Data Science

Contents

1 PostgreSQL For Data Science

1.1 SQL Query Basics (Filtering)

The SQL language is case insensitive, but the data within the cells is case sensitive.
However, the industry standard is to capitalize the SQL keywords

```
select * from employees;
SELECT * FROM employees; -- Industry Standard
-- These two statements are the same and will produce same output
```

SQL Keywords:

· SELECT - this will specify which column/attribute you want to see in a query
· FROM - specifies which table you want to select the column/attribute from
· WHERE - a condition that lets you set parameters on a given column
· LIKE - used in addition with WHERE and '% %', lets you search for parts of a word

```
SELECT * FROM employees
WHERE department LIKE 'F%nitu%' -- such as Furniture (case sensitive)
-- word must start with 'F' and contains 'nitu' somwhere in middle
```

We can filter on multiple conditions by using the AND operator or the OR operator
We can also filter with these operators together by using ( )

```
SELECT * FROM employees
WHERE department = 'Clothing' AND salary > 90000 AND region_id = 2
-- all 3 conditions must be met in order to be added to query

SELECT * FROM employees
WHERE department = 'Clothing' OR salary > 150000
-- will add any row that meets either of these conditions to the query

SELECT * FROM employees
WHERE salary < 40000 AND (department = 'Clothing' OR department = 'Pharmacy')
-- will add any row where salary is less than 40k and in either department to query
```

We can use filtering operators to further process our data. Some key operators are:

· NOT - gives us value that are not in the given search ('NOT =' is same as '!=').
· IS NULL - will return all values that are null (use IS NOT NULL to get all non-null values).
· IN - will return all values that are found within the given parameters.
· BETWEEN - will search for values within a given range (inclusive).

```
SELECT * FROM employees
WHERE NOT department = 'Clothing'; -- gives us all departments besides clothing

SELECT * FROM employees
WHERE email IS NOT NULL; -- gives all emails that don't have null value

SELECT * FROM employees
WHERE department IN ('Sports', 'Firs Aid', 'Garden');
-- return values in any of the given departments

SELECT * FROM employees
WHERE salary BETWEEN 80000 AND 100000; -- all salaries in this range
```

We can change the way the information is displayed in the output of our queries with certain keywords:

· ORDER BY - sorts the data by given column, either ascending (ASC) or descending (DESC)

· DISTINCT - will return all unique values for a given column

· LIMIT - will set how many records to show from our query (same as FETCH FIRST _ ROWS ONLY)

· AS - allows us to rename a column to a given parameter name (good for exporting queries)

```
SELECT * FROM employees
ORDER BY employee_id DESC; -- orders from employee_id 1000 down to 1

SELECT DISTINCT department FROM employees -- selects unique departments in table
ORDER BY 1 -- orders by first parameter for select statement
LIMIT 10; -- only show first 10 records

SELECT first_name AS "First Name", salary AS yearly_salary -- use " " when spaces
FROM employees
FETCH FIRST 10 ROWS ONLY; -- only show first 10 records (same as LIMIT 10)
```

1.2 Using Functions

We can use functions with the SELECT statement to alter our data in output queries (not in the table).

· UPPER( ) - converts output to all uppercase

· LOWER( ) - converts output to all lowercase

· LENGTH( ) - gives you the length for a each string in all rows for a given column

· TRIM( ) - will take off any extra space in beginning or end (good for cleaning data)

```
-- all uppercase first name, all lowercase department
SELECT LENGTH(first_name), LOWER(department)
FROM employees;

-- we can test a function without selecting from a table
SELECT LENGTH(TRIM('   HELLO   ')) -- trim extra space, return length = 5
```

We can combine multiple columns together by using concatenation || or a Boolean expression

```
-- combine first name and last name into new column called full_name
SELECT first_name || ' ' || last_name AS full_name
FROM employees;

-- use a boolean to create a new column of True or False
SELECT (salary > 140000) AS highly_paid
FROM employees
ORDER BY salary DESC; -- show all true values first

SELECT department, ('Clothing' IN department) -- creates a new column of T/F
FROM employees;

SELECT department, (department LIKE '%oth%') -- creates a new column of T/F
FROM employees; -- T if department has 'oth' in its name, otherwise F
```

1.2.1 String Functions

We can perform functions on strings in a given table and output them in our query (not in the table).

