SQL PROJECT:

Exercise 1:

Dr. Daniel Salinas, Minister of Public Health, has learned that you are taking a course in Data Analysis with SQL and trusts that you will be able to help model and design his future database.

From an interview you obtain the following information: in this database he wants to be able to save information related to hospitals, therefore it is required to store at least the address, telephone numbers, name of each hospital and type, for example: if it is specialized.

Each one has several rooms, so you want to store their code, name and beds quantity.

Each hospital can have several rooms and each of them belongs to a single hospital.

In different hospitals there may be rooms with the same code, but this cannot happen within the same hospital.

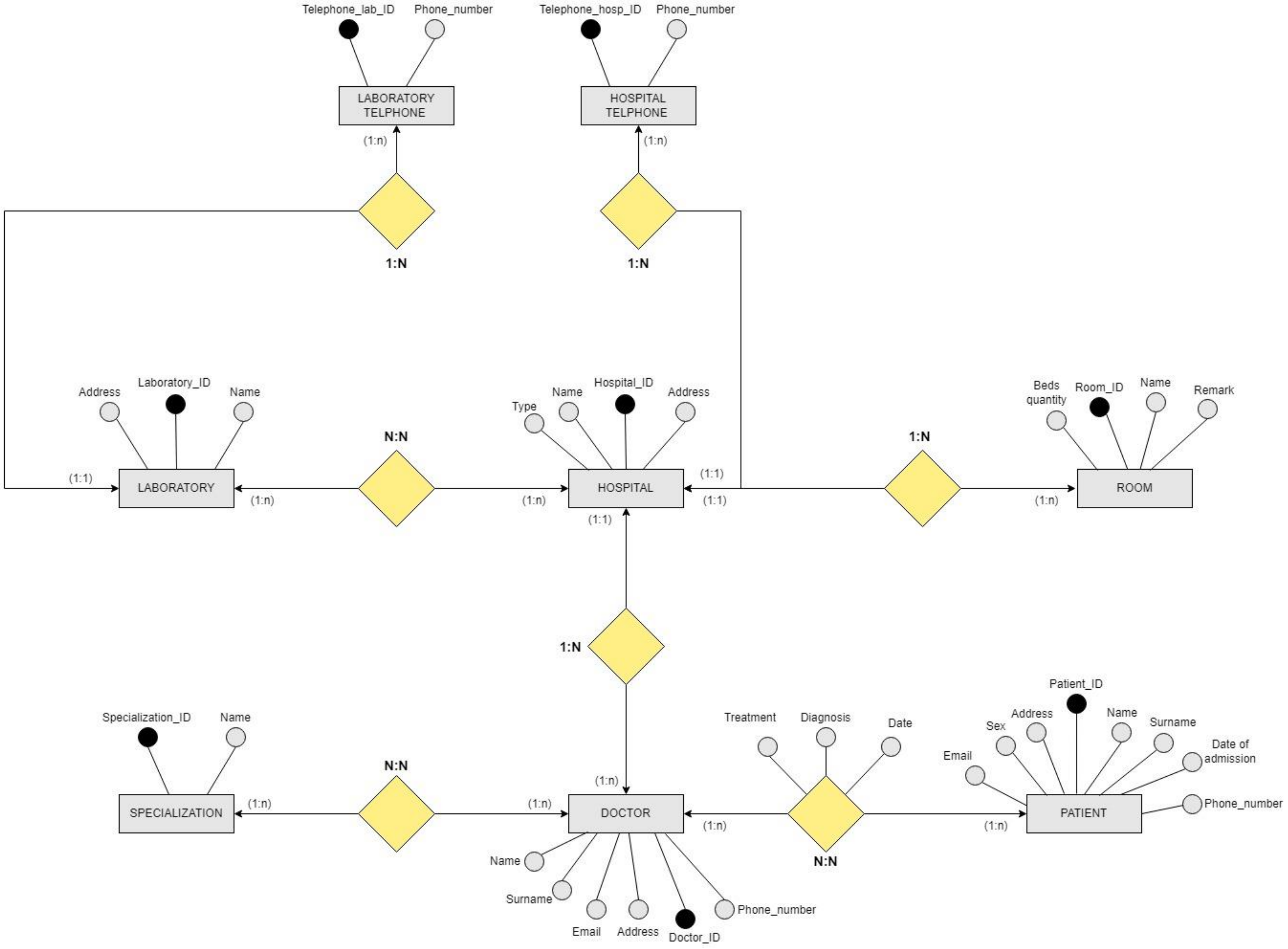
You also want to store basic information about each doctor and their specializations. Because hospitals compete with each other for a quality award, their doctors are full-time.

It is vitally important to have the patient's information, in addition to the basic one, you want to have a history of the patient's diagnoses, and their date, as well as which doctor was responsible for each diagnosis.

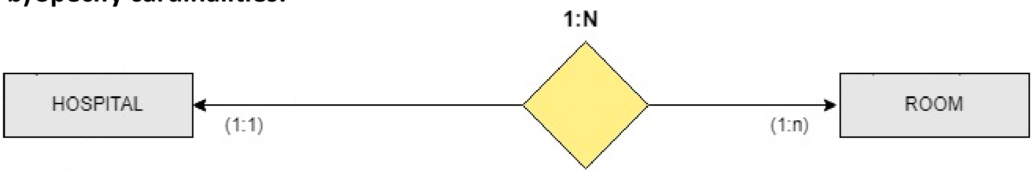
In turn, there are laboratories that provide services to hospitals. It is required to store the name, address and telephone number of each laboratory. Hospitals may work with more than one laboratory. In the first instance, it is not known how many telephones each hospital and each laboratory may have, so it is necessary to be able to store all of them.

The following is requested:

a) Design the Entity Relationship Model that represents the previous scenario.



b) Specify cardinalities.

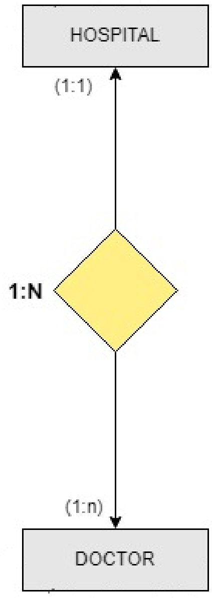


Each hospital has multiple rooms, and one room belongs to only one hospital.

Since the HOSPITAL entity has cardinality (1,1), the HOSPITAL-ROOM relationship does not generate a table, but the PK of HOSPITAL is passed as FK into the table of the ROOM entity.

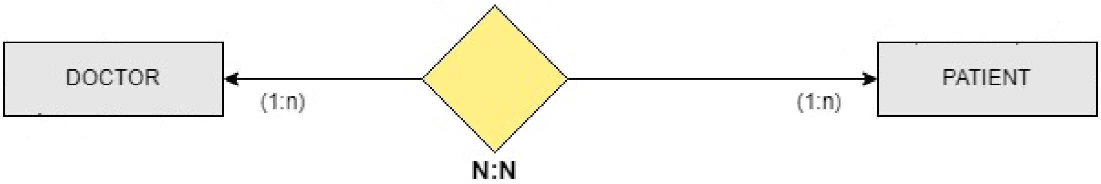
A compound PK is used when no field meets the condition on its own, then, as in different hospitals there may be rooms with the same code, but not within the same hospital.

The PK of this entity is formed from Room_ID and Hospital_ID fields.



Several doctors work in a hospital, but because hospitals compete with each other for a quality award, their doctors are full-time doctors, so each doctor works in only one hospital.

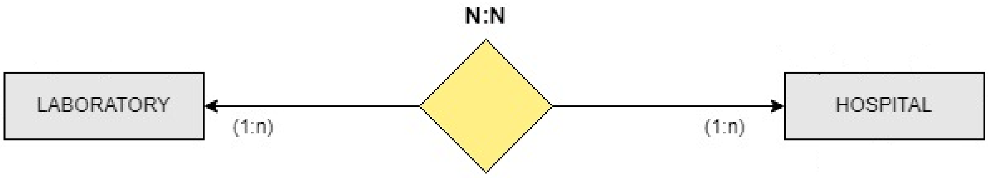
Since the HOSPITAL entity has cardinality (1,1), the HOSPITAL-DOCTOR relationship does not generate a table, but the PK of HOSPITAL is passed as FK in the DOCTOR entity table.



A doctor checks on several patients and a patient is seen by several doctors.

