Oracle Tutorials

PL/SQL

Procedural Language / Structured Query Language

Zbigniew Baranowski

Agenda

- Overview of PL/SQL
- Blocks
- Variables and placeholders
- Program Flow Control Statements
- Cursors
- Functions and Procedures
- Error Handling
- Packages
- Triggers
- Jobs

PL/SQL

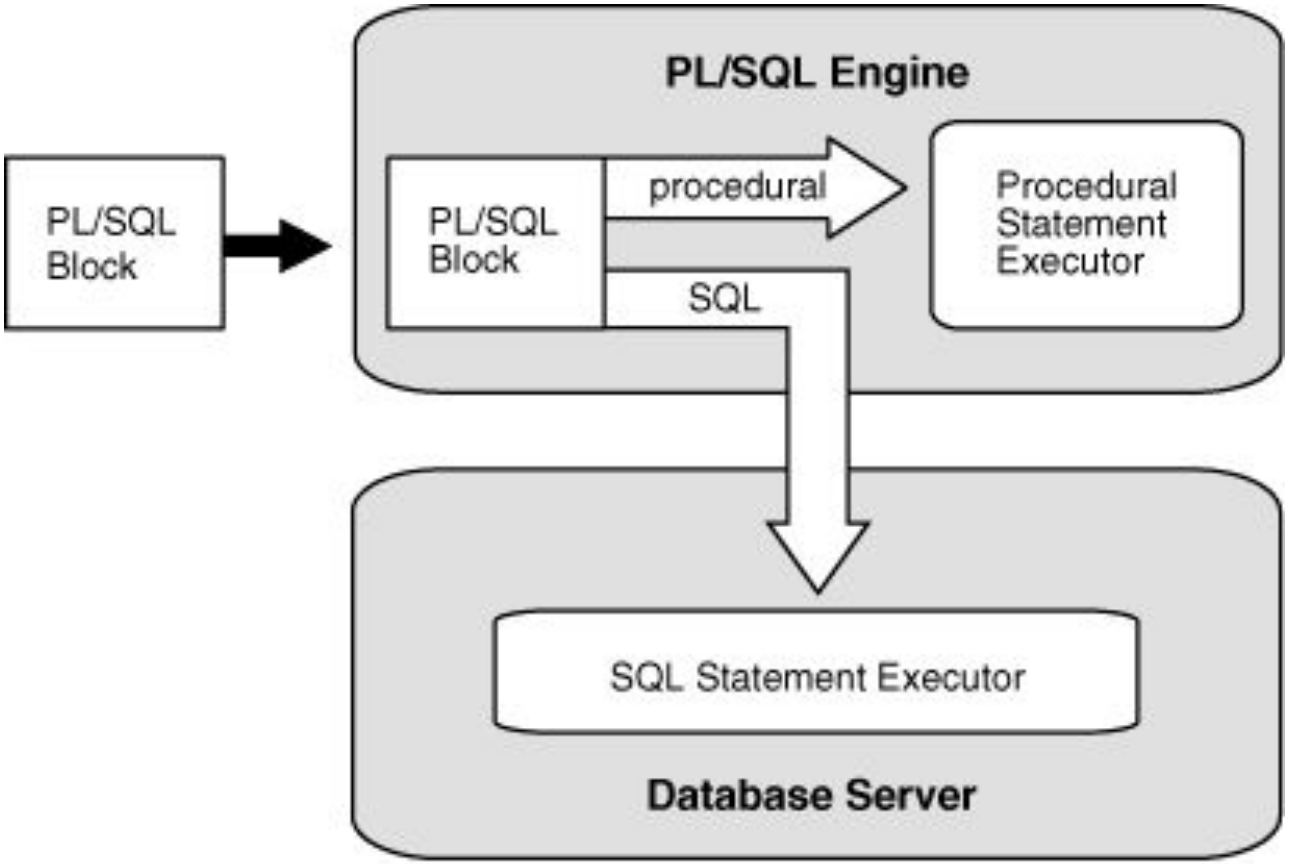
- Procedural language extension to SQL
  - procedural data manipulation
    - conditionals, loops etc.
- High-level language features
  - Complex data types
  - Data encapsulation
  - Modular programming
- Integrated with the ORACLE database server
  - Server-side
    - parsing / compilation
    - execution / interpretation
  - End-user platform independent (like SQL)

Writing PL/SQL programs

- Each program is a block consisting of
  - PL/SQL statements – logic
  - SQL statements – data manipulation

- Type of block
  - Anonymous
    - External scripts (file or input)
    - Nested blocks
  - Named / Stored (on the database)

