19/9/24/Thu.

DBMS

Different categories of DBMS commands?

1. DDL
   - Create, Alter, Drop, Truncate
   - mainly deals with structure
2. DML - Database Manipulation Language
   - Insert, Update, Delete
   - mainly for dealing or manipulating data
3. DCL - Database Control Language
   - Grant, Revoke
   - By administrators to grant or revoke permissions.
4. TCS - Transaction Control Statement
   - commit, rollback, savepoint.
   - to make all transactions
   - commit - to commit what we have done
   - rollback - Undo.
   - savepoint

★ CREATE - not just used to create a table.
- we can create any object eg: cursor.

Composite primary key:

- Multiple attributes together form a primary key.
- primary key: unique & notnull values.
- only constraint which cannot be specified in column level is composite primary key.
- create table Branch (br_name varchar,
  bank_name varchar,
  . . . . . .
  . . . . . .
  constraint con1 primarykey (br_name, bank_name));
  ↓
  constraint name.

This is how we create a composite primary key constraint. This says that the PK is a combination of br_name and bank_name.

- It is not required to name a constraint but it is better to do so as it is required if we need to drop a constraint ~~later~~ later.

★ Foreign key - at table level and at column level constraint
↓
References.
(Refer ppt)

20/9/24/ Friday.

- If we try to delete rows from a master table which have references to some other table, it will not work.
We will first have to delete all the ~~dependies~~ dependencies with reference values to master table then only delete.

To avoid this big process we can use:
create table Emp(eno ...,
ename ...,
dno vchar references
Dept(deptno) on delete cascade);

This automatically deletes the dependent rows whenever we delete the rows in master table.

Instead if we put on delete set null, it will become null.

Alter:

- To alter the structure.
- Syntax:

1. Alter table tab_name add column col_name datatype;
2. Alter table tab_name drop column col_name;
3. Alter table tab_name rename column col_name to new_colname;
4. Alter table tab_name alter column col_name set not null / drop not null;
5. Alter table tab_name alter column col_name set default value;
6. Alter table tab_name add constraint con_name keyword;
7. Alter table tab_name rename to new_tab_name;

→ used for adding all constraints except not null and default constraints. The syntax for those are written as 4 and 5.

Drop & Truncate:

- for dropping columns - drop
- drop - drops all values including structure, tuples etc.
- truncate - values deleted but structure is maintained.

DML- Insert, Update, Delete

- Insert into tab_name values (val1, val2, ...)
- update tab_name set col_name = value [where condition]
condition can be specified in delete unlike truncate where we delete all rows.
- delete from tab_name [where condition]
- select [all / distinct] col1, col2, ... from tab_name [where condition] [group by groupby-column] [having conditions] [order by col_name]

Q. Consider the following relation:

Accounts (acc_no, cust_no, ac_type, bs_name, bal_amt);

Q1. Retrieve all account balance?
Q2. Find account type for customer C1?
Q3. Find all accounts having balance amount greater than 50 thousand?
Q4. Find all customer numbers which ~~are~~ have balance in the range 50k to 1 lakh.
Q5. Find all account numbers with ac_type as savings or checking?
Q6. Find the cust_no who belong to Kollam branch and having amount < 25k

A:

1. select ~~*~~ bal_amt from Accounts;
2. select ac_type from Accounts where cust_no = 'c1';
3. select * from Accounts where bal_amt > 50000;
4. select cust_no from Accounts where (bal_amt > 50000 and bal_amt < 100000);
5. select ac_no from Accounts where (ac_type = 'savings' or ac_type = 'checking');
6. select cust_no from Accounts where (bs_name = 'Kollam' and bal_amt < 25000);

A:

1. select bal_amt from Accounts;
2. select ac_type from Accounts where cust_no = 'c1';
3. select * from Accounts where bal_amt > 50000;
4. select cust_no from Accounts where (bal_amt > 50000 and bal_amt < 100000);
5. select ac_no from Accounts where (ac_type = 'saving' or ac_type = 'checking');
6. select cust_no from Accounts where (br_name = 'Kollam' and bal_amt < 25000);

23/9/24/Mon.

'like' operator:

- to match patterns
- _ - single unknown character
- % - unknown number of unknown characters.

Q. From Emp(eno, ename, sal, dno), find all employee names starting with 'A'?

A: select ename from Emp where ename like 'A%';

★ '_a%' - eg: Pakvathy, Salicylic

'%a%' - eg: Pickaxe

Sorting:

- select * from Emp by sal desc;

  Sorted in descending order of salary.

- Select * from Emp by ename, sal desc;

  First sorted in ascending order of ename. If ename is same then ordered by descending order of salary.

Group / Aggregate Functions

- built-in functions.
- max(), min(), avg(), sum(), count(), count(*)

  All avoid null values

  ↳ takes null values.

eg:

select count(eno) from Emp where dno = 'D1';

↳ counts the no: of employees in dept D1.

Sorting:

- select * from Emp by sal desc;

sorted in descending order of salary.

- select * from Emp by ename, sal desc;

first sorted in ascending order of ename. If ename is same then ordered by descending order of salary.

Group / Aggregate functions

- built-in functions.
- max() min() avg() sum() count(), count(*)

All avoid null values ↳ takes null values.

eg:
select count(eno) from Emp where dno = 'D1';
↳ counts the no: of employees in dept D1.

3/10/24 Thursday

Consider this relation: Emp(eno, ename, sal, dno, mgr_id)

1. find the no: of employees who are assigned to a manager?

A: ~~count(~~ select count(mgr_id) from Emp;

OR

select count(ename) from Emp where mgr_id is not null;

2. Find the department number and the no: of employees in each department.

A: ~~select count~~
select dno, count(eno) from Emp group by dno;
here first and foremost, the groupby operation happens only after that does the select operation occur for each group.

3. What if in this we need to check whether there are more than 10 employees in the department?

A: select dno, count(eno) from Emp group by dno having count(eno) > 10;

We cannot use 'where' clause in along with group by. We must use 'having' wherever group by is used.

Q. Consider the relations:
Account (ac_no, cno, ac_type, bal_amt, veri_emp_no)
Customer (cno, cname, ctype)

1. Find the no: of accounts verified by each employee.
2. Find the customer numbers having more than one account in the bank.

A:
1. select count(veri_emp_no) from Account;
2. select ~~ac_no~~ cno, count(ac_no) from Account group by cno having count(ac_no) > 1;

Set Operations:

1. Union → the queries must be union compatible.
2. Intersect
3. Except
   ↳ 'minus' in Oracle.

Union Compatibility Rules:

1. The no: of attributes or columns in both the queries must be the same.
2. The datatype of corresponding attributes must be the same.

~~Only~~ If these rules are followed then union blw the queries is possible

Q. Consider the relations:
Customer_FD (cno, fd_no, fd_amt)
Customer_Loan (cno, ln_no, int_rate, ln_amt)

1. Find the customer numbers of those customers having a loan or fd at the bank.
2. Find those customer numbers having both loan and fd.
3. Find customer numbers having an fd but not a loan at the bank.

A: 1. select cno from Customer_FD ~~un~~ union select cno from Customer_Loan;

2. select cno from Customer_FD intersect select cno from Customer_Loan;

3. select cno from Customer_FD except select cno from Customer_Loan;

Customer_Loan;

4/10/24 Friday.

DBMS

DB

- more structured storage.
- data can be interrelated.

MS

- set of programs to manage data.
- ~~provide security~~
- ~~data sharing~~.

Goals Of DBMS:

- Structure for storage of data
- Manipulation of data
- Security
- Sharing of data

Drawbacks of storing data within File System in OS:

1. Data redundancy & inconsistency
2. Difficulty in accessing data
3. Data isolation
4. Integration problems
5. Concurrent access anomaly
6. Atomicity problem
7. Security

Before proper DBMS setups were done, data was stored within the File System in OS. ← The drawbacks were

1. Data redundancy & inconsistency.

Say for example in a university management system, a student can join multiple courses.

BTech

SID	sname	addr	...

Cisco

SID	sname	address	...

here same data will have to be repeated in both ⇒ redundancy.

Now if one value is changed or updated in one table but not done in the other, the same person will have two addresses causing data inconsistency.

2. Difficulty in accessing data.

Writing application programs for accessing data which are stored in different formats is difficult.

Say for example there can be files of extensions .csv, .doc etc. Files can be stored in various formats and this could make it difficult to access.

3. Data isolation.

It is difficult to write application programs to retrieve data from different files.

4. Integrity Problems.

Integrity constraints such as primary key, foreign key, ~~sal~~ salary not less than incentive etc are rules incorporated by organizations.

~~There are no way of adding those integ~~ Enforcing these integrity constraints in file based system is very difficult as we'll have to update the app. pgm each time.

5. Concurrent Access Anomaly:

UMS:

cno	....	vacant seats
c_1		2
c_2		1

Here, say 2 users access the data at the same time and then update it at the same time. This could cause problems.
This problem ~~which~~ occurs due to concurrent access and updation of data ~~is called~~

6. Atomicity Problem:

A $\xrightarrow{1000}$ B

read(A) → making a copy.
A := A - 1000
write(A) → writing it in disk memory
read(B)
B := B + 1000
write(B)

This process should occur in the order.
Say an issue happened at the time of write(A).
Then what happens is 1000 Rs is gone from A but it is not added to B.

Atomicity: performing all the steps of the task, or none even in case of interruptions. This is not ensured in file based system.

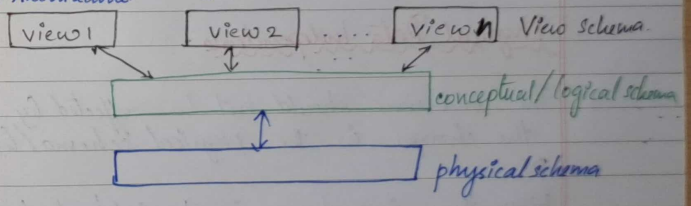
7. Security.

Anyone who has access to the file can see everything in the file which should not be happening.
eg: Salary of one person should not be visible to another person but anyone who has access to the salary file can see everyone's salary.

Data Abstraction

- Complexities is hidden from the end users.
- There are different levels of data abstraction:

3 Layer Architecture:



Physical Schema:

- What implementations are done.
- What database is used.
- Access methods etc.

Conceptual Schema:

- What data is stored
- Is there any relation btw the data. ~~in the~~

View Schema.

- There can be multiple view schemas.
- This is where the end users interact.
- There are diff: views because for different users, the requirements are diff:
- eg: in UMS, a librarian and a student should not have the same view as their requirements are diff:.

Physical Data Independence:

The end users should not be affected by the changes in the physical schema/level.
OR.
When there is a change in physical level, the users in other levels ~~need~~ should not be affected.

are done.
used.
te.

n btw the data. ~~in the~~

ple view schemas.
d users interact-
ws because for
requirements are diff:
librarian and a student
same view as their
..

dence:

not be affected by
physical schema/level.

in physical level, the
should not be affected.

what is contained in the database at a particular time. ↗ overall design

Instance & Schema:

Instance: content of database at a particular time.

Schema: Overall design.

Page No.
Date
FABER-CASTELL

what is contained in the database at a particular time. → overall design

Instance & Schema:

Instance: content of database at a particular time.

Schema: Overall design.

7/10/24 Mon.

Data Model

- underlying structure of a database system.
- allows us to specify constraints, relationships, semantics etc.
  - 4 categories

1. Relational Model.
2. Entity Relationship (ER) Model.
3. Object based data model.
4. Semi Structured data model.

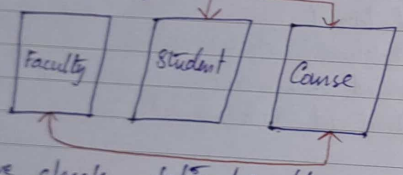
1. Relational Model:

- describes data in the form of small relations.
- table format models.
- The whole system is defined in the form of small relations
- RDBMS: any DBMS which follows the relational model. eg: Oracle, Postgres etc.

2. Entity Relationship Model:

- Data is defined in terms of entities.
- We identify entities from the working of the system.
- widely used in design phase.
- eg: an University Management System: in this Students, Faculty, Courses etc are entities.
- Anything about which we want to store data in the database is called an entity.
- Entities are not independent. ~~on each other~~

eg:



- So we check relationship blw entities in this model.

3. Object based data

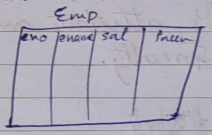
- How to store object directly in data?
- extension of ER model but supports objects.
- They also support certain object paradigms

- ORDBMS - supports Object based models and Relational model.
  - eg: Postgres.

