IDUG® 2004 – North America

Triggers in DB2 for z/OS

Jay Yothers
DB2 for z/OS Development

Enabling Your
On Demand DB2 World

INTERNATIONAL DB2 USERS GROUP

Agenda

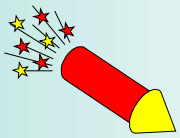
Trigger Description
Trigger Granularity
Triggered Actions
Raising Errors
Accessing Modified Data

Triggers Overview

- Triggers provide automatic execution of a set of SQL statements when a specific data change operation (UPDATE, INSERT, DELETE) occurs
  - Bring application logic into the database
  - Transform DB2 from a passive to active DBMS
- Benefits of triggers include
  - Code reuse
  - Faster application development
  - Easier maintenance

- As most of you probably know, triggers are one of DB2's oldest requirements. A trigger is a set of SQL statements that is associated with a particular table. When that table gets updated, deleted, or inserted (including updates or deletes that occur because of on delete set null or on delete cascade) those SQL statements get executed, either before or after the triggering event.
- Triggers (esp. after triggers) let you bring application logic into the database engine.
- Benefits of triggers include the ability to let a single trigger control changes to that table. For example, if you want to control updates to the salary table, let triggers do this rather than writing those checks into every application that updates the salary table. If the business rules change, it's easier to change the triggers than to change every application that updates the salary table.

Common Uses for Triggers



- Enforce business rules based on changing conditions
- Validate input data
- Generate new values for inserted / updated rows
- Cross-reference other tables
- Maintain audit, summary or mirror data in other tables
- Support "alerts"
  - E-mail notification
  - Initiate external actions

- Why use triggers? What do they offer beyond the constraint system?
- 1.Check constraints and referential constraints are limited in their ability to enforce changeable business rules. For example, a check constraint can ensure that an updated salary is within a certain range, but a before trigger can ensure that a newly inserted salary is never more than 30% of the old value.
- 2.Before triggers can validate input data, much as I described before.
- 3. A before trigger can generate values for input data based on other values. Perhaps for the salary raise that was above 30%, the trigger can reset that value to a valid value and then invoke a user-defined function to send an e-mail to an administrator about the attempt to raise the salary above the defined amount.
- 4.Triggers are not limited to referring to values in the triggering table; they can contain statements that refer to other tables.
- 5. After triggers are good for activating statements that cause updates to another table. This is a good way to create an audit trail of events that occurred in the triggering table.
- 6. Triggers can invoke user-defined functions or stored procedures, which gives you the power to invoke actions outside of the database, such as sending an e-mail or writing something to a file.

Trigger Flow

Insert 'Z-Files' row into Video_Table

Video_Table

Title	Code	Price
Toy Glory	C28	14.95
Star Trak	S31	14.95
Indiana Bones	A07	15.95
Sixth Cents	R67	19.95
Z-Files	S31	25.95

AFTER TRIGGER

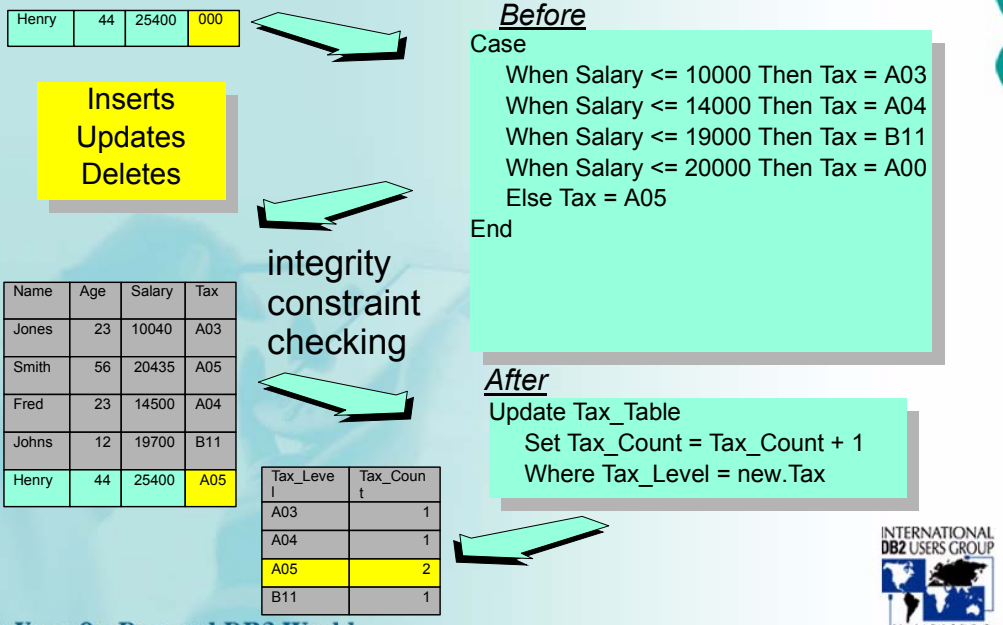
```
Update Category_Table
    Set Cat_Total = Cat_Total + 1
    Where CatNo = new.Code
```

