SQL QUERIES – Data Manipulation

Consider the schemata, **DreamHome** for queries in SQL.

Branch(bno, street, area, city, pcode, tel_no, fax_no)

Staff(sno, fname, lname, address, tel_no, position, sex, dob, salary, nin, bno)

Renter(rno, fname, lname, address, tel_no, pref_type, max_rent)

Owner(ono, fname, lname, address, tel_no)

Property_For_Rent(pno, street, area, city, pcode, ptype, rooms, rent, ono, sno, bno)

Viewing(rno, pno, vdate, komment)

Registration(rno, bno, sno, joindt)

Configure the schema relations and populate them using the script 'dreamHome.sql'.

```
SELECT [ DISTINCT | ALL ] { * | [ columnExpression [AS newName]] [, ... ] }
    FROM TableName [alias] [, . . . ]
    [ WHERE condition ]
    [ GROUP BY columnList ] [ HAVING condition ]
    [ ORDER BY columnList ]
```

columnExpression represents a column name or an expression, TableName is the name of an existing database table or view that you have access to, and alias is an optional abbreviation for TableName.

The sequence of processing in a SELECT statement is:

FROM : specifies the table or tables to be used
WHERE : filters the rows subject to some condition
GROUP BY : forms groups of rows with the same column value
HAVING : filters the groups subject to some condition
SELECT : specifies which columns are to appear in the output
ORDER BY : specifies the order of the output

The order of the clauses in the SELECT statement cannot be changed. The only two mandatory clauses are the first two: SELECT and FROM; the remainder are optional. **The SELECT operation is closed**: the result of a query on a table is another table .

- List the property numbers for properties that were viewed alongwith the date it was viewed.

  ```
  SELECT pno, vdate FROM Viewing;
  SELECT DISTINCT pno, vdate FROM Viewing; --removes duplicates in result
  ```

- Produce a listing of monthly salaries for all staff. List staff number, staff name (as composite of fname & lname), and the salary details.

  ```
  SELECT sno, fname||' '||lname as "staffName", salary/12 as "monthlySalary"
     FROM Staff;
  ```

  Here **monthlySalary** is a calculated or computed or derived field.

Row selection (WHERE clause)

Usually it is required to restrict the rows that are retrieved. This can be achieved with the WHERE clause. The five basic search conditions (or predicates) are as follows:

1. Comparison: Compares the value of one expression to the value of another expression.
2. Range: Test whether the value of an expression falls within a specified range of values.
3. Set membership: Test whether the value of an expression equals one of a set of values.
4. Pattern match: Test whether a string matches a specified pattern.
5. Null: Test whether a column has a null (unknown) value.

SQL, the following simple comparison operators are available:

=	equals	!=	is not equal to (allowed in some dialects)
<>	is not equal to (ISO)	<=	is less than or equal to
<	is less than	>=	is less than or equal to
>	is greater than		

More complex predicates can be generated using the logical operators AND, OR, and NOT, with parentheses (if needed or desired) to show the order of evaluation. The rules for evaluating a conditional expression are:

1. an expression is evaluated left to right;
2. subexpressions in brackets are evaluated first;
3. NOTs are evaluated before ANDs and ORs;
4. ANDs are evaluated before ORs.

Use of parentheses is always recommended to remove any possible ambiguities.

- List the sddresses of all branch offices in London or Glasgow.
  [Compound Condition]

```
SELECT bno, street||' '||area||' '||city as "Address", pcode as "PostCode"
   FROM Branch
   WHERE city = 'London' OR city = 'Glasgow';
```

- List the names of staff with salary between 15000 and 25000.
  [Range Search Condition]

```
SELECT sno, fname||' '||lname as "staffName", position, salary
   FROM Staff
   WHERE salary BETWEEN 15000 AND 25000;
```

```
SELECT sno, fname||' '||lname as "staffName", position, salary
   FROM Staff
   WHERE salary >= 15000 AND salary <= 25000;
```

The BETWEEN test does not add much to the expressive power of SQL, because it can be expressed equally well using two comparison tests. However, the BETWEEN test is a simpler way to express a search condition when considering a range of values.

