# Structured Query Language and Microsoft Access

## In This Lecture

- SQL
  - History: The SQL language
  - SQL, the relational model, and
  - CREATE TABLE
    - Primary Keys
    - Foreign Keys
    - Constraints
  - ALTER TABLE
  - INSERT  RECORD
  - UPDATE TABLE
  - DELETE RECORD

## Structured Query Language - SQL

Why to use SQL in addition to the Design View?

-not all SQL functions can be used from the Design View level

-SQL can be used in other applications (Excel, Word, Visual Basic)

-SQL is a standard query language which can be used outside the Access program

One can easily switch between Design View and SQL View

SQL is a procedure language which tells what to do, and not how to do.

## History

SQL_ Structured Query Language and was first created in 1970. used to be called SEQUEL- Structured English Query Language.

Implemented in a number of database management system (DBMS) platforms,

Rules for SQL vary slightly from one DBMS to another. Because of the variations of SQL, each DBMS refers to SQL using a distinct name that is specific to the DBMS. For example, the Oracle DBMS refers to SQL as PLSQL (Procedural Language extensions to SQL),

Microsoft SQL Server refers to SQL as Transact-SQL, and Microsoft Access refers to SQL as Access SQL.

# SQL Components

- The SQL is broken up into three components: DDL, DML, and DCL.
- The **Data Definition Language (DDL**) component is used to create tables and establish relationships among tables.
- The **Data Manipulation Language (DML)** component is used to manage the database by performing such operations as retrieving data, updating data, deleting data, and navigating through data.
- The **Data Control Language (DCL)** component is used to provide security features for the database.

**Note:** The DCL commands can only be executed in the Visual Basic environment of Microsoft Access. An error message will be returned if used through the Access SQL view user interface. Visual Basic is the host language for the Jet DBMS, which handles the translation of database queries into Access SQL.

## The Commands within each of these Components

| DDL | DML | DCL |
|---|---|---|
| CREATE TABLE | INSERT INTO | ALTER DATABASE |
| DROP TABLE | SELECT INTO | CREATE GROUP |
| ALTER TABLE | UPDATE | DROP GROUP |
| CREATE INDEX | DELETE | CREATE USER |
| | SELECT | ALTER USER |
| | UNION | DROP USER |
| | TRANSFORM | ADD USER |
| | PARAMETER | GRANT PRIVILEGE |
| | | REVOKE PRIVILEGE |

## SQL Syntax

- In order to implement SQL, you must follow a series of rules that state how SQL script must be scripted. These rules are referred to as *syntax*.
- The SQL language is made up of a series of keywords, statements, and clauses. The keywords, statements, and clauses are combined to enable users to create queries that extract meaningful data from the database. A *query* is a question or command posed to the database, and *keywords* are reserved words used within queries and SQL statements. Keywords are considered reserved because they cannot be used to name parts of the database. For example, you cannot use a keyword to name the database, tables, columns, or any other portion of the database.
- *Statements* combine keywords with data to form a database query, and a *clause* is a segment of an SQL statement. Since you cannot have an actual conversation with the database like you would a person, keywords, statements, and clauses help you convey what you need to accomplish.

## SQL in Microsoft Access

- Although you don’t actually need to know SQL to create and maintain databases in Microsoft Access, knowing SQL gives you an extra edge that many users overlook.
- Understanding the SQL language gives you more power and control over your database. You can create more powerful queries using SQL.
- For example, with SQL you can create tables, queries that pass through Access to an external server (pass- through queries), combined queries (unions), and nested queries (subqueries). Additionally, you’ll understand system- generated queries more fully, enabling you to manually edit Access-generated queries to create your own customized queries.

## Opening MS Access and Switching to SQL View

- Create a new database in Microsoft Access
- **Switching to SQL View:-** click **Queries** on the left, and then click the **New** button located near the top of the screen. When the New Query dialog box appears, select **Design View** and click **OK**.
- To switch to SQL view, use the View button and select **SQL View**. (Click the down arrow located on the View button to find the SQL View option.) This is the view in which you will type most of the SQL script
- Run button to execute script typed in SQL view

## Creating Tables and Inserting Records

Definitions

- CREATE TABLE — Keywords that are used to instruct the database to create a new table.
- ALTER TABLE — Keywords that are used to modify columns and constraints in an existing table.
- Constraints — Used to restrict values that can be inserted into a field and to establish referential integrity.
- Data type — Specifies the type of data a column can store.
- DELETE statement — Used to remove records from a table.

## Creating Tables

**CREATE TABLE** — Keywords that are used to instruct the database to create a new table.

**CREATE TABLE Syntax**

```
CREATE TABLE Tablename (
Columnname Datatype Field Size, [NULL | NOT NULL] [optional constraints]
);
```

The following SQL script creates a table named Toys:

.

```
CREATE TABLE Toys (
ToyID INTEGER,
ToyName CHAR (30),
    Price MONEY,  Description CHAR (40)
);
```

## Creating Tables and Data types

A *data type* specifies the type of data a column can store. For example, if you create a column that can only store numbers, you must assign it a specific data type that will only allow numbers to be stored in the column.

SQL view supports a variety of different data types

**Note:** When you create table and column names that contain spaces, enclose the names in brackets ([ ]). For example, the following script creates a table named Furniture with column names that contain spaces:

```
CREATE TABLE Furniture (
[Furniture ID] INTEGER,  [Furniture Name] CHAR (30),
  [Furniture Price] MONEY
);
```

## Common Microsoft Access data types

| Data Type | Description |
|---|---|
| **Numeric**: | |
| DECIMAL | An exact numeric data type that holds values from –10^28 – 1 to 10^28 – 1. |
| FLOAT | Stores double-precision floating-point values. |
| INTEGER | Also called INT. Stores long integers from –2,147,483,648 to 2,147,483,647. |
| REAL | Stores single-precision floating-point values. |
| SMALLINT | Stores integers from –32,768 to 32,767. |
| TINYINT | Stores integers from 0 to 255. |
| **String**: | |
| CHAR | A fixed-length data type that stores a combination of text and numbers up to 255 characters. |
| TEXT | A variable-length data type that stores a combination of text and numbers up to 255 characters. The length is determined by the Field size property. The string can contain any ASCII characters including letters, numbers, special characters, and nonprinting characters. |

# Constraints

**Constraints** enable you to further control how data is entered into a table and are used to restrict values that can be inserted into a field and to establish referential integrity

| Constraint | Description |
|---|---|
| NULL/NOT NULL | Used to indicate if a field can be left blank when records are entered into a table. |
| PRIMARY KEY | Used to uniquely identify every record in a table. |
| FOREIGN KEY | Used to link records of a table to the records of another table. |
| UNIQUE | Used to ensure that every value in a column is different. |