· SUBSTRING( ) - lets us extract part of a string (FROM start FOR length)
· REPLACE( ) - lets us change the name of strings to a new string
· POSITION( ) - lets us find a position of a given character
· COALESCE( ) - lets us add a value into any null cells for the output

```
-- test the substring function with a given string
SELECT SUBSTRING('This is test data' FROM 1 FOR 4) AS test_data; -- returns 'This'
-- excluding the FOR will go from position 1 to end of string

-- replace every occurance of Clothing with Attire in a new column
SELECT department, REPLACE(department, 'Clothing', 'Attire') AS modified_depart
FROM employees;

-- return a column of integers that contain the position of @ in the email column
SELECT POSITION('@' IN email)
FROM employees;

-- use position and substring to extract email domain names into a new column
SELECT email, SUBSTRING(email FROM POSITION('@' IN email)+1) AS email_domain
FROM employees; -- for above statement, +1 ignores @ sign (increment start position)

-- create new column from email with null values filled in as NONE
SELECT COALESCE(email, 'NONE') AS email_filled
FROM employees;
```

1.2.2 Grouping Functions

We can perform calculations on data and get statistical insight from these queries (for numeric data). Note that grouping functions take in multiple rows, but only output one row (the calculation).

· MAX( ) - returns the highest numeric value in a given column
· MIN( ) - returns the lowest numeric value in a given column
· AVG( ) - returns the average numeric value in a given column
· ROUND(,) - rounds our data to a given decimal place value
· COUNT( ) - gives the total number of records in a column (excludes null values)
· SUM( ) - sums the numeric values in a given column

```
-- find the highest, lowest, and average paid salary in our table
SELECT MAX(salary) FROM employees;
SELECT MIN(salary) FROM employees;
SELECT ROUND(AVG(salary),3) FROM employees; -- average rounded to 3 decimal places

-- find total number of records in our table
SELECT COUNT(*) FROM employees;
-- note: we often use count(*) to make sure all records are counted incase of null's

-- find the yearly amount paid to employees
SELECT SUM(salary) FROM employees;
```

1.3 Grouping Data / Computing Aggregates

We can use the GROUP BY command to group records that are the same in a given column.

· Note that 'group by' comes after *where* clause but before *order by*.

· We can call the parameter for 'group/order by' with the column position in the select statement.

· Any non-aggregate columns in select list must also by mentioned in group by clause.

```sql
-- group salaries by department for region's 4, 5, 6, and 7
SELECT department, SUM(salary)
FROM employees
WHERE region_id IN (4,5,6,7)
GROUP BY department;

-- get number of employees, average/min/max salary for each department and
-- order from highest employee count to lowest
SELECT department, COUNT(*) AS num_of_employees,
  ROUND(AVG(salary), 3) AS average_salary,
  MIN(salary) AS lowest_salary,
  MAX(salary) AS highest_salary
FROM employees
GROUP BY 1 -- department
ORDER BY 2 DESC; -- num_of_employees

-- get the number of employees by gender for each department
SELECT department, gender, COUNT(*) AS num_of_employees
FROM employees
GROUP BY 1, 2 -- gender is non-aggregate function (must be included)
ORDER BY 1; -- order by department (easier to read)

-- how many people have the same first name (name and count)
SELECT first_name, COUNT(*) as occurances
FROM employees
GROUP BY 1
HAVING COUNT(*) > 1;

-- get unique domain names for email and the number of employees having that domain
SELECT SUBSTRING(email FROM POSITION('@' IN email)+1) AS email_domain,
COUNT(*) AS num_of_employees
FROM employees
WHERE email IS NOT NULL
GROUP BY 1
ORDER BY 2 DESC;

-- get the min/max/avg salary for each gender in every region
SELECT gender, region_id, MIN(salary) AS min_salary,
MAX(salary) AS max_salary,
ROUND(AVG(salary)) AS avg_salary -- round nearest whole num
FROM employees
GROUP BY 1,2
ORDER BY 1,2 ASC;
```

We can use the HAVING command to filter data that has been grouped (similar to where clause).

