**DATABASE**

A database is a collection of [information](http://searchsqlserver.techtarget.com/definition/information) that is organized so that it can be easily accessed, managed and updated.

Data is organized into rows, columns and tables, and it is indexed to make it easier to find relevant information. Data gets updated, expanded and deleted as new information is added.

**TABLE**

A database table is composed of records(horizontal rows) and fields(vertical columns -identifiable by name) that hold data

Each table in a database holds data about a different, but related, subject.

A **table** has a specified number of columns, but can have any number of rows.

**Fields(columns)**

**column** is a set of data values of a particular simple type, one for each row of the table

Columns define the data in a table, while rows populate data into the table. The **columns**provide the structure according to which the rows are composed.

Each row is identified by one or more values appearing in a particular column subset. The columns subset which uniquely identifies a row is called the [primary key](https://en.wikipedia.org/wiki/Primary_key).

**Records(rows)**

a **row** also called a record or tuple represents a single, implicitly structured data item in a table. Each row in a table represents a set of related data

## **JOIN**

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

**INNER JOIN Syntax**

SELECT table1.column1, table2.column2...

FROM table1

INNER JOIN table2

ON table1.common\_field = table2.common\_field;

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate  
FROM Orders  
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;

Let's look at a selection from the "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **CustomerID** | **OrderDate** |
| 10308 | 2 | 1996-09-18 |
| 10309 | 37 | 1996-09-19 |
| 10310 | 77 | 1996-09-20 |

Then, look at a selection from the "Customers" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mexico |

|  |  |  |
| --- | --- | --- |
| **OrderID** | **CustomerName** | **OrderDate** |
| 10308 | Ana Trujillo Emparedados y helados | 9/18/1996 |
| 10365 | Antonio Moreno Taquería | 11/27/1996 |
| 10383 | Around the Horn | 12/16/1996 |
| 10355 | Around the Horn | 11/15/1996 |
| 10278 | Berglunds snabbköp | 8/12/1996 |

Almost every **join** is an **equijoin**, because the condition for matching rows is based on the equality of two values—one from each of the tables being joined. So that's what makes it an **equijoin**: the ON condition is equality. This includes **inner** joins and all three types of outer joins.

SQL EQUI JOIN performs a JOIN against equality or matching column(s) values of the associated tables. An equal sign (=) is used as comparison operator in the where clause to refer equality.

You may also perform EQUI JOIN by using JOIN keyword followed by ON keyword and then specifying names of the columns along with their associated tables to check equality.

**Syntax:**

SELECT column\_list

FROM table1, table2....

WHERE table1.column\_name =

table2.column\_name;

or

SELECT \*

FROM table1

JOIN table2

[ON (join\_condition)]

1. Inner join can have equality (=) and other operators (like <,>,<>) in the join condition.
2. Equi join only have equality (=) operator in the join condition.
3. Equi join can be an Inner join, Left Outer join, Right Outer join

## **LEFT JOIN Keyword**

The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.

### **LEFT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
LEFT JOIN table2ON table1.column\_name=table2.column\_name;

**Note:** In some databases LEFT JOIN is called LEFT OUTER JOIN.

## **SQL RIGHT JOIN Keyword**

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

### **RIGHT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2ON table1.column\_name=table2.column\_name;

**Note:** In some databases RIGHT JOIN is called RIGHT OUTER JOIN.

Repeating data results in repeating

## **The SQL MIN() and MAX() Functions**

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

### **MIN() Syntax**

SELECT MIN(column\_name)  
FROM table\_name  
WHERE condition;

### **MAX() Syntax**

SELECT MAX(column\_name)  
FROM table\_name  
WHERE condition;

We'll use the same Order table as before.

|  |  |  |  |
| --- | --- | --- | --- |
| OrderId | CustomerId | Total | OrderDate |
| 1 | 321 | 10 | 1/2/2014 |
| 2 | 455 | 40 | 3/2/2014 |
| 3 | 456 | 20 | 3/10/2014 |