PL/SQL execution



PL/SQL Block Structure

```
DECLARE --declaration section (types, variables, ..)
 l_commission       NUMBER;
 L_COMM_MISSING EXCEPTION;
BEGIN --executable section (program body)
 SELECT commission_pct / 100 INTO l_commission
 FROM employees WHERE employee_id = emp_id;
 IF l_commission IS NULL THEN RAISE COMM_MISSING;
 ELSE    UPDATE employees
     SET salary = salary + bonus*l_commission
     WHERE employee_id = emp_id;
 END IF;
EXCEPTION --exception section (error handling)
 WHEN L_COMM_MISSING THEN DBMS_OUTPUT.PUT_LINE('This
employee does not receive a commission.');
END;
```

PL/SQL placeholders

- All SQL types are supported by PL/SQL
  - Numerical types
    - NUMBER, PLS_INTEGER
    - Many derived types, e.g. POSITIVE
  - Character types
    - CHAR, VARCHAR2, NCHAR,…
  - Other scalar types
    - BOOLEAN, DATE, UROWID, RAW

PL/SQL placeholders

- Scalar type
  - variable
  - constant
- Composite/vector type
  - record
    - used for reading rows from table
- Collections
  - Associative Array - dictionary
  - Variable-sized Array (VARRAY) – fixed size
  - Nested Tables – dynamic size

PL/SQL placeholders

- Scalar type
  - variable
  - constant

```
DECLARE
  l_x NUMBER := 20000;
  l_message VARCHAR2(40);
  C_PI CONSTANT NUMBER(3,2):=3.14;
BEGIN
   l_x := 1000 * C_PI;
   l_message := 'Hello world';
END;
```

PL/SQL placeholders

- Scalar type
  - variable
  - constant
- Single composite/vector type
  - record
    - used for reading rows from table

```
TYPE T_TIME IS RECORD (minutes INTEGER, hours NUMBER(2));
current_time_rec  T_TIME;
Current_time_rec.hours := 12;
s_Name stud.sname%TYPE ;#data  type similar to column
sname.
```

PL/SQL placeholders

```
DECLARE
  TYPE T_POPULATION IS TABLE OF NUMBER INDEX BY VARCHAR2(64);
  l_city_population  T_POPULATION;
  l_i number;
BEGIN
  l_city_population('Smallville')  := 2000;
  l_i:= l_city_population('Smallville') ;
END;
/
```

- Collections
  - **Associative Array**

PL/SQL placeholders

```
 DECLARE
   TYPE T_FOURSOME IS VARRAY(4) OF VARCHAR2(15);
   l_team T_FOURSOME := T_FOURSOME('John', 'Mary', 'Alberto');
BEGIN
    l_team.EXTEND;                    -- Append one null element
    l_team(4):='Mike';          -- Set 5th element element
    DBMS_OUTPUT.PUT_LINE( l_team( l_team.first ) ); -- Print first element
    DBMS_OUTPUT.PUT_LINE( l_team( l_team.last ) );  -- Print last element

END;
 /
```

- Collections
  -
  - **Variable-sized Array (VARRAY)**
  -

PL/SQL placeholders

```
 DECLARE
   TYPE T_ROSTER IS TABLE OF VARCHAR2(15);
   l_names T_ROSTER := T_ROSTER('D Caruso', 'J Hamil', 'D Piro', 'R Singh');
   l_i number;
BEGIN
   FOR l_i IN l_names.FIRST .. L_names.LAST LOOP  --For first to last
   element
      DBMS_OUTPUT.PUT_LINE(l_names(l_i));
   END LOOP;
 END;
 /
```

- Collections
  -
  -
  - **Nested Tables**

Attributes %TYPE & %ROWTYPE

- %TYPE references type of a variable or a database column
- %ROWTYPE references type of a record structure, table row or a cursor
- Advantages:
  - Actual type does not need to be known
  - referenced type had changed -> will be recompiled automatically

%TYPE & %ROWTYPE Examples

variable declarations

```
balance                 NUMBER(7,2);
minimum_balance         balance%TYPE := 10.00;
my_dname                scott.dept.dname%TYPE;
dept_rec                dept%ROWTYPE;
```

```
SELECT deptno, dname, loc  INTO dept_rec
FROM dept WHERE deptno = 30;
```

using record variable to read a row from a table

PL/SQL Control Structures

- Conditional Control
  - Using IF and CASE statements

```
DECLARE
    l_sales NUMBER(8,2) := 20000;
    l_bonus NUMBER(6,2);
BEGIN
    IF l_sales > 50000 THEN l_bonus := 1500;
       ELSIF l_sales > 35000 THEN l_bonus := 500;
     ELSE l_bonus := 100;
    END IF;
    UPDATE employees SET salary = salary + l_bonus;
END;
```

```
JE('Excellent');
JE('Very Good');
JE('Good');
JE('Fair');
JE('Poor');
:h grade');
    END CASE;
END;
```

