# INFS1603/COMM1822 Demo Script



## Table of Content







*You can "<ctrl> + click" on the week name (e.g., Week 3B SQL Basics) to access the demo scripts.*

## Introduction

In the lab, students will need to get familiar with a range of personal computer software, including SQLDeveloper. The labs are available for students to do this on a self-taught basis, using the recommended workbooks or equivalent alternatives. Please see Course Outline to see the lab information (e.g., location and time).

This document provides lab demos.

Last update in January 2022.

## Run the following first

```
---------------------------------------------------------------
ALTER SESSION SET nls_date_format = 'DD-MON-RR';
--Comment and will be by ignored by Oracle SQL Developer
```

# Week 3A SQL Basics – Table Creation and Select Statements

```
-------------------------------------------------------------
-- SELECT Statement                                        --
-------------------------------------------------------------

-- [1] Select all data from a table, in this case, the employees table:
SELECT * FROM employees;

-- [2] Retrieve column names / metadata from a table:
DESCRIBE employees;

-- [3] Select one column only:
SELECT last_name
FROM employees;

-- [4] Select more than one column:
SELECT employee_id, last_name, first_name
FROM employees;

-- [5] Four common heading alias methods
-- Note: (a) Use of double quotes (") for heading alias
--           (b) Do not use single quotes (') for alias
SELECT employee_id AS Employee,
  employee_id AS "Employee Id",
  last_name Surname,
  first_name "First Name"
FROM employees;

-------------------------------------------------------------
-- SELECT DISTINCT Statement                               --
-------------------------------------------------------------

-- [6] Without DISTINCT, you will retrieve 107 records:
SELECT job_id FROM employees;

-- [7] With DISTINCT, you only retrieve distinct/unique jobs
-- in our example, 19 distinct jobs:
SELECT DISTINCT job_id FROM employees;

-- [8] UNIQUE will give you the same result
SELECT UNIQUE job_id FROM employees;

-- [9] Select distinct department_id
-- it returns 12 records:
SELECT DISTINCT department_id FROM employees;

-- [10] Select distinct the manager_id
-- it returns 19 records:
SELECT DISTINCT manager_id FROM employees;

-- [11] Select distinct combinations of department_id and manager_id
-- it returns 28 records:
SELECT DISTINCT department_id,
manager_id
FROM employees;



-------------------------------------------------------------
-- Concatenation                                           --
```

```
-----------------------------------------------------------------

-- [12] Use single quotes to display a string/literal,
-- Use double bars (||) to concatenate strings.
SELECT first_name ||
' ' ||
last_name ||
' is a(n) ' ||
        job_id AS "Employee's Job"
FROM employees;

-- [13] Use q'< and >' as delimiters to
-- to include a quotation mark in the literal:

SELECT first_name ||
        ' ' ||
        last_name ||
        q'<'s job id is >' ||
job_id AS "Employee's Job"
FROM employees;

-----------------------------------------------------------------
-- Arithmetic operations                                       --
-----------------------------------------------------------------

-- [14] Use multiplications and additions in the query:

SELECT last_name,
        salary,
        salary * 0.10 AS "Increase",
        salary * 1.10 AS "New Salary",
        100 * 12 AS "Expenses",
        (salary * 1.10) + (100 * 12) AS "New Sal w/Exp"
FROM employees;
```

```
-----------------------------------------------------------------
-- Week 3 Part B
-- Table Creation, Management, Constraints
-----------------------------------------------------------------


-----------------------------------------------------------------
-- Assume you know:
--  (a) What is a Primary Key
--  (b) What is a Foreign Key
--  (c) What is a Constraint
-----------------------------------------------------------------

-----------------------------------------------------------------
-- Delete Tables:                                --
--   You can only delete a table if it does not violate       --
--   any constraints. For instance, you can only delete       --
--   a table if no column from any other tables references    --
--   this table. Thus, the construction or deletion of        --
--   constraints will determine the order of creating or      --
--   removing tables.                                         --
-----------------------------------------------------------------
-- Tidy Up tables if they exists in the database
-- You do not have to worry if you see error message or similar
-- such as:
--    SQL Error: ORA-00942: table or view does not exist
--    00942. 00000 -  "table or view does not exist"

DROP TABLE departments_History;
DROP TABLE employees_history;
DROP TABLE employees_hist_2011;
DROP TABLE locations_temp;
DROP TABLE global_region_temp;

-----------------------------------------------------------------
-- Create a New Table:                           --
--   (a) Without Data                            --
--   (b) With Data                               --
-----------------------------------------------------------------

-- [1] Create a table without data!
-- Create a table called departments_history.
-- Initialise the table with default values.
CREATE TABLE departments_history
  (
    department_id       NUMBER(4),
    department_name     VARCHAR2(30),
    location_id         NUMBER(4),
    approved_employee_id NUMBER(6),
    active_flag CHARACTER(1) DEFAULT 'Y',
    modify_date DATE DEFAULT SYSDATE
  );

-- [2###] Returns the properties of the table which you just created
DESCRIBE departments_history;

-- [3] No data is added
SELECT * FROM departments_history;
```

```
-- [4] List all the tables in your database.
SELECT table_name
FROM   user_tables
ORDER BY table_name;

----------------------------------------------------------------

-- [5]Create a Table With Data:
--  Create employees_history table data from employees table data
--  Note: not all the constraints are not created
CREATE TABLE employees_history
  AS (SELECT * FROM employees);

-- [6###]
DESCRIBE employees_history;

-- [7]Data is now added!
SELECT * FROM employees_history;

----------------------------------------------------------------
-- Adding, modifying and removing a column from a table       --
----------------------------------------------------------------
-- [8]Add a new column
ALTER TABLE employees_history
  ADD (modify_date DATE);

-- [9]
DESCRIBE employees_history;

-- [10] Remove a column
ALTER TABLE employees_history
  DROP COLUMN phone_number;

-- [11] Modify a column
-- Change email from VARCHAR2(25) to VARCHAR2(30)
-- Remember: varchar = letters/numbers
ALTER TABLE employees_history
  MODIFY (email VARCHAR2(30));

-- [12] It will also reflect in your table descriptions
DESCRIBE employees_history;

----------------------------------------------------------------
-- Making a column to be UNUSED                               --
----------------------------------------------------------------

-- [13] Instead of removing a column immediately, which requires
-- excusive locking* of a table, you might want to make
-- the column unavailable to the users by defining the column
-- as UNUSED.
-- *) we will cover locking in a later course
ALTER TABLE employees_history
  SET UNUSED (email);

-- [14] Notice that you don't see the email column anymore
DESCRIBE employees_history;

-- [15] You can remove all the unused columns at a later stage
ALTER TABLE employees_history
DROP UNUSED COLUMNS;
```

**-- [16###]**
DESCRIBE employees_history;

---
-- Renaming a table, and recoving a deleted table --
---

**-- [17] Renaming a table**
RENAME employees_history TO employees_hist_2011;

**-- [18###]**
DESCRIBE employees_hist_2011;

**-- [19] Notice the table name changed**
SELECT table_name FROM USER_TABLES;

**-- [20] Deleting a table**
DROP TABLE employees_hist_2011;

**-- [21]'Oh No - delete the wrong table'**
-- Recovering a table
-- Find the deleted table in the recycle bin like in Windows
SELECT object_name, original_name
FROM recyclebin;



**-- [22] Recovering from the recycle bin**
FLASHBACK TABLE employees_hist_2011 TO BEFORE DROP;



**-- [23###]**
DESCRIBE employees_hist_2011;

**-- [24] To delete a table permanently by referencing**
-- the exact object name, copy and paste the object name
-- from Query 19



