Online Application Upgrade Using Edition-Based Redefinition

Alan Choi

PL/SQL Database Sever Technology
Oracle
alan.choi@oracle.com

Abstract

This paper describes Edition-Based Redefinition (EBR) in the Oracle database — a novel technology and methodology to build database application patches so that installation of these patches does not require application downtime. It discusses the challenge of zero-downtime application patch installation and shows how EBR can meet the challenge. Customer experience shows that EBR provides online application upgrade in several industries.

Categories and Subject Descriptors H.2.m [**Information System**]: Database Management – Miscellaneous D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – Version control

General Terms: Management, Languages.

Keywords: database, availability, application upgrade, database application development

1. Introduction

Many database applications have a very high availability (HA) requirement. Examples include electrical and telecommunication utility management systems. Those systems must be available all the time to manage the systems; for electrical utility system, it includes adjusting production, or energy trading. Another example is global customer support system for companies like Oracle. A support system, with database application as the backend, must be available to log for each incoming request; calls come 24/7 for global companies. Manufacturers rely on database applications for inventory accounting and have large-scale factories running round the clock. Taking applications offline shuts down assembly lines. Firms lose millions from production and shipment delays. Database applications have become an integral part of business; there are business needs from many industries for HA software systems.

HA means not only that the application should run in the event of unplanned hardware failure, but that it should also be available during planned changes to the software. Although hardware failure is relatively rare, many database systems employ replication technology to avoid or minimize hardware downtime. But database application patches or upgrades occur far more frequently than hardware failure; some application vendors issue hundreds of patches per week. Yet there has not been a satisfactory way to install upgrades without taking downtime. Online application upgrade is a critical but unsolved part of HA requirement.

When an application patch modifies its database components, the changes can be classified into two main categories: physical changes and changing the meaning of the database objects. For physical changes, Oracle has developed various technologies to make changes online. However, changing the meaning of database objects without causing disruption to the running application has not been addressed. This is the problem that we are looking at: online application upgrade of database components[1].

In Oracle Database 11g Release2, we introduce revolutionary new capabilities that solve the problem. This paper presents the new technologies and how they can be used to achieve online application upgrade.

2. Background

There are two key challenges to the problem. First, the installation of the patch into the production database must not perturb live users of the pre-upgrade application. Second, pre-upgrade application and post-upgrade application must be available at the same time: that is, there must be *hot rollover*.

Database objects can be broadly divided into two categories: code objects such as stored procedures and data objects such as tables. While a code object is only relevant to a particular version of an application, the essence of the data (and therefore data stored by data objects) usually spans across multiple versions of the application even though table structure can change. Given these observations, there are three requirements:

1. A set of code objects must be changed in an isolated environment.
2. Common data must be visible and consistent in both environments.
3. Non-common data must be transformed and reflected in the other environment.

If these requirements are achieved, the end user can enjoy hot rollover without downtime.

It has been an on-going effort to tackle this problem at Oracle. Various technologies have been developed to minimize the disturbance caused by changing the database component of an application.

For physical changes, a number of Data Definition Languages (DDLs) has been made transparent. For example, online index creation and rebuild [1] and some ALTER TABLE DDLs have been made non-blocking so that insert/update/delete (DML) is-

HotSWUp'09 October 25, 2009, Orlando, Florida, USA.

[1] It should be noted that this is not upgrading the Oracle Database from release 10.2 to 11.2 say.

sued by the running application won't be blocked [1]. This allows online database maintenance.

For logical changes, fine grained dependency [2] prevents unnecessary invalidation of objects if a new signature remains compatible with the old one: for example, adding a new column to a table or adding a new subprogram to a package specification need not invalidate the dependants. Further, online table redefinition [3] provides a mechanism to make table structure modifications without significantly affecting the availability of the table.

These technologies improved the availability during modification of a single object. Although they laid the foundation of our work, none of them addressed the need to modify a set of objects in concert, the norm for patch installation.

