When and How to Write SQL in Oracle PL/SQL
Do we *really* need to talk about SQL in PL/SQL?

• Consider the typical (and often silent) conversations we have about SQL.

**What we say about SQL**

- *It's so easy to write SQL in PL/SQL!* **Taken For Granted**
- Oracle will optimize it all for me automagically. **Drinking the Kool Aid**
- *This INSERT will never raise a DUP_VAL error.* **Rationalizing Laziness**
- No one else will ever need to write a SQL statement like this. **Exceptionalism**
SQL/Database—Most Critical Part of Application

• SQL statements directly reflect our business models.
  – And those models are always changing. And then there's the data.

• SQL statements cause most of the performance problems in our applications.
  – Tuning SQL and the way that SQL is called in PL/SQL overwhelms all other considerations.

• Many runtime errors in applications result from integrity and check constraints on tables.

• Finally, just because NoSQL and Node.js and whatever else devalue SQL and database, doesn't mean they're right.
  – But it does mean we have our work cut out for us. Starting with us.
We Take SQL for Granted

• So we write SQL whenever and wherever we need it, over and over again.
  – We don't set rules on how, when and where SQL should be written.

• We don't stay current.
  – Fall back into PL/SQL (or "worse") when going gets tough.

The result? Slow, buggy code that is difficult to optimize and maintain.
Two key SQL questions for your next project

- At a minimum, before starting your next application, ask yourselves explicitly:

  1. Are we **taking full advantage of SQL**, particularly new features in our version?
     - You should do as much as possible in "pure" SQL.

  2. Do we **want standards**, or should we just keep on doing whatever we want, whenever we want?
     - That way, you are at least making a **conscious** decision.
Fully leverage SQL in your PL/SQL code

• Oracle continually adds significant new functionality to the SQL language.

• If you don't keep up with SQL capabilities, you will write slower, more complicated PL/SQL code than is necessary.
  – I am a pretty good example of what you don't want to do or how to be.
  – And, by the way, table functions are a great way to escape from SQL!

• So take the time to refresh your understanding of Oracle SQL in Oracle Database 11g and Oracle Database 12c.
  – Start with the New Features list in the doc set.
  – See if anything stands out immediately and dive in from there.
  – Also check Tom Kyte's resources on OTN and AskTom.
Some exciting recently added SQL features

• Analytical Functions
  – Such as LAG, LEAD, ranking etc.; these allow you to look to previous and following rows to calculate differences.

• WITH clause (subquery factoring)
  – Allows the definition of 'views' inside a query that can be used and reused; they allow procedural top-down logic inside a query
  – In 12.1 you can also define a function inside a query!

• Flashback query
  – No more need for journal tables, history tables, etc.

• ANSI JOIN syntax
  – Replaces the (+) operator and introduces FULL OUTER JOIN

• SYS_CONNECT_BY_PATH and CONNECT_BY_ROOT for hierarchical queries

• Scalar subquery
  – Adds a subquery to a query like a function call.

```sql
select d.deptno, (select count(*) from emp e where e.deptno = d.deptno) number_staff from dept
```
So do we want standards for writing SQL?

• *Please*, say yes!
  – Without standards we will not be able to take full advantage of Oracle Database and we will not be (as) successful (as we could be).

• In this presentation, I will focus on a small number of the most important best practices.

• Many excellent and detailed recommendations from Bryn Llewellyn
  • Search for "Doing SQL from PL/SQL: Best and Worst Practices"

• But first I will try and convince you that....

Every SQL statement you write is a hard-coding.
And SQL hard-codings are the *worst kind*. 
How is a SQL statement different from a literal?

- When a literal changes, you must find all occurrences and change them.
  - With values like "45098", a global search/replace gets the job done (if you are very careful).
- When a table changes, you must find all the SQL statements that are affected.
  - When your hard-coding is a DML statement, you've got a serious problem.
  - How can you be sure you found all the same logical occurrences?
- Logically, a SQL statement and a literal value are the same, when it comes to hard-coding.
  - So if you don't like hard-coded literals, you should hate hard-coded SQL.
Writing SQL is all about defining the Now

Every SQL statement you write says:

At this moment in time, here's how to capture this relationship or business rule.

But our ERDs are quite complex and they don't get simpler over time.
What to do about SQL hard coding

• You have to (and should) write SQL statements in your application.
  – Use Oracle Database!

• The question becomes: *where* you should put those statements?

• DON'T
  – Put SQL statements in Java or .Net or Javascript. Fight the good fight!

• DO
  – Hide complex queries behind views, and use those views in reports.
  – Put all your other SQL statements in PL/SQL packages (cursors, subprograms), hiding the SQL behind an API of procedures and functions.
Hide SQL statements behind a PL/SQL API

• Think of SQL as a *service* that is provided to you, not something you *write*.
  – Or if you write it, you put it somewhere so that it can be easily found, reused, and maintained.

□ This service consists of views and programs defined in the data access layer.
  – Views hide complex query construction
  – Packaged APIs – for tables, transactions and business entities

• That intermediate ("intercept") layer offers tremendous benefits.
With a data access layer, I can...

• Change my implementation with minimal impact on application code.
  – The underlying data model is constantly changing.
  – We can depend on Oracle to add new features.
  – We learn new ways to take advantage of PL/SQL.

