Advanced Management of Data

Extensions of SQL

Extensions of SQL

Extensions of SQL to overcome some (historical) limitations of RDBMS:

- SQL as a programming language
  - control statements
  - exceptions
  - cursors
  - stored procedures
- Recursive Queries
- Triggers

The SQL Programming Language(s)

Initial versions of SQL have been computationally incomplete (without programming constructs).

Later versions of SQL could be embedded in a high-level programming language, but produced an impedance mismatch, because of the mixing of different programming paradigms.

SQL/PSM

- SQL has been extended to a full programming language
- the extensions are known as SQL/PSM (Persistent Stored Modules)

PL/SQL

> PL/SQL (Procedural Language SQL) is the Oracle version of an SQL programming language and has concepts similar to modern programming languages, such as variable and constant declarations, control structures, exception handling, and modularization.

Following we will present PL/SQL but also comment on implementation aspects in SQL/PSM:

PL/SQL - Blocks

PL/SQL is a block-structured language: blocks can be entirely separate or nested within one another.

The basic units that constitute a PL/SQL program are procedures, functions, and anonymous (unnamed) blocks.

```
DECLARE
<declarations>

BEGIN
<executable statements>

EXCEPTION
<exception handlers>

END;
```

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 - Declarations

Variables and constant variables must be declared before they can be referenced in other statements, e.g.

```
vStaffNo VARCHAR2(5);
vRent NUMBER(6,2) NOT NULL := 600;
MAX_PROPERTIES CONSTANT NUMBER := 100;
```

It is also possible to declare a variable to be of the same type as a column in a specified table or another variable using the `%TYPE` attribute:

```
vStaffNo Staff.staffNo%TYPE;
vStaffNo1 vStaffNo%TYPE;
```

Similarly, we can declare a variable to be of the same type as an entire row of a table or view using the `%ROWTYPE` attribute.

```
vStaffRec Staff%ROWTYPE;
```

SQL/PSM:

`%TYPE` and `%ROWTYPE` are not standard SQL!

PL/SQL - Assignments

In the executable part of a PL/SQL block, variables can be assigned in two ways:

- using the normal assignment statement (`:=`)
- as the result of an `SELECT` or `FETCH` statement

Example

```
vStaffNo := 'SG14';
vRent := 500;
SELECT COUNT(*) INTO x
FROM PropertyForRent
WHERE staffNo = vStaffNo;
```

SQL / PSM:

Assignments use the `SET` keyword at the start of the line with the = symbol, instead of the `:=`, e.g. `SET vStaffNo = 'SG14'`

PL/SQL - Control Statements

Conditional `IF` statement

```
IF (condition) THEN
    <SQL statement list>
[ELSIF (condition) THEN
    <SQL statement list>]
[ELSE <SQL statement list>]
END IF;
```

Example

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

SQL/PSM:

The SQL standard specifies `ELSEIF` instead of `ELSIF`.

PL/SQL - Control Statements

Conditional CASE statement

The CASE statement allows the selection of an execution path based on a set of alternatives:

```
CASE (operand)
[WHEN (whenOperandList) | WHEN (searchCondition)
   THEN <SQL statement list>]
[ELSE <SQL statement list>]
END CASE;
```

Example

```
CASE lowercase(x)
    WHEN 'a' THEN x := 1;
    WHEN 'b' THEN x := 2;
                  y := 0;
    WHEN 'c' THEN x := 3;
END CASE;
```

```
UPDATE Staff
SET salary = CASE
       WHEN position = 'Manager'
         THEN salary * 1.05
       ELSE
         THEN salary * 1.02
       END;
```

PL/SQL - Control Statements

Iteration statement (LOOP)

```
[labelName:]
LOOP
    <SQL statement list>
    EXIT [labelName] [WHEN (condition)]
END LOOP [labelName];
```

Example

```
x := 1;
myLoop:
LOOP
  x := x + 1;
  IF (x > 3) THEN
    EXIT myLoop;
END LOOP myLoop;
y := 2;
```

SQL/PSM:

The SQL standard specifies `LEAVE` instead of `EXIT WHEN (condition)`.