SELECT MAX(Total) FROM Order

SELECT MAX(Total) FROM Order

WHERE OrderDate BETWEEN '3/1/2014' AND '3/31/2014'

ANSWER:40

SELECT MIN(Total) FROM Order

ANS:10

SELECT MIN(Total) FROM Order

WHERE OrderDate BETWEEN '3/1/2014' AND '3/31/2014'

ANS:20

## **The SQL COUNT(), AVG() and SUM() Functions**

The COUNT() function returns the number of rows that matches a specified criteria.

The AVG() function returns the average value of a numeric column.

The SUM() function returns the total sum of a numeric column.

### **COUNT() Syntax**

SELECT COUNT(column\_name)  
FROM table\_name  
WHERE condition;

### **AVG() Syntax**

SELECT AVG(column\_name)  
FROM table\_name  
WHERE condition;

### **SUM() Syntax**

SELECT SUM(column\_name)  
FROM table\_name  
WHERE condition;

## **GROUP BY Statement**

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

### **GROUP BY Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)ORDER BY column\_name(s);

 this example, we have a table called employees with the following data:

| employee\_number | last\_name | first\_name | salary | dept\_id |
| --- | --- | --- | --- | --- |
| 1001 | Smith | John | 62000 | 500 |
| 1002 | Anderson | Jane | 57500 | 500 |
| 1003 | Everest | Brad | 71000 | 501 |
| 1004 | Horvath | Jack | 42000 | 501 |

Enter the following SQL statement:

[Try It](https://www.techonthenet.com/sql/group_by_try_sql.php)

SELECT dept\_id, SUM(salary) AS total\_salaries

FROM employees

GROUP BY dept\_id;

There will be 2 records selected. These are the results that you should see:

| dept\_id | total\_salaries |
| --- | --- |
| 500 | 119500 |
| 501 | 113000 |

## Using GROUP BY with the COUNT function

Let's look at how to use the GROUP BY clause with the [COUNT function](https://www.techonthenet.com/sql/count.php) in SQL.

In this example, we have a table called products with the following data:

| product\_id | product\_name | category\_id |
| --- | --- | --- |
| 1 | Pear | 50 |
| 2 | Banana | 50 |
| 3 | Orange | 50 |
| 4 | Apple | 50 |
| 5 | Bread | 75 |
| 6 | Sliced Ham | 25 |
| 7 | Kleenex | NULL |

Enter the following SQL statement:

[Try It](https://www.techonthenet.com/sql/group_by_try_sql.php)

SELECT category\_id, COUNT(\*) AS total\_products

FROM products

WHERE category\_id IS NOT NULL

GROUP BY category\_id

ORDER BY category\_id;

There will be 3 records selected. These are the results that you should see:

| category\_id | total\_products |
| --- | --- |
| 25 | 1 |
| 50 | 4 |
| 75 | 1 |

## Using GROUP BY with the MIN function

Let's next look at how to use the GROUP BY clause with the [MIN function](https://www.techonthenet.com/sql/min.php) in SQL.

In this example, we will use the employees table again that is populated the following data:

| employee\_number | last\_name | first\_name | salary | dept\_id |
| --- | --- | --- | --- | --- |
| 1001 | Smith | John | 62000 | 500 |
| 1002 | Anderson | Jane | 57500 | 500 |
| 1003 | Everest | Brad | 71000 | 501 |
| 1004 | Horvath | Jack | 42000 | 501 |

Enter the following SQL statement:

[Try It](https://www.techonthenet.com/sql/group_by_try_sql.php)

SELECT dept\_id, MIN(salary) AS lowest\_salary

FROM employees

GROUP BY dept\_id;

There will be 2 records selected. These are the results that you should see:

| dept\_id | lowest\_salary |
| --- | --- |
| 500 | 57500 |
| 501 | 42000 |

Enter the following SQL statement:

[Try It](https://www.techonthenet.com/sql/group_by_try_sql.php)

SELECT dept\_id, MAX(salary) AS highest\_salary

FROM employees

GROUP BY dept\_id;

There will be 2 records selected. These are the results that you should see:

| dept\_id | highest\_salary |
| --- | --- |
| 500 | 62000 |
| 501 | 71000 |

## **The SQL WHERE Clause**

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified condition.

