



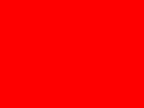
ORACLE[®]

Edition-based Redefinition:

**Testing Live Application Upgrades (Without
Actually Being Live)**

Melanie Caffrey

Senior Development Manager, Unbreakable Linux Network, Oracle Linux



The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. 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 for Oracle's products remain at the sole discretion of Oracle.



What Problem Are We Trying to Solve?

PL/SQL Application Upgrades that Require Downtime (and Testing ...)

- Not possible to obtain long (or frequent) downtime windows
- The testing window during downtime can be inadequate
- An upgraded application can be difficult to back out of, if necessary



Edition-based Redefinition: Edition Object Type and EBR Features

By the way, it's free!

- Not to keep you in suspense...
- EBR is not a priced option
- Nor is it even restricted to just the Enterprise Edition
- Available with any licensed version of Oracle Database 11g Release 2, or later

Edition Object Type

- 11.2 introduces the new object type, *edition* – each edition can have its own private occurrence of “the same” object
- A database must have at least one edition (by default this is ora\$base)
- You create a new edition as the child of an existing edition (and an edition can't have more than one child)

Edition-based Redefinition Features

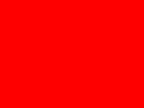
- Edition
- Editioning View
- Cross-edition Trigger
- Code changes are installed in the privacy of a new edition (namespace addition)
- Data changes can write to new columns or tables (and not be seen by old edition)

Edition-based Redefinition Features

- Edition
- Editioning View
- Cross-edition Trigger
- Exposes a different projection of a table into each edition to allow each to see just its own columns

Edition-based Redefinition Features

- Edition
- Editioning View
- Cross-edition Trigger
- Propagates data changes made by the old edition into the new edition's columns, or (in hot-rollover) vice-versa

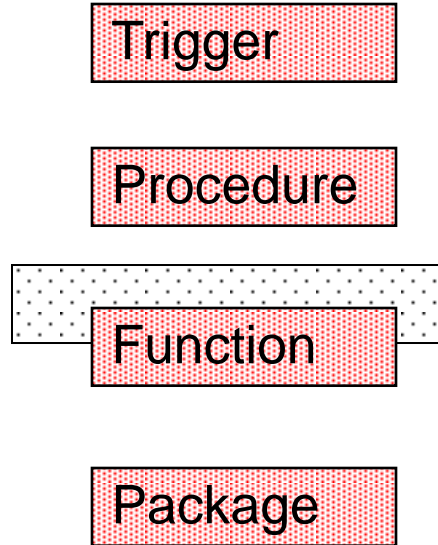


Ready Your Application for Editions and EBR

Editable and non-editable object types

- Not all object types are editable
 - Synonyms, views, and PL/SQL units of all kinds (including triggers, procedures and packages) are editable
 - Objects of all other object types – for example, tables – are non-editable
- However, you *can* achieve the goal of table-editing with an editing view. You version the structure of a table manually.
 - Instead of *changing* a column, you *add a replacement* column.
 - Then you rely on the fact that a *view* is editable

Pre-Upgrade implementation model



Ora\$base edition
– App v1

Edition Setup

As of 11gR2, each database has at least one edition

```
CONN / AS SYSDBA
```

```
SELECT property_value  
  FROM database_properties  
 WHERE property_name = 'DEFAULT_EDITION';
```

```
PROPERTY_VALUE
```

```
-----
```

```
ORA$BASE
```

Edition Setup

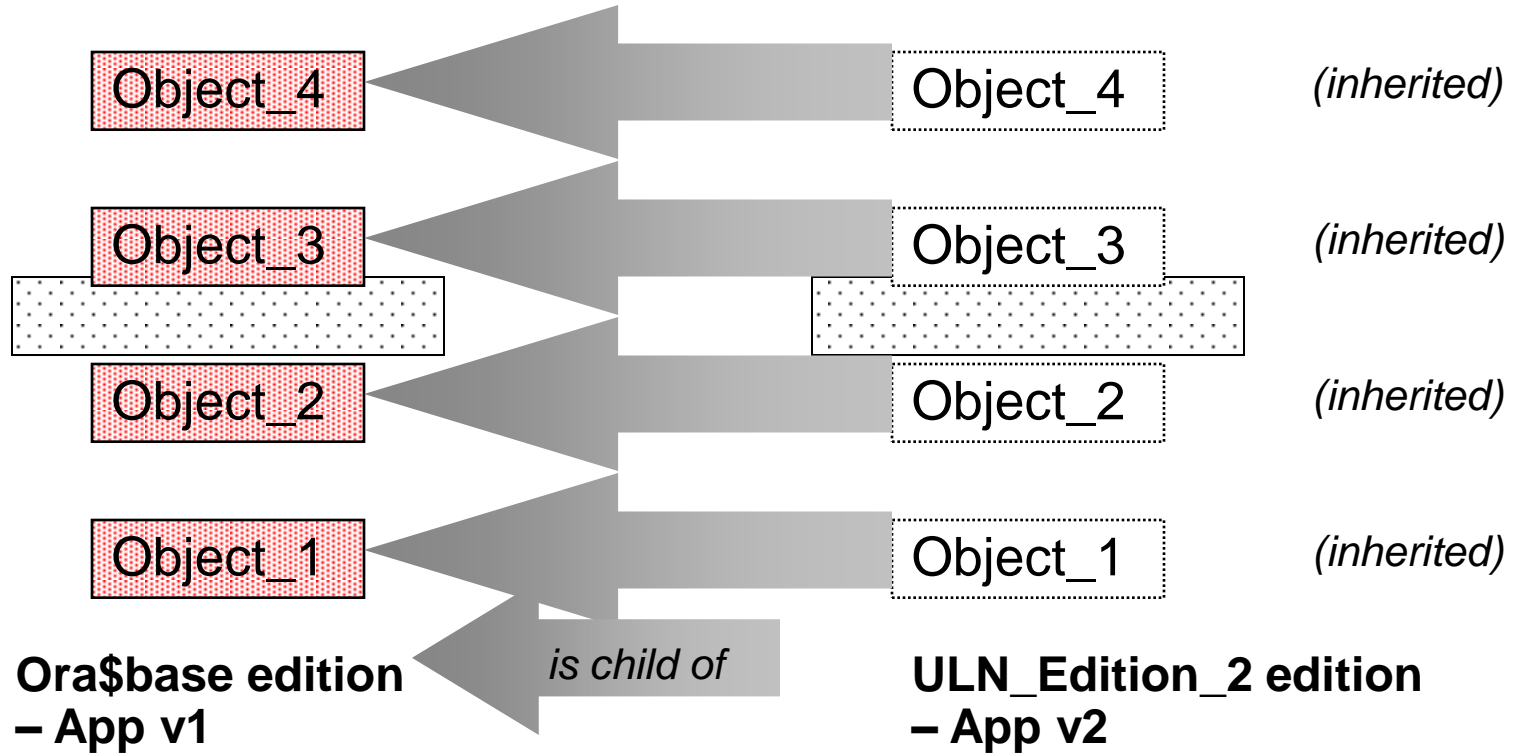
- You need the CREATE ANY EDITION or DROP ANY EDITION system privilege to create or drop editions

```
SQL> create edition uln_edition_2
      2 as child of ora$base;
Edition created.
```

```
SQL> select * from dba_editions;
```