4. Semi structured model:

- all other models are structured.

eg:



for every employee the same values are stored.

- In semi structured model, not all employees would have same values.
  - eg: there might be an employee who has an attribute 'qualification' which is not present for the other employees.

- eg: XML

Entity Relationship Model.

1. Entities
   - Regular/Strong entities: exist independently
   - Weak entities: depend on other entities to exist.

2. Attributes :

- characteristics of an entity.

Faculty (fid, fname, ~~dno~~ ...)
Dept (dno, dname)

'dno' is not a direct attribute of faculty as it describes dept and not the faculty. So 'dno' is Dept's attribute and not the attribute of ~~Dept~~ Faculty.

Emp
eno
ename
sal
addr
phone
~~pno~~
email
dob
age.

1. Simple & Composite Attribute :

- attributes which can be further divided : composite.
  eg: ename → fname, middle name, last name. , addr. etc.
- which can't be further divided: simple. eg: eno.

2. Single valued & Multivalued Attribute.

- single valued: only one value can be assigned for a single entity.
  eg: eno, ename etc.
- multivalued: multiple values for single entity.
  eg: addr, phone, email etc.

3. Null attributes.

- Can be zero.
- eg: sal, email etc.

4. Derived attributes.
- can be derived from another attribute.
- eg: age. (it can be derived from dob)
- ~~No~~ No derived attributes are put in the database at implementation phase. It may be present in the design phase.

4. Derived attributes.
- can be derived from another attribute.
- eg: age. (it can be derived from dob)
- ~~No~~ No derived attributes are put in the database at implementation phase. It may be present in the design phase.

14/10/24 | Monday.

Relationship sets.

- association blw entities in a system.
- Two concepts:
  1. Cardinality ratio (mapping cardinality)

say two entity sets: entity & entity

E_1 —— R —— E_2

how many entities from E_2 can be associated to one entity in E_1.

There can be: one in E_1 to one in E_2 & vice versa.

1. one to one
2. one to many
3. many to one
4. many to many.

reverse

one from E_1 associated to many in E_2

but only one in E_2 is associated ONLY to ONE in E_1.

one to one:

example in a UMS,

E_1 ← Faculty — Relationship: HOD — Dept → E_2

one HOD (one faculty) to one dept.

one to many:

Dept — Faculty

one dept has many faculties.

Reverse for many to one.

many to many:

Student — Course.

Many student in many courses.

2. Participation constraint: entity & relationship.

E_1 ← Student — Advisor (R) — Faculty → E_2

Total participation ← Every student must have an advisor.
But every faculty need not participate in being advisor. → Partial participation

Descriptive Attributes:

If an attribute is equally important ~~between~~ to the entities then we assign the attribute to the relationship.
Attributes that define or are associated with the relationship are called descriptive attributes.

★ degree of a relation: no: of columns / attributes
cardinality of a relation: no: of rows / tuples.

Cardinality of a relationship: no: of entities participating in a relationship.

- $E_1 - R$: unary
- $E_1 - R - E_2$: binary
- $E_1 - R - E_2$ (with E_3 below R) : tertiary.
- $E_2 - R - E_3$ (with E_1 above R, E_4 below R) : quaternary.

Unary Relationship:

- Must include the roles just above the connection lines.
- eg: a course is related to another course through a pre-req.

Course — course_id — pre-req — pre-req_id

Regular & Weak Entities:

Weak Entity: eg: Bank has Branch

branch depends on bank for its existence.
So branch is a weak entity.
Entities which depend on other entities for its existence: weak entity.
Weak entities do not have a primary key on their own. They have an attribute from ~~Bank~~ strong entity and then an attribute from weak called partial key / discriminator.

Attribute to distinguish blw diff: instances of a particular strong entity — partial key.

eg: branchname is a discriminator / partial key as if we have two ~~banks : A~~ instances of bank A, one could be in Vallikavu & other can be in TVM (which is written as branch name).

Identifying relationship:
The relationship which connects an ~~dependent~~ weak entity to its dependent entity.

Here Bank → dependent
~~bb~~ Branch → weak.

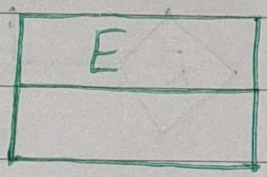
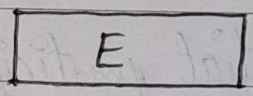
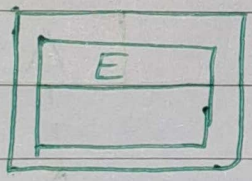
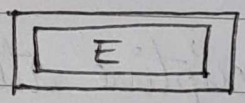
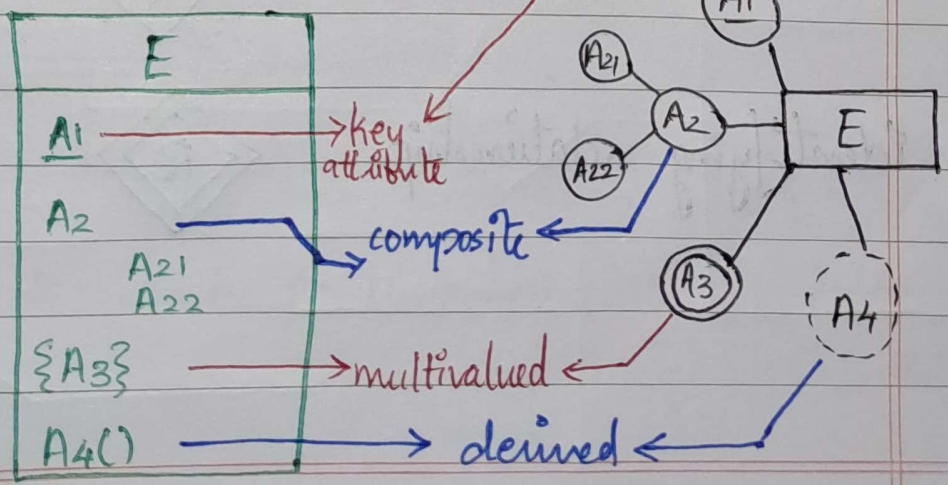
Identifying relationship:
The relationship which connects a ~~dependent~~ weak entity to its dependent entity.

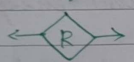
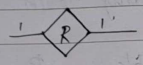
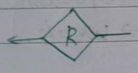
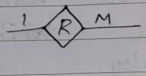
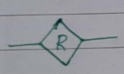
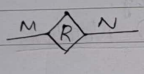
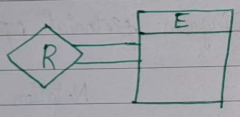
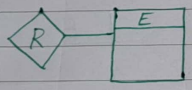
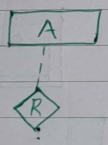
Here Bank → dependent
~~Bank~~ Branch → weak.

15/10/24 Tuesday.

ERD Notations:

ER Diagram: pictorial representation of ER model.

	Notation 1	Notation 2.
Entity set.		
Weak entity		
Attributes		

	Notation 1	Notation 2.
Relationship		
1-1 R		
1-M R		
M-M R		
total participation		
partial participation		
descriptive attribute		
Identifying relationship		

Q. Draw an ERD for university database application with the following description

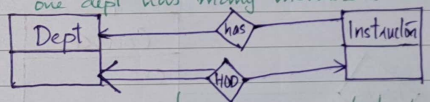
1. A university has many departments
2. Each dept has multiple instructors, one among them is the HOD of the dept.
3. An instructor belongs to only one dept
4. Each dept offers multiple courses, each of which is taught by a single instructor
5. A student may enroll for many courses offered by different dept

A: Entities present & their attributes:

1. Dept - dno, dname, dloc
2. Instructor - ino, iname, sal, quali
3. Course - cno, cname, credit
4. Student - sno, sname, sadd.

The relationships present:

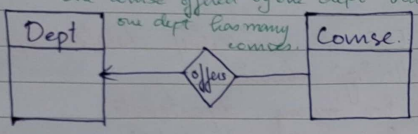
one dept has many instructors but one inst has one dept



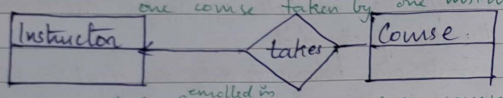
↳ one HOD per dept & only one inst becomes HOD. every dept must have HOD.

* for 1-1 R check participation constraint each time.

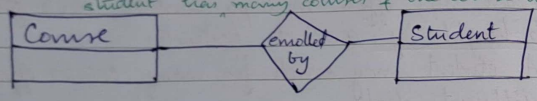
- one course offered by one dept but one dept has many courses.



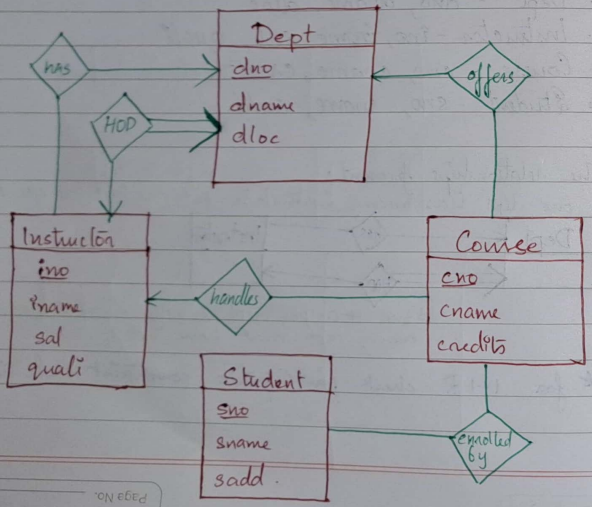
- one course taken by one instructor.



- student enrolled in has many courses & one course has many students.



Now make all this into single ERD,



Q. Consider a banking scenario given below & draw an ERD for the same.

1. Assume in a city there are multiple banks & each bank has many branches.
2. Each branch has multiple customers.
3. Customers have various types of accounts.
4. Some customers have also taken diff types of loans from these bank branches.
5. One customer can have multiple accounts & loans.

A: Entities & attributes:

- Bank - bname
- Branch
- Customer
- Account
- Loan.

17/10/24/Thursday.

Subquery:

Emp (eno, ename, sal, dno)

Dept (dept_no, dname, dloc)

Q. Find name of dept in which emp E_1 works.

select dname from Dept where dept_no = ?

select dno from Emp where eno = 'E1';

(we do not know this query for this)

Thus a query inside a query, is called subquery, is used.

→ select dname from Dept where dept_no = (select dno from Emp where eno = 'E1');

The subquery within the innermost () is executed first.

Q. Find the name of all employees who work in 'sales' department?

A: select ename from Emp where dno = (select dno from Dept where dname = 'sales');
dept_no

Q. Find those employees (names) of employee 'E1'?

A: select ename from Emp where sal > (select sal from Emp where eno = 'E1');

Q. Consider the given schema

Customer (cno, cname, ctype)
Accounts (ac_no, cno, ac_type)
Customer_Loan (cno, ln_no, ln_amt)
Customer_FD (cno, fd_no, fd_amt, int_rate)

1. Find the name of all customers who have taken a loan of amount > 50k?
2. Find those customer numbers who have same ac_type as customer 'C1'
3. Find those customer names who have not taken any loan.
4. Find those customer names who have a fixed deposit at the bank.

A: 1. select cname from Customer where cno in (select cno from Customer_Loan where ln_amt > 50000);

2. select cno from Accounts where ac_type = (select ac_type from Accounts where cno = 'C1');

3. select cname from ~~Accounts~~ Customer where cno not in (select cno from Customer_Loan);

4. select cname from Customer where cno in (select cno from Customer_FD);

We can only use '=' operator if value is one. If more values then use 'in'.

Subqueries or inner queries which can work independently ~~of~~ without needing anything ~~is called~~ from outer query is called Independent subquery.

↓opposite

Correlated subquery.

↳needs value from outer query.

18/10/24 | Friday.

Q. Consider the given relation.

Emp (eno, ename, sal, dno)
Emp-dependent (eno, dep_no, dep_name, gender)
Emp-Proj (eno, pno, work_hrs)
Proj (pno, pname)

1. Find the ename of all employees getting salary greater than that of all employees in dno 'D1'.

1: select ename from EMP where sal > ALL (select sal from EMP where dno = 'D1')

~~if we have~~

if we have an inner query which returns multiple values & we need to compare those with a single value; we use 'ALL'.

OR.

select ename from EMP where sal > (select max(sal) from EMP where dno = 'D1')

2. List the name of employees getting sal greater than that of any employee in dept 'D1'.

1: select ename from EMP where sal > ANY (select sal from EMP where dno = 'D1')

or

select ename from EMP where sal > (select min(sal) from EMP where dno = 'D1').

