7. PL/SQL

7.1 Introduction

PL/SQL = **P**rocedural **L**anguage extensions to **SQL**

Overcome declarative SQL's inability to specify control aspects of DB interaction.

Add procedural capabilities to Oracle programming.

An Oracle-specific language combining features of:

- modern, block-structured programming language
- database interaction via SQL

Example 1:

Consider the following query:

If a user attempts to withdraw more funds than they have from their account, then indicate "Insufficient Funds", otherwise update the account

A possible implementation in SQL (SQL*Plus):

```
ACCEPT person PROMPT 'Name of account holder: '
ACCEPT amount PROMPT 'How much to withdraw: '

UPDATE Accounts
SET    balance = balance - &amount
WHERE  holder = '&person' AND balance > &amount;
SELECT 'Insufficient Funds'
FROM   Accounts
WHERE  holder = '&person' AND balance <= &amount;
```

Two problems:

- doesn't express the "business logic" nicely
- performs *both* actions when $(balance - amount < amount)$

We could fix the second problem by reversing the order ($SELECT$ then $UPDATE$).

But in SQL there seems no way to avoid executing *both* the $SELECT$ *and* the $UPDATE$

PL/SQL allows us to specify the control more naturally:

```
-- A sample PL/SQL procedure
PROCEDURE withdrawal(person IN varchar2,
                     amount IN REAL        ) IS
   current REAL;
BEGIN
   SELECT balance INTO current
   FROM   Accounts
   WHERE  holder = person;
   IF (amount > current) THEN
      dbms_output.put_line('Insufficient Funds');
   ELSE
      UPDATE Accounts
      SET    balance = balance - amount
      WHERE  holder = person AND balance > amount;
      COMMIT;
   END IF;
END;
```

It can be executed if embedded into a package Y:

```
SQL>  EXECUTE Y.withdrawal('John Shepherd', 100.00);
```

7.2 PL/SQL Syntax

PL/SQL is *block-structured*, where a block consists of:

```
DECLARE
   declarations for
   constants, variables and local procedures
BEGIN
   procedural and SQL statements
EXCEPTION
   exception handlers
END;
```

The syntax is similar to Ada/Modula/Pascal.

7.3 PL/SQL Engine

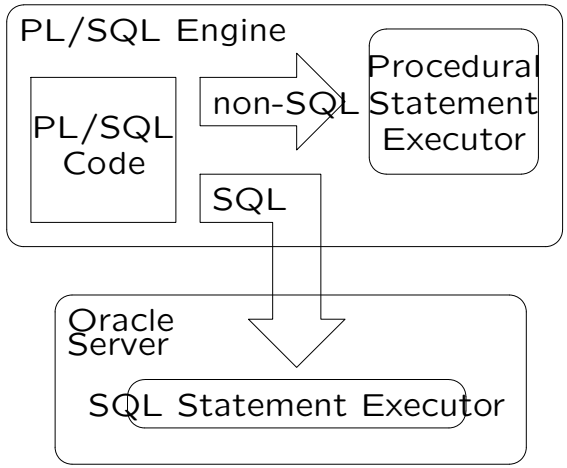
PL/SQL is implemented via a *PL/SQL engine:*

- which can be embedded in Oracle tools (e.g. Forms, Reports, ...)
- which is also usually available in the Oracle server

The PL/SQL engine

- executes procedural code and manages PL/SQL variables
- calls Oracle server to handle SQL statements

More tightly integrated with Oracle than JDBC, SQL-in-C, etc.


PL/SQL Engine
PL/SQL
Code
non-SQL
Procedural
Statement
Executor
SQL
Oracle
Server
SQL Statement Executor

7.4 Data Types

PL/SQL constants and variables can be defined using:

- standard SQL data types $(CHAR, DATE, NUMBER, ...)$
- built-in PL/SQL types $(BOOLEAN, BINARY_{INTEGER})$
- PL/SQL structured types $(RECORD, TABLE)$

Users can also define new data types in terms of these.

There is also a CURSOR type for interacting with SQL.

7.4.1 Record Types

Correspond to Modula $RECORD$s or C $struct$s, and also closely related to SQL table row type.