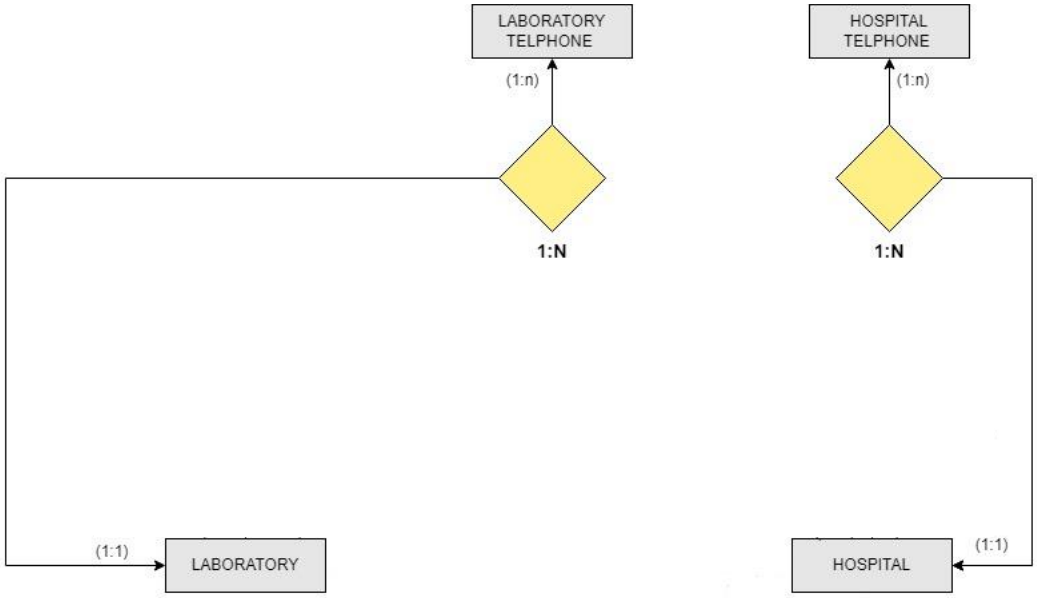
In this case, the DOCTOR-PATIENT relationship generates a table because it has N:N cardinality.

The relationship will keep its attributes and also inherit the PKs from both tables.



A laboratory supplies more than one hospital, and a hospital may work with more than one laboratory.

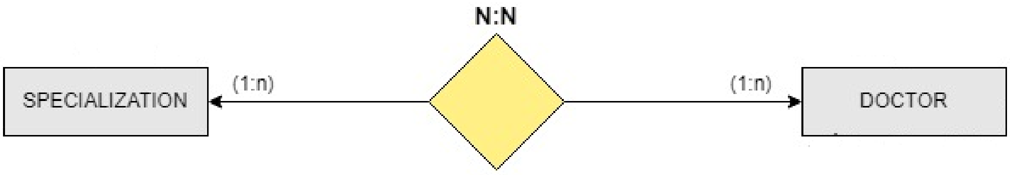
In this case, the LABORATORY-HOSPITAL relationship generates a table because it has N:N cardinality. The relationship inherits the PKs from both tables.



Since it is not known how many telephones each hospital and each laboratory may have, it is necessary to be able to store them all. So, a lab owns multiple phones, and a phone belongs to a lab. A hospital owns several telephones, and a telephone belongs to a hospital.

We define two telephone tables, one for laboratories and the other for hospitals; not a single telephone table for both to avoid having many fields with null values and thus optimize queries.

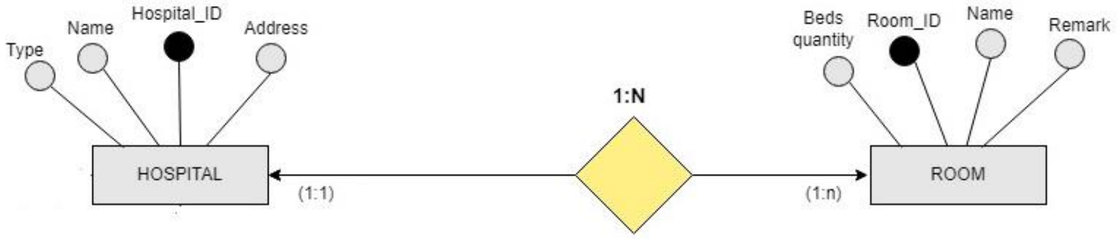
Since the entities HOSPITAL and LABORATORY have cardinality (1,1), the relationship LABORATORY-LABORATORY PHONE and HOSPITAL-HOSPITAL PHONE does not generate a table, but the PK of HOSPITAL goes as FK in the entity table HOSPITAL PHONE and the PK of LABORATORY goes as FK in the entity table LABORATORY PHONE.



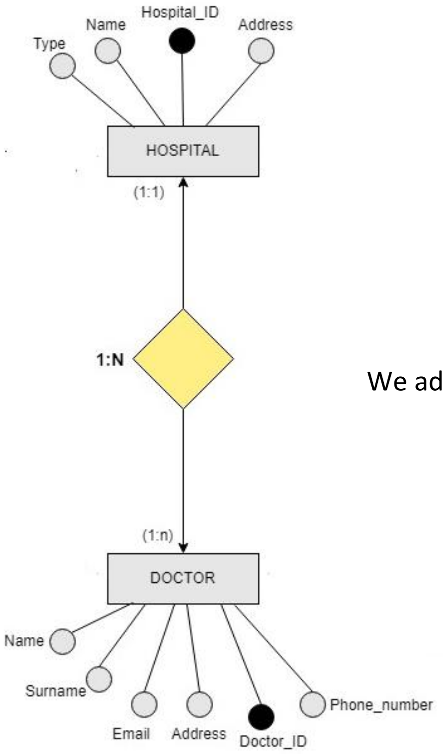
It is also wanted to store basic information of each doctor and their specializations, so each doctor has several specializations, and one specialization belongs to several doctors.

In this case, the DOCTOR-SPECIALIZATION relationship generates a table because it has N:N cardinality. The relationship inherits the PKs from both tables.

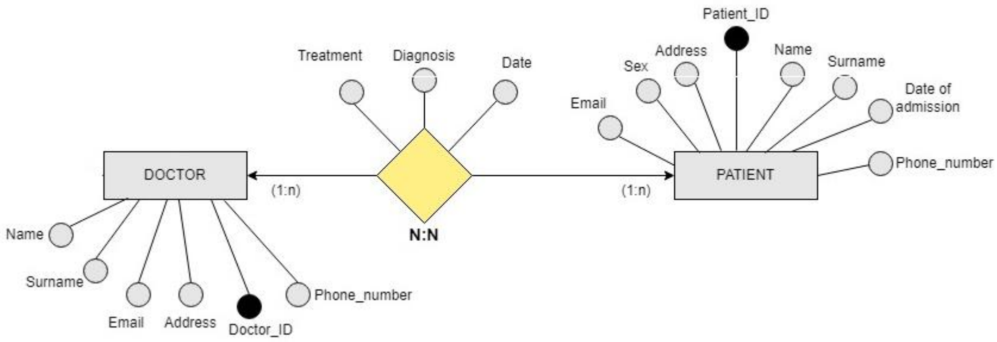
c) Define at least four attributes for each entity.



The **Remark** attribute refers to whether the room is shared or private. The **Type** attribute means whether the hospital is specialized or not.

