Pragma Exception_Init():

=> Some errors have names. Some errors does not have names.
=> To define name for unnamed exception we use pragma exception_init().

Syntax:

```
pragma exception_init(<error_name>,
<error_code>)
```

Example:

```
check_violate EXCEPTION;
pragma exception_init(check_violate, -2290);
```

-1476 => error code
divisor is equal to zero => Error Message
zero_divide => Exception name

-1 => Error Code
unique constraint violated => Error Message
dup_val_on_index => Exception Name

-2290 => Error Code
check constraint violated => Error Message
no error name

-1400 => Erorr Code
cannot insert NULL => Error Message
no error name

What is pragma?

=> it is a compiler directive.
=> pragma exception_init()   => compiler directive.
=> directive => command / instruction.
=> if any line started with pragma, that is instruction to compiler.
=> it instructs before compiling program, first execute this line.

Example on pragma exception_init():

```
CREATE TABLE student
(
sid NUMBER(4) CONSTRAINT c1 PRIMARY KEY,
sname VARCHAR2(10),
m1 NUMBER(3) CONSTRAINT c2 CHECK(m1 BETWEEN 0 AND 100)
);
```

Program to insert student record into student table.
sid is PK. if user is inserting dup val in PK RTE occurs. Handle it.
if check constraint violated RTE occurs. Handle it:

```
DECLARE
   check_violate EXCEPTION;
   pragma exception_init(check_violate, -2290);
BEGIN
   INSERT INTO student VALUES(&sid, '&sname', &m1);
   COMMIT;
   dbms_output.put_line('record inserted');

   EXCEPTION
```

```
     WHEN dup_val_on_index THEN
       dbms_output.put_line('sid already assigned');
     WHEN check_violate THEN
        dbms_output.put_line('marks must be b/w 0 to 100');
END;
/
```

NOTE:

SQLERRM => is a built-in function. it returns error message.

SQLCODE => is a built-in function. it returns error code.

Example:

```
  DECLARE
     x NUMBER(4);
     y NUMBER(4);
     z NUMBER(4);
  BEGIN
     x := &x;
     y := &y;

     z := x/y;

     dbms_output.put_line('z=' || z);

   EXCEPTION
     WHEN others THEN
        dbms_output.put_line(SQLERRM);
  END;
  /
```

TRIGGERS

Monday, December 2, 2024 11:21 AM

hike(7369, 2000) —calls→ PROCEDURE hike(... , ...)
--stmts

DML / DDL —calls→ TRIGGER t1
--stmts

=> TRIGGER is a named block of statements that gets executed automatically when we submit DML or DDL command.

=> TRIGGER is same as PROCEDURE.
For PROCEDURE execution explicit call is required.
For TRIGGER execution explicit call is not required.
When we submit DML or DDL command imlictly trigger will be called.

NOTE:
to perform DMLs, define PROCEDURE.
to control DMLs, define TRIGGER.

Trigger can be used for following purposes:

1. to control DMLs.
examples:
if DML performed on SUNDAY raise error.
if not office timings, raise error.
office timings: 10AM to 4PM

2. to audit the tables or databases.
examples:
which user
on which date
at which time
which operations
all above things can be recorded in a table

3. to implement our own constraints [rules].
Example:
min sal 5000 => sal<5000 raise error

Types of Triggers:

3 Types:
1. Table Level Trigger / DML Trigger [SQL DEVELOPER]
a. Statement Level Trigger
b. Row Level Trigger

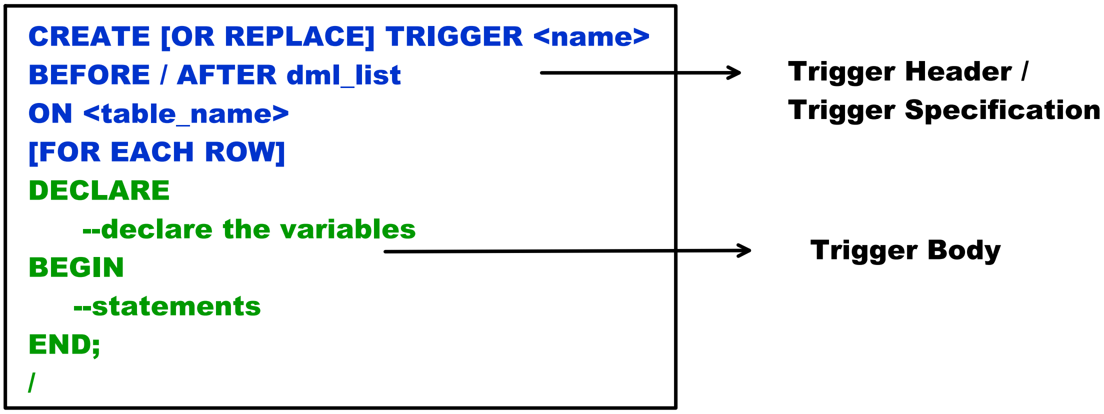
2. Schema Level Trigger / DDL Trigger / System Triggers [DBA]
3. Database Level Trigger / DDL Trigger / System Triggers [DBA]

Table Level Trigger:
If a trigger is defined on table then it is called
Table Level Trigger.
It has 2 sub types. they are:
=> statement level trigger
=> row level trigger

statement level trigger:
it gets executed once for 1 DML statement.

row level trigger:
it gets executed once for every row affected by DML.

