# Stored Procedures and Functions

Rose-Hulman Institute of Technology
Curt Clifton

# Outline

- Stored Procedures or “Sprocs”
- Functions
- Statements Reference

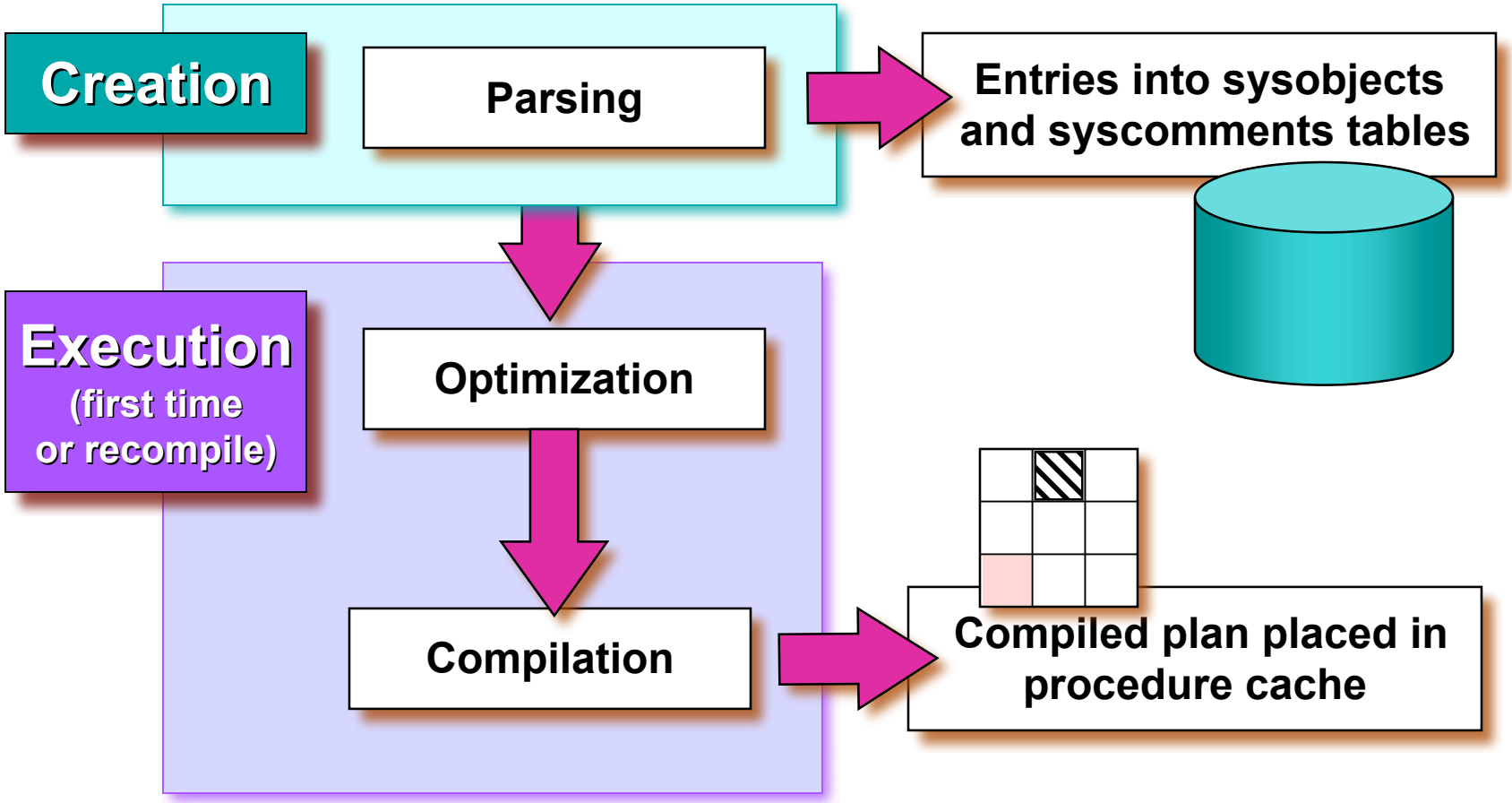
# Defining Stored Procedures

- Named Collections of Transact-SQL Statements
- Accept Input Parameters and Return Values
- Return Status Value to Indicate Success or Failure
- Encapsulate Repetitive Tasks

