



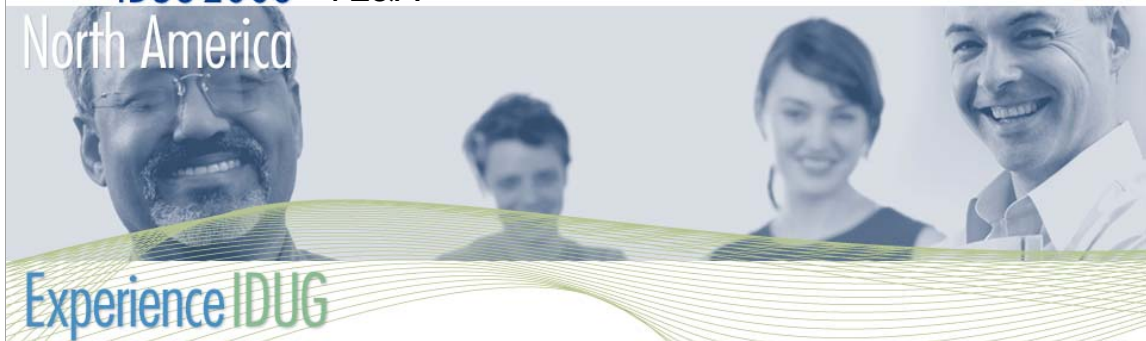
Session: B10

Desperate Table Designs

IDUG 2008

North America

Susan Lawson and Dan Luksetich
YL&A



Susan Lawson is an internationally recognized consultant and lecturer with a strong background in system and database administration. She currently works with several large clients to help development, implement and tune some of the world's largest and most complex DB2 databases and applications. She also performs Performance Audits for many clients to help reduce costs through proper performance tuning. Her other activities have included authoring articles, presenting at user group meetings, and authoring white papers. She is an IBM GOLD Consultant for DB2 and z/Series, and has authored the IBM 'DB2 for z/OS V8 DBA Certification Guide', 'DB2 for z/OS V7 Application Programming Certification Guide' and 'DB2 9 for z/OS DBA Certification Guide' - 2008. She is also the co-author of several DB2 books including 'DB2 High Performance Design and Tuning' and 'DB2 Answers'.

Abstract

This presentation covers new, bold, creative solutions to achieve high availability and high performance. New challenges mean thinking outside the old rules. We also look at how to synergize creative table designs with applications to achieve our goals.

Objectives:

- **Discuss some new innovative ways to create tables**
- **Discuss new challenges and opportunities for index design**
- **Discuss how to integrate designs with applications for best performance and availability**
- **Discuss how to use new features of DB2 to solve problems**
- **Discuss ways to think differently about designs and see examples from real implementations**

This presentation was developed by Susan Lawson and Dan Luksetich of YLA. They can be reached at Susan_Lawson@ylassoc.com and Dan_Luksetich@ylassoc.com respectively.





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 1999-2008, YL&A, All rights reserved.



© YL&A 1999-2008

3

Dan Luksetich is a senior DB2 DBA. 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 18 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.

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.



No animals were harmed during testing

- **24X7 Does Not Mean 24X7**
 - It Means Hiding Your Outages
- **Providing for Logical Recovery from Data Corruption**
- **Partial Denormalization for Performance**
 - Indicator Columns for “Denormalization Light”
 - “Full Partial Denormalization”???



24X7 Does Not Mean 24X7

**Hiding Your Outages are Key to
Apparent High Availability**

High Availability Requirement for Summary Table

■ High Profile Retailer

- Online Sales
- 24X7 Television Show

■ Management Gets a Summary Every 15 Minutes

- How Many Orders are being Taken
- How Many Credit Cards have been Processed
- How Many Products placed in Boxes
- How Many Boxes placed on Trucks
- How Many Trucks in Shipment
- How Many Products Delivered to Customers

■ The “Big Guy” Has a Screen in His Office

- Information Constantly Displayed
- Screen Cannot Go Blank!



In one situation the user needed a high availability table. This table had to contain a significant amount of near real-time information that was derived from many other tables.

The information absolutely had to be available all the time. No outages would be tolerated. How do you make a table that is always available? You don't!

High Availability Summary Table Challenge

■ Problem

- **Summary Table Cannot be Populated in 15 Minutes using SQL**
- **Data can be Gathered in Less than 15 Minutes via Summary Query Unload**
- **Complicated Summary SQL Joins make Triggers Undesirable**
- **LOAD REPLACE of Summary Table Data Results in an Outage**

■ Solution

- **Two Summary Tables and Switching Logic**
 - **Application Reads the Active Table**
 - **LOAD Replaces Inactive Table**

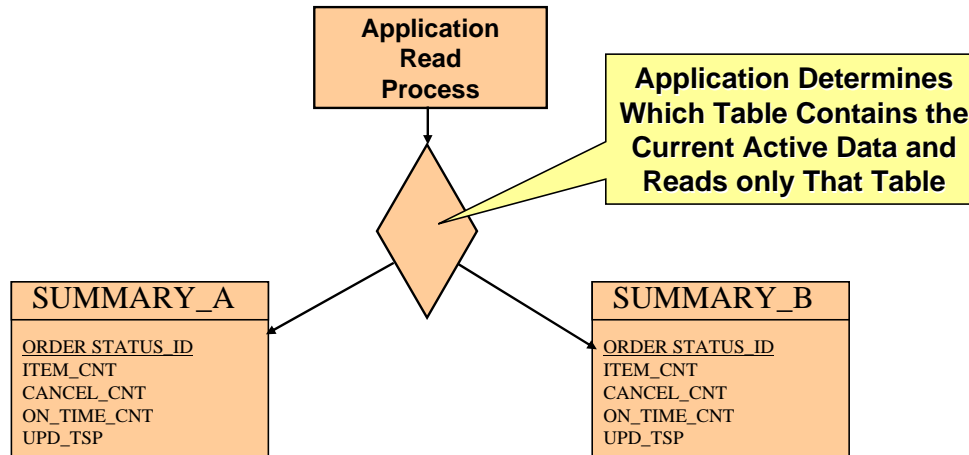


The challenge with this table was that it could be refreshed with SQL or with a load. However, the data had to be refreshed every 15 minutes. The SQL statements required were not fast enough to do the refresh, and data could be modified and incomplete during a read operation. A DB2 load could replace all the data quickly, and replace it in one shot. However, the table would not be available during the load and that was not acceptable.

