SQL

09 June 2024 19:16

- SQL is a standard language for storing, manipulating and retrieving data in databases.

What Can SQL do?

- SQL can create new databases
- SQL can create new tables in a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can execute queries against a database
- SQL can retrieve data from a database

RDBMS

- RDBMS stands for Relational Database Management System.
- RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- The data in RDBMS is stored in database objects called tables.
- Every table is broken up into smaller entities called fields and records, where fields are the column names and records is the data that is filled in the table.

Some of The Most Important SQL Commands

- `SELECT` - extracts data from a database
- `UPDATE` - updates data in a database
- `DELETE` - deletes data from a database
- `INSERT INTO` - inserts new data into a database
- `CREATE DATABASE` - creates a new database
- `ALTER DATABASE` - modifies a database
- `CREATE TABLE` - creates a new table
- `ALTER TABLE` - modifies a table
- `DROP TABLE` - deletes a table
- `CREATE INDEX` - creates an index (search key)
- `DROP INDEX` - deletes an index

SQL SELECT Statement

```
SELECT column1, column2, ...
FROM table_name;
```

Ex : `SELECT CustomerName, City FROM Customers;`

SQL SELECT DISTINCT Statement

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

Ex : `SELECT COUNT(DISTINCT Country) FROM Customers;`

SQL WHERE Clause

- The WHERE clause is used to filter records.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Ex :
```
SELECT * FROM Customers
 WHERE CustomerID=1;
```

Different Operators

=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

SQL ORDER BY Keyword

- The ORDER BY keyword is used to sort the result-set in ascending or descending order.

```
SELECT column1, column2, ...
FROM table_name
```

```
ORDER BY column1, column2, ... ASC|DESC;
```

```
SELECT * FROM Products
ORDER BY Price DESC;
```

ORDER BY Several Columns

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column. This means that it orders by Country, but if some rows have the same Country, it orders them by CustomerName:

```
SELECT * FROM Customers
ORDER BY Country, CustomerName;
```

Using Both ASC and DESC

The following SQL statement selects all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "CustomerName" column:

```
SELECT * FROM Customers
ORDER BY Country ASC, CustomerName DESC;
```

SQL AND Operator

- The AND operator displays a record if *all* the conditions are TRUE.
- The OR operator displays a record if *any* of the conditions are TRUE.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
```

```
SELECT * FROM Customers
WHERE Country = 'Germany'
AND City = 'Berlin'
AND PostalCode > 12000;
```

Combining AND and OR

```
SELECT * FROM Customers
WHERE Country = 'Spain'  AND (CustomerName LIKE 'G%' OR CustomerName LIKE 'R%');
```

SQL OR Operator

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

```
SELECT * FROM Customers
WHERE City = 'Berlin'  OR CustomerName LIKE 'G%'  OR Country = 'Norway';
```

SQL NOT Operator

- The NOT operator is used in combination with other operators to give the opposite result, also called the negative result.

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

```
SELECT * FROM Customers
WHERE NOT Country = 'Spain';
```

```
SELECT * FROM Customers
WHERE CustomerName NOT LIKE 'A%';
```

```
SELECT * FROM Customers
WHERE CustomerID NOT BETWEEN 10 AND 60;
```

```
SELECT * FROM Customers
WHERE City NOT IN ('Paris', 'London');
```

```
SELECT * FROM Customers
WHERE NOT CustomerID > 50;
```

```
SELECT * FROM Customers
WHERE NOT CustomerId < 50;
```

- **Note:** There is a not-less-than operator: `!<` that would give you the same result.

SQL INSERT INTO Statement

- The `INSERT INTO` statement is used to insert new records in a table.

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');
```

Insert Data Only in Specified Columns

```
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
```

Insert Multiple Rows

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES
('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway'),
('Greasy Burger', 'Per Olsen', 'Gateveien 15', 'Sandnes', '4306', 'Norway'),
('Tasty Tee', 'Finn Egan', 'Streetroad 19B', 'Liverpool', 'L1 0AA', 'UK');
```

SQL NULL Values

- The `IS NULL` operator is used to test for empty values (NULL values).

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NULL;
```

- The `IS NOT NULL` operator is used to test for non-empty values (NOT NULL values).

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NOT NULL;
```

SQL UPDATE Statement

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

- Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

SQL DELETE Statement

```
DELETE FROM table_name WHERE condition;
```

```
DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';
```

Delete All Records

```
DELETE FROM table_name;
```

```
DELETE FROM Customers;
```

Delete a Table

```
DROP TABLE Customers;
```

SQL LIMIT

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

```
SELECT * FROM Customers
LIMIT 3;
```

FETCH FIRST

```
SELECT * FROM Customers
FETCH FIRST 3 ROWS ONLY;
```

SQL TOP PERCENT

```
SELECT TOP 50 PERCENT * FROM Customers;
```

SQL Aggregate Functions

- Aggregate functions are often used with the GROUP BY clause of the SELECT statement.

The most commonly used SQL aggregate functions are:

- MIN() - returns the smallest value within the selected column
- MAX() - returns the largest value within the selected column
- COUNT() - returns the number of rows in a set
- SUM() - returns the total sum of a numerical column
- AVG() - returns the average value of a numerical column

SQL MIN() and MAX() Functions

