

H9

# Advanced SQL

Fundamentals of Databases

Alvaro A A Fernandes, SCS, UoM

[COMP23111 Handout 09 of 12]

# Acknowledgements

- These slides are adaptations (mostly minor, but some major) of material authored and made available to instructors by **Thomas Connolly and Carolyn Begg** to accompany their textbook **Database Systems: A Practical Approach to Design, Implementation, and Management, 5th Edition. Addison-Wesley, 2010, 978-0-321-52306-8**
- Copyright remains with them and the publishers, whom I thank.
- Some slides had input from Sandra Sampaio, whom I thank.
- Some were inspired by material made available in <http://www.techonthenet.com/> and I thank them too.
- All errors are my responsibility.

# In Previous Handouts

- We learned about the relational algebra and SQL, both its DDL and DML capabilities and its querying constructs.
- We also learned how to design databases for implementation in a relational DBMS.
- We have also had some indication of additional semantics that cannot be directly captured by schema design (e.g., certain forms of integrity constraint enforcement).

# In This Handout

- We'll learn some advanced features of SQL that can be used to capture more application semantics in the DBMS environment.
- We'll explore features of SQL that make it Turing-complete and lead to the notion of stored procedures and stored functions in DBMSs.
- We'll also explore triggers, a kind of construct that enables a DBMS to respond to events such as a modification of the database instance.

# SQL Then and Now

- Initial versions of SQL were not computationally complete, i.e., they had no programming constructs.
- One approach to addressing this shortcoming is to allow programming languages to request the evaluation of SQL statements.
- This is referred to with the umbrella term of **Programmatic SQL**, i.e., strategies that make SQL act as a sublanguage within a host, Turing-complete, language.

# Programmatic SQL

6

- One approach to Programmatic SQL uses **embedding**.
- This means mixing SQL statements directly with source-code statements of some host language (e.g., C, or Java).
- In this approach, a special pre-compiler modifies the source code to replace SQL statements with calls to DBMS routines.
- The source code is then compiled and linked in the normal way.
- Another approach to Programmatic SQL uses a **standard API** that can be used in an application language to invoke SQL capabilities.
- Since it does not use embedding, it removes the need for any pre-compilation and arguably provides a cleaner interface and generates more manageable code.
- The best-known API is the Open Database Connectivity (ODBC) standard.
- There is a version for Java known as JDBC.
- We will not cover Programmatic SQL further in this course unit, but see the link below for an online chapter with more information.

# SQL Then and Now

- The main problem with embedding is **impedance mismatch**.
- Impedance mismatch is partly a consequence of mixing different programming paradigms:
  - ▶ SQL is a declarative language, whereas common high-level languages such as C or Java are procedural languages.
  - ▶ SQL handles sets of rows of data at a time, whereas common high level programming languages, such as C or Java, handle only one row of data at a time.
  - ▶ This means that, in the host language, one is constantly iterating over SQL results.
- Impedance mismatch also arises from the use of different type systems:
  - ▶ For example, SQL has **Date** and **Interval** types, whereas C does not.
  - ▶ SQL is not object-oriented, whereas Java is.
  - ▶ Some estimates suggest that 30% of programming effort in Programmatic SQL approaches goes into mapping from one data type system into the other.

# SQL Then and Now

- Another approach is to computational completeness is to extend SQL with programming language (procedural) constructs.
- This gives rise to what has become known as **SQL/PSM** (Persistent Stored Modules).
- Oracle's procedural extension to SQL is called **PL/SQL** (Procedural Language/SQL).
- PL/SQL is a limited programming language, compiled for speed.
- While SQL works on sets of rows, PL/SQL treats a table as a flat file accessed one row at a time.



- PL/SQL has concepts similar to modern programming languages, such as variable and constant declarations, control structures, exception handling, and modularization.
- PL/SQL is a block-structured language: blocks can be entirely separate or nested within one another.
- A PL/SQL program can be built with the following basic units: procedures, function and anonymous (unnamed) blocks.
- A PL/SQL block has up to three parts:
  - ▶ an optional declaration part, in which variables, constants, cursors, and exceptions are defined and possibly initialized;
  - ▶ a mandatory executable part, in which the variables are manipulated;
  - ▶ an optional exception part, to handle any exceptions raised during execution.