Table Switching Example

- **Table Switching is an Application Based High Availability Solution**

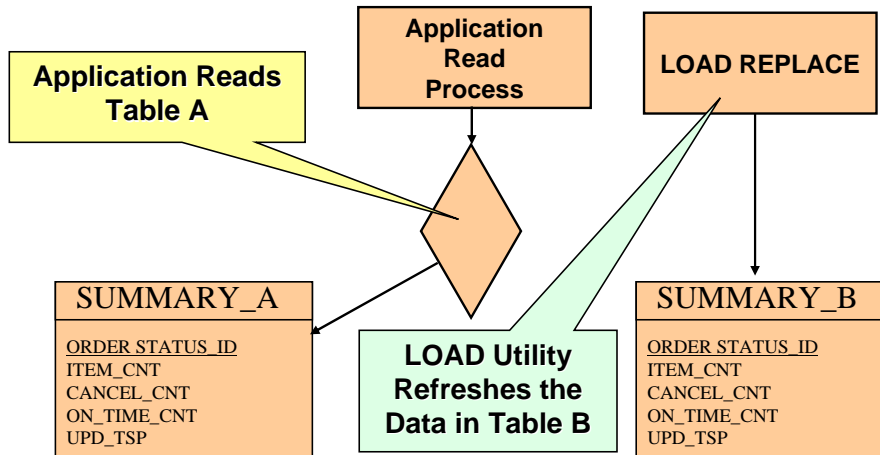
- Very Flexible
- Has to be Coded in Application
- Can Use DB2 Database Features to Simplify Application Design



The solution is to have two tables. One table would be refreshed while the other table is read. The application would have to determine which table is the active table, and read that one.

Table A is Active

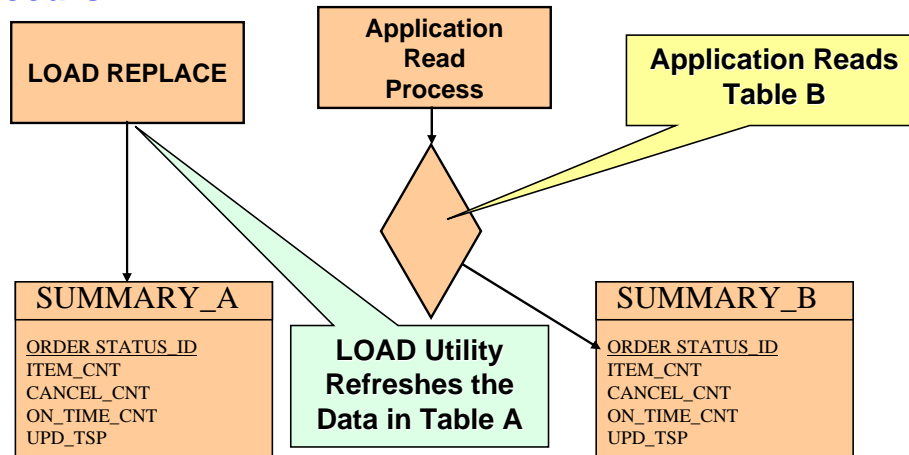
- **When Table A is Active**
 - The Application Reads Table A
 - The LOAD Utility Replaces the Data in Table B
- **The Application is “On Line” While Refresh Occurs**



When Table A is the active table the application would read it. During that time Table B would be loaded with the next refresh of the data.

Table B is Active

- **When Table B is Active**
 - The Application Reads Table B
 - The LOAD Utility Replaces the Data in Table A
- **Again, the Application is “On Line” While Refresh Occurs**



Likewise, when table B is active then the application reads that table, and table A is refreshed with a LOAD.

Table Switch Logic

■ Table Switching Logic can be Application Based or SQL Based

—We Prefer SQL Based for Maximum Flexibility and Performance

- Automated or Manual Switch Possible
- External Switch Possible
- Minimal Trips to the Database for Query
 - Also, No Query Timing Issues!

■ Switching Controlled by DB2 Table



The application has to determine which table is active. This is easily accomplished with the addition of a special DB2 table used to switch the application to which ever table is active.

Switch Table

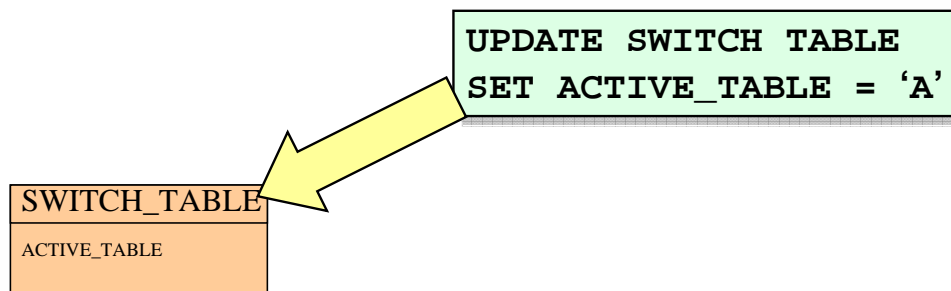
■ The Switch Table Facilitates the Table Switching

—One Column, Two Values

- “A” – Table A is Active
- “B” – Table B is Active

■ This Allows for Flexible Control of Active Table

—Column Can be set by Application or Manually



The switching table is a very simple design. One row, one column. The columns simply contains a value that indicates which table is active.

