PL/SQL Questions

Q.1.) Write a pl/sql block accepts a number from user.

Ans1.)
```
declare
    n number := &num;
    result number;
begin

result := n*n;
    dbms_output.put_line('The square of a number'
                        ||n|| 'is' || result);
end;
```

→ Enter the value of num 10;

Q.2.) Write a PL/SQL block to display addition, subtraction, multiplication and division of any two numbers entered by user.

```
declare
    n1 number:= &n1;
    n2 number:= &n2; ~~result number;~~
    result number;
  ~~begin~~
begin
  result:= n1+n2;
        dbms_output.put_line("The addition of 2
                numbers" ||n1|| "and" ||n2||
                "is" ||result);
  result:= n1 - n2;
        dbms_output.put_line("The subtraction of 2
                numbers" ||n1|| "and" ||n2||
                "is" ||result);
  result:= n1 / n2;
        dbms_output.put_line("The division of 2 numbers
                ||n1|| "and" ||n2|| "is" ||result);
  result:= n1* n2;
        dbms_output.put_line("The multiplication of 2 numbers
                ||n1|| "and" ||n2|| "is" ||result);
end;
```

PAGE No.
DATE / /

Q.3.) Write a PL/SQL block which displays job location of an employee whose ID is to be accepted by user.
Emp
(Empno, ename, sal, job, comm, dept).

Ans3.)
```
declare
    id number:=&id;
    v varchar2(20);
begin
  Select job into v from emp where empno=id;

dbms_output.put_line(v);

end;
```

Q.4.) Write a PL/SQL to display employee name and salary of an employee whose ID is to be accepted by user.
Emp
(Emp no, ename, sal, job, comm, dept)

Ans4.)
```
declare
    id number:=&id;
    v1 varchar2(20);
    v2 number(20);
begin
 Select ename, sal into v1, v2 from emp where empno=
id;
dbms_output.put_line(v1||' '||v2);
end;
```

DATE / / /

Q.5) Write a PL/SQL block to display all the details of department table using unconstrainted loop.
Dept
(Deptno, dname, loc)

```
declare
    i number:= 10;
    dnm varchar2(20);
    l varchar2(20);
begin
  loop
  Select dname, loc into dnm, l from dept
  where deptno=i;
  dbms_output.put_line(i||' '||dnm||' '||l);
  i:= i+10;
  exit when i=50;
  end loop;
end;
```

Q.6) Write a PL/SQL block to display table of three using unconstrainted loop.

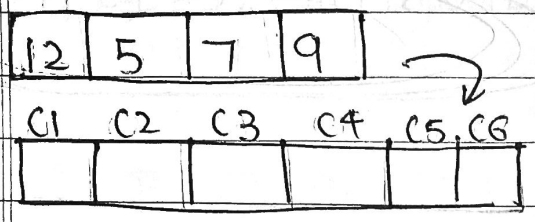
Ans 6.)

```
declare
    i number:=3;
    j number:=1;

begin
  loop
  dbms_output.put_line(i*j);
  j:= j+1
  exit when j=11;
  end loop;
    end;
```

PAGE No.
DATE / /

Record Variable → we extract whole row into one record variable
↳ simply table name is enough



E.g:
```
declare
    v emp%ROWTYPE;
    id number = &:id;

begin
Select * into v from emp where empno = id;
dbms_output.put_line(v.ename || ' ' || v.sal);
end;
```

Q.1) Write a PL/SQL block to display department name and location of a department 20 using record variable.

var tablename%ROWTYPE;

Dept
(Dept no, dname, loc)

Ans:-
```
declare
    v Dept%ROWTYPE;
begin
Select * into v From Dept where dept no = 20;
dbms_output.put_line(v.dname || ' ' || v.loc);

end;
```

PAGE No.
DATE / /

→ we don't have to guess data type of col.

Percent-type variable → Hold value for only one single column.

[var Tablename.col%type]

```
declare
  dn dept.dname%type;
  l dept.loc%type;

begin
Select dname, loc into dn, l where dept no=10;
dbms_output.put_line(dn||' '||l);
end;
```

Q.2.) Write a PL/SQL block to display all the details of department table using ~~display all the details~~ unconstrained loop.
~~Display~~ Dept → using percent type variable
(dept no, dname, loc)

→ Dept no. is starting from 10 with increment 10, like 10, 20, ...

Ans2.)

```
declare
  dno dept.deptno%type;
  dn dept.dname%type;
  l dept.loc%type;
  i number:=10;
begin
 loop
Select dept no, dname, loc into dno, dn, l where
deptno=i;
dbms_output.put_line(dno||' '||dn||' '||l);
 i=i+10
exit loop when i=50;
end loop;
   end;
```

PAGE No. Tuesday
DATE 05/07/2022

Sequences →

To alter only we just not need to write start

Create or replace sequence seq_name.

[Start with <integer value> |
increment by <integer value>
minvalue <integer_value> | NOMIN VALUE |
maxvalue <integer_value> | NOMAXVALUE |
CACHE <integer_value> |
CYCLE |NOCYCLE|
ORDER | NOORDER|];

Already create this → user input and default

Q.1) Create a sequence which will start with 301 and end up at 350 with an interval 4.

Ans1) Create or replace sequence S1

Start with 301
increment by 4
NOMIN VALUE
maxvalue 350 ;
~~CACHE~~
~~CYCLE~~
~~ORDER~~ ;

Bank → Two counter → Token numbers → every time

Two diff time

105

10

Two work

1028102X

sequences numbers

increment sequence

PAGE No.

DATE

multiple machine

work

should recover

→ Here, we can handle increament on multiple machine.

ORDER → To use on multimachine.

Q. 2.) Write a sequence which will start with 51 and ends up with 200. with an interval of 2 let this sequence be ordered on different machine and cycle. Store 5 number into cache memory ~~itself~~ as well.

Create or replace sequence S2

Start with 51
Increment by 200
Cycle
noorder ;

Q.3.) Alter a sequence created in question number 1 by making it cycled sequence

Alter sequence S1
cycle ;

Procedure, full

PAGE No.
DATE / /

To print current value select S1.currval from dual;

To print next value select S1.nextval from dual;

~~Q.4)~~ Alter value sequence

Alter sequence seq_name

~~Ans4) Create or replace~~

Q.4) Drop a sequence that you have created in previous question.

Ans4) drop sequence S1;

[insert into emp (emp_no) values (S1.nextval)] → In Practical

Q.5) ~~Create~~ Insert a sequence into table and generate its value. Token (token_no, date, time, purpose)

Ans5) >insert into token(token_no)
values(S1.nextval);

Stored Procedures

→ 7/07/2022 Thursday

Create or replace procedure proc_name
(variable 1 IN|OUT|INOUT datatype,
variable 2 IN|OUT|INOUT datatype, ...)
is/as
——— ... variable
——— ... declaration block
begin
—— .... execution block
end;

Q.1) Create a procedure which displays square of a number. Using in, out and inout parameter.