Syntax of Table Level Trigger:



```
CREATE [OR REPLACE] TRIGGER <name>
BEFORE / AFTER dml_list
ON <table_name>
[FOR EACH ROW]
DECLARE
     --declare the variables
BEGIN
    --statements
END;
/
```

Trigger Header /
Trigger Specification

Trigger Body

NOTE:

Before Trigger:

- First Trigger gets executed.
- Then DML operation will be performed.

After Trigger:

- First DML operation will be performed.
- Then Trigger gets executed.

Example:

Define a trigger to don't allow the user to perform DMLs on SUNDAY:

```
CREATE OR REPLACE TRIGGER t1
BEFORE insert or update or delete
ON emp
BEGIN
  IF to_char(sysdate,'dy')='sun' THEN
    raise_application_error(-20050, 'you cannot perform DMLs on sunday');
  END IF;
END;
/
```

Testing:
From mon to sat:

```
UPDATE emp SET sal=sal+1000;
```

Output:

```
14 rows updated
```

On Sunday:

```
UPDATE emp SET sal=sal+1000;
```

Output:

```
ERROR:
ORA-20050: you cannot perform DMLs on sunday
```

Example:

Define a trigger to don't allow user to perform DMLs on emp table before or after office timings:
[office timings: 10AM to 4PM]

```
CREATE OR REPLACE TRIGGER t2
BEFORE insert or update or delete
ON emp
DECLARE
      h INT;
BEGIN
     h := to_char(sysdate,'HH24');

    IF h NOT BETWEEN 10 AND 15 THEN
        raise_application_error(-20050, 'DMLs allowed b/w 10am to 4pm only');
    END IF;
END;
/
```

Testing:
From 10am to 3.59pm:

```
UPDATE emp SET sal=sal+1000;
```

Output:

```
14 rows updated.
```

Before 10am or after 3.59pm:

```
UPDATE emp SET sal=sal+1000;
```

Output:

```
ERROR:
ORA-20050: DMLs allowed b/w 10am to 4pm only
```

Example:

Define a trigger to don't allow user to update empno:

```
CREATE OR REPLACE TRIGGER t3
BEFORE update OF empno
ON emp
BEGIN
    raise_application_error(-20050, 'you cannot update empno');
END;
/
```

:NEW and :OLD:

- These are built-in variables.
- These are %ROWTYPE variables.
- :NEW holds new row.
- :OLD holds old row.
- These are called pseudo records.
- These can be used in row level trigger only.

DML	:NEW	:OLD
INSERT	New row	null
UPDATE	New row	Old row
DELETE	null	Old row

STUDENT

SID	SNAME

```
INSERT INTO student
VALUES(1001,'A');
```

:new

SID	SNAME
1001	A

:old

SID	SNAME
null	null

```
UPDATE student
SET sname='B'
WHERE sid=1001;
```

:new

SID	SNAME
1001	B

:old

SID	SNAME
1001	A

```
DELETE FROM student
WHERE sid=1001;
```

:new

SID	SNAME
null	null

:old

SID	SNAME
1001	B

Example:

Define a trigger to record deleted rows in emp_resign table:

EMP

Empno	Ename	,....	..

EMP_RESIGN

DOR	EMPNO	ENAME	JOB	SAL

```
CREATE OR REPLACE TRIGGER t4
AFTER delete
ON emp
FOR EACH ROW
BEGIN
     INSERT INTO emp_resign
     VALUES(sysdate, :old.empno, :old.ename, :old.job, :old.sal);
 END;
/
```

Testing:

```
DELETE FROM emp WHERE job='MANAGER';
```

Output:

```
3 rows deleted.
```

[trigger gets executed for 3 times]

```
SELECT * FROM emp_resign;
```

Trigger Predicates:

Trigger Predicates are keywords.
To identify operation type we can use trigger predicates.