New record types can be defined via:

```
TYPE TypeName IS RECORD
     (Field_1 Type_1, Field_2 Type_2, ...);
```

Example:

```
TYPE Student IS RECORD (
   id#    NUMBER(6),
   name   VARCHAR(20),
   course NUMBER(4)
);
```

Row types of defined Oracle tables are available to PL/SQL:

```
RecordName TableName%ROWTYPE;
```

Record components are accessed via *Var.Field* notation.

```
fred Student;
...
fred.id#    := 123456;
fred.name   := 'Fred';
fred.course := 3978;
```

Record types can be nested.

```
 TYPE Day IS RECORD
   (day NUMBER(2), month NUMBER(2), year NUMBER(4));

TYPE Person IS RECORD
   (name VARCHAR(20), phone VARCHAR(10), birthday Day);
```

7.4.2 Constants and Variables

Variables and constants are declared by specifying:

```
Name [ CONSTANT ] Type [ := Expr ] ;
```

Examples:

```
amount       INTEGER;
part_number  NUMBER(4);
in_stock     BOOLEAN;
owner_name   VARCHAR(20);

tax_rate     CONSTANT REAL := 0.23;
max_credit   CONSTANT REAL := 5000.00;

my_credit    REAL := 2000.00;
```

Variables can also be defined in terms of:

- the type of an existing variable or table column
- the type of an existing table row (implict $RECORD$ type)

Examples:

```
quantity    INTEGER;
start_qty   quantity%TYPE;

employee    Employees%ROWTYPE;

name        Employees.name%TYPE;
```

7.5 Assigning Values to Variables

A standard assignment operator is available:

```
in_stock := FALSE;
tax      := price * tax_rate;
amount   := TO_NUMBER(SUBSTR('750 dollars',1,3));
person1  := person2;  -- record assignment
```

Values can also be assigned via *SELECT...INTO*:

```
SELECT price * (1+tax_rate) INTO cost
FROM   StockList
WHERE  item = 'Cricket Bat';
total := total + cost;
```

SELECT...INTO can assign a whole row at once:

```
DECLARE
   emp     Employees%ROWTYPE;
   my_name VARCHAR(20);
   pay     NUMBER(8,2);
BEGIN
   SELECT * INTO emp
   FROM   Employees
   WHERE  id# = 966543;
   my_name := emp.name;
   ...
   SELECT name,salary INTO my_name,pay
   FROM   Employees
   WHERE  id# = 966543;
END;
```

7.6 Control Structures

PL/SQL has conventional set of control structures:

- **;** for sequence (note: ; is a *terminator*)
- **IF** for selection
- **FOR**, **WHILE**, **LOOP** for repetition

Along with exceptions to interrupt normal control flow.

And a NULL statement to do nothing.

7.6.1 Selection

Selection is expressed via:

```
IF Cond_1 THEN Statements_1;
ELSIF Cond_2 THEN Statements_2;
ELSIF Cond_3 THEN Statements_3;
...
ELSE Statements_n;
END IF;
```

ELSIF and **ELSE** parts are optional.

Example: See the preceding example.

7.6.2 Iteration

Iteration is expressed via:

```
   LOOP
   Statements;
   ...  EXIT  ...
   MoreStatements;
END LOOP;

WHILE Cond LOOP
   Statements;
END LOOP;

FOR Var IN LoVal..HiVal LOOP
   Statements;
END LOOP;
```

Example: See examples in the section **Cursors**.

EXIT WHEN Cond;
is shorthand for
IF Cond THEN EXIT; END IF;

Loops can be named to allow multi-level exits

```
<<outer>>LOOP
   ...
   <<inner>>LOOP
      ... EXIT outer WHEN i > 100; ...
   END LOOP;
   ...
END LOOP;
```

7.7 Cursors

A *cursor* is a variable that can be used to access one row of the result of a particular SQL query.

Cursors can move sequentially from row to row (cf. file pointers in C).

Every SQL query statement in PL/SQL has an implicit cursor.

It is also possible to declare and manipulate cursors explicitly:

```
DECLARE
   CURSOR e IS
         SELECT * FROM Employees
         WHERE salary > 30000.00;
BEGIN
   ...
```

Cursors provide flexibility in processing rows of a query.

Simplest way to deal with a cursor is to loop over all rows using a FOR loop:

```
DECLARE
   CURSOR e IS
         SELECT * FROM Employees
         WHERE salary > 30000.00;
   total INTEGER := 0;
BEGIN
   FOR emp IN e LOOP
      total := total + emp.salary;
   END LOOP;
   dbms_output.put_line(
      'Total Salaries: ' || total);
END;
```

Cursor loop variables are implicitly declared as the ROWTYPE for the SELECT result.

