**Database Systems**
**Practical 4**
**SQL Select Command/ Aggregate Functions**
***Version 1.1***

In this section, we will play with the most vital statement of SQL- SELECT statement. Remember, you can do so many things with this single statement along with others that a book can be written. Even many software developers do not know many tricks that can be played with SELECT. So, try to catch more tricks, just don't sit with this practical.

The general syntax for SELECT statement is-

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

In this case, only SELECT and FROM are must for this expression. All others are optional. Order of the clauses cannot be changed.

| | |
|---|---|
| FROM | Specifies table(s) to be used. |
| WHERE | Filters rows. |
| GROUP BY | Forms groups of rows with same column value. |
| HAVING | Filters groups subject to some condition. |
| SELECT | Specifies which columns are to appear in output. |
| ORDER BY | Specifies the order of the output |

## Demonstration

Run the script **employee2.sql** by "START **drive:/employee2.sql;**" command. Then try to follow the steps provided below one by one and see the results. You can also take a note of them if you wish. Not all of the clauses used it SELECT statement will be covered in this section. We will go through them next week.

### *All columns and all rows*

```
SELECT fname, mi, lname, ssn, bdate, address, salary, superssn, dno
FROM employee;

SELECT * FROM employee;
```

**Nb. In this case, "all columns" are substituted by "*" sign.**

### *Specific Columns, All Rows*

```
SELECT fname, lname, dno
FROM employee;
```

### *Use of DISTINCT*

DISTINCT is used to eliminate repeating elements.

```
SELECT dno FROM employee;
SELECT DISTINCT (dno) FROM employee;
```

Take a look at the difference between them.

### *Calculated Fields*

You can make numerical calculations on related columns of a table also. In this case, we will divide the salary of employees who have department number 5.

```
SELECT (salary/5) FROM employee WHERE dno=5;
```

### *Giving the column a different name*

When you are showing data by SELECT command, you can put the column name to be shown as your wish by AS clause. Remember, this naming is just for showing. It has no impact on the actual column name of the table.

```
SELECT (salary/5) AS salary_divide_by_five FROM employee;
```

### *Comparison Search Condition*

```
SELECT fname FROM employee;

SELECT fname FROM employee
WHERE dno>1;
```

See the difference.

### *Compound Comparison Search Condition*

```
SELECT fname, lname
FROM employee
WHERE dno=1 OR dno=5;
```

You can use AND operator also. See the result.

### *Range Search Condition*

```
SELECT fname, lname
FROM employee
WHERE salary BETWEEN 40000 AND 50000;
```

Take a look at the BETWEEN clause here.

```
SELECT fname, lname
FROM employee
WHERE salary NOT BETWEEN 40000 AND 50000;
```

Take a look at the NOT BETWEEN clause here.

```
SELECT fname, lname
FROM employee
WHERE salary>= 40000 AND salary <=50000;
```

Look at the range operators <= and >= here. These are more flexible than BETWEEN and NOT BETWEEN clauses.

### *Set Membership*

```
SELECT fname, lname
FROM employee
WHERE salary IN (30000, 40000);
```

This finds out the first and last name of those who has salary exactly 30000 or 40000. Similarly the following statement negates it.

```
SELECT fname, lname
FROM employee
WHERE salary NOT IN (30000, 40000);
```

### *Pattern Matching*

Try to execute following statements one by one.

```
SELECT fname, lname, address
FROM employee;

SELECT fname, lname, address
FROM employee
WHERE address LIKE '%houston%';

SELECT fname, lname, address
FROM employee
WHERE address LIKE '%HOUSTON%';
```

Found any difference? The second one does not work as the table has data in all capitals. That is why the third one works.

### *Single Column Ordering*

This time we will use ORDER BY clause. It is used to represent data in the tables in some particular (and desired) order.

```
SELECT fname, lname, salary, dno
FROM employee
ORDER BY salary;

SELECT fname, lname, salary, dno
FROM employee
ORDER BY salary desc;
```

Which one is by default- ascending or descending?

### *Multiple Columns Ordering*

When you are using multiple columns in ORDER BY clause, please be careful. Because, sometimes it is difficult to see the difference in actual results.

```
SELECT fname, lname, salary, dno
FROM employee
ORDER BY salary, dno;
```

In this case, the result will be-

| FNAME | LNAME | SALARY | DNO |
|---|---|---|---|
| ALICIA | ZELAYA | 25000 | 4 |
| AHMAD | JABBAR | 25000 | 4 |
| JOYCE | ENGLISH | 25000 | 5 |
| JOHN | SMITH | 30000 | 5 |
| RAMESH | NARAYAN | 38000 | 5 |
| FRANKLIN | WONG | 40000 | 5 |
| JENNIFER | WALLACE | 43000 | 4 |
| JAMES | BORG | 55000 | 1 |

```
SELECT fname, lname, salary, dno
FROM employee
ORDER BY salary, dno desc;
```

In this case the result will be-

| FNAME | LNAME | SALARY | DNO |
|---|---|---|---|
| JOYCE | ENGLISH | 25000 | 5 |
| ALICIA | ZELAYA | 25000 | 4 |
| AHMAD | JABBAR | 25000 | 4 |
| JOHN | SMITH | 30000 | 5 |
| RAMESH | NARAYAN | 38000 | 5 |
| FRANKLIN | WONG | 40000 | 5 |
| JENNIFER | WALLACE | 43000 | 4 |
| JAMES | BORG | 55000 | 1 |

You can find out the difference in the first 3 rows where salary is all 25000. take a look at the change in ordering regarding to the department number.