PL/SQL provides following Trigger Predicates:

- **INSERTING**
- **UPDATING**
- **DELETING**

Trigger Predicate	Insert	Update	Delete
Inserting	T	F	F
Updating	F	T	F
Deleting	F	F	T

Define a Trigger to audit emp table:

Emp_audit

Uname	Op_date_time	Op_type	Old_empno	Old_ename	Old_sal	New_empno	New_ename	New_sal
user	systimestamp	op	:old.empno	:old.ename	:old.sal	:new.empno	:new.ename	:new.sal

```
CREATE OR REPLACE TRIGGER t10
AFTER insert or delete or update
ON emp
FOR EACH ROW
DECLARE
         op VARCHAR2(10);
BEGIN
        IF inserting THEN
   op := 'INSERT';
        ELSIF deleting THEN
   op := 'DELETE';
        ELSIF updating THEN
   op := 'UPDATE';
        END IF;

        INSERT INTO emp_audit
        VALUES(user, systimestamp, op, :old.empno, :old.ename, :old.sal,
         :new.empno, :new.ename, :new.sal);
END;
/
```

Testing:

```
INSERT INTO emp(empno,ename,sal)
```

```
VALUES(5001, 'ABC', 8000);

COMMIT;

SELECT * FROM emp_audit;

UPDATE  ...
COMMIT;
SELECT * FROM emp_audit;

DELETE ....
COMMIT;
SELECT * FROM emp_audit;
```

Example:
Define a trigger to don't allow user decrease the salary:

```
CREATE OR REPLACE TRIGGER t11
BEFORE update
ON emp
FOR EACH ROW
BEGIN
      IF :new.sal<:old.sal THEN
          raise_application_error(-20050, 'you cannot decrease salary..');
      END IF;
END;
/
```

Testing:

```
UPDATE emp SET sal=sal-1000;
```

Output:

```
ERROR:
ORA-20050: you cannot decrease salary..
```

Schema Level Trigger / DDL Trigger / System Trigger:

- **If Trigger is created on SCHEMA [user] then it is called "Schema Level Trigger".**
- **DBA creates it.**
- **To control 1 user's DDL actions DBA defines it.**

Syntax:

```
CREATE OR REPLACE TRIGGER <name>
BEFORE / AFTER <ddl_list>
ON <user_name>.SCHEMA
DECLARE
    --declare the variables
BEGIN
    --statements
END;
/
```

Example:

Define a trigger to don't allow c##batch730am user to drop any DB object:

Login as DBA:
Username: system

```
CREATE OR REPLACE TRIGGER st1
BEFORE drop
ON c##batch730am.schema
BEGIN
      raise_application_error(-20050, 'you cannot drop any db object');
END;
/
```

Testing:
Login as c##batch730am:

```
DROP TABLE emp;
Output:
ERROR:
ORA-20050: you cannot drop any db object

DROP PROCEDURE addition;
Output:
ERROR:
ORA-20050: you cannot drop any db object
```

Database Level Trigger / DDL Trigger / System Trigger:

- **If trigger is created on database then it is called "Database Level Trigger".**
- **DBA defines it.**
- **To control multiples users or all users DDL actions**

We define it.

Syntax:

```
CREATE OR REPLACE TRIGGER <name>
BEFORE / AFTER <ddl_list>
ON database
DECLARE
   --declare the variables
BEGIN
   --statements
END;
/
```

Example:

Define a trigger to don't allow c##batch730am, c##batch9am users to drop any DB Object:

Login as DBA:

```
CREATE OR REPLACE TRIGGER dt1
BEFORE drop
ON database
BEGIN
    IF user IN('C##BATCH730AM' , 'C##BATCH9AM') THEN
         raise_application_error(-20080, 'you cannot drop any db object');
    END IF;
END;
/
```

Testing:
Login as c##batch9am:

```
DROP TABLE emp;
Output:
ERROR:
ORA-20080: you cannot drop any db object
```

Login as c##batch730am:

```
DROP TABLE emp;
Output:
ERROR:
ORA-20080: you cannot drop any db object
```

Disabling and Enabling Trigger:

Syntax:

```
ALTER TRIGGER <name> DISABLE / ENABLE;
```

Example:

```
ALTER TRIGGER t2 DISABLE;
--temporarily t2 will not work

ALTER TRIGGER t2 ENABLE;
--again t2 will work
```

Dropping Trigger:

Syntax:

```
DROP TRIGGER <name>;
```

Example:

```
DROP TRIGGER t2;
```

User_Triggers
User_Source

User_Triggers:

- It is system table.
- It maintains all triggers info.

To see triggers info:

```
DESC user_triggers;
```

```
SELECT trigger_name, trigger_type, triggering_event, table_name
FROM user_triggers;
```

User_Source:

- **It maintains all procedures, functions, packages and triggers information.**

To see trigger info:

```
SELECT DISTINCT name
FROM user_source
WHERE type='TRIGGER';
```

To see trigger code:

```
SELECT text
FROM user_source
WHERE name='T10';
```

NOTE:
If we drop the table,
All triggers created on it will be dropped.