· This command is used to filter *aggregated* data.

· Note that the having command comes after the *group by* and before the *order by* clauses

```sql
-- get all department names that have more than 35 employees
SELECT department, COUNT(*)
FROM eployees
GROUP BY 1
HAVING COUNT(*) > 35
ORDER BY 1;
```

1.4 Using Subqueries

We can refer to columns of specific sources by specifying the table name before the column in the SELECT statement. We can make this even more simple by giving the sources aliases as their reference name.

· Note for future - we can have select statements nested in from statements (new source of data).

```
-- aliase employees table as 'e' and departments as 'd'
SELECT d.department -- get department column from departments table
FROM employees e, departments d;
```

We can use subqueries in the WHERE/FROM clause (acts as a source from which data can be pulled from). In the from clause, we need to give the subquery an alias.

· The inner query (subquery) returns a list of data that we can pull from.

· Renaming a subqueries column names means we must reference these names in the outer query.

· We can have multiple sources of data (subqueries) in our FROM clause.

```
-- select all employees who work in a department not listed in departments table
SELECT * FROM employees
WHERE department NOT IN (SELECT department from departments);

-- select first name and salary from subquery 'a' of employees with salary > 150k
SELECT a.first_name, a.salary
FROM (SELECT * FROM employees WHERE salary > 150000) a

-- renaming inner query names (same as above problem)
SELECT a.employee_name, a.yearly_salary
FROM (SELECT first_name AS employee_name, salary AS yearly_salary
      FROM employees WHERE salary > 150000) a

-- select all employees who work in the electronics division
SELECT * FROM employees
WHERE department IN (SELECT department
FROM departments WHERE division = 'Electronics');

-- select all employees that work in Asia or Canada and make over 130k
SELECT * FROM employees
WHERE region_id IN (SELECT region_id FROM regions WHERE country IN ('Asia','Canada'))
AND salary > 130000;
```

We can use subqueries in the SELECT statement, but we must make sure that the subquery only returns one record. This can be used to compare all records of outer query to one value from inner query (see example below).

```
-- INVALID SYNTAX EXAMPLE
SELECT first_name, salary, (SELECT first_name FROM employees)
FROM employees;
/* this will not run, the inner query will try to return all 1000 first_names
  for each record in outer query. */

/* select first_name and department of employee and how much less they make than
   the highet paid employee (in Asia and Canada) */
SELECT first_name, department,
       ((SELECT MAX(salary) from employees) - salary) AS less_income
FROM employees
WHERE region_id IN (SELECT region_id FROM regions
                    WHERE country IN ('Asia','Canada'));
```

We can us the ALL/ANY clause in the where/having clause, and they have the following function:

· ANY - will return true if any of the subquery values meet the condition.

· ALL - will return true if all subquery values meet the condition.

You can use all operator $>, <, >=, <=$ when using these clauses.

Note that ANY will still return values inside the subquery (tricky to use).

```sql
-- select all employees who's region_id is greater than US region id's (1,2,3)
SELECT * FROM employees
WHERE region_id > ALL (SELECT region_id FROM regions WHERE country='United States');

/* select all employees that work in the kids devision and the dates
at which they were hired is greater than all hire dates of employees
in the maintenance department */
SELECT * FROM employees
WHERE department IN (SELECT department FROM departments WHERE division = 'Kids')
AND hire_date > ALL (SELECT hire_date FROM employees
                     WHERE department = 'Maintenance');

-- select the salary that occures the most often (and is the highest value)
SELECT salary, COUNT(*) FROM employees
GROUP BY 1
ORDER BY 2 DESC, 1 DESC
LIMIT 1;
-- same as...
SELECT salary FROM employees
GROUP BY salary
HAVING COUNT(*) >= ALL(SELECT COUNT(*) FROM employees GROUP BY salary)
ORDER BY 1 DESC
LIMIT 1;

-- find the average salary excluding the lowest and highest paid employee
SELECT ROUND(AVG(salary)) FROM employees
WHERE salary < ALL (SELECT MAX(salary) FROM employees)
AND salary > ALL (SELECT MIN(salary) FROM employees);
```

