Edition-Based Redefinition

Christian Antognini

Trivadis AG

Zurich, Switzerland

Upgrading critical applications can be very difficult. One of the main problems is that for reasons of availability, long downtimes cannot be periodically scheduled. Therefore, for such applications, it is desirable to implement online upgrades. This requires that the application in question, as well as any software used by the application (e.g. the database engine) all support online upgrades. Oracle has recognized this problem for years. Unfortunately, up to and including Database 11g Release 1, only a limited number of features have been implemented for that purpose. As of Oracle Database 11g Release 2, this situation has changed greatly. With edition-based redefinition, Oracle Database offers real support for implementing online upgrades. The aim of this paper is to provide an overview of this new feature.

Concept

In an Oracle database, an object is identified by its name, the user (schema) it belongs to, and its *edition*. The aim of the latter, which is new in Oracle Database 11g Release 2, is to support several copies (versions) of a given object. The objects for which several copies might be created are called *editioned objects*. Objects that cannot have several copies, which are referred to as *noneditioned objects*, have no edition (in the data dictionary, the edition is set to `NULL`).

The aim of supporting two or more versions of a given object is straightforward. While one set of objects is used by an application, another set can not only be redefined, but also used for test purposes. In other words, it is possible to redefine and test a set of objects in an isolated environment before making them live.

Not all object types support editions. In fact, only the following object types, which are called *editionable object types*, support this new concept.

- `FUNCTION`
- `LIBRARY`
- `PACKAGE` and `PACKAGE BODY`
- `PROCEDURE`
- `SYNONYM (PUBLIC SYNONYM is a noneditionable object type)`
- `TRIGGER`
- `TYPE` and `TYPE BODY`
- `VIEW`

Basically, these are the object types used to extend the functionality of the database engine through PL/SQL code or external libraries. The only exceptions are the views and the synonyms. Notice that neither object types related to Java nor object types that store data (e.g. tables and indexes) are editionable. Objects that do not support editions are called *noneditionable object types*.

Enabling Editions for a User

The first thing to do when you want to use edition-based redefinition is to enable it for the owner of the objects that have to be redefined. In fact, editioned objects can only be owned by users that have been explicitly enabled to do so. To do this, you either have to specify the `ENABLE EDITIONS` clause at the time you create a user or, for a user that already exists, execute an `ALTER USER` statement as shown in the following example.

```
SQL> ALTER USER cha ENABLE EDITIONS;
```

It is important to point out that the operation of enabling a user to create editioned objects is irreversible. In other words, something like a DISABLE EDITIONS clause does *not* exist.

To check whether a user has the capability of creating editioned objects, you can use a query like the following.

```
SQL> SELECT editions_enabled
  2  FROM dba_users
  3  WHERE username = 'CHA';


EDITIONS_ENABLED
----------------
Y
```

Once an owner for the editioned objects is available, you have to create an edition to identify a specific version of the database objects.

Creating Editions

Every database must have at least one edition. Therefore, when you create a new database, an edition called ORA$BASE is created by default. A database supports several editions organized as a hierarchy with one root edition (by default ORA$BASE) and a parent-child relationship between every other edition. For example, Figure 1 illustrates the case of a database that has four editions, the default one and three user-defined editions (let's say that every one of the user-defined editions is related to a release of the application; hence, the names REL1, REL2 and REL3).

Figure 1: Editions are organized in a hierarchy

To create an edition, you have to use the CREATE EDITION statement. To execute this SQL statement, you require the CREATE ANY EDITION system privilege. For example, if you want to create the editions shown in Figure 1, you have to execute the following SQL statements. Notice how the parent-child relationship is specified with the AS CHILD OF clause.

```
SQL> CREATE EDITION rel1 AS CHILD OF ora$base;
SQL> CREATE EDITION rel2 AS CHILD OF rel1;
SQL> CREATE EDITION rel3 AS CHILD OF rel2;
```

By default, a new edition can only be used by the user who created it and by SYS. You can provide the necessary privileges to other users by granting the USE ON EDITION privilege. As the following SQL statements shows, you can grant the privilege either to a specific user or to all users by specifying PUBLIC as grantee.

```
SQL> GRANT USE ON EDITION rel2 TO cha;
SQL> GRANT USE ON EDITION rel3 TO PUBLIC;
```

To display information about all available editions, you can select either ALL_EDITIONS or DBA_EDITIONS. The following example shows a hierarchical query that you can use for that purpose. Note that editions are objects that do not belong to a specific schema and, therefore, ALL_EDITIONS and DBA_EDITIONS do not provide a column specifying the owner of the edition.

```
SQL> SELECT *
  2  FROM all_editions
  3  CONNECT BY parent_edition_name = PRIOR edition_name
  4  START WITH parent_edition_name IS NULL
  5  ORDER BY level;


EDITION_NAME   PARENT_EDITION_NAME  USABLE
-------------  -------------------- -------
ORA$BASE                            YES
REL1           ORA$BASE             YES
REL2           REL1                 YES
REL3           REL2                 YES
```

Since a database supports several editions, a user (or, more likely, an application) must be able to specify which edition must be used to reference the right version of the editioned objects. In other words, the so-called *session edition* has to be selected.

