ORACLE

Edition-Based Redefinition Technical Deep Dive

An Oracle Database capability for online application upgrades

February 27, 2020

Confidential: Public

PURPOSE STATEMENT

This document provides an overview of features for Edition-Based Redefinition. It is intended solely to help you assess the business benefits of zero downtime application upgrade and to plan your I.T. projects.

DISCLAIMER

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle software license and service agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This document is for informational purposes only and is intended solely to assist you in planning for the implementation and upgrade of the product features described. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

Due to the nature of the product architecture, it may not be possible to safely include all features described in this document without risking significant destabilization of the code.

TABLE OF CONTENTS

INTRODUCTION

Large mission critical applications may experience downtime for tens of hours, or even longer, while the application's database components are updated during an application upgrade. Oracle Database introduced Edition-Based Redefinition (EBR), a revolutionary capability that allows online application upgrade with uninterrupted availability of the application.

EBR functions by maintaining two versions of the application simultaneously. When the installation of the upgrade is complete, the pre-upgrade application and the post-upgrade application can be used at the same time. Therefore, an existing session can continue to use the pre-upgrade application until its user decides to end it; and all new sessions can use the post-upgrade application. The pre-upgrade application can be retired after all sessions have disconnected from it. In other words, the application as a whole enjoys hot rollover[1] from the pre-upgrade version to the post-upgrade version.

To take advantage of the capability, the application's database backend must be enabled to use EBR by making some one-time schema changes. Also, the script that performs the application upgrade must be written in such a way to use EBR's features. Therefore, EBR adoption and subsequent use is the perogative of the development shop.

To achieve online application upgrade[2], the following conditions must be met:

- The installation of changed database objects must not perturb live users of the pre-upgrade application.
- Transactions done by the users of the pre-upgrade application must be reflected in the post-upgrade application.
- Transactions done by the users of the post-upgrade application must be reflected in the pre-upgrade application.

Oracle Database enables this through a revolutionary capability called *Edition-Based Redefinition* (EBR).

Using EBR:

- Code changes are installed in the privacy of a new *edition*.
- Data changes are made safely by writing only to new columns or new tables that are invisible to the old edition. This is done via an *editioning view* which exposes a different projection of a table into each edition so that each edition just sees its own columns.
- A *cross-edition trigger* propagates data changes made by the old edition into the new edition's common columns, or (in hot-rollover) vice-versa.

This whitepaper explains EBR in detail through the concepts that underpin it and by illustrating the basic operations with minimal code samples. It then presents a series of realistic use cases in order of increasing complexity. The discussion of these use cases should prepare the user for designing and implementing scripts for the online upgrade of real-world applications.

This whitepaper is not a reference manual. The relevant SQL syntax and PL/SQL APIs are documented in the Oracle Database SQL Language Reference book, the Oracle Database PL/SQL Language Reference book, and the Oracle Database PL/SQL Packages and Types Reference book; and the catalog views that expose facts about the relevant objects are documented in the Oracle Database Reference book. Rather, it aims to explain the concepts and the use of EBR at a depth that is not practical in the Oracle Database Documentation Library. In this way, it complements and extends the treatment in the Oracle Database Advanced Application Developer's Guidebook and the Oracle Database Administrator's Guidebook.

[1] Transactions done by the users of the post-upgrade application must be reflected in the pre-upgrade application.

[2] The term upgrade will be used in this whitepaper to denote both that and patch. The term patch is conventionally used to denote changes that are made to a system to correct behavior which deviates from its current functional specification; and the term upgrade is conventionally used to denote changes that are made to enhance behavior so that it conforms to a new version of the functional specification. However, this distinction in intention has no consequence for the nature of the changes that are made to an application's database objects. A change, for example, to a PL/SQL unit, to table data, or to table structure requires the same steps and has the same consequences whether the intention is to patch or to upgrade.

EDITION-BASED REDEFINITION

EBR depends upon three new kinds of objects: the *edition*, the *editioning view*, and the *cross-edition trigger*. Each of them is used for the following use cases.

1. If the application upgrade will change only views, synonyms, and PL/SQL objects, then the *edition alone* is sufficient to allow these changes to be made while the application remains online. This type of change is common when, for example, new presentations of data or new workflows are required.
2. If changes to table data or structure are restricted to only those tables that are not changed via the ordinary end-user interfaces, then *the edition together with the editioning view* are sufficient to allow these changes to be made while the application remains online. Tables whose data parameterizes the user interface layout or workflows meet this condition. So do tables that hold the catalog of wares for a shopping application.
3. If changes to table data or structure are required for those tables that are changed routinely by the end-user, then *the edition, the editioning view, and the crossedition trigger* must be used in concert to allow these changes to be made while the application remains online.

EBR Use Case

Suppose that an application has 1,000 mutually dependent tables, views, PL/SQL units, and triggers, all of which are owned by more than one user, and that the source code of these objects makes references to other objects, often by schema-qualified name. Suppose that the upgrade needs to change only 10 of these. Figure 1. illustrates this.

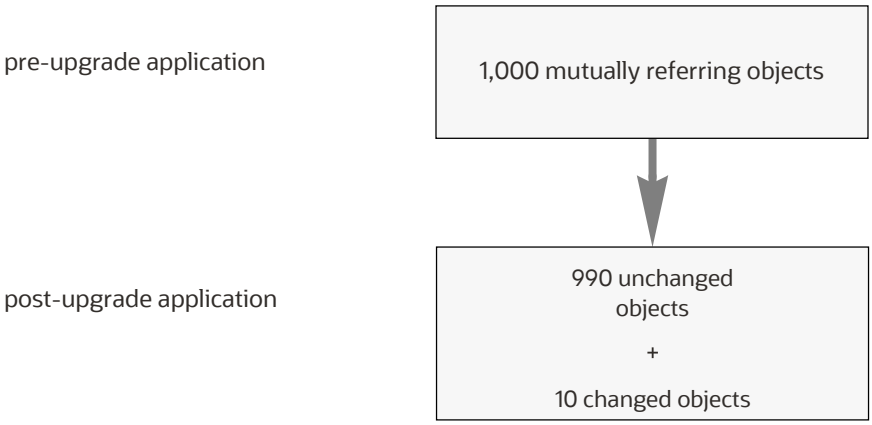

Figure 1. The challenge of changing dependent objects

Of course, the 10 objects cannot be changed in place because many of the other 990 refer to them and doing so would change the meaning of the pre-upgrade application. The only dimensions that identify the intended object when one object refers to another, are its name and its owner: these naming mechanisms are not rich enough to support online application upgrade.

A short digression on the viability of an approach that uses schemas and synonyms to explicitly enrich the naming mechanisms manually will be useful. It would be possible for a customer to impose a discipline where every reference from an object to another "primary" object in a different schema is made via a "secondary" private synonym in the referring object's schema. In such a regime, it might seem that online application upgrade could be achieved by installing the complete upgraded application in a new set of schemas with appropriately redefined private synonyms. This would, at least, allow the source text of the 900 "primary" objects for which no change was intended to remain unchanged in the source control system. There would, however, be some effort in redefining the synonyms in the source control system, but this could, presumably be done automatically. This approach suffers from a number of disadvantages with respect to using EBR:

- It requires specific design by the application developer.
- Every "primary" object needs to be duplicated which costs both space and the time it takes to run the DDL statements.
- The strategy for handling changes to table data and structure might be feasible to keep both pre-upgrade and post-upgrade versions in sync, and vice versa, but it would be dauntingly complex and, because of that, subject to an appreciable risk of error.

- Some applications issue DDL statements as part of their normal response to ordinary end-user interaction. The effort to design and implement a scheme to reflect such changes forwards and backwards between the pre-upgrade and the post-upgrade applications would be huge.

With applications of sufficient size and complexity, various issues arise (too complicated to describe in this whitepaper) that defeat the scheme. It is, quite simply, not generally viable "in the large". EBR supports this high-level philosophy of the manual approach just described but overcomes all of its disadvantages.

The Edition

An edition is a new, nonschema object type, uniquely identified, therefore, by just its name. Editions are listed in the *DBA_Objects* catalog view family where, just like the nonschema object type *directory*, they appear to be owned by *SYS*[3]. Every database, whether brand new or the result of an upgrade from an earlier version, non-negotiably has at least one edition. Immediately on creation or upgrade of an Oracle database, there is exactly one edition with the name *Ora$Base*.

