"Unlocking" Oracle Developer with Rdb

Jeffrey S. Haidet

Cheryl P. Jalbert

JCC Consulting, Inc. Granville OH 43023

Abstract

Oracle Forms was originally built to work natively with the Oracle database server using the native Oracle Call Interface. It also talks to other database servers via ODBC. When Rdb gained the OCI interface this gave Rdb users the opportunity to build Oracle Developer applications using the more-efficient OCI interface as well.

However, Rdb is definitively not Oracle(n). Accordingly, one must significantly alter the design of Oracle Developer applications which would interface with Rdb instead of traditional Oracle. Some areas where these designs must be altered include:

- Default transactions
- Holdable cursors
- Controlling transactions within Developer
- Controlling transactions in a multi-form Developer application
- Automatic deadlock control
- Controlling stale read-only transactions

This presentation will include real world examples of how we address these issues at JCC. This methodology includes a practical object-based model and has been applied to applications involved with hundreds of forms without editing hundreds of files!

Default Transactions

- Closest Rdb transaction state to the Oracle model is ‘set transaction read write isolation level read committed’
- Read committed transactions still hold locks
- In a true application, inserts/updates and queries must all be supported

Default Transactions

- Rdb automatically starts an implicit transaction if a transaction has not been started
  - Due to default transaction state, this is a Read Write transaction
- Implications
  - Locks can be held for long periods of time
  - Read Write transactions opened automatically
    - Unless user (or the application) does something, these transaction can remain for long periods of time
- Is there a way around these issues?

Requirements

- To take advantage of these recommendations, Quiet Commit must be turned on
  - Quiet Commit was added in Rdb V7.0, but wasn't documented until the Rdb V7.0.2.1 release notes.
  - Enable Quiet Commit in the SQL initialization script for the service

```
SQL> commit;
%SQL-F-NO_TXNOUT, No transaction outstanding
SQL> rollback;
%SQL-F-NO_TXNOUT, No transaction outstanding
SQL>
SQL> set quiet commit ’on’;
SQL> commit;
SQL> rollback;
```

Holdable Cursors

- This is the default cursor state when using OCI
- Holdable cursors keep cursors open across transactions
  - Open Cursor
  - Fetch from cursor (fetches row 1)
  - Rollback
  - Fetch from cursor (fetches row 2)
  - etc

Leveraging Developer Built-Ins to Leverage Rdb

- FORMS_DDL (Forms)
  - Unrestricted Oracle built-in function that takes as a parameter one string and sends this string into the database to be compiled and executed
  - Example
    - Forms_ddl (‘my string to pass’);
- SRW.DO_SQL (Reports)
  - Built in package procedure that is basically the Oracle Reports equivalent to FORMS_DDL
  - Example
    - Srw.do_sql (‘my string to pass’);

FORMS_DDL and SRW.DO_SQL Failures

- What happens if the routines fail to execute the statement successfully?
  - Client-side exception FORM_SUCCESS set to ‘false’
  - The exception must be handled by the application

```
Begin
  forms_ddl (‘set transaction read only’);
  if not(forms_success)
  then
        < Process exception here >
  end if;
End;
```

Controlling Transactions inside Developer

- Can the Rdb transaction model be controlled from within Developer?
  - Things to consider
    - Implicit transactions
    - Holdable cursors
    - Default transaction state
    - FORMS_DDL/SRW.DO_SQL
    - “Stale” data

Toggling Transaction State Within a Form or Report

- Leverage the Oracle built-ins SRW.DO_SQL and FORMS_DDL to alter your current transaction state
  - Example

```
Begin /* Forms example */
   forms_ddl ('rollback');
   forms_ddl ('set transaction read only');
   <…>
   forms_ddl ('commit');
End;
Begin /* Reports Example */
   srw.do_sql ('rollback');
    srw.do_sql ('set transaction read only');
   <…>
    srw.do_sql ('commit');
End;
```

Implications on Transaction State Toggling

- If we programmatically change the transaction states with FORMS_DDL or SRW.DO_SQL what implications will this have on our application?
  - Since we are using holdable cursors, nothing.
  - Example:

    Form Opens

    Use FORMS_DDL/SRW.DO_SQL – ‘set transaction read only’

    Query issued

        Cursor Opened

        Records Fetched (1 –20 returned to application)

    Use FORMS_DDL/SRW.DO_SQL – ‘rollback’

    Query Continued

        Records Fetched (21 – n returned to application)

    Etc.