TABLE
rows and columns
constraints
triggers
indexes

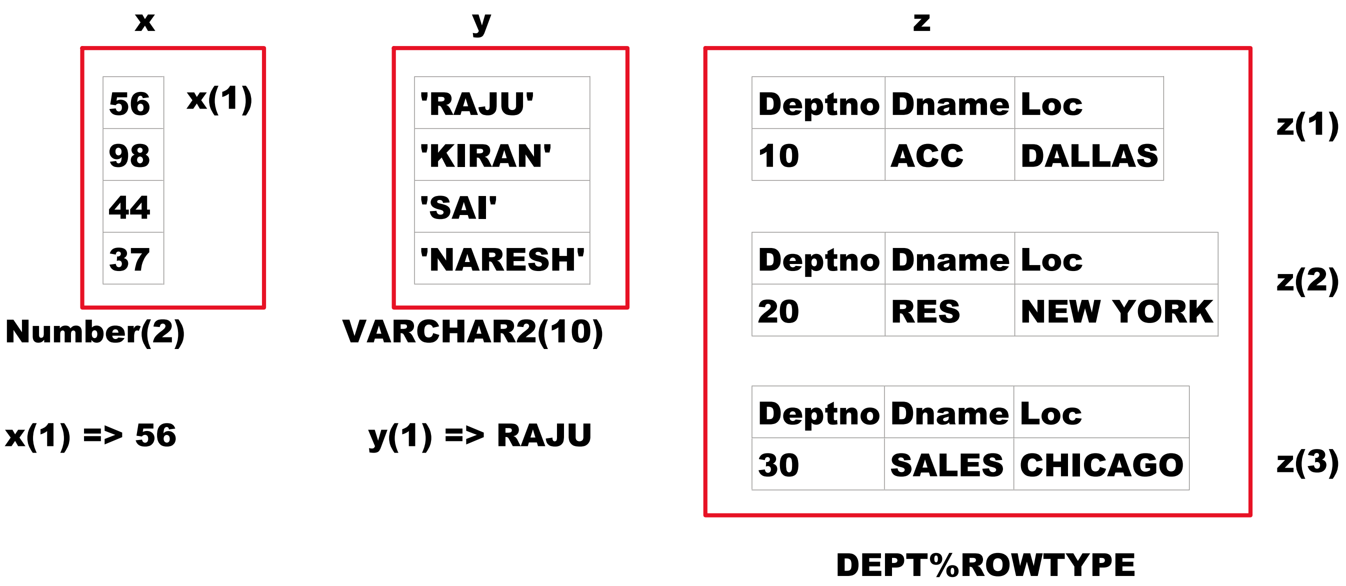
COLLECTIONS

Thursday, December 5, 2024 7:46 AM

COLLECTION:

- COLLECTION is a set of elements of same type.

Example:



x

56	x(1)
98	
44	
37	

Number(2)

y

'RAJU'
'KIRAN'
'SAI'
'NARESH'

VARCHAR2(10)

z

Deptno	Dname	Loc	
10	ACC	DALLAS	z(1)

Deptno	Dname	Loc	
20	RES	NEW YORK	z(2)

Deptno	Dname	Loc	
30	SALES	CHICAGO	z(3)

DEPT%ROWTYPE

x(1) => 56

y(1) => RAJU

z(1).deptno => 10
z(1).dname => ACC

NOTE:
CURSOR is used to hold multiple rows and process them.
COLLECTION is used to hold multiple rows and process them.

CURSOR has some drawbacks. To avoid them we use COLLECTION.

Types of Collections:

3 Types:

- Associative Array / PL SQL Table / Index By Table
- Nested Table
- V-Array

Associative Array:

- **Associative Array is a table of 2 columns.**
  They are:
  - **INDEX**
  - **ELEMENT**
- **In this, INDEX can be NUMBER type or VARCHAR2 type.**

Examples:

x

INDEX	ELEMENT
1	78
2	94
3	34
4	50

x(1) => 78

y

INDEX	ELEMENT
DELHI	1200000
HYD	900000
BLR	1000000

y('DELHI') => 1200000

Creating Associative array:

2 steps:

- **Define our own data type**
- **Declare variable for that data type**

Defining our own Associative Array data type:

Syntax:

```
TYPE <name> IS TABLE OF <element_type>
INDEX BY <index_type>;
```

Example:

```
TYPE num_array IS TABLE OF number(4)
INDEX BY binary_integer;
```

NOTE:

If INDEX is number type use binary_integer (or)

pls_integer.

Declaring variable for that data type:

Syntax:

```
<variable> <data_type>;
```

Example:

```
x NUM_ARRAY;

x(1) := 50;
x(2) := 78;
x(3) := 44;

  (or)

x := num_array(50,78,44);     --oracle 21c
```

x

INDEX	ELEMENT
1	50
2	78
3	44

- num_array is collection constructor.
- Collection Constructor is a special function.
- When we define our own data type implicitly a special function will be defined with data type name. It is called "Collection Constructor".
- It is used to bring values into collection.

