DBMS | Page No.
Date / /

DBMS

Database Management system

⇓

collection of inter related data & their relationships } Database system

handled → set of programs (collection of operations) } DBMS

→ Operations for handling data

- create
- insert
- update
- modify
- delete
- append

→ Problems / Redudancies in DBMS
↳ Wrong info should not be present

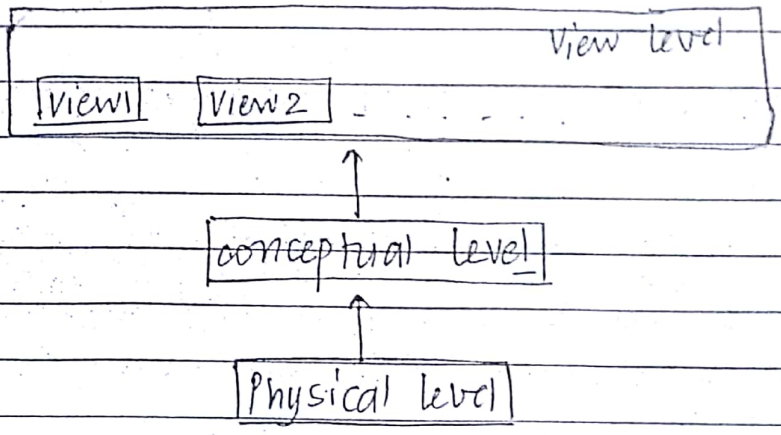
info should not be present at many places unless necessary

- Data redundancy and inconsistency
- Difficulty in accessing data
- Data isolation → Each data should be independent of another
- Integrity problem
- Atomicity problem → In case of system failure also, data in the d/c should remain consistent
- Concurrent access problem
- Security problems

DOMS | Page No. | Date / /

→ Views of Data (DATA ABSTRACTION)

1. Physical view → ~~logical~~ physical level (how data is stored)
2. Logical view → conceptual level (what data is stored & relationships.)
3. Views → user level / view level (applications ~~programs~~ of how it will be used)



* Physical level data can only be changed by the administration.

→ Instances and Schemas → Overall design of a database

~~Database information~~
at a particular pt. of time

Schema

	S.NO	NAME	CLASS	ROLL NO
instance →				
(entries in table)				

Date / /

Instances & Schemas are as follows

1. physical schema → physical level — DDL
2. logical schema → logical level
3. Subschema → highest level / view level / intermediate level (DML)

all the work done on physical and logical level is done through subschema ∴ it is known as intermediate level.

DOMS | Page No. | Date / /

→ Data independence

↓

ability to change physical logical schemas without affecting their data definition.

* Physical data independence: ability to change physical schema definition without causing the application program to be re-written.

* Logical data independence: same definition physical ⟷ Logical

→ Data Models

* Object oriented data Model: used for describing data in all the 3 levels

Submodels of object oriented/based data model:

=> Entity relationship model
=> Object oriented model (collection of objects → variables which will be called) [C++]
=> Binary model:
=> Semantic data model
=> functional data model

* Record based data Model

=> Relational model (relations/relationships) (tables)
=> Network model (linked lists)
=> Hierarchical model (collection of nodes)

* Physical data model

⇒ Unifying model

⇒ Frame. memory model

→ Database languages

DDL (Data Definition Language) → data dictionary
meta data
data storage

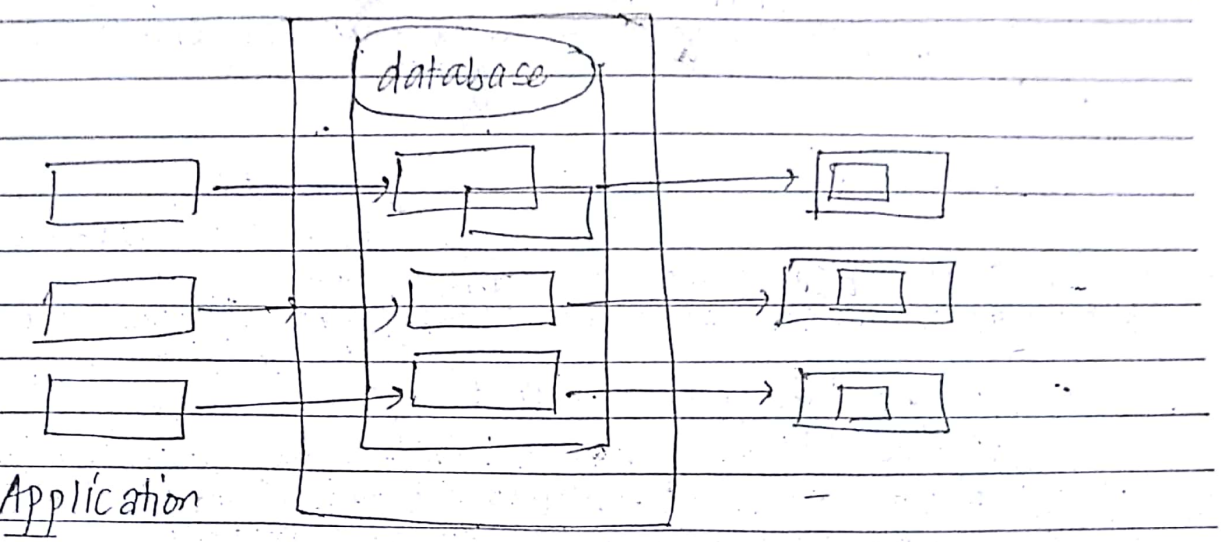
DML (Data Manipulation Language)

Procedural query language
- procedure to get data is defined
- Relational algebral

Non-Procedural query language
- data is defined but procedure is not
- Tuple relational calculus
- Domain relational calculus

DOMS | Page No.
Date / /

→ Data, hardware, software, users } Components of DBMS



→ Transaction Management

⇓

requirement : all transactions / operations should be independent.

Transaction manager manages the operations

→ Storage Management

Storage manager is an application program that manages storage.

Functions of storage manager:

(i) Interaction with file manager
(ii) Integrity Enforcement
(iii) Security Enforcement
(iv) Backup & Recovery
(v) Concurrency control

→ Data base administrator

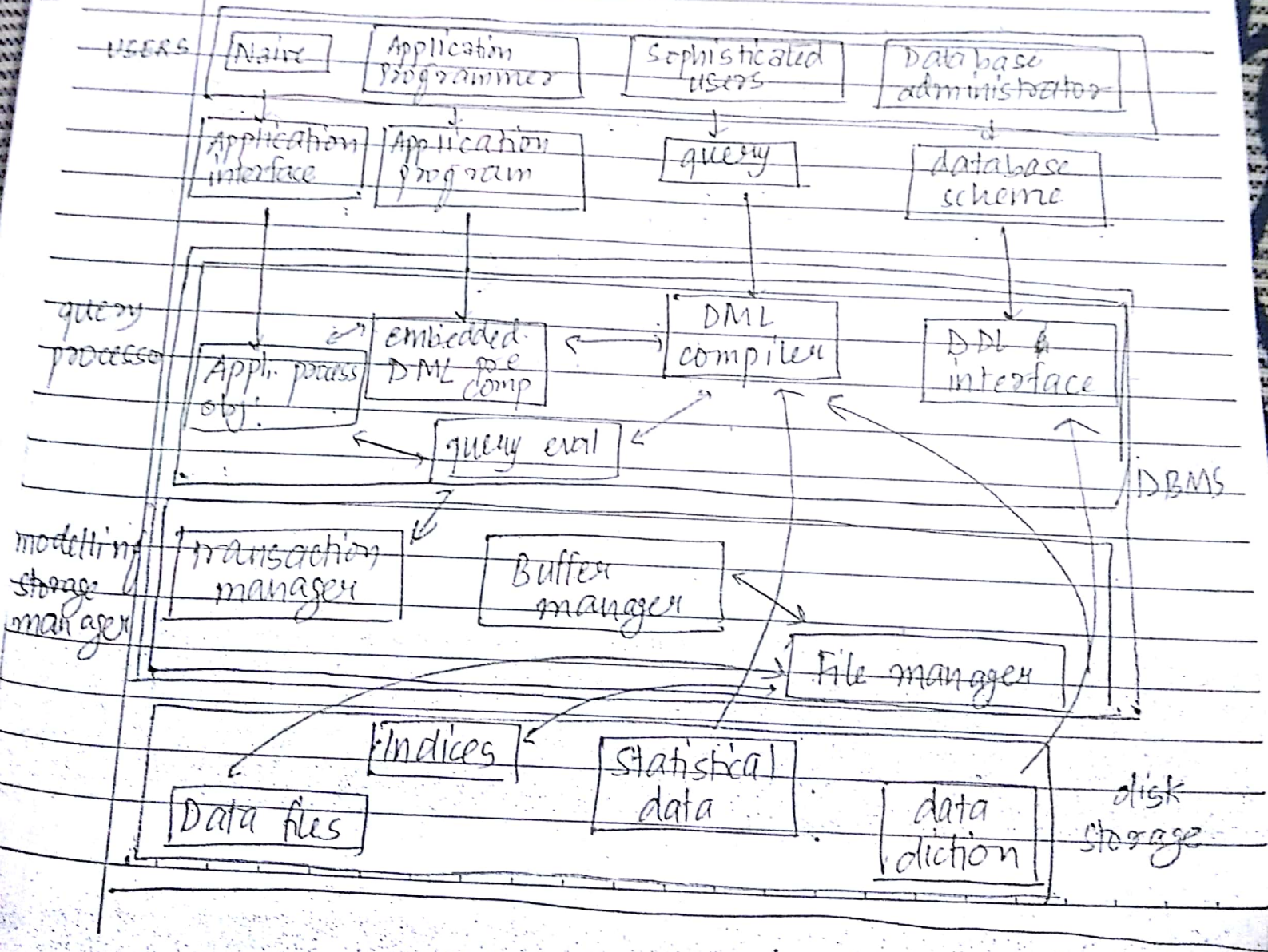
- gives schema definition
- decide storage structure
- decide access methods
- give physical organization
- give grantings of authorization
- define integrity constraints