Category_Table

Category	CatNo	Cat_Total
Adventure	A07	1
Comedy	C28	1
Drama	R67	1
Science Fiction	S31	2

- This slide shows a high level view of an after trigger. In this example, the application is maintaining
- a table of inventory. The after trigger is used to maintain a summary table of the number of videos in each category. Notice how the where predicate refers back to the incoming (new) code value. This is done by referencing to the new transition variable, which we'll talk more about later.

More on Trigger Flow



- This slide shows both a before and after trigger, and it also shows when integrity constraints are checked. The SQL is not necessarily valid syntax for a before trigger... this just shows the basic idea.
- The input data comes in with no value for tax code, because the before trigger is used to assign a tax code to the row. In this case, the salary is greater than 20000 dollars, so the system assigns tax code A05 to the input row. (All before triggers are executed in order of creation.) the change is made to the table, and then DB2 applies referential constraints, check constraints, checks that are due to updates of the table through views that are defined WITH CHECK OPTION. If the new row violates those constraints, DB2 rolls back all changes that are made by the constraint or by the triggering statement.
- Then all after triggers are processed, including after triggers on tables that were modified as a result of referential constraints. In the example shown here, the after trigger is maintaining a summary table of the number of people in a given tax category.

Trigger Characteristics

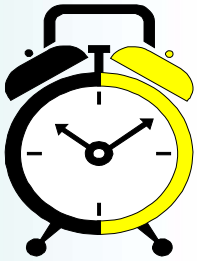
```
CREATE TRIGGER Payroll
    AFTER UPDATE OF salary ON Paytable
    FOR EACH STATEMENT MODE DB2SQL
    VALUES(PAYROLL_LOG(User, 'UPDATE',
    CURRENT  TIME, CURRENT DATE));
```

- Trigger Name: Currently limited to 8 characters
- Triggering Table: Table on which the trigger is defined
- Triggering Event:
  - An SQL Data Change Operation (INSERT,DELETE,UPDATE)
    - UPDATE can be qualified by column
  - ON the triggering table
- Trigger Activation Time: BEFORE or AFTER
- Trigger Granularity: for each row or for each statement

- Here are the basic characteristics of a trigger. We'll go into more detail on some of these in later slides.
- 1. you give the trigger a name. You can qualify the name, or you can let DB2 qualify the name for you. The 8-character limit is one that the developers are looking to lift, but for right now, assume 8 characters.
- 2. Give the name of the table with which the trigger is associated.
- 3.Indicate which event will cause the trigger to fire.
- 4.Indicate if this is a BEFORE trigger or an AFTER trigger; that is, is this trigger fired before the triggering even or after the triggering event.
- 5. Indicate whether the trigger is fired once for each changed row or once for each statement. (BEFORE triggers can never be statement-level triggers.)

Trigger Activation Time

```
CREATE TRIGGER Purchase
    NO CASCADE BEFORE INSERT ON Order
    REFERENCING NEW AS New_Order
    FOR EACH ROW
    MODE DB2SQL
    SET New_Order.Date = CURRENT_DATE;
```



BEFORE

- Evaluated entirely before triggering event
- Can be considered an extension of the constraint system
  - Prevent invalid update operations
- Useful for conditioning of input data
  - Validate or directly modify input values
- SET allows you to modify values of affected rows
  - No UPDATE, INSERT, or DELETE statements in BEFORE trigger body

▸ Let's look a little more closely at the trigger activation time. The activation time you choose is dependent on the type of action you want the trigger to perform. BEFORE triggers really have a much different purpose than AFTER triggers, and the rules for each type of trigger are different. As we mentioned earlier before triggers are really for massaging input data and validating input data. BEFORE triggers are not allowed to actually update the database, so no UPDATE, INSERT, or DELETE operations are allowed in a BEFORE trigger. There is a new SET assignment statement that lets you modify the values of the affected rows before they are entered into the database.

