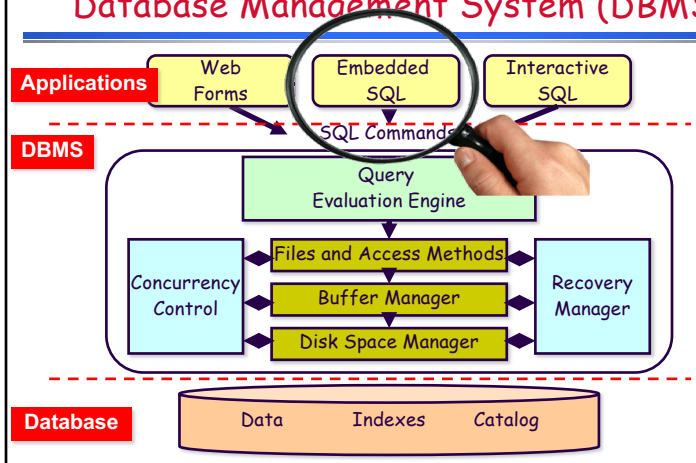
Database Programming at Large

Stored Procedures and Embedded SQL

Database Management System (DBMS)



Database Programming

- Objective:
  - To access a database from an **application** program (as opposed to interactive interfaces)
- Why?
  - An interactive interface is convenient but not sufficient
    - A majority of database operations are made thru application programs (increasingly thru web applications)

Database Programming Approaches

- Embedded commands:
  - Database commands are **embedded** in a general-purpose programming language
- Library of database functions:
  - Available to the host language for database calls; known as an **API** (Application Program Interface)
  - *e.g., JDBC, ODBC, PHP*
- A brand new, full-fledged language
  - e.g., Oracle PL/SQL
  - **P**rocedural **L**anguage extensions to SQL

Approach 3: SQL/PL

- Functions/procedures can be written in SQL itself, or in an external programming language
- Functions are very useful with specialized data types
  - E.g. functions to check if polygons overlap, or to compare images for similarity
- Some databases support **table-valued functions**, which can return a relation as a result
- SQL3 also supports a rich set of imperative constructs
  - Loops, if-then-else, case, assignment + exception handling
  - Similar to CSH script language
- Many DBMS have proprietary procedural extensions to SQL that differ from SQL3.

SQL Functions

authors (author, title, author_order)

- Definition of a Function

```
create or replace function author_count (name varchar(20))
    return integer
begin
    declare a_count integer;            -- local variable declaration
    select count(author) into a_count   -- into is a tuple assignment operator
    from authors
    where authors.title=name;
    return a_count;
end;
/
```

```
SELECT title
FROM books4
WHERE author_count(title)> 1;
```

- '/': Executes a PL/SQL block
- Invocation ?

SQL Procedures

- Definition of a procedure:

```
create or replace procedure author_count_proc (in title varchar(20),
                                               out a_count integer )
  begin
    select count(author) into a_count
    from authors
    where authors.title = title;
  end;
/
```

- Parameters Options: **IN, OUT, INOUT**
  - **Oracle syntax:** (title **in** varchar(20), a_count **out** integer)
- Invocation ?

ANSI SQL Procedures: Invocation

- Procedures can be invoked either within a trigger, an SQL procedure, or from embedded SQL, using the ***Call*** statement.
- E.g., from an SQL procedure block

```
begin
declare a_count integer;
call author_count_proc(`Database Systems' , a_count);
end;
```

- SQL3 allows name **overloading** for function and procedures, as long as the number or types of arguments is different.

ANSI SQL: Procedures in Triggers

```
CREATE OR REPLACE TRIGGER Update_ShipDate
    AFTER INSERT OR UPDATE OF ShipDate
    ON Orders
    FOR EACH ROW
    BEGIN ATOMIC
        CALL UpdateShipDate(:new)
    END;
/
```


Oracle PL/SQL Procedure Invocation

- two ways to execute a procedure.

