

ORACLE 11GR2 EDITIONS YOUR FIRST NO DOWNTIME APPLICATION UPGRADE

Gary Gordbamer, GE Healthcare

PRESENTATION ABSTRACT

This presentation introduces the Oracle 11gR2 Editions feature. Implementing Editions enables the ability to do application upgrades with no downtime, including changing table structures, updating data, and doing this with multiple versions of applications simultaneously accessing the database. We will attempt to give a basic example, explain how to get started, and give references to additional documentation and next steps.

TABLE OF CONTENTS

Presentation Abstract	1
Table of Contents	1
Introduction	2
What is Edition Based Redefinition?	2
Application Changes	2
What can you do with editions?.....	2
Application upgrade process	2
Types of Database Changes.....	3
What’s in an Edition?.....	3
How does it work?.....	4
basic design rules for editions.....	5
Changing your Application for the first time.....	5
A basic example.....	5
Step 1 – Install the edition	6
Step 2 – Design your change	8
Step 3 – Install your change.....	8
preparing for the new edition	9
creating the new edition objects	10
Seeing the changes	11
Step 4 – Run both simultaneously	11
Step 5 – Move to the new edition.....	12
Step 6 – Optionally remove the old edition	12
Designing and coding for Editions	12
Next steps, references	13
References.....	13

INTRODUCTION

This paper is a simple introduction to the new feature of Oracle database 11gR2 called Edition Based Redefinition. The topic is large and actual expertise will require the experience of using it. Here we are just trying to provide some basic information to get that process started. I hope that this paper does get you interested in the topic and provides enough information for you to play a little bit with this new feature. The sooner you start, the sooner you will be an expert.

This paper is not intended to provide a cookbook to build a new application. Instead, it is intended to raise the general awareness in the development community.

WHAT IS EDITION BASED REDEFINITION?

Change is a natural part of the information professional's job. However, over the years, the methods to deal with change have, well changed. Databases are traditionally difficult to affect changes to. This is due to the central nature of its usage. Every user has to connect to a database directly or indirectly. Databases now connect to other databases, and there are still endless amounts of batch processing to load, unload, calculate and keep data current.

Availability of the database has become one of the largest design requirements. This can be segmented into two basic categories, planned downtime and unplanned downtime. Technology such as clustering, replication, and highly available hardware has helped to significantly reduce or eliminate unplanned downtime for databases. So now, we need to focus on planned downtime. A lot of progress has been made here as well.

Before we dig into the new Editions feature, lets try to understand what we can and cannot do with oracle 10g in the area of planned downtime. First, there is database maintenance. As of 10g it is possible to have very little downtime if your infrastructure and database are designed properly. Everything from minor patches, gathering statistics, re-organizing data or indexes, and even changing physical database storage can be done while users still access the system. Database upgrades still require a full outage, as do any change to the system Tablespace, or data dictionary.

Now I am not trying to say that running a no-downtime system is easy or takes little effort., often it is much easier to just take an outage and do the maintenance activity, but not every system will allow for this.

APPLICATION CHANGES

One of the largest areas of planned downtime is application changes, or changes to the database schema of your application. This could be something simple like adding a column to a table, or as complex as many structural changes and updated packages and procedures. Again lets start with 10g, you can already add a column, and redefine a column with no down time (although the application code may or may not like this). Indexes can be changed, but could have a performance impact to the code if you are not careful. Finally, stored procedures and packages could be updated, but again this could adversely affect the application if not done very carefully. Major structural changes to data are almost impossible to do while the application is being used. There is the ability to do some of this with third party applications, but even these are short of complete functionality.

Generally, we avoid application changes or updates because of the amount of effort and the risk of doing it.

So, what does 11g give you now? For database maintenance you can now do full database upgrades with no downtime. In fact, many patches can now be applied without even shutting down the database instance. In the area of application changes, a lot of progress has been made. It is possible to do some package, function and procedure changes while your application is running with no impact.

Edition Based Redefinition is a new feature included in the Oracle 11gR2 Enterprise Edition database product that allows the revolutionary ability to change your application while it is still being used. In fact, you can even have multiple versions of your application accessing the same data at the same time in the same database, in the same schema, with the same object name!

WHAT CAN YOU DO WITH EDITIONS?

Well in a simplistic manner, you can make just about any structural or programmatic change to the Oracle database components of your application with no downtime. You can also simultaneously use your old and new application code base. This requires a completely new mindset as to application change methodology.

APPLICATION UPGRADE PROCESS

Let's talk about current database application upgrade methodology. The normal process goes something like this:

1. Shutdown the application and lock out all users.
2. Backup the database.
3. Make changes to the database, structural changes and data changes.
4. Verify changes, test before releasing the application.
5. Startup the application, allow general user access.

This may not be an exact plan, but from a high level, it fits most applications that have used databases for the last 15-20 years.

So, what are the downsides to this process?

Well there is the extended amount of downtime for backup and making changes to the schema. If there are large amounts of data changes, the change window can be extremely long. As the age of the application increases, data volume goes up, and future data changes take longer.