A new edition must be created as the child of an existing one; the syntax of the *create edition* statement allows that the parent edition be identified using the *as child of* clause. An edition may have no more than one child. The *create edition* statement allows the *as child of* clause to be omitted to mean that the new edition is created as the child of the *leaf edition.*

Every foreground database session, at every moment throughout its lifetime, non-negotiably *uses* a particular edition[4]. This is reflected as the value of the new parameter *Current_Edition_Name* in the *Userenv* namespace for the *Sys_Context ()* builtin. A new *not null* database property, *Default_Edition*, listed in *Database_Properties*, specifies the edition that a session will use immediately on connection if the connect syntax does not nominate an explicit edition. A side effect of making an edition the default is to grant *Use* on it to *public*.

Code_1 shows the SQL statement to set this.

```
-- Code_1
alter database default edition = Some_Edition
```

When a new connection is made, it is possible to specify the edition the session should (initially) use. A new alter session command allows the edition that a session is using to be changed. However, this command is legal only as a top-level server call; an attempt to issue it using PL/SQL's dynamic SQL will cause an error. Further, an attempt to change the edition that a session is using will fail if there is any uncommitted DML[5].

Editionable object types, Editions-enabled users, and Editioned objects

Views (and therefore editioning views), synonyms, and all the kinds of PL/SQL objects type[6] (and therefore cross-edition triggers) are **editionable object types**. There are no other editionable object types. For example, *table* is not an editionable object type; nor is *java class*.

The nonschema object type *user* has a new *Y/N* property, shown in *DBA_Users.Editions_Enabled*. The users that have this property as Y are called **editions-enabled users**. This can be set with the *create user* command or changed with the *alter user* command, but only from *N* to Y. However, certain users (*SYS*, *SYSTEM*, and any user listed in the *DBA_Registry*) cannot be *editions-enabled* and the attempt to enable editions on them will cause an error.

A database object that is an editionable object type and is also owned by an editions-enabled user is called an **editioned object**. An object that is not of an editionable object type can never be editioned. An object of an editionable object type that is owned by a user that is not editions-enabled is not editioned, but it will irrevocably become so when its owner is altered to become editions-enabled. An object that is not editioned is uniquely identified, by just its owner, name and namespace. The context of reference defines the namespace so that references mention only the owner and name as explicit references. For example, a *package* is in the namespace *1*, and a *package body* is in the namespace *2*[7]. The *create package* statement

[3] A nonschema object, just as the name implies, is not owned by a schema and is potentially visible to all users, identified by just its name. The fact that DBA_Objects shows the owner of an edition or directory to be SYS is an artefact of the implementation and has no practical significance.

[4] Some background sessions, most notably *MMON*, also always use exactly one edition.

[5] The attempt causes ORA-38814: Alter session set edition must be first statement of transaction.

[6] All the PL/SQL object types are potentially listed in the *DBA_PLSQL_Object_Settings* catalog view family. This includes *library*.

[7] The *DBA_Objects* catalog view family gained the column *Namespace* to advertise this property that, hitherto, had been somewhat obscure.

establishes the namespace as *1*; the *create package body* statement establishes the namespace as *2*; and the invocation of *DBMS_Output.Put_Line()* in a PL/SQL unit establishes that the identifier *DBMS_Output* is in namespace *1*.

An editioned object is uniquely identified by its owner, name, namespace *and* the value of current edition that issued the SQL statement that created or changed it[8]. This fact is the *sine qua non* of EBR; it lets two or several occurrences of the "same" object, as identified by owner, name, namespace, exist in the same database. The *DBA_Objects* catalog view family has a new column, *Edition_Name*. It is always *null* for an object that is not editioned; for an editioned object, it is always *not null* and shows the name of the edition where the object was created or changed.

Actual objects, Inherited objects, and Name resolution

There is no edition-extended syntax. When an editioned object is to be identified, the name of the edition is always supplied implicitly by the context of the reference. For a DDL statement, the current edition provides the value; and for a reference from the source code of an editioned object, the referring object's edition provides the value. Therefore, the source code of an object that is not editioned may not refer to an editioned object; such an attempt will cause a compilation error. As a corollary, an attempt to editions-enable a user will sometimes fail.

When the source code of an editioned object refers to another editioned object, then this reference is resolved to that occurrence where the *Edition_Name* is that of the edition which is the closest ancestor to the one denoted by the *Edition_Name* of the referring object. When the *Edition_Name* of the referenced object is the same as that of the referring object, then the referenced object is said to be ***actual object*** from the point of view of the referring object; and when the *Edition_Name* of the referenced object denotes an ancestor to that of the referring object, then the referenced object is said to be an ***inherited object*** from the point of view of the referring object.

A DDL statement that changes an existing inherited editioned object (for example *create or replace* or *alter*) causes that object to become actual in the current edition of the session that issued the DDL; in other words, it *actualizes* a new occurrence of the target object. This means that the changes are not seen in ancestor editions. If an editioned object is the target of a DDL statement in a particular edition (including *drop*), if that edition has descendants, and if the object in question is not actual in any of these descendants, then the effect of the change is visible in the descendants. If the object in question *is* actual in one of these descendants, then the change is visible in the intervening descendants up to, but not including, the descendant where it is actual.

Retiring an edition

When an EBR exercise is complete, it is useful to ensure that no new sessions will use the pre-upgrade edition. This is simply achieved by revoking the *USE* privilege on the to-be-retired edition from every user and role in the database. Notice that *SYS*, being beyond the normal notions of privilege, can still use the retired edition. Advantage can be taken of this to drop objects that are actual in such retired editions and that are not visible in any non-retired edition because they are actual in a descendant of the retired edition.

Dropping an edition

It is useful to drop the new child edition that was used for an EBR exercise should the exercise for some reason fail, or should the result be deemed unsatisfactory. For this use case, use the *drop edition… cascade* command to drop all objects that are actual in the to-be-dropped edition.

While it is never necessary to drop the *root edition*, this may be done when the conditions given below are met. The current *root edition* may be dropped, and then the new *root edition* may be dropped, until the database has only a single edition: the *leaf edition* as existed when these successive drops of the *root edition* were started. Customers may occasionally like to do this in pursuit of hygiene. But doing this has no practical benefit except to remove unnecessary clutter.

The *drop edition… cascade* command, just like the *drop user… cascade* command, is not atomic. This means that if the instance is shut down while the command is in progress, some of the edition's actual objects will have been dropped but others, and the edition itself, will remain. However, unlike the case if the instance is shut down while a *drop user… cascade* command is in progress (where connecting as the to-be-dropped user is still safe), it is *not* now safe to use the to-be-dropped edition. For this reason, such an edition is marked unusable. This status is reflected in the *Usable* column in the

[8] As will be seen, "change" includes not only the effect of the *create or replace* or *alter* statements but also statements like *grant* and *revoke*.

DBA_Editions catalog view family. If a session attempts to make an unusable edition its current edition, either with the *alter session* command or at connect time, then an error occurs.

An edition can be dropped only when the following conditions are met:

- The edition is not the only one in the database
  - and either it has no child edition (i.e. is the leaf edition)
  - or both it has no parent edition (i.e. is the root edition) and it has non-editioned objects that are inherited by its child edition.
- No session is using the edition.
- The edition is not the database default edition.

Notice that the *MMON* background process, just as the foreground processes, always use an edition. This is because, unlike other "primitive" background processes like *SMON* or *PMON*, it issues SQL. Some other background processes also issue SQL.

The EBR lifecycle

Most EBR exercises will follow this simple pattern:

- Before starting, the database will have only one non-retired edition, say *Pre_Upgrade*.
- During the EBR exercise, the database will have two non-retired editions, *Pre_Upgrade* and its child, say *Post_Upgrade*.
- When no sessions any longer need to use *Pre_Upgrade*, then this will be retired and the starting state for the next exercise will be restored: the database has only one non-retired edition.

As long as the *Pre_Upgrade* edition is still available for ordinary use, then *Post_Upgrade* can be simply dropped[9]. This might be done if it were realized that the upgrade install script is irrevocably incompatible with some customizations that have to be made at the particular deployed site.