```
SELECT MIN/MAX(column_name)
FROM table_name
WHERE condition;
```

Use MIN() with GROUP BY

```
SELECT MIN(Price) AS SmallestPrice, CategoryID
FROM Products
GROUP BY CategoryID;
```

SQL COUNT() Function

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

```
SELECT COUNT(ProductID)
FROM Products
WHERE Price > 20;
```

Ignore Duplicates

```
SELECT COUNT(DISTINCT Price)
FROM Products;
```

Use COUNT() with GROUP BY

```
SELECT COUNT(*) AS [Number of records], CategoryID
FROM Products
GROUP BY CategoryID;
```

SQL SUM() Function

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

```
SELECT SUM(Quantity)
FROM OrderDetails
WHERE ProductId = 11;
```

Use SUM() with GROUP BY

```
SELECT OrderID, SUM(Quantity) AS [Total Quantity]
FROM OrderDetails
GROUP BY OrderID;
```

SUM() With an Expression

```
SELECT SUM(Quantity * 10)
FROM OrderDetails;
```

SQL AVG() Function

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

```
SELECT AVG(Price)
FROM Products
WHERE CategoryID = 1;
```

Use AVG() with GROUP BY

```
SELECT AVG(Price) AS AveragePrice, CategoryID
FROM Products
GROUP BY CategoryID;
```

Higher Than Average

```
SELECT * FROM Products
WHERE price > (SELECT AVG(price) FROM Products);
```

SQL LIKE Operator

- The LIKE operator is used in a WHERE clause to search for a specified pattern in a column

There are two wildcards often used in conjunction with the LIKE operator:
- The percent sign % represents zero, one, or multiple characters
- The underscore sign _ represents one, single character

```
SELECT * FROM Customers
WHERE CustomerName LIKE '_a%';
```

SQL IN Operator

- The IN operator allows you to specify multiple values in a WHERE clause.

```
SELECT * FROM Customers
WHERE Country IN ('Germany', 'France', 'UK');
```

NOT IN

```
SELECT * FROM Customers
WHERE Country NOT IN ('Germany', 'France', 'UK');
```

SQL Aliases

- SQL aliases are used to give a table, or a column in a table, a temporary name.

```
SELECT CustomerID AS ID
FROM Customers;
```

AS is Optional

```
SELECT CustomerID ID
FROM Customers;
```

Alias for Columns

```
SELECT CustomerID AS ID, CustomerName AS Customer
FROM Customers;
```

Using Aliases With a Space Character

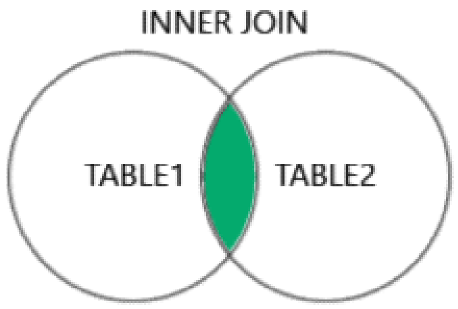
```
SELECT ProductName AS [My Great Products]
FROM Products;
```

SQL Joins

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

Here are the different types of the JOINs in SQL:

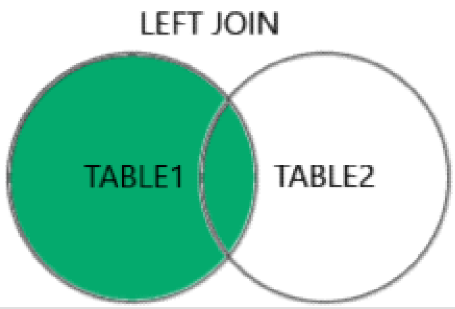
- `(INNER) JOIN`: Returns records that have matching values in both tables
- `LEFT (OUTER) JOIN`: Returns all records from the left table, and the matched records from the right table
- `RIGHT (OUTER) JOIN`: Returns all records from the right table, and the matched records from the left table
- `FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table



```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
```

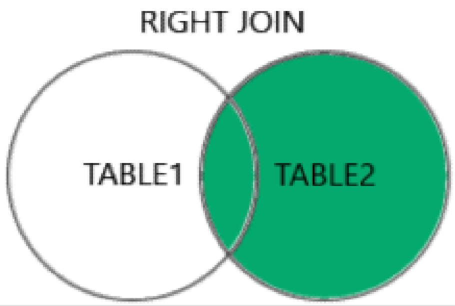
```
INNER JOIN table2
ON  table1.matching_column = table2.matching_column;
```

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```



```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```

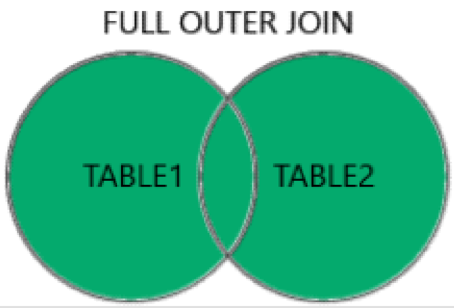
```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```



```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
```

```
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```



```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;
```

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

SQL UNION Operator

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

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

UNION ALL Syntax

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

```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```

```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION ALL
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

SQL GROUP BY Statement

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

GROUP BY With JOIN Example

```
SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID
GROUP BY ShipperName;
```

SQL HAVING Clause

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

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

SQL EXISTS Operator

- The EXISTS operator is used to test for the existence of any record in a subquery.
- The EXISTS operator returns TRUE if the subquery returns one or more records.