Collection members	Purpose	Example
First	First index	x.first
Last	Last index	x.last
Next	Next index	x.next(2) => 3 Next index of 2
Prior	Previous Index	x.prior(2) => 1 Prev index of 2

Example on Associative Array:

Create an associative array as following:

x

INDEX	ELEMENT
1	50
2	78
3	44

```
DECLARE
    TYPE num_array IS TABLE OF number(4)
    INDEX BY binary_integer;

    x NUM_ARRAY;
BEGIN
    x := num_array(50,78,44);

    dbms_output.put_line(x(2));
    dbms_output.put_line('first ind=' || x.first);
    dbms_output.put_line('last ind=' || x.last);
    dbms_output.put_line('next ind of 2=' || x.next(2));
    dbms_output.put_line('prev ind of 2=' || x.prior(2));

    dbms_output.put_line('elements are:');
    FOR i IN x.first .. x.last
    LOOP
        dbms_output.put_line(x(i));
    END LOOP;
END;
/
```

Create an associative array, hold all dept table records in it and print them:

d

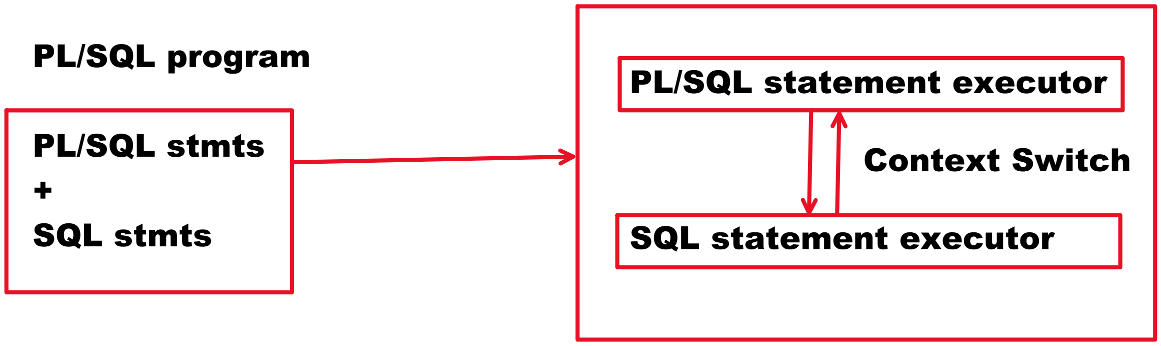
INDEX	ELEMENT
1	DEPTNO DNAME LOC 10 ACC NEW YORK
2	DEPTNO DNAME LOC 20 RES DALLAS
3	DEPTNO DNAME LOC 30 SALES CHICAGO

```
DECLARE
     TYPE dept_array IS TABLE OF dept%rowtype
     INDEX BY binary_integer;

     d   DEPT_ARRAY;
BEGIN
     SELECT * INTO d(1) FROM dept WHERE deptno=10;
     SELECT * INTO d(2) FROM dept WHERE deptno=20;
     SELECT * INTO d(3) FROM dept WHERE deptno=30;
     SELECT * INTO d(4) FROM dept WHERE deptno=40;

     FOR i IN d.first .. d.last
     LOOP
        dbms_output.put_line(d(i).deptno || '    ' || d(i).dname);
     END LOOP;
END;
/
```

PL/SQL Engine



PL/SQL program

PL/SQL stmts
+
SQL stmts

PL/SQL statement executor

Context Switch

SQL statement executor

PL/SQL engine = PL/SQL stmt executor + SQL stmt executor

PL/SQL stmt executor can execute only PL/SQL statements.
SQL stmt executor can execute only SQL stmts.

If SQL statement is submitted to PL/SQL stmt executor,
It submits it to SQL stmt executor.
SQL stmt executor executes SQL command and gives result back to PL/SQL stmt executor. It is called one "Context Switch".

Context Switch:
Travelling from PL/SQL stmt executor to SQL stmt executor and From SQL stmt executor to PL/SQL stmt executor is called "Context Switch".

Above program degrades performance.
If number of context switches are increased then Performance will be degraded.
In above program, 4 context switches will occur.
It degrades performance.

