Date

Page

DBMS

Data :- Collection of Raw bytes.
- → Quantitative → numerical form
- → Qualitative → Descriptive

Information : processed, organised & structured data.
→ It provides context of data & enables decision making.

→ RDBMS (Database management System)
Relational
↳ A electronic place where data is stored
* Creating, updating, deleting & inserting

DBMS : collection of interrelated data & a set of program to acces those data.
* Interface to perform data operation.

Need of DBMS :

optimisation : By minimizing storage & time of data retrieval.

Advantages :
- → Controls redundancy
- → Data sharing
- → Backup & recovery
- → Multiple user interface.

Disadvantage : → Size → Complexity → Cost

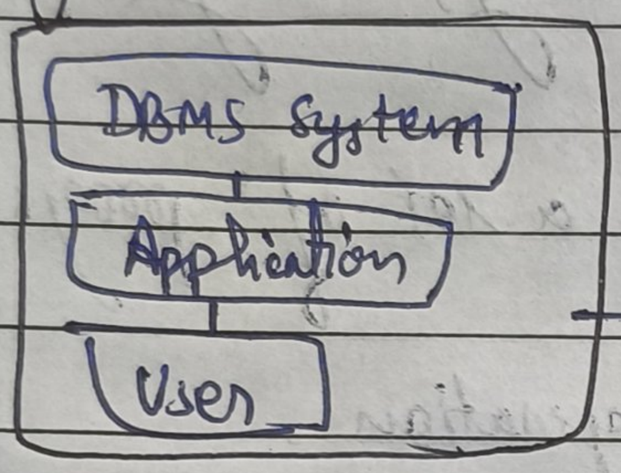
DBMS V/s File System

DBMS	FS
→ No need to write procedures	→ Need to write procedures
→ Gives abstract view, hides the details	→ Always provides detailed view & storage representation.

→ Provides crash recovery mechanism. Protects users from system failure	→ Doesn't provide crash mechanism, system crash while entering data, data will be lost.
→ Contains wide variety of techniques to store & retrieve data.	→ FS can't effectively store data.

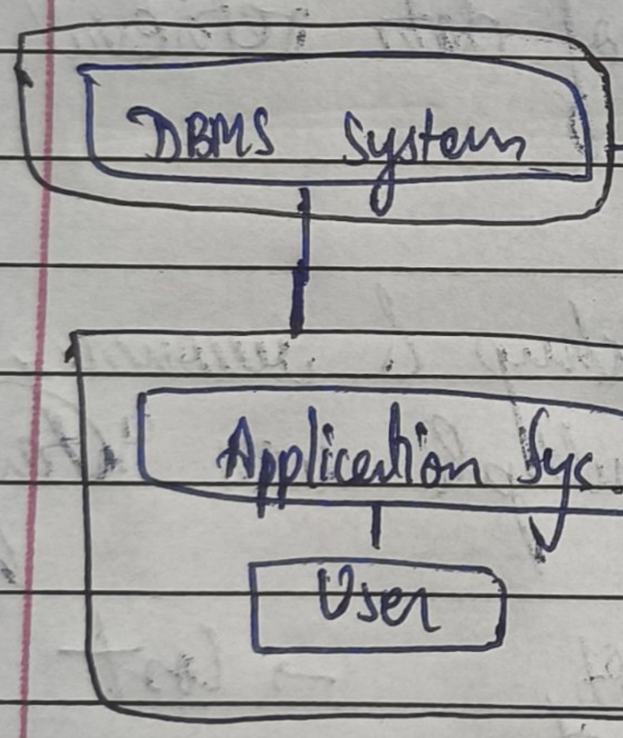
DBMS Application Architecture

1. Single tier :



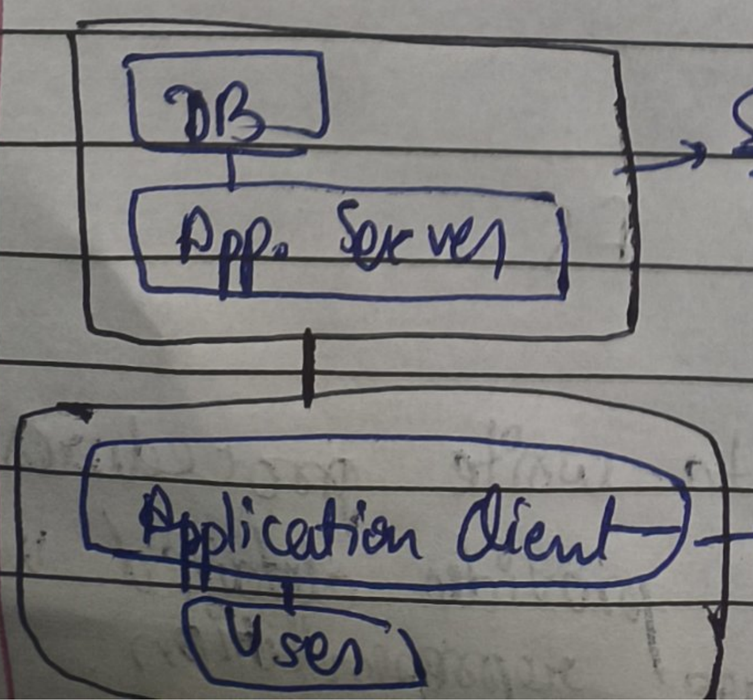
* Everything is available at client m/c.
* Used for local development

2. Second tier :



* DB at Server m/c.
* Connected via reliable n/w
* for API integration like ODBC, JDBC.

3. Third tier :



* Most popular & reliable
* DB & application servers can or can not be together.
* everything is integrated using API.

Date
Page

Data abstraction :

→ main purpose is to isolate data independence

Hiding irrelevant details from users & providing abstract view of data to users helping easy, interactive & efficient user database interaction.

Three level of abstraction

① Physical level / Internal level.

↳ Tells how it is stored?
Access = Random or Sequential?
DB type used?

② logical level / Conceptual level

↳ Stores relationships of tables.

③ View level / External level

↳
→ Highest level of abstraction
→ Only part of DB viewed by user.

Data Independence :

ability to modify schema at one level of DB, without affecting higher & next level of schema.

i.e : Changing HDD to SDD at physical level should not affect conceptual or external levels.

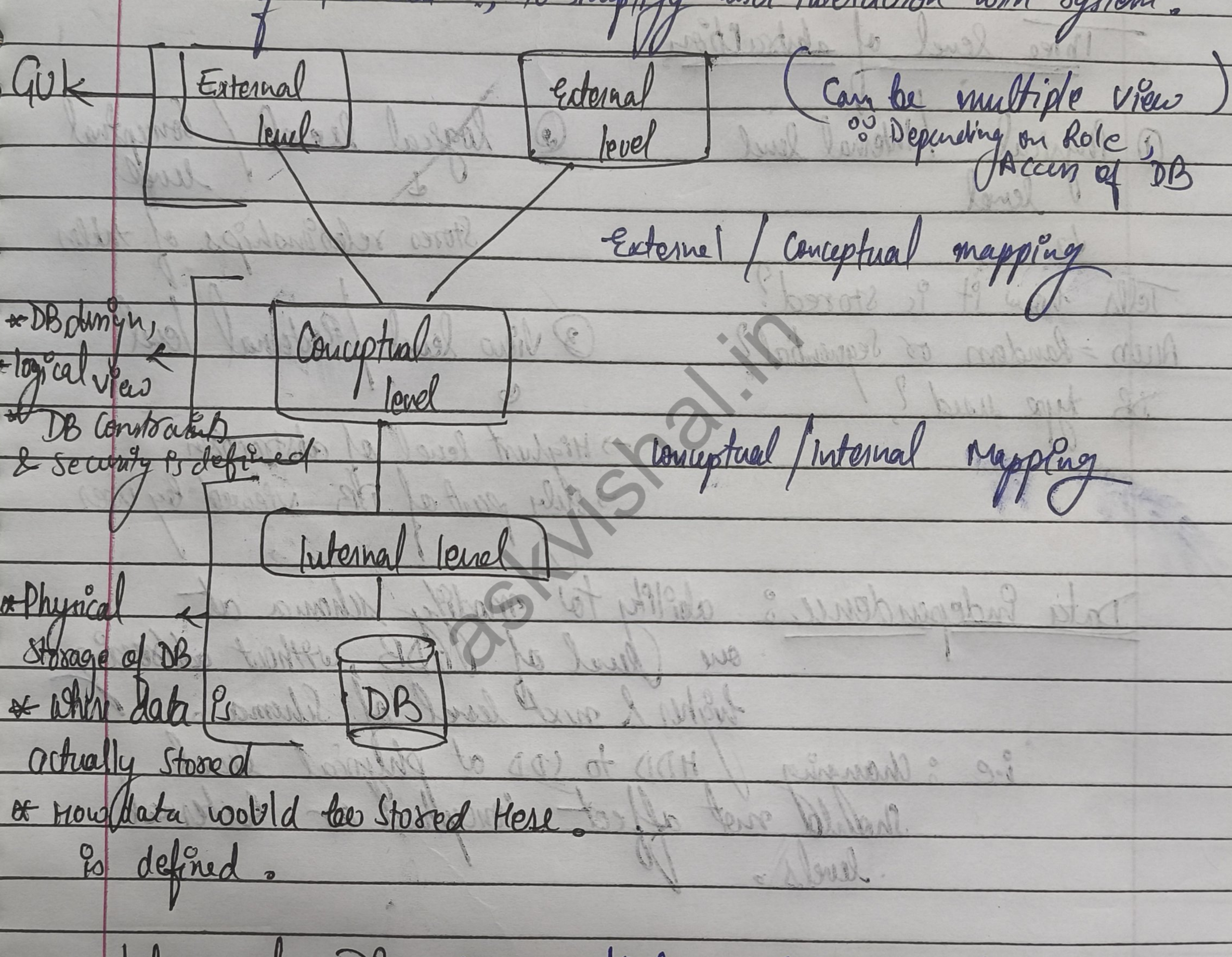
Date

Page

Schema: Design of the database (layout)

View of Data (Three Schema Architecture)

→ The major purpose of DBMS is to provide user with an abstract view of the data., to simplify user interaction with System.



Instance of DB At a particular point in time how many data points are in storage.

	Name	Ph. no.	addres
①	___	___	___
②	___	___	
③	___	___	___

12:00 AM : 2 instance
12:15 AM : 3 instance

Database languages

A dbms has appropriate languages and interface to express data queries & database updates.

Types of database language

① Data Definition language : specify the database schema.
* create, alter, drop, truncate & rename.

② Data manupulation language : expose database queries & updates.
* Select, Insert, Update, delete, merge

diff. b/w delete, drop & truncate.

feat.	delete	drop	truncate
action	removes specific rows	removes all rows	removes entire table
Type	DML	DDL	DDL
where clause	Supported	Not Supported	Not Supported
Rollback	Possible	Database dependent	Harder to undo
Speed	Slow	fast	fast
Space	Keeps empty space	Resets storage	Reclaims all storage

③ Data Control language : we specify consistency constraints.
→ which must be checked every time DB is updated.
* Grant & Revoke

④ Transaction control language :
→ when transactions are successfully committed only then changes in the database is committed otherwise Rollback. to Save at point : savepoint.

Data Models

→ provides a way to describe the design of a DB, at logical level.

→ underlying structure of DB is the Data Model; a collection of conceptual tools for describing data, data relationships, data semantics & consistency constraint

Eg: ① ER model
② Object Oriented Model
③ Relational Model
④ Hierarchical Model
⑤ Network Model.

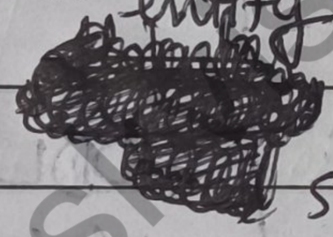
ER Model { Entity. Relationship Model }

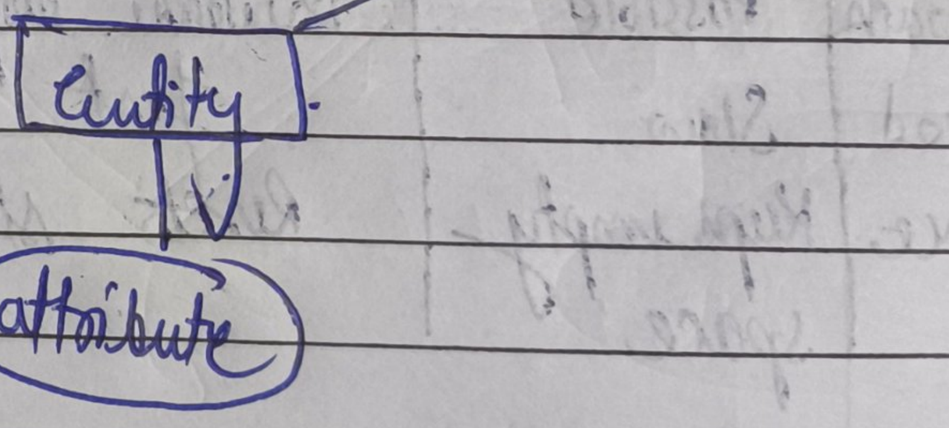
{ * Refer codehelp dbms vale for detailed explanation }

real world objects → entity, properties used to describe entity ⇒ attribute

relationship b/w entity → relation

should be precised & limited.



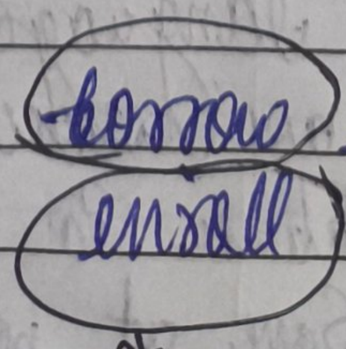


→ entity can be uniquely identified by a primary attribute (aka primary key)

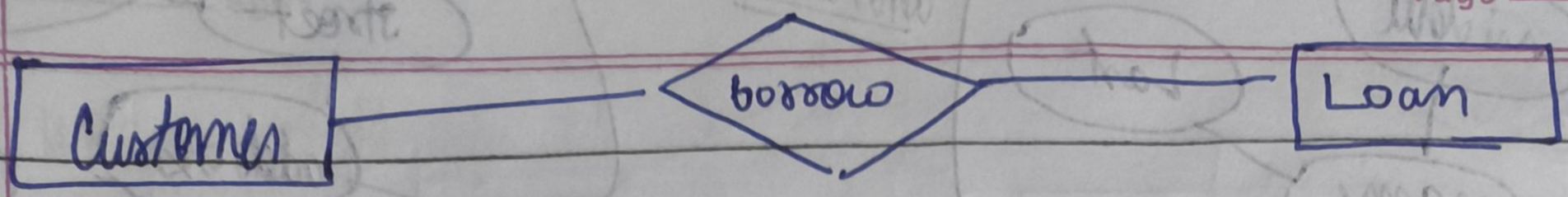
Entity set: a set of entities of the same type that share same properties or attributes

Eg: Student, Customer.

Relations:



Date
Page



ER Model : high level data model based on perception of real world that consist of basic objects, called entities. and of relationship among these objects.
→ acts as a blue print.

* Attribute
→ domain, values
→ Permitted / Non permitted actions
→ consistency constraints.

types of attribute

① Simple → which can not be further divided. Eg. aadhar acc. no.

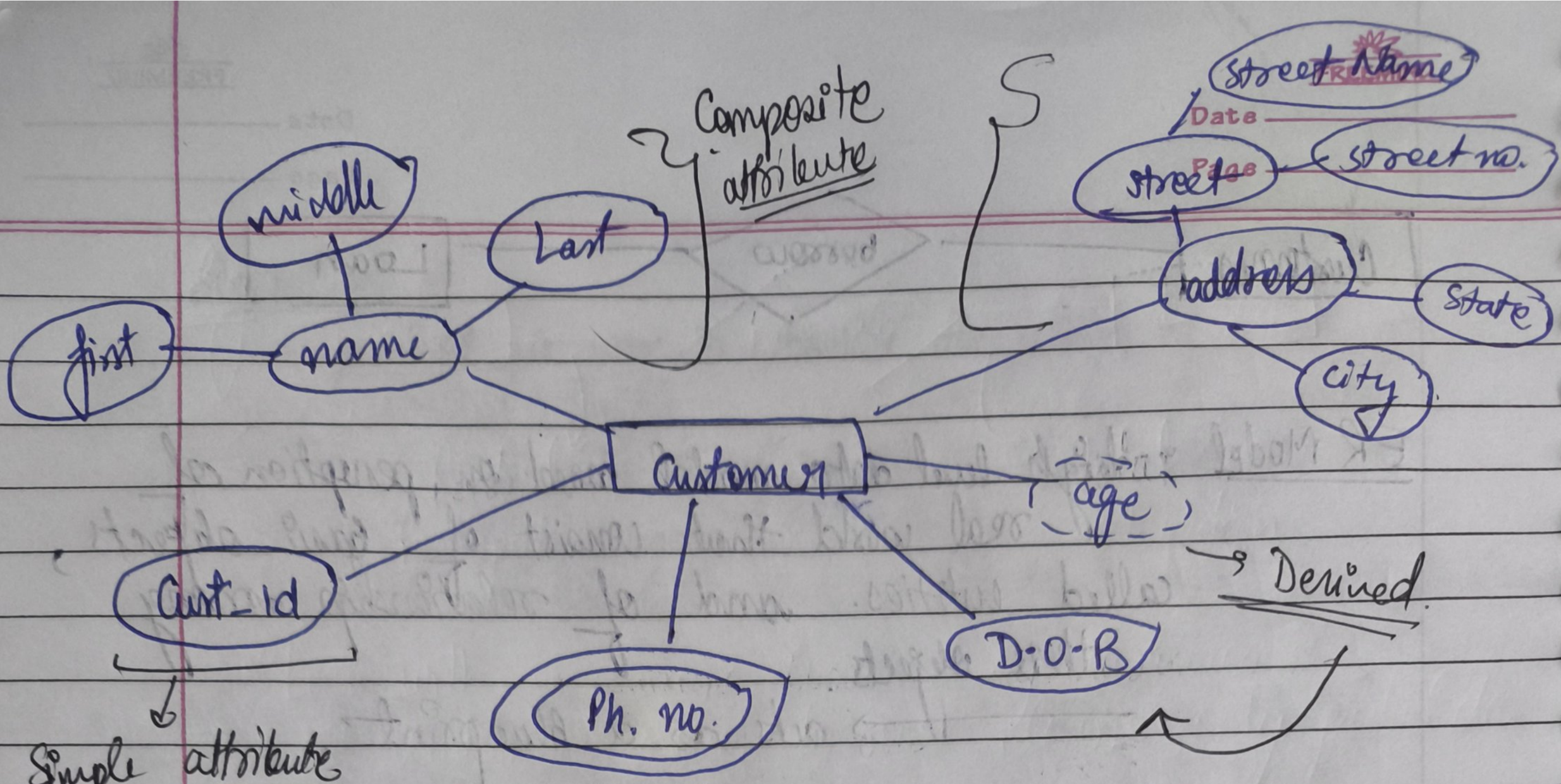
② Composite → can be divided into subparts. Eg. name of person.