PURGE TABLE "BIN$VxWQy4n/0/gUKuVSZ5IEA==$0";

**-- [25] Alternatively, you can empty a recycle bin like in Windows**
PURGE recyclebin;

**-- [26]Your recycle bin should be empty**
SELECT object_name, original_name
FROM recyclebin;

---
-- End of Oracle Lab Demo Week 8 --
---

# Week 4 SQL – Restricting Rows, Single-row Functions

```
-------------------------------------------------------------
-- The WHERE clause                                        --
-------------------------------------------------------------

-- [1] Select a records/rows based on a condition
-- using a specific numeric value:
SELECT employee_id,
            last_name,
            first_name,
            Manager_id
FROM employees
WHERE manager_id = 102;

-- [2] Select records/rows based on a condition
-- using a specific character string:
SELECT employee_id,
            last_name,
            First_name
FROM employees
WHERE last_name = 'King';

-- [3] Select records/rows based on a condition
-- using a specific date:
SELECT employee_id,
            last_name,
            first_name,
            Hire_date
FROM employees
WHERE hire_date = '08-MAR-08';

-- Hint: Not working? Date must be in the format DD-MON-RR!
-- This can be set up via > Tools > Preferences > Database > NLS
-- Or via the command ALTER SESSION SET nls_date_format = 'DD-MON-RR';

-------------------------------------------------------------
-- Comparing values                                        --
-------------------------------------------------------------

-- [4] Condition based on values being above a numeric value (not inclusive)
SELECT employee_id,
            last_name,
            first_name,
            Salary
FROM employees
WHERE salary > 10000;

-- [5] Condition based on a range of numeric values (not inclusive)
SELECT employee_id,
            last_name,
            first_name,
            Salary
FROM employees
WHERE salary > 10000 AND  salary < 11000;




-- [6] Condition based on a range of numeric values (inclusive)
SELECT employee_id,
```

```
            last_name,
            first_name,
            Salary
FROM employees
WHERE salary >= 10000 AND  salary <= 11000;

-- [7] Condition based on NOT selecting a value
SELECT department_id, department_name, location_id
FROM departments
WHERE location_id <> 1700;

-- [8] Alternatively, you can use BETWEEN ... AND ... operator.
-- Border values are included in result:
SELECT employee_id,
  last_name,
  first_name,
  salary
FROM employees
WHERE salary BETWEEN 10000 AND 11000;

-- [9] Selecting a range of records based on a character
SELECT employee_id, last_name, first_name
FROM employees
WHERE last_name > 'T';

-- [10] Comparing alphanumeric values, i.e. values contain numbers,
-- alphabets and space, this is case sensitive:

SELECT location_id, postal_code, country_id
FROM locations
WHERE postal_code = 'YSW9T2';

-- Hint: How does it work? See http://www.asciitable.com/ (for example)

-- [10] Can also use >, <, >= and <= for characters/string
-- Has the first characteran Ascii value of higher than '9'?
SELECT location_id, postal_code, country_id
FROM locations
WHERE postal_code > '9';

-- [10] Condition selecting a range of strings
SELECT employee_id, last_name, first_name
FROM employees
WHERE last_name >= 'King' AND  last_name <= 'Lee';

-- [11] Alternatively, you can use the BETWEEN ... AND ... operator
-- but you must specify the lower limit first:
SELECT employee_id, last_name, first_name
FROM employees
WHERE last_name BETWEEN 'King' AND 'Lee';

-- [12] If you specify the upper limit first, no record will be retrieved:
SELECT employee_id, last_name, first_name
FROM employees
WHERE last_name BETWEEN 'Lee' AND 'King';



-- [13] Remember value inside the quotes is case sensitive:
SELECT employee_id, last_name, first_name
FROM employees
```

```
WHERE last_name = 'King';

-- [14] you will not retrieve any records if 'KING' is entered:
SELECT employee_id, last_name, first_name
FROM employees
WHERE last_name = 'KING';

-- [15] Selecting a range of values based on a date
SELECT employee_id,
            last_name,
            first_name,
            Hire_date
FROM employees
WHERE hire_date >= '08-MAR-08';

-- [16] Both Mar and MAR work here
-- (because it is a date, not a string)
SELECT employee_id,
            last_name,
            first_name,
            hire_date
FROM employees
WHERE hire_date >= '08-Mar-08';

-------------------------------------------------------------
-- IN, NOT, NULL and LIKE                                  --
-------------------------------------------------------------

-- [17] Find employees working in the department of
--Human Resources (id = 40) or IT (60) - 6 rows
SELECT employee_id, last_name, first_name, department_id
FROM employees
WHERE department_id IN (40, 60);

-- [18] Find employees NOT working in the
-- department of HR(department id = 40) or IT (60) - 100 rows
SELECT employee_id, last_name, first_name, department_id
FROM employees
WHERE department_id NOT IN (40, 60);

-- Hint: There are 107 employees but the two above SELECT statements
-- only retrieved 106 employees. Why is one employee is missing?
-- One employee s/he does not have a department_id.
-- You can only find the missing employee by checking for NULL.

-- [19] Checking for NLULL values in department_id:
SELECT employee_id, department_id
FROM employees
WHERE department_id IS NULL;

-- [20] Comparisons including '=' do not work with NULL
SELECT employee_id, department_id
FROM employees
WHERE department_id = NULL;
-- should use WHERE department_id is NULL;


-------------------------------------------------------------
-- Wildcards                                               --
-------------------------------------------------------------
```

```
-- [21] '%' (percentage): match any characters
-- Find all employees with surname starts with 'S':
SELECT employee_id,
        last_name,
        First_name
FROM employees
WHERE last_name LIKE 'S%';

-- [22] Find all employees with surname ends with double 'l's - 6 rows
SELECT employee_id, last_name, first_name
FROM employees
WHERE last_name LIKE '%ll';

-- [23] Find all employees with surname contains double 'l's - 13 rows
SELECT employee_id, last_name, first_name
FROM employees
WHERE last_name LIKE '%ll%';


-- [23] '_' (underscore): match one character
-- Find employee id - match patterns starts with 1,
-- any number in the middle, and ends with 9,
-- and start working in the year 2008.
SELECT employee_id, last_name, first_name, hire_date
FROM employees
WHERE employee_id LIKE '1_9' AND hire_date LIKE '%-08';
```



```
-------------------------------------------------------------
-- ORDER BY                                                --
-------------------------------------------------------------
```



```
-- [24] Hint: for rules of precedence
SELECT employee_id, last_name, first_name
FROM employees
ORDER BY last_name;
```



```
-- [25] Sort by column number (here second column) in the SELECT list
-- Not recommended! But you will see it is commonly used in
-- system/auto-generated codes:
SELECT employee_id, last_name, first_name
FROM employees
ORDER BY 2;

-- [26] Sort by both manager_id and then hire_date:
SELECT employee_id,
        last_name,
