Laboratory Worksheet Two

NOTE: *You should complete Worksheet One before beginning Worksheet Two*

SQL RETRIEVAL EXERCISES

Use database file **projemp** on SQL Server - Execute **USE projemp;** in a New Query window

projemp relational schema:

DEPT (dno, dname, location)
EMP (eno, ename, salary, age, supno, dno*)
WORKS (eno*, pno*, role, duration)
PROJ (pno, pname, ptype, budget)

Attributes eno, dno and pno are all text values (e.g., e1, e2, etc). All other attributes are text except salary [decimal(8,2)], age [tinyint], duration [tinyint] and budget [decimal(9,2)]

To Do: Explore the structure of the projemp database by expanding the tables in the Object Explorer. Pay particular attention to the data types used in the table attributes and the foreign keys used to establish relationships between the tables.

NOTE: SQL Server is not case sensitive and does not require that each clause in a query should start on a new line. However, in this module we will use the following convention in constructing SQL queries:

- SQL keywords should be in caps (e.g., **USE**, **SELECT**, etc)
- Table names should be in caps (e.g., **DEPT**, **EMP**, etc)
- Caps should not be used in database file names (e.g., projemp) or attribute names (e.g., dno, pname, etc)
- Each new clause in an SQL statement should begin on a new line and indentation should be used as appropriate to aid readability (see examples of nested queries towards the end of this worksheet).

Click on ***New Query*** on the top menu to create a new query.

```
USE projemp;
GO
```

Placed before each query ensures the correct database is current

Queries can be saved as a query file using ***File|Save SQLQueryX.sql*** (where X is the number of the query), allowing the name to be changed to add some meaning if required and the saving location to be chosen. – or alternatively cut and paste into a Word file.

Query files can be read with Notepad or Wordpad and rerun using ***File|Open|File…*** to select and open as a query window.

Enter the following Queries in a new query window and run (!Execute)

You can display all the records in a table (all columns, all rows) using:

```
SELECT *
FROM EMP;
```

NOTE: * is shorthand for all attributes
The Messages tab gives the number of records returned.

Examine the result table**.**

1. Get the employee names and ages of those employees in department number 'd1'.

```
SELECT ename, age
FROM EMP
WHERE dno ='d1';          9 records
```

2. Get the employee number and name of all employees aged above 30 in department number ‘d2’

```
SELECT eno, ename
FROM EMP
WHERE age > 30
AND dno ='d2';            2 records
```

NOTE: Values of a character datatype attribute (e.g., dno) must be shown in inverted commas (single not double), whereas numeric datatype attributes are not.

3. Get the names of employees earning less than £20,000 or more than £30,000.

```
SELECT ename
FROM EMP
WHERE salary <20000
OR salary > 30000;        14 records
```

4. How many employees are in department number 'd3'?

```
SELECT COUNT(eno)         (should count using PK or *)
FROM EMP
WHERE dno = 'd3';         7 records
```

Or alternatively

```
SELECT COUNT(*)
FROM EMP
WHERE dno = 'd3';         NOTE: * can be used in COUNT( ) only
```

Note: The result table has no column name – to replace this with a more meaningful value use the subclause **AS <columnname>** where columnname contains *NO* spaces

e.g.
```
SELECT COUNT(eno) AS Num_Employees
FROM EMP
WHERE dno = 'd3';
```

5. Get the total salary of all employees in department number ‘d1’ (adds up all salaries in department ‘d1’)

```
SELECT SUM(salary)
FROM EMP
WHERE dno = 'd1';         314000.00
```

6. Get the employee names and their salaries in ascending order of salary.

```
SELECT ename, salary
FROM EMP
ORDER BY salary ASC;
```

NOTE: **ASC** is the default value if **ASC** or **DESC** is not entered; **DESC** sorts in descending order

Try changing **ASC** to **DESC** and changing the **ORDER BY** attribute to ename

7. Get the employees who have salaries of £18,000 or £20,000 (Use of the **IN** predicate)

```
SELECT ename
FROM EMP
WHERE salary IN(18000, 20000);
```