## Creating Tables and Inserting Records

### Example 1

Say you want to alter the Toys table script created earlier in the chapter. You want to add constraints that will ensure that every Toy ID is unique and that the ToyID, ToyName, and Price columns always contain values when new records are entered into the Toys table. Look at the following script:

```
CREATE TABLE Toys (
ToyID INTEGER CONSTRAINT ToyPk PRIMARY KEY, ToyName CHAR (30) NOT
NULL,
Price MONEY NOT NULL,
Description CHAR (40) NULL
);
```

This script creates a new table named Toys with four columns (ToyID, ToyName, Price, and Description). A primary key constraint is defined for the ToyID column and the NOT NULL constraint is defined for the ToyName and Price columns. The Description column contains a NULL constraint. Following is an explanation of the NULL/NOT NULL and primary key constraints.

## Creating Tables

- NULL/NOT NULL constraint: is used to indicate whether or not a field can be left blank when records are entered into a table.
- **Note:**In Microsoft Access, when you do not state NULL or NOT NULL during the creation of a column, it is automatically set to NULL
- PRIMARY KEY constraint: is used to uniquely identify every record in a table. The specification of a primary key ensures that there are no duplicate values in a column. Additionally, primary key fields are stored in ascending order and default to NOT NULL.
- FOREIGN KEY Constraint: used to link records of one table to the records of another. To define a FOREIGN KEY constraint same name must exist as a primary key in another table = referential integrity.
- The CONSTRAINT, REFERENCES, and PRIMARY KEY keywords are used to define the foreign key.
- The UNIQUE constraint is used to ensure that every value in a column is different. The Phonenumber column in the Create Manufacturers table contains a UNIQUE constraint The UNIQUE constraint is very similar to the PRIMARY KEY constraint; however, the UNIQUE constraint can be defined more than once in a single table, and a column defined as unique does not automatically default to NOT NULL.

**Example 2**: Say you want to link the Toys table in Example 1 to a new table named Manufacturers. Additionally, you want to ensure that all phone numbers entered into the Phonenumber column in the Manufacturers table are unique and that all updates and deletions made to the Manufacturers table affect corresponding records in the Toys table. Take a look at the following script:

```
CREATE TABLE Manufacturers (
ManufacturerID INTEGER CONSTRAINT ManfID PRIMARY KEY, ToyID
INTEGER NOT NULL,
CompanyName CHAR (50) NOT NULL, Address CHAR (50) NOT NULL,
City CHAR (20) NOT NULL,
State CHAR (2) NOT NULL,
Postalcode CHAR (5) NOT NULL, Areacode CHAR (3) NOT NULL,
Phonenumber CHAR (8) NOT NULL UNIQUE,
CONSTRAINT ToyFk FOREIGN KEY (ToyID) REFERENCES Toys (ToyID)
);
```

## Adding Constraints to Existing Tables

Constraints can also be added to tables that have already been created. To add a constraint to an existing table you must use the ALTER TABLE statement.

This statement is used to add or delete columns and constraints in an existing table. Following is the basic syntax to alter an existing table:

```
ALTER TABLE Tablename
ADD COLUMN ColumnName ColumnType (Size) ColumnConstraint | DROP
COLUMN ColumnName |
ADD CONSTRAINT ColumnConstraint | DROP CONSTRAINT
ColumnConstraint;
```

In the script, the ALTER TABLE keywords are used to specify the table name (Toys), and the ADD CONSTRAINT keywords are used to specify the name of the constraint (ToyNameUnique).

## Example 3

Say you want to add a UNIQUE constraint to the ToyName column in the Toys table. Look at the following script:

```
ALTER TABLE Toys
ADD CONSTRAINT ToyNameUnique UNIQUE (ToyName);
```

This SQL script uses the ALTER TABLE statement to add the UNIQUE constraint to the ToyName column in the Toys table. The ALTER TABLE keywords are used to specify the table name (Toys). The ADD CONSTRAINT keywords are used to specify the constraint name (ToyNameUnique), the type of constraint (UNIQUE), and the name of the column (ToyName) to add the constraint to.
To delete the UNIQUE constraint from the ToyName column in the Toys table, simply type the following:

```
ALTER TABLE Toys
DROP CONSTRAINT ToyNameUnique;
```

## Inserting Records

After you create a table, you can insert records into it using insert statements. Each insert statement inserts a single record into a table. Look at the following syntax for the insert statement:

```
INSERT INTO Tablename [(ColumnNames, ...)]  VALUES (values,
...);
```

Each insert statement contains the INSERT INTO and VALUES keywords. The INSERT INTO keywords are used to specify the table name and the column names to insert values into.

The VALUES keyword is used to specify the values to insert into a table. Take a look at Example 5, which inserts five columns into the Toys table.

## Inserting Records

**Example 5**

This example inserts five records into the Toys table created earlier in the chapter.

```
INSERT INTO Toys (ToyID, ToyName, Price, Description)
VALUES (1, 'ToyTrain1', 11.00, 'Red/blue battery powered train');

INSERT INTO Toys (ToyID, ToyName, Price, Description)
VALUES (2, 'ToyTrain2', 11.00, 'Green/red/blue battery powered  train');

INSERT INTO Toys (ToyID, ToyName, Price, Description)
VALUES (3, 'ElectricTrain', 15.00, 'Red/white AC/DC powered train');

INSERT INTO Toys (ToyID, ToyName, Price, Description)  VALUES (4,
'LivingDoll1', 12.00, 'Asian American Doll');

INSERT INTO Toys (ToyID, ToyName, Price, Description)  VALUES (5,
'LivingDoll2', 12.00, 'African American Doll');
```

**Note:** The preceding insert statements insert five records into the Toys table. Since each insert statement contains a closing semi- colon, it is easy to see where each statement begins and ends. Each insert statement inserts one record with four values.

As a side note, each time you execute an insert statement, Microsoft Access verifies the insertion of the new record by displaying a message/question that says:

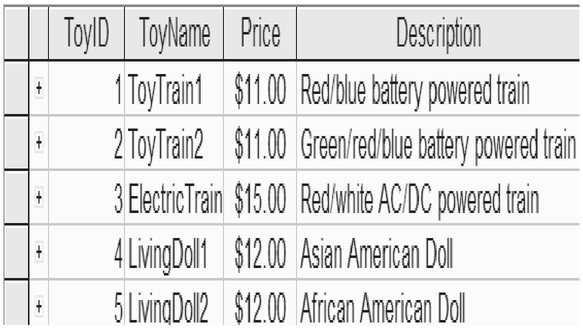
“You are about to append 1 row (s). Once you click yes, you can’t use the undo command to reverse the changes. Are you sure you want to append the selected rows?”

