DBMS

Course Structure

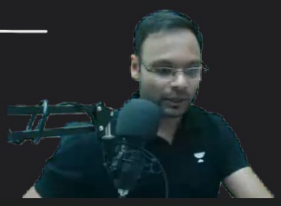
→ 2 hours lectures → daily revision → dpp

→ every 4th class → doubt class & dpp } next lecture

Every - weekend ⇒ Revision

Course finish ⇒ detailed Revision

Topics fixed ⇒ PYQs



unacademy

Course Structure

Topics
Basics ✓
DBMS Designing ✓
E-R Modelling ✓
Relational Database Design ✓
SQL ✓ → 1 Questn
Relational Algebra ✓
Transaction & Concurrency ✓
Indexing ✓

unacademy

Course Structure

Topics
Basics
DBMS Designing
E-R Modelling

PYQ

- Part- 1 -> https://unacademy.com/class/sql-pyq-discussion-part-1/FIVDGXH3
- Part- 2 -> https://unacademy.com/class/sql-pyq-discussion-part-2/USA9WKOY
- Part- 3 -> https://unacademy.com/class/sql-pyq-discussion-part-3/9IHKNZVM
- Part- 4 -> https://unacademy.com/class/sql-pyq-discussion-part-4/EAQ4FZLP
- Part- 5 -> https://unacademy.com/class/sql-rdbms-pyq-discussion/HX1N8R7P
- Part- 6 -> https://unacademy.com/class/rdbms-pyq-discussion-part-3/NA5SHPWS
- Part- 7 -> https://unacademy.com/class/relational-algebra-pyq-discussion-part-1/AFHZQ2C9

SQL Link

- Link -> https://www.w3schools.com/sql/trysql.asp?filename=trysql_asc
- Better one -> https://sqliteonline.com/

Course Link

- Link -> https://unacademy.com/course/complete-course-on-database-management-system-247/OBHXIW40

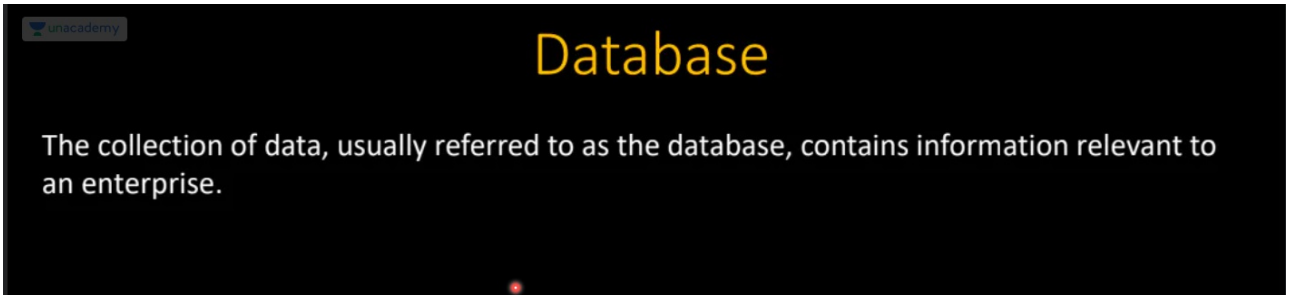
Introduction to dbms (1) [23rd June 2023]

Data

- Anything which is special we want to remember.
- Data is needed.

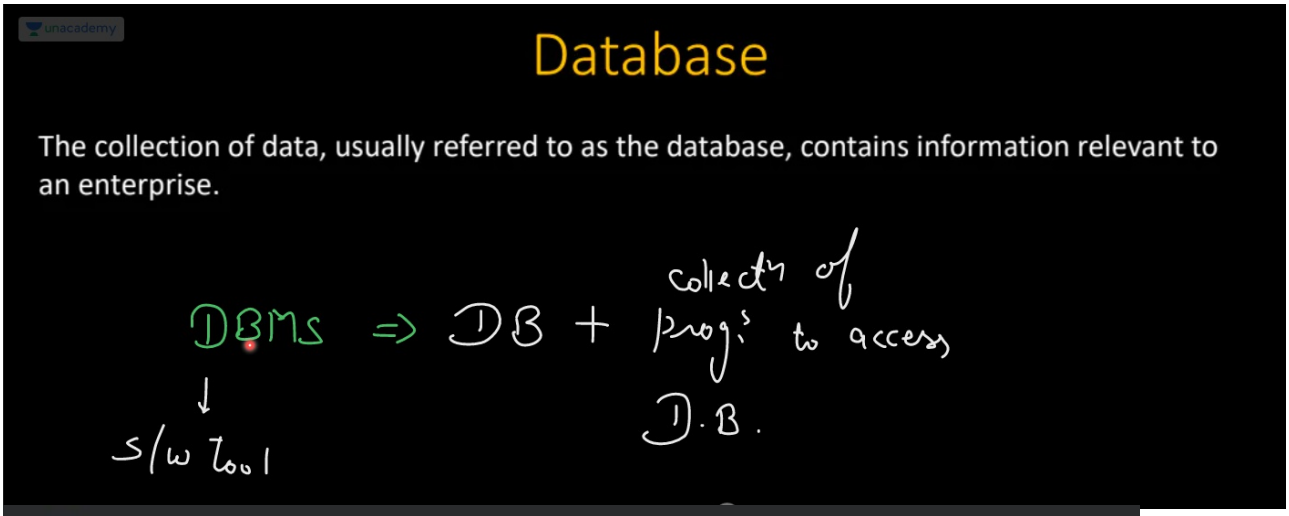
Database

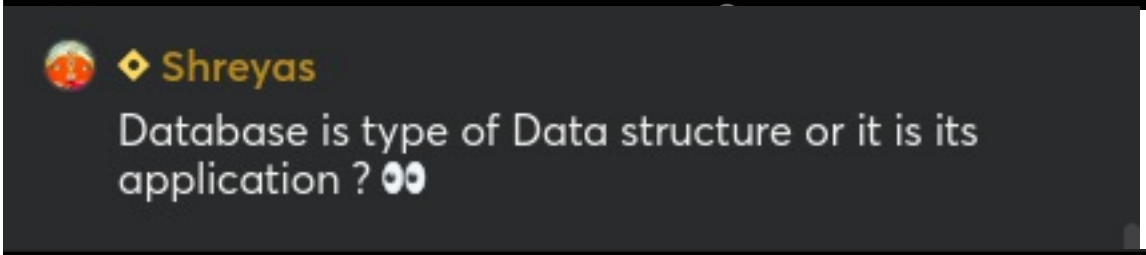
> The **collection of data**, usually referred to as the database, contains information relevant to an enterprise.

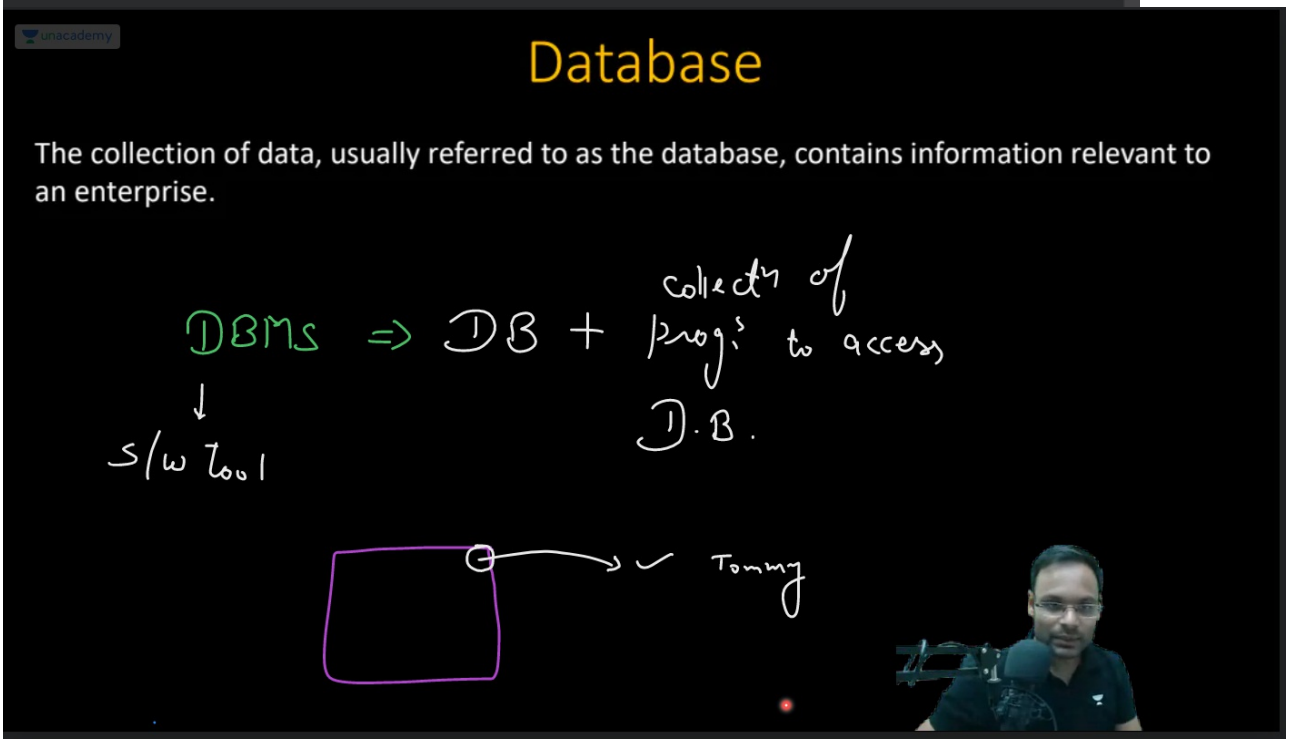


DBMS

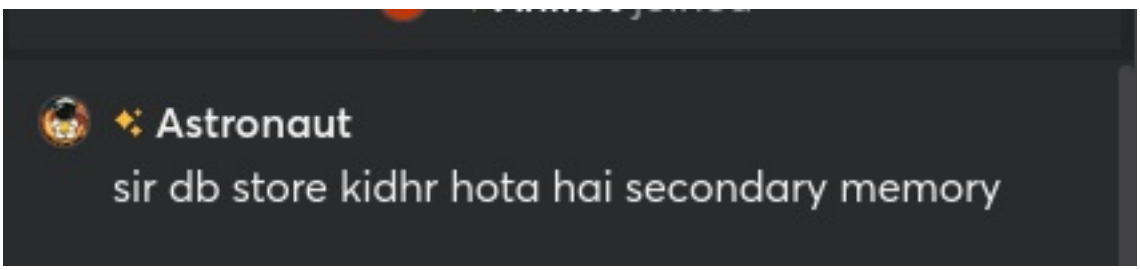
- Software tool
- Database + collection of programs to access Database.



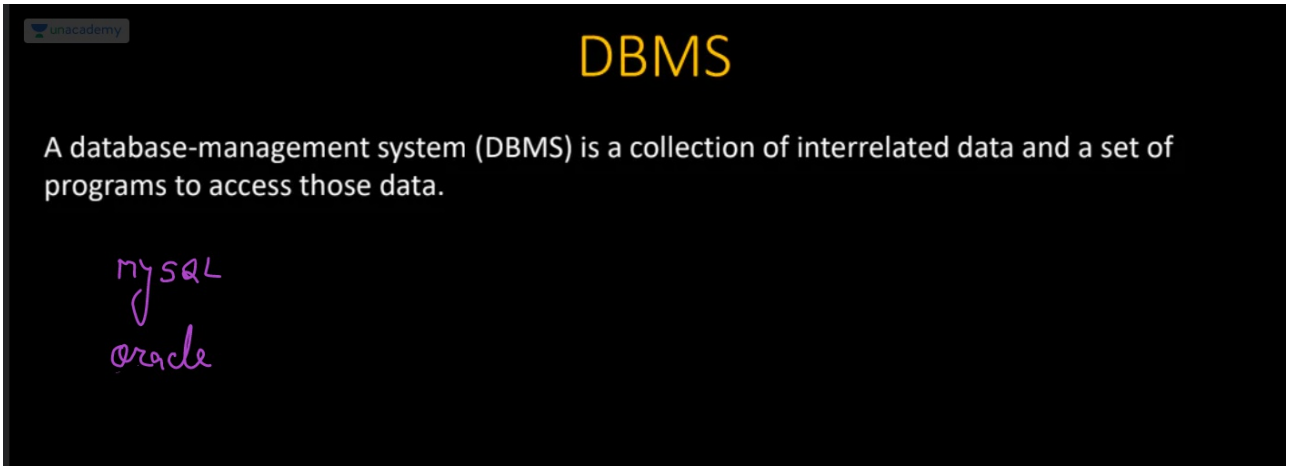




- If there is a **database** then the data is **very well organized**.



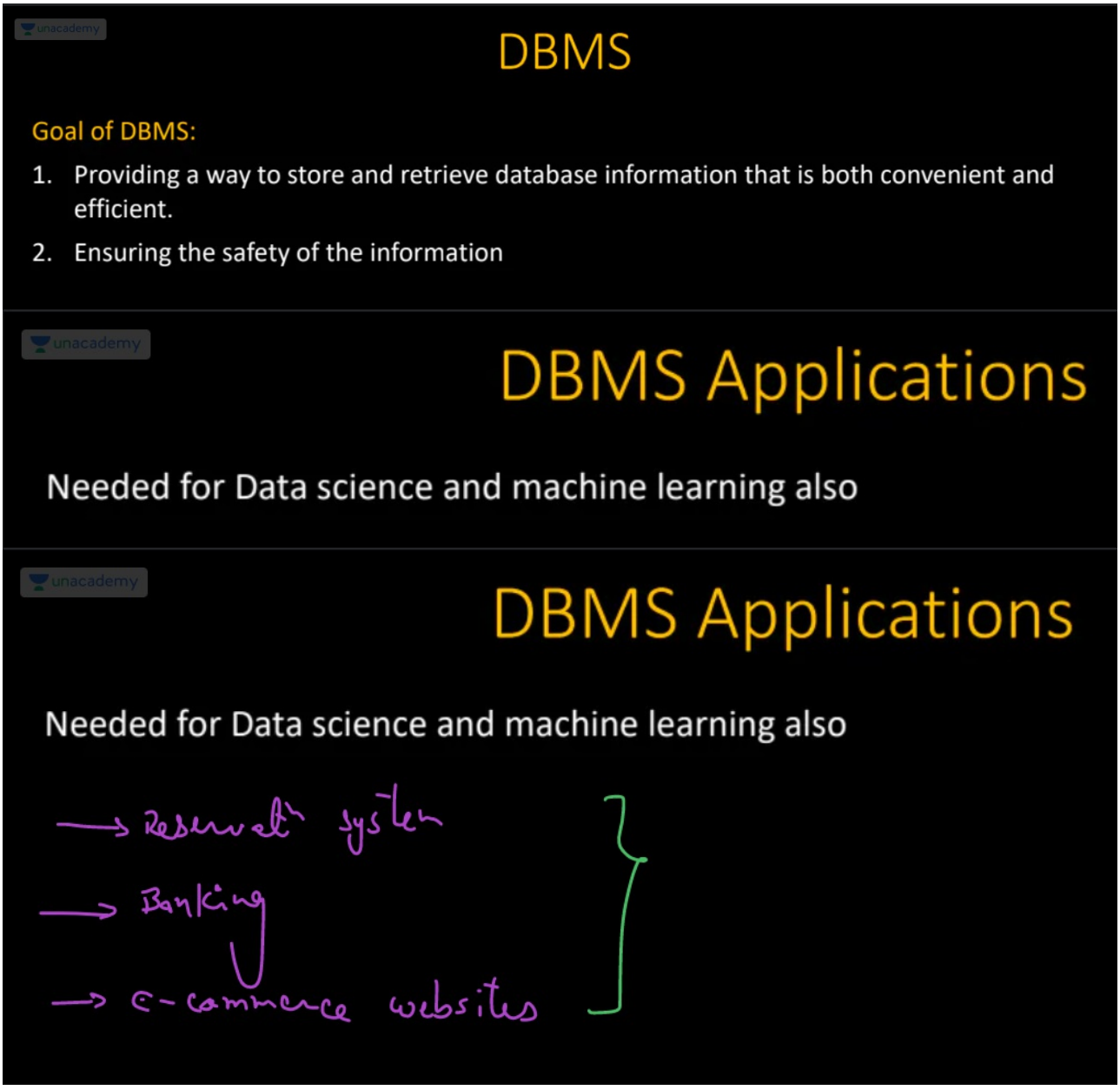
- YES.
- We don't want the database to go away after we shut down the computer.



> A database-management system(DBMS) is a collection of interrelated data and a set of programs to access those data.

Gola of DBMS

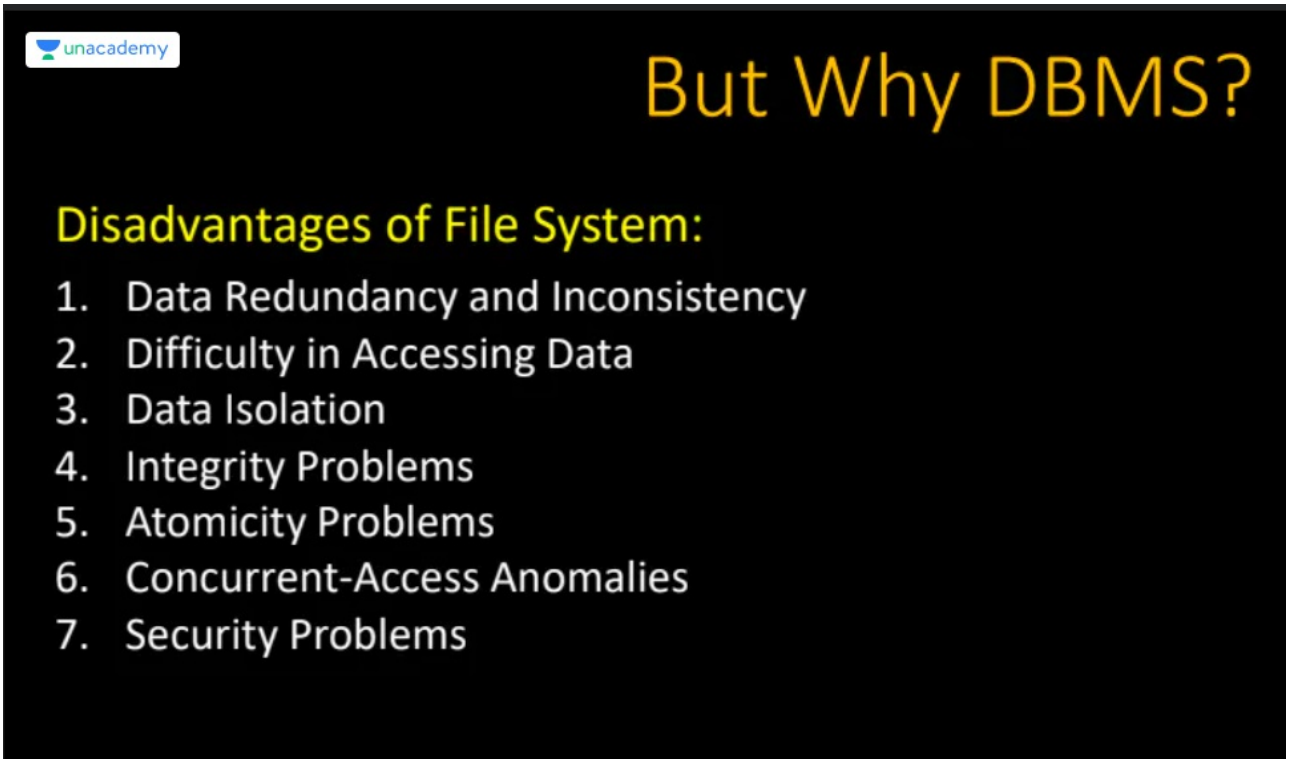
1. Providing a way to store and retrieve database information that is both convenient and efficient
2. Ensuring the safety of the information.



- Reservation System
- Banking
- E-commerce websites

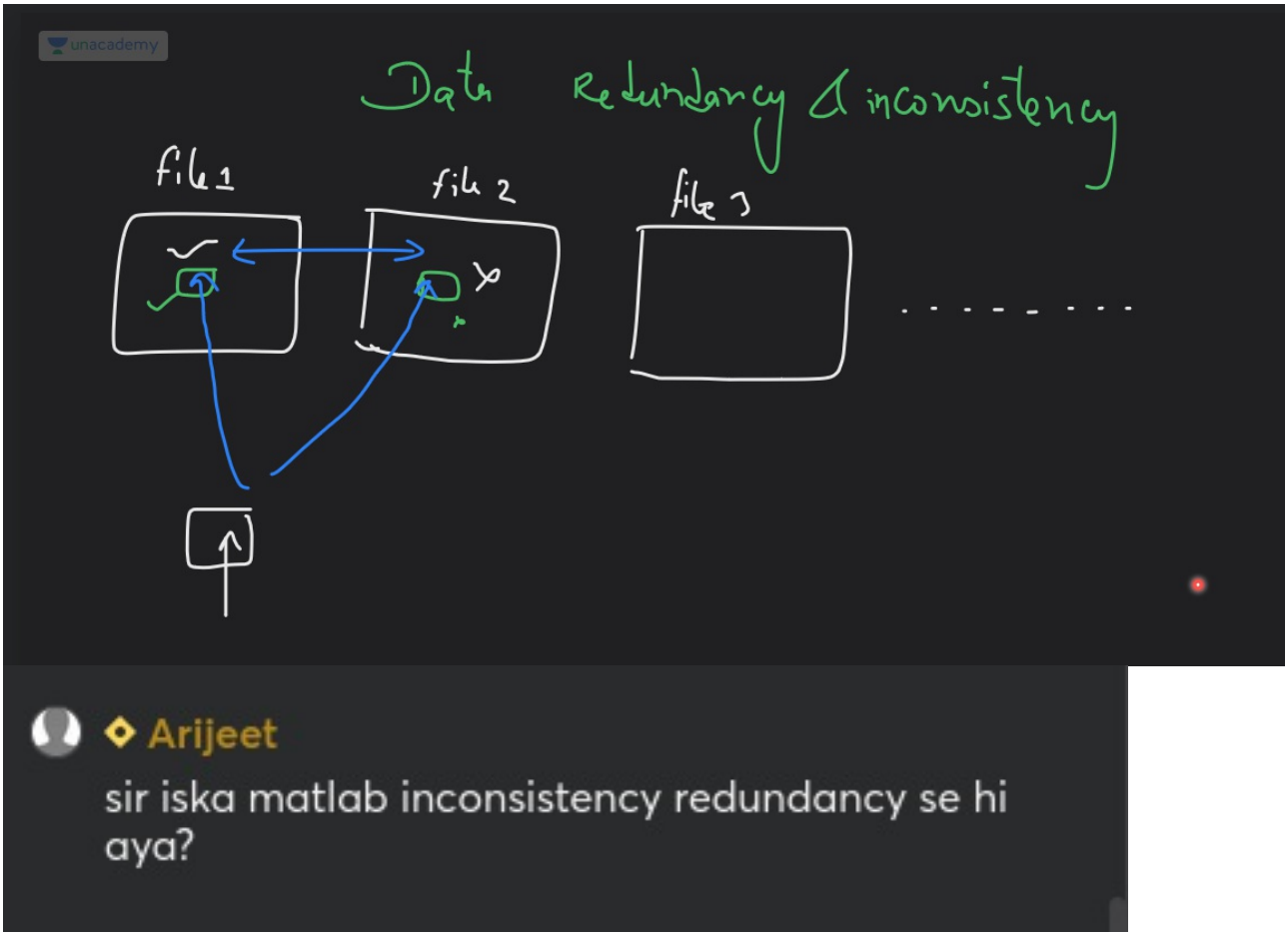
Why DBMS?

- Before DBMS, data were stored in files.



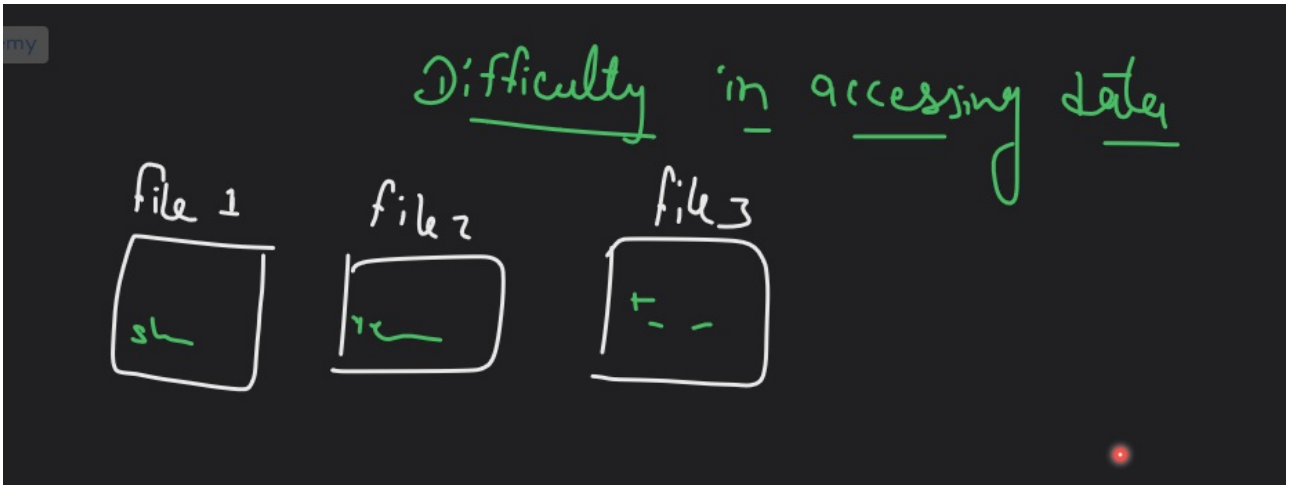
- Disadvantage of file system:-

1. Data Redundacy and consistency -> No linking between the same data on two files.

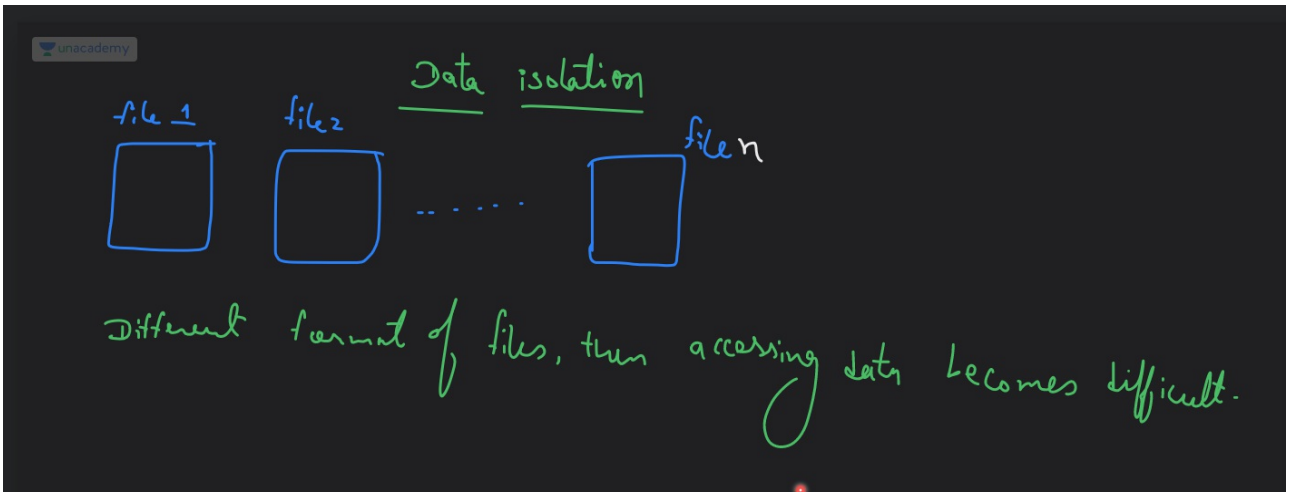


- Yes.

2. Difficulty in accessing data

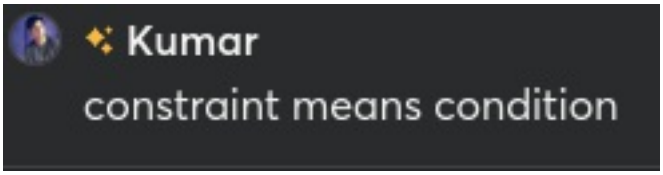


3. Data Isolation

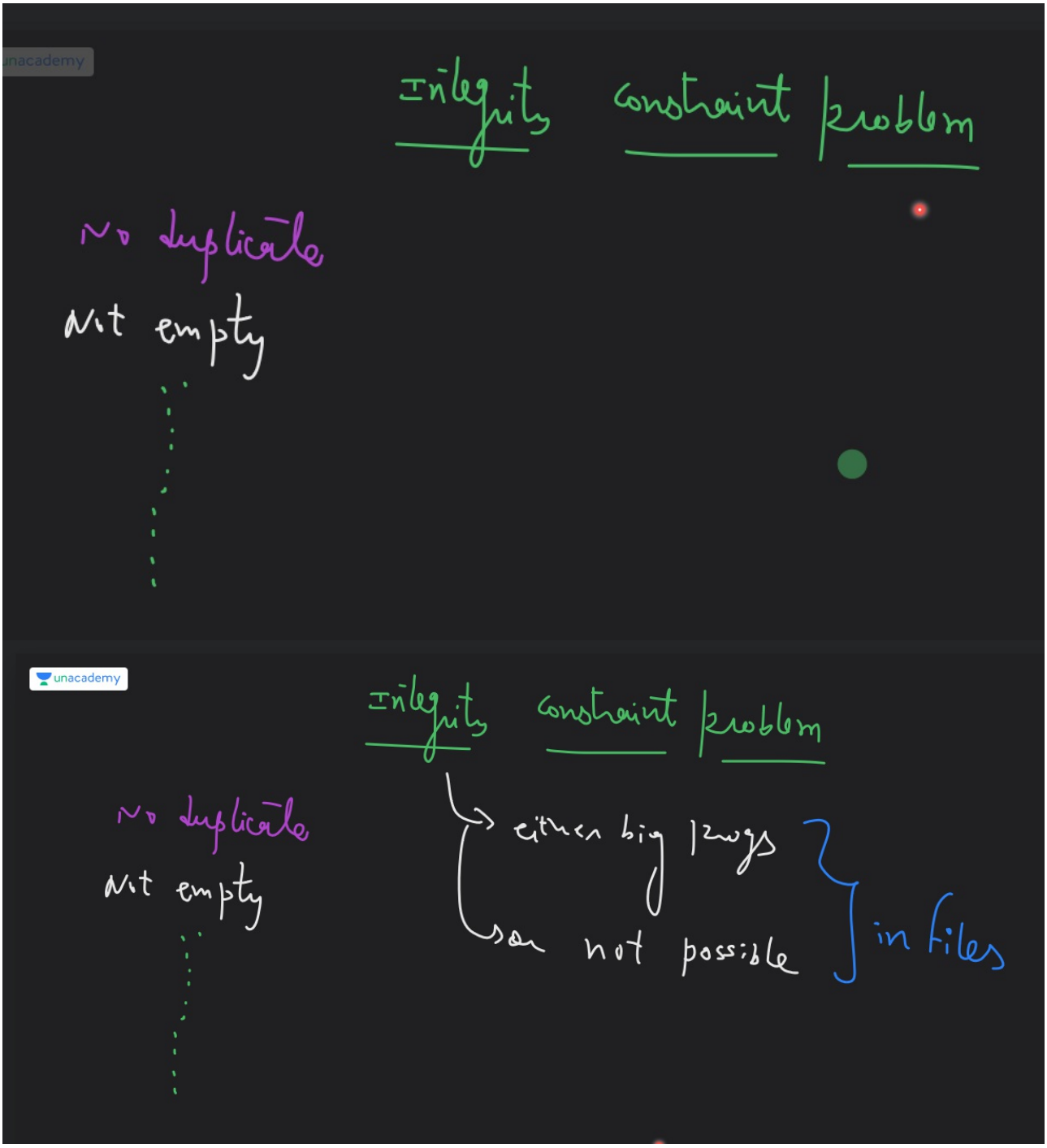


- Different formats of files are there and then accessing the data from the different files becomes difficult.

4. Integrity problem

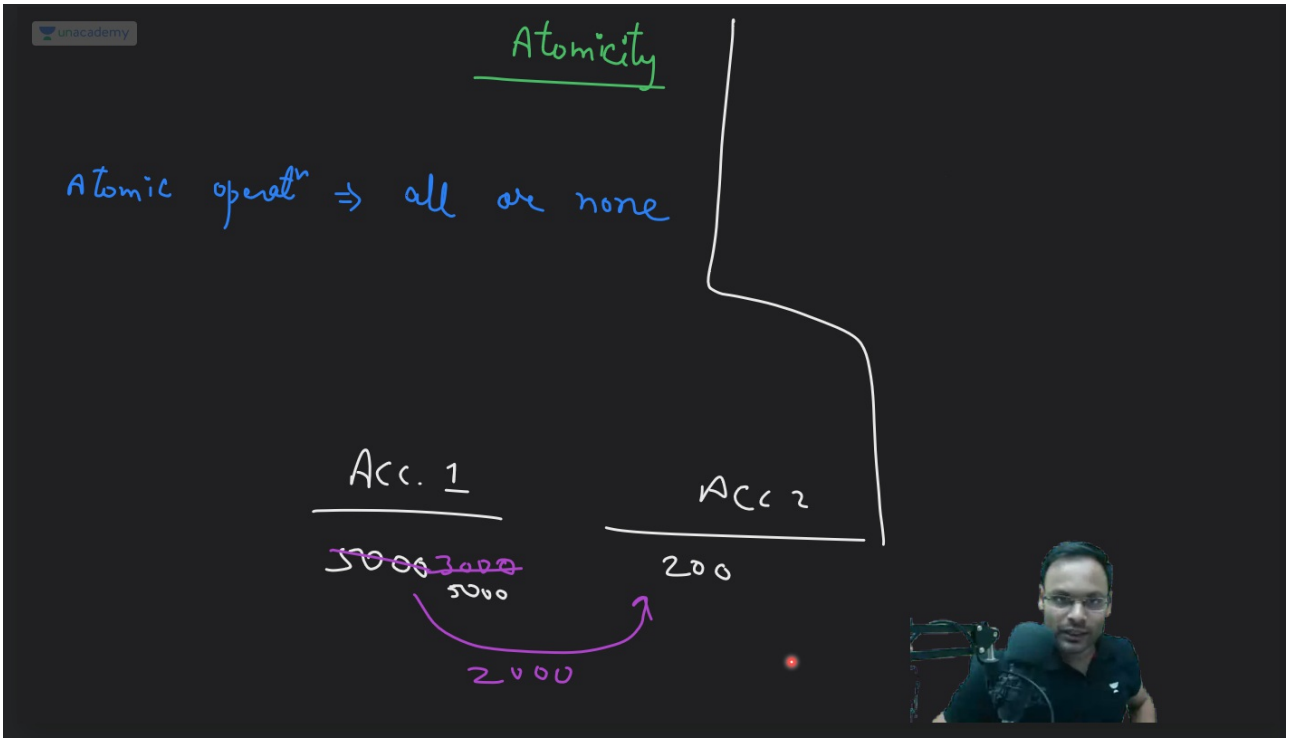


- Yes.



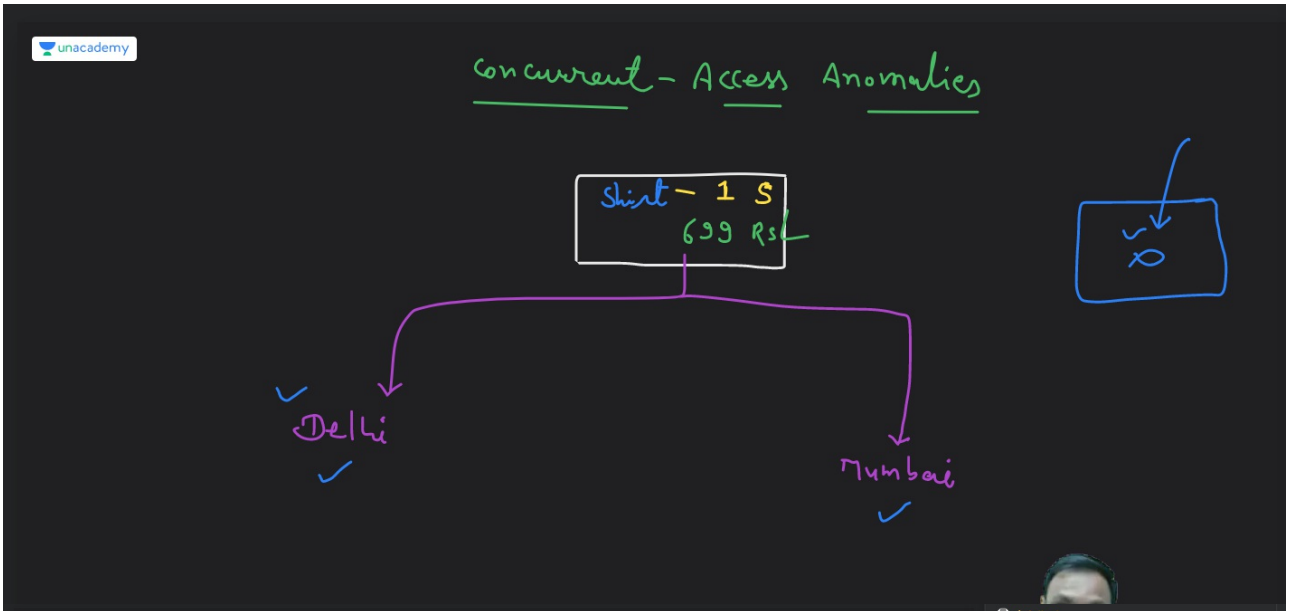
5. Atomicity Problem

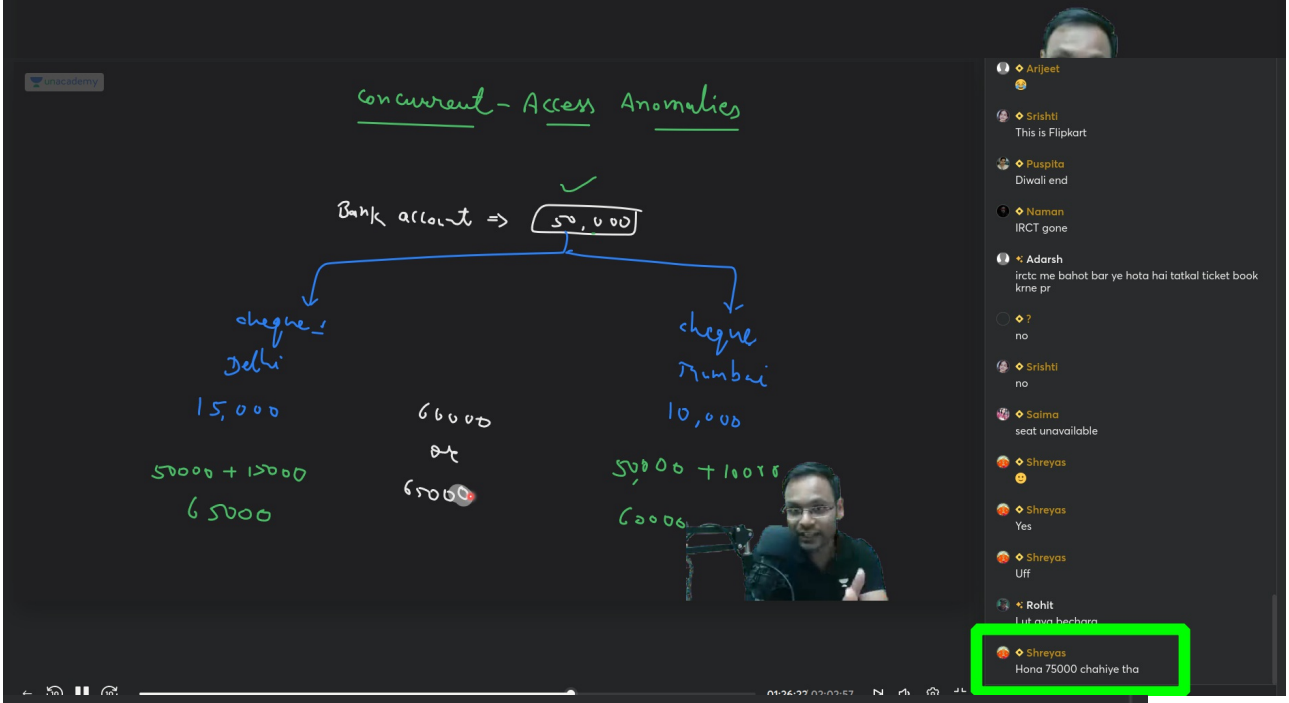
- **Atomic operation** -> All or none

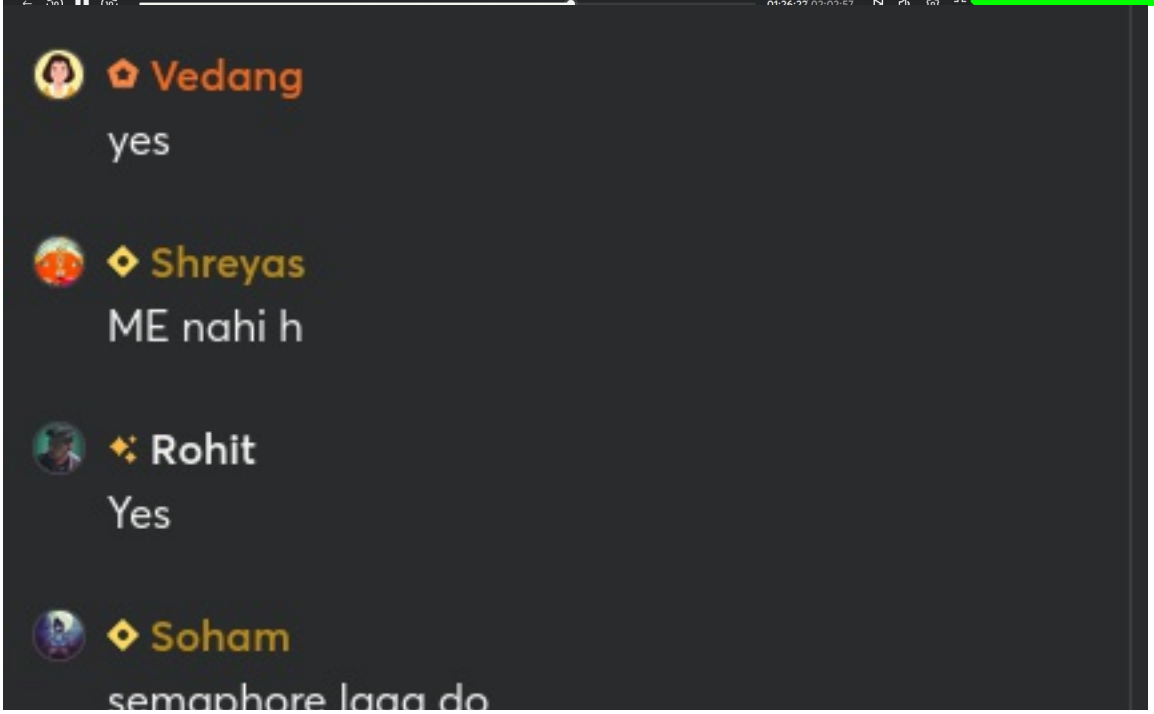


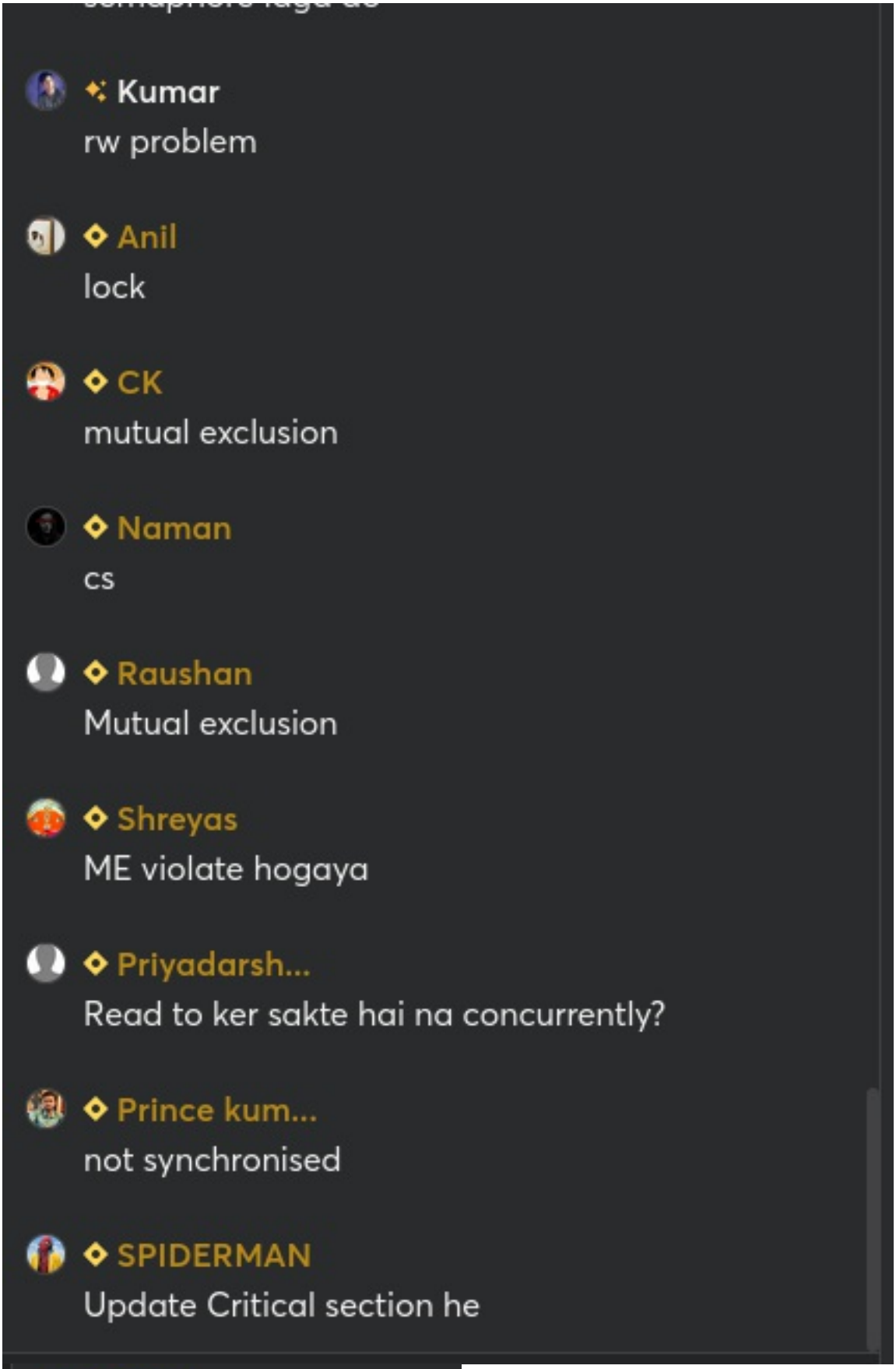
6. Cocurrent-Access Anomalies

- Anomalies -> Problem.


unacademy
Concurrent - Access Anomalies
Shirt - 1 S
699 Rs
Delhi
Mumbai


unacademy
Concurrent - Access Anomalies
Bank account => 50,000
cheque - Delhi
15,000
50000 + 15000
65000
66000
or
65000
cheque Mumbai
10,000
50000 + 10000
60000
Arijeet
Srishti
This is Flipkart
Puspita
Diwali end
Naman
IRCT gone
Adarsh
irctc me bahot bar ye hota hai tatkal ticket book krne pr
?
no
Srishti
no
Saima
seat unavailable
Shreyas
Shreyas
Yes
Shreyas
Uff
Rohit
Shreyas
Hona 75000 chahiye tha

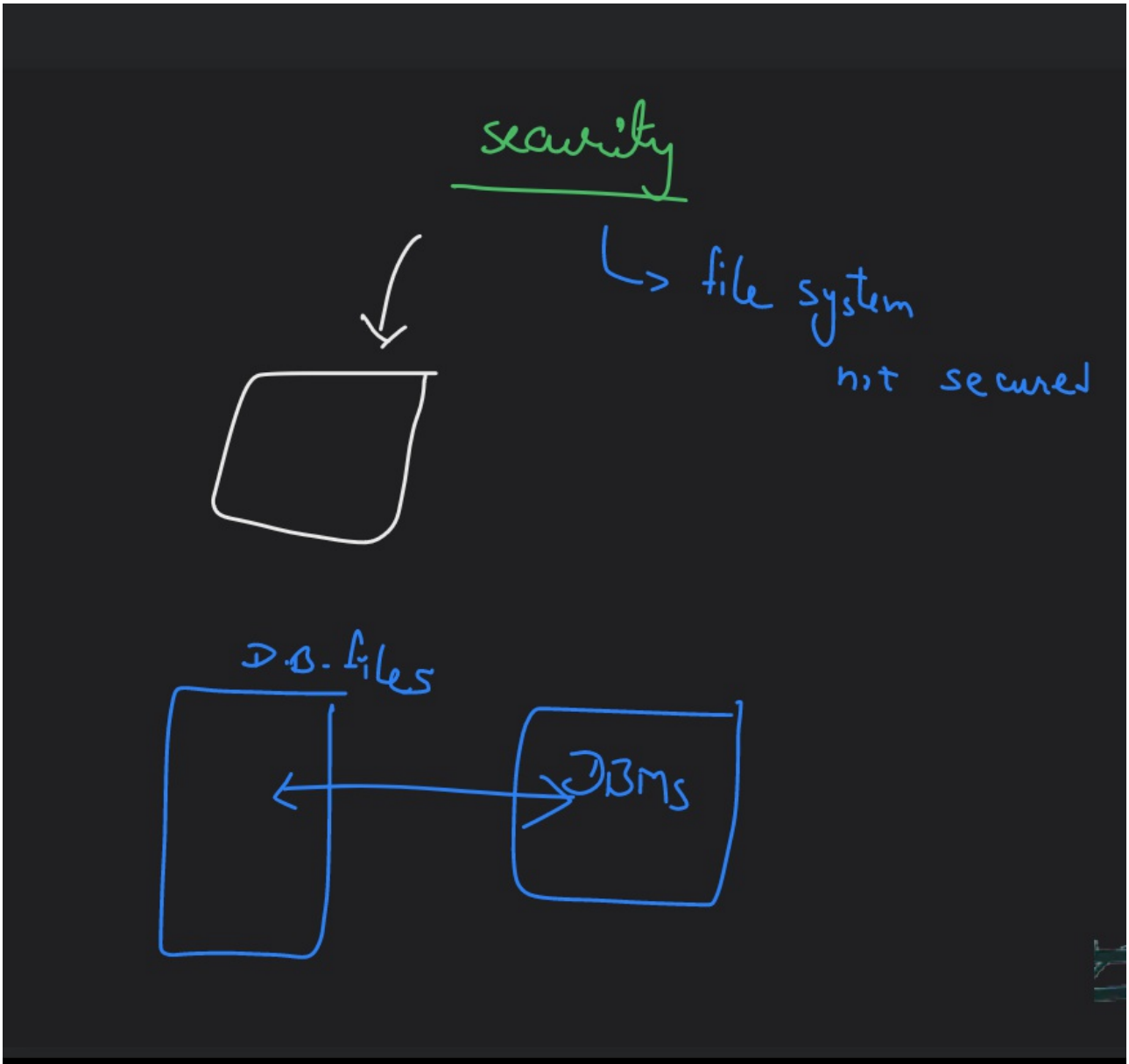

Vedang
yes
Shreyas
ME nahi h
Rohit
Yes
Soham
semaphore laga do


Kumar
rw problem
Anil
lock
CK
mutual exclusion
Naman
cs
Raushan
Mutual exclusion
Shreyas
ME violate hogaya
Priyadarsh...
Read to ker sakte hai na concurrently?
Prince kum...
not synchronised
SPIDERMAN
Update Critical section he
CK
DBMS mai ME hota h

- Yes. Mutual Exclusion(ME) is there.

7. Security Problem

- In Files system -> Files are not secured.



unacademy

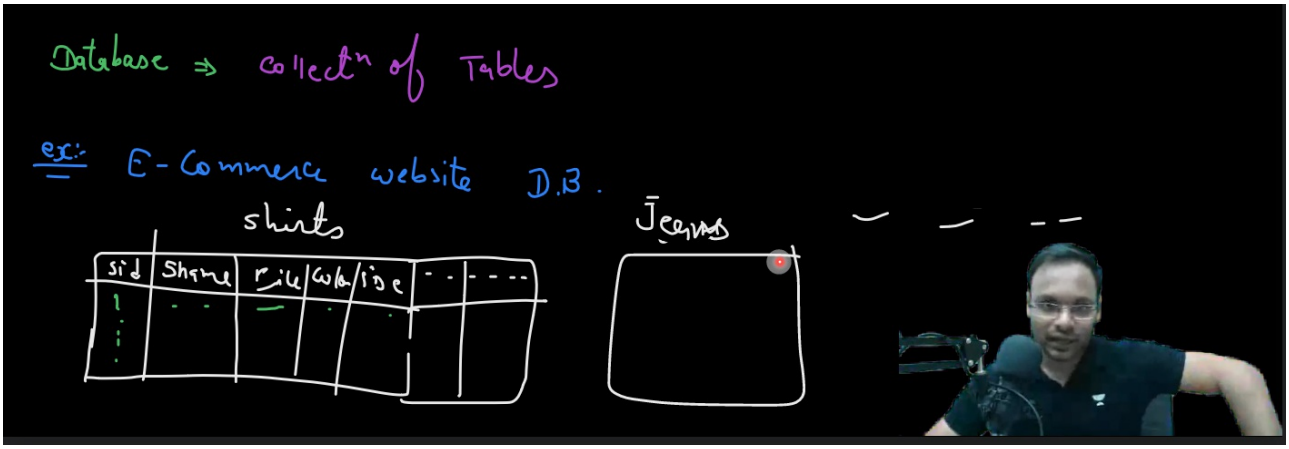
But Why DBMS?

Disadvantages of File System:

1. Data Redundancy and Inconsistency
2. Difficulty in Accessing Data
3. Data Isolation
4. Integrity Problems
5. Atomicity Problems
6. Concurrent-Access Anomalies
7. Security Problems

How Database or Data is stored?

- Database -> Collection of Tables.



- Logical Level

View of Data

1. **Physical Level** -> When the DBMS will store the database, where ever the database is stored is called as the **physical level**. The level where the database is actually stored in the disk.
2. **Logical Level** -> All of the tables, the details of the tables, the details of the items stored in the tables. Everything in front of my eyes.
3. **View Level** -> The way it is filtered to be seen by the users on the website. Seen by the users. Part of **logical level**.

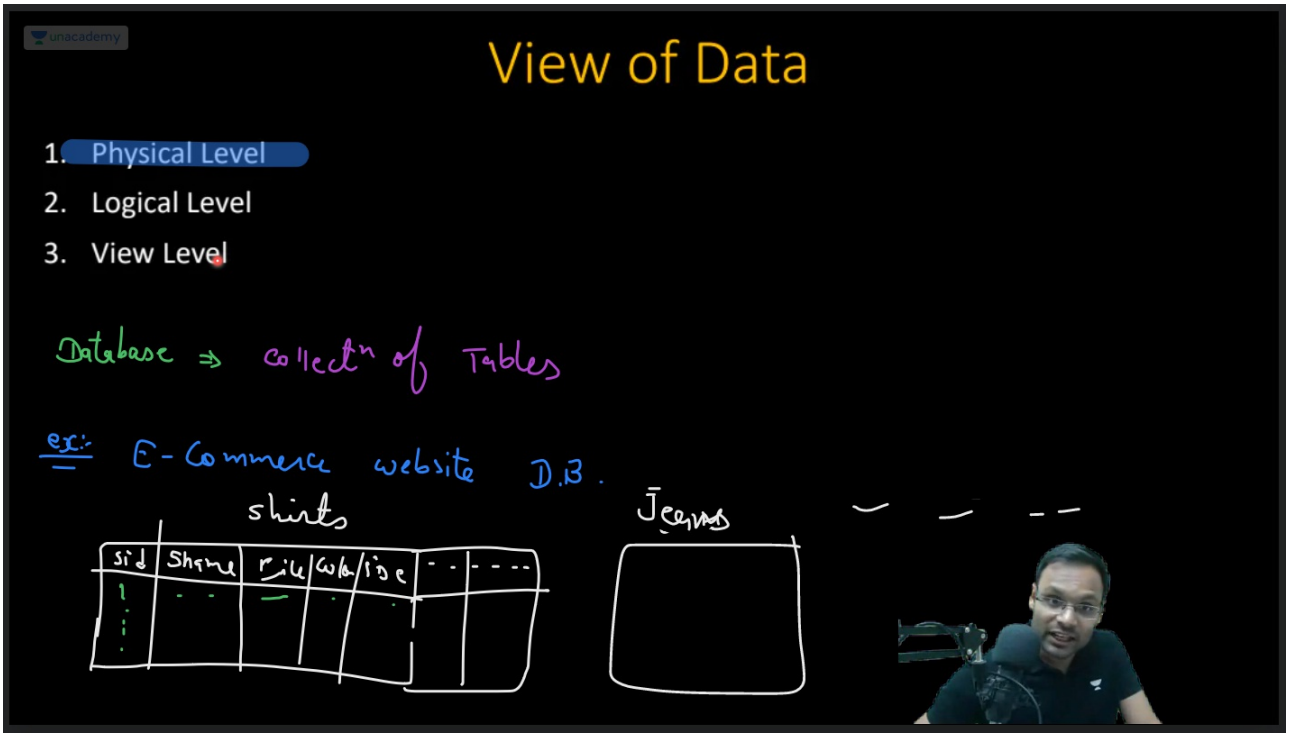
- How will see logical level?

> Not the **user**. The creator/maker of the database.

> Everything will be stored in the disk, but we don't know where it is **stored**. The tool(DBMS) only knows where the database is stored in the disk/memory.

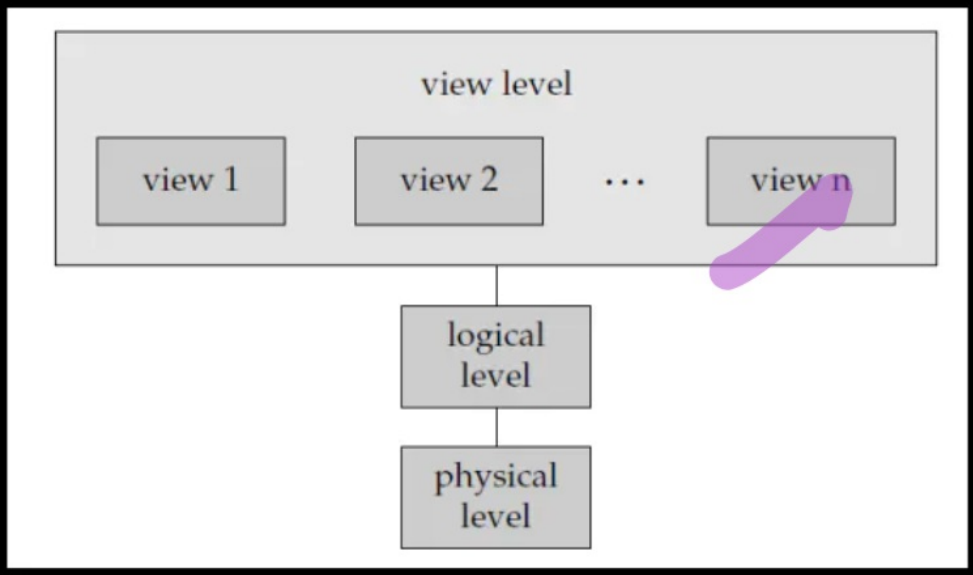
- From the logical level can be see the physical level info?

> **NO**. In-between there is the DBMS tool.

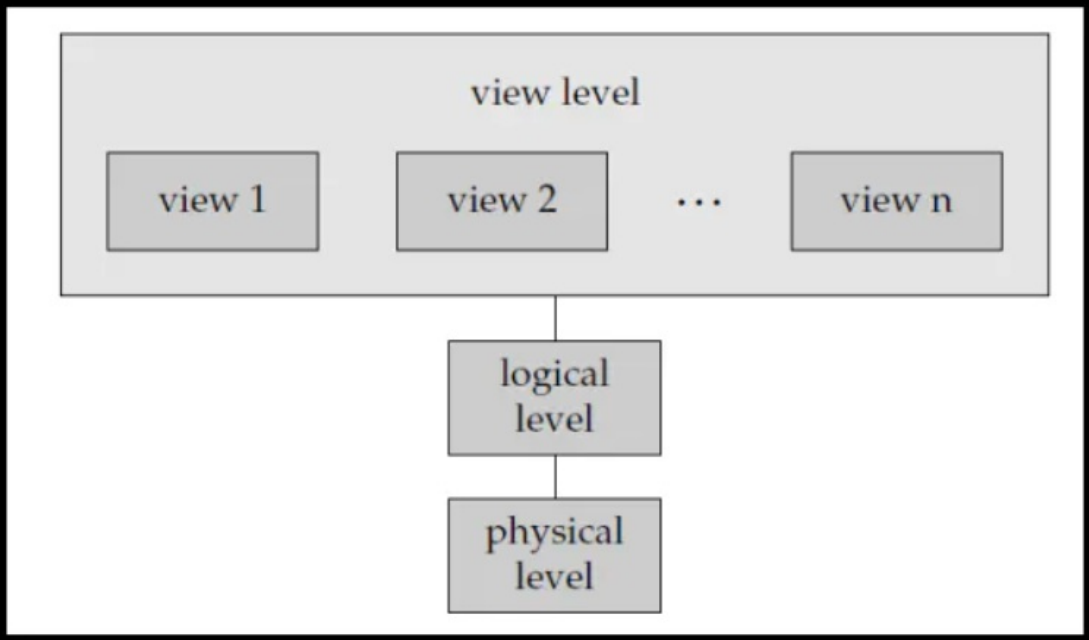


- Physical level is hidden from the **logical level**.

View of Data



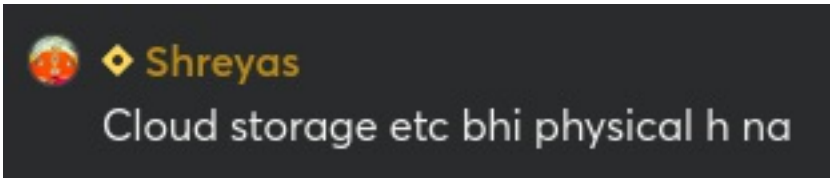
View of Data



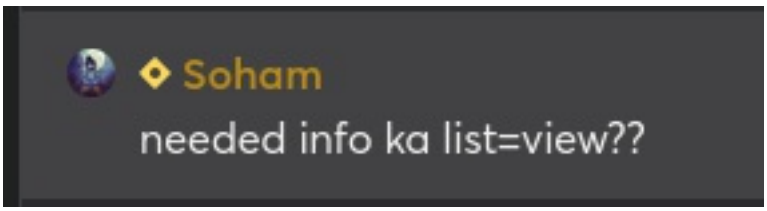
Rohit

Sir logical bs designing mai kaam aata h bs ?

- Yes.



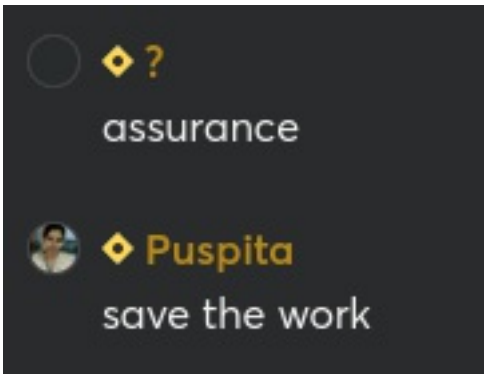
- Yes.
- From **view level**, **logical level** is **hidden**.



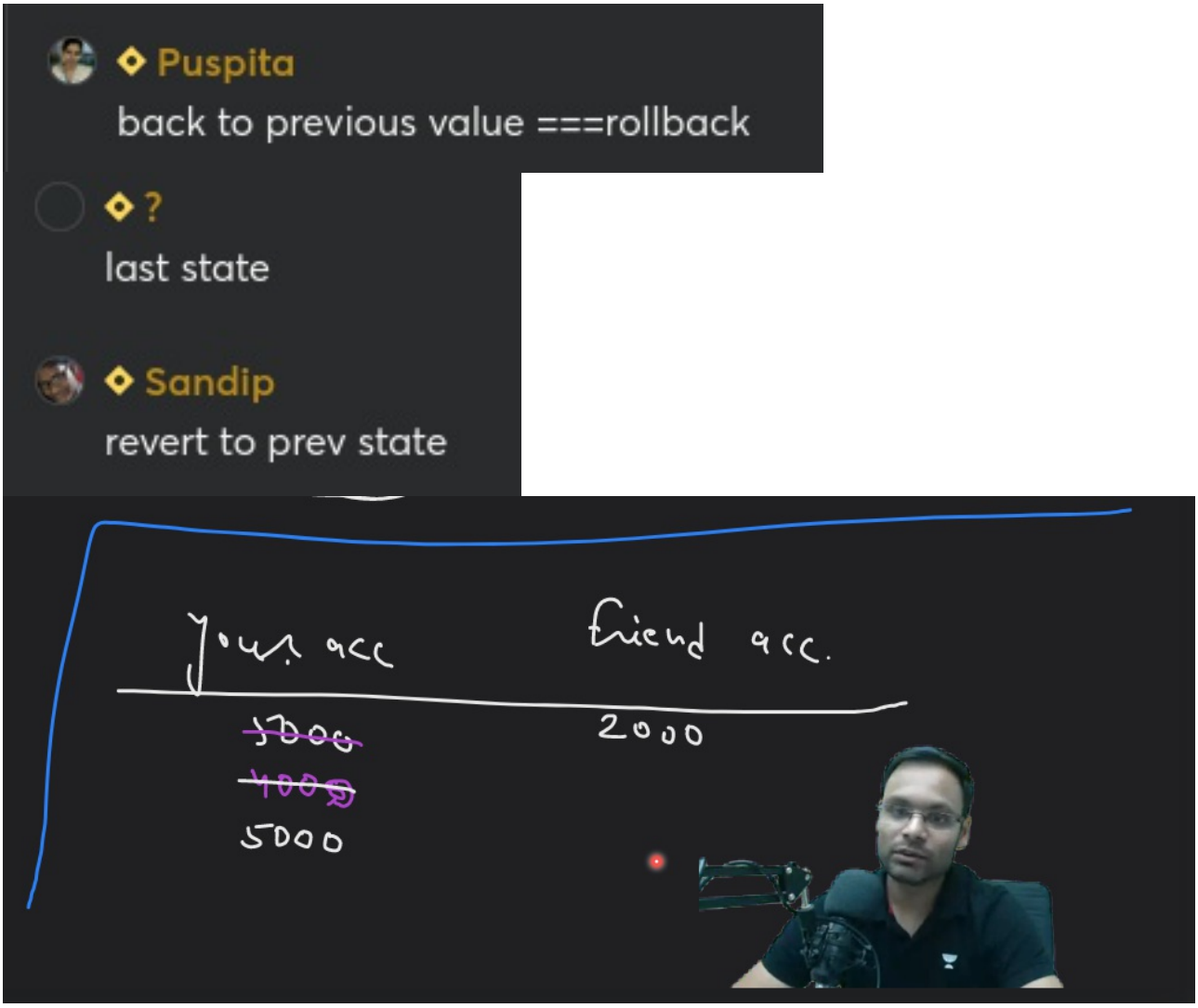
- Yes.

DBMS designing and er modeling (2) [24th June 2023]

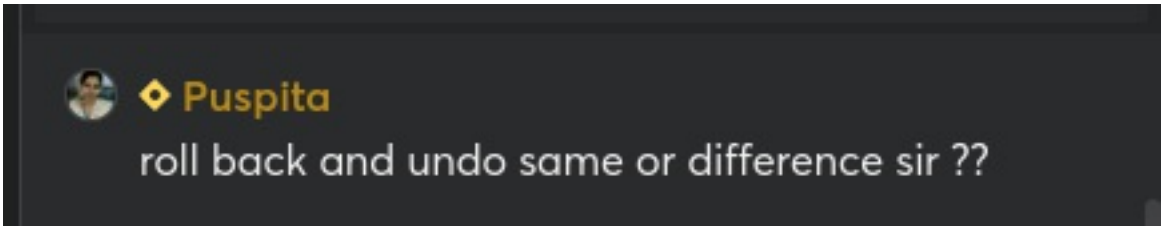
- Commit -> To make the changes permanent we use **commit**.



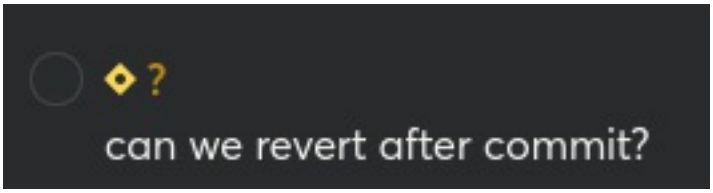
- Rollback -> Revert back all of the changes and go back to the last state where the DB was saved/commited(before the changes were made).



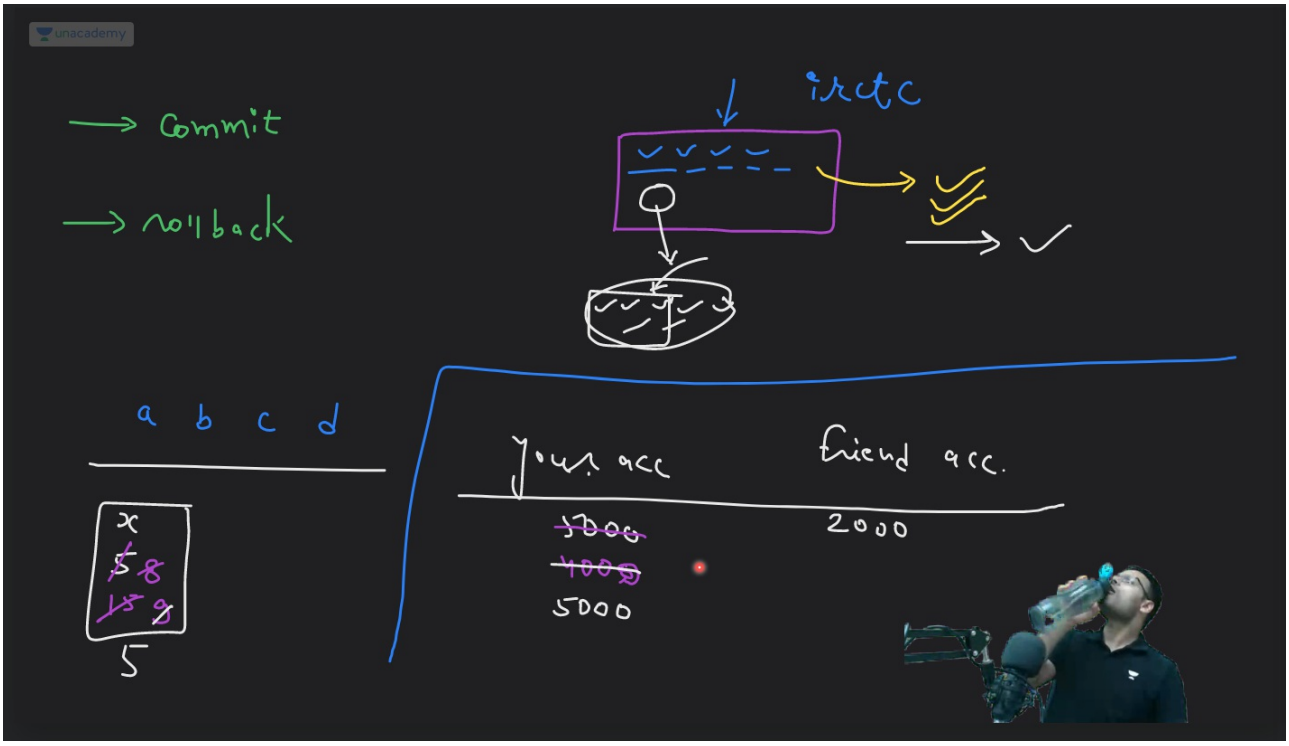
- Rollback.



- Different



- No.



- Atomicity happens on a complete transaction.
- We have created a file and we have not saved the file ourselves and there is no auto-save option as well. We have closed that file. What will happen?

> All the data is gone. This is called as **atomicity**.

- Atomicity -> **Full commit** if we **saved** the file otherwise the data in the file is lost(zero(0)). It is fully **automatic**, we didn't do anything.

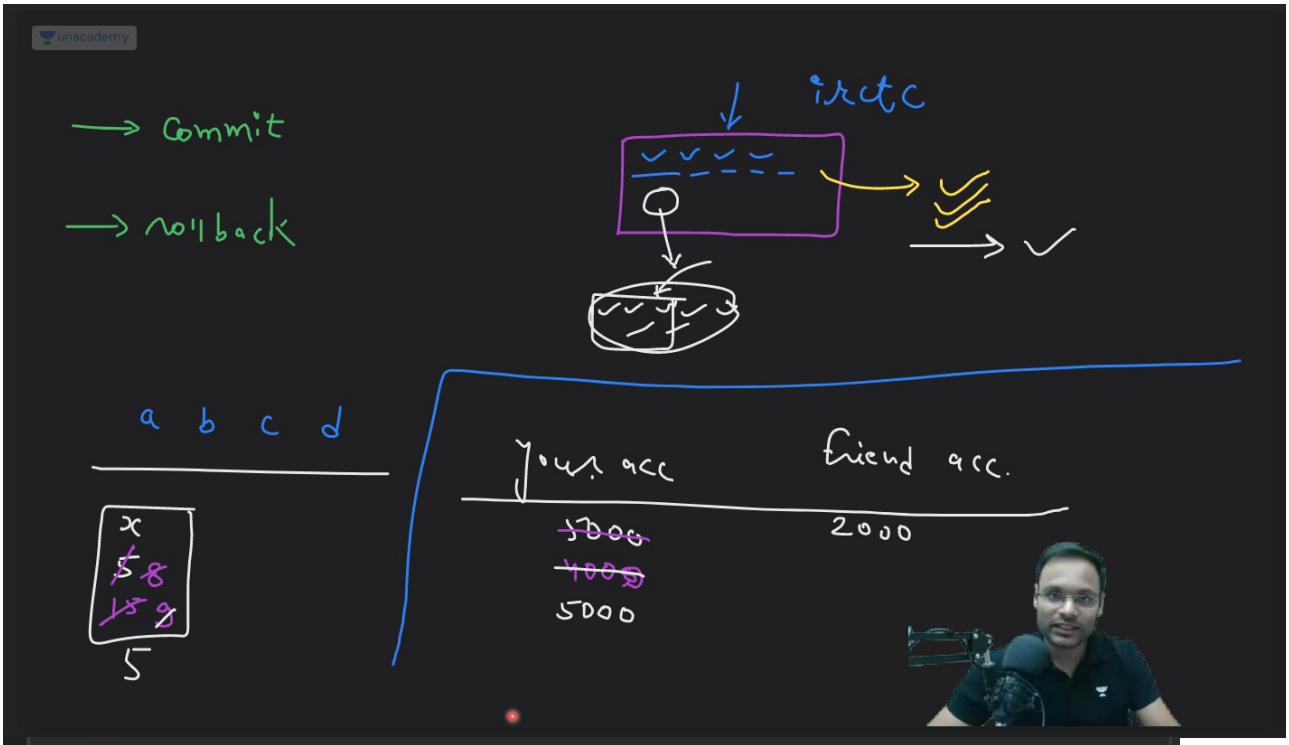
> We created a text file and one day's work is done. We have created a different copy of that file. Next day we created another copy of the file and started making changes on that file. We have saved all of the changes as well but we don't like the changes. We will take the previous copy as the original copy and delete the modified one.

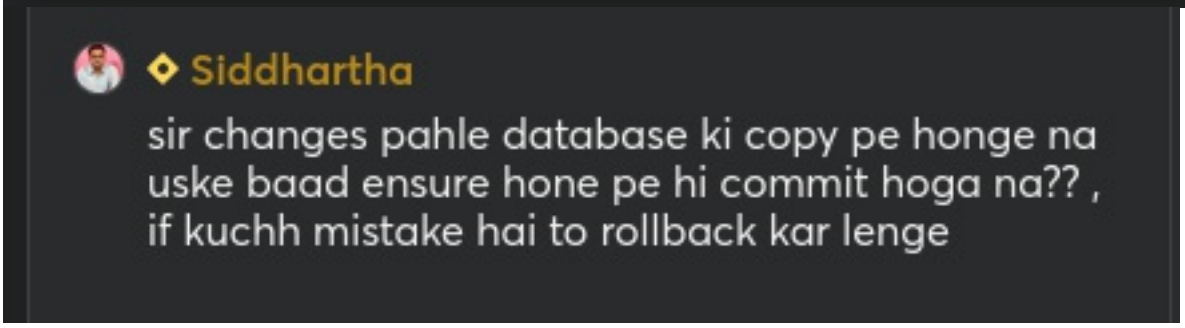
> This is called as **rollback**. It has happened because of our decisions.

- Atomicity -> Automatic rollback to the previous value, done by the DBMS tool.

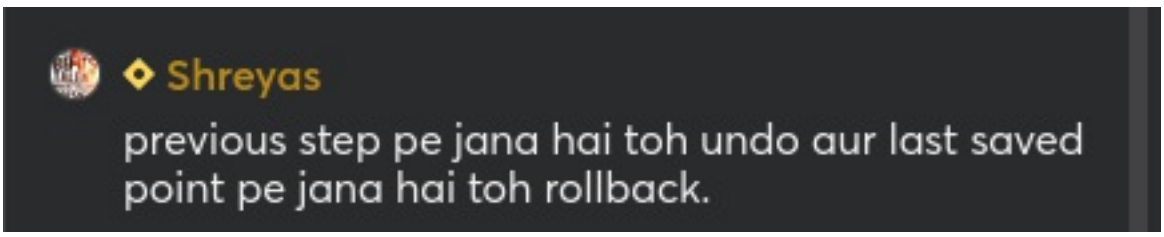
> When the transactions stops in the **middle**, fails. It is called as **Atomicity**.

- Rollback -> We can rollback to a previous state when we wish like it until a commit is done.

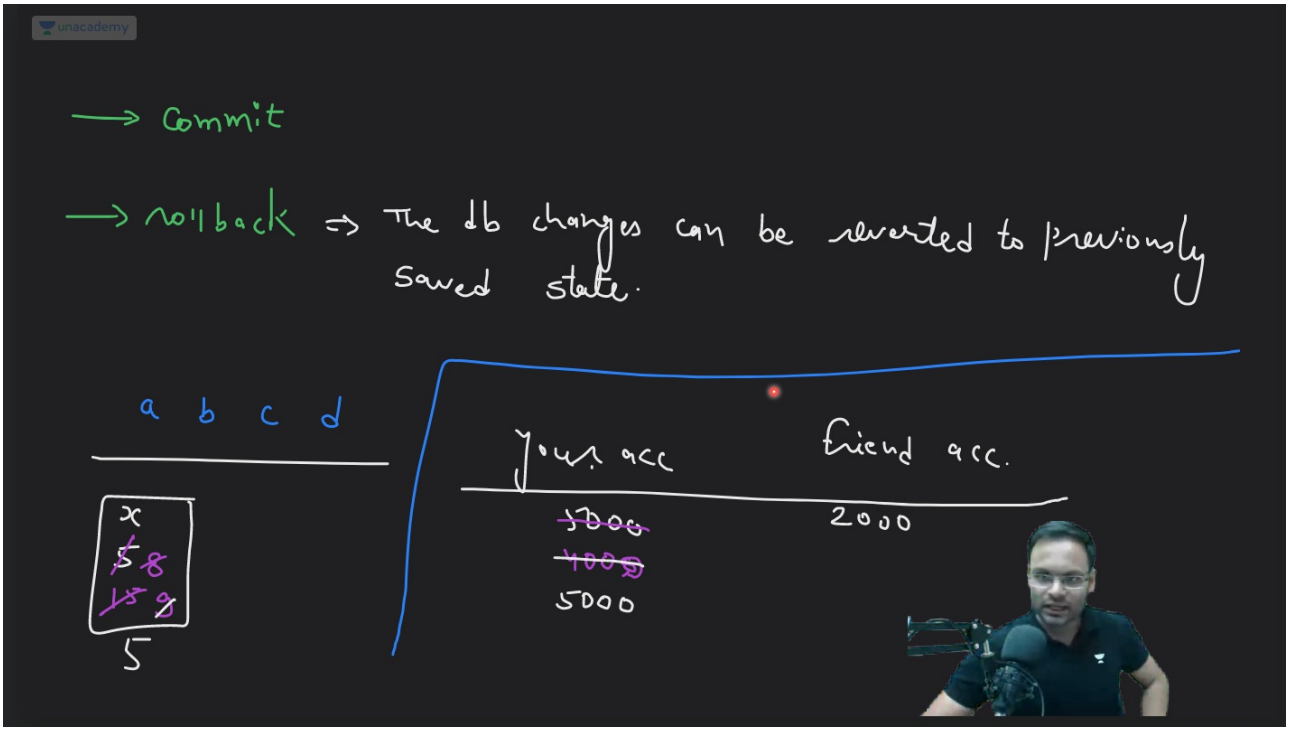

unacademy
Commit
rollback
irctc
a b c d
x
5
8
9
5
your acc
friend acc.
5000
2000
5000


Siddhartha
sir changes pahle database ki copy pe honge na uske baad ensure hone pe hi commit hoga na?? , if kuchh mistake hai to rollback kar lenge

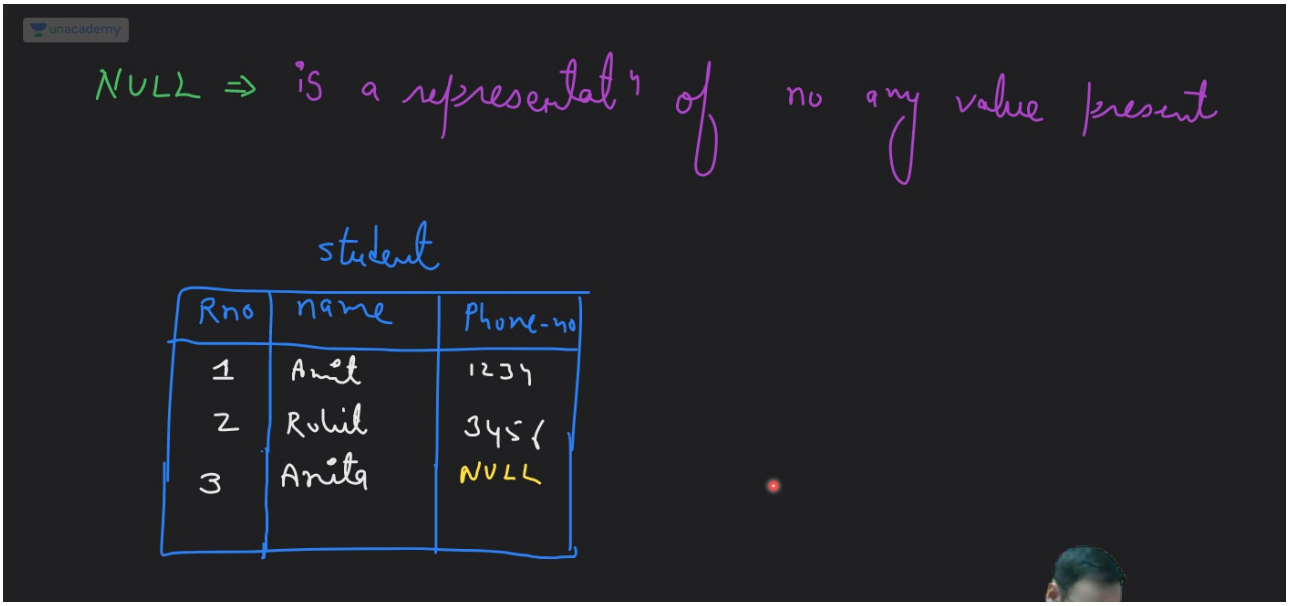
- Yes


Shreyas
previous step pe jana hai toh undo aur last saved point pe jana hai toh rollback.

- Yes



- Rollback -> The DB changes can be reverted to previously saved state.
- NULL -> It is a representation of no any value present.



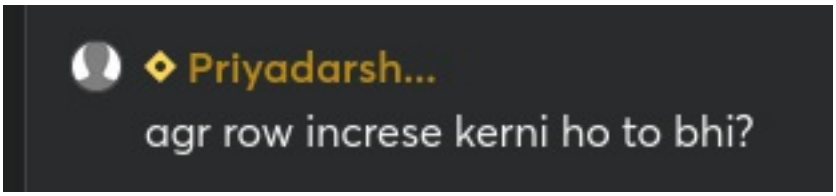
Database Languages

1. Data-Definition Language(DDL) -> If we want to work on the design of the tables, the statements which we have to write for that, they will come in **Data-Definition Language(DDL)**

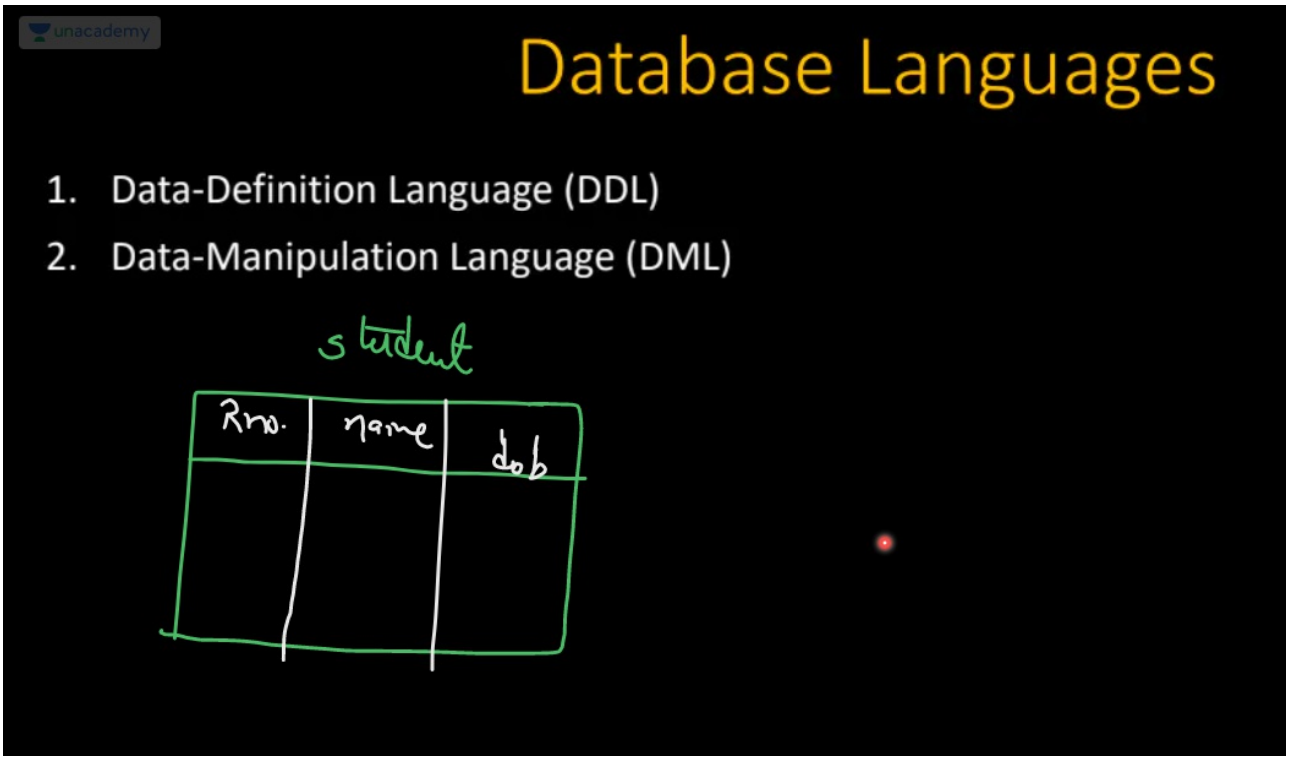
- **Working on DB design**.
- Creating, deletion tables, updating columns, name, deleting columns, adding new column

2. Data-Manipulation Langugae(DML) -> If we work on the data of the table, the data within the table, they come under **DML**.

- **Working on data**.
- Insert, update data, delete data, fetching data.
- If we are working on **data** then it is **DML**.



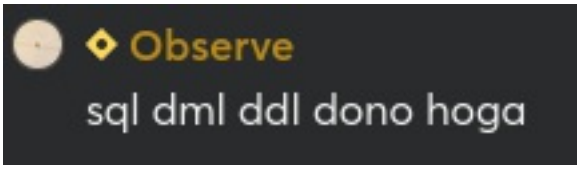
- Working on the design of the table. So it is **DDL**.
- Making the table's design is **DDL**.



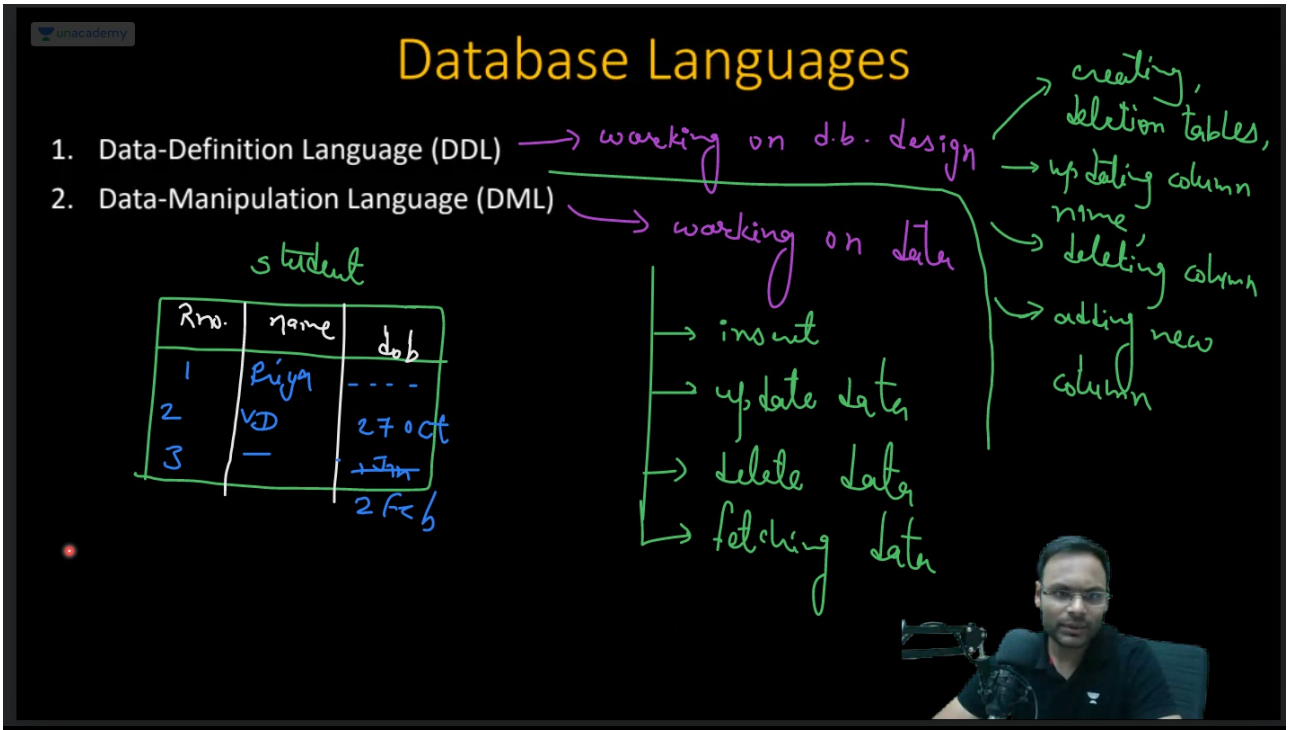
- We want student's info whose **Rno** is greater than **10**?

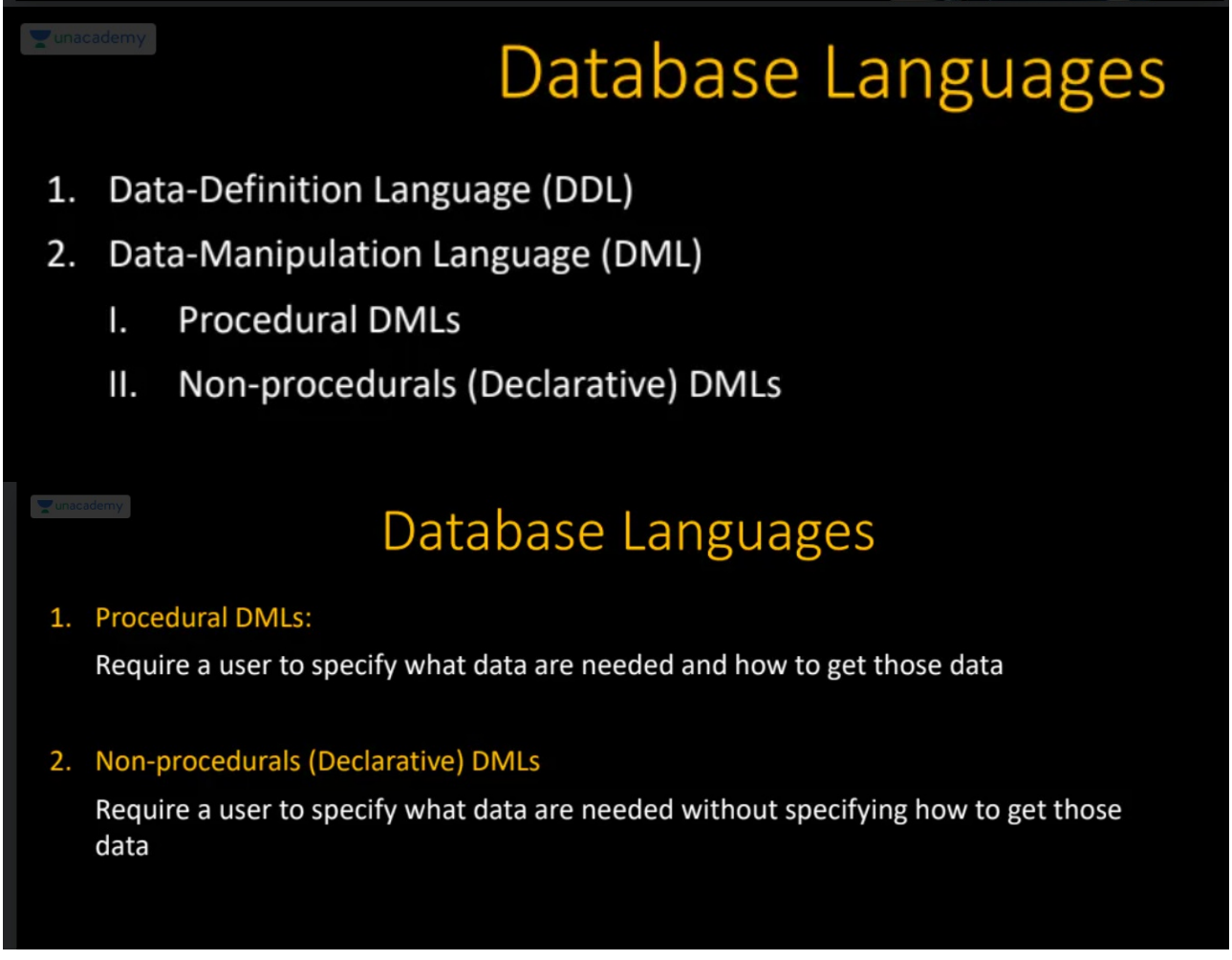
> It is done using **DML**.