- Sequential Control
  - Using GOTO statement

PL/SQL Control Structures

- Iterative loops
  - Simple loop (infinite)
  - WHILE loop
  - FOR loop
    - Numeric range
      - Reversed
    - Cursor based

```
DECLARE
    l_i          NUMBER := 0;
BEGIN
    LOOP
      DBMS_OUTPUT.PUT_LINE (TO_CHAR(l_i));
      l_i:=l_i+1;
    END LOOP;

    WHILE l_i < 10 LOOP
      DBMS_OUTPUT.PUT_LINE (TO_CHAR(l_i));
      l_i := l_i + 1;
    END LOOP;

    FOR l_i IN 1..500 LOOP
       DBMS_OUTPUT.PUT_LINE (TO_CHAR(l_i));
    END LOOP;

    FOR l_i IN REVERSE 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(l_i));
    END LOOP;
END;
```

PL/SQL Control Structures

- Iterative loops
  - Named loops
- Exiting loops
  - EXIT statement
- Loop skipping
  - CONTINUE

```
DECLARE
    l_i NUMBER := 0;
    l_j NUMBER := 0;
    l_s NUMBER :=0;
BEGIN
    <<outer_loop>>
    LOOP
      l_i := l_i + 1;
       <<inner_loop>>
      LOOP
            l_j := l_j + 1;
            l_s := l_s + l_i * l_j;
            EXIT inner_loop WHEN (l_j > 5);
            EXIT outer_loop WHEN ((l_i * l_j) > 15);
      END LOOP inner_loop;
      DBMS_OUTPUT.PUT_LINE('Sum:'||TO_CHAR(l_s));
      IF l_s > 100 THEN EXIT;
        END IF;
    END LOOP outer_loop;
END;
```

Accessing Data in the Database

- Selecting at most one row:
  - SELECT INTO statement

```
SELECT COUNT(*) INTO variable FROM table;
SELECT * INTO record FROM table WHERE …;
```

- Selecting Multiple rows:
  - Cursors
- Inserting and updating

```
INSERT INTO table VALUES (var1, var2, …);
```

Cursors

- Every SQL query produces a result set - cursor
  - set of rows that answer the query
  - resides on the server in the client process memory
- PL/SQL program can read the result set in interating fashon

```
select
   emp_no
  ,emp_name
  ,emp_job
from employees
 where emp_no > 500;
```

Result Set

EMP_NO	EMP_NAME	EMP_JOB	EMP_HIREDATE	EMP_DEPTNO
380	KING	CLERK	1-JAN-1982	10
381	BLAKE	ANALYST	11-JAN-1982	30
392	CLARK	CLERK	1-FEB-1981	30
569	SMITH	CLERK	2-DEC-1980	20
566	JONES	MANAGER	5-JUL-1978	30
788	SCOTT	ANALYST	20-JUL-1981	10
876	ADAMS	CLERK	14-MAR-1980	10
902	FORD	ANALYST	25-SEP-1978	20

Defining explicit cursors

- The simplest cursor:

```
CURSOR my_cursor IS  SELECT * from table;
```

- Full cursor syntax

```
CURSOR name(parameter_list) RETURN rowtype IS SELECT …;
```

  - The SQL select statement is static (hardcoded)
    - But may be parameterized
  - The return type clause is useful in packages
- Attributes
  - %FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN

Using explicit cursors

- Fetching results of a query into RECORD

```
DECLARE
    l_employees employees%ROWTYPE;
    CURSOR l_c (p_low NUMBER DEFAULT 0, p_high NUMBER DEFAULT 99) is
      SELECT * FROM employees WHERE job_id > p_low AND job_id < p_high;
BEGIN
    OPEN l_c(3,20);
    LOOP
       FETCH l_c INTO l_employees;
       EXIT WHEN l_c%NOTFOUND;
       DBMS_OUTPUT.PUT_LINE(l_employees.last_name ||  l_employees.job_id );
    END LOOP;
    CLOSE l_c;
END;
```

Implicit cursor

```
DECLARE
   l_rows number(5);
BEGIN
   UPDATE employee SET salary = salary + 1000;
   IF SQL%NOTFOUND THEN
       dbms_output.put_line('None of the salaries where  updated');
   ELSIF SQL%FOUND THEN l_rows := SQL%ROWCOUNT;
dbms_output.put_line('Salaries for ' || l_rows ||    'employees are
updated');
   END IF;
END;
```

