Oracle PL/SQL Advanced

Features

www.lingaro.com

lingaro

The Value of Business Intelligence

Training Notes

- Course contains Lingaro PL/SQL Standard rules
  - Rule is indicated by blue stars
  - More stars means more important rule

★★★
★★
★

Training Agenda

- Performance and Tuning
- Other Compilation Flags
- Schema Created Types
- Cursor Variables
- Dynamic SQL
- PL/SQL in Data Warehouse
- Miscellany PL/SQL Features
- Profiling and Tracing

Topic Agenda

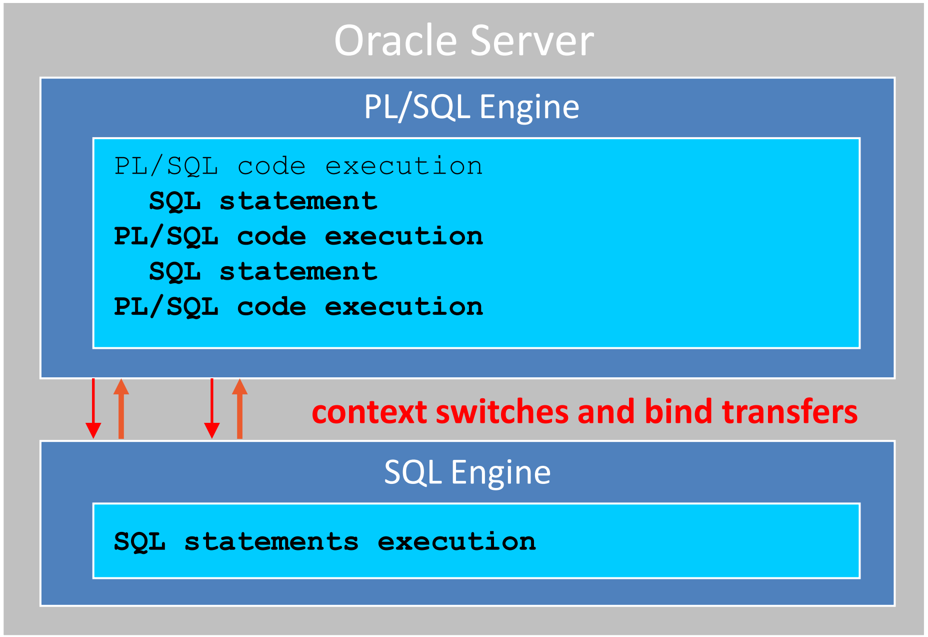
Performance and Tuning

- Code Tuning
  - Context Switches
  - Bulk Binds
  - Data Type Conversion
  - Constraint Issues
  - PL/SQL Engine Data Types
  - Smaller Executable Sections
  - Comparing SQL with PL/SQL
  - NOCOPY Hint
  - Rephrasing Conditional Statements
- Compilation Flags
  - Compilation Code Type
  - Intraunit Inlining
  - PL/SQL Compiler Optimizer
- Avoiding Memory Overhead

Code Tuning

Context Switches

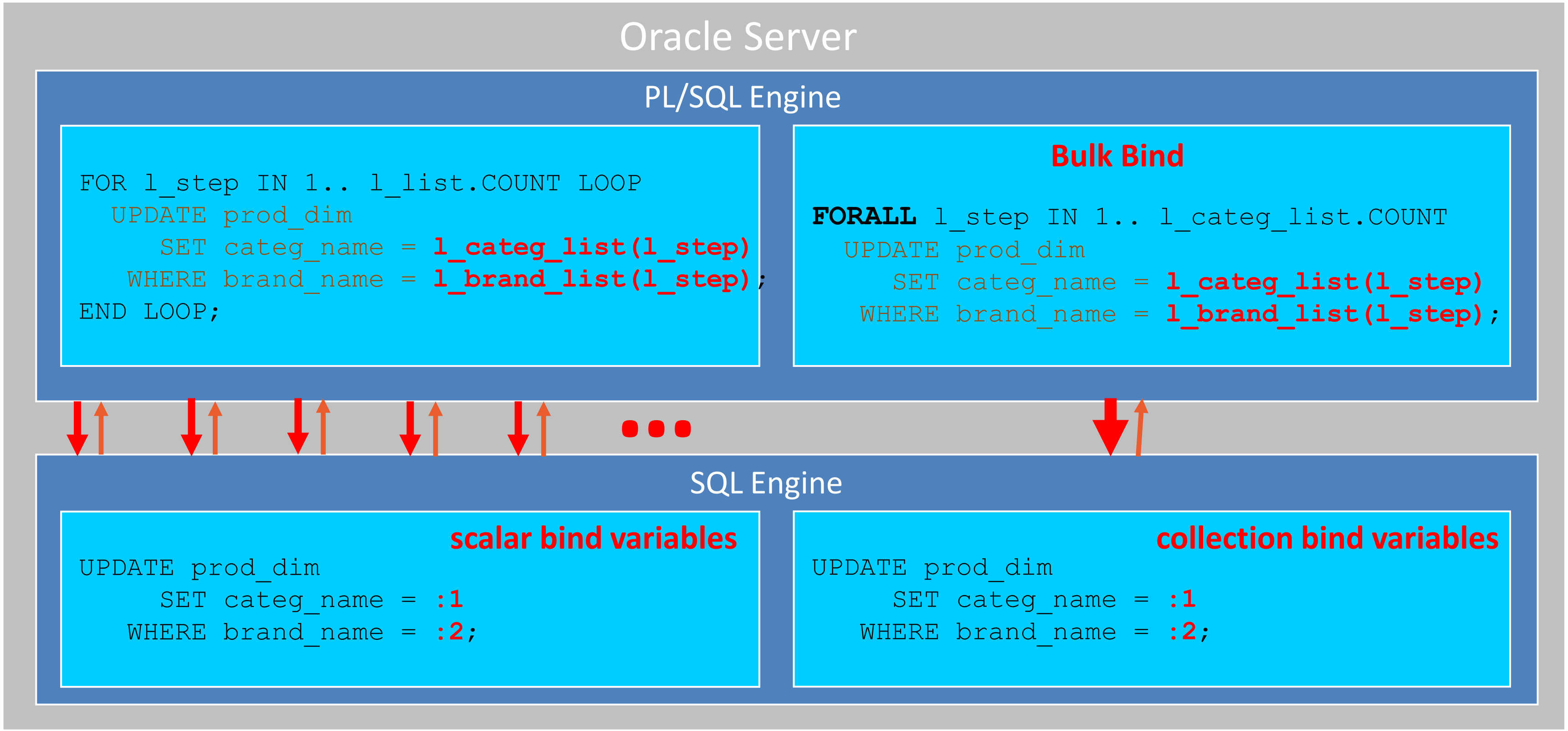
- PL/SQL engine needs SQL engine to execute SQL statements
- Before and after SQL execution between SQL and PL/SQL engines
  - context is switched
  - bind variables transferred (if used)
- Switch generates cost
  - Can be in loops
- Less switches save performance
- One SQL in PL/SQL
  - Often not possible
  - Not always optimal
  - Can be overkill



Code Tuning

Bulk Binds

- Use bulk binds to reduce context switches between engines ★ ★ ★
- FORALL keyword



Oracle Server

PL/SQL Engine

```
FOR l_step IN 1.. l_list.COUNT LOOP
  UPDATE prod_dim
     SET categ_name = l_categ_list(l_step)
   WHERE brand_name = l_brand_list(l_step);
END LOOP;
```

Bulk Bind

```
FORALL l_step IN 1.. l_categ_list.COUNT
  UPDATE prod_dim
     SET categ_name = l_categ_list(l_step)
   WHERE brand_name = l_brand_list(l_step);
```

•••

SQL Engine

scalar bind variables

```
UPDATE prod_dim
     SET categ_name = :1
   WHERE brand_name = :2;
```

collection bind variables

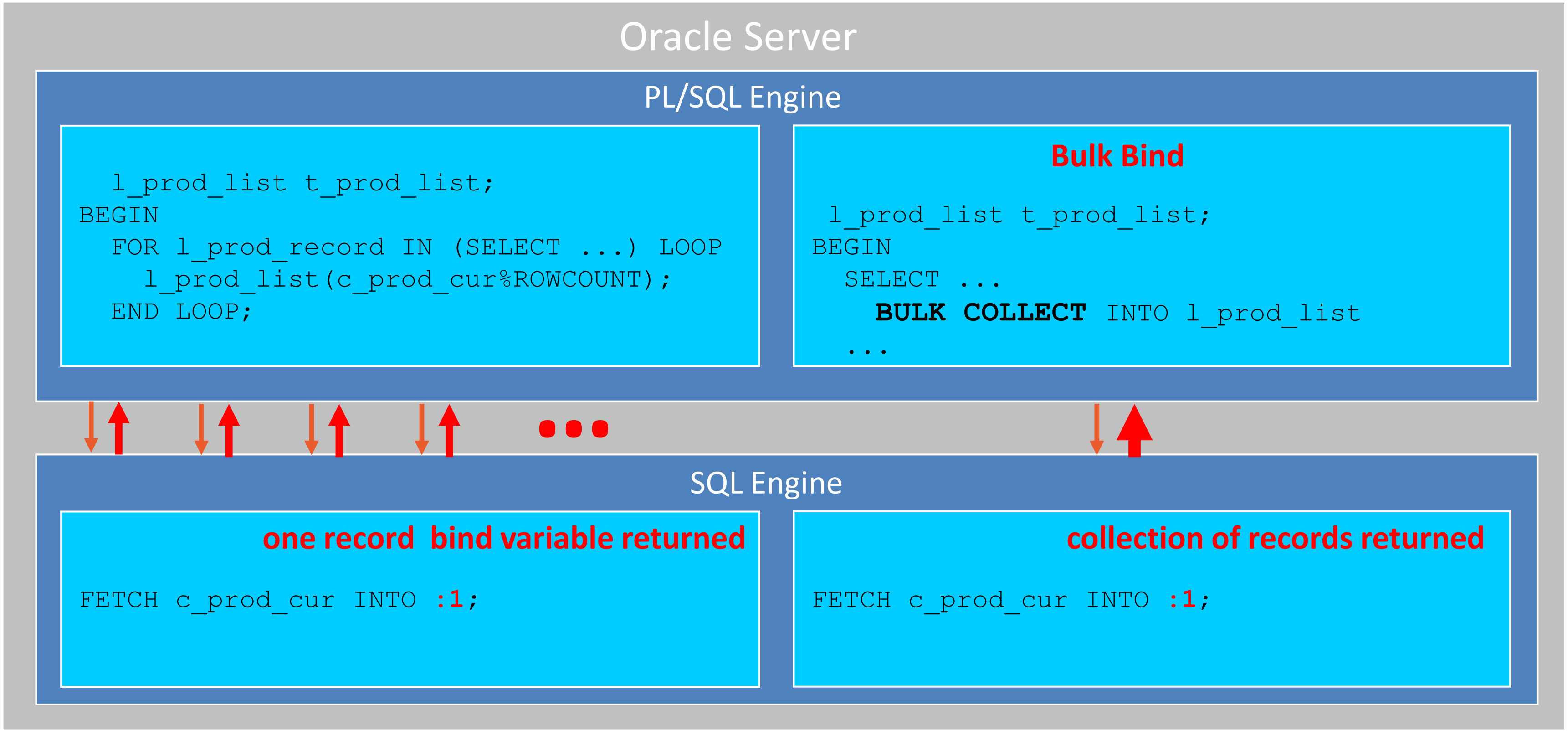
```
UPDATE prod_dim
     SET categ_name = :1
   WHERE brand_name = :2;
```

Code Tuning

Bulk Binds

- BULK COLLECT keyword
- Used to fill collection in one step

```
TYPE t_prod_list IS TABLE OF prod_dim%ROWTYPE INDEX BY PLS_INTEGER;
```


Oracle Server
PL/SQL Engine
Bulk Bind
SQL Engine
one record bind variable returned
collection of records returned

```
  l_prod_list t_prod_list;
BEGIN
  FOR l_prod_record IN (SELECT ...) LOOP
    l_prod_list(c_prod_cur%ROWCOUNT);
  END LOOP;
```

```
  l_prod_list t_prod_list;
BEGIN
  SELECT ...
    BULK COLLECT INTO l_prod_list
  ...
```

```
FETCH c_prod_cur INTO :1;
```

```
FETCH c_prod_cur INTO :1;
```

Code Tuning

Data Type Conversion

- Avoid implicit data type conversion ★★★
  - PL/SQL performs implicit conversions between structurally different data types.
  - Example: When assigning a `PLS_INTEGER` variable to a `NUMBER` variable

```
DECLARE
  l_num NUMBER;
BEGIN
  l_num := l_num + 15.0;    ← not converted
  l_num := l_num + 15;      ← implicit converted
...
END;
```

- Use explicit conversion if conversion is needed

```
DECLARE
  l_date DATE;
BEGIN
  l_date := TO_DATE('10-JUN-2014', 'DD-MON-YYYY');
  l_date := '10-JUN-2014';          ← implicit converted
...
END;
```

Code Tuning

Constraint Issues

- NOT NULL constraints are checked on each assignment
  - Remove NOT NULL from variable declaration
  - Test NULL only ones on the end

```
DECLARE
  l_tot_sales NUMBER ~~NOT NULL~~ := 0;
BEGIN
  LOOP
    ...
    l_tot_sales := l_tot_sales + ...;
    ...
  END LOOP
  IF l_tot_sales IS NOT NULL THEN
      pro_update_sales(in_tot_sales => l_tot_sales);
  ...
END;
```

Code Tuning

Constraint Issues

- SIMPLE_INTEGER
  - Is a predefined subtype
  - Does not include a null value
  - Range from -2147483648 to 2147483648
  - **Eliminates the overhead of overflow checking**
  - Is allowed anywhere in PL/SQL where the PLS_INTEGER type
  - Up to 10 times faster when compared with the PLS_INTEGER
  - Use it instead of PLS_INTEGER but only if this issue is not a problem

```
l_cnt SIMPLE_INTEGER := 2147483648;
l_cnt := l_cnt + 1;     ← value is -2147483648 after overflow
```

  - Beware of silent overflow

```
l_cnt SIMPLE_INTEGER; ← PLS-00218: ... NOT NULL must have an initialization assignment
l_cnt := NULL; ← PLS-00382: expression is of wrong type
```

  - PLS_INTEGER is checking range constraint

```
l_cnt PLS_INTEGER := 2147483648;
l_cnt := l_cnt + 1; ← ORA-01426: numeric overflow
```

Code Tuning

PL/SQL Engine Data Types

- Use PL/SQL only types instead of SQL supported types
  - `BOOLEAN` instead of `VARCHAR2(1)`
  - `PLS_INTEGER` or `SIMLE_INTEGER` instead of `INTEGER` or `NUMBER`
  - `BINARY_FLOAT` and `BINARY_DOUBLE` instead of `NUMBER`
  - `RECORD` instead of `OBJECT`
  - `TABLE OF ... INDEX BY` associated arrays instead of `TABLE OF` collections
- Are faster and more efficient in PL/SQL engine than SQL types
- Require less memory
- Modified by machine arithmetic faster than library arithmetic
- Exception - variables used as SQL statement bind variables
  - To avoid implicit type conversion

```
SQL> CREATE TABLE task_prc (..., run_cnt NUMBER(38), ...);

l_run_cnt task_prc.run_cnt%TYPE  ← NUMBER(38)
...
FOR ...
  SELECT run_cnt INTO l_run_cnt
FROM task_prc
...
```

Code Tuning

Smaller Executable Sections

- Modularizing Your Code
  - Limit the number of lines between a `BEGIN` and `END` to 60 (without comments).
  - Place business process and rule discrete chunks separately.
  - Avoid code redundancy.
  - Reuse code whenever possible.
  - Use packaged programs to keep each executable section small.
  - Use local procedures and functions to hide process logic.
  - Use a functions to hide formulas and business rules.

