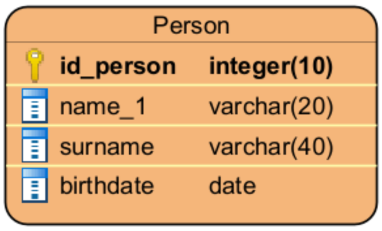
2023-02-28

During the lecture I will use the table:



Person	
id_person	**integer(10)**
name_1	varchar(20)
surname	varchar(40)
birthdate	date

SQL Scripts
In programming, scripts are the series of commands (sequence of instructions) or a program that will be executed in another program rather than by the computer processor (compiled programs are executed by computer processor –> please notice that the script is not compiled).
Same stands for SQL scripts. The only thing that is specific is that commands in such scripts are SQL commands. And these commands could be any combination of DDL (Data Definition Language) or DML (Data Manipulation Language) commands. Therefore, you could change the database structure (CREATE, ALTER, DROP objects) and/or change the data (perform INSERT/UPDATE/DELETE commands).
It's desired that you use scripts, especially when you're deploying a new version and you want to keep current data as they were before that change.
SQL Script – Comment

Indicates user-provided text. Comments can be inserted on a separate line, nested at the end of a Transact-SQL command line, or within a Transact-SQL statement. The server does not evaluate the comment.

```
-- text_of_comment  or
*/ text.....
.....
*/
```

Declaring a Transact-SQL Variable
The DECLARE statement initializes a Transact-SQL variable by:

- Assigning a name. The name must have a single @ as the first character.
- Assigning a system-supplied or user-defined data type and a length. For numeric variables, a precision and scale are also assigned. For variables of type XML, an optional schema collection may be assigned.
- Setting the value to NULL.

For example, the following **DECLARE** statement creates a local variable named **@mycounter** with an int data type.

```
DECLARE @MyCounter INT;
DECLARE @LastName NVARCHAR(30), @FirstName NVARCHAR(20), @StateProvince
NCHAR(2);
```

The scope of a variable is the range of Transact-SQL statements that can reference the variable. The scope of a variable lasts from the point it is declared until the end of the batch or stored procedure in which it is declared.

Setting a Value in a Transact-SQL Variable
When a variable is first declared, its value is set to NULL. To assign a value to a variable, use the SET statement. This is the preferred method of assigning a value to a variable. A variable can also have a value assigned by being referenced in the select list of a SELECT statement.
To assign a variable a value by using the SET statement, include the variable name and the value to assign to the variable. This is the preferred method of assigning a value to a variable. The following batch, for example, declares two variables, assigns values to them, and then uses them in the `WHERE` clause of a `SELECT` statement:

```
USE AdventureWorks2014;
GO
-- Declare two variables.
DECLARE @FirstNameVariable NVARCHAR(50),
   @PostalCodeVariable NVARCHAR(15);
```

```
-- Set their values.
SET @FirstNameVariable = N'Amy';
SET @PostalCodeVariable = N'BA5 3HX';

-- Use them in the WHERE clause of a SELECT statement.
SELECT LastName, FirstName, JobTitle, City, StateProvinceName,
CountryRegionName
FROM HumanResources.vEmployee
WHERE FirstName = @FirstNameVariable
   OR PostalCode = @PostalCodeVariable;
GO
```

A variable can also have a value assigned by being referenced in a select list. If a variable is referenced in a select list, it should be assigned a scalar value or the SELECT statement should only return one row. For example:

```
USE AdventureWorks2014;
GO
DECLARE @EmpIDVariable INT;

SELECT @EmpIDVariable = MAX(EmployeeID)
FROM HumanResources.Employee;
GO
```

If a SELECT statement returns more than one row and the variable references a non-scalar expression, the variable is set to the value returned for the expression in the last row of the result set. For example, in the following batch **@EmpIDVariable** is set to the **BusinessEntityID** value of the last row returned, which is 1:

```
USE AdventureWorks2014;
GO
DECLARE @EmpIDVariable INT;

SELECT @EmpIDVariable = BusinessEntityID
FROM HumanResources.Employee
ORDER BY BusinessEntityID DESC;

SELECT @EmpIDVariable;
```

PROCEDURE (Transact-SQL)

Creates a Transact-SQL or common language runtime (CLR) stored procedure in SQL Server, Azure SQL Database, Parallel Data Warehouse and Parallel Data Warehouse. Stored procedures are similar to procedures in other programming languages in that they can:

- Accept input parameters and return multiple values in the form of output parameters to the calling procedure or batch.
- Contain programming statements that perform operations in the database, including calling other procedures.
- Return a status value to a calling procedure or batch to indicate success or failure (and the reason for failure).

