Outline

- Embedded SQL
- Dynamic SQL
  - Many host languages: C, Cobol, Pascal, etc.
  - (we covered lightly)
- JDBC (API)
- Java
- Stored procedures:
  - PL/SQL (Oracle-specific), covered in Murach Chap. 13-15
  - Similar standard SQL/PSM, described in R&G sec. 6.5.3 (we won’t cover)

Why Stored Procedures?

- So far, all data processing is done at the client
  - Lots of data may have to be transferred
  - Functionality (code) replicated at each client
  - Lots of state (e.g., locks, transaction data) at the DBMS
    - While client processes the data
- Stored procedures execute in same process space as DBMS
  - Encapsulates application logic and is close to the data
  - Reuse of common functionality by different clients
- Vendors introduced their own procedural extensions
  - e.g., Oracle’s PL/SQL

SQL/PSM

- SQL Persistent Stored Modules
  - SQL standard for stored procedures, available in SQL:2003
  - Commercial vendors may offer own extensions of PSM
- Standard language for stored procedures
  - Supports both procedures and functions
  - Functions can return results through RETURN statement
  - Procedures can return results in parameters
- In this course we focus on Oracle PL/SQL
PL/SQL (Procedural Language SQL)

- Procedural extension to SQL developed by Oracle
- Most prominent DBMS procedural language
- Another language is T-SQL from Microsoft (MS SQL)
- Only DML is allowed in PL/SQL
- DDL such as creating or dropping tables NOT allowed, except by EXECUTE IMMEDIATE
- Basic program structure is a block
- There can be nested blocks
- PL/SQL syntax is not case sensitive (variable names as well)

PL/SQL Program (“Script”) Structure

DECLARE
variable_declarations
BEGIN
procedural_code
EXCEPTION
error_handling
END;

Assignments and Branches

- Assignment
  \( A := B + C \)
- Branch
  IF condition THEN statements;
  ELSEIF (condition) statements;
  ELSE statements;
  END IF;

- For other programming syntax (loops, etc.), see Murach, Chapter 13

Data Types

- It is possible to use ORACLE SQL types
  NUMBER, VARCHAR, etc
- PL/SQL allows directly referring to a column type
tablename.columnname%TYPE
e.g., SAILORS.SNAME%TYPE
- Also possible to define a row type (e.g., tuple)
tablename%ROWTYPE
- Can later refer to individual fields using column names
DBMS_OUTPUT.PUT_LINE("Name:" || sailor_rec.name ||
  "Age:" || sailor_rec.age);
  || means string concatenation (like + in Java)

Branch Example

DECLARE
A NUMBER(6) := 10;
B NUMBER(6);
BEGIN
  A := 23;
  B := A * 5;
  IF A < B THEN
    DBMS_OUTPUT.PUT_LINE("A is less than B");
  ELSE
    DBMS_OUTPUT.PUT_LINE("is less-or-equal than B");
  END IF;
END;

• Output is: 23 is less than 115
SQL Statements

• Data can be manipulated (DML) from PL/SQL

```plsql
DECLARE
  SID NUMBER(6);
BEGIN
  SID := 20;
  INSERT INTO Sailors (sid, name) VALUES (SID, 'Rusty');
  SID := SID + 1;
  INSERT INTO Sailors (sid, name) VALUES (SID, 'Yuppy');
END;
```

SQL Statements – retrieving data

• As before, there are two cases

1. Single-tuple result (the easy case)

```plsql
SELECT selectfields INTO declared_variables FROM table_list WHERE search_criteria;
DECLARE
  VAR_NAME Sailors.name%TYPE;
  VAR_AGE Sailors.age%TYPE;
BEGIN
  SELECT name, age INTO VAR_NAME, VAR_AGE FROM Sailors WHERE SID = 10;
  DBMS_OUTPUT.PUT_LINE('Age of ' || VAR_NAME || ' is ' || VAR_AGE);
END;
```

2. Multiple-tuples result: cursors are needed, one way:

```plsql
CURSOR cursorname IS SELECT_statement;
OPEN cursorname;
FETCH cursorname INTO variable_list;
-- FETCH loop
CLOSE cursorname;
```

Another way: use row-type loop variable.