Code Tuning

Comparing SQL with PL/SQL

- SQL
  - Accesses data using SQL engine in the database – less context switches
  - Treats data as sets – best for processing large number of rows
  - Can be not enough if logic is complicated but use SQL if possible and efficient
    - MODEL, PIVOT, UNPIVOT clauses
    - Analytical functions
    - ROLLUP, CUBE, GROUPING SETS clauses
    - DML Error logging
    - MERGE, multitable INSERT, RETURNING clause
  - SQL engine functions are more efficient than from "standard" package
  - Never use PL/SQL for action which SQL can do

```
FOR l_record IN (SELECT ... FROM source_fct WHERE ...) LOOP
  INSERT INTO destination_fct (...) VALUES l_record;
END LOOP
```

```
SQL> INSERT INTO destination_fct (...)
SQL> SELECT ... FROM source_fct WHERE ...;
```

Code Tuning

Comparing SQL with PL/SQL

- PL/SQL benefits
  - Provides procedural capabilities
  - Has additional functionalities
    - ✓ Dynamic SQL,
    - ✓ Exception handling
  - Has more flexibility built into the language so can be used to
    - ✓ Complex SQL logic simplification,
    - ✓ SQL statement retry,
    - ✓ Fine grained security,
    - ✓ Automate correlated actions,
    - ✓ Implement diagnostic and logging
  - Use PL/SQL when

    SQL can’t do it at all – e.g. dynamic SQL needed

    SQL is to complicated or inefficient
  - Avoid SQL inside loops
  - COMMIT and ROLLBACK are better from API layer

Code Tuning

Comparing SQL with PL/SQL

- PL/SQL benefit example
  - Avoid each execution parse CPU load using bind variables ★★★

```
SQL> INSERT INTO task_prc (taks_skid, taks_name, ...)
       VALUES ( 421, ‘LOAD TABLE’, ...);
```

```
PROCEDURE pro_add_task(in_skid IN NUMBER,
                       in_name IN VARCHAR2, ...)
IS
BEGIN
  INSERT INTO task_prc (taks_skid, taks_name, ...)
    VALUES (in_skid, in_name, ...);
END;

pro_add_task(in_skid => 421,
             in_name => ‘LOAD TABLE’,
             ...);
```

Code Tuning

Guidance For Warehouse PL/SQL Code

1. Use SQL whenever you can
2. When SQL can't do the job use PL/SQL but SQL within it
3. If 1 and 2 won't do COLLECTIONS and BULK PROCESSING
4. Still can't do it? Consider pipelined functions or Java

Code Tuning

NOCOPY Hint

- IN OUT parameters are delivered into subprograms as copy
- Use NOCOPY to avoid copy and additional memory costs
- Argument will be partially modified after subprogram error

```
DECLARE
   TYPE t_prod_list IS TABLE OF prod_dim%ROWTYPE INDEX BY PLS_INTEGER;
   l_prod_list t_prod_list;
   PROCEDURE pro_map_prod(in_out_prod_list IN OUT NOCOPY t_prod_list)
   ...
BEGIN
  SELECT *
    INTO l_prod_list;
    FROM prod_dim
    WHERE
  pro_map_prod(in_out_prod_list => l_prod_list)
  FORALL l_step IN 1.. l_prod_list.COUNT
    UPDATE SET ROW = l_prod_list(l_step)
      WHERE prod_skid = l_prod_list(l_step).prod_skid;
END;
```

Code Tuning

Rephrasing Conditional Statements

- In logical expressions, PL/SQL stops evaluating the expression
  - as soon as the result is determined
  - order of evaluation is from left to right
- Put logical expression parts in correct order

```
            mostly TRUE
   IF (l_categ_cnt > 10)  OR (l_sales_target_amt IS NOT NULL) THEN
      ...
      ...
   END IF;

             mostly FALSE
   IF (l_sales_tot < 5000) AND fn_cust_valid_ind(cust_skid) THEN
     ...
   END IF;
```

Code Tuning

Rephrasing Conditional Statements

- Use the `ELSIF` syntax for mutually exclusive clauses

```
IF (l_acct_mgr = 145)
THEN
  process_acct_145;
ELSIF (l_acct_mgr = 147) THEN
  pro_prcss_acct_147;
ELSIF (l_acct_mgr = 148) THEN
  pro_prcss_acct_148;
ELSIF (l_acct_mgr = 149) THEN
  pro_prcss_acct_149;
END IF;
```

```
IF (l_acct_mgr = 145) THEN
  pro_prcss_acct_145;
END IF;
IF (l_acct_mgr = 147) THEN
  pro_prcss_acct_147;
END IF;
IF (l_acct_mgr = 148) THEN
  pro_prcss_acct_148;
END IF;
IF (l_acct_mgr = 149) THEN
  pro_prcss_acct_149;
END IF;
```

Function Result Caching

- Add DETERMINISTIC keyword into function if function
  - Is slow
  - Executed many times e.g. for each row from in query
  - Returns limited number of values
  - Return same value if used same parameter value - deterministic
- Returned values will be cached and reused on session

```
FUNCTION fn_factorial(in_val INTEGER) RETURN NUMBER DETERMINISTIC IS
```

- RESULT_CACHE keyword uses cache shared between sessions
  - Can be more efficient if function results are shared between session
  - Can be used even results are queried from table inside function
  - Can be problematic if result cache is to small or to excessive used

```
FUNCTION fn_tax(in_salary NUMBER)
  RETURN NUMBER RESULT_CACHE RELIES ON (tax_pct_prc) IS
```

- BEWARE !
  - Incorrect cachce usage can lead to wrong results or performance problems

Compilation Flags

- Get from database dictionary

```
  SQL> SELECT name, type,
              plsql_code_type, plsql_optimize_level
              plsql_debug, plsql_warnings, nls_length_semantics
              plsql_ccflags, plscope_settings
         FROM user_plsql_object_settings;
```

- ALTER SESSION SET *flags_parameter* = *value*;

  ALTER … COMPILE;

- ALTER … COMPILE *flags_parameter* = *value* REUSE SETTING;

```
  PLSQL_CODE_TYPE = INTERPRETED|NATIVE
  PLSQL_OPTIMIZE_LEVEL = 0|1|2|3
  PLSQL_DEBUG = TRUE;
  PLSQL_WARNINGS = 'ENABLE:SEVERE','DISABLE:PERFORMANCE', 'ERROR:05003';
  NLS_LENGTH_SEMANTICS = 'CHAR‘;
  PLSQL_CCFLAGS = ‘platform:ADW3U‘
  PLSCOPE_SETTINGS='IDENTIFIERS:ALL';
```

- To recompile procedure without flags modification

```
  SQL> ALTER PROCEDURE pro_load_tbl COMPILE REUSE SETTINGS;
```

Compilation Flags

Compilation Code Type - PLSQL_CODE_TYPE parameter

- INTERPRETED
  - Used by default
  - Units are compiled to PL/SQL bytecode format
  - Bytecode interpreter overhead during execution
  - Faster compilation - good for development environment
- NATIVE
  - `PLSQL_CODE_TYPE = NATIVE`
  - Units are compiled to platform native CPU code
  - Execution up to 60% faster (extremely faster if not contains SQLs)
  - No improvement for SQL and low for complex mathematical expressions
  - Compilation takes longer
  - No debugging symbols generated
  - Use native compilations on production servers if
    - ✓ Unit have complex logic but almost no SQL statements
    - ✓ Beware of context switches

Compilation Flags

Intraunit Inlining

- Replacing local subprogram call with subprogram body
- Intelligent PL/SQL compiler decision to inline if subprogram
  - Contains small static logic
  - Less parameterized
  - Frequently used
- Less subprogram call overhead but larger main body
- Performance better up to 30%
- Part of PL/SQL compiler optimizer
- PLSQL_OPTIMIZE_LEVEL = 0 or 1
  - Turned off
- PLSQL_OPTIMIZE_LEVEL = 2
  - Used only when inline directive statement is used just before subprogram call

```
PRAGMA INLINE (pro_unit, 'YES');
pro_unit(…);
```

- PLSQL_OPTIMIZE_LEVEL = 3
  - Automatically used

Compilation Flags

PL/SQL Compiler Optimizer

- Code modification during compilation to gain better performance
- PLSQL_OPTIMIZE_LEVEL = 0
  - no modification
- PLSQL_OPTIMIZE_LEVEL = 1
  - elimination of redundant and unnecessary computations and exceptions

```
   l_dead_var  := 0;
```

← e.g. remove not used later variable assignment from loop

  - not move source code out of its original source order
- PLSQL_OPTIMIZE_LEVEL = 2
  - restructure source code, refactoring code relatively far from its original look
- PLSQL_OPTIMIZE_LEVEL = 3
  - automatic inlining before code optimization

Compilation Flags

Compiler Optimization Example - Before

```
PROCEDURE pro_small(in_num2)
IS
  l_num1 NUMBER;
  l_num3 NUMBER;

  PROCEDURE pro_touch(in_out_x IN OUT NUMBER,
                      in_y         IN NUMBER)
  IS
  BEGIN
    IF (in_y > 0) THEN
      in_out_x := in_out_x * in_out_x;
    END IF;
  END;

BEGIN
  l_num1 := in_num2;
  FOR l_step IN 1..10 LOOP
    pro_touch(in_out_x => l_num1, in_y => -17);
    l_num3 := l_num1 * in_num2;
  END LOOP;
  ...
END pro_small;
```

Compilation Flags

Compiler Optimization Example - Steps

After inlining

```
l_num1 := in_num2;
FOR l_step IN 1..10 LOOP
   IF (-17 > 0) THEN
     l_num1 := l_num1 * l_num1;
   END IF;
   l_num3 := l_num1 * in_num2;
END LOOP;
```

Optimization steps

```
l_num1 := in_num2;
FOR l_step IN 1..10 LOOP ...
     IF false THEN
       l_num1 := l_num1 * l_num1;
     END IF;
     l_num3 := l_num1 * in_num2;
END LOOP;

l_num1 := in_num2;
FOR l_step IN 1..10 LOOP ...
  l_num3 := l_num1 * in_num2;
END LOOP;

l_num1 := in_num2;
l_num3 := l_num1 * in_num2;
FOR l_step IN 1..10 LOOP ...
END LOOP;

l_num3 := in_num2 * in_num2;
```

Avoiding Memory Overhead

★★★

- Declare VARCHAR2 variables of 4000 or more characters
  - It save memory – oracle allocates exactly is needed during assignment
  - Less then 4000 characters declaration allocate full length at variable initialization
- Group related subprograms into packages
- Pin large units in the shared pool

```
      SQL> SELECT owner,name,type,sharable_mem
             FROM v$db_object_cache
            WHERE kept=‘NO‘ AND sharable_mem > 10000;
      …
      SQL> exec dbms_shared_pool.keep(‘pkg_appl_log’,’P’);
```

P - Package, Procedure or Function. This is the default value.
T - Type. R - Trigger. Q - Sequence. C - Cursor

- Apply Advice of Compiler Warnings

Topic Agenda

Other Compilation Flags

- Compilation Time Warnings
- Conditional Compilation

Compilation Time Warnings

- Disabled by default
  - Only error messages generated during compilation
- Enabled per unit before or during compilation

```
PLSQL_WARNINGS = 'enable:severe',
                 'enable:performance',
                 'disable:informational',
                 'error:05003',
                 'disable:07203';
```

5000-5999 for severe
6000-6249 for informational
7000-7249 for performance

- Displayed same way as error messages

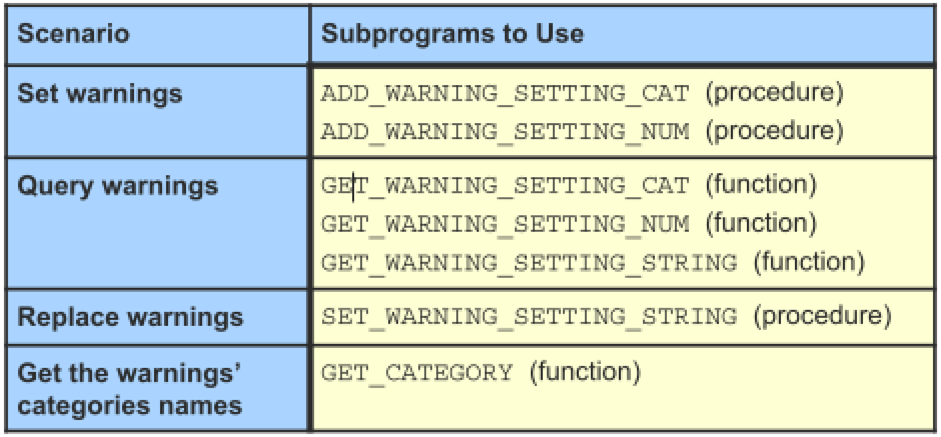
```
SQL> SHOW ERRORS
SQL> SELECT ... FROM user_errors ...
```