DOMS

→ TYPES OF USERS

1. Application Programmers → do the coding part
2. Sophisticated users → interact with the system without writing the application (query language)
3. Unsophisticated users → specialised users which design the
   - database framework
   - knowledge base
   - expert systems
   - environment modelling
4. Naive users
   ↓
   just use the application (users of my application)

→ OVERALL STRUCTURE OF DBMS

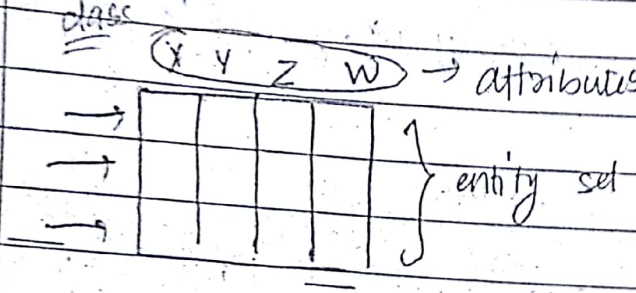


→ ER Model (Entity Relationship Model)

- entity sets
- relationship sets
- attributes

* entity → object / thing

* entity set → collection of information

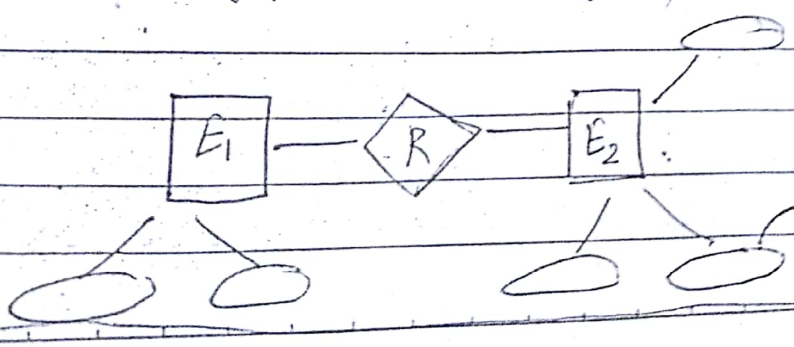


* attributes → describing items

* relationship sets → coln. of info in relationships b/w diff. entities.

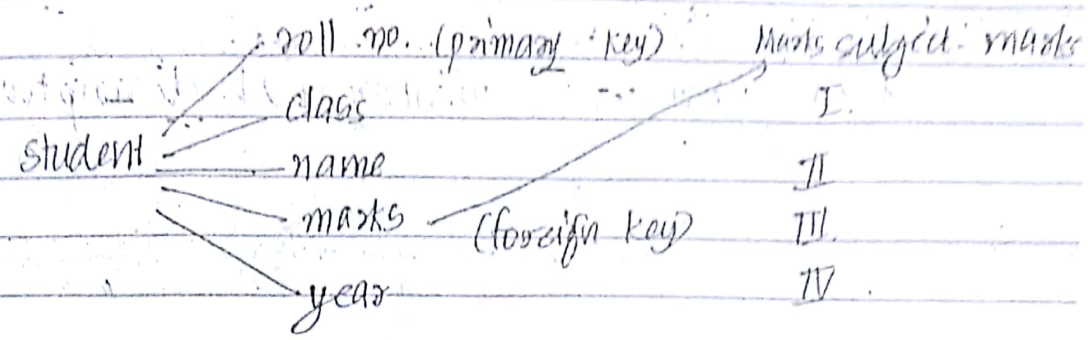
Relationships / Mapping

One to one	One to Many	Many to one	Many to Many
(↔)	(←)	(→)	(–)



DOMS | Page No.
Date / /

Eg:



Super-key : roll no. + class
+ name
+ marks
+ year

candidate key : class
class + name
marks + year.

→ Super-key should contain primary key always.

→ Relationship Set

entity — relationship

attributes : prim(E)UR → all keys of relationship.
↓
primary key
of entity set

primary key : primary (E) ∨ primary (R)

If ans no primary(R) exists

then primary : primary(E) ∨ descriptive attributes of R.

↑ attributes to describe R

If there is more than one entity then

primary : primary(E1) ∨ primary(E2) . . . ∨ R

⇒ If there is no primary key of the entity set then we say that it is a WEAK ENTITY SET.

DOMS | Page No.
Date / /

→ Attributes

① Simple and Composite attribute

Roll no. — simple

collection of sub attributes → customer name → first, middle, last

OR

first name.
surname.

② Single valued and multivalued

single → cannot have more than one value (Loan number)

multivalued → phone no. dependants.

③ Null attribute → dependants (may be a person has no dependent - father, mother ...)

↓

Null value

NAN → not applicable

④ Derived attribute

↓

calculated

age → present date - date of birth

experience → present date - date of joining

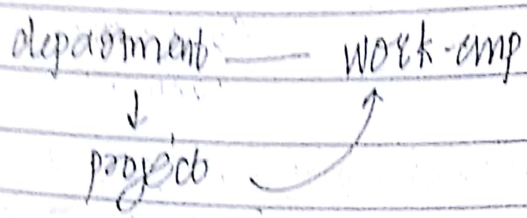
Relationships b/w entities

2 entities → binary

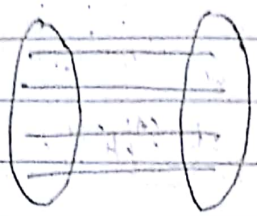
3 → ternary

:

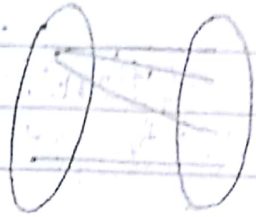
n → n-ary



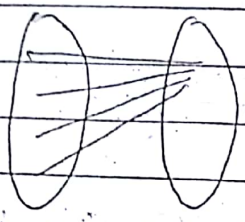
→ Mapping cardinality.



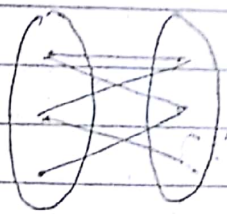
one - one



one to many



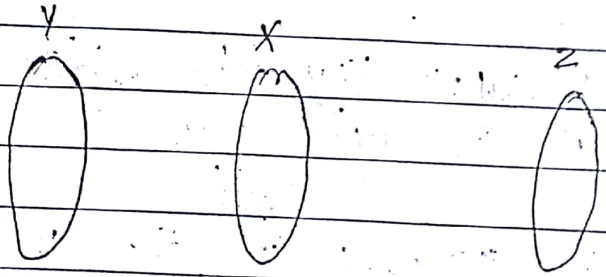
many to one



many to many

→ Existence dependancy

⇒ Loan entity : depends on customer (if he has taken loan or not)



TOTAL RELATIONSHIP

customer loan entity payment

↓

depends on loan entity

⇒ if Y is deleted, X & Z will be deleted.

DOMS Page No. Date / /

* Thus, Y → dominant entity
X → subordinate entity

=> Any relationship is said to be TOTAL if atleast one entity set depends on another.

=> Any relationship is PARTIAL b/w entity E & relationship R. if entity E participates in relationship R (not dependent)

→ KEY(S)
↓
attribute(s) in entity set which describes the entity.

=> Super key : set of one or more attributes that is taken collectively & it allows to identify the entity in the entity set uniquely.

=> Candidate key : A super key for which no subset is a super key.

=> Primary key : An attribute in the entity set which uniquely identifies the entity in entity set.
↓
unique key
(cannot be further subdivided)

=> Foreign key : for references.