Or alternatively:

```
SELECT ename
FROM EMP
WHERE salary = 18000
OR salary = 20000;
```

2 records

NOW ANSWER THE FOLLOWING QUERIES AND SAVE THEM

a. Get the names of employees who earn a salary above £25,000.
There should be 11 records.

b. Get the names of employees who earn a salary above £25,000 and are in department number 'd1'.
There should be 6 records.

c. What is the average age of employees in department number 'd2'?
Should be 33.

d. List the names, salaries and ages of all employees in department number 'd1' in descending order of age (oldest to youngest).
There should be 9 records.

e. How many employees have a salary greater than £20,000 and are in department number 'd1?'
Answer should be 7

USE OF GROUP BY AND HAVING STATEMENTS

GROUP BY clauses group the data *BY VALUE* using a field that *MUST* therefore have duplicate values (such as the foreign key dno) so that at least some groups have multiple records (e.g. department 'd1' employees, department 'd2' employees, etc).
You can then apply an aggregate function to each group.

8. Get the average salary in each department (or by department)

```
SELECT dno, AVG(salary)
FROM EMP
GROUP BY dno;
```

NOTE: The attribute (dno) in the **SELECT** clause *MUST* agree with the **GROUP BY** attribute (dno).

The **HAVING** clause can *ONLY* be used when a **GROUP BY** clause is present and is used to choose between groups.

9. How many employees work on each project (using **WORKS** table)?

```
SELECT pno, COUNT(eno)
FROM WORKS
GROUP BY pno;
```

(p13 - 8; p15 - 5; p19 - 10; p23 - 9; p26 - 7)

10. Get the departments having an average salary below £30,000

```
SELECT dno
FROM EMP
GROUP BY dno
HAVING AVG(salary) < 30000;
```

Note: Only departments d2 and d3 are selected because their average salary is less than £30,000.

HAVING clauses choose between groups and *MUST* conform to the syntax:

Aggregate function	*compared with*	*a value*
(count( ), avg( ), etc)	(=, <, >, etc)	(number)

11. Get the projects having less than 9 employees working on them (i.e. **HAVING** a **COUNT** of employees less than 9)

```
SELECT pno
FROM WORKS
GROUP BY pno
HAVING COUNT(eno) < 9;
```
(p13, p15 and p26)

12. Get a unique list of department numbers (Use of **DISTINCT** predicate - eliminates duplicate records from result)

Compare the results of:

```
SELECT dno
FROM EMP;
```
(21 records including many duplicates)

With the results of:

```
SELECT DISTINCT dno
FROM EMP;
```
(3 records - no duplicate values)

NOW ANSWER THESE QUERIES AND SAVE THEM

f. How many employees are there in each department? (See query 8)
Answer should be d1 - 9; d2 - 5; d3 – 7.

g. List the departments (by department number) that have a total salary of more than £300,000. (see query 9)
Answer: department 'd1'.

h. How many employees earn £20,000 or £25,000?
Answer: 2.

INSERT, DELETE and UPDATE

General **INSERT** syntax

```
INSERT INTO tablename
VALUES (record values);
```

General **DELETE** syntax:

```
DELETE FROM tablename
WHERE selection condition ;
```

General **UPDATE** syntax

UPDATE *tablename*
SET *update expression*
WHERE *selection condition*;

Now try the following, examining the **EMP** table *BEFORE* and *AFTER* each query.
(You may need to tell the system that you are using the **projemp** database (**USE projemp;**) if the error *Invalid object name 'EMP'* appears)

13. Insert a new record with eno 'e51', ename 'mallon', salary 26000.00 , age 32, supno 'e17' and dno 'd2'

```
INSERT INTO EMP
VALUES('e51', 'mallon', 26000.00, 32, 'e17', 'd2');
```

NOTE: The full **INSERT** command also specifies which columns are being used – can be omitted if a complete record is being added

```
INSERT INTO EMP (eno, ename, salary, age, supno, dno)
VALUES('e51', 'mallon', 26000.00, 32, 'e17', 'd2');
```

Check the database to see if this record has been entered.