It might seem that, in principle, name resolution in the many-edition regime would be appreciably slower than in the single-edition regime because most lookups would involve a recursive search backwards in the edition ancestor chain. However, the implementation, which faithfully preserves the conceptual model, transparently uses a denormalization to avoid the recursive search. *Moreover, name resolution takes place at compile time and not at run-time*[10]. It turns out, therefore, that there is no noticeable difference between using a database where the only non-retired edition is that database's only edition and using one where the only non-retired edition has an ancestor chain of, say, several hundred retired editions.

Diagramatically illustrated example

Figure 2. shows the kind of situation that might exist after a few distinct EBR exercises have been undertaken.

- The starting point is that the database has exactly one edition, e1. The procedures p1 and p2 and the views v1 and v2 are editioned objects and are actual, as they must be, in e1. The table t1, because it is not an editioned object, is drawn outside of the containing box that represents e1.
- Then e2 is created as the child of e1.
- Then a session that uses e2 does create or replace on p2 and v1, causing them to be actualized in e2. A session using e2 sees p2 and v1 as actual and p1 and v2 as inherited. The sessions using respectively e1 and e2 see the same occurrence of p1 and v2 and each edition sees its own distinct occurrence of p2 and v1, each with its own defining source code. Each edition sees the same t1 because there can never be more than one occurrence of an object that is not editioned. When no sessions any longer need to use e1, it is retired.
- Then e3 is created as the child of e2. The session that uses e3 does create or replace on p1 and v2, causing them to be actualized in e3; and it drops v1. A session using e3 sees p1 and v2 as actual and p2 as inherited. Of course, it cannot see the dropped v1; and it sees the one-and-only occurrence of t1. Though v1 is dropped in e3, it is still visible in e1 and in e2. When no sessions any longer need to use e2, it is retired.

[9] As soon as the post-upgrade application is used to record end-user transactions that cannot be represented by the pre-upgrade application, then the possibility for a simple return to the pre-upgrade application vanishes. This is determined by ordinary logic and not by any restrictions imposed by EBR.

[10] The compilation of a stored PL/SQL unit is very visible, because it requires a separate step. The compilation of a SQL statement, often referred to as *parsing*, is less visible to users because interfaces like PL/SQL's embedded SQL disguise the distinction between SQL compilation and SQL execution; nevertheless, the distinction is clear—and the famous so-called soft-parse skips the SQL compilation and goes straight to the execution.

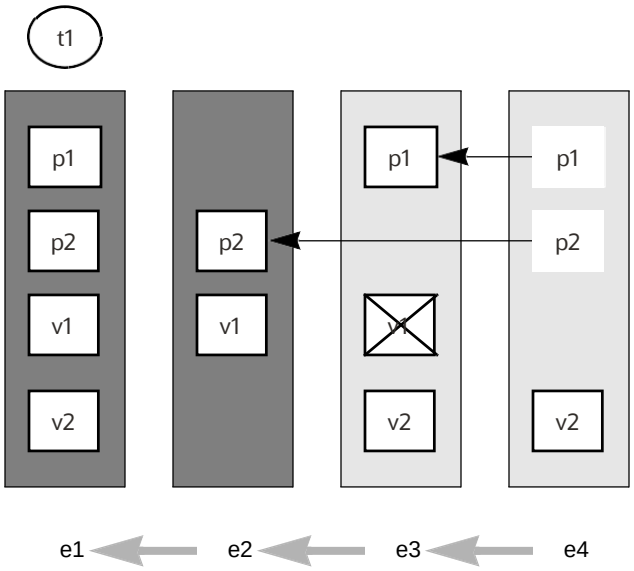

Figure 2. The situation after three EBR exercises. Actual editioned objects are shown as squares with a solid border; inherited editioned objects are shown as squares with no border; objects that are not editioned are shown as circles with a solid border; active editions are shown with a light gray fill; and retired editions are shown with a dark gray fill.

- Then e4 is created as the child of e3. The session that uses e4 does create or replace on v2, causing it to be actualized in e4. A session using e4 sees v2 as actual and p1 and p2 as inherited. Of course, it too, like e3, cannot see the dropped v1; and it, too, sees the one-and-only occurrence of t1. When no sessions any longer need to use e3, it is retired.
- *e5* is created as the child of *e4*.
- Then a session that uses *e4* does *create or replace* on *v2*; this change is denoted by the asterisk in Figure 3. A session using *e5* sees the same modified *v2* because it sees *v2* as inherited.
- Then a session that uses *e5* creates package *v1*. Because, just before it does this, *e5* sees no object called *v1*, there is no reason why this name cannot now be used for an editioned object of a different type from that which the name denotes in *e1* and *e2*. Notice that had an attempt been made to create an object called *v1* that was not editioned (for example a table called *v1*), then this would have failed because of name collisions in *e1* and *e2*.

Figure 3. illustrates the situation that might exist after the next EBR exercise.

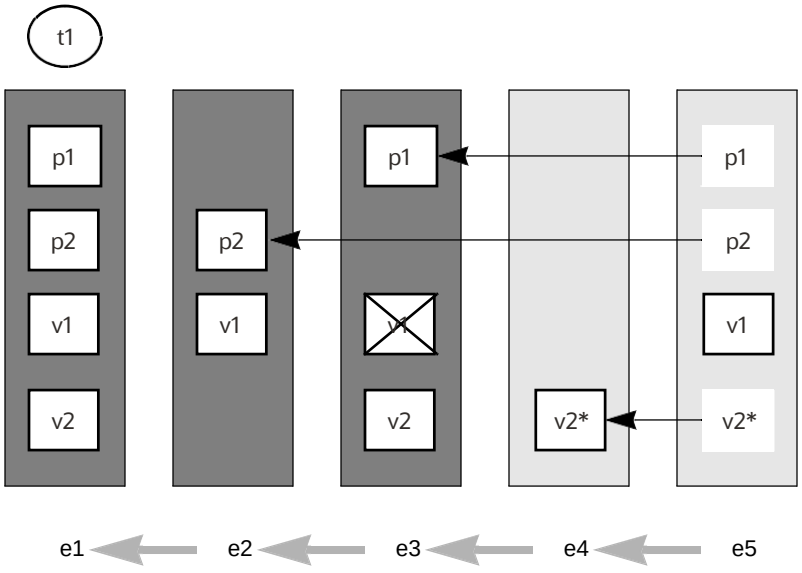

Figure 3. The situation after four EBR exercises

A minimal, complete EBR exercise code example

The starting point is a database that has exactly one edition, *Pre_Upgrade*. The application architect has worked out that objects whose type is editionable and that are owned by *App_Owner* should be editioned. Therefore, the DBA has executed the SQL statement shown in Code_2.

```
-- Code_2
alter user App_Owner enable editions
```

App_Owner connects and inevitably uses edition *Pre_Upgrade*. The query shown in Code_3 is then executed.

```
-- Code_3
select  Text
from    User_Source
where   Name = 'HELLO' and Type = 'PROCEDURE' order by  Line
```

The output is as shown in Code_4.

```
-- Code_4 procedure Hello is begin
DBMS_Output.Put_Line('Hello from Pre_Upgrade'); end Hello;
```

Of course, when *Hello* is executed, it shows "*Hello from Pre_Upgrade*".

In preparation for the EBR exercise, a user who has the *Create Any Edition* system privilege creates *Post_Upgrade*, and allows *App_Owner* to use it, using the SQL*Plus script shown in Code_5.

```
-- Code_5
create edition Post_Upgrade as child of Pre_Upgrade
/
grant use on edition Post_Upgrade to App_Owner
/
```

App_Owner is now able to execute the SQL statement shown in Code_6.

```
-- Code_6
alter session set Edition = Post_Upgrade
```

If *Hello* is executed, it still shows "*Hello from Pre_Upgrade*". Now *App_Owner* executes exactly the same DDL statement that would have been used to modify *Hello* in versions of Oracle Database as shown in Code_7.

```
-- Code_7
create or replace procedure Hello is begin
DBMS_Output.Put_Line('Hello from Post_Upgrade'); end Hello;
```

App_Owner now executes the SQL*Plus script shown in Code_8.

```
-- Code_8
begin Hello(); end;
/
select  Text
from    User_Source
where   Name = 'HELLO' and Type = 'PROCEDURE' order by  Line
/
alter session set edition = Pre_Upgrade
/
select Sys_Context('Userenv', 'Current_Edition_Name') from  Dual
/
-- Notice that the spelling that follows is identical
```

```
-- to that used before the current edition was changed
begin Hello(); end;
/
select  Text
from    User_Source
where   Name = 'HELLO' and Type = 'PROCEDURE' order by  Line
/
```

While *App_Owner* is using *Post_Upgrade*, the output of *Hello* is "*Hello from Post_Upgrade*" and the code shown in *User_Source* is that of the new, modified occurrence; and while *App_Owner* is using *Pre_Upgrade*, the output of *Hello* is "*Hello from Pre_Upgrade*" and the code shown in *User_Source* is that of the old, original occurrence.

When all are satisfied that the application as represented in *Post_Upgrade* is an improvement on the one represented in *Pre_Upgrade*, and no sessions any longer are using *Pre_Upgrade*, then a suitably privileged user will retire the *Pre_Upgrade* edition.

In this trivial example, *Pre_Upgrade* now has no actual editioned objects that are inherited by its child (and has no parent); there is no reason, therefore, not to drop it. However, in the general case, it is very likely that *Pre_Upgrade* would have editioned objects that are inherited by *Post_Upgrade* and it would not be cost-beneficial to actualize all of these in *Post_Upgrade*. Therefore, in the general case, *Pre_Upgrade* would be retired but not dropped.

If, for some reason, it is decided to abandon the changes made in *Post_Upgrade*, then a user who has the *Drop Any Edition* system privileges ensures that no session is using *Post_Upgrade* and then executes the SQL*Plus script shown in Code_9.

```
-- Code_9
drop edition Post_Upgrade cascade
/
```

Consequential actualization of dependants and fine-grained dependency tracking

When an editioned object refers to, and therefore depends upon, another editioned object, then, of course, the referenced editioned object[11] must be visible in the edition where the dependant is actual. The referenced object might be actual in the same edition as the dependant or might be actual in an ancestor edition to the dependant's and therefore seen as inherited in the dependant's edition. This rule implies that when a referenced object is first actualized in a particular edition, then all its direct and recursive dependants, that are not yet actual in that edition, will be consequentially actualized in that same edition[12].

Oracle Database has a fine-grained dependency tracking model. In earlier releases, *any* change to a referenced object caused all objects that depended on it to become invalid. This was because only coarse-grained dependency information (object *p* depends on object *q*) was recorded. The fine-grained model records dependency information at the level of the *element* within the referenced object. For example:

- If procedure *p* depends only on procedure *x* in the package *Pkg* and if *Pkg* also exposes other subprograms, variables, type declarations, and so on, then the dependency information records that p depends on *Pkg.x*.
- If view *v* mentions only columns *c1*, *c2* and *c3* in table *t*, then the dependency information records exactly this.

This means that when a referenced object is changed without changing the elements that an object that depends on it refer to, then the dependant remains valid.

This understanding needs to be extended when the referenced object, and therefore the dependant too, are editioned. If the dependant is already actual in the same edition as the referenced object (after this has suffered the DDL), or in a descendant of that edition, then the full benefit of fine-grained dependency tracking is available and invalidation that is not logically

[11] The term *referenced object* reflects the names of the columns in the *DBA_Dependencies* catalog view family: *Referenced_Owner*, *Referenced_Name*, and so on.

[12] The rule is a consequence of logic: an object cannot depend on another that it cannot see.

required is avoided. However, if on completion of the DDL to the referenced object, it is now in a younger edition than the dependant, then the dependant is actualized into the referenced object's edition in an invalid state[13].

Deliberate invalidation and revalidation of editioned objects

In an ordinarily installed Oracle Database, any user can invoke *DBMS_Utility.Validate()* or *DBMS_Utility.Compile_Schema()*[14] but only the owner, *SYS*, can invoke the *Utl_Recomp* APIs.

DBMS_Utility.Validate() has two overloads. One takes *Object_ID* and the other takes *Owner*, *ObjName*, *Namespace*, and *Edition*. (*Edition* is defaulted to the current edition.) If the target object is not actual in the current edition, then it is not actualized into this but remains actual in the edition where it was found. Notice that this is different from how *alter... compile* behaves; here, the target object is actualized into the current edition.

DBMS_Utility.Compile_Schema() and the *Utl_Recomp* APIs can be understood as wrappers that apply *DBMS_Utility.Validate()* to all the invalid objects in all editions in the specified schema or database-wide. As a consequence, using these APIs never causes actualization.

It is likely that an EBR exercise will make changes to editioned objects where at least some of these will have dependent objects. This will cause the dependent objects to be actualized into the new edition in an invalid state. It would be sensible to revalidate such objects as soon as all the intended DDLs have completed in the new edition and before proceeding to the next steps. *Utl_Recomp.Recomp_Parallel()* is the natural choice. There are no privilege concerns; implicit validation of invalid objects in the closure of dependency parents of an invalid object that is referenced for compilation or execution will anyway take place with no special privileges.

DBMS_Utility.Invalidate() has only one overload; this identifies the target object using *Object_ID*. Its only use in an EBR exercise would be to enable the values of the PL/SQL compilation parameters for a large number of units to be changed in the new edition with optimal efficiency. For example, an upgrade script might intend to compile each of the application's PL/SQL objects *native*. This is done efficiently by first invoking *DBMS_Utility.Invalidate()* for each object, using an appropriate actual for *p_plsql_object_settings*, and then invoking *Utl_Recomp.Recomp_Parallel()*. Oracle recommends against invoking *DBMS_Utility.Invalidate()* on an object that is not actual in the current edition.

The effect of DDL in an edition with a child

Suppose that a database has exactly N editions, e_1 through e_N, where e_2 is the child of e_1 and so on. Let $x[e_1]$ denote an editioned object x that is actual in e_1 and that has no dependencies on any editioned objects. As long as no DDL has been done on x, while using edition e_2 or one of its descendants, then $x[e_1]$ will be visible in e_2 and its descendants because no actual occurrence of x exists in these editions. Notice that if $x[e_1]$ does have a dependency on an editioned object, y, then it will be actualized as $x[e_M]$ in edition e_M should y be actualized there as $y[e_M]$.

In other words, when an editioned object suffers DDL using a particular edition, then the change is visible in all descendent editions up to, but not including, the closest descendent edition where another actual occurrence exists. (This actual occurrence might have *Object_Type* = *non-existent* if a DDL had been issued in the descendent edition to drop the object in question[15].) It can be seen, therefore, that in general, the effect of DDL in any descendent editions it might have, depends on specific circumstances and history: it might well happen that the effect "shines through" to all descendent editions; but this result is not guaranteed.

Using *DBMS_Sql.Parse()* to execute SQL outside of the current edition

DBMS_Sql.Parse() has some new overloads. Some support working with crossedition triggers; these will be described in "The cross-edition trigger". One new overload is provided to execute a single SQL statement in a specifically nominated edition. This allows a PL/SQL unit to execute SQL in two or more different editions and can be useful for automating DBA tasks. The remote session that supports access via a database link can use only the remote database's default edition. By using the

[13] It turns out that, because of various internal optimizations, an invalid object that is the result of consequential invalidation does not show up immediately in the DBA_Objects and DBA_Objects_AE catalog view families. However, it will show up after a call to DBMS_Utility.Compile_Schema() or to one of the Utl_Recomp APIs. It will show up, too, after an attempt to reference it (in either a compilation or an execution context).

[14] The *Execute* privilege on *DBMS_Utility* is granted to *public* and the package has a public synonym.

[15] Objects with *Object_Type* = *non-existent* can be seen in the *DBA_Objects_AE* catalog view family but not in the *DBA_Objects* catalog view family. This is a deliberate design which shows the latter view if a user uses only a single edition. The former view enables the user to understand the bigger picture and to predict what objects will be visible in any edition.

remote database's *DBMS_Sql* package, then at least single SQL statements can be executed in the chosen edition in the remote database.

Package state when the same package is instantiated in more than one edition

Suppose that the database has two editions, *Pre_Upgrade* and *Post_Upgrade* and that the editioned package

Pkg, with the source shown in Code_10[16], is actual in *Pre_Upgrade* and inherited in *Post_Upgrade*.

```
-- Code_10
package Pkg authid Current_User is State simple_integer := 0;
end Pkg;
```

The SQL*Plus script shown in Code_11 runs without error.

```
-- Code_11
alter session set Edition = Pre_Upgrade
/
begin Pkg.State := 1; end;
/
alter session set Edition = Post_Upgrade
/ begin
if Pkg.State <> 0 then Raise_Application_Error(-20000,
'Unexpected Pkg.State: '||Pkg.State); end if;
end;
/
alter session set Edition = Pre_Upgrade
/ begin
if Pkg.State <> 1 then Raise_Application_Error(-20000,
'Unexpected Pkg.State: '||Pkg.State); end if;
end;
/
```

This shows that the same editioned package is instantiated distinctly in each distinct edition from which it is referenced during the lifetime of a session and that its state for each edition's instantiation is preserved independently. It is important to understand this when a forward crossedition trigger references an editioned package that is referenced also by ordinary application code.

Notice that the opposite is the case for a noneditioned package. This has just a single instantiation. This can be seen by re-running Code_11 when the owner of *Pkg* is not editions-enabled. Now, the value of *Pkg.State* that was set in *Pre_Upgrade* is visible in *Post_Upgrade*.

The Editioning View

Only some object types are editionable. Editionable objects do not consume quota—they are represented entirely by metadata (rows in various tables in the SYS schema) and cannot contain data. It is convenient to refer to these as *code objects*. Non-Editionable objects *do* consume quota and can be refered as *data objects*. The obvious examples of data objects are *tables* and *indexes*. Tables can contain terabytes of data.

Editioned objects can have many occurrences in different editions relying on name-resolution that supplies the *Edition_Name* implicitly because, as code objects, they are small enough to allow many distinct, but similar, occurrences to exist without representing differences.

[16] The datatype *simple_integer* has a *not null* constraint.

However, the potential enormous size of data objects makes such an approach impractical. The only practical approach, then, is to let the user control the differencing explicitly. If the aim is to change a column, for example by widening it, then the original column is left in place and a new wider *replacement column* (or columns) is *added* to the table.

A further practical reason drives this design. Typical table changes during an application upgrade are incremental: the pre-upgrade and post-upgrade applications see most of the table's data in common. Therefore, during an EBR exercise, it is natural and efficient to share this common data explicitly rather than to use mechanisms to keep two separate copies of nominally the same data synchronized.

How, then, can such a table be presented to editioned code objects so that these see only the logical intention of the table at each new version and are not troubled by physical details? A view provides exactly the right mechanism; but an ordinary view is too general in its power of expression, and because of this forbids it being treated like a table with respect to some application requirements. For example, it is not allowed to create table-style triggers[17] on an ordinary view.

EBR introduces a new kind of view, the editioning view. It is created using special syntax and its defining *select* statement must satisfy strict restrictions if the creation is to succeed. An editioning view, as a special kind of view, is editionable. It might help to think that while the physical table cannot be editioned, the editioning view allows different occurrences of its logical projection to be presented in different editions. Indexes and constraints remain in the physical domain at the table level.

The conditions that an editioning view must satisfy

An editioning view's defining select statement must obey several restrictions[18]. The following list is not intended to be complete; rather, it is intended to make the spirit of the design clearer. The restrictions reflect the intention that an editioning view must simply return every row from a single table (and only those rows), without explicit ordering, and project, and maybe rename, a subset of the columns.

Because a successfully created editioning view has been confirmed to have satisfied all the restrictions, various operations on an editioning view can be supported that cannot be supported on an ordinary view. In particular, all memory of the fact that an editioning view stands in front of a table is lost during SQL compilation. The resulting execution plan is identical to the one for a query with the same meaning that targets the table(s) directly. In other words, the use of an editioning view is guaranteed to bring no performance penalty.

- **An editioning view must be owned by an editions-enabled user** as an editioning view's specific purpose is to provide an editioned API to a projection of the data that is stored in the base table.
- **An editioning view must be owned by its table's owner because** the table for an editioning view cannot be in a different database denoted by a database link.
- **There can be no more than one visible editioning view for a particular table in a particular edition** as it is meaningless to have more than one logical projection of the same table data in the same edition.
- **The subquery factoring clause is not allowed** because of the other restrictions, the subquery factoring clause could anyway have no practical usability benefit.
- **The subquery must be a single query block** which means that the keywords *union [all]*, *minus*, and *intersect* are not allowed.
- **The for update clause is not allowed**. but *is* always allowed in a query targeting an editioning view.
- **The query block must identify exactly one table.** The *from list* must have just one item which must be a *table*. A self-join is not permitted[19] and the item cannot be a view or a synonym.
- **The select list must mention only column names and optional aliases.** No column can be mentioned more than once. No kind of expression is allowed in the *select list*. For example, columns cannot be arithmetically combined; SQL and PL/SQL functions are prohibited.
- **The where clause, group by clause, and having clause are not allowed.** This is consistent with the basic intention to provide a logical cover for a physical table. Application upgrades typically change the structure of tables and apply corrections, for every row, to values in particular columns. It is rare that they need to add or remove rows in

[17] A table-style trigger is one whose timing point is *before statement*, *before each row*, *after each row*, or *after statement*. An ordinary view allows only *instead of* triggers.

[18] An attempted *create editioning view* statement that fails to satisfy the restrictions will cause an error and the view will not be created. The error message may seem obscure. For example, inclusion of a *where clause* causes *ORA-00933: SQL command not properly ended*; and inclusion of the *distinct* keyword causes *ORA-00936: missing expression*. If the statement succeeds without the *editioning* keyword but fails with it, then the reason is that the defining statement does not respect the restrictions. This suggests an approach to debugging a failed *create editioning view* statement: try it again without the *editioning* keyword.

[19] ANSI join syntax is therefore disallowed.

a table. For such scenarios, different occurrences of the editioning view must denote different physical tables in different editions[20].

- **The order by clause is not allowed.** This, too, is consistent with the basic intention. In particular, without this restriction the requirement could not be met that the execution plan for a query that targets an editioning view must be identical to the one for a query with the same meaning that targets the table directly.
- **Other restrictions** are that *distinct*, *unique*, and *all* keywords are not allowed before the *select list*. The *hierarchical query clause* and the *model* clause are not allowed. The *flashback query* clause is not allowed.

Allowed freedoms when defining an editioning view

The following semantics are allowed in addition to the basic rule that an editioning view merely projects a single table, maps the names of its columns, and does no restriction.

- **The with read only clause is allowed.** Sometimes the amount of data in a table that needs to be changed in an application upgrade is small. This is typically the case for lists of values and for data that configures the behavior of the application. Moreover, such data is normally not modifiable by ordinary end-user actions but, rather, is changed only by an administrator. In such cases, a very straightforward approach to online application upgrade is possible. A new table is defined and populated ordinarily and is then exposed using an editioning view with the same name and logical meaning in the new edition as the one that exposed the old table into the old edition. By setting these editioning views with read only, the intention that the table content is not changed by end-users is formally enforced. Of course, the alter view command can be used to make an editioning view either read-only or read/write. Notice that there is no special alter editioning view syntax.
- **Primary key constraints are allowed but foreign key constraints are disallowed.** Primary key and foreign key constraints can be created on an ordinary view, but the keywords *disable novalidate* must be used. The benefit is mainly that tools can generate diagrammatic representations of the logical database design. However, an editioning view must be editioned and an editioned object cannot be the source or the target of a foreign key constraint. Therefore, an editioning view cannot be the source or the target of a foreign key constraint. An editioning view can have a disable novalidate primary key constraint.

Operations supported by an editioning view that are not supported by an ordinary view

The fact that the following operations are allowed on an editioning view reflects the intention that, once an editioning view is in place in front of every table, then the rest of the application design and implementation can treat these editioning views as if they were tables and will never, therefore, need to refer to a table explicitly.

The following are examples. However, rather than listing every single property that distinguishes an editioning view from an ordinary view, it is more useful to state the overall principle:

DML Operations support

Any *select*, *insert*, *update*, *delete*, *merge*, *lock table* or *explain plan* SQL statement[21] that will run without error on a table will run without error on an editioning view that covers that table.

