Tips and Techniques for Data Fixes in a Repository Data Warehouse

Ty Lee
Genentech Inc., South San Francisco, CA

Abstract
A repository data warehouse is popular for statistical analysis and reporting. It is often necessary to update a data warehouse. This paper will discuss tips and techniques for data fixes in a SAS® repository data warehouse. Specifically, the paper will discuss alternatives for deleting, inserting and changing records, combining tables, and validating results. The paper will also demonstrate how to use metadata and macro facility to automate global data fixes in a SAS® repository data warehouse.

To make it more interesting, we will employ the case study approach to explore the following topics:

- The Data Warehouse Environment
- Types of Data Fix
- Efficient Data Updates: A Data Driven Approach
- PROC SQL vs. DATA Step
- Validation Issues
- Where to fix? - SAS® Data Dictionary

Data Warehouse vs. Database
Both database and data warehouse are collections of data. They differ in their purposes and structures.

A relational database (RDB) is typically used for data acquisition and storage, applications known as on-line transaction processing (OLTP). Therefore a RDB has to satisfy certain rules (normalization rules for instance) to guarantee data integrity and update efficiency. A database is transaction oriented and dynamic. As a result, a database is not friendly to end-users for queries and decision support analyses.

A data warehouse, on the other hand, is “a subject-oriented, integrated, time-variant, nonvolatile collection of data in support of management’s decision-making process.” (Building the Data Warehouse, Bill Inmon)

In a repository data warehouse, data structure is defined such that it ensures full integration of data and application in a new environment. Data are pre-processed and relatively stable; tables are merged and de-normalized to facilitate applications known as on-line analytical processing (OLAP). Unlike in a RDB, data redundancy is positive in a data warehouse.

Data Fix in a Nutshell
Basically, there are two types of data fix: for correcting erroneous data (dirty data), or for facilitating analyses (e.g., fixing messy data, logical errors, etc.).
Data entry errors may have occurred in the relational database already, and are transported into the data warehouse. And errors may also occur in the process of building a data warehouse. Ideally, any data changes should be made from the data acquisition source and later re-extracted for the warehouse. This ensures that the data in the RDB and the warehouse are in sync. However, due to the resource concerns for re-building the data warehouse and the time restrictions for conducting analyses, it is advantageous to apply data fixes directly in the data warehouse.

In addition, for analysis purpose, some dummy observations need to be furnished to handle key missing values. Derived variables need to be computed. Tables need to be combined to facilitate integrated analyses. In short, if your data warehouse doesn’t provide meaningful information, you’d better get the problems fixed.

This paper is meant for data fix in its general term. To explore various data fix techniques, we will go through a few examples of data fixes in a SAS® clinical research data warehouse.

A Data Driven Approach

Case 1: In a clinical trial data warehouse, some data errors have been detected in a study termination data set TERM. Also, a few patients’ case report forms were never received at the data entry. Since these CRFs contain key information for the statistical analysis, these records need to be obtained and then added to the TERM data set.

In this case, it’s intuitive to write a program like this:

Program 1.1:

```sas
DATA WAREHOUSE TERM;
SET WAREHOUSE TERM END=LAST;
BY KEY;
IF KEY = 'AAA' THEN DATE=NEWDATE1;
IF KEY = 'BBB' THEN DATE=NEWDATE2;
IF KEY = 'CCC' THEN DATE=NEWDATE3;
IF KEY = 'DDD' THEN DATE=NEWDATE4;
IF KEY = 'EEE' THEN DATE=NEWDATE5;
ELSE DATE;
END;
OUTPUT;
RUN;
```

There are two major problems with this piece of code. First of all, it’s inefficient. The program compares each condition with every single pass through, even though the conditions are mutually exclusive. Secondly, the program is not re-executable. The problem lies in its way of adding observations. If this program is run repeatedly, duplicate records will be added to the data set.

From an efficiency standpoint, a better alternative is to use IF-THEN/ELSE, and arrange these conditions in accordance to the sorting order of the key fields. The objective is to minimize the number of comparisons. The efficiency gain could be quite significant when working with a lot of updates to a large table.

Better still, use a SELECT group rather than a series of IF-THEN/ELSE statements when you have a long series of mutually exclusive conditions. It makes the program easier to read and maintain.

Program 1.2:

```sas
... SELECT KEY;
WHEN ('AAA') DATE=NEWDATE1;
WHEN ('BBB') DATE=NEWDATE2;
WHEN ('CCC') DATE=NEWDATE3;
WHEN ('DDD') DATE=NEWDATE4;
WHEN ('EEE') DATE=NEWDATE5;
END;
... OTHERWISE;
```

The following program using PROC SQL is an equally efficient alternative.

Program 1.3:

```sas
PROC SQL;
UPDATE WAREHOUSE. TERM
SET DATE=
CASE KEY
WHEN 'AAA' THEN 'NEWDATE1';
WHEN 'BBB' THEN 'NEWDATE2';
WHEN 'CCC' THEN 'NEWDATE3';
WHEN 'DDD' THEN 'NEWDATE4';
WHEN 'EEE' THEN 'NEWDATE5';
ELSE DATE;
END;
QUIT;
```

The best solution is to create a transaction data set and use the UPDATE statement to apply these changes to the master data set.

