



ORACLE

Edition-Based Redefinition Technical Deep Dive

An Oracle Database capability for online application upgrades

May, 2024, Version 4.0
Copyright © 2024, Oracle and/or its affiliates
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 upgrades 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 the 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. Delivering any material, code, or functionality is not a commitment and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remain 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

| | |
|---|----------|
| Purpose Statement | 2 |
| Disclaimer | 2 |
| Introduction | 5 |
| Edition-Based Redefinition | 7 |
| EBR Use Case | 7 |
| The Edition | 8 |
| Editionable object types, Editions-enabled users, and Editioned objects | 9 |
| Actual objects, Inherited objects, and Name resolution | 9 |
| Retiring an edition | 10 |
| Dropping an edition | 10 |
| The EBR lifecycle | 10 |
| A minimal, complete EBR exercise code example | 12 |
| Consequential actualization of dependants and fine-grained dependency tracking | 14 |
| Deliberate invalidation and revalidation of editioned objects | 15 |
| The effect of DDL in an edition with a child | 15 |
| Using <i>DBMS_Sql.Parse()</i> to execute SQL outside of the current edition | 16 |
| Package state when the same package is instantiated in more than one edition | 16 |
| The Editioning View | 17 |
| The conditions that an Editioning View must satisfy | 17 |
| Allowed freedoms when defining an Editioning View | 18 |
| Operations supported by an Editioning View that are not supported by an ordinary view | 19 |
| EBR using only editions and Editioning Views | 20 |
| The Cross-edtion Trigger | 20 |
| Basic firing rules for cross-edtion triggers | 22 |

| | |
|---|-----------|
| Advanced firing rules for cross-edition triggers | 25 |
| The <i>apply</i> step: systematically visiting every row to transform the pre-upgrade representation to the post-upgrade representation | 26 |
| Combining several bug fixes in a single EBR exercise | 28 |
| Conclusion | 29 |

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 upgrades with uninterrupted application availability.

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 using the pre-upgrade application until its user decides to end it; 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 a hot rollover¹ from the pre-upgrade version to the post-upgrade version.

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

To achieve an online application upgrade², the following conditions must be met:

- Installing 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 tables 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 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. Discussing 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 appropriate 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. Instead, it aims to explain the concepts and the use of EBR at a

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.

depth impractical in the Oracle Database Documentation Library. This way, it complements and extends the treatment in the Oracle Database Advanced Application Developer's Guidebook and the Oracle Database Administrator's Guidebook.

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. Suppose the application upgrade only changes views, synonyms, and PL/SQL objects. In that case, the *Edition alone* allows 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. Suppose table data or structure changes are restricted to only those not changed via the standard end-user interfaces. In that case, *the Edition and 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. Suppose changes to table data or structure are required for those tables that are changed routinely by the end user. In that case, *the Edition, the Editioning View, and the Cross-edition Trigger* must be used to make these changes while the application remains online.