DOMS | Page No.
Date / /

→ ER model

entity set → ▭ rectangle

attribute → ⬭ ellipse

relationship sets → ◇ diamond

links → lines — (many to many)
→ (many to one)
← (one to many)
↔ (one to one)

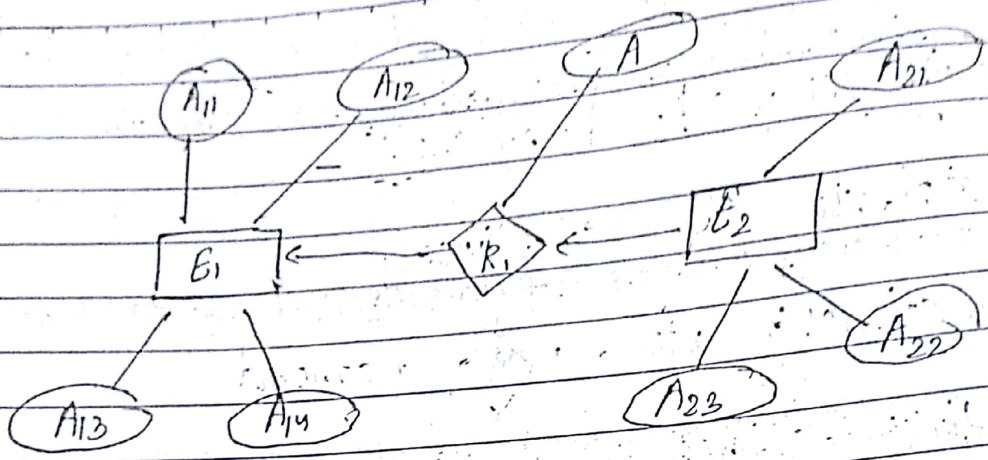
multivalued attribute → ⬭ double ellipse

derived attribute → ⬭ dashed ellipse.

total participation → double line ═

primary key → underlined

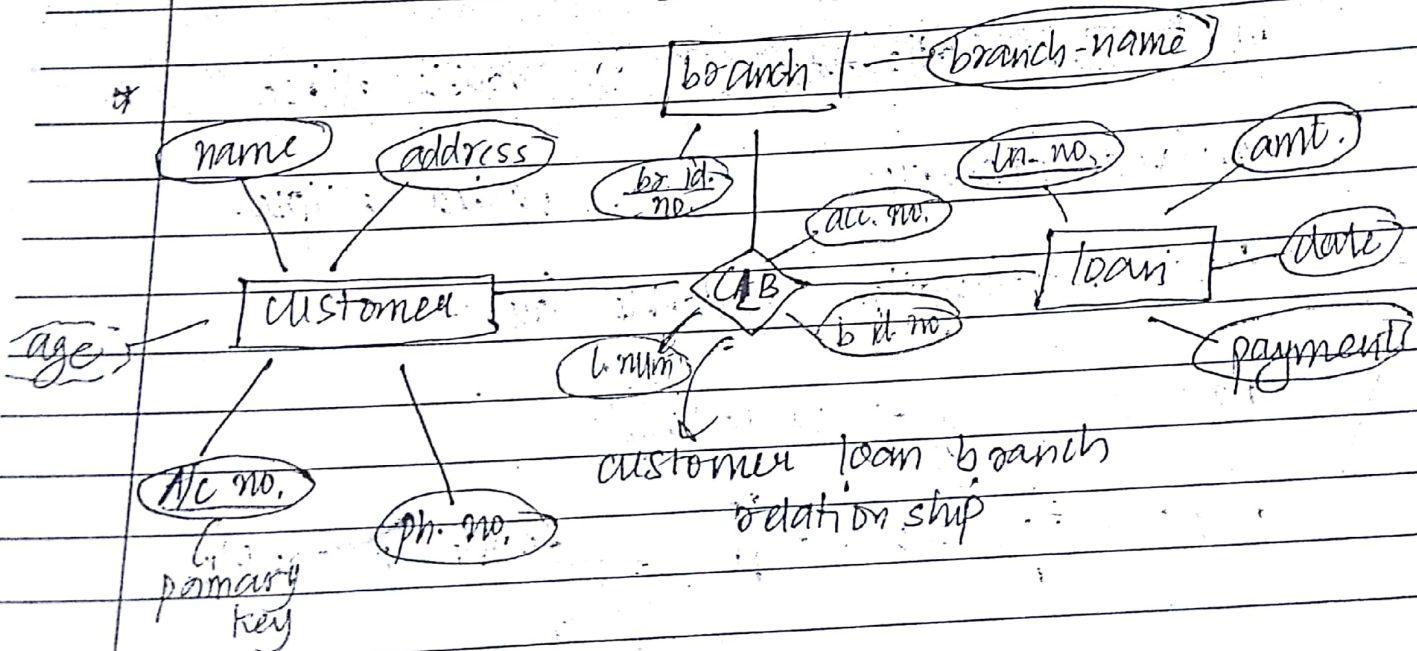
weak entity set → ▣ double rectangle



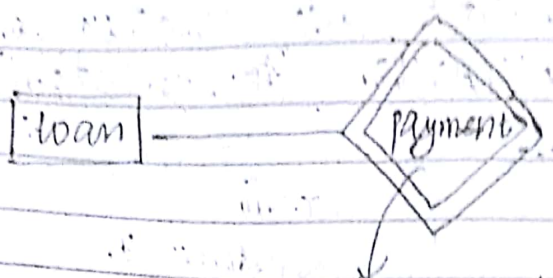
2 entities → binary

3 → tertiary

n → n-ary relationship



payment (weak entity) — pay-date, pay-amt, pay-no.

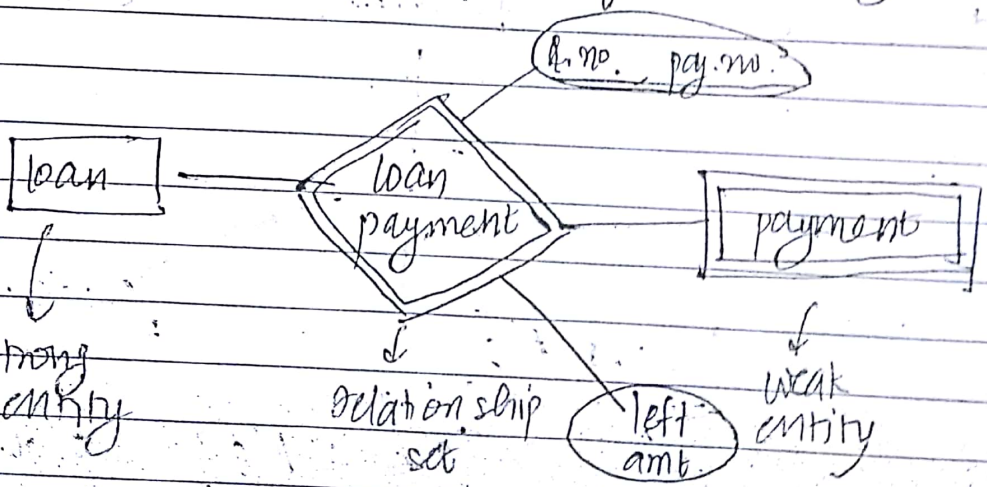


diamond ◇ : it is a relationship set
double ◇ : it is weak.

* no. attribute primary key
↓
weak entity relationship set

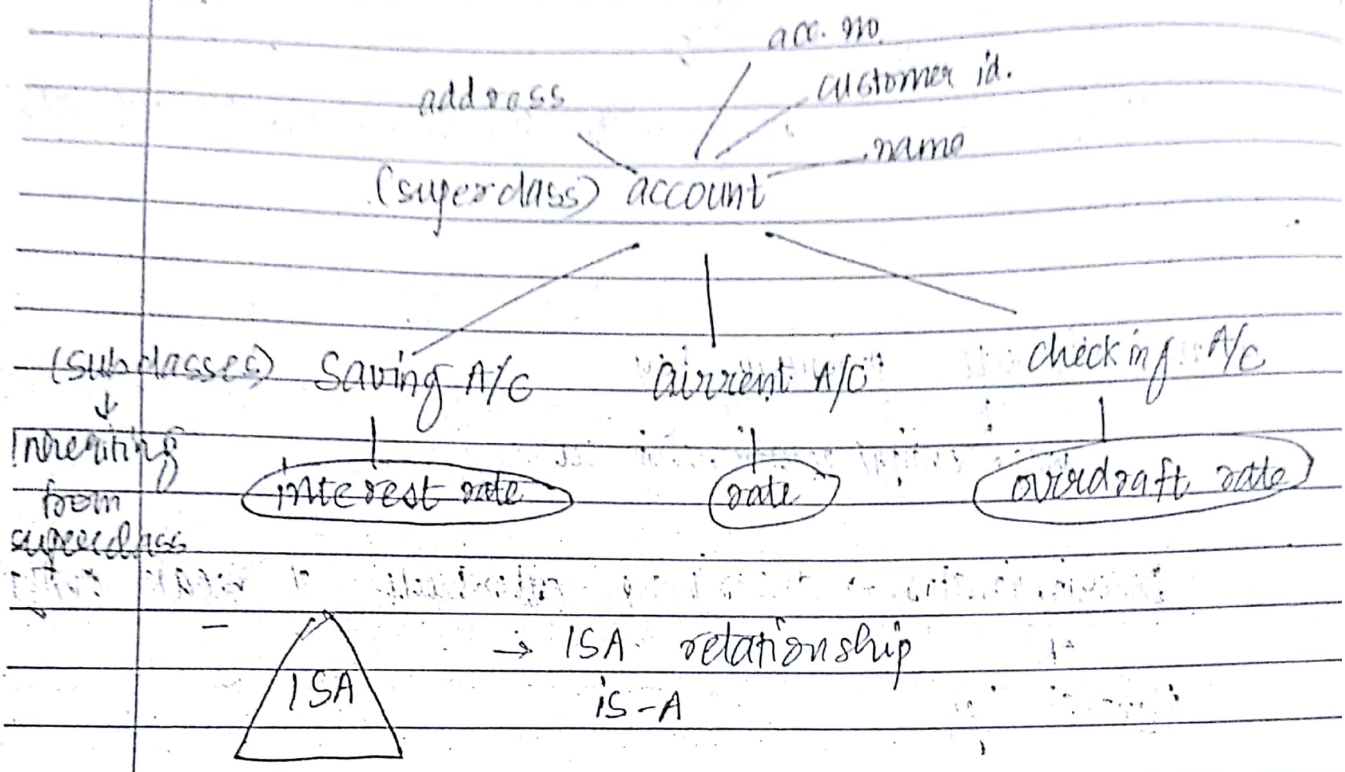
Discriminator → describing attribute of weak entity
OR
Partial key

Primary key of weak entity/ relationship set is discriminator + primary key of the strong entity.