1.5 The CASE Clause

We can use the CASE clause as a conditional expression to create a new column in the SELECT clause. We can also use CASE statements to transpose tables (rows to columns, columns to rows).

```sql
-- select the total number for each category of the pay scale
SELECT a.pay_category, COUNT(*) FROM (
  SELECT first_name, salary, -- subquery case statement
    CASE
      WHEN salary < 100000 THEN 'UNDER PAID'
      WHEN salary > 100000 AND salary < 160000 THEN 'PAID WELL'
      WHEN salary > 160000 THEN 'EXECUTIVE'
      ELSE 'UNPAID'
    END AS pay_category -- m name
  FROM employees
  ORDER BY salary DESC ) a -- access variable
GROUP BY 1;

-- transpose the above table using case statements (rows will have employee count)
SELECT SUM( CASE WHEN salary < 100000 THEN 1 ELSE 0 END) as under_paid,
  SUM( CASE WHEN salary > 100000 AND salary < 160000 THEN 1 ELSE 0 END) as paid_well,
  SUM( CASE WHEN salary > 160000 THEN 1 ELSE 0 END) as executive
FROM employees;
```

```sql
-- find total number of employees for the given departments
SELECT department, count(*)
FROM employees
WHERE department IN ('Sports', 'Tools', 'Clothing', 'Computers')
GROUP BY 1;

-- transpose the above table using case statements (rows will have employee count)
SELECT SUM( CASE WHEN department = 'Sports' THEN 1 ELSE 0 END) as sports_employees,
  SUM( CASE WHEN department = 'Tools' THEN 1 ELSE 0 END) as tools_employees,
  SUM( CASE WHEN department = 'Clothing' THEN 1 ELSE 0 END) as clothing_employees,
  SUM( CASE WHEN department = 'Computers' THEN 1 ELSE 0 END) as computers_employees
FROM employees;

-- county how many employees are in each country
SELECT COUNT(a.region_1) + COUNT(a.region_2) + COUNT(a.region_3) AS united_states,
  COUNT(a.region_4) + COUNT(a.region_5) AS asia,
  COUNT(a.region_6) + COUNT(a.region_7) AS canada
FROM (SELECT first_name,
  CASE WHEN region_id = 1 THEN (SELECT country FROM regions WHERE region_id = 1)
  END AS region_1,
  CASE WHEN region_id = 2 THEN (SELECT country FROM regions WHERE region_id = 2)
  END AS region_2,
  CASE WHEN region_id = 3 THEN (SELECT country FROM regions WHERE region_id = 3)
  END AS region_3,
  CASE WHEN region_id = 4 THEN (SELECT country FROM regions WHERE region_id = 4)
  END AS region_4,
  CASE WHEN region_id = 5 THEN (SELECT country FROM regions WHERE region_id = 5)
  END AS region_5,
  CASE WHEN region_id = 6 THEN (SELECT country FROM regions WHERE region_id = 6)
  END AS region_6,
  CASE WHEN region_id = 7 THEN (SELECT country FROM regions WHERE region_id = 7)
  END AS region_7
  FROM employees) a; /* subquery 'a' has a column for the name of each employee and a
  column for each region, if an employee works in the region it has the country name,
  otherwise it has a null value */
```

1.6 Correlated Subqueries

A correlated subquery means that the subquery portion is correlated to the outer query (in other words, the nested subquery uses values from the outer query). This also means that the nested query will run for every single record of the outer query since we are forming a link between the two. Note that as the number of records grow, this method becomes more and more time consuming.

```sql
-- get all employees who make over the average salary in their department
SELECT first_name, salary
FROM employees e1
WHERE salary > (SELECT ROUND(AVG(salary)) -- subquery for average of each department
                FROM employees e2 WHERE e1.department = e2.department);

-- get first name, department, salary, and average department salary
SELECT first_name, department, salary,
  (SELECT ROUND(AVG(salary))  FROM employees e2 WHERE e1.department = e2.department)
FROM employees e1;

-- get all departments that have more than 38 employees and their max salary
SELECT department, (SELECT MAX(salary) FROM employees WHERE department= d.department)
FROM departments d
WHERE 38 < (SELECT COUNT(*) FROM employees e2 WHERE e2.department = d.department);
```

