Basics of Transact-SQL

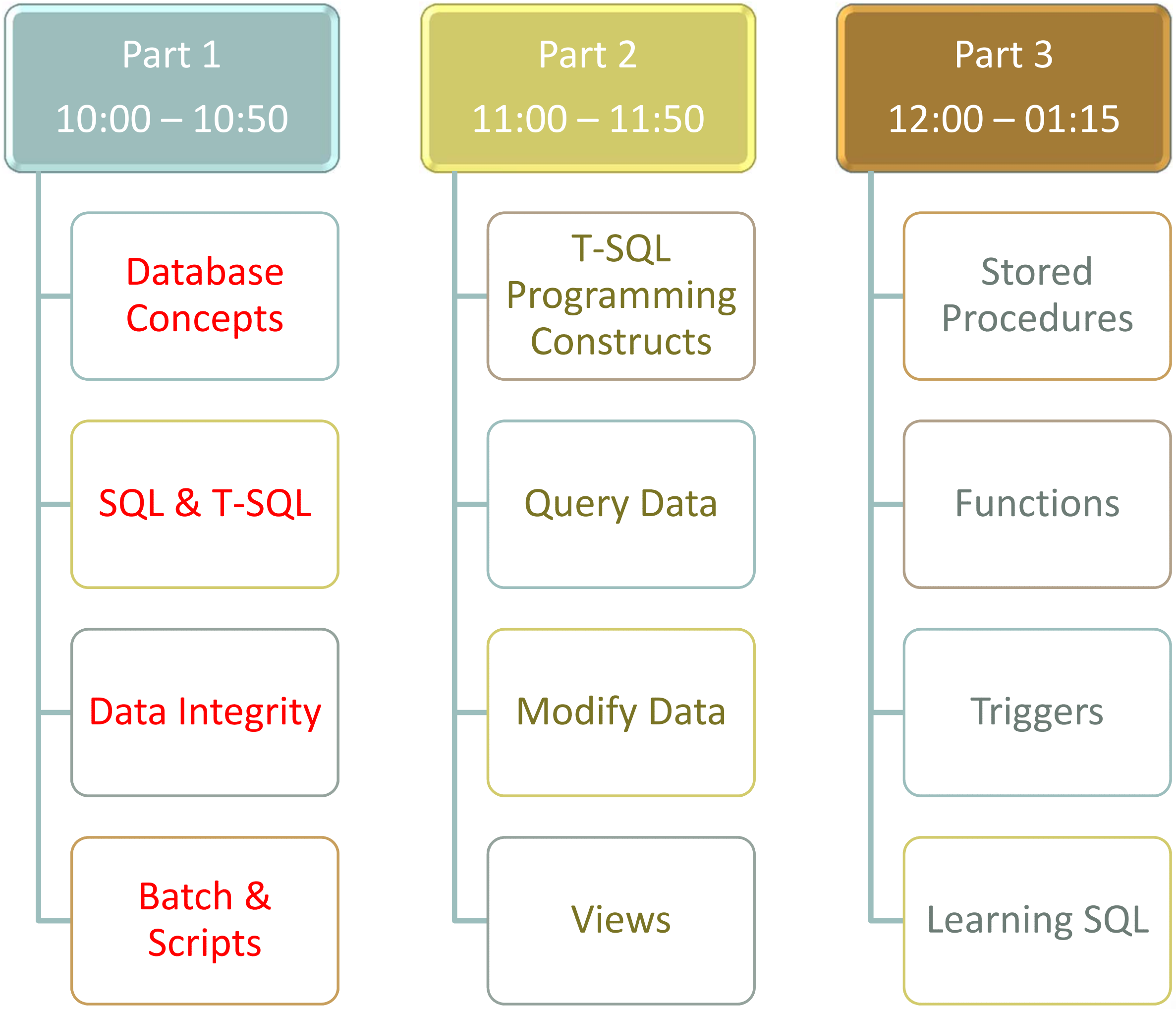
(Beginner Level)

Abhishek SHARMA

Pre-requisites

- Experience in any programming language
- Basic understanding of SQL
- Software required
  - SQL Server 2005
  - SQL Server 2005 Management Studio
- Databases
  - AdventureWorks
  - Northwind
  - Pubs

Agenda



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

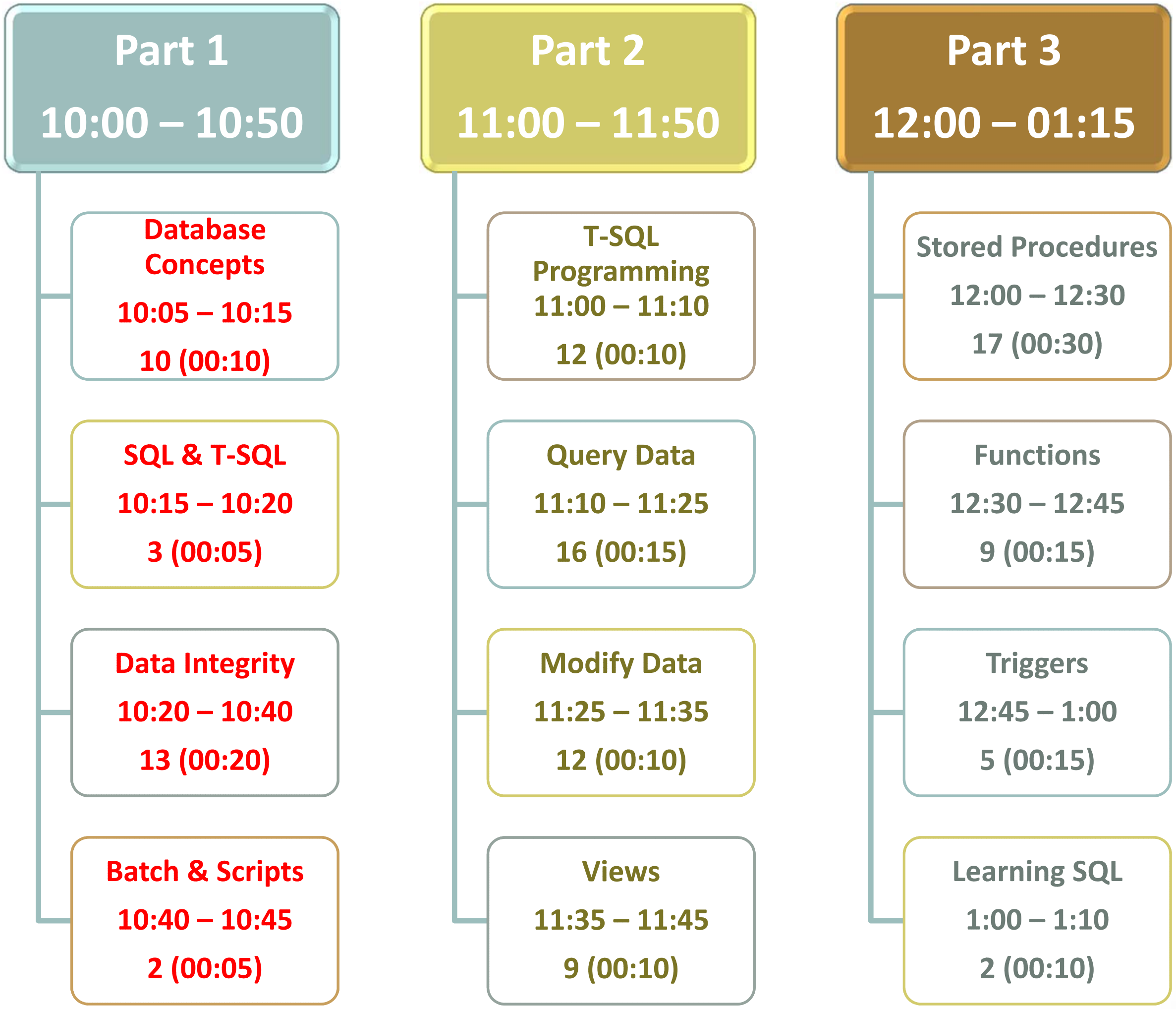
- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

Time plan



Part 1

10:00 – 10:50

- Database Concepts
10:05 – 10:15
10 (00:10)
- SQL & T-SQL
10:15 – 10:20
3 (00:05)
- Data Integrity
10:20 – 10:40
13 (00:20)
- Batch & Scripts
10:40 – 10:45
2 (00:05)

Part 2

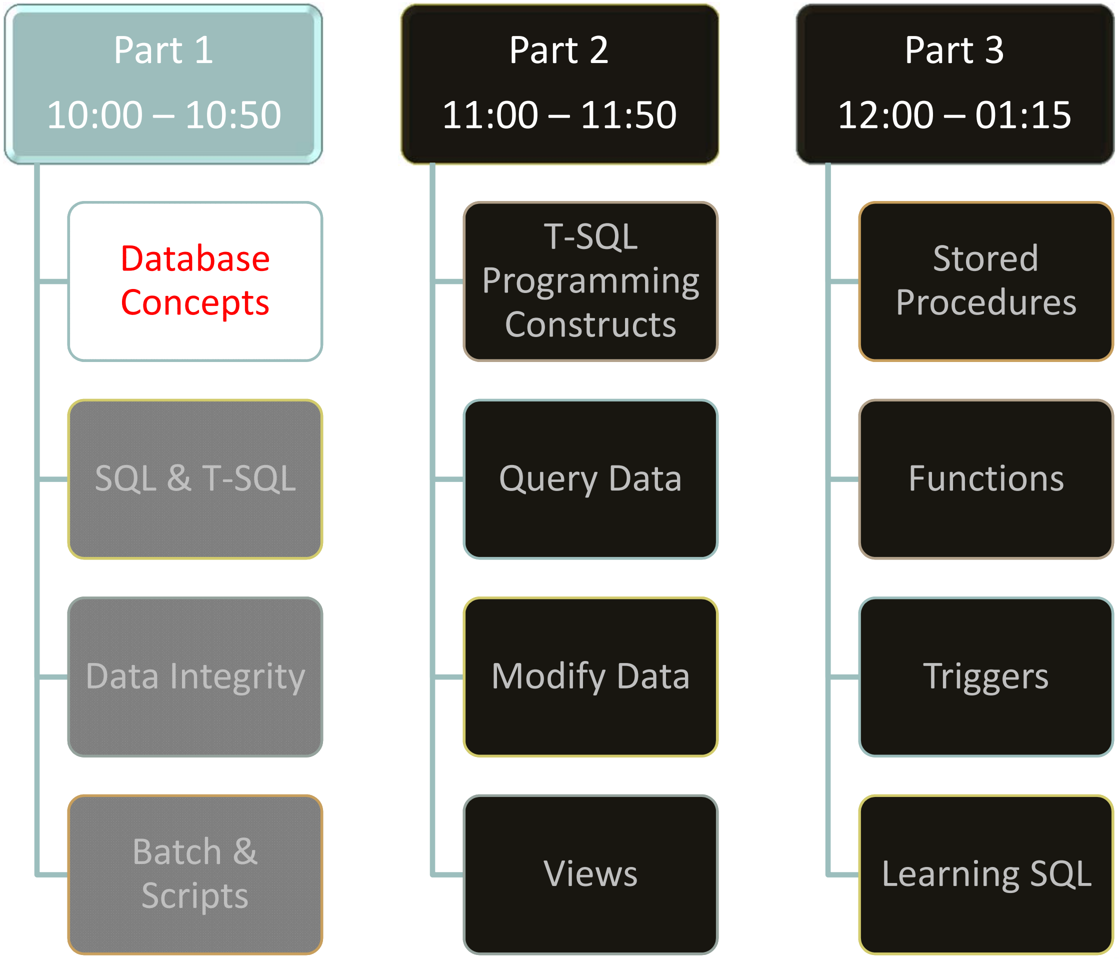
11:00 – 11:50

- T-SQL Programming
11:00 – 11:10
12 (00:10)
- Query Data
11:10 – 11:25
16 (00:15)
- Modify Data
11:25 – 11:35
12 (00:10)
- Views
11:35 – 11:45
9 (00:10)

Part 3

12:00 – 01:15

- Stored Procedures
12:00 – 12:30
17 (00:30)
- Functions
12:30 – 12:45
9 (00:15)
- Triggers
12:45 – 1:00
5 (00:15)
- Learning SQL
1:00 – 1:10
2 (00:10)



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

Introducing Database

- A database is a structured collection of data
- More accurate definition – A database is a container for objects that not only store data, but also enable data storage and retrieval to operate in a secure and safe manner
  - Database are computer files that are optimized to store data in a structured way
- DBMS is the software to access and / or maintain the database file
- In a relational database, data is stored in the form of tables
  - Developed by Dr. Codd in 1970
- SQL Server is a RDBMS
- Meaning of Relation

Codd’s 12 Rules for an RDBMS

- Rule 1: The Information Rule
- Rule 2: Guaranteed Access Rule
- Rule 3: Systematic Treatment of NULL Values
- Rule 4: Dynamic On-Line Catalog Based on the Relational Model
- Rule 5: Comprehensive Data Sublanguage Rule
- Rule 6: View Updating Rule
- Rule 7: High-Level Insert, Update, and Delete
- Rule 8: Physical Data Independence
- Rule 9: Logical Data Independence
- Rule 10: Integrity Independence
- Rule 11: Distribution Independence
- Rule 12: Non-Subversion Rule

Benefits of Relational DBMS

- Data types for data correctness
- Eliminate redundant data
- Safe operations so that related data are not deleted accidently
- Efficient retrieval and manipulation
- Secured access to the data
- Common language for data manipulation (SQL)
- Reliability
- Availability

Introducing Normalization

- Normalization is the process of design to structure the tables of database to
  - eliminate redundancy and
  - provide easy data access & safe manipulation
  - Keep balance between disk space & performance
- Normalization is done by defining keys, new relationships and entities
- During the process the relation moves from a lower to higher normal form

3 Normal Forms

- A relation is in
  - First Normal Form (1NF)
    - It has a primary key
    - Each column is atomic
    - No repeating group of columns
  - Second Normal Form (2NF)
    - It is in 1NF
    - Every non-key column is completely functional dependent on the PK
  - Third Normal Form (3NF)
    - It is in 2NF
    - Every non-key column depend only on the PK

Example – Normal Forms

VendorName	InvoiceNumber	Item1	Item2	Details
HP	1023	LT D120	DT P650	Laptop; Desktop
Dell	D1278	PC P230	NULL	Desktop

InvoiceID	VendorName	InvoiceNumber	InvoiceSequence	ItemDescription
1	HP	1023	1	LT D120
1	HP	1023	2	DT P650
2	Dell	D1278	1	PC P230

InvoiceID	VendorName	VendorAdress	VendorPhone	InvoiceNo
1	HP	1023	40-4536475	HP01
2	HP	1023	44-8347374	DL05

Summarizing a Well-Designed Database

- A table should have an identifier
- A table should store only data for a single type of entity
- A table should avoid nullable columns
- A table should not have repeatable values or columns

A Table

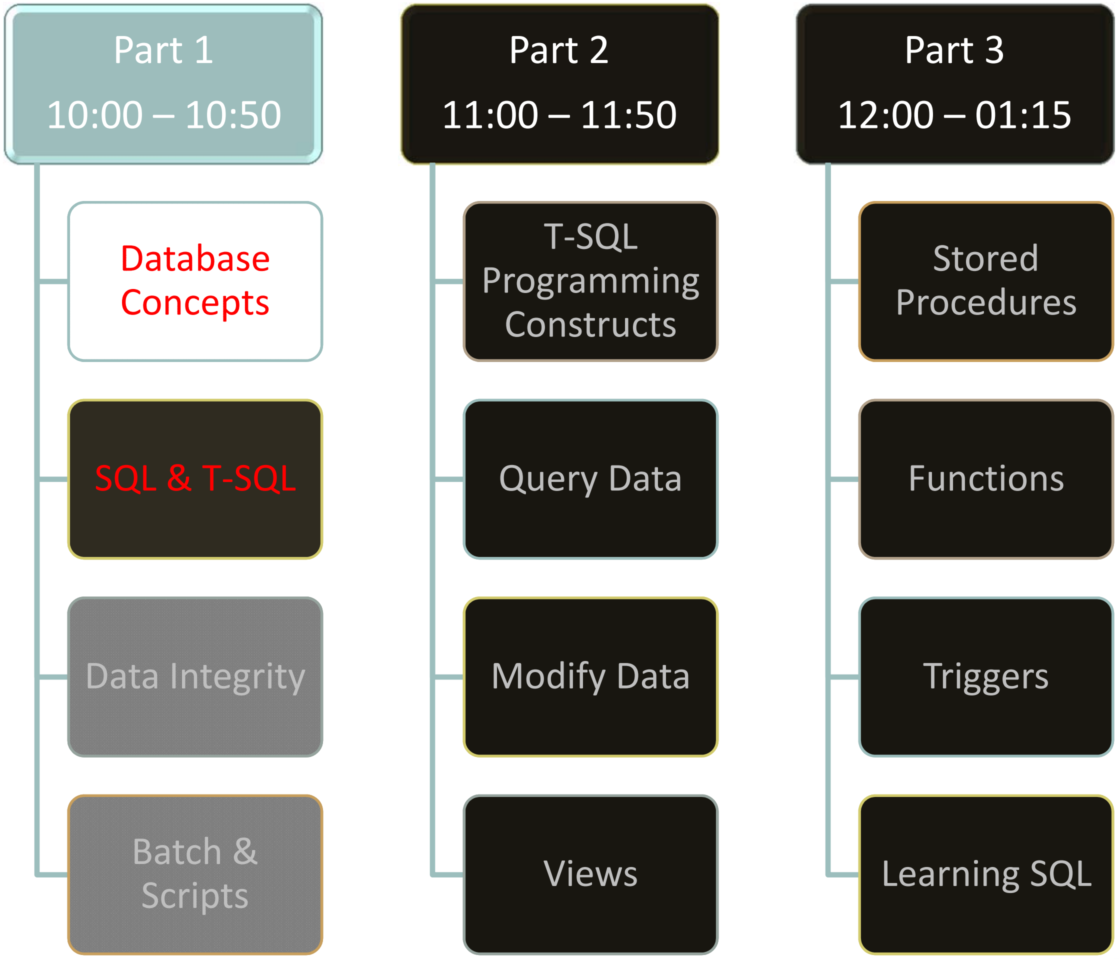
- Tables consists of rows and columns
  - Also known as Entity or Relation
- Rows (record or tuples)
  - Are unordered in the table
  - No of rows known as cardinality
- Columns (fields or attributes)
  - Data in column is atomic
  - Has a type
    - Maintain data integrity
    - Allocate the proper amount of physical space
- A table is modeled after a real world entity (though not always)

Database objects in SQL Server

- Entities
  - Tables
  - Views
- Programming Objects
  - Functions
  - Procedures
  - Triggers
- Data Integrity
  - Constraints
  - Indexes

