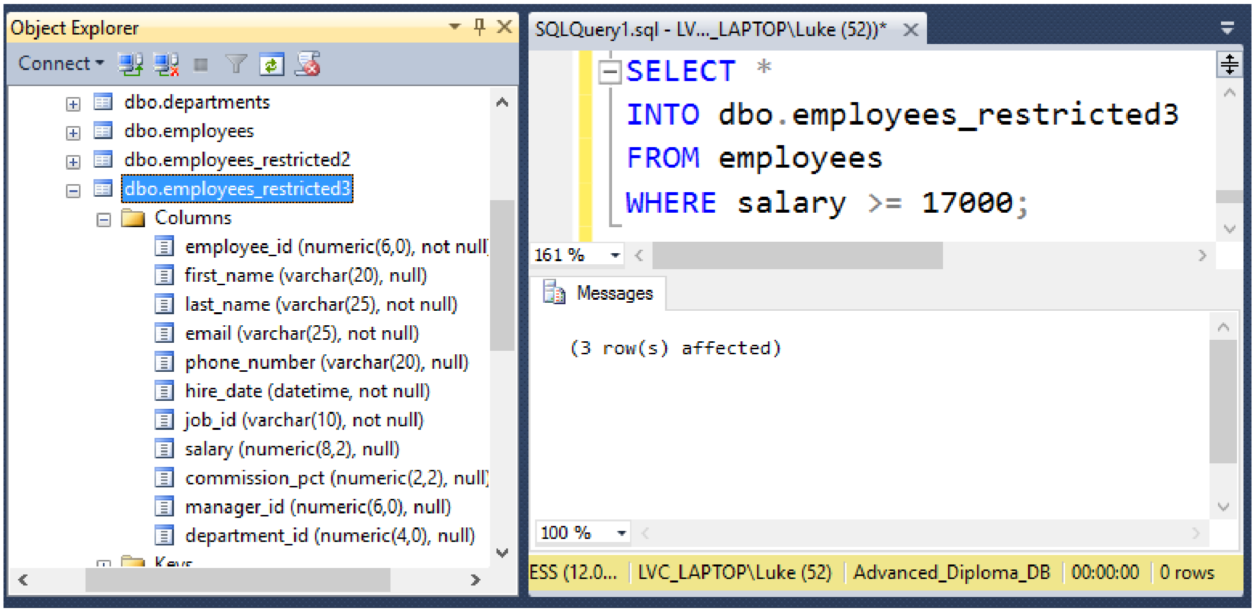

Object Explorer
Connect
VC_LAPTOP\SQLEXPRESS (SQL Server 12.0.2269 - LVC_LA
Databases
System Databases
Advanced_Diploma_DB
Database Diagrams
Tables
System Tables
FileTables
dbo.countries
dbo.departments
dbo.employees
dbo.employees_restricted2
Columns
employee_id (numeric(6,0), not null
first_name (varchar(20), null)
salary (numeric(8,2), null)
Keys
SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
LVC_LAPTOP\SQLEX...a_DB - Diagram_0*
161 %
Messages
100 %
_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 0 rows

```
SELECT employee_id, first_name, salary
INTO dbo.employees_restricted2
FROM employees;
```

```
(108 row(s) affected)
```

Figure 19 - Result of example 11: extracting 3 rows with the data and placing them in a new table

Example 12: Write a query that will copy all columns and rows of data for those employees who earn 17000 or more. One single statement should be used to extract the desired information and place it in a new table named employees_restricted3. Note that this time round the new table will only have 3 rows of data inserted in it as we included a WHERE condition.