EDITION_NAME	PARENT_EDITION_NAME	USA
ORA\$BASE		YES
ULN_EDITION_2	ORA\$BASE	YES

Editions: implementation model



Edition Setup

- Alter your application user to be *editions-enabled* and grant them the ability to *use* the newly-created edition

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

```
SQL> grant use
      2 on edition uln_edition_2
      3 to app_user;
      Grant succeeded.
```

Switch to the New Edition to Make Code Changes

```
CONN uln_app_user/pw
```

```
SQL> alter session  
2 set edition = uln_edition_2;
```

```
Session altered.
```

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

```
EDITION
```

```
-----  
ULN_EDITION_2
```

Editioning Views

- Physical *Table* = Scott.Emp_T
- Logical *View* = Scott.Emp
- Ora\$Base.Scott.Emp <> ULN_Edition_2.Scott.Emp
- If Scott owns Emp_T, then Scott must also own Emp
- All Application code refers only to Scott.Emp (NOT Scott.Emp_T)
- Drop all Triggers from Emp_T and Recreate them on Emp

Readying the application for editions

- “Slide in” an editioning view in front of every table
 - Rename each table you want to *edition* (e.g. *rpm* becomes *rpm_t* (to distinguish it now as a *table*, *_t*, as opposed to an *editioning view*, which *rpm* will become))
 - alter table rpm rename to rpm_t;
 - Create an editioning view for each table that has the same name that the table originally had
 - create editioning view rpm as select * from rpm_t;
 - **NOTE:** *You will need an outage to create your editioning views.*

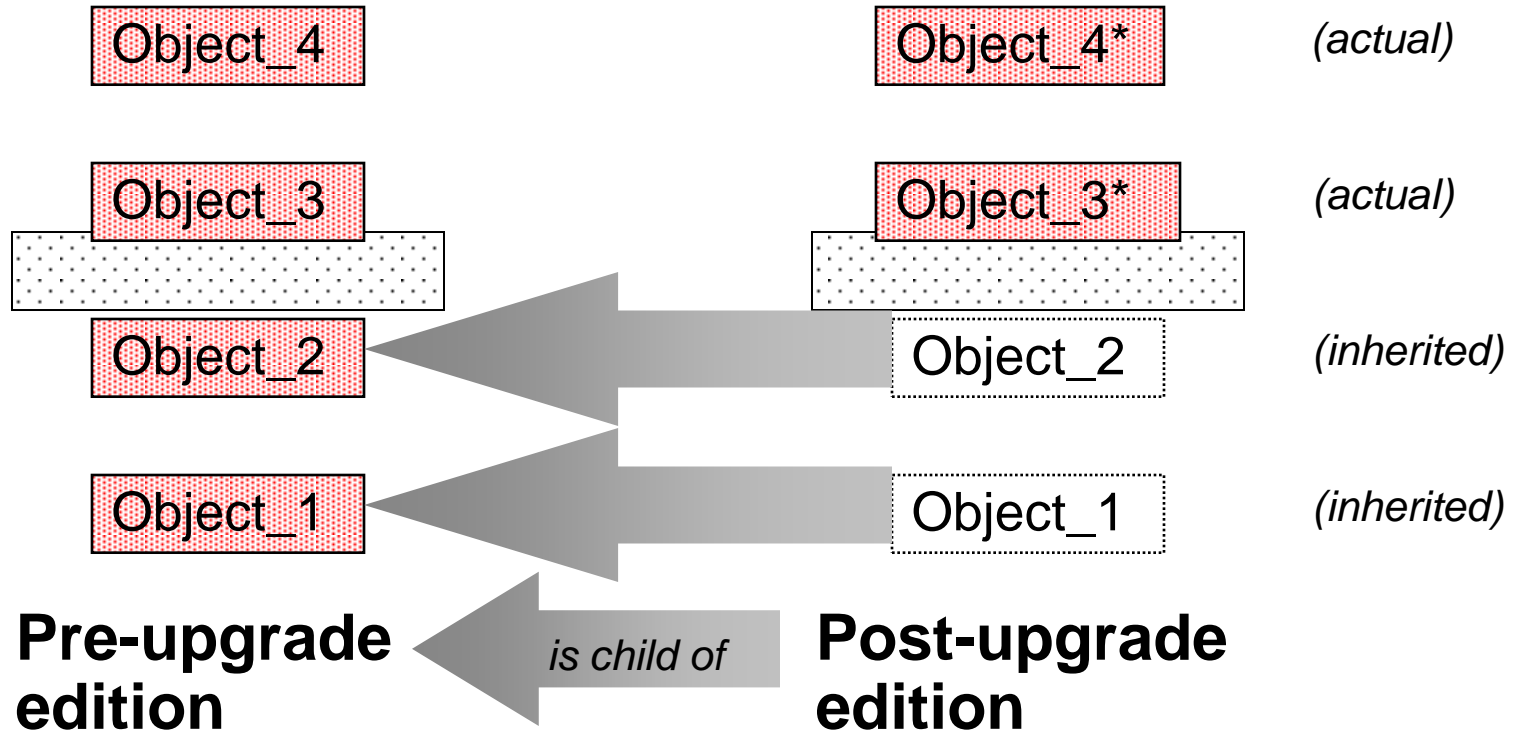
Readying the application for editions

- Alter your *real and actual* tables as needed:
 - alter table *rpm_t* add (vers1 number(10), vers2 number(10) rel1 number(10), rel2 number(10) ...);
- “Move” triggers to the editioning views ... (*next slide*)
- Revoke privileges from the tables and grant them to the editioning views
- Move VPD policies to the editioning views

