Lecture 8: Stored Procedures

* Supported by most RDBMS
* Pieces of executable code stored in the database (small programs)
* Managed by DBMS

**Stored Procedure Examples**

* Archive last year’s data
* Remove customers data when the customer hasn’t bought anything for more than 3 years
* Implement a price change across a category of products

**Stored Procedures**

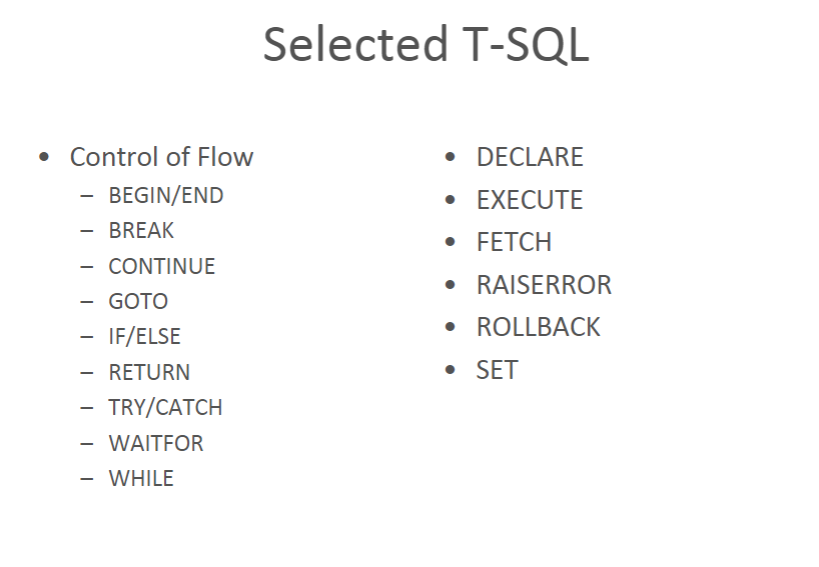
* Automate and standardize processes
* Compiled, therefore faster
* Usually written in Transact-SQL (T-SQL)
* Can also be written in a Common Language
* Runtime (CLR) programming languages: C#, Visual Basic.NET, ...

**Preferred Database Access Method**

* C#, VB.NET, Java, ... can form SQL and execute it
  + This is not recommended
* Best practice is for programming languages to call stored procedures for database access
  + Executes faster
  + Reduces network traffic
  + More secure
  + Consistent
  + DBA approved

**Stored Procedure Body Statements**

* SQL is non-procedural and set oriented
* Stored procedures require procedural code
* Each database vendor extends SQL with procedural code
  + Microsoft uses T-SQL
  + Oracle uses Procedural Language SQL (PL/SQL)
  + IBM uses SQL Procedural Language (SQL PL)
  + MySQL uses SQL/Persistent Stored Module (SQL/PSM)



**Stored Procedures**

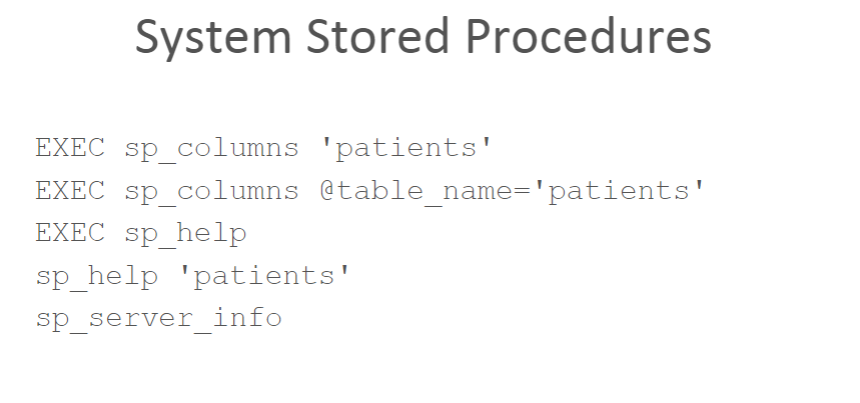
* Invoked by EXECUTE (EXEC) statement in T-SQL

**SQL**

* EXEC is optional, but a good idea
  + It allows autocomplete to help
* Can accept command line parameters, separated by commas
* Parameters can be positional or named