        manager_id,
        Hire_date
FROM employees
ORDER BY manager_id, hire_date;


-- [27] Descending ordering (instead of the default ascending ordering)
-- Same query as above, sorted by both manager_id and hire_date
-- but now hire_date is sorted in descending order:
SELECT employee_id,
        last_name,
        manager_id,
        Hire_date
FROM employees
```

```
ORDER BY manager_id, hire_date DESC;


-----------------------------------------------------------------
--              Section 2 – Single-Row Functions          --
-----------------------------------------------------------------


-- [1] Case Conversion Functions
-- Examples of using UPPER(), LOWER() and INITCAP():
SELECT last_name,
               UPPER(last_name),
               LOWER(last_name),
               email,
               INITCAP(email)
FROM employees
WHERE UPPER(last_name) = 'KING';

-- [2] Character Manipulation Functions
-- String starts from one (1) and not zero (0)
-- Padding related to the desired length of the result, not the
-- number of characters your want to insert:
SELECT first_name,
               SUBSTR(first_name,1,3),
               LENGTH(first_name),
               LPAD(salary,10,'*'),
               RPAD(salary,10,'*')
FROM employees
WHERE UPPER(last_name) = 'KING';

-- [3] Number Functions
-- The round function needs the position of the rounding,
-- e.g., 0 is rounding to the decimal place
-- 1 is rounding to one digit behind the decimal place
-- -1 is rounding to one digit before the decimal place
-- 2    5     .   0   0
-- (-2) (-1) (0) (1) (2)
-- Example of the ROUND Function:
SELECT commission_pct,
               commission_pct * 100,
               ROUND(commission_pct * 100,-1),
               commission_pct * 10,
               ROUND(commission_pct * 10,0),
               ROUND(commission_pct * 10,1)
FROM employees
WHERE department_id = 80 AND employee_id BETWEEN 151 AND 154;

-- [4] The Truncation function cuts later numbers off (does not round)
-- It doesn't reduce the length of the string though
-- Example of the TRUNC Function:
SELECT commission_pct,
               commission_pct * 100,
               TRUNC(commission_pct * 100,-1),
               commission_pct * 10,
               TRUNC(commission_pct * 10,0),
               TRUNC(commission_pct * 10,1)
FROM employees
WHERE department_id = 80 AND employee_id BETWEEN 151 AND 154;

-- [5] How many years here? – No rounding, no truncation:
SELECT last_name,
```

```
        hire_date,
        MONTHS_BETWEEN(sysdate,hire_date)/12 "Approx.(years)"
FROM employees
ORDER BY hire_date DESC;

-- [6] How many years here? – With rounding: (5 places after the decimal point)
SELECT last_name,
        hire_date,
        ROUND(MONTHS_BETWEEN(sysdate,hire_date)/12,5) "Approx.(years)"
FROM employees
ORDER BY hire_date DESC;

-- [7] How many years here? – With truncating: (5 places after the decimal point)
SELECT last_name,
        hire_date,
        TRUNC(MONTHS_BETWEEN(sysdate,hire_date)/12,5) "Approx.(years)"
FROM employees
ORDER BY hire_date DESC;

-----------------------------------------------------------------
-- Dual Table - A "dummy table"                                 --
-----------------------------------------------------------------

-- Hint: Purpose is for calculations and system functions.
-- This dummy table is used when you're not actually interested in
-- the data, but instead just wan to execute calculation/functions:

-- [8] Get the system date:
SELECT sysdate FROM dual;

-- [9] What is actually in the table: One column, one row, one value:
SELECT * FROM dual;

-- [10] ABS is mathematical function that returns the absolute (positive)
-- value of the specified numeric expression:
SELECT -12.5, ABS(-12.5)
FROM dual;

-- [11] Demonstrate the use of POWER function:
SELECT 2*2*2, POWER(2,3)
FROM dual;

-- [12] An example of using DATE functions to calculate how many
-- years (approx.) have the employees work for the company
-- Here you use the months_between function, to calculate the time
-- between the current date and the hire date
-- this is divided by 12 to show years. It is then truncated to 1
-- decimal place:
SELECT last_name,
        hire_date,
        TRUNC(MONTHS_BETWEEN(sysdate,hire_date)/12,0) "Approx. (years)"
FROM employees
ORDER BY hire_date DESC;

-- [13] Use the NVL function to substitute a NULL value with 0 (zero).
-- Problem without NVL: Any numbers multipled by a NULL is NULL.
-- Use Query Result option to show the NULL value.
-- This is highlighted in the example below.
-- Without NVL you get NULL for products:
SELECT department_id, salary, commission_pct,
        salary + (salary * commission_pct) "Salary and Comm."
```

```
FROM employees
WHERE employee_id BETWEEN 143 AND 146
ORDER BY employee_id;

-- [14] Use NVL function to substitute a NULL value with 0 (zero)
-- When compared with the last query, this gives Salary and
-- comm a value, whereas otherwise it would be equal to null
SELECT department_id, salary, commission_pct,
  salary + (salary * NVL(commission_pct,0)) "Salary and Comm."
FROM employees
WHERE employee_id BETWEEN 143 AND 146
ORDER BY employee_id;

-- [15] Use of the NVL2 function.
-- The syntax is NVL2(x,y,z):
--    if x
--    is NOT NULL then use y
--    is NULL then use z
SELECT department_id, salary, commission_pct,
            NVL2(commission_pct, salary * commission_pct, 100) "$100 or Comm."
FROM employees
WHERE department_id BETWEEN 60 AND 80
ORDER BY department_id;

-- [16] Use of the NULLIF function.
-- Return NULL if the manager_id is equal to 100
-- Note: that if manager_id is NULL, it will also return NULL
-- Use Query Result option to show the NULL value.
SELECT employee_id, manager_id, NULLIF(manager_id, 100)
FROM employees
WHERE employee_id LIKE '1%'
ORDER BY manager_id NULLS FIRST, employee_id;

-- [17] Change format of data values via TO_CHAR.
-- Example 1:
-- Exampe Other example: if you would like spaces in phone numbers
SELECT salary,
            TO_CHAR(salary, '$99,999') "Salary",
            commission_pct,
            TO_CHAR(commission_pct, '.99') "Commission",
            hire_date,
            TO_CHAR(hire_date, 'YYYY') "Year Hire"
FROM employees
WHERE department_id BETWEEN 60 AND 80;

-- [18] CASE expression:
-- Follows the SQL standard
-- Do not use DECODE function
SELECT last_name,
            TO_CHAR(salary, '$99,999') "Salary",
            CASE
              WHEN (salary >= 10000) THEN 'Level 5'
              WHEN (salary >= 8000) THEN 'Level 4'
              WHEN (salary >= 5000) THEN 'Level 3'
              WHEN (salary >= 2500) THEN 'Level 2'
              ELSE 'Level 1'
            END AS "Salary Level"
FROM employees
ORDER BY salary DESC;

-- Hint: Regular Expressions – can be used for finding regular patterns
```

```
-- such as telephone numbers etc. We will not cover this topic in the lab
-- but it is covered in Casteel (pp. 357-359).

-- Hint: We have not covered the syntax JOIN yet,
-- so do not worry if you do not understand
-- some of the examples in Casteel.

-----------------------------------------------------------------
-- End of Oracle Lab Demo Week 4.                    --
-----------------------------------------------------------------
```

