Optimize SQL in PL/SQL

Optimize SQL in PL/SQL programs

- Take advantage of PL/SQL-specific enhancements for SQL.
  - BULK COLLECT and FORALL
  - Table functions
- Top Tip: Stop writing so much SQL
  - A key objective of this presentation is to have you stop taking SQL statements for granted inside your PL/SQL code.
  - Instead, you should think hard about when, where and how SQL statements should be written in your code.

What is wrong with this code?

- Time to “process” 5,000 employees in a department....

```
CREATE OR REPLACE PROCEDURE process_employee (
   department_id IN NUMBER)
IS
   l_id         INTEGER;
   l_dollars    NUMBER;
   l_name       VARCHAR2 (100);

   /* Full name: LAST COMMA FIRST (ReqDoc 123.A.47) */
   CURSOR emps_in_dept_cur IS
      SELECT employee_id, salary
           , last_name || ',' || first_name lname
        FROM employees
       WHERE department_id = department_id;
BEGIN
   OPEN emps_in_dept_cur;

   LOOP
      FETCH emps_in_dept_cur INTO l_id, l_dollars, l_name;

      analyze_compensation (l_id, l_dollars);

      UPDATE employees SET salary = l_salary
       WHERE employee_id = employee_id;

      EXIT WHEN emps_in_dept_cur%NOTFOUND;
   END LOOP;
END;
```

For a particular department ID, get all the employees, construct the “full name” and update the salary.

I found at least 15 items that needed fixing (not all of them SQL-related!).

wrong_code.sql

Turbo-charged SQL with Bulk Processing Statements

- Improve the performance of multi-row SQL operations by an order of magnitude or more with bulk/array processing in PL/SQL!

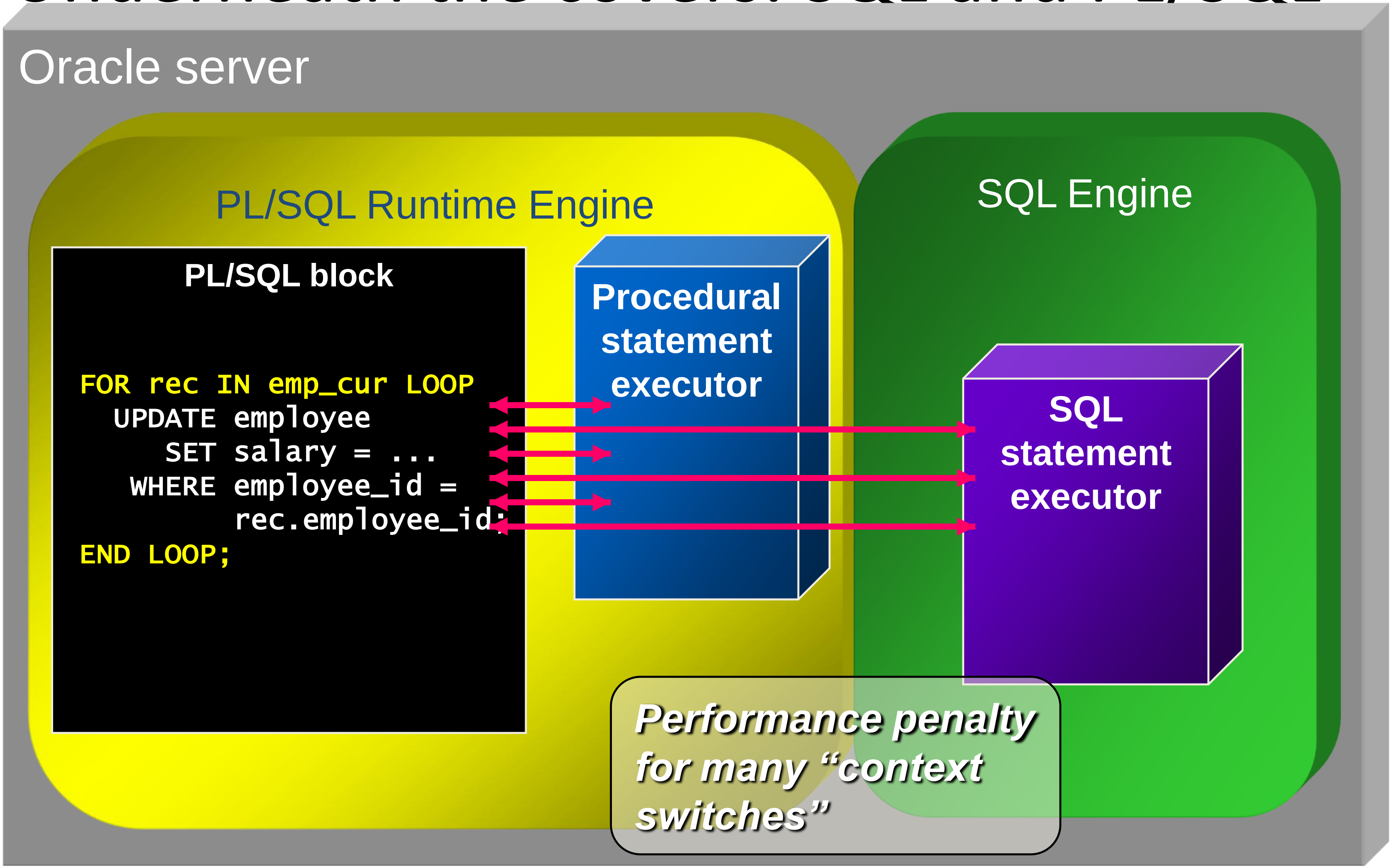
```
CREATE OR REPLACE PROCEDURE upd_for_dept (
   dept_in IN employee.department_id%TYPE
  ,newsal_in IN employee.salary%TYPE)
IS
   CURSOR emp_cur IS
      SELECT employee_id,salary,hire_date
        FROM employee WHERE department_id = dept_in;
BEGIN
   FOR rec IN emp_cur LOOP

      adjust_compensation (rec, newsal_in);

      UPDATE employee SET salary = rec.salary
       WHERE employee_id = rec.employee_id;
   END LOOP;
END upd_for_dept;
```