- Insertion of rows, deletion of rows, deleting data, fetching data -> Working on data -> **DML**.
- Table's name, column names, no. of columns, add new column, delete a column, delete whole table -> Working on table's design -> **DDL**



- YES.

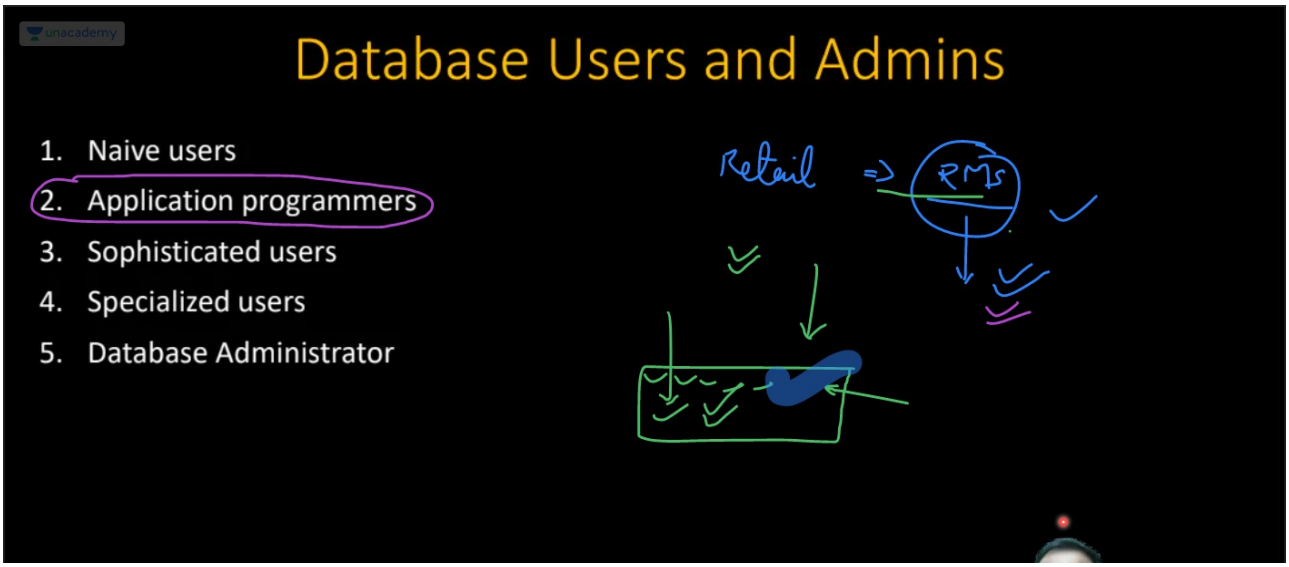




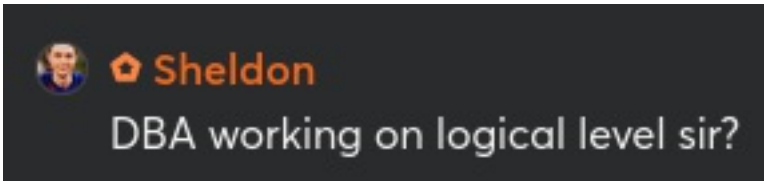
1. Procedural DMLs
2. Non-procedurals(Declarative) DMLs -> SQL

- SQL is **Non-Procedural** language.

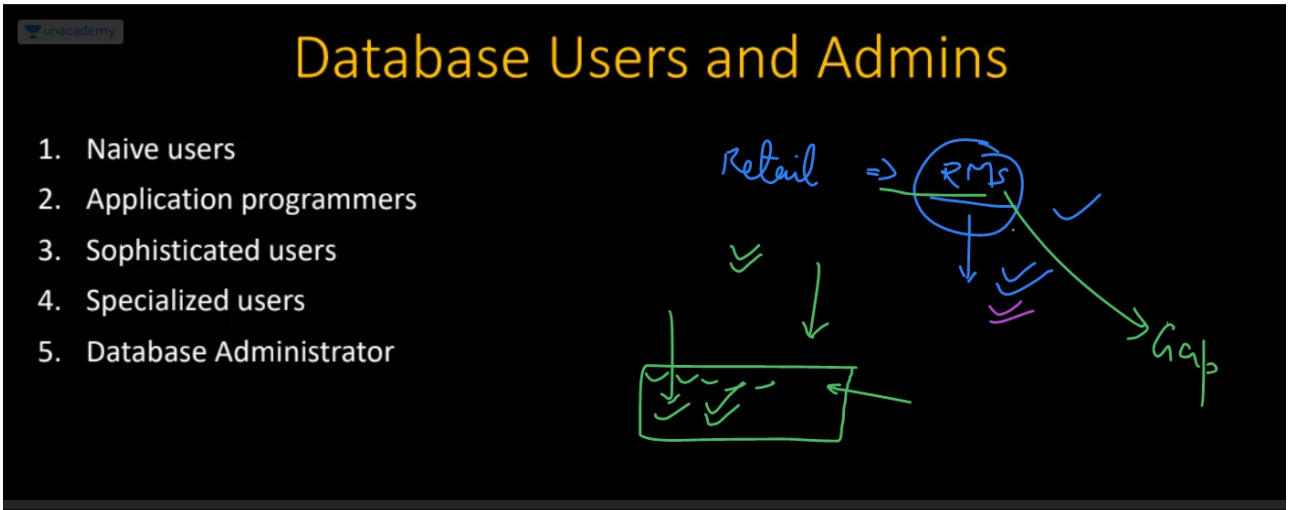
Database users and admins

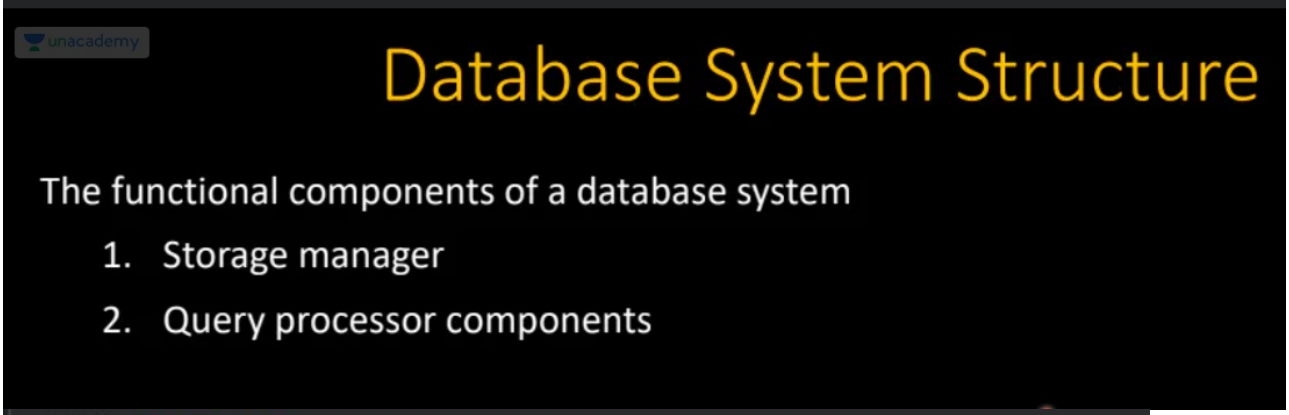


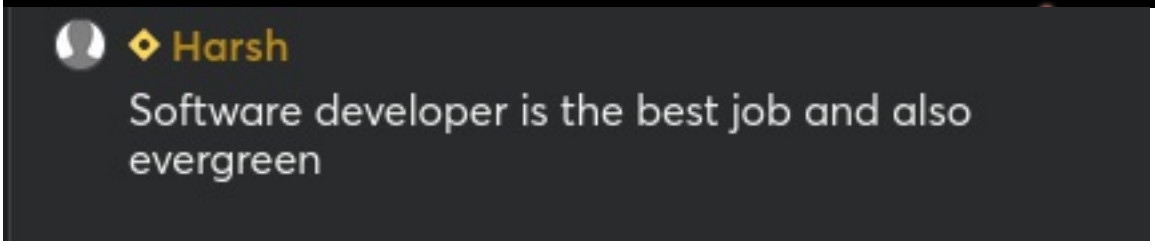
1. Naive Users -> No idea that DB is there -> Normal people
2. Appliocation programmers -> Writing Application -> Developing softwares.
3. Sophisticated users -> Use/write query to access DB.
4. Specialized users -> Developes query language or DBMS tool.
5. Database Administrator -> DBA -> Any DBMS highest level person which manages database. Databae's admin.



- Yes.







Data Model

- When we store the data in the **database**, the data is represented in which format. That is Data Model.
- How do you reperesent the data when we create the **database**?

> That is Data Model.

1. ER modelling -> Entity-Relationship Model
2. Relational Model

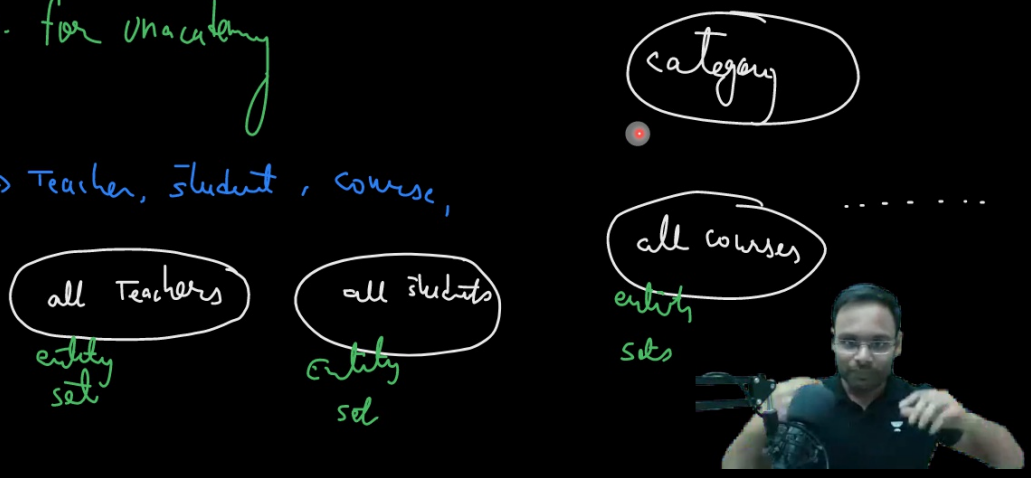
The Entity-Relationship Model

The Entity-Relationship Model

The entity-relationship (E-R) data model consists of a collection of basic objects, called entities, and of relationships among these objects

ex:- D.B. for unacademy

obj. ⇒ Teacher, Student, Course,

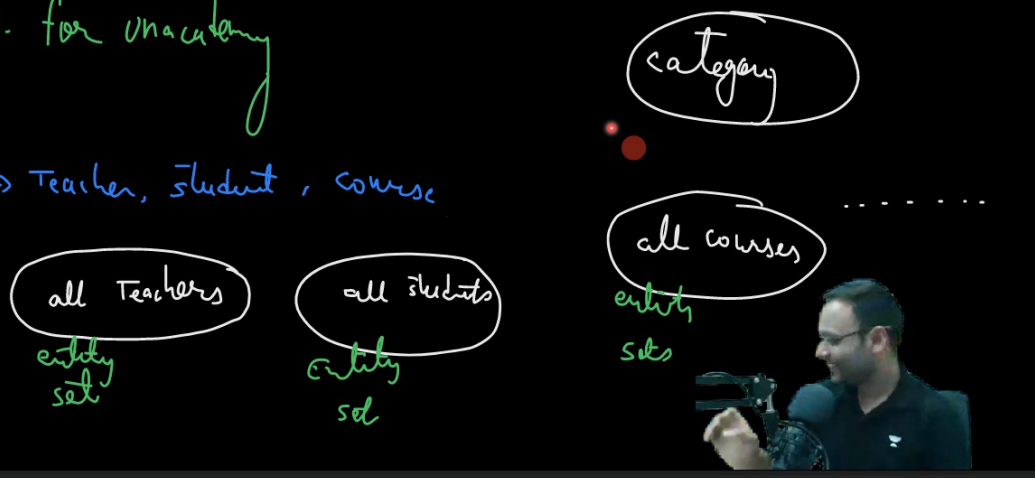


The Entity-Relationship Model

The entity-relationship (E-R) data model consists of a collection of basic objects, called entities, and of relationships among these objects

ex:- D.B. for unacademy

obj. ⇒ Teacher, Student, Course

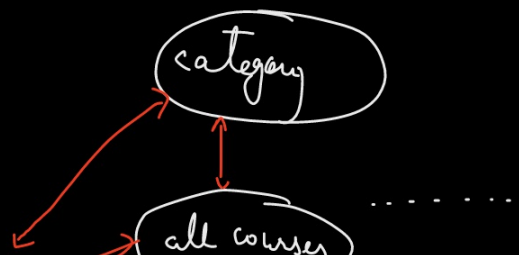


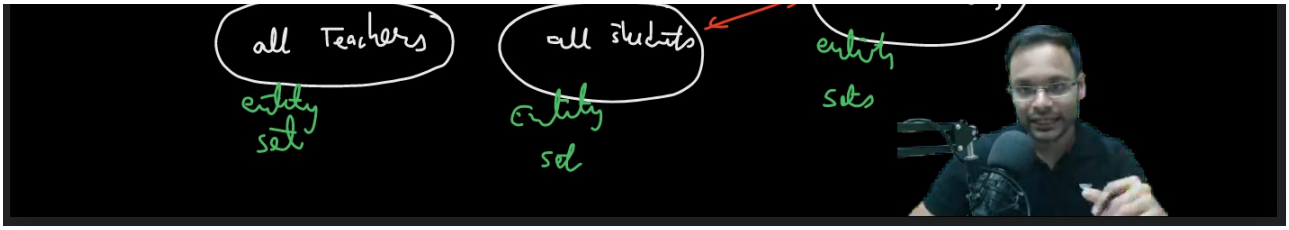
The Entity-Relationship Model

The entity-relationship (E-R) data model consists of a collection of basic objects, called entities, and of relationships among these objects

ex:- D.B. for unacademy

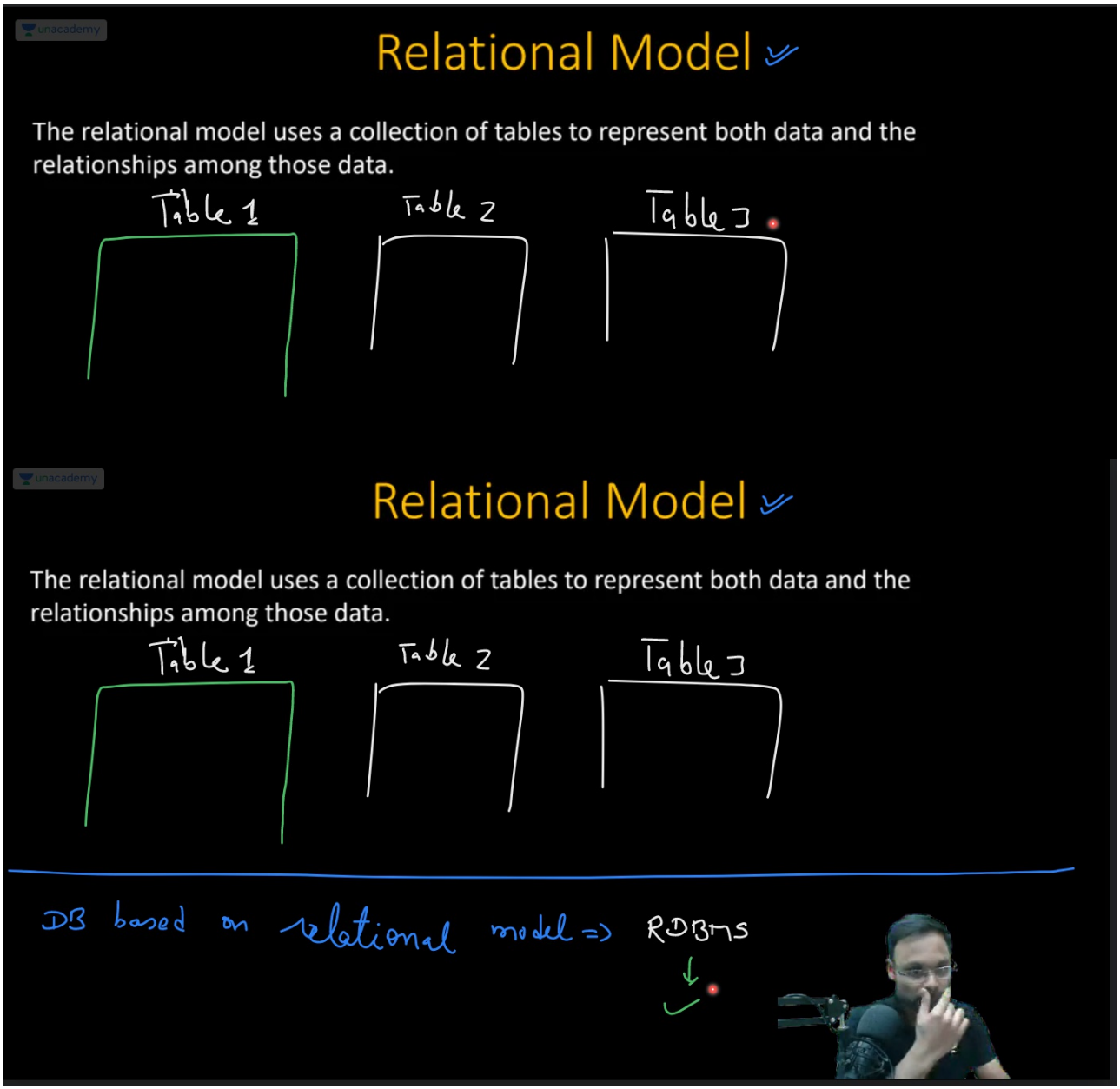
obj. ⇒ Teacher, Student, Course





- Entity -> Just one teacher's information
- Entity set -> Will become the table later on.

Relational Model



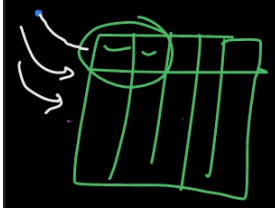
- DB based on relational model -> **RDBMS**
- ER Modelling is the basic way to start database design. ER Modelling is the first step to database design even if the database is for **Relational Model**.

Database Design

1. Requirement Analysis
2. Conceptual Database Design -> ER- modeling
3. Schema Refinement -> ER to relational and refinement
4. Logical Database Design -> Use DB tool and make DB
5. Physical Database Design -> Decisions related to physical level of DB (Example -> Indexing)
6. Security Design -> Related to security.

Database Design

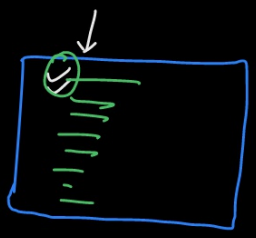
1. Requirement Analysis
2. Conceptual Database Design → E-R modeling
3. Schema Refinement → E-R to relational & refinement
4. Logical Database Design → Use DB tool & make db
5. Physical Database Design → decisions related to physical level of db (ex:- Indexing)
6. Security Design



A _

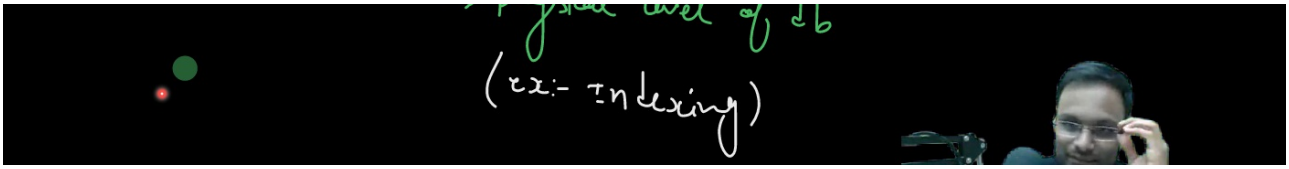
Database Design

1. Requirement Analysis
2. Conceptual Database Design → E-R modeling
3. Schema Refinement → E-R to relational & refinement
4. Logical Database Design → Use DB tool & make db
5. Physical Database Design → decisions related to physical level of db (ex:- Indexing)
6. Security Design



Database Design

1. Requirement Analysis
2. Conceptual Database Design → E-R modeling
3. Schema Refinement → E-R to relational & refinement
4. Logical Database Design → Use DB tool & make db
5. Physical Database Design → decisions related to physical level of db
6. Security Design



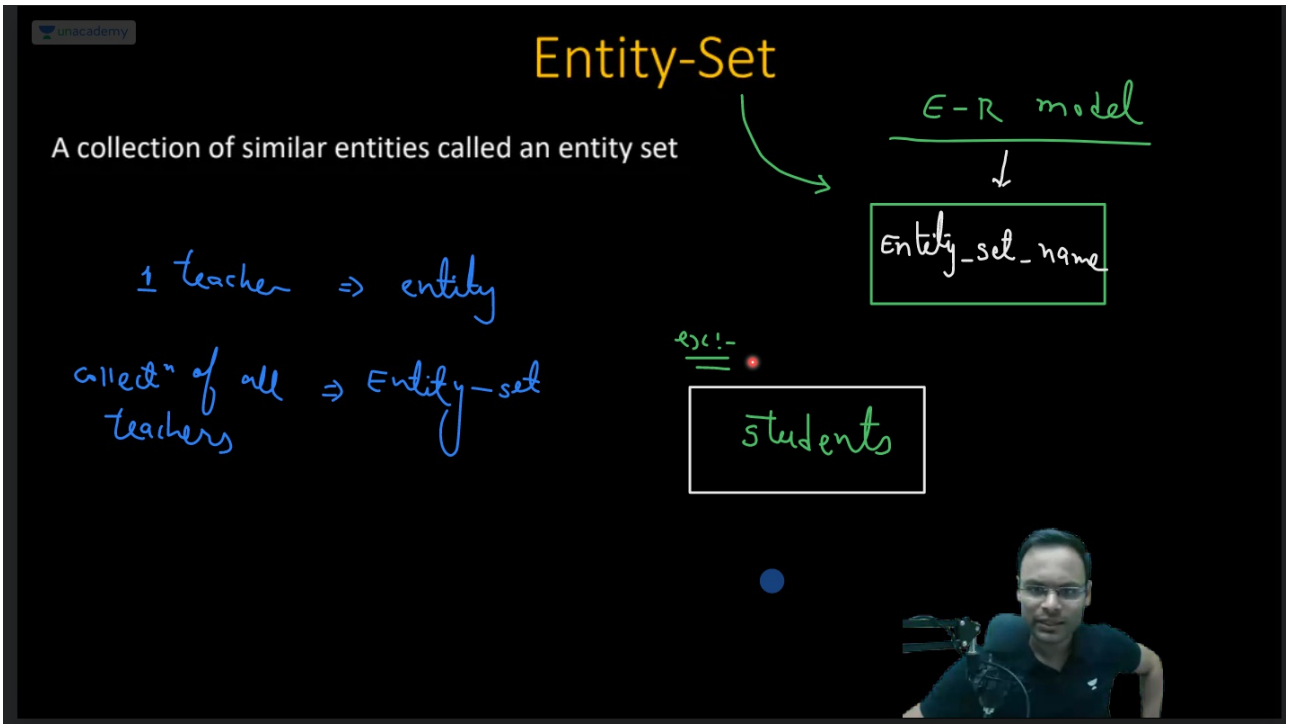
Er modeling part-II (3) [24th June 2023]

Entity

- Object in the real world that is distinguishable from another object.

Entity Set

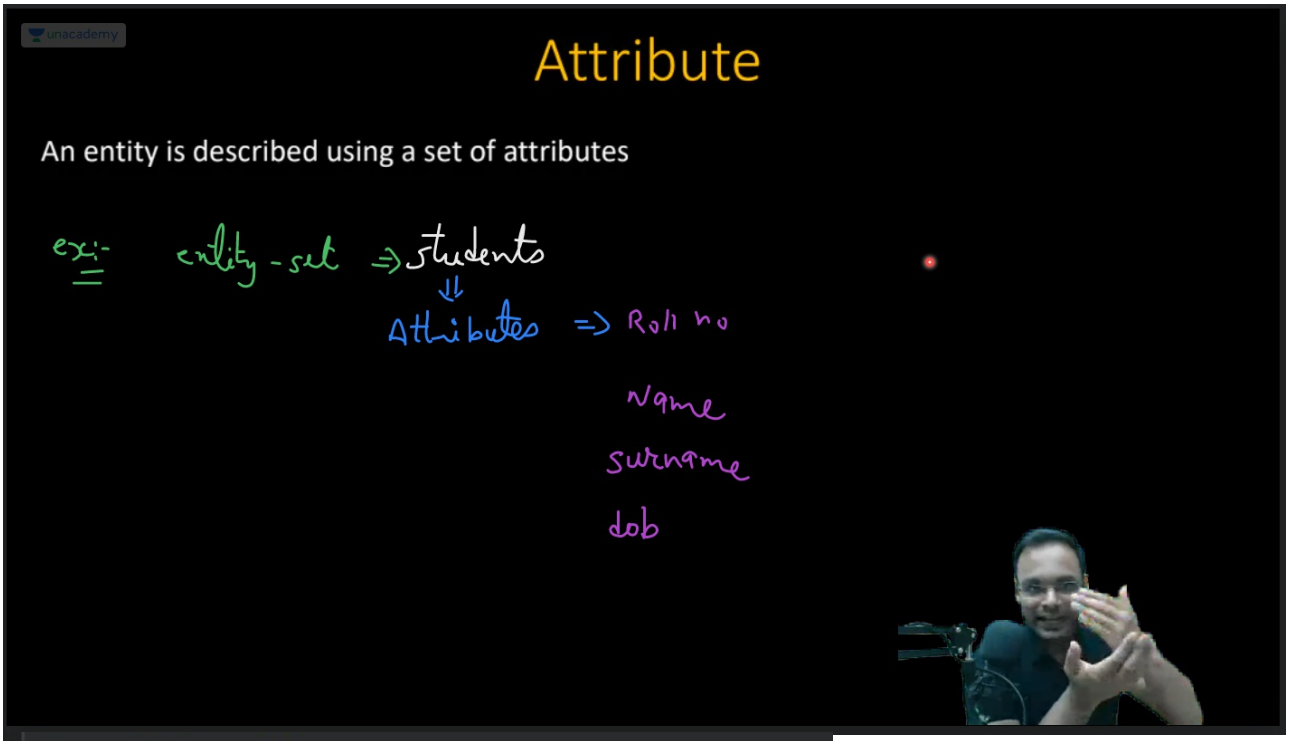
- A **collection** of similar entities is called as entity set.
- 1 Teacher -> Entity
- Collection of all teachers -> Entity-set

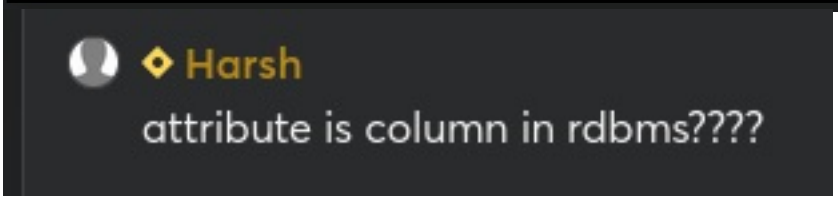


- An Entity set has multiple different entities.

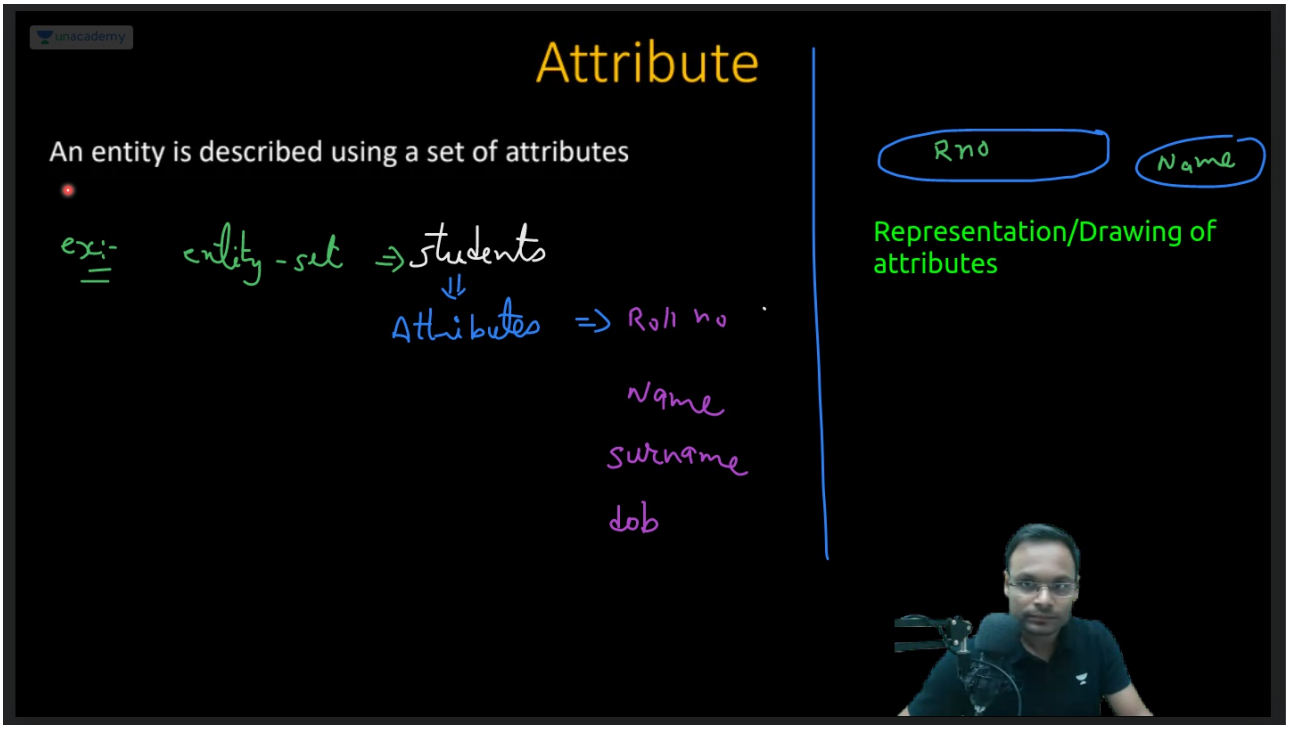
Attribute

- An **entity** is described using a set of attributes.
- To **describe** students/student(entity).





- Kind of yes.
- Every **entity** will have it's **attributes**. Ultimately the **attributes** will become **attributes** of the **entity set** as well.

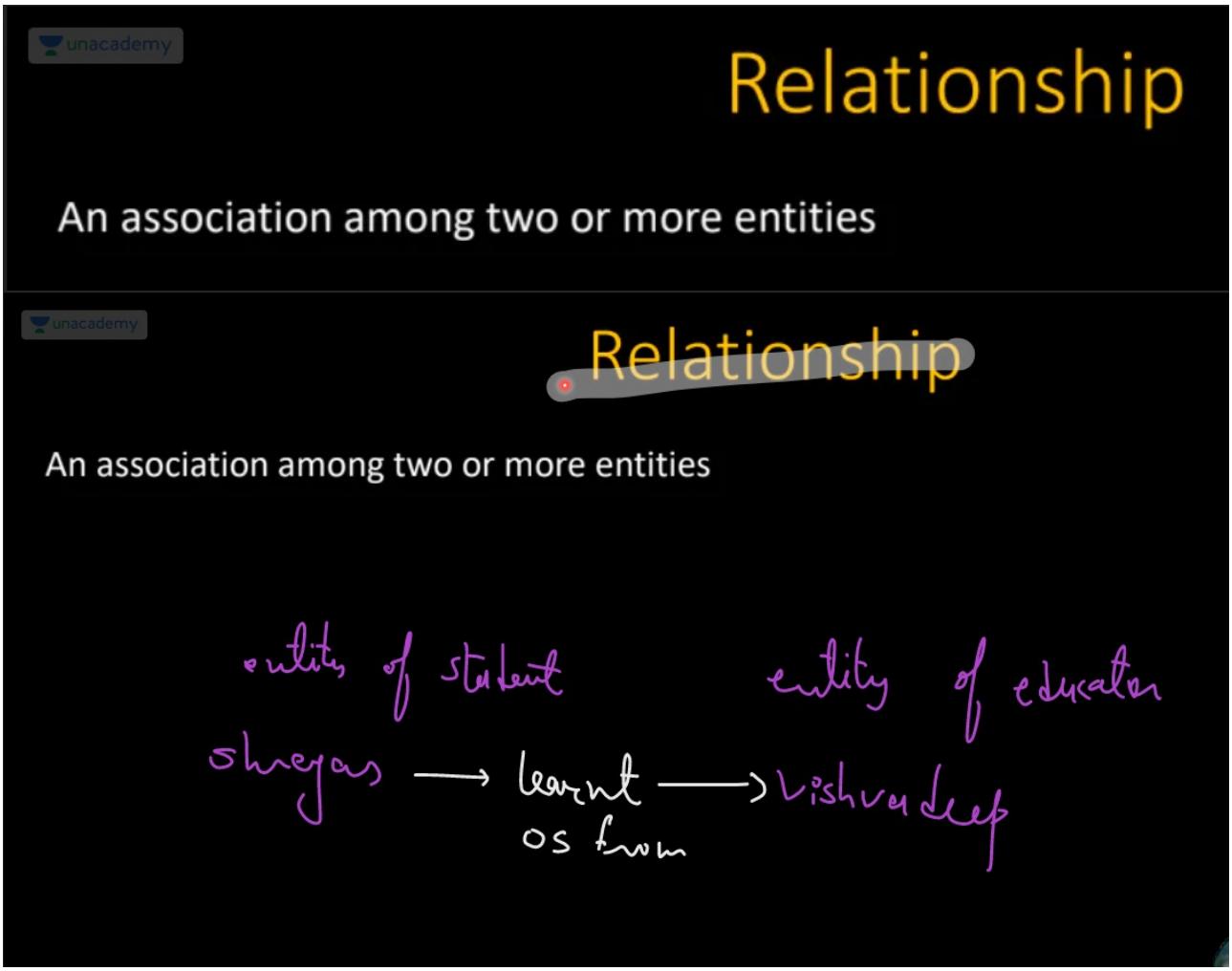


Domain

- A unique set of values permitted for an attribute.
- The **permitted values** which can come to the **attribute** are called as **domains**.

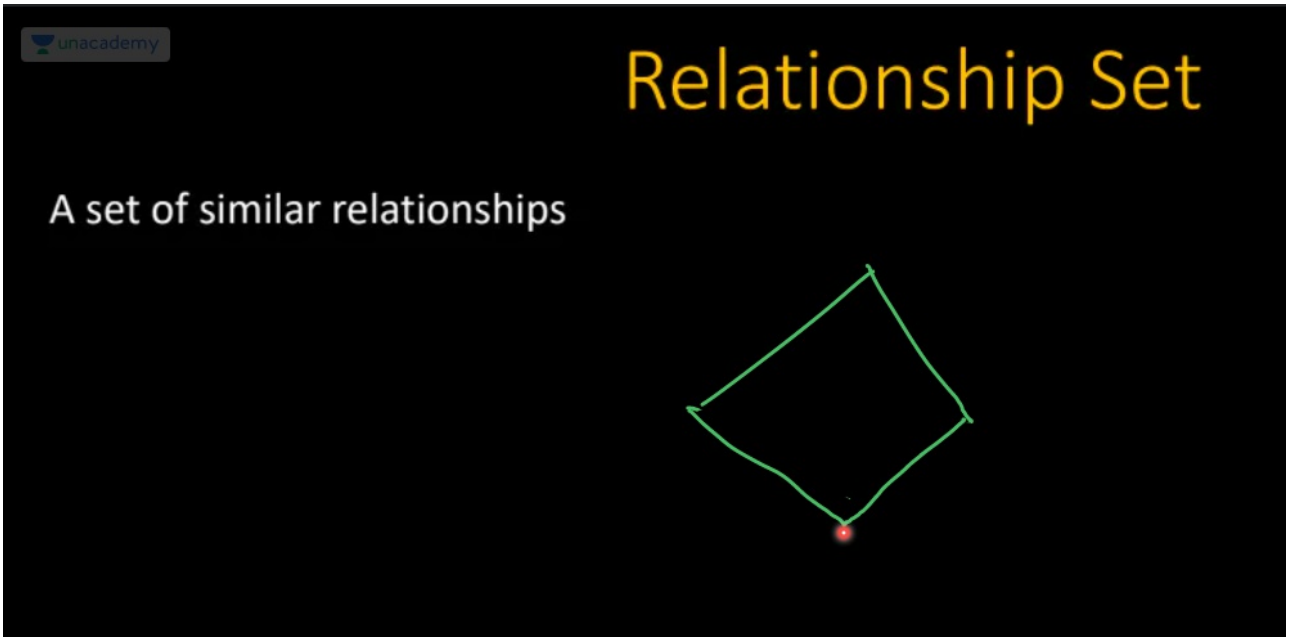
Relationship

- An association among two or more entities.



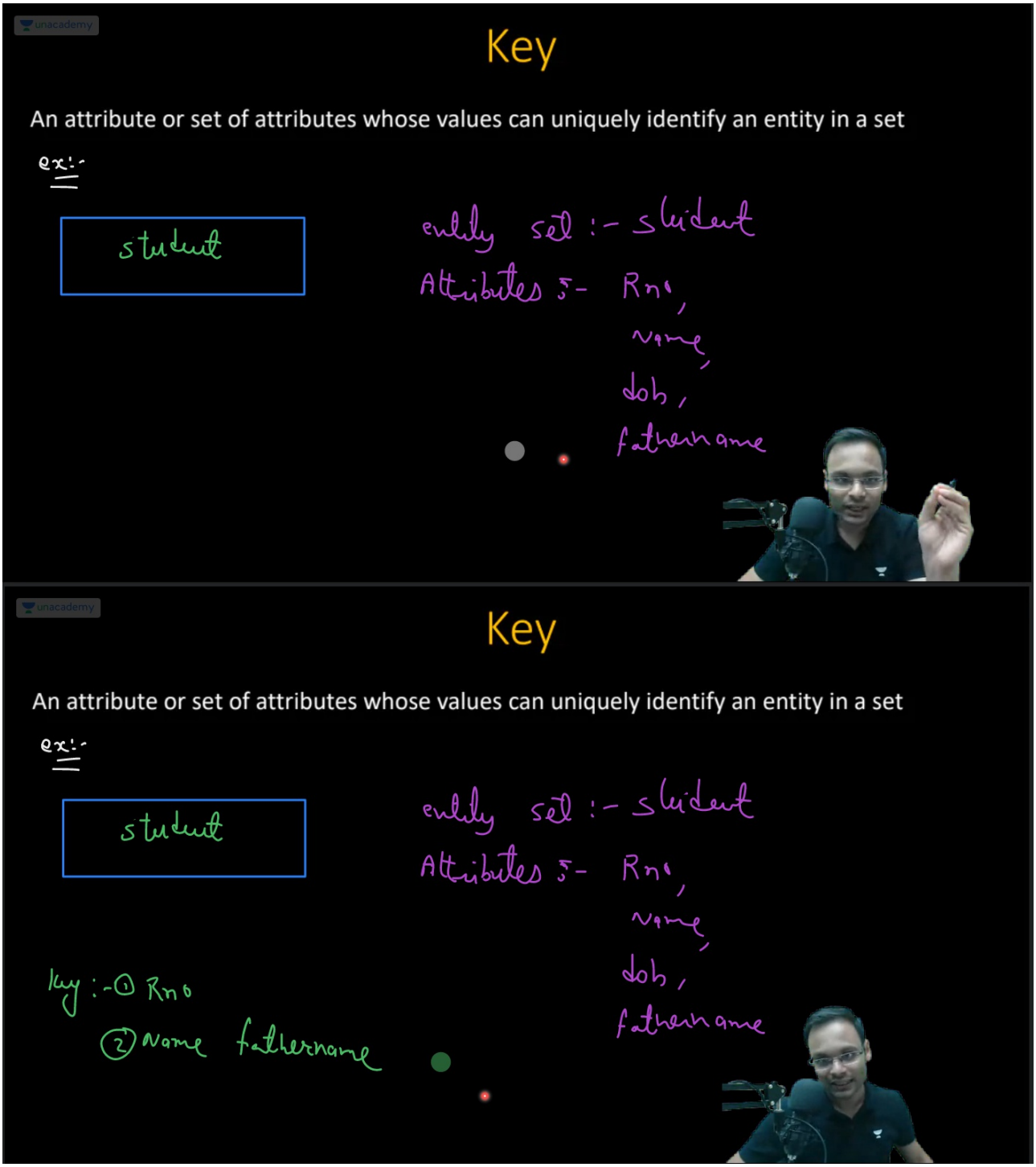
Relationship set

- A set of similar relationships.



Key

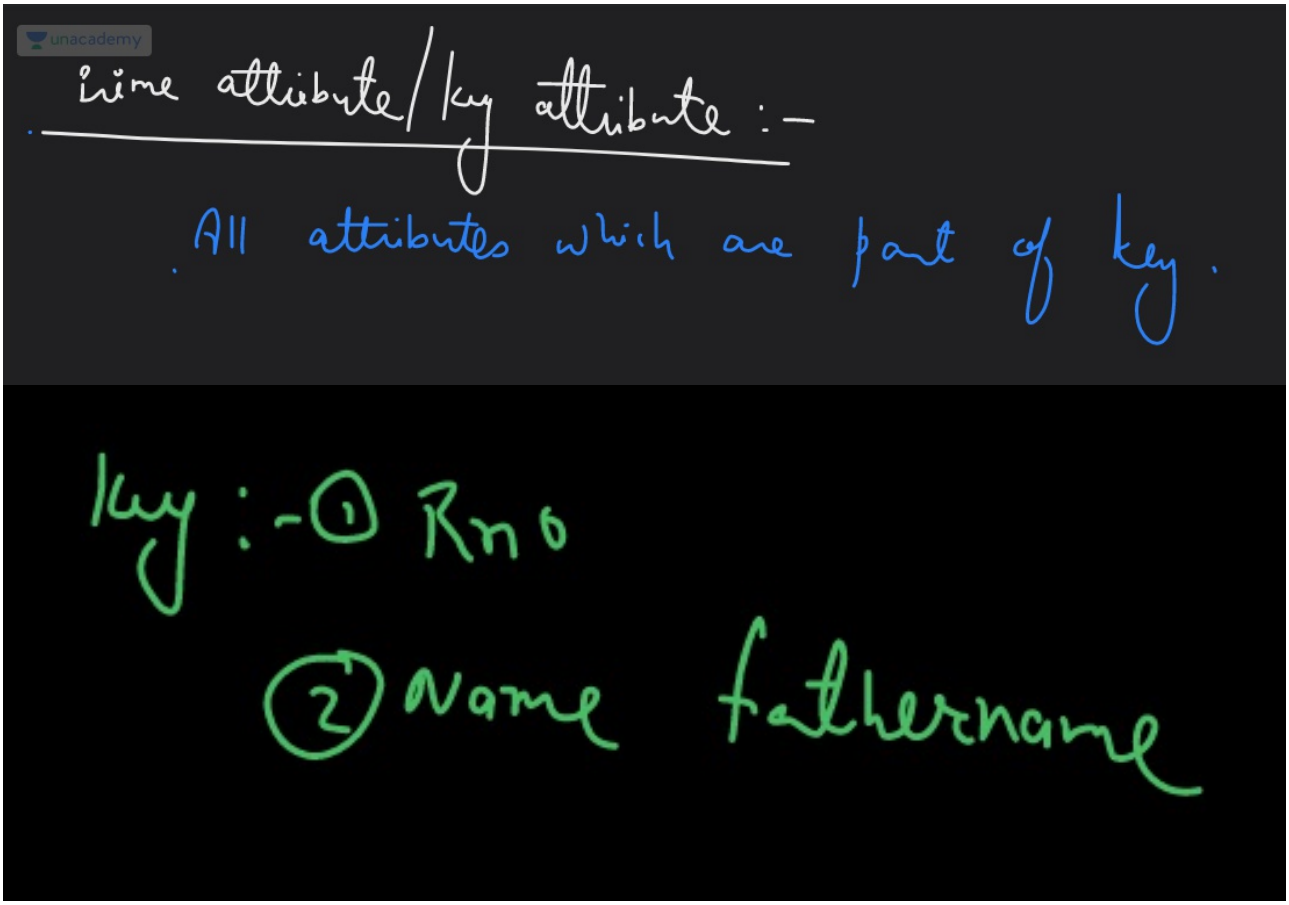
- An attribute or set of attributes whose values can uniquely identify an entity in a set.
- More than **one attribute** can come into a **key**.



- Name + fathername -> The combination is called as a **key**.

Prime Attribute/Key attribute

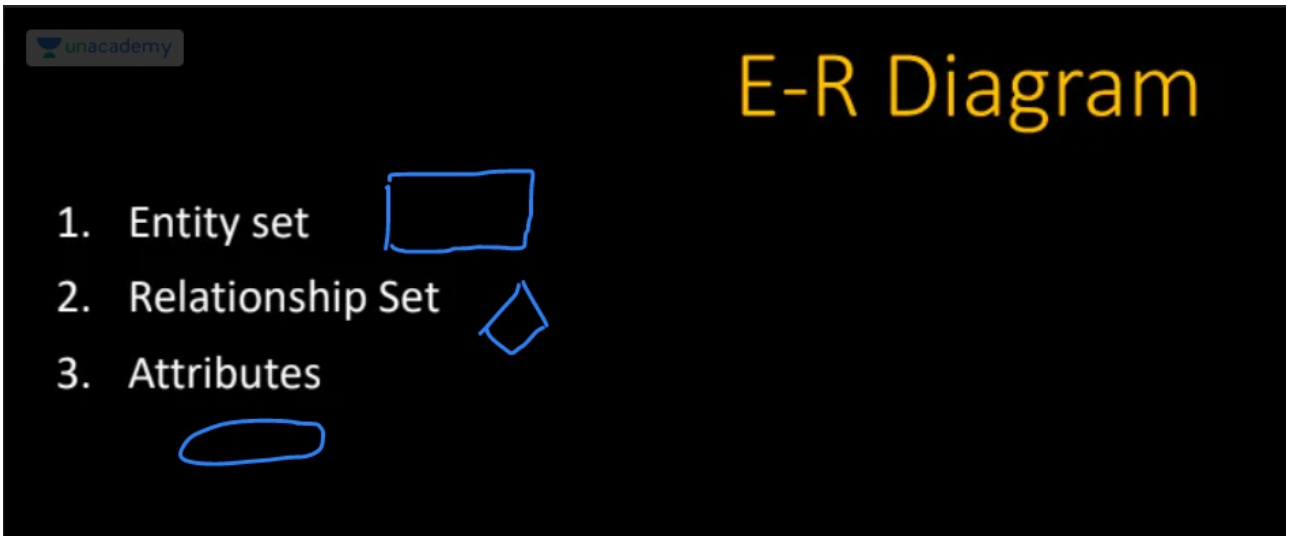
- All attributes which are part of key.



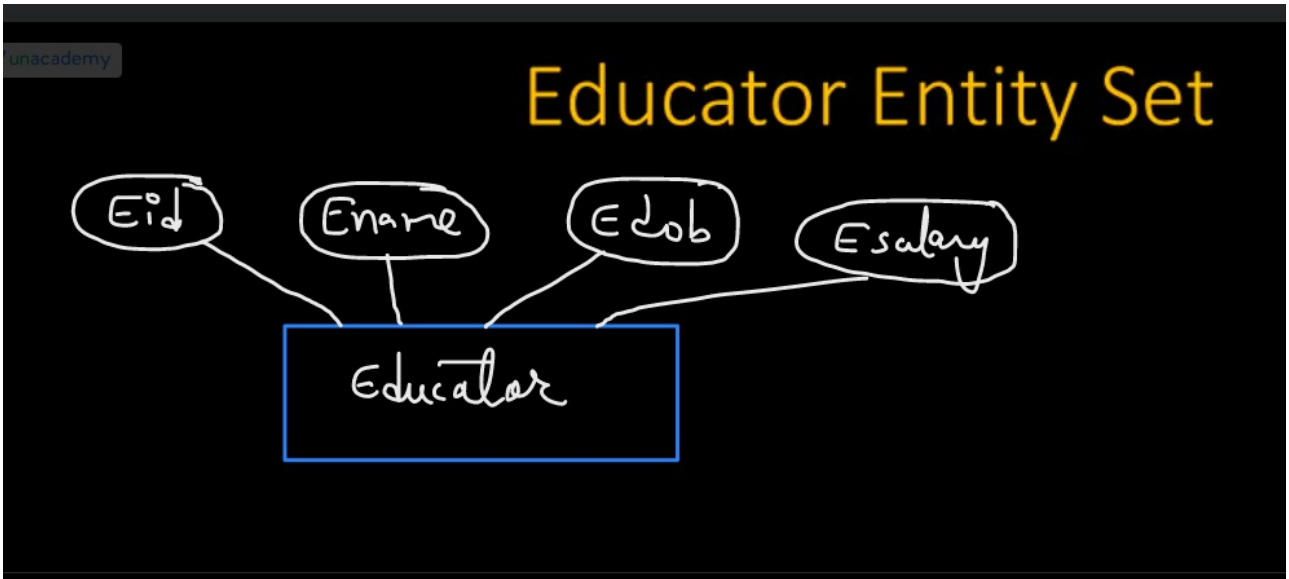
- Rno, name, fathername -> **Prime Attribute**
- Is it compulsory that every entity set have keys?

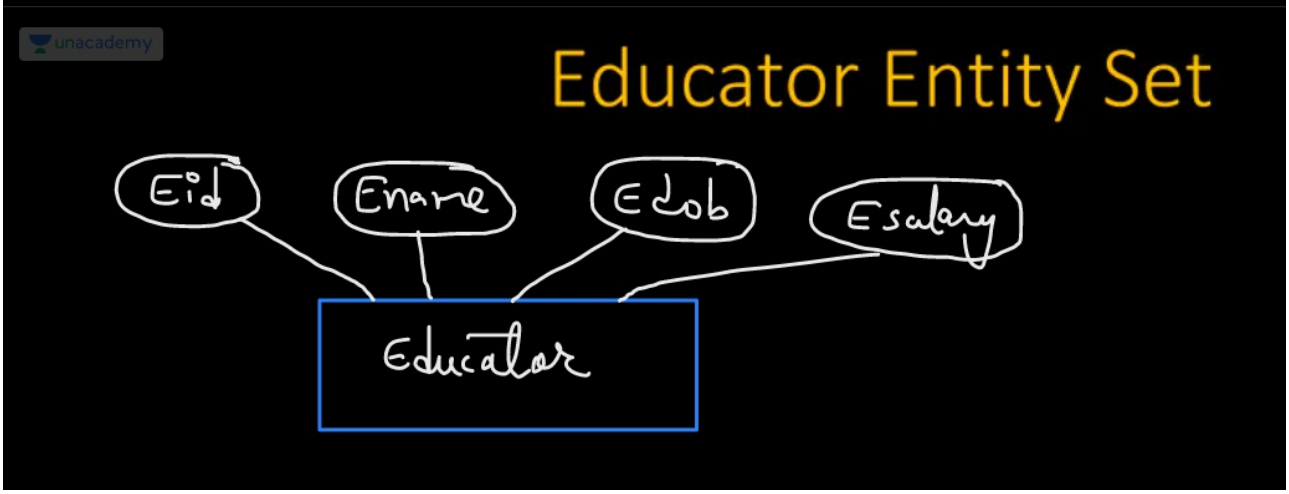
> **No**. If needed then we will make it, not compulsory.

ER diagram representation shapes



1. Entity Set -> Square
2. Relationship set -> Diamond
3. Attributes -> Circle.

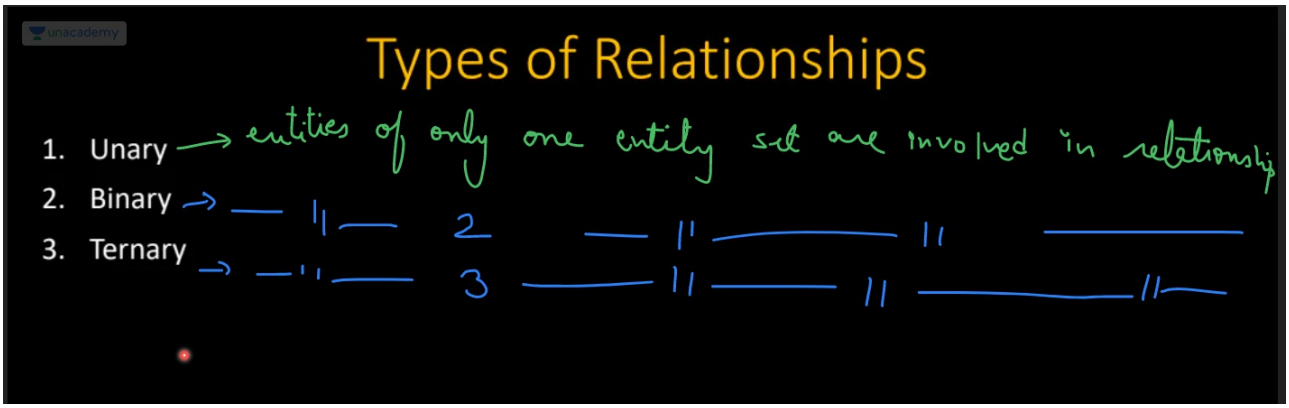




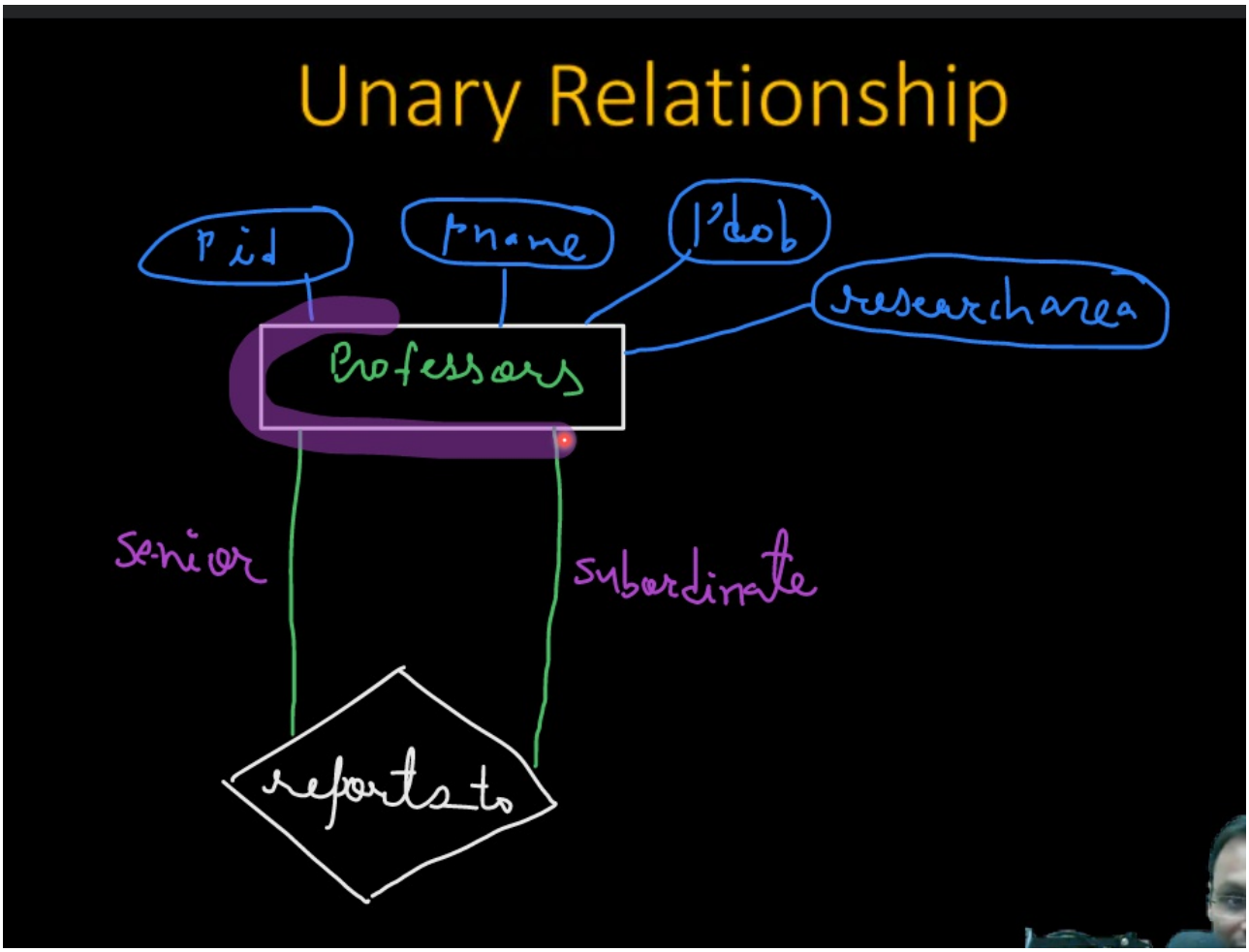
- Entity Set diagram.
- Entity Set -> Educator
- Attributes -> Eid, Ename, Edob, Esalary
- The Entity Set, Educator can have many educators. We don't know how many it has.

Types of relationships

1. Unary -> Entities of only **one** entity set are involded in relationships.
2. Binary -> Entities of only **two** entity set are involded in relationships.
3. Ternary -> Entities of only **three** entity set are involded in relationships.



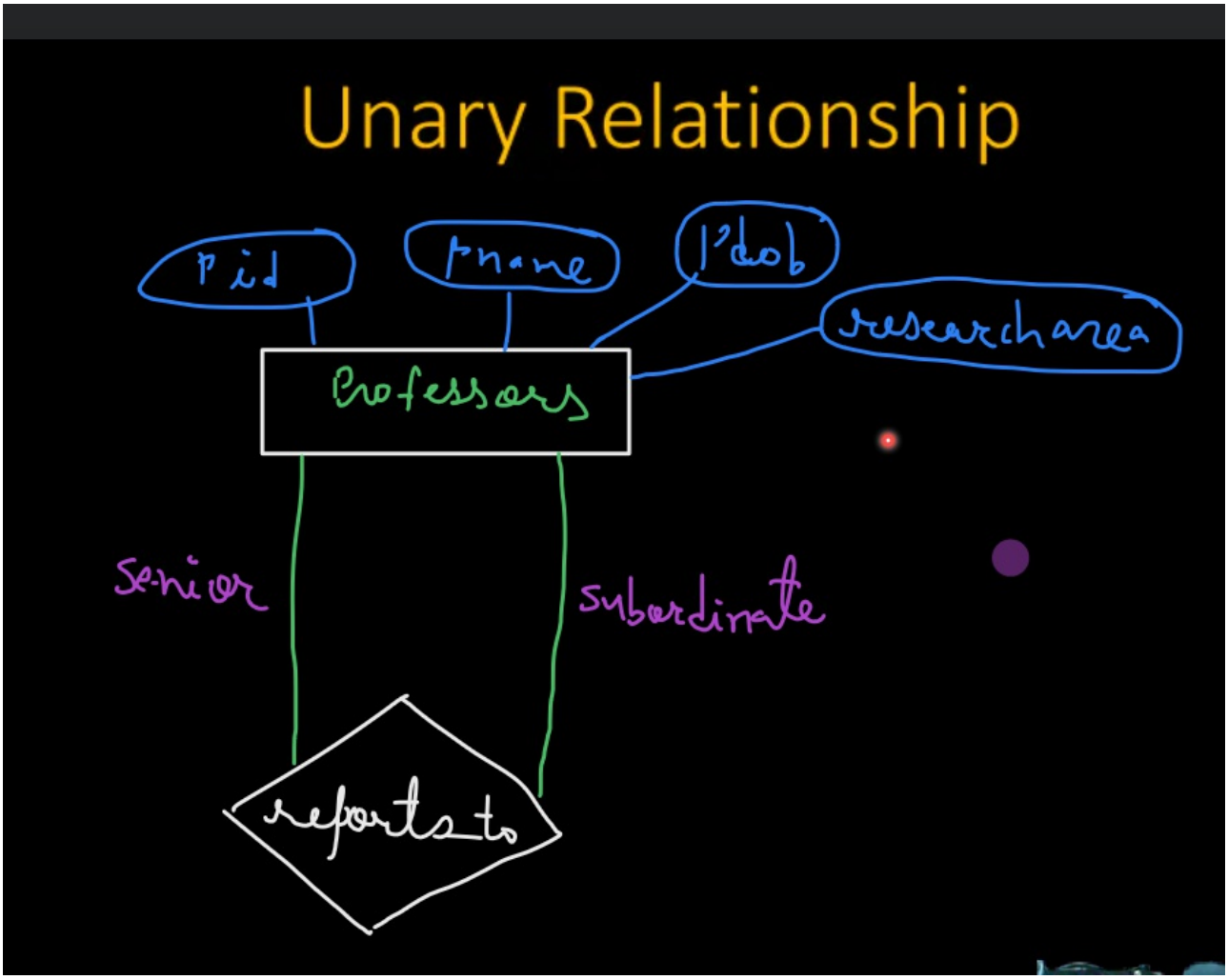
Unary relationship



- Professor reports to other professor.
- Relationship -> reports
- Senior
- Sub-ordinate professor reports to which professor?

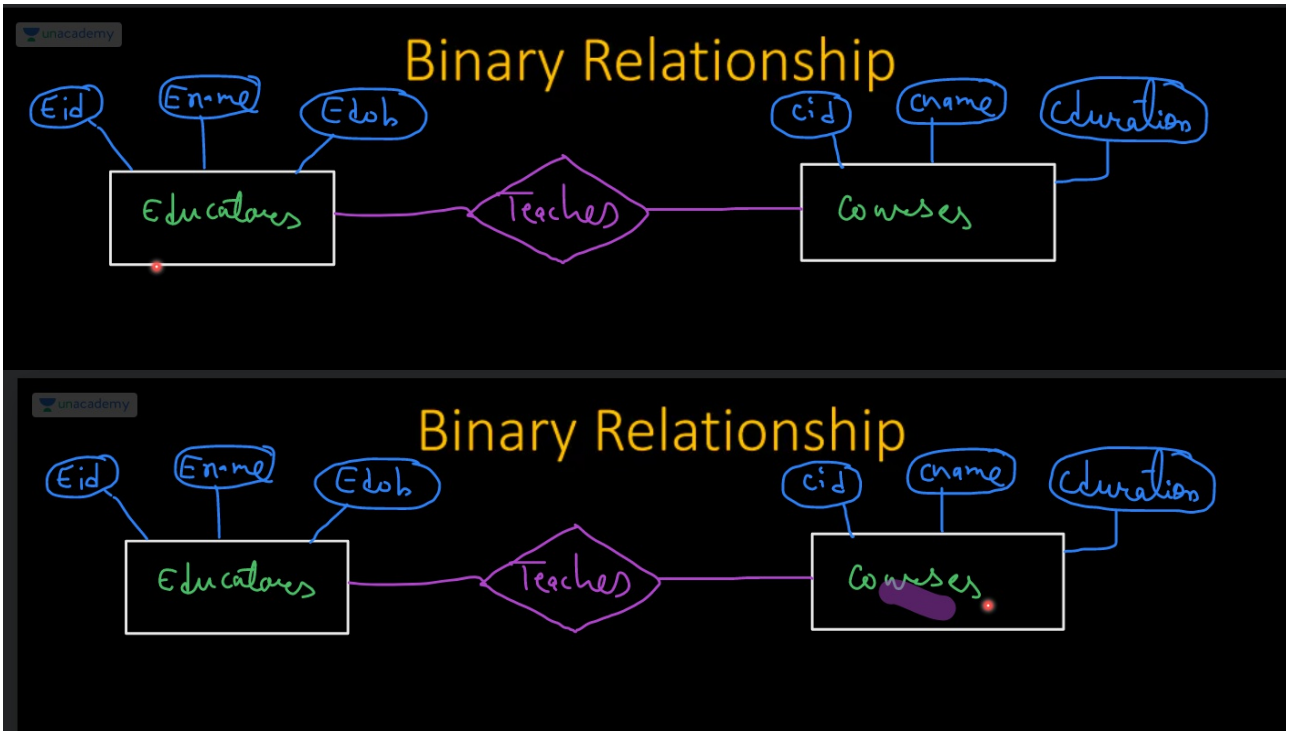
Senior professor.

Professor is connected to another professor in some relationship but the professor information is written in one table only.

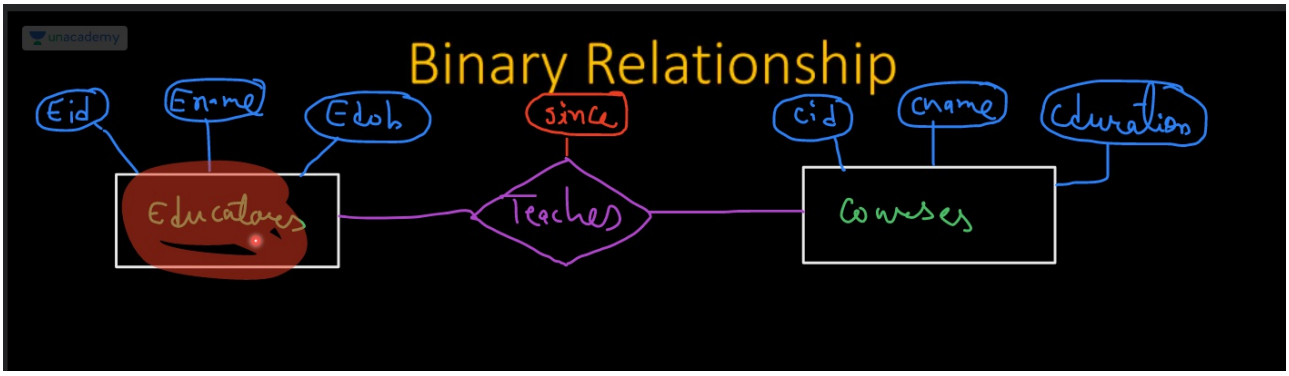


- One entity set and it creates **relationship** among the entities.

Binary relationship



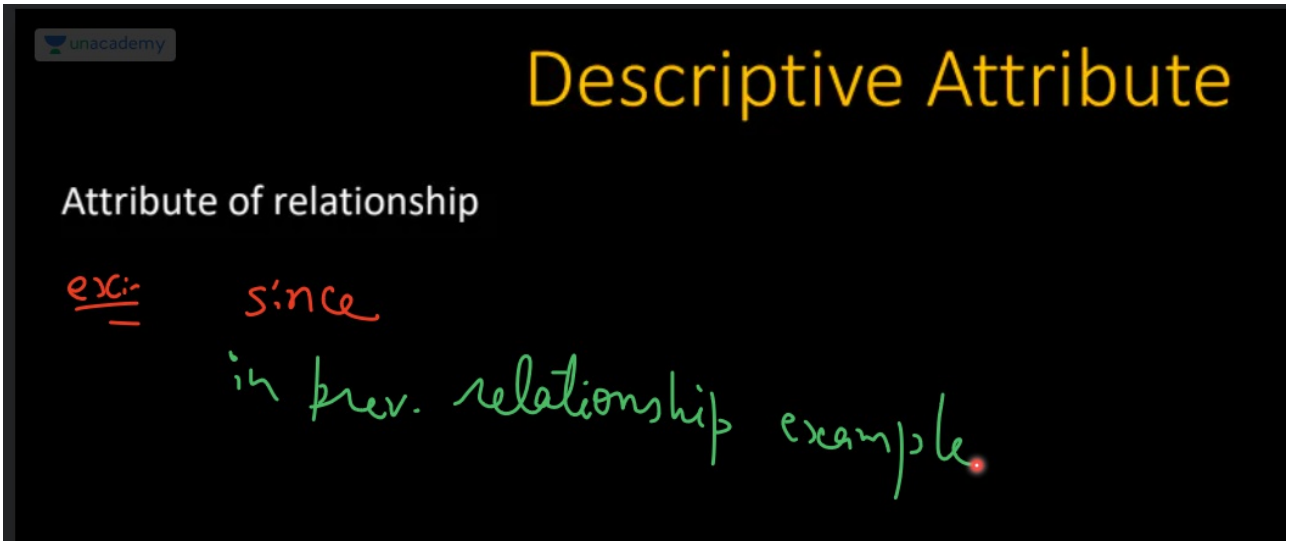
- Educators teaches courses.
- Many educators are teaching many different courses.
- Teaches -> Relationship set.
- Educator set's entities are participating in the **teaches** relationships with courses.
- Which educator teaches which courses.
- **Direction** is not needed, we can understand directly.
- Some attributes are there that come after a relationship is formed. They are called as **relationship set attributes**.
- A educator is teaching a course, from which time the educator is teaching the course.
- The information is **since**.
- The information/attribute of **since** is only valid when the relationship, **Educator teachers a course**, is **made/shown**.



- Vishvadeep sir teaches **COA** since two thousand 11(2011).
- Vishvadeep sir teaches **C-programming** since two thousand 7(2007).
- The **since** attribute has come into **existance**, **when the relationship**, **Educator teachers a course**, is **made**.
- Any attribute which is a part of the **relationship set attribute** then they are called as **descriptive attribute**.

Descriptive Attribut

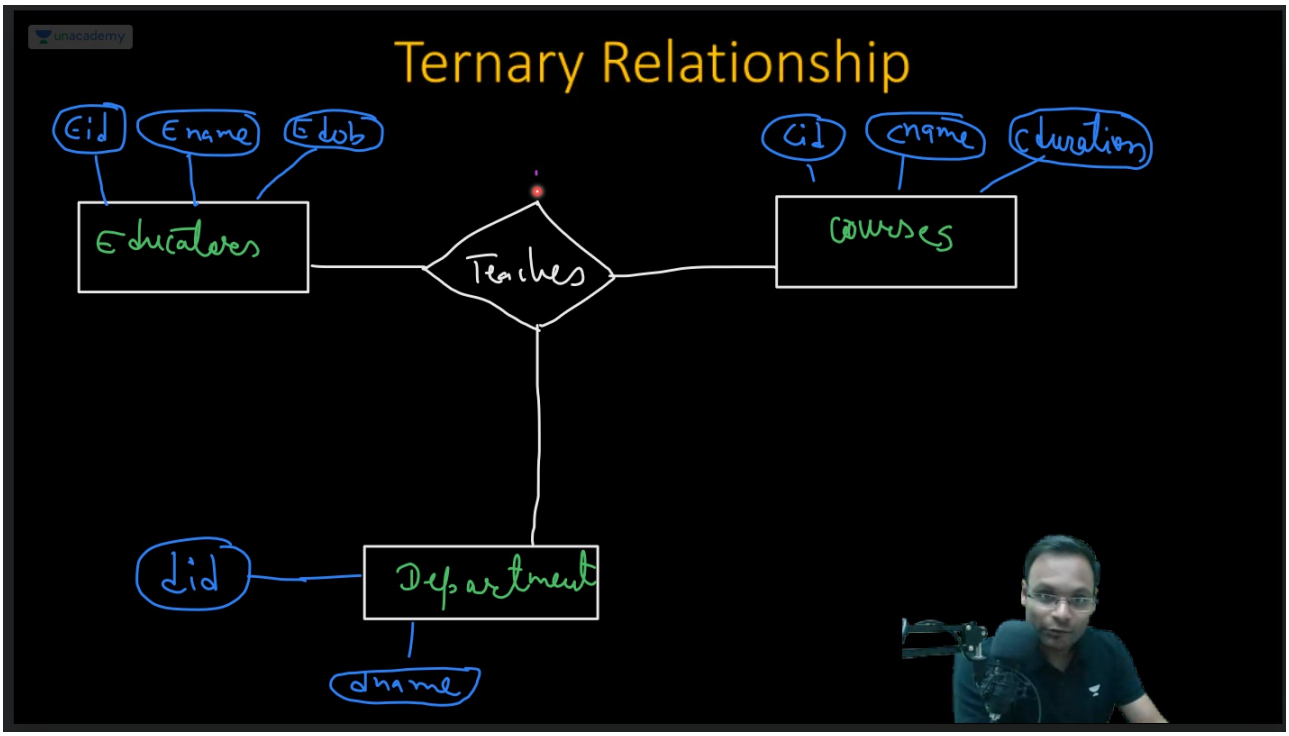
- Attribute of relationship
- **Since** in above relationship **example**.



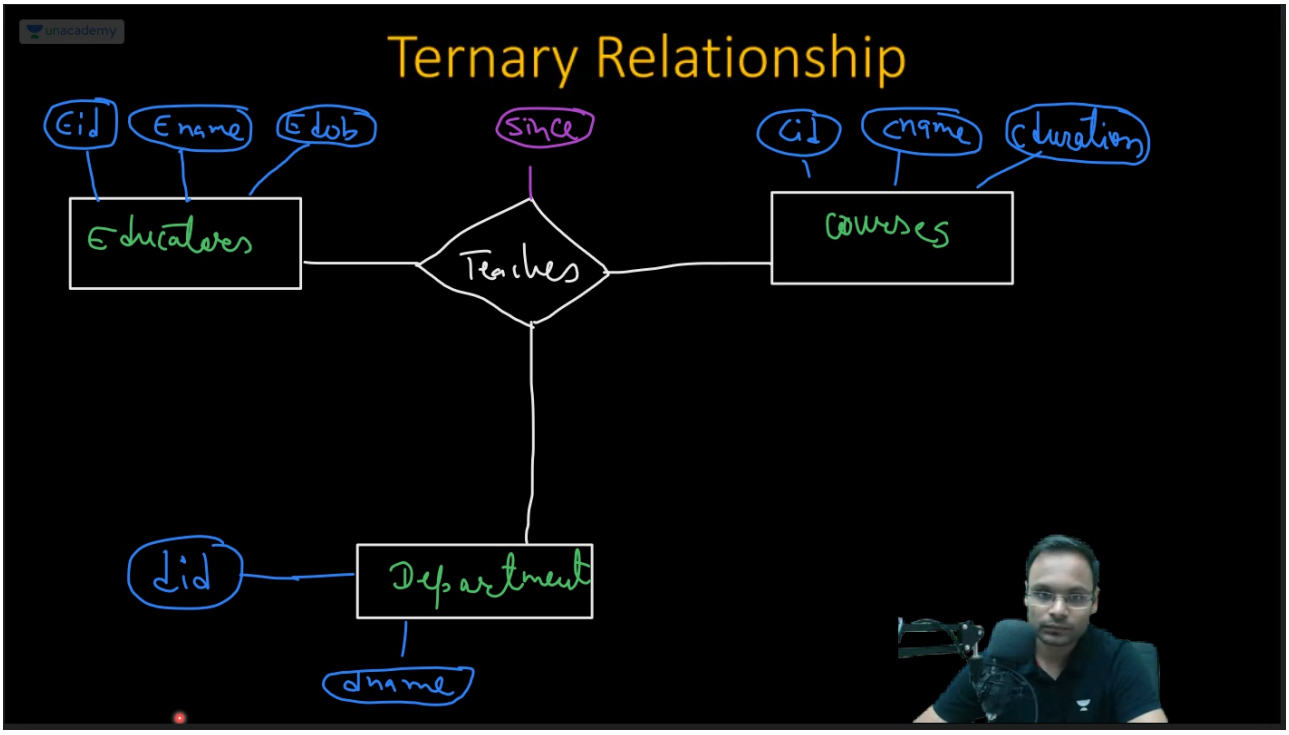
- There is no existance of **since** until the relationship between the **educator teaches courses**, shows.
- If relationship is there then only **since** will come otherwise not.

Ternary Relationship

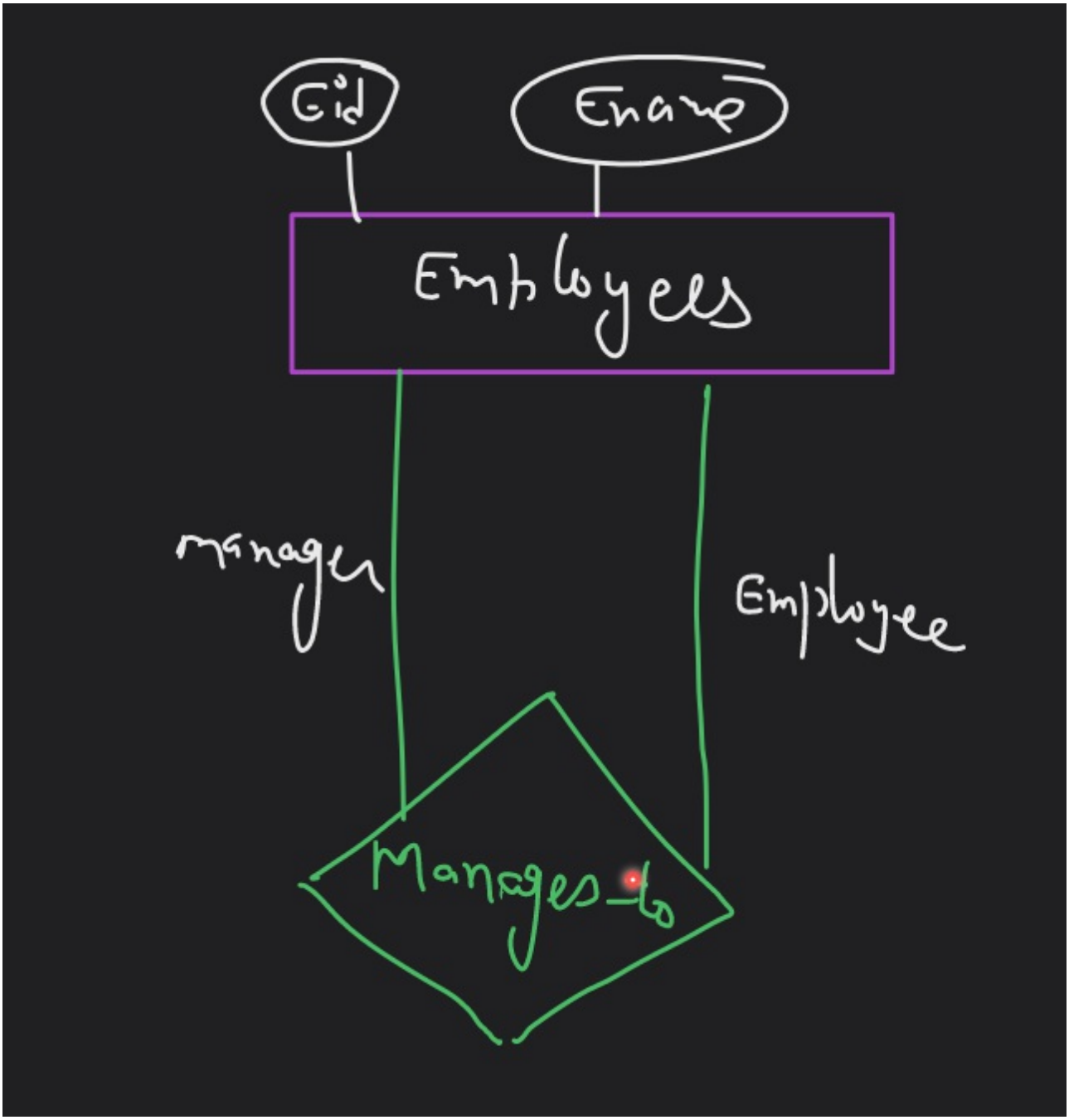
- Relationship Set -> Teaches
- 3 Entity sets are involved.
- Vishvadeep teaches COA in CSE department.



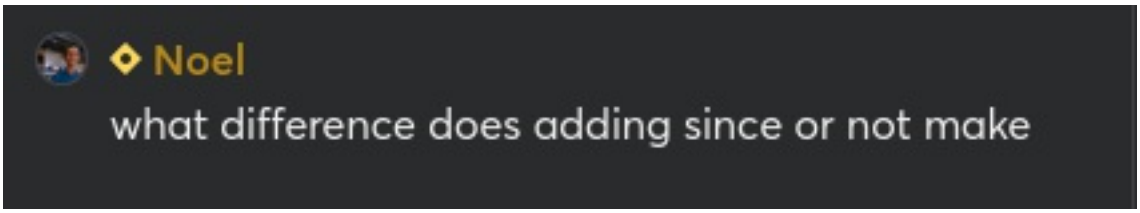
- Vishvadeep teaches COA in ESE department since 2016.
- Vishvadeep teaches COA in Electrical department since 2018.



- If **three** entity sets are involved then it is a **ternary relationship**.
- We can use **descriptive attributes** in **uninary relationship** as well.



- **Uninary relationship**



- If we don't put **since** then we will not know from what time that educator is teaching that course.
- We want to store some information but we don't know if it will be educator's info or the courses's info. The information is there because of the **relationship**.
- Usually we read the ER diagram from **left to right** only.

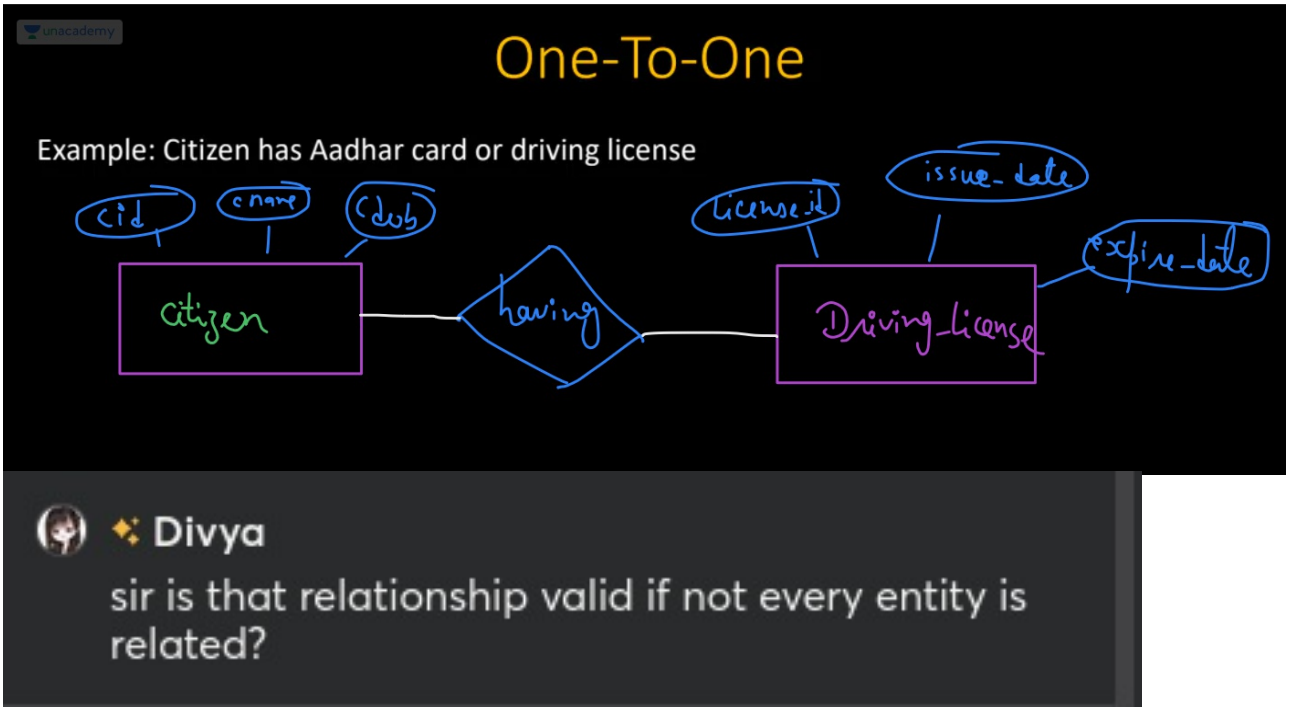
Mapping cardinality

- If two entity sets are in **relationship** then how many entities from the two entity sets are forming/showing one **relationship**.

1. One to One
2. One to Many
3. Many to One
4. Many to Many

One-To-One

- One **cititzen** can **have** only one **driving license**.
- One **driving license** can **have** only one **cititzen**.
- Citizen having driving-license.
- One To One relationship.



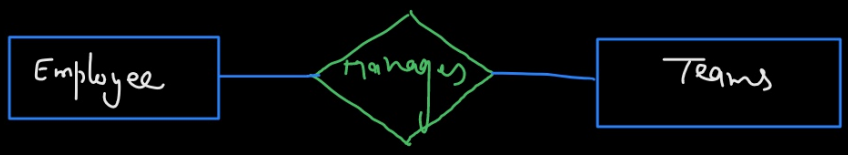
- It is not necessary that all of the entities will participate in the **relationship**.

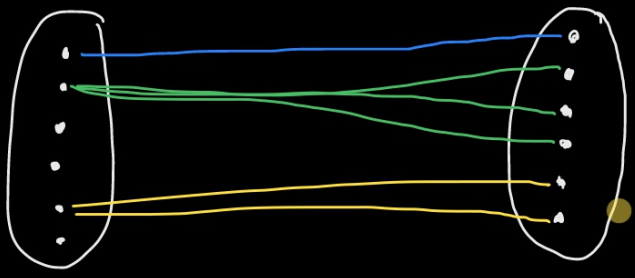
One-To-Many

- One entity of **employee** can have **relationships** with **multiple entities** of **teams**.
- One employee can manage one team.
- One emplyee can manage **more than one team**.

One-To-Many

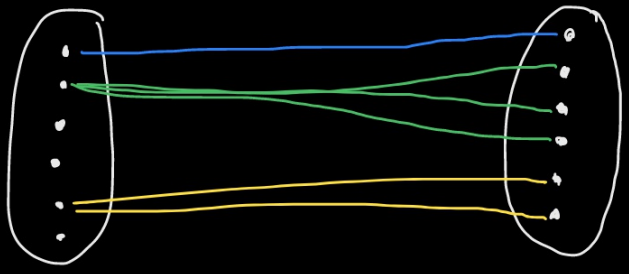
Example: Employee manages teams





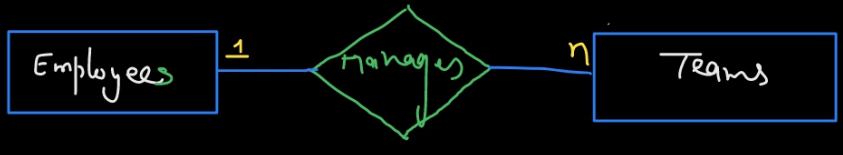
One-To-Many

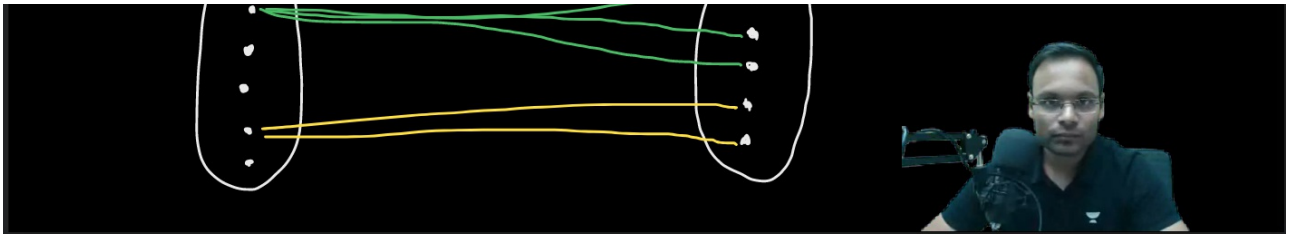
Example: Employee manages teams



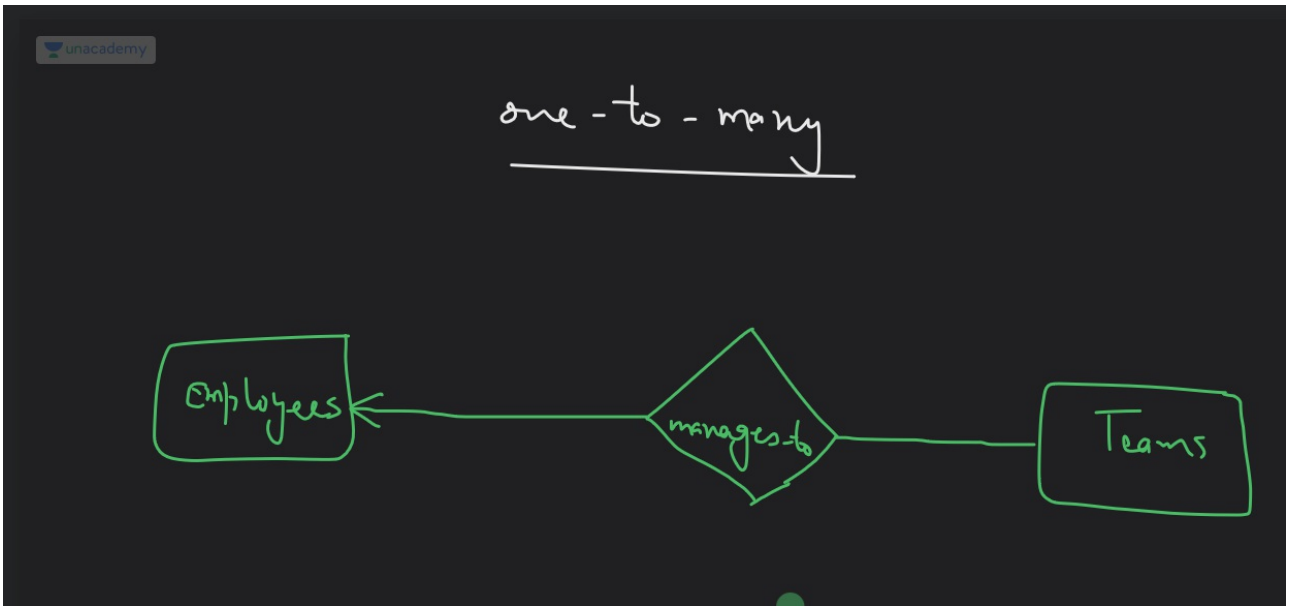
One-To-Many

Example: Employee manages teams



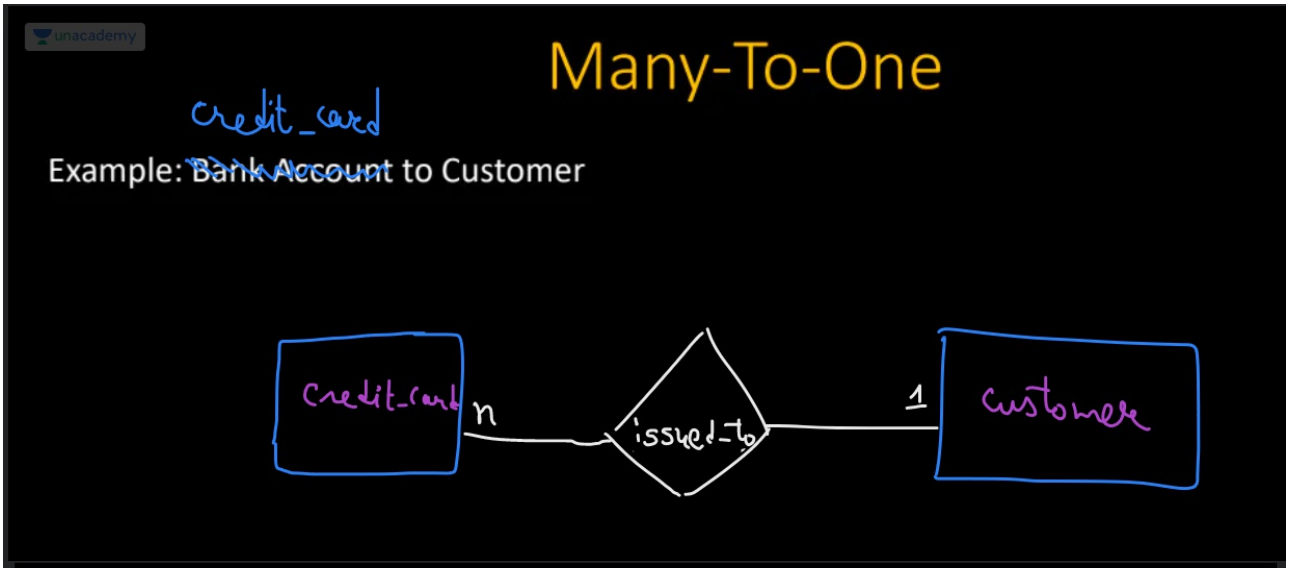


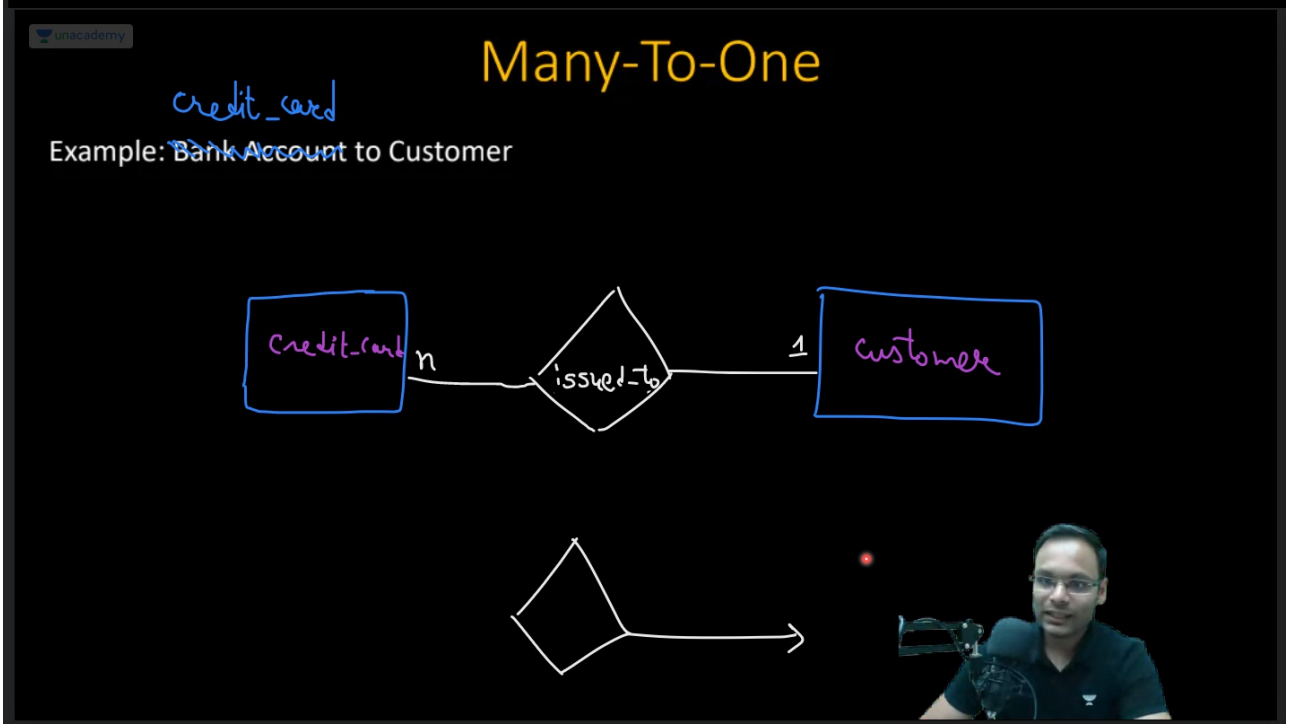
- If one entity in the relationship is showing relationship of more than one on the right side then it is **One to Many** relationship.
- Towards the **one** side put an **arrow**. The other side is the **many**.



