

::ISUG

techcast series

New Features in ASE 15.0.2: User Defined SQL Functions & Instead Of Triggers

June 19, 2007

SYBASE®

::ISUG

techcast series

New Features in ASE 15.0.2: User Defined SQL Functions & Instead Of Triggers



Steve Glasgow
President, International
Sybase User Group

SYBASE[®]

::ISUG

techcast series

New Features in ASE 15.0.2: User Defined SQL Functions & Instead Of Triggers



Peter Schneider
Technical Director, ASE Development,
Sybase Inc.

SYBASE®

- **SQL User Defined Functions (UDF)**
 - User created functions written as a collection of SQL statements.
- **INSTEAD OF Triggers (IOT)**
 - Triggers created on views that are executed instead of INSERT, DELETE or UPDATE operations on the view.
- **Using SQL UDFs and IOTs together to implement encryption in the database.**
- **Plans for extending SQL UDF functionality in future ASE releases.**

- **The TSQL language provides a set of **built in** functions:**
 - getdate(), upper(), left(), etc.
 - Are a compact notation to perform a specific task in TSQL.
 - Can be used anywhere in TSQL where an expression is legal.
 - Are coded in C/C++ and linked into the datasever binary.
 - Their functionality is fixed: cannot be modified by users.
- **An extension to built in functions is to let users create (or define) their own functions (User Defined Functions (UDF)).**

- **What language should be used for UDFs?**
 - Procedural language like C/C++.
 - **Functions could be linked to ASE as DLLs.**
 - **Available in ASE when embedded in SQL UDFs.**
 - JAVA
 - **Java methods can be installed in the database and invoked in TSQL directly or as SQLJ functions.**
 - **Available in ASE since version 12.0**
 - SQL
 - **Functions are written as a collection of SQL statements – much like stored procedures.**
 - **Available in ASE as of version 15.0.2**

- **Why add SQL UDFs to ASE?**
 - Simplicity of use
 - Performance
 - Customer requests
 - Compatibility and Portability
 - Functionality

- **There are two kinds of SQL UDFs:**
 - **Scalar SQL UDFs.**
 - **Return a single value.**
 - **Can have multiple SQL statements in the body of the function.**
 - **Can be used anywhere an expression is legal in TSQL.**
 - **Table-valued SQL UDFs.**
 - **Return a table (set of rows).**
 - **Can have multiple SQL statements in the body of the function.**
 - **Can only be used in the FROM list in TSQL.**
- **ASE 15.0.2 supports Scalar SQL UDFs.**

- **Scalar SQL UDFs.**

- An example:

```
CREATE FUNCTION date_my_fmt (@d datetime = NULL)
RETURNS char(11) AS
RETURN str_replace(convert(char(11), @d, 106), ' ', '-')
```

```
1> select getdate()
2> go
```

```
-----
          Jun  6 2007  2:42PM
```

```
1> select dbo.date_my_fmt(getdate())
2> go
```

```
-----
06-Jun-2007
```

- The parameters to `date_my_fmt()` can be **literals**, **variables**, **procedure parameters** or **column references**.
- `date_my_fmt()` is a simple UDF, whose body could have been inlined:

```
SELECT date_my_fmt(getdate())
```

```
SELECT str_replace(convert(char(11), getdate(), 106), ' ', '-')
```

- **Using a UDF instead of inlining it's code:**
 - Stores the processing logic (body of the UDF) only once, in the database.
 - **Maintenance is simpler because modification of the UDF is localized to a single instance of the UDF definition.**
 - Provides a *handle* (the UDF name) to invoke the processing logic in SQL statements that need it without writing the code of the UDF each time.
 - **Makes the SQL clearer.**
 - **Avoids mistakes when a complex SQL expression is repeatedly used in multiple SQL statements.**
 - SQL UDFs with multiple statements cannot be inlined.

