Stored Procedures and Functions

Objectives

- Batches and Routines
- Block Statements
- Exception Handling
- Stored Procedures
- Functions
- User Defined Functions

Batches and Routines

- Batch
  - a sequence of Transact-SQL statements and procedural extensions
  - Can increase performance benefits
  - Can be stored as:
    - a database object
    - A stored procedure
    - A user defined function
- Routine – A Stored Procedure or User Defined Function

Batch Restrictions

- The following statements must be the only statement in a batch
  - CREATE VIEW
  - CREATE PROCEDURE
  - CREATE TRIGGER
- GO

Block Statements

- One or more T-SQL Statements
- Every block begins with a BEGIN
- Every block ends with END
- Can be used inside the IF statement

```
BEGIN
statement_1
statement_2
…
END
```

Example

```
USE sample;
IF (SELECT COUNT(*)
    FROM works_on
    WHERE project_no = 'p1'
    GROUP BY project_no ) > 3
     PRINT 'The number of employees in the project p1 is 4 or more'
ELSE
     BEGIN
       PRINT 'The following employees work for the project p1'
       SELECT emp_fname, emp_lname
       FROM employee, works_on
       WHERE employee.emp_no = works_on.emp_no AND project_no = 'p1'
     END
```

While Statement

- Executes repeatedly while Boolean expression evaluates to true

```
WHILE (SELECT SUM(budget)
                    FROM project) < 500000
            BEGIN
              UPDATE project SET budget = budget*1.1
              IF (SELECT MAX(budget)
                        FROM project) > 240000
                BREAK
             ELSE CONTINUE
            END
```

Local Variables

- Store Values
- Referenced only within the same batch declared
- Defined using DECLARE
- Set
  - Using the special form of the SELECT statement
  - Using the SET statement
  - Directly in the DECLARE statement using the = sign (for instance, @extra_budget MONEY = 1500)

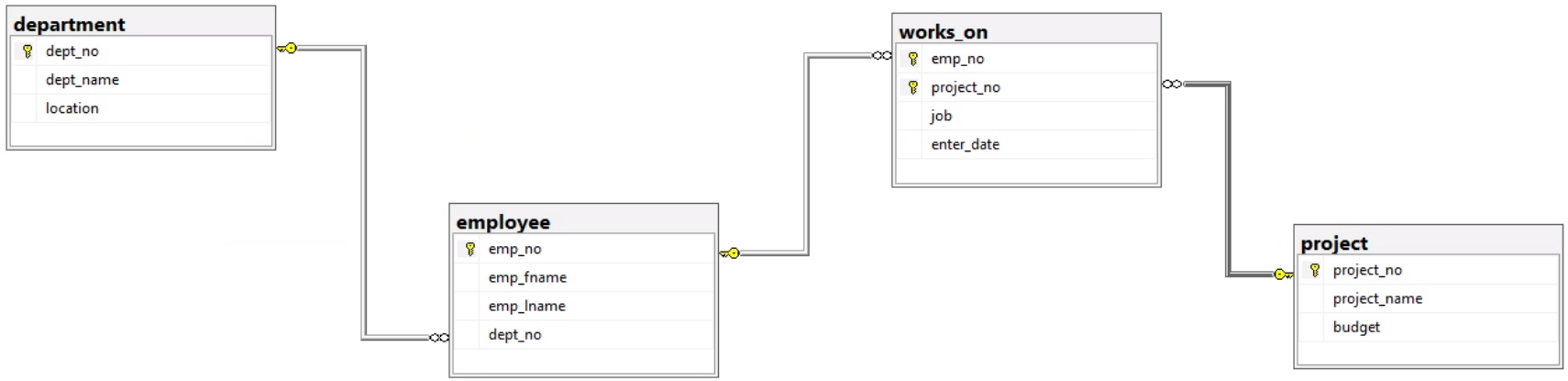
```
DECLARE @avg_budget MONEY, @extra_budget MONEY
SET @extra_budget = 15000
SELECT @avg_budget = AVG(budget) FROM project
 IF (SELECT budget
     FROM project
     WHERE project_no='p1') < @avg_budget
           BEGIN
             UPDATE project
                    SET budget = budget + @extra_budget
                     WHERE project_no ='p1'
              PRINT 'Budget for p1 increased by @extra_budget'
           END
  ELSE PRINT 'Budget for p1 unchanged'
```