③ Single Valued → Only one value attribute : Eg : Student ID

④ Multi-valued → Attribute with multiple values : Eg. Ph. no.
(Limit constraints may be applied)

⑤ Derived → Value of this type of attribute can be derived from the other related attribute.
Eg: Age from D.O.B

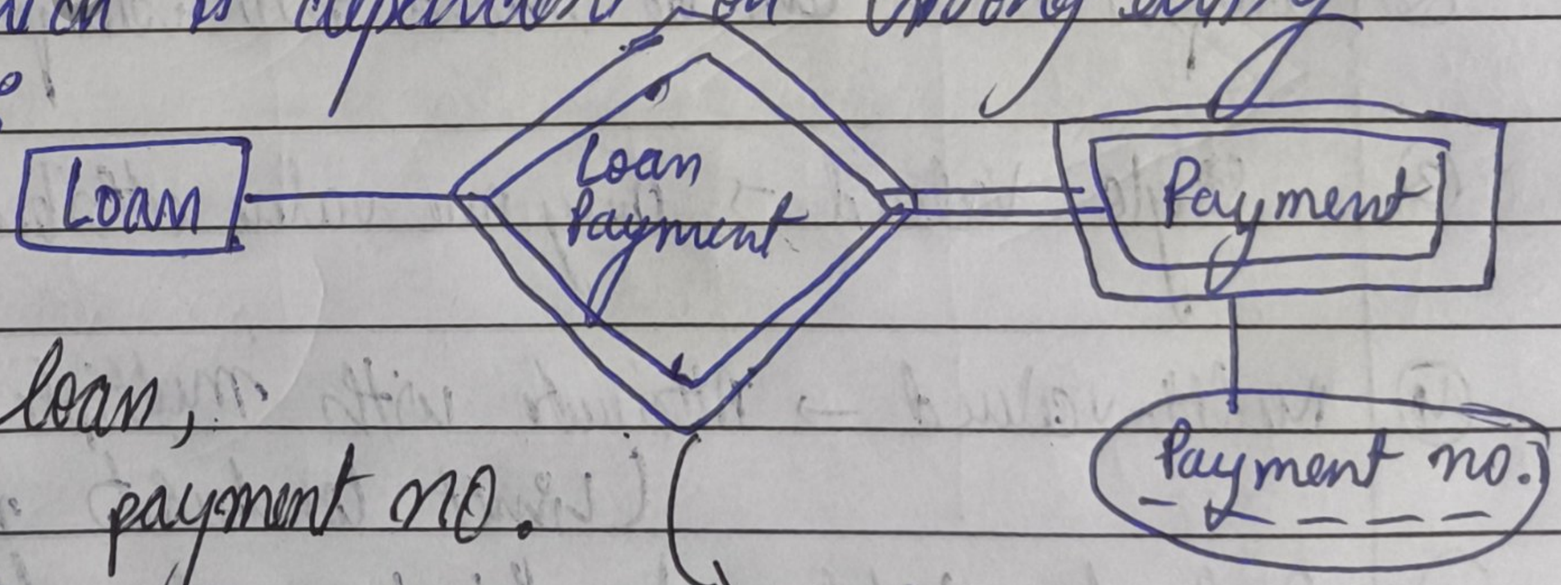
⑥ Null → an attribute with no value.
means : 'not applicable', 'value does not exist'.
ie → person with no middle name.
Eg : P.T.O
↳ unknown
↳ Missing



Relationships

a) Strong entity : which itself could be uniquely identified.
eg: Student with primary key as roll no.

b) weak entity : which is dependent on strong entity
eg:



→ If no loan,
No payment no.

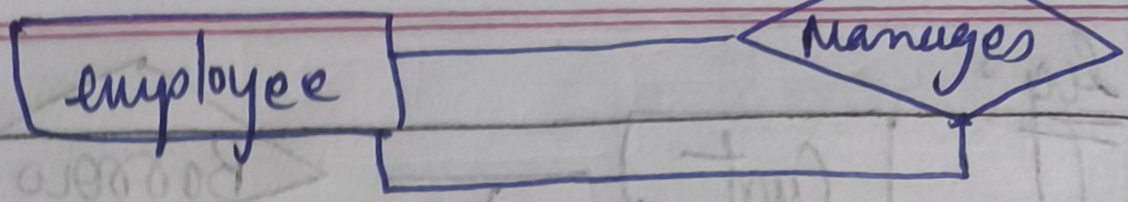
weak relation

Strong relations

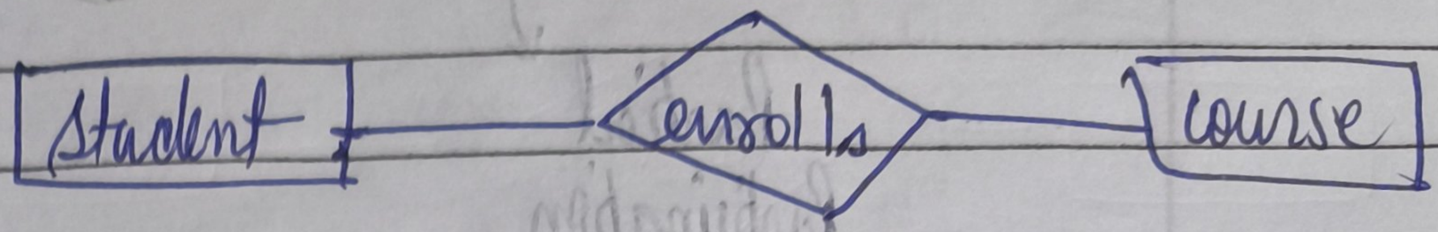
unique_id ← customer places Order → unique_id

Degree of Relation : total no. of entities participating in a relationship.

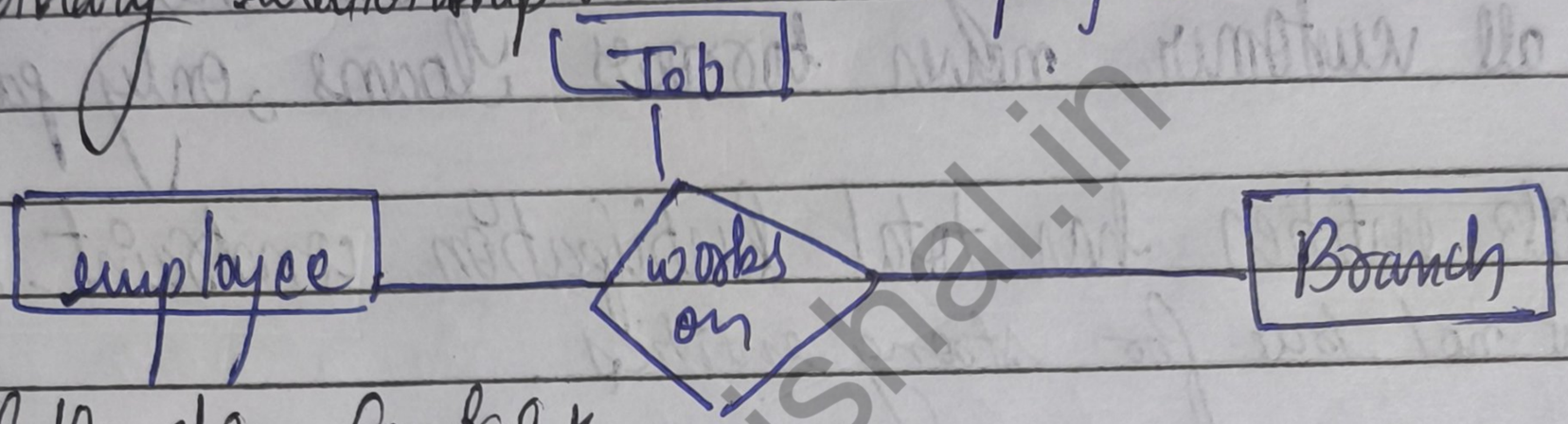
a) unary relationship :



b) Binary relationship :



c) Ternary relationship :



Relationship Constraints

1) Mapping cardinality :

citizen has Pan-card → one to one , → one to many

→ Many to one , → many to many

citizen has vehicle

Course taken by Student

Customer buys Product

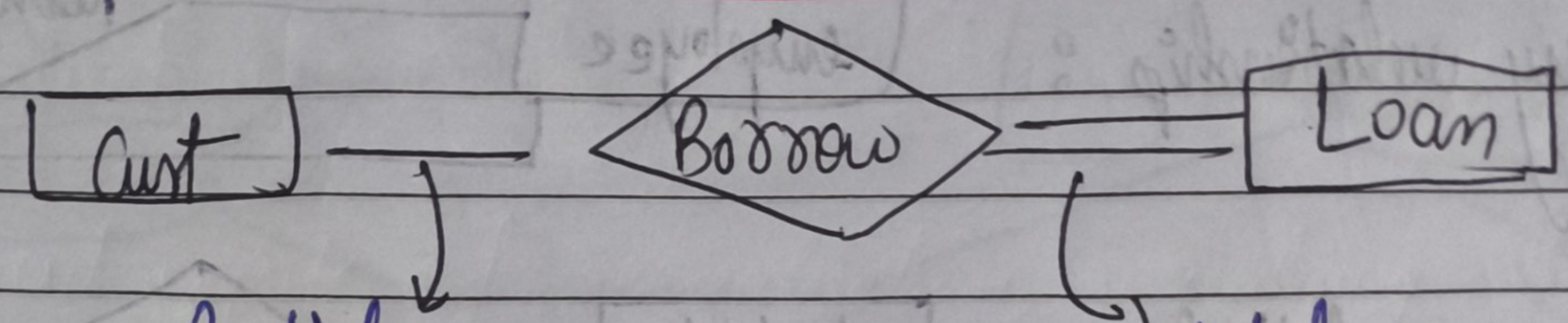
2) Participation Constrain :

aka minimum cardinality constraint.

Types : a) Partial → not all entities are involved

b) total → all entities are involved.

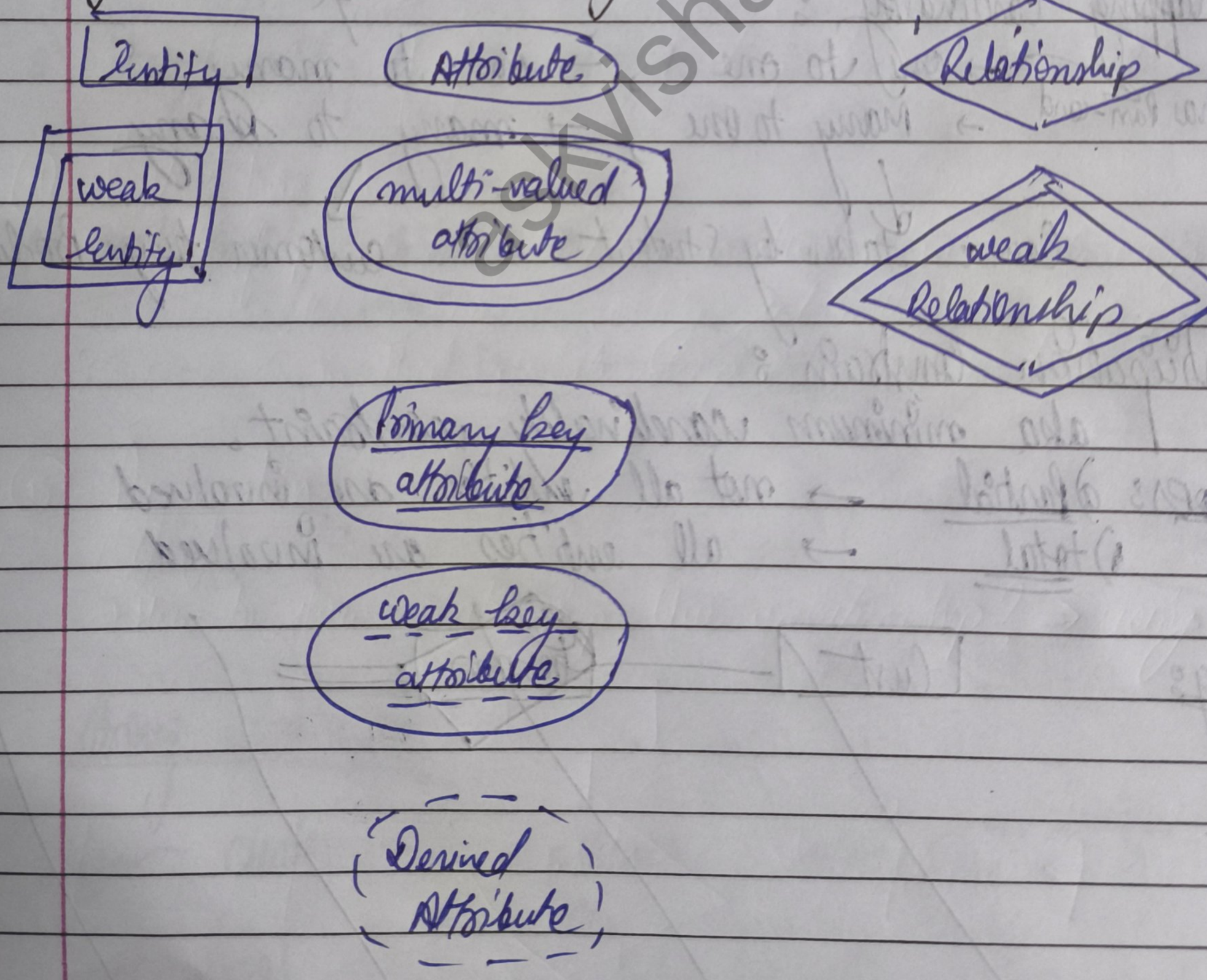
Eg:



- Loan will be always borrowed by customer.
- all customer never borrows loans, only partial do.

* weak entities has total participation constraint always, but not true for strong entities.

Symbols used in ER Diagram



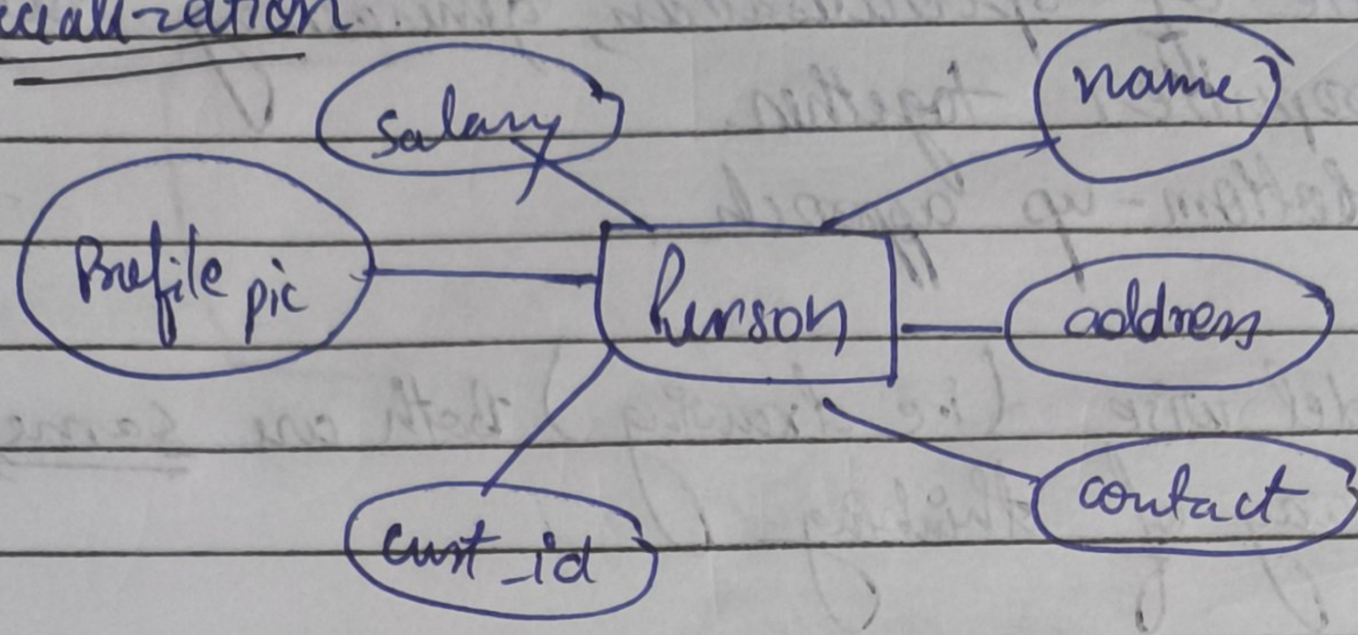
FREEMIND

Date

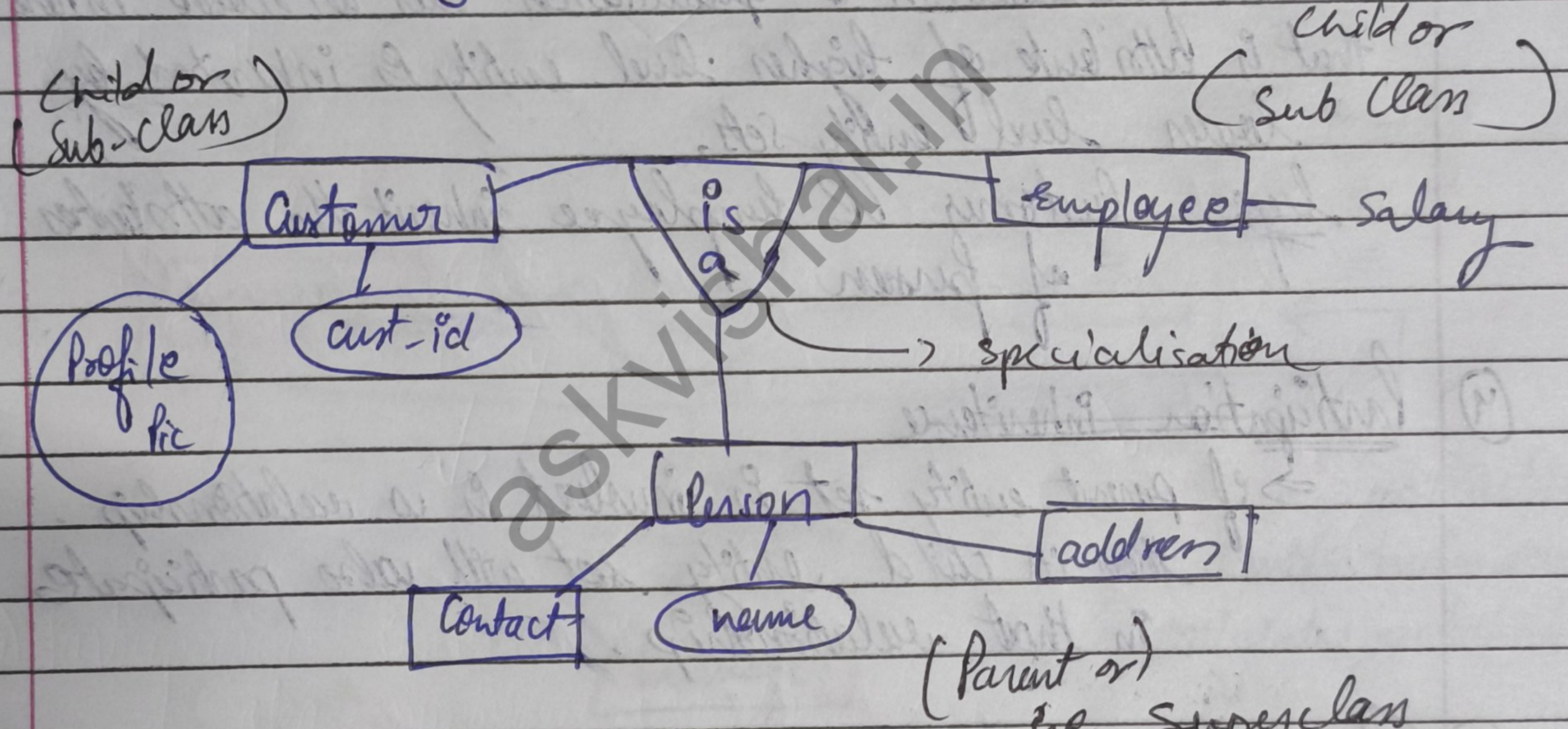
Page

Extended ER features

① Specialization



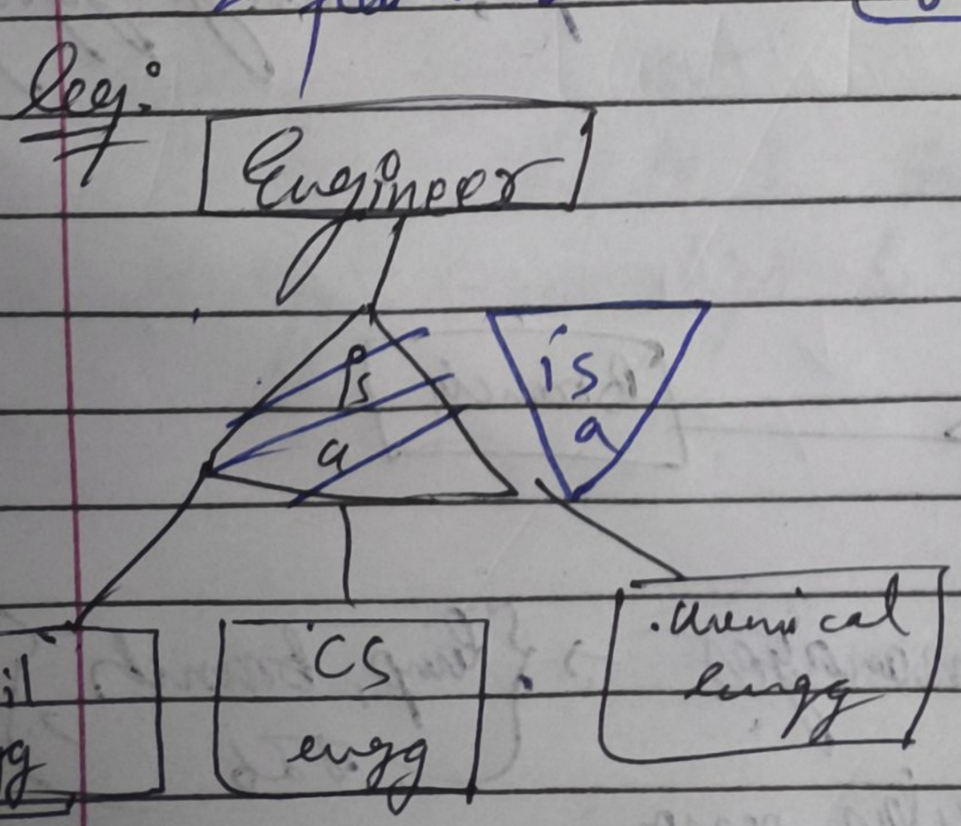
→ to many details which are merged together.

