**Structure Query Language (SQL)**

SQL was developed in 1970’s in an IBM laboratory “San Jose Research Laboratory” (now the Amaden Research center). SQL is derived from the SEQUEL one of the database language popular during 1970’s. SQL established itself as the standard relational database language. Two standard organization (ANSI) and International standards organization (ISO) currently promote SQL standards to industry. In 1986 ANSI & ISO published an SQL standard called SQL-86. In 1987, IBM published its own corporate SQL standard, the system application Architecture Database Interface (SAA-SQL). In 1989, ANSI published extended standard for SQL called, SQL-89. The next version was SQL-92, and the recent version is SQL: 1999.

Basic Term and Terminology

**Query**: is a statement requesting the retrieval of information.

**Query language:** language through which user request information from database.

These languages are generally higher level language than programming language.

The two types of query language are:

**(i) Procedural language**

• User instructs the system to perform sequence of operation on the

database to compete the desired result. Example : relational algebra

ii) Non- procedural language

• User describes the desired information without giving a specific procedure

for obtaining that desired information.

• Examples: tuple relational calculus and domain relational calculus.

Database Languages

Two types of database language

1. Data Definition Language (DDL)

2. Data Manipulation Language (DML)

**Data Definition Language**

• Specifies the database schema.

• For e.g.: The following statement in SQL defines relation named `student’.

**CREATE TABLE student**

**(**

**student\_id VARCHAR2(3),**

**address VARCHAR(30)**

**);**

The execution of this DDL statement creates the `student’ table. It also updates a special set of tables called *data dictionary* or *data directory*. A data dictionary contains metadata, that is data about data. The schema of table is an example of metadata. A database system consults data dictionary or data directory. Through the set of special type of DDL, called data storage definition language, we may specify the storage structure (like size of database, size of table etc) and access

methods.

The DDL allow to enforce constrains in the database. For example: student\_id

should begin with `S’, address could not be null etc.

CREATE TABLE STUDENT

(

student\_id VARCHAR2 (3),

address VARCHAR2 NOT NULL,

CONSTRAINT ch\_student\_id CHECK (student\_id LIKE `S%’)

);

**Data Manipulation Language:**

• A data manipulation language is a language that enables users to access or

manipulate the data in database. The data manipulation means :

o Retrieval of information stored in database.

o The insertion of new information into database.

o Deletion of information from database.

o Modification of data in database.

Basic SQL Commands

SQL General Data Types

Each column in a database table is required to have a name and a data type. SQL developers have to decide what types of data will be stored inside each and every table column when creating a SQL table. The data type is a label and a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.

The following table lists the general data types in SQL:

###### Integers

[bigint](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ia-iz_3ss4.htm)

Integer (whole number) data from -2^63 (-9223372036854775808) through 2^63-1 (9223372036854775807).

[int](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ia-iz_3ss4.htm)

Integer (whole number) data from -2^31 (-2,147,483,648) through 2^31 - 1 (2,147,483,647).

[smallint](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ia-iz_3ss4.htm)

Integer data from 2^15 (-32,768) through 2^15 - 1 (32,767).

[tinyint](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ia-iz_3ss4.htm)

Integer data from 0 through 255.

###### bit

[bit](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ba-bz_2it0.htm)

Integer data with either a 1 or 0 value.

###### decimal and numeric

[decimal](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_de-dz_3grn.htm)

Fixed precision and scale numeric data from -10^38 +1 through 10^38 –1.

[numeric](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_de-dz_3grn.htm)

Functionally equivalent to **decimal.**

###### money and smallmoney

[money](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ma-mz_49q1.htm)

Monetary data values from -2^63 (-922,337,203,685,477.5808) through 2^63 - 1 (+922,337,203,685,477.5807), with accuracy to a ten-thousandth of a monetary unit.

[smallmoney](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ma-mz_49q1.htm)

Monetary data values from -214,748.3648 through +214,748.3647, with accuracy to a ten-thousandth of a monetary unit.

##### Approximate Numerics

[float](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_fa-fz_6r3g.htm)

Floating precision number data from -1.79E + 308 through 1.79E + 308.

[real](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_fa-fz_6r3g.htm)

Floating precision number data from -3.40E + 38 through 3.40E + 38.

##### datetime and smalldatetime

[datetime](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_da-db_9xut.htm)

Date and time data from January 1, 1753, through December 31, 9999, with an accuracy of three-hundredths of a second, or 3.33 milliseconds.

[smalldatetime](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_da-db_9xut.htm)

Date and time data from January 1, 1900, through June 6, 2079, with an accuracy of one minute.

##### Character Strings

[char](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ca-co_7tpu.htm)

Fixed-length non-Unicode character data with a maximum length of 8,000 characters.

