Database Procedural Programming PL/SQL and Embedded SQL

CS2312

PL/SQL

* PL/SQL is Oracle's procedural language extension to SQL.
* PL/SQL combines SQL with the procedural functionality of a structured programming language, such as IF ... THEN, WHILE, and LOOP.
* The PL/SQL engine used to define, compile, and execute PL/SQL program units.
* A component of many Oracle products, including Oracle Server.

Procedures and Functions

* A set of SQL and PL/SQL statements grouped together as a unit (*block*) to solve a specific problem or perform a set of related tasks.
* An *anonymous block* is a PL/SQL block that appears within your application and it is not named or stored in the database. In many applications, PL/SQL blocks can appear wherever SQL statements can appear.
* A *stored procedure* is a PL/SQL block that Oracle stores in the database and can be called by name from an application.
* Functions always return a single value to the caller; procedures do not return values to the caller.
* Packages are groups of procedures and functions.

Procedure PL/SQL Example

```
CREATE PROCEDURE credit_labmark (sno NUMBER, cno CHAR,
   credit NUMBER) AS
         old_mark  NUMBER;
         new_mark  NUMBER;
BEGIN
  SELECT labmark INTO old_mark FROM enrol
     WHERE studno = sno and courseno = cno FOR UPDATE OF
   labmark;
  new_ mark := old_ mark + credit;
  UPDATE enrol SET labmark = new_mark
     WHERE studno = sno and courseno = cno;
  COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
     INSERT INTO enrol(studno, courseno, labmark, exammark)
       VALUES(sno, cno, credit, null);
  WHEN OTHERS THEN  ROLLBACK;
END credit_labmark;
```

Locks enrol

SQL statement.

PL/SQL statement.

EXECUTE credit_labmark (99234,'CS2312',20)

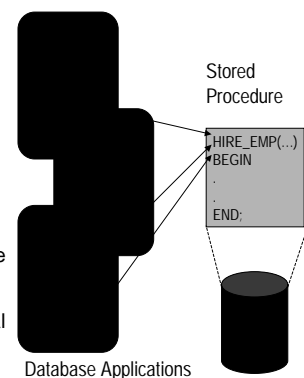
Function

```
create function get_lab_mark(sno number, cno char)
     return number
as f_lab_mark number;
  no_mark exception;
  begin
     select labmark
           into f_lab_mark from enrol
           where studno = sno and courseno = cno;
     if f_lab_mark is null
     then raise no_mark;
     else return(f_lab_mark);
     end if
  exception
     when no_mark then …..return(null);
  end;
```
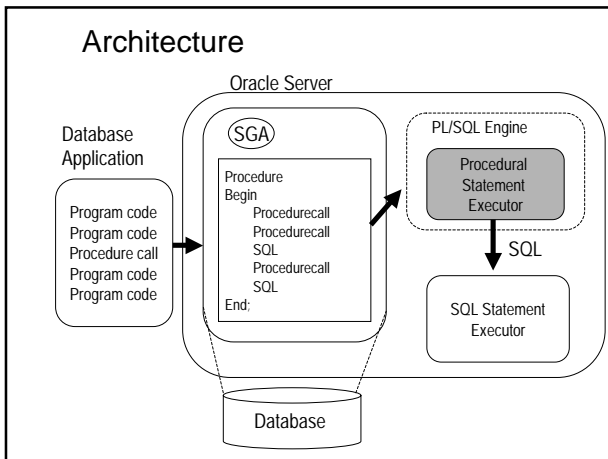

Stored Procedures

Created in a user's schema and stored, centrally, in *compiled form* in the database as a named object that can be:

* interactively executed by a user using a tool like SQL*Plus
* called explicitly in the code of a database application, such as an Oracle Forms or a Pre compiler application, or in the code of another procedure or trigger

When PL/SQL is not stored in the database, applications can send blocks of PL/SQL to the database rather than individual SQL statements → reducing network traffic. .

Stored Procedure

```
HIRE_EMP(…)
BEGIN
.
.
END;
```

Database Applications

Architecture

Oracle Server

Database Application

SGA

PL/SQL Engine

Procedural Statement Executor

Procedure
Begin
    Procedurecall
    Procedurecall
    SQL
    Procedurecall
    SQL
End;