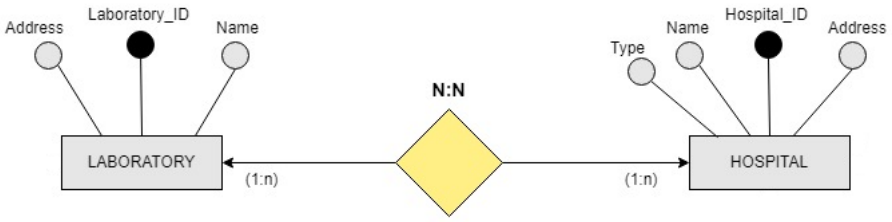


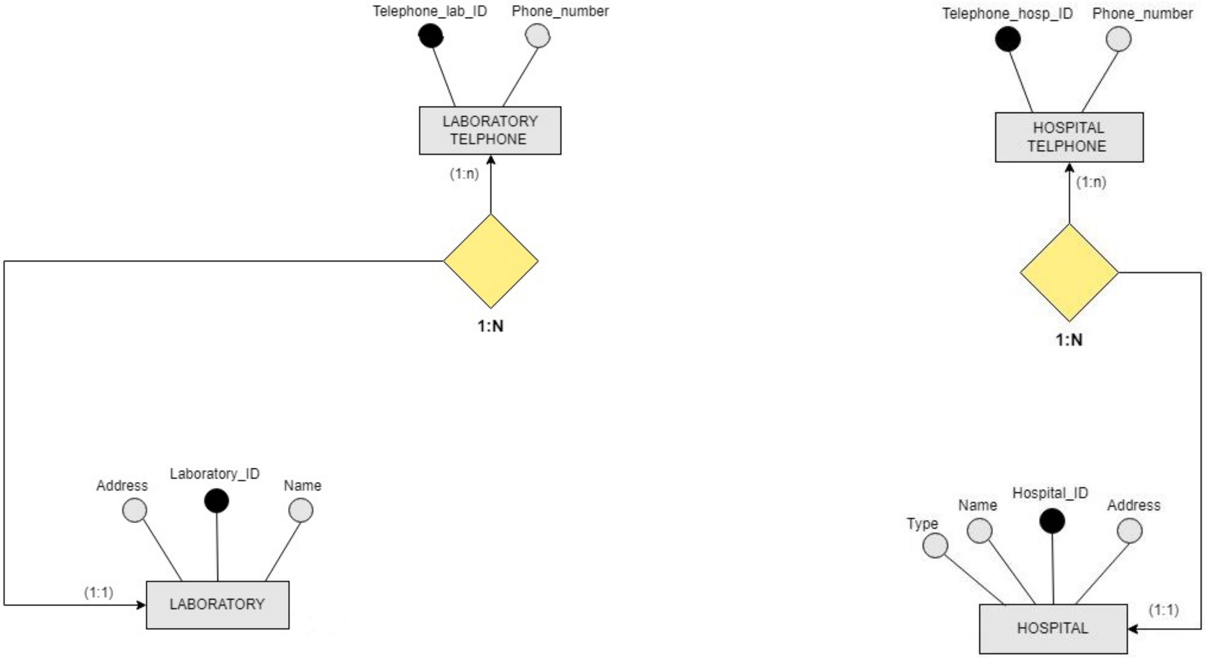
We add the **Email** attribute to the DOCTOR entity.

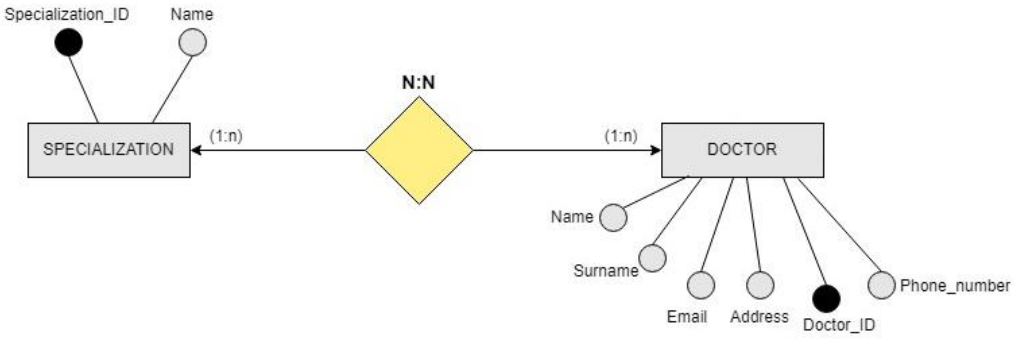


To learn more about the patient, we add the attributes: Email, Gender and Date of admission. As it is important to have the history of the patient's diagnoses and their dates, we place these attributes in the DOCTOR-PATIENT relationship, in addition to the Treatment attribute.

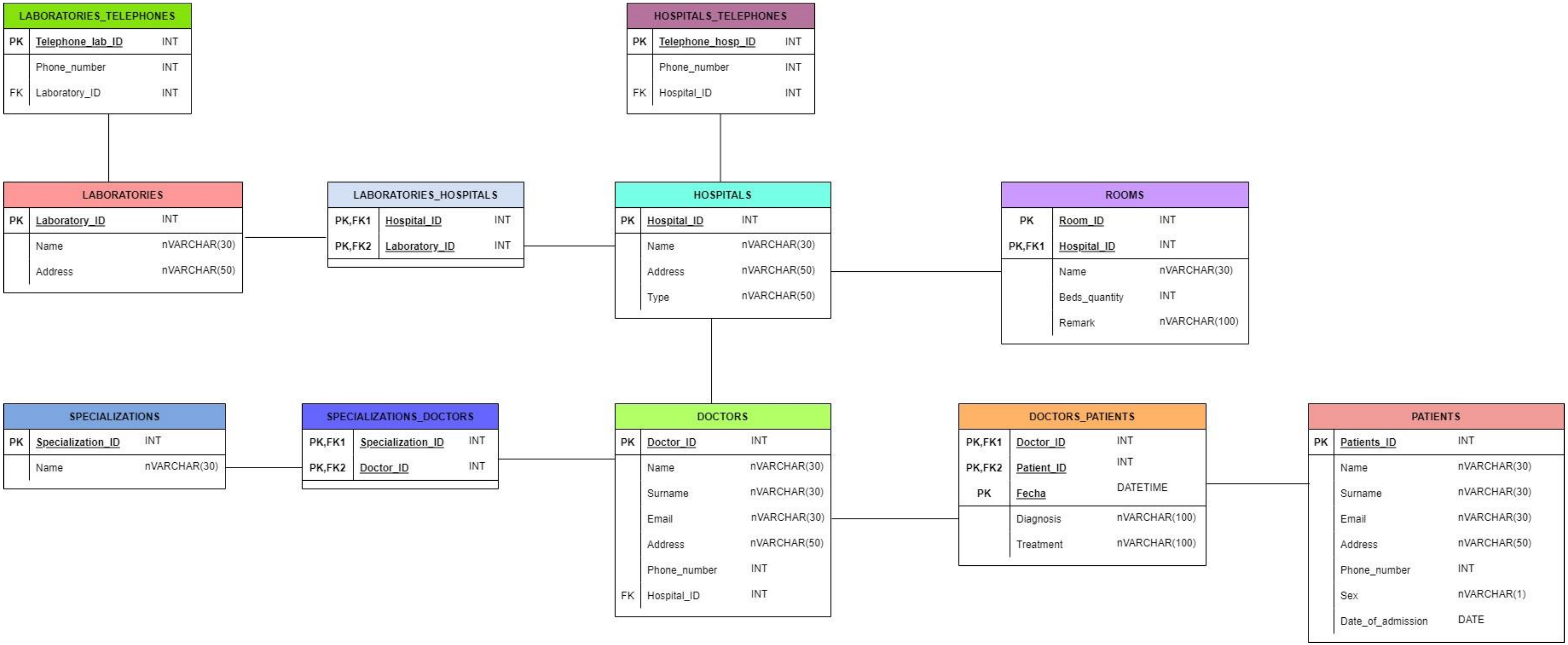
The attributes Doctor_ID and Patient_ID could be replaced by the person’s "ID" number, since it is a field identifier that will not be repeated.







d) Present the tables with their fields, unique keys, foreign keys and data types that emerge from the Entity Relationship Model made in points a, b and c.



LABORATORIES_TELEPHONES		
PK	Telephone_lab_ID	INT
	Phone_number	INT
FK	Laboratory_ID	INT

HOSPITALS_TELEPHONES		
PK	Telephone_hosp_ID	INT
	Phone_number	INT
FK	Hospital_ID	INT

LABORATORIES		
PK	Laboratory_ID	INT
	Name	nVARCHAR(30)
	Address	nVARCHAR(50)

LABORATORIES_HOSPITALS		
PK,FK1	Hospital_ID	INT
PK,FK2	Laboratory_ID	INT

HOSPITALS		
PK	Hospital_ID	INT
	Name	nVARCHAR(30)
	Address	nVARCHAR(50)
	Type	nVARCHAR(50)

ROOMS		
PK	Room_ID	INT
PK,FK1	Hospital_ID	INT
	Name	nVARCHAR(30)
	Beds_quantity	INT
	Remark	nVARCHAR(100)

SPECIALIZATIONS		
PK	Specialization_ID	INT
	Name	nVARCHAR(30)

SPECIALIZATIONS_DOCTORS		
PK,FK1	Specialization_ID	INT
PK,FK2	Doctor_ID	INT

DOCTORS		
PK	Doctor_ID	INT
	Name	nVARCHAR(30)
	Surname	nVARCHAR(30)
	Email	nVARCHAR(30)
	Address	nVARCHAR(50)
	Phone_number	INT
FK	Hospital_ID	INT

DOCTORS_PATIENTS		
PK,FK1	Doctor_ID	INT
PK,FK2	Patient_ID	INT
PK	Fecha	DATETIME
	Diagnosis	nVARCHAR(100)
	Treatment	nVARCHAR(100)

PATIENTS		
PK	Patients_ID	INT
	Name	nVARCHAR(30)
	Surname	nVARCHAR(30)
	Email	nVARCHAR(30)
	Address	nVARCHAR(50)
	Phone_number	INT
	Sex	nVARCHAR(1)
	Date_of_admission	DATE

Exercise 2:

1) Given the following query:

```
SELECT c.CompanyName, SUM(soh.TotalDue)
AS TotalFROM SalesLT .Customer AS c
LEFT JOIN SalesLT.SalesOrderHeader
AS sohON c.CustomerID =
soh.CustomerID
GROUP BY
c.CompanyName
ORDER BY Total
DESC
```

a) Explain conceptually what it returns (information).

