MODULE TITLE: **Use Advanced Structured Query Language**

NOMINAL DURATION: **200** hrs

LO1. Write advanced SQL statement to retrieve and sort data

1.1. Concepts in Advanced database system

A database system is a way of organizing information on a computer, implemented by a set of computer programs. This kind of organization should offer:

* Simplicity - an easy way to collect, access-connects, and display information;
* Stability - to prevent unnecessary loss of data;
* Security - to protect against unauthorized access to private data;
* Speed - For fast results in a short time and for easy maintenance in case of changes in the organization that stores the information.

Advanced database system concepts include:

- Proper use of indexes.
- Triggers
- Stored procedures

A **Trigger** is a special kind of stored procedure that executes whenever an attempt is made to modify data in a table that the trigger protects.

Stored procedure is set of Structured Query Language (SQL) statements that perform particular task.

1.2. Review of SQL

Structured Query Language

- Standard for relational database management systems
- Specify syntax/semantics for data definition and manipulation.
- Define data structures
- Enable portability
- Specify minimal standards
- Allow for later growth/enhancement to standard

Benefits of a standardized relational language

- Reduced training costs
- Productivity
- Application portability
- Application longevity
- Reduced dependence on a single vendor
- Cross-system communication

SQL Environment

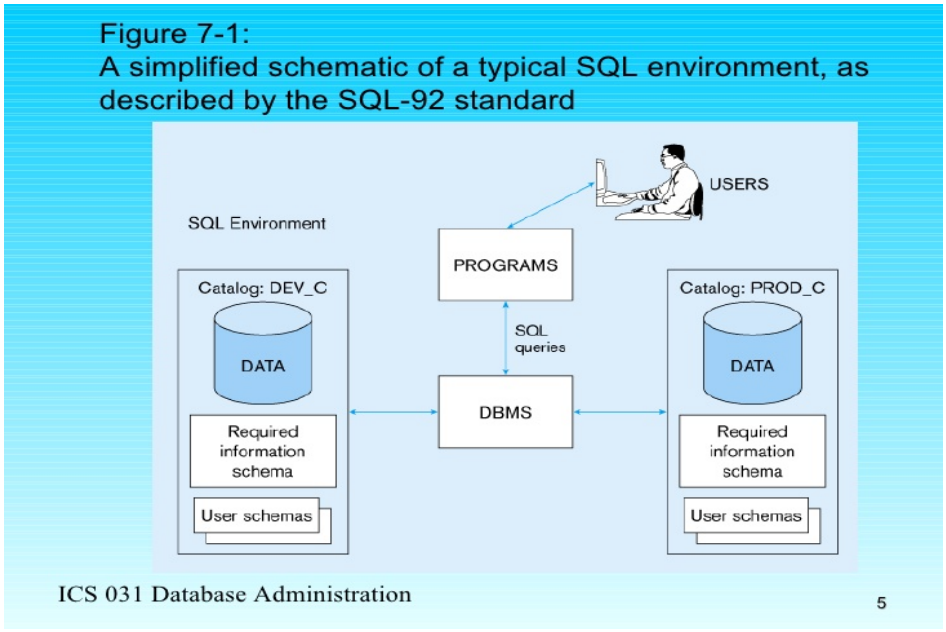
Catalog: a set of schemas that constitute the description of a database.

Schema: the structure that contains descriptions of objects created by a user (base tables, views, constraints …)

DDL: commands that define a database, including creating, altering, and dropping tables and establishing constraints

DML: commands that maintain and query a database.

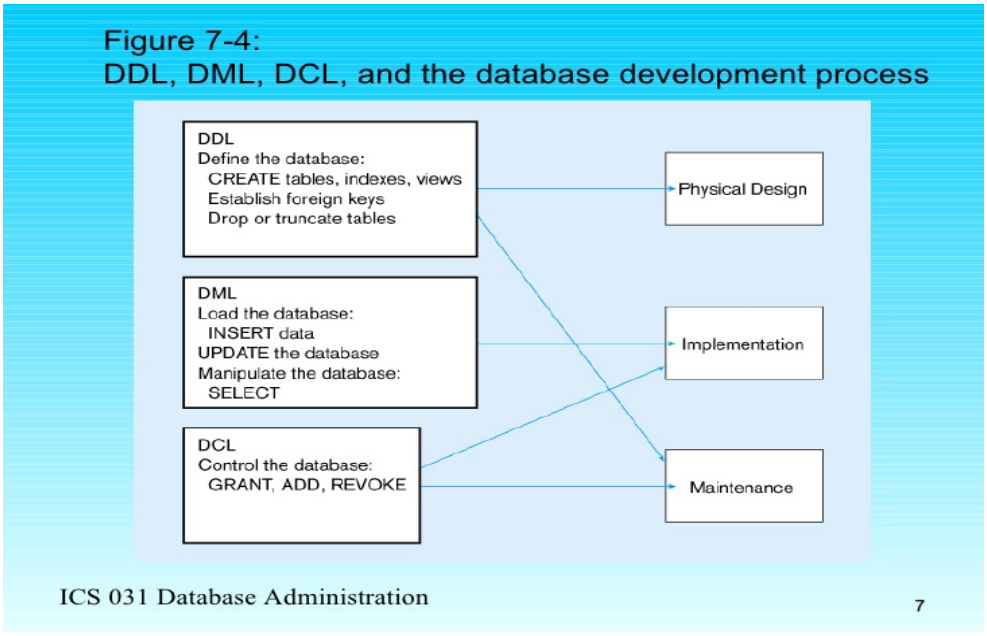
ACL (Data Control Language): commands that control a database, including administering privileges and committing data.