By using a DB2 table to control the switching the control can be very dynamic, and set by an application process or manually by a user or developer.

LEFT OUTER JOIN – During Join Predicates

```

SELECT PROJNO, PROJNAME, P.DEPTNO, DEPTNAME
FROM   PROJECT P LEFT OUTER JOIN DEPARTMENT D
ON     P.DEPTNO = D.DEPTNO
AND    P.DEPTNO = 'D01'
    
```

PROJNO	PROJNAME	DEPTNO
AD3100	ADMIN SERVICES	D01
IF1000	QUERY SERVICES	C01
IF2000	USER EDUCATION	E01
MA2100	WELD LINE AUTOMATION	D01
PL2100	WELD LINE PLANNING	B01

DEPTNO	DEPTNAME
A00	SPIFFY COMPUTER SERVICE DIV.
B01	PLANNING
C01	INFORMATION CENTER
D01	DEVELOPMENT CENTER

Preserved Row Table

NULL Supplying Table

PROJNO	PROJNAME	DEPTNO	DEPTNAME
AD3100	ADMIN SERVICES	D01	DEVELOPMENT CENTER
IF1000	QUERY SERVICES	C01	-
IF2000	USER EDUCATION	E01	-
MA2100	WELD LINE AUTOMATION	D01	DEVELOPMENT CENTER
PL2100	WELD LINE PLANNING	B01	-



Our design is going to take advantage of the “during join” predicate. This type of predicate will allow the switching to occur in a single SQL statement.

ON clause, or during join predicates do not limit the resultant rows which are returned, only which rows are joined.

In this example, since there are no WHERE clause predicates to limit the result, all rows of the preserved row table are returned. But the ON clause dictates that the join only occurs when P.DEPTNO = 'D01'. When the ON clause is false (ie. P.DEPTNO <> 'D01'), then the row is supplied NULLs for those columns selected from the NULL supplying table.

Switching Inside a Query

■ Switching Inside a Query

- Utilizes a During-Join Predicate
- Reduces Application Logic
- Reduces Potential Outages
 - Read of Switch and Read of Active Table in One Statement Instead of Two

```
SELECT COALESCE(A.ORDER_STATUS_ID, B.ORDER_STATUS_ID)
      ,COALESCE(A.ITEM_CNT, B.ITEM_CNT)
      ,COALESCE(A.CANCEL_CNT, B.CANCEL_CNT)
      ,COALESCE(A.ON_TIME_CNT, B.ON_TIME_CNT)
FROM   SWITCH_TABLE AS SW
LEFT JOIN
      SUMMARY_A AS A
ON SW.ACTIVE_TABLE = 'A'
LEFT JOIN
      SUMMARY_B AS B
ON SW.ACTIVE_TABLE = 'B';
```

The first non-null value is returned. The outer join will supply nulls for the table that is not "active"

SUMMARY_A will only return the summary data when this join predicate is true

SUMMARY_B will only return the summary data when this join predicate is true



Here is our single query to return the data from the active table. The switch table drives the query and is the preserved row table. The other two tables are both null-supplying tables. The value of the ACTIVE_TABLE column actually controls which summary table is physically accessed within the query. So, only the table that is active is accessed, and the other table supplies nulls.

The COALESCE function will return the first non null value. So, each invocation of the function will return the data from the active table since the other table supplies nulls.

Table Switching Using Clone Support in DB2 9

- **DB2 9 simplifies this type of high availability applications**
 - A clone can be created for a table
 - The EXCHANGE statement does the switch
- **The refresh process is simplified**
 - The clone table is truncated
 - Then fresh data is inserted
 - Then the exchange – online LOAD REPLACE!

```
ALTER TABLE SUMMARY_A  
ADD CLONE SUMMARY_B;
```

```
SELECT A.ORDER_STATUS_ID,  
A.ITEM_CNT, A.CANCEL_CNT,  
A.ON_TIME_CNT  
FROM SUMMARY_A AS A;
```

Returns the data from the base table. No complex SQL required!

```
TRUNCATE SUMMARY_B;  
INSERT INTO SUMMARY_B  
SELECT ....;  
EXCHANGE DATA BETWEEN  
SUMMARY_A AND SUMMARY_B;  
COMMIT;
```

This script refreshes the data in the clone table, and then exchanges the clone and base tables. Now the base has become the clone and the clone the base! No outage! No switch table required.

This type of functionality can be greatly enhanced with DB2 9.

In this modified design the switch table is eliminated, and table switching is controlled by DDL. The clone table is inserted into while the base table is accessed. Once the inserts are complete the data can be exchanged, instantly refreshing the base table. Then a mass delete and insert to the clone can start again.