~~if we~~

ANY: for each tuple the salary will be taken & compare it with the returned value of inner query & if it is greater than it will break there & won't look into the others.

Q. Same relation as previous qn except.
EMP(eno, ename, sal)

1. Find the name of employee getting highest sal?
2. List the ename having female dependents.
3. Find the ename of all emps who worked in 'Order Entry' project for more than 10 hrs.
4. Find ename getting sal more than avg sal of all employees.

A:
1. select ename from EMP where sal = (select max(sal) from EMP);

2. select ename from EMP where eno = (select eno from Emp_dependent where gender = 'F');

3. select ename from EMP where eno = (select eno from Emp_Proj where pno = (select pno from Proj where pname = 'Order Entry'));

4. select ename from EMP where sal > (select avg(sal) from EMP);

5. Find the names of emp who is not working in any proj.

A: select ename from EMP where eno not in (select eno from Emp_Proj);

A. 3. select ename from EMP where eno in (select eno from Emp_Proj where work_hrs > 10) and pno = (select pno from Proj where pname = 'Order Entry');

Q. Consider these relations
EMP(eno, ename, sal, dno, job_id)
Accounts(ac_no, cur_bal, br_no)
Job(job_id, jname)

1. Find all dept getting min sal > dept 'D1'?
2. Find eno & name of those emp who are not IT programmer & getting sal ≤ Any IT programmer.

A:
1. select dno, min(sal) from Emp Group by dno having min(sal) > (select min(sal) from EMP where dno = 'D1');

2. select eno, ename from EMP where job_id = (select job_id from Job where jname = 'IT programmer') and sal ≤ ANY (select sal from EMP where job_id = (select job_id from Job where jname = 'IT programmer'));

Correlated subquery:

- Inner queries that require some sort of reference from outer subquery.
- The inner query will be executed as many times as the no. of tuples in the outer query for a correlated subquery where as for an independent subquery it executes only once.

Q. Customer (cno, cname)
Cust_FD(cno, fd_no, fd_amt)
Cust_loan(cno, ln_no, ln_amt, int_rate)

1. List all customers who have a fixed deposit of amount less than sum of all loans taken by that customer.

A: select cno from Cust_FD F where
fd_amt < (select sum (ln_amt) from Cust_loan where cno = F.cno);

Cust_FD F → temporary name.
↳ Here F refers to the 'alias' name of the Cust_FD, so that whenever we get/use 'F' from here after then it means that table.

§ cno = F.cno
cno that we have selected from F.
Here we are saying that in the inner query, the 'cno' should be equal to that of 'F.cno'.

If in the inner query we wrote as:
(... where cno = cno) The system would return an ambiguity error.

2. List all cust names who have both FD & Loan.

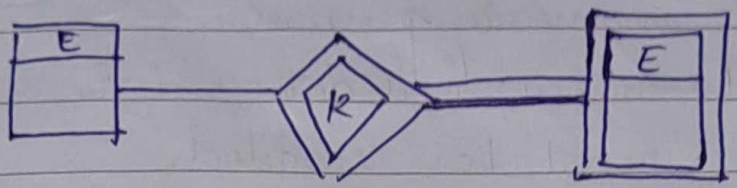
A: select cname from Customer F where cno in (select cno from Cust_FD where cno = F.cno) and cno in (select cno from Cust_loan where cno = F.cno);

3. List accounts along with cus_bal and bn_no having a balance more than the avg bal of the branch to which the account belongs.
Accounts (ac_no, cus_bal, bn_no).

A:- select ac_no from Accounts A where cus_bal > (select avg (cus_bal) from Accounts where bn_no = A.bn_no);

21/10/24/Mon.

continued here from (17/10/24 (Thu)



every weak entity must have a total relation (identifying relation) with dependent relation.

(solution put in NUMS)

Q. Draw an ERD for the following company database.

1. The company is organised into departments.
2. Each dept has a unique no., name and a particular employee who manages the dept.
3. We keep track of the starting date, that particular employee started managing the dept.
4. A dept may have several locations.
5. A dept controls a no. of projects each of which has a unique number, name and a single location.
6. We store each employees' id, name, addr, salary, gender and birth date.
7. An employee is assigned to only one dept, but may work on several projects which are controlled by different departments.
8. We also keep track of the no. of hours an employee works on each project.
9. There is a supervisor for each employee who is another

employee.

10. We want to keep track of the dependence of each employee for insurance purpose.

11. The dependent's name, birth date and the relationship with employee must be recorded.

A: Entities & attributes.

~~• Department~~

• Dept - dno, dname, ~~dhead dmanager, startdate~~, ~~• Employee~~ dloc.

relation ← attributes

• Employee - eid, empname, addr, salary, gender, dob, ~~dept dno~~

• Project - pno, pname, ploc.

• Dependence - depname, ~~dob~~ dob, relation.

Relationships · starting date

1. Dept — manages — Employee; Dept — assigned — Employee

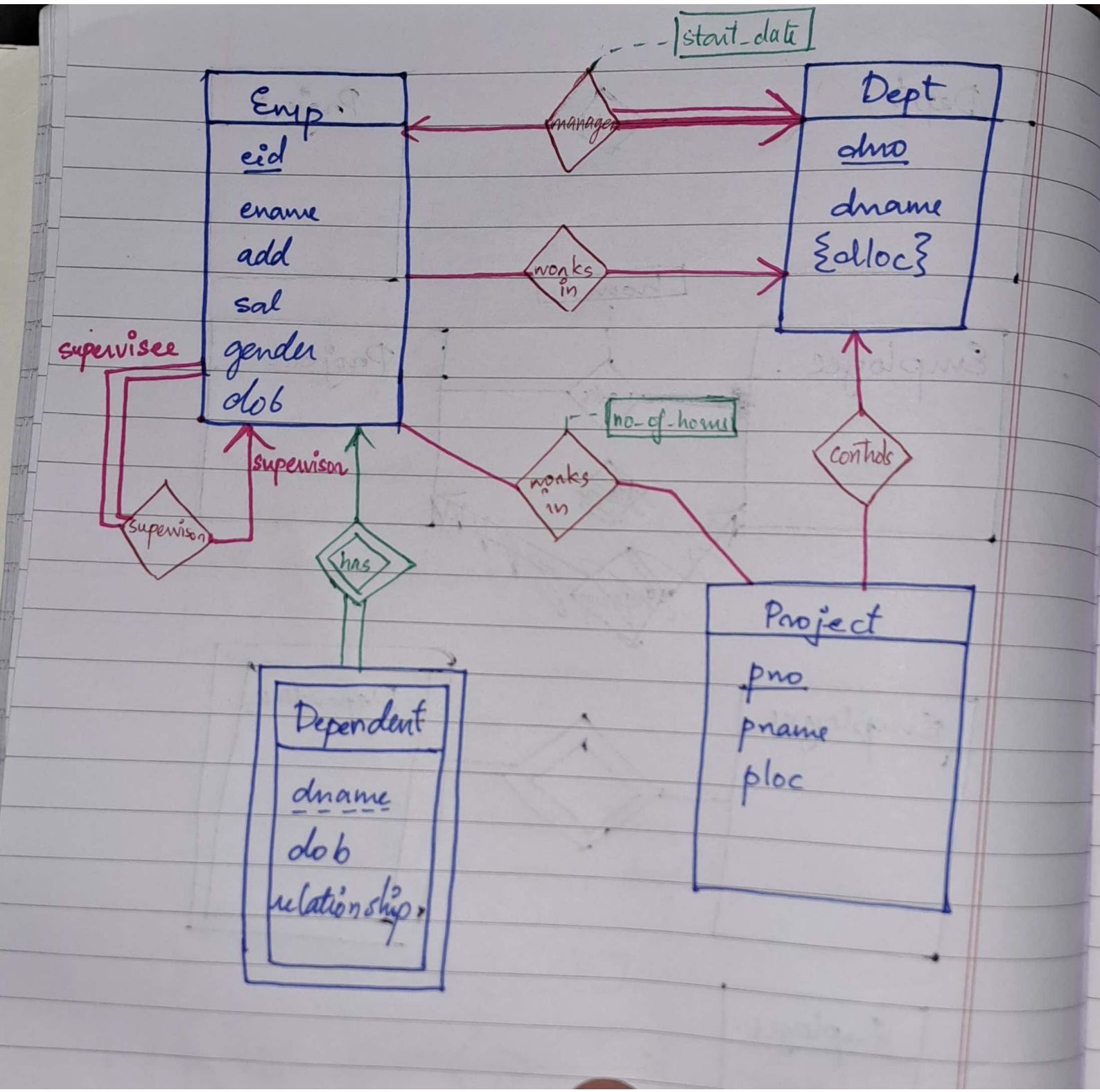
2. Dept — control — Project; Employee — works — Project; Employee — dependence — Dependence

3. Employee — Dependence

4. Employee — supervise (Supervisee) / supervision (Supervisor); many

Date / Page No.

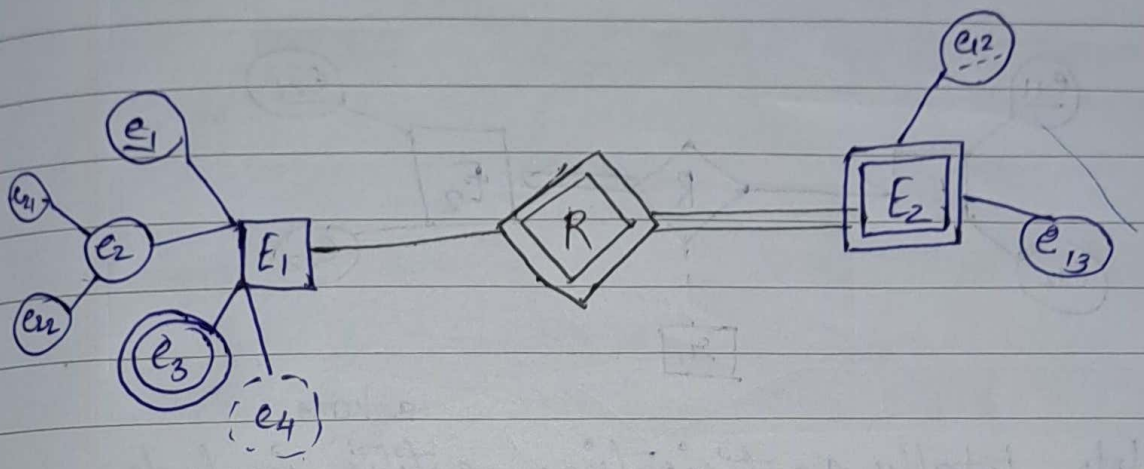
FABER-CASTELL


start_date
Emp
eid
ename
add
sal
gender
dob
manages
Dept
dno
dname
{dloc}
works in
supervisee
supervison
supervison
no_of_hours
works in
controls
has
Project
pno
pname
ploc
Dependent
dname
dob
relationship

24/10/24 | Thursday.

ER to Relational Mapping:

1. Regular & Weak Entities



- ▷ No derived attributes included into schema.
- ▷ Components of composite attribute are added to schema.
- ★▷ Include primary key of dependent entity as a foreign key of weak entity

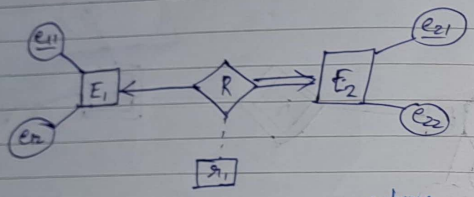
components of composite attribute e_2.

E_1 ($\underline{e_1}$, e_{21}, e_{22})

E_2 ($\underline{e_1}$, e_{12}, e_{13})

↳ primary key of E_1 added as foreign key of E_2.

2. Mapping 1-1 Relationship:



▷ Into totally participating entity's schema include primary key of partially participating entity as a foreign key.

▷ The descriptive attributes, if present, should be added to totally participating entity's schema.

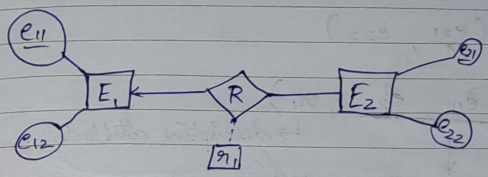
$E_1(\underline{e_{11}}, e_{12})$ → partially participating entity.

$E_2(\underline{e_{21}}, e_{22}, e_{11}, r_1)$ → total participation entity.

↓

descriptive attribute.

3. Mapping 1-M / M-1 Relationship:



Same a previous (1-1 relationship).