Some SQL Data Types

- **strings type**
  - Char(n) - fixed length character data, n characters long. Maximum length= 2000 bytes.
  - varchar(n) - variable length character data
    - Maximum length=4000 bytes
  - long – variable length character data, up to 4 GB . Maximum 1 per table.

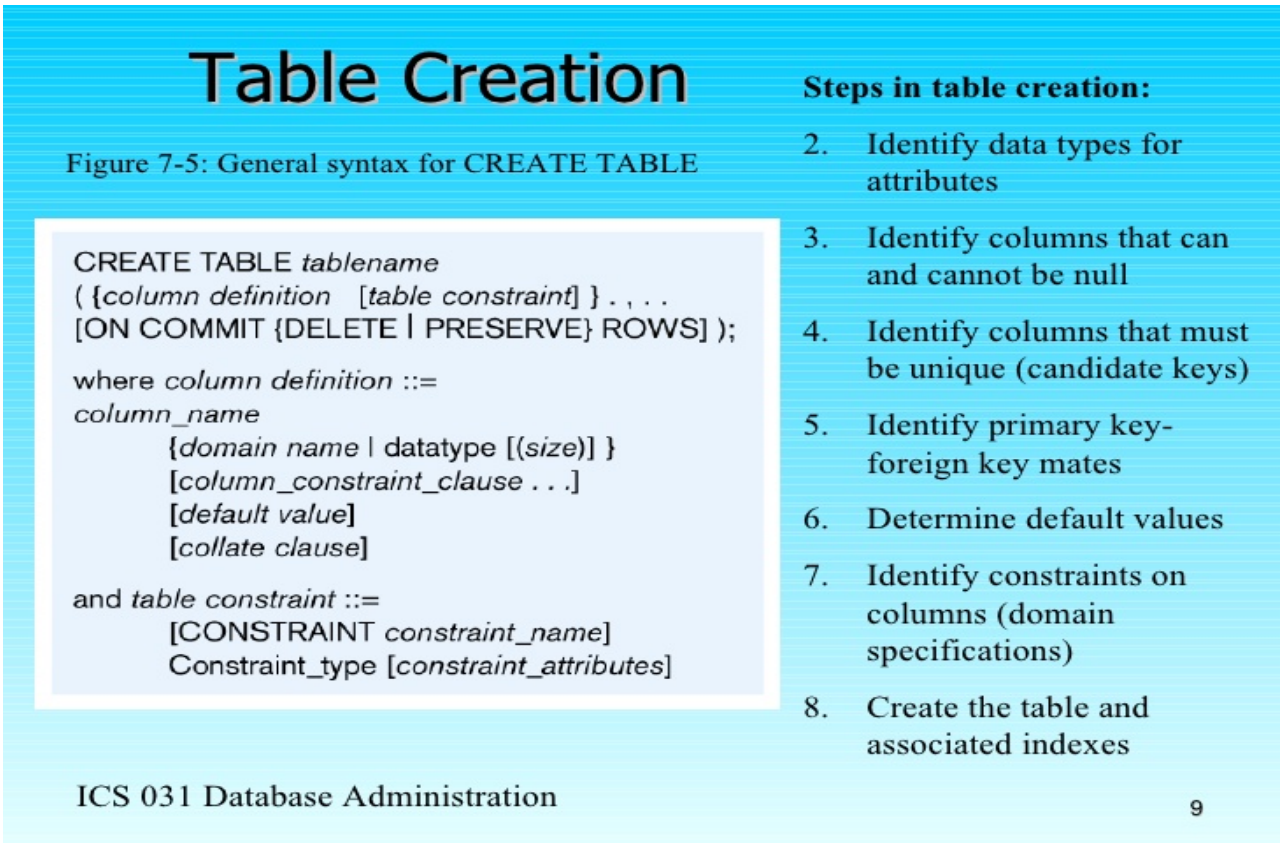
- **Numeric types**
  - Number(p,q) – general purpose numeric data type
  - Integer(p) – signed integer, p digits wide
  - Float(p) - floating point in scientific notation with p binary.
  - Date/Time type - Date - fixed length date/time in **dd-mm-yyyy** form



SQL Database Definition

DDL: The Major create statements include:

- **Create schema** – defines a portion of the database owned by a particular user
- **Create table** - defines a table and its columns
- **Create views** - defines a logical table from one or more tables.
- Other creates statements such as **create function, create trigger, create ind**ex, and so on.



Data Integrity Controls

Data integrity controls refer to the consistency and accuracy of data that is stored in a database.

Data Integrity validates the data before getting stored in the columns of the table.

types of data integrity:

- Domain Integrity
- Entity Integrity
- Referential Integrity

Each type of data integrity – domain, Entity, and referential – is enforced with separate types of constraints.

- **Domain Integrity**:

Domain integrity is the validity of entries for a given column.

You can enforce **domain integrity** by restricting the type (through data types), the format (through CHECK constraints and rules), or the range of possible values (through FOREIGN KEY constraints, CHECK constraints, DEFAULT definitions, NOT NULL definitions, and rules).