***Providing for Logical Recovery
from Data Corruption***

**Remove Bad Data Instantly
Without an Outage**

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”!!!**

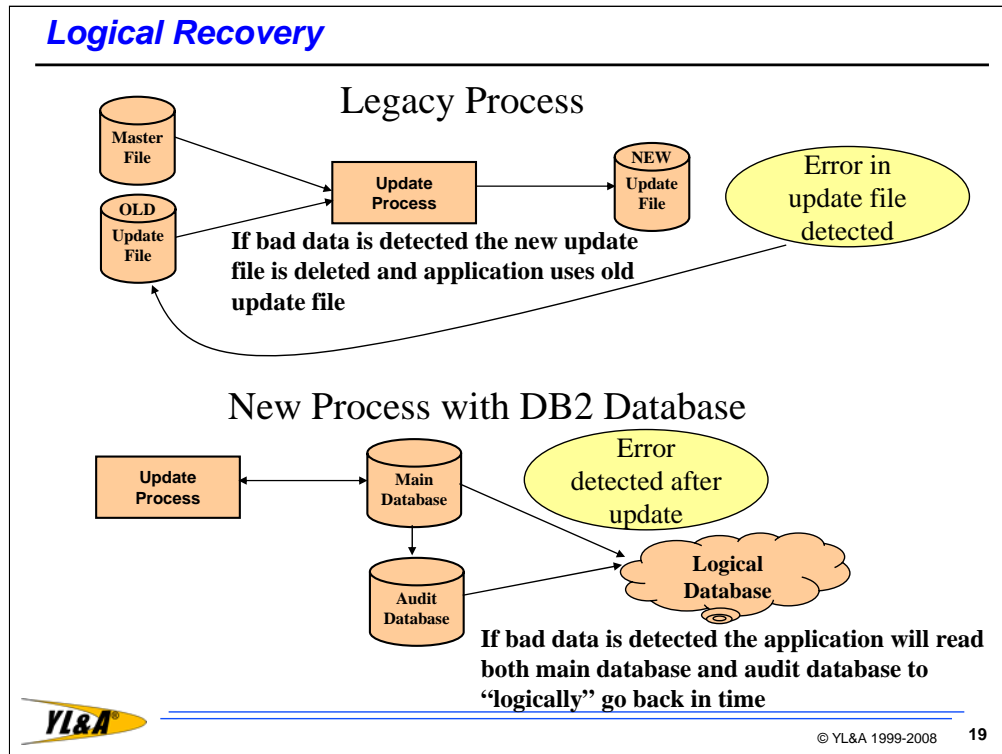


In one situation in which we were migrating a legacy data store from a flat file base technology to DB2 we had to provide legacy functionality within the DB2 database. In this situation that legacy functionality was a fast recovery process from application data corruption.

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 the 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).

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

■ Pros

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

■ Cons

- Additional Complexity to Database
 - Especially for Migrations, Changes, and Tests
- More Overhead for the Update Processes
 - Trigger Invocation
 - Inserts into Audit Tables
- Potential Availability Impact
 - Application Dependent Upon Audit Tables for Normal Operations



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

Logical Recovery Table

■ The Logical Recovery Table is the Heart of the Logical Recovery

— Initiates a Logical Recovery

- Normally Empty
- Timestamp Inserted to Indicate Logical Recovery
 - The Timestamp is the “As Of” Time

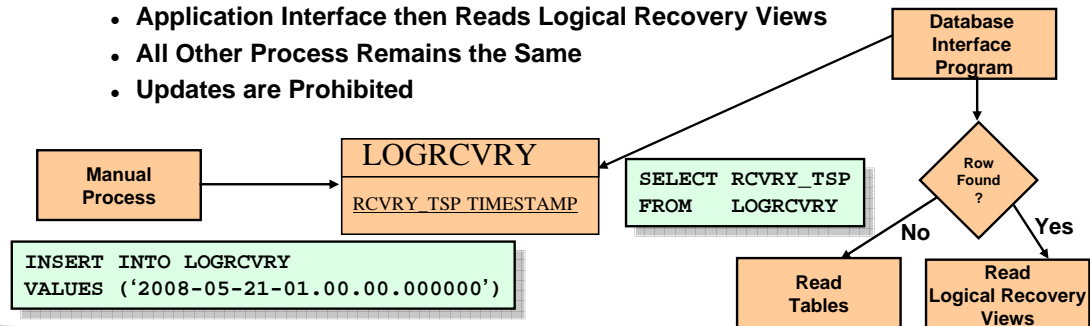
— Objective of Logical Recovery

- All Table Access Data as it Appeared as of the “As Of” Time
- Keep Availability 100% During a Backout Process

■ Database Interface Program Always Read Logical Recovery Table

— It a Row Exists then Logical Recovery Mode is Activated

- Application Interface then Reads Logical Recovery Views
- All Other Process Remains the Same
- Updates are Prohibited

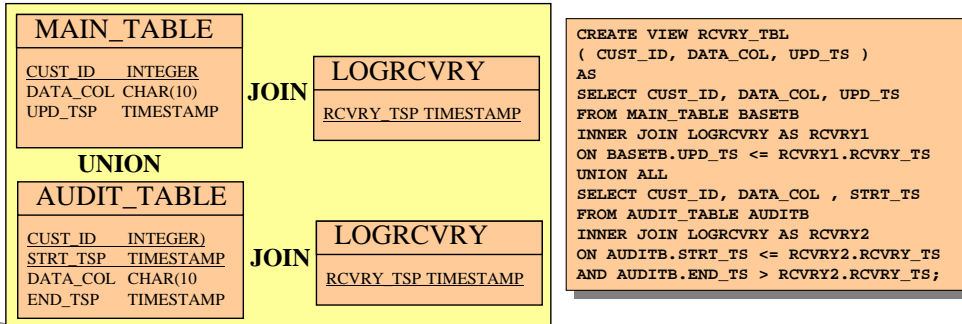


A logical recovery table is created. This table is empty almost all of the time. However, if data corruption is detected then the logical recovery table is populated manually with a timestamp. This timestamp is simply a point in time that is after the last valid update and before any invalid update.

This is an all or nothing proposition. So, if there are many applications and only one corrupted data then this will not. Although a more advanced logical recovery table design could recognize individual applications.

