Using SQL Procedural Language for Triggers in DB2 UDB

Paul Yip

October 24, 2001

Author Paul Yip reviews the enhancements to trigger functionality made available in Version 7.2 of IBM's DB2 Universal Database. The article describes how SQL Procedural Language extensions can be added inside triggers, greatly enhancing their functionality. Examples show how to take advantage of this new capability.

©International Business Machines Corporation 2001. All rights reserved.

Examples were developed and tested for DB2 Personal Edition, Workgroup Edition and Enterprise Edition. If you have problems with the examples, please contact the author.

With the release of DB2 UDB Version 7.2 (or DB2 V7.1 with Fixpak 3), trigger functionality has been significantly enhanced. This article will discuss briefly what triggers are for, and then walk through several examples of how to take advantage of the new features.

To begin, triggers are objects that are associated with tables and define actions that occur automatically because of an INSERT, UPDATE, or DELETE. Here are some examples of cases where you might want to use triggers:

1. When inserting, to validate or manipulate data before allowing the insert operation to occur.
2. When updating, to compare the new value with the existing value to validate it for correctness. This is especially useful if you have a column on a table that maintains state information and you wish to define valid state transitions.
3. Upon delete, to automatically insert logging information into another table for audit trail purposes.

Triggers can be used to centralize enforcement of business rules at the database level, freeing all applications and users from having to check data validity themselves. Also, if there is ever a change in the business rules, those changes are centralized at the database level rather than having to be propagated through all enforcing applications.

Example

To illustrate, the following example creates tables and triggers for a fictional Widget company's database that maintains data for orders and customer information. The following business rules are to be enforced:
1. When an order is received by the Widget company, the sum of the value of the order and the value of all unpaid invoices for the client cannot exceed the line of credit provided to that client.

2. An order can have several statuses: PENDING, CANCELLED, SHIPPED, DELIVERED, and COMPLETED. Only the following state transitions are valid:
   PENDING -> SHIPPED -> DELIVERED -> COMPLETED
   PENDING -> CANCELLED

3. Deleting an order is only allowed if it has been cancelled. Also, we'll log deleted orders into another table for audit purposes.

We'll define the following tables to illustrate our example. To download a script to create all the database objects needed for this sample, click here. (Open the file for instructions on how to run the script.)

```sql
create table customer_t (
    cust_id INT NOT NULL PRIMARY KEY,
    company_name VARCHAR(100),
    credit DECIMAL(10,2))
create table product_t (
    product_id INT NOT NULL PRIMARY KEY,
    product_name VARCHAR(100))
create table orders_t (
    order_id INT NOT NULL PRIMARY KEY,
    cust_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    status CHAR(9) NOT NULL,
    FOREIGN KEY (cust_id) REFERENCES customer_t,
    FOREIGN KEY (product_id) REFERENCES product_t)
create table delete_log_t (
    Text varchar(1000))
```

And, we'll define the following sequence objects (also new for DB2 UDB 7.2) to generate unique IDs:

```sql
create sequence cust_seq
create sequence prod_seq
create sequence ord_seq
```

**Note:** For more information on sequence objects, see the Release Notes for DB2 7.2/7.1 FixPak 3.

Next, we'll insert the following data:

```sql
insert into customer_t values
    (NEXTVAL FOR cust_seq, 'Nancys Widgets', 100)
insert into product_t values
    (NEXTVAL FOR prod_seq, 'Blue Widgets')
```

Finally, we'll create the triggers to enforce the business logic defined above, and walk through them.
Insert Triggers

The first trigger we'll create enforces business rule #1:

"When an order is received by the widget company, the sum of the value of the order and the value of all unpaid invoices for the client cannot exceed the line of credit provided to that client."

```
1: CREATE TRIGGER verify_credit
2: NO CASCADE BEFORE INSERT ON orders_t
3: REFERENCING NEW AS n
4: FOR EACH ROW MODE DB2SQL
5: BEGIN ATOMIC
6:   DECLARE current_due DECIMAL(10,2) DEFAULT 0;
7:   DECLARE credit_line DECIMAL(10,2);
9:   /*
10:     * get the customer's credit line
11:     */
12:   SET credit_line = (SELECT credit
13:     FROM customer_t c
14:     WHERE c.cust_id=n.cust_id);
15:   -- sum up the current amount currently due
16:   FOR ord_cursor AS
17:     SELECT quantity, price
18:     FROM orders_t ord
19:     WHERE ord.cust_id=n.cust_id AND
20:       status not IN ('COMPLETED','CANCELLED')
21:   DO
22:     SET current_due = current_due +
23:       (ord_cursor.price * ord_cursor.quantity);
24:   END FOR;
25:   IF (current_due + n.price * n.quantity) > credit_line THEN
26:     SIGNAL SQLSTATE '80000' ('Order Exceeds credit line');
27:   END IF;
28: END
```

The CREATE TRIGGER statement in line 1 simply says we are creating a trigger object with the name verify_credit.

NO CASCADE BEFORE INSERT ON orders_t means that the triggered action will occur before data is actually inserted into the table and actions of the trigger will not cause other triggers to be activated. For all BEFORE triggers, the NO CASCADE keywords are required.

REFERENCING NEW AS n identifies n as the required qualifier when referencing the columns of new data being inserted.

FOR EACH ROW means that this trigger will be activated once for every row that is inserted. The other option is FOR EACH STATEMENT (for AFTER triggers only), which is activated once for each SQL statement. In other words, if an INSERT statement inserts values by selecting 10 rows from another table, using FOR EACH ROW will cause the trigger to be activated 10 times. If you used FOR EACH STATEMENT, the trigger is executed only once. MODE DB2SQL is just a clause that must be specified.
BEGIN ATOMIC on line 5 and END on line 19 define the body of the trigger. Because it is BEGIN ATOMIC, the actions defined within the trigger are all or nothing actions. If any errors occur in mid-flight of this trigger, all actions are rolled back to maintain data integrity.

DECLARE <variableName> <type> [DEFAULT <value>] on lines 6 and 7 define local variables that the trigger will need to use to process the business rules.

Lines 9 and 11 illustrate the two forms of commenting that are accepted in DB2 SQL Procedure Language. You can use /* and */ to do multi-line commenting, and use ' - ' to do single line commenting.

In line 10, we SELECT the customer’s credit line value from the customer table. The predicate for this query is WHERE ord.cust_id=n.cust_id which guarantees that at most one row is returned (otherwise, an SQL error will be thrown). The portion of the predicate 'n.cust_id' refers to the value in the column supplied by the INSERT statement that activated this trigger.

Next, in line 12, we define a FOR loop that selects all the records of outstanding payments on orders and defines a read-only cursor called ord_cursor. The cursor selects the price and quantity of each existing order whose status is not COMPLETED (payment received) nor CANCELLED (order never shipped). With each row of the resulting data, we sum the value of unpaid orders to calculate total payments currently outstanding.

Finally, in line 16, the trigger compares the sum of existing balance due and value of the new order with the credit line available to the customer. If not enough credit is available, an application error is thrown with a custom SQLSTATE 80000 (using the SIGNAL statement) and an error message "Order Exceeds credit line" which can be retrieved by the application. The insert is rejected and any changes are rolled back. The error causes an SQLException to be thrown that can be handled by the calling application.

Note: The current limitation on length of the error message is 70 characters. If the message exceeds this limit, the message will be truncated without warning.

In the above example, we are assuming that only one order is created per insert. If the application were to insert multiple rows within a single INSERT statement, we would have to make this trigger an AFTER trigger because the sequence of events occurs in the following order:

1. The user or application issues an INSERT statement
2. Before the data is actually inserted, the INSERT trigger activates and executes to completion
3. If the trigger completes without error, the row is inserted.

Because the BEFORE triggers execute to completion before the data is inserted, in situations where a single INSERT statement inserts more than one row, the FOR loop will not see all the rows that the user or application is trying to insert.