# Advantages of Stored Procedures

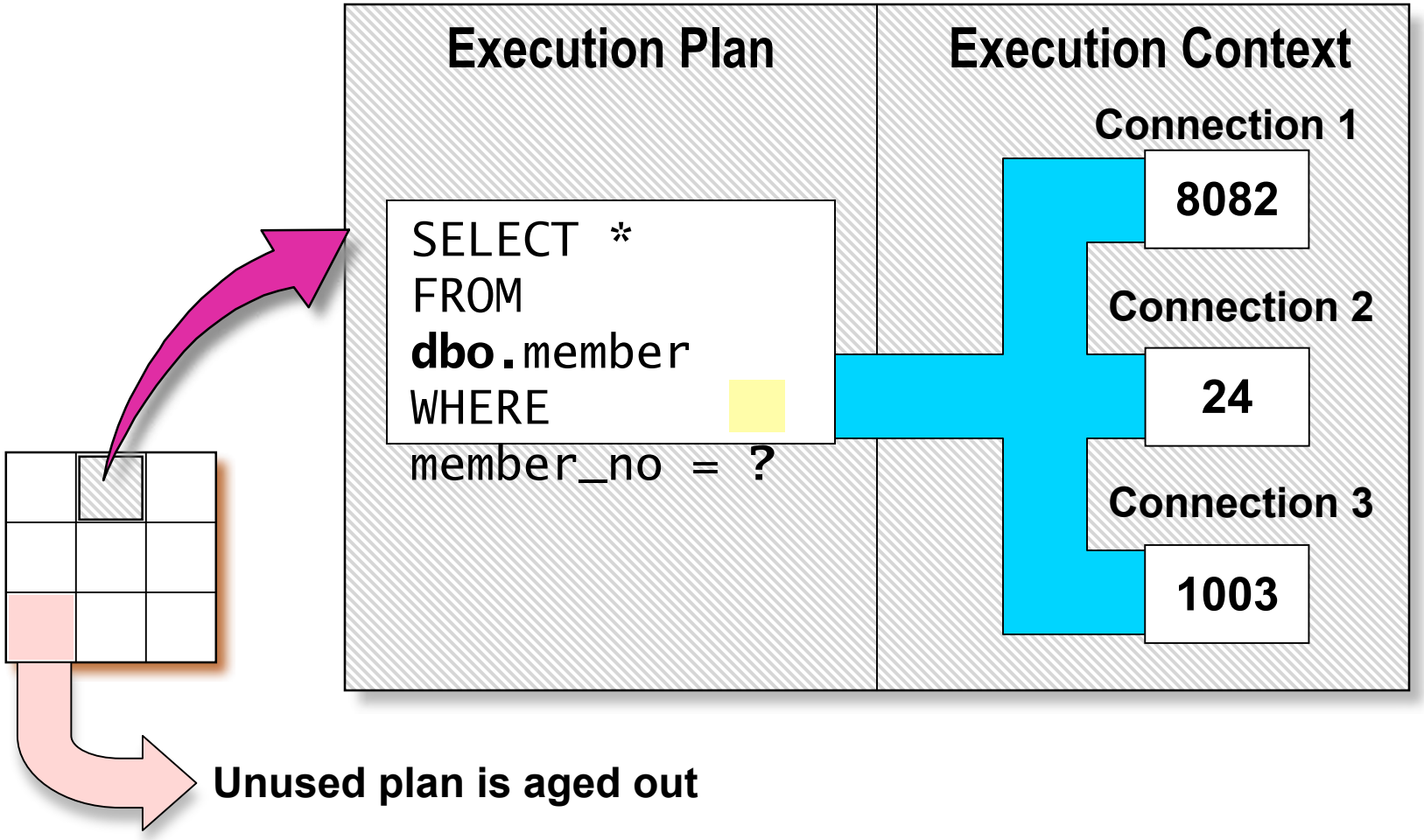
- Share Application Logic
- Shield Database Schema Details
- Provide Security Mechanisms
- Improve Performance
- Reduce Network Traffic

# Initial Processing of Sprocs



# Subsequent Processing of Sprocs

## Execution Plan Retrieved



# Creating Stored Procedures

- Create in Current Database Using the CREATE PROCEDURE (or CREATE PROC) Statement

```
USE Northwind
GO
CREATE PROC dbo.OverdueOrders
AS
  SELECT *
   FROM dbo.Orders
   WHERE RequiredDate < GETDATE() AND ShippedDate IS Null
GO
```

- Can Make Recursive Calls (but stack is limited)
- Use sp_help to Display Information
  - sp_help <procedure name>

# Executing Stored Procedures

- Executing a Stored Procedure by Itself

```
EXEC OverdueOrders
```

- Executing a Stored Procedure Within an INSERT Statement

```
INSERT INTO Customers
EXEC EmployeeCustomer
```