Logical Recovery

- **Each Table in the Database Has a Corresponding View**
 - The View UNIONS each Main Table and its Audit Table
 - The Tables are Joined to the Logical Recovery Table
- **During Logical Recovery Data Comes from Either the Main Table or Audit Table Dependent upon the Logical Recovery Timestamp**
 - If Last Update to the Base Table was Prior to the Recovery Timestamp then it is Used
 - If the Logical Recovery Timestamp is During the Active Period of the Audit Data then that Audit Table Data is Used

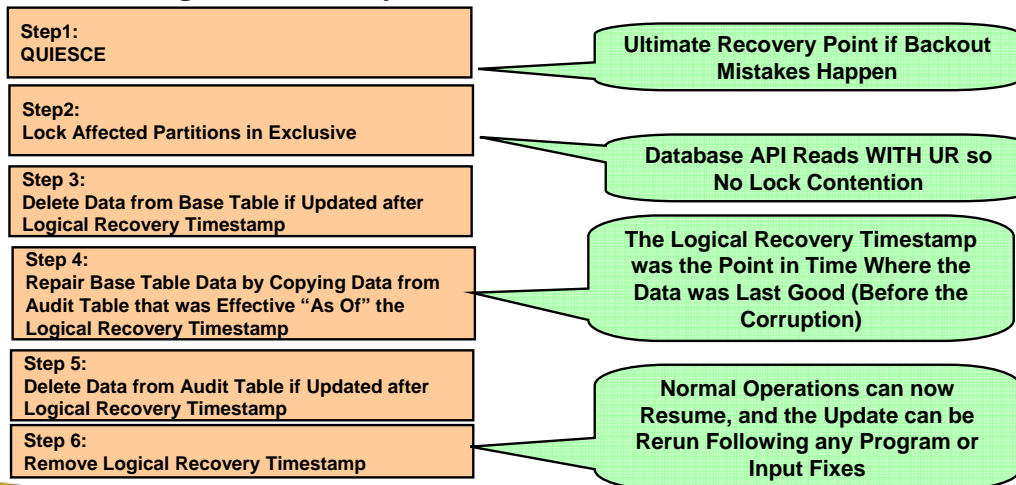


Views have been established for each base table and audit table combination. These are known as the *logical recovery views*. There is a logical recovery table that controls the logical recovery process. This logical recovery table contains a segment, subsegment, and a timestamp that indicates the point 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.

Logical Recovery Backout Process

■ Once the Database is in Logical Recovery Mode the Bad Data Must be Backed Out and Old Data Restored

- This is Done While the Database is Still Online
- Backout Scripts are Employed
- The Logical Recovery Table Drives the Process



Once the application readers are in logical recovery mode then the bad data can be removed from the base tables, and replaced by the old data in the audit tables. Backout scripts can be written to facilitate this process.

The backout scripts can be run while applications have full access. Data will still be viewed as of the logical recovery timestamp.

Once the backout process is finished the logical recovery timestamp can be deleted, and all returns to normal. All traces of the bad data have been removed from the base and audit tables.

Impacts of Logical Recovery Mode

- **Read Performance is Seriously Degraded During a Logical Recovery**
- **Additional Complexity is Introduced**
 - For Update Processing
 - For any Readers
 - When Going in or Coming Out of Logical Recovery Mode
- **Benefits**
 - Ability to Surgically Recover
 - Works Like Magic!
 - Full Availability During a Backout
 - Very Little Additional Application Programming



***Partial Denormalization for
Performance***

**Indicator Columns for
“Denormalization Light”**

The Situation

- **Migration from Legacy Application to DB2 Database**

- Single Record Design to Several Tables
- Performance Expected to be “Same or Better than Legacy”

- **Normalized Design Important**

- Great for Future Applications
- Must be Retained for Business Flexibility

- **Performance is Unacceptable**

- Yes, it Does Take Longer to Read Many DB2 Tables Versus one VSAM File



In many legacy migration situations you are expected to convert single record systems into highly available high performing DB2 relational databases. In many of these situations the management wants all the benefits of a relational database; flexibility, adaptability, availability, without any of the performance impacts.

The first attempt at any design should be to build the database according to the business requirements. Then see if it can performance to expectations, or at least acceptable.

If performance is unacceptable then measures can be taken to improve it. The last choice in performance tuning should be serious database changes.

The Solution

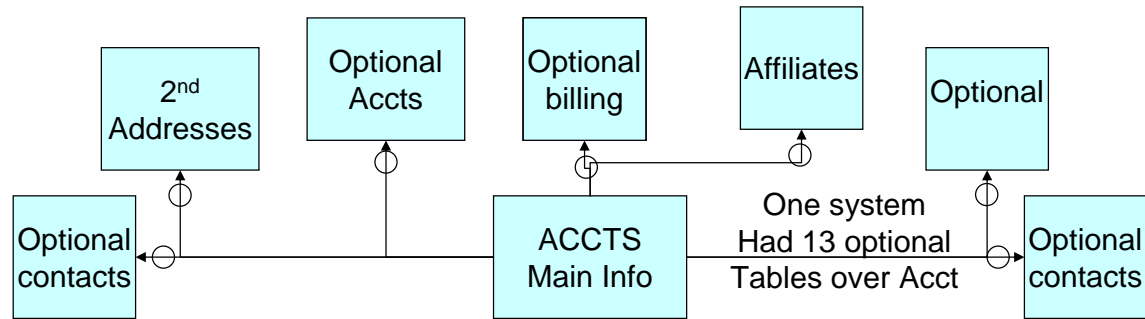
- **Do Not Denormalize**
 - Normalization is One Reason for the Move to a Relational Database
 - Future Development will be More Expensive
- **Try Other Solutions**
 - Increase Buffers
 - Minimize Readers
 - Add Special Codes from Legacy to Request Limited Information
 - Add or Modify Indexes
 - Make Frequent Queries Index Only
 - Modify Clustering to Match Major Processes
 - And Many Others!
- **Normal Solutions Don't Succeed?**
 - Try Indicator Columns
 - AKA "Denormalization Light"
 - This Once Again Takes Advantage of During Join Predicates



In some situations where many attempts at performance improvements still do not bring the performance to an acceptable level, some denormalization may need to be done. However, denormalization should always be a last resort.

