

ANSI Database Triggers

Ramakrishna Tirunagari V

Fred Kaufmann

Mark Hodgens

Carol Thomas

ANSI Database Triggers

Table of Contents

<i>Executive Summary</i>	2
<i>Introduction</i>	2
<i>What are Triggers?</i>	3
<i>Attributes of Triggers</i>	3
<i>Statement and Row Triggers</i>	5
<i>Multiple Triggers and Cascading</i>	7
<i>How Triggers can Enforce Business Rules</i>	8
<i>Triggers can Modify and Set Default Values in Rows</i>	8
<i>Teradata Database Triggers are Unique</i>	9
<i>New Trigger Functionality</i>	10
<i>Summary</i>	12

Executive Summary

This white paper introduces database triggers and explains all the attributes associated with them. It also discusses the importance of database triggers in commercial applications and how they form the backbone of Active Data Warehousing.™

Introduction

In today's competitive world, organizations that leverage customer data improve business execution because they quickly adjust strategies to enhance performance and outpace the competition. In the airline industry, these data might range from the number of passengers flying to particular locations in a particular month to information about a particular traveler on a particular flight. The first kind of data could be used to have the correct numbers of planes in the correct places at the correct time. The second kind of data could be used to determine that a frequent flyer will be inconvenienced by a late flight and recommend an offer that will lessen the inconvenience or provide them with an appropriate benefit.

These types of examples can be made in many different industries. They also demonstrate the evolution of the data warehouse. The first example, collecting data, probably via a batch job on a monthly basis, represents a traditional data warehouse used for decision support. Decisions are most likely not made as a result of a single event. The example of an inconvenienced flyer being accommodated within minutes of an event represents an active data warehouse (ADW). Database triggers are a key component of an ADW.

Trigger features new to Teradata® Database V2R6 include: User Defined Functions called by triggers, Stored Procedures called by triggers, and Large Object (LOB) access in triggers.

ANSI Database Triggers

What are Triggers?

A trigger is a device used to release or activate a mechanism. A database trigger is a set of synchronous Structured Query Language (SQL) statements that execute automatically when a specific operation, such as changing data in a table, occurs.

Triggers, being automatically driven database events, can provide a greater deal of security, implement complex business rules, reduce costs of maintenance, and provide better performance because of less traffic in typical client-server architecture. Having complex business rules embedded inside the server ensures that everyone uses the same logic to accomplish the same process. This increases reusability and can provide platform independence as well.

Triggers can be very useful for automatically initiating update, delete, and insert statements that must be executed on a regular basis based on the occurrence of a predefined event. You need not remember to schedule these statements or execute them manually.

You can add data validation and security by defining a trigger that would roll back any actions on data when unauthorized users try to modify it.

Triggers can also be used for maintaining complex referential integrity and data consistency. For instance, a trigger can be defined to update the inventory data and sales data whenever a product has been purchased.

Triggers can also be used to:

- > Restrict a particular user from accessing a database object during specific time periods.
- > Ensure referential and data integrity.
- > Secure and audit the access of confidential business data.
- > Enforce business rules.
- > Validate data being modified or updated in the database.

Triggers have been available in the Teradata Database since release V2R3.0. As of Teradata Database release V2R5.1, triggers have been greatly enhanced to totally comply with and even extend beyond the ANSI SQL3 standards.

Attributes of Triggers

Triggers are normally designated by three attributes: what causes them to execute, when they execute, and how often they execute.

Event

A statement that causes a trigger to execute is called the triggering statement or triggering event. A trigger will be defined against a table and the trigger will specify whether an Update, Delete, or Insert statement will cause it to be executed. The triggering statement determines what will cause a trigger to execute.

Action Time

A trigger can execute BEFORE or AFTER the triggering statement. This is known as the trigger activation time or action time. At one time, triggers could also execute instead of the triggering statement but ANSI removed this option. The activation time determines when the trigger will fire.

Granularity

There are two basic types of triggers: statement and row. Statement and row triggers dictate how often a trigger will execute. This is termed the granularity of the trigger. A statement trigger executes once per statement; a row trigger executes once for each modified row. In the CREATE TRIGGER statement, the FOR EACH ROW clause denotes a row trigger and the FOR EACH STATEMENT denotes a statement trigger.