- Example

```
PLW-07203: Parameter 'IO_TBL' may benefit from use of the NOCOPY
compiler hint
```

Compilation Time Warnings

- Benefits
  - Make programs more robust and avoid problems at run time
  - Identify potential performance problems
  - Identify factors that produce undefined results
- Anonymous blocks do not produce any warnings
- DBMS_WARNING



Scenario	Subprograms to Use
Set warnings	ADD_WARNING_SETTING_CAT (procedure) ADD_WARNING_SETTING_NUM (procedure)
Query warnings	GET_WARNING_SETTING_CAT (function) GET_WARNING_SETTING_NUM (function) GET_WARNING_SETTING_STRING (function)
Replace warnings	SET_WARNING_SETTING_STRING (procedure)
Get the warnings' categories names	GET_CATEGORY (function)

Conditional Compilation

Concept

- Uses preprocessor before compilation
  - Preprocessor produces different source code variants conditionally
- Multiple versions of the same program in one source e.g.
  - For different platform (Fastlane, ADW)
  - For different environment (DEV, QA, UAT, PROD)
  - For different Oracle Database versions (10g,11g,11gr2,12c)
- Source code must include preprocessor control tokens

```
$IF, $THEN, $ELSE, $ELSIF, $END, $$, $ERROR
```

- Conditions can use compiler flags or static constants

```
PLSQL_CCFLAGS = 'debug_mode:TRUE,platform:1';
```

Conditional Compilation

Code Example

```
SET SERVEROUTPUT ON

BEGIN
$IF ($$debug_mode=TRUE) $THEN
  dbms_output.put_line('debug is set');
  ...
$ELSE
  dbms_output.put_line('debug is not set');
 ...
$END
$IF ($$platform=1) $THEN
  dbms_output.put_line(‘Fastlane code variant');
 ...
$ELSE
  dbms_output.put_line(‘ADW code variant');
 ...
$END
END;
```

Conditional Compilation

Static Expressions

- Are determined during compilation time
- Conditions must use static expression containing
- Boolean static expressions:

```
TRUE, FALSE, NULL, IS NULL, IS NOT NULL
> , < , >= , <= , = , <>, NOT, AND, OR
```

- PLS_INTEGER static expressions:

```
-2147483648 to 2147483647, NULL
```

- VARCHAR2 static expressions include:

```
||, NULL, TO_CHAR
```

- Static constants (usually in separate package):

```
CREATE OR REPLACE PACKAGE pkg_plsql_flags
IS
  trace_level CONSTANT PLS_INTEGER := 2;
END;
```

Conditional Compilation

Static Expressions

- Oracle provided package with version static constants

```
PACKAGE DBMS_DB_VERSION IS
  VERSION CONSTANT PLS_INTEGER := 11; -- RDBMS version
  RELEASE CONSTANT PLS_INTEGER := 1;  -- RDBMS release number
  ver_le_9_1    CONSTANT BOOLEAN := FALSE;
  ver_le_9_2    CONSTANT BOOLEAN := FALSE;
  ver_le_9      CONSTANT BOOLEAN := FALSE;
  ver le 10 1   CONSTANT BOOLEAN := FALSE;
  ver_le_10_1   CONSTANT BOOLEAN := FALSE;
  ver_le_10_2   CONSTANT BOOLEAN := FALSE;
  ver_le_10     CONSTANT BOOLEAN := FALSE;
  ver_le_11_1   CONSTANT BOOLEAN := TRUE;
  ver_le_11     CONSTANT BOOLEAN := TRUE;
END DBMS_DB_VERSION;
```

```
SUBTYPE t_my_real IS
$IF (dbms_db_version.verion < 10) $THEN
  NUMBER;
$ELSE
  BINARY_DOUBLE;
$END
```

Other Compilation Flags

Conditional Compilation

$$ and $ERROR

- $$ is used to reference flag value
- $ERROR is used to produce preprocessor compilation error

```
SQL> ALTER SESSION SET PLSQL_CCFLAGS = ‘trace_level:1';
```

```
$IF ($$trace_Level = 0) $THEN
  ...;
$ELSIF ($$trace_Level = 1) $THEN
  ...;
$ELSIF ($$trace_Level = 2) $THEN
  ...;
$ELSE
  $ERROR
    'Bad: '||$$trace_Level||’ trace lvl’
  $END
$END
```

Other Compilation Flags

PL/Scope

- Collect information about identifiers declared and used
- Help in program analyze and understand
- Disabled by default
- Enabled per unit before or during compilation

```
PLSCOPE_SETTINGS='identifiers:all';
```

- Identifiers info displayed from database dictionary view

```
USER_IDENTIFIERS
```

NAME – identifier name

TYPE – identifier type e.g. CONSTANT, PACKAGE, RECORD …

SIGNATURE – unique string to distinguish different but same name identifiers

OBJECT_NAME, OBJECT_TYPE – unit containing the identifier

USAGE – **DEFINITION, CALL, DECLARATION, ASSIGNMENT, REFERENCE**

USAGE_ID – identifier sequentially generated integer, unique within its program unit

USAGE_CONTEXT_ID – parent of this identifier appearance

LINE, COL – code location

PL/Scope - Query Examples

- List the identifiers declared

```
SELECT name, signature, type
  FROM user_identifiers
 WHERE usage = 'DECLARATION'
 ORDER BY object_type, usage_id;
```

- List the identifiers declared as CURSOR

```
SELECT name, signature, object_name, type
  FROM user_identifiers
 WHERE usage = 'DECLARATION'
   AND type   = 'CURSOR'
 ORDER BY object_type, usage_id
```

- List the redundant identifiers not used inside the executable section

```
SELECT name, object_name, type, signature
  FROM user_identifiers t
 WHERE usage = 'DECLARATION'
   AND NOT EXISTS (SELECT 1
                     FROM user_identifiers
                    WHERE signature = t.signature
                      AND usage <> 'DECLARATION')
```

- List the actions on a specific identifier

```
SELECT name, object_name, type, usage, line
  FROM user_identifiers t
 WHERE signature = 'C6DC4D2D5770696415F7EC524AFADAE4'
```

Topic Agenda

Schema Created Types

- Overview
- Object
- Nested Table
- Varray
- Collections Additional Features

Stored Type

Overview

- Created, modified and dropped in database using DDL

```
SQL> CREATE OR REPLACE TYPE <name>
            AS <SQL composite type declaration>;
```

- Can only be composite type

```
OBJECT, TABLE, VARRAY
```

- Can be used as SQL type in table column
- Can be used as PL/SQL type (variable, parameter, function)
- Good to avoid type conversion between SQL and PLSQL
- Object privilege needed to use another schema type
- Can't be replaced or dropped if used in database
- Alter support for schema types is limited

Object Type

Overview

- Oracle implementation of **Object Oriented**
  - Modeling
  - Programming
- High-level **abstraction** of real world entity
- Object class is created only as schema type
- Contains the data definition - **attributes**
  - Similar to package variables
- Contains optional functions and procedures – **methods**
  - Similar to package subprograms
  - Data structures along with the methods – **encapsulation**
  - Predefined constructor method used to create object instance
    - ✓ Same name as type name
    - ✓ Can be replaced by custom constructor

Object Type

Example

- Interface
  - similar to package specification (and to RECORD type if has no methods)

```
CREATE OR REPLACE TYPE t_location_obj
AS OBJECT
  (street_name   VARCHAR2(60),
   bldg_num      INTEGER,
   city_name     VARCHAR2(30),
   zip_code      CHAR(10),
   MEMBER FUNCTION  fn_address RETURN VARCHAR2,
   MEMBER PROCEDURE ...
/
```

- Implementation - body
  - Needed if interface include methods declarations
  - Always can be replaced
  - Similar to package body

```
CREATE OR REPLACE TYPE BODY t_location_obj
IS
   MEMBER FUNCTION fn_address RETURN VARCHAR2
   IS
   BEGIN
     RETURN street_name || ‘ ’ || bldg_num || CHR(10) ||
            zip_code || ‘ ’ || city_name;
   END;
   ...
END;
/
```

Object Type

Redefinition

- In interface attributes and methods can be added, altered, dropped

```
SQL> ALTER TYPE t_location_obj ADD ATTRIBUTE cntry_name VARCHAR2(40);
SQL> ALTER TYPE t_location_obj ADD MEMBER
       PROCEDURE pro_set (in_street_name street_name%TYPE,
                          in_bldg_num    bldg_num%TYPE,
                          in_city_name   city_name%TYPE,
                          in_zip_code    zip_code%TYPE);
```

- Body can be simply replaced

```
CREATE OR REPLACE TYPE BODY t_location_obj
...
     RETURN street_name || ` ` || bldg_num || CHR(10) ||
            zip_code || ` ` || city_name || ` ` ||cntry_name;
   END;
   MEMBER PROCEDURE pro_set ...
   IS
   BEGIN
     street_name := in_street_name;
     bldg_num ...
   END;
END;
/
```

Object Type

Usage

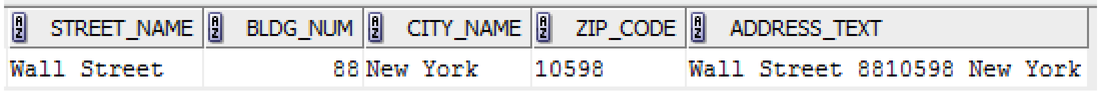
- As table type
  - Create objects table

```
SQL> CREATE TABLE address_obj_prc OF t_location_obj;

SQL> INSERT INTO address_obj_prc (street_name, bldg_num, city_name, zip_code)
          VALUES (‘Wall Street’, 88, ‘New York’, ‘10598’);
```

  - Access row object function methods

```
SQL> SELECT street_name, bldg_num, city_name, zip_code,
            obj.fn_address() address_text
       FROM address_obj_prc obj;
```



STREET_NAME	BLDG_NUM	CITY_NAME	ZIP_CODE	ADDRESS_TEXT
Wall Street	88	New York	10598	Wall Street 8810598 New York

Object Type

Usage

- As column type

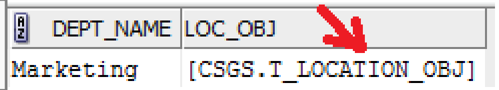
```
SQL> CREATE TABLE dept_prc (dept_name VARCHAR2(100),
                              loc_obj t_location_obj);
```

  - Use constructor to create object instance

```
SQL> INSERT INTO dept_prc (dept_name, loc_obj)
         VALUES ('Marketing', t_location_obj('Wall Street',88,
                  'New York', '10598'));
```

  - Object column value display type/constructor identifier name

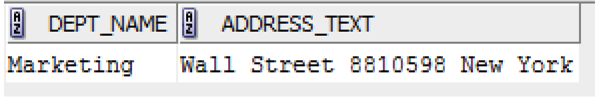
```
SQL> SELECT * FROM dept_prc;
```



DEPT_NAME	LOC_OBJ
Marketing	[CSGS.T_LOCATION_OBJ]

  - Access object column function methods

```
SQL> SELECT dept_name, d.loc_obj.fn_address() address_text
       FROM dept_prc d;
```



DEPT_NAME	ADDRESS_TEXT
Marketing	Wall Street 8810598 New York

Object Type

Usage

- As object attribute type

```
SQL> CREATE OR REPLACE TYPE t_person_obj
     AS OBJECT
        (last_name   VARCHAR2(20),
         first_name  VARCHAR(20),
         birth_date  DATE,
         address_obj t_location_obj,
         MEMBER FUNCTION fn_age RETURN INTEGER)
     NOT FINAL;
     /
```

- Nested table can be collection of objects

```
SQL> CREATE OR REPLACE TYPE t_people_n AS TABLE OF t_person_obj;
     /
```

- Object attribute can be nested table

```
SQL> CREATE OR REPLACE TYPE t_bldg_obj
     AS OBJECT
        (flore_cnt INTEGER,
         residents_n t_people_n,
         address_obj t_location_obj);
     /
```