# Week 5 Joining Table

```
-------------------------------------------------------------
-- Cartesian and Cross Joins                                --
-------------------------------------------------------------

-- Cartesian Join: Type of join which joins every row of
-- one table with every row of another table
-- ** Query 1 **
SELECT * FROM regions, countries;

--  Cartesian and cross joins produce the same query results
-- ** Query 2 **
SELECT * FROM regions CROSS JOIN countries;

-------------------------------------------------------------
-- Traditional Join and alias                               --
-------------------------------------------------------------

-- If the same column name occurs in both tables,
-- you must specify which table the column is referred to
-- with the WHERE clause, otherwise an error message
-- will appear
-- ** Query 3 **

SELECT last_name, department_id, department_name
FROM employees, departments
WHERE department_id = department_id; --why the error occurs here? Whose department_id?

-- To specify table names, type the table names after the FROM clause
-- ** Query 4 **
SELECT employees.last_name, employees.department_id, departments.department_name
FROM employees, departments
WHERE employees.department_id = departments.department_id;

-- Alternatively, you can use 'alias' to distinguish different tables.
-- It is a good practice to use alias to identify all the tables by assigning
-- a unique identifier. Use the identifier to specify column where the table
-- belongs to
-- ** Query 5 **
SELECT e.last_name, e.department_id, d.department_name
FROM  employees   e,  departments d
WHERE e.department_id = d.department_id;

-------------------------------------------------------------
-- Traditional Join Versus ANSI JOIN                        --
-------------------------------------------------------------

-- If you accidentally forget to enter the condition to test
-- for location_id, the query will still run when specifying
-- using the WHERE clause
-- You will get 2,438 rows instead of 106 rows.
-- ** Query 6 **
SELECT e.last_name, e.department_id, d.department_name, l.city
FROM  employees   e,
      departments d,
      locations   l
WHERE e.department_id = d.department_id;
-- Missing: AND d.location_id = l.location_id;

-- On the other hand, if you're using the JOIN clause then you
```

```
-- will need to specify a condition or else an error will occur
-- ** Query 7 **
SELECT e.last_name, e.department_id, d.department_name, l.city
FROM  employees e
JOIN  departments d
 ON   e.department_id = d.department_id
JOIN  locations l;
-- Missing: ON d.location_id = l.location_id

--Should be
SELECT e.last_name, e.department_id, d.department_name, l.city
FROM  employees e
JOIN  departments d
 ON   e.department_id = d.department_id
JOIN  locations l
ON     d.location_id = l.location_id;

-------------------------------------------------------------
-- NATURAL JOIN                                             --
-------------------------------------------------------------

-- Example of using NATURAL JOIN (default: using location_id)
-- SQL Developer will automatically assign the column used to join
-- This may not be the column you want, don't use this method
-- ** Query 8 **
SELECT department_name, location_id
FROM departments
NATURAL JOIN locations;

-- NATURAL JOIN uses manager_id instead of department_id to link
--   the two tables. (Note: manager_id in the employees table
--   comes before department_id) - SQL automatically chooses
--   manager_id for joining
-- ** Query 9 **
SELECT last_name, department_id, department_name
FROM  employees
NATURAL JOIN departments;

-------------------------------------------------------------
-- JOIN ... USING ...                                       --
-------------------------------------------------------------

-- Instead, you need to specify the column(s) you want to link
--     the tables:
--     JOIN ... USING ...
-- Column Name must be the same in both tables
-- Almost the same as query 9 but allows user to
-- choose the column - department_id, hence more results
-- ** Query 10 **
SELECT last_name, department_id, department_name
FROM  employees
JOIN departments
USING (department_id);




-- Using alias - example of selecting department_id = 90
-- ** Query 11 **
SELECT e.last_name, department_id, d.department_name
```

```
FROM  employees e
JOIN departments d
USING (department_id)
WHERE department_id = 90;

-------------------------------------------------------------
-- JOIN ... ON ...                                          --
-------------------------------------------------------------

-- Using JOIN ... ON ... - example of selecting department_id = 90
-- ** Query 12 **
SELECT e.last_name, e.department_id, d.department_name
FROM  employees e
JOIN  departments d
 ON   e.department_id = d.department_id
WHERE e.department_id = 90;

-- Using JOIN ... ON ... for three tables
-- ** Query 13 **
SELECT e.last_name, e.department_id, d.department_name,
  d.location_id, l.city
FROM  employees e
JOIN  departments d
 ON   e.department_id = d.department_id
JOIN  locations l
 ON   d.location_id = l.location_id
WHERE e.department_id = 90;

-- Using JOIN ... USING ... will give you the same results
-- ** Query 14 **
SELECT last_name, department_id, department_name,
  location_id, city
FROM  employees
JOIN  departments
USING (department_id)
JOIN  locations l
USING (location_id)
WHERE department_id = 90;

-------------------------------------------------------------
-- Self-Join: Joining the table back to itself              --
-------------------------------------------------------------

-- Self-Join: Find employee's manager
-- Note manager_id and employee_id columns are from the same table
-- ** Query 15 **
SELECT e.last_name "Employee", me.last_name "Manager"
FROM  employees e
JOIN  employees me
 ON   e.manager_id = me.employee_id
WHERE e.employee_id = 103;

-------------------------------------------------------------
-- Outer Joins                                              --
-------------------------------------------------------------

-- Normal SELECT will not retrieve employee King's record
--   because manager_id is NULL
-- ** Query 16 **
SELECT e.last_name "Employee", me.last_name "Manager"
FROM  employees e
```

```
JOIN  employees me
ON   e.manager_id = me.employee_id
WHERE e.employee_id <= 110
ORDER BY e.employee_id;

-- King does not have a manager, as there is NULL value in the column
-- ** Query 17 **
SELECT employee_id, last_name, manager_id
FROM employees
WHERE employee_id = 100;

 -- Find employee's manager using LEFT OUTER JOIN
 -- Think the LEFT OUTER JOIN as an 'optional link'
 -- ** Query 18 **
SELECT e.last_name "Employee", me.last_name "Manager"
FROM  employees e
LEFT OUTER JOIN  employees me
 ON   e.manager_id = me.employee_id
WHERE e.employee_id <= 110
ORDER BY e.employee_id;

-- Find employee's job history using RIGHT OUTER JOIN
-- In this case, all the records exist in both tables
-- ** Query 19 **
SELECT jh.employee_id, j.job_id, jh.department_id, jh.start_date, jh.end_date, j.job_title
from job_history jh
RIGHT OUTER JOIN jobs j
 ON  j.job_id = jh.job_id
WHERE jh.employee_id <= 110;

-- Find employee's job history using RIGHT OUTER JOIN
-- If someone has not changed job in the company, then there will be no
-- job history - i.e. picking up NULL values.
-- ** Query 20 **
SELECT e.employee_id, e.job_id "CurrentJob", jh.job_id "Past Job",
  jh.start_date, jh.end_date
FROM job_history jh
RIGHT OUTER JOIN employees e
 ON  e.employee_id = jh.employee_id
WHERE e.employee_id <= 105;

-------------------------------------------------------------------
-- Home activities  - Not covered in Lab                          --
-------------------------------------------------------------------

-- Using UNION Operator: Retrieve all rows and removes duplicates
-- Think of it as 'Big happy family'
-- Note: '...' replaces for column that does not exist in that table
-- ** Query 21 **
SELECT 'From employees', employee_id, last_name, job_id, department_id "Department"
FROM employees
UNION
SELECT 'From job_history', employee_id, '...', job_id, department_id
FROM job_history
ORDER BY employee_id;

-- Using UNION Operator (example of eliminate duplicate records)
-- Depending on number of columns you want to display
-- Remove two columns
-- ** Query 22 **
SELECT employee_id, last_name
```

```
FROM employees
UNION
SELECT employee_id, '...'
FROM job_history
ORDER BY employee_id;

-- INTERSECT Operator: Retrieve values that are in common across both tables
-- E.g. employees changed their jobs but
-- since then have gone back to one of their pervious jobs.
-- Minor error
-- Before
-- ** Query 23 **
SELECT employee_id, job_id, department_id, hire_date
FROM employees
WHERE employee_id = 200;

-- ** Query 24 **
SELECT employee_id, job_id, department_id, start_date, end_date
FROM job_history
WHERE employee_id = 200;

-- After
-- ** Query 25 **
SELECT employee_id, job_id
FROM EMPLOYEES
INTERSECT
SELECT employee_id, job_id
FROM JOB_HISTORY
ORDER BY EMPLOYEE_ID;

-- Using MINUS Operator: Retrieve all values in first query minus second query
-- Employees have not changed jobs since they started working for the company
-- ** Query 26 **
SELECT employee_id
FROM employees
MINUS
SELECT employee_id
FROM job_history
ORDER BY employee_id;

-- Advanced: If you want to list the names of the employees who
-- have not changed jobs since they started working for the company.
-- We will discuss more on sub-queries later in this course.
-- ** Query 27 **
SELECT e.employee_id,
  e.last_name,
  e.first_name,
  e.job_id
FROM employees e
WHERE e.employee_id IN
  (SELECT en.employee_id FROM employees en
   MINUS
   SELECT employee_id FROM job_history)
ORDER BY e.employee_id;

-------------------------------------------------------------
-- End of Oracle Lab Demo Week 5                            --
-------------------------------------------------------------
```

