

# Who Puts the Data into the Database?

Leslie M. Tierstein,  
W R Systems, Ltd.

## Introduction

Whether they are building a data warehouse, or re-engineering legacy systems, system developers are faced with getting vast amounts of potentially filthy, possibly inconsistent, probably non-normalized data into brand new, squeaky clean Oracle databases. The ultimate deliverable of the warehouse or conversion effort is, of course, the data. But interim deliverables are also needed, in the form of requirements analyses, design specifications, and the actual code for converting the data from its legacy format to the target database structures.

To produce these deliverables, the conversion team needs not only those skills usually associated with a conversion task: SQL\*Loader, SQL (or PL/SQL) programming, and knowledge of legacy database or file systems like COBOL; the team also needs to be able to use the same design and development tools that application developers do -- to produce specifications and generate forms and reports.

This paper documents some of the standards, methodology, and procedures used by the WRS conversion team to convert the data from multiple applications, running on assorted hardware and software platforms, into one integrated database. The development of the target application, a Logistics Management System, was sponsored by a large government agency.

## Requirements Analysis

The requirements analysis for our application included some market research on which commercial off-the-shelf (COTS) product(s), if any, could be used for the conversion effort. The criteria for evaluating products were simple:

- ?? The product had to be integrated with the toolset chosen to develop the application, the Oracle development and design tools and the Designer/2000 repository.
- ?? It had to handle the conversion of many types of input files (including Prime COBOL, Britton-Lee databases, and other proprietary database formats) into Oracle tables.
- ?? It had to run on our hardware (an HP UNIX server with PC clients).
- ?? It had to be affordable.

Unfortunately, no product met these requirements. Commercially available data warehouse front-ends seem to be oriented to mainframe databases and mainframe shops (with mainframe budgets). So, the

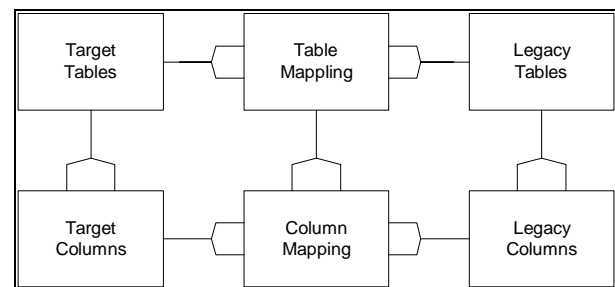
buy or build decision was simple. We had to build tools which integrated with the Oracle toolset to document the data cleansing and conversion specifications, and, where possible, to automate some of the process of implementing the conversion programs.

## Data Model

The Oracle design and development toolset includes a well-documented metamodel for its repository. An analysis of this metamodel indicated that it would have to be extended to incorporate the information needed to design and implement a data cleansing and conversion application.

### Logical Data Model

At the entity level, the logical data model for a data cleansing and conversion process looks simple:



The application's data is stored in **target tables**, each of which has one several **target columns**. The development team is responsible for developing the definitions for these tables.

The legacy data is stored in **legacy tables**. For the sake of consistency, let's call them "tables", although they may actually be Oracle, Progress, or other database tables, flat files, or COBOL files. Each legacy table has one or more **legacy columns**. (Similarly, read "column" for field, row, or any other appropriate nomenclature, based on the data source.)

In a data warehouse or an application which is replacing multiple legacy systems, one target table, almost by definition, receives data from more than one legacy table. However, this may also be true if only one system is being replaced. For example, a non-normalized legacy table will map to (at least) two target tables which contain header and detail information:

	Legacy Table	Target Table 1	Target Table 2
<b>Table :</b>	promis	pr	pr_line
<b>Cols.:</b>	idnum item qtydue dmr	pr_doc_nbr   default_dmr	pr_doc_nbr pr_line_nbr pr_line_qty

As another example, the legacy system could consist of “stovepipe” applications, which contain different versions of the “same” table, that is, with different (improper) subsets of columns. The data must be consolidated into one target table.

	Legacy Table 1	Legacy Table 2	Target Table
<b>Table :</b>	daas_load	erpal_head	unit
<b>Cols.:</b>	service_cd unit_code addr_line  district_cd	unit_code unit_name allow_fact	srvc_pt_cd opfac_nbr unit_nm allow_factor district_cd

So, an intersection table, for **table mapping**, is required.

This analysis also shows the need for a many-to-many **column mapping** between **legacy columns** and **target columns**.

### Implementing the Data Model

We wanted to piggy back this model onto the existing Designer/2000 metamodel, in which the application developers were already specifying table and column definitions.

To store information on the legacy data in the repository:

1. Set up one or more applications for the legacy tables.
2. Define the legacy tables in the appropriate application.
3. Take one of the above applications, or create a new one. This is the **conversion application**. This application will own additional tables needed for the cleansing and conversion effort. It will also own the modules which are needed as part of the conversion effort.
4. Share the target tables and legacy tables into the conversion application.

**One or more applications?** If several systems are being converted, setting up multiple applications allows you to report separately to the owners of each system on your plans for their data. Users are intensely interested in their own data, but not much interested in anyone else’s -- unless they think it will affect theirs.

**Defining the legacy tables.** Getting the legacy table definitions into the repository is an interesting sub-task. The owners of the legacy data have to provide a written or electronic copy of the format of each table to be converted. If this specification resembles -- or can easily be made to resemble -- a SQL CREATE TABLE command, you can create the table in the database and then reverse engineer it into the repository.

### Tracking Table and Column Mappings

To store information on the relationship of the legacy data to the target data structures, you still need to:

1. Set up a means to track the table-to-table mappings.
2. Set up a means to track the column-to-column mappings.

Some options for tracking these mappings have been discussed at previous ODTUG conferences. These approaches concentrated on extending the Designer/2000 repository to hold extra data, setting up new associations, and using matrix reporting to report on these associations.

It wasn’t possible for us to use these approaches. With multiple applications, matrixes can be set up for table-to-table associations, but not for column-to-column associations -- columns are “secondary access elements”, which can’t be shared between applications. We needed more robust data entry and reporting capabilities than those provided by the Matrix Diagrammer. Plus, we had a repository administrator who had been badly burned by trying to use extensions in CASE 5.1.8, and wanted nothing to do with the new version.

So, we opted to supplement the Oracle-provided data entry screens and repository by:

- ?? Defining the repository metamodel in the repository, so these tables and views could be used in generated forms and reports.
- ?? Defining additional mapping tables in the repository, with appropriate constraints to the “base” tables.
- ?? Generating (or coding) forms and reports for entering data into the mapping tables and reporting or querying on all the data in the conversion application’s data model.

Start by defining a table for each view in the repository metamodel that you need to manipulate. This isn't too tedious, since the tables need to include only selected columns. In fact, the sizes of these columns can be changed, to suit your own development standards (as shown in boldface, below).

“Table”	Columns	
cdi_application_systems	appl_system	VARCHAR2(12)
	appl_version	NUMBER(3)
	text_id	NUMBER(38)
sdd_tab_defs	td_table_type	VARCHAR2(5)
	td_comment	VARCHAR2(240)
	td_as_owned_by	NUMBER(38)
	td_table	VARCHAR2(24)
	td_tid	NUMBER(38)
sdd_columns	col_colid	NUMBER(38)
	col_column	VARCHAR2(24)
	col_dtype	VARCHAR2(10)
	col_max_length	NUMBER(6)
	col_nulls	VARCHAR2(10)
	col_dom_reference	NUMBER(38)
	col_sequence_ref	NUMBER(38)
	col_tabid	NUMBER(38)
	col_comment	VARCHAR2(240)
sdd_domains	dom_name	VARCHAR2(24)
	dom_reference	NUMBER(38)

Note to Oracle support: We wrote no code to modify these “tables” -- all changes to their contents were done via standard dialogs and property sheets. (In future efforts, we would investigate using the APIs to implement specialized data entry screens.) Developers used the *comments* field on the Table Definition dialog to enter comments relevant to conversion. We weren't using these fields for anything constructive, anyway. They were populated with text like:

Comment   
 OR:

Comment

When defining the tables, remember to keep the Create? check box set to N(o).

Next, we defined the table- and column-mapping tables:

Table	Columns	
tab_xref	legacy_tid	NUMBER(38)
	target_tid	NUMBER(38)
	comments_txt	VARCHAR2(240)
col_xref	legacy_col_id	NUMBER(38)
	target_col_id	NUMBER(38)
	comments_txt	VARCHAR2(240)
	conv_code_txt	VARCHAR2(240)

The final step in specifying the conversion data model is to set up constraints between the tables, again keeping the Create? check box off. And, to be doubly sure that the Designer/2000 Server Generator doesn't try to create constraints to the repository tables (which, remember, aren't really tables), define the constraints as *Client* only. An extract from a repository-based report on constraints is given below.

Table Name : CDI\_APPLICATION\_SYSTEMS

Type	Constraint Name	Column Name	Join Column
PRIMARY	AP_PK	TEXT_ID	
UNIQUE	AP_NM_UK	APPL_SYSTEM	
UNIQUE	AP_NM_UK	APPL_VERSION	

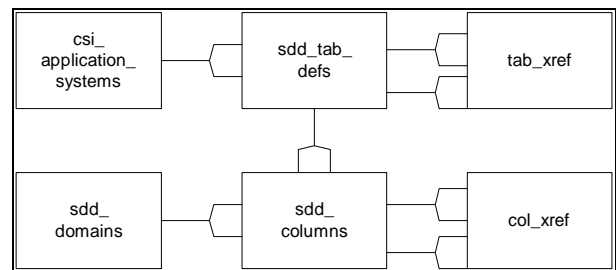
Table Name : SDD\_TAB\_DEFS

Type	Constraint Name	Column Name	Join Column
PRIMARY	TD_PK	TD_TID	
UNIQUE	TD_TABLE_UK	TD_TABLE	
FOREIGN	TD_OWNED_BY	TD_AS_OWNED_BY	TEXT_ID

Table Name : TAB\_XREF

Type	Constraint Name	Column Name	Join Column
PRIMARY	TX_PK	LEGACY_TID	
PRIMARY	TX_PK	TARGET_TID	
FOREIGN	TX_LGC_TAB	LEGACY_TID	TD_TID
FOREIGN	TX_TGT_TAB	TARGET_TID	TD_TID

You end up with a “physical” model that looks like:



To simplify writing reports and queries, you'll want some views, joining the mapping tables to the repository tables:

```
CREATE OR REPLACE VIEW tab_xref_vw
(legacy_tid, legacy_tab_nm,
target_tid, target_tab_nm,
legacy_app_nm
...) AS
(SELECT x.legacy_tid,
t1.td_table
x.target_tid,
t2.td_table,
```

```

        a.appl_system
        ...
FROM tab_xref x,
     sdd_tab_defs t1,
     sdd_tab_defs t2,
     cdi_application_systems a
WHERE x.legacy_tid = t1.td_tid
      AND x.target_tid = t2.td_tid
      AND t1.td_as_owned_by = a.text_id)

CREATE OR REPLACE VIEW col_xref_vw AS
(SELECT x.legacy_colid,
       c1.legacy_col_nm,
       c1.legacy_tab_nm,
       x.target_colid,
       c2.target_col_nm,
       c2.target_tab_nm,
       d.domain_name, c2.col_nulls,
       ...
FROM tab_xref x,
     sdd_columns c1,
     sdd_columns c2,
     sdd_domains d
WHERE x.legacy_colid = c1.col_colid
      AND x.target_colid = c2.col_colid
      AND c2.col_dom_reference =
           d.dom_reference (+)

```

## Processing Requirements

The ultimate product of the cleansing and conversion effort is clean, converted data. In order to produce this deliverable, several interim deliverables are also required:

- ?? Design Reports
- ?? Data Cleansing Reports
- ?? Data Cleansing/Conversion Strategy
- ?? Conversion Code.

## Design Reports

Design reports document the table- and column-level mappings. These reports play the same role as prototypes and repository-based reports for on-line system development: they get preliminary designs in front of the customers, for their review. These design specifications will, in fact, be used to generate additional reports and code as development proceeds.

We developed the following reports:

Report	Description
Legacy to Target Table Overview	Organized by legacy application, a list of the tables in each application, the target table(s) to which each is mapped (if any), and comments on the conversion
Legacy to Target Column Mapping	For each legacy table, all columns (in all target tables) to which the column is mapped, and any comments on the conversion.
Target to Legacy Table Overview	A list of all target tables (organized by application), the legacy table(s) to which each is mapped (if any), and comments on the conversion

Report	Description
Target to Legacy Column Mapping	A 1-to-1 mapping, used mainly by conversion staff, to ensure the completeness of the conversion design and pinpoint any open issues

## Table-Level Overviews

From a user perspective, the table-level design reports ([Legacy to Target Table Overview](#)) verify that all legacy tables are accounted for -- either each table has at least one legacy-target table mapping, or there is a good reason why there is no mapping, as documented in the comment. For example, batch processing tables or temporary tables used to transfer data from one stovepipe application to another are not expected to be converted and should have no mappings. But it is important to document this, so that users realize no data has been overlooked.

From a developer perspective, the table-level design reports ([Target to Legacy Table Overview](#)) verify that all target tables that are expected to hold legacy data are, in fact, receiving it from somewhere. These reports also identify tables which are not expected to receive legacy data -- for example, reference tables used for on-line verification, matrix tables which replace hard-coded IF-THEN-ELSE logic, or tables for data required by new functionality. This report should be distributed to the system developers, so that they can make sure they have data entry screens or SQL scripts to populate the tables that the conversion will not load.

The SELECT command to generate the table-level reports will have to be coded, something like this pseudo-code:

```

{ * tables with mappings * }
SELECT legacy_tab_nm,
       target_table_nm,
       comments_txt
FROM tab_xref_vw
WHERE legacy_app_nm =

UPPER(' &&legacy_app_nm' )
UNION
{ * tables without mappings * }
SELECT legacy_tab_nm, '',
       comments_txt
FROM legacy_tab lt
WHERE legacy_app_nm =

UPPER(' &&legacy_app_nm' )
AND NOT EXISTS
( SELECT 1 FROM tab_xref tm
  WHERE tm.legacy_tab_nm =
        lt.legacy_tab_nm )
ORDER BY 1;

```

## Detail Reports

Two column-level reports were produced -- one from the user (legacy) perspective, and the second from the developer (target) perspective.

The Legacy to Target Column Mapping Report is typically generated for each legacy application. For each legacy table, it shows all columns with all their target column mappings, and any special treatment required. (The special treatment is usually determined through analysis or the cleansing reports discussed below.)

App.: MSAS      Table: sadmstr

Column	Target Table	Target Column	Comments
stkno	inv_item	fsc_cd	SUBSTR(stkno, 1,4)
	inv_loc	fsc_cd	SUBSTR(stkno, 1,4)
stkno	inv_item	niin	SUBSTR(stkno, 5,9)
	inv_item	niin	SUBSTR(stkno, 5,9)
loc	inv_item	loc_cd	
loc	inv_loc	loc_cd	
purp	inv_item	purp_cd	DECODE(purp, NULL, 'N', purp)
cond	inv_item	cond_cd	

The Target to Legacy Column Mapping Report is not distributed to users, but is used by developers. The report is generated for every legacy-target table-level mapping. It is a detailed design specification for cleansing and conversion coding.

Target: inv\_item    Legacy: stkmstr

Target Column	Legacy Column	NULLS?	Domain
fsc_cd	stkno	NOT NULL	
niin	stkno	NOT NULL	
loc_cd	loc	NOT NULL	loc_cd
purp_cd	purp	NOT NULL	purp_cd
from_activ_cd	activ1		activ_cd
to_activ_cd	activ2		activ_cd

## Data Cleansing Reports

Data cleansing reports are produced after a sample of the user data has been loaded. These reports pinpoint the dirty data. In order to write data cleansing reports, you need to know: “What constitutes dirty data, and how does the repository know about it, so it can report on it?”

### Check Scripts

The table mappings and definitions of the target tables in the repository can help you find two types of dirty data:

- ?? If the target column has an associated domain, the value of any legacy column mapped to that column must be in that domain.
- ?? If the target column is defined as NOT NULL, a non-blank value must be supplied for any legacy column mapped to that column.

Since the table mappings as well as the detailed column definitions of the target table are in the repository, a program to diagnose these problems can be generated. We called such programs “check scripts”, since they checked the validity of the legacy data and reported on problems.

The code generator is a SQL\*Plus command file which accepts as input the name of the legacy table being cleansed and generates another SQL\*Plus command file to inspect the contents of the specified table.

The heart of the SQL\*Plus code generator looks like:

```

SPOOL chk_&&legacy_tab_nm\ .sql

SELECT 'SELECT ' || legacy_col_nm ||
      ', count(*) FROM ' ||
      legacy_tab_nm, ' WHERE ' ||
      legacy_col_nm || ' NOT IN ',
      '(SELECT rv_low_value',
      ' FROM cg_ref_codes',
      ' WHERE rv_domain = ' ||
      ' ' || dom_name || ' ' ||
      ' ' ||
      ' GROUP BY ' || legacy_col_nm
      ' ' ;
FROM tab_xref_vw
WHERE legacy_tab_nm='&&legacy_tab_nm'
AND col_dom_reference IS NOT NULL
ORDER BY legacy_col_nm
/
SELECT 'SELECT count(*) ' ||
      'NULL values for ' ||
      legacy_col_nm ||
      ' FROM ' || legacy_tab_nm ||
      ' WHERE ' || legacy_col_nm ||
      ' IS NULL;'
FROM tab_xref_vw
WHERE
legacy_tab_nm='&&legacy_tab_nm'
AND col_nulls = 'NOT NULL'
/
SPOOL OFF

```

Given a mapping like:

Legacy Column	Target Column	NULLS?	Domain Name
cond	cond_cd	NULLS	cond_cd
address1	addr_1_txt	NOT NULL	

The generated program for the legacy table *stkmstr* would be named *chk\_stkmstr.sql* and would contain a series of SQL commands:

```

SELECT cond, count(*)
FROM sadmstr
WHERE cond NOT IN
      (SELECT rv_low_value
      FROM cg_ref_codes
      WHERE rv_domain = 'COND_CD')
GROUP BY cond;
SELECT count(*)
      'NULL values for address1'
FROM sadmstr
WHERE address1 IS NULL;

```

Since the output of a check scripts is aimed at the owners of the data, these reports are always organized by legacy table, rather than target table. Similar

reports, organized by target table, could be used by the programmers as programming aids during the conversion tasks.

### Consistency Checks on Dirty Data

The check scripts do not look at the consistency of the values of data items. For example, they won't reveal that the same vendor name is spelled eight different ways in fifteen records, or that abbreviations are used inconsistently in street addresses. Some data cleansing programs to do jobs like these are available, mostly geared toward cleaning customer address lists.

In building a data warehouse, identifying "identical" data which is spelled differently may be critical to performing dimensional analyses and summaries correctly. But you'll get the same sort of problem with non-normalized data from one source. For example, what if instead of having one vendor record, to which a purchase record had a foreign key constraint, the vendor name were entered into each purchase record (don't laugh)? The way you propose to clean up the data in these special cases will have to be covered in the data cleansing/conversion strategy.

## Data Cleansing/Conversion Strategy

The cleansing/conversion strategy is the basis for the programming standards the developers have to follow. It specifies the default behavior you intend to code when you find dirty data. It may also provide for alternate solutions, to be applied on a table-by-table basis.

The strategy also specifies whose responsibility it is to fix each particular type of dirty data. The users (and their IRM staff) also have to buy into the strategy, since who does what work affects how much work there is for them.

The cleansing/conversion strategy should cover such issues as:

- ?? What do you do with a code value which fails domain verification?
- ?? What do you do with a row that cannot be inserted because it would violate referential integrity constraints?
- ?? How do you go about supplying missing data?
- ?? What do you do with numerics that aren't?
- ?? What do you do with invalid dates?
- ?? What reports will be provided to allow staff to review data that wasn't converted in its original form?
- ?? What means, if any, will be provided for re-processing data that fails to convert?

These topics are considered in detail in our Conversion Standards. This paper can only summarize some of the major issues and document what our decisions were.

### Invalid Values

If a column with code (domain) verification or a referential integrity constraint fails to meet the constraint, what should the conversion code do?

1. Don't convert the row.
2. Convert the row, keeping the invalid value as is.
3. Convert the row, but translate the invalid value into a dummy, instantly recognizable value.
4. Convert the row, but translate the invalid value into another, valid value.

The users hated #1, and rightly so. From their viewpoint, they'd rather have dirty data (and a way to find it), than no data at all. From the programmers' viewpoint, not inserting a row now might have far-reaching consequences later -- any rows in other tables with foreign key constraints to the missing row couldn't be inserted either.

For domains, we generally used solution #2, and gave the users reports on all such data.

For foreign key columns, we used solution #3, choosing a series of question marks (????) as the dummy value. This allowed related rows from other tables to be inserted later.

When possible, and when the users or our cleansing scripts could identify the problems, we substituted another value for an invalid value (solution #4). In simple, low volume cases, this called for the use of a DECODE function in the conversion program. In high volume cases, this called for the creation of a cross-reference table relating the new and old values.

### Missing Data

More problematic was the case of missing data. To cite a few of the more memorable examples:

- ?? The new application called for storing financial obligations and commitments at the line item level, but the legacy systems only stored this information at the document level.
- ?? The new application required an item manager to be assigned to each inventory item, but this functionality was not part of the old system.
- ?? The new database design stated that a vendor name was required; however, the old system only stored the vendor (actually, CAGE - Commercial and Government Entity) code.

Ways of supplying missing data had to be designed and implemented on a case-by-case basis. These included:

- ?? Supplying a default vendor name (for example, “VENDOR 9456” for vendor #9456)
- ?? Giving the users special data entry screens to enter the missing data after conversion had taken place.
- ?? Giving the users special data entry screens to enter additional data (for example, the item manager to assign to each inventory item) before the conversion was scheduled to take place, and using the cross-reference table in the conversion program.

## Conversion Code

For a variety of reasons (prominent among them, the need to use standardized procedures to load data from different hardware/software sources and the inavailability of alternate software), we decided not to load data directly from its native format into Oracle. Instead, all legacy data would be exported to ASCII format, transferred to the target machine, and loaded via SQL\*Loader.

Further, we determined that most loads would have to be done in two steps:

- ?? First, each legacy table would be loaded into a load table whose format matched that of the legacy table. This step would use a SQL\*Loader **control file** and associated script.
- ?? Then, the load table contents would be moved into the application tables by a **conversion program**, which we usually called **conversion scripts**.

The load process was divided into two parts, because the degree of data cleansing and integrity checking we needed to do was more than SQL\*Loader could handle. For example, data in multiple rows of a table had to be concatenated to form one field; or a look-up to determine the new unique key constraint for a record had to be performed. So, the loads could be relatively simple. The actual conversion scripts would be more difficult.

### Loader Control Files

First cuts at most loader control files can automatically be produced by a code generator similar to the one that produces check scripts. The generator produces a control file based on the legacy table definition stored in the repository. The control file can then be used as a parameter to the shell script which invokes SQL\*Loader.

In practice, we sometimes were able to produce a control file that needed no post-generation tweaking. A generated file for the legacy table *tranship* would look like:

```
LOAD DATA
INFILE '/dbclean/data/tranship.dat'
REPLACE
INTO TABLE tranship
(
  t_shnum POSITION(1:7) INTEGER
  EXTERNAL
      NULLIF t_shnum = BLANKS,
  t_gbl POSITION(8:19) CHAR
      NULLIF t_gbl = BLANKS,
  t_doc POSITION(20:34) CHAR
      NULLIF t_doc = BLANKS,
  ...)
```

More frequently, some customization was required, either to the control file or to the definition of the table into which data was to be loaded, or to both. Hand-coding was required for:

- ?? Redefining one column into two or more values
- ?? Handling dates in different formats, for example, ‘YYYYMMDD’, ‘YDDD’, and more
- ?? Processing COBOL-derived SCALED numbers, with implicit decimal points.

It was standard procedure to create extra columns in addition to the original legacy column which consisted of a concatenated string:

```
(
  t_stkno POSITION(1:15) CHAR
      NULLIF t_stkno = BLANKS,
  ...
  fsc_cd POSITION(1:4) CHAR
      NULLIF fsc_cd = BLANKS,
  niin POSITION(5:13) CHAR
      NULLIF niin = BLANKS)
```

In such cases, the new column names complied with the target application’s naming standards, which made their special nature apparent. This “duplication” eliminated having to code substring (SUBSTR) functions, which are both tedious to write and inefficient to execute, in the conversion scripts

And sometimes, both the control file and table definition had to be virtually written by hand. This was done when the legacy table used extensive record typing. For example, one legacy table had forty different record formats, based on the value of the *dic* (*document identifier code*) column.