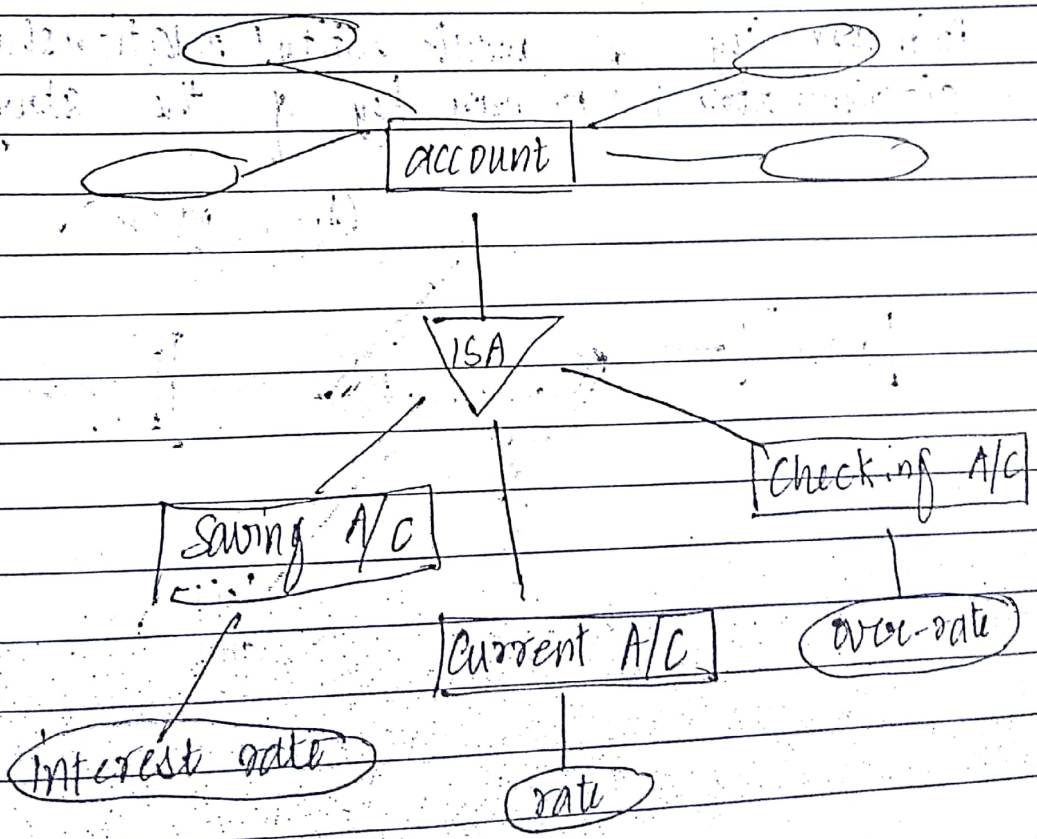


(dividing)

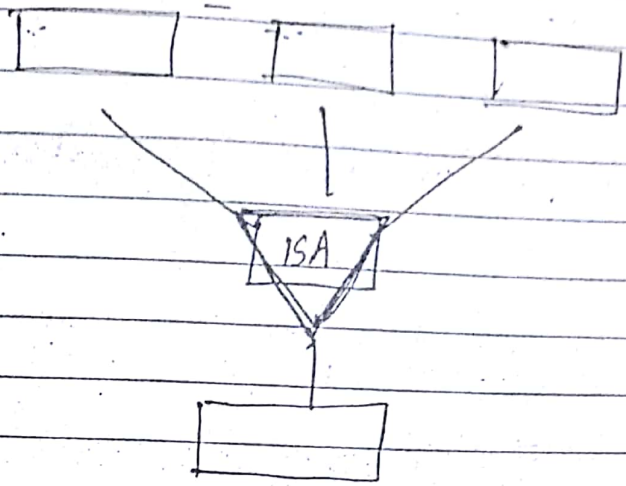
* Subgrouping of entities into another entity is known as SPECIALISATION (top to bottom)



*

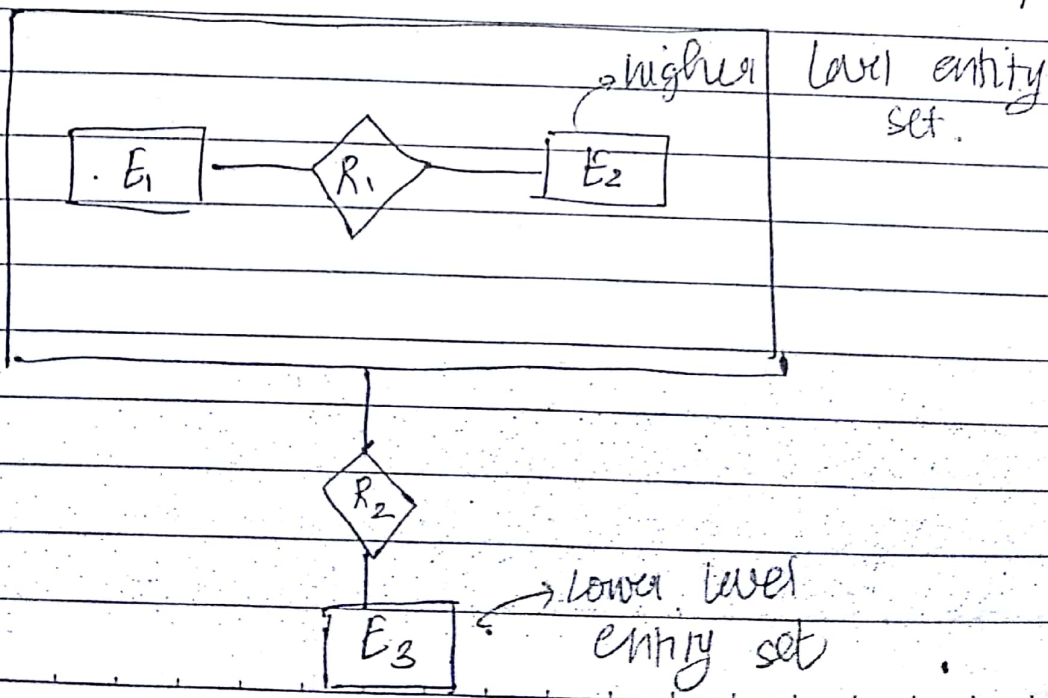


* Combining of groups into one is GENERALISATION (bottom to top)



* If higher level entity is totally involved in lower level entity ⇒ TOTAL - PARTICIPATION
otherwise
PARTIAL

* Aggregation: used to express relationships b/w diff. entities & relationships.



→ Relational Model

- tables
  - attribute → columns
  - entity → table name
  - entity set → table contents

E_1

	a_1	a_2	a_3	a_4
t_1				
t_2				
⋮				
t_m				

rows

size $m \times n$ — columns

Domain of attributes $a_1, a_2 \dots a_n$

$\downarrow \quad \downarrow \quad \downarrow$

$D_1 \quad D_2 \dots D_n$

Domain = $D_1 \times D_2 \times D_3 \dots \times D_n$

Customer

cust. id	cust. name	cust. add	cust. ph. no.
int (Domain)	string	char	int

→ Database schema

Customer schema = (cust-id, cust-name, cust-add, cust-ph.no.);

(schema of table; attributes)

Customer (cust-id, cust-name, cust-add, cust-ph.no);

KEYS

{cust-id} is primary key.

* Small alphabet = relational table
Capital " = representation of schema

* tupple value : $t_1[k]$

(k → attribute of key)

$t_1[k] = t_2[k]$ (If 2 customer-id's are same or not)

$t_1[k_1] = t_2[k_2]$

↳ changed both key & attribute

In relationship set

* If an entity does not have any descriptive attribute, then it will contain primary keys of all the other entities.

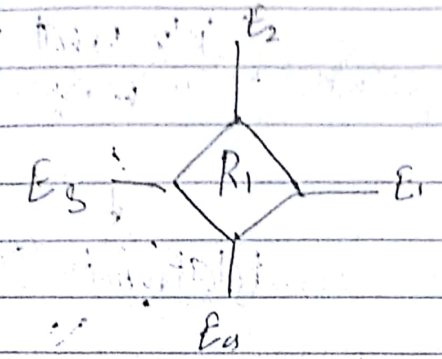


Table : R_1

K_1	K_2	K_3	K_4

primary keys of E_1, E_2, E_3, E_4

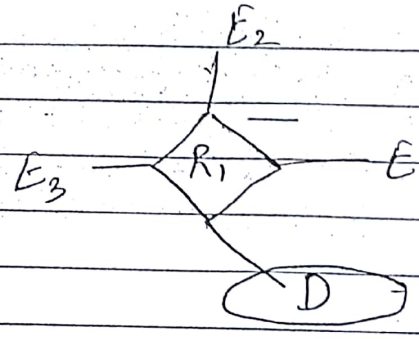
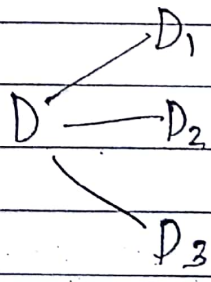


Table R_1

K_1	K_2	K_3	D

* In case of multivalued attributes all the attributes belonging to the entity form a column in the table.