In these cases you should look at perhaps be looking at indicator columns in parent tables rather than full denormalization.

Optional Tables: Original Design



Original table design

Account no	names	addr	details
------------	-------	------	---------

```

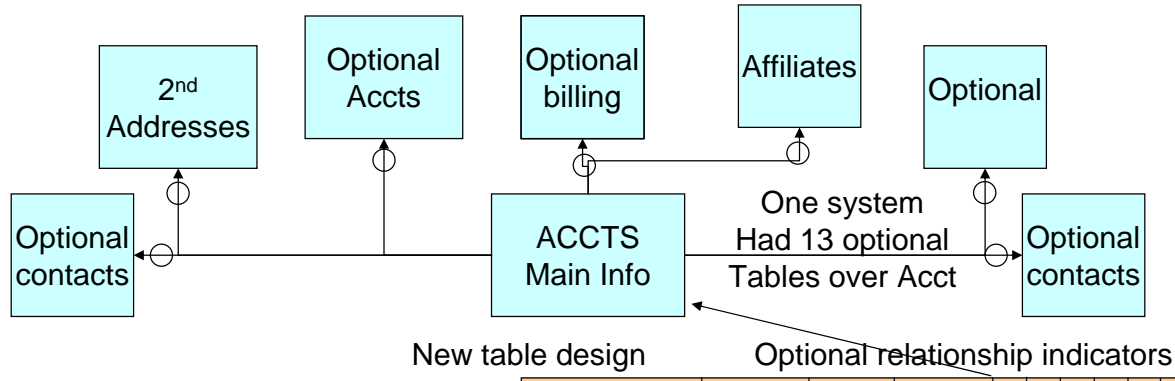
SELECT columns
FROM ACCTS A LEFT JOIN OPT1 O1
ON   A.ACCT_NO = O1.ACCT_NO
LEFT JOIN OPT2 O2
ON   A.ACCT_NO = O2.ACCT_NO
.....
WHERE A.ACCT_NO = 1
    
```

DB2 must join to the optional tables. In many cases the join will not result in a match.



In this example we have an account table with many optional child tables. The legacy application has to read all of these tables in order to recreate the original data (the single record). Even if none of the child tables has data DB2 still has to probe each of them to find out. This may introduce a lot of additional I/O, even when there is no data.

Optional Tables: New Design



```

SELECT columns
FROM ACCTS A LEFT JOIN OPT1 O1
ON  A.ACCT_NO = O1.ACCT_NO
AND  A.FLAG1 = 'Y'
LEFT JOIN OPT2 O2
ON  A.ACCT_NO = O2.ACCT_NO
AND  A.FLAG2 = 'Y'
.....
WHERE A.ACCT_NO = 1
    
```

Account no	names	addr	details	X	X	X	X
------------	-------	------	---------	---	---	---	---

DB2 will only join to optional tables when ON clause is true. Avoids unnecessary joins when optional data does not exist.



Indicator columns can be added to the parent table that indicate the presence of data in the corresponding child tables. A SQL join can utilize a during join predicate. This will cause DB2 to avoid accessing the child tables if the corresponding indicator columns indicates no data present. The result can be a significant reduction in I/O, and improved query performance.

- **Readers Must Use Indicator Columns for Performance**
 - **In During Join Predicates**
 - **DB2 Join Will almost always Outperform a Program Join**
 - **In IF-THEN Logic After Initial Read**
- **Updaters Must Care for Indicator Columns**
 - **Broken Relationships Between Indicators and Tables can Result in Anomalies**
 - **Additional Performance Impact to Updaters to Maintain Indicators**



There is a cost with the indicator columns. Any updates will have to maintain the columns in order to maintain integrity and performance. The update processes will be impacted by this additional maintenance so that impact should be monitored.

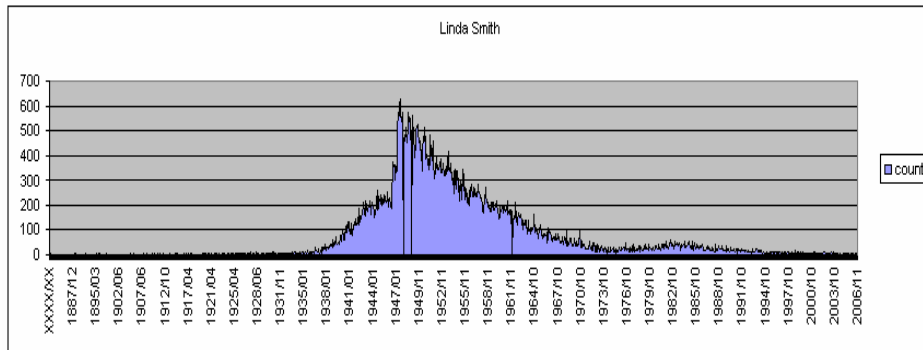
Also, any future processes may have to take the indicator columns into consideration. This may force those columns to remain in existence long into the future.

***Partial Denormalization for
Performance***

**Full Partial Denormalization
AKA “The Pineapple Martini Design”**

The Situation

- Migration from 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!!!



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 Linda Smith July 1947 proved to be a challenge to performance, as that was the most popular name/date combination in history.