1.7 Working with Multiple Tables

1.7.1 Table Unions

We can use the UNION clause to stack data on top of each other (note that this will remove any duplicates and only show unique values). However, we can use the UNION ALL clause to stack data on top of each other without removing any duplicates (show all records from both data sources). Also note that the columns you are using UNION on must match for both tables.

```
-- select the number of employees for each department and a total count as the bottom
SELECT department, COUNT(*)
FROM employees
GROUP BY 1
UNION ALL
SELECT 'TOTAL', COUNT(*)
FROM employees;
```

We can use the EXCEPT clause to take the first result set and removes from it all the rows found in the second result set (records that exist in first data source but not second data source).

```
-- select all department from employees not in departments table
SELECT DISTINCT department
FROM employees
EXCEPT
SELECT department
FROM departments;
```

1.7.2 Table Joins

Joining is a way to link two or more tables together through a common column. We can do a basic join through the WHERE clause, and specifying which matching columns we want to join on. Note that calling a column that is ambiguous (same in multiple tables) must be specified which table you want to pull it from (by table name or alias). We can join subqueries since they are a source of data.

```
/* get first name, email (non-null values), department, division, and country for
   all employees */
SELECT first_name, email, e.department division, country -- specify department from e
FROM employees e, departments d, regions r
WHERE (e.department = d.department) -- join e with d
  AND (e.region_id = r.region_id) -- join (e/d) with r
AND email IS NOT NULL;

-- get the number of employees in each country (make regions a subquery)
SELECT country, COUNT(employee_id) AS num_of_employees
FROM (SELECT * from regions) r, employees e
WHERE e.region_id = r.region_id
GROUP BY 1;
```

We can use the INNER JOIN... ON clauses to combine two or more tables on a specific column with *matching* data. Note that for all data that does not match, it will be left out of the combined table.

```
SELECT first_name, email, division, country
FROM employees INNER JOIN departments
  ON employees.department = departments.departments
INNER JOIN regions ON employees.region_d = regions.region_id
WHERE email IS NOT NULL;
```

We can use the LEFT JOIN... ON clauses to combine two or more tables, and keep all data from the first table even if it does not match any data in the second table. Similarly, we can use RIGHT JOIN... ON clauses to keep all data from the second table even if it's not in the first table. Note that for both, any missing values will be filled in with *null*.

```sql
-- get all department records that are in employees but not in departments
SELECT DISTINCT employees.department
FROM employees LEFT JOIN departments
ON employees.department = departments.department
WHERE departments.department IS NULL;
```

The CROSS JOIN clause lets us find the Cartesian product of two data sources (multiply each data record in the first source with every data record in the second source, all possible combinations).

```sql
-- employees table (1000 records) x departments table (24 records) = 24000 records
SELECT * FROM employees CROSS JOIN departments;
```

Join, Union, and Subquery combined example:

```sql
-- select the first and last employee hired along with department and country
(SELECT first_name, department, hire_date, country
FROM employees INNER JOIN regions
  ON employees.region_id = regions.region_id
WHERE hire_date = (SELECT MIN(hire_date) FROM employees e2)
LIMIT 1) -- closing in parentheses lets us use the LIMIT clause
UNION ALL
SELECT first_name, department, hire_date, country
FROM employees INNER JOIN regions
  ON employees.region_id = regions.region_id
WHERE hire_date = (SELECT MAX(hire_date) FROM employees e2)
```