Each time you insert a new record, be sure to click Yes to this message.

Notice the insert statements that contain character strings enclosed in quotes. Whenever a table contains a column data type that accepts character strings, all character string values pertaining to the column must be enclosed in quotes. Since the ToyName and Description columns contain data types that accept character strings, the character string values in the insert statements are enclosed in quotes.

Type the following script to view the populated Toys table:

```
SELECT *
FROM Toys;
```



| | ToyID | ToyName | Price | Description |
|---|---|---|---|---|
| + | 1 | ToyTrain1 | $11.00 | Red/blue battery powered train |
| + | 2 | ToyTrain2 | $11.00 | Green/red/blue battery powered train |
| + | 3 | ElectricTrain | $15.00 | Red/white AC/DC powered train |
| + | 4 | LivingDoll1 | $12.00 | Asian American Doll |
| + | 5 | LivingDoll2 | $12.00 | African American Doll |

## Inserting Data without Specifying Column Names

INSERT statements can also be executed without the specifica- tion of column names. To execute an INSERT statement with- out typing the column names, specify the values in the same order that the columns appear in the table. Look at Example 6, which inserts an additional record into the Toys table.

### Example 6

Say you want to insert a complete record into the Toys table but you do not want to type the column names. Look at the following script:

```
INSERT INTO Toys
VALUES (6, 'Doll House', 17.00, 'Grand Town House');
```

The preceding script inserts one record containing four values into the Toys table. Because the values are typed in the same order in which the columns appear in the table, it is not necessary to type the column names.
As a side note, if you want to insert values into specific columns only, specify only the column names you want to insert values into. Next, specify values in the same order as they appear in your INSERT statement.

# Inserting NULL Values

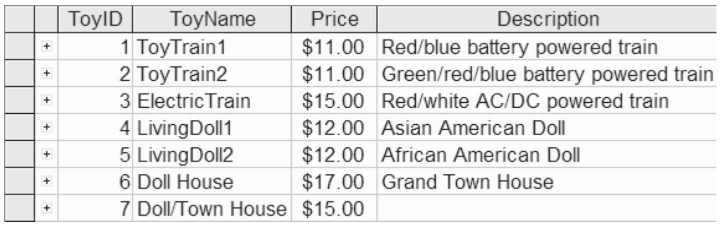
**Example 7**

Say you want to insert a record with a missing value. Take a look at the following script:

```
INSERT INTO Toys
VALUES (7, 'Doll/Town House', 15.00, NULL);
```

This script inserts one record containing three values into the Toys table. It inserts NULL for a missing value. Recall that NULL means no value. Figure 3-5 shows the Toys table after the insertion of the record containing the NULL value.
As a side note, you cannot insert NULL into columns that contain a NOT NULL constraint.

. Toys table containing a NULL value



| | | ToyID | ToyName | Price | Description |
|---|---|---|---|---|---|
| | + | 1 | ToyTrain1 | $11.00 | Red/blue battery powered train |
| | + | 2 | ToyTrain2 | $11.00 | Green/red/blue battery powered train |
| | + | 3 | ElectricTrain | $15.00 | Red/white AC/DC powered train |
| | + | 4 | LivingDoll1 | $12.00 | Asian American Doll |
| | + | 5 | LivingDoll2 | $12.00 | African American Doll |
| | + | 6 | Doll House | $17.00 | Grand Town House |
| | + | 7 | Doll/Town House | $15.00 | |

The following script copies the records from the Toys table into the ToysTest table:

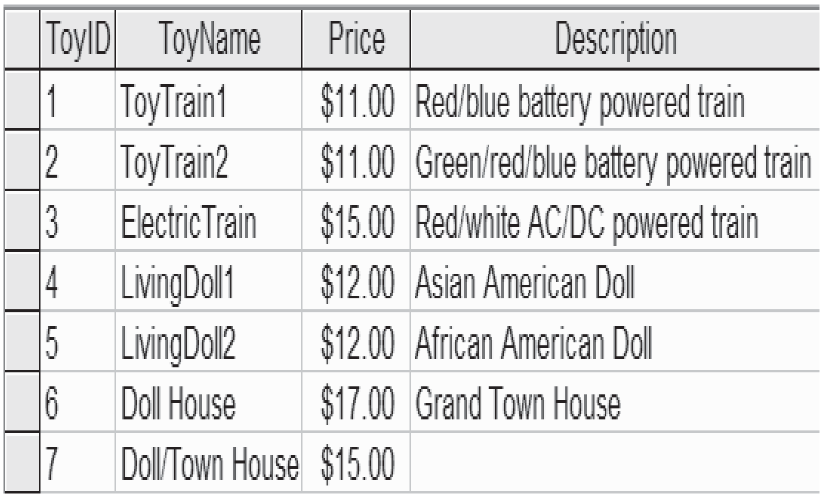
```
INSERT INTO ToysTest (ToyID, ToyName, Price, Description) SELECT
ToyID, ToyName, Price, Description
FROM Toys;
```

This script uses the INSERT INTO keywords to specify the table name and column names to insert records into.

The SELECT and FROM keywords are used to specify the table name and column names from which to retrieve the records to insert.

The SELECT keyword is used to specify the column names from the Toys table and the FROM keyword is used to specify the Toys table.
As a side note, the ToysTest and Toys tables do not have to have the same column names, but they do have to have similar data types and field sizes. The Next table shows the populated ToysTest table.

**Populated ToysTest table**



| | ToyID | ToyName | Price | Description |
|---|---|---|---|---|
| | 1 | ToyTrain1 | $11.00 | Red/blue battery powered train |
| | 2 | ToyTrain2 | $11.00 | Green/red/blue battery powered train |
| | 3 | ElectricTrain | $15.00 | Red/white AC/DC powered train |
| | 4 | LivingDoll1 | $12.00 | Asian American Doll |
| | 5 | LivingDoll2 | $12.00 | African American Doll |
| | 6 | Doll House | $17.00 | Grand Town House |
| | 7 | Doll/Town House | $15.00 | |

## Copying Records from One Table to an Existing Table

### Example 8

Sometimes it is necessary to populate a table with records from an existing table. Say, for example, you need to create a test table and you want to use data that is already stored in another table. Take a look at the following queries. The first one creates a new table named ToysTest and the second one copies the records from the Toys table to the ToysTest table:

```
CREATE TABLE ToysTest (
ToyID CHAR (7) CONSTRAINT ToyPk PRIMARY KEY, ToyName
CHAR (30) NOT NULL,
Price MONEY NOT NULL,
Description CHAR (40) NULL
);
```

This script creates a table named ToysTest. The ToysTest table contains the same data types and field sizes as the Toys table.