Secondly, there is the process of testing or verifying the application before release. To do this you must bring up the application, but ensure that general usage is not available. This also limits your rollback or testing capability. Either you can do limited testing, in order to secure a good rollback plan, or you do extended testing which then eliminates the ability to easily back out. This revolves around the ability to insert new data into the changed application.

Finally, rollback is not a process anyone looks forward. Rolling back the upgrade will generally take just as long (if not longer) than the initial upgrade process.

So there are some down sides to this traditional approach, but realistically it was the only choice.

With the use of some 3rd party products in specific situations, there has been the ability to duplicate infrastructure for everything in production and with some specialized application coding, test your changes. Not historically a highly utilized process, however if you have used this method before then editions will feel somewhat familiar.

TYPES OF DATABASE CHANGES

Since we are interested in application changes, lets get a list of things that we can now do while the application is running.

Change Type	10g	11g	Technology
Add column to table	Yes	Yes	Alter table, Redefinition
Change column width or precision	Yes	Yes	Alter table, Redefinition
Change data type of column (comparable)	Yes	Yes	Alter table, Redefinition
Change data type of column (to any data type)	Yes	Yes	Redefinition
Add column(s) to a table with calculated value	Yes	Yes	Redefinition
Break one table into multiple tables	Yes	Yes	Redefinition
Combine multiple tables into one table	Yes	Yes	Redefinition
Simultaneously access “old” and “new” model at same time	No	Yes	Editions
Keep object names, but contain different logic	No	Yes	Editions
Completely change object, but keep name	No	Yes	Editions

Editions allows you to create a new set of views, synonyms, packages, triggers, procedures, and types that are identical in name, but only visible to a specific session. In fact, each session can be using a different edition. This sounds great; nevertheless, it requires your application to be designed to use these features.

WHAT'S IN AN EDITION?

With the new Oracle database 11gR2 you can now create an edition. This is a new object in the database, but I think of it more like a container or maybe a tag in a version control system. It allows you to collect non-structural (source) objects into a grouping. This includes all PL/SQL objects like procedures, functions and packages as well as views and synonyms. Structural objects like tables, indexes, LOB's, etc... are not edition-able.

Suddenly you think, well that doesn't help. In reality, this is a huge leap, and it's included in your license. It absolutely requires you to redesign your application, and re-think how you design your database schema. The emphasis with editions is the abstraction between what is stored (structural objects) and how the data is processed (source type objects).

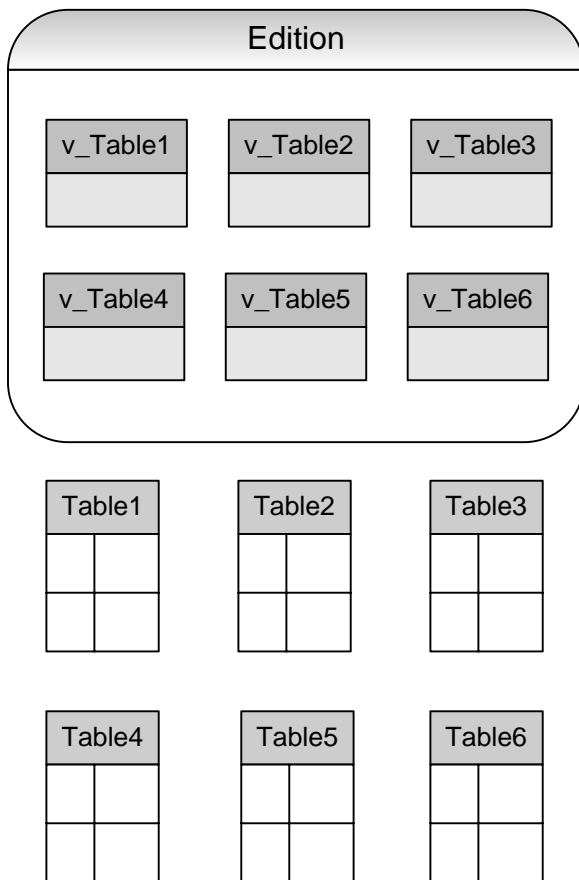


Figure 1 – Edition logical view

Figure 1 – Edition Logical view

Here we have an example of six tables.

Each of these six tables has an individual view.

The Edition then contains, or is a property of each view.

This allows you to create a new edition, and update the views in that new edition. No changes in the new edition are visible to the old edition.

Additional features not shown:

Table changes can be transparent to the views (depending on the change).

You can also have PL/SQL objects in the edition.

New cross-edition triggers can make transparent data changes between editions.

Only source type objects have the edition property.

Notes:

Your application should only reference the objects in the edition.

In the new edition, if no new definition of an object exists, the previous edition definition is used.

HOW DOES IT WORK?

An Edition is a new object type in the Oracle database. You create it just like any other object with a create statement and the given schema has to have the ability to create and or use the object. Every database has one edition by default, and everything starts out as part of this edition. You can think of this as the root edition, and is called “ora\$base”.

