Using Thick Database Principles to Leverage Oracle SQL and PL/SQL Part III:

Implementation Techniques

Peter Koletzke

Technical Director & Principal Instructor

ORACLE ACE Director

ORACLE Developer Community

NoCOUG

Me

- 34 yrs. database industry
- 30 yrs. consulting in Oracle arena
  - Since Oracle 5.1C, SQL*Forms 2.3
- 38 yrs. as trainer/presenter
- User groups
  - 350+ presentations, 12 awards
  - 7+ yrs. total on boards of directors
    - IOUG(-A), NYOUG, UTOUG
- Oracle ACE Director
  - Since program inception in Aug. 2005
- Oracle Certified Master
  - Since program inception in Dec. 2001
- 8 Oracle Press books coauthored
  - 6262 pages total

You

- Job responsibilities?
  - DBA, developer
- Development tools?
  - Oracle Developer Forms/Reports
  - APEX
  - ADF, MAF
  - MAX, VBCS
  - Other JavaScript tools
  - .NET
  - PL/SQL
  - Other

Series Overview

Earlier sessions

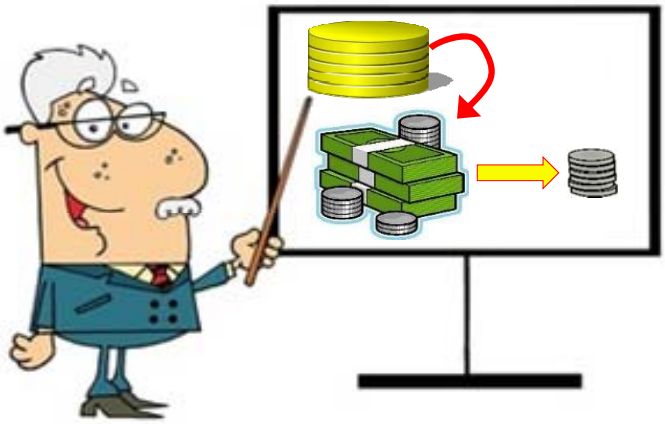
Thick Concepts
- What is Thick Database?
- Why is Thick faster?
- Thick Advice

Business Rules
- The subject is business rules
- Sample business rules systems
- Where to place business rules code

Techniques
- Thick Database objects
- Sample code
- Level of Thick

Slides will be available on the NoCOUG website.

A Brief Review



Review

About Thick Database

Review

- A code development strategy
  - Maximize use of database code to simplify the user interface
  - The user's device (client) runs minimal code
- Name plays off the term "thin client"
  - A "Year of the Internet" term
  - Means most processing occurs on a server
  - Slightly outmoded now
- Thick database means "thin client"

A.k.a.

- Thick Database Approach
- Thick Database Paradigm
- Smart Database
- SmartDB
- Fat Database

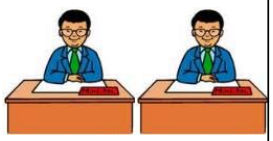
Oracle prefers

Review

Guiding Principles for Code

Review

- Database code that implements business rules in PL/SQL
  - Using database features to enforce data integrity defined by business rules seems obvious
- Database views to represent complex business objects (SQL)
  - Each view has an accompanying application programming interface (API)
    - Written in PL/SQL
  - Interaction is with view and API



Review

Benefits

Review

- Application accuracy
  - Business rules match application code
  - Test plans can be generated from business rules system
- Productivity
  - Can greatly simplify user interface code
- Code reusability
  - Ease of application maintenance
- Faster performance
  - Code is close to data storage – fewer messages, easy access
  - Views also reduce the number of round trips needed
- Proper use of staff
  - User interface developers can concentrate on UI code
  - Database code developers can concentrate on database code to support the UI

Review

More Benefits

Review

- Save cloud database processing time
  - Application's use of database is more efficient
  - Proof in next section
  - Less database processing time == lower cloud costs
  - Save application processing time, too
- Simplify user interface development
  - Database views and PL/SQL form API to the database
  - Application code is reduced
  - UI code technology can change without total application rewrite

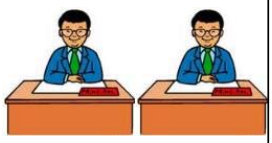
Review

Two Related Strategies

Review

1. System for tracking business rules
   - Store definitions in database tables
   - Code implementations linked to rules
   - Linked to, or serves as, requirements documentation
2. Database features implement the business rules
   - Constraints
   - Database API: PL/SQL code, updatable views

