Cross-loading of Legacy Data Using the Designer/2000 Repository Data Model

Jeffrey M. Stander
ANZUS Technology International
Presented at ODTUG 1996 Meeting

OBJECTIVES

To design and implement a methodology for migrating legacy data into the a new database using information extracted from the Designer/2000 Repository Data Model to validate data integrity.

ABSTRACT

When migrating legacy data from an old system into a new Oracle database, difficulties often arise when the old data violates referential or table integrity on the new database.

One way to deal with this problem is to define each integrity constraint with an `EXCEPTIONS INTO` clause which causes Oracle to write information to a special table for each row that violates integrity.. Several problems exist with this method, the most critical being the inability to validate constraints defined for client-enforcement and secondary failures due to invalidation of a row by another constraint.

A way around these drawbacks is to use the methodology of *cross-loading*. This technique consists of loading the raw data into unconstrained interim tables that mirror the constrained final tables, validating the data in the interim tables, and finally loading all data that passes the integrity checks into final tables. The SQL scripts that are used to validate referential and client-side constraints are automatically generated from the Repository Data Model.

Cross-loading of Legacy Data Using the Designer/2000 Repository Data Model

Jeffrey M. Stander
WillStand Consultants P/L

INTRODUCTION

When migrating legacy data from an old system into a new Oracle database, difficulties often arise when the old data violates referential or table integrity on the new database.

EXCEPTIONS TABLE METHOD

One way to deal with this problem is to define each integrity constraint with an `EXCEPTIONS INTO` clause which causes Oracle to write information to a special table for each row that violates integrity. Data is loaded directly into tables (probably using SQL*Loader) with the integrity constraints disabled. When the constraints are enabled the exceptions table is written with information which can be used to delete or extract the offending data, after which the constraint enabling should proceed without difficulty. Several problems exist with this method:

1. So-called *client-side* constraints (those not enforced by the database server) are not included in the validation
2. Each constraint must be modified with the `EXCEPTIONS INTO` clause
3. The only information available on the constraint failure is the table name, constraint name and rowid of the offending row.
4. There is no way with this method to invalidate the reference for optional constraints while leaving the rest of the data in the row intact.

CROSS-LOADING METHOD

A way around these drawbacks is to use the methodology of *cross-loading*. This technique consists of loading the raw data into unconstrained interim tables that mirror the constrained final tables, validating the data in the interim tables, storing failure data in a special violations table, and finally loading all data that passes the integrity checks into final tables. The SQL scripts that are used to validate referential and client-side constraints are automatically generated from the Repository Data Model.. The advantages of using this method are:

- validation scripts can be generated automatically by direct access to the Repository
- constraint `EXCEPTION` clauses are not necessary
- client-side constraints can be enforced
- rowid references to offending data are not required
- invalid optional foreign keys can be nullified and the child row kept
- recursive validation can be done to eliminate secondary referential integrity violations
- more details on the constraint failures can be kept for post-loading analysis and data cleanup

Some disadvantages are that additional space is required for the interim tables and more processing time is necessary.

GENERAL METHODOLOGY OF CROSS-LOADING

In order to validate data integrity, special cross-loading validation scripts are run against the interim tables prior to final loading. The Validation Code Generator used to create these scripts is made up of a combination of SQL plus in-line functions and special views on the Repository Data Model. (See Appendix B.). It was not necessary to use any PL/SQL except for inline functions, which makes for fast execution.

The general method follows these steps:

1. Create interim load tables in a separate schema which are identical to the final tables.
2. Create additional columns in the interim tables to flag any violations.
3. Load data into interim tables.
4. Run the primary constraint validation scripts (Note: the order in which the scripts are run is important and can save time analysing secondary failures). (*Note: As part of this process, custom SQL scripts can be run to validate special conditions such as mandatory subtype constraints or to perform any required custom updates*.)
5. After all primary failures are marked then a recursive validation must take place to eliminate secondary failures.
6. Disable or drop all foreign key constraints on the final tables.
7. Delete or truncate all data in the final tables.
8. Load the final tables by selecting all rows from the interim tables (excluding the additional columns added in step 2) which do not have a primary or secondary mandatory failure. All rows with primary or secondary optional failures will have the offending foreign key nullified at this time.

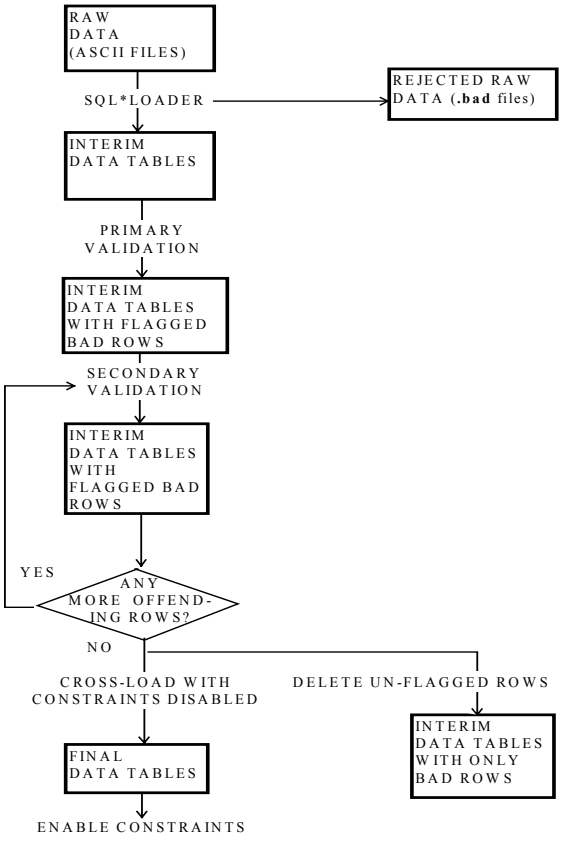

Figure 1. Validation and Cross-Loading

9. Enable or create the foreign key constraints on the final tables.
10. Delete all rows from interim tables which have no failures. This leaves all rows behind which were either rejected (mandatory failures) or modified (optional failures). This offending data must be dealt with separately.

Note that steps 1 to 4 are all that is necessary to support a data cleanup effort. These steps can be run a number of times to identify offending raw data which are candidates for additional cleanup.

INTERIM TABLES

Interim tables are created in another schema and are initially identical to the final tables to which the data will be loaded. Each table must then have additional rows added to it which are used to flag a violation and to contain a key linking the row to the violation data stored in the special violations table. No referential or check constraints are defined on these tables, but primary and unique key constraints must be enabled and indexes should be generated.

SQL*LOADER

If the Conventional Path is used for SQL*Loader then all server-side primary key, unique key and check constraints (i.e. non-referential table constraints) will reject any failures of these constraints.

If Direct Path loading is used then primary and unique keys must be disabled or dropped. This is because the Direct Path method loads all data, tries to rebuild the unique indexes, and then leaves the indexes in the "DIRECT LOAD" state if the data violates uniqueness. Since Direct Path does not test any check constraints these have to be handled by validation scripts in order to separate optional from mandatory primary failures.

The easiest thing to do is to use Conventional Path loading if at all possible, which means that only client-side constraints and all foreign key constraints must be validated after loading.

PRIMARY FAILURES

The validation scripts will cause an initial modification or rejection of bad data at load time. All failures at this stage are called *primary failures* as they are due to missing or invalid parent keys or failure to satisfy a check constraint.

A *primary mandatory failure* rejects the entire row when the parent key is missing for a mandatory foreign key constraint.. A *primary optional failure* rejects only the foreign key by nullifying an optional foreign key when the parent key is not found.

Optionality as it applies to cross-loading determines whether the entire row is rejected if a foreign key fails to find a parent reference. Clearly all check constraints are mandatory. A foreign key is mandatory if it contains NOT NULL key components or is flagged as mandatory in the repository. In the cross-loading exercise, a foreign key constraint can also be promoted to mandatory status if one of it's key components is a member of a unique or primary key. The Validation Code Generator uses the function *xlutl.unique_constraint_in* (see Appendix B) to determine if an optional constraint is promoted.

SECONDARY FAILURES

After all primary failures are detected and flagged in the interim tables a recursive validation must take place to eliminate *secondary failures*. These occur when a valid parent key exists but the parent row itself has incurred a primary failure. The validation must be recursive because the row being invalidated for a secondary failure may contain one or more parent keys referenced by other tables. A *secondary mandatory failure* rejects the entire row. A *secondary optional failure* rejects the foreign key only.