# PL/SQL: General Structure of a Block

10

```
[DECLARE      -- optional
    -- declarations
]

BEGIN          -- mandatory
    -- executable statements

[EXCEPTION    -- optional
    -- exception handlers
]

END;
```

# PS/SQL: Example Anonymous Block

11

```
DECLARE
    amount NUMBER := 0;
    v_error_code NUMBER;
    v_error_message VARCHAR2(255);

BEGIN
    SELECT COUNT(*) INTO amount
    FROM Student
    WHERE Student.class = 3;

    DBMS_OUTPUT.PUT_LINE(amount);

EXCEPTION
    WHEN OTHERS THEN ROLLBACK;
    v_error_code := SQLCODE
    v_error_message := SQLERRM
    INSERT INTO t_errors VALUES ( v_error_code, v_error_message);
END;
```

not assigned an  
identifying name

Print the number of  
students in a given  
class

to log caught exceptions

'OTHERS'  
matches any undeclared  
exceptions

see the  
documentation

# PL/SQL: Variable Declarations

12

```
variable_name  
[CONSTANT]  
datatype  
[NOT NULL]  
[:= | DEFAULT initial_value];
```

```
pi CONSTANT NUMBER := 3.141592653589;  
country VARCHAR(50) DEFAULT 'UK';
```

```
empId Employee.employeeId%TYPE NOT NULL;  
emp Employee%ROWTYPE;
```

- Variables and constant variables must be declared (and can be initialized) before they can be referenced.
- Type declarations can refer to schemas in the data dictionary:
  - ▶ **<var>%TYPE** declares <var> to have the same type as an attribute in the database schema
  - ▶ **<var>%ROWTYPE** declares <var> to have the same type as the rows of a relation in the database schema.

# PL/SQL: Forms of Assignment

13

- Variables can be assigned in two ways:
  - Using the assignment operator **:=**
  - Using the keyword **INTO** to capture the result of a **SELECT** or **FETCH** (on which more later) statement.

- For example:

```
empId := '8798725';
```

```
SELECT * INTO emp
```

```
FROM Employee
```

```
WHERE employeeId = empId;
```

```
FETCH empCursor INTO emp;
```

# PL/SQL: Control Statements - IF

14

```
IF      condition1 THEN {statement list}
ELSIF   condition2 THEN {statement list} -- optional
ELSE    {statement list} -- optional
END IF;
```

```
IF (position = "Manager")
    THEN salary := salary*1.05;
END IF;
```

```
IF (position = "Manager")
    THEN salary := salary*1.05;
ELSE  salary := salary*1.03;
END IF;
```

```
IF (position = "Manager")
    THEN salary := salary*1.05;
ELSIF (position = "Graduate")
    THEN salary := salary*1.03;
END IF;
```

```
IF (position = "Manager")
    THEN salary := salary*1.05;
ELSIF (position = "Graduate")
    THEN salary := salary*1.03;
ELSE  salary := salary*1.01;
END IF;
```

# PL/SQL: Control Statements - CASE

15

```
CASE [ expression ]
    {WHEN condition THEN result}
    .
    .
    .
    [ELSE result]
END CASE;
```

```
CASE
    WHEN a < b THEN 'hello'
    WHEN d < e THEN 'goodbye'
END CASE;
```

```
CASE
    WHEN owner='SYS' THEN 'system'
    WHEN owner='USR' THEN 'user'
END CASE;
```

```
CASE owner
    WHEN 'SYS' THEN 'system'
    WHEN 'USR' THEN 'user'
END CASE;
```

```
SELECT table_name,
    CASE
        WHEN owner='SYS' THEN 'Owner is system'
        WHEN owner='USR' THEN 'Owner is user'
        ELSE 'Owner is neither system nor user'
    END CASE;
FROM all_tables;
```

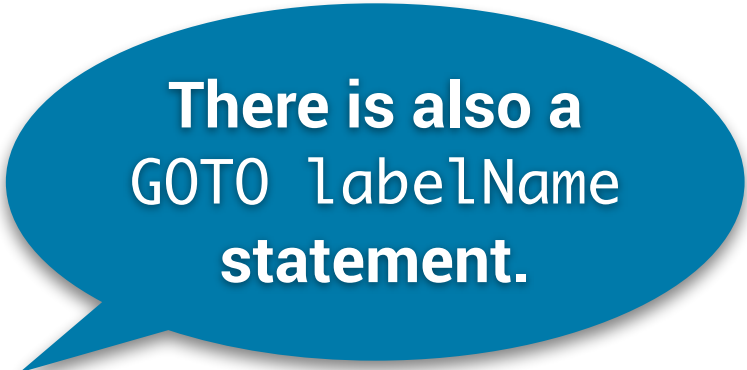


**embedding  
into SQL block**

# PL/SQL: Control Statements - LOOP

16

```
[labelName:]  
LOOP  
    {statement list}  
END LOOP [labelName];
```



**There is also a  
GOTO labelName  
statement.**

```
x := 1;  
myLoop:  
LOOP  
    x := x+1;  
    IF (x > 3) THEN  
        EXIT myLoop; -- exit immediately  
    END LOOP myLoop;  
-- execution resumes here
```