Object Names in SQL Server

- Fully qualified name
  - [ServerName.[DatabaseName.[SchemaName.]]]ObjectName
- Schema Name
  - Was referred to as owner in previous versions of SQL Server
  - Schema is an ANSI compliant term
  - Ownership meant ‘who own the object’
  - In SQL Server 2005 object is assigned to schema instead of owner
  - A schema can now be shared across multiple logins



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

History of SQL

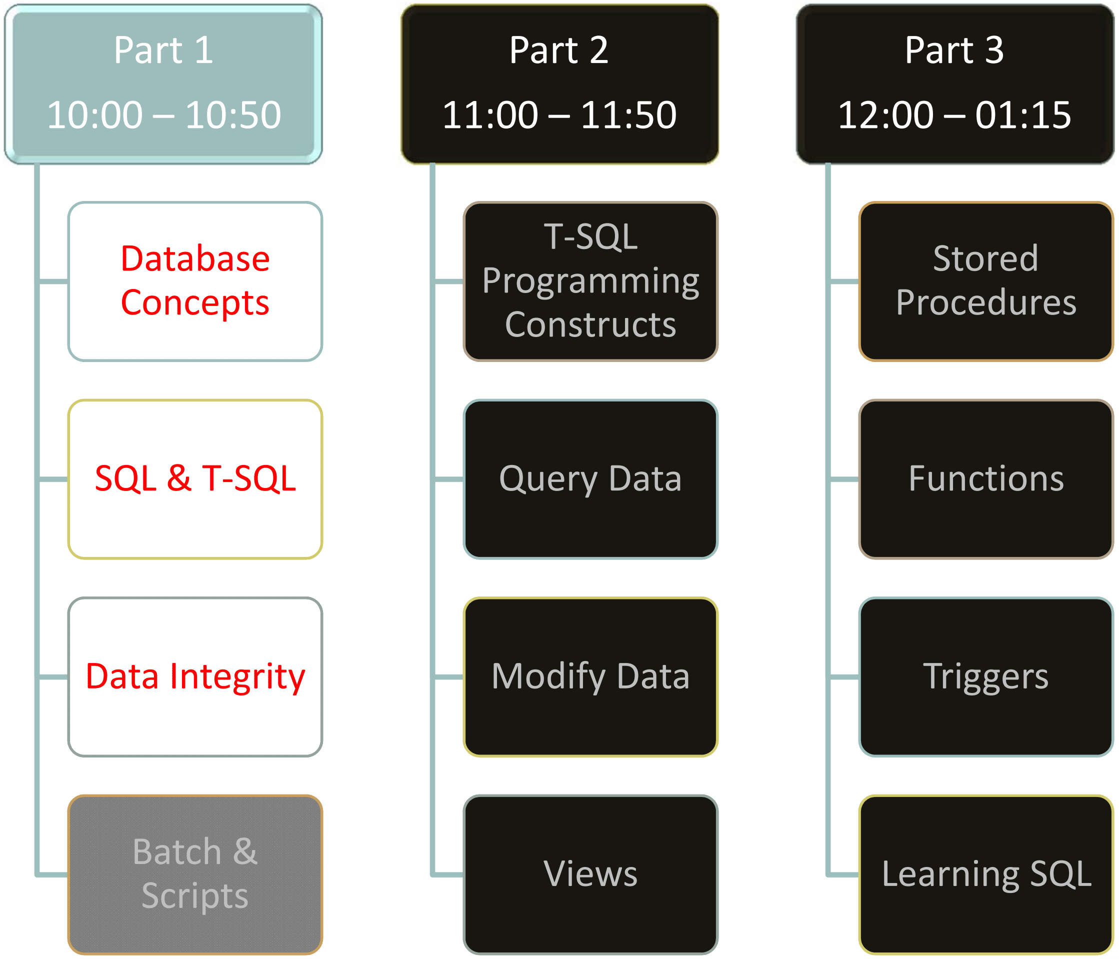
- 1970 – Dr. Codd developed Relational Data Model
- 1978 – IBM developed Structured English Query Language (SEQUEL)
- 1979 – Relational Software Inc. released the first RDBMS, Oracle
- 1985 – IBM released DB2
- 1987 – Microsoft released SQL Server
- 1989 – ANSI published the first set of standards for database query language called ANSI/ISO SQL-89
- 1992 – ANSI published revised standard ANSI/ISO SQL-92
- 1999 – ANSI/ISO SQL-99 published
- 2003 – ANSI/ISO SQL-2003 published

Introducing T-SQL

- Transact-SQL is Microsoft’s implementation of the ANSI / ISO SQL standard
- SQL Server 2005 implements ANSI-99 (Note: not fully compliant)
- Row Vs Set-based operations
- Programming Language Or Query Language
    - SQL was designed with the exclusively purpose of data retrieval and data manipulation

SQL Statement Categories

- DDL (Data Definition Language)
  - Used to create and manage the objects in a database
  - CREATE, ALTER, DROP
- DCL (Data Control Language)
  - Controls data access and ensure security
  - GRANT, DENY
- DML (Data Manipulation Language)
  - To work with data
  - INSERT, UPDATE, SELECT, DELETE



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

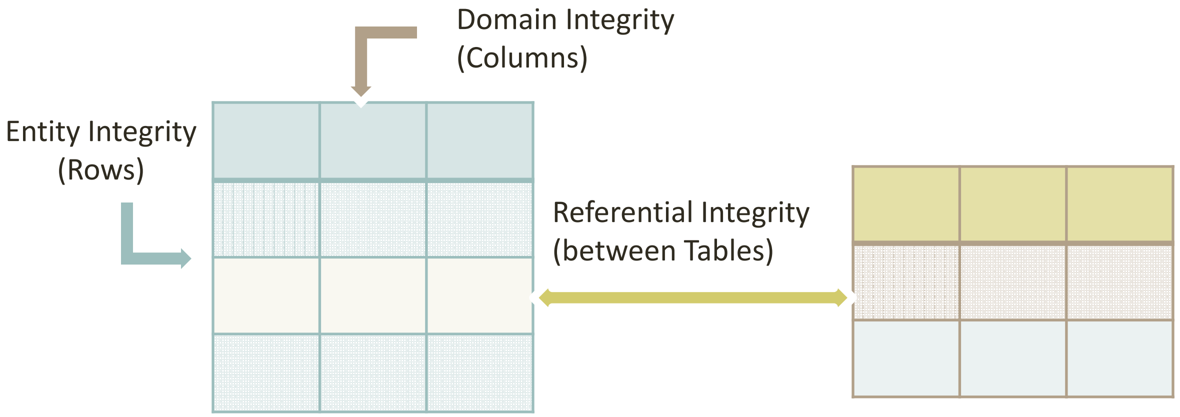
- Stored Procedures
- Functions
- Triggers
- Learning SQL

Introducing Data Integrity

- Data integrity is the ability of the database to make sure data is correct (avoid dirty data going into the system)
- Enforced using
  - Declarative Data Integrity
  - Procedural Data Integrity

- Data integrity in three forms
  - Entity integrity
  - Domain integrity
  - Referential integrity

Types of Data Integrity



- Entity Integrity
  - All rows must have a unique identifier
  - Most often enforced using Primary Key
  - IDENTITY column is used if no obvious PK exists
    - Generates a surrogate key

Types of Data Integrity

- Domain Constraint
  - Valid set of values for a column
  - Also know as column integrity
- Referential Integrity
  - Value in one column match the value in another column in either the same table or different table
  - Uses Primary & Foreign Key

IDENTITY

- Very important in database design
- Is a Property associated with a column in a table
  - SQL Server automatically assign a sequence number
  - Must be numeric
  - Only one identity column per table is allowed
  - The newly inserted id is retrieved by @@IDENTITY
- Reasons to use identity column
  - When table does not have unique identifier (natural key)
  - Whenever the unique identifier is non-numeric

Example Identity

```
IF OBJECT_ID ('Employee') IS NOT NULL
            DROP TABLE Employee
CREATE TABLE Employee
(
            EmployeeID INT IDENTITY NOT NULL, -- Cannot be null
            FirstName CHAR(10) NOT NULL,
            LastName CHAR (10) NOT NULL,
)
GO
INSERT Employee VALUES ('MOHAN', 'DAS')
SELECT @@IDENTITY

-- Following will failed
INSERT Employee VALUES ('JACK WILLIAM', 'DAVIS')
--- What will be identity value
INSERT Employee VALUES ('JACK W', 'DAVIS')
SELECT * FROM Employee
```

Understanding data types

- Type of data that can be stored in a column
- Limits the range of possible values
- Most critical decision
- Categories of data types
  - Exact numeric (int, smallint, numeric)
  - Approximate numeric (float)
  - Monetory (money, smallmoney)
  - Date & Time (datetime, smalldatetime)
  - Character (char(n), varchar(n))
  - Binary (image)
  - Specialized (bit, timestamp, table)

Constraints

- Constraints are ways to enforce Data Integrity
- Check constraints
  - Limit the range of possible values in a column with respect to business
  - Always evaluate to boolean value
  - Cannot refer to columns in another table
  - Can be created at two levels
    - Column level
    - Table level

- Rules
  - Provides the same functionality as check but not limited to a specific table or column
  - Will be removed in the future version of SQL Server

Example – Meaningful Constraint Name

```
IF OBJECT_ID ('Employee') IS NOT NULL
        DROP TABLE Employee
CREATE TABLE Employee
(
        AccountNo CHAR (10) NOT NULL
                CONSTRAINT [Employee.AccountNo must be 6 chars]
                CHECK (LEN (AccountNo) = 6),
        FirstName CHAR(10) NOT NULL,
        LastName CHAR (10) NOT NULL

        CONSTRAINT [Employee.AccountNo must be Unique]
        UNIQUE (AccountNo)
)
GO
-- error out
INSERT Employee VALUES ('A0123481', 'MOHAN', 'DAS')
-- insert a row
INSERT Employee VALUES ('A01234', 'MOHAN', 'DAS')
-- error unique constraint
INSERT Employee VALUES ('A01234', 'JACK', 'DAVIS')
```

Constraints

- Default constraints
  - Applies to column
  - Default value is used when user don’t specify a value
  - Use to avoid NULLs

- Unique constraints
  - Prohibit column(s) from allowing duplicate values
  - Can have null value

Key Constraints

- Primary Key Constraints
  - Uniquely identify a row in a table
  - Does not allow null
  - Only one PK can exist on a table
- Foreign Key Constraints
  - Implements Referential Integrity
  - Ensures that values that can be entered in a particular column exist in a specified table
  - Enforces dependency chain
- Primary Key Vs Unique Key

Example – Data Type & Constraints

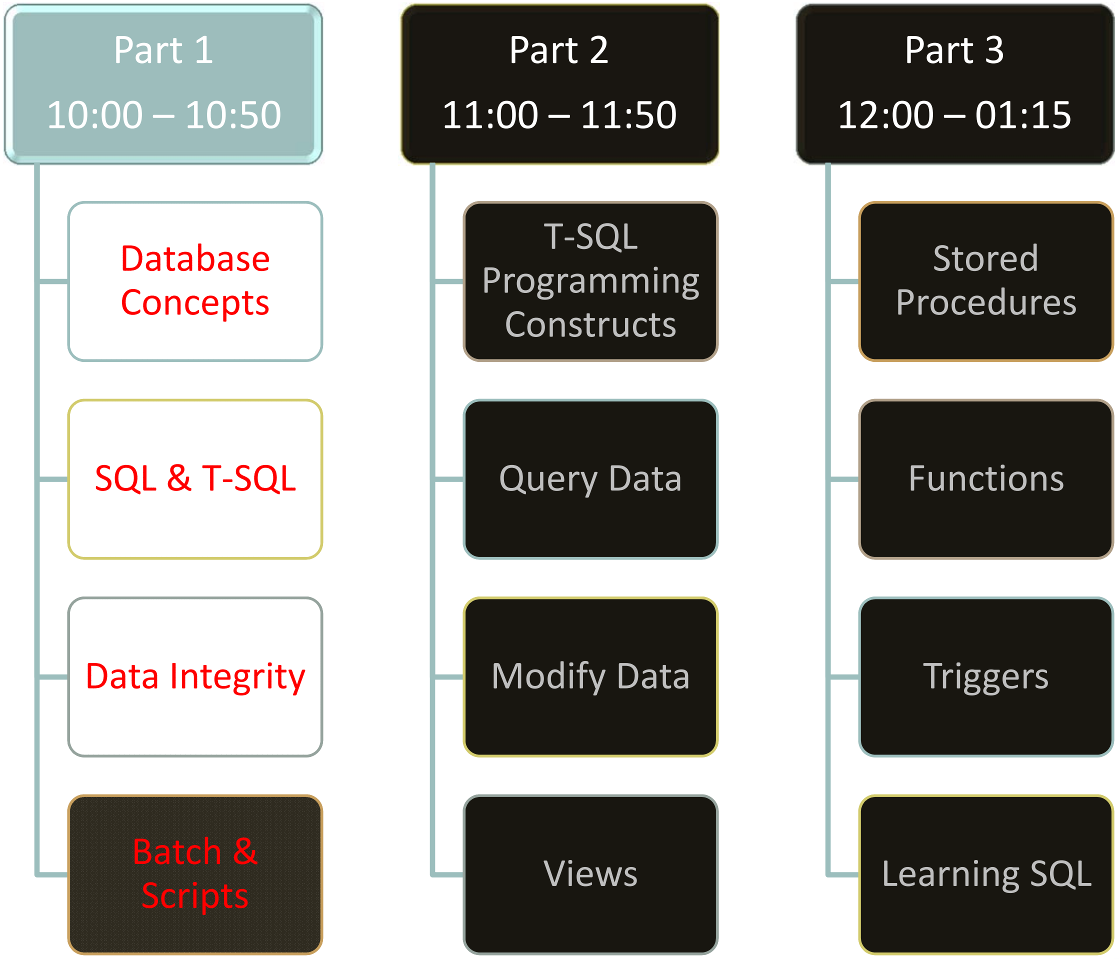
```
CREATE TABLE Customer (
            CustomerID              int             IDENTITY PRIMARY KEY,
            CustomerPANNo           varchar(20)     UNIQUE,
            CreditLimit             money           NULL CHECK (CreditLimit >= 0 and CreditLimit <= 50000),
            OutstandingBalance      int             DEFAULT 0,
            AvailableCredit         AS              (CreditLimit - OutstandingBalance),
            CreationDate            smalldatetime               NOT NULL DEFAULT getdate(),
)

CREATE TABLE Country (
            CountryID               INT             IDENTITY PRIMARY KEY,
            Country                                 VARCHAR (20)            NOT NULL
)

CREATE TABLE CustomerAddress (
            CustomerAddressID       INT             PRIMARY KEY,
            AddressLine1                            VARCHAR(20)             NOT NULL,
            AddressLine2                            VARCHAR(20)             NULL,
            CountryID                               INT
                        CONSTRAINT FK_CAToCountryID FOREIGN KEY REFERENCES Country (CountryID)
)
```

Keys - Summary

- Natural Keys
  - Not a system generated such as SSN
  - Can be unique
- Primary Keys
  - Uniquely identifies the row in a table
- Surrogate Keys
  - System generated unique key
  - Usually generated by IDENTITY
- Foreign Keys
  - Reference to PK in other table
- Index Keys
  - Indexes available on a table
- Composite Keys
  - PK is composition of more than one key
- Candidate Keys
  - Possible combination of column(s) that serves as unique identifier

Data Integrity – Guidelines

- Keys
  - Use always Primary Key
  - Prefer int over other data types
  - Use Unique Key wherever possible
- Use DEFAULT to avoid NULL values
- CHECK
  - ANSI Compliant
  - Can reference other columns
  - Cannot refer to columns in other table
  - Are fast
- Rules
  - Are independent objects
  - Slow
  - Meant for backward compatibility



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

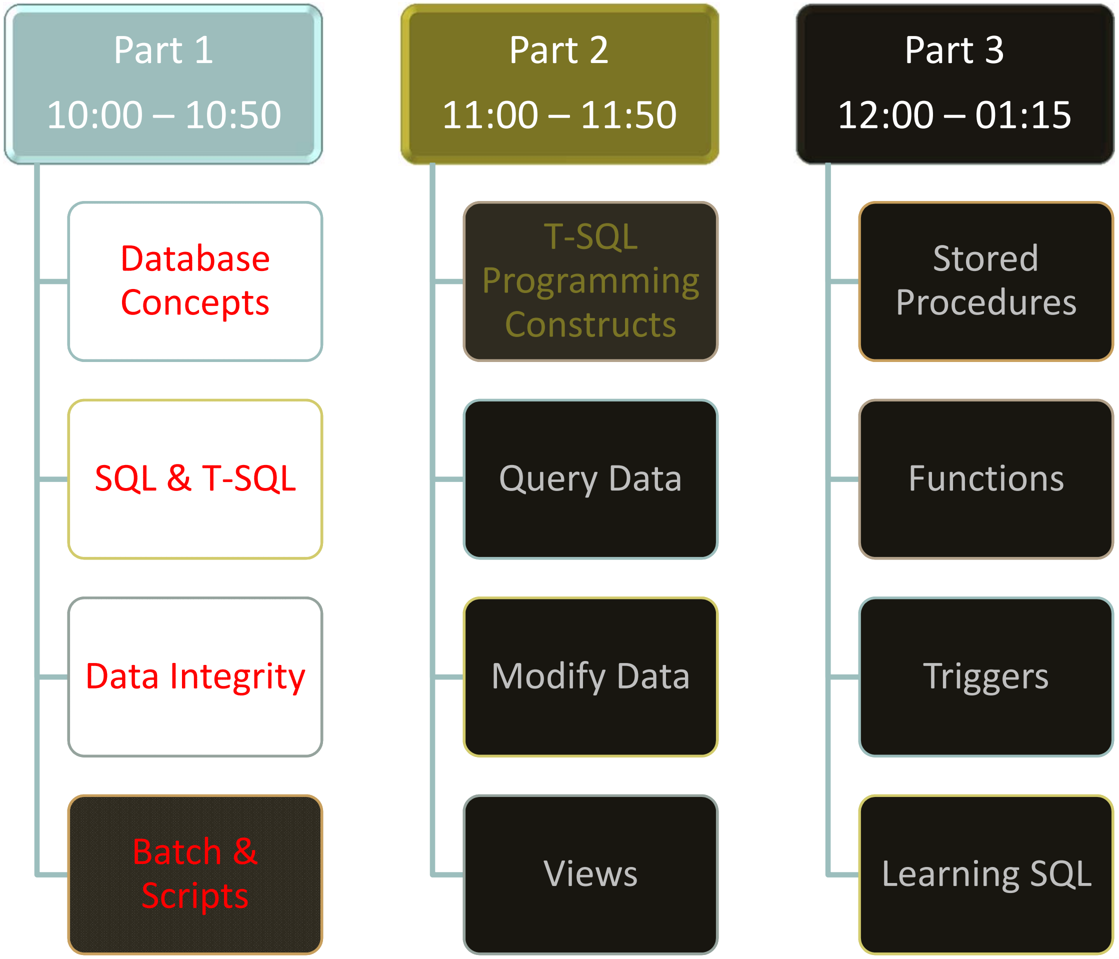
Introducing Batch & Scripts

- A batch is a group of one or more T-SQL commands sent at one time to SQL Server
- A compile error aborts the batch
- Runtime error can either stop or continue with the next statement
- The GO command
  - Is not a SQL keyword
- A Script is series of SQL statement stored in a file