- **Entity Integrity:**

Entity integrity defines a row as a unique entity for a particular table. We can enforce the Entity Integrity through the indexes, UNIQUE constraints, PRIMARY KEY constraints, or IDENTITY properties

- **Referential Integrity**

Referential integrity maintains the defined relationships between tables when records are entered or deleted.
When you enforce referential integrity, SQL Server prevents users from:

- Adding records to a related table if there is no associated record in the primary table.
- Changing values in a primary table that result in orphaned records in a related table.
- Deleting records from a primary table if there are matching related records.

Changing and Removing Tables

- Alter table statement allows you to change column specifications.
- Drop table statement allows you to remove tables from your schema.

1.3. SQL query keywords

- SELECT clause
- WHERE Clause
- ORDER BY Clause
- INNER JOIN Clause
- INSERT Statement
- UPDATE Statement
- DELETE Statement

Insert statement: - Insert statement is used to add/insert new records or from another tables to a particular table.

Delete statement: - Remove/delete row/s from a table.

Update statement: - Modifies data in existing rows

Select statement: - Used to queries from a single or multiple tables.

- **Select** - lists the column/s that should be returned from the query.
- **From** - indicate the table/s or view/s from which data will be obtained
- **Where** - indicate the conditions under which row/s will be included in the result.
- **Group by** – indicate categorization of results.
- **Having** – indicate the conditions under which a category (group) will be included.
- **Order by** – sorts the result according to specified criteria.

Use Aggregate function: - Use aggregate function such as sum, average, product, count, max, min, and so on.

Use Boolean operators: - Use AND, OR, and NOT operators for customizing conditions in **where** clause.

1.4. Basic SELECT (Query) statement

You can write SQL statements on one line or on many lines.

- ✓ The SQL statements can be run on SQL tables and views, and database physical and logical files.
- ✓ Character strings specified in an SQL statement (such as those used with WHERE or VALUES clauses) are case sensitive; that is, uppercase characters must be entered in uppercase and lowercase characters must be entered in lowercase.

```
WHERE ADMRDEPT='a00'     (does not return a result)
WHERE ADMRDEPT='A00'     (returns a valid department number)
```

A SELECT statement can include the following:

1. The name of each column you want to include
2. The name of the table or view that contains the data
3. A search condition to uniquely identify the row that contains the information you want
4. The name of each column used to group your data
5. A search condition that uniquely identifies a group that contains the information you want
6. The order of the results so a specific row among duplicates can be returned.

The general syntax of SELECT statement looks like this:

```
SELECT column names
  FROM table or view name
  WHERE search condition
  GROUP BY column names
  HAVING search condition
  ORDER BY column-name
```

The SELECT and FROM clauses must be specified. The other clauses are optional.
With the SELECT clause, you specify the name of each column you want to retrieve.

The FROM clause specifies the table that you want to select data from. You can select columns from more than one table.

You can specify that only one column be retrieved or as many as 8000 columns. The value of each column you name is retrieved in the order specified in the SELECT clause.

If you want to retrieve all columns (in the same order as they appear in the table's definition), use an asterisk (*) instead of naming the columns.

1.5. **Basic SELECT Query**

- ✓ Simple selects
- ✓ Joins/ join types
- ✓ Aggregate operators
- ✓ Aggregation by groups and groups condition
- ✓ Sub queries/ sub queries in FROM
- ✓ Union, Intersect, Except

Simple selects

The simple select statement **select** clause which used to specify the columns you want to retrieve, **from** clause which used to specify the tables from which the columns are to be retrieved, and the where clause which used to limits the rows returned by your query.

Here is the basic syntax: `SELECT <column_list> FROM <table_list> <[WHERE <search_criteria>]>`

Example:

```
SELECT distinct  salary FROM Employee where Gender ='female'
```

Joins/ join types

Joining data from more than one table

Sometimes the information you want to see is not in a single table. To form a row of the result table, you might want to retrieve some column values from one table and some column values from another table. You can retrieve and join column values from two or more tables into a single row.

Several different types of joins are:

- inner join
- left outer join
- right outer join
- left exception join
- right exception join
- Cross join

- An **Inner Join** returns only the rows from each table that have matching values in the join columns. Any rows that do not have a match between the tables will not appear in the result table.
- **A Left Outer Join** returns values for all of the rows from the first table (the table on the left) and the values from the second table for the rows that match. Any rows that do not have a match in the second table will return the null value for all columns from the second table.
- A **Right Outer Join** return values for all of the rows from the second table (the table on the right) and the values from the first table for the rows that match. Any rows that do not have a match in the first table will return the null value for all columns from the first table.

- A **Left Exception Join** returns only the rows from the left table that do not have a match in the right table. Columns in the result table that come from the right table have the null value.
- A **Right Exception Join** returns only the rows from the right table that do not have a match in the left table. Columns in the result table that come from the left table have the null value.
- A **Cross Join** returns a row in the result table for each combination of rows from the tables being joined (a Cartesian product).

Subqueries:

A Subquery or Inner query or Nested query is a query within another SQL query, and embedded within the WHERE clause.

- You use subqueries to break down a complex query into a series of logical steps and, as a result, to solve a problem with single statements.
- Each select statement in the subquery has its own:
  - select list
  - where clause

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements (or inside another subquery) along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

There are a few rules that subqueries must follow:

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery; however, the BETWEEN can be used within the subquery.

