PCWorkshops

SQL Intermediate

Day 3 Notes
SQL Intermediate 3-Day Course,
An anthology of practical, illustrative SQL Query examples

By: Sarah Barnard

SQL Intermediate 3-Day Course: Day 3

Thank you to my assistant, Dr Mary Smith., for assisting with formatting and editing this document.

SQL Intermediate 3-Day Course: Day 3

Table of Content

Bulk Insert. MS SQL Server.. 7

Bulk Insert Example MS SQL SERVER.. 8

MySQL:.. 8

Inserting records 9

Insert records into a table (1 by 1) 10

Left, Right, Full Join 11

Update records.. 12

Update using Join 13

Delete a specific record 14

SQL - Data Types.. 15

Make a new table = method 1 17

Make a new Table = Method 2 17

Create Tables Exercise 19

Alter Table.. 21

Constraints.. 22

NOT Null.. 24

Not Null Exercise 24

Primary Keys – While creating the table 25

Creating foreign key 27

Primary key and Foreign Key exercise 27

Unique 28

Default 29

CHECK Constraint 30

CHECK CONSTRAINT using ALTER table.. 31

To DROP a CHECK Constraint.. 31

Identity (Auto-increment in MySQL) 32

Identity EXERCISE (Auto-increment in MySQL).. 33

Sequencing.. 34

Indexes.. 35

Drop constraint 37

Drop column 37

Truncate Table / Delete records... 38

Drop table 38

Temporary Tables in SQL Server 40

OPTIONAL Table Variable 41

Table variable.. 42

Difference between CTE and Temp Table and Table Variable.................................... 43

Transaction Control:.. 46

SQL - Clone Tables 53

SQL - Injection 54

Cursor 65

Course Description

DML (Data Manipulation Language) Queries

a. Insert records
b. Update records
c. Delete records

DDL (Data Definition Language) Queries

d. Data types
e. Create table
    a. Constraints
        i. NOT NULL
        ii. UNIQUE
        iii. PRIMARY KEY
        iv. FOREIGN KEY
        v. CHECK
        vi. DEFAULT
        vii. Auto-increment / Identity

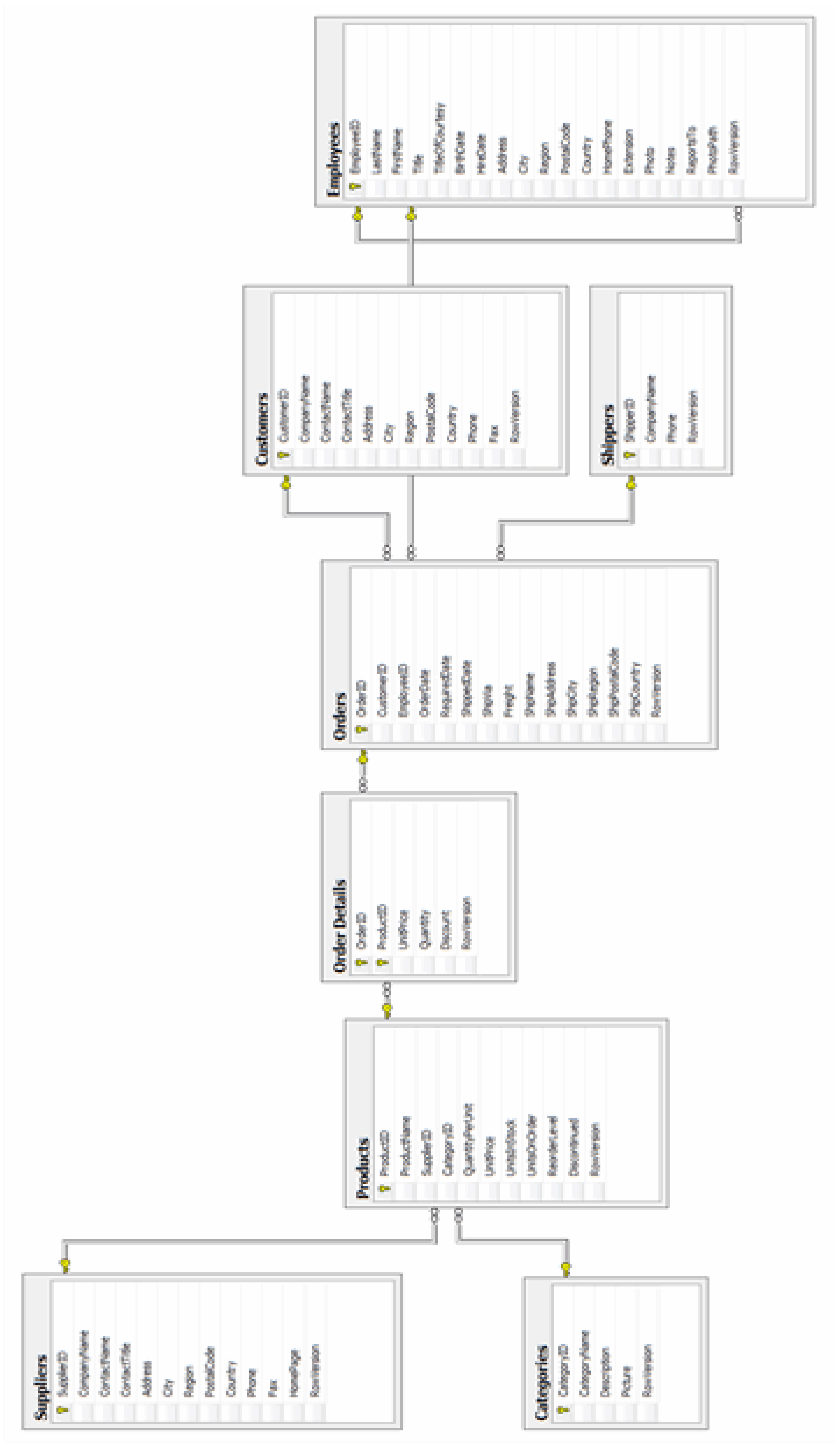
f. Alter Table
    a. Add column
    b. Alter column
    c. Add constraint
    d. Drop Constraint
    e. Drop Column

g. Drop table
h. Indexes

i. Temporary Tables
    i. Local
    ii. Global
    iii. Table variables,
    iv. CTE

j. Transaction Control : Commit, RollBack, Savepoint, Release savepoints

k. SQL Injection


Employees
EmployeeID
LastName
FirstName
Title
TitleOfCourtesy
BirthDate
HireDate
Address
City
Region
PostalCode
Country
HomePhone
Extension
Photo
Notes
ReportsTo
PhotoPath
RowVersion
Customers
CustomerID
CompanyName
ContactName
ContactTitle
Address
City
Region
PostalCode
Country
Phone
Fax
RowVersion
Shippers
ShipperID
CompanyName
Phone
RowVersion
Orders
OrderID
CustomerID
EmployeeID
OrderDate
RequiredDate
ShippedDate
ShipVia
Freight
ShipName
ShipAddress
ShipCity
ShipRegion
ShipPostalCode
ShipCountry
RowVersion
Order Details
OrderID
ProductID
UnitPrice
Quantity
Discount
RowVersion
Products
ProductID
ProductName
SupplierID
CategoryID
QuantityPerUnit
UnitPrice
UnitsInStock
UnitsOnOrder
ReorderLevel
Discontinued
RowVersion
Suppliers
SupplierID
CompanyName
ContactName
ContactTitle
Address
City
Region
PostalCode
Country
Phone
Fax
HomePage
RowVersion
Categories
CategoryID
CategoryName
Description
Picture
RowVersion

Bulk Insert. MS SQL Server

```
Create the file:
'C:\Users\pis\Documents\shippers.txt'
49,'paul',;
77,'mike',;
```

```
BULK INSERT shippers               -Existing table
FROM 'C:\Users\pis\Documents\shippers.txt'
WITH
  (
        FIELDTERMINATOR = ',',
      rowTERMINATOR=';'
  );
```