A script retrieving one row, from Murach Ch13

```plsql
CONNECT ap/ap;
SET SERVEROUTPUT ON;
DECLARE
  sum_balance_due_var NUMBER(9, 2);
BEGIN
  SELECT SUM(invoice_total - payment_total - credit_total)
  INTO sum_balance_due_var
  FROM invoices
  WHERE vendor_id = 95;
  IF sum_balance_due_var > 0 THEN
    DBMS_OUTPUT.PUT_LINE('Balance due: $' || ROUND(sum_balance_due_var, 2));
  ELSE
    DBMS_OUTPUT.PUT_LINE('Balance paid in full');
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An error occurred');
END;
/```

The response from the system

Connected anonymous block completed
Balance due: $171.01

```sql
SQL> select invoice_total - payment_total - credit_total from invoices where vendor_id = 95;
```

<table>
<thead>
<tr>
<th>INVOICE_TOTAL</th>
<th>PAYMENT_TOTAL</th>
<th>CREDIT_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>46.21</td>
<td>39.77</td>
<td>19.67</td>
</tr>
<tr>
<td>32.7</td>
<td>16.33</td>
<td>16.33</td>
</tr>
</tbody>
</table>

6 rows selected.

Cursor Example (FETCH loop)

```plsql
DECLARE
  S Sailors%ROWTYPE;
CURSOR SAILORCURSOR IS
  SELECT * FROM Sailors;
BEGIN
  OPEN SAILORCURSOR;
  LOOP
    FETCH SAILORCURSOR INTO S;
    EXIT WHEN SAILORCURSOR%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Name: ' || S.sname || ', Age: ' || S.age);
  END LOOP;
  CLOSE SAILORCURSOR;
END;
```

Cursor Attributes

%NOTFOUND: Evaluates to TRUE when cursor has no more rows to read. FALSE otherwise
%FOUND: Evaluates to TRUE if last FETCH was successful and FALSE otherwise
%ROWCOUNT: Returns the number of rows that the cursor has already fetched from the database
%OPEN: Returns TRUE if this cursor is already open, and FALSE otherwise

Note: you have loaded this table group for hw3

Note: We don’t need this connect (just connect as usual to dbs3)
The syntax for declaring a cursor
CURSOR cursor_name IS select_statement;

The syntax for declaring a variable for a row
row_variable_name table_name%ROWTYPE;

The syntax for getting a column value
from a row variable
row_variable_name.column_name

The syntax for the loop itself:
FOR row_variable IN cursor_name LOOP
   …
END LOOP

Cursor loops can also use row-type variables

A script that uses a cursor, from Murach ch13
DECLARE
  CURSOR invoices_cursor IS
    SELECT invoice_id, invoice_total FROM invoices
    WHERE invoice_total - payment_total - credit_total > 0;
  invoice_row invoices%ROWTYPE;
BEGIN
  FOR invoice_row IN invoices_cursor LOOP
    IF (invoice_row.invoice_total > 1000) THEN
      UPDATE invoices
      SET credit_total = credit_total + (invoice_row.invoice_total * .1)
      WHERE invoice_id = invoice_row.invoice_id;
      DBMS_OUTPUT.PUT_LINE('1 row updated where invoice_id = ' || invoice_row.invoice_id);
    END IF;
  END LOOP;
END;
/

A script that uses a cursor, from Murach ch13
DECLARE
  CURSOR invoices_cursor IS
    SELECT invoice_id, invoice_total FROM invoices
    WHERE invoice_total - payment_total - credit_total > 0;
  invoice_row invoices%ROWTYPE;
BEGIN
  FOR invoice_row IN invoices_cursor LOOP
    IF (invoice_row.invoice_total > 1000) THEN
      UPDATE invoices
      SET credit_total = credit_total + (invoice_row.invoice_total * .1)
      WHERE invoice_id = invoice_row.invoice_id;
      DBMS_OUTPUT.PUT_LINE('1 row updated where invoice_id = ' || invoice_row.invoice_id);
    END IF;
  END LOOP;
END;
/

The response from the system
1 row updated where invoice_id = 3
1 row updated where invoice_id = 6
1 row updated where invoice_id = 8
1 row updated where invoice_id = 19
1 row updated where invoice_id = 34
1 row updated where invoice_id = 81
1 row updated where invoice_id = 88