(3 phone nos.)

multivalued attribute

→ QUERY LANGUAGES

Procedural

(What we want & how will we get it)

Relational Algebra

Non-Procedural

(Only what we want)

Relational Calculus

Tuple

Domain

→ RELATIONAL ALGEBRA

∞ Set-difference : $r - s$

(a) r & s should have similarity.
(b) domain of r & domain of s must be same for all.

∞ Cartesian product : renaming :→ ρ (rho)

$(r \times s)$ $\quad$ $\boxed{\rho_x(E)}$

gives name x to exp. E.

will combine all the columns of the table.
(borrow x loan)

For renaming attributes : $\rho_x(A_1, A_2 \ldots A_n) : E$

new name of attributes

→ RELATIONAL CALCULUS

Tuple / Domain

$\{t \mid P(t)\}$

tuple Predicate

$\{t \mid t \in loan \wedge t[amount] > 1500\}$

* $\exists t \in r(Q(t))$

⇓ same

there exists, t in your relation r which satisfies predicate $Q(t)$

$\{t \mid \exists s \in loan(t[l.number] = s[l.number] \wedge s[amount] > 1500)\}$

* $\forall$: for all

* $\{t \mid \exists s \in borrower(t[cust\_name] = s[cust\_name]) \wedge \forall u \in depositor(t[cust\_name] = u[cust\_name])\}$

Properties to be satisfied in tuple calculus:

(a). $s \in r$

where s is a tuple variable & r is a relation

(b). $s[x] \; \theta \; u[y]$ → comparison attribute

s & u are tuple variables, x & y are attributes

(c). $s[x] \; \theta \; c$ → operator

s is tuple variable, x is attribute & c is constant.

2015 | Page No.
Date / /

* atom formula is build up of atoms

Rules:

(i) An independent atom is a formula

(ii) If P_1 is a formula, then (negation of P_1) [$\neg P_1$ or $\neg(P_1)$] are also formulas

(iii) $P_1 \vee P_2$
$P_1 \wedge P_2$
$P_1 \rightarrow P_2$
} are also formulas.

(iv) If $P(s)$ is a formula, then $\exists s \in r[P(s)]$ is also a formula. $\forall$

→ Safety of expression

Any expression $\{t \mid P(t)\}$ is safe if all values that appear in the results are values in the domain of t.

→ DOMAIN RELATIONAL CALCULUS

$\{ \langle x_1, x_2 \ldots x_n \rangle / P(x_1, x_2, \ldots x_n) \}$

$x_1, x_2 \ldots x_n$: domain variables

P : predicate

∴ Any formula is an atom.

(a) $\langle x_1, x_2 \ldots x_n \rangle \in r$

(b) $x \ominus y$

$\ominus$: comparison operator ($<, >, <=, >=, \neq, =$)

(c) $x \ominus c$

x : domain variable

c : constant

* FORMULA :

(i) Any atom is a formula.

(ii) If P_1 is a formula then $\neg P_1$ is also a formula

(iii) If P_1 & P_2 are formulas, then $P_1 \vee P_2$, $P_1 \wedge P_2$, $P_1 \rightarrow P_2$ are formulas

(iv) If $P_1(x)$ is a formula, the $\exists x (P_1(x))$, $\forall x (P_2(x))$ are also formulas.

DOMS Page No. Date / /

A formula is safe expression if:

(i) All the values that appear in the tuples of expression are the values from dom(P).
(domain(P))

(ii) For every $\exists x(P_1(x))$, the sub formula is true if there is a x in dom(P_1) s.t. $P_1(x)$ is true.

(iii) For every "for all" sub formula of the form $\forall x(P_1(x))$, the sub formula is true if & only if $P_1(x)$ is true for all the values of x from dom(P_1).

→ How queries are written?

$\{ \langle b, l, a \rangle \mid \langle b, l, a \rangle \in loan \wedge a > 1500 \}$

(b: branch no., l: loan no., a: amt.)

$\{ \langle l \rangle \mid \exists b, a \{ \langle b, l, a \rangle \in loan \wedge a > 1500 \} \}$

Names of customers who have loan

$\{ \langle C \rangle \mid \exists l (\langle C, l \rangle \in borrower \wedge \exists b, a (\langle b, l, a \rangle \in loan \wedge b = "ABC"))$
$\vee \exists a (\langle C, a \rangle \in depositor \wedge \exists b, n (\langle b, a, n \rangle \in account \wedge b = "ABC")) \}$

→ OPERATIONS

1. Set ~~instruction~~. Intersection :: $r \cap s = r - (r - s)$
2. Natural Join
3. Division operation.
4. Assignment
5. Insert
6. Delete
7. Update
8. Modify
9. Creating views.
10. Aggregate Functions.

=> to find customers who has an a/c as well as have taken a loan.

$\Pi_{cust\text{-}name}(borrower) \cap \Pi_{cust\text{-}name}(depositor)$

(2) Natural Join : binary operation.

$|X|$

↳ combination of cartesian prod. & selection operation.

=> Customers who have taken loan & amt. of loan:

$\Pi_{cust\_name, Lnum, amt}(borrower\ |X|\ loan)$

DOMS | Page No. | Date / /

or Customers who live in city ABC, have taken a loan, have an acc.

$\Pi_{b\_name}(\sigma_{cust\_city = "ABC"}(customer \bowtie account \bowtie depositor))$

* NATURAL JOIN

full outer join ⟗ | left outer join ⟕ | right outer join ⟖

		Matched		
X	X	Null	Null	All entries from left table.
Null	Null	X	X	All entries from right table.

(Not matching - tagged with NULL)

All entries from both left & right table.

* Extension of Natural Join ← Theta join

$r \bowtie_\theta s = \sigma_\theta(r \times s)$

allows to combine cartesian & selection operation as one.

(3) Division : ÷ "for all"

$\Pi_{cust\_name, b\_name}(depositor \bowtie account) \div \Pi_{b\_name}(\sigma_{city = "ABC"}(branch))$

The foll. 2 conditions should be satisfied in $r \div s$:

↳ relation

(i) t is in $\Pi_{R-S}(r)$

↳ tuple ↳ relation schema (attributes)

(ii) for every t_s in S
t_r in r

(a) $t_r[S] = t_s[S]$ (b) $t_r[R-S] = t$

o> Also, division is given by:

$$r \div S = \Pi_{R-S}(r) - \Pi_{R-S}\left(\left(\Pi_{R-S}(r) \times S\right) - \Pi_{R-S,S}(r)\right)$$

(4). Assignment. ($\leftarrow$)

temp1 $\leftarrow \Pi_{R-S}(r)$.

temp2 $\leftarrow \Pi_{R-S}\left(\left[\Pi_{R-S}(r) \times S\right) - \Pi_{R-S,S}(r)\right)$

temp3 $\leftarrow$ temp1 - temp2

DOM5 | Page No. | Date / /

→ AGGREGATE FUNCTIONS

1. sum
2. avg
3. count
4. max.
5. min

} NUMERIC VALUES AS OUTPUT

eg) sum of salaries of part-time workers.

sum_{sal} (pt_work)

eg) count distinct b_name (pt_work);

similar will count only once

* Grouping G (to grp.)

b_name $G_{sum_{sal}}$ (pt_work)

sum of salaries of workers of every branch

* For n-grouping,

$G_1, G_2, G_3 \ldots G_{F_1A_1, F_2A_2 \ldots}$

(5). Insertion : $r \leftarrow r \cup E$ (E: new exp; query; tuple value; union)

(6) Delete : $r \leftarrow r - E$

(7). Update : $r \leftarrow \Lambda_{F_1, F_2 \ldots F_n}(\sigma_{p=}(r) \cup (r - \sigma_p(r)))$

(8). View : used if we want to store temporarily

create view V as <query expr>

Eg :

create view all_abc
as $\Lambda_{cust\_name}(\sigma_{branch\_name = "ABC"}(deposit \bowtie account))$

a) View can be created inside another view.

DOMS | Page No.
Date : / /

SQL

STRUCTURE QUERY LANGUAGE

↳ combination of relational algebra and calculus

SQL

- DDL
- DMI
- Embedded DML
- View definitions
- Authorization
- Transaction control
- Integrity constraints

→ Basic constructs

select $A_1, A_2 \ldots A_n$
from $r_1, r_2 \ldots r_m$
where (P)
↳ predicate

$\equiv \Pi_{A_1 A_2 \ldots A_n}(\sigma_P(r_1 \times r_2 \ldots \times r_m))$

* If we want that attributes / tuples do not repeat:

select distinct A
from r
where P.

* select L_num
from loan.
where b_name = "ABC"

(=, >, <, <=, >=, !=)

* If we want to match attributes of diff. tables:

select l_num acc_no.
from loan branch
where ~~l_num~~ loan.~~acc_no~~ acc_no = branch.acc_no

* Renaming.

select c_name, b_name
from account as A, branch as B

where A.c_name = B.C_name

customer name should be present in both the tables.

* Strings : % (to compare)

_ (for char)

select
from
where b_name : "ABC%"

string starts with ABC & can extend further.

b_name : "___"

string should be of 3 chars.

* like (can be used for strings in place of =)

select
from
where c.city like "%---"

* Ordering

~~sle~~ select ___
from ___
order by amt. desc, t-num asc.