Batch - Example

```
Use pubs
GO
-- compile error
SELECT * FROM sales
SELECT * FOM titleauthor
GO
-- runtime error
SELECT * FROM sales
SELECT * FROM titleautor
-- ---
SELECT * FROM titleautor
SELECT * FROM sales
```

```
Use pubs
GO
SELECT * FROM authors
/*
GO
SELECT * FROM sales
GO
SELECT * FROM publishers
GO
*/
SELECT * FROM titles
GO
```



Part 1 10:00 – 10:50	Part 2 11:00 – 11:50	Part 3 12:00 – 01:15
Database Concepts	T-SQL Programming Constructs	Stored Procedures
SQL & T-SQL	Query Data	Functions
Data Integrity	Modify Data	Triggers
Batch & Scripts	Views	Learning SQL

Minimal T-SQL Statements

- USE – change the database context
- PRINT – Returns a message to the client
- DECLARE – declare a local & table variable
- SET – Set the local variable to a value
  - SELECT is used to assign variables from table
- Comments (Line Vs Block)
- DMLs
  - SELECT
  - INSERT
  - UPDATE
  - DELETE

SS3

Slide 38

SS3

```
USE Northwind
IF EXISTS (SELECT * FROM Shippers)
BEGIN
        DECLARE @number_rows INT
        SELECT @number_rows = count(*) FROM Shippers
--      SET @number_rows = (SELECT count(*) FROM Shippers)
        PRINT 'There are '+ CAST(@number_rows AS VARCHAR(10))
        + ' rows in the Shippers table'
END
ELSE
        PRINT 'This table does not contain any rows'
GO
```

Sanjay Singh, 2/22/2007

Example – Minimal TSQL

```
USE Northwind
/* Block comments
  Checking for existing of records
*/

IF EXISTS (SELECT * FROM Shippers)
BEGIN
                -- declare local variables
                DECLARE @number_rows INT
                SELECT @number_rows = count(*) FROM Shippers
--              SET @number_rows = (SELECT count(*) FROM Shippers)
                PRINT 'There are '+ CAST(@number_rows AS VARCHAR(10))
                + ' rows in the Shippers table'
END
ELSE
                PRINT 'This table does not contain any rows'
GO
```

Controlling the Flow of Execution

- IF…ELSE – conditional & alternate execution
- BEGIN…END – statement block
- WHILE – basic looping construct
  - Can be used to process individual row
- BREAK – exits the innermost loop
- CONTINUE – restarts the loop
- GOTO – unconditionally changes the flow
  - Mostly used for error handling
- RETURN – Exits unconditionally
- EXISTS – check for existence of a row
  - Returns as soon as the first row is found

SS5

SS7

SS81

SS6

SS82

Slide 40

SS5

```
DECLARE @var1 char(10)
IF @var1 = NULL
    Print 'Is null'
ELSE
    Print 'Not null'

--- right way
DECLARE @var1 char(10)
IF @var1 IS NULL
    Print 'Is null'
ELSE
    Print 'Not null'
```

Sanjay Singh, 2/22/2007

SS6

```
PRINT 'First step'
RETURN
PRINT 'Second step (this is not executed)'
GO
```

Sanjay Singh, 3/3/2007

SS7

```
DECLARE @count int
SET @count = 0
WHILE @count < 10
BEGIN
    IF @count = 3
        BREAK
    SET @count = @count + 1
    PRINT 'This line is executed'
    CONTINUE
    PRINT 'This line is never executed'
END
GO
```

Sanjay Singh, 2/22/2007

SS81

```
IF OBJECT_ID ('dbo.TestGOTO') IS NOT NULL
    DROP PROC TestGOTO
GO
CREATE PROC TestGOTO
AS
BEGIN
    INSERT Orders (CustomerId, EmployeeId, OrderDate)
    VALUES ('ZZZZZ', 9999, 'Jan 1, 2007' )
```

Slide 40 (Continued)

```
    IF @@ERROR <> 0
        GOTO ErrorHandler

    RETURN -- unconditional exit from a stored proc

ErrorHandler:
    BEGIN
        PRINT 'Error in inserting data'
        RETURN -100 -- used to indicated a failure
    END
END
GO

-- Testing the code
EXEC TestGOTO
```

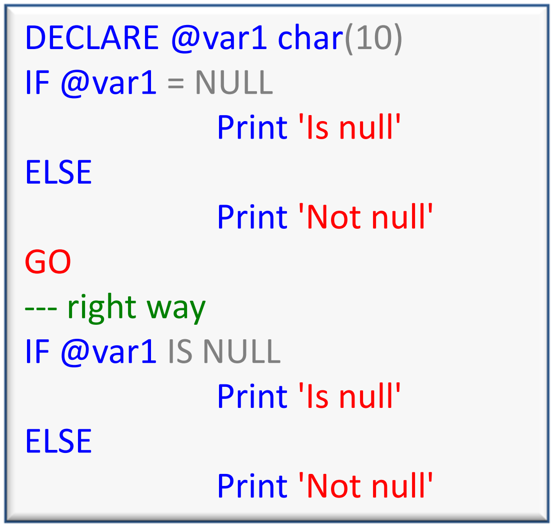
Sanjay Singh, 2/25/2007

SS82

```
USE Northwind
IF EXISTS (SELECT * FROM Shippers WHERE Phone LIKE '%555%')
BEGIN
    DECLARE @number_rows INT
    SELECT @number_rows = count(*) FROM Shippers
    PRINT 'There are '+ CAST(@number_rows AS VARCHAR(10))
            + ' rows in the Shippers table'
END
ELSE
    PRINT 'This table does not contain any rows'
```

Sanjay Singh, 2/25/2007

Importance of NULL



```
DECLARE @var1 char(10)
IF @var1 = NULL
            Print 'Is null'
ELSE
            Print 'Not null'
GO
--- right way
IF @var1 IS NULL
            Print 'Is null'
ELSE
            Print 'Not null'
```

- Indicates absence of value
  - Represents data as not applicable or not known
- Best to avoid wherever possible

CAST Vs CONVERT

- Implicit Vs Explicit conversion
- CAST function explicitly convert an expression from one data type to another
- CONVERT is same as CAST except that it provides an additional arg to specify the format
- CAST is ANSI compliant

SS8

SS9

SS10

SS11

Slide 42

SS8

```
declare @varint int
declare @varmoney money
set @varint = 100
set @varmoney = @varint
print @varint
print @varmoney
```

Sanjay Singh, 2/22/2007

SS9

```
select 70/cast(4 as float)
```

Sanjay Singh, 2/22/2007

SS10

```
-- using cast to use numeric for string comparison
USE AdventureWorks;
GO
SELECT SUBSTRING(Name, 1, 30) AS ProductName, ListPrice
FROM Production.Product
WHERE CAST(ListPrice AS int) LIKE '3%';
GO
```

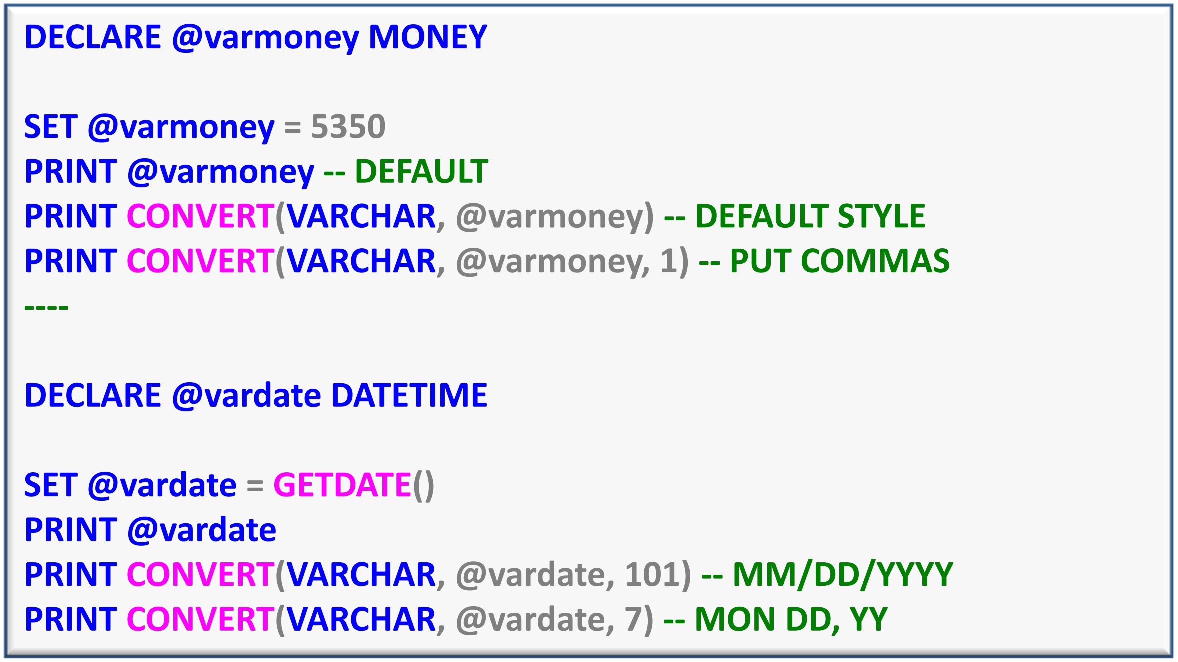
Sanjay Singh, 2/22/2007

SS11

```
declare @varmoney money
set @varmoney = 5350
print @varmoney -- default
print convert(varchar, @varmoney) -- default style
print convert(varchar, @varmoney, 1) -- put commas
----
declare @vardate datetime
set @vardate = getdate()
print @vardate
print convert(varchar, @vardate, 101) -- mm/dd/yyyy
print convert(varchar, @vardate, 7) -- Mon dd, yy
```

Sanjay Singh, 2/22/2007

Example - Datetime & Money



```
DECLARE @varmoney MONEY

SET @varmoney = 5350
PRINT @varmoney -- DEFAULT
PRINT CONVERT(VARCHAR, @varmoney) -- DEFAULT STYLE
PRINT CONVERT(VARCHAR, @varmoney, 1) -- PUT COMMAS
----

DECLARE @vardate DATETIME

SET @vardate = GETDATE()
PRINT @vardate
PRINT CONVERT(VARCHAR, @vardate, 101) -- MM/DD/YYYY
PRINT CONVERT(VARCHAR, @vardate, 7) -- MON DD, YY
```

String Functions

- LEN – returns the no of chars in the string SS12
- LTRIM, RTRIM – remove leading & trailing spaces
- LEFT, RIGHT – Returns specified no of chars SS13
- REPLACE
- CHARINDEX – returns the position of first occurrence of the find string
- LOWER, UPPER – change cases

Slide 44

SS12

```
print LEN(' Hello   ')
----- if you want to find the length including the spaces
select LEN(REPLACE (' Hello   ', ' ', '-'))
```

Sanjay Singh, 2/22/2007

SS13

```
-- creating trim function
print LTRIM(RTRIM(' Hello   '))
```

Sanjay Singh, 2/22/2007

Date / Time Functions

- GETDATE – returns the current date & time
- DAY, MONTH, YEAR – return int for day, month & year
- DATEPART – returns the part of the date specified as int
- DATENAME – return the part as string
- DATEADD, DATEDIFF – addition & subtraction operations

SS14

SS15

Slide 45

SS14

```
declare @vardate datetime
set @vardate = getdate()
select @vardate
select datepart (year, @vardate)
select datepart (month, @vardate)
select datepart (day, @vardate)
select datepart (hour, @vardate)

select datename (year, @vardate)
select datename (month, @vardate)
select datename (day, @vardate)
select datename (weekday, @vardate)
```

Sanjay Singh, 2/22/2007

SS15

```
declare @vardate datetime
set @vardate = getdate()
select datediff (year, '2001-10-30', @vardate)
select datediff (month, '2001-10-30', @vardate)
```

Sanjay Singh, 2/22/2007

Example – Date / Time Functions

```
DECLARE @vardate DATETIME
SET @vardate = GETDATE()
SELECT @vardate
SELECT DATEPART (YEAR, @vardate)
SELECT DATEPART (MONTH, @vardate)
SELECT DATEPART (DAY, @vardate)
SELECT DATEPART (HOUR, @vardate)

SELECT DATENAME (YEAR, @vardate)
SELECT DATENAME (MONTH, @vardate)
SELECT DATENAME (DAY, @vardate)
SELECT DATENAME (WEEKDAY, @vardate)
```

```
DECLARE @vardate DATETIME
SET @vardate = GETDATE()
SELECT DATEDIFF (YEAR, '2001-10-30', @vardate)
SELECT DATEDIFF (MONTH, '2001-10-30', @vardate)
```

Aggregate Function

- Returns a scalar value after applying on range of data
- Impact of NULL columns
  - Aggregate functions does not consider NULL when returning results
- Examples
  - AVG()
  - COUNT()
  - MIN(), MAX()
  - SUM()

SS98

SS99

Slide 47

SS98

```
USE pubs
GO
SELECT AVG(discount) FROM discounts
SELECT AVG(highqty) FROM discounts
```

Sanjay Singh, 2/21/2007

SS99

```
use pubs
GO
select count(discount) from discounts
select count(highqty) from discounts
```

Sanjay Singh, 2/21/2007

CASE & ISNULL

- CASE evaluates list of conditions and returns one of multiple possible result expressions
- Simple CASE function
  - Test the expression in the CASE clause against the expression in the WHEN clause

SS17

- Searched CASE function
  - Test the conditional expression in each WHEN clause

SS16

- ISNULL – replaces a NULL with expression with specified value

SS18

Slide 48

SS16

```
USE AdventureWorks;
GO
DECLARE @varint int
SET @varint = 50
SELECT   ProductNumber, Name, 'Price Range' =
    CASE
      WHEN ListPrice =  0 AND @varint IS NULL THEN 'Mfg item - not for resale'
      WHEN ListPrice < 50 THEN 'Under $50'
      WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'
      WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under $1000'
      ELSE 'Over $1000'
    END
FROM Production.Product
ORDER BY ProductNumber ;
GO
```

Sanjay Singh, 2/22/2007

SS17

```
USE AdventureWorks;
GO
SELECT   ProductNumber, Category =
    CASE ProductLine
      WHEN 'R' THEN 'Road'
      WHEN 'M' THEN 'Mountain'
      WHEN 'T' THEN 'Touring'
      WHEN 'S' THEN 'Other sale items'
      ELSE 'Not for sale'
    END,
  Name
FROM Production.Product
ORDER BY ProductNumber;
GO
```

Sanjay Singh, 2/22/2007

SS18

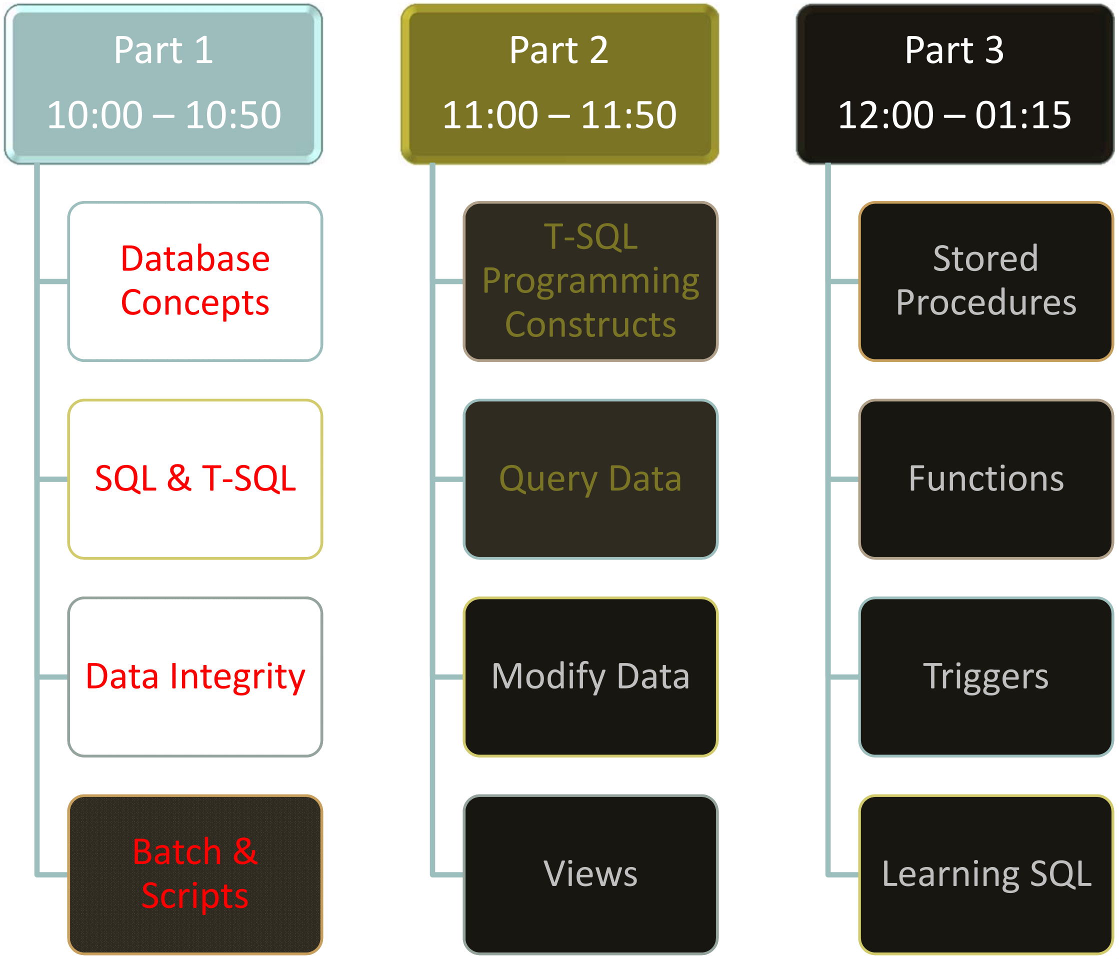
```
USE AdventureWorks;
GO
SELECT Description, DiscountPct, MinQty, ISNULL(MaxQty, 0.00) AS 'Max Quantity'
FROM Sales.SpecialOffer;
GO
```

