Database Project

Pharmacy

This database is designed to track and manage various economic aspects of the pharmacy's operations, including customer information, medication inventory, employee information, and financial transactions. Let’s take a loot at some of the more important tables.

The Medication table stores detailed information about the pharmacy's medications, including the name, dosage, and instructions for use. This information is used to accurately fill prescriptions and track medication inventory.

The Prescriptions table stores information about prescriptions written by doctors and filled by the pharmacy, including the patient's name and medical history, the medication prescribed, and the dosage. This information is used to track the prescriptions filled by the pharmacy and ensure that patients receive the correct medication.

The pharmacy has a coupon system in place to offer discounts to customers. The Coupons table stores information about the available coupons, including the discount amount or expiration dates. The Promotions table stores information about current promotions, including the products or services being promoted and any applicable discounts.

The Employees and Departments tables store information about the pharmacy's employees and departments, respectively. This information is used for employee management and to track the performance of different departments.

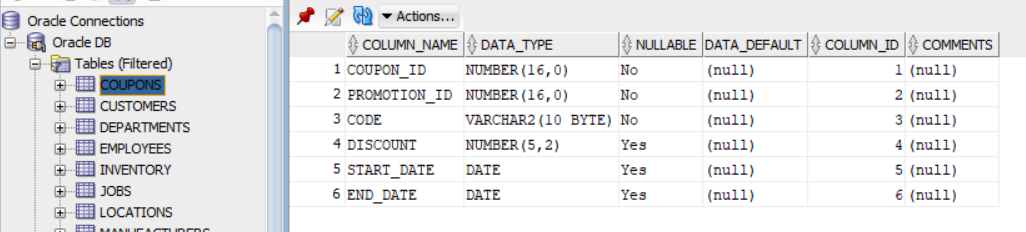
The Payments and Orders tables store information about payment methods and orders placed by customers, respectively. This information is used to process payments and track customer orders. The Transactions table stores information about completed transactions, including the payment method and the total cost of the transaction.

Overall, the pharmacy database is an important tool for managing the economic aspects of the pharmacy's operations and ensuring the smooth and efficient running of the business.

Let’s first take a loot at the database as a whole, after which we can see how all of this was accomplished.

Constructing the database: How it all came to be

When I first created the table most of it was done in a hurry, and I didn’t think all the data through, of course the tables that I ended up with were very superficial, let me show some examples on how those tables would have looked like:



The “coupons” table was done poorly, I created it using this command:

CREATE TABLE Coupons (

coupon\_id number(16) PRIMARY KEY,

promotion\_id number(16) NOT NULL,

code varchar2(10) NOT NULL,

FOREIGN KEY (promotion\_id) REFERENCES Promotions(promotion\_id)

);

After realizing that my first table was a bit too slim I decided on adding a few more important things to it.

ALTER TABLE Coupons

ADD (discount NUMBER(5,2),

start\_date date,

end\_date date);

I also created a few more tables like “transactions” and modified the default “departments” table adding a few more things like, the budget, number of employees and contact info

CREATE TABLE Transactions (

transaction\_id number(16) PRIMARY KEY,

employee\_id number(16) NOT NULL,

patient\_id number(16) NOT NULL,

prescription\_id number(16) NOT NULL,

date\_of\_transaction date NOT NULL,

FOREIGN KEY (employee\_id) REFERENCES Employee(employee\_id),

FOREIGN KEY (patient\_id) REFERENCES Patient(patient\_id),

FOREIGN KEY (prescription\_id) REFERENCES Prescription(prescription\_id)

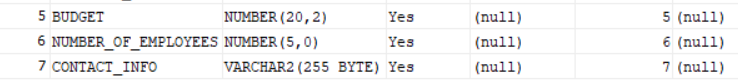
);

ALTER TABLE Departments

ADD (budget NUMBER(20,2),

number\_of\_employees NUMBER(5),

contact\_info VARCHAR2(255));



After all of that I realized that one of my tables became obsolete, since it was mostly a duplicate of the “Transactions” table so I decided to drop it using the “Drop table” command, unfortunately, I forgot the name of the table, but I remember that it was something related to the orders, its only purpose was being a “link table” in order to avoid a many-to-many relationship between the “payments” and “orders” tables, something that the new “transactions” table does much better.

Dropping the table was easily accomplished by:

DROP TABLE name\_of\_the\_table\_that\_i\_forgot;

I also created other tables without much problems and added things that I may have forgotten, like some foreign keys:

CREATE TABLE Patient (

patient\_id number(16) PRIMARY KEY,

first\_name varchar2(255) NOT NULL,

last\_name varchar2(255) NOT NULL,

address varchar2(255) NOT NULL,

date\_of\_birth date NOT NULL,

medical\_history TEXT

);

CREATE TABLE Prescription (

prescription\_id number(16) PRIMARY KEY,

patient\_id number(16) NOT NULL,

medication\_id number(16) NOT NULL,

dosage varchar2(255) NOT NULL,

frequency varchar2(255) NOT NULL,

FOREIGN KEY (patient\_id) REFERENCES Patient(patient\_id),

FOREIGN KEY (medication\_id) REFERENCES Medication(medication\_id)

);

ALTER TABLE Jobs

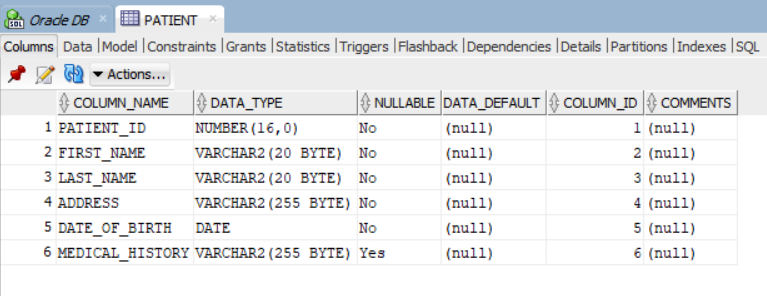
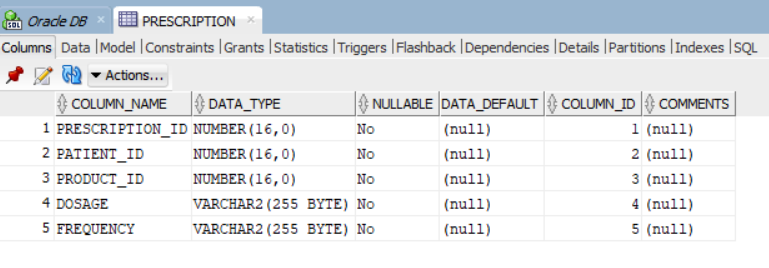
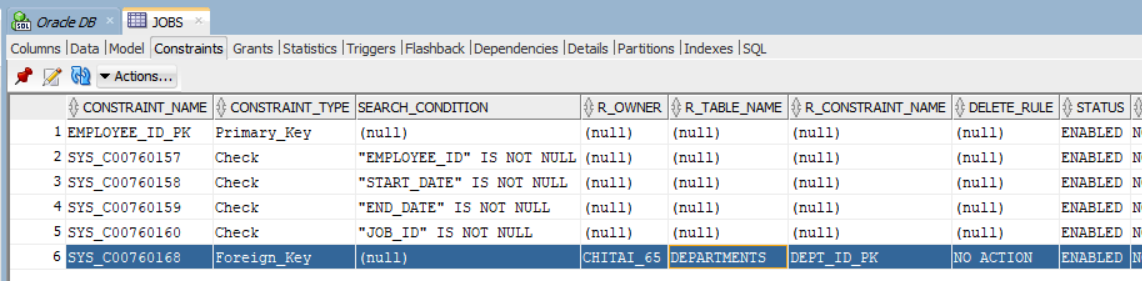
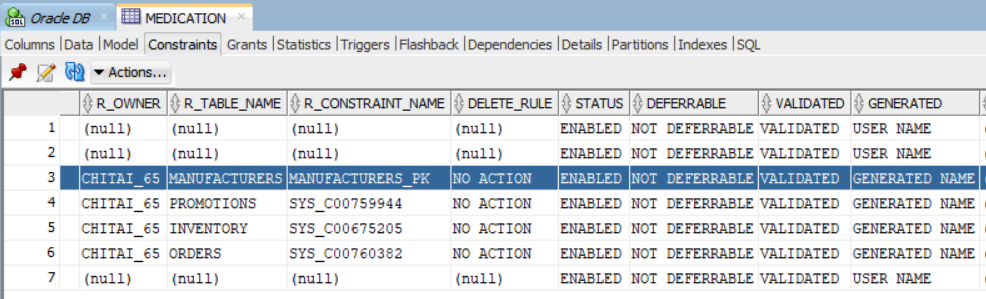
ADD FOREIGN KEY (department\_id) REFERENCES Departments(department\_id);

ALTER TABLE Medication

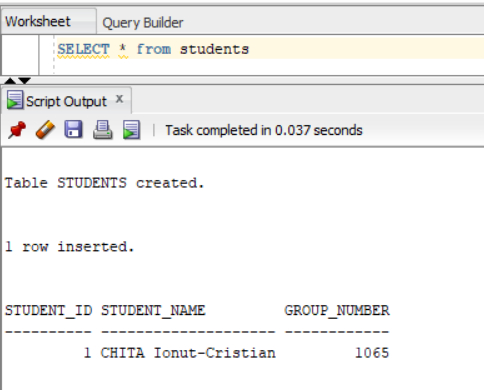
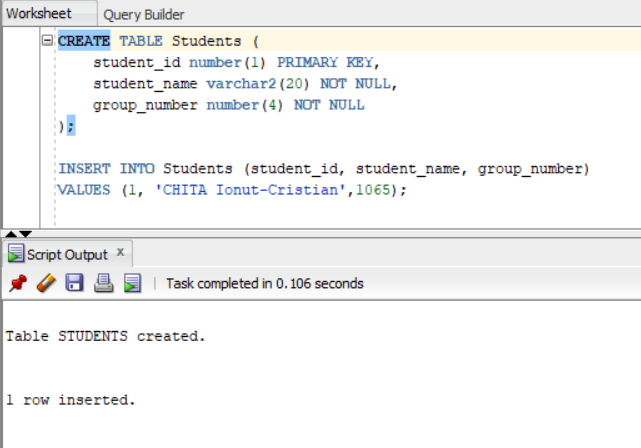
ADD (manufacturer\_id number(16),

FOREIGN KEY (manufacturer\_id) REFERENCES Manufacturers(manufacturer\_id));

All of these commands worked flawlessly, but I unfortunately forgot to take screenshots when I first ran them, running them now on my database will most likely cause problems, but since all of the modifications are present in the database schema there’s no doubt that they work.