Trigger Activation Time

```
CREATE TRIGGER Purchase
   AFTER INSERT ON Order
   FOR EACH STATEMENT
   MODE DB2SQL
   CALL E-MAIL_CONFIRMATION;
```

AFTER

- Evaluated entirely after the triggering event
- Can be considered an encapsulation of application logic that normally would be performed by the updating application
- Perform audit trail logging or maintain summary data
- Perform actions outside the database such as writing to an external data set or sending an e-mail message

▸ AFTER triggers are not fired until after the change has already been made to the database. They are a way of pushing down application logic into the database. In this example, the after trigger is invoking a stored procedure that sends an e-mail confirmation indicating that the order has been received. Note that this is a statement level trigger--a confirmation is not sent for each video that has been ordered, just once for the entire order.

Trigger Granularity

```
CREATE TRIGGER AddOrder
    NO CASCADE
    BEFORE INSERT ON Order
    REFERENCING NEW AS NewRow
    FOR EACH ROW MODE DB2SQL
    SET NewRow.Date = CURRENT_DATE;
```

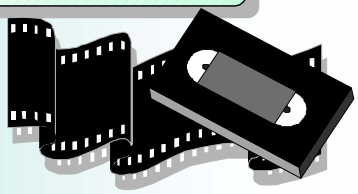
```
CREATE TRIGGER Purchase
    AFTER INSERT ON Order
    FOR EACH STATEMENT
    MODE DB2SQL
    CALL E-MAIL_CONFIRMATION;
```

Granularity controls how many times the trigger is executed

- **FOR EACH ROW: Executed once for each row modified by the triggering event**
  - **Referred to as a row trigger or a row-level trigger**
- **FOR EACH STATEMENT: Executed once each time the triggering SQL statement is issued**
  - **Referred to as a statement trigger or a statement-level trigger**

Triggered Action Condition

```
CREATE TRIGGER ReOrder
    AFTER UPDATE OF InStock ON Video_Table
    REFERENCING NEW AS N
    FOR EACH ROW MODE DB2SQL
    WHEN (N.InStock < 0.10 * N.MaxStock)
        CALL ORDER_VIDEO(N.MaxStock - N.InStock, N.Video_Num);
```



Triggered Action Condition

- Optional
- In the form of a WHEN clause (similar syntax to a WHERE clause)
- Trigger will not fire if WHEN clause not satisfied

- What if you don't want the trigger to be fired unconditionally? What if you only want a confirmation sent, or some external event to occur, only when the data is in a particular state? You can specify a condition in the form of a WHEN clause. WHEN is very similar to the WHERE clause. If the condition is not satisfied, the trigger will not fire. For a row trigger, DB2 evaluates the trigger once for each modified row of the triggering table. For a statement trigger, DB2 evaluates the condition once for each execution of the triggering SQL statement.
- The trigger in this example invokes a stored procedure that will order more of a particular video when the quantity on hand is less than 10 percent of the maximum amount that can be in stock.

Triggered SQL Statements

```
CREATE TRIGGER AddVideo
   AFTER INSERT ON Video_Table
   REFERENCING NEW AS Newrow
   FOR EACH ROW MODE DB2SQL
   BEGIN ATOMIC
      UPDATE Item_Table SET Item_cnt = Item_cnt + 1
         WHERE ItemNo = Newrow.ItemNo ;
      CALL E_MAIL_CUSTOMERS ;
   END !
```

Triggered SQL Statements

- One or more SQL statements that are executed if WHEN clause evaluates true
- Multiple statements are enclosed in BEGIN ATOMIC...END and delimited with semicolons
  - Use statement delimiter (!) for DSNTEP2, DSNTIAD, and SPUFI
- Can include stored procedure call and functions
- If trigger fails, invoking statement fails

▸ Finally, we get to the trigger body, the set of statements that are executed when a trigger is fired. More than one SQL statement must be enclosed within BEGIN ATOMIC and END. These statements can include calls to stored procedures and user-defined functions.
▸

Statements Allowed as Triggered SQL

Allowed in both BEFORE and AFTER triggers:
- CALL stored-procedure
- VALUES (expression, expression,...)
  - Normally used to invoke a user-defined function
- SELECT
  - Used to invoke user-defined functions
- SIGNAL SQLSTATE statement