Dynamic PL/SQL

- Execution of statement composed in strings
- For SQL which text is unknown at compiling time
  - Some parts of SQL cannot be bind by variables
    - table name
    - database link
    - …
- Be aware of SQL injections!
- Use dynamic SQL when it is really needed

Dynamic SQL & PL/SQL

- Inserting

```
sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';
EXECUTE IMMEDIATE sql_stmt USING a, b; -- using variables
```

- Selecting data from dynamic table_name

```
EXECUTE IMMEDIATE 'select id form '||table_name||' where name=:a '
using job_name returning into job_id;
```

- Dynamic PL/SQL

```
plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;';
EXECUTE IMMEDIATE plsql_block USING a, b;
```

PL/SQL Subprograms

- Named block
  - stored in the database
  - can have set of parameters
  - invocation
    - from named block
    - from anonymous blocks
    - recursively
- Subprogram types
  - Procedures
    - complex data processing
  - Functions
    - frequent, simple operations
    - returns a value

PL/SQL Subprograms

- The header specifies:
  - Name and parameter list
  - Return type (function headers)
- Parameters:
  - Any of them can have a default value
  - Parameter input modes:
    - IN (default)
      - Passes value to that cannot be changed by the subprogram
    - OUT
      - Return value. Should be initialized in the subprogram
    - IN OUT
      - Passes a value and returns updated one by subprogram

PL/SQL Procedures

- Procedure definition

```
CREATE OR REPLACE PROCEDURE EXE$RAISE_SALARY (p_emp_id  IN NUMBER
 , p_amount IN NUMBER) IS
BEGIN
   UPDATE employees SET salary = salary + p_amount
   WHERE employee_id = p_emp_id;
END EXE$RAISE_SALARY;
```

- Procedure invocation

```
EXE$RAISE_SALARY(emp_num, bonus);
EXE$RAISE_SALARY(l_amount => bonus, l_emp_id => emp_num);
EXE$RAISE_SALARY(emp_num, l_amount => bonus);
```

PL/SQL Functions

- Function definition

```
CREATE OR REPLACE FUNCTION STF$HALF_OF_SQUARE (p_original NUMBER)
RETURN NUMBER IS
BEGIN
   RETURN (p_original * p_original)/2 + (p_original * 4);
END STF$HALF_OF_SQUARE;
```

- Function invocation

```
square INTEGER := STF$HALF_OF_SQUARE(25);
```

```
select STF$HALF_OF_SQUARE( a ) from squers;
```

Subprograms privileges

- Creator/owner has full privileges on stored objects
- Invoker that is not an owner has to have EXECUTE privilege granted

```
-- USER1

create or replace function my_fuction1 is…
grant execute on my_procedure1 to user2;

-- USER2

execute user1.myprocedure;
```

- Granted privs can be checked in USER_TAB_PRIVS

Subprograms rights

- Definer rights (default for named blocks)

```
create or replace procedure procedure_name    [authid definer]
is…
```

- Invoker rights

```
create or replace function procedure_name authid current_user
is…
```

- Anonymous blocks have always invoker rights!

Error Handling

- An error interrupts the execution of the program
  - An exception is raised
- Exception to be handled
  - in the exception section or
  - will be propagated to the enclosing block
- After the exception is handled, the control passes to the enclosing block

PL/SQL Exceptions

- The programmer can create, name and raise exception
- Exceptions can by caught and handled by the user's code
- Exceptions does not rollback or commit changes!
- Categories
  - Internally defined (without name, just error code)
  - Predefined (with name and error code)
  - **User-defined** (with name, raised always explicitly)

PL/SQL Exceptions

```
DECLARE
   l_out_of_stock  EXCEPTION;
   l_number_on_handNUMBER := 0;
BEGIN
   IF l_number_on_hand < 1 THEN
    RAISE l_out_of_stock;
   END IF;
   EXCEPTION
    WHEN l_out_of_stock THEN
    DBMS_OUTPUT.PUT_LINE ( 'Encountered out of stock error' );
    WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ( 'Houston we''ve got a problem!' );
   END;
END;
```

Packages

- Group logically related PL/SQL types, items and modules
- 2 parts:
  - Specification ≡ public interface
  - Body ≡ private implementation
- Packages are global
  - Cannot be called, parameterized, or nested.
  - Package state persist for the duration of the database session

Why use Packages

- Modularity
- Encapsulation of data and functionality
- Clear specifications independent of the implementation
- Easier development
- Added functionality:
  - global variables
  - global types
- Better performance