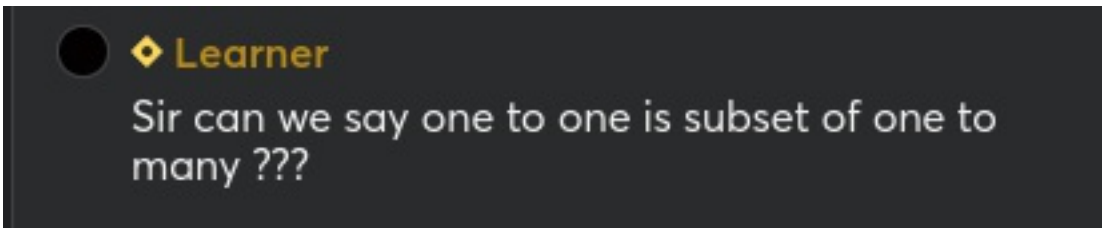
- Another way to represent **One to Many**.

Many-To-one





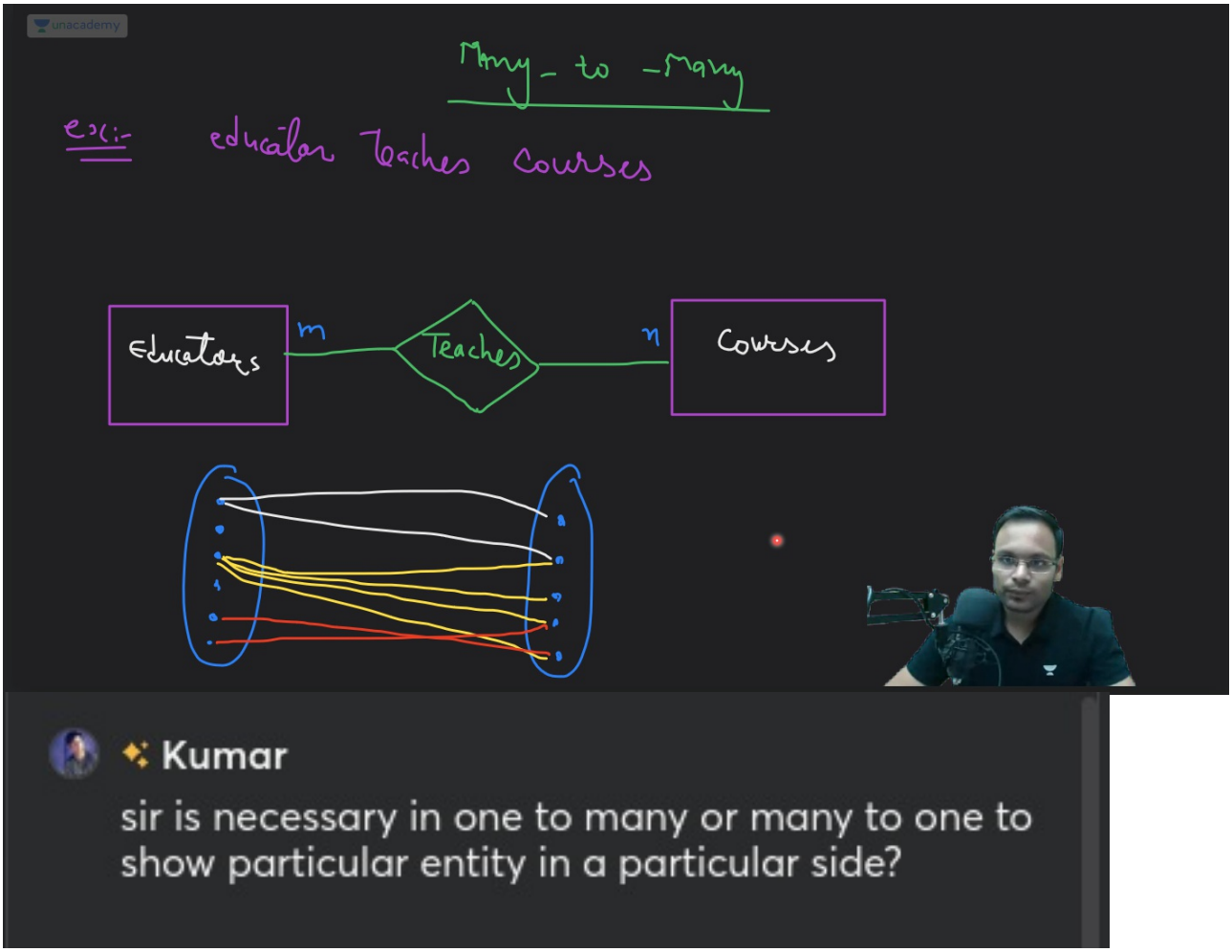
- Mirror/opposite of **One to Many**.



- No.

Many to Many

- Multiple educator can teach one course
- One educator can teach multiple courses.

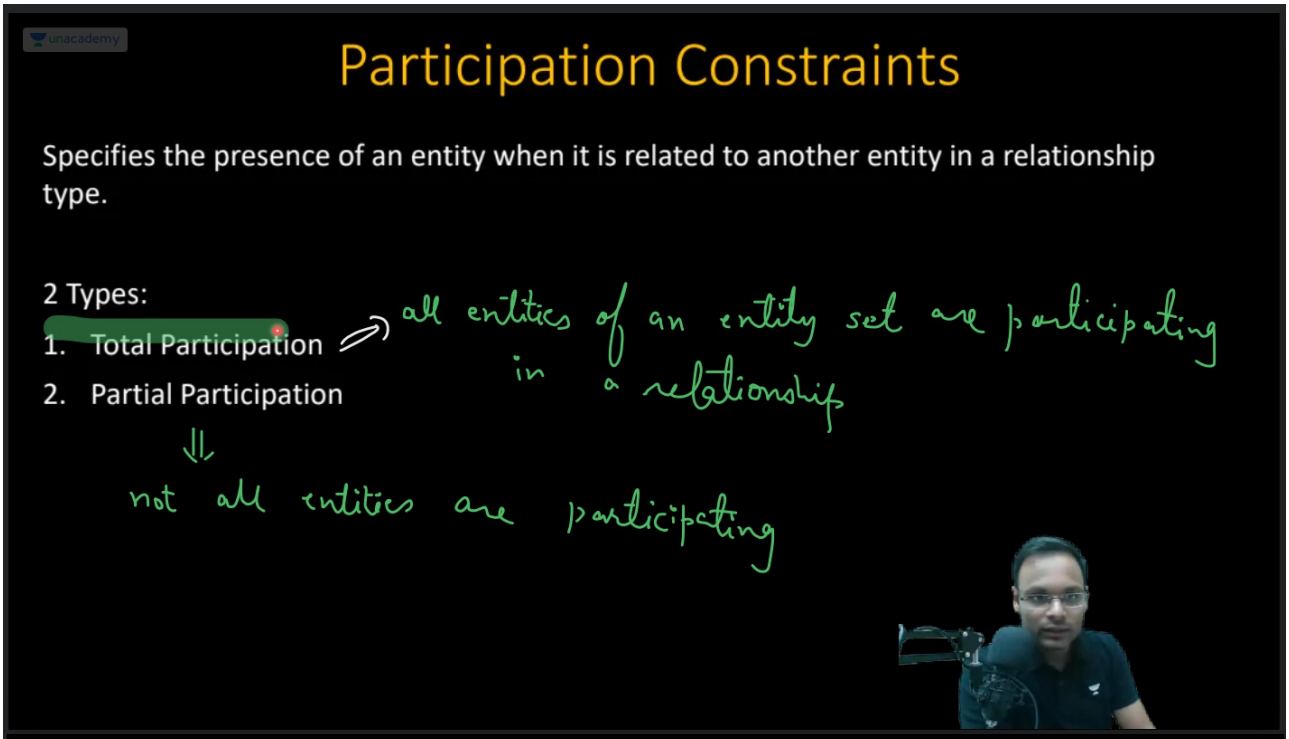


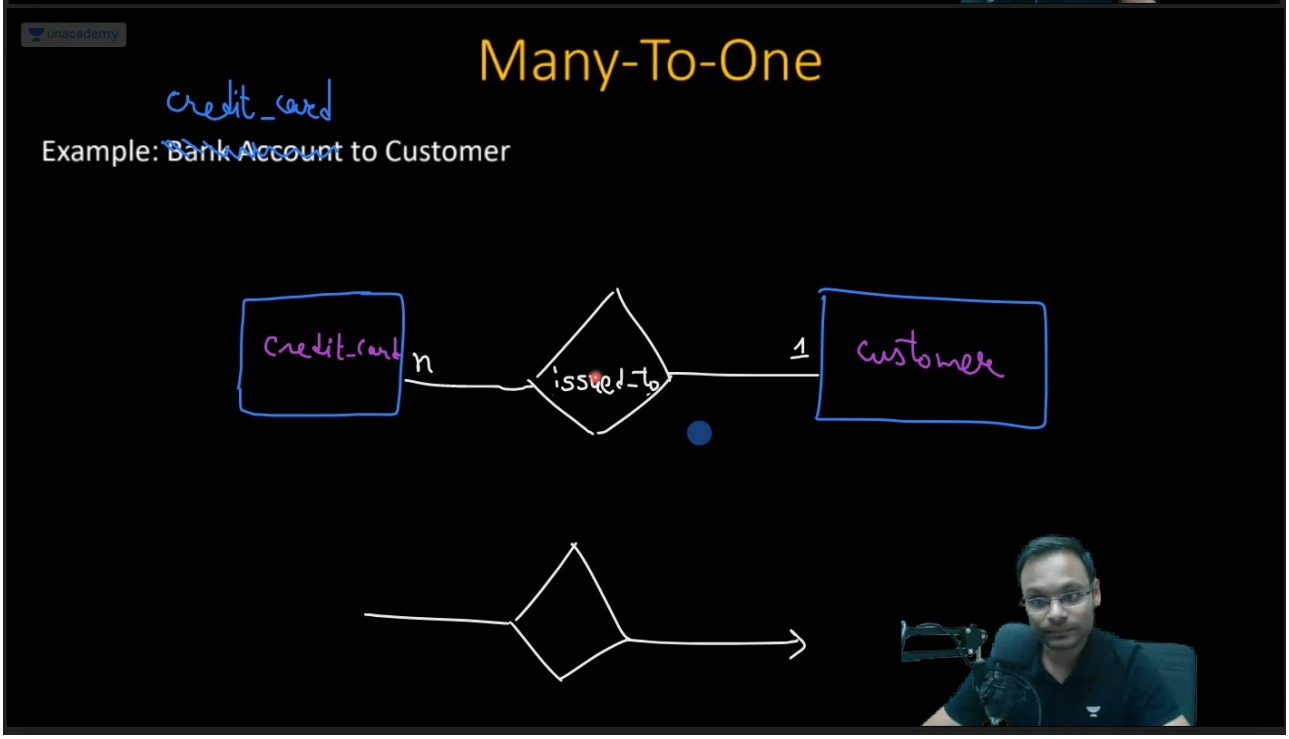
- No.
- It is not necessary to show the **entity** but we will show the **entity set's relationship** that's it.

Participation Constraints

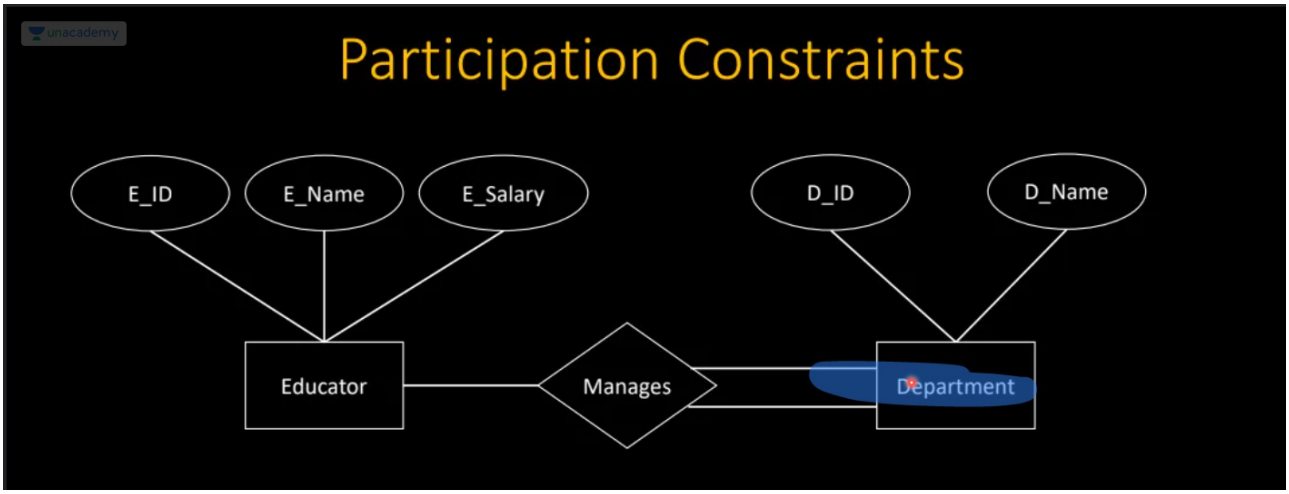
- Specifies the presence of an entity when it is related to another entity in a relationship type.

1. Total participation -> All entities of an entity set are particiating in a relationship
2. Partial participation -> Not all entities are participating.

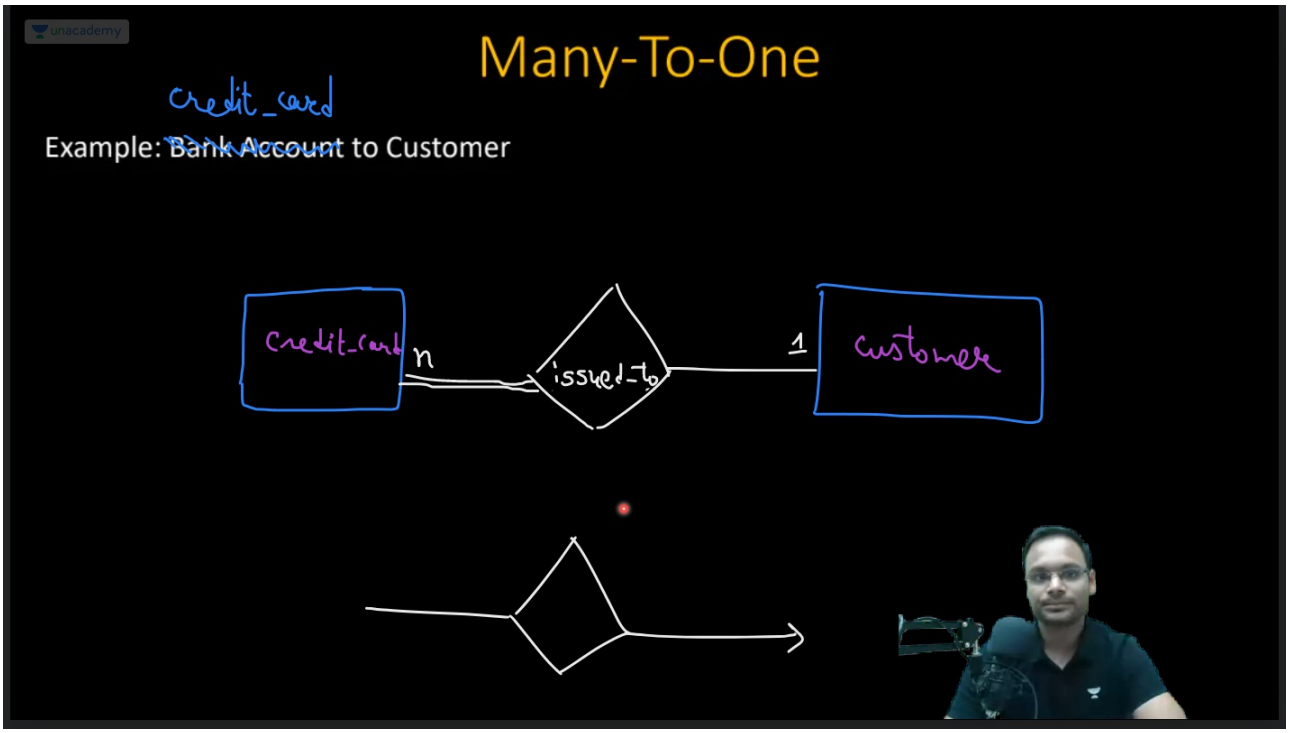




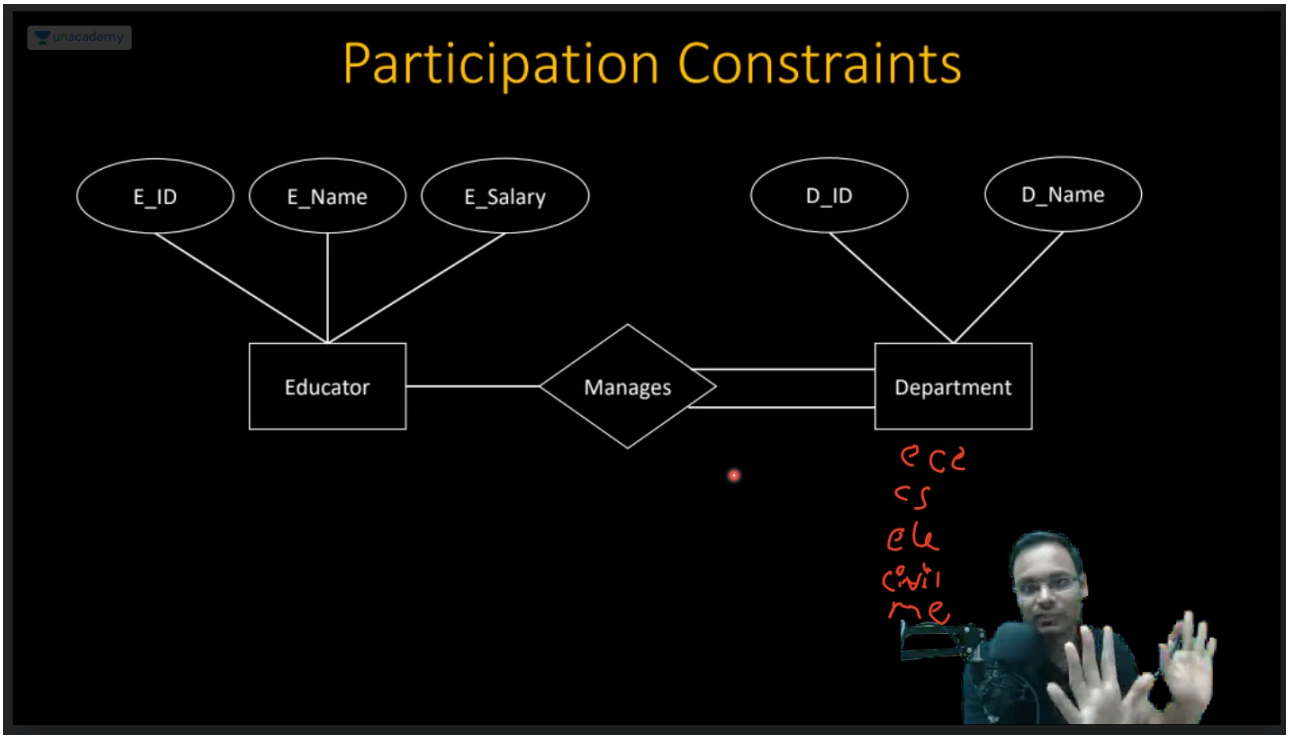
- Customer's particiation will be **partial** here.
- Credit card's particiation will be **total** here.



- Where there is **total particiation**, we will put **double lines** on that side.

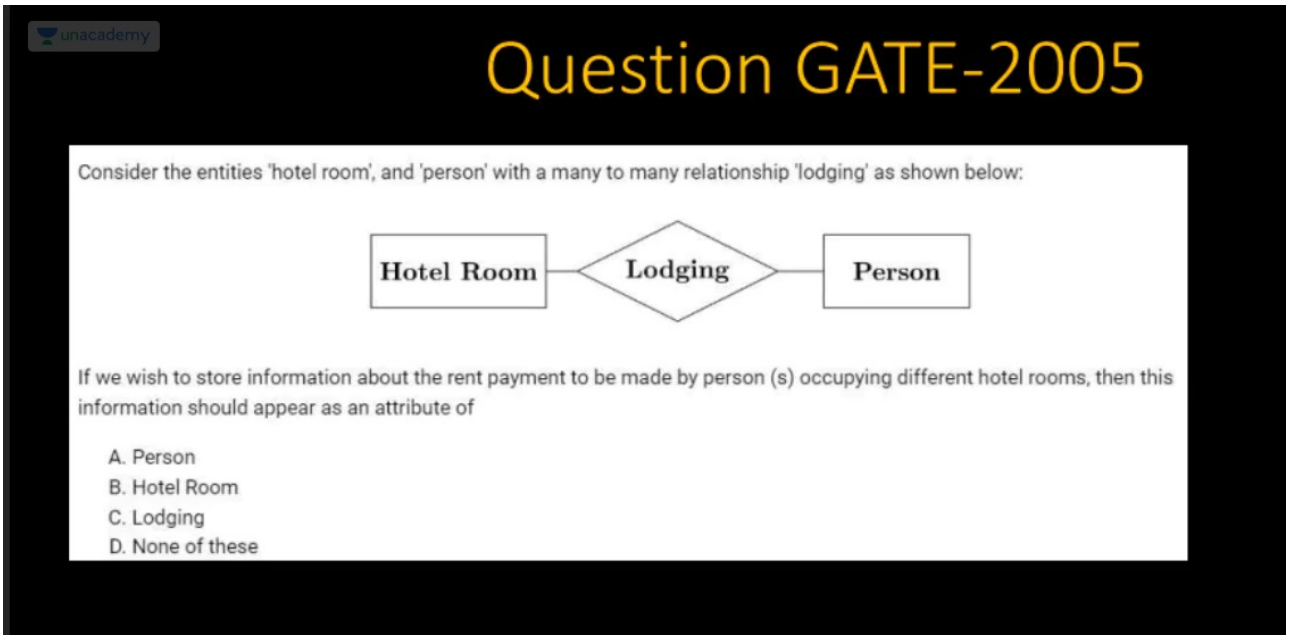


- Example.



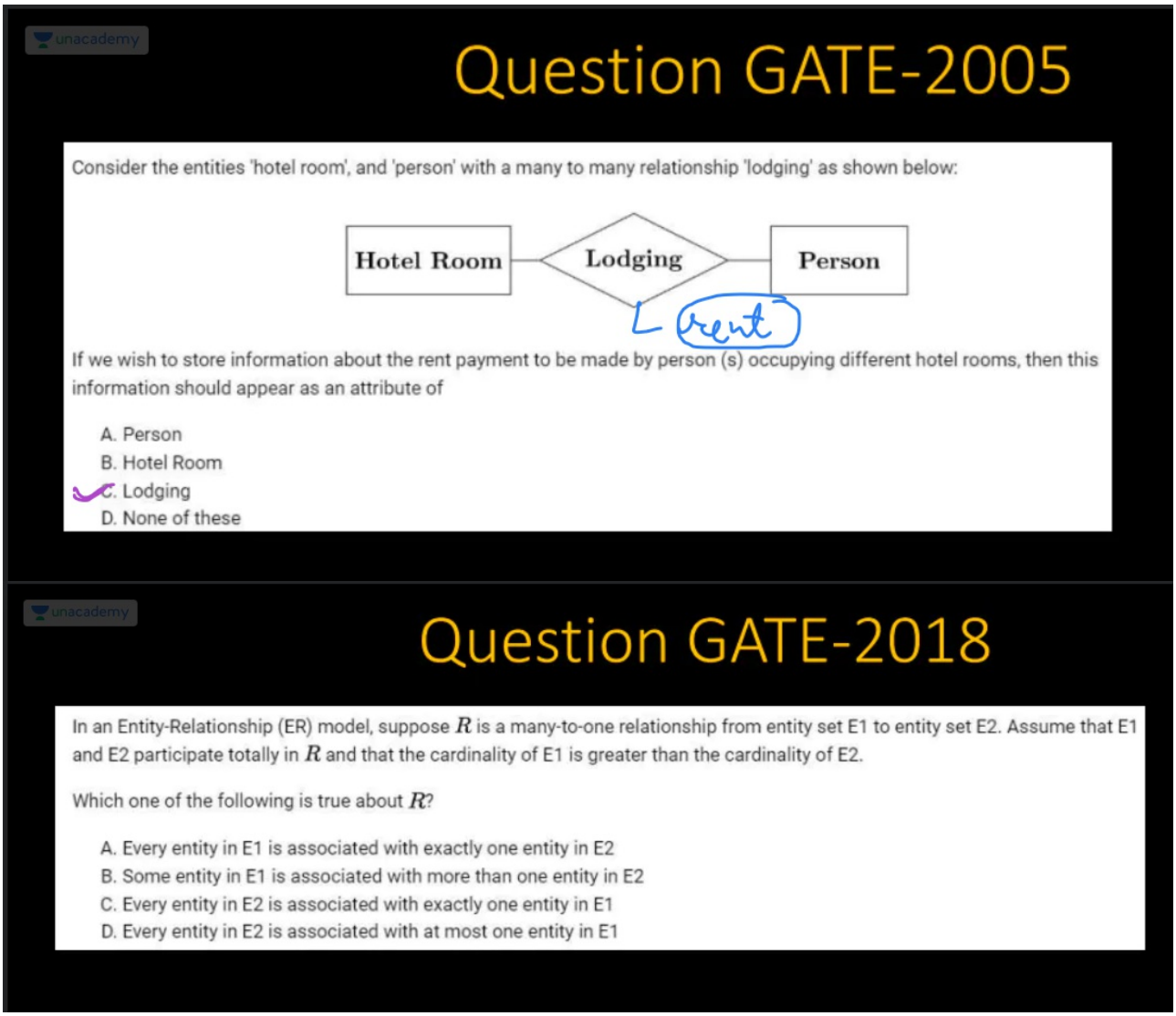
- **Departments** participation will be **total particiation** as every department must have a manager.
- **Educator's** participation will be **partial particiation** as every educator will not manage a department, not necessarily.

Questions



- Question
- Option C.
- **Rent payment** is not soly on **Person** because it is dependent on what room the person has booked. It is not solely on the **Hotel room** as well because the **rent price** of the **hotel rooms** chance from person to person.
- So, **Rent payment** is an **description attribute** which is possible because of the

relationship between the **Hotel room and Person**.



- Question.
- Option **A**.

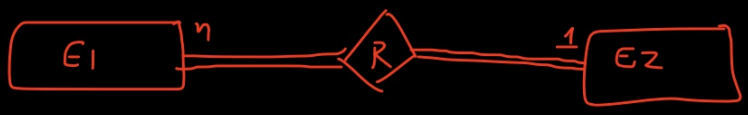
Question GATE-2018

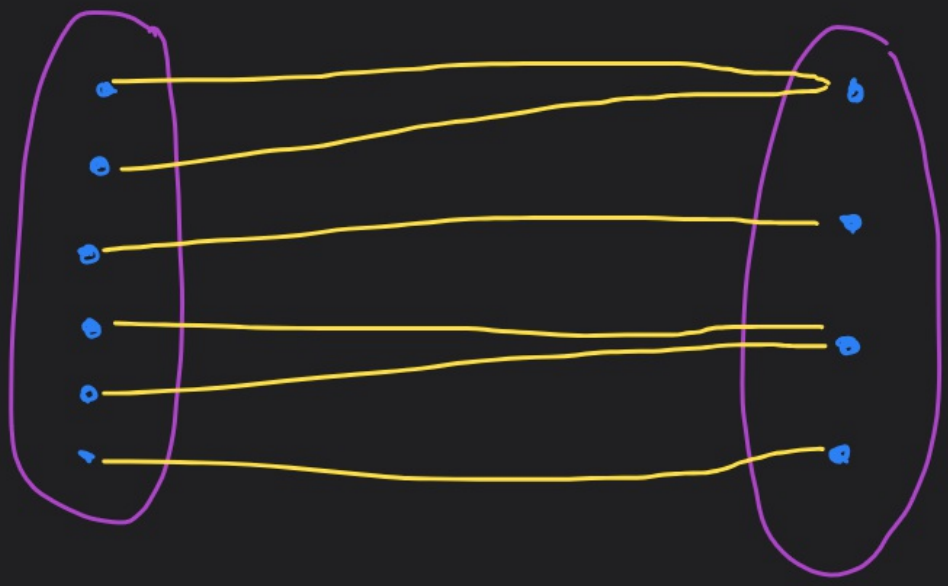
In an Entity-Relationship (ER) model, suppose R is a many-to-one relationship from entity set E1 to entity set E2. Assume that E1 and E2 participate totally in R and that the cardinality of E1 is greater than the cardinality of E2.

↳ no. of entities in entity set

Which one of the following is true about R?

A. Every entity in E1 is associated with exactly one entity in E2
B. Some entity in E1 is associated with more than one entity in E2
C. Every entity in E2 is associated with exactly one entity in E1
D. Every entity in E2 is associated with at most one entity in E1





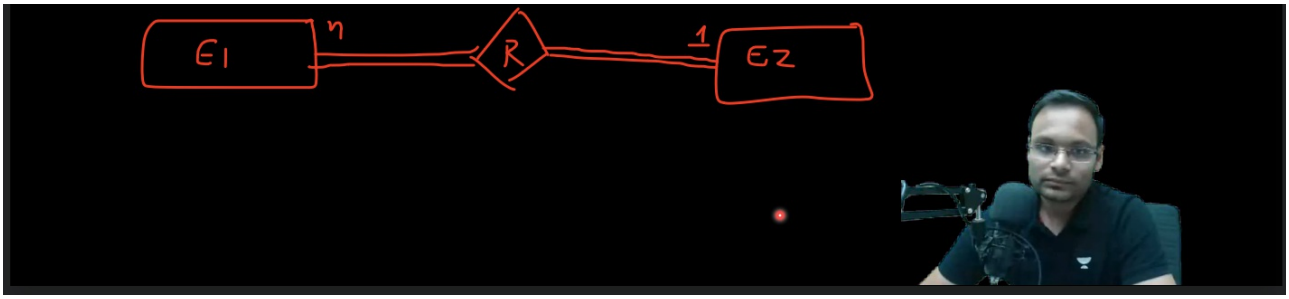
Question GATE-2018

In an Entity-Relationship (ER) model, suppose R is a many-to-one relationship from entity set E1 to entity set E2. Assume that E1 and E2 participate totally in R and that the cardinality of E1 is greater than the cardinality of E2.

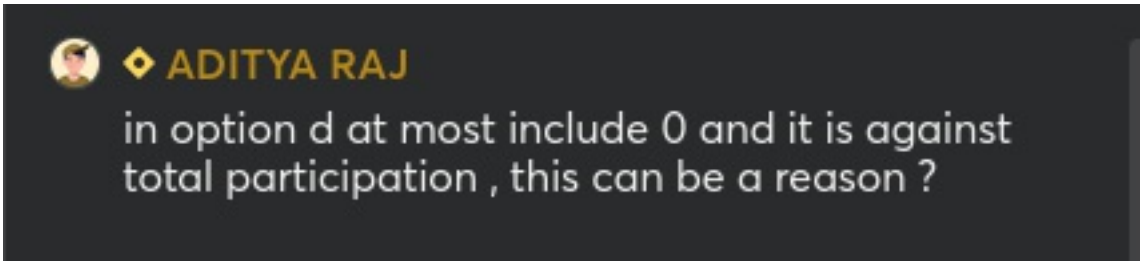
↳ no. of entities in entity set

Which one of the following is true about R?

A. Every entity in E1 is associated with exactly one entity in E2
B. Some entity in E1 is associated with more than one entity in E2
C. Every entity in E2 is associated with exactly one entity in E1
D. Every entity in E2 is associated with at most one entity in E1



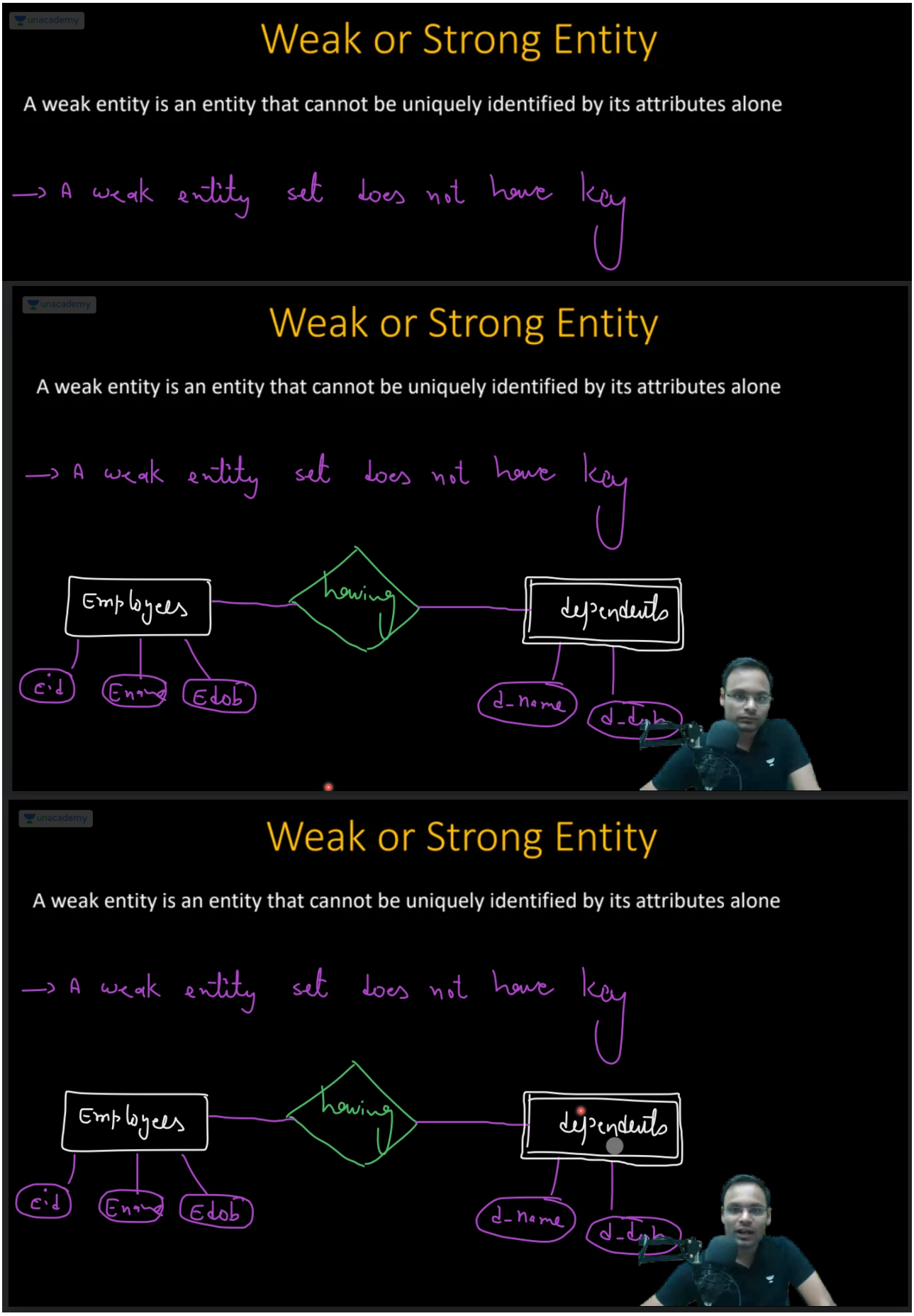
- Drawing the **set diagram** will make the questions easier to solve.



- AND more than one also there.

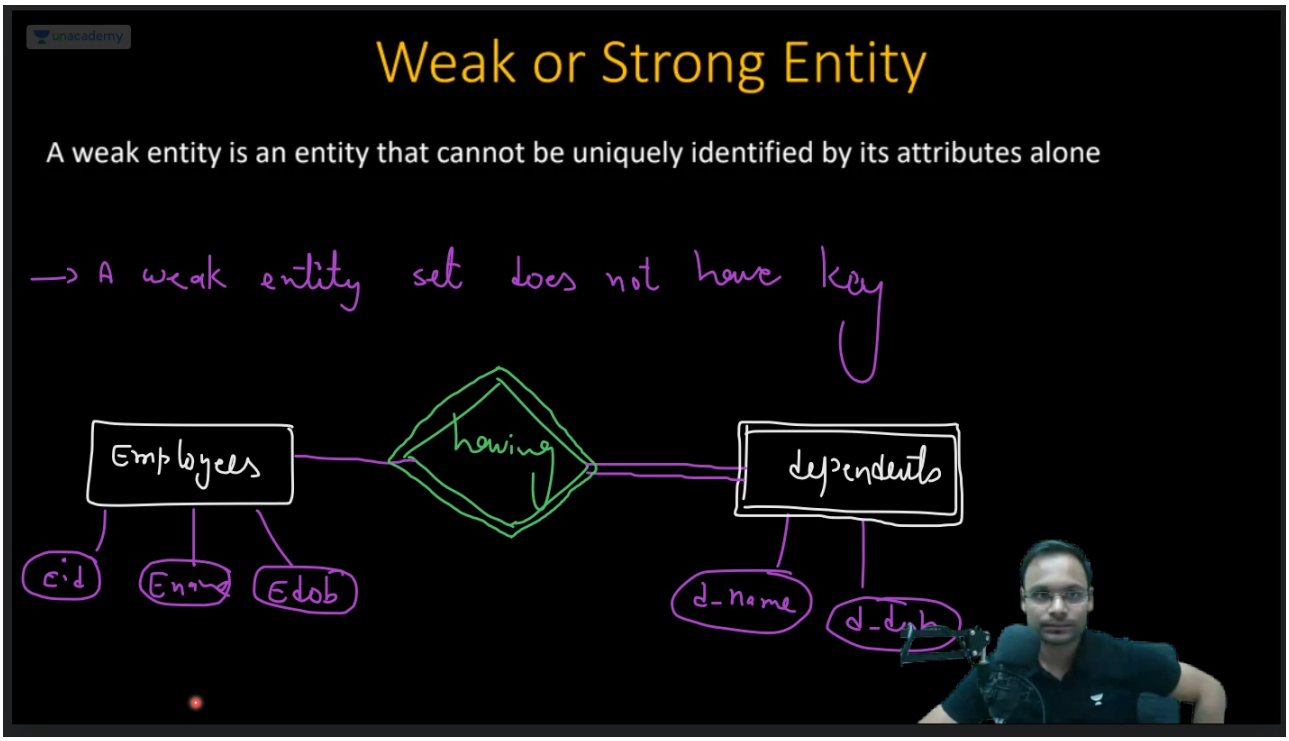
Weak or Strong Entity

- A weak entity is an entity that cannot be uniquely identified by it's attributes alone.
- A weak entity set does not have **key**.

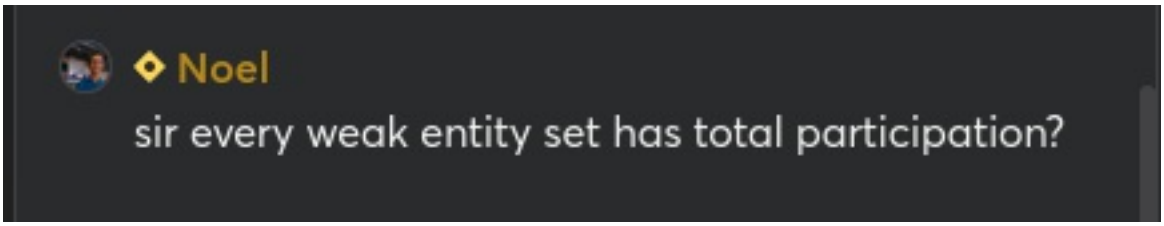


- Weak entity set is marked with **double rectangular box** around it.

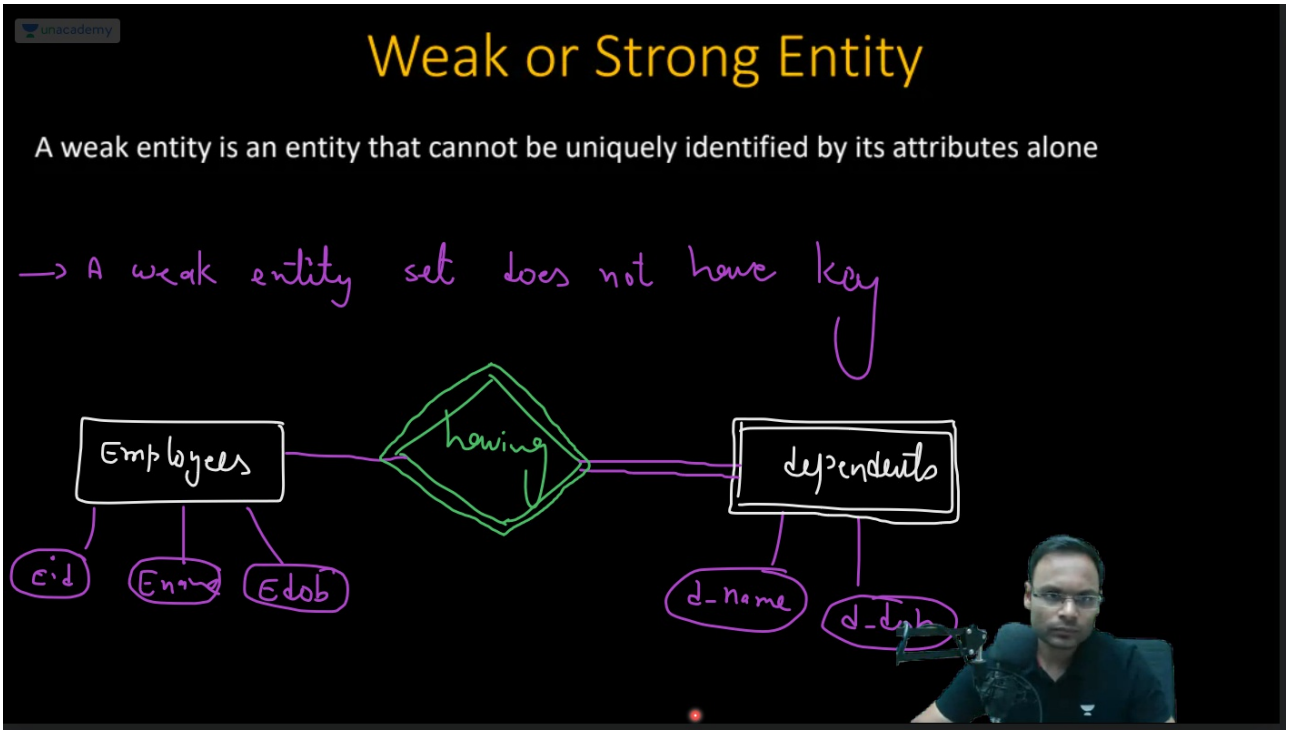
- Every **dependent** doesn't need to be given a **new ID**. It doesn't have it's own **ID**.
- Two weak entity set cannot make a **relationship**.
- One is a strong entity set and one is a weak entity set then the **relationship** is **weak**.



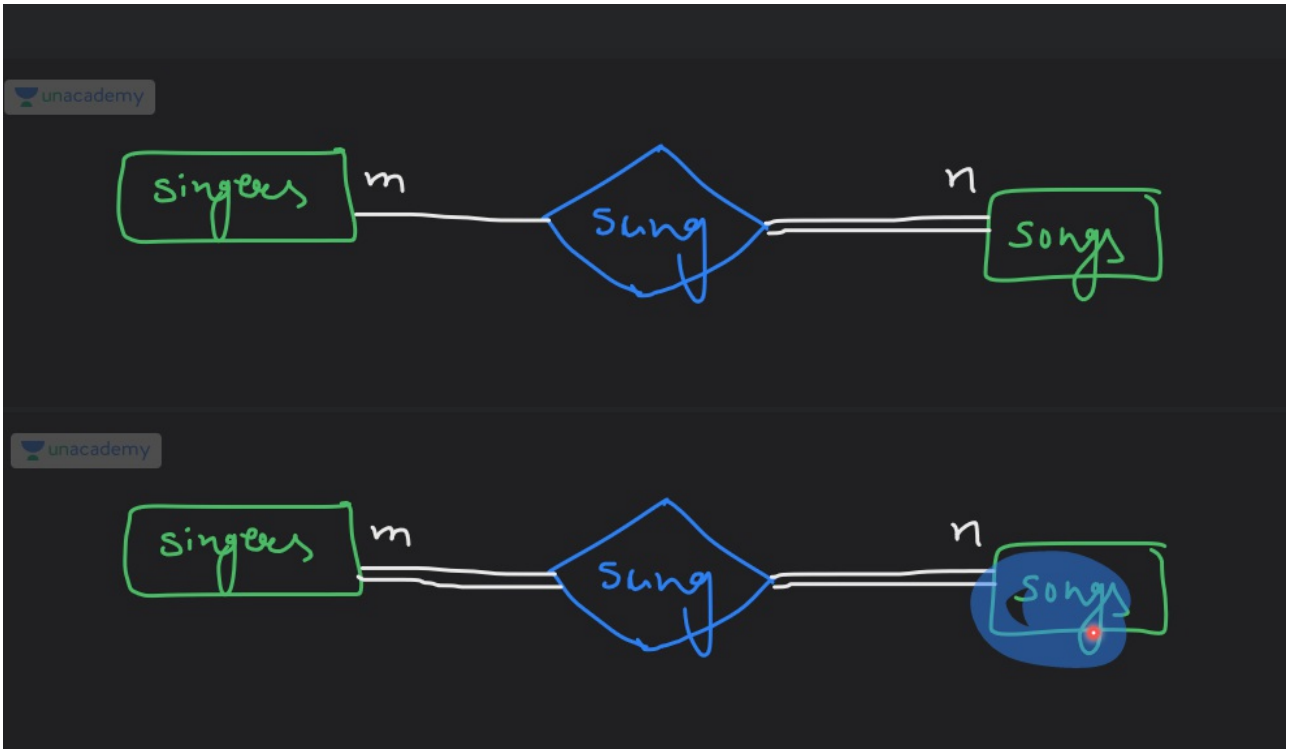
- **Weak relationship** -> Double Diamonds box.



- True.

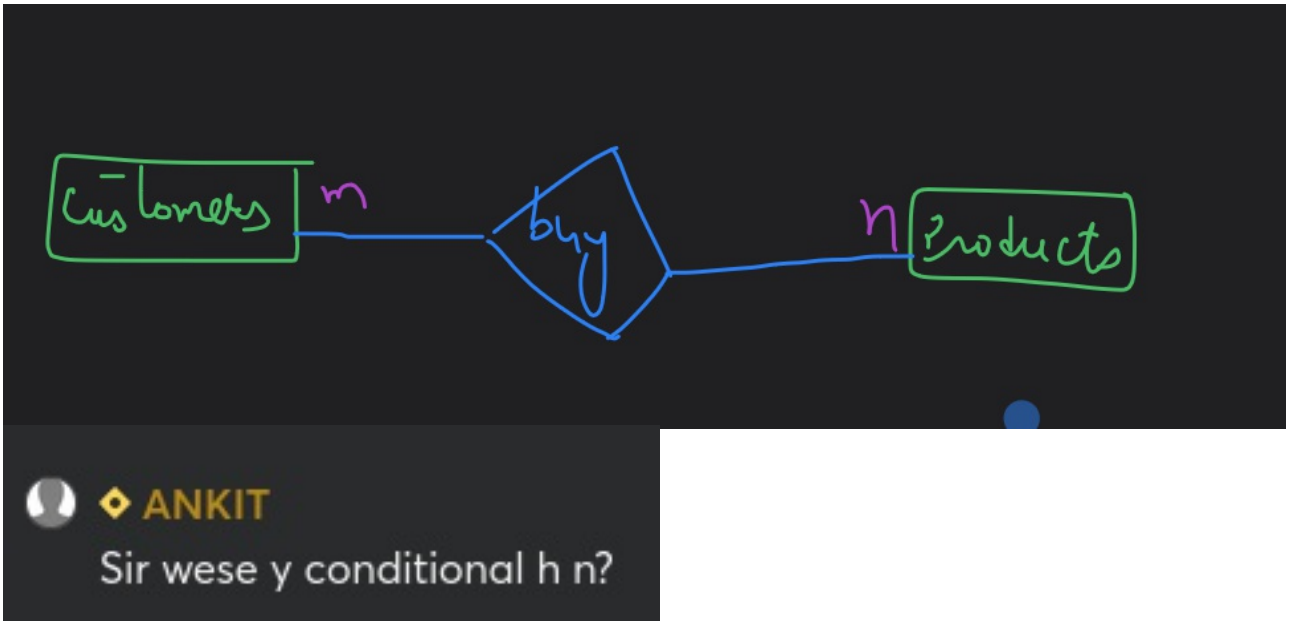


- One Singer can sing many songs.
- In one song many singers possible
- So, **Many to Many** relationship.
- Songs has **total participation**, as songs cannot be created without singers.
- Singer has **total participation**, as singer has sung songs that's why he is a singer.

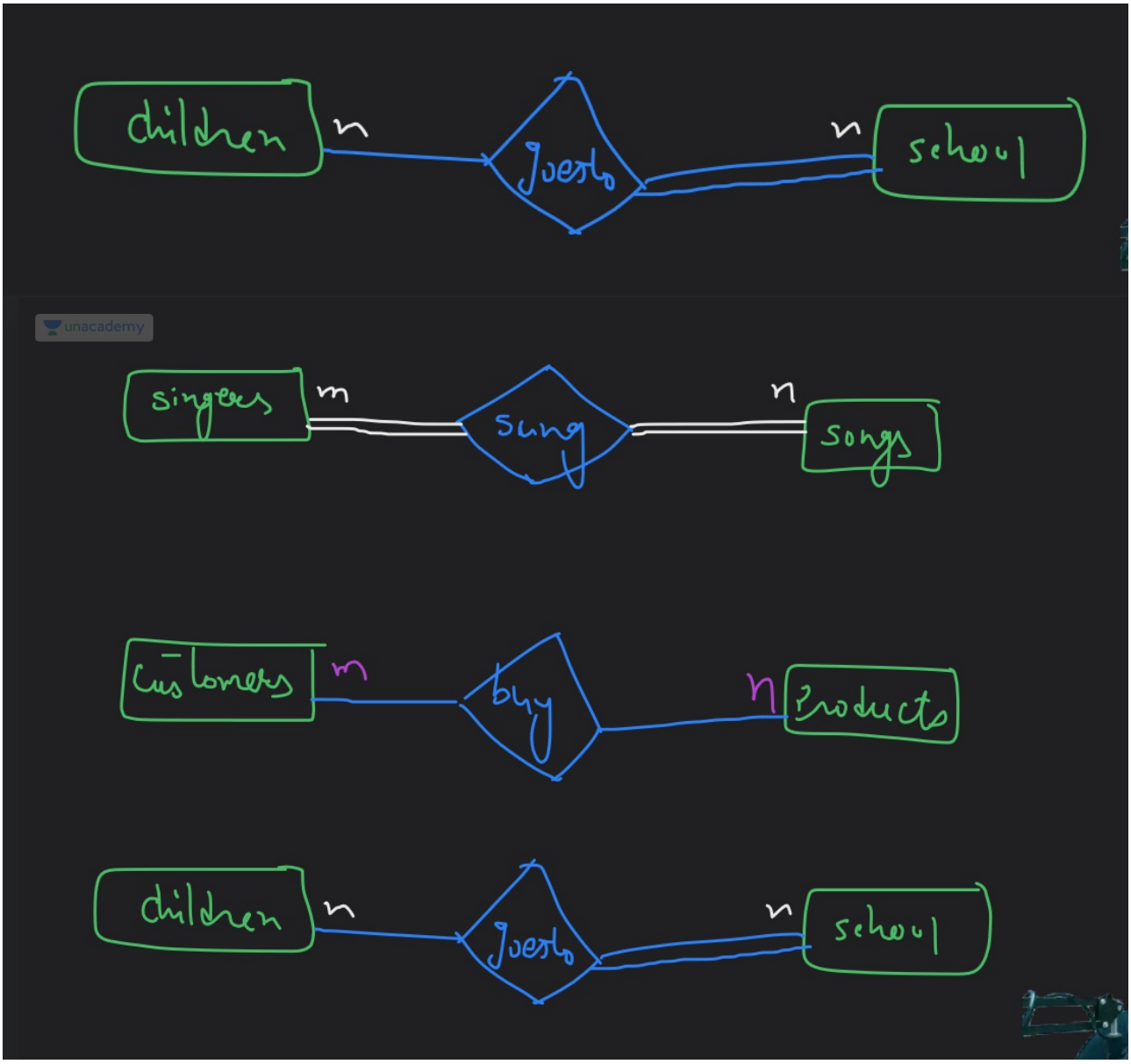


- One customer can buy many products
- One product can be bought by many customers
- That's why **many to many**.
- Not necessary that all customers buys products, that's why **partial participation** in **customers** entity set.

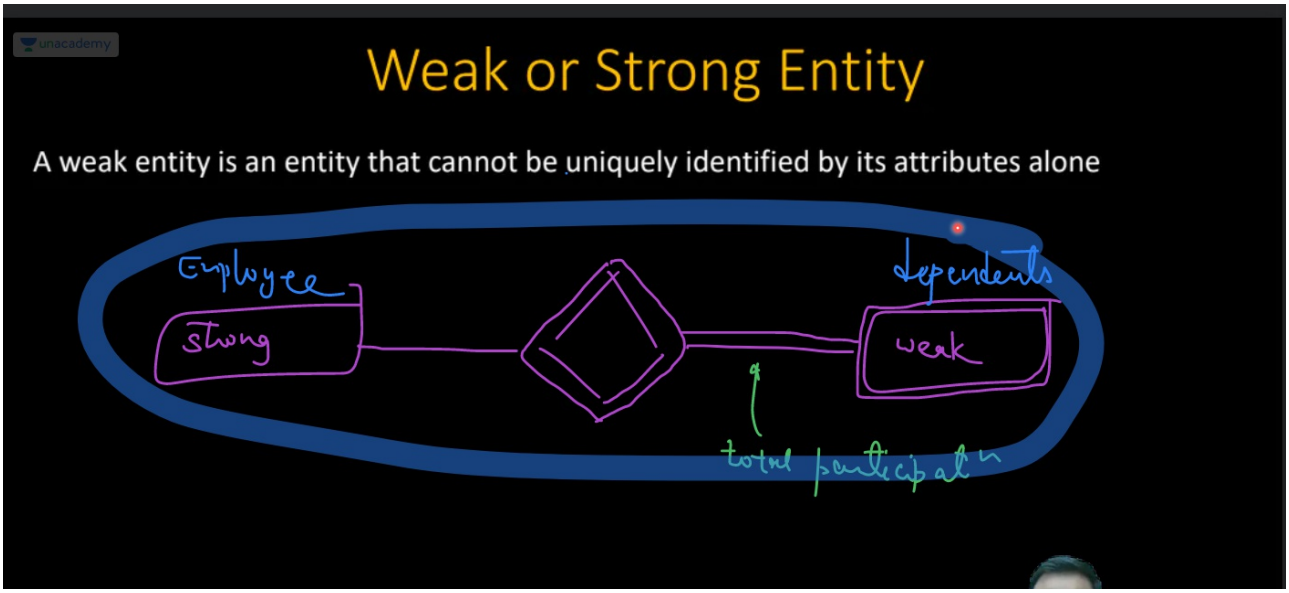
- Not necessary that all products are bought by customers, that's why **partial participation** in **products** entity set.



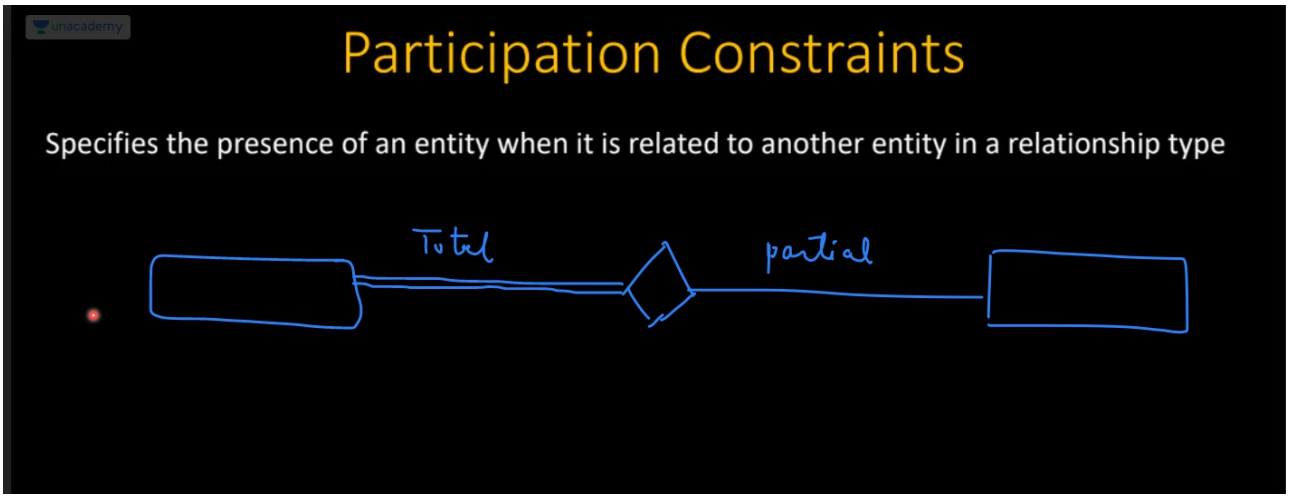
- Yes.
- One children can go to many schools, different times(Secondary from one school, higher secondary from another school)
- One School can have many children.
- So, **many to many** relationship.
- Not necessary that all children go to school, that's why **partial participation** in **children** entity set.
- It is necessary that all schools have children, that's why **total participation** in **school** entity set.


children
n
goes to
n
school
unacademy
singers
m
sung
n
songs
Customers
m
buy
n
Products
children
n
goes to
n
school

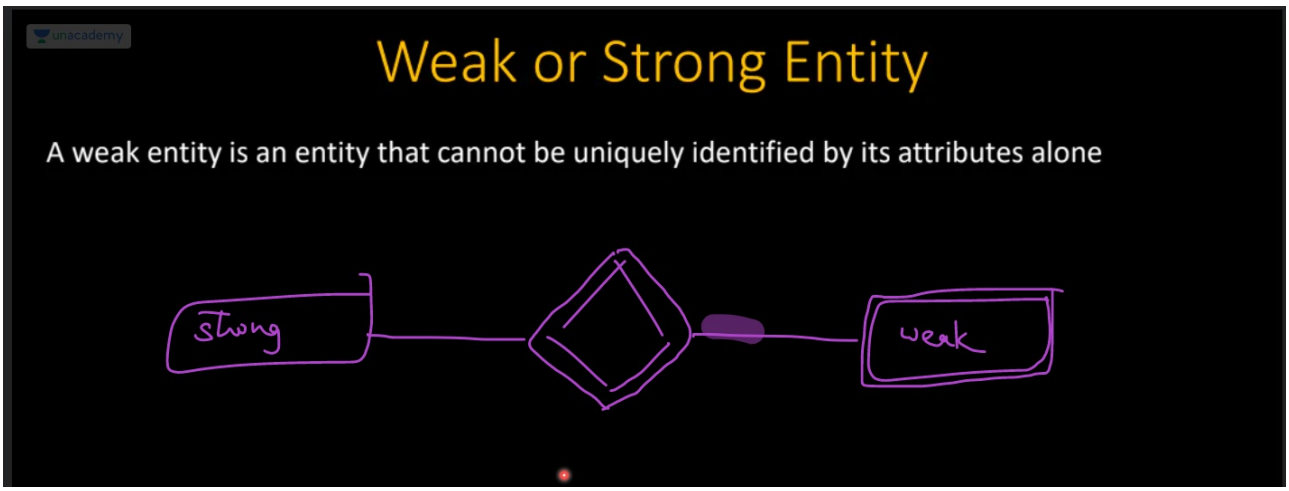
- Examples.


unacademy
Weak or Strong Entity
A weak entity is an entity that cannot be uniquely identified by its attributes alone
Employee
strong
dependents
weak
total participation

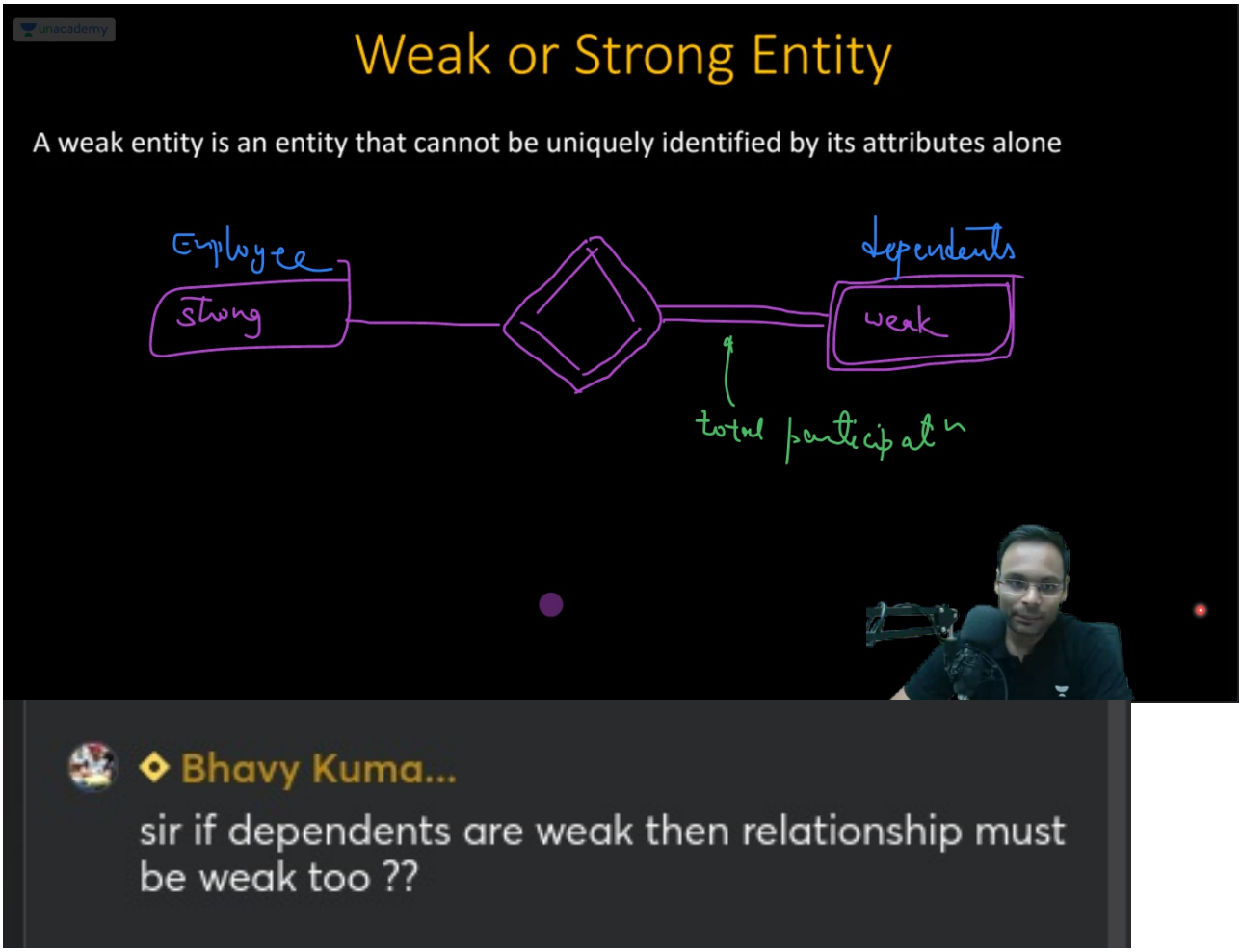
Doubt clearing session relational modeling (4) [25th June 2023]



- Total and partial participation.



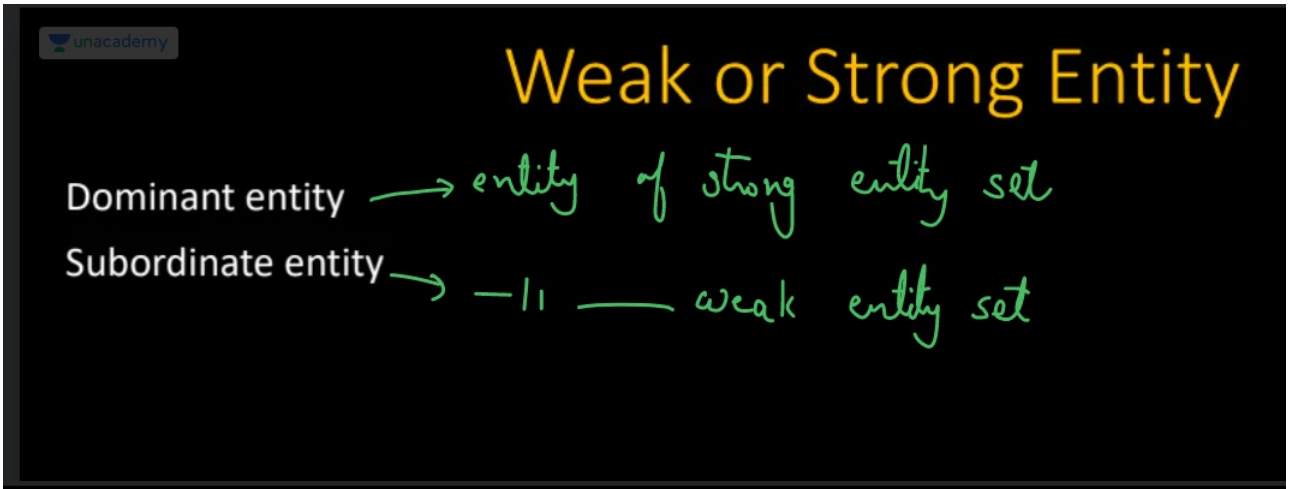
- Strong and weak entity set.
- Weak entity set participation in a **relationship** will always be **total participation**.

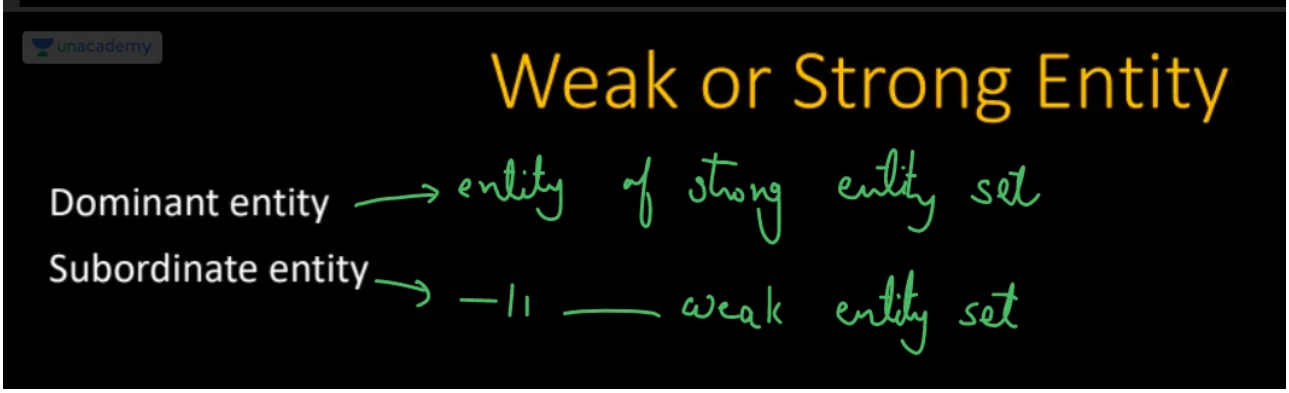


- Yes.

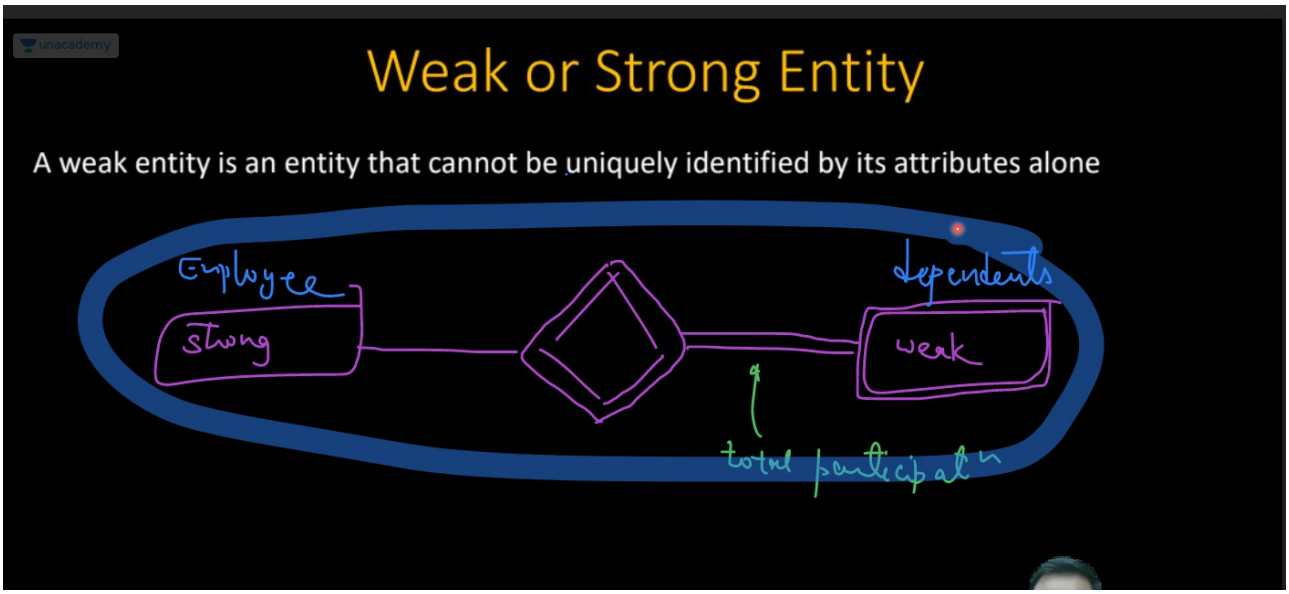
Weak or Strong Entity

- Dominant Entity -> Entity of strong entity set
- Subordinate Entity -> Entity of weak entity set.





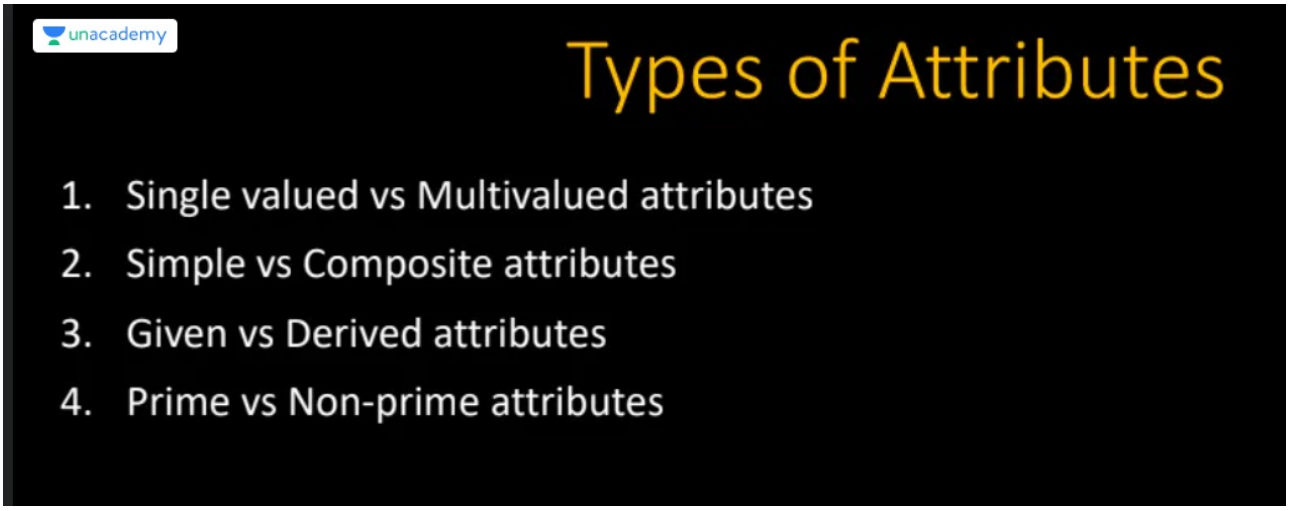
- If one **entity** of **Employee enity set** which is **strong entity set** then it is called as **Dominant Entity**.
- If one **entity** of **dependents enity set** which is **weak entity set** then it is called as **Subordinate Entity**.



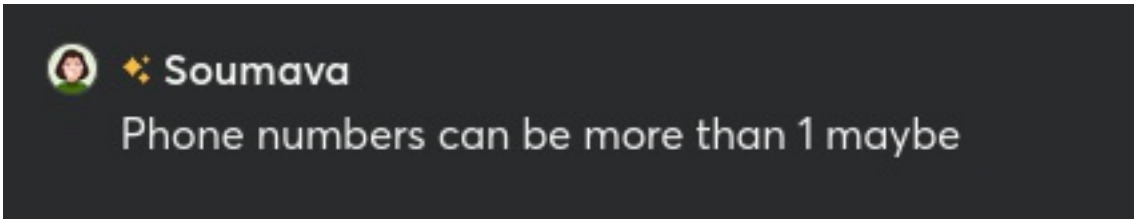
- Both the **Dominant and Subordinate** terms will come into **existance** when we are talking about the **above relationship** of **Employee and dependents** which is a relationship between a **strong entity set(Employee) and weak entity set(dependents).
- If we take one **entity** and not the whole set from the **strong entity set(Employee)** then it is called as **Dominant entity**.

- If we take one **entity** and not the whole set from the **weak entity set(Dependets)** then it is called as **subordinate entity**.

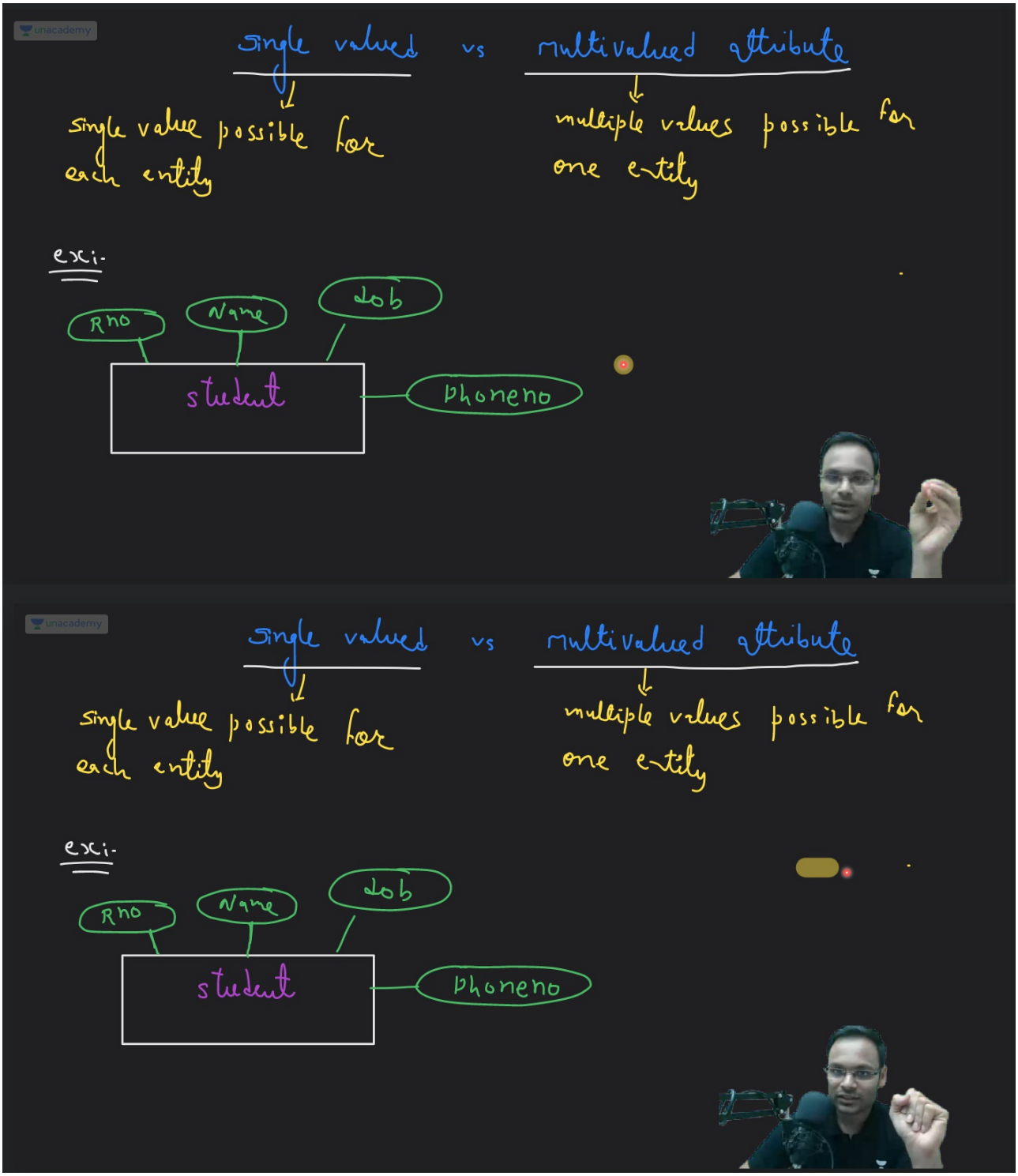
Types of Attributes



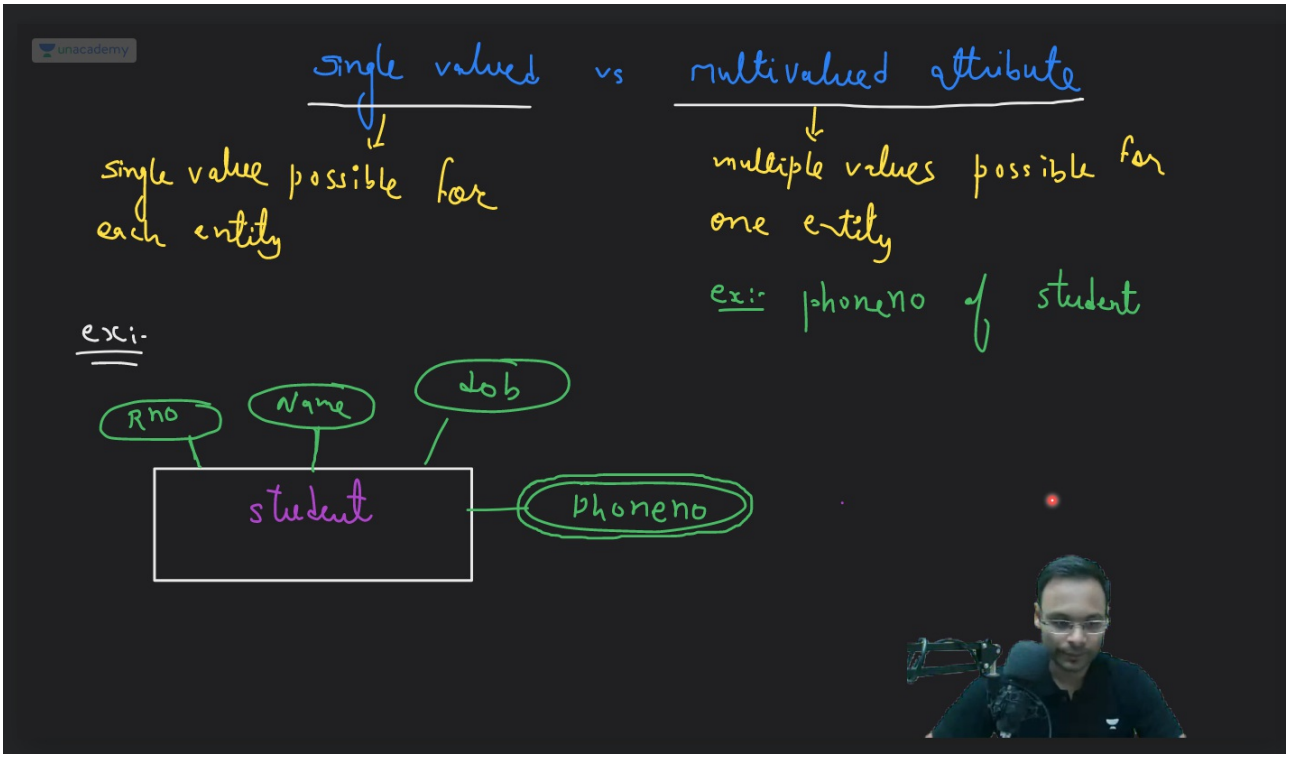
Single VS multivalued attribute



- More than **1** possible.

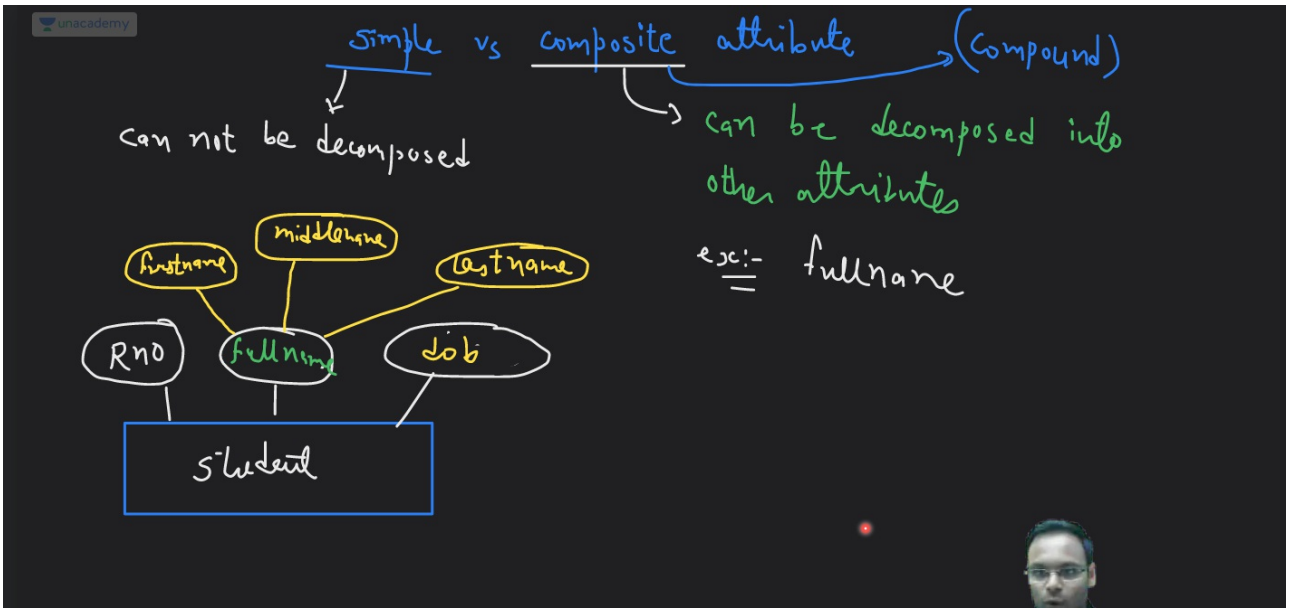


- Single Valued -> Single value possible for each entity(Rno, name, dob).
- Multi valued -> Multiple values possible for on entity(Phone no).
- To draw **Multi valued**, we need to put **double circle/ellipse** around the value in ER diagram.

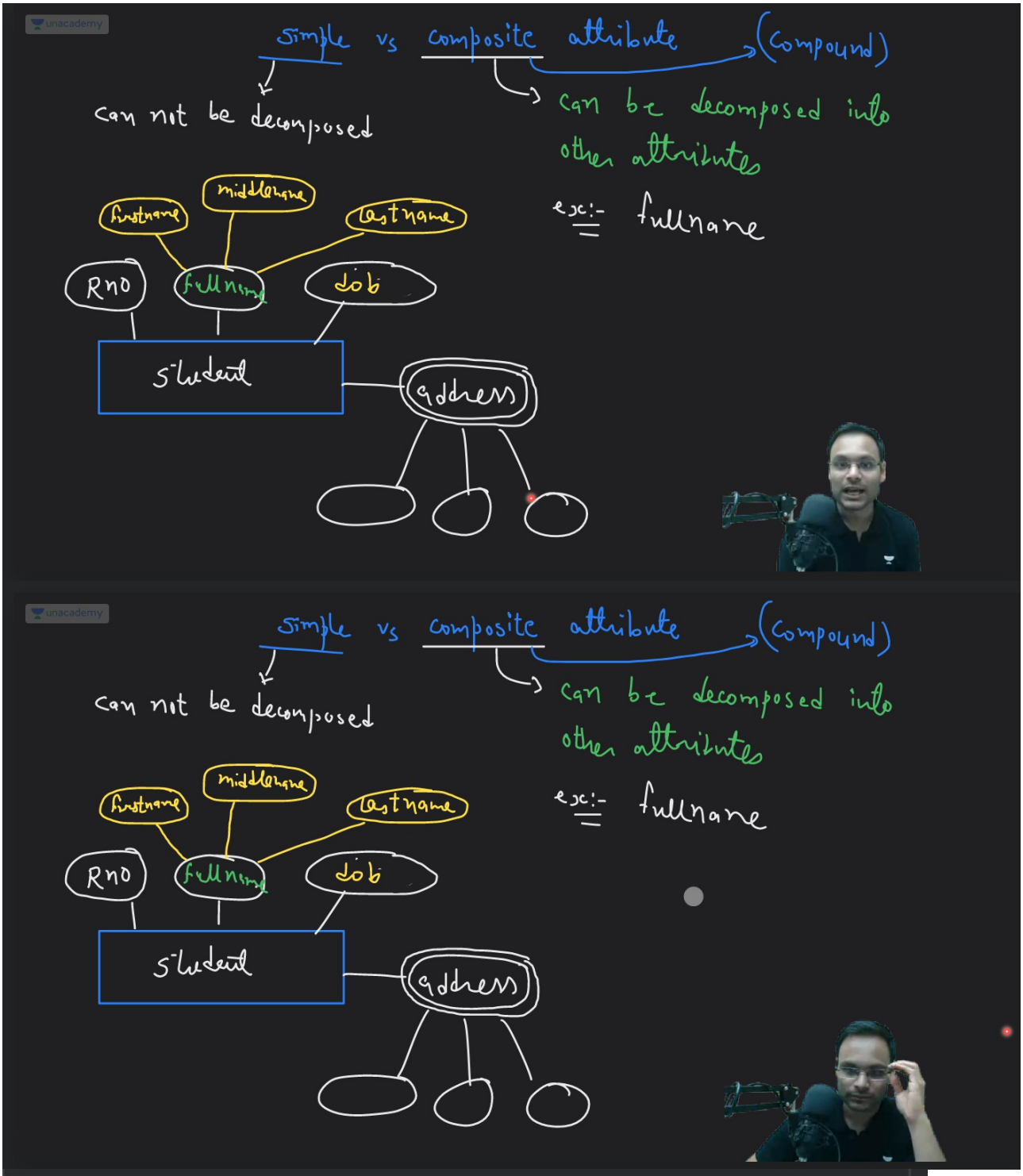


Simple VS Composite/Compound attribute

- Attributes we have that can be decomposed into more attributes then those attributes are called as **composite attributes**.
- Simple -> Single attribute cannot be decomposed into further attributes.



- Simple -> Cannot be decomposed
- Composite/Compound -> Can be decomposed into other attributes(fullname).



Yukta

total attribute of student entity ky hogi

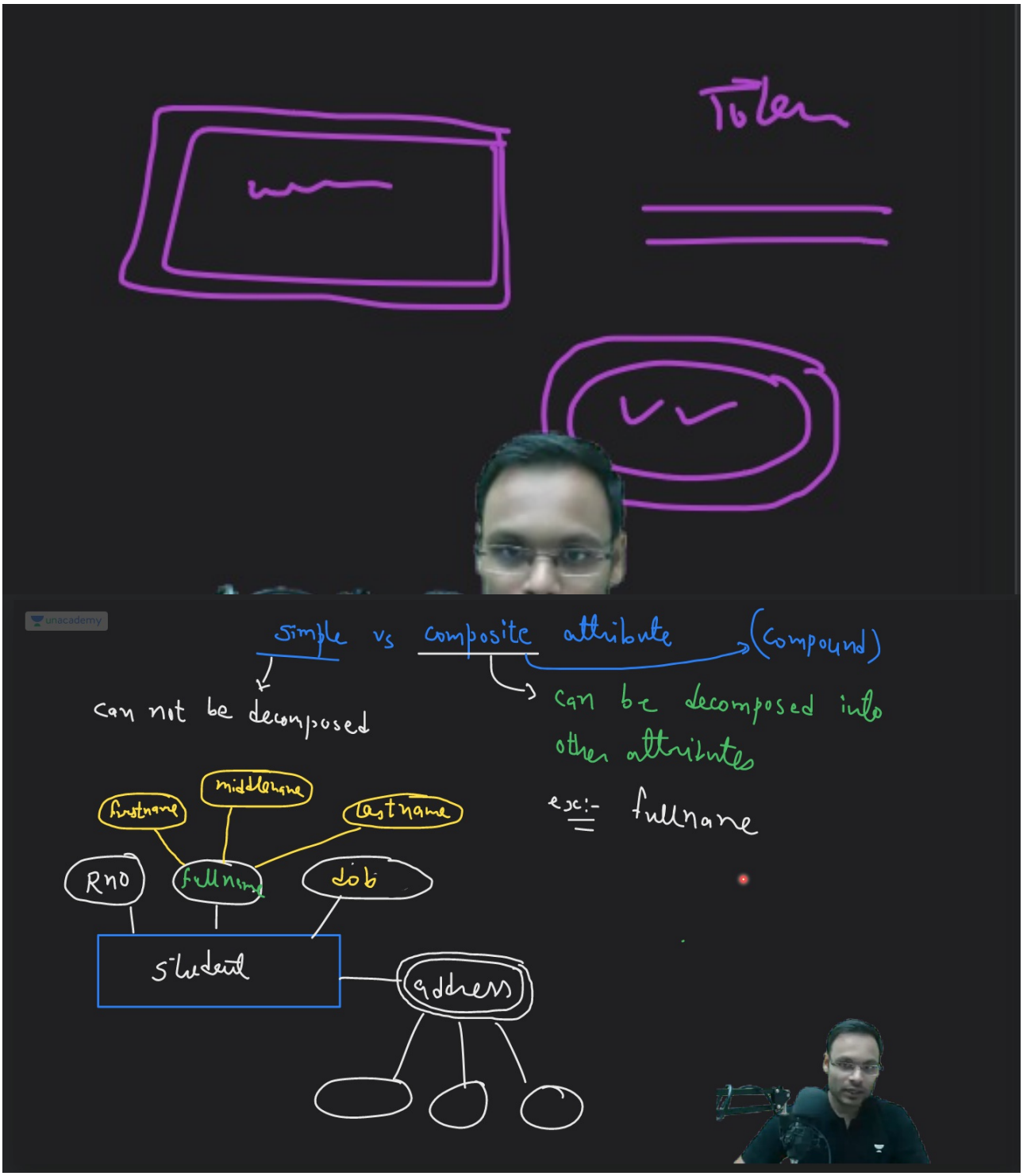
Shreyas

Ok

Yukta

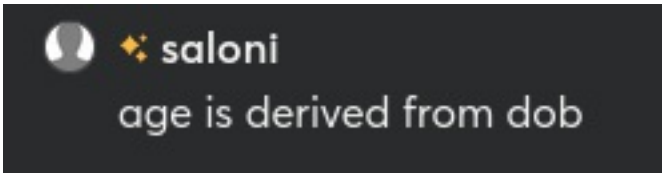
total no, of?

- We have **4** attributes here, **Rno, fullname, dob, address**.
- **fullanme** is a **composite attribute** of **firstname, middlename, last name**.

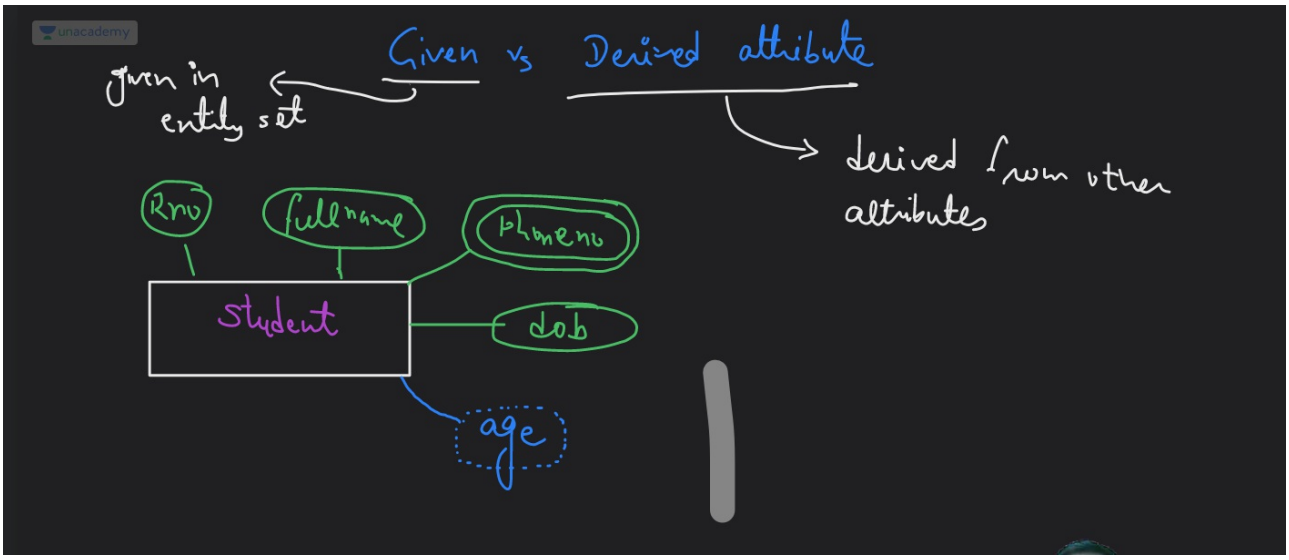


Given VS Derived attributes

- Given -> Given in entity set
- Derived -> Derived from other attributes(age) -> Dotted lines.