There are two notable approaches to online application upgrade: Schema Clone and log-based replication. Schema Clone will split data objects into one schema and stored procedures into another. During patch installation, the whole stored procedure schema will be cloned into a new schema and changes will be made in the new schema. However, this scheme has several drawbacks. First, the complete clone is time consuming and takes space because every object cloned will be recompiled and stored separately. Second, this scheme requires all objects to be stored in a single schema. Schema separation and security is lost.

The second approach is log-based replication. Golden Gate software has successfully used log-based replication to perform online application upgrade for Seibel [4]. This approach requires a replica database. All the changes will be made in the replicated database, which is completely isolated from the production one. Therefore, unlike Schema Clone, it does not have the single-schema restriction. Data changes in the production are captured by mining the transaction (redo) log and then propagated to the replica and *vice versa* for hot rollover deployments. The heterogeneous nature allows this approach not only to upgrade database application, but it can also be used to minimize downtime during database upgrade [5]. However, the flexibility comes with a cost. First, requiring a replica means doubling resources. The second drawback is the lack of transactional synchronization between production and replica because synchronization happens only after commit. Therefore, complex conflict resolution schemes are required during hot rollover.

In Oracle 11.2, we introduce Edition-Based Redefinition (EBR) to solve the zero downtime application upgrade problem. With EBR, we achieve the goal using a single database and can span multiple schemas, eliminating the drawbacks of the two existing approaches. In the rest of the paper, we first go over the new technologies briefly, present a sample case study, and then discuss the technologies in detail.

3. Edition-Based Redefinition

In Oracle 11.2, we bring three new features to help achieve online application upgrade: the *edition*, the *editioning view*, and the *crossedition trigger*. With an *edition*, the same object can occur many times in one database; each occurrence is in a different edition. Thus, the *edition* provides an isolation mechanism: object changes in one edition will not affect the object in other editions. Data, which is private to the new edition, is stored in new columns or new tables, an *editioning view* presents logical projections (that is, a subset of data) of the underlying table to each edition: a different projection of a table into each edition to allow each to see just its own columns. All application code should be targeting logical schema projected by editioing views, instead of the underlying table[2]. The post-upgrade application can make data changes safely by writing only to the new columns or new tables not seen by the old edition. Uncommon data between the old and the new edition is synchronized by a *crossedition trigger*. Whenever there are data changes in the old edition's column, *crossedition triggers* will propagate the changes into the new edition's column, and *vice versa*. The following case study will demonstrate how these three technologies can be used to produce and install a patch online.

4. Case Study

The HR sample schema, as shipped by Oracle Corp, has a single column for phone number. For globalization purpose, the new version of the HR application will have separate country code and number within the country. A few PL/SQL stored procedures and views will be updated to support the new business logic related to country code while most of the remaining ones will remain unchanged because primary functionalities remain the same: adding, modifying, and querying personal information. The characteristic of this patch — some schema changes, some code changes, reusing existing data — is a good representation of a typical patch. We will show how this upgrade can be done without forcing any downtime.

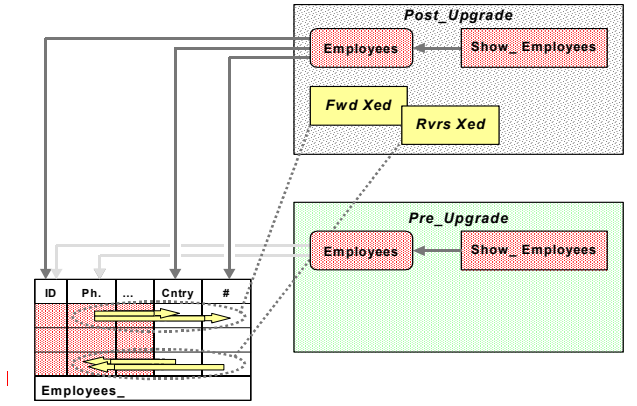

Figure 0. upgrade installation.