Terminating character	Indicated by
Tab	\t This is the default field terminator.
Newline character	\n This is the default row terminator.
Carriage return/line feed	\r
Backslash*	\\
Any printable character (control characters are not printable, except null, tab, newline, and carriage return)	(*, A, t, I, and so on)
String of up to 10 printable characters, including some or all of the terminators listed earlier	(**\t**, end, !!!!!!!!!!, \t—\n, and so on)

Bulk Insert Example MS SQL SERVER

```
BULK INSERT
    [dbo].[products3] -- { database_name.schema_name.table_or_view_name
}
      FROM 'C:\Users\u\Desktop\deeq\Data products2.csv'
 WITH
    (
    FIRSTROW = 2
   , KEEPNULLS
   , LASTROW = 70
   , ORDER ( categoryid desc    )
     , ROWTERMINATOR = '\n'
  , FIELDTERMINATOR = ','
    )
```

MySQL:

```
SET @@GLOBAL.local_infile = 1;


LOAD DATA LOCAL INFILE 'C:/Users/u/Desktop/deeq/Data
Customers2.csv'

INTO TABLE `customers` FIELDS TERMINATED BY ','

ENCLOSED BY '"' LINES TERMINATED BY '\n';


select  *  from customers
```

Exercise

Get all the records from the Excel file:

Save txt file and Bulk Insert

The use queries to append them into the tables

Inserting records

Insert records: Method 1

- Insert Into (inserting new records 1 by 1 into a table)

```
INSERT INTO table_name (Column1, Column2, ….)
 VALUES (value1, value2, ….')
```

Insert records: Method 2

- Insert into … select
  - (use select instead of values)
  - (inserting into an EXISTING TABLE from existing table)

```
INSERT INTO categories (categoryID, categoryName) --existing table
SELECT customerID, customername
FROM customers where customerID > 1; --existing table
```

Insert records: Method 3 – NOT MySQL

- Select * … into
  - (inserting into a NEW TABLE from existing table)

```
SELECT *
INTO Shippers5 --new table
FROM customers -- existing table
WHERE customers.customerID>3;
```

Insert records into a table (1 by 1)

```
SYNTAX:
INSERT INTO table_name (Column1, Column2, ….)
 VALUES (value1, value2, ….')
```

e.g.

```
insert into categories (CategoryID, CategoryName, Description)
 values (9, 'food', 'lots')
```

Exercise

1. Insert records into the Category table

Category ID	CategoryName	Description
10	'Deserts'	'Tortuffo'

2. Insert records into the Shipper table

ShipperID	ShipperName	Phone
14	'Deliveroo'	(020) 555-2020'
15	'Mr delivery'	'(020) 333-9999'
16	'Mr Shipper'	'4546767'

3. insert into the Employees Table

employeeID	LastName	FirstName	Birthdate	Photo	Notes
999	Schmidt	Alfred	12/12/2001	jpg.jpg	the new record'
348	Jones	Paul			Marketing Manager
350	Barrett	Sue			Sale Rep
352	Moncado	Yvonne			Sales Agent
346	Smith	John			Owner

Left, Right, Full Join

Exercise

– write down the number of records found for all queries

Do inner join

- Show the ProductName , CategoryName and Description for all the categories
    - – Number of Records: _______
- Show all shipperNames, OrderID – Number of Records: _______
- Show employeeFirstName and OrderID – Number of Records: _______

Do left join

- Show the ProductName , CategoryName and Description for all the categories
    - – Number of Records: _______
- Show all shipperNames, OrderID – Number of Records: _______
- Show employeeFirstName and OrderID – Number of Records: _______
- Find out which categories have no products
- Show all shippers who are not assigned to an order
- Show employees who have not taken orders

Update records

```
UPDATE tablename
SET column=value, column2= value
WHERE column operator condtion
```

Example:

```
UPDATE Customers
SET ContactName='Alfred Schmidt', City='Hamburg'
WHERE CustomerName like 'd%';
```

Exercises (Use Customer Table)

1. In the Customers table, Update the Country from Uk to United Kingdom
2. In the Customers table, Update the Country from USA to United States
3. In the Suppliers table Update the Country **UK** to **United Kingdom**
4. In the Supplierd table, Update the Country from USA to United States

Use Employees Table:

1. Schmidt , Alfred 's Birthdate to 4/7/1982
2. Jones, Paul 's Birthdate to 17/8/1977
3. Barrett, Sue 's Birthdate to 22/2/1965
4. Moncado, Yvonne 's Birthdate to 1/12/1999
5. Smith, John 's Birthdate to 18/1/2001

Update using Join

SQL Server

using join in an update query SQL Server

```
UPDATE Table1
SET Col2 = t2.Col2, Col3 = t2.Col3

        FROM Table1 t1
        INNER JOIN Table2 t2 ON t1.Col1 = t2.Col1
        WHERE t1.Col1 IN (21, 31)
```

Eg:

```
UPDATE Products
SET  Price = 10

        FROM products p
        INNER JOIN categories c  ON  p.categoryID = c.categoryID
        WHERE  categoryname like ‘bevearges%’
```

Exercise

1. For all products where the supplier is on the USA, change the price to Price * 1.24
2. For all orders where the employs name is Nancy, change the order date to today’s date

Mysql

```
UPDATE  products p,
     (SELECT supplierid FROM suppliers
      WHERE country = 'UK')
          AS sup
   SET p.price = p.price* 0.9
   WHERE p.id = sup.id;
```

Delete a specific record

```
DELETE FROM tablename
WHERE       column operator condtion
```

Example:

```
DELETE FROM Customers
WHERE       CustomerName='Alfreds Futterkiste'
        AND ContactName='Maria Anders'
```

Delete Exercises:

Delete the category Deserts
Delete the shippers: 'Deliveroo' , 'Mr delivery' and 'Mr Shipper'
Delete the employees:

- Schmidt , Alfred
- Jones, Paul
- Barrett, Sue
- Moncado, Yvonne
- Smith, John

Delete Exercise:

- Select from orderdetails, where productID = 5
- Keep the ORDERID's somewhere
- Delete from orderdetails, where productID = 5
- Delete these orders from the orders table
- Delete this product
- Do the same for productId 77

SQL - Data Types

SQL Data Type is an attribute that specifies the type of data of any object. Each column, variable and expression has a related data type in SQL. You can use these data types while creating your tables. You can choose a data type for a table column based on your requirement.
SQL Server offers six categories of data types for your use which are listed below −

Exact Numeric Data Types

DATA TYPE	FROM	TO
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1
money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Approximate Numeric Data Types

DATA TYPE	FROM	TO
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

Date and Time Data Types

DATA TYPE	FROM	TO
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	
time	Stores a time of day like 12:30 P.M.	