• Improve my SQL-related error handling.
  – Do you take the time to handle dup_val_on_index for INSERTs,
    too_many_rows for SELECT INTOs, etc?

• Greatly increase my productivity
  – I want to spend as much time as possible implementing business
    requirements.
CREATE OR REPLACE PACKAGE employee_rp
AS
  SUBTYPE fullname_t IS VARCHAR2 (200);
  -- The formula
  FUNCTION fullname (l employee.last_name%TYPE, f employee.first_name%TYPE)
  RETURN fullname_t;
  -- Retrieval function
  FUNCTION fullname (
    employee_id_in IN employee.employee_id%TYPE
  )
  RETURN fullname_t;
END;

CREATE OR REPLACE PROCEDURE process_employee (employee_id IN number)
IS
  l_name VARCHAR2(100);
BEGIN
  SELECT last_name || ',' || first_name INTO l_name FROM employee
  WHERE employee_id = process_employee.employee_id;
  ... 
END;

A simple example of hiding SQL behind a function
Data Access Layer Tips and Gotchas

• Give yourself the "space" to do it right.
  – Doesn't take all *that* much time to move query to function and call function.
  – But it does take some discipline and patience.

• Tom Kyte strongly recommends transaction APIs.
  – I agree and add that table APIs can also be very helpful, when used correctly.

• Do not call a "cascade" of functions, when you can execute a single query (in a possibly *new* function).
  – It's kind of like of nested cursor FOR loops instead of a join! Both bad news....

• Generate as much of the code as you can to ensure consistency.

```sql
BEGIN
BAD CASCADE
IS
  l_employee := employees_pkg.one_row (employee_id_in);
  l_department := departments_pkg.one_row (l_employee.department_id);
END
```
Other Recommendations for SQL in PL/SQL

• When fetching a single row, use SELECT-INTO (and put it inside a function).
  – Or EXECUTE IMMEDIATE-INTO with a dynamic query.

• When fetching and changing multiple rows, use bulk processing features: BULK COLLECT and FORALL.
  – Covered in separate webinar in this series.

• For dynamic SQL...
  – Bind rather than concatenate whenever possible.
  – Always EXECUTE IMMEDIATE a variable and include an exception handler so that the variable can be analyzed easily.
  – Executing dynamic DDL? Make it an autonomous transaction!
Always use SELECT-INTO for single row fetches

- A great example of how "everything changes" and we must update our thinking.
  - "Long, long ago..." an explicit cursor was faster than an implicit. Ancient history.

- SELECT-INTO and EXECUTE IMMEDIATE-INTO....
  - Perform better, more accurately "tell the story", raise errors when reality doesn't fit the story.
Bind rather than concatenate with dynamic SQL

• When you concatenate....
  – Your new, different SQL statement must be parsed;
  – It is more vulnerable to SQL injection;
  – It is much harder to read and maintain.

• Bind all variables instead.
  – Much higher likelihood that the parsed SQL statement can be "reused".
  – You can't inject into a bind variable.
  – The resulting code is much simpler and more readable.
EXECUTE IMMEDIATE a variable, not an expression

- The hardest part about dynamic SQL is getting that string right.
  - Missing a comma, space between clauses, etc.
- If you construct the string inside EXECUTE IMMEDIATE, it's "lost".
- Instead, assign to a variable and EXECUTE IMMEDIATE that.
- Then when something goes wrong, you can log/display the variable in your exception handler.
  - The problem is usually quite obvious.
Dynamic DDL commits, so make it autonomous

• Every parse (and certainly execute) of a DDL statement causes an implicit commit in your session.

• Dynamic DDL should be done with great care in a production application.
  – Can cause a cascading ripple of invalidations.

• You can "limit" the effect of the implicit commit to just that DDL statement by adding the AUTONOMOUS_TRANSACTION pragma.

• Oh, and you should also probably use AUTHID CURRENT_USER.
  – That way, you ensure that the effect of the dynamic DDL (and SQL) is on the invoking, not defining, schema.
When and How to Write SQL - Conclusions

• Leverage SQL fully.
  – Reserve PL/SQL for the logical flow and requirements that cannot be met with set-oriented SQL.
  – Keep up to date on the many powerful enhancements to SQL.

• Avoid repetition of SQL statements.
  – Signals a loss of control in your application.
  – Makes it very hard to optimize and maintain your code.
  – Hide your SQL behind views and package APIs.

• SET STANDARDS for writing SQL in your applications!
Some Useful Websites for PL/SQL Developers

• asktom.oracle.com: Tom Kyte's famed Q&A on Oracle Database
• oracle.com/plsql: starting point for Oracle's resources on PL/SQL.
• plsqlchallenge.com: quizzes on PL/SQL, SQL, database design, logic and more
• oracle-developer.net: Adrian Billington's repository of articles
• Oracle-BASE.com: Tim Hall's set of resources and scripts

All My Code available here:
oracle.com/webfolder/technetwork/tutorials/plsql/sfdemo.zip
or here: goo.gl/FPG79Z
My PL/SQL blog: stevenfeuersteinonplsql.blogspot.com
My PL/SQL twitter handle: @sfonplsql
Hardware and Software
Engineered to Work Together