Edition-Based Redefinition: Testing App Upgrades Without Being Live

Melanie Caffrey
Oracle America, Inc.
Round Hill, VA USA

Keywords:

11gR2 Edition Redefinition Patch Upgrade

Introduction

The problem of how to successfully upgrade a PL/SQL application without incurring much downtime, but allowing yourself adequate post-upgrade testing has been an on-going problem since PL/SQL was first introduced. 11gR2 changed that with the introduction of Edition-Based Redefinition. Using Edition-Based Redefinition, it is now possible to have multiple versions of your code in place on production: one that is visible and usable by the every-day users of the application, and one that is visible only to developers, testers, and other privileged users.

As most developers and DBAs tasked with rolling out PL/SQL application upgrades can attest to, almost all PL/SQL application upgrades, no matter how small, require some amount of downtime in order to avoid the otherwise inevitable locking and blocking that occurs. It is often not possible to obtain long or frequent downtime windows, the testing window during downtime may not be long enough to be deemed adequate for the testers, and last, but certainly not least, an upgraded PL/SQL application can be difficult to back out of, if necessary. Oracle's 11gR2 version of the database has changed this situation by providing developers and DBAs with a high-availability tool, Edition-Based Redefinition, that gives those responsible for rolling out PL/SQL application upgrades the ability to have more than one version (or, *edition*) of a PL/SQL application running in a database instance and schema at the same time.

EBR: Edition-Based Redefinition

Edition-Based Redefinition, or simply *EBR*, allows you to have more than one occurrence of an *editionable* object. PL/SQL objects are *editionable*, for example. This is so due to a change in the Oracle namespace resolution scheme as of Oracle 11gR2. PL/SQL objects are no longer required to adhere to the <schema>.<object> resolution scheme that was always the norm prior to Oracle 11gR2. For PL/SQL objects, the resolution scheme has been expanded to <schema>.<*edition*>.<object>. The edition is implied, never hard-coded by the developer.

As you can see, it is now possible, for example, for the SCOTT user to have more than one occurrence of, say, a procedure called UPDATE_SALARY. This user could deploy the edition of UPDATE SALARY that is used by the general user population SCOTT.*EDITION_1*.UPDATE_SALARY. However, if the developers discover a bug in UPDATE_SALARY and would like to perform an online application upgrade to fix that bug, and have ample time to test it with a few privileged users, then if they have created a second edition of their application, they can perform this upgrade into, say, EDITION_2, and have it running and testable, all while their regular application users continue to work with EDITION_1. Only when they are ready to have their users work with EDITION_2 do the developers/DBAs need to actually switch the users to use EDITION_2. This type of flexibility in online PL/SQL application upgrade capability allows developers and DBAs the heretofore unheard of

ability to provide more high availability for their PL/SQL applications and increased testing windows (if desired) for their QA personnel.

The Pieces and Parts of EBR

1) The Edition

As of 11.2, each database (whether you choose to use EBR or not) comes with a new object type: an *edition*. Each edition can have its own private occurrence of the same object (see the example earlier with SCOTT's EDITION_1 occurrence of the UPDATE_SALARY procedure and his EDITION_2 occurrence of the same procedure). By default, every 11.2 (and later) database has at least one edition: ORA$BASE. You can verify this default edition by running the following query as SYSDBA.

```
CONN / AS SYSDBA

SELECT property_value
  FROM database_properties
 WHERE property_name = 'DEFAULT_EDITION';

PROPERTY_VALUE
------------------------------------------------------------
ORA$BASE
```

And you can always verify which edition you are currently using or looking at by running the following SQL to check your session's current edition.

```
SQL> SELECT SYS_CONTEXT('USERENV', 'SESSION_EDITION_NAME')
   2   AS edition FROM dual;

 EDITION
 ------------------------------------------------------------
 ORA$BASE
```

When you choose to start using EBR, the first edition you create will be a child of ORA$BASE. Code changes (like an update to UPDATE_SALARY) can be installed in this new edition without being seen by the old (original, say, ORA$BASE) edition. The most common PL/SQL application changes are for PL/SQL bug fixes or functionality enhancements that involve changes to only stored PL/SQL objects like packages, procedures, functions and triggers, to name a few. For example, the following function takes in an employee ID and percentage value and returns the new calculated salary value for an employee.

```
SQL> CREATE OR REPLACE FUNCTION sal_increase
  2                              (p_increase IN  VARCHAR2,
  3                               p_employee IN NUMBER)
  4          RETURN NUMBER
  5         IS
  6          v_new_salary NUMBER := 0;
  7         BEGIN
  8               SELECT (salary * p_increase) + salary
  9                 INTO v_new_salary FROM employee
 10              WHERE employee_id = p_employee;
 11        RETURN v_new_salary;
```

```
 12        END;
```

And to find out with which edition(s) this function is associated, you can always look to see in which edition(s) your PL/SQL objects have been installed by querying the USER_OBJECTS data dictionary view which has a new column called EDITION_NAME as of 11gR2, as the following query demonstrates:

```
SQL> select object_name, object_type, status, edition_name
 2     from user_objects;
OBJECT_NAME    OBJECT_TYPE    STATUS    EDITION_NAME
-----------    -----------    -------   --------------------
EMPLOYEE       TABLE          VALID
SAL_INCREASE   FUNCTION       VALID     ORA$BASE
```

As you can see, the SAL_INCREASE function is currently associated with only one edition, the default ORA$BASE edition. Notice also, that there is no edition name value for the EMPLOYEE table. This is because only PL/SQL objects like packages, procedures, functions, triggers and views (though not materialized views), private synonyms, and all of these objects' related metadata such as GRANT privileges, are editionable. Tables are not editionable. (Though we will see shortly how tables can be made to *seem* editionable.)

Now let's say that someone has requested an enhancement to the SAL_INCREASE function, and that it would be more correct to also pass in a date value that checks to see when the employee was hired. This function should now only return an increased salary value if the passed-in employee was hired before a specified time period. An additional set of mandates are that this code change be made with minimal downtime, be given sufficient testing by the QA team, and be implemented in such a way that its changes can be easily reversed. This situation is perfect for using EBR. But first, you'll need to ready your application to use EBR.

Edition Setup

First, you'll need to create a new edition. You need the CREATE ANY EDITION or DROP ANY EDITION system privileges to perform either of those actions. Once you have the CREATE ANY EDITION system privilege you may create a new edition using the CREATE EDITION command as follows:

```
SQL> create edition app_edition_2
   2  as child of ora$base;
 Edition created.

SQL> select * from dba_editions;

EDITION_NAME            PARENT_EDITION_NAME      USA
----------------------  ------------------------ ---
ORA$BASE                                         YES
APP_EDITION_2           ORA$BASE                 YES
```

The query above illustrates one of the new data dictionary views available in 11gR2, DBA_EDITIONS. This data dictionary view lists all of the editions available for your database, alongside each edition's parent (in the result set above, since ORA$BASE is the default edition that is installed out of the box with any installation of 11gR2, it has no parent), and lets you know whether the listed edition is USABLE (since you can set an edition to be unusable). After you've created a new

edition, you must enable your application user to be able to use the new edition (execute code) and/or alter PL/SQL units within the new edition (run CREATE OR REPLACE statements). In order to allow your application user to alter editionable objects within more than the default ORA$BASE edition, it must be altered to be *editions-enabled*. The following query demonstrates this new syntax:

```
SQL> alter user app_user
   2 enable editions;
  User altered.
```

Now the application user, APP_USER, can run CREATE OR REPLACE and GRANT statements in multiple editions. However, it still needs to be granted explicit access to the newly-created edition, APP_EDITION_2, created earlier. This access is achieved with the following SQL:

```
SQL> grant use
   2  on edition app_edition_2
   3  to app_user;
  Grant succeeded.
```

Now the application user will be able to switch to the new version and execute code within it by running SQL such as the following:

```
CONN app_user/pw

SQL> alter session
  2  set edition = app_edition_2;
Session altered.

SQL> SELECT SYS_CONTEXT('USERENV', 'SESSION_EDITION_NAME')
 2   AS edition FROM dual;

EDITION
------------------------------------------------------------
APP_EDITION_2
```

And if the user executes the same query from USER_OBJECTS demonstrated earlier, he will see the following results:

```
SQL> select object_name, object_type, status, edition_name
 2     from user_objects;
OBJECT_NAME    OBJECT_TYPE    STATUS    EDITION_NAME
-----------    -----------    -------   --------------------
EMPLOYEE       TABLE          VALID
SAL_INCREASE   FUNCTION       VALID     ORA$BASE
```

The reason for this is that when an edition is created as a child of another edition it *inherits* its parent editions' editionable objects. These inherited objects look no different between the parent and child editions when a child edition is first created. No change to the data dictionary for the inherited objects is necessary because the inherited objects are currently merely pointers to the parent edition's objects. It is only once the application user makes a change to one of the editionable objects in the new edition that the query above will yield different results. Now let's consider the following patch enhancement to the SAL_INCREASE function:

```
SQL> CREATE OR REPLACE FUNCTION sal_increase
  2                           (p_increase IN  VARCHAR2,
  3                            p_employee IN NUMBER, p_hire IN DATE)
  4          RETURN NUMBER
  5         IS
  6          v_new_salary NUMBER := 0;
  7        BEGIN
  8       SELECT (salary * p_increase) + salary
  9         INTO v_new_salary FROM employee
 10        WHERE employee_id = p_employee AND hire_date <= p_hire;
 11        RETURN v_new_salary;
 12        END;
```

With this new version of the SAL_INCREASE procedure installed into the APP_EDITION_2 edition, we can execute the following query to see just how many editions accessible to our application user have unique versions of this function:

```
SQL> select object_name, object_type, status, edition_name
 2     from user_objects_ae;
OBJECT_NAME    OBJECT_TYPE    STATUS    EDITION_NAME
-----------    -----------    -------   --------------------
EMPLOYEE       TABLE          VALID
SAL_INCREASE   FUNCTION       VALID     ORA$BASE
SAL_INCREASE   FUNCTION       VALID     APP_EDITION_2
```

A query against the USER_OBJECTS_AE data dictionary view (new as of 11gR2) demonstrates how, once our application user makes a code change to the SAL_INCREASE function in the APP_EDITION_2 edition, that change *actualizes* the function within the edition and it is no longer pointing to the version of the function that exists in the parent edition. This is why you can now see two rows listed for the SAL_INCREASE function within the application user's list of objects.

2) Editioning View

In many PL/SQL application upgrade patch scenarios you will most likely be making changes to PL/SQL units similar to the change shown with the SAL_INCREASE function. However, if you need to make only column additions or change the structure of transaction tables, you will need to create an *editioning view* for each table you wish to modify. An editioning view exposes a different projection of a table into each edition to allow each edition to see only its own columns. Since a table is not editionable, it cannot have the same name as any other object in your edition. Also, since you want your application to reference a table name in the same way it always has, and you don't want to incur downtime to make structural changes to this table (and run into locking/blocking issues or having-to-recompile code issues), it becomes important to devise a way to accomplish both tasks.
The editioning view allows you to not have to change your application's references to a particular table while, at the same time, make structural changes behind the scenes. First things first, to enable your application's tables to be able to use editioning views and, therefore, use EBR functionality you'll need to rename each table. For this particular act of renaming your tables, you will require an outage (hopefully a one-time outage). You can rename each table to differentiate it slightly from the editioning view you will create for it:

```
alter table rpm rename to rpm_t;
```

Then create an editioning view for each of your renamed tables with the original table name:

```
create editioning view rpm as select * from rpm_t;
```

If your intent is to change the structure of a column, then you do not change it (as this would invalidate any dependent code and would defeat the purpose of using EBR), but instead, you add a replacement column. For example, I work with the Unbreakable Linux Network product for Oracle Corporation. This means I work with RPMs, packages that are used to update and enhance a user's Red Hat or Oracle Linux installation. These RPMs often have a structure that looks like this:

Name	Epoch	Version	Release
kernel	(null)	2.6.32	100.21.1.el5
kernel	(null)	2.6.18	92.1.6.el5

That is: name.epoch.version.release, together all comprise a single package (RPM). Supplying information about the latest, greatest RPM a user must download and install on their system is of paramount importance to us. All four parts are stored in VARCHAR2 columns. Figuring out the sort mechanism for the above can be done in SQL (using, for example, some carefully-coded SUBSTR and INSTR constructs), but can be a bit unwieldy at times. In one exercise we wanted to pre-parse RPM parts like VERSION and RELEASE as they are concatenated pieces in and of themselves. The goal was to try and store each dot-delimited piece of the VERSION and RELEASE columns in their own separate columns. For purposes of brevity the below example includes only versions and releases with four parts:

Name	Epoch	V1	V2	V3	V4	R1	R2	R3	R4
kernel	(null)	2	6	32	(0)	100	21	1	el5 (000)
kernel	(null)	2	6	18	(0)	92	1	6	el5 (000)

The new VERS1 … and REL1 … etc. columns were created with the NUMBER datatype. This way, instead of comparing VARCHAR2 strings, we can compare individual numeric values.

To achieve this, we altered our table (in this example, RPM_T) as needed.

```
alter table rpm_t add (vers1 number(10), vers2 number(10) …. rel1
number(10), rel2 number(10) … );
```

In the new example table layout above, (vers1, etc, is shortened to V1, etc. so that all columns can display in the example). At this point, the table contains the old VERSION and RELEASE columns, as well as the new VERS1 … and REL1 … and so forth, columns.
Since triggers are PL/SQL units, you will need to drop all triggers that refer to (in this example) RPM_T and recreate them on RPM. You will also need to revoke privileges from RPM_T and grant them to RPM. Indexes and constraints, however, will remain in force on RPM_T. They follow the

rename from RPM to RPM_T as they are not editionable, themselves, and are associated only with other non-editionable objects. Many, if not most, of your application upgrades can be completed just by using editions and editioning views.

3) Crossedition Triggers

If, while you have both editions available and usable, you cannot stop DML, then your application needs to keep pace with such changes. The most complex piece of EBR (and the one you will most likely use only seldomly) is the *crossedition trigger*. The crossedition trigger, for all intents and purposes, looks just like any other trigger except that it has special firing rules and a few extra keywords that tell Oracle that it is a trigger used specifically to keep multiple versions (editions) of applications in synch. Any crossedition trigger that needs to keep a parent and child edition of an application in synch is created in the child edition. The special firing rules work as follows:

1. When the parent edition's table column values are changed, you need to propagate these changes to the child edition's table columns. For this instance, you must put in place a *forward* crossedition trigger.
2. Conversely, when the child edition's table column values are changed, you need to propagate these changes to the parent edition's table columns. And, for this instance, you must put in place a *reverse* crossedition trigger.

For this example, our forward crossedition trigger looked similar to the following:

```
SQL> create or replace trigger rpm_fwdxedition
 2    before insert or update of version, release on rpm_t
 3    for each row
 4    forward crossedition
 5    declare
 6    v_verstring VARCHAR2(50) := '.'||:new.version||'.';
 7    v_relstring  VARCHAR2(50) := '.'||:new.release||'.';
 8     begin
 9        :new.ver1 := substr( v_verstring,
10        instr(v_verstring,'.',1,1)+1, instr(v_verstring,'.',1,2) -
11           instr(v_verstring,'.',1,1)-1);
12        …
…
21        :new.rel1 := substr( v_relstring,
22        instr(v_relstring,'.',1,1)+1, instr(v_relstring,'.',1,2) -
23           instr(v_relstring,'.',1,1)-1);
24        …
33       end;
34    /
Trigger created.
```

And our reverse crossedition trigger looked similar to the following:

```
SQL> create or replace trigger rpm_revxedition
 2    before insert or update of ver1, ver2, ver3, ver4, rel1, rel2,
 3    rel3, rel4, on rpm_t
 4    for each row
 5    reverse crossedition
```

```
 6    begin
 7       :new.version :=
 8          rtrim(:new.ver1||'.'||:new.ver2||'.'||:new.ver3||'.'||
 9         :new.ver4, '.');
10      :new.release :=
11         rtrim(:new.rel1||'.'||:new.rel2||'.'||:new.rel3||'.'||
12        :new.rel4, '.');
13   end;
```

As you can see what differentiates these particular triggers as crossedition triggers are the keywords, in each trigger example, just after the FOR EACH ROW command. The new keywords are FORWARD CROSSEDITION (which effectively means, whenever this trigger is fired, before making the requested DML changes, please make the changes that follow the BEGIN keyword in this trigger in the *child* edition for the RPM_T table), and REVERSE CROSSEDITION (whenever this trigger is fired, before making the requested DML changes, please make the changes that follow the BEGIN keyword in this trigger in the *parent* edition for the RPM_T table). If you are using reverse crossedition triggers, you are performing a *hot rollover,* since you are keeping both editions of your application in synch simultaneously.
Note also that these types of triggers are created upon the actual table, RPM_T, even though our other PL/SQL triggers have been recreated on the editioning view, RPM. This is because the crossedition trigger is a short-term high-availability solution that is intended solely to assist in an online application upgrade. Once you are satisfied with your upgrade results, there should be no need to continue firing any crossedition triggers, because there should be no need to continue supporting multiple versions of table columns. At some point, you will choose and stick with the version of the columns that best meets your needs, and mark the other columns as unused, and recoup the space used by the unused columns at a convenient later time.