Note − Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

Character Strings Data Types

Sr.No.	DATA TYPE & Description
1	**char** Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
2	**varchar** Maximum of 8,000 characters.(Variable-length non-Unicode data).
3	**varchar(max)** Maximum length of 231characters, Variable-length non-Unicode data (SQL Server 2005 only).
4	**text** Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

Unicode Character Strings Data Types

Sr.No.	DATA TYPE & Description
1	**nchar** Maximum length of 4,000 characters.(Fixed length Unicode)
2	**nvarchar** Maximum length of 4,000 characters.(Variable length Unicode)
3	**nvarchar(max)** Maximum length of 231characters (SQL Server 2005 only).(Variable length Unicode)
4	**ntext** Maximum length of 1,073,741,823 characters. (Variable length Unicode)

Make a new table = method 1

```
Select * into newtable
 from Existing table
```

Example:

```
SELECT *
 INTO Shippers5 --new table
 FROM Shippers -- existing table
 WHERE Shippers.ShipperID>1;
```

Make a new Table = Method 2

-- create the table

```
CREATE TABLE LocalDeliveries
 (
	Local_ID int primary key NOT NULL,
	Local_ContactName nvarchar(45),
	Local_Name nvarchar(30),
	Local_Phone nvarchar(30),
	Local_ShipperID int
 )
```

Data Types Exercise

1.Create a table with different data types

```
CREATE TABLE  DataTypeExamples
 (
ClubMemberID  int,

Registration Fee float,
MonthlySubscriptionFee real,

OriginalMortage numeric,
MortgageOutstanding decimal,
MortgageRepaymentAmount money,

ClubMemberSalary smallint,
ClubMemberNewSalary int,
ClubMemberIdealSalary bigint,
ClubMemberAge tinyint,
NumberOfHeads bit,

MembersCV nvarchar,
FirstName nvarchar(10),
LastName nchar(10),
FullAddress ntext,

DOB datetime
    )
```

2. Data types Exercise

```
Insert into ( ClubMemberSalary              -- smallint,
            ,ClubMemberNewSalary           ---int,
            ,ClubMemberIdealSalary         -- bigint,
            ,ClubMemberAge                 --tinyint,
            ,NumberOfHeads )               --bit,
Values (100123456789, 100123456789, 100123456789 ,  1000000, 100000)
Insert into (  ClubMemberSalary, ClubMemberAge, ClubMemberIdealSalary)
Values (12.55, 12.50)
Select * from DataTypeExamples

Insert into (  OriginalMortage, MonthlySubscriptionFee)
Values (‘100123456789.50, ‘100123456789.00’)
Select * from DataTypeExamples

Insert into (  Firstname, Lastname ) Values (‘Antonia margaretha’, ‘Smith-van
Heednenberg’)
Select * from DataTypeExamples

Insert your birthday into the DOB field
Select * from DataTypeExamples
```

Create Tables Exercise

```
CREATE TABLE Suppliers(
      SupplierID int NOT NULL,
      SupplierName nvarchar(255) NULL,
      ContactName nvarchar(255) NULL,
      Address nvarchar(255) NULL,
      City nvarchar(255) NULL,
      PostalCode nvarchar(255) NULL,
      Country nvarchar(255) NULL,
      Phone nvarchar(255) NULL,
      Europe nvarchar(20) NULL  )

CREATE TABLE Shippers(
      ShipperID int NOT NULL,
      ShipperName nvarchar(255) NULL,
      Phone nvarchar(250) NOT NULL  )

CREATE TABLE Products(
      ProductID int NOT NULL,
      ProductName nvarchar(255) NULL,
      SupplierID int NULL,
      CategoryID int NULL,
      Unit nvarchar(255) NULL,
      Price int NULL  )

CREATE TABLE Orders(
      OrderID int NOT NULL,
      CustomerID int NULL,
      EmployeeID int NULL,
      OrderDate datetime NULL ,
      ShipperID int NULL  )

CREATE TABLE Orderdetails(
      OrderDetailID int NOT NULL,
      OrderID int NULL,
      ProductID int NULL,
      Quantity float NULL  )

CREATE TABLE Employees(
      EmployeeID int NOT NULL,
      LastName nvarchar(255) NULL,
      FirstName nvarchar(255) NULL,
      BirthDate datetime NULL,
      Photo nvarchar(255) NULL,
      Notes nvarchar(max) NULL  )
```

```
CREATE TABLE Customers(
	CustomerID int NOT NULL,
	CustomerName nvarchar(255) NULL,
	ContactName nvarchar(255) NULL,
	Address nvarchar(255) NULL,
	City nvarchar(255) NULL,
	PostalCode nvarchar(255) NULL,
	Country nvarchar(255) NULL,
	Europe nvarchar(20) NULL   )


CREATE TABLE Categories(
	CategoryID int NOT NULL,
	CategoryName nvarchar(255) NOT NULL,
	Description nvarchar(255) NULL   )
```

Alter Table

With Alter Table , the column structure of a table is altered – You can Alter Table : Add column, Alter Column , Drop column, Add constraint, drop constrainet

ADD a column

```
ALTER TABLE tablename
ADD column  datatype
```

Examples::

```
ALTER TABLE Shippers
ADD City nvarchar(40)

ALTER TABLE Shippers
ADD RegistrationDate datetime

ALTER TABLE Shippers
ADD RegistrationFee int

Select * from Shippers
```

ALTER a column e.g. change the data type

```
ALTER TABLE tablename
ALTER COLUMN column  datatype Not Null
```

Examples::

```
ALTER TABLE Products
ALTER COLUMN Price float
Select * from Products
```

DROP (delete) a column

```
ALTER TABLE tablename
 DROP COLUMN column
```

Examples::

```
ALTER TABLE Shippers
drop column RegistrationFee
Select * from Shippers
```

Add constraint

Refer to individual examples

Drop constraint

```
alter table orders  drop constraint df_ConstraintNAme
```

Constraints

- **NOT NULL** - Indicates that a column cannot store NULL value
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have a 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
- **UNIQUE** - Ensures that each row for a column must have a unique value
- **DEFAULT** - Specifies a default value for a column
- **CHECK** - Ensures that the value in a column meets a specific condition
- **Auto-increment / Identity**

Add constraint

Refer to individual examples

Drop constraint

```
alter table orders  drop constraint df_ConstraintNAme
```

```
MYSQL add constraint

Not null constraint : Simply alter the column

ALTER TABLE Persons
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);

ALTER TABLE User
  ADD CONSTRAINT userProperties
  FOREIGN KEY(properties)
  REFERENCES Properties(ID)

ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);

ALTER TABLE Persons
ALTER City SET DEFAULT 'Sandnes';

ALTER TABLE Persons
ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');
!! : check constraint will be created but ignored in mysql

Drop constraint
MYSQL Drop constraint:
ALTER TABLE Orders DROP primary KEY pk_PerOrders
ALTER TABLE Orders DROP FOREIGN KEY fk_PerOrders
!! : ALTER TABLE Persons DROP INDEX UC_Person; ( Unique
Constraint )
ALTER TABLE Persons ALTER City DROP DEFAULT;
ALTER TABLE Persons DROP CHECK CHK_PersonAge;


etc
```

NOT Null

Example

```
Alter table [dbo].[Categories]
alter column  [CategoryName] nvarchar(255) not null
```

Not Null Exercise

1. Categories Table: Put the NOT NULL constraint on the Categories name
2. Suppliers Table: Put the NOT NULL constraint on the SupplierName
3. Shippers Table: Put the NOT NULL constraint on the ShipperName
4. Customers Table: Put the NOT NULL constraint on the CustomerName
5. Products Table: Put the NOT NULL constraint on the ProductName

Primary Keys – While creating the table

Examples:

a. Create Primary Key WITHOUT a Constraint name

One Primary Key

```
CREATE TABLE CUSTOMERS(
  ID   INT       NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  PRIMARY KEY (ID)
);
```

Composite Primary Key

```
CREATE TABLE CUSTOMERS(
  ID   INT         NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  PRIMARY KEY (ID, NAME)      );
```

--To remove the pk : right click on the explorer on Keys , click modify , right click on the PK, remove the key

b. Create Primary Key with Constraint name

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

c. *Use ALTER TABLE to Add a primary key without a constraint name*

```
Step 1:  (first ensure the field is NOT NULL)
alter table tablename
alter column column int not null

Step 2: (add primary key constraint )
alter table tablename
add PRIMARY KEY (column)
```

Example:

```
alter table Employees
alter column EmployeeID int not null
alter table Employees
add PRIMARY KEY (EmployeeID)
```

d. *Use ALTER TABLE to Add a primary key WITH a constraint name*

```
Step 1:  (first ensure the field is NOT NULL)
alter table tablename
alter column column int not null

Step 2:

ALTER TABLE tablename

  ADD CONSTRAINT constraintname

  PRIMARY KEY ( fieldname );
```