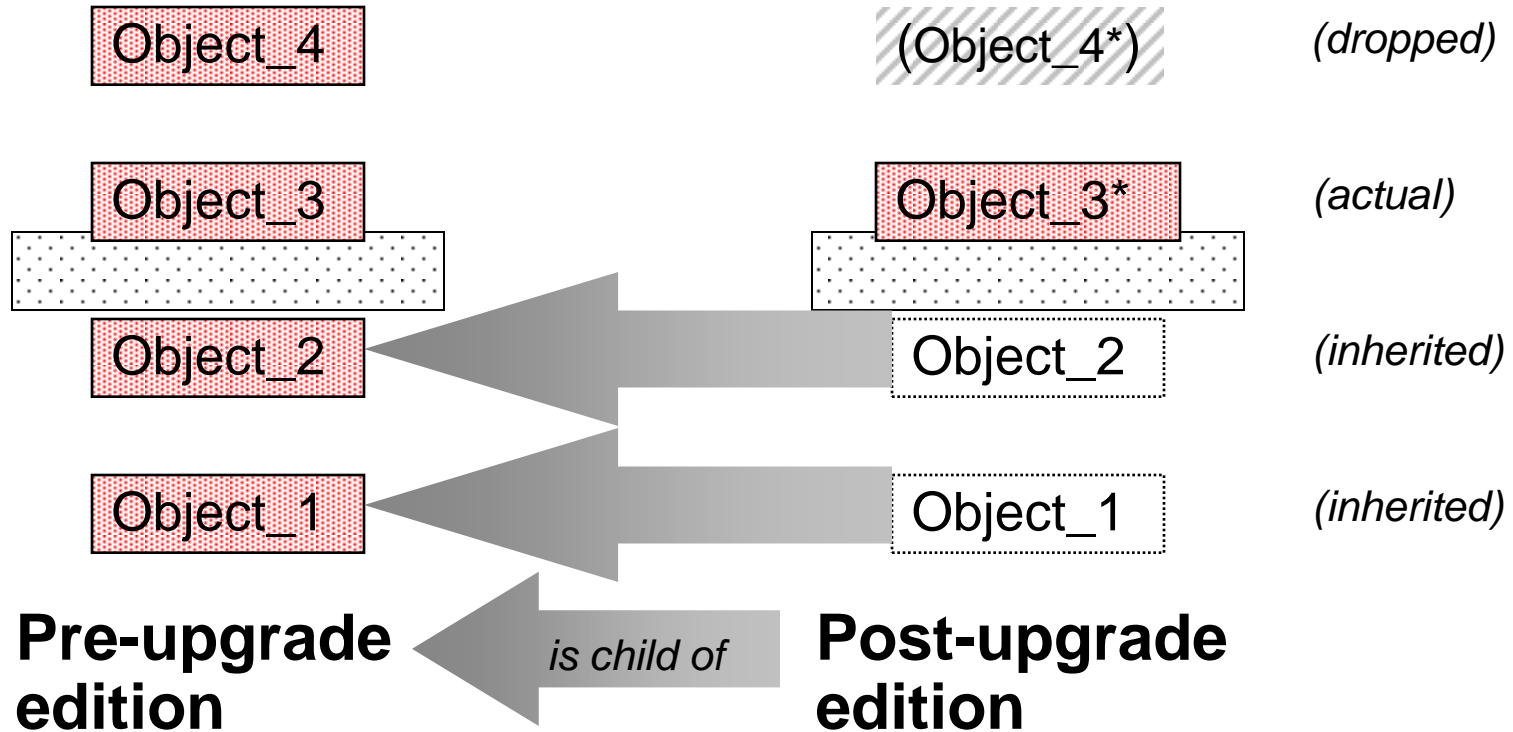
Readying the application for editions

- Of course,
 - All indexes on the original *RPM* table remain valid but *User_Ind_Columns* now shows the new values for *Table_Name* and *Column_Name*
 - All constraints (foreign key and so on) on the original *RPM* remain in force for *RPM_T*
- However,
 - Triggers don't fully "follow" the rename
 - Just drop the trigger and re-run the original create trigger statement to "move" the trigger onto the editioning view

Editions: implementation model



Editions: implementation model





**Many, if not Most, of Your Application
Upgrades Can Be Completed Just By
Using Editions and Editioning Views**



Here is Where it Starts To Get Tricky ...

What if DML cannot stop during upgrade?

- If the upgrade needs to change the structure that stores transactional data – like the RPM data customers use with ULN – then the installation of values into the replacement columns must keep pace with these changes
- Triggers have the ideal properties to do this safely
- Each trigger must fire appropriately to propagate changes to pre-upgrade columns into the post-upgrade columns – and vice versa

The solution: crossedition triggers

- Crossedition triggers directly access the table.
- The 11gR2 crossedition trigger has special firing rules
- You create crossedition triggers in the *Post_Upgrade* (child) edition
 - The paradigm is: don't interfere with the *Pre_Upgrade* (parent) edition
- The firing rules assume that
 - Pre-upgrade column values are changed – by ordinary application code – only by sessions using the *Pre_Upgrade* (parent) edition
 - Post-upgrade column values are changed only by sessions using the *Post_Upgrade* (child) edition

The solution: crossedition triggers

- A *forward* crossedition trigger is fired by application DML issued by sessions using the *Pre_Upgrade* (parent) edition
- A *reverse* crossedition trigger is fired by application DML issued by sessions using the *Post_Upgrade* (child) edition
- Both types of crossedition triggers are owned by the *Post_Upgrade* (*child*) edition

(even though, for a forward crossedition trigger, the session that fires it is using the *Pre_Upgrade* (parent) edition)

Case study – The edition-based redefinition exercise proper



Case study

- The Oracle Linux RPM packages, downloadable when Unbreakable Linux Network Support is purchased, are stored as four components in four columns:

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

- It is necessary to *parse* those “dot-delimited” parts of the version and release strings into their own separate components in order to evaluate and compare one kernel RPM to another, to determine which is more recent

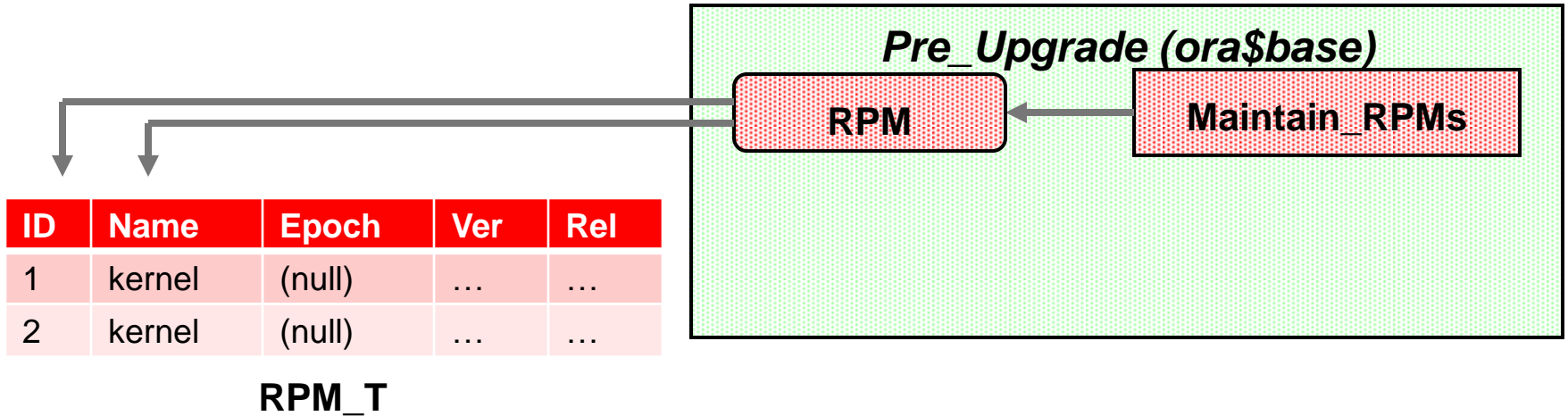
Case study (continued)

- So we want a uniform representation with as many version-related and release-related columns as necessary (for purposes of brevity, this example includes only versions and releases with four *parts*):