Total participation == Many side.

Partial participation == One side.

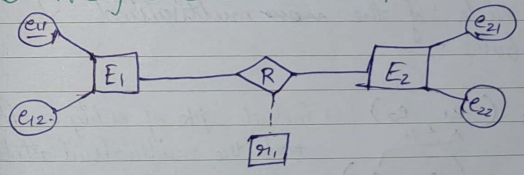
$E_1(\underline{e_{11}}, e_{12})$ → one.

$E_2(\underline{e_{21}}, e_{22}, e_{11}, r_1)$ → many.

↓

descriptive attribute.

4. Mapping M-M relationship:



For each M-M relationships, add new table schema.

E_1 (e_{11}, e_{12})

fk E_2 (e_{21}, e_{22})

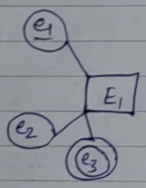
R (e_{11}, e_{21}, a_1)

↳ descriptive attribute.

↓

participating entities' primary keys.

5. Multivalued Attributes

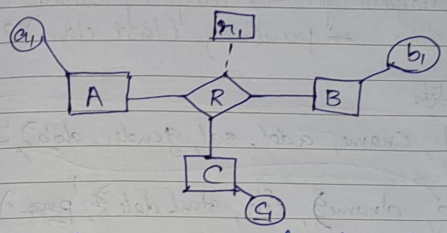


We create a new relation corresponding to each of the ~~more~~ multivalued attributes.

E_1 (e_1, e_2) ◦ include pk of entity as fk.

$E_1\_e_3$ (e_3, e_1) ◦ the multivalued attribute with pk becomes the pk of the relation

6. Relation's degree > 2.



create new table for the relation with degree > 2.

Include pks of all the participating entities as foreign key & merge them & use as pk.

Include descriptive attributes as well.

A (a_1)

B (b_1)

C (c_1)

R (a_1, b_1, c_1, r_1)

The output of ER to relational mapping is called schema diagram.

It should have attributes of entities & relations (if fks used etc)

Schema

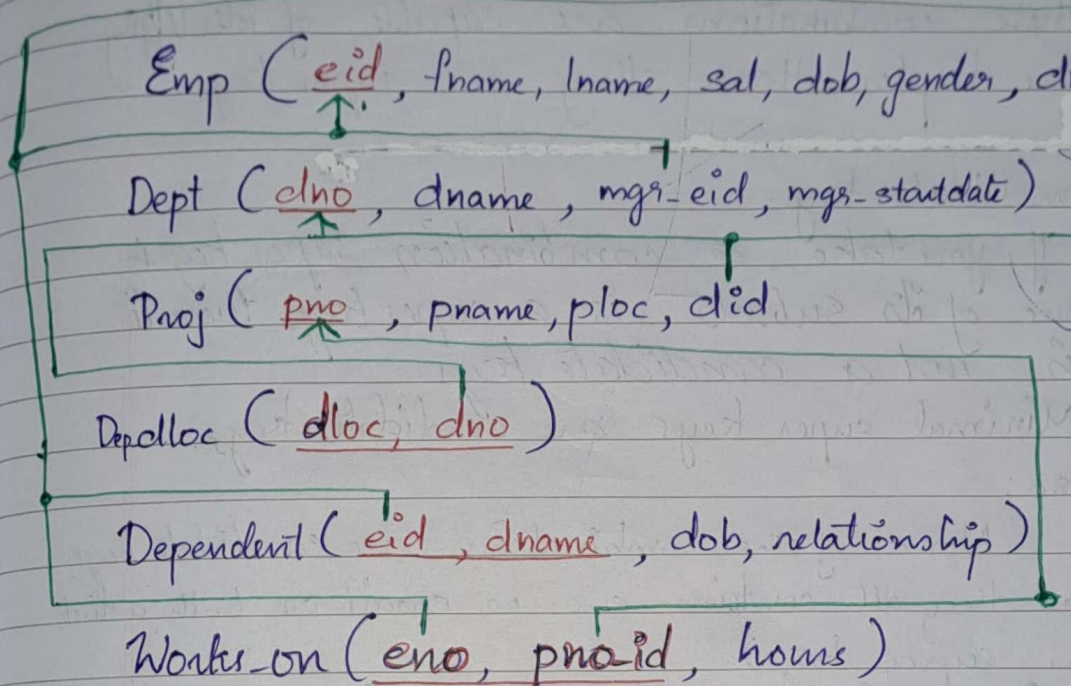
Q. Schema diagram for the company database question done previously (last class)?

A:

Regular entities

25/10/24/Friday.

(Tables need not have same names as entities)



Procedure of selecting primary keys:

1. Select the super keys
2. From there select candidate keys
3. And thereby select primary keys.

eg: Emp (eno, ename, email, age)

Super keys:

{eno}, {email}

{eno, ename}, {eno, email}, {eno, age}

{ename, email}, {email, age}

{eno, ename, email}, {eno, email, age}

{eno, ename, email, age}

These combinations are capable of identifying the tuples. Hence they are super keys.

Candidate key.

If you take a combination super key & one of its subsets is a super key then it is not a candidate keys.

Minimal super keys are candidate keys.

so:

None of the combinations are candidate keys (as they all contain eno or email or both which are super keys).

Hence: {eno} and {email} are the candidate keys.

Out of the candidate keys, the developers usually choose the key which is easy to remember or do not change often.

Here {eno} is chosen.

Extended ER (EER) features:

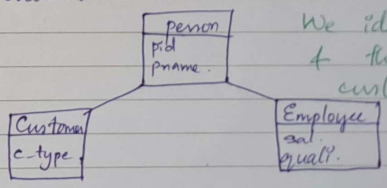
Certain features of ~~is~~ OOR such as inheritance, polymorphism etc cannot be displayed using our normal ERD. So we ~~extend~~ included them in EER.

The features are:

1. Generalisation / Specialization
2. Super class / Sub class.
3. Attribute Inheritance
4. Aggregation.

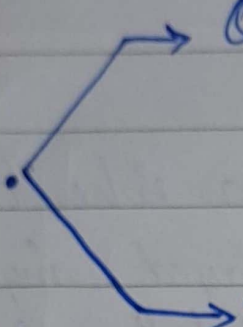
1. Generalisation / Specialization:

- Top down approach. - Specialization.
- Identify major entity & then identify the sub entities.



We identify person & then identify customer & employee.

- Bottom - up approach - Generalisation
- Same thing but we identify employee & customer first & then person.



Overlapping - a customer can also be an employee.

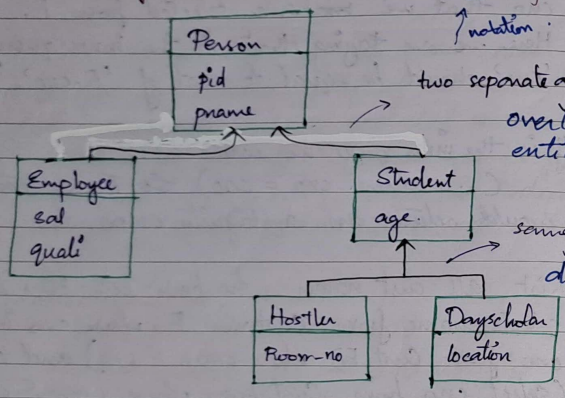
Disjoint - a student can either be a day scholar or a hostlers.

Here the person → super class.
customer & employee → sub class.

Attributes of person are inherited by customer & employee.

30/10/24/Wed.

EER features:



d/o
disjoint
overlapping
Employee
Stud
on
notation.

two separate arrows
↓
overlapping entities.

same arrow
↓
disjoint entities.

↓ relational schema.

Person(pid, pname)
Employee(eid, sal, quali)
Student (sid, age)

- Create a table for the head entity (Person)
- Then create separate table for the sub entity with their specific attributes + then add the pk of the ~~two~~ super entity as a fk in the sub entity tables.

Disadv:
To get the whole information related to a sub entity then you'll have a hard time.

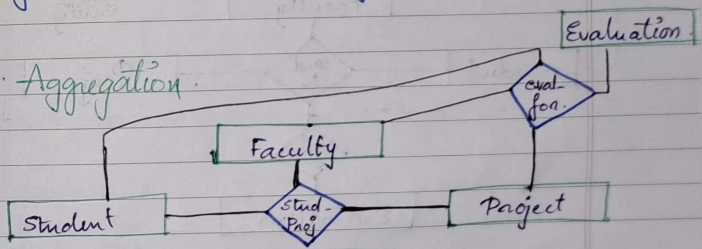
↓ how to remove.

In the relational schema do not create a table for the higher level entity. ~~Create only on~~ Only create the tables corresponding to the lower level entity.

↓ Disadv.

In case of overlapping entities, there would be repetitions in the overlapping entities.
Hence we only use this kind of relational schema if all the entities are disjoint.

2. Aggregation.



Faculty(...) Project(...)
Student(...) Stud_Project(sid, fid, pid)

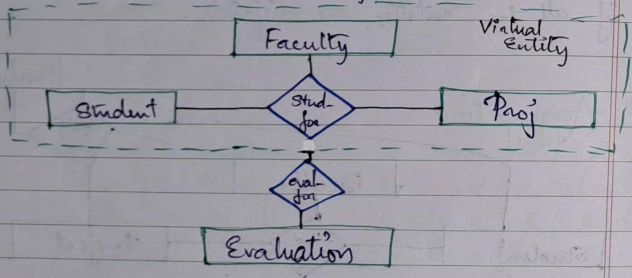
eval_for (sid, fid, pid, evalid)
Stud_Proj (sid, fid, pid)

There is redundancy

So can we remove Stud_Proj ?
↳ No, we cannot. Some projects may have just started & not yet had its evaluation.

We can remove repetition using aggregation.

In aggregation,
we will consider Stud_Proj connected with other 3 entities as an abstract entity. And then eval_for will be connected to that abstract / virtual entity.



Rule: If aggregation present, the relationship considered as a virtual entity will not have a table created for it. Instead we will only create a table for the other relationship (eval_for) & the components of virtual entity will be the pk and the pk from the entity (Evaluation) will just be treated as a fk.

Thus the tables would be:

Student (sid, ...)
Faculty (fid, ...)
Proj (pid, ...)
Evaluation (eval_id, ...)
Eval_for (sid, fid, pid, eval_id)

the pk for the Stud_Proj

Going back to sql...

Joins:

1. Inner join
2. Equi join
3. Natural join
4. Outer join
5. Self join
6. Cross join.

Joins are used to get info from more than one table at a time

1. Inner Join & Cross join & Natural join :

Student (sid, sname, age, cno)

Course (course_no, cname, credits)

Q. How to find info. of student details AND course details?

We use joins.

→ select * from Student, Course

↓

If we say only this much it will be a cross join.

1st tuple in Student will be combined with all tuples in Course, 2nd tuple in Student will be combined with all tuples in Course etc.

A (10 tuples) B (5 tuples) A×B = 50 tuples ↳ cross join.

This is not what we want. So what.

Internally:

The system will do cross join & then apply conditions for the other joins onto the cross join output & get the output for other joins.

actual answer:

▷ select * from Student, Course where cno = course_no ;

or.

▷ select * Student natural join Course;

↓

In natural join, whichever column is common for both tables, it will consider that as the condition and give output. Common col ⇒ there must be reference

or

▷ select * Student inner join Course ~~on~~ on cno = course_no;

Natural join only works if the two tables have same columns & that refer each other.

actual answer:

▷ select * from Student, Comse where cno = course_no ; → This is also equi join because we use equality operator

or.

▷ select * from Student natural join Comse;

↓

In natural join, whichever column is common for both tables, it will consider that as the condition and give output. Common col ⇒ there must be reference.

or

▷ select * from Student inner join Comse ~~on~~ on cno = course_no;

Natural join only works if the ~~two~~ tables ~~columns~~ have same columns & that refer each other.

12/11/24/Tuesday.

2. Equi join:

Inner join with equality ~~operator~~ condition.

eg: select sname, cname from Student, Course where cno = course_no;

Q. Emp(eno, ename, city, sal)

Customer(cno, cname, type, city, sal)

Q1 Find the name of customers getting salary > that of employees

Page No.

Date

FABER-CASTELL

A: ~~select ename from Customer where sal > ANY~~

select cname from Customer C, Emp E where C.sal > E.sal.

↓ This is an inner join. Because we are combining two tables who have no referencing between each other. ~~Thus not sm.~~

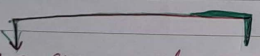
2. Find employee names & customer names who belong to the same city?