# PL/SQL: Control Statements - 'REPEAT'

17

```
[labelName:]  
LOOP  
    {statement list}  
    EXIT [labelName] [WHEN condition]  
END LOOP [labelName];
```

```
LOOP  
    monthly_value := daily_value * 31;  
    EXIT WHEN monthly_value > 4000;  
END LOOP;
```

# PL/SQL: Control Statements - WHILE

18

```
WHILE condition  
LOOP  
    {statement list}  
END LOOP;
```

```
WHILE monthly_value <= 4000;  
LOOP  
    monthly_value := daily_value * 31;  
END LOOP;
```

# PL/SQL: Control Statements - FOR

19

```
FOR variable IN [REVERSE] lowerBound .. upperBound
LOOP
    {statement list}
END LOOP;
```

```
FOR x IN 1..31
LOOP
    y := x * 10;
END LOOP;
```

```
FOR x IN REVERSE 1..31
LOOP
    y := x * 10;
END LOOP;
```

# PL/SQL: Exceptions

20

- An exception is denoted by PL/SQL identifier (either declared by the programmer or provided by the system)
- If raised during the execution of a block, it transfers the execution to the corresponding exception handler.
- Exception handlers in PL/SQL are separate routines that handle raised exceptions.
- User-defined exceptions are defined in the declarative part of a PL/SQL block.
- SQL/PSM also supports condition handling, i.e., responses to changes in the database state (see reading assignment).

# PL/SQL: Exception Declaration, Initialization and Handling

21

```
DECLARE
    vCount NUMBER;
    vProj Project.id%TYPE := '31415';

    -- No Type A project can have less than 100 employees allocated

    e_not_enough_employees EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_not_enough_employees, -1);

BEGIN
    SELECT COUNT(*) INTO vCount
    FROM    AllocatedTo E, Project P
    WHERE   P.id = vProj AND E.projId = P.id AND P.type = 'A';

    IF vCount < 100
    THEN RAISE e_not_enough_employees;
    END IF;

EXCEPTION
    WHEN e_not_enough_employees THEN
        DBMS_OUTPUT.PUT_LINE('Type A project' || vProj || 'has less than 100 employees');
END;
```

**declaring then  
initializing an exception**

**raising an exception**

**sending  
the output of PL/SQL to  
a client**

- Previous examples used hardwired initialization values so that the SELECT in the body returned a single result
- If a SQL query returns an arbitrary number of (i.e., zero, one or more) tuples, we must use a cursor.
- A cursor allows the rows of a query result to be accessed one at a time (i.e., it emulates a pointer to a row in the result, over which we iterate).
- A cursor
  - ▶ must be declared and opened before it is used;
  - ▶ must be closed to deactivate it after it is no longer required;
  - ▶ can be used to update rows (see reading assignment);
  - ▶ can receive passed-into parameters.
- We use a **FETCH** statement to retrieve each result row.

# PL/SQL: Example Cursor

23

- The example assumes a database of properties for rent.
- The relevant relation schema is

PropertyForRent

(pId, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)

- Each tuple records that a property with a given id is has an address (composed of street, city and postcode), is of a certain type (e.g., flat or house), has a number of rooms, is available for a given rent value per month, belongs to a given owner, is managed by a certain member of staff in a given branch of the company.

# PL/SQL: Example Cursor

24

**creating/updating a  
stored procedure**

**passing  
parameter into the  
procedure**

```
CREATE OR REPLACE PROCEDURE PropertiesForStaff (v_staffNo IN PropertyForRent.staffNo%TYPE)
AS
```

```
  v_pId      PropertyForRent.pId%TYPE;
  v_street   PropertyForRent.street%TYPE;
  v_city     PropertyForRent.city%TYPE;
  v_postcode PropertyForRent.postcode%TYPE;
```

**parameterizing the  
cursor**

```
  CURSOR propertyCursor (v_staffNo PropertyForRent.staffNo%TYPE) IS
    SELECT  pId, street, city, postcode
    FROM    PropertyForRent
    WHERE   staffNo = v_staffNo
    ORDER BY pID;
```

**anticipating error  
handling**

```
  v_error_code NUMBER;
  v_error_message VARCHAR2(255);
```

```
-- ... continues in the next slide ...
```



