Database and Information Systems

Michal Krátký & Radim Bača

Department of Computer Science
Faculty of Electrical Engineering and Computer Science
VŠB – Technical University of Ostrava

2020/2021

Content

T-SQL

1. A procedural extension for the DBMS Sybase and MS SQL Server.
2. We use the SQL Server Management Studio as a client for writing and executing T-SQL code.
3. Code can be executed and debugged, however, debugging has limitations (you need to be the admin).
4. Instance: dbsys.cs.vsb.cz\student
   User name: $< login >$, Password: see email

Variables

- Local variables are valid only in the block where we define them.
- Local variables are defined by the keyword DECLARE and the name starts with the @ symbol.
- Data types can be user defined or system.
- A declaration of an int variable @CNT:[1]
  ```
  DECLARE @CNT INT
  ```
- We can declare more variables using one DECLARE statement, for example: `DECLARE @CNT INT, @X INT, @Y INT, @Z CHAR(10)`

[1] T-SQL is a prefix language, therefore, the expressions does not have to be ended by the ; symbol

Variable definition

- Each local variable is initialized to the NULL value by default.
- A variable can be defined by:
  - the SET command:

```
DECLARE @CNT INT
SET @CNT = 1
```

  - the SELECT command[2]

```
DECLARE @ROWCNT INT
SELECT @ROWCNT=COUNT(*) FROM authors
```

[2] We do not use the SELECT INTO like in PL/SQL.

Variable definition, example

```
DECLARE @Country varchar(25)
SET @Country = 'Germany'

SELECT CompanyName FROM Customers
  WHERE Country = @Country
```

Remarks

```
CREATE TABLE Product
-- A table named Product
(
...

CREATE TABLE Product
/* A table named Product
   with login as Primary Key */
(
...
```

Conditions, IF ... ELSE

```
IF <boolean condition> <statement>

IF <boolean condition> <statement>
ELSE <statement>
```

- If we want to process more than one statement then we have to encapsulate these statements by the **BEGIN** and **END** keywords.
- Example:

```
DECLARE @x INT
SET @x = 29
IF @x = 29 PRINT 'The number is 29'
IF @x = 30 PRINT 'The number is 30'
```

Condition, Example

- The SELECT command has to be in parenthesis

```
IF (select count(*) from Pubs.dbo.Authors where
     lname like '[A-D]%') > 0
   print 'Found A-D Authors'
```

It prints out a text when the table contains author starting by A, B, C, or D.

- This example prints out a text when the current database is named 'student'.

```
IF db_name() = 'student'
BEGIN
   print 'student is the current database'
END
```

Condition, Example

- A test whether the record exists or not.

```
IF exists(select * from Customers
          where CustomerId = 'kra28')
  print 'Use update for the record kra28'
ELSE
  print 'Use insert for the record kra28'
```

- A condition can contain any complex expression:

```
IF db_name() = 'student' and (select count(*)
     from sysobjects where name='Customers') = 1
  print 'Table Customers Exist'
ELSE
  print 'It is not the student database' +
     ' or Table Customer does not exist'
```

While cycle

- The while cycle has a condition at the begining
  WHILE <Boolean expression> <code block>
- Example:

```
DECLARE @counter int
SET @counter = 0
WHILE @counter < 10
BEGIN
   SET @counter = @counter + 1
    print 'The counter is ' +
          cast(@counter as char)
END
```

Notice: Type-casting is processed by the cast function.

While cycle, example 1/3

```
DECLARE @id int
DECLARE @categoryId int
DECLARE @desc varchar(50)

CREATE TABLE Product(id int, categoryId int,
   desc varchar(50))
```

While cycle, example 2/3

```
SET @id = 0
SET @categoryId = 0
WHILE @id < 2
BEGIN
 SET @id = @id + 1
 WHILE @categoryId < 3
 BEGIN
  SET @categoryId = @categoryId + 1
  SET @description = 'id is ' + cast(@id as char(1)) +
      ' categoryId ' + cast(@categoryId as char(1))
  INSERT INTO Product values(@id,
                             @categoryId,
                             @description)
 END
```

