Edition Based Redefinition Best Practices

Oren Nakdimon
DB Oriented
Israel

Keywords:

Online Application Upgrade; EBR; PL/SQL; Cutover; Rollover

Introduction

Edition-Based Redefinition (EBR) is a powerful feature of Oracle that enables application upgrades with zero downtime, while the application is actively used and operational. In this presentation we look at some common online application upgrade use cases, and see how to address them using EBR, based on real-life experience.

Application Upgrades

The upgrade is an organic part of every software system lifecycle. Whether it is a bug fix, a new application version introducing new features, or even a downgrade after an unsuccessful upgrade, upgrades usually require some level of interference in the ongoing operation of the system.

There are three major methods for performing an application upgrade:

- **Cold Cutover** - this is an offline method, in which the system is shut down, the new application version is installed, and then the system is restarted. During the upgrade the users cannot use the system being upgraded.
- **Warm Cutover** - in this online method the users continue to use the system while it is being upgraded, and the switch from the old version to the new version is instantaneous.
- **Hot Rollover** - this is also an online method, like the Warm Cutover, but in this method the switch between versions is smoother, and may span a longer period of time, during which some of the users still use the old version while other users already use the new version.

There are several issues which may cause difficulties in an online upgrade of a PL/SQL application:

- **Blocking** - compilation of program units is blocked by active clients that use that program units. In addition, clients that need to use a program unit are blocked during the compilation of this program unit.
- **Invalidation** - when a program unit is recompiled, other units that depend on it are automatically marked as Invalid, and have to be recompiled themselves. The problem is even bigger if multiple interrelated objects need to be changed.
- **Discarded Package State** - if between two calls of one session to some package's routines this package is recompiled, and this package has a state (e.g., it has global variables), then upon the second call the session will get the error "ORA-04068: existing state of packages has been discarded".

We'll see how EBR helps us eliminating or reducing these problems.

What is Edition Based Redefinition?

EBR is a feature set that lets you upgrade the database component of an application while it is in use, thereby minimizing or eliminating downtime. It was introduced in Oracle version 11g Release 2, and it is supported in all editions, and requires no special license.
The fundamental ability is that different copies of the same object may coexist. This ability is not supported for all the object types. In general, it is supported for "code object types" (packages, types, triggers, etc.).

Edition is a non-schema object. There must be at least one edition in the database (the initial one is ORA$BASE).
There is always one database-level default edition.
Every new edition is created as a child of an existing edition. Currently (in versions 11gR2 and 12cR1) each edition may have at most one child edition.

Each editioned object in a specific edition is either **Actual** (has its own copy in the specific edition) or **Inherited** (uses an actual object from an ancestor edition).

Use Case 1

In the first use case we need to replace a package body, which leads to the first tip:

Use Editions

Assuming that at this point there is a single edition in the database (e,g., ORA$BASE), and the clients are using intensively the package whose body is defined in the ORA$BASE edition, we'll create a new edition:

```
CREATE EDITION V1 AS CHILD OF ORA$BASE;
```

and create the new package body in the new edition, while the old package body is still in use:

```
ALTER SESSION SET EDITION=V1;
CREATE OR REPLACE PACKAGE BODY...
```

As soon as the clients move to edition V1, they will be exposed to the change in the package body.

Use Case 2

Now we need to replace a package specification, which has dependent objects. In a new edition, we'll create the new version of the package specification, and as a result the dependent objects should be recompiled as well. This can happen automatically upon the first use of each object, but this is too late in an online application upgrade scenario. We want that by the time the new edition is exposed to the clients, all its objects will already be valid. Therefore:

Explicitly Actualize All Dependent Objects Before Exposing the New Edition

It is important to note that the dependent objects may still be in status VALID until the first time they are actually used, so it is not enough to recompile only objects in status INVALID. We can do it, for example, by calling DBMS_UTILITY.VALIDATE for all the objects that are not actual in the current edition, repeatedly, until no new object is actualized.

Use Case 3

In this use case a new column, representing a new business logic, should be added to a table. An existing package should be changed accordingly.
When adding a column to a table, program units (in all the editions) that refer to the table directly become invalidated and need to be recompiled. Obviously, this is bad for online application upgrades. However, if the program unit refers to a view that is built on the table, then adding a column to the table will not invalidate the program unit.
Edition Based Redefinition introduces a new type of views - the **Editioning Views**. An Editioning View is a view (hence editionable), that is deliberately limited:

- There can be only one editioning view per table
- It may contain only the SELECT and FROM clauses
- The FROM clause may refer to exactly one table
- The SELECT list may contain only columns and aliases (no expressions)

In addition, and unlike with regular views, DML triggers may be defined on editioning views.
All of this makes the Editioning Views ideal as the interface between the application and the tables.

Do Not Refer to Tables in Your Code, Only to Editioning Views

For example, we'll create a table, define an editioning view that covers the table, and refer to the editioning view (rather than the table) in our program unit:

```
CREATE TABLE PEOPLE$T (
  ID NUMBER(9) NOT NULL CONSTRAINT PEOPLE_PK PRIMARY KEY,
  FIRST_NAME   VARCHAR2(15) NOT NULL,
  LAST_NAME    VARCHAR2(20) NOT NULL,
  PHONE_NUMBER VARCHAR2(20)
);

CREATE EDITIONING VIEW PEOPLE AS SELECT * FROM PEOPLE$T;

CREATE PROCEDURE UPDATE_PHONE_NUMBER (
  I_PERSON_ID IN PEOPLE.ID%TYPE,
  I_PHONE_NUMBER IN PEOPLE.PHONE_NUMBER%TYPE
) AS
BEGIN
  UPDATE PEOPLE
     SET PHONE_NUMBER = I_PHONE_NUMBER
     WHERE ID = I_PERSON_ID;
END UPDATE_PHONE_NUMBER;
/
```

Use Case 4

In this use case a new column should be added to a table, and it replaces an existing column. The upgrade method that is required here is "warm cutover".
Assuming that our code now refers to editioning views and not to the actual tables, we can add the new column without causing invalidations. In a new edition, we'll replace the old column by the new column in the corresponding editioning view. We will also copy the data from the old column to the new column, including any needed conversions. But we need a way to apply changes done to the old column, by clients of the old edition, to the new column, until we expose the new edition and make all the clients use the new edition.
For this, EBR introduced a new type of triggers - the **Crossedition Triggers**. A crossedition trigger is a "bridge" for moving data between editions. It is temporary in nature - it is relevant only for the transition period from the old edition to the new edition.
In our use case, we need to use a FORWARD CROSSEDITION trigger. Forward crossedition triggers transform pre-upgrade representation to post-upgrade representation.

Use Forward Crossedition Triggers to Propagate Changes Done in the Old Edition to the New Edition

In the following example, the new column keeps the same value as the old column, but multiplied by 10:

```
CREATE OR REPLACE TRIGGER PEOPLE_FCE_TRIG
    BEFORE INSERT OR UPDATE OF OLD_COLUMN ON PEOPLE$T
    FOR EACH ROW
    FORWARD CROSSEDITION
BEGIN
    :NEW.NEW_COLUMN := :NEW.OLD_COLUMN * 10;
END PEOPLE_FCE_TRIG;
/
```

Note that this trigger will be fired by changes done in the old edition, and will not be fired by changes done in the new edition.

Use Case 5

This use case is identical to use case 4, except for the upgrade method. Here, it should be a "hot rollover". This new requirement implies that both the old edition and the new edition will be used by clients at the same time. It means not only that changes done in the **old** edition should be applied in the **new** edition (as we did with the forward crossedition trigger), but also that changes done in the **new** edition should be applied in the **old** edition. For this we'll use a REVERSE CROSSEDITION trigger. Reverse crossedition triggers transform post-upgrade representation to pre-upgrade representation.

Use Reverse Crossedition Triggers to Propagate Changes Done in the New Edition to the Old Edition

Continuing the previous example:

```
CREATE OR REPLACE TRIGGER PEOPLE_RCE_TRIG
    BEFORE INSERT OR UPDATE OF NEW_COLUMN ON PEOPLE$T
    FOR EACH ROW
    REVERSE CROSSEDITION
BEGIN
    :NEW.OLD_COLUMN := :NEW.NEW_COLUMN / 10;
END PEOPLE_FCE_TRIG;
/
```

Note that this trigger will be fired by changes done in the new edition, and will not be fired by changes done in the old edition.

Unlike other editioned objects, Crossedition Triggers has an immediate effect on active (pre-upgrade) editions, and therefore extra care should be taken when creating them.

Create Crossedition Triggers as DISABLED, and Enable Them Only if They Compile Successfully

Exposing the New Edition

For a **cutover**, exposing the new edition can be done by changing the database default edition:

```
ALTER DATABASE DEFAULT EDITION = <edition-name>;
```

Note: if an open session has already accessed a package in the old edition, and this package has a state (e.g., it has global variables), and this package has been changed in the new edition, then upon the first call of this session to that package in the new edition it will get the error "ORA-04068: existing state of packages has been discarded".

For a **rollover**, exposing the new edition can be done by an AFTER LOGON trigger, so new sessions will use the new edition, while already open sessions will continue using the old edition. For example:

```
CREATE OR REPLACE TRIGGER SET_EDITION_ON_LOGON_TRIG
  AFTER LOGON ON DATABASE
BEGIN
  DBMS_SESSION.SET_EDITION_DEFERRED('<edition-name>');
END SET_EDITION_ON_LOGON_TRIG;
/
```

The edition to use can be also specified at the statement level (using DBMS_SQL.PARSE), at the service level (in version 11.2.0.2 and later), or by the client when opening the connection (from SQL*Plus, JDBC, etc.).

Conclusion

We covered several common use cases of online application upgrade and saw how EBR allows us to perform them with no down time. We did not cover advanced topics, restrictions, limitations and more; for example:

- Enabling editions to users

- Granting use of editions
- Retiring editions
- Dropping editions
- Jobs and editions
- Using editions with schema changes (other than the ones discussed)
- And more...

To get more information please don't hesitate to contact me (see contact details below).

Contact address:

Oren Nakdimon
DB Oriented
POB 145
Zurit 2010400
Israel

Phone:	+972(0)54-4393763
Email	oren@db-oriented.com
Internet:	www.db-oriented.com
Twitter:	@DBoriented