- List the names of all managers and supervisors.
  [Set Membership Search Condition – **IN / NOT IN**]

```
SELECT sno, fname||' '||lname as "staffName", position
   FROM Staff
   WHERE position IN ( 'Manager', 'Supervisor');


SELECT sno, fname||' '||lname as "staffName", position
   FROM Staff
   WHERE position = 'Manager' OR position = 'Supervisor';
```

Like BETWEEN, the **IN** test does not add much to the expressive power of SQL. However, the IN test provides a more efficient way of expressing the search condition, particularly if the set contains many values.

- List the names of all owners residing in Glasgow.
  [Pattern Match Search Condition – **LIKE / NOT LIKE**]

```
SELECT ono, fname||' '||lname as "staffName", address
   FROM Owner
   WHERE address LIKE '%Glasgow%';
```

SQL has two special pattern-matching symbols:

% percent character represents any sequence of zero or more characters.
_ underscore character represents any single character.

If the search string can include the pattern-matching character itself, we can use an **escape** character to represent the pattern-matching character. For example, to check for the string '15%', we can use the predicate:

```
LIKE '15#%' ESCAPE '#'   -- to check for specific flavor of SQL
```

- List details of all viewings on property PG4 with no supporting comment.
  [Null Search Condition – **NULL / NOT NULL**]

```
SELECT rno, vdate
   FROM Viewing
   WHERE pno = 'PG4' AND komment IS NULL;
```

Try following search conditions and observe result [the result is incorrect].

1. `(pno = 'PG4' AND komment = '')`
2. `(pno = 'PG4' AND komment <> 'too remote');`

- Produce a list of salaries for all staff, arranged in descending order of salary. [Single-Column Ordering]

  The rows of an SQL query result table are not arranged in any particular order (although some DBMSs may use a default ordering based, for example, on a primary key). To ensure that the results of a query are sorted the **ORDER BY** clause is used in the SELECT statement.

  The **ORDER BY** clause consists of a list of column identifiers that the result is to be sorted on, separated by commas. A column identifier may be either a column name or a column number that identifies an element of the SELECT list by its position within the list.

  Column numbers could be used if the column to be sorted on is an expression and no AS clause is specified to assign the column a name that can subsequently be referenced.

  The ORDER BY clause allows the retrieved rows to be ordered in ascending (**ASC**) or descending (**DESC**) order on any column or combination of columns, regardless of whether that column appears in the result. The ORDER BY clause must always be the last clause of the SELECT statement.

```
SELECT sno, fname||' '||lname as "staffName", salary
   FROM Staff
   ORDER BY salary DESC;
```

- Produce an abbreviated list of properties arranged in order of property type. [Multiple Column Ordering]

```
SELECT pno, ptype, rooms, rent
   FROM Property_For_Rent
   ORDER BY ptype, rent DESC;
```

  Now, the result is ordered first by property type, in ascending alphabetic order (ASC being the default setting), and within property type, in descending order of rent.

SQL AGGREGATE FUNCTIONS

As well as retrieving rows and columns from the database, we often want to perform some form of summation or aggregation of data, similar to the totals at the bottom of a report.

The ISO standard defines five aggregate functions:

- COUNT – returns the number of values in a specified column;
- SUM – returns the sum of the values in a specified column;
- AVG – returns the average of the values in a specified column;
- MIN – returns the smallest value in a specified column;
- MAX – returns the largest value in a specified column.

These functions operate on a single column of a table and return a single value. COUNT, MIN, and MAX apply to both numeric and non-numeric fields, but SUM and AVG may be used on numeric fields only. Apart from COUNT(*), each function eliminates nulls first and operates only on the remaining non-null values. COUNT(*) is a special use of COUNT, which counts all the rows of a table, regardless of whether nulls or duplicate values occur.

If we want to eliminate duplicates before the function is applied, we use the keyword DISTINCT before the column name in the function. The ISO standard allows the keyword ALL to be specified if we do not want to eliminate duplicates, although ALL is assumed if nothing is specified. DISTINCT has no effect with the MIN and MAX functions. However, it may have an effect on the result of SUM or AVG, so consideration must be given to whether duplicates should be included or excluded in the computation. In addition, DISTINCT can be specified only once in a query.

It is important to note that an aggregate function can be used only in the SELECT list and in the HAVING clause (see Section 5.3.4). It is incorrect to use it elsewhere. If the SELECT list includes an aggregate function and no GROUP BY clause is being used to group data together, then no item in the SELECT list can include any reference to a column unless that column is the argument to an aggregate function. For example, the following query is illegal:

```
SELECT sno, COUNT(salary)
      FROM Staff ;
```

because the query does not have a GROUP BY clause and the column staffNo in the SELECT list is used outside an aggregate function.