The schema that owns all the objects you want to use with editions has to have editions enabled.

```
ALTER USER MYAPP ENABLE EDITIONS
```

The application code will need to look at only views from this point forward. It's best to start naming your edition views like tables. Then your actual tables should have some name that denotes they should not be used. A simple example might be change the table PERSON to PERSON_X and create a view of type PERSON. A very simple code example would look like this:

```
ALTER TABLE PERSON RENAME TO PERSON_X;
CREATE VIEW PERSON (PERSON_ID, FIRST_NAME, LAST_NAME) AS
SELECT PERSON_ID, FIRST_NAME, LAST_NAME FROM PERSON_X;
```

Editions views have specific rules:

Each view can reference only a single table

Each column can only appear once

Must be a single query (no unions, etc...)

All rows must be represented, no WHERE, HAVING, GROUP BY etc....

ORDER BY clause is not allowed

The user must have the EDITION right, and the table and view have to be in the same schema

Users in the database can then be granted rights to see (or use) the edition or not.

BASIC DESIGN RULES FOR EDITIONS

Now beyond the views, just about any other source type object can be part of an edition. This includes packages, procedures, triggers, and functions. Therefore, from an application design standpoint you will want to rely on the source object types. So rely on views for all INSERT, UPDATE, DELETE and SELECT statements. For operations that hit multiple tables, consider using procedures. Consider wrapping complex calculations into functions. Finally, if you have a lot of PL/SQL code, wrap them into packages.

One note on security, if you are using Virtual Private Database (VPD) or Fine Grained Auditing, adjust your code as required. More on this is outlined in the whitepaper and manuals listed in the reference section.

CHANGING YOUR APPLICATION FOR THE FIRST TIME

As you can see to use editions, you will have to change your application. The process of this change will require you to take an outage. You have two options then to start using edition-based redefinition

1. Edition only upgrade, plan an application upgrade that will:
 - a. Alter the application schema to enable editions
 - b. Rename all the tables
 - c. Create views on-top of tables, with the old table name
 - d. Rebuild application triggers against the edition views
 - e. Optionally generate relational constraints between all the edition views
 - f. Recompile or rebuild any other PL/SQL
2. Edition with other application and database changes, plan an application upgrade that will:
 - a. Alter any physical tables, or add any new tables
 - b. Do any mass data updates
 - c. Implement any new triggers
 - d. Implement all the items of option 1
 - e. Install any PL/SQL changes

As this might lead you to believe, application testing will be imperative. In scenario one you will need to make sure that all application functionality is sound after implementing the needed editions based objects. For scenario two you will also need to do this, but in addition, you will have to make sure that all other changes are done in the correct sequence to get to the proper result. In reality, future updates with editions will be very similar to scenario two, but at that time, it will not be a new process for the application.

A BASIC EXAMPLE

So lets walk through a very basic example. For this I have the following schema called DRGG, and it has the following basic data model shown in Figure 2.

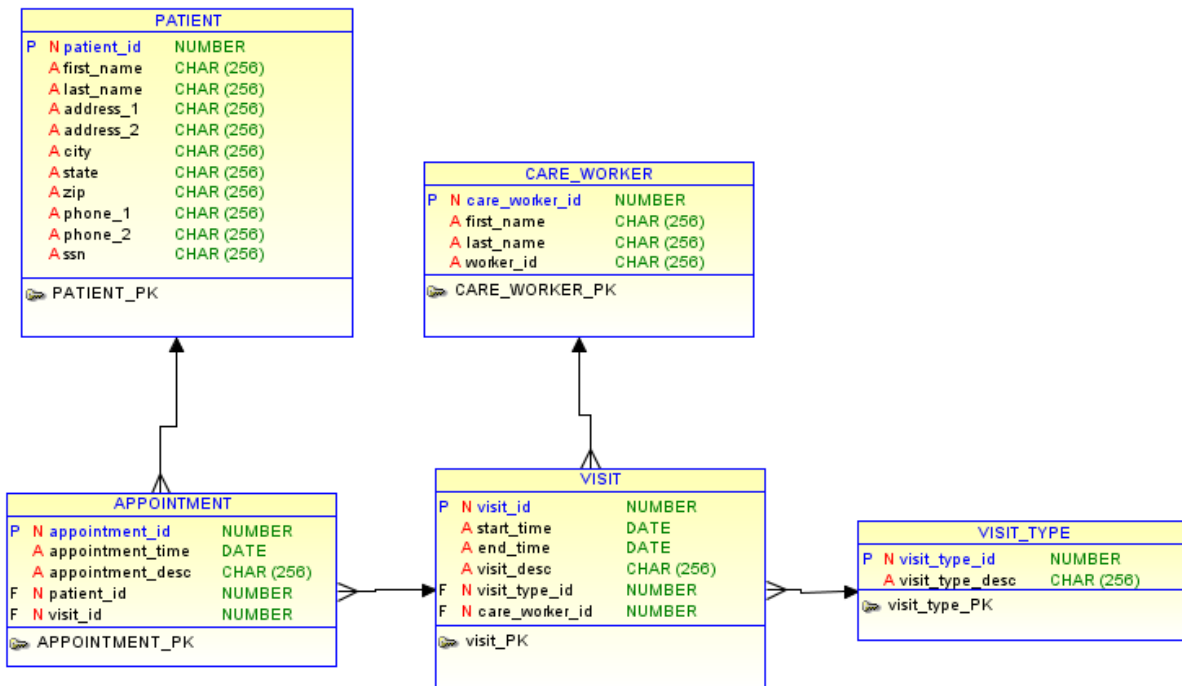


Figure 2 – V1 data model

Just to walk through each table quickly:

PATIENT – one record for each patient with basic contact information

APPOINTMENT – one record for each scheduled appointment

VISIT – one record for each interaction with a care worker during a given visit

CARE_WORKER – one record for each employee that works with patients

VISIT_TYPE – a lookup table that represents the type of interaction between the care worker and the patient

Note: Don't read into any of this, it's just very basic for the purpose of our example.

STEP 1 – INSTALL THE EDITION

Ok, so first we need to put in the needed objects to support editions. This starts by giving the schema owner the ability to use editions:

```
ALTER USER drgg ENABLE EDITIONS;
GRANT CREATE ANY EDITION TO drgg;
```

Now we can change the data model for our simple update. Here are a few lines of code showing the changes to one table (VISIT) for brevity, I'll exclude code for all the tables.

```
ALTER TABLE VISIT RENAME TO VISIT_X;
CREATE OR REPLACE VIEW VISIT
( VISIT_ID, START_TIME, END_TIME, VISIT_DESC,
  VISIT_TYPE_ID, CARE_WORKER_ID )
AS SELECT
  VISIT_ID, START_TIME, END_TIME, VISIT_DESC,
  VISIT_TYPE_ID, CARE_WORKER_ID
FROM VISIT_X ;
```

After making these changes our data model now looks like the data model in Figure 3.