When database triggers are discussed, they are normally referred to by one of the three attributes mentioned above. When asked what kind of trigger is defined on a table, one typically responds with something like “a before row update”. This would mean that an update statement causes the trigger to execute; the trigger will execute before the update and once for each qualified row.

Triggered Action Statements

The trigger body can contain these statements: INSERT, INSERT/SELECT, UPDATE, ATOMIC UPSERT, DELETE, and ABORT or ROLLBACK AND EXEC macro. Note that the macro executed by the exec statement can only contain the statements listed in the previous sentence. These statements are known as the triggered action statements. There can be multiple triggered action statements.

ANSI Database Triggers

A simple trigger to track all updates on an employee can be written as:

```
CREATE TRIGGER employee_trigger
AFTER UPDATE ON employee
FOR EACH STATEMENT
    INSERT INTO employee_log
        (employee_name, date, time, user);
```

In the above trigger, 'employee_trigger' defined on table 'employee', the following are the attributes:

- > Triggering event is UPDATE.
- > Trigger action time is AFTER (executed after the triggering statement).
- > Trigger granularity is STATEMENT (executed only once).
- > INSERT is the triggered action statement.

REFERENCING Clause

With the capabilities described above, triggers are a potent database feature. But, the ability to reference changed data and the WHEN clause really make them powerful. What follows is a simple trigger that is written only to illustrate the idea of changed data.

The triggering statement is:

```
UPDATE inventorytable SET qty_on_hand = qty_on_
hand - 100
    WHERE partnumber > 3000;
```

Assume that seven rows are affected by this update. These seven rows form the transition table. A transition table is populated with the set of rows affected by the triggering statement. For this UPDATE statement, there will be a transition table containing seven rows before they are changed and a transition table containing the seven rows after they are changed. Important things to note about transition variables are:

- > A DELETE triggering statement will only generate a transition table containing the old rows.

- > An INSERT triggering statement will only generate a transition table containing new rows.

As can be seen from the example, an UPDATE triggering statement will generate transition tables containing both old and new rows.

In order for the trigger to look at these old and/or the new values, it must include a REFERENCING clause. The following trigger demonstrates the use of a REFERENCING clause:

```
REPLACE TRIGGER inventory_update_trigger
BEFORE UPDATE ON inventorytable
REFERENCING NEW AS NewRow newrow OLD AS
OldRowoldrow
FOR EACH ROW
    ROLLBACK "ALARM - Inventory does not have enough
stock."
    WHERE (NewRownewrow.qty_on_hand -
OldRowoldrow.qty_on_hand) < 0;
```

The REFERENCING clause allows a trigger to alias the transition rows and tables. In this case, the transition rows are named NewRow and OldRow. Once aliased, they can be used in the trigger body and the WHEN clause. In the above example, when any updated inventory table row causes a stock underflow, the transaction is rolled back.

For row triggers, changed data are referenced through the OLD and NEW as well as old_table and new_table aliases. For statement triggers, changed data are referenced through the old_table and new_table aliases. The 'old' alias refers to the content of the changed row before it was modified. It's only valid when used with the DELETE and UPDATE triggering events. It refers to the current row in the transition table. The 'new' alias refers to the content of the changed row after it was modified. It is only valid when used with the INSERT and UPDATE triggering events. It refers to the current row in the transition table. The old_table alias refers to a transition table containing all the rows before the

ANSI Database Triggers

triggering statement modified them. This is valid only for AFTER triggers when used with the DELETE and UPDATE triggering events. Note that `old_table` can be used in statement and in row triggers. The `old_table` alias refers to a table containing all the rows after the triggering statement modified them. This is valid only for AFTER triggers when used with the INSERT and UPDATE triggering events. Note that `new_table` can be used in statement and in row triggers.

WHEN Clause

REFERENCING allows a trigger to examine and use changed data. But it does nothing to govern when a triggered action statement will execute. Sometimes, a trigger must be conditionally fired based on changed data. A WHEN clause can have a Boolean expression that determines whether or not a set of triggered action statements can be executed.