Object Explorer
Connect
dbo.departments
dbo.employees
dbo.employees_restricted2
dbo.employees_restricted3
Columns
employee_id (numeric(6,0), not null
first_name (varchar(20), null)
last_name (varchar(25), not null)
email (varchar(25), not null)
phone_number (varchar(20), null)
hire_date (datetime, not null)
job_id (varchar(10), not null)
salary (numeric(8,2), null)
commission_pct (numeric(2,2), null)
manager_id (numeric(6,0), null)
department_id (numeric(4,0), null)
Keys
SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
161 %
Messages
100 %
ESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 0 rows

```
SELECT *
INTO dbo.employees_restricted3
FROM employees
WHERE salary >= 17000;
```

```
(3 row(s) affected)
```

Modifying Rows of data

Whenever data in a table is to be modified the UPDATE statement is to be used. The standard UPDATE statement will be considered in this subject but T-SQL offers a number of extensions which can be used. These extensions require the use of more complex material (some of which will be covered in the coming weeks).

Considering the table below we can be in a situation where we will need to change any of the values in the table below. A common change in such a table is that of the price or the supplier. If we consider a shop which sells the below products, there could be a situation where the price for a particular product changes due to certain circumstances.

Pid	Product Name	Product Price	SupplierId
1	Orange	0.50	10
2	Apple	0.40	10
3	Banana	0.60	20

↓

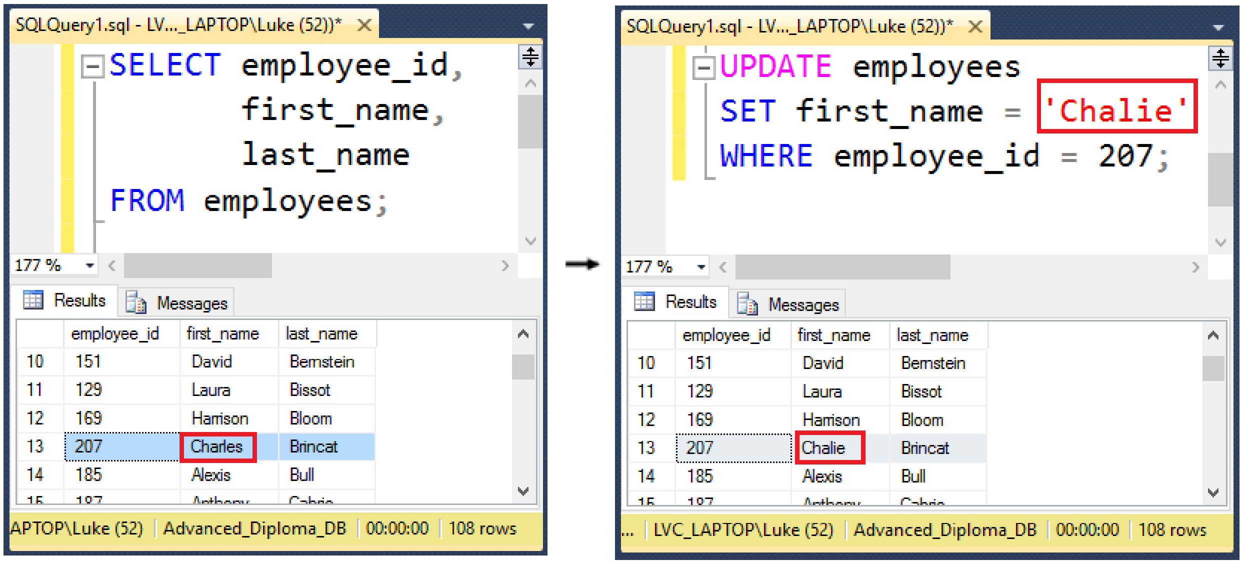
Pid	Product Name	Product Price	SupplierId
1	Orange	0.50	10
2	Apple	0.40	10
3	Banana	1.10	20

In order to be able to perform such changes, you are to make use of the following UPDATE syntax:

```
UPDATE table

SET column = value [, column =value, . . .]

[WHERE condition];
```

The above syntax is made up of the UPDATE keyword which should be followed by the name of the table which is to have the modifications. Soon after the table is declared, the SET keyword is to be used, together with all the columns which are to be effected and their new values. Multiple columns can have their values updated with a single UPDATE statement but it is important that between each column-value combination a comma is included. The WHERE clause is an optional part which needs to be handled very carefully. If the WHERE clause is not included then all the rows will be updated. If you want to restrict the changes to particular row/s, then it is very important to specify the correct WHERE condition.

