Sub Query (or) Nested query (or) Inner query

→ A query within another query is called a sub-query (or) a sub query.

ex: select ename from employee where sal = (select max(sal) from employee);
(outer query) (inner query)

→ A nested sub query is a select statement that is nested within another select statement.

→ SQL evaluates the inner query (or) sub query first (first).

Q:- Retrieve the emp.name who is having highest salary in emp.table

→ if the sub query returns a single row, then it is called single row sub query.

→ Here, we use '=' operator.

→ If the sub query returns multiple rows then it is called multi-row sub query.

→ we use sub queries like in, not in, exict, All, ANY, operators.

ex-1: o/p:-

ename
bbb

e.no	e.name	sal
1	aaa	200
2	bbb	900
3	ccc	800

ex-2: select ename from employee where sal in (select max(sal) from employee);

o/p:

ename
bbb
ddd

e.no.	e.name	sal
1	aaa	200
2	bbb	900
3	ccc	800
4	ddd	900

Q:- Retrieve ename and deptno from employee table which is having the 2nd highest salary.

```
select ename, deptno from employee where
sal = (select max(sal) from employee where
sal < (select max(sal) from emp));
```

syntax: select <columnlist> from <tablename> where <relational operator> any (select <columnlist> from <tablename> where <condn>);

co-related sub query:

→ A query is called co-related sub query when both the inner query and outer query are inter dependent.

→ An inner query depends on outer query before it can be processed.

sub query:-

relational operator ⇒ <, >, <=, >=, =, in, not in.

→ A subquery typically appears within the where clause of a query.

→ The first query in an sql statement is called outer query. The query inside SQL statement is known as inner query.

→ The inner query is executed first.

→ The o/p of inner query is used as i/p for mainquery.

Single row return:

→ If a sub query returns a single row then use single row return '=' operator

Multi row operators :-

in operator

→ is equal to any member in a list.

not in operator

→ not equal to any member in a list.

ANY operator:

→ returns rows that match any value in a list

ALL operator:

→ It returns rows that match all the values in a list.

Co-related sub query:

→ They are used for row by row processing.

→ each sub query is executed once for every row of the outer query.

→ In co-related nested query inner query executes everytime when outer query executes.

⇒ find the employees who earns the salary more than their avg. salary of their department.

emp

name	sal	dept
aaa	2000	CSE
bbb	3000	DS
ccc	8500	CSE
ddd	3500	DS

→ select name, from emp e where > (select sal,dept avg(sal) from emp where dept = e.dept);

output:

name	sal	dept
ccc	2500	CSE
ddd	3500	DS

group by clause:

→ These clauses will be used in select command to retrieve the data.

→ It groups the rows based on columns.

→ Here, we use aggregate functions.

Syntax:

select <columnlist> from <tablename> group by <columnlist>;

Having clause:

→ It is also used with aggregate functions.

→ Conditions on result of group by clause.

Syntax:

select <columnlist> from <tablename> group by <columnlist> having <condition>;

ex: emp

Sid	name	per	gender	Branch
101	hari	99	male	CSE
102	Ramya	90	female	ECE
103	Seetha	95	female	CSE
104	Ram	78	male	CSE
105	sony	88	female	CIVIL

⇒ find the no. of students branch wise.

> select branch, count(sid) from emp group by branch;

output:

Branch	sid
CSE	3
ECE	1
CIVIL	1

⇒ find the no. of female and male students

> select gender, count(sid) from emp group by gender;;

output:

gender	sid
male	2
female	3

⇒ find avg % of male and female students.

> select gender, avg(per) from emp group by gender;

output:

gender	per
male	93.5
female	91

⇒ find maximum % from male and female students.

> select gender, max(per) from emp group by gender;

output:

gender	per
male	99
female	75

⇒ finding branches having only 1 student

```
select branch, count(sid) from emp group by branch
having count(sid) = 1 ;
```

output:

branch	sid
ECE	1
CIVIL	1

JOINS:

→ The SQL Join clause is used to combine records from 2 (or) more tables in a db.

→ A join means combining fields from 2 tables by using values common to each.

→ There are different types of JOINS available in SQL

1. INNER JOIN:

→ It returns rows when there is a match in both tables.

Syntax:

```
select columns from <tablename1> inner join
    table2 on table1.column = table2.column;
```

2. LEFT OUTER JOIN:

→ It returns all rows from the left table even if there are no matches in the right table.

Syntax:

```
select columns from <tablename1> left [outer] join
    table2 on table1.column = table2.column;
```

3. RIGHT OUTER JOIN:

→ It returns all rows from the right table even

if there are no matches in the left table.

Syntax:

Select columns from <table1> right [outer] join table2 on table1.column = table2.column;

4. FULL JOIN:

→ It returns rows when there is a match in one of the tables.

5. EQUI JOIN:

Syntax:

select <columnlist> from table1, table2 where table1.<columnname> = table2.<column name>;

TCL (Transaction Control language)

→ TCL commands are used manage transactions in database.

1. commit 2. Rollback 3. Savepoint

commit;

→ It is used to permanently save a transaction.

→ Changes made by insert, delete, update statements are not permanent

emp

eid	ename	sal
101	aaa	200
102	bbb	500
103	ccc	600

> update emp set sal = 800 where eid = 1003;

> 1 row updated

> select * from tablename ;
emp

> commit ;
commit successfully.

Roll back : → It performs undo operation.
→ It restores the db to be committed state.
→ It is also used with save back.

Syntax: Rollback to savepoint ;

> update emp set sal = 800 where eid = 103 ;
1 row updated →

empid	ename	sal
101	aaa	200
102	bbb	500
103	ccc	800

> Roll back ;
> select * from emp ;
→

empid	ename	sal
101	aaa	200
102	bbb	500
103	ccc	600

Savepoint:
→ Used to temporarily save a transaction so that, we can rollback to that point.
Syntax: savepoint savepointname ;

Cursors :
→ After creation of table, table will be stored in system memory.
→ If we want to perform any operations on this table (insertion, deletion, updation) then cursor will be generated for the rows.
→ It will display records based on the

condition.