4.1 Installing An Upgrade

Before the upgrade begins, the existing application is using the pre-upgrade edition, as shown in figure 1. It holds the stored procedure Show_Employees and the editioning view Employees. The Employees editioning view only selects columns relevant to the current edition from the Employee_ table, which is visible from across all editions. In the pre-upgrade version, only ID and Ph are the relevant columns. When the patch installation begins, the post-upgrade edition will be created to host the new version of the editioning view Employees and the new Show_Employees stored procedure. Because the installation happens in the post-upgrade edition, it won't affect the objects in the pre-upgrade edition. The

[2] There is a readying step to use EBR. It must first create an editioning view for each table, and have all application code referring to the editioning views. Refer to section 5.2 for details.

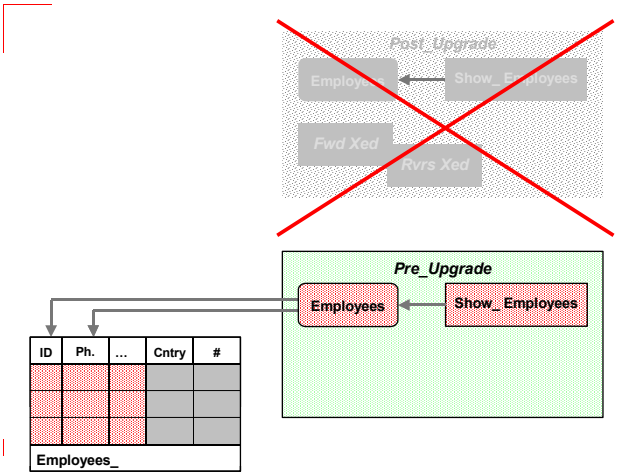

Figure 1. Rolling back an upgrade.

patch will also add two new columns to the table Employees. Because no object, other than the editioning view, is referring to the table directly, adding new columns will not invalidate any object. The pre-upgrade application, therefore, can continue to operate undisturbed.

FwdXed and RvrsXed are the *crossedition triggers* for data synchronization between the pre-upgrade and post-upgrade application. When the pre-upgrade application modifies the old phone number column, FwdXed will reflect the corresponding changes in the new country code and new phone number columns within the same transaction. On the other hand, when the post-upgrade application modifies the country code or the new phone number columns, RvsXed will propagate the corresponding changes back to the old phone number column. Both pre-upgrade and post-upgrade applications can run simultaneously, sharing a consistent set of data across both editions. End user sessions can enjoy hot rollover. Once all the end-use sessions have migrated to the post-upgrade application, the pre-upgrade application can then be retired.

4.2 Rolling Back An Upgrade

If the post-upgrade application has not gone live yet, EBR provides an easy way to rollback the upgrade. It involves two steps: dropping the *post_upgrade* edition and setting any new columns or tables to *unused*. Optionally, the unused columns/tables space can be recovered later. None of the steps affect the availability of the pre-upgrade application.

5. Edition-Based Redefinition In Depth

We have demonstrated how EBR enabled online patch installation. We now move forward to discuss EBR in depth.

5.1 Edition

The primary goal for Edition is to provide an isolated environment so that a set of code objects, such as PL/SQL, views and trigger, can be changed in concert. However, not all object types should be private to an edition. Data objects, such as tables, must be visible and identical across all editions for data sharing purpose. In terms of performance, the creation of an edition must be very fast but in terms of semantics, it must provide a complete copy of the pre-upgrade objects. This criterion is critical because the overhead must be small and the upgrade cost should reflect the size of the upgrade compared to the overall application size.