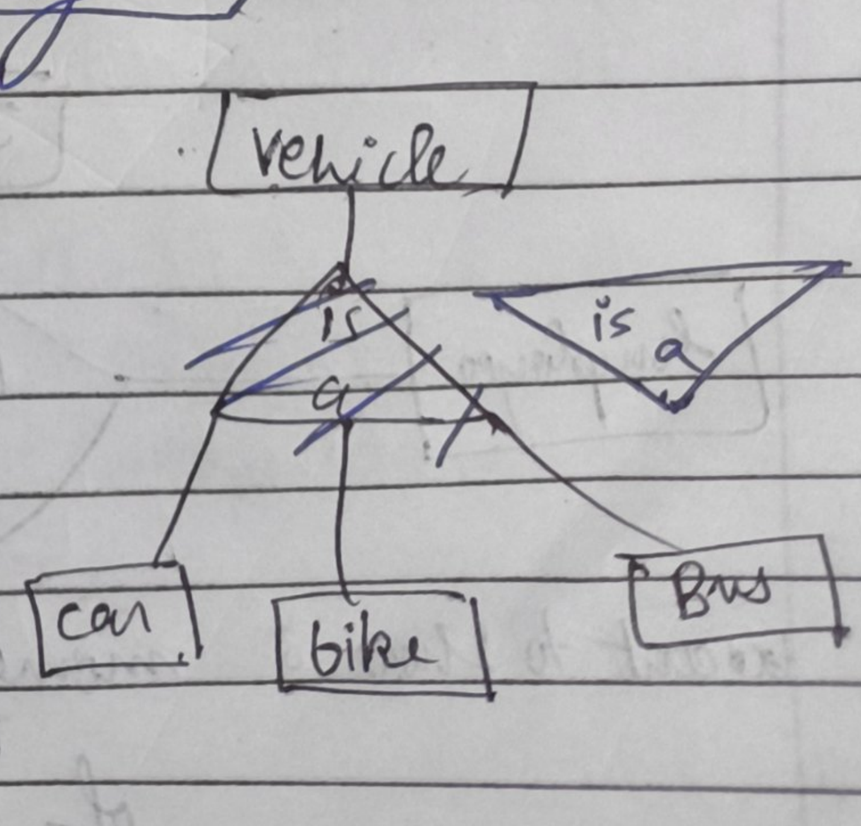


Specialisation is a top-down approch where entities are splitted, based on functionalities, specialities & features.

onto sub-entity set

Eg:





Date

Page

② Generalisation :

⇒ exact oppsite of Specialisation, generalising the similar properties together.

⇒ Its a bottom-up approch.

* ER term model wise (i.e drawing) Both are same
  its just a way of thinking.

③ Attribute Inheritance :

⇒ Both generalisation & Specialisation has attribute Inheritence. that is attribute of higher level entity is inherited by lower level entity sets.

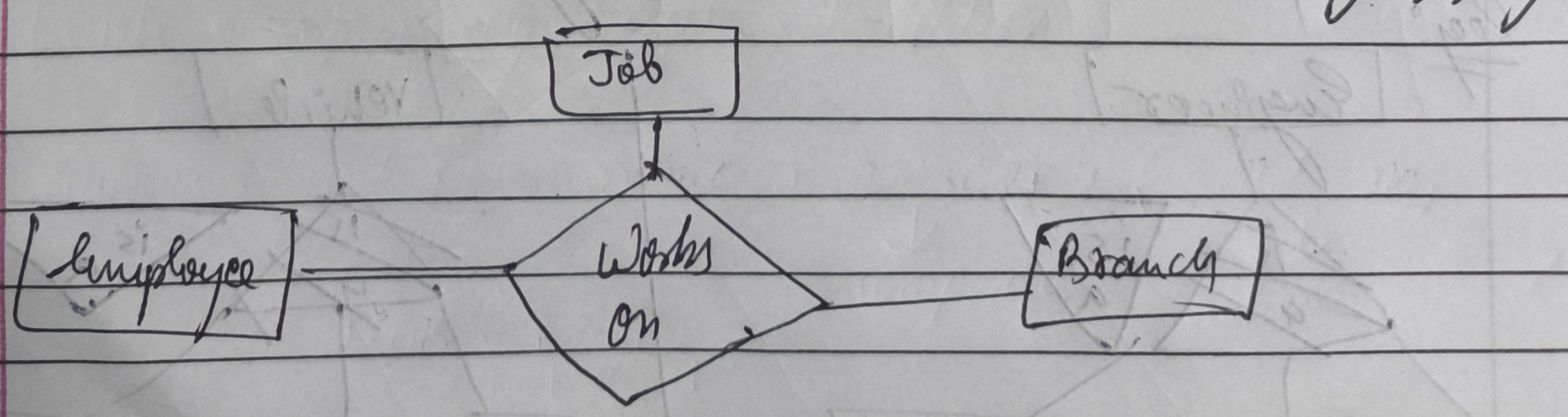
Eg: Customer & Employee inherit the attributes of person

④ Participation Inheritance

⇒ If parent entity set is involved in a relationship. then its child entity set will also participate in that relationship.

⑤ Aggregation

⇒ How to show relationship b/w relationship ? ⇒ using aggregate.



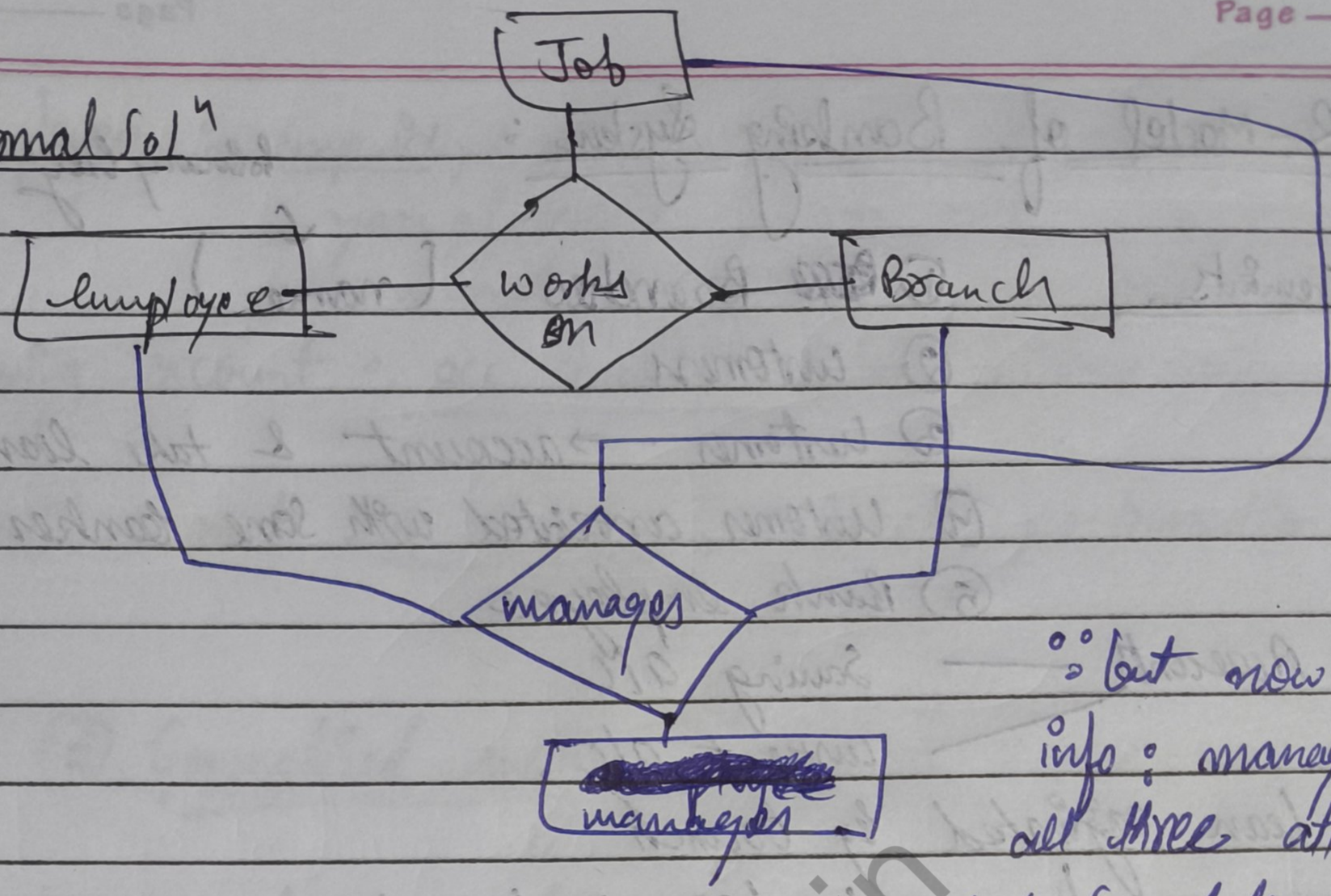
want to show : manages → manages → {Emp, Branch, Job}

of a particular person.

Date

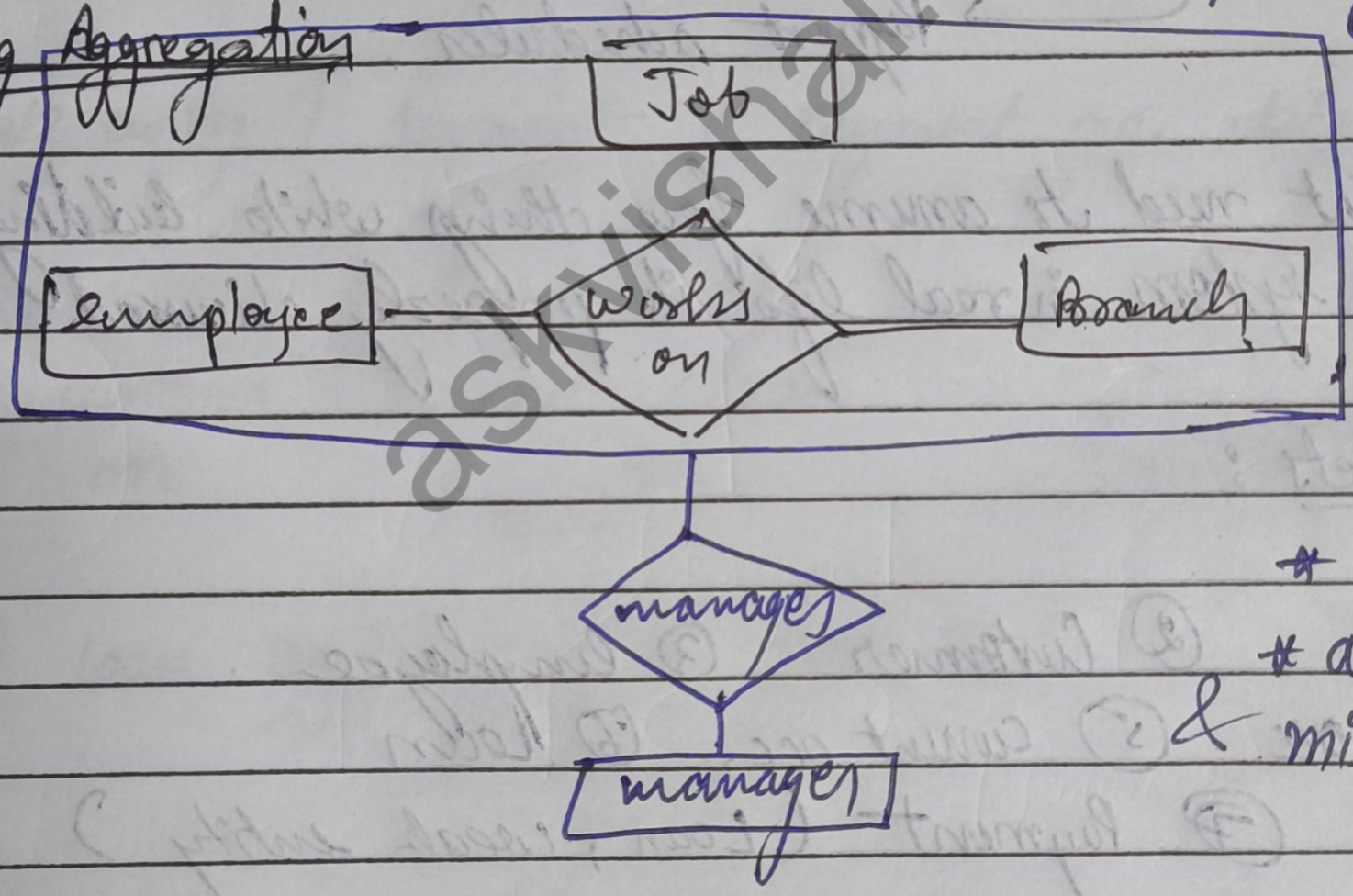
Page

Normal Sol^n



∴ but now redundant info: manager manages all three attribute but separately.

Using ~~Aggregation~~



* adds abstraction.
* avoid redundancy & misinterpretation.

Steps to make ER diagram:

1. Identify entity sets
2. Identify attribute & their types.
3. Identify relationships & constraints.
   - → Mapping
   - → Participation.

ER-Model of Banking System :

Binary key

Think of Requirements

① Branches (name)
② customers
③ Customer → account & take loan
④ Customer associated with some banker.
⑤ Bank employee
⑥ Accounts — Saving a/c
current a/c.
⑦ Loan originated by branch
↳ Loan ≥ 1 customer
→ Payment schedules.

+ We might need to assume few things while building banking system. in real life it properly discussed.

① Entity Sets :

① Branch ② Customer ③ Employee.
④ Saving acc. ⑤ current acc. ⑥ Loan
⑦ Payment (Loan, weak entity)

② Attributes

① branch : name, city, assets, liabilities

② customer : cust-id, name, address, contact no.
address → composite
contact no → multi-valued
DOB, age
age → derived

FREEMIND

Date

Page

③ Employee : emp_id ; name , contact no , dependent name ,
multi-valued
year of service , start_date.
derived attribute & Single value

④ Saving account : acc_number , balance , interest_rate , daily withdraw limit

⑤ Current account : acc_number , balance , per-transaction charges ,
overdraft_amount

⑥ Generalized entity "Account" → acc_no, balance

⑦ Loan : loan_number , amount

⑧ (weak entity) Payment : Payment no. , date , amount

③ Relationship & constraints.

① customer borrow loan
M : N (mapping cardinality)

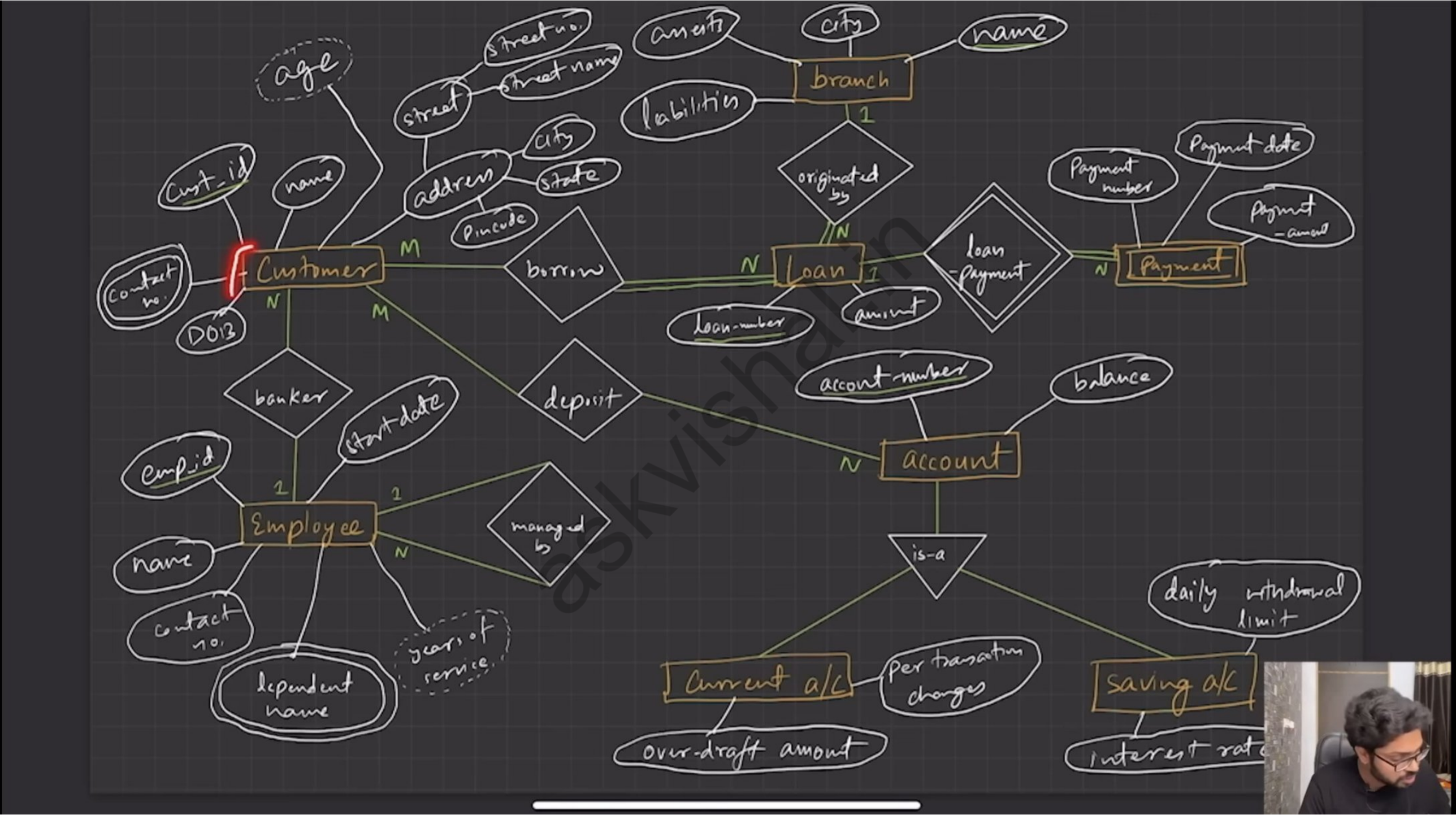
② loan originated by branch
N : 1

③ loan loan_payment Payment
1 : N

④ customer deposit account
M : N

⑤ customer Banker employee
N : 1

⑥ employee managed by employee
N : 1 (unary relationship)


age
street no.
street name
street
assets
city
name
branch
liabilities
1
Cust_id
name
address
city
state
Pincode
originated by
N
Payment number
Payment date
Payment amount
Customer
M
borrow
N
Loan
1
loan-payment
N
Payment
Contact no.
DOB
N
M
Loan-number
amount
banker
deposit
account-number
balance
start-date
emp_id
1
1
account
N
Employee
managed by
is-a
name
N
contact no.
dependent name
years of service
daily withdrawal limit
Current a/c
per transaction charges
Saving a/c
over-draft amount
interest rat

Facebook — DB formulate using ER model

* ER diagram

① Features & use case.