EBR Use Case

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

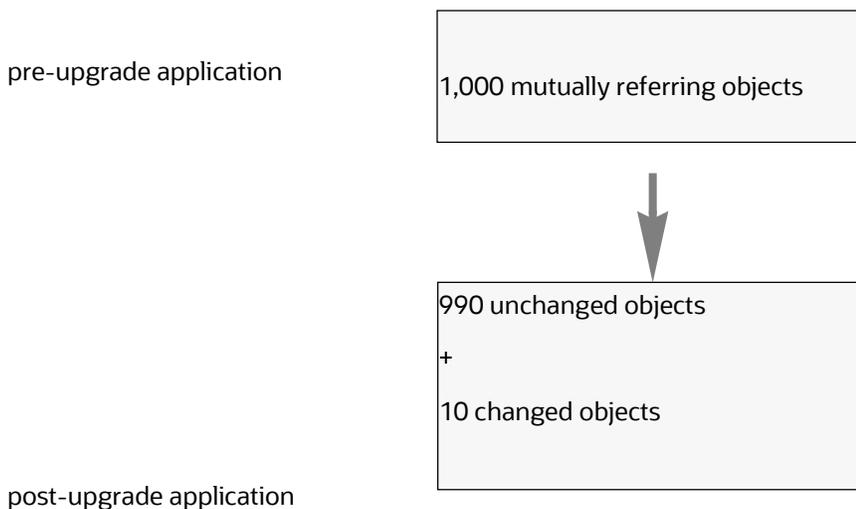


Figure 1. The challenge of changing dependent objects

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

A short digression on the viability of an approach that uses schemas and synonyms to enrich the naming mechanisms manually explicitly will be helpful. 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 an 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. However, there would be some effort in redefining the synonyms in the source control system, but this could presumably be done automatically. This approach suffers from several disadvantages concerning using EBR:

- It requires a specific design by the application developer.
- Every “primary” object must be duplicated, which costs both space and time 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. Still, it would be dauntingly complex and, because of that, subject to an appreciable risk of error.
- Some applications issue DDL statements as a normal response to ordinary end-user interaction. The effort to design and implement a scheme to reflect such changes forward and backward 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 its disadvantages.

The Edition

An *edition* is a new, non-schema object type uniquely identified by its name. Editions are listed in the *DBA_Objects* catalog view family where, just like the non schema 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-negotiable has at least one Edition. Immediately on creating or upgrading an Oracle database, there is precisely one Edition named *Ora\$Base*.

A new edition must be created as the child of an existing one; the syntax of the *create edition* statement allows the parent edition to be identified using the *as the child of a* clause. An edition may have no more than one child. The *create edition* statement allows the *as the child of a* 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 edition⁴. 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 granting use to the *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 a session uses 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 effort to change the Edition that a session uses will fail if there is any uncommitted DML⁵.

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.

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

Views (and therefore Editioning Views), synonyms, and all the kinds of PL/SQL objects type⁶ (and therefore Cross-edition Triggers) are **editionable object types**. There are no other editionable object types. For example, the *table* is not an editionable object type or *Java class*.

The non-schema 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 not of an editionable 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. Still, it will irrevocably become so when its owner is altered to become editions-enabled. An object not editioned is uniquely identified by 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 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 the current Edition that issued the SQL statement that created or changed it⁸. This fact is the *sine qua non* of EBR; it lets two or several occurrences of the “same” object, as identified by owner, name, and 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 reference context. For a DDL statement, the current Edition provides the value; 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 not editioned may not refer to an editioned object; such an attempt will cause a compilation error. As a result, an effort 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 an **actual object** from the point of view of the referring object. 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, 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

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.

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*.

descendants. Suppose the object in question is actual in one of these descendants. In that case, the difference 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, ensuring that no new sessions will use the pre-upgrade Edition is useful. This is achieved by revoking the *USE* privilege on the to-be-retired Edition from every user and role in the database. Notice that *SYS*, 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 not visible in any non-retired edition because they are actual in a descendant of the retired Edition.

Dropping an edition

It is helpful to drop the new child edition used for an EBR exercise should the exercise fail or the result be 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 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 is not atomic, just like the *drop user... cascade* command. 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 *DBA_Editions* catalog view family. An error occurs if a session attempts to make an unusable edition its current Edition, either with the *alter session* command or at connect time.

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 inherited by its child edition.
- No session is using the Edition.
- The Edition is not the database's default edition.