A: select ename, cname from Customer C, Emp E where C.city = E.city;

OR

select ename, cname from Emp E inner join Customer C on E.city = C.city;

Self Join:



Emp (eno, ename, sal, mgr_no)

E_1	a	–	E_1
E_2	b	–	E_1
E_3	c	–	E_2
E_4	d	–	E_1

A join which joins a table to itself.

1. Find employee names with corresponding manager names?

A: The answer we need:

ename	mgr_name
a	d
b	a
c	b
d	a

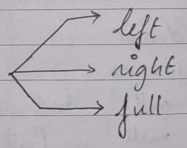
We will consider one more copy of the same table. Thus:

E_1 (eno, ename, sal, mgr_no)
E_2 (eno, ename, sal, mgr_no).

The query becomes:

select E1.ename, E2.ename from Emp E1, Emp E2 where E1.mgr_no = E2.eno;

Outer Join



Customer (cno, cname, cadd)
Cust_FD (cno, fd_no, fd_amt)

A join in which it will print matching tuples and non-matching tuples.

1. Find customer details with corresponding fd details if any?
   ↳ We have to print all customer details as well as the fd details if they have fd.
   Thus we have to use outer join.

```
select * from Customer C left outer join
Cust_FD CFD on C.cno = CFD.cno;
```

left outer join:
all tuples from the table on left side is printed in the output and only the matching condition values in right table is printed other values are written as null.

cno	cname	cadd	cno	fd_no	fd_amt
C_1	a	x	a	123	120000
C_2	b	y	null	null	null
C_3	c	z	null	null	null

→ had no value in Cust_FD

right outer join:
Same as left just that instead of left table it will be the right table.

Full outer join:
from both tables, all matching and unmatching tuples are printed out.

right outer join:
Same as left just that instead of left table it will be the right table.

Full outer join:
from both tables, all matching and unmatching tuples are printed out.

14/11/24 Thursday.

Normalization:

Stud_Course_Grade (sno, sname, cno, cname, credit, mark, grade)

S_1	a	C_1	OS	3	85	A
S_1	a	C_2	WT	4	90	A+
S_2	b	C_1	OS	3	35	F
S_3	c	C_1	OS	3	70	B
S_4	d	C_3	CCNA	2	85	A

We do normalization if anomalies are present.

There are 3 types of anomalies that can occur:

→ Insert

Assume a new course has been introduced but no student has joined the course. sno + cno form PK of the schema so we cannot insert the new course because we do not have any student in it.

Thus this is an insert anomaly.

→ Update

Say the student S1 joins another course and we insert / update the schema. When there is repetition of information, whenever we want to update on the repeated data, we need to make changes in multiple rows. Thus creates an update anomaly.

→ Delete

Assume student S4 withdraws from the course. Then we will need to remove the whole row and also the course because that course is only taken by S4.

Hence this is delete anomaly.

Functional Dependency:

R(x, y, z)

x → y ⇒ x determines y or y is functionally dependent on x.

(x: determinant)

For a particular value of x, we must always get a unique value of y attribute. Then x determines y.

$x_1 \rightarrow y_1$
$x_2 \rightarrow y_1$
$x_1 \rightarrow y_1$

x_1 can only have y_1.

Thus in the given schema, we find functional dependency of every non-key attribute

sno → sname
cno → cname
cno → credit
sno, cno → mark
~~sno, cno → grade~~
mark → grade

} all non-key attributes.

alt

1. Partial Functional Dependency:

If a non key attribute is functionally dependent on a part of the primary key then it is partial FD.

eg: sname, cname, credit.

2. Full Functional Dependency:

De For the non key attribute determinant is the full primary key.

eg: mark.

3. Transitive FD:

We derive the FD from other functional dependencies.

eg: mark → grade - ①

sno, cno → mark - ②

from ① & ②:

sno, cno → grade.

Normal forms:

Normalization is based on certain normal forms.

1. First Normal form (1NF)

A relation is in 1NF if all the attributes should be atomic in nature.

↳ not composite

not multivalued.

In the given schema (Stud_ Course_ Grade), all attributes are atomic and hence is in 1NF (one NF (pronunciation))

2. Second Normal form (2NF)

Conditions:

1. Must be in 1NF
2. No partial dependency b/w non-key attribute & key attribute.

3. Third Normal form (3NF):

Conditions:

1. Must be in 2NF
2. No transitive dependency between non-key attributes and key attributes.

In the given schema, there are partial dependencies hence it is not in 2NF.

↳ So now we ~~dbe~~ subdivide relations so that it becomes 2NF.

3. Third Normal Form (3NF):

Conditions:

1. Must be in 2NF
2. No transitive dependency between non-key attributes and key attributes.

In the given schema, there are partial dependencies hence it is not in 2NF.

↳ So now we ~~dw~~ subdivide relations so that it becomes 2NF.

15/11/24 | Friday.

The partial dependencies were:

sno → sname

cno → cname, credits

Now we split this,

Student (sno, sname)

Course (cno, cname, credits)

Now ~~one~~ two more dependency left

sno, cno → mark

mark → grade.

This becomes:

Mark Grade (sno, cno, mark, grade)

Thus Stud_Course_Grade has been divided into 3 relations: Student, Course & Mark Grade.

How to check if we can stop normalization?
We check if any anomalies are left.

Here:

Student (sno, sname)
Course (cno, cname, credits)
Mark_Grade (sno, cno, mark, grade)

We can see that no more insert, delete or update anomalies are present in Student or Course.

For Mark_Grade:

Mark_Grade (sno, cno, mark, grade)

S_1	C_1	85	A
S_1	C_2	70	B-
S_2	C_2	85	~~B~~ A
S_3	C_1	80	B^+
S_4	C_2	85	A
-	-	-	-
-	-	-	-

are we repeating any data?
↳ No but check A grade and mark. A particular mark has particular grade is being repeated here. That is in some way an update anomaly is present. Say grade for 85 marks has changed to A-, we'll have to change values in multiple rows.

Any delete anomaly?
Assume student S_1 is deleted. Then 70 marks gives B- grade is also deleted. Thus delete anomaly occurs here.

Is insert anomaly present?
No.

But since ~~in~~ delete & update anomalies are present we go for nxt normal form.

3NF

Mark_Grade (sno, cno, mark, grade)

only this relation has anomalies so we only check this relation.

Mark_Grade → is in 2NF
→ but has transitive FD
→ so not in 3NF.
↓
split relation.

Thus table divided as:
Stud_Mark (sno, cno, mark)
Mark_grade (mark, grade)

Any info repeated?
No info repeated in either tables. Thus no update anomaly.

Any delete anomaly?
No delete anomaly. marks & grade not affected when you delete a student info.

Thus no more anomalies. The relation becomes:
Student (sno, sname)
Course (cno, cname, credits)
Stud_Mark ~~(grade)~~ (sno, cno, mark)
Mark_grade (mark, grade)

Q. Assume it was given like the following & we have to normalise it.
Stud-Course_Grade

Stud_Details		Course_Details			Mark_details	
S_1	a	C_1	OS	3	85	A
S_2	b	C_1	DBMS	4	80	B+
S_3	c	C_2	WT	3	70	B-
S_1	a	C_2	WT	3	90	A+

Here the attributes are:
Stud_Details ← S_1, a
Course_Details ← C_1, OS, 3
Mark_Details ← 85, A

They are all composite attributes.
Thus this is NOT IN 1NF.

Thus we split it into:
Stud-Course_Grade (sno, sname, cno, cname, credits, mark, grade)
The remaining is done as we did previously.

Here the attributes are:

Stud_Details ← S1, a

Course_Details ← C1, OS, 3

Mark_Details ← 85, A

They are all composite attributes. Thus this is NOT IN 1NF.

Thus we split it into:

Stud_Course_Grade (sno, sname, cno, cname, credits, mark, grade)

The remaining is done as we did previously.

18/11/24/Monday.

Q. Salesperson_Product (sl_no, sname, pno, pr_name, unit_price, city, status, qty_sold);

Assumption: one prod. sold by many salesman. One salesman sells many products.

	sl_no	sname	pno	pr_name	unit_price	city	status	qty_sold
Sample tuples:	S1	a	P1	aa	10	C1	A	100
	S1	a	P2	bb	15	C1	B	200
	S2	b	P1	aa	10	C2	B	150
	S2	bb	P2	bb	15	C2	C	500
	S3	d	P3	cc	20	C3	D	250
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Page No. Date

FABER-CASTELL

Is there insert anomaly?

▷ Say a new salesman joined. We are not able to insert the value because they have not yet sold anything ⇒ Insert Anomaly.

▷ Say a new product has been introduced. It has not yet been sold. So again we cannot insert it ⇒ Insert Anomaly.

Is there update anomaly?

▷ Name of salesman and other info related to salesman is repeated.

▷ ~~Poo~~ If same product is sold by 2 different salesmen then again prod details are repeated.

▷ Thus ~~update anomaly~~ ⇒ Update anomaly.

Is there delete anomaly?

▷ Only salesman ~~S~~ S_3 has sold product P_3. Say salesman S_3 leaves. If we delete salesman S_3 ~~S~~ info related to P_3 is also deleted ⇒ Delete anomaly.

Functional dependencies of the relation:

sl_no → sname

~~sl_no →~~

pno → pr_name } partial FD

pno → unit_price }

sl_no → city }

city → status } transitive FD

sl_no, pno → qty_sdd → full FD

derived as sl_no → status

1NF

- all attributes are atomic in nature.
- Relation is in 1NF.

2NF

- should be in 1NF
- no partial dependency.
- Has partial dependency so not in 2NF.

Now divide the relation. It should be lossless

↓

no attribute in original relation must be left out. It should be present in atleast one of the subdivided relations.

SalesPerson (sl_no, sname, city, status)

Product (pno , pr_name, unit_price)

Salesman_Prod (~~sl~~ sl_no, pno , qty_sold)

Page No.

Date

FABER-CASTELL

Now check:
Is there ~~insert~~ any Anomaly?
~~Sales~~ Salesman_Prod → no anomalies
Product → no anomalies.
SalesPerson → update anomaly present.
→ say there are multiple salesman from same city, ~~then update~~ then city and status gets repeated. Thus 'a city has a particular status' such info: is repeated ⇒ Update anomaly.

3NF
- should be in 2NF
- no transitive FD.

SalesPerson (sl_no, sname, city)
CityStatus (city, status)
} No more anomalies.

Thus the relation has been divided into:
Salesman (sl_no, sname, city)
Product (pno, pname, unit_price)
Salesman_Prod (sl_no, pno, qty_sold)
CityStatus (city, status)

28/11/24/Thu.

Boyce Codd Normal Form (BCNF)

- Higher form normal form.

Conditions:

1. Must be in 3NF
2. every determinant is a candidate key.

eg: Stud_Mark(S#, semail, C#, mark)

This is in 3NF.

(mark depends on (S#, C#) or (semail, C#)).

Overlapping candidate keys.

S# ⟶ semail

S#, C# ⟶ mark

we take S# as one of the key attribute

or.

semail ⟶ S#

semail, C# ⟶ mark.

↳ if semail is taken.

Now check if in BCNF?

not candidate ✗ S# ⟶ semail

S#, C# ⟶ mark.

↳ candidate key.

So we split,

Stud(S#, semail)

SM(S#, C#, mark)

Now in BCNF.

Functional Dependency.

FD Closure: (F+)

Given a FD set, the FDs that are infered or derived from that, those are called FD closure.

Amstrong's Axioms: (Rules)

~~Reflexivity Rule~~

1. Reflexivity Rule:
   α - set of attributes
   ~~$\beta \subseteq \alpha$ then~~
   $\beta \subseteq \alpha$ then $\alpha \rightarrow \beta$ holds

2. Augmentation Rule:
   $\alpha \rightarrow \beta$ holds and γ is a set of attributes
   $\gamma\alpha \rightarrow \gamma\beta$ holds.

3. Transitivity Rule:
   $\alpha \rightarrow \beta$ & $\beta \rightarrow \gamma$ then $\alpha \rightarrow \gamma$ holds.

Additional Rules:

1. Union rule:
   $\alpha \rightarrow \beta$ & $\alpha \rightarrow \gamma$ then
   $\alpha \rightarrow \beta\gamma$ holds

2. Decomposition rule:
   $\alpha \rightarrow \beta\gamma$ then
   $\alpha \rightarrow \beta$ & $\alpha \rightarrow \gamma$ holds.

3. Pseudo transitivity rule:
   $\alpha \rightarrow \beta$ holds & $\gamma\beta \rightarrow \delta$ holds then,
   $\gamma\alpha \rightarrow \delta$ holds.