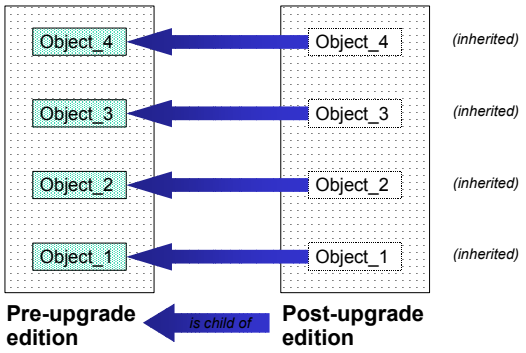

Figure 2. Initial state of a new edition.

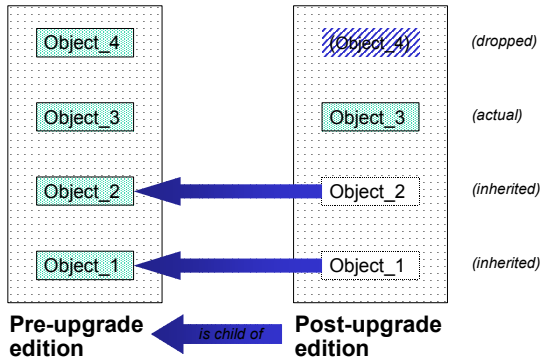

Figure 3. Inherited, actual, and dropped Object in an Edition.

With these objectives in mind, we have designed Edition with these semantics:

1. There are two classes of object types: *editionable types* and *non-editionable types*. An object whose type is editionable can have its own private occurrence of *the same* object. An object whose type is non-editionable is identical and visible across all editions.

2. When a new edition is created, it will have a complete snapshot of all the editionable objects from the latest edition at edition creation time.

3. Changes to editioned objects are private to the edition in which the changes happened.

We choose an inheritance model as our implementation, and this model gives us some nice performance characteristics. Figure 2 and 3 demonstrate how the inheritance model works.

As shown in figure 2, when the post-upgrade edition is created, for each object in the pre-upgrade edition, a placeholder pointing to the corresponding object in the pre-upgrade edition will be created. An object denoted by such a placeholder is called an *inherited* object. Creating inherited objects is an extremely low cost operation. It does not require compilation nor does it copy metadata. The low cost allows all inherited objects to be created

in a single transaction – in effect, a snapshot of the pre-upgrade edition.

Figure 3 shows that the patch modifies object_3 and drops object_4 in the post-upgrade edition, but leaves object_1 and object_2 unchanged. Because object_3 is modified, it becomes *actual* and is compiled with all its metadata written to disk. The cost of this action would be the same as if a brand new object is created. The drop operation on object_4 simply requires deleting the placeholder. Because there is no metadata, this drop operation is even more efficient than dropping an actual object.

Execution of an object always happens inside a PL/SQL block or a SQL statement which requires compilation. During compilation, the name of the object will be resolved and all its data will be loaded. For an inherited object, all its metadata comes from the object that it is pointing to and therefore name resolution performs one level of re-direction. The re-direction cost is negligible, and it only occurs during compilation. Once the PL/SQL or SQL is compiled, executing an inherited object is no different from executing an actual object. Compilation of PL/SQL or SQL in a live production system is rare. Most of the compilation usually happens during system cold start; the compiled PL/SQL or SQL should remain in memory afterwards.

5.2 Editioning View

A table has two responsibilities: storing data and presenting a schema. The commonality of the data must be shared by both pre-upgrade and post-upgrade applications. However, the non-common data and the table schema must be private to each edition. The *editioning view* solves this problem. A table is now responsible for storing all the data for all editions while the editioning view is responsible for providing the correct subset of data by projecting a subset of columns from the underlying table. An editioning view is a special kind of view that may only project and rename columns. Under EBR, application code should always refer to editioning views instead of the underlying tables. Editioning views shield the application code from any irrelevant changes to the underlying table.

To use EBR, an existing application has to cover all tables with editioning views. Only editioning views and crossedition triggers should refer to the base tables directly. Given that requirement, the editioning view has these characteristics:

1. Replacing tables with editioning views should be a mechanical task (that is, little human effort).
2. There should be no further application code changes (because all operations directed to the table now see the view, the view must support all these operations).
3. There should be no performance degradation.