[varchar](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ca-co_7tpu.htm)

Variable-length non-Unicode data with a maximum of 8,000 characters.

[text](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ta-tz_2838.htm)

Variable-length non-Unicode data with a maximum length of 2^31 - 1 (2,147,483,647) characters.

##### Unicode Character Strings

[nchar](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_na-nop_9msy.htm)

Fixed-length Unicode data with a maximum length of 4,000 characters.

[nvarchar](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_na-nop_9msy.htm)

Variable-length Unicode data with a maximum length of 4,000 characters. **sysname** is a system-supplied user-defined data type that is functionally equivalent to **nvarchar(128)** and is used to reference database object names.

[ntext](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_nos-nz_0lyd.htm)

Variable-length Unicode data with a maximum length of 2^30 - 1 (1,073,741,823) characters.

##### Binary Strings

[binary](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ba-bz_75yx.htm)

Fixed-length binary data with a maximum length of 8,000 bytes.

[varbinary](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ba-bz_75yx.htm)

Variable-length binary data with a maximum length of 8,000 bytes.

[image](mk:@MSITStore:C:\Program%20Files\Microsoft%20SQL%20Server\80\Tools\Books\tsqlref.chm::/ts_ia-iz_9rfp.htm)

Variable-length binary data with a maximum length of 2^31 - 1 (2,147,483,647) bytes.

**The SQL CREATE DATABASE Statement**

The CREATE DATABASE statement is used to create a database.

SQL CREATE DATABASE Syntax

*CREATE DATABASE dbname;*

SQL CREATE DATABASE Example

The following SQL statement creates a database called "student\_db":

CREATE DATABASE student\_db;

Database tables can be added with the CREATE TABLE statement.

**Drop Database :- The** *drop* command is used to remove the database from the database server and once removed can not be recorved.

Syntax:

**Drop database databaseName;**

**For example:**

**Drop database** student\_db;

* **The SQL CREATE TABLE Statement**

The CREATE TABLE statement is used to create a table in a database.

Tables are organized into rows and columns; and each table must have a name.

SQL CREATE TABLE Syntax

CREATE TABLE *table\_name*  
(  
*column\_name1 data\_type*(*size*),  
*column\_name2 data\_type*(*size*),  
*column\_name3 data\_type*(*size*),  
....  
)

The column\_name parameters specify the names of the columns of the table.

The data\_type parameter specifies what type of data the column can hold (e.g. varchar, integer, decimal, date, etc.). The size parameter specifies the maximum length of the column of the table.

SQL CREATE TABLE Example

Now we want to create a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City. We use the following CREATE TABLE statement:

CREATE TABLE Persons  
(  
PersonID int,  
LastName varchar(25),  
FirstName varchar(25),  
Address varchar(25),  
City varchar(25)  
)

* **The SQL INSERT INTO Statement**

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

SQL INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two forms.

The first form does not specify the column names where the data will be inserted, only their values:

INSERT INTO *table\_name*  
VALUES (*value1*,*value2*,*value3*,...);

The second form specifies both the column names and the values to be inserted:

INSERT INTO *table\_name* (*column1*,*column2*,*column3*,...)  
VALUES (*value1*,*value2*,*value3*,...);

**For example:**

**Insert into person**

**(**PersonID ,LastName ,FirstName ,Address ,City )

Values(1001,’singh’,’sunil’,’BH-10’,’chitwan’)

The above command to add only specified columns values may not all columns

Or

**Insert into person**

Values(1001,’singh’,’sunil’,’BH-10’,’chitwan’)

To add values to all columns and follow the table attributes order

To repeat above statements many times to add more records.

To View The Records

1. **Select \* from person :- To list all records**
2. **Select personid, firstname, city from person :-** To view only personid, firstname and city of all person.

## SQL Constraints

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

 If there is any violation between the constraint and the data action, the action is aborted by the constraint. Constraints can be specified when the table is created (inside the CREATE TABLE statement) or after the table is created (inside the ALTER TABLE statement).

### SQL CREATE TABLE + CONSTRAINT Syntax

CREATE TABLE table\_name  
(  
column\_name1 data\_type(size) constraint\_name,  
column\_name2 data\_type(size) constraint\_name,  
column\_name3 data\_type(size) constraint\_name,  
....  
);

 In SQL, we have the following constraints:

* **NOT NULL** - Indicates that a column cannot store NULL value
* **UNIQUE** - Ensures that each rows for a column must have a unique value
* **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have an unique identity which helps to find a particular record in a table more easily and quickly
* **FOREIGN KEY** - Ensure the referential integrity of the data in one table to match values in another table
* **CHECK** - Ensures that the value in a column meets a specific condition
* **DEFAULT** - Specifies a default value when specified none for this column

## SQL NOT NULL Constraint

The NOT NULL constraint enforces a column to NOT accept NULL values. The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field. The following SQL enforces the "P\_Id" column and the "LastName" column to not accept NULL values:

**CREATE TABLE Persons  
(  
P\_Id int NOT NULL,  
LastName varchar(25) NOT NULL,  
FirstName varchar(25),  
Address varchar(25),  
City varchar(25)  
)**

## SQL UNIQUE Constraint

The UNIQUE constraint uniquely identifies each record in a database table. The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.