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

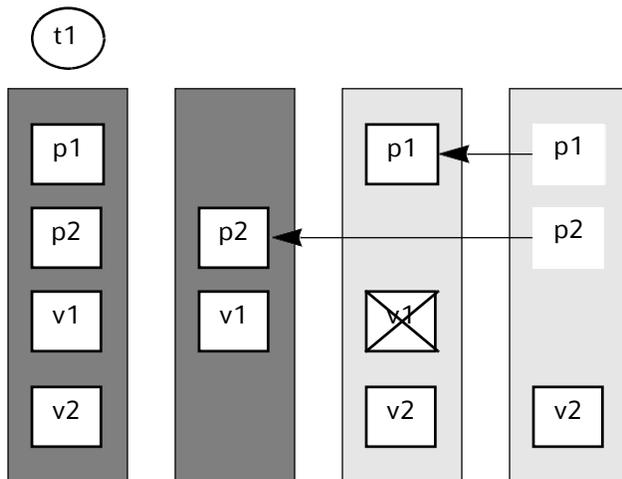
If the *Pre_Upgrade* edition is still available for ordinary use, then *Post_Upgrade* can be dropped⁹. This might be done if it were realized that the upgrade install script is irrevocably incompatible with some customizations that have to be made at the particular deployed site.

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 backward in the edition ancestor chain. However, the implementation, which faithfully preserves the conceptual model, transparently uses 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.

Diagrammatically illustrated example

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

- The starting point is that the database has precisely 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 the box representing e1.
- Then e2 is created as the child of e1.
- Then a session that uses e2 creates or replaces 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 distinct occurrence of p2 and v1, each with its defining source code.
- Then e3 is created as the child of e2. The session that uses e3 creates or replaces 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; it considers the one-and-only occurrence of t1. Though v1 is dropped in e3, it is still visible in e1 and e2. When no sessions 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 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; it also considers the one-and-only occurrence of t1. When no sessions need to use e3, it is retired.
- e5 is created as the child of e4.
- Then a session that uses e4 creates or replaces 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, so 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), 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.

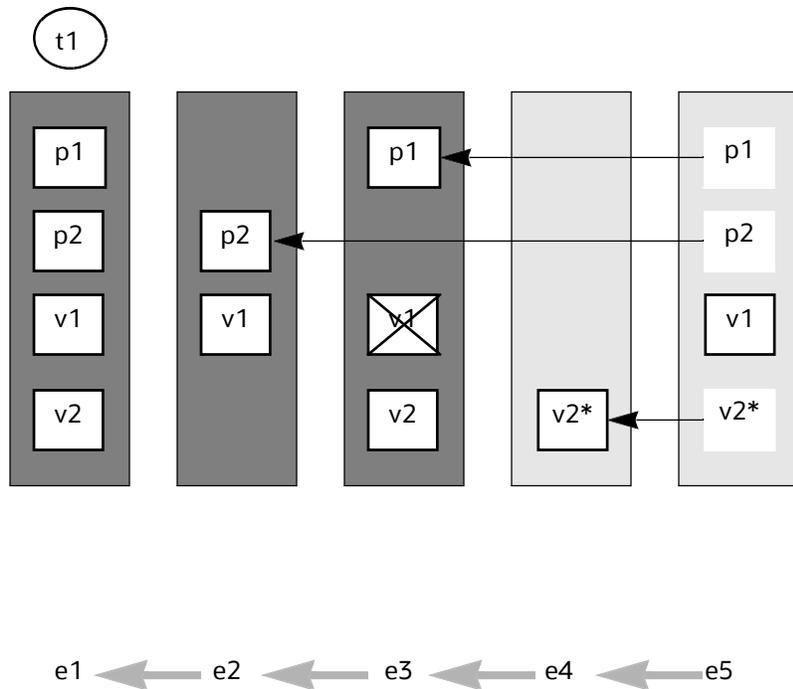


Figure 3. The situation after four EBR exercises

A minimal, complete EBR exercise code example

The starting point is a database that has exactly one Edition, *Pre_Upgrade*. The application architect has determined that objects whose type is editionable and 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 with the Create Any Edition system privilege creates *Post_Upgrade*, allowing *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 the same DDL statement that would have been used to modify *Hello* in versions of the 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 represented in *Post_Upgrade* improves the one represented in *Pre_Upgrade*, and no sessions no longer using *Pre_Upgrade*, a suitably privileged user will retire the *Pre_Upgrade* edition.