① profile → user_profiles → friends.

② user can post

③ Post → contains → text content, images, videos.

④ Post → like, comment.

① identify Entity sets.

① user_profile ② user_post ③ post_comment

④ post_like

② Attributes + types.

① user_profile → Name, username, email, password, Contact no.

Name → composite. DOB, age.

age → derived.

email → multivalued.

Contact no. → multivalued.

② user_post → post_id, text content, image, video,

image → multivalued

video → multi

created_timestamp, modified_timestamp,

③ post_comment → post_comment_id, text content, timestamp

④ post_like → postlike_id, timestamp

③ Rel^n & Constants

① user_profile — friendship — user_profile.
M : N

② user_profile — posts — user_post.
1 : N

③ user_profile can post_likes

1 : N

④ user_profile comments post_comments

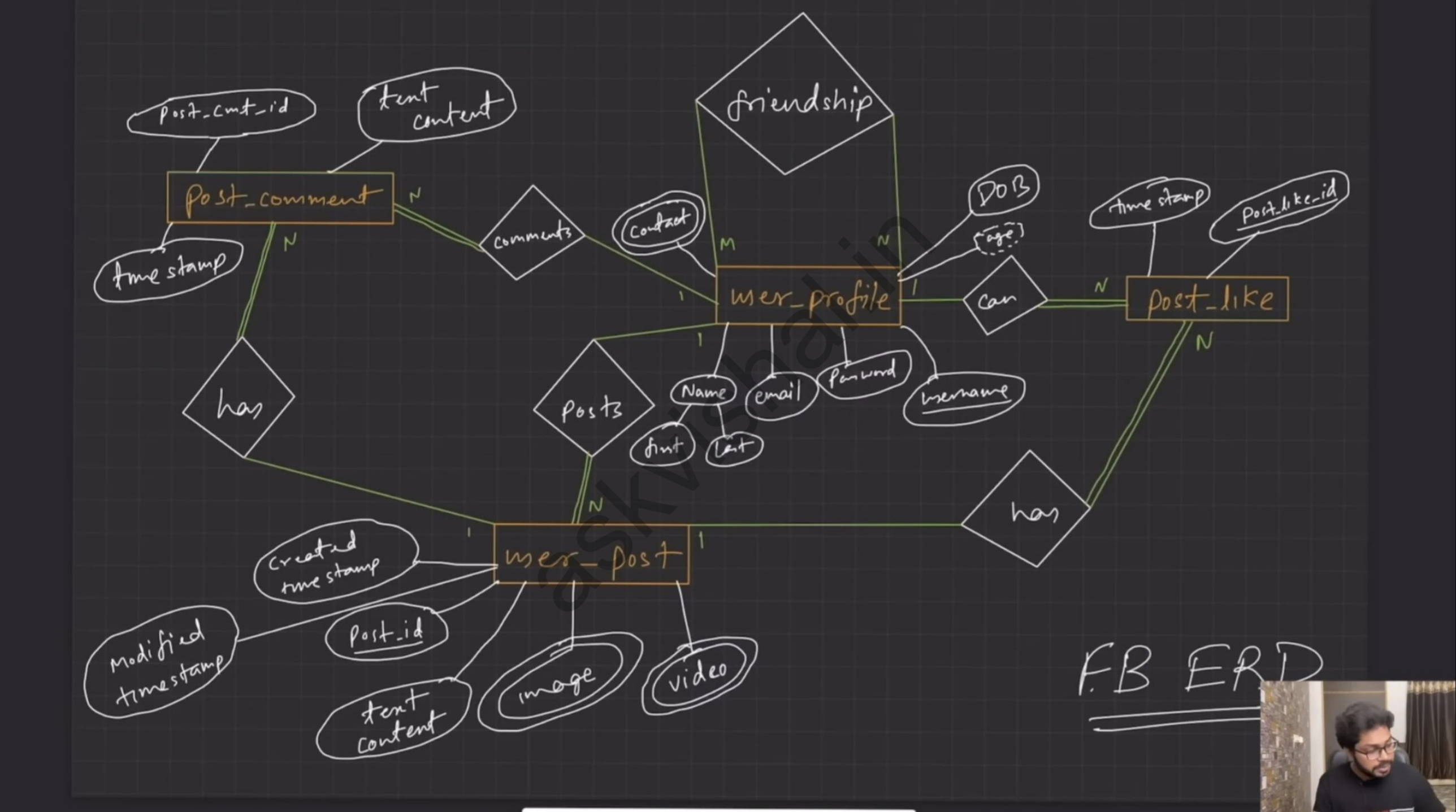
1 : N

⑤ user_post has post_comment

1 : N

⑥ user_post has postlike

1 : N


Post_cmt_id
text content
Post_comment
N
N
time stamp
comments
friendship
Context
M
N
DOB
age
time stamp
Post_like_id
user_profile
1
1
can
N
post_like
N
has
Posts
Name
email
Password
username
first
Last
N
has
1
1
Created time stamp
user_post
Modified timestamp
Post_id
text content
image
video
FB ERD

FREEMIND

Date

Page

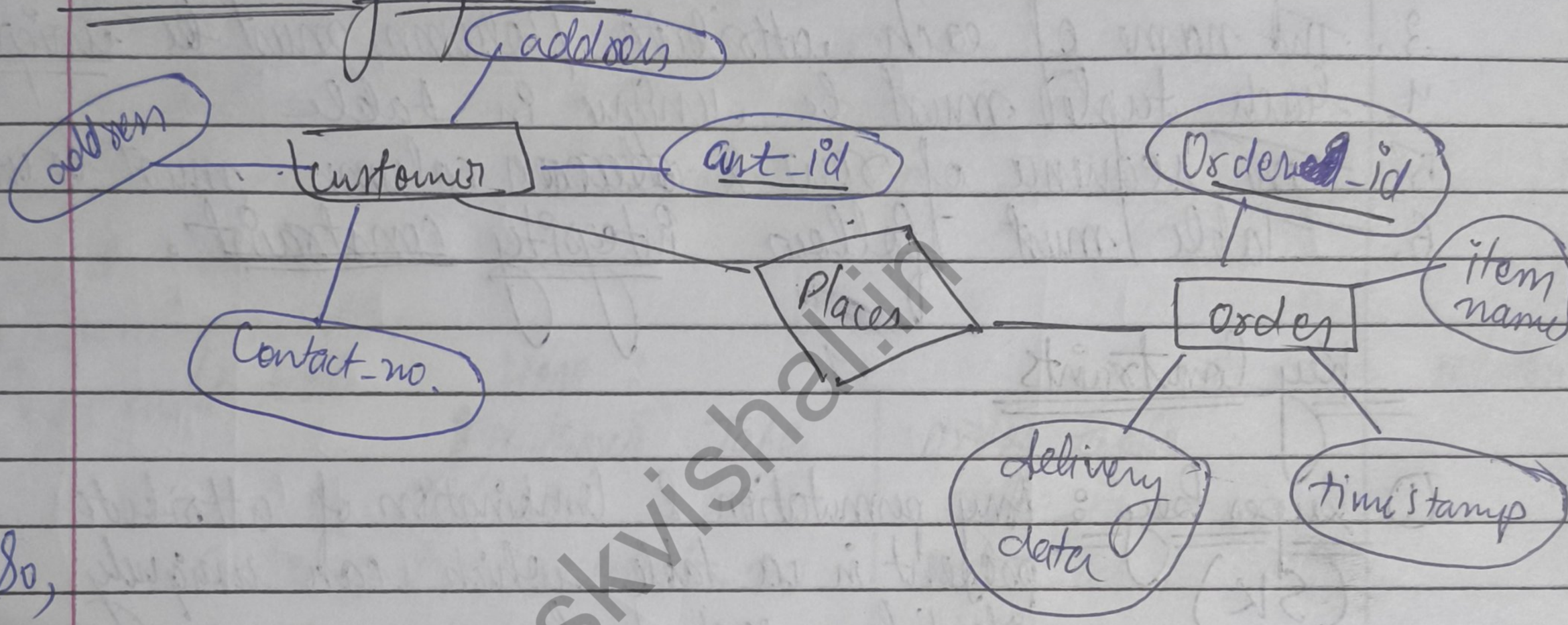
Relational Model

→ The relation is represented in table form where:

from E-R Model : Relation ⟶ Defined by table
to Relational Model Entity ⟶ table name
Attribute ⟶ table ~~cols~~ rows / tuple

Eg:

Online delivery System



So,

① Customer (cust_id, name, address, contact no.)
② Order (order_id, timestamp, delivery_data, item_name)

key attributes

cust_id	name	address	contact no.

→ Tuple or row

Degree : No. of attributes
Cardinality : No. of tuples.

DB design has two steps:

① E-R Model ⟶ ② Relational Model.

FREEMIND

Date

Page

* Software Implementation of relational model is RDBMS.
  eg: mySQL, MS Access, Oracle etc.

Important Property of relational table :

1. Name of relation is distinct among all relation.
2. the value must be atomic, can't be broken down further.
3. The name of each attribute / column must be unique.
4. Each tuple must be unique in table
5. In sequence of row & column must be unique
6. Table must follow Integrity constraint.

Key Constraints

① Super key (SK) : Any permutation & combination of attributes present in a table which can uniquely identify each tuple

② Candidate key (CK) : number of minimum subset of superkey, which can uniquely identify each tuple.
   → It contains no redundant values.
   → CK value should not be Null.

③ Primary key (PK) : Selected out of CK set, has the least numbers of attributes.

Date
Page

④ **Alternate key (AK)** : All CK except PK

⑤ **foreign key** : ⇒ It creates relation between two tables
⇒ Its an attribute or set of attribute that references to primary key of same table or another table.

called referenced relation or Parent table

key :
→ customer (cust_id, name, address, contact no)
→ Order (order_id, timestamp, delivery date, cust_id)

called referencing relation or child table

F.K to create relation

Here :

Customer Table	Order table
PK : cust_id	order_id
CK : cust_id, contact_no	order_id
AK : contact_no	None
SK : {cust_id}, {contact_no}	{order_id}
{cust_id, contact no}	{order_id, cust_id}
{cust_id, address}	{order_id, timestamp}
... etc any superset containing cust_id or contact no for unique identification	... etc any superset containing order_id

* Every candidate key is a super key, but vice versa not true.

⑥ Composite key : PK formed using atleast 2 attributes

⑦ Compound key : PK which is formed using 2 P.K.

⑧ Surrogate key : Synthetic PK
→ Generated automatically by DB, usually Int value
→ May/be used as PK.

Eg: Table A

reg no.	name
101	Ram
102	Monika
103	tata

Sum_no.	reg_no.	name
1	101	Ram
2	102	Monika
3	103	tata
4	AB01	Tanmay
5	AB102	Nota

Table B

reg. no.	name
AB101	Tanmay
AB102	Nota

* Because both table have same reg_no for a particular tuple but we know candidate, are unique.

Integrity Constraints

- DB operations (CRUD) must be done integrity policy so that DB is always consistent.
Introduced so that we do not actually corrupt the DB.

Date

Page

① Domain Constraints:

① Restricts the value in the attribute of relation; specifies the domain

② Restricts the datatype of every attribute

- Eg: we want to specify that the enrollment should be happen for candidate 18th year ≥ 2003

```
CREATE TABLE Student (
    student_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    birth_year INT
        CHECK (birth_year < 2003),
)
```

* for Notes: Sql queries will be in small letters but for jobs it should be CAPS only.
  ⇒ writing SQL keywords in CAPS is convention not rule.

It is easy to distinguish b/w SQL keywords (CAPS) lowercase (table & columns name). SQL engine completely ignores case of keywords except when we put values in DB!

② Entity Constraints:

① Every relation should have PK: PK != NULL
  → basically tuples should be unique.

③ Referential Constraints:

→ specified between two relations & helps consistency among tuples of two relation!

customer :

cust_id	Name	add	contact no.
1	Kaz	— —	— — —
2	Jao	— —	— — —
3	dada	— —	— — —

↓ Parent or referenced table

F.K

Order :

order_id	timestamp	delivery date	cust_id
21	— — —	— — —	1
22	— — —	— — —	2
23	— — —	— — —	3
24	— — —	— — —	4 ✗

↓ Child or referencing table

Can not be added →

① Insert Constraint : Value can't be inserted in the child table if value is not in the parent table.

② Delete Constraint : Value can't be deleted from parent table, if it is present in child table.

On Delete Cascade :

Ques : Can we delete a value from parent table if the value is present in child table w/o violating delete constraint.

Ans : Yes → delete value from parent → delete corresponding value from child table to called on delete cascade

follow up question :-

Can FK value be Null ?

On delete Null : Rather than completely deleting the entry in F.K we can mention Null.

④ Key Constraints :

① Not Null : by default a column/attribute can be NULL.

(So,) → enforce Not Null

Eg: create table cust (
ID int not null,
Name varchar(50) not null,
age int
)

② Unique constraints : ensure all values in col are diff.
- Both unique & P.K constraint provide uniqueness.
- you may have many unique constraint per table but only one P.K constraint per table.

Eg: create table cust (
ID int not null,
Name varchar(50) not null,
unique (ID)
)

FREEMIND

Date

Page

③ Default constraint : set default value of col.

eg :

```
create table customer (
;
Prime_status int default 0,
;
)
```

④ check : - limit of value range.

eg :

```
_____ (
;
CHECK ( age >= 18 )
;
)
```

⑤ Primary key : uniquely identify each tuple
P.K ≠ NULL

eg :

```
_____ (
;
PRIMARY KEY (ID)
;
)
```

⑥ foreign key : keeps relation between two tables.

eg :

```
create table order (
;
PRIMARY KEY (order_ID),
FOREIGN KEY (cust_id)
  REFERENCES customer(cust_id)
);
```

LEC-8: Transform - ER Model to Relational Model

1. Both **ER**-Model and **Relational** Model are abstract logical representation of real world enterprises. Because the two models implies the similar design principles, **we can convert ER design into Relational design**.
2. Converting a DB representation from an ER diagram to a table format is the way we arrive at Relational DB-design from an ER diagram.
3. ER diagram **notations to relations**:
    1. **Strong Entity**
        1. Becomes an **individual table** with entity name, attributes becomes columns of the relation.
        2. Entity's Primary Key (PK) is used as Relation's PK.
        3. **FK** are added to establish relationships with other relations.
    2. **Weak Entity**
        1. A table is formed with all the attributes of the entity.
        2. PK of its corresponding Strong Entity will be added as **FK**.
        3. **PK** of the relation will be a composite PK, {FK + Partial discriminator Key}.
    3. **Single Values Attributes**
        1. Represented as **columns** directly in the tables/relations.
    4. **Composite Attributes**
        1. Handled by **creating a separate attribute** itself in the original relation for each composite attribute.
        2. e.g., **Address**: {street-name, house-no}, is a composite attribute in customer relation, we add address-street-name & address-house-name as new columns in the attribute and ignore "address" as an attribute.
    5. **Multivalued Attributes**
        1. **New tables** (named as original attribute name) are created for each multivalued attribute.
        2. PK of the entity is used as column **FK** in the new table.
        3. Multivalued attribute's similar name is added as a column to define multiple values.
        4. **PK** of the new table would be {FK + multivalued name}.
        5. e.g., For Strong entity **Employee**, **dependent-name** is a multivalued attribute.
            1. New table named dependent-name will be formed with columns emp-id, and dname.
            2. PK: {emp-id, name}
            3. FK: {emp-id}
    6. **Derived Attributes**: Not considered in the tables.
    7. **Generalisation**
        1. **Method-1:** Create a table for the higher level entity set. For each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the primary key of the higher-level entity set.
        For e.g., Banking System generalisation of Account - saving & current.
            1. Table 1: account (account-number, balance)
            2. Table 2: savings-account (account-number, interest-rate, daily-withdrawal-limit)
            3. Table 3: current-account (account-number, overdraft-amount, per-transaction-charges)
        2. **Method-2:** An alternative representation is possible, if the generalisation is disjoint and complete—that is, if no entity is a member of two lower-level entity sets directly below a higher-level entity set, and if every entity in the higher level entity set is also a member of one of the lower-level entity sets. Here, do not create a table for the higher-level entity set. Instead, for each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the higher-level entity sets.
        Tables would be:
            1. Table 1: savings-account (account-number, balance, interest-rate, daily-withdrawal-limit)
            2. Table 2: current-account (account-number, balance, overdraft-amount, per-transaction-charges)
        3. **Drawbacks of Method-2:** If the second method were used for an overlapping generalisation, some values such as balance would be stored twice unnecessarily. Similarly, if the generalisation were not complete—that is, if some accounts were neither savings nor current accounts—then such accounts could not be represented with the second method.
    8. **Aggregation**

---

1. Table of the relationship set is made.
2. Attributes includes primary keys of entity set and aggregation set's entities.
3. Also, add descriptive attribute if any on the relationship.

Lec-8

*Transformation from ER model to Relational model.

Loan → | Loan-number | amount |

Weak Entity Payment →

F.K

| Loan-number | Payment no. | Payment-date | Payment-amount |

P.K

* Compunt attribut → → sep. attribute for each compunet.

Customer table

Custmer-name	addres-city	add-state	add-pincode	add-streetno.	add-streetname

* Multi-value attribute

F.K	dependent-name

-new relation.

emp_id	d name.