→ A cursor is a temporary work area created in a system memory when a SQL statement is executed.

→ A cursor contains information on a select statement and the rows of data accessed by it.

→ This temporary work area is used to store the data retrieved from db and manipulate the data.

→ A cursor can hold more than 1 row but can process only 1 row at a time.

→ The set of rows the cursor holds is called the 'active set'.

→ There are 2 types of cursors

1. Implicit cursor
2. Explicit cursor

Implicit cursor:

→ These are created by default when DML statements like insert, update and delete statements are executed.

→ They are also created when a select statement that returns just 1 row is executed.

→ These are pre-defined cursors defined by the Oracle software.

Attributes:

1. % found
2. % NOTFound

3. %RowCount

4. %ISOPEN

%Found:

→ It returns value is true if DML statements like insert, delete, update effects atleast one row (or) more rows (or) a select into statement return 1 (or) more rows.

→ Otherwise, it returns false.

%NotFound:

→ It returns value is true if DML statements like insert, delete, update effects no rows (or) a select into statements returns no rows.

→ otherwise, it returns false

→ It is just opposite of %Found.

%RowCount:

→ It returns the no. of rows effected by DML statements (or) return by select into statement

%ISOPEN

→ It always return false for implicit cursors because the SQL cursor is automatically closed after executing its associated SQL statements.

```
DECLARE
    VAR ROWS Number(5);
BEGIN
    UPDATE EMPLOY SET SAL = SAL + 5000;
IF SQL%NOTFOUND THEN
    DBMS_OUTPUT.PUTLINE ("NO rows updated");
ELSEIF SQL%FOUND THEN
    VAR_ROWS := SQL%ROWCOUNT;
    DBMS_OUTPUT.PUT_LINE(VAR_ROWS || 'updated
                                    records');
END IF;
END;
/
```

ouput:

```
SQL >SET SERVEROUTPUT ON;
SQL > @c:\SQL\ cursor.SQL;
107 updated records
PL/SQL procedure successfully completed.
```

Explicit cursor:

→ These cursors are user-defined cursor.
→ They are declared explicitly by the user along with the identifiers to be used.
→ They must be created when we are executing select statement that returns more than one row even though the cursor stores multiples row/ records.

→ only 1 record can be processed at a time which is called as current row.

→ When we fetch a row the current row position moves to next row.

→ Both implicit and explicit cursors have same functionality but they differ in the way they are accessed.

1. DECLARE
2. OPEN
3. FETCH
4. CLOSE

DECLARING:

→ We use the word cursor.

CURSOR <cursorname> IS <selectstatement>;

ex: DECLARE
CURSOR C1 IS select * from emp where sal > 5000;

OPEN:

OPEN <cursor_name>;

ex: OPEN C1;

FECTHING / retrieving:

FETCH <cursor_name> INTO <variables>;

ex: FETCH C1 INTO EMP_id, Emp_name;

CLOSING:

CLOSE <cursor_name>; ex: close C1;

```
DECLARE
    STDNO STUDENT.SNO%TYPE;
    // datatype of sno will be alloted to stdno.
    STDNAME STUDENT.SNAME%TYPE;
    CURSOR C2 IS SELECT SNO, SNAME FROM
                    STUDENT where SNO=123;

BEGIN
    OPEN C2;
LOOP
    FETCH C2 INTO STDNO, STDNAME;
    IF C2%FOUND THEN
    DBMS_OUTPUT_LINE('STDNO'||'    '||'STDNAME'||
                                    ' ');
ELSE
EXIT;
ENDIF;
END LOOP
CLOSE C2;
END;
/
```

Stored procedures :

→ Procedures are a named block and they can be reused.

(or)

→ A procedure is a module that perform one (or) more actions.

→ A procedure doesn't return any value but will pass some parameters.

→ procedures are stored in db dictionary.

→ procedures are similar to functions.
→ Difference between procedure and function is function returns a value while procedure perform action and it doesn't return any value.

Syntax: (creation):

```
CREATE or Replace procedure <procedure_name>
                                [parameter 1..
                                ....]

IS / AS
   // declare variables
BEGIN
      <executable statements>
END ;
```

execution:

```
Execute procedurename ;
Exec procedurename ;
```

Deletion:

```
Drop procedure procedurename;
```

Parameter modes:

→ There are 3 parameters to write a procedure.

1. IN parameter:
→ It is a default mode.
→ Read only parameter.

2. OUT parameter:
→ It returns a value to the calling program.

→ We can change its value.

3. INOUT parameter :

→ It passes initial value to a sub program and returns an updated value to the caller.

* Write a procedure for multiplication of two numbers.

```
create procedure mul
Declare
      a number;
      b number;
      c number;
procedure mul(x IN number, y IN number,
                        z OUT number)
IS
BEGIN
    z := x * y ;
END;
BEGIN
    a := 50;
    b := 10 ;
    mul(a, b, c)
dbms_output.put_line("multiplication is "||c);
END;
SQL > Execute mul ;
```

* Create a procedure for square of a number.

```
DECLARE
    a number;
procedure squarenum(x IN OUT parameter)
```

```
IS
BEGIN
   x := x * x ;
END ;
BEGIN
    a := 5
squarenum(a) ;
dbms_output.put_line('squarenum' || a);
END ;
```

* Create a procedure to view some specified columns from a table.

```
Declare
Create procedure p_sail (sid1 in number)
IS
V_Sname varchar(10) ;
V_age number(10);
Begin
   select sname, age into V_Sname, V_age from
          sailors where sid = sid1
   dbms_output.put_line('Sname' || V_Sname);
   dbms_output.put_line('age' || V_age);
End ;
/
> Execute p_sail(22)
Sname : Dustin
age: 45.0
procedure executed successfully.
```

ex: Create procedure selectallcus

```
BEGIN
    select * from student;
END;
```

> Execute selectallcus;

o/p:

sid	s.name	age

---

* Write a procedural language SQL code for modification of procedure to view specified columns from a table.

```
Create procedure p_sail (sid1 in number)
IS
V_Sname varchar(10);
V_age number(10);
Begin
    update psailo set sname = 'Susmitha', age = 18
    where sid = 22;
  dbms_output.put_line('sname' || V_sname)
  dbms_output.put_line('age' || V_age)
End;
/
```

> Execute p_sail(22);

S.name : Susmitha

age : 18

procedure executed successfully.