An editioning view allows table-style triggers

Triggers that are defined on renamed tables become invalid because they are still attached to those tables[22]. However, to honor the principle that application code should not refer explicitly to tables, the triggers should be recreated on the editioning view that now has the table's former name. This is trivially achieved by dropping the triggers and then re-running the DDL that created them[23].

---

[20] It hardly needs pointing out that rows come and go, and are changed, as part of the routine operation of every application. The capability to do this comfortably in a multiuser environment is well-established. It would be appropriate to use EBR to stage the visibility of such ordinary changes only when the content of the tables in some way defines the behavior and meaning of the application, and, of course, especially when both the context and the structure of such configuration tables needs to be changed.

[21] For example, *select Rowid, ev.* from ev* is legal when *ev* is an editioning view.

[22] When a table is renamed, the opening part of the source text of a trigger on the table is automatically updated to reflect the new name. The same happens when columns are renamed and the they are mentioned in the *when clause*. However, the source text of the PL/SQL that implements the trigger action is not updated. This will leave the trigger in an invalid state when the text refers to other tables that have been renamed.

[23] The DDL will run without error because the new editioning view exposes exactly the same identifiers as the table it covers. This holds also for compound triggers that may been defined on the renamed table.

Notice that when DML is done using an editioning view, then not only will triggers defined on the editioning view fire, but also ones defined on its base table will fire. However, when DML is done using a table, then only the triggers defined on the table will fire—and triggers defined on the editioning view will not fire. The paradigm requires that all regular application DML be done using editioning views; as shall be seen (see *"The crossedition trigger"*) only crossedition triggers are allowed to do DML using tables.

Hint in a SQL statement targeting editioning view can identify index by listing the names of its columns

This, again, allows extant application code to remain correct after the introduction of an editioning view to cover a table.

Queries against an editioning view allow partition extended syntax

When an editioning view's base table is partitioned, then the same query extended syntax that can be used against the table can be used against the editioning view. The SQL*Plus script shown in Code_12 illustrates this.

```
-- Code_12
create table t(PK integer primary key, Info varchar2(10)) partition by range(PK)
(partition p1 values less than (10), partition p2 values less than (maxvalue) )
/ begin
insert into t(PK, Info) values ( 5, 'in p1'); insert into t(PK, Info) values (15, 'in p2'); commit;
end;
/
create view v as select a.PK, a.Info from t a
/
-- Causes ORA-14109
select * from v partition(p1)
/
create editioning view ev as select a.PK, a.Info from t a
/
-- Runs without error
select * from ev partition(p1)
/
```

EBR using only editions and editioning views

If an application upgrade will change only those tables whose data is not changed via the ordinary end-user interfaces, then the edition together with the editioning view are sufficient to allow these changes to be made while the application remains online. The most obvious example is configuration data—data that determines the behavior of the application and that is changed only as part of an upgrade. Such data is typically not voluminous and so it would be natural to create a replacement table for the upgrade so that an editioning view with a particular owner and name selects from one table in the pre-upgrade edition and from a different table in the post-upgrade edition. The upgrade installation script can simply populate the replacement table as required. According to the requirements of the upgrade, the editioning view that covers the post-upgrade table may, or may not, have the same shape as the editioning view that covers the pre-upgrade table.

The Crossedition Trigger

Sometimes, an application upgrade has to change one or more tables whose content is queried and changed by ordinary end-user interaction. Consider a use case for example: a single column that represents a telephone number as it would be used when dialling within the USA is to be split into two columns, one for the country code and one for the within-country number. A bulk transformation of the data is not, by itself, sufficient to ensure correctness of the transformed data. A mechanism is needed to keep pace with changes that end-users of the pre-upgrade application make to the old

representation of the data, transforming it into the new representation, both during the bulk transformation and after it is complete as some users continue to use the pre-upgrade application while others start to use the post-upgrade application.

Moreover, changes that end-users of the post-upgrade application make to the new representation of the data must be transformed back into the old representation for the benefit of end-users of the pre-upgrade application.

Triggers have exactly the right properties to affect the proper responses to the changes that end-users make during the bulk forward transformation of data and during the hot rollover period. Moreover, the use of a trigger for this purpose meets the high level requirement that application code *itself* can be written to implement only what is needed for its ordinary pre- and post-upgrade operation and need not implement special logic to accommodate the period when an EBR exercise is in progress. Special triggers, understood to be distinct from the application code, can be deployed during the EBR exercise and dropped when it is complete.

A cross-edition trigger is a special kind of trigger; and a trigger is an editionable object type. However, unlike other objects whose type is editionable, a cross-edition trigger *must* be owned by an editions-enabled user; in other words, a cross-edition trigger is always editioned[24]. The reason for this restriction is that the firing rules for a crossedition trigger are defined with respect to the relationship between the edition in which it is actual and the current edition of the session that issues the DML. Further, a crossedition trigger is visible only in the edition in which it is actual. As a consequence, the SQL*Plus script shown in Code_13 runs without error.

```
-- Code_13
alter session set edition = e2
/
create trigger x
before insert or update or delete on t for each row
forward crossedition disable
begin
...
end x;
/
-- e3 is the child of e2
alter session set edition = e3
/
-- Notice that we don't need "or replace"
create trigger x
before insert or update or delete on t for each row
forward crossedition disable
begin
...
end x;
/
```

It is unimportant with respect to the firing rules that a crossedition trigger is visible only in the edition in which it is actual because these rules are explicitly defined; but this has the consequence that dependencies between crossedition triggers (by virtue of *follows* or *precedes* relationships) can exist only between sets of crossedition triggers that are actual in the same edition[25]. If the clause is *follows*, then the target must be a forward crossedition trigger; and if the clause *precedes*, then the target must be a reverse crossedition trigger.

[24] If a user that is not editions-enabled attempts to create a crossedition trigger, this causes *ORA-25030*.

[25] This restriction ensures that no contradictions about firing order can be expressed. As will be seen, the firing order of crossedition triggers in a particular edition cannot be interleaved with that of crossedition triggers in a different edition.

The compilation of a crossedition trigger follows the normal rules for the compilation of any editioned object: names are resolved to objects that are visible in the edition in which it is actual. But in contrast to other editioned objects, a crossedition trigger and all code it calls always *runs* using the edition in which it is actual. Code_20 is a SQL*Plus script that shows this.

A crossedition trigger may be created only directly on a table—and not on either a regular view or an editioning view[26]. This implies that only the *before statement*, *before each row*, *after each row*, and *after statement* variants may be specified; the *instead of* variant is not legal for a crossedition trigger. A crossedition trigger may be a compound trigger.

Basic firing rules for crossedition triggers

The firing rules were designed on the assumption that the crossedition triggers required to implement a particular upgrade are all installed in the post-upgrade edition. This is consistent with the overall paradigm that (in order that the pre-upgrade application will be unperturbed) *all* DDL to editioned objects is done in the

post-upgrade edition. The rules assume that pre-upgrade columns are changed (by ordinary application code) only by sessions using the pre-upgrade edition and that post-upgrade columns are changed (again by ordinary application code) only by sessions using the post-upgrade edition. There are therefore two kinds of crossedition trigger:

- A *forward crossedition trigger* is fired by application DML issued by sessions using the pre-upgrade edition. Such a trigger is used to implement transformations from the old representation forwards into the new representation.
- A *reverse crossedition trigger* is fired by application DML issued by sessions using the post-upgrade edition. Such a trigger is used to implement transformations from the new representation backwards into the old representation.

The following is a more careful statement of the rules, acknowledging the fact that three or more editions might be active during an EBR exercise:

- A forward crossedition trigger is fired by application DML issued by a session using any ancestor edition to that in which the trigger is actual.
- A reverse crossedition trigger is fired by application DML issued by a session using the edition in which the trigger is actual or any descendant of that edition.

The following demonstration illustrates these basic firing rules for crossedition triggers. The database has five editions, *e1*, *e2* (child of *e1*), and so on through to *e5* (child of *e4*).

The procedure *Trace*, shown in Code_14, is owned by SYS and is therefore not editioned.

```
-- Code_14
procedure Trace(
t1 in varchar2, t2 in varchar2 := null) authid Definer
is
f Utl_File.File_Type := Utl_File.Fopen( Location => 'MY_DIR',
Filename => 't.txt', Open_Mode => 'a', Max_Linesize => 32767);
begin
if t2 is null then Utl_File.Put_Line(f, t1);
else
Utl_File.Put_Line(f, Rpad(t1, 30, '.')||' '||t2); end if;
Utl_File.Fclose(f); end Trace;
```

There is a public synonym for *Sys.Trace*, and *Execute* on *Sys.Trace* is granted to public.