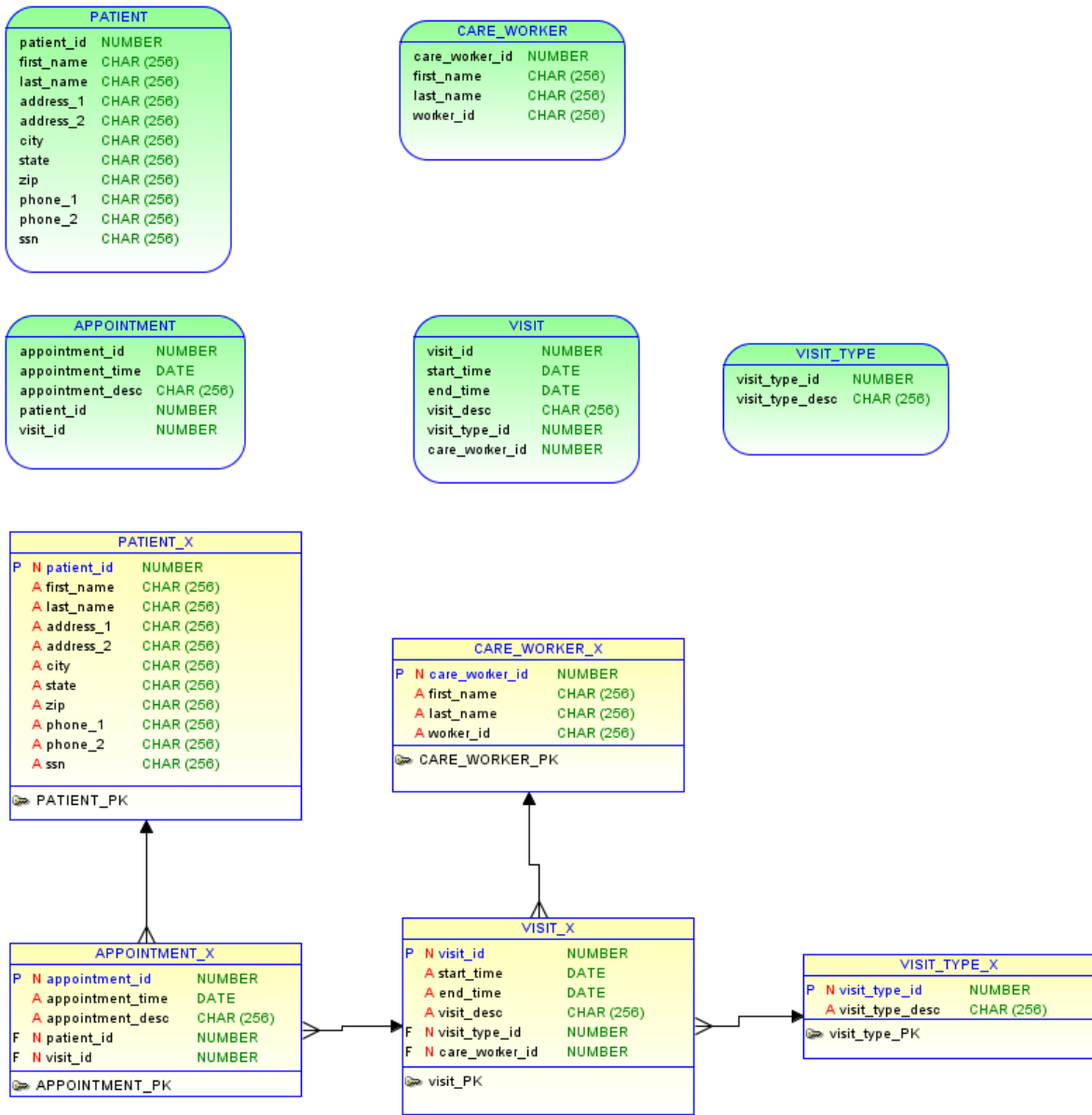


Figure 3 – V1 data model with editions

If you query the USER_OBJECTS table you will see a new column called EDITION_NAME. Looking at that for our example schema we now see (indexes and sequences removed for brevity):

OBJECT_NAME	OBJECT_TYPE	EDITION_NAME
APPOINTMENT_X	TABLE	
CARE_WORKER_X	TABLE	
PATIENT_X	TABLE	
VISIT_TYPE_X	TABLE	
VISIT_X	TABLE	
APPOINTMENT	VIEW	ORA\$BASE
CARE_WORKER	VIEW	ORA\$BASE
PATIENT	VIEW	ORA\$BASE
VISIT	VIEW	ORA\$BASE
VISIT_TYPE	VIEW	ORA\$BASE

A couple of things you should note. The indexes and primary key constraints for our tables were not renamed. We also could have done this step, but just note that it does not happen automatically. Second, you can see that only the views we created have an edition associated with them. We will discuss that new column a little bit more when we get to our new edition.

STEP 2 – DESIGN YOUR CHANGE

Ok, lets see what we can change about our application. A simple update for this day and age would be the ability to store more than two phone numbers. One method would be to add more columns, but lets be a little more flexible and add a new table so a given person could have unlimited contact methods. The problem is that all application code that looks at the PHONE_1 and PHONE_2 fields would need to be re-written, and application screens will have to be re-designed.

Lets use edition based redefinition to allow this with minimal application down time and enable both the old and new methods to be used simultaneously (though hopefully not endlessly).

Our design will need to add two new tables, two new edition based views, update one existing edition based view and implement a cross edition trigger to keep data in-sync while both the previous version and new version are in use.

Our version two (V2) data model looks something like the data model in Figure 4.