Sometimes, the condition inside the WHEN clause can be thought of as an equivalent to the SQL WHERE clause.

For example, the above trigger could have also been defined as:

```
REPLACE TRIGGER inventory_update_trigger
BEFORE UPDATE ON inventorytable
REFERENCING NEW AS NewRow OLD AS OldRow
FOR EACH ROW
WHEN ((NewRow.qty_on_hand - OldRow.qty_on_hand)
< 0)
    ROLLBACK "ALARM – Inventory does not have enough
stock.";
```

Remember that the triggering update affected seven rows in `inventorytable`. Without the WHEN clause, the triggered action will be executed seven times. But, if the new quantity only exceeds the old price three times, then the triggered action statement will only be executed three times.

A WHEN condition is very useful when the same condition needs to be applied to multiple triggered action statements inside the trigger. A WHEN clause typically performs better than a WHERE clause inside the triggered action statements because the trigger may fire fewer times.

Statement and Row Triggers

Statement Triggers

To frame a discussion about statement versus row triggers, it's worthwhile to once again state their difference. The WHEN condition is evaluated only once for a statement trigger, but, it's evaluated for each transition table row for a row trigger. If there is no WHEN condition, then a statement trigger will fire once per statement while a row trigger will fire once per changed row.

The following example has a DELETE statement trigger that propagates information in several tables before deleting a row in the `employees` table. Each associated employee record (from the tables that have foreign keys referring to the primary key in the employee record) is deleted. If the employee being deleted is also a manager, then the information is also updated in the corresponding tables. Note that if no granularity is specified in the FOR EACH clause, then the default is STATEMENT.

ANSI Database Triggers

```
CREATE TRIGGER EMPLOYEE_ID_CASCADE_DELETE
AFTER DELETE ON employees
REFERENCING OLD TABLE AS del_employee
(
  -- Delete the information about that employee
  -- from all other associated tables
  DELETE FROM salary_history SH
  WHERE sh.employee_id = del_employee.employee_id;
  -- Also, if an employee is terminated and that employee
  -- is the manager of a department, set the manager_id
  -- column to null for that department.

  UPDATE departments D
    SET d.manager_id = NULL
  WHERE d.manager_id = del_employee.employee_id;
  -- Also, update all the employee information
  -- to indicate that they do not have a manager
  -- by setting it to NULL.

  UPDATE employee E
    SET e.manager_id = NULL
  WHERE e.manager_id = del_employee.employee_id;
);
```

While the above trigger could also be written as a row trigger, performance results dictate that a statement trigger is better. In this case, a statement trigger will perform better because the Teradata Database will not independently process each row, and processing overhead will be decreased. A statement trigger also works here because row order is not important.

Row Triggers

An excellent application of triggers is in the parts and suppliers problem. Consider a company where many suppliers can supply a product and a supplier can supply many products. Assume there is an inventory table to hold the information for each product (product number, product name, quantity on hand, quantity on order, safety stock, and reorder quantity) and a supplier table to hold the information about each supplier (supplier number, product number, supplier name, and unit cost). An invoice table (invoice number, product number, supplier number, and quantity requested) holds the information for each invoice generated. Triggers can be used in many scenarios, some of which are given here:

A business rule might specify that no more than four suppliers can supply a single product. A trigger can be coded on the supplier table to check that a new supplier will not be inserted into the supplier table if the data violates this requirement. The following trigger would achieve this:

```
CREATE SET TABLE sm.supplier_table ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT
(
  productno INTEGER,
  name CHAR(15) CHARACTER SET LATIN NOT
  CASESPECIFIC)
PRIMARY INDEX ( productno );

CREATE TRIGGER supplier_limit
AFTER INSERT ON supplier_table
REFERENCING NEW_TABLE AS newsupplier
FOR EACH ROW
WHEN (4 < (SEL COUNT (*))
FROM supplier_table, newsupplier WHERE
  supplier_table.productno = newsupplier.productno))
ABORT 'No more than 4 suppliers are allowed to supply
one product.';
```