The user *Usr* is editions-enabled and is granted only *Create Session*, *Resource*, and *Use* on each of *e1*

through *e5*.

The function *Usr.Curr_Edn*, shown in Code_15, is actual in edition *e1*.

```
-- Code_15
```

[26] The attempt causes ORA-42306: a crossedition trigger may not be created on an editioning view.

```
function Curr_Edn return varchar2 authid Definer is e constant varchar2(30) not null :=
Sys_Context('Userenv', 'Current_Edition_Name'); begin
return e; end Curr_Edn;
```

The table *Usr.t* has a column *n* of datatype *number*; the editioning view *Usr.ev* covers it and selects *n*. The regular trigger *Usr.Regular*, shown Code_16, is actual in edition *e2*.

```
-- Code_16
trigger Regular after update on ev
begin
Trace('From Regular', Curr_Edn()); end Regular;
```

The forward crossedition trigger *Usr.Fwd_Xed*, shown in Code_17, is actual in edition *e3*.

```
-- Code_17
trigger Fwd_Xed after update on t
forward crossedition begin
Trace('From Fwd_Xed. Expect E3', Curr_Edn()); end Fwd_Xed;
```

The reverse crossedition trigger *Usr.Rev_Xed*, shown in Code_19, is actual in edition *e4*.

```
-- Code_18
trigger Rev_Xed after update on t
reverse crossedition begin
Trace('From Rev_Xed. Expect E4', Curr_Edn()); end Rev_Xed;
```

Finally, the procedure *Usr.Do_Update*, shown in Code_19, is actual in edition *e1*.

```
-- Code_19
Do_Update authid Definer is begin
Trace('From Do_Update', Curr_Edn()); update ev set n = n + 1;
commit;
end Do_Update;
```

The SQL*Plus script shown in Code_20

```
-- Code_20
alter session set edition = e1
/ begin
Trace(Chr(10)||'App using e1'); Do_Update();
end;
/
alter session set edition = e2
/ begin
Trace(Chr(10)||'App using e2'); Do_Update();
end;
/
alter session set edition = e3
/ begin
Trace(Chr(10)||'App using e3'); Do_Update();
end;
/
```

```
alter session set edition = e4
/ begin
Trace(Chr(10)||'App using e4'); Do_Update();
end;
/
alter session set edition = e5
/ begin
Trace(Chr(10)||'App using e5'); Do_Update();
end;
/
```

will then produce this output to the trace file *t.txt*:

```
-- Code_21
App using e1
From Do_Update.E1
From Fwd_Xed. Expect E3.      E3
App using e2
From Do_Update.E2
From Regular.  E2
From Fwd_Xed. Expect E3.      E3
App using e3
From Do_Update.E3
From Regular.  E3
App using e4
From Do_Update.E4
From Regular.  E4
From Rev_Xed. Expect E4.      E4
App using e5
From Do_Update.E5
From Regular.  E5
From Rev_Xed. Expect E4.      E4
```

When a database has no more than two active editions during an EBR exercise and when no crossedition trigger issues DML, then it is sufficient just to understand these basic firing rules.

Advanced firing rules for crossedition triggers

We will use the term *crossedition trigger DML* for DML issued directly, using embedded SQL or native dynamic SQL, from the PL/SQL unit that is a crossedition trigger, and we will use the term *regular DML* for DML issued from *any other site*. Notice that this definition means that DML that is issued from a PL/SQL unit that is invoked by a crossedition trigger is *regular DML*. In particular, DML issued by using the *DBMS_Sql* API is by default regular DML, even when the invocation of these subprograms is made directly from the implementation of a crossedition trigger. However, if the name of the crossedition trigger that invokes the *DBMS_Sql* API is included in the actual *Applying_Crossedition_Trigger()* formal parameter to *DBMS_Sql.Parse()*, then the DML that the *DBMS_Sql* API issues will be crossedition trigger DML.

- Regular DML always fires both visible regular triggers and appropriately selected crossedition triggers.
- The firing order of crossedition triggers in a particular edition is never interleaved with that of crossedition triggers in a different edition. All forward crossedition triggers in edition *e* will fire before any in a descendent edition of edition *e*, and all reverse crossedition triggers in edition *e* will fire after any in an ancestor edition of edition *e*.

- Crossedition trigger DML from a forward crossedition trigger actual in edition *e* will fire forward crossedition triggers that are actual in descendents of edition *e* but will never fire reverse crossedition triggers or regular triggers.
- Correspondingly, crossedition trigger DML from a reverse crossedition trigger actual in edition *e* will fire reverse crossedition triggers that are actual in ancestors of edition *e* but will never fire forward crossedition triggers or regular triggers.
- Recall the fact that DML done to a table does not fire triggers on an editioning view that covers the table (see "An editioning view allows table-style triggers"). This means that, in practice, even DML to tables that a crossedition trigger issues using the *DBMS_Sql* API or a helper PL/SQL unit that in turn does the DML (which is therefore regular DML) will not fire regular triggers because these, following the paradigm, will not be created on tables but will be created only on editioning views.
- Crossedition trigger DML from a unit that is actual in edition *e* does not, unless special programming steps (described in the next two bullet points) are taken, fire crossedition triggers that are actual in edition *e*.
- If forward crossedition trigger *Fwd_Xed_1*, on table *t1*, issues crossedition trigger DML to table *t2*, then forward crossedition trigger *Fwd_Xed_2*, on table *t2*, will fire if and only if there is an ordering relationship between *Fwd_Xed_2* and *Fwd_Xed_1*. Either *Fwd_Xed_2* may be defined using the *follows Fwd_Xed_1* syntax; or the ordering relationship between *Fwd_Xed_1* and *Fwd_Xed_2* may be established transitively (through one or several intervening crossedition triggers).
- Correspondingly, if reverse crossedition trigger *Rev_Xed_1*, on table *t1*, issues crossedition trigger DML to table *t2*, then reverse crossedition trigger *Rev_Xed_2*, on table *t2*, will fire if and only if there is an ordering relationship between *Rev_Xed_2* and *Rev_Xed_1*. Again, the ordering may be direct or transitive[27].

The *apply* step: systematically visiting every row to transform the pre-upgrade representation to the post-upgrade representation

While forward crossedition triggers are necessary in order to propagate changes that happen to be made to the pre-upgrade representation by user activity, just having them in place is, of course, not sufficient to ensure that every row will be transformed. The simplest way to ensure that every row is transformed is to use a batch process to force each forward crossedition trigger to fire. This is trivially achieved by updating each forward crossedition trigger's base table to set a column that fires the trigger on update to itself. There is, however, a little more to this than you might at first think.

Using *DBMS_Sql.Parse()* to apply a forward crossedition trigger

The firing rules for crossedition triggers dictate that regular DML issued by a session using edition *e* will not fire forward crossedition triggers that are actual in edition *e*. But the paradigm for EBR requires that a session that is installing the upgrade should use the post-upgrade edition. How, then, can such a session make a relevant forward crossedition trigger fire?

DBMS_Sql.Parse() has overloads with the formal parameter *Apply_Crossedition_Trigger*. These overloads also have the formal parameters *Edition* and *Fire_Apply_Trigger*. *Apply_Crossedition_Trigger* has no default value, *Edition* has the default value *null*, and *Fire_Apply_Trigger* has the default value *true*. (Other overloads have just the formal parameter *Edition*; in these, it has no default value.) Code_22 shows the simple use of the overload with *Apply_Crossedition_Trigger* to fire the forward crossedition trigger *Fwd_Xed*, on table *t*, for each of its rows.

```
-- Code_22
DBMS_Sql.Parse(
c       => The_Cursor,
Language_Flag  => DBMS_Sql.Native,
Statement      => 'update t set c1 = c1', Apply_Crossedition_Trigger => 'Fwd_Xed');
```

[27] Of course, neither the use of the *precedes* clause nor the use of the *follows* must specify circularity. The attempt causes *ORA-25023: Cyclic trigger dependency is not allowed.*

When *Edition* is *null*, then names are resolved in the current edition of the session that invokes *DBMS_Sql.Parse()*. The significance of *Fire_Apply_Trigger* is explained in *"Using explicit SQL for the apply step"*. Forward crossedition triggers are the only triggers that you can *apply* (cause to fire on every row of the table on which they are defined).