Allowed only in BEFORE triggers:
- SET transition variable

Allowed only in AFTER triggers:
- INSERT
- Searched UPDATE (not a cursor UPDATE)
- Searched DELETE (not a cursor DELETE)
- All modifications are part of triggerring statement's unit of recovery

▸ This chart summarizes which statements are allowed in either a before trigger or an after trigger. Either type of trigger can call stored procedures, invoke UDFs, or raise error conditions (SIGNAL SQLSTATE). Only BEFORE triggers contain the SET assignment statement, used to massage input data. BEFORE triggers cannot modify the database using insert, update, or delete because thus leading to the possibility of nondeterministic results.Before triggers cannot make further modifications to the database because that would result in a nested stack of un-applied modifications. Which modification persists?
▸ Given that, here's an interesting example:
▸ INSERT INTO T1 (SELECT * FROM T2 WHERE C1 > 5)
▸ assume there's a before insert trigger on T1 and that trigger does this
▸
▸ INSERT INTO T2 (C1) VALUES(6)
▸ INSERT INTO T2 (C1) VALUES(7)
▸
▸ Then there's a before trigger on T2 that does this:
▸
▸ DELETE * FROM T1
▸
▸ What should the result be? When you throw in RI constraints, it's even more difficult.
▸ Only AFTER triggers are allowed to modify the database with INSERT, non-cursor UPDATES and non-cursor DELETES.

Invoking UDFs and Stored Procedures

3 ways from within a trigger body

```
1.VALUES(UDF1(NEW.COL1),UDF2(NEW.COL2);
2.SELECT UDF1(COL1), UDF2(COL2)
        FROM NEW_TABLE
        WHERE COL1 > COL3;
3.CALL StorProc(NEW.COL1, NEW.COL2);
```

- Triggers can only perform SQL operations
- Ability to invoke stored procedures and user-defined functions expands types of possible triggered actions to include:
  - Conditional logic and looping
  - Initiation of external actions
  - Access to non-DB2 resources, including remote databases
- User-defined functions cannot be invoked as a standalone call
  - Must be part of an expression in an SQL statement

▸ Triggers can only perform SQL operations. If you want to include more power within the trigger, such as the ability to initiate external actions, do more complex logic, or to process data that is not in DB2, you can invoke a stored procedure or user-defined function. Use the CALL statement to invoke a stored procedure, and use either VALUES (when the UDF doesn't need to reference a table) or SELECT to invoke a user-defined function.

Raising Error Conditions

```
CREATE TRIGGER Creditck
    AFTER UPDATE OF Balance ON Customer
    REFERENCING NEW AS Newrow
    FOR EACH ROW MODE DB2SQL
    WHEN (Newrow.Balance > Newrow.CreditLimit)
        SIGNAL SQLSTATE '75001' ('Credit Limit Exceeded -
        Shred Card');
```

Triggers can be used for stopping invalid updates and for detecting other invalid conditions.

- **SIGNAL SQLSTATE - New SQL statement that halts processing and returns the requested SQLSTATE and message to the application. Format:**
  SIGNAL SQLSTATE sqlstate-string-constant (diagnostic-string-constant)
  - **Only valid in triggered actions**

▸ Use the new statement SIGNAL SQLSTATE from within a trigger to indicate that some invalid operation is being performed and to back out the proposed changes. When DB2 executes the SIGNAL SQLSTATE statement, it returns a -438 to the SQLCA of the invoking application, but it also lets you provide your own SQLSTATE and diagnostic message.

Transition Variables

```
CREATE TRIGGER Increase
   BEFORE UPDATE OF Salary_Table ON Employee
   REFERENCING OLD AS Oldrow
                NEW AS Newrow
   FOR EACH ROW MODE DB2SQL
   WHEN (Newrow.Salary > Oldrow.Salary * 1.20)
   SET Newrow.Salary = Oldrow.Salary * 1.20;
```

Transition Variables:

- **Contain column values of row affected by triggering operation**
- **REFERENCING clause enables a correlation name to be assigned to the before and after states of the row**
  - **OLD AS Oldrow: Value of row before triggering SQL operation**
  - **NEW AS Newrow: Value of row after triggering SQL operation**

- Part of the power of triggers is your ability to look at both before and after values of a changed row.This ability is what lets you do such things as ensuring that an updated salary value is not more than a certain percentage more than the original salary.
- These old and new row values are called transition variables. There is both an old transition variable, the old value for the row, and a new transition variable, the new value of the row. Use the REFERENCING clause to give those transition variables. Row transition variables are like correlation names.