Miscellaneous Procedural Statements

- RETURN
  - Causes execution of batch to stop
- GOTO
  - Branches to a label
- RAISERROR()
  - Generates a user defined error
- WAITFOR
  - Defines a time interval

```
DECLARE @Counter int;
SET @Counter = 1;
WHILE @Counter < 10
BEGIN
    SELECT @Counter
    SET @Counter = @Counter + 1
    IF @Counter = 4 GOTO Branch_One --Jumps to the first branch.
    IF @Counter = 5 GOTO Branch_Two  --This will never execute.
END
Branch_One:
    SELECT 'Jumping To Branch One.'
    GOTO Branch_Three; --This will prevent Branch_Two from executing.
Branch_Two:
    SELECT 'Jumping To Branch Two.'
Branch_Three:
    SELECT 'Jumping To Branch Three.';
```

Sample Database



Exception Handling

- TRY
- CATCH
- THROW
  - Like RAISERROR

Example 8.4

```
USE sample;
BEGIN TRY
   BEGIN TRANSACTION
   insert into employee values(11111, 'Ann', 'Smith','d2');
   insert into employee values(22222, 'Matthew', 'Jones','d4'); --
referential integrity error
   insert into employee values(33333, 'John', 'Barrimore', 'd2');
   COMMIT TRANSACTION
   PRINT 'Transaction committed'
END TRY
BEGIN CATCH
  ROLLBACK
  PRINT 'Transaction rolled back';
  THROW
END CATCH
```

Stored Procedures

- Stored as a database object
- Usually don’t return values
- Created using DDL
- Precompiled
- Group of one or more SQL statements
- Normally called from a remote program
- Can take input parameters and can also return values to the calling program
- Can help protect against SQL Injection

Stored Procedure Benefits

- Network Efficiency
- Encapsulate Business Logic
- Maintainable
- Stronger Security

Stored Procedures

```
CREATE PROC[EDURE] [schema_name.]proc_name
[({@param1} type1 [ VARYING] [= default1] [OUTPUT])] {, …}
[WITH {RECOMPILE | ENCRYPTION | EXECUTE AS 'user_name'}]
[FOR REPLICATION]
AS batch | EXTERNAL NAME method_name
```

```
CREATE PROCEDURE increase_budget
@percent INT=5
AS
UPDATE project
SET budget = budget + budget*@percent/100;
```

Run a Stored Procedure

- Execute or Exec

```
[[EXEC[UTE]] [@return_status =] {proc_name
        | @proc_name_var}
        {[[@parameter1 =] value | [@parameter1=] @variable [OUTPUT]]
DEFAULT}..
        [WITH RECOMPILE]
```

```
EXECUTE increase_budget 10;
```

```
CREATE PROCEDURE increase_budget
@percent INT=5
AS
UPDATE project
SET budget = budget + budget*@percent/100;
```

WITH RESULT SETS Clause

- Change the form of the result set of a stored procedure

```
CREATE PROCEDURE employees_in_dept
@deptname CHAR(10)
AS
SELECT emp_no, emp_lname
FROM employee
WHERE dept_no IN (SELECT dept_no FROM department
                  where dept_name = @deptname)

EXEC employees_in_dept 'Accounting'
With Result Sets
(([Employee Number] int not null,
  [Name of Employee] Char(20) not null))

Select * from employee
select * from department
```

Results | Messages

Employee Number	Name of Employee
2582	Bertoni
9031	Hansel
29346	James

Modifying a Procedure

- ALTER PROCEDURE
  - Existing permissions don’t need to be reassigned
- DROP PROCEDURE

```
CREATE PROCEDURE Emp_Info
AS
SELECT emp_lname, emp_fname
FROM Employee

exec emp_info

go
ALTER PROCEDURE Emp_Info
as
SELECT emp_lname, emp_fname , dept_no
FROM Employee

DROP PROCEDURE Emp_Info
```

User Defined Functions

- Always have one return value
- Scalar
  - Single value returned
- Table-Valued
  - Returns data of a table type

```
CREATE FUNCTION [schema_name.]function_name
        [({@param } type [= default])  {,...}
        RETURNS {scalar_type | [@variable] TABLE}
        [WITH {ENCRYPTION | SCHEMABINDING}
        [AS] {block | RETURN (select_statement)}
```

Syntax

Allowable Statements

- Assignment statements such as SET
- Control-of-flow statements such as WHILE and IF
- DECLARE statements defining local data variables
- SELECT statements containing SELECT lists with expressions that assign to variables that are local to the function
- INSERT, UPDATE, and DELETE statements modifying variables of the TABLE data type that are local to the function

Scalar

```
CREATE FUNCTION compute_costs (@percent INT =10)
       RETURNS DECIMAL(16,2)
    BEGIN
     DECLARE @additional_costs DEC (14,2), @sum_budget dec(16,2)
    SELECT @sum_budget = SUM (budget) FROM project
    SET @additional_costs = @sum_budget * @percent/100
    RETURN @additional_costs
   END
--Use the function
go
USE sample;
SELECT project_no, project_name
      FROM project
     WHERE budget < dbo.compute_costs(25)
USE sample;
GO
```

User Defined Functions

- Scalar
  - Performance issue
  - Does not allow for parallel execution of rows
  - SQL Server versions prior to 2019
- Scalar UDF inlining
  - Improves performance
  - Allows certain scalar UDFs to have their definition placed directly into the query
    - Query doesn't call the UDF for each row (Chap 28)
    - SQL Server 2019

Table Valued Functions

```
--table valued function
CREATE FUNCTION employees_in_project (@pr_number CHAR(4))
    RETURNS TABLE
    AS RETURN (SELECT emp_fname, emp_lname
               FROM works_on, employee
               WHERE employee.emp_no = works_on.emp_no
                      AND project_no = @pr_number
                )
Go
 --Use function
 USE sample;
SELECT *
FROM employees_in_project('p3')
```

CROSS and OUTER

- CROSS APPLY
  - Same as INNER JOIN
  - Microsoft extension of SQL standard
  - Better performance
- OUTER APPLY
  - Equivalent to LEFT OUTER JOIN

```
-- generate function
CREATE FUNCTION dbo.fn_getjob(@empid AS INT)
       RETURNS TABLE AS
RETURN
       SELECT job
          FROM works_on
          WHERE emp_no = @empid
       AND job IS NOT NULL  AND project_no = 'p1';
```

```
-- use CROSS APPLY
SELECT E.emp_no, emp_fname, emp_lname, job
 FROM employee as E
   CROSS APPLY dbo.fn_getjob(E.emp_no) AS A
-- use OUTER APPLY
SELECT E.emp_no, emp_fname, emp_lname, job
 FROM employee as E
   OUTER APPLY dbo.fn_getjob(E.emp_no) AS A
```

Table Valued Parameters

- Need to send in several parameters

```
--Creating a user defined data type

CREATE TYPE departmentType AS TABLE
  (dept_no CHAR(4),dept_name CHAR(25),location CHAR(30));
GO
--Creating a temporary table
CREATE TABLE #dallasTable
  (dept_no CHAR(4),dept_name CHAR(25),location CHAR(30));
GO
--Creating a procedure
CREATE PROCEDURE insertProc
  @Dallas departmentType READONLY
  AS SET NOCOUNT ON
  INSERT INTO #dallasTable (dept_no, dept_name, location)
   SELECT * FROM @Dallas
GO
--inserting data into the temporary table
DECLARE @Dallas AS departmentType;
INSERT INTO @Dallas( dept_no, dept_name, location)
SELECT * FROM department
WHERE location = 'Dallas'

--run procedure
EXEC insertProc @Dallas;
```

Modifying or Removing a UDF

- ALTER FUNCTION
  - Usually to remove schema binding
- DROP FUNCTION