Use of COUNT(*)

How many properties cost more than £350 per month to rent?

```
SELECT COUNT(*) as myCount
      FROM Property_For_Rent
      WHERE rent > 350;
```

Use of COUNT(DISTINCT)

How many different properties were viewed in May 2004?

```
SELECT COUNT(DISTINCT pno) as myCount
      FROM Viewing
      WHERE vdate BETWEEN '01-MAY-04' AND '31-MAY-04';
```

Use of COUNT and SUM

Find the total number of Managers and the sum of their salaries.

```
SELECT COUNT(sno) as myCount, SUM(salary) as mySum
      FROM Staff
      WHERE position = 'Manager';
```

Use of MIN, MAX, AVG

Find the minimum, maximum, and average staff salary.

```
SELECT MIN(salary) as myMin, MAX(salary) as myMax, AVG(salary) as myAvg
      FROM Staff;
```

Here as all staff are considered, WHERE clause is not required.

GROUP BY Clause

When it is necessary to have subtotals in reports, we can use the GROUP BY clause of the SELECT statement. A query that includes the GROUP BY clause is called a grouped query, because it groups the data from the SELECT table(s) and produces a single summary row for each group. The columns named in the GROUP BY clause are called the grouping columns. The ISO standard requires the SELECT clause and the GROUP BY clause to be closely integrated. When GROUP BY is used, each item in the SELECT list must be single-valued per group.

Further, the SELECT clause may contain only:

1. column names;
2. aggregate functions;
3. constants;
4. an expression involving combinations of the above.

All column names in the SELECT list must appear in the GROUP BY clause unless the name is used only in an aggregate function.

Find the number of staff working in each branch and the sum of their salaries.

```
SELECT bno, COUNT(sno) as myCount, SUM(salary) as mySum
      FROM STAFF
      GROUP BY bno
      ORDER BY bno;
```

The alternative code using subqueries -

```
SELECT bno, (SELECT COUNT(sno)
    FROM Staff s
    WHERE s.bno = b.bno) AS "myCount",
        (SELECT SUM(salary)
            FROM Staff s
            WHERE s.bno = b.bno) AS "mySum"
    FROM Branch b
    ORDER BY bno;
```

Restricting Groupings (HAVING clause)

The HAVING clause is designed for use with the GROUP BY clause to restrict the groups that appear in the final result table. Although similar in syntax, HAVING and WHERE serve different purposes. The WHERE clause filters individual rows going into the final result table, whereas HAVING filters groups going into the final result table.

The ISO standard requires that column names used in the HAVING clause must also appear in the GROUP BY list or be contained within an aggregate function. In practice, the search condition in the HAVING clause always includes at least one aggregate function, otherwise the search condition could be moved to the WHERE clause and applied to individual rows.

Remember that aggregate functions cannot be used in the WHERE clause.

For each branch office with more than one member of staff, find the number of staff working in each branch and the sum of their salaries.

```
SELECT bno, COUNT(sno) as myCount, SUM(salary) as mySum
      FROM STAFF
      GROUP BY bno
      HAVING COUNT(sno) > 0
      ORDER BY bno;
```

SubQueries

Let us examine the use of a complete SELECT statement embedded within another SELECT statement. The results of this inner SELECT statement (or subselect) are used in the outer statement to help determine the contents of the final result. A sub-select can be used in the WHERE and HAVING clauses of an outer SELECT statement, where it is called a subquery or nested query. Subselects may also appear in INSERT, UPDATE, and DELETE statements.

There are three types of subquery:

1. A **scalar subquery** returns a single column and a single row; that is, a single value. In principle, a scalar subquery can be used whenever a single value is needed.

2. A **row subquery** returns multiple columns, but again only a single row. A row subquery can be used whenever a row value constructor is needed, typically in predicates.
3. A **table subquery** returns one or more columns and multiple rows. A table subquery can be used whenever a table is needed, for example, as an operand for the IN predicate.

List the staff who work in the branch at '163 Main St'.
[Using a subquery with equality]

```
SELECT sno, fName, lName, position
      FROM Staff
      WHERE bno = (SELECT bno
                        FROM Branch
                        WHERE street = '163 Main St');
```

Alternatively,

```
SELECT sno, fName, lName, position
    FROM Staff
    WHERE bno = 'B3';
```

Using a subquery with an aggregate function

List all staff whose salary is greater than the average salary, and show by how much their salary is greater than the average.

```
SELECT sno, fName, lName, position,
       salary - (SELECT AVG(salary) FROM Staff) AS salDiff
    FROM Staff
    WHERE salary > (SELECT AVG(salary) FROM Staff);
```

Alternatively,

```
SELECT sno, fName, lName, position, salary - 17000 AS salDiff
    FROM Staff
    WHERE salary > 17000;
```

The following rules apply to subqueries:

1. The ORDER BY clause may not be used in a subquery (although it may be used in the outermost SELECT statement).
2. The subquery SELECT list must consist of a single column name or expression, except for subqueries that use the keyword EXISTS.
3. By default, column names in a subquery refer to the table name in the FROM clause of the subquery. It is possible to refer to a table in a FROM clause of an outer query by qualifying the column name.
4. When a subquery is one of the two operands involved in a comparison, the subquery must appear on the right-hand side of the comparison.

But, this otherwise formulated query would generate incorrect result,

```
SELECT sno, fName, lName, position, salary
    FROM Staff
    WHERE (SELECT AVG(salary) FROM STAFF ) < salary;
```

Nested subqueries: use of IN

List the properties that are handled by staff who work in the branch at '163 Main St'.

```
SELECT pno, street, city, pcode, ptype, rooms, rent
    FROM Property_For_Rent
    WHERE sno IN (SELECT sno
                  FROM Staff
                  WHERE bno = (SELECT bno
                               FROM Branch
                               WHERE street = '163 Main St')
                );
```

ANY & ALL

The words ANY and ALL may be used with subqueries that produce a single column of numbers. If the subquery is preceded by the keyword ALL, the condition will only be true if it is satisfied by all values produced by the subquery. If the subquery is preceded by the keyword ANY, the condition will be true if it is satisfied by any (one or more) values produced by the subquery. If the subquery is empty, the ALL condition returns true, the ANY condition returns false. The ISO standard also allows the qualifier SOME to be used in place of ANY.

Find all staff whose salary is larger than the salary of at least one member of staff at branch B3.

[**Use of ANY/SOME**]

```
SELECT sno, fName, lName, position, salary
    FROM Staff
    WHERE salary > SOME (SELECT salary
                         FROM Staff
                         WHERE bno = 'B3');
```

Find all staff whose salary is larger than the salary of every member of staff at branch B3.

[**Use of ALL**]

```
SELECT sno, fName, lName, position, salary
    FROM Staff
    WHERE salary > ALL (SELECT salary
                        FROM Staff
                        WHERE bno = 'B3');
```

MULTI-TABLE QUERIES

A join operation combines columns from several tables into a result. The SQL join operation combines information from two tables by forming pairs of related rows from the two tables.

While obtaining information from more than one table, the choice is between using a subquery and using a join. If the final result table is to contain columns from different tables, then a join must be used. An alias can be used to qualify a column name whenever there is ambiguity regarding the source of the column name.

List the names of all clients who have viewed a property along with any comment supplied.

```
SELECT r.rno, fName, lName, pno, komment
    FROM Renter r, Viewing v
    WHERE r.rno = v.rno;
```

The multi-table queries involve two tables that have a one-to-many (**1 : ***) (or a parent/child) relationship . In an RDBMS the parent/child relationship is craeted using primary key and foreign keys: the table containing the primary key is the parent table and the table containing the foreign key is the child table.

The SQL standard provides the following alternative ways to specify this join:

1. `FROM Renter r JOIN Viewing v ON r.rno = v.rno`
2. `FROM Renter r JOIN Viewing USING rno`
3. `FROM Renter r NATURAL JOIN Viewing`

For each branch office, list the numbers and names of staff who manage properties and the properties that they manage.

[Sorting a Join]

```
SELECT s.bno, s.sno, fName, lName, pno
      FROM Staff s, Property_For_Rent p
      WHERE s.sno = p.sno
      ORDER BY s.bno, s.sno, pno;
```

For each branch office list the numbers and names of staff who manage properties, including the city in which the branch is located and the properties that the staff manage.