Syntax

```
CREATE [ OR ALTER ] { PROC | PROCEDURE } [schema_name.] procedure_name
    [ { @parameter data_type } [ NULL | NOT NULL ] [ = default ]
        [ OUT | OUTPUT ] [READONLY]
    ] [ ,... n ]
  WITH NATIVE_COMPILATION, SCHEMABINDING [ , EXECUTE AS clause ]
AS
{
  BEGIN ATOMIC WITH (set_option [ ,... n ] )
sql_statement [;] [ ... n ]
 [ END ]
}
 [;]
```

FUNCTION (Transact-SQL)
Creates a user-defined function in SQL Server and Azure SQL Database. A user-defined function is a Transact-SQL or common language runtime (CLR) routine that accepts parameters, performs an action, such as a complex calculation, and returns the result of that action as a value. The return value can either be a scalar (single) value or a table. Use this statement to create a reusable routine that can be used in these ways:

- In Transact-SQL statements such as SELECT
- In applications calling the function
- In the definition of another user-defined function
- To parameterize a view or improve the functionality of an indexed view
- To define a column in a table
- To define a CHECK constraint on a column
- To replace a stored procedure
- Use an inline function as a filter predicate for a security policy

Syntax
```
-- Transact-SQL Scalar Function Syntax
CREATE [ OR ALTER ] FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
 [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN scalar_expression
    END
[ ; ]
```

BEGIN...END (Transact-SQL)
Encloses a series of Transact-SQL statements so that a group of Transact-SQL statements can be executed. BEGIN and END are control-of-flow language keywords.
```
BEGIN
    { sql_statement | statement_block }
END
```

IF...ELSE (Transact-SQL)
Imposes conditions on the execution of a Transact-SQL statement. The Transact-SQL statement that follows an IF keyword and its condition is executed if the condition is satisfied: the Boolean expression returns TRUE. The optional ELSE keyword introduces another Transact-SQL statement that is executed when the IF condition is not satisfied: the Boolean expression returns FALSE.
```
IF Boolean_expression
     { sql_statement | statement_block }
[ ELSE
     { sql_statement | statement_block } ]
```

WHILE (Transact-SQL)
Sets a condition for the repeated execution of an SQL statement or statement block. The statements are executed repeatedly as long as the specified condition is true. The execution of statements in the WHILE loop can be controlled from inside the loop with the BREAK and CONTINUE keywords.
```
WHILE Boolean_expression
     { sql_statement | statement_block | BREAK | CONTINUE }
```

Operator in SQL
An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. These Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators

- Logical operators
- Operators used to negate conditions

Assume 'variable a' holds 10 and 'variable b' holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator.	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand.	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator.	a * b will give 200
/	Division - Divides left hand operand by right hand operand.	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder.	b % a will give 0

Here are a few simple examples showing the usage of SQL Arithmetic Operators:
Examples:
select 10+ 20
select 10 * 20
select 10 / 5
select 12 % 5

SQL Comparison Operator
Assume 'variable a' holds 10 and 'variable b' holds 20, then

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
! =	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a ! = b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
> =	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
< =	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a ! < b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a ! > b) is true.

Comparison Operators - Examples
SELECT * FROM CUSTOMERS
SELECT * FROM CUSTOMERS WHERE SALARY > 5000
SELECT * FROM CUSTOMERS WHERE SALARY = 2000
SELECT * FROM CUSTOMERS WHERE SALARY != 2000
SELECT * FROM CUSTOMERS WHERE SALARY <> 2000
SELECT * FROM CUSTOMERS WHERE SALARY >= 6500

SQL Logical Operators
Here is a list of all the logical operators available in SQL

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list as per the condition.

BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criterion.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. This is a negate operator.
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

Logical Operators - Examples
Here are some simple examples showing usage of SQL Comparison Operators:
Examples:
SELECT * FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500
SELECT * FROM CUSTOMERS WHERE AGE >= 25 OR SALARY >= 6500;
SELECT * FROM CUSTOMERS WHERE AGE IS NOT NULL
SELECT * FROM CUSTOMERS WHERE NAME LIKE 'Ko%';
SELECT * FROM CUSTOMERS WHERE AGE IN (25, 27)
SELECT * FROM CUSTOMERS WHERE AGE BETWEEN 25 AND 27

SELECT AGE FROM CUSTOMERS
WHERE EXISTS (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500)

SELECT * FROM CUSTOMERS WHERE AGE > ALL (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500)
SELECT * FROM CUSTOMERS
WHERE AGE > ANY (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500)

SQL expressions
An expression is a combination of one or more values, operators and SQL functions that evaluate to a value. These SQL EXPRESSIONs are like formulae and they are written in query language. You can also use them to query the database for a specific set of data.
Sy ntax
Consider the basic syntax of the SELECT statement as follows:
SELECT columnl, column2, columnN
FROM table_name
WHERE [CONDITION|EXPRESSION];
There are different types of SQL expressions, which are mentioned below:

- Boolean
- Numeric
- Date
- Let us now discuss each of these in detail.

Boolean Expressions
SQL Boolean Expressions fetch the data based on matching a single value. Following is the syntax:
SELECT * FROM CUSTOMERS
The following table is a simple example showing the usage of various SQL Boolean Expressions:
SELECT * FROM CUSTOMERS WHERE SALARY = 10000

Numeric Expressions
These expressions are used to perform any mathematical operation in any query. Following is the syntax:
SELECT numerical_expression as OPERATION_NAME [FROM table_name WHERE CONDITION] ;
Here, the numerical_expression is used for a mathematical expression or any formula. Following is a simple example showing the usage of SQL Numeric Expressions:
SELECT (15 + 6) AS ADDITION
SELECT COUNT(*) AS "RECORDS" FROM CUSTOMERS

Date Expressions
SELECT CURRENT_TIMESTAMP;

Another date expression is as shown below:
SELECT GETDATE()

SQL create database
The SQL CREATE DATABASE statement is used to create a new SQL database.
Syntax
The basic syntax of this CREATE DATABASE statement is as follows:
Always the database name should be unique within the RDBMS.
Example
If you want to create a new database <testDB>, then the CREATE DATABASE statement would be as shown below:
CREATE DATABASE testDB
Make sure you have the admin privilege before creating any database. Once a database is created, you can check it in the list of databases as follows:
SHOW DATABASES;

SQL drop database
The SQL DROP DATABASE statement is used to drop an existing database in SQL schema.
Syntax
The basic syntax of DROP DATABASE statement is as follows:
DROP DATABASE DatabaseName
Always the database name should be unique within the RDBMS.
Example
If you want to delete an existing database <testDB>, then the DROP DATABASE statement would be as shown below:
DROP DATABASE testDB;
NOTE: Be careful before using this operation because by deleting an existing database would result in loss of complete information stored in the database.
Make sure you have the admin privilege before dropping any database. Once a database is dropped, you can check it in the list of the databases as shown below:

```
EXEC sp_databases
```

SQL - SELECT Database, Use Statement
When you have multiple databases in your SQL Schema, then before starting your operation, you would need to select a database where all the operations would be performed.
The SQL USE statement is used to select any existing database in the SQL schema.
Sy ntax
The basic syntax of the USE statement is as shown below:
USE [DatabaseName]
Always the database name should be unique within the RDBMS.

SQL -CREATE Table

Creating a basic table involves naming the table and defining its columns and each column's data type.
The SQL CREATE TABLE statement is used to create a new table.
Syntax
The basic syntax of the CREATE TABLE statement is as follows:

```
CREATE TABLE table_name(
columnl datatype,
column2 datatype,
column3 datatype,
columnN datatype,
PRIMARY KEY( one or more columns )
);
```

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.
Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with the following example.
A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. You can check the complete details at Create Table Using another Table.
Example

The following code block is an example, which creates a CUSTOMERS table with an ID as a primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table:

```
CREATE TABLE CUSTOMERS(
ID INT                    NOT     NULL,
NAME VARCHAR (20)  NOT     NULL,
AGE INT                           NOT     NULL,
ADDRESS CHAR (25) ,
SALARY DECIMAL (18,           2),
PRIMARY KEY (ID)
);
```

You can verify if your table has been created successfully by looking at the message displayed by the SQL server.
Now, you have CUSTOMERS table available in your database which you can use to store the required information related to customers.

SQL - Creating a Table from an Existing Table
A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. The new table has the same column definitions. All columns or specific columns can be selected. When you will create a new table using the existing table, the new table would be populated using the existing values in the old table.
Syntax
The basic syntax for creating a table from another table is as follows:

```
SELECT column1, column2 ....
INTO    [dbo].[new]
FROM    [dbo].[old]
```

Here, column1, column2... are the fields of the existing table and the same would be used to create fields of the new table.
Example

```
SELECT * into salary  FROM CUSTOMERS
```

SQL DROP TABLE
The SQL DROP TABLE statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.
NOTE: You should be very careful while using this command because once a table is deleted then all the information available in that table will also be lost forever.
Syntax
The basic syntax of this DROP TABLE statement is as follows:
DROP TABLE table_name;
Example

This means that the CUSTOMERS table is available in the database, so let us now drop it as shown below.
DROP TABLE CUSTOMERS;

SQL - INSERT Query
The SQL INSERT INTO Statement is used to add new rows of data to a table in the database.

```
INSERT INTO { database_name.schema_name.table_name | schema_name.table_name
| table_name }
    [ ( column_name [ ,...n ] ) ]
    {
      VALUES ( { NULL | expression } )
      | SELECT <select_criteria>
    }
    [ OPTION ( <query_option> [ ,...n ] ) ]
[;]
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Example

The following statements would create six records in the CUSTOMERS table
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00);
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2, 'Khilan', 25, 'Delhi', 1500.00);
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (3, 'kaushik', 23, 'Kota', 2000.00);
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00);
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00);

You can create a record in the CUSTOMERS table by using the second syntax as shown below.
INSERT INTO CUSTOMERS
VALUES (7, 'Muffy', 24, 'Indore', 10000.00);

SQL SELECT Query
The SQL SELECT statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

SELECT *select_list* [INTO *new_table*]

[FROM *table_source*] [WHERE *search_condition*]

[GROUP BY *group_by_expression*]

[HAVING *search_condition*]

[ORDER BY *order_expression* [ASC | DESC]]

The following code is an example, which would fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.
SELECT ID, NAME, SALARY FROM CUSTOMERS
If you want to fetch all the fields of the CUSTOMERS table, then you should use the following query.
SELECT * FROM CUSTOMERS

SQL WHERE Clausule
The SQL WHERE clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records.
The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement, etc., which we would examine in the subsequent chapters.
Syntax
The basic syntax of the SELECT statement with the WHERE clause is as shown below.
SELECT columnl, column2, columnN FROM table_name WHERE [condition]
You can specify a condition using the comparison or logical operators like >, <, =, LIKE, NOT, etc. The following examples would make this concept clear.
Example
SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE SALARY > 2000;
The following query is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table for a customer with the name Hardik.
Here, it is important to note that all the strings should be given inside single quotes (''). Whereas, numeric values should be given without any quote as in the above example.
SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE NAME = 'Hardik'

SQL AND & OR

The SQL AND & OR operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called as the conjunctive operators.
These operators provide a means to make multiple comparisons with different operators in the same SQL statement.
Syntax
The basic syntax of the AND operator with a WHERE clause is as follows:
SELECT columnl, column2, columnN FROM table_name
WHERE [conditionl] AND [condition2]...AND [conditionN];
Example

Following is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 and the age is less than 25 years.
SELECT ID, NAME, SALARY FROM CUSTOMERS
WHERE SALARY > 2000 AND age < 25;

The OR Operator

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
Syntax
The basic syntax of the OR operator with a WHERE clause is as follows:
SELECT columnl, column2, columnN FROM table_name
WHERE [conditionl] OR [condition2]...OR [conditionN]
You can combine N number of conditions using the OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, the only any ONE of the conditions separated by the OR must be TRUE.

Example
The following code block has a query, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 OR the age is less than 25 years.
SELECT ID, NAME, SALARY FROM CUSTOMERS
WHERE SALARY > 2000 OR age < 25;

SQL UPDATE Query

The SQL UPDATE Query is used to modify the existing records in a table. You can use the WHERE clause with the UPDATE query to update the selected rows, otherwise all the rows would be affected.
UPDATE [database_name . [schema_name] . | schema_name .] table_name
SET { column_name = { expression | NULL } } [,...n]
[FROM from_clause]
[WHERE <search_condition>]
[OPTION (LABEL = label_name)]
[;]
The following query will update the ADDRESS for a customer whose ID number is 6 in the table.
UPDATE CUSTOMERS SET ADDRESS = 'Pune'
If you want to modify all the ADDRESS and the SALARY column values in the CUSTOMERS table, you do not need to use the WHERE clause as the UPDATE query would be enough as shown in the following code block.
UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;

SQL DELETE Query

The SQL DELETE Query is used to delete the existing records from a table.
You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.
Syntax
The basic syntax of the DELETE query with the WHERE clause is as follows:
DELETE FROM table_name WHERE [condition];
You can combine N number of conditions using AND or OR operators.
The following code has a query, which will DELETE a customer, whose ID is 6.
DELETE FROM CUSTOMERS WHERE ID = 6;
If you want to DELETE all the records from the CUSTOMERS table, you do not need to use the WHERE clause and the DELETE query would be as follows:
DELETE FROM CUSTOMERS

The SQL LIKE clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator.

- The percent sign (%)
- The underscore (_)

The percent sign represents zero, one or multiple characters. The underscore represents a single number or character. These symbols can be used in combinations.
Syntax
The basic syntax of % and _ is as follows:

SELECT FROM table_name WHERE column LIKE 'XXXX%'
or
SELECT FROM table_name WHERE column LIKE '%XXXX%'
or
SELECT FROM table_name WHERE column LIKE 'XXXX_'
or
SELECT FROM table_name WHERE column LIKE '_XXXX'
or
SELECT FROM table_name WHERE column LIKE ' XXXX '
You can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.
Example
The following table has a few examples showing the WHERE part having different LIKE clause with '%' and operators:

Statement	Description
WHERE SALARY LIKE '200%'	Finds any values that start with 200.
WHERE SALARY LIKE '%200%'	Finds any values that have 200 in any position.
WHERE SALARY LIKE '_00%'	Finds any values that have 00 in the second and third positions.
WHERE SALARY LIKE '2_%_%'	Finds any values that start with 2 and are at least 3 characters in length.
WHERE SALARY LIKE '%2'	Finds any values that end with 2.
WHERE SALARY LIKE '_2%3'	Finds any values that have a 2 in the second position and end with a 3.
WHERE SALARY LIKE '2 3'	Finds any values in a five-digit number that start with 2 and end with 3.

Following is an example, which would display all the records from the CUSTOMERS table, where the SALARY starts with 200.
Select * from CUSTOMERS WHERE SALARY LIKE '200%';

SQL TOP
The SQL TOP clause is used to fetch a TOP N number or X percent records from a table.
Note: All the databases do not support the TOP clause. For example, MySQL supports the LIMIT clause to fetch a limited number of records, while Oracle uses the ROWNUM command to fetch a limited number of records.
Syntax
The basic syntax of the TOP clause with a SELECT statement would be as follows.
SELECT TOP number|percent column_name(s) FROM table_name WHERE [condition]
The following query is an example on the SQL server, which would fetch the top 3 records from the CUSTOMERS table.
SELECT TOP 3 * FROM CUSTOMERS;
The SQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.
The following code block has an example, which would sort the result in an ascending order by the NAME and the SALARY.
SELECT * FROM CUSTOMERS ORDER BY NAME,
The following code block has an example, which would sort the result in the descending order by NAME.
SELECT * FROM CUSTOMERS ORDER BY NAME DESC;

SQL GROUP BY
The SQL GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax
The basic syntax of a GROUP BY clause is shown in the following code block. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.
SELECT columnl, column2 FROM table_name WHERE [conditions]
GROUP BY columnl, column2 ORDER BY columnl, column2
If you want to know the total amount of the salary on each customer, then the GROUP BY query would be as follows.
SELECT NAME, SUM(SALARY) FROM CUSTOMERS GROUP BY NAME;
Now again, if you want to know the total amount of salary on each customer, then the GROUP BY query would be as follows:
SELECT NAME, SUM(SALARY) FROM CUSTOMERS GROUP BY NAME;

SQL DISTINCT
The SQL DISTINCT keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.
There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only those unique records instead of fetching duplicate records.
First, let us see how the following SELECT query returns the duplicate salary records.
SELECT SALARY FROM CUSTOMERS ORDER BY SALARY
SELECT DISTINCT SALARY FROM CUSTOMERS ORDER BY SALARY

SQL Server @@ Keywords (Transact-SQL) in SQL Server

@@CONNECTIONS
Returns the number of attempted connections, either successful or unsuccessful since SQL Server was last started.
@@MAX_CONNECTIONS is the maximum number of connections allowed simultaneously to the server.
@@CONNECTIONS is incremented with each login attempt, therefore @@CONNECTIONS can be greater than @@MAX_CONNECTIONS.

@@CPU_BUSY
Returns the time that SQL Server has spent working since it was last started. Result is in CPU time increments, or "ticks," and is cumulative for all CPUs, so it may exceed the actual elapsed time.

@@CURSOR_ROWS
Returns the number of qualifying rows currently in the last cursor opened on the connection.

@@DATEFIRST
Returns the current value, for a session, of SET DATEFIRST.
SET DATEFIRST specifies the first day of the week. The U.S. English default is 7, Sunday.

@@DBTS
@@DBTS returns the last-used timestamp value of the current database. A new timestamp value is generated when a row with a timestamp column is inserted or updated.

@@ERROR
Returns the error number for the last Transact-SQL statement executed.

@@FETCH_STATUS
Returns the status of the last cursor FETCH statement issued against any cursor currently opened by the connection.

@@IDENTITY
Is a system function that returns the last-inserted identity value.

@@IDLE
Returns the time that SQL Server has been idle since it was last started. The result is in CPU time increments, or "ticks," and is cumulative for all CPUs, so it may exceed the actual elapsed time.

@@IO_BUSY
Returns the time that SQL Server has spent performing input and output operations since SQL Server was last started. The result is in CPU time increments ("ticks"), and is cumulative for all CPUs, so it may exceed the actual elapsed time.

@@LANGID
Returns the local language identifier (ID) of the language that is currently being used.

@@LANGUAGE
Returns the name of the language currently being used.

@@LOCK_TIMEOUT
SET LOCK_TIMEOUT allows an application to set the maximum time that a statement waits on a blocked resource. When a statement has waited longer than the LOCK_TIMEOUT setting, the blocked statement is automatically canceled, and an error message is returned to the application.
@@LOCK_TIMEOUT returns a value of -1 if SET LOCK_TIMEOUT has not yet been run in the current session.

@@MAX_CONNECTIONS
Returns the maximum number of simultaneous user connections allowed on an instance of SQL Server. The number returned is not necessarily the number currently configured.

@@MAX_PRECISION
Returns the precision level used by decimal and numeric data types as currently set in the server.
By default, the maximum precision returns 38.

@@NESTLEVEL
Returns the nesting level of the current stored procedure execution (initially 0) on the local server.
When @@NESTLEVEL is executed within a Transact-SQL string, the value returned is 1 + the current nesting level. When @@NESTLEVEL is executed dynamically by using sp_executesql the value returned is 2 + the current nesting level.

@@OPTIONS
Returns information about the current SET options.

@@PACK_RECEIVED
Returns the number of input packets read from the network by SQL Server since it was last started.

@@PACK_SENT
Returns the number of output packets written to the network by SQL Server since it was last started.

@@PACKET_ERRORS

Returns the number of network packet errors that have occurred on SQL Server connections since SQL Server was last started.

@@PROCID
Returns the object identifier (ID) of the current Transact-SQL module. A Transact-SQL module can be a stored procedure, user-defined function, or trigger. @@PROCID cannot be specified in CLR modules or the in-process data access provider.

@@REMSERVER
@@REMSERVER enables a stored procedure to check the name of the database server from which the procedure is run.

@@ROWCOUNT
Returns the number of rows affected by the last statement. If the number of rows is more than 2 billion, use ROWCOUNT_BIG.

@@SERVERNAME
Returns the name of the local server that is running SQL Server.

@@SERVICENAME
Returns the name of the registry key under which SQL Server is running. @@SERVICENAME returns 'MSSQLSERVER' if the current instance is the default instance; this function returns the instance name if the current instance is a named instance.

@@SPID
Returns the session ID of the current user process.
@@SPID can be used to identify the current user process in the output of sp_who.

@@TEXTSIZE
Returns the current value of the TEXTSIZE option.

@@TIMETICKS
Returns the number of microseconds per tick.
The amount of time per tick is computer-dependent. Each tick on the operating system is 31.25 milliseconds, or one thirty-second of a second.

@@TOTAL_ERRORS
Returns the number of disk write errors encountered by SQL Server since SQL Server last started.

@@TOTAL_READ
Returns the number of disk reads, not cache reads, by SQL Server since SQL Server was last started.

@@TOTAL_WRITE
Returns the number of disk writes by SQL Server since SQL Server was last started.

@@TRANCOUNT
Returns the number of active transactions for the current connection.

The BEGIN TRANSACTION statement increments @@TRANCOUNT by 1. ROLLBACK TRANSACTION decrements @@TRANCOUNT to 0, except for ROLLBACK TRANSACTION savepoint_name, which does not affect @@TRANCOUNT. COMMIT TRANSACTION or COMMIT WORK decrement @@TRANCOUNT by 1.

@@VERSION
Returns version, processor architecture, build date, and operating system for the current installation of SQL Server.