Ans1) Using in parameter

PAGE No.
DATE / /

```
Create procedure P1( n in number)
is
begin
dbms_output.put_line ('Square of a number' ||n|| 'is'
                        mod (n, 2));
>execute procedure P1(10);
 100
```

Using OUT parameter

```
Create procedure P1(n in number, sq out number)
is
begin
    sq: n*n;
end;

>variable v1 number;
> execute procedure P1(10, :v)
> print v;
> 100
```

Using in-out parameter

```
Create procedure P1(n in out parameter)
is
declare
  num number := &num;
begin := n*n;
P1 (num)
dbms_output.put_line(num);
end;
```

PAGE No.
DATE / /

Q.2) Create a procedure to display addition, subtraction, multiplication and division using in, then out and then inout parameter.

Using IN

```
Create procedure P2
    (n1 in number,
     n2 in number)
is
begin
dbms_output.putline('Addition of two number is'||n1+n2);
dbms_output.put_line('Subtraction of two number is'||n1-n2);
dbms_output.put_line('Multiplication of two number is'||n1*n2);
dbms_output.put_line('Division of two number is'||n1/n2);

>execute procedure P2(10,5);
Addition of two number is 15
Subtraction of two number is 5
Multiplication of two number is 50
Division of two number is 2
```

Using Out

```
Create procedure P2
    (n1 in number,
     n2 in number,
     add out number,
     sub out number,
     mul out number,
     div out number)
```

PAGE No.
DATE / /

```
is
begin
    add:= n1+n2
    sub:= n1-n2
    mul:= n1*n2
    div:= n1/n2
end;

>variable v1 number
>variable v2 number
>variable v3 number
>variable v4 number
>execute procedure P2(10, 8, :v1, :v2, :v3, :v4)
>print v1, v2, v3, v4;
18 2 80 2.5
```

Using in-out

```
Create procedure P2
  (n1 in out number,
   n2 in out number,
   n3 out number,
   n4 out number)
is/as
    t1 number;
    t2 number;
begin
    t1:= n1+n2
    t2:= n1-n2
    n3:= n1*n2
    n4:= n1/n2
```

PAGE No.
DATE / /

```
    n1:=t1;
    n2:=t2;
end;
declare
    n1 number:= 10;
    n2 number:= 5;
    n3 number;
    n4 number;

begin
  P2(n1, n2, n3,n4);
 dbms_output.put_line(n1||' '||n2||' '||n3||' '||n4);
end;
```

Q.3) Create a procedure to check if the number is even or odd.

Ans 3.) Using in

```
Create procedure P3 (n1 in number)
is
begin
If mod(n1,2)=0
  then
    dbms_output.put_line ('The number' ||n1||'is even');
else
  then
    dbms_output.put_line('The number'||n1||'is odd');
end;
>execute P3(9);
The number 9 is odd
```

PAGE No.
DATE / /

Using inout

```
Create procedure P1 (n in out number)
is
begin
    if mod(n,2)=0 then
        n:=1;
    else
        n:=0;
end if;
    end;

declare
    number num:= & num;
begin
    P1(num);
 if num=1 then
  dbms_output.putline ('The number is 'even');
 else
  dbms_output.putline('The number is odd');
   end if;
end;
```

Using out

```
Create procedure P1
    (n in number,
     result out varchar2)
is
begin
if mod(n,2) = 0 then
```

PAGE No.
DATE / /

```
    result := 'The number' || n || 'is even';
else
    result := 'The number' || n || 'is odd';
end if;
```

```
> variable v varchar2(20);
> execute procedure P1(10, :v);
> print & v;
```

Q.4) Create a procedure which displays employee name & his salary. Accept employee no from the user. Emp(empno, ename, sal, job, comm, dept) by using in, out, in-out parameter.

(H.W)

Functions

```
Create or replace function
  f(variable 1   IN/OUT/INOUT datatype ,
    variable n   IN/OUT/INOUT datatype)

return datatype
 is/as
  --- declaration block

begin
  --- execution block

 return abc → variablename
end;
```

Q.1) Write a function to calculate cube of a number.

Ans1)
```
Create function f(n in number)
   return number
is/as
    cube number;
begin
   cube := nxnxn;
return cube
end;

declare
result number := &n;

begin
```

PAGE No.
DATE / /

```
result:= f(n);
dbms_output.put_line(result);
end;
```

Q.2.) Create a function to display the designation and commission of an employee whose employee number is to be accepted by user.

Ans2.)

```
Create or replace function f(eno in number, desi out varchar2)
return number
is/as
    c number;
begin
Select job, comm into desi, c from emp where
empno= eno;
    return c;
end;

declare
    designation varchar2(20);
    c number;
    e number:= & empno;
begin
& c:= f(e, designation);
    dbms_output.put_line(designation ||' ' ||c);
end;
```

c:=f(n);
a=function(01);

For Logical Operation Q.4.) Create a pl/sql block which displays an appropriat message based on the no. of rooms of a rom of a particular hotel.

no. of rows	message
1-2	small
3-4	~~average~~ fairly big
5-7	~~little bigger~~ Lots of rooms

Hotel (Roomid, no_of_rooms, floor)

Ans 1:
```
declare
    rid number:=&roomid;
    nor number;

begin
Select no_of_rooms into nor from Hotel where roomid=
 rid;

if nor = 1 or nor = 2 then
dbms_output.put_line('small');

elsif nor = 3 or nor = 4 then
dbms_output.put_line('fairly big');

else nor = 5 or nor = 7 then
dbms_output.put_line('Lots of rooms');

  end if;
end;
```

Q.2.) Create a PL/SQL block which accepts a number from the user for an arithmetic operation. Perform that arithmetic operation and display the result. Accept two number from the user.

PAGE No.
DATE / /

Ans2:-

```
declare
        n1 number := & n1;
        n2 number := & n2;
        o number := &operator;
begin
dbms_output.put_line('Select your operation: ');
dbms_output.put_line ('1] Addition: ');
dbms_output.put_line ('2] Subtraction: ');
dbms_output.put_line('3] Multiplication: ');
dbms_output.put_line('4] Division: ');
if o=1 then
dbms_output.put_line ('Addition is '||n1+n2);
elsif o=2 then
dbms_output.put_line('Subtraction is '||n1-n2);
elsif o=3 then
dbms_output.put_line('Multiplication is '||n1*n2);
else
dbms_output.put_line('Division is '||n1/n2);
endif;
end;
```

PAGE No.
DATE 20/7/2022

Exceptions

i) Predefined
ii) User defined
iii) Redeclaring predefined exception

i) Predefined

Syntax:
```
declare
begin
  ___
  ___
  ___
Exception
  When <exception_name> then
  ___
  ___

  When others then
  ___
  ___
end;
```

Q.1] Write PL/SQL block which prints employee designation whose employee no is 7902. Fire a suitable exception.