# Guidelines for Creating Sprocs

- dbo User Should Own All Stored Procedures
  - E.g., dbo.OverdueOrders
- One Stored Procedure for Each Task!
- One Task for Each Stored Procedure!
- Create, Test, and Troubleshoot
- Avoid sp_ Prefix in Stored Procedure Names
  - Used for system store procedures

# Altering and Dropping Sprocs

- Altering Stored Procedures

```
USE Northwind
GO
ALTER PROC dbo.OverdueOrders
AS
SELECT CONVERT(char(8), RequiredDate, 1) RequiredDate,
  CONVERT(char(8), OrderDate, 1) OrderDate,
  OrderID, CustomerID, EmployeeID
  FROM Orders
WHERE RequiredDate < GETDATE() AND ShippedDate IS Null
ORDER BY RequiredDate
GO
```

- Check dependencies:
  sp_depends dbo.OverdueOrders
- Dropping sprocs: DROP dbo.OverdueOrders

# Using Parameters in Sprocs

- Using Input Parameters
- Executing Using Input Parameters
- Returning Values Using Output Parameters

# Using Input Parameters

- Validate All Incoming Parameter Values First
- Provide Default Values or Null Checks

```
CREATE PROCEDURE dbo.[Year to Year Sales]
  @BeginningDate DateTime, @EndingDate DateTime
AS
IF @BeginningDate IS NULL OR @EndingDate IS NULL
BEGIN
   RAISERROR('NULL values are not allowed', 14, 1)
   RETURN
END
SELECT O.ShippedDate,
       O.OrderID,
       OS.Subtotal,
       DATENAME(yy,ShippedDate) AS Year
FROM ORDERS O INNER JOIN [Order Subtotals] OS
   ON O.OrderID = OS.OrderID
WHERE O.ShippedDate BETWEEN @BeginningDate AND @EndingDate
GO
```

# Executing Sprocs with Parms

- By name:

```
EXEC AddCustomer
    @CustomerID = 'ALFKI',
    @ContactName = 'Maria Anders',
    @CompanyName = 'Alfreds Futterkiste',
    @ContactTitle = 'Sales Representative',
    @Address = 'Obere Str. 57',
    @City = 'Berlin',
    @PostalCode = '12209',
    @Country = 'Germany',
    @Phone = '030-0074321'
```

- By position:

```
EXEC AddCustomer 'ALFKI2', 'Alfreds
Futterkiste', 'Maria Anders', 'Sales
Representative', 'Obere Str. 57',
'Berlin', NULL, '12209', 'Germany',
'030-0074321'
```

# Returning Values: Output Parms

Creating Stored Procedure

```
CREATE PROCEDURE dbo.MathTutor
    @m1 smallint,
    @m2 smallint,
    @result smallint OUTPUT
AS
    SET @result = @m1 * @m2
GO
```

Executing Stored Procedure

```
DECLARE @answer smallint
EXECUTE MathTutor 5,6, @answer OUTPUT
SELECT 'The result is: ', @answer
```

Results of Stored Procedure

```
The result is:  30
```

# Handling Error Messages

- RETURN Statement Exits Query or Procedure Unconditionally
- sp_addmessage Creates Custom Error Messages
- @@error Contains Error Number for Last Executed Statement
- RAISERROR Statement
  - Returns user-defined or system error message
  - Sets system flag to record error

# What Is a User-defined Function?

- Scalar Functions (do not reference tables)
  - Similar to a built-in function
- Multi-Statement Table-valued Functions
  - Content like a stored procedure
  - Referenced like a view
- In-Line Table-valued Functions
  - Similar to a view with parameters
  - Returns a table as the result of single SELECT statement

# Creating a User-defined Function

```
USE Northwind
CREATE FUNCTION fn_NonNull
   (@myinput nvarchar(30))
   RETURNS nvarchar(30)
BEGIN
   IF @myinput IS NULL
   SET @myinput = 'Not Applicable'
   RETURN @myinput
END
```

# Altering and Dropping Functions

- Altering Functions

```
ALTER FUNCTION dbo.fn_NewRegion
   <New function content>
```

  - Retains assigned permissions
  - Causes the new function definition to replace existing definition

- Dropping Functions

```
DROP FUNCTION dbo.fn_NewRegion
```

# Three Examples of Functions

- Scalar User-defined Function
- Multi-Statement Table-valued Function
- In-Line Table-valued Function

# Scalar User-defined Function

- RETURNS Clause Specifies Data Type
- Function Is Defined Within a BEGIN and END Block
- Return Type Is Any Data Type Except text, ntext, image, cursor, or timestamp

