

5-9 November  
Athens Hilton  
Athens, Greece

Session: D09

**Effective Uses of DB2 Triggers**

IDUG® 2007  
Europe

Dan Luksetich  
YLA  
Dan\_Luksetich@ylassoc.com

7 November 2007 11:00-12:00

IDUG  
The Worldwide DB2 User Community

Platform: All Platforms

GoFurther



Triggers are an important database feature. They allow for the centralization of database logic, enabling faster application development, portability, and flexibility. They can be used for a variety of data intensive functions, and can be a great performance advantage. Proper use of triggers is important, however, as they can be abused. This presentation will introduce the attendee to triggers, and how they have been successfully used in real world implementations.

Daniel Luksetich is a consultant for YL&A, the leader in large high volume DB2 database design and tuning.

He works as a DBA, application architect, presenter, author, and teacher. Dan has been in the information technology business for over 22 years, and has worked with DB2 for over 17 years. He has been a COBOL and BAL programmer, DB2 system programmer, DB2 DBA, and DB2 application architect. His experience includes major implementations on z/OS, AIX, and Linux environments.

Dan's experience includes: Application design and architecture, Database administration, Complex SQL, SQL tuning, DB2 performance audits, Replication, Disaster recovery, Stored procedures, UDFs, and triggers. Dan works 8-16 hours a day, everyday, on some of the largest and most complex DB2 implementations in the world. He is a certified DB2 DBA and application developer, and the author of several DB2 related articles as well as co-author of the DB2 9 for z/OS Certification Guide.

## Abstract

---

Triggers are an important database feature. They allow for the centralization of database logic, enabling faster application development, portability, and flexibility. They can be used for a variety of data intensive functions, and can be a great performance advantage. Proper use of triggers is important, however, as they can be abused. This presentation will introduce the attendee to triggers, and how they have been successfully used in real world implementations.

Outline:

- Introduction to Triggers
- Types of triggers
- Usefulness and Purpose of Triggers
- Performance implications
- Alternatives to Triggers
- Real World Trigger Examples



Outline:

Introduction to Triggers

Types of triggers

Usefulness and Purpose of Triggers

Performance implications

Alternatives to Triggers

Real World Trigger Examples



**Yevich, Lawson & Assoc. Inc.  
2743 S. Veterans Pkwy PMB 226  
Springfield, IL 62704**

**www.ylassoc.com  
www.db2expert.com**

---

IBM is a registered trademark of International Business Machines Corporation.

DB2 is a trademark of IBM Corp.

© Copyright 1998-2007, YL&A, All rights reserved.



© YL&A 2007 – IDUG EU 2007

3



## **Disclaimer PLEASE READ THE FOLLOWING NOTICE**

- **The information contained in this presentation is based on techniques, algorithms, and documentation published by the several authors and companies, and in addition is the result of research. It is therefore subject to change at any time without notice or warning.**
- **The information contained in this presentation has not been submitted to any formal tests or review and is distributed on an “As is” basis without any warranty, either expressed or implied.**
- **The use of this information or the implementation of any of these techniques is a client responsibility and depends on the client’s ability to evaluate and integrate them into the client’s operational environment.**
- **While each item may have been reviewed for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere.**
- **Clients attempting to adapt these techniques to their own environments do so at their own risks.**
- **Foils, handouts, and additional materials distributed as part of this presentation or seminar should be reviewed in their entirety.**

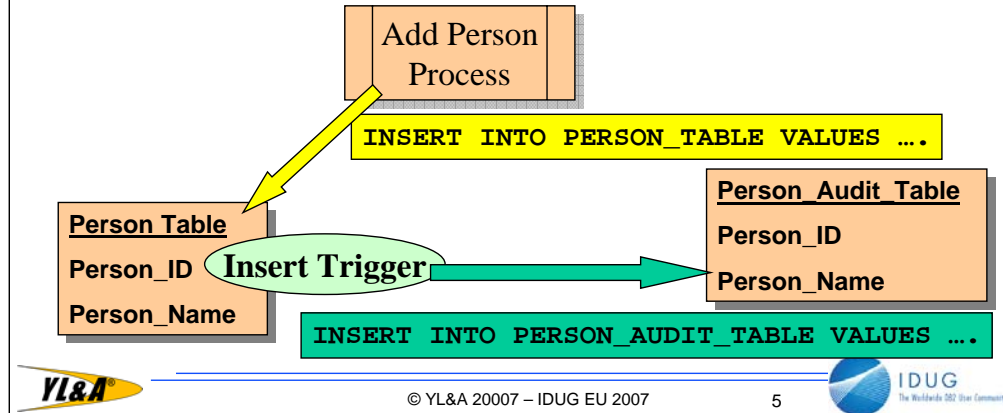


DB2 is a registered trademark of IBM Corporation.

No animals were harmed during testing.

## Introduction to Triggers

- **Triggers are Advanced Database Objects**
  - They Contain Application Logic
  - They are Installed into a Database
- **Triggers are Active Objects**
  - They are Activated by Statements Issued Against Tables
    - Can be Activated by Changes to Data
    - Can be Activated by the Issue of a Statement

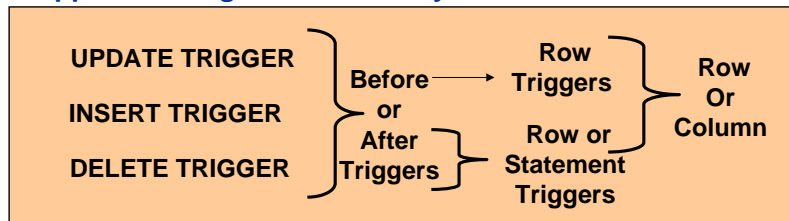


A trigger is a set of actions that will be executed when a defined event occurs. SQL INSERT, UPDATE, and DELETE statements can be triggering events. In addition, execution of a MERGE statement can cause the activation of a trigger as a result of an underlying INSERT, UPDATE, and/or DELETE.

You define triggers for a specific table or view (for INSTEAD OF triggers). Once defined, a trigger is automatically active. A given table can have multiple triggers defined for it; in this case, the order of trigger activation is based on the trigger creation timestamp (i.e., the order in which the triggers were created). Triggers can be fired based on the execution of statements (INSERT, UPDATE, DELETE); based on the modification, creation, or deletion of rows; or if specific columns of a table change.

## Types of Triggers

- **Triggers are Attached to Tables**
  - Invoked by Updates, Delete, and Inserts Either Before the Event Occurs or After
    - Before Triggers Fire Before a Change to a Table
    - After Triggers Fire After a Change to a Table
  - Row Triggers Fire Once for Each Row Changed
  - Statement Triggers Fire Once per Statement (INSERT, UPDATE, DELETE)
- **Instead Of Triggers**
  - Attached to Views Instead of Tables
  - Supports Changes to Read-Only Views



You can define a trigger to *fire* (be activated) in one of three ways:

A *before trigger* fires for each row in the set of affected rows before the triggering SQL statement is executed. Therefore, the trigger body sees the new data values before anything is inserted or updated into the table.

An *after trigger* fires for each row in the set of affected rows after the statement has successfully been completed (depending on the defined granularity). Therefore, the trigger body sees the table as being in a consistent state (i.e., all transactions have been completed).

An *instead of trigger* fires instead of the INSERT, UPDATE, or DELETE statement that activates the trigger. Unlike the other triggers, this trigger type can be defined only against a view.

Another important feature of triggers is that they can fire other triggers (or the same trigger) or other constraints. Triggers that behave this way are known as *cascading triggers*.