E.g. *emp* is implictly declared as Employees

The cursor FOR loop is convenient shorthand for iteration implemented using the basic cursor operations as follows:

```
DECLARE
      CURSOR e IS
            SELECT * FROM Employees
            WHERE salary > 30000.00;
      total INTEGER := 0;
      emp e%type;
BEGIN
   OPEN e;
   LOOP
      FETCH e INTO emp;
      EXIT WHEN e%NOTFOUND;
      total := total + emp.salary;
   END LOOP;
   CLOSE e;
END;
```

The FETCH operation can also extract components of a row:

```
   FETCH e INTO my_id, my_name, my_salary;
```

There must be one variable, of the correct type, for each column in the result.

Cursors have several built-in attributes:

- FOUND ... true whenever a row is successfuly fetched
- ISOPEN ... true if cursor is currently active
- NOTFOUND ... true after last row has been read
- ROWCOUNT ... returns number of rows in cursor-relation

Yet another method for cursor iteration:

```
-- assume declarations as before
OPEN e;
FOR i IN 1..e%ROWCOUNT LOOP
   FETCH e INTO emp;
   -- process emp in some way
END LOOP;
```

The CURRENT OF operator allows us to operate on the current tuple via SQL.

Example: give low-paid workers a higher pay rise

```
DECLARE
   CURSOR Emps IS SELECT * FROM Employees;
BEGIN
   FOR e IN Emps LOOP
      IF e.salary < 20000.00 THEN
         UPDATE Employees SET Employees.salary
                              = Employees.salary*1.20
         WHERE CURRENT OF Emps;
      ELSIF e.salary > 80000.00 THEN
         UPDATE Employees SET Employees.salary
                              = Employees.salary*1.05
         WHERE CURRENT OF Emps;
      ENDIF
   END LOOP;
END:
```

7.8 Cursor Example

Consider the problem of buying football players to make up a team with the constraint of a ”salary cap”.

We want to buy the best (most expensive) players first, but we have to stop taking them once we reach the salary cap.

Assume we can then make up the rest of the team with ”cheap” young players.

```
DECLARE
   CURSOR potentialPlayers IS
         SELECT * FROM Players
         ORDER BY salary DESCENDING;
   totalCost NUMBER := 0.00;
   salaryCap CONSTANT NUMBER := 100000000.00;
BEGIN
   FOR p IN potentialPlayers LOOP
      EXIT WHEN totalCost+p.salary > salaryCap;
      dbms_output.put_line(p.name);
      totalCost := totalCost + p.salary;
   END LOOP;
END;
```

7.9 Tables

PL/SQl TABLEs combine characteristics of SQL tables and C/Pascal arrays.

Like SQL tables:

- consist of records (must have a numeric primary key)
- can grow/shrink as elements are added/removed

Like C/Pascal arrays:

- access individual elements via an index
  (however, the set of index values can be non-contiguous)

New table types can be defined via:

```
TYPE TypeName IS TABLE OF BaseType
    INDEX BY BINARY_INTEGER;
```

Example: a type for tables of employees

```
TYPE EmpTab IS TABLE OF Employees%ROWTYPE
      INDEX BY BINARY_INTEGER;

first_table    EmpTab;
another_table  EmpTab;
```

Elements of tables are accessed via *Table (Expr)* notation.

The expression must be convertable to type *BINARY_INTEGER* (e.g. INT).

Example: setting up a table from a relation

```
DECLARE
   -- assume type declaration from above
   rich_emps EmpTab;
   n  INTEGER;
BEGIN
   FOR emp IN (SELECT * FROM Employees) LOOP
      n := n + 1;
      rich_emps(n) := emp;
   END LOOP;
END;
```

A number of built-in operators are defined on PL/SQL tables:

```
COUNT   Number of elements currently in table

DELETE  Deletes one or more elements from table

FIRST   Returns smallest index into table

LAST    Returns largest index into table

NEXT    Returns next defined index into table

EXISTS  Tests whether index value is valid
```

Example: Scanning over an arbitrary Employees table

```
DECLARE
   -- assume type declaration from above
   rich EmpTab;
   i    INTEGER;
BEGIN
   i = rich.FIRST;
   WHILE i <= rich.LAST LOOP
      IF NOT rich.EXISTS(i) THEN
         -- should never happen
         dbms_output.put_line('Something''s wrong.');
      ELSE
         dbms_output.put_line(rich(i).name);
      END IF;
      i = rich.NEXT;
   END LOOP;
END;
```