Toggling Transaction States

- The Good News: By using the built-in routines to control your transactions, you are overriding the default transaction state for that OCI server
- The Bad News: By using the built-in routines to control your transactions, you are overriding the default transaction state for that OCI server
  - Increased overhead due to compilation of command by database
- In an application that supports both writes and queries, you must now be certain of your current transaction state.
  - Otherwise the ORA-01456 – attempting to update during a read only transaction exception might be raised

JCC Standard for Toggling Transactions

- JCC develops forms and reports to read all data in read only transactions, and switch to read write transactions long enough to do our database writes
  - PRE-QUERY trigger (forms)
    - Forms_ddl (‘rollback’) – get rid of any implicit transactions that may be started
    - Forms_ddl (‘set transaction read only’);
    - Execute the query
  - BEFORE-REPORT trigger (reports)
    - Srw.do_sql (‘rollback’); -- get rid of any implicit transactions that may be started
    - Srw.do_sql (‘set transaction read only’)
    - Execute the query
- NOTE: If Quiet Commit is not enabled, exceptions will be raised

Multi-Form Applications

- In a true Windows style multi-form application, we don’t know what the user is going to do next
  - Could request menu functionality
  - Could issue a toolbar command
  - Could execute/enter a query
  - Etc.
- So how do we control the transactional model?

Multi-Form Transaction Control

- The JCC methodology for controlling transactions in a multi-form application is as follows
  - Control the transactional model in the following Developer events
    - WHEN-NEW-FORM-INSTANCE trigger
    - PRE-QUERY trigger
    - POST-QUERY trigger
    - KEY-COMMIT trigger
    - POST-FORMS-COMMIT
    - POST-DATABASE-COMMIT

WHEN-NEW-FORM-INSTANCE

- This fires once for each new instance of a form
- We want to put the form in a read only transaction while we wait for instructions from the user
  - Example of WHEN-NEW-FORM-INSTANCE trigger

```
Begin
   /* get rid of any implicit / left over transactions */
   forms_ddl ('rollback');
   /* start the read only transaction */
   forms_ddl ('set transaction read only');
End;
```

PRE-QUERY

- This fires once before the query is sent into the database
- JCC Rule: Make sure that ALL queries are done in read only transactions
  - Example of PRE-QUERY trigger

```
Begin
   /* get rid of any implicit / left over transactions */
   forms_ddl (‘rollback’);
   /* start the read only transaction */
   forms_ddl (‘set transaction read only’);
End;
```

POST-QUERY

- Fires once for each record returned from the database
- Warning: Developer returns a variable number of records per block depending on how many are displayed and how many are buffered in memory (both properties of the block)
  - Could leave a read only transactions open for a long time
    - Example: User queries and gets back 5 rows out of the possible 50 in the database. The transaction was read only (PRE-QUERY trigger). The user studies the data for 2 hours. The read only transaction would remain open for that entire time interval.

POST-QUERY (Continued)

- Therefore, since we don’t know (right away) how many rows are going to be returned, we must end the read only transaction and start a new one for each row returned from the database
  - Number of transactions increases
    - Forms displays 20 rows, and buffers 2 in memory, takes 22 read only transactions to get all the data to the client
  - Chance of “stale” transactions greatly reduced
  - Example of POST-QUERY trigger

```
Begin
   /* get rid of any implicit / left over transactions */
   forms_ddl (‘rollback’);
   /* start the read only transaction */
   forms_ddl (‘set transaction read only’);
End;
```

KEY-COMMIT

- This trigger fires once at the start of the commit process
  - This will always be the case if you begin the commit process by using the built in do_key (‘commit_form’)
- End the read only transaction and start the read write transaction to commit data
  - Example of KEY-COMMIT trigger

```
Begin
    /* get rid of any implicit / left over transactions */
    forms_ddl (‘rollback’);
    /* start the read write transaction */
    forms_ddl (‘set transaction read write’);
End;
```

POST-FORMS-COMMIT

- Fires once right after the forms process commits
- We want to make sure the read write transaction (started by the key-commit trigger) is committed to the database
  - Example of POST-FORMS-COMMIT trigger

```
Begin
   /* commit the transaction(s) */
   forms_ddl ('commit');
End;
```

POST-DATABASE-COMMIT

- Fires once after the database has successfully committed the transaction started by the commit_form process
  - In the JCC architecture, this would be the key-commit trigger
- We should start up a read only transaction
  - Example of POST-DATABASE-COMMIT trigger

```
Begin
   /* get rid of any implicit the read write was already committed */
   forms_ddl (‘rollback’);
   /* start the read only transaction */
   forms_ddl (‘set transaction read only’);
End;
```

Warning!

- If you encounter an error during the commit process, JCC strongly suggests that you rollback your transaction and begin a read only transaction
  - Otherwise, the post-forms-commit trigger will never execute and the read write transaction will never be ended

- Example:

```
Begin
   insert into ….

Exception when others then
   forms_ddl (‘rollback’)
   forms_ddl (‘set transaction read only’);
   < Tell user about error >
End;
```

So, What Does All This Mean for Multi-Form Applications?

- If all this code is added to all your forms:
  - All queries would be done in read only transactions
  - All writes done in read write transactions
- Implications to this approach
  - More transactions
  - Higher I/O to database root file
    - Much reduced in Rdb 7.0.5 if number of nodes is set to 1
  - TANSTAAFL

Queries in Multi-Form Applications

- So, how does this approach work?
  - All queries are translated into cursors with hold through OCI
    - This means that queries are open across transactions
      - Essence of the post-query trigger
  - Start and stop transactions without affecting the currently open database cursors used to fetch data through OCI

Queries in Multi-Form Applications (Continued)

- With this approach, if the user doesn’t do anything, then transactions can remain open for long durations
  - Think about the post-query trigger where we set the transaction state to read only
    - If the user simply stares at the data all day, that read only transaction will remain open
- The user is truly not seeing a snapshot of the data at the time they issue the query
  - Again, the post query trigger stops and starts transactions
  - JCC rule, if the user wanted a snapshot of the data, then we would use Reports to generate a report of what the data looked like at the time it was requested
  - JCC rule, users want to see the most up-to-date data possible

Minimizing “Stale” Read-Only Transactions

- Long running transactions are not good
  - Read Only Transactions
    - Increase snapshot size
    - Increase chances of viewing “stale” data
  - Read Write Transactions
    - Potentially hold locks
    - If data has been updated in your transaction, other users cannot view this data until the transaction has committed

- Can we prevent long running transactions?

Preventing Long Running Transactions

- Unfortunately, preventing long running transactions in a Windows-style environment is nearly impossible
  - You never know what the user is going to do next
- Minimizing the potential for long duration transactions is the best answer
  - JCC's answer
    - Use more transactions as opposed to fewer
    - Only switch to read write transactions for the time it takes to do actual insert
    - Use timers to check for inactivity
    - This by no means covers every possibility. We are still learning about what the users are doing and altering the transactional design to compensate for any "wierdnesses".

Preventing Long Running Transactions

- Use timers to check for inactivity
  - Consider the case where the user stares at one record all day
    - Read only transaction started by post-query trigger
  - After “n” number of minutes of inactivity, we should end the current transaction and start a new one
    - If the user isn’t doing anything on the screen, they probably left for the day
    - Shouldn’t hold any locks
    - Shouldn’t do anything to the screen
      - Auto logout is very frustrating
  - Consider using the WHEN-TIMER-EXPIRED trigger while also calling the Windows API to check for screen inactivity

WHEN-TIMER-EXPIRED

- This trigger fires once each time a client side timer expires
  - Use create timer to start a client side timer
    - Do this in the POST-QUERY trigger
  - When the timer expires, the trigger will fire
    - Rollback the existing read only transaction and start a new one

  - Note, PL/SQL is single threaded. This trigger cannot fire during the commit process. Therefore, we can never lose data.

Calling The Windows API

- JCC actually uses D2KUWTIL (Oracle package for Developer) that contains routines to reference the operating system
  - Win_API_Session.Timeout_get_inactive_time
    - This call uses the operating system to determine the amount of inactive time for the calling window (Developer)
- We check for inactivity for 5 minutes.
  - No transactions will remain open for longer than the 5 minute active timeout
  - Implication: If the user continues to look at the same data and the screen is active, this approach will fail

WHEN-TIMER-EXPIRED Example

```
Declare
   hWind                     PLS_INTEGER;
   CheckTimer                Timer;

Begin
   /* timeout every 5 minutes */
   if Win_API_Session.Timeout_get_inactive_time > 5 then
        forms_ddl (‘rollback’);
        forms_ddl (‘set transaction read only’);
        /* Delete the current timer, both API and FORMS  */
        Win_API_Session.timeout_delete_timer;
        delete_timer(‘CHECK_TIMEOUT_TIMER’);
        /* Start a new timer to catch next timeout */
        hWind := get_window_property(FORMS_MDI_WINDOW,WINDOW_HANDLE);
        Win_API_Session.Timeout_Start_Timer(hWind);
        default_value('1000','global.timeout_check_interval');
        /* Create FORMS timer to check for inactive timeout */
        CheckTimer := Create_Timer('CHECK_IIMEOUT_TIMER',1,repeat);
   end if;
end;
```

WHEN-TIMER-EXPIRED Trigger

- Warning: After the timer expires be sure to start another timer or else you will have one timeout and then a long running read only transaction
- Implications:
  - You'll rack up transactions – TANSTAAFL
    - In previous example, 1 Read Only transaction per 5 minutes per user

How To Implement The Transactional Model in Developer

- Writing all the specific code in each form to maintain the transaction context would be overwhelming
  - If a change needed to be made to the 5 triggers to maintain in 300 forms, I’d give up (1500 + code changes)
    - Good Luck Testing

- Is there a way to “write it once and forget it”?
  - Yes.
    - Property Classes
    - Object Libraries

Property Classes

- Property Classes are groups of Developer properties that can be applied to all objects of the same characteristics.
  - Similar to Rdb Domains
  - Examples
    - Date fields
    - Username fields
    - Money fields
    - Etc
  - Application of property classes is done through inheritance
    - Each object inherits all the properties of the applied property class
- Property classes can contain
  - Format Masks
  - Fonts Sizes/Weight/etc
  - Client-side triggers

Object Libraries

- Object Libraries are nothing more than a collection of objects
  - Blocks
  - Canvases
  - PL/SQL program units
  - Property Classes
- JCC recommends
  - Placing all the property classes into Object Libraries
  - All forms should inherit the property classes from the object library
  - All (most) objects should be associated with a property class

Implementing the Transactional Model

- JCC has applied these transactional routines to Object Libraries via property classes
  - Single point of coding inherited to all forms via Property Classes
  - If a change needs to be made, alter the object and upon recompilation all forms receive the updated code
  - Allows for consistency in the application
    - Each form does the same thing
- In other words, to make a change to hundreds of forms, alter the parent objects which will propagate the changes to all children forms.
  - Therefore, if 5 triggers needed updated in 300 forms, we would have 5 updates
    - Recompile all the forms modules and the inheritance will pull in new PL/SQL code

Automatic Deadlock Control

- Detection of Rdb Deadlocks can be done by trapping the exception message thrown by Rdb on an insert/update and searching the message for the word deadlock
  - JCC recommends writing all inserts/updates in procedures and client triggers
    - Override Developer defaults for writing to the database
    - More coding = Better control of application
      - The extra 15 minutes it takes to write the code is worth it
- If the exception contains the word deadlock, assume that a deadlock occurred, rollback and attempt to save the transaction again

Deadlock Code Example

- Example: (forms)

```
Declare
   my_error_text                   varchar2 (2000);
   pos                             number;
Begin
   <insert/update statement>
Exception when others then
   forms_ddl (‘rollback’);
/* DBMS_ERROR_TEXT is built in that returns the error from the database
   engine */

   my_error_text:=SQLERRM ||‘  ‘|| DBMS_ERROR_CODE ||’-’|| DBMS_ERROR_TEXT ;
   pos := instr (upper(my_error_text), ‘DEADLOCK’);
   if pos > 0
   then
     -- Deadlock found, re-save
     do_key (‘commit_form’);
   else
     -- Deadlock not found, stop processing
     raise form_trigger_failure;
   end if;
End
```

Automatic Deadlock Implications

- Applications should expect deadlocks
  - This method traps the error from the user
- Limit the number of auto-retries on deadlock processing
  - You don’t want application to go into an infinite loop
- The deadlock error messages can still show in the log files, yet be hidden from the user.

Conclusions

- It is possible to manage the transactions for Developer against Rdb
  - Leverage and understand Developer’s trigger topology to utilize FORMS_DDL and SRW.DO_SQL
  - Leverage the power of Rdb by understanding the database’s transactional model
- Use Object Libraries/Property Classes to encapsulate the logic of your application and inherit this functionality in your detail forms

For More Information…
Developer and Rdb Seminar Dates

- October 16-20, 2000 -- Sold Out
- January 22-26, 2001 -- Openings still available

- The 5 day seminar starts with forms basics and proceeds to object oriented design while including many tips and tricks for leveraging Rdb
  - Topics include:
    - Transactional control
    - Calling Stored Procedures/Functions
    - Error/Exception handling
    - Object Libraries, Property Classes and Object Groups
    - Etc.

Questions

Any further comments or questions…

Jeffh@jcc.com