PL/SQL - Control Statements

FOR loop in PL/SQL

```
FOR indexVariable
IN lowerBound .. upperBound LOOP
<SQL Statement list>
END LOOP;
```

Example

```
numberOfStaff NUMBER;
SELECT COUNT(*) INTO numberOfStaff
FROM PropertyForRent
WHERE staffNo = 'SG14';

FOR iStaff IN 1 .. numberOfStaff LOOP
…
END LOOP;
```

FOR loop in Standard SQL

```
FOR indexVariable
AS querySpecification DO
<SQL statement list>
END FOR;
```

Example

```
FOR iStaff AS SELECT COUNT(*)
FROM PropertyForRent
WHERE staffNo = 'SG14' DO
…
END FOR;
```

PL/SQL - Control Statements

Iteration Statement (WHILE)

- in PL/SQL

```
WHILE (condition) LOOP
  <SQL statement list>
END LOOP [labelName];
```

- in SQL/PSM

```
WHILE (condition) DO
  <SQL statement list>
END WHILE [labelName];
```

PL/SQL - Exceptions

An exception is an identifier in PL/SQL raised during the execution of a block that terminates its main body of actions, although some final actions can be performed.

An exception can be raised automatically or explicitly using the **`RAISE`** statement.

To handle raised exceptions, separate routines called exception handlers are specified.

A user-defined exception is defined in the declarative part of a PL/SQL block

In the executable part, a check is made for the exception condition, and, if found, the exception is raised.

The exception handler itself is defined at the end of the PL/SQL block.

Example (1)

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

```
DECLARE
   vpCount       NUMBER;
   vStaffNo PropertyForRent.staffNo%TYPE := 'SG14';
-- define an exception for the enterprise constraint that prevents a member of staff
-- managing more than 100 properties
   e_too_many_properties EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_too_many_properties, -20000);
```

[Connolly & Begg]

Example (2)

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

```
BEGIN
        SELECT COUNT(*) INTO vpCount
        FROM PropertyForRent
        WHERE staffNo = vStaffNo;
        IF vpCount  = 100
-- raise an exception for the general constraint
            RAISE e_too_many_properties;
        END IF;
        UPDATE PropertyForRent SET staffNo = vStaffNo WHERE propertyNo = 'PG4';
EXCEPTION
   -- handle the exception for the general constraint
   WHEN e_too_many_properties THEN
            dbms_output.put_line('Member of staff ' || staffNo || 'already managing 100 properties');
END;
```

[Connolly & Begg]

PL/SQL - Cursors

A **`SELECT`** statement can be used if the query returns exactly one row.

To handle a query that can return an arbitrary number of rows, PL/SQL uses cursors to allow the rows of a query result to be accessed one at a time.

The cursor can be advanced by one to access the next row.

A cursor must be

- declared and opened before it can be used
- closed to deactivate it after it is no longer required

Once the cursor has been opened, the rows of the query result can be retrieved one at a time using a **`FETCH`** statement.

Cursors - Example (1)

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

Print the properties managed by staff member ‚SG14'

```
DECLARE
   vPropertyNo   PropertyForRent.propertyNo%TYPE;
   vStreet       PropertyForRent.street%TYPE;
   vCity         PropertyForRent.city%TYPE;
   vPostcode     PropertyForRent.postcode%TYPE;

   CURSOR propertyCursor IS
     SELECT propertyNo, street, city, postcode
     FROM PropertyForRent
     WHERE staffNo = 'SG14'
     ORDER BY propertyNo;
```

[Connolly & Begg]

Cursors - Example (2)

```
BEGIN
  OPEN propertyCursor;
  LOOP
    FETCH propertyCursor INTO vPropertyNo,vStreet,vCity,vPostcode;
    EXIT WHEN propertyCursor%NOTFOUND;

    dbms_output.put_line('Property number' || vPropertyNo);
    dbms_output.put_line('Street         ' || vStreet);
    dbms_output.put_line('City           ' || vCity);
    IF vPostcode IS NOT NULL THEN
      dbms_output.put_line('Post Code      ' || vPostcode);
    ELSE
      dbms_output.put_line('Post Code      NULL');
    END IF;
  END LOOP;
  IF propertyCursor%ISOPEN THEN CLOSE propertyCursor END IF;
END;
```

PL/SQL - Cursors

Useful Cursor attributes

`%NOTFOUND` evaluates to true if the most recent fetch does not return a row

`%FOUND` evaluates to true if the most recent fetch returns a row

`%ISOPEN` evaluates to true if the cursor is open

`%ROWCOUNT` evaluates to the total number of rows returned so far

PL/SQL - Cursors

Passing parameters to cursors

PL/SQL allows cursors to be parameterized, so that the same cursor definition can be reused with different criteria.