In this session, we will take a look at the Aggregate Functions- one of the handy functionalities provided by SQL.

ISO standard defines five aggregate functions:

1. COUNT returns number of values in specified column.
2. SUM returns sum of values in specified column.
3. AVG returns average of values in specified column.
4. MIN returns smallest value in specified column.
5. MAX returns largest value in specified column.

| **Each operates on a single column of a table and returns a single value.** |
|---|
| COUNT, MIN, and MAX apply to 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 remaining non-null values. |
| COUNT(*) counts all rows of a column, regardless of whether nulls or duplicate values occur. |
| Can use DISTINCT before column name to eliminate duplicates. |
| DISTINCT has no effect with MIN/MAX, but may have with SUM/AVG. |
| Aggregate functions can be used only in SELECT list and in HAVING clause. |

**Demonstration**
Run the script named **"lab4table.sql", and "lab4data.sql".**
Now you will see the maximum salary from all of the employees.

```
SELECT salary FROM employee;
SELECT MAX(salary) FROM employee;
```

With the first syntax, take a look that sale_price column is NULLABLE. It has NULL values as well. Take a look at how COUNT, SUM and AVG works on NULLABLE columns.

```
Describe cust_order;
SELECT COUNT(*), COUNT(sale_price) FROM cust_order;
SELECT COUNT(*), SUM(sale_price), AVG(sale_price)
FROM cust_order;
```

There may be situations where you want an average to be taken over all the rows in a table, not just the rows with non-NULL values for the column in question. In those situations you have to use the NVL function within the AVG function call to assign 0 (or some other useful value) to the column in place of any NULL values.

```
SELECT AVG(NVL(sale_price,0)) FROM cust_order;
```

Most aggregate functions allow the use of DISTINCT or ALL along with the expression argument. DISTINCT allows you to disregard duplicate expression values, while ALL causes duplicate expression values to be included in the result. Notice that the column cust_nbr has duplicate values. Observe the result of the following SQL:

```
SELECT COUNT(cust_nbr), COUNT(DISTINCT cust_nbr), COUNT(ALL cust_nbr)
FROM cust_order;
```

An important thing to note here is that ALL doesn't cause an aggregate function to consider NULL values. For example, COUNT(ALL SALE_PRICE) in the following example still returns 14, and not 20:

```
SELECT COUNT(ALL sale_price) FROM cust_order;
```

**GROUP BY clause**

The GROUP BY clause, along with the aggregate functions, groups a result set into multiple groups, and then produces a single row of summary information for each group. For example, if you want to find the total number of orders for each customer, execute the following query:

```
SELECT cust_nbr, COUNT(order_nbr)
FROM cust_order
GROUP BY cust_nbr;
```

| **Aggregate expressions generally require a GROUP BY clause** |
|---|
| GROUP BY clause must include all non-aggregate expressions |
| Aggregate functions not allowed in GROUP BY clause |
| You are not required to show your GROUP BY columns |
| When you GROUP BY a column that contains NULL values for some rows, all the rows with NULL values are placed into a single group and presented as one summary row in the output |

While producing summary results using the GROUP BY clause, you can filter records from the table based on a WHERE clause, as in the following example, which produces a count of orders in which the sale price exceeds $25 for each customer:

```
SELECT cust_nbr, COUNT(order_nbr)
FROM cust_order
WHERE sale_price > 25
GROUP BY cust_nbr;
```

**HAVING clause**

The HAVING clause is closely associated with the GROUP BY clause. The HAVING clause is used to put a filter on the groups created by the GROUP BY clause. If a query has a HAVING clause along with a GROUP BY clause, the result set will include only the groups that satisfy the condition specified in the HAVING clause. Let's look at some examples that illustrate this. The following query returns the number of orders per customer:

```
SELECT cust_nbr, COUNT(order_nbr)
FROM cust_order
GROUP BY cust_nbr
HAVING cust_nbr < 6;
```

A better version of the previous query would be:

```
SELECT cust_nbr, COUNT(order_nbr)
FROM cust_order
WHERE cust_nbr < 6
GROUP BY cust_nbr;
```

The next example shows a more appropriate use of the HAVING clause:

```
SELECT cust_nbr, COUNT(order_nbr)
FROM cust_order
GROUP BY cust_nbr
HAVING COUNT(order_nbr) > 2;
```

The syntax for the HAVING clause is similar to that of the WHERE clause. However, there is one restriction on the conditions you can write in the HAVING clause. A HAVING condition can refer only to an expression in the SELECT list, or to an expression involving an aggregate function. If you specify an expression in the HAVING clause that isn't in the SELECT list, or that isn't an aggregate expression, you will get an error. For example:

```
SELECT cust_nbr, COUNT(order_nbr)
FROM cust_order
GROUP BY cust_nbr
HAVING order_dt < SYSDATE;
```

However, you can use an aggregate expression in the HAVING clause, even if it doesn't appear in the SELECT list, as illustrated in the following example:

```
SELECT cust_nbr
FROM cust_order
GROUP BY cust_nbr
HAVING COUNT(order_nbr) < 5;
```