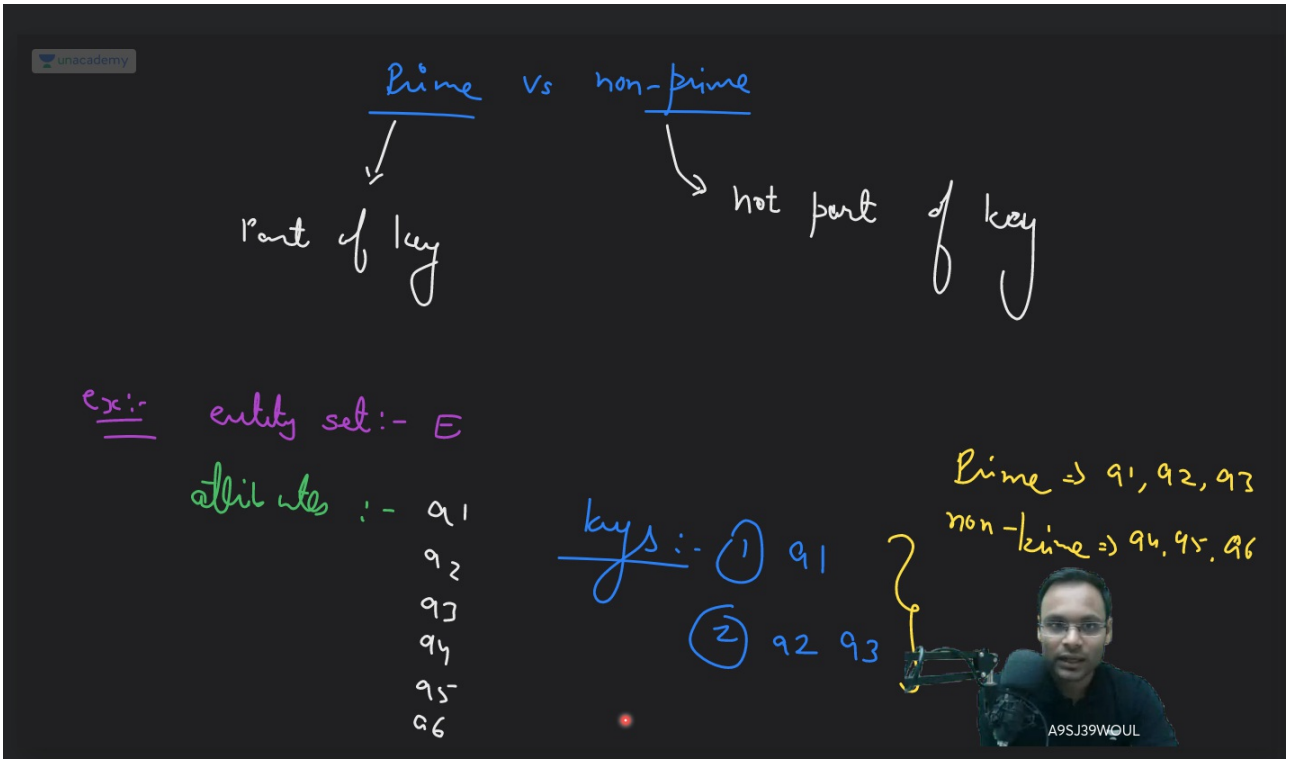


- **Age** can be derived from **dob**.
- **Age** is not a direct part of the database. We are showing that **age** can be derived from different attributes.



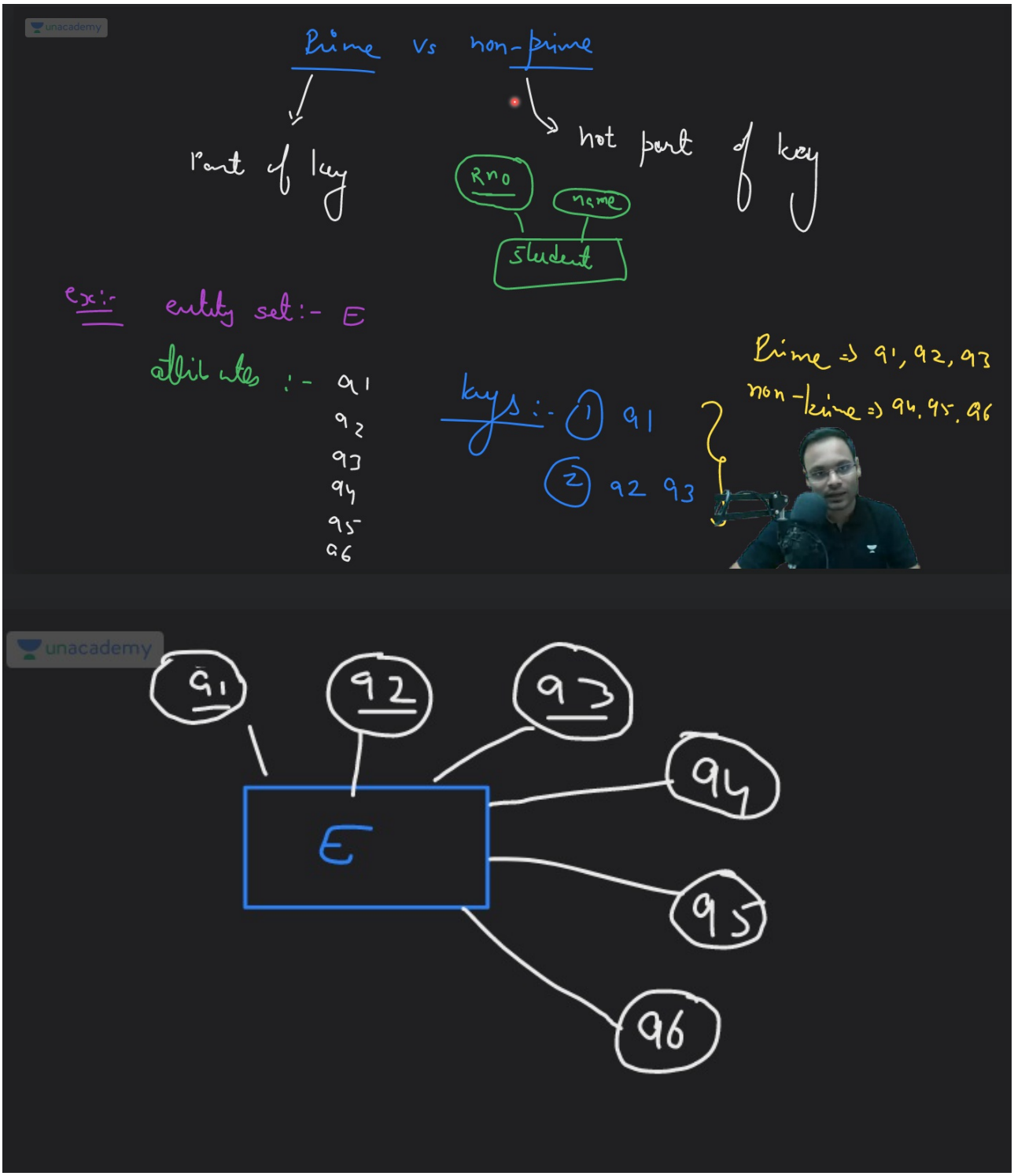
Prime VS Non-prime attributes

- Prime -> Part of key
- Non-prime -> Not part of key.

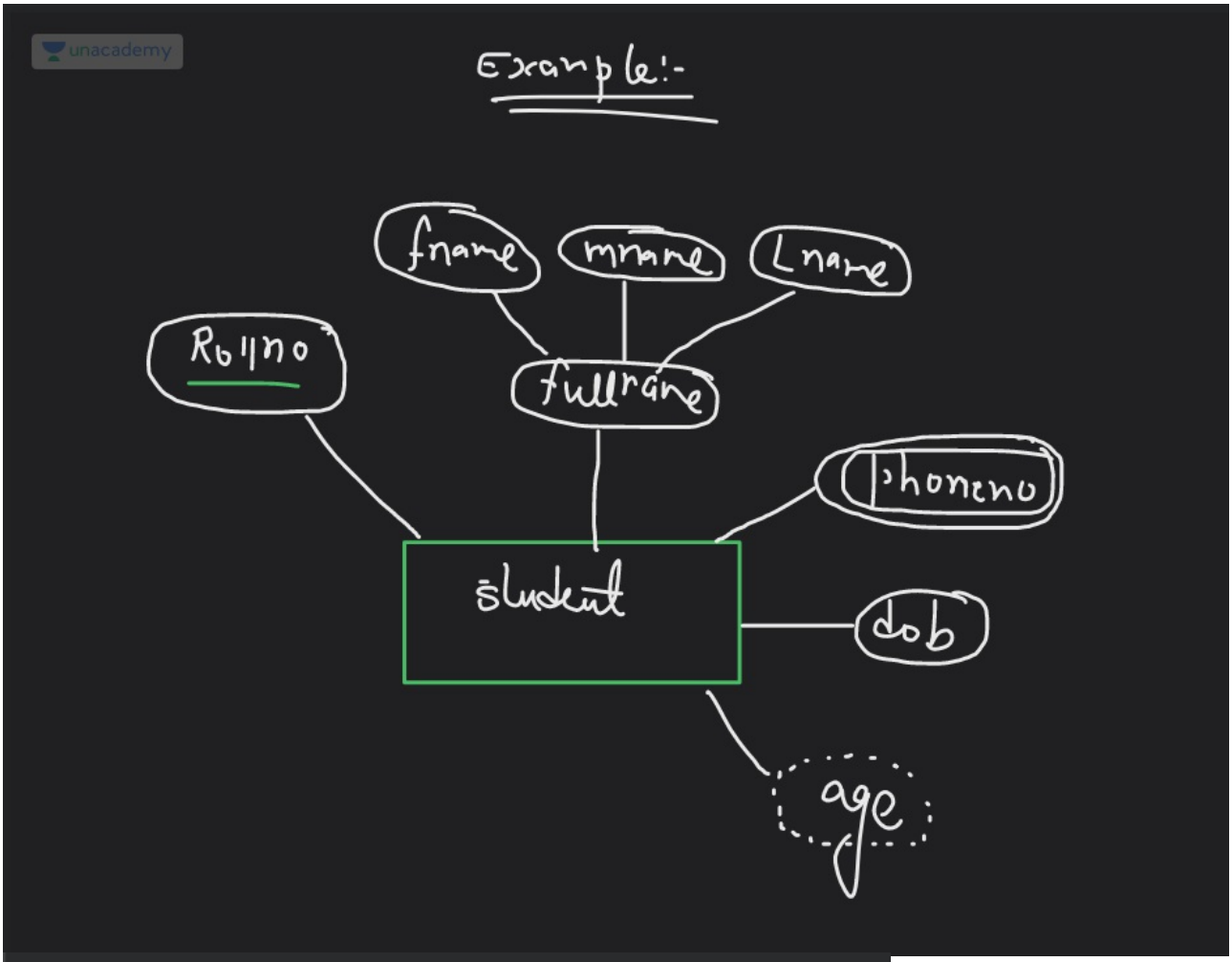


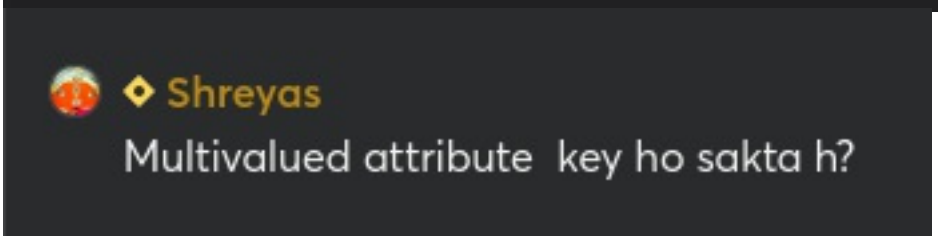
- To represent **prime attributes**, we will put an **underline** under the **prime attributes**.

We have not said that theya are **keys**.



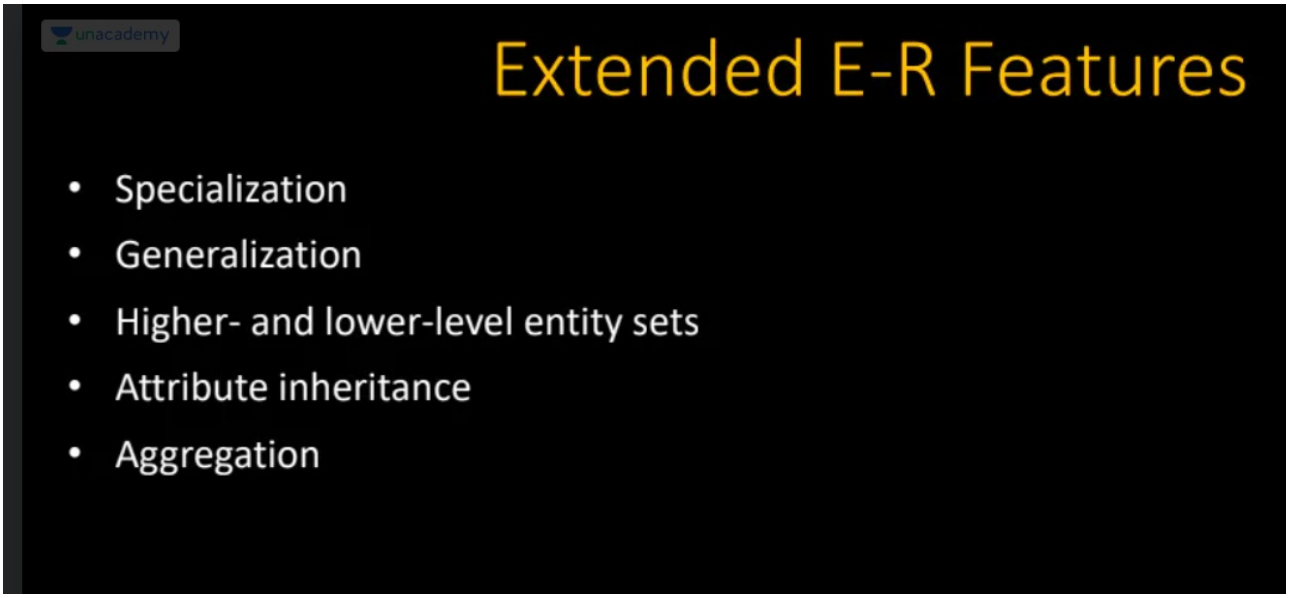
- Rollno -> Prime attribute
- fullname -> Composite attribute
- Phoneno -> Multivalue attribute
- dob -> Normal attribute
- Age -> Derived attribute





- No.

Entended ER features



1. Specialization -> Specializing the information of an entity set.
2. Generalization
3. Higher and lower level entity sets
4. Attribute inheritance
5. Aggregation

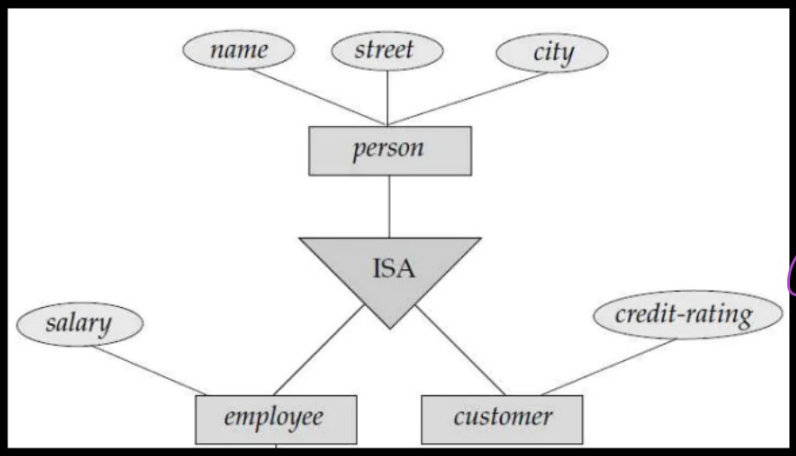
Specialization

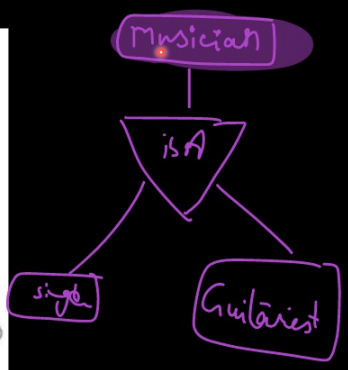
The process of designating subgroupings within an entity set is called specialization.

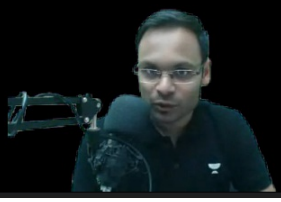
Generalization

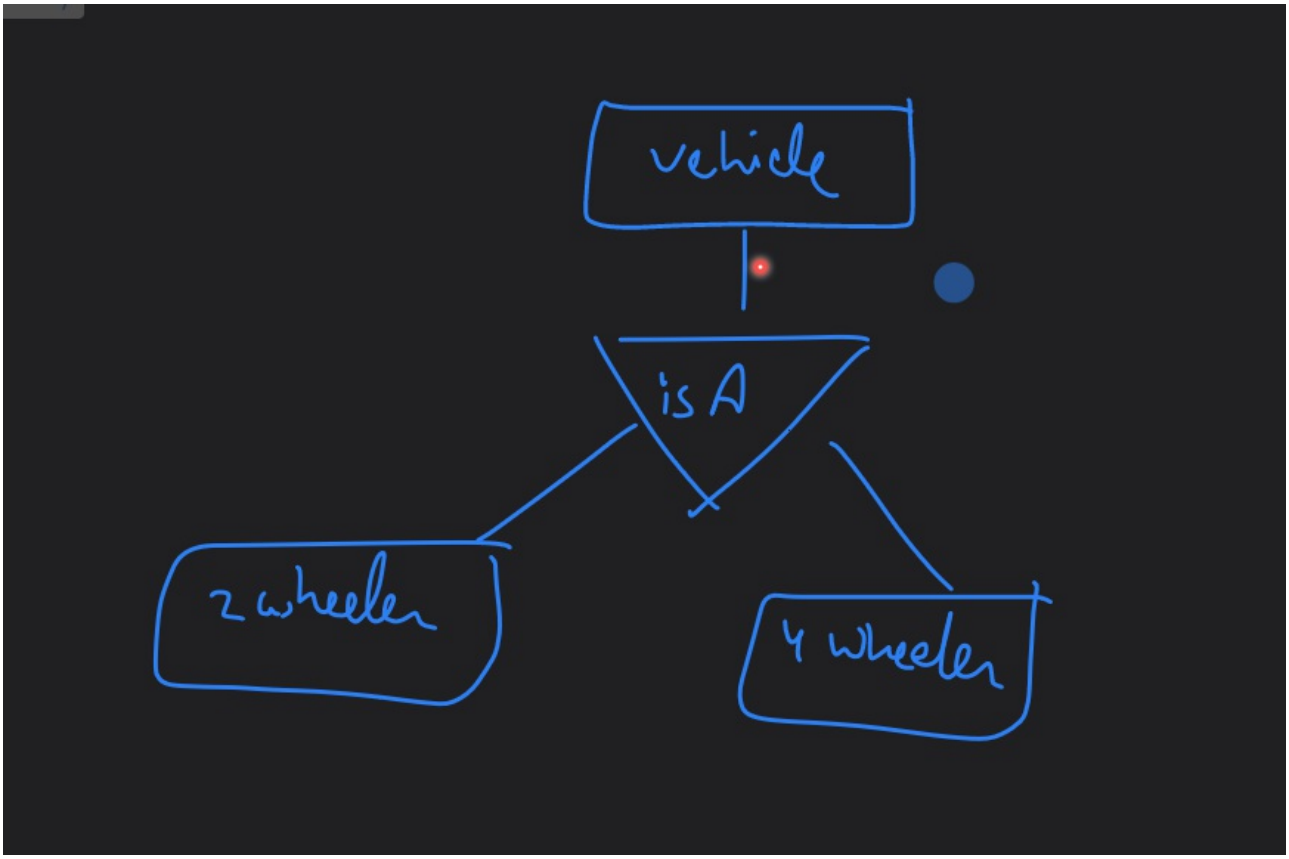
This commonality can be expressed by generalization, which is a containment relationship that exists between a higher-level entity-set and one or more lower-level entity sets

Generalization

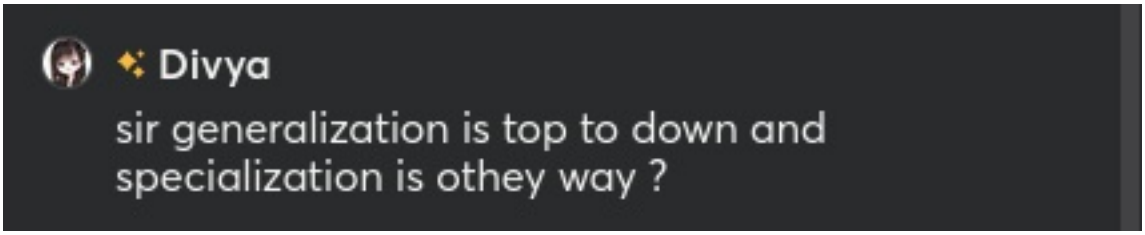




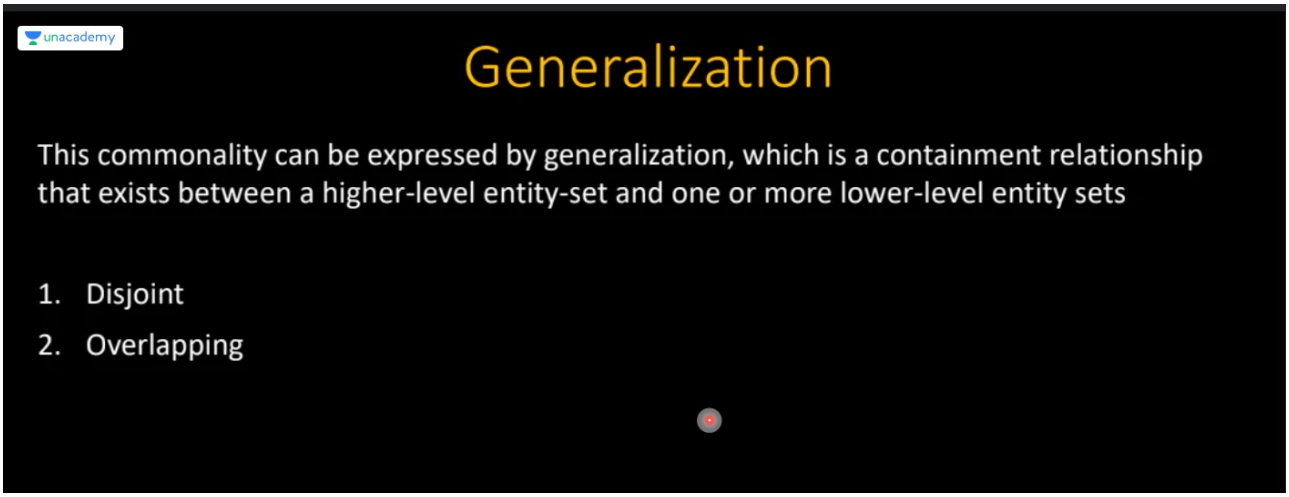




- Vehicle is a **2-wheeler** and **4-wheeler** -> **Specialization**
- **2-wheeler** and **4-wheeler** are **vehicles** only -> **Generalization**.
- In one **entity set**, we are specializing two different **special categories**.
- When we are going towards **specilization**, we have to make the **special categories** when the **specialized entity sets** have some **special attributes**.
- If there is no **special attributes** for the **specialized entity sets** then there is no point in creating the **special categories**.

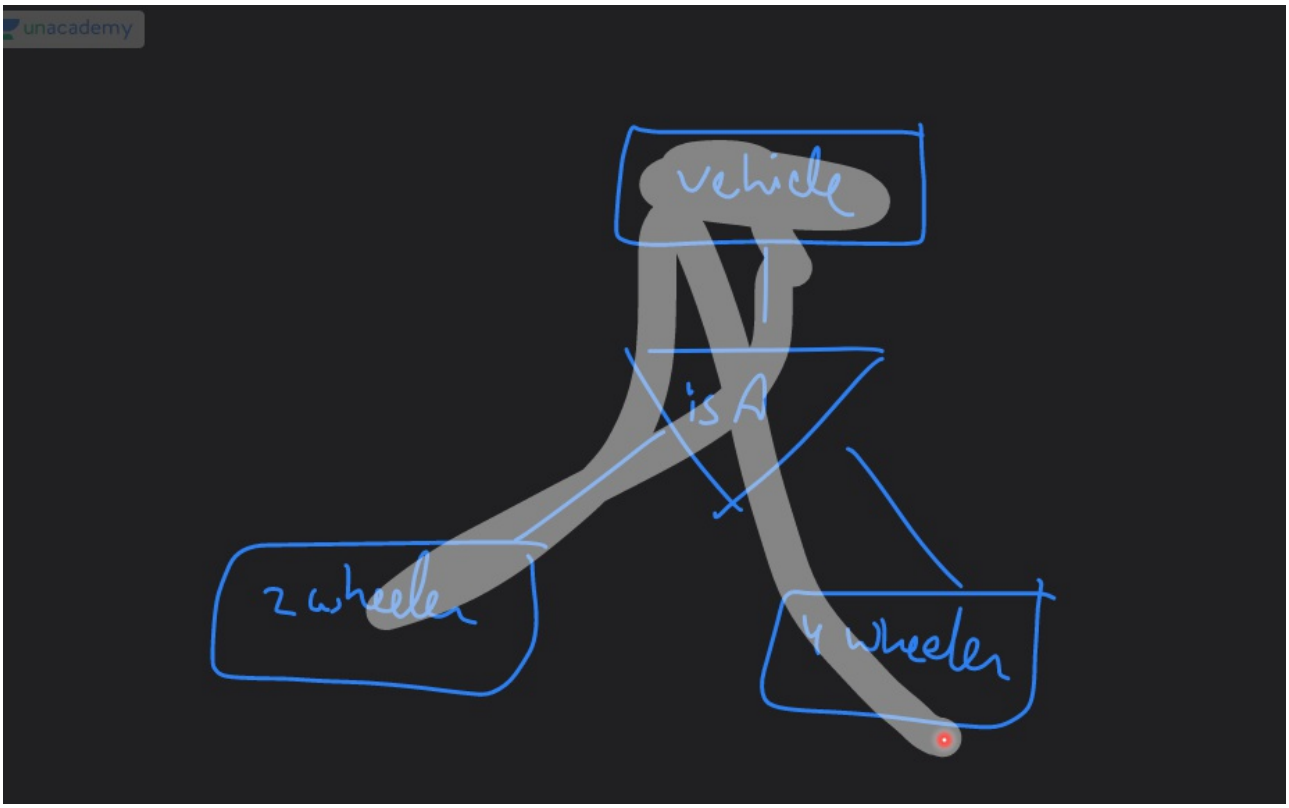


- Top to bottom is **specialization**
- Bottom to Top is **generalization**.



1. Disjoint set -> Any entity is **mutual exclusively** going to either one of the **specialized entity sets**.

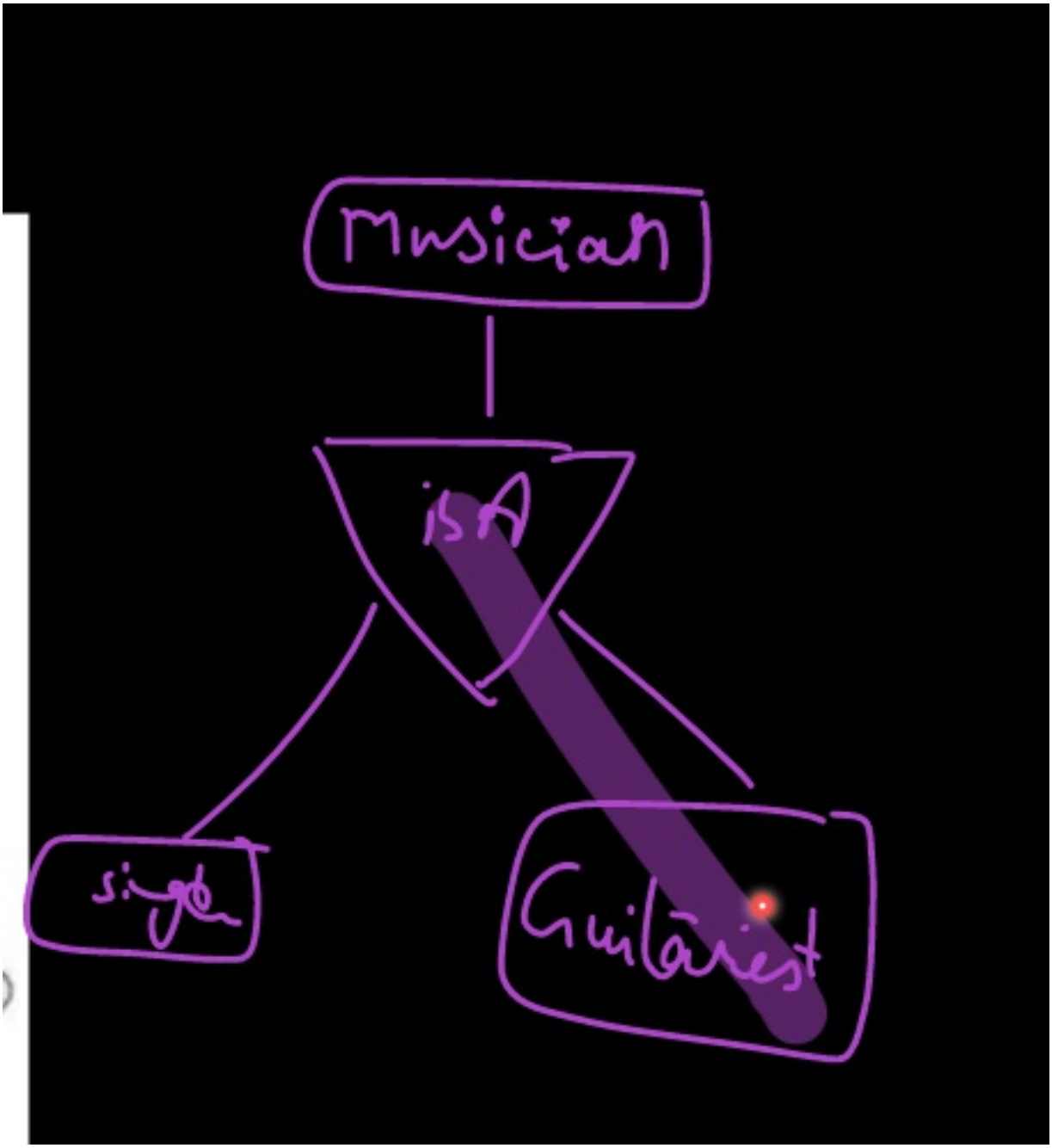
- One entity of higher-level entity set(**Vehicle**) is present in only one lower-level entity set(**2-wheeler** and **4-wheeler**).



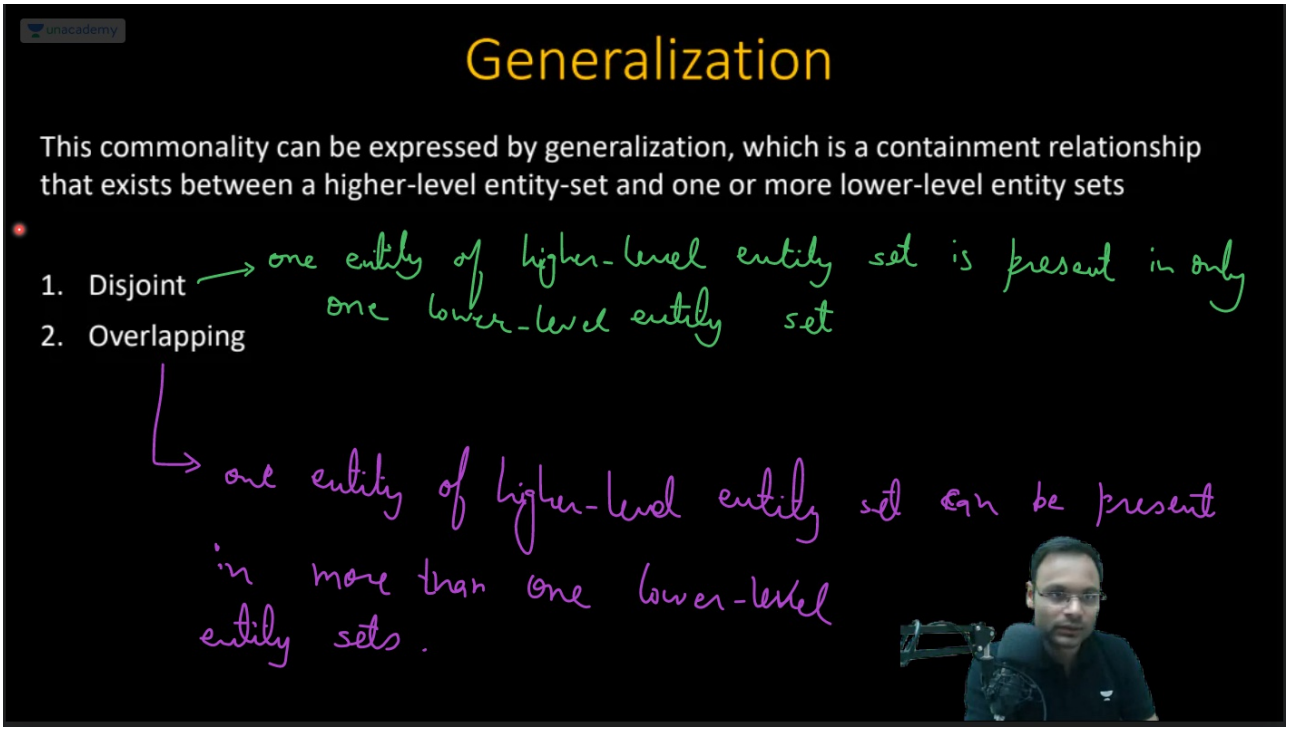
- Vehicle will be either **2-wheeler** or **4-wheeler**.

2. Overlapping -> Present in more than one.

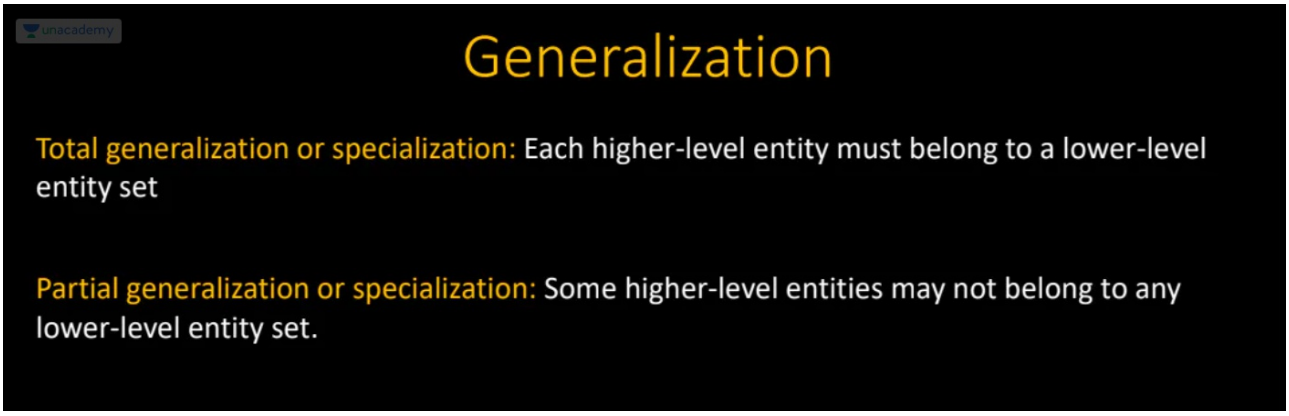
- One entity of higher-level entity(Musician) set can be present in more than one lower-level entity sets(**singler** and **guitarist**).



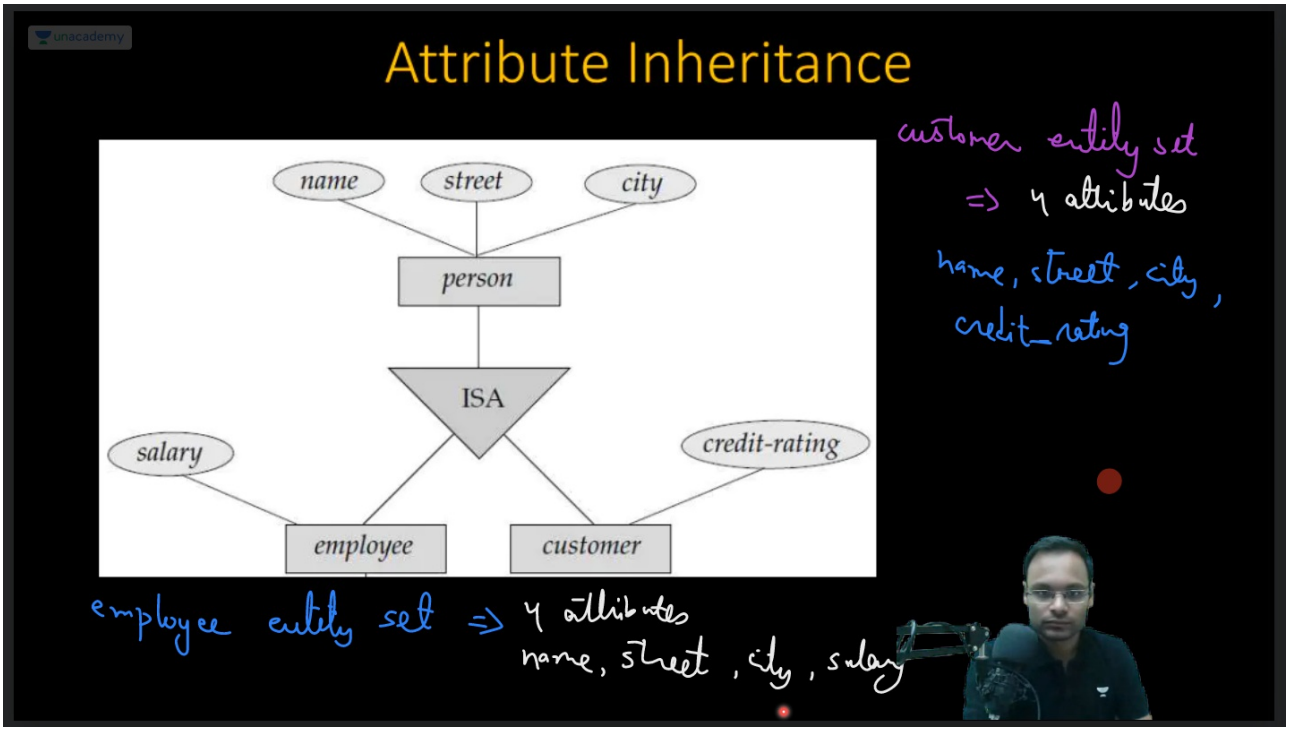
- Musician can be **singler** and **guitarist** as well.



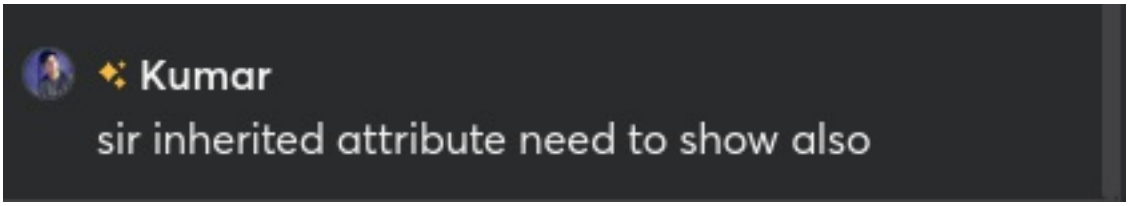
- **Total Generalization or specialization** -> Each higher-level entity must belong to a lower-level entity set.
- **Partial Generalization or specialization** -> Some higher-level entity may not belong to any lower-level entity set.



Attribute Inheritance



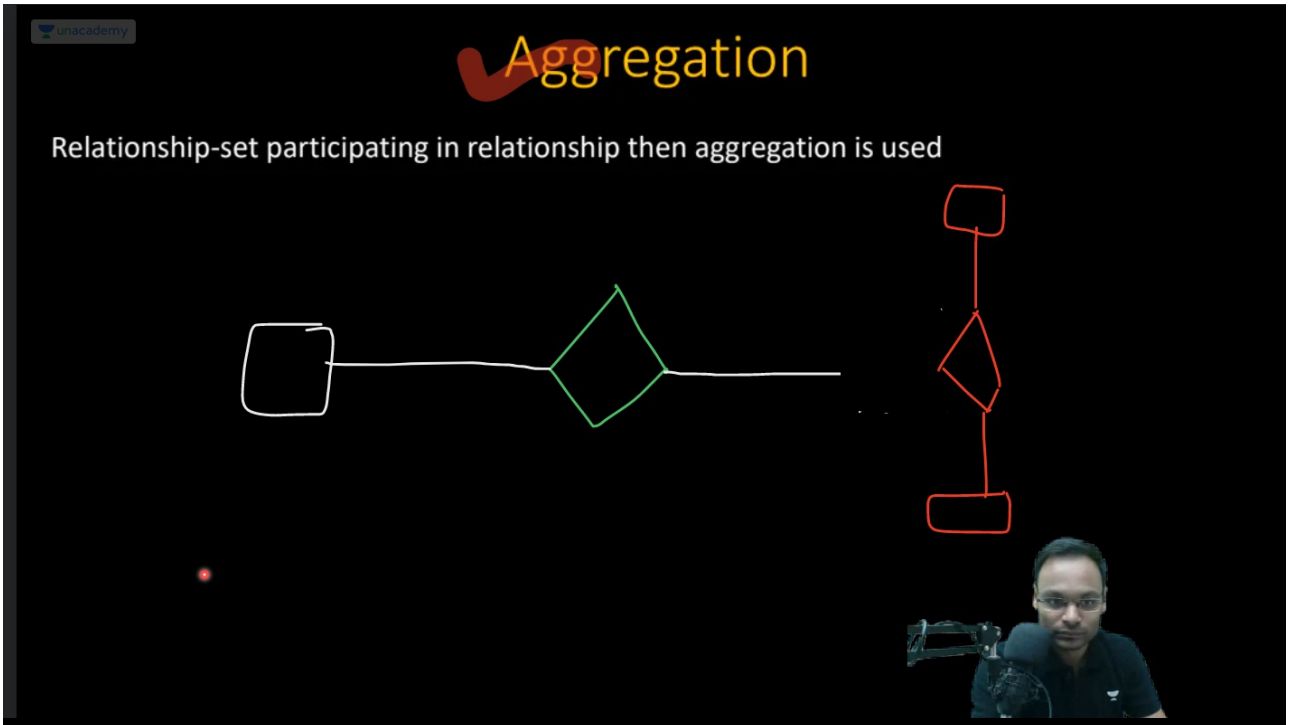
- Employee entity set -> 4 attributes -> Name, street, city, salary
- Customer entity set -> 4 attributes -> name, street, city, credit_rating.

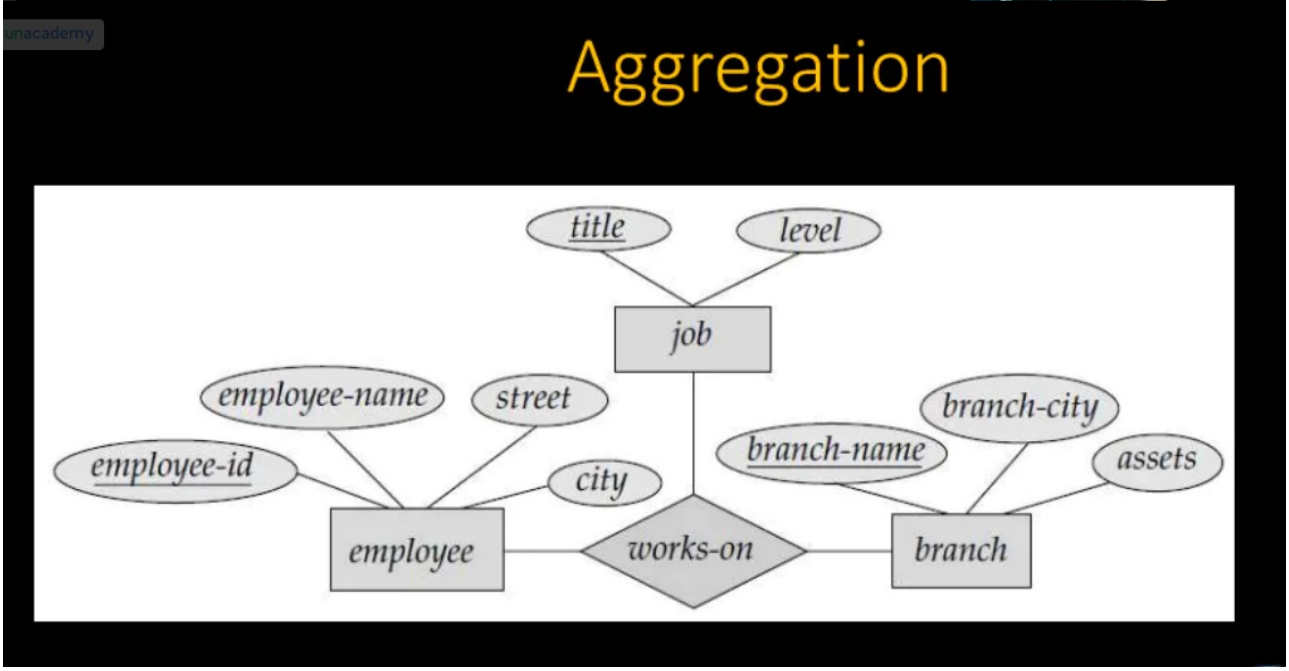


- No.
- Inheritance is **top to bottom** only.

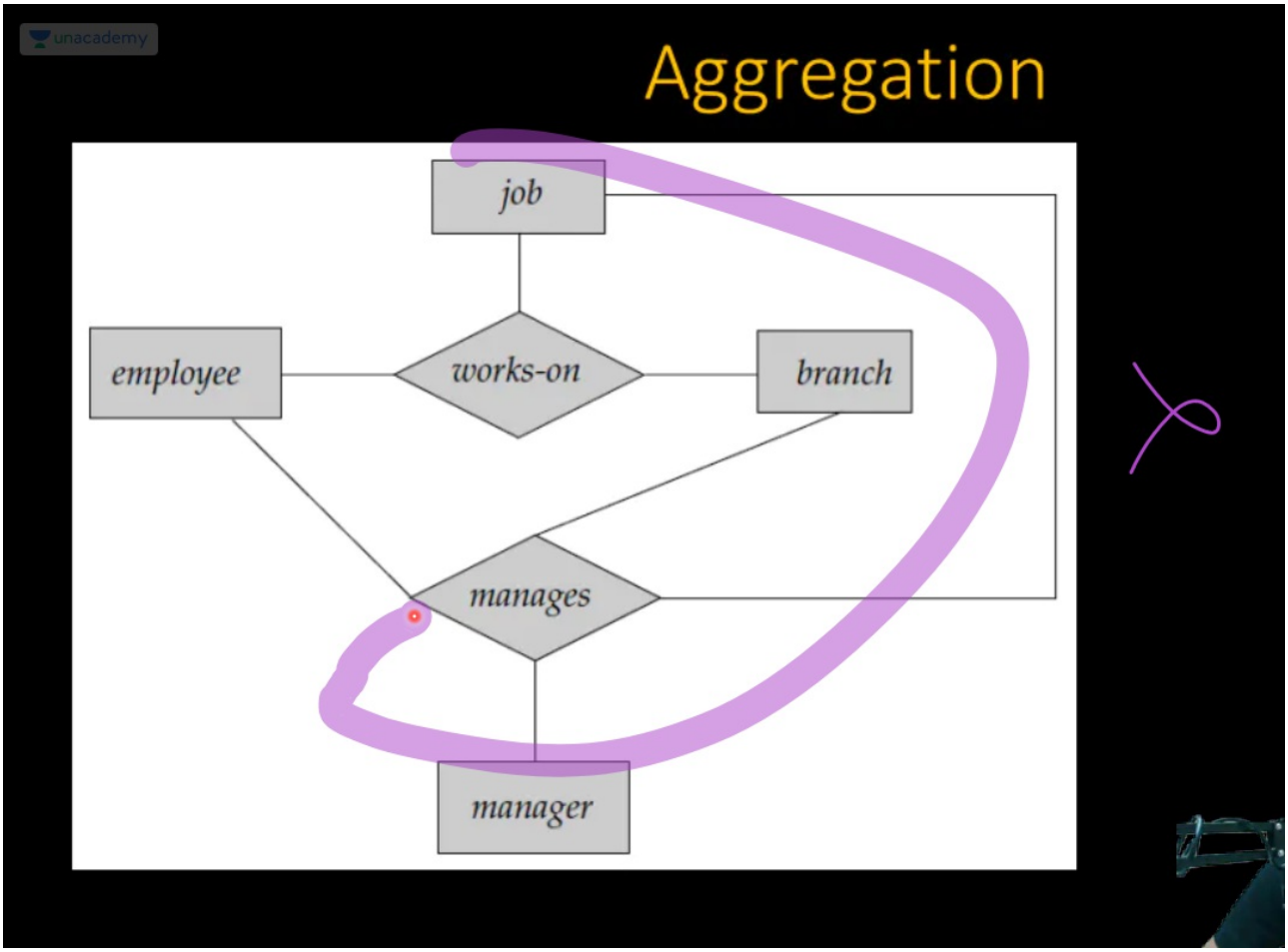
Aggregation

- Relationships is shown betweeen **entity sets**.
- Relationships-set participating in relationship then aggregation is used.

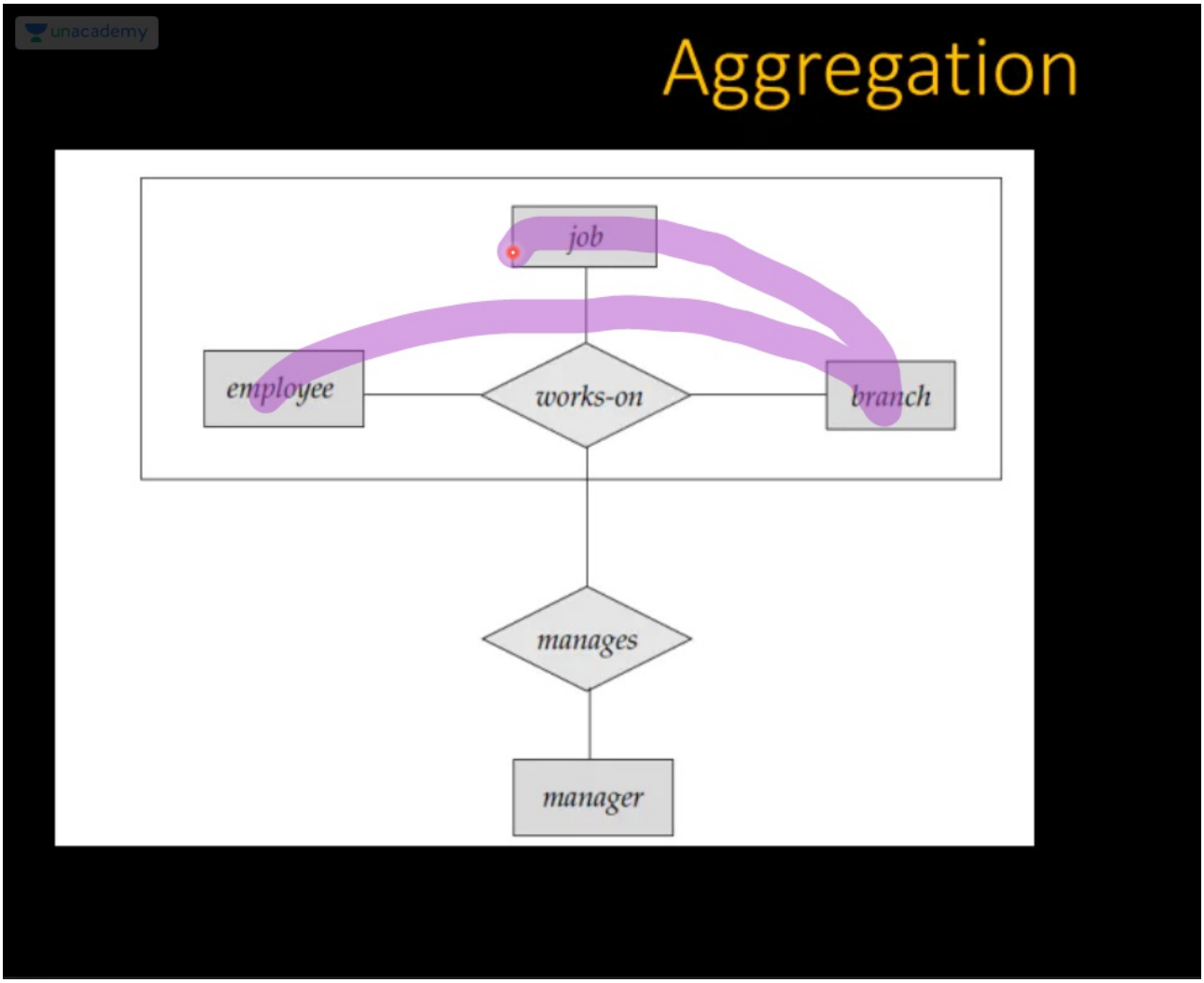

unacademy
Aggregation
Relationship-set participating in relationship then aggregation is used


unacademy
Aggregation
title
level
job
employee-name
street
branch-city
branch-name
assets
employee-id
city
employee
works-on
branch

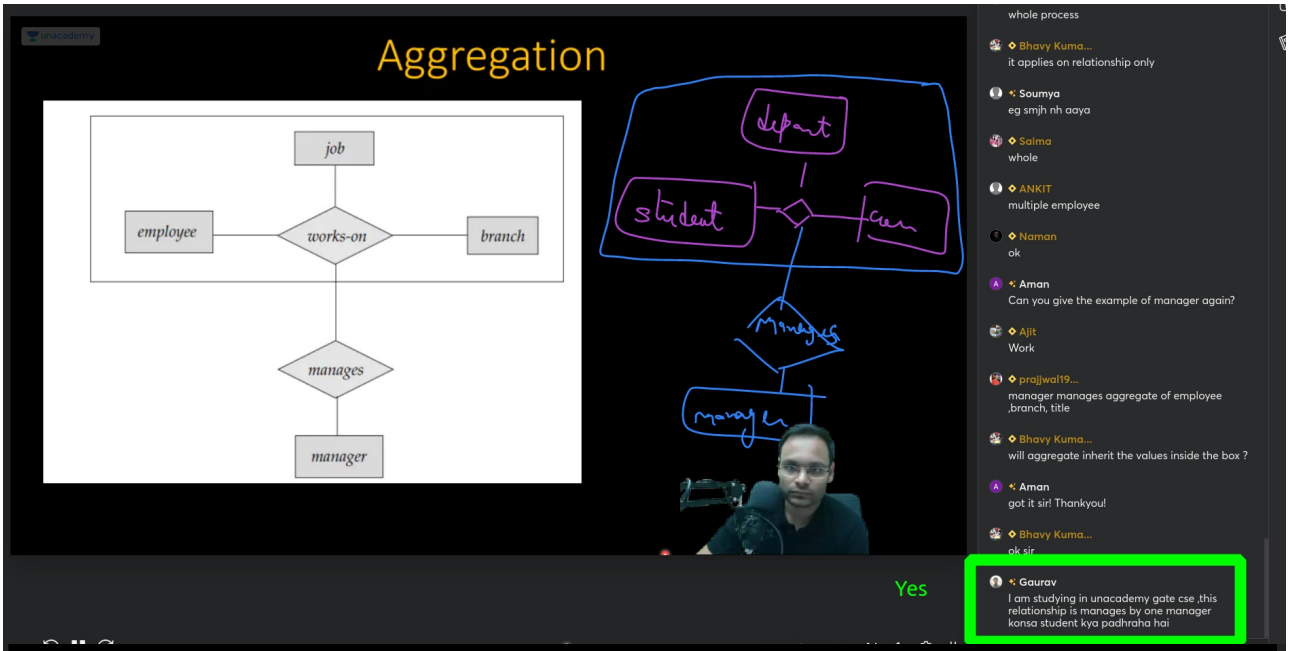
- Example

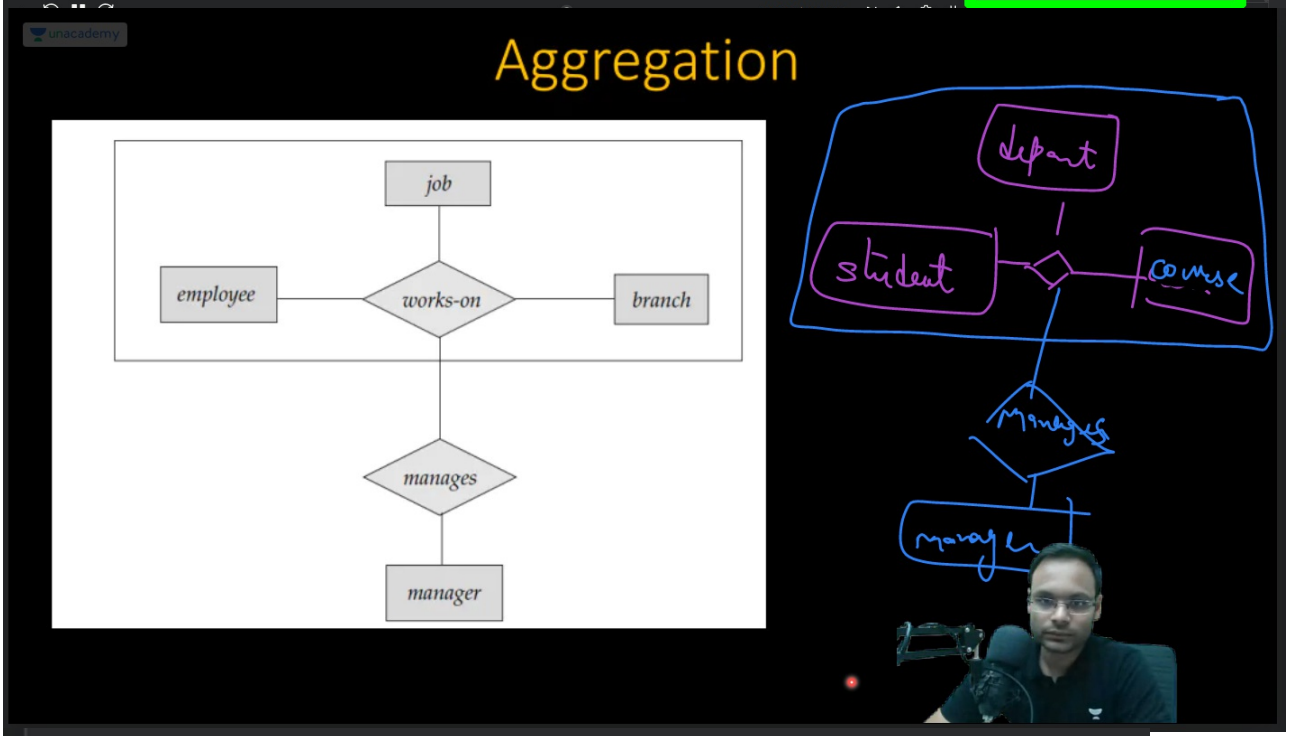

unacademy
Aggregation
job
employee
works-on
branch
manages
manager

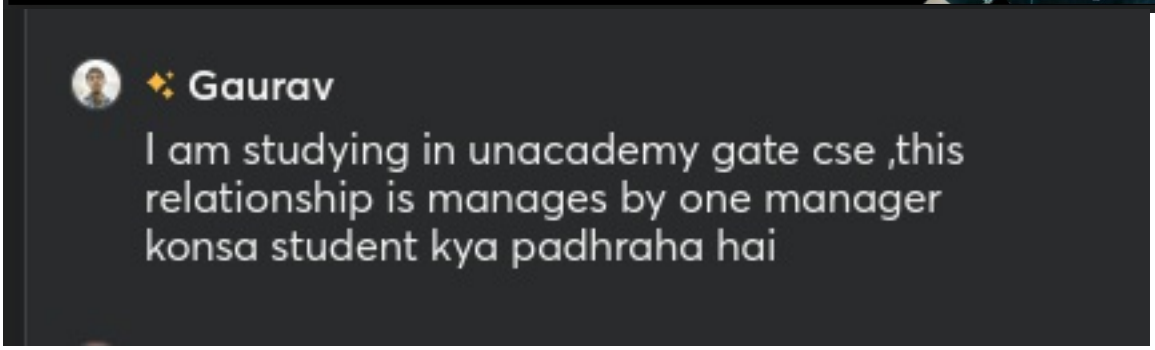
- Wrong way.



- Correct way.
- The manager is not **managing** individual job, employee or branch. **Manager** manages the already exisiting relationship.
- We want to show a relationship of a relationship with a different entity set is called as **Aggregation**.



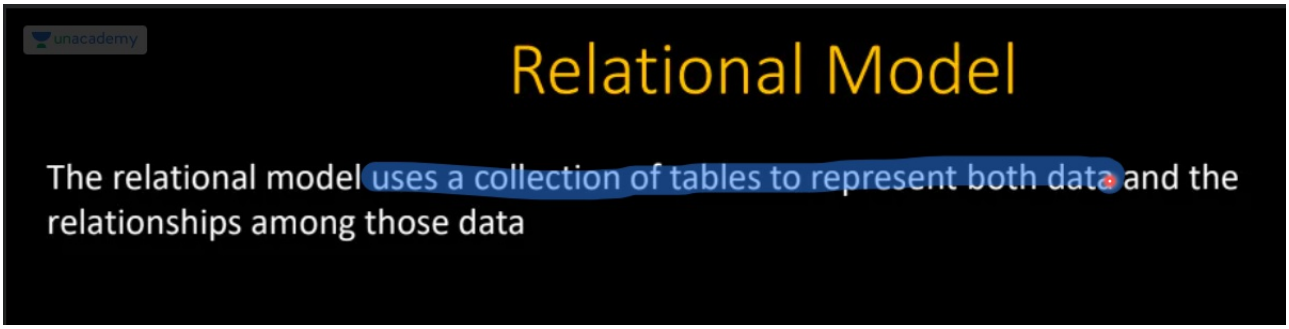




Keys and sql part-I (5) [25th June 2023]

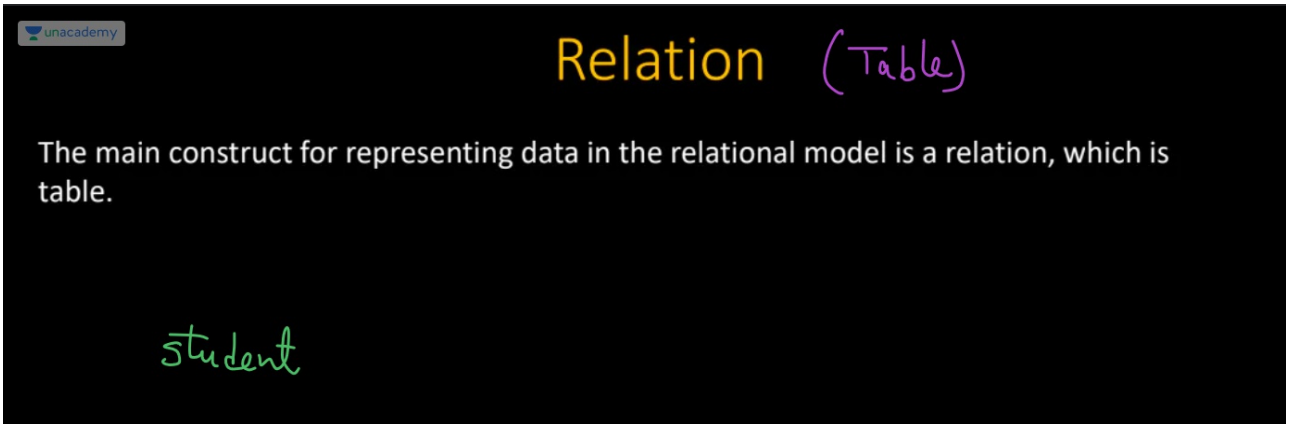
Relational Model

- The relational model uses a collection of tables to represent both data and the relationships among those data.



Relation

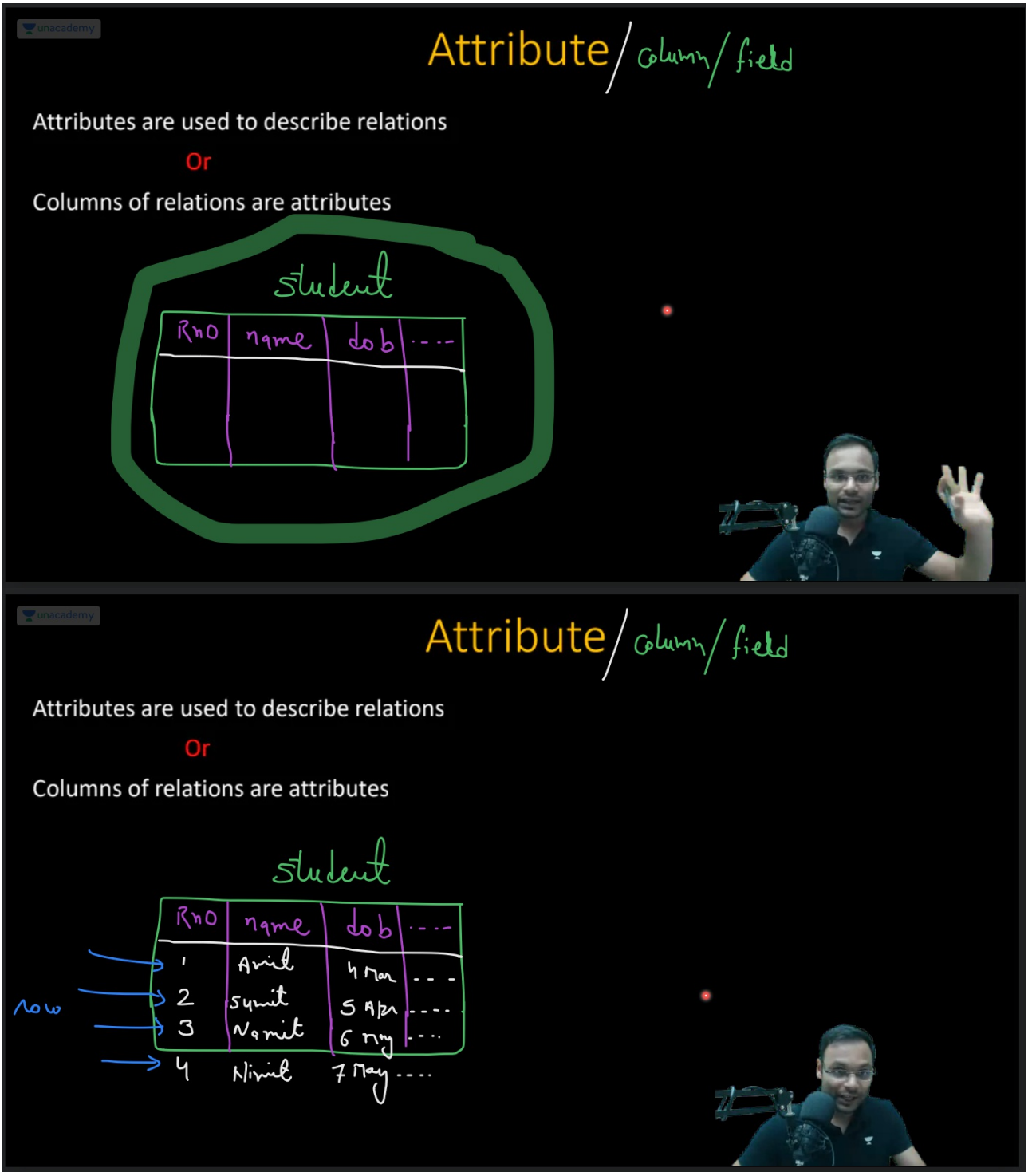
- The main construct for representing data in the relational model is a relation, which is table.



- Relation -> Table.
- There is a relation **Student** which means that there is a **student** table.

Attribute/Column/Filed

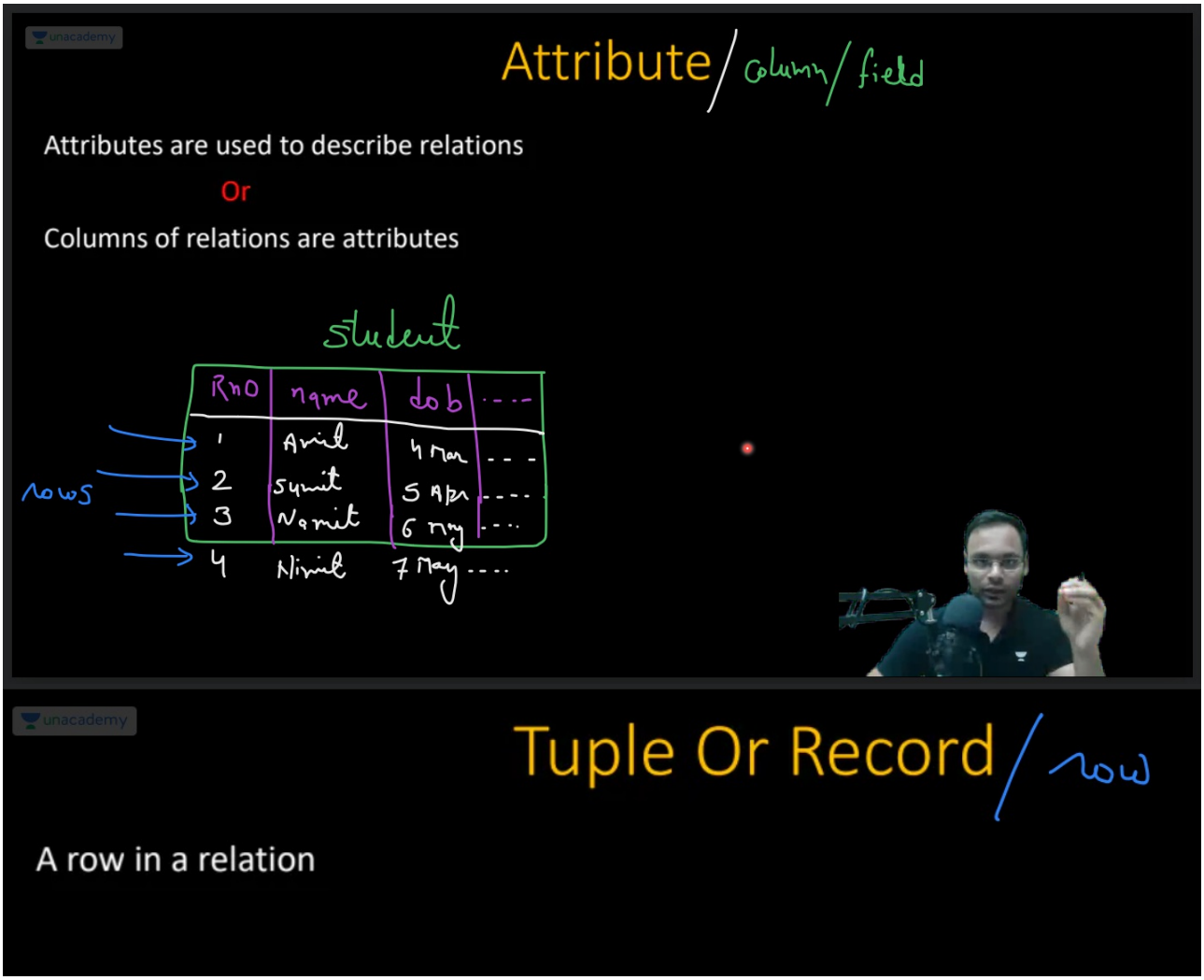
- Attributes are used to describe relations
- Columns of relations are attributes.
- Table Name -> Relation(student)
- Columns -> Attributes(Rno, name)



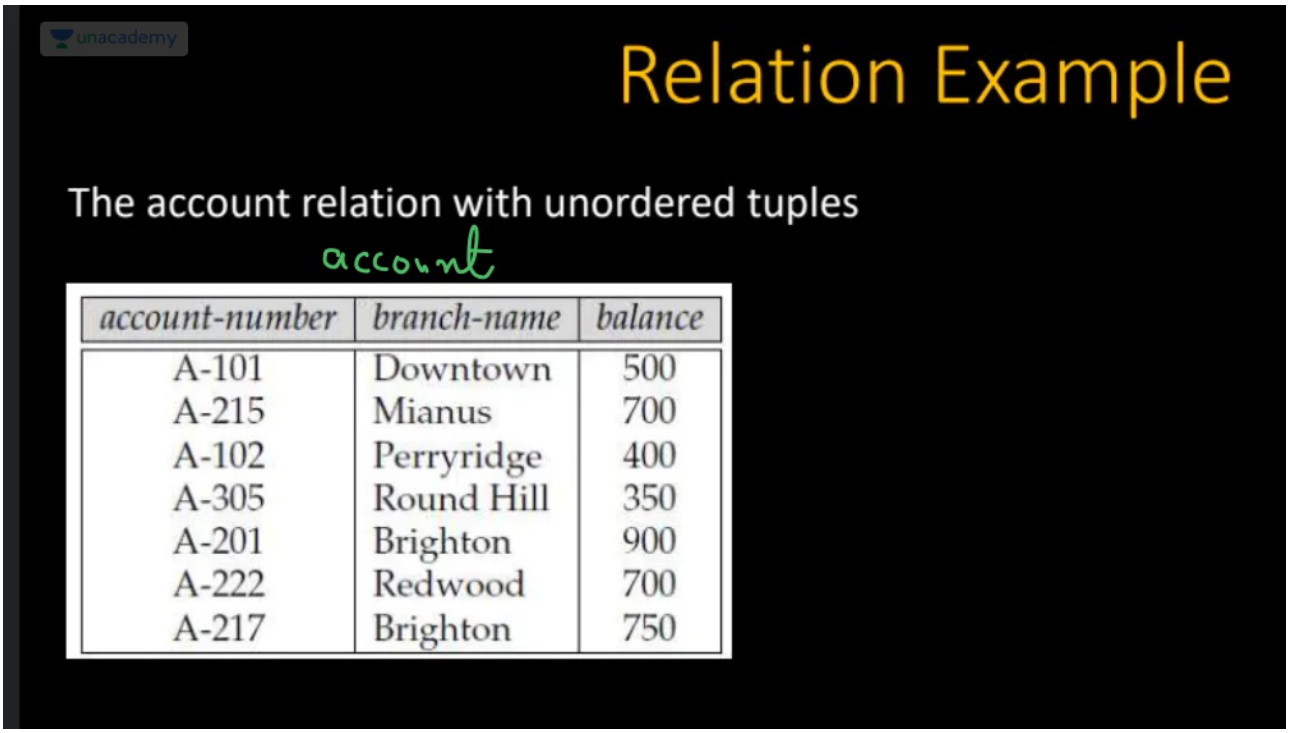
- Row
- 4 rows, so we have **4 tuples/records** within the relation.

Tuple/Record

- A row in a relation.
- A row inside the table is called as **tuple**.



Relation example

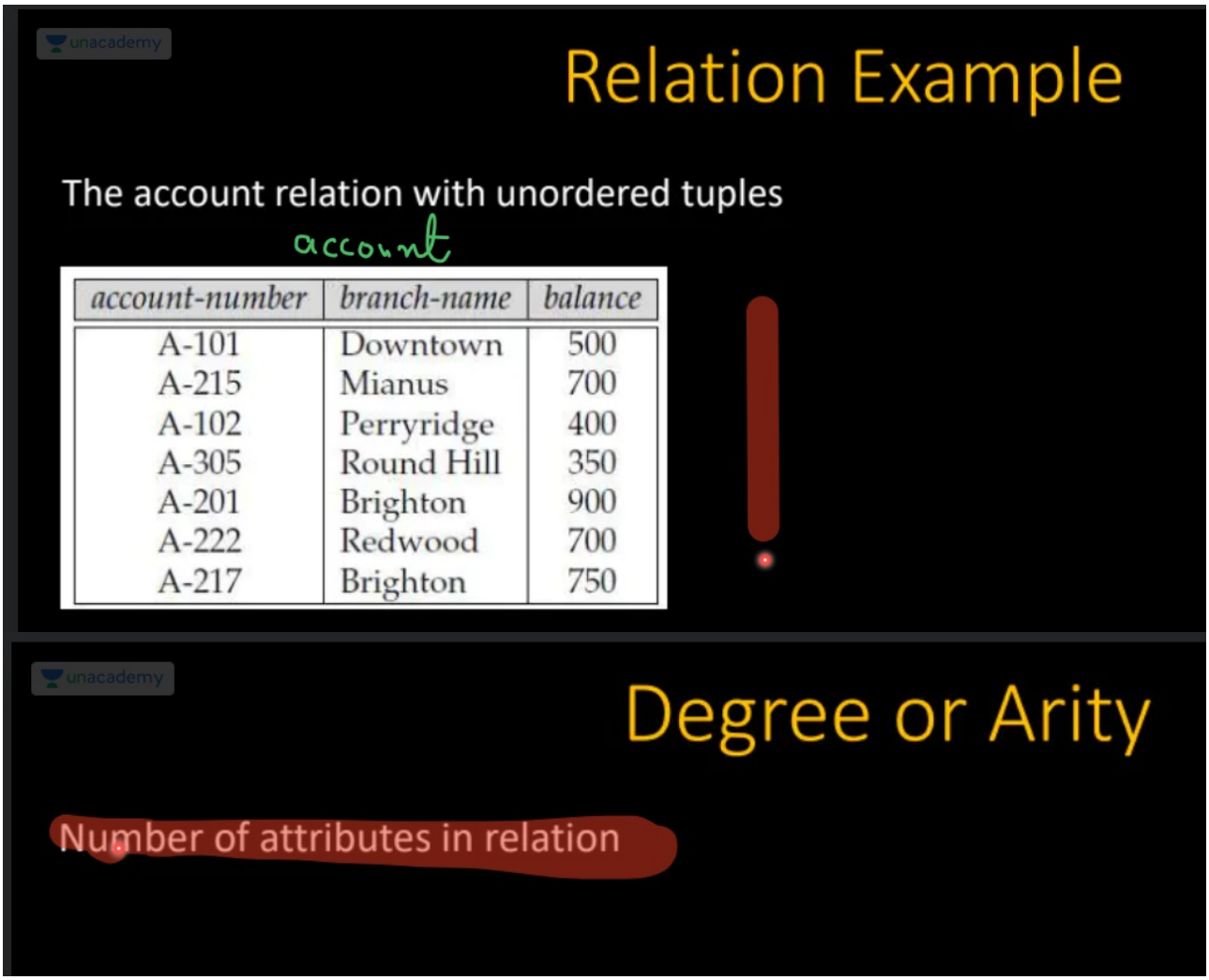


account-number	branch-name	balance
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

- We have **7 rows**, so there are **7 tuples**.

Degree or Arity

- Number of attributes in relation.
- Column -> Degree or Arity

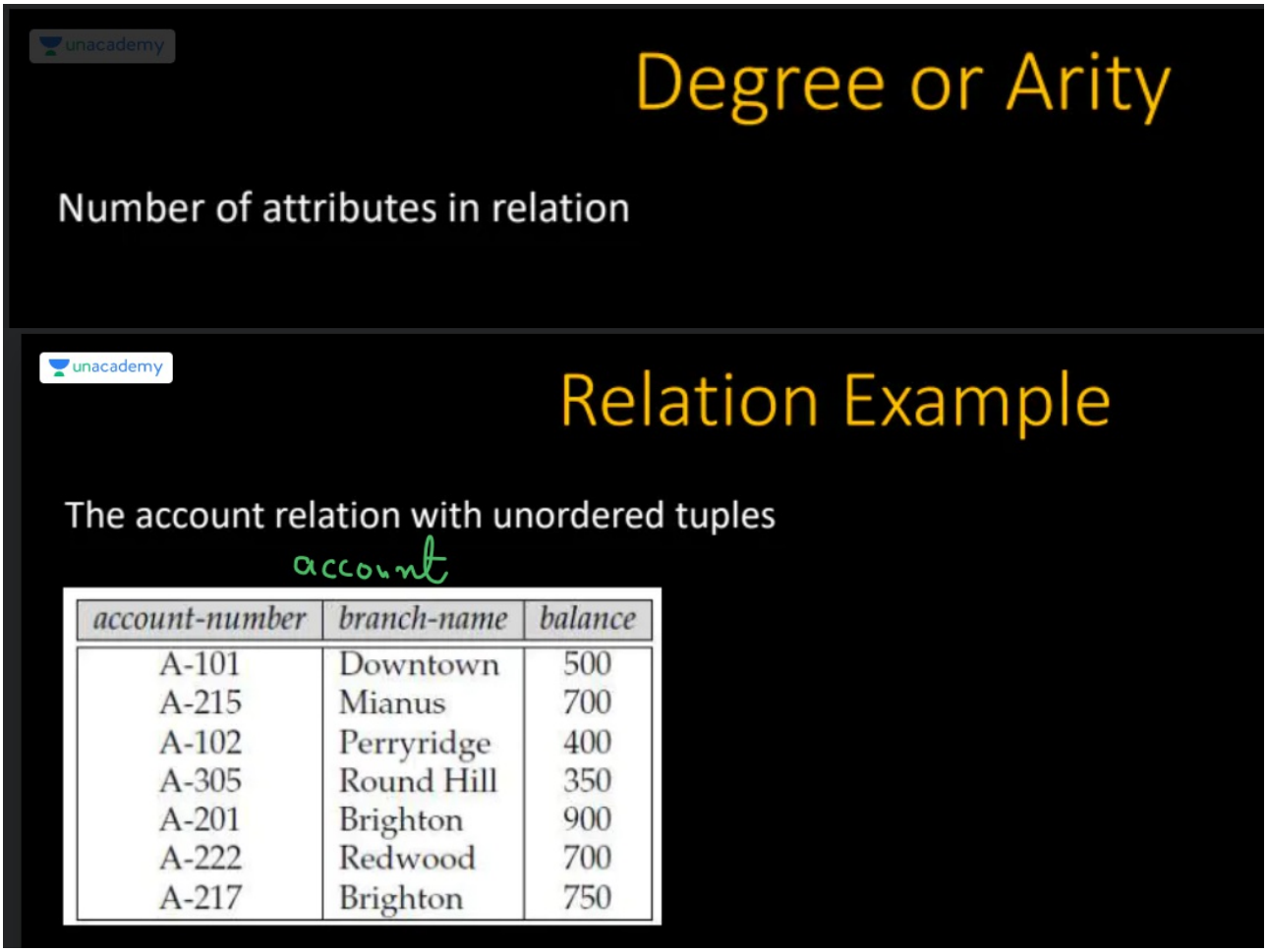


- What is the **degree/arity** of the relation?

> **3** as we have **3 columns** here.

Cardinality

- No. of tuples in a relation.
- No. of rows or records.

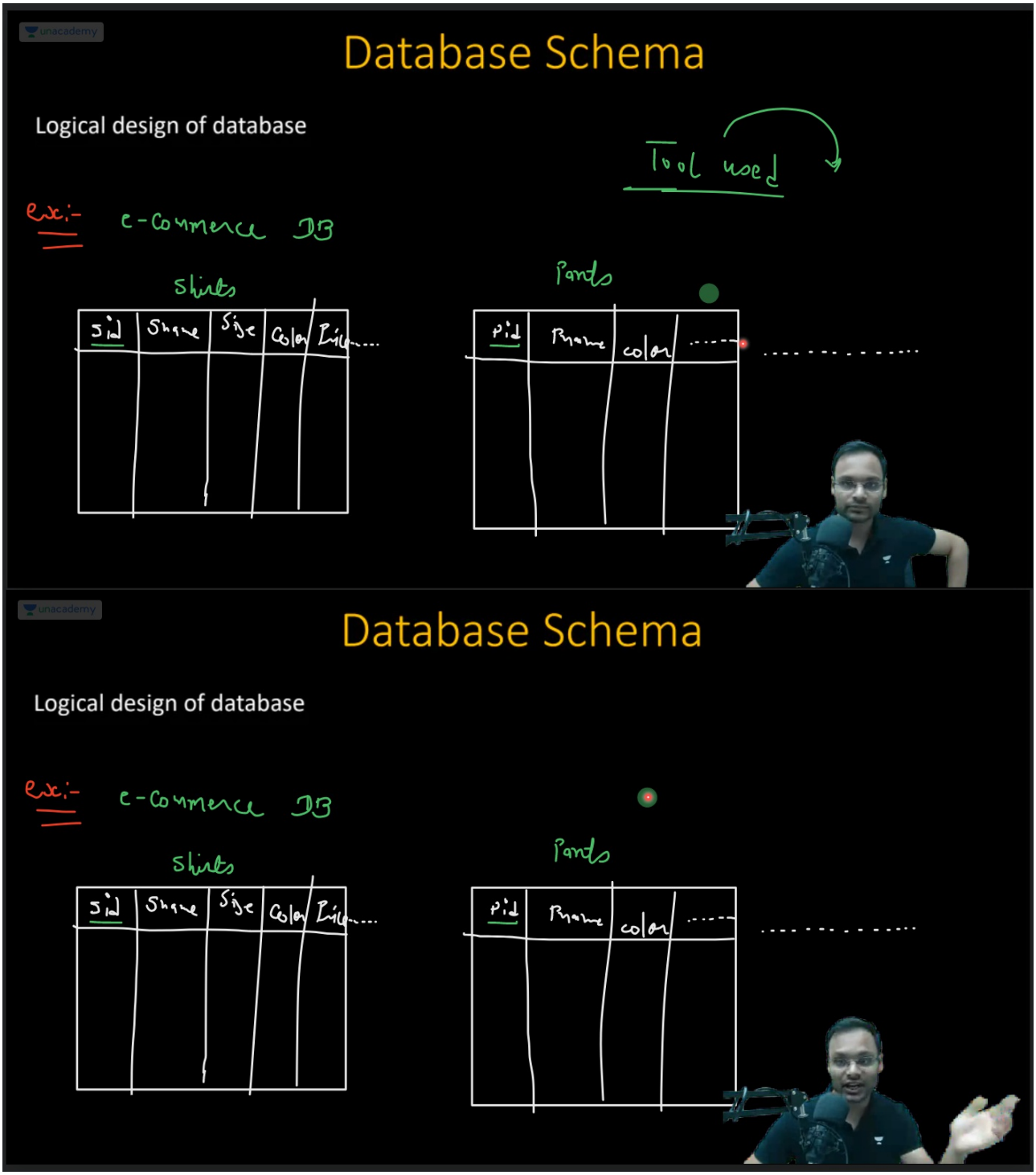


- What is the **cardinality** of the relation?

> **7** as we have **7** rows.

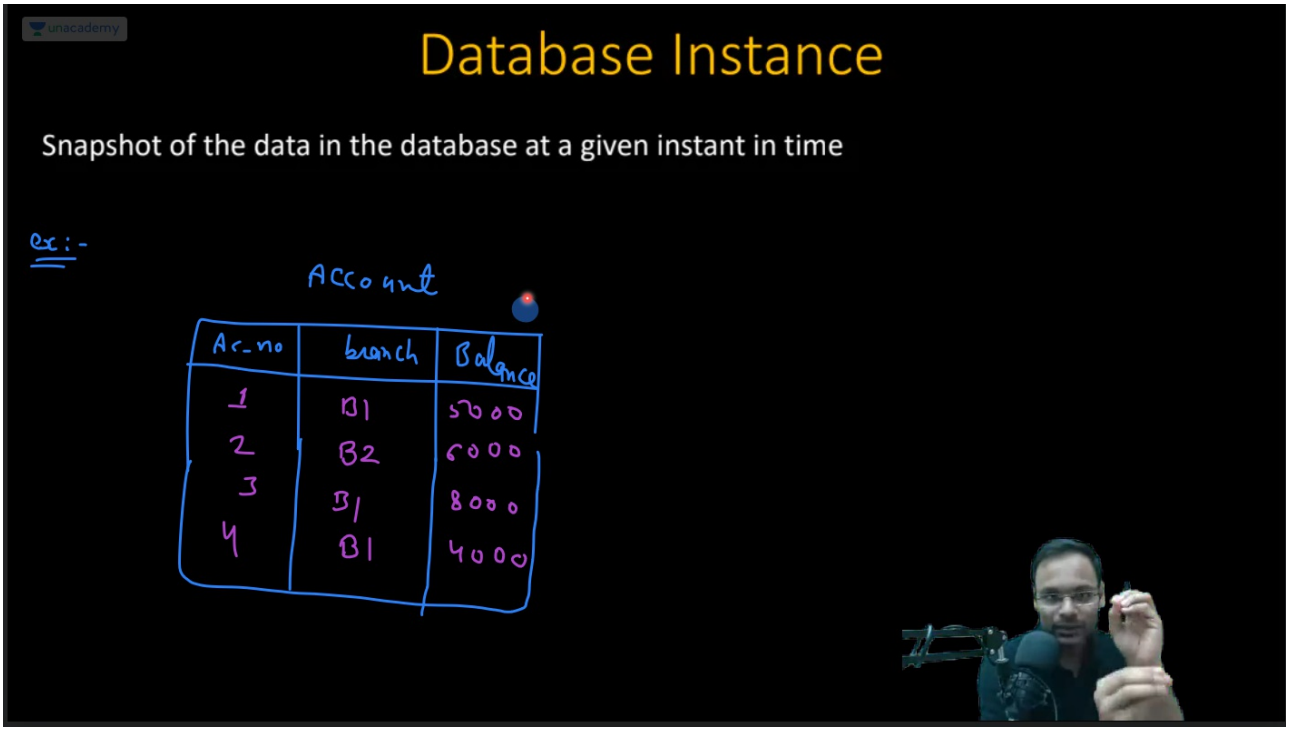
Database Schema

- Tool used.
- Logical Design of database.
- Table is not called as the schema.
- All of the **constraints** that we have put and all of the database construct used to construct the database. All of them combined is called as the **Database Schema**.



Database Instance

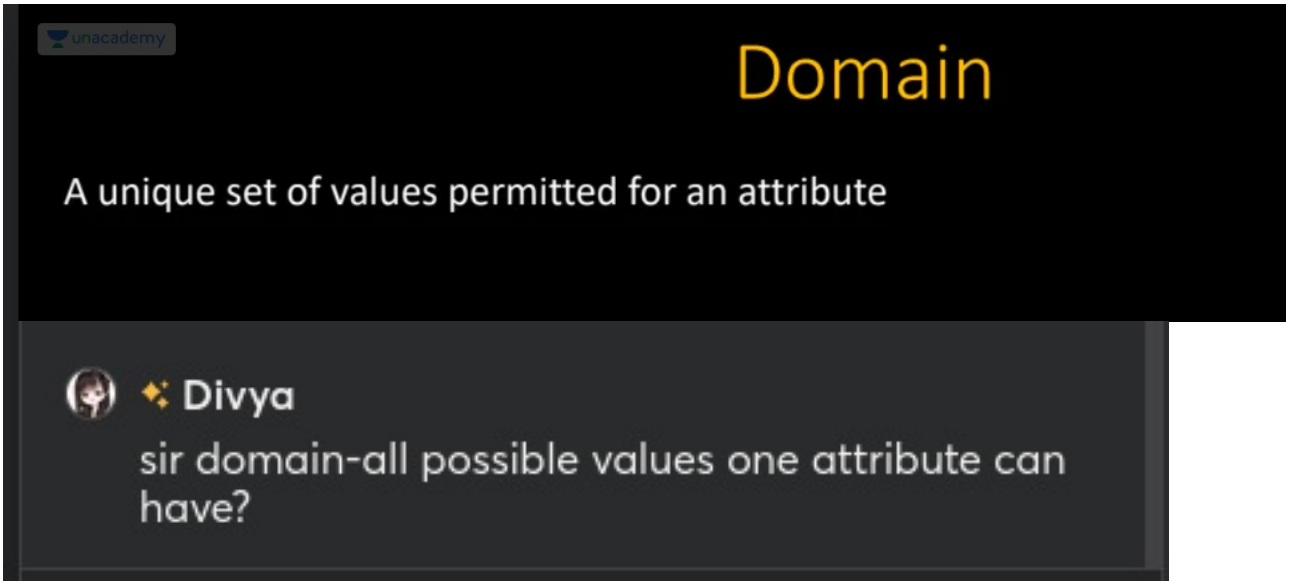
- Snapshot of the data in the datbase at a given instant in time.



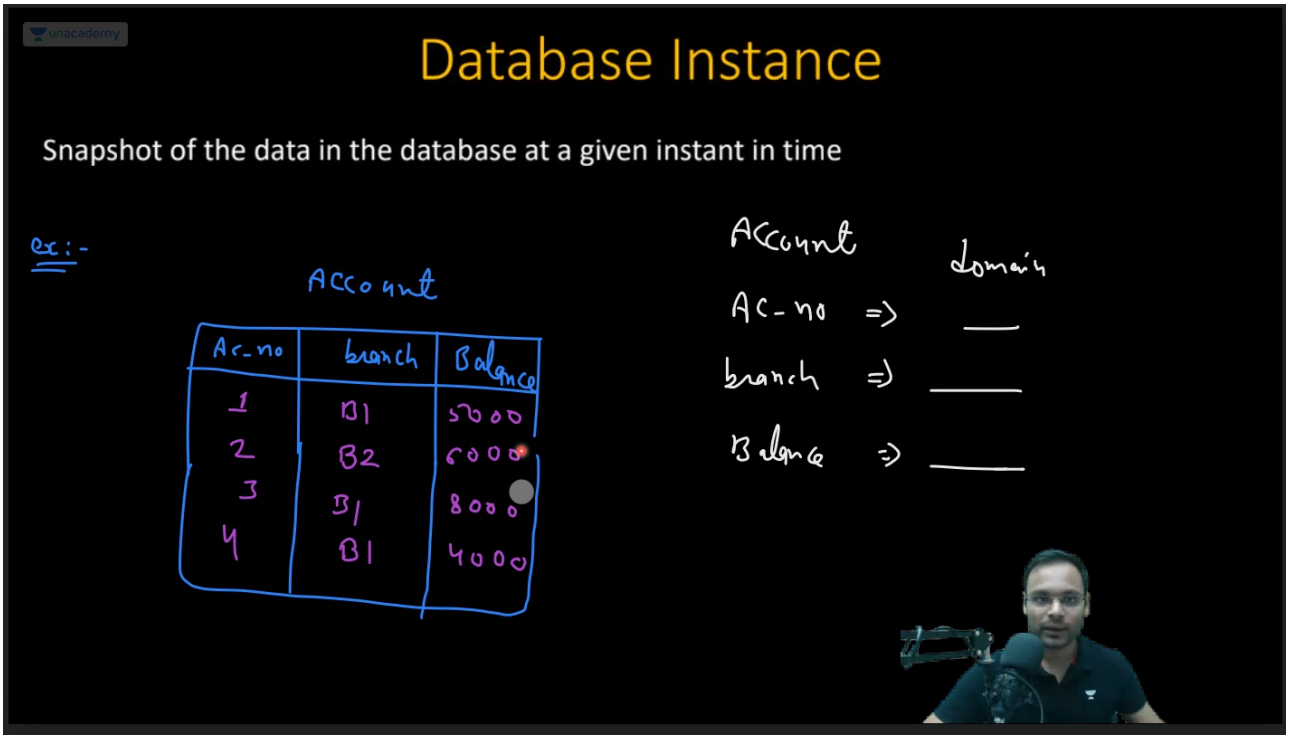
- Snapshot of Database.
- It will change.

Domain

- It is not a **constraint**.
- Domain means when some values comes **what type of value(int, varchar, char)** it will be. Kind of datatype.
- A unique set of values permitted for an attribute.



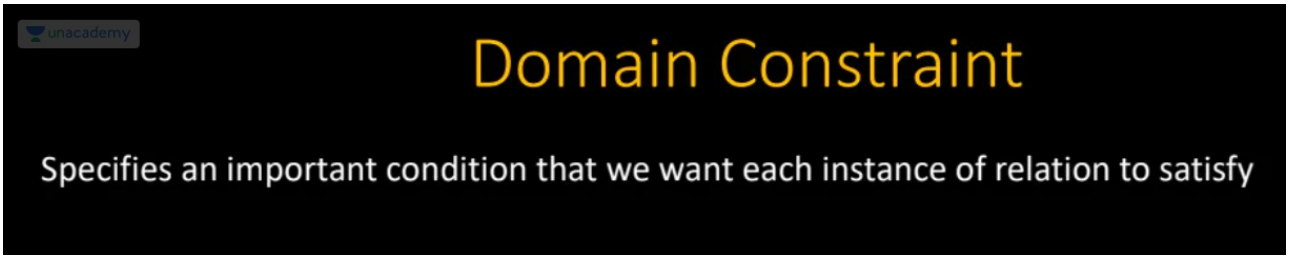
- Exactly



- Difference between **Domain** and **constraint**:-
- Domain -> What is the datatype of the value.
- Constraint -> Even if the datatype of the value is correct, then also the value is **rejected/not allowed**.