To improve performance of above program we use BULK COLLECT.

```
SELECT * INTO d(1) FROM dept WHERE deptno=10;
SELECT * INTO d(2) FROM dept WHERE deptno=20;
SELECT * INTO d(3) FROM dept WHERE deptno=30;
SELECT * INTO d(4) FROM dept WHERE deptno=40;
```

4 context switches

```
SELECT * BULK COLLECT INTO d FROM dept;
```

1 context switch

BULK COLLECT:

- It is used to collect entire data at a time with single context switch.
- It reduces number of context switches.
- It improves performance.

```
DECLARE
      TYPE dept_array IS TABLE OF dept%rowtype
      INDEX BY binary_integer;

      d   DEPT_ARRAY;
BEGIN
      SELECT * BULK COLLECT INTO d FROM dept;

      FOR i IN d.first .. d.last
      LOOP
          dbms_output.put_line(d(i).deptno || '    ' || d(i).dname);
      END LOOP;
END;
/
```

Output:

```
10    ACCOUNTING
20    RESEARCH
30    SALES
40    OPERATIONS
```

Example:

EMPLOYEE

EMPNO	ENAME	SAL
1001	A	5000
1002	B	3000
1003	C	8000

HIKE

EMPNO	PER
1001	20
1002	10
1003	15

Increase salary to all employees according to HIKE table percentages:

h

INDEX	ELEMENT
1	EMPNO PER 1001 20
2	EMPNO PER 1002 10
3	EMPNO PER 1003 15

```
DECLARE
    TYPE hike_array IS TABLE OF hike%rowtype
     INDEX BY binary_integer;

     h HIKE_ARRAY;
BEGIN
     SELECT * BULK COLLECT INTO h FROM hike;

    FOR i IN h.first .. h.last
    LOOP
          UPDATE employee SET sal=sal+sal*h(i).per/100
          WHERE empno=h(i).empno;
     END LOOP;

     COMMIT;

     dbms_output.put_line('sal increased to all emps..');
END;
/
```

Above program degrades performance.

```
 FOR i IN h.first .. h.last
 LOOP
     UPDATE employee SET sal=sal+sal*h(i).per/100          3 context switches
     WHERE empno=h(i).empno;
 END LOOP;
```

To improve performance of above program
We use BULK BIND.

BULK BIND:

- For BULK BIND, we define FORALL loop.
- It is used to submit BULK UPDATE / BULK DELETE / BULK INSERT commands.

Syntax of FORALL:

```
FORALL <variable> IN <lower> .. <upper>
     --DML stmt
```

```
FORALL i IN h.first .. h.last
      UPDATE employee SET sal=sal+sal*h(i).per/100
      WHERE empno=h(i).empno;
```

1 context switch

```
DECLARE
     TYPE hike_array IS TABLE OF hike%rowtype
      INDEX BY binary_integer;

      h HIKE_ARRAY;
BEGIN
      SELECT * BULK COLLECT INTO h FROM hike;

     FORALL i IN h.first .. h.last
           UPDATE employee SET sal=sal+sal*h(i).per/100
            WHERE empno=h(i).empno;

      COMMIT;

      dbms_output.put_line('sal increased to all emps..');
END;
/
```

Nested table:

- Nested table is a table of 1 column. i.e: ELEMENT.
- INDEX is always NUMBER type.
- It is same as single dimensional array in C/Java.

Example:

x

ELEMENT	
56	x(1)
78	x(2)
60	
45	
55	

Creating Nested Table:

2 steps:

- Create our own data type
- Declare variable

Create our own data type:

Syntax:

```
TYPE <name> IS TABLE OF <element_type>;
```

Example:

```
TYPE num_array IS TABLE OF number(4);
```

Declare variable:

Syntax:

```
<variable> <data_type>;
```

Example:

```
X NUM_ARRAY;
```

Example on nested table:

Create a nested table as following:

x

ELEMENT
50
78
44

```
DECLARE
     TYPE num_array IS TABLE OF number(4);

     x NUM_ARRAY;
BEGIN
     x := num_array(50,78,44);

     FOR i IN x.first .. x.last
     LOOP
          dbms_output.put_line(x(i));
     END LOOP;
END;
/
```

Example:

Create a nested table and hold dept table records in it:

d

ELEMENT

DEPTNO	DNAME	LOC
10	ACC	NEW YORK

DEPTNO	DNAME	LOC
20	RES	DALLAS

DEPTNO	DNAME	LOC
30	SALES	CHICAGO

```
DECLARE
    TYPE dept_array IS TABLE OF dept%rowtype;
    d    DEPT_ARRAY;
```

```
BEGIN
    SELECT * BULK COLLECT INTO d FROM dept;

    FOR i IN d.first .. d.last
    LOOP
        dbms_output.put_line(d(i).deptno || '    ' || d(i).dname);
    END LOOP;
END;
/
```

V-ARRAY:

- **It is same as nested table. It means, it is a table of 1 column. i.e: ELEMENT**
- **INDEX is always NUMBER type.**
- **We must specify size WHERE AS for nested table we don't specify size.**

Associative array	**=>**	**we can store unlimited num of elements**
Nested table	**=>**	**unlimited**
V-Array	**=>**	**limited**