Example 13: Write a query that will change the name of employee 207 from Charles to Chalie.



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

```sql
SELECT employee_id,
       first_name,
       last_name
FROM employees;
```

	employee_id	first_name	last_name
10	151	David	Bernstein
11	129	Laura	Bissot
12	169	Harrison	Bloom
13	207	Charles	Brincat
14	185	Alexis	Bull
15	187	Anthony	Cabrio

APTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 108 rows

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

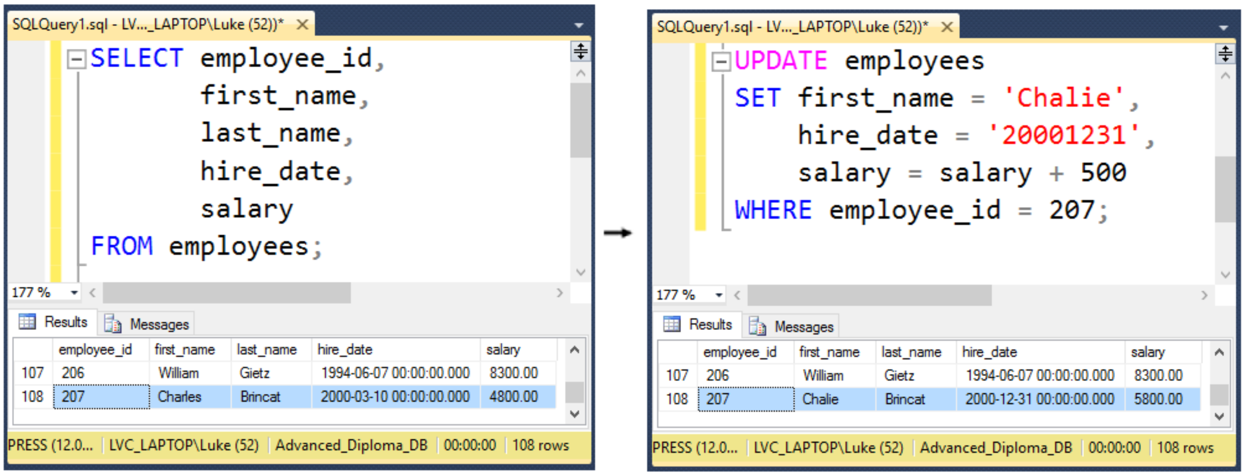
```sql
UPDATE employees
SET first_name = 'Chalie'
WHERE employee_id = 207;
```

	employee_id	first_name	last_name
10	151	David	Bernstein
11	129	Laura	Bissot
12	169	Harrison	Bloom
13	207	Chalie	Brincat
14	185	Alexis	Bull
15	187	Anthony	Cabrio

... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 108 rows

Figure 20 - Result of example 13: Update statement with a single column update

Example 14: Write a query that will change three columns (first_name, hire_date, salary) for the same employee. Employees 207 should be updated such that his first_name is changed to Chalie, hire_date to the last day of the year 2000 and the salary should be an increase of 500 to the current salary.



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

```sql
SELECT employee_id,
       first_name,
       last_name,
       hire_date,
       salary
FROM employees;
```

	employee_id	first_name	last_name	hire_date	salary
107	206	William	Gietz	1994-06-07 00:00:00.000	8300.00
108	207	Charles	Brincat	2000-03-10 00:00:00.000	4800.00

PRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 108 rows

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

```sql
UPDATE employees
SET first_name = 'Chalie',
    hire_date = '20001231',
    salary = salary + 500
WHERE employee_id = 207;
```