The information obtained with this query is the names of the companies of all the clients, whether or not they have made purchases, and the sum of all of them, ordered from highest to lowest.

b) Explain with what intention the LEFT JOIN clause was used.

In this case, a LEFT JOIN was used because we want to display the companies of all existing customers in the **Customer** table, whether or not they are in the **SalesOrderHeader** table.

The **Customer** table shows the information of the names of the companies in which the clients buy and the **SalesOrderHeader** table indicates the information of the purchase orders and also shows the amount consumed by each client.

With LEFT JOIN we give priority to the left table and search in the right table. If there is no match for any of the rows in the left table, all results from the first table are displayed in the same way.

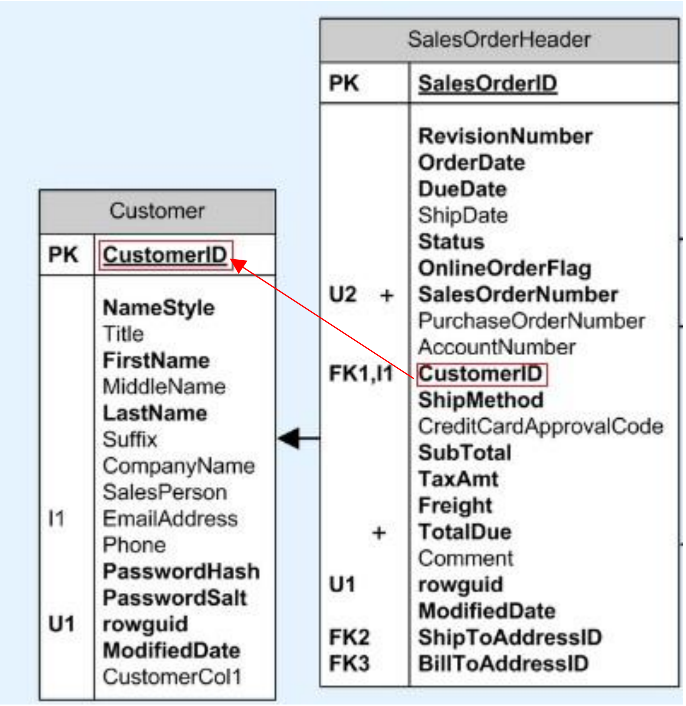
The **Customer** table is the LEFT table (renamed with the alias "c"), so all of its rows will show up in the results.

The **SalesOrderHeader** table (renamed with the alias "soh") is the table on the right since it appears after the LEFT JOIN, therefore, if matches are found, the corresponding values will be shown, but otherwise NULL will appear in the results.

In turn, the LEFT JOIN fetches whatever is at the intersection of those two tables. The two tables are joined by specifying the field in common, in this case that field is the PK in the **Customer** table which is in turn the FK in the **SalesOrderHeader** table.

c.CustomerID = soh.CustomerID

This tells us that the CustomerID field in the Customer table is equal to the CustomerID field in the SalesOrderHeader table.



c) Explain why the GROUP BY clause was used.

```
GROUP BY c.CompanyName
```

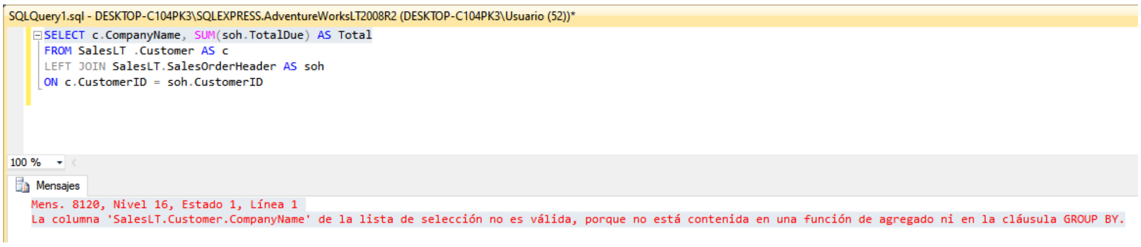
Group by the CompanyName field to show total purchases by company. That is, it will add the number of purchases that there are in each company.

If we execute the following query, SQL will show us the total of the sum of all customer purchases in all companies, as a single value.

```
SELECT SUM(soh.TotalDue) AS Total
FROM SalesLT .Customer AS c

LEFT JOIN SalesLT.SalesOrderHeader AS soh
ON c.CustomerID = soh.CustomerID
```

If we leave the query without the GROUP BY, we are telling SQL to fetch the name of the company, so there are going to be several registers, and on the other hand we tell it to fetch a register that is the sum of all the purchases of clients, so there will be an error.

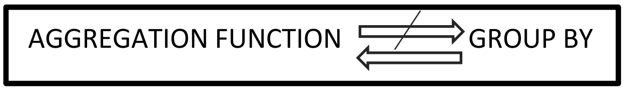
```
SELECT c.CompanyName, SUM(soh.TotalDue)
AS Total

FROM SalesLT .Customer AS c

LEFT JOIN SalesLT.SalesOrderHeader AS soh
```


SQLQuery1.sql - DESKTOP-C104PK3\SQLEXPRESS.AdventureWorksLT2008R2 (DESKTOP-C104PK3\Usuario (52))*
100 %
Mensajes

```sql
SELECT c.CompanyName, SUM(soh.TotalDue) AS Total
FROM SalesLT .Customer AS c
LEFT JOIN SalesLT.SalesOrderHeader AS soh
ON c.CustomerID = soh.CustomerID
```

```
Mens. 8120, Nivel 16, Estado 1, Línea 1
La columna 'SalesLT.Customer.CompanyName' de la lista de selección no es válida, porque no está contenida en una función de agregado ni en la cláusula GROUP BY.
```

So, every time we have an aggregation function in a SELECT (a sum in this case), we're going to have to use a GROUP BY clause, but we can use a GROUP BY without having an aggregation function. It must be grouped by the field that is not found in the aggregation function.



Finally, the values of the totals are ordered in descending order to display the companies with the highest spending first. We can refer to the alias “Total” because ORDER BY is the last thing that executes.

```sql
ORDER BY Total DESC
```

FROM clause will be executed first, then ON, then GROUP BY, then the SELECT and finally ORDER BY.

2) Given the following query:

```
SELECT count(c.CustomerID) AS 'Number of Customers', a.City
FROM SalesLT.Address AS a
INNER JOIN SalesLT.CustomerAddress AS ca
ON a.AddressID=ca.AddressID
INNER JOIN SalesLT.Customer AS c
ON c.CustomerID=ca.CustomerID
WHERE a.StateProvince='Texas'
GROUP BY a.City
HAVING count(c.CustomerID) > 3
ORDER BY 'Number of Customers' DESC
```

a) Explain conceptually what it returns (information).

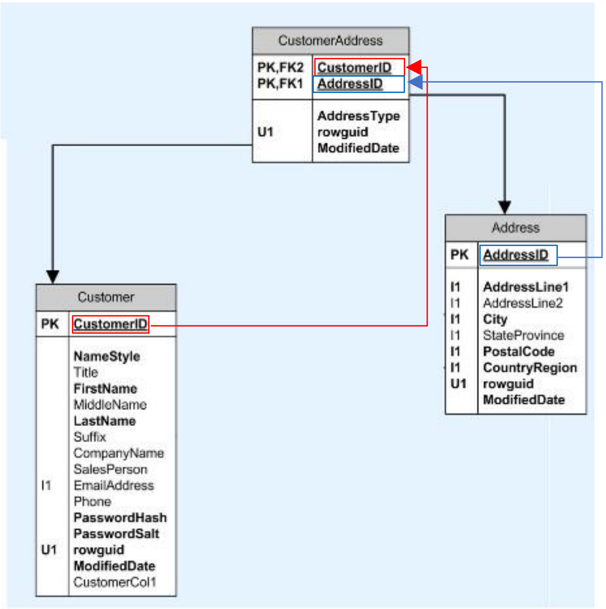
The information obtained with this query is the number of clients for each city, where the state is Texas but only bringing the cities that have more than 3 clients, grouped by city and ordered from highest to lowest.

b) Explain with what intention the INNER JOIN clause was used.