Row by row processing of data

Underneath the covers: SQL and PL/SQL



Oracle server

PL/SQL Runtime Engine

PL/SQL block

```
FOR rec IN emp_cur LOOP
  UPDATE employee
     SET salary = ...
   WHERE employee_id =
         rec.employee_id;
END LOOP;
```

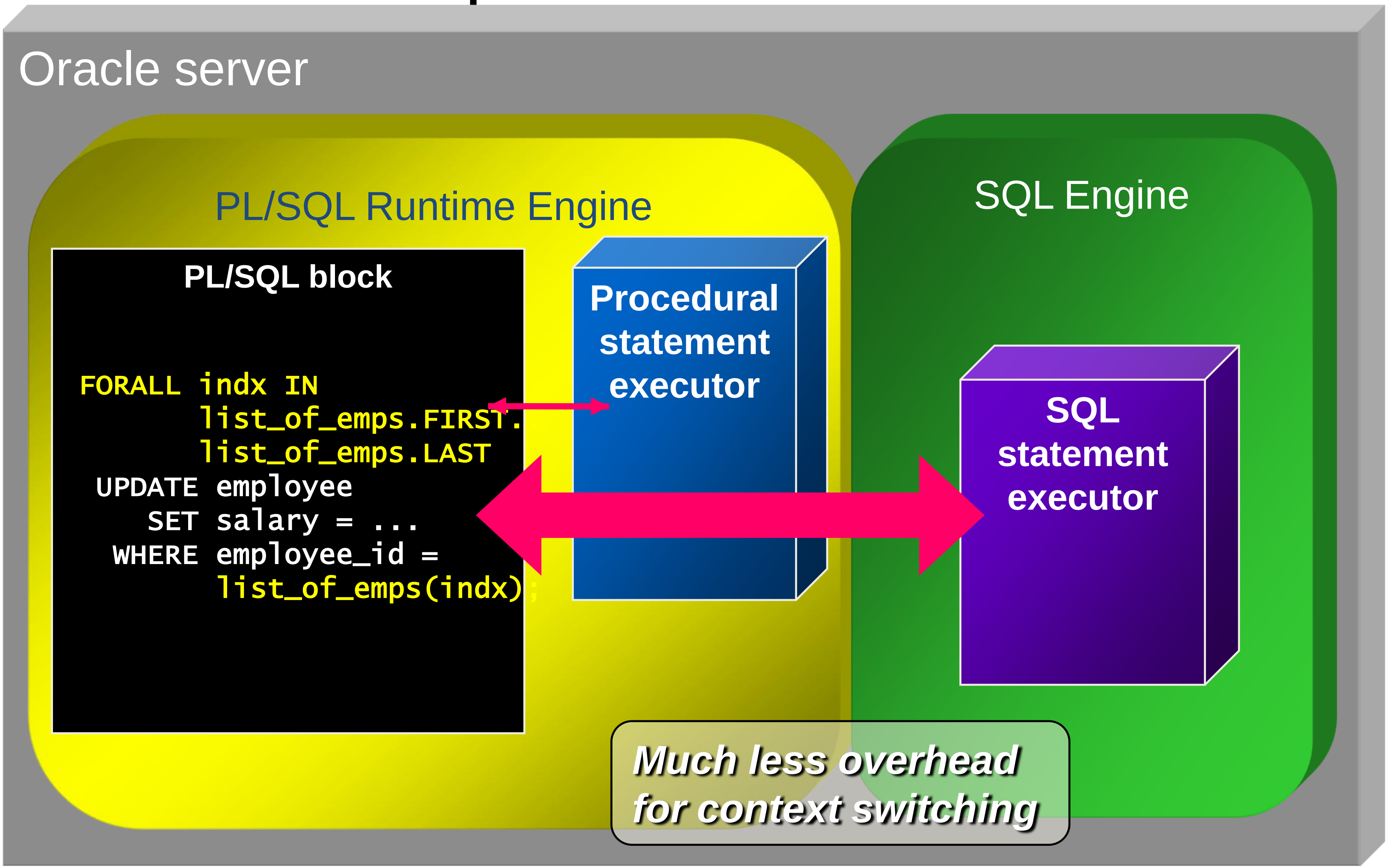
Procedural statement executor

SQL Engine

SQL statement executor

Performance penalty for many “context switches”

A different process with FORALL



```
FORALL indx IN
       list_of_emps.FIRST.
       list_of_emps.LAST
 UPDATE employee
    SET salary = ...
  WHERE employee_id =
        list_of_emps(indx);
```

Much less overhead for context switching

Bulk Processing in PL/SQL

- You should consider moving all multi-row SQL processing over to these new approaches.
- BULK COLLECT
  - Use with implicit and explicit queries.
  - Move data from tables into collections.
- FORALL
  - Use with inserts, updates and deletes.
  - Move data from collections to tables.

Use BULK COLLECT INTO for Queries

Declare a collection of records to hold the queried data.

Use BULK COLLECT to retrieve all rows.

Iterate through the collection contents with a loop.