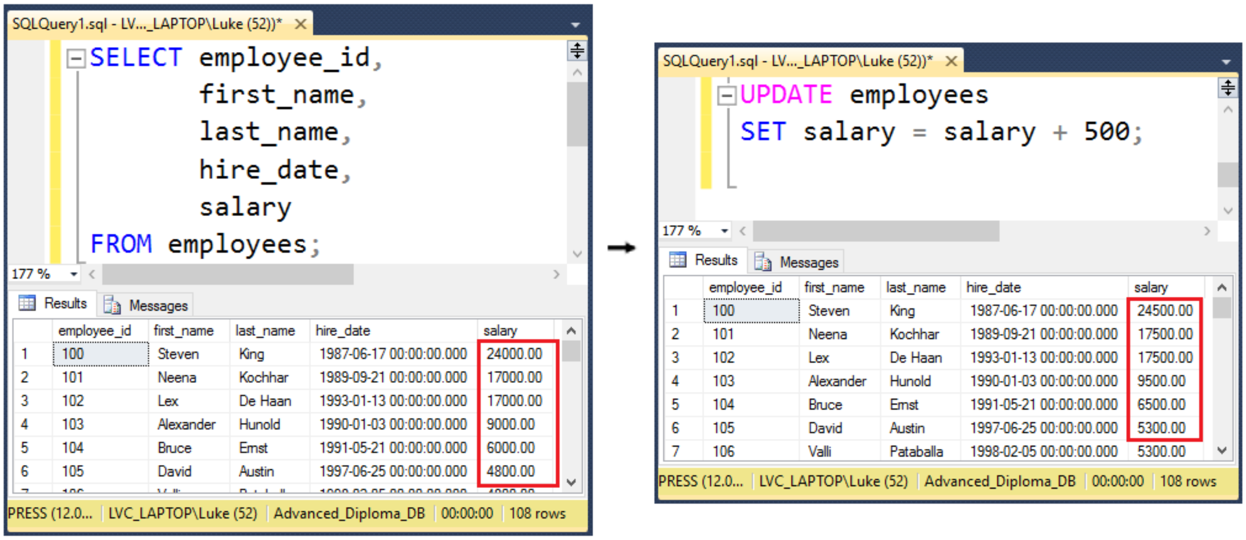
	employee_id	first_name	last_name	hire_date	salary
107	206	William	Gietz	1994-06-07 00:00:00.000	8300.00
108	207	Chalie	Brincat	2000-12-31 00:00:00.000	5800.00

PRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 108 rows

Figure 21 - Result of example 14: the update statement used to change multiple columns

Note: In the above screenshot, it evident that columns storing character and date should be enclosed in single quotes, but numerical columns should not.

__Example 15__: Write a query that will increase the employees' salary by 500.


SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
177 %
Results
Messages
PRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 108 rows

```sql
SELECT employee_id,
       first_name,
       last_name,
       hire_date,
       salary
FROM employees;
```

	employee_id	first_name	last_name	hire_date	salary
1	100	Steven	King	1987-06-17 00:00:00.000	24000.00
2	101	Neena	Kochhar	1989-09-21 00:00:00.000	17000.00
3	102	Lex	De Haan	1993-01-13 00:00:00.000	17000.00
4	103	Alexander	Hunold	1990-01-03 00:00:00.000	9000.00
5	104	Bruce	Ernst	1991-05-21 00:00:00.000	6000.00
6	105	David	Austin	1997-06-25 00:00:00.000	4800.00