A regular view cannot satisfy the second and third objectives because of some subtle restrictions, although it can satisfy the first one, by renaming all table X to X' and then create a view X as select * from X'. Triggers, for example, can only be created on a table and can't be created on a view [1][6]. In terms of performance, Oracle's index hint only accepts table names, not views. It is not uncommon for a highly tuned application to use this hint to squeeze extra performance. Simply replacing a table with a regular view would nullify the hint.

Editioning views solve both of these problems: triggers can be defined on them and hints can refer to them. After replacing tables with editioning views, application code will require no further changes.

5.3 Crossedition Trigger

If the upgrade needs to change the structure of a table that stores transactional data, then the installation of values into the replacement columns must keep pace with these changes in the old columns, and *vice versa*. A trigger has two key properties that make it an ideal synchronizer: transactionalilty and low overhead. Being transactional guarantees that values in the old and new columns are always in sync: either both are committed or both are aborted. Conflict is detected at the transaction level, which the application must have existing mechanisms to manage. We have measured the trigger overhead by creating an empty per-row trigger (which does nothing in the trigger body) on all the tables in TPCC benchmark. The experiment shows that the overall throughput drops 3%, when compared to a regular TPCC setup, an acceptable price for a zero-downtime patch installation.

We introduced two kinds of triggers: *forward crossedition trigger* and *reverse crossedition trigger* with special firing rules to help developers to code these transformation triggers. A forward crossedition trigger is responsible for bringing old columns values to the replacement column and will fire in response to DML in the pre-upgrade edition. A reverse crossedition trigger is responsible for bringing replacement column values back to the old columns and will fire in response to DML in the post-upgrade edition.

We also introduced trigger-firing order, essential for a correct transformation. Consider the case where an application triggers in the pre-upgrade application modifies a column value during insert. That is the value that should be propagated. Therefore, crossedition triggers should fire after all other application triggers have been fired. Placing crossedition triggers as the last ones to fire guarantees that the value propagated is the *final* value of the DML. Because trigger-firing order could be useful for general use, we've extended this feature for regular triggers.

With a special firing rule and user-defined firing order, crossedition triggers provide an efficient and manageable way to synchronize data between the pre-upgrade and post-upgrade application.

6. Conclusion

Online application upgrade is not a niche feature for a few companies. From the business perspective, taking critical software components offline means stopping the entire business flow, which translates to loss of revenue, loss of customer satisfaction, or in the most severe case, a violation of contract. The demand is real and urgent. Achieving it gives business a strong competitive edge.

We've demonstrated our technology, edition-based redefinition, can be used to upgrade the database component of a database application without causing any disruption to the end-user. Unlike other HA features, which are used by the DBA at the deployed site, this technology has to be understood by the developers of the database application when preparing the application for online upgrade and when implementing an upgrade script. Only then can the application be upgraded without losing availability.

At the time of this paper, Oracle 11.2 has just been released for a month. Therefore, no customers are using it in production. However, customers who participate in the 11.2 Beta Program tested it extensively, declared it fit for its purpose, and are intending to use EBR in production at the earliest opportunity. Our customers Betfair, which operates a 24/7 global business, and IFS, which provides applications for round-the-clock global business, hope to enable online application through EBR. These customers show the real world demand for online application upgrade and validate EBR.

References

[1] Oracle Database SQL Language Reference, 11*g*Release 2 (11.2)

[2] Oracle Database New Features Guide 11*g*Release 1(11.1)

[3] Oracle Database Administrator's Guide 11*g*Release2 1(11.2)

[4] Doug Reid: Upgrading Oracle Siebel CRM with zero downtime: best practices, March 2009

[5] Database Rolling Upgrade Using Data Guard SQL Apply, Oracle Database 11g and 10gR2

[6] ISO/IS Database Language SQL – Part 1: SQL/Framework, section 4.6.6.4, July 1999