Q. Consider the relation

$R = (A, B, C, D, E)$
$F = \{A \rightarrow B, CD \rightarrow E, A \rightarrow C, B \rightarrow D, E \rightarrow A\}$
find F^+?

A: Using transitivity rule:

1. $A \rightarrow B$ & $B \rightarrow D$ $\Rightarrow$ $A \rightarrow D$
2. $CD \rightarrow E$ & $E \rightarrow A$ $\Rightarrow$ $CD \rightarrow A$
3. $E \rightarrow A$ & $A \rightarrow B$ $\Rightarrow$ $E \rightarrow B$
4. $E \rightarrow A$ & $A \rightarrow C$ $\Rightarrow$ $E \rightarrow C$

Using Union:

1. $A \to B$ & $A \to C \Rightarrow A \to BC$

Using Pseudotransitivity rule:

1. $A \to C$ & $CD \to E \Rightarrow AD \to E$
2. ~~$B \to D$~~ & $CD \to E \Rightarrow$ ~~$BD \to E$~~ $BC \to E$

Thus F^+ is:

$F^+ = \{A \to D, CD \to A, E \to B, E \to C, A \to BC, AD \to E, BC \to E\}$

We generally do not use augmentation rule because it is applicable almost everywhere.. too much trouble.

eg: $A \to C$ possible thus
$BA \to BC$
$DA \to DC$
: etc possible.

Q. $R(A, B, C, D, G, H, I)$

$F = \{A \to B, A \to C, CG \to H, CG \to I, B \to H\}$

A: Using transitivity rule:

1. $A \to B$ & $B \to H \Rightarrow A \to H$

Union:

1. $A \to B$ & $A \to C \Rightarrow A \to BC$
2. $CG \to H$ & $CG \to I \Rightarrow CG \to HI$

Pseudotransitivity rule:

1. $A \to C$ & $CG \to H \Rightarrow AG \to H$
2. $A \to C$ & $CG \to I \Rightarrow AG \to I$

$F^+ = \{A \to H, A \to BC, CG \to HI, AG \to H, AG \to I\}$

Attribute Closure: (α^+ / A^+)

The attributes which can be derived from a particular attribute based on a functional dependency.

Algorithm to find it:

$\alpha_i^+ = \alpha$
repeat
  for each fd $\beta \to \gamma$ in F
    if $\beta \subseteq \alpha^+$ then $\alpha^+ = \alpha^+ \cup \gamma$
until α^+ stops changing.

Q. $R = (A, B, C, D, G, H, I)$
$F = \{ A \to B, A \to C, CG \to H, CG \to I, B \to H \}$
compute $(AG)^+$?

A:

$\alpha^+ = AG$ - 1st iteration

$A \to B$ $\quad A \subseteq AG$ so,
$\alpha^+ \cup B$

$\alpha^+ = \{AG, B\}$

$A \to C$
$\alpha^+ = \{AG, B, C\}$

$B \to H$
$\alpha^+ = ACGBH$

$CG \to I$
$\alpha^+ = ABCGHI$

$AG^+ = AG$
$= ABCG \quad (\because A \to B, A \to C)$
$= ABCGH \quad (\because B \to H)$
$= ABCGHI \quad (\because CG \to I)$

If all attributes are present for an attributes closure, then that attribute is a ~~candidate key~~ superkey.

compute A^+ ?

$A^+ = A$
$= ABC \quad (\because A \to B, A \to C)$

★ To check whether $\alpha \to \beta$ holds we check α^+. If β is present in α^+ then $\alpha \to \beta$.
Using this we can also find FD^+.

Thus Applications of α^+:
1. To find super keys.
2. To check whether a $\alpha \to \beta$ holds
3. To find FD^+.

compute A^+ ?

$A^+ - A$

$= ABC$ ($\because A \to B$, $A \to C$)

★ To check whether $\alpha \to \beta$ holds we check α^+. If β is present in α^+ then $\alpha \to \beta$.

Using this we can also find FD^+

Thus Applications of α^+:

1. To find super keys.
2. To check whether a $\alpha \to \beta$ holds
3. To find FD^+.

8/2/24/Thursday.

Canonical Covers / Minimal set:

Assume a relation $R = (A, B, \ldots)$

& F.D. $F = (\ldots)$

When we do updations on relations, they should also satisfy the FDs.
But if the FD is a large set then it becomes a tedious task. So instead of checking through

all, we check only some of them.
Such a set of FDs is called canonical cover or minimal set. Such a minimal set has the same closure as the FD set.

↳ How to find the canonical set?
↳ Steps:
1. $\alpha \rightarrow \beta\gamma \Rightarrow \alpha \rightarrow \beta$ & $\alpha \rightarrow \gamma$
2. Check for extraneous attribute in any FD.
3. Check for redundant FD.

extraneous attribute:
if in a FD $\alpha\beta \rightarrow \gamma$ ~~then~~ and if $\beta \rightarrow \gamma$ then α is extraneous.

eg. Find canonical cover for the following FD set?

Q1. $R = (A, B, C)$
$FD_s = \{ A \rightarrow BC, A \rightarrow B, B \rightarrow C, AB \rightarrow C \}$

A:
* $A \rightarrow BC \Rightarrow A \rightarrow B$, $A \rightarrow C$
$AB \rightarrow C$ & $B \rightarrow C$ so A is extraneous.

↳ $\{ A \rightarrow B, A \rightarrow C, A \rightarrow B, B \rightarrow C, AB \rightarrow C \}$
Taking A as extraneous. Taking only $B \rightarrow C$
↳ $\{ A \rightarrow B, A \rightarrow C, A \rightarrow B, B \rightarrow C, B \rightarrow C \}$

Page No. Date
FABER-CASTELL

Now removing ~~redundancies~~ repetitions,

$\{ A \rightarrow B, A \rightarrow C, B \rightarrow C \}$
Removing redundancy.

$\{ A \rightarrow B, B \rightarrow C \}$ → This is the canonical cover.

Q2. $R = (A, B, C, D, E)$
$FD_s = \{ A \rightarrow BC, CD \rightarrow E, B \rightarrow D, E \rightarrow A \}$

A: $A \rightarrow BC$; $A \rightarrow B$, $A \rightarrow C$

$\{ A \rightarrow B, A \rightarrow C, CD \rightarrow E, B \rightarrow D, E \rightarrow A \}$
$CD \rightarrow E$ → there is no extraneous attribute here.

Canonical cover is the same as the given FD.

Q3. $R = (A, B, C, D)$
$FD = \{ B \rightarrow A, D \rightarrow A, AB \rightarrow D \}$

A: ~~AB~~
$\{ B \rightarrow A, D \rightarrow A, AB \rightarrow D$
$B \rightarrow A$ by rule of augmentation,
$BB \rightarrow AB \Rightarrow B \rightarrow AB$ & we have $AB \rightarrow D$
this says $B \rightarrow D$ (transitivity rule)
↓ B alone can determine D

$AB \rightarrow D$
↳ A is extraneous. So only need to write $B \rightarrow D$.

Page No. Date
FABER-CASTELL

$\{B \to A, D \to A, B \to D\}$

$B \to D$ & $D \to A \Rightarrow B \to A$ → redundant.

$\{B \to D, D \to A\}$ → canonical cover.

Relational Algebra: (RA)

Query languages
- Procedural
  - eg: RA, PL-SQL
- Non procedural
  - eg: SQL.

procedural:
- we specify what is required & how to get it in the query language.
- when we add procedural features to SQL it becomes PL-SQL.

Non-procedural:
- We only specify what is required
- eg: SQL.

Sort of like Machine language + Pgmming language.
↓ PL-SQL. ↓ SQL

Every SQL query is converted into RA format. One query can be converted into many RA format. The system will choose which is the best RA query. The part which decides this is called Query optimisation.

For each command in SQL there are corresponding operators in RA.

Operators in RA
- Basic operators
- Derived operators / Extended Operators

Basic Operators:

1) select (σ)
- $\sigma_{condition}(relation)$ $\quad \sigma_{condition}(Relation)$
- eg: Emp (eno, ename, sal, dno)

select * from Emp where sal > 50000

$\sigma_{sal > 50000}(Emp)$

$\sigma_{sal>50000 \wedge dno='D_1'}(Emp)$

↳ for more than one condition we use logical operators:
$\wedge$ - and $\vee$ - or etc.

2. Project (π)
To select specific attributes within a relation.
Select (σ) selects the whole relation.

$\pi_{col_1, col_2, \ldots}(Relation)$

eg: $\pi_{ename, sal}(Emp)$

↳ name & sal of all employees from Emp.

☆ Find name & sal of those employees working in dno = 'D1' & getting salary > 50000.

$\pi_{ename, sal}\left(\sigma_{sal>50000 \wedge dno='D_1'}(Emp)\right)$

3. Cartesian Product (x)
$R_1 \times R_2$

$\pi_{cname, ename}(Customer \times Emp)$

Find employee names along with corresponding dnames using cartesian product.
Emp (eno, ename, sal, dno)
Dept (dno, dname)
Customer (eno, cname, sal)

$\pi_{ename, dname}\left(\sigma_{Emp.dno = Dept.dno}(Emp \times Dept)\right)$

Find the customer names getting sal > that of employees using cartesian product.

$\pi_{cname}\left(\sigma_{Customer.sal > Emp.sal}(Customer \times Emp)\right)$

4. Union (U)
$R_1 \cup R_2$ R_1 & R_2 have same no. of attributes & corresponding attribute have same domain/data type.

5. Set Difference (−)

$R_1 - R_2$ — R_1 & R_2 are union compatible.

6. Rename (ρ) ...also

$\rho(R_2, R_1)$ — R_1 renamed to R_2

$= \rho(R_2, \Pi_{col1, col2}(R_1))$

↳ col1 & col2 of R_1 is taken & renamed to a new relation R_2.

7. Assignment ($\leftarrow$)

$R_1 \leftarrow Q_1$

$R_2 \leftarrow Q_2$

olp of query Q_1 is assigned to relation R_1.
" " " Q_2 " " " " R_2.

In RA we are not able to do union, differ. etc in query so we assign them to relation to do the operation. eg: $R_1 \cup R_2 \Rightarrow$ olp of $Q_1 \cup$ olp of Q_2.

Extended / Derived Operations:

1. Intersection ($\cap$) $R_1 \cap R_2$
2. Conditional Join ($\bowtie_c$)
   Another way to replace cartesian product.

$\Pi_{ename}(\sigma_{Emp.sal > Cust.sal}(Emp \times Customer))$

⇓ using conditional join.

$\Pi_{ename}(Emp \bowtie_{Emp.sal > Cust.sal} Customer)$

* $\Pi_{ename, ename}(Emp \bowtie Cust)$
  ↳ natural join

3. Outer join
   - left ($\bowtie$)
   - right ($\bowtie$)
   - full ($\bowtie$)

Same as SQL outer join

~~4. Division ($\div$)~~

4. Division (÷) conditions ↓

1. B ⊂ A
2. result attsi = attsi. A − attsi. B
3. result tuples = tuples from A which are associated to every B's tuple.

eg. Student_Sports (sno, sname, sports_time)
Sports (sports_time)

Sports

sports_item
X_1
X_2

Student_Sports

sno	sname	sports_item
S_1	aaa	X_1
S_1	aaa	X_2
S_2	bbb	X_2
S_3	ccc	X_2

Here Sports ⊂ Student_Sports.
result attsi = (sno, sname)
result tuples = (S_1, aaa)
↓
only S_1 is associated to both X_1 & X_2 (all attsi in Sports)

5/12/24 Thursday.

Views, index & sequence → self study.

PL-SQL

- Procedural language capabilities added to SQL.
- plpgsql - the procedural language for postgres.

Syntax:

- var_name := value; ← assigning value to a variable syntax.
- var_name % ~~data~~type; ← declaring a variable
  var_name % rowtype;
  ~~var_name %~~ ~~rec~~
  var_name record;

eg: a Emp.eno % type; → a is of same data type as eno of Emp

eg2: emp Emp % rowtype; → emp is a variable which can hold a whole row from Emp table.
emp can hold a tuple of Emp table only while empl can hold a tuple of any table.

eg3: empl record; → record → one tuple
→ empl can hold one tuple of any table. Does not have a specific structure.

- IF condition then
  statement;
  ~~ELSE~~
  ELSIF condition then
  statement;
  ELSE
  statement;
  END IF;

Not case sensitive.

← IF ~~format~~ syntax.