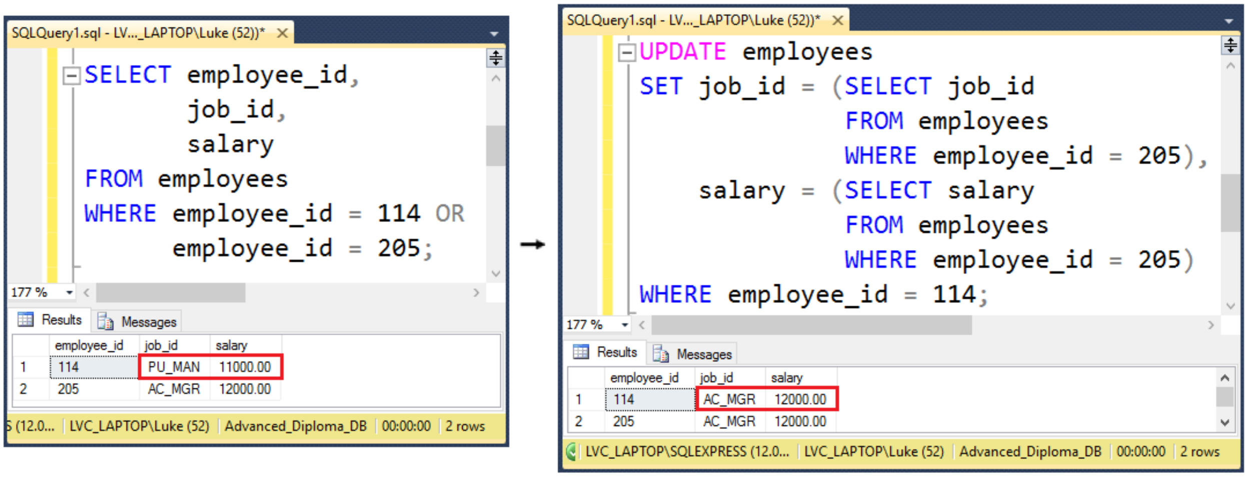
```sql
UPDATE employees
SET salary = salary + 500;
```

	employee_id	first_name	last_name	hire_date	salary
1	100	Steven	King	1987-06-17 00:00:00.000	24500.00
2	101	Neena	Kochhar	1989-09-21 00:00:00.000	17500.00
3	102	Lex	De Haan	1993-01-13 00:00:00.000	17500.00
4	103	Alexander	Hunold	1990-01-03 00:00:00.000	9500.00
5	104	Bruce	Ernst	1991-05-21 00:00:00.000	6500.00
6	105	David	Austin	1997-06-25 00:00:00.000	5300.00
7	106	Valli	Pataballa	1998-02-05 00:00:00.000	5300.00

Figure 22 - Results of example 15: all the salaries in the employees table added by 500

Note: given that all the rows have been updated, the WHERE clause was not used.

__Example 16__: Write a query that will change the job_id and salary of employee 114, such that the values become equal to the one's of employee 205


SQLQuery1.sql - LV..._LAPTOP\Luke (52))*
177 %
Results
Messages
5 (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 2 rows
LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 2 rows

```sql
SELECT employee_id,
       job_id,
       salary
FROM employees
WHERE employee_id = 114 OR
      employee_id = 205;
```

	employee_id	job_id	salary
1	114	PU_MAN	11000.00
2	205	AC_MGR	12000.00

```sql
UPDATE employees
SET job_id = (SELECT job_id
              FROM employees
              WHERE employee_id = 205),
    salary = (SELECT salary
              FROM employees
              WHERE employee_id = 205)
WHERE employee_id = 114;
```

	employee_id	job_id	salary
1	114	AC_MGR	12000.00
2	205	AC_MGR	12000.00

Figure 23 - Result of example 16: change the values using sub queries

Removing Rows of data

Sometimes it is necessary to be able to remove rows of data from a particular table. The reasons behind the use of such an operation might be many but the idea is to use a condition which will select the row/s which are to be permanently removed from the database. In T-SQL there are two main statements that help the user to perform this operation - DELETE and TRUNCATE.

Pid	Product Name	Product Price	SupplierId
1	Orange	0.50	10
2	Apple	0.40	10
3	Banana	1.10	20

↓

Pid	Product Name	Product Price	SupplierId
1	Orange	0.50	10
2	Apple	0.40	10

The above is what happens when a row is deleted from a particular table. As shown in the above, the third product (Pid=3) has been removed and the current table is now left with only two rows of data.