ANSI Database Triggers

An order from a customer causes an update of the inventory. In addition, a trigger can be created to reorder products whose quantity has fallen below a threshold limit.

```
CREATE TABLE invoice_table(productno int, qtyordered int,
qty_on_hand int);
CREATE TABLE product_table(productno int, threshld int,
    qty_on_hand int, reorderqty int);
CREATE TRIGGER invoice
    AFTER INSERT ON invoice_table
    REFERENCING NEW ROW AS newrow NEW_TABLE AS
    newtab
    FOR EACH ROW
    WHEN (newrow.qtyordered <
        (SEL product_table.qty_on_hand FROM
        product_table, newtab
            WHERE product_table.productno =
            newtab.productno))
    UPDATE product_table
        SET qty_on_hand = qty_on_hand -
        newrow.qtyordered
        WHERE product_table.productno =
        newtab.productno;
```

Note that these are row trigger examples. There are probably other ways to write them. A more complete application would have a trigger that would report an error if the quantity ordered exceeded the stock on hand.

Whenever the price of a particular part increases by a specific amount, log the event.

```
REPLACE TRIGGER priceincreasetrigger
AFTER UPDATE ON partstable
REFERENCING NEW ROW AS newrow OLD ROW AS oldrow
FOR EACH ROW
WHEN ( newrow.price > oldrow.price + 100)
    INSERT INTO priceincreaselogtable
        VALUES(oldrow.partnumber, newrow.price,
        oldrow.price);
```

General Guidelines about Row and Statement Triggers

A relational database does not guarantee the order of rows processed by an SQL statement. ANSI also does not impose any rules on the order in which the rows inside the transition table should be processed. Therefore, don't create triggers that depend on the order in which rows will be processed.

Multiple Triggers and Cascading

Multiple Triggers

Multiple triggers are when more than one trigger is defined on a table. These triggers abide by normal restrictions. There can be any number and type of triggers defined on the same table. Typically, the activation time dictates when a trigger will fire. If there are BEFORE and AFTER triggers defined on the same table, then, obviously, the BEFORE trigger will fire first. However, if there are several triggers defined on the same table with the same activation time, then two other factors determine trigger firing time: ORDER clause and timestamp. A user can specify an optional ORDER clause. For two triggers with the same activation time, the one with the lowest ORDER value will fire first. If no ORDER is specified, then the default value is 32,767. If multiple triggers have the same action and ORDER values, then a timestamp based upon trigger creation or alteration is used. If the action time and ORDER clauses are equal, then the trigger with the oldest timestamp will fire first. The ALTER TRIGGER statement has been extended to allow a user to change the timestamp. The simple algorithm for multiple trigger ordering is:

BEFORE, ORDER, TIMESTAMP

AFTER, ORDER, TIMESTAMP

Triggers Can Cascade

Triggers can cascade, i.e., fire other triggers. A triggered action statement can also be a triggering statement and also cause a trigger to fire. Triggered action statements in the second trigger can also cause triggers to fire. In fact, a triggered action statement can cause a change to a table that causes a trigger to fire that has already been fired as part of the cascading process. This is known as a backward

ANSI Database Triggers

reference. If done improperly, this could loop forever. For this reason, cascading is limited to 16 iterations.

How Triggers Can Enforce Business Rules

Triggers can be used to maintain complex data validation and integrity. A trigger can be used to rollback SQL transactions performed with invalid or incomplete data, before being applied to the database. Triggers can be used to maintain security at a higher level by restricting access to privileged users.

Triggers can be used to maintain complex check constraints. Teradata Database does provide column level check constraints, but it cannot help validate a column against a column in another table. You can define a row trigger to perform this complex task. CRM applications can use triggers to generate logs of their best customers when they make big purchases, or frequent travelers, or buyers of high-profit items.

Referential integrity on a child table requires the referenced parent key column to be unique. Triggers can be used as an alternative where business applications cannot guarantee that the referenced parent key column is unique. The number of business rules that can be implemented using triggers is truly limited only by your imagination or, more appropriately, your business needs.