Sanjay Singh, 2/22/2007

Example – CASE

```
USE AdventureWorks;
GO
SELECT   ProductNumber, Category =
      CASE ProductLine
         WHEN 'R' THEN 'Road'
         WHEN 'M' THEN 'Mountain'
         WHEN 'T' THEN 'Touring'
         WHEN 'S' THEN 'Other sale items'
         ELSE 'Not for sale'
      END,
   Name
FROM Production.Product
ORDER BY ProductNumber;
GO
```

```
USE AdventureWorks;
GO
DECLARE @varint int
SET @varint = 50
SELECT   ProductNumber, Name, 'Price Range' =
      CASE
         WHEN ListPrice =  0 AND @varint IS NULL
                                    THEN 'Mfg item - not for resale'
         WHEN ListPrice < 50 THEN 'Under $50'
         WHEN ListPrice >= 50 and ListPrice < 250
                                    THEN 'Under $250'
         WHEN ListPrice >= 250 and ListPrice < 1000
                                    THEN 'Under $1000'
         ELSE 'Over $1000'
      END
FROM Production.Product
ORDER BY ProductNumber ;
```



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

Introducing SELECT

- The most used statement in SQL
- Used to query data from tables and views
- Can do multiple things
  - Assign variables
  - Return rows
  - Create tables

```
SELECT [DISTINCT][TOP n]
          <columns to be chosen, optionally eliminating
                    duplicate rows from result set or limiting
                    number of rows to be returned>
[FROM] <table names>
[WHERE] <criteria that must be true for a row to be chosen>
[GROUP BY] <columns for grouping aggregate functions>
[HAVING] <criteria that must be met for aggregate functions>
[ORDER BY] <optional specification of how the results should be sorted>
```

Example – Filter Using Operators

```sql
USE Northwind
-- Returns all employees whose last name begins with 'b'
SELECT lastname, firstname FROM Employees
WHERE lastname LIKE 'b%'

-- Returns all employees who don't live in Seattle, Redmond or Tacoma
SELECT lastname, firstname, city FROM Employees
WHERE city NOT IN ('seattle','redmond','tacoma')

-- Returns all employees that were hired between 1/1/1993 and 12/31/1993
SELECT lastname, firstname, hiredate FROM Employees
WHERE hiredate BETWEEN '1993.1.1'AND '1993.12.31'

-- Returns all employees that live in any other city than London
-- and first name not starts with N
SELECT lastname, firstname, city FROM Employees
WHERE city <> 'london'
AND firstname NOT LIKE 'N%'
```

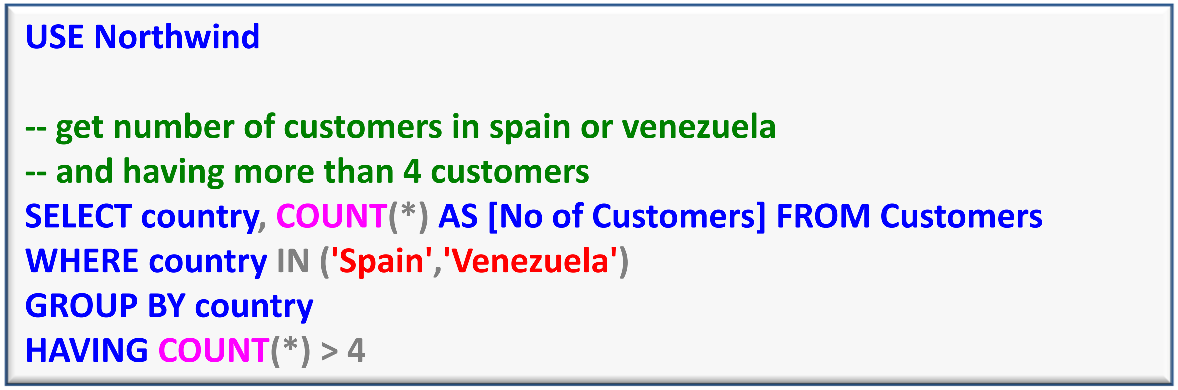
Example – Importance of NULL

- Use IS NULL or IS NOT NULL

```
USE Northwind
-- Retrieves all suppliers whose region is NULL (or unknown)
SELECT companyname, contactname, region FROM Suppliers
WHERE region = NULL

SELECT companyname, contactname, region FROM Suppliers
WHERE region IS NULL
```

Example – Using GROUP BY

- Groups the rows of a result set based on one or more columns
- HAVING clause specifies a search condition for a group
- Order of evaluation the query - WHERE, GROUP BY, HAVING



```
USE Northwind

-- get number of customers in spain or venezuela
-- and having more than 4 customers
SELECT country, COUNT(*) AS [No of Customers] FROM Customers
WHERE country IN ('Spain','Venezuela')
GROUP BY country
HAVING COUNT(*) > 4
```

Introducing Joins

- In real world database consists of multiple tables
- Used to retrieve data from two or more tables
- Usually constructed between one PK of table with FK of another table
- Four types of join
  - INNER JOIN
  - OUTER JOIN (LEFT, RIGHT, FULL)
  - CROSS JOIN
  - FULL JOIN

Table and Column aliases

- Table alias
  - Usually used when working with multiple tables
  - Once an alias is specified it must be used in the rest of the query
- Column alias
  - Used to give meaningful name to the column
  - In the query original column must be used

```
Use pubs

-- will error as authors is aliased
SELECT authors.au_lname AS [Last Name], t.title Book_Title
FROM authors a
JOIN titleauthor ta
            ON a.au_id = ta.au_id
JOIN titles t
            ON t.title_id = ta.title_id
```

Sub Queries

- A subquery is a query nested inside another DML statement
- Conceptually SQL Server runs the inner query and then the outer query
- Correlated query is different from a subquery in the sense subquery can be executed separately
- Derived table is a subquery in the FROM clause

SS84

Slide 57

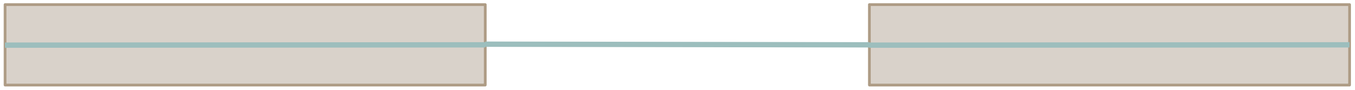
SS84

```
Use Northwind
-- using subquery
SELECT OrderID, Orders.CustomerID, EmployeeID, OrderDate
FROM Orders WHERE CustomerID IN (
        SELECT CustomerID FROM Customers WHERE City = 'London'
)
ORDER BY OrderID

-- using join
SELECT OrderID, Orders.CustomerID, EmployeeID, OrderDate
FROM Orders
JOIN Customers
        ON Orders.CustomerID = Customers.CustomerID
WHERE Customers.City = 'London'
ORDER BY OrderID
```

Sanjay Singh, 2/25/2007

Understanding Join

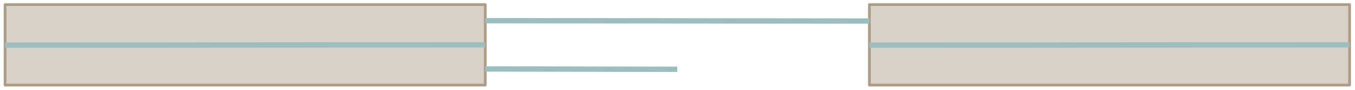
Table A INNER JOIN Table B



SS19

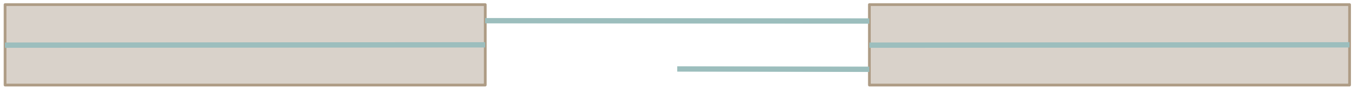
SS20

Table A LEFT OUTER JOIN Table B



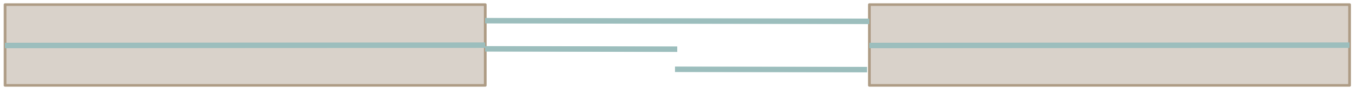
SS21

Table A RIGHT OUTER JOIN Table B



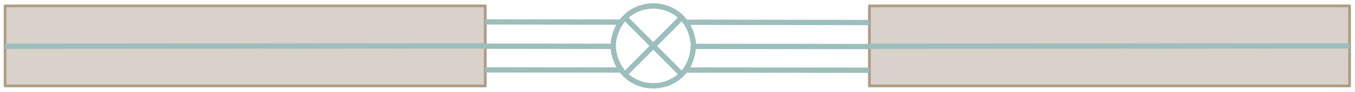
SS22

Table A FULL OUTER JOIN Table B



SS23

Table A CROSS JOIN Table B



SS24

Slide 58

SS19

```
USE Northwind
GO
SELECT * FROM Products
INNER JOIN Suppliers
        ON Products.SupplierID = Suppliers.SupplierID
```

Sanjay Singh, 2/21/2007

SS20

```
-- ambiguous & usage of alias column
SELECT Products.*, SupplierID
FROM Products p
INNER JOIN Suppliers
        ON Products.SupplierID = Suppliers.SupplierID
-------------------
Use Northwind
GO
SELECT Products.*, SupplierID
FROM Products
INNER JOIN Suppliers
        ON Products.SupplierID = Suppliers.SupplierID
```

Sanjay Singh, 2/21/2007

SS21

```
USE pubs
GO
SELECT discounttype, discount, s.stor_name
FROM discounts d
LEFT OUTER JOIN stores s
        ON d.stor_id = s.stor_id
```

Sanjay Singh, 2/21/2007

SS22

```
USE pubs
GO
select * from stores
GO
SELECT discounttype, discount, s.stor_name
FROM discounts d
RIGHT OUTER JOIN stores s
        ON d.stor_id = s.stor_id
```

Sanjay Singh, 2/21/2007

SS23

```
USE pubs
GO
SELECT discounttype, discount, s.stor_name
```

Slide 58 (Continued)

```
FROM discounts d
FULL OUTER JOIN stores s
        ON d.stor_id = s.stor_id
```

Sanjay Singh, 2/21/2007

SS24

```
USE pubs
GO
SELECT discounttype, discount, s.stor_name
FROM discounts d
CROSS JOIN stores s
        ON d.stor_id = s.stor_id -- will give an error
```

Sanjay Singh, 2/21/2007

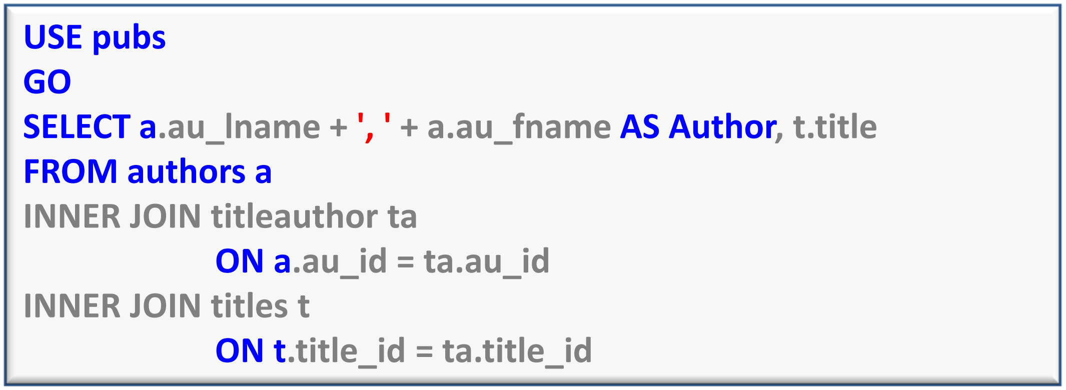
Inner Join

- Return common rows based on the join condition that matches in both tables
- Also known as equi-join
- Can also join values in two columns that are not equal
- INNER keyword is optional
  - However it is better to use it

Example – Inner Join

```
-- ambiguous & usage of alias column
Use Northwind
GO
SELECT Products.*, SupplierID
FROM Products p
INNER JOIN Suppliers
                ON Products.SupplierID = Suppliers.SupplierID
-------------------

SELECT Products.*, SupplierID
FROM Products
INNER JOIN Suppliers
                ON Products.SupplierID = Suppliers.SupplierID
```

Example – Inner Join 3 tables



```
USE pubs
GO
SELECT a.au_lname + ', ' + a.au_fname AS Author, t.title
FROM authors a
INNER JOIN titleauthor ta
                    ON a.au_id = ta.au_id
INNER JOIN titles t
                    ON t.title_id = ta.title_id
```

Outer Join

- Deciding whether to use outer join or not?
  - Outer join returns matching AND non-matching rows
  - The non-matching rows denotes the need for a outer join

```
use pubs

-- left outer join
SELECT discounttype, discount, s.stor_name
FROM discounts d LEFT OUTER JOIN stores s
            ON d.stor_id = s.stor_id

--- right outer join
SELECT discounttype, discount, s.stor_name
FROM discounts d RIGHT OUTER JOIN stores s
            ON d.stor_id = s.stor_id
```

Cross Join

- Cross Join
  - There is no ON operator
  - Joins every record in one table to every record in other table
  - Cartesian product of all the records
  - Mainly used to create test data

```
USE Northwind

-- cross join
SELECT e.FirstName, c.CompanyName
FROM Employees e
CROSS JOIN Customers c
```

Example – A Practical Scenario

ID	Fname	Lname	Country
1	Ram	Mohan	India
2	Jasbir	Singh	India
3	Mac	George	US

ID	DateType	DateValue
1	BD	1955-10-20
1	JD	1980-04-10
1	AD	1975-12-15

First Name	Last Name	Birth Date	Anniversary
Ram	Mohan	Oct 20 1955	Dec 15 1975

```
SELECT PT.FName, PT.LName,
        CAST (DPTBD.Date AS VARCHAR(11)) AS [Birth Date],
        CAST (DPTAD.Date AS VARCHAR(11)) AS [Anniversary]
FROM ProfileTbl PT
JOIN DateProfileTbl DPTBD
        ON PT.ProfileID = DPTBD.ProfileID AND DPTBD.DateType = 'BD'
JOIN DateProfileTbl DPTAD
        ON PT.ProfileID = DPTAD.ProfileID AND DPTAD.DateType = 'AD'
```

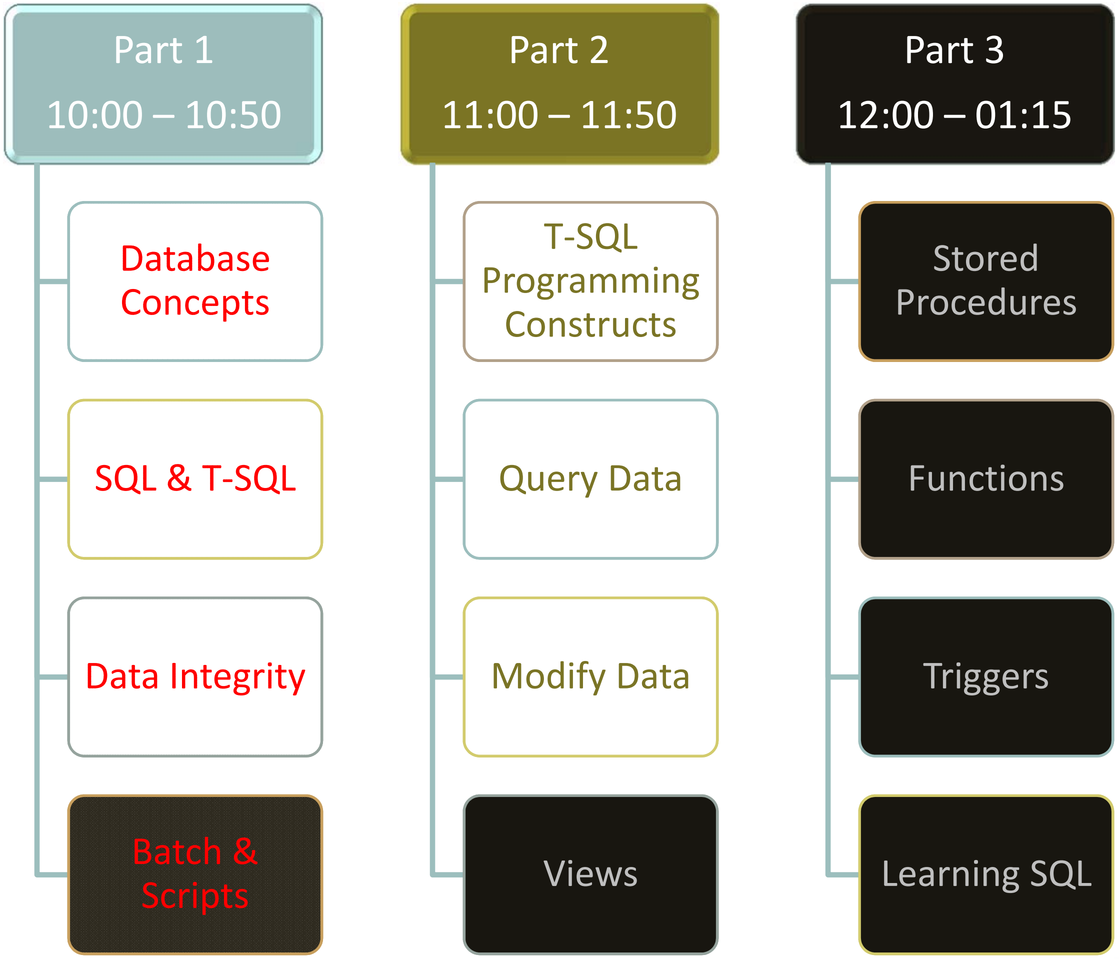
UNION

- Combines two or more result sets into a single result set
- All queries must have the same no of columns
- The headings returned is of the first query
- Datatypes of each column must be compatible
- Default return option is DISTINCT

```
USE Northwind
GO
SELECT CompanyName AS Customer, Country FROM Customers WHERE Country = 'USA'
UNION
SELECT CompanyName AS Supplier, Country FROM Suppliers WHERE Country = 'USA'
```

SELECT – Guidelines

- Use column names instead of *
- Use table and column aliases
- Take care when dealing with columns having NULL values
- Use positive conditions such as IN instead of NOT IN
- Avoid using leading wildcards
- Prefer WHERE over HAVING clause wherever possible
- Use JOINs on columns having indexes
- Use OUTER JOINs only when required



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

Three Ds

- DML (Data Manipulation Language)
  - INSERT
  - UPDATE
  - DELETE
  - SELECT
- DDL (Data Definition Language)
- DCL (Data Control Language)

- DMS (Data Modification Statements)
  - INSERT
  - UPDATE
  - DELETE

Basic INSERT

```
INSERT INTO <table_name>
        (column_list,...)
    VALUES
        (value_list,..._
```

- INTO is optional but mandatory by ANSI
- In case of inserting data for all rows column names can be omitted
- Order of the value must match the column list
- For default values, the column name can be omitted

Example – INSERT

```
CREATE TABLE Employeetbl (
                emp_id INT PRIMARY KEY,
                first_name CHAR(10) NOT NULL,
                last_name CHAR(10),
                age INT NOT NULL CHECK(age > 10),
                phone_no VARCHAR(20) DEFAULT 'UNKNOWN'
)

INSERT INTO Employeetbl
                VALUES (1, 'MAC', 'MOHAN', 30, '91-6447849-949')

-- is clear and explicit
INSERT INTO Employeetbl (EMP_ID, first_name, age)
VALUES (2, 'JOHN', 42)
-- using default keyword
INSERT INTO Employeetbl (EMP_ID, first_name, last_name, age)
VALUES (3, 'SAM', DEFAULT, 24)
```

Multiple Row Insert

- INSERT…SELECT
  - Table must already exist
  - The operation is atomic
  - Handy to create test data
- INSERT…EXEC
  - Similar to insert…select except that select is replaced with a execute proc
  - Result set of exec must match the table
  - Useful when getting data from another sp
    - Promotes reusability
- SELECT…INTO
  - Directly builds the table
  - Can help in creating a table with same definition

Example – Multiple Row Insert

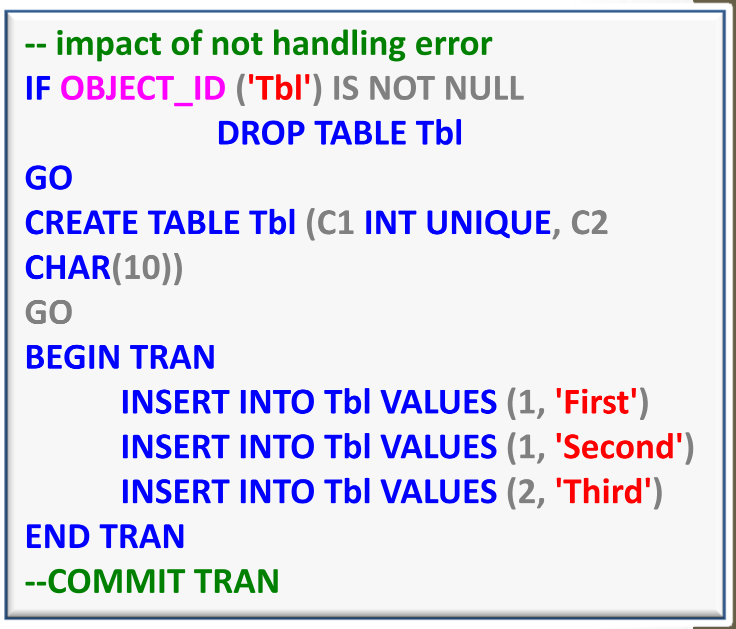
```
-- using INSERT…SELECT
CREATE TABLE demo (col1 int, col2
varchar(10))
GO

INSERT INTO demo SELECT 1, 'hello'
INSERT INTO demo SELECT 10, 'world'
```

```
-- creating an empty table
Use Pubs
GO

SELECT *
INTO NewSales
FROM Sales WHERE 1 = 2
```

```
-- using  SELECT…INTO
SELECT IDENTITY(int, 1, 1) AS OrderID,
*
INTO NewSales
FROM Sales
GO
```

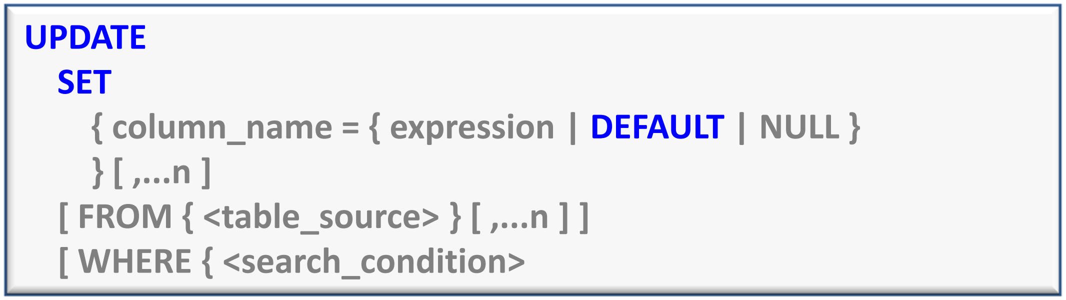
INSERT – Guidelines

- Always use the column list
- Follow ANSI standard
  - Use INTO clause
- Handle error after every insert
  - Even if it is in a transaction



```
-- impact of not handling error
IF OBJECT_ID ('Tbl') IS NOT NULL
                DROP TABLE Tbl
GO
CREATE TABLE Tbl (C1 INT UNIQUE, C2
CHAR(10))
GO
BEGIN TRAN
        INSERT INTO Tbl VALUES (1, 'First')
        INSERT INTO Tbl VALUES (1, 'Second')
        INSERT INTO Tbl VALUES (2, 'Third')
END TRAN
--COMMIT TRAN
```

UPDATE Statement



```
UPDATE
   SET
      { column_name = { expression | DEFAULT | NULL }
      } [ ,...n ]
   [ FROM { <table_source> } [ ,...n ] ]
   [ WHERE { <search_condition>
```

- Changes existing data in a table
- Use a FROM clause to update data based on condition from multiple tables
- If no WHERE clause provided all rows are updated

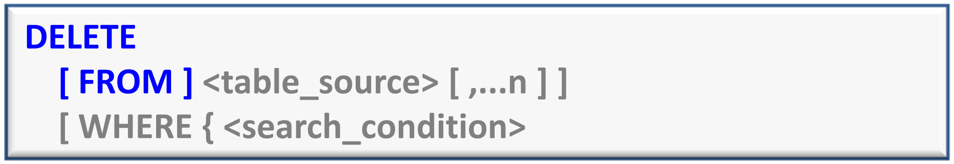
Example – UPDATE All Vs Selected Rows

```
USE pubs
GO
-- Raise the price of every title by 12%
-- No WHERE clause, so it affects every row in the table
UPDATE titles
   SET price=price * 1.12
GO

-- Change a specific employee's last name after his marriage
UPDATE employee
   SET lname='David-Mohan'
WHERE emp_id='GHT50241M'
```

Example – UPDATE Using Join

```
--Table to hold new last names
CREATE TABLE dbo.tnew_zip
(
                  au_id VARCHAR(40)
                  ,zipcode CHAR(5) NULL
)

--Working dbo.authors table
SELECT *  INTO dbo.authors_new FROM dbo.authors

--Insert dbo.authors ID numbers
-- check the full code

--Update original dbo.authors table with new zip
UPDATE a
SET zip = n.zipcode
FROM dbo.authors_new a
INNER JOIN dbo.tnew_zip n
                  ON a.au_id = n.au_id
WHERE n.zipcode IS NOT NULL
```

DELETE Statement



```
DELETE
   [ FROM ] <table_source> [ ,...n ] ]
   [ WHERE { <search_condition>
```

- Be careful
  - DELETE <table> is dangerous
  - Permanently removes rows from the table
- Only table name is specified (no columns)
- Join can be used to delete table based on conditions from multiple tables
- TRUNCATE is another way to clean up rows

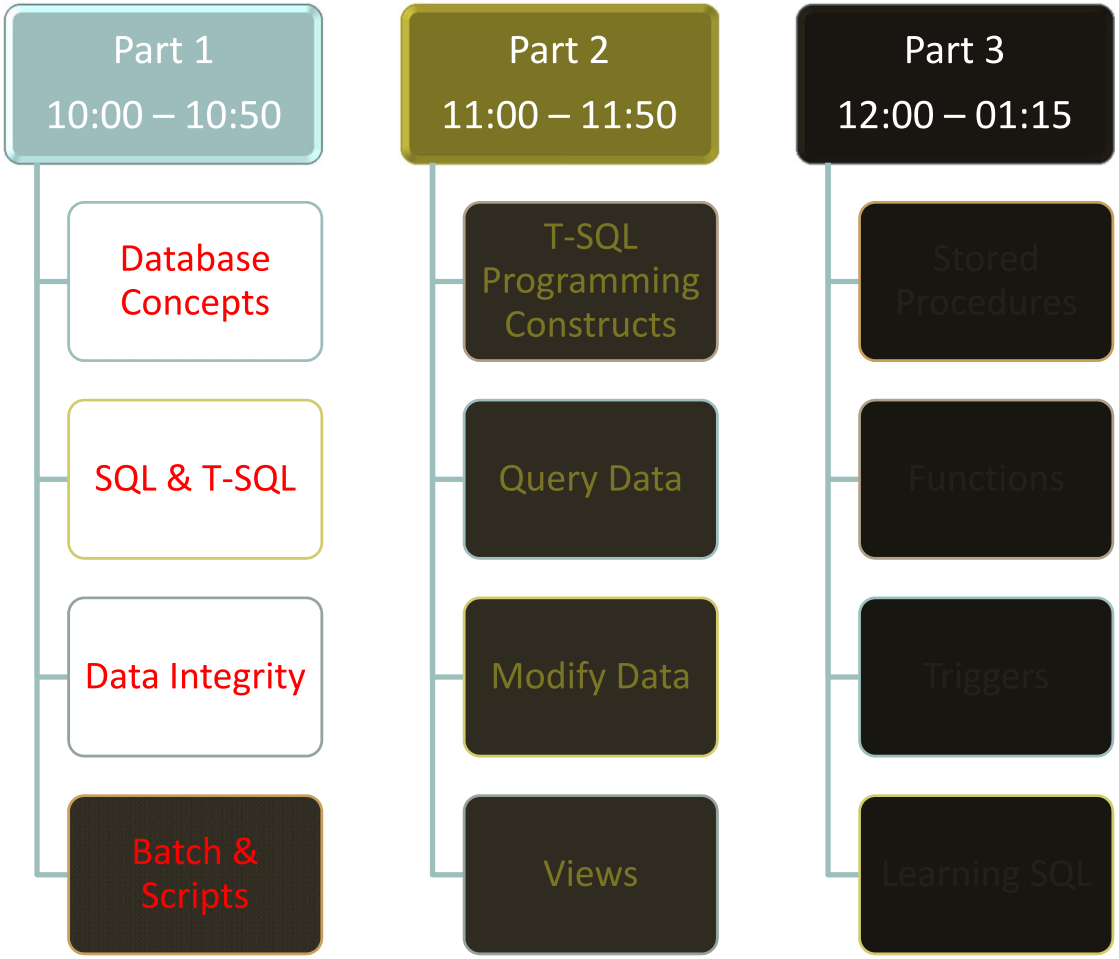
Example – DELETE

```
USE TestDb

IF OBJECT_ID ('products') IS NOT NULL
                DROP TABLE products

SELECT  *  INTO products FROM
northwind.dbo.products

--- simple delete
SELECT * FROM Products WHERE SupplierID = 1
-- be careful
DELETE products
WHERE SupplierID = 1
```

```
---- incorrect syntax
DELETE
FROM  products p
INNER JOIN categories c
   ON p.categoryid =  c.categoryid
WHERE  c.description  LIKE  '%fish%'

--- correct syntax
DELETE p
FROM  products p
INNER JOIN categories c
   ON p.categoryid =  c.categoryid
WHERE  c.description  LIKE  '%fish%'
```

DELETE – Guidelines

- Always use BEGIN…TRAN when testing delete operation on a server
- Never…Never use DELETE <table_name> on a single line
  - Use filters such as a WHERE clause
  - Use alias with FROM on a separate line
- Check for error after every DELETE statement
- Check for rows affected after DELETE statement



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

Introducing View

- Is a virtual table whose contents are defined by a query (also termed as stored query)
- The data is produced dynamically when the view is referenced
- Tables referenced in views are called base tables
- Types of Views
  - Standard Views
  - Indexed Views
  - Partitioned Views

SS39

SS40

SS39

```
Use pubs
GO
IF OBJECT_ID ('Customer') IS NOT NULL
	DROP TABLE Customer
GO
CREATE TABLE Customer (
	CustID int IDENTITY (1, 1) NOT NULL,
	CustName varchar (10)
)
GO
INSERT Customer VALUES ('Dell')
INSERT Customer VALUES ('IBM')
INSERT Customer VALUES ('Toshiba')
GO
IF OBJECT_ID ('TestView') IS NOT NULL
	DROP VIEW TestView
GO
CREATE VIEW TestView
AS
SELECT * FROM Customer
GO
SP_RENAME 'Customer.CustID', 'ClientID', 'COLUMN';
GO
SELECT * FROM TestView
SELECT ClientID FROM TestView
```

Sanjay Singh, 2/21/2007

SS40

```
Use pubs
GO
IF OBJECT_ID ('Customer') IS NOT NULL
	DROP TABLE Customer
GO
CREATE TABLE Customer (
	CustID int IDENTITY (1, 1) NOT NULL,
	CustName varchar (10)
)
GO
INSERT Customer VALUES ('Dell')
INSERT Customer VALUES ('IBM')
INSERT Customer VALUES ('Toshiba')
```

Slide 81 (Continued)

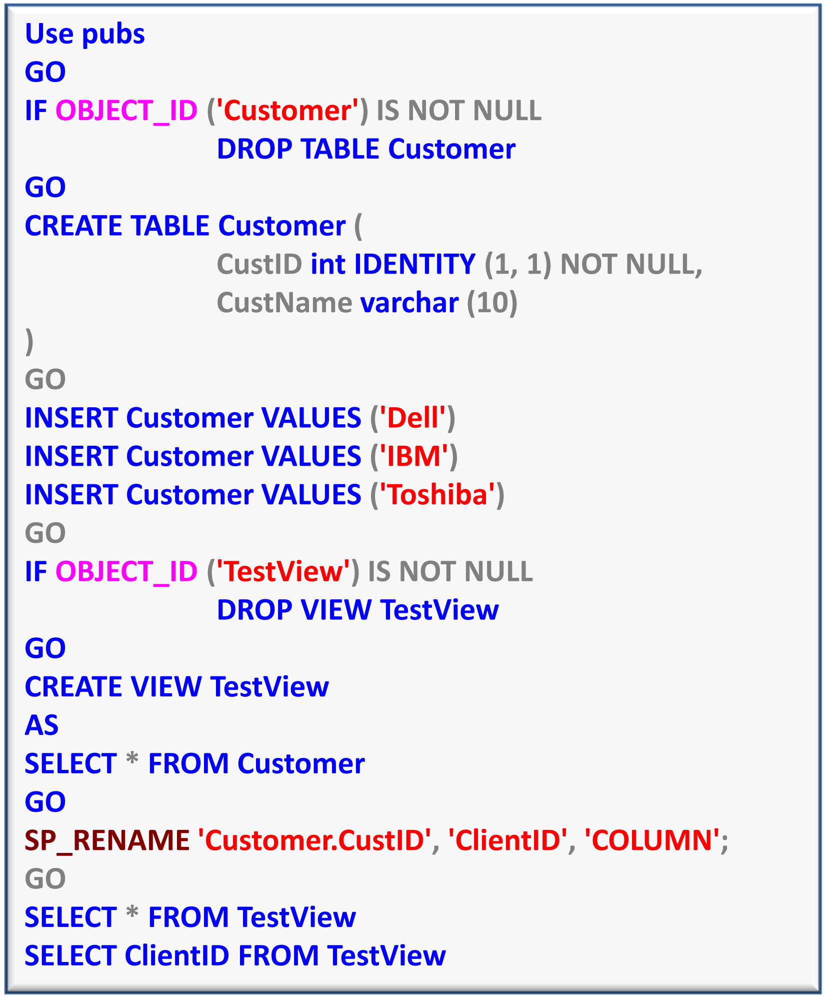
```
GO
IF OBJECT_ID ('TestView') IS NOT NULL
        DROP VIEW TestView
GO
CREATE VIEW TestView
AS
SELECT CustID, CustName FROM Customer
GO
SP_RENAME 'Customer.CustID', 'ClientID', 'COLUMN';
GO
SELECT * FROM TestView -- Select * from (select custid, custname from customer)
SELECT ClientID FROM TestView

SELECT CustID FROM TestView
```

Sanjay Singh, 2/21/2007

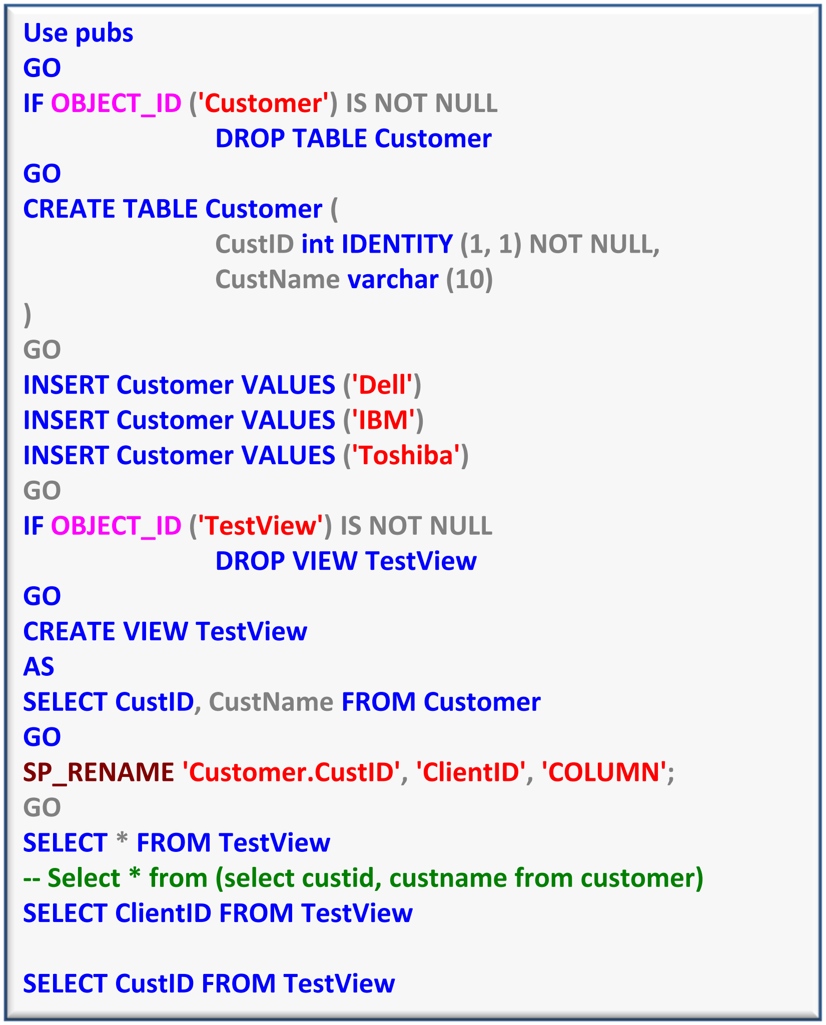
Benefits

- Design independence for application using the view
- Data security (provide access only to specific data in the view)
- Flexibility (custom views for different needs)
- Simplified queries (hide the complexity)
- Updatability (base table can be changed with certain restrictions)

Example – SELECT All Vs SELECT Columns



```sql
Use pubs
GO
IF OBJECT_ID ('Customer') IS NOT NULL
                DROP TABLE Customer
GO
CREATE TABLE Customer (
                CustID int IDENTITY (1, 1) NOT NULL,
                CustName varchar (10)
)
GO
INSERT Customer VALUES ('Dell')
INSERT Customer VALUES ('IBM')
INSERT Customer VALUES ('Toshiba')
GO
IF OBJECT_ID ('TestView') IS NOT NULL
                DROP VIEW TestView
GO
CREATE VIEW TestView
AS
SELECT * FROM Customer
GO
SP_RENAME 'Customer.CustID', 'ClientID', 'COLUMN';
GO
SELECT * FROM TestView
SELECT ClientID FROM TestView
```



```sql
Use pubs
GO
IF OBJECT_ID ('Customer') IS NOT NULL
                DROP TABLE Customer
GO
CREATE TABLE Customer (
                CustID int IDENTITY (1, 1) NOT NULL,
                CustName varchar (10)
)
GO
INSERT Customer VALUES ('Dell')
INSERT Customer VALUES ('IBM')
INSERT Customer VALUES ('Toshiba')
GO
IF OBJECT_ID ('TestView') IS NOT NULL
                DROP VIEW TestView
GO
CREATE VIEW TestView
AS
SELECT CustID, CustName FROM Customer
GO
SP_RENAME 'Customer.CustID', 'ClientID', 'COLUMN';
GO
SELECT * FROM TestView
-- Select * from (select custid, custname from customer)
SELECT ClientID FROM TestView

SELECT CustID FROM TestView
```

Creating View

```
CREATE VIEW view_name
AS
select_statement
```

- Cannot reference temporary tables.
- Can include ORDER BY only if TOP is used

```
CREATE VIEW CustomerOrders_vw
AS
SELECT cu.CompanyName,
          o.OrderID,
          o.OrderDate,
          od.ProductID,
          p.ProductName,
          od.Quantity,
          od.UnitPrice,
          od.Quantity * od.UnitPrice AS ExtendedPrice
FROM Customers AS cu
INNER JOIN Orders AS o
          ON cu.CustomerID = o.CustomerID
INNER JOIN [Order Details] AS od
          ON o.OrderID = od.OrderID
INNER JOIN Products AS p
          ON od.ProductID = p.ProductID
```

```
--- for managers
SELECT * FROM CustomerOrders_vw
```

Restrictions When Creating Views

- ORDER BY cannot be used
  - Unless TOP is specified
- A view cannot refer to a temporary table
- SELECT * can be used in the view as long as SCHEMABINDING is not specified
- COMPUTE clause cannot be used

Using SCHEMABINDING

- Creates a association between the view and the objects it refers to
- Prevents the view from becoming orphan
- Column names must be provided and not *

```
Use pubs
CREATE TABLE Customer (
                CustID int IDENTITY (1, 1) NOT NULL,
                CustName varchar (10)
)
GO
INSERT Customer VALUES ('Dell')
GO
CREATE VIEW TestView
WITH SCHEMABINDING
AS
SELECT CustID, CustName
FROM Customer --must be two part name
dbo.Customer
GO
DROP TABLE Customer
```

Using View to Change Data

- If a view contain join then INSERT, DELETE, UPDATE cannot be used unless INSTEAD OF trigger is used except in some cases
- Required field must appear in the view or have default value
- WITH CHECK OPTION – the resulting row must qualify to appear in the view results

SS42

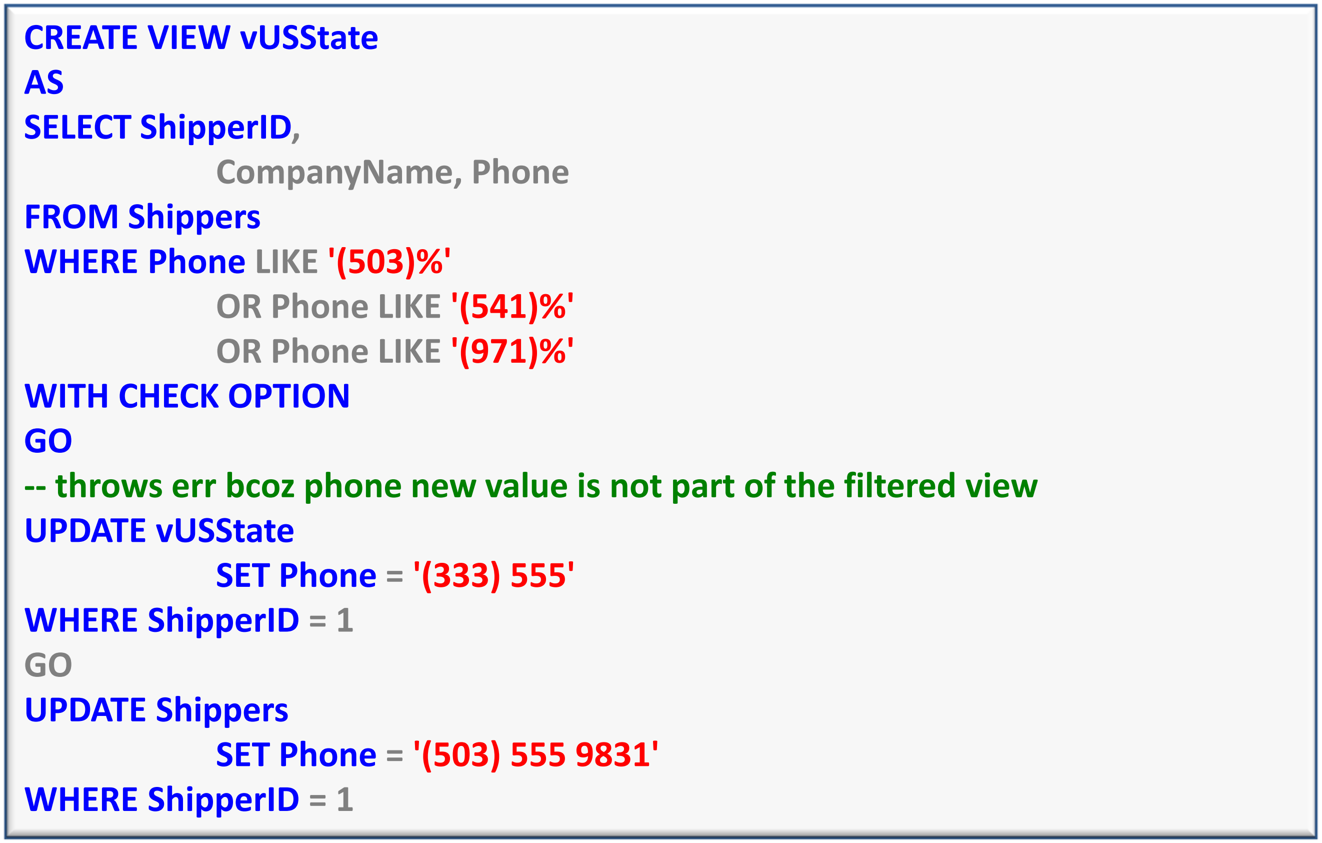
Slide 87

SS42

```
Use Northwind
GO
CREATE VIEW vUSState
AS
SELECT ShipperID,
        CompanyName, Phone
FROM Shippers
WHERE Phone LIKE '(503)%'
        OR Phone LIKE '(541)%'
        OR Phone LIKE '(971)%'
WITH CHECK OPTION
GO
-- throws an error
UPDATE vUSState
        SET Phone = '(333) 555'
WHERE ShipperID = 1
GO
SELECT * FROM vUSState
GO
UPDATE Shippers
        SET Phone = '(333) 555 9831'
WHERE ShipperID = 1
GO
```

Sanjay Singh, 2/22/2007

Example – Update using View

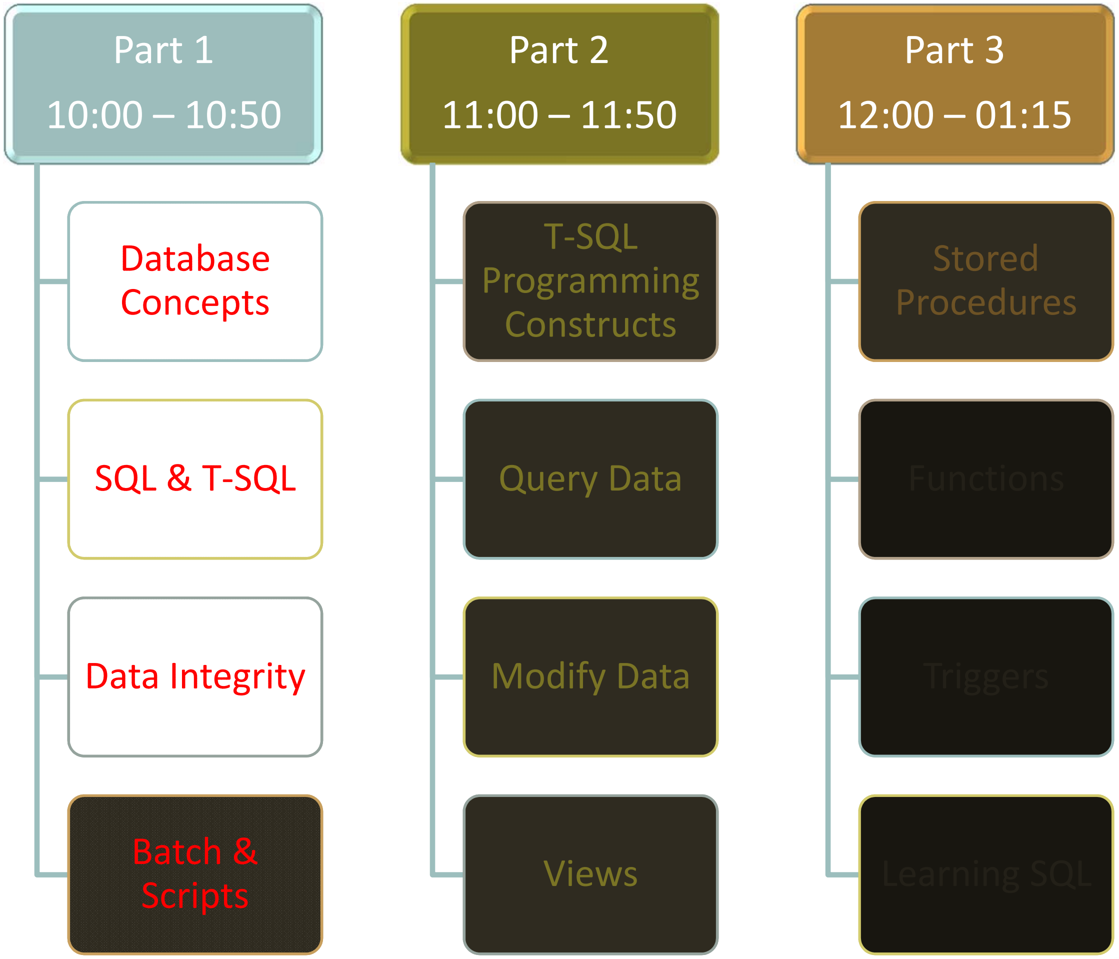


```
CREATE VIEW vUSState
AS
SELECT ShipperID,
		CompanyName, Phone
FROM Shippers
WHERE Phone LIKE '(503)%'
		OR Phone LIKE '(541)%'
		OR Phone LIKE '(971)%'
WITH CHECK OPTION
GO
-- throws err bcoz phone new value is not part of the filtered view
UPDATE vUSState
		SET Phone = '(333) 555'
WHERE ShipperID = 1
GO
UPDATE Shippers
		SET Phone = '(503) 555 9831'
WHERE ShipperID = 1
```

Views Information & Guidelines

- View Information
  - Sp_helptext
  - Sp_help
  - Sp_depends

- Best Practices
  - Use a standard naming convention
  - Use column names explicitly
  - Verify object dependencies before dropping objects
  - Think before using views in views



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

Introducing Stored Procedure

- Collection of compiled SQL commands
- Stored as an object in the SQL Server
- Similar to procedures in other programming languages
  - Accept input parameters & return values
  - Contains programming statements
  - Return status to the caller
- Types of Stored Procedures
  - User-defined
  - Extended
  - System

Benefits

- Encapsulation of business rules in one place
- Sharing of logic by different applications (encourages code reusability)
- Controlled access to database objects
- Shield database schema detail
- Reduce network traffic
- Improves application performance

Impact of using sp_

- Do not create any stored proc using sp_ prefix
- SQL Server uses sp_ to denote system stored procedures
- System proc are special proc provided by SQL Server for getting meta data & administrative tasks
- A sp having same name as system sp will never get executed
- Is different from others as it runs within the context of the database from where it was run

SS43

SS44

Slide 93

SS43

```
USE pubs;
GO
CREATE PROCEDURE dbo.sp_who
AS
    SELECT au_fname, au_lname FROM pubs;
GO
EXEC sp_who;
EXEC dbo.sp_who;
GO
DROP PROCEDURE dbo.sp_who;
GO
```

Sanjay Singh, 2/20/2007

SS44

```
Use TestDB
create proc sp_testsp
as
select db_name()
----
exec sp_testsp

use master
create proc sp_testsp
as
select db_name()

Use TestDB
exec sp_testsp

use master
sp_rename testsp, sp_testsp

exec sp_testsp
```

Sanjay Singh, 2/20/2007

Delayed Name Resolution

- Creating a procedure referencing a table that doesn't exist
- Creating a procedure referencing a table with invalid column SS45
- Try this – referencing another proc that does not exist

➢ Create & Execute before putting a stored proc into the application build SS46

Slide 94

SS45

```
USE TestDB
GO
IF OBJECT_ID ('TestSp') IS NOT NULL
	DROP PROCEDURE TestSp
GO
CREATE PROC TestSP
AS
SELECT * FROM TableThatDoesNotExists
GO
EXEC TestSp
```

Sanjay Singh, 2/20/2007

SS46

```
USE pubs
GO
IF OBJECT_ID ('TestSp') IS NOT NULL
	DROP PROCEDURE TestSp
GO
CREATE PROC TestSP
AS
SELECT InvalidColumn FROM authors
```

Sanjay Singh, 2/20/2007

Creating A Stored Procedure Using Template Explorer

```
-- ================================================
-- Author:                    <Author Name>
-- Create date:               <Create Date>
-- Description:               <Description>
-- ================================================
CREATE PROCEDURE <Procedure_Name>
            -- Add the parameters for the stored procedure here
            <@Param1> <Datatype> = <Default>,
            <@Param2> <Datatype> = <Default> OUTPUT
AS
BEGIN
            -- SET NOCOUNT ON added to prevent extra result sets from
            -- interfering with SELECT statements.
            SET NOCOUNT ON;

            -- declare variables, Logic, validations, control flow statements,
            -- Insert T-SQL Statementstatements for procedure here
END
```

Example – Stored Procedure

```
CREATE PROC dbo.GetAuthorCount
        @state char(10) = NULL,
        @count int OUTPUT
AS
BEGIN
        DECLARE @error INT

        IF @state IS NULL
                SELECT * FROM authors
        ELSE
                SELECT * FROM authors WHERE state = @state

        SET @count = @@ROWCOUNT
        SET @error = @@ERROR

RETURN (@error)
END
```

```
DECLARE @cnt INT, @i int
EXEC @i = dbo.GetAuthorCount 'CA', @cnt OUTPUT
SELECT @cnt as [Count], @i AS [Status]
```

Executing Stored Procedure

- Procedure can be executed using
  - EXEC / EXECUTE
  - Procedure name as first statement in a batch
  - sp_executesql
- Using the variable name in the argument list
  - Variables can be passed in any order
  - Facilitates the usage of default values

SS47

SS48

SS47

```
USE AdventureWorks;
GO
IF OBJECT_ID ( 'HumanResources.usp_GetAllEmployees', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.usp_GetAllEmployees;
GO
CREATE PROCEDURE HumanResources.usp_GetAllEmployees
AS
    SELECT LastName, FirstName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment;
GO

EXECUTE HumanResources.usp_GetAllEmployees;
GO
-- Or
EXEC HumanResources.usp_GetAllEmployees;
GO
-- Or, if this procedure is the first statement within a batch:
HumanResources.usp_GetAllEmployees;
GO
-- Using sp_executesql
DECLARE @str NVARCHAR(100)
SET @str = 'EXEC HumanResources.usp_GetAllEmployees'
EXEC SP_EXECUTESQL @str
GO
DROP PROCEDURE HumanResources.usp_GetAllEmployees;
GO
```

Sanjay Singh, 2/20/2007

SS48

```
USE AdventureWorks;
GO
IF OBJECT_ID ( 'HumanResources.usp_GetEmployees2', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.usp_GetEmployees2;
GO
CREATE PROCEDURE HumanResources.usp_GetEmployees2
    @firstname varchar(20) = '%',
    @lastname varchar(40)
AS
    SELECT LastName, FirstName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment
    WHERE FirstName LIKE @firstname
```

Slide 97 (Continued)

```
        AND LastName LIKE @lastname;
GO

-- will give error as first parameter is mandatory
EXEC HumanResources.usp_GetEmployees2 'Kevin';
-- will not use default
EXEC HumanResources.usp_GetEmployees2 'Kevin', 'Brown';
-- uses default wildchar
EXEC HumanResources.usp_GetEmployees2 @lastname = 'Brown';
```

Sanjay Singh, 2/20/2007

Example – Executing Procedure

```
CREATE PROCEDURE HumanResources.usp_GetAllEmployees
AS
   SELECT LastName, FirstName, JobTitle, Department
   FROM HumanResources.vEmployeeDepartment;
GO

EXECUTE HumanResources.usp_GetAllEmployees;
GO
-- Or
EXEC HumanResources.usp_GetAllEmployees;
GO
-- Or, if this procedure is the first statement within a batch:
HumanResources.usp_GetAllEmployees;
GO
-- Using sp_executesql
DECLARE @str NVARCHAR(100)
SET @str = 'EXEC HumanResources.usp_GetAllEmployees'
EXEC SP_EXECUTESQL @str
```

Example – Using Default Values

```
CREATE PROCEDURE HumanResources.usp_GetEmployees2
   @firstname varchar(20) = '%',
   @lastname varchar(40)
AS
   SELECT LastName, FirstName, JobTitle, Department
   FROM HumanResources.vEmployeeDepartment
   WHERE FirstName LIKE @firstname
      AND LastName LIKE @lastname;
GO

-- will give error as first parameter is mandatory
EXEC HumanResources.usp_GetEmployees2 'Kevin';
-- will not use default
EXEC HumanResources.usp_GetEmployees2 'Kevin', 'Brown';
-- uses default wildchar
EXEC HumanResources.usp_GetEmployees2 @lastname = 'Brown';
```

Returning Values from Stored Proc

- RETURN
  - Used to exit from a procedure
  - Default returns 0
  - Can specify int to return status
- OUTPUT parameter
  - Similar to passing a variable by reference
  - Can be used to return non-int values
  - The OUTPUT keyword must be specified
- SELECT statement
  - Return informative messages
  - Returning in a table variable

SS49

SS50

SS51

SS52

Slide 100

SS49

```
USE AdventureWorks;
GO
IF OBJECT_ID ('checkstate') IS NOT NULL
        DROP PROCEDURE checkstate
GO
CREATE PROCEDURE checkstate @param varchar(11)
AS
DECLARE @City varchar(10)
SELECT @City = City FROM Person.vAdditionalContactInfo WHERE ContactID = @param
IF @City = 'Seattle'
   RETURN 1
ELSE IF @City = 'Redmond'
   RETURN 2
ELSE IF @City = 'Edmonds'
        RETURN 'Found'
ELSE
        RETURN
GO


DECLARE @return_status int;
EXEC @return_status = checkstate '1';
SELECT 'Return Status' = @return_status;
GO
DECLARE @return_status int;
EXEC @return_status = checkstate '3';
SELECT 'Return Status' = @return_status;
GO
DECLARE @return_status int;
EXEC @return_status = checkstate '4';
SELECT 'Return Status' = @return_status;
GO
DECLARE @return_status int;
EXEC @return_status = checkstate '2';
SELECT 'Return Status' = @return_status;
GO
```

Sanjay Singh, 2/20/2007

SS50

```
USE AdventureWorks;
GO
```

Slide 100 (Continued)

```
IF OBJECT_ID ('checkoutput') IS NOT NULL
        DROP PROCEDURE checkoutput
GO
CREATE PROCEDURE checkoutput @param varchar(11),
@cityname varchar(10) output
AS
SELECT @cityname = City FROM Person.vAdditionalContactInfo WHERE ContactID = @param
GO

-- without using output in the parameter
DECLARE @city varchar(10);
EXEC checkoutput '1',  @city --output;
SELECT 'City Name' = @city;
GO
-- using output in the parameter
DECLARE @city varchar(10);
EXEC checkoutput '1',  @city output;
SELECT 'City Name' = @city;
GO
```

Sanjay Singh, 2/20/2007

SS51

```
USE AdventureWorks;
GO
IF OBJECT_ID ('checkoutput') IS NOT NULL
        DROP PROCEDURE checkoutput
GO
CREATE PROCEDURE checkoutput @param varchar(11)
AS
declare @cityname varchar(10)
SELECT @cityname = City FROM Person.vAdditionalContactInfo WHERE ContactID = @param
IF @cityname = 'Seattle'
        SELECT '0' as ErrorID, 'Success' as ErrorMessage
ELSE
        SELECT '-1' as ErrorID, 'Failed to get City Name' as ErrorMessage
GO

-- without using output in the parameter
EXEC checkoutput '1'
GO
EXEC checkoutput '2'
```

Slide 100 (Continued)

```
GO
```

Sanjay Singh, 2/20/2007

SS52

```
USE AdventureWorks;
GO
IF OBJECT_ID ('checkoutput') IS NOT NULL
        DROP PROCEDURE checkoutput
GO
CREATE PROCEDURE checkoutput @param varchar(11)
AS
declare @cityname varchar(10)
SELECT @cityname = City FROM Person.vAdditionalContactInfo WHERE ContactID = @param
IF @cityname = 'Seattle'
        SELECT '0' as ErrorID, 'Success' as ErrorMessage
ELSE
        SELECT '-1' as ErrorID, 'Failed to get City Name' as ErrorMessage
GO

declare @tblstatus table (errid int, errmsg varchar(10))
INSERT @tblstatus EXEC checkoutput '1'
SELECT * FROM @tblstatus
```

Sanjay Singh, 2/20/2007

Example – Return Status in Table

```
CREATE PROCEDURE checkoutput @param VARCHAR(11)
AS
BEGIN
        DECLARE @cityname VARCHAR(10)

        SELECT @cityname = City FROM
        Person.vAdditionalContactInfo
        WHERE ContactID = @param

        IF @cityname = 'Seattle'
                SELECT '0' AS ErrorID, 'Success' AS ErrorMessage
        ELSE
                SELECT '-1' AS ErrorID, 'Failed to get City Name' AS ErrorMessage

END
```

```
-- Execute the proc
DECLARE @tblstatus TABLE (errid INT, errmsg VARCHAR(10))
INSERT @tblstatus EXEC checkoutput '1'
SELECT * FROM @tblstatus
```

Handling Errors

- @@ERROR
  - Is the primary means of detecting errors in T-SQL statements
  - Returns 0 if the last statement was successful
  - IF statement resets the value
  - Always set @@ERROR & @@ROWCOUNT to a variable immediately after the T-SQL statement

SS53

SS54

- RAISERROR
  - To return message using the same format as SQL Server sends system messages
  - Can be customized

SS55

Slide 102

SS53

```
USE pubs
GO
DELETE FROM dbo.authors WHERE city = 'Unknown'
SELECT @@ERROR
GO
```

Sanjay Singh, 2/20/2007

SS54

```
DECLARE @ErrorVar INT

RAISERROR(N'Message', 16, 1);
-- the expectation is it returns error no 50000
-- but it is not printed in the following statement. why?
IF @@ERROR <> 0
    -- This PRINT statement prints 'Error = 0' because
    -- @@ERROR is reset in the IF statement above.
    PRINT N'Error = ' + CAST(@@ERROR AS VARCHAR(8));
GO
-----------------------------------
--Use a variable to store the error no

DECLARE @ErrorVar INT

RAISERROR(N'Message', 16, 1);
-- Save the error number before @@ERROR is reset by
-- the IF statement.
SET @ErrorVar = @@ERROR
IF @ErrorVar <> 0
-- This PRINT statement correctly prints 'Error = 50000'.
    PRINT N'Error = ' + CAST(@ErrorVar AS NVARCHAR(8));
GO
```

Sanjay Singh, 2/20/2007

SS55

```
DECLARE @DBID INT;
SET @DBID = DB_ID();

DECLARE @DBNAME NVARCHAR(128);
SET @DBNAME = DB_NAME();

RAISERROR
    (N'The current database ID is:%d, the database name is: %s.',
```

Slide 102 (Continued)

```
    10, -- Severity.
    1, -- State.
    @DBID, -- First substitution argument.
    @DBNAME); -- Second substitution argument.
GO
```

Sanjay Singh, 2/20/2007

Example – Right Way To Capture Error

```
DECLARE @ErrorVar INT

RAISERROR('Message', 16, 1);
-- the expectation is it returns error no 50000
-- but it is not printed in the following statement. why?
IF @@ERROR <> 0
    -- This PRINT statement prints 'Error = 0' because
    -- @@ERROR is reset in the IF statement above.
    PRINT 'Error = ' + CAST(@@ERROR AS VARCHAR(8));
```

```
DECLARE @ErrorVar INT

RAISERROR('Message', 16, 1);
-- Save the error number before @@ERROR is reset by
-- the IF statement.
SET @ErrorVar = @@ERROR
IF @ErrorVar <> 0
-- This PRINT statement correctly prints 'Error = 50000'.
    PRINT 'Error = ' + CAST(@ErrorVar AS NVARCHAR(8));
```

Example – DROP Vs ALTER Proc

- DROP removes all the permission and they need to be re-created.

```
IF OBJECT_ID('dbo.GetReport') IS NULL
BEGIN
  -- Create dummy Procedure
  EXECUTE ( 'CREATE PROCEDURE dbo.GetReport AS RETURN 0' )
END
GO
ALTER procedure [dbo].[GetReport]
   -- <parameters>
AS
BEGIN
                -- Stored procedure body
END
GO
   IF ( object_id('dbo.GetReport') IS NOT NULL )
   BEGIN
      GRANT EXECUTE ON dbo.GetReport TO RequiredRole
   END -- Object exists
```

Viewing Stored Procedure Info

- Viewing the definition (contents of the sp)
  - Sp_helptext
- View information about the stored proc
  - Sp_help
- View the dependencies
  - Sp_depends

SS56

SS57

SS58

Slide 105

SS56

```
USE AdventureWorks;
sp_helptext uspGetBillOfMaterials
```

Sanjay Singh, 2/20/2007

SS57

```
USE AdventureWorks;
GO
sp_help uspGetBillOfMaterials
```

Sanjay Singh, 2/20/2007

SS58

```
USE AdventureWorks;
GO
sp_depends uspLogError
```

Sanjay Singh, 2/20/2007

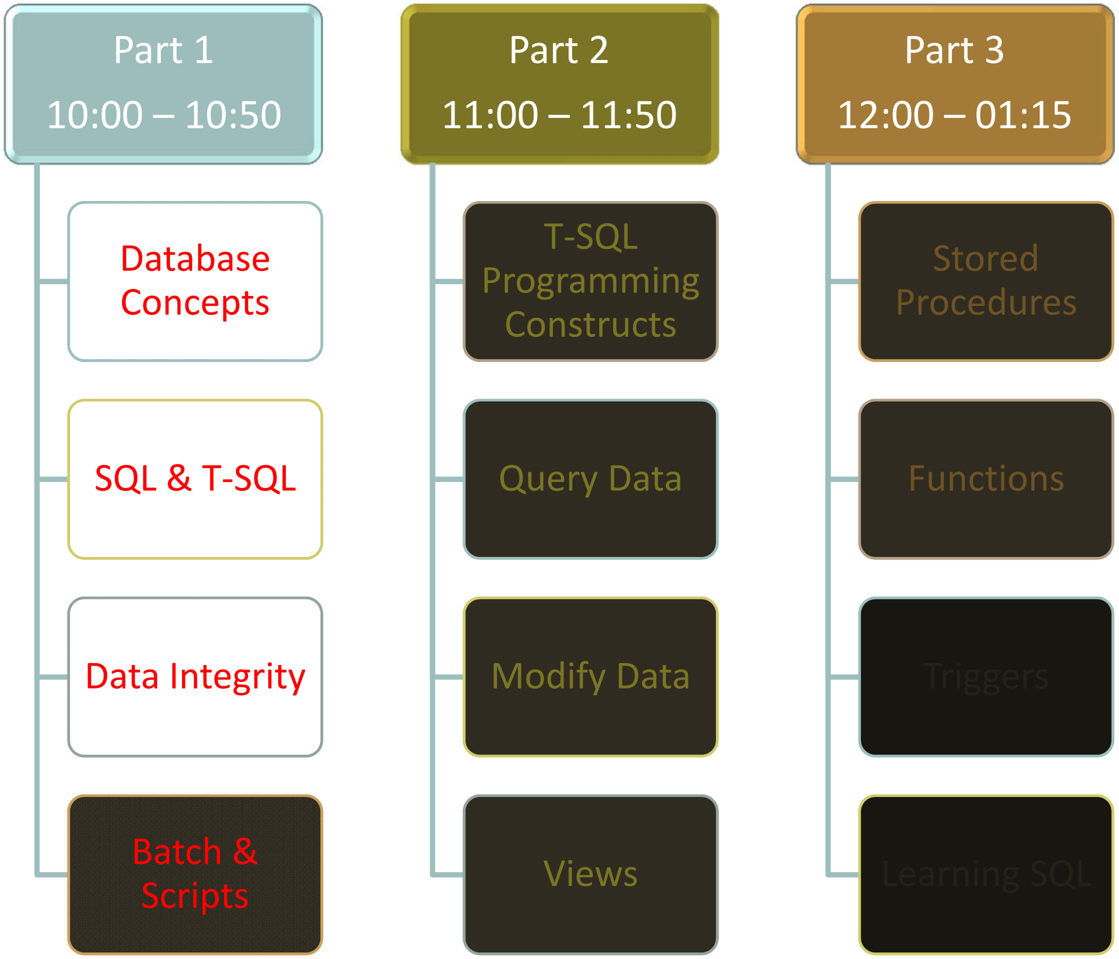
View Dependencies

```
CREATE PROC spFactorial
   @ValueIn int,
   @ValueOut int OUTPUT
AS
BEGIN
   DECLARE @InWorking int
   DECLARE @OutWorking int

   IF @ValueIn != 1
   BEGIN
      SELECT @InWorking = @ValueIn - 1
      -- calling a sp that does not exist
      EXEC NonexistenceSp @InWorking, @OutWorking OUTPUT
      -- In 2005 it has become smart
      --EXEC spFactorial @InWorking, @OutWorking OUTPUT
      SELECT @ValueOut = @ValueIn * @OutWorking
   END
   ELSE
      SELECT @ValueOut = 1
   RETURN
END
```

```
SP_DEPENDS spFactorial

DECLARE @result INT
EXEC spFactorial 3, @result OUTPUT
SELECT @result
```

Stored Procedure – Guidelines

- Verify input parameters
- Design each stored procedure to accomplish a single task
- Keep your transaction short
- Check error after every DML statement
- Have a consistent error handling
- Never, never put a stored procedure into server without complete testing



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

Introducing UDFs

- User Defined Functions provides the capability to create our own functions
- Its only purpose is to return data
- Types of Functions
  - Deterministic Functions
    - Always returns the same information if the parameters are same
  - Non-Deterministic Functions
    - Can return different results each time they are called with same input values

SS89

Slide 109

SS89

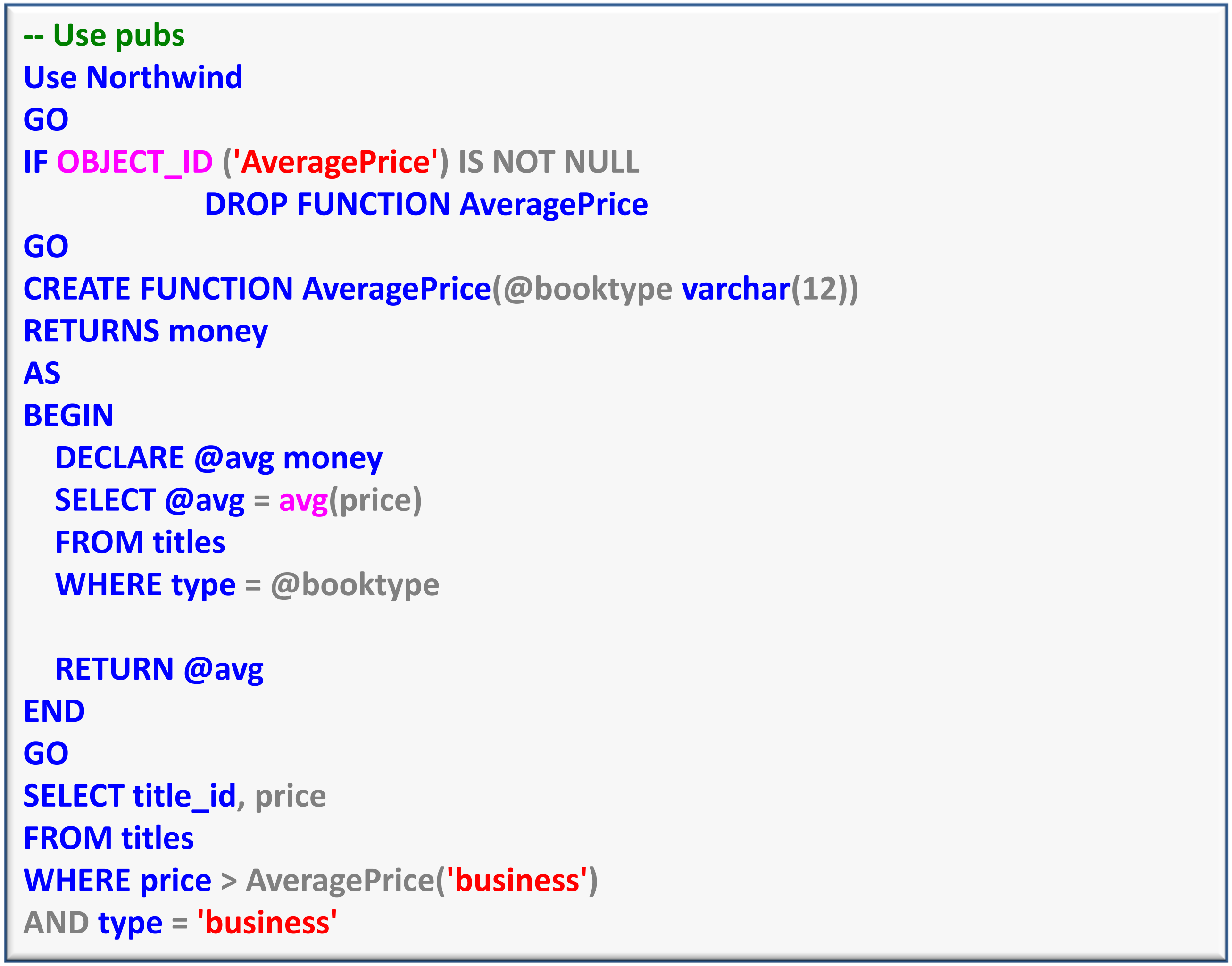
```
-- Example of deterministic
SELECT UPPER('xyz')
-- Non deterministic
SELECT GETDATE()
```

Sanjay Singh, 2/27/2007

User Defined Function

- Types of UDFs according to their return value
    - Scalar functions that return a scalar value
        - Used wherever expression are used (column list, WHERE, GROUP BY, ORDER, HAVING)
        - Cannot be used in FROM clause
    - Table valued function that return a result set
        - Used in the FROM clause
    - Inline UDF (special case of table valued) but limited to a single SELECT statement

Example – Creating A Function



```
-- Use pubs
Use Northwind
GO
IF OBJECT_ID ('AveragePrice') IS NOT NULL
                DROP FUNCTION AveragePrice
GO
CREATE FUNCTION AveragePrice(@booktype varchar(12))
RETURNS money
AS
BEGIN
   DECLARE @avg money
   SELECT @avg = avg(price)
   FROM titles
   WHERE type = @booktype

   RETURN @avg
END
GO
SELECT title_id, price
FROM titles
WHERE price > AveragePrice('business')
AND type = 'business'
```

UDF Benefits

- Similar to the one provide by Stored Proc
  - Modular programming
  - Faster execution
  - Reduced network traffic
- The return table can be used directly in a FROM clause
  - No need to have an intermediate table
- Creating parameterized views
- Note
  - Data modification to an existing table cannot happen inside a UDF
  - If you want to use temp tables, then use table variables
  - UDFs are local to a database
  - If Execute is used to call UDF, then owner name is optional
  - ORDER BY cannot be used in inline UDF
    - TOP 100 percent

SS91

SS93

SS94

SS95

Slide 112

SS91

```
IF OBJECT_ID ('dbo.fn_Test') IS NOT NULL
        DROP FUNCTION [dbo].[fn_Test]
GO
ALTER FUNCTION [dbo].[fn_Test] ()
RETURNS datetime
AS
BEGIN
        INSERT INTO TestTable VALUES ('10', 'Test Data')
        RETURN GETDATE()
END
```