```
DECLARE
   TYPE employees_aat IS TABLE OF employees%ROWTYPE
      INDEX BY BINARY_INTEGER;

   l_employees employees_aat;
BEGIN
   SELECT *
   BULK COLLECT INTO l_employees
     FROM employees;

   FOR indx IN 1 .. l_employees.COUNT
   LOOP
      process_employee (l_employees(indx));
   END LOOP;
END;
```

bulkcoll.sql
bulktiming.sql

WARNING! BULK COLLECT will *not* raise NO_DATA_FOUND if no rows are found.
Always check contents of collection to confirm that something was retrieved.

Limit the number of rows returned by BULK COLLECT

```
CREATE OR REPLACE PROCEDURE bulk_with_limit
   (deptno_in IN dept.deptno%TYPE)
IS
   CURSOR emps_in_dept_cur IS
      SELECT  *
        FROM emp
       WHERE deptno = deptno_in;

   TYPE emp_tt IS TABLE OF emps_in_dept_cur%ROWTYPE;
   emps emp_tt;
BEGIN
   OPEN emps_in_dept_cur;
   LOOP
      FETCH emps_in_dept_cur
         BULK COLLECT INTO emps
         LIMIT 1000;

      EXIT WHEN emps.COUNT = 0;

      process_emps (emps);
   END LOOP;
END bulk_with_limit;
```

Use the LIMIT clause with the INTO to manage the amount of memory used with the BULK COLLECT operation.