Triggers Can Modify and Set Default Values in a Row

Triggers can modify default values in a row. For example, consider a business that provides discounts to customers based on the current purchase amount. An INSERT trigger can be defined on the sales table that sets the discount as follows:

```
CREATE TRIGGER discount_trig
BEFORE INSERT ON sales_table
REFERENCING NEW AS current_order
```

```
FOR EACH ROW
SET current_order.bonus_percent =
CASE
WHEN (current_order.purchase_amount > 10000)
THEN 50
WHEN (current_order.purchase_amount <= 10000
AND
current_order.purchase_amount > 5000) THEN 25
WHEN (current_order.purchase_amount <= 5000
AND
current_order.purchase_amount > 1000) THEN 10
ELSE 0
END;
```

The SET statement provides the capability to dynamically change the input data based on complex expressions. It can also be used to set default values to fields in a table. For instance, you may use the SET statement to fill in the current DATE and TIME automatically.

The SET statement can also be used inside an UPDATE BEFORE ROW trigger to determine the new values based on values in the old rows. You can use the SET statement to set the updated salary of an employee based on his previous salary as follows:

```
CREATE TRIGGER employee_increment
BEFORE UPDATE ON employee_table
REFERENCING NEW AS new_emp_row
OLD AS old_emp_row
FOR EACH ROW
SET new_emp_row.salary
= compute_salary (old_emp_row.salary,
old_emp_row.designation);
```

This example uses a user-defined function (UDF), compute_salary, that takes the salary and designation as input and returns the updated salary, taking into consideration the number of working hours for that employee and the number of holidays in that pay period.

ANSI Database Triggers

The SET statement inside a BEFORE row trigger can be used to automatically rectify the data being modified in the database. For example, you may define a trigger to uppercase or trim all blanks in a name of an employee table.

```
CREATE TRIGGER employee_name
BEFORE INSERT ON employee_table
REFERENCING NEW AS new_emp_row
FOR EACH ROW
    SET new_emp_row.emp_name
    = UPPERCASE(TRIM(new_emp_row.emp_name));
```

Teradata Database Triggers are Unique

In addition to being ANSI SQL 3 compliant, Teradata Database provides some useful extensions to the standard. A brief list of extensions follows:

ALTER TRIGGER TIMESTAMP

“ALTER TRIGGER <trigger name> TIMESTAMP” is a new Teradata SQL statement provided to assist users who would like to redefine the order in which triggers should be executed when there are multiple triggers defined on the same table with similar action time and event. ANSI specifies that the oldest trigger based on the creation time be executed first. A business application might have a number of triggers on a single table created at different times, and need a particular new trigger to be executed before other triggers on the table. When the new trigger is created, it becomes the youngest and will be executed after all triggers. The business application can create the trigger and use the ‘ALTER TRIGGER <trigger name> TIMESTAMP’ on the older triggers to force the trigger to execute first.

ENABLED/DISABLED Option

Because triggers need to execute the action statements inside the trigger body, they definitely cause some overhead on the normal SQL processing. A business situation might need triggers to enforce rules like auditing and security only during off-peak hours or during the weekends. ‘ALTER TRIGGER <trigger name> DISABLED’ option can be used to disable the trigger. Later, using the ENABLED option, the trigger can be enabled. Teradata Database also provides a facility with which the database administrator can disable or enable all the triggers defined on a single table in one go, by using the ‘ALTER TRIGGER <table name> DISABLED/ENABLED’ option.

ORDER Clause

You can specify an ORDER clause to control the order of trigger execution within a request when multiple triggers are defined on a subject table. The ORDER values, if specified, determine the execution sequence overriding the ANSI-specified order.

REPLACE TRIGGER

The REPLACE TRIGGER statement allows you to change the DDL definition of a trigger without having to drop and recreate it. The trigger, when replaced in this fashion, however, retains its original attributes such as its creation timestamp.

Backward References

Teradata Database, unlike other vendors’ products, allows backward references. Inside a trigger, the user can reference and even modify the contents of the table being modified through correlation names.