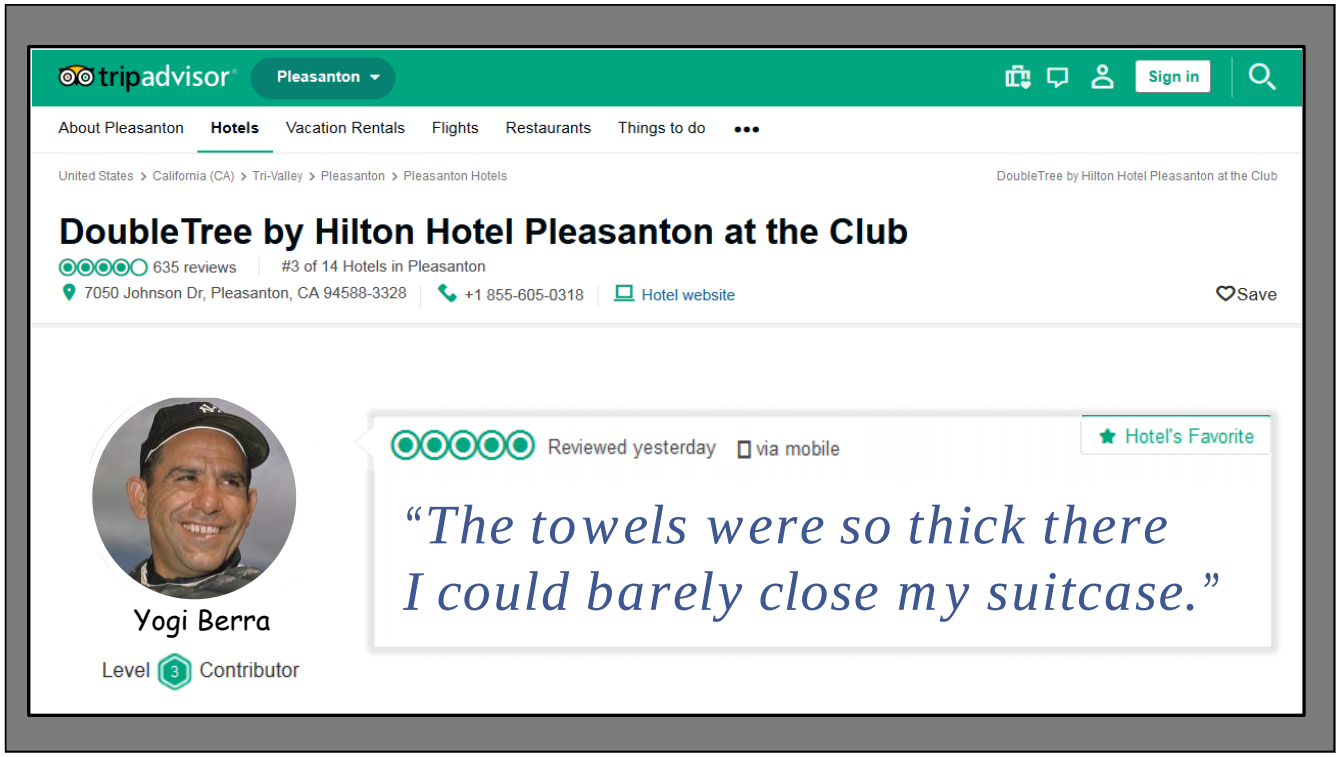
Oracle is considering support for SQL Assertions:
https://community.oracle.com/ideas/13028



Agenda

Thick Techniques

- Thick Database objects
- Sample code
- Level of Thick



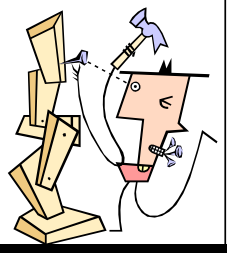
How Many Schemas?