Example:

```
alter table Employees
add constraint PK_EmpID
PRIMARY KEY (EmployeeID)
```

Creating foreign key

a. foreign key when creating the table

```
CREATE TABLE Orders
(
Order_Id int NOT NULL PRIMARY KEY,
OrderNo int NOT NULL,
P_Id int FOREIGN KEY REFERENCES Persons(P_Id)
)
```

b. foreign key when altering the table

```
alter table Orders--table name

add constraint FK_EID – constraint name

foreign key (ShipperID)  -- column that will become a foreign key

references Shippers (ShipperID) -– table and column name to which it links
```

Primary key and Foreign Key exercise

- Change all tables to have a primary keys as per schema
- Add all the foreign keys as per schema

Unique

Example

```
ALTER TABLE [dbo].[Categories]
ADD UNIQUE ([CategoryName]);


ALTER TABLE  [dbo].[Categories]
ADD CONSTRAINT CatUn
UNIQUE ([CategoryName])

ALTER TABLE  [dbo].[Employees]
ADD CONSTRAINT EmpUn
UNIQUE ([FirstName],[LastName])
```

Unique Exercise

1. Categories Table: Put the UNIQUE constraint on the Categories name
2. Suppliers Table: Put the UNIQUE L constraint on the SupplierName
3. Shippers Table: Put the UNIQUE L constraint on the ShipperName
4. Customers Table: Put the UNIQUE constraint on the CustomerName
5. Products Table: Put the UNIQUE constraint on the ProductName

Default

Example: Default

```
CREATE TABLE Persons
(    P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
City varchar(255) DEFAULT 'London'
)
```

Example: Default

```
CREATE TABLE Orders
(    O_Id int NOT NULL,
OrderNo int NOT NULL,
P_Id int,
OrderDate date DEFAULT GETDATE() )
```

Example

```
alter table orders
add constraint df_ConstraintName
default getdate() for orderdate
```

Default Exercise

1. Suppliers Table: Put the DEFAULT constraint on the Country as UK
2. Suppliers Table: Put the DEFAULT constraint on the City as London
3. Customers Table: Put the DEFAULT constraint on the Country as UK
4. Products Table: Put the DEFAULT constraint on the Price as 100
5. Orderdetails Table: Put the DEFAULT constraint on the Quantity as 1
6. Alter the orders table, create a constraint with a constraint name and add a default of the system date to the orderdate field

CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.
If you define a CHECK constraint on a single column it allows only certain values for this column.
If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

a. SQL 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.

```
CREATE TABLE Persons
(
P_Id int NOT NULL CHECK (P_Id>0),
LastName varchar(255) NOT NULL,

)
```

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

```
CREATE TABLE Persons
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
City varchar(255),
CONSTRAINT chk_Person CHECK (P_Id>0 AND City='Sandnes')
)
```

CHECK CONSTRAINT using ALTER table

b. WITHOUT constraint name

```
ALTER TABLE products
ADD CHECK (Price > 0)
```

c. WITH constraint name

```
ALTER TABLE  products
ADD CONSTRAINT chk_Price
CHECK (Price>0 )
```

Example:

```
insert into products (ProductID, ProductName,Price)
values (100,'sock', -10)

insert into products (ProductID, ProductName,Price)
values (100,'sock', 10)
```

To DROP a CHECK Constraint

```
ALTER TABLE  products
drop  CONSTRAINT chk_Price
```

CHECK Constraint Exercise

- Add a constraint to the quantity field on OrderDetails Table, so that quantity >=0

Identity (Auto-increment in MySQL)

The following SQL statement defines the "ID" column to be an auto-increment primary key field in the "Persons" table:

```
CREATE TABLE y
(
  id int IDENTITY(101,1) PRIMARY KEY NOT NULL,  -- 101 is the start value, 1 the step value
  name varchar(20),
);


INSERT INTO y (name) VALUES ('alpha');
INSERT INTO y (name) VALUES ('beta');
```

By default, the starting value for IDENTITY is 1, and it will increment by 1 for each new record.

Start and Step increment values:

ID int NOT NULL identity (100,5),

After the table already has records, you cannot add this constraint to a column. You will have to create a new column and transfer the data

Using the sequence

```
create sequence Seq_cats
as integer
start with 110
increment by 1


insert into categories ( CategoryID, CategoryName )
values ( next value for Seq_cats , 'Pears')
```

Identity EXERCISE (Auto-increment in MySQL)

1. Create a new shippers table, called **shippersNew** with the same fields as on the Shippers
2. Make sure to give the primary key constraint a constrain name
3. Put the identity property on the Primary Key
4. Add 3 new records:

Shippername	tel
Shippers A,	020 7777 3333
Shippers B,	020 7777 7777
Shippers C,	020 7777 9999

```
create table ShipperNew (
SID int  identity(1,1) not null,
SName nvarchar(20),
SPhone nvarchar(20),
CONSTRAINT PK_col1 PRIMARY KEY (SID)
)

insert into ShipperNew( SName, SPhone)  values ('A','020 7777 3333')
insert into ShipperNew( SName, SPhone)  values ('B','020 7777 7777')
insert into ShipperNew( SName, SPhone)  values ('C','020 7777 9999')
select * from ShipperNew
```

MySQL auto-increment
Havinga separate sequence not supported

```
CREATE TABLE employees (
   emp_no INT AUTO_INCREMENT PRIMARY KEY,
   first_name VARCHAR(50),
   last_name VARCHAR(50)
);
```

Sequencing

Renumbering an Existing Sequence

There may be a case when you have deleted many records from a table and you want to re-sequence all the records. This can be done by using a simple trick, but you should be very careful to do this and check if your table is having a join with another table or not.

- If you determine that resequencing an IDENTITY column is unavoidable, the way to do it is
    - to drop the column from the table,
    - then add it again.

```
CREATE SEQUENCE Invoice_Seq
 AS INTEGER
 START WITH 1
 INCREMENT BY 1
 MINVALUE 1
 MAXVALUE 99;

CREATE TABLE Service_Tickets
(ticket_nbr INTEGER DEFAULT NEXT VALUE FOR Invoice_Seq,
 department_code CHAR(1) NOT NULL
   CHECK (department_code IN ('M', 'F')));


INSERT INTO Service_Tickets (department_code)
VALUES ('M');

SELECT * FROM Service_Tickets;
```

Exercise :

- Change Shippernew as created in the last exercise on the previous page,
- Drop primary key constraint and the primary key column
- Add a new column with the constraint again with identity
- Select all records again

Solution :

```
alter table ShipperNew drop constraint PK_col1
alter table ShipperNew drop column SID

alter table ShipperNew add SID int identity (1000,1)
alter table ShipperNew add constraint PK_col1 primary key (SID)

select * from ShipperNew
```

my sql – use auto-increment only

Indexes

An index can be created in a table to find data more quickly and efficiently.
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 you should only create indexes on columns (and tables) that will be frequently searched against.

The CREATE INDEX Command

Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

```
CREATE INDEX index_name
ON table_name (column_name);
```

Composite Indexes

A composite index is an index on two or more columns of a table. Its basic syntax is as follows.

```
CREATE INDEX index_name
on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions. Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

MYSQL1

ALTER TABLE users

ADD UNIQUE INDEX username_unique (username ASC) ;

Add a `UNIQUE` index for the `email` column:1

```
ALTER TABLE users
ADD UNIQUE INDEX  email_unique (email ASC) ;
```

Exercise

1. Create an index on First name in the employee table
2. Create an index on the country in the Customers table
3. Create an index on the country in the Suppliers table
4. Create an index on the ProductName in the Products table
5. Create an index on the CategoryName in the Categories table

DROP an INDEX Constraint

To drop an INDEX constraint, use the following SQL syntax.