Note that you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

## SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "P\_Id" column when the "Persons" table is created:

**MSSQL:**

CREATE TABLE Persons  
(  
P\_Id int NOT NULL,  
LastName varchar(25) NOT NULL,  
FirstName varchar(25),  
Address varchar(25),  
City varchar(25),  
UNIQUE (P\_Id)  
)

**Or**

CREATE TABLE Persons  
(  
P\_Idint NOT NULL UNIQUE,  
LastName varchar(25) NOT NULL,  
FirstName varchar(25),  
Address varchar(25),  
City varchar(25)  
)

**Or**

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

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

CREATE TABLE Persons  
(  
P\_Id int NOT NULL,  
LastName varchar(25) NOT NULL,  
FirstName varchar(25),  
Address varchar(25),  
City varchar(25),  
CONSTRAINT uc\_PersonID UNIQUE (P\_Id,LastName)  
)

## SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "P\_Id" column when the table is already created, use the following SQL:

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

ALTER TABLE Persons  
ADD UNIQUE (P\_Id)

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

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

ALTER TABLE Persons  
ADD CONSTRAINT uc\_PersonID UNIQUE (P\_Id,LastName)

## To DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

**MSSQL:**

ALTER TABLE Persons

DROP uc\_PersonID

Or

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

ALTER TABLE Persons  
DROP CONSTRAINT uc\_PersonID

## SQL PRIMARY KEY Constraint

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

Primary keys must contain unique values. A primary key column cannot contain NULL values.

Each table should have a primary key, and each table can have only ONE primary key.

## SQL PRIMARY KEY Constraint on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "P\_Id" column when the "Persons" table is

**For example**

CREATE TABLE Persons  
(  
P\_Idint NOT NULL PRIMARY KEY,  
LastNamevarchar(255) NOT NULL,  
FirstNamevarchar(255),  
Address varchar(255),  
City varchar(255)  
)

**Composite key**

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  
(  
P\_Id int NOT NULL,  
LastName varchar(25) NOT NULL,  
FirstName varchar(25),  
Address varchar(25),  
City varchar(25),  
CONSTRAINT pk\_PersonID PRIMARY KEY (P\_Id,LastName)  
)

## SQL PRIMARY KEY Constraint on ALTER TABLE

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

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

ALTER TABLE Persons  
ADD PRIMARY KEY (P\_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\_PersonID PRIMARY KEY (P\_Id,LastName)

## To DROP a PRIMARY KEY Constraint

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

ALTER TABLE Persons  
DROP CONSTRAINT pk\_PersonID

## SQL FOREIGN KEY Constraint

A FOREIGN KEY in one table points to a PRIMARY KEY in another table.

Let's illustrate the foreign key with an example. Look at the following two tables:

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

**The “Order “ Table**

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

## SQL FOREIGN KEY Constraint on CREATE TABLE

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

CREATE TABLE Orders  
(  
O\_Idint NOT NULL PRIMARY KEY,  
OrderNoint NOT NULL,  
P\_Id int FOREIGN KEY REFERENCES Persons(P\_Id)  
)

## SQL FOREIGN KEY Constraint on ALTER TABLE

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

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

ALTER TABLE Orders  
ADD FOREIGN KEY (P\_Id)  
REFERENCES Persons(P\_Id)

## CHECK Constraint on CREATE TABLE

The following SQL creates a CHECK constraint on the "P\_Id" column when the "Persons" table is created. The CHECK constraint specifies that the column "P\_Id" must only include integers greater than 0.

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

CREATE TABLE Persons  
(  
P\_Idint NOT NULL CHECK (P\_Id>0),  
LastNamevarchar(255) NOT NULL,  
FirstNamevarchar(255),  
Address varchar(255),  
City varchar(255)  
)

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

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

CREATE TABLE Persons  
(  
P\_Idint NOT NULL,  
LastNamevarchar(255) NOT NULL,  
FirstNamevarchar(255),  
Address varchar(255),  
City varchar(255),  
CONSTRAINT chk\_Person CHECK (P\_Id>0 AND City='Sandnes')  
);

## SQL CHECK Constraint on ALTER TABLE

To create a CHECK constraint on the "P\_Id" column when the table is already created, use the following SQL:

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

ALTER TABLE Persons  
ADD CHECK (P\_Id>0)

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

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

ALTER TABLE Persons  
ADD CONSTRAINT chk\_Person CHECK (P\_Id>0 AND City='Sandnes')

## To DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:

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

ALTER TABLE Persons  
DROP CONSTRAINT chk\_Person

## SQL DEFAULT Constraint

The DEFAULT constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified.