### **WHERE Syntax**

SELECT column1,column2, ...  
FROM table\_name  
WHERE condition;

**Note:** The WHERE clause is not only used in SELECT statement, it is also used in UPDATE, DELETE statement, etc.!

| customer\_id | last\_name | first\_name | favorite\_website |
| --- | --- | --- | --- |
| 4000 | Jackson | Joe | techonthenet.com |
| 5000 | Smith | Jane | digminecraft.com |
| 6000 | Ferguson | Samantha | bigactivities.com |
| 7000 | Reynolds | Allen | checkyourmath.com |
| 8000 | Anderson | Paige | NULL |
| 9000 | Johnson | Derek | techonthenet.com |

Now enter the following SQL statement:

[Try It](https://www.techonthenet.com/sql/where_try_sql.php)

SELECT \*

FROM customers

WHERE favorite\_website = 'techonthenet.com'

AND customer\_id > 6000;

There will be 1 record selected. These are the results that you should see:

| customer\_id | last\_name | first\_name | favorite\_website |
| --- | --- | --- | --- |
| 9000 | Johnson | Derek | techonthenet.com |

Let's use the products table again for this example.

| product\_id | product\_name | category\_id |
| --- | --- | --- |
| 1 | Pear | 50 |
| 2 | Banana | 50 |
| 3 | Orange | 50 |
| 4 | Apple | 50 |
| 5 | Bread | 75 |
| 6 | Sliced Ham | 25 |
| 7 | Kleenex | NULL |

Now enter the following SQL statement:

[Try It](https://www.techonthenet.com/sql/where_try_sql.php)

SELECT \*

FROM products

WHERE (product\_id > 3 AND category\_id = 75)

OR (product\_name = 'Pear');

There will be 2 records selected. These are the results that you should see:

| product\_id | product\_name | category\_id |
| --- | --- | --- |
| 1 | Pear | 50 |
| 5 | Bread | 75 |

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

### **HAVING Syntax**

SELECT expression1, expression2, ... expression\_n,

aggregate\_function (aggregate\_expression)

FROM tables

[WHERE conditions]

GROUP BY expression1, expression2, ... expression\_n

HAVING condition;

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT      orderID, COUNT(productID) products  FROM      orderdetails  GROUP BY orderID  HAVING products > 5; |

### SQL HAVING COUNT

SELECT department, COUNT(\*) AS "Number of employees"

FROM employees

WHERE salary > 25000

GROUP BY department

HAVING COUNT(\*) > 10;

## **PRIMARY KEY Constraint**

The PRIMARY KEY constraint uniquely identifies each record in a database table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

**MySQL:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    PRIMARY KEY (ID)  
);

**SQL Server / Oracle / MS Access:**

CREATE TABLE Persons (  
    ID int NOT NULL PRIMARY KEY,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
);

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName)  
);

**Note:** In the example above there is only ONE PRIMARY KEY (PK\_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

## **SQL PRIMARY KEY on ALTER TABLE**

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

**MySQL / SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
ADD PRIMARY KEY (ID);

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
ADD CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName);

**Note:** If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

## **DROP a PRIMARY KEY Constraint**

To drop a PRIMARY KEY constraint, use the following SQL:

**MySQL:**

ALTER TABLE Persons  
DROP PRIMARY KEY;

**SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
DROP CONSTRAINT PK\_Person;

## **FOREIGN KEY Constraint**

A FOREIGN KEY is a key used to link two tables together.

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

Look at the following two tables:

"Persons" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **PersonID** | **LastName** | **FirstName** | **Age** |
| 1 | Hansen | Ola | 30 |
| 2 | Svendson | Tove | 23 |
| 3 | Pettersen | Kari | 20 |

"Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **OrderNumber** | **PersonID** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

**MySQL:**

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)  
);

**SQL Server / Oracle / MS Access:**

CREATE TABLE Orders (  
    OrderID int NOT NULL PRIMARY KEY,  
    OrderNumber int NOT NULL,  
    PersonID int FOREIGN KEY REFERENCES Persons(PersonID)  
);

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    CONSTRAINT FK\_PersonOrder FOREIGN KEY (PersonID)  
    REFERENCES Persons(PersonID)  
);

## **SQL FOREIGN KEY on ALTER TABLE**

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

**MySQL / SQL Server / Oracle / MS Access:**

ALTER TABLE Orders  
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

ALTER TABLE Orders  
ADD CONSTRAINT FK\_PersonOrder  
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

## **DROP a FOREIGN KEY Constraint**

To drop a FOREIGN KEY constraint, use the following SQL:

**MySQL:**

ALTER TABLE Orders  
DROP FOREIGN KEY FK\_PersonOrder;

**SQL Server / Oracle / MS Access:**

ALTER TABLE Orders  
DROP CONSTRAINT FK\_PersonOrder;

Duplicated records in a table.

U might have come across a situation like there are number of duplicated records in a table.

Now Its Ur duty to remove duplicacy of data from the table, So the first step would be to find out duplicate records against each column.

Suppose our tablename is masterorder with column orderid for which we are goin to find duplicate records.

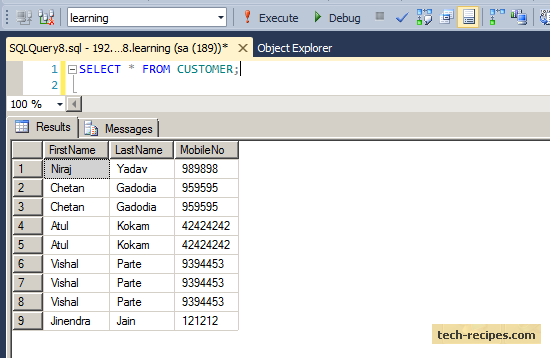
SELECT orderid, COUNT(\*) TotalCount

FROM masterorder

GROUP BY orderid

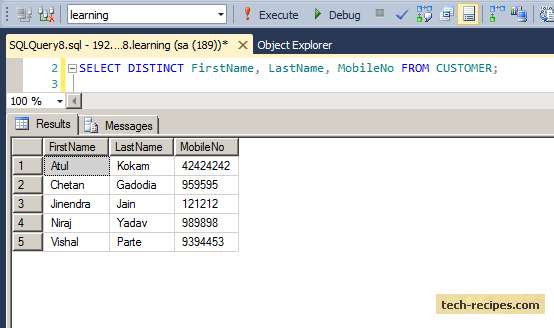
HAVING COUNT(\*) > 1

There are various times when we need to find duplicate records in SQL Server. It is possible to find duplicates using **DISTINCT, ROW NUMBER as well as the GROUP BY** approach.



Using the **DISTINCT**approach, we can quickly get unique rows in a table.

SELECT DISTINCT FirstName, LastName, MobileNo FROM CUSTOMER;



However, this does not show how many times a row has been duplicated. Using the GROUP BY approach, we can find this.