Code:
```
declare
eno number:= 7902;
var varchar(20);
begin
Select job into var from emp where empno=eno;
dbms_output.put_line(var);
exception
when no_data_found then
```

PAGE No.
DATE / /

```
dbms_output.put_line(eno||'does not exist');
end;
```

OUTPUT
```
SQL>
7902 does not exit
PL/SQL procedure successfully completed
```

Q.2] Write a PL/SQL block which prints division of two numbers accept both the numbers from user.

CODE:
```
declare
n1 number:=&n1;
n2 number:=&n2;
n3 number;
begin
n3:=n1/n2;
dbms_output.put_line(n3);
exception
when zero_divide then
dbms_output.put_line('As one of the no. is zero. Division cannot happen');
end;
```

Q.3] Write a PL/SQL block using pre-defined exception to display the details of employee whose dept no is 20. The details empno and ename should be displayed. Fire an suitable exception.

CODE:
```
declare
eno number;
empname varchar2(20);
```

PAGE No.
DATE / /

```
begin
Select empno, ename into emp where deptno=20;
dbms_output.put_line (en||' '||empname);
exception
when too_many_rows then
dbms_output.put_line('Too many rows for this column');
end;
```

OUTPUT: SQL>
too many rows for this column

Q.4] Write a PL/SQL block to insert a new record in employee table. fire an suitable exception

```
declare
begin
Insert into emp values(123,'abc','clerk',10,'12-Jan-2017',
                5000, 1000, 10);
exception
when dup_val_on_index then
dbms_output.put_line('Record cannot be included);
end;
```

User-defined Exceptions:

Syntax:
```
declare
   exception_name Exception;
begin
   if condition then
   Raise exception_name;
end if;
```

Automation Testing

PAGE No.

DATE

Repetitive tasks

Exception
when exception_name then
statement;
end;

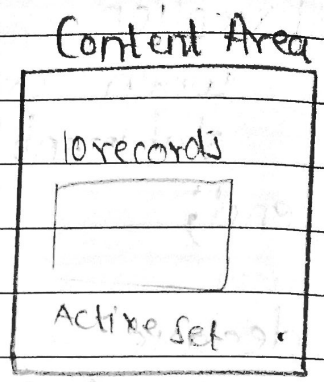
Using previous question by user_defined exception

Q.1]
```
declare
eno number:=7902;
var varchar2(20);
  Not_found Exception;
begin
Select job into var from emp
where empno=eno;
if var is Null then
Raise ~~unavailable~~ Not_Found;
end if;
dbms_output.put_line(var);
end;
```

Q.2]
```
declare
n1 number:=&n1;
n2 number:=&n2;
n3 number;
cannot_divide Exception;
begin
if n1=0 or n2=0
Raise cannot_divide
n3:=n1/n2;
end if;
dbms_output.put_line(n3);
end;
```

PAGE No.
DATE / /

Cursors



2) -

Cursors are of two types
1.) Implicit cursor
2.) Explicit cursor

→ How to create explicit ~~func~~ cursor

Explicit cursor

Defining a cursor

→ Cursor <cursor_name> is
<Select statement>; → declare block

All 3 should come otherwise no use of cursor

① Open
Open<cursor_name>; → execution block

② Fetch
Fetch cursor_name into variable 1, variable 2.....
OR
Fetch cursor_name into record_variable;

③ Close
Close<cursor_name>;

Q.1. Display the details of department table using cursor.

1

Dept
C

Implicit - Works without us → to modify records.
Explicit - Works with variable defined by us

PAGE NO. 45
DATE / /

[Cursor points to only 1 data hence (loop)
Fetch comes in loop

Attribute of cursor,

```
declare
cursor c1 is select * from dept;
record dept%ROWTYPE;
begin
open c1;
loop
fetch c1 into record;
exit when c1%NOTFOUND;
dbms_output.put_line(record.deptno||' '||record.loc);
end loop c1%NOTFOUND;
end loop;
close c1;
end;
```

Attribute of cursor → (explicit)

→ Depends on fetch

Returns Boolean <cursor_name>%NOTFOUND → True when cursor (data) is not found.

Returns Boolean <cursor_name>%FOUND → True when cursor found

Returns numeric value <cursor_name>%ROWCOUNT → Gives no. of rows

Returns Boolean <cursor_name>%ISOPEN

Implicit Cursor

→ Depends on Select/Update/Delete

Attributes of cursor (implicit):
SQL%NOTFOUND
SQL%FOUND
SQL%ROWCOUNT
SQL%ISOPEN → Always returns true value.

PAGE No.
DATE / /

Q.2. Write a pl/sql block that uses cursors to display department name, column of table department. 1.) It should check whether the cursor is open. (using explicit)
2.) It should use %FOUND attribute.

```
declare
    cursor c1 is Select * from dept;
    record dept%ROWTYPE;
begin
open c1;
if c1%ISOPEN then
loop
fetch c1 into record;
if c1% FOUND then
dbms_output.put_line(record.deptname);
else
 exit;
 end if;
end loop;
end;
```

Implicit cursor

Q.3. Write a pl/sql block to delete ename from emp table empno is 7788.

```
declare
name varchar2(20);
begin
select ename into name from emp where empno=7788;
if sql%found then
delete from emp where empno=7788;
else EXIT;
```

rao%COUNT → if 5 salary then 5

PAGE No.
DATE / /

```
end if;
end;
```

loop is compulsory in fetch statement

Output: SQL>
PL/SQL procedure successfully completed.

Q4 Write a pl/sql block to update record in table. Use ~~%~~ %notfound attribute.

```
begin
update emp set comm=1000 where empno=7876;
if sql%notfound then
dbms_output.put_line ('Record not updated');
else
dbms_output.put_line ('Record updated');
end if;
end;
```

Output: SQL>
Record updated
PL/SQL procedure successfully completed.

Q.5. Write a pl/sql block to delete a record having salary as 1500 from emp table

(M)

```
declare
begin
delete from emp where sal=1500;
dbms_output.put_line(sql%rowcount);
end;
```

Output: SQL>

Cursor-for loop:

Note: We need not to write open, fetch and close in cursor for loop.

```
declare
cursor c1 is Select* from dept;
record dept%ROWTYPE;
begin
  for i in c1
loop
dbms_output.put_line(c1%ROWTYPE||' . '||deptno||' '||
                     i.dname||' '||i.loc);
end loop;
end;
```

PAGE No.
DATE / /

Q.1. Write a pl/sql block to display ~~empname~~ employee name, job and salary of all employees from the employee table, working in given department number. Using cursor

```
declare
    dptnum number:= &dept no;
    cursor c1 is Select* from ~~other~~ emp where deptno=dptnum;
    record emp%ROWTYPE;
begin
  for i in c1
  loop
  dbms_output.put_line(c1%ROWTYPE||':]'||i.empname||' '||
    i.empjob||' '||i.empsal);
  end loop;
end;
```

cl = val
variable

Transaction

Transaction is DBM's abstract view of a user program: a sequence of reads and write.

Properties (ACID)

Atomic

- Either all actions are carried out or none are.
- Not worry about incomplete transaction.

Consistency

- DBMS assumes that the consistency holds for each transaction.

Isolation

- Translations are isolated, or protected, from the effects of concurrently scheduling other transactions.

Durability

- The effects of transaction is present if DBMS informs the user successful execution.

Schedule:

- A list of actions from ~~which~~ a set of transactions as seen by DBMS.

R → Read
W → Write

- The order in which two actions of a transaction T appear in a schedule must be the same as the order they appear in T.

Classification:

- Serial schedule: Schedule does not ~~have~~ interleave the actions of different transactions.

Transaction :- Reading N writing of information done on pertaining to some object. E.g tables.
For users
There only 1

PAGE No.
DATE / /

Problem when RW

- **Equivalent schedules:** For any database state; the effect of executing the first schedule is identical to the effect of executing the second schedule.

Serial sequences for T_1 complete fully then T_2 equivalent. Twosh getting same result

- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions on any consistent database instance.

Example:

Interleaved (and its Problem

S2 + S3
S1 + S3

1. Serial Schedule — S4

T_1	T_2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

2. Interleaved Schedule — S5

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

4. Serializable Schedule — S3

T_1	T_2
R(A)	
	R(B)
W(A)	
	W(B)

Result 3

3. Equivalent Schedule

S1

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)

Result 1

S2

T_1	T_2
	R(B)
	W(B)
R(A)	
W(A)	

Result 2

problems: Anomalies with Interleaved Execution.

Reading Uncommited Data (WR Conflicts, "dirty reads"):

E.g: T_1: A + 100, B+100, T_2: A*106, B*106

T_1:	R(A),	W(A),		R(B),	W(B), Abort
T_2:			R(A), W(A), C		

Unrepeatable Reads (RW Conflicts):

E.g:- T_1: R(A), check if A>0, decrement, T_2: R(A), decrement

T_1:	R(A),			R(A),	W(A), C
T_2:		R(A),	W(A), C		

Overwriting Uncommited Data (WW Conflicts):

T_1:	W(A),		W(B),	C
T_2:		W(A),	W(B),	C

Recoverable Schedule

Next → Lock based database

Transactions commit only after all transactions whose changes they read also commit.

In such a case, we can do cascading abort.

Eg below: Note that T_2 cannot commit before T_1, therefore when T_1 aborts, we can abort T_2 as well.

PAGE No.
DATE / /

T_1:	R(A), W(A),	Abort
T_2:	R(A), W(A), R(B), W(B)	

A transaction reads changes only of committed transactions. Advantage of this approach is: the schedule is recoverable, and we will never have to cascade aborts.

30/07/2022 Lock-Based Concurrency Control

- Only serializable, recoverable schedules are allowed.
- No action of commited transaction are lost while aborted transaction.
- Lock protocol.

Strict Two-phase Locking (Strict 2PL) Protocol:

- Rule 1: Each Transaction must obtain a S (Shared) Lock on object before reading and an X (exclusive) lock on object before writing.
- Rule 2: All locks held by a transaction are released when the transaction complete.

Scheduling illustrating Strict 2PL

T_1:	X(A), R(A), W(A), X(B), R(B), W(B), C	
T_2:		X(A), R(A), W(A), X(B), R(B), W(B), C

PAGE No.
DATE / /

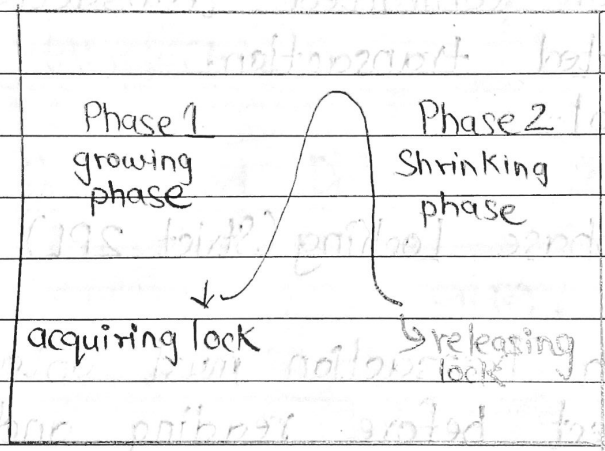
T1 :	S (A), R(A), X(C), R(C), W(C) S
T2 :	S(A), R(A), X(B), R(B), W(B), C

Phase 1:

1. A transaction may obtain Locks.
2. Transaction may not released Lock

Phase 2:

1. Transaction may release locks
2. Transaction may not obtain locks.



Rule 1: Each transaction must obtain S (shared) on object before reading and an X (exclusive) lock on object before writing

Rule 2: A transaction cannot request adding lock after its completion.

PAGE No.
DATE / /

Deadlocks :- Non

- Deadlocks : Two transactions are waiting for locks from each other.
- eg: T_1 holds exclusive locks on A, request an exclusive lock on B, and is queue. T_2 holds an exclusive lock on B, and request lock on A and queue.

T_1	T_2
X(A)	X(B)
R(A)	R(B)
W(A)	W(R)
request X(B)	request X(A)

Lock Conversions

Two phase locking with lock conversions

<u>First phase</u>
- Can acquire a lock-S on item
- Can acquire a lock-X on item
- Can convert a lock-S to a lock(x) (upgrade)

<u>Second phase</u>
- Can release a lock-S
- Can release a lock-X
- Can convert a lock-X to a lock-S (downgrade)

Crash Recovery

- Basic Idea: Logging

- Record REDO and UNDO information, for every update, in a log.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info written to log, so multiple updates fit in a single log page.

- Log: An ordered list of REDO/UNDO actions
  - Log record contains:
    <XID, pageID, offset, length, old data, new data>
  - and additional control info

T_1	P500	70	10	ABC	PQR

Write-Ahead Logging (WAL)

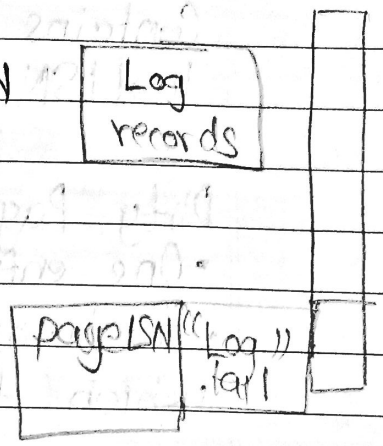
The Write-Ahead Logging Protocol:
- Must force the log record for an update before corresponding data page gets to disk.
- Must write all log records for a Xact before commit.

- #1 guarantees Atomicity
- #2 guarantees Durability

Q. Exactly how is logging (and recovery) done?
- ARIES algorithm

The Log

- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each data page contains a pageLSN
  - for an update to that page.



Log Records

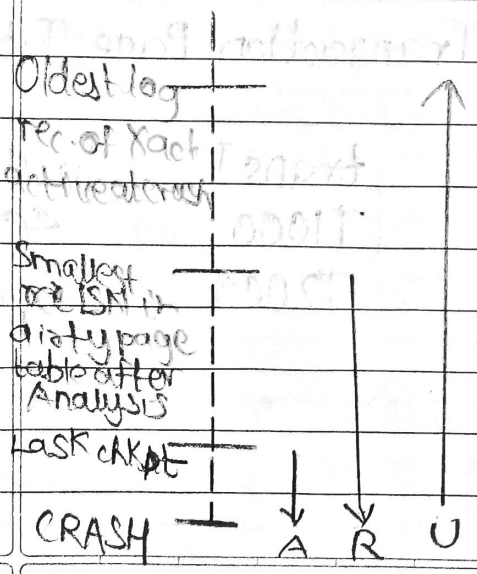
Log Record fields:

- prevLSN
- XID
- type
- pageID
- length
- offset
- before-image
- after-image

(pageID, length, offset, before-image, after-image: update records only)

Possible log records types:

- Update
- Commit
- Abort
- End (signifies end of commit or abort)
- Compensation Log Records (CLRs)
  - for UNDO actions.



- Start from a checkpoint
- Three phases. Need to:
  - Figure out which Xacts committed since checkpoint, which failed (Analysis).
  - REDO all actions. (repeat history)
  - UNDO effects of failed Xacts

PAGE No.
DATE / /

Recovery related structures

Transaction Table:

One entry per active Xact.
Contains XID, status(running/commited/aborted), and lastLSN.

Dirty Page Table:

- One entry per dirty page in buffer pool.
- Contains recLSN --- the LSN of the log record which first caused the page to be dirty.

LSN	prev LSN	transID	Type	pageID	Length	Offset	Before image	After image
10	Null	T1000	Update	P500	3	23	ABC	XYZ
20	Null	T2000	Update	P600	3	45	GHI	MDSF
30	20	T2000	Update	P500	3	78	DF	JKP
40	10	T1000	Update	P505	3	98	EYR	IOP

Dirty Page Table

PageID	recLSN
P500	10
P600	20
P505	40

Transaction Page Table

transID	lastLSN
T1000	40
T2000	30

Last LSN

PAGE No.
DATE / /

Check pointing

Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash. Write to log:

- begin_checkpoint record: Indicates when chkpt began.
- end_checkpoint record: Contains current Xact table and dirty page table.
- fuzzy checkpoint → Master record store
- Store LSN of checkpoint record in a safe place (master record).

Recovery: The Analysis Phase

- Reconstruct state at chekpoint.
  - via end_checkpoint record
- Scan log forward from checkpoint
  → End record: Remove Xact from Xact table.
  → Other record: Add Xact to Xact table, set lastLSN = LSN, change Xact status on C if log record is commit record otherwise U.
  → Update record: If P not in Dirty Page Table,
  • Add P to D.P.T., set its recLSN = LSN.

Recovery: The REDO Phase

- We repeat History to reconstruct state at crash:
  - Reapply all updates (even of aborted Xacts).

loser → a transaction that doesnot commit.

Any (1) R 12 5 marks

- Scan forward from log rec containing smallest recLSN. For each update log recLSN, REDO the action unless:
  - Affected page is not in the Dirty Page Table (D.P.T)

↳ E.g

- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN. No additional logging!

Recovery: The UNDO Phase

10 Trup P3

ToUndo = { lastLSN of a "loser" Xact }

Repeat

- Choose largest LSN among ToUndo.
- Determine if LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is Empty.

Eg:- ~~Transaction Table~~

DPT Table

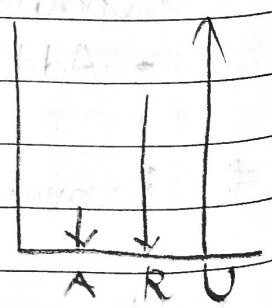
Page ID	recLSN
P1	20
P2	40
P3	10

1st LSN

Transaction Table

Tid	LastLSN
T1	10
T2	40
T3	30

Last LSN



Log LSN refer to undo

Pre LSN	LSN			
N	10	T1	UP	P3
N	20	T2		P1
N	30	T3		P3
20	40	T4		P2

D.P.T

PageID	recLSN
P1	20
P2	40
P3	10

T.T

PAGE No.	Last LSN
DATE T1	10
T2	40
T3	30

recLSN → 13 always first

To Undo {40, 30, 20}

	Pre LSN	LSN	~~TT~~	~~Upd~~		To Undo = {10, ~~40~~, 30} 20
	N	10	T1	update	P3	{10, 20, ~~30~~}
60 CLR: undo T2	N	20	T2	update	P1	{10, ~~20~~}
LSN 10	N	30	T3	update	P3	{10}
65 T2 end						
	20	40	T2	update	P2	

50 CLR: undo T2 LSN 40
60,65 CLR: undo T3 LSN 30, T3 end;
70,75 CLR: undo T2 LSN 20, T2 end;
80,85 CLR: undo T1 LSN 10, T1 end;

CLR

- Compensation Log Record
- Created only while undoing any action
- CLRs has its own LSN
- Format:

  50,55 CLR: undo T1 LSN 10, T1 end

Unit 3 → Transaction Management; Trash Recovery

PAGE No.
DATE / /

LSN	LOG
00,05	Begin checkpoint, end checkpoint
10	Update: T1 writes p5
20	Update: T2 writes p3
30	T1 abort
40,45	CLR: undo T1 LSN 10, T1 end
50	Update: T3 writes p1
60	Update: T2 writes p5
	crash. Restart
70	CLR: undo T2 LSN 60
80,85	CLR: undo T3 LSN 50, T3 end
90,95	CLR: undo T2 LSN 20, T2 end

UndoNextLSN

nit3

ynamic
el

Dynamic SQL

NDS
PAGE → PL/SQL allows
DATE SQL statements

Dynamic SQL

You use dynamic SQL to create a SQL statement whose text is not completely known in advance.
Dynamic SQL:

- Is constructed and stored as a character string within a subprogram
- Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables).
- Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL.