Example: For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
SELECT DNUMBER, COUNT (*) FROM DEPARTMENT, EMPLOYEE
    WHERE DNUMBER=DNO AND SALARY>40000 AND
          DNO IN (SELECT DNO FROM EMPLOYEE
                        GROUP BY DNO
                        HAVING COUNT (*) > 5)
          GROUP BY DNUMBER
```

Union, Intersect, Except

With the help of **set** operators, the results of individual table expressions can be combined. This type of combination is called **UNION**. SQL supports other set operators besides the UNION operator.
Here is the complete list:

- UNION
- UNION ALL
- INTERSECT
- INTERSECT ALL
- EXCEPT
- EXCEPT ALL

Combining tables with union

The *UNION* operator is used to combine the result-set of two or more SELECT statements.

- Each SELECT statement within the UNION must have the same number of columns and similar data types.
- Also, the columns in each SELECT statement must be in the same order.

Syntax: `SELECT <column_name_list> FROM <table_name> UNION SELECT <column_name_list> FROM <table_name>`

Note: The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL.

Rules for using UNION

The following rules must be applied to use the UNION operator:

- The SELECT clauses of all relevant table expressions must have the same number of expressions
- Each SELECT statement within the *UNION* must have similar data types. If this applies, the table expressions are union compatible.
- An ORDER BY clause can be specified only after the last table expression. The sorting is performed on the entire end result; after all intermediate results have been combined.
- The SELECT clauses should not contain DISTINCT because SQL automatically removes duplicate rows when using UNION.

Combining with INTERSECT

INTERSECT returns all rows that are both in the result of query1 and in the result of query2. Duplicate rows are eliminated unless ALL is specified.
If two table expressions are combined with the INTERSECT operator, the end result consists of those rows that appear in the results of both table expressions.

Example: `SELECT * FROM EMPLOYEE INTERSECT SELECT * FROM PROJECT;`

- Just as with the UNION operator, duplicate rows are automatically removed from the result.

Combining with EXCEPT

EXCEPT returns all rows those are in the result of query1 but not in the result of query2. Again, duplicates are eliminated unless ALL is specified.
If two table expressions are combined with the EXCEPT operator, the end result consists of only the rows that appear in the result of the first table expression but do not appear in the result of the second.

Example: `SELECT * FROM EMPLOYEE EXCEPT SELECT * FROM PROJECT;`

Just as with the UNION operator, duplicate rows are automatically removed from the result.

1.6. Basics of the SELECT Statement and conditional selection

The SELECT statement is used to select data from a database.
The result is stored in a result table, called the result-set.

Syntax of the basic select statement includes: **Select-From-Where Statements**

SELECT desired attributes
FROM one or more tables
WHERE condition about tuples of the tables

A **conditional selection** statement allows to choose a set of statements for execution depending on a condition.
If statement and *switch...case* statements are the two conditional selection statements.

```
SELECT
      if (selectField1 = true) Field1 ELSE do not select Field1
      if (selectField2 = true) Field2 ELSE  do not select Field2
FROM Table
```

If the condition is true, statement1 and statement2 is executed; otherwise statement 3 and statement 4 is executed.
The expression or condition is any expression built using relational operators which either yields true or false condition.

1.7. Operators in SQL

1.7.1. Comparison operators

A **comparison (**or **relational) operator** is a mathematical symbol or a keyword which is used to compare between two values.

- Comparison operators are used in conditions that compare one expression with another.
- The result of a comparison can be TRUE, FALSE, or UNKNOWN.
- SQL Comparison operator is used mainly with the SELECT statement to filter data based on specific conditions.

Comparison operator	Description
=	equal to
<>, !=	is not equal to
<	less than
>	greater than
>=	greater than or equal to
<=	less than or equal to
In	"Equivalent to any member of" test, Equivalent to "= ANY".
Not In	Equivalent to "! = ANY".
All	Compares a value with every value in a list
[Not] between	[Not] greater than or equal to *x* and less than or equal to *y*.
Is [not]null	Tests for nulls

Note: The *!*= operator is converted to <> in the parser stage.

1.7.2. Boolean operators in SQL

Boolean Operators are simple words (AND, OR, or NOT) used as conjunctions to combine or exclude keywords in a search.

- When you want retrieving data using a SELECT statement, you can use logical operators in the WHERE clause to combine more than one condition.

Example: Retrieve all students' first name, last name, subject who are studying either 'Maths' or 'Science'.

```
SELECT first_name, last_name,subject FROM student
     WHERE subject = 'Maths' OR subject = 'Science'
```

The **AND** and **OR** operator combines two logical operands.

- The NOT operator inverts the result of a comparison expression or a logical expression.

1.7.3. SQL Equality Operator Query

Operators are the mathematical and equality symbols which are used to compare, evaluate, or calculate values. Equality operators include the (<), (>), and (=) symbols, which are used to compare one value against another. Equality involves comparing two values. To do so requires the use of the (<), (>), or (=) special characters.

Does X = Y? Is Y < X?

These are both questions that can be answered using a SQL Equality Operator expression.

SQL operators are generally found inside of queries-- more specifically, in the conditional statements of the ***WHERE*** clause.