Example

```
CURSOR propertyCursor (vStaffNo VARCHAR2) IS
  SELECT propertyNo, street, city, postcode
  FROM PropertyForRent
  WHERE staffNo = vStaffNo
  ORDER BY propertyNo;
```

and we could open the cursor using the following example statements:

```
vStaffNo1 PropertyForRent.staffNo%TYPE := 'SG14';
OPEN propertyCursor('SG14');
OPEN propertyCursor('SA9');
OPEN propertyCursor(vStaffNo1);
```

PL/SQL - Subprograms

Subprograms are named PL/SQL blocks that can take parameters and be invoked.

PL/SQL has two types of subprogram called (stored) procedures and functions.

Both can modify and return data passed to them as a parameter, but a function can only return a single value to the caller.

Parameters

A parameter has a specified name and data type, and can also be designated as:

- `IN` parameter is used as an input value only
- `OUT` parameter is used as an output value only
- `IN OUT` parameter is used as both an input and an output value

Example

```
CREATE OR REPLACE PROCEDURE PropertiesForStaff (IN vStaffNo VARCHAR2) AS
   DECLARE
     vPropertyNo  PropertyForRent.propertyNo%TYPE;
     vStreet      PropertyForRent.street%TYPE;
     vCity        PropertyForRent.city%TYPE;
     vPostcode    PropertyForRent.postcode%TYPE;
     CURSOR propertyCursor (vStaffNo VARCHAR2) IS
       SELECT propertyNo, street, city, postcode
       FROM PropertyForRent
       WHERE staffNo = vStaffNo
       ORDER BY propertyNo;
   …
```

The procedure could then be executed as

```
SQL> SET SERVEROUTPUT ON;
SQL> EXECUTE PropertiesForStaff('SG14');
```

PL/SQL - Packages

A package is a collection of procedures, functions, variables, and SQL statements that are grouped together and stored as a single program unit.

A package has two parts: a specification and a body.

Specification

- declares all public constructs of the package

Body

- defines all constructs (public and private) of the package
- implements the specification

In this way, packages provide a form of encapsulation.

Example

For the previous example, we could create a package specification as follows:

```
CREATE OR REPLACE PACKAGE StaffPropertiesPackage AS
  procedure PropertiesForStaff(vStaffNo VARCHAR2);
END StaffPropertiesPackage;
```

and we could create the package body (that is, the implementation of the package) as:

```
CREATE OR REPLACE PACKAGE BODY StaffPropertiesPackage AS
  …
END StaffPropertiesPackage;
```

To reference the items declared within a package specification, we use the dot notation. For example, we could call the *PropertiesForStaff* procedure as follows:

```
StaffPropertiesPackage.PropertiesForStaff('SG14');
```

Extensions of SQL

Extensions of SQL to overcome some limitations of RDBMS:

- SQL as a programming language
  - control statements
  - exceptions
  - cursors
  - stored Procedures
- Recursive Queries
- Triggers

Recursion

How to find all the managers who directly or indirectly manage staff member S005?

Recursive / Transitive Closure

An extension to relational algebra that has been proposed to handle this type of query is the unary transitive closure, or recursive closure, operation.

The transitive closure of a relation R with attributes (A_1, A_2) defined on the same domain is the relation R augmented with all tuples successively deduced by transitivity; that is, if (a,b) and (b,c) are tuples of R, the tuple (a,c) is also added to the result.