## Updating Records

The UPDATE statement is used to update records in a table. Look at the following syntax to update a table:

```
UPDATE Tablename
SET ColumnName = Value  WHERE Condition;
```

**Example 10**

What if you want to add a value to one of the records in the Toys table. Look at the following script:

```
UPDATE Toys
SET Description = ‘Town House’  WHERE ToyID = 7;
```

The preceding script inserts a value into one of the records stored in the Toys table. It uses the UPDATE keyword to spec- ify the table (Toys) to update. The SET keyword is used to specify the column (Description) to update and the value (Town House) to insert into the column. The WHERE keyword is used to set conditions on retrieved data. It is commonly referred to as the WHERE clause. You will learn more about clauses and the WHERE clause in Chapter 5. In this example, the WHERE keyword is used to specify value 7 in the ToyID column. Figure 3-8 shows the Toys table containing the new value.

## Deleting Records

The DELETE statement is used to remove records from a table. Look at the following delete syntax:

```
DELETE FROM Tablename  WHERE Condition
```

Example 11 shows how to delete a record from the Toys2 table.

**Example 11**

This example shows how to remove one record from the Toys2 table:

```
DELETE FROM Toys2  WHERE ToyID =  7;
```

## Retrieving Records : The SELECT Statement

- The *SELECT* statement is used to retrieve records from the database. Records retrieved from the database are often referred to as a *result set*.
- Every SELECT statement contains the SELECT keyword and the FROM keyword.
- Syntax for the SELECT statement:

```
SELECT Columnname(s)
FROM TableName(s);
```

**Example 1:**

```
SELECT ToyName, Price
FROM Toys;
```

## The SELECT Statement

- **Example 2**

```
SELECT *
FROM Toys;
```

- This script combines an asterisk (*) with the SELECT keyword, which tells the DBMS to select every column from a table.

**The ORDER BY Clause**

- The *ORDER BY* clause is often used in the SELECT statement to sort retrieved records in descending or ascending order.

**Example 3**

```
SELECT *
FROM Toys
ORDER BY ToyName DESC;
```

This script specifies the ToyName column after the ORDER BY keywords. The DESC keyword is specified after the column name and causes the DBMS to sort the values in the ToyName column in descending order

## The ORDER BY Clause

ASC used to sort in ascending order.

```
SELECT *
FROM Toys
ORDER BY ToyName ASC;
```

**Sorting Multiple Columns**

```
SELECT Lastname, Firstname FROM Employees
ORDER BY Lastname, Firstname;
```

## Handling Duplicate Values

- When tables contain duplicate column values, the DISTINCT, DISTINCTROW, TOP, and TOP PERCENT keywords are used to single out specific values among the duplicate values.

- **The DISTINCT Keyword**

  The *DISTINCT* keyword is used to display unique values in a column. In Access, this feature is determined by the Unique Values property of the query.

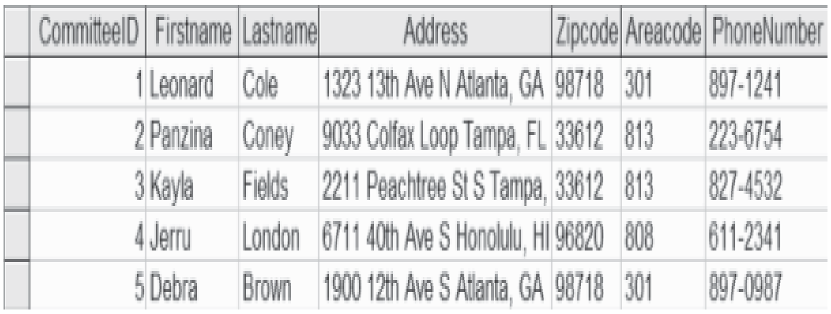
  ```
  SELECT DISTINCT Price
  FROM Toys;
  ```

  This causes the DBMS to display only the unique values in the Price column

## Creating an Alias

- An *alias* is an alternate name for a table or column. Aliases are created using the *AS* keyword

**Example**



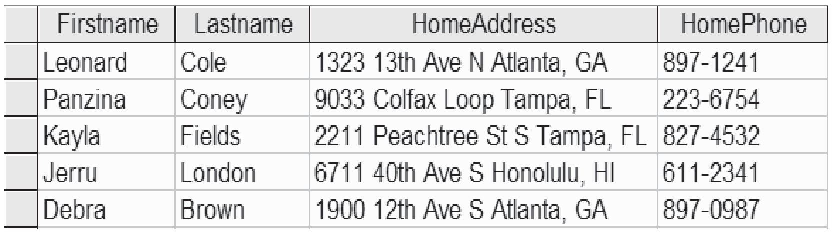
| CommitteeID | Firstname | Lastname | Address | Zipcode | Areacode | PhoneNumber |
|---|---|---|---|---|---|---|
| 1 | Leonard | Cole | 1323 13th Ave N Atlanta, GA | 98718 | 301 | 897-1241 |
| 2 | Panzina | Coney | 9033 Colfax Loop Tampa, FL | 33612 | 813 | 223-6754 |
| 3 | Kayla | Fields | 2211 Peachtree St S Tampa, | 33612 | 813 | 827-4532 |
| 4 | Jerru | London | 6711 40th Ave S Honolulu, HI | 96820 | 808 | 611-2341 |
| 5 | Debra | Brown | 1900 12th Ave S Atlanta, GA | 98718 | 301 | 897-0987 |

Committee2

## Creating an Alias

- Suppose you want to display the names, addresses, and phone numbers from the Committee2 table in Additionally, you want to create alternate column names for the Address and PhoneNumber columns. Look at the following script:

```
SELECT Firstname, Lastname, Address AS HomeAddress, PhoneNumber AS
    HomePhone
FROM Committee2;
```



| Firstname | Lastname | HomeAddress | HomePhone |
|---|---|---|---|
| Leonard | Cole | 1323 13th Ave N Atlanta, GA | 897-1241 |
| Panzina | Coney | 9033 Colfax Loop Tampa, FL | 223-6754 |
| Kayla | Fields | 2211 Peachtree St S Tampa, FL | 827-4532 |
| Jerru | London | 6711 40th Ave S Honolulu, HI | 611-2341 |
| Debra | Brown | 1900 12th Ave S Atlanta, GA | 897-0987 |

Results

## Concatenation

Used to Merging values or columns

Performed in MS Access using the ampersand (&) symbol or the plus (+) symbol.

The main difference between them is how they handle NULL fields.

When you use the ampersand (&), if one of the fields is NULL it is replaced by an empty string.

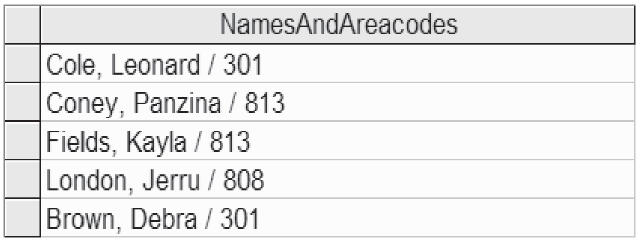
When using the plus (+) symbol, if one of the fields is NULL the result of the concatenation is NULL.

## Concatenation

Say you want to concatenate the names and area codes from the Committee2 table
Also, insert a comma between the last name and first name and a space on either side of a backslash character between the names and the area codes and display the concatenated columns under an alternate name.

**Script:**

```
SELECT Lastname & ','+"+ Firstname&'/'+ Areacode AS NamesAndAreacodes
FROM Committee2;
```



| | NamesAndAreacodes |
|---|---|
| | Cole, Leonard / 301 |
| | Coney, Panzina / 813 |
| | Fields, Kayla / 813 |
| | London, Jerru / 808 |
| | Brown, Debra / 301 |

Output (Results)

## Filtering Retrieved Records

- The *WHERE* clause used to filter retrieved records.
- Commonly used in the SELECT statement.
- Several different operators that can be used in the WHERE clause. Each operator falls into one of two categories:
  - Comparison
  - Logical.

**Comparison Operators**
Used to perform comparisons among expressions.
An *expression* is any data type that returns a value.

**Logical Operators**
Used to test for the truth of some condition

## Logical Operators

| Name | Symbol or Description |
|---|---|
| Greater Than | > |
| Greater Than or Equal To | >= |
| Equal | = |
| Less Than | < |
| Less Than or Equal To | <= |
| Not Equal | <> |
| BETWEEN | Used to determine whether a value of an expression falls within a specified range of values. |
| IN | Used to match conditions in a list of expressions. |
| LIKE | Used to match patterns in data. |
| IS NULL | Used to determine if a field contains data. |
| IS NOT NULL | Used to determine if a field does not contain data. |

## Logical Operators and Operator Precedence

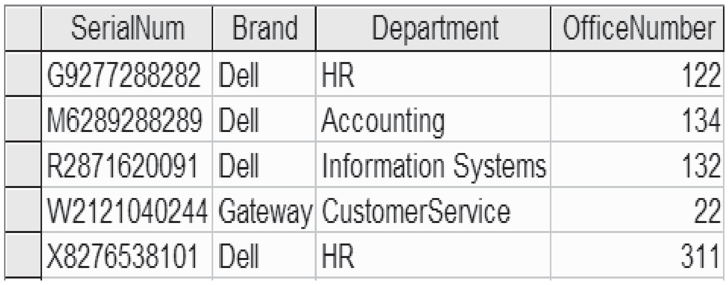
| Operator | Description |
|---|---|
| AND | Requires both expressions on either side of the AND operator to be true in order for data to be returned. |
| OR | Requires at least one expression on either side of the OR operator to be true in order for data to be returned. |
| NOT | Used to match any condition opposite of the one defined. |

**Operator Precedence:** When multiple operators are used in the WHERE clause, operator precedence determines the order in which operations are performed. The following list shows the order of evaluation among operators.

- =, >, <, >=, <=, <>
- AND, OR
- NOT
- AND
- BETWEEN, IN, LIKE

## Example 1

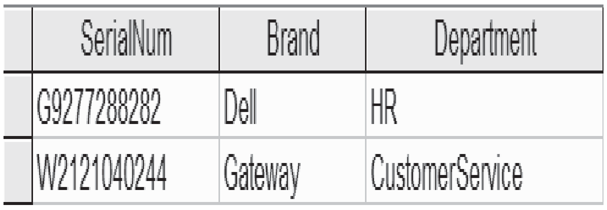
**Computer Table**



| SerialNum | Brand | Department | OfficeNumber |
|---|---|---|---|
| G9277288282 | Dell | HR | 122 |
| M6289288289 | Dell | Accounting | 134 |
| R2871620091 | Dell | Information Systems | 132 |
| W2121040244 | Gateway | CustomerService | 22 |
| X8276538101 | Dell | HR | 311 |

Suppose you want to query the Computers table. You want to display the SerialNum, Brand, and Department columns for computers located in office numbers less than 130 and with a brand name of either Dell or Gateway.

```
SELECT SerialNum, Brand, Department FROM Computers
WHERE (Brand = 'Dell' OR Brand = 'Gateway') AND OfficeNumber < 130;
```



| SerialNum | Brand | Department |
|---|---|---|
| G9277288282 | Dell | HR |
| W2121040244 | Gateway | CustomerService |

Results (output)

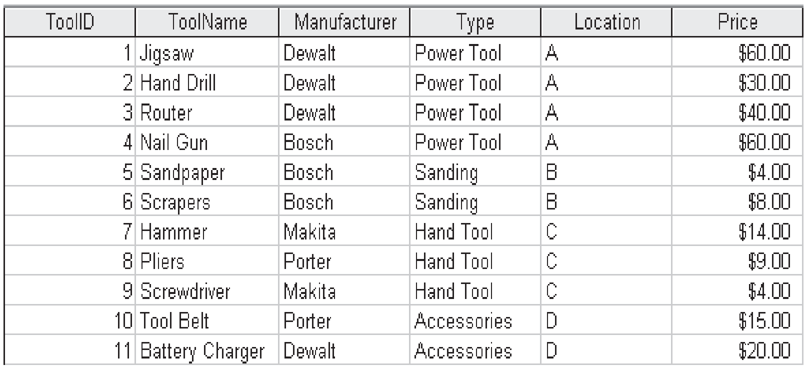
## The LIKE Operator

The *LIKE* operator uses wildcard characters to match patterns in data. These are special characters used to match parts of a value.