Triggers:

→ A Trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs.

→ ex:- A trigger can be invoked when the row is inserted in to a specified table (or) when certain table columns are being updated.

Syntax:

```
Create trigger [trigger_name]
[before/after]
{insert/delete/update}
on [table_name]
[for each row]
[trigger_body]
```

create trigger [trigger_name] :

→ It specifies create (or) replace an existing trigger with the trigger name.

[before/after]:

→ This specifies when the trigger will be executed.

{insert/delete/update}:

→ specifies DML operation

on [table_name]:

→ It specifies the name of the table associated with the trigger.

[for each row]:

→ It specifies a row level trigger i.e. the trigger will be executed for each row being effected.

[trigger_body]:

→ This provides the operation to be performed as trigger is fired.

Create a trigger to perform some calculations:

1. Create table student(id int, name varchar(10), sub1 int, sub2 int, sub3 int, total int, perc int);

2. Create trigger stu1
   before insert
   on student
   for each row
   set student.total = student.sub1 + student.sub2 + student.sub3;
   student.perc = student.total * 60/100;

3. Trigger execute

4. insert into student values(1, 'college', 20, 20, 20, 0);
   1 row affected

5. Select * from student;

id	name	sub1	sub2	sub3	total	perc
1	college	20	20	20	60	36

→ triggers are occured when we perform the follo-
-wing events :

i) Data Manipulation statements
ii) Data definition statements

→ triggers could be defined on the table, view, schema (or) db with which the event is associated.

Benefits of triggers :

→ triggers can be written for the following purposes.

1) generating some derived column values automatically
2) Enforcing referential integrity
3) Event logging and storing information on table access
4) auditing.
5) Synchronous replication of tables.
6) Imposing security authorizations.
7) Preventing invalid transactions.

Write a PL SQL code for creation of trigger to insert data into a table.

1. Create table student(id int, name varchar (10));

2. create trigger stu1
before insert
on student
for each row
begin

```
    sname := Upper(sname);
end;
```

Trigger created

```
insert into student values(101, 'cmrit');
```

1 row affected

```
select * from student;
```

id	name
101	CMRIT

* Write a PL SQL code for creation of trigger to update data into a table.

```
create table student(id int, name varchar(10));
create trigger std
after update of id
on student
for each row
begin
    if(sid.id < 105) then;
    raise_application_error(20080, 'can't update');
end if;
end;
/
```

Trigger created

```
Update student set id = 104 where name = 'cmrit';
```

Can't update

```
update student set id = 106 where name = 'cmrit'
```

1 row affected

```
select * from student;
```

id	name
106	cmrit

* Write a PL SQL code for creation of trigger to delete data from a table.

```
create table student (id int, name varchar(10));
create trigger std
after delete on student
for each row
begin
if( sid = 105) then ;
    raise_application_error(-20080, 'cant delete');
end if;
end;
/
Trigger created
delete from student where sid = 105;
                    display error
```

Cursors :

```
DECLARE
    pet_count NUMBER;
    CURSOR number_of_pets is
        SELECT CURRENT_INVENTORY_COUNT
        FROM PRODUCT
        WHERE PRODUCT_NAME = 'Canary';
BEGIN
    OPEN number_of_pets;
    FETCH number_of_pets INTO pet_count;
    DBMS_OUTPUT.PUT_LINE ('Number of Canary's =
                          || pet_count);
    CLOSE number_of_pets;
END;
```

```
DECLARE
    var_1 employee.emp_name%TYPE;
    var_2 employee.salary%TYPE;
        CURSOR C1 IS SELECT emp_name, salary from
                employee;
BEGIN
  OPEN C1;
  FETCH C1 INTO var_1, var_2;
  DBMS_OUTPUT.PUT_LINE (var_1 || 'Salary is Rs.' ||
                                  var_2);
  CLOSE C1;
END;
/
```

Susmitha salaray is Rs. 1,00,000/-

Triggers:

```
CREATE OR REPLACE TRIGGER encryptInserts
      BEFORE INSERT ON EMPLOYEES
      FOR EACH ROW
DECLARE
      input VARCHAR2(100);
      pass VARCHAR2(100);
      output VARCHAR2(100);
BEGIN
      pass := 'flamingo';
      input := :NEW.LAST_NAME;
      output := enc_tp.Caseii(input, pass);
      :NEW.LAST_NAME := output;

      input := NEW.PHONE_NUMBER;
      output := enc_tp_alphanum(input, pass);
      :NEW.PHONENUMBER := output;
END;
/
```

Stored procedure:

```
CREATE PROCEDURE GetloginDetails
    @studentname nvarchar(30),
    @ loginID  nvarchar(30) OUTPUT

AS
BEGIN
   SELECT @ loginID = Login
   FROM [DBO] - [USER]
   WHERE name = @studentname

END;

$
```

unit-4 :

Schema refinement and Normal forms

→ Introduction of S.R and functional dependencies
→ reasoning about FD's
→ Normalization
→ Normal forms - 1NF
2NF
3NF
BCNF
→ Multi-valued dependencies - 4NF
→ joint dependency - 5NF
→ properties of decomposition, dependency preservation.