```
SELECT customer,day_of_order FROM orders
WHERE  day_of_order > '7/31/08'
```

1.7.4. SQL - Mathematical Operators

SQL mathematical operators are reserved words or characters which are used primarily in a SQL statement's `WHERE` clause to perform operation(s), such as comparisons and arithmetic operations.

SQL mathematical operations can be performed using mathematical operators (+, -, *, /, and % (Modulo) and comparisons operators ((<, >, and =)).

```
SELECT EmpID, FirstName, LastName FROM EMPLOYEE  WHERE (Salary + Bonus)>5000;
```

LO2: Write Advanced SQL statements that use functions

2.1. Arithmetical operators in SQL

Arithmetic operators are used to perform mathematical functions in SQL—the same as in most other languages.

Arithmetic Addition in SQL:

```
Select Salary, (salary+100) ,(salary+1000),(salary+5000) from EMPLOYEE
```

Arithmetic subtraction in SQL:

```
Select Salary, (salary-100) ,(salary-1000),(5000-salary) from EMPLOYEE
```

Arithmetic Multiplication in SQL:

```
Select Salary, (salary*100) ,(salary*1000),(salary*5000) from EMPLOYEE
```

Arithmetic Division in SQL:

```
Select Salary, (salary/100) ,(salary/1000),(salary/5000) from EMPLOYEE
```

You can do all kinds of Arithmetic combinations in this.

```
select (100+Salary)*200 ,salary,(salary-Bonus),( (Salary+Bonus)/0.100) from EMPLOYEE
```

2.2. Mathematical functions in SQL

A function is a predefined formula which takes one or more arguments as input then processes the arguments and returns an output.

SQL **mathematical function** executes a mathematical operation usually based on input values that are provided as arguments, and return a numeric value as the result of the operation. Mathematical functions operates on numeric data such as decimal, integer, float, real, smallint, and tinyint.

Some of the Arithmetic functions are:

- abs()
- floor()
- sqrt()
- exp()
- power()
- Round()

- abs() : This SQL ABS() returns the absolute value of a number passed as argument

Example: `SELECT ABS(-17.36)FROM TRAINEE`

- floor() : The SQL FLOOR() rounded up any positive or negative decimal value down to the next least integer value. SQL DISTINCT along with the SQL FLOOR() function is used to retrieve only unique value after rounded down to the next least integer value depending on the column specified.

Example: `SELECT floor(17.26)FROM TRAINEE`

- exp() : The SQL EXP() returns e raised to the n-th power(n is the numeric expression), where e is the base of natural algorithm and the value of e is approximately 2.71828183.

Example:
```
SELECT EXP(2) AS e_to_2s_power FROM TRAINEE
```

- power() : This SQL POWER() function returns the value of a number raised to another, where both of the numbers are passed as arguments.

Example: `SELECT POWER(2,3)FROM TRAINEE`

- sqrt() : The SQL SQRT() returns the square root of given value in the argument.

Example: `SELECT SQRT(Credit)FROM TRAINEE`

SQL Character Function

SQL character or string function is a function which takes one or more characters as parameters and returns a character value. Some Character functions are -

- lower() : The SQL LOWER() function is used to convert all characters of a string to lower case.

Syntax:
```
SELECT LOWER('TESTING FOR LOWER FUNCTION')
        AS Testing_Lower
        FROM TRAINEE
```

- upper(): The SQL UPPER() function is used to convert all characters of a string to uppercase.

syntax:
```
SELECT UPPER('testing for upper function')
        AS Testing_Upper
        FROM TRAINEE
```

2.3. Date functions in SQL

The most difficult part when working with dates is to be sure that the format of the date you are trying to insert, matches the format of the date column in the database.

As long as your data contains only the date portion, your queries will work as expected.

The following table lists the most important built-in date functions in SQL Server:

- GETDATE()
- DATEPART()
- DATEADD()
- DATEDIFF()

- GETDATE() : The GETDATE() function returns the current date and time from the SQL Server.

Syntax: `GETDATE()`

Example:
```
CREATE TABLE Orders(
        OrderId int NOT NULL PRIMARY KEY,
        ProductName varchar(50) NOT NULL,
        OrderDate datetime NOT NULL DEFAULT GETDATE())

SELECT GETDATE() AS CurrentDateTime
```

- DATEPART() : The DATEPART() function is used to return a single part of a date/time, such as year, month, day, hour, minute, etc.

Syntax:
```
DATEPART(datepart,date)
```

Example:
```
SELECT DATEPART(yyyy,OrderDate) AS OrderYear,
       DATEPART(mm,OrderDate) AS OrderMonth,
       DATEPART(dd,OrderDate) AS OrderDay
  FROM Orders
    WHERE OrderId=1
```

Note: the dateparts include:

Datepart	Abbreviation	Datepart	Abbreviation
Year	yy, yyyy	hour	hh
quarter	qq, q	minute	mi, n
month	mm, m	second	ss, s
dayofyear	dy, y	millisecond	ms
day	dd, d	microsecond	mcs
week	wk, ww	nanosecond	ns
weekday	dw, w		

- DATEADD() : The DATEADD() function adds or subtracts a specified time interval from a date.

Where date is a valid date expression and number is the number of interval you want to add. The number can either be positive, for dates in the future, or negative, for dates in the past.

Syntax:
```
DATEADD(datepart,number,date)
```

Example:
```
SELECT OrderId,DATEADD(day,45,OrderDate) AS OrderPayDate
  FROM Orders
```

- DATEDIFF() : The DATEDIFF() function returns the time between two dates.

Syntax:
```
DATEDIFF(datepart,startdate,enddate)
```

Example: Now we want to get the number of days between two dates. We use the following SELECT statement:

```
SELECT DATEDIFF(day,'2008-06-05','2008-08-05') AS DiffDate
```

Result: 61

Now we want to get the number of days between two dates (notice that the second date is "earlier" than the first date, and will result in a negative number). We use the following SELECT statement:

```
SELECT DATEDIFF(day,'2008-08-05','2008-06-05') AS DiffDate
```

SQL Date Data Types

SQL Server comes with the following data types for storing a date or a date/time value in the database:

- DATE - format YYYY-MM-DD
- DATETIME - format: YYYY-MM-DD HH:MM:SS
- SMALLDATETIME - format: YYYY-MM-DD HH:MM:SS

SQL Working with Dates

You can compare two dates easily if there is no time component involved!

Assume we have the following "Orders" table:

OrderId	ProductName	OrderDate
1	computer	2008-11-11
2	Digital Camera	2008-11-09
3	Printer	2008-11-11
4	Scanner	2008-10-29

Now we want to select the records with an OrderDate of "2008-11-11" from the table above.
We use the following SELECT statement:

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11'
```

The result-set will look like this:

OrderId	ProductName	OrderDate
1	Computer	2008-11-11
3	Printer	2008-11-11

Now, assume that the "Orders" table looks like this (notice the time component in the "OrderDate" column):

OrderId	ProductName	OrderDate
1	Computer	2008-11-11 13:23:44
2	Digital Camera	2008-11-09 15:45:21
3	Printer	2008-11-11 11:12:01
4	Scanner	2008-10-29 14:56:59

If we use the same SELECT statement as above:

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11'
```

We will get no result! This is because the query is looking only for dates with no time portion.

Tip: If you want to keep your queries simple and easy to maintain, do not allow time components in your dates!

2.4. SQL aggregate functions

An aggregate function operates on many records and produces a summary; works with GROUP BY.

Useful SQL Aggregate Functions

MIN	returns the smallest value in a given column
MAX	returns the largest value in a given column
SUM	returns the sum of the numeric values in a given column
AVG	returns the average value of a given column
COUNT	returns the total number of values in a given column
COUNT(*)	returns the number of rows in a table
SUM()	Returns the sum

Aggregate functions are used to compute against a "returned column of numeric data" from your SELECT statement. They basically summarize the results of a particular column of selected data.

- The **count function** Returns the number of items in *expression.* The data type returned is of type *int.*

Syntax: COUNT ([ALL | DISTINCT] <*expression*> | *)
Example: select COUNT(*), AVG(Employee.Salary) from dbo. Employee

- With the exception of the COUNT (*) function, all aggregate functions return a NULL if no rows satisfy the WHERE clause. The COUNT (*) function returns a value of zero if no rows satisfy the WHERE clause.

- The **MAX function** Returns the maximum value from *expression*. Max ignores any *NULL* values.
  Syntax: MAX ([ALL | DISTINCT] <*expression*>)
  Example: select MAX(Employee.Salary) from dbo. Employee

- The **MIN function** Returns the smallest value from *expression*. Min ignores any *NULL* values.
  Syntax: MIN ([ALL | DISTINCT] <*expression*>)
  Example: select MIN(Employee.Salary) from dbo. Employee

- The **SUM function** Returns the total of all values in *expression*. Sum ignores any *NULL* values.
  Syntax: SUM ([ALL | DISTINCT] <*expression*>)
  Example: select SUM(Employee.Salary) from dbo. Employee

- The **AVERAGE function R**eturns the average of the values in *expression.* The *expression* must contain numeric values. Null values are ignored.
  syntax: AVG ([ALL | DISTINCT] <expression>)
  Example: select ID, avg(Employee.Salary) from dbo. Employee

- **The Variance and Standard Deviation function**

  - **STDEV:** Returns the standard deviation of all values in *expression*. Stdev ignores any *NULL* values. **Syntax**: STDEV(<*expression*>)
    Example: select STDEV(Employee.Salary) from dbo. Employee

  - **STDEVP:** Returns the standard deviation for the population of all values in *expression*. Stdevp ignores any *NULL* values.
    Syntax: STDEVP(<*expression*>)
    Example: select STDEVP(Employee.Salary) from dbo. Employee

  - **VAR:** Returns the variance of all values in *expression*. Var ignores any *NULL* values.
    syntax: VAR(<*expression*>)
    Example: select VAR(Employee.Salary) from dbo. Employee

  - **VARP:** Returns the variance for the population of all values in *expression.* Varp ignores any *NULL* values. **syntax**: VARP(<*expression*>)
    Example: select VARP(Employee.Salary) from dbo. Employee

LO3. Write advanced SQL statements that use aggregation and filtering

3.1. SQL - Grouping By Multiple Columns

When an aggregate function is executed, SQL Server summarizes values for an entire table or for groups of columns within the table, producing a single value for each set of rows for the specified columns.

- You can use aggregate functions with the SELECT statement or in combination with the **GROUP BY** clause
- Use the **GROUP BY** clause on columns or expression to organize rows into groups and to summarize those groups. The **GROUP BY** clause groups rows on the basis of similarities between them.

When you use the **GROUP BY** clause, consider the following facts and guidelines:

- SQL Server returns only single rows for each group that you specify; it does not return detail information.
- All columns that are specified in the GROUP BY clause must be included in the select list.
- If you include a WHERE clause, SQL Server groups only the rows that satisfy the search conditions.
- Do not use the GROUP BY clause on columns that contain multiple null values.

Example1: For each department, retrieve the department name, the number of employees in the department, and their average salary; catagorize by the TraineeID, Departement and name respectively.