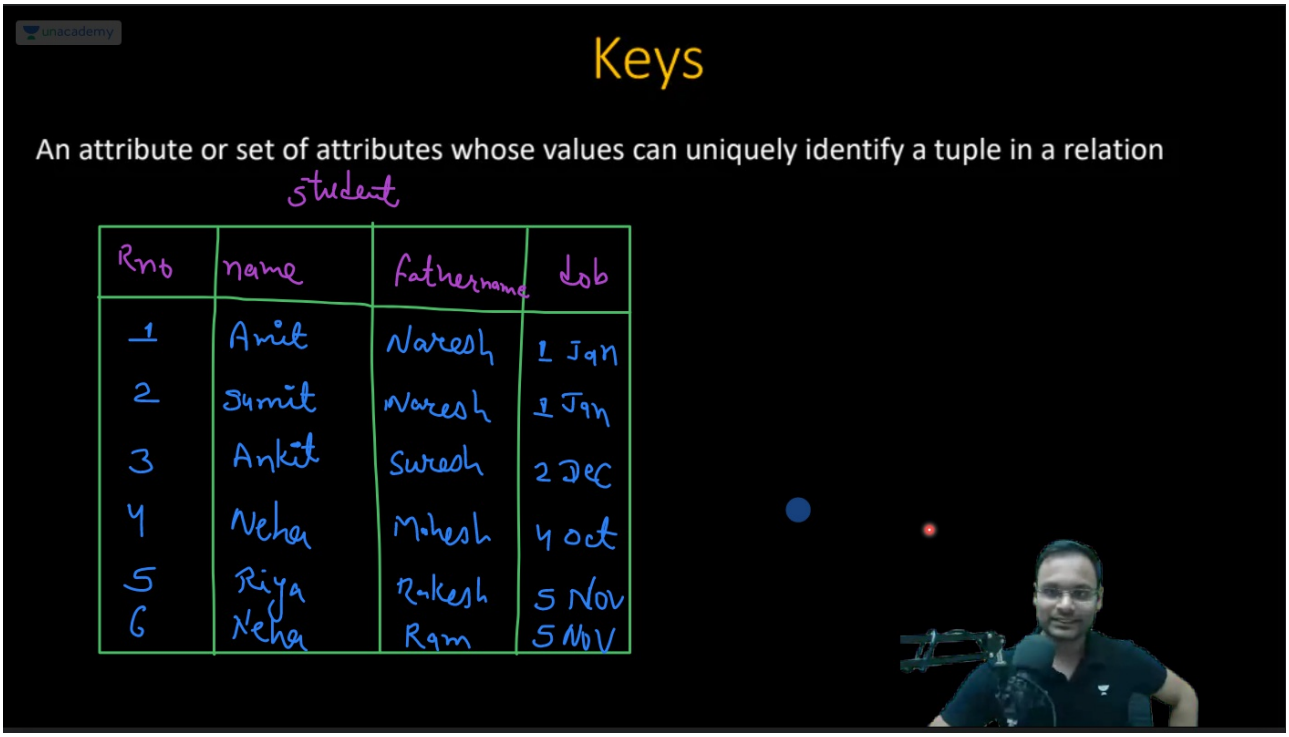
Domain constraint

- Specifies an important condition that we want each instance of relation to satisfy.

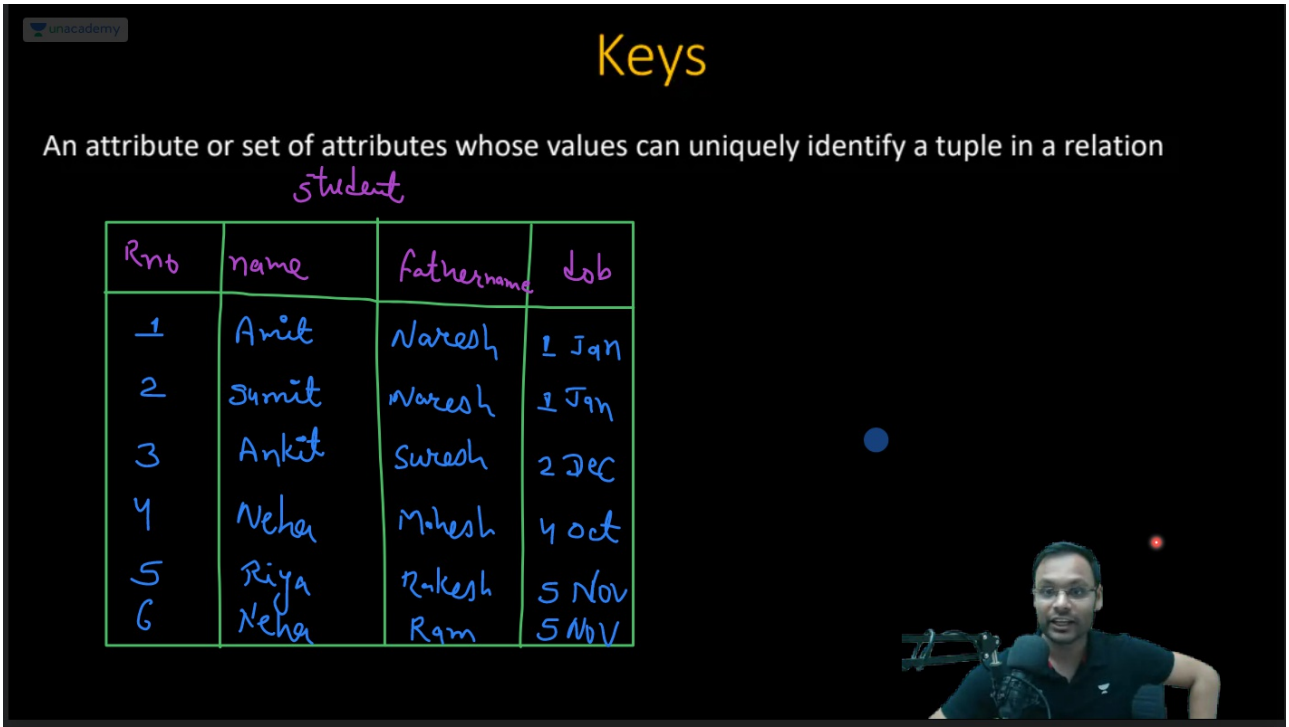


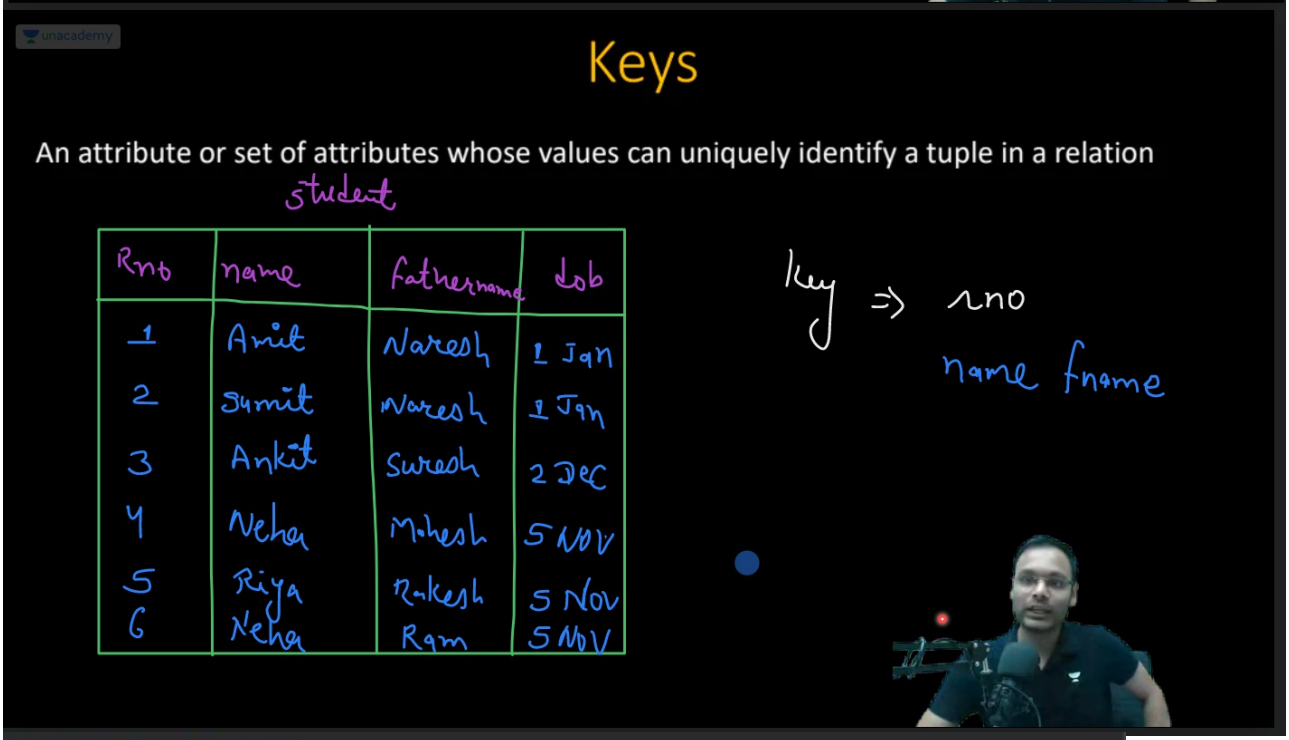
Keys

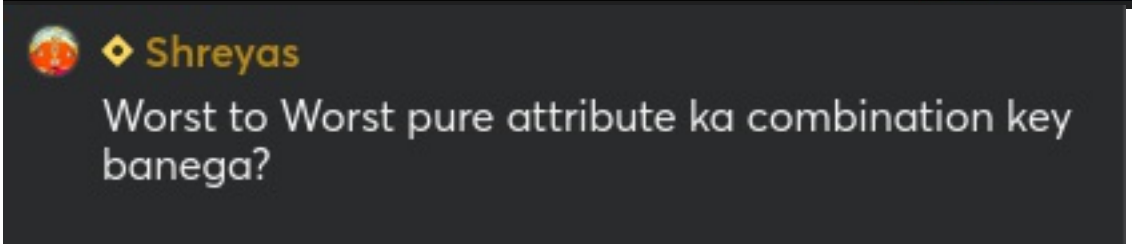
- An attribute or set of attributes whose values can uniquely identify a tuple in a relation.



- For name **Neha**, we are getting **two rows**.
- **Name** column cannot singly identify all rows. So, **Name** column cannot be the **key** of **student** table.
- Key -> Rno, (name + fathername)

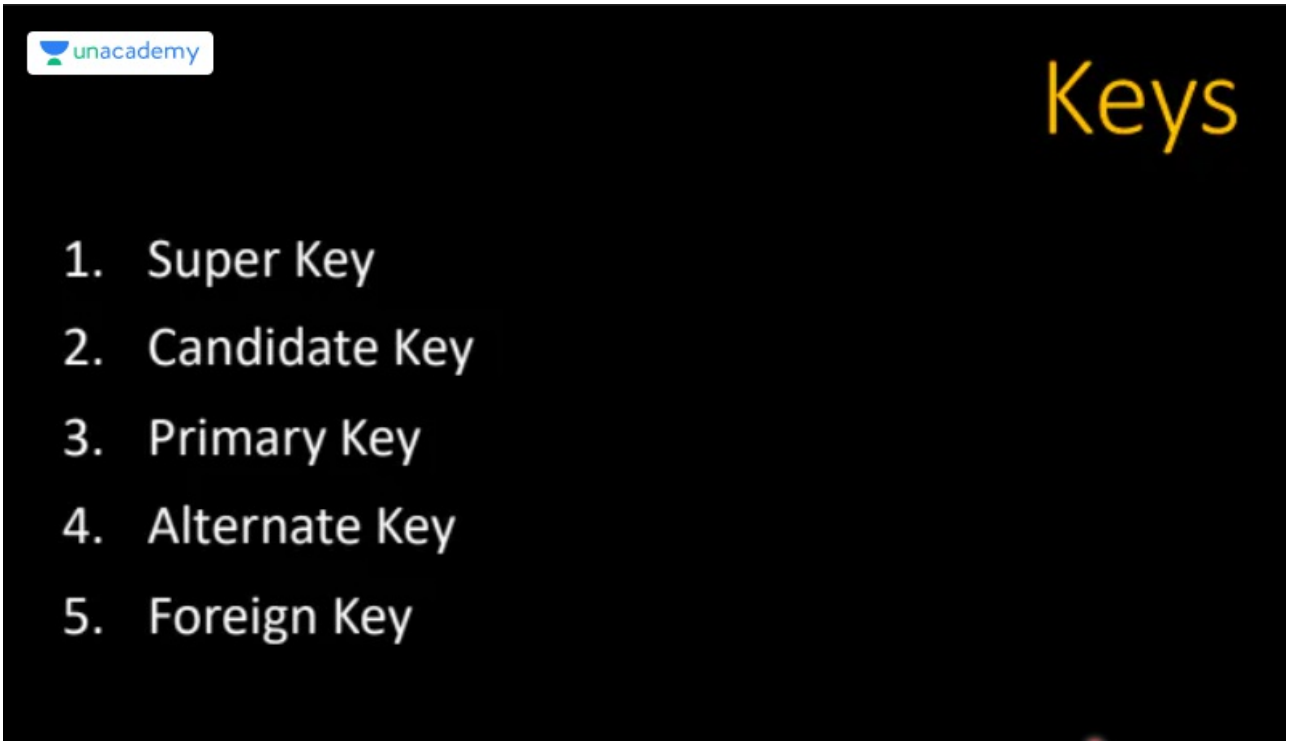






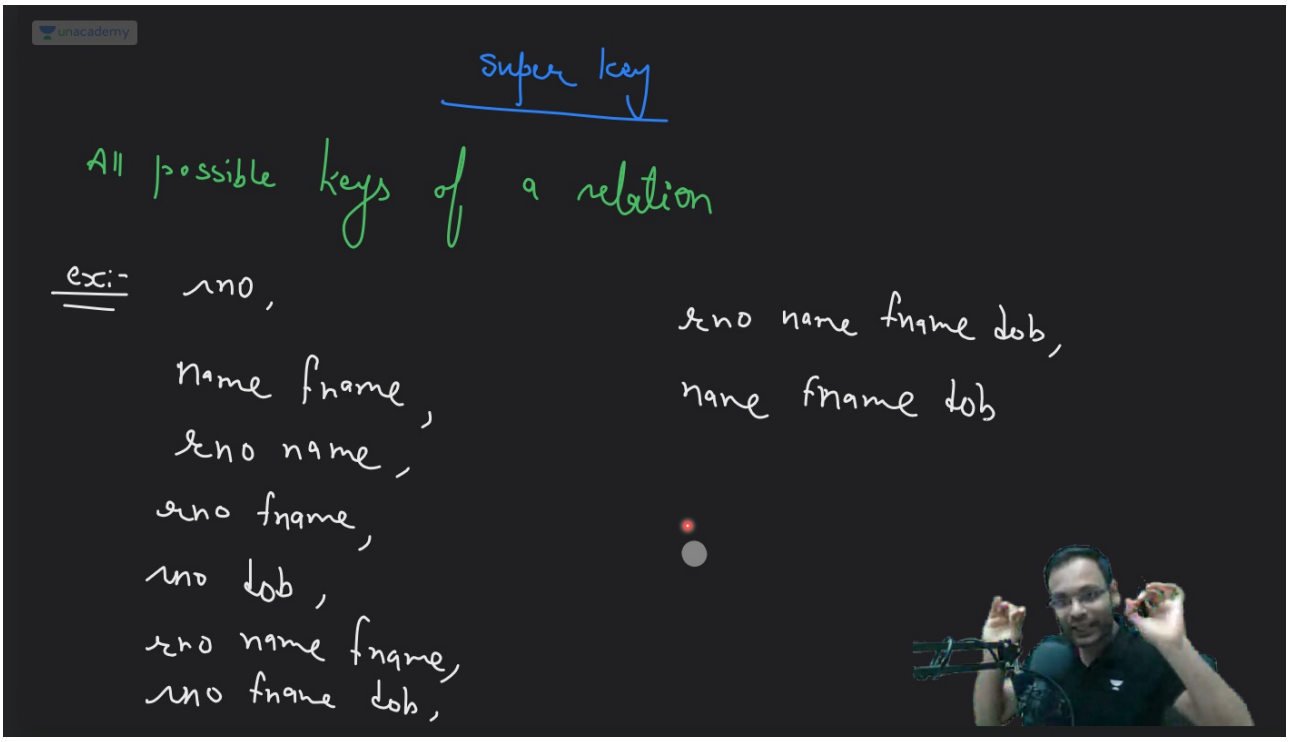
- Yes.

Types of Keys



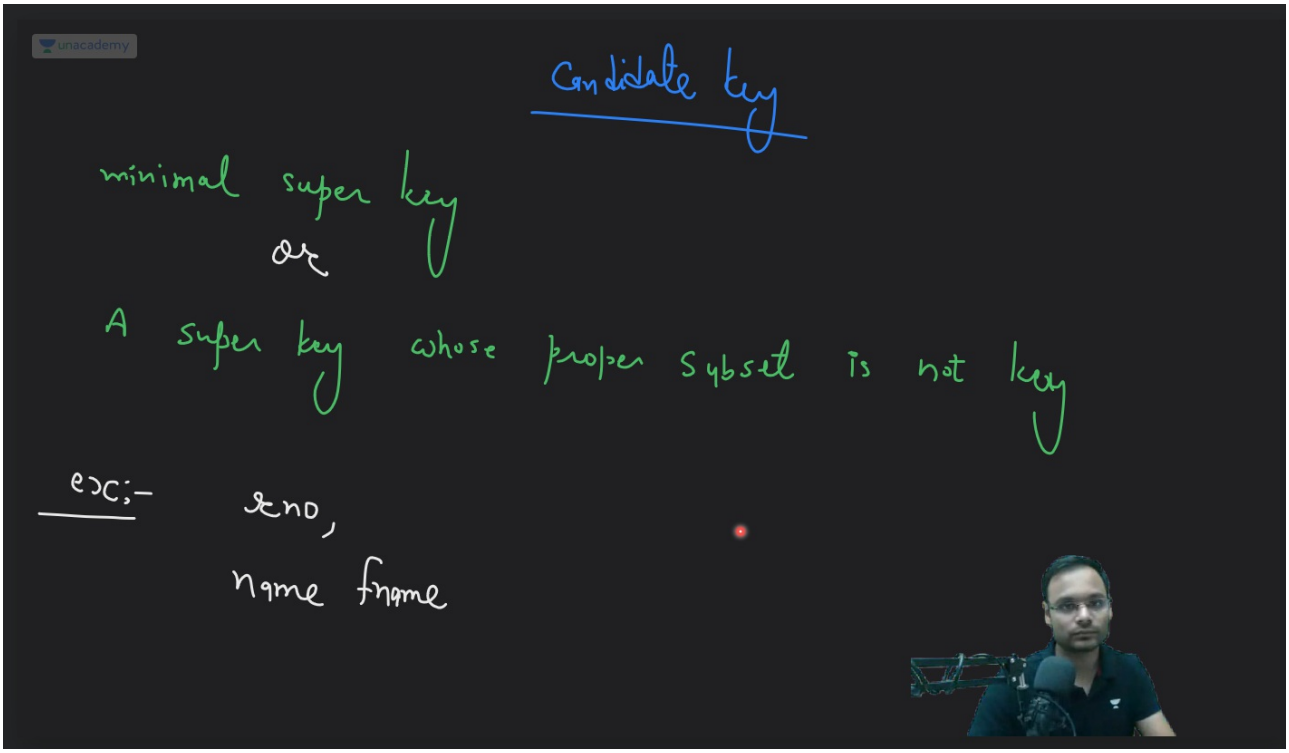
Super Key

- All possible keys of a relation.



- All possible keys combination is making super keys.

Candidate key



- Minimal super key
- A super key whose prope subset is not key
- Example -> rno, name + fname
- **name + fname** combination because the subset of them is **name and fname** and none of them are **keys** individually. That's why they are **candidate keys**.

unacademy

ex:-

Relation ⇒ R

Attributes => A, B, C, D, E, F

candidate key ⇒ ABC

A
B
C
AB
AC
BC

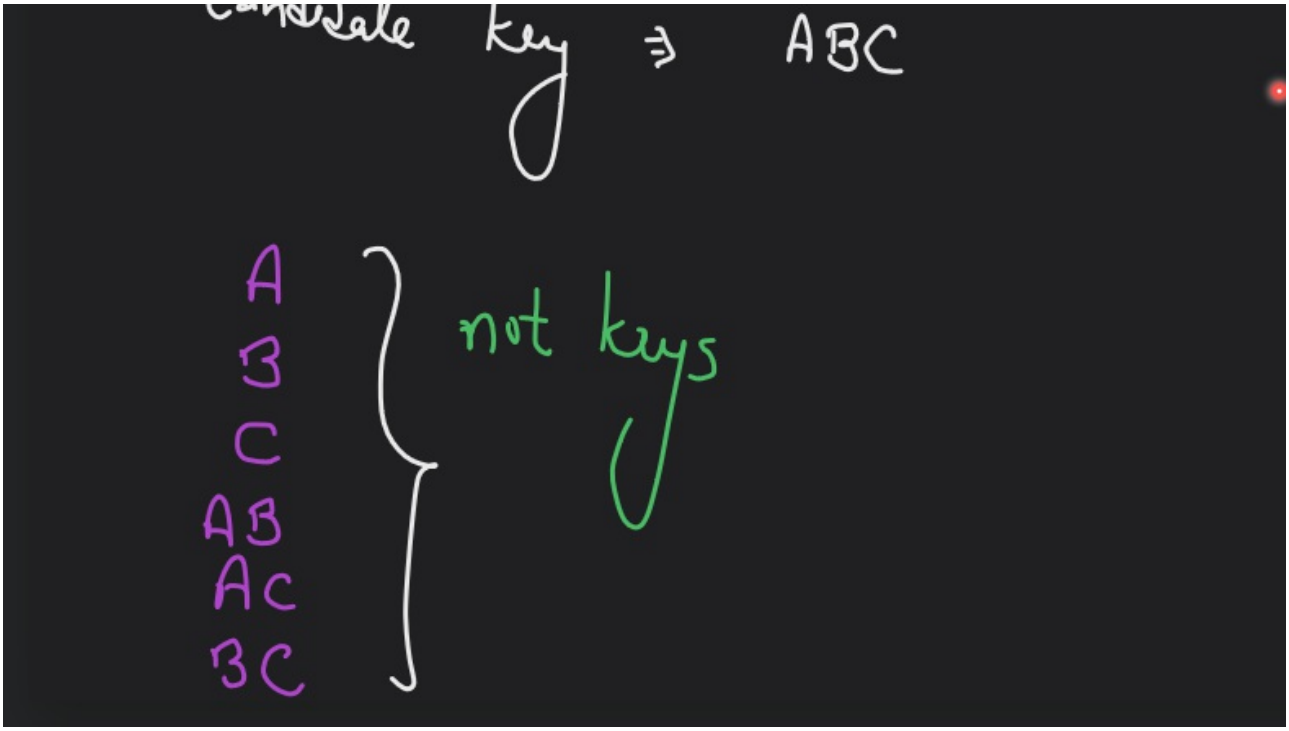
} not keys

unacademy

ex:-

Relation ⇒ R

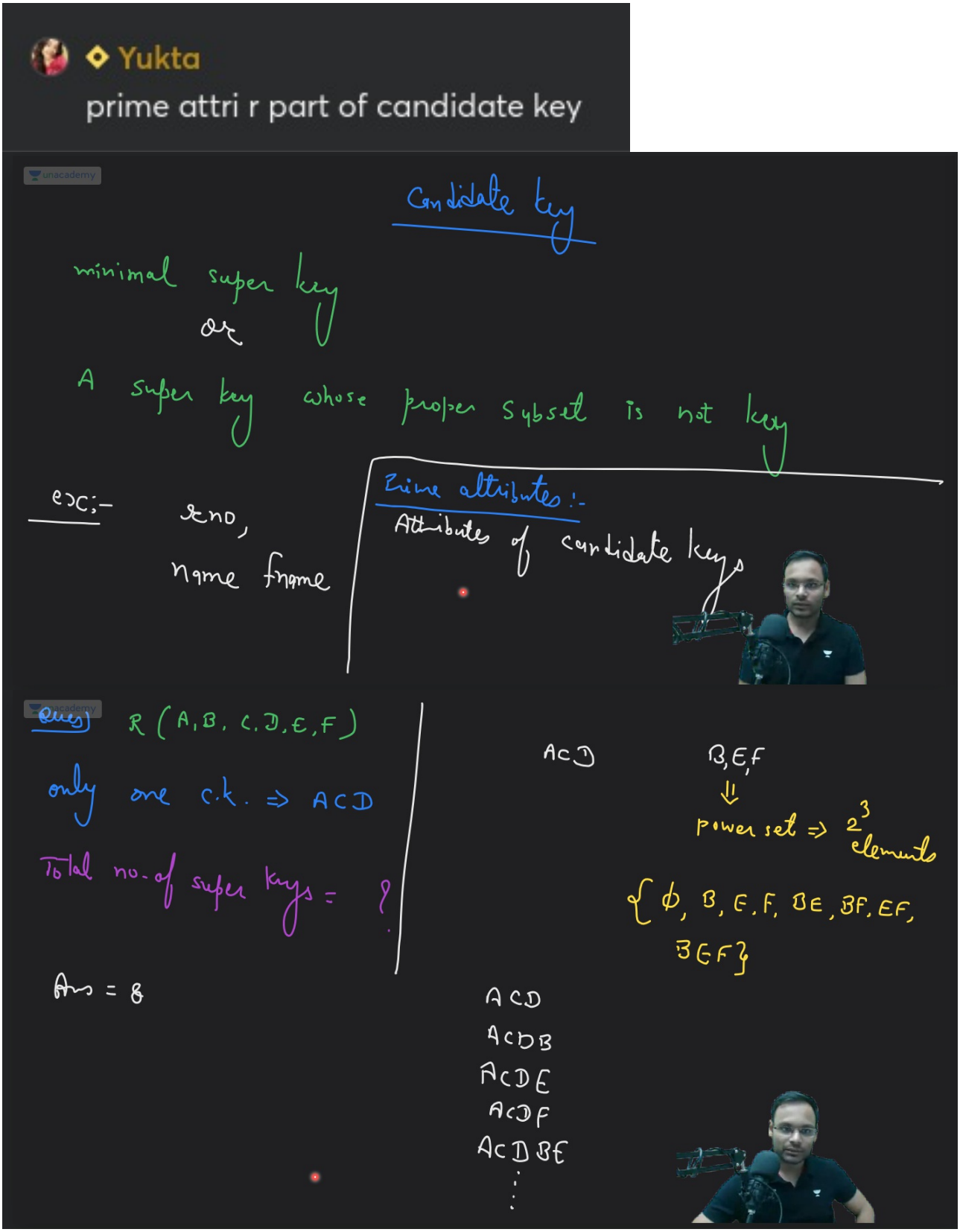
Attributes => A, B, C, D, E, F



- If **ABC** is a **candidate key** then **A, B, C , AB, AC, BC** are not **keys**.
- **ABCD** is a **super key** but not **candidate key**.

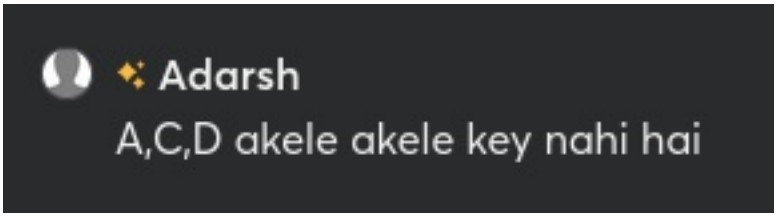
Prime Attributes:-

- Attributes of candidate keys.

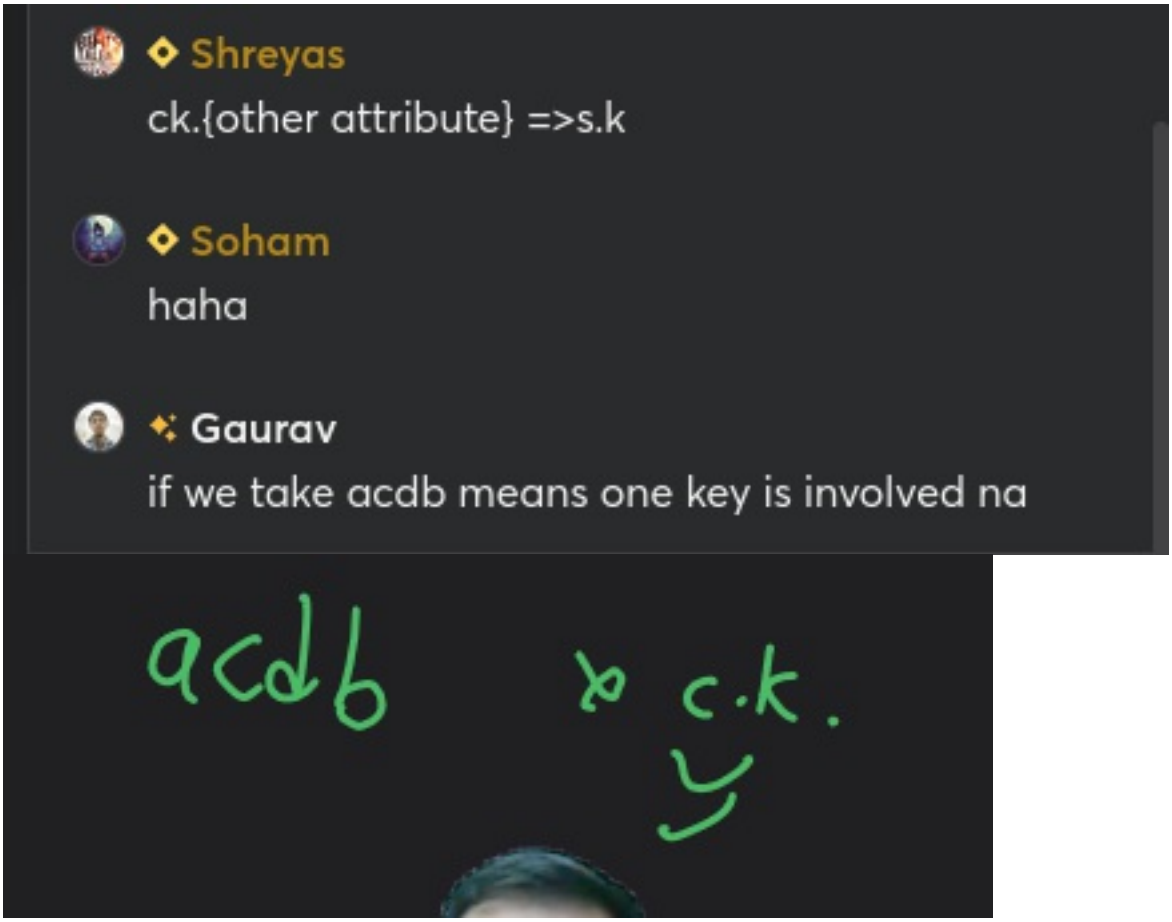


- Question.
- Total no. of super keys which are not candidate keys?

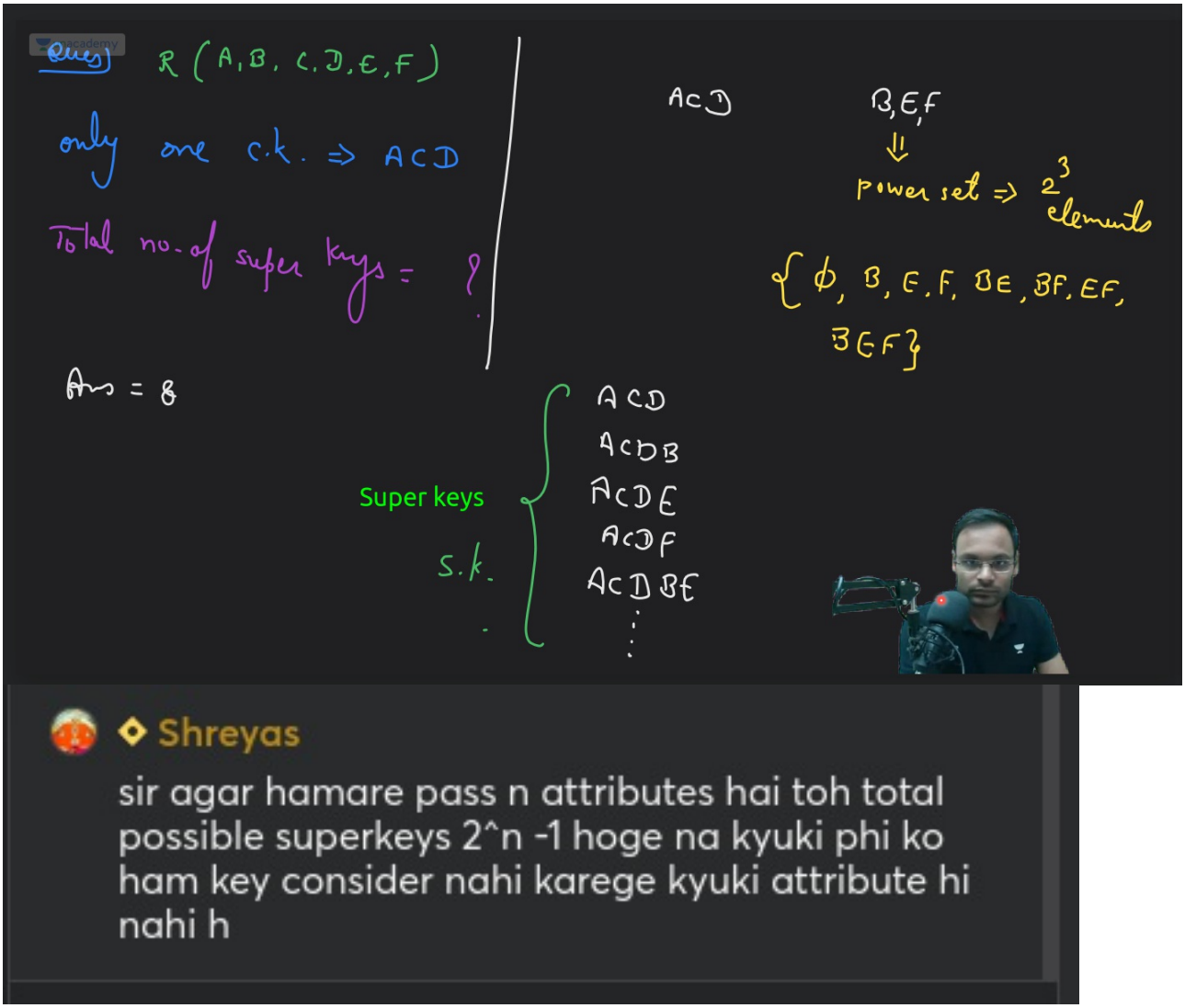
7, except **ACD**, all of them.



- Yes.



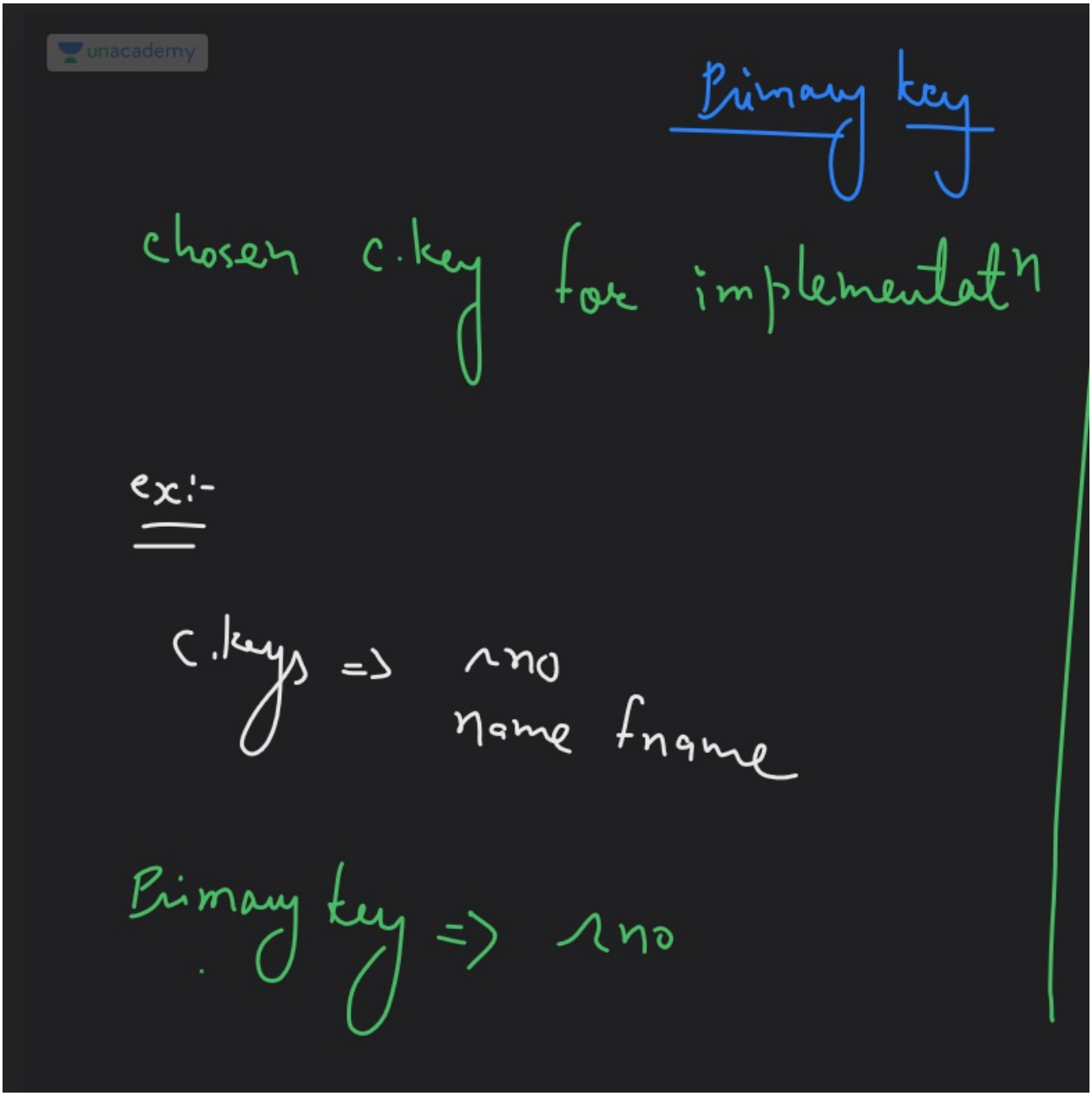
- ACDB -> Not a candidate key as **ACD** alone is a key. **ACDB** can be a **super key**, yes.



- possible.

Primary Key

- One table has only one **primary key**.
- Chosen candidate key for implementation.
- Example -> rno, name + fname
- Primary Key -> rno



Alternate key

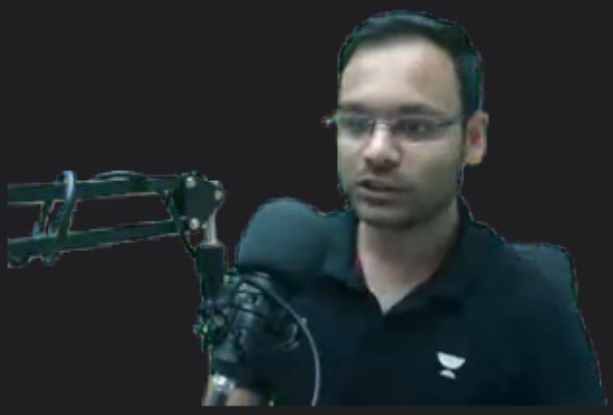
- All other candidate keys apart from primary key
- Example -> name, fname

Alternate key

all other candidate keys apart from primary key

ex:-

name fname



unacademy

Primary key

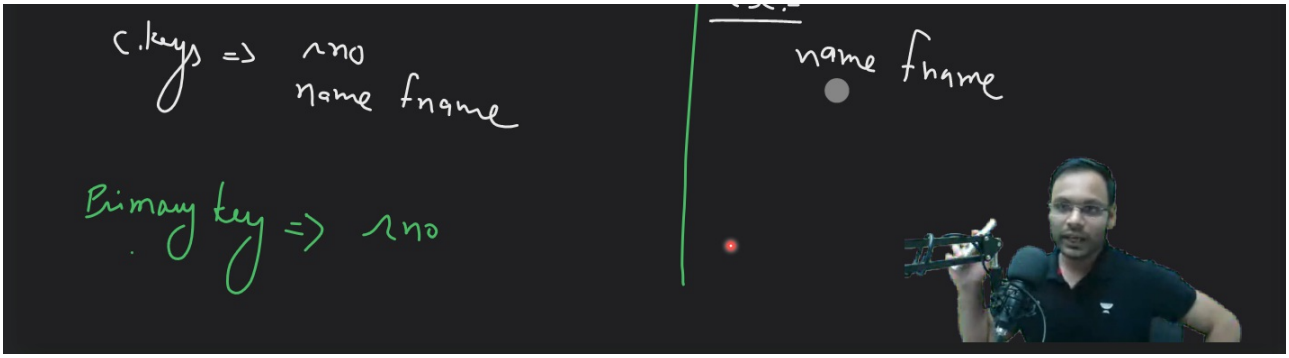
chosen c.key for implementatn

ex:-

Alternate key

all other candidate keys apart from primary key

ex:-



Foreign Key

Foreign key

Branch

Branch_id	Branch_name
B1	MG Road
B2	Indira nagar
B3	New road

Account

AC_no.	Branch_id	Amount
1	B1	5000
2	B2	7000
3	B1	9000
4	~~B9~~	5000

Foreign key

Branch

Branch_id	Branch_name
B1	MG Road
B2	Indira nagar
B3	New road

Account

AC_no.	Branch_id	Amount
1	B1	5000
2	B2	7000
3	B1	9000
4	~~B9~~	5000

Foreign key

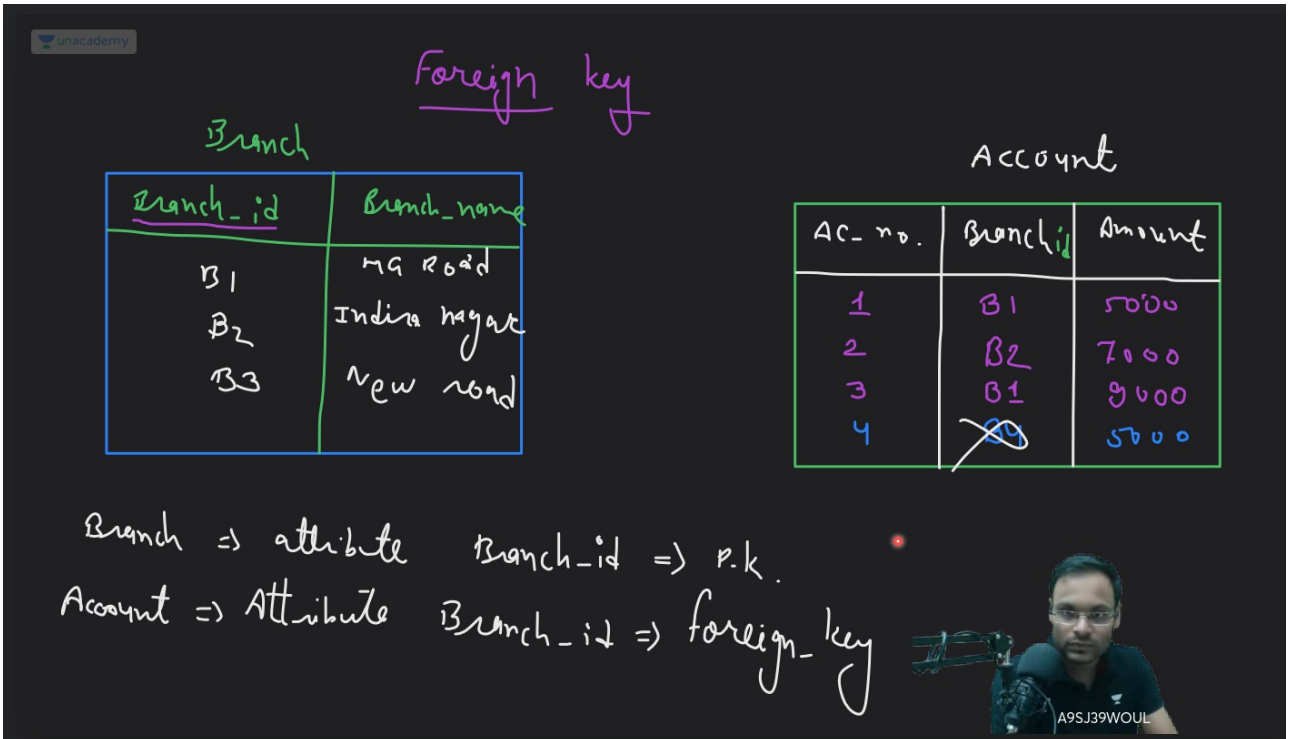
Branch

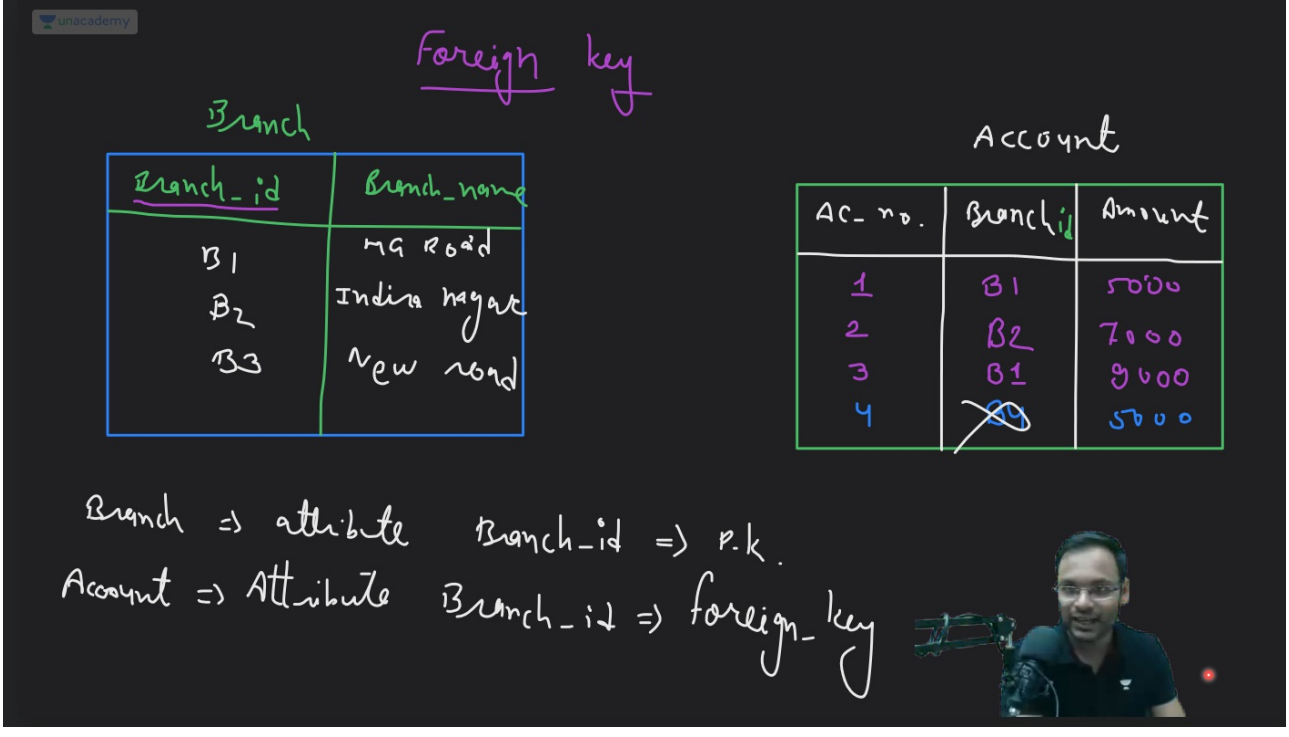
Branch_id	Branch_name
B1	MG Road
B2	Indira nagar
B3	New road

Account

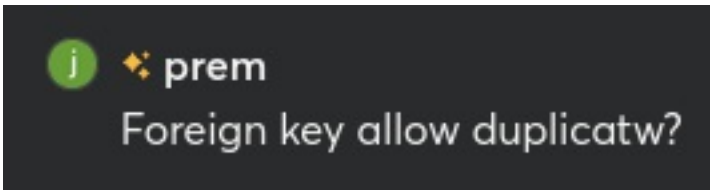
AC_no.	Branch_id	Amount
1	B1	5000
2	B2	7000
3	B1	9000
4	~~B9~~	5000

The constraint put on the **Account.Branch_id** column is called as **Referencial Integrity constraint**.

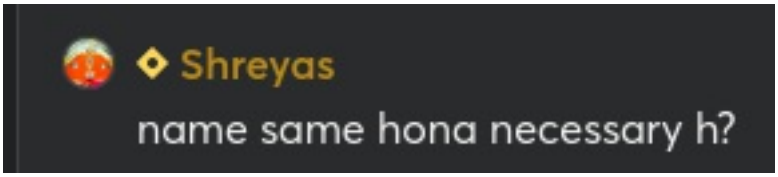




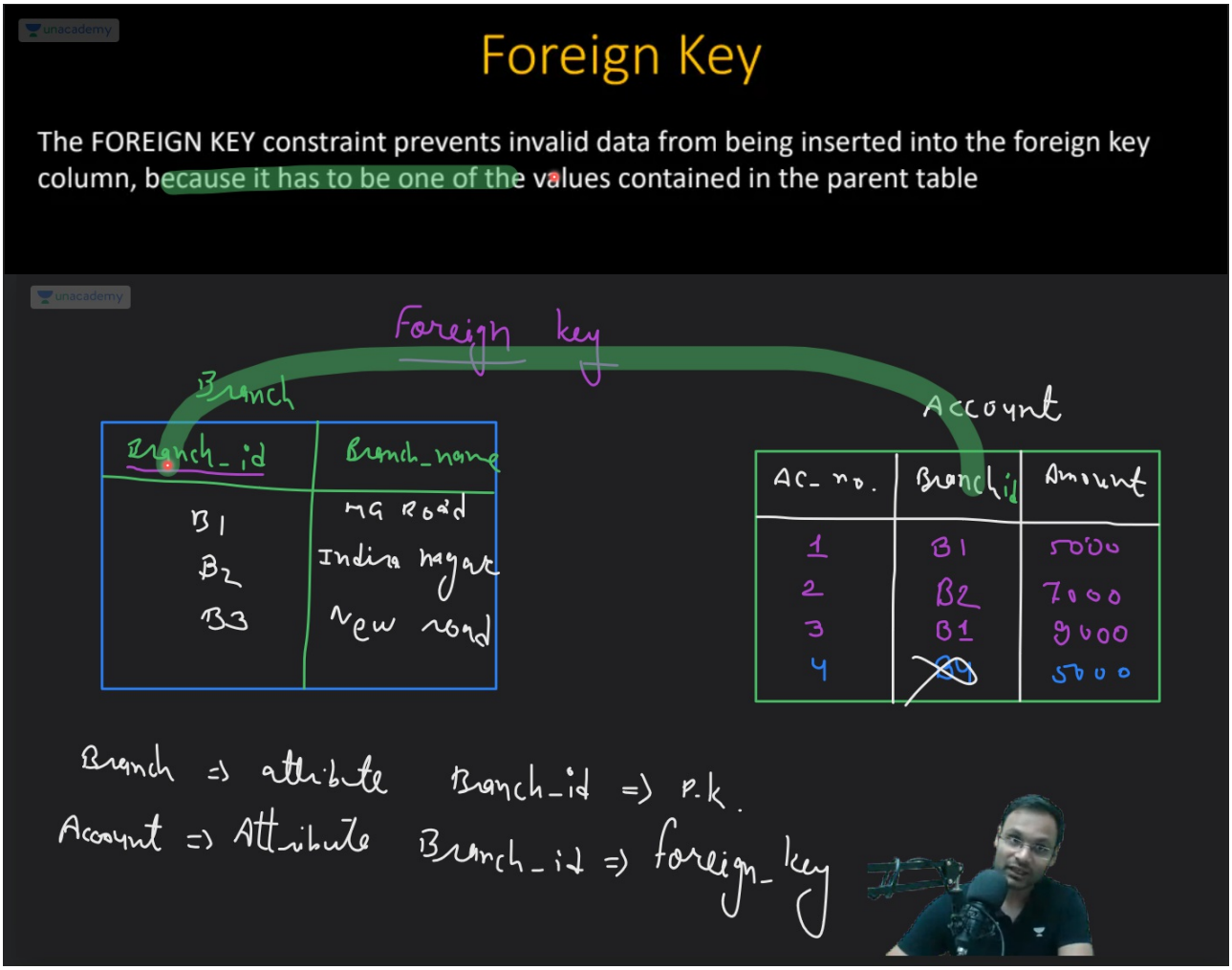
- The value we are referring from should be a **primary key** of that table. **Compulsory**.
- The one who is referring the value is called as **foreign key**.
- Branch table -> Attribute which is **Branch_id** -> Primary Key.
- Account table -> Attribute which is **Branch_id** -> Foreign Key.



- Ofcourse

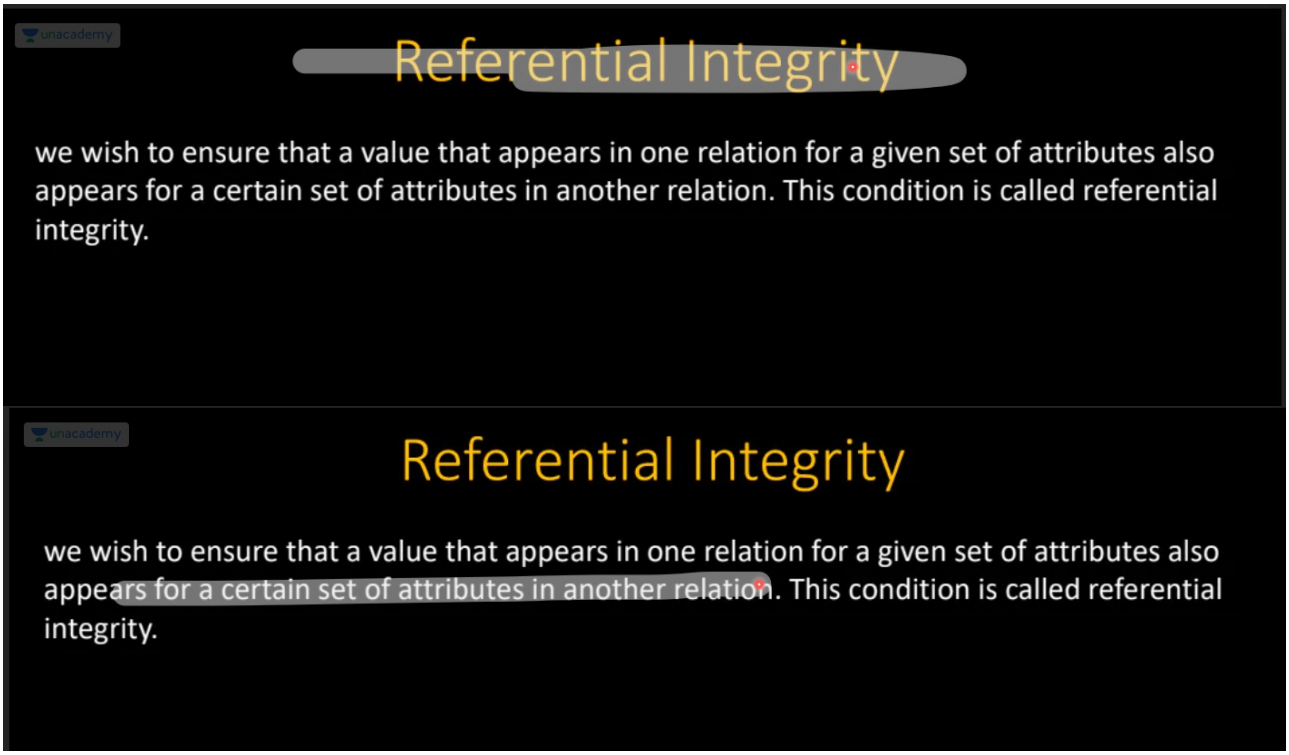


- Not necessary but easier to remember if same name given.

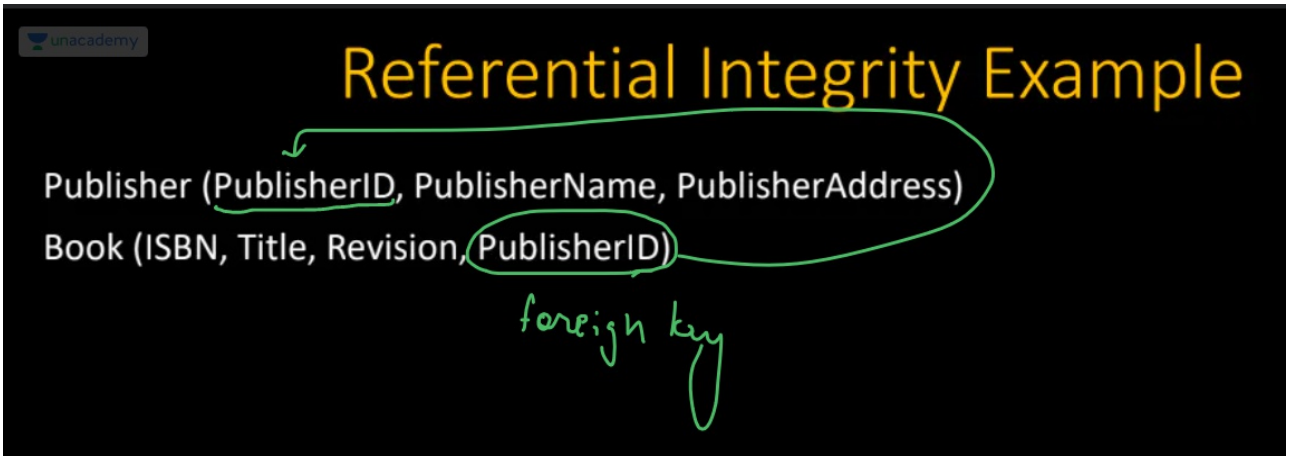


- Branch table is **parent** table.
- The **FOREIGN KEY** constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

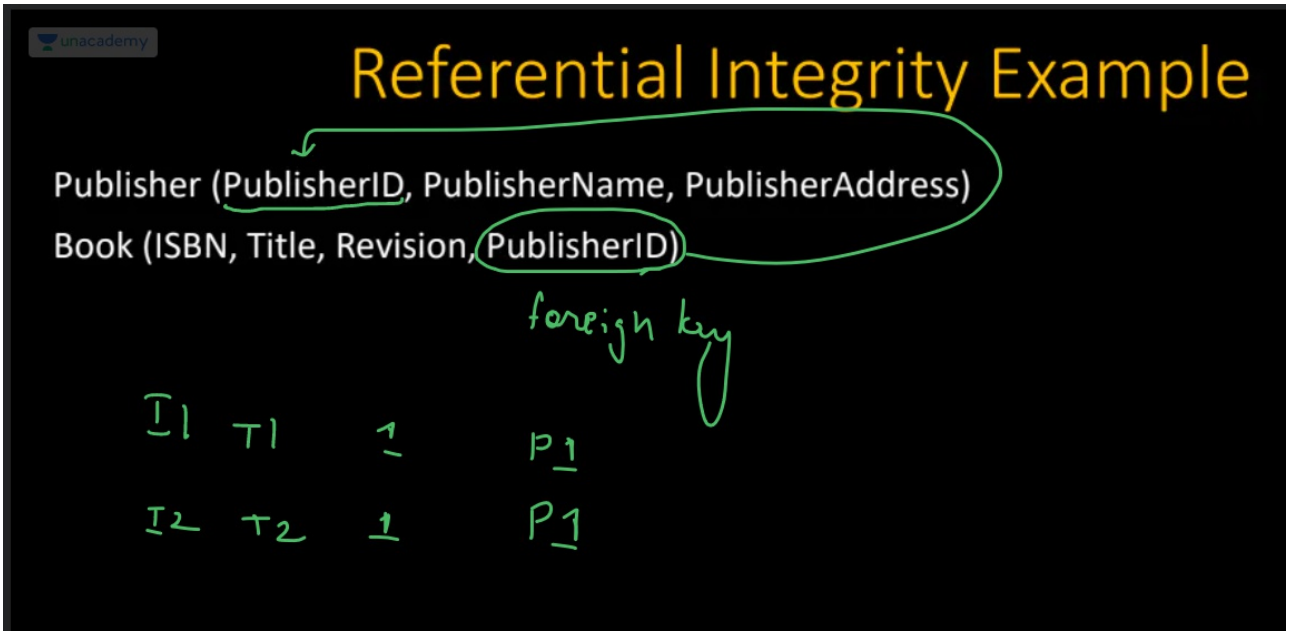
Referencial Integrity constraint



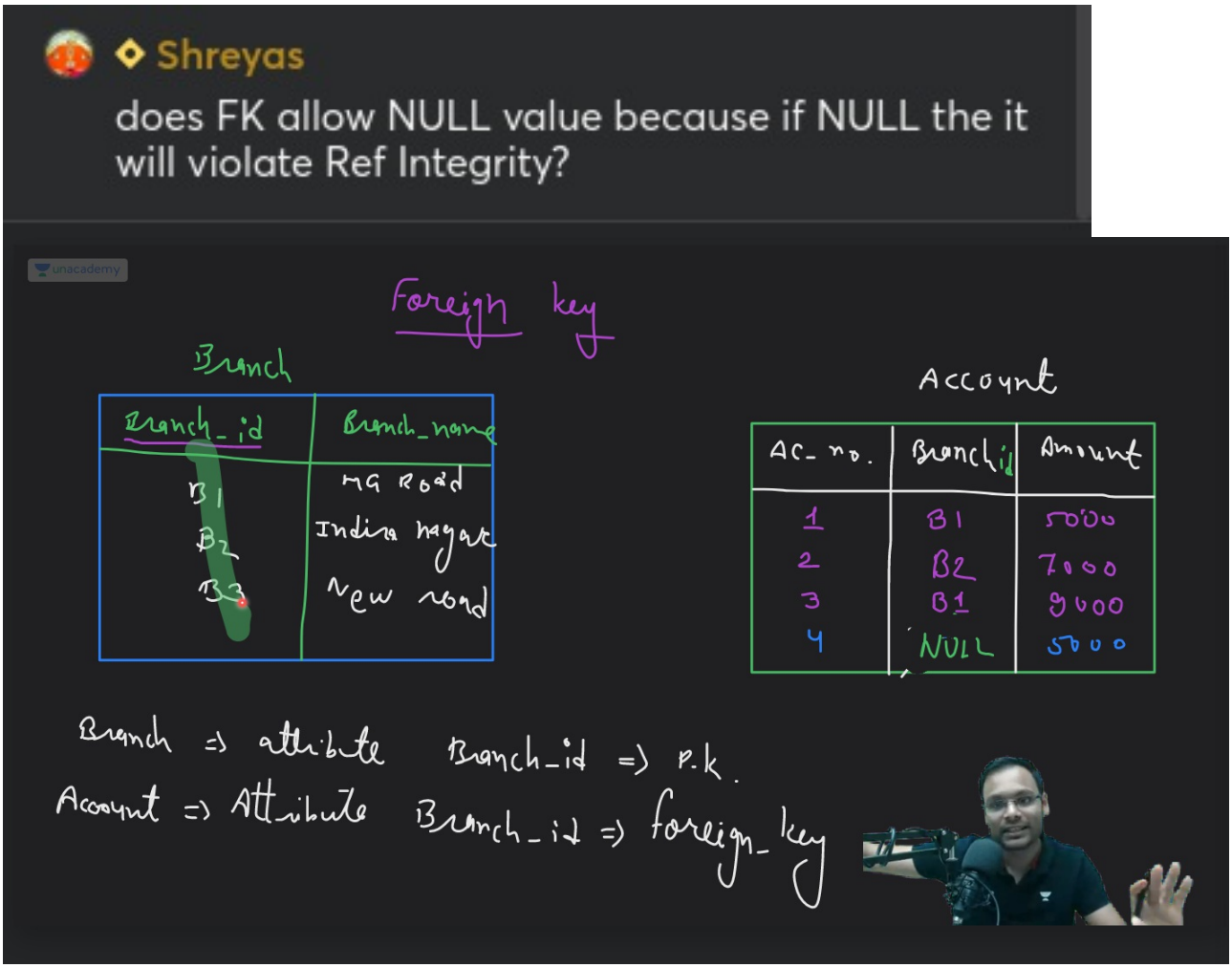
Referencial Integrity Example



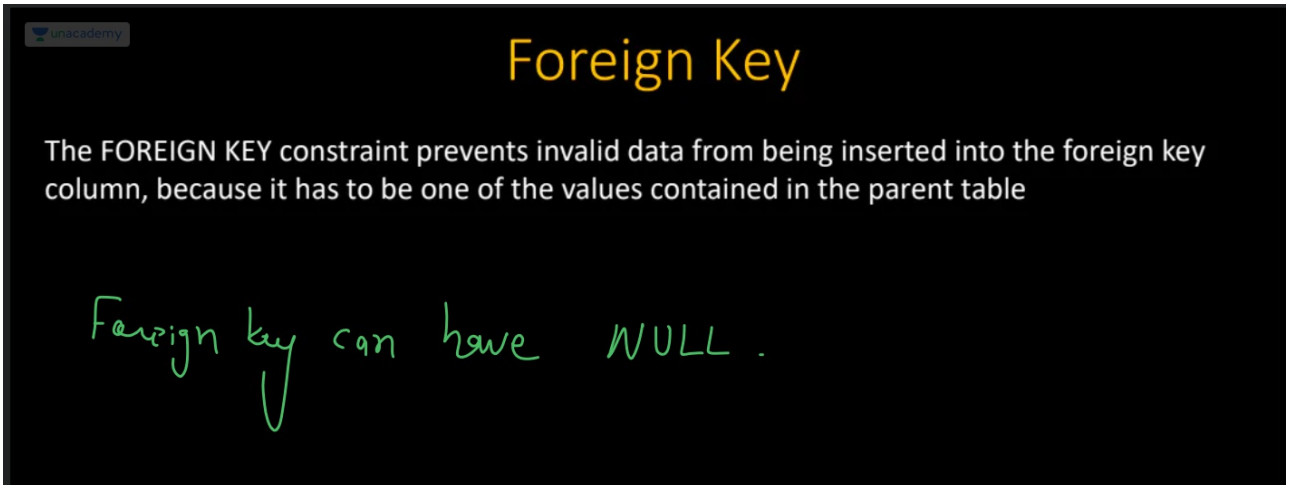
- Publisher.PublisherID -> Primary Key.
- Book.PublisherID -> Foreign Key.



- Foreign Key may not be **unique**.



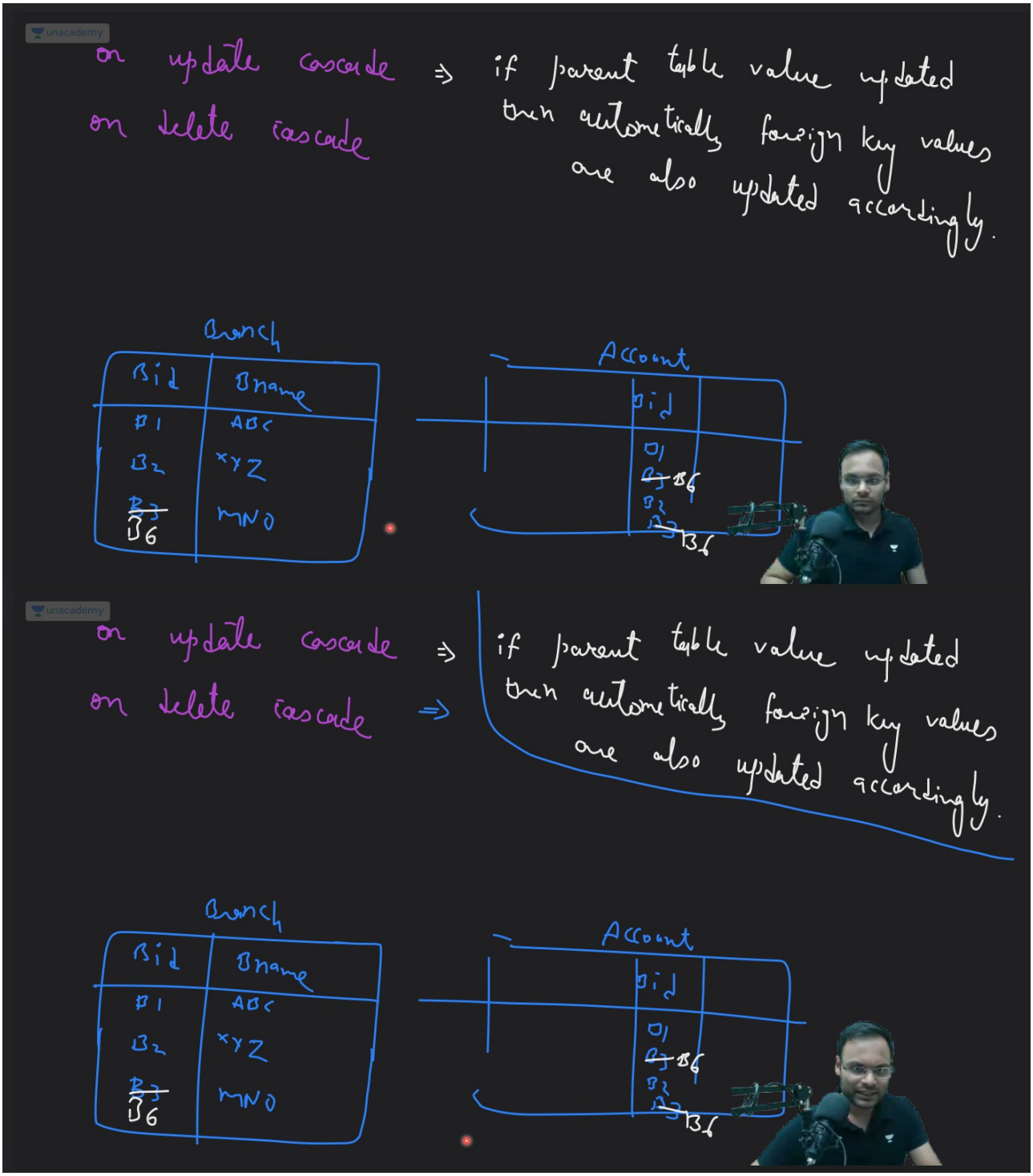
- Integrity constraint is still maintained.
- Foreign Key can have **NULL**.



Cascading effect

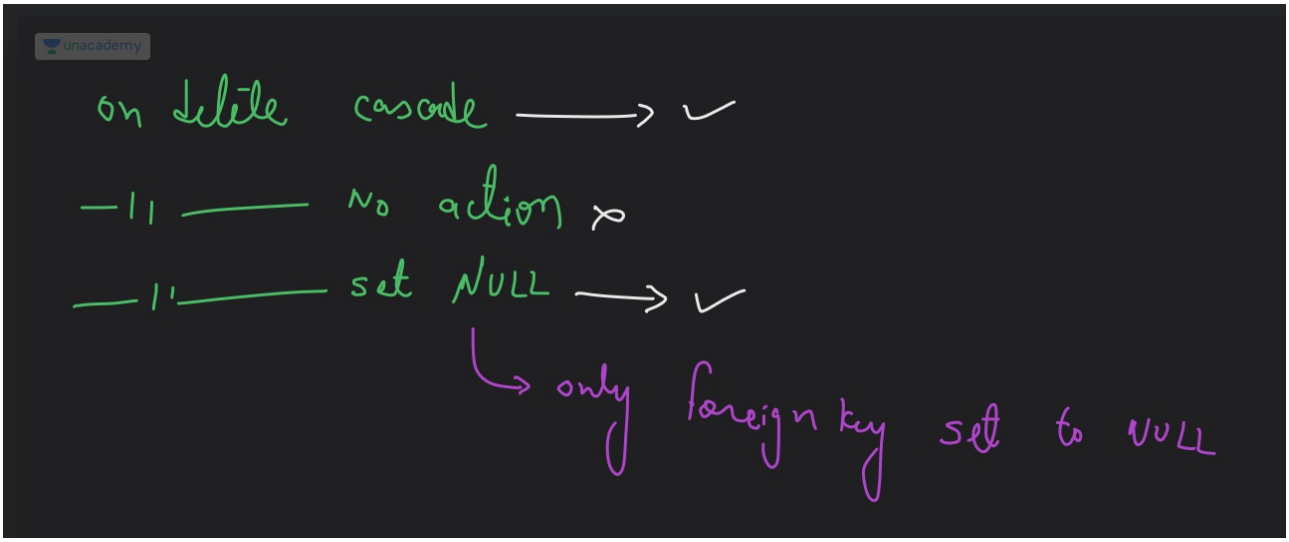
On update cascade

- It says that foreign key value refers to which ever parent table value, if there is **updation** at tha parent table then at the **foreign key** updation will also happen.
- DBMS will do it **automatically**.
- If parent table value updated then automatically foreign key values are also updated accordingly.

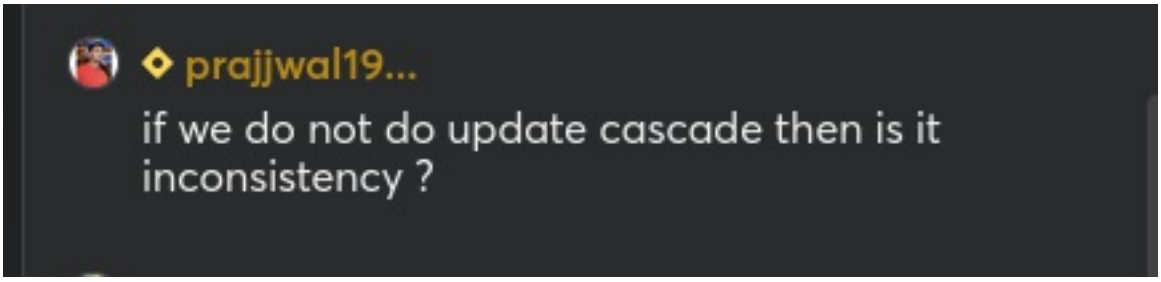


On delete cascade

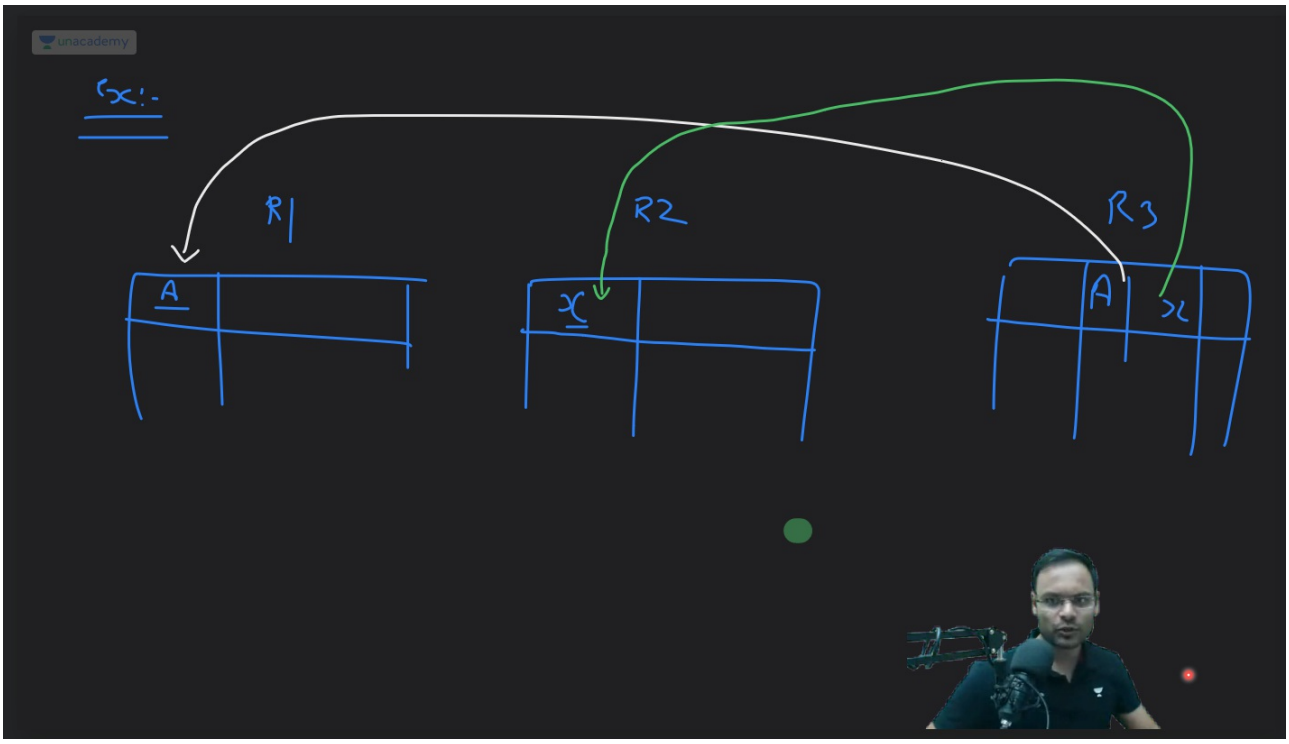
- On delete cascade -> Careful implementation
- On delete No action -> Makes no sense.
- On delete set NULL -> Careful implementation -> Only foreign key set to **NULL**.



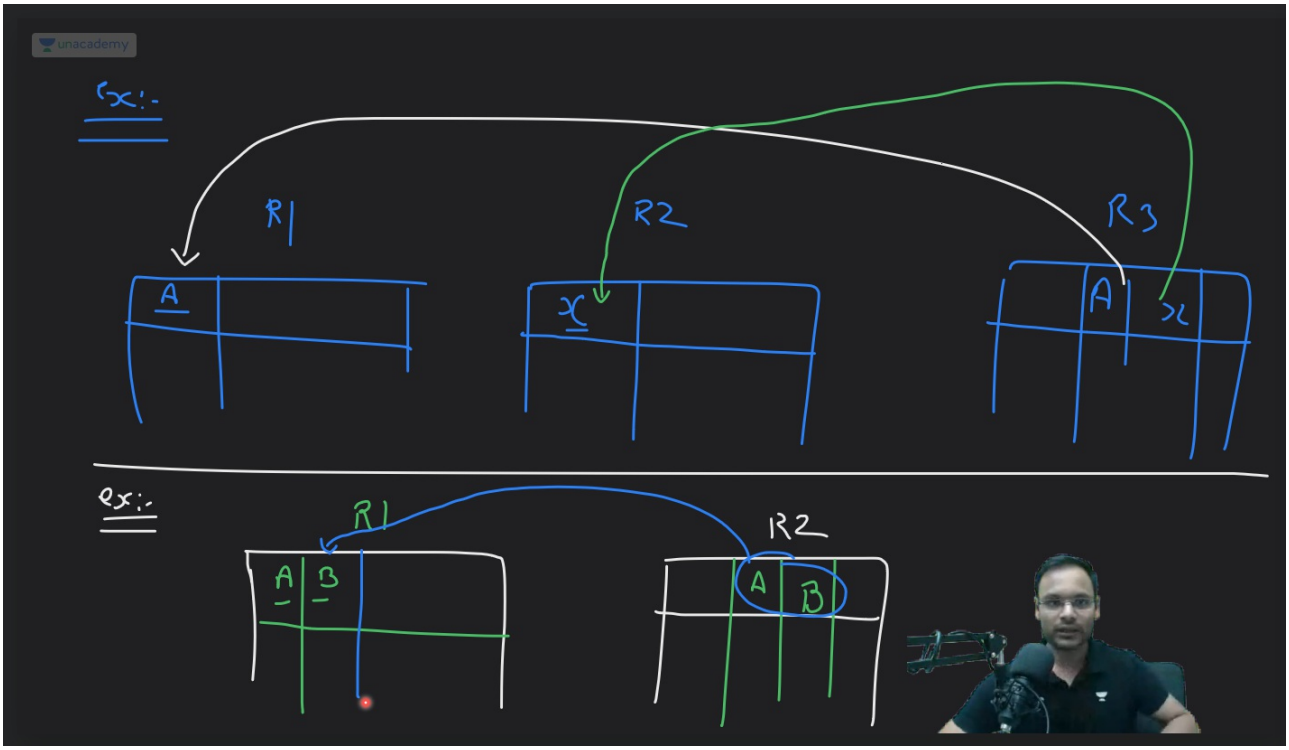
- More cascade options.



- Yes.



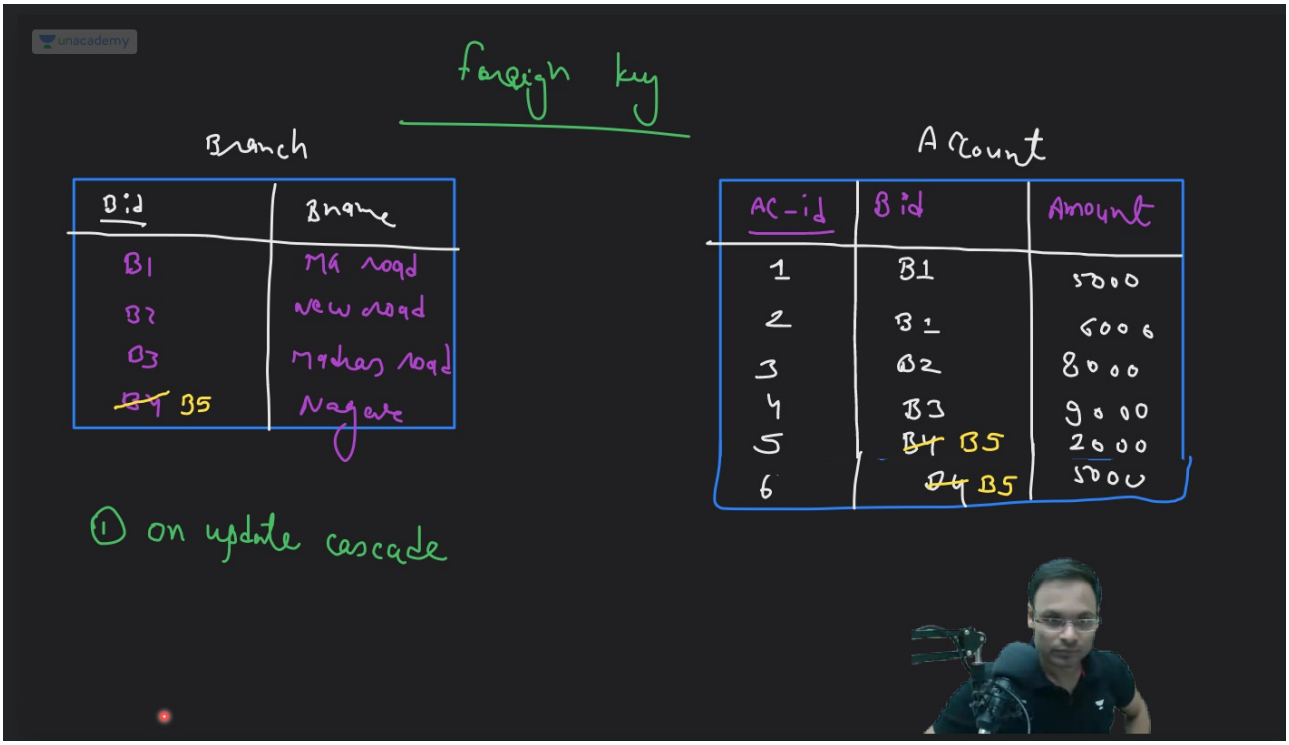
- R1.A -> Primary Key
- R2.X -> Primary Key



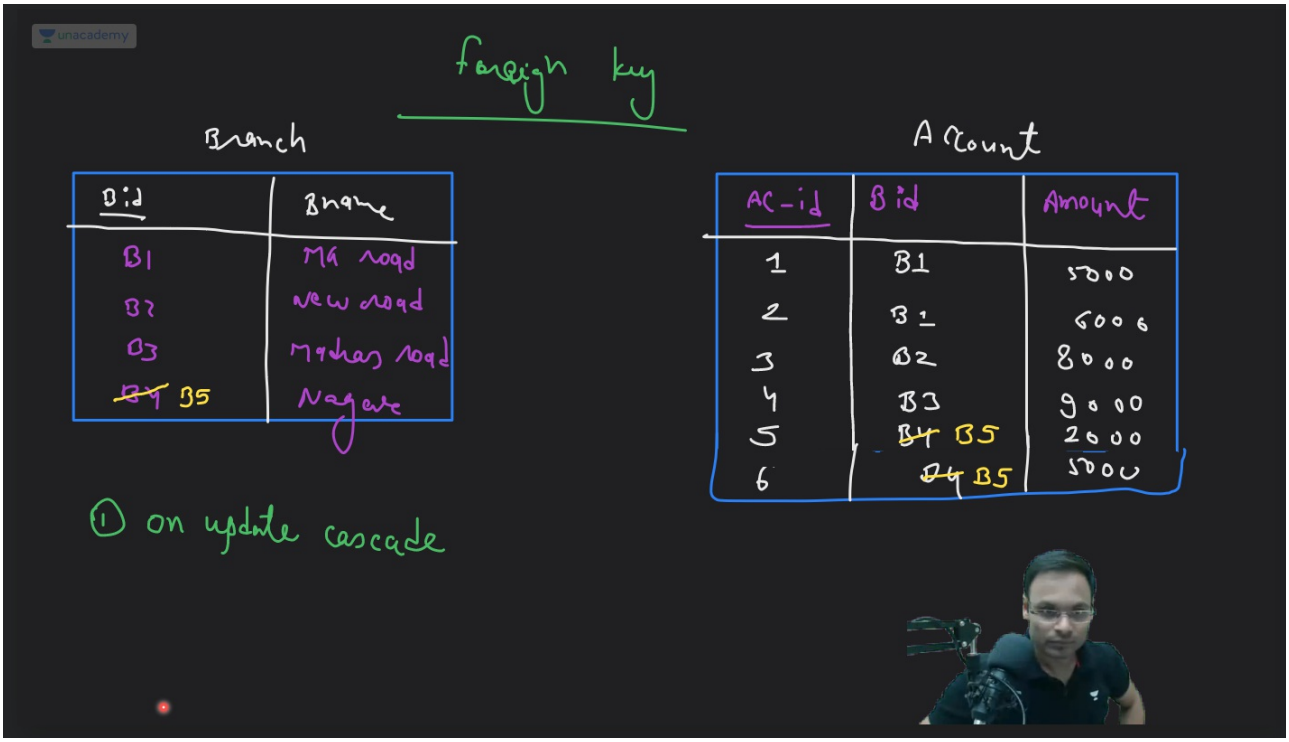
- Set of attributes.

SQL part-II (6) [26th June 2023]

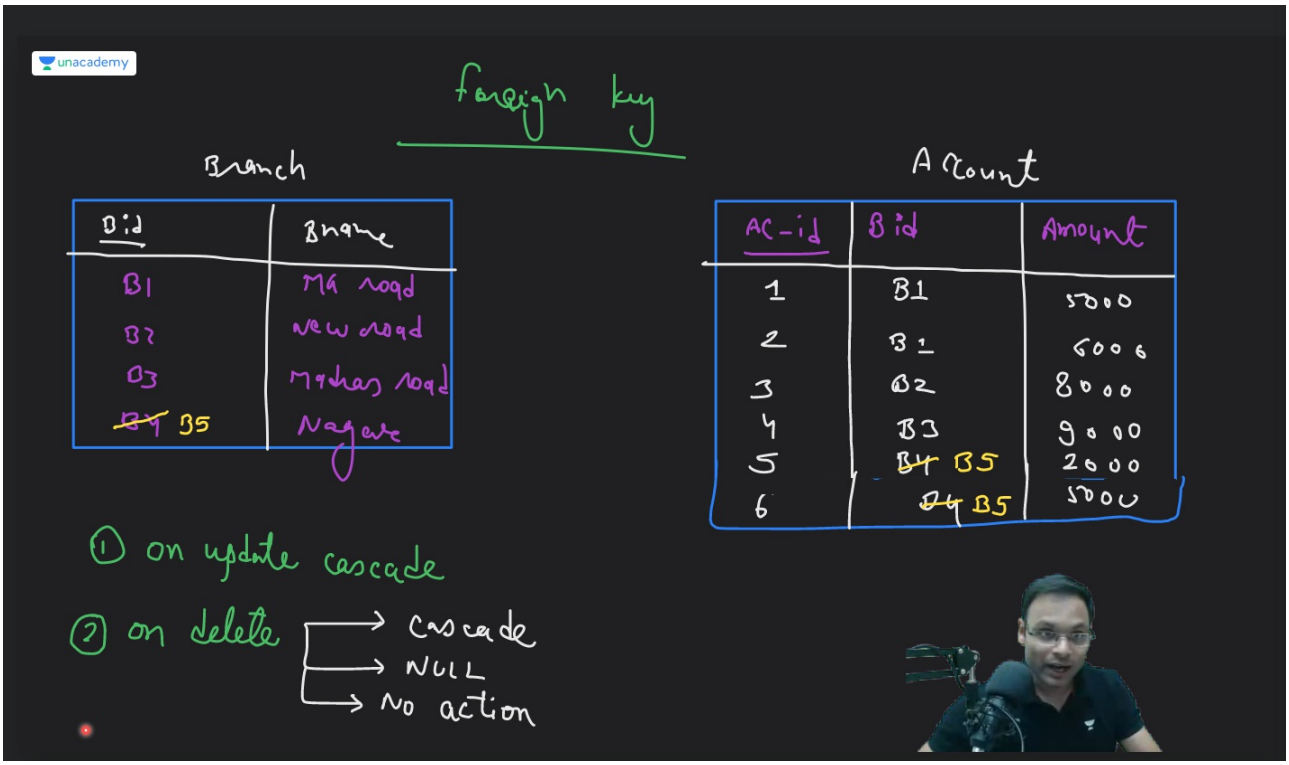
Foreign Key



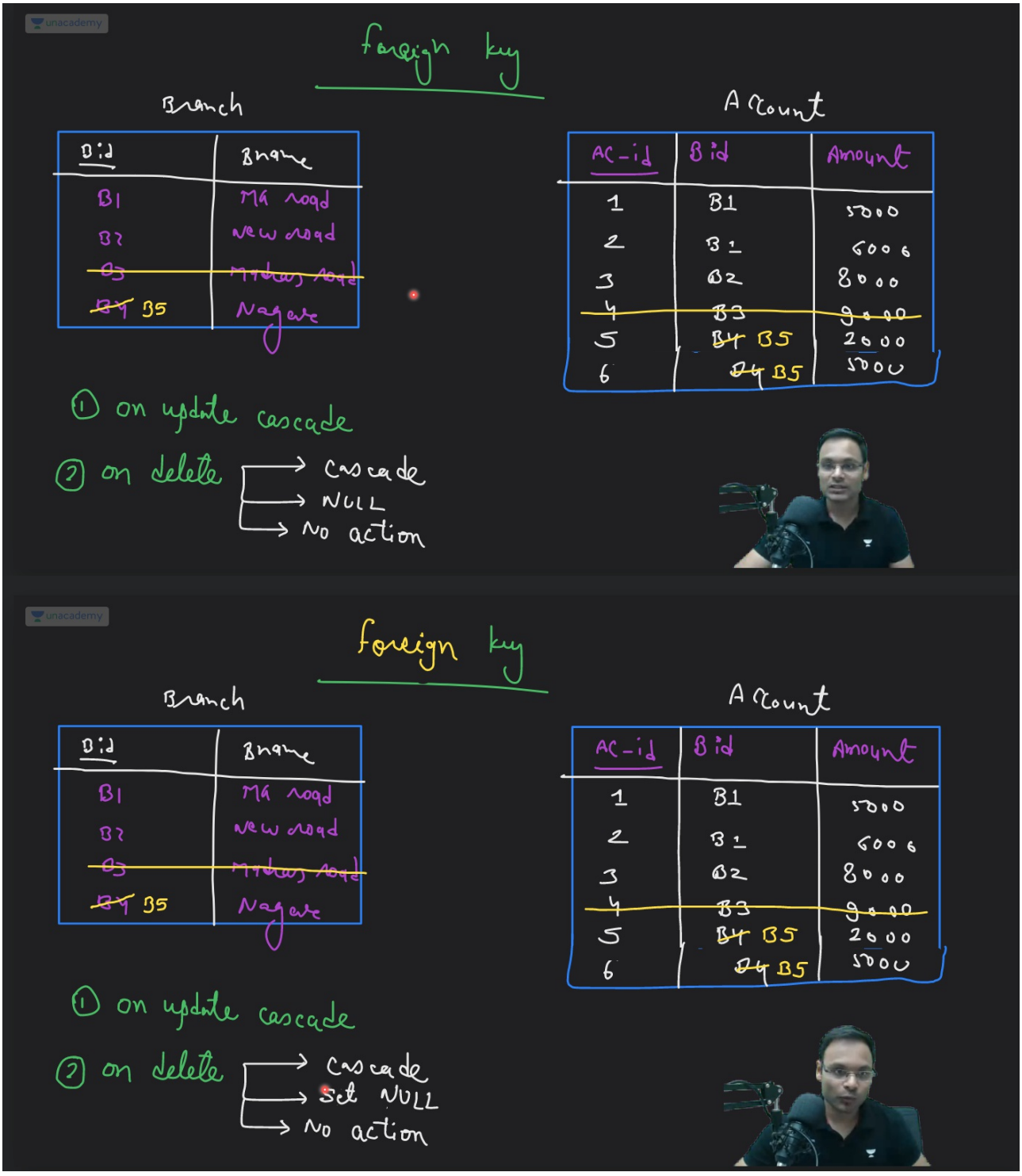
- On Update Cascade -> If we **update** the column **Branch.Bid** and the value of **B4** was updated to **B5** then the column **Account.Bid** values which had **B4** will be updated to **B5**.