Definitely the preferred approach in production applications with large or varying datasets.

bulklimit.sql

Tips and Fine Points for BULK COLLECT

- Can be used with implicit and explicit cursors
- Collection is always filled sequentially, starting at row 1.
  - So you are *always* safe using a FOR loop to iterate through, as shown on previous page.
- Production-quality code should generally use the LIMIT clause to avoid excessive memory usage.
- Note: Oracle will automatically optimize cursor FOR loops to BULK COLLECT performance levels.
  - But it will not be sufficient if the cursor FOR loop contains DML statements!

emplu.pkg
10g_optimize_cfl.sql

The FORALL Bulk Bind Statement

- Instead of executing repetitive, individual DML statements, you can write your code like this:

```
PROCEDURE upd_for_dept (...)  IS
BEGIN
   FORALL indx IN list_of_emps.FIRST .. list_of_emps.LAST
      UPDATE employee
         SET salary = newsal_in
       WHERE employee_id = list_of_emps (indx);
END;
```

- Things to be aware of:
  - **You MUST know how to use collections to use this feature!**
  - **Only a single DML statement is allowed per FORALL.**
  - **New cursor attributes: SQL%BULK_ROWCOUNT returns number of rows affected by each row in array. SQL%BULK_EXCEPTIONS...**
  - **Prior to Oracle10g, the *binding array* must be sequentially filled.**
  - **Use SAVE EXCEPTIONS to continue past errors.**

bulktiming.sql
bulk_rowcount.sql

Better Exception Handling for Bulk Operations

- Allows you to continue past errors and obtain error information for each individual operation (for dynamic and static SQL).

```
CREATE OR REPLACE PROCEDURE load_books (books_in IN
book_obj_list_t)
IS
   bulk_errors EXCEPTION;
   PRAGMA EXCEPTION_INIT ( bulk_errors, -24381 );
BEGIN
   FORALL indx IN books_in.FIRST..books_in.LAST
      SAVE EXCEPTIONS
      INSERT INTO book values (books_in(indx));
EXCEPTION
    WHEN BULK_ERRORS THEN
       FOR indx in 1..SQL%BULK_EXCEPTIONS.COUNT
       LOOP
          log_error (SQL%BULK_EXCEPTIONS(indx));
       END LOOP;
END;
```

Allows processing of all rows, even after an error occurs.

New cursor attribute, a pseudo-collection

bulkexc.sql

Tips and Fine Points for FORALL

- Use whenever you are executing multiple single-row DML statements.
  - Oracle suggests you will see benefit with 5 or more rows.
- Can be used with any kind of collection.
- Collection subscripts cannot be expressions.
- You cannot reference fields of collection-based records inside FORALL.
  - But you can use FORALL to insert and update entire records.
- Prior to Oracle10g Release 2, the bind collections must be densely filled.
  - The newer VALUES OF and INDICES OF clauses give you added flexibility (coming right up!).

Collections impact on "Rollback segment too small" and "Snapshot too old" errors

- Rollback segment too small...
  - Cause: so many uncommitted changes, the rollback segment can't handle it all.
  - FORALL will cause the error to occur even sooner. Use a variation on incremental commits with FORALL.
- Snapshot too old...
  - Cause: a cursor is held open too long and Oracle can no longer maintain the snapshot information.
  - Solution: open-close cursor, or use BULK COLLECT to retrieve information more rapidly.

forall_incr_commit.sql

Cursor FOR Loop ... or BULK COLLECT?