Transition Tables

```
CREATE TRIGGER Large_Order
  AFTER INSERT ON Invoice
  REFERENCING NEW_TABLE AS N_Table
  FOR EACH STATEMENT MODE DB2SQL
  SELECT
      LARGE_ORDER_ALERT(Cust_No, Total_Price, Delivery_Date)
          FROM N_Table WHERE Total_Price > 10000
```

Transition Tables:

- Contains entire set of rows affected by triggering operation
- Apply aggregations over the set of affected rows (MAX, MIN, AVG)
- REFERENCING clause specifies a table identifier
  - OLD_TABLE AS identifier: Table of BEFORE values
  - NEW_TABLE AS identifer: Table of AFTER values
- Only valid for AFTER triggers
- Can be referenced from invoked stored procedure or UDF

- Another way of accessing affected rows is through a transition table. A transition table is a hypothetical read-only table that contains all modified rows, as they appeared either before or after the triggering event. Again, you can give these transition tables any name you like, but they can only be used with AFTER triggers. Transition tables are useful when you need to perform some kind of aggregate function. You might want to know, for example, how many rows are changed by the triggering event by doing a SELECT COUNT(*) from the new transition table.
- These transition tables can be passed to stored procedures or UDFs with the use of table locators, which we'll talk about later.

Accessing trigger transition table

Trigger transition table is the set of changed rows that the triggering SQL statement modifies

Trigger can invoke UDF or stored procedure, and that UDF or stored procedure can refer to values in the transition table

Use ***table locators***

```
CREATE TRIGGER EMPRAISE
  AFTER UPDATE ON EMP
REFERENCING NEW_TABLE AS NEWEMPS
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
    VALUES (CHECKEMP(TABLE NEWEMPS));
```

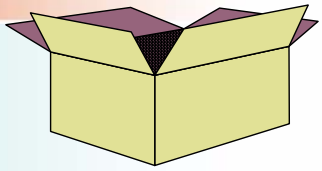
```
CREATE FUNCTION CHECKEMP(TABLE LIKE EMP AS LOCATOR)
  RETURNS INTEGER
  EXTERNAL NAME 'CHECKEMP'
  PARAMETER STYLE SQL
  LANGUAGE C;
```

Valid Trigger Characteristic Combinations

Granularity	Activation Time	Triggering Operation	Transition Variables Allowed	Transition Tables Allowed
ROW	BEFORE	INSERT	NEW	NONE
		UPDATE	OLD, NEW	
		DELETE	OLD	
	AFTER	INSERT	NEW	NEW_TABLE
		UPDATE	OLD, NEW	OLD_TABLE, NEW_TABLE
		DELETE	OLD	OLD_TABLE
STATEMENT	BEFORE	INVALID TRIGGER		
	AFTER	INSERT	NONE	NEW_TABLE
		UPDATE		OLD_TABLE, NEW_TABLE
		DELETE		OLD_TABLE

- This chart summarizes the valid combinations of trigger characteristics and transition tables or variables. For example, we see that transition tables are not allowed for a before row trigger but that they are allowed for after row triggers. This may seem like an anomaly, but it's because the set of rows to be modified is computed (and the new and old transition variables are defined) before any after row triggers are executed.
- We also see that no transition variables are allowed for after statement triggers. This is because a single statement can affect multiple rows, thereby making the assignment of single-row transition variables impossible.

Trigger packages



- When you create a trigger, DB2 creates a *trigger package*
  - Qualifier of trigger name determines package collection
    - For static, authorization ID of QUALIFIER bind option
    - For dynamic, CURRENT SQLID
  - Trigger packages are different than regular packages
    - You cannot bind them, can rebind only locally
    - They can be rebound with new REBIND TRIGGER PACKAGE command
    - Change subset of default bind options (CURRENTDATA, EXPLAIN, FLAG, ISOLATION, RELEASE)
    - Useful for picking up new access paths
    - Trigger packages cannot be freed or dropped. To delete trigger package, use DROP TRIGGER SQL statement.
    - Trigger packages cannot be copied

Trigger Performance

- SQL statements are synchronous with the application
  - All statements issued by a Trigger execute as part of the triggering statement
- After Trigger Transition Tables
  - Prior to V8, always placed in a work file
    - Even for a conditional trigger with a false condition
  - In V8, up to 4K is placed in memory