In Teradata Database, triggers are treated as if they’re multi-statement requests. Teradata Database’s optimizer can thus provide a scalable, efficient, and reliable plan that can execute operations inside the triggers in parallel with the triggering SQL.

ANSI Database Triggers

New Trigger Functionality

Triggers are not a new feature, but some of the functionality is new. Some of the functionality which is new to Teradata Database release V2R5.1 is:

- > Complete row triggers
- > ANSI SQL3 compliance
- > Backward references
- > SET statement

There is also some trigger functionality new to Teradata Database release V2R6.0.

User Defined Functions (UDFs) and Triggers

Triggers can call UDFs.

Example: UDFs and Triggers

```
/* concat3 is the UDF.*/
REPLACE FUNCTION concat3
  (A BLOB AS LOCATOR,
  B VARBYTE(64000),
  C BLOB AS LOCATOR)
  RETURNS BLOB AS LOCATOR
  LANGUAGE C
  NO SQL
  PARAMETER STYLE TD_GENERAL
  RETURNS NULL ON NULL INPUT
  EXTERNAL NAME 'SS!concat3!lobudf001_concat3.c';
CREATE TABLE tab1(col1 INT, col2 BLOB(1m));
CREATE TABLE temp(col1 INT col2 BLOB(1m));
CREATE TRIGGER trgudf
AFTER INSERT ON tab1
REFERENCING NEW AS cur
FOR EACH ROW
BEGIN ATOMIC
( INSERT temp(cur.col1, concat3(cur.col2, '2'xb, '3'xb));
)
END;
```

Stored Procedures and Triggers

A stored procedure (SP) can be invoked by using the CALL statement in the body of an AFTER trigger. Both row and statement triggers can call an SP.

Since statements inside the body of a trigger can not change the transaction state, the following statements are not allowed inside a stored procedure that is called from a trigger: all DDL and DCL statements, BEGIN/END TRANSACTION, COMMIT, and all exception handling statements.

Stored procedures support three types of parameters: IN, OUT, and INOUT. Since a triggered action cannot return output to the user, OUT and INOUT parameters are not allowed in an SP that is called from a trigger. The old-values-correlation name and the new-values-correlation name can be passed as parameters, but the old-values-table alias and new-values-table alias cannot be passed as parameters.

Example: Stored Procedures and Triggers (SQL)

```
CREATE TABLE tab1(col1 INT, col2 BLOB(10));
CREATE TABLE tab2(col1 INT, col2 BLOB(10));
REPLACE PROCEDURE sptrg(IN pi INT, IN pb BLOB(10))
BEGIN
  DELETE tab2;
  INSERT tab2(pi, pb );
  INSERT tab2(pi + 1, pb || '10'xb);
  INSERT tab2(pi + 2, pb || '20'xb);
END;
.COMPILE FILE sptrg.sp1
REPLACE TRIGGER trg1
AFTER INSERT ON tab1
REFERENCING NEW AS cur
FOR EACH ROW
( CALL sptrg(cur.col1, cur.col2);
);
```

ANSI Database Triggers

Example: External Stored Procedures (XSP) and Triggers

```
/* This is the SQL piece of the XSP puzzle */
REPLACE PROCEDURE xsp_trg003(
  IN param1_i VARBYTE(50), IN param1_o VARBYTE(50))
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'SS!xsp_trg003!xsp_trg003.c';

CREATE table t1(col1 VARBYTE(50), col2 BLOB(50));
CREATE TABLE t3(col1 VARBYTE(50), col2 BLOB(50));

/* SP file */
REPLACE PROCEDURE sp003(IN p1 VARBYTE(50),IN p2
VARBYTE(50))
BEGIN
  BEGIN REQUEST
    INSERT INTO t3 VALUES (p1, p2);
  END REQUEST;
END;

.COMPILE FILE xsp_trgsp03.spl

REPLACE TRIGGER trgXsp003
AFTER INSERT ON t1
REFERENCING NEW AS n
FOR EACH ROW
(
  CALL xsp_trg003(n.col1, n.col2);
);
```

Large Objects and Triggers

Large Objects (LOBs) can be accessed in triggers.