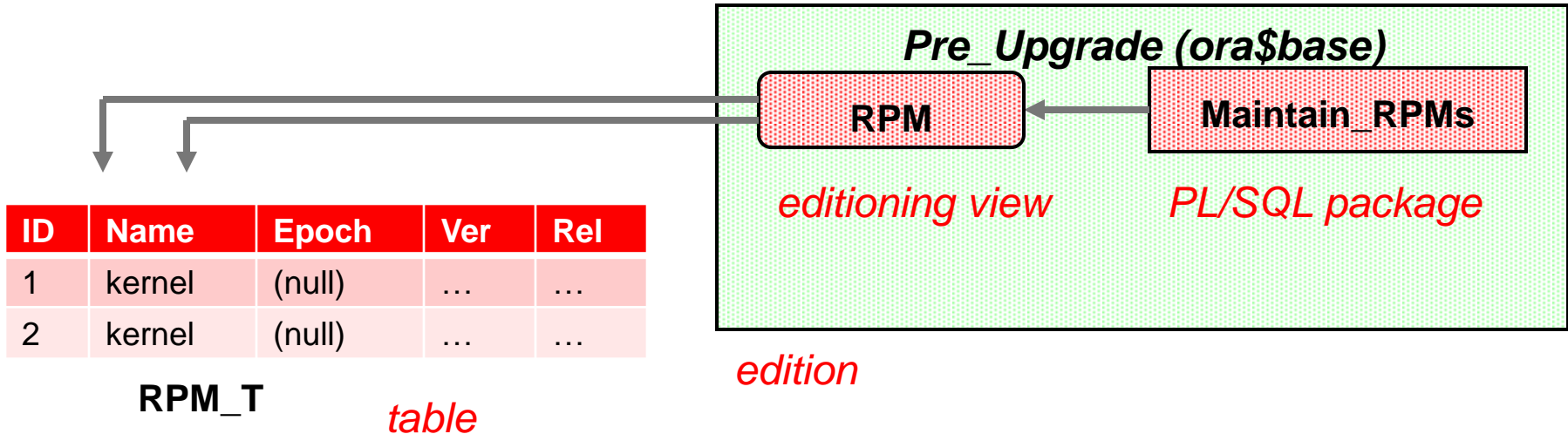
Name	Epoch	V1	V2	V3	V4	R1	R2	R3	R4
kernel	(null)	2	6	32	(0)	100	21	1	el5 (000)
kernel	(null)	2	6	18	(0)	92	1	6	el5 (000)

- This way, instead of comparing Varchar2 strings, we can compare individual numeric values

Starting point.
Pre-upgrade app in normal use.



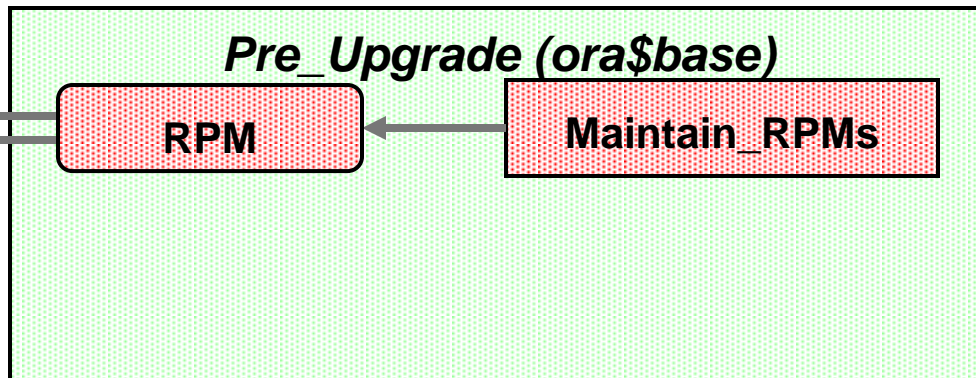
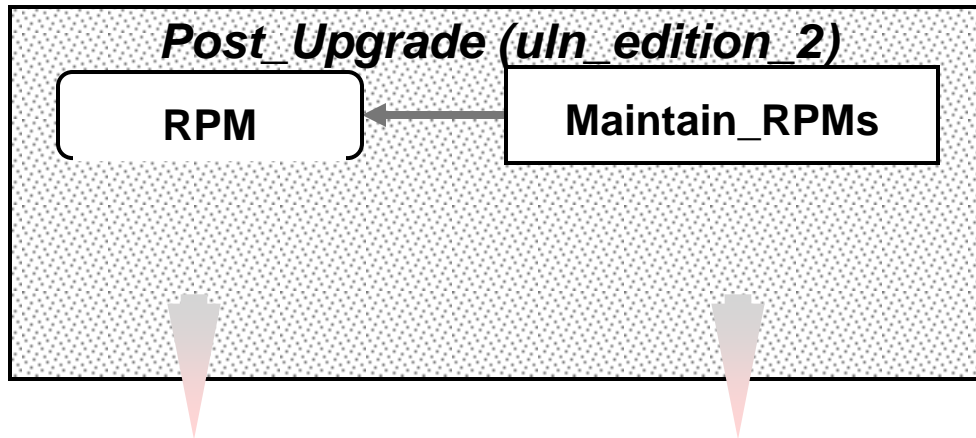
Starting point.
Pre-upgrade app in normal use.



Start the edition-based redefinition exercise.

Create the new edition as the child of the existing one.

This is fast because initially all the editioned objects are just inherited.

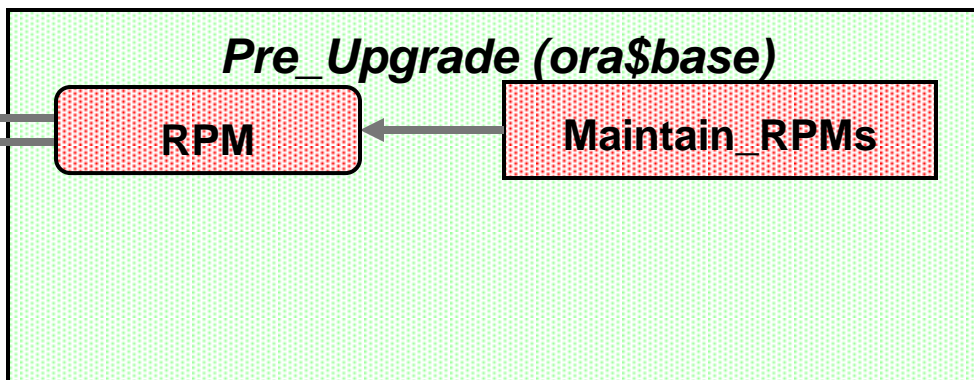
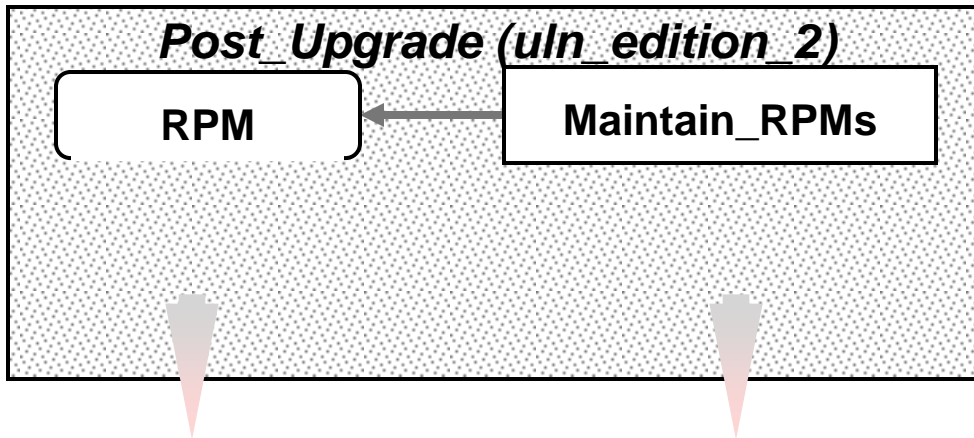
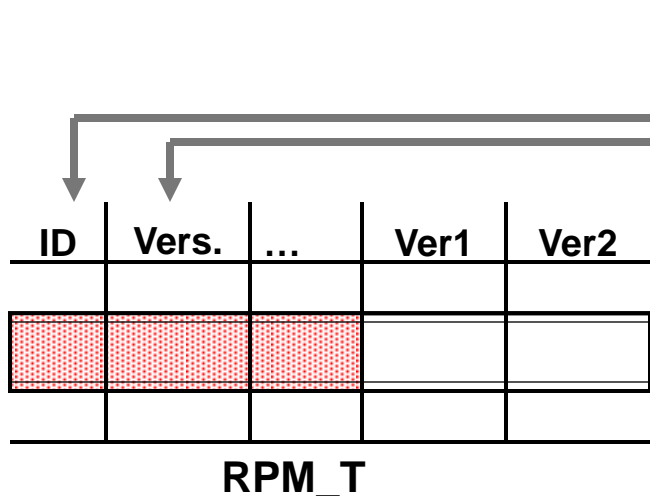