# PL/SQL: Example Cursor

25

```
-- ... continuing from the previous slide ...
BEGIN
-- open the cursor to execute the query
  OPEN propertyCursor(v_staffNo);
  DBMS_OUTPUT.PUT_LINE('Properties managed by staff' || v_staffNo);

-- now loop to fetch each row in the result table
-- until no further tuple is found
  LOOP
    FETCH propertyCursor INTO v_pId, v_street, v_city, v_postcode;
    EXIT WHEN propertyCursor%NOTFOUND;

-- display the data
    DBMS_OUTPUT.PUT_LINE('Property number:' || v_pId);
    DBMS_OUTPUT.PUT_LINE('Street:           ' || v_street);
    DBMS_OUTPUT.PUT_LINE('City:           ' || v_city);
    IF v_postcode IS NOT NULL
      THEN DBMS_OUTPUT.PUT_LINE('Postcode:           ' || v_postcode);
      ELSE DBMS_OUTPUT.PUT_LINE('Postcode:           NULL');
    END IF;
  END LOOP;
  IF propertyCursor%ISOPEN
    THEN CLOSE propertyCursor;
  END IF;
-- ... continues in the next slide ...
```

**notice how  
cursors have properties**

**see reading  
assignment**

**another cursor  
property here**

# PL/SQL: Example Cursor

26

```
-- ... continuing from the previous slide ...
```

```
-- catch any error
```

```
EXCEPTION
```

```
  WHEN OTHERS
```

```
  THEN
```

```
    v_error_code := SQLCODE;
```

```
    v_error_message := SUBSTR(SQLERRM, 1 , 255);
```

```
    DBMS_OUTPUT.PUT_LINE('The error code is ' || v_error_code || '- ' || v_error_message);
```

**Oracle functions for  
error handling**

```
  IF propertyCursor%ISOPEN
```

```
    THEN CLOSE propertyCursor;
```

```
  END IF;
```

```
END;
```

```
-- ... see the next slide an example invocation in SQL*Plus ...
```

# PL/SQL: Example Cursor

27

```
-- ... given the procedure in the previous slides ...  
--  
-- in SQL*Plus we set the client to receive server output  
-- then invoke the procedure  
--  
SQL> SET SERVEROUTPUT ON;  
SQL> EXECUTE PropertiesForStaff('SG14');
```

# PL/SQL: Subprograms, Functions, Procedures, Functions

28

- **Subprograms** are named PL/SQL blocks that can be invoked and take parameters.
  - ▶ Subprograms provide extensibility with modularity.
  - ▶ They aid abstraction, and promote reusability and maintainability.
  - ▶ They resemble functions and methods in high-level, general-purpose programming languages.
- There are two types of PL/SQL subprograms:
  - ▶ **Functions**
  - ▶ **(Stored) Procedures**
- Both can modify input and return output parameters.
- The difference between them is that a function must return a single value to the caller, whereas a procedure need not.
- A function must therefore use a **RETURN <parameter>;** statement to pass back the output value.

# PL/SQL: Parameter Passing

29

- Parameters have a name and a data type and are declared to be one of three kinds:
  - ▶ **IN**, which specifies that the parameter is used as an input only, i.e., for passing values into the subprogram
  - ▶ **OUT**, which specifies that the parameter is used as an output only, i.e., for passing values out of the subprogram
  - ▶ **IN OUT**, which allows the parameter to be used both as input and output

# PL/SQL: An Example Function

30

**declare and store**

```
CREATE OR REPLACE FUNCTION FindCourse
(cname IN VARCHAR2) RETURN NUMBER
AS
    cnumber NUMBER;
    CURSOR c IS
        SELECT course_number
        FROM   courses
        WHERE  course_name = cname;
BEGIN
    OPEN c;
    FETCH c INTO cnumber;
    IF c%NOTFOUND
        THEN cnumber := 9999;
    END IF;
    CLOSE c;
    RETURN cnumber;
END;
```

**invoke**

```
SELECT course_name, FindCourse(course_name) AS course_id
FROM   courses
WHERE  subject = 'Mathematics';
```

# PL/SQL: Packages

31

- A **package** associates a name to a collection of subprograms, grouping and storing them together (as in a module)
- Packages provide an encapsulation mechanism, i.e., they can be used control what is publicly visible and what isn't.
- When a package is created, Oracle compiles it, stores the compile code in memory and stores it in the database.