Example: Alternative scan method, while deleting

```
DECLARE
   -- assume type declaration from above
   rich EmpTab;
   i    INTEGER;
BEGIN
   -- Generates all possible indexes
   FOR i IN rich.FIRST..rich.LAST LOOP
      IF NOT rich.EXISTS(i) THEN
     NULL;
         -- this is now necessary
         -- to ignore invalid indexes
      ELSE
         rich.DELETE(i);
      END IF;
   END LOOP;
END;
```

Note: could have done much more simply via *rich.DELETE*;

Example: Dumping a PL/SQL table into an Oracle TABLE

```
DECLARE
   -- assume type declaration from above
   emps EmpTab;
   i    INTEGER;
BEGIN
   i = emps.FIRST;
   WHILE i <= emps.LAST LOOP
      -- Unfortunately, can't do this
      --    INSERT INTO Employees VALUES emps(i);
      -- so we do this ...
      INSERT INTO Employees VALUES
         (emps(i).id#, emps(i).name, emps(i).salary);
      i = emps.NEXT;
   END LOOP;
END;
```

7.10 Exceptions

An *exception* is an unusual/erroneous condition encountered during execution:

- system error (e.g. ”out of memory”)
- error caused by user program
- warning issued by application

PL/SQL’s exception handling allows these to be handled ”cleanly” in a central place.

Syntax for exception handlers:

```
BEGIN
    ...  Statements ...
 EXCEPTION
    WHEN ExcepName_1 THEN Statements_1;
    WHEN ExcepName_2 THEN Statements_2;
    ...
 END;
```

If an error occurs in *Statements*, control is transferred to:

- the exception handler in this block
- the exception handler at the next enclosing block
- ... and so on out to the system level

Example: Computing stock-market price/earnings ratio

```
DECLARE
   pe_ratio NUMBER(5,1);
BEGIN
   SELECT price/earnings INTO pe_ratio
   FROM Stocks WHERE company_name = 'Acme';
   INSERT INTO Statistics(co_name, ratio)
      VALUES ('Acme', pe_ratio);
   COMMIT;
EXCEPTION
   WHEN ZERO_DIVIDE THEN -- divide-by-zero errors
      INSERT INTO Statistics(co_name, ratio)
         VALUES ('Acme', 0.0);
   -- other exception handlers
END;
```

7.11 Predefined Exceptions

PL/SQL provides exceptions for low-level/system errors:

```
INVALID_CURSOR  Attempt to use non-open cursor

INVALID_NUMBER  Non-numeric-looking string used in context
        where number needed

NO_DATA_FOUND   SELECT..INTO returns no results

NOT_LOGGED_ON   Attempted SQL operation without being
        connected to Oracle

STORAGE_ERROR   PL/SQL store runs out or is corrupted

VALUE_ERROR Arithmetic conversion, truncation,
        size-constraint error
```

7.11.1 User-defined Exceptions

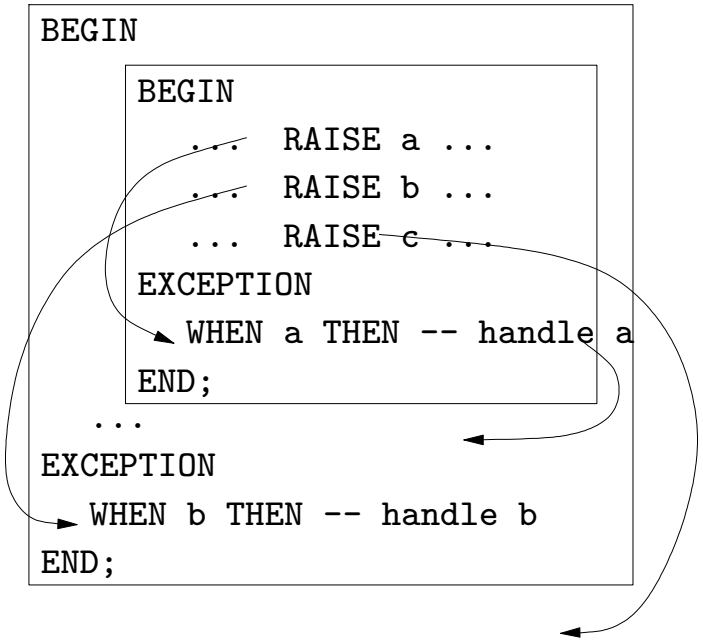
Exceptions are defined by name; used by RAISE.