## SQL DEFAULT Constraint on CREATE TABLE

The following SQL creates a DEFAULT constraint on the "City" column when the "Persons" table is created:

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

CREATE TABLE Persons  
(  
P\_Id int NOT NULL,  
LastName varchar(25) NOT NULL,  
FirstName varchar(25),  
Address varchar(25),  
City varchar(25) DEFAULT 'Chitwan'  
)

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():

CREATE TABLE Orders  
(  
O\_Id int NOT NULL,  
OrderNo int NOT NULL,  
P\_Id int,  
OrderDate datetime DEFAULT GETDATE()  
);

## SQL DEFAULT Constraint on ALTER TABLE

**SQL Server / MS Access:**

ALTER TABLE Persons  
ALTER COLUMN City SET DEFAULT 'chitwan'

**Oracle:**

ALTER TABLE Persons  
MODIFY City DEFAULT 'SANDNES'

The DROP TABLE Statement

The DROP TABLE statement is used to delete a table.

DROP TABLE table\_name

|  |
| --- |
| More example of creating tables |

**The SQL UPDATE Statement**

The UPDATE statement is used to update existing records in a table.

SQL UPDATE Syntax

UPDATE *table\_name*  
SET *column1*=*value1*,*column2*=*value2*,...  
WHERE *some\_column*=*some\_value*;

|  |  |
| --- | --- |
| **lamp** | **Notice the WHERE clause in the SQL UPDATE statement!** The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated! |

SQL UPDATE Example

Assume we wish to update the customer "AlfredsFutterkiste" with a new contact person and city.

We use the following SQL statement:

UPDATE Customers  
SET ContactName='Alfred Schmidt', City='Hamburg'  
WHERE CustomerName='AlfredsFutterkiste';

* **The SQL DELETE Statement**

The DELETE statement is used to delete rows in a table.

SQL DELETE Syntax

DELETE FROM *table\_name*  
WHERE *some\_column*=*some\_value*;

SQL DELETE Example

Assume we wish to delete the customer "AlfredsFutterkiste" from the "Customers" table.

We use the following SQL statement:

DELETE FROM Customers  
WHERE CustomerName='AlfredsFutterkiste' AND ContactName='Maria Anders';

Delete All Data

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

DELETE FROM *table\_name*;  
  
or  
  
DELETE \* FROM *table\_name*;

**The SQL SELECT Statement**

The SELECT statement is used to select data from a database.

The result is stored in a result table, called the result-set.

SQL SELECT Syntax

SELECT *column\_name*,*column\_name*  
FROM *table\_name*;

and

SELECT \* FROM *table\_name*;

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | AlfredsFutterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglundssnabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

**SELECT Column Example**

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table:

Example

SELECT CustomerName,City FROM Customers;

SELECT \* Example

The following SQL statement selects all the columns from the "Customers" table:

Example

SELECT \* FROM Customers;

1. **The SQL SELECT DISTINCT Statement**

In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.

The DISTINCT keyword can be used to return only distinct (different) values.

SQL SELECT DISTINCT Syntax

SELECT DISTINCT *column\_name*,*column\_name*  
FROM *table\_name*;

**Demo Database**

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | AlfredsFutterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglundssnabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

SELECT DISTINCT Example

The following SQL statement selects only the distinct values from the "City" columns from the "Customers" table:

Example

*SELECT DISTINCT City FROM Customers*;

The WHERE clause is used to filter records.

1. **The SQL WHERE Clause**

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

SQL WHERE Syntax

SELECT *column\_name*,*column\_name*  
FROM *table\_name*  
WHERE *column\_name operator value*;

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | AlfredsFutterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglundssnabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

WHERE Clause Example

The following SQL statement selects all the customers from the country "Mexico", in the "Customers" table:

Example

SELECT \* FROM Customers  
WHERE Country='Mexico';

**Query 0.** Retrieve the birth date and address of the employee(s) whose name

is ‘John B. Smith’.

**Q0: SELECT** Bdate, Address

**FROM** EMPLOYEE

**WHERE** Fname=‘John’ **AND** Minit=‘B’ **AND** Lname=‘Smith’;

Text Fields vs. Numeric Fields

SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

Example

SELECT \* FROM Customers  
WHERE CustomerID=1;

Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

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

Retrieve all employees in department 5 whose salary is between

$30,000 and $40,000.

