Error Management Features of Oracle PL/SQL

Steven Feuerstein
Oracle Developer Advocate for PL/SQL
Oracle Corporation
steven.feuerstein@oracle.com
@sfonplsql
stevenfeuersteinonplsql.blogspot.com
Practically Perfect PL/SQL (YouTube)

Resources for Oracle Database Developers

- Official home of PL/SQL - oracle.com/plsql
- SQL-PL/SQL discussion forum on OTN https://community.oracle.com/community/database/developer-tools/sql_and_pl_sql
- PL/SQL and EBR blog by Bryn Llewellyn - https://blogs.oracle.com/plsql-and-ebr
- Oracle Learning Library - oracle.com/oll
- Weekly PL/SQL and SQL quizzes, and more - devgym.oracle.com
- Ask Tom - asktom.oracle.com – 'nuff said
- LiveSQL - livesql.oracle.com – script repository and 12/7 12c database
- oracle-developer.net - great content from Adrian Billington
- oracle-base.com - great content from Tim Hall

Error Management in PL/SQL

- Defining exceptions
- Raising exceptions
- Handling exceptions

- Let’s start with quizzes to test your knowledge of the basic mechanics of exception handling in PL/SQL.

All referenced code is available at livesql.oracle.com and in my demo.zip file from the PL/SQL Learning Library: oracle.com/oll/plsql.

Quiz: When strings don't fit...(1)

- What do you see after running this block?

```
DECLARE
   aname VARCHAR2(5);
BEGIN
   BEGIN
      aname := 'Big String';
      DBMS_OUTPUT.PUT_LINE (aname);
   EXCEPTION
      WHEN VALUE_ERROR THEN
         DBMS_OUTPUT.PUT_LINE ('Inner block');
   END;
   DBMS_OUTPUT.PUT_LINE ('What error?');
EXCEPTION
   WHEN VALUE_ERROR THEN
      DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
```

excquiz1.sql
livesql.oracle.com "Test Your PL/SQL Exception Handling Knowledge"

Quiz: When strings don't fit...(2)

- What do you see after running this block?

```
DECLARE
   aname VARCHAR2(5);
BEGIN
   DECLARE
      aname VARCHAR2(5) := 'Big String';
   BEGIN
      DBMS_OUTPUT.PUT_LINE (aname);

   EXCEPTION
      WHEN VALUE_ERROR
      THEN
         DBMS_OUTPUT.PUT_LINE ('Inner block');
   END;
   DBMS_OUTPUT.PUT_LINE ('What error?');
EXCEPTION
   WHEN VALUE_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
/
```

excquiz2.sql

Defining Exceptions

- The EXCEPTION is a limited type of data.
  - Has just two attributes: code and message.
  - You can RAISE and handle an exception, but it cannot be passed as an argument in a program.
- Associate names with error numbers with the EXCEPTION_INIT pragma.

The EXCEPTION_INIT Pragma

```
CREATE OR REPLACE PROCEDURE upd_for_dept (
   dept_in   IN   employee.department_id%TYPE
 , newsal_in IN   employee.salary%TYPE
)
IS
   e_forall_failure   EXCEPTION;
   PRAGMA EXCEPTION_INIT (e_forall_failure, -24381);
```

- Use EXCEPTION_INIT to...
  - Give names to Oracle error codes for which no exception was defined.
  - Assign an error code other than 1 to a user-defined exception (usually in the -20NNN range used by RAISE_APPLICATION_ERROR).

exception_init.sql
livesql.oracle.com search "exception_init"

Raising Exceptions

- RAISE raises the specified exception by name.
  - RAISE; re-raises current exception. Callable only within the exception section.
- RAISE_APPLICATION_ERROR
  - Communicates an application specific error back to a non-PL/SQL host environment.
  - Error numbers restricted to the -20,999 - -20,000 range.

8

The RAISE Statement

- Use RAISE to raise a specific, named exception or to *re-raise* the current exception.
  - Raise pre-defined and user-defined exceptions.
- "RAISE;" re-raises the current exception.
  - You can only use RAISE; from within an exception handler.

```
   e_forall_failure EXCEPTION;
   PRAGMA EXCEPTION_INIT (e_forall_failure, -24381);
BEGIN
   ...
   RAISE e_forall_failure;
EXCEPTION
 WHEN e_forall_failiure THEN
      log_error();
      RAISE;
 WHEN OTHERS THEN
      log_error();
      RAISE;
```

raise.sql
reraise.sql

Communicating Application Errors

- When Oracle raises an exception, it provides the error code and error message.
- When an application-specific error occurs, such as "Account balance too low", it is up to *you*, the developer, to communicate all necessary information to the user.
  - This can involve an error code, error message, or both.
- For this, you must use RAISE_APPLICATION_ERROR.
  - And, really, that is the only time.

Defining application-specific messages

- You can raise an application-specific error with RAISE, but you cannot pass back an application-specific *message* that way.
- With RAISE_APPLICATION_ERROR, you can issue your own "ORA-" error messages.
  - This built-in replaces the values that would normally be returned with a call to SQLCODE and SQLERRM with *your* values.
- This information can be displayed to the user or sent to the error log.

RAISE_APPLICATION_ERROR Example

```
IF :NEW.birthdate > ADD_MONTHS (SYSDATE, -1 * 18 * 12)
THEN
   RAISE_APPLICATION_ERROR
      (-20070, 'Employee must be 18.');
END IF;
```

- This code from a database trigger (indicated by ":NEW") shows a typical (and typically *problematic*) usage of RAISE_APPLICATION_ERROR.
  - It's a business rule requiring a "business" message.
- Typically problematic:
  - Hard-coding of error code and message
  - Duplication of "rule" information: 18 years.

employee_must_be_18.sql
Livesql.oracle.com search "raise_application_error"

12

RAISE_APPLICATION_ERROR Details

```
RAISE_APPLICATION_ERROR (
   num binary_integer
 , msg varchar2
 , keeperrorstack boolean default FALSE);
```

- Defined in the DBMS_STANDARD package.
- The *num* argument range: -20,999 and -20,000.
  - Use something else and *lose* the error message!
- Error messages can be up to 2048 bytes*.
  - You can pass to a string of up to 32K bytes, but you won't be able to get it "out"; your users will not be able to see the full string.
- Third argument determines if the full *stack* of errors is returned or just the most recent.

raise_application.sql
max_rae_string_length.sql
errorstack.sql

Handling Exceptions

- The EXCEPTION section consolidates all error handling logic in a block.
  - But only traps errors raised in the executable section of the block.
- Several useful functions usually come into play:
  - SQLCODE and SQLERRM
  - DBMS_UTILITY.FORMAT_CALL_STACK
  - DBMS_UTILITY.FORMAT_ERROR_STACK
  - DBMS_UTILITY.FORMAT_ERROR_BACKTRACE
  - Plus, new to 12.1, the UTL_CALL_STACK package
- The DBMS_ERRLOG package
  - Quick and easy logging of DML errors

SQLCODE and SQLERRM

- SQLCODE returns the error code of the most recently-raised exception.
  - You cannot call it inside an SQL statement (even inside a PL/SQL block).
- SQLERRM is a generic lookup function: return the message text for an error code.
  - And if you don't provide an error code, SQLERRM returns the message for SQLCODE.
- But....SQLERRM might truncate the message.
  - Very strange, but it is possible.
  - So Oracle recommends that you *not* use this function.

sqlerrm.sql
livesql.oracle.com search "sqlerrm"

DBMS_UTILITY.FORMAT_CALL_STACK

- The "call stack" reveals the *path* taken through your application code to get to that point.
- Very useful whenever tracing or logging errors.
- The string is formatted to show line number and program unit name.
  - But it does not reveal the names of *subprograms* in packages.

callstack.sql
callstack.pkg
livesql.oracle.com search "call_stack"

DBMS_UTILITY.FORMAT_ERROR_STACK

- This built-in returns the error *stack* in the current session.
  - Possibly more than one error in stack.
- Returns NULL when there is no error.
- Returns a string of maximum size 2000 bytes (according to the documentation).
- Oracle recommends you use this instead of SQLERRM, to reduce the chance of truncation.

errorstack.sql
big_error_stack.sql
livesql.oracle.com search "error_stack"

DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

- The backtrace function answers the question: "Where was my error raised?
  - Prior to 10.2, you could not get this information from within PL/SQL.
- Call it whenever you are logging an error.
- When you re-raise your exception (RAISE;) or raise a different exception, subsequent BACKTRACE calls will point to *that* line.
  - So before a re-raise, call BACKTRACE and store that information to avoid losing the original line number.

backtrace.sql
bt.pkg
livesql.oracle.com search "backtrace"

18

Or Use UTL_CALL_STACK (12.1 and higher)

- This new package provides a granular API to the call stack and back trace.
- It also now provides complete name information, down to the nested subprogram.
- But the bottom line is that generally you will call your functions in the exception section (thru a generic error logging procedure, I hope!) and then write that information to your error log.
- When you need to diagnose an error, go to the log, grab the string, and analyze.

12c_utl_call_stack*.sql
liveSQL.oracle.com search "utl_call_stack"

Continuing Past Exceptions

- What if you want to continue processing in your program even if an error has occurred?
- Three options...
  - Use a nested block
  - FORALL with SAVE EXCEPTIONS
  - DBMS_ERRLOG

continue_past_exception.sql

Exception handling and FORALL

- When an exception occurs in a DML statement....
  - That statement is rolled back and the FORALL stops.
  - All (previous) successful statements are *not* rolled back.
- Use the SAVE EXCEPTIONS clause to tell Oracle to continue *past* exceptions, and save the exception information for later.
- Then check the contents of the pseudo-collection of records, SQL%BULK_EXCEPTIONS.
  - Two fields: ERROR_INDEX and ERROR_CODE

FORALL with SAVE EXCEPTIONS

- Suppress errors at the *statement* level.

```
CREATE OR REPLACE PROCEDURE load_books (books_in IN book_obj_list_t)
IS
  bulk_errors EXCEPTION;
  PRAGMA EXCEPTION_INIT ( bulk_errors, -24381 );
BEGIN
  FORALL indx IN books_in.FIRST..books_in.LAST
    SAVE EXCEPTIONS
    INSERT INTO book values (books_in(indx));
EXCEPTION
   WHEN bulk_errors THEN
      FOR indx in 1..SQL%BULK_EXCEPTIONS.COUNT
      LOOP
         log_error (SQL%BULK_EXCEPTIONS(indx).ERROR_CODE);
      END LOOP;
END;
```

Allows processing of all statements, even after an error occurs.

Iterate through pseudo-collection of errors.

bulkexc.sql
livesql.oracle.com search "save_exceptions"

LOG ERRORS and DBMS_ERRLOG

- Use this package (added in Oracle Database 10g Release 2) to enable row-level error logging (and exception suppression) for DML statements.
  - Compare to FORALL SAVE EXCEPTIONS, which suppresses exceptions at the *statement* level.
- Creates a log table to which errors are written.
  - Lets you specify maximum number of "to ignore" errors.
- Better performance than trapping, logging and continuing past exceptions.
  - Exception handling is *slow*.

dbms_errlog.sql
dbms_errlog_helper.pkg
dbms_errlog_vs_save_exceptions.sql
livesql.oracle.com search "log errors"

LOG ERRORS and DBMS_ERRLOG

- Suppress DML row-level errors!
- Impact of errors on DML execution
- Introduction to LOG ERRORS feature
- Creating an error log table
- Adding LOG ERRORS to your DML statement
- "Gotchas" in the LOG ERRORS feature
- The DBMS_ERRLOG helper package

Impact of errors on DML execution

- A single DML *statement* can result in changes to multiple *rows*.
- When an error occurs on a change to a row....
  - All previous changes from that statement are rolled back.
  - No other rows are processed.
  - An error is passed out to the calling block (turns into a PL/SQL exception).
  - No rollback on completed DML in that session.
- *Usually* acceptable, but what if you want to:
  - Avoid losing all prior changes?
  - Avoid the performance penalty of exception management in PL/SQL?

errors_and_dml.sql
livesql.oracle.com search "Exceptions Do Not Rollback"

Row-level Error Suppression in DML with LOG ERRORS

- Once the error propagates out to the PL/SQL layer, it is too late; all changes to rows have been rolled back.
- The only way to preserve changes to rows is to add the LOG ERRORS clause in your DML statement.
  - Errors are suppressed at row level *within* the SQL Layer.
- But you will first need to created an error log table with DBMS_ERRLOG.

Terminology for LOG ERRORS feature

- **DML table**: the table on which DML operations will be performed
- **Error logging table** (aka, error table): the table that will contain history of errors for DML table
- **Reject limit**: the maximum number of errors that are acceptable for a given DML statement
  - "If more than 100 errors occur, something is badly wrong, just stop."

Step 1. Create an error log table

- Call DBMS_ERRLOG.CREATE_ERROR_LOG to create the error logging table for your "DML table."
  - Default name: ERR$_<your_table_name>
- You can specify alternative table name, tablespace, owner.
  - Necessary if DML table name > 25 characters!
- The log table contains five standard error log info columns and then a column for each VARCHAR2-compatible column in the DML table.

dbms_errlog.sql

Step 2: Add LOG ERRORS to your DML

```
UPDATE employees
       SET salary = salary_in
LOG ERRORS REJECT LIMIT UNLIMITED;

UPDATE employees
       SET salary = salary_in
LOG ERRORS REJECT LIMIT 100;
```

- Specify the limit of errors after which you want the DML statement to stop – or UNLIMITED to allow it to run its course.
- Then...make sure to check the error log table after you run your DML statement!
  - Oracle will *not* raise an exception when the DML statement ends – *big difference* from SAVE EXCEPTIONS.

"Gotchas" in the LOG ERRORS feature

- The default error logging table is missing some critical information.
  - When the error occurred, who executed the statement, where it occurred in my code
- Error reporting is often obscure: "Table or view does not exist."
- It’s up to you to grant the necessary privileges on the error log table.
  - If the “DML table” is modified from another schema, that schema must be able to write to the log table as well.
- Use the DBMS_ERRLOG helper package to get around many of these issues.

The DBMS_ERRLOG helper package

- Creates the error log table.
- Adds three columns to keep track of user, timestamp and location in code.
- Compiles a trigger to populate the added columns.
- Creates a package to make it easier to manage the contents of the error log table.

dbms_errlog_helper.sql
dbms_errlog_helper_demo.sql
livesql.oracle.com search "helper package"

LOG ERRORS - Conclusions

- When executing multiple DML statements or affecting multiple rows, decide on your error policy.
  - Stop at first error or continue?
- Then decide on the level of granularity of continuation: statement or row?
  - LOG ERRORS is the only way to perform row-level error suppression.
- Make sure that you check and manage any error logs created by your code.

Some Recommendations for Error Management

- Set standards before you start coding
  - It's not the kind of thing you can easily add in later
- Always call a log procedure in your exception handler.
  - And everyone should use the *same* log procedure!
- Decide where in the stack you handle exceptions.
  - Always at top level block to ensure that users don't see ugly error messags.
  - If you need to log local block state, handle in that block and re-raise.
- Just use logger. https://github.com/OraOpenSource/Logger

Always call a log procedure in your exception handler

- Always log errors to a table.
- Never insert into your log table in the handler.
- Everyone uses the same procedure.
- Avoid multiple loggings for the same error.

Decide where in the stack you handle exceptions.

- Some suggest that only the top-level block should trap an exception.
  - The error utility functions "remember" where it came from.
- But you lose information about the application state at the moment the exception was raised.
- What I do:
  - Always handle at top level block to ensure that users don't see ugly error messags.
  - If Ineed to log local block state (parameters, variables, etc.), I handle in that block and re-raise.

Error Management Summary

- Exceptions raised in the declaration section always escape unhandled.
  - Consider assigning default values in the executable section instead.
- Call DBMS_UTILITY or UTL_CALL_STACK functions whenever you are logging errors and tracing execution.
- Suppress errors at statement level with FORALL-SAVE EXCEPTIONS
- Suppress errors at row level with LOG_ERRORS
  - But don't use the ERR$ table "as is".
- Rely on a standard, reusable error logging utility.


ORACLE®