In this trivial example, *Pre_Upgrade* now has no actual editioned objects that its child inherits (and has no parent); there is no reason, therefore, not to drop it. However, in the general case, *Pre_Upgrade* would likely have editioned objects that *Post_Upgrade* inherits, 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, the referenced editioned object¹¹ must be visible in the Edition where the dependant is actual. The referenced object might be actual in the identical 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, all its direct and recursive dependants, which are not yet actual in that Edition, will be consequentially actualized in that identical Edition¹².

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

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.

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 precisely this.

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

This understanding needs to be extended when the referenced object, and therefore the dependant too, are editioned. Suppose the dependant is already actual in the identical Edition as the referenced object (after this has suffered the DDL) or in a descendant of that Edition. In that case, the full benefit of fine-grained dependency tracking is available, and invalidation that is not logically required is avoided. However, suppose on completion of the DDL to the referenced object, it is now in a younger edition than the dependant. In that case, the dependant is actualized into the referenced object's Edition in an invalid state¹⁵.

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; 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* defaults 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 differs from how *alter... compile* behaves; here, the target object is actualized into the current Edition.

DBMS_UTILITY.Compile_Schema() and the *Utl_Recomp* APIs are 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.

An EBR exercise will likely change editioned objects where some 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 been completed in the new Edition before proceeding to the next steps. *Utl_Recomp.Recomp_Parallel()* is a 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.Invalidite() has only one overload; this identifies the target object using *Object_ID*. Its only use in an EBR exercise would be to enable the PL/SQL compilation parameter values for many 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.Invalidite()* 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.Invalidite()* on an object not actual in the current Edition.

The effect of DDL in an edition with a child

Suppose a database has exactly *N* editions, *e1* through *eN*, where *e2* is the child of *e1*. Let *x[e₁]* denote an editioned object *x* that is actual in *e₁* and has no dependencies on any editioned objects. If no DDL has been done on *x* while

¹⁵ 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.

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, 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 to drop the object in question in the descendent Edition.) It can be seen, therefore, that, in general, the effect of DDL in any descendent editions might depend 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 cross-edition 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. Using the remote database’s *DBMS_Sql* package, at least a single SQL statement 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](#)¹⁵, 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
```

¹⁵ The datatype *simple_integer* has a *not null* constraint.

```

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 essential to understand this principle when a forward cross-edition trigger references an editioned package referenced by an ordinary application code.

Notice that the opposite is the case for a non-editioned 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 referred to as *data objects*. The prominent 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 circumstances to exist without representing differences.

However, the potentially enormous size of data objects makes such an approach impractical. The only practical approach is to let the user explicitly control the differencing. If the aim is to change a column, for example, by widening it, 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 precisely the suitable mechanism, but an ordinary view is too general in its power of expression. This forbids it from being treated like a table concerning some application requirements. For example, creating table-style triggers on an ordinary view is not allowed.

EBR introduces a new kind of view, the Editioning View. It is created using a 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 various 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 restrictions¹⁶. The following list is not intended to be complete; rather, it is intended to clarify the spirit of the design. The restrictions reflect the intention that an Editioning

¹⁶ 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

View must 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 that cannot be supported on an ordinary view can be supported. In particular, all memory of an Editioning View standing in front of a table is lost during SQL compilation. The resulting execution plan is identical to a query with the same meaning that targets the table(s) directly. In other words, using 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 stored in the base table.
- **The table's owner must own an Editioning View** 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 specific edition**, as having more than one logical projection of the same table data in the identical Edition is meaningless.
- **The subquery factoring clause is not allowed** because of the other restrictions; the subquery factoring clause could have no practical usability benefit.
- **The subquery must be a single query block**, meaning 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 precisely one table.** The *from list* must have just one item: a *table*. A self-join is prohibited¹⁷; the item cannot be a view or 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. They rarely need to add or remove rows in a table. For such scenarios, different occurrences of the Editioning View must denote different physical tables in various editions¹⁸.
- **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 permitted.