**Q14: SELECT** \*

**FROM** EMPLOYEE

**WHERE** (Salary **BETWEEN** 30000 **AND** 40000) **AND** Dno = 5;

The condition (Salary **BETWEEN** 30000 **AND** 40000) in Q14 is equivalent to the con

1. **The SQL ORDER BY Keyword**

The ORDER BY keyword is used to sort the result-set by one or more columns. The ORDER BY keyword sorts the records in ascending order by default. To sort the records in a descending order, you can use the DESC keyword.

SQL ORDER BY Syntax

SELECT *column\_name*,*column\_name*  
FROM *table\_name*  
ORDER BY *column\_name*,*column\_name* ASC|DESC;

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | AlfredsFutterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglundssnabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

ORDER BY Example

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

Example

SELECT \* FROM Customers  
ORDER BY Country;

ORDER BY DESC Example

The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column:

Example

SELECT \* FROM Customers  
ORDER BY Country DESC;

## 12 SQL Functions /Aggregate functions

SQL aggregate functions return a single value, calculated from values in a column.

Useful aggregate functions:

* AVG() - Returns the average value
* COUNT() - Returns the number of rows
* MAX() - Returns the largest value
* MIN() - Returns the smallest value
* SUM() - Returns the sum

## The SQL SELECT TOP Clause

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause can be very useful on large tables with thousands of records. Returning a large number of records can impact on performance.

### SQL Server / MS Access Syntax

SELECT TOP number|percent column\_name(s)  
FROM table\_name;

## SQL SELECT TOP Example

The following SQL stament selects the two first records from the "Customers" table:

## Example

SELECT TOP 2 \* FROM Customers;

## SQL SELECT TOP PERCENT Example

The following SQL stament selects the first 50% of the records from the "Customers" table:

## Example

SELECT TOP 50 PERCENT \* FROM Customers;

## The AVG() Function

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

### SQL AVG() Syntax

SELECT AVG(column\_name) FROM table\_name

## SQL AVG() Example

The following SQL statement gets the average value of the "Price" column from the "Products" table:

## Example

SELECT AVG(Price) AS PriceAverage FROM Products;

The following SQL statement selects the "ProductName" and "Price" records that have an above average price:

## Example

SELECT ProductName, Price FROM Products  
WHERE Price>(SELECT AVG(Price) FROM Products);

### SQL COUNT(column\_name) Syntax

The COUNT(column\_name) function returns the number of values (NULL values will not be counted) of the specified column:

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

### SQL COUNT(\*) Syntax

The COUNT(\*) function returns the number of records in a table:

SELECT COUNT(\*) FROM table\_name;

### SQL COUNT(DISTINCT column\_name) Syntax

The COUNT(DISTINCT column\_name) function returns the number of distinct values of the specified column:

SELECT COUNT(DISTINCT column\_name) FROM table\_name;

**Note:** COUNT(DISTINCT) works with ORACLE and Microsoft SQL Server, but not with Microsoft Access.

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10265 | 7 | 2 | 1996-07-25 | 1 |
| 10266 | 87 | 3 | 1996-07-26 | 3 |
| 10267 | 25 | 4 | 1996-07-29 | 1 |

## SQL COUNT(column\_name) Example

The following SQL statement counts the number of orders from "CustomerID"=7 from the "Orders" table:

## Example

SELECT COUNT(CustomerID) AS OrdersFromCustomerID7 FROM Orders  
WHERE CustomerID=7;

## SQL COUNT(\*) Example

The following SQL statement counts the total number of orders in the "Orders" table:

## Example

SELECT COUNT(\*) AS NumberOfOrders FROM Orders;

## SQL COUNT(DISTINCT column\_name) Example

The following SQL statement counts the number of unique customers in the "Orders" table:

## Example

SELECT COUNT(DISTINCT CustomerID) AS NumberOfCustomers FROM Orders;

### SQL COUNT(column\_name) Syntax

The COUNT(column\_name) function returns the number of values (NULL values will not be counted) of the specified column:

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

### SQL COUNT(\*) Syntax

The COUNT(\*) function returns the number of records in a table:

SELECT COUNT(\*) FROM table\_name;

### SQL COUNT(DISTINCT column\_name) Syntax

The COUNT(DISTINCT column\_name) function returns the number of distinct values of the specified column:

SELECT COUNT(DISTINCT column\_name) FROM table\_name;

## The GROUP BY Statement

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.