ID	Name	Epoch	Ver	Rel
1	kernel	(null)
2	kernel	(null)

RPM_T

Create the replacement columns in the underlying table.

The editioning view shields the app from this change.



Altering Your Underlying Table(s)

- Put your replacement columns in place

```
SQL> alter table rpm_t add
      2      (ver1 number, ver2 number, ver3 number, ver4 number,
      3      rel1 number, rel2 number, rel3 number, rel4 number);
Table altered.
```

(You can successfully avoid the error message, ORA-00054: resource busy and acquire with NOWAIT specified)

- Prepare to migrate the relevant data to these newly added columns
- You will do so in your child (next version) edition

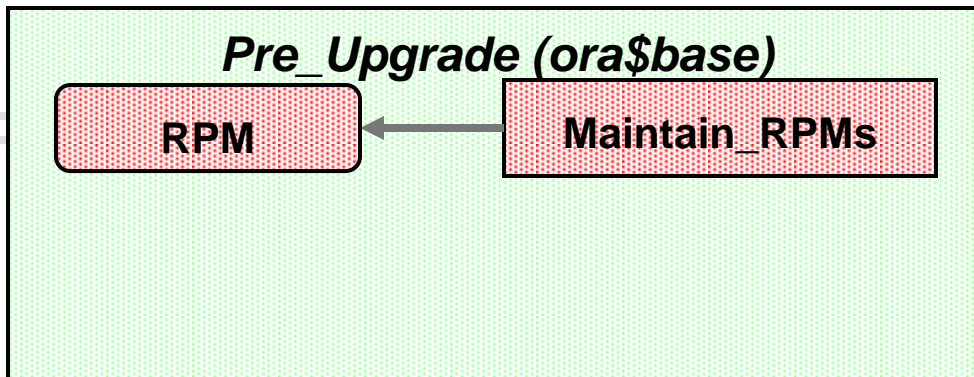
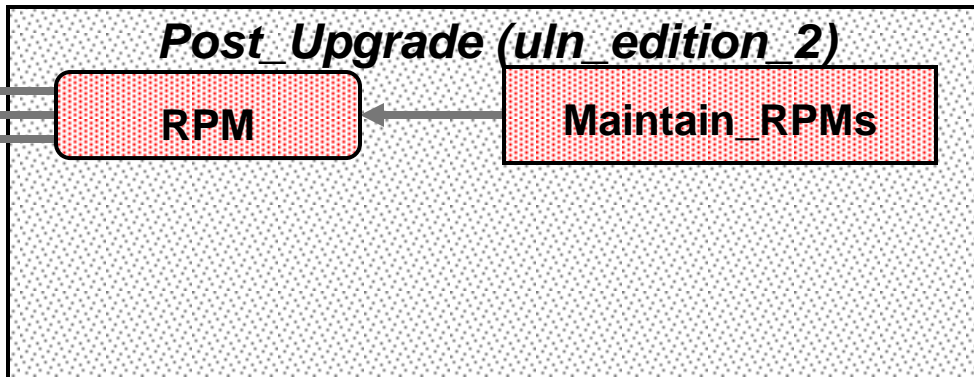
```
SQL> alter session set edition = uln_edition_2;
```

Change *RPM* DML code to select the new columns.

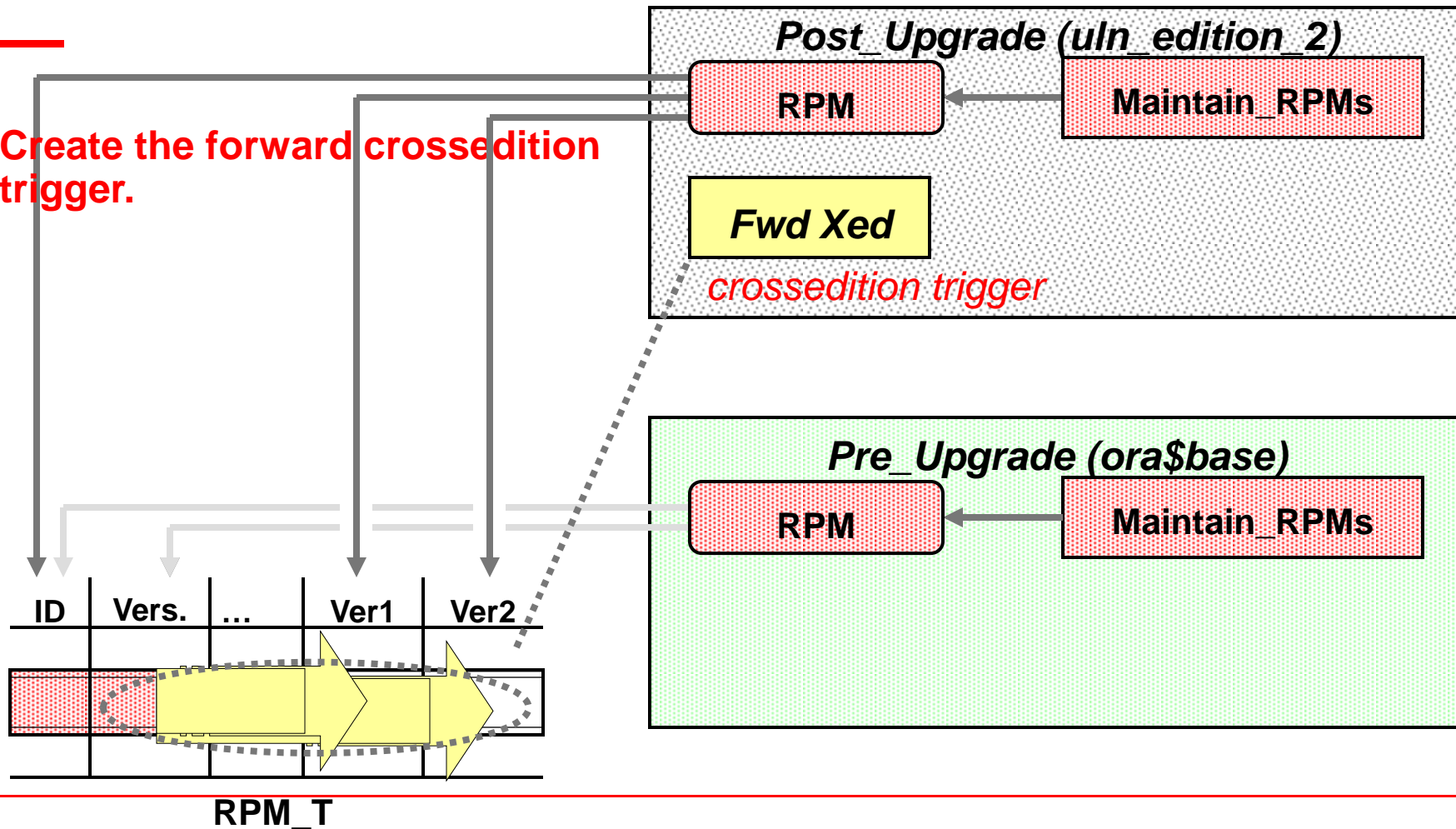
Change *Maintain_RPMs* to implement the new behavior.