There are some optimizations that can be done to make this trigger execute faster. The trigger has been coded primarily to illustrate how to use some new trigger functionality and not for performance. See Performance Tips for more information.
**Update Triggers**

Update triggers are very similar to insert triggers except that references to both the new and existing value can be accessed. From our business rules above, we want to use triggers to define valid state transitions and enforce this across all applications.

**Valid state transitions:**

PENDING -> SHIPPED -> DELIVERED -> COMPLETED

PENDING -> CANCELLED

The following trigger can be used to enforce the transitions:

```sql
1: CREATE TRIGGER verify_state
2: NO CASCADE BEFORE UPDATE ON orders_t
3: REFERENCING OLD AS o NEW AS n
4: FOR EACH ROW MODE DB2SQL
5: BEGIN ATOMIC
6:   IF o.status='PENDING' and n.status IN ('SHIPPED','CANCELLED') THEN
7:     -- valid state
8:   ELSEIF o.status='SHIPPED' and
9:       n.status = 'DELIVERED' THEN
10:   -- valid state
11:   ELSEIF o.status='DELIVERED' and
12:     n.status = 'COMPLETED' THEN
13:   -- valid state
14: ELSE
15: SIGNAL SQLSTATE '80001' ('Invalid State Transition');
16: END IF;
17: END
```

In this case, the trigger is called `verify_state` and it is a trigger that activates before any updates on `orders_t` table. Another difference is that we reference the existing (old) value by qualifying it with an 'o', and new value using 'n'.

The way in which the state transitions are verified in lines 5 to 16 is straightforward. If the transition is not expected, we assume that it is an error and throw an application error with the message "Invalid State Transition" and the operation is rejected. Of course, the logic could also have been written as:

```sql
IF NOT((o.status='PENDING' and n.status IN ('SHIPPED','CANCELLED')) OR
    (o.status='SHIPPED' and n.status = 'DELIVERED' OR
     (o.status='DELIVERED' and n.status = 'COMPLETED'))) THEN
  SIGNAL SQLSTATE '80001' ('Invalid State Transition');
END IF;
```

...but was expanded out for clarity, and to illustrate the full syntax of the IF/THEN/ELSE construct.

**Delete Triggers**

For the last business rule, we'll illustrate simpler forms of triggers that were already available before DB2 UDB 7.2, and break it up into two parts:
3a) "An order cannot be deleted if it has not been cancelled"

3b) "Deleted orders are logged for audit purposes"

Here is the trigger to enforce 3a:

```
1: CREATE TRIGGER restrict_delete
2: NO CASCADE BEFORE DELETE ON orders_t
3: REFERENCING OLD AS o
4: FOR EACH ROW MODE DB2SQL
5: WHEN (o.status <> 'CANCELLED')
6: SIGNAL SQLSTATE '80003' ('Cannot Delete an order that has not been cancelled')
```

After Trigger

For rule 3b, we'll use an AFTER trigger to log all delete operations from the orders_t table.

```
1: CREATE TRIGGER log_delete
2: AFTER DELETE ON orders_t
3: REFERENCING OLD AS o
4: FOR EACH ROW MODE DB2SQL
5: INSERT INTO delete_log_t VALUES (  
    'Order #' || CHAR (o.order_id) ||  
    'Was deleted on ' || CHAR(CURRENT_TIMESTAMP));
```

The main difference between the delete triggers and the previous two is the absence of BEGIN ATOMIC and END, which are not needed if we wish to do only a single SQL statement in the trigger. The above trigger obviously doesn't log much useful information to support an audit, but it serves to illustrate how an INSERT on one table can cause an INSERT to occur to another table using triggers. The trigger will be activated whenever a delete occurs and when the condition defined by the WHEN clause is true. If you leave out the WHEN clause completely (as in the above AFTER trigger), the trigger will always be activated.