## Week 6

**Flexibility Week**

# Week 7 Aggregate Functions (GROUP BY)

```
-------------------------------------------------------------
-- Aggregate Functions (COUNT, MIN, MAX, SUM, AVG)
-------------------------------------------------------------

-- [1] Aggregate multiple rows together to retrieve:
-- Minimum, Maximum, Sum and Average.
-- Format the columns with TO_CHAR to make them more readable.
SELECT TO_CHAR(MIN(salary), '$99,999') "Minimum",
                TO_CHAR(MAX(salary), '$99,999') "Maximum",
                TO_CHAR(SUM(salary), '$999,999') "Sum",
                TO_CHAR(AVG(salary), '$99,999.99') "Average"
FROM employees;

-- [2] Find the number of managers in the company.
-- COUNT counts only non-NULL values.
SELECT COUNT(DISTINCT manager_id) AS "Manager"
FROM employees;

-- [3] Find how many employees who do not have a manager.
-- Count all where value IS NULL.
SELECT COUNT(*) "Nb Employees w/out Mgr"
FROM employees
WHERE manager_id IS NULL;

-- [4 ] What is the hiring date of the longest-serving employee of the company?
-- What is the hiring date of the most recent employee of the company?
-- You can use MIN and MAX, same as above.
SELECT MIN(hire_date) "Longest Serving",
           MAX(hire_date)  "Recent"
FROM employees;

-- [5] Average of the commissions paid for some departments.
-- Remember that you can replace NULL values with 0 via NVL.
SELECT COUNT(*),
  TO_CHAR((AVG(salary * NVL(commission_pct, 0))), '$99,999')
       "Average Commission Paid"
FROM employees
WHERE department_id IN (80,90,100,110);

-------------------------------------------------------------
-- GROUP BY versus ORDER BY
-------------------------------------------------------------

-- [7] Aggregate (MIN, MAX, AVG) for the different departments.
-- You can use GROUP BY to specific a particular column;
-- in this case department_id.
-- You can use ORDER BY to sort the results.
SELECT department_id,
  COUNT(*) "Number of employees",
  TO_CHAR(MIN(salary), '$99,999') "Minimum",
  TO_CHAR(MAX(salary), '$99,999') "Maximum",
  TO_CHAR(AVG(salary), '$99,999.99') "Average"
FROM employees
GROUP BY department_id
ORDER BY department_id
;
```

```
-- [8] Aggregrate (MIN, MAX, AVG) in total (not per department)
-- You can run a similar query, just without GROUPBY, same as above.
SELECT 'Total: ',
            COUNT(*) "Number of employees",
            TO_CHAR(MIN(salary), '$99,999') "Minimum",
            TO_CHAR(MAX(salary), '$99,999') "Maximum",
            TO_CHAR(AVG(salary), '$99,999.99') "Average"
FROM employees;

-- [9] Aggregate for a certain set of departments.
-- You can combine WHERE/IN with GROUP BY to specify a set of values.
-- Remember that GROUP BY is to group aggregate functions.
-- Remember that ORDER BY is to sort results.
SELECT department_id,
  COUNT(*) "Number of employees",
  TO_CHAR(MIN(salary), '$99,999') "Minimum",
  TO_CHAR(MAX(salary), '$99,999') "Maximum",
  TO_CHAR(AVG(salary), '$99,999.99') "Average"
FROM employees
WHERE department_id IN (80,90,100,110)
GROUP BY department_id
ORDER BY department_id;

-----------------------------------------------------------------
-- WHERE versus HAVING
-----------------------------------------------------------------
-- [10] Aggregate for a certain set of departments AND
-- ...show only those where the average salary per department is >= 10,000.
-- You need to use the phrase HAVING for conditions based on grouped aggregates.
-- WHERE is used for conditions regarding what data to get from the DB.
-- HAVING is used later, for conditions regarding the newly created aggregates.
SELECT department_id,
            COUNT(*) "Number of employees",
            TO_CHAR(MIN(salary), '$99,999') "Minimum",
            TO_CHAR(MAX(salary), '$99,999') "Maximum",
            TO_CHAR(AVG(salary), '$99,999.99') "Average"
FROM employees
WHERE department_id IN (80,90,100,110)
GROUP BY department_id
HAVING AVG(salary) >= 10000
ORDER BY department_id;

-- [11] Aggregate for a certain set of departments and
-- ...include only those employees with an individual salary >= 10,000.
-- Note: This illustrates the difference between WHERE versus HAVING!
SELECT department_id,
            COUNT(*) "Number of employees",
            TO_CHAR(MIN(salary), '$99,999') "Minimum",
            TO_CHAR(MAX(salary), '$99,999') "Maximum",
            TO_CHAR(AVG(salary), '$99,999.99') "Average"
FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (80,90,100,110) AND  salary >= 10000
GROUP BY department_id
ORDER BY department_id;



-----------------------------------------------------------------
```

```
-- Complex Queries with GROUP BY, ORDER BY and HAVING
-------------------------------------------------------------------

-- [12] Aggregate per department, and display department names.
-- To get the department name, we need data from the departments table.
-- You can combine GROUP BY with joins.
SELECT department_id, department_name,
            COUNT(*) "Number of employees",
            TO_CHAR(MIN(salary), '$99,999') "Minimum",
            TO_CHAR(MAX(salary), '$99,999') "Maximum",
            TO_CHAR(AVG(salary), '$99,999.99') "Average"
FROM employees
JOIN departments
 USING (department_id)
GROUP BY department_id, department_name
HAVING AVG(salary) >= 5000
ORDER BY department_id;

-------------------------------------------------------------------
-- End of Oracle Lab Week 07            Demo Script
-------------------------------------------------------------------
```

aggregate & order by –no complex queries

# Week 8 Data Manipulation Language (DML)

```
-- Tidy database up for lab demo...
DELETE FROM countries WHERE country_id = 'NZ';
DELETE FROM countries WHERE country_id = 'FJ';
DELETE FROM countries WHERE country_id = 'BI';
DELETE FROM regions WHERE region_id = 5;
DELETE FROM regions WHERE region_id = 6;

----------------------------------------------------------------
-- Inserting a new row to a table
----------------------------------------------------------------

-- [1] Check countries table
SELECT * FROM countries WHERE region_id = 3;

-- [2] Insert "New Zealand" by explicitely stating columns/attributes
INSERT INTO countries (country_id, country_name, region_id)
  VALUES ('NZ', 'New Zealand', 3);

-- [3] Check if Fiji was inserted?
SELECT * FROM countries WHERE region_id = 3;

-- [4] Insert "Fiji" by implicately using order of columns
INSERT INTO countries
  VALUES ('FJ', 'Fiji', 3);

-- [5] Check if Fiji was inserted?
SELECT * FROM countries WHERE region_id = 3;

----------------------------------------------------------------
-- COMMIT versus ROLLBACK
--
-- By default, records are not saved automatically to the database.
-- Execute COMMIT to save the records permanently in the database.
-- Alternatively, if you do not want to save, excute ROLLBACK.
----------------------------------------------------------------

-- [6] Execute the COMMIT command to save the records.
COMMIT;

-- [7] You can INSERT this record because 'nz' is different from 'NZ'.
INSERT INTO countries
  values ('nz', 'New Zealand', 3);

-- [8] See if it was inserted.
SELECT * FROM countries WHERE region_id = 3;

-- [9] Execute ROLLBACK command not to save the record
ROLLBACK;

-- [10] Is it still there?
SELECT * FROM countries WHERE region_id = 3;
```