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID =
Suppliers.supplierID AND Price < 20);
```

SQL SELECT INTO Statement

- The SELECT INTO statement copies data from one table into a new table.

```
SELECT column1, column2, column3, ...
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

```
SELECT * INTO CustomersBackup2017 IN 'Backup.mdb'
FROM Customers;
```

SQL CASE Expression

- The CASE expression goes through conditions and returns a value when the first condition is met (like an if-then-else statement).

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  WHEN conditionN THEN resultN
  ELSE result
END;
```

```
SELECT OrderID, Quantity,
CASE
  WHEN Quantity > 30 THEN 'The quantity is greater than 30'
  WHEN Quantity = 30 THEN 'The quantity is 30'
  ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

SQL Stored Procedures

- A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.
- So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

Execute a Stored Procedure

```
EXEC procedure_name;
```

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

```
EXEC SelectAllCustomers;
```

Stored Procedure With Multiple Parameters

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
GO;
```

```
EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';
```

SQL Comments

- Comments are used to explain sections of SQL statements, or to prevent execution of SQL statements.

Single Line Comments

Single line comments start with `--`.

```
-- Select all:
SELECT * FROM Customers;
```

Multi-line Comments

```
/*Select all the columns
of all the records
in the Customers table:*/
SELECT * FROM Customers;
```

SQL CREATE DATABASE Statement

```
CREATE DATABASE databasename;
```

```
CREATE DATABASE testDB;
```

SQL DROP DATABASE Statement

```
DROP DATABASE databasename;
```

```
DROP DATABASE testDB;
```

SQL BACKUP DATABASE

- The BACKUP DATABASE statement is used in SQL Server to create a full back up of an existing SQL database.

```
BACKUP DATABASE databasename
TO DISK = 'filepath';
```

The SQL BACKUP WITH DIFFERENTIAL Statement

- A differential back up only backs up the parts of the database that have changed since the last full database backup.

```
BACKUP DATABASE databasename
TO DISK = 'filepath'
WITH DIFFERENTIAL;
```

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak'
WITH DIFFERENTIAL;
```

SQL CREATE TABLE Statement

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

SQL DROP TABLE

```
DROP TABLE table_name;
```

```
DROP TABLE Shippers;
```

SQL TRUNCATE TABLE

```
TRUNCATE TABLE table_name;
```

SQL ALTER TABLE Statement

- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

ALTER TABLE - ADD Column

```
ALTER TABLE table_name
ADD column_name datatype;
```

```
ALTER TABLE Customers
ADD Email varchar(255);
```

ALTER TABLE - DROP COLUMN

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

```
ALTER TABLE Customers
DROP COLUMN Email;
```

ALTER TABLE - RENAME COLUMN

```
ALTER TABLE table_name
RENAME COLUMN old_name to new_name;
```

ALTER TABLE - ALTER/MODIFY DATATYPE

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

SQL Constraints

- SQL constraints are used to specify rules for data in a table.

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
  ....
);
```

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Prevents actions that would destroy links between tables
- CHECK - Ensures that the values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column if no value is specified
- CREATE INDEX - Used to create and retrieve data from the database very quickly

SQL NOT NULL Constraint

- The NOT NULL constraint enforces a column to NOT accept NULL values.

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

ALTER TABLE Persons
MODIFY COLUMN Age int NOT NULL;
```

SQL UNIQUE Constraint

- The UNIQUE constraint ensures that all values in a column are different.

- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.
- However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

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

SQL PRIMARY KEY Constraint

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

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

SQL FOREIGN KEY Constraint

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

- The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

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

SQL CHECK Constraint

- The CHECK constraint is used to limit the value range that can be placed in a column.

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int CHECK (Age>=18)
);
```

SQL DEFAULT Constraint

- The DEFAULT constraint is used to set a default value for a column.

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
```

SQL CREATE INDEX Statement

- The CREATE INDEX statement is used to create indexes in tables.
- Indexes are used to retrieve data from the database more quickly than otherwise. 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 index_name
ON table_name (column1, column2, ...);
```

CREATE UNIQUE INDEX Syntax

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

```
CREATE INDEX idx_pname
ON Persons (LastName, FirstName);
```

SQL AUTO INCREMENT Field

- Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

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

SQL Working With Dates

- DATE - format YYYY-MM-DD
- DATETIME - format: YYYY-MM-DD HH:MI:SS
- TIMESTAMP - format: YYYY-MM-DD HH:MI:SS
- YEAR - format YYYY or YY

SQL Views

- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

```
SELECT * FROM [Brazil Customers];
```

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```