```
SELECT DEPARTEMENT.Dname ,
        COUNT (EMPLOYEE.ID)'number of employee',
        AVG (EMPLOYEE.salary)'average salary'
 FROM EMPLOYEE,DEPARTEMENT
where EMPLOYEE.dnum =DEPARTEMENT.Dnumber
GROUP BY DEPARTEMENT.Dname
```

Example2: Retrieve all trainees' TraineeID, Name, Departement , the number of course taken by each trainee and catagorize by the TraineeID, Departement and name.

```
select TRAINEE .TraineeID ,
          TRAINEE .Name,
          TRAINEE .Departement,
          COUNT(COURSE.Course_Code )'number of course taken'
   from TRAINEE ,GRADE_REPORT ,COURSE
   where TRAINEE .TraineeID =GRADE_REPORT .TraineeID and GRADE_REPORT .Course_Code =COURSE .Course_Code
    group by TRAINEE .TraineeID , TRAINEE .Departement,TRAINEE .Name
```

3.1.1. SQL - Group By Aggregate

The SQL Aggregate Functions are functions that provide mathematical operations. If you need to add, count or perform basic statistics, these functions will be of great help.

Basic syntax of Aggregate Functions:

```
SELECT <column_name1>, <column_name2> <aggregate_function(s)>
FROM <table_name>
GROUP BY <column_name1>, <column_name2>

HAVING <aggregate_function(column_name> < operator> <values>
```

Example:

```
select customer_name, SUM(amount),MIN(amount),MAX(amount),AVG(amount),COUNT(amount)
        from customers c join purchases p on c.cid = p.cid
        group by customer_name
        having SUM(amount) > '20.00';
```

3.1.2. Filtering SQL: HAVING Clause

The Having clause is optional and used in combination with the Group By clause. It is similar to the Where clause, but the Having clause establishes restrictions that determine which records are displayed after they have been grouped.

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

Basic syntax of having clause:

```
SELECT column1, ... column_n, aggregate_function (expression)
FROM table_name
[WHERE condition]
[GROUP BY column1, ... column_n]
HAVING condition
```

3.1.3. SQL: Sub-select, sorting by aggregate review data

Subquery or Inner query or Nested query is a query in a query. A subquery is usually added in the WHERE Clause of the sql statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value.

Subqueries can be used with select, insert, update, and delete along with the comparision operators like =, <, >, >=, <= etc.

Example:
```
SELECT first_name, last_name, subject
FROM student_details
WHERE games NOT IN ('Cricket', 'Football');
```

1) Usually, a subquery should return only one record, but sometimes it can also return multiple records when used with operators like IN, NOT IN in the where clause. The query would be like,

The output would be similar to:

first_name	last_name	subject
Shekar	Gowda	Badminton
Priya	Chandra	Chess

2) Lets consider the student_details table which we have used earlier. If you know the name of the students who are studying science subject, you can get their id's by using this query below,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN ('Rahul', 'Stephen');
```

but, if you do not know their names, then to get their id's you need to write the query in this manner,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN (SELECT first_name
FROM student_details
WHERE subject= 'Science');
```

Output:

id	first_name
100	Rahul
102	Stephen

In the above sql statement, first the inner query is processed first and then the outer query is processed.

3) Subquery can be used with INSERT statement to add rows of data from one or more tables to another table. Lets try to group all the students who study Maths in a table 'maths_group'.

```
INSERT INTO maths_group(id, name)
SELECT id, first_name || ' ' || last_name
FROM student_details WHERE subject= 'Maths'
```

4) A subquery can be used in the SELECT statement as follows. Lets use the product and order_items table defined in the sql_joins section.

```
select p.product_name, p.supplier_name, (select order_id from order_items where product_id =
101) as order_id from product p where p.product_id = 101
```

product_name	supplier_name	order_id
Television	Onida	5103

Correlated Subquery

A query is called correlated subquery when both the inner query and the outer query are interdependent. For every row processed by the inner query, the outer query is processed as well. The inner query depends on the outer query before it can be processed.

```
SELECT p.product_name FROM product p
WHERE p.product_id = (SELECT o.product_id FROM order_items o
WHERE o.product_id = p.product_id);
```

NOTE:
1) You can nest as many queries you want but it is recommended not to nest more than 16 subqueries in oracle.
2) If a subquery is not dependent on the outer query it is called a non-correlated subquery.

SQL SubSelect-SubQueries

```
A sub query or sub select is a select statement that returns a single value
output result and it can be nested inside other subquery or any SELECT, INSERT,
DELETE OR UPDATE statement.

Example:
USE NORTHWIND

SELECT P.PRODUCTNAME,
(SELECT CATEGORYNAME FROM CATEGORIES WHERE CATEGORYID = P.CATEGORYID)
FROM PRODUCTS P
Subquery used in the above example returns the category name of each product
in every tuple.
Example:
```

```
SELECT P.PRODUCTNAME, P.UNITPRICE, P.CATEGORYID
FROM
PRODUCTS P
WHERE
P.PRODUCTID = (SELECT PRODUCTID FROM PRODUCTS WHERE PRODUCTNAME='VEGIE-SPREAD')

Here subquery returns product id as single value to the main SQL query.



Example:

SELECT C.CATEGORYNAME,

(SELECT TOP 1 P.PRODUCTNAME FROM PRODUCTS P WHERE P.CATEGORYID=C.CATEGORYID ORDER BY
P.UNITPRICE DESC),

(SELECT TOP 1 P.UNITPRICE FROM PRODUCTS P WHERE P.CATEGORYID=C.CATEGORYID ORDER BY
P.UNITPRICE DESC),

(SELECT MAX(P.UNITPRICE) FROM PRODUCTS P WHERE P.CATEGORYID=C.CATEGORYID)
```

4.1 Single and nested queries

What is query?

A **query** is a request for information from a database (Queries are Questions).

Single query is a Single Block query.

Example: `SELECT distinct  salary FROM Employee where Gender ='female'`

A SQL **nested query** is a SELECT query that is nested inside a SELECT, UPDATE, INSERT, or DELETE SQL query.

Nested Queries are queries that contain another complete SELECT statements nested within it, that is, in the WHERE clause.

- The nested SELECT statement is called an “inner query” or an “inner SELECT.”
- The main query is called “outer SELECT” or “outer query.”
- The use of nested query in this case is to avoid explicit coding of JOIN which is a very expensive database operation and to improve query performance.

Example:
```
SELECT ID,LNAME, FNAME FROM EMPLOYEE
    WHERE (SELECT COUNT (*) FROM DEPENDENTED WHERE DEPENDENTED.EmpID =EMPLOYEE.ID) >= 2
```

4.2 Subqueries

A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery.

- Subquery is an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
- You use subqueries to break down a complex query into a series of logical steps and, as a result, to solve a problem with single statements.
- Each select statement in the subquery has its own:
    - . select list
    - . where clause

Example: For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
SELECT DNUMBER, COUNT (*) FROM DEPARTMENT, EMPLOYEE
    WHERE DNUMBER=DNO AND SALARY>40000 AND
          DNO IN (SELECT DNO FROM EMPLOYEE
                  GROUP BY DNO
                  HAVING COUNT (*) > 5)
```

```
                        GROUP BY DNUMBER;
```

4.3. Operators in sub queries

➢ **The IN, ALL, and ANY operators in subquery.**

- **IN** operator

The IN operator is an operator that allows you to specify multiple values in a WHERE clause.
A row from a table satisfies a condition with the IN operator if the value of a particular column occurs in a set of expressions. The expressions in such a set entered one by one by a user.

```
Syntax to use IN operator: SELECT column_name(s)
                           FROM table_name
                           WHERE column_name IN (value1,value2,...)
```

```
Examle: SELECT * FROM Persons WHERE LastName IN ('Aster','Maru')
        SELECT * FROM employee
                 WHERE Dnum IN(SELECT Dnum FROM DEPARTEMENT
                               WHERE Dname = 'Research');
```

- **Any o**perator

ANY operator is an operator that compares a value to each value in a list or results from a query and evaluates to true if the result of an inner query contains at least one row. **ANY** must be preceded by comparison operators.

```
Syntax to use Any operator: SELECT column_name(s)
                            FROM table_name
                            WHERE column_name =any(value1,value2,...)
```

```
Example: SELECT ID ,fname,lname FROM EMPLOYEE
         where Dnum =any(select COUNT(*) from DEPARTEMENT
                         where Dnumber='ict001')
```

- **All** operators

ALL operator is used to select all records of a SELECT STATEMENT. It compares a value to every value in a list or results from a query. The **ALL** must be preceded by the comparison operators and evaluates to **TRUE** if the query returns no rows. For example, **ALL** means greater than every value, means greater than the maximum value. Suppose **ALL** (1, 2, 3) means greater than 3.

```
Syntax to use All operator: SELECT column_name(s)
                            FROM table_name
                            WHERE column_name =all(value1,value2,...)
```

```
Enample: SELECT LNAME, FNAME FROM EMPLOYEE
          WHERE SALARY> ALL (SELECT SALARY FROM EMPLOYEE
                             WHERE Dnum=5);
```

- **SOME** operators

SOME operator is the same as ANY operator (SOME and ANY perform the same function).

SOME compares a value to each value in a list or results from a query and evaluates to true if the result of an inner query contains at least one row.

SOME must match at least one row in the subquery and must be preceded by comparison operators.

```
Example: SELECT ID,Fname,Lname FROM EMPLOYEE
         WHERE ID =SOME(SELECT EmpID FROM DEPENDENTED
                        WHERE Gender='Female');
```

- **EXISTS** operator

The **EXIST operator** checks the existence of a result of a subquery. The **EXISTS** subquery tests whether a subquery fetches at least one row. When no data is returned then this operator returns '**FALSE**'.

- A valid **EXISTS** subquery must contain an outer reference and it must be a correlated subquery.
- You can use the EXISTS and NON EXISTS operators to determine whether data exists in a list of values.
- Use the EXISTS and NOT EXISTS operators with correlated subqueries to restrict the result set of an outer query to rows that satisfy the subquery.

```
Example: SELECT DepID ,Fname ,Gender FROM DEPENDENTED
         WHERE exists(SELECT ID FROM EMPLOYEE
```