1 row updated where invoice_id = 113

Declaring a Procedure
CREATE OR REPLACE
PROCEDURE procedure_name ( parameters ) IS
variable declarations
BEGIN
procedure_body
END;

• Parameters can be IN, OUT or INOUT, default is IN

CREATE OR REPLACE
PROCEDURE SUM_AB (A INT, B INT, C OUT INT) IS
BEGIN
C := A + B;
END;
/

Calling this procedure from a PL/SQL script:
SET SERVEROUTPUT;
DECLARE
SUM2 INT;  -- Don't use a var named SUM!!
BEGIN
SUM_AB(10, 20, SUM2);  -- Fill in SUM2
DBMS_OUTPUT.PUT_LINE('SUM2 = ' || SUM2);
END;
/

We are not required to code COMMIT and ROLLBACK in
the PL/SQL stored procedure. They can be done in the caller
Note: Homework 5 #5 defines a procedure that modifies
tables.

A stored procedure that updates a table
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
( invoice_number_param VARCHAR2,
  credit_total_param NUMBER ) AS
BEGIN
UPDATE invoices
SET credit_total = credit_total_param
WHERE invoice_number = invoice_number_param;
COMMIT;
EXCEPTION
WHEN OTHERS THEN
ROLLBACK;
END;
/

We are not required to code COMMIT and ROLLBACK in
the PL/SQL stored procedure. They can be done in the caller
Note: Homework 5 #5 defines a procedure that modifies
tables.

A stored procedure that updates a table
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
( invoice_number_param VARCHAR2,
  credit_total_param NUMBER ) AS
BEGIN
UPDATE invoices
SET credit_total = credit_total_param
WHERE invoice_number = invoice_number Param;
COMMIT;
EXCEPTION
WHEN OTHERS THEN
ROLLBACK;
END;
/

We are not required to code COMMIT and ROLLBACK in
the PL/SQL stored procedure. They can be done in the caller
Note: Homework 5 #5 defines a procedure that modifies
tables.
A statement that calls the stored procedure
CALL update_invoices_credit_total('367447', 300);

A script that calls the stored procedure
BEGIN
update_invoices_credit_total('367447', 300);
END;
/

A script that passes parameters by name
BEGIN
update_invoices_credit_total(
  credit_total_param=>300,
  invoice_number_param=>'367447');
END;
/

A script that calls the stored procedure (with no commit/rollback itself) and commits it or rolls it back
BEGIN
update_invoices_credit_total('367447', 300);
commit;
EXCEPTION
  when others
   rollback;
END;
/

The syntax for declaring parameters
parameter_name_1 [IN|OUT|IN OUT] data_type

A stored procedure that uses parameters
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
  (invoice_number Param IN VARCHAR2,
   credit_total_param IN NUMBER,
   update_count OUT INTEGER
  )
AS
BEGIN
  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    SELECT 0 INTO update_count FROM dual;
    ROLLBACK;
END;
/

The stored procedure (continued)
SELECT COUNT(*)
  INTO update_count
  FROM invoices
  WHERE invoice_number = invoice_numberParam;

Again we could move the commit/rollback and exception handling up to the caller.

Declaring a Function
CREATE OR REPLACE
FUNCTION function_name (function_params) RETURN return_type IS
  variable declarations
BEGIN
  function_body
  RETURN something_of_return_type;
END;
• Example
CREATE OR REPLACE
FUNCTION ADD_TWO (A INT, B INT) RETURN INT IS
BEGIN
  RETURN (A + B);
END;
Online (in caltest.sql in the plsql directory of cs630)

Calling the Function
• Example
CREATE OR REPLACE
FUNCTION ADD_TWO (A INT, B INT) RETURN INT IS
BEGIN
  RETURN (A + B);
END;
Calling this function from PL/SQL code:
SET SERVEROUTPUT ON;
DECLARE
  SUM2 INT;
BEGIN
  SUM2 := ADD_TWO(10, 20);
  DBMS_OUTPUT.PUT_LINE('SUM2: ' || SUM2);
END;
Note: homework 5 #2 defines a computational function (no table access)
Calling the function from SQL

- Example