- Two
  - Connect schema
  - Table owner schema (owns PL/SQL code and tables)
- OR Three
  - Connect schema
  - PL/SQL schema (bus rules and TAPI code; views) – (AUTHID DEFINER)
  - Table owner schema (owns tables)
- OR Four [1]
  - Connect schema
  - API PL/SQL schema (owns API code; views; package per page)
  - BR PL/SQL schema (owns bus rules code)
  - Table owner schema (owns tables)
- Synonyms - optional
- DDL triggers – to guard against additional grants or disabling triggers or creating synonyms

(1) *Why Use PL/SQL?* By Bryn Llewellyn

Database Components

- Tables – the usual
  - No grants or synonyms to other schemas
- Table API packages
  - INSERT, UPDATE, DELETE, (SELECT) procedures
  - Call business rules validation code
- Views on the tables
  - Queries can be arbitrarily complex
- INSTEAD OF triggers on the views
  - Call the table API procedures



Agenda

Thick Techniques

- Thick Database objects
- Sample code
- Level of Thick

Table API

- A PL/SQL package per table
  - All data modification ("DML") is accomplished through procedures
    - INS()
    - UPD()
    - DEL()
    - LCK()
- Procedures are called only from INSTEAD OF view triggers
- No grants to table at all

Code samples are available in Appendix A.

Demo 1

Optional Table API Components

- A function can act as SELECT
  - A bit trickier and not always necessary
  - Virtual Private Database policies can filter data to all SELECT statements instead
- Package enforcement global variable
  - Trigger uses it to prevent “DML” statements outside of the package
    - Applies only to table owner because table has no grants
  - Access only by Table API

Package Enforcement Global Variable

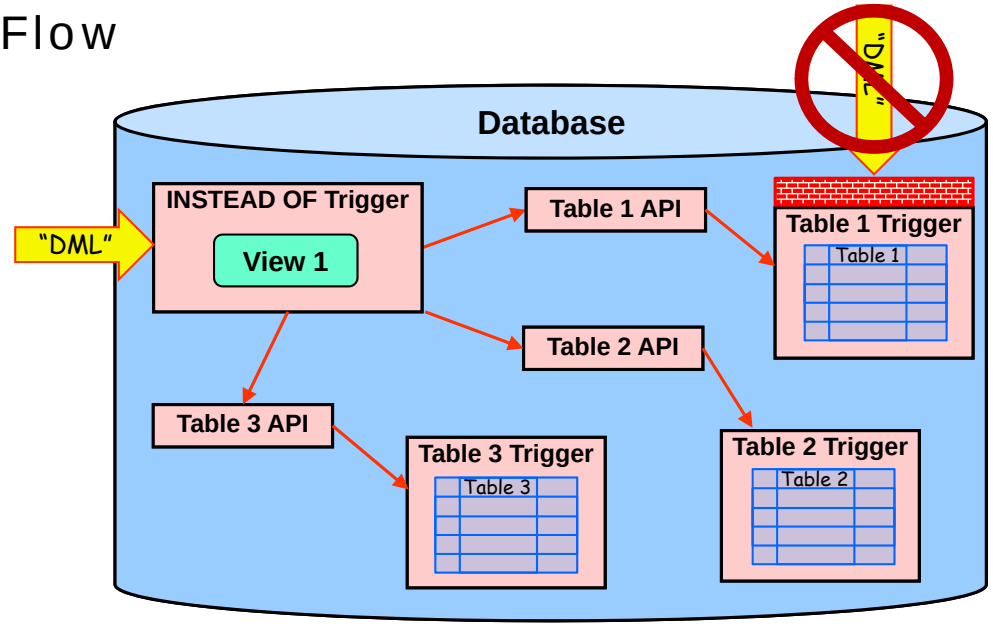
```
CREATE OR REPLACE TRIGGER employees_trbr
   BEFORE INSERT OR UPDATE OR DELETE
   ON employees
   FOR EACH ROW
BEGIN
   --
   IF NOT employees_pkg.g_allow_dml
   THEN
      RAISE_APPLICATION_ERROR(-20199,
         'You may not issue INSERT, UPDATE, or ' ||
         'DELETE statements to this table.');
   END IF;
   -- other code for validating rules
END employees_trbr;
```

Database Views and Triggers

- Views on tables requiring access
- INSTEAD OF triggers on the views
  - INSERT, UPDATE, DELETE row-level trigger
    - Call Table API procedures
  - Exceptions
    - Cross-row validation requires statement-level triggers on tables or application code
    - Cross-table validation requires application code