Wildcard characters used with the LIKE operator

| **Character** | **Description** |
|---|---|
| ? | Any single character. |
| * | Zero or more characters. |
| # | Any single digit (0-9). |
| [characters] | Any single character in a group of one or more characters. |
| [!characters] | Any single character not in a group of one or more characters. |

## Example



| ToolID | ToolName | Manufacturer | Type | Location | Price |
|---|---|---|---|---|---|
| 1 | Jigsaw | Dewalt | Power Tool | A | $60.00 |
| 2 | Hand Drill | Dewalt | Power Tool | A | $30.00 |
| 3 | Router | Dewalt | Power Tool | A | $40.00 |
| 4 | Nail Gun | Bosch | Power Tool | A | $60.00 |
| 5 | Sandpaper | Bosch | Sanding | B | $4.00 |
| 6 | Scrapers | Bosch | Sanding | B | $8.00 |
| 7 | Hammer | Makita | Hand Tool | C | $14.00 |
| 8 | Pliers | Porter | Hand Tool | C | $9.00 |
| 9 | Screwdriver | Makita | Hand Tool | C | $4.00 |
| 10 | Tool Belt | Porter | Accessories | D | $15.00 |
| 11 | Battery Charger | Dewalt | Accessories | D | $20.00 |

Suppose you want to query the Tools table in above to retrieve tools made by manufacturers that begin with the letter D and are located in warehouse sections A through C.

```
SELECT *
FROM Tools
WHERE Manufacturer LIKE 'D*' AND Location LIKE '[A-C]';
```

The preceding script uses the asterisk (*) wildcard character and the brackets ([ ]) with the LIKE operator in the WHERE clause. The asterisk (*) is used to specify the letter D. The letter D is placed in front of the asterisk to instruct the DBMS to retrieve manufacturers that begin with the letter D. The brackets ([ ]) are used to instruct the database to retrieve locations from A to C.

**More examples show implementations of other LIKE operator search patterns**.

To retrieve manufacturers that end with the letter H, type the following:

```
SELECT *
FROM Tools
WHERE Manufacturer LIKE '*H';
```

To retrieve any occurrence of the word Dewalt, type the following:

```
SELECT *
FROM Tools
WHERE Manufacturer LIKE '*Dewalt*';
```

To retrieve data that matches a single character in the Manufacturer column, type the following:

```
SELECT *
FROM Tools
WHERE Manufacturer LIKE 'Bos?h';
```

The question mark (?) is used as a character placeholder.

To retrieve data that matches a single digit in the ToolID column, type the following:

```
SELECT *
FROM Tools
WHERE ToolID LIKE '1#';
```

The pound sign (#) is used as a digit placeholder.

To retrieve warehouse locations that are not A to C, type the following:

```
SELECT *
FROM Tools
WHERE Location LIKE '[!A-C]';
```

The ! symbol means NOT.

## The BETWEEN Operator

The BETWEEN operator is used to determine whether a value of an expression falls within a specified range of values.

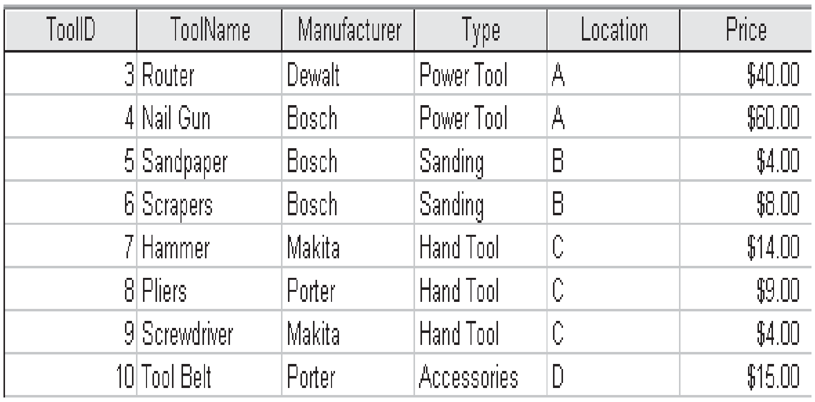
### Example

Suppose you want to query the Tools table to retrieve tool IDs equal to or between 3 and 10. Look at the following script:

```
SELECT *
FROM Tools
WHERE ToolID BETWEEN 3 AND 10;
```

This script uses the BETWEEN operator in the WHERE clause to retrieve tool IDs equal to or between 3 and 10. The AND operator is used to specify values 3 and 10. Note that the BETWEEN operator always includes the expressions specified on either side of the AND operator.

**Results/Output**



| ToolID | ToolName | Manufacturer | Type | Location | Price |
|---|---|---|---|---|---|
| 3 | Router | Dewalt | Power Tool | A | $40.00 |
| 4 | Nail Gun | Bosch | Power Tool | A | $60.00 |
| 5 | Sandpaper | Bosch | Sanding | B | $4.00 |
| 6 | Scrapers | Bosch | Sanding | B | $8.00 |
| 7 | Hammer | Makita | Hand Tool | C | $14.00 |
| 8 | Pliers | Porter | Hand Tool | C | $9.00 |
| 9 | Screwdriver | Makita | Hand Tool | C | $4.00 |
| 10 | Tool Belt | Porter | Accessories | D | $15.00 |

This query is equivalent to the preceding query:

```
SELECT *
FROM Tools
WHERE ToolID >= 3 AND ToolID <=10;
```

## The IN and NOT Operators

The *IN* operator is used to match conditions in a list of expressions.
The *NOT* operator is used to match any condition opposite of the one defined. Take a look at Example 11.

**Example 11**
Say you want to query the Tools table to retrieve information on every tool except the ones manufactured by Bosch, Porter, or Makita. Look at the following script:

```
SELECT *
FROM Tools
WHERE Manufacturer NOT IN ('Bosch', 'Porter', 'Makita');
```

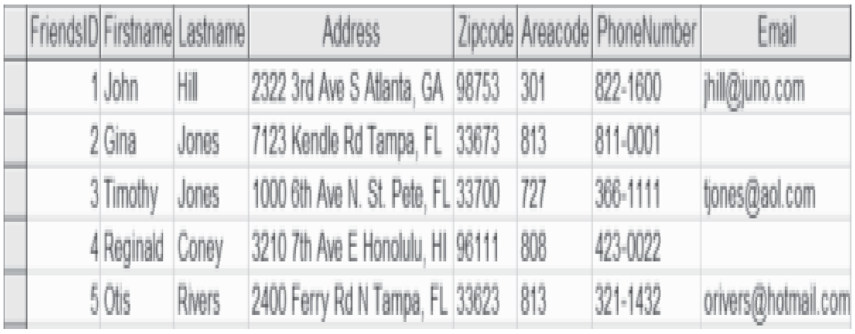
The preceding script uses the IN operator to specify three values (Bosch, Porter, and Makita). The values are enclosed in parentheses and each individual value is enclosed in quotes.

```
SELECT *
FROM Tools
WHERE Manufacturer IN ('Bosch', 'Porter', 'Makita');
```

## The IS NULL and IS NOT NULL Operators

The *IS NULL* operator is used to determine if a field contains data. The *IS NOT NULL* operator is used to determine if a field does not contain data.