```
DROP INDEX index_unit ON dbo.products;
```

Drop constraint

```
alter table orders  drop constraint df_ConstraintNAme
```

Drop column

```
ALTER TABLE OrderDetails
drop column  Quantity2
```

Truncate Table / Delete records

The SQL **TRUNCATE TABLE** command is used to delete complete data from an existing table.

```
TRUNCATE TABLE  table_name;
```

Example:

```
TRUNCATE TABLE CUSTOMERS;
```

```
Truncate table customers
-- bo where clause allowed, all records removed
is the Same as :
delete from Customer
  --note that thee is no WHERE clause so all records will be deleted
  -- where clause is allowed to remove some records only
```

Drop table

You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.

```
Drop table tablename
```

Example:

```
drop table customers
```

Temporary Tables My SQL

By default, all the temporary tables are deleted by MySQL when your database connection gets terminated. Still if you want to delete them in between, then you can do so by issuing a DROP TABLE command.

Following is an example on dropping a temporary table.

```
mysql> CREATE TEMPORARY TABLE SALESSUMMARY (
  -> product_name VARCHAR(50) NOT NULL
  -> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
  -> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
  -> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SALESSUMMARY
  -> (product_name, total_sales, avg_unit_price, total_units_sold)
  -> VALUES
  -> ('cucumber', 100.25, 90, 2);

mysql> SELECT * FROM SALESSUMMARY;
+--------------+-------------+----------------+------------------+
| product_name | total_sales | avg_unit_price | total_units_sold |
+--------------+-------------+----------------+------------------+
| cucumber     |      100.25 |          90.00 |                2 |
+--------------+-------------+----------------+------------------+
1 row in set (0.00 sec)
mysql> DROP TABLE SALESSUMMARY;
mysql>  SELECT * FROM SALESSUMMARY;
ERROR 1146: Table 'TUTORIALS.SALESSUMMARY' doesn't exist
```

Temporary Tables in SQL Server

In SQL Server, temporary tables are created at run-time and you can do all the operations which you can do on a normal table. These tables are created inside Tempdb database. Based on the scope and behavior temporary tables are of two types as given below-

Local Temp Table

Local temp tables are only available to the SQL Server session or connection (means single user) that created the tables. These are automatically deleted when the session that created the tables has been closed. Local temporary table name is stared with single hash ("#") sign.

```
CREATE TABLE #LocalTemp
(
 UserID int,
 Name varchar(50),
 Address varchar(150)
)
GO
insert into #LocalTemp values ( 1, 'Shailendra','Noida');
GO
Select * from #LocalTemp
```

The scope of Local temp table exist to the current session of current user means to the current query window. If you will close the current query window or open a new query window and will try to find above created temp table, it will give you the error.

Global Temp Table

Global temp tables are available to all SQL Server sessions or connections (means all the user). These can be created by any SQL Server connection user and these are automatically deleted when all the SQL Server connections have been closed. Global temporary table name is stared with double hash ("##") sign.

```
CREATE TABLE ##GlobalTemp
(
 UserID int,
 Name varchar(50),
 Address varchar(150)
)
GO
insert into ##GlobalTemp values ( 1, 'Shailendra','Noida');
GO
Select * from ##GlobalTemp
```

Global temporary tables are visible to all SQL Server connections while Local temporary tables are visible to only current SQL Server connection.

OPTIONAL
Table Variable

This acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory. This also allows you to create primary key, identity at the time of Table variable declaration but not non-clustered index.

```
GO
DECLARE @TProduct TABLE
(
SNo INT IDENTITY(1,1),
ProductID INT,
Qty INT
)
--Insert data to Table variable @Product
INSERT INTO @TProduct(ProductID,Qty)
SELECT DISTINCT ProductID, Qty FROM ProductsSales ORDER BY ProductID ASC
--Select data
Select * from @TProduct

--Next batch
GO
Select * from @TProduct --gives error in next batch
```

Note

1. Temp Tables are physically created in the Tempdb database. These tables act as the normal table and also can have constraints, index like normal tables.
2. CTE is a named temporary result set which is used to manipulate the complex sub-queries data. This exists for the scope of statement. This is created in memory rather than Tempdb database. You cannot create any index on CTE.
3. Table Variable acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory.

Table variable

Unlike **Temporary Tables**, they cannot be dropped explicitly. Once the batch execution is finished, the **Table Variables** are dropped automatically. The**Temporary Tables** are stored in tempdb database of **SQL** server. The **Table Variables** are stored in both the memory and the disk in the tempdb database.

Table variables are created like any other variable, using the DECLARE statement. Many believe that table variables exist only in memory, but that is simply not true. They reside in the tempdb database much like local temporary tables. Also like local temporary tables, table variables are accessible only within the session that created them. However, unlike temporary tables the table variable is only accessible within the current batch. They are not visible outside of the batch, meaning the concept of session hierarchy can be somewhat ignored.

As far as performance is concerned table variables are useful with small amounts of data (like only a few rows). Otherwise a temporary table is useful when sifting through large amounts of data. So for most scripts you will most likely see the use of a temporary table as opposed to a table variable. Not to say that one is more useful than the other, it's just you have to choose the right tool for the job.

Table variable

```
declare @beverages AS table
     (supplierName nvarchar(255),
      orderdate datetime,
      orderid Int,
      productname nvarchar(255),
      price float,
      quantity float,
      Suppliercountry nvarchar(255),
      customersCountry nvarchar(255),
      year int,
      month int,
      day int,
      monthname varchar(20)
     )

insert into @beverages
     select supplierName, orderdate, orders.OrderID, productname, price,
      quantity, suppliers.Country,customers.Country,
     year(orderdate), month (OrderDate),
     day(orderdate), datename (month, OrderDate)
     from products
     join orderdetails on products.ProductID =orderdetails.ProductID
     join orders on orderdetails.orderID =orders.orderID
     join customers on orders.customerID =customers.customerID
     join suppliers on products.supplierID=suppliers.supplierID

SELECT year, productname, sum(quantity)
     FROM @beverages
     where Suppliercountry ='brazil' or Suppliercountry ='germany'
     group by year, productname
     order by year desc, productname
```

Difference between CTE and Temp Table and Table Variable

Temp Table or Table variable or CTE are commonly used for storing data temporarily in SQL Server.
Cte: valid for the query
Table Variable: Valid for the batch
Local Temp table : valid for the window
Global Temp Table: Valid for the session

CTE

CTE stands for Common Table expressions. It was introduced with SQL Server 2005. It is a temporary result set and typically it may be a result of complex sub-query. Unlike temporary table its life is limited to the current query. It is defined by using WITH statement. CTE improves readability and ease in maintenance of complex queries and sub-queries. Always begin CTE with semicolon.

A sub query without CTE is given below :

```
SELECT * FROM (
 SELECT Addr.Address, Emp.Name, Emp.Age From Address Addr
 Inner join Employee Emp on Emp.EID = Addr.EID) Temp
WHERE Temp.Age > 50
ORDER BY Temp.NAME
```

By using CTE above query can be re-written as follows :