1) From the SQL prompt:

   EXECUTE [or EXEC] procedure_name;

2) Within another procedure – simply use the procedure name:

   procedure_name;

SQL*PLUS: Execute a PL/SQL Block

- To execute a PL/SQL block (procedure, trigger etc.), its "End;" should be followed by either
  - a slash '/': execute/process without showing the content of the SQL buffer
  - **run**: first shows the content of the SQL buffer and then executes it.
- Note that the dot (.), if entered as first character on the line ends inputting lines to the SQL buffer, without executing its content
- "show errors": List all the errors of latest SQL invocation
  also: "show errors trigger <name of trigger>

Procedural Constructs: Exceptions

- Signaling of exception conditions, and declaring handlers for exceptions

  declare *out_of_stock* **condition**
  declare *exit* **handler for** *out_of_stock*
  begin
  …
  .. **signal** out_of_stock
  end

- The handler here is **exit** -- causes enclosing begin..end to be exited
- Other actions possible on exception

Oracle PL/SQL

- Based on ADA
- Assignments: direct (:=) and retrieval (INTO)
- Conditional Statements:

```
IF <condition>
THEN
        {statement;}
ELSE
        {<statement;>}
END IF;
```

```
IF <condition>
THEN
        {<statement;>}
ELSIF <condtion>
THEN
        {<statement;>}
ELSE
        {<statement;>}
END IF;
```


Oracle PL/SQL…

- Iterative Statements
  - Simple Loop

```
LOOP
    {<statement;>}
    EXIT; [or EXIT WHEN condition;]
END LOOP;
```

  - While Loop

```
WHILE <condition> LOOP {<statement;>} END LOOP;
```

  - For Loop

```
FOR counter IN val1..val2
    {<statement;}
END LOOP;
```


Oracle PL/SQL: General Structure

```
Declare                          -- optional
  x integer := 0;
  y student.sid%type;
  bad_data exception;
Begin                            -- mandatory
  select count(*) into x
  from STUDENT
  where major = 'CS' ;
  if x < 1 then RAISE bad_data;
  else dbms_output.put_line ("Number of CS Majors =" || x);
  end if;
Exception                        -- optional
   when bad_data then
      dbms_output.put_line ("troubles");
End;
```


Oracle PL/SQL: Var & Const

- **DECLARE**: introduces variables, constrants & records
- **Variables & Constants**

  <variable_name> datatype [NOT NULL := value];

  <constant_name> CONSTANT datatype := VALUE;
- Declaration of variables/constants based on a column from database table

  <variable_name> table_name.column_name%type;
- E.g., y student.sid%type;

Oracle PL/SQL: Records

- **Record type**

  TYPE <record_type_name> IS RECORD
  (<1st_col_name> datatype,
  <2nd_col_name> datatype, ...);

  - Declare fields based on a column from database table

    col_name table_name.column_name%type;

- **Record variable declaration**
  - User-defined: record_name record_type_name;
  - DB-based: record_name table_name%ROWTYPE;
  - E.g., student_rec Student%rowtype;

Cursors: Multiple Tuple Retrieval

- If more than one tuples can be selected, then tuples must be processed one at a time by means of a cursor
  - This is similar to the record-at-a-time processing
- A cursor is a "pointer" to a tuple in a result of a query
  - Current tuple w.r.t. a cursor is the tuple pointed by the cursor
- CURSOR <cursor_name> IS <query>
  - It declares a cursor by defining a query to be associated with a cursor with it
- OPEN <cursor_name> brings the query result from the DB and positions the cursor before the first tuple
- CLOSE <cursor_name> closes the named cursor and deletes the associated result table

PL/SQL Cursor Retrieval

- General Syntax

  FETCH <cursor-name> INTO <record_name>;

  FETCH <cursor-name> INTO <variable-list>;
  - copies into variables the current tuple and advances the cursor
- Explicit Cursor Attributes
  - <cursor_name>%FOUND - TRUE if tuple is returned
  - <cursor_name>%NOTFOUND - TRUE if no tuple is returned
  - <cursor_name>%ROWCOUNT - # tuple returned
  - <cursor_name>%ISOPEN - TRUE if cursor is opened