ID	Vers.	...	Ver1	Ver2

RPM_T



Create the forward crossedition trigger.



Create Your Forward Cross-edition Trigger

Your cross-edition trigger is necessary for ongoing data migration/population during an online application upgrade

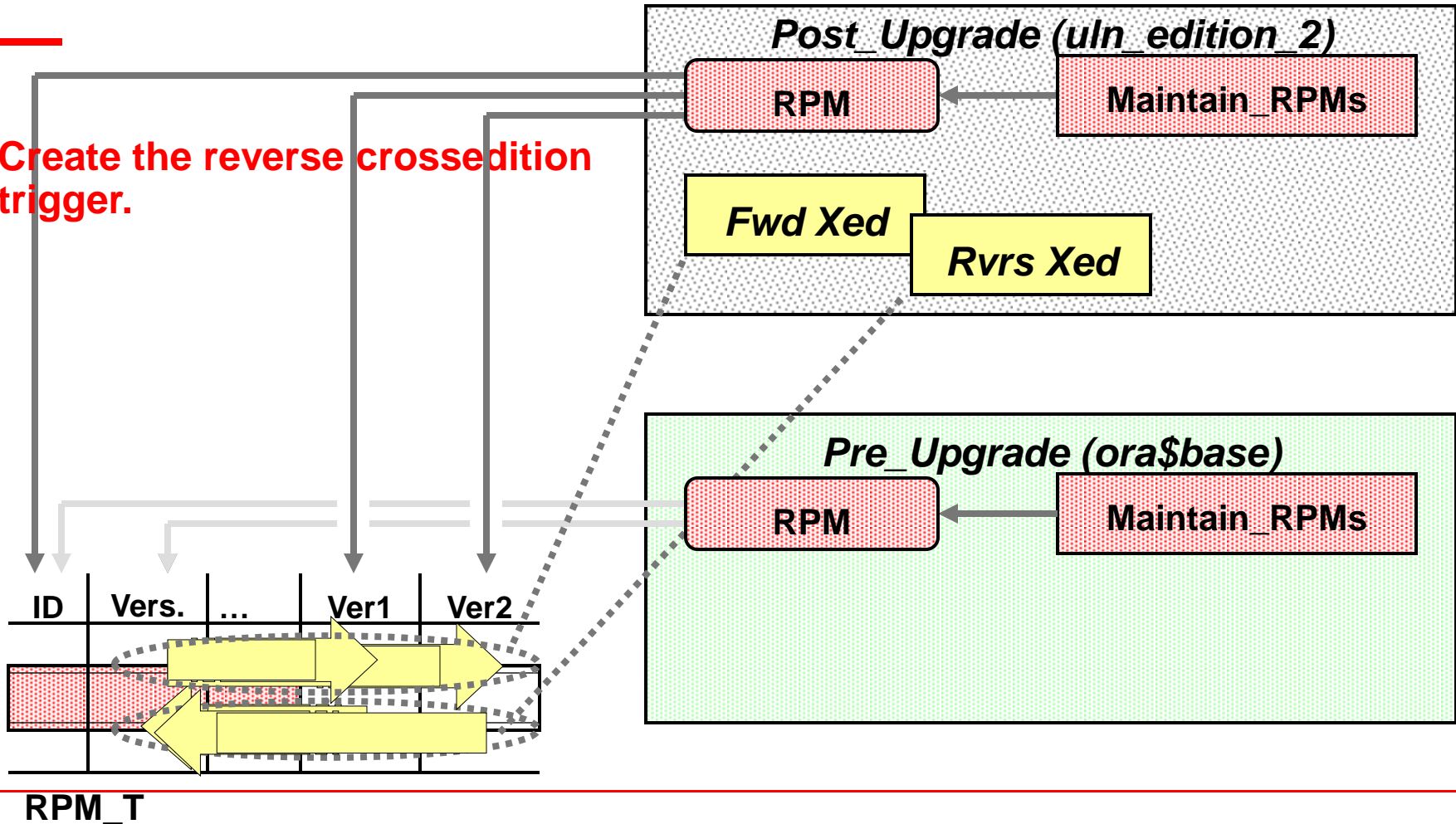
```
SQL> create or replace trigger rpm_fwdxedition
  2  before insert or update of version, release on rpm_t
  3  for each row
  4  forward crossedition
  5  declare
  6  v_verstring VARCHAR2(50) := '.'||:new.version||'.';
  7  v_relstring VARCHAR2(50) := '.'||:new.release||'.';
  8  begin
  9      :new.ver1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


Create Your Forward Cross-edition Trigger

```
21      :new.rel1 := substr( v_relstring,  
22          instr(v_relstring,':',1,1)+1, instr(v_relstring,':',1,2) -  
23          instr(v_relstring,':',1,1)-1);  
24      ...  
33  end;  
34  /
```

Trigger created.

Create the reverse crossedition trigger.



Create your Reverse Cross-edition Trigger

- Your reverse cross-edition trigger is necessary for hot rollover purposes