This operation cannot be performed with just a fixed number of relational algebra operations, but requires a loop along with the Join, Projection, and Union operations.

staffNo	managerstaffNo
S005	S004
S004	S003
S003	S002
S002	S001
S001	NULL

staffNo	managerstaffNo
S005	S004
S004	S003
S003	S002
S002	S001
S001	NULL
S005	S003
S005	S002
S005	S001
S004	S002
S004	S001
S003	S001

Recursion

```
WITH RECURSIVE
AllManagers (staffNo, managerStaffNo) AS
   (SELECT staffNo, managerStaffNo
    FROM Staff
    UNION
    SELECT in.staffNo, out.managerStaffNo
    FROM AllManagers in, Staff out
    WHERE in.managerStaffNo = out.staffNo);
SELECT * FROM AllManagers;
```

Staff

staffNo	managerstaffNo
S005	S004
S004	S003
S003	S002
S002	S001
S001	NULL

AllManagers

staffNo	managerstaffNo
S005	S004
S004	S003
S003	S002
S002	S001
S001	NULL
S005	S003
S005	S002
S005	S001
S004	S002
S004	S001
S003	S001

Recursion

Example

Calculating Factorial of 4:

```
WITH RECURSIVE factorial(F,n) AS (
    SELECT 1 F, 4 n
UNION ALL
    SELECT F*n F, n-1 n from factorial where n>1
)
SELECT F from factorial where n=1
```

➔ 24

factorial	
F	**n**
1	4
4	3
12	2
24	1

Extensions of SQL

Extensions of SQL to overcome some limitations of RDBMS:

- SQL as a programming language
  - control statements
  - exceptions
  - cursors
  - stored Procedures
- Recursive Queries
- Triggers

Triggers

A trigger is an SQL statement that is executed automatically by the DBMS as a side effect of a modification - the triggering event - to a named table.

The act of executing a trigger is sometimes known as firing the trigger.

Some Use Cases

- validating input data and maintaining complex integrity constraints
- supporting alerts that action needs to be taken when a table is updated in some way
- refreshing derived attributes after an update operation
- maintaining audit information, by recording the changes made, and by whom
- supporting replication

Triggers

Event-Condition-Action (ECA) model

A rule in the ECA model has three components:

- The event(s) that triggers the rule: These events are usually database update operations that are explicitly applied to the database.
- The condition that determines whether the rule action should be executed: Once the triggering event has occurred, an optional condition may be evaluated. If no condition is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only if it evaluates to true the rule action will be executed.
- The action to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

Triggers

The basic format of the `CREATE TRIGGER` statement is as follows:

```
CREATE TRIGGER TriggerName
BEFORE | AFTER | INSTEAD OF <triggerEvent> ON <TableName>
[REFERENCING <oldOrNewValuesAliasList>]
[FOR EACH {ROW | STATEMENT}]
[WHEN (triggerCondition)]
<triggerBody>
```

- Triggering events include insertion, deletion, and update of rows in a table. In the latter case only, a triggering event can also be set to cover specific named columns of a table.
- A trigger has an associated timing of either `BEFORE`, `AFTER`, or `INSTEAD OF`.
  - A `BEFORE` trigger is fired before the associated event occurs
  - an `AFTER` trigger is fired after the associated event occurs
  - an `INSTEAD OF` trigger is fired in place of the trigger event

Triggers

- To enable the comparison of values before an update with the corresponding values after an update the <oldOrNewValuesAliasList> can refer to:
  - an old or new row (OLD/NEW or OLD ROW/NEW ROW), in the case of a row-level trigger
  - an old or new table (OLD TABLE/NEW TABLE), in the case of an *statement-level* trigger

Triggers

Row-Level Triggers (`FOR EACH ROW`)

- execute for each row of the table that is affected by the triggering event

Statement-Level Triggers (`FOR EACH STATEMENT`)

- execute only once even if multiple rows are affected by the triggering event.

Cascading Triggers

Triggers can also activate themselves one after the other.

This can happen when the trigger action makes a change to the database that has the effect of causing another event that has a trigger associated with it to fire.

Triggers

Order of Firing

As more than one trigger can be defined on a table, the order of firing of triggers is important.

Triggers are fired as the trigger event (`INSERT`, `UPDATE`, `DELETE`) is executed. The following order is observed:

1. Execution of any `BEFORE` statement-level trigger on the table.
2. For each row affected by the statement:
    a) execution of any `BEFORE` row-level trigger
    b) execution of the statement itself
    c) application of any referential constraints
    d) execution of any `AFTER` row-level trigger
3. Execution of any `AFTER` statement-level trigger on the table.