Schema refinement:

→ It is a technique to organize data in a db.
→ It is a systematic approach of decomposing table to eliminate data redundancy and undesirable characteristics (anomolies):
insert, delete & update
→ Schema refinement refers to refine the schema by using some techniques.
→ The best technique is decomposition.
→ Another method to eliminate redundancy is Normalization.
→ Identify and clearing the future problems in db is called Schema refinement.
→ Normalization is multi-step process that puts data in tabular form and removes duplicate data from relational table.

Data redundancy:

→ storing the same information repeatedly i.e. in more than 1 place with in db is called data redundancy.

Anamolies:

→ The problems caused by data redundancy.

Insertion Anomolies:

→ They occur during insertion of data into a table.

→ It occurs in a relation db when some attributes (or) data items are inserted into a db without existing of other attributes.

Deletion Anomolies:

→ It occurs when deleting one part of the data deletes other necessary information from the db.

Updation Anomolies:

→ It occurs when same data items are repeated with the same values are not linked to each other.

→ If one copy of such repeated data is updated and inconsistency is created unless all copies are similarly updated.

Functional dependency: (FD)

→ One (or) more attributes of a relation (or) table are dependent on one (or) more attributes of same table.

→ Functional dependency is denoted by '→'

ex: A → B

A determines B

↓ ↓

Determinant dependent

⇓

B dependent on A.

Sid	Sname	course
72	Ramesh	c
72	Ramesh	java
73	Mahesh	python
74	Mahesh	python
75	Naresh	java

if $t_1.A = t_2.A$

then $t_1.B = t_2.B$

Sid → sname ✓

Sid → course ✗

Sname → course ✓

(Sid, sname) → course ✗

Types of Functional dependency:

→ There are 4 types of FD's. They are:

1. Trivial FD
2. Non-trivial FD
3. Multi-valued FD
4. transitive FD.

Trivial FD :

→ $A \rightarrow B$ is functional dependent if B is a subset of A.

ex: sid, sname → sid

Non-Trivial FD :

→ $A \rightarrow B$ is functional dependent if B is not a subset of A.

ex: sid, sname → Age

Multi-valued FD :

→ $A \rightarrow BC$ is functional dependent then B & C should not be dependent.

$B \rightarrow C$ } not FD.
$C \rightarrow B$ }

ex: Sid → Sname, age ✓

Sname → age ✗

age → sname ✗

Transitive FD :

→ $A \rightarrow B$ is a functional dependent, B → x is functional dependent then A → C is functional dependent.

ex:

Sid	Sname	age
101	aaa	24
101	aaa	24

Sid → Sname FD ✓

Sname → age FD ✓

Properties of FD's (or) Armstrong Axioms

- Reflexivity
- Augmentation
- Transitivity

Reflexivity :

→ If A – set of attributes , B – subset of A
then A → B is functional dependent

Augmentation :

→ if A → B is FD
if C is attributes (or) set of attributes
AC → BC is also FD.

Transitivity :

→ if A → B is FD
B → C is FD
then A → C is also FD.

Inference rules derived from Armstrong Axioms :

- Union
- composition
- Decomposition (dividing)
- pseudo transitivity

Union

if
A → B is FD
A → C is FD
then
A → BC is also FD.

composition

if
A → B is FD
C → D is FD
then
AC → BC is FD

Decomposition (dividing)

if
A → BC is FD
then
A → B is FD
A → C is FD

pseudo transitivity

if
A → B is FD
BC → D is FD
then,
AC → D is also FD.

Finding the closure of an attribute:

closure:

→ It is a set of attributes that are functionally determined by attribute set.

→ We can apply closure for single attribute (or) set of attributes.

→ we can find super keys and Candidate keys.

→ It is represented by $+$

→ closure of $X = (X)^+ = \{ \ldots \}$

ex: $R = (A, B, C, D, E)$

F.D's

$A \rightarrow B$

$B \rightarrow C$

$C \rightarrow D$

$D \rightarrow A$

closure of $A = (A)^+ = \{A, B, C, D\}$

closure of $B = (B)^+ = \{B, C, D, A\}$

closure of $C = (C)^+ = \{C, D, A, B\}$

closure of $AB = (AB)^+ = \{A, B, C, D\}$

closure of $ABC = (ABC)^+ = \{A, B, C, D\}$

closure of $E = (E)^+ = \{E\}$

ex: $R = (A, B, C, D, E, F, G)$

FD:

$A \rightarrow BC \Rightarrow A \rightarrow B$

$A \rightarrow C$

$BC \rightarrow DE \Rightarrow BC \rightarrow D$

$BC \rightarrow E$

$D \rightarrow F$

$CF \rightarrow G$

closure of A = $(A)^+$ = {A, B, C, D, E, F, G} ⇒ Super key.

Resultant set
(it contains all attributes)

closure of B = $(B)^+$ = {B, C, D, E, F, G}

closure of D = $(D)^+$ = {D, F}

closure of BCD = $(BCD)^+$ = {B, C, D, E, F, G}

Super Key:

→ If closure of a single attribute (or) set of attributes determines all the attributes of the relation.

→ With the help of FD's, we are going to find out the closure and if the closure contains all the attributes of a relation then we call it as a super key.

Candidate Key:

→ We have to find the super key.

→ We need to find the minimal set of super key i.e candidate key.

→ We have to check whether a relation is having one (or) multiple candidate keys.

A → B

B → C

closure of (A-BC) = $(A\text{-}BC)^+$ = {A, B, C} ⇒ Super key

closure of (A-B) = $(AB)^+$ = {A, B, C} ⇒ super key

closure of (A) = $(A)^+$ = {A, B, C} ⇒ Super key

↓

Candidate key.