---

```
-- Constraint Violation
--
-- Remember: Contraints relate to PK, FK, UNIQUE...
---------------------------------------------------------------

-- [11] Cannot INSERT the same record again                    PK also have UNIQUE
--  Violate the primary key constraint
INSERT INTO countries
  VALUES ('NX', 'New Zealand', 3);

-- [12] Check Regions: There is no region_id 5.
SELECT * FROM regions;

-- [13] Attempt to insert the record will violate the FK constraint.
INSERT INTO countries
  VALUES ('BI', 'Brunei', 5);

-- [14] Create a record for region_id 5 (to overcome the FK constraint)
INSERT INTO regions (region_id, region_name)
  VALUES (5, 'Asiapacific');

SELECT * FROM regions;

-- [15] Now, you can add the record
INSERT INTO countries
  VALUES ('BI', 'Brunei', 5);

-- [16###] You can retrieve the record now
SELECT * FROM countries WHERE region_id = 5;

-- [17] A NULL value will be initialised if a column is not
--   specified.
INSERT INTO regions (region_id)
  VALUES (6);

-- [18]
SELECT * FROM regions;
-- [19]
ROLLBACK;

---------------------------------------------------------------
-- Insert a new row from an existing table                   --
---------------------------------------------------------------
-- [20]For demonstration purpose:
--  Recreate employees_history table and drop the table
--  if it exists
DROP TABLE employees_history;                                 delete table =drop

-- [21] Duplicating a table using EMPLOYEES table data
CREATE TABLE employees_history
  AS (SELECT * FROM employees);

-- [22] Add Date column to table
ALTER TABLE employees_history
  ADD (modify_date DATE);

-- [23]  Drop Email column
ALTER TABLE employees_history
  DROP COLUMN email;

-- [24] Delete the existing employee history record for employee 206
```

```
DELETE FROM employees_history WHERE employee_id = 206;

-- [25] There is an existing employee history record for employee 206
-- Take record from Employees table and insert into Employee History table
-- Assign department_id to 80 and modify_date to today's date
INSERT INTO employees_history
        (employee_id, first_name, last_name, hire_date, job_id,
         manager_id, department_id, modify_date)
  SELECT employee_id, first_name, last_name, hire_date, job_id,
         manager_id, 80, SYSDATE
  FROM employees
  WHERE employee_id = 206;

-- [26] View the changes
SELECT employee_id, first_name, last_name,
  department_id, modify_date
FROM employees_history
WHERE employee_id = 206;

-- [27] Undo all the changes made in employees_history table
ROLLBACK;

-----------------------------------------------------------------
-- Updating a row of a table                                   --
-----------------------------------------------------------------

-- [28] Update data value of a row
SELECT * FROM countries
  WHERE country_id = 'NZ';

-- [29]
UPDATE countries
  SET country_name = 'All Blacks'
  WHERE country_id = 'NZ';

-- [30]
SELECT * FROM countries
  WHERE country_id = 'NZ';

-- [31] Without WHERE clause, it will update all the records
--   in a table! Thus, be careful!
UPDATE countries
  SET country_name = 'New Zealand';

-- [32] See how all records have changed.
SELECT * FROM countries WHERE region_id = 3;

-- [33] Rollback changes.
ROLLBACK;

-----------------------------------------------------------------
-- Using Subsitution Variables (&)                             --
-----------------------------------------------------------------

-- [34] You can use subsitution variables instead of having
-- fixed data values – allowing to enter them "on the go"
-- Enter 'All Blacks Rugby Team' for the Country Name
-- Enter 'NZ' for the Country Id
UPDATE countries
  SET country_name = '&Country_name'
  WHERE country_id = '&Country_id';
```

```
-- [35]
SELECT * FROM countries
  WHERE country_id = 'NZ';

ROLLBACK;

-----------------------------------------------------------------
-- Deleting a row from a table                                 --
-----------------------------------------------------------------

-- [36] Is Fiji there?
SELECT * FROM countries
  WHERE country_id = 'FJ';

-- [37] Delete Fiji (row).
DELETE FROM countries
  WHERE country_id = 'FJ';

-- [38] Is Fiji there?
SELECT * FROM countries
  WHERE country_id = 'FJ';

-- [39]
COMMIT;

-----------------------------------------------------------------
-- Create Sequence Number                                      --
-----------------------------------------------------------------

-- [40] Tidy up - drop sequence if exists
DROP SEQUENCE region_id_seq;

-- [41] Create a new sequence for region_id (auto-incrementing)
CREATE SEQUENCE region_id_seq
  INCREMENT BY 1
  START WITH 20
  MAXVALUE 9999
  NOCACHE
  NOCYCLE;

-- [42] We are inserting the next value
-- Notice the automatic numbering
INSERT INTO regions (region_id, region_name)
   VALUES (region_id_seq.NEXTVAL, 'ZZZZZZZZZZZ');

SELECT * FROM regions;


-- [53] Value for the next sequence number of region_id
SELECT region_id_seq.NEXTVAL
  FROM dual;
```

```
-----------------------------------------------------------------
-- Adding Constraints to a table                              --
-----------------------------------------------------------------
-- [27] Add Primary and Foreign Keys, and Constraints to a table
-- Create a table and define constraints for a table.
-- Remember:  Number (4,0) means four digits, no decimal points
--            Number (4,2) means four digits including two decimal points
CREATE TABLE locations_temp
  (
    location_id    NUMBER(4,0)
          CONSTRAINT loc_temp_loc_id_pk
          PRIMARY KEY,
    street_address VARCHAR2(40),
    country_id     CHAR(2),
          CONSTRAINT loc_temp_c_id_fk
          FOREIGN KEY (country_id)
          REFERENCES countries (country_id),
    active_flag CHARACTER(1) DEFAULT 'Y',
    modify_date DATE DEFAULT SYSDATE
  );

-- [28] Creating a composite primary key for a table
-- DROP TABLE global_region_temp;
CREATE TABLE global_region_temp
  (
    manager_id     NUMBER(6,0),
    region_id      NUMBER(2,0),
    job_id         VARCHAR2(10),
   CONSTRAINT global_region_temp_mgr_reg_pk
     PRIMARY KEY (manager_id, region_id),
   CONSTRAINT global_region_temp_mgr_id_fk
     FOREIGN KEY (manager_id)
     REFERENCES employees (employee_id),
   CONSTRAINT global_region_temp_reg_id_fk
     FOREIGN KEY (region_id)
     REFERENCES regions (region_id)
  );

-- [29] Add a new column location_mnemonic,
-- and add a unique constraint for this column
-- What is UNIQUE?
-- Hint: UNIQUE = DISTINCT = every values need to be unique (in this column)
ALTER TABLE locations_temp
  ADD location_mnemonic CHAR(2)
    CONSTRAINT loc_temp_l_mn_uk
    UNIQUE;

-- [30] For Demo - delete the column first from the table:
ALTER TABLE locations_temp
   DROP COLUMN location_mnemonic;

-- [31] Alternatively, you can add the new column location_mnemonic
ALTER TABLE locations_temp
  ADD location_mnemonic CHAR(2);

-- [32] Then add another constraint separately
ALTER TABLE locations_temp
  ADD CONSTRAINT loc_temp_l_mn_uk UNIQUE (location_mnemonic);

-- [33] Insert records to test the UNIQUE constraint
```

```
INSERT INTO LOCATIONS_TEMP (location_id, location_mnemonic) VALUES (95, 'L5');
INSERT INTO LOCATIONS_TEMP (location_id, location_mnemonic) VALUES (96, 'L6');
INSERT INTO LOCATIONS_TEMP (location_id) VALUES (97);
INSERT INTO LOCATIONS_TEMP (location_id, location_mnemonic) VALUES (98, NULL);
SELECT location_id, active_flag, modify_date, location_mnemonic FROM locations_temp;

-- [34] Will get an error, as noted from the error message
-- the unique constraint for location_mnemonic column is being violated
-- which was set by Query 29
INSERT INTO LOCATIONS_TEMP (location_id, location_mnemonic) VALUES (99, 'L5');

-- [35] Modify the column first_name so it must have a value,
--   i.e. no NULL value is accepted.
ALTER TABLE employees_hist_2011
  MODIFY (first_name CONSTRAINT first_name_nn NOT NULL);

-- [36] To check to ensure that salary is less than $50,000 per month.
--ALTER TABLE employees_hist_2011 DROP CONSTRAINT emp_sal_ck;
ALTER TABLE employees_hist_2011
  ADD CONSTRAINT emp_sal_ck CHECK (salary <= 50000);

-- [37] Delete CONSTRAINT emp_sal_ck if exists in the database
ALTER TABLE employees_hist_2011 DROP CONSTRAINT emp_sal_ck;

-- [38] Check constraints
-- Note: you need to use single quotes ('  ') for the name of
--   the tables
-- Note: user_constraints table is another system table
SELECT constraint_name,
  constraint_type,
  search_condition,
  r_constraint_name
FROM user_constraints
WHERE table_name = 'EMPLOYEES_HIST_2011';

-- [39###] Delete the Primary Key of employees_hist_2011
-- An error message is raised as the primary key does not exist (already dropped)
ALTER TABLE employees_hist_2011
  DROP PRIMARY KEY;

-- [40] Create a Primary key for employees_hist_2011 table
ALTER TABLE employees_hist_2011
  ADD CONSTRAINT loc_temp_emp_id_pk PRIMARY KEY (employee_id);

-- [41] Delete Primary key in employees_hist_2011 table
ALTER TABLE employees_hist_2011
  DROP PRIMARY KEY;

-- [42] Delete a constraint
ALTER TABLE employees_hist_2011
  DROP CONSTRAINT first_name_nn;

-- [43] The two constraints should have been dropped
SELECT constraint_name,
            constraint_type,
            search_condition,
            R_constraint_name
FROM user_constraints
WHERE table_name = 'EMPLOYEES_HIST_2011';

-- Hint: If you would like to reset / clean up the database:
```

```
-- DROP TABLE departments_history;
-- DROP TABLE employees_history;
-- DROP TABLE employees_hist_2011;
-- DROP TABLE locations_temp;
-- DROP TABLE global_region_temp;

---------------------------------------------------------------
-- End of Oracle Lab Demo Week 8                    --
---------------------------------------------------------------
```