PL/SQL Cursor Retrieval Example

Student(SID,Name,Major,QPA)

```
DECLARE
  CURSOR st_cursor  IS
     SELECT SID, Name, Major, QPA
     FROM Student;
  student_rec Student%rowtype;
BEGIN
   IF NOT st_cursor%ISOPEN
     THEN OPEN st_cursor;
   END IF;
   LOOP
     FETCH st_cursor INTO student_rec;
     EXIT WHEN st_cursor%NOTFOUND;
     dbms_output.put_line(student_rec.SID || ' ' || student_rec.Name || ' ' || student_rec.QPA);
   END LOOP;
   close st_cursor;
END;
```

Loop Cursor

```
CREATE OR REPLACE PROCEDURE proc_confirm_cost
AS
   CURSOR reservation_cursor IS
      SELECT *
      FROM Reservation;
BEGIN
   -- Loop across all reservation numbers & prints them out
   FOR reservation_record IN reservation_cursor
   LOOP
     dbms_output.put_line (reservation_record.Reservation_Number);
   END LOOP;
END;
/
```


Implicit Cursors

- For each SQL statement, an *implicit cursor is* created to process it
- SQL is the default name of an implicit cursor
- SQL implicit cursor shares the basic attributes as the explicit ones:
  - SQL%FOUND - TRUE if tuple is returned
  - SQL%NOTFOUND - TRUE if no tuple is returned
  - SQL%ROWCOUNT - # tuple returned

PL/SQL Exception Handling

- Exceptions: *Named System, Unnamed System, User-defined*
- General Syntax

```
DECLARE
  <ex_name1> EXCEPTION;
  <ex_name2> EXCEPTION;
BEGIN
   RAISE <ex_name1>;
EXCEPTION
  WHEN <ex_name1> THEN <Error handling statements>
  WHEN <ex_name2> THEN <Error handling statements>
  WHEN Others THEN  <Error handling statements>
END;
```


List of Named Exception

CURSOR_ALREADY_OPEN	When you open a cursor that is already open.
INVALID_CURSOR	When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened.
NO_DATA_FOUND	When a SELECT...INTO clause does not return any row from a table.
TOO_MANY_ROWS	When you SELECT or fetch more than one row into a record or variable.
ZERO_DIVIDE	When you attempt to divide a number by zero.

- Example:

```
BEGIN
   Execution section
EXCEPTION
  WHEN NO_DATA_FOUND THEN  <Error handling statements>
END;
```

Naming the Unnamed Exceptions

- Unnamed exceptions correspond to ORA-error-#
- Two ways to handle unnamed *system* exceptions:
  - by using the WHEN OTHERS exception handler
  - by associating the exception code to a name
- An exception is named by using a **Pragma** called **EXCEPTION_INIT** within the DECLARE section:

```
<exception_name> EXCEPTION;
PRAGMA
  EXCEPTION_INIT (<exception_name>, Err_code);
```


Naming ORA-00001: Primary Key constraint

Student(SID,Name,Major,QPA)

```
DECLARE
  duplicatePK EXCEPTION;
  PRAGMA
    EXCEPTION_INIT (duplicatePK, -00001);
BEGIN
  INSERT INTO Student VALUES (199, ' PJ', CS, 3.95);
  dbms_output.put_line(student_rec.SID || ' ' || student_rec.Name || ' Inserted');
EXCEPTION
  WHEN duplicatePK  THEN
    dbms_output.put_line(student_rec.SID || ' already in Student');
END;
```


External Language Functions/Procedures

- Declaring external language procedures and functions
  - In C/C++

```
create procedure author_count_proc (in title varchar(20),
                                    out count integer)
language C
external name' /usr/db/bin/author_count_proc'
```

  - In Java

```
create function author_count ( title varchar(20) )
  returns integer
language Java
external name '/usr/db/bin/author_count.jar'
```

External Routines: Performance Vs. Security

- Benefits of external language functions/procedures:
  - more efficient for many operations, and
  - more expressive power
- Drawbacks
  - Code to implement function may need to be executed in the database system's address space
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data

  - Use *sandbox* techniques
    - that is use a safe language like Java
- Direct execution in the database system's space is used when efficiency is more important than security