### Conversion Scripts

Our conversion standards specified that the basic format for a conversion script would be a SQL INSERT/SELECT command. Timing tests indicated that SQL commands were an order of magnitude faster than PL/SQL commands and cursors in performing large batch inserts.

Yet another code generator produced the first cut at the conversion scripts, based on the legacy-to-target table and column mappings. The generator used two input parameters: the legacy table name and the target table name.

The heart of the code generator looks like:

```
SPOOL '&&legacy_tab_nm' '&&target_tab_nm'.sql

SELECT 'INSERT INTO ' ||
       LOWER('&&target_tab_nm') ||
       ' ('
FROM dual
/
SELECT lpad(' ', 6, ' ') ||
       DECODE(ROWNUM, 1,
             rpad(' ', 2, ' '),
             rpad(' ', 2, ' '))
||
       LOWER(target_col_nm)
FROM col_xref_vw
WHERE legacy_tab_nm =
      UPPER('&&legacy_tab_nm')
      AND target_tab_nm =
      UPPER('&&target_tab_nm')
/
SELECT ' ) (SELECT '
FROM dual
/
SELECT lpad(' ', 4, ' ') ||
       DECODE(ROWNUM, 1, NULL,
             rpad(' ', 2, ' ')) ||
       DECODE(conv_code_txt, NULL,
             DECODE(col_sequence_ref,
                   LOWER(legacy_col_nm),
                   target_col_nm ||
                   '_seq.NEXTVAL'),
             conv_code_txt) || ' '
NULL,
       FROM col_xref_vw
WHERE legacy_tab_nm =
      UPPER('&&legacy_tab_nm')
      AND target_tab_nm =
      UPPER('&&target_tab_nm')
/
SELECT ' FROM ' ||
       LOWER('&&legacy_tab_nm') ||
       ';'
FROM dual
/
SPOOL OFF
```

This produces a script which INSERTs into the target table all the SELECTed rows from the legacy table:

```
INSERT INTO carrier (
  carrier_id
  , carrier_nm
  , carrier_stat_cd
  , std_carrier_alpha_cd
  , icc_license_nbr
) (SELECT
  carrier_id_seq.NEXTVAL
  , c_name
  , DECODE(c_status, 'X', 'I',
          c_status)
  , c_scac
  , c_icc
FROM transcarrier
```

This is a good starting point. Unfortunately, it is as far as we got with automating conversion code generation. In addition to column-to-column mappings, the generator knew how to do the following:

- ?? If some special conversion code (conv\_code\_txt) were defined in the column mapping, it was used in the SELECT list instead of the legacy column name. This allowed us to DECODE column values.
- ?? If the target column had a sequence assigned (col\_sequence\_ref was NOT NULL), the sequence.NEXTVAL was inserted.

We had a fairly hefty set of programming standards to guide the programmers in coding what the generator didn't know how to do. The most obvious omission, added to every script, was to not try to insert a row with a primary or unique key that already existed:

```
INSERT INTO carrier (
  ...
) (SELECT
  ...
FROM transcarrier t
WHERE NOT EXISTS
  (SELECT 1 FROM carrier c
   WHERE c.std_carrier_alpha_cd =
         t.c_scac)
```

or, in general terms:

```
) (SELECT
  ...
FROM legacy_tab_nm l
WHERE NOT EXISTS
  (SELECT 1 FROM target_tab_nm t
   WHERE t.unique_key =
         l.unique_key)
```

Other coding techniques, based on the cleansing and conversion strategies adopted, were also used extensively.

## Conclusion

On the one hand, there is no conclusion ... yet. Most of the analysis and cleansing work has been done. Most of the conversion code is written and tested, but on the test machine, using small samples of the legacy data. Within the month, we will transfer the code to the production machine and start testing the code using full-data extracts. The actual cutover and system deployment is scheduled for December. Wish us luck.

On the other hand, I think we've answered the question, "Who puts the data in the database?" - the data conversion team (with help from the users and development team). God bless 'em. And grant 'em the respect they deserve.