Package Specification

- Header
- Declarations of global types and variables
- Specification of cursors
  - With RETURN clause, but no SELECT statement
- Specification of public modules

Package Specification

```
CREATE OR REPLACE PACKAGE KNL_EMP_ADM AS
  TYPE T_EMPRECTYP IS RECORD (emp_id NUMBER, sal NUMBER);
  CURSOR desc_salary RETURN T_EMPRECTYP ;
  invalid_salary EXCEPTION;

  PROCEDURE EXE$FIRE_EMPLOYEE (p_emp_id NUMBER);
  PROCEDURE EXE$RAISE_SALARY (p_emp_id NUMBER,p_amount NUMBER);
  FUNCTION STF$HIGHEST_SALARY (p_n NUMBER) RETURN T_EMPRECTYP;
END KNL_EMP_ADM;
```

Package Body

- Header
- Additional declarations of types and variables
- Specification and SELECT statements of cursors
- Specification and body of modules
- Initialization code
  - Execution and exception sections
  - Executed once when the package is first accessed

Package Body

```
CREATE OR REPLACE PACKAGE BODY KNL_EMP_ADM AS
    number_hired NUMBER;
    CURSOR desc_salary RETURN T_EMPRECTYP IS
     SELECT employee_id, salary FROM employees ORDER BY salary DESC;
    PROCEDURE EXE$FIRE_EMPLOYEE (p_emp_id NUMBER) IS
    BEGIN
     DELETE FROM employees WHERE employee_id = p_emp_id;
    END EXE$FIRE_EMPLOYEE;
    PROCEDURE EXE$RAISE_SALARY (p_emp_id NUMBER,p_amount NUMBER) IS
    ...
    BEGIN
       INSERT INTO emp_audit VALUES (SYSDATE, USER, 'EMP_ADMIN');
       number_hired := 0;
    END;
END KNL_EMP_ADM;
```

Oracle Supplied Packages

- Extend the functionality of the database
- Some example packages:
  - DBMS_JOB: for task scheduling
  - DBMS_PIPE: for communication between sessions
  - DBMS_OUTPUT: display messages to the session output device
  - UTL_HTTP: makes HTTP callouts.
  - Many others…

Triggers

- Stored procedure
- Execute automatically when:
  - data modification (DML Trigger)
    - INSERT, UPDATE, UPDATE column or DELETE
  - schema modification (DDL Trigger)
  - system event, user logon/logoff (System Trigger)
- Basic DML triggers types:
  - BEFORE statement
  - BEFORE each row modification
  - AFTER each row modification
  - AFTER statement
  - INSTEAD OF - to enable data modification by views

When To Use Triggers

- Automatic data generation
  - Auditing (logging), statistics
  - Derived data
  - Data replication
- Special referential constrains
  - Complex logic
  - Distributed constrains
  - Time based constrains
- Updates of complex views
- Triggers may introduce hard to spot interdependencies to the database schema

Trigger Body

- Built like a PL/SQL procedure
- Additionally:
  - Type of the triggering event can be determined inside the trigger using conditional predicators

```
IF inserting THEN … END IF;
```

  - Old and new row values are accessible via **`:old`** and **`:new`** qualifiers (record type variables)

Trigger Example

```
CREATE OR REPLACE TRIGGER audit_sal
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
   INSERT INTO emp_audit
   VALUES( :old.employee_id, SYSDATE, :new.salary, :old.salary );
   COMMIT;
END;
```

Jobs

- Job
  - Schedule
  - PL/SQL subprogram (but not only)
- Many possibilities for the scheduling
- Creation
  - Using DBMS_SCHEDULER internal package
    - Alternative DBMS_JOB is old and should by avoided
  - Privileges needed
    - **execute** on DBMS_SCHEDULER
    - **create job**

Jobs example

- Daily execution (everyday at 12) of *my_saved_procedure*

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB (
   job_name           =>  'my_new_job1',
   program_name       =>  'my_saved_procedure',
   repeat_interval    =>  'FREQ=DAILY;BYHOUR=12',
   comments           =>  'Daily at noon');
END;
/
```

Advantages of PL/SQL

- Tightly integrated with SQL
- Reduced network traffic
- Portability - easy deployment and distribution
- Data layer separated from client language
  - Modification without changing of application code
  - Can be shared by many platform
- Server-side periodical data maintenance (jobs)

References

- Oracle Documentation
  - http://www.oracle.com/pls/db112/homepage
- PL/SQL language reference
  - http://docs.oracle.com/cd/E11882_01/appdev.112/e25519/toc.htm
- PL/SQL packages refernece
  - http://docs.oracle.com/cd/E11882_01/appdev.112/e25788/toc.htm