R = (A, B, C, D, E)

FD: $A \rightarrow B$
$C \rightarrow D$
$D \rightarrow E$

closure of ABCDE $= (ABCDE)^+ = \{A, B, C, D, E\}$

closure of (ACDE) $= (ACDE)^+ = \{A, B, C, D, E\}$

closure of (ADE) $= (ADE)^+ = \{A, B, D, E\}$

closure of AE $= (AE)^+ = \{A, B, E\}$

closure of A $= (A)^+ = \{A, B\}$

closure of E $= (E^+) = \{E\}$

Normalization:

→ It is a process of organizing the data in db.

→ It is used to minimize redundancy from a relation (or) set of relations.

→ It is also used to eliminate undesirable characteristics like insertion, deletion and updation.

→ It is used to reduce redundancy from the db table.

Different phases of Normalization:

Normal forms	Description
1NF	A relation is in NF, if it contains a atomic value ↓ It cannot be divided further.
2NF	A relation will be in 2NF, if it is in 1NF and all non-key attributes are fully functional dependent on primary key.

3NF → A relation will be in 3NF, if it is in 2NF and no transition dependency exists.

BCNF → A stronger definition of 3NF is known as BCNF
↓
Boyce codd

4NF → A relation will be in 4NF, if it is in BCNF and has no multi-valued dependencies.

5NF → A relation is in 5NF, if it is in 4NF and doesn't contain any joint dependency and joining should be loss less.

Advantages :

→ It helps to minimize data redundancy
→ Greater overall db organization.
→ Data consistency within the db.
→ Much more flexible db design.
→ Enforce the concept of relational integrity.

Disadvantages :

→ We cannot start building the db before knowing what the user needs.
→ The performance degrades when normalizing the relations to higher NF's i.e 4NF and 5NF.
→ It is time consuming and difficult to normalize relations of a higher degree
→ Careless decomposition may lead to a bad db design leading to serious problems.

1NF:

→ A relation will be in 1NF, if it contains atomic values.

→ It states that an attribute of a table cannot hold multiple values, it must hold any single value attribute.

→ 1NF disallows multi-valued attribute, composite attribute and their combination.

ex: Relation with unnormalized data.

Sid	Sname	Course
101	sai	c, c++, java
102	Ram	c++, python
103	seetha	java, python

↓ INF

Sid	Sname	Course
101	sai	c
101	sai	c++
101	sai	java
102	Ram	c++
102	Ram	python
103	seetha	java
103	seetha	python

2NF:

→ In 2NF, relation must be in 1NF

→ All non-key attributes are fully functional dependent on the primary key i.e. No partial dependency.

Full functional dependency:

$X \longrightarrow Y$ is full FD

i.e. we remove any attribute of X violates the FD rule.

ex: $(A, B) \longrightarrow C$ is FD

$A \longrightarrow C$ (not FD)

$B \longrightarrow C$ (not FD)

Partial functional dependency:

$X \longrightarrow Y$ is partial FD

i.e. we remove any attribute of X not violates the FD rule.

ex: $(A, B) \longrightarrow C$ is FD

$A \longrightarrow C$ is FD

$B \longrightarrow C$ is FD

Let's assume a school can store the data of teachers, subjects they teach.

T.id	subject	T.age
25	chem	30
25	Bio	30
47	eng	35
83	math	38
83	comp	38

In the given table, non-prime attributes T.age is dependent on T.id which is a proper subset of candidate key.

Thats why it violates the rule of 2NF

→ to convert the given table in to 2NF we decompose this table into 2 tables.

T.id	T.age
25	30
47	35
83	38

T.id	subject
25	chem
25	Bio
47	eng
83	math
83	comp

3NF:

→ A table will be in 3NF, if it is in 2NF and no transitive dependency for non-prime attributes.

ex:

name	roll.no	branch	fees
dustin	21	CSE	80,000
Lubber	22	DS	20,000
Charlie	23	AIML	10,000
Horatio	24	ECE	5,000

→ 1 non-primary key attribute fees is dependent on one more non-primary key attribute branch.

→ Here, there is transitive dependency.

→ In order to remove transitive dependency, we need to remove any of the column from the table and create another table.

name	roll.no	branch
dustin	21	CSE
lubber	22	DS
Charlie	23	AIML
Horatio	24	ECE

branch	fees
CSE	80,000
DS	20,000
AIML	10,000
ECE	5,000

→ 1st table has no partial dependency, so it is in 3NF.

→ 2nd table is not in 3NF.

BCNF:

Boycee Codd Normal Form :

→ BCNF is an extension of 3NF.

→ It is also known as 3.5NF.

→ BCNF is the advanced version of 3NF.

→ It is stricter than 3NF.

→ It should be in 3NF.

→ for any dependency A → B } conditions.

A should be a super key.

→ If B column (or) B attribute is dependent on attribute A then attribute A must be a super key → primary key + no duplicates → superkey.

roll.no	s.name	B.id	B.name
1	dustin	121	CSE
2	Lubber	122	AIML
3	charlie	123	DS
4	Horatio	121	CSE

roll no	s.name
1	dustin
2	Lubber
3	charlie
4	Horatio

↓ superkey

Bid	Bname
121	CSE
122	AIML
123	DS

↓ superkey

4NF:

→ A relation will be in 4NF, if it is in BCNF and has no multi-valued dependency.

2 non-primary key attributes are dependent on single primary key attribute.

sid	course	hobby
1	CSE	hackathon
2	DS	dancing
3	AIML	collegebunk

→ hobby and course are dependent on 1 primary key attribute i.e sid.

⇓

s.id	course
1	CSE
2	DS
3	AIML

s.id	hobby
1	hackathon
2	dancing
3	collegebunk

5NF:

→ A relation is in 5NF, if it is in 4NF and doesn't consists any join dependency.