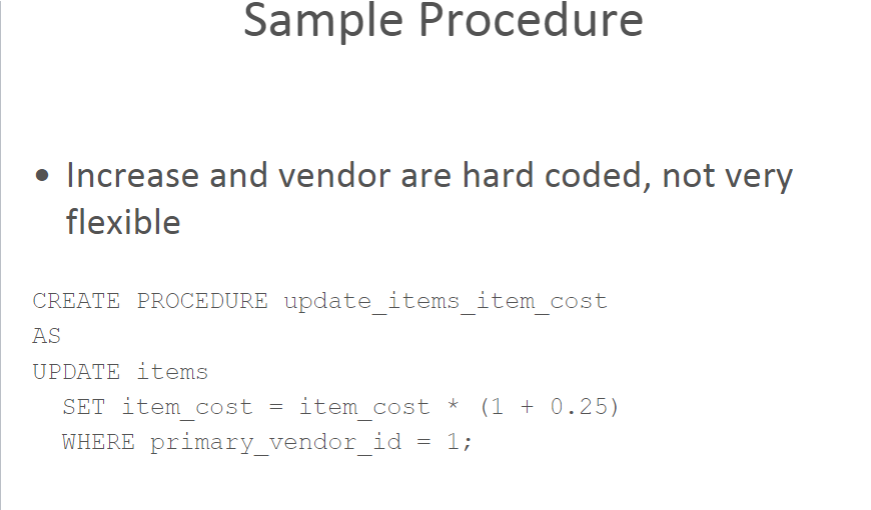
**System Stored Procedures**

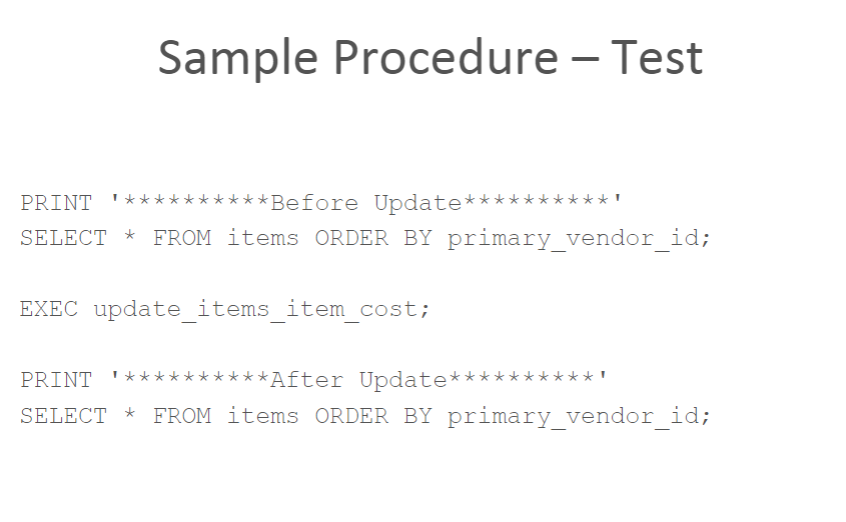
* SQL Server provides many stored procedures
* Many are used for database administration
* Typing the name of a system stored procedure in a query window and pressing F1 will get help on the procedure
* Right clicking the procedure in the Object Explorer and selecting Modify will reveal the procedure code
* sp\_columns, sp\_help, sp\_server\_info, ...