{emp_id, dname}

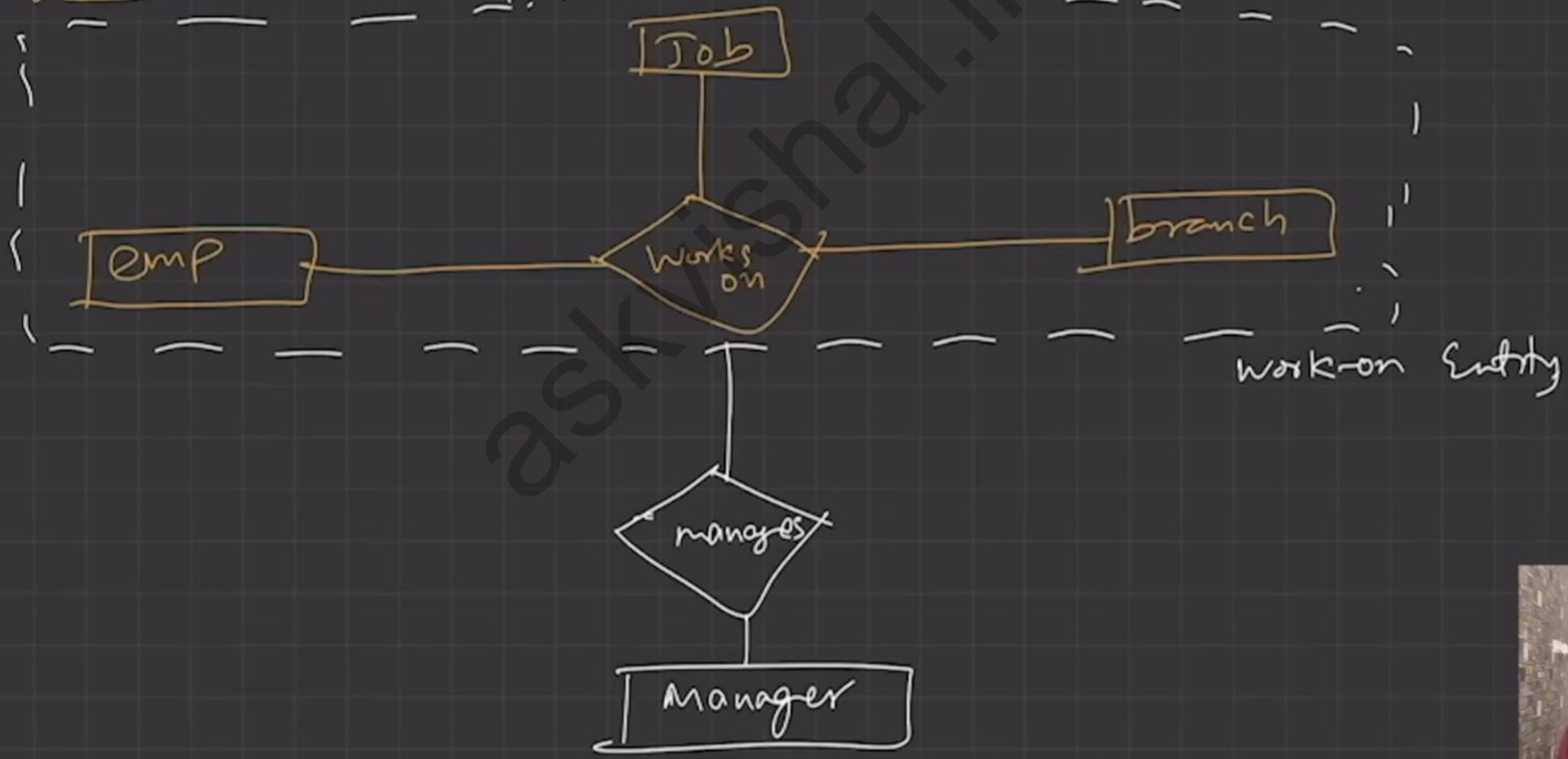
↓

P.K..

* Generalization

Method 1

* Aggregation =>



* Aggregation ⇒

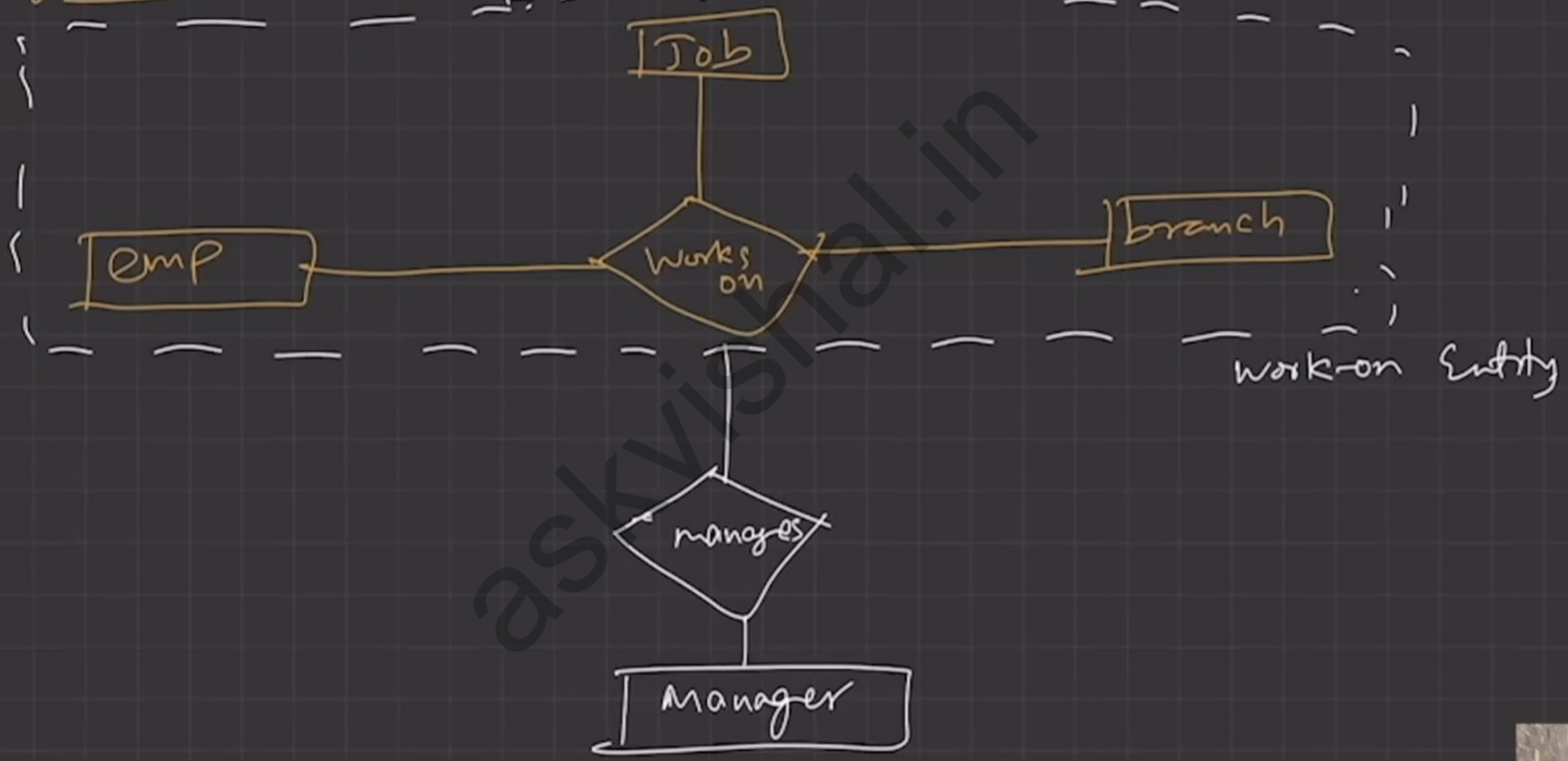
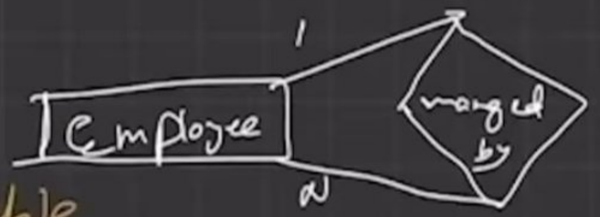


Table: manages (mgr_id, emp_id, job_id, branch_id)

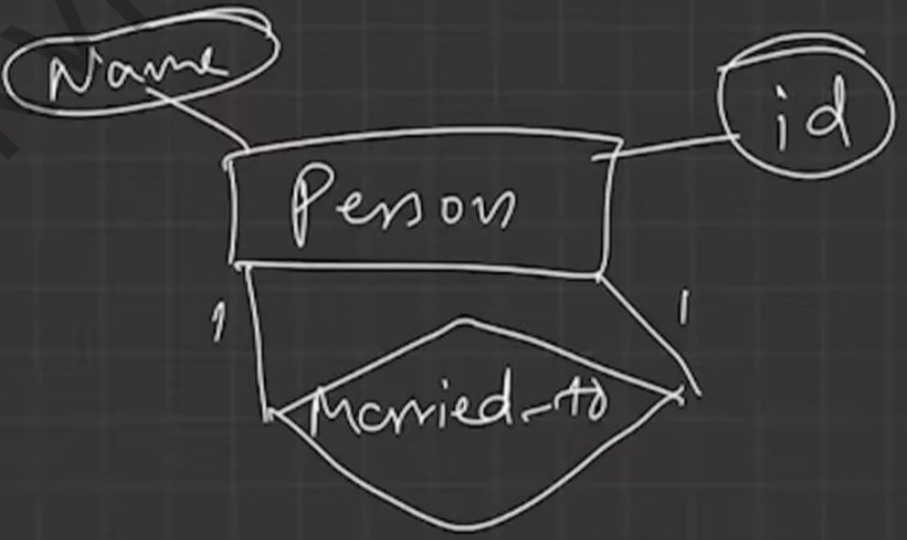
* Unary Relationship :-



we will add another attribute in Employee table.
which will be F.K.

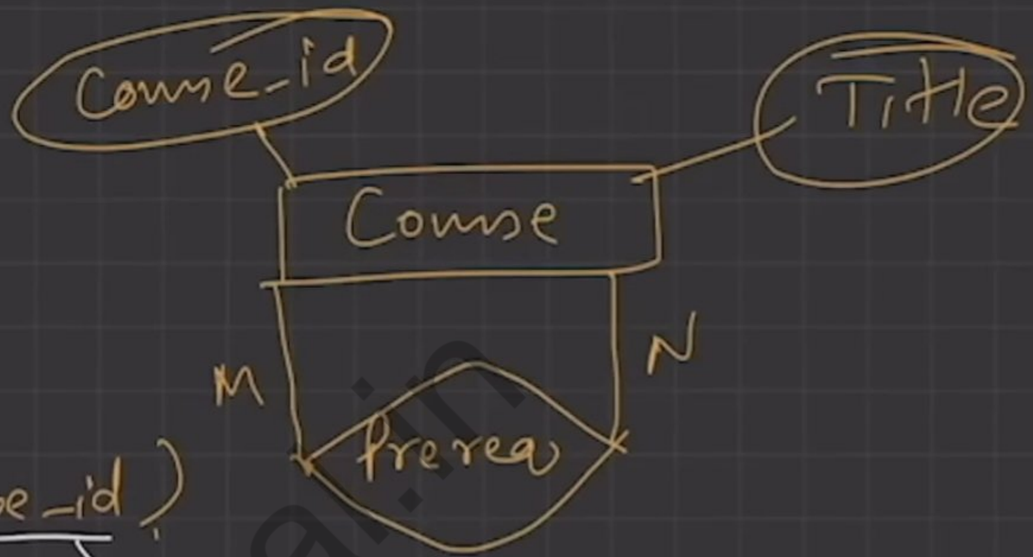
Emp_id	name	joining date	Emp_mgr_id
201	---	---	205
202	---	---	205
205	---	---	Null.

* 1:1 →



Person(id, Name, spouse_id)
{F.K}

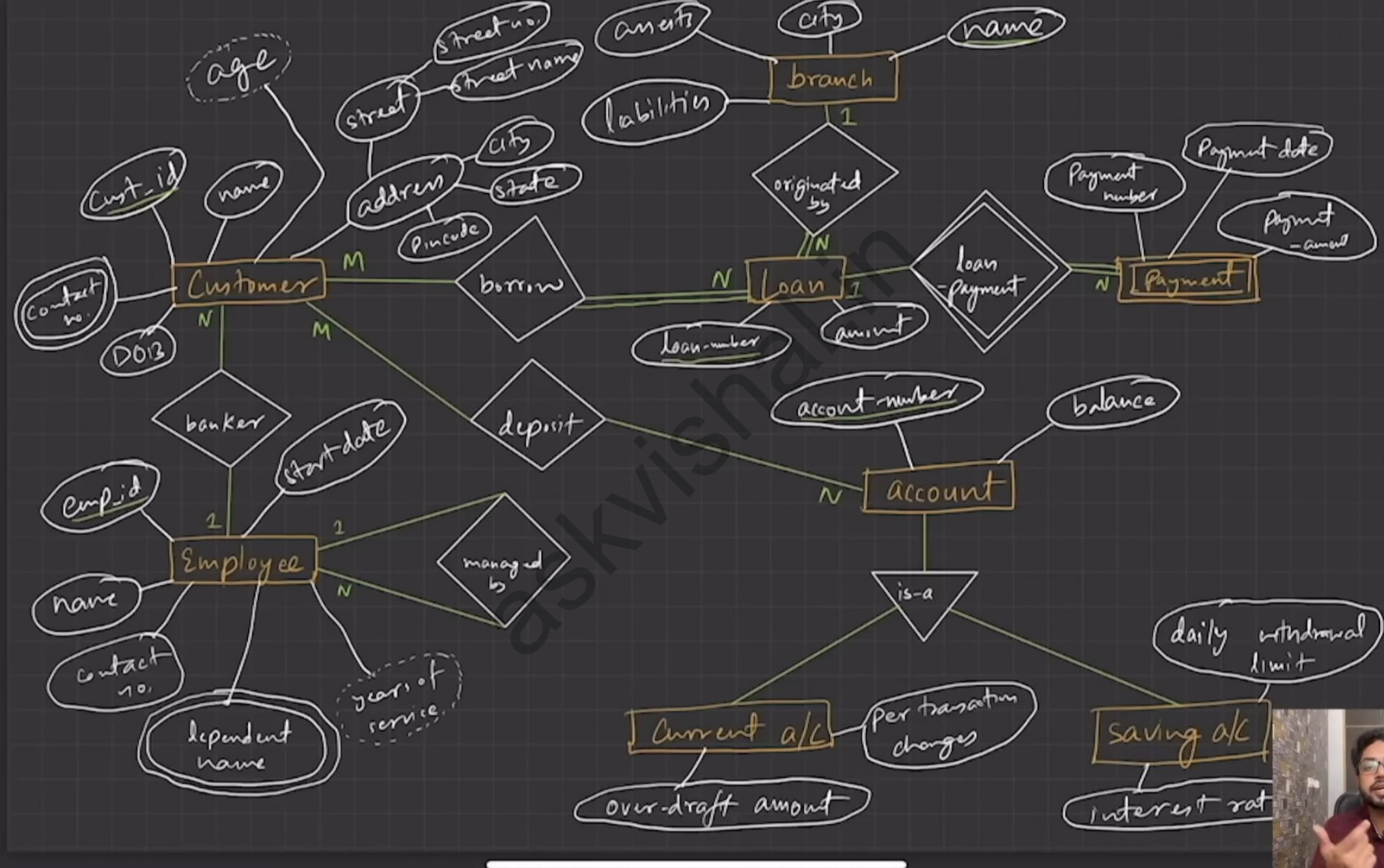
* M : N ⇒



① Course (id → Title)

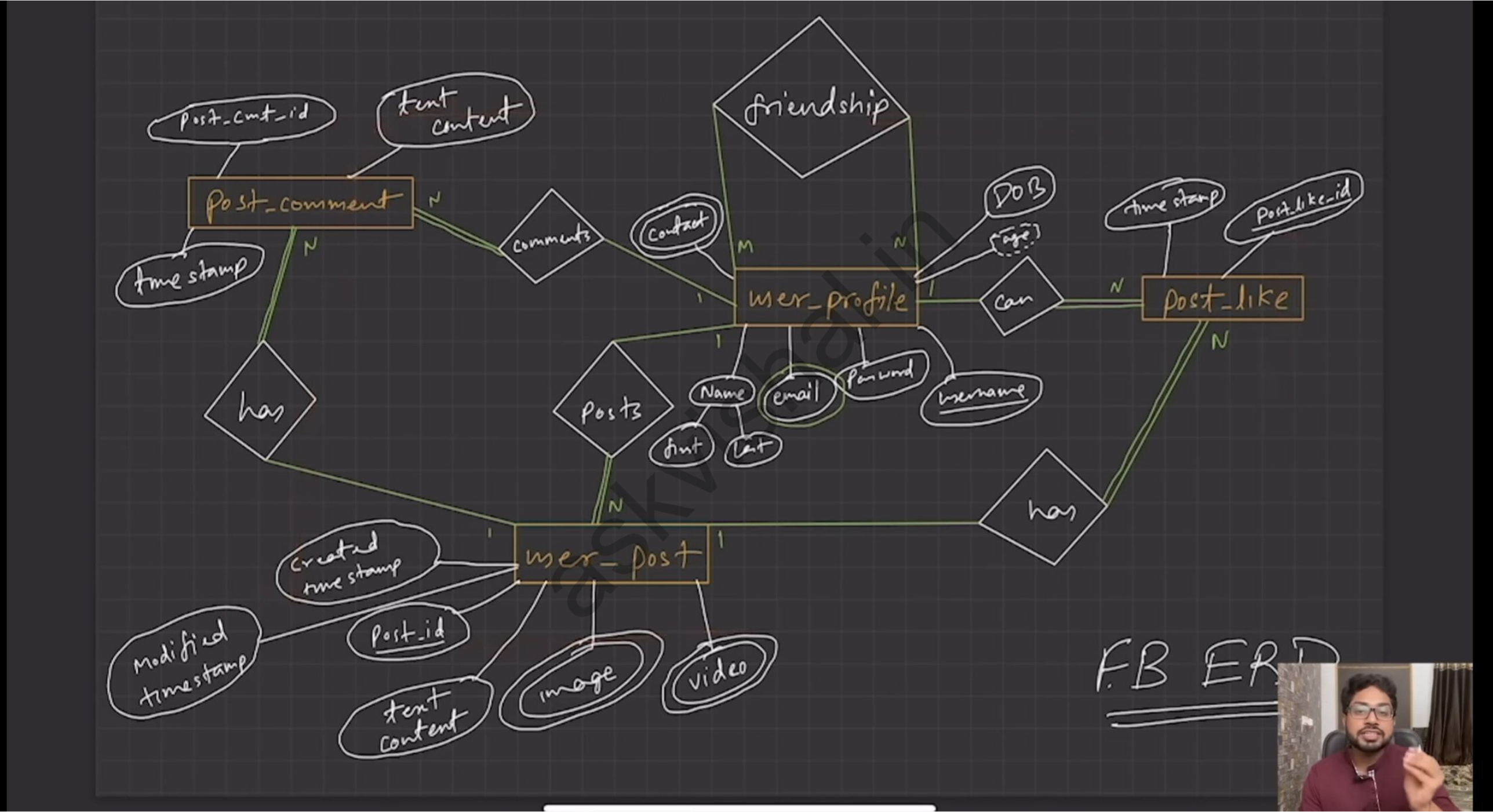
② Prereq (id , Prereq_Course_id)

{FK} {FK} → P.K


age
street no.
street name
street
assets
city
name
branch
liabilities
1
Cust_id
name
address
city
state
Pincode
originated by
Payment number
Payment date
Payment amount
contact no.
Customer
M
borrow
N
Loan
1
loan-payment
N
Payment
DOB
N
M
amount
loan-number
banker
deposit
account-number
balance
start date
emp_id
1
1
Employee
N
N
account
managed by
name
is-a
contact no.
dependent name
years of service
daily withdrawal limit
Current a/c
Per transaction charges
saving a/c
over-draft amount
interest rat

* FB Relational Model

(1) user_profile (username, name-first, name-last, password, DOB)
(2) user-profile-email (username {F.K}, email)
(3) user-profile-contact (username {f.k}, contact-number)
(4) friendship (profile_req {f.k}, profile_accept {f.k}) ⟶ Compound key
(5) post-like (post_like_id, timestamp, post_id {f.k}, username {f.k})
(6) user-post (post-id, created-timestamp, modified-timestamp, text-content, username {f.k})
(7) user-post-image (post_id {f.k}, image_url)
(8) user-post-video (post-id {F.K}, video_url)
(9) post-comment (post-comment-id, text-content, timestamp, post-id {f.k}, username {f.k})


Post_cmt_id
text content
friendship
Post_comment
N
comments
Contact
M
N
DOB
{age}
time stamp
Post_like_id
time stamp
N
N
user_profile
1
can
N
post_like
has
1
Posts
Name
email
Password
username
first
last
N
has
1
1
user_post
Created time stamp
Modified timestamp
Post_id
text content
image
video
FB ERD

LEC-9: SQL in 1-Video

1. **SQL**: Structured Query Language, used to access and manipulate data.
2. SQL used **CRUD** operations to communicate with DB.
    1. **CREATE** - execute INSERT statements to insert new tuple into the relation.
    2. **READ** - Read data already in the relations.
    3. **UPDATE** - Modify already inserted data in the relation.
    4. **DELETE** - Delete specific data point/tuple/row or multiple rows.
3. **SQL is not DB, is a query language.**
4. What is **RDBMS**? (Relational Database Management System)
    1. Software that enable us to implement designed relational model.
    2. e.g., MySQL, MS SQL, Oracle, IBM etc.
    3. Table/Relation is the simplest form of data storage object in R-DB.
    4. **MySQL** is open-source RDBMS, and it uses SQL for all CRUD operations
5. **MySQL** used client-server model, where client is CLI or frontend that used services provided by MySQL server.
6. **Difference between SQL and MySQL**
    1. SQL is Structured Query language used to perform CRUD operations in R-DB, while MySQL is a RDBMS used to store, manage and administrate DB (provided by itself) using SQL.