→ 5NF is satisfied when all the tables present in dbms are broken down into as many more tables to ensure there is no redundancy.

subject	faculty	year
c	dustin	1
c	Horatio	1
java	Lubber	2
DBMS	charlie	3

Subject	faculty
C	dustin
C	Horatio
java	lubber
DBMS	charlie

faculty	year
dustin	1
Horatio	1
lubber	2
charlie	3

subject	year
C	1
java	2
DBMS	3

1NF : Atomic values

2NF : 1NF + No partial dependency

3NF : 2NF + No transitive dependency

BCNF: 3NF + superkey

4NF : BCNF + No multi-valued dependency

5NF : 4NF + No join dependency

Decomposition :

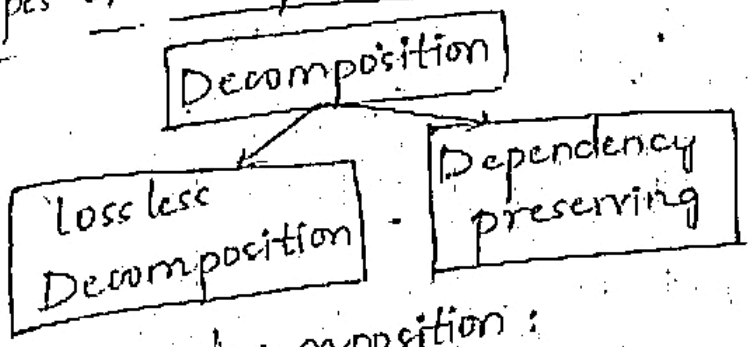
→ It means dividing / breaking the tables in to multiple tables in order to avoid redundancy to eliminate anamolies like insertion, deletion and updation and data inconsistency.

Properties of decomposition :

→ When the relation in the relational model is not in appropriate NF, then decomposition of the relation is required.

→ When the relation has no proper decomposition then it may lead to problems like loss of information.

Types of decomposition:



Loss less decomposition:

→ If the information is not lost from the relation that is decomposed then the decomposition will be loss less.

→ The loss less decomposition guarantees that the join of relations will result in the same relation as it was decomposed.

→ The relation is said to be loss less decomposition if natural joins of all the decomposition gives the original relation.

emp-dept details:

eid	ename	eage	ecity	D.id	D.name
22	Dustin	32	Hyd	E001	DS
23	Charlie	33	Banglore	E002	AIML

→ The above relation is decomposed in to 2 relations

eid	ename	eaage	ecity
22	Dustin	32	hyd
23	charlie	33	Banglore

D.id	e.id	D.name
E001	22	DS
E002	23	AIML

→ We can join this 2 tables with common attribute eid then the resultant relation will look like | emp ⋈ Department

Lossy decomposition:

e.id	ename	e.age	e.city
22	Dustin	32	hyd
23	charlie	33	banglore

D.id	D.name
E001	DS
E002	AIML

→ Here, we are not having any common attribute. So, we cannot join this two table
→ Hence, the above relation has lossy decomposition.

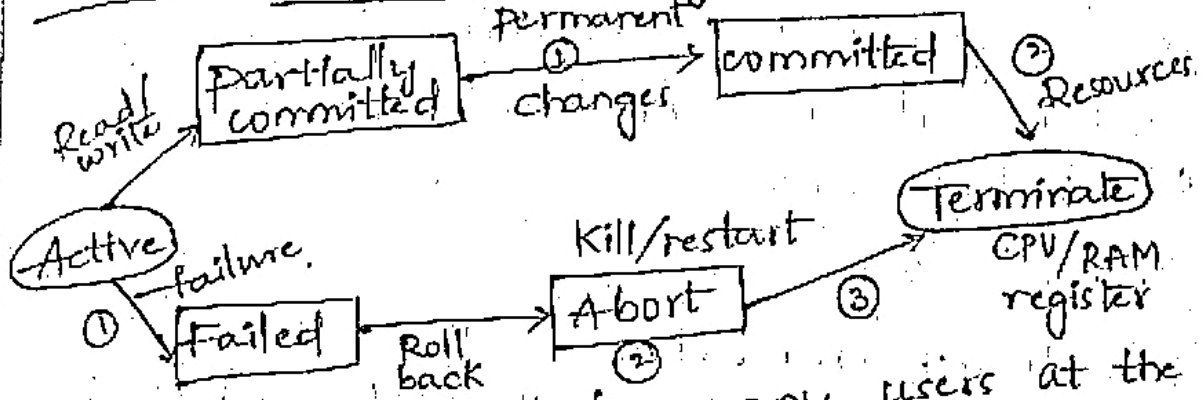
Dependency preserving:

→ It is an important constraint of the db.
→ In the dependency preservation, atleast one decomposed table must satisfy every dependency
→ If a relation 'R' is decomposed in to relations R_1 and R_2. Then, the dependencies of R, either must be a part of R_1 (or) R_2 (or) must be derivable from the combination of functional dependencies of R_1 and R_2.
→ Suppose, there is a relation R with attributes A, B, C, D with functional dependencies, R (A, B, C, D) with FD

$$A \longrightarrow BC$$

The relation R is decomposed in to $R_1(A, B, C)$ and $R_2(A, D)$ which is dependency preserving because FD $A \longrightarrow BC$ is a part of relation R_1 with attributes A, B, C.

Unit-5: Transaction Management



→ A database is used by many users at the same time as it is a shared resource.

→ ex: IRCTC Website is used by millions of users at the same time

→ Many people are concurrently using and booking tickets as such, as a large db is always consistent and concurrently managed.

Active state:

→ It is the 1st state of every transaction.

→ In this state the transaction is being executed.

Partially committed state:

→ Here, a transaction executes its final operation but the data is still not saved to db.

Committed state:

→ Transaction has executed all its operations Successfully.

→ In this state all the effects are now permanently saved in the database.

Failed state:

→ If any checks made by the db recovery system fails then the transaction is said to be failed state.