Program code
Program code
Procedure call
Program code
Program code

SQL

SQL Statement Executor

Database

Benefits of Stored Procedures I

* Security
  * Control data access through procedures and functions.
  * E.g. grant users access to a procedure that updates a table, but not grant them access to the table itself.
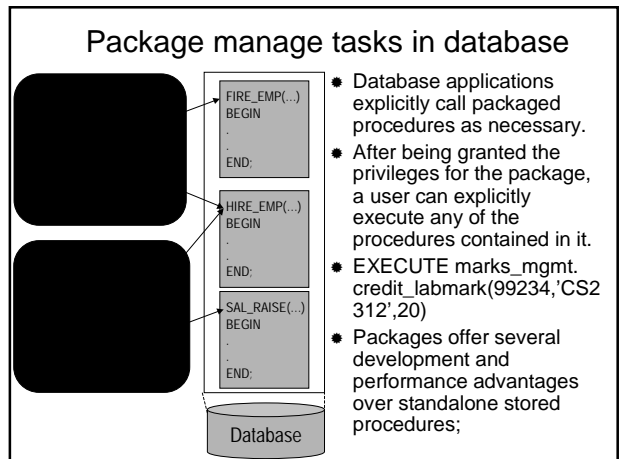
* Performance
  The information is sent only once between database and application and thereafter invoked when it is used.
  * Network traffic is reduced compared with issuing individual SQL statements or sending the text of an entire PL/SQL block
  * A procedure's compiled form is readily available in the database, so no compilation is required at execution time.
  * The procedure might be cached

Benefits of Procedures II

* Memory Allocation
  * Stored procedures take advantage of the shared memory capabilities of Oracle
  * Only a single copy of the procedure needs to be loaded into memory for execution by multiple users.

* Productivity
  * By designing applications around a common set of procedures, you can avoid redundant coding and increase your productivity.
  * Procedures can be written to insert, update, or delete rows from a table and then called by any application without rewriting the SQL statements necessary to accomplish these tasks.
  * If the methods of data management change, only the procedures need to be modified, not all of the applications that use the procedures.

Benefits of Procedures III

* Integrity
  * Stored procedures improve the integrity and consistency of your applications. By developing all of your applications around a common group of procedures, you can reduce the likelihood of committing coding errors.
  * You can test a procedure or function to guarantee that it returns an accurate result and, once it is verified, reuse it in any number of applications without testing it again.
  * If the data structures referenced by the procedure are altered in any way, only the procedure needs to be recompiled; applications that call the procedure do not necessarily require any modifications.

Packages

* A method of encapsulating and storing related procedures, functions, variables, cursors and other package constructs together as a unit in the database for continued use as a unit.
* Similar to standalone procedures and functions, packaged procedures and functions can be called explicitly by applications or users.
  * Organize routines
  * Increased functionality (e.g. global package variables can be declared and used by any procedure in the package) and
  * Increased performance (e.g. all objects of the package are parsed, compiled, and loaded into memory once).

Package manage tasks in database

FIRE_EMP(...)
BEGIN
.
.
.
END;

HIRE_EMP(...)
BEGIN
.
.
END;

SAL_RAISE(...)
BEGIN
.
.
END;

Database

* Database applications explicitly call packaged procedures as necessary.
* After being granted the privileges for the package, a user can explicitly execute any of the procedures contained in it.
* EXECUTE marks_mgmt. credit_labmark(99234,'CS2 312',20)
* Packages offer several development and performance advantages over standalone stored procedures;
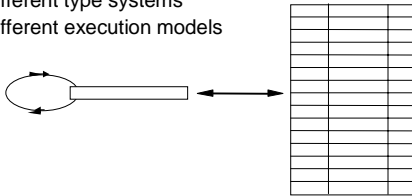
Benefits of Packages

- Encapsulation of related procedures and variables providing:
  - Better organization during the development process and for granting privileges
- Declaration of *public* and *private* procedures, variables, constants, and cursors
- Better performance
  - An entire package is loaded into memory when a procedure within the package is called for the first time in one operation, as opposed to the separate loads required for standalone procedures. When calls to related packaged procedures occur, no disk I/O is necessary to execute the compiled code already in memory.
  - A package body can be replaced and recompiled without affecting the specification. Objects that reference a package's constructs (always via the specification) need not be recompiled unless the package specification is also replaced. Unnecessary recompilations can be minimized, so in less impact on overall database performance.

Triggers vs Procedures and Packages

- Triggers are similar to stored procedures. A trigger can include SQL and PL/SQL statements to execute as a unit and can invoke stored procedures. Triggers are stored in the database separate from their associated tables.
- Procedures and triggers differ in the way that they are invoked.
  - A procedure is explicitly executed by a user, application, or trigger.
  - Triggers (one or more) are implicitly fired (executed) by Oracle when a triggering INSERT, UPDATE, or DELETE statement is issued, no matter which user is connected or which application is being used.