Object Type

Usage

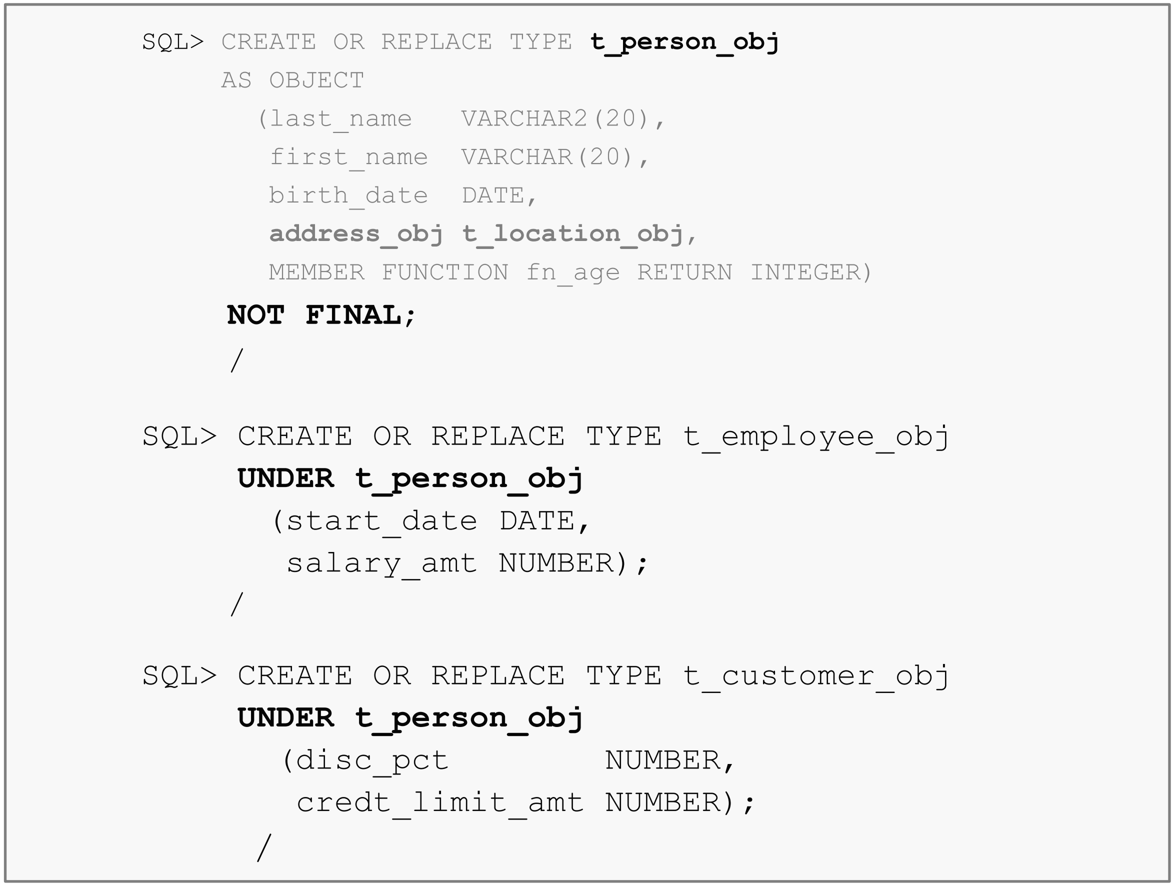
- As PL/SQL type in
  - Variables, parameters and returned by functions
  - Object type can’t be declared in PL/SQL
  - Use constructor to create new object instance
  - Can use object PL/SQL variable as bind variable in SQL command

```
DECLARE
  l_address_obj t_location_obj;
  PROCEDURE pro_add_address(in_address_obj t_location_obj)
  ...
  FUNCTION fn_address(...) RETURN t_location_obj
  ...
BEGIN
  l_address_obj := t_location_obj(street_name => ’Wall Street’, bldg_num => 88,
                                  city_name => ‘New York’, zip_code => ‘10598’,
                                  cntry_name => ‘USA’);
  pro_add_address(l_address_obj);
  l_address_obj.pro_set(in_street_name => l_address_obj.street_name,
                        in_bldg_num    => 11,
                        ...);
  ...
  UPDATE address_obj_prc
      SET ROW = l_address_obj
     WHERE ...
```

Object Type

Inheritance

- Object can **inherit** attributes and methods from parent
  - If parent is **NOT FINALE** (like t_person_obj from previous example)



```
SQL> CREATE OR REPLACE TYPE t_person_obj
     AS OBJECT
       (last_name    VARCHAR2(20),
        first_name   VARCHAR(20),
        birth_date   DATE,
        address_obj t_location_obj,
        MEMBER FUNCTION fn_age RETURN INTEGER)
     NOT FINAL;
     /

SQL> CREATE OR REPLACE TYPE t_employee_obj
     UNDER t_person_obj
       (start_date DATE,
        salary_amt NUMBER);
     /

SQL> CREATE OR REPLACE TYPE t_customer_obj
     UNDER t_person_obj
       (disc_pct         NUMBER,
        credt_limit_amt NUMBER);
       /
```

Object Type

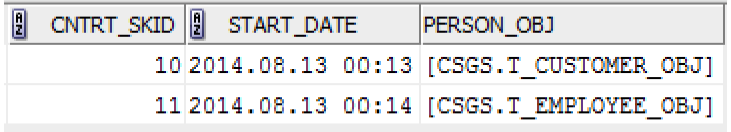
Polimorfizm

```
SQL> CREATE TABLE cntrt_prc
          (cntrt_skid INTEGER,
           start_date DATE,
           person_obj t_person_obj);
```

```
INSERT INTO cntrt_prc (cntrt_skid, start_date, person_obj)
  VALUES (10, SYSDATE,
    t_customer_obj(last_name  => 'Smith',
                   first_name => 'John',
                   birth_date => TO_DATE(…),
        address_obj =>
        t_location_obj(street_name => 'Wall Street',
                       bldg_num    => 88,
                       city_name   => 'New York',
                       zip_code => '10598',
                       cntry_name  => SA'
                       ),
                    disc_pct         => 5,
                    credt_limit_amt => 10000
                   )
          );
```

```
INSERT INTO cntrt_prc (cntrt_skid, start_date, person_obj)
  VALUES (11, SYSDATE,
    t_employee_obj(last_name  => 'White',
                   first_name => 'Tom',
                   birth_date => TO_DATE(…),
        address_obj =>
        t_location_obj(street_name => 'Wall Street',
                       bldg_num    => 99,
                       city_name   => 'New York',
                       zip_code => '10598',
                       cntry_name  => 'USA'
                       ),
                    start_date => SYSDATE,
                    salary_amt => 9000
                   )
          );
```

```
SQL> SELECT * FROM cntrt_prc;
```



CNTRT_SKID	START_DATE	PERSON_OBJ
10	2014.08.13 00:13	[CSGS.T_CUSTOMER_OBJ]
11	2014.08.13 00:14	[CSGS.T_EMPLOYEE_OBJ]

Object Type

REF and DEREF keywords

- Use REF keyword is used to define pointer to object
  - Enables referencing the same object in many places
  - Can be used in table columns type and PL/SQL type
- Use REF and DEREF SQL functions to convert pointer in both directions

```
SQL> CREATE OR REPLACE TYPE t_dept_obj
     AS OBJECT
       (dept_name VARCHAR2(100),
        parent_dept_obj_ref REF t_dept_obj);
     /

DECLARE
  l_dept_obj t_dept_obj;
  l_dept_obj_ref REF t_dept_obj;
BEGIN
  SELECT REF(d)
    INTO l_dept_obj_ref
    FROM dept_prc d
   WHERE d.dept_name='DEPT 1';

  UPDATE dept_prc d
     SET d.parent_dept_obj_ref = l_dept_obj_ref
   WHERE d.dept_name='DEPT 2';

  SELECT DEREF(d.parent_dept_obj_ref)
    INTO l_dept_obj
    FROM dept_prc d
   WHERE d.dept_name='DEPT 2';
END;
/
```

```
SQL> CREATE TABLE dept_prc OF t_dept_obj;

SQL> INSERT INTO dept_prc (dept_name)
       VALUES ('DEPT 1');

SQL> INSERT INTO dept_prc (dept_name)
       VALUES ('DEPT 2');
```

Object Type

Other Features

- Method types (keywords before MEMBER)
  - STATIC - invoked on the object type, not its instances
  - CONSTRUCTOR – custom constructor used to create object instance
  - OVERRIDING – to redefine inherited method
  - NOT INSTANTIABLE – placeholder implemented in child objects

  To put instances in order for comparisons or sorting:
  - ORDER – returns object instances order value
  - MAP - returns value to position instance relative to another instance

    zero – same position,
    negative – before position,
    positive - after position
- SELF keyword – to object self reference

Nested Table

Overview

- Unbounded Collection of scalar or composite elements

```
SQL> CREATE OR REPLACE TYPE t_address_list_n IS TABLE OF t_location_obj;
     /

DECLARE
  TYPE t_num_list IS TABLE OF NUMBER;
  TYPE t_dept_list IS TABLE OF dept_prc%ROWTYPE;
  TYPE t_person_list IS TABLE OF t_person_obj;
  l_num_list t_num_list;
  l_address_list t_address_list_n;
  l_dept_list t_dept_list;
  ...
```

- If contains objects and used in column then create nested table
  - Nested table data are stored out of line

```
SQL> CREATE TABLE cust_prc
     (cust_skid NUMBER,
      address_list t_address_list_n, ...)
     NESTED TABLE address_list STORE AS cust_address_list_n;
```

- Has predefined methods

```
COUNT DELETE EXISTS FIRST LAST PRIOR NEXT EXTEND TRIM
```

Nested Table

Usage in SQL

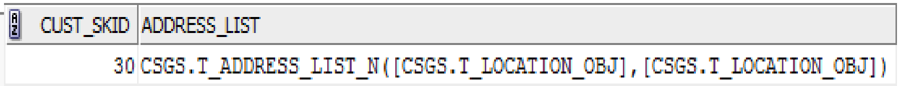
- Insert row with nested rows

```
SQL> INSERT INTO cust_prc (cust_skid, address_list, ...)
       VALUES (30, t_address_list_n(
                      t_location_obj(...), t_location_obj(...), ...
                                   ) , ...);
```

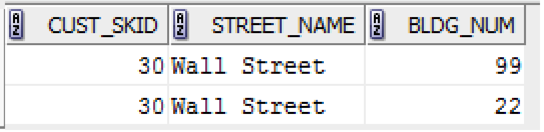
- Quering

```
SQL> SELECT * FROM cust_prc;


SQL> SELECT c.cust_skid, a.street_name, a.bldg_num
       FROM cust_prc c, TABLE(c.address_list) a;
```



CUST_SKID	ADDRESS_LIST
30	CSGS.T_ADDRESS_LIST_N([CSGS.T_LOCATION_OBJ],[CSGS.T_LOCATION_OBJ])



CUST_SKID	STREET_NAME	BLDG_NUM
30	Wall Street	99
30	Wall Street	22

- Piecewise DML

```
INSERT INTO TABLE(SELECT c.address_list FROM cust_prc c WHERE cust_skid = 30)
  VALUES (t_location_obj(...));

UPDATE TABLE(SELECT c.address_list FROM cust_prc c WHERE cust_skid = 30) al
   SET al.cntry_name = 'UNITED STATES'
 WHERE al.cntry_name = 'USA';

DELETE TABLE(SELECT c.address_list FROM cust_prc c WHERE cust_skid = 30) al
  WHERE al.cntry_name = 'POLAND';
```

Nested Table

Usage in PL/SQL

- Must be initialized by constructor or BULK COLLECT INTO clause

```
  ...
  l_num_list t_num_list := t_num_list
    (1, 5, 35, -23, 0, -330);
  l_num_list2 t_num_list;
  l_address_list t_address_list_n;
BEGIN
  l_address_list := t_address_list_n(
    t_location_obj(street_name => 'Wall Street', bldg_num => 99,
                   city_name   => 'New York',    zip_code => '10598',
                   cntry_name  => 'USA'),
    t_location_obj(street_name => 'Wall Street', bldg_num => 22,
                   city_name   => 'New York',    zip_code => '10598',
                   cntry_name  => 'USA'));
  SELECT d.loc_obj.bldg_num BULK COLLECT INTO l_num_list2
    FROM dept_prc d WHERE ...
```

Index key	Collection element
1	‘Wall Street’ | 99 | . . .
2	‘Wall Street’ | 22 | . . .
3	. . .

- Collection indexes are automatically assigned
  - during initialization and during EXTEND method execution
  - during initialization starting from 1
  - Increased by 1

Nested Table

Usage in PL/SQL

- Looping on

```
...
FOR l_step IN 1.. l_address_list.COUNT LOOP
  IF (l_address_list(l_step).EXISTS) THEN
    l_city_name := l_address_list(l_step);
    ...
  END IF;
  ...
END LOOP;
...
```

- Increase size and append element

```
...
l_address_list.EXTEND;
l_address_list(l_address_list.LAST) := t_location_obj(...);
...
```

- Decrease size by removing last 2 elements

```
...
l_address_list.TRIM(2);
...
```

VARRAY

VARRAY vs Nested Table

- Has limited number of collection elements

```
SQL> CREATE OR REPLACE TYPE t_address_array IS TABLE(35) OF t_location_obj;
     /
```

- Has additional predefined method - LIMIT - to get defined bound

```
EXIT WHEN (l_step > l_address_array.LIMIT)
...
```

- If nested in table column - stored inline if size < 4000 bytes
  - Otherwise stored as LOB
  - LOB storage always can be declared

```
SQL> CREATE TABLE dept_email_prc ( ... email_list t_email_array)
       VARRAY email_list STORE AS LOB dept_emails_prc_lob;
```

www.lingaro.com

Collections

Additional Features

- Multiset operations
  - Generate collection as result of set operation on 2 collections

    ```
    MULTISET UNION
    MULTISET EXCEPT
    MULTISET INTERSEC
    ```
  - In SQL

```
SELECT collection1_column MULTISET UNION collection2_column FROM ...
```

  - IN PL/SQL

```
l_collection3 := l_collection1 MULTISET UNION l_collection2;
```

  - Multiset Comparison

```
SELECT …
 WHERE collection1_column SUBMULTISET OF collection2_column
    OR t_table_type(...) MEMBER OF collection1_column;
```

- Type filter

```
SELECT ...
 WHERE VALUE(table_alias) IS OF (t_type_name);
```

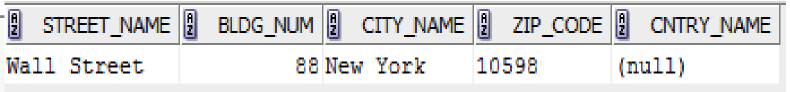
Collections

Additional Features

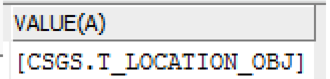
- VALUE function can be also used to:
  - return object

```
SQL> SELECT * FROM address_obj_prc a;

SQL> SELECT VALUE(a) FROM address_obj_prc a;
```



STREET_NAME	BLDG_NUM	CITY_NAME	ZIP_CODE	CNTRY_NAME
Wall Street	88	New York	10598	(null)



VALUE(A)
[CSGS.T_LOCATION_OBJ]

  - update object in table

```
SQL> UPDATE address_obj_prc a SET VALUE(a) = t_location_obj(...)  WHERE ...
```

- COLLECT function - aggregates data into a collection
- CAST function – converts collection type

```
SQL> SELECT CAST(COLLECT(column) AS t_collection1_type)
       INTO l_collection1_variable
       FROM ...
```

- Convert query results into collection

```
SQL> SELECT CAST(MULTISET( SELECT … ) AS t_collection1_type)
       INTO l_collection1_variable
       FROM ...
```

Collections

Additional Features

- Multilevel Collection Types
  - Nested table of nested table type
  - Nested table of varray type
  - Varray of nested table type
  - Varray of varray type
  - Nested table or varray of object that has an attribute as nested table or varray type
- Increasing element size or precision

```
SQL> CREATE TYPE t_email_array AS VARRAY(10) OF VARCHAR2(80);
     /
SQL> ALTER TYPE t_email_array MODIFY ELEMENT TYPE VARCHAR2(100) CASCADE;
```