Aborted state:

→ If the db transaction is in failed state, the db recovery system check the db in its previous consistent state.

→ If not, it will abort (or) roll back the transaction to bring the db to consistent state.

→ After aborting the transaction, the db recovery module will select one of the 2 operations.

1. to restart the transaction
2. to kill the transaction

Operations of transactions:

1) Read(X):

→ Read operation is used to read the value of 'X' from the db and store it in main memory.

2) Write(X):

→ It is used to write the value back to the db from the buffer.

Debit transaction :

Assume X = 40,000

1. R(X) ; // Reads value of X from db to buffer
2. X = X - 500 ; // decrease value of X by 500
   ⇒ Buffer = 3500
3. W(X) ; // write buffer value to db ⇒ X = 3500

→ If there is a failure of hardware, software (or) power etc, the transaction may fail before finishing all the operations in the set.

→ To solve this problem, we have 2 important operations :

1. commit
2. rollback.

commit :

→ It is used to save the work done permanently.

rollback :

→ It is used to undo the work done

→ There are various properties of transactions occuring in the db are managed. They are called "ACID properties".

ACID → Durability, Isolation, consistency, Atomicity

Atomicity :

→ Any partial updates carried out either partial happen (or) not happen at all.

Consistency:

→ It ensures db property change its state after successfully committed.

→ It means before the transaction started and after the transaction completed the sum of money should be same

A $\xrightarrow{1000}$ B

A = 2000
B = 3000
} A+B = 5000 ↓ Before transaction

T_1	T_2
Read (A)	Read (B)
A = A − 1000	B = B + 1000
Write (A)	write (B)
A = 1000	B = 4000

} A+B = 5000 ↓ after transaction

Isolation:

→ It enables transaction to operate independently of transparent to each other.

Durability:

→ Ensure that the result (or effect of committed transaction persists incase of system failure.

→ Any changes made in db should be permanent.

*How to implement atomicity and durability in transaction?

Recovery mechanism component of DBMS:

→ It supports Atomicity and durability.

Assumption: only one transaction is active at a time

Shadow copy schema:

1. Based on making copies of db.
2. Assumption only one transaction at a time.
   Assume
3. A pointer called DB-pointer is maintained on the disk which at any instance points to the current copy of db.
4. Transaction wants to update. DB first creates a complete copy of db
   new
5. All further updates are done on new db copy, leaving the original copy untouched.
6. If at any point the transaction has to be aborted then the system deletes the new copy and the old copy is not effected.
7. If transaction is succeded, it is committed as,
   a) Operating system make sure all the pages of new copy of db written on disk.
   b) db system updates the DB-pointer to point to the new copy of db.
   c) new copy is now the current copy of db.
   d) The old copy is deleted.
   e) The transaction is said to have been updated committed at the point where the DB-pointer return to disk.
8. Atomicity
   a) If transaction fails at any time, DB-pointer is updated, the old content of db

are not effected.

b) Transaction abort can be done by just deleting the new copy of db.

c) Hence, either all updates are reflected or None

9. Durability

a) Suppose system fails at any time before the updated DB-pointer is returned to the db

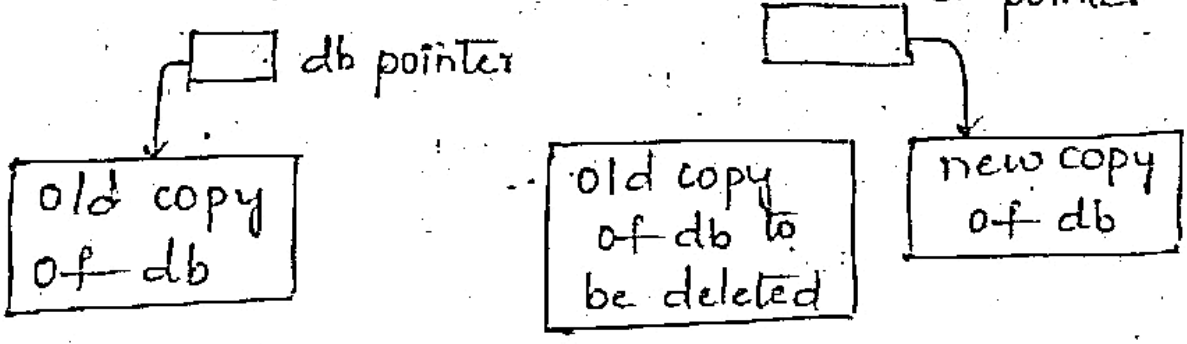
b) When the system restarts it will read DB-pointer and will thus see the original content of db and none of the effects of transaction will be visible.

c) Transaction is assumed to be successful when db pointer is updated.

d) If system fails after DB-pointer has been updated before that all the pages of new copy were returned to disk. Hence, when system restarts it will read new db copy.

Disadvantages of shadow copy schema:

→ Inefficient - as entire db is copied for new transaction.



Concurrency control:

→ Executing multiple transactions at the same time is called 'concurrency'.

Advantages of concurrency:

→ It reduces waiting time
→ It reduces response time
→ It ↑es resource utilization
→ It increases efficiency.

Disadvantages:

→ Lost data update
→ Simultaneous execution of transactions over a shared db can create several data integrity and consistency problems.
→ The three main problems are:
1. Lost updates
2. Uncommitted data
3. Inconsistent retrievals.

→ 4 problems occur during read (or) write

1. reading uncommitted data (Dirty Read Problem) (or) Read Write (WR) conflict
2. Unrepetable read problem (or) Read Write (RW) conflict
3. Lost update problem (or) overwriting uncommitted data. (or) Write Write (WW conflict).
4. Phantom read problem.