DELETE statement

The delete statement is used to remove rows of data from the database. The good thing with regards to this statement is that you can select which rows you want to delete by specifying a condition in the WHERE clause

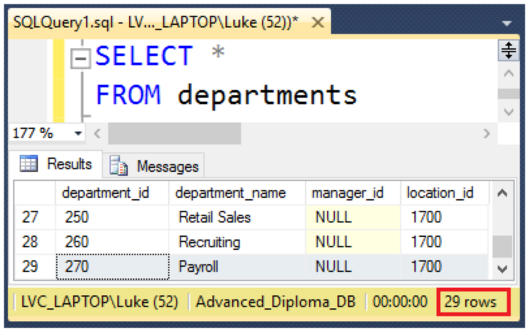
```
DELETE FROM table

[WHERE condition];
```

As with the case of the UPDATE statement it is very important to note that the use of the WHERE clause is paramount. Although this clause is optional, failing to include it will result in the loss of all the rows in the particular table.

Also note, that if a row is being used in a FK relationship you will not be allowed to remove it.

Another important thing to note, while using this statement is that whenever the database increases in size, delete operations might introduce time delays. Given the fact that delete operations will be logged in the transaction file, this may result in a huge number of entries in this file – hence a lot of storage will be required to store the file. Also locking mechanisms might be applied on the rows or even on the whole table, restricting access to the user.

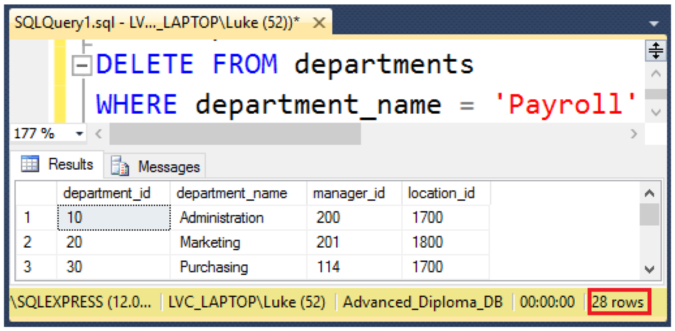
Example 17: Write a query that will remove the payroll department from the departments table



```sql
SELECT *
FROM departments
```

	department_id	department_name	manager_id	location_id
27	250	Retail Sales	NULL	1700
28	260	Recruiting	NULL	1700
29	270	Payroll	NULL	1700

LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 29 rows

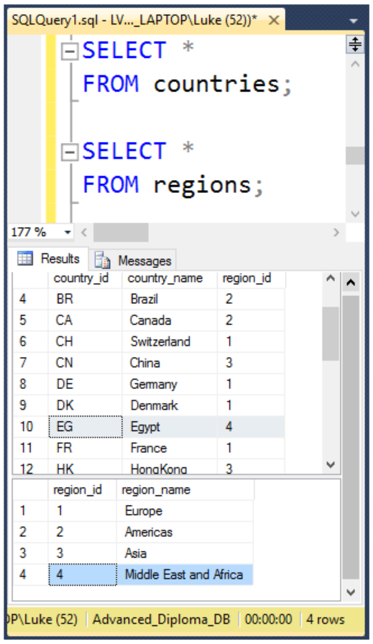


```sql
DELETE FROM departments
WHERE department_name = 'Payroll'
```

	department_id	department_name	manager_id	location_id
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700

\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 28 rows

Example 18: Write a query that will delete all the countries in the countries table which include an 'and' in the region name. The Delete query is using a sub-query which is a query within another query.