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 "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 data used to configure the application's behavior. Moreover, such data usually is not modifiable by ordinary end-user actions but is changed only by an administrator. A very straightforward approach to online application upgrade is possible in such cases. A

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.

¹⁷ ANSI join syntax is therefore disallowed.

¹⁸ 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.

new table is defined and populated ordinarily. It 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 in the old Edition. By setting these Editioning Views with read-only, the intention that end-users do not change the table content is formally enforced. Of course, the alter view command can make an Editioning View 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 and foreign key constraints can be created on an ordinary view, but the keywords *disable novalidate* must be used. The benefit is 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 beneficial to state the overall principle:

DML Operations support

Any *select*, *insert*, *update*, *delete*, *merge*, *lock table*, or *explain plan* SQL statement¹⁹ running without error on a table will run without error on an Editioning View covering that table.

An Editioning View allows table-style triggers

Triggers defined on renamed tables become invalid because they are still attached to those tables²⁰. 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 them²¹.

Notice that when DML is issued using an Editioning View, triggers not only defined on the Editioning View will fire but also those defined on its base table. However, only the triggers defined on the table will fire when DML is issued using a table. Triggers defined on the Editioning View will not fire. The paradigm requires that all regular application DML be issued using Editioning Views; as seen (see "[The cross-edition trigger](#)"), only cross-edition triggers are allowed to do DML using tables.

Hint in a SQL statement targeting Editioning View can identify an index by listing the names of its columns

Again, this allows extant application code to remain correct after introducing 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, 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.

¹⁹ For example, *select Rowid, ev.* from ev* is legal when *ev* is an editioning view.

²⁰ 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 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.

²¹ 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 be defined on the renamed table.

```

-- 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. P.K., a.Info from t a
/
-- Causes ORA-14109
select * from v partition(p1)
/
create Editioning View ev as select a. P.K., a.Info from t a
/
-- Runs without error
select * from ev partition(p1)
/

```

EBR using only Editions and Editioning Views

Suppose an application upgrade changes only those tables whose data is not changed via the standard end-user interfaces. In that case, the Edition and the Editioning View are sufficient to allow these changes to be made while the application remains online. The most obvious example is configuration data, which determines the application's behavior and is changed only as part of an upgrade. Such data is typically not voluminous, 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 a different table in the post-upgrade Edition. The upgrade installation script can populate the replacement table as required. According to the upgrade requirements using EBR, 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 Cross-edition Trigger

Sometimes, an application upgrade has to change one or more tables whose content is queried and altered by ordinary end-user interaction. Consider a use case; for example, a single column representing a telephone number as it would be used when dialing 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 the 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, converting 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 the right properties to affect the proper responses to end-user changes during the bulk forward data transformation and the hot rollover period. Moreover, using a trigger for this purpose meets the high-level requirement that the 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 complete.

A Cross-edition Trigger is a special kind of trigger; 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 editioned²². This restriction is because the firing rules for a cross-edition trigger are defined for the relationship between the Edition in which it is actual and the current Edition of the session that issues the DML. Further, a cross-edition 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 concerning the firing rules that a cross-edition 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 cross-edition triggers (by virtue of *follows* or *precedes* relationships) can exist only between sets of cross-edition triggers that are

²² If a user that is not editions-enabled attempts to create a cross-edition trigger, this causes ORA-25030.

actual in the identical Edition²³. If the clause is “follows,” the target must be a forward cross-edition trigger; if the clause *precedes*, the target must be a reverse cross-edition trigger.

The compilation of a cross-edition trigger follows the standard 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 cross-edition trigger and all code it calls always *runs* using the Edition in which it is actual. [Code 20](#) is an SQL*Plus script that shows this.

A cross-edition trigger may be created only directly on a table—and not on either a regular view or an Editioning View²⁴. 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 cross-edition trigger. A cross-edition trigger may be a compound trigger.