## Week 9 Subqueries and Merge Statements

```
---------------------------------------------------------------
-- A subquery is a nested query -
--   one complete query inside another query.
-- This means you will have one SELECT statement in
--   another SELECT statement.
---------------------------------------------------------------
--  How to do subqueries:

--   (1) Inner query is executed first
--   (2) The result is then passed from the inner query
--       to the parent query
--   (3) Parent query is executed
--
---------------------------------------------------------------
-- Let us try to find all employee who have a higher pay then the
-- highest paid employee from department 80

-- [1] First query finds the highest paid employee from department 80
SELECT MAX(em.salary)
FROM employees em
WHERE em.department_id = 80;

-- [2] Second query finds the salary of all employees
SELECT e.last_name,
               e.department_id,
               E.salary
FROM employees e;

-- [3] Combine the query to find employees who are paid more
-- than the highest paid employee in department 80, notice the
-- WHERE clause
SELECT e.last_name,
  e.department_id,
  e.salary
FROM employees e
WHERE e.salary >
  (SELECT MAX(em.salary)
    FROM employees em
    WHERE em.department_id = 80
  );

---------------------------------------------------------------
-- A single Row Subquery in a WHERE clause
-- 'Single row' means that we are only returning a single value (from one row)
```

```
-- to the query.
-- We can use single row operators like <,= etc...
-------------------------------------------------------------

-- [4] Notice this query returns more than one record

SELECT em.salary
  FROM employees em
  WHERE em.department_id = 80;




-- [5] As the nested query returns more than one row (all employeee
-- salaries in department 80) we get an error:
-- "single-row subquery returns more than one row"
SELECT e.last_name,
  e.department_id,
  e.salary
FROM employees e
WHERE e.salary >
  (SELECT em.salary
    FROM employees em
    WHERE em.department_id = 80
  );

-------------------------------------------------------------
-- Subquery in HAVING clause                                --
-------------------------------------------------------------
-- [6] Find all departments in which the average salary is greater than
-- average salary of the employees in department 80

SELECT e.department_id,
  AVG(e.salary)
FROM employees e
GROUP BY e.department_id
HAVING AVG(e.salary) >
  (SELECT AVG(em.salary)
    FROM employees em
    WHERE em.department_id = 80
  );

-------------------------------------------------------------
-- Subquery in FROM clause                                  --
-------------------------------------------------------------
-- [7] This query will list the last names, departments, and salaries of employees
-- who have a salary above the average salary of the department

SELECT last_name,
  department_id,
  to_char(salary,'99,999') "Salary"
FROM employees em JOIN
  (SELECT em.department_id, AVG(em.salary) avg_salary
    FROM employees em
    GROUP BY em.department_id) EAVG
   USING (department_id)
WHERE em.salary >= EAVG.avg_salary
ORDER BY department_id,
  last_name;

-- We are using a JOIN statement to join EAVG; a temporary table we have
```

```
-- created using a subquery. We create EAVG to determine the average salary
-- (avg_salary) grouped by the department_ID to get the average salary
-- per department.



------------------------------------------------------------
-- Common operators used in subquery statements  --
------------------------------------------------------------
-- Demonstrating ALL versus ANY

-- [9]First we find the salary of the two employees in
-- department 110 ($8,300 and $12,008)
SELECT to_char(salary,'99,999') "Salary"
FROM employees
WHERE department_id = 110;

-- [10] The subquery (inner query, in brackets) pulls the employee salary
-- information for department 110. The outer query is saying
-- "only return results where salary is greater than ALL of the values"
-- (which means they must be higher than the highest value, $12,008)
-- returned by the subquery.

SELECT e.last_name,
  e.first_name,
  e.department_id,
  TO_CHAR(e.salary,'99,999') "Salary"
FROM employees e
WHERE e.salary > ALL
  (SELECT em.salary
   FROM employees em
   WHERE em.department_id = 110);

-- [11] The subquery (inner query, in brackets) pulls the employee salary
-- information for department 110. The outer query is saying
-- "only return results where salary is greater than ANY of the values"
-- (which means they must be higher than the lowest value, $8,000)
-- returned by the subquery.

SELECT e.last_name,
  e.first_name,
  e.department_id,
  TO_CHAR(e.salary,'99,999') "Salary"
FROM employees e
WHERE e.salary > ANY
  (SELECT em.salary
   FROM employees em
   WHERE em.department_id = 110);

-- [12] Here we can see the average salary for all departments
SELECT department_id "Department",
  to_char(AVG(salary),'99,999') "Average Salary"
FROM employees
GROUP BY department_id
ORDER BY department_id;
```

```
-- [13] We want to identify employees whose salary is exactly the same as
-- the department's average salary.
-- To do this using a subquery, We would use the IN operator.
-- This operator means that the outer query (the query not in brackets)
-- will only return values which are returned by the subquery (the bracketed
-- query).



SELECT e.last_name,
               e.first_name,
               e.department_id,
               TO_CHAR(e.salary,'99,999') "Salary"
FROM employees e
WHERE (e.department_id, e.salary) IN
  (SELECT em.department_id, AVG(em.salary)
   FROM employees em
   GROUP BY em.department_id)
ORDER BY e.department_id,
  e.last_name;

-- The query is saying "Select all records from the employees table
-- where the value pair (e.department_id, e.salary) are IN the records
-- returned by the query finding the department ID and
-- average salary for the groups in the subquery"

-----------------------------------------------------------------
  -- MERGE Statements                                         --
-----------------------------------------------------------------

-- Merge statements conditonally update tables, e.g., they update based on
-- specified conditions depending on whether their data matches that
-- of another table

-- [17] Tidy up by deleting tables
DROP TABLE regions_temp_one;
DROP TABLE regions_temp_two;

-- [18] Create two temporary tables copied from regions table
CREATE TABLE regions_temp_one AS (SELECT * FROM regions);
CREATE TABLE regions_temp_two AS (SELECT * FROM regions);

-- [19] Show Records
SELECT * FROM regions_temp_one;
SELECT * FROM regions_temp_two;

-- [20] Using DML we add a new record to regions_temp_one table
INSERT INTO REGIONS_TEMP_ONE VALUES (5, 'New World');

-- [21] Here we update the Region Name values in regions_temp_two table so that
-- they are now different
UPDATE REGIONS_TEMP_TWO SET REGION_NAME = 'Table Two 1'
                        WHERE REGION_ID = 1;
UPDATE REGIONS_TEMP_TWO SET REGION_NAME = 'Table Two 2'
                        WHERE REGION_ID = 2;
UPDATE REGIONS_TEMP_TWO SET REGION_NAME = 'Table Two 3'
                        WHERE REGION_ID = 3;
UPDATE REGIONS_TEMP_TWO SET REGION_NAME = 'Table Two 4'
                        WHERE REGION_ID = 4;
```

```
-- [22] These queries show that there are different values in both tables
SELECT * FROM REGIONS_TEMP_ONE;
SELECT * FROM REGIONS_TEMP_TWO;




-- [23]Here we use merge to merge the regions temp tables together in
-- regions temp two
MERGE INTO REGIONS_TEMP_TWO T
  USING REGIONS_TEMP_ONE O
    ON (T.REGION_ID = O.REGION_ID)
  WHEN MATCHED THEN
    UPDATE SET T.REGION_NAME = O.REGION_NAME
  WHEN NOT MATCHED THEN
    INSERT (REGION_ID, REGION_NAME)
      VALUES (O.REGION_ID, O.REGION_NAME);

-- Our statement has updated regions temp two so if a condition is met
-- Temp two ID = Temp one ID (T.REGION_ID = O.REGION_ID), the
-- REGION_NAME column is updated to that of table one.
-- If a match is not found then it adds the row from table one.

-- [24] As a result, the values in both tables are the same
SELECT * FROM REGIONS_TEMP_ONE;
SELECT * FROM REGIONS_TEMP_TWO;

------------------------------------------------------------
-- End of Oracle Lab Demo Week 9                          --
------------------------------------------------------------
```