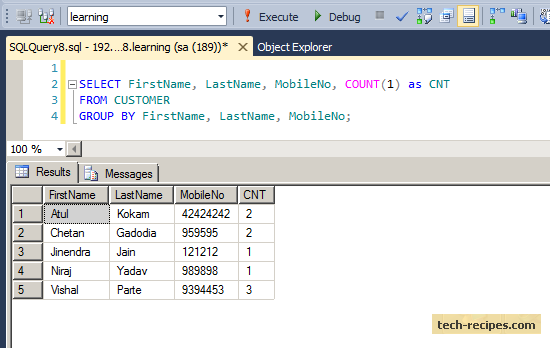
**Finding Duplicates Using GROUP BY**

Adding grouping and a counting field to our display of FirstName, LastName and MobileNo combination shows how many times each customer’s name appears.

SELECT FirstName, LastName, MobileNo, COUNT(1) as CNT

FROM CUSTOMER

GROUP BY FirstName, LastName, MobileNo;



**GROUP BY will show just one record for each combination of FirstName, LastName and MobileNo.**

The count CNT shows how many times the row has been duplicated.  
CNT = 1 indicates that row appears only once.

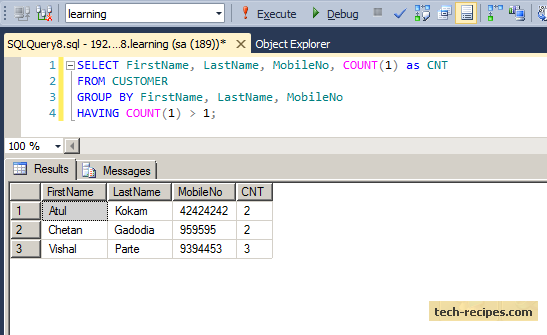
Let us filter out using the **Having clause** to exclude rows that appear only once.

SELECT FirstName, LastName, MobileNo, COUNT(1) as CNT

FROM CUSTOMER

GROUP BY FirstName, LastName, MobileNo

HAVING COUNT(1) > 1;



**how to delete duplicate rows**

**stored procedure**

A stored procedure is a set of [Structured Query Language (SQL)](http://searchsqlserver.techtarget.com/definition/SQL) statements with an assigned name

A stored procedure is nothing more than prepared SQL code that you save so you can reuse the code over and over again.  So if you think about a query that you write over and over again, instead of having to write that query each time you would save it as a stored procedure and then just call the stored procedure to execute the SQL code that you saved as part of the stored procedure.

In addition to running the same SQL code over and over again you also have the ability to pass parameters to the stored procedure, so depending on what the need is the stored procedure can act accordingly based on the parameter values that were passed.

A stored procedure provides an important layer of security between the [user interface](http://searchmicroservices.techtarget.com/definition/user-interface-UI)and the database. It supports security through data access controls because [end users](http://whatis.techtarget.com/definition/end-user) may enter or change data, but do not write procedures. A stored procedure preserves [data integrity](http://searchdatacenter.techtarget.com/definition/integrity) because information is entered in a consistent manner. It improves productivity because statements in a stored procedure only must be written once.

Stored procedures offer advantages over embedding[queries](http://searchsqlserver.techtarget.com/definition/query) in a [graphical user interface (GUI)](http://searchwindevelopment.techtarget.com/definition/GUI). Since stored procedures are modular, it is easier to troubleshoot when a problem arises in an application. Stored procedures are also tunable, which eliminates the need to modify the GUI[source code](http://searchmicroservices.techtarget.com/definition/source-code) to improve its performance. It's easier to code stored procedures than to build a query through a GUI.

Use of stored procedures can reduce network traffic between clients and [servers](http://whatis.techtarget.com/definition/server), because the commands are executed as a single batch of code. This means only the call to execute the procedure is sent over a [network](http://searchnetworking.techtarget.com/definition/network), instead of every single line of code being sent individually.

Like functions in programming languages, SQL Server user-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value. The return value can either be a single scalar value or a result set.

They are developed in [Transact-SQL (T-SQL)](http://searchsqlserver.techtarget.com/definition/T-SQL) or a reference to [Microsoft](http://searchwindowsserver.techtarget.com/definition/Microsoft)

## **User-defined functions**

Why use them?