```
CREATE OR REPLACE
FUNCTION ADD_TWO (A INT, B INT) RETURN INT IS
BEGIN
  RETURN (A + B);
END;
```

Example use in SQL:
```
Select * from Sailors where age = add_two(10, 20);
```

A function that returns a vendor ID

```
CREATE OR REPLACE FUNCTION get_vendor_id
( 
   vendor_name_param VARCHAR2
) RETURN NUMBER
AS
  vendor_id_var NUMBER;
BEGIN
  SELECT vendor_id
  INTO vendor_id_var
  FROM vendors
  WHERE vendor_name = vendor_name_param;
RETURN vendor_id_var;
END;
/
```

Note: Homework 5 #4 defines a function that accesses tables.

A SELECT statement that uses the function

```
SELECT invoice_number, invoice_total
FROM invoices
WHERE vendor_id = get_vendor_id('IBM')
```

The response from the system

<table>
<thead>
<tr>
<th>invoice_number</th>
<th>invoice_total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 234567891234</td>
<td>214</td>
</tr>
<tr>
<td>2 245678912345</td>
<td>324</td>
</tr>
</tbody>
</table>

The function called from a PL/SQL block (script or stored procedure):
```
vendor := get_vendor_id('IBM');
```

A function that calculates balance due

```
CREATE OR REPLACE FUNCTION get_balance_due
( 
   invoice_id_param NUMBER 
) RETURN NUMBER
AS
  balance_due_var NUMBER;
BEGIN
  SELECT invoice_total - payment_total - credit_total
  AS balance_due
  INTO balance_due_var
  FROM invoices
  WHERE invoice_id = invoice_id_param;
RETURN balance_due_var;
END;
/
```

A statement that calls the function

```
SELECT vendor_id, invoice_number,
get_balance_due(invoice_id) AS balance_due
FROM invoices
WHERE vendor_id = 37;
```

The response from the system

<table>
<thead>
<tr>
<th>vendor_id</th>
<th>invoice_number</th>
<th>balance_due</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>234567891234</td>
<td>214</td>
</tr>
<tr>
<td>37</td>
<td>245678912345</td>
<td>324</td>
</tr>
</tbody>
</table>

Exceptions

- Exceptions defined per block (similar to Java)
  - Each BEGIN...END has its own exception handling
  - If blocks are nested, exceptions are handled in an "inside to outside" fashion
  - If no block in the nesting handles the exception, a runtime error occurs
- There are multiple types of exceptions
  - Named system exceptions (most frequent) – we only cover these
  - Unnamed system exceptions
  - User-defined exceptions
- As in Java, we can let exceptions be thrown by the low-level procedures/functions, and handle them in top-level code, even crossing language boundaries (PL/SQL to Java, for example)
- Thus we don't need to handle exceptions in the functions and procedures if we are careful to handle them at top level
Handling Exceptions

DECLARE
    in here, call functions and/or procedures, or do other PL/SQL code
BEGIN
    EXCEPTION
        WHEN ex_name1 THEN
            error handling statements
        WHEN ex_name2 THEN
            error handling statements
        WHEN Others THEN
            error handling statements
    END;

See Murach, pg. 435 for a script doing an insert with handled exceptions online. Murach, pg. 439 for a script dropping a table and ignoring the expected exception when it's not there (as done in JdbcCheckup) online. Murach, pg. 475 for a script calling a stored procedure that throws, and catching that exception online.

Named System Exceptions

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Reason</th>
<th>Error Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURSOR_ALREADY_OPEN</td>
<td>When you open a cursor that is already open.</td>
<td>ORA-06511</td>
</tr>
<tr>
<td>INVALID_CURSOR</td>
<td>When you perform an invalid operation on a cursor like closing a cursor or fetching data from a cursor that is not opened.</td>
<td>ORA-01001</td>
</tr>
<tr>
<td>NO_DATA_FOUND</td>
<td>When a SELECT...INTO clause does not return any row from a table.</td>
<td>ORA-01403</td>
</tr>
<tr>
<td>TOO_MANY_ROWS</td>
<td>When you SELECT or fetch more than one row into a record or variable.</td>
<td>ORA-01422</td>
</tr>
<tr>
<td>ZERO_DIVIDE</td>
<td>When you attempt to divide a number by zero.</td>
<td>ORA-01476</td>
</tr>
</tbody>
</table>