# PL/SQL: Example Package Declaration/Use

32

```
CREATE OR REPLACE PACKAGE StaffPackage
AS
    PROCEDURE PropertiesForStaff (v_staffNo PropertyForRent.staffNo%TYPE);
    -- others would come in here
END PropertiesForStaffPackage

-- an example invocation
StaffPackage.PropertiesForStaff('SG14')
```



# Triggers: What and What For

33

- A **trigger** defines an **action** that the database should take when some **event** occurs in the application.
- A trigger can be programmed to execute on the event of a table being modified.
- If the event associated with a trigger takes place, we say that the trigger **fires**.
- Triggers are useful for:
  - ▶ (re)computing the value of derived attributes (e.g., amount of income tax due)
  - ▶ enforcing value-dependent integrity constraints (e.g., overseas customers must provide a country)
  - ▶ audit changes in data (e.g., monitor lower bounds on stock levels)

# Triggers: Informal Semantics

34

- Triggers are a reactive mechanism, whose semantics obeys an **event-condition-action (ECA) model**.
- The events on a specific table or view that, after detection, cause a trigger to fire are:
  - ▶ insertion,
  - ▶ deletion or
  - ▶ update
- Oracle also supports triggers on the following events on schema objects:
  - ▶ creation
  - ▶ alteration or
  - ▶ dropping
- Oracle further supports the firing of triggers on error messages
- A trigger can be specified to fire before, after or instead of the processing of the associated event.
- The optional condition determines whether the action of a fired trigger is executed in fact.
- The action is a block that is executed if the rule fires and the condition evaluates to true.

- There are two types of triggers:
  - ▶ **row-level** triggers execute the action for each row of the table that is affected by the triggering event
  - ▶ **statement-level** triggers execute only once per triggering event even if multiple rows are affected.

# Triggers: Cascading

- The action of a trigger can cause events that, if they have triggers on them, may themselves fire too.
- A trigger  $T$  fires due to event  $E$  and takes action  $A$  which raises an event  $E'$  on which a trigger  $T'$  takes action  $A'$ .
- This is **cascading**, and cascading can cause non-termination.
- If  $A$  always raises  $E'$  and  $A'$  always raises  $E$ , an infinite loop could occur.

# Trigger: A Simplified Template

37

```
CREATE TRIGGER triggerName
  (BEFORE | AFTER | INSTEAD OF)
  (INSERT | DELETE | UPDATE [OF triggerColumnList])
ON tableName
  [REFERENCING (OLD | NEW) AS (oldName | newName)]
  [FOR EACH (ROW | STATEMENT)]
  [WHEN condition]
  triggerAction
;
```

# Trigger: An Example AFTER Row-Level Trigger

38

```
CREATE TRIGGER StaffAfterInsert
  AFTER INSERT
ON Staff
  REFERENCING NEW AS new
  FOR EACH ROW
BEGIN
  INSERT INTO StaffAudit
  VALUES (:new.staffNo, :new.fName, :new.lName, :new.Position,
          :new.sex, :new.DOB, :new.salary, :new.branchNo
          )
END;
```


**In a trigger, you can refer to the old and the new values**

# Trigger: An Example BEFORE Row-Level Trigger

39

```
CREATE TRIGGER StaffLimit
  BEFORE INSERT
ON Staff
  REFERENCING NEW AS new
  FOR EACH ROW
DECLARE
  v_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO v_count
  FROM   PropertyForRent
  WHERE  staffNo = :new.staffNo;

  IF v_count = 100 THEN
    RAISE_APPLICATION_ERROR(-20000, 'Staff ' || :new.staffNo || ' is fully loaded.');
```



**note the  
application-specific  
message**

```
  END IF;
END;
```

# Triggers: Advantages

40

- *Elimination of redundant code:* the same trigger serves many client applications that would otherwise have to implement it
- *Simplification of modifications:* code/change once (and test/evaluate well) leads to many applications benefitting
- *Increased security:* triggers, unlike applications, are as protected as data items are
- *Improved integrity:* more business rules can be enforced consistently, efficiently and securely
- *Improved processing power:* triggers run on well-provisioned servers and therefore can be very efficient
- *Good fit with the client-server architecture:* a client can keep things simple whilst relying on the server to take on the complex processing of the events the client cause.



# Triggers: Disadvantages

41

- *Performance overhead*: when an event is detected, the DBMS will have to spend processing time identifying which triggers the event has fired, then process the trigger and the event (or vice-versa), which is an overhead to subtract from the potential performance benefits
- *Cascading effects*: triggers can generate complex event chains that could have effects that are difficult to anticipate (e.g., they may lead to non-termination).
- *Impossibility of scheduling*: because they are reactive, triggers cannot be scheduled and therefore make it harder to optimize performance
- *Lack of good portability*: though implemented in most DBMS, they tend to be implemented slightly differently in different system, which causes portability problems.

# In The Next Handout

- We'll explore transactions, concurrency and recovery management in DBMSs.