Creating V-Array:

2 steps:

- **Define our own data type**
- **Declare variable**

Defining our own data type:

Syntax:

```
TYPE <name> IS VARRAY(<size>) OF <element_type>;
```

Example:

```
TYPE num_array IS VARRAY(10) OF number(4);
```

Declaring variable:

Syntax:

```
<variable> <data_type>;
```

Example:

x NUM_ARRAY;

Example on V-Array:

Create v-array as following:

x

ELEMENT
50
78
44

```
DECLARE
     TYPE num_array IS VARRAY(10) OF number(4);

     x NUM_ARRAY;
BEGIN
     x := num_array(50,78,44);

     FOR i IN x.first .. x.last
     LOOP
          dbms_output.put_line(x(i));
     END LOOP;
END;
/
```

Example:

Create v-array and hold dept table records in it:

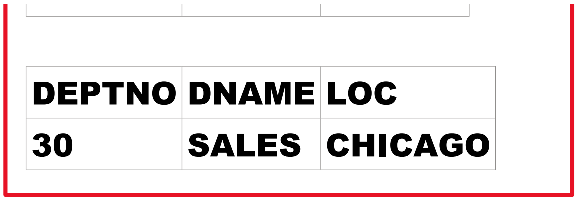
d

ELEMENT

DEPTNO	DNAME	LOC
10	ACC	NEW YORK

DEPTNO	DNAME	LOC
20	RES	DALLAS

DEPTNO	DNAME	LOC



DEPTNO	DNAME	LOC
30	SALES	CHICAGO

```
DECLARE
    TYPE dept_array IS VARRAY(10) OF dept%rowtype;
    d    DEPT_ARRAY;
BEGIN
    SELECT * BULK COLLECT INTO d FROM dept;

    FOR i IN d.first .. d.last
    LOOP
        dbms_output.put_line(d(i).deptno || '    ' || d(i).dname);
    END LOOP;
END;
/
```

Differences b/w CURSOR and COLLECTION:

CURSOR	COLLECTION
• It can move forward only.	• It can move in any direction.
• It supports to sequential accessing. It does not support to random accessing.	• It supports to random accessing Also.
• It is slower.	• It is faster.

Differences among Associative Array, Nested Table and V-Array:

COLLECTION	INDEX	ELEMENT	DENSE or SPARSE

Associative Array	**NUMBER (or) VARCHAR2**	**Unlimited**	**Dense (or) Sparse**
Nested Table	**NUMBER**	**Unlimited**	**Starts as Dense** **It can become Sparse**
V-Array	**NUMBER**	**Limited**	**Dense**

Dense => No gaps

INDEX	ELEMENT
1	50
2	45
3	90

Sparse

INDEX	ELEMENT
10	50
20	45
35	90

```
x(10) := 50;
x(20) := 45;
x(35) := 90;
```

SQL Developer
tables
views
indexes
proc
func
packages
triggers

DB Designer
Normalization:
to create well structured
tables we follow 1 process.
i.e: Normalization

Working with LOBs

Saturday, December 7, 2024 7:48 AM

LOB => Large Object => Multimedia object => Image, Audio, Video

Binary Related Data Types:

- **BFILE**
- **BLOB**

BFILE:

- **BFILE => Binary File Large Object.**
- **It is used to maintain multimedia object path.**
- **It is a pointer to multimedia object.**
- **BFILE => External Large Object.**
- **It is stored out of the database. In side of database only path is stored.**

D1 => Directory Object
D1 => D:\photos

Example:

DATABASE

EMP1

EMPNO	ENAME	EPHOTO [BFILE]
1234	Ravi	Bfilename('D1', 'ravi.jpg')

D:
photos folder

ravi.jpg

Directory Object:

- **Directory Object is pointer to specific folder.**

Syntax:

```
CREATE DIRECTORY <name> AS <folder_path>;
```

Example:
Login as DBA:
Username: system

```
CREATE DIRECTORY d1 AS d:\photos;

GRANT read, write
ON DIRECTORY d1
TO c##batch730am;
```

Login as c##batch730am:

EMP1

EMPNO	ENAME	EPHOTO [BFILE]

```
CREATE TABLE emp1
(
Empno NUMBER(4),
Ename VARCHAR2(10),
Ephoto BFILE
);

INSERT INTO emp1
VALUES(1234, 'A', bfilename('D1', 'ellison.jpg'));

COMMIT;
```

BLOB:

- BLOB => Binary Large Object
- It is used to maintain multimedia object inside of table.
- It can be also called as "Internal Large Object".
- It is secured one.