Testing the Rules

To test business rule 1, we'll insert two orders for Nancy's Widgets. This client's credit line is only $100, and so the first order succeeds, while the second order fails. (The value of each order is $90.)

```
Insert into orders_t values (nextval for ord_seq, 1, 1, 9, 10.0, 'PENDING')
Insert into orders_t values (nextval for ord_seq, 1, 1, 9, 10.0, 'PENDING')
```

We can test business rule 2 by building on the previous action. Given our valid state transitions for an order (above), an order that is 'PENDING' can be 'SHIPPED'. Once an order has been shipped, it cannot be cancelled. The first update below will succeed, while the second will fail.

```
Update orders_t set status='SHIPPED' where order_id=1
Update orders_t set status='CANCELLED' where order_id=1
```

To test business rule 3, we can simply try deleting the order that we just processed. The following delete statement will fail because its status is not 'CANCELLED'. Because the before trigger failed, the AFTER DELETE trigger (which logs a delete action) is never activated.
Delete from orders_t where order_id=1

To test the logging of deletes, we can add an order that is within 'Nancy's Widgets' credit line, cancel the order, and then delete it.

Insert into orders_t values (nextval for ord_seq, 1, 1, 1, 10.0, 'PENDING')
Update orders_t set status='CANCELLED' where order_id=(prevval for ord_seq)
Delete from orders_t where order_id=(prevval for ord_seq)
Select * from delete_log_t

Performance Tips

- BEFORE triggers should be used to modify user-provided values or generate new values such as primary keys. Trying to modify rows from the transition tables in an AFTER trigger is more complex.
- DB2 is a powerful relational engine. However, it currently does not optimize procedural logic (control) statements as well as other SQL statements.

For example the verify_credit trigger can be rewritten as follows:

```sql
1: CREATE TRIGGER verify_credit
2: NO CASCADE BEFORE INSERT ON orders_t
3: REFERENCING NEW AS n
4: FOR EACH STATEMENT MODE DB2SQL
5: WHEN ((SELECT SUM(price * quantity) FROM orders_t
6:             WHERE cust_id = n.cust_id
7:             AND status NOT IN ('COMPLETED', 'CANCELLED'))
8: + n.price * n.quality
9: > (SELECT credit FROM customer_t WHERE cust_id=n.cust_id))
10: SIGNAL SQLSTATE '80000' ('Order Exceeds credit line')
```

- For updates, use FOR EACH ROW triggers instead of FOR EACH STATEMENT triggers when you have to compare the new values with the old ones. Better performance is achieved using FOR EACH ROW triggers because they have the values they need for comparisons without having to work with multiple old and new values stored in a transition table to complete the trigger operation.

Other Tips

- WHILE loops are supported. The syntax is described in the SQL Reference.
- GET DIAGNOSTICS <var : int> = ROW_COUNT can be used to determined how many rows were affected by the most recent update, delete or insert statement called within the body of a trigger.
- SELECT .... INTO syntax is not supported. To select multiple columns into multiple variables, use SET (x,y) = (SELECT x_col, y_col FROM mytable).
- Avoid recursive triggers. A recursive trigger is a trigger that activates itself by having the same kind of statement in its body. For example, if we defined a DELETE trigger on table mytable, and the trigger body also contained a DELETE statement on mytable, it would be a recursive trigger and can lead to problems if not coded carefully. If you need to use recursion, limit it to a single iteration.
- If more than one trigger is defined on a table (that is, two BEFORE INSERT triggers are defined), they will be executed in the order in which they are created. Of course, BEFORE
triggers are always activated before AFTER triggers, regardless of creation order. Also, other constraints (that is, primary/foreign key constraints, unique constraints and check constraints) that may exist on a table are checked after BEFORE triggers and before AFTER Triggers.

Summary

Triggers are useful for enforcing business rules that are central to all applications that use the database. With the new functionality provided by DB2 V7.2 or DB2 V7.1 + FixPak 3, triggers now can encapsulate more complex business rules than ever before. The new functionality also makes code maintenance much simpler because the code structure is easier to read.

Acknowledgements

The author would like to thank Serge Rielau, Richard Swagerman, and Drew Bradstock for their technical review of this article.

DB2, DB2 Universal Database and IBM are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

IBM copyright and trademark information

© Copyright IBM Corporation 2001
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)