SQL DATA TYPES (Ref: https://www.w3schools.com/sql/sql_datatypes.asp)

1. In SQL DB, data is stored in the form of tables.
2. Data can be of different types, like INT, CHAR etc.

DATATYPE	Description
CHAR	string(0-255), string with size = (0, 255], e.g., CHAR(251)
VARCHAR	string(0-255)
TINYTEXT	String(0-255)
TEXT	string(0-65535)
BLOB	string(0-65535)
MEDIUMTEXT	string(0-16777215)
MEDIUMBLOB	string(0-16777215)
LONGTEXT	string(0-4294967295)
LONGBLOB	string(0-4294967295)
TINYINT	integer(-128 to 127)
SMALLINT	integer(-32768 to 32767)
MEDIUMINT	integer(-8388608 to 8388607)
INT	integer(-2147483648 to 2147483647)
BIGINT	integer (-9223372036854775808 to 9223372036854775807)
FLOAT	Decimal with precision to 23 digits
DOUBLE	Decimal with 24 to 53 digits

DATATYPE	Description
DECIMAL	Double stored as string
DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS
TIMESTAMP	YYYYMMDDHHMMSS
TIME	HH:MM:SS
ENUM	One of the preset values
SET	One or many of the preset values
BOOLEAN	0/1
BIT	e.g., BIT(n), n upto 64, store values in bits.

3. Size: TINY < SMALL < MEDIUM < INT < BIGINT.
4. **Variable length Data types** e.g., VARCHAR, are better to use as they occupy space equal to the actual data size.
5. Values can also be unsigned e.g., INT UNSIGNED.
6. **Types of SQL commands:**
    1. **DDL** (data definition language): defining relation schema.
        1. **CREATE**: create table, DB, view.
        2. **ALTER TABLE**: modification in table structure. e.g, change column datatype or add/remove columns.
        3. **DROP**: delete table, DB, view.
        4. **TRUNCATE**: remove all the tuples from the table.
        5. **RENAME**: rename DB name, table name, column name etc.
    2. **DRL/DQL** (data retrieval language / data query language): retrieve data from the tables.
        1. **SELECT**
    3. **DML** (data modification language): use to perform modifications in the DB
        1. **INSERT**: insert data into a relation
        2. **UPDATE**: update relation data.
        3. **DELETE**: delete row(s) from the relation.
    4. **DCL** (Data Control language): grant or revoke authorities from user.
        1. **GRANT**: access privileges to the DB
        2. **REVOKE**: revoke user access privileges.
    5. **TCL** (Transaction control language): to manage transactions done in the DB
        1. **START TRANSACTION**: begin a transaction
        2. **COMMIT**: apply all the changes and end transaction
        3. **ROLLBACK**: discard changes and end transaction
        4. **SAVEPOINT**: checkout within the group of transactions in which to rollback.

MANAGING DB (DDL)

1. **Creation of DB**
    1. **CREATE DATABASE IF NOT EXISTS db-name;**
    2. **USE** db-name; //need to execute to choose on which DB CREATE TABLE etc commands will be executed. //make switching between DBs possible.
    3. **DROP** DATABASE IF EXISTS db-name; //dropping database.
    4. **SHOW** DATABASES; //list all the DBs in the server.
    5. **SHOW** TABLES; //list tables in the selected DB.

DATA RETRIEVAL LANGUAGE (DRL)

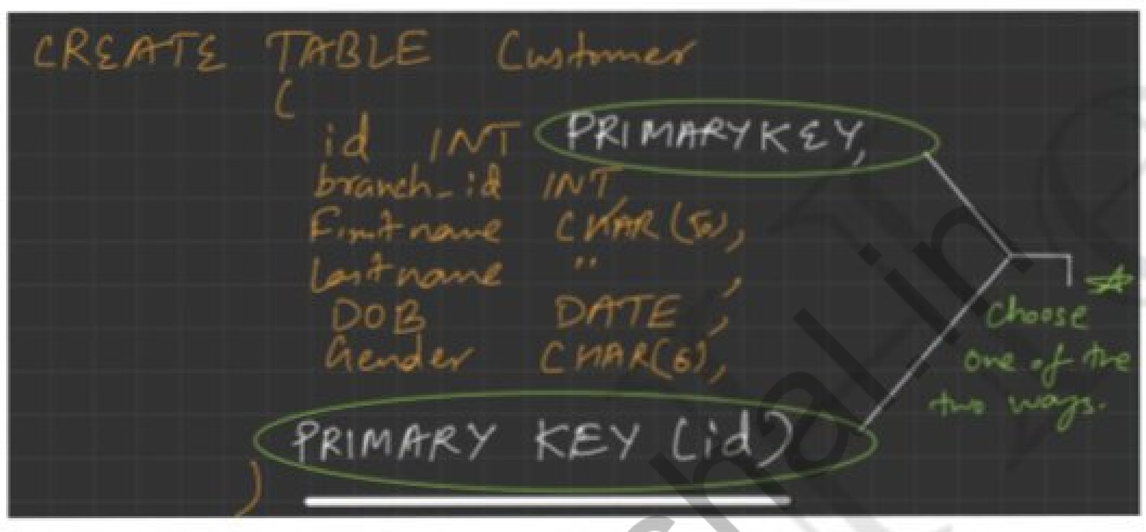
1. Syntax: SELECT <set of column names> FROM <table_name>;
2. Order of execution from RIGHT to LEFT.
3. Q. Can we use SELECT keyword without using FROM clause?
    1. Yes, using DUAL Tables.
    2. Dual tables are dummy tables created by MySQL, help users to do certain obvious actions without referring to user defined tables.
    3. e.g., SELECT 55 + 11;
       SELECT now();
       SELECT ucase(); etc.
4. **WHERE**
    1. Reduce rows based on given conditions.
    2. E.g., SELECT * FROM customer WHERE age > 18;
5. **BETWEEN**
    1. SELECT * FROM customer WHERE age between 0 **AND** 100;
    2. In the above e.g., 0 and 100 are inclusive.
6. **IN**
    1. Reduces **OR** conditions;
    2. e.g., SELECT * FROM officers WHERE officer_name IN ('Lakshay', 'Maharana Pratap', 'Deepika');
7. **AND/OR/NOT**
    1. AND: WHERE cond1 AND cond2
    2. OR: WHERE cond1 OR cond2
    3. NOT: WHERE col_name NOT IN (1,2,3,4);
8. **IS NULL**
    1. e.g., SELECT * FROM customer WHERE prime_status is NULL;
9. **Pattern Searching / Wildcard ('%', '_')**
    1. '%', any number of character from 0 to n. Similar to '*' asterisk in regex.
    2. '_', only one character.
    3. SELECT * FROM customer WHERE name **LIKE** '%p_';
10. **ORDER BY**
    1. Sorting the data retrieved using **WHERE** clause.
    2. ORDER BY <column-name> DESC;
    3. DESC = Descending and ASC = Ascending
    4. e.g., SELECT * FROM customer ORDER BY name DESC;
11. **GROUP BY**
    1. GROUP BY Clause is used to collect data from multiple records and group the result by one or more column. It is generally used in a SELECT statement.
    2. Groups into category based on column given.
    3. SELECT c1, c2, c3 FROM sample_table WHERE cond GROUP BY c1, c2, c3.
    4. All the column names mentioned after SELECT statement shall be repeated in GROUP BY, in order to successfully execute the query.
    5. Used with aggregation functions to perform various actions.
        1. COUNT()
        2. SUM()
        3. AVG()
        4. MIN()
        5. MAX()
12. **DISTINCT**
    1. Find distinct values in the table.
    2. SELECT DISTINCT(col_name) FROM table_name;
    3. GROUP BY can also be used for the same
        1. "Select col_name from table GROUP BY col_name;" same output as above DISTINCT query.

2. SQL is smart enough to realise that if you are using GROUP BY and not using any aggregation function, then you mean "DISTINCT".

13. **GROUP BY HAVING**
    1. Out of the categories made by GROUP BY, we would like to know only particular thing (cond).
    2. Similar to WHERE.
    3. Select COUNT(cust_id), country from customer GROUP BY country HAVING COUNT(cust_id) > 50;
    4. WHERE vs HAVING
        1. Both have same function of filtering the row base on certain conditions.
        2. WHERE clause is used to filter the rows from the table based on specified condition
        3. HAVING clause is used to filter the rows from the groups based on the specified condition.
        4. HAVING is used after GROUP BY while WHERE is used before GROUP BY clause.
        5. If you are using HAVING, GROUP BY is necessary.
        6. WHERE can be used with SELECT, UPDATE & DELETE keywords while GROUP BY used with SELECT.

CONSTRAINTS (DDL)

1. **Primary Key**



    1. PK is not null, unique and only one per table.
2. **Foreign Key**
    1. FK refers to PK of other table.
    2. Each relation can having any number of FK.
    3. CREATE TABLE ORDER (
       id INT PRIMARY KEY,
       delivery_date DATE,
       order_placed_date DATE,
       cust_id INT,
       FOREIGN KEY (cust_id) REFERENCES customer(id)
       );
3. **UNIQUE**
    1. Unique, can be null, table can have multiple unique attributes.
    2. CREATE TABLE customer (
       ...
       email VARCHAR(1024) UNIQUE,
       ...
       );
4. **CHECK**
    1. CREATE TABLE customer (
       ...
       CONSTRAINT age_check CHECK (age > 12),
       ...
       );
    2. "age_check", can also avoid this, MySQL generates name of constraint automatically.

5. **DEFAULT**
    1. Set default value of the column.
    2. CREATE TABLE account (
       ...
       saving-rate DOUBLE NOT NULL DEFAULT 4.25,
       ...
       );
6. An attribute can be **PK and FK both** in a table.
7. **ALTER OPERATIONS**
    1. Changes schema
    2. **ADD**
        1. **Add new column.**
        2. ALTER TABLE table_name ADD new_col_name datatype ADD new_col_name_2 datatype;
        3. e.g., ALTER TABLE customer ADD age INT NOT NULL;
    3. **MODIFY**
        1. **Change datatype of an attribute.**
        2. ALTER TABLE table-name MODIFY col-name col-datatype;
        3. E.g., VARCHAR TO CHAR
           ALTER TABLE customer MODIFY name CHAR(1024);
    4. **CHANGE COLUMN**
        1. **Rename column name.**
        2. ALTER TABLE table-name CHANGE COLUMN old-col-name new-col-name new-col-datatype;
        3. e.g., ALTER TABLE customer CHANGE COLUMN name customer-name VARCHAR(1024);
    5. **DROP COLUMN**
        1. **Drop a column completely.**
        2. ALTER TABLE table-name DROP COLUMN col-name;
        3. e.g., ALTER TABLE customer DROP COLUMN middle-name;
    6. **RENAME**
        1. **Rename table name itself.**
        2. ALTER TABLE table-name RENAME TO new-table-name;
        3. e.g., ALTER TABLE customer RENAME TO customer-details;

DATA MANIPULATION LANGUAGE (DML)

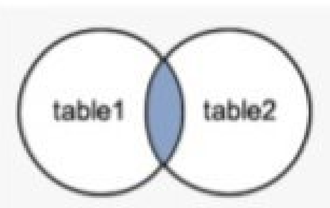
1. **INSERT**
    1. INSERT INTO table-name(col1, col2, col3) VALUES (v1, v2, v3), (val1, val2, val3);
2. **UPDATE**
    1. UPDATE table-name SET col1 = 1, col2 = 'abc' WHERE id = 1;
    2. Update multiple rows e.g.,
        1. UPDATE student SET standard = standard + 1;
    3. **ON UPDATE CASCADE**
        1. Can be added to the table while creating constraints. Suppose there is a situation where we have two tables such that primary key of one table is the foreign key for another table. if we update the primary key of the first table then using the ON UPDATE CASCADE foreign key of the second table automatically get updated.
3. **DELETE**
    1. DELETE FROM table-name WHERE id = 1;
    2. DELETE FROM table-name; //all rows will be deleted.
    3. **DELETE CASCADE - (to overcome DELETE constraint of Referential constraints)**
        1. What would happen to child entry if parent table's entry is deleted?
        2. CREATE TABLE ORDER (
           order_id int PRIMARY KEY,
           delivery_date DATE,
           cust_id INT,

FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE CASCADE
);

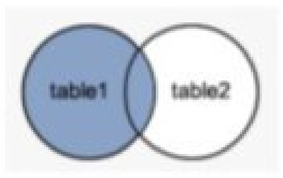
3. **ON DELETE NULL - (can FK have null values?)**
    1. CREATE TABLE ORDER (
    order_id int PRIMARY KEY,
    delivery_date DATE,
    cust_id INT,
    FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE SET NULL
    );
4. **REPLACE**
    1. Primarily used for already present tuple in a table.
    2. As UPDATE, using REPLACE with the help of WHERE clause in PK, then that row will be replaced.
    3. As INSERT, if there is no duplicate data new tuple will be inserted.
    4. REPLACE INTO student (id, class) VALUES(4, 3);
    5. REPLACE INTO table SET col1 = val1, col2 = val2;

JOINING TABLES

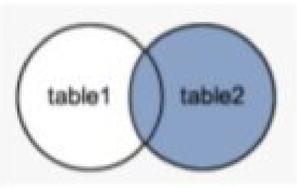
1. All **RDBMS** are relational in nature, we refer to other tables to get meaningful outcomes.
2. FK are used to do reference to other table.
3. **INNER JOIN**
    1. Returns a resultant table that has matching values from both the tables or all the tables.
    2. SELECT column-list FROM table1 INNER JOIN table2 ON condition1
    INNER JOIN table3 ON condition2
    ...;



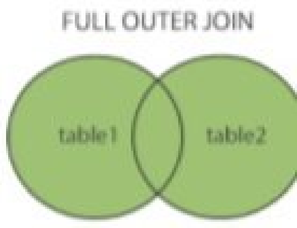
    3. **Alias in MySQL (AS)**
        1. Aliases in MySQL is used to give a temporary name to a table or a column in a table for the purpose of a particular query. It works as a nickname for expressing the tables or column names. It makes the query short and neat.
        2. SELECT col_name AS alias_name FROM table_name;
        3. SELECT col_name1, col_name2,... FROM table_name AS alias_name;
4. **OUTER JOIN**
    1. **LEFT JOIN**
        1. This returns a resulting table that all the data from left table and the matched data from the right table.
        2. SELECT columns FROM table LEFT JOIN table2 ON Join_Condition;



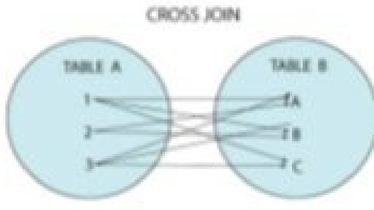
    2. **RIGHT JOIN**
        1. This returns a resulting table that all the data from right table and the matched data from the left table.
        2. SELECT columns FROM table RIGHT JOIN table2 ON join_cond;



    3. **FULL JOIN**
        1. This returns a resulting table that contains all data when there is a match on left or right table data.
        2. **Emulated** in MySQL using LEFT and RIGHT JOIN.
        3. LEFT JOIN UNION RIGHT JOIN.
        4. SELECT columns FROM table1 as t1 LEFT JOIN table2 as t2 ON t1.id = t2.id
        UNION
        SELECT columns FROM table1 as t1 RIGHT JOIN table2 as t2 ON t1.id = t2.id;
        5. UNION ALL, can also be used this will duplicate values as well while UNION gives unique values.



5. **CROSS JOIN**
    1. This returns all the cartesian products of the data present in both tables. Hence, all possible variations are reflected in the output.
    2. Used rarely in practical purpose.
    3. Table-1 has 10 rows and table-2 has 5, then resultant would have 50 rows.
    4. SELECT column-lists FROM table1 CROSS JOIN table2;



6. **SELF JOIN**

1. It is used to get the output from a particular table when the same table is joined to itself.
2. Used very less.
3. Emulated using INNER JOIN.
4. SELECT columns FROM table as t1 INNER JOIN table as t2 ON t1.id = t2.id;

7. **Join without using join keywords.**
   1. SELECT * FROM table1, table2 WHERE condition;
   2. e.g., SELECT artist_name, album_name, year_recordedFROM artist, albumWHERE artist.id = album.artist_id;

SET OPERATIONS

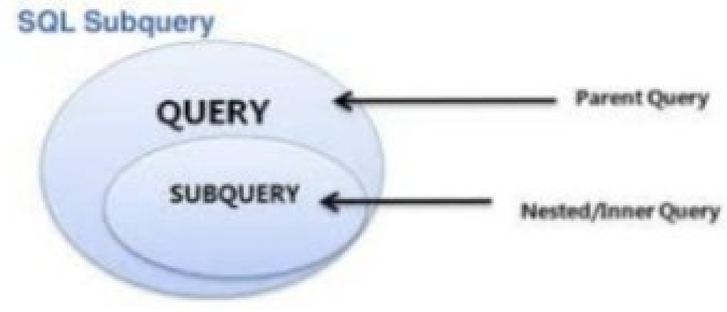
1. Used to combine multiple select statements.
2. Always gives distinct rows.

JOIN	SET Operations
Combines multiple tables based on matching condition.	Combination is resulting set from two or more SELECT statements.
Column wise combination.	Row wise combination.
Data types of two tables can be different.	Datatypes of corresponding columns from each table should be the same.
Can generate both distinct or duplicate rows.	Generate distinct rows.
The number of column(s) selected may or may not be the same from each table.	The number of column(s) selected must be the same from each table.
Combines results horizontally.	Combines results vertically.

3. **UNION**
   1. Combines two or more SELECT statements.
   2. SELECT * FROM table1
      UNION
      SELECT * FROM table2;
   3. Number of column, order of column must be same for table1 and table2.
4. **INTERSECT**
   1. Returns common values of the tables.
   2. Emulated.
   3. SELECT DISTINCT column-list FROM table-1 INNER JOIN table-2 USING(join_cond);
   4. SELECT DISTINCT * FROM table1 INNER JOIN table2 ON USING(id);
5. **MINUS**
   1. This operator returns the distinct row from the first table that does not occur in the second table.
   2. Emulated.
   3. SELECT column_list FROM table1 LEFT JOIN table2 ON condition WHERE table2.column_name IS NULL;
   4. e.g., SELECT id FROM table-1 LEFT JOIN table-2 USING(id) WHERE table-2.id IS NULL;

SUB QUERIES

1. Outer query depends on inner query.
2. Alternative to joins.
3. Nested queries.
4. SELECT column_list (s) FROM table_name WHERE column_name OPERATOR (SELECT column_list (s) FROM table_name [WHERE]);
5. e.g., SELECT * FROM table1 WHERE col1 IN (SELECT col1 FROM table1);
6. Sub queries exist mainly in 3 clauses
   1. Inside a WHERE clause.