Selecting Editions

To select the session edition, APIs (e.g. OCI and JDBC) and tools (e.g. SQL*Plus and Data Pump) make it possible to specify it when a new database connection is created. Note that when no edition is specified, the *default database edition*, which is defined by the database property named DEFAULT_EDITION, is used. By default, the property is set to ORA$BASE. As with any other database property, you can modify it with the ALTER DATABASE statement. The following example shows how to set the default database edition to REL1.

```
SQL> ALTER DATABASE DEFAULT EDITION = rel1;


SQL> SELECT property_value
  2  FROM database_properties
  3  WHERE property_name = 'DEFAULT_EDITION';


PROPERTY_VALUE
---------------
REL1
```

It is essential to stress two points about such a configuration. First, when you modify the default database edition, the database engine automatically grants the USE ON EDITION privilege for the specified edition to PUBLIC. This is necessary because the default database edition must be available for all users. Second, the concept of default database edition is basically available for compatibility purposes with applications that are not able to set the edition when they connect to the database. Hence, the session edition should be specified at connect time whenever possible.

As of version 11.2.0.2, it is also possible to specify a default edition at the service level. To do so a new parameter named EDITION is available when creating or modifying a service through the CREATE_SERVICE or MODIFY_SERVICE procedure of the DBMS_SERVICE package. The following example illustrates this.

```
SQL> BEGIN
  2    dbms_service.create_service(service_name => 'CRM',
  3                               network_name => 'CRM.trivadis.com',
  4                               edition      => 'CRM_REL1');
  5    dbms_service.start_service(service_name => 'CRM');
  6  END;
  7  /
```

This feature is especially useful when several applications are consolidated into a single database. In fact, in such a case, a single default database edition for the whole database is useless. It is much better to have a single service with its own default edition for each application.

Once connected, you can also modify the session edition through the ALTER SESSION statement. Just be aware that this SQL statement cannot be executed when the current session has an open transaction. The following SQL statement shows how to set REL1 as session edition.

```
SQL> ALTER SESSION SET EDITION = rel1;
```

It is also important to point out that the previous SQL statement must be a top-level SQL statement and, therefore, you cannot executed it in PL/SQL. So, if you need to modify the session edition in PL/SQL (e.g. in a logon trigger), you have to use the SET_EDITION_DEFERRED procedure in the DBMS_SESSION package.

Two methods are available to display the session edition currently in use. First, and only for the current session, you can use the SESSION_EDITION_NAME attribute of the USERENV context. SQL*Plus users can also take advantage of the SHOW EDITION command as a shortcut. Second, and for all sessions, you can select the V$SESSION view. The following queries illustrate.

```
SQL> SELECT sys_context('USERENV','SESSION_EDITION_NAME') edition_name
  2  FROM dual;

EDITION_NAME
-------------
ORA$BASE

SQL> SHOW EDITION

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

SQL> SELECT DISTINCT object_name AS edition_name
  2  FROM v$session, dba_objects
  3  WHERE session_edition_id = object_id;

EDITION_NAME
-------------
ORA$BASE
```

When both the owner of the editioned objects and the edition are ready, then it is time to create the editioned objects themselves.

Creating Editioned Objects

Editioned objects are created like any other object. The only thing that you have to do, in addition to using a user that can own editioned objects, is select the edition with which the objects must be associated. For instance, you can use the following SQL statements to create a procedure that is associated to the edition REL1.

```
SQL> ALTER SESSION SET EDITION = rel1;
```

```
SQL> CREATE OR REPLACE PROCEDURE hello IS
  2  BEGIN
  3    dbms_output.put_line('Hello from REL1');
  4  END;
  5  /
```

Another way to create editioned objects, is to enable the editions for a user that already owns objects. In this case, all objects are automatically associated with the root edition.

The aim of creating editioned objects is to be able to redefine them. So let's discuss how such an operation is performed.