Unit 2 → Native Dynamic SQL
— To create

- PL/SQL does not support DDL (Data Definition Language) statements written directly in a program.
- Native Dynamic SQL (NDS) allows you to work around this by constructing and storing SQL as a character string within a subprogram.
- NDS:
  - Provides native support for Dynamic SQL directly in the PL/SQL language.
  - Enable data-definition, data-control, or session-control statements to be written and executed from PL/SQL.

NDS:
- Is executed with Native Dynamic SQL statements (EXECUTE IMMEDIATE) or the DBMS_SQL package.

using holds all bind arguments.

PAGE No.
DATE / /

- Provides the ability to execute SQL statements whose structure is unknown until execution time.
- Can also use the OPEN-FOR, FETCH, and CLOSE PL/SQL statements.

Using the EXECUTE IMMEDIATE

- Use the EXECUTE IMMEDIATE statements for NDS in PL/SQL anonymous blocks or subprograms:

```
EXECUTE IMMEDIATE dynamic_string
[into {define_variable
      [, define_variable] ... | record}]
[USING [IN|OUT|IN OUT] bind_argument
      [, [IN|OUT|IN OUT] bind_argument] ...];
```

- INTO is used for single-row queries and specifies the variables or records into which column values are retrieved.
- USING holds all bind arguments.
- The default parameter mode is IN, if not specified.

- dynamic_string is a character variable or literal containing the text of a SQL statement.
- define_variable is a PL/SQL variable that stores a selected column value.
- ~~define~~ record is a user-defined or %ROWTYPE record that stores a selected row.

[ROWCOUNT → previous SQL statement count]

Sep (last week → Prac)
1st 2nd → syballus complete

PAGE No.
DATE

Diff btwn Immediate/BCL Package → [illegible]

Example 1: Dynamic SQL with a DDL Statement

- Constructing the dynamic statement in-line: with placeholder

```
CREATE PROCEDURE drop_any_table (p_table_name
VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE' ||p_table_name;
END;
```

without placeholder

- Constructing the dynamic statement in a variable:

```
Create procedure drop_any_table(p_table_name varchar2)
IS v_dynamic_stmt VARCHAR2(50);
BEGIN
   v_dynamic_stmt := 'DROP TABLE ' || p_table_name;
   EXECUTE IMMEDIATE v_dynamic_stmt;
END;
```

```
BEGIN drop_any_table('EMPLOYEE_NAMES'); END;
```

Example 2:
- Deleting all the rows from any table and returning a count:

```
Create Function del_rows (p_table_name varchar2)
return number is
begin
   execute immediate 'delete from ' || p_table_name;
   return SQL%ROWCOUNT;
end;
```

PAGE No.
DATE / /

- Invoking the function:

```
Declare
    v_count NUMBER;
Begin
    v_count := del_rows('EMPLOYEE_NAMES');
    DBMS_OUTPUT.PUT_LINE(v_count || ' rows deleted.');
End;
```

Example 3:
- Here is an example of inserting a row into a table with two columns and invoking the procedures.
- Note the use of escape single quotes.

```
Create procedure add_row(p_table_name VARCHAR2,
    p_id number, p_name varchar2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO' || p_table_name ||
      'VALUES(' || p_id || ', ''' || p_name || ''')';
END;
```

```
BEGIN
  add_row('EMPLOYEE_NAMES', 250, 'Chang');
END;
```

Using the DBMS_SQL Package

Some of the procedures and functions of the DBMS_SQL package are:

- OPEN_CURSOR
- PARSE
- BIND_VARIABLE
- EXECUTE
- FETCH_ROWS
- CLOSE_CURSOR

Example 1: Example of deleting rows

```
Create or replace function del_rows
 (p_table_name VARCHAR2) Return number IS
  v_csr_id      INTEGER;
  v_rows_del    NUMBER;
Begin
  v_csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE (v_csr_id,
     'DELETE FROM ' || p_table_name, DBMS_SQL.NATIVE);
  v_rows_del := DBMS_SQL.EXECUTE (v_csr_id);
  DBMS_SQL.CLOSE_CURSOR (v_csr_id);
  RETURN v_rows_del;
END;
```

- Compare this with the del_rows function earlier in this
- They are functionally identical, but which is simpler?

PAGE No.
DATE / /

SQL Injection

- SQL injection is a code injection technique that might destroy your database.
- SQL injection is one of the most common web hacking techniques.
- SQL injection is the placement of malicious code in SQL statements, via web page input.

SQL in Web Pages

SQL injection usually occurs when you ask a user for input, like their username/uaterid, and instead of a name/id, the user gives you an SQL statement that you will unknowingly run on your database.

Look at the following example which creates a SELECT statement by adding a variable ~~(USER~~ (txtUserId) to a select string. The variable is fetched from user input (getRequestString);

Example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = "+
txtUserId;
```

SQL Injection Based on 1 = 1 is Always True

Look at the example above again. The original purpose of the code was to create an SQL statement to

Batch Statements → Many SQL statements (separated by ;)

@parameter value(@0, @1, @2);

PAGE No. 1
DATE / /

Select a user, with a given user Id.
If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

UserId: 105 OR 1=1;

Then, the SQL statement will look like this:

Select * From Users where UserId = 105 OR 1=1;

SQL Injection Based on Batched SQL Statements

- Most databases support batched SQL statement.
- A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.
- The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

Example.

Select * from Users; DROP TABLE Suppliers;

Eg: txtUserId = getRequestString("UserId")
txtSQL = SELECT * FROM Users Where UserId = "

collection User id: 105; Drop Table Suppliers;

The valid sql statement would look like this:

Result SELECT * From Users where UserId = 105; Drop Table Suppliers;

Unit 3

Triggers, Packages

PL/SQL Packages

- PL/SQL package is a group of related functions, procedures, types, cursors, etc.
- PL/SQL package is like a library once written stored in the Oracle database and can be used by many applications.
- A PL/SQL package has two parts: package specification and package body.
- A package specification is the public interface of your applications. The public means the stored function, procedures, types, etc., are accessible from other applications.
- A package body contains the code that implements the package specification.

Package Specification

```
1 Create [Or Replace] Package package_name
2 [ AUTHID { Current_User | Definer}]
3 { IS | AS }
4
5 [definitions of public TYPES
6 , declarations of public variables, types, and objects
7. , declarations of exceptions
8. , declarations of cursors, procedures, and functions
9. , headers of procedures and functions]
10. END [package_name];
```

Example: Creating a new package specification called personnel. The personnel package contains two functions: get_fullname() and get_salary() based on employee's ID.

Database -> Shared ;

AuthorId

Explicitly defining the packages (Eg:- ~~we've diff notes here~~)