Example
Suppose you want to retrieve individuals who do not have an e-mail address but do have a phone number listed in the Friends table



| FriendsID | Firstname | Lastname | Address | Zipcode | Areacode | PhoneNumber | Email |
|---|---|---|---|---|---|---|---|
| 1 | John | Hill | 2322 3rd Ave S Atlanta, GA | 98753 | 301 | 822-1600 | jhill@juno.com |
| 2 | Gina | Jones | 7123 Kendle Rd Tampa, FL | 33673 | 813 | 811-0001 | |
| 3 | Timothy | Jones | 1000 6th Ave N. St. Pete, FL | 33700 | 727 | 366-1111 | tjones@aol.com |
| 4 | Reginald | Coney | 3210 7th Ave E Honolulu, HI | 96111 | 808 | 423-0022 | |
| 5 | Otis | Rivers | 2400 Ferry Rd N Tampa, FL | 33623 | 813 | 321-1432 | orivers@hotmail.com |

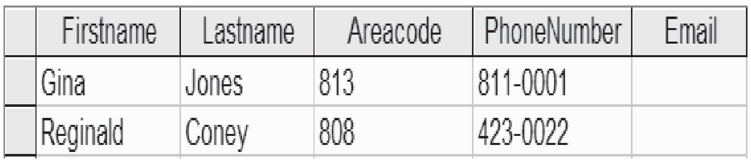
Friends table

## The IS NULL and IS NOT NULL Operators

Look at the following script:

```
SELECT Firstname, Lastname, Areacode, PhoneNumber, Email FROM Friends
WHERE Email IS NULL AND PhoneNumber IS NOT NULL;
```

The preceding script implements the IS NULL and IS NOT NULL keywords in the WHERE clause. The IS NULL keywords are used to locate NULL values in the Email column. The IS NOT NULL keywords are used to locate values in the PhoneNumber column.



| Firstname | Lastname | Areacode | PhoneNumber | Email |
|---|---|---|---|---|
| Gina | Jones | 813 | 811-0001 | |
| Reginald | Coney | 808 | 423-0022 | |

Results (output)

## Calculated Fields

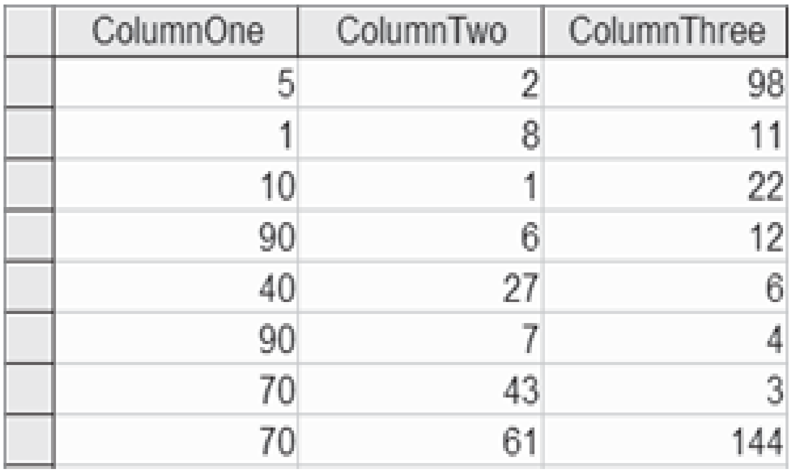
- So far your have learned about an array of operators that can be used in the WHERE clause.
- The ability to perform mathematical calculations enables you to collect information beyond the data actually stored in the database.
- Another set of operators that you should become familiar with is the arithmetic and aggregate operators .
- The *arithmetic operators* are used to perform mathematical calculations and can be used throughout a query.

- The following table shows the arithmetic operators used in Microsoft Access's SQL view.

## Arithmetic Operators

| Operator | Description |
|---|---|
| Negation (–) | Used to take the negative of a number. |
| Exponentiation (^) | Used to perform exponentiation. |
| Divide (/) | Used to perform division. |
| Multiply (*) | Used to perform multiplication. |
| Modulus (%) | Used to return the remainder in division. |
| Plus (+) | Used to perform addition. |
| Minus (–) | Used to perform subtraction. |

**Take a look at Example 1, which implements two of the arithmetic operators.**

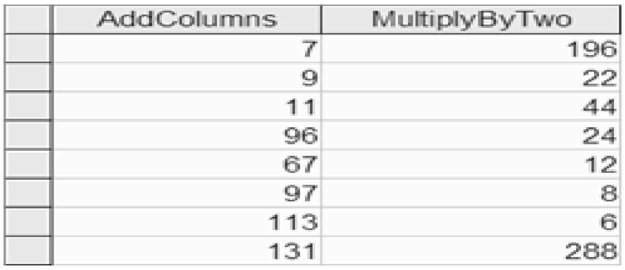


| ColumnOne | ColumnTwo | ColumnThree |
|---:|---:|---:|
| 5 | 2 | 98 |
| 1 | 8 | 11 |
| 10 | 1 | 22 |
| 90 | 6 | 12 |
| 40 | 27 | 6 |
| 90 | 7 | 4 |
| 70 | 43 | 3 |
| 70 | 61 | 144 |

**Numbers table**

- Suppose you want to add two values that are stored in two separate columns and multiply a value that is stored in a column by a specified value. Use the following script:

```
SELECT (ColumnOne + ColumnTwo) AS AddColumns, (ColumnThree * 2)
AS MultiplyByTwo
FROM Numbers;
```