The interim tables are modified to contain additional columns (see below). When a primary or secondary failure is detected in a row, that row is tagged using these columns. One column is a permanent flag which is set when a primary or secondary mandatory failure is detected. Another column is transient, being cleared and reset for each constraint being validated. A third column contains a unique number for each row that fails any constraint. This is the *validation number* and becomes the link for that row to a special table with the name `XL_VIOLATIONS`. There is a many-to-one relationship between this table and the interim tables. XL_VIOLATIONS stores the table name, constraint name, violation number, type of constraint, type of violation, rowid of the offending row, run number, and the date and time of the failure. (See Appendix A for a table description and definition of constraint and violation types)

After validation is completed, data loading to the final tables can then proceed with all data conforming to integrity constraints. The offending data left behind will have the failure reason at hand to allow for cleaning, reloading or rejection.

SELECTING CONSTRAINTS FOR VALIDATION — EXTENDING THE REPOSITORY

Normally the Validation Code Generator is set to select all constraints for the selected tables which have their CREATE STATUS property set in the RON (Repository Object Navigator). The user extensibility option allows creation of a new constraint property: `VALIDATE IN CROSS-LOAD`. This can be used to signal the Validation Code Generator directly that a constraint must be included in the cross-load validation. I am not clear if Designer/2000 upgrades will carry the user extensibility forward.

A less elegant but simpler method is to keep a list of all tables with the constraints to be checked in a special table created for that purpose. I experimented with the former method but chose the latter because of reasons having to do with repository access. A custom package of inline functions was used (Appendix B) to access the table with the list of constraints for validation.

SUBTYPE FOREIGN KEY CONSTRAINT AND VALIDATION

On our particular project subtypes were defined as updatable views on a supertype base table. These become candidates for validation when they have foreign key constraints which are mandatory on the view but optional on the base table. Naturally, these are marked as client-side enforced constraints.

Since the Repository Data Model does not handle subtypes well, particularly when instantiated as views, this situation requires special handling. Automatic generation of validation scripts is difficult for this situation. Three solutions present themselves:

1. Modify a copy of the subtype view to allow updating of the base table special validation columns and allow the Validation Code Generator to create code from the foreign key constraints defined in the view..
2. Modify by hand the SQL generated by the Validation Code Generator for the base table to include the subtype discriminator and to then check for a mandatory foreign key constraint. Two scripts need to be created: one to detect primary failures and another to detect secondary failures.
3. Modify the repository (using user extensibility or perhaps special text in the COMMENTS field) and extend the Validation Code Generator to recognise subtypes and create the appropriate code.

The third option is best but difficult to write. The first option is the most practical and is what I chose to use.

SPECIFIC METHODOLOGY

CUSTOM REPOSITORY VIEWS AND THE VALIDATION CODE GENERATOR

In order to extract information from the Repository Data Model a number of custom views have been written. These are described in more detail in the author's paper *Accessing the Designer/2000 Repository Tables.* Each view returns information from the Repository Data Model only for the application system which we are validating..

TAB_V	— Return id number, table names and aliases (short names)
PK_V	— Get primary key name and columns for a table
UK_V	— Get unique key name and columns for a table
FK_ALL_V	— Return information on foreign keys
FK_COLS_V	— Return information on foreign keys and key components
CC_ALL_V	— Return text for check constraints on tables

The Validation Code Generator depends on these views to create SQL statements for primary and secondary validation. It is too lengthy to append to this paper and is available from the author. In brief it functions as follows, writing its output to a file which serves as the master controller for primary and secondary validation.

1. Read a validation parameter file to determine interim and final schema names and other parameters. (Example in Appendix D)
2. Find the check constraints and foreign key constraints which are designated for validation.
3. Writing to a work table, denormalize the key component references onto a single row.
4. Truncate the `XL_VIOLATIONS` table.
5. Using a view (Appendix E) which reads from the work table, write the SQL statements for primary validation.
6. Write calls to programs that will dynamically generate the SQL for secondary validation. This will be similar to step 5.
7. Select from DML generating views to create the scripts to disable and enable constraints in the final table schema and, most importantly, to transfer the data from the interim schema to the final schema (see Appendix F)

INTERIM TABLES

The first step is to modify the interim tables to add three new columns.

XL_VIOLATION	— Temporary flag marks a row modified or rejected on current pass
XL_VIOLATION_FLAG	— Permanent flag indicates row is not to be loaded to final tables
XL_VIOLATION_NUMBER	— Permanent sequence number assigned to a row on failure

This is done for all interim tables by the xload_mod.sql script listed in Appendix C. Each time a complete validation is run, all of these fields must be set to NULL.

PRIMARY FAILURES

Three script components are generated for each check constraint or foreign key constraint.

1. UPDATE the target table, set XL_VIOLATION='T", XL_VIOLATION_NUMBER to the next sequence number (if not already set), and XL_VIOLATION_FLAG='T' only if this is a check constraint or mandatory foreign key constraint (i.e. not an optional foreign key constraint)..
2. For each occurrence of a non-null value of XL_VIOLATION, INSERT into the XL_VIOLATIONS table the table name, constraint name, the violation number, the type of constraint, the type of violation, the rowid of the offending row, the run number, and the date and time. Primary failures are marked '**PO**' or '**PM**' (primary optional or primary mandatory) in the XL_VIOLATIONS table.
3. UPDATE the target table and nullify any values of XL_VIOLATION. (NOTE: This is a transient flag. It is used over again for every foreign key constraint or check constraint on a given table. XL_VIOLATION_FLAG is permanent for an entire validation run, as is XL_VIOLATION_NUMBER which allows a many-to-one reference from the XL_VIOLATIONS table to the offending row).

For example, to validate the optional foreign key constraint PROD_EMP_FK, which links the PRODUCT table to the EMPLOYEE table, the Validation Code Generator produced the code listed in Figure 2 below.

SECONDARY FAILURES

Secondary failures occur when a valid parent key belongs to a row which is itself rejected for a primary or secondary failure. Secondary failures can only be checked for after all primary failures have been registered in the interim tables (or at least all tables with referential dependencies). Because secondary failures can themselves result in additional secondary failures, secondary failure testing must be run recursively. To minimise the recursion:

1. Restrict testing only to foreign key constraints (check constraint do not apply)
2. Do all self-referencing foreign key constraints on a given table in one pass. Use a tree walk if possible. (In practice this may not be necessary. It wasn't on our project).
3. Restrict validation only to tables dependent on tables with invalidations that have occurred since the time of the last validation. This can be done by re-running the Validation Code Generator only for those tables that meet the following condition or by including the condition in the generated secondary validation script. The *run_number* field in the XL_VIOLATIONS table starts at 1 for every complete validation. It is incremented by one after each secondary validation. Thus, to find which tables need secondary validation:

```
SELECT distinct table_name
FROM   XL_VIOLATIONS
WHERE  run_number = MAX(run_number);
```

Recursive validation terminates when no more violations are found, i.e. when

```
SELECT count(*)
FROM   XL_VIOLATIONS;
```

returns the same number before and after the secondary failure validation. A Unix shell script is used to re-run the validation until the above condition is met.


```
PROMPT ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
PROMPT Checking PRODUCT for primary failures

PROMPT Validating foreign key PROD_EMP_FK on table PRODUCT
PROMPT Implementation Level = CLIENT
PROMPT Mandatory Flag       = N

set timing on
SET TRANSACTION USE ROLLBACK SEGMENT RMASSIVE;

UPDATE xload.PRODUCT a
SET    XL_VIOLATION = 'T',
       XL_FLAG = 'T',
       XL_VIOLATION_NUMBER =
            DECODE(XL_VIOLATION_NUMBER,NULL,XL_VIOLATION_SEQ.NEXTVAL,XL_VIOLATION_NUMBER)
WHERE  a.EMP_PARTY_ID IS NOT NULL  -- If this constraint were MANDATORY, this line would be missing
AND NOT EXISTS
      (SELECT 1
       FROM   xload.EMPLOYEE b
       WHERE  a.EMP_PARTY_ID = b.PARTY_ID
      )
/

set feedback off
set timing off
COMMIT;
set feedback on

PROMPT Inserting failure data into XL_VIOLATIONS table

set timing on
INSERT INTO XL_VIOLATIONS
SELECT 'PRODUCT',
       'EMP_PARTY_ID',
       'PROD_EMP_FK',
       'R',
       1,
       'PO',
       XL_VIOLATION_NUMBER,
       1,
       ROWID,
       'CLIENT',
       SYSDATE
FROM   xload.PRODUCT
       WHERE XL_VIOLATION IS NOT NULL
/

set timing off
set feedback off
COMMIT;
set timing on

UPDATE xload.PRODUCT
SET    XL_VIOLATION = null
WHERE  XL_VIOLATION is not null
/

set timing off
COMMIT
```

Figure 2. Example of Primary Optional Validation SQL statements. Note the only difference between a this and a Mandatory Validation is the `WHERE..IS NOT NULL` statement would be missing and the 'PO' value would be 'PM'.

Three scripts are generated for each foreign key constraint:. Self-referencing foreign key constraints will have a different version of script 2.

1. For each row, check that there does not exist a parent key reference which itself has a violation recorded against it (**`b.XL_VIOLATION_NUMBER IS NOT NULL`**). If there is a violation, determine if the parent row will be deleted. In Figure 3 below, note the use of the inline function *`xl_chop`* to validate this.
2. Next, determine if this violation is already registered in the table. If not, go ahead and `UPDATE` the target table as with a primary constraint. — set `XL_VIOLATION_NUMBER` to the next sequence number (if not already set), and set `XL_VIOLATION_FLAG='T'` only if this is a mandatory foreign key constraint (i.e. not an optional foreign key constraint). See Figure 3 below.
3. `INSERT` into the `XL_VIOLATIONS` in the same way as for a primary constraint. Secondary failures are marked '**SO**' or '**SM**' (secondary optional or secondary mandatory).
4. `UPDATE` the target table and nullify any values of `XL_VIOLATION`.

```
UPDATE xload.PRODUCT a
SET    XL_VIOLATION = 'T',
       XL_FLAG = 'T',
       XL_VIOLATION_NUMBER =
            DECODE(XL_VIOLATION_NUMBER,NULL,XL_VIOLATION_SEQ.NEXTVAL,XL_VIOLATION_NUMBER)
WHERE a.EMP_PARTY_ID IS NOT NULL -- If this constraint were MANDATORY, this line would be missing
AND EXISTS
      (SELECT 1
       FROM   xload.EMPLOYEE b
       WHERE  a.EMP_PARTY_ID = b.PARTY_ID
       AND    b.XL_VIOLATION_NUMBER IS NOT NULL
       AND    (b.XL_FLAG='T' OR xlutl.xl_chop(b.XL_VIOLATION_NUMBER)=1)
      )
      AND ( a.XL_VIOLATION_NUMBER IS NULL
      OR NOT EXISTS
             (SELECT 1 from XL_VIOLATIONS x
              WHERE  x.VIOLATION_NUMBER = a.XL_VIOLATION_NUMBER
              AND    'EMP_PARTY_FK' = x.CONSTRAINT_NAME
              AND    x.column_seq = 1
             )
      )
/
```

Figure 3. Example of a Secondary Optional Validation SQL statement (partial).

CROSS-LOADING

Cross-loading proceeds by inserting into each table in the target schema with a select from the interim schema. The only conditions are that where an optional secondary failure is found (XL_FLAG is null and XL_VIOLATION_NUMBER is not null), then we need to look up the specific violation for the table, column, and row in question to see if that particular field was invalidated. This is done by an inline function *xl_chop (violation_number,column_name).* If *xl_chop* returns a value of 1, the field is set to null. (See example in Figure 4)


```
TRUNCATE TABLE final.PRODUCT;
SET TRANSACTION USE ROLLBACK SEGMENT RMASSIVE;
INSERT INTO final.FEE_THRESHOLD
(
         id,
        ,party_id
        ,area_id
        ,owner_id
        ,name
)
SELECT
         ID,
        ,decode(xl_violation_number,null,PARTY_ID,
           decode(xload.xlutl.xl_chop(xl_violation_number,'PARTY_ID'),1,null,PARTY_ID)) PARTY_ID
        ,AREA_ID
        ,OWNER_ID
        ,NAME
FROM    xload.PRODUCT
WHERE   xl_flag is null
/

COMMIT;
```

Figure 4. Example of a SQL statement to cross-load a table. If PARTY_ID has a validation failure lodged against it, it is set to null.

To speed up the cross-loading process indexes on the target tables can be dropped and recreated after loading. At that time the constraints can be enabled or created as the case may be and there you have it — a fully loaded database with all referential integrity in place and ready to go.

APPENDICES

The full text of this paper including the appendices containing the SQL code are available from the author:

Jeffrey M. Stander
ANZUS Technology International
www.anzustech.com