```
Create or replace package personnel AS
--get employee's fullname
Function get_fullname(n_emp_id NUMBER)
    return varchar2;
--get employee's salary
function get_salary(n_emp_id NUMBER)
    return number;
End personnel;
```

Create or replace only if we create new/ones

Package Body

If already there in PL/SQL package no need to specify

```
CREATE [Or replace] Package Body package_name
  { IS|AS }
   [definitions of private TYPES.
   ,declarations of private variables, types, and objects
   , full definition of cursors
   , full definition of procedures and functions]
[BEGIN
   sequence_of_statements
[Exception
    exception_handlers]]
End [package_name];
```

Example:

```
Create or replace package body personnel AS
  --get employee's fullname
Function get_fullname(n_emp_id NUMBER) Return VARCHAR2
IS
    v_fullname VARCHAR2(46);
BEGIN
Select first_name || ' ' || last_name into v_fullname
from employees where employee_id = n_emp_id;
return v_fullname;
```

PAGE No.
DATE / /

```
Exception
When NO_DATA_FOUND then return null;
When TOO_MANY_ROWS then return null;
end; --end get_fullname
--get salary

Function get_salary (n_emp_id NUMBER) Return number IS
n_salary NUMBER (8,2)
BEGIN
SELECT salary into n_salary from employees
where employee_id = n_emp_id;
return n_salary;

Exception
When NO_DATA_FOUND THEN RETURN NULL;
When TOO_MANY_ROWS THEN RETURN NULL;
end;

END personnel;
```

Example →

```
Declare
  n_salary Number (8,2);
  v_name varchar2(46);
  n_emp_id number;
Begin
v_name := personnel.get_fullname(n_emp_id);
n_salary := personnel.get_salary(n_emp_id);
If v_name is not null and
   n_salary is not null
then
dbms_output.put_line('Employee: '||v_name);
dbms_output.put_line ('Salary: '||n_salary);
end if;
end;
```

NULL compare done always T S
↳

TRIGGER (↳ does need any human command)

- What is Trigger?
  - Oracle engine invokes automatically whenever a specified event occurs.
  - Trigger is stored into database and invoked repeatedly, when specific condition match.

- Component of Trigger:-

  - Triggering SQL statements
    SQL DML (Insert, Update and Delete) statement that execute and implicitly called trigger to execute.

  - Triggering Restriction:
    We can specify the condition inside trigger to when trigger is to be fixed.

  - Trigger, Action:
    When the triggering SQL statements is executed, trigger automatically calls trigger action.

- Types of Trigger :-

1.) BEFORE Trigger

→ Before trigger execute before the triggering DML
→ Statement (Insert, Update, Delete) execute.
→ Triggering SQL statement is may or may not execute, depending on the BEFORE trigger conditions block.

PAGE No.
DATE / /

2.) AFTER Trigger

→ After trigger execute after the trigerring DML.
→ (Insert, Update, Delete) executed, Triggering SQL statement is executed as soon as followed by the code of trigger before performing database operation.

3.) Row-level triggers for data-related activities

→ Row-level triggers execute once for each row in a transition.
→ Check individual values and can update them.
→ Have access to :new and :old vectors.
→ Row-level triggers is identified by the FOR EACH ROW clause in the CREATE TRIGGER command.

4.) Statement level triggers for transaction-related activities

→ Statement-level triggers execute once for each transaction. For example, if a single transaction nested so rows into the customer table, then a statement-level trigger on that table would only be executed once.
→ Do not have access to :new or :old vectors (only for row level)
→ Execute once for the entire statement regardless how many records are affected.
→ Statement-level triggers are the default type of triggers created and or identified by emitting the FOR EACH ROW clause in the CREATE TRIGGER command.

DATE / /

5) Combination Trigger:

Combination trigger are combination of two trigger types.

1. Before Statement trigger:
Trigger fire only once for each statement before the triggering DML statement.

2. Before Row Trigger:
Trigger fire for each and every record before the trigger DML statement.

3. After Statement trigger:
Trigger fire only once for each statement after the trigerring DML execution.

4. After Row Trigger:
Trigger fire for each and every record after the trigerring DML ~~stat~~ execution.

PAGE No.
DATE 30/08/2022

```
Create [or replace] trigger trigger_name
{ Before | After | Instead of }
{ Insert [OR] | Update [OR] | DELETE }
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
When (condition)
Begin
--- sql statements
end;
```

Syntax Trigger Description

- CREATE [OR REPLACE] TRIGGER trigger_name :- This clause creates a trigger with the given name or overwrites an existing trigger with the same name.

- {BEFORE | AFTER | INSTEAD OF } - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.

- {INSERT [OR] | UPDATE [OR] | DELETE} - This clause determines the triggering event. More than one triggering events can be used together separated by OR Keyword. The trigger gets fired at all specified triggering event.

- [OF col_name] - This clause is used with update triggers. This clause is used when you want to

PAGE No.
DATE / /

trigger an event only when a specific column is updated.

- [REFERENCING OLD AS o NEW AS n]- This clause is used to reference the old and new values of the data being changed. By default, you reference the values as: old.column name or :new.column name. The reference names can also be changed from old (or new) to any other user-defined names. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.

- [FOR EACH ROW]- This clause is used to determine whether a trigger must fire when each row gets affected (i.e a Row Level Trigger)

- WHEN (condition) - This clause is valid only for row level triggers. The triggers is fired only for rows that satisfy the condition specified.

DATE / /

PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

1.) BEFORE statement trigger fires first.
2.) Next BEFORE trigger fires first.
3.) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.
4.) Finally the AFTER statement level trigger fires.

Commands

- DROP TRIGGER trigger_name
- ALTER TRIGGER trigger_name [{DISABLE| ENABLE}]
- ALTER TABLE table_name [{DISABLE |ENABLE}. [ALL TRIGGERS]]

PL/SQL Triggers Example

This trigger execute BEFORE to convert ename field lowercase to uppercase.

```
CREATE OR REPLACE TRIGGER trg1
    BEFORE
    INSERT ON emp1
    FOR EACH row
BEGIN
    : new.ename := upper(:new.ename);
END;
```

for each row

Deleting Trigger

This trigger is preventing to deleting row

Delete Trig

```
CREATE OR REPLACE TRIGGER trg1
    BEFORE
    DELETE ON emp1
    FOR EACH ROW
BEGIN
    IF:old:eno = 1 THEN
        raise_application_error(-20015, 'You can't delete
                                this row');
    END IF;
END
/
```

Create a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between old values and new values.