While cycle, example 3/3

```
  SET @categoryId = 0
END

SELECT * FROM Product
-- DROP TABLE Product
```

```
1 1 id is 1 categoryId 1
1 2 id is 1 categoryId 2
1 3 id is 1 categoryId 3
2 1 id is 2 categoryId 1
2 2 id is 2 categoryId 2
2 3 id is 2 categoryId 3
```

Transactions

- Transaction starts by
  BEGIN TRANSACTION <**transaction_name**>
- Transaction ends by **ROLLBACK** or **COMMIT**.
- Isolation level setting:
  SET TRANSACTION ISOLATION LEVEL <**level**>

Transaction, Example 1/2

```
DECLARE @v_upd1 INT
DECLARE @v_upd2 INT
BEGIN TRAN UpdateTransaction
UPDATE Product SET description='desc4' WHERE id=1
SET @v_upd1 = @@rowcount
UPDATE Product SET description='desc5' WHERE id=4
SET @v_upd2 = @@rowcount
-- error or noupdate
IF @@ERROR <> 0 OR @v_upd1 = 0 OR @v_upd2 = 0
BEGIN
   print 'rollback...'
   ROLLBACK
END
ELSE
BEGIN
   print 'commit...'
   COMMIT
END
```

Transaction, example 2/2

Output 1:

```
(11 row(s) affected)

(3 row(s) affected)
commit...
```

Output 2:

```
(11 row(s) affected)

(0 row(s) affected)
rollback...
```

Transaction and exceptions

```
BEGIN TRY
   BEGIN TRAN UpdateTransaction
   UPDATE Product SET description='desc4' WHERE id=1
   UPDATE Product SET description='desc5' WHERE id=4
    print 'commit...'
   COMMIT
END TRY
BEGIN CATCH
    print 'rollback...'
   ROLLBACK
END CATCH
```

Output:

```
(11 row(s) affected)
(0 row(s) affected)
commit...
```

Stored procedures

- A stored procedure is a compiled code stored in a database (it is the same in all procedural extensions of SQL).
- **Advantages:**
  - We reduce cost of the transfer across the network.
  - Stored procedures are cached (and compiled).
  - Logic expressed by the procedures can be easily shared across the applications written in different languages.

Stored procedures, syntax

```
CREATE [OR ALTER] PROC[EDURE] procedure_name [;number]
    [{ @parameter data_type }
        [VARYING] [=default] [OUTPUT]
    ][,...]
[ WITH
    {
        RECOMPILE
      | ENCRYPTION
      | RECOMPILE, ENCRYPTION
    }
]
[FOR REPLICATION]
AS
    sql_statement
```

Stored procedures, syntax

- `procedure_name` – procedure name.
- `number` – optional parameter allowing to group the procedures; they can be easily dropped by the `DROP PROCEDURE` command.
- `RECOMPILE` – SQL server do not cache the procedure with this parameter.

Stored procedures, syntax

Parameters of the procedure:

- `@parameter` – name of the parameter
- `data_type` – data type of the parameter. **The data type must be written with the length if necessary.**
- `VARYING` – specifies the set of the records as an output of a procedure (applies only to cursor variables).
- `default` – default value of the parameter.
- `OUTPUT` – it means that the parameter is output.

A procedure can have more than a one parameter.

Execution, syntax

```
[EXEC[UTE]]
{ procedure_name [;number] |
        @procedure_name_variable }
   [ [@parameter=] { value |
       @variable [OUTPUT] | [DEFAULT] ,... } ]
[WITH RECOMPILE]
```

- `procedure_name` – name of a procedure.
- `number` – optional parameter allowing to group the procedures
- `@procedure_name_variable` – a local variable which represents the stored procedure.

Execution, syntax

Parameters:

- `@parameter` – name of a procedure as it is defined in the `CREATE PROCEDURE` command.
- `value` – value of a procedure parameter; if the parameter names are not specified then the parameter has to be written in the defined order.
- `@variable` – variable containing the parameter value or the return value.
- `OUTPUT` – specify that the parameter is a return parameter.
- `DEFAULT` – means that the value is specified in the procedure.