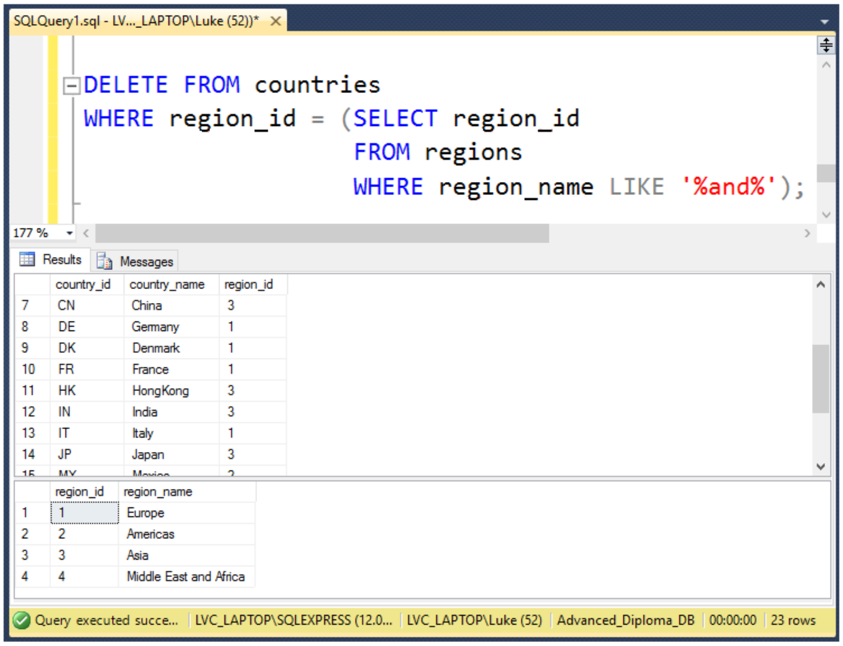


```sql
SELECT *
FROM countries;

SELECT *
FROM regions;
```

	country_id	country_name	region_id
4	BR	Brazil	2
5	CA	Canada	2
6	CH	Switzerland	1
7	CN	China	3
8	DE	Germany	1
9	DK	Denmark	1
10	EG	Egypt	4
11	FR	France	1
12	HK	HongKong	3

	region_id	region_name
1	1	Europe
2	2	Americas
3	3	Asia
4	4	Middle East and Africa

P\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 4 rows



```sql
DELETE FROM countries
WHERE region_id = (SELECT region_id
                   FROM regions
                   WHERE region_name LIKE '%and%');
```

	country_id	country_name	region_id
7	CN	China	3
8	DE	Germany	1
9	DK	Denmark	1
10	FR	France	1
11	HK	HongKong	3
12	IN	India	3
13	IT	Italy	1
14	JP	Japan	3
15	MY	Mexico	2

	region_id	region_name
1	1	Europe
2	2	Americas
3	3	Asia
4	4	Middle East and Africa

Query executed succe... | LVC_LAPTOP\SQLEXPRESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 23 rows

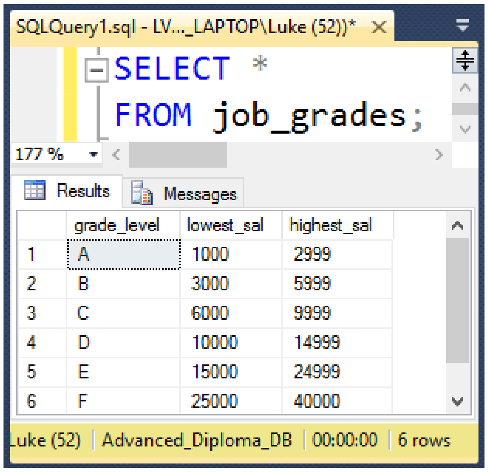
Note that 6 rows will be deleted and in the screen shot above one of the countries which was deleted (Egypt) is being displayed.