Write Read Conflict:

→ Reading uncommitted data means transaction-2 reads a value which is not committed in transaction-1.

ex:

	T_1	T_2
A = 10	Read(A)	
	A = A+10	
A = 20	Write(A)	Read(A) = 20 ✓ uncommitted data (or) Dirty read (or) WR conflict.
	Read(B)	
	Write(B)	
	Read(A)	
	↑ Abort (rollback / restart)	

local buffer: A = 10

local buffer: A = 20

2. Read Write Conflict:

→ Unrepeatable read problem occurs when 2 (or) more read operations of the same transaction reads different values of same variable.

ex:

	T_1	T_2
A = 10	R(A)	
	A = A+10	
	W(A)	R(A) = 10
	commit	R(A) = 20
A = 20		

} RW conflit → It creates confusion.

→ In above ex, once T_2 reads a value A write operation in T_1 changes the value of variable A. Thus, when another read operation is performed by T_2, It reads the new value of A which was updated by T_1.

3. Write Write conflict:

ex:

	T_1	T_2
$x = 10$	R(x)	
	x = x+10	
	W(x)	W(x) → 60 → blind write
$x = 20$		

→ In above ex, T_2 changes the value of x but it will get overwritten by the write commit by T_1 on x

→ Therefore, the update done by T_2 will be lost

→ Basically, the write commit done by the last transaction will override all previous write commits

	T_1	T_2	
$x = 10$	R(x)		
	x = x+10	x = x+20	x = 30
		write(x)	

Phantom read problem:

→ It occurs when a transaction reads a variable once but when it tries to read that same variable again an error occurs saying that the variable doesn't exist.

T_1	T_2
R(x)	Read(x)
Delete(x)	Read(x)

ex, T_2 reads variable x, T_1 once

→ In the above deletes variable x without T_2's knowledge

→ thus, when T_2 tries to read x, it is not able to do it.

→ To prevent concurrency problems in DBMS transaction, several concurrency control techniques can be used including:

1. Locking
2. Time stamp ordering
3. Optimistic concurrency control

Locking:
→ It involves acquiring locks on the data items used by transactions.
→ preventing other transactions from accessing the same data until the lock is released.
→ There are different types of locks such as.
i) shared locks and ii) Exclusive locks.
→ They can be used to prevent dirty read and non-repeatable read.

Time stamp ordering:
→ It assigns a unique time stamp to each transaction and ensures the transaction executes in time stamp order.
→ It can prevent unrepeatable read problem and phantom read problem.

Optimistic concurrency control:
→ It assumes that conflicts between transactions are rare and allows transactions to proceed without acquiring locks initially.

→ If a conflict is deleted, the transaction is rolled back and the conflict is resolved.
→ It can prevent dirty read, unrepeatable read problem, phantom read problem.

Conclusion:
→ concurrency control is crucial in DBMS transaction to ensure data consistency and prevent concurrency problems such as dirty read, unrepeatable and phantom read by using techniques like locking, Time stamp ordering and optimistic concurrency control.
→ By using this techniques developers can build robust db system that support concurrent access while maintaining database consistency.

Serializability:

Transaction:
→ It is a set of operations {Read, write}
→ for a database millions of users can use execute multiple transactions at a time concurrently
→ Collection of transactions is called a Schedule

Serializability:
→ It is possible to convert any non-serial schedule in to its equivalent serial schedule
→ It is a concept that helps us to check which schedules are serializable

→ When all the schedules are serializable then that db will be in consistent state.

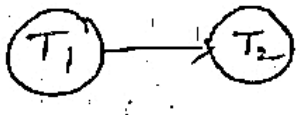
→ Check whether schedules are executed serially (or) not.

ex: Schedule - S_1

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

→ serial schedule

Precedence graph



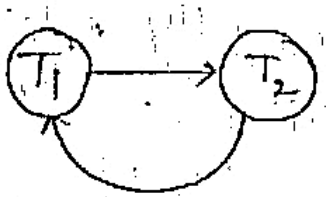
→ If we take T_1 we have set of operations read and write, we take a precedence graph to convert the schedule in to graphical representation to check whether a schedule is serializable (or) not.

→ Here, T_1 and T_2 becomes nodes.

→ Draw an edge from T_1 to T_2 after completing operations in T_1 then we are moving to T_2.

ex: S_2

T_1	T_2
r(a)	
	r(a)
w(a)	
	w(a)



Cycle / loop

→ This precedence graph contains cycles / loop. we call it as parallel schedule (or) non-serial schedule.

→ We have 2 types of serializabilities to convert parallel schedule to serial schedule.

1. Conflict serializability
2. View serializability

Conflict serializability:

→ We need to find where the conflict rises and then we need to solve the conflicts and then make parallel schedule to

→ A Schedule is conflict serializable, if we can convert it in to serial schedule after swapping its non-conflicting operations.

Conflict operations:

→ When we are reading an object A we can't perform any other operation when object A is accessed by a transaction.

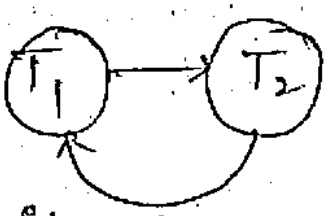
→ Write(A): When we are performing write operation on object A we shouldn't perform any read operation in other transaction.

conflict operation	Non-conflict operation
r(a) — w(a)	w(a) — w(b)
w(a) — r(a)	w(a) — r(b)
w(a) — w(a)	r(b) — w(a)
	r(b) — r(a)

→ Check whether 2 schedules are equal or not.

S_1 →

T_1	T_2
r(a)	
w(a)	
	r(a)
	w(a)
r(b)	



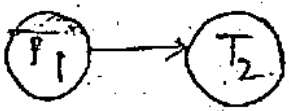
it is a parallel schedule

⇓

to convert it in to serial schedule swap the non-conflicting operations

∴ r(b) − w(a) ⇒ non-conflicting operation

T_1	T_2
r(a) w(a) r(b)	
	r(a) w(a)



⇒ it is a serial schedule