- Increasing VARRAY Limit Size

```
SQL> ALTER TYPE t_email_array MODIFY LIMIT 100 INVALIDATE;
```

Topic Agenda

Cursor Variables

- Declare
- Using

Declare

- Cursor variable is like static cursor but
  - query is assigned during opening not declaration
  - Query can be changed after close and reopen cursor variable
- Strong type REF CURSOR
  - Can by assigned only to query with the same columns as selected record type
  - This is checked during compilation

```
TYPE t_c_prod_cur IS REF CURSOR RETURN prod_dim%ROWTYPE;
```

- Cursor variable

```
l_c_prod_cur t_c_prod_cur;
```

- Week type is predefined

```
l_c_cur SYS_REFCURSOR;
```

  - It is very flexible variable - cane be assigned to any query
  - Incorrect query assignment is not checked during compilation
  - It can lead to FETCH runtime errors

Usage

- Assign Query with cursor variable in OPEN statement
- FETCH and CLOSE statements are the same as for static cursor
- After CLOSE variable can be used to different query

```
  TYPE t_categ_rec (skid NUMBER,
                    name VARCHAR2);
  TYPE t_categ_cur IS REF CURSOR RETURN t_categ_rec;
  l_categ_cur t_categ_cur;
  l_categ_rec t_categ_rec;
BEGIN
  OPEN l_prod_cur FOR
    SELECT prod_skid, categ_name
      FROM prod_dim;
  LOOP
    FETCH l_prod_cur INTO l_categ_rec;
    ...
  END LOOP
  CLOSE l_prod_cur;
  OPEN l_prod_cur FOR
    SELECT categ_skid, categ_name
      FROM categ_dim;
  ...
```

Usage

- Query can be chosen conditionally

```
IF (l_income_target > in_income_target_limit) THEN
  OPEN l_income_cur FOR
    SELECT categ_name,
           SUM(sales_amt) - SUM(all_costs_amt) AS income_amt
      FROM ...
      GROUP BY categ_name
      ...
ELSE
  OPEN l_sales_cur FOR
    SELECT sub_categ_name,
           SUM(sales_amt) - SUM(basic_costs_amt) AS income_amt
      FROM ...
      GROUP BY sub_categ_name
    ...
```

- Cursor can pass trough REF CURSOR type parameter

```
PROCEDURE pro_open_prod(out_prod_cur OUT t_prod_cur) ...
PROCEDURE pro_check_prod(in_prod_cur IN  t_prod_cur) ...
pro_open_prod(l_prod_cur);
pro_check_prod(l_prod_cur);
```

Considerations

- CURSOR function produces REF cursor from query

```
CREATE FUNCTION fn_num(l_cur SYS_REFCURSOR,
                       mgr_hiredate DATE) RETURN NUMBER
...
```

```
SELECT e1.last_name
  FROM emplee_prc e1
 WHERE fn_num(CURSOR(SELECT e2.hire_date
                       FROM emplee_prc e2
                      WHERE e1.emplee_id = e2.mgr_id),
              e1.hire_date) = 1;
```

- Cursor parameters are good for performance
  - Sharing pointer to same results set between many subprograms
  - Saving memory and CPU
- Cursor variable not need to be closed before reopen
- Cannot be
  - Declared as a public construct of a package
  - Used with FOR … LOOP
  - Used as parameter to subprogram in different database is not possible
  - Used to lock rows using FOR UPDATE clause
  - Stored in the database
  - Used to specify data type for column or collection attribute
  - Assigned the NULL value

Topic Agenda

Dynamic SQL

- Overview
- Dynamic Cursor Variable
- EXECUTE IMMEDIATE

Overview

- Used if SQL text is full or partially unknown before runtime
- Enable to execute SQL code generated by application code
- Compilation is done during runtime (runtime error prone)
- Can lead to problems with
  - Security
  - Performance
  - Reliability
  - Manageability
- Use only if needed ★ ★
- Is very flexible – can remove business logic hardcoding
- Can be used to run unsupported SQL in PL/SQL (like DDL,DCL)
- Dynamic PL/SQL is also available

Dynamic Cursor Variable

- Assign text variable or expression during OPEN … FOR
- Only week type is supported

```
    l_sql_stmt VARCHAR2(32767);
    l_cur SYS_REFCURSOR;
  BEGIN
    OPEN l_cur FOR ‘SELECT (‘ || in_dsply_cols ||
                   ‘) FROM  ‘ || in_tbl_name  ||
                   ‘ WHERE  ‘ || in_filtr;
    OPEN l_cur FOR l_sql_stmt;
```

- Using bind variables

```
    l_sql_stmt VARCHAR2(32767) := ‘SELECT prod_name, salary_amt * :1 ‘ ||
                                  ‘ FROM prod_dim WHERE categ_name = :2 AND ‘ ||
                                  in_filtr;
    l_cur SYS_REFCURSOR;
    ...
  BEGIN
    OPEN l_cur FOR l_sql_stmt USING IN in_ratio, in_categ
    FETCH l_cur INTO l_prod_name, l_salary
```

EXECUTE IMMEDIATE

Overview

- This is native dynamic SQL and PL/SQL statement
  - Better then DBMS_SQL
  - Easier to use
  - Better performance
  - More features
  - Extensions in new database versions
  - Can execute all kind of SQL statements dynamically
- Syntax

```
EXECUTE IMMEDIATE <SQL_or PL/SQL_string>
   [INTO {defined_variable[, defined_variable]... | defined_record}]
   [USING [IN | OUT | IN OUT] bind_argument
       [, [IN | OUT | IN OUT] bind_argument]...];
```

- Is used as implicit cursor

```
EXECUTE IMMEDIATE l_sql_stml;
dbms_output.put_line(‘Number of rows processed is ’ || SQL%ROWCOUNT);
```

EXECUTE IMMEDIATE

Usage

- DDL or DCL in PL/SQL code

```
EXECUTE IMMEDIATE ‘CREATE TABLE ‘ || in_tbl_name ||
                  ‘ (‘ || in_col_list || ‘)‘ || in_options;

EXECUTE IMMEDIATE ‘ALTER SESSION ENABLE PARALLEL DML’;
EXECUTE IMMEDIATE ‘GRANT SELECT ON ‘ || in_tbl_name ||
                  ’ TO ‘ || in_grantee;
```

- Dynamic one row result query

```
EXECUTE IMMEDIATE ‘SELECT ‘ || in_2_cols_list ||
                  ‘  FROM ‘ || in_tbl_name ||
                  ‘ WHERE ‘ || in_filtr
  INTO l_col1, l_col2;
```

- Dynamic INSERT

```
EXECUTE IMMEDIATE ‘INSERT INTO ‘ || l_tbl_name ||
                  ‘ SELECT ...’;
```

EXECUTE IMMEDIATE

Usage

- Bind variables can be used in queries and DMLs

```
      EXECUTE IMMEDIATE ‘SELECT ‘ || in_2_cols_exprn || ‘ * :1’
                        ‘  FROM ‘ || in_tbl_name ||
                        ‘ WHERE categ_name = :2 AND (‘ || in_filtr || ‘)’
        INTO l_col1, l_col2
        USING IN l_ratio, l_categ_name;

      EXECUTE IMMEDIATE ‘UPDATE ‘ || in_tbl_name ||
                        ‘   SET categ_name = :1’ ||
                        ‘ WHERE categ_name = :2’
        USING IN in_new_categ_name, in_old_categ_name;
```

- Use bind variables to minimalize parse CPU costs
- Use bind variables to minimalize SQL injection security traits
- Number of bind variables can’t be changed dynamically
- Bind variables are assigned by position not by name

EXECUTE IMMEDIATE

Usage

- Dynamic PL/SQL
  - Execute anonymous block generated at runtime by application
  - Most flexible
  - Possible to change PL/SQL expressions and subprograms names at runtime
  - Most problematic
  - Can use bind variables to send values from and to dynamic block

```
l_tax_exprn := ‘ :1 * :2 / 100 ’;
l_pls  := ‘DECLARE
              l_tax NUMBER;
            BEGIN
              l_tax := ’ || l_tax_exprn || ‘;
              :3  := l_tax
            END;’;
EXECUTE IMMEDIATE l_pls
  USING IN OUT l_tax_pct, l_amt, l_tax;
```

EXECUTE IMMEDIATE

Usage

- Bulk bindings with dynamic code
  - Can be used only with collection type declared in schema e.g.

```
SQL> CREATE OR REPLACE TYPE t_num_list IS TABLE OF NUMNER;
     /
DECLARE
  l_num_list t_num_list;
```

  - Dynamic query

```
EXECUTE IMMEDIATE ‘SELECT prod_skid
                     FROM ‘ || in_tbl_name || ‘
                    WHERE ‘ || in_filtr
  BULK COLLECT INTO l_num_list;
```

  - Dynamic PL/SQL

```
EXECUTE IMMEDIATE ‘DECLARE
                       l_num_list2 t_num_list;
                     BEGIN
                       l_num_list2 := :1;
                       ...
                       :2 := l_num_list2;
                     END;’;
   USING IN OUT l_tax_pct, l_amt, l_tax;
```

DBMS_SQL

- Do not use DBMS_SQL to execute dynamic code
  - Has poor performance
  - Is too complicated and creates large code
  - Has less features and supporting less data types
- Has only one advantage over EXECUTE IMMEDIATE
  - Binding variables by name not position
  - So number od columns and bind variables can change dynamically

```
v_cur_hdl  := DBMS_SQL.OPEN_CURSOR;
v_stmt_str := 'SELECT ename, sal FROM emp WHERE job = :g_jobname';
DBMS_SQL.PARSE(v_cur_hdl,v_stmt_str,DBMS_SQL.NATIVE);
DBMS_SQL.BIND_VARIABLE(v_cur_hdl, 'g_jobname', 'SALESMAN');
DBMS_SQL.DEFINE_COLUMN(v_cur_hdl, 1, v_name, 200);
DBMS_SQL.DEFINE_COLUMN(v_cur_hdl, 2, v_salary);
v_rows_processed := DBMS_SQL.EXECUTE(v_cur_hdl);
LOOP
  IF DBMS_SQL.FETCH_ROWS(v_cur_hdl) > 0 THEN
    DBMS_SQL.COLUMN_VALUE(v_cur_hdl, 1, v_name);
    DBMS_SQL.COLUMN_VALUE(v_cur_hdl, 2, v_salary);
  ELSE
    EXIT;
END IF;
END LOOP;
DBMS_SQL.CLOSE_CURSOR(v_cur_hdl);
```

Topic Agenda

PL/SQL in Data Warehouse

- Error Logging
- RETURNING clause
- Using Bulk Binding
  - BULK COLLECT INTO
  - FORALL
  - Using SAVE EXCEPTIONS
- Pipelined Functions
- Parallel Execution

Error Logging

- By default when DML statement fails
  - Whole statement is rolled back
  - Even only 1 from 10 000 000 records makes problem with e.g. NOT NULL
  - It is time and server resources consuming to repeat whale operation
- Use error log table to continue execution after error

```
dbms_errorlog.create_error_log(dml_table_name => ‘desttbl');
```

```
desc err$_desttbl
  ORA_ERR_NUMBER$
  NUMBER ORA_ERR_MESG$
  ORA_ERR_ROWID$
  ORA_ERR_OPTYP$ - I,U,D
  ORA_ERR_TAG$
  data columns ...
```

- Use LOG ERRORS INTO clause during DML statement

```
INSERT INTO desttbl
   SELECT * FROM srcetbl
   LOG ERRORS INTO err$_desttbl ('INSERT') REJECT LIMIT UNLIMITED;
```

RETURNING clause

- Can be added to DML statement
- Works like SELECT clause included in DML
- DML can return processed rows using INTO clause

```
BEGIN
  UPDATE prod_dim
     SET categ_name = l_categ_name
   WHERE prod_skid = l_prod_skid;
   RETRUNING sub_brand_name, brand_name
        INTO l_sub_brand_name, l_brand_name;
```

- Can return many rows if BULK COLLECT INTO used

```
BEGIN
  UPDATE prod_dim
     SET categ_name = l_new_categ_name
   WHERE categ_name = l_old_categ_name
   RETRUNING sub_brand_name, brand_name
        BULK COLLECT INTO l two name list n;
```

- Can return rows in EXECUTE IMMEDIATE but without BULK COLLECT

```
EXECUTE IMMEDIATE 'DELETE table_prc WHERE id = :1
                     RETURNING salary_amt INTO :2'
  USING l_id RETURNING l_salary_amt;
```

Using Bulk Binding

BULK COLLECT INTO

- Used to send collection from SQL to PL/SQL engine
- Collection of records type bind variable needed

```
TYPE t_rec_list IS TABLE OF table_prc%ROWTYPE INDEX BY PLS_INTEGER;
l_rec_list t_rec_list;
```

- Collection of scalar type bind variables also used

```
TYPE t_num_list IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
TYPE t_name_list IS TABLE OF VARCHAR2(100) INDEX BY PLS_INTEGER;
l_num_list t_num_list;
l_name_list t_name_list;
```

- Can be used in SELECT, RETURNING, FETCH
  - Implicit cursor query

```
BEGIN
  ...
  SELECT * FROM table_prc
    BULK COLLECT INTO l_rec_list
    WHERE ...;
```

Using Bulk Binding

BULK COLLECT INTO

- DML with RETURNING clause (described before)
- FETCH from static cursor and cursor variable

```
BEGIN
 ...
    FETCH c_table_prc_cur
      BULK COLLECT INTO l_rec_list LIMIT l_rec_limit;
```

- EXECUTE IMMEDIATE with dynamic multirow query

```
BEGIN
 ...
    EXECUTE IMMEDIATE ‘SELECT * FROM table_prc’
      BULK COLLECT INTO l_rec_list;
```

- EXECUTE IMMEDIATE with DML BULK RETURNING clause not working

```
BEGIN
 ...
    EXECUTE IMMEDIATE ‘DELETE table_prc ...
                        ~~RETURNING * BULK COLLECT INTO :1’~~
      ~~RETURNING BULK COLLECT l_rec_list;~~
```

Using Bulk Binding

FORALL

- Similar to FOR loop but executed on SQL engine ones
- Only one DML statement can be nested inside loop
- Implicit cursor attributes
  - SQL%FOUND: refers to the last execution of the nested statement
  - SQL%NOTFOUND: refers to the last execution of the nested statement
  - SQL%ROWCOUNT: total number of rows affected by the whole bulk operation
- Syntax

```
FORALL index IN
[
  lower_bound..upper_bound |
  INDICES OF indexing_collection |
  VALUES OF indexing_collection
]
[SAVE EXCEPTIONS]
DML statement
[RETURNING ... BULK COLLECT INTO ...];
```

Using Bulk Binding

FORALL

- On dense collection

```
DECLARE
  TYPE t_task_list IS TABLE OF tasks_prc%ROWTYPE;
  l_task_list t_task_list := t_task_list();
BEGIN
  FOR l_step IN 1 .. 1000 LOOP
    l_task_list.extend;
    l_task_listl_task_list.LAST).id    := l_step;
    l_task_list(l_task_list.LAST).desc := ‘Task: ' || TO_CHAR(l_step);
  END LOOP;

 FORALL l_step IN 1..l_task_list.COUNT
  UPDATE task_prc
     SET desc = l_task_list(l_step)
   WHERE id = l_step;
```

- On sparse collection

```
-- Make collection sparse.
  l_task_list.DELETE(301);
  l_task_list.DELETE(601);
  l_task_list.DELETE(901);

 FORALL l_step IN INDICES OF l_task_list
   INSERT INTO task_prc VALUES l_task_list(i)
END;
```

Using Bulk Binding

FORALL

- Values of one collection as index pointers to another

```
  TYPE t_index_list IS TABLE OF BINARY_INTEGER;
  l_index_list t_index_list := t_index_list();
BEGIN
  FOR l_step IN 1 .. 1000 LOOP
    IF MOD(l_step, 100) = 0 THEN
      l_index_list.extend;
      l_index_list(l_index_list.LAST) := l_step;
    END IF;
  END LOOP;

FORALL l_step IN VALUES OF l_index_list
  INSERT INTO task_prc VALUES l_task_list(l_step);
```

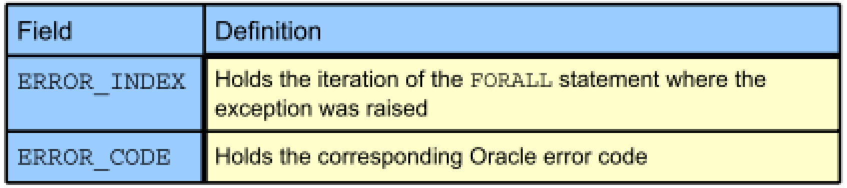
- Using SQL%BULK_ROWCOUNT instead of SQL%ROWCOUNT

```
FORALL ...;
FOR l_step IN l_task_list.FIRST..l_task_list.LAST LOOP
  dbms_output.put_line
   ('Element: ' || l_task_list(i) ||
    '   Rows: ' || SQL%BULK_ROWCOUNT(l_step));
END LOOP;
```

Using Bulk Binding
FORALL Exceptions

- By default when DML statement fails
  - Whole statement is rolled back like in outside FORALL DML without and error logging

- Add SAVE EXCEPTIONS keyword into FORALL statement
  - Exceptions are saved in the %BULK_EXCEPTIONS cursor attribute
  - Attribute is a collection of records with two fields:



Field	Definition
ERROR_INDEX	Holds the iteration of the FORALL statement where the exception was raised
ERROR_CODE	Holds the corresponding Oracle error code

  - Values always refer to the most recently executed FORALL statement
  - To convert error_code to message use SQLERRM function with - sign

```
SQLERRM(-SQL%BULK_EXCEPTIONS(index).error_code)
```

Using Bulk Binding

FORALL Exceptions

```
DECLARE
  TYPE t_num_list IS TABLE OF NUMBER;
  l_num_list t_num_list := t_num_list(100,0,110,300,0,199,200,0,400);
  e_bulk_err EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_bulk_err, -24381 ); ← ORA-24381: error(s) in array DML
BEGIN
  FORALL l_step IN l_num_list.FIRST.. l_num_list.LAST
    SAVE EXCEPTIONS
    DELETE FROM orders_fct
      WHERE order_tot_amt < 500000/l_num_list(l_step);
EXCEPTION WHEN bulk_errors THEN
  dbms_output.put_line(‘Errors count: ‘ ||SQL%BULK_EXCEPTIONS.COUNT);
  FOR l_step in 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
    dbms_output.put_line(
      TO_CHAR(SQL%BULK_EXCEPTIONS(l_step).ERROR_INDEX) || ' / ' ||
      SQLERRM(-SQL%BULK_EXCEPTIONS(l_step).ERROR_CODE));
  END LOOP;
END;
```

Table Function

- Creation

```
CREATE OR REPLACE TYPE t_sales_obj IS OBJECT (...);
CREATE OR REPLACE TYPE t_sales_tbl_n IS TABLE OF t_sales_obj;
CREATE FUNCTION fn_sales_tbl(in_rows INTEGER) RETURN t_sales_tbl_n
IS
 l_sales_tbl t_sales_tbl_n := t_sales_tbl_n();
BEGIN
  FOR l_step IN 1..in_rows LOOP
    l_sales_tbl.EXTEND;
    l_sales_tbl(l_sales_tbl.LAST) := t_sales_tbl_n(...);
  END LOOP;
  RETURN l_sales_tbl;
END;.
```

- Is used in query FROM clause

```
SELECT ... FROM TABLE(fn_sales_tbl(...));
```

- Useful if source of query can’t be table or view
- Keeps all collection in memory
- **Large memory used when processing large volume**

Pipelined Table Function

Overview

- No need to build all collection before return outside
- Build and pipe collection elements outside one by one
- Keeps only small number of elements in memory e.g. one
- **Good to process large volume**
- Example

```
CREATE FUNCTION fn_sales_tbl(in_rows IN PLS_INTEGER)
  RETURN t_sales_tbl_n PIPELINED
IS
  l_sales_obj t_sales_obj;
BEGIN
  FOR l_step IN 1..in_rows LOOP
    ...
    l_sales_obj := t_sales_obj(...);
    PIPE ROW (l_sales_obj);
  END LOOP;
  RETURN;
END;
```

Pipelined Table Function

Ref Cursor

- Source of function data can be static or ref cursor
- Ref cursor can be used as pipelined function parameter
  - So we have collection of rows on input and on output

```
CREATE PACKAGE pkg_ref_cur IS
  TYPE t_ref_cur IS REF CURSOR RETURN sales_fct%ROWTYPE;
END;

CREATE FUNCTION fn_sales_tbl(in_ref_cur IN pkg_ref_cur.t_ref_cur)
  RETURN t_sales_tbl_n PIPELINED
  l_sales_srce sales_fct%ROWTYPE;
BEGIN
  LOOP
    FETCH in_ref_cur INTO l_sales_srce_rec;
    ...
    PIPE ROW (l_sales_obj);
  EXIT WHEN in_ref_cur%NOTFOUND;
  END LOOP;
  RETURN;
END;
```

Pipelined Table Function

NO_DATA_NEEDED Exception

- Raised when not all rows returned by PIPE ROW are consumed by external query which uses pipelined function

```
SELECT ... FROM TABLE(fn_sales_tbl(...)) WHERE ROWNUM < 5;
```

- Terminate function to cleanup resources and avoid next PIPE
- Do not terminate external query
- Can be handled to customize clean operation

```
EXCEPTION
  WHEN NO_DATA_NEEDED THEN
       pkg_sales.pro_clean_up;
       RAISE NO_DATA_NEEDED;
```

- **Do not handle this exception in OTHERS**
  - **It treating NO_DATA_NEEDED as an unexpected error**

Pipelined Table Function

Returned Rows Cardinality

- Oracle SQL Optimizer do not know returned rows cardinality
- Always assumes 8168 rows (when default block size is used)
- Cardinality can be customized by using
  - CARDINALITY hint (9i+): Undocumented
  - OPT_ESTIMATE hint (10g+): Undocumented
  - Extensible Optimizer (9i+): Below
- Extensible optimizer requires **cardinality parameter** in function

```
FUNCTION get_tab_ptf (in_cardinality IN INTEGER DEFAULT 1, ...)
  RETURN t_sales_tbl_n PIPELINED ...
```

  - Parameter isn't used anywhere in the function itself
  - **Create** function statistics handling **object type** (next slide)
  - Let Oracle use this type to provide function returned rows cardinality

```
SQL> ASSOCIATE STATISTICS WITH FUNCTIONS fn_sales_tbl USING t_pipelined_stats;
```

  - Provide cardinality during function invocation

```
SELECT ...
FROM TABLE(fn_sales_tbl(in_cardinality => 10, ...));
```

Pipelined Table Function

Extensible Optimizer Object Type Example

```
CREATE OR REPLACE TYPE BODY t_pipelined_stats AS

  STATIC FUNCTION ODCIGetInterfaces (out_interfaces OUT SYS.ODCIObjectList)
    RETURN NUMBER
  IS
  BEGIN
    out_interfaces := SYS.ODCIObjectList(SYS.ODCIObject('SYS', 'ODCISTATS2'));
    RETURN ODCIConst.success;
  END ODCIGetInterfaces;

  STATIC FUNCTION ODCIStatsTableFunction(
                    in_function    IN  SYS.ODCIFuncInfo,
                    out_stats      OUT SYS.ODCITabFuncStats,
                    in_args        IN  SYS.ODCIArgDescList,
                    in_cardinality IN INTEGER)
    RETURN NUMBER
  IS
  BEGIN
    out_stats := SYS.ODCITabFuncStats();
    out_stats.num_rows := in_cardinality;
    RETURN ODCIConst.success;
  END ODCIStatsTableFunction;

END;
/
```

Pipelined Table Function

Implicit Shadow Type

- Pipelined (unlike regular table functions) can base on record and table types defined in a package specification.

```
CREATE OR REPLACE PACKAGE pkg_sales IS
  TYPE t_sales_rec IS RECORD ( ... );
  TYPE t_sales_tbl IS TABLE OF t_sales_rec;
  FUNCTION fn_sales_tbl(...) RETURN t_sales_tbl PIPELINED;
END;
```

- In this situation Oracle implicitly creates object types
  - To support the types required by the pipelined table function
  - This shadow types uses system generated names

Pipelined Table Function

ETL Pipelines

- Can be used in ETL transformation pipelines
- One function is source for another
- Step 1 function can get data from external table

```
PROCEDURE pro_load_data IS
BEGIN
  EXECUTE IMMEDIATE 'ALTER SESSION ENABLE PARALLEL DML';
  EXECUTE IMMEDIATE 'TRUNCATE TABLE dest_fct';

  INSERT /*+ APPEND PARALLEL(t4, 5) */
    INTO dest_fct t4
    SELECT /*+ PARALLEL(t3, 5) */ *
      FROM TABLE(fn_step_2(CURSOR(
         SELECT /*+ PARALLEL(t2, 5) */ *
           FROM TABLE(fn_step_1(CURSOR(
             SELECT /*+ PARALLEL(t1, 5) */ *
               FROM source_ext t1))) t2))) t3;
   COMMIT;
 END load_data;
```

Parallel Execution

PARALLEL_ENABLE

- By default functions are executed serially
- Add PARALLEL_ENABLE to your function
  - This tells Oracle that function do not share any session data
  - So many parallel slaves can run function independently in parallel query

```
CREATE OR REPLACE FUNCTION fn_factorial(in_val INTEGER)
  RETURN INTEGER PARALLEL_ENABLE
IS
  l_minus1 PLS_INTEGER;
BEGIN
  l_minus1 := in_val - 1;
  IF (l_minus1 > 0)
  THEN
    RETURN in_val * fn_factorial(l_minus1);
  END IF;
  RETURN in_val;
END;
/
```

Parallel Execution

Pipelined Table Function

- PARTITION BY clause needed in pipelined function
  - If parallelized by PARALLEL_ENABLE keyword
  - Used to define workload partitioning to parallel slaves

```
PARALLEL_ENABLE(PARTITION parameter-name BY [{HASH | RANGE} (column-list) | ANY ])
[ORDER | CLUSTER] parameter-name BY (column-list)
```

  - Weakly typed ref cursors can only use the PARTITION BY ANY clause
    - ✓ Random partitioning - sometime wrong distribution and results order

```
CREATE FUNCTION fn_sales_tbl(in_week_cur IN SYS_REFCURSOR)
  RETURN t_sales_tbl_n
  PARALLEL_ENABLE(PARTITION in_week_cur BY ANY) PIPELINED
IS ...
```

  - Range or Hash partitioning
    - ✓ best advantage - same key value -> same function instance

```
PARTITION in_ref_cur BY RANGE (cust_name, categ_name)
```

Parallel Execution

Pipelined Table Function

- RANGE vs HASH
  - HASH uses hash function do map key value to function instance
  - Range maps function instances to range of key values
  - HASH is quicker than RANGE and usually used with CLUSTER keyword
- Order streaming option
  - To get input records from parameter ref cursor in selected order
  - To avoid e.g. incomplete summary per key calculated inside function

```
... PARTITION in_ref_cur BY RANGE (cust_name, categ_name)
    ORDER in_ref_cur BY (cust_name)
```

- CLUSTER option
  - **Only guarantee to deliver same key value together**
  - **Not guarantee order of delivering key value records groups**

Parallel Execution

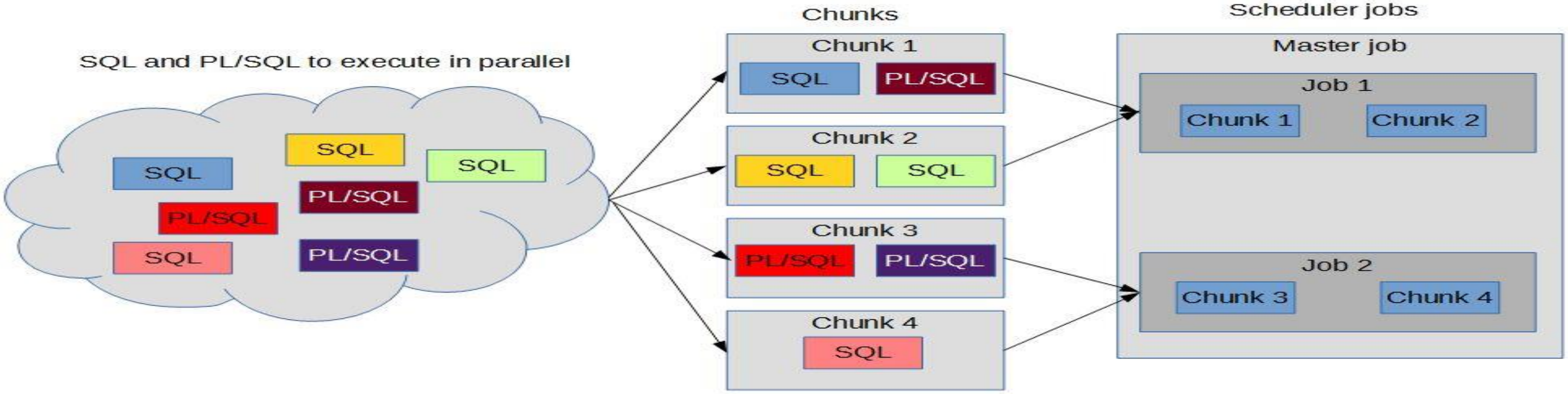
DBMS_PARALLEL_EXECUTE Purpose

- Enables data incremental parallel update in a large table
  - Group sets of rows in the table into smaller chunks.
  - Apply the desired UPDATE statement to the chunks in parallel, committing each time you have finished processing a chunk.
- Recommended when updating a lot of data
- Advantages
  - Lock only one set of rows at a time, for a relatively short time
  - You do not lose work if something fails before entire operation finishes
  - You reduce rollback space consumption
  - You improve performance

Parallel Execution

DBMS_PARALLEL_EXECUTE Overview

- Use only if Parallel DML is not enough
- Can execute SQL and PL/SQL code in parallel
- Uses background jobs similar to DBMS_SCHEDULER **but**
  - Task (Master job) can be divided into many serial jobs (Job 1, Job2)



  - Code or data is divided into chunks
  - Chunks are distributed to jobs to load balance workload
  - One job is executing chunks serially and sequentially

Parallel Execution

DBMS_PARALLEL_EXECUTE Example

```
CREATE OR REPLACE PROCEDURE pro_updt_salary( in_start_rowid IN ROWID,
                                             in_end_rowid   IN ROWID) IS
BEGIN
  UPDATE sales_fct s
     SET s.salary = salary * 0.10
   WHERE s.rowid BETWEEN in_start_rowid AND in_end_rowid;
END;

dbms_parallel_execute.create_task(task_name => 'MyTask);


dbms_parallel_execute.create_chunks_by_rowid
 (task_name  => 'MyTask', table_owner => USER,
  table_name => ‘SALES_FCT', chunk_size => 1000 );


dbms_parallel_execute.run_task
  (task_name => 'MyTask',
   sql_stmt => 'BEGIN pro_updt_salary(:start_id, :end_id ); END;',
   parallel_level => 4 );
```

create_chunks_by_rowid
create_chunks_by_number_col
create_chunks_by_sql

Topic Agenda

Miscellany PL/SQL Features

- Autonomous Transactions
- Calling a Java Class
- Mutating in Trigger

Autonomous Transaction

Overview

- Leaving the calling transaction unfinished and unchanged
- Perform an independent transaction
- Return to the calling transaction without affecting it's state
- Cover all unit where defined
- Defined by adding on the beginning of declaration section

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

- Available in
  - Stored procedure or function
  - Local procedure or function defined in a PL/SQL declaration block

```
PROCEDURE ...
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
```

  - Packaged procedure or function
  - Object type method
  - Top-level anonymous block

```
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
  ...
```

  - Trigger by using anonymous block

Autonomous Transaction

Rules

- All SQL in pragma block and its subprograms belongs to it
- Pragma block must be finished by COMMIT or ROLLBACK

```
      ...
      COMMIT;
   END pro_appl_log;
```

- Exceptions raised in an autonomous transaction cause
  - a transaction-level rollback
  - not a statement-level rollback
- If called subprogram include pragma then have separate transaction
- Without pragma used
  - Only one transaction exist on database session in particular point of time
  - Commit or rollback in any PL/SQL code point finished old and start new transaction
  - All table modifications done in procedure is done on current session transaction
- Works almost like opening separate sub-session
  - Deadlock when attempts to access a resource held by the main transaction
  - Not see main transaction changes
  - Its modifications visible after commit

Calling a Java Class

Introduction

- Oracle database can store Java classes and Java source

```
CREATE JAVA CLASS USING BFILE (java_dir, 'Agent.class')
/
CREATE JAVA SOURCE NAMED "Welcome"
  AS public class Welcome {
       public static String welcome() {
         return "Welcome World"; } }
/
```

```
$ loadjava  -u  user/password@database [options]
           file.java | file.class | file.jar | resourcefile | URL...
```

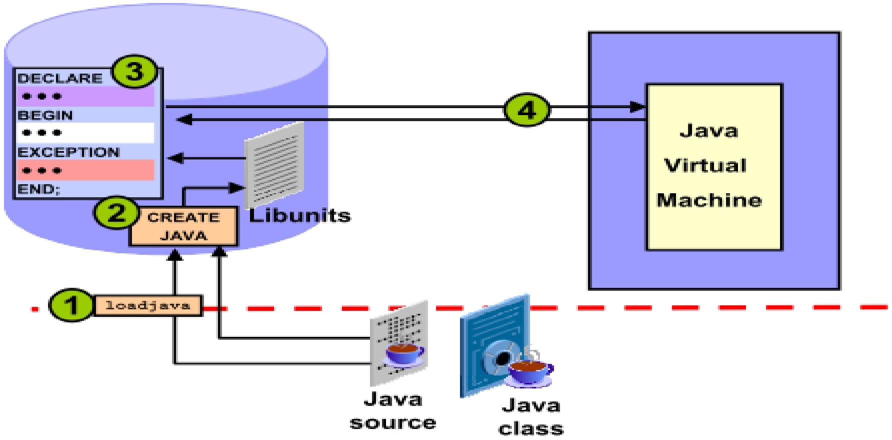
- Oracle provides many Java packages preinstalled
  - Can be used from our loaded java code e.g.

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.*;
```

  - **DBMS_JAVA Package -** Entry point for accessing RDBMS functionality from Java
  - [Documentation: Database Java Developer's Guide]()

Calling a Java Class

Calling Java Method From PL/SQL



- Publish method as procedure, function or trigger with external body

```
CREATE OR REPLACE FUNCTION fn_java(in_num  BINARY_INTEGER,
                                   in_name VARCHAR2,
                                   in_date DATE) RETURN NUMBER
IS
  LANGUAGE JAVA
  NAME 'package.method(int, java.lang.String, java.sql.Date) return int';
```

- Execute as PL/SQL unit

Mutating

Error

- Oracle mutating error occurs during table DML when
  - PL/SQL function references table is used in SQL on the same table
  - trigger references the table that owns the trigger

    `"ORA-04091: table name is mutating, trigger/function may not see it.‟`
- SQL statement can’t see consistent snapshot of data
- **To avoid** this error in triggers
  - Don't use triggers

    use package subprograms or object-oriented methods
  - Use an "after" or "instead of" timings for statement level triggers
  - Use autonomous transactions

    Independent transaction not mutate tables
  - Use **combination of row-level and statement-level** triggers
    - ✓ Store rows processed info in collection from row level trigger
    - ✓ Use this info in statement level trigger
    - ✓ Global temporary table can be used instead of collection

Mutating

Example

```
CREATE TABLE task_prc(id NUMBER(10), desc VARCHAR2(50));

 CREATE TABLE task_audit(id NUMBER(10), actn VARCHAR2(10), tbl_id NUMBER(10),
                        rec_cnt NUMBER(10), crtn_time TIMESTAMP);
```

```
CREATE OR REPLACE PACKAGE BODY pkg_task AS
  PROCEDURE pro_change(in_id IN task_prc.id%TYPE, in_actn IN VARCHAR2) IS
    l_count NUMBER(10) := 0;
  BEGIN
    SELECT COUNT(*) INTO l_count FROM task_prc;
    INSERT INTO task_audit(ID, actn, tbl_id, rec_cnt, crtn_time)
      VALUES (task_audit_id_seq.NEXTVAL, in_actn, in_id, l_count, SYSTIMESTAMP);
  END;
END;

CREATE OR REPLACE TRIGGER task_prc_traiu
AFTER INSERT OR UPDATE ON task_prc
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    pkg_task.pro_change(in_id => :new.id, in_actn => 'INSERT');
  ELSE
    pkg_task.pro_change(in_id => :new.id, in_actn => 'UPDATE');
  END IF;
END;

INSERT INTO task_prc(id, desc) VALUES (task_audit_id_seq.NEXTVAL, 'ONE')
ORA-04091: table TASK_PRC is mutating, trigger/function may     see it
```

Mutating - Solution

```
CREATE OR REPLACE PACKAGE BODY pkg_task AS
  TYPE t_change_rec IS RECORD (
    ID task_prc.ID%TYPE,
    actn task_audit.actn%TYPE
  );
  TYPE t_change_tab IS TABLE OF t_change_rec;
  g_change_tab  t_change_tab := t_change_tab();

  PROCEDURE pro_change(in_id IN task_prc.id%TYPE, in_actn IN VARCHAR2) IS
  BEGIN
    g_change_tab.EXTEND;
    g_change_tab(g_change_tab.LAST).id   := in_id;
    g_change_tab(g_change_tab.LAST).actn := in_actn;
  END;

  PROCEDURE pro_statement_change IS
    l_count  NUMBER(10);
  BEGIN
    FOR i IN g_change_tab.first .. g_change_tab.last LOOP
      SELECT COUNT(*)
      INTO   l_count
      FROM   task_prc;
      INSERT INTO task_audit (ID, actn, task_id, rec_cnt, crtn_time)
      VALUES(task_audit_id_seq.NEXTVAL, g_change_tab(i).actn, g_change_tab(i).id, l_count, SYSTIMESTAMP);
    END LOOP;
    g_change_tab.delete;
  END;
END;

CREATE OR REPLACE TRIGGER task_prc_traiu
AFTER INSERT OR UPDATE ON task_prc
BEGIN
  pkg_task.pro_stmt_change;
END;
```

All kind of table triggers can be created as one Compound trigger e.g Statement and Row level triggers

Topic Agenda

Profiling and Tracing

- Tracing
- Profiling

Tracing

Tracing Session

- Used to better understand the program execution path
- Trace session steps
  - Enable specific subprograms for tracing (optional)

```
ALTER SESSION SET PLSQL_DEBUG = TRUE;
CREATE OR REPLACE ....
ALTER [PROCEDURE | FUNCTION | PACKAGE] name COMPILE DEBUG [BODY];
```

  - Start tracing session

```
dbms_trace.set_plsql_trace(trace_level => constant1 + constant2 ...);
```

  - Run application to be traced
  - Stop tracing session

```
dbms_trace.clear_plsql_trace;
```

www.lingaro.com

Tracing

Options

- trace_level INTEGER parameter
  - Provided as sum of _all_ or _enabled_ version constants
  - Trace-level constants:

```
trace_all_calls
trace_all_sql
trace_all_exceptions
trace_all_lines

trace_enabled_calls
trace_enabled_sql
trace_enabled_exceptions
trace_enabled_lines

trace_stop
trace_pause
trace_resume
```

Tracing

Display Trace Information

```
event_seq        -- unique sequence number within run
event_time       -- timestamp
event_unit       -- unit where the event happened
event_line       -- line in the unit where event happened
event_proc_name     -- if not empty, procedure where event happened
proc_name        -- if not empty, name of procedure called
```

```
    SELECT proc_name, proc_line,
           event_proc_name, event_comment
      FROM plsql_trace_events
     WHERE event_proc_name = ‘PRO_LOAD_TBL'
        OR proc_name = ‘PRO_LOAD_TBL';
```

```
PROC_NAME    PROC_LINE  EVENT_PROC_NAME  EVENT_COMMENT
------------ ---------- ---------------- ---------------
PRO_LOAD_TBL          1                  Procedure Call
FN_GET_PARM           1 PRO_LOAD_TBL     Procedure Call
PRO_ADD_LOG          11 PRO_LOAD_TBL     Procedure Call
PRO_GET_DATA         15 PRO_LOAD_TBL     Procedure Call
PRO_ADD_LOG          18 PRO_LOAD_TBL     Procedure Call
PRO_GET_DATA         15 PRO_LOAD_TBL     Procedure Call
PRO_ADD_LOG          18 PRO_LOAD_TBL     Procedure Call
```

Hierarchical Profiler

Overview

- Used to identify hotspots and performance tuning
- Reports the execution profile of a PL/SQL program
  - Organized in tree of subprograms calls
  - Reports SQL and PL/SQL execution
  - Reports number of calls and time sped in execution
  - Reports subprogram and any sub-tree under subprogram
  - Provides subprogram level summaries
- Benefits:
  - Provides more information than a flat profiler (DBMS_PROFILER)
  - Can be used to understand the complex programs structure and flow

Hierarchical Profiler

Using

- Data Collection Component
  - Part of the PL/SQL engine
  - Step 1 - turn on profiler data collecting into text file

```
BEGIN
  DBMS_HPROF.START_PROFILING('PROFILE_DIR', ‘pro_load_tbl.txt');
```

  - Step 2 - run profiled PL/SQL unit