Retrieval: Impedance Mismatch

- What happens when the query returns several rows? The host variables can only hold one value.
- Oracle will only pass the *first* row returned by the query to the PL/SQL block (or host language program).
- Re-executing the SELECT operation will only run the query again and so the first row will be selected again.
- Different type systems
- Different execution models

Cursors

- When a query returns multiple rows a cursor must be declared to process each row returned by the query and to keep track of which row is currently being processed.
- The rows returned by a query are stored in an area called the *Active Set*.
- A cursor can be thought of as pointing to a row in the active set.

Cursors and Retrieval

```
PROCEDURE apply_marks IS
CURSOR marks_cursor IS
  SELECT sno, cno, kind, amount FROM marks
WHERE status = 'Pending' ORDER BY time_tag FOR UPDATE OF
  marks;
BEGIN
  FOR marks IN marks_cursor LOOP   /* implicit open and
  fetch */
    new_status := 'Accepted';
    IF marks.kind = 'L' THEN
      credit_labmark(marks.sno, marks.cno, marks.amount);
      ELSIF trans.kind = 'E' THEN
        credit_exammark(marks.sno, marks.cno,
  marks.amount);
      ELSE new_status := 'Rejected';
    END IF;
    UPDATE marks SET status = new_status
          WHERE CURRENT OF marks_cursor;
  END LOOP; COMMIT;
END apply_marks;
```

Embedded SQL

- SQL statements placed within a program. The source program is called the *host program*, and the language in which it is written is called the *host language*
- You can execute any SQL statement using *embedded SQL* statements just as if you were in SQL*Plus.
  - CREATE, ALTER and DROP database tables
  - SELECT, INSERT, UPDATE and DELETE rows of data
  - COMMIT transactions (make any changes to the database permanent)

Embedded SQL Statements

* Embedded SQL statements incorporate DDL, DML, and transaction control statements within a procedural language program. They are used with the Oracle pre-compilers, e.g. Pro*C.
* Embedded SQL statements enable you to
  * define, allocate, and release cursors (DECLARE CURSOR, OPEN, CLOSE)
  * declare a database name and connect to Oracle (DECLARE DATABASE, CONNECT)
  * assign variable names (DECLARE STATEMENT)
  * initialize descriptors (DESCRIBE)
  * specify how error and warning conditions are handled (WHENEVER)
  * parse and execute SQL statements (PREPARE, EXECUTE, EXECUTE IMMEDIATE)
  * retrieve data from the database (FETCH).

Executable and Declarative Statements

* Embedded SQL includes all the interactive SQL statements plus others that allow you to transfer data between Oracle and a host program. There are two types of embedded SQL statements:
* Executable:
  * used to connect to Oracle, to define, query and manipulate Oracle data, to control access to Oracle data and to process transactions. They can be placed wherever host-language executable statements can be placed.
* Declarative:
  * do not operate on SQL data. Use them to declare Oracle objects, communication areas and SQL variables which will be used by Oracle and your host program. They can be placed wherever host-language declarations can be placed.

Binding Variables

* A host variable is prefixed with a colon (:) in SQL statements but must not be prefixed with a colon in C statements.

```
EXEC SQL BEGIN DECLARE SECTION;
    INT sno;
    VARCHAR cno[5];
    INT labmark;
EXEC SQL END DECLARE SECTION;

...
EXEC SQL SELECT labmark INTO :labmark
  FROM enrol
        WHERE studno = :sno and courseno = :cno
```

database attribute

host variable

* The case of the host variable is significant when referencing them.

SELECT

INTO clause specifies the host variables which will hold the values of the attributes returned.

Attributes in the staff table.

```
EXEC SQL SELECT courseno, subject
INTO :courseno, :subject
FROM course
WHERE courseno = :menu_selection;
```

Host variable used to supply the WHERE clause with a value to base the query on. In SQL*Plus this would be done using a literal value. Pro*C allows variables to be used to specify a value. Host variables used in this way must contain a value before the SELECT statement is used.

Example

Declare any host variables
```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR studname[21];
    VARCHAR cno[5];
    INT labmark;
    VARCHAR o_connect_uid[18];
EXEC SQL END DECLARE SECTION;
```
Include the error handlers
```
EXEC SQL INCLUDE sqlca;
EXEC SQL INCLUDE oraca;
```
Log on procedure
```
void Oracle_Connect(void)              {
(void)strcpy(o_connect_uid.arr,"/@t:ora-srv:mucs7");
o_connect_uid.len = strlen(o_connect_uid.arr);
EXEC SQL CONNECT :o_connect_uid;
}
```