On Delete

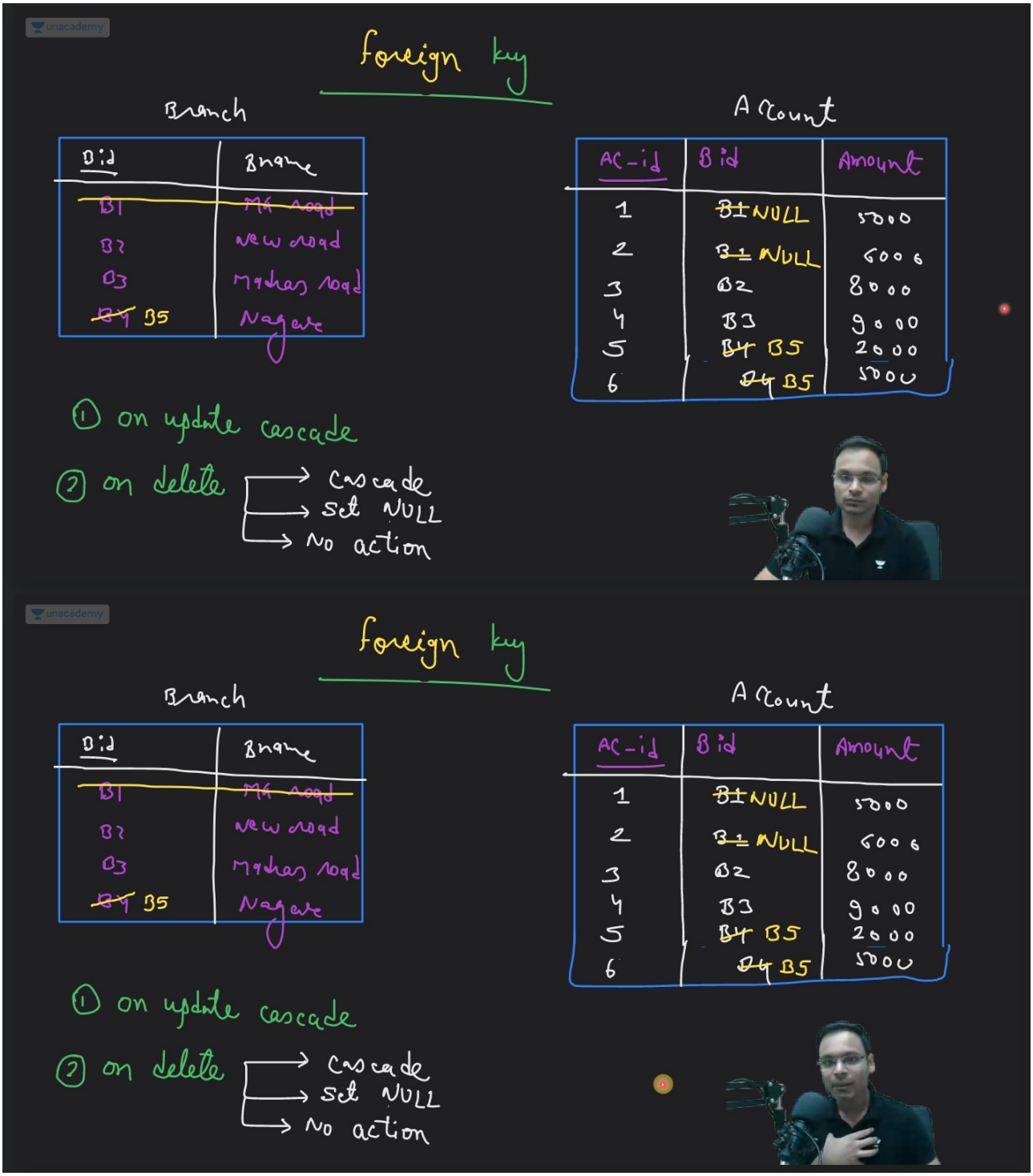


1. **Cascade** -> If we **delete** the **B3** rows in **Branch.Bid**, then the **B3** values present in **Account.Bid**, their whole/entire **row** entry will also be **deleted**.



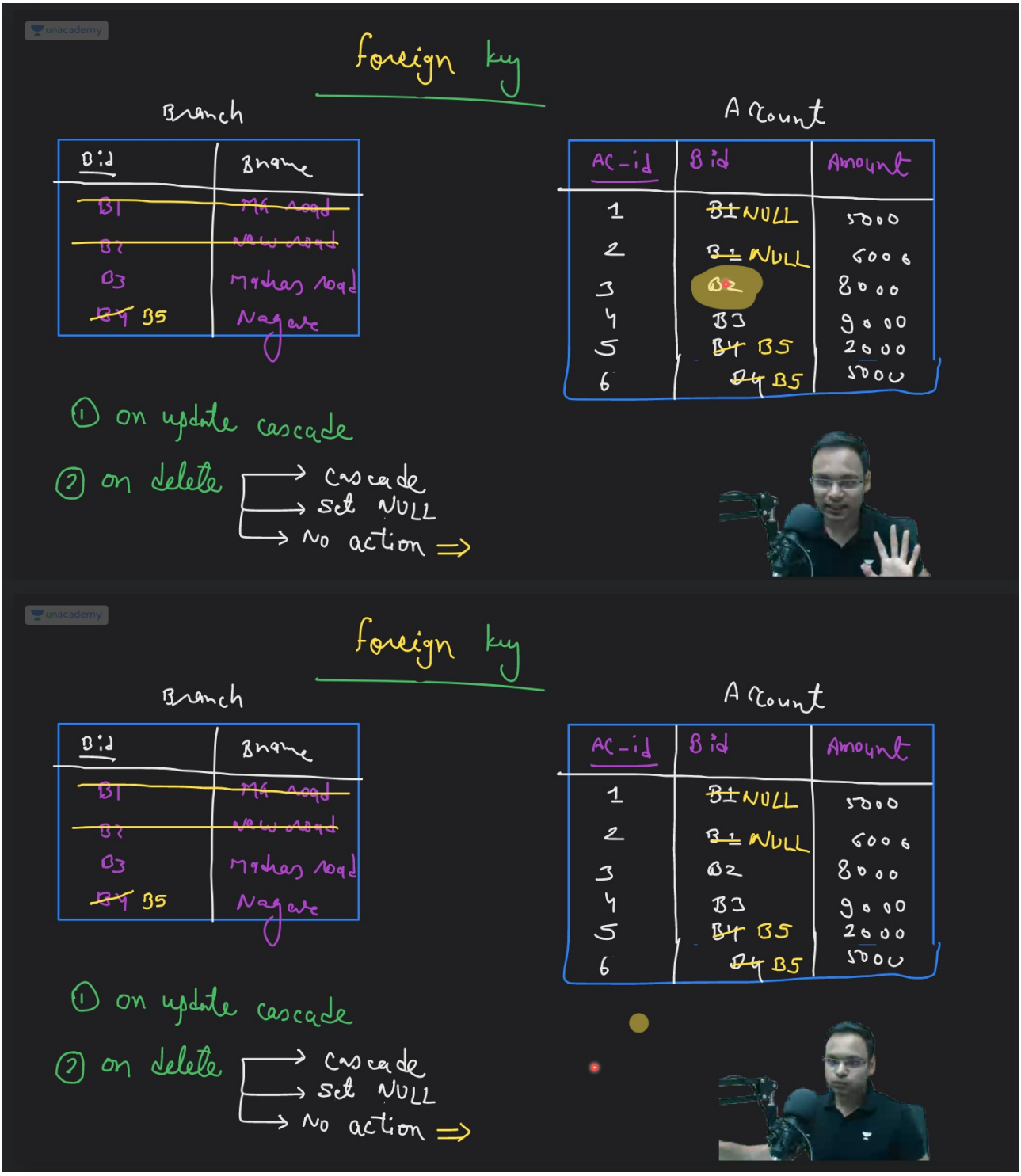
- Not only the value is deleted, the whole row is **deleted**.

2. **Set NULL** -> If we **delete** the **B1** row from the **Branch.Bid** then the values in **Accound.Bid**, their values will be set/updated to **NULL**. We will have **no value** there.



In **Account.Bid**, **NULL** values should be allowed otherwise **On Delete Set NULL** will not work.

3. **No action** -> If we **delete** the **B2** row from the **Branch.Bid** then there will be **no changes** to the values in the **Accound.Bid** column.



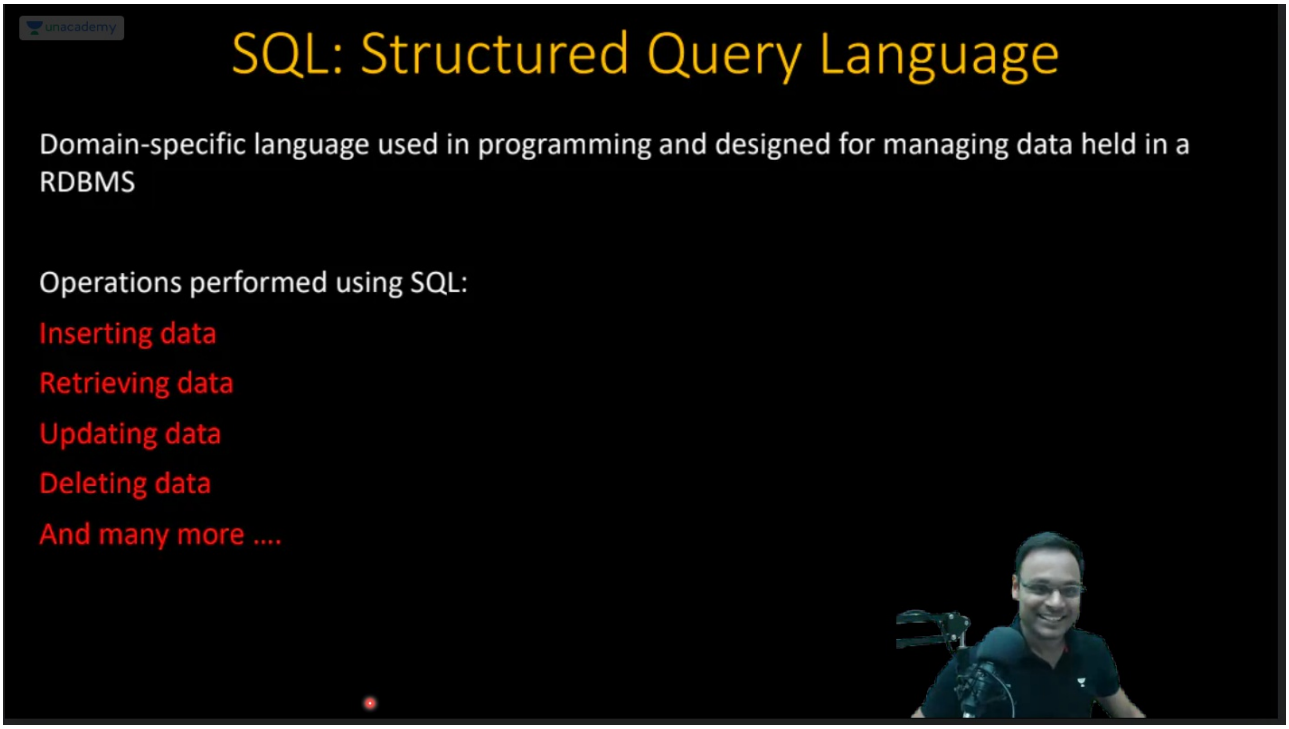
SQL

- Structured Query Language
- Domain-specific language used in programming and designed for managing data held in a **RDBMS**.
- Operations in SQL:-
- Inserting data
- Retrieving data

- Updating data
- Deleting data
- and more ....
- How we will access ER diagram data?

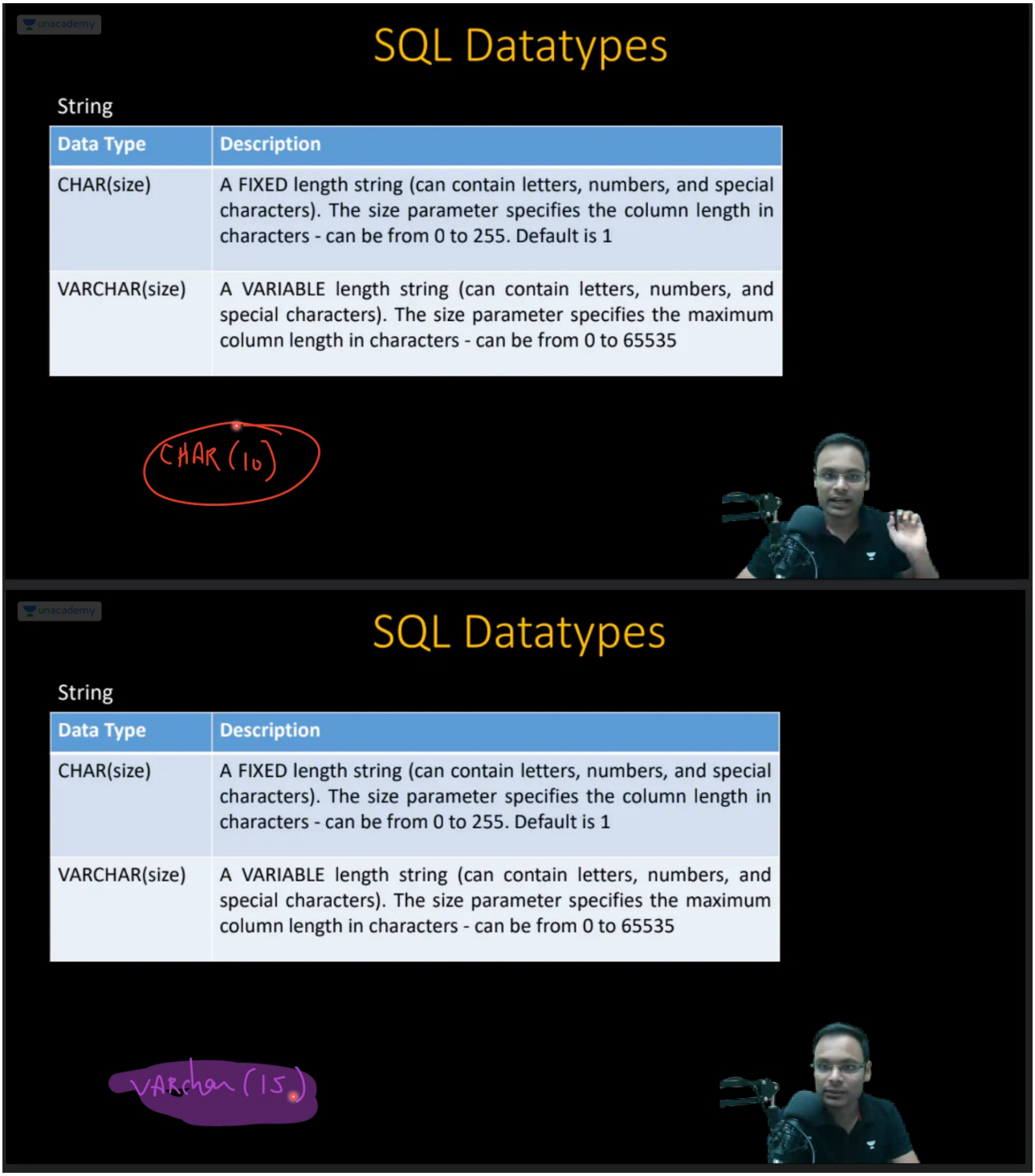
> We do not store anything in **ER diagram**, it is just a **representation**. From **ER diagram**, we create a **table**, and we actually store that **table**.

- ER diagram -> Paper thing(Representation).



SQL Datatypes

- CHAR(size) -> A **fixed** length string.
- VARCHAR(size) -> A **variable** length string.



- CHAR(10)
- VARCHAR(15)

SQL Datatypes

Numeric

Data Type	Description
Bit(size)	A bit-value type. The number of bits per value is specified in size. The size parameter can hold a value from 1 to 64. The default value for size is 1
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
INT(Size)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width (which is 255)
INTEGER(Size)	Equal to INT(size)

SQL Datatypes

Numeric

Data Type	Description
Bit(size)	A bit-value type. The number of bits per value is specified in size. The size parameter can hold a value from 1 to 64. The default value for size is 1
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
INT(Size)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width (which is 255)
INTEGER(Size)	Equal to INT(size)

SQL Datatypes

Numeric

Data Type	Description
BIGINT(size)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The size parameter specifies the maximum display width (which is 255)
FLOAT(size, d)	A floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in

	the d parameter. This syntax is deprecated in MySQL 8.0.17
FLOAT(p)	A floating point number. MySQL uses the p value to determine whether to use FLOAT or DOUBLE for the resulting data type. If p is from 0 to 24, the data type becomes FLOAT(). If p is from 25 to 53, the data type becomes DOUBLE()

unacademy

SQL Datatypes

Numeric

Data Type	Description
BIGINT(size)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The size parameter specifies the maximum display width (which is 255)
FLOAT(size, d)	A floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. This syntax is deprecated in MySQL 8.0.17
FLOAT(p)	A floating point number. MySQL uses the p value to determine whether to use FLOAT or DOUBLE for the resulting data type. If p is from 0 to 24, the data type becomes FLOAT(). If p is from 25 to 53, the data type becomes DOUBLE()

Float (10,4)

unacademy

SQL Datatypes

Numeric

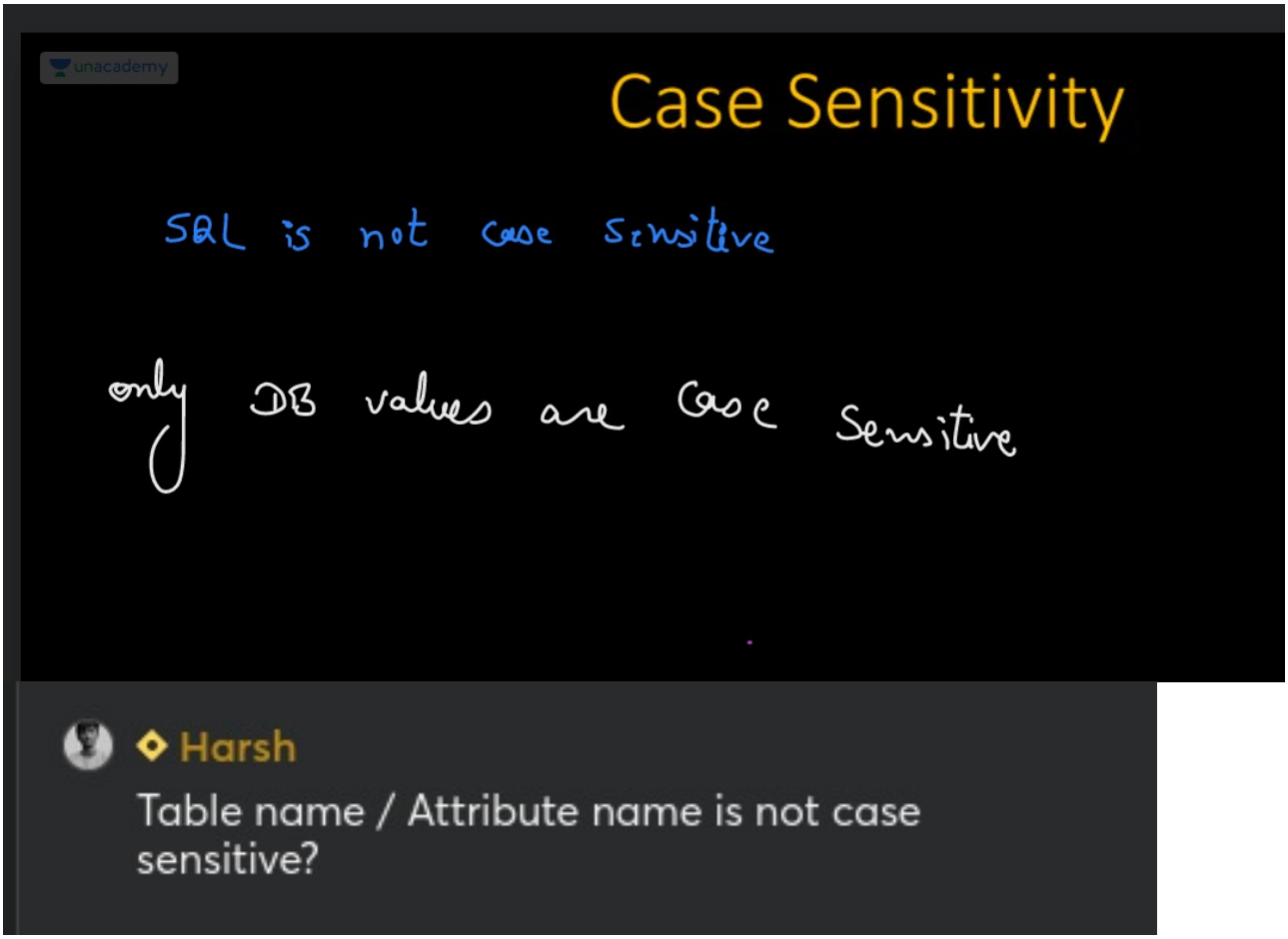
Data Type	Description
DOUBLE(size, d)	A normal-size floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter
DECIMAL(size, d)	An exact fixed-point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. The maximum number for size is 65. The maximum number for d is 30. The default value for size is 10. The default value for d is 0.
DEC(size, d)	Equal to DECIMAL(size,d)

- Integer/Int
- Float

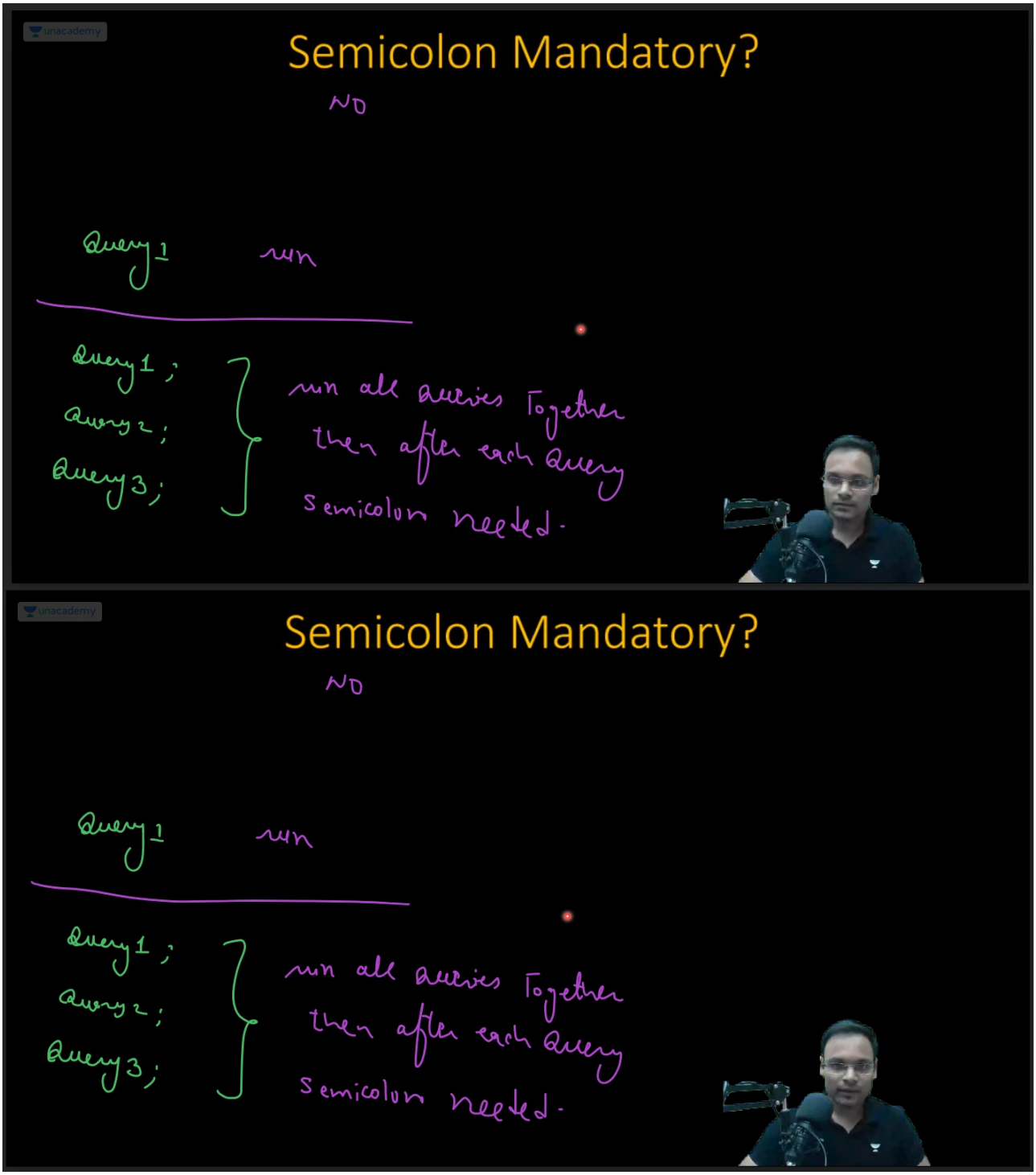
- Varchar
- These **three** are mostly needed.

Case Sensitivity

- SQL is not **case sensitive**.
- Only **DB** values are **case sensitive**.



- Yes.



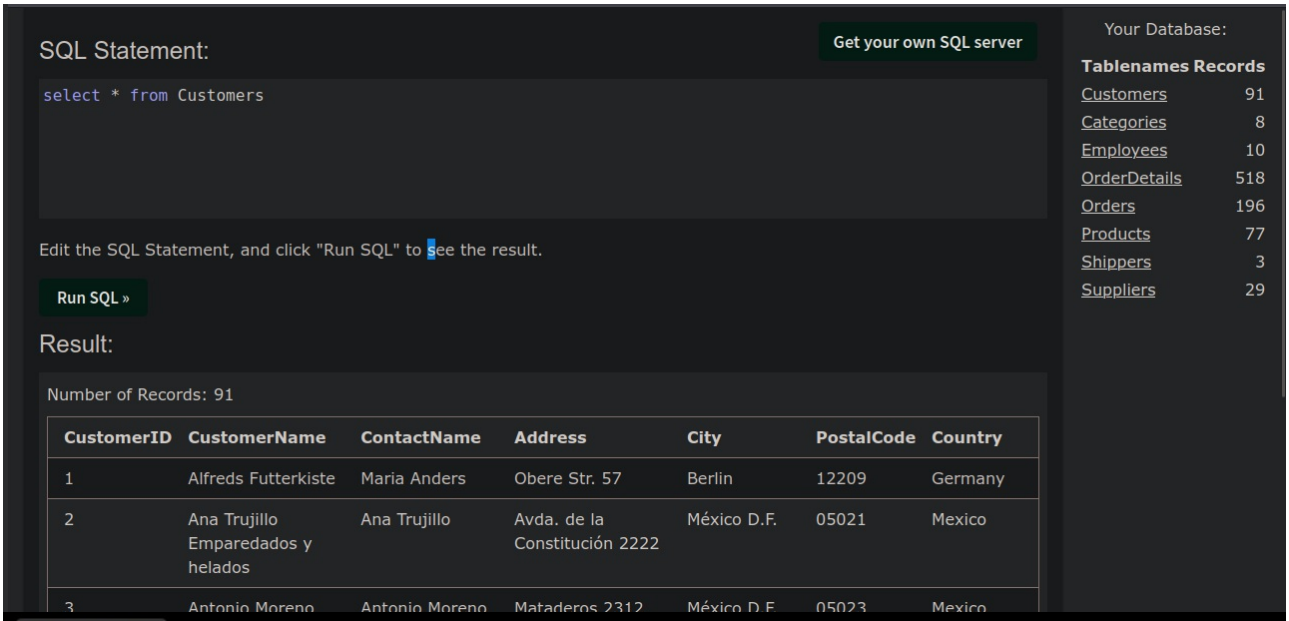
- Semicolon Mandatory? -> **NO**.
- Run all queries together then after each query semicolon needed.

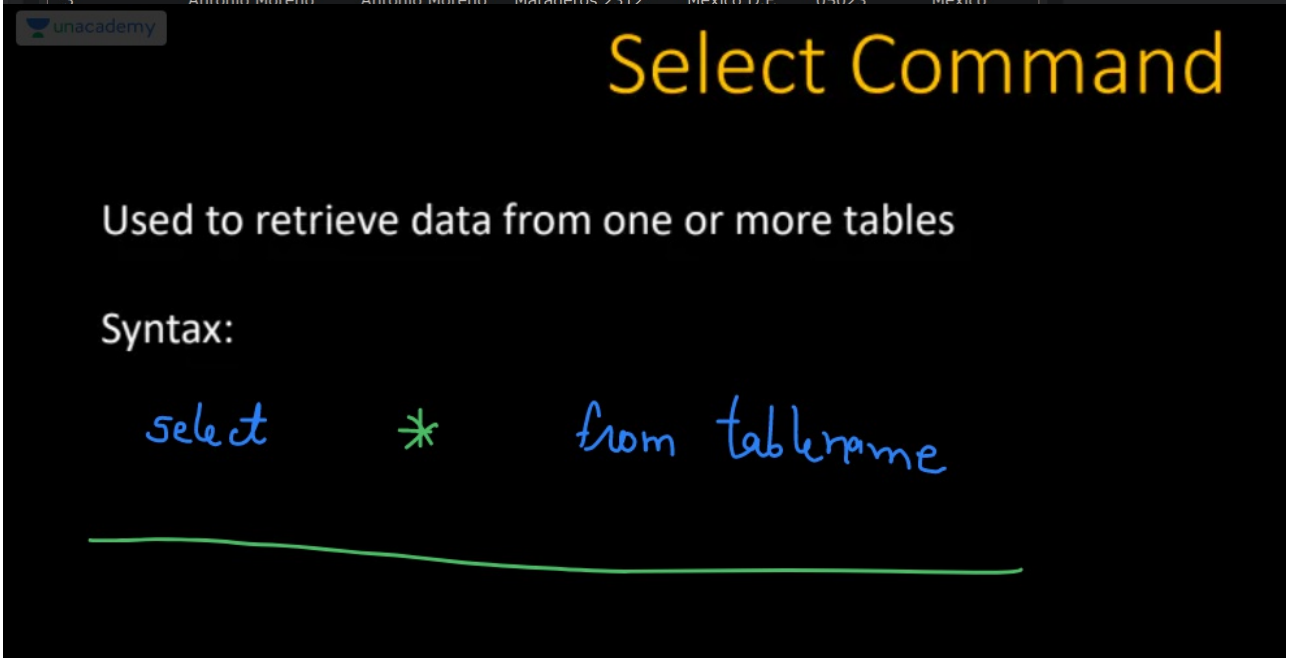
Retrieving Data

Select Command

- Used to retrieve data from one or more tables.
- **Syntax** -> Select * from tablename;
- '*' means we are selecting everything(data) from the **table**.

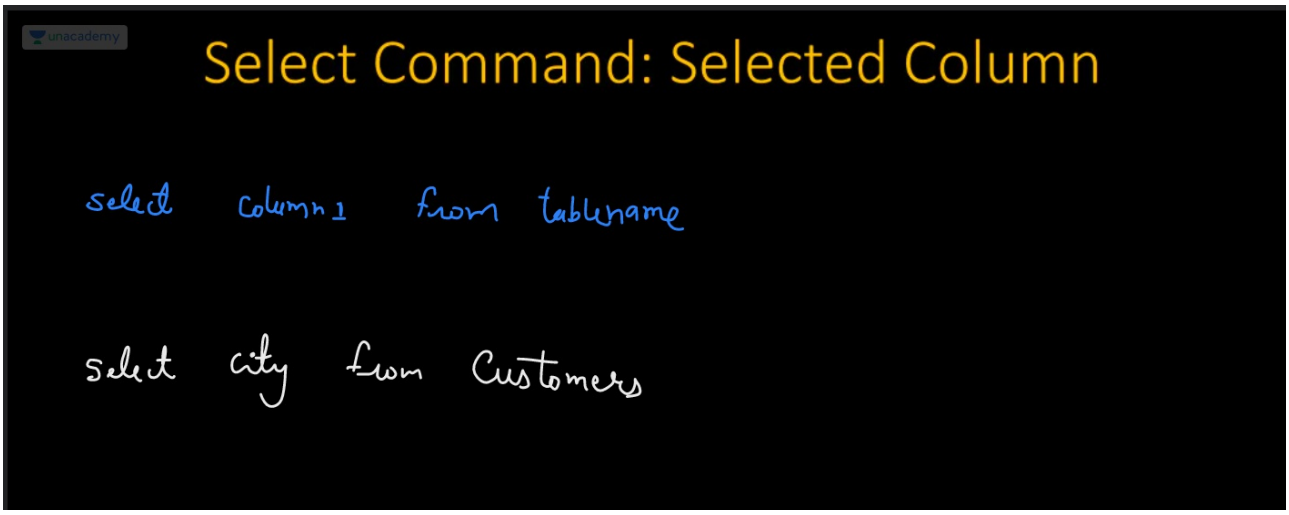
- select * from Customers;



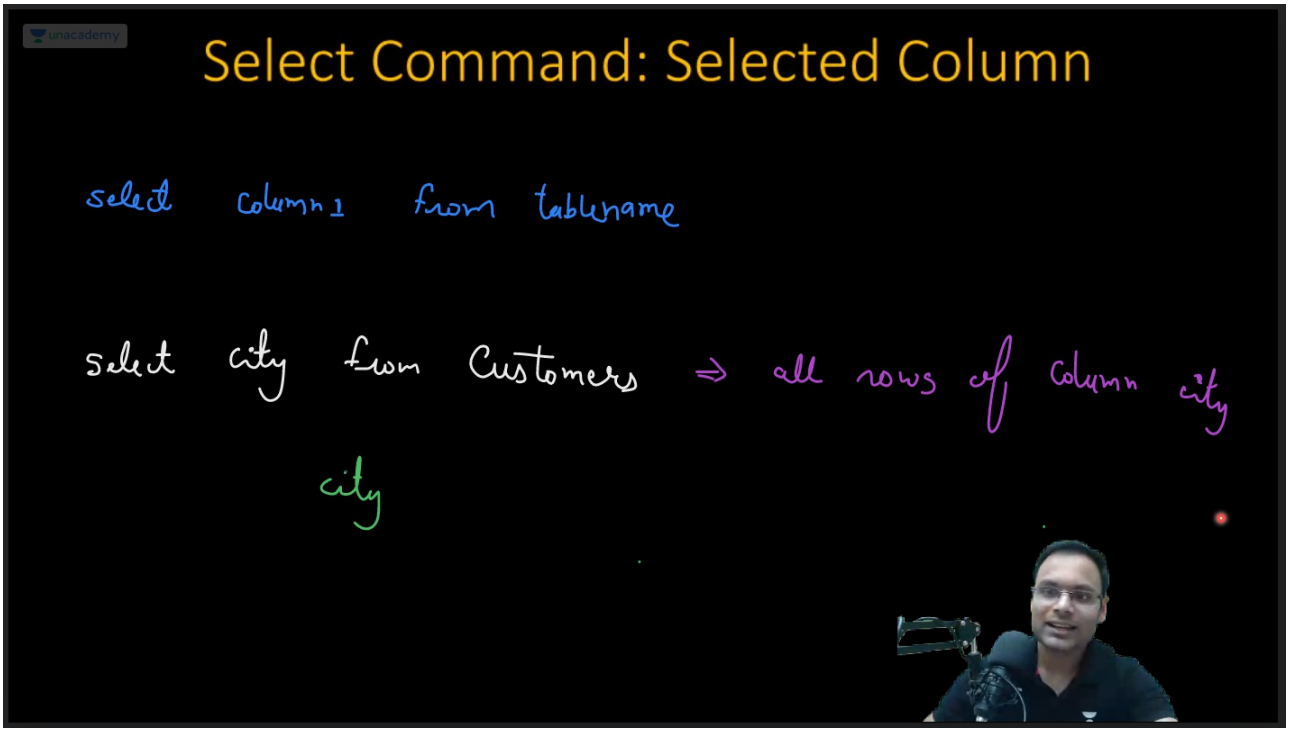


Selected Column

- **Select** command is used for **column** filtering as well.



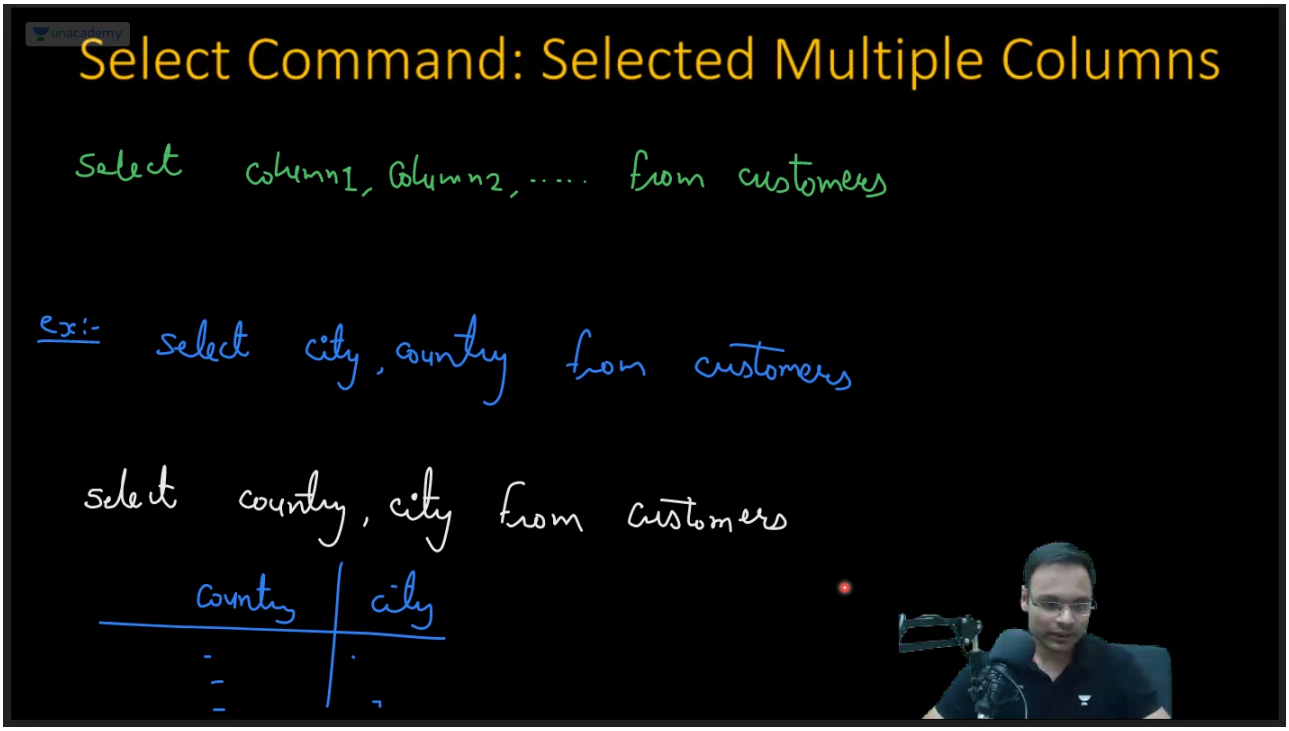
- select City from Customers -> All rows of column city.



Selecting multiple columns

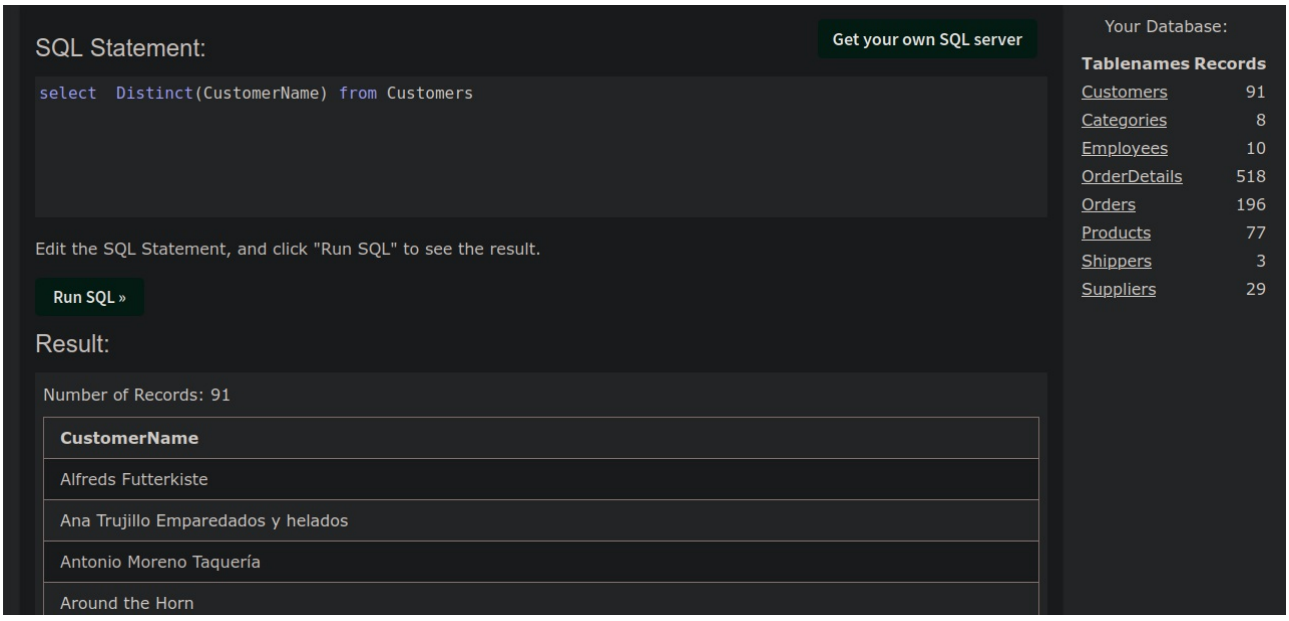
- select City, Country from Customers

- select Country, City, CustomerName from Customers



Distinct keyword

- Need to be used with select, to fetch only unique values of designated columns(s)



- select Distinct(CustomerName) from Customers

SQL Statement:

Get your own SQL server

```
select  Distinct(Country) from Customers
```

Edit the SQL Statement, and click "Run SQL" to see the result.

Run SQL »

Result:

Number of Records: 21

Country
Argentina
Austria
Belgium
Brazil

Your Database:

Tablenames	Records
Customers	91
Categories	8
Employees	10
OrderDetails	518
Orders	196
Products	77
Shippers	3
Suppliers	29

SQL Statement:

Get your own SQL server

```
select  Country from Customers
```

Edit the SQL Statement, and click "Run SQL" to see the result.

Run SQL »

Result:

Number of Records: 91

Country
Germany
Mexico
Mexico
UK

Your Database:

Tablenames	Records
Customers	91
Categories	8
Employees	10
OrderDetails	518
Orders	196
Products	77
Shippers	3
Suppliers	29

SQL Statement:

Get your own SQL server

```
select  distinct Country from Customers
```

Edit the SQL Statement, and click "Run SQL" to see the result.

Run SQL »

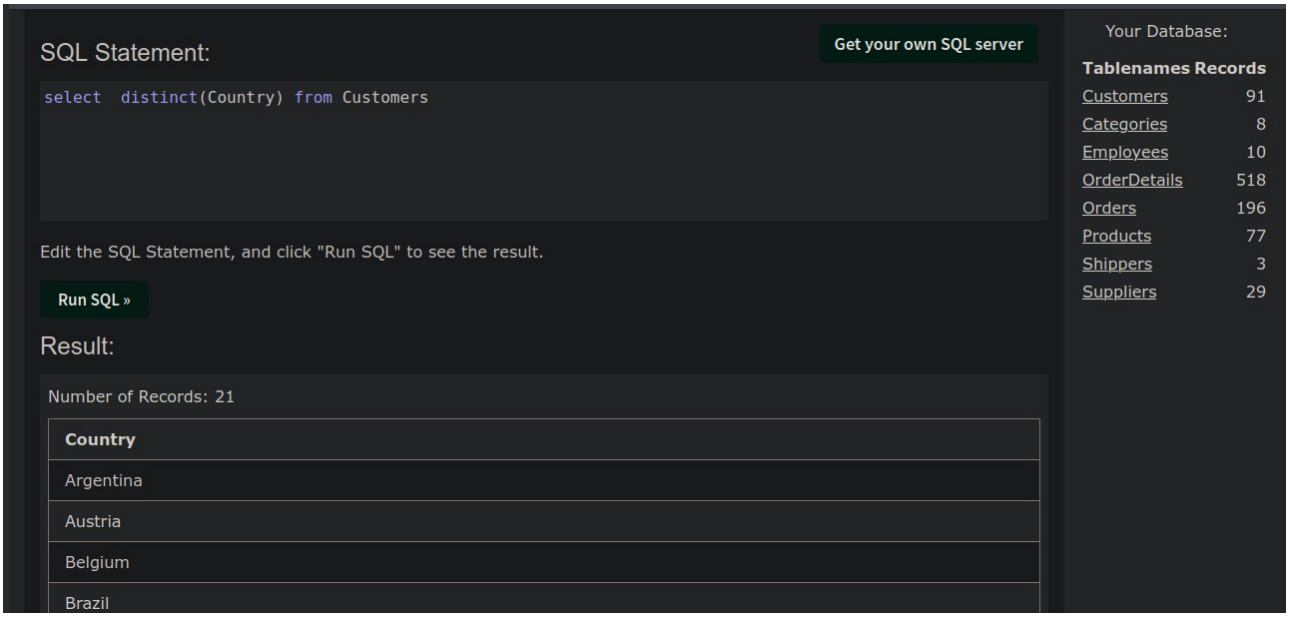
Result:

Number of Records: 21

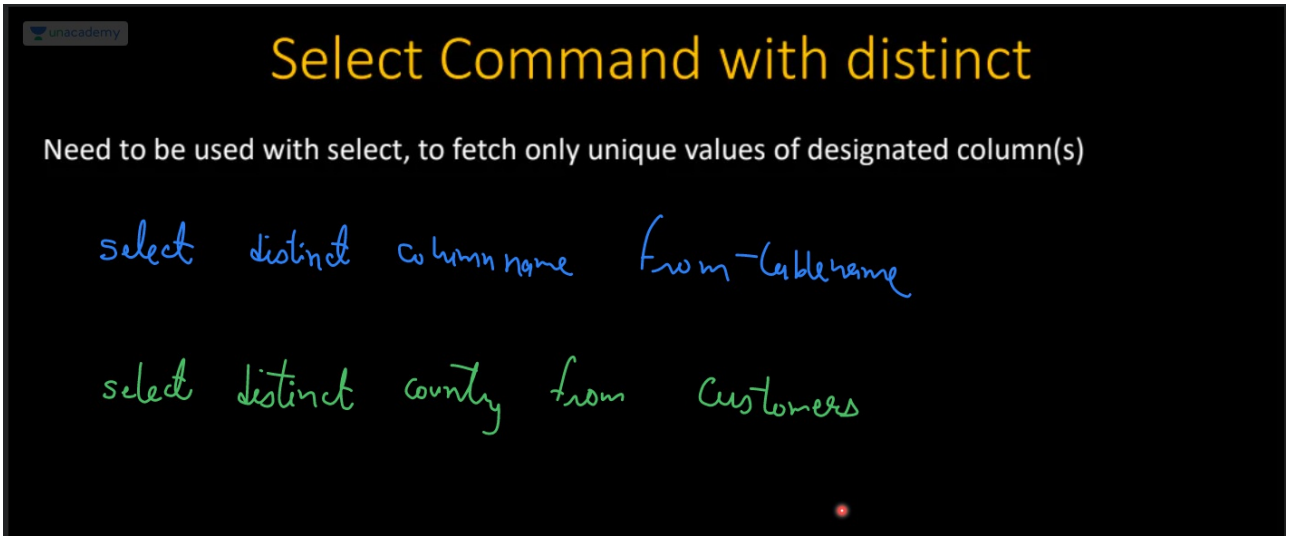
Country
Argentina
Austria
Belgium
Brazil

Your Database:

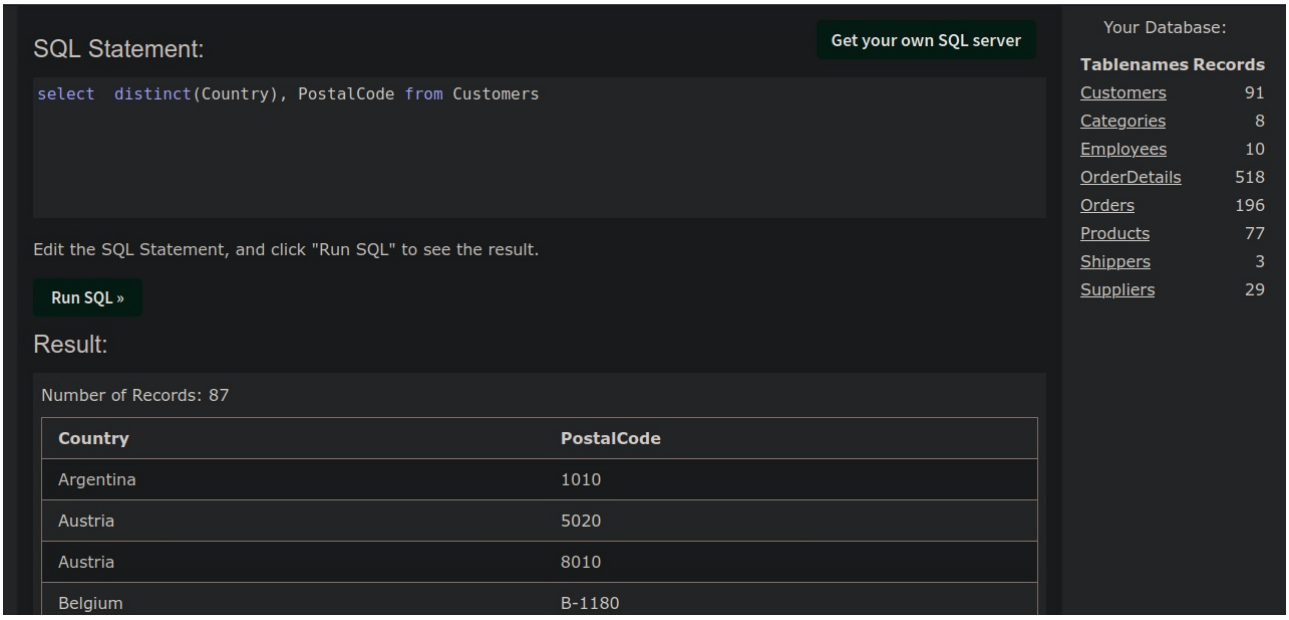
Tablenames	Records
Customers	91
Categories	8
Employees	10
OrderDetails	518
Orders	196
Products	77
Shippers	3
Suppliers	29



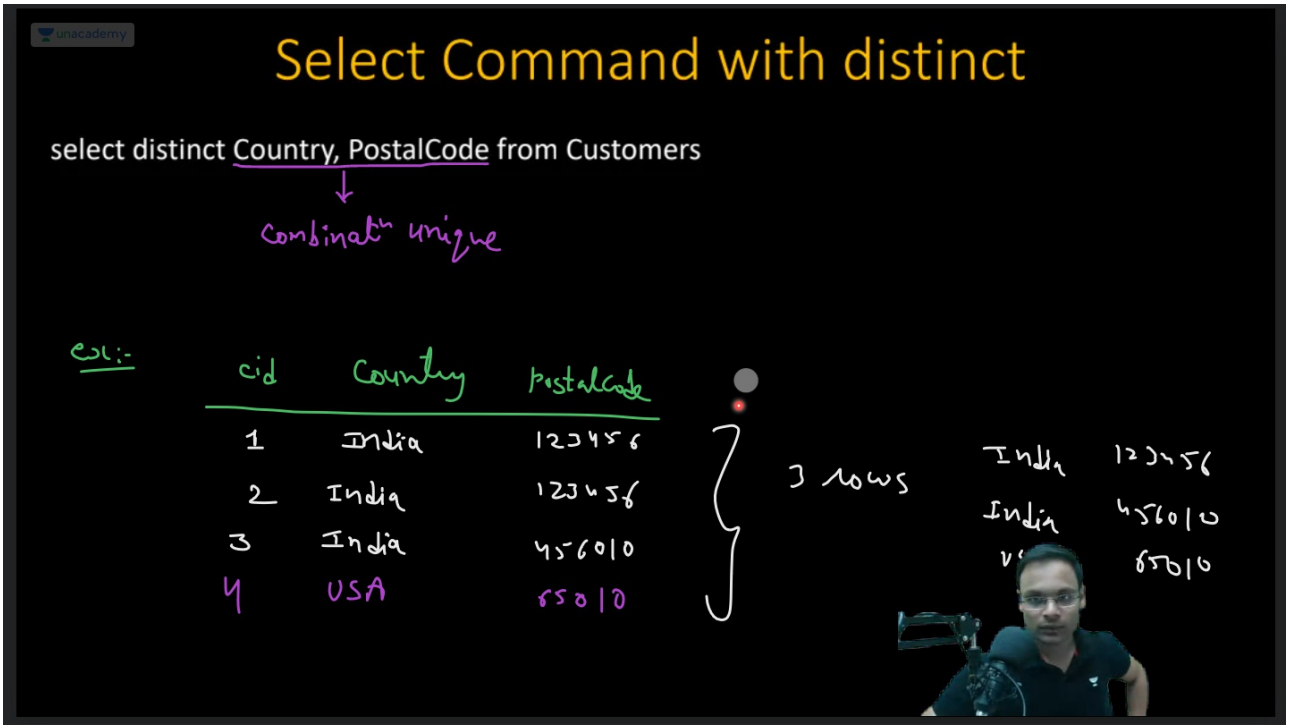
- select distinct columnname from tablename
- slect distinct country from customers

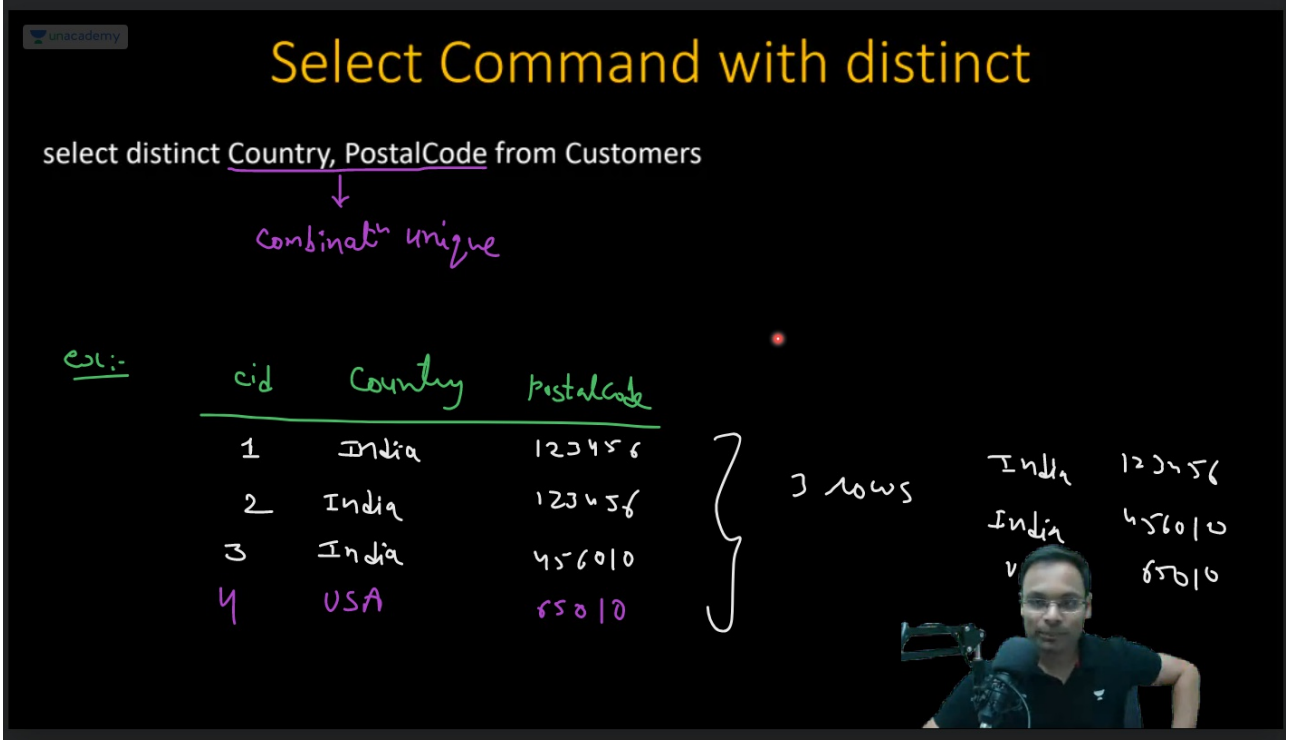


Command with distinct

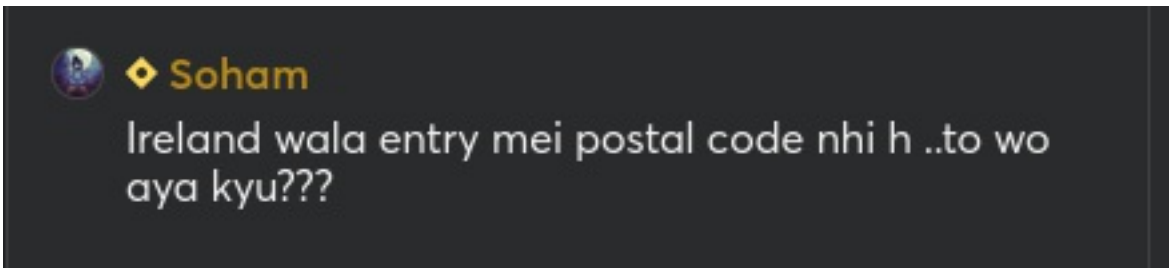


- select distinct(Country), PostalCode from Customers

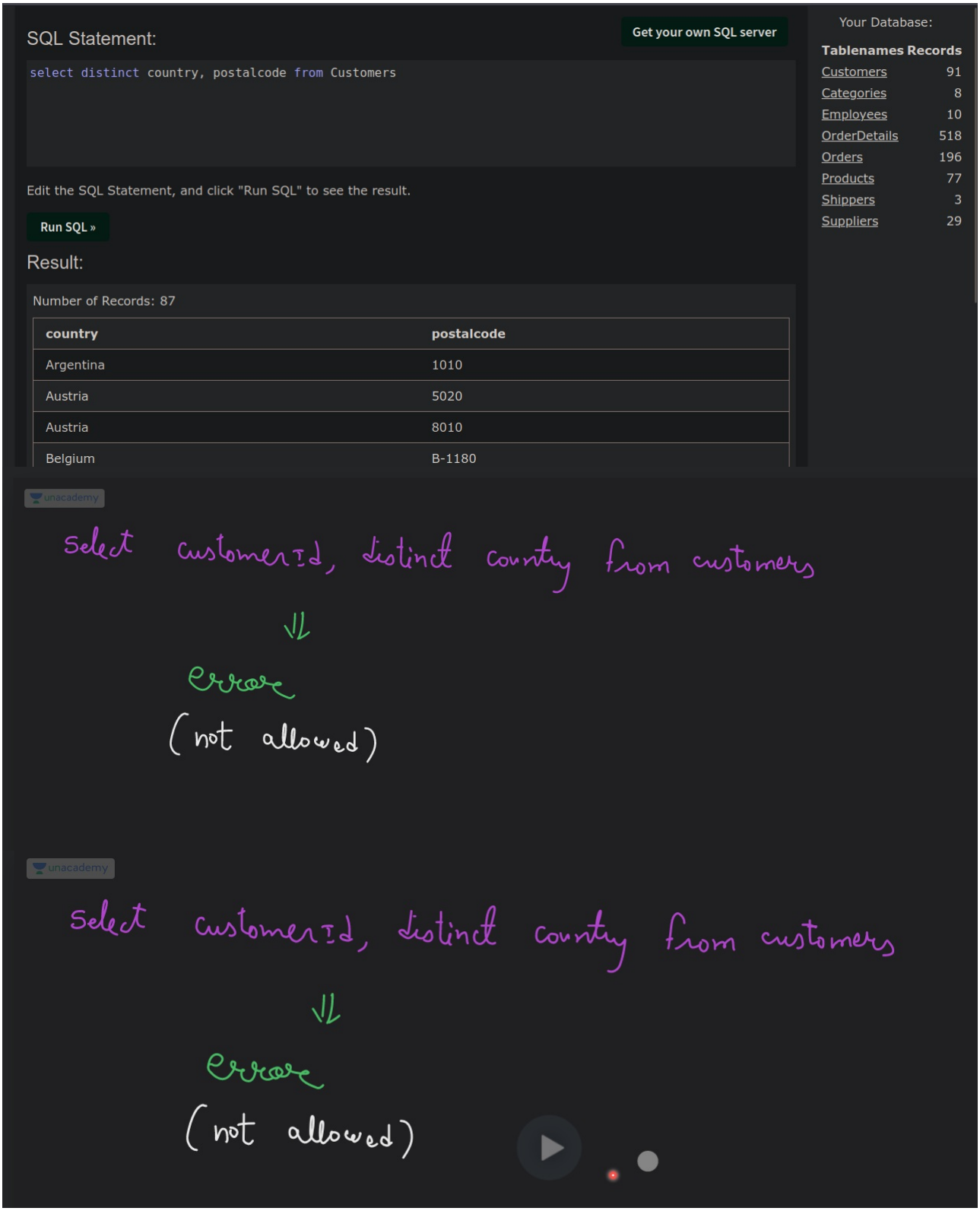




- Combination of **Country and Postalcode** are **unique** that are **selected**.
- distinct combinations

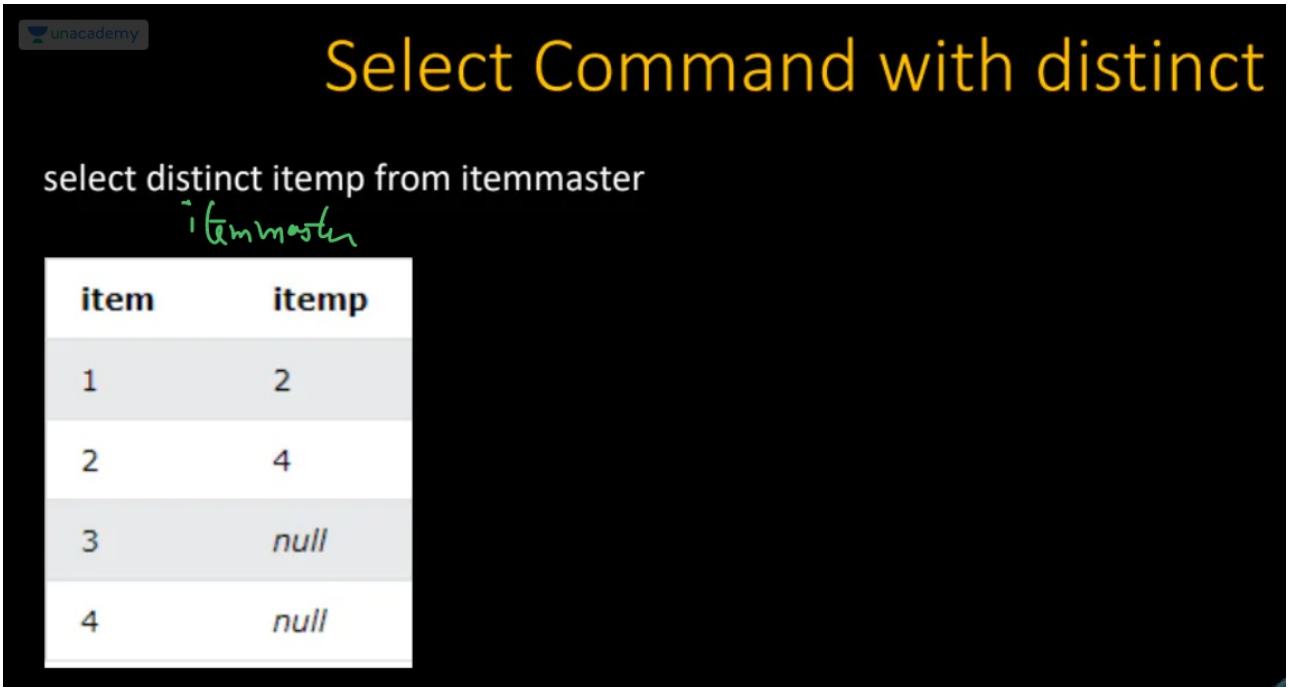


- If **combination is unique** then it will come.

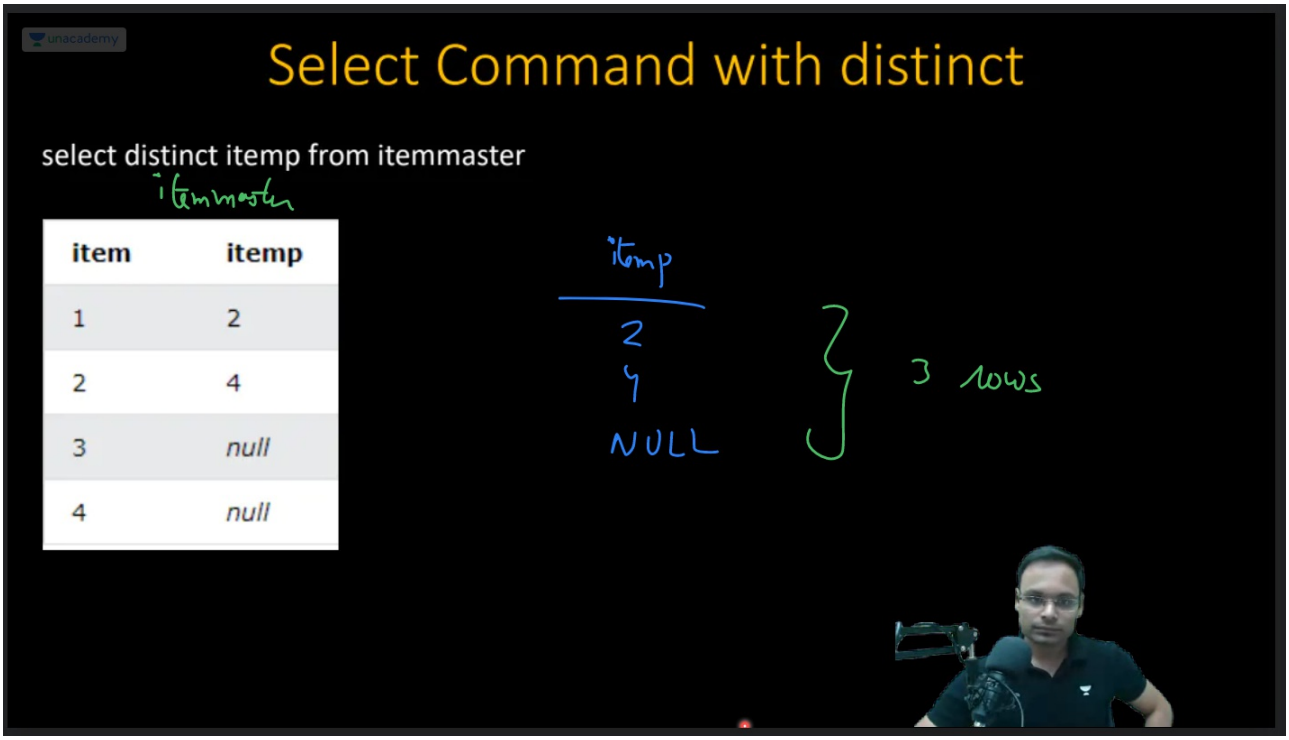


- Error, not allowed.
- We cannot write **distinct** in one column and not in the other.
- If we write more than **one column** then **distinct** should be applied to all **columns**, or **none of the columns** should have **distinct**.
- We have to put **distinct** on all of the columns we want to retrieve or none of the columns should have **distinct** keyword.

Command with distinct



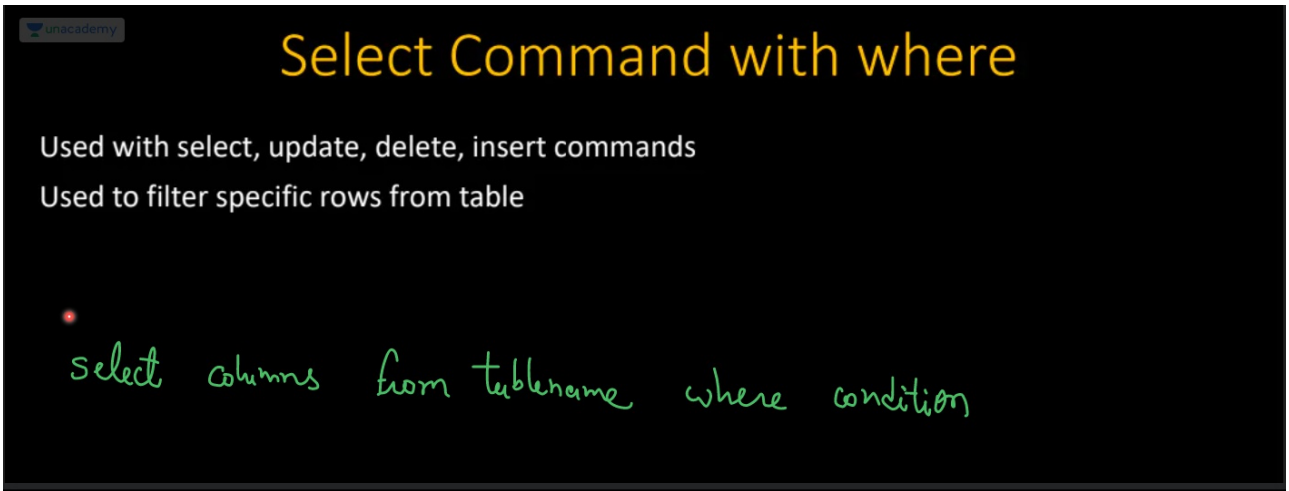
- select distinct itemp from itemmaster;
- **3** distinct values -> 3 rows -> 2, 4 and NULL.

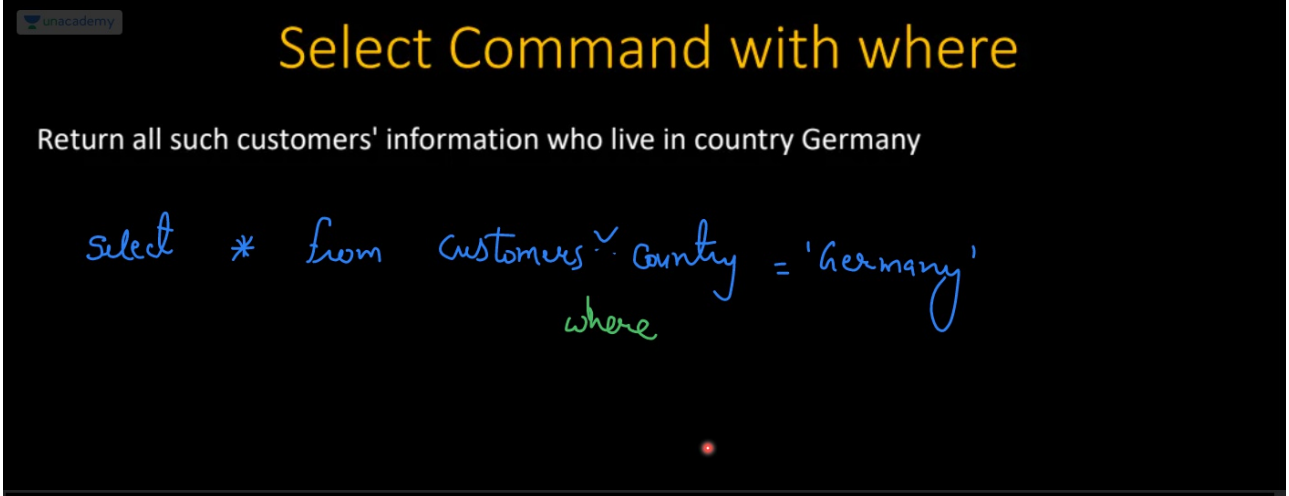


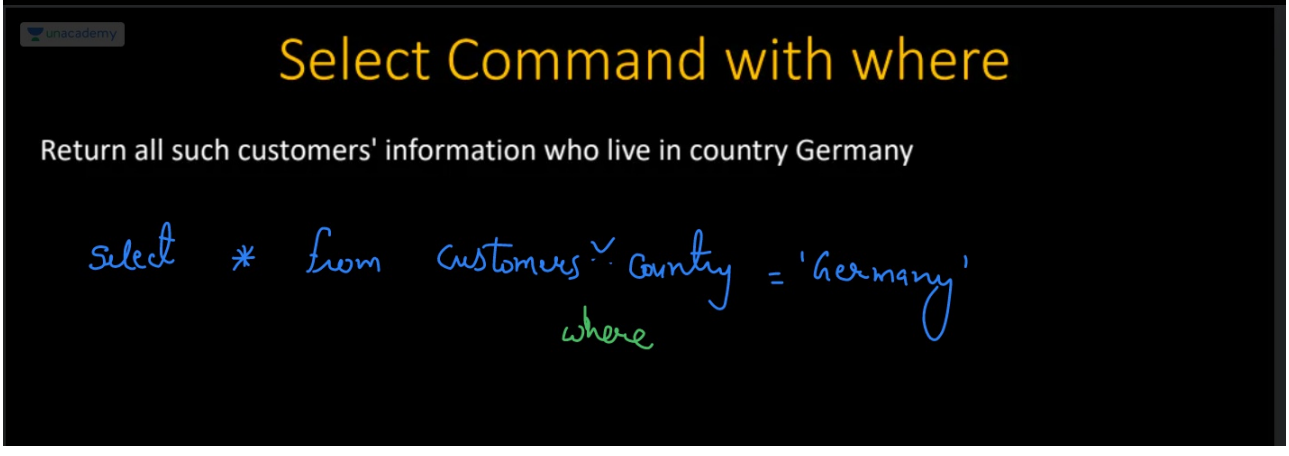
- select distinct Rno from itemmaster;
- **4** distinct values -> 4 rows

Where command

- Used with select, update, delete, insert commands
- Used to filter specific rows from table.
- Syntax -> select columns from tablename where condition;







SQL Statement:

Get your own SQL server

```
select * from customers where country = 'Germany';
```

Edit the SQL Statement, and click "Run SQL" to see the result.

Run SQL »

Result:

Number of Records: 11

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
6	Blauer See Delikatessen	Hanna Moos	Forsterstr. 57	Mannheim	68306	Germany
17	Drachenblut Delikatessend	Sven Ottlieb	Walserweg 21	Aachen	52066	Germany

Your Database:

Tablenames	Records
Customers	91
Categories	8
Employees	10
OrderDetails	518
Orders	196
Products	77
Shippers	3
Suppliers	29

- select * from customers where country = 'Germany';
- **Germany** is a database value, it is **case sensitive** unlike **SQL**.

- This comes under DML.

SQL Statement:

Get your own SQL server

```
select * from customers where country = 'USA';
```

Edit the SQL Statement, and click "Run SQL" to see the result.

Run SQL »

Result:

Number of Records: 13

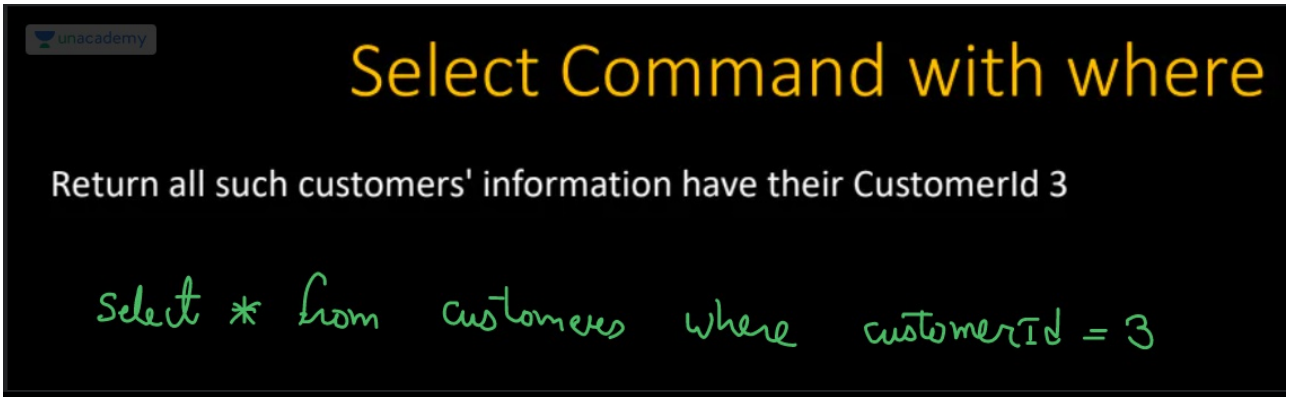
CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
32	Great Lakes Food Market	Howard Snyder	2732 Baker Blvd.	Eugene	97403	USA
36	Hungry Coyote Import Store	Yoshi Latimer	City Center Plaza 516 Main St.	Elgin	97827	USA
43	Lazy K Kountry Store	John Steel	12 Orchestra	Walla Walla	99362	USA

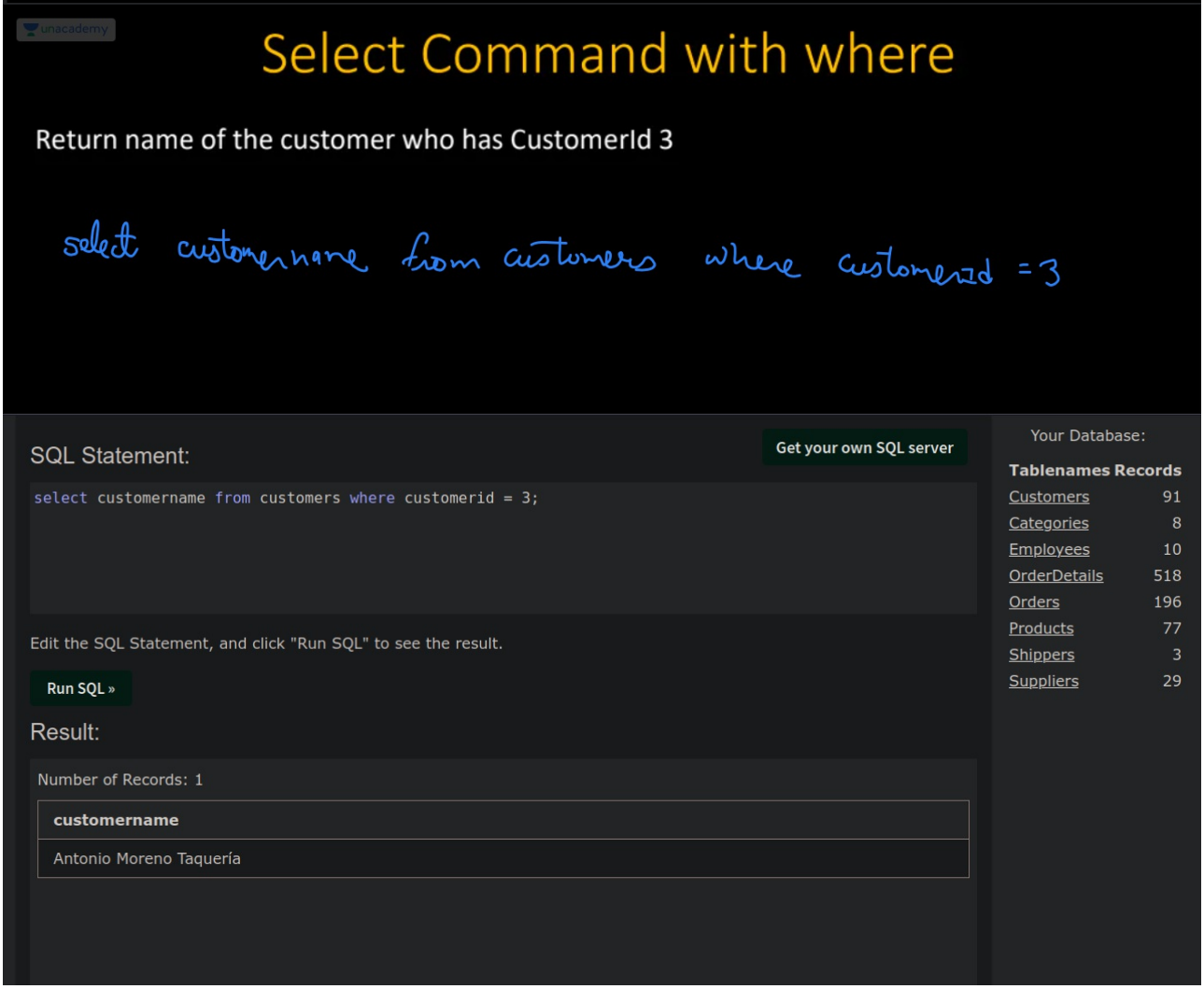
Your Database:

Tablenames	Records
Customers	91
Categories	8
Employees	10
OrderDetails	518
Orders	196
Products	77
Shippers	3
Suppliers	29

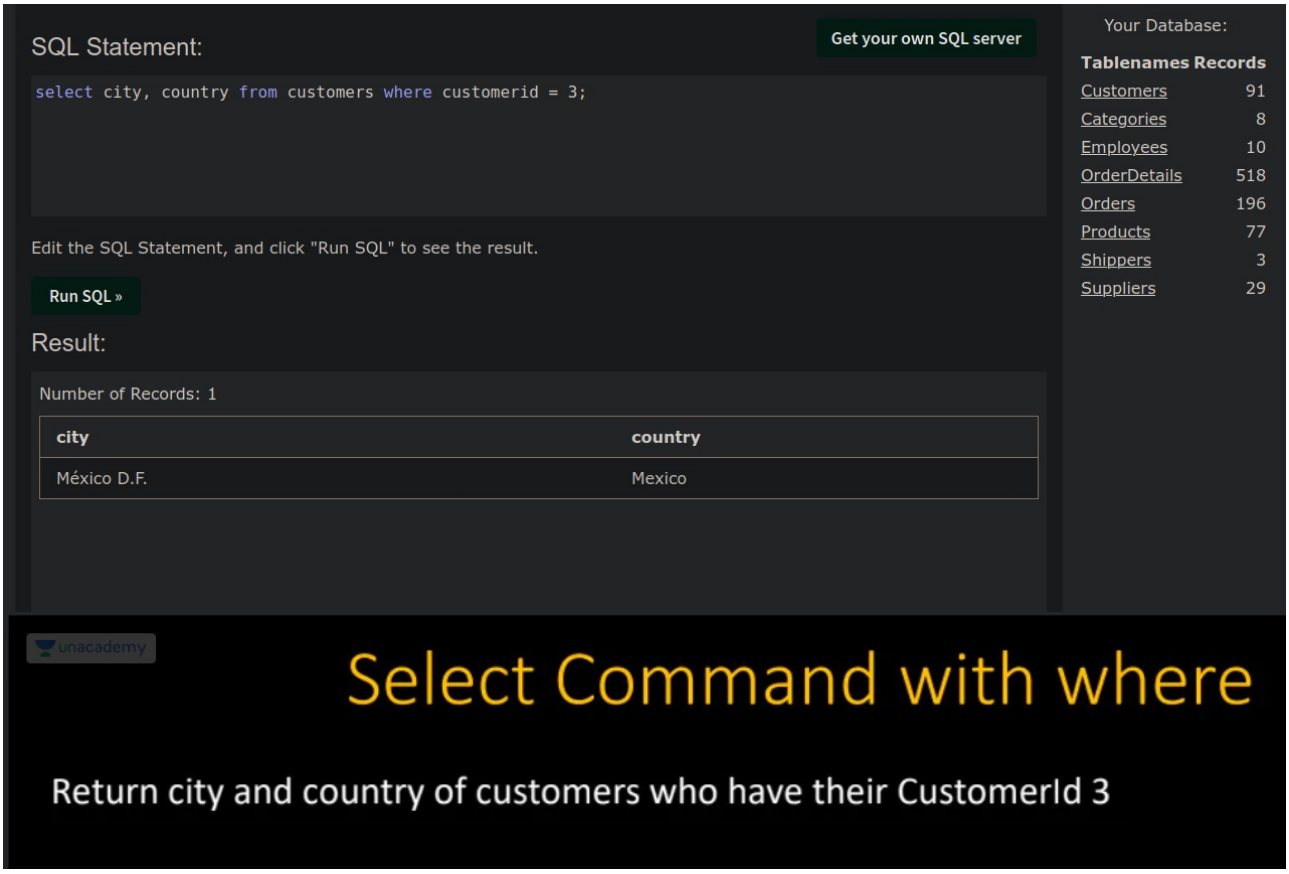
- select * from customers where country = 'USA';

- select * from customers where customerid = 3;
- As **3** is **integer** so no need of **quotes**.





- select customername from customers where customerid = 3;



- select city, country from customers where customerid = 3;

operators with where

unacademy

Select Command with where

Relational Operators used in where clause:

1. Equal
2. Not Equal
3. Less than
4. Less than or equal to
5. Greater than
6. Greater than or equal to

unacademy

Select Command with where

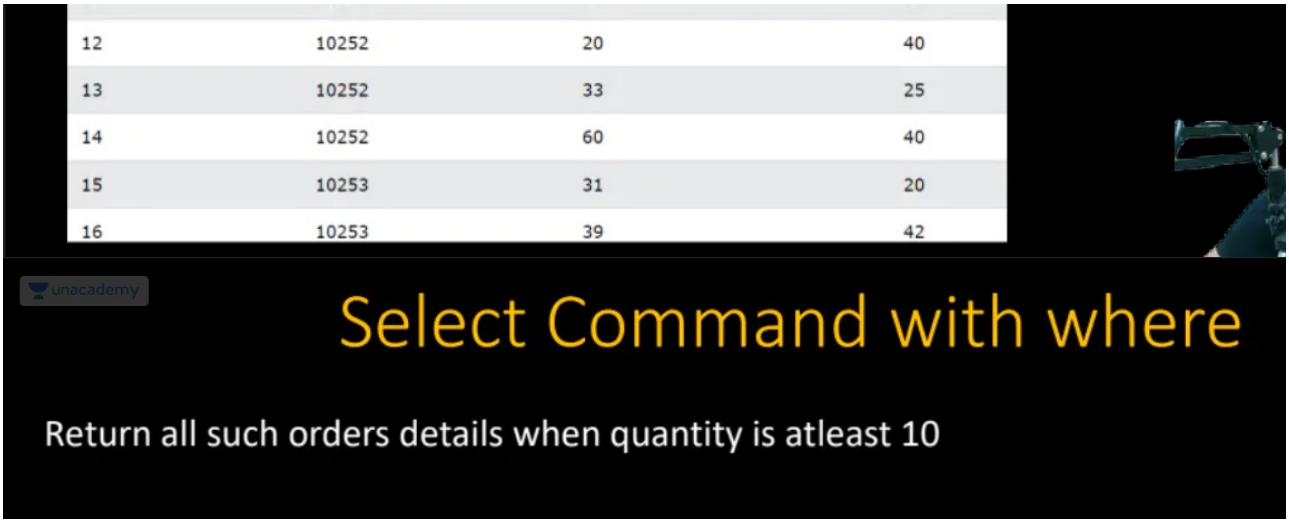
Relational Operators used in where clause:

1. Equal =
2. Not Equal != or <>
3. Less than <
4. Less than or equal to <=
5. Greater than >
6. Greater than or equal to >=

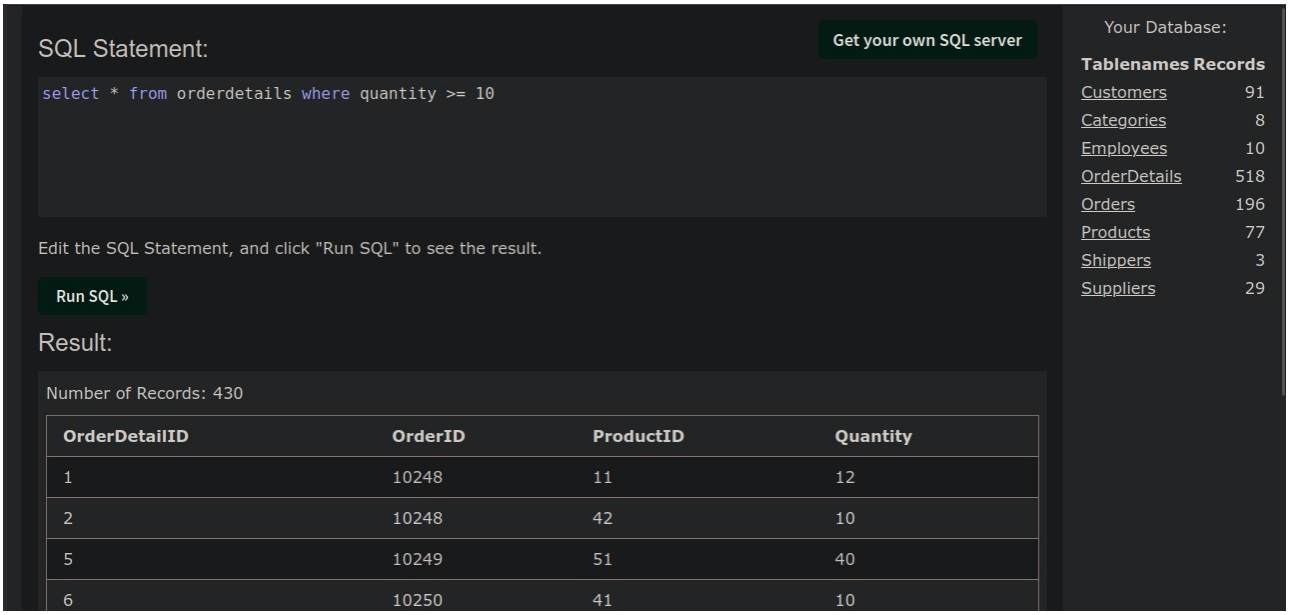
unacademy

OrderDetails Table

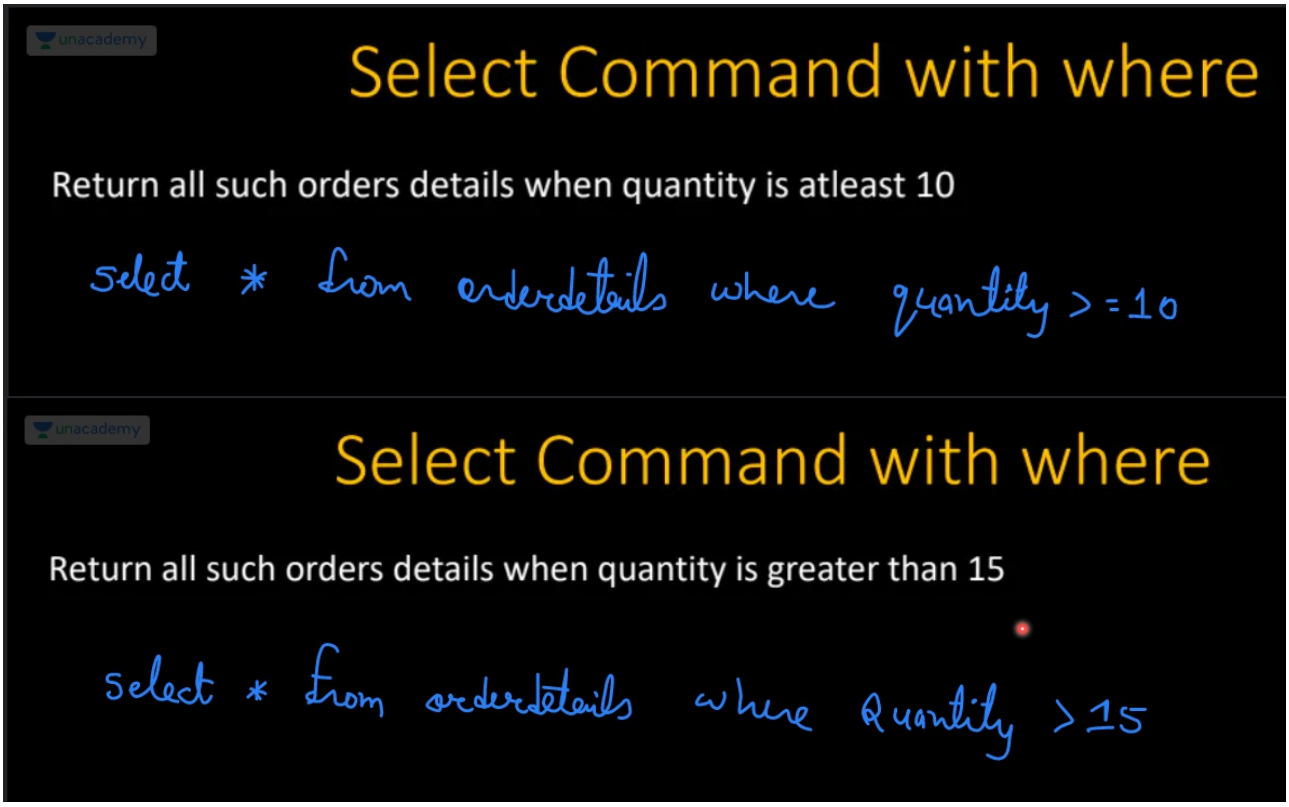
OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	12
2	10248	42	10
3	10248	72	5
4	10249	14	9
5	10249	51	40
6	10250	41	10
7	10250	51	35
8	10250	65	15
9	10251	22	6
10	10251	57	15
11	10251	65	20



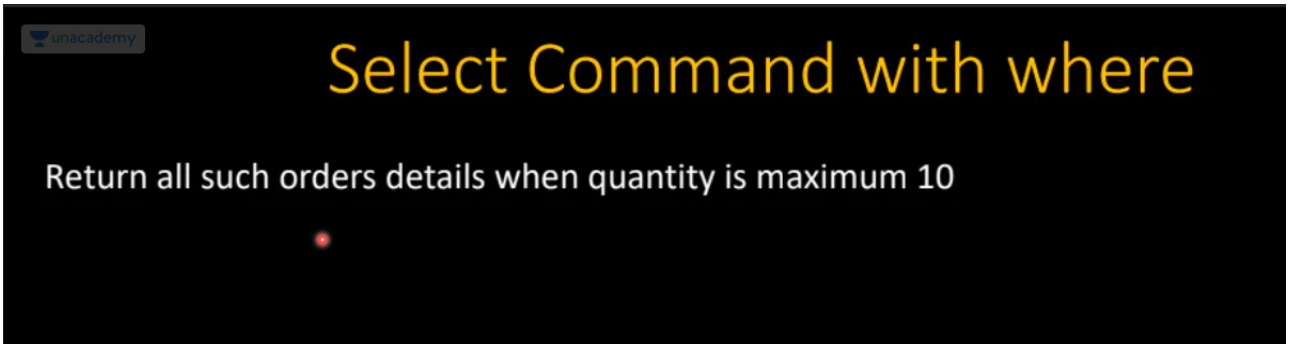
- Atleast 10 -> Min of 10 -> >= 10 -> Greater than equal to 10.