Sanjay Singh, 3/4/2007

SS93

```
-- will not work
SELECT sanjayms.Northwind.dbo.MaxProductID()
-- will work
SELECT Northwind.dbo.MaxProductID()
```

Sanjay Singh, 2/27/2007

SS94

```
USE Northwind
GO
--Declare a variable to receive the result of the UDF
DECLARE @Total money

EXECUTE @Total = TotalPrice 12, 25.4, 0.0
--EXECUTE @Total = dbo.TotalPrice 12, 25.4, 0.0
SELECT @Total
```

Sanjay Singh, 2/27/2007

SS95

```
USE Northwind
GO
-- Returns Products ordered by ProductName
CREATE FUNCTION dbo.OrderedProducts()
RETURNS TABLE
AS
RETURN (SELECT TOP 100 PERCENT *
FROM Products
ORDER BY ProductName ASC)
GO
-- Test the function
SELECT TOP 10 ProductID, ProductName
FROM dbo.OrderedProducts()
```

Sanjay Singh, 2/27/2007

Scalar-Valued Functions

- Can accepts parameters (up to 1024)
- Returns one value
- All statement is enclosed with BEGIN…END
- Must contain a RETURN statement
- Invoking Scalar Functions
  - As Expression
  - In SELECT
  - Using EXECUTE
    - Must not specify parenthesis

SS60

SS61

SS62

SS85

SS92

Slide 113

SS60

```
-- Use pubs
Use Northwind
GO
IF OBJECT_ID ('AveragePrice') IS NOT NULL
          DROP FUNCTION AveragePrice
GO
CREATE FUNCTION AveragePrice(@booktype varchar(12))
RETURNS money
AS
BEGIN
    DECLARE @avg money
    SELECT @avg = avg(price)
    FROM titles
    WHERE type = @booktype

    RETURN @avg
END
GO
SELECT title_id, price
FROM titles
WHERE price > AveragePrice('business')
AND type = 'business'
```

Sanjay Singh, 2/27/2007

SS61

```
SELECT dbo.AveragePrice ('business')
```

Sanjay Singh, 2/21/2007

SS62

```
DECLARE @avg money
EXEC @avg = dbo.AveragePrice ('business')
SELECT @avg
```

Sanjay Singh, 2/21/2007

SS85

```
Use Northwind

IF OBJECT_ID ('dbo.GetNoOfSubordinates') IS NOT NULL
          DROP FUNCTION dbo.GetNoOfSubordinates
GO
CREATE FUNCTION GetNoOfSubordinates (@EmployeeID INT)
RETURNS INT
AS
BEGIN
```

```
        RETURN (SELECT COUNT(*) FROM Employees WHERE ReportsTo = @EmployeeID)
END

-- first get all the employeeid
-- then for each employeeid it calls the function
SELECT * FROM Employees
WHERE dbo.GetNoOfSubordinates (EmployeeID) > 0
```

Sanjay Singh, 2/25/2007

SS92

```
-- example showing the different places where scalar UDF can be used
Use Northwind
SELECT dbo.MaxProductID()
GO
SELECT ProductID, dbo.MaxProductID() AS 'MaxID'
FROM Products
GO
UPDATE [Order Details]
SET ProductID = dbo.MaxProductID()
WHERE ProductID = 25
GO
SELECT ProductID, MaxID
FROM Products
CROSS JOIN (SELECT dbo.MaxProductID() AS 'MaxID') AS MI
GO
SELECT P.ProductID, Quantity
FROM Products AS P
JOIN [Order Details] AS OD
ON P.ProductID = OD.ProductID
AND P.ProductID = dbo.MaxProductID()
GO
SELECT P.ProductID, Quantity
FROM Products AS P
JOIN [Order Details] AS OD
ON P.ProductID = OD.ProductID
WHERE P.ProductID = dbo.MaxProductID()
GO

SELECT P.ProductID, SUM(Quantity)
FROM Products AS P
JOIN [Order Details] AS OD
```

Slide 113 (Continued)

```
ON P.ProductID = OD.ProductID
GROUP BY P.ProductID
HAVING P.ProductID = dbo.MaxProductID()
GO
SELECT ProductID, ProductName,
CASE ProductID
WHEN dbo.MaxProductID() THEN 'Last Product'
ELSE ''END AS Note
FROM Products
GO
DECLARE @ID int
SET @ID = dbo.MaxProductID()
```

Sanjay Singh, 2/27/2007

Example – Scalar Function

```
Use TestDB
GO
IF OBJECT_ID ('dbo.fn_Test') IS NULL
                DROP FUNCTION [dbo].[fn_Test]
GO
ALTER FUNCTION [dbo].[fn_Test] ()
RETURNS datetime
AS
BEGIN
                -- Will error out (invalid use of side-effecting operator)
                INSERT INTO TestTable VALUES ('10', 'Test Data')
                RETURN GETDATE()
END
```

```
Use Pubs
DECLARE @avg money
-- parenthesis will give error
EXEC @avg = dbo.AveragePrice ('business')
SELECT @avg
```

Table-Valued Functions

- Returns a result set
- RETURNS clause use TABLE as return type
- Two ways to write table valued functions
  - As inline functions
    - Specifies TABLE as returns with no definition
    - No function body delimited by BEGIN…END
    - There is only one SELECT statement
  - Multi-statement functions
    - Table variable declared
    - Returns the table variable specified in RETURNS

SS63

SS64

Slide 115

SS63

```
USE pubs
GO
IF OBJECT_ID ('SalesByStore') IS NOT NULL
          DROP FUNCTION SalesByStore
GO
CREATE FUNCTION SalesByStore(@storid varchar(30))
RETURNS TABLE
AS
RETURN (SELECT title, qty
       FROM sales s, titles t
       WHERE s.stor_id = @storid AND t.title_id = s.title_id)
GO
SELECT * FROM dbo.SalesByStore ('8042')
```

Sanjay Singh, 2/27/2007

SS64

```
USE pubs
GO
IF OBJECT_ID ('dbo.SalesByStore_MS') IS NOT NULL
          DROP FUNCTION dbo.SalesByStore_MS
GO
CREATE FUNCTION SalesByStore_MS(@storid varchar(30))
RETURNS @sales TABLE(title varchar(80), qty int)
AS
BEGIN
    INSERT @sales
       SELECT title, qty
       FROM sales s, titles t
       WHERE s.stor_id = @storid AND t.title_id = s.title_id
    RETURN
END
GO
SELECT * FROM dbo.SalesByStore_MS ('8042')
-- following works for table valued function
SELECT * FROM SalesByStore_MS ('8042')
```

Sanjay Singh, 2/25/2007

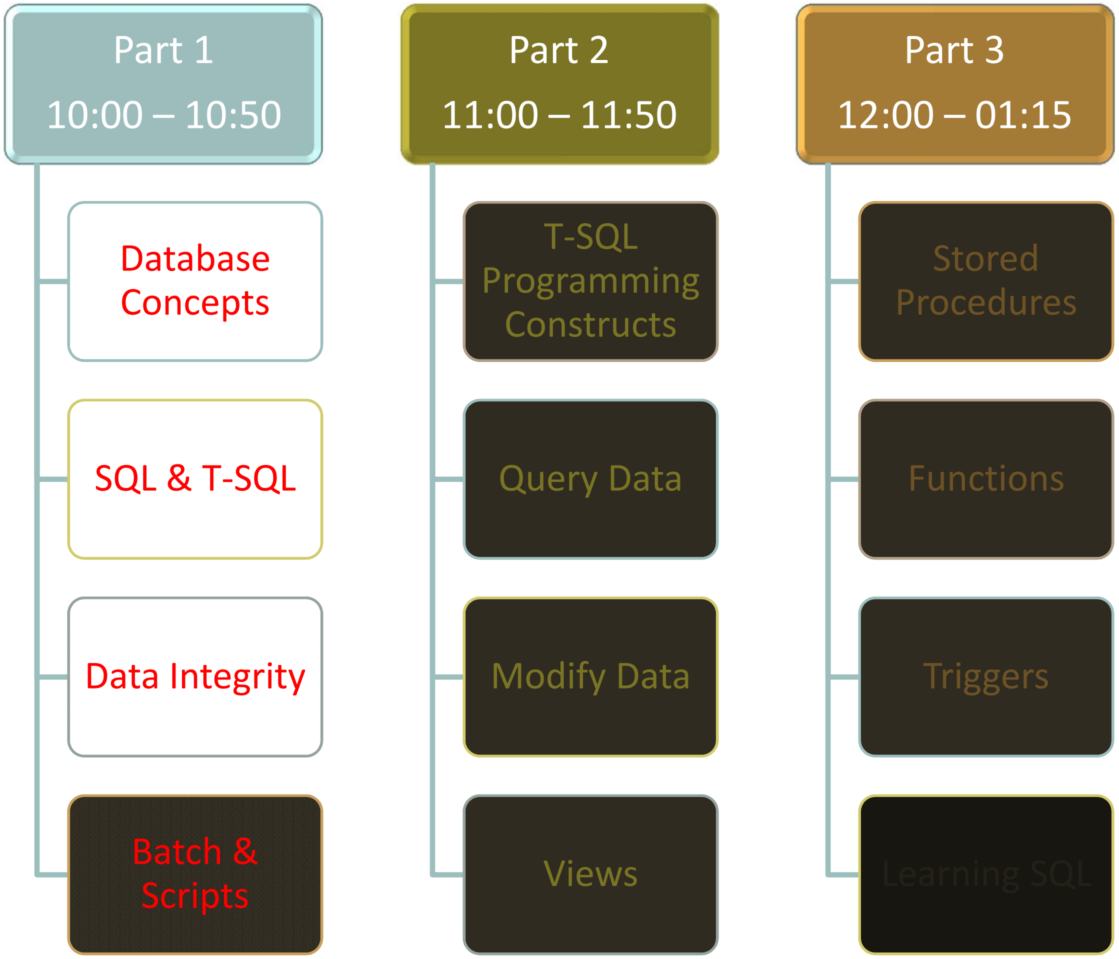
Example –Table-Valued Function

```
CREATE FUNCTION SalesByStore(@storid varchar(30))
RETURNS TABLE
AS
RETURN (SELECT title, qty
      FROM sales s, titles t
      WHERE s.stor_id = @storid AND t.title_id = s.title_id)
GO
```

```
CREATE FUNCTION SalesByStore_MS(@storid varchar(30))
RETURNS @sales TABLE(title varchar(80), qty int)
AS
BEGIN
   INSERT @sales
      SELECT title, qty
      FROM sales s, titles t
      WHERE s.stor_id = @storid AND t.title_id = s.title_id
   RETURN
END
```

UDF – Guidelines

- A UDF must do only one specific and simple task
- Use scalar function on small result sets
- Use multi-line statement function instead of stored procedures that returns table
- Convert a SELECT only view into parameterized view using Inline-Functions



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

Introducing Trigger

- SQL Server provides two primary mechanism business rules & data integrity
  - Constraints
  - Triggers
- Trigger is a special type of stored proc that is executed on an event
  - DML Triggers
  - DDL Triggers
- It cannot be called directly
- A trigger does not accept any parameter

DML Triggers

- Is invoked when a DML event takes place
  - INSERT, UPDATE, DELETE
- Type of DML Triggers
  - AFTER Triggers
    - Is executed after the DML statement is performed
  - INSTEAD OF Triggers
    - Executed in place of the original DML
    - Often used in views to update base table

DML Trigger

- INSERT trigger
  - Is executed when a row is inserted
  - For each row inserted SQL Server creates a copy of the row in a special table INSERTED
- DELETE trigger
  - A copy of each deleted row is inserted into DELETED
- UPDATE trigger
  - Treat each update as delete followed by update
  - Copies data into both INSERTED & DELETED table

SS67

SS67

```
Use tempdb
GO

IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
GO

CREATE TABLE dbo.T1
(
  keycol INT          NOT NULL PRIMARY KEY,
  datacol VARCHAR(10) NOT NULL
);
GO

CREATE TRIGGER trg_T1_iud ON dbo.T1 FOR INSERT, UPDATE, DELETE
AS
DECLARE @rc AS INT;
SET @rc = @@rowcount;

IF @rc = 0
BEGIN
        PRINT 'No rows affected';
        RETURN;
END

IF EXISTS(SELECT * FROM inserted)
BEGIN
        IF EXISTS(SELECT * FROM deleted)
        BEGIN
                PRINT 'UPDATE identified';
                SELECT * FROM DELETED
                SELECT * FROM INSERTED
        END
        ELSE
        BEGIN
                PRINT 'INSERT identified';
                SELECT * FROM INSERTED
        END
END
        ELSE
```

Slide 121 (Continued)

```
BEGIN
        PRINT 'DELETE identified';
        SELECT * FROM DELETED
END
GO

--- Testing the code
--INSERT INTO T1 SELECT 1, 'A' WHERE 1 = 0;
--
--INSERT INTO T1 SELECT 1, 'A';
--
--UPDATE T1 SET datacol = 'AA' WHERE keycol = 1;
--
--DELETE FROM T1 WHERE keycol = 1;
```

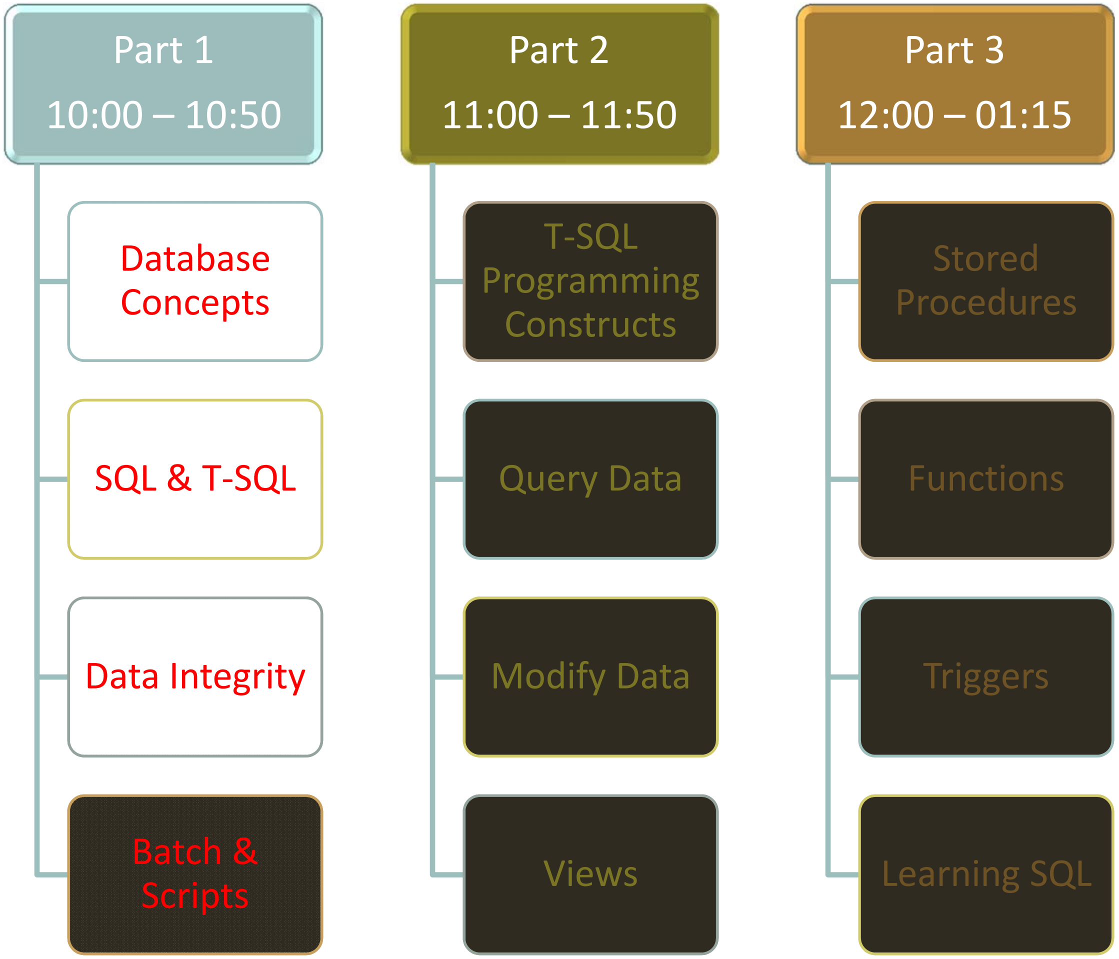
Sanjay Singh, 2/24/2007

Example – Trigger

```
CREATE TRIGGER trg_T1_iud ON dbo.T1 FOR INSERT, UPDATE, DELETE
AS
DECLARE @rc AS INT;
SET @rc = @@rowcount;
IF @rc = 0
        RETURN;
IF EXISTS(SELECT * FROM inserted)
BEGIN
        IF EXISTS(SELECT * FROM deleted)
        BEGIN
                SELECT * FROM DELETED
                SELECT * FROM INSERTED
        END
        ELSE
                SELECT * FROM INSERTED
END
        ELSE
BEGIN
        SELECT * FROM DELETED
END
```

Triggers – Guidelines

- Use Triggers only when necessary
- Keep trigger definition statements as simple as possible
- Provide comments wherever triggers are used for example in the create table
- When fixing a bug, look for the existence of triggers and its impact



Part 1

10:00 – 10:50

- Database Concepts
- SQL & T-SQL
- Data Integrity
- Batch & Scripts

Part 2

11:00 – 11:50

- T-SQL Programming Constructs
- Query Data
- Modify Data
- Views

Part 3

12:00 – 01:15

- Stored Procedures
- Functions
- Triggers
- Learning SQL

SQL Server Management Studio

- Pin SSMS it to the start menu
- Use F1 to get help on a command
- CTRL-R to hide the result screen
- Use Parse to verify syntax
- Convert Tabs to Spaces
- Template Explorer Example
  - Table -> add_column
  - Stored Procedure -> basic template
  - You can add your own template
    - Organization wide common templates can be put here

SS72

Slide 125

SS72

```
Use pubs

-- use CTRL+F5
-- it will work even though ajunk is present
-- next change FROM to FRO and press CTRL+F5
SELECT ajunk.au_lname AS [Last Name], t.title Book_Title
FROM authors a
JOIN titleauthor ta
        ON a.au_id = ta.au_id
JOIN titles t
        ON t.title_id = ta.title_id
```

Sanjay Singh, 2/25/2007

Learning SQL

- SQL Server 2005 BOL (Books Online)
    - Saving topics as Favorites
- Websites
    - Tools -> Help -> Online
        - It displays the SQL server sites
    - www.transactsql.com
    - Video Series: SQL Server 2005 for Beginners
    - http://www.sql-server-performance.com/
- SQL Server 2000 FAQ
- Books
    - Murach’s SQL for SQL Server
    - Beginning SQL Server 2005 Programming
    - SQL Server 2005 T-SQL Recipes: A Problem-Solution Approach
    - Insider SQL Server 2000
    - Inside SQL Server 2005: T-SQL Querying

http://sqltips/