- **How is the query with the UDF executed?**

```
CREATE FUNCTION date_my_fmt(@d datetime = NULL)
RETURNS char(11) AS
    RETURN str_replace(convert(char(11), @d, 106), ' ', '-')
```

- Treated like a stored procedure:
 - **Row in sysobjects.**
 - **Rows in syscolumns for the parameters.**
 - **Row in sysprocedures for the query tree.**
 - **Row in syscomments for the definition SQL text.**

- Getting information about SQL UDFs.

```

1> sp_help date_my_fmt
2> go
Name                Owner          Object_type
    Create_date
-----
date_my_fmt         dbo            SQL function
    Jun  6 2007  2:37PM
Parameter_name      Type           Length        Prec
Scale              Param_order    Mode
-----
@d                  datetime       8             NULL
NULL
    1              in
Return Type        char           11           NULL
NULL
    2              out
(return status = 0)

```


- **How is the query with the SQL UDF executed?**

```
SELECT dbo.date_my_fmt(crdate) FROM sysobjects
```

```
1> CREATE PROCEDURE p1 @d datetime = NULL AS
    SELECT str_replace(convert(char(11), @d, 106), ' ', '-')
1> DECLARE c1 CURSOR FOR SELECT crdate FROM SYSOBJECTS
1> DECLARE @v datetime
1> OPEN c1
1> FETCH c1 into @v
1> WHILE (@@sqlstatus = 0)
    BEGIN
        EXEC p1 @v
        FETCH c1
    END
```

- **Properties of SQL UDFs**
 - They are database objects.
 - **They are managed and stored in the database.**
 - **They can be CREATE'd and DROP'd.**
 - **They can be saved/restored using dump/load.**
 - The SQL statements in the body of the UDF cannot have any side effects.
 - **No INSERT, DELETE or UPDATE statements that effect objects external to the UDF body.**
 - **No transactional statements (begin, commit, abort).**
 - **Cannot return data directly to the user.**

- **SQL UDFs cannot have non-deterministic built ins.**
 - rand(), getdate(), newid()

```
CREATE FUNCTION mygetdate ()  
RETURNS datetime AS  
RETURN getdate()
```

```
SELECT * FROM mytable WHERE mygetdate() = mygetdate()
```

- **Allowed SQL statements include:**
 - DECLARE of variables and cursors.
 - Assignment to variables with SELECT or SET statements.
 - Cursor commands on local cursors that are declared, opened, closed and deallocated in the UDF.
 - FETCH commands must fetch into local variables (with INTO clause).
 - Control flow statements.
 - EXECUTE of extended stored procedures.
- **SQL UDFs can have up to 2048 parameters.**
- **SQL UDF names must follow the rules for TSQL identifiers.**
- **SQL UDFs must be invoked with owner name: `dbo.sqlfunc()`**

- **SQL User Defined Functions (UDF)**
 - User created functions written as a collection of SQL statements.
- **INSTEAD OF Triggers (IOT)**
 - Triggers created on views that are executed instead of INSERT, DELETE or UPDATE operations on the view.
- **Using SQL UDFs and IOTs together to implement encryption in the database.**
- **Plans for extending SQL UDF functionality in future ASE releases.**

- **Consider the following example:**

```
1> SELECT name INTO t2 FROM sysobjects WHERE id < 3
```

```
2> go
```

```
1> CREATE VIEW t2view AS SELECT NAME = upper(name)  
    from t2
```

```
2> go
```

```
1> SELECT * FROM t2view
```

```
2> go
```

```
NAME
```

```
-----
```

```
SYSOBJECTS
```

```
SYSINDEXES
```

INSTEAD OF Triggers

```
1> INSERT t2view VALUES ('foo')
```

```
2> go
```

Msg 4406, Level 16, State 1: View 't2view' is not updateable because a field of the view is derived or constant.

- **To avoid the error, in ASE 15.0.2 we could try:**

```
1> CREATE TRIGGER t2vInsertTrig ON t2view INSTEAD OF  
INSERT AS
```

```
2> BEGIN
```

```
3> INSERT INTO t2 SELECT NAME from inserted
```

```
4> END
```

```
5> go
```

INSTEAD OF Triggers

```
1> INSERT t2view VALUES ('foo')
```

```
2> go
```

```
(1 row affected)
```

```
1> SELECT * FROM t2view
```

```
2> go
```

```
NAME
```

```
-----
```

```
SYSOBJECTS
```

```
SYSINDEXES
```

```
FOO
```