* They allow modular programming.

You can create the function once, store it in the database, and call it any number of times in your program. User-defined functions can be modified independently of the program source code.

* They allow faster execution.

Similar to stored procedures, Transact-SQL user-defined functions reduce the compilation cost of Transact-SQL code by caching the plans and reusing them for repeated executions. This means the user-defined function does not need to be reparsed and reoptimized with each use resulting in much faster execution times.

CLR functions offer significant performance advantage over Transact-SQL functions for computational tasks, string manipulation, and business logic. Transact-SQL functions are better suited for data-access intensive logic.

* They can reduce network traffic.

An operation that filters data based on some complex constraint that cannot be expressed in a single scalar expression can be expressed as a function. The function can then invoked in the WHERE clause to reduce the number or rows sent to the client.

### **Stored procedure vs. function**

Stored procedures and [functions](http://whatis.techtarget.com/definition/function) can be used to accomplish the same task. Both can be custom-defined as part of any application, but

functions are designed to send their output to a query or T-SQL statement.

Stored procedures are designed to return outputs to the application,

while a user-defined function returns table variables and cannot change the server environment or operating system environment.

***indexing***

*Sooner or later there comes a moment when database performance is no longer satisfactory. One of the very first things you should turn to when that happens is database indexing.*

in general, a database index is a data structure used to improve queries execution time

A database index allows a query to efficiently retrieve data from a database.  Indexes are related to specific tables and consist of one or more keys.  A table can have more than one index built from it.  The keys are a fancy term for the values we want to look up in the index.  The keys are based on the tables’ columns.  By comparing keys to the index it is possible to find one or more database records with the same value.

Since an index drastically speeds up data retrieval, it is essential the correct indexes are defined for each table.  Missing indexes won’t be noticed for small databases, but rest assured, once your tables grow in size, queries will take much longer.

Indexes are used to retrieve data from the database very fast. The users cannot see the indexes, they are just used to speed up searches/queries.

**Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

### **CREATE INDEX Syntax**

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX index\_name  
ON table\_name (column1, column2, ...);

### **CREATE UNIQUE INDEX Syntax**

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX index\_name  
ON table\_name (column1, column2, ...);

**Note:** The syntax for creating indexes varies among different databases. Therefore: Check the syntax for creating indexes in your database.

## **CREATE INDEX Example**

The SQL statement below creates an index named "idx\_lastname" on the "LastName" column in the "Persons" table:

CREATE INDEX idx\_lastname  
ON Persons (LastName);

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

CREATE INDEX idx\_pname  
ON Persons (LastName, FirstName);

## **DROP INDEX Statement**

The DROP INDEX statement is used to delete an index in a table.

**MS Access:**

DROP INDEX index\_name ON table\_name;

**SQL Server:**

DROP INDEX table\_name.index\_name;

**DB2/Oracle:**

DROP INDEX index\_name;

**MySQL:**

ALTER TABLE table\_nameDROP INDEX index\_name;

An index is defined by a field expression that you specify when you create the index. Typically, the field expression is a single field name, like EMP\_ID. An index created on the EMP\_ID field, for example, contains a sorted list of the employee ID values in the table. Each value in the list is accompanied by references to the records that contain that value. A database driver can use indexes to find records quickly. An index on the EMP\_ID field, for example, greatly reduces the time that the driver spends searching for a particular employee ID value. Consider the following Where clause: WHERE emp\_id = 'E10001' Without an index, the driver must search the entire database table to find those records having an employee ID of E10001. By using an index on the EMP\_ID field, however, the driver can quickly find those records. Indexes may improve the performance of SQL statements. You may not notice this improvement with small tables but it can be significant for large tables; however, there can be disadvantages to having too many indexes. Indexes can slow down the performance of some inserts, updates, and deletes when the driver has to maintain the indexes as well as the database tables. Also, indexes take additional disk space.