Example

```
CREATE OR ALTER PROCEDURE spDisplayAll
AS
   SELECT * FROM Students
GO

EXEC spDisplayAll;
EXECUTE spDisplayAll;
```

Example

We pass just the values of parameters or specify their names as well. In the second case we can have any order of parameters.

```
CREATE OR ALTER PROCEDURE spSelectProduct
                                    (@id INTEGER)
AS
BEGIN
   SELECT * FROM Products WHERE id=@id
END
GO

EXEC spSelectProduct 1;
EXEC spSelectProduct @id=1;
```

Example

We can use implicit values of parameters.

```
CREATE OR ALTER PROCEDURE spSelectProduct
                                    (@id INTEGER = 1)
AS
BEGIN
   SELECT * FROM Products WHERE id=@id
END
GO

EXEC spSelectProduct;
EXEC spSelectProduct @id=2;
```

Drop Procedure

```
DROP PROCEDURE procedure_name, ...
```

where the `procedure_name` is a name of a stored procedure or a set of procedures.

Example:

```
DROP PROCEDURE spSelectProduct;
```

Altering the procedure

ALTER PROC[EDURE] procedure_name

- Syntax is the same with the `CREATE PROCEDURE` command.
- Since SQL Server 2016, we can use **CREATE OR ALTER**.

Renaming the procedure

`sp_rename 'procedure_name1', 'procedure_name2'`

Example:

```
EXEC sp_rename 'spSelectProduct', 'spSelectProduct_old';
```

Information about the procedures

- `sp_helptext procedure_name` – definition of a stored procedure
- `sp_help procedure_name` – information about a stored procedure
- `sp_depends procedure_name` – dependency of a stored procedure

```
sp_depends spDisplayAll;

name           type         updated selected  column
kra28.Product user table  no      yes       id
kra28.Product user table  no      yes       categoryId
kra28.Product user table  no      yes       description
```

Stored function, syntax

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name.]
        parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN scalar_expression
    END
[ ; ]
```

Stored functions, restrictions

- Functions in T-SQL has many restrictions: it is not possible to use TRY - CATCH, DML an so on (see `http://msdn.microsoft.com/en-us/library/ms191320%28SQL.90%29.aspx`)
- The solution is to use a procedure with output parameters.

Cursors

- They allows us to read the result of a query with many result rows.
- Syntax:

  DECLARE cursor_name **CURSOR FOR** select_statement

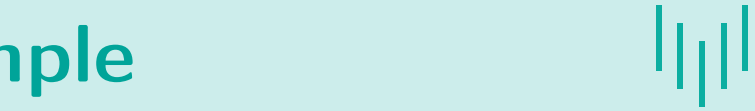
- We have to use: : **OPEN**, $2\times$ **FETCH**, **CLOSE**, **DEALLOCATE**
- `@@FETCH_STATUS` is set to non zero value if the result is empty.

Cursors, example 1

```
DECLARE @id nchar(5)
DECLARE @rowNum int
DECLARE productList CURSOR FOR select top 5 id from Product
OPEN productList
FETCH NEXT FROM productList INTO @id
SET @rowNum = 0
WHILE @@FETCH_STATUS = 0
BEGIN
   SET @rowNum = @rowNum + 1
   PRINT cast(@rowNum as char(1)) + ' ' + @id
   FETCH NEXT FROM productList INTO @id
END
CLOSE productList
DEALLOCATE productList
```

Notice: The query returns id of the first five records (it is not compatible with the relational data model and SQL specification).

Cursors, example 2, **negative example**



The same problem is possible to solve by the query returning just one record using the top 1 expression.

```
DECLARE @id nchar(5)
DECLARE @rowNum int
select top 1 @id=id from Product
SET @rowNum = 0
WHILE @rowNum < 5

BEGIN
   SET @rowNum = @rowNum + 1
   print cast(@rowNum as char(1)) + ' ' + @id
   select top 1 @id = id from Product where id > @id
END
```

However, it sends 5 queries instead of only one query.

Dynamic SQL

- Dynamic SQL is executed by the `sp_executesql`:

```
sp_executesql [@stmt =] stmt
[
 {, [@params =] N'@parameter_name data_type [,...n]' }
 {, [@param1 =] 'value1' [,...n] }
]
```

- Where:
  - `[@stmt=] stmt` is a dynamic T-SQL statement.
  - `[@params=] N'@parameter_name data_type [,...n]'` set of variables and data types for each dynamic variable.
  - `[@param1=] 'value1' [,...n]` values of variable of a dynamic T-SQL.

Dynamic SQL

The following dynamic T-SQL returns the number of records in a table according to the query.