INSTEAD OF Triggers

- **What we have done is to create an INSTEAD OF TRIGGER (t2vInsertTrig) to make the view, t2view updateable.**
- **INSTEAD OF triggers are stored procedures that override the default action of a triggering statement (INSERT, DELETE or UPDATE) and instead perform user-defined actions.**
 - In our example, the triggering statement is:
`INSERT t2view VALUES ('foo')`
- **The existence of t2vInsertTrig allowed the example query to successfully execute.**

- **What happened during the execution of the triggering query?**
 - Because of the existence of t2vInsertTrig, ASE internally transformed the triggering query:
 - From:** Insert t2view values ('foo')
 - To:** Insert **inserted** values ('foo')
 - The row that was to be inserted into t2view was actually inserted into **inserted** instead.
 - After the transformed query finished executing, t2vInsertTrig was automatically executed.
 - **The SQL statement in t2vInsertTrig inserted the rows of **inserted** directly into table t2.**

INSTEAD OF Triggers

- An **INSTEAD OF** trigger uses the **inserted** and **deleted** tables.
 - **inserted** and **deleted** are tables that are used to store the rows that would have been modified by the triggering statement.
 - **The triggering statement inserts rows into them.**
 - **The statements in the trigger can read rows from them.**
 - The **inserted** and **deleted** tables have the same schema as the view upon which the trigger is defined.
 - The **inserted** and **deleted** tables are automatically created and managed by ASE as memory-resident tables.

- **INSTEAD OF triggers are very similar to FOR triggers:**
 - Both are a special form of user-created stored procedures.
 - Neither can be executed directly by a SQL statement: Both are indirectly executed by the execution of a triggering INSERT, DELETE or UPDATE statement.
 - Both are created with the CREATE TRIGGER command.
 - Both are dropped with the DROP TRIGGER command.
 - Both use the inserted and deleted tables to read rows that are effected by the triggering statement.

- **INSTEAD OF triggers are different than FOR triggers:**
 - INSTEAD OF triggers can only be created on views and FOR triggers can only be created on tables.
 - INSTEAD OF triggers execute **instead** of the operation of the triggering SQL statement.
 - **The triggering statement is executed to identify qualifying rows, but the INSTEAD OF trigger is executed instead of the actual INSERT, DELETE or UPDATE operation.**
 - FOR triggers execute **after** the execution of the triggering statement has been completed.

INSTEAD OF Triggers

- **Getting information about INSTEAD OF triggers.**
 - `sp_help`

```
1> sp_help t2vInsertTrig
2> go
Name                                Owner      Object_type
      Create_date
-----
t2vInsertTrig                       dbo        instead of trigger
      May 31 2007  4:14PM
```

```
(1 row affected)
Trigger is enabled.
(return status = 0)
```

- **Getting information about INSTEAD OF triggers.**
 - sp_helptext

```
1> sp_helptext t2vInsertTrig
2> go
# Lines of Text
-----
                1
text
-----
-----
-----
-----
-----
-----
create trigger t2vInsertTrig on t2view
instead of insert as
insert into t2 select NAME from inserted

(return status = 0)
```

INSTEAD OF Triggers

- **Getting information about INSTEAD OF triggers.**
 - `sp_depends`

```
1> sp_depends t2vInsertTrig
```

```
2> go
```

```
Things the object references in the current database.
```

object	type	updated	selected
-----	-----	-----	-----

dbo.t2view	view	no	no
dbo.t2	user table	yes	no

```
(return status = 0)
```

- **Getting information about INSTEAD OF triggers.**
 - `sp_depends`

```
1> sp_depends t2view
```

```
2> go
```

```
Things the object references in the current database.
```

```
object          type                updated          selected
```

```
-----
```

```
dbo.t2          user table           no               no
```

```
Things inside the current database that reference the object.
```

```
object          type
```

```
-----
```

```
-----
```

```
dbo.t2vInsertTrig          instead of trigger
```

```
(return status = 0)
```

INSTEAD OF Triggers

- **Example of using INSTEAD OF triggers in archiving.**
 - A table that keeps track of sales transactions and is constantly updated with each new transaction:

```
CREATE TABLE CurrentSales (  
    TransId          int primary key,  
    PartNo           int,  
    Quantity         int,  
    Price            money,  
    SaleDate         date)
```