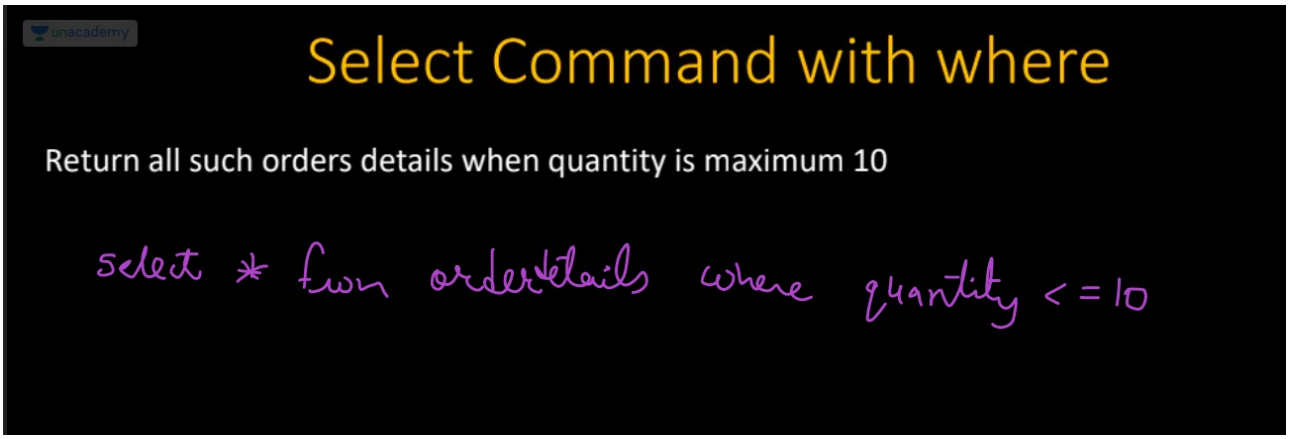
- select * from orderdetails where quantity >= 10



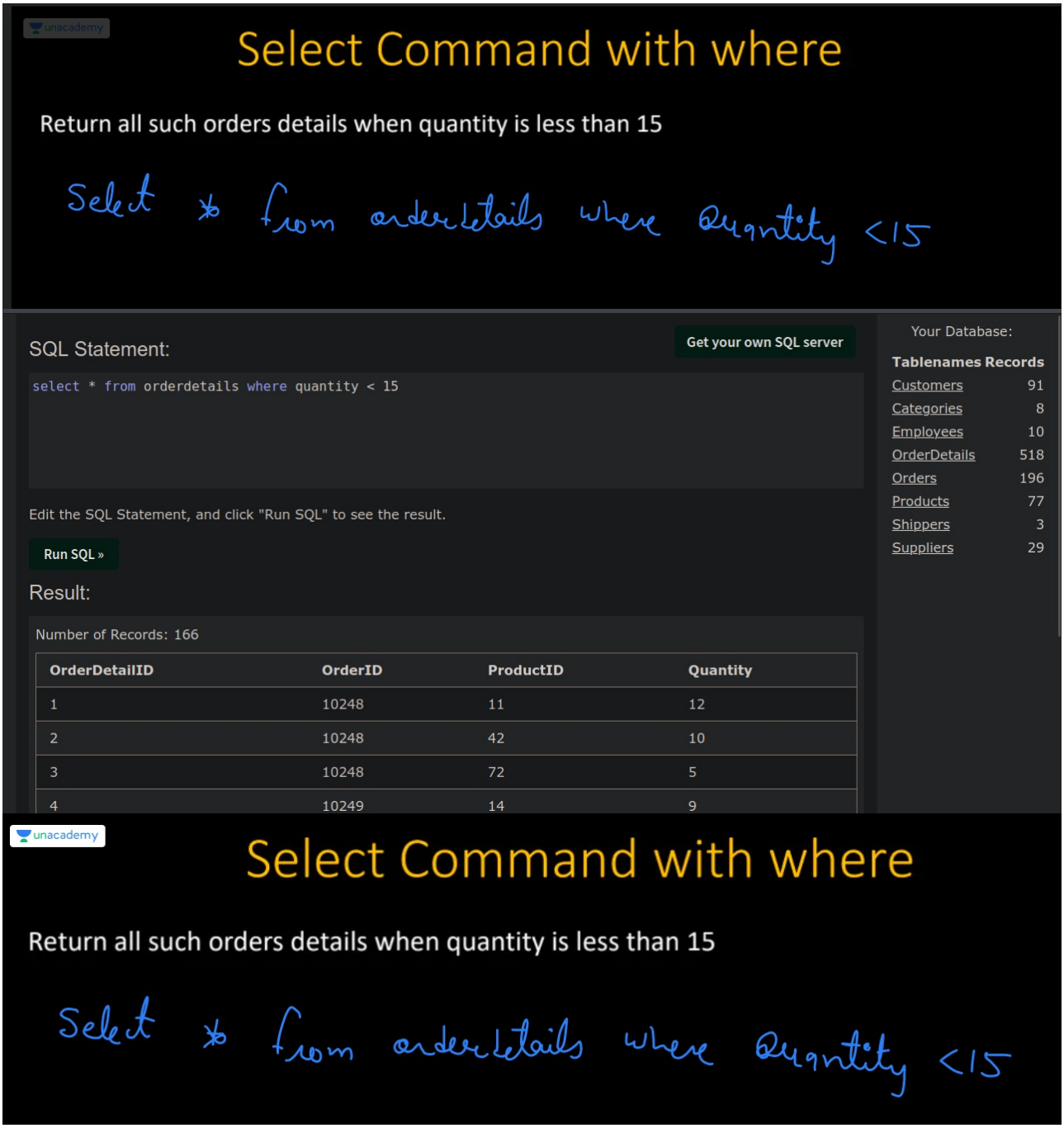
- Greater than 15



- select * from orderdetails where quantity <= 10

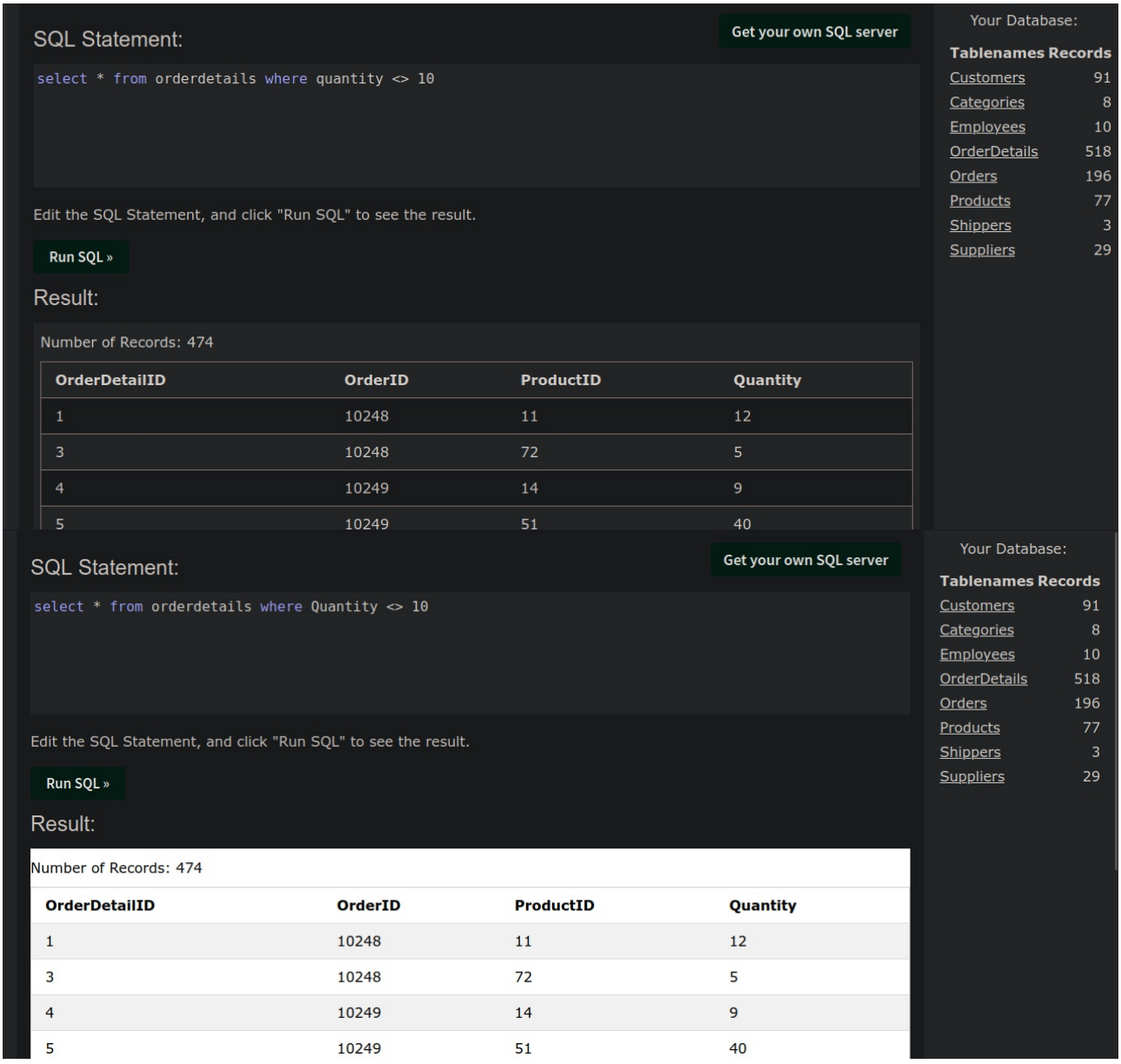


- Less than equal to 10.

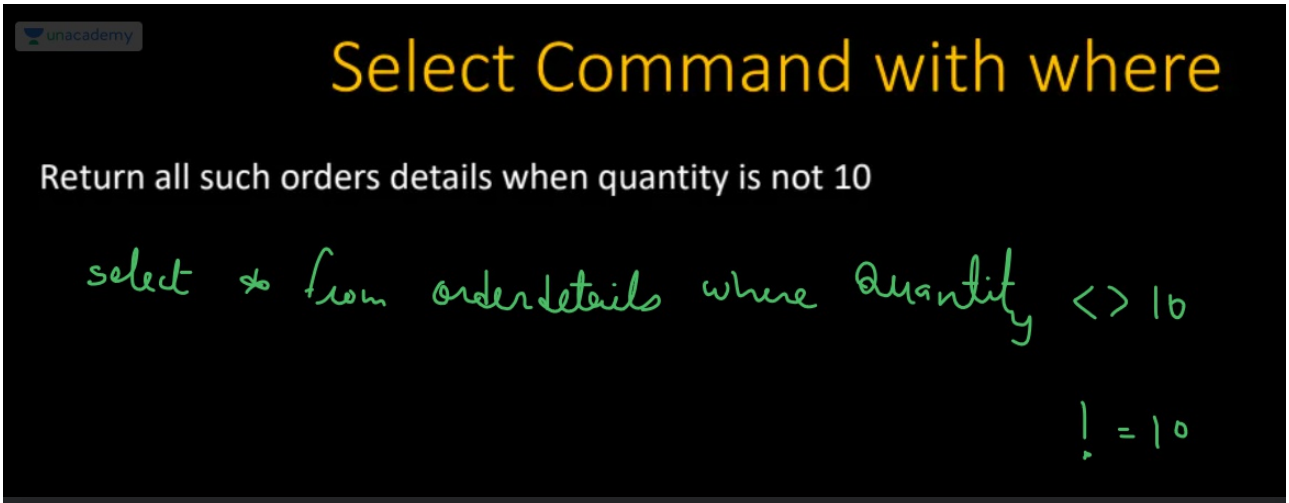

unacademy
Select Command with where
Return all such orders details when quantity is less than 15
Select * from orderdetails where Quantity <15
SQL Statement:
Get your own SQL server
select * from orderdetails where quantity < 15
Edit the SQL Statement, and click "Run SQL" to see the result.
Run SQL »
Result:
Number of Records: 166
OrderDetailID OrderID ProductID Quantity
1 10248 11 12
2 10248 42 10
3 10248 72 5
4 10249 14 9
Your Database:
Tablenames Records
Customers 91
Categories 8
Employees 10
OrderDetails 518
Orders 196
Products 77
Shippers 3
Suppliers 29

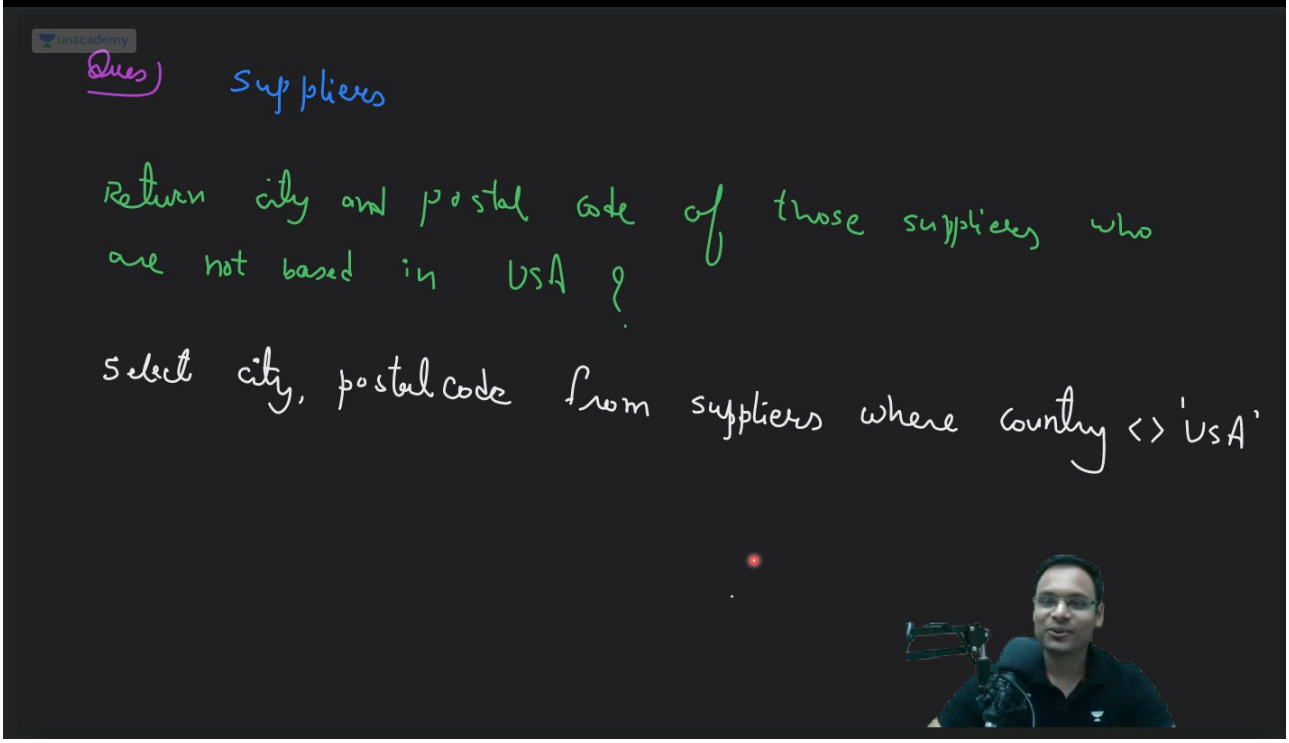
unacademy
Select Command with where
Return all such orders details when quantity is less than 15
Select * from orderdetails where Quantity <15

- Less than 15



- Not equals to 10.





- select city,postal from suppliers where country <> 'USA:

Logical Operators

1. AND -> True when conditiom 1 and condition both are true.
2. OR -> True when either of the conditions are true.
3. NOT

Select Command with where

Logical Operators:

1. AND
2. OR
3. NOT

Cond1 AND cond2 => True when cond1, Cond2 both are true

Cond1 OR Cond2 => — || — either of condit's is true

Select Command with where

Logical Operators:

1. AND
2. OR
3. NOT

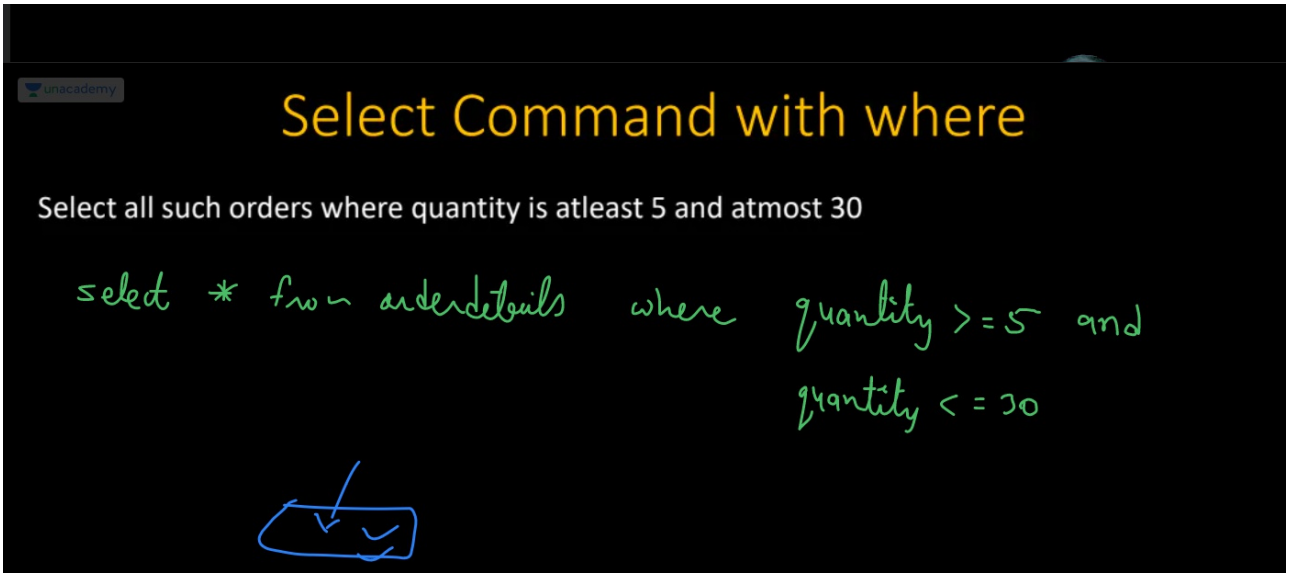
Cond1 AND cond2 => True when cond1, Cond2 both are true

Cond1 OR Cond2 => — || — either of condit's is true

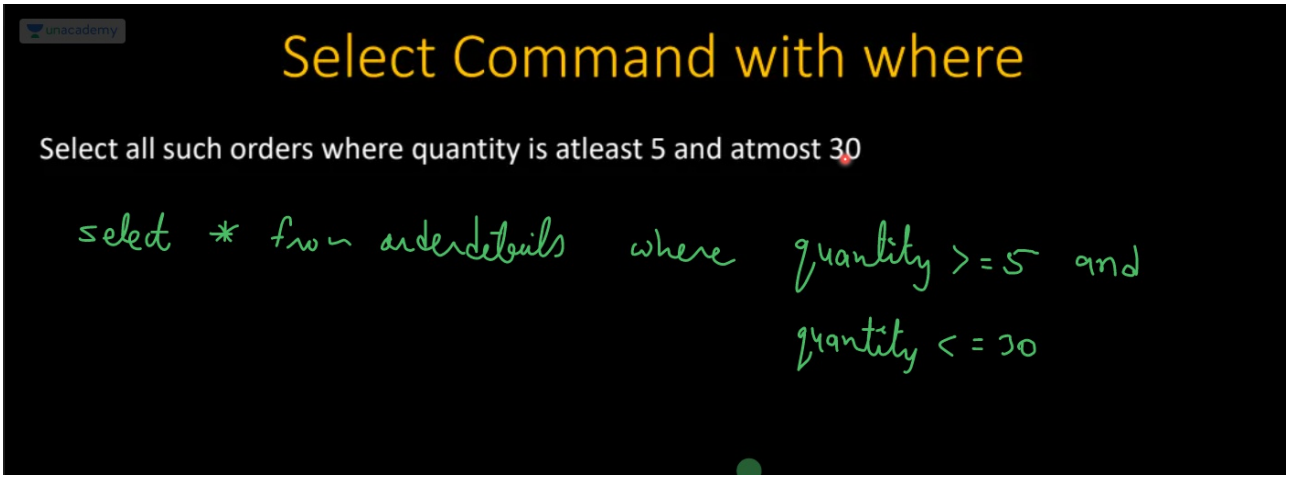
Select Command with where

Select all such orders where quantity is atleast 5 and atmost 30

```
select * from orderdetails where quantity >= 5 and
quantity <= 30
```

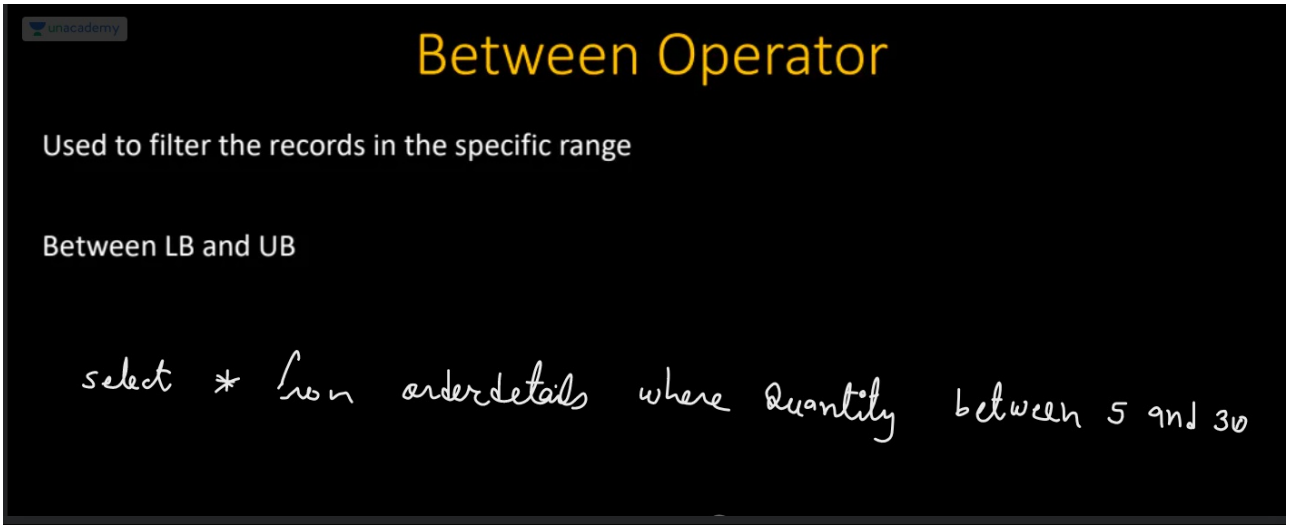


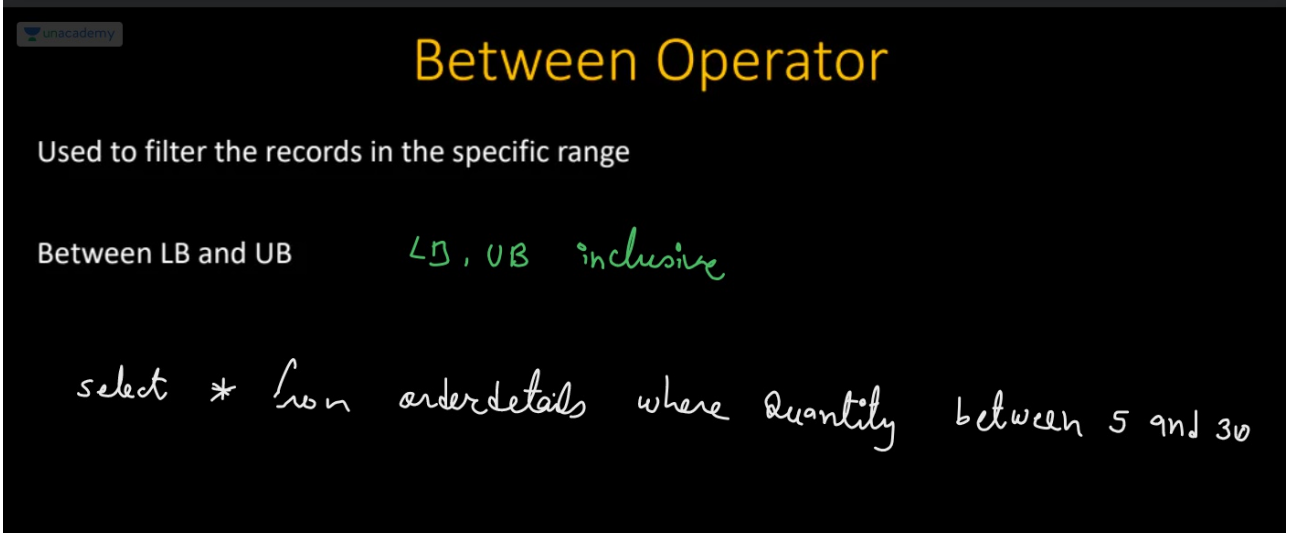
- Quantity is atleast **5** and atmost **30**.



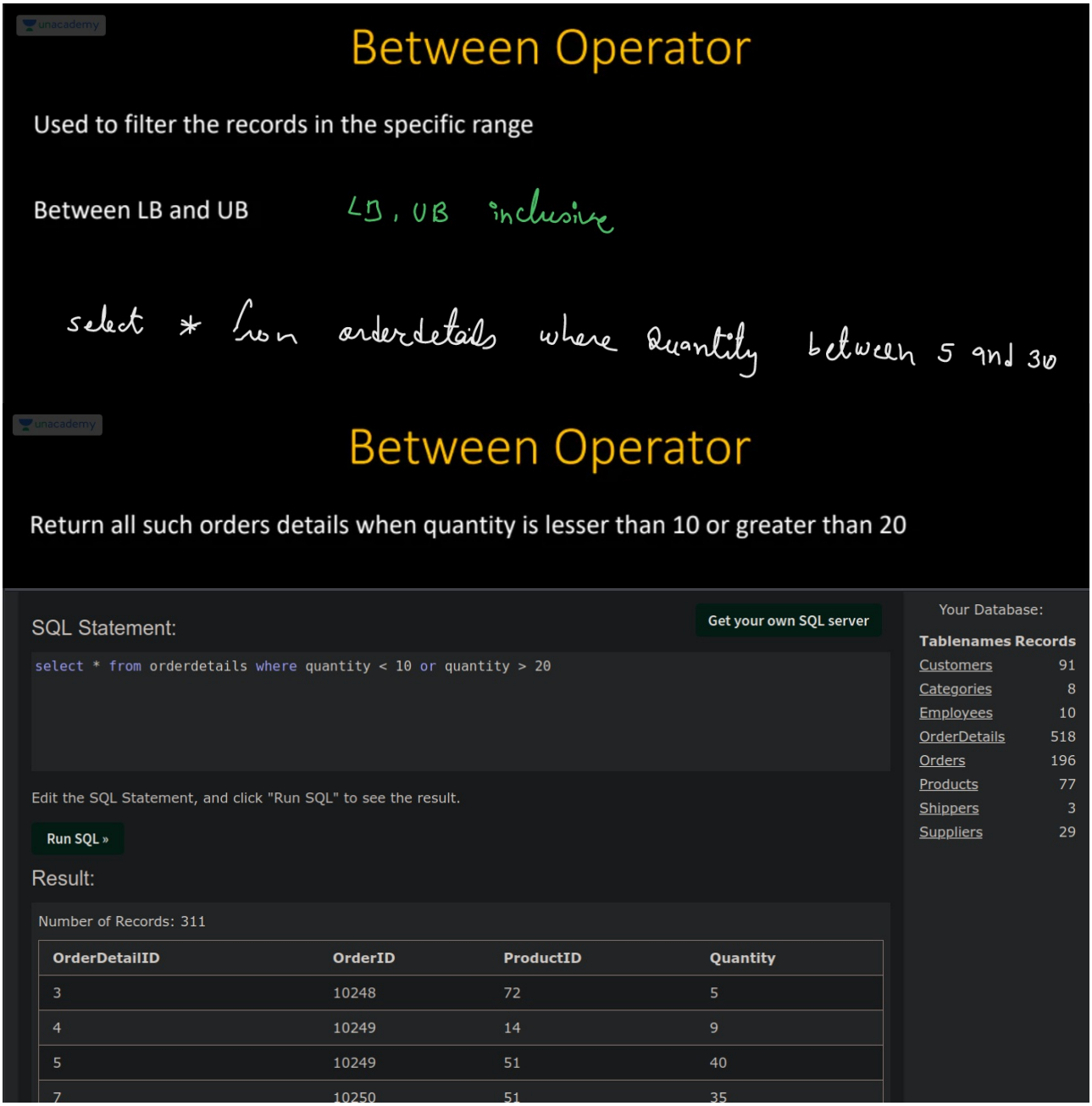
Between keyword

- Between lowerbound(LB) and upperbound(UB).
- lowerbound(LB) and upperbound(UB) are **inclusive(included)**.
- Used to filter the records in the specific range.

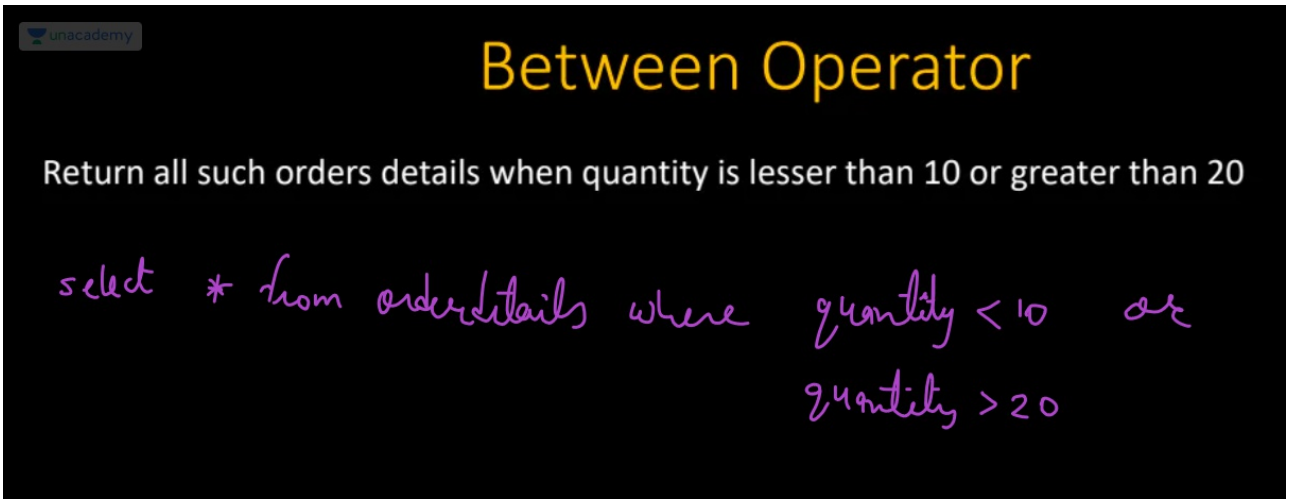




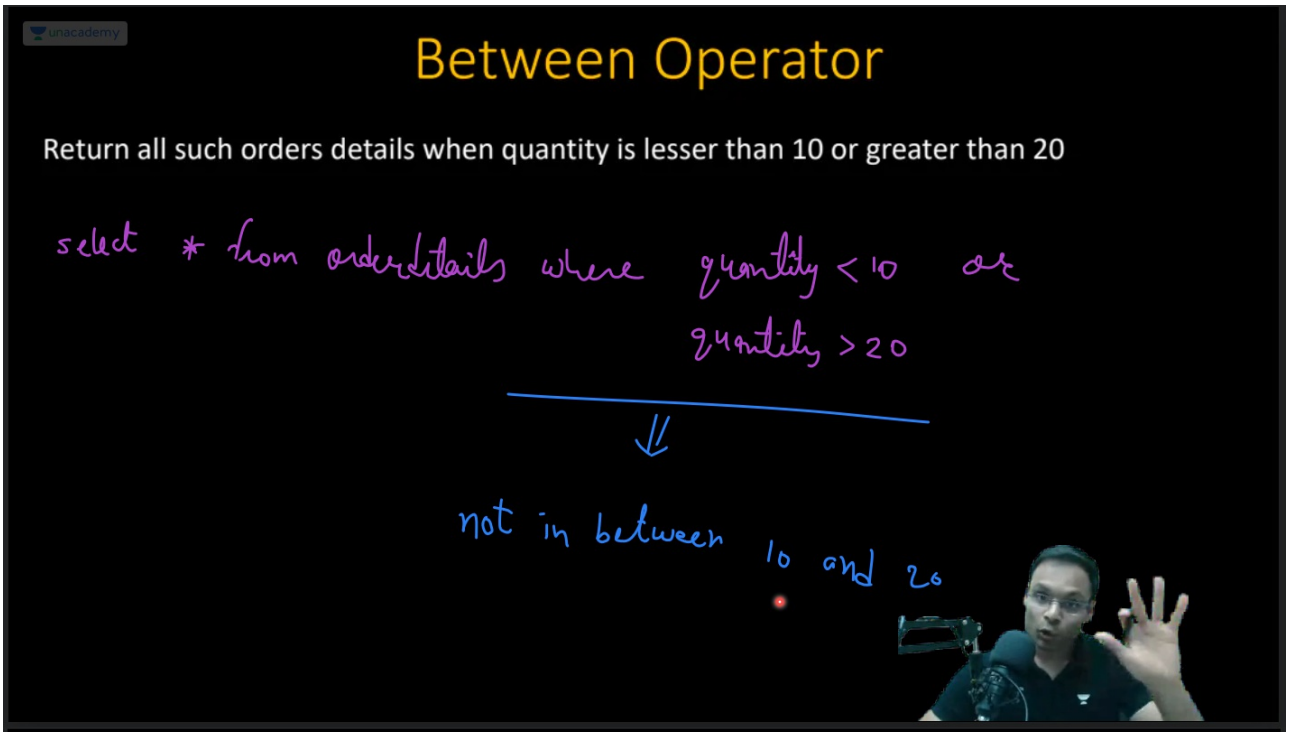
- select * from orderdetails where Quantity between 5 and 30.

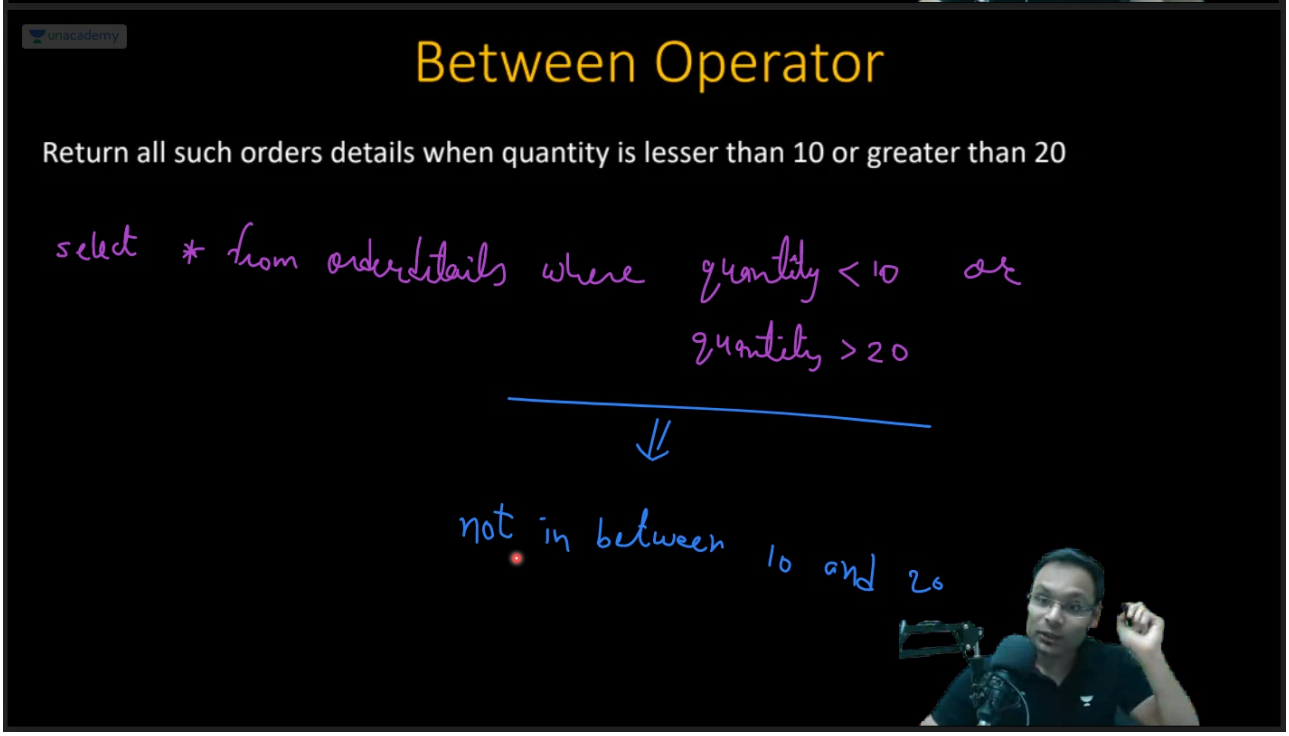


- select * from orderdetails where quantity < 10 or quantity > 20



- less than 10 or greater than 20.





- Another way

SQL Statement:

Get your own SQL server

```
select * from orderdetails where quantity not between 10 and 20
```

Edit the SQL Statement, and click "Run SQL" to see the result.

Run SQL »

Result:

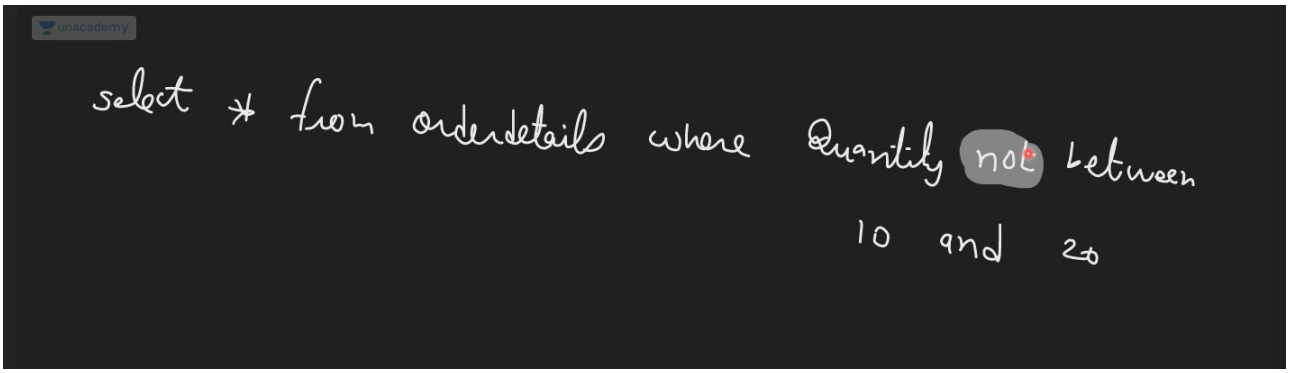
Number of Records: 311

OrderDetailID	OrderID	ProductID	Quantity
3	10248	72	5
4	10249	14	9
5	10249	51	40
7	10250	51	35

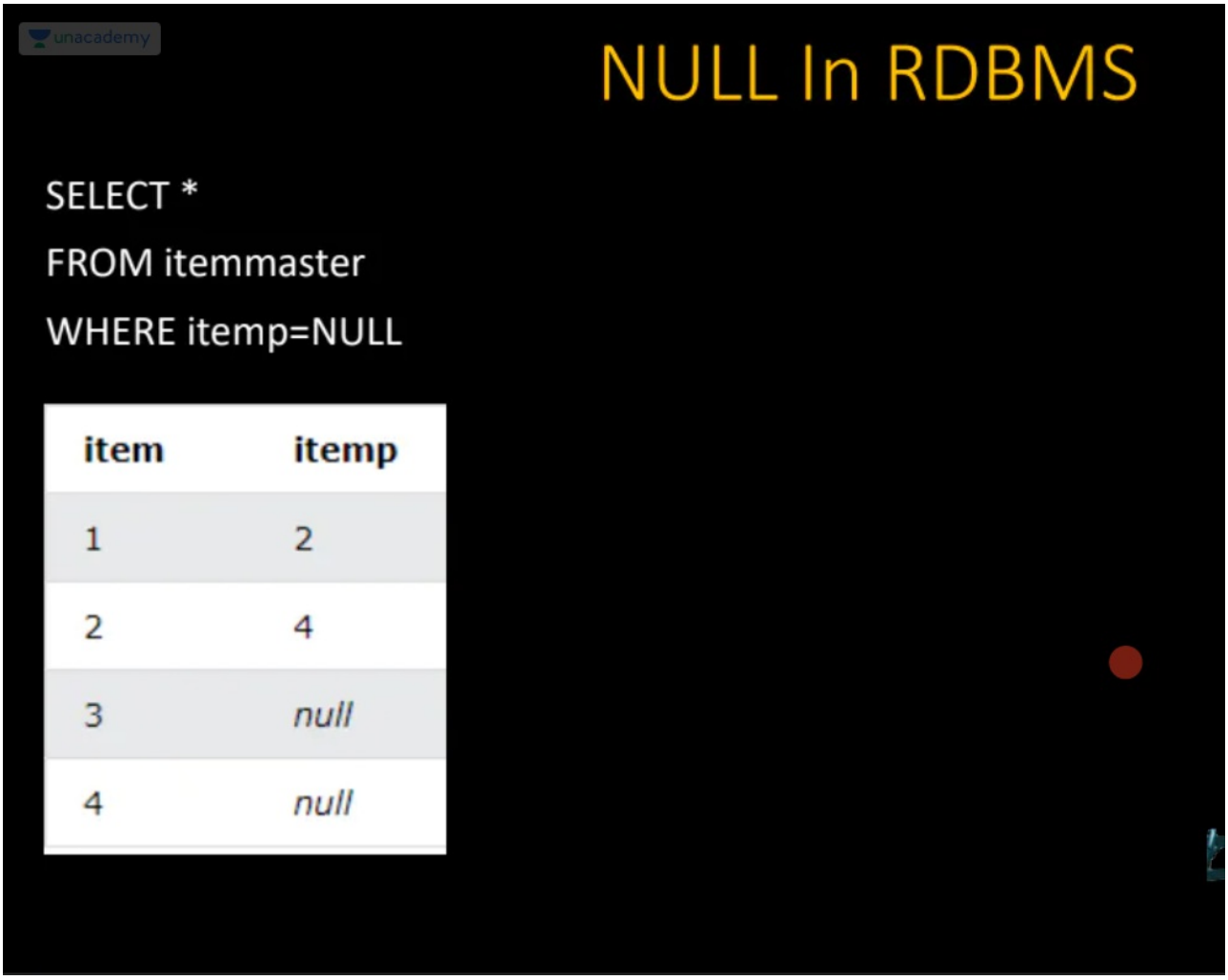
Your Database:

Tablenames	Records
Customers	91
Categories	8
Employees	10
OrderDetails	518
Orders	196
Products	77
Shippers	3
Suppliers	29

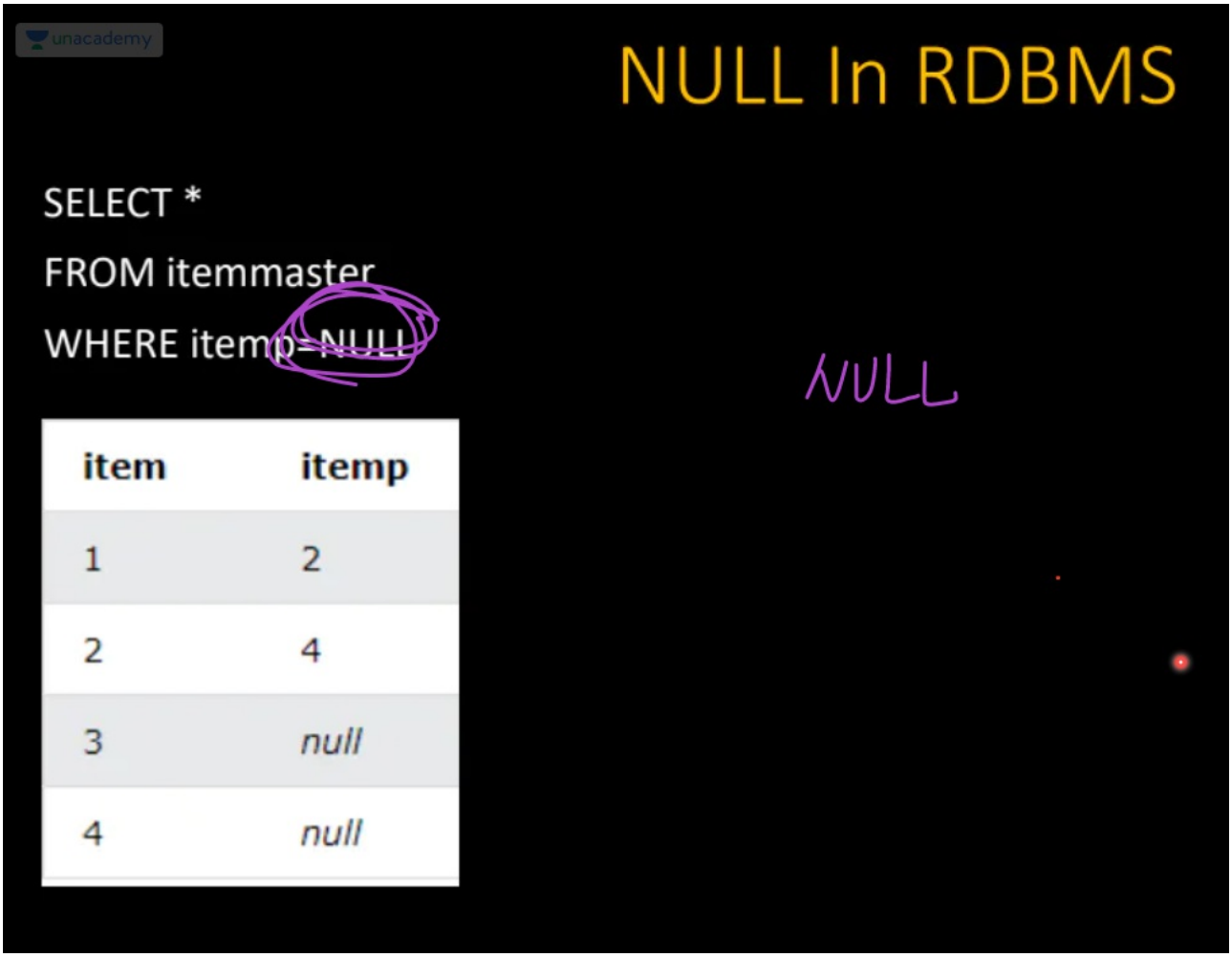
- same
- select * from orderdetails where quantity not between 10 and 20



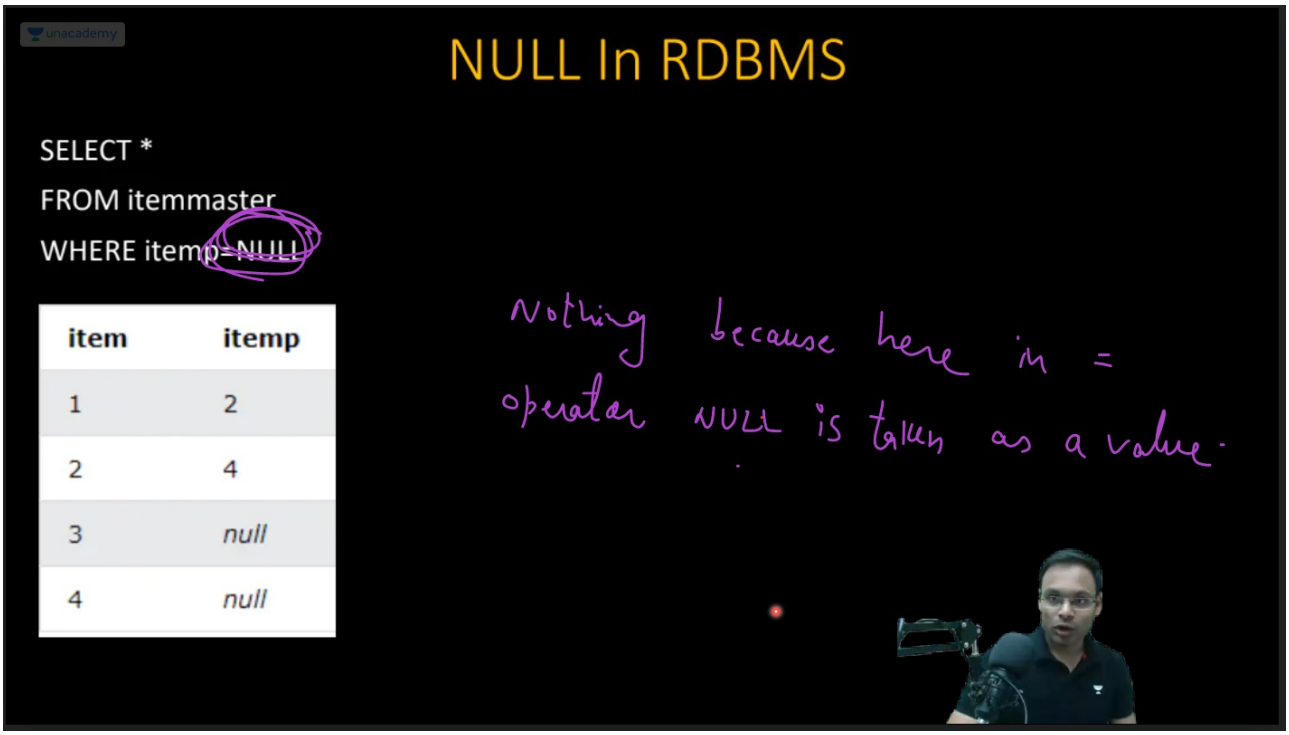
NULL in RDBMS



- Will not work. **NULL** is representation of string and not **nothing**.



- We have to use the **is NULL** keyword here.



- Nothing because here in **=** operator, **NULL** is taken as a value.
- To compare **NULL** values, or to find where **NULL** are there, we have to use **is NULL**

keyword.

- select * from itemmaster where itemp is NULL.

unacademy

NULL In RDBMS

```
SELECT *
FROM itemmaster
WHERE itemp is NULL
```

item	itemp
1	2
2	4
3	*null*
4	*null*

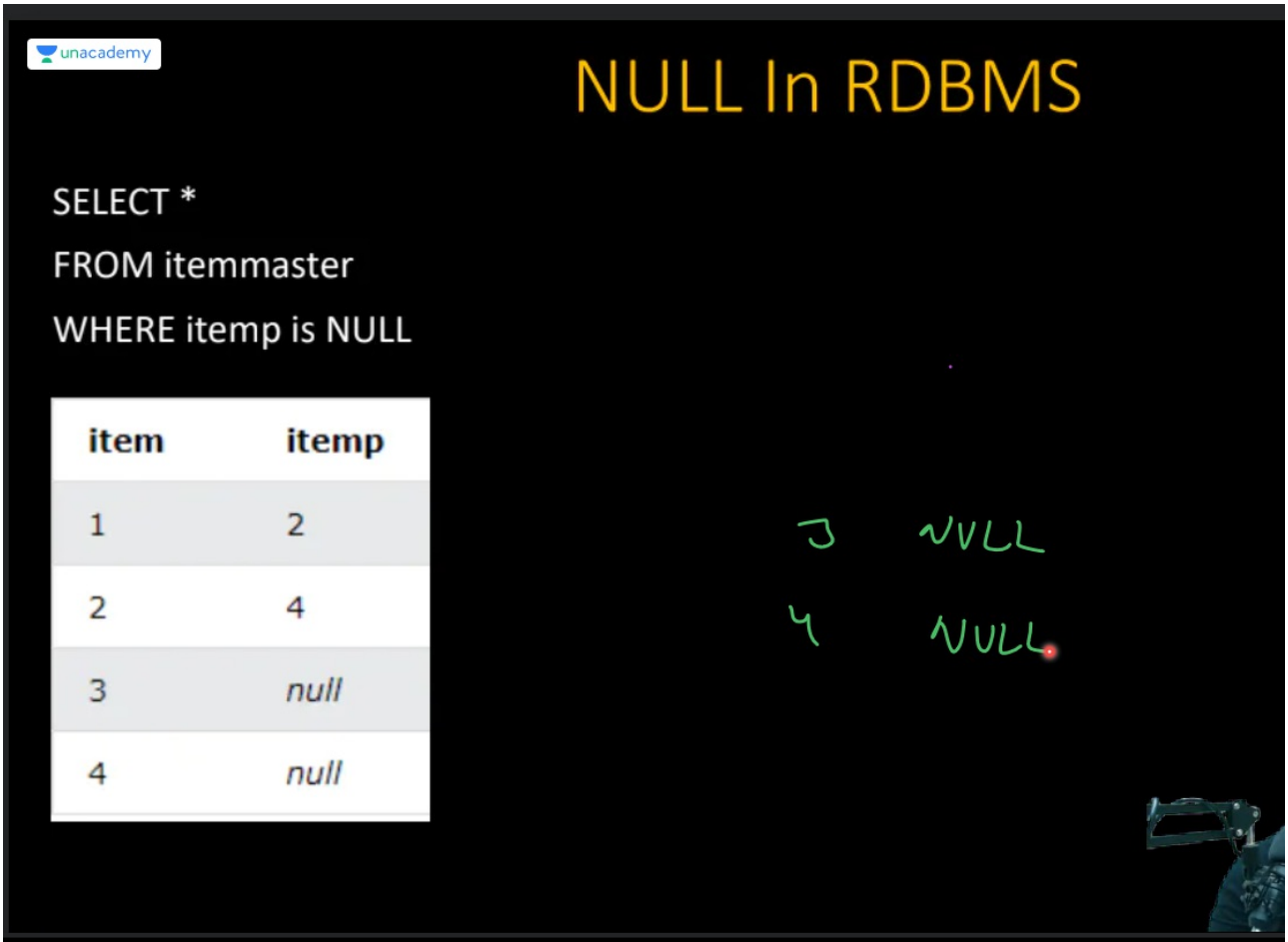
unacademy

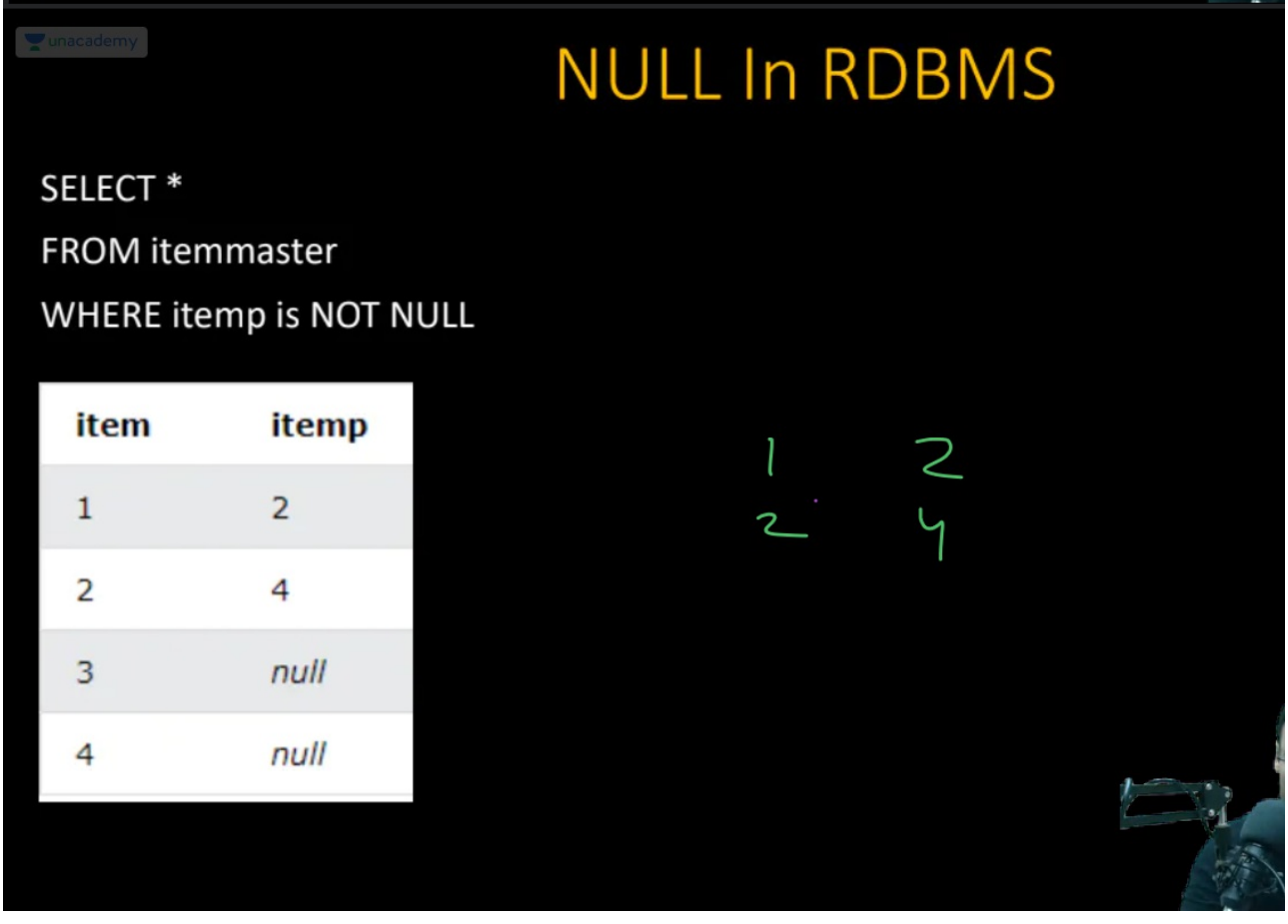
NULL In RDBMS

```
SELECT *
FROM itemmaster
WHERE itemp is NOT NULL
```

item	itemp
1	2
2	4
3	*null*
4	*null*

- If we want **NOT NULL**.

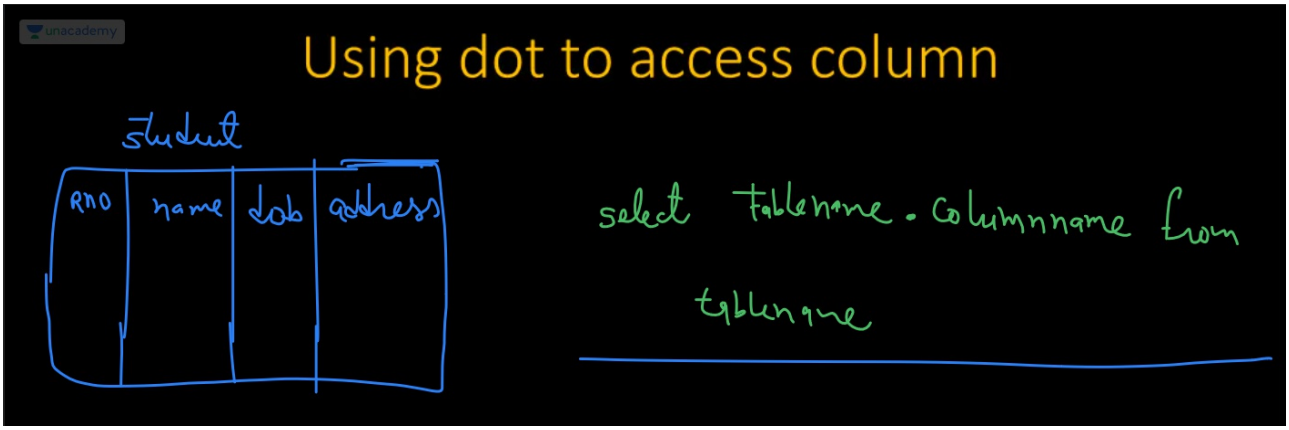




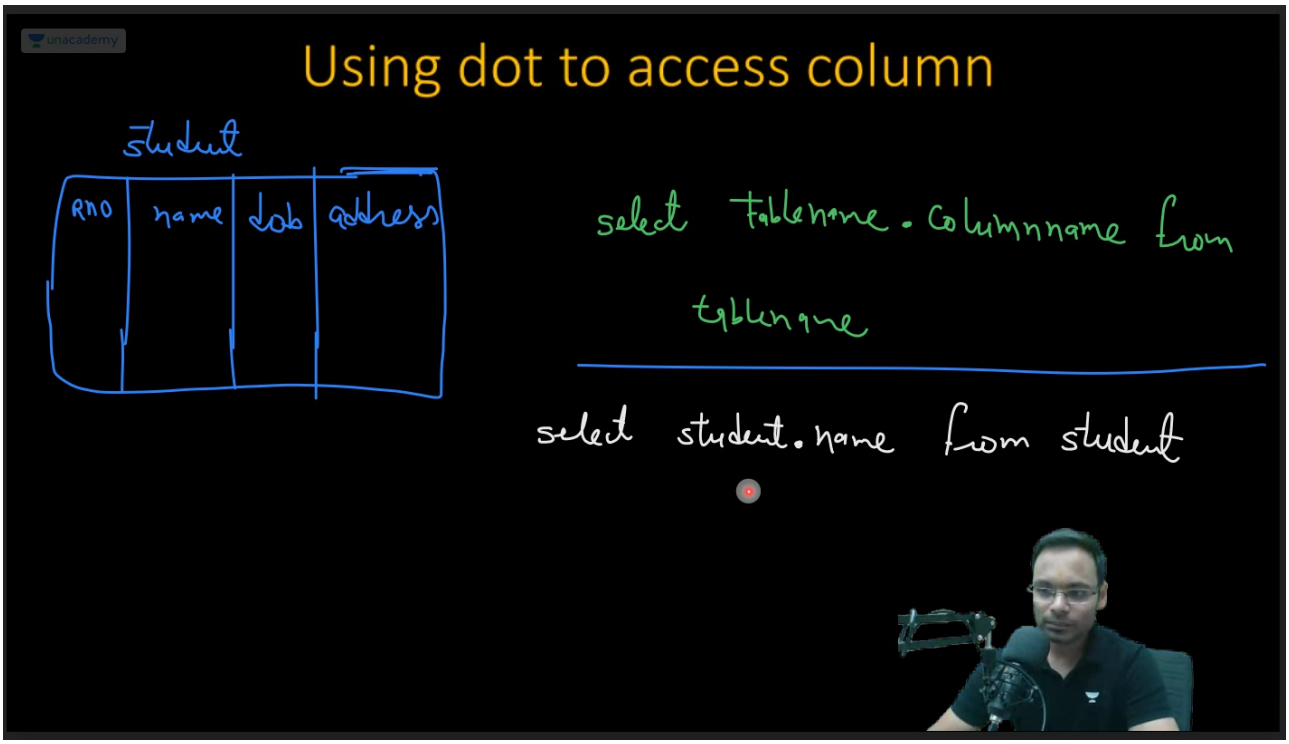
- Outputs from SQL query.

sql part-III (7) [26th June 2023]

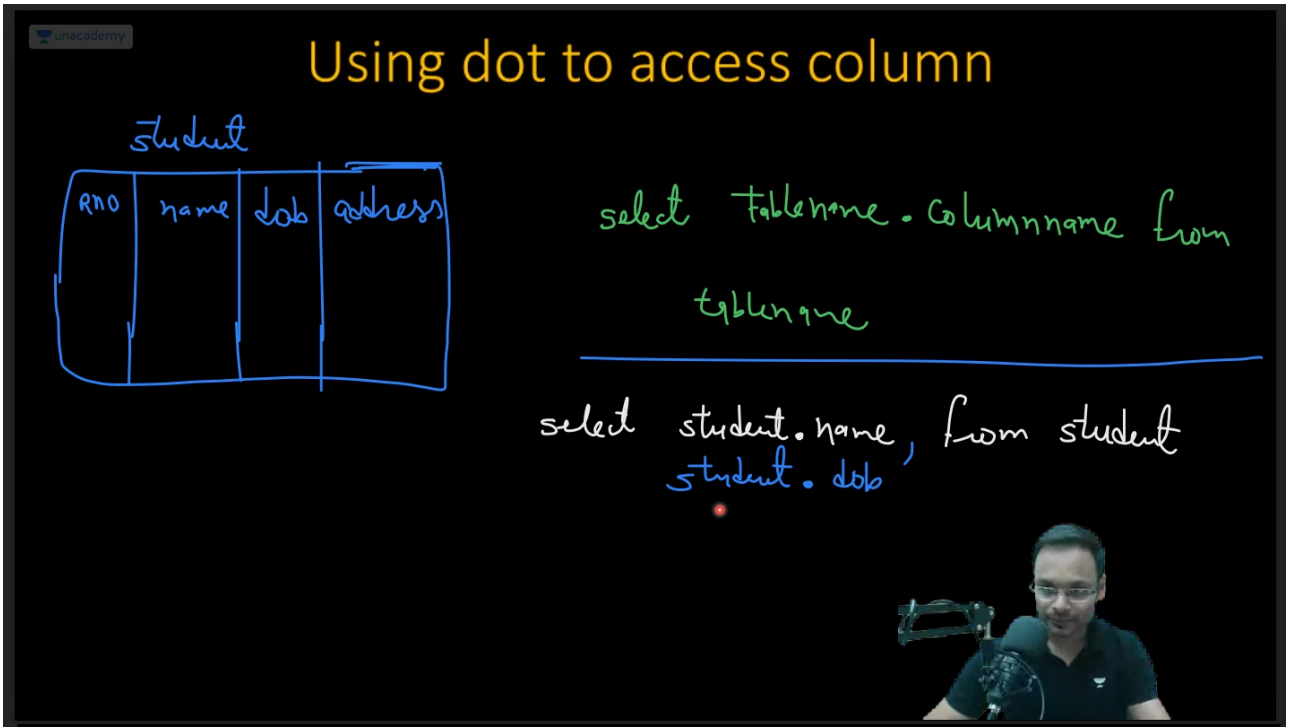
- In a single table, two columns cannot have exact same name.

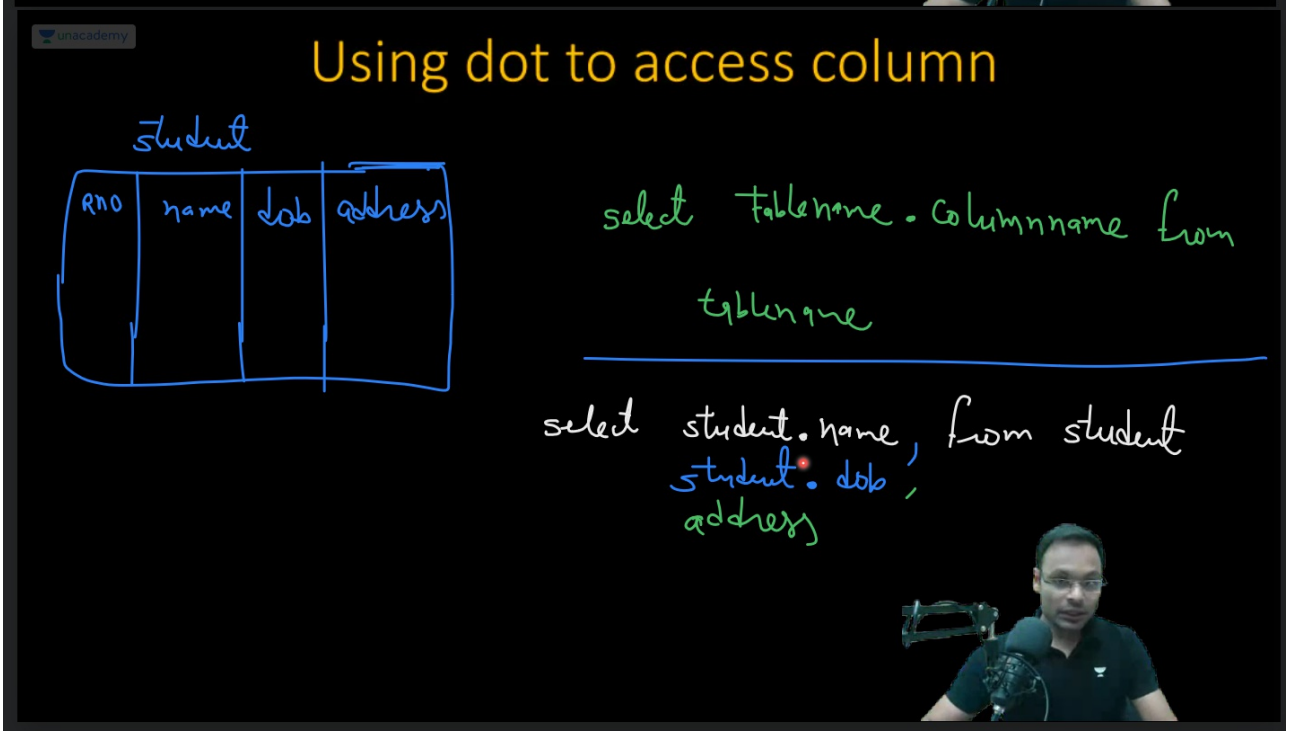


- Access column using **dot(.)**.



- select student.name from student;
- select student.name, student.address from student;





- Why is **where** used?

> For **filtering rows** or **filter** out some rows.

> We are writing the **condition** so that we can **filter** out some rows which we need, to get the **specific rows**.

- **= NULL** -> We are comparing with the string NULL or "NULL".
- We have to write **is NULL** to compare with **NULL**, nothing value.

NULL In RDBMS

SELECT *
FROM itemmaster
WHERE itemp is NULL

= NULL

is NULL

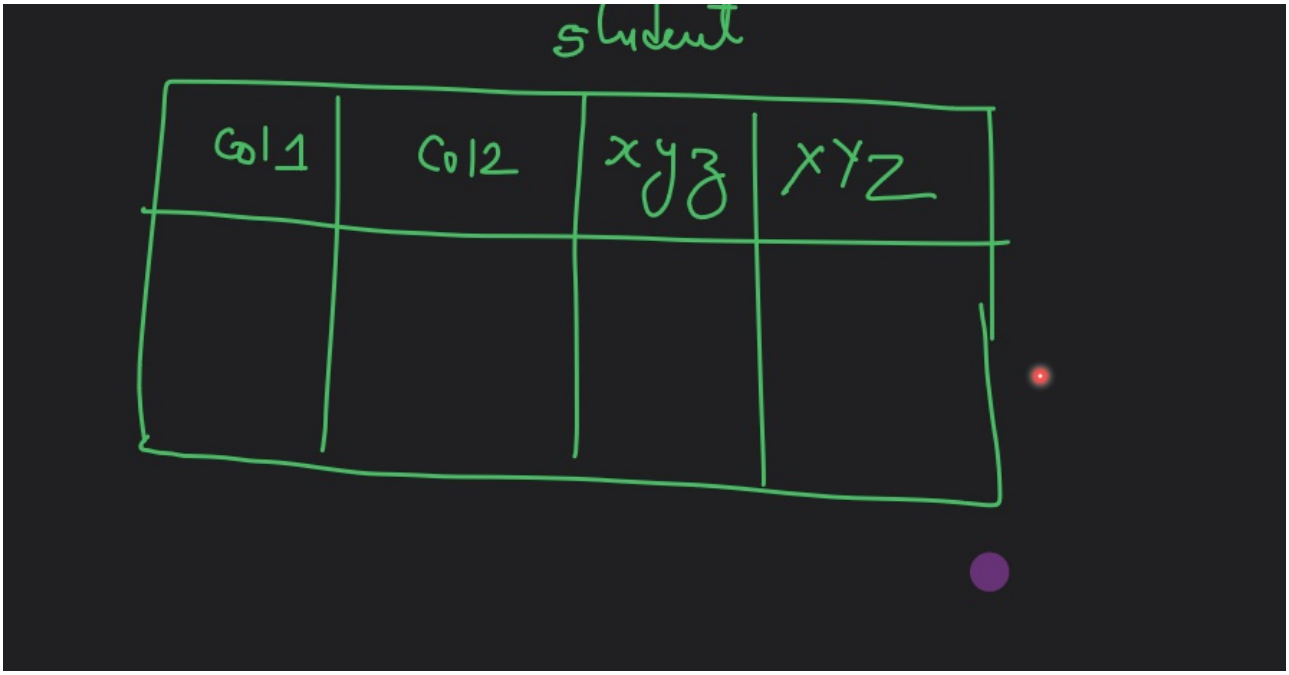
item	itemp
1	2
2	4
3	*null*
4	*null*

NULL In RDBMS

SELECT *
FROM itemmaster
WHERE itemp is NOT NULL

item	itemp
1	2
2	4
3	*null*
4	*null*

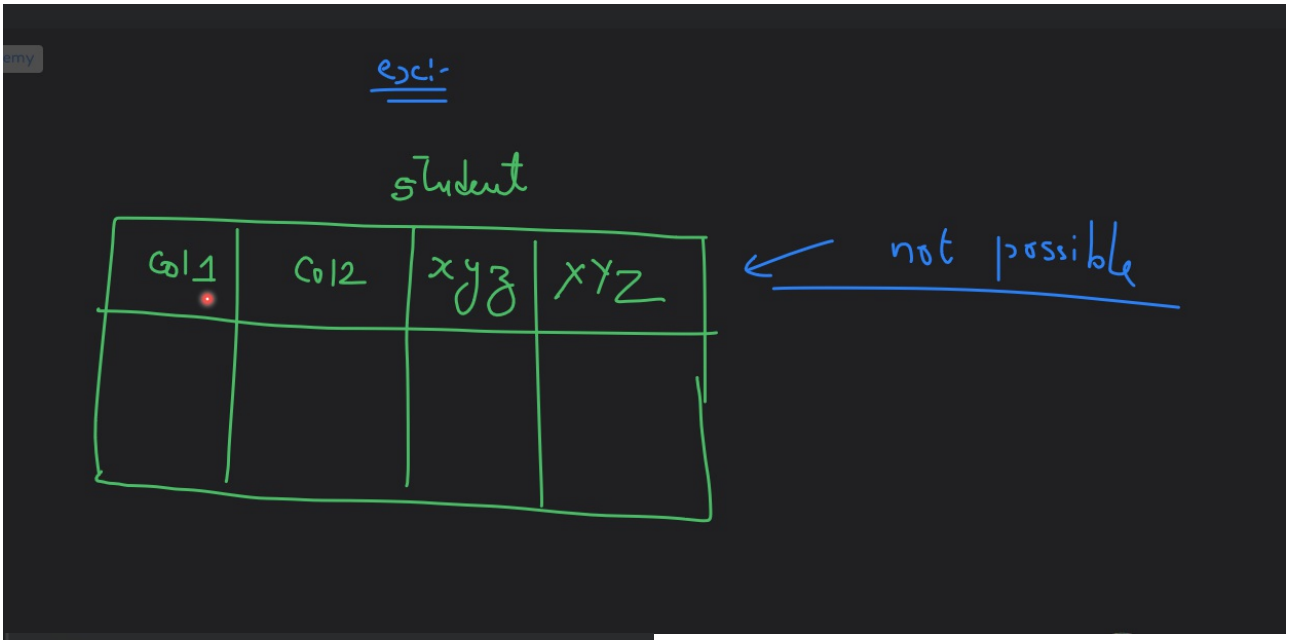
Esc!-

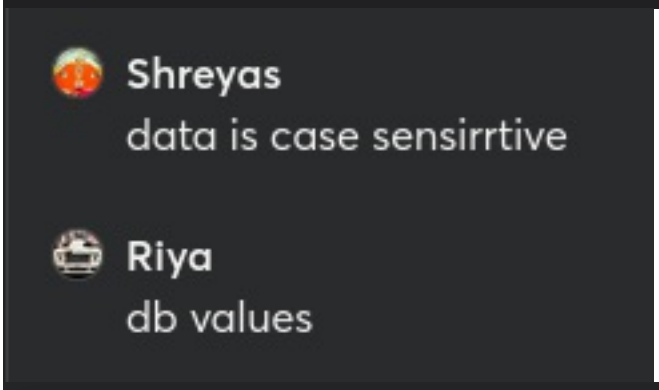


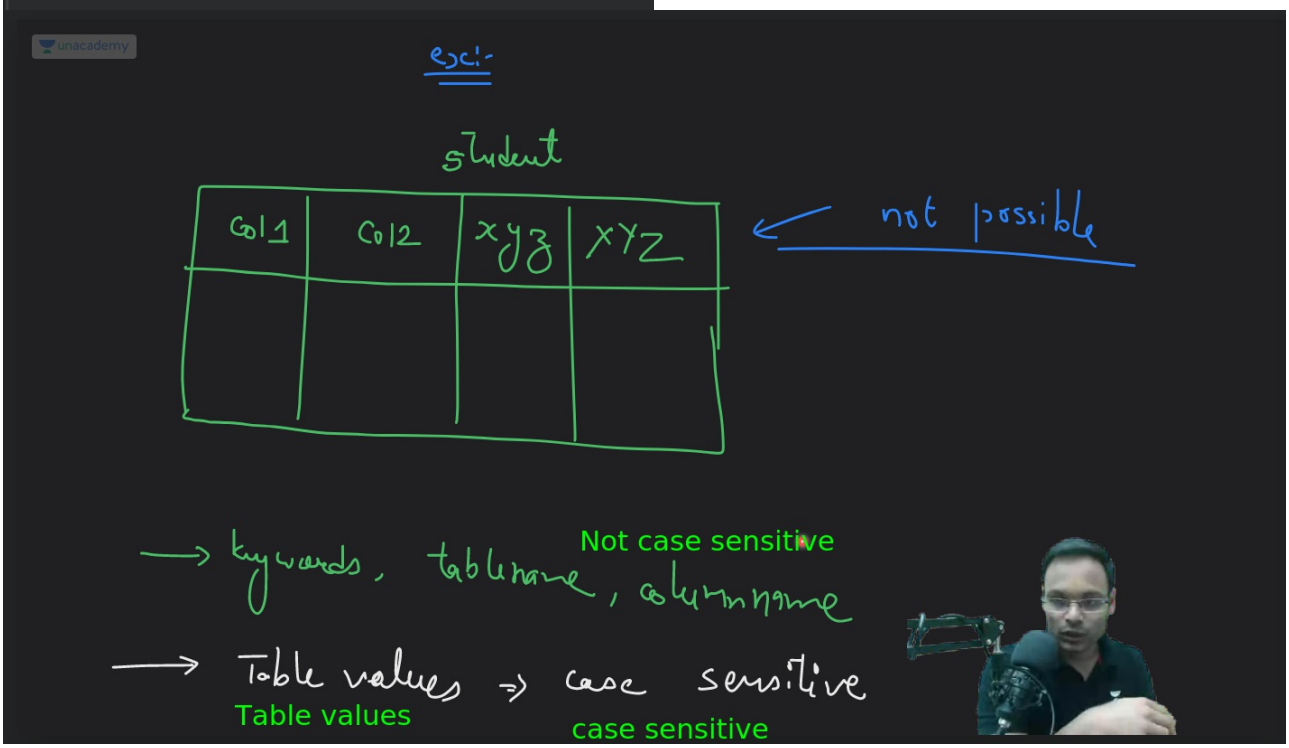
- Is the table possible or not?

> **NO**, the table is not possible.
> As **Xyz** and **XYz** are both same as there is **no case sensitivity**. We also know that **same name columns** are also not possible.

- Column name, table name, keyword are **not case sensitive**.

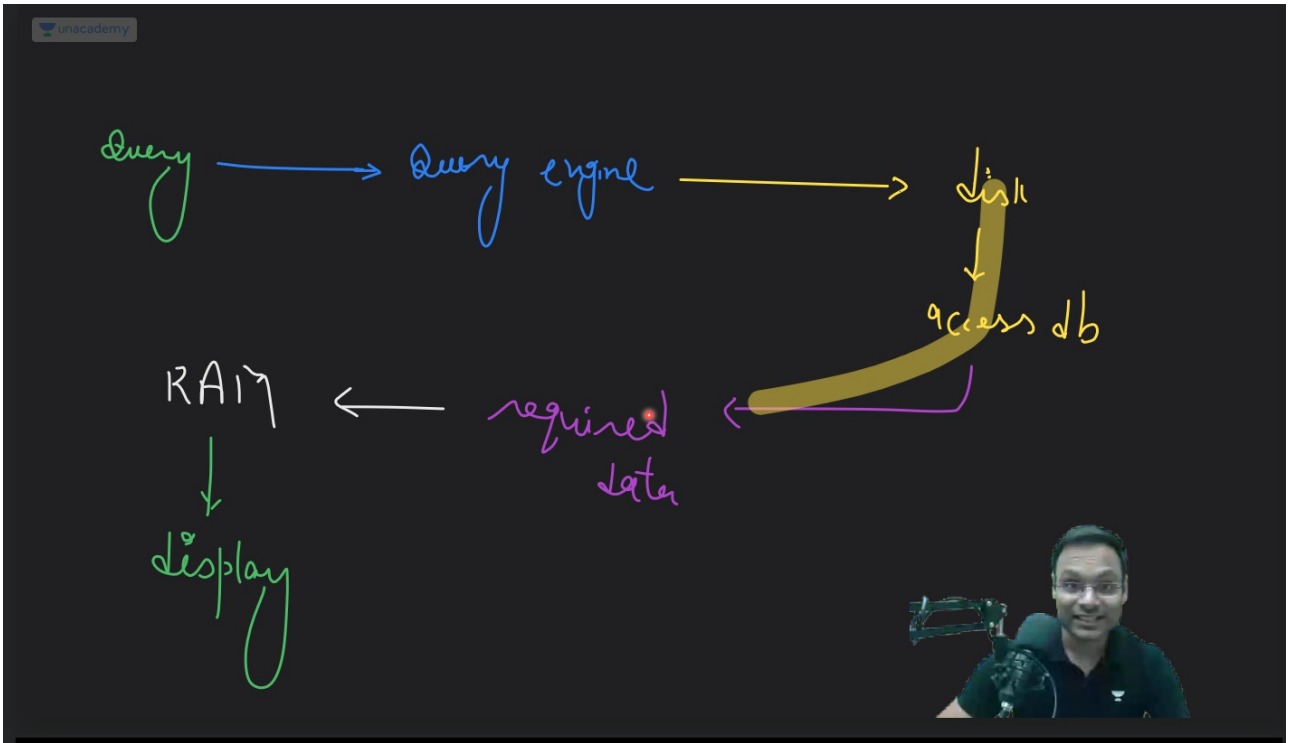


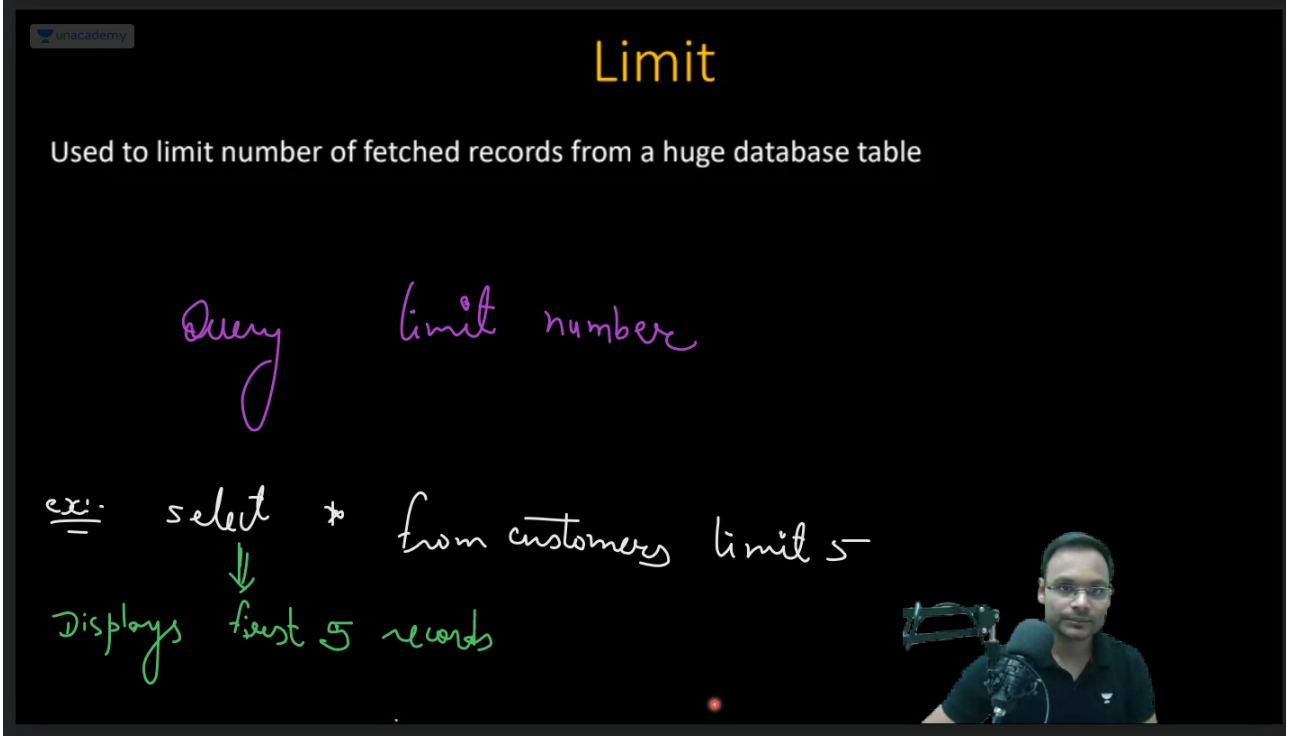




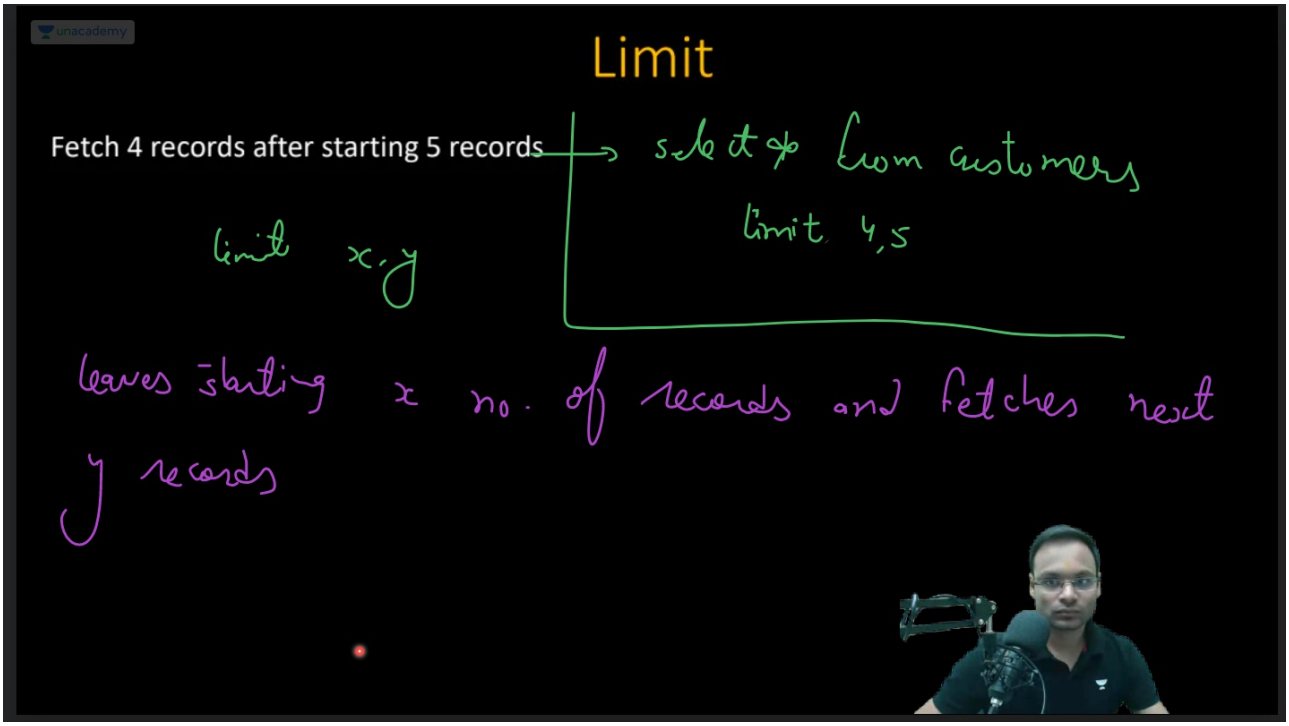
Limit

- Used to limit number of fetched records from a huge database table.

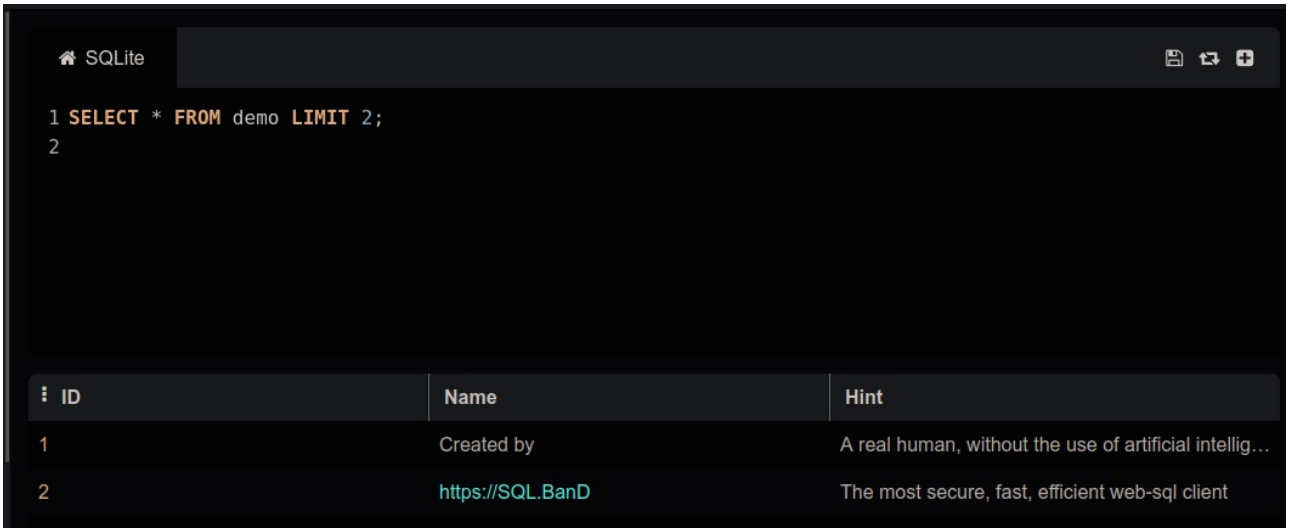




- select * from customers limit 5;



- Fetch **4 records** after starting 5 records
- limit x,y
- leaves starting **x** no. of records and fetches next **y** records.
- select * from customers limit 5, 4



- select * from demo limit 8, 12;

- select * from demo limit 5, 10;

Aggregate Functions

- Performs a calculation on multiple values and returns a single value.

min()

- SELECT min(price) FROM Products;

max()

- SELECT max(price) FROM Products;

sum()

avg()

- SELECT min(customername) FROM Customers;

- Arrangement

- **min() and max()** -> Works both on char data and numbers. For char data it returns the first or last record according to **lexicographic order**.
- **avg() and sum()** -> Will work only on numbers. Returns zero(0) for others.

Count()

- SELECT count(price) from products

- Return how many customers are from USA?

select(customerID) from customers country= 'USA'.

- Average -> Summation/Count -> (5 + 7 + 7 + 7)/4 -> 26/4 -> 6.5.
- We will not count the **NULL** as there is nothing over there.

- Select count(*) from itemmaster

8 rows will come.

- Aggregate functions return a **number**.

- count(*) will look at **no. of rows** that's why it will be **8 rows**.

- Yes.
- create table itemmaster(item int, itemp int)
- **Creating table**
- insert into itemmaster values(1,2)
- **Inserting values into the table**.

- NULL means **nothing**.
- As it is **nothing**, so no need to include it when counting average.

- For sum we will take the score of students who have scores and ignore the **NULL** rows.

- Question
- select * from customers where country = 'USA' or country = 'Germany' or country = 'UK';

IN operator

- select * from customers where country in ('USA', 'Germany', 'UK')
- select * from customers where country in ('USA', 'UK', 'France')

- List of all customers which are not from **Mexico or Sweden**.

- select * from customers where country not in ('Mexico', 'Sweden')
- **Not** is in the **full condition**.

- Sign change

- select * from customers where country != 'Mexico' and country != 'Sweden';
- select * from demo where name <> 'Created by' AND name <> 'SQL Online' AND name != 'Twitter'
- select * from demo where name != 'Created by' AND name != 'SQL Online' AND name != 'Twitter'
- Different way of writing the above query.
- != -> <>
- Both give same output.
- Link -> https://blog.sqlauthority.com/2013/07/08/sql-difference-between-and-operator-used-for-not-equal-to-operation/

- Both give the **same output**.

- Subqueries

Alias

- Used to give a temporary name toa table, or a column in a table
- No spaces allowed

- Alias should be unique.
- select * from customers as cust
- select * from customers cust

- select cust.customerid ID from customers cust.

- select avg(price) from products
- select avg(price) as Average from products

- If we give **keywords** in **single or double quotes** then they work as aliases.

- Alias examples, where can we use alias and where we cannot.

Query Summary

- from -> where -> select
- Limit runs at the end.

SQL Joins

- Needed to retrieve records from more than one table collectively.

Types of Join

- Inner join -> As called as **natural join**.
- Left Join
- Right Join
- Full Join

- Self Join
- Cartesian Product

Cartesian Product

- Cartesian Product will return the product of each row from one table to all of the other rows in the different table.
- When we do **joins**, how many columns we have?

> Left side table + right side table combination is the **no. of columns*.

- It return combination of each row of student table .

- No. of rows -> 4 * 3 -> 12.

- Cartesian Product between **customers and employees** table.
- select customers.customername, employees.EmployeeID from customers, employees

- select customers.customername as CustomerName, employees.EmployeeID from customers, employees

- Cartesian Product is **costly operation**.

Doubt clearing session sql queries(8) [28th June 2023]

- Table design refinement.

- Independence -> We can store the data anywhere.
- Flexibility
- Data Integrity

- Tuples -> Rows.

1. select * from customers where country IN ("Germany', 'Berlin');
2. select customername, address, city, postalcode, country where contactname ='Yang Wang';
3. select * from customers where customerID <= 19;
4. select * from customers where country NOT IN ('Germany, 'UK', "USA');

- We don't know if the table is sorted/ordered using customerID or not, it is bettter to write **customerID <= 19**.

- Yes, limit could be wrong there as well.

- Yes. All rows will come.

1. select * from products where supplierid IN (1, 2, 3);
2. select productname from products where price between 5 and 25;
3. select supplierid from products where categoryid = 2;
4. select * from products where supplierid =2 and price > 30;
5. select * from products where price > 50 and supplierid != 6;
6. select * from products where price < 30 and supplierid not in (2, 6);

OR

select * from products where price < 30 and supplierid != 2 and supplierid != 6;

- Cartesian product.

Inner Join

- We need **one** common column for **inner join** to work.
- Inner Join -> It will give those records that are in the left table and are matching with the column records of the right table.

- select * from table 1 Inner Join table2 on table1.column = table2.column
- select * from table1, table2 where table1.column = table2.column

- select * from student inner join library on student.rno = library.rno
- SELECT * FROM products inner join suppliers on products.supplierid = suppliers.supplierid;
- select * from products, suppliers where suppliers.supplierid = products.supplierid;

- Cartesian Product -> Cross Product

- select * from T1 inner join T2 on T1.A = T2.A

Left Join/ Left Outer Join

- Left Join/ Left Outer Join -> Entire rows of left table adnd matching rows of right table.

- select * from student left join library on student.rno = library.rno

- Result set

- Not necessary
- Min. no. of rows in result = no. of rows in left table.
- In the left table we have **4** rows and **3 rows** in the right table. How many max rows we will have after **left outer join**?

We will have **12** rows max.

All **4 + 3 -> 7** rows from the two tables are all identical then **4 * 3 -> 12**.

- If all the values are **distinct**?

The no. of rows will be -> No. of rows in left table

- Min. no. of rows in result = no. of rows in left table
- It is **true** when common column has unique values in both tables.

Right Join / Right Outer Join

- Right Join / Right Outer Join -> All rows of right table and only matching rows from left table.
- select * from student right join library on student.rno = libbrary.rno

- Resultant set
- Left and eight outer join are **complement** to each other.

- Yes.

Full Join/ Full outer join

- select * from student full outer join library on student.rno = library.rno

- **Full Join/ Full outer join** -> We taking all the rows from left and right table and, whatever matching data left table has with right table we will write and if it is not there we will write **NULL**. We will take the matching data from right table which are matching with left table and if not there we will write **NULL**.
- Inner Join -> Only the matching data we will get from the left and right tables from the **common column** between the two tables.

> We can say that **Inner join** is a subset of **left and right** joins.

- Left Join
- Right Join
- Full join -> It is the **superset** of **left and right** join.
- They are **subsets** and not proper subsets.

1. Result set of inner join ->(subset) -> Result of left outer join
2. Result set of inner join ->(subset) -> Result of right outer join
3. Result set of right join ->(subset) -> Result of full join
4. Result set of left join ->(subset) -> Result of full join
5. Result set of inner join ->(subset) -> Result of full join

- Question
- Option **D**

- Yes.

- Question
- Option **D**

- Cartesian Product.

- Yes.

SQL part-IV (9) [29th June 2023]

- Yes.

- select customername from customers inner join orders on customers.customerid = orders.customerid
- select customername from customers, orders where customers.customerid = orders.customerid

- select distinct(customers.customerid), customername from customers, orders where customers.customerid = orders.customerid

- select orderid from orders inner join employees on orders.employeeid = employees.employeeid where firstname= 'nancy'

- Join will be on the **filtered** data.

- Ordering.