Redefining Editioned Objects

The first operation you have to perform in order to redefine an object is to create a new edition and, if this operation is not executed by the owner of the editioned object, to grant the necessary privileges. How to perform these operations is described in the previous sections. When a new edition is created, all objects that are associated with the parent edition are inherited by the child edition. But, be careful, the objects will not be associated with the child edition. They are only made visible to the child edition. For instance, based on the hierarchy depicted in Figure 1, a procedure created in the REL1 edition is also visible to REL2 and REL3 editions (provided that it is not dropped).

To redefine editioned objects, you have to select the new edition and either alter, add, drop or, in case of stored programs, recompile the editioned objects through regular DDL statements. For example, with the following SQL statements, the procedure previously created under the REL1 edition is altered in REL2 edition and dropped in REL3 edition.

```
SQL> ALTER SESSION SET EDITION = rel2;

SQL> CREATE OR REPLACE PROCEDURE hello IS
  2  BEGIN
  3    dbms_output.put_line('Hello from REL2');
  4  END;
  5  /

SQL> ALTER SESSION SET EDITION = rel3;

SQL> DROP PROCEDURE hello;
```

Now that the procedure has been redefined, two independent objects are stored in the data dictionary. Consequently, you can choose which one you want to use. To make this choice, you have to select the corresponding edition as shown in the following example. Also notice that USER_OBJECTS (the same applies to DBA_OBJECTS and ALL_OBJECTS) shows the version associated with the session edition.

```
SQL> ALTER SESSION SET EDITION = rel1;

SQL> SELECT edition_name
  2  FROM user_objects
  3  WHERE object_name = 'HELLO';

EDITION_NAME
-------------
REL1

SQL> EXECUTE hello
Hello from REL1
```

```
SQL> ALTER SESSION SET EDITION = rel2;

SQL> SELECT edition_name
  2  FROM user_objects
  3  WHERE object_name = 'HELLO';

EDITION_NAME
-------------
REL2

SQL> EXECUTE hello
Hello from REL2

SQL> ALTER SESSION SET EDITION = rel3;

SQL> SELECT edition_name
  2  FROM user_objects
  3  WHERE object_name = 'HELLO';

no rows selected
```

If you want to display all versions that are stored in the data dictionary, you have to select either USER_OBJECTS_AE, DBA_OBJECTS_AE or ALL_OBJECTS_AE. In the following example notice that the object associated with the REL3 edition is still available but marked has non-existent.

```
SQL> SELECT edition_name, object_type
  2  FROM user_objects_ae
  3  WHERE object_name = 'HELLO';

EDITION_NAME  OBJECT_TYPE
------------- -------------
REL1          PROCEDURE
REL2          PROCEDURE
REL3          NON-EXISTENT
```

In summary, the typical steps that you have to carry out to redefine editioned objects are the following:
- Create a new edition and set it as session edition.
- Modify the editioned objects and ensure that all objects are valid.
- Check whether the application works as expected with the new edition.
- Permanently switch to the new edition.

The techniques described up to now are useful to redefine objects of an editionable object type. Redefining tables, which are not editionable, requires the introduction of another concept, called *editioning view.*

Editioning Views

Tables are not editionable and, therefore, you cannot take advantage of editions to redefine their structure. Instead, you can cover every table with an editioning view and, at the same time, in the application you can replace all[1] references to tables with references to editioning views. Simply put, you can build a (transparent) layer between the application and the physical database design.

---

[1] It might not be possible to replace all references. For example, a TRUNCATE statement cannot be executed on a view. That said, it is not necessarily a problem. In fact, a TRUNCATE statement works independently of the table's structure.

An editioning views is a regular view with a few special characteristics:

- An editioning view is intended only to select and, optionally, to provide aliases for a subset of columns in a single table. It does not support operations like joins, subqueries, set operators, aggregations and hierarchical queries.
- An editioning view supports DML triggers. As a result, the triggers that you would usually specify at the table level should be specified at the view level. Such triggers fire only when DML operations target the editioning view. In other words, they do not fire when DML operations target the table on which the editioning view is based.
- An editioning view does not support INSTEAD OF triggers.