Connect to Oracle Server and Do the Query
```
main()
{ EXEC SQL WHENEVER SQLERROR DO sqlerror()
EXEC ORACLE OPTION (ORACA=YES)
oraca.orastxtf = 1; Oracle_Connect(); printf("Connected to
Oracle\n");
```
* **Cursor for query**
```
EXEC SQL DECLARE studcursor CURSOR FOR
    SELECT s.name, e.courseno, e.labmark,
    FROM student s, enrol e WHERE s.studno = e.studno;
```
* **Do the query**
```
EXEC SQL OPEN studcursor;
printf("Name/Course/LabMark\n");
```
* **Loop to fetch rows**
```
while (sqlca.sqlcode == 0) {
        EXEC SQL FETCH studcursor
        INTO :studname, :cno, :labmark
        printf("%s,%s,%d", studname, cno, labmark);
}
printf("%ld rows selected.\n",sqlca.sqlerrd[2]);
EXEC SQL CLOSE studcursor;
EXEC SQL COMMIT WORK RELEASE;
exit(1);}
```

Examples of Packages and Procedures

Create Package Specification

```
create package marks_mgmt (null) as

max_mark  CONSTANT NUMBER := 100.00;

PROCEDURE apply_marks;

PROCEDURE enter_marks(sno number,cno
  char, kind char, credit number);

end marks_mgmt;
```

Create Package Body

```
CREATE PACKAGE BODY marks_mgmt AS
new_status  CHAR(20);   /* Global variable to record status of
   transaction being applied. Used for update in enter_marks. */

PROCEDURE do_journal_entry (sno NUMBER, cno CHAR, kind
   CHAR) IS
/* Records a journal entry for each marks credit applied by the
   enter_marks procedure. */
BEGIN
   INSERT INTO journal
       VALUES (sno, cno, kind, sysdate);
   IF kind = 'L' THEN new_status := 'Lab credit';
   ELSIF kind = 'E' THEN new_status := 'Exam credit';
   ELSE new_status := 'New enrolment';
   END IF;
END do_journal_entry;
```

Create Package Body I

```
CREATE PROCEDURE credit_labmark (sno NUMBER, cno CHAR,
   credit NUMBER) AS

   old_mark  NUMBER; new_mark  NUMBER;
mark_overflow EXCEPTION;
  BEGIN
     SELECT labmark INTO old_mark FROM enrol
       WHERE studno = sno and courseno = cno
       FOR UPDATE OF labmark;
     new_ mark := old_ mark + credit;

     IF new_mark <= max_mark THEN
       UPDATE enrol SET labmark = new_mark
       WHERE studno = sno and courseno = cno ;
        do_journal_entry(sno, cno, L);
     ELSE RAISE mark_overflow
     ENDIF;
```

Create Package Body II

```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
          /* Create new enrolment if not found */
   INSERT INTO enrol (studno, courseno, labmark, exammark)
      VALUES(sno, cno, credit, null);
  do_journal_entry(sno, cno, 'N');
   WHEN mark_overflow THEN
      new_status := 'Mark Overflow';
   WHEN OTHERS THEN
     /* Return other errors to application */
     new_status := 'Error: ' || SQLERRM(SQLCODE);
END credit_labmark;

CREATE PROCEDURE credit_exammark (sno NUMBER, cno CHAR,
   credit NUMBER) AS…
END credit_exammark;
```

Create Package Body

```
PROCEDURE apply_marks IS …. complete shortly...
END apply_marks;

PROCEDURE enter_marks(sno  NUMBER, cno CHAR, kind
   CHAR, credit NUMBER) IS
/* A new mark is always put into this 'queue' before
   being applied to the specified enrolment instance by
   the APPLY_MARKS procedure. */
BEGIN
   INSERT INTO marks
   VALUES (sno, cno, kind, amount, 'Pending', sysdate);
   COMMIT;
END enter_marks;

END marks_mgmt ; /* end package */
```

Additional material

An error handling procedure

```
void sqlerror(void)
{ int o_errl;
  int len = 550;
  char o_err[550];

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = NULL;
  printf("\nOracle Error:\n%s", sqlca.sqlerrm.sqlerrmc);

  oraca.orastxt.orastxtc[oraca.orastxt.orastxtl] = NULL;
  printf("ERROR statement:%s\n", oraca.orastxt.orastxtc);

  sqlglm(o_err, &len, &oerrl);
  o_err[o_errl] = NULL;
  printf("ERROR Details: %s\n",o_err);

  oraca.orasfnm.orasfnmc[oraca.orasfnm.orasfnml] = NULL;
  printf("ERROR at line %ld in %s\n",
  oraca.oraslnr,oraca.orasfnm.orasfnmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(0);}
```