### SQL GROUP BY Syntax

SELECT column\_name, aggregate\_function(column\_name)  
FROM table\_name  
WHERE column\_name operator value  
GROUP BY column\_name;

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10248 | 90 | 5 | 1996-07-04 | 3 |
| 10249 | 81 | 6 | 1996-07-05 | 1 |
| 10250 | 34 | 4 | 1996-07-08 | 2 |

And a selection from the "Shippers" table:

And a selection from the "Employees" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **EmployeeID** | **LastName** | **FirstName** | **BirthDate** | **Photo** | **Depart No** | **Notes** |
| 1 | Davolio | Nancy | 1968-12-08 | EmpID1.pic | 1 | Education |
| 2 | Fuller | Andrew | 1952-02-19 | EmpID2.pic | 2 | math |
| 3 | Leverling | Janet | 1963-08-30 | EmpID3.pic | 1 | Edunation |
| 4 | Singh | Sunil | 15/2/1880 | Emppic.pic | 1 | Education |
| 5 | Singh | Aryan | 15/1/7878 | Emp12.pic | 1 | Education |

Select departNO,count(employeeid) as ‘total employees’ from employee

Group by departNO

**ADVANCE SQL COMMAND**

## SQL SELECT TOP Clause

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause can be very useful on large tables with thousands of records. Returning a large number of records can impact on performance.

**Note:** Not all database systems support the SELECT TOP clause.

### SQL Server / MS Access Syntax

SELECT TOP number|percent column\_name(s)  
FROM table\_name;

i.e. SELECT TOP 2 \* FROM Customers;

i.e SELECT TOP 50 PERCENT \* FROM Customers;

**Substring Pattern Matching and Arithmetic Operators**

In this section we discuss several more features of SQL. The first feature allows comparison

conditions on only parts of a character string, using the **LIKE** comparison

operator. This can be used for string **pattern matching**. Partial strings are specified

using two reserved characters: % replaces an arbitrary number of zero or more

characters, and the underscore (\_) replaces a single character. For example, consider

the following query.

**Query .** Retrieve all employees whose address is in Houston, Texas.

**Q12: SELECT** Fname, Lname

**FROM** EMPLOYEE

**WHERE** Address **LIKE** ‘%Houston,TX%’;The LIKE operator is used to search for a specified pattern in a column.

### SQL LIKE Syntax

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name LIKE pattern;

i.e. SELECT \* FROM Customers

SQL statement selects all customers with a City starting with the letter "s":  
WHERE City LIKE 's%';

SQL statement selects all customers with a City ending with the letter "s":

SELECT \* FROM Customers  
WHERE City LIKE '%s';

## The IN Operator

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

### SQL IN Syntax

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name IN (value1,value2,...);

**i.e.**

SELECT \* FROM Customers  
WHERE City IN ('Paris','London');

## 4 The SQL BETWEEN Operator

The BETWEEN operator selects values within a range. The values can be numbers, text, or dates.

### SQL BETWEEN Syntax

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_nameBETWEEN value1 AND value2;

SELECT \* FROM Products  
WHERE Price BETWEEN 10 AND 20;

2. SELECT \* FROM Products  
WHERE Price NOT BETWEEN 10 AND 20;

3.SELECT \* FROM Products  
WHERE (Price BETWEEN 10 AND 20)  
AND NOT CategoryID IN (1,2,3);

SELECT \* FROM Products  
WHERE ProductName BETWEEN 'C' AND 'M';

SELECT \* FROM Orders  
WHERE OrderDate BETWEEN #07/04/1996# AND #07/09/1996#;

## SQL Aliases

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

Basically aliases are created to make column names more readable.

### SQL Alias Syntax for Columns

SELECT column\_name AS alias\_name  
FROM table\_name;

### SQL Alias Syntax for Tables

SELECT column\_name(s)  
FROM table\_nameAS alias\_name;

i.e.

SELECT CustomerName AS Customer, ContactName AS [Contact Person]  
FROM Customers;

## The HAVING Clause

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

### SQL HAVING Syntax

SELECT column\_name, aggregate\_function(column\_name)  
FROM table\_name  
WHERE column\_name operator value  
GROUP BY column\_name  
HAVING aggregate\_function(column\_name) operator value;

Select departNO,count(employeeid) as ‘total employees’

from employee

Group by departNO

Having count(empoyeeid)> 2

**Retrieve the names of all employees who do not have supervisors.**

**SELECT** Fname, Lname **FROM** EMPLOYEE **WHERE** Super\_ssn **IS** NULL;

**Using Subqueries as Lists**

Subqueries begin to shine when used as lists. A single value, commonly a column, in the outer query is compared with the subquery’s list by means of the in operators. The subquery must return only a single column; multiple columns will fail.