To create an editioning view, you have to add the keyword EDITIONING to the CREATE VIEW statement. The following SQL statements show an example where a table is covered by an editioning view to map the physical name of the columns to a logical name. In addition, a DML trigger is associated with the view as well (note that the same trigger could not be created on a regular view).

```
SQL> DESCRIBE persons
 Name                         Null?     Type
 ---------------------- -------- ----------------
 ID                     NOT NULL NUMBER(10)
 FIRST_NAME             NOT NULL VARCHAR2(100)
 LAST_NAME              NOT NULL VARCHAR2(100)
 EMAIL                  NOT NULL VARCHAR2(100)

SQL> RENAME persons TO persons_tab;

SQL> CREATE EDITIONING VIEW persons AS
  2   SELECT id, first_name AS firstname, last_name AS lastname, email
  3   FROM persons_tab;

SQL> CREATE TRIGGER persons_bi_trg
  2   BEFORE INSERT ON persons FOR EACH ROW
  3   BEGIN
  4     :new.id := persons_seq.nextval;
  5   END;
  6   /
```

During an upgrade, editioning views might be set in read-only or read-write mode. To make an upgrade easier, you should set in read-only mode all editioned views based on tables that have to be redefined. However, as a result, the application is only able to read data from those tables, not to modify it. When you cannot consider the read-only mode, you have to assess whether it is necessary to create so-called *crossedition triggers*.

Crossedition Triggers

A crossedition trigger is used when two conditions are met. First, the structure of a table has to be redefined during an upgrade. Second, the editioning view covering the table to be redefined cannot be set in read-only mode. For instance, let's say that an application need full access to a table that contains a column storing email addresses (like the one used as example in the previous section). Your task is to split that column into two columns: one storing the recipient name and the other storing the domain name. Adding the two columns to the table is, most of the time[2], not a problem. For instance, you might simply execute the following SQL statement.

---

[2] Getting a table lock might be problematic on very busy tables. As a result, you might have to set a timeout with the DDL_LOCK_TIMEOUT initialization parameter. To redefine tables, the DBMS_REDEFINITION package might also be useful.

```
SQL> ALTER TABLE persons_tab ADD (
  2    email_recipient VARCHAR2(100),
  3    email_domain VARCHAR2(100)
  4  );
```

Redefining the table is a good starting point. But, what happens to the data stored into it? Basically, you have to fulfill two requirements to finish the redefinition correctly. First, the data which is modified by the application during the upgrade has to be stored in the new structure. Second, the data already stored in the redefined table has to be converted in the new structure as well.

To fulfill the first requirement, you can create a *forward crossedition trigger* in the new edition (you should *never* modify the old edition). The aim of such a trigger is to transform a row from the old structure into the new one. In the example discussed here, the following SQL statement fulfills this requirement.

```
SQL> CREATE TRIGGER persons_fc_trg
  2    BEFORE INSERT OR UPDATE ON persons_tab FOR EACH ROW
  3    FORWARD CROSSEDITION
  4    DISABLE
  5  BEGIN
  6    :new.email_recipient :=
  7      regexp_substr(:new.email,'(.*)@',1,1,NULL,1);
  8    :new.email_domain :=
  9      regexp_substr(:new.email,'@(.*)',1,1,NULL,1);
 10  END;
 11  /
```

Note that even though the trigger is created in the new edition, it fires only when a DML statement is executed by a session using an ancestor edition to that in which the trigger was created. For this reason, you have to explicitly define that it is a forward crossedition trigger by specifying the FORWARD CROSSEDITION clause. Also notice that you should create the trigger in the disabled state. This is important to prevent an invalid trigger from affecting the availability of the application. Hence, you should enable it only when you know that it is valid.

To fulfill the second requirement, you have to apply the transformation to the already stored data. To do this, there are several possibilities. The easiest one, which is not the best one from a performance point of view, it is to fire the trigger for every row already stored in the redefined table. To do so, the DBMS_SQL package provides a new version of the PARSE function. The following PL/SQL blocks illustrates. Notice how a dummy UPDATE statement is used to fire the trigger specified through the APPLY_CROSSEDITION_TRIGGER parameter.

```
SQL> DECLARE
  2    c INTEGER;
  3    r INTEGER;
  4  BEGIN
  5    c := dbms_sql.open_cursor;
  6    dbms_sql.parse(
  7      c => c,
  8      statement => 'UPDATE persons SET email_domain = email_domain',
  9      language_flag => dbms_sql.native,
 10      apply_crossedition_trigger => 'persons_fc_trg'
 11    );
 12    r := dbms_sql.execute(c);
 13    dbms_sql.close_cursor(c);
 14    COMMIT;
 15  END;
 16  /
```

It is essential to recognize that in order to avoid an inconsistent state, you have to carry out the operations needed to fulfill these two requirements in the following order:

- Enable the forward edition trigger.

```
SQL> ALTER TRIGGER persons_fc_trg ENABLE;
```

- Wait for every transaction on the redefined table to either be committed or rolled back. For that purpose, a new function called WAIT_ON_PENDING_DML has been added to the DBMS_UTILITY package. If this step is not carried out, the transactions that were open while enabling the forward edition trigger might lead to an inconsistent state.

```
SQL> DECLARE
  2    r BOOLEAN;
  3    scn INTEGER;
  4  BEGIN
  5    r := dbms_utility.wait_on_pending_dml(
  6          tables => 'CHA.PERSONS_TAB',
  7          timeout => NULL,
  8          scn => scn
  9        );
 10  END;
 11  /
```

- Start the PL/SQL block that fires the forward edition trigger for every row.