- A table with the same schema that holds transaction records that are older than one week:

```
CREATE TABLE ArchiveSales ( ... )
```

INSTEAD OF Triggers

- **A view is created to provide access to all transaction records:**

```
CREATE VIEW Sales AS  
SELECT * from CurrentSales  
UNION ALL  
SELECT * FROM ArchiveSales
```

- **INSERT and DELETE INSTEAD OF triggers are created to make Sales updateable.**
 - The INSERT trigger will insert all rows into the CurrentSales table.
 - The DELETE trigger will not delete rows, but move them from the CurrentSales table to the ArchiveSales table.

INSTEAD OF Triggers

- **The INSERT INSTEAD OF trigger:**
CREATE TRIGGER SalesInsertTrig ON Sales
INSTEAD OF INSERT AS
BEGIN
 INSERT INTO CurrentSales SELECT * FROM inserted
END
- **The DELETE INSTEAD OF trigger:**
CREATE TRIGGER SalesDeleteTrig ON Sales
INSTEAD OF DELETE AS
BEGIN
 DELETE CurrentSales WHERE TransId in (SELECT TransId
 FROM deleted)
 INSERT ArchiveSales SELECT * from deleted
END

INSTEAD OF Triggers

- **With the INSERT instead of trigger on Sales, a query such as:**
`INSERT ... INTO Sales VALUES (...)`
always inserts rows into CurrentSales.
- **With the DELETE instead of trigger on Sales, a nightly batch can run a query such as:**
`DELETE Sales where datediff(week, SaleDate, getdate()) > 0`
to move rows from CurrentSales to ArchiveSales once they are older than 1 week.

- **SQL User Defined Functions (UDF)**
 - User created functions written as a collection of SQL statements.
- **INSTEAD OF Triggers (IOT)**
 - Triggers created on views that are executed instead of INSERT, DELETE or UPDATE operations on the view.
- **Using SQL UDFs and IOTs together to implement encryption in the database.**
- **Plans for extending SQL UDF functionality in future ASE releases.**

- **Ingredients**
 - Functions to encrypt/decrypt the data.
 - **encrypt is applied when data is stored.**
 - **decrypt is applied when data is retrieved.**
 - Table to store data on disk.
 - **No access permissions to users.**
 - View defined upon the table.
 - **Presents data to users in decrypted form.**
 - Instead of triggers on the view.
 - **Makes view updateable.**
 - **Encrypts data that is inserted into the table.**

- **Example:**
 - Encrypt
 - **inttohex()** – converts an integer value into a hex string.
 - Decrypt
 - **hextoint()** – converts a hex string into an integer value.
 - Table
 - **CREATE TABLE employee_t (id int primary key, name varchar(30), salary varchar(30))**
 - View
 - **CREATE VIEW employee as SELECT id, name, salary = hextoint(salary) FROM employee_t**

Encryption With UDFs and IOTs

- **Example cont'd.**
 - An INSERT instead of trigger to be able to insert values into employee:

```
CREATE TRIGGER emplInsert ON employee INSTEAD OF
INSERT AS
BEGIN
INSERT INTO employee_t SELECT id, name, inttohex(salary)
FROM inserted
END
```
- **For complete functionality, we also need instead of triggers for DELETE and UPDATE.**

Encryption With UDFs and IOTs

```
CREATE TRIGGER empDelete ON employee INSTEAD OF  
DELETE AS
```

```
BEGIN
```

```
DELETE employee_t WHERE id in (SELECT id FROM  
deleted)
```

```
END
```

```
CREATE TRIGGER empUpdate ON employee INSTEAD of  
UPDATE AS
```

```
BEGIN
```

```
DELETE employee_t WHERE id in (SELECT id FROM  
deleted)
```

```
INSERT INTO employee_t SELECT id, name, intohex(salary)  
FROM inserted
```

```
End
```

- **When a new employee is added:**

```
INSERT employee VALUES (1, 'Sam Smith', 100000)
```

- The inserted table seen by the empInsert trigger contains the new row:

```
(1, 'Sam Smith', 100000)
```