1.7.3 Creating Views

Views are created based on queries (can't insert or delete data from a view) and are saved to the database we are working within. We can access columns from the view just like other data sources. The standard to name these views are to put a v_ before the name so that anyone accessing knows it is a view. Also, every time you access a view, it is running the query that the view is made of.

```sql
-- save view to be accessed later
CREATE VIEW v_employee_information AS
SELECT first_name, email, e.department, salary, division, region, country
FROM employees e, departments d, regions r
WHERE e.department = d.department
  AND e.region_id = r.region_id;

SELECT first_name, region FROM v_employee_information;
```

1.8 Window Functions

The OVER() keyword allows us to detach a given aggregate function from a subquery and group it on a particular window by passing PARTITION BY inside the parenthesis. Note that leaving parenthesis blank will cause the function to use the entire data source as the window.

```sql
-- select the first name, department, and how many employees in the given department
SELECT first_name, department, COUNT(*) OVER(PARTITION BY department)
FROM employees

/* select name, department, number of employees in the current department, and
   the number of employees in the current region */
SELECT first_name, department, region_id,
COUNT(*) OVER(PARTITION BY department) AS dept_count,
COUNT(*) OVER(PARTITION BY region_id) AS region_count
FROM employees;
```

We can use the ORDER BY and RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW (*this is optional since it is default*) clause, meaning sum all values from first record to the current row, in the OVER() function to compute data in a given order.

```
-- select name, hire date, salary, and the running total of salaries
SELECT first_name, hire_date, salary,
SUM(salary) OVER(ORDER BY hire_date RANGE BETWEEN UNBOUNDED PRECEDING AND
                 CURRENT ROW) AS running_total_of_salaries
FROM employees;

-- find the running total of salaries for each department
SELECT first_name, hire_date, department, salary,
SUM(salary) OVER(PARTITION BY department ORDER BY hire_date) AS salary_running_total
FROM employees;
```

We can pass ROWS BETWEEN _ PRECEDING AND CURRENT ROW to add the value from the _ previous row(s) and current row together (compute calculations on adjacent rows).

```
SELECT first_name, hire_date, department, salary,
SUM(salary) OVER(ORDER BY hire_date ROWS BETWEEN 1 PRECEDING AND
                 CURRENT ROW) AS salary_running_total
FROM employees;
```

We can use the RANK() clause to rank a given partition by on the parameters within the OVER() clause.

```
-- get all employees who are in the 8th rank for their given department
SELECT * FROM (
  SELECT first_name, email, department, salary,
  RANK() OVER(PARTITION BY department ORDER BY salary DESC)
  FROM employees ) a
WHERE rank = 8;
```

We can use the NTILE() function, where we pass a parameter of how many groups to split the data into, to rank groups of rows (think of as creating brackets).

```
-- split each department into 5 brackets based on their salary
SELECT first_name, email, department, salary,
NTILE(5) OVER(PARTITION BY department ORDER BY salary DESC) AS salary_bracket
FROM employees;
```

We can use the FIRST_VALUE() function to get the first value of a specified column, which is passed as a parameter, for each group that we are partitioning by. We can also use NTH_VALUE(,) by passing the column name and the value position we wish to get.

```
-- get the first salary for each department (highest paid)
SELECT first_name, email, department, salary,
FIRST_VALUE(salary) OVER(PARTITION BY department ORDER BY salary DESC)
FROM employees;
```

We can use the LAG() function to get the previous value of a column that is passed as a parameter. Similarly, we can use the LEAD() function to get the next value of a column that is passed as a parameter.

```
-- get the salary of the employee that is one below the current employee
SELECT first_name, department, salary,
LEAD(salary) OVER(PARTITION BY department ORDER BY salary DESC) AS lower_salary
FROM employees;
```

We can group by multiple values to see each result in one query by using the GROUPING SETS( ) clause and passing the columns we wish to group by (it will fill in null for all missing records in the column not being grouped by). Note that passing () to the grouping sets clause will execute the aggregate function on all of the data.

```sql
/* group by continent, country, and city to see total units sold for each and also
   the total units sold in the entire table */
SELECT continent, country, city, sum(units_sold)
FROM SALES
GROUP BY GROUPING SETS(continent, country, city, ());
```