- Why would you ever use a cursor FOR loop (or other LOOP) now that you can perform a BULK COLLECT?
  - If you want to do complex processing on each row as it is queried – and possibly halt further fetching.
  - You are not executing DML within your cursor FOR loop and you are on Oracle Oracle Database 10g – Oracle will automatically optimize the code for you.
- Otherwise, moving to BULK COLLECT is a smart move!

cfl_vs_bulkcollect.sql
cfl_to_bulk.sql

More flexibility for FORALL

Oracle10g

- In Oracle10g, the FORALL driving array no longer needs to be processed sequentially.
- Use the INDICES OF clause to use only the row numbers defined in *another* array.
- Use the VALUES OF clause to use only the *values* defined in another array.

Using INDICES OF

- It only processes the rows with row numbers matching the *defined rows* of the driving array.

```
DECLARE
   TYPE employee_aat IS TABLE OF
employee.employee_id%TYPE
      INDEX BY PLS_INTEGER;
   l_employees           employee_aat;
   TYPE boolean_aat IS TABLE OF BOOLEAN
      INDEX BY PLS_INTEGER;
   l_employee_indices   boolean_aat;
BEGIN
   l_employees (1) := 7839;
   l_employees (100) := 7654;
   l_employees (500) := 7950;
   --
   l_employee_indices (1) := TRUE;
   l_employee_indices (500) := TRUE;
   l_employee_indices (799) := TRUE
   --
   FORALL l_index IN INDICES OF l_employee_indices
      BETWEEN 1 AND 500
      UPDATE employee
         SET salary = 10000
       WHERE employee_id = l_employees (l_index);
END;
```

10g_indices_of.sql
10g_indices_of2.sql

Using VALUES OF

- It only processes the rows with row numbers matching the *content* of a row in the driving array.

```
DECLARE
   TYPE employee_aat IS TABLE OF
employee.employee_id%TYPE
      INDEX BY PLS_INTEGER;
   l_employees           employee_aat;

   TYPE values_aat IS TABLE OF PLS_INTEGER
      INDEX BY PLS_INTEGER;
   l_employee_values   values_aat;
BEGIN
   l_employees (-77) := 7820;
   l_employees (13067) := 7799;
   l_employees (99999999) := 7369;
   --
   l_employee_values (100) := -77;
   l_employee_values (200) := 99999999;
   --
   FORALL l_index IN VALUES OF l_employee_values
      UPDATE employee
         SET salary = 10000
       WHERE employee_id = l_employees (l_index);
END;
```

10g_values_of.sql

The Wonder Of Table Functions

- A table function is a function that you can call in the FROM clause of a query, and have it be treated as if it were a *relational table.*
- Table functions allow you to perform arbitrarily complex transformations of data and then make that data available through a query.
  - Not everything can be done in SQL.
- Combined with REF CURSORs, you can now more easily transfer data from within PL/SQL to host environments.
  - Java, for example, works very smoothly with cursor variables

Building a table function

- A table function must return a nested table or varray based on a schema-defined type.
  - Types defined in a PL/SQL package can only be used with pipelined table functions.
- The function header and the way it is called must be SQL-compatible: all parameters use SQL types; no named notation.
  - In some cases (streaming and pipelined functions), the IN parameter must be a cursor variable -- a query result set.

Simple table function example

- Return a list of names as a nested table, and then call that function in the FROM clause.

```
CREATE OR REPLACE FUNCTION lotsa_names (
   base_name_in IN VARCHAR2, count_in IN INTEGER
)
   RETURN names_nt
IS
   retval   names_nt := names_nt ();
BEGIN
   retval.EXTEND (count_in);

   FOR indx IN 1 .. count_in
   LOOP
      retval (indx) :=
         base_name_in || ' ' || indx;
   END LOOP;

   RETURN retval;
END lotsa_names;
```

```
SELECT column_value
  FROM TABLE (
         lotsa_names ('Steven'
               , 100)) names;

COLUMN_VALUE
------------
Steven 1
...
Steven 100
```

tabfunc_scalar.sql

Streaming data with table functions

- You can use table functions to "stream" data through several stages within a single SQL statement.
  - Example: transform one row in the stocktable to two rows in the tickertable.

tabfunc_streaming.sql

```
CREATE TABLE  stocktable (
  ticker VARCHAR2(20),
  trade_date DATE,
  open_price NUMBER,
  close_price NUMBER
)
/
CREATE TABLE tickertable (
  ticker VARCHAR2(20),
  pricedate DATE,
  pricetype VARCHAR2(1),
  price NUMBER)
/
```

Streaming data with table functions

- In this example, transform each row of the stocktable into two rows in the tickertable.

```
CREATE OR REPLACE PACKAGE refcur_pkg
IS
   TYPE refcur_t IS REF CURSOR
      RETURN stocktable%ROWTYPE;
END refcur_pkg;
/

CREATE OR REPLACE FUNCTION stockpivot (dataset refcur_pkg.refcur_t)
   RETURN tickertypeset ...

BEGIN
   INSERT INTO tickertable
      SELECT *
        FROM TABLE (stockpivot (CURSOR (SELECT *
                                          FROM stocktable)));
END;
/
```

tabfunc_streaming.sql

Use pipelined functions to enhance performance.

```
CREATE FUNCTION StockPivot (p refcur_pkg.refcur_t)
     RETURN TickerTypeSet PIPELINED
```

- Pipelined functions allow you to return data iteratively, asynchronous to termination of the function.
  - As data is produced within the function, it is passed back to the calling process/query.
- Pipelined functions can only be called within a SQL statement.
  - They make no sense within non-multi-threaded PL/SQL blocks.

Applications for pipelined functions

- Execution functions in parallel.
  - In Oracle9i Database Release 2 and above, use the PARALLEL_ENABLE clause to allow your pipelined function to participate fully in a parallelized query.
  - Critical in data warehouse applications.
- Improve speed of delivery of data to web pages.
  - Use a pipelined function to "serve up" data to the webpage and allow users to being viewing and browsing, even before the function has finished retrieving all of the data.

Piping rows out from a pipelined function

Add PIPELINED keyword to header

Pipe a row of data back to calling block or query

RETURN...nothing at all!

```
CREATE FUNCTION stockpivot (p refcur_pkg.refcur_t)
   RETURN tickertypeset
   PIPELINED
IS
   out_rec   tickertype  :=
       tickertype (NULL, NULL, NULL);
   in_rec    p%ROWTYPE;
BEGIN
   LOOP
      FETCH p INTO in_rec;
      EXIT WHEN p%NOTFOUND;
      out_rec.ticker := in_rec.ticker;
      out_rec.pricetype := 'O';
      out_rec.price := in_rec.openprice;

      PIPE ROW (out_rec);
   END LOOP;
   CLOSE p;

   RETURN;
END;
```

tabfunc_setup.sql
tabfunc_pipelined.sql

Enabling Parallel Execution

- You can use pipelined functions with the Parallel Query option to avoid serialization of table function execution.
- Include the PARALLEL_ENABLE hint in the program header.
  - Choose a partition option that specifies how the function's execution should be partitioned.
  - "ANY" means that the results are independent of the order in which the function receives the input rows (through the REF CURSOR).

```
{[ORDER | CLUSTER] BY column_list}
PARALLEL_ENABLE ({PARTITION p BY
   [ANY | (HASH | RANGE) column_list]} )
```

Table functions - Summary

- Table functions offer significant new flexibility for PL/SQL developers.
- Consider using them when you...
  - Need to pass back complex result sets of data through the SQL layer (a query);
  - Want to call a user defined function inside a query and execute it as part of a parallel query.

Top Tip: Stop writing so much SQL!

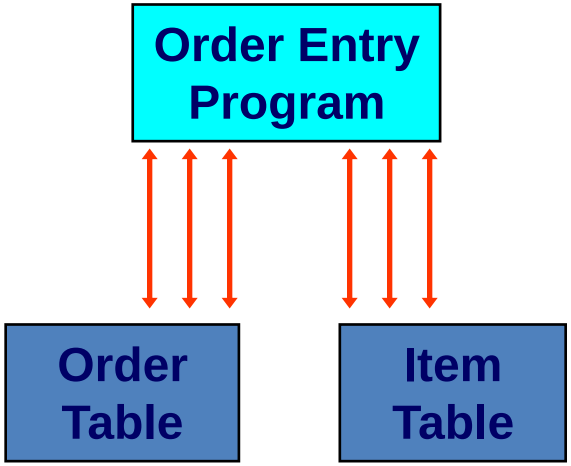
"Why does Steven make such a big deal about writing SQL inside PL/SQL? It's a **no-brainer** in PL/SQL, the *last* thing we have to worry about!"