- Looping Statement

1. Loop
   statement;
   End Loop;

2. While condition
   loop
   statement;
   end loop;

3. for i in [reverse] start .. end → is optional
   loop
   statement;
   end loop;

→ eg: for i in 1..10
;
for i in reverse 1..10

Structure of PLSQL pgm:

<< label >>
[Declare
  declare statements;] → optional
[Begin
  SQL statements;] → core part of pgm
End;
<< label >>

Each pgm must have atleast one begin & end pgm.

~~eg:~~ Proper format of plpgsql ↓

Create function fn_name (arg list) returns return datatype as
$ label $
Declare
  declare statements;
Begin
  SQL statements;
  Return variable;
End;
$ label $
~~L~~anguage 'plpgsql';

Q. Write a function which accepts two integer values & returns their sum?

A:

```
Create function sum (a int, b int)
                returns int as
```

used instead of $label$ <- $$

```
    Begin
        return a+b;
    End;
$$ Language 'plpgsql';
```

OR

```
Create function sum( int, int )
                returns int as
$$
  Begin
        return $1 + $2 ;
  End;
$$ 'plpgsql';
```

$1, $2, ... → positional parameters
→ we do not need to declare variables though this. 1st i/p → $1 ... etc.

Q. Consider an Emp table with eno, ename & sal ~~which accepts~~ write a function which accepts an employee numbers and returns their sal.

A:

```
Create function emp_sal (emp_no Emp.eno%type)
                returns int as
$$
  Declare
        sal1    Emp.sal %type;
  Begin
        select sal into sal1 from Emp where eno = emp_no;
        return sal1;
  End;
$$
Language 'plpgsql';
```

an employee

Ex: Create function emp_sal (emp_no Emp.eno%type)
returns int as

```
$$
Declare
    sal1  Emp.sal %type;
Begin
    select sal into sal1 from Emp where eno = emp_no;
    return sal1;
End;
$$
Language 'plpgsql';
```

6/12/24 Friday

Emp(eno, ename, sal, dno)

```
Create function emp_fn (IN emp_no int,
                        OUT emp_name varchar,
                        OUT dept_no varchar);
```

IN & OUT can let us avoid usage of 'returns'.

Q. Write a fn which accepts an eno & returns the name & sal of that emp?

A: Create function EmpNameSal(IN Eeno int,
OUT empname varchar,
OUT emp_sal ~~varchar~~
Emp.sal%type) as

No need to declare variables (IN, OUT used)

$$
Begin
select ename into emp_name from Emp
where ~~Eno~~ eno = Eeno;
select sal into emp_sal from Emp
where eno = Eeno;
End;
$$
~~Language~~ 'plpgsql';

can be written as single select statement.

select ename, sal into emp_name, emp_sal from Emp where eno = Eeno;

★ By default; if OUT not mentioned, whatever is IN is considered as OUT.

★ INOUT → accepts & returns.

Q. Product(pno, pname, unit_price)
Emp(eno, ename, sal)
Emp_Sales(sl_no, eno, pno, qty_sold)

Q. Write a function which accepts an eno & checks whether the employee is eligible for getting commission. Only if an employee has sold a quantity of 1000 or more in total is eligible for getting commission? assume: one emp sells many prod. one prod sold by many emp.

A:
Create function Eligible (IN emp_no Emp.eno%type → even if you do not write it, it will be there by default
OUT eligibility varchar) as

$$
Declare
Sale Emp_Sales.qty_sold%type;
Begin
can also write query without group by & just using 'where' { select sum(qty_sold) into sale from Emp_Sales Group by eno having eno = emp_no;
If sale > 1000 then
eligibility := 'Eligible';
else
eligibility := 'Not eligible';
end if;
End;
$$
Language 'plpgsql';

Q. Student (sno, sname)
Course (cno, cname)
Stud_Course (sno, cno)

* For each course max: 60 students can register.

1. Write a function to register a student to a course only if the current no: of students does not exceed the limit.

A: Create function Register (IN stud Student.sno%type
IN cou Course.cno%type
OUT msg varchar) as

```
$$
Declare
  tot_cou int ;
  sc Stud_Course %rowtype ;
```

Q. Student (sno, sname)
Course (cno, cname)
Stud_Course (sno, cno)

* For each course max: 60 students can register.

1. Write a function to register a student to a course only if the current no: of students does not exceed the limit.

A: Create function Regist (IN stud Student.sno%type
IN cou Course.cno%type
OUT msg varchar) as

$$
Declare
tot_cou int ;
sc Stud_Course %rowtype ;

12/12/24 Thursday

Trigger

Emp(eno, ename, sal, dno)
Dept(~~deptno~~ dno, dname, emp-count)

We need emp_count to increase each time a new tuple is added to Emp.
We use trigger function for this.

Diff: the trigger function & normal PLSQL function is that trigger function occurs automatically unlike PLSQL fn where we have to call it

Two parts of Trigger fn: (we write it together only)
→ Event
→ Action

Syntax:
Create trigger tri_name {insert/update/delete/truncate or a combination combined using or operator}
{before / after} event
[of col_name]
on tab_name [for each row / statement]
[when cond]
what action to be executed. ← execute procedure fn_name ();

for each row:
for each of the affected rows the action occurs once.

statement:
The action part occurs only once irrespective of no: of rows affected.
By default it is statement.

eg: (The qn to the left) event: new row inserted to Emp
action: emp-count +1 for dno = dno
Create trigger emp_update after insert on Emp for each row
execute procedure emp_countUpdate();
action

using insert we can insert many rows so we use for each row

Now we need to write this fn.

Function associated with trigger has no arguments.

```
Create function emp_count_update()
    returns trigger as
$$
Begin
  Update Dept set emp_count = emp_count+1
  where dept_no = NEW.dno;
  returns NEW;
End;
$$
Language 'plpgsql';
```

→ Triggers associated functions do not return anything other than trigger

newly added dno.

NEW, OLD etc.

Record Parameters is they are case sensitive.

↓ if delete or truncate then return OLD.

if insert return NEW

if update return NEW or OLD.

Q. Emp(eno, ename, sal)
Emp_Dependent(eno, dep_no, dname, relationship)

Whenever an employee leaves the company, the corresponding value in Emp_Dependent table should be deleted.

A: Trigger:

```
Create trigger Emp_leave after
  delete on Emp for each row
  execute procedure Emp_Dep_Del();
```

```
Create function Emp_Dep_Del()
    returns trigger as
$$
Begin
  Delete from Emp_Dependent where eno = OLD.eno;
  returns OLD;
End;
$$
Language 'plpgsql';
```

- Enforcing business rules
- Audit operations
- Input-data Validations

↳ uses of trigger.

Q. Emp(eno, ename, sal, dno)
Emp_Dependent(eno, dep_no, dname, relationship)
Sal_Audit(eno, old_sal, new_sal, user_id, date_time)

Whenever the sal of any employee changes, a corresponding row must be inserted into the Audit table called Sal_Audit which has the eno, old_sal, new_sal, the user who has performed the action and the date & time of updation.

A: Create trigger sal_update after
  Update of sal on Emp for each row
  execute procedure AuditUpdate();

```
Create function AuditUpdate()
    returns trigger as
$$
Begin
    Insert into Sal_Audit values
    (OLD.eno, OLD.sal, NEW.sal, USER, now());
    return OLD;
End;
$$
Language 'plpgsql';
```

USER → system variable & case sensitive.

can use NEW as well

on current_timestamp — these fns give current date & time

Q. Consider Emp table. Whenever any user performs an insertion, updation or deletion on Emp table, an audit of that operation must be inserted into a table Emp_Audit which has attributes like operation, user and the date&time.

Emp_Audit (operation, user, date_time)

- if insertion operation : I
  updation " : U
  deletion " : D
- TG_OP - system variable
  - returns I for insertion, U for updation & D for deletion.
  - case sensitive

A: Create trigger Emp_Operation ~~after~~ before Delete or Insert or Update on Emp
execute procedure Emp_Audit_Update();

before — can be after as well.

~~(Using~~ (Using before instead of after).
~~(You can use either for this Q).~~
(You can use after as well)

```
Create function Emp_Audit_Update()
    returns trigger as
$$
Begin
    Insert into Emp_Audit values
    (TG_OP, USER, current_timestamp);
    return NULL;
End;
$$
Language 'plpgsql';
```

↳ we did not use OLD/NEW because for insertion only NEW for deletion only OLD. Thus choose NULL.

Input Data Validation ~~in~~ using Trigger:

```
If NEW.~~ename~~ = NULL then
    Raise Exception 'Name cannot be null';
End if;
```

Input Data Validation using Trigger:

```
Create function emp_audit() returns
        trigger as
$$
Begin
If NEW.ename = NULL then
   Raise Exception 'Name cannot be null';
End if;
If NEW.sal <= 0 then
   Raise Exception 'salary cannot be zero or negative';
End if;
End;
$$
Language 'plpgsql';
```

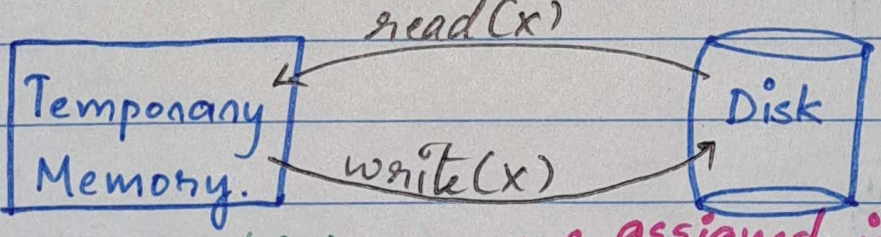
Once an exception is raised, that event will not happen.

DBMS continued.

13/12/24 | Friday.

Transactions:

- Performs some logical task
- Has a number of steps and the task must be executed in the order of the steps.
- read(x) } Two main instructions
  write(x) } of a transaction.
- When a transaction is being executed, the system will assign a temporary space to it until it is executed.



- eg: Read(A) → A is assigned in Temp Mem
  A := A-100 → computation occurs in temp space
  write(A) → new value updated in disk
  :
  commit; → checks if all updations have been made onto the disk.

Why we commit?
If some failure occurs ~~in~~ while the transaction is occuring we need to make sure that after it occurs completely every updations are reflected in the disk if not it should be aborted. It is ~~an~~ for assurance.

Transaction Properties

A	C	I	D
Atomicity	Consistency	Isolation	Durability

Atomicity:
- This makes sure that the transaction either occurs completely or it should not occur at all.
- Completely or none are executed

How do we do it?
(2 methods.)
1 → Undoing the operations executed
2 → Restricting ~~some~~ operations that are executed

- Log Record: This transaction is executing this instruction. Using this we can know which all instructions have been executed and ~~ensume~~ ensure atomicity.

Consistency:
- ~~Makes~~ Every transaction has its own consistency conditions.
- DB ensures that it is always consistent before and after a transaction.
- Serial execution helps in maintaining consistency.

Isolation:
- Transactions:

T_1	T_2	T_3
read(A)	---	---
---	read(A)	---
write(A)	---	read(A)
	write(A)	---
		write(A)

- In the example above, it could create inconsistency in DB.
- Isolation makes sure that this does not occur.

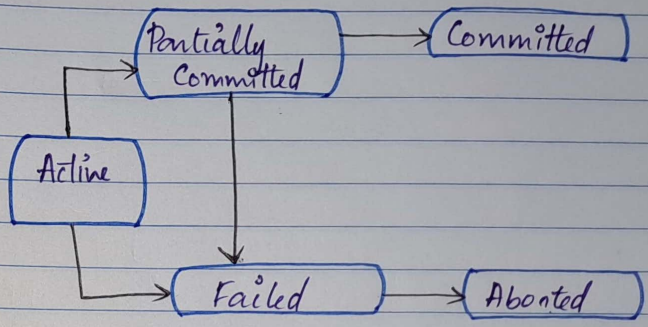
- Even when multiple transactions occur concurrently, they are reading only the updated value.
- T_2 is only allowed after T_1 is done & similarly T_3 is only allowed after T_2 is done. Thus avoiding inconsistency.

Durability:

- Whatever transaction updates MUST BE REFLECTED in the disk copy, if the transaction has been completed.

Who ensures these properties?
↳ Transaction Manager: The component in database system which ensures these properties during transactions.

Transaction States:



Active: transaction has started.
Partially Committed: all instructions except the last commit instruction has been executed.
Failed: An interruption occurred and the transaction did not go through.
Committed: Commit statement has also been executed and all the disk changes have been made.
Abort: We roll back all the changes made during the failed transaction & go back to the state it was before the transaction occured.

16/12/24 Monday.