## What Can Triggers Do?

- **Triggers Should be Used to Support “Data Intensive” Application Processes**
  - Auditing Changes to Tables
  - Replication
  - Enforcement of Business Rules
  - Generation of Data
- **When “Table A” Changes does “Table B” Need to Change as Well?**
  - Use a Trigger!
- **Triggers can Only Contain SQL**
  - But SQL can Invoke Stored Procedures and User-Defined Functions
    - **Any Underlying Process can be Written to Satisfy Many Types of Business Requirements**
      - The Stored Procedures Could Update Other Tables, Causing Other Triggers to be Fired.



Some of the uses of a trigger include the following:

**Data validation.** Ensures that a new data value is within the proper range. This function is similar to table-check constraints but is a more flexible data-validation mechanism.

**Data conditioning.** Implemented using triggers that fire before data record modification. This technique lets the new data value be modified or conditioned to a predefined value.

**Data integrity.** Can be used to ensure that cross-table dependencies are maintained. The triggered action could involve updating data records in related tables. This use is similar to referential integrity but it is a more flexible alternative.

You can also use triggers to enforce business rules, create or edit column values, validate input data, and maintain summary or cross-reference tables. Triggers enable enhanced enterprise and business functionality, faster application development, and global enforcement of business rules.

Limited only by your imagination, triggers give you a way to obtain control in order to perform an action whenever a table's data is modified. For example, a single trigger invoked by an update on a financial table could invoke a user-defined function (UDF) and/or call a stored procedure to invoke another external action, which could trigger an e-mail to a pager to notify the DBA of a serious condition. Far-fetched? No, applications such as this one are already being done.

## Transition Variables and Tables

- **Transition variables allow row triggers to access row data**
  - See row data as is existed before the triggering operation
  - See row data as is existed after the triggering operation
- **Implemented by a REFERENCING clause in the definition**
  - REFERENCING OLD AS OLD\_ACCOUNTS  
NEW AS NEW\_ACCOUNTS
- **Transition tables allow after triggers to access set of affected rows**
  - All affected rows are they were before the triggering operation
  - All affected rows are they were after the triggering operation
- **Implemented by a REFERENCING clause in the definition**
  - REFERENCING OLD\_TABLE AS OLD\_ACCT\_TABLE  
NEW\_TABLE AS NEW\_ACCT\_TABLE

*Transition variables* permit row triggers to access columns of affected row data to see the row data as it existed before and after the triggering operation. To implement these variables, you include a REFERENCING clause in the definition, with OLD specifying the column values before the change and NEW specifying the column values after the change:

REFERENCING OLD AS OLD\_ACCOUNTS  
NEW AS NEW\_ACCOUNTS

*Transition tables* enable after triggers to access a set of affected rows and see how they were before and after the triggering operation. As with transition variables, you implement transition tables using the REFERENCING clause in the trigger definition:

REFERENCING OLD\_TABLE AS OLD\_ACCT\_TABLE  
NEW\_TABLE AS NEW\_ACCT\_TABLE

Transition tables enable an SQL statement embedded in the trigger body to access the entire set of affected data in the state it was in before or after the change. In the following example, a fullselect reads the entire set of changed rows to pass qualifying data to a user-defined function.

## Trigger Types – Before and After

- **Before triggers**
  - Always row triggers
  - **FIX or CONDITION** data before it is entered
  - **Validate or modify** input values
  - **Modify or set** column values
  - **Prevent invalid** update operations
  - **Cannot manipulate** the database
    - **YES SELECT, VALUES, SIGNAL, CALL, SET**
    - **NO UPDATE, INSERT, DELETE**

```
CREATE TRIGGER UPDATE_TS
NO CASCADE BEFORE INSERT ON ACCT_TBL
REFERENCING NEW AS NEW_ROW
FOR EACH ROW
MODE DB2SQL
SET NEW_ROW.UPDATE_TIMESTAMP = CURRENT_TIMESTAMP;
```



The following before trigger ensures that any employee who is going to be assigned a management position has put in at least 100 hours on projects in that department.

```
CREATE TRIGGER CHKHOURS
NO CASCADE BEFORE UPDATE OF MGRNO ON DEPT
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN ((SELECT SUM(A.EMPTIME)
FROM EMPPROJECT A
INNER JOIN
      PROJ B
ON      A.PROJNO = B.PROJNO
INNER JOIN
      DEPT C
ON      C.DEPTNO = B.DEPTNO
WHERE  A.EMPNO = N.MGRNO
) < 100)
BEGIN ATOMIC
SIGNAL SQLSTATE '70003'
('Not Enough Hours for Management!');
END!
```

If the assigned manager has not put in the 100 hours on projects in the target department, the SQL update statement will actually fail with an SQLSTATE of 70003 and the SQL diagnostic message “Not Enough Hours for Management!”

## Trigger Types – Before and After

- **After triggers**
  - Evaluated totally after the triggering event
  - Can be a Row Trigger and can be a Statement Trigger
  - Could initiate actions outside the database (Stored Procs, UDFs)
  - Do normal application logic that would have been done by any program/transaction that was updating the database
  - Can operate on any table in the database
    - YES SELECT, VALUES, SIGNAL, CALL
    - YES INSERT, DELETE, UPDATE

```
CREATE TRIGGER UPDATE_IMS_TG
AFTER INSERT ON ACCT_TBL
REFERENCING NEW_TABLE AS NEW_ACCTS
FOR EACH STATEMENT
MODE DB2SQL
CALL IMSUPROG (TABLE NEW_ACCTS);
```



The following example defines a trigger to add \$100 to the value of the BONUS column in the EMP table when the employee has put in more than 100 work units on projects. (Note that we added the BONUS column for this scenario.) We've named the trigger CHKBONUS. Once this trigger is created, it is active.

```
CREATE TRIGGER CHKBONUS
AFTER INSERT ON EMPPROJACT
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
WHEN ((SELECT SUM(EMPTIME)
FROM EMPPROJACT
WHERE EMPNO = N.EMPNO) > 100)
begin atomic
UPDATE emp
SET bonus = bonus + 100;
END!
```

The CHKBONUS trigger is an AFTER, INSERT, and FOR EACH ROW trigger. Every time a row is inserted into the EMPPROJACT table, this trigger fires. The trigger body section performs an UPDATE statement to set the value of the BONUS column for the newly inserted row. The column is populated with the current bonus plus 100 dollars.

## Trigger Types – Instead of

- Valid only on DB2 for LUW or DB2 9 for z/OS
- Useful for non-updateable views
- Allows an update to a view
  - Trigger is used to update the underlying table
- The **INSTEAD OF** trigger fires instead of the statement that activated it

```
CREATE VIEW EMPV (EMPNO, FIRSTNME, MIDINIT, LASTNAME,  
                PHONENO, HIREDATE, DEPTNAME) AS  
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,  
       PHONENO, HIREDATE, DEPTNAME  
FROM EMP A  
INNER JOIN  
DEPT B ON A.WORKDEPT =B.DEPTNO;
```

```
CREATE TRIGGER EMPV_DELETE  
INSTEAD OF DELETE ON EMPV  
REFERENCING OLD AS OLDEMP  
FOR EACH ROW MODE DB2SQL  
DELETE  
FROM EMP E  
WHERE E.EMPNO = OLDEMP.EMPNO;
```

```
DELETE FROM EMPV  
WHERE EMPNO = '000010';
```



To be able to insert, update, and delete against a complex view, you simply need to define one or more INSTEAD OF triggers. Suppose you have the following view defined within the DB2 sample database.

```
CREATE VIEW EMPV  
(EMPNO, FIRSTNME, MIDINIT, LASTNAME, PHONENO, HIREDATE, DEPTNAME) AS  
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,  
       PHONENO, HIREDATE, DEPTNAME  
FROM EMP A  
INNER JOIN  
DEPT B  
ON A.WORKDEPT = B.DEPTNO;
```

This view is a read-only view because it is a join between two tables. Therefore, you cannot delete from the view. If you want to be able to delete from this view, you can create an INSTEAD OF trigger to perform this operation:

```
CREATE TRIGGER EMPV_DELETE  
INSTEAD OF DELETE ON EMPV  
REFERENCING OLD AS OLDEMP  
FOR EACH ROW MODE DB2SQL  
DELETE  
FROM EMP E  
WHERE E.EMPNO = OLDEMP.EMPNO;
```

You could define similar triggers against the EMPV view to support inserts and updates to the view. Note that INSTEAD OF triggers can be row triggers only, and they cannot contain a WHEN condition.

## Understanding When a Trigger Executes is Important

- Program Issues A SQL Statement
- DB2 Retrieves the Data Page and Prepares new Values
- “Before” Triggers Happen First
- Then Integrity Constraint Checking
  - Referential Integrity
  - Table Check Constraints
  - Unique Constraints
- Then “After” Triggers
- Finally, DB2 Completes the SQL Statement and Returns



One of the things that is important to understand is the order of execution within a SQL statement.

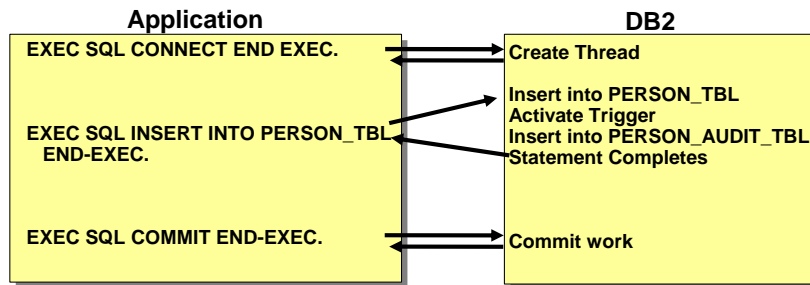
So, if a row from a table is deleted, and the table has a delete trigger defined upon it. Then DB2 will first access the page of data, modify the contents (remove the row), and then first any before trigger defined upon the table.

After control returns from the before trigger then any constraint checking is performed. This includes DB2 enforced referential integrity.

Once constraint checking is finished then any after triggers are fired, and once these complete then control is finally returned to the application that issued the delete statement.

## Triggers Are a Synchronous Event

- They Execute under the Thread of the Application that Issues the Invoking Statement
- They are a Continuation of the Unit of Work of the Invoking Transaction
- They Must Complete Before the Triggering Statement Finishes
  - This Includes Invocation of other Triggers, UDFs, Stored Procedures, etc.
  - Want an Asynchronous Process?
    - Use MQSeries
    - Write your Own Within a Stored Procedure or UDF



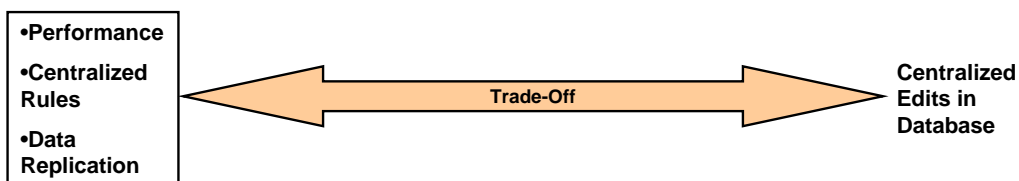
One of the most important points that people misunderstand when they use or consider using triggers is that the firing of a trigger is a synchronous process. That is the trigger processing happens under the thread of the process that issued the triggering SQL statement.

The firing of triggers is not asynchronous, and the triggering application will wait on the triggering SQL statement it issued until the execution of the trigger completes. This includes any nested activity in the trigger, such as the firing of user-defined functions, stored procedures, or other triggers.

The only way to circumvent this is to create your own asynchronous processing within the execution of the trigger.

## Trigger Performance Trade-Offs

- **Triggers and Constraints**
  - **Performance When Used Properly**
    - **Push Data Intense Application Logic to Database**
    - **Proper Indexes a Must**
    - **NOT a Replacement for Edits**
      - **Edits Best Done in Application Code**
  - **Flexibility**
    - **Centralized Business Rules**
    - **Replication of Data**
    - **Audit and Historical (Transaction History)**



Triggers and constraints ease the programming burden because the logic, in the form of SQL, is much easier to code than the equivalent application programming logic. This helps make the application programs smaller and easier to manage. In addition, since the triggers and constraints are connected to DB2 tables, they're centrally located rules and universally enforced. This helps to ensure data integrity across many application processes. Triggers can also be used to automatically invoke UDFs and stored procedures, which can introduce some automatic and centrally controlled application logic.

There are advantages to using triggers and constraints. They certainly provide for better data integrity, faster application delivery time, and centrally located reusable code. Since the logic in triggers and constraints is usually data-intensive, they typically outperform the equivalent application logic simply because no data has to be returned to the application when these automated processes fire. There's one trade-off for performance, however. When triggers, RI or check constraints are used in place of application edits, they can be a serious performance disadvantage. This is especially true if several edits on a data-entry screen are verified at the server. It could be as bad as one trip to the server and back per edit. This would seriously increase message traffic between the client and the server. For this reason, data edits are best performed at the client when possible.

## Trigger Issues

- **If Multiple Triggers are Needed, you May want to Consider Stored Procedures Instead**
  - Body of a Trigger could Consist of a CALL to a Stored Procedure
  - This is Probably Most Effective if There are Many Triggers
- **Do Not Generically use Triggers as a Replacement for Replication Methods**
  - Good only for Certain Situations
- **Do Not use for Simple Code Checking**
  - Use Check Constraints or Application Edits
- **Responsibility of Trigger Logic and Maintenance Falls Within Both DBA and Application Programming Groups**
  - You Must Have a Strong Partnership Between These Groups to Avoid Errors



If you are concerned about performance then you must use triggers properly. If multiple triggers need to be defined against a table you may want to consider using only one trigger and putting all of the logic into a stored procedure that will do all the work. This is only a recommendation for large number of triggers. In our testing of up to 4 triggers versus a stored procedure invocation, the 4 triggers were faster.

Triggers are a bad idea for generic replication. Since they are synchronous they will add overhead to the application, as well as dependency upon any tables accessed by the triggers. This increases response time for the triggering application, and could reduce availability. It is better for a generic replication to use an asynchronous process, such as log based replication, or a fast copy technique.

Triggers are also a bad idea from a performance perspective for code checking. Since a trip needs to be made to the database every time a table is changed, then it could take multiple trips to validate all data in an entry form. It is far better to cache the values locally for validation.

## Effective Uses of DB2 Triggers Examples

---

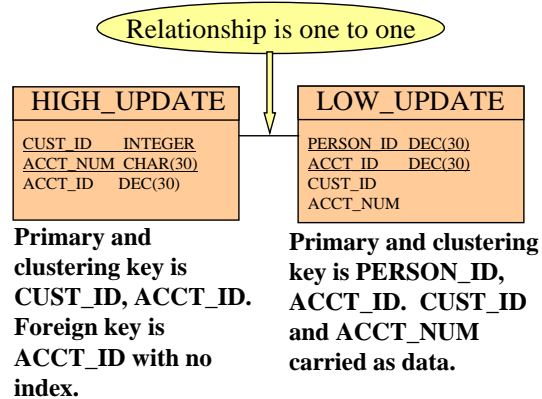
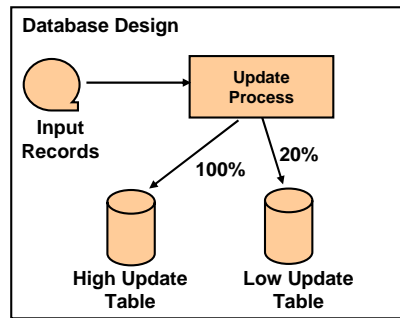
- **Using a Trigger to Extend Database Enforced Referential Integrity**
- **Enabling Auditing and “Logical Recovery” Using Triggers**
- **Real Time “Partial” Denormalization for Search Performance**



## **Using a Trigger to Extend Database Enforced Referential Integrity**

## Using a Trigger to Extend Database Enforced RI

- **Higher Performance Database Design Called for a High Speed Account Update Table**
  - Up to 170 Million Updates per Day
  - One-to-One Relationship with Main Account Table
  - Clustered and Partitioned Differently
  - Did Not Want an NPSI

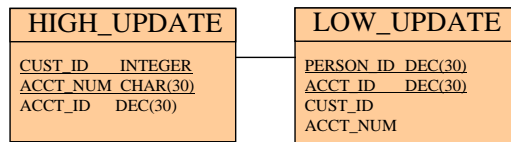


There was a situation with a major application design that required a high-speed update process. Extensive research revealed that we could achieve extreme performance by separating the regularly updated columns from those columns that are not updated that often. Clustering of the tables was dependent upon the major processes that used them. So each table had a different clustering/partitioning key.

This table design enabled us to achieved the desired transaction rates, but eliminated the ability to use DB2 enforced RI because we didn't have a supporting index in the high speed update table, nor did we want one.

## Delete of Account is to Low Update Table

- When an Account is Deleted All Data must be Removed
  - We Would Like the Delete Cascaded to the Low Update Table
- DB2 Enforced Referential Integrity is Not Possible
  - Foreign Key Must Reference a Primary Key
  - Lack of an Index Would Hurt Performance
- A Trigger is Used to Extend the RI Functionality



```
CREATE TRIGGER DELHIUPD
AFTER INSERT ON LOW_UPDATE
REFERENCING OLD AS OLD_ROW
FOR EACH ROW
MODE DB2SQL
DELETE FROM HIGH_UPDATE
WHERE CUST_ID = OLD_ROW.CUST_ID
AND ACCT_NUM = OLD_ROW.ACCT_NUM;
```



The solution was to create a trigger that allowed deletes to use the indexed key of the high update table. This avoided having to code extra programming logic to perform this function within the application. This would also be a higher performance solution than an application based solution because the delete of an account is only one SQL statement transmitted from the application to the database, and the second delete happens inside the database. If the two deletes were managed by the application then it would require two trips to the database instead of one.

## **Enabling Auditing and “Logical Recovery” Using Triggers**

## The Situation

- **Migrating a Database From a Flat File Technology to DB2**
- **Current Application Update Process Creates Updates in an Update File**
  - Update Files Are Concatenated Before the Main File
  - If Application Corrupts the Update File
    - It is Deleted
    - Updates are Instantly Removed
- **DB2 Will Have to Provide This Exact Functionality**
  - However, We Update DB2 Directly
  - We Do Not Want to Create an Update Database
  - We Cannot Use RECOVER to Remove Application Errors
    - Will Create an Outage
  - We Cannot Use SQL to Remove Application Errors
    - Will Result in “Dirty” Updates During Removal Processing
  - We Cannot Use an Application Process to Remove Errors
    - Each Error Can be Different
    - We Must Remove the Errors Quickly
- **The Solution is an Audit Database and “Logical Recovery”!!!**

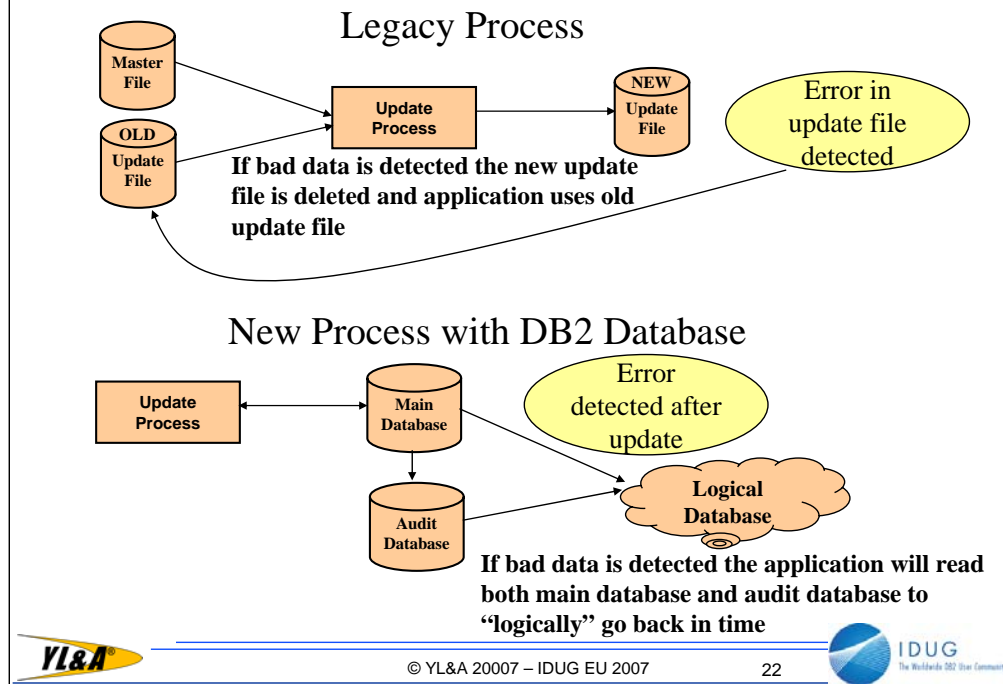


In one situation in which we were migrating a legacy data store from a flat file base technology to DB2 we had to provide legacy functionality within the DB2 database. In this situation that legacy functionality was a fast recovery process from application data corruption.

In the legacy application updates are performed via a nightly batch update process. Any changed records are not written to the original file, but instead a series of “update” files are created. Each night a new update file is created, and then concatenated in front of the main file.

So, if the update file is found to be corrupted by the application in the nightly run, it is simply deleted and all the updates from the previous update file are used. So, instant backout! How do we do this with DB2?

## Logical Recovery



The physical recovery in the legacy application simply involves deleting the new update file after an error.

We can simulate this in DB2 if we somehow track the changes to the data, and then go back in time if we find an error in the current data. The challenge is how to keep these changes, and go back quickly without interruption to the application.

## Main Database and Audit Database

- **The Audit Database Contains Before Change Images of All Tables**
  - **When Main Table Changes (DELETE or UPDATE) Audit Table Gets and Old Data**
    - **For Updates**
      - **STRT\_TSP in Audit Table is UPD\_TSP from Main Table Before the Update**
      - **END\_TSP in Audit Table is UPD\_TSP from Main Table After the Update**
    - **For Deletes**
      - **STRT\_TSP in Audit Table is UPD\_TSP from Main Table Before the Delete**
      - **END\_TSP in Audit Table is the CURRENT TIMESTAMP When the Delete is Executed**

MAIN_TABLE	
CUST_ID	INTEGER
DATA_COL	CHAR(10)
UPD_TSP	TIMESTAMP

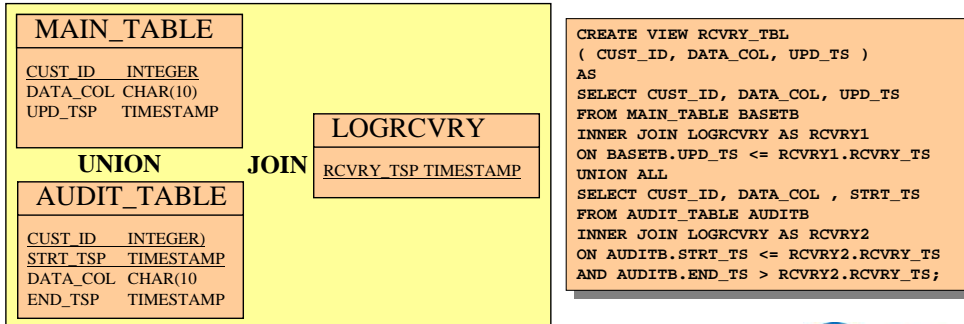
AUDIT_TABLE	
CUST_ID	INTEGER
STRT_TSP	TIMESTAMP
DATA_COL	CHAR(10)
END_TSP	TIMESTAMP



Before Images of changed data is captured in audit tables. Each base table has a corresponding audit table. The audit table is an exact replica of the base table, except for an additional timestamp column indicating when the row was created in the audit table (updated or deleted in the base table).

## Logical Recovery

- **Logical Recovery Table Holds a Single Timestamp Column**
  - Empty 99% of the Time
  - If Recovery is Required then Timestamp of Recovery Time is Inserted
  - Application will Read Logical Recovery Table
    - If Row Exists Then Logical Recovery is Activated
    - Logical Recovery Views Read Instead of Tables
    - All Other Application Processing is Exactly the Same
- **During Logical Recovery Data Comes from Either the Main Table or Audit Table Dependent upon the Logical Recovery Timestamp**



Views have been established for each base table and audit table combination. These are known as the *logical recovery views*. There is a logical recovery table that controls the logical recovery process. This logical recovery table contains a segment, subsegment, and a timestamp that indicates the point of time to recover to for a particular segment and subsegment. The logical recovery views use the logical recovery table to return either the appropriate base table row or audit table row.

## We Need to Populate the Audit Tables During Update

- **Need a Replication Methodology**
  - Triggers
  - Log Analysis - Replication
- **Triggers**
  - Fast Replication Technique
  - Easy to Implement
  - Audit Data Immediately Available
  - Online Application is Dependent Upon Audit Tables
    - Could Affect Performance
    - Could Affect Availability
- **Log Analysis**
  - Asynchronous Process So No Online Impact
  - More Complex Process
    - Need to Generate Data and Load
  - Audit Data Not Immediately Available
    - This is Critical!

Triggers are the Appropriate Choice Here!



Careful consideration was made as to how data was going to move from the base tables to the audit tables for each update or delete. We needed a way to replicate the data. There were two choices; log analysis and triggers.

Given the requirement that the database needed to be instantly recoverable to a prior point in time, triggers were choice. While it was understood that this was the lesser performing option, and created more table dependencies for the update process, it provided instant replication and satisfied the business requirement. Additionally, analysis of the update process revealed that most changes to the database were in the form of inserts, and so the triggers would have only a small impact.

## Trigger to Populate the Audit Tables

---

- **The Triggers Need To**
  - **Prevent “Blind Updates”**
    - **The Application will send Updates even though Nothing has Changed**
  - **Replicate Before Images for Updates**
  - **Replicate Before Images for Deletes**

The triggers needed to do three things: prevent unnecessary updates, replicate the before image of a row that is updated, and replicate the before image of a row before it is deleted.

## Trigger to Prevent Blind Updates

- **Application Updates all Data**
  - We Need to Prevent This
  - A Before Trigger Will Do!
- **Trigger Detects No Change**
  - A Trigger Condition is Used
  - Update Timestamp and Key are Ignored
  - “No Update” Communicated Back to Application
  - SIGNAL Statement
  - Update Does Not Happen
- **Application Must Handle SQLCODE -438**
  - And Ignore It!

```
CREATE TRIGGER UPDTRG1
NO CASCADE BEFORE UPDATE ON MAIN_TABLE
REFERENCING OLD AS OLDROW
NEW AS NEWROW
FOR EACH ROW MODE DB2SQL
WHEN ( (OLDROW.DATA_COL = NEWROW.DATA_COL) )
BEGIN ATOMIC
SIGNAL SQLSTATE '75001' ('NO UPDATED DATA')
;END!
```



The way the API worked was that it sent data to an application that updated the data and sent it back to the API. This is a blind update. Because there was no mechanism for the application to communicate a change to the API. The API would subsequently issue update statements regardless of whether or not data changed.

These unnecessary updates needed to be avoided. First of the they weren't needed, and secondly they would cause the triggers to fire when in fact no data has changed. A trigger was the solution.

The trigger was designed such that the WHEN clause tested the new value of every column against the old value of every column. The key fields and update timestamp were not compared. If there were no differences between the new and old data, then a SIGNAL SQLSTATE statement was issued to throw an error back to the API, and prevent the update from occurring.

The API programs were design to handle a SQLCODE of -438 with a SQLSTATE of '75001', and ignore it.

## Triggers to Replicate Updates and Deletes

- **These Triggers Replicate Before Images to the Audit Tables**
  - **Appropriate Timestamps are Replicated or Generated**
    - **For Updates the Start and end Timestamps Come from the Before and After Images of the Main Table**
    - **For Deletes the Start Timestamp Comes from the Main Table and End Timestamp Comes from the CURRENT TIMESTAMP**

```
CREATE TRIGGER UPDTRG2
  AFTER UPDATE ON MAIN_TABLE
  REFERENCING OLD AS OLDROW
              NEW AS NEWROW
  FOR EACH ROW MODE DB2SQL
  INSERT INTO AUDIT_TABLE
  VALUES (OLDROW.CUST_ID, OLDROW.UPD_TSP, OLDROW.DATA_COL , NEWROW.UPD_TSP);
END!
```

```
CREATE TRIGGER PPOCSSR.DELTRG1
  AFTER DELETE ON MAIN_TABLE
  REFERENCING OLD AS OLDROW
  FOR EACH ROW MODE DB2SQL
  INSERT INTO AUDIT_TABLE
  VALUES (OLDROW.CUST_ID, OLDROW.UPD_TSP , OLDROW.DATA_COL , CURRENT TIMESTAMP);
END!
```



Simple after triggers were set up on each table to perform the replication from the base table to the audit table. One update trigger, and one delete trigger. The before image of the data was transmitted to the audit table with the appropriate start and end timestamps.

For updates the start timestamp was the update timestamp of the before image of the row. The end timestamp was the update timestamp of the after image of the row. Thus the start and end timestamp of the audit table represented the range in time in which the row was active.

For deletes the start timestamp was the update timestamp of the before image of the row. The end timestamp was the current timestamp, since no after image of the row was available.

## Result of Trigger Usage

---

- **Instant Replication of the Data to the Audit Tables**
  - Allows for Instant Logical Recovery
- **No Application Programming Required for Replication**

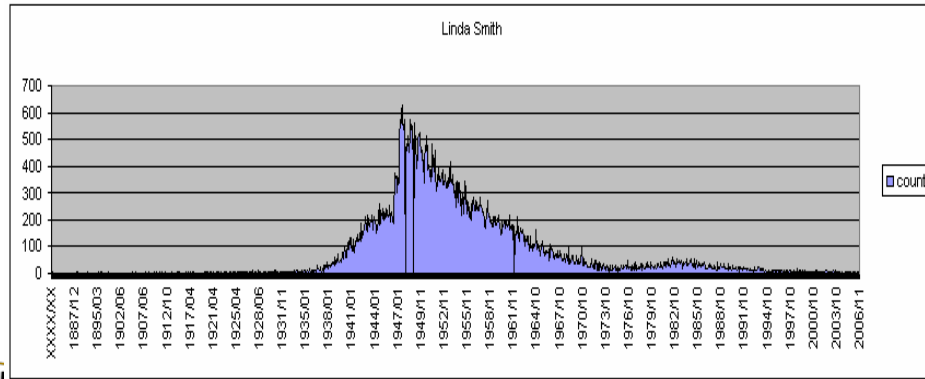


The triggers proved to be a success, and helped us to meet the business requirements.

**Real Time “Partial” Denormalization  
for Search Performance**

## The Situation

- Migration fro Legacy Data Store to DB2
- No Application Rewrite Allowed for Name Search
- Current Application Name Search Process Very Fast
  - Single Record Read
- Name Search Using DB2 is Fast for Only Some Names
  - Two Tables Searched
- Application Will Eventually be Rewritten for Better Name Search in DB2
  - However, We Need Performance Today!!!



© YL&A 20007 – IDUG EU 2007

31

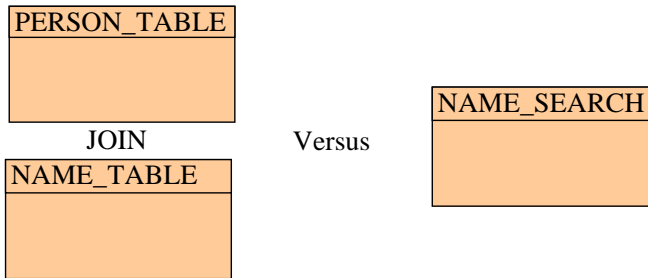
The Worldwide DB2 User Community

During performance testing of a legacy application accessing a DB2 database that was migrated from a flat file design it was determined that while performance of the name search query for less usual names was acceptable (sub second), the searches for more common names was not (up to 15 seconds). This was compounded by the fact that there could be a search of up to 60 consecutive months for one given name key.

This name key consisted of an improved Russell soundex value, the first four positions of the first name, year and month of birth. Common names during certain periods of time, such as Mary Smith July 1947 proved to be a challenge to performance, as that was the most popular name/date combination in history.

## Attempts to Make Name Search Faster for 1 Billion Names

- **Increase Buffers**
  - Not Enough Memory for 1 Billion Names
- **Put All Data Needed into Indexes**
  - Don't Want to add an Extra Index
- **Denormalize**
  - Not Good for the Future
  - Still Not Fast Enough
    - Denormalized Table is Extremely Large
  - Huge Table Size Increases Maintenance Problems



There were several tests conducted to determine what changes could be made to improve the performance for common names. These included index changes, some modest denormalization, or even a full denormalization. None of these solutions delivered the desired level of performance for common names.

In the situation with a fully denormalized table it appeared that since the denormalized table would be so large, that the resulting random I/O would still result in an unacceptable level of performance. In addition, the maintenance of such a large table would be difficult.

## The Solution

- **Partial Denormalization of Two Main Tables for a Special Name Search Table**
  - This is Done for Only the Common Names
  - Analysis of the Data
    - Regular Query Would Perform Well with Under 50 Keys per Month
      - Soundex
      - First 4 Letters of First Name
      - Year and Month of Birth
    - So We Only Need Keys with 50 or More Occurrences per Month
      - 600,000 Key Values
      - 30,000,000 Rows of Data
- **A Test Was Run**
  - The Special Names Search Table Was Accessed First
  - If Nothing is Found in the Special Names Table Then the Regular Name Search Query is Run
  - Result: All Queries are Subsecond!
- **PROBLEM!**
  - How Do We Maintain the Special Names Search Table?

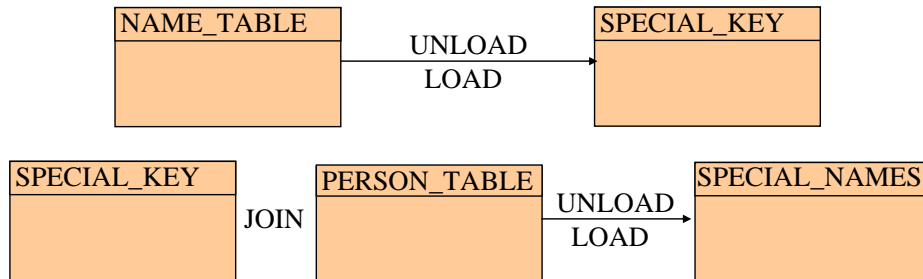


A theory was established that if we can put only the “special names”, those being the most common names, into a denormalized table then the table could be relatively small and the index could contain all table columns. The name search query would read the special names table first, and return the names found. If the query would return nothing then the normal name search query would run.

A series of tests were conducted to determine which names qualified as special by means of extracting commonly occurring names, and putting the denormalized data into a test table. After this series of tests were concluded it was determined that acceptable performance could be achieved if only the data for names occurring 50 or more times per month was placed into the special names table. This represented approximately 600,000 keys, and 31,000,000 rows.

## Populating Special Names via Refresh

- A Special Keys Table is Created
  - This Table Contains the 600,000 Keys that are Common
  - The Keys can be Unloaded from the PERSON and PERSON\_NAME Tables Anytime for Keys That Occur More than 50 Times per Month
- The Special Keys Table Can Then be Used to Repopulate The Special Names Table via UNLOAD and LOAD
  - The Application will Revert to the Normal Query When the Special Names Query Returns a SQLCODE -904
- This Process will Run Once per Year or On Demand



Before the special names table was created a special keys table was created. This special keys table was used as an indicator as to which keys were special.

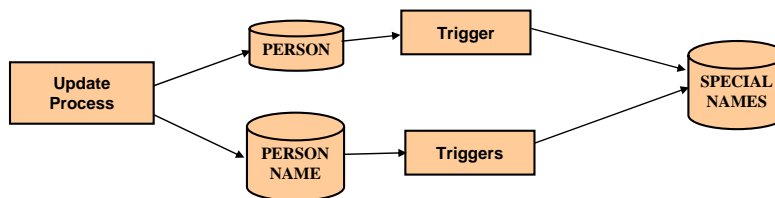
This table served two purposes; first it was used to refresh the special names table. That is, as time goes on additional names might become special (actually common). So, a query was run to see which names newly qualified, and the special keys table was refreshed with the keys for these names. Then an unload/load process was run to refresh the special names table from the special keys table.

This enabled several things to happen; yearly or on demand full refresh of special names, addition of a single key if a complaint is raised about the performance of that key, and detection of which name keys are indeed special.

## Maintaining Special Names

- This is a Temporary Solution
  - We Don't Want Programming Changes
  - We Can't Run the Refresh Process Every Night
    - 6 Hours Elapsed
  - Update Process Runs Every Night in Batch
- DB2 Triggers are the Solution to Nightly Maintenance Issue
- We Need 5 Triggers
  - PERSON Table UPDATE
  - PERSON\_NAME Table INSERT
  - PERSON\_NAME Table DELETE
  - 2 PERSON\_NAME Table UPDATE Triggers

We don't need a PERSON insert or delete trigger because that is handled by the PERSON\_NAME triggers and DB2 RI



The bottom line here is that this is a temporary solution. As we move the data into a DB2 database, it becomes quite obvious that the application should be changed to perform a more sophisticated name search. One that can take advantage of the relational database design, and filter the data more efficiently inside DB2.

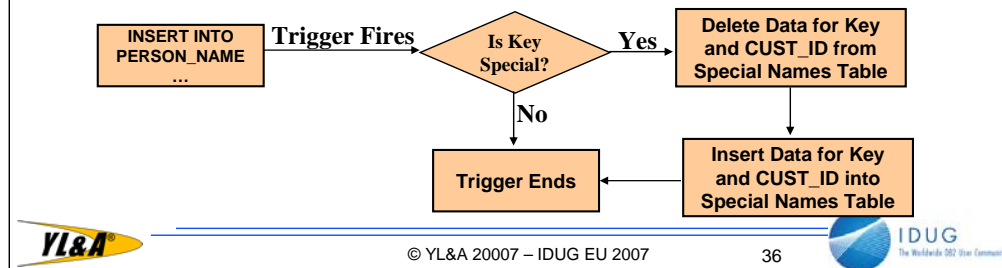
However, on a temporary basis we needed to create the special names table until the name search application can be rewritten. At that time the special names table can be thrown away.

So, why waste a bunch of application programming logic to maintain this partial denormalized table design. Instead we'll use triggers.

Several triggers need to be created to support all of the changes to the normalized tables.

## Trigger Example – Insert Into PERSON\_NAME

- This Trigger has to do the Following:
  1. See if the New Name Belongs in Special Names by Looking for the Key in the Special Key Table
    - We're Going to use a Trigger Condition
  2. Delete all Entries from Special Names for the Key and the Customer Identifier
  3. Insert into Special Names by Running the Normal Query for the Key and the Customer Identifier
    - This is Basically Running the Normal Query as a Subselect to an INSERT



There will be an AFTER INSERT trigger on the name table. The purpose of this trigger will be to update the special names table with the new name being inserted if the name matches a name in the special keys table. When this trigger searches the special names table it does not use the CUST\_ID of the name being inserted. This is because this may be an entirely new customer that has a special name, but therefore is not yet on the special name table. The trigger process is in these steps:

In the WHEN condition check to see if the new name key is in the special key table without using the customer id. Execute the trigger body only if the key is found.

Execute the trigger body (conditional on #1)

Delete the rows for the special names table for the new name key plus the CUST\_ID

Insert into the special names table the result of the normal name search query using in a WHERE clause predicates on the new name key plus CUST\_ID.

## Step One – See if the New Name is “Special”

- A Trigger Condition uses a WHEN Clause to Determine Whether or Not the Trigger Body is Executed
- The Trigger Body
  - Starts with the BEGIN ATOMIC Clause
  - Ends with an END Clause
  - Can Contain Multiple SQL Statements

```
CREATE TRIGGER NAME11
AFTER INSERT ON PERSON_NAME
REFERENCING NEW AS NEWNAME
FOR EACH ROW MODE DB2SQL
WHEN ( EXISTS ( SELECT 1
                FROM   SPECIAL_KEY
                WHERE  SNDX_NUM = NEWNAME.SNDX_NUM
                AND    FNM4      = NEWNAME.FNM4
                AND    DOB_CEN   = NEWNAME.DOB_CEN
                AND    DOB_YR    = NEWNAME.DOB_YR
                AND    DOB_MNTH  = NEWNAME.DOB_MNTH ) )
BEGIN ATOMIC
.
.
.
```

Transition variables are used to probe the special key table

## Step Two – Delete all Entries From Special Names for Key

- This Statement Within the Trigger Body Begins the Refresh Process

```
.  
. .  
. .  
DELETE FROM SPECIAL_NAMES  
WHERE CUST_ID = NEWNAME.CUST_ID  
  AND SNDX_NUM = NEWNAME.SNDX_NUM  
  AND FNM4     = NEWNAME.FNM4  
  AND DOB_CEN = NEWNAME.DOB_CEN  
  AND DOB_YR  = NEWNAME.DOB_YR  
  AND DOB_MNTH = NEWNAME.DOB_MNTH;  
. . .
```

CUST\_ID is included since we are just dealing with one customer

Transition variables are used to delete the data from the special names table for the person and the name

### Step Three – Insert the Denormalized Data into Special Names

- This Final Step Within the Trigger Body Runs the Regular Query for the Person and the Name to Re-populate the Special Names Table with the Latest Data

```
.  
. .  
. .  
INSERT INTO SPECIAL_NAMES  
SELECT A.CUST_ID, B.SNDX_NUM, ..., SEX_CDE, ...  
. .  
FROM  
    PERSON A  
INNER JOIN  
    PERSON_NAME B  
ON A.CUST_ID = B.CUST_ID  
WHERE B.CUST_ID = NEWNAME.CUST_ID  
    AND B.SNDX_NUM = NEWNAME.SNDX_NUM  
AND  
. .  
. .  
; END!
```

This is an example of the actual “regular” name search query

CUST\_ID is included since we are just dealing with one customer

## Trigger Example – Update PERSON Table

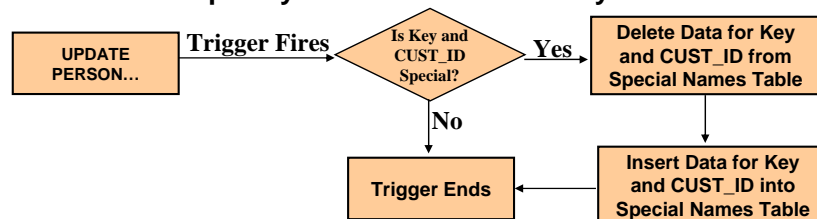
- **We Don't Need an INSERT or DELETE Trigger**
  - **DELETES Cascade to PERSON\_NAME so a DELETE Trigger There Will Take Care of Special Names**
  - **The Business Rules State That When An INSERT to PERSON Occurs then there will also be at Least One Name Inserted into PERSON\_NAME**
- **There is a Special Situation with the PERSON Table**
  - **There Trigger Transition Variables are only for the Table Affected by the Trigger**
    - **The Key to Special Names Includes Name Fields**
    - **The PERSON Table Contains No Name Fields**
    - **The Trigger Has to Get the Names for the Customer Id**
      - **We can Get Them from PERSON\_NAME for the Customer**
  - **However, When We Delete from SPECIAL\_NAMES We Will No Longer Know the Person is "Special"**
    - **So, We Need to Save the Key Plus CUST\_ID**
    - **We'll Use a Created Global Temporary Table**



The person table can be updated, and that update can impact the data in the special names table. So, we need an AFTER UPDATE trigger on the person table, but only for the columns that are used in the special names table. Since the person table contains no names, it has to acquire the names by using both the name table and the special names table. First, to get the names, and then by determining if the names qualify (by existing in the special names table). This presents a problem in that the refresh process involves deleting the special names, and then inserting the special names. If we delete the special names in this case then when we go to insert we won't know what the names that qualified are. This problem is resolved by using the saved names created global temporary table. This temporary table is used to save the names before they are deleted, and then use those saved names in the name search query.

## Trigger Example – Update PERSON Table

- This Trigger has to do the Following:
  1. See if the Current Name Belongs in Special Names by Looking for the Key in the Special Names Table (Including CUST\_ID)
    - We're Going to use a Trigger Condition
  2. Insert all Entries from Special Names for the Key and the Customer Identifier into a Temporary Table
  3. Delete all Entries from Special Names for the Key and the Customer Identifier
  4. Insert into Special Names by Running the Normal Query for the Key and the Customer Identifier
    - This is Basically Running the Normal Query as a Subselect to an INSERT
    - The Temporary Table Provides the Key Values



The trigger process is in these steps:

In the WHEN condition use the CUST\_ID of the person table to get the names for the CUST\_ID from the names table, then check to see if the name key, including CUST\_ID is in the special names table. Execute the trigger body only if the name key is found.

Execute the trigger body (conditional on #1)

Insert into the saved names table by using the CUST\_ID from the client table to get the name key for the names table, and ultimately the qualifying data from the special names table.

Delete the rows for the special names table using the data from the saved names table

Insert into the special names table the result of the name search query using in a WHERE clause predicate the data from the saved names table

Delete the data from the saved names table

## Step One – See if Name Plus CUST\_ID is “Special”

- The CUST\_ID is Used Because this is an UPDATE and so the Name Should Already Exist in the Special Names Table if it is “Special”
- The PERSON\_NAME Table Contains The Names so We Need to get the Key There

```
CREATE TRIGGER PERSONU1
AFTER UPDATE ON PERSON
REFERENCING OLD AS OLDPERSON
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT 1
              FROM   SPECIAL_NAMES
              WHERE  (CUST_ID, SNDX_NUM, FNM4,
                    DOB_CEN, DOB_YR, DOB_MNTH)
              IN
              (SELECT CUST_ID, SNDX_NUM
                  ,FNM4, DOB_CEN, DOB_YR, DOB_MNTH
              FROM   PERSON_NAME OLDNAME
              WHERE  OLDNAME.CUST_ID = OLDPERSON.CUST_ID)))
BEGIN ATOMIC
.
.
.
```

## Step Two – Save the Special Data

- Since We Will Delete from Special Names We Won't Know What the Special Names are for the Big Query

```
.  
. .  
. .  
BEGIN ATOMIC  
INSERT INTO SAVED_NAMES  
  SELECT *  
  FROM   SPECIAL_NAMES  
  WHERE  (CUST_ID, SNDX_NUM  
         ,FNM4, DOB_CEN, DOB_YR, DOB_MNTH)  
  IN  
  (SELECT CUST_ID, SNDX_NUM  
   ,FNM4, DOB_CEN, DOB_YR, DOB_MNTH  
   FROM   PERSON_NAME OLDNAME  
   WHERE  OLDNAME.CUST = OLDPERSON.CUST_ID);  
. . .
```

Created Temporary Table



## Step Three – Delete the Special Names for This Person

- We can Use the Data We Just Inserted to do the Delete

```
.  
. .  
. .  
DELETE FROM SPECIAL_NAMES  
WHERE (CUST_ID, SNDX_NUM  
      ,FNM4, DOB_CEN, DOB_YR, DOB_MNTH)  
IN  
  (SELECT CUST_ID, SNDX_NUM  
   ,FNM4, DOB_CEN, DOB_YR, DOB_MNTH  
   FROM   SAVED_NAMES);  
. . .
```

Created Temporary  
Table

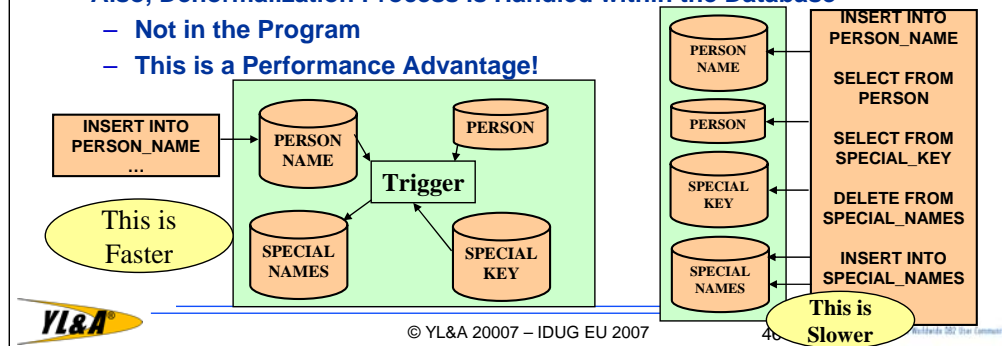
## Step Four – Insert the Denormalized Data into Special Names

- Once Again we Use the Saved Key Information to Refresh the Special Names Table

```
.  
. .  
. .  
INSERT INTO SPECIAL_NAMES  
SELECT A.CUST_ID, B_SNDX_NUM, ..., SEX_CDE, ...  
. .  
. .  
FROM  
    PERSON A  
INNER JOIN  
    PERSON_NAME B  
ON A.CUST_ID = B.CUST_ID  
WHERE (CUST_ID, SNDX_NUM, FNM4, DOB_CEN, DOB_YR, DOB_MNTH)  
    IN  
    (SELECT CUST_ID, SNDX_NUM  
     ,FNM4, DOB_CEN, DOB_YR, DOB_MNTH  
     FROM SAVED_NAMES).  
. .  
; END!
```

## Result of Trigger Usage

- **Special Names Table Allows Us to Meet our Service Level Agreement**
- **Application Does Not have to Update Special Names Table**
  - Triggers Take Care of That
  - No Extra Programming
- **When Name Search Process is Eventually Rewritten**
  - We DROP the Name Search Table
  - We DROP the Triggers
  - Not One Line of a Program Has to Change
  - Months of Programming Changes Avoided!
- **Also, Denormalization Process is Handled within the Database**
  - Not in the Program
  - This is a Performance Advantage!



So, we were able to create the partially denormalized saved names table, and have the table be maintained automatically using triggers. This saved a significant amount of application programming, and when the name search process is rewritten then the special names table, and the triggers, can be dropped with no impact whatsoever to the update application. Brilliant!!!!

## Summary

---

- **Triggers are a Good Thing if Used Properly**
  - **Replace Complicated Application Programming**
  - **Keep Data Intensive Logic Inside the Database**
    - **Performance Advantage**
  - **Solve Complex Problems Quickly**
  - **Don't Need the Logic Anymore?**
    - **DROP the Trigger**
    - **Instant Change!**
  - **Easy and Fun to Use!**
- **Trigger Maintenance Belongs to Both Application Programmers and DBAs**
  - **You Need a Strong Agreement Between These Groups**
  - **You Need to Track Trigger and Table Relationships**

5-9 November  
Athens Hilton  
Athens, Greece

Session: D09

**Effective Uses of DB2 Triggers**

IDUG® 2007  
Europe

Dan Luksetich  
YLA  
[Dan\\_Luksetich@ylassoc.com](mailto:Dan_Luksetich@ylassoc.com)

7 November 2007 11:00-12:00

IDUG  
The Worldwide DB2 User Community

Platform: All Platforms

GoFurther



Daniel Luksetich is a consultant for YL&A, the leader in large high volume DB2 database design and tuning.

He works as a DBA, application architect, presenter, author, and teacher. Dan has been in the information technology business for over 22 years, and has worked with DB2 for over 17 years. He has been a COBOL and BAL programmer, DB2 system programmer, DB2 DBA, and DB2 application architect. His experience includes major implementations on z/OS, AIX, and Linux environments.

Dan's experience includes: Application design and architecture, Database administration, Complex SQL, SQL tuning, DB2 performance audits, Replication, Disaster recovery, Stored procedures, UDFs, and triggers. Dan works 8-16 hours a day, everyday, on some of the largest and most complex DB2 implementations in the world. He is a certified DB2 DBA and application developer, and the author of several DB2 related articles as well as co-author of the DB2 9 for z/OS Certification Guide.