[Three-Table Join]

```
SELECT b.bno, b.city, s.sno, fName, lName, pno
      FROM Branch b, Staff s, Property_For_Rent p
      WHERE b.bno = s.bno AND s.sno = p.sno
      ORDER BY b.bno, s.sno, pno;
```

Alternatively,

```
SELECT b.bno, b.city, s.sno, fName, lName, pno
      FROM Branch b JOIN Staff s USING bno) AS bs
            JOIN Property_For_Rent p USING sno
      ORDER BY b.bno, s.sno, pno;
```

Find the number of properties handled by each staff member.

[**Multiple Grouping columns**]

```
SELECT s.bno, s.sno, COUNT(*) AS myCount
      FROM Staff s, Property_For_Rent p
      WHERE s.sno = p.sno
      GROUP BY s.bno, s.sno
      ORDER BY s.bno, s.sno;
```

COMPUTING A JOIN

A join is a subset of a more general combination of two tables known as the **Cartesian product**. The Cartesian product of two tables is another table consisting of all possible pairs of rows from the two tables. The columns of the product table are all the columns of the first table followed by all the columns of the second table.

If we specify a two-table query without a WHERE clause, SQL produces the Cartesian product of the two tables as the query result.

```
SELECT [ DISTINCT | ALL ] { * | columnList }
      FROM TableName2 CROSS JOIN TableName2
```

Conceptually, the procedure for generating the results of a SELECT with a join is as follows:

1. Form the Cartesian product of the tables named in the FROM clause.
2. If there is a WHERE clause, apply the search condition to each row of the product table, retaining those rows that satisfy the condition. In terms of the relational algebra, this operation yields a restriction of the Cartesian product.
3. For each remaining row, determine the value of each item in the SELECT list to produce a single row in the result table.
4. If SELECT DISTINCT has been specified, eliminate any duplicate rows from the result table. In the relational algebra, Steps 3 and 4 are equivalent to a projection of the restriction over the columns mentioned in the SELECT list.
5. If there is an ORDER BY clause, sort the result table as required.

OUTER JOINS

The join operation combines data from two tables by forming pairs of related rows where the matching columns in each table have the same value. If one row of a table is unmatched, the row is omitted from the result table. The ISO standard provides another set of join operators called outer joins. The outer join retains rows that do not satisfy the join condition.

Let us consider the (Inner) join on two tables – Branch, Property_For_Rent:

```
SELECT b.*, p.*
      FROM Branch b, Property_For_Rent p
      WHERE b.city = p.city;
```

When it is needed to include the unmatched rows in the result table, an Outer join may be used. There are three types of Outer join: Left, Right, and Full Outer joins.

List all branch offices and any properties that are in the same city.

[**Left Outer join**]

```
SELECT b.*, p.*
      FROM Branch b LEFT JOIN Property_For_Rent p
      ON b.city = p.city;
```

List all properties and any branch offices that are in the same city.

[**Right Outer join**]

```
SELECT b.*, p.*
      FROM Branch b RIGHT JOIN Property_For_Rent p
      ON b.city = p.city;
```

List the branch offices and properties that are in the same city along with any unmatched branches or properties .

[**Full Outer join**]

```
SELECT b.*, p.*
      FROM Branch b FULL JOIN Property_For_Rent p
      ON b.city = p.city;
```

EXISTS and NOT EXISTS

The keywords EXISTS and NOT EXISTS are designed for use only with subqueries. They produce a simple true/false result. EXISTS is true if and only if there exists at least one row in the result table returned by the subquery; it is false if the subquery returns an empty result table.

NOT EXISTS is the opposite of EXISTS. Since EXISTS and NOT EXISTS check only for the existence or non-existence of rows in the subquery result table, the subquery can contain any number of columns.

Find all staff who work in a London branch office.

[Query using EXISTS]

```
SELECT sno, fName, lName, position
      FROM Staff s
      WHERE EXISTS (SELECT *
                        FROM Branch b
                        WHERE s.bno = b.bno AND city = 'London');
```

This query could be rephrased as "**Find all staff such that there exists a Branch row containing his/her branch number, branchNo, and the branch city equal to London**". The test for inclusion is the existence of such a row. If it exists, the subquery evaluates to true.

Alternatively,

```
SELECT sno, fName, lName, position
      FROM Staff s, Branch b
      WHERE s.bno = b.bno AND city = 'London';
```