```
Create or Replace Trigger display_salary_changes
Before delete or Insert or Update on customers
for each row
When (New.ID > 0)
Declare
   sal_diff number;
Begin
sal_diff := NEW.salary - :OLD.salary;
```

```
dbms_output.put_line ('Old salary ' || :OLD.salary);
dbms_output.put_line('New salary' || :NEW.salary);
dbms_output.put_line('Salary difference ' || sal_diff);
END;
```

→ Nested tables → key in my methods // → Array

Unit 2

Nested Tables

PAGE No. / DATE: Mon Day 5/09/22

Nested Tables in PL/SQL

- The nested tables are like a single column database table or 1-dimensional array where the array size is dynamic.
- A nested table can be held in a column of a database. It can also be used for manipulating SQL operations by joining tables.
- A nested table can have both dense and sparse collection characteristics which means any element can be deleted randomly (making it sparse) with the help of the DELETE procedure.
- ~~A nested tabl~~ Since the data is stored in the form of a table, it can be retrieved with the help of SELECT statements.

Differences between Array and Nested table are listed below:-

- The size of the nested tables can be increased dynamically. The upper limit of an array is fixed but it is not fixed for nested tables.
- The array has consecutive subscripts which makes it dense. However, a nested table is dense at the time of creation but becomes sparse once the elements are deleted in between.

dense → compact

sparse → left sth in btw

Syntax of the nested table:

TYPE <<type>> IS TABLE OF <<element>> [NOT NULL];

Here, 'type' is the type specifier. 'element' is the data type.

DATE / /

Code implementation with nested tables:

```
Declare
    Type subject is table of varchar2(15);
    Type teacher is table of varchar2(20);
    subjectnames subject;
    subjectteacher subject;
    summ integer;
Begin
 subjectnames := subject('PLSQL', 'SELENIUM', 'JMETER');
 subjectteacher := teacher('Sashi', 'Mala', 'Mukund');
 summ := subject teacher.count;
 dbms_output.put_line('Total Number of Teachers: '||summ);
 For i in 1..summ Loop
  dbms_output.put_line('Subject:'||subjectnames(i)||'Teacher:'
 || subjectteacher(i));
 end loop;
 END;
 /
```

Associative Array or Index-By Tables

- The index-by table is commonly called the associative array. In terms of structure, both the indexes by table and nested tables are similar and have subscript to access the elements.
- An associative array is represented by a Key-value pair.
- Each of the unique keys is used to identify the value in the array.
- The data type of the key can be a string or an integer defined while creating it.
- One difference between the index-by table and the

key → what type it is

PAGE No.
DATE / /

nested table is that the former cannot be stored in the column of the database but the nested table can be stored.

Syntax of the index-by table:

```
TYPE type IS TABLE OF element [NOT NULL]
INDEX BY [BINARY_INTEGER | PLS_INTEGER | VARCHAR2(size)];
INDEX BY key;
```

'element' is the data type.
'size' is the maximum no of elements. It is a positive integer.
'type' is the type specifier.

Code implementation with the index-by table:-

```
Declare
  TYPE age IS TABLE OF NUMBER INDEX BY VARCHAR2(25);
  age_empl age;
  employee varchar2(25);
Begin
    age_empl('Sam') := 30;
    age_empl('Venu') := 35;
    employee := age_empl.FIRST;
    WHILE employee IS NOT null LOOP
    dbms_output.put_line
    ('Employee name is '||employee||'and age is '||
      TO_CHAR(age_empl(employee)));
    employee := age_empl.NEXT(employee);
  END LOOP;
END;
/
```

Varrays → element → Datatype

PAGE No.
DATE / /

Varrays in PL/SQL

The Varray store a fixed number of elements and are like a one-dimensional array.

- However, the number of elements can be modified at runtime.
- It is a consecutive set of elements of similar data types.
- It can be stored in a database table can be handled with the help of SQL statements.

Syntax of Varray :-

```
TYPE <<type>> IS { VARRAY | VARYING ARRAY } (<<size>>)
OF <<element>> [NOT NULL];
```

Here,
'type' is the type specifier.
'element' is the data type.
'size' is the maximum number of elements in an array. It is a positive integer.

Varray Variables Declaration and Initialization

After creating a Varray, we can declare it in the way described below:

Syntax:

```
name type_n [:= type_n(...)];
```

Here,
'name' is the Varray variable name.
'type_n' is the type of Varray.
'type_n(...)' is the constructor of type Varray. The

argument lists are mentioned below
Eg: country_var. country;

The initializing Varray:

Syntax:
name type_n := type_n ();
This will initialize the variable with zero elements.
In order to populate elements in the varray variables, the syntax is:
name type_n := type_n(e1, e2, ...)

Accessing elements of Varray:

The elements can be accessed by the process described below:
name(m);
where m is the element index which starts with 1 and terminates with the maximum cou

Modifying the size of Varray:-

The size of Varray can be modified with the help of ALTER statement

Code implementation with ALTER
→ Create or Replace TYPE names as VARRAY (2) OF VARCHAR(5);

PAGE No.
DATE / /

Code implementation ~~of~~ with Varray:

```
Declare
    type countrynames IS VARRAY (3) OF VARCHAR2(50);
    type currency IS VARRAY(3) OF VARCHAR2(15);
    country countrynames;
    cur currency;
    addition integer;
Begin
    country := countrynames('INDIA', 'USA', 'UK');
    cur := currency('INR', 'DOLLAR', 'POUND');
    addition := country.count;
    dbms_output.put_line('Total Number of countries: '||addition);
FOR i in 1..addition LOOP
    dbms_output.put_line('Country: '||country(i)||'Currency:'
                        ||cur(i));
END LOOP;
END;
```

Collection Methods

Collection Methods

Extend(null) null

null treated first then stored it in

A variety of methods exist for collections, but not all are relevant for every collection type.

- EXIST(n) - Returns TRUE if the specified element exists.
- COUNT - Returns the number of elements in the collection.
- LIMIT - Returns the maximum number of elements for a VARRAY, or NULL for nested tables.
- FIRST - Returns the index of the first element in the collection.
- LAST - Returns the index of the last element in collection

(5, 2)

ABC ABC ABC

PAGE No.
DATE / /

- PRIOR(n) - Returns the index of element of the prior to the specified element.
- NEXT(n) - Returns the index of the next element after the specified element.
- EXTEND - Appends single null element to the collection
- EXTEND(n) - Appends n null elements to the collection.
- EXTEND(n1, n2) - Appends n1 copies of the n2th element to the collections.
- TRIM - Removes a single element from the end of the collection.
- TRIM(n) - Removes n elements from the end of the collection.
- DELETE - Removes a single element from the end of the collection.
- DELETE(n) - Removes all elements from the collection.
- DELETE(n1, n2) - Removes all elements from n1, n2 from the collection.