The Situation

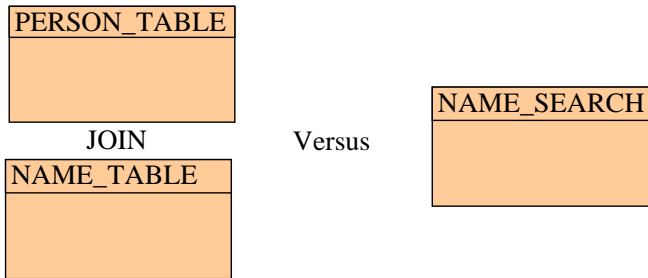
- **Name Search is Driven by a Simple Key**
 - Soundex
 - First Four Characters of First Name
 - Year and Month of Birth
- **Name Search is Fast for Uncommon Names**
 - L232, DANI, 1963, 10 is less than 100ms
 - There is Only 1 Dan Luksetich in Existence in the USA!
 - L250,SUSA, <content edited>, 04 is less than 100ms
 - Only 1 Susan Lawson Born during that Month!
 - S530,LIND, 1947, 07 is at least 3.5 seconds
 - Over 600 Linda Smiths Born during that Month!
- **Name Searches over Multiple Months for Common Names Took up to 10 Minutes!**
 - This Completely Blew Our SLA
 - We Could Not Change the Key!



Name searches were quite simple, and based upon a simple key. The vast majority of searches in our database were very fast since many of the names are fairly unique. However, name searches for more common names were extremely slow, and did not meet our SLA.

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 (Invented over a Pineapple Martini)

- **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?

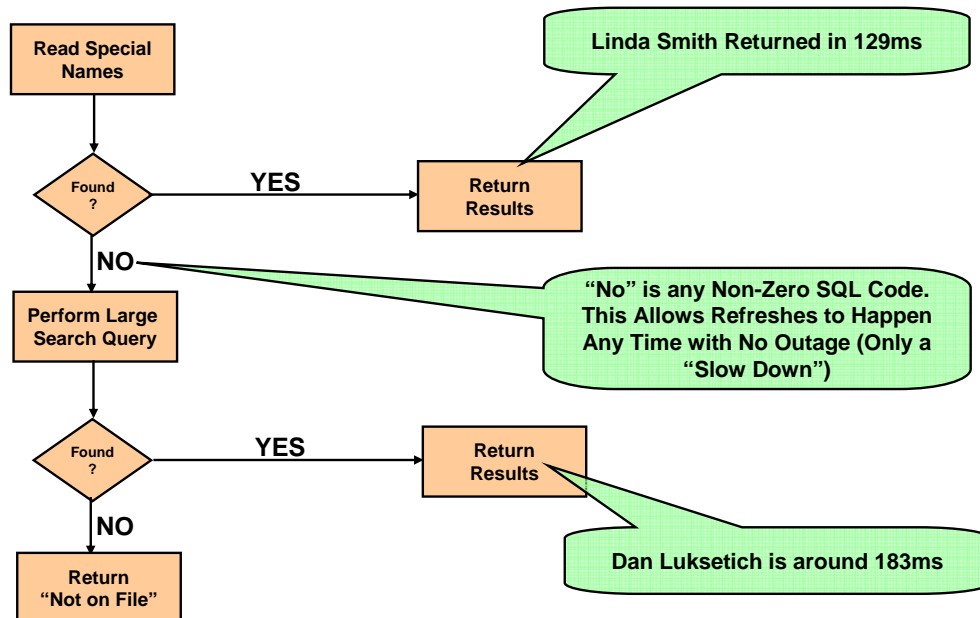


A theory was established that if we can put only the “special names”, those being the most common names, into a denormalized table then the table could be relatively small and the index could contain all table columns. The name search query would read the special names table first, and return the names 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.

New API Access for Name Search

■ The Database API Program was Modified to Check “Special Names” First



YL&A

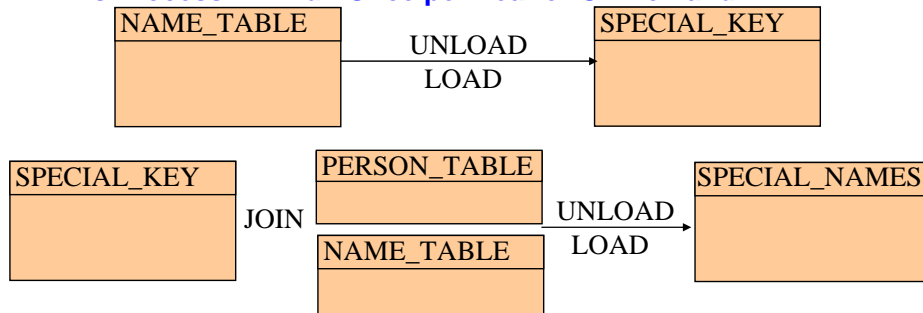
The name search API was modified slightly. This modification was to read the special names table always first. If the result of the search was found on special names then that result was returned to the caller. If a non-zero return was encountered then the regular name search query was run.

This allowed the vast majority of common names to be returned extremely quickly, while only slightly impacting the performance of the rest of the name searches. The overall result was acceptable performance for all name searches.

The API continued with the regular search if the special names search got any sort of error or SQLCODE 100. This let us take down special names for a LOAD REPLACE without an application outage.

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.

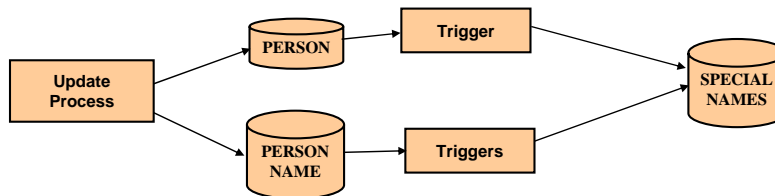
This table served two purposes; first it was used to refresh the special names table. That is, as time goes on additional names might become special (actually common). So, a query was run to see which names newly qualified, and the special keys table was refreshed with the keys for these names. Then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

- **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.