Migrating Any Remaining Data

Though crossedition triggers will help you to keep any changes in synch between columns of a table in two editions, eventually you'll want to ensure that all data values have been successfully migrated from the old columns to the new columns. Remember that unless your crossedition triggers touch every old value, some values will have to be manually migrated to the new columns. If your table is not very big and it won't adversely affect your application to lock the entire table, you can force every row to be migrated (via the forward crossedition trigger you have in place) by executing an UPDATE statement similar to the following:

```
SQL> update rpm_t
  2     set version = version,
  3         release = release;
```

However, if this table is quite large, consider updating it a little at a time with the (new as of 11gR1) DBMS_PARALLEL_EXECUTE package. You can create a task that will update (in the below example, ROWID) chunks of a table, at a time, therefore only locking small portions of a table at a time:

```
SQL> begin
  2   dbms_parallel_execute.create_task(
  3    'update rpm_t');
  4   dbms_parallel_execute.create_chunks_by_rowid
  5    ( task_name   => 'update rpm_t',
```

```
  6      table_owner => user,
  7       table_name  => ‘RPM_T',
  8       by_row      => false,
  9       chunk_size  => 10);
 10   end;
 11   /
```

Then run the task with a chosen range of ROWIDs and level of parallelism:

```
SQL> begin
  2    dbms_parallel_execute.run_task
  3    ( task_name   => 'update rpm_t',
  4      sql_stmt    => 'update rpm_t
  5                          set version = version, release = release
  6                       where rowid between :start_id and :end_id',
  7      language_flag  => DBMS_SQL.NATIVE,
  8      parallel_level => 2 );
  9  end;
```

When you are satisfied with the result, you can simply drop this task:

```
SQL> begin
 2     dbms_parallel_execute.drop_task('update rpm_t');
 3  end;
 4  /
PL/SQL procedure successfully completed.
```

Moving to the New Edition

Once you are ready to migrate your end users to the new edition you can grant them access to the new edition:

```
SQL> grant use on edition app_edition_2 to public;
Grant succeeded.
```

And create a logon trigger that sets the new default edition any time a user logs directly into the database:

```
SQL> create or replace trigger set_edition_on_logon
   2  after logon on database
   3  begin
   4          dbms_session.set_edition_deferred( ‘APP_EDITION_2' );
   5  end;
   6  /
Trigger created.
```

And if you are using a connection pool and are using APEX you can change your configuration as follows:

```
SQL> begin
      dbms_epg.set_dad_attribute('APEX', 'database-edition',
```

```
      'APP_EDITION_2');
           end;  --If using the PL/SQL Embedded Gateway

In your dads.conf file:  PlsqlDatabaseEdition*
  --If using the Oracle Apache Http Server
```

Rolling Back

If you haven't gone live with the new edition of your application, you can drop the new child edition (cascade), set any new replacement columns you created unused, and recoup any space at a convenient later time. Keep in mind that, without a hot rollover in place (that is, without the use of REVERSE CROSSEDITION triggers), your grace period for being able to rollback an application upgrade ends once you go live with (start using) the new edition of the application.

Oracle 12c Enhancements

When this feature was introduced in Oracle 11gR2, the cardinal rule of thumb was always noneditionable objects can never depend on editionable objects because, for the noneditionable object, editionable objects are invisible during name resolution. As of 12c, however, there are some exceptions to this rule. Two types of noneditionable objects can now depend on editionable objects with the use of an *evalutation edition*. An evaluation edition is simply an edition. But to a materialized view or a virtual column, it is also a set of key words that must be present when either of these two objects is created or changed in order for them to be able to depend on editionable objects. For example:

```
CREATE MATERIALIZED VIEW refresh_sal_vals
EVALUATE USING EDITION app_edition_2
ENABLE QUERY REWRITE
UNUSABLE BEFORE EDITION app_edition_2
UNUSABLE BEGINNING WITH EDITION ora$base . . .
;
```

Conclusion

EBR is not for the convenience of the developer or the DBA. It is a high-availability solution. And like all high-availability solutions it requires that you implement it with planning and care. If as-close-to-zero downtime when performing PL/SQL application upgrades is one of your company mandates, then you can easily be brought closer to that goal with EBR. And, in the spirit of saving the best for last, it is nice to be able to inform you that EBR is freely available to any user of any version of Oracle 11gR2 on up.

About the Author

Melanie Caffrey is a senior development manager for Oracle Corporation. She is co-author of several technical publications including Expert PL/SQL Practices for Oracle Developers and DBAs, and Expert Oracle Practices: Oracle Database Administration from the Oak Table (Apress), and the SQL 101 series of articles for Oracle Magazine. She instructed students in Columbia University's Computer Technology and Applications program in New York City, teaching advanced Oracle database administration and PL/SQL development, and she is a frequent Oracle conference speaker.