Example: LOBs and Triggers

```
REPLACE FUNCTION clob_compare
  (P1 CLOB AS LOCATOR, P2 CLOB AS LOCATOR)
RETURNS INTEGER
LANGUAGE C
NO SQL
EXTERNAL NAME 'SS!clob_compare!clob_compare.c'
PARAMETER STYLE TD_GENERAL;

REPLACE TRIGGER equality_constraint
AFTER INSERT ON tab1
REFERENCING NEW AS n FOR EACH ROW
WHEN (n.order_num IN (SEL order_num FROM orders)
WHERE (clob_compare (tab3.invoice, tab1.invoic) = 0)))
(CALL duplicate_entry (n.order_num, n.id, n.invoice); );
```

This example demonstrates that LOBs can be used for comparison in a trigger and passed to a UDF. Similar to the trigger/select statement restriction, no SP Call statements can return data.

There are performance issues to consider when triggering a stored procedure or User Defined Function with LOB parameters. An SP or UDF whose formal parameter is BLOB could be passed a VARBYTE as the actual parameter. The system converts the VARBYTE into a temporary BLOB and passes that to the SP. Since temporary BLOBs are stored on disk, the performance cost of the conversion is significant.

Another case is when the SP formal parameter is declared BLOB (100000) but the actual parameter passed was declared as BLOB. The maximum length of the source type defaults to 2,097,088,000 bytes. Whenever the source type is longer than the target type, truncation may occur. Therefore, an automatic conversion operation, which results in a data copy and the creation of a temporary BLOB, is generated.

ANSI Database Triggers

Summary

Sir Isaac Newton's third law of motion states that for every action there is an equal and opposite reaction. Had Sir Isaac been alive today, he might have applied a similar law to database triggers. Newton in a white paper on triggers is a bit dramatic, but with a little imagination, his third law can become a database law as well. For every database event on a table, such as an update, delete, or insert, there can be equal and even greater reactions from triggers defined on that table.

The Teradata Database (V2R5.1 and later) supports database triggers that are ANSI compliant. Multiple triggers can be defined on a table and statements in a trigger can, in turn, fire triggers. In Teradata Database V2R6, triggers can use UDFs, call Stored Procedures, and access Large Objects (LOBs).

Triggers take advantage of scalability and parallelism of Teradata Database and are powerful tools that can lift burdens such as security and data validation from the application and place them in the database.

Fred Kaufmann is a senior member of the Teradata technical staff. During his tenure, Fred has architected and developed such features as Database Triggers, Updateable Cursors, and Large Objects.

Mark Hodgens is a senior member of the Teradata technical staff. During his tenure, Mark has architected and developed such features as Database Triggers, Referential Integrity, and Large Objects.

Ramakrishna Tirunagari is a senior member of the Teradata technical staff. Ramakrishna was the primary designer and implementer of Row Triggers.

Carol Thomas has worked for Teradata Engineering since 2000. She is currently testing and documenting the LOB and Stored Procedure features for Teradata Database V2R6.

This document, which includes the information contained herein, is the exclusive property of Teradata Corporation. Any person is hereby authorized to view, copy, print, and distribute this document subject to the following conditions. This document may be used for non-commercial, informational purposes only and is provided on an "AS-IS" basis. Any copy of this document or portion thereof must include this copyright notice and all other restrictive legends appearing in this document. Note that any product, process, or technology described in this document may be the subject of other intellectual property rights reserved by Teradata and are not licensed hereunder. No license rights will be implied. Use, duplication, or disclosure by the United States government is subject to the restrictions set forth in DFARS 252.227-7013(c)(1)(ii) and FAR 52.227-19.

Active Data Warehousing is a trademark and Teradata and the Teradata logo are registered trademarks of Teradata Corporation and/or its affiliates in the U.S. and worldwide. Teradata continually improves products as new technologies and components become available. Teradata, therefore, reserves the right to change specifications without prior notice. All features, functions, and operations described herein may not be marketed in all parts of the world. Consult your Teradata representative or Teradata.com for more information.

Copyright © 2004-2009 by Teradata Corporation All Rights Reserved. Produced in U.S.A.