# Week 10 Views

```
-------------------------------------------------------------
-- Oracle Lab Week 10 Demo Script                           --
-------------------------------------------------------------

-------------------------------------------------------------
-- Views
-- Dropping views
-- Querying views
-- Creating a complex view
-- Adding contraints to views
-------------------------------------------------------------

ALTER SESSION SET nls_date_format = 'DD-MON-RR';

-- [1] Tidy Up by dropping views
DROP VIEW employees_view;
DROP VIEW emp_salary_view;
DROP TABLE check_demo;
DROP VIEW check_demo_view;

-- [2] This is a complex view (virtual table) already created.
-- The view shows information about employees.
-- The information is pulled from multiple tables.
SELECT *
FROM emp_details_view;

-- [3] Here is a query which identifies employees whose salary is
-- exactly the same as the respective department's average salary.
SELECT e.last_name,
    e.first_name,
    e.department_id,
    TO_CHAR(e.salary,'99,999') "Salary"
  FROM employees e
  WHERE (e.department_id, e.salary) IN
    (SELECT em.department_id, AVG(em.salary)
     FROM employees em
     GROUP BY em.department_id)
  ORDER BY
    e.department_id,
    e.last_name;

-- [4] Instead of writing a query to get this information, we can create
-- a view  with the same information.
CREATE OR REPLACE VIEW emp_salary_view
  AS SELECT e.last_name,
    e.first_name,
    e.department_id,
    TO_CHAR(e.salary,'99,999') "Salary"
  FROM employees e
  WHERE (e.department_id, e.salary) IN
    (SELECT em.department_id, AVG(em.salary)
     FROM employees em
     GROUP BY em.department_id)
  ORDER BY
    e.department_id,
    e.last_name;

-- [5] Describe the newly created view.
DESCRIBE emp_salary_view;
```

```
-- [6] Show the contents of the view.
SELECT * FROM emp_salary_view;

-- [7] We can use the view like a table.
-- For example, use the WHERE clause to select particular data.
SELECT * FROM emp_salary_view
WHERE department_id = 70;

-- [8] Here we are going to create a demo table and add values.
-- Note the values in the last column.
CREATE TABLE check_demo (
  C1 varchar(3),
  C2 varchar(3),
  C3 varchar(10)
);

INSERT INTO check_demo VALUES ('a','b','Employee');
INSERT INTO check_demo VALUES ('d','e','Employee');
INSERT INTO check_demo VALUES ('f','g','CEO');

-- [9] Show the contents of the demo table.
SELECT * FROM check_demo;

-- [10] Create a view that only selects rows from the demo table
-- that have a value of 'Employee' for the attribute C3.
CREATE OR REPLACE VIEW check_demo_view
  AS SELECT C1, C2, C3
  FROM check_demo
  WHERE C3 = 'Employee';

-- [11] Show the contents of the view (based on the demo table).
-- Note that only 'Employee' values are in the view.
SELECT * FROM check_demo_view;

-- [12] We are able to still insert C3 = "CEO" values.
INSERT INTO check_demo_view VALUES ('e','f','CEO');

-- [13] Show the demo table (the "base table" of the view).
-- We have been able to modify our table with the query.
SELECT * FROM check_demo;

-- [14] Show the view.
-- As before, only C3 = "Employee" rows are returne.
SELECT * FROM check_demo_view;

-- [15] To prevent invalid values being inserted into the table,
-- add the WITH CHECK OPTION CONSTRAINT.
CREATE OR REPLACE VIEW check_demo_view
  AS SELECT C1, C2, C3
  FROM check_demo
  WHERE C3 = 'Employee'
  WITH CHECK OPTION CONSTRAINT check1;

-- [16] Error will be triggered when attempting to add a "CEO" value.
INSERT INTO check_demo_view VALUES ('e','f','CEO');

-- [17] No error will be triggered when adding a "Employee" value.
INSERT INTO check_demo_view VALUES ('e','f','Employee');

----------------------------------------------------------------
```

-- End of Oracle Lab Demo Week 10 --
---