- The SQL in the instead of trigger applies the intohex() function to salary to generate the row to insert into employee_t for storage on disk:

```
(1, 'Sam Smith', 000186A0)
```

Encryption With UDFs and IOTs

- **Give Sam Smith a raise: Requires an UPDATE trigger:**

```
CREATE TRIGGER empUpdate ON employee INSTEAD OF
  UPDATE AS
BEGIN
  DELETE employee_t WHERE id in (SELECT id FROM deleted)
  INSERT INTO employee_t SELECT id, name, inttohex(salary)
  FROM inserted
END
```

```
UPDATE employee SET salary = (salary * 1.1) WHERE name =
  'Sam Smith'
```

```
Inserted row: (1, 'Sam Smith', 110000)
```

```
Deleted row: (1, 'Sam Smith', 100000)
```

```
Employee_t row: (1, 'Sam Smith', 0001ADB0)
```

Encryption With UDFs and IOTs

- **Using UDFs for stronger encryption.**
- **The encrypt() and decrypt() functions can be written as:**
 - Pure SQL UDFs
 - JAVA UDFs as SQLJ functions
 - SQL UDFs with extended stored procedures.

- **Example SQL UDFs with extended stored procedures:**
 - Write C-functions called `encrypt_func()` and `decrypt_func()`.
 - Compile and link into DLL, `encryption.so`
 - `CREATE PROCEDURE encrypt_func (@p1 varchar(30), @p2 binary(33) output) AS EXTERNAL NAME "/work/encryption"`
 - `CREATE FUNCTION encrypt (@p1 varchar(30)) RETURNS binary(33) AS`
 - `DECLARE @p2 binary(33)`
 - `EXECUTE encrypt_func @p1, @p2 output`
 - `RETURN @p2`
 - Repeat for `decrypt_func()`.

- **Example SQL UDFs with extended stored procedures:**

- Create an INSERT instead of trigger :

```
CREATE TRIGGER empInsert ON employee INSTEAD OF  
INSERT AS  
BEGIN  
    INSERT INTO employee_t SELECT id, name,  
    dbo.encrypt(salary) FROM inserted  
END
```

- **SQL User Defined Functions (UDF)**
 - User created functions written as a collection of SQL statements.
- **INSTEAD OF Triggers (IOT)**
 - Triggers created on views that are executed instead of INSERT, DELETE or UPDATE operations on the view.
- **Using SQL UDFs and IOTs together to implement encryption in the database.**
- **Plans for extending SQL UDF functionality in future ASE releases.**

- **Table valued SQL UDFs.**
 - **Inline table valued UDFs**
 - **Return a table (set of rows).**
 - **The function body must be a single SELECT statement that returns the rows.**
 - **Can only be used in the FROM list in TSQL.**
 - **Multi-Statement table valued UDFs.**
 - **Just like inline table valued functions except they can have multiple SQL statements in the function body.**
 - **Require table variables.**

- **Example Inline table-valued SQL UDF:**

```
CREATE FUNCTION EmpsInDept (@dept_name varchar(30))
RETURNS TABLE AS
BEGIN
    RETURN SELECT Name = e.name, Dept = @dept_name
           FROM employee e
           WHERE dept_id = (SELECT dept_id FROM department
                           WHERE dept_name = @dept_name)
END
```

```
1> SELECT * FROM EmpsInDept ("Receiving")
```

- **Example multi-statement table-valued SQL UDFs:**

```
CREATE FUNCTION EmpsInAllDepts
RETURNS @edTab TABLE (emp_name varchar(30), dept_name varchar(30))
AS
BEGIN
    DECLARE @dname varchar(30)
    DECLARE c1 CURSOR FOR SELECT dept_name FROM dept
    OPEN c1
    FETCH c1 INTO @dname
    WHILE @@sqlstatus = 0
    BEGIN
        INSERT @edTab
        SELECT * FROM EmpsInDept (@dname)
        FETCH c1 INTO @dname
    END
    RETURN
END
```

- **Available in ASE 15.0.2:**
 - Scalar SQL UDFs.
 - Instead Of Triggers.
- **Get much more detailed information about these features in the ASE 15.0.2 New Features Guide.**
- **Under development for a future release of ASE:**
 - Table valued UDFs.
 - Table variables.