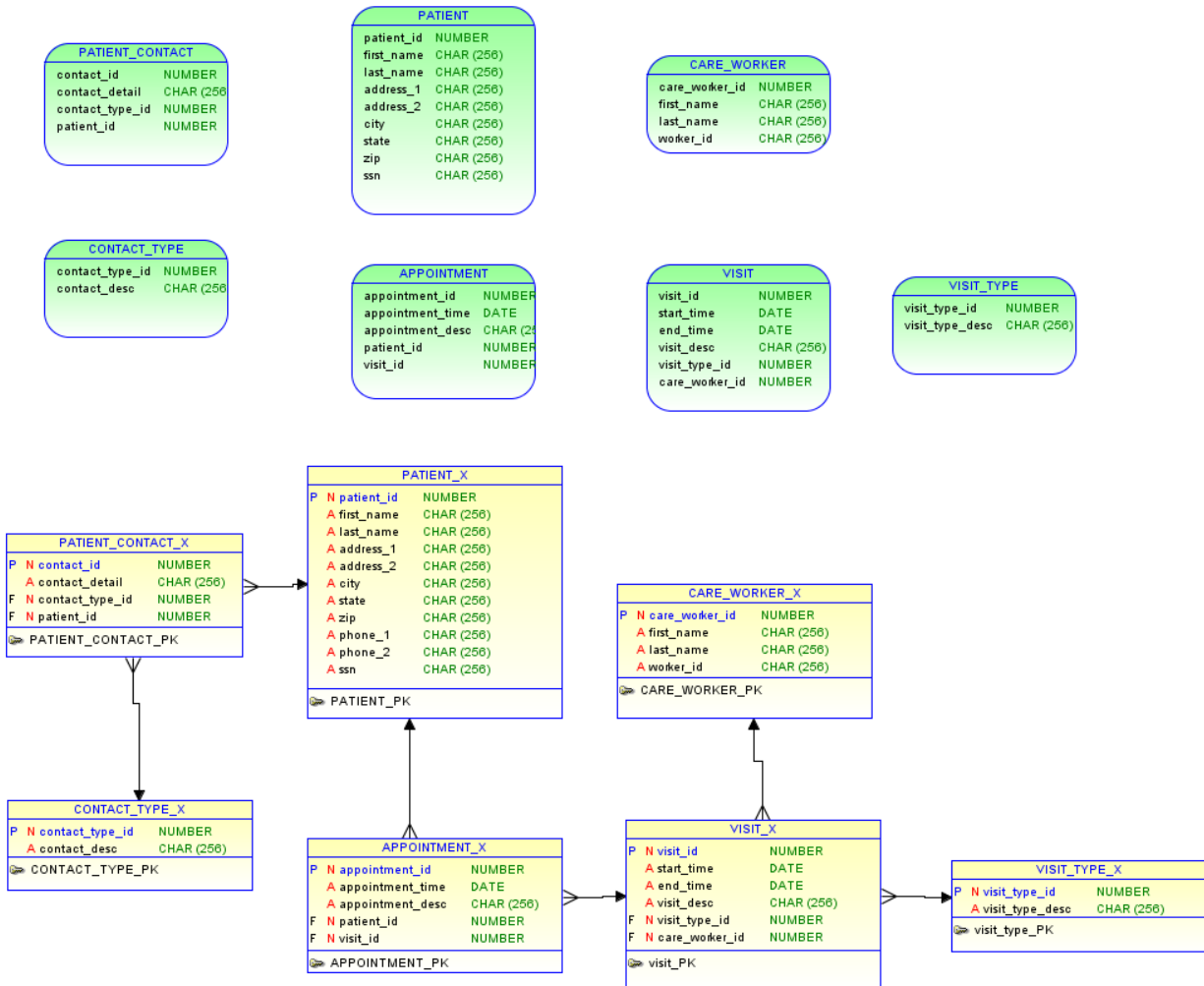


Figure 4 – V2 data model

STEP 3 – INSTALL YOUR CHANGE

All of the changes we are going to make can be done while the application is running. I'm going to step through them in some detail.

PREPARING FOR THE NEW EDITION

First we need to create a new edition, this is run as our schema owner DRGG since we granted it the proper rights:

```
CREATE EDITION drgg_v2;
```

Now we create our new tables, indexes, and constraints. I'll show just one table for brevity, but these changes have no direct affect on the edition:

```
CREATE TABLE PATIENT_CONTACT_X
  ( contact_id NUMBER NOT NULL ,
    contact_detail CHAR (256) ,
    contact_type_id NUMBER NOT NULL ,
    patient_id NUMBER NOT NULL )
  LOGGING ;
```

To keep data consistent we will use cross edition triggers. Since triggers are source objects, we need to make the forward moving trigger in the base edition. This will allow the V1 application to put data into the new table. An example trigger that updates PHONE_1 looks like this:

```
CREATE OR REPLACE TRIGGER patient_contact_edtr
  BEFORE INSERT OR UPDATE OR DELETE ON patient_x
  FOR EACH ROW
  FORWARD CROSSEDITION
DECLARE
  v_contact_id          patient_contact_x.contact_id%TYPE;

BEGIN
  IF INSERTING THEN
    IF :NEW.phone_1 IS NOT NULL THEN
      INSERT INTO patient_contact_x
        (contact_id, contact_detail, contact_type_id, patient_id)
      VALUES
        (contact_id_seq.NEXTVAL, :NEW.phone_1, '1', :NEW.patient_id);
    END IF;

  ELSIF UPDATING THEN

    IF NVL(:NEW.phone_1,'x') != NVL(:OLD.phone_1,'x') THEN
      BEGIN
        SELECT contact_id INTO v_contact_id
          FROM patient_contact_x
          WHERE patient_id = :NEW.patient_id AND contact_type_id = 1;

        UPDATE patient_contact_x
          SET contact_detail = :NEW.phone_1
          WHERE contact_id = v_contact_id;
      EXCEPTION
        WHEN NO_DATA_FOUND THEN
          DBMS_OUTPUT.PUT_LINE('No data to update.');
      END;
    END IF;

  ELSIF DELETING THEN
    BEGIN
      DELETE FROM patient_contact_x
        WHERE patient_id = :OLD.patient_id;
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No data to delete.');
    END;

  ELSE
    DBMS_OUTPUT.PUT_LINE('This code is not reachable.');
  END IF;
END PATIENT_CONTACT_EDTR;
```

Finally, we need to copy all the contact data from the PATIENT table to the PATIENT_CONTACT table. Let's assume the application is still running, and our new cross edition trigger is already putting new records in. So let's do a simple insert statement.

```
INSERT INTO patient_contact_x
  (contact_id, contact_detail, contact_type_id, patient_id)
VALUES
  SELECT contact_id_seq.NEXTVAL, p1.phone_1, '1', p1.patient_id
  FROM patient_x p1
  WHERE NOT EXISTS
    (SELECT 'X' FROM patient_contact_x p2
     WHERE p2.patient_id = p1.patient_id
     AND p2.contact_type_id = '1');
```

CREATING THE NEW EDITION OBJECTS

After our new data structures are in place, we need to create and update our views. These are directly part of the edition. The first command we run for this step is to change our current edition. Again, this is run as the application schema for our example:

```
ALTER SESSION SET EDITION = drgg_v2;
```

Note that if a different user created the edition, then the appropriate rights need to be granted to the application schema in order to use it. That statement would look like this:

```
GRANT USE ON EDITION drgg_v2 TO drgg;
```