Basic firing rules for cross-edition triggers

The firing rules were designed on the assumption that the cross-edition triggers required to implement a particular upgrade are all installed in the post-upgrade Edition. This is consistent with the overall paradigm that (so that the pre-upgrade application will be unperturbed) *all* DDL to editioned objects are 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 standard application code) only by sessions using the post-upgrade Edition. There are, therefore, two kinds of cross-edition triggers:

- A *forward cross-edition trigger* is fired by the application DML issued by sessions using the pre-upgrade Edition. Such a trigger transforms the old representation forwards into the new representation.
- A *reverse cross-edition trigger* is fired by the application DML issued by sessions using the post-upgrade Edition. Such a trigger transforms the new representation backwards into the old representation.

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

- A forward cross-edition trigger is fired by application DML issued by a session using any ancestor edition to that in which the trigger is actual.
- A reverse cross-edition 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 cross-edition 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, 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);
```

²³ This restriction ensures that no contradictions about firing order can be expressed. As will be seen, the firing order of cross-edition triggers in a particular edition cannot be interleaved with that of cross-edition triggers in a different edition.

²⁴ The attempt causes ORA-42306: a cross-edition trigger may not be created on an editioning view.

```

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
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 cross-edition 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 cross-edition 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 no cross-edition trigger issues DML, it is sufficient to understand these basic firing rules.

Advanced firing rules for cross-edition triggers

We will use the term *cross-edition trigger DML* for DML issued directly, using embedded SQL or native dynamic SQL, from the PL/SQL unit that is a cross-edition trigger. We will use the term *regular DML* for DML issued from *any other site*. Notice that this definition means that DML issued from a PL/SQL unit invoked by a cross-edition trigger is *regular DML*. In particular, DML issued using the `DBMS_Sql` API is, by default, regular DML, even when these subprograms are invoked directly from implementing a cross-edition trigger. However, if the name of the cross-edition trigger that invokes the `DBMS_Sql` API is included in the actual `Applying_Cross-edition_Trigger()` formal parameter to `DBMS_Sql.Parse()`, the DML that the `DBMS_Sql` API issues will be cross-edition trigger DML.

- Regular DML always fires both visible regular triggers and appropriately selected cross-edition triggers.
- The firing order of cross-edition triggers in a particular edition is never interleaved with that of cross-edition triggers in a different edition. All forward cross-edition triggers in edition e will fire before any in a descendent edition of edition e. All reverse cross-edition triggers in edition e will fire after any in an ancestor edition of edition e.
- Cross-edition trigger DML from a forward cross-edition trigger actual in edition e will fire forward cross-edition triggers that are actual in descendants of edition e but will never fire reverse cross-edition triggers or regular triggers.
- Correspondingly, cross-edition trigger DML from a reverse cross-edition trigger actual in edition e will fire reverse cross-edition triggers that are actual in ancestors of edition e but will never fire forward cross-edition triggers or regular triggers.

- Recall 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 cross-edition 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.
- Cross-edition trigger DML from a unit that is actual in edition *e* does not, unless special programming steps (described in the following two bullet points) are taken, fire cross-edition triggers that are actual in edition *e*.
- If forward cross-edition trigger `Fwd_Xed_1`, on table *t1*, issues cross-edition trigger DML to table *t2*, then forward cross-edition 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 cross-edition triggers).
- Correspondingly, if reverse cross-edition trigger `Rev_Xed_1`, on table *t1*, issues cross-edition trigger DML to table *t2*, then reverse cross-edition 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 transitive²⁵.

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

While forward cross-edition triggers are necessary to propagate changes that happen to be made to the pre-upgrade representation by user activity, having them in place is insufficient 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 cross-edition trigger to fire. This is trivially achieved by updating each forward cross-edition trigger's base table to set a column that fires the trigger on update to itself. However, there is a little more to this than you might initially think.