Using DML Elements: Now that we created most of the tables, we also have to put it to use, first I’ll create a table called Students, in which I’ll insert my name using this command:



It worked! Let’s play around some more, next let’s insert some data into the “Patients” table:

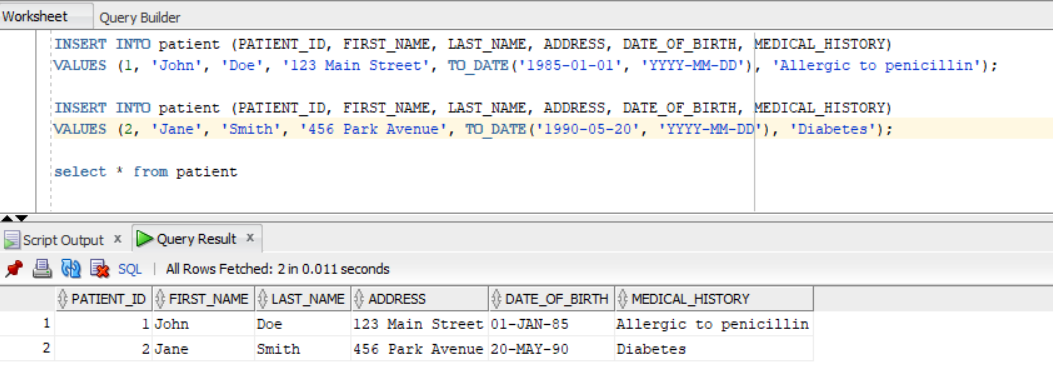
INSERT INTO patients (PATIENT\_ID, FIRST\_NAME, LAST\_NAME, ADDRESS, DATE\_OF\_BIRTH, MEDICAL\_HISTORY)

VALUES (1, 'John', 'Doe', '123 Main Street', TO\_DATE('1985-01-01', 'YYYY-MM-DD'), 'Allergic to penicillin');

INSERT INTO patients (PATIENT\_ID, FIRST\_NAME, LAST\_NAME, ADDRESS, DATE\_OF\_BIRTH, MEDICAL\_HISTORY)

VALUES (2, 'Jane', 'Smith', '456 Park Avenue', TO\_DATE('1990-05-20', 'YYYY-MM-DD'), 'Diabetes');

Note: I’m using the “to date” function because in Oracle, the default date format is DD-MON-YY. This let’s me change it to whatever date format I wish.

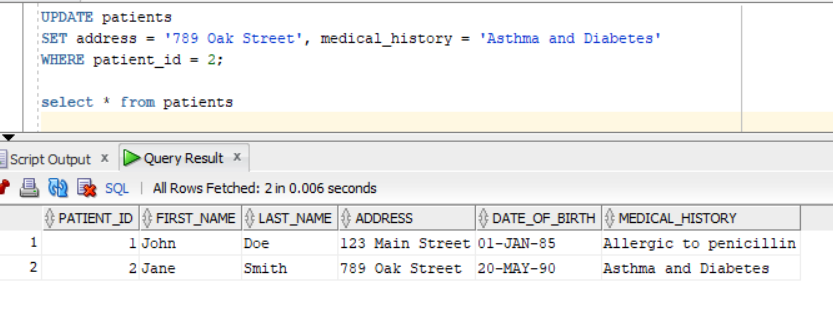


Now, let’s say that one of our patients moved out and discovered that they have other ailments too, let’s change that too, we’ll use the UPDATE statement:

UPDATE patients

SET address = '789 Oak Street', medical\_history = 'Asthma and Diabetes'

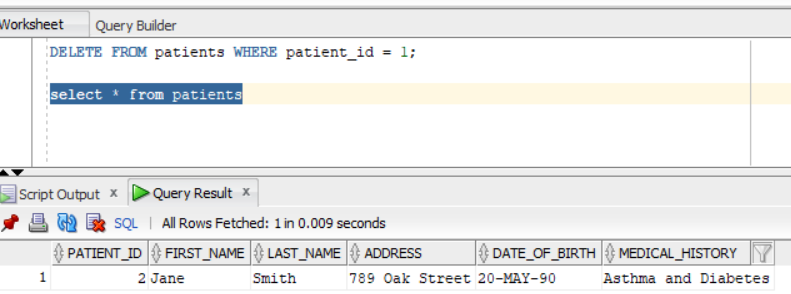
WHERE patient\_id = 2;



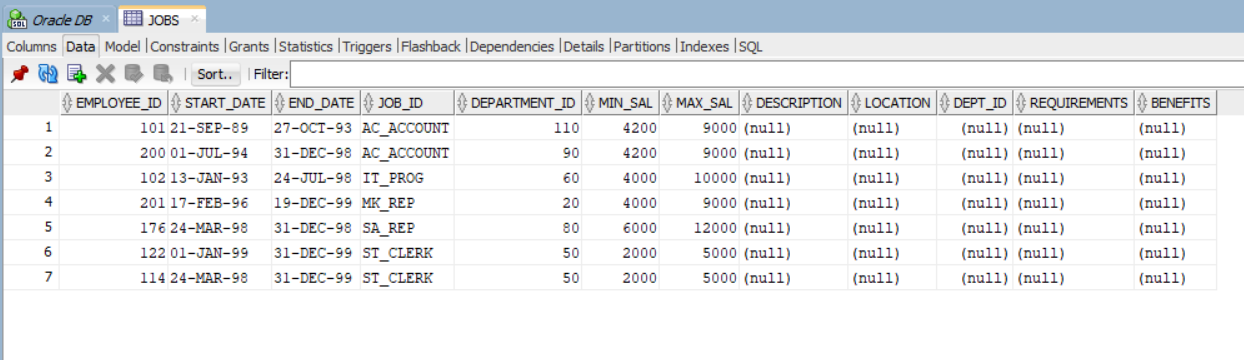
Let’s say that one of our patients chose another pharmacy (very unlikely), how would we delete their records ? For this we have the “DELETE” statement, let’s try it:

DELETE FROM patients WHERE patient\_id = 1;

This will delete the patient with the id = 1, in our case, Mr. John Doe.



We have quite a lot of data on the employees by default, let’s play around with that shall we ? We also have quite a few jobs by default, all jobs have a minimum salary and a maximum salary, each job has it’s own id, let’s give them all a raise using the “MERGE” statement



In order to do that we’ll have to merge the salaries into the employees table, we can accomplish that by using:

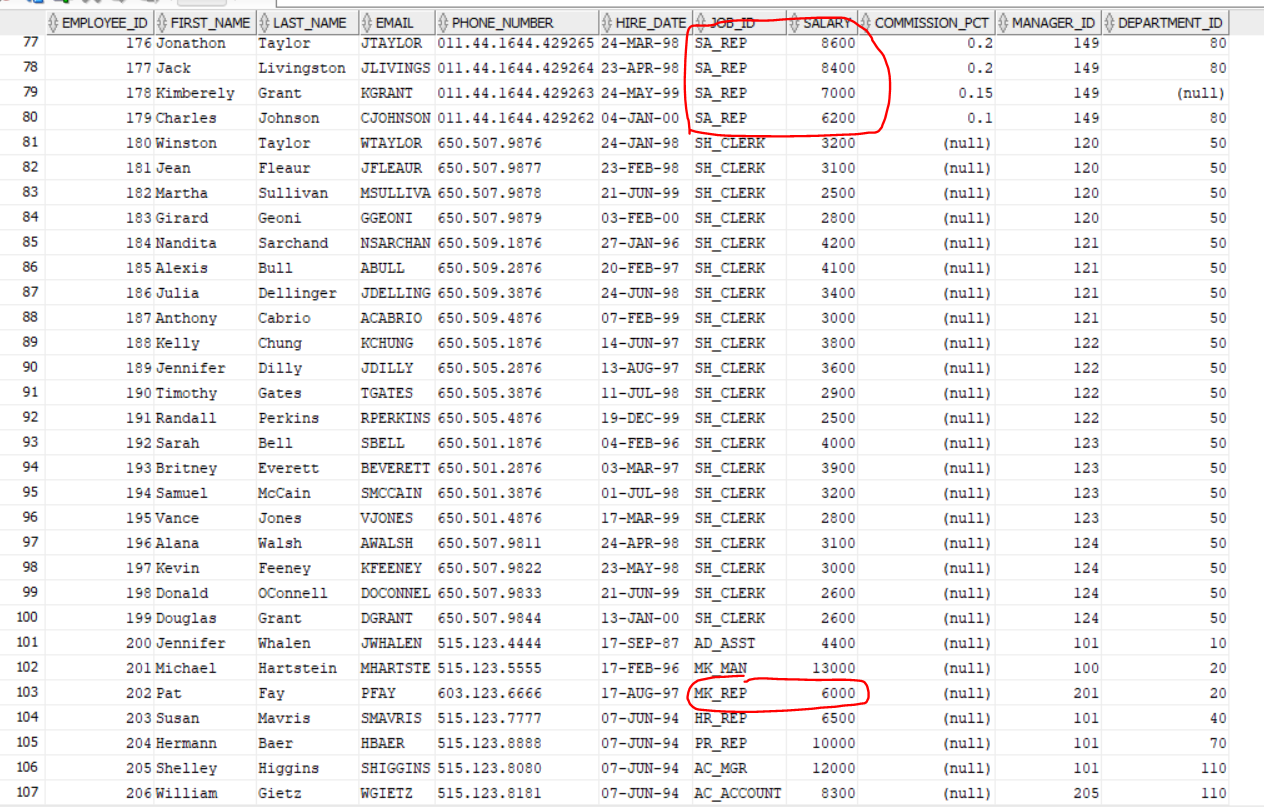
MERGE INTO employees e

USING jobs j

ON (e.job\_id = j.job\_id)

WHEN MATCHED THEN

UPDATE SET e.salary = j.max\_salary

This will update the salary of all the rows in the employees table with the corresponding max\_salary value from the jobs table. Before running it, let’s first take a look at our employees table and look at their salaries: 

These are some of the lucky guys that will get a raise, Lets run it and see what happens



Looks like it didn’t work, this happens because the MERGE statement returns multiple rows for a single row in the target table. To fix this error, we need to ensure that we have a USING clause that returns at most one row for each row in the target table

MERGE INTO employees e

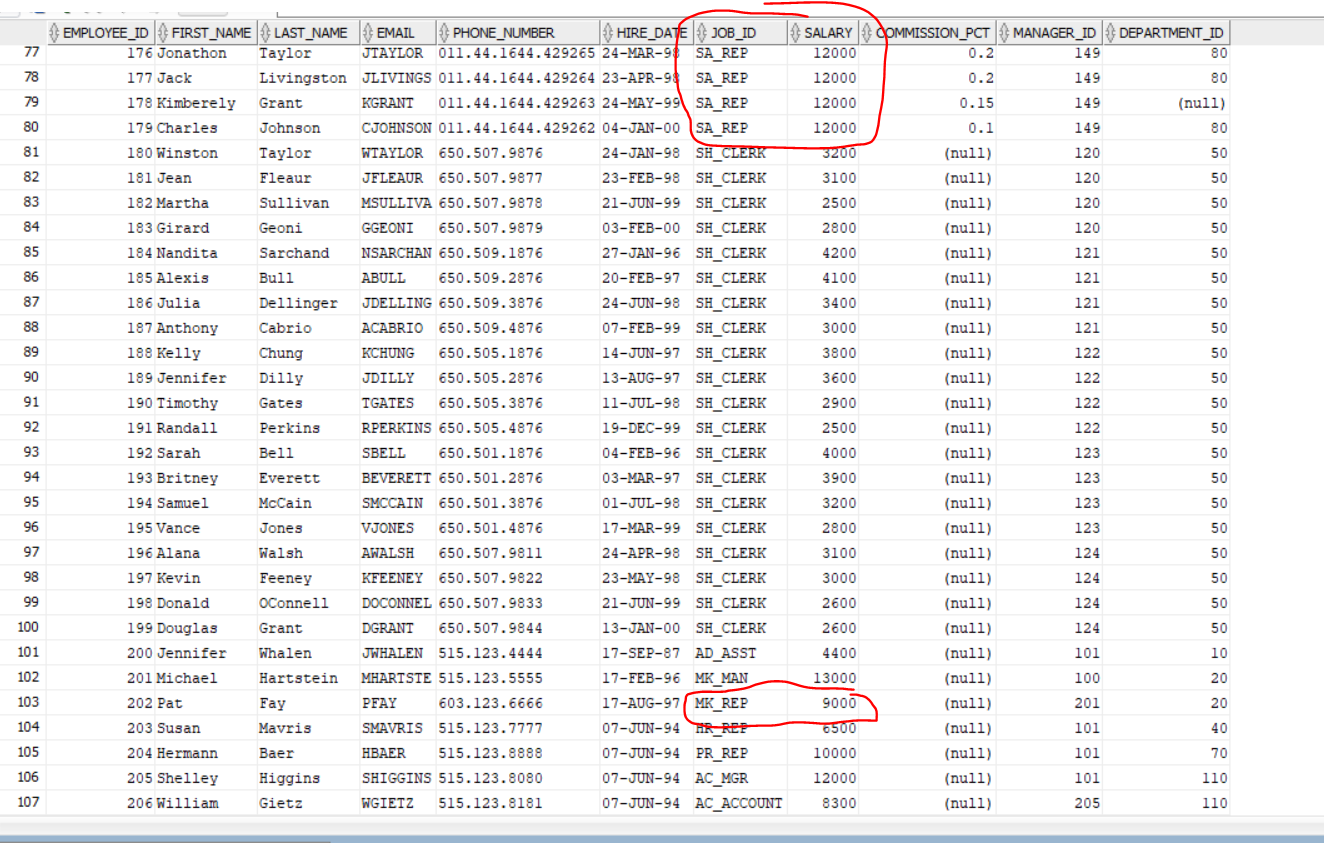
USING (SELECT j.max\_salary FROM jobs j WHERE j.job\_id = e.job\_id) s

ON (1=1)

WHEN MATCHED THEN

UPDATE SET e.salary = s.max\_salary

As you can see, this time around it worked:



Now, let’s also get some more patients, in order to get a bunch of people fast I’ll get the data from the “customers” table and insert it into the “patients” table using this command:

INSERT INTO Patients (PATIENT\_ID, FIRST\_NAME, LAST\_NAME, ADDRESS, DATE\_OF\_BIRTH, MEDICAL\_HISTORY)

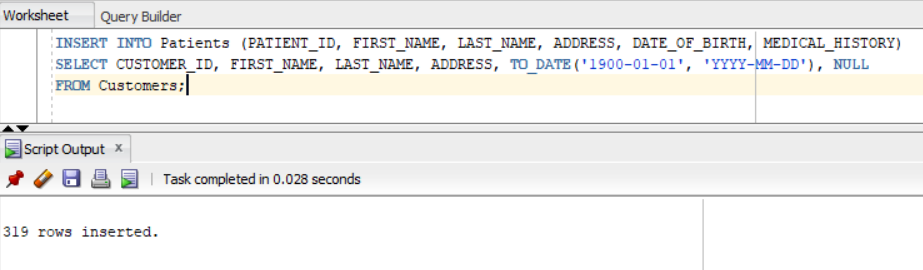
SELECT CUSTOMER\_ID, FIRST\_NAME, LAST\_NAME, ADDRESS, TO\_DATE('1900-01-01', 'YYYY-MM-DD'), NULL

FROM Customers;

Note: the reason we use the “to date” function this time is because we first created the “Patients” table we set the date of birth as not null, which means that we have to provide a value, to circumvent that we can either use the “to date” function to set a default value that we can change later on. Alternatively, we can alter the table using:

ALTER TABLE Patients MODIFY DATE\_OF\_BIRTH DATE NULL;

This removes the constraint and lets us input null data



As you can see, it was a success, now we have some more data to work with. Let’s also insert some products into our inventory:

INSERT INTO Inventory (INVENTORY\_ID, PRODUCT\_ID, NAME, QUANTITY, COST, PRICE)

VALUES (2, 1002, 'Ibuprofen', 1000, 0.75, 1.50);

INSERT INTO Inventory (INVENTORY\_ID, PRODUCT\_ID, NAME, QUANTITY, COST, PRICE)

VALUES (3, 1003, 'Aspirin', 1000, 0.65, 1.30);

INSERT INTO Inventory (INVENTORY\_ID, PRODUCT\_ID, NAME, QUANTITY, COST, PRICE)

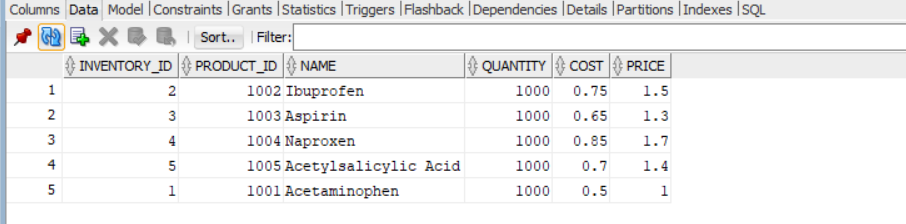
VALUES (4, 1004, 'Naproxen', 1000, 0.85, 1.70);

INSERT INTO Inventory (INVENTORY\_ID, PRODUCT\_ID, NAME, QUANTITY, COST, PRICE)

VALUES (5, 1005, 'Acetylsalicylic Acid', 1000, 0.70, 1.40);

INSERT INTO Inventory (INVENTORY\_ID, PRODUCT\_ID, NAME, QUANTITY, COST, PRICE)