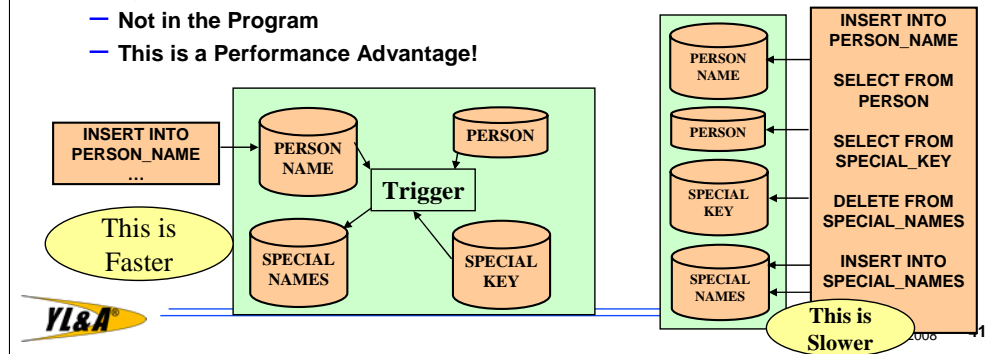
However, on a temporary basis we needed to create the special names table until the name search application can be rewritten. At that time the special names table can be thrown away.

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.

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!



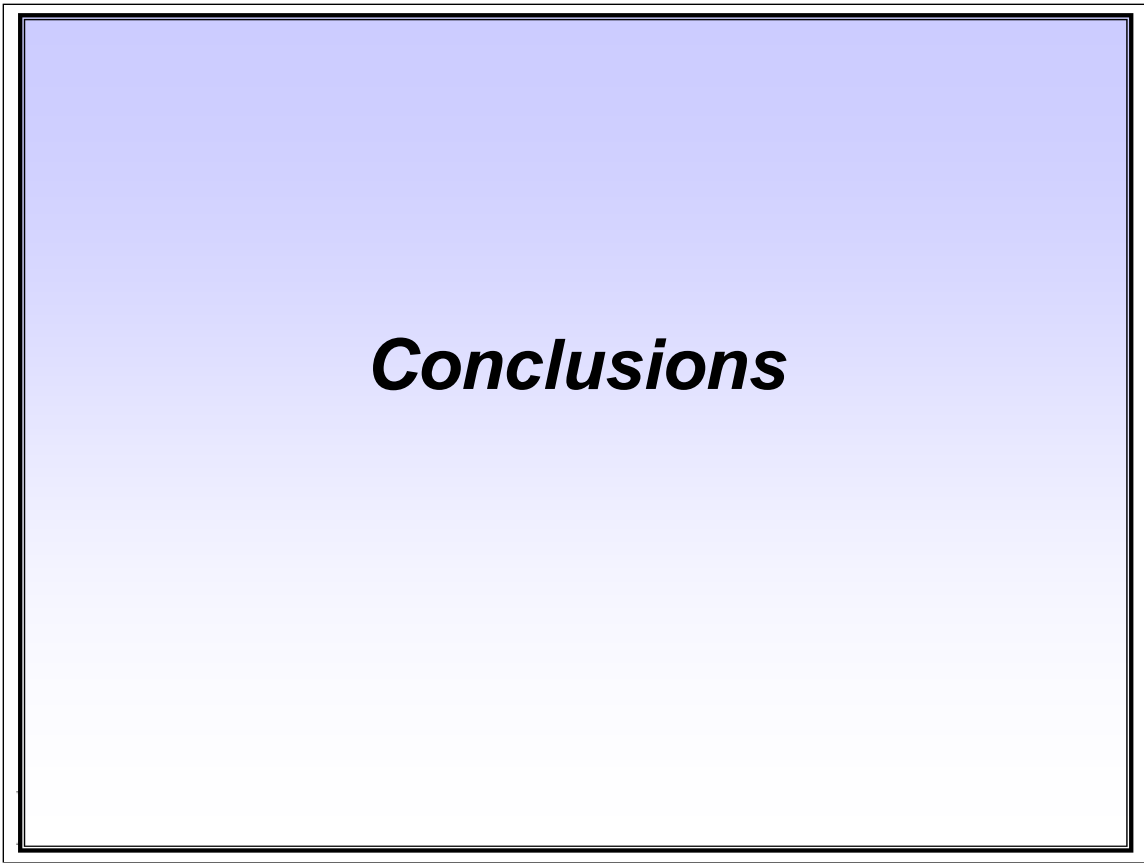
So, we were able to create the partially denormalized saved names table, and have the table be maintained automatically using triggers. This saved a significant amount of application programming, and when the name search process is rewritten then the special names table, and the triggers, can be dropped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

Impacts of Full Partial Denormalization

- **Search Performance Dramatically Improved**
 - Could Not Have Implemented Without This
- **Triggers for Updates**
 - Firing Triggers When Names Change Impacts Performance
 - Additional Complexity to Database Maintenance
 - Trigger Logic is Complicated
- **Need Refresh Processes**
 - Need Scripts/Processes for Periodic Refresh
- **Additional Tables Mean More Database Maintenance**
- **Benefits**
 - Meet SLA
 - Minimum Application Changes
 - Implementation Took Days
 - Can Drop Everything with No Application Impact



There are always trade-offs. We ended up with a slightly slower update process due to the triggers maintaining the special names table. There was also more complexity with table migrations, and additional REORGs. However, the benefits outweighed the cost as we would not have been able to implement our database without this design.



Conclusion

- **There are Always Alternatives**
 - Creative Solutions
 - Maximize Performance
 - Minimize Application Impact
 - Maintain Integrity and Ease of Future Development
- **The Easy Way Out Always Means More Work Later**
 - Don't Denormalize
 - Don't "Work Around" DB2
- **Take Advantage of DB2 Features**
 - During Join Predicates
 - Triggers, UDFs, Stored Procedures
 - UNION
 - Clone Table Support
 - And Many, Many More





Session: B10

Desperate Table Designs

Susan Lawson and Dan Luksetich
YL&A

IDUG 2008
North America