The **Address** table (renamed with alias “a”) shows all addresses by state and city, the **Customer** table (renamed with the alias “c”) shows customer information, and the **CustomerAddress** table (renamed with the alias “ca ”) is the relationship between the two previous tables; the existence of this last table occurs because the **Address** and **Customer** tables have N:N cardinality, so the PK of the previously mentioned tables become PK and FK in the relationship.

This clause searches for matches between 2 tables based on a column they have in common, so only the intersection will be shown in the results.

Between **Address** and **CustomerAddress** an **INNER JOIN** is used to show only the addresses of the clients and then an **INNER JOIN** is used again with **Customer** to show only who are those clients.



The tables are joined specifying their field in common, between the **Address** and **CustomerAddress** tables that field is the PK of the **Address** table, which in turn is the FK of the **CustomerAddress** table.

a.AddressID = ca.AddressID

This tells us that the AddressID field of the Address table is equal to the AddressID field of the CustomerAddress table.

It is then joined to the **Customer** table using its PK which is FK in the **CustomerAddress** table.

c.CustomerID=ca.CustomerID

This tells us that the CustomerID field in the Customer table is equal to the CustomerID field in the CustomerAddress table.

c) Explain what the HAVING clause was used for and why was this not done in the WHERE clause.

WHERE is applied to individual registers, that is, register by register in the table, while with HAVING a condition can be established on a group of registers, accompanied by the GROUP BY clause, when you want to perform an extra filter on an aggregation function.

```
WHERE a.StateProvince='Texas'
```

This sentence tells us that it will be filtered according to the StateProvince field of the **Address** table and only the records that say "Texas" will be shown.

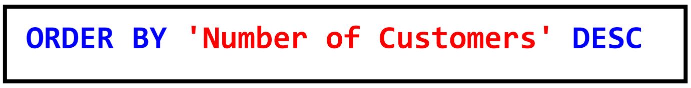
```
GROUP BY a.City
```

It is grouped by the City field to show the cities in Texas and the number of customers each has.

```
HAVING count(c.CustomerID) > 3
```

For this second filter we cannot use another WHERE because the data has already been processed in the first one and then grouped in the GROUP BY, so we must use HAVING. Also, since we have to filter using the aggregation function, we know that we are definitely going to need a HAVING. Here we indicate that we want to keep only the cities that have more than three clients.

Finally, the number of customers is ordered in descending order to display the cities with the largest number of customers first. We can refer to the alias “Number of Customers” because ORDER BY is the last thing executed.



```
ORDER BY 'Number of Customers' DESC
```

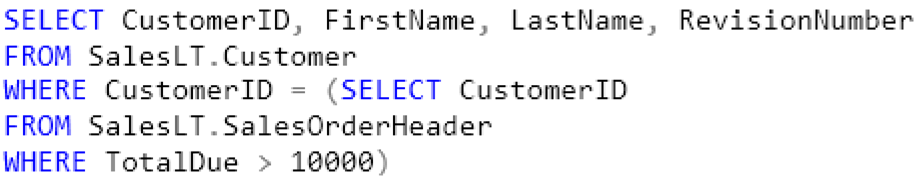
First the FROM clause will be executed, then ON, then WHERE, then GROUP BY, then HAVING, then SELECT and finally ORDER BY.

Exercise 3:

Given the following queries solve:

- If they are correct or not, justify if they are not and proceed to correct them.
- Explain conceptually what it returns (what information).

a)

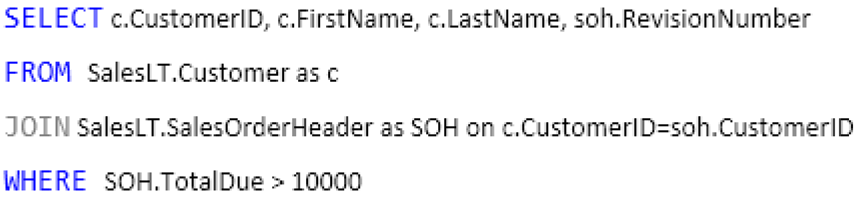


```
SELECT CustomerID, FirstName, LastName, RevisionNumber
FROM SalesLT.Customer
WHERE CustomerID = (SELECT CustomerID
FROM SalesLT.SalesOrderHeader
WHERE TotalDue > 10000)
```

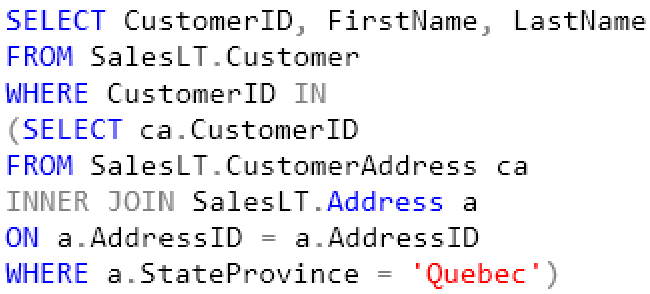
The query is not correct because there is no column called "RevisionNumber" in the Customer table, it does exist in the SalesOrderHeader table.

The WHERE clause must be followed by an IN, because the subquery returns a list and not a single value.

The information it returns is the identification of the clients, their name and surname where they spent over 10000.



```
SELECT c.CustomerID, c.FirstName, c.LastName, soh.RevisionNumber

FROM  SalesLT.Customer as c

JOIN SalesLT.SalesOrderHeader as SOH on c.CustomerID=soh.CustomerID

WHERE  SOH.TotalDue > 10000
```

b) 

```
SELECT CustomerID, FirstName, LastName
FROM SalesLT.Customer
WHERE CustomerID IN
(SELECT ca.CustomerID
FROM SalesLT.CustomerAddress ca
INNER JOIN SalesLT.Address a
ON a.AddressID = a.AddressID
WHERE a.StateProvince = 'Quebec')
```

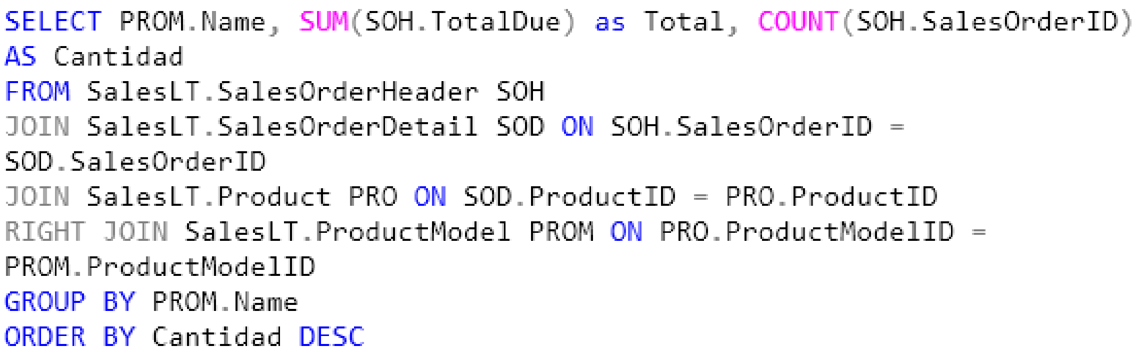
The query is incorrect, because "ON" is referencing a.AddressID twice, the correct way would be: ON ca.AddressID = a.AddressID

It returns the identification of the clients, their name and surname where the state is "Quebec". In this case, a subquery is used.

Between **Address** and **CustomerAddress** an **INNER JOIN** is used to show only the customer addresses and the table is filtered according to the StateProvince field to show only those that say Quebec. This results in a list of customer IDs that will match the customer IDs in the **Customer** table.

Exercise 4:

You want to obtain a report with the total billing and number of invoices issued for each product model (show name) ordered in descending order by number of invoices (show the information of all models with or without an invoice issued).



```
SELECT PROM.Name, SUM(SOH.TotalDue) as Total, COUNT(SOH.SalesOrderID)
AS Cantidad
FROM SalesLT.SalesOrderHeader SOH
JOIN SalesLT.SalesOrderDetail SOD ON SOH.SalesOrderID =
SOD.SalesOrderID
JOIN SalesLT.Product PRO ON SOD.ProductID = PRO.ProductID
RIGHT JOIN SalesLT.ProductModel PROM ON PRO.ProductModelID =
PROM.ProductModelID
GROUP BY PROM.Name
ORDER BY Cantidad DESC
```

The **SalesOrderHeader** table stores information about a sale made and the **SalesOrderDetail** table stores data about a product that is sold, providing data such as quantity sold and unit price.

We use a JOIN between the **SalesOrderHeader** and **SalesOrderDetail** tables to get the sales that have the detail about the products. Then we use another JOIN with the **Product** table to see the products that have details.

Finally, we use a RIGHT JOIN with the **ProductModel** table to obtain the information of all the models, whether or not they have an invoice issued.

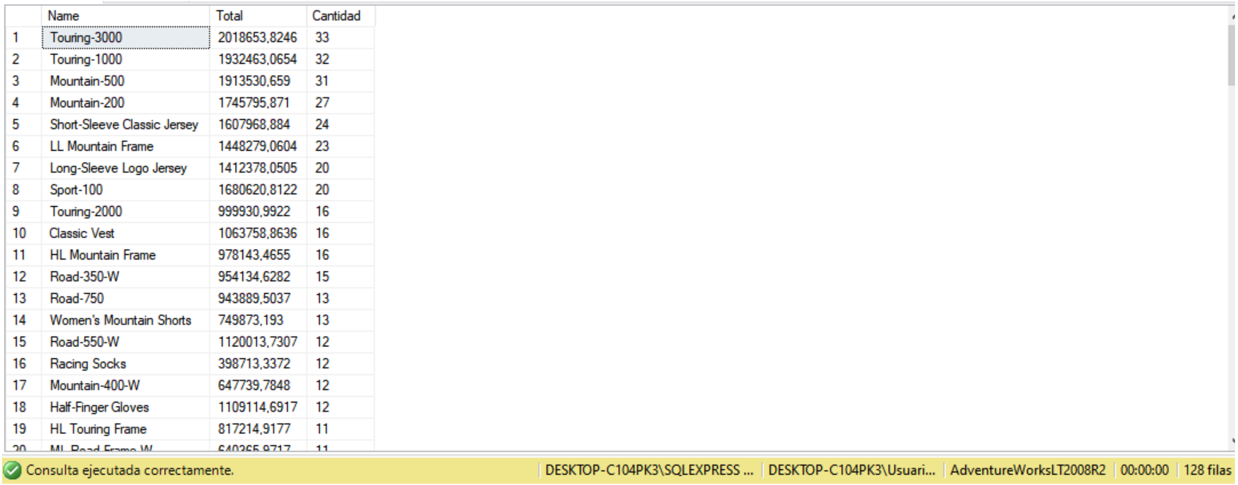
```
GROUP BY PROM.Name
```

It is grouped by the Name field of the ProductModel table to show the total billing according to each model and its respective quantity.

```
ORDER BY Cantidad DESC
```

Lastly, we order in descending order by number of invoices. We can refer to the alias "Cantidad" because ORDER BY is the last thing executed. Cantidad = Amount.

The output of the query is displayed below:

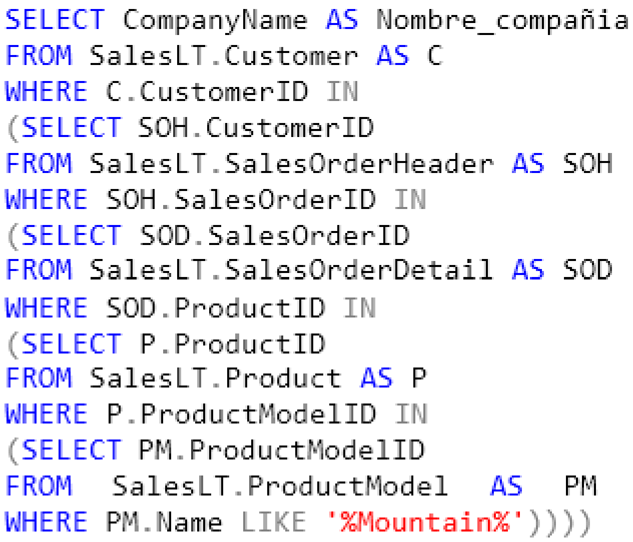


	Name	Total	Cantidad
1	Touring-3000	2018653,8246	33
2	Touring-1000	1932463,0654	32
3	Mountain-500	1913530,659	31
4	Mountain-200	1745795,871	27
5	Short-Sleeve Classic Jersey	1607968,884	24
6	LL Mountain Frame	1448279,0604	23
7	Long-Sleeve Logo Jersey	1412378,0505	20
8	Sport-100	1680620,8122	20
9	Touring-2000	999930,9922	16
10	Classic Vest	1063758,8636	16
11	HL Mountain Frame	978143,4655	16
12	Road-350-W	954134,6282	15
13	Road-750	943889,5037	13
14	Women's Mountain Shorts	749873,193	13
15	Road-550-W	1120013,7307	12
16	Racing Socks	398713,3372	12
17	Mountain-400-W	647739,7848	12
18	Half-Finger Gloves	1109114,6917	12
19	HL Touring Frame	817214,9177	11
20	ML Road Frame-W	640365,9717	11

Consulta ejecutada correctamente. | DESKTOP-C104PK3\SQLEXPRESS ... | DESKTOP-C104PK3\Usuari... | AdventureWorksLT2008R2 | 00:00:00 | 128 filas

Exercise 5:

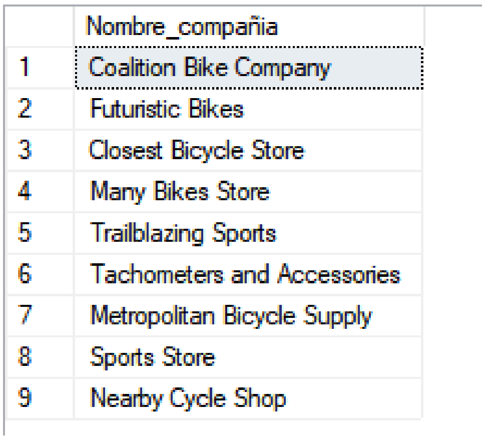
You want to obtain a report with the name of the companies of the clients who bought a product from Montaña. Do it only with sub-queries.



```
SELECT CompanyName AS Nombre_compañia
FROM SalesLT.Customer AS C
WHERE C.CustomerID IN
(SELECT SOH.CustomerID
FROM SalesLT.SalesOrderHeader AS SOH
WHERE SOH.SalesOrderID IN
(SELECT SOD.SalesOrderID
FROM SalesLT.SalesOrderDetail AS SOD
WHERE SOD.ProductID IN
(SELECT P.ProductID
FROM SalesLT.Product AS P
WHERE P.ProductModelID IN
(SELECT PM.ProductModelID
FROM  SalesLT.ProductModel  AS  PM
WHERE PM.Name LIKE '%Mountain%'))))
```

Nombre_compañia = Company_name

The output of the query is displayed below:

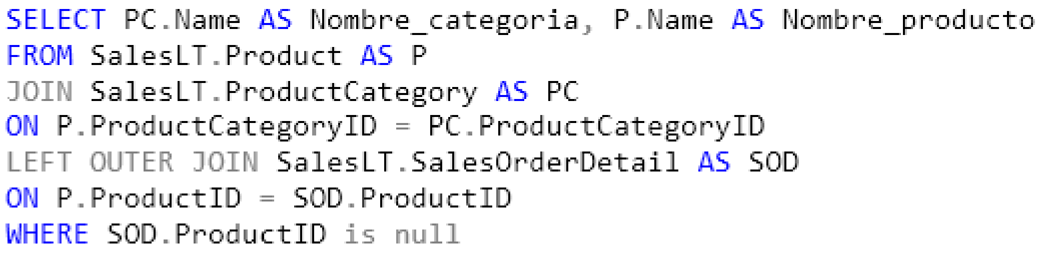


	Nombre_compañia
1	Coalition Bike Company
2	Futuristic Bikes
3	Closest Bicycle Store
4	Many Bikes Store
5	Trailblazing Sports
6	Tachometers and Accessories
7	Metropolitan Bicycle Supply
8	Sports Store
9	Nearby Cycle Shop

Exercise 6:

You want to get a report with the category name and the product name of the products that have not been ordered.

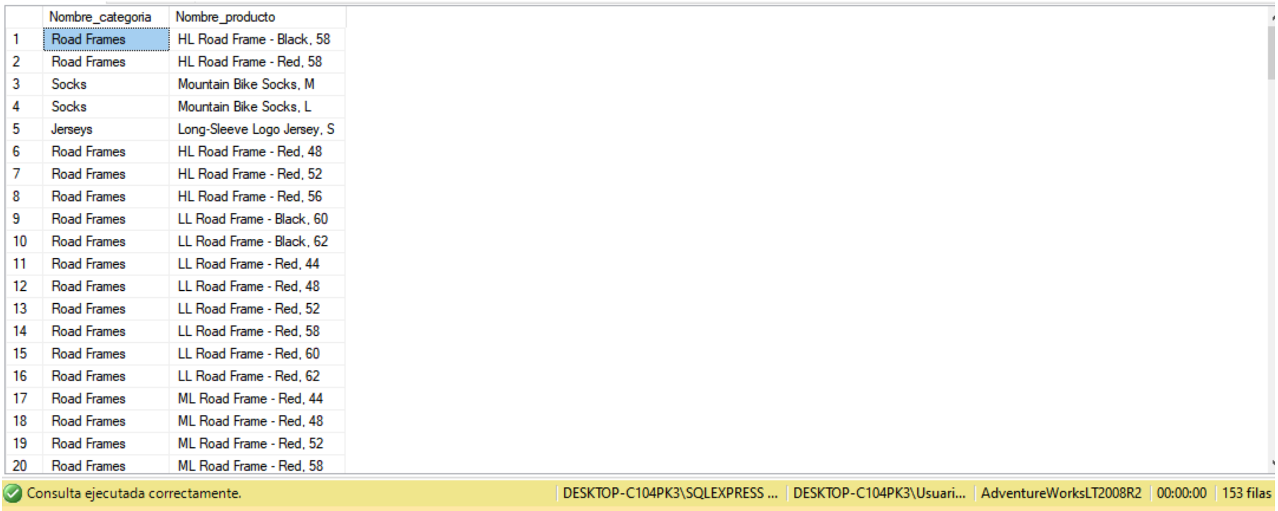
Nombre_producto = Product_name



```
SELECT PC.Name AS Nombre_categoria, P.Name AS Nombre_producto
FROM SalesLT.Product AS P
JOIN SalesLT.ProductCategory AS PC
ON P.ProductCategoryID = PC.ProductCategoryID
LEFT OUTER JOIN SalesLT.SalesOrderDetail AS SOD
ON P.ProductID = SOD.ProductID
WHERE SOD.ProductID is null
```

We use a JOIN between **Product** and **ProductCategory** tables to get the products and their respective categories. Then we do a LEFT OUTER JOIN with the **SalesOrderDetail** table, where the ProductID key of the SalesOrderDetail table is null to get the records that have a category but were not ordered.

The output of the query is displayed below:

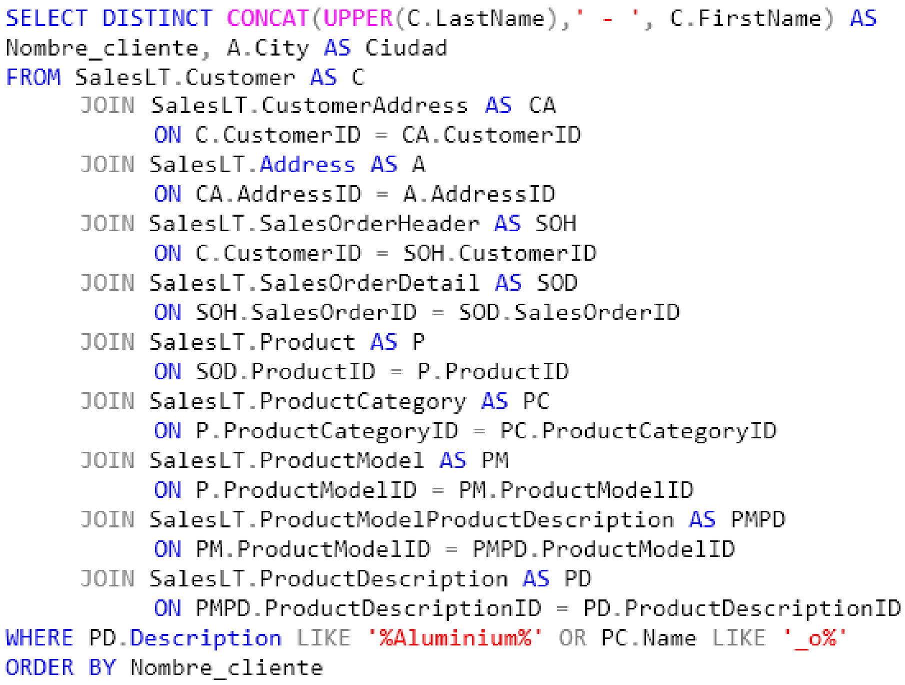


	Nombre_categoria	Nombre_producto
1	Road Frames	HL Road Frame - Black, 58
2	Road Frames	HL Road Frame - Red, 58
3	Socks	Mountain Bike Socks, M
4	Socks	Mountain Bike Socks, L
5	Jerseys	Long-Sleeve Logo Jersey, S
6	Road Frames	HL Road Frame - Red, 48
7	Road Frames	HL Road Frame - Red, 52
8	Road Frames	HL Road Frame - Red, 56
9	Road Frames	LL Road Frame - Black, 60
10	Road Frames	LL Road Frame - Black, 62
11	Road Frames	LL Road Frame - Red, 44
12	Road Frames	LL Road Frame - Red, 48
13	Road Frames	LL Road Frame - Red, 52
14	Road Frames	LL Road Frame - Red, 58
15	Road Frames	LL Road Frame - Red, 60
16	Road Frames	LL Road Frame - Red, 62
17	Road Frames	ML Road Frame - Red, 44
18	Road Frames	ML Road Frame - Red, 48
19	Road Frames	ML Road Frame - Red, 52
20	Road Frames	ML Road Frame - Red, 58

Consulta ejecutada correctamente. | DESKTOP-C104PK3\SQLEXPRESS ... | DESKTOP-C104PK3\Usuari... | AdventureWorksLT2008R2 | 00:00:00 | 153 filas

<u>Exercise 7:</u>

You want to obtain a report that shows the name, surname and city of customers who have purchased a product whose description contains the word "Aluminium" or whose category contains the letter "o" in the second position of the name. The first and last name must appear in a single column (concatenated) separated by a hyphen, and the last name must be in uppercase.



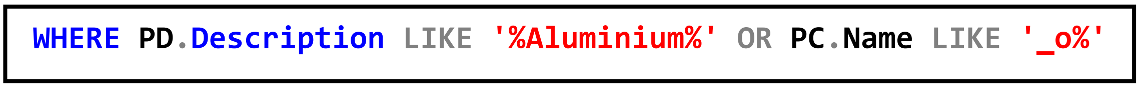
```sql
SELECT DISTINCT CONCAT(UPPER(C.LastName),' - ', C.FirstName) AS
Nombre_cliente, A.City AS Ciudad
FROM SalesLT.Customer AS C
     JOIN SalesLT.CustomerAddress AS CA
          ON C.CustomerID = CA.CustomerID
     JOIN SalesLT.Address AS A
          ON CA.AddressID = A.AddressID
     JOIN SalesLT.SalesOrderHeader AS SOH
          ON C.CustomerID = SOH.CustomerID
     JOIN SalesLT.SalesOrderDetail AS SOD
          ON SOH.SalesOrderID = SOD.SalesOrderID
     JOIN SalesLT.Product AS P
          ON SOD.ProductID = P.ProductID
     JOIN SalesLT.ProductCategory AS PC
          ON P.ProductCategoryID = PC.ProductCategoryID
     JOIN SalesLT.ProductModel AS PM
          ON P.ProductModelID = PM.ProductModelID
     JOIN SalesLT.ProductModelProductDescription AS PMPD
          ON PM.ProductModelID = PMPD.ProductModelID
     JOIN SalesLT.ProductDescription AS PD
          ON PMPD.ProductDescriptionID = PD.ProductDescriptionID
WHERE PD.Description LIKE '%Aluminium%' OR PC.Name LIKE '_o%'
ORDER BY Nombre_cliente
```

Ciudad = City

Nombre_cliente = Client_name

We use JOIN between the **Customer, CustomerAddress** and **Address** tables to link the customers with their addresses. Then we use JOIN with **SalesOrderHeader** and **SalesOrderDetail** tables to see the sales that have the detail about the products.