Crossedition triggers must be idempotent

It is impossible to predict whether a particular row that is to be transformed by a forward crossedition trigger will be visited first by ordinary end-user activity or by the *apply* step.Therefore, it is possible that, when the *apply* step happens second, the same transform will be applied twice to the same row. The action of a forward crossedition trigger must therefore, by explicit design, be *idempotent*. (Similar rationale holds for the design of a reverse crossedition trigger—even though these are never the subject of an *apply* step.)

When a replacement table is used, then every row in the original table needs to be reflected in the replacement. If the source row is visited first by ordinary end-user activity, then when the same row is visited by the *apply* step, no further cation is needed. (This is because the current state of the source row is already reflected in the target replacement table.) The *Ignore_Row_On_Dupkey_Index* is provided to allow the rule to be simply implemented. It is, however, necessary to detect that the *apply* step is in progress if this is implemented simply by causing the forward crossedition trigger that implements the transform to fire for every row. The boolean function *Applying_Crossedition_Trigger()* in the package *DBMS_Standard* is provided for this purpose.

It is possible, of course, that when the forward crossedition trigger fires in response to ordinary end-user activity, the source row is already reflected in the target table. If this is the case, then the functional equivalent of a *merge* must be done. The *Change_Dupkey_Error_Index* hint is provided to allow this functionality to be programmed conveniently.

When to enable crossedition triggers—*DBMS_Utility.Wait_On_Pending_DML()*

In order that there be no "lost updates" during the *apply* step, the following logic must be used.

- Enable the forward crossedition triggers that are mutually related by the follows relationship.
- Invoke DBMS_Utility.Wait_On_Pending_DML(). This waits until all transactions (other than the caller's own) that have locks on the listed tables and that began prior to the invocation of this function have either committed or been rolled back.
- Start the apply step.

Using the DBMS_Parallel.Execute() API

If the table which will suffer the *apply* step has very many rows, then should the operation be done as a single transaction, ordinary users attempting to change rows in the same table would be very likely to suffer unacceptable waits. Therefore, the availability of the pre-upgrade application will be improved if the *apply* step is conducted in separately committed chunks of reasonable size. Because the transform is required to be idempotent, there is no requirement to complete the *apply* step in a single commit unit and no requirement to keep the wall clock time between the commit of the separate chunks short. The *DBMS_Parallel.Execute()* package provides a convenient way to achieve this. It exposes just the same degrees of freedom as does the *DBMS_Sql.Parse()* overload shown in Code_22 on *page 22*.

Using explicit SQL for the *apply* step

While it takes less effort on behalf of the developers making use of EBR to implement the *apply* step simply by causing the forward crossedition trigger(s) that implement the transform for each row of the table, this is not always the approach that produces the most performant result. This is especially the case when a replacement table is used. A SQL statement that has the same effect (if one can be written) will use less computational resource than the row-by-row approach (with associated per row SQL to PL/SQL to SQL context switches) that reusing the forward crossedition trigger(s) implies. Code_23 shows how, to achieve this, *DBMS_Sql.Parse()* is used with *Fire_Apply_Trigger* set to *false* to indicate that rather than firing the forward crossedition trigger designated by *Apply_Crossedition_Trigger*, the real SQL statement designated by *Statement* will be used.

```
-- Code_23
DBMS_Sql.Parse(
c       => The_Cursor,
Language_Flag  => DBMS_Sql.Native,
Statement      => The_Real_SQL_Statement, Apply_Crossedition_Trigger => 'Fwd_Xed', Fire_Apply_Trigger
       => false);
```

It is necessary to specify the name of the forward crossedition trigger, *Fwd_Xed*, that implements the same transform so that the closure of other forward crossedition triggers in *follows* relationship the *Fwd_Xed* will fire. Of course, the *DBMS_Parallel.Execute()* approach may be used for this approach to the *apply* step.

Combining several bug fixes in a single EBR exercise

Real applications are often very large and complex, may be developed and maintained by a large team, and they suffer from many independent bugs. Each bug fix might be implemented independently of others by a different developer. There are two ways to implement a set of fixes at a deployed site.

- *Either*, a single patch script is developed to make the transformation corresponding to *N* distinct bug fixes, going from the start state to the end state in an optimal fashion
- *or N* separate patch scripts are developed, each to implement the fix for one bug, and these *N* scripts are run in succession in an order that has been designed to be appropriate.

The first approach is potentially more efficient; but the second approach is likely to require less effort from the team that develops and maintains the application. Moreover, especially when the application is delivered by an ISV, different sites where the same application is deployed might need to apply different bug fixes; in such cases, the second approach offers more flexibility.When the first approach is implemented using EBR, it is very unlikely that the advanced firing rules for crossedition triggers will be useful. The exercise will use only a single new edition, and no crossedition trigger *Trg2* will implement logic to respond to a change that a different crossedition trigger *Trg1* will make. (Rather, *Trg1* will implement directly the logic that *Trg2* otherwise would have implemented.)

However, when the second approach is implemented using EBR, it might happen that one crossedition trigger *Trg2* must fire only after another crossedition trigger *Trg1* has fired because, in the ordering scheme for individual fixes, it is realized that *Trg2* (on table *t2*) must read data that *Trg1* (on table *t1*) must first have changed. In relatively rare cases, not only might *Trg1* do DML to *t2* but also *Trg2* might do DML to *t1*—in other words, a possibility of circularity might arise.

The conceptually simple way to avoid such circularity is to use a new edition for each fix, where the parent-child order of the editions reflects the designed order of applying the fixes. End-user sessions would use only the ultimate ancestor edition and the ultimate descendent edition. The fact that crossedition trigger DML from a forward crossedition trigger will fire only those forward crossedition triggers in descendent editions (and correspondingly for reverse crossedition triggers) avoids circular firing. However, it is less cumbersome to use only a single new edition; in this case, that fact that crossedition trigger DML will never fire crossedition triggers in the same edition unless this is explicitly requested with a *follows* or *precedes* mutual relationship avoids circular firing.

CONCLUSION

This whitepaper has explained how edition-based redefinition (EBR) is used to allow an application's database objects to be patched or upgraded online while the application is in use. It also draws attention to the characteristics that distinguish the capability markedly from other Oracle Database capabilities that support the other subgoals of overall high availability.

An application's database backend must be specifically prepared to use EBR. This will need a new version of the application as the vehicle. The new version will be designed by the architect and will be delivered by upgrade scripts created by the developers. The upgrade to the EBR-readied version must be done in downtime because tables will be renamed, and dependent objects will be invalidated. Only when an editioning view covers each table and restores its former name will revalidation be possible.

An existing application (before it is EBR-readied) needs to be redesigned with some non-trivial changes by the application architect if it meets any of the below conditions:

- Has unfavorable occurrences of objects that cannot be editioned that depend on objects that will be editioned
- Has occurrences of evolved ADTs owned by users that will be editions-enabled
- Has occurrences of views that are the source or target of foreign key constraints owned by users that will be editions-enabled

Once the application has been EBR-readied, then subsequent upgrades and patches may be done online. Such scripted EBR exercises, just like scripted classical offline upgrades and patches, will be designed by the application's architect and implemented by the application's developers. An administrator at the deployed site of an application cannot perform an online application upgrade unless the application's developers have delivered the upgrade scripts as an EBR exercise. Obviously, this is not an issue for a new application which can easily be built to be EBR-readied right front the onset.

Below are typical use cases in which EBR can be used to maintain application availability during an application upgrade. A one-time configuration step is needed to enable EBR for a database.

CHANGE TYPE	FEATURE
PL/SQL object changes	Editions
Table structure changes	Editions and Editioning Views
Table data changes	Editions, Editioning views and Cross-edition triggers

CONNECT WITH US

Call +1.800.ORACLE1 or visit oracle.com.
Outside North America, find your local office at oracle.com/contact.

blogs.oracle.com facebook.com/oracle twitter.com/oracle

Edition-Based Redefinition
June, 2020