# Example

- Creating a function:

```
USE Northwind
CREATE FUNCTION fn_DateFormat
      (@indate datetime, @separator char(1))
   RETURNS Nchar(20)
   AS
   BEGIN
      RETURN
      CONVERT(Nvarchar(20), datepart(mm,@indate))
      + @separator
      + CONVERT(Nvarchar(20), datepart(dd, @indate))
      + @separator
      + CONVERT(Nvarchar(20), datepart(yy, @indate))
   END
```

- Calling the function:

```
SELECT dbo.fn_DateFormat(GETDATE(), ':')
```

# Multi-Statement Table-valued Fn.

- BEGIN and END Enclose Multiple Statements
- RETURNS Clause Specifies **table** Data Type
- RETURNS Clause Names and Defines the Table

# Example

- Creating the Function

```
USE Northwind
GO
CREATE FUNCTION fn_Employees (@length nvarchar(9))
RETURNS @fn_Employees table
   (EmployeeID int PRIMARY KEY NOT NULL,
   [Employee Name] nvarchar(61) NOT NULL)
AS
BEGIN
   IF @length = 'ShortName'
      INSERT @fn_Employees SELECT EmployeeID, LastName
      FROM Employees
   ELSE IF @length = 'LongName'
      INSERT @fn_Employees SELECT EmployeeID,
      (FirstName + ' ' + LastName) FROM Employees
RETURN
END
```

- Calling it:

```
SELECT * FROM dbo.fn_Employees('LongName')
Or
SELECT * FROM dbo.fn_Employees('ShortName')
```

# In-Line Table-valued Function

- Content of the Function Is a SELECT Statement
- Do Not Use BEGIN and END
- RETURN Specifies table as the Data Type
- Format Is Defined by the Result Set

# Example

- Creating the Function

```
USE Northwind
GO
CREATE FUNCTION fn_CustomerNamesInRegion
   ( @RegionParameter nvarchar(30) )
RETURNS table
AS
RETURN (
   SELECT CustomerID, CompanyName
   FROM Northwind.dbo.Customers
   WHERE Region = @RegionParameter
   )
```

- Calling it:

```
SELECT * FROM fn_CustomerNamesInRegion('WA')
```

# Types of Statements (1/2)

- RETURN <expression>
- DECLARE <name> <type>
  - used to declare local variables
- BEGIN . . . END
  - Coalesce groups of statements
  - Separate by semicolons
  - Like { … } in Java, C, …

# Types of Statements (2/2)

- SET <variable> = <expression>;
  - Assignment
- SELECT <var1> = <expr1>, <var2> = <expr2> …
  - Multi-variable assignment
- IF <expr> <statement> [ELSE <statement>]
- WHILE <expr> <statement>

# Example: Assignment/Query

- When right-hand side is single value:

```
SET @p = (SELECT price FROM Sells
          WHERE    rest = 'Joe''s'
          AND      soda = 'Pepsi');
```

# Multi-variable Assignment

- Example:
  - SELECT @ph = phone, @addr = addr
    FROM Customer
    WHERE name = 'Rumi'

# IF statements

- Basic form:
  - IF <condition>
    <statement>
- Need BEGIN … END for multi-statement body:
  - IF <condition>
    BEGIN
        <statement>;
        <statement>;
    END
- Can use ELSE if needed:
  - IF <condition> <statement> ELSE <statement>

# WHILE Loops

- Syntax: WHILE <condition> <statement>
- Again, use BEGIN … END for longer body
- Can be like "normal" while loops:
  - 
```
WHILE (SELECT avg(price) FROM Sells) < 3
BEGIN
        UPDATE Sells
        SET price = price * 1.05
        IF (SELECT max(price) FROM Sells) > 5
                BREAK
END
```
- Or use "cursors" to loop through query results

# Cursor Example

```
DECLARE @name nvarchar(10); DECLARE @result int;
DECLARE NameCursor CURSOR LOCAL FOR
    SELECT LTRIM(RTRIM(username)) FROM [dbo].[Students]
OPEN NameCursor
FETCH NEXT FROM NameCursor INTO @name
WHILE @@FETCH_STATUS = 0
BEGIN
    if ( 0 = (select count(*) from master.sys.syslogins where loginname=@name) )
    BEGIN
        EXEC('CREATE LOGIN ' + @name + ' WITH PASSWORD='' + @name + '')
        exec sp_addsrvrolemember @loginame=@name, @rolename='dbcreator'
        set @result = @result + 1
    END
    else
        select (@name + ' Already There') as result
    FETCH NEXT FROM NameCursor INTO @name
END
```

# Remember CRUD!

- Want interface code to easily manipulate data
- Define stored procedures to:
  - Create new entries in tables
  - Retrieve data from tables
  - Update entries in tables
  - Delete entries from tables