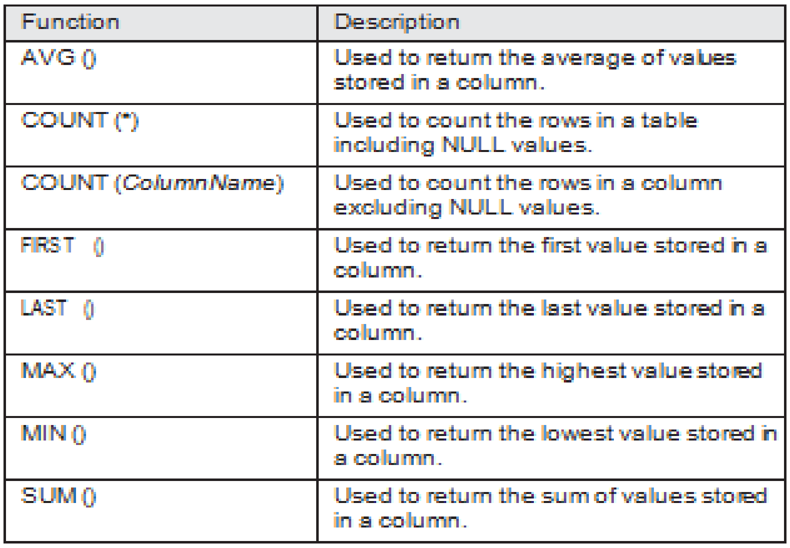
| AddColumns | MultiplyByTwo |
|---:|---:|
| 7 | 196 |
| 9 | 22 |
| 11 | 44 |
| 96 | 24 |
| 67 | 12 |
| 97 | 8 |
| 113 | 6 |
| 131 | 288 |

Results (output)

## Aggregate Functions

- *Aggregate functions* can also be used to perform mathematical calculations. They operate on several rows at one time and are used to return a single value based on values stored in a column.
- Unlike arithmetic operators, they cannot be used to calculate values stored in multiple columns.
- Instead of retrieving actual information stored in the database, you can use aggregate functions to summarize data that is stored in the database. For example, aggregate functions can be used to average and sum values stored in a column.

## Aggregate Functions



| Function | Description |
|---|---|
| AVG () | Used to return the average of values stored in a column. |
| COUNT (*) | Used to count the rows in a table including NULL values. |
| COUNT (*ColumnName*) | Used to count the rows in a column excluding NULL values. |
| FIRST () | Used to return the first value stored in a column. |
| LAST () | Used to return the last value stored in a column. |
| MAX () | Used to return the highest value stored in a column. |
| MIN () | Used to return the lowest value stored in a column. |
| SUM () | Used to return the sum of values stored in a column. |

## Using the AVG (), FIRST (), LAST (), SUM (), MAX (), and MIN () Functions

- Suppose you want to use the Numbers table to average the values stored in the ColumnOne column, find the first and last values stored in the ColumnOne column, sum the values stored in the ColumnTwo column, and find the highest and lowest values stored in the ColumnTwo column. Take a look at the following script:

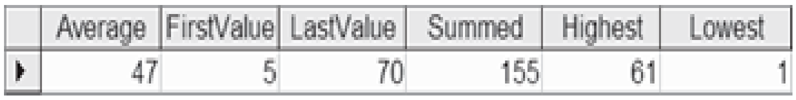
```
SELECT AVG (ColumnOne) AS Average, FIRST (ColumnOne) AS
FirstValue, LAST (ColumnOne) AS LastValue, SUM (ColumnTwo) AS
Summed, MAX (ColumnTwo) AS Highest, MIN (ColumnTwo) AS
Lowest FROM Numbers;
```

## Using the COUNT () Function

- The COUNT () function can be used in two ways.
  - Count the number of rows in a table or to count the rows in a specified column.
- To count the number of rows in a table use the asterisk (*) within the function (COUNT (*)).
- To count the number of rows in a specified column, specify the name of a column in the function (COUNT (*ColumnName*)).

- **Note:** When you use COUNT (*ColumnName*), NULL values are excluded in the count. When you use COUNT (*), NULL values are included in the count. Example 3 shows an example using both the COUNT (*) and COUNT (*ColumnName*) functions.

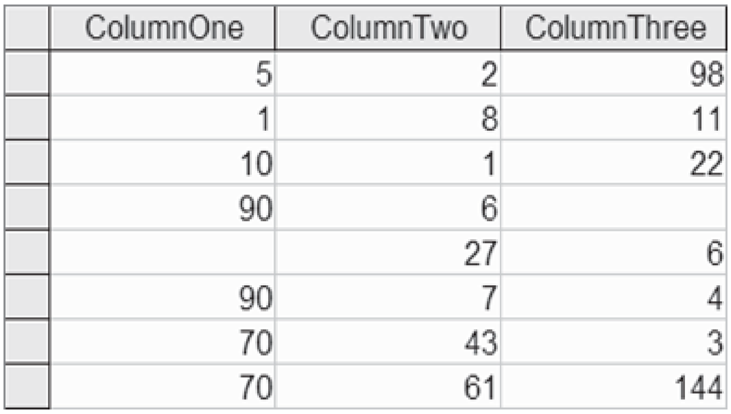
## Results(output)



| | Average | FirstValue | LastValue | Summed | Highest | Lowest |
|---|---|---|---|---|---|---|
| ▶ | 47 | 5 | 70 | 155 | 61 | 1 |

Results (output)

## The Numbers2 table contains two NULL values.



| | ColumnOne | ColumnTwo | ColumnThree |
|---|---|---|---|
| | 5 | 2 | 98 |
| | 1 | 8 | 11 |
| | 10 | 1 | 22 |
| | 90 | 6 | |
| | | 27 | 6 |
| | 90 | 7 | 4 |
| | 70 | 43 | 3 |
| | 70 | 61 | 144 |

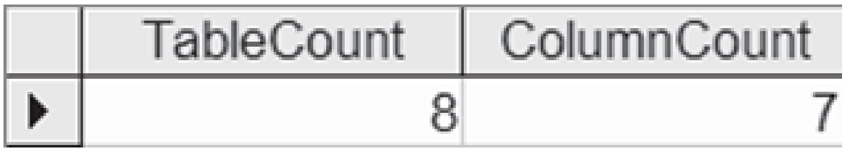
Numbers2 table

- Suppose you want to use the Numbers2 table to count the rows in the table and the rows in the ColumnThree column. Us the following script:

  **SELECT COUNT (*) AS TableCount, COUNT (ColumnThree) AS ColumnCount FROM Numbers;**

- The preceding script uses the COUNT (*) function to count the total number of rows in the Numbers table, including NULL values. The COUNT (ColumnThree) function is used to count the total number of rows in the ColumnThree column, excluding NULL values.

## Results/output



| | TableCount | ColumnCount |
|---|---|---|
| ▶ | 8 | 7 |