Then we use JOIN between **Product** and **ProductCategory** to get the category of each product and finally we use JOIN between **ProductModel**, **ProductModelProductDescription** and **ProductDescription** tables since it gives us the description of each product model.



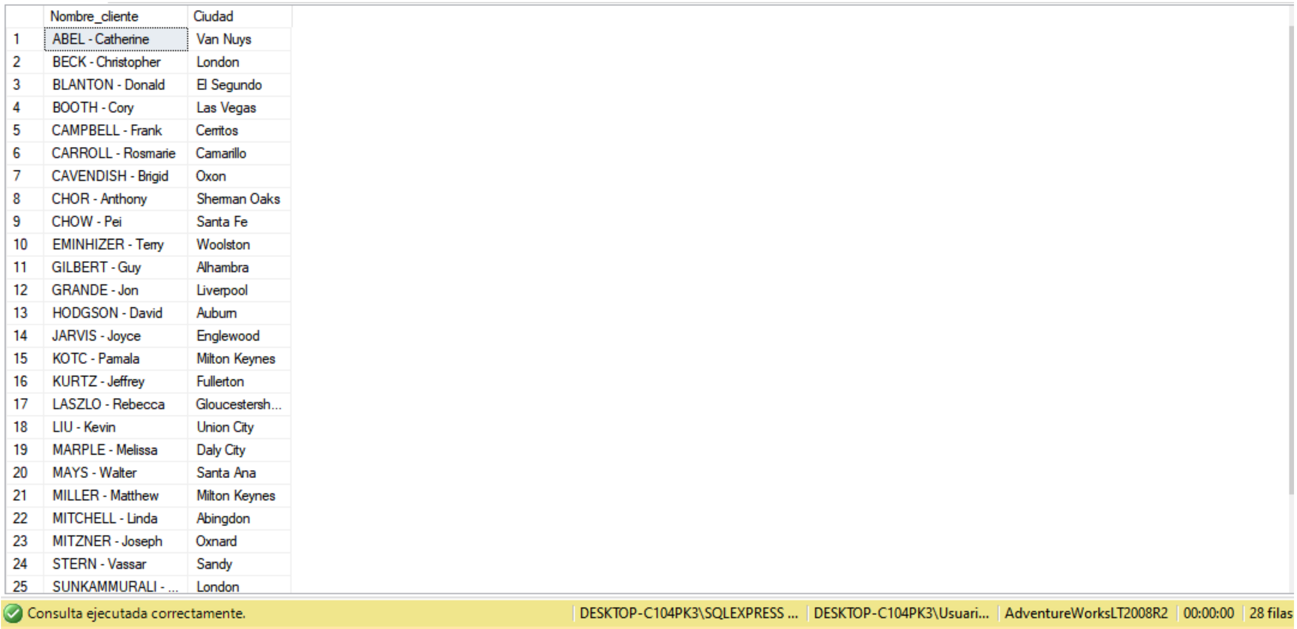
```sql
WHERE PD.Description LIKE '%Aluminium%' OR PC.Name LIKE '_o%'
```

This sentence tells us that it is going to be filtered according to the Description field of the **ProductDescription** table where the registers that contain the word "Aluminium" or the registers that contain the letter "o" in the second position of the name in the column Name of the **ProductCategory** table will be displayed.

In this case, a DISTINCT clause is added so as not to repeat records.

First the FROM clause will be executed, then the ON, then the WHERE, then the GROUP BY, then the SELECT, then the DISTINCT and finally the ORDER BY.

The output of the query is displayed below:



	Nombre_cliente	Ciudad
1	ABEL - Catherine	Van Nuys
2	BECK - Christopher	London
3	BLANTON - Donald	El Segundo
4	BOOTH - Cory	Las Vegas
5	CAMPBELL - Frank	Cerritos
6	CARROLL - Rosmarie	Camarillo
7	CAVENDISH - Brigid	Oxon
8	CHOR - Anthony	Sherman Oaks
9	CHOW - Pei	Santa Fe
10	EMINHIZER - Terry	Woolston
11	GILBERT - Guy	Alhambra
12	GRANDE - Jon	Liverpool
13	HODGSON - David	Auburn
14	JARVIS - Joyce	Englewood
15	KOTC - Pamala	Milton Keynes
16	KURTZ - Jeffrey	Fullerton
17	LASZLO - Rebecca	Gloucestersh...
18	LIU - Kevin	Union City
19	MARPLE - Melissa	Daly City
20	MAYS - Walter	Santa Ana
21	MILLER - Matthew	Milton Keynes
22	MITCHELL - Linda	Abingdon
23	MITZNER - Joseph	Oxnard
24	STERN - Vassar	Sandy
25	SUNKAMMURALI - ...	London

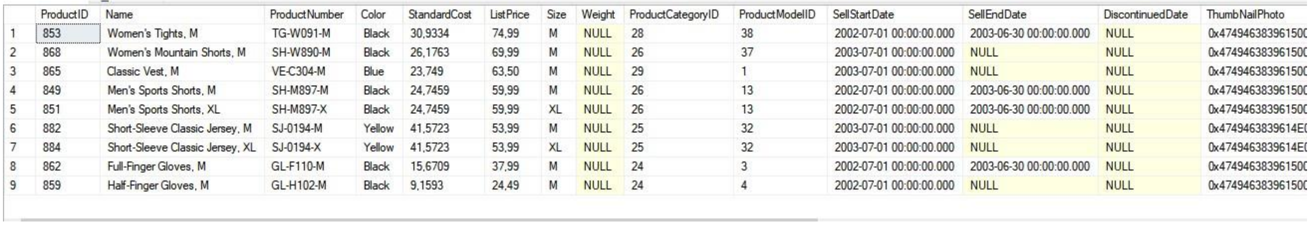
Consulta ejecutada correctamente. | DESKTOP-C104PK3\SQLEXPRESS ... | DESKTOP-C104PK3\Usuari... | AdventureWorksLT2008R2 | 00:00:00 | 28 filas

Exercise 8:

Obtain all the data of the products that have a list price between 10 and 500, are exclusively in Red, Blue, Yellow or Black, and that the size is M or XL. Sort them by list price from highest to lowest.

```
SELECT *
FROM SalesLT.Product PRO
WHERE PRO.ListPrice BETWEEN 10 AND 500 AND PRO.Color IN ('Red',
'Blue', 'Yellow', 'Black') AND PRO.Size IN ('M', 'XL')
ORDER BY ListPrice DESC
```

The output of the query is displayed below:



	ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size	Weight	ProductCategoryID	ProductModelID	SellStartDate	SellEndDate	DiscontinuedDate	ThumbNailPhoto
1	853	Women's Tights, M	TG-W091-M	Black	30,9334	74,99	M	NULL	28	38	2002-07-01 00:00:00.000	2003-06-30 00:00:00.000	NULL	0x47494638396150
2	868	Women's Mountain Shorts, M	SH-W890-M	Black	26,1763	69,99	M	NULL	26	37	2003-07-01 00:00:00.000	NULL	NULL	0x47494638396150
3	865	Classic Vest, M	VE-C304-M	Blue	23,749	63,50	M	NULL	29	1	2003-07-01 00:00:00.000	NULL	NULL	0x47494638396150
4	849	Men's Sports Shorts, M	SH-M897-M	Black	24,7459	59,99	M	NULL	26	13	2002-07-01 00:00:00.000	2003-06-30 00:00:00.000	NULL	0x47494638396150
5	851	Men's Sports Shorts, XL	SH-M897-X	Black	24,7459	59,99	XL	NULL	26	13	2002-07-01 00:00:00.000	2003-06-30 00:00:00.000	NULL	0x47494638396150
6	882	Short-Sleeve Classic Jersey, M	SJ-0194-M	Yellow	41,5723	53,99	M	NULL	25	32	2003-07-01 00:00:00.000	NULL	NULL	0x4749463839614E
7	884	Short-Sleeve Classic Jersey, XL	SJ-0194-X	Yellow	41,5723	53,99	XL	NULL	25	32	2003-07-01 00:00:00.000	NULL	NULL	0x4749463839614E
8	862	Full-Finger Gloves, M	GL-F110-M	Black	15,6709	37,99	M	NULL	24	3	2002-07-01 00:00:00.000	2003-06-30 00:00:00.000	NULL	0x47494638396150
9	859	Half-Finger Gloves, M	GL-H102-M	Black	9,1593	24,49	M	NULL	24	4	2002-07-01 00:00:00.000	NULL	NULL	0x47494638396150