```
  pro_pload_tbl( ... );
```

  - Step 3 - turn off profiler data collecting

```
  DBMS_HPROF.START_PROFILING('PROFILE_DIR', ‘pro_load_tbl.txt');
END;
```

Hierarchical Profiler

Using

- Data Analyzer – method 1 – inside database tables
  - Create tables

    ```
    SQL> @?/rdbms/admin/dbmshptab.sql
    ```

  - Run analyze

    ```
    DECLARE
      l_run_id NUMBER;
    BEGIN
      l_run_id := dbms_hprof.analyze(LOCATION => 'PROFILE_DATA',
                                    FILENAME => 'pd_cc_pkg.txt');
      dbms_output.put_line('Run ID: ' || v_run_id);
    END;
    ```

  - Execute query reports on tables

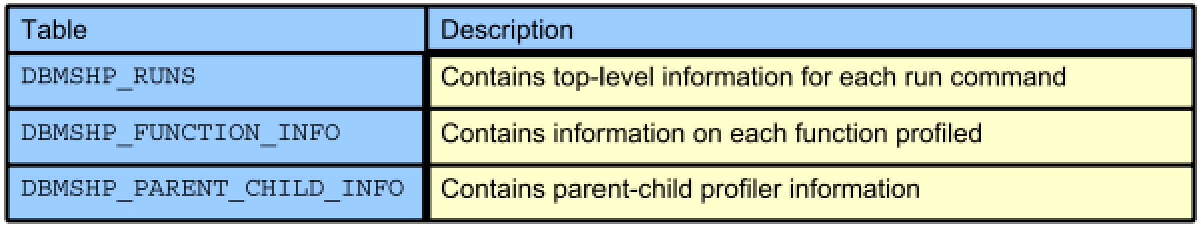


Table	Description
DBMSHP_RUNS	Contains top-level information for each run command
DBMSHP_FUNCTION_INFO	Contains information on each function profiled
DBMSHP_PARENT_CHILD_INFO	Contains parent-child profiler information

Hierarchical Profiler

Using

- Analyzer optional parameters

```
DBMS_HPROF.ANALYZE (
  location     IN VARCHAR2,
  filename     IN VARCHAR2,
  summary_mode IN BOOLEAN     DEFAULT FALSE,
  trace        IN VARCHAR2    DEFAULT NULL,
  skip         IN PLS_INTEGER DEFAULT 0,
  collect      IN PLS_INTEGER DEFAULT NULL,
  run_comment  IN VARCHAR2    DEFAULT NULL ) RETURN NUMBER;
```

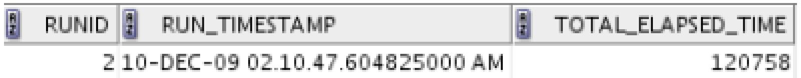
  - summary_mode: (TRUE) generate only top-level summary
  - trace: analyze only the subtrees rooted at the specified trace entry
  - skip: analyzes only the subtrees but ignores first “skip” invocations
  - collect: number of invocations to “collect” starting from “skip” + 1
  - run_comment: a comment for your run

Hierarchical Profiler

Using

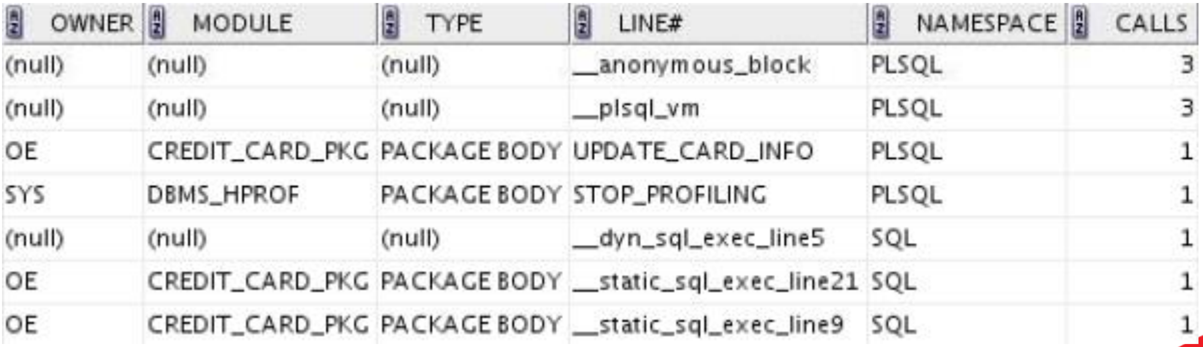
- Report Examples

```
SELECT runid, run_timestamp, total_elapsed_time
FROM dbmshp_runs WHERE runid = 2;
```



RUNID	RUN_TIMESTAMP	TOTAL_ELAPSED_TIME
2	10-DEC-09 02.10.47.604825000 AM	120758

```
SELECT owner, module, type, function line#, namespace, calls
  FROM dbmshp_function_info WHERE runid = 2;
```



OWNER	MODULE	TYPE	LINE#	NAMESPACE	CALLS
(null)	(null)	(null)	__anonymous_block	PLSQL	3
(null)	(null)	(null)	__plsql_vm	PLSQL	3
OE	CREDIT_CARD_PKG	PACKAGE BODY	UPDATE_CARD_INFO	PLSQL	1
SYS	DBMS_HPROF	PACKAGE BODY	STOP_PROFILING	PLSQL	1
(null)	(null)	(null)	__dyn_sql_exec_line5	SQL	1
OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line21	SQL	1
OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line9	SQL	1

Hierarchical Profiler

Using

- Data Analyzer – method 2 – HTML Report Generator
  - generate simple HTML reports directly from the raw profiler file
  - HTML reports can be browsed in any web browser
  - Navigational capabilities combined with the links
  - Analyze multilevel PL/SQL performance profile from one HTML report
  - plshprof - command-line utility
    - ✓ On OS shell change current directory to row profiler file directory
    - ✓ HTML output will be generated to current directory

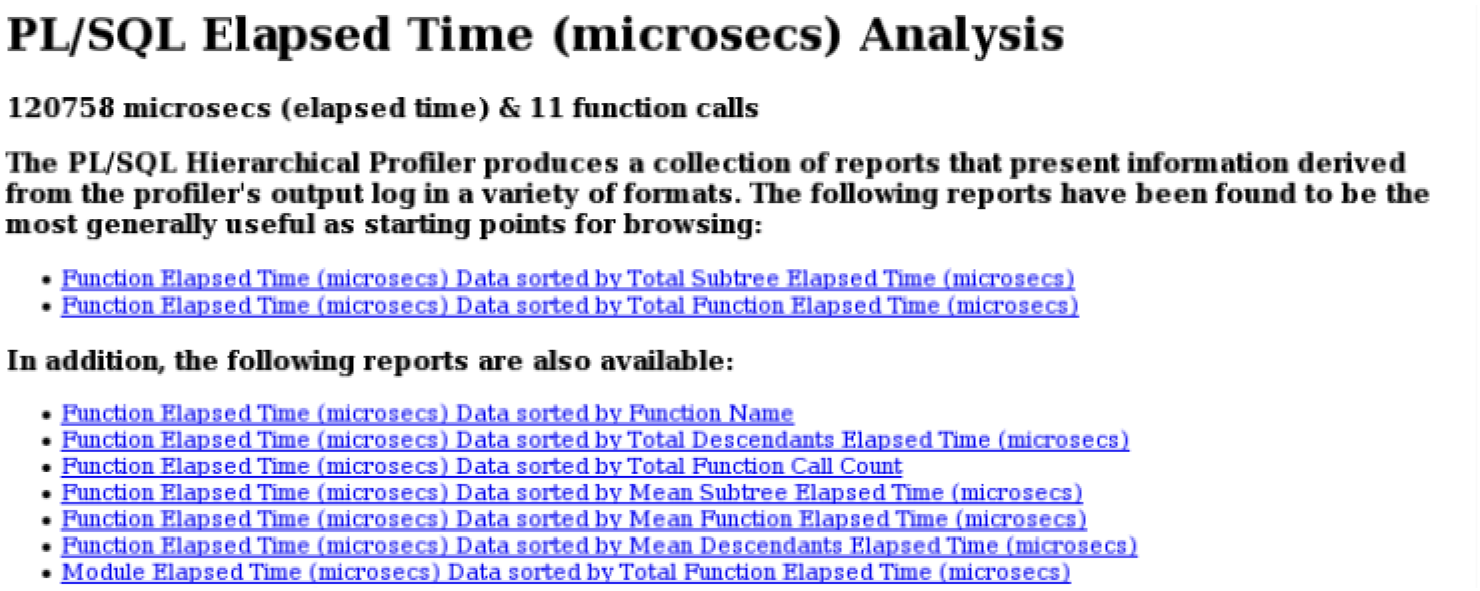
```
plshprof [option…] input_raw_file output_file_name
```

    - ✓ The main *`output_file_name`*`.html` file and other html files will be generated.
  - Open main html file in web brawser

Hierarchical Profiler

Using

- Main HTML file



PL/SQL Elapsed Time (microsecs) Analysis

120758 microsecs (elapsed time) & 11 function calls

The PL/SQL Hierarchical Profiler produces a collection of reports that present information derived from the profiler's output log in a variety of formats. The following reports have been found to be the most generally useful as starting points for browsing:

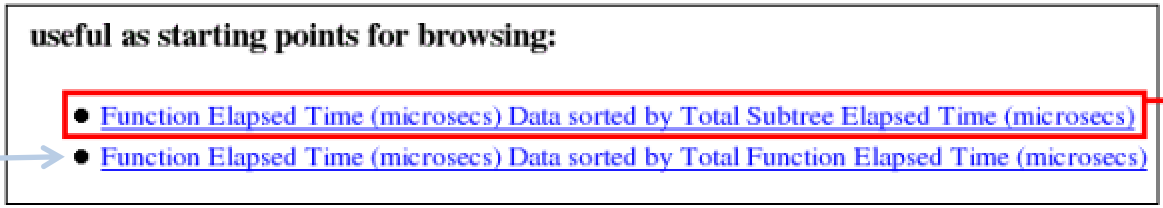
- Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)

In addition, the following reports are also available:

- Function Elapsed Time (microsecs) Data sorted by Function Name
- Function Elapsed Time (microsecs) Data sorted by Total Descendants Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Function Elapsed Time (microsecs) Data sorted by Mean Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Mean Function Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Mean Descendants Elapsed Time (microsecs)
- Module Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)

Hierarchical Profiler

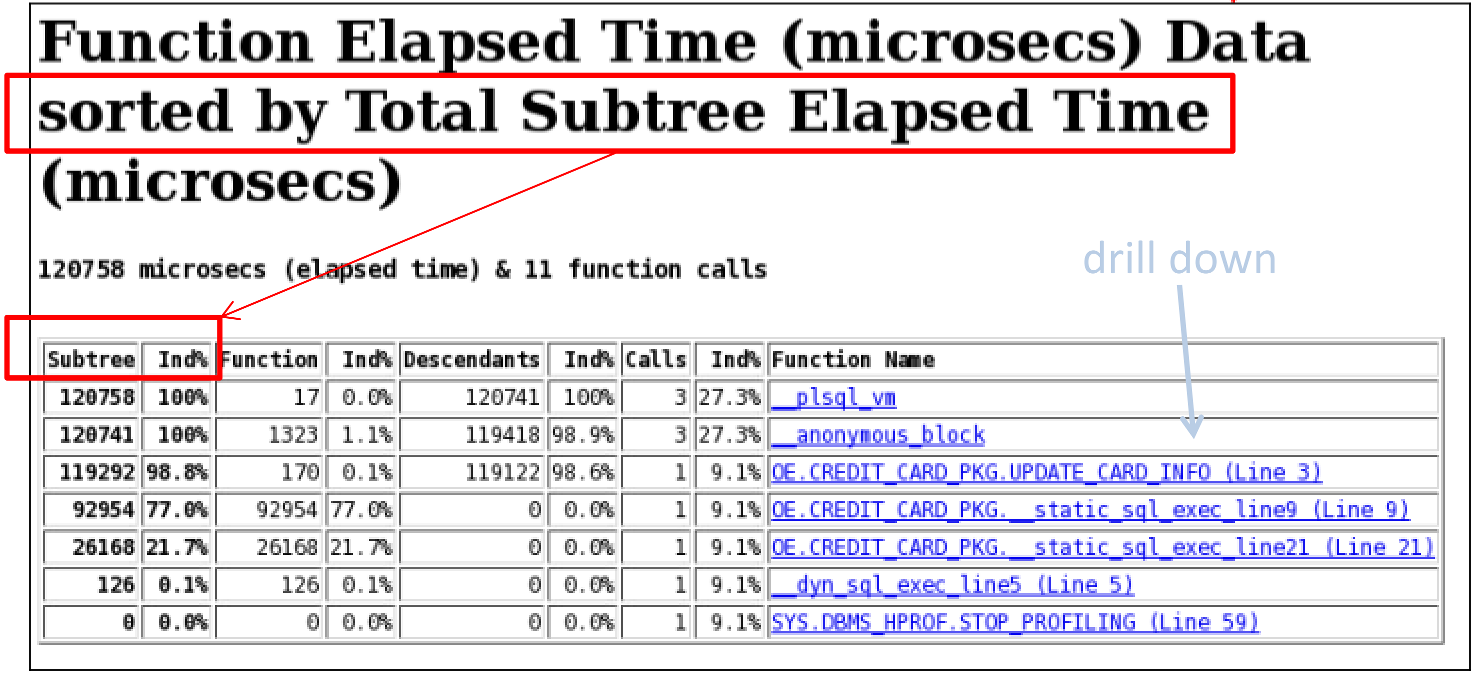
Using



useful as starting points for browsing:

- Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)

by module



Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)

```
120758 microsecs (elapsed time) & 11 function calls
```

drill down

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
120758	100%	17	0.0%	120741	100%	3	27.3%	__plsql_vm
120741	100%	1323	1.1%	119418	98.9%	3	27.3%	__anonymous_block
119292	98.8%	170	0.1%	119122	98.6%	1	9.1%	OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 3)
92954	77.0%	92954	77.0%	0	0.0%	1	9.1%	OE.CREDIT_CARD_PKG.__static_sql_exec_line9 (Line 9)
26168	21.7%	26168	21.7%	0	0.0%	1	9.1%	OE.CREDIT_CARD_PKG.__static_sql_exec_line21 (Line 21)
126	0.1%	126	0.1%	0	0.0%	1	9.1%	__dyn_sql_exec_line5 (Line 5)
0	0.0%	0	0.0%	0	0.0%	1	9.1%	SYS.DBMS_HPROF.STOP_PROFILING (Line 59)

Training Agenda

- Not included
  - Finding PL/SQL Code Information
  - Debugging PL/SQL Code
  - DBMS_METADATA Package
  - Job Scheduler - DBMS_SCHEDULE
  - Manipulating LOB Data
  - XML Database Features

Q & A

www.lingaro.com

PL/SQL Resources

- Oracle Database Documentation Library

  http://docs.oracle.com/cd/E11882_01/index.htm

- Oracle Advanced PL/SQL Developer Professional Guide

  ISBN 978-1-84968-722-5

- O’REILLY - Oracle PL/SQL Programming

  ISBN 9780-0596-51446-4