2. Inside a FROM clause.
3. Inside a SELECT clause.

7. **Subquery using FROM clause**
   1. SELECT MAX(rating) FROM (SELECT * FROM movie WHERE country = 'India') as temp;
8. **Subquery using SELECT**
   1. SELECT (SELECT column_list(s) FROM T_name WHERE condition), columnList(s) FROM T2_name WHERE condition;
9. **Derived Subquery**
   1. SELECT columnLists(s) FROM (SELECT columnLists(s) FROM table_name WHERE [condition]) as new_table_name;
10. **Co-related sub-queries**
    1. With a normal nested subquery, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

```
SELECT column1, column2, ....
FROM table1 as outer
WHERE column1 operator
                      (SELECT column1, column2
                       FROM table2
                       WHERE expr1 =
                                  outer.expr2);
```

JOIN VS SUB-QUERIES

JOINS	SUBQUERIES
Faster	Slower
Joins maximise calculation burden on DBMS	Keeps responsibility of calculation on user.
Complex, difficult to understand and implement	Comparatively easy to understand and implement.
Choosing optimal join for optimal use case is difficult	Easy.

MySQL VIEWS

1. A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table.
2. In MySQL, the View is a **virtual table** created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own.
3. The View and table have one main difference that the views are definitions built on top of other tables (or views). If any changes occur in the underlying table, the same changes reflected in the View also.
4. CREATE VIEW view_name AS SELECT columns FROM tables [WHERE conditions];
5. ALTER VIEW view_name AS SELECT columns FROM table WHERE conditions;
6. DROP VIEW IF EXISTS view_name;
7. CREATE VIEW Trainer AS SELECT c.course_name, c.trainer, t.email FROM courses c, contact t WHERE c.id = t.id; (View using Join clause).

NOTE: We can also import/export table schema from files (.csv or json).

Normalisation

A step towards DB optimisation.

Functional dependency (FD) :
relationship between primary key attribute (generally) of the relation to that of the other attributes of relation.
⟹ x pata ho .. to Y nikal lenge.

Given → x → y ← to find
x → Determinant
y → Dependent

type of FD

ⓐ Trivial FD : A → B , B is a subset of A. ($B \subseteq A$)
eg : {emp_id, Name} → emp_id

ⓑ Non trivial FD : $A \to B, B \not\subseteq A$; $A \cap B = NULL$
eg : {emp_id, Name} → address

Rules of FD (Armstrong's axioms)

a) **Reflexive** : Y is a subset of x, then, x → y
eg : x = {a, b, c, d, e}
Subset Y = {a, b, c}
then x → y.

b) **Augmentation** : If B can be determined from A, then adding an attribute to the functional dependency won't change anything.
Relation
eg : x → y ; R(X Y, Z)
∴ XZ → YZ

FREEMIND

Date

Page

c) Transitive : If $A \to B$ & $B \to C$
then, $A \to C$.

Eg : Author_Book (Author_name, Book_title, age)

Book_title → Author_name
Author_name → Age

then,

Book_title → Age

Anomalies

1. Anomalies means abnormalities, there are three types of anomalies introduced by data redundancy.
2. **Insertion anomaly**
    1. When certain data (attribute) can not be inserted into the DB without the presence of other data.
3. **Deletion anomaly**
    1. The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
4. **Updation anomaly** (or modification anomaly)
    1. The update anomaly is when an update of a single data value requires multiple rows of data to be updated.
    2. Due to updation to many places, may be **Data inconsistency** arises, if one forgets to update the data at all the intended places.
5. Due to these anomalies, **DB size increases** and **DB performance become very slow.**
6. To rectify these anomalies and the effect of these of DB, we use **Database optimisation technique** called **NORMALISATION**.

What is Normalisation?

1. Normalisation is used to minimise the redundancy from a relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
2. Normalisation divides the composite attributes into individual attributes OR larger table into smaller and links them using relationships.
3. The normal form is used to reduce redundancy from the database table.

Date

Page

What do we do in normalisation :

Table → decompose → into multiple tables till SRP is achieved.

↳ Single Responsibility Principle.

Types of Normalisation / Normal Forms

① 1NF : Every relation cell must have atomic value.
Relation must not have multi-valued attributes.

② 2NF : · Relation must be 1NF.
→ There should not be any partial dependency.
① All non prime attributes must be fully depend on PK.
② Non prime attribute can not depend on the part of PK.

⟹ R (A B C D)
↳ {A,B} → PK
A,B → Prime attribute, C,D → Non Prime attribute

FD ⇒ B → C ← Partial dependent ✗

AB → C ✓ (should always be full)
AB → D

Also, R₁ (ABD) ⇒ AB → D
R₂ (BC) ⇒ B → C ← itself is a seprate table

2NF Table (Student Project)

StudentID	ProjID	StudentName	Project name
S89	P09	Aria	Geo
S76	P07	Jacob	Chatbot
S56	P03	Ava	IOT
S92	P05	Alex	Cloud

P.K : {StudentID, ProjID}

FD :⇒ StudentID → StudentName

ProjectID → ProjectName

2NF form

Student

Student ID	Project ID	Student Name
S89	P09	Olvia
S76	P07	Jacob
S56	P03	Ava
S92	P05	alex

Project

Project ID	Project Name
P09	Geo
P07	Chitter
P03	IoT
P05	Cloud

3NF : Relation must be 2NF.

→ No transitive dependency exists.

→ Non prime attribute should not find a non prime attribute.

⟹ #3NF : R(ABC)

P.K = {A}

F.D ⇒ A → B ,

why need? ⟸ B → C

If : A → C. (directly)

3NF

R(A B C)

P.K { A }

F.D :- A → B
B → C

2NF ✓

⇒

A	B	C
a	1	x
b	1	x
c	1	x
d	2	y
e	2	y
f	3	z
g	3	z

A → Prime.

B → C

↓ Non-P

→ Prime.

$B \Rightarrow C$ — F.D (Transitive dependency)

2) decompose 3NF

$R_1(\underline{A}\,B)$ $R_2(\underline{B}\,C)$

R_1

A	B
a	1
b	1
c	1
d	2
e	2
f	3
g	3

B	C
1	x
2	y
3	z.

4. **BCNF (Boyce-Codd normal form)**
    1. Relation must be in 3NF.
    2. FD: A -> B, A must be a super key.
        1. We must not derive prime attribute from any prime or non-prime attribute.

8. **Advantages of Normalisation**
    1. Normalisation helps to minimise data redundancy.
    2. Greater overall database organisation.
    3. Data consistency is maintained in DB.

BCNF

eg

Stud_ID	Subject	Professor
101	Java	PJ
101	CPP	PC
102	Java	PJ2
103	C#	PC#
104	Java	PJ

— One student can enroll in multiple subjects
— for each subject, a professor is assigned to a student
— multiple professor can teach a single subject
— One professor can teach only one subject

P.K = ? {Stud_ID, Subject}

① {Stud_ID, Subject} → Professor
② Professor → Subject

=> BCNF convers

Student

id	p-id.
101	1
101	2
:	:
(	(

Professor

p-id	Professor	subject
1	PJ	Java
2	PC	CPP.
:	:	:
(	(	(

FREEMIND

Date

Page

Transactions

Task : Transfer ₹50 from A to B.

- for user : 1 step operation,
but for DB :

T_i :
```
read (A)
A = A - 50;
write (A);
read (B)
B = B + 50
write (B)
```

→ all 6 (logical) are connected.

Steps:

⇓

A Single transaction

Transactions are Always atomic operation.

→ either complete or totally fail.
↳ rollback.

ACID Properties

to ensure the integrity of data, DB system maintains the following properties of the transaction.

① A = Atomicity :

↳ this all operations of the transaction are reflected properly in DB or none.

Rollback →

Key:	A	B
	read 1000	
	950 = 1000 - 50	
	write (A)	
		read 2000
		2050 = 2000 + 50
		write (B)

Assume system crash.
↓
150 lost if operation is not atomic

So, DB maintains : → Old State
↓ (needed if rollback)
intermediate State
↓ (if success)
latest state.

② C = Consistency :-
- Integrity constraints must be maintained before & after transaction
→ DB must be consistent after transaction.

③ I = Isolation :-
→ Even though multiple transaction may execute concurrently, the system guarantees that, for every pair of transaction T_i & T_j, it appears to T_i that either T_j finished execution before T_i started or T_j started the execution after T_i finished.

Thus each transaction is unaware of other transactions executing concurrently in

the system, without interfering in each other.

→ Multiple transaction can happen in the system. in isolation without interfering in each other.

eg: Banking System:

T_1 T_2 T_3 T_4

T_0 = T_1 (Google Pay) : T_2 (Net Banking)

read (A) ⇒ 1000	read A ⇒ 1000
A = A − 50 ⇒ 950	1000 − 50 = 950
A = 950	write (950)
write (950)	

B → 50 + 50 → extra Money

→ As both transaction started at same time.

Sol^n: through the req. comes at same time one will either be waiting & 1st second execute. (based on ms diff) or would get rolled back.

(4) D = Durability

→ After transaction completes successfully, the changes it has to made to the database persist, even if there are system failure.

eg:

T_1 → Succes

↓ all 6 steps.

↓

DB updated permanent (persist)

↳ even if system failure after T completion.

should recover

Date

Page

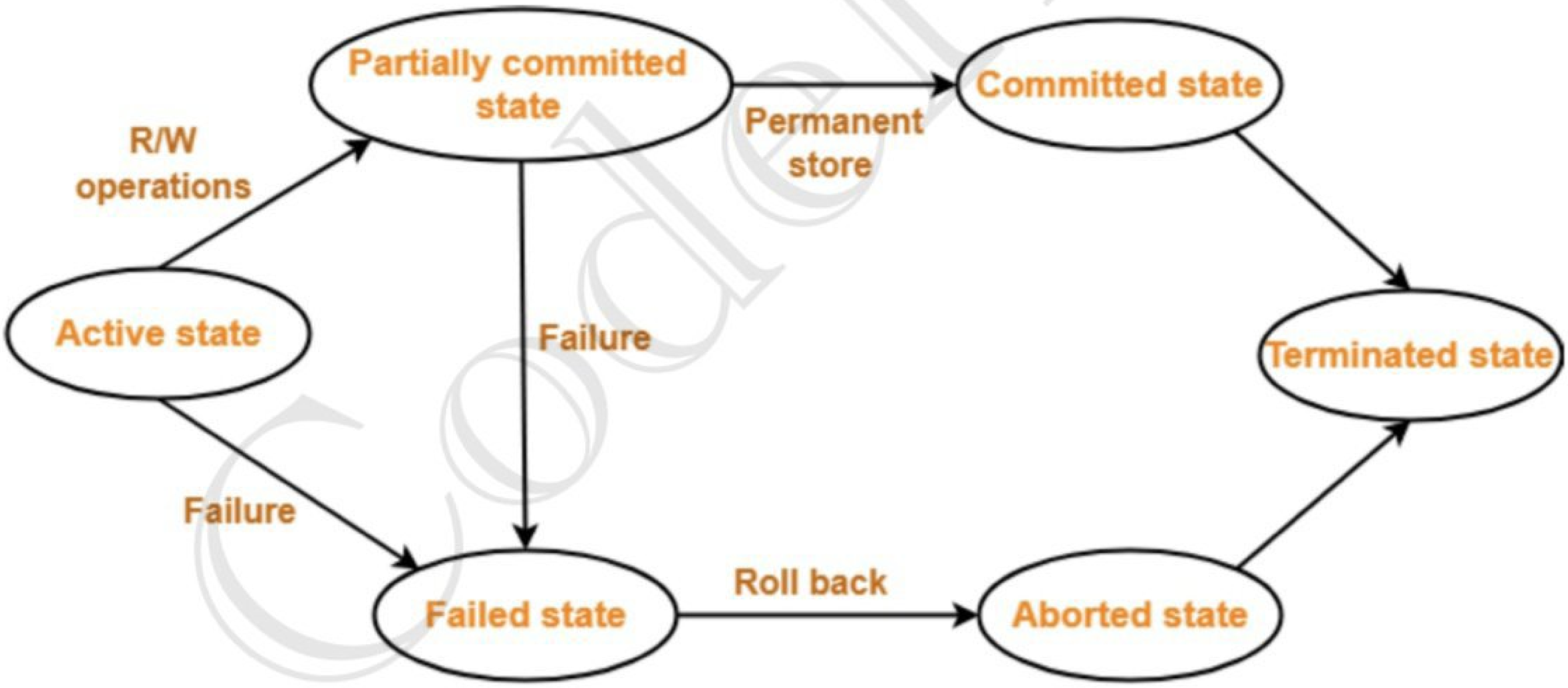
recovery: maintained by recovery management component.

... So, when the system restarts is recovery management role to commit those changes

→ using logs, ↑ etc.

utility ROM to save intermediate state, checkpoints.

3. **Transaction states**



Transaction States in DBMS

1. **Active state**
    1. The very first state of the life cycle of the transaction, all the read and write operations are being performed. If they execute without any error the T comes to Partially committed state. Although if any error occurs then it leads to a Failed state.
2. **Partially committed state**
    1. After transaction is executed the changes are saved in the buffer in the main memory. If the changes made are permanent on the DB then the state will transfer to the committed state and if there is any failure, the T will go to Failed state.
3. **Committed state**
    1. When updates are made permanent on the DB. Then the T is said to be in the committed state. Rollback can't be done from the committed states. New consistent state is achieved at this stage.
4. **Failed state**
    1. When T is being executed and some failure occurs. Due to this it is impossible to continue the execution of the T.
5. **Aborted state**
    1. When T reaches the failed state, all the changes made in the buffer are reversed. After that the T rollback completely. T reaches abort state after rollback. DB's state prior to the T is achieved.
6. **Terminated state**
    1. A transaction is said to have terminated if has either committed or aborted.

LEC-13: How to implement Atomicity and Durability in Transactions

Recovery Mechanism Component of DBMS supports **atomicity and durability.**

Shadow-copy scheme

1. Based on making copies of DB (aka, **shadow copies)**.
2. Assumption only one Transaction (T) is active at a time.
3. A pointer called **db-pointer** is maintained on the **disk**; which at any instant points to current copy of DB.
4. T, that wants to update DB first creates a complete copy of DB.
5. All further updates are done on new DB copy leaving the original copy (shadow copy) untouched.
6. If at any point the **T has to be aborted** the system deletes the new copy. And the old copy is not affected.
7. If T success, it is committed as,
    1. OS makes sure all the pages of the new copy of DB written on the disk.
    2. DB system updates the db-pointer to point to the new copy of DB.
    3. New copy is now the current copy of DB.
    4. The old copy is deleted.
    5. The T is said to have been **COMMITTED** at the point where the updated db-pointer is written to disk.
8. **Atomicity**
    1. If T fails at any time before db-pointer is updated, the old content of DB are not affected.
    2. T abort can be done by just deleting the new copy of DB.
    3. Hence, either all updates are reflected or none.
9. **Durability**
    1. Suppose, system fails are any time before the updated db-pointer is written to disk.
    2. When the system restarts, it will read db-pointer & will thus, see the original content of DB and none of the effects of T will be visible.
    3. T is assumed to be successful only when db-pointer is updated.
    4. If **system fails after** db-pointer has been updated. Before that all the pages of the new copy were written to disk. Hence, when system restarts, it will read new DB copy.
10. The implementation is dependent on write to the db-pointer being atomic. Luckily, disk system provide atomic updates to entire block or at least a disk sector. So, we make sure db-pointer lies entirely in a single sector. By storing db-pointer at the beginning of a block.
11. **Inefficient**, as entire DB is copied for every Transaction.

Log-based recovery methods

1. The log is a sequence of records. Log of each transaction is maintained in some **stable storage** so that if any failure occurs, then it can be recovered from there.
2. If any operation is performed on the database, then it will be recorded in the log.
3. But the process of storing the logs should be done **before** the actual transaction is applied in the database.
4. **Stable storage** is a classification of computer data storage technology that guarantees atomicity for any given write operation and allows software to be written that is robust against some hardware and power failures.
5. **Deferred DB Modifications**
    1. Ensuring **atomicity** by recording all the DB modifications in the log but deferring the execution of all the write operations until the final action of the T has been executed.
    2. Log information is used to execute deferred writes when T is completed.
    3. If system crashed before the T completes, or if T is aborted, the information in the logs are ignored.
    4. If T completes, the records associated to it in the log file are used in executing the deferred writes.
    5. If failure occur while this updating is taking place, we preform redo.
6. **Immediate DB Modifications**
    1. DB modifications to be output to the DB while the T is still in active state.
    2. DB modifications written by active T are called uncommitted modifications.
    3. In the event of crash or T failure, system uses old value field of the log records to restore modified values.
    4. Update takes place only after log records in a stable storage.
    5. Failure handling
        1. System failure before T completes, or if T aborted, then old value field is used to undo the T.
        2. If T completes and system crashes, then new value field is used to redo T having commit logs in the logs.

① Deferred DB Modifications

---

② Immediate DB modifications

<T₀ Start>

<T₀, A, 1000, 950> → old item value, new item value.

<T₀, B, 2000, 2050>

<T₀ Commit>

Lec-14

SQL = Select Where.

DBMS

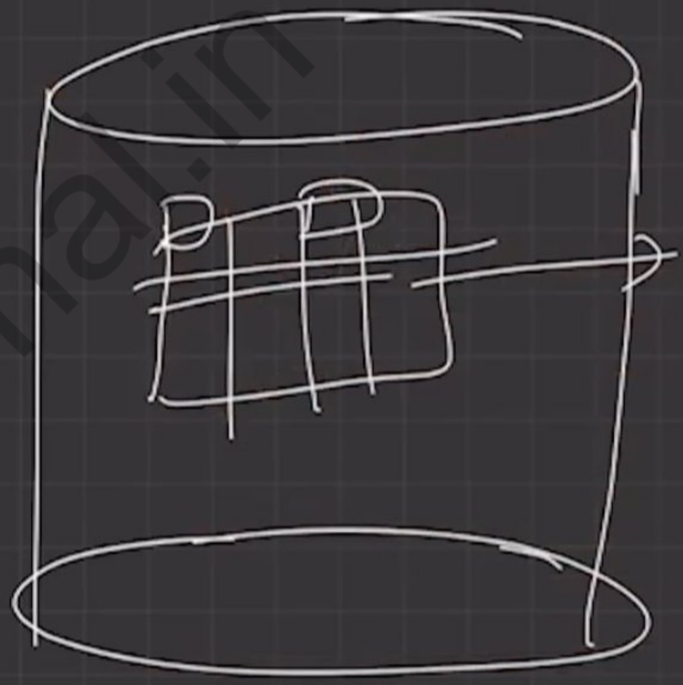


Table
Student

Disk

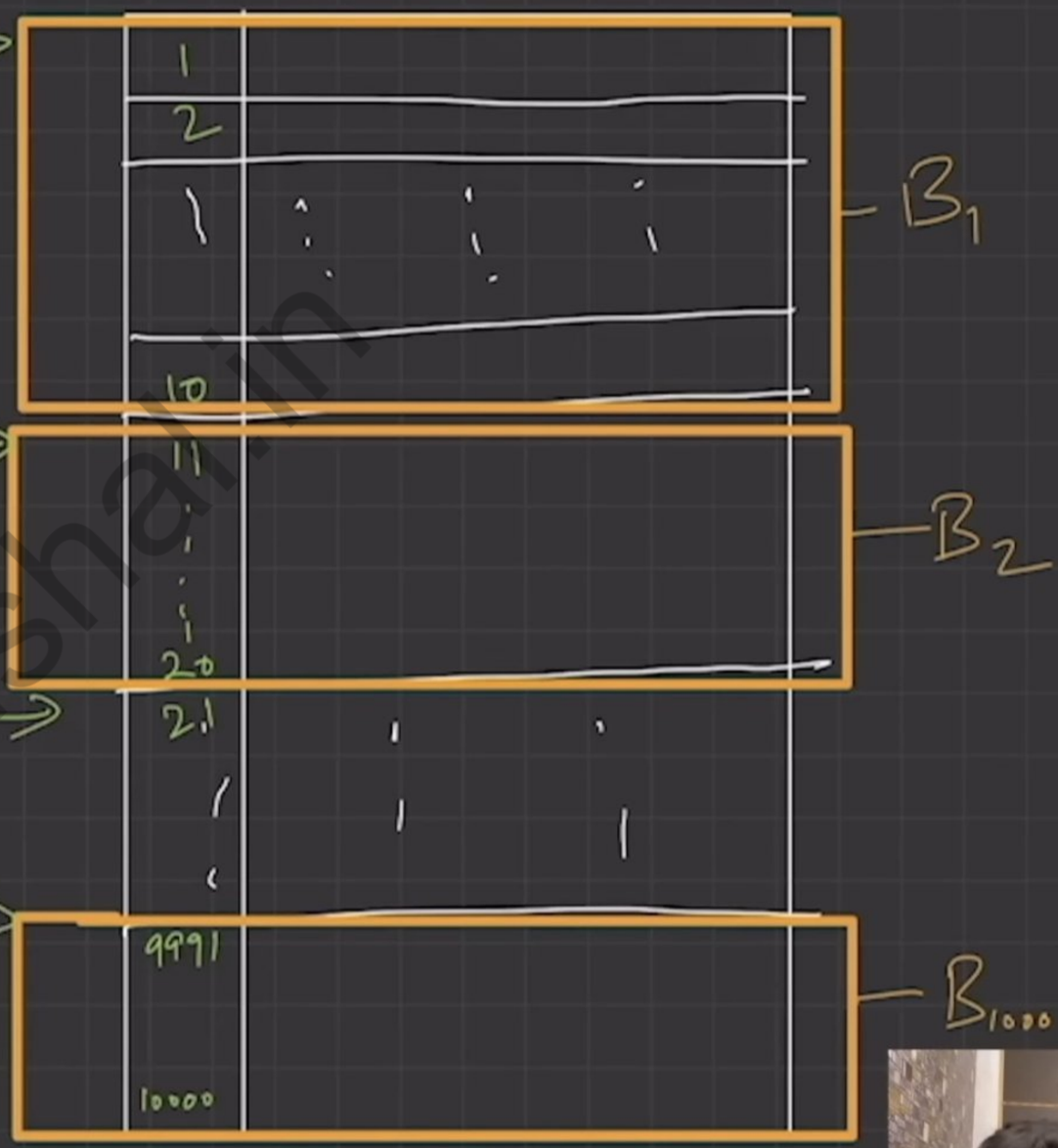
Data structure

INDEX

Data file in Disk. (10,000 records)

INDEX file.

Rollno.	BP
1	B_1
11	B_2
21	B_3
⋮	⋮
9991	B_{1000}



each Block has 10 records → Total Bl

Types of Indexing

① Primary index

① Based on key attribute (ordering datafile based on key attribute)

→ P.K / C.K

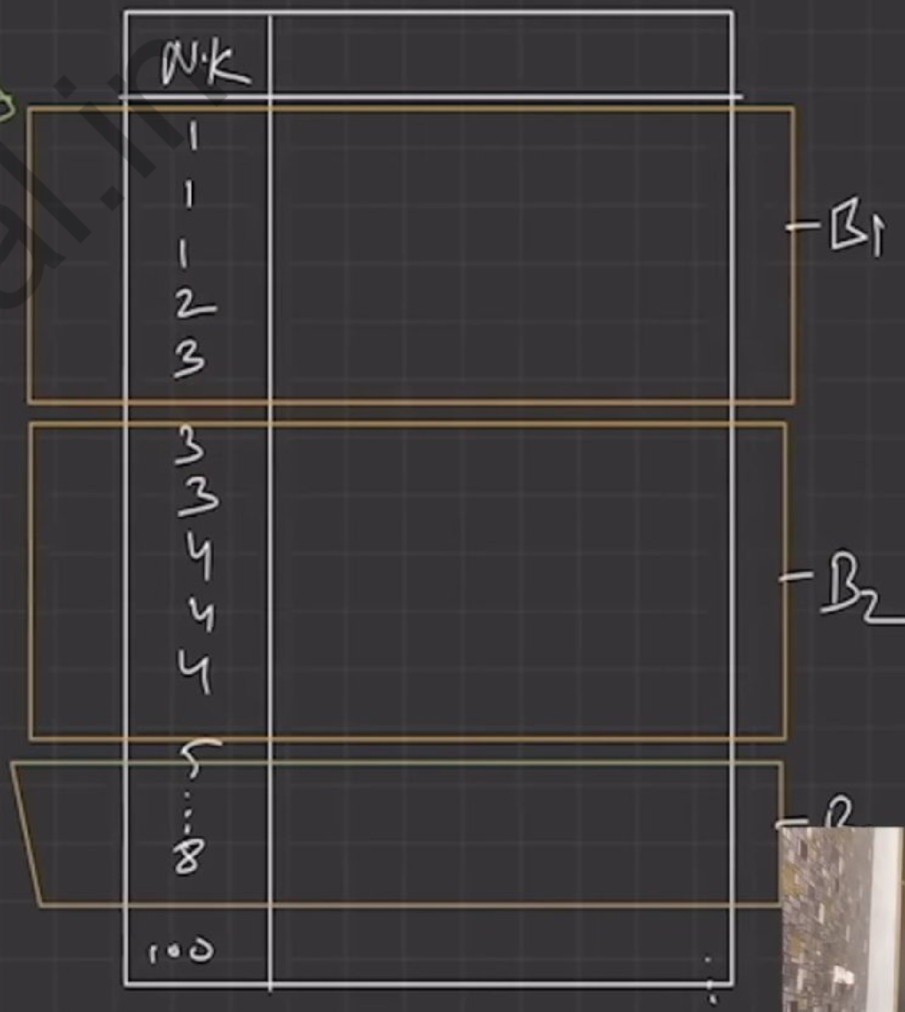
↳ indexing → Sparse indexing

(2) Based on non-key attribute

→ **Clustering Indexing**

2) INDEX

N.K	BP
1	B_1
2	B_1
3	B_1
4	B_2
5	B_3
:	:



② Based on non-key attribute

→ **Clustering Indexing**

2) INDEX

NK	BP
1	B_1
2	B_1
3	
4	
5	

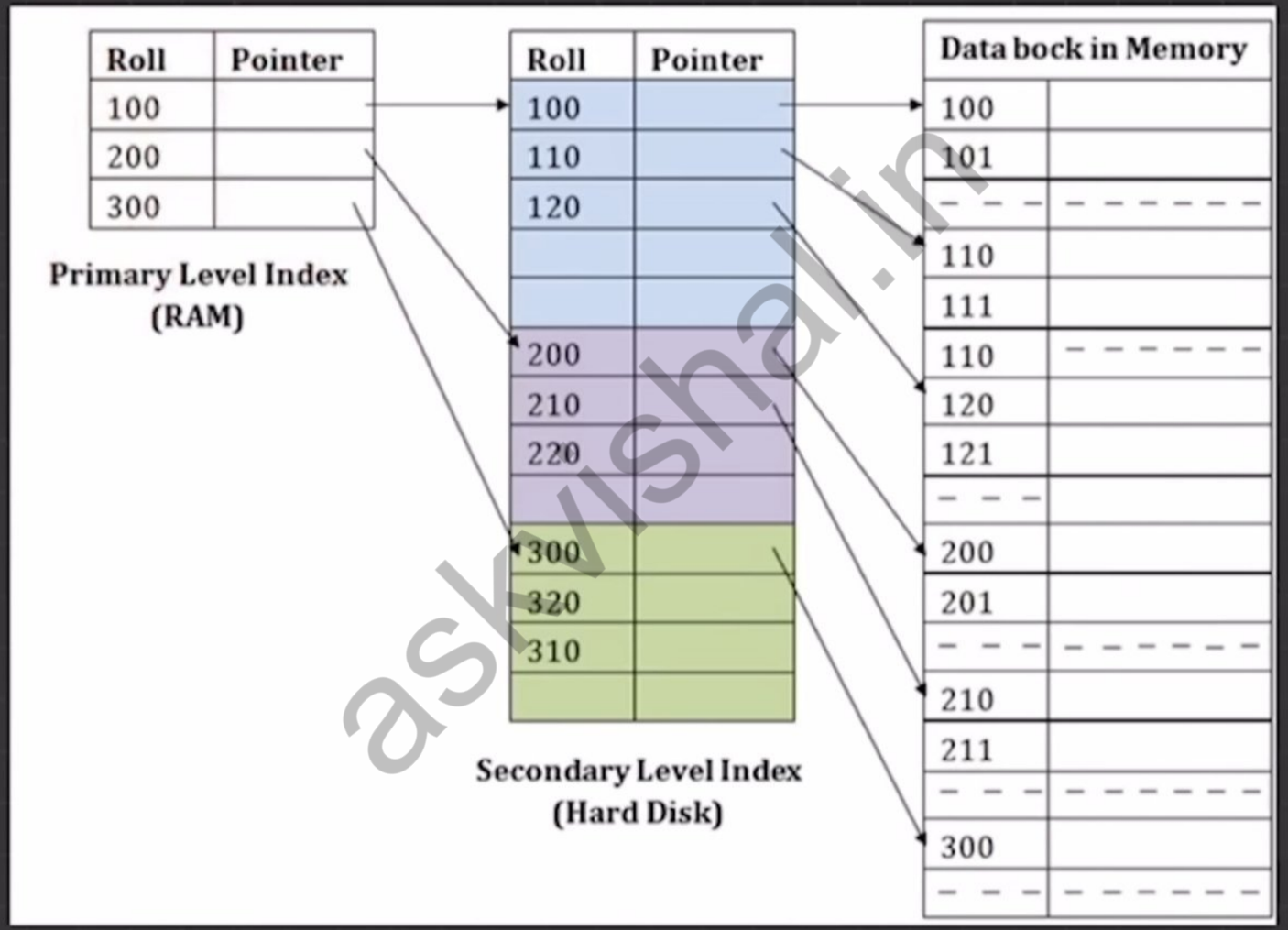
NK
1
1
2
3
3
4
4
5
8
100

B_1

B_2

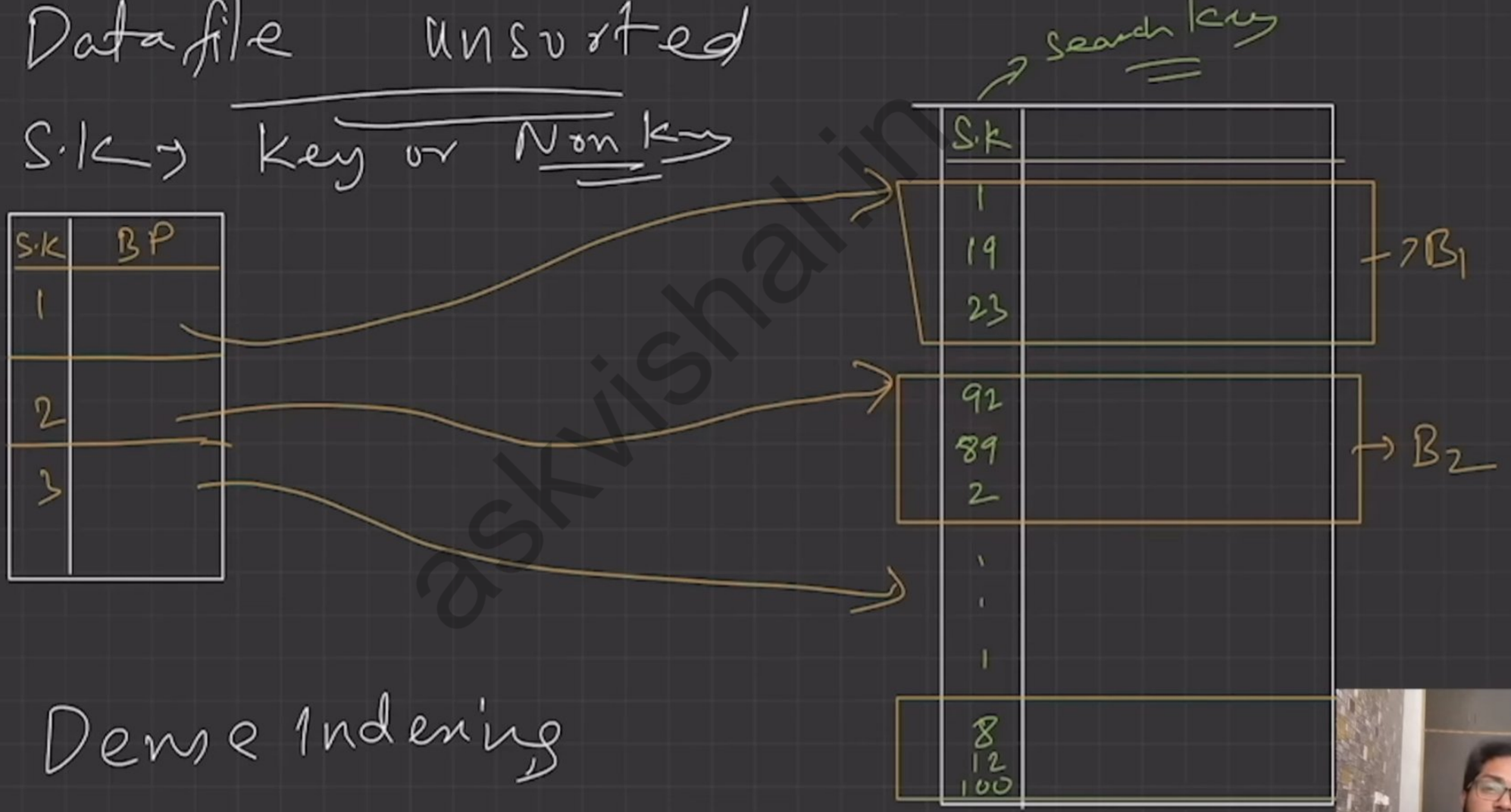
B_3

Multi Level INDEXING

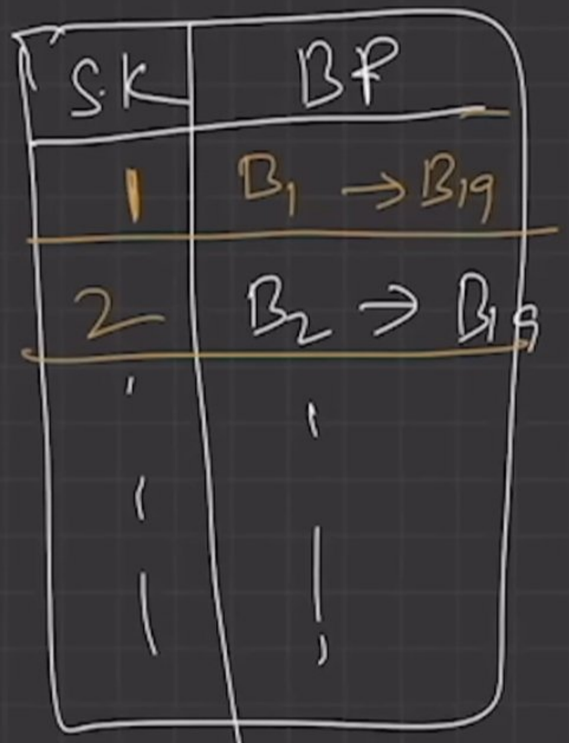


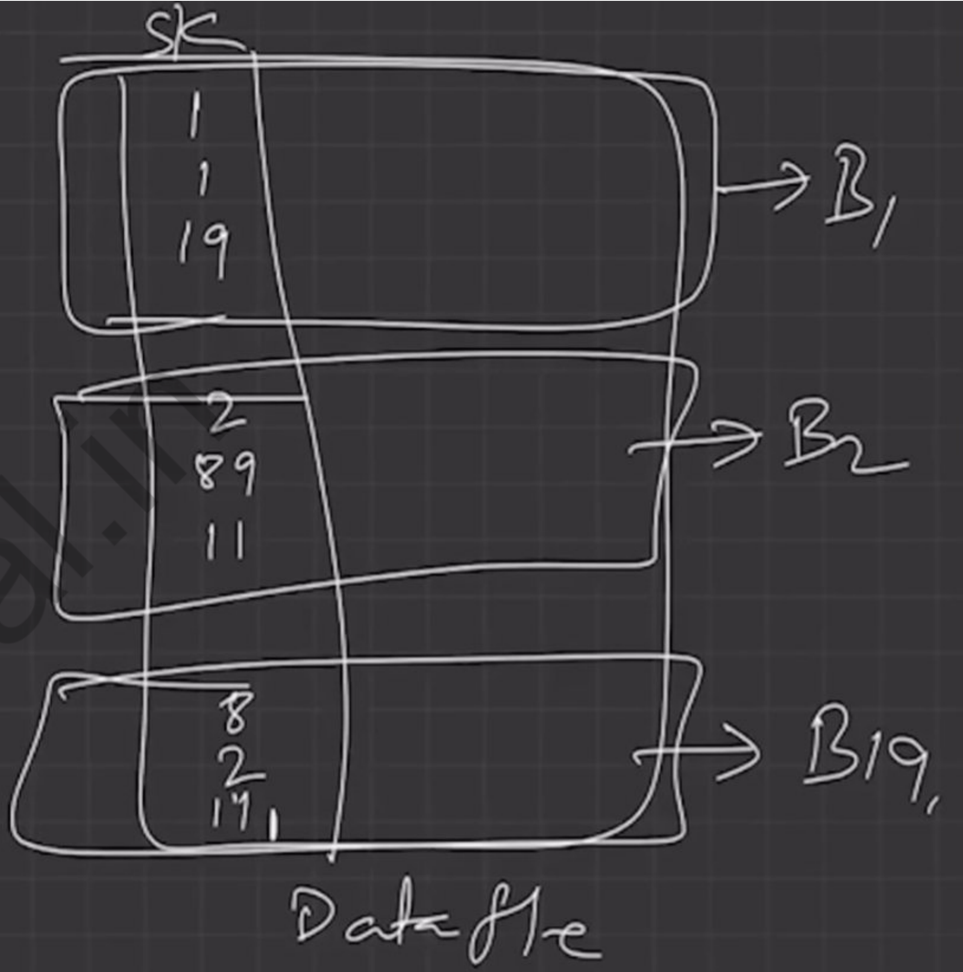
Secondary Indexing

- Datafile unsorted
- S.key key or Non key



- Dense Indexing





⇒ Data Modeling in SQL vs NoSQL

① SQL

users

ID	first_name	last_name	cell	city
1	Tata	Salt	811	Mumbai

hobbies

ID	user_id	hobby
10	1	scrapbooking
11	1	Games,
12	1	Biking

⇒ Data Modeling in SQL vs NoSQL

① SQL

mers

ID	first_name	last_name	cell	city
1	Tata	Salt	811	Mumbai

Hobbies

ID	mer_id	hobby
10	1	scrapbooking
11	1	Games
12	1	Biking

⇒ Data Modeling in SQL vs NoSQL

① SQL

users

ID	first_name	last_name	cell	city
1	Tata	Salt	811	Mumbai

hobbies

ID	user_id	hobby
10	1	scrapbooking
11	1	Games.
12	1	Biking

② NoSQL.

```
{
  "id": 1,
  "first_name": "Tata",
  "last_name": "Salt",
  "cell": "811",
  "city": "Mumbai",
  "hobbies": ["scrapbooking", "Games", "Biking"]
}
```

ACID vs BASE – Comparison Table

Feature	ACID	BASE
Full Form	Atomicity, Consistency, Isolation, Durability	Basically Available, Soft State, Eventual Consistency
Database Type	Relational (RDBMS)	NoSQL
Consistency Model	Strong consistency	Eventual consistency
Availability	Lower compared to BASE	High availability
Transaction Support	Full transaction support	Limited / relaxed transactions
Scalability	Vertical scaling (hard)	Horizontal scaling (easy)
Schema	Fixed schema	Flexible / schema-less
Failure Handling	Rolls back on failure	System remains available
Performance	Slower due to strict rules	Faster due to relaxed rules
Best Use Cases	Banking, finance, critical systems	Big data, social media, real-time analytics
Examples	MySQL, PostgreSQL, Oracle	MongoDB, Cassandra, DynamoDB

One-line memory tip:

ACID = Correctness first | BASE = Availability first