```
;With CTE1(Address, Name, Age)--Column names for CTE, which are optional
AS
(
SELECT Addr.Address, Emp.Name, Emp.Age from Address Addr
INNER JOIN EMP Emp ON Emp.EID = Addr.EID
)
SELECT * FROM CTE1 --Using CTE
WHERE CTE1.Age > 50
ORDER BY CTE1.NAME
```

When to use CTE

1. This is used to store result of a complex sub query for further use.
2. This is also used to create a recursive query.

Temp table vs table variable	
Temp table	**Table variable**

Below is the sample example of Creating a Temporary Table, Inserting records into it, retrieving the rows from it and then finally dropping the created Temporary Table.
`-- Create Temporary Table`
`CREATE TABLE #Customer`
`(Id INT, Name VARCHAR(50))`
`--Insert Two records`
`INSERT INTO #Customer`
`VALUES(1,'Basavaraj')`
`INSERT INTO #Customer`
`VALUES(2,'Kalpana')`
`--Reterive the records`
`SELECT * FROM #Customer`
`--DROP Temporary Table`
`DROP TABLE #Customer`
`GO`
	Below is the sample example of Declaring a Table Variable, Inserting records into it and retrieving the rows from it.
	`-- Create Table Variable`
	`DECLARE @Customer TABLE`
	`(`
	` Id INT,`
	` Name VARCHAR(50)`
	`)`
	`--Insert Two records`
	`INSERT INTO @Customer`
	`VALUES(1,'Basavaraj')`
	`INSERT INTO @Customer`
	`VALUES(2,'Kalpana')`
	`--Reterive the records`
	`SELECT * FROM @Customer`
	`GO`
	RESULT:

2. MODIFYING STRUCTURE	

Temporary Table structure can be changed after it's creation it implies we can use DDL statements ALTER, CREATE, DROP.
Below script creates a Temporary Table #Customer, adds Address column to it and finally the Temporary Table is dropped.
`--Create Temporary Table`
`CREATE TABLE #Customer`
`(Id INT, Name VARCHAR(50))`
`GO`
`--Add Address Column`
`ALTER TABLE #Customer`
`ADD Address VARCHAR(400)`
`GO`
`--DROP Temporary Table`
`DROP TABLE #Customer`
`GO`
	Table Variables doesn't support DDL statements like ALTER, CREATE, DROP etc, implies we can't modify the structure of Table variable nor we can drop it explicitly.

3. STORAGE LOCATION	
One of the most common MYTH about Temporary Table & Table Variable is that: Temporary Tables are created in TempDB and Table Variables are created In-Memory. Fact is that both are created in TempDB, below Demos prove this reality.	
4. TRANSACTIONS	
Temporary Tables honor the explicit transactions defined by the user.	Table variables doesn't participate in the explicit transactions defined by the user.
5. USER DEFINED FUNCTION	
Temporary Tables are not allowed in User Defined Functions.	Table Variables can be used in User Defined Functions.

6. INDEXES	
Temporary table supports adding Indexes explicitly after Temporary Table creation and it can also have the implicit Indexes which are the result of Primary and Unique Key constraint.	Table Variables doesn’t allow the explicit addition of Indexes after it’s declaration, the only means is the implicit indexes which are created as a result of the Primary Key or Unique Key constraint defined during Table Variable declaration.
7. SCOPE	
There are two types of Temporary Tables, one Local Temporary Tables whose name starts with single # sign and other one is Global Temporary Tables whose name starts with two # signs.Scope of the Local Temporary Table is the session in which it is created and they are dropped automatically once the session ends and we can also drop them explicitly. If a Temporary Table is created within a batch, then it can be accessed within the next batch of the same session. Whereas if a Local Temporary Table is created within a stored procedure then it can be accessed in it’s child stored procedures, but it can’t be accessed outside the stored procedure.Scope of Global Temporary Table is not only to the session which created, but they will visible to all other sessions. They can be dropped explicitly or they will get dropped automatically when the session which created it terminates and none of the other sessions are using it.	Scope of the Table variable is the Batch or Stored Procedure in which it is declared. And they can’t be dropped explicitly, they are dropped automatically when batch execution completes or the Stored Procedure execution completes.

Transaction Control:

Transaction Control:

There are following commands used to control transactions:

- **COMMIT:** to save the changes.
- **ROLLBACK:** to rollback the changes.
- **SAVEPOINT:** creates points within groups of transactions in which to ROLLBACK
- **SET TRANSACTION:** Places a name on a transaction.

Transactional control commands are only used with the DML (data management Language) commands INSERT, UPDATE and DELETE only.
They can not be used while creating tables or dropping them because these operations are automatically commited in the database.

The COMMIT Command:

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.
The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.
The syntax for COMMIT command is as follows:

```
COMMIT;
```

Commit Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

+----+----------+-----+-----------+----------+
```

Following is the example which would delete records from the table having age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS

     WHERE AGE = 25;

SQL> COMMIT;
```

As a result, two rows from the table would be deleted and SELECT statement would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

+----+----------+-----+-----------+----------+
```

The ROLLBACK Command:

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.
The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.
The syntax for ROLLBACK command is as follows:

```
ROLLBACK;
```

RollBack Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+
```

Following is the example, which would delete records from the table having age = 25 and then ROLLBACK the changes in the database.

```
SQL> DELETE FROM CUSTOMERS

     WHERE AGE = 25;

SQL> ROLLBACK;
```

As a result, delete operation would not impact the table and SELECT statement would produce the following result:

```
+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+---------
```

The SAVEPOINT Command:

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.
The syntax for SAVEPOINT command is as follows:

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as follows:

```
ROLLBACK TO SAVEPOINT_NAME;
```

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state:

SavePoint Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Now, here is the series of operations:

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
1 row deleted.
```

ROLLBACK

Now that the three deletions have taken place, say you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone:

```
SQL> ROLLBACK TO SP2;

Rollback complete.
```

Notice that only the first deletion took place since you rolled back to SP2:

```
SQL> SELECT * FROM CUSTOMERS;

+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+

6 rows selected.
```

The RELEASE SAVEPOINT Command:

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for RELEASE SAVEPOINT is as follows:

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the SAVEPOINT.

The SET TRANSACTION Command:

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows.
For example, you can specify a transaction to be read only, or read write.
The syntax for SET TRANSACTION is as follows:

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

SQL - Clone Tables

There may be a situation when you need an exact copy of a table and the CREATE TABLE ... or the SELECT... commands does not suit your purposes because the copy must include the same indexes, default values and so forth.

If you are using MySQL RDBMS, you can handle this situation by adhering to the steps given below –

- Use SHOW CREATE TABLE command to get a CREATE TABLE statement that specifies the source table's structure, indexes and all.
- Modify the statement to change the table name to that of the clone table and execute the statement. This way you will have an exact clone table.
- Optionally, if you need the table contents copied as well, issue an INSERT INTO or a SELECT statement too.

Example

Try out the following example to create a clone table for **TUTORIALS_TBL** whose structure is as follows –

Step 1 – Get the complete structure about the table.

```
SQL> SHOW CREATE TABLE TUTORIALS_TBL \G;
*************************** 1. row ***************************
   Table: TUTORIALS_TBL
Create Table: CREATE TABLE 'TUTORIALS_TBL' (
 'tutorial_id' int(11) NOT NULL auto_increment,
 'tutorial_title' varchar(100) NOT NULL default '',
 'tutorial_author' varchar(40) NOT NULL default '',
 'submission_date' date default NULL,
 PRIMARY KEY  ('tutorial_id'),
 UNIQUE KEY 'AUTHOR_INDEX' ('tutorial_author')
) TYPE = MyISAM
1 row in set (0.00 sec)
```

Step 2 – Rename this table and create another table.

```
SQL> CREATE TABLE `CLONE_TBL` (
 -> 'tutorial_id' int(11) NOT NULL auto_increment,
 -> 'tutorial_title' varchar(100) NOT NULL default '',
 -> 'tutorial_author' varchar(40) NOT NULL default '',
 -> 'submission_date' date default NULL,
 -> PRIMARY KEY  (`tutorial_id'),
 -> UNIQUE KEY 'AUTHOR_INDEX' ('tutorial_author')
-> ) TYPE = MyISAM;
Query OK, 0 rows affected (1.80 sec)
```

Step 3 – After executing step 2, you will clone a table in your database. If you want to copy data from an old table, then you can do it by using the INSERT INTO... SELECT statement.

```
SQL> INSERT INTO CLONE_TBL (tutorial_id,
  ->                    tutorial_title,
  ->                    tutorial_author,
  ->                    submission_date)
  -> SELECT tutorial_id,tutorial_title,
  ->        tutorial_author,submission_date,
  -> FROM TUTORIALS_TBL;
Query OK, 3 rows affected (0.07 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Finally, you will have an exact clone table as you wanted to have.

SQL - Injection

If you take a user input through a webpage and insert it into a SQL database, there is a chance that you have left yourself wide open for a security issue known as the **SQL Injection**. This chapter will teach you how to help prevent this from happening and help you secure your scripts and SQL statements in your server side scripts such as a PERL Script.

Injection usually occurs when you ask a user for input, like their name and instead of a name they give you a SQL statement that you will unknowingly run on your database. Never trust user provided data, process this data only after validation; as a rule, this is done by **Pattern Matching**.

In the example below, the **name** is restricted to the alphanumerical characters plus underscore and to a length between 8 and 20 characters (modify these rules as needed).

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches)) {
   $result = mysql_query("SELECT * FROM CUSTOMERS
      WHERE name = $matches[0]");
} else {
   echo "user name not accepted";
}
```

To demonstrate the problem, consider this excerpt −

```
// supposed input
$name = "Qadir'; DELETE FROM CUSTOMERS;";
mysql_query("SELECT * FROM CUSTOMSRS WHERE name='{$name}'");
```

The function call is supposed to retrieve a record from the CUSTOMERS table where the name column matches the name specified by the user. Under normal circumstances, **$name** would only contain alphanumeric characters and perhaps spaces, such as the string ilia. But here, by appending an entirely new query to $name, the call to the database turns into disaster; the injected DELETE query removes all records from the CUSTOMERS table.

Fortunately, if you use MySQL, the **mysql_query()** function does not permit query stacking or executing multiple SQL queries in a single function call. If you try to stack queries, the call fails.

However, other PHP database extensions, such as **SQLite** and **PostgreSQL** happily perform stacked queries, executing all the queries provided in one string and creating a serious security problem.

Preventing SQL Injection

You can handle all escape characters smartly in scripting languages like PERL and PHP. The MySQL extension for PHP provides the function **mysql_real_escape_string()** to escape input characters that are special to MySQL.

```
if (get_magic_quotes_gpc()) {
   $name = stripslashes($name);
}
$name = mysql_real_escape_string($name);
mysql_query("SELECT * FROM CUSTOMERS WHERE name='{$name}'");
```

The LIKE Quandary

To address the LIKE quandary, a custom escaping mechanism must convert user-supplied '%' and '_' characters to literals. Use **addcslashes()**, a function that lets you specify a character range to escape.

```
$sub = addcslashes(mysql_real_escape_string("%str"), "%_");
// $sub == \%str\_
mysql_query("SELECT * FROM messages
   WHERE subject LIKE '{$sub}%'");
```

Binary Data Types

Sr.No.	DATA TYPE & Description
1	**binary** Maximum length of 8,000 bytes(Fixed-length binary data)
2	**varbinary** Maximum length of 8,000 bytes.(Variable length binary data)
3	**varbinary(max)** Maximum length of 231 bytes (SQL Server 2005 only). (Variable length Binary data)
4	**image** Maximum length of 2,147,483,647 bytes. (Variable length Binary Data)

The `BINARY` and `VARBINARY` types are similar to `CHAR` and `VARCHAR`, except that they contain binary strings rather than nonbinary strings. That is, they contain byte strings rather than character strings. This means they have the `binary` character set and collation, and comparison and sorting are based on the numeric values of the bytes in the values.
The permissible maximum length is the same for `BINARY` and `VARBINARY` as it is for `CHAR` and `VARCHAR`, except that the length for `BINARY` and `VARBINARY` is a length in bytes rather than in characters.
The `BINARY` and `VARBINARY` data types are distinct from the `CHAR BINARY` and `VARCHAR BINARY` data types. For the latter types, the `BINARY` attribute does not cause the column to be treated as a binary string column. Instead, it causes the binary (`_bin`) collation for the column character set to be used, and the column itself contains nonbinary character strings rather than binary byte strings. For example, `CHAR(5) BINARY` is treated as `CHAR(5) CHARACTER SET latin1 COLLATE latin1_bin`, assuming that the default character set is `latin1`. This differs from `BINARY(5)`, which stores 5-bytes binary strings that have the `binary` character set and collation. For information about differences between binary strings and binary collations for nonbinary strings, see Section 10.1.8.5, “The binary Collation Compared to _bin Collations”.

Additional Topics

- @@CONNECTIONS
- @@MAX_CONNECTIONS
- @@CPU_BUSY
- @@ERROR
- @@IDENTITY
- @@IDLE
- @@IO_BUSY
- @@LANGID
- @@LANGUAGE
- @@MAXCHARLEN
- @@PACK_RECEIVED
- @@PACK_SENT
- @@PACKET_ERRORS
- @@ROWCOUNT
- @@SERVERNAME
- @@SPID
- @@TEXTSIZE
- @@TIMETICKS
- @@TOTAL_ERRORS
- @@TOTAL_READ / @@TOTAL_WRITE
- @@TRANCOUNT
- @@VERSION

@@CONNECTIONS

The number of logins or attempted logins since SQL Server was last started.

Return type: `int`

Example

```
SELECT GETDATE() AS 'Today''s Date and Time',
@@CONNECTIONS AS 'Login Attempts'
```

Output

```
Today's Date and Time   Login Attempts
----------------------- --------------
2009-08-19 21:44:32.140 1430
```

@@MAX_CONNECTIONS

The maximum number of simultaneous connections that can be made with SQL Server in this computer environment. The user can configure SQL Server for any number of connections less than or equal to the value of @@max_connections with sp_configure ''number of user connections''.

Return type: int

Example

```
SELECT @@MAX_CONNECTIONS AS 'Max Connections'
```

Output

```
Max Connections
---------------
32767
```

@@CPU_BUSY

The amount of time, in ticks, that the CPU has spent doing SQL Server work since the last time SQL Server was started.

Return type: int

Example

```
SELECT @@CPU_BUSY * CAST(@@TIMETICKS AS FLOAT) AS 'CPU microseconds',
   GETDATE() AS 'As of' ;
```

Output

```
CPU microseconds       As of
---------------------- -----------------------
2812500                2009-08-19 21:47:27.187
```

@@ERROR

Commonly used to check the error status (succeeded or failed) of the most recently executed statement. It contains 0 if the previous transaction succeeded; otherwise, it contains the last error number generated by the system. A statement such as:

Return type: int

Example

```
IF @@ERROR <> 0
    PRINT  'Your error message';
```

Output

```
Your error message
```

IF @@ERROR != 0 return causes an exit if an error occurs.

Every Transact-SQL statement resets @@error, including print statements or if tests, so the status check must immediately follow the statement whose success is in question.

@@IDENTITY

The last value inserted into an `IDENTITY` column by an `insert` or `select` into statement. `@@identity` is reset each time a row is inserted into a table. If a statement inserts multiple rows, `@@identity` reflects the `IDENTITY` value for the last row inserted. If the affected table does not contain an `IDENTITY` column, `@@identity` is set to `0`.

The value of `@@identity` is not affected by the failure of an `insert` or `select` into statement, or the rollback of the transaction that contained it. `@@identity` retains the last value inserted into an `IDENTITY` column, even if the statement that inserted it fails to commit.

Return type: `numeric(38,0)`

Example

```
INSERT INTO [TempE].[dbo].[CaseExpression]
           ([Code])   VALUES (5)
GO
SELECT @@IDENTITY AS 'Identity';
```

Output

```
Identity
---------------------------------------
5
```

@@IDLE

The amount of time, in ticks, that SQL Server has been idle since it was last started.

Return type: `int`

Example

```
SELECT @@IDLE * CAST(@@TIMETICKS AS float) AS 'Idle microseconds',
   GETDATE() AS 'as of'
```

Output

```
Idle microseconds      as of
---------------------- -----------------------
11340000000            2009-08-19 22:07:19.903
```

@@IO_BUSY

The amount of time, in ticks, that SQL Server has spent doing input and output operations since it was last started.

Return type: int

Example

```
SELECT @@IO_BUSY*@@TIMETICKS AS 'IO microseconds',
   GETDATE() AS 'as of'
```

Output

```
IO microseconds as of
--------------- -----------------------
5906250         2009-08-19 22:09:44.013
```

@@LANGID

The local language id of the language currently in use (specified in syslanguages.langid).

Return type: smallint

Example

```
SET LANGUAGE 'Italian'
SELECT @@LANGID AS 'Language ID'
SET LANGUAGE 'us_english'
SELECT @@LANGID AS 'Language ID'
```

Output

```
L'impostazione della lingua è stata sostituita con Italiano.
Language ID
-----------
6
Changed language setting to us_english.
Language ID
-----------
0
```

@@LANGUAGE

The name of the language currently in use (specified in `syslanguages.name`).

Return type: `nvarchar`

Example

```
SELECT @@LANGUAGE AS 'Language Name';
```

Output

```
Language Name
-------------
us_english
```

@@MAXCHARLEN

The maximum length, in bytes, of a character in SQL Server's default character set.

Return type: `tinyint`

Example

```
SELECT @@MAX_PRECISION AS 'Max Precision'
```

Output

```
Max Precision
-------------
38
```

@@PACK_RECEIVED

The number of input packets read by SQL Server since it was last started.

Return type: `int`

Example

```
SELECT @@PACK_RECEIVED AS 'Packets Received'
```

Output

```
Packets Received
----------------
8998
```

@@PACK_SENT

The number of output packets written by SQL Server since it was last started.

Return type: `int`

Example

```
SELECT @@PACK_SENT AS 'Pack Sent'
```

Output

```
Pack Sent
-----------
9413
```

@@PACKET_ERRORS

The number of errors that have occurred while SQL Server was sending and receiving packets.

Return type: `int`

Example

```
SELECT @@PACKET_ERRORS AS 'Packet Errors'
```

Output

```
Packet Errors
-------------
0
```

@@ROWCOUNT

The number of rows affected by the last command. `@@rowcount` is set to `0` by any command which does not return rows, such as an `if` statement. With cursors, `@@rowcount` represents the cumulative number of rows returned from the cursor result set to the client, up to the last fetch request.

Return type: `int`

Example

```
IF @@ROWCOUNT = 0
 PRINT 'Warning: No rows were updated';
```

Output

```
'Warning: No rows were updated'
```

@@SERVERNAME

The name of the local SQL Server. You must define a server name with `sp_addserver`, and then restart SQL Server.

Return type: `varchar`

Example

```
SELECT @@SERVERNAME AS 'Server Name'
```

Output

```
MY_SERVER_WINDOWS_2003
```

@@SPID

The server process ID number of the current process.

Return type: `smallint`

Example

```
SELECT @@SPID AS 'ID', SYSTEM_USER AS 'Login Name', USER AS 'User Name'
```

Output

```
ID     Login Name                                       User Name
------ ----------------------------------------------------
55     MY_SERVER_WINDOWS_2003\Administrator                dbo
```

@@TEXTSIZE

The current value of the set textsize option, which specifies the maximum length, in bytes, of text or image data to be returned with a `select` statement. Defaults to 32K.

Return type: `smallint`

Example

```
SET TEXTSIZE 2048
SELECT @@TEXTSIZE AS 'Text Size'
```

Output

```
Text Size
-----------
2048
```

@@TIMETICKS

The number of microseconds per tick. The amount of time per tick is machine dependent.

Return type: int

Example

```
SELECT @@TIMETICKS AS 'Time Ticks';
```

Output

```
Time Ticks
-----------
31250
```

@@TOTAL_ERRORS

The number of errors that have occurred while SQL Server was reading or writing.

Return type: int

Example

```
SELECT @@TOTAL_ERRORS AS 'Errors', GETDATE() AS 'As of'
```

Output

```
Errors      As of
----------- -----------------------
0           2009-08-19 22:47:51.937
```

@@TOTAL_READ / @@TOTAL_WRITE

The number of disk reads by SQL Server since it was last started.

Return type: int

Example

```
SELECT @@TOTAL_READ AS 'Reads', @@TOTAL_WRITE AS 'Writes', GETDATE() AS 'As of'
```

Output

```
Reads       Writes      As of
----------- ----------- -----------------------
861         91          2009-08-19 23:36:26.763
```

@@TRANCOUNT

The nesting level of transactions. Each begin transaction in a batch increments the transaction count. When you query `@@trancount` in chained transaction mode, its value is never zero since the query automatically initiates a transaction.

Return type: `int`

Example

```
PRINT @@TRANCOUNT
--  The BEGIN TRAN statement will increment the
--  transaction count by 1.
BEGIN TRAN
    PRINT @@TRANCOUNT
    BEGIN TRAN
        PRINT @@TRANCOUNT
--  The COMMIT statement will decrement the transaction count by 1.
    COMMIT
    PRINT @@TRANCOUNT
COMMIT
PRINT @@TRANCOUNT
```

Output

```
0
1
2
1
0
```

@@VERSION

The date of the current version of SQL Server.

Return type: `nvarchar`

Example

```
SELECT @@VERSION AS 'SQL Server Version'
```

Output

```
Jul  9 2008 14:43:34
Copyright (c) 1988-2008 Microsoft Corporation
Enterprise Edition on Windows NT 5.2 <X86> (Build 3790: Service Pack 2)
```

Cursor

In computer science, a database **cursor** is a control structure that enables traversal over the records in a database. Cursors facilitate subsequent processing in conjunction with the traversal, such as retrieval, addition and removal of database records. The database **cursor** characteristic of traversal makes cursors akin to the programming language concept of iterator.

```
-- cursor keeps the whole record set and
-- then allows you to iterate over all the rows in the resultset
DECLARE
    @product_name VARCHAR(255),
    @list_price   DECIMAL;

DECLARE cursor_product CURSOR
FOR SELECT productname,  price  FROM products where price > 100;
--Next, open the cursor:

OPEN cursor_product;
--Then, fetch each row from the cursor
--and print out the product name and list price:

FETCH NEXT FROM cursor_product INTO  @product_name,  @list_price;
WHILE @@FETCH_STATUS = 0
    BEGIN
        print  @product_name
                    + ', Price: $'
                    + cast(@list_price*1.33 as nvarchar)
             -- vs select @product_name ,@list_price;
        FETCH NEXT FROM cursor_product INTO @product_name, @list_price;
    END;
--After that, close the cursor:

CLOSE cursor_product;
--Finally, deallocate the cursor to release it.

DEALLOCATE cursor_product;
```

```
DECLARE
    @product_name VARCHAR(255),
    @price    float;

SELECT *
INTO #temp1
 FROM (
 SELECT productname,  price  FROM products where price > 100
 ) AS x

DECLARE @Count INT
DECLARE cursor_product CURSOR
             FOR SELECT productname,  price  FROM #temp1;

OPEN cursor_product
FETCH NEXT FROM cursor_product INTO @product_name, @price
    WHILE (@@FETCH_STATUS = 0)
        BEGIN
                    print  @product_name
                           + ', Price: $'
                           + cast(@price*1.33 as nvarchar)
             FETCH NEXT FROM cursor_product INTO @product_name, @price
        END
CLOSE cursor_product
DEALLOCATE cursor_product
DROP TABLE #temp1
```