- select dname from drivers inner join driver on drivers.did = drives.did;
- select distinct(dname) from drivers, driver where drivers.did = drives.did;
- We want those drivers, that are both in **drivers and drives** table. That's why we did **inner join**.
- select drivers.did, dname from drivers, driver where drivers.did = drives.did;

- Find name of all those students who have enrolled for atleast one course?
- select student.sid, sname from student, enrolled where student.sid = enrolled.sid;

- Find name of all those students who have enrolled for course with id=5.
- select distict student.sid, sname from student, enrolled where student.sid= enrolled.sid

and courseid = 5;

- find name of all those students who have enrolled for atleast one course during 2022.
- select distinct student.sid, sname from student, enrolled where student.sid = enrolled.sid and duration = 2022;

- Left Join
- list out all student information and if they enrolled for any courses then those information also.
- select * from student left join enrolled on student.sid = enrolled.sid;

- find name of all students who have enrolled for course 'DBMS'.
- select distinct student.sid, student.sname from student, course, enrolled where student.sid = enrolled.sid and course.courseid = enrolled.courseid and coursename = 'DBMS';

- Yes.

Self Join

- Point.
- select name from employee where eno = managerid -> **Wrong **.
- Find name of all such employees who are managers also.
- We need to make a copy of the **table**.

- If we have to do joining then we will do **joining** with the **same table**.
- But we don't have **two** tables.
- We have to join the table with **itself** only. It is called as **self join**.
- If we have to do join the table with **itself** then we will have to do **aliasing**. It is **compulsory**.
- We have to assume they are **two different** tables and then do **aliasing**.

- Find name of all such employees who are managers also.
- select distinct E1.name from employee E1, employee E2 where E1.Eno = E2.ManagerID

- Yes.

- Find name of all employees who are having managers?
- select distinct ename from employee where managerid is not NULL.

- No.

- Matched with **NULL**. If **matched** with **NULL** then **return**, otherwise **no return**.

Relational db design functional dependency(10) [30th June 2023]

Group By clause

- Used in collaboration with the select statement to arrange identical data into groups.
- Select information in **groups**.

- It will give all **rows and columns**. All records of the **student** table will be given.

- Select * from demo group by name;

- For all of the **unique values** of **dob** column, we are creating **separate** groups for them.

- select * from customers group by country;
- select country, count(customersID) from customers group by country;

- Yes.

- Yes.

- 4 rows are selected.
- select * from student group by Rno;

> **5 rows(1,2,3,4,5)** will come.

- select * from student group by name;

> **3 rows(A, C, D)** will come.

- Automatically **distinct**.

- Syntax.

- select categoryid, sum(price) from products group by categoryid;

- The column on which group by is done, that must be included in select. [**IMPORTANT**]

- Yes.

- select categoryid, min(price) from products where price > 20 group by categoryid;

- select categoryid, sum(price) from products where sum(price) > 200 group by categoryid; [**Wrong query**]
- We are putting the condition after the **group by** clause but **where clause** runs before **group by** clause. We need a new clause here, which is **Having** clause.

Having clause

- select categoryid, sum(price) from products group by categoryid having sum(price) > 200.
- **Where clause** gives the **condition** in which the rows are **filtered**, and on those **filtered** rows which matches the **where** condition, on them the **group by** clause runs.
- Having clause **sequence**.
- Where VS Having clause **difference**.
- Yes, **having** cannot be written alone without **group by** clause.

Order By

- select * from customers order by country;
- select * from customers order by country desc.
- select * from customers order by country, postalcode desc.
- Default ordering is **ascending** order.
- Order by doesn't make any changes to the table.
- Ordering the fetched data.
- Order by runs after **select** clause.

- If two **country** names are **same/equal** then those rows are **sorted** using **postal code**, otherwise we are doing the sorting with **country** only.

- select categoryid, sum(price) from products group by categoryid having sum(price) > 200 order by sum(price) desc
- select categoryid, sum(price) from products group by categoryid having sum(price) > 200 order by sum(price)

Subquery

- Query in a Query

- Subqueries can be within **select, where and from**.

1. Single row subquery -> returns single row of one column(One value)
2. Multiple row subquery -> returns multiple rows of a column(Multiple values)
3. Corelated subquery

Single row subquery

- ..... where (Inner query)

1. Run Inner Query -> Return it's result
2. Run outer query using result

- select customername from customers where country='USA'

- If countryname was directly not given, and we had to find the country name and for that we had to write a query.
- This is **subquery**.

- Fetch the name of all customers which are from same country of customer name 'The Cracker Box'
- select customername from customers where country = (select country from customers where customename = 'The Cracker Box')

- Self Join.
- Why not **self join** and why **subquery**.

- Find all such products which are having price less than price of product name 'Tofu'.
- select * from products where price < (select price from products where productname = 'Tofu)

- Multiple row subquery.

- Yes.

- Highest product price -> max(price)
- Find the product with second highest price.
- select max(price) from product where price < (select max(price) from products)
- (select max(price) from products) -> max(price)
- select max(price) from product where price -> Except **max(price)** we will get the rest rows -> Out of those rows we will select the **max(price)**.

- Same Price products.

- Multiple ways possible.

- Find the product with third highest price.
- select max(price) from product where price < (select max(price) from product where price < (select max(price) from products))

Multiple Row Subquery

- returns multiple rows of a column
- Inner query returns multiple values. So, **equal to(=), greater than (>), etc** they don't work.
- We have to use some **keywords** like **In, any, all, exist**.

Multiple row subquery

- We have to use **keywords** here.
- **=, <, >** don't work in **Multiple row subquery**.

Keywords

1. In
2. Any
3. All
4. Exist

- Find all customers that are from the same countries as the suppliers.
- select * from customers where country IN (select distinct country from supplier)

- **=** is for comparision of **one value** but here we have **multiple copuntry names**.
- So, **=** will not work.

- Will talk about it later.

- As we are getting **multiple rows** as a **result set** from the **innere query**, we cannot use **=, <,>**, we have to use the **IN** keyword.

- In -> multiple OR.
- Find all customers, that are from those countries where there is not any suppliers
- select * from customers where country NOT IN (select country from suppliers)

- Find all customers, who have placed more than 2 orders.
- (select customereid from orders

- select * from customers where customerid IN (select customerid from orders group by customerid having count(customerid) > 2)
- select * from customers where customerid IN (select customerid from orders group by customerid having count(orderid) > 2)
- We can do this subquery using **inner join** as well but **inner join or joins** in general they are quite **costly operations**. That's why using **subquery** is better.

- Yes.
- It is not always that **subqueries** are **always better** but in the above example, **yes** subquery was better there than **join**.
- Subquery can be better sometimes.
- **Query optimization**.

- **IN** operator can only has **Equal to** comparision.
- If we have to **lesser than, greater than** then **IN** operator will not work.
- We have to use **Any/All** operator.

Any Operator.

- Operators used with **ANY** keyword -> =, <>, !=, >=, >, <, <=.
- Any means **anyone**.
- Any -> **OR** operation.

- Greater than **any** of those values in the **brackets**.
- **= any** will work **same** as **IN** keyword.
- If condition is **True** for one value also then result is **True**.

- Find the productname of all those products which have their orders quantity larger than 50.
- select productname from products where productid = any (select productid from orderdetails where quantity > 50)

- Find the productname of all those products which have their productIDs less than any of the product having orders quantity equal to 1
- select productname from products where productid < any (select productid from orderdetails where quantity = 1)

- Another way of writing the query.
- select productname from products where productid < (select max(productid) from orderdetails where quantity = 1)
- We will directly get **one value** which is the **max(productid)** and we don't have to use the **any** keyword here.

- If it was **> any**, then we ould have used the **min()**, **min(productid)**.

All subquery

- Any -> Anyone one value.
- All -> Comparing with everyone
- All -> **AND** operation.
- Operators used with **All** keyword -> =, <>, !=, >=, >, <, <=.

- Find the productname of all those products which have their productsids less than all of the product have orders quantiy equal to 1.
- select productname from products where productid < all (select productid from orderdetails where quantity = 1)

- **! All** -> Behaves like **Not In**.
- Finds all employees whose salaries are greater than the salary of all the employees in the sales department with departmentID is 2.
- select * from employee where salary > All (select salary from employee where departmentid = 2)

- select * from employee where salary < any (select salary from employee where departmentid = 2)
- select * from employee where salary < ANY (select salary from employee where department=2)

Exists subquery

- Checking if the **result set** returned by the **inner query** is present or not.
- If **Yes**, then we will get the result from the outer query.

- select * from customers where customerID = 1 -> All details of customers where id is '1' -> Returns 1 tuple/row.

- We are checking if **result** coming from **inner query** or not.
- **Existance** of output there in the **inner query**.

- Yes.

- select * from orders where exists (select NULL)
- (select NULL) -> It selects **NULL** as the **only one value**.

- In the **result set**, we are getting **NULL**.

Co-related subquery

- For each row of outer query, either inner query runs again and again.

- Yes, we are taking the **courseid** as well.
- select Rno, name from student S where exist (select * from enrolled E where S.ro =

E.rno);

- With **corelated subqueries** we are doing a type of **inner join** in the above question.

- write a query to select all such customers record which have atleast one order placed.
- select * from customers C where exists (select * from order O where C.customerID = O.customerID)

- select distinct C.* from customers C LEFT JOIN orders O ON C.customerID = O.customerID;
- select distinct C.* from customers C, order O where O.customerID = C.customerID;

- (customers.*) -> Prints the columns only from the **customer's table**.

- select distinct C.* from customers C, orders O where O.customerID = C.customerID;

- select distinct C.* from customers C LEFT JOIN orders O ON C.customerID = O.customerID;

Doubt clearing session (12) [3rd July 2023]

1. select customername from customers where country = (select country from customers where customername = 'Around the Horn')
2. select postalcode, count (*) Counter from customers group by postalcode.
3. select country, max(Counter) from (select country, count(*) Counter from customers group by country);
4. select country from customers group by country having count(customerid) = 1

- Subquery's result set became a **table** for the **outer query**.

- Question

- Tables.

1. select * from product where price = (select price from products where productname = 'Aniseed Syrup')
2. select suppliername from suppliers where supplierid IN (select supplierid from products where price > 20)
3. select * fom products where productname != 'Tofu' and categoryid = (select categoryid from products where productname = 'Tofu')

Set Operators

- Set operators are used to combine results from two or more select statements
- SQL rule -> Both the **result sets** from the **select statements**, **no. of columns** should be **equal**.
- **Set operators** doesn't change the **columns**.
- Same no. of columns as in left or right select statement.
- **Set operators** is happening on the **data**.

- In **union**, duplicates are **eliminated**.

- We have done **union** of the **rows**.
- (select city from customers) UNION (select country from suppliers)

- select 'customer', country from customers
- select 'customer' as type, country from customers
- (select 'customer' as type, city from customers) UNION (select 'supplier', city from suppliers).

- UNION -> Eliminates duplicates
- Union All -> Duplicates are not eliminated. Shows all **tuples**.

- Intersection -> Common
- Intersect -> Gives common tuples.
- Minus or Except -> It gives **set difference**.

- select * from T1 except select * from T2 -> We will get those rows from T1 which are not in T2.
- select * from T2 except select * from T1 -> We will get those rows from T2 which are not in T1.

- Find out those cities where we don't have customers but only suppliers.
- select city from suppliers EXCEPT select city from customers.

- No.

- Create Table

- From another table.

- Constraints

- No value i. e NULL, it is fine
- If it is **value** then it should be **unique**.

- Primary Key

- Yes.

- Foreign Key

Doubts

- Wrong question.

- Yes, with **IN** keyword.

- Correct.

Normalization part-II (13) [4th July 2023]

- Completed.

- Relational Model

- Relation -> Table.

- Attributes

- Tuple

- Relation example

- DB Schema

- DB Instance

- Domain -> Datatype.

- Domain Constraint

- Degree or Arity

- Cardinality

- Functional Dependency

Functional Dependency

- If we have **one value of 'A'** then give **one value of 'B'** and not **two values**.

- For **a1** value of **A** column we are getting **two different values** of **B** column which are **b1 and b3**. So **a1** is giving **not unique** values.
- So, **A -> B** does not **hold**.

- For **a1** value of **A** column we are getting **unique value** of **B** column which is **b1**. So **a1** is giving **unique** value.
- For **a2** value of **A** column we are getting **unique value** of **B** column which is **b2**. So **a2** is giving **unique** value.
- So, **A -> B** does **hold**.

- Yes.

- A -> B -> doesn't Hold
- B -> C -> Holds
- C -> A -> doesn't Hold
- C -> B -> doesn't Hold
- A -> C -> doesn't Hold
- AB -> C -> Holds
- Ac -> B -> Holds

Closure of n Attribute

- What all attributes we can derive from given attribute.
- Closure of attribute **A**(A+) -> {A}
- No dependency is given which can be derived from **A**.
- Closure of attribute **B**(B+) -> {B, C}
- Closure of attribute **C**(C+) -> {C}

- A+ -> {A, B}
- B+ -> {B}
- C+ -> {C, D}
- D+ -> {D}.

- A+ -> {A, B}
- B+ -> {B}
- C+ -> {C, D, A, B}
- D+ -> {D, A, B}.

- Trivial Functional Dependency

- Rules.

- Reflexivity Rule
- If **A to B** holds then **AC to BC** will also hold.

- Augmentation Rule.
- If **E -> F** holds then **ABE to ABF** alos holds.

- Transitivity Rule

- Addition Rule.

Closure of a set of functional dependencies

- In **F+**, we will use **transitivity rule** and we will get **A -> B**.

- We have to **find** more **dependencies**.
- Pseudo transitivity Rules.

- Question, Find **FD+**.
- FD+ -> {A -> B, B -> C, AB -> D, A -> C, A -> D}

- Question
- Option **B**.
- [**IMPORTANT**]
- Do it using **closures** better.

- Question
- FD+ ={AC -> XY, A -> C, BY -> XY, AY -> XY, A -> XY}
- If **A -> B** holds then **AC -> BC** also holds.
- If **AC -> BC** holds then **A -> B** also holds that is **not True or false**.

- Yes.

Finding keys using FDs

- Keys -> Candidate keys.
- **Minimal super key** is **candidate key**.

- If **ABC** is a **candidate key**, then **ABCD** cannot be a **candidate key**.
- If **ABC** is a **candidate key** then we cannot attach attributes to **ABC** to make **candidate keys**. We will not **check them**.
- **ADB** could be a **candidate key** as it is **different** from **ABC**. They are not **proper**

subsets of eah other.

- Key -> Uniquely identify **one single row**.
- If **A** is a key then we can find **B, C and D**.
- From **A** we can derive **B, C and D**, which is **A -> BCD**. Directly or indirectly.

- In closure of **A**(A+) -> {A, B, C, D}
- If we find all of the attributes in closure of **A**(A+) then only we can say that **A** is a **key**.

- Question
- In closure of **A**(A+), we found all of the **attributes** within it, so **A** is a **candidate key**.

- When we are finding **keys**, we will check on the **right hand side**, what we don't have. Whatever is not there, that will be coming in the **key**. [**TRICK**] [**IMPORTANT**]

- A+ -> A, B, C, D
- B+ -> B, C, D, A
- C+ -> C, D, A, B
- D+ -> D, A, B, C.
- All of them have **all of the attributes** in their **closures**.
- So, **A, B, C and D** are **candidate key**.

- Question
- Find Keys
- B+ -> B, D,
- A+ -> A
- C+ -> C, E
- D+ -> D
- AB+ -> A, B, C, D, E, F
- CD+ -> C, D, E, F.
- Only **one** of them have **all of the attributes** in their **closures**.

- So, **AB ** is the **candidate keys**.

- Question
- Find Keys
- Attributes not in the **right hand side** -> A
- A+ -> A
- AB+ -> A, B, D, C, F, E
- **B** can be replaced in **AB+**, which is **C**
- AC+ -> A, C, B, D, F, E
- **C** can be replaced in **AC+**, which is **D**
- AD+ -> A, D, C, F, B, E
- Only **three** of them have **all of the attributes** in their **closures**.
- So, **AB, AC, AD** is the **candidate keys**.
- We have to check if the values can be derives from other values or not **like above**. [**IMPORTANT**]

ER diagram to relational model conversion (14) [5th July 2023]

- We are doing **union**.

- Question
- AB+ -> A, B, D, F, E, C, G
- GB+ -> G, B, A, D, F, E, C
- AC+ -> A, C, B, D, F, E, G
- GC+
- AE+ -> A, E, C, B, D, F, G
- GE+
- These are the **6** keys.

- AD+ -> A, D, E, G [Not possible, incomplete]

- AB+ -> A, B, C, D, E, G
- AC+ -> A, C, B, D, E, G
- BC+ -> B, C, A, D, E, G
- These **three** are the **keys**.

- E+ -> C
- BC -> B, C, D, A,
- AEG+ -> A, E, G, C, B, D
- DEG+
- BEG+
- So, **three** keys are coming.

- **BCEG** not needed as we can get **C** from **E**
- So, **BEG** will work.

- yEs.

- Minimal Set.

- Exactly.

Normalization

- Normalizing the table **R** and decomposing table **R** into **R1 and **R2** table.
- If we didn't allow **NULL** value in **P-Id, P_Name and P_Price**. They should be **Not**

NULL.
- There is an **anomoly** due to **insertion** into the table.

- Yes.

- Insert anomaly.

- It is called as **insert anomaly**.

- **Update anomaly**.

- Assuming product jeans price has been chnaged to 2500.
- Rencho purchased shirt in 2200 but now my DB shows in 2500.

- Delete product jeans
- Deletion Anomaly
- Customer with id **c3** also deleted.
- To **remove** the above anomalies we use **normalization**.

- GATE syllabus till **BCNF**.
- NFs -> 1NF, 2NF, 3NF, BCNF.

- Till **BCNF** is **needed**.
- Remove multivalued attributes.

1NF

- A relation R is said to be in 1NF if there is no any multivalued attribute in R.

- Phono is **multivalued** attribute
- Eliminate **multivalued** attribute

1. Increase rows.
2. Increase columns.
3. Decompose relation.

1) Increase rows.

- Problems:-

1. **Rno** column is no longer a primary key
2. Too mcuh of redundent information.

2) Increase columns.

- This works perfectly fine.
- Problems:-
- Wastage of space if multiple NULL values in phno 2.
- Wastage may increase if one row has 3 phone umbers.

3) Decompose relation.

- We removed the **Rno** column.
- Keep multivalued attribute along with **primary key** of table in another table.

- This is also a **good solution**.

2NF

- If we have to normalize the DB table in **2NF** form, then they should be **normalized** in **1NF** form.

- Yes.

- No any non- prime or attribute.
- There is **no any **, Non non-rprime attributes,

- Partial Dependency.

- Prime and non-prime attribute.
- if key is a **composite** key and from the **sub-part**. If the **sub-part** derives some **non-prime** attribute.
- This is calle as **Partial Dependency**.

- If **A** is a **composite** row then **no. of poossibilities of partial dependency**..

- Ofcourse.

- Example
- Candidate Key is **AB**.
- Prime attribute -> A, B
- Non-prime attribute -> C, D
- If **table's data** is not given, then we can assume that the **table** is in **1NF** form.

- For the table to be in **2NF** form, there are **two** conditions:-

1. Table should be in **1NF**.
2. No partial dependency.

- **A or B** alone can derive any **non-prime** dependency?

> **Yes**, **B** derives **D**.

- It is a **partial dependency**.

- Given relation **R** is not in **2NF**.
- We have to get it to **2NF**.
- To get the relation in **2NF**, we have to **decompose** the relation.
- To bring **R** in 2NF, decompose **R** into **two** relations such that we remove **partially dependent** attribute from **R** and keep it in another relation.
- We have to keep the **partially dependent** attribute with whom it is **dependent** on.

- Not with the **key**.

- Key in **R1** tabel is **AB**.
- Key in **R2** table is **B**.
- So, there is **no partial dependency**.

- Both **R1 and R2** are in **2NF**.

- Will see later.

- Very important.

- Yes.

- Partial dependency

- Candidate Key -> AB
- Prime attributes -> A, B
- Non-prime attributes -> C, D, E
- We have **two partial dependencies**.
- A -> D and B -> E
- We have to keep **each partial dependency** in separate tables.

- Now, **R, R1 and R2** are in **2NF**, **partial dependencies** are removed.

- Question.
- Option **C and D**.

Questions on relational modeling fd normalization (15) [7th July 2023]

- Normalization -> To Remove **redundancy**.

- **AB** is a **candidate key**.
- **B -> C** is **possible**.
- **C -> B** is **not possible**.
- We have a **partial dependency**, as **AB** is a **candidate key** and **B -> C** is **possible**.
- So, the **partial dependency** is **B -> C**.

- In the **R2** table, why write the **same values** multiple times, write it **once**.

- Reduced or reducing **redundancy** of **partial dependency**.

- Question.

- Steps to get **2NF**, Method(Decompose into 2NF):-

1. Find the **Candidate Key**.
2. Find the **prime and non-prime attributes**.
3. Check if **individual values** from the **Candidate Key** have **dependency** on **non-prime attributes**.
4. If so, then they are **partial dependencies**.

- As we found that **AB -> C** is causing **partial dependency**, so remove **C** from **R** table.
- Keep **C** in a **separate** table with it's **dependency** which is **AB**, which is **R2** table.
- As we found that **D -> C** is causing **partial dependency**, so remove **E** from **R** table.
- Keep **E** in a **separate** table with it's **dependency** which is **D**, which is **R3** table.
- Don't remove the **dependencies**, which are **AB and D** from the **original table(R)**.
- Only remove **C and E** from the **original table(R)**.
- R1(A, B, D) -> No Need of **FDs** as **ABD** together is making a **key**. No sense of **deriving** a fourth attribute.
- R2(A,B, E)
- R3(D -> E)

- For every **partial dependency** we will create a **separate table**.

- Question

- In **2NF**.

3NF

- **No**. Not dependent on the **key**, it is dependent on **A**, whch is not the **key**.
- Transitively dependent
- Key -> Non-prime
- Non-prime -> Non-prime

- Candidate key -> AB
- Prime -> A, B
- Non-prime -> C, D
- Check for 2NF -> No partial dependency -> **2NF satisfied** -> It is in **2NF**..
- AB -> C [Key -> non-prime]
- C -> D [non-prime -> non-prime]
- So, AB -> D [Transitively dependent]
- **D** is **Transitively dependent** on the **key(AB)**.
- So, **3NF** not satisfied.

- **D** is **dependent**, **D** is the **issue/problem**.
- If **D** is the **issue/problem**, then **remove 'D'**.
- Keep **D** in a **separate table** with **C**.
- We will do **decomposition** here.

- Decompose **R** into **2 relations**.
- Whoever is **Transitively dependent**, **D** here, we will remove that from the **original table**. [**IMPORTANT**]

- Yes rule of **3NF**.

- To whom **D** is **directly dependent** on, that should go with **D**.

- If **D** is transitively dependent on **key** then, **remove 'D'** from the **original table** and keep it in **another table** along with the **attribute** with which **D** is **directly dependent**.
- Don't remove **C** from the **original table**.

- Yes.

- There is **transitive dependency** as **A** is the **key**.
- A -> C
- C -> D
- A -> D
- **D** is **transitive dependency** on **A**.
- So **3NF** not **satisfied**.
- We have to **decompose** the **R3** table into more tables to satisfy **3NF**.
- If the **candidate key** is a **single column key** then there is **no partial dependency**.

- Option **C**.

- np -> Non-prime to Non-prime

- So, there will be **transitive dependency**.

- [**IMPORTANT**]
- 3NF.

- Then also **3NF** is adequate.

BCNF

- Candidate Keys -> AB and AD.
- No any partial or transitive dependency. Hence already in **3NF**.

- In the **FDs** or **functional dependency**, the **LHS** should be a **key** for it to be in **BCNF**. [**IMPORTANT**]
- Not in **BCNF**.
- The solution is to do **decomposition**.
- Remove **B** from the original table.
- Put **B** in another table with **D**.
- Do not remove **D** from original table.
- Remove **B** from the original table and keep it in another table with **D**.

- There is so **data loss** as earlier the **FD** was **AB -> CD** in the original(R) table but in **R1** table it is **A -> CD**.
- This **decomposition** is called as **lossy decomposition**.

- The **original FD**, **AB -> CD**, is not **preserved**. It is now **A -> CD**.

- **BCNF** can create problems. There is **no any solution** for this.
- When we do **decomposition** till **3NF**, everything is **fine**. **No data loss**, no **FDs** are lost as well.
- When we go to **BCNF**, there is a **possibility/chance** that **FD** is **not preserved** or the **decomposition** that happened, causes **data loss**.

- Upto 3NF decomposition -> Definitely **lossless and dependency preserving** -> **Compulsory**.
- BCNF decomposition -> Not necessarily -> **lossless and dependency preserving** -> **possibility/chance**.
- BCNF **always provides** lossless and dependency preserving decomposition?

> **False**.

- BCNF **may provide** lossless and dependency preserving decomposition?

> **True**.

- Yes.

- Solved in my copy.

- Question
- Option **A**.
- Partial Dependency are there -> A -> C and B -> D.
- So it is not in **2NF**.

- Question.

Doubt clearing session(16) [7th July 2023]

- Relation is already in **3NF**.
- Also in **BCNF** because for each **FD**, the **LHS** is a **key**.

- Yes.

- Question.
- Answer -> **a, b, c**.

- BCNF
- [**IMPORTANT**]
- A -> B
- No partial or transative dependencies
- LHS is a **key**
- So **BCNF**.
- Same for **B -> A, AB -> AB**.

- We said that if more than **2 attributes** are given then it is not possible to find **relation** without **FDs**.
- If **2 attributes** are there then it is possible without **FDs**.

- Question.
- Option **D**.

- For **partial dependency**, all of the **above problems** happen.
- To remove them we apply **normalization**.

- **Inconsistency** comes because of **redundancy**.

- Yes.

Lossy VS Lossless Decomposition.

- We will do **inner join** and the **common column** is **sname**.

- select * from student_details S inner join s_department d on S.sname = d.sname;
- We will get **4 columns** and **4 rows**.

- If we want the **department** of **roll 12 vishvadeep**, then the **query engine** will be **confused** as we have **two roll 12 vishvadeep**, one of them has **CSE** as their **department** and the other has **AI** as their **department**.
- We didn't get the **correct** information, we wanted.
- THis is **lossy decomposition**.

- It will be **lossy decomposition** when we do **inner join** on the **decomposition tables** and after doing **inner join** the **result set** that we will get. If the **result set** is **not equal** to the **original table** then it is **lossy decomposition**.
- If the **result set** is **same to same** as the **original table** then it is **lossles decomposition**.

- We don't know which is the **original data**.

- Lossy Join Decomposition
- The join is **inner join**.

- Lossless Join Decomposition

- select * from student_details S, S_department d where S.rno = d.rno

- Inner Join
- R becomes a **subset** of the **result set** -> Lossy decomposition.

- Question.
- **Y** is the **common table**.
- We will get **lossy decomposition** because **5** is there **two** times in **Y** column. We will get more columns than there was in the **original taable**.

- Good **observation**.

- Question.

- If the **common column(Rno)** is the **key** in **both the tables** then it should be**lossles**.

- If the **common column(Sname)** is **not a key** in **both the tables** then it will be**lossy**.

- We are doing **intersection** between the **attributes** of **R1 and R2 table**.
- If the **common attribute(D)** we got is a **candidate key** in anyone of **R1 or R2** -> **Lossless decomposition**.
- **D** is **key** in **R2** hence it is **Lossless decomposition**.

- Question

- **R1 intersection R2**, between **R1(A, B, C) and R2(A, D, E)**. So, we have **A** as the **common attribute**.
- Is **A** a key in **R1 or R2**?
- As it is **yes**, so it is **lossles**.

- Question.

- Lossy.

- Question.
- Common between **R1 and R2** is **B, C**. **Two Attributes**.
- If we have **more than one common attribute** then we have to write condition on all of the attributes.
- We have to write **join condition** on all of the **attributes**.
- We have to do **AND**.
- R1's FDs, A -> BC, BC -> A
- R2's FDs, B -> CD
- R1's key -> A, BC
- R2's Key -> B

- As **BC** is a **key** in **R1**, so it is **lossless**.

- lossless
- [**IMPORTANT**]

- If **B** was alone then also **lossles**.

- Question.
- If we do **inner join** of **three tables** then we have to write **two join conditions**.

- Combine **two-two** tables.

- Yes.

- Lossless.

- Question.
- There is **no common attribute** between **R1 and R2**.
- So, **lossy**.

- Yes.

- **y** is the **partial dependency**, we **remove** it from the **original table**.
- **y** was **dependent** on **x**.
- **x** also goes to the **new table** with **y** and **x** becomes the **key** of that **table**.
- **x** is also present in the **original table** also.
- So, **x** is common in the **new as well as the original table** and **x** is **key** in **one of the tables**.
- That's why **2NF and 3NF** always gives **lossless decomposition**.

> First we will check **common attributes**, then we will check if among them we have any **keys** or not.

- **C** was the key in **R1**.
- **AB** was the key in **R2**.
- Common was **A**.
- **A** was not a key in **R1 and R2**.
- So, **Lossy** then.
- Doesn't matter with **prime attributes**.

Dependency Preserving Decomposition

- So, **Dependency Preserving Decomposition** example.
  *From the **union** of the **FDs** of **R1 and R2**, we couldn't get **D -> A**. So it is a **not Dependency Preserving Decomposition** example.

- Not
- D -> C, lost

- Question.
- Option **C**.

- Question
- Option **2**.

Doubts

relational-algebra-part-i (17) [11 July 2023]

ER Diagram to relational model

- 1 Entity Set -> 1 Table.

- Using **both Eid and Cid** together we can make a **composite key** for the **Teachers table**.

- Teaches.Eid -> Foreign Key referring to **Educators.Eid**
- Teaches.Cid -> Foreign Key referring to **Courses.Cid**
- **No partial dependency** present in **Educators, Courses and Teachers** tables.
- So, all the **three** tables will be in **2NF**.
- **No transitive dependency** present in **Educators, Courses and Teachers** tables.
- So, all the **three** tables will be in **3NF**.
- All **3 relations** are in **BCNF**.
- Keys are in **LHS**.

- Option 2 -> Keep relationship information along with an entity set
- Relationship towards Educators -> 2 tables.

- **Primary Key** in **Educators** table is **Eid Cid**, combination of both.
- We will get **partial dependencies** because **Eid** alone can derive **non-prime** attributes **Ename and ESalary**.
- If we do **decomposition** here to remove the **partial dependencies**, we will get a table with **Eid, Ename and ESalary** and another table with **Eid, Cid and Since** which is **same** as the earlier(option **1**) one we created.

- Problems:-

1. Partial Dependency
2. Cid -> Cname, Cfee

- The above one is a **many to many** relationship , so the ***option '1'** is the **correct**

answer.

- We have to keep **one stuent's information** only **once**.

- **Sid** is the **primary key** in **students** table.
- **Eid** is the **primary key** in **Educators** table.

- **Eid** is not **primary key** and **too much redundancy**.

- So we will go towards **many**, which is **students**..

- There is **no point** of creating **three** tables when we can get the work done in **two** tables.

- FDs.

- **Sid** is the **primary key** in **student and Guides** tables.
- So, we can keep them **together** and make **2 tables** instead of **3 tables**.

- **Sid** is **primary key**. So it is better to keep it in same table.
- **One to Many** and **Many to one** is **same thing**.

One to One Relationship

- It is will be in **2 tables** only.
- **3 tables** scenario comes when there is **Many** relationship.

- Whichever side we will have **total participation**, we will keep the **relationship details** on that side only.

- **Option 2** is better because license has total participation in **relationship**.

- Keep **two tables** and set **guide_eid** as **Not NULL**.
- So **student** table will be **total participation**.

- There cannot be any **department** without any managers.

- Many TO Many -> 3 tables.

- How to control participation in many to many?

HW.

- If there is **overlapping** then it makes sense to go for **3 tables**.

- Overlapping
- Dis-joint.

- Thre can be **multiple dependents**.
- The key can be a **summation** of all the attributes, **Eid Dname and Dbod** which is the &&key**.
- **Eid** cannot be the **primary key** in **departments table**.

Relational Algebra

- SQL -> Non-procedural query language
- Relational Algebra -> Procedural query language

- We will get all of the **rows and the columns**.

- Which **columns** we want?

> We want all of the **columns**.

- We have to use **project** here.
- Without **project**, data will not be **fetched**.

- We will get all of the **rows and the columns**.

Select

- Example
- sigma(rating > 8) (sailors)
- Sigma -> Filtering rows.

- Find all sailors who haverating greater than 8 but age less than 25.
- '^' -> AND symbol
- 'V' -> OR symbol

- Comparison operators.

- We are **filtering** out the rows based on **conditions**.

Doubt

- To get **BCNF** or satisfy **BCNF**.

relational-algebra-part-II (18) [14th July 2023]

- **select** filters rows based on the condition.

- **Same** as **where clause** in **SQL**.

- **select** is **commutative**.
- ^ -> AND.

- Find all products with prices between **15 and 30**.
- (Sigma) price >= 15 ^ price <= 30 (Products)

Project

- It is like **select clause** of SQL.
- It **eliminates** duplicate from tuples/rows.
- It takes only **distinct** values from tuples/rows.
- We do **project** at the **last/end**.

- pie cname (customers)
- We will get **distinct names**.

- The **project** operation results in a set of a distinct tuple as the project operation removes duplicate tuples.

- pie Ename, Salary (Employee)

- pie Salary (Employee)
- We are getting **distinct salary** which are **3** only.

- pie A, B (R)
- We got **combination** of the **distinct values**.

- **Sequence** of **execution**.
- pie Ename (sigma Dno = 2 ^ salary > 17000 (Employee))

- Yes, fetching the records.

- Yes.
- Fetch name and dno of all male employees who are having salary less than 30000.
- pie ename, dno (sigma salary < 30000 ^ sex = 'M' (Employee))

- Yes.

Set Operations

1. union
2. Intersection
3. Set-difference

- If we want to use a **set-operator** then we have to keep **two** things in mind:-

1. No. of columns must be **same**.
2. Data Types in the corresponding columns must be **same**.

- No. of columns and data types of corresponding columns must be **same**.
- If the **condition** are met then **any set operation** will work.
- Operations is performed on **tuples**.

- The **whole tuple** needs to be **duplicate** otherwise we will write the **tuple** in the **union**.

- We got the **distinct name** from **E1 union E2**.

- We got the **distinct age** from **E1 union E2**.

- Relational Algebra(RA) doesn't remove them automatically but **project** removes them automatically.
- There is a **difference**.

- Question.
- (pie sname (sigma dob = '27-10-1988' (Students))) U (pie ename (sigma salary > 15000 (employees)))

Intersection

- Common Only.

set-difference

- All those tuples of **E1** which are not present in **E2**.

- E1 - E2.

- E2 - E1.

- Union and intersection are commutative and associative.

- **Set-difference** is not applicable.

- Question.

- **Yes**, they are **equivalent**.

- **Project** is **distributive** over **union**.

- **Project** is **not distributive** over **intersection**.

- Both are **not equivalent**.

- Question.
- (pie eno, ename, dno (sigma gender = 'M' ^ salary > 20000 (Employee))) U (pie eno, ename, dno (sigma gender = 'F' ^ salary > 25000 (Employee)))

Cartesian Product(Cross product)

- Cartesian Product(Cross product)

- Cartesian Product(Cross product) -> pie A = D (R X S)

- Question.

Joins.

- When we join **two tables** and we filtered out the result based on the condition.
- This is called as **condition or conditional join**.

- In **conditional join**, the join condition would be **>=, >, <=, <**.
- There will be **no equal too(=)** condition.

- In **Equi join**, we are going to use **=** operator.

- We will get **two** rows.

- In **natural join**, we don't have to write any conditions.

- We will write it directly.
- There will be **one common column**, the **column name** would be **same** as well..

- The **Rollno** is the **common column** here.
- The condition will be automatically applied on them(common column).
- The **common column** name should be **same**.
- If the **common column** name is not the **same** then we have to **explicitely** write it. Like below.

- **Implicitely equi join** is used for all **common columns**.

- There are **two** common columns, **A and B**.

- Attribute/column name which is the **common column** does not have the **same name**.
- Then, we have to **explicitely** write the **condition**.
- Otherwise, we can write like **bnatural join**, no conditions needed.

- Question.
- pie cname(sigma shipperid = 3 (Customers natural join(damru) orders))

- The **common column** which is **cid** has the **same** name.

relational-algebra-part-iii (19) [16th July 2023]

- Conditional Join -> <, >
- Equi Join -> =

- Natural Join -> No condition. [The tables have the common column with the same name on both tables]
- In **natural join**, automatically the **common column** join condition is there. We don't have to write it.
- In **equi join**, we have to write the **common column** join condition. The **name** of the **common column** is **different**.

- Question.
- pie dname (sigma drivers.did = drives.did ^ cars.cid = drives.cid ^ ccolor = 'Blue' (Cars (Cross join) Drives (Cross join) Drivers))

- Question.
- pie dname (sigma drivers.did = drives.did ^ cars.cid = drives.cid ^ (ccolor = 'Blue' V ccolor = 'Black') (Drivers X Cars X Drives))

- Question.
- pie dname (sigma drivers.did = drives.did ^ cars.cid = drives.cid ^ ccolor = 'Blue' ^ ccolor = 'Black' (Drivers X Cars X Drives))

- (pie dname (sigma drivers.did = drives.did ^ cars.cid = drives.cid ^ ccolor = 'Blue' (Drivers X Cars X Drives))) (Intersection(U)) pie dname (sigma drivers.did = drives.did ^ cars.cid = drives.cid ^ ccolor = 'Black' (Drivers X Cars X Drives))

- Question.
- (pie drivers.did, dname (sigma drivers.did = drives.did ^ cars.cid = drives.cid ^ (ccolor = 'Green' V ccolor = 'Red') ^ (Drivers X Cars X Drives)) - (pie drivers.did, dname (sigma drivers.did = drives.did ^ cars.cid = drives.cid ^ ccolor = 'Green' (Drivers X Cars X Drives))) intersection pie drivers.did, dname (sigma drivers.did = drives.did ^ cars.cid = drives.cid ^ (ccolor = 'red')

- Left Outer Join -> We get all values(rows) of the **left table**.

- Right Outer Join -> We get all values(rows) of the **right table**.

- Full Outer Join -> Extra records of both the tables will come that do not participate in natural join.

Rename Operator

- Yes.

- When we want to **rename** the names of the columns as well as the **table** name.

- When we want to **rename** the names of the columns only, we are aliasing the table as well but we are not giving a name to the table.
- New instance of **student table** is created. We haven't given a name to the table, we have just renamed the column names.

- When we want to **rename** the names of a **specific column**.

- Sequential renaming.
- If we don't want to rename all of the **columns** then we have to put **arrows(->)** for the column renaming.
- Otherwise it will not work.

- Question.
- It is conditional join.
- (name of all female students) - (name of all female students who have scored equal or lesser than any male student) -> name of all female students who have scored more marks than male students.

- Option **D**.

- It is conditional join.

- Question.

- We got the **female sid** who have not enrolled into a **particular course**.

- The **cid** of **c1, c3 and c4** are the courses where some students have enrolled.

- we will get **those courses** where not all of the female students have enrolled but some of them have enrolled.

- Option **B**.
- [**IMPORTANT**]

Doubt

doubt-clearing-session (20) [17th July 2023]

- Questions.

1. sigma country = 'Germany' V country = 'UK' (customers)

- As we want all of the customers('*' in SQL), so we don't have to write **project(pie)** here.

2. pie cname, address, postalcode, country (sigma contactname = 'Yang Wang' (customers))
3. sigma customerid <=19 (customers)
4. sigma country != 'Germany' ^ country != 'UK' ^ country != 'USA' (customers)

- Set-difference.

- Question.

1. sigma supplierid = 1 V supplierid = 2 V supplierid = 3 (products)

2. pie productname (sigma price >= 5 ^ price <= 25 (products))

- As range is mentioned, so, it will be **inclusive** only.

3. pie supplierID (sigma category = 2 (products)).

- If supplier details needed, then use suppliers relations. We have to do **join** then.

- We are writing **pie suppliername, address, city(project)** because we only want the **supplier details**. If we don't write **pie(project)**, then we will get the details of **products** as well which we don't want.

4. sigma supplierid = 2 ^ price > 30 (products)
5. sigma price > 50 ^ supplierid != 6 (products)
6. sigma price < 30 ^ supplierid != 2 ^ supplierid != 6 (products)

Division Operator

- R1 / R2.
- R1 -> Numerator
- R2 -> Denominator.
- **R2** should be **smaller** for good/smooth division.

- It will work.

- It will work.

- It will not work.
- R1 / R2
- The result is the attributes that are not in **R2** but are in **R1**.

- Single value of **B column** associated with **A1 and A4** from the **R2** table's **A column**.

- **Division** operator is **for all**.
- After the **division** we will get **pid** but which **pid** we will get. The **pid** that are associated with every **prodid**.

- We want that **pid** which is infront of all the **prodid**.

- Question.

- Division is possible.
- Result set -> StudentName.
- Those **student names** will come that are associated with **CA,CB and CC**.

- Yes.
- CarID and Date should be **same** value with the **Cid**.

- [**IMPORTANT**]

- Women enrolled in a course question.

Doubt

- The condition would be **True** for one row.

- Will work.

trc-drc (21) [17th July 2023]

- RA -> Relational Calculus.
- Predicate Logic(P)

- Yes.
- Domain -> Concentration on Attributes.

- Tuple -> Focused on tuples -> Bring those tuples by writing conditions.
- Domain -> Focused on attributes -> Bring those attribute values which satisfy the conditions.

- Relational Algebra is procedural.

- tuple(t)
- t -> On the **left side** of the **bar** is the **tuples** we want to **select**.
- P(t) -> On the **right side** of the **bar** is the **conditions, from where** we will ge the tuples. Conditions that are satisfied by the tuples. [Predicate]
- t -> tuple variable.
- {t | p(t)} -> fetch all tuples(t) which are satisfying the predicate, p(t).

- {t | t belongs to student} -> fetch all tuples(t) which belong to the table, student.

- Yes.

- To get a **specific attribute/row** from a tuple.

- We just want the **name** of the students from the student table.
- Attribute -> name.

- Full row is **tuple(t)** only.

- t[name] OR t.name
- Both are **same** and correct.

- Separating multiple rows with commas(,).

- This is also **correct**.

- {t | t belongs to Shopkeeper} -> Fetch all of the tuples from the shopkeeper table.
- {t | t belongs to Shopkeeper ^ t.rating > 8} -> Fetch the tuples where the **rating > 8**.
- ^ -> AND
- It is like **where** clause in SQL.

- Find firstname of all shopkeepers who are having rating between 7 to 10
- {t.firstname | t belongs to Shopkeeper ^ t.rating >= 7 ^ t.rating <= 10}

Tuple Relational Calculus

- TRC eliminates duplicate.

- Select fname, Lname from students where gender= 'Male' and marks < 20.
- {t.fname, t.lname | t belongs to students ^ t.gender = 'Male' ^ t.makrs < 20}

- Find all such students whoo's marks are greater than 40.
- {t | t belongs to students ^ t.marks > 40}

- Empty set because there is no any attribute age in relation students.

Quantifiers

- Existential -> There exists a tuple(t), which belongs to relation abd satisfies predicate(p).
- Universal -> All tuples(t) satisfy the predicate, P(t) then only we can bring the tuples. Otherwise not.

- Question.

- Yes.

- For **two table**.
- [**IMPORTANT**]
- {t | t belongs to students ^ t.Gender = 'Female' ^ there exists s {s belongs to Department ^ s.Dname = 'CS' ^ S.Dno = t.Dno}}

- No.

- Yes.

- Question.
- {t.Dname | t belongs to Department ^ there does not exist(~) S {S belongs to Students ^ S.Gender = 'Male' ^ S.Dno = t.Dno}}

- Symbols for AND, OR and NOT.

- Question.
- 3 tables.
- {t.cmodel | t belongs to Cars ^ there exists D {D belongs to Drives ^ D.cid = t.cid ^ there exists E {E belongs to Drivers ^ E.dname = 'Mitchell' ^ E.did = D.did}}}

- Whichever is asked first in the question that table comes first.

- In general, when we have to do **joins**, like above, then **there exists** is fine. It will work.

- [**IMPORTANT**]

- It is **non- procedural**. It is like a sequence of expressions.

- Question.
- {t.sname | t belongs to Student ^ there exists E {E belongs to Enrolled ^ E.rno = t.rno ^ E.duration= 2023 ^ there exists C {C belongs to Courses ^ C.cname = 'DBMS' ^ C.cno = E.cno}}}

Tuple Variable

- Bound -> Where we added **quantifiers**.
- Free -> No **quantifiers**.

- Here, we have to write the **commonn table** first.

- There Exists -> The combination between the rows should match.
- For all -> There should be combination with all the rows on the right hand side, with the single row on the left side.

Domain Relational Calculus

- {<r,f,l,m> | <r,f,l,m> belongs to Students}
- **First part/left side** of the **|** bar is what we want.
- **Second part/Right side** of the **|** bar is the **conditions**.

- Similar to select * from students.

- We just want the **first name** of all students.

- We want the **first name** of those students who have **marks(m) greater than 90**

- The table name we have to write always. All of the columns names we wil also write.

- Yes.
- It shows the particular attribute.

- Question.
- Name of all such drivers who are having rating > 8 and dob = '27-10-1988'
- {<dn> | <d, dn, r, dob> belongs to Drivers ^ <r> > 8 ^ <dob> = '27-10-1988'}

- We can write the **variable names** same as the **column names**. Just remember to **maintain** the sequence of the columns in the table. They should come in **sequence**.

- Question.

- [**IMPORTANT**]

- If we have one/single value with **equal too condition** then we can **replace** the **domain variable** with that **number/value**.

- We can do this in **equal too operator** only. Doesn't work with **>, <** operators.

- Question.
- {<rno, name> | <rno, name, lname, marks> belongs to students ^ <lname> = 'Kumar' ^ <marks> > 50}

- Both are **same** and they work.

- Question.
- {<cid> | <accno, cid, amt> belongs to Accounts ^ <amt> > 50000}

transaction-concurrency-control-part-i(22) [25th July 2023]

- Many to many participation contraint

- We cannot control the **participation contraint** in **many to many**.
- We can control the **participation contraint** in **one to many, many to one and one to one**.

Transaction

- DB operations -> Read, Write, Update
- Sequence of **DB access operations** is called as **transactions**.

- Transaction -> Sequence of **DB access operations**.

States of transactions

- Commit -> To save the state of the transactions.
- Rollback -> Revert the database state to that state where transaction has not even started. It will take us back to the **last commit**.

- As there was **rollback** at the end, so we **reverted back** to the **last commited value of 'x'** which is **5** that's why we got **x = 5**.

- States of transactions.

- If a transaction has **failed** then it has to be **rollback**. It is not the job of the users to **rollback** the transaction. It is the job of **DBMS**.
- The **rollback** of the transaction is done by **DBMS**.

- If the **commit** was successful or the **transactions** were committed.

Schedule

- Collection of multiple concurrent transactions.

Concurrency

- Parallel ->

- Parallel work.
- Concurrency ->

- Concurrency work.

- Parallel and concurrent run.

ACID property

- ACID -> Atomicity Consistency Isolution Durability

- Atomicity -> All or none. Either it will run fully or it will not run.
- Consistency -> Expected output/result from DB.
- Isolation -> The concurrent transactions should run in such a way that they provide final result same as they were running like one after another.
- Durability -> We should be able to see the results for a long period of time.

- Why Concurrency

- Problems with concurrency

Dirty Read or Temporary Update Problem

- Reading someone's dirty written value.

Phantom Read Problem

- Wanted to read again and the value was gone.

Unrepeatable Read Problem

- When **reading** two times, we got the **old value** once and we got the **new value** the next time.

Lost Update Problem

- **X= 7** is **lost**.

transaction-concurrency-control-part-ii (23) [27th July 2023]

- Some transaction **directly** writes and does nothing else, nor reading or updating anything then it is called as **blind write**.

Good VS Bad schedule

- Good schedule -> Final result as expected.
- Bad schedule -> Final result not as expected.

Serial VS Non-Serial Schedule

- Serial -> Sequential System.
- Serial -> First a whole transaction runs then another whole transaction runs. We are not talking about which transaction will run first. Anyone transaction's whole statements are completely run then another transaction's whole statements are completely run.

- Non-Serial Schedule -> Also called as **concurrent schedule**.

- They run in **interleaving** manner. There is no manner/way in which the scheduler runs, anyone of the statements can be run at anytime. We got no idea how the statements are run.

Serializable Schedule

- It is a schedule which is **concurrent** but it's final result comes in the way of **serial schedule**.

- A concurrent schedule which can provide final output as a serial schedule.

- Serial Schedule, **rune T1 then T2**.
- We are getting the **same output** as a **serial schedule**. So, it is working like a **serial schedule**.
- Serial Schedule, **rune T2 then T1**.

- So, it is a **serializable schedule**.
- X= 5/10/20
- Y= 5/15
- Cocurrent schedule -> X=20, Y=15.
- X= 5/10/20
- Y= 5/15
- Serial Schedule, T1 then T2 -> X=20, Y=15.
- X= 5/10/15
- Y= 5/15
- Serial Schedule, T2 then T1 -> X=15, Y=15.

- Question.

- T1 then T2

- T2 then T1

- It is **serializable schedule**.
- As the **concurrent schedule's output** is matching with **one of the serial schedule** outputs which is **T1 then T2**. That's why we can say that the **above** is a **serializable schedule**.
- If it(concurrent schedule) matches with **anyone of the serial schedule output** then it is a **serializable schedule**.

- Serializability -> Method to prove that a **concurrent schedule** is **serializable**.
- **Types**:-

1. Conflict Serializability
2. View Serializability

Conflict Serializability

- Find conflicts and then based on it prove that a given schedule is conflict serializable or not.

Conflict

- Two(2) database access statements are conflict statements if and only if all of the following conditions are satisfied.

1. Both statements should be in **2 different transactions**.
2. Both statements should access same data item.
3. One of them should be a **write operation**.

- There will be **no conflict** between **two read operations**, which is **Read to Read** operation.
- **Conflict** will be between **Read to Write**, **Write to Read** and **Write to Write**. [**IMPORTANT**]
- When **checking for conflicts**, we should be checking the **below/bottom** statements than the **above** statements compared to the **current statement**. [**IMPORTANT**]

- 2 conflicts we got.

- 2 conflicts we got.
- R(X) -> W(X)
- R(Z) -> W(Z)

Conflict Equivalent Schedules

- **Different schedules** where we have **same type of conflicts** on the same data items, same sequence.
- **Two schedules** having **same conflicts** in the same order.

- **S1 and S2** are **Conflict Equivalent Schedules**.

- Question.

- Not **Conflict Equivalent schedules**.

Conflict Serializability

- Given schedule 'S' is conflict serializable if it is conflict equivalent to S' .
- Where S' is a **serial schedule**.

- Conflicts.
- Precedence Graph -> Directed Graph.
- Vertices/nodes -> Transactions
- Edges -> Conflicts.

- After drawing the **graph**, check if it has **cycles** or not.
- Cycle -> Started from an edge and followed vertices and was able to return to the starting edge. Then it is a **cycle**.
- If **cycle** present in graph -> Not cconlict serializable

- Question.

- As it is **conflict serializable**, we have to tell the **sequence** also.
- Sequence -> T1 -> T3 -> T2.

- Question.

- R/W -> Read/ Write
- Number -> Transaction Number
- A, B, C -> Data Item

- Wherever **commit** comes after that we don't have to check.
- No checking after commit for **conflict**.

- No Cycle, so conflict serializable.

- There will be **no conflict** between **two read operations**, which is **Read to Read** operation.
- **Conflict** will be between **Read to Write**, **Write to Read** and **Write to Write**.

doubt-clearing-session(24) [28th July 2023]

- Question

- Answers.

- Doubt.

- Doubt.

View Serializability

- There are **many good schedules** which we couldn't identify with the help of **conflict serializability**.

View Equivalence

1. Who is reading first from database

T1 is reading the **X** value from **database**.

- **T2** is reading the **written value of 'X' by T1**. It is not reading the original value of **X** from database.
- Who is reading first from database?

T1.

- Who is reading first from database? -> Who is directly reading the value of **X** from the database before any other **transaction** writes.

- So according to the **first rule**, we will say that **T1 and T2** transactions have read first the value of **X** from the database. It is because **T2** has not used **W(X)** which means that the value of **X** has been **updated** from the original value.

2. Who is reading from other

- In this **T2** transaction is the one who is reading from **T1**.
- **T2** reads **X** from **T1**.

- T1 reads **X** first
- T2 reads **X** from **T1**.

- Who is reading from from database?

> T1 and T2.

- Who is reading from other?

> **T3** is reading from **T1** as **T1** has last written to **X** before **T3** tried to read **X**.

3. Who is writing last

- **T1** has written last to **X**.

- **View Equivalent** -> When both schedules S1, S2 are following same all 3 points for all data items(Above 3 points).

- We have to check if they are **view equivalent** or not.

- S1:-

1. T1 reads X first from DB

- S2:-

1. T1 and T2 reads X first from DB

> As they are **not the same**. So it is **not view eqquivalent**.

- Yes.
- If anyone of the **3 conditions** are violated then it is **not view equivalent**.
- All of the **3 conditions** are to be **matched/satisfied** for it to be **view equivalent**.

1. T2 and T3
2. T1
3. T2

- All of the **3 conditions** are to be **matched/satisfied**.
- So **S1 and S2** are **view equivalent**.

1. T2 and T3
2. T1
3. T2

- All of the **3 conditions** are to be **matched/satisfied**.
- So **S1 and S3** are **view equivalent**.

1. T2 and T3
2. T1
3. T2

- All of the **3 conditions** are to be **matched/satisfied**.
- So **S2 and S3** are **view equivalent**.

- Question.
- It is **view equivalent**.

- Yes, we have to check **9 things**.

View Serializability

- A schedule is called as view serializable if it is view equivalent to any serial schedule.

- We have to **try multiple times**.

- Question.
- **T1 and T3** cannot come after **T4**.
- **T2** should come after **T1**. **T2** has to come **immediately after T1** as there is **W(X)** at **T(3)**.
- Otherwise **T2** will read **X** from **T3**.

- We have to make a **serial schedule** of **T3, T1, T2 and T4** and check the **view equivalent**.

- Yes it is **view equivalent**.

- It is **view serializable**.
- The **given schedule** should be **view equivalent** to **one of the serial schedules**.

- Question.

- We are not able to make a **serial schedule** to check **view equivalent**.
- So it is **not view serializable**.

- After **T3**, **T1** should run because in **Y** data point, **T1** writes last.
- **T2** should come after **T1** as **T2** writes last in **X** data point.

- HW.

Role of abort or rollback

- If **abort or rollback** is written in a **transaction** then we will **not include** that **transaction**.

- Do not include **Transaction, T1** above.
- Do not incloude transactiosn which are having **abort or rollback**.

- In the given example **T2 and T3** are **only checked**.

- Given schedule is **view serializable**.
- Sequence:-
- T2 -> T3

- T3 -> T2 [Wrong]
- It is because **final/last write** is **T3** and not **T2**.

Doubts

- Yes.

- Question.

- 100 outside is **easier** than **200** outside.

- Better possibility.

transaction-concurrency-control-part-iii(25) [29th July 2023]

- We have proved that a **given schedule** is **view serializable** then it is a **good schedule**.
- Not all **good schedule** are **view serializable**.
- All **good schedule** are **view serializable**?

> **NO**.

- All **view serializable** are **conflict serializable**?

> **NO**.

- All **conflict serializable** are **view serializable**?

> **YES**.

- Not conflict serializable but not view serializable.

- No. of nodes **equals to** the **no. of transactiosn**.
- **Runtime complexity** to **find the cycle** -> n + |E|.
- No. of nodes(n)
- No. of edges(E)

- Conflict Serializability

- View Serializability
- Time(K)
-
  - No. of nodes(n)
- **Runtime complexity** to **find the cycle** -> O(k * n!)

- NP hard problem.

Recoverability

- Recoverable Schedule -> When no any committed transactions should be rolled back.

- Not recoverable schedule

- There is a **problem** that **database** will not allow.

- If **T1** failed just before **commit**.
- Not **recoverable**.

- If there is a possibility of **rolling back a committed** transaction, if **yes**, then it is **non-recoverable schedule**.
- It is **recoverable schedule** only when for any case/condition and for any transaction that fails then also we don't have to **rollback any committed** transactions.

- To make **recoverable schedule**, sequence of commits, should be same as sequence of reading dirty values.

- Recoverable schedule example

- Correct.

- Types.

- Cascading Recoverable Rollback
- If **T1** fails just before commit then **T1, T2 and T3** rolled back.
- If **T2** fails just before it's commit, then **T2 and T3** rolled back.

- Cascadeless Recoverable Rollback
- When a transaction fails only that transaction will be rolled-back.

- Exactly.

- Question.
- Not recoverable.

- Question.
- Recoverable schedule.
- Rollback -> Ignore that transaction.

- Question
- Cascadeless Recoverable schedule.

- Strict Recoverable Rollback -> The **second transaction** can **dirty read or write** only after the **commit** of the **first transaction**.

- Dirty read and write can be done in another transaction only after commit of first transaction.
- Cascadeless recoverable is also included in **Strict Recoverable Rollback**.
- If it is a **Strict Recoverable schedule** then it is also a **Cascadeless recoverable schedule**.

- Yes.

- Recoverable schedules.

- Question.

- Yes.

- Recoverability definition -> It is dependent on **dirty read**.
- **Dirty read** -> The upper/ before transaction has **written** and the **next transcation** has **read** the value before it could be **committed**.
- If we include **write on write** then it becomes **strict recoverable schedule**. We will not do **write on write, read on write and write on read**, until the upper transaction is committed.
- In **strict recoverable schedule** -> If the upper/ before transaction has **read** then the **next transcation** can only **read** and cannot **write**. If the upper/ before transaction has **written** then the **next transcation** cannot do either **read** or **write** unless it is **committed**.

- The **conflict serializability** method we had cannot be used **practically** because for **practical** implementation we have to know the **future transaction statements** and the sequence of run of the transactions, which is **not possible**.

- **Conflict and view serializability** is not implemented practically.

Locking Protocols

Lock

- Lock
- How many locks?

- Yes.
- We can put **locks** on **different data items** at the **same time**.
- One lock for **each data item**.

- It depends on the **operation**.
- If it is **both read-read** then there is **no problem**.
- If on a **database item**, the **transaction** wants to do **only read** then it will take **shared lock**. If the **transaction** wants to do **write/read both** then it will take **exclusive lock**.
- There is **no problem** with **read operation**.

- Shared Lock -> For only read operations
- Exclusive lock -> For read/write operations both.

- Lock_Ex(X) -> It is allowed only when **T1** unlocks **X**.
- **Shared lock** doesn't have **write** permissions. **Shared lock** has to be **updated** to an **exclusive lock** for **write operations**.
- **0** -> Available, Lock not taken.
- **1** -> Not Available, Lock have been taken.

- Lock: Busy waiting

- **Busy waiting** for **exclusive lock** on **X**.

- We **blocked** ourselves. When the **lock** is available, we will be informed and we can now take **lock**.

Doubt

transaction-concurrency-control-part-iv(26) [30th July 2023]

- We didn't get **exclusive lock** because there was already **shared lock** taken on **X** by **T1**.
- **T2** is in **busy waiting** state.

- After the **shared lock** taken on **X** by **T1** was removed then the **exclusive lock** is

taken on **X** by **T2**.

- Busy waiting -> The transaction will run continuously and it will try to acquire the **lock**. It will not be able to **acquire the lock** and it will keep trying again and again.

- The solution of **busy waiting** -> Whichever transaction wants to take a **lock** which is **not available** then the transaction is **blocked** and it falls under **blocked transactions**.

- After sometime, **T1** has unlocked the lock taken on **X**. As soon as it(X) is available, **T2** transaction is **unblocked**. **T2** transaction has come back and the two statements of **T2** are run.
- **T2** transaction didn't do any **busy waiting**.
- **Advantage of removing busy waiting** -> The unnecessary usage of resources of DBMS has been **saved**.
- Time is **saved**.

- On the **same data item**, we can put **multiple shared locks**?

YES.

- **W(X)** on **T2** tried to take **exclusive lock** on **X** but it shouldn't take because **T1** has already taken a **shared lock** on **X**.
- So, the **transaction, T2** is in **blocked state**.
- **T3** has **R(X)** and it tries to take **shared lock** on **X**. **T3** is successful in taking **shared lock** on **X** even though there is a **shared lock** already taken by **T1** on **X**.
- It is because **shared locks** can be taken on the **same data item**, multiple times.

- **T1** has released it's **shared lock** on **X** and because of that **T3's shared lock on X** is also gone. Now, **T2** is **unblocked** and **T2** can take **exclusive lock** on **X** now.

- So now, **T3** has **shared lock** on **X** and **T2** has **exclusive lock** on **X**.
- This is a **problem**.
- If **two shared locks** are taken on the **same data item** then **no. of unlocks** should be the **same** as the **no. of locks** taken. As in here, **two shared locks**, so **two unlocks** should be there before **T2** can take **exclusive lock** on **X**.

- When the **count = 0** then only the **lock** should be actually **unlocked**. After that only **T2** transaction should be **unlocked**.

- Blocked transactions is unblocked only when **count** becomes **zero(0)** for **multiple shared locks**.

- Here, when a new transactions keep asking for shared locks on **X**, then **T2** may **starve**.
- There is a **solution** for **starvation**.

- When **T4** tries to take **shared lock** on **X**, then there is a **condition check** that happens. If no any blocked transaction on **X** then allowed otherwise blocked.

- We are checking that there is any transactions that are in **blocked state** due to **exclusive locks** that are taken on **X**. If so then **shared lock** cannot be taken and the transaction trying to take **shared lock** will be **blocked** as well.

- If **T1** transaction has **unlocked** the **shared lock** taken on **X**. Then there are **two ways/cases** which can be taken. Either **one of the blocked transactions** will be **unlocked** in **FIFO order** or **all of the blocked** transactions are **unlocked**.

- Upgrade -> The **shared lock** taken **X** and now that **shared lock** has been **upgraded** to **exclusive locks**.

- Downgrade.

- Rules to **acquiring the lock**:-

Locking protocols

- 2-phase locking protocol(2PL)

1. Basic 2PL
2. Strict 2PL

3. Rigorous 2PL
4. Conservative 2PL

- Basic 2-phase locking protocol

- Yes.

- **Correct** according to **basic 2PL**.

- **lock(z)** not allowed after **unlock**.
- We can take **locks** at anytime but after **unlock** we cannot take **any locks**. This is the **only condition**.

- **Correct** according to **basic 2PL**.

- **Upgrade** happens within the **same transaction**.

- The **given schedule** didn't run as it is because of **basic 2PL**, the style of the run has been changed.
- So, we will say that the **given schedule** is not allowed under **2PL**.

- Given schedule is not allowed ubder basic 2PL because it does not run same as given.

- Every schedule which is allowed under basic 2PL, is conflict serializable also.

- Not serializable.

Basic 2 Phase Locking Protocol

- Whenever needed **write 'lock** and once we have **written 'unlock'** then we cannot **write 'lock'** again.
- We will acquire all the locks at the start and when we start releasing the locks then we will only release the locks and nothing else, we cannot take anymore **locks** after an **unlock** is done.
- After **unlock**, we cannot take any **locks**. It is for **any data-item**.
- Take all of the **locks possible** at once and when we started **unlocking** then we will only do **unlocking**, no more taking **locks**.

- Once we start **unlocks**, we cannot do **any locks** on **any data-items**.

- This is why it is called as **2 phase locking protocol**.

- Yes.

- If we did **unlock(X)** before **W(Y)** then also it would be **correct according to Basic 2PL** because we are not accessing **X** anymore after that. So no problem.

- **lock(Z)** is not allowed after **unlock**.

- This will work **no problem**. **T2** will run after **T1** has unlocked **X**.
- **T2** was put in **blocked state**.

- **Upgrade** happens within the **same transaction**.

- In the **given schedule**, when we applied **basic 2PL protocol**, the schedule didn't **run as it is**. It run like **below(Run actually like**.

- In the **given schedule**, it didn't **run as it is** because when we applied **basic 2PL protocol**, the style of it's run **is changed**, then we will say that the **given schedule** is not allowed under **basic 2PL protocol**

- Given schedule is not allowed under basic 2PL because it does not run same as given.
- The **sequence of run** was **changed** due to application of **basic 2PL protocol**.

- Question.

- Not Allowed under **2PL**.

- In **one transaction**, we cannot take **lock** after an **unlock**. [**IMPORTANT**]
- But we can take **lock** in the **other transaction** though.
- We **unlock(X)** in **T1** and we did **lock_Ex(X)** which was in **T2** transaction. So we are taking **lock** after an **unlock(X)** but in a **different transaction(T2)**, so **no problem** here.

- **Unlock** keh badh **lock** cannot happen in the **same/single transaction**. **Single transaction** will not do that.
- It is **possible** in **different transactions**.

- Question.

- Allowed under **2PL**.

- Every schedule which is allowed under **basic 2PL**, is **conflict serializable** also.

transaction-concurrency-control-part-v (27) [1st Aug 2023]

- Won't be allowed under **Basic 2PL**.

- Not allowed in **Basic 2PL**.

- We will run **unlock(X) and unlock(Y)** at **10:15AM** so that we can do **lock(Y)** in **T3**.

- Not allowed in **Basic 2PL**.

- Won't be allowed under **basic 2PL**.
- **T1's released lock** can be used by **T2**, **YES**.
- **T2's released lock** can be used by **T3**, **YES**.
- **T3's released lock** can be used by **T1**, **NO**, it is not possible.
- It is because when **T3 releases it's lock**, then **T1** cannot take **lock**.

- Yes.
- The lock **T1** has taken on and **T2** wants to use that then after **T1's unlock**, **T2** can use it.
- After **T2's unlock***, **T3** can use it.
- **T3's released lock** cannot be used by **T1 and T2** because we cannot do **locking later** after **unlock**.

- So there is **no chance of 'cycle'** when we apply **basic 2PL**.

- Yes.

- Yes.

- **lock(Z)** in **T3** is **not working** because there is already a **lock(Z)** in **T1**.

- We are talking about **conflicting locks**.

- There is **no cycle** of **conflict** and as there is **no cycle**, so it is **conflict serializable**.
- If allowed in **basic 2PL** then it is **conflict serializable** as well.

- Yes.
- We cannot implement **conflict serializability** practically.

- Question
- Not allowed under **2PL**.

- **Deadlock**.

- Starvation -> Indefinite waiting.

- **Strict schedule**.

- **Strict 2PL** says to follow **Strict schedule**.

- **Commit** is **important** here.
- If the **upper one** has done **write operation** then we will do **commit** of it and after that only the **lower one** can do **either read or write** operation.

- After **every write** operation, there should be a **commit**.

- Yes.

- Strict 2PL.
- Exclusive lock should not be release until **commit**.
- At the **end of the transaction** when the **commit is do** then we will release the **exclusive lock**.

- **Strict and basic** 2PL are **different**.
- **Before commit**, we cannot **unlock the exclusive lock**.

- We can **unlock**, **shared locks** anytime.

- Strict 2PL rules not broken.
- This is allowed under **Strict 2PL**.

- Question.
- This is **not allowed** under **Strict 2PL**.

- So, **Strict 2PL** only allows **Strict schedules**.

- Question.
- This is **not allowed** under **Strict 2PL**.

- Question.
- Wherever we want, we can **write commit** statement, here.
- This is **not allowed** under **Strict 2PL**.

- We are getting **deadlock** here as well.

- Commit happens at the **last/end**.

- We cannot write commit in the **middle**.

- Committed -> Finished/Terminated.
- Wapas bhejna ho to **new transaction**.

- Rigorous 2PL
- Every lock(Shared and Exclusive) should be released after commit.

- Yes, **Strict 2PL** had **shared problems**.

- In **rigorous 2PL** we can also get **deadlock**.

- Question.
- Not allowed under **rigorous 2PL**.

- Question.
- It is allowed under **basic 2PL**.
- There is **no relation** of **unlock and commit** in **basic 2PL**.

- Recoverability

- We will have **Recoverability** problem.
- **Basic 2PL** which allows have to be **recoverable**, which is **not the case**.
- **Basic 2PL** does not ensure **Recoverability**.

- **Strict and Rigorous 2PL** don't have **dirty read**.

2. Even if the schedule is **recoverable**, it does not mean that the schedule is **cascadeles**. **Cascading** can happen as there is **no trust(no bharosa)** of **commit**. We don't know where **commit** is done(no bharosa).

- **Basic 2PL** allows **dirty read**.

- Allowed under **Basic 2PL**.
- It is **recoverable** but it is not ***cascadeless**.

- It will be **cascaded recoverable**. **T1** transaction will take **T2** transaction with it as well.

- Yes.

- If **T1** transaction **failed** then **T2** transaction is also **rolled back**.

- **Basic 2PL** allows **non-recoverable** schedules also.

3. It will be **cascadeless** because we are doing **commit** first then **unlock** and after that someother schedule accesses that **data-item**. **Cascading** cannot happen and on top of that it will be **recoverable** as well.

- Yes, rec -> recoverable

4. All of them, **basic, strict and rigorous 2PL** may suffer from **deadlock**.

- Conservative or static 2PL.
- Lock all the items before the transaction begins execution by predeclaring it's read-set and write-set.
- It will **declare** it's **read and write sets**.

- Read set will have the **shared locks**.
- Write set will have the **exclusive locks**.
- **After all of the locks** are given then only it will go forward.
- **unless all of the locks** are provided/available till then it will **wait**.

- **No deadlock** condition here.

- There will be **heavy starvation** situation here.
- **Hold and wait** situation should be avoided here.
- **Either hold all of the locks** or **wait for all of the locks**.
- It is an **advanced** version of **basic 2PL**.

- 2PL.
- A schedule is allowed in **rigorous 2PL** then that schedule is allowed in **strict, basic 2PL and it is conflict serializable** as well.
- A schedule is allowed in **strict 2PL** then that schedule is allowed in **basic 2PL and it is conflict serializable** as well but we **don't know or not confirmed** if that schedule is **rigorous 2PL** or not.

- Example of **basic 2PL**.

- Question.
- It is **not allowed** under **basic 2PL**.

- It is **conflict serializable**.

- Sequence.

Timestamp

- **DOB(Date Of Birth)** of the transaction.

- At what time transaction started.

- **1999** is **greater**.
- According to **birthday**, the **timestamp value** of the **older transaction** should be **bigger or smaller**?

smaller.

- Assume there are 2 transactions T1 and T2 and their ressperective timestamps TS(T1) and Ts(T2)
- TimeStamp(TS)
- So, **T1** is **older transaction** and **T2** is **younger transaction**.

- Deadlock Prevention.

doubt-clearing-session (28) [2nd Aug 2023]

- Yes.

- Not view serializable.

Timestamp

- Timestamp -> Arrival Time -> DOB.
- The time when a **transaction** arrives for running/execution.
- TS(T1) -> Timestamp of T1 transaction.
- TS(T2) -> Timestamp of T2 transaction.

- Who came earlier?

> Whoever's **arrival time** is **smaller** that came **earlier**. So, here **T1** came **earlier** than **T2**.

- Came Earlier -> Old
- Came Late/Later -> Young.

- So, **T1** is **older** and **T2** is **younger**.

- Deadlock Prevention

Wait and Die

- They are not **shared locks**.
- **Tj** has taken the **X** item. It is already **acquired**.
- **Ti**, **tries to acquire** the **X** item.

- Now, we have to check their **timestamps**, who is **younger and older** among **T1 and Tj**.

- It means that if a **lock** is already acquired by a **younger transaction** then we are giving permission to the **older transaction** to **wait** but if a **lock** is already acquired by an **older transaction** then we are not making the **younger transaction wait**, we are **aborting** the **younger transaction**.
- The **advantage** is that a **younger transaction** will **never wait** for an **older transaction** and there will be **no circles**.

- Because of **circular wait**, **deadlock** will happen.
- We are **preventing circular wait** here(Wait_Die).
- **Wait_Die** -> Older transaction **waits** and the younder transaction **dies**.

- The **timstamps** of **two transactions** will **never ever** be the **same**.

- **Ti** which is an **older transaction** as the **timestamp(TS) of Ti** is **smaller** than **Tj**, hence **Ti** can **wait**.

- **Ti** which is a **younger transaction** as the **timestamp(TS) of Ti** is **greater/bigger** than **Tj**, hence **Ti** is **aborted** and restarted with the **same timestamp(TS)**.

Wait and Wound

- It is just the opposite of **Wait and Die**.

- Yes.
- Younger -> Wait
- Older -> Wound.

- Ti -> Older
- Tj -> Younger
- Tj has the **lock** and **older** wants to acquire the **lock**.
- We will **abort Tj** as it is a **younger transaction** which has the **lock** and **Ti** which is the **older transaction** will **acquire the 'lock'**.

- Ti -> Younger
- Tj -> Older
- Tj has the **lock** and the **younger transaction** wants to acquire the **lock**.
- So, **Ti** being the **younger transaction** has to **wait** for the **older transaction** which is **Tj**.

- In **wait and die**, a **younger transaction** has taken a **lock** and a **older transaction** is **waiting** for the **younger transaction** to release the lock. So, at a given time the **younger transaction** has released the **lock** but the **lock** was not immediately given to the **older transaction** someother **younger transaction** got the **lock**. The **older transaction** will again **wait**. The **younger transaction** relased the **lock** and another **younger transaction** got the **lock** instead of the **older transaction**.
- The **older transaction** will keep on **waiting**.
- There maybe **starvation** for **older transactions**.

- There maybe **starvation** for **older transactions** in **Wait_Die**.

- There is **no starvation** for **older transactions** but **younger transactions** may **starve** if **older transactions** keep taking **locks** for **indefinite time** in **wait-wound**.
- After a **definite** amount of time, the **older transactions** will release their **locks** and the **younger transaction** will definitely be able to take the **locks**.
- The **older transactions** will eventually **terminate**. So, there is **no sense** of **starvation**. [**IMPORTANT**]

- Question.
- T1 is older than T2
- T2 -> Acquired the lock.
- T1 -> Requests for lock.
- W -> T1 Aborted(Die)
- X -> T1 Waits
- Y -> T1 Waits
- Z -> T2 Aborted(Wound)

- Question.
- T1 -> Younger
- T2 -> Older
- Option **A**, **Wait and Wound**, example.
- **Transaction** ends in **definite time**.

- Transaction if not killed will eventually terminate and timely termiante. So there is **no chance** of **starvation**.
- After a **definite** time, the **younger transaction** will get the **lock**, guranteed.
- So, it is **deadlock free and starvation free**.

- There is **no starvation** for **older transactions** but **younger transactions** may **starve** if **older transactions** keep taking **locks** for **indefinite time** in **wait-wound**.

- After a **definite** amount of time, the **older transactions** will release their **locks** and the **younger transaction** will definitely be able to take the **locks**.
- The **older transactions** will eventually **terminate**. So, there is **no sense** of **starvation**. [**IMPORTANT**]

- Timestamp based algorithms for **concurrency control**.
- Initially **read and write** timestamps are **zero(0)**.

- Yes.
- R_TS(X) -> Youngest transaction who read X.
- W_TS(X) -> Youngest transaction who write X.
- TS(T1) = 1
- TS(T2) = 2
- So, **T1** is **older** and **T2** is **younger**.

- Basic Timestamp algorithm.
- In whichever sequence the transactions arrive in that sequence/order only they(all transaction) should follow to run. If there is a confict in the sequence then the **Basic Timestamp algorithm** will reject that sequence.

- There is **no conflict** in **read-read** operations but in **read-write, write-read and write-write** there is **conflict**.

- TS(T1) = 1
- TS(T2) = 2.
- **T1** is **older** and **T2** is **younger**.
- First **old** then **young** which is **T1** then **T2**.
- T1 -> T2.

- There is **conflict** between **read-write** and **write-write**. So, we will not allow.
- We will not allow **conflicting statements** in the **opposite sequence(T2 -> T1)**.

- This not allowed.

- After the **young transaction's** read or write, we will not allow the **old transaction's write** operation.
- For **every statement**, the **algorithm runs**.

- If a **younger transaction** has done **read or write** operation then the current transaction's **write** operation is **rejected and aborted** and the current transaction is **rollbacked**.
- So the **write** operation of **T1** transaction is **aborted** and the whole **T1** transaction is **aborted** and **rollbacked**.

- If the **T1** transaction is **aborted** and **rollbacked** then the **T1** transaction will **restart** with a **younger/young timestamp**.

- If either of the **two condition** are true then we will **execute** the **write operation** of **T2** transaction.

- The **youngest transaction** that has **written on X**, it is **T2** transaction. So, we have to update the value of **W_TS(X)** which will be **W_TS(X) = 2** now. Who has **written** that transaction's timestamp(T2 = 2) will come to **W_TS(X)**.

- As the **write** operation is allowed so the **young transaction** has got the permission to **write**. So, we have to set the current transaction's timestap who has performed **write** operation in **W_TS(X)**.

- We are simply **checking** that if someone has to perform **write** operation then is it **allowed or not**. Before the **write** operation, someone else has done **read or write** operation then the **write** operation is **not allowed**.
- If before the **write** operation an **older transaction or the same transaction** has done the **read or write** operation then the **write** operation is **allowed**
- If before the **write** operation an **younger transaction** has done the **read or write** operation then the **write** operation is **not allowed**

- **Younger transaction** should not do the **read or write** operations before the **write** operation.
- If a **Younger transaction** does then we **abort** the **write** operation.

- **R(X)** operation of **T1** transaction is **rejected** and the **T1** transaction is **rollbacked**.

- R_TS(X) = max(R_TS(X), TS(T)).

- Yes.

- Ultimately we have to **check** after the **younger transaction's operation** we have to stop the **older transaction's operation** otherwise we will get **conflict**.

- Example.

- **No conflict** in **read-read** operation.

- Yes.

- Maximum wala point:-
- **R_TS(X)**, we will not update **read's timestamp** with the **current transaction's timestamp** always.
- On **W_TS(X)**, we will always update **write's timestamp** with the **current ttransaction's timestamp** because we will never allow the **young transaction** to perform **write operation**.

- Read operation timestamp.

- All statements are allowed.

- Example.

- **Read-Read** meh **no conflict**.

- For the **W(X)** in **T1**, we will check the timestamps of both **read and write** operations.
- Any **young transaction** has done **read or write** operation before the **W(X)** in **T1**?

> **Yes** [Read timestamp].

- So we will reject **W(X)** operation and whatever **T1** operation has done **rollback** those operations as well.
- **T1** operation is **aborted and rolledback**.
- So, **T2 and T3** transactions are **completed** . **T1** is **aborted**.

- Question.
- **T2, T3 and T1** transactions are **aborted**.
- **T4** transaction is only **completed**.
- For **R(X)** in **T1**, **younger transaction** has **written** before it, so **abort**, **T1** transaction.
- **T2** transaction is **aborted** because a **younger transaction(T4)** has **read** before **W(X)** in **T2**.
- **T3** transaction is **aborted** because a **younger transaction(T4)** has **read** before **W(X)** in **T3**.
- **T4** transaction is **not aborted** because a **not younger transaction** has **read** before **W(X)** in **T4**. Only **T4's**, **R(X)** has **read** which is within **T4** transaction only and **not a younger transaction**. So, **T4** transaction is **completed**.

- Question.
- **T2 and T3** are **completed**
- **T1** is **aborted**.
- **T1** transaction is **aborted** because a **younger transaction(T2)** has **written** before **W(X)** in **T1**.
- Rollback -> T1 transaction never came.

- That's why we set **R_TS(X) = 0**.
- We will **reverse/undo** all of the **changes** made by **T1 transaction**.

- We are getting the **final result** as **expected**.
- The middle **W(X)** in **T2 and T3** are getting overwritten by the **W(X)** in **T3**.

- For a **serial sequence** of **T1 -> T2**, the **final value** of **X** must be the value writthn by **T2**.

- The **benefit** is that we don't have to **revert** the **T1** transaction.

- Allow the operation according to **Thomas Write Rule**.
- Skip -> Don't perform the **operation** and don't make any timestamp changes as well because of the **operation**.

- We got the **final value** as **expected**.

- No, it is for **write-write** operations comparision only.

file-organization-and-indexing-part-i (29) [3rd Aug 2023]

- **Basic Timestamp algorithm** is also called as **Timestamp ordering** as well.

- It is because **W(X)** in **T2** has **written** earlier/before than the **W(X)** in **T1** and **T2** is a **younger transaction** compared to **T1**. That's why **T1** transaction is **aborted and rolledback**.
- **W(X)** in **T2** has **written** earlier/before the **W(X)** in **T1**.

Thomas Write Rule

- Thomas Write Rule -> If a **young transaction** has written earlier and an old transaction has come to **write** then **basic timestamp algorithm** will **reject** it. **Old transaction** is not allowed to **write** after an **young transaction** has written earlier. **Thomas Write Rule** says that if there is a **write** in the **older transaction** then **allow** than **write** operation.

- If we write it in-order then first we will **read** then there is a **write** in **T1** and we will do another **write** which is in **T2**. The **final value** of **X** will be the **value** that was **written** by **W(X)** in **T2**.

- **Skip/do not perform** the **write operation** in **T1** and tell that it is **done**.

- Advantage -> We don't have to **rollback** the **T1** operation now. We also got what the **expected result** was.
- If a **younger transaction** had done **read** operation before the **write** operation in **T1**, then we had to **abort T1** operation, **thomas write rule** can't do anything.

- According to **thomas write rule**.
- **T1** transaction is **aborted** because before the **W(X)** in **T1**, **younger transactions(T2, T3, T4)** have done **read/write** operations and the **older transaction** which is **T1** wants to do **write** operation which is **not allowed**.
- The **W(X)** in **T3** is **skipped** because first we checked if a **younger transaction** has done **read** opration than **T3** but there is **none**. So now we checked if a **younger transaction** has done **write** operation than **T3** and we found that **W(X)** in **T4** and **T4** is a **younger transaction** compard to **T3**. Because of **thomas write rule** the **W(X)** in **T3** is **skipped**.

- Yes.
- Anyways we are getting whatever the **written value**, the **T4** transaction has done at the **end**.

- The **basic timestamp algorithm** will run the process/transactions in such a way that the transactions which is **arriving first is run** then the next then the next and so no.

- Timestamp of T1 -> 1
- Timestamp of T2 -> 2
- Timestamp of T3 -> 3
- Timestamp of T4 -> 4
- First **T1** had arrived then **T2** then **T3** then **T4**.
- T1 -> T2 -> T3 -> T4
- In the **above order** only we have to run the **transactions**.

- Question.
- The **2nd W(X)** in **T2** is **skipped** because of the **W(X)** in **T3** which is a **younger transaction** and had **written** earlier then **T2**.
- The **2nd W(X)** in **T3** is **skipped** because of the **W(X)** in **T4** which is a **younger transaction** and had **written** earlier then **T3**.
- The schedule is **allowed**.

- As there is **no waiting**, so there is **no deadlock** here.

- Timestamp ordering algo:-

1. Serializability.
2. No deadlock.

- Starvation may happen but it depends, if a **restarted transaction gets a younger timestamp** then **Starvation** will not happn.

- Eliminates the possibility of **starvation**.
- Arrival time changes.

- Timestamp.
- All the **Timestamp schedules** are under **conflict serializability**.

- Not conflict serializable.

- TS -> Timestamp.

- Full/complete/whole schedule allowed as it is.

- Yes. Good point.

Doubts

Memory Structure

- RAM -> Main Memory -> Nano seconds(ns)
- Disk or SSD -> Secondary Memory -> Mili seconds(ms).
- **DB tool** run on **CPU**.

- Select * from table -> Many many records(rows) will come. Reading many many records(rows) from **secondary memory** and bringing them to **RAM** will take a lot of time.
- **Access time** of **secondary memory** is **more(greater)** than **main memory**.

- Optimization.
- Query optimizer -> Runs query in such a way that **minimum time** taken to fetch records from disk.

- select ename from employees where dataofjoining > 2001 and salary > 50000.

- First we **filtered out** the **table rows** with the **dataofjoining > 2001** condition. We will get **5000** rows. On those rows we used the **salary > 50000** condition and we did comparision on the **5000 and 100** and then we got **82** rows which match both the conditions.

- First we **filtered out** the **table rows** with the **salary > 50000** condition then we would have gotten only **100** rows. On these **100** rows, we would have run the **dataofjoining > 2001** condition and then we got **82** rows which match both the conditions.
- In the **second query run**, the **no. of rows** fetched in the **first time** is **significantly lower** than the **first query run**, which is from **5000 to 100 rows**. So we have to do some much **less comparisions** here. **Disk access** would have been **less** and **time for comparision** would have been **less** as well.

- Yes.
- When we **optimize the query** in such a way that it **runs** and it take **min. no. of comparisions** and **time to run entire query** will be **less**.
- That's why **SQL** is called as **non-procedural query language**.
- We are just writing the **query** and we are not telling the **DB tool** how to **run the query**.
- How the query has to be run -> It is the work of the **optimizer**.

Disk Blocks and Record Storages

- **Block size** is from **512 bytes to 4 kilobytes**.

- Whole table will come within a **single block**.
- Entire table stored in **one block** only.

- 16 different blocks needed to stored the **single table**.

- Nobody knows where the **employee** with **EID = 76** is **stored** in the table.
- We have to **check all of them** to find the **employee** with **EID = 76**.

- With the **help of somebody** we can already know **where, which** record is **stored** then we can **directly** reach to that **record**. If not, then we have to do **linear search**.
- In the **regular cases**, search each record linearly to get the required record.

- Sorted with **employee ID(Eid)**.

- In general, all blocks are accessed linearly.
- But if records are stored in sorted order of eid, then only till that block where eid becomes 90.
- This Chapter -> Physical level of database design.

- Condition -> Salary > 50000
- We **sorted** the **table** in the **basis of EID**.
- According to **salary**, will we get **sorted content**?

No.

- Now again, we have to do **one by one** checking of **all the blocks** and their content.
- It is also a **big decision** that when we **store the data** in the **database**, storing the data in the **disk** in the **physical design**, it is **very important** to check on **who's basis** we should **order the table** so that **max. no. of queries** run in the **minimum time**.
- Who is **needed more** keep that in **ordered manner**.
- Who is **not needed** much keep that **randomly**. If we have to check, we will do it **linearly**.

- **Physical storage** of DB table:-

1. Basis on which column the records must be ordered in disk so that maximum number of times. Queries can provide quick results.
2. Can we reach to a specific record directly. It is done through **indexing**.

file-organization-and-indexing-part-ii(30) [4th Aug 2023]

Spanned VS Unspanned File organization

- After putting/storing all of the **6 records**. We have filled **6 * 5 -> 30 bytes**. We are left with **32 - 30 -> 2 bytes**.
- How we will store the **7th record**?

> We can start storing the **7th record** in the **remaining 2 bytes of block '0'** and the **rest 3 bytes** in the **next block** which is **block 1**.

- If we keep a **record** in **two different blocks** like above because exactly fixed no. of records were not coming to **block '0'**. Some space was **left over**. We didn't want to **waste that left over** space. So we started storing the **7th record's 2 bytes** in **block 0** and the remaining **3 bytes** in **block 1**.

- This is called as **Spanned file organization**.

- **Yes**, if we have to **access 1 record** then we need **two blocks**. As the **7th record** is stored in **two different blocks**.

- Assuming table has **100** records. No. of blocks to store db table.
- Disk block size -> 32 bytes
- Record size -> 5 bytes
- Total size -> 100 records * 5 bytes -> 500 bytes
- No. of blocks -> Ceil(500/32) -> Ceil(15.6) -> 16 blocks.

- Spanned.
- In **OS language** it is called as **internal fragmentation**.

Unspanned

- A record is stored completely in only one block always.

- Ofcourse.

- Question.

- In **unspanned** we cannot keep **half record** some where else. The extra space which is **left**, that will not be **used**. That's why we took **floor** value when finding the **no. of records per block**. We can keep **34** full/complete records.

- No. of blocks.
- 14 complete blocks and **0.7 or 70%** of the **15th block**.
- So that's why we took **ceil** value when finding the **no. of blocks**.

Indexing

- Used to access records in **less time**.
- For **indexing**, we need **separate pages** for them.

- Yes.
- To store the **indexes** we need **blocks** and these **blocks** are called as **index blocks**.

- Where we have kept the DB records/table rows, they are called as **data blocks**.

- When the **DB table** is stored then in technical term, it is called as **Database file**.

- When the **index** is stored then in technical term, it is called as **index file**.

- Indexing techniques.

- Indexing techniques [Related to sequences].

1. Clustered Indexing
2. Non-Clustered Indexing

- In the sequence we have arranged the pages in the book, chapter 1 then chapter 2 then chapter 3 and so no. In that **same sequence** only we have done the **indexing**.
- In the sequence, we have arranged the **files**, in that **same sequence** we have arranged the **index**. This is called as **Clustered Indexing**.

- Clustered Indexing -> Data order and index order are **same**.
- In what order we have arranged the data in the blocks?

In the order of **Rno**.

- We will follow the **same sequence** in **indexing** as well.

- Clustered indexing -> Storing the records in **DB file** and storing the **indexing** in **index file** and the **sequence of storing** in both **DB file and index file** is the **same** then it is an example of **Clustered indexing**.
- **Data and index** order are the **same**. Like in **books**.

- Non-clustered Indexing -> Data order and index order are not the **same**.

- In the **sequence**, the **indexing** has been done in that **same sequence** the **data** in the **DB file** is **not stored/kept** and **vice-versa**.

- There is a **difference** between the **sequence** of the **data stored** and the **sequence** of the **indexing**. That is called as **Non-clustered Indexing**.

- Dense VS Sparse index.
- Dense -> Index record is for each database record
- Sparse -> Index record is for a few database records only.

- Indexing techniques.

Primary Indexing

- We can do **indexing** on any of the c**olumns**.
- If **primary key** is one column then **indexing** is done on **one column** values.
- If **primary key** is combination of two columns then **indexing** is done on **two column values**.
- **Indexing** has to be done on **primary key**.
- **Index and data** order must be the **same**.
- It is always **sparse index**.

- Sparse Index.

- Primary Indexing
- Indexing done on primary key or any super key.
- Data must be ordered on index which is **primary key**.
- It always sparse index.

- Anchor records.

- Yes.

- Question.
- How many blocks are accessed?

> **1 block access** for the **index block** and **1 block access** for **data block**.

- So, total **2 block access** are **required**.

- Question.
- In the **index block** we accessed **2 blocks** and in the **data block** we accessed **1 block**.
- So, total **3 block access** are **required**.

- We don't have to calculate the **no. of tuples(rows)**.

- Yes.

- Question.

- **32 byte** record we got.
- If **database table** is stored in **4 blocks** then **indexing** will be of **4 records** as in **primary indexing** we know that we do **sparse indexing**.

- No. of index records = No. of blocks to store DB file.

- [**IMPORTANT**]

- Question.
- No. of DB recods per block = (Block size) / (DB record size)
- No. of blocks to store DB file = (DB file) / (No. of DB recods per block)

- Index record size = (Key size) + (block pointer size)
- No. of index recods per block = (Block size) / (index record size)

- No. of blocks to store index file = (No. of blocks to store DB file) / (No. of index recods per block)

file-organization-and-indexing-part-iii(31) [6th Aug 2023]

- No issue between **spanned and unspanned**.

- If there is **free space** in every block then there will be difference between **spanned and unspanned**.

- Question.

- Spanned.

- Unspanned.
- In the **index block** if we have to do **searching** then we have to do **binary search** instead of **linear search**. The **index and the data** are **ordered** only.
- No. of comparisions?

  > For the **108** blocks, **no. of comparisions = log 108 base2 -> 7**.

- With the access of **7** blocks, we can access the block, we want to access.
- We will **search** first in the **index**.
- Every **index** has a **block pointer** which is pointing to a **block**.
- We will do **searching** on the **index block** and after **searching** we will find out the **block** in which the **data** that we want is **located/stored**.
- We are doing **binary search** on the **index block** and the **no. of block** we have to access is **7** blocks. We accessed **7** blocks and the **job** is done.
- Out of the **108 blocks**, we have to access only **7 blocks** and the job is done.
- This is where we can **optimize the access time**.
- **Binary search time complexity** -> O(log n base 2) -> logn base 2.

- **Index** is **metadata** not **overhead**.

- Yes, **spanned** can also have **internal fragmentation** because the **last block** may not be **completely filled**.
- In **unspanned**, every block can have the possibility of **internal fragmentation**.

- Yes.

- Yes.

- Clustering Indexing

- Indexing is done for **each unique value of non-key field**.
- **Non-key field** means repeated values are possible.

- This is **correct**.
- **Indexing** will not be for **every single record**, **NO**. For **every unique value**, we do **indexing**.

- We have to **store** the **starting blocks** of **every unique value**.

- Sparse Indexing.
- In **clustering indexing**, it can be **dense** as well as **sparse**.
- **Sparse** when -> When non-key field has duplicate values.
- **Dense** when -> When non-key field has all unique values.
- The indexing is done for each unique value of non-key field.

- We have to search the records in **sequence**.

- Question.
- **Clustering indexing** is not directly mentioned but the definition of **clustering**

indexing is given.

- Yes.

- In **primary indexing**, **the no. of records in index file** will be **no. of blocks to store the db file**.
- In **clustering indexing**, **the no. of records in index file** will be **no. of unique values**.
- This is the **only difference** everything is **same** in terms of solving the question.
- **Indexing kiski karni ha** woh dimag(brain) meh ah gaya toh, everything else is **same**.

- Question.

- Spanned.

- Unspanned.

- Secondary Key Indexing.
- As the data is **not ordered**, we will not know the **sequence** and for this reason we will have to do **dense index**.
- For **every record** we have to do **indexing**.
- Bi, Rj -> Block 'i' and in that record 'j'.

- **NULL** is also one of the **unique values**.

- Question.

doubt-clearing-session(32) [8th Aug 2023]

- Secondary non-key indexing.
- For **every unique value** there will be **indexing**.
- **B5** is index block for value **A**.

- **Primary, clustered and secondary key** indexing are the **most important**.

- **First level** is **sparse** and the **next level** is **dense**.

- Summary.

- Question.
- HW.

Multilevel indexing.

- Do **indexing** on the **index**.
- As they are **keys** so they will be **unique values**. **Indexes** are **unique** and on top of that they are in **sorted** order.
- We will do **primary indexing** of the **index**.
- We can do **sparse indexing** of the **anchor records**.

- Yes.

- For **2 block access**, we will get the **index's record pointer**.
- **No. of levels** = **No. of blocks accessed**.

- Without **multilevel index**
- To search indexes, **no. of blocks** to be accessed = **log b base 2**.
- 'b' -> No. of blocks required to store indexes.

- Ofcourse.

- **Note** -> One block access extra to access DB record.

- n-level multilevel indexing -> To access index(record pointer), nop. of blocks accessed = 'n'.
- **Note** -> One block access extra to access DB record.

- Question.

- For **2-level indexing** we accessed **2 blocks** and we got the **record pointer**. We have to access the **record** as well. **Another block access** to get to the **record** we wanted to **access**. We wanted the **DB row**. For that we need **+1 block access**. That why **2 + 1 -> 3 block access**.

- Yes.
- If the **DB file** is **ordered** on the **non-key field** then **DB file** is **not ordered** on the **key field**.
- **Secondary indexing** is **dense**.
- The **no. of index records** needed is **16384 or 2 ^ 14** as the indexing is **dense**.
- As it is **unspanned**, so we have to see **block by block**.

- For **every block** we need **one index**.

- Option **C**.

- We are storing **index file** here and not the **DB file**. THat's why we ignored the **32 bytