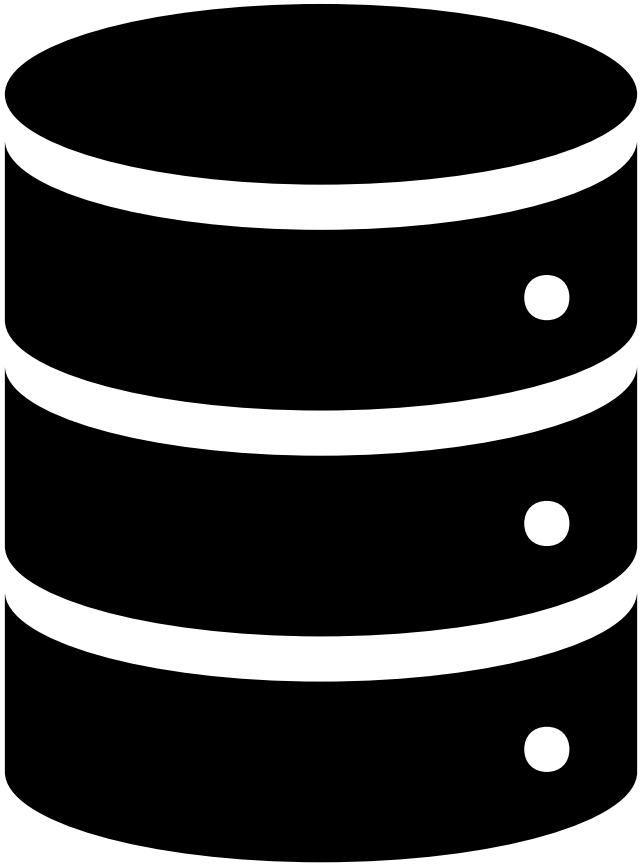


# COM106: Introduction to Databases

## Stored Procedures and Triggers

# Stored Procedures and Triggers

## Stored Procedure

SQL Server normally processes SQL queries sent from a client machine across a network

An SQL statement (or a group of related SQL statements) can be stored on the server as a **Stored Procedure** and used for repetitive tasks

In SQL Server SET NOCOUNT ON is often set to stop number of rows affected being output (reduced network traffic - @@ROWCOUNT can be used if number of rows affected is required)

## Advantages

### Improves performance

- Precompiled
- Allows INPUT & OUTPUT values to be used
- Reduced network traffic

### Improves security

- Prevents alteration of clauses (minimises SQL injection attack) due to precompilation

### Ease of maintenance

- Modular
- Easily changed (Stored logic independent of application)

# Stored Procedures

SQL server has several commands for Stored Procedure

- CREATE PROCEDURE
- ALTER PROCEDURE
- DROP PROCEDURE
- A Stored Procedure is called by the **EXEC**(UTE) command

The list of stored created procedures is kept in *sys.procedures*

SQL server has two types of Stored Procedure:

**System Stored Procedure (built-in)**

- Stored in master database and prefixed by sp_
- For example:
  - *sp_helpdbfixedrole* - returns a list of fixed database roles
  - *sp_who* - returns information about which users are running which processes
  - *sp_helptext objectname* - returns the definition of any object (e.g. function, trigger, stored procedure, view)

**User Defined Stored Procedures**

- Created by the user via SQL code….

# Stored Procedures

**Create Procedure Syntax**

CREATE PROCEDURE *procedurename*
[INPUT @parameter1 datatype [length] ]
[OUTPUT @parameter2 datatype [length] ]
AS
[BEGIN]
    [@parameter1 datatype [length] ]
    SQL statement(s)
[END]

**Drop Procedure Syntax**

DROP PROCEDURE *procedurename*

SQL Server acts as a DB server - calls are made in front end programs (C#, Visual Basic, Java etc) to run the stored procedures

A stored procedure for an INSERT, UPDATE or DELETE query just requires **INPUT** parameters

A stored procedure for a retrieval query requires:

- **INPUT** parameters (to send input values from calling program)
- **OUTPUT** parameters (to return result values to calling program)

# Stored Procedures

For example, consider the following database schema:

**EMP** (enum, ename, salary, dnum*)
**DEPT** (dnum, dname, numemp)

Where: *numemp* is number of employees
*enum*, *dnum* and *numemp* are defined as tinyint
*ename* and *dname* defined as char(15)
*salary* is defined as int

**Example of an INSERT Procedure**

Write a Stored Procedure to:

*insert a new department with input values for department number, name and number of employees*

```
CREATE PROCEDURE InsDept (@dnum tinyint, @dname char(15), @numemp tinyint  )
AS
INSERT INTO DEPT
VALUES (@dnum, @dname, @numemp);
```

To insert a new record in the DEPT table - execute the stored procedure InsDept with the input data required. For example:

```
EXEC InsDept  @dnum = 41, @dname = 'finance', @numemp = 0;

EXEC InsDept  @dnum = 51, @dname = 'personnel', @numemp = 0;
```

# Stored Procedure

**Example of an UPDATE Procedure**

*Change the salary of a named employee to a new value*

```
CREATE PROCEDURE UpdateEmp (@ename char(15), @sal int)
AS UPDATE emp
SET salary = @sal
WHERE ename = @ename;
```

To update the salary of a named employee - execute the stored procedure **UpdateEmp** with the input data required. For example:

```
EXEC UpdateEmp @enam = 'anderson', @sal = 33333;
```

**Example of a RETREVAL Procedure**

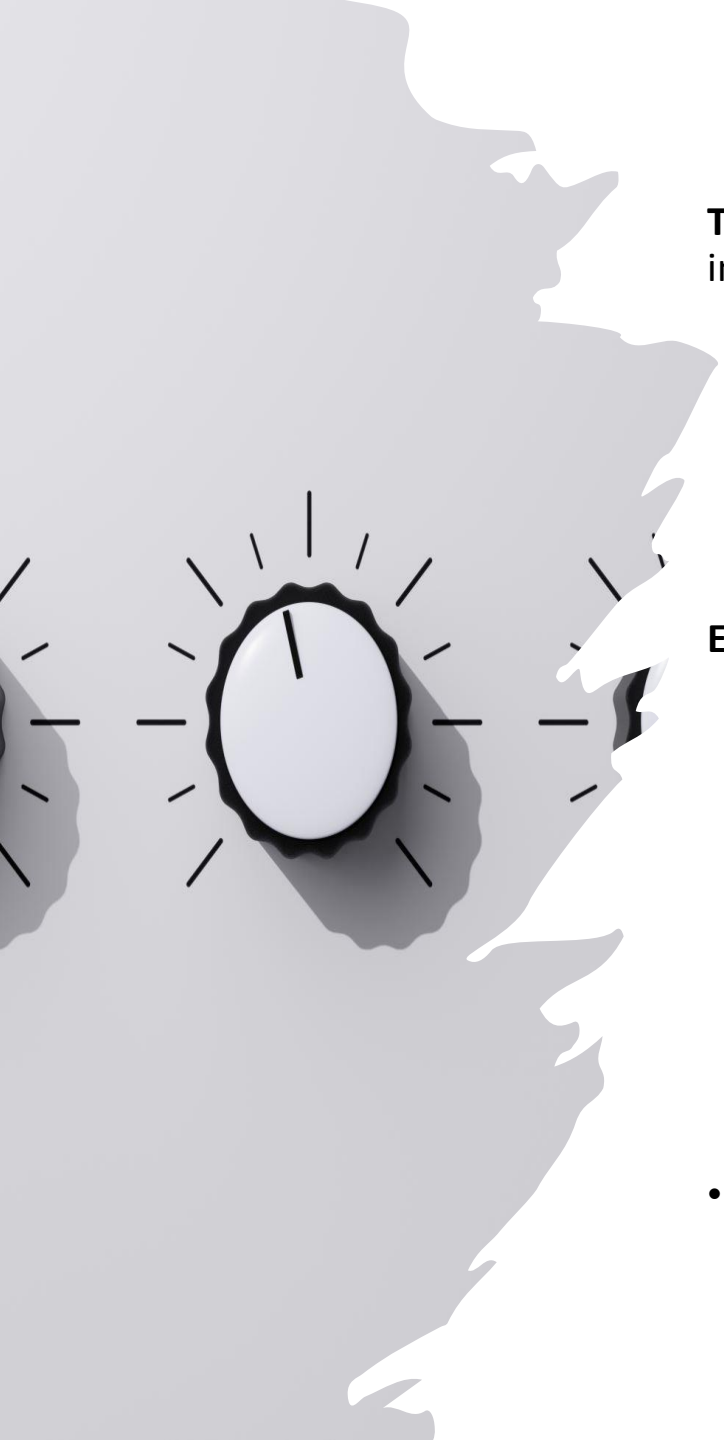
*Get the salary of an employee for an input employee number*

```
CREATE PROCEDURE GetSalByEnum (@enum tinyint,  @sal   int   OUTPUT)
AS SELECT @sal  = salary
FROM emp
WHERE enum = @enum;
```

*For queries returning multiple row result tables, the DBMS provides methods for moving through each row - CURSOR, OFFSET and FETCH*

To get the salary of employee number 5:

```
DECLARE @salary int
EXEC GetSalByEnum  @enum = 5,  @sal=@salary OUTPUT
SELECT  @salary       /* displays salary value where enum = 5  */
```



# Triggers

**Triggers** are special **stored procedures** which are automatically implemented (fired) in response to a database **event** such as:

- A database manipulation (DML) statement (**DELETE**, **INSERT**, or **UPDATE**).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN)

**Examples of Trigger use**

- Archive a copy of data that is being deleted
- Maintain a log of the updates made to the database (audit trail)
- Automatically perform important updates (e.g. decrement the stock level whenever an order is processed)
- Prevent invalid transactions
- Execute rules (e.g. send an email to management whenever changes are made to certain records)
- Impose security authorisations

- **Trigger Commands**
  - CREATE TRIGGER
  - ALTER TRIGGER
  - DISABLE TRIGGER
  - DROP TRIGGER
  - ENABLE TRIGGER

# Triggers

**Create Trigger Syntax**

CREATE TRIGGER triggername ON table
INSTEAD OF |AFTER INSERT|DELETE|UPDATE
AS BEGIN
SQL statement(s)
END

In SQL Server, a list of triggers can be examined using
SELECT * FROM sys.triggers;

**AFTER Triggers**

Trigger statements are automatically executed **after** a data modification occurs
Uses two virtual tables, **Inserted** and **Deleted** (containing new and old values of objects being modified)

**Inserted** holds all new rows

**Deleted** holds all deleted (old) rows

**Example of Trigger use**

Consider a database for a bank with an account table

ACCOUNT (accnum, balance) where *accnum* is the account number and *balance* is the account balance

Details of any deleted account should be **automatically** archived

Details of any transactions on an account should be **automatically** logged

# Triggers

This can be achieved using two Triggers:

**tr1** copies any deleted account details into an archive table ARCHACC (accnum, balance)

**tr2** copies any updated balance values (old and new) into

AUDIT (ID, accnum, old_balance, new_balance, atime)

where *ID* is autoincrement , *old_balance* and *new_balance* are the old and new balances and a*time* is the time the transaction occurred

**Trigger tr1:**

```
CREATE TRIGGER tr1 ON ACCOUNT AFTER DELETE AS
BEGIN
    INSERT INTO ARCHACC (accnum, balance)
    SELECT deleted.accnum, deleted.balance
    FROM deleted
END;
```

***deleted*** *is the old value of the record*

***inserted*** *is the new value of the record*

**Trigger tr2:**

```
CREATE TRIGGER tr2 ON ACCOUNT AFTER UPDATE AS
BEGIN
    INSERT INTO AUDIT (accnum, old_balance, new_balance, atime)
    SELECT deleted.accnum, deleted.balance, inserted.balance, getdate()
    FROM  deleted, inserted
END;
```

# Triggers

Consider the schema from earlier:

**EMP** (enum, ename, salary, dnum*)
**DEPT** (dnum, dname, numemp)

Where: *numemp* is number of employees

*numemp* should be updated **automatically** whenever a new employee is inserted in EMP

This can be achieved with Trigger **tr3:**

```
CREATE TRIGGER tr3 ON EMP AFTER INSERT
AS
BEGIN
        UPDATE DEPT
        SET numemp = (SELECT count(*)
                              FROM EMP
                              WHERE EMP.dnum = DEPT.dnum)
END;
```

**Exercise:**

Write a Trigger to update *numemp* whenever an employee leaves.

# Triggers

**INSTEAD OF Triggers**

Allows examination of changes before they happen

As with AFTER Triggers, two virtual tables are used - Inserted and Deleted

Except - Deleted holds all rows that would have been deleted
Inserted holds all rows that would have been inserted

**Example of an INSTEAD OF Trigger**

*Update all salaries by 10% unless they are above £50,000*

```
CREATE TRIGGER tr4 ON EMP INSTEAD OF UPDATE
AS BEGIN
    BEGIN
        IF (Salary > 50000) ROLLBACK;
    END
ELSE
    BEGIN
        UPDATE EMP
        SET Salary = Salary * 1.1;
        COMMIT;
    END;
END;
```

*We will see **ROLLBACK** and **COMMIT** later in the semester when we look at Transaction Management*