Using `DBMS_Sql.Parse()` to apply a forward cross-edition trigger

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

`DBMS_Sql.Parse()` has overloads with the formal parameter `Apply_Cross-edition_Trigger`. These overloads also have the formal parameters `Edition` and `Fire_Apply_Trigger`. `Apply_Cross-edition_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_Cross-edition_Trigger` to fire the forward cross-edition trigger `Fwd_Xed` on table *t* for each row.

-- Code_22

```
DBMS_Sql.Parse(
C => The_Cursor,
Language_Flag => DBMS_Sql.Native,
Statement      => 'update t set c1 = c1', Apply_Cross-edition_Trigger => 'Fwd_Xed');
```

When `Edition` is `null`, 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 cross-edition triggers are the only triggers you can *apply* (cause to fire on every table row on which they are defined).

²⁵ 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`.

Cross-edition triggers must be idempotent

It is impossible to predict whether a particular row that is to be transformed by a forward cross-edition 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. Therefore, the action of a forward cross-edition trigger must be idempotent by explicit design. (Similar rationale holds for the design of a reverse cross-edition trigger—even though these are never the subject of an *apply* step.)

When a replacement table is used, every row in the original table must be reflected in the replacement. If the source row is visited first by ordinary end-user activity, no further action is needed when the *apply* step visits the same row. (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 implemented. It is, however, necessary to detect that the *apply* step is in progress if this is implemented by merely causing the forward cross-edition trigger that implements the transform to fire for every row. The boolean function *Applying_Cross-edition_Trigger()* in the package *DBMS_Standard* is provided for this purpose.

It is possible, of course, that when the forward cross-edition 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 cross-edition triggers—*DBMS_UTILITY.Wait_On_Pending_DML()*

So that there be no “lost updates” during the *apply* step, the following logic must be used.

- Enable the forward cross-edition 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 that began prior to this function’s invocation have either been committed or rolled back.
- Start the *apply* step.

Using the *DBMS_Parallel.Execute()* API

If the table that 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 likely 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 need 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 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 using EBR to implement the *apply* step simply by causing the forward cross-edition trigger(s) that implement the transform for each row of the table, this is not always the approach 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 cross-edition 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 cross-edition trigger designated by *Apply_Cross-edition_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_Cross-edtion_Trigger => 'Fwd_Xed',
Fire_Apply_Trigger  => false);
```

It is necessary to specify the name of the forward cross-edtion trigger, *Fwd_Xed*, that implements the same transform so that the closure of other forward cross-edtion triggers in *follows* relationship with 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 vast and complex, may be developed and maintained by a large team, and 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 transform 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 will likely 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 different bug fixes; in such cases, the second approach offers more flexibility. When the first approach is implemented using EBR, it is implausible that the advanced firing rules for cross-edtion triggers will be helpful. The exercise will use only a single new edition and no cross-edtion trigger *Trg2* will implement logic to respond to a change that a different cross-edtion 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 cross-edtion trigger *Trg2* must fire only after another cross-edtion 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 and descendent editions. The fact that cross-edtion trigger DML from a forward cross-edtion trigger will fire only those forward cross-edtion triggers in descendent editions (and correspondingly for reverse cross-edtion triggers) avoids circular firing. However, it is less cumbersome to use only a single new edition; in this case, the fact that cross-edtion trigger DML will never fire cross-edtion triggers in the identical 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) allows 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 supporting the other subgoals of 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 architect will design the latest version, which upgrade scripts created by the developers will deliver. The upgrade to the EBR-readied version must be done in downtime because tables will be renamed, and dependent objects will be invalidated. Revalidation is possible only when an Editioning View covers each table and restores its former name.

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, 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. This is not an issue for a new application that can quickly be built to be EBR-readied right before 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](https://www.oracle.com). Outside North America, find your local office at: [oracle.com/contact](https://www.oracle.com/contact).

 blogs.oracle.com

 facebook.com/oracle

 twitter.com/oracle

Copyright © 2024, Oracle and/or its affiliates. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.