Serial Execution:

- The transactions occur one after another.
- The order of serial execution of transactions is called Serial Schedule.

eg:

T_1	T_2
read(A)	read(A)
A := A-50	temp := A×0.1
write(A)	A := A-temp
read(B)	write(A)
B := B+50	read(B)
write(B)	B := B+temp
	write(B)

A=1000
B=2000

1. $T_1 \rightarrow T_2$

A = 1000
A = 950 →
B = 2000
B = 2050 ↗

T_2
A = 950
temp = 95
A = 855
B = 2050
B = 2145

Thus we get final value A = 855 & B = 2145

2. $T_2 \rightarrow T_1$

A = 1000
temp = 100
A = 900 →
B = 2000
B = 2100 →

T_2
A = 900
A = 850
B = 2100
B = 2150

Thus we get final values as
A = 850 B = 2150

Consistency here is:
in both cases database is in a consistent state which is the sum of A & B before and after execution is the same in 6th.

No: of serial schedules = n!
n - no: of transactions.

Disadvantages:

1. At a time either the CPU or the temp memory is busy. The System Utilization is less.
2. The throughput is less. Throughput is the

no: of processes executed per unit time.

Although consistent, these ~~disadvantages~~ disadv: of serial execution makes us do concurrent execution.

Concurrent Execution.

T_1	T_2
read (A)	
A := A-50	
write (A)	
	read (A)
	temp := A * 0.1
	A := A - temp
	write (A)
read (B)	
B := B+50	
write (B)	
	read (B)
	B := B + temp
	write (B)

time ↓

We do not wait until the whole transaction is done before going to next ~~no~~ transaction

A = 1000
A = 950 → A = 950
temp = 95
A = 855

B = 2000
B = 2050 → B = 2050
B = 2145

This leaves the database in consistent state

Not all concurrent execution leaves the database in a consistent state.

How does the database know which ~~is~~ concurrent schedule is consistent?

The database only allows those concurrent schedules who have a result same as a serial schedule. Such schedules are called ~~Serialization concurrent schedules.~~ Serializable schedule.

Conflict Serializability:

To check for serializability in a concurrent schedule.

Conflicting Instructions:

Two instructions are conflicting if they act on the same data item and one of them is a write operation.

T_2 \ T_1	read(A)	write(A)
read(A)	X	✓
write(A)	✓	✓

✓ - conflicting
X - not conflicting

T_1	T_2
read(A)	
write(A)	read(A)
read(B)	
~~read~~ write(B)	write(A)
	read(B)
	write(B)

Check whether we can swap two instructions belonging to two diff: transactions

1. write(A) & read(A) → conflicting?
   → Yes
   → So no swapping.

2. read(A) & read(B) → conflicting?
   → No.
   → Then swap.

T_1	T_2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

write(B), write(A)
↓
conflicting?
↓
No
(diff: data item)
→ Swap.

↓ Now it becomes

T_1	T_2
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

read(A) & write(B)
↓
swappable (diff: data items)

for that we use Precedence graph.

↓

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

This is a serial → execution.
Thus the initial concurrent transaction is a conflict serializable schedule.

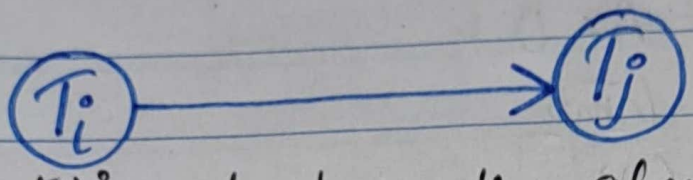
But this is not feasible for long transactions & when no. of transactions is more

for that we use Precedence graph.

19/12/24 / Thursday.

Precedence Graph:

R(Q) - read (Q)
W(Q) - write(Q)



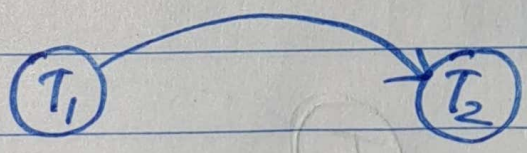
Conditions to draw the above directed edge:

1. T_i executes R(Q) before T_j executes W(Q)
2. T_i executes W(Q) before T_j executes R(Q)
3. T_i executes W(Q) before T_j executes W(Q)

If no cycles are present after making the graph, the transactions are conflict serializable.

eg:

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

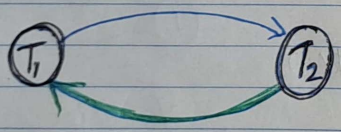


No cycles
↓
conflict serializable.

Q.

T_1	T_2
read(A)	
A := A − 50	
	read(A)
	temp := A * 0.1
	A := A − temp
	write(A)
	read(B)
write(A)	
read(B)	
B := B + 50	
write(B)	
	B := B + temp
	write(B)

A.



Cycle is present so not ~~serial~~ conflict serializable.

Q.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
	commit
failed abort	

Irrecoverable Schedules.

A: T_2 is a transaction which depends on T_1. But T_1 has failed and aborted. But we cannot ~~recover~~ rollback a committed transaction.

Such transactions which cannot be recovered are called irrecoverable schedules.

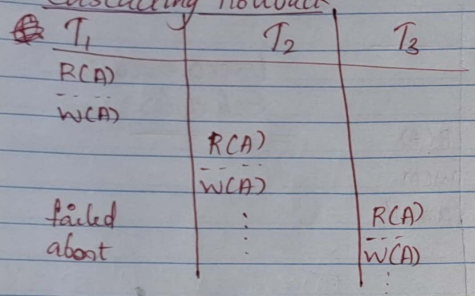
But we need recoverable schedules.

Recoverable Schedules:

The reading transaction should ONLY COMMIT AFTER the writing transaction has committed.

↑ The ~~oh~~ only condition needed to be followed.

Cascading Rollback



Here T_2 depends on T_1 and T_3 depends on T_2.
But T_1 has failed and aborted. So we need to abort or fail T_2 & T_3.
This condition of one transaction failing causing many other transactions to fail is called cascading rollback.
But we want our schedules to be cascade-less.

Cascade-less Schedule:

The writing transaction's commit should happen BEFORE the reading transaction's reading operation.

All cascade-less schedules are recoverable schedules but the vice-versa need not be always true.

Concurrency Control:

How is isolation (one of the ACID properties) property implemented / ensured?
If we allow transactions to occur without concurrency control, the transactions need not have isolation.
One of the concurrency control methods is:

1. Lock-based concurrency control:
   - When one transaction ~~is~~ must ~~using~~ use a data item, the transaction must request a lock on the data item.
   - The concurrency control manager is responsible for locking the data item & when one data item is locked by one transaction, the other transaction cannot lock it.

Locks
- Shared lock (S(A))
  - reading operation.
- Exclusive Lock (X(A))
  - for updating values
  - write operation (read operation)

T_2 \ T_1	S(A)	X(A)
S(A)	✓	X
X(A)	X	X

Multiple transactions can have shared lock on same data item granted by the concurrency control manager.

After using the data item the transaction must unlock the data item as well.

T_1	T_2
lock-X(A)	
read(A)	
A := A-50	
W(A)	
Unlock(A)	
	lock-S(A)
	lock-S(B)
	read(A)
	read(B)
	display(A+B)
	unlock(A)
	unlock(B)
unlock(A)	
lock X(B)	
read(B)	
B := B+100	
W(B)	
unlock(B)	

Value displayed = 250.

But if we move unlock(A) later the value changes.

If on T_2 will wait until T_1 is completely done (and then only continue on T_2).

Dead lock:

Drawback for locking protocol.

T_i waiting for T_j to unlock a data item & T_j waiting for T_i to unlock a data item.

Dead lock:

Drawback for locking protocol.

T_i waiting for T_j to unlock a data item & T_j waiting for T_i to unlock a data item.

21/12/24/Saturday

~~Hardline~~

Starvation Situation:

Another drawback of lock based concurrency control.

$T_1 \longrightarrow S(A)$
$T_2 \longrightarrow X(A)$ ✗
$T_3 \longrightarrow S(A)$ ✓
$T_4 \longrightarrow S(A)$ ✓
:

Here T_2 waits indefinitely meanwhile every other shared lock requiring transactions are done.

Conditions to avoid Starvation:

1. If currently requesting transaction's lock is compatible with ~~this~~ ~~then~~ lock.
2. If ~~there~~ ~~die~~ no other transaction which is waiting for the locking the same data item.

Lock is granted only if these conditions are true. Then starvation can be avoided.

Two Phase Locking Protocol (2PL):

1) Growing Phase - transactions gain locks
2) Shrinking Phase - locks are released.

- Concurrency control mechanism based on locking mechanism.

~~Not acquiring~~

★ No more lock requests can be made in shrinking phase.

★ Lock-Point:

The point where the transaction has attained its last lock.

That is; upto that is growing phase & after that is the shrinking phase.

Q.

T_1	T_2
lock X(A)	
R(A)	
A := A-50	
W(A)	
	lock_S(A)
	lock_S(B)
	display(A+B)
	unlock_A
Lock Point → lock_X(B)	unlock_B
R(B)	
B := B+50	
W(B)	
unlock_A	
unlock_B	

Transaction T_1 is in 2PL protocol. Every lock occurs and only after that unlocking happens.

Similarly transaction T_2 is also in 2PL.

Adv: • A schedule in 2PL is always conflict serializable.

Drawback: • Not free from Deadlock.
• There is a chance of cascading rollback.

Deadlocks:

• If more chances of deadlock - Prevention of Deadlock

• Less chances of deadlock - Detection & Recovery from Deadlock.

↓

let it happen & recover from it.

Deadlock

Prevention

Detection & Recovery.

ensure no cyclic wait

Preemption & rollback

↳ Drawback:
- Unnecessary transaction rollbacks.

↓ To avoid ex: 'time stamps'.

Lock all data items at beginning itself

Ordering of data items

drawback
- low data/resource utilization
- hard to predict which all data items are required by a transaction from the beginning.

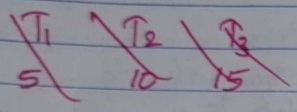
Ordering of data items.

or assign unique value to
- We number the data items
- Transactions are only able to request for data items in the increasing number of unique values.

Preemption & Rollback:

When one trans. is holding onto a data item, that data item is preempted & given to the other trans: which ~~requires~~ requested the data item And the first transaction undergoes rollback.

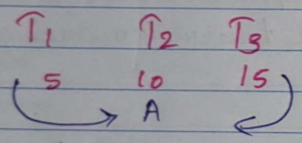
Timestamp:
- to avoid the drawback.
- a unique value assigned to transaction
- one method: system ~~time~~ clock time
  2nd method: counter usage.
- Based on the timestamp we have two deadlock prevention methods.
  - Wait - die
  - Wound - wait.



Wait-die:

T_i is allowed to wait only if its timestamp is less than T_j (T_i is older than T_j). Else T_i dies (roll back).

$T_i \longrightarrow T_j$ (T_i requests for a data item used by T_j)



$T_1 \longrightarrow T_2$
(A) (A)

$5 < 10 \rightarrow T_1$ waits.

$T_3 \longrightarrow T_2$
(A) (A)

$15 > 10 \rightarrow T_3$ rollsbacks.

Wound-wait:

T_i is allowed to wait only if its timestamp is greater than T_j.

Else

T_j rollsback and allocates its data items to T_i.

$T_i \longrightarrow T_j$

~~Time-out Based Scheme:~~

~~* When a transaction has rolled back & restarted, it will be assigned the same time stamp.~~

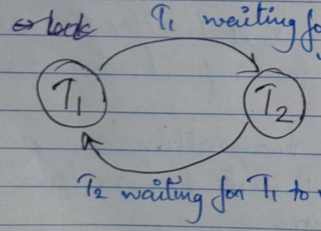
Time-out Based Scheme:

- The transaction rollsback if the data item is not allocated after waiting for a particular amount of time.
- When a transaction has rolled back & restarted, it will be assigned the same time stamp.

Detection & Recovery Of Deadlock:

T_1	T_2
lock_X(A)	
R(A)	
W(A)	
:	
	lock_S(B)
	R(B)
	...
	lock_S(A)
lock_X(B)	:

Wait-for graph:



Which transaction to be rolled back in this case?

Factors:

1. How long has the transaction been running for?
2. How many data items is the transaction locking?
3. How many transactions are dependent on this transaction?

After this the system chooses whether to do:

- Complete rollback
  - Easier
  - Rolling back all the way to the beginning
- Partial rollback
  - Must have other info: (such as until where it rolled back) saved.
  - Rolling back to a particular point.

There is a chance of starvation in this method.

Solution: How many times has the

transaction undergone a roll back:
This condition is also checked while selecting the victim transaction to be rolled back.