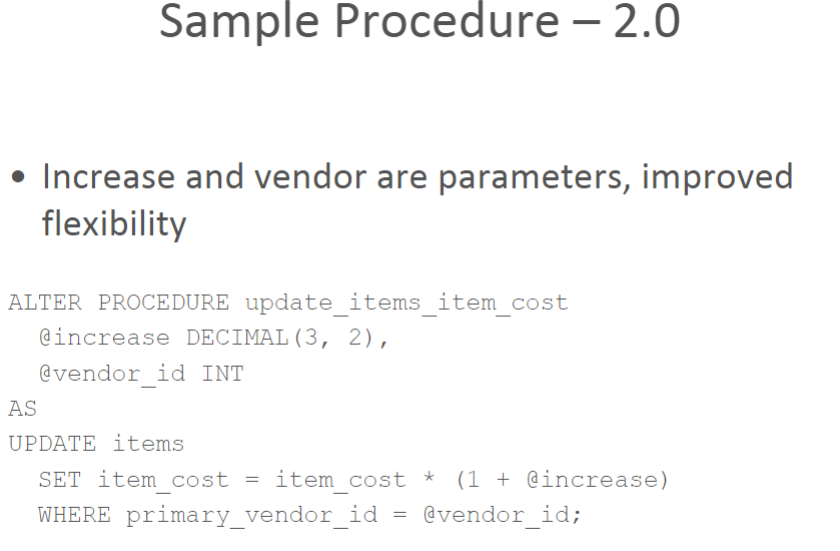


**Stored Procedure Structure**

* A stored procedure consists of:
  + Procedure name
  + Set of optional parameters
  + Routine body