* Union : ~~elimal~~ eliminates duplicates

=> select ( )
union
select ( )

=> select ( )
union (all) → now duplicates are also present
select ( )

* Intersect / Intersect all

* Except / except all (minus)

→ AGGREGATE FUNCTIONS IN SQL

sum
avg
max
min
count

* To find sum of all the balances of accounts in branch XYZ:

```
select sum(bal)
from account
where ~~bal~~ b-name = "XYZ";
```

** for making groups in SQL, use group by

```
select b-name, avg(bal)
from account
group by b-name();
```

(optional)

** To count distinct ~~tuples~~, of those customers who deposited some amount have both an A/C & have ~~taken a loan~~:

```
select b-name, count (distinct cust-name)
from depositor, account
where depositor.acc_num = account.acc_num
group by b-name
```

DOMS | Page No. | Date / /

- With condition

```
select b_name, avg(bal)
from account
group by b_name
having avg(bal) > 10,000;
```

- to count no. of tuples.

```
select count(*)
from account
```

- to find customers who have loan amount as zero

```
select L_num
from loan
where amt. is null;
```

- NESTING

```
select ____
    select ____
        select ____
```

(1) set membership (To put one table in another)
in, not in

Who have both a/c & taken a loan from branch XYZ

```
select distinct cust-name
from borrower, loan
where borrower, loan.l-num = loan.l-num
and b-name = "XYZ"
and b-name, cust_name
    in (select b_name, cust_name
        from depositor, account
        where depositor.acc_num = account.acc_num);
```

can be written using intersection.

* not in : for customers who have taken a loan but do not have an a/c.

(II) Set comparison

>, <, =, >=, <=

>some, <some, =some, >=some, <=some → to compare some tuples

>all, <all, =all, >=all, <=all → to compare all tuples.

Q) find all branch names that have highest avg. balance

```
select b-name
from account
group by b-name
```

having avg (bal) > all
(select avg bal from
from account
group by b_name)

⇒ to find existence of a relationship use
exists, not exists

If there is a relationship b/w a & b, then
a not exists b = b except a.

⇒ To find uniqueness of a relationship:

unique, not unique
↓
there are no
duplicate tuples.

* To find customers who have an a/c only ~~at~~
~~branch "XYZ"~~.

Select T.cust_name
from depositor T // renaming (depositor as T)
where unique (select R.cust_name
from account, depositor R
where T.cust_name = R.cust_name
and R.acc_num = account.acc_num)

as result (c_name)
↳ creates new table result to store ~~the~~ the result
T.cust_name is renamed as c_name.

Result

c-name

→ VIEWS IN SQL

temp. storage

⇒ create view V as <query exp>

⇒ Nesting of views

→ MODIFICATION OF DATABASE

- Insertion
- Deletion

delete from r
where P.

~~insert into table~~

insert into table values ( )

→ update

update account
set bal = bal * 1.25
where bal > select avg(bal) //condition also
from account.

→ JOIN OPERATION

(i). condition join / inner join
(ii) Natural join
(iii) Outer join (left, right, full)

⇒ loan (inner join) borrower on loan.l_num = borrower.l_num
(1st table) (second table) (condition)
as lb (branch, l_num, amt, cust, cust_l_num).

generates new table with the attributes

⇒ loan natural right outer join borrower

all the entries of right table will be there

⇒ loan full outer join borrower using (l_num);

* Find customers who have an a/c but have not taken a loan from any branch.

select d_CN
from (depositor left outer join borrower on
depositor.cust_name = borrower.cust_name)
as db1(d_CN, acc_num, b_CN, l_num)
where b_CN is NULL.

→ DATA DEFINITION LANGUAGE (to define schema)

DOMAIN - char(n)
varchar(n)
int, small int
numeric(p,d) [How many digits & how many decimal places]
real, double
float, date, time

* create table r (A1 D1, A2 D2, ... An Dn)
<integrity constraints 1>
<integrity constraints n>

table name

Attribute 1 → Domain 1

Integrity constraints: Primary key, NULL values, candidate key

→ create table branch (b_name char(15) not null,
b_city char(30),
assets integer,
primary key (b_name)
check (assets >= 0))

DOMS | Page No.
Date / /

→ INTEGRITY CONSTRAINTS

1. Domain constraints.

keyword 'check'

* create table t (A1 D1, A2 D2 ... An Dn)
<integrity const. 1>
<integrity const. n>

* create table b
(
check (bal > 500);

=> keys can also be defined — primary key / foreign key

* create table t
(
check (bal > 500),
primary key (no);

2) Referential integrity

* Two relations r & s ...
$r \bowtie s$.

t_r in r, there is no t_s in s such that $t_r[R \cap S] = t_s[R \cap S]$

↳ Dangling tuples

* Two relations $r_1(R_1)$, $r_2(R_2)$
Primary key K_1, K_2

CONDITION FOR REFERENTIAL INTEGRITY: Subset α of R_2 is a foreign key referencing K_1 in r_1, if it is reqd. that for every t_2 in r_2 there must be a tuple t_1 in r_1 s.t. $t_1[K_1] = t_2[K_2]$.

» To preserve referential integrity in

(i). Insertion: $t_1[K] = t_2[\alpha]$ i.e. $t_2[\alpha] \in \Pi_K(r_1)$ } for any tuple to be inserted

(ii). Deletion: $\sigma_{\alpha = t_1[K]}(r_2)$

(iii). Update — Referenced relation
Referencing relation

⇒ If tuple t_2 is being updated in r_2, then

$t_2'[\alpha] \in \Pi_K(r_1)$

↳ updated tuple

⇒ If r_1 is updated, $\sigma_{\alpha = t_1[K]}(r_2)$

* FOREIGN KEY :

foreign key (b_name) references branch

attribute table

⇒ To define foreign key based on operation:

foreign key (b_name) references branch on delete cascade
on update cascade

used in some SQL languages

→ System Constraints

↓

Assertions : a predicate which expresses conditions which we want that database should always satisfy.

system ⇒ Trigger.

* create assertion <assert_name>
check <predicate>

=> create assertion check_sum
(not exists (select * from branch
where (select sum(amt) from loan
where loan.b-name = branch.b-name)
>= (select sum(amt) from account
where loan.b-name = branch.b-name)))

=> TRIGGER (Condition & action)

define trigger overdraft
on update of account T
(if new T.bal < 0
then (insert into loan
values (T.b-name, T.acc-no - new T.bal)

insert into borrower
(select cust-name, acc-no
from depositor
where T.acc-no = depositor.acc-no)
update account S
set S.bal = 0
where S.acc-no = T.acc-no.))

V. imp:

→ FUNCTIONAL DEPENDENCY

$\alpha \subseteq R$ and $\beta \subseteq R$
R is relation schema
α, β are attributes.

Functional dependency $\boxed{\alpha \rightarrow \beta}$
hold on R if in any legal relation $r(R)$ for all tuples t_1 & t_2 in r,

such that $t_1[\alpha] = t_2[\alpha]$.
It is also the case that
$t_1[\beta] = t_2[\beta]$

* $\boxed{\alpha \rightarrow\rightarrow \beta.}$ (Multivalued func. Dep.)

* loan info = (b_name, l_num, cust_name, amt)
schema — ~~tables~~

~~loan~~ l_num → amt ✓
l_num → b_name ✓
l_num → cust_name ✗
↳ ~~many customers can have same name~~

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_2	b_3	c_2	d_4

$A \to C$ ✓

$C \to A$ ✗

$AB \to D$ ✓

* closure of set of FD's (F+)

~~if there is~~ if $A \to B$, $B \to C$, then $A \to C$ (REFLEXIVITY ~~RULE~~ TRANSITIVITY)

t_1, t_2

$t_1[A] = t_2[A]$

$A \to B$

$t_1[B] = t_2[B]$

=> $R = (A, B, C, G, H, I)$

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

$A \to H$ ✓ ($A \to B$ & $B \to H$)

=> Reflexivity Rule:
If α is set of attributes & $\beta \subseteq \alpha$, then $\alpha \to \beta$ exists.

=> Augmentation Rule:
If $\alpha \to \beta$ holds and γ is set of attributes then

$\gamma\alpha \rightarrow \gamma\beta$ holds

- Transitivity Rule :-
  If $\alpha \rightarrow \beta$ holds
  and $\beta \rightarrow \gamma$ holds, then
  $\alpha \rightarrow \gamma$ holds.

- Union Rule: If $\alpha \rightarrow \beta$ holds
  and $\alpha \rightarrow \gamma$ holds, then
  $\alpha \rightarrow \beta\gamma$ holds.

- Decomposition Rule: If $\alpha \rightarrow \beta\gamma$ holds,
  then
  - $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.

- Pseudo transitivity Rule: If $\alpha \rightarrow \beta$ holds
  and $\gamma\beta \rightarrow \delta$ holds, then
  $\alpha\gamma \rightarrow \delta$ holds.