- I moan and groan about SQL because it is the "Achilles Heel" of PL/SQL.
  - It's so easy to write SQL, it is *too* easy.
  - The result is that most programmers take SQL totally for granted and write it whenever and wherever they need. Bad idea!
  - Let's see why I say this....

Why We Write PL/SQL Code

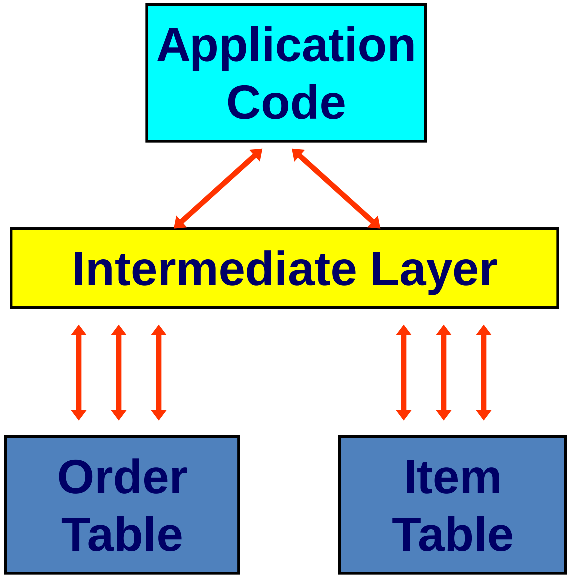
- PL/SQL is an embedded language. Its purpose is to provide high-speed, easy access to underlying datasets.
- Those datasets are *always* changing – both the data and the structure of the tables.
  - So the code should not get in the way of that change, but should support and enable that change.

Bottom line: if everyone writes SQL whenever and wherever they want to, it is *very* difficult to maintain and optimize the code.



Single Point of (SQL) Robust Definition

- General principle: figure out what is volatile and then hide that stuff behind a layer of code to minimize impact on application logic.
- For SQL: if the same statement appears in multiple places in application code, very difficult to maintain and optimize that code.
- So we should avoid repetition at all costs!
- But how….???



How to Avoid SQL Repetition

- You should, as a rule, not even *write* SQL in your PL/SQL programs
  - **You can't repeat it if you don't write it**
- Instead, rely on pre-built, pre-tested, written-once, used-often PL/SQL programs.
  - **"Hide" both individual SQL statements and entire transactions.**
  - **And *revoke privileges on tables*!**

About comprehensive table APIs

- Many (not all!) of the SQL statements we need to write against underlying tables and views are very common and predictable.
  - Get me all rows for a foreign key.
  - Get me one row for a primary key.
  - Insert a row; insert a collection of rows.
- Why write these over and over? Instead, rely on a standard, *preferably generated,* programmatic interface that takes care of this "basic plumbing."

SOA for PL/SQL Developers!
SQL is a *service*.
Error mgt is a *service*.