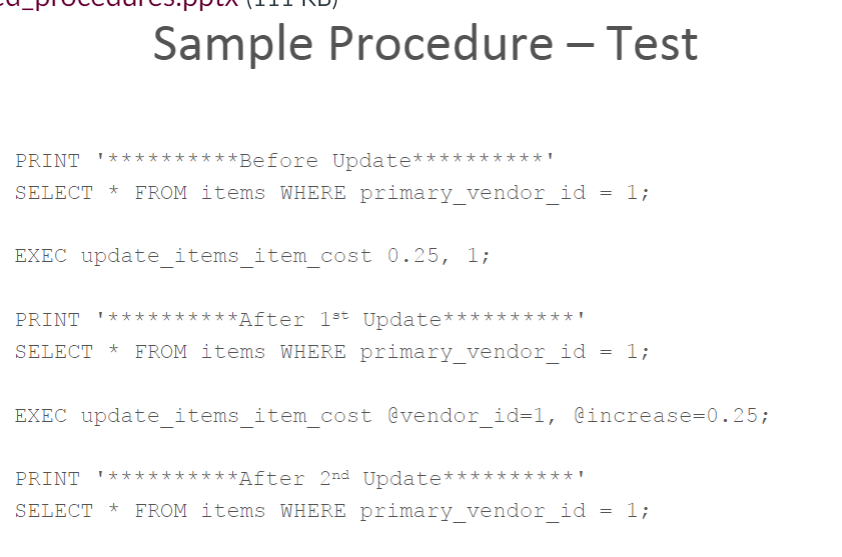


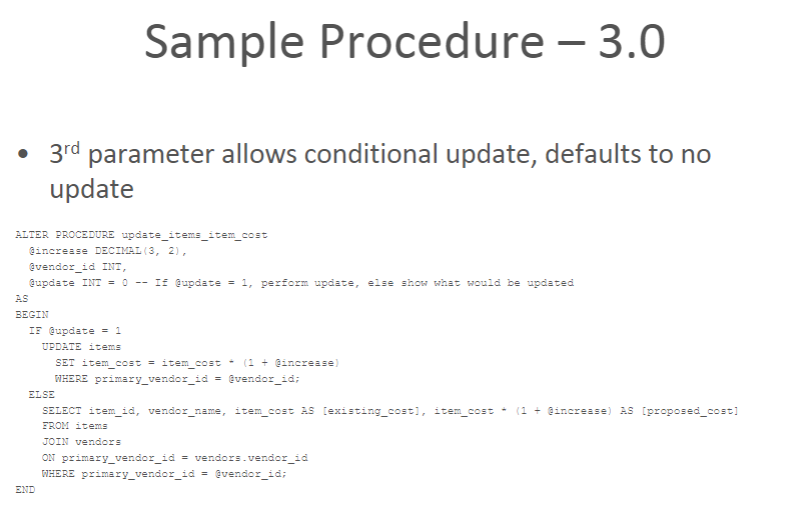


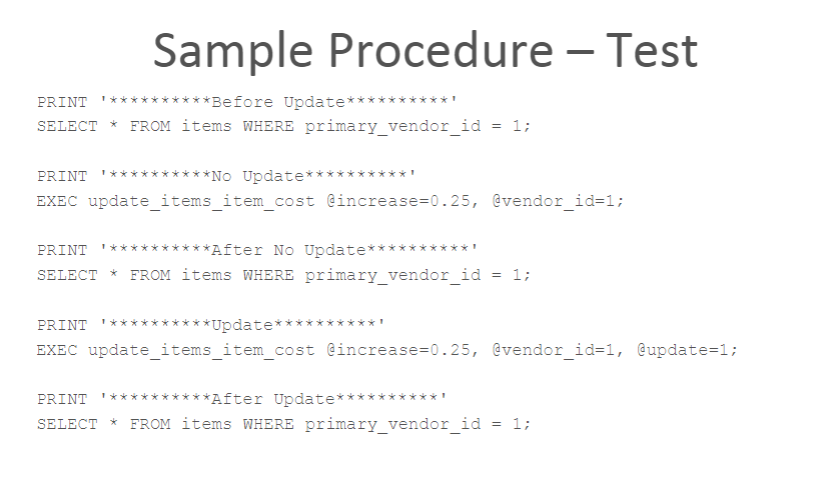


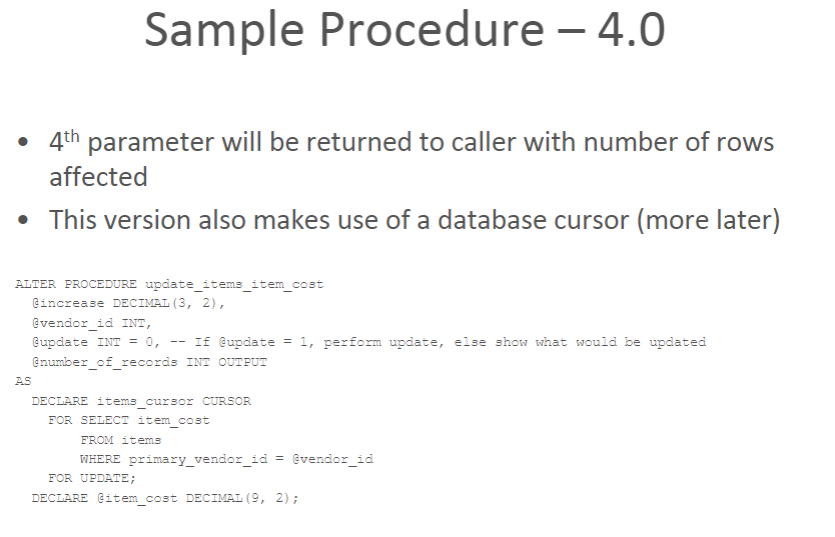
**ALTER SQL Statement**

* ALTER PROCEDURE allows changing an existing stored procedure
* ALTER TABLE allows for changing table structure
  + Adding/Removing/Changing of columns, constraints, ...
* ALTER DATABASE, ALTER VIEW, ...









Sample Procedure – 4.0

BEGIN

IF @update = 1

BEGIN

SET @number\_of\_records = 0;

OPEN items\_cursor;

FETCH NEXT FROM items\_cursor

INTO @item\_cost;

WHILE @@FETCH\_STATUS = 0

BEGIN

SET @item\_cost = @item\_cost \* (1 + @increase);

UPDATE items

SET item\_cost = @item\_cost

WHERE CURRENT OF items\_cursor;

SET @number\_of\_records = @number\_of\_records + 1;

FETCH NEXT FROM items\_cursor

INTO @item\_cost;

END

CLOSE items\_cursor;

END

ELSE -- No Update

SELECT item\_id, vendor\_name, item\_cost AS [existing\_cost], item\_cost \* (1 + @increase) AS [proposed\_cost]

FROM items

JOIN vendors

ON primary\_vendor\_id = vendors.vendor\_id

WHERE primary\_vendor\_id = @vendor\_id;

DEALLOCATE items\_cursor;

END

Sample Procedure – Test

PRINT '

\*\*\*\*\*\*\*\*\*\*Before

Update

\*\*\*\*\*\*\*\*\*\*

'

SELECT \* FROM items WHERE primary\_vendor\_id = 1;

-- Need to declare a variable to receive output

DECLARE @num\_of\_rows INT;

PRINT '

\*\*\*\*\*\*\*\*\*\*

No Update

\*\*\*\*\*\*\*\*\*\*

'

EXEC update\_items\_item\_cost @increase=0.25, @vendor\_id=1, @update=0, @number\_of\_records=@num\_of\_rows OUTPUT;

SELECT @num\_of\_rows AS [number\_of\_rows];

PRINT '

\*\*\*\*\*\*\*\*\*\*After

No Update

\*\*\*\*\*\*\*\*\*\*

'

SELECT \* FROM items WHERE primary\_vendor\_id = 1;

PRINT '

\*\*\*\*\*\*\*\*\*\*

Update

\*\*\*\*\*\*\*\*\*\*

'

EXEC update\_items\_item\_cost @increase=0.25, @vendor\_id=1, @update=1, @number\_of\_records=@num\_of\_rows OUTPUT;

SELECT @num\_of\_rows AS [number\_of\_rows];

PRINT '

\*\*\*\*\*\*\*\*\*\*After

Update

\*\*\*\*\*\*\*\*\*\*

'

SELECT \* FROM items WHERE primary\_vendor\_id = 1;

**Variable Declarations**

* Declare local variables used for
  + Calculations
  + Assignment to output parameters
  + Assignment to columns for database updates
  + As input parameters passed by calling programs
  + Error handling

**Assignment Statement SET**

* Used to assign values to
  + Input and output parameters
  + Local variables
* Conform to SQL arithmetic operators
* Compatible data types of target and source

SET @record\_count = @record\_count + 1;

SET @credit\_limit = @credit\_limit \* 1.20;

SET @num\_orders = NULL;

SET @max\_credit\_limit = (SELECT MAX(credit\_limit) FROM

customers);

**Conditional Control**

* IF
* IF-ELSE

IF @ref\_error = 1

SET @error\_msg = 'NOT FOUND';

**IF**

* Tests a simple condition
* If the condition evaluates to TRUE, the next line of code only is executed
* If the condition evaluates to FALSE, the control of the program is passed to the next statement after the test

IF @ref\_error = 1

SET @error\_msg = 'NOT FOUND';

**IF-ELSE**

* Similar to the IF structure
* The difference is that when the condition evaluates to FALSE, the statement following the ELSE keyword is executed

IF @ref\_error = 0

SET @error\_msg = 'FOUND';

ELSE

SET @error\_msg = 'NOT FOUND';

**BEGIN-END**

* Used to enclose a block of statements where a single statement can be used

IF @ref\_error = 0

BEGIN

SET @error\_msg = 'FOUND';

SET @found = @found + 1;

END

ELSE

BEGIN

SET @error\_msg = 'NOT FOUND';

SET @not\_found = @not\_found + 1;

END

**BEGIN-END**

* Think of BEGIN-END as a more verbose version of curly braces

IF @ref\_error = 0

{

SET @error\_msg = 'FOUND';

SET @found = @found + 1;

}

ELSE

{

SET @error\_msg = 'NOT FOUND';

SET @not\_found = @not\_found + 1;

}

**Nested IFs**

IF @evaluation = 100

SET @new\_salary = salary \* 1.3;

ELSE

BEGIN

IF @evaluation >= 90

SET @new\_salary = salary \* 1.2;

ELSE

SET @new\_salary = salary \* 1.1;

END

**WHILE**

* Loop while the condition is true
* Used with BEGIN-END
* Frequently used with cursors

WHILE @@FETCH\_STATUS = 0

BEGIN

<loop processing>

END

**BREAK**

* Used to exit a WHILE loop

WHILE @@FETCH\_STATUS = 0

BEGIN

<loop processing>

IF @count = 5

BREAK;

END

**CONTINUE**

* Used to advance to the next iteration of a WHILE loop

WHILE @@FETCH\_STATUS = 0

BEGIN

<loop processing>

IF @price = 5.0

CONTINUE;

<loop processing>

END

**Deleting A Procedure**

* To delete an existing procedure use:

DROP PROCEDURE procedure\_name

**Data Dictionary Tables For Procedures**

* Every CREATE PROCEDURE statement generates entries in
  + sys.procedures
  + sys.sql\_modules

**Cursors**

* Used to access a SELECT result set one row at a time
* How to use a cursor
  + DECLARE a cursor with a SELECT statement
  + OPEN the cursor
    - This executes the SELECT statement and populates the cursor
  + FETCH one row at a time from the result set INTO variables
    - Each column fetched must have a correlating variable
    - Perform whatever processing desired for each row
* CLOSE the cursor
* DEALLOCATE the cursor

**@@FETCH\_STATUS**

* Returns the status of the last cursor FETCH statement
  + 0 = The FETCH statement was successful
  + -1 = The FETCH statement failed or the row was beyond the result set
  + -2 = The row fetched is missing