Program 1.4:

```sas
DATA WAREHOUSE TERM;
UPDATE WAREHOUSE TERM FIXES;
BY KEY;
RUN;
```
The transaction data set FIXES only contains changed and additional observations. The output data set contains one observation for each observation in the master data set, as well as any transaction observations that are not matched with the master observations. This program successfully avoids the problem of adding observations repeatedly with multiple runs.

The MERGE statement can also be used to update observations in a file. Their difference in the following two aspects makes MERGE less popular:

- UPDATE statement leaves the values in the master file intact if the corresponding variables have missing values in the transaction file, but the MERGE statement replaces existing values in the master file with missing values from the transaction file.
- The MERGE statement writes duplicate values of the BY variable as two observations in the output data set.

By using UPDATE, we have moved away from the traditional approach, and made our data updates data-driven.

**DELETE and DROP**

The DROP statement excludes variables from an output data set; the DELETE statement excludes observations. In practice, it's more efficient to substitute DROP/DELETE statement with DROP= / KEEP= / WHERE= clause for the input data set. It saves resources for reading in the data that are not needed.

**Combining Tables**

Tables can be combined horizontally (side-by-side) by key values using join operations, or vertically (one on top of the other) using set operations.

Typically, a DATA step with a MERGE / BY statement can perform equivalent operations that SQL join can do. Here are some tips:

- In the SQL procedure, the two common columns used for joining are not overlaid by default. The COALESCE function can be used to overlay two columns.

- PROC SORT / DATA steps are usually more efficient than an SQL join when combining small tables.
- SQL joins are usually more efficient than PROC SORT / DATA steps when combining large unsorted tables.
- SQL joins are more efficient when combining a large, indexed table with a small table.
- When considering efficiency, take CPU time, I/O operations, and most importantly, human efforts into account.

**Case 2:** Patient populations for two studies need to be pooled for analysis.

This requires set operations (or UNION). The following two programs give equivalent output as requested.

**Program 2.1:**

```sql
DATA WAREHSE.ALLDEMO;
  SET WAREHSE.DEMO1;
  WAREHSE.DEMO2;
RUN;
```

**Program 2.2:**

```sql
PROC SQL;
  CREATE TABLE WAREHSE.ALLDEMO
  SELECT *
  FROM WAREHSE.DEMO1;
  OUTER UNION CORR
  SELECT *
  FROM WAREHSE.DEMO2;
QUIT;
```

In fact, for almost all set operators in PROC SQL (e.g., EXCEPT, INTERSECT, UNION, OUTER UNION), you can write an equivalent program by using DATA steps. Here are some tips with regard to which method to use:

- With the DATA step, you can combine many tables at a time, whereas, with SQL, you work on two tables at a time.
- When logical conditions are involved, SQL is generally more convenient than the DATA step equivalent, although SQL also requires more computer resource.
- If multiple DATA steps and PROC SORT procedures are required to perform the task, consider using SQL.
- When using SQL, use the ALL keyword whenever there are no duplicate rows, or duplicate rows are tolerable. This will save one pass of the data.
PROC APPEND

PROC APPEND is the fastest way to perform a simple concatenation of two tables, because only the second table is completely read.

In Case 2, if there is a data set ALLDEMO, which already contains data in DEMO1, a program could be:

Program 2.3:

```
PROC APPEND BASE=WAREHSE.ALLDEMO
      DATA=WAREHSE.DEMO2;
RUN;
```

The drawbacks for PROC APPEND are:

- It can only perform on two data sets at a time.
- Only limited logical conditions can be applied.
- The first data set will be overwritten.

Validation Issues

Unlike a RDB, which typically has the rollback and audit trail facilities, a SAS data warehouse doesn't have the luxury. It is important to verify the results before applying any permanent data fixes to tables in a data warehouse. PROC COMPARE can be a useful tool to perform comparison of two data sets. By default, PROC COMPARE procedure generates the following five reports:

1. data set summary
2. variables summary
3. observation summary
4. values comparison summary
5. value comparison results for all variables judged unequal.

These reports are helpful but lengthy. Options in the PROC COMPARE procedure can be specified to produce customized reports. The following macro prints out the differences for the data set before and after change.

```
* Macro to compare a data set before and after data fix
%MACRO COMPARE(DATASET, KEYS);
PROC COMPARE
      BASE=WAREHSE.DATASET
      COMPARE=TEMP &DATASET
      NOPRINT OUT=RESULT OUTNOEQUAL
      OUTBASE OUTCOMPARE OUTDIFF;
```

A more sophisticated approach is to develop your own audit trail utilities in a data warehouse.

Data "Where" House: Knowing Where to Fix

We have covered the nuts and bolts for fixing specific tables. A more critical issue is knowing which tables to fix. Since redundant data have been populated all over the place in a data warehouse, identifying where to fix could dramatically improve your effectiveness.

Case 3: One investigator was disqualified for the final analysis. Also, a few patients were transferred from one site to another site. Global data fixes are needed to address these changes.

In this case, a data dictionary becomes handy. Data dictionary, also known as metadata, contains data about the data. Dictionary tables are generated at run time and are read-only. The following SAS dictionary tables are available by default:

- **MEMBERS**: contains members of each library and general information about members.
- **TABLES**: detailed information on members of each library.
- **COLUMNS**: information on variables and their attributes.
- **CATALOGS**: catalog entries in libraries.
- **VIEWS**: lists views in libraries.
- **INDEXES**: information on indexes defined for data files.

For the benefit of readers, let's simplify the case and assume that investigator id, investigator site and patient id are common fields for all tables in a clinical research data warehouse. Otherwise, we can still use the same technique, just that we have to take one step further by using the COLUMNS table to identify which tables to fix.
Program 3.1:

* Global changes to data warehouse:
* - exclude Dr. Jones (ID=875)
* - change investigator site for patients

*** Get table names in the library ***;
PROC SQL NOPRINT;
CREATE TABLE TO_CHG AS
  SELECT MENNAME
  FROM DICTIONARY.TABLES
  WHERE LIBNAME = 'WAREHOUSE';
*** The data fix macro ***;
%MACRO CHG_SITE;
DATA TEMP.DATASET;
  SET WAREHOUSE.DATASET
  WHERE=(INV NE 975);
  IF PT IN ('2488', '2486', '2131', '2485')
    AND INVEST=7391
    THEN INVEST=0051;
RUN;
%MEND;
*** Macro to apply global changes ***;
%MACRO CHG_ALL;
%DO i = 1 %TO &SQLOBS;
  DATA NULL;
  SET TO_CHG POINT=1;
  CALL SYMPUT(DATASET, COMPRESS(MENNAME));
  STOP;
  RUN;
%CHG SITE %COMPARE(&DATASET, INV PT)
%MEND;
*** Call the macro for global changes ***;
%CHG_ALL RUN;

This program will automatically update tables in the warehouse one after another, and print the differences before and after change.

A couple of tips:

- The SQL procedure sets up a macro variable SQLOBS, which contains the number of rows processed.
- COMpress in SAS® OPTIONS needs to be set to NO, so that the POINT= option can work.

Conclusion

We live in a fast-paced, dynamic information age. Data fixes in a data warehouse can quickly capture the data changes without going through the laborious data acquisition and extraction procedure. However, due to the characteristics of a data warehouse, special considerations are needed to accommodate fixes in a data warehouse environment. We must consider both efficiency (doing it right) and effectiveness (doing the right thing) of our data fixes.

First of all, a data warehouse may not have the sophisticated audit trail and rollback capabilities that a traditional RDBMS has. Data backup prior to each change is critical. And data validation is essential, even for a simple fix.

Appropriate documentation of data fixes needs to be maintained. This is extremely important in a regulated environment such as a clinical research information system. The documentation should include the original requests, test plan, expected and actual results, implementation procedure, etc. A logging system will help keep track of the data fix requests. A SAS® audit trail facility can also be developed.

The most critical issue about data fixes is how to identify which tables to fix. SAS® data dictionary can be very useful. Knowing where to fix would fundamentally improve the effectiveness of data fixes.

Due to the data redundancy in a data warehouse, efficiency is often a concern for data fix programs. A small efficiency gain in a macro program can be multiplied when applying global fixes. Consider the tradeoff of computer resources and human efforts. After all, efficiency is measured by weighted average of your CPU, I/O and human costs.

Use the data driven approach rather than writing a new program each time. Use the macro facility to generate SAS® program code, instead of writing them explicitly. SAS/ASSIST® can also be utilized as a CASE tool for code generation. All these techniques are geared towards minimizing chances for human errors in data fixes. Let your programs work as hard as you do!
Acknowledgments

The author is grateful to Lindley Frahm, Amy Caron and Lee-Hwa Jane who have reviewed the earlier versions and offered many helpful comments and suggestions.

About the Author

Ty Lee started as a systems analyst at Eli Lilly and Company, during which time he developed a SAS/AF® GUI system which won the company's technological achievement award. He also worked as a Project Analyst at the Center for Biometrics and Database Management, Syntex Corporation. His related area of expertise includes DB2®, Oracle®, SAS/AF® (SCL), SAS/FSP®, SAS/GRAPH®, SAS/STAT®, SAS/ACCESS®, and SAS/ASSIST®.

Ty received his Master's degree in Econometrics & Computer Science, and B.S. in Information Systems.

Author contact:

Biostatistics
Genentech, Inc. Mail Stop 66
460 Point San Bruno Boulevard
South San Francisco, CA 94080-4990

(415) 225-8896
internet: lee.ty@gene.com

® SAS® is the registered trademark of SAS Institute, Inc. of Cary, NC.
® DB2® is the registered trademark of IBM.
® Oracle® is the registered trademark of Oracle Corporation of Redwood City, CA.