14. Delete the employee with an employee number of 'e51'

```
DELETE FROM EMP
WHERE eno = 'e51';
```

Check the database to see if this record has been deleted

15. Using similar SQL code, try to delete the employee with an employee number of 'e19'

You should find that an error is generated. Study the error message and determine why the deletion operation is prevented. Ask a demonstrator if you are unsure.

16. Change the budget of project number 'p13' to £650,000 (is £520,000 at present)

```
UPDATE PROJ
SET budget= 650000
WHERE pno = 'p13';
```

Check the database to see if this record has been changed.

NOW TRY THESE QUERIES, SAVE THEM AND CHECK THE DATABASE

i. Change the employee named 'oliver' to 'pearson'.

j. Add a new record with the following values:
eno 'e60', ename 'young', salary 60000.00, age 51, supno 'e1', dno 'd1'

k. Now delete this record using the SQL DELETE command.

RETRIEVAL QUERIES USING TWO OR MORE TABLES (JOINS)

To access attributes from different tables (either attributes required for result table or attributes used in selection conditions):

- examine the relational schema to see which tables are involved

- add these tables (and any intermediate tables used to link these tables) into the **FROM** clause
- Identify the join attributes (usually primary/foreign key matches) required to join the tables and add a **WHERE/AND** clause for every join *OR* use the subsidiary **ON** clause.

NOTE: All attributes involved in the query *CAN* be specified with the table they come from (**Table.Attribute**) but only those duplicate attribute names repeated in more than one query table (i.e. PF/FK attributes) *MUST* be specified like this.

Now, try the following queries:

17. Get a list of project names with the employee numbers of the employees working on them.

pname is from **PROJ**, eno is from **WORKS** (could use **EMP**, but **WORKS** is closer to **PROJ**)
pno is the primary/foreign key match between **PROJ** and **WORKS**

```
SELECT pname, eno
FROM PROJ, WORKS
WHERE PROJ.pno = WORKS.pno;
```

NOTE: The attribute pno must be specified with its table to show how the join is performed:

```
WHERE PROJ.pno = WORKS.pno
```

An alternative Join approach using an **ON** clause:

```
SELECT pname, eno
FROM PROJ INNER JOIN WORKS
ON PROJ.pno = WORKS.pno;
```

This removes the join details from the **WHERE/AND** clause into an **ON** clause and uses the **INNER JOIN** statement. **JOIN** may be used instead of **INNER JOIN** (SQL Server will assume that **INNER JOIN** is required).

18. Get the names of employees in the 'information' department

ename is from **EMP**, dname (used for selection clause) is from **DEPT**
dno is the primary/foreign key match between **DEPT** and **EMP**

```
SELECT ename
FROM DEPT, EMP
WHERE DEPT.dno = EMP.dno
AND dname = 'information';
```

OR

```
SELECT ename
FROM DEPT INNER JOIN EMP
ON DEPT.dno = EMP.dno
WHERE dname = 'information';
```

19. Get the names of all projects worked on by the employee named ‘pearse’

pname is from **PROJ**; ename is from **EMP**; **PROJ** and **EMP** join through **WORKS**
eno is the primary/foreign key match between **EMP** and **WORKS**
pno is the primary/foreign key match between **WORKS** and **PROJ**

```
SELECT pname
FROM EMP, WORKS, PROJ
WHERE EMP.eno = WORKS.eno
AND WORKS.pno = PROJ.pno
AND ename = 'pearse';
```

OR

```
SELECT pname
FROM EMP INNER JOIN WORKS ON EMP.eno = WORKS.eno
INNER JOIN PROJ ON WORKS.pno = PROJ.pno
WHERE ename = 'pearse';
```

CREATE and SAVE SQL STATEMENTS TO ANSWER THE FOLLOWING

l. Get a list of employee names with their department names
There should be 21 rows in the resulting list.

m. Get a list of employee names with their department names for employees earning more than £25,000
There should be 11 rows in the resulting list.

n. Get a list of project names with the names of all employees.
There should be 39 rows in the resulting list.