* $R = (A, B, C, G, H, I)$

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

$F^+ = ?$

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

$A \rightarrow B,\ B \rightarrow H$

$A \rightarrow H$ (Transitivity)

$CG \rightarrow HI$ (Union Rule)

$A \rightarrow C,\ CG \rightarrow I$

$AG \rightarrow I$ (PT Rule)

~~$AG \rightarrow H$~~ → We take only those functional dependencies that do not derive other FD's

→ CLOSURE OF ATTRIBUTE SET (α^+)

α

Let of all the attributes in F^+

Algo:

result = α;
while (changes to result) do:
for each functional dependency
$\beta \to \gamma$ in F do
begin
if $\beta \subseteq$ result then
result = result $\cup \gamma$
end

$AG^+ = \{A, B, C, G, H, I\}$

Take FD's corresponding to A, G

$A \to B \Rightarrow A, B$
$A \to C \Rightarrow C$
$CG \to H \Rightarrow C, G, H$
$CG \to I \Rightarrow I$

→ FINDING CANONICAL COVER (Fc)

Fc is set of dependencies s.t. F logically implies all the FD's in Fc & Fc logically implies all the FD's in F.

repeat

use the union rule to replace any dep. in F of the form
$\alpha_1 \to \beta$ & $\alpha_1 \to \beta_2$ with
$\alpha_1 \to \beta_1 \beta_2$.

Find a FD $\alpha \to \beta$ with extraneous attribute either in α or in β.

If an extraneous attribute is found, delete it from $\alpha \to \beta$.

until F does not change.

* F F.D.
$\alpha \to \beta$ in F.

(1) attribute A is extraneous in (α) if $A \in \alpha$, and F logically implies

$(F - \{\alpha \to \beta\}) \cup \{(\alpha - A) \to \beta\}$

(2) attribute A is extraneous in (β) if $A \to \beta$, the set of FD

$(F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$

logically implies F.

* (A, B, C)

$F = \{ A \to BC$
$B \to C$
$A \to B$
$AB \to C \}$

$F_c = \{ A \to B, B \to C \}$

$A \to BC, A \to B$

(B)

~~$A \to BC$~~, ~~$B \to C$~~

$A \to B, AB \to C$

(A)

→ PITFALLS

- repetition of information
- Inability to represent certain information

How to remove pitfalls?

1. Decomposition

$R = \{R_1, R_2 \ldots R_n\}$

$R = R_1 \cup R_2 \cup R_3 \ldots \cup R_n$

$r \subseteq r_1 \bowtie r_2 \bowtie r_3 \bowtie \ldots \bowtie r_n$

* $r_i = \Pi_{R_i}(r)$

$r \subseteq r_i \bowtie r_2 \bowtie r_3 \ldots \bowtie r_n$

* lending_schema : (b_name, b_city, assets, cust-name, l-num, amt)

Decompose the Lending schema:

branch_schema = (b_name, b_city, assets, cust-name)

loan_schema = (cust-name, l-num, amt)

* loss less join decomposition

↓

no information must be lost while decomposing

→ Desirable properties of decomposition

(i) loss-less join decomposition.

R, F
loss less join decomp. if atleast one of prop. hold in F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

(ii). Dependancy Preservation.

restriction of F to R_i is set of FD's in F^+ that includes only attributes of R_i.

ALGO:

$D = \{R_1\ R_2 - - - R_n\}$
F

compute F^+
for each R_i in D do
begin
F_i = the restriction of F^+ to R_i
end
$F' = \phi$;
for each restriction F_i
begin
$F' = F' \cup F_i$
end

compute F'^+;
if ($F'^+ = F^+$) then
    return true;
else
    return false;

(iii). No repetition of information.

* See examples from book.

NORMALIZATION

↓

decomposition of information

1st normal form → when data does not have any redundant information.

2nd
3rd
BCNF
} based on FD's

4th NF
5th NF
6th NF
} (not in syllabus) based on multivalued FD's

→ BCNF : Boyce-Codd Normal form

R is in BCNF w.r.t.
set of FD's for all FD's in F^+ of the form
$\alpha \rightarrow \beta$ where $\alpha \subseteq R$, $\beta \subseteq R$
atleast one of the properties hold:

- $\alpha \rightarrow \beta$ is a trivial FD
  i.e. $\beta \subseteq \alpha$.
- α is a superkey for schema R.

* Eg : cust_schema = (cust_name, cust_street, cust_city)

cust_name → cust_street, cust_city

* Branch-schema = (b-name, assets, b-city)

b-name → assets, b-city

* loan-info-schema = (b-name, cust-name, l-number, amt)

l-number → amt, b-name (not a BCNF → decomposition)

l-number is not a superkey

l-number, b-name is a superkey.

Decomposition:

loan-schema = (b-name, l-num, amt)

borrower-schema = (cust-name, l-num)

loss less join decomp (now in BCNF)

* Algo

result = {R};
done = false;

compute F+;
while (not done) do
    if (there is a schema Ri in result that is not in BCNF)
    then
        begin
            let α → β be a non trivial FD that holds on Ri, such that

$\alpha \to R_i$ is not in F^+ and $\alpha \cap \beta = \phi$.
result = (result - R_i) $\cup$ (R_i - β) $\cup$ (α, β).
end
else done = true;

→ 3rd Normal Form (3NF) : Weaker Normal Form

Properties:
- $\alpha \to \beta$ is trivial F.D.
- α is a superkey of R
- Each attribute A in $\beta \to \alpha$ is contained in candidate key of R.

1) If any relation is in BCNF, then it is in 3NF always but vice versa is not true.

2) loss less join & dependency preservation decomposition

* Algo:

let F_c be canonical cover of F;
i = 0
for each FD $\alpha \to \beta$ in F_c do
if none of the schema R_j &
j → 1, 2 ... i contains
$\alpha\beta$
then
begin
i = i+1;
$R_i = \alpha\beta$;
end

Date / /

if none of the schemas R_j
$j = 1, 2 \ldots i$ contains
a candidate key for R
then
begin
$v = i + 1$;
R^v = any candidate key for R;
end
return ($R_1 R_2 \ldots R_i$)

MULTIVALUED DEPENDENCY

Relation schema R
$\alpha \subseteq R$, $\beta \subseteq R$
Multivalued depen : $\alpha \rightarrow\rightarrow \beta$
holds on R for all t_1 & t_2
in r such that $t_1[\alpha] = t_2[\alpha]$.

t_3 & t_4 st.

(i). $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$

(ii). $t_2[\beta] = t_1[\beta]$

(iii). $t_2[R-\beta] = t_2[R-\beta]$

(iv). $t_4[\beta] = t_2[\beta]$

(v). $t_4[R-\beta] = t_1[R-\beta]$

	α	β	$R-\alpha-\beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

* $\alpha \rightarrow\rightarrow \beta$ is <u>trivial</u>
if $\beta \subseteq \alpha$ or $\beta \cup \alpha = R$

* D - set of FD's and ~~MVD's~~ MVD's

Closure of D as D^+

(i) Reflexive Rule — $\alpha \to \beta$ holds

(ii) Augmentation Rule — $\gamma\alpha \to \gamma\beta$ holds

(iii) Transitivity Rule — $\alpha \to \gamma$ holds

(iv) Complementation Rule — if $\alpha \to\to \beta$ holds, then $\alpha \to\to R - \beta - \alpha$ holds

(v) Multivalued Augmentation Rule — if $\alpha \to\to \beta$, $\gamma \subseteq R$, $\delta \subseteq \gamma$, then $\gamma\alpha \to\to \delta\beta$ holds

(vi) Multivalued Transitivity Rule — if $\alpha \to\to \beta$ holds, $\beta \to\to \gamma$ holds, then $\alpha \to\to \gamma - \beta$ holds.

(vii) Replication rule — if $\alpha \to \beta$ holds, Then $\alpha \to\to \beta$ holds.

(viii) Coalescence Rule — if $\alpha \to\to \beta$ holds and $\gamma \subseteq \beta$, $\delta \subseteq R$ and $\delta \cap \beta = \phi$, $\delta \to \gamma$ Then $\alpha \to \gamma$ holds

---

* $R = (A, B, C, G, H, I)$

$D = \{ A \to\to B,$
$B \to\to HI,$
$CG \to H \}$

$D^+ = ?$

→ Multivalued union rule

$D^+ = \{ A \to\to CGHI$
$A \to\to HI$
$B \to H$
$A \to\to CG \}$

(ii) Multivalued union Rule

if $\alpha \rightarrow\rightarrow \beta$, $\alpha \rightarrow\rightarrow \gamma$ then
$\alpha \rightarrow\rightarrow \beta\gamma$ holds

(iii) Intersection rule

if $\alpha \rightarrow\rightarrow \beta$, $\alpha \rightarrow\rightarrow \gamma$, then
$\alpha \rightarrow\rightarrow \beta \cap \gamma$ holds

(iv) Difference rule

if $\alpha \rightarrow\rightarrow \beta$, $\alpha \rightarrow\rightarrow \gamma$, then
$\alpha \rightarrow\rightarrow \beta - \gamma$ holds
and $\alpha \rightarrow\rightarrow \gamma - \beta$ holds.

→ 4th NF

↳ Based on Multivalued dependency (MVD)

atleast one of them should satisfy.

* $\alpha \rightarrow\rightarrow \beta$ is trivial
* α is superkey of R.

→ RECOVERY SYSTEM

Transaction failure:

(i). logical error
(ii). System crash
(iii). Disk failure

→ STORAGE STRUCTURE

(i). volatile storage
(ii). non-volatile storage
(iii). stable storage → RAID → 7 levels
(Redundant Array of Independent or Indispensable Disks)

* Main operations related to transactions:

(1) Read
(11) Write

* transferring information from storage to buffer.

* transaction: operation on database
(read, write)
operation / collection of operations.