Example

EMPLOYEE

Name	Ssn	Salary	Dno	Supervisor_ssn

DEPARTMENT

Dname	Dno	Total_sal	Manager_ssn

Scenario

- two relations EMPLOYEE and DEPARTMENT describe a simplified Company scenario
- each employee has a name (*Name*), Social Security number (*Ssn*), salary (*Salary*), department to which the employee is currently assigned (*Dno*, a foreign key to DEPARTMENT), and a direct supervisor (*Supervisor_ssn*, a (recursive) foreign key to EMPLOYEE).
- NULL is allowed for Dno, indicating that an employee may be temporarily unassigned to any department.
- Each department has a name (Dname), number (Dno), the total salary of all employees assigned to the department (Total_sal), and a manager (Manager_ssn, which is a foreign key to EMPLOYEE).
- Notice that the Total_sal attribute is really a derived attribute whose value should be the sum of the salaries of all employees who are assigned to the particular department.

Example

EMPLOYEE

Name	Ssn	Salary	Dno	Supervisor_ssn

DEPARTMENT

Dname	Dno	Total_sal	Manager_ssn

Maintaining the correct value of a derived attribute can be done via triggers.

First we have to determine the events that may cause a change in the value of *Total_sal*, which are as follows:

1. Inserting (one or more) new employee tuples
2. Changing the salary of (one or more) existing employees

```
R1:
CREATE TRIGGER Total_sal1
AFTER INSERT ON EMPLOYEE
FOR EACH ROW
WHEN ( NEW.Dno IS NOT NULL )
  UPDATE DEPARTMENT
  SET Total_sal = Total_sal
                + NEW.Salary
  WHERE Dno = NEW.Dno;
```

```
R2:
CREATE TRIGGER Total_sal2
AFTER UPDATE OF Salary ON EMPLOYEE
FOR EACH ROW
WHEN ( NEW.Dno IS NOT NULL )
  UPDATE DEPARTMENT
  SET Total_sal = Total_sal
          + NEW.Salary – OLD.Salary
  WHERE Dno = NEW.Dno;
```

Example

EMPLOYEE

Name	Ssn	Salary	Dno	Supervisor_ssn

DEPARTMENT

Dname	Dno	Total_sal	Manager_ssn

Maintaining the correct value of a derived attribute can be done via triggers.

First we have to determine the **events** that *may cause* a change in the value of *Total_sal*, which are as follows:

3. Changing the assignment of existing employees from one department to another
4. Deleting (one or more) employee tuples

```
R3:
CREATE TRIGGER Total_sal3
AFTER UPDATE OF Dno ON EMPLOYEE
FOR EACH ROW
BEGIN
  UPDATE DEPARTMENT
  SET Total_sal = Total_sal + NEW.Salary
  WHERE Dno = NEW.Dno;
  UPDATE DEPARTMENT
  SET Total_sal = Total_sal – OLD.Salary
  WHERE Dno = OLD.Dno;
END;
```

```
R4:
CREATE TRIGGER Total_sal4
AFTER DELETE ON EMPLOYEE
FOR EACH ROW
WHEN ( OLD.Dno IS NOT NULL)
  UPDATE DEPARTMENT
  SET Total_sal = Total_sal
                - OLD.Salary
  WHERE Dno = OLD.Dno;
```

Triggers

Difficulties in designing triggers

There are no easy-to-use techniques for designing, writing, and verifying rules.

How can we verify that a set of rules is consistent, meaning that two or more rules in the set do not contradict one another?

How can we guarantee termination of a set of rules under all circumstances?

Example (termination problem)

Rule 1:

```
CREATE TRIGGER T1
AFTER INSERT ON TABLE1
FOR EACH ROW
UPDATE TABLE2
SET Attribute1 = ... ;
```

Rule 2:

```
CREATE TRIGGER T2
AFTER UPDATE OF Attribute1 ON TABLE2
FOR EACH ROW
INSERT INTO TABLE1
VALUES ( ... );
```

Triggers

Advantages

- Elimination of redundant code
- Simplifying modifications
- Increased security
- Improved integrity
- Improved processing power
- Good fit with the client-server architecture

Triggers

Disadvantages

- Performance Overhead
- Complexity
- Hidden functionality
- Cascading effects
- Cannot be scheduled
- Less portable