Similar to grouping sets, we can use the ROLLUP() clause. Lets assume we want to group by 3 columns, then rollup will show us the results if we grouped by nothing (only shows the aggregate function value), all columns, the first 2 columns, and the first column all in one query. We can also use the CUBE() clause to show all possible combinations of groupings for any columns that are passed as parameters.

```sql
SELECT continent, country, city, sum(units_sold)
FROM SALES
GROUP BY ROLLUP(continent, country, city);
```

1.9 Practice Problems with Solutions

(1.1 SQL Query Basics)

```sql
/* First name and email of females that work in the tools department having a
   salary greater than 110,000 */
SELECT first_name, email FROM employees
WHERE gender = 'F' AND department = 'Tools' AND salary > 110000;

/* First name and hire date of employees who earn more than 165,000 as well as
   employees that work in the sports department and are men */
SELECT first_name, hire_date FROM employees
WHERE salary > 165000 OR (department = 'Sports' and gender = 'M');

/* First name and hire date of employees hired during Jan 1, 2002 and Jan 1, 2004 */
SELECT first_name, hire_date FROM employees
WHERE hire_date BETWEEN '2002-01-01' AND '2004-01-01';

/* All columns from male employees who work in the automotive department and earn
   more than 40,000 and less than 100,000 as well as females that work in the
   toy department */
SELECT * FROM employees
WHERE gender = 'M' AND department = 'Automotive'
AND (salary BETWEEN 40000 and 100000)
OR (gender = 'F' AND department = 'Toys');
```

(1.2 Using Functions)

```sql
/* Write a query against the professors table that can output the following in the
   result: "Chong works in the Science department" */
SELECT last_name || ' works in the ' || department || ' department'
FROM professors
WHERE last_name = 'Chong';

/* Write a query that says if a professor mis highly paid (above 95000)
   in the format "It is false that professor Chong is highly paid" */
SELECT 'It is ' || (salary > 95000) || ' that professor ' || last_name ||
       ' is highly paid'
FROM professors;
```

```sql
/* Write a query that returns all of the records and columns from the professors
   table but shortens the department names to only the first three characters in
   upper case. */
SELECT last_name, UPPER(SUBSTRING(department FROM 1 FOR 3)) AS department_abrv,
       salary, hire_date
FROM professors;

/* Write a query that returns the highest and lowest salary from the professors
   table excluding the professor named 'Wilson'. */
SELECT MAX(salary) AS max_salary, MIN(salary) AS min_salary
FROM professors
WHERE last_name != 'Wilson';

/* Write a query that will display the hire date of the professor that has been
   teaching the longest. */
SELECT MIN(hire_date) FROM professors;
```

(1.3 Grouping Data / Computing Aggregates)

```sql
-- Write a query that displays only the state with the largest amount of fruit supply
SELECT state
FROM fruit_imports
GROUP BY 1
ORDER BY SUM(supply) DESC
LIMIT 1;

-- Write a query that returns the state that has more than 1 import of the same fruit
SELECT state
FROM fruit_imports
GROUP BY 1, name
HAVING COUNT(name) > 1;

-- Write a query that returns the seasons that produce either 3 fruits or 4 fruits.
SELECT season
FROM fruit_imports
GROUP BY 1
HAVING ((COUNT(name) = 3) OR (COUNT(name) = 4));

/* Write a query that takes into consideration the supply and cost_per_unit columns
   for determining the total cost and returns the most expensive state with the total
   cost. */
SELECT state, SUM(supply * cost_per_unit) AS total_cost
FROM fruit_imports
GROUP BY 1
ORDER BY 2 DESC
LIMIT 1;

-- Execute the below SQL script and write a query that returns the count of 4.
CREATE table fruits (fruit_name varchar(10));
INSERT INTO fruits VALUES ('Orange');
INSERT INTO fruits VALUES ('Apple');
INSERT INTO fruits VALUES (NULL);
INSERT INTO fruits VALUES (NULL);

SELECT COUNT(COALESCE(fruit_name, 'SOMEVALUE'))
FROM fruits;
```