```
SQL> create or replace trigger rpm_revxedition
```

```
2 before insert or update of ver1, ver2, ver3, ver4, rel1, rel2,
```

```
3 rel3, rel4, on rpm_t
```

```
4 for each row
```

```
5 reverse crossedition
```

```
6 begin
```

```
7 :new.version :=
```

```
8 rtrim(:new.ver1||'.'||:new.ver2||'.'||:new.ver3||'.'||
```

```
9 :new.ver4, '.');
```

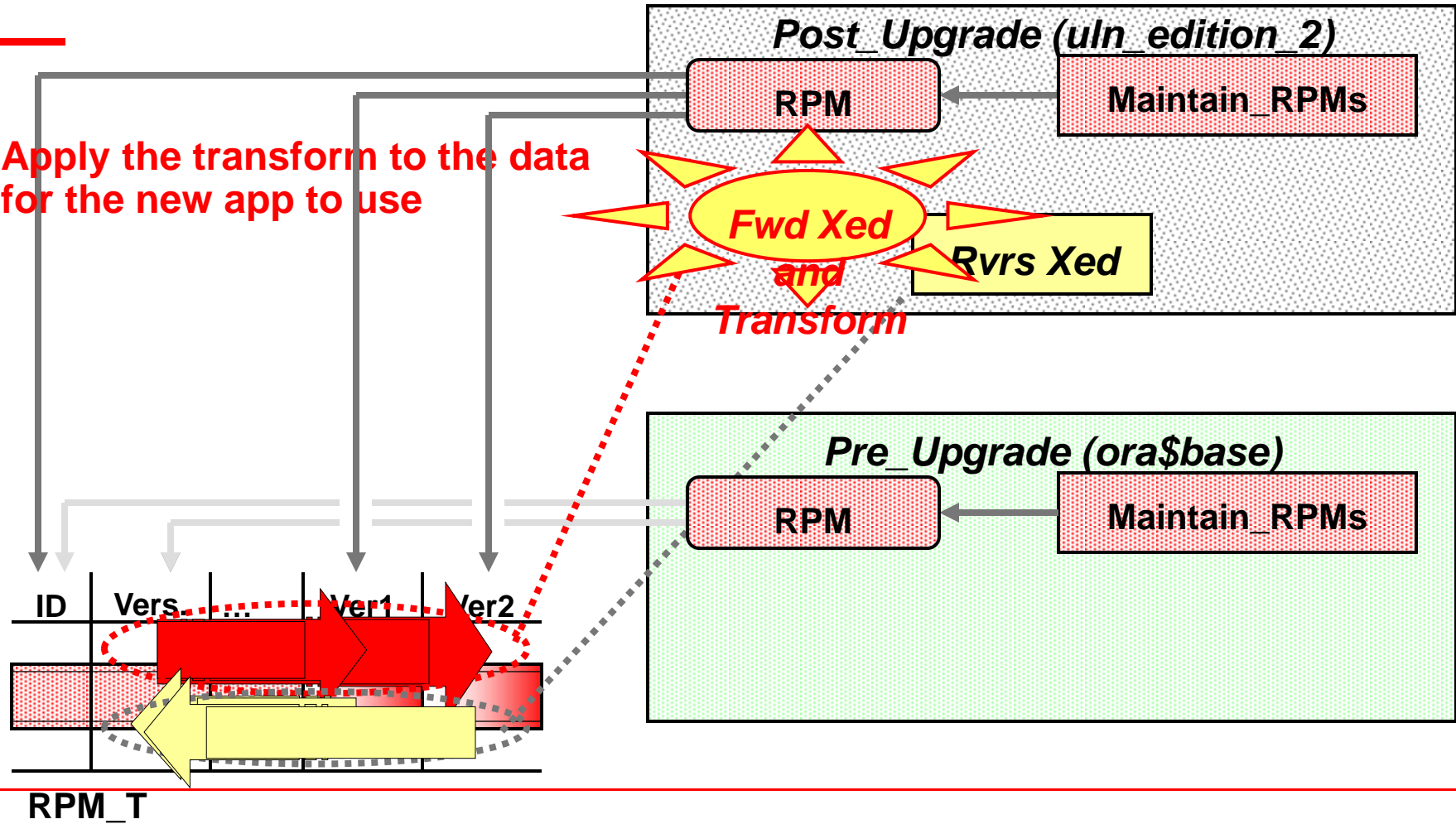
```
10 :new.release :=
```

```
11 rtrim(:new.rel1||'.'||:new.rel2||'.'||:new.rel3||'.'||
```

```
12 :new.rel4, '.');
```

```
13 end;
```

Apply the transform to the data for the new app to use



Transform Your Data for New Columns

- Get the data from the old columns into the new columns

- You could do the following

```
SQL> update rpm_t
```

```
2      set version = version,
```

```
3      release = release;
```

- **Beware:** This action locks the entire table
- Consider DBMS_PARALLEL_EXECUTE if your tables are large

```
SQL> begin
```

```
2  dbms_parallel_execute.create_task(  
3    'update rpm_t');
```

```
4  dbms_parallel_execute.create_chunks_by_rowid
```

```
5    ( task_name => 'update rpm_t',
```

```
6      table_owner => user,
```

```
7      table_name => 'RPM_T',
```

Transform Your Data for New Columns

```
8   by_row    => false,  
9   chunk_size => 10);  
10 end;  
11 /
```

PL/SQL procedure successfully completed.

- Running the task

SQL> begin

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

Transform Your Data for New Columns

- When satisfied with the results, simply drop the task

```
SQL> begin
```

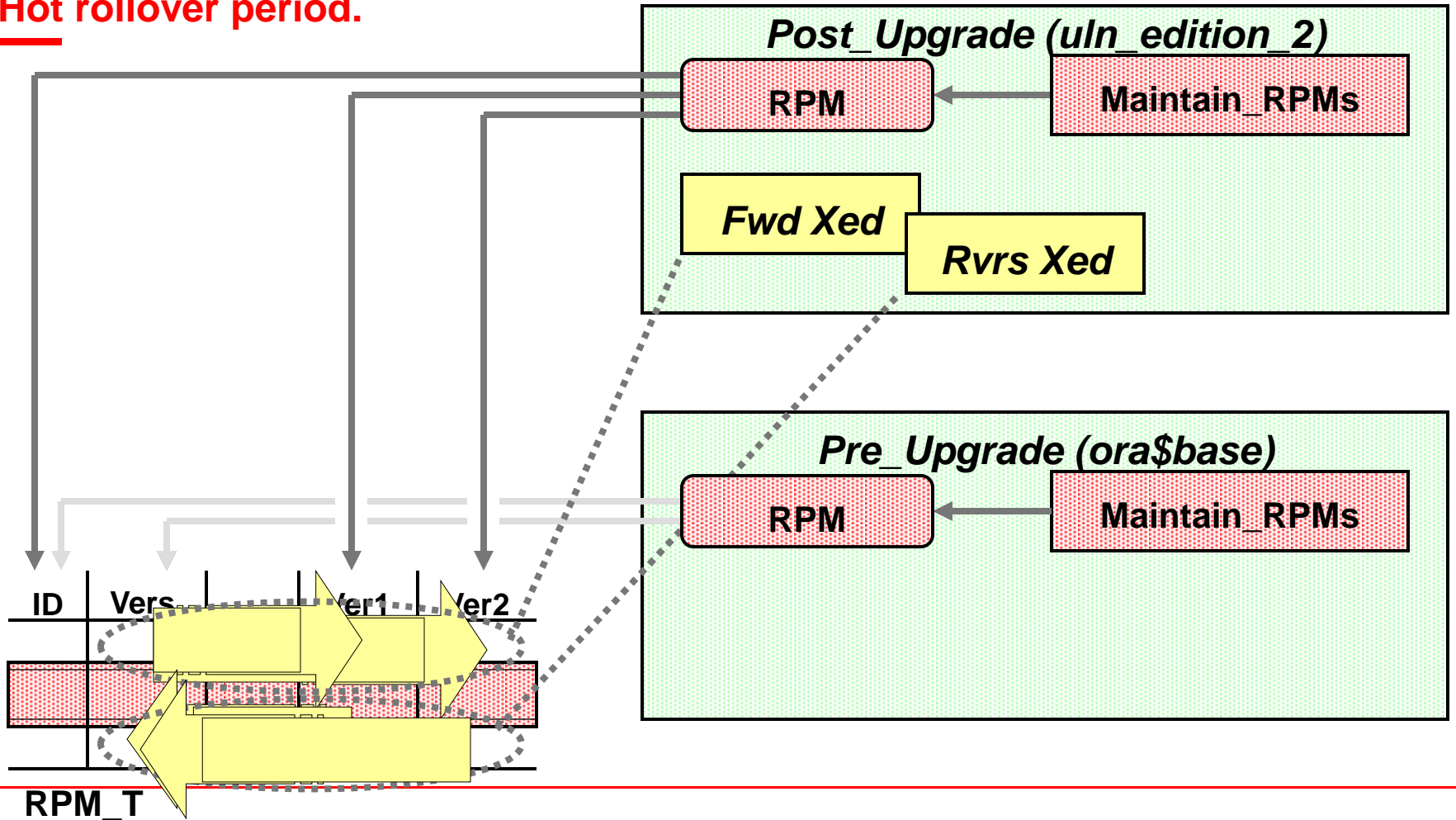
```
  2  dbms_parallel_execute.drop_task('update rpm_t');
```

```
  3  end;
```

```
  4  /
```

PL/SQL procedure successfully completed.

Hot rolover period.

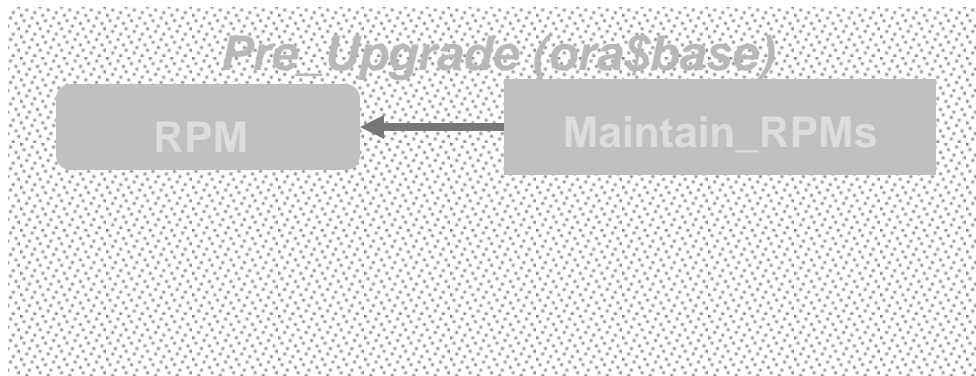
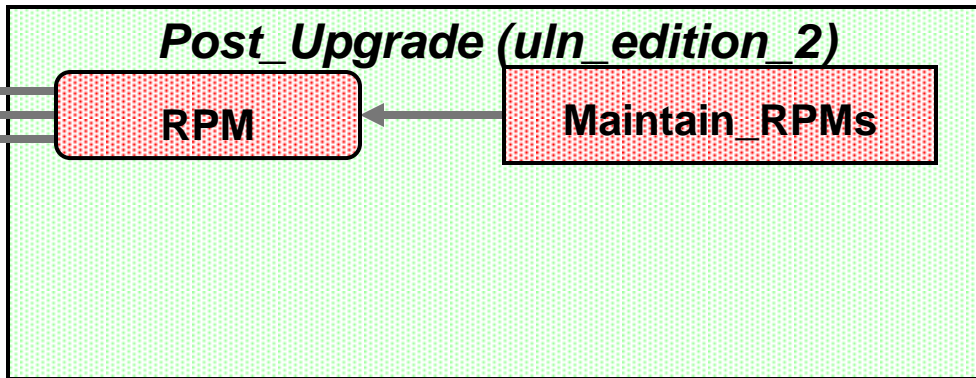


The *Pre_Upgrade* edition is retired.

The edition-based redefinition exercise is complete.

ID	Vers.	...	Ver1	Ver2

RPM_T



Move Your End-users to the New Edition

Set a logon trigger for sessions to use the new edition once they log on or reconnect

```
SQL> grant use on edition uln_edition_2 to public;  
Grant succeeded.
```

```
SQL> create or replace trigger set_edition_on_logon  
2 after logon on database  
3 begin  
4     dbms_session.set_edition_deferred( 'ULN_EDITION_2' );  
5 end;  
6 /
```

Trigger created.

Move Your End-users to the New Edition

Or ... if you are using a connection pool

SQL> begin

```
    dbms_epg.set_dad_attribute('APEX', 'database-edition',  
    ULN_EDITION_2');  
end; --If using the PL/SQL Embedded Gateway
```

In your dads.conf file: PlsqlDatabaseEdition*

--If using the Oracle Apache Http Server

Case study – continued

Rolling back the upgrade



Rolling back an online app upgrade

- Rolling back an application upgrade that's been installed classically is easy until you go live with the post-upgrade application
 - Presumably you took a backup at the start of the offline period and you just restore to that
- But once you go live with the post-upgrade application, you can't rollback to the pre-upgrade one
 - If you did this, you'd lose transactions made during the live use of the post-upgrade application
- It's just the same with online application upgrade
 - Without a hot rollover, your grace-period ends when you go live with the post-upgrade application

Rolling back an online app upgrade

- If you haven't gone live with the post-upgrade application
 - Drop the *Post_Upgrade* (child) edition (cascade)
 - Set any new replacement columns you created *unused*
 - At a convenient later time, recoup the space

EBR exercise vs offline upgrade: incremental extra effort

Proportional occurrence

editions

editioning views

forward crossedition triggers

reverse crossedition triggers

Very often	Change only editioned objects	✓			
Often	Make only additive table changes	✓	✓		
Less Often	Change only non-transaction tables	✓	✓		
Infrequent	Change the structure of transaction tables non-additively	✓	✓	✓	
<i>Very Seldom</i>	Support hot rollover	✓	✓	✓	✓

In Summary

- Online application upgrade is a high-availability sub-goal
- Edition-based redefinition helps make that possible
- Not for the ease of the developer or administrator – definitely for the convenience of the end-user
- If as-close-to-zero downtime is one of your company mandates, then you can easily be brought closer with EBR
- And best of all, it's available to any user of any version of Oracle 11gR2

Q&A