Now we can create our new views, I'll show just one of them here:

```
CREATE OR REPLACE VIEW PATIENT_CONTACT
  ( CONTACT_ID, CONTACT_DETAIL, CONTACT_TYPE_ID,
    PATIENT_ID )
AS SELECT
  CONTACT_ID, CONTACT_DETAIL, CONTACT_TYPE_ID,
  PATIENT_ID
FROM PATIENT_CONTACT_X ;
```

We need to update the PATIENT view to remove the PHONE_1 and PHONE_2 columns.

```
CREATE OR REPLACE VIEW PATIENT
  ( PATIENT_ID, FIRST_NAME, LAST_NAME, ADDRESS_1, ADDRESS_2,
    CITY, STATE, ZIP, SSN )
AS SELECT
  PATIENT_ID, FIRST_NAME, LAST_NAME, ADDRESS_1, ADDRESS_2,
  CITY, STATE, ZIP, SSN
FROM PATIENT_X ;
```

Finally, we make the proper trigger updates to allow the new application and new table PATIENT_CONTACT_X to update the old fields in PATIENT in order for the V1 version to continue to work. We don't want the trigger on PATIENT_X to fire so we drop that in the new edition, and we add a 2nd trigger to the PATIENT_CONTACT_X table that will only fire in the new edition.

```
DROP TRIGGER patient_contact_edtr;
CREATE OR REPLACE TRIGGER patient_contact2_edtr
  BEFORE INSERT OR UPDATE OR DELETE ON patient_contact_x
  FOR EACH ROW
  REVERSE CROSS EDITION
BEGIN
  IF INSERTING OR UPDATING THEN
    IF :NEW.contact_type_id = '1' THEN
      UPDATE patient_x SET phone_1 = :NEW.contact_detail
      WHERE patient_id = :NEW.patient_id;
    END IF;
  ELSIF DELETING THEN
    IF :OLD.contact_type_id = '1' THEN
      UPDATE patient_x SET phone_1 = NULL
      WHERE patient_id = :NEW.patient_id;
    END IF;
  ELSE
    DBMS_OUTPUT.PUT_LINE('This code is not reachable.');
```

```
END patient_contact2_edtr;
```

Since the two triggers are specific to each edition, they fire only for sessions using that edition.

SEEING THE CHANGES

Lets go ahead, query the USER_OBJECTS table again, and see what we have. This first list is from the root edition:

```
ALTER SESSION SET EDITION = ora$base;
```

OBJECT_NAME	OBJECT_TYPE	EDITION_NAME
APPOINTMENT_X	TABLE	
CARE_WORKER_X	TABLE	
CONTACT_TYPE_X	TABLE	
PATIENT_CONTACT_X	TABLE	
PATIENT_X	TABLE	
VISIT_TYPE_X	TABLE	
VISIT_X	TABLE	
PATIENT_CONTACT_EDTR	TRIGGER	ORA\$BASE
APPOINTMENT	VIEW	ORA\$BASE
CARE_WORKER	VIEW	ORA\$BASE
PATIENT	VIEW	ORA\$BASE
VISIT	VIEW	ORA\$BASE
VISIT_TYPE	VIEW	ORA\$BASE

We see in our list of tables that our new objects are there, as well as our new trigger. To the old application, these are new table objects that are not referenced by the old application code. Then we have the new trigger, so the only thing that needs to be tested here is the new trigger. That can be unit tested with basic INSERT, UPDATE, and DELETE statements. Our impact V1 is now complete.

Lets look at our V2 schema, first we set our edition and then run the same query against the USER_OBJECTS:

```
ALTER SESSION SET EDITION = drgg_v2;
```

OBJECT_NAME	OBJECT_TYPE	EDITION_NAME
APPOINTMENT_X	TABLE	
CARE_WORKER_X	TABLE	
CONTACT_TYPE_X	TABLE	
PATIENT_CONTACT_X	TABLE	
PATIENT_X	TABLE	
VISIT_TYPE_X	TABLE	
VISIT_X	TABLE	
PATIENT_CONTACT2_EDTR	TRIGGER	DRGG_V2
APPOINTMENT	VIEW	ORA\$BASE
CARE_WORKER	VIEW	ORA\$BASE
CONTACT_TYPE	VIEW	DRGG_V2
PATIENT	VIEW	DRGG_V2
PATIENT_CONTACT	VIEW	DRGG_V2
VISIT	VIEW	ORA\$BASE
VISIT_TYPE	VIEW	ORA\$BASE

Here we can see the new views, and the new trigger. Also note the trigger we don't want to fire is not listed, since in the current edition it does not exist.

STEP 4 – RUN BOTH SIMULTANEOUSLY

Now we have the opportunity to run both the new application and old application at the same time. Your application has to set the edition at the session level when creating a connection. The old application (in this case) does not need any changes since everything is in the base edition. Only the new application needs to be updated.