Example:

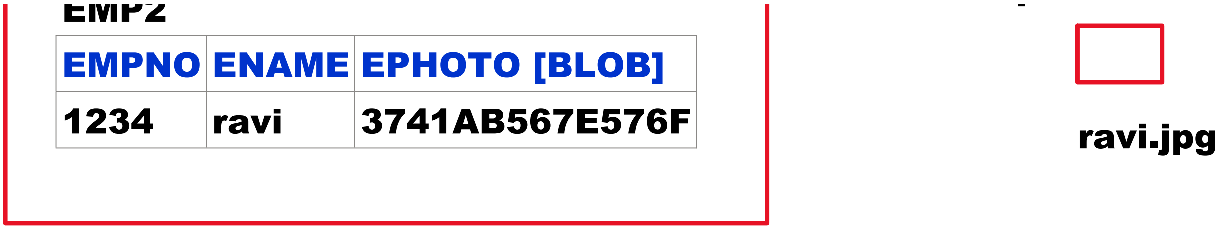
Database

EMP2

EMPNO	ENAME	EPHOTO [BLOB]

D:

photos folder



EMP2

EMPNO	ENAME	EPHOTO [BLOB]
1234	ravi	3741AB567E576F

Example on BLOB:

EMP2

EMPNO	ENAME	EPHOTO [BLOB]

```
CREATE TABLE emp2
(
Empno NUMBER(4),
Ename VARCHAR2(10),
Ephoto BLOB
);

INSERT INTO emp2
VALUES(1234, 'ELLISON', empty_blob());
```

Define a procedure to update image:

```
CREATE OR REPLACE PROCEDURE
update_photo(p_empno NUMBER, p_fname VARCHAR2)
AS
      s BFILE;
      t BLOB;
      length NUMBER;
BEGIN
      s := bfilename('D1', p_fname);   --stores img path in s

       SELECT ephoto INTO t FROM emp2
       WHERE empno=p_empno FOR UPDATE;      --locks record

       dbms_lob.open(s, dbms_lob.lob_readonly);  --opens file in read mode
       length := dbms_lob.getlength(s);              --finds img size
```

```
        dbms_lob.LoadFromFile(t, s, length);            --writes img into t
                                                        --t has img

         UPDATE emp2 SET ephoto=t
        WHERE empno=p_empno;                            --t img updates in table

        COMMIT;

        dbms_lob.close(s);                               --closes opened file

        dbms_output.put_line('img saved in table..');
END;
/
```

Calling:

```
SQL> exec update_photo(1234, 'ellison.jpg');
```

Output:

```
img saved in table..
```

Dynamic SQL

Tuesday, December 10, 2024 7:38 AM

Dynamic SQL:

- **DRL, DML, TCL commands can be used directly in PL/SQL program.**
- **DDL, DCL commands cannot be used directly in PL/SQL program. To use them, we use DYNAMIC SQL.**

Static Query:

- **In SQL, we have written many queries.
  All those are static queries.**

Example:

```
DROP TABLE emp;
TRUNCATE TABLE dept;
```

Dynamic Query:

- **A query which is built at run time is called "Dynamic Query".**

  Example:

  ```
  EXECUTE IMMEDIATE 'DROP TABLE ' || v_tname;

  EXECUTE IMMEDIATE 'TRUNCATE TABLE ' || v_tname;
  ```

Dynamic SQL:

- **Dynamic SQL concept is used to execute dynamic queries.**
- **Submit Dynamic query as string to EXECUTE IMMEDIATE command.**

- **To use DDL or DCL commands in PL/SQL we use Dynamic SQL.**
- **When we don't know exact table name or column name**

at compilation time then we Dynamic SQL.

Examples on Dynamic SQL:

Define a procedure to drop a table:

```
CREATE OR REPLACE PROCEDURE
drop_table(p_tname VARCHAR2)
AS
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE ' || p_tname;

   dbms_output.put_line(p_tname || ' table dropped');
END;
/
```

Calling:

```
SQL> EXEC drop_table('salgrade');
```

Output:

```
salgrade table dropped
```

Define a procedure to drop any DB Object:

```
CREATE OR REPLACE PROCEDURE
drop_object(p_obj_type VARCHAR2, p_obj_name VARCHAR2)
AS
BEGIN
        EXECUTE IMMEDIATE 'DROP ' || p_obj_type || ' ' || p_obj_name;

       dbms_output.put_line(p_obj_name || ' ' || p_obj_type || ' dropped');
END;
/
```

Module-1: Tables

SQL commands
Built-In Functions
Clauses
Joins
Sub Queries
Constraints
Set operators

Module-2: PL/SQL

PL/SQL Basics
Control Structures
Cursors
Stored Procedures
Stored Functions
Packages
Triggers
Exception Handling
COLLECTIONS
Working with LOBs
Dynamic SQL

Module-3: Other DB objects

SEQUENCES
VIEWS
INDEXES
M.VIEWS
SYNONYMS