(1.4 Using Subqueries)

```sql
/* Write a query that returns the names of those students that are taking
   Physics and US History. */
SELECT student_name FROM students
WHERE student_no IN (SELECT student_no FROM student_enrollment
                     WHERE course_no IN (SELECT course_no FROM courses
                                         WHERE course_title
                                         IN ('Physics', 'US History')));

-- Write a query that returns the name of the student that is taking the most courses
SELECT student_name FROM students
WHERE student_no = (SELECT student_no FROM student_enrollment
                    GROUP BY 1
                    ORDER BY COUNT(student_no) DESC
                    LIMIT 1);

-- Write a query to find the student that is the oldest (no LIMIT or ORDER BY)
SELECT student_name FROM students
WHERE age = (SELECT MAX(age) FROM students);
```

(1.5 The CASE Clause)

```sql
/* Write a query that displays 3 columns. The query should display the fruit and
it's total supply along with a category of either LOW, ENOUGH or FULL */
SELECT name, total_supply,
CASE
  WHEN total_supply < 20000 THEN 'LOW'
  WHEN total_supply > 20000 AND total_supply < 50000 THEN 'ENOUGH'
  WHEN total_supply > 50000 THEN 'FULL'
END AS supply_category
FROM (SELECT name, sum(supply) AS total_supply
    FROM fruit_imports
    GROUP BY name) a;

/* Taking into consideration the supply column and the cost_per_unit column, you
should be able to tabulate the total cost to import fruits by each season. Write
a query that would transpose this data so that the seasons become columns and
the total cost for each season fills the first row */
SELECT SUM( CASE WHEN season = 'Winter' THEN total_cost ELSE 0 END) AS "Winter",
  SUM( CASE WHEN season = 'Summer' THEN total_cost ELSE 0 END) AS "Summer",
  SUM( CASE WHEN season = 'All Year' THEN total_cost ELSE 0 END) AS "All Year",
  SUM( CASE WHEN season = 'Spring' THEN total_cost ELSE 0 END) AS "Spring",
  SUM( CASE WHEN season = 'Fall' THEN total_cost ELSE 0 END) AS "Fall"
FROM (SELECT season, SUM(supply * cost_per_unit) AS total_cost
    FROM fruit_imports
    GROUP BY season) a;
```

(1.6 Correlated Subqueries)

```sql
/* Write a query that gets the min and max salary for each department and says which
   is the HIGHEST and LOWEST salary (should have 2 records for each department in
   final query */
SELECT department, first_name, salary,
CASE WHEN salary = d_max THEN 'HIGHEST SALARY'
     WHEN salary = d_min THEN 'LOWEST SALARY'
END AS salary_in_department
FROM (SELECT department, first_name, salary,
       (SELECT MAX(salary) FROM employees WHERE department = e1.department) AS d_max,
       (SELECT MIN(salary) FROM employees WHERE department = e1.department) AS d_min
      FROM employees e1
      ORDER BY 1) a
WHERE salary = dep_max OR salary = dep_min
ORDER BY 1;
```

(1.7 Multiple Tables)

```sql
/* Write a query that shows the student's name, the courses the student is taking
   and the professors that teach that course.*/
SELECT student_name, se.course_no, p.last_name
FROM students s
INNER JOIN student_enrollment se
  ON s.student_no = se.student_no
INNER JOIN teach t
  ON se.course_no = t.course_no
INNER JOIN professors p
  ON t.last_name = p.last_name
ORDER BY student_name;

/* In the above problem, you discovered why there is repeating data. How can we
   eliminate this redundancy? Make every record distinct. */
SELECT student_name, course_no, min(last_name)
FROM (
  SELECT student_name, se.course_no, p.last_name
  FROM students s
  INNER JOIN student_enrollment se
    ON s.student_no = se.student_no
  INNER JOIN teach t
    ON se.course_no = t.course_no
  INNER JOIN professors p
    ON t.last_name = p.last_name
  ) a
GROUP BY student_name, course_no
ORDER BY student_name, course_no;

/* write a query that returns employees whose salary is above average for
   their given department. */
SELECT first_name
  FROM employees outer_emp
  WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department = outer_emp.department);
```