Another interesting thing to note is that you can have multiple editions live at the same time. As you might guess, it could get extremely complex keeping track of multiple levels of cross edition triggers, views and table structures. But nonetheless, it is possible.

STEP 5 – MOVE TO THE NEW EDITION

At some point, you will want to move all your users to the new version. Depending on your application technology and design this may be as simple as giving everyone a new URL, or as complex as rolling out a new application to every desktop computer.

STEP 6 – OPTIONALLY REMOVE THE OLD EDITION

Once you have moved the entire user base to the new edition you may want to remove any access to the old edition. There are a few scenarios for this we will walk through at a high level. However, that being said, there may be no specific reason to remove previous editions. The most likely reason may be to prevent accidental usage, but outside of this, it should not interfere with normal database operations.

The first scenario is if the edition objects are in a separate schema from the application. Generally, this means that the application schema is locked and no direct access is allowed. In this case, you can revoke access to the edition from any other users in the database. Without these rights, the edition will be locked off from use. Though this is a perfectly good method, many applications are not designed to work like this.

The second option involves a little bit of back and forth work. Once our old edition is (V1 in our example) is no longer used, we could apply all the V2 changes to V1 (the root or ora\$base). Then once the editions are equivalent we could move the application sessions back to the root, and eventually drop the V2 edition. This is probably of little value, but would be a viable way to remove the possibility of accidentally using the wrong version.

Finally, we could programmatically make the old or root version unavailable. This could be through a logon trigger forcing the current edition to be set or by using the database setting:

```
ALTER DATABASE DEFAULT_EDITION = drgg_v2;
```

This setting will persist across database restarts, and affects all nodes of a RAC cluster. This also grants use of this edition to PUBLIC.

Another scenario we did not discuss is if there are multiple editions, and you would like to remove one that is not the current edition, and is not the root edition. You can do this by dropping the edition. If the edition contains objects, you will have to use the cascade option, or remove those edition specific objects before dropping.

DESIGNING AND CODING FOR EDITIONS

This is where the major work comes in. Since application design tends to be measured in years, not days, the adoption of edition-based redefinition should not be expected over night. The main hurdle will be tying your application to this (or any) technology. If it is already an Oracle database centric application then possibly, that choice is easy. I guess the other question is if this level of sophistication will show up in other RDBMS products. Also very possible, which would drive a large amount of adoption.

If at this time you have decided to use the Oracle RDBMS feature set for your application, you will then have to design for editions. The primary feature is to move away from physical object references in your code. Clearly source type objects are going to have the ability to transition across editions. This by no means dismisses the need for physical table design, or proper data modeling for performance and capacity purposes. It does introduce a new layer of design though. Depending on your organization or experiences, this level of design may exist today, or may not.

The level of design depth, or Oracle feature usage becomes an interesting discussion. Do you for example, rely on edition views to physical objects and then encapsulate all the logic in a your application layer? Conversely, would there be benefit to putting more application logic in database-stored code (PL/SQL) that can then transition from edition to edition and may require little or no application layer change. There is no single answer to these design questions. Consideration must be made for both options, and at a minimum, the decision should be discussed and made, not assumed. Often application design issues arise when one assumes a given path without any detail on why that path was chosen, or what the benefits or detractors are for that path. It would be somewhat disastrous for database coders to make edition-based updates while application (perhaps Java) coders are assuming they control the level of change in the application code. It's clear to me that this new level of abstraction requires more transparency in the design rather than less.

Production change to systems will have to go through a transformation. One of the biggest benefits of edition-based redefinition is the ability to run multiple versions of an application at the same time. This functionality has been almost non-existent in the past. Almost all of our history of application change is around the no-going-back method of data and

application changes. Once a specific threshold of application usage or change is in place, that new version has to be used. Our new paradigm says that given the right design, any change could be backed out. In fact, any change could be specific to a user set and only that user set would see it. Risk analysis, impact of change, and organizational rigor of change will have to consider these new rules. This will take a lot of education in the IT and business community, as well as some brave IT professionals to be the first to pave the way. Ultimately success stories will prove this new technology out, much like clustering with RAC, consolidated storage with SAN and even something as ubiquitous as the Internet over a cell phone.

NEXT STEPS, REFERENCES

Ok, so now you are probably interested in what you could do with edition based redefinition, and wondering where to start. I hope that the basic example has given you a few ideas of where to start. The biggest step will be modifying your application to take advantage of editions. Beyond that, it really will start to affect the entire development methodology. In general, database schemas (or data models) have been sacred objects. Once the model is set, it can be very difficult to change or at minimum a high-risk discussion around the process of changing. With Oracle's new feature, that discussion can be changed, as can the database model. Admittedly, it might be 10 years before the industry is at that level of maturity. The more a technology is used, the more mature it will become and the more the industry will drive towards it.

REFERENCES

Oracle Database Advanced Application Developer's Guide, 11g Release 2 (11.2), Chapter 19, Oracle Corporation © 1996, 2009. Part Number E10471-04

"Edition-Based Redefinition", Byron Llewellyn, Oracle Corporation July 2009