The in operator returns a value of true if the column value is found anywhere in the list supplied by the subquery, in the same way that where ... in returns a value of true when used with a hard-coded list:

SELECT \*

FROM dbo.Contact

WHERE HomeRegion IN (‘NC’, ‘SC’, ‘GA’, ‘AL’, ‘VA’)

A list subquery serves as a dynamic means of generating the where ... in condition list:

SELECT \*

FROM dbo.Contact

WHERE Region IN (Subquery that returns a list of states)

SELECT ProductName, ProductID

FROM dbo.Product

WHERE ProductID IN

(SELECT ProductID

FROM sales

ORDER BY ProductID)

ORDER BY ProductID

**Using Unions**

The union operation is different from a join. In relational algebra terms, a union is addition, whereas a join is multiplication. Instead of extending a row horizontally as a join would, the union stacks multiple result sets into a single long table.

SELECT \* FROM Table1

UNION

SELECT \* FROM Table2

By default, the UNION operator removes duplicate rows from the result set. If you use ALL, all rows are included in the results and duplicates are not removed.

SELECT \* FROM TableA

UNION ALL

SELECT \* FROM TableB

**Using Joins**

In relational algebra, a join is the multiplication of two data sets followed by a restriction of the result so that only the intersection of the two data sets is returned. The whole purpose of the join is to horizontally merge two data sets (usually tables, but it could be a subquery, view, common table expression, or user-defined function) and produce a new result set from.

**Inner Joins**



The inner join is by far the most common join. In fact, it’s also referred to as a common join, and was originally called a natural join by E. F. Codd. The inner join returns only those rows that represent a match between the two data sets. An inner join is well named because it extracts only data from the inner portion of the intersection of the two overlapping data sets.

**Creating Inner Joins within SQL Code**

Within SQL code, joins are specified within the from portion of the select statement. The keyword join identifies the second table, and the on clause defines the common ground between the two tables. The default type of join is an inner join, so the keyword inner is

optional:

SELECT \*

FROM Table1

[INNER] JOIN Table2

ON Table1.column = Table2.column

Because joins pull together data from two data sets, it makes sense that SQL needs to know how to match up rows from those sets. SQL Server merges the rows by matching a value common to both tables. Typically, a primary key value from one table is being matched with a foreign key value from the secondary table. Whenever a row from the first table matches a row from the second table, the two rows are merged into a new row containing data from both tables.

FOR EXAMPLE.

USE pubs

SELECT titles.title, publishers.pub\_name

FROM titles JOIN publishers

ON titles.pub\_id = publishers.pub\_id

ORDER BY publishers.pub\_name

**Multiple Table Joins**

a select statement isn’t limited to one or two data sources; a SQL Server select statement may refer to up to 256 data sources. That’s a lot of joins. Because SQL is a declarative language, the order of the data sources is not important. Multiple joins may be combined in multiple paths, or even circular patterns (A joins B joins C joins A). An interesting thing happens when joins across multiple tables are combined with a whereclause restriction (that is, when the joins carry with them the where-clause restriction). A restriction in any one table means that only those rows that meet the restriction condition

The following SQL select statement begins with the “who” portion of the question and specifies the join tables and conditions as it works through the required tables. The query that is shown graphically in Management Studio (refer to Figure 9-5) is listed as raw SQL in the following code sample. Notice how the where clause restricts the ProductCategory table rows and yet affects the contacts selected:

SELECT LastName, FirstName, ProductName

FROM Contact

JOIN Order

ON Contact.ContactID = Order].ContactID

JOIN dbo.OrderDetail

ON Order.OrderID = OrderDetail.OrderID

JOIN dbo.Product

ON OrderDetail.ProductID = Product.ProductID

JOIN dbo.ProductCategory

ON Product.ProductCategoryID = ProductCategory.ProductCategoryID

WHERE ProductCategoryName = ‘Kite’

ORDER BY LastName, FirstName

**Outer Joins**

Whereas an inner join contains only the intersection of the two data sets, an outer join extends the inner join by adding the nonmatching data from the left or right data set Some of the data in the result set produced by an outer join will look just like the data from an inner join. There will be data in columns that come from each of the data sources, but any rows from the outer-join table that do not have a match in the other side of the join will return data only from the outer-join table. In this case, columns from the other data source will have null values.

In SQL code, an outer join is declared by the keywords left outer or right outer before the

join (technically, the keyword outer is optional):

SELECT \*

FROM Table1

LEFT|RIGHT [OUTER] JOIN Table2

ON Table1.column = Table2.column

LEFT [OUTER]

Specifies that all rows from the left table not meeting the join condition are included in the result set, and output columns from the other table are set to NULL in addition to all rows returned by the inner join

SELECT au\_lname, au\_fname, titleauthor.title\_id

FROM authors LEFT OUTER JOIN titleauthor

ON authors.au\_id = titleauthor.au\_id

RIGHT [OUTER]

Specifies all rows from the right table not meeting the join condition are included in the result set, and output columns that correspond to the other table are set to NULL, in addition to all rows returned by the inner join.