Example:

```
DECLARE
   outOfStock EXCEPTION;
   qtyOnHand  INTEGER;
BEGIN
   ...
   IF qtyOnHand < 1 THEN
      RAISE outOfStock;
   END IF;
   ...
EXCEPTION
   WHEN outOfStock THEN
      -- handle the problem
END;
```

User-defined exceptions are local to a block and its sub-blocks.

7.11.2 Exception Propagation



Unhandled exceptions are passed to the system containing the PL/SQL engine.

Exceptions can be re-raised in their handler; passes control to handler in outer block.

Exceptions can be ignored by using NULL as the handler.

Generic exception handler implemented by:

```
WHEN Exception_1 THEN Statements_1;
WHEN Exception_2 THEN Statements_2;
...
WHEN OTHERS THEN Statements_n;
END;
```

7.12 Transactions

A transaction is a sequence of SQL statements to accomplish a single task.

Example: Transfer funds between bank accounts.

```
-- assume source_acct, dest_acct, amount,
-- and source_balance are defined}
BEGIN
   SELECT balance INTO source_balance
   FROM Accounts WHERE acct# = source_acct;
   -- check whether sufficient funds
   UPDATE Accounts SET balance = balance-amount
   WHERE acct# = source_acct;
   UPDATE Accounts SET balance = balance+amount
   WHERE acct# = dest_acct;
   COMMIT;
END;
```

Oracle treats such a sequence as an indivisible unit, to ensure that database is left in a consistent state.

PL/SQL gives fine-grained control over progress of transaction.

This also gives responsibility to ensure that transaction completes ok.

The first SQL statement begins a transaction.

COMMIT forces any changes made to be written to database.

ROLLBACK restores database to state at start of transaction.

Both COMMIT and ROLLBACK finish the transaction.

Can achieve finer-grained rollback via savepoints:

```
BEGIN
   ...
   UPDATE Employees SET ...  WHERE id# = emp_id;
   DELETE FROM Employees WHERE ...
   ...
   SAVEPOINT more_changes;
   ...
   -- make changes to Employees
   -- possibly raise some_exception
   ...
   COMMIT;
EXCEPTION
   WHEN some_exception THEN ROLLBACK TO more_changes;
END;
```

7.13 Locking with Cursors

When accessing tables via a cursor, normally the table is locked.

PL/SQL provides a mechanism to lock individual rows instead:

```
DECLARE
   CURSOR managers IS
      SELECT emp#, job, salary
      FROM Employees WHERE job = 'Manager'
      FOR UPDATE OF salary;
BEGIN
   FOR e IN managers LOOP
      UPDATE Employees SET salary = new_sal
      WHERE CURRENT OF managers;
      COMMIT;
   END LOOP;
END;
```

7.14 Procedures and Functions

PL/SQL provides packaging mechanism for small blocks of procedural code:

```
PROCEDURE ProcName(ArgList) IS
   declarations;
BEGIN
   statements;
   EXCEPTION handlers;
END ProcName;

FUNCTION FuncName(ArgList) RETURN Type IS
   eclarations;
 BEGIN
   statements; -- including RETURN Expr;
   EXCEPTION handlers;
 END FuncName;
```

Each argument has a *mode*:

```
IN  parameter is used for input only (default)

OUT paramter is used to return a result

IN OUT  returns result, but initial value is used
```

Can also specify a DEFAULT value for each argument.

Procedures can be called in the usual manner:

- same number of arguments as formal parameters
- arguments given in same order as formal parameters

Or can be called via named parameters e.g.

```
PROCEDURE p(a1 IN NUMBER DEFAULT 13,
            a2 OUT CHAR,  a3 IN OUT INT)
...
p(a2 => ch, a3 => my_int);
```

7.15 Procedure Example A procedure to raise the salary of an employee:

```
PROCEDURE raise(emp# IN INTEGER, increase IN REAL) IS
   current_salary REAL;
   salary_missing EXCEPTION;
BEGIN
   SELECT salary INTO current_salary
   FROM Employees WHERE id# = emp#;
   IF current_salary IS NULL THEN
      RAISE salary_missing;
   ELSE
      UPDATE Employees
  SET salary = salary + increase
      WHERE id# = emp#;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN INSERT INTO Audit
     VALUES (emp#, "Unknown employee");
   WHEN salary_missing THEN INSERT INTO Audit
     VALUES (emp#, "Null salary");
END;
```

To run the procedure, you may either:

- embed it into a package, or
- embed it into a PL/SQL program, or
- store it by adding create or replace as a head.

Then execute it.