* Redoing the Transaction
  ↓
  start from the beginning

* Undoing the transaction
  ↓
  start from the pt. where transaction was interrupted

→ RECOVERY

* Log based recovery

(1) Deferred log based recovery.
(2) Immediate log based recovery.

o) Log record : log files which will be stored
   ↓ Structure/class → describes single database

oo) Log record will contain.

- Transaction identifier
- Data item identifier
- Old value
- New Value

oo) After write operation, we are changing the database, thus log record is generated.

- Various ways of defining log record.
- $\langle T_i \text{ start} \rangle$: Transaction T_i has started.
- $\langle T_i, X_j, V_1, V_2 \rangle$: Transaction T_i has performed a write operation on database X_j whose initial value was V_1 & final value is V_2
- $\langle T_i \text{ commit} \rangle$: Transaction T_i has successfully completed / executed.
- $\langle T_i \text{ abort} \rangle$: if transaction T_i has failed / aborted then we have to redo the transaction from start.

→ CONCURRENT TRANSACTION

- Thaoughput
- Utilization
- less idle time
- faster response time
- less seek time
- fast execution time
- less waiting time

- 2 transactions T_1 & T_2

schedule 1 → first T_1 is executed, then T_2

T_1	
	T_2

schedule 2 → first T_2 is executed, then T_1

T_2	
T_1	

schedule 3

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

→ first part of T_1 is executed then part of T_2 is executed then again T_1, then T_2 and so on...

- In every case, ACID property should be satisfied.

Transaction T

read write

=> T_1

read (A)
A = A - 500;
write (A)
read (B)
B = B + 500;
write (B)

T_2:

read (A)
temp = A * 0.5
~~write (A)~~
A = A - temp;
write (A)
read (B)
B = B + temp
write (B)

* A = 1000
B = 1000

SCHEDULE 1 : T_1 : A = 1000
A = 1000 - 500 = 500
B = 1000
B = 1500

~~SCHEDULE 2~~ T_2 : A = 500
temp = 25
A = 500 - 25 = 475
B = 1500
B = 1500 + 25 = 1525

SCHEDULE 2 T_2 : A = 1000
~~A = 500~~ temp = 1000 * 0.5 = 50
A = 950
B = 1000
B = 1000 + 50
B = 1050

T_1 : A = 950

A = A - 500 = 950 - 500 = 450

B = 1050

B = 1050 + 500 = 1500

Schedule 3 : T_1 : A = 450

A = A - 500

A = -50

T_2 : A = -50

temp = A * 0.5 = -0.25

A = A - temp

* Each schedule gives diff. results (inconsistent)

* Schedule 4

read A	read (A)
A = A - 500	temp = A * 0.5
	A = A - temp
	write (A)
	read (B)
write (A)	B = B + temp
write (B)	write (B)

SERIALIZABILITY

(i) Conflict serializability

Any schedule which consists 2 transactions T_i & T_j will be conflict serializable if:

(i) I_i = read (Q) .. I_j = read (Q)
T_i T_j

(ii) I_i = read (Q) I_j = write (Q) } Conflicting

(iii) I_i = write (Q) I_j = read (Q) }

(iv) I_i = write (Q) I_j = write (Q)

* Any schedule S that can be transferred into another schedule S' by swapping of instructions, then S & S' are known as CONFLICT EQUIVALENT.

* Schedule is conflict serializable if it is equivalent to serial schedule.

Serial schedule

read(A)	
:	
read(B)	
	read(A)
	:
	read(B)

(ii). View serializability

A schedule S is view serializable, w.r.t. to S' if :

(i). For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then T_i must in schedule S', also read the initial value of Q.

(ii). For each data item Q, if transaction T_i executes read(Q) in S, and that value was produced by T_j, then T_i must in schedule S' also reads Q that was produced by transaction T_j.

(iii). For each data item Q, the transaction that performs write(Q) operation in S must perform final write(Q) operation in schedule S'.

* Useful during recovery.

=> Recoverable schedules

→ RECOVERABILITY — disk failure, logical failure, system failure

(i) Recoverable schedules : A recoverable schedule is one where for each pair of transactions (T_i, T_j) s.t. T_j reads data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j

(ii) Cascadeless schedules

A cascadeless schedule is one where for each pair of transaction (T_i, T_j) s.t. T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

→ RECOVERY

* Log Based Recovery

1. Deferred database modification
2. Immediate database modification

T_0	T_1
read(A)	read(C)
A = A-100	C=C-500
write (A)	write (C)
read(B)	
B = B+100	
Write B	

Q) Log file: (A = 1000, B = 1000, C = 1000)

<T0, start>
<T0, A, 900>
<T0, B, 1100>
<T0, commit>

<T1, start>
<T1, C, 500>
<T1, commit>

Date

- ~~Deferred~~ Deferred

log
<T0 start>
<T0, A, 900>
<T0, B, 1100>
<T0, commit>

Database
A = 900
B = 1100

We modify the database after commit operation in Deferred Database modification.

<T1, start>
<T1, C, 500>
<T1, commit>

C = 500

- If T1 has started but not committed, ~~then~~ at ~~after~~ system failure, then it will ~~redo T1~~. undo T1

- If failure occurs at <T0, B, 1100>, then it will restart the entire transaction.

=> Immediate

log
<T0, start>
<T0, A, 900>
<T0, B, 1100>
<T0, commit>
<T1, start>
<T1, C, 500>
<T1, commit>

Database
A = 900
B = 1100
C = 500

database is modified as soon as value changes.

Date / /

checkpoints : to check if transaction has committed successfully.

before commit

usually after modification of database.

log

<T0, start>

<T0, A, 900>

~~<T0, B, 111~~ <T0, checkpoint>

<T0, B, 1100>

<T0, commit>

Concurrency Control

→ Protocols
  ↳ system generated

① lock based protocols

lock → shared lock [A transaction can only read, but not write ~~vice versa~~]
  ↳ exclusive lock [any transaction can do both read & write operations]

compatibility matrix

	S	X
S	true	false
X	false	false

compatibility is true only when both locks are shared.

* T1 :
lock - X(B)
read(B)
B = B+100
write(B)
unlock(B)
lock - X(A)
read(A)
A = A+100
write(A)
unlock(A)

T2 :
lock - S(A)
read(A)
unlock(A)
lock S(B)
read(B)
unlock(B)
display(A+B)

* Concurrency - control Manager
stores: information about locking & unlocking

T_1	T_2	Concurrent_mgr
lock-X(B)		
read(B)		grant X(B, T_1)
⋮		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock S(B)	
		grant-S(B, T_2)
	read(B)	
	⋮	
	display(A+B)	
		~~lock-X(A, T_1)~~
lock X(A)		
		grant X(A, T_1)
read(A)		
⋮		
unlock(A)		

* granting of locks should not lead to deadlock, starvation.

Waiting by one transaction for lock to be released by another transaction

Locking protocol

If there are set of transactions $T_0, T_1 \ldots T_n$ in schedule S, any transaction T_i precedes T_j in schedule S written as:

$T_i \rightarrow T_j$

if there is a data item Q s.t. T_i has held lock mode A on Q and T_j has held lock mode B on Q, then compatibility of A, B [comp (A, B)] = false.

If $T_i \rightarrow T_j$, then that precedence implies that any equivalent serial schedule T_i must appear before T_j.

A locking protocol ensures conflict serializability if & only if for all legal schedules, the association arrow ($\rightarrow$) relation is acyclic.

↓

STARVATION

To avoid starvation, the locks can be granted as follows:

(i) When any transaction T_i requests a lock on data item Q in a mode M, then the lock will be granted provided that:

(a) there is no other transaction holding a lock on Q in a mode that conflicts with M.

DOMS | Page No.
Date / /

- there is no other transaction that is waiting for a lock on B and that made its request before T_i.

* TWO-PHASE LOCKING PROTOCOL

2 phases :→ Growing Phase
└→ & Shrinking Phase

In growing phase a transaction may obtain a lock, but cannot release a lock.

In shrinking phase a transaction may release a lock, but cannot obtain a lock.

- We need to ensure conflict serializability

(i) Strict 2 phase Locking protocol.

(ii) Rigorous 2 phase Locking protocol.

* GRAPH BASED PROTOCOL

↓

A tree will be generated for locks assigned to a transaction.

(2) TIME-STAMP BASED PROTOCOL

↓

assigning a time span when transaction starts, reads, write, ends.

* System lock
* Logical Counter Lock.

* W-timestamp (B).
  R-timestamp (B)
  TS(Ti)

Conditions for protocol design:

(A) read (B) by Ti

(i) TS(Ti) < W-timestamp (B) → read will be rejected: Ti reads the value of B that is already overwritten & Ti will be rolled back.

(ii) TS(Ti) ≥ W-timestamp (B) → read opn. will be performed & R-TS = max [TS(Ti), W-TS]

(B) Ti issues write (B)

(i) TS(Ti) < R-Timestamp (B) → write opn. will be rejected & Ti will be rolled back.

(ii) TS(Ti) < W-timestamp (B) → write opn. will be rejected & Ti will be rolled back.

(iii) Otherwise

↓

write opn. will be executed & W-TS = TS(Ti).

* Obsolete write

↳ Thoma's write Rule to avoid obsolete write.

Same as above conditions. Only (B) (ii) write will be performed.

(3) Validation Based protocol

3 phases:

(i) Read phase → perform read opn.
(ii) Validation phase → validate what is being done
(iii) Write phase → perform write opn.