SELECT au\_lname, au\_fname, titleauthor.title\_id

FROM authors RIGHT OUTER JOIN titleauthor

ON authors.au\_id = titleauthor.au\_id

**FULL [OUTER]**

Specifies that a row from either the left or right table that does not meet the join condition is included in the result set, and output columns that correspond to the other table are set to NULL. This is in addition to all rows usually returned by the INNER JOIN.

SELECT au\_lname, au\_fname, titleauthor.title\_id

FROM authors FULL OUTER JOIN titleauthor

ON authors.au\_id = titleauthor.au\_id

**Discussion and Summary of SQL Queries**

A retrieval query in SQL can consist of up to six clauses, but only the first two SELECT and FROM—are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. The clauses are specified in the following order, with the clauses between square brackets [ ... ] being optional:

**SELECT** <attribute and function list>

**FROM** <table list>

[ **WHERE** <condition> ]

[ **GROUP BY** <grouping attribute(s)> ]

[ **HAVING** <group condition> ]

[ **ORDER BY** <attribute list> ];

The SELECT clause lists the attributes or functions to be retrieved. The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed. GROUP BY specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions

COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they

can also be applied to all the selected tuples in a query without a GROUP BY clause.

Finally, ORDER BY specifies an order for displaying the result of a query.

**Implementing Views**

A view is simply a SELECT statement that has a name and is stored in Microsoft SQL Server. Views act as virtual tables to provide several benefits. A view gives developers a standardized way to execute queries, enabling them to write certain common queries once as views and then include the views in application code so that all applications use the same version of a query. A view can also provide a level of security by giving users access to just a subset of data contained in the base tables that the view is built over and can give users a more friendly, logical view of data in a database. In addition, a view with indexes created on it can provide dramatic performance improvements, especially for certain types of complex queries. Most views allow only read operations on underlying data, but you can also create updateable views that let users modify data via the view.

# CREATE VIEW

Creates a virtual table that represents the data in one or more tables in an alternative way. CREATE VIEW must be the first statement in a query batch.

Syntax

Create view view\_name

(columns)

with ….

As

Select statement

CREATE VIEW titles\_view

AS

SELECT title, type, price, pubdate

FROM titles

**Use WITH ENCRYPTION**

CREATE VIEW accounts (title, advance, amt\_due)

WITH ENCRYPTION

AS

SELECT title, advance, price \* royalty \* ytd\_sales

FROM titles

WHERE price > $5

Use built-in functions within a view

This example shows a view definition that includes a built-in function. When you use functions, the derived column must include a column name in the CREATE VIEW statement

CREATE VIEW categories (category, average\_price)

AS

SELECT type, AVG(price)

FROM titles

GROUP BY type

**Assertion**

* 1. An assertion is a predicate expressing a condition we wish the database to always satisfy
  2. Domain constraints, functional dependency and referential integrity are special forms of assertion.
  3. If a constraint cannot be expressed in these forms, we use an assertion

e.g.

o Sum of loan amounts for each branch is less than the sum of all account

balances at the branch.

o Every loan customer keeps a minimum of $1000 in an account.

General syntax for creating assertion in SQL is

**create assertion** <assertion-name> **check** <predicate>

for example

find the employees who are not gotten the salary till now.

**create assertion** *salary\_status*

**check**

**(not exists (**Select \* from employee

where not exists (select \* from empsalary where employee.empid=empsalary.empid))

o When an assertion is created, the system tests it for validity. If the assertion is valid then

only allow further modification. if test found assertion is violated then it can not go ahead.

o Assertion testing may introduce a significant amount of overhead, especially if the assertions

are complex; hence assertions should be used with great care.

**Trigger**

A trigger is a statement that is automatically executed by the system as a side effect of a

modification to the database. While writing a trigger we must specify

o conditions under which the trigger is executed

o actions to be taken when trigger executes

Triggers re useful mechanism to perform certain task automatically when certain condition/s met. Sometime trigger is also called rule or action rule.

Basic syntax for trigger

CREATE OR REPLACE TRIGGER <TRIGGER NAME>

{BEFORE,AFTER}

{INSERT|DELETE|UPDATE [OF column, . .]} ON <table name>

[FOR EACH ROW [WHEN <condition>]]

DECLARE

Variable declaration;

BEGIN

. . .

END;

**create table student**

**(**

**sid int,**

**sname char(20)**

**)**

**insert into student**

**values(2,'Snil anil')**

**For example 1:**

**CREATE TRIGGER TR1**

**on student**

**for delete**

**as**

**begin**

**select 'deletion is not allowed'**

**rollback tran**

**end**

**delete student**

**where sid=1**

**output:- deletion is not allowed**

Another Example

**CREATE TRIGGER TR2**

**on student1**

**for update,delete**

**as**

**begin**

**select 'updation and deletion is not allowed'**

**rollback tran**

**end**