NOTE: Since **PROJ** and **EMP** tables only join through the **WORKS** table, all 3 tables are needed in the **FROM** clause with two join clauses.

NESTED QUERIES

Nested queries involve linking two or more query blocks.

Brackets are used to designate the order in which the query is performed (normally the bottom query block first)

Query blocks must be joined, normally using a primary/foreign key match (join attributes must minimally be defined on the same domain) using = or **IN**

- = is used where only *ONE* value is passed
- **IN** is used where *MORE THAN ONE* value is passed (it is equivalent to a set of multiple **OR** clauses).

As the nested query is evaluated, only the selected values from the join attribute of the lower query block can be passed to the corresponding attribute in the query block above.

Nested queries can be used for

- joins where the result table requires only **SELECT** attributes from the top query block
- complex queries where the result of an aggregate function is passed to the top query block
- complex queries where selected groups (using **GROUP BY** and **HAVING** clauses in a lower query block) can be passed to a top query to ask for more information about those selected groups.

20. Query 18 (Get the names of employees in the information department) could be answered using a nested query approach. Try it. (Query 17 cannot use a nested approach).

```
SELECT ename
FROM EMP
WHERE dno IN
   (SELECT dno
   FROM DEPT
   WHERE dname = 'information');
```

Note that the join is achieved by having dno in the **WHERE** clause of the top query block and also in the **SELECT** clause of the bottom query – in this example = could have been used instead of **IN** to join the query blocks.

The query evaluates from the bottom up

- i.e. first works out which department number the information department has
- then passes this value into dno in the top query block where the name of employees in that department is returned

NOTE: = could have been used instead of **IN** because only one value was passed upwards. However for *nested joins* using query blocks, **IN** is the *best* choice since normally multiple values may be passed

21. Query 19 could also be specified as a nested query.

```
SELECT pname
FROM PROJ
WHERE pno IN
   (SELECT pno
   FROM WORKS
   WHERE eno IN
      (SELECT eno
      FROM EMP
      WHERE ename = 'pearse'));
```

NOTE: the bracket sequence allows the query to evaluate from the bottom query block upwards.

Alternatively, a combination of single query block join and nested query could be used, e.g.:

```
SELECT pname
FROM PROJ, WORKS
WHERE PROJ.pno = WORKS.pno
AND eno IN
   (SELECT eno
   FROM EMP
   WHERE ename = 'pearse');
```

OR

```
SELECT pname
FROM PROJ INNER JOIN WORKS ON PROJ.pno = WORKS.pno
WHERE eno IN
   (SELECT eno
   FROM EMP
   WHERE ename = 'pearse');
```

ALTERNATIVE NESTED QUERY TYPE

22. Get the employees with a lower than average salary

Remember, aggregate functions such as **AVG( )** are *NOT* allowed in a **WHERE** clause.

Strategy: Work out the average salary for all employees (lower query block)
Pass the value into salary in a top query block
Get the names of employees who earn below that salary value

```
SELECT ename
FROM EMP
WHERE salary <
   (SELECT AVG(salary)
   FROM EMP);
```

NOTE: < is used because the average salary passed from the lower query block *MUST* be a single value

NOW TRY THESE QUERIES and SAVE THEM

o. How many employees are there in the 'information' department (i.e. dname)?
Try this as a single query block join and also as a nested query.
Answer - 9

p. Get a list of employee names for projects named 'payroll' or 'database'.
Again, try this as a single query block join and also as a nested query.
There should be 13 records in result

q. Get the names of all employees with an above average salary.
There should be 9 records in result

r. Get the names and salaries of all employees in the 'information' department with salaries above the average for employees in the 'information' department.
There should be 5 records in result

s. Get the names and salaries of employees in the 'information' department who have a higher salary than the maximum salary in the 'service' department.
There should be 5 records in result