Demo 2

SQL Flow

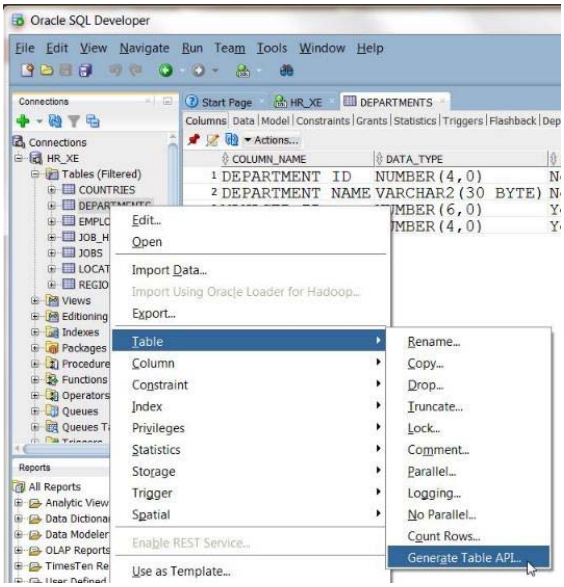


Generate the Stub Code

- It's all cookie cutter stuff at the start
  - Table API – triggers and packages
  - View INSTEAD OF trigger
- Use a prebuilt generator
  - http://www.dbartisans.com/oracle/docs/PLSQL_Frameworks_and_Libraries.pdf
  - https://www.oddgen.org/
  - André Borngräber, Ottmar Gobrecht
    - https://github.com/OraMUC/table-api-generator
- Or roll your own generator

SQL*Developer Solution

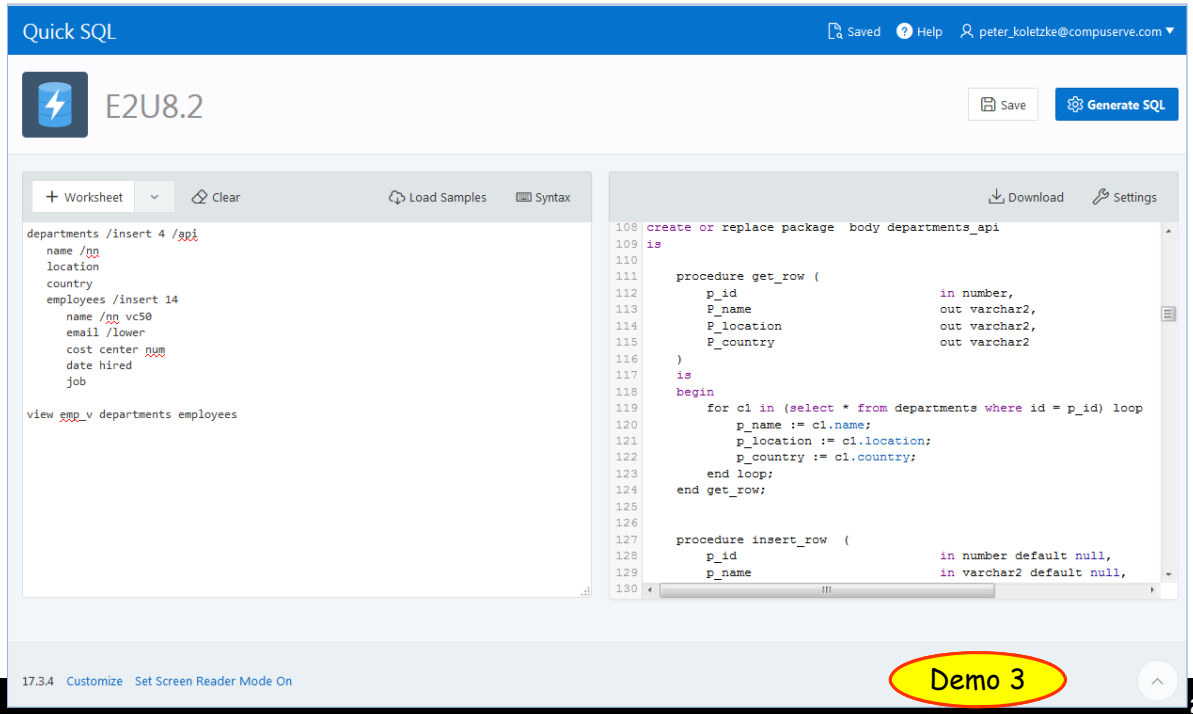
- Select table
- From right-click menu: Table → Generate Table API
- Be sure to Ctrl-F7 to reformat it
- You may want to use it just as a basis for your own code



“NEW!” – Quick SQL

- quicksql.oracle.com
- An APEX app
- Youtube video published Feb. 2017
- Generates code for tables, views, constraints, etc. (even sample data!) based on shorthand specs
- Settings control variations
  - Table API is one of the options

Quick SQL



Demo 3

A Gotcha: Cross-row Rules

- Row-level triggers can only check a single record
- Master-detail form
  - One commit for changes to multiple records
  - Update could be to master or detail(s) or both records
  - Validation must occur between master and all details
- Also: between rows of the same table
  - Row-level actions will not work
- In general: need to check business rules after all insert, update, delete statements and before commit
  - Database does not offer a commit trigger

Example

- Department entity object
  - StartDate and EndDate define active period
- Employees entity object
  - Includes all employees hired or retired
  - StartDate and EndDate define active period
- Business rules
  1. Only one employee can be active in a particular job at a time
  2. The active period for an employee must fall within the active period for the department.

Sample Scenarios

Departments	
DepartmentId	90
DepartmentName	Executive
StartDate	6/16/2003
EndDate	

Employees					
Employee ID	LastName	StartDate	EndDate	JobId	Department ID
100	King	6/17/2003		AD_PRES	90
101	Kochhar	9/21/2005	1/12/2007	AD_VP	90
102	DeHaan	1/13/2007		AD_VP	90

- Change DeHaan *StartDate* to 1/1/2007 and commit
- Change Dept 90 *StartDate* to 6/18/2003 and commit

Cross-row Validation Solution

- Call cross-table procedure from the UI code before after other updates, before commit
  - Uses current session and can query uncommitted data
  - Query (or pass to procedure) master values and detail records
  - Process the values
  - Return success or failure

Other Gotchas

- Can't use a RETURNING clause for an INSERT on a view with an INSTEAD OF trigger
- Views cannot have a ROWID column

"Real World" Code Examples

- Table APIs
  - BR_RULE_PKG
  - BR_RULE_TRBR
- Database views
  - BR_RULE_VW
- INSTEAD OF triggers
  - BR_RULE_VW_TRBR
- Business rules procedures
  - BR_RULE_PKG.VALIDATE_BRRULE_ID()

Demo 4

Agenda

Thick Techniques

- Thick Database objects
- Sample code
- Level of Thick

Main Levels

Complexity, Flexibility

- Light
- Moderate
- Deep
- Extreme

Light: Application Code Only

- The simplest architecture
  - Not really “Thick Database”
- Business rule statements in printed documents only
  - No link between business rules statements and programming code
  - Difficult to maintain and report on business rules statements
- Most business rules code in the application
- Use database constraints

Moderate: Application Code with Business Rules Repository

- “Modified Thick Database Approach”
- Business rules statements are stored in database tables: a business rules repository
- Maintenance and reporting of business rules is easier and more flexible
  - Can report on groups of business rules
  - Names of programmatic objects can be stored in the repository
  - ID numbers for business rules can be added to comments in the application code
- Link requirements and test plans to business rules

Deep: Code Generation Engine

Independent of Table API, trigger, etc. stub generators

- Extremely Thick Database Approach
- Business rules repository as in Moderate
- The system generates **application** code (database and user interface) from the business rules repository
  - Business rules statements are tightly coupled to application code
- Downside: time consuming to produce an engine that can handle all possibilities
  - Possible compromise: settle for less than 100% generation
  - For example, hand-code the UI application and generate the validation code

Benefits of Deep

- The code is immediately useable (100% generation)
- Development time is minimized
- Time spent on proper definitions of business rules
  - Same for maintenance and enhancements
- A single engine can serve multiple applications
- Relatively immune to technology shifts
  - Except the UI generator would need to be altered for a different UI technology

Extreme: Business Rules Runtime Engine

- Generic code reads the business rules repository at runtime
  - Generates the user interface dynamically
  - Data are validated using the business rules repository definitions
  - The tightest coupling between application and business rules
- Development process consists only of defining the business rules
  - Maintenance and enhancements are simple
- Downside: even more time consuming to produce this system
  - Needs experts to create and maintain
- Relatively immune to technology shifts
  - Although the runtime engine would need to be altered for a different UI technology

Summary

- Leverage SQL: Database views, no grants to tables
- Leverage PL/SQL: View INSTEAD OF triggers, table triggers, table APIs
- Different levels of thick depending on available time and talent
- Incorporating it requires some ramp-up time: use a phased approach

- **Please fill out the evaluations**
- **7 of 8 books co-authored with Dr. Paul Dorsey, Avrom Roy-Faderman, & Duncan Mills**
- **Slides will be on the NoCOUG website**