TRUNCATE statement

TRUNCATE is the alternative of the DELETE statement when a user intends to remove rows of data from a table. Unlike the DELETE option, this statement does not allow the use of the WHERE clause, therefore you either delete all the rows or nothing. Once that you use the TRANCATE statement the target table will become empty. The syntax for the TRUNCATE statement is listed below:

```
TRUNCATE TABLE table
```

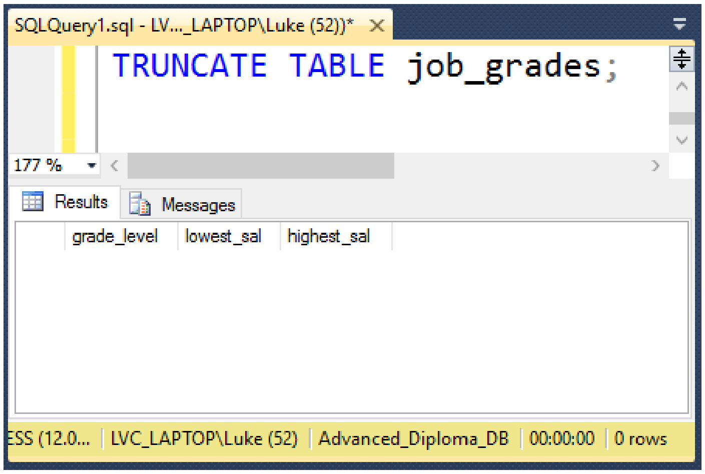
Example 19: Write a query that will delete all the rows in the job_grades entity



```
SELECT *
FROM job_grades;
```

	grade_level	lowest_sal	highest_sal
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000

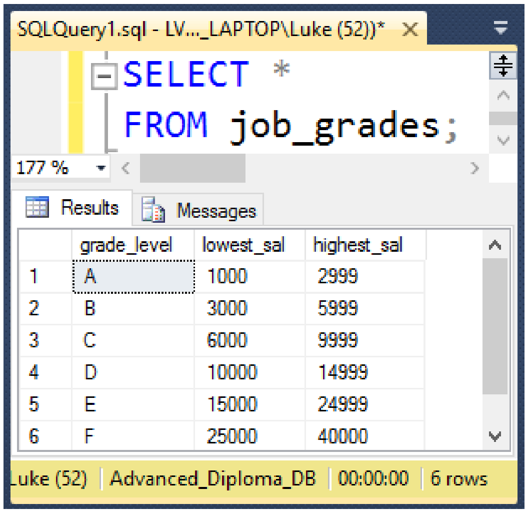
Luke (52) | Advanced_Diploma_DB | 00:00:00 | 6 rows



```
TRUNCATE TABLE job_grades;
```

grade_level	lowest_sal	highest_sal

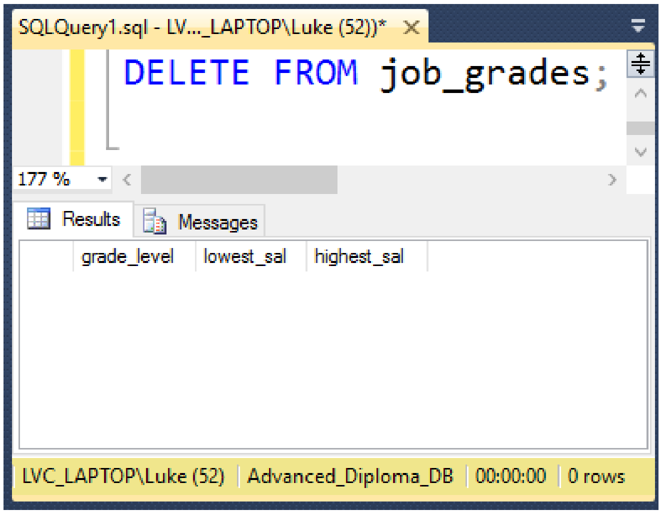
ESS (12.0... | LVC_LAPTOP\Luke (52) | Advanced_Diploma_DB | 00:00:00 | 0 rows



```
SELECT *
FROM job_grades;
```

	grade_level	lowest_sal	highest_sal
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000

Luke (52) | Advanced_Diploma_DB | 00:00:00 | 6 rows



```
DELETE FROM job_grades;
```

grade_level	lowest_sal	highest_sal

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

Note that the two queries above perform the exact same thing