VALUES (1, 1001, 'Acetaminophen', 1000, 0.50, 1.00);



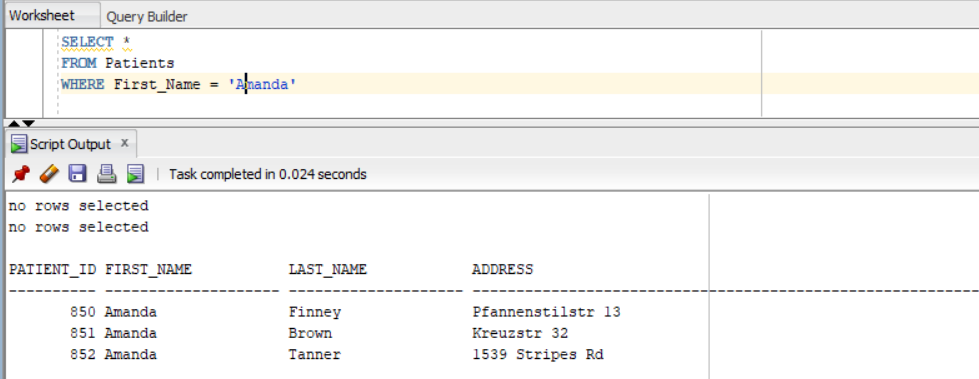
Now that we have all of this data, let’s play around with the SELECT statement:

1. Let’s first Select all of the patients whose first name is “Amanda”

SELECT \*

FROM Patients

WHERE First\_Name = 'Amanda'

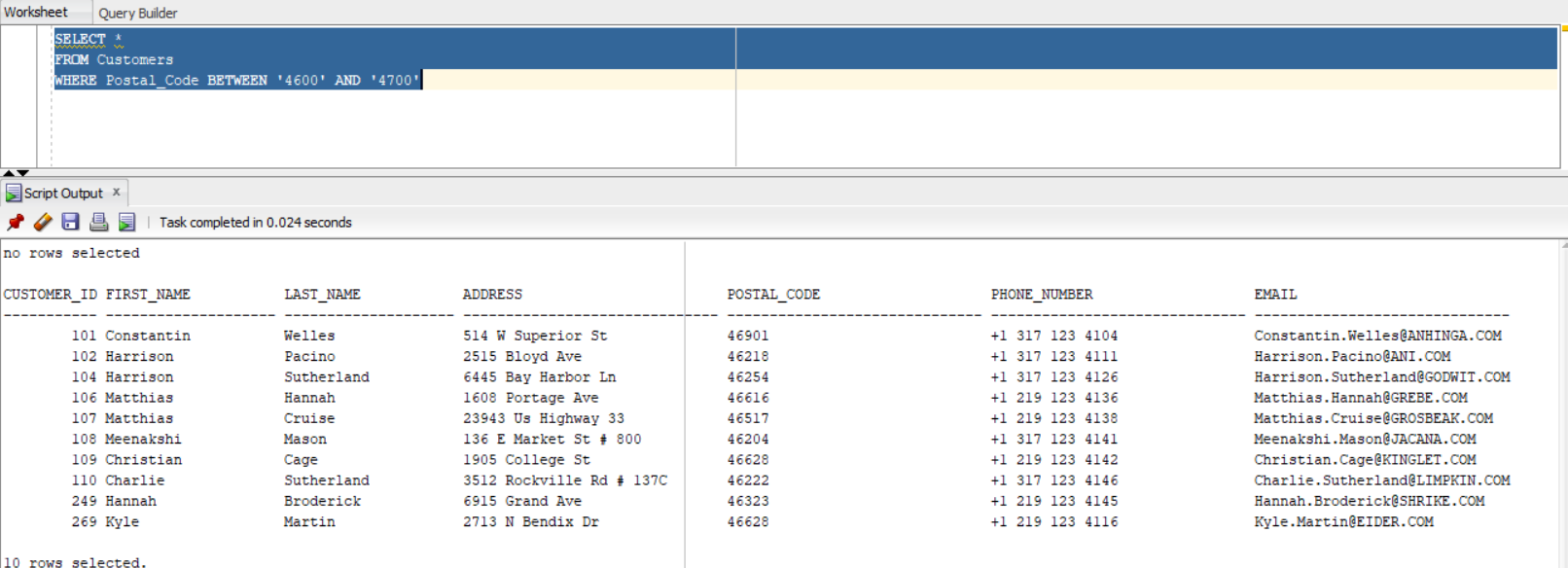


1. What about selecting customers based by their postal code, let’s select customers with postal code between 4600 and 4700

SELECT \*

FROM Customers

WHERE Postal\_Code BETWEEN '4600' AND '4700'

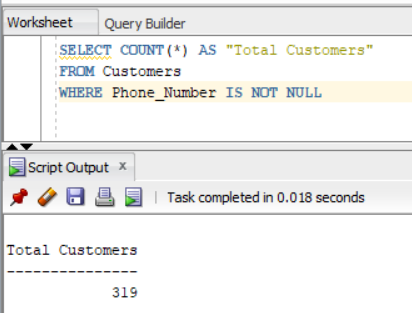
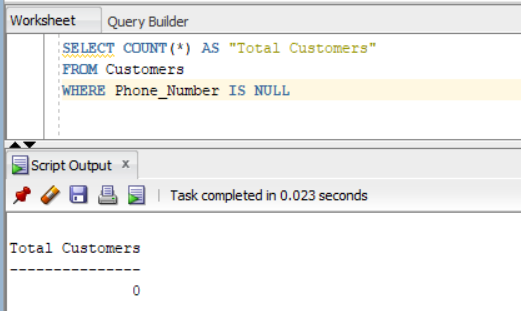


1. Let’s also count how many of our customers own a phone / phone number

SELECT COUNT(\*) AS "Total Customers"

FROM Customers

WHERE Phone\_Number IS NOT NULL

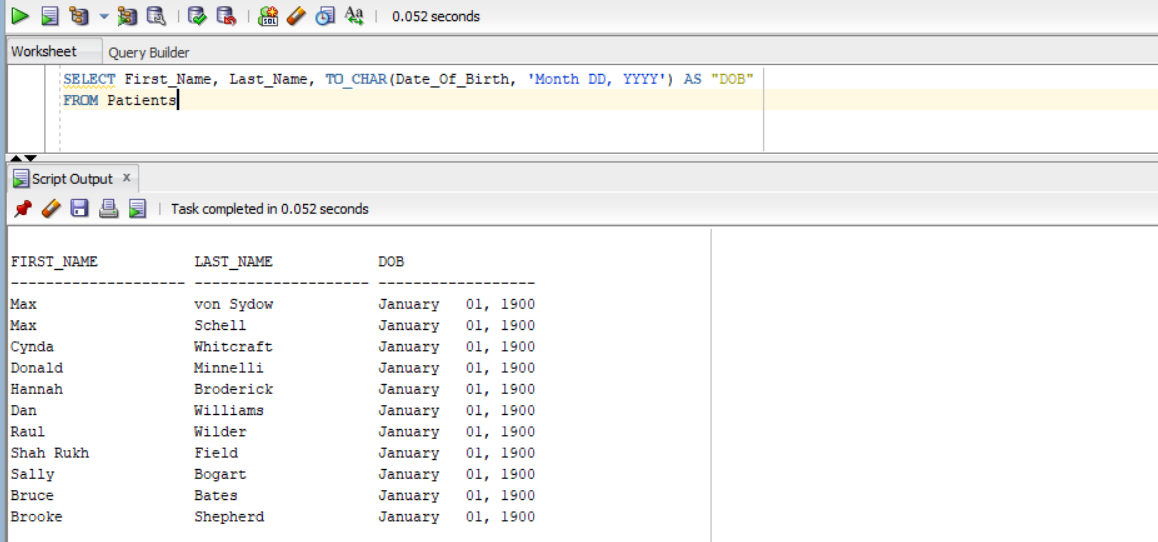
 

Looks like all of them own a phone, we can see the opposite by slightly altering the condition, replacing IS NOT NULL with IS NULL.

1. What about using the TO\_CHAR function, it is used in order to convert a number or a date into a string, let’s extract our patients’ date of birth and convert it into a string

SELECT First\_Name, Last\_Name, TO\_CHAR(Date\_Of\_Birth, 'Month DD, YYYY') AS "DOB"

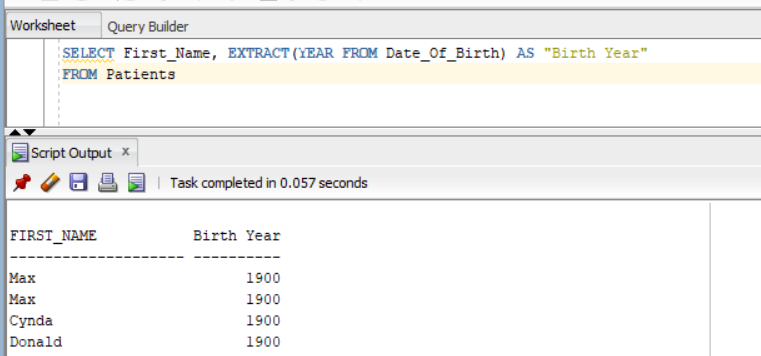
FROM Patients



1. But what if we only care about the year ? We can use the EXTRACT function:

SELECT First\_Name, EXTRACT(YEAR FROM Date\_Of\_Birth) AS "Birth Year"

FROM Patients



1. Let’s do something more daring, shall we ? Let’s see whose medical history we have on file and whose we don’t. We can do that by using this SELECT statement:

SELECT p.First\_Name, p.Last\_Name,

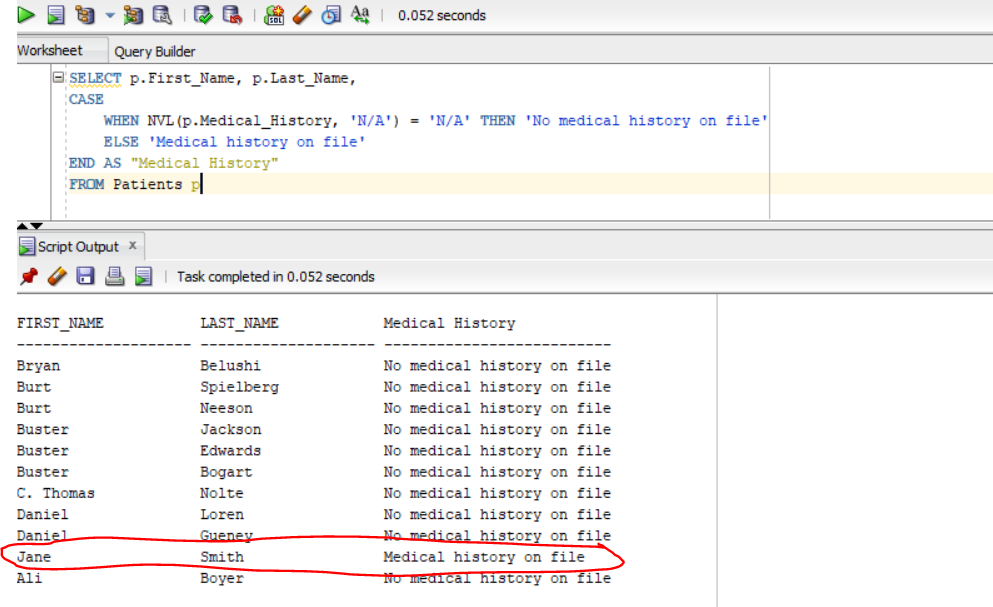
CASE

WHEN NVL(p.Medical\_History, 'N/A') = 'N/A' THEN 'No medical history on file'

ELSE 'Medical history on file'

END AS "Medical History"

FROM Patients p

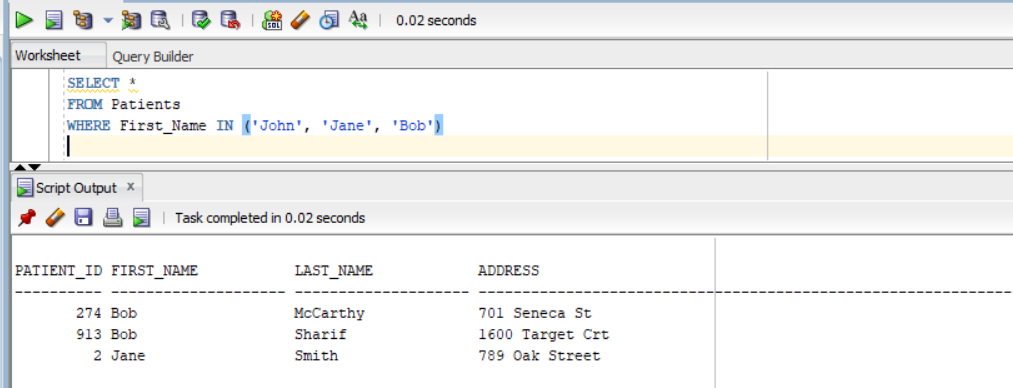


1. Let’s try some more functions:

SELECT \*

FROM Patients

WHERE First\_Name IN ('John', 'Jane', 'Bob')

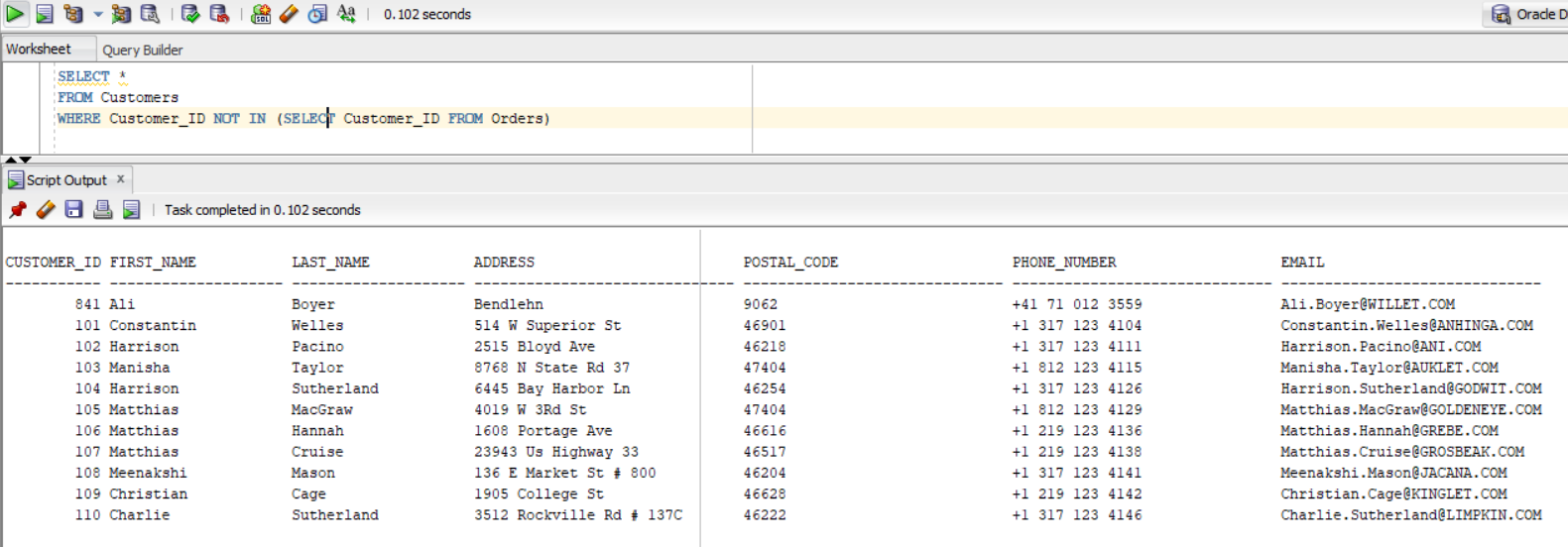


1. We can do the opposite of IN by using NOT IN

SELECT \*

FROM Customers

WHERE Customer\_ID NOT IN (SELECT Customer\_ID FROM Orders)



This selects everything from customers whose id doesn’t match the ones from the Orders table

1. A bit earlier we added all of our customers to the patients table, but before that, we added jane, let’s see if we can find her using the MINUS function

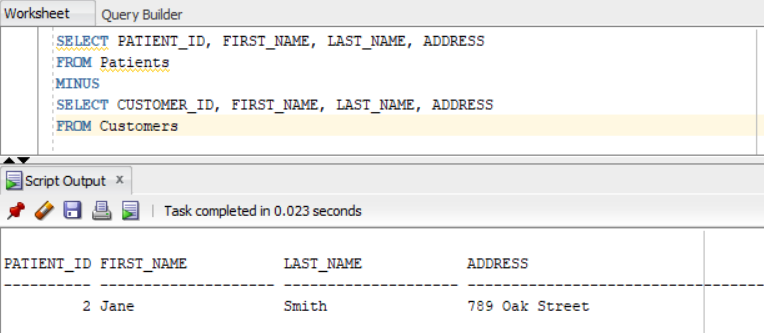
SELECT PATIENT\_ID, FIRST\_NAME, LAST\_NAME, ADDRESS

FROM Patients

MINUS

SELECT CUSTOMER\_ID, FIRST\_NAME, LAST\_NAME, ADDRESS

FROM Customers



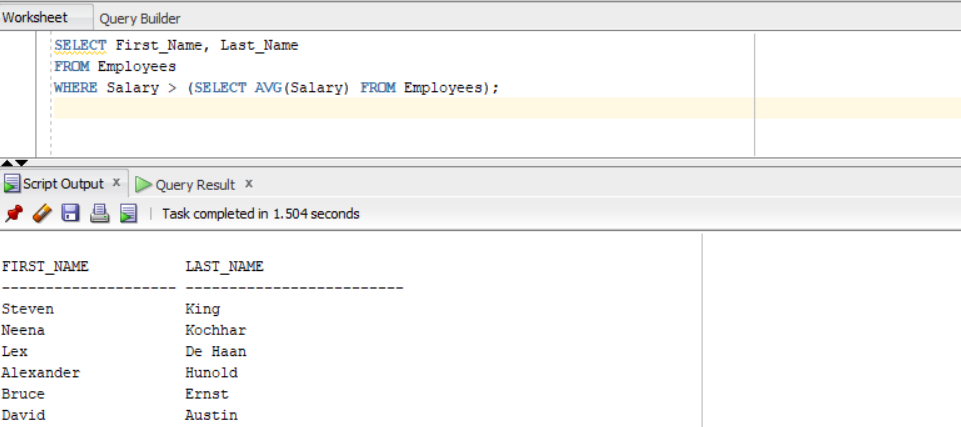
And here she is! Note: We didn’t select the whole tables, that’s because they have a different number of columns which will cause error ORA-01789 to occur, to fix this we can simply select only the columns that we need.

1. Some simple and correlated subqueries: Simple:

SELECT First\_Name, Last\_Name

FROM Employees

WHERE Salary > (SELECT AVG(Salary) FROM Employees);

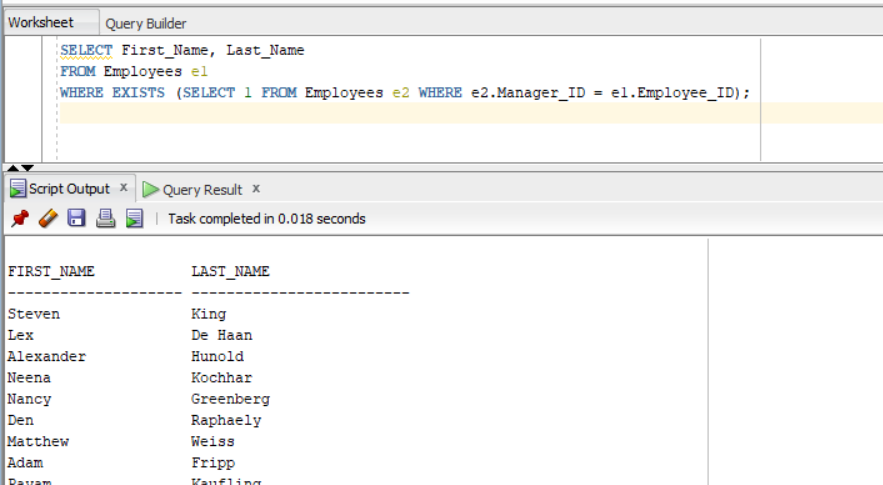


1. Correlated

SELECT First\_Name, Last\_Name

FROM Employees e1

WHERE EXISTS (SELECT 1 FROM Employees e2 WHERE e2.Manager\_ID = e1.Employee\_ID);

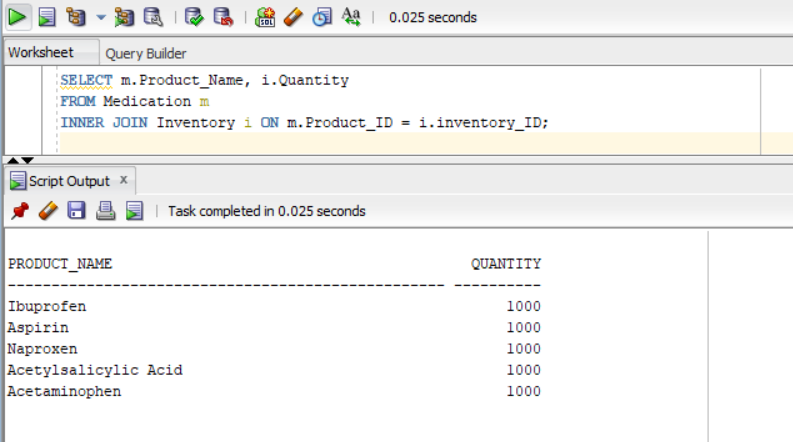


1. Let’s also see some inner and outer joints. I’ll use an inner join to show what products I have in my inventory and how many:

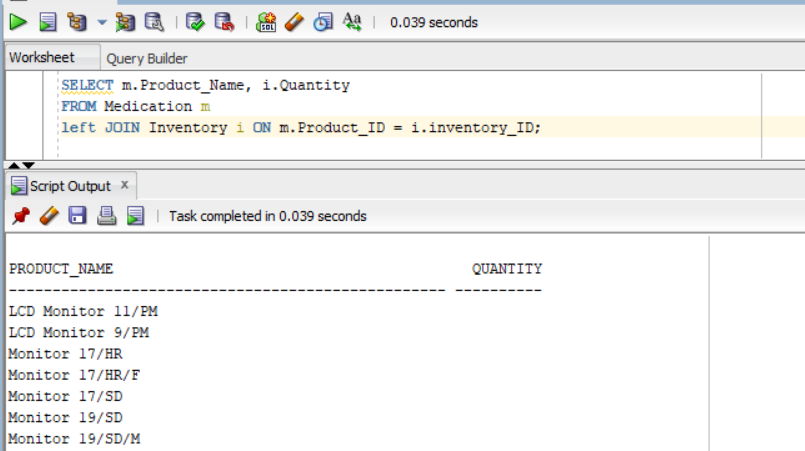
SELECT m.Product\_Name, i.Quantity

FROM Medication m

INNER JOIN Inventory i ON m.Product\_ID = i.inventory\_ID;

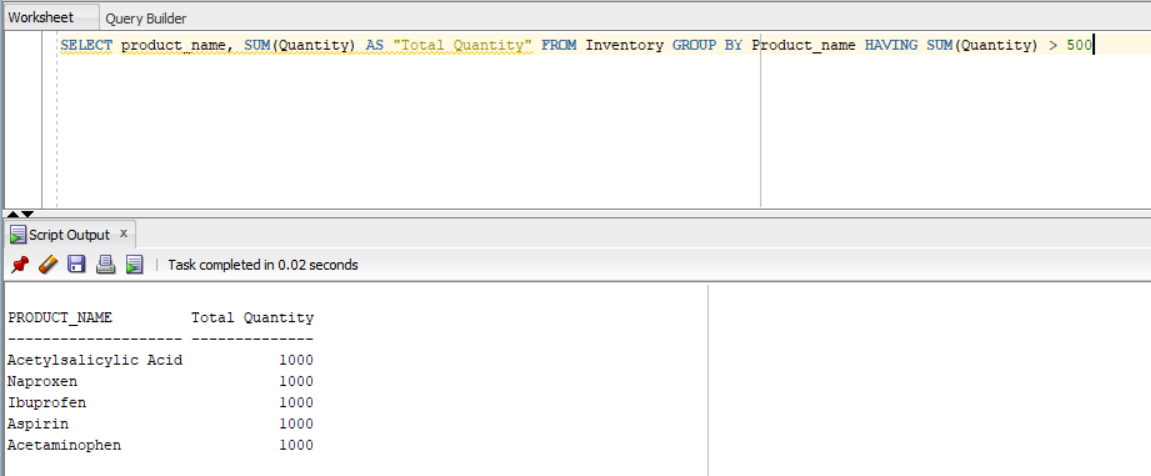


1. Let’s see some outer joints too:



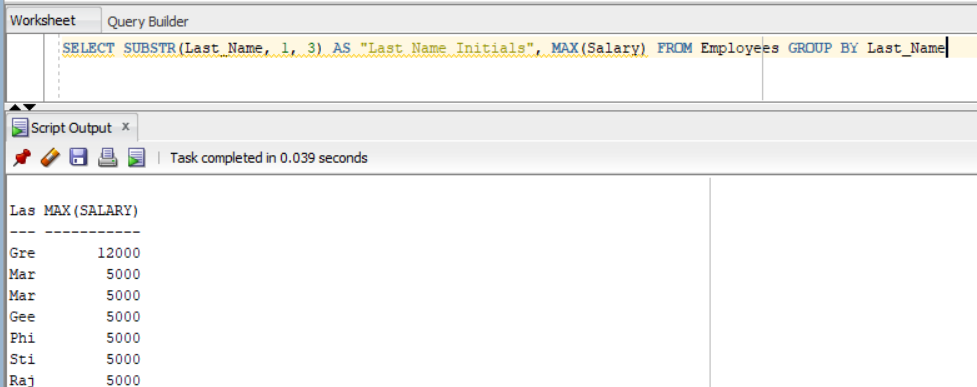
1. Using aggregate function HAVING and GROUP BY

SELECT Product\_Name, SUM(Quantity) AS "Total Quantity" FROM Inventory GROUP BY Product\_Name HAVING SUM(Quantity) > 500

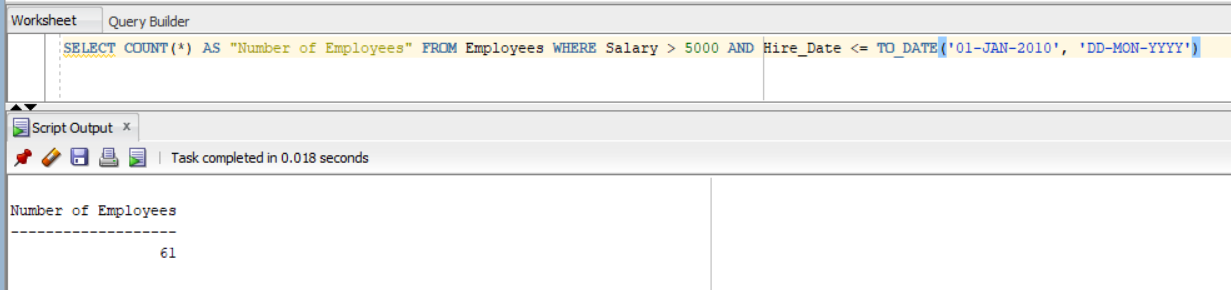


1. SUBSTR function

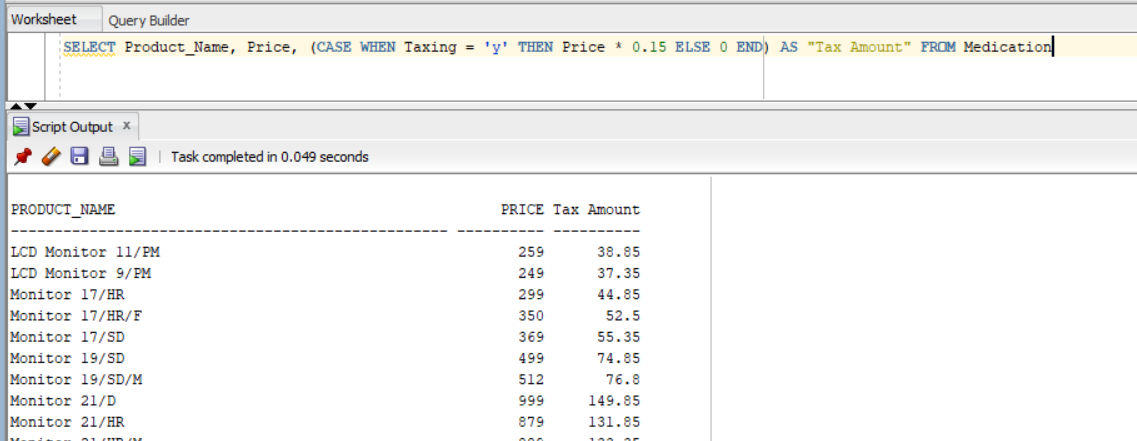
SELECT SUBSTR(Last\_Name, 1, 3) AS "Last Name Initials", MAX(Salary) FROM Employees GROUP BY Last\_Name



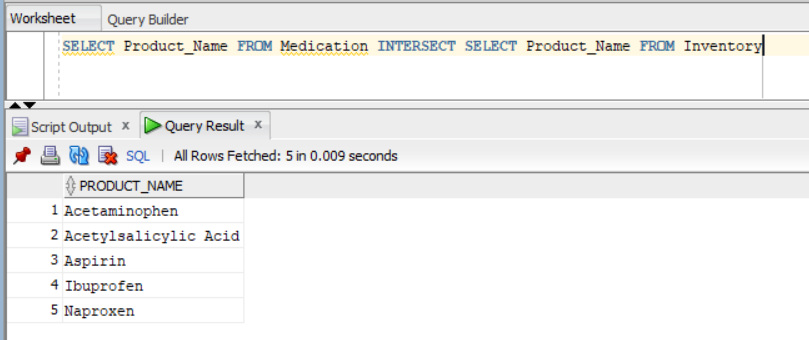
1. Let’s see how many employees have a salary over 5000 and who were hired before 2010:



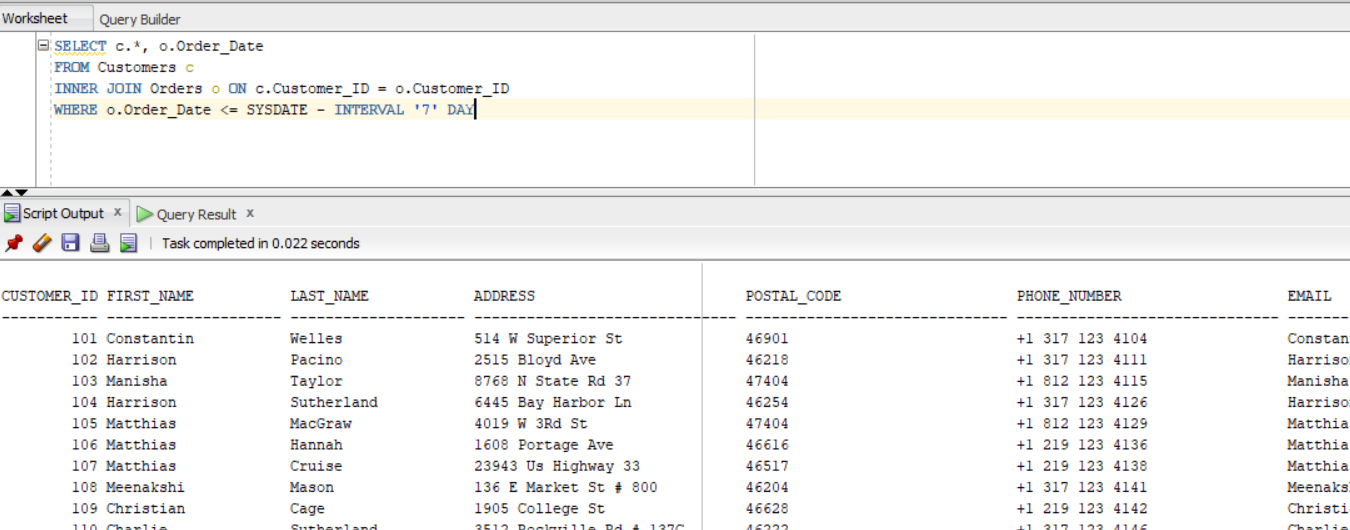
1. What about some taxes? This SELECT statement will compute the tax amount based on the price of the product



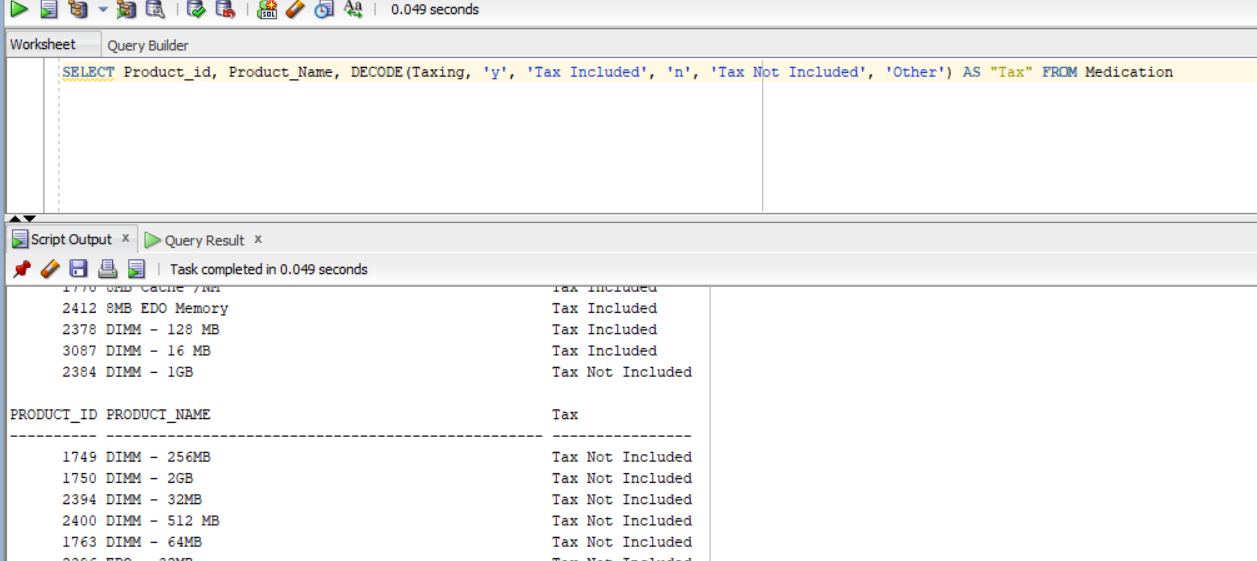
1. Previously we used both a joint and a minus in order to single out some products, let’s try using intersect and see how it goes:



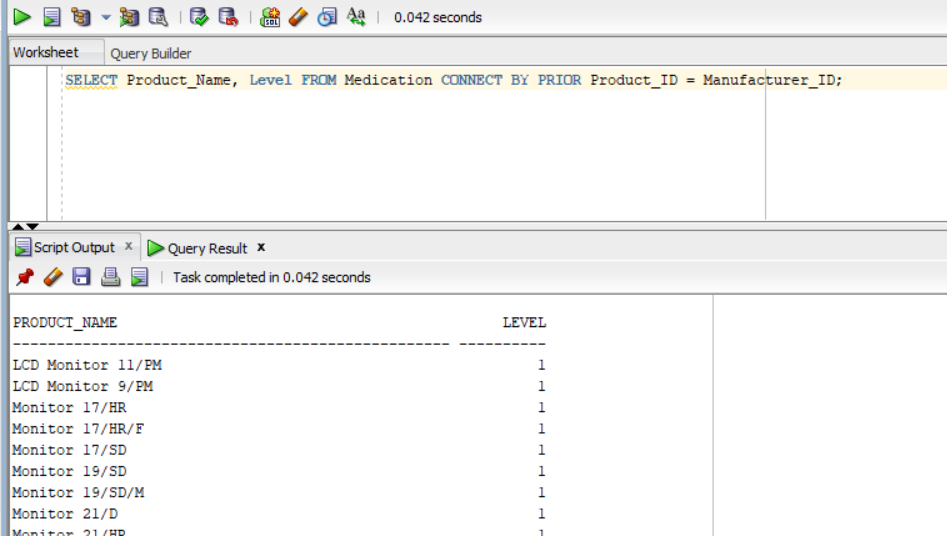
1. Let’s also use SYSDATE and an INNER JOIN to show orders that happened in the past



1. Another useful thing to use is DECODE, as the name implies it is used to decode my binary y/n system into something more human readable



1. Hierarchical queries:

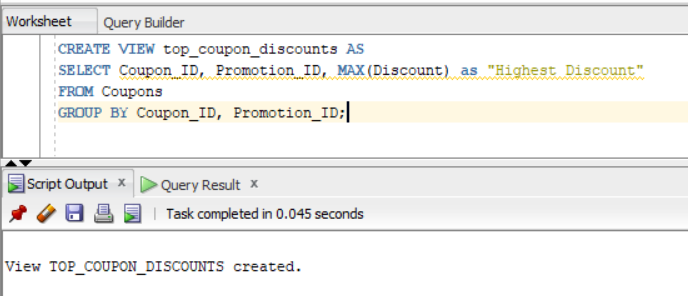


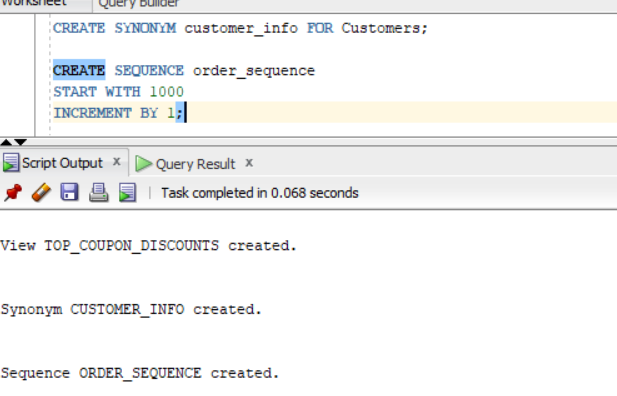
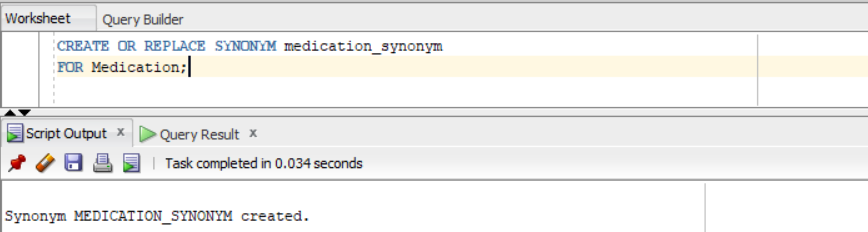
This query retrieves the Product\_Name and Level values from the Medication table, and the PRIOR operator is used to specify that the Manufacturer\_ID column in the Medication table should be used to connect rows in the hierarchy.

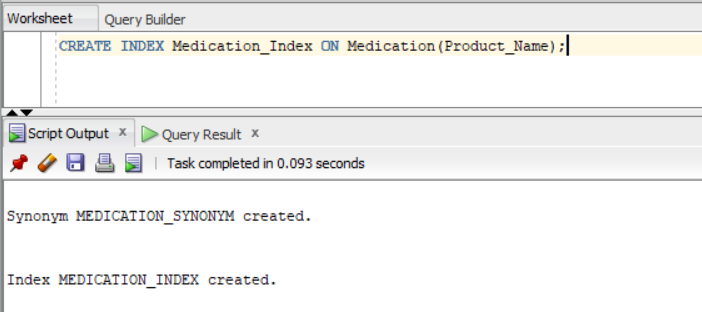
The LEVEL pseudocolumn is used to display the level of each row in the hierarchy.

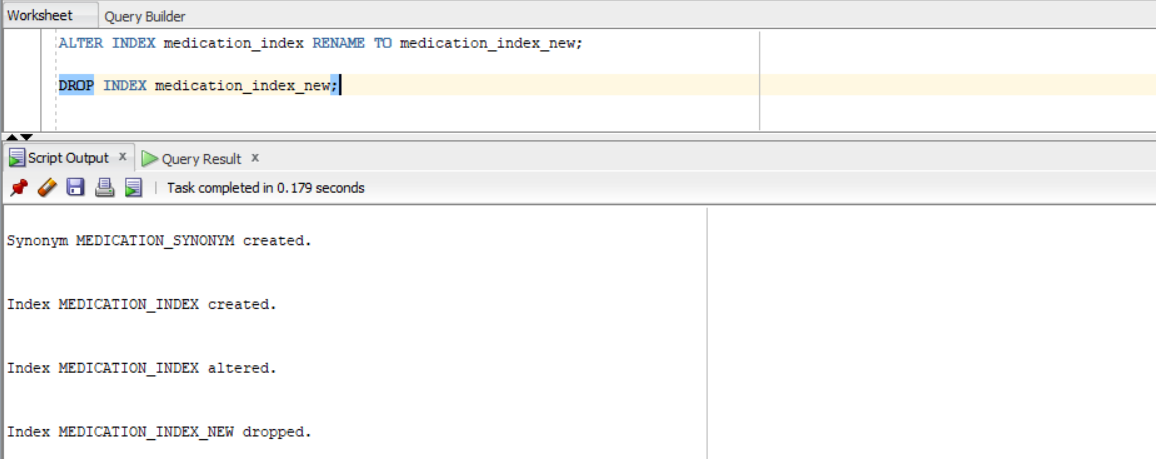
The query returns a result set with a row for each product in the Medication table, and the rows are organized into a hierarchy based on the Manufacturer\_ID values. The Level column shows the level of each product in the hierarchy.

1. Some examples of constructing and manipulating other objects of the database: views, indexes, synonyms and sequences:

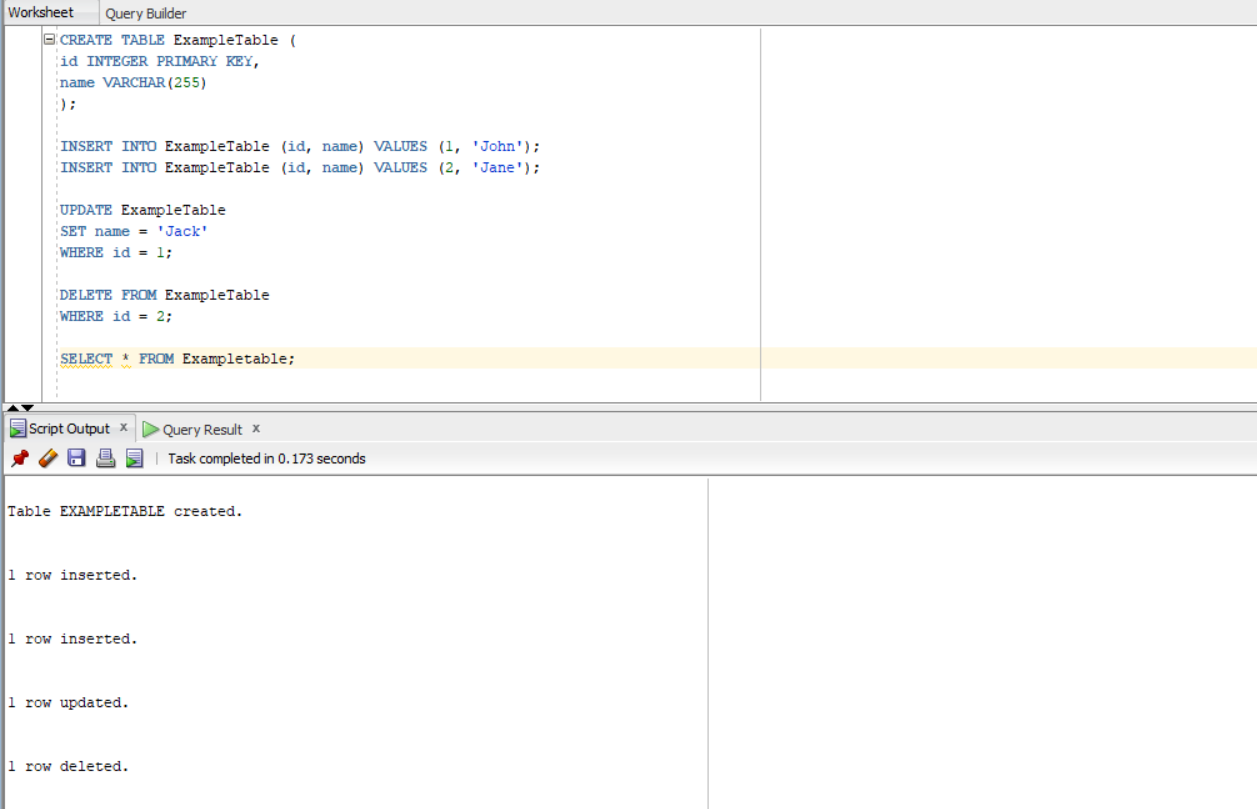








1. Create/Insert/Update/Delete + Select





1. Let’s also use SYS\_CONNECT\_BY\_PATH :



This SELECT statement retrieves the Product\_Name column values from the Medication table and creates a new column called Hierarchy. The SYS\_CONNECT\_BY\_PATH function is used to generate a string of Product\_Name values, separated by the | character, for each row in the Medication table. The CONNECT BY PRIOR clause is used to indicate that the Product\_Name values should be connected in a hierarchy, with the Product\_ID column serving as the parent key and the Manufacturer\_ID column serving as the child key. The resulting hierarchy will show the relationships between the different products in the Medication table, with the Product\_Name values of the parent products appearing before the Product\_Name values of the child products.

In conclusion, our pharmacy database has proven to be a valuable resource for managing and organizing all aspects of our business. With tables for customers, employees, medication, orders, and more, we have been able to efficiently track and analyze data in order to make informed decisions and improve operations. The use of foreign keys and relationships has ensured the integrity of our data, and the implementation of views, indexes, synonyms, and sequences will allow us to easily access and manipulate the information in our database. Overall, the development and utilization of this database has greatly benefited our pharmacy and we look forward to continuing to use it in the future.