```
DECLARE @RECCNT int
DECLARE @ORDID varchar(10)
DECLARE @CMD Nvarchar(100)
SET @ORDID = 10436
SET @CMD = 'SELECT @RECORDCNT=count(*) from [Orders]' +
           ' where OrderId < @ORDERID'
PRINT @CMD
EXEC sp_executesql @CMD,
                   N'@RECORDCNT int out, @ORDERID int',
                   @RECCNT out,
                   @ORDID
PRINT 'The number of records that have an OrderId' +
      ' greater than ' + @ORDID + ' is ' +
      cast(@RECCNT as char(5))
```

Trigger

Trigger is an action automatically invoked when a database operation occurs.

```
CREATE TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement [ ; ] [ ,...n ] |
     EXTERNAL NAME <method specifier [ ; ] > }
```

Trigger

- `FOR` | `AFTER` – trigger is fired only if the SQL operation is successfully executed.
- `INSTEAD OF` – operations of the trigger are executed instead of the SQL operation.

Triggers, example 1/2

```
CREATE TRIGGER trigAddProduct
ON Product
FOR INSERT
AS
BEGIN
   DECLARE @id INT;
   SELECT @id = (SELECT id FROM INSERTED);
   PRINT 'THE PRODUCT ' + cast(@id as VARCHAR) + ' IS ADDED.';
END;
```

Triggers, example 2/2

```
INSERT INTO Product VALUES (1, 1 , 'Desc');

Output:
THE PRODUCT 1 IS ADDED.

(1 row(s) affected)
```

Triggers, alternative

The same trigger can be written, but the code is not good readable without **BEGIN** and **END**:

```
CREATE TRIGGER trigAddProduct
ON Product
FOR INSERT
AS
   DECLARE @id INT
   SELECT @id = (SELECT id FROM INSERTED)
   PRINT 'THE PRODUCT ' + cast(@id as VARCHAR) + ' IS ADDED.'
```

Example

SQL is a basis of T-SQL, let us have a task:
Write a stored procedure `PrintReport()`, which prints a print report of teachers which are at departments having more than one teacher. Print login, fname, lname, email, and department id for each teacher. The procedure has to be implement with one cursor (and query).

```
DECLARE tchList CURSOR FOR SELECT * FROM Teacher_pract6
  WHERE department IN (SELECT department FROM Teacher_pract6
                                    GROUP BY department HAVING COUNT(*)>1)
```

- The nested select returns departments with the number of teachers greater than 1.
- The outer select returns values of all attributes of the teachers.

Bulk Operations for MS SQL Server

- Although in T-SQL there are no commands like `BULK COLLECT` and `FORALL`, we can use some bulk operations:
  - `BULK INSERT` of a data file:

    ```
    BULK INSERT Table
    FROM '\\computer\table.txt';
    ```

    However, the files have to be stored on a computer with SQL Server.
  - `OPENROWSET (BULK)`:

    ```
    INSERT INTO Table SELECT a.* FROM
    OPENROWSET (BULK N'D:\table_data.csv',
    FORMATFILE = 'D:\table.txt',
    CODEPAGE = '65001') AS a;
    ```
- https://docs.microsoft.com/en-us/sql/relational-databases/import-export/import-bulk-data-by-using-bulk-insert-or-openrowset-bulk-sql-server

PL/SQL vs T-SQL

- T-SQL does not provide any operators similar to `%TYPE` and `%ROWTYPE`.
- T-SQL restricts constructions which can be used in fuctions.
- In T-SQL, we must use the commands OPEN, FETCH (2x), CLOSE, DEALLOCATE for cursors.
- In T-SQL, we must use the cursor's FETCH twice.
- In T-SQL, we must use the length in the case of procedure's and function's parameters if necessary.

But ...

- T-SQL does not provide any operators similar to %TYPE and %ROWTYPE.
  - Instead of %ROWTYPE we can use a temporary table (titled by a prefix #):

```
SELECT * INTO #tempStudent FROM Student
WHERE ...
```

References

- **Transact-SQL Reference (Database Engine):**
  `http://msdn.microsoft.com/en-us/library/bb510741.aspx`
- **DevGuru:**
  `http://www.devguru.com/content/technologies/t-sql/home.html`