Once these operations are terminated, you might also need to create some new constraints for the redefined table. For instance, in this case the new columns are made not nullable.

```
SQL> ALTER TABLE persons_tab MODIFY (
  2    email_recipient NOT NULL,
  3    email_domain NOT NULL
  4  );
```

The example illustrated so far shows what you can do to propagate the changes applied by an application working with the old edition to the data structures used by the new edition. Another situation that you have to consider is what happens if both the old and the new edition are used concurrently. This happens when a hot rollover to the new edition is required. To fulfill this new requirement, you have to use another kind of trigger, the *reverse crossedition trigger*. Simply put, the aim of such a trigger is to do the opposite of a forward crossedition trigger. In other words, its function is to transform a row from the new structure to the old one. The following SQL statement shows an example.

```
SQL> CREATE TRIGGER persons_rc_trg
  2    BEFORE INSERT OR UPDATE ON persons_tab FOR EACH ROW
  3    REVERSE CROSSEDITION
  4    DISABLE
  5  BEGIN
  6    :new.email := :new.email_recipient || '@' || :new.email_domain;
  7  END;
  8  /
```

In this case you have to create the trigger in the new edition as well (again, you should never modify the old edition). In addition, note that a reverse crossedition trigger fires only when a DML statement is executed by a session using the edition in which the trigger was created (or a descendent of it). For this reason, you have to explicitly define that it is a reverse crossedition trigger by specifying the REVERSE CROSSEDITION clause.

The examples of crossedition triggers shown here are simple. When the complexity of the redefinition increases, for example when SQL statements have to be executed inside the triggers, several other issues must be considered. Since this short introduction cannot deal with all such details, it is advised to refer to the documents listed at the end of the paper for additional information.

Dropping Editions

To drop an edition you have to use the `DROP EDITION` statement. To execute this statement, you require the `DROP ANY EDITION` system privilege. In addition, the following conditions must be met:
- The edition is not the latest database edition.
- The edition is not the default database edition.
- The edition is not in use as session edition.
- The edition is either the root or the leaf of the hierarchy.
- If the edition is the leaf of the hierarchy and it still has associated objects, the `CASCADE` option must be specified.
- If the edition is the root of the hierarchy, the child edition must not inherit objects from the root edition. In other words, all inherited objects must be redefined by the child edition.

Dropping an edition that is no longer used is optional. Instead of dropping an edition which is no longer used, you might also choose to simply retire it by revoking the `USE ON EDITION` privilege for every user.

Limitations

In Oracle Database 11g Release 2, there are two important implementation limitations related to the hierarchies that can be built with editions:
- Every edition can have at most one child. As a result, a hierarchy like the one shown on the left of Figure 2 is *not* allowed.
- Every database has one, and only one, root edition. Therefore, a database cannot have two hierarchies like the ones shown on the right of Figure 2.

While the former limitation is not decisive, the latter might be critical. In fact, it is not permitted to have independent editions for every application sharing the same database for consolidation purposes. For example, in Figure 2, the idea would be to have one hierarchy for the CRM application and another one for the HR application.

Figure 2: Examples of hierarchies that are not allowed

The other important limitations are related to indexes and constraints. Since both of them can only be created on tables, there is no way to take advantage of editions with them. In case of new indexes, you can limit their impact by making them invisible. With constraints, however, it might not be possible to use a definition that works correctly with more than one edition. The only general exception is related to check constraints. In fact, with them, you can base their logic on the session edition.

Conclusion

Even though there are still some limitations, edition-based redefinition provides completely new features that make online upgrades possible. That said, the complexity of such upgrades must not be underestimated.

References

Oracle White Paper, Edition-Based Redefinition in Oracle Database 11g Release 2

Oracle Database 11g Release 2 documentation, Advanced Application Developer's Guide

Oracle Database 11g Release 2 documentation, SQL Language Reference

Oracle Database 11g Release 2 documentation, PL/SQL Packages and Types Reference

About the Author

Since 1995, Christian Antognini has focused on understanding how the Oracle database engine works. He is currently working as a principal consultant at Trivadis in Zurich, Switzerland. If Christian is not helping one of his customers get the most out of Oracle, he is somewhere lecturing on application performance management. He is a proud member of the OakTable Network and the author of the book *Troubleshooting Oracle Performance* (Apress, 2008).