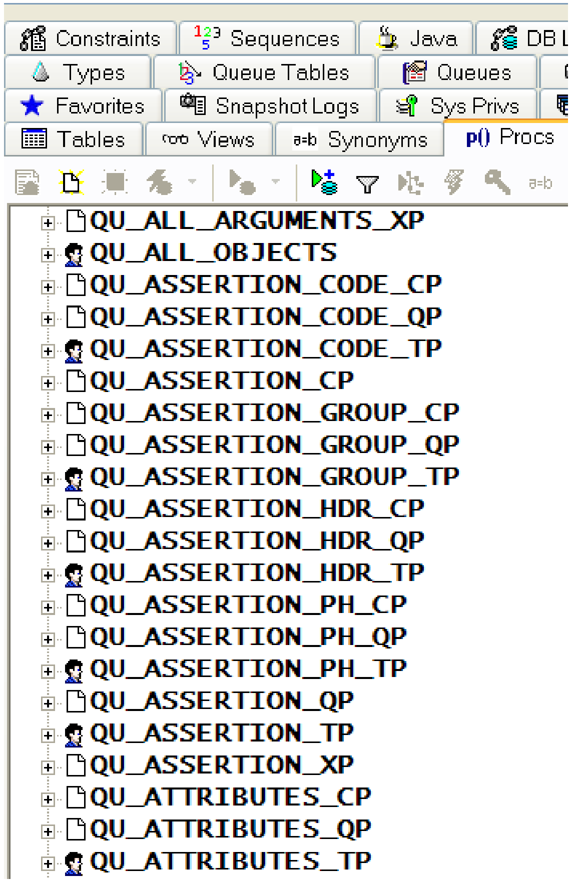
The Quest CodeGen Utility
www.qcgu.net

Clear benefits of encapsulated SQL

- Change/improve implementation without affecting application layer of code.
  - Switch between types of queries (implicit vs explicit)
  - Take advantage of data caching, bulk processing, SQL enhancements like MERGE.
- Consistent error handling
  - INSERT: dup_val_on_index?
  - SELECT: too_many_rows?
  - Much less likely to be ignored when the developer writes SQL directly in the application.

Example: Quest Code Tester backend

- For each table, we have three generated packages:
  - <table>_CP for DML
  - <table>_QP for queries
  - <table>_TP for types
- And for many an "extra stuff" package with custom SQL logic and related code:
  - <table>_XP



```
QU_ALL_ARGUMENTS_XP
QU_ALL_OBJECTS
QU_ASSERTION_CODE_CP
QU_ASSERTION_CODE_QP
QU_ASSERTION_CODE_TP
QU_ASSERTION_CP
QU_ASSERTION_GROUP_CP
QU_ASSERTION_GROUP_QP
QU_ASSERTION_GROUP_TP
QU_ASSERTION_HDR_CP
QU_ASSERTION_HDR_QP
QU_ASSERTION_HDR_TP
QU_ASSERTION_PH_CP
QU_ASSERTION_PH_QP
QU_ASSERTION_PH_TP
QU_ASSERTION_QP
QU_ASSERTION_TP
QU_ASSERTION_XP
QU_ATTRIBUTES_CP
QU_ATTRIBUTES_QP
QU_ATTRIBUTES_TP
```

qu_outcome_xp.qu_outcomes
qu_outcome_xp.int_create_outcomes

Let’s correct that “wrong code.”

- Page 1 of 2

```
CREATE OR REPLACE PROCEDURE adjust_compensation (
   department_id_in IN employees.department_id%TYPE
)
IS
   TYPE id_aat IS TABLE OF employees.employee_id%TYPE
      INDEX BY PLS_INTEGER;

   l_employee_ids   id_aat;

   TYPE salary_aat IS TABLE OF employees.salary%TYPE
      INDEX BY PLS_INTEGER;

   l_salaries       salary_aat;

   TYPE fullname_aat IS TABLE OF employee_rp.fullname_t
      INDEX BY PLS_INTEGER;

   l_fullnames      fullname_aat;
BEGIN
```

wrong_code.sql

Let’s correct that “wrong code.”

Page 2 of 2

```
BEGIN
   SELECT employee_id, salary, employee_rp.fullname (first_name, last_name)
   BULK COLLECT INTO l_employee_ids, l_salaries, l_fullnames
     FROM employees
    WHERE department_id = department_id_in;

   FOR indx IN 1 .. l_employee_ids.COUNT
   LOOP
      analyze_compensation (l_employees (indx).employee_id
                          , l_employees (indx).salary
                           );
   END LOOP;

   FORALL indx IN 1 .. l_employees.COUNT
      UPDATE employees
         SET ROW = l_employees (indx).salary
       WHERE employee_id = l_employees (indx).employee_id;
END adjust_compensation;
```

wrong_code.sql

Optimizing SQL in PL/SQL: Think "services" and leverage key features!

- Don't take SQL for granted.
  - Just because it's easy, doesn't mean it's not significant.
- Hide SQL behind an API: serve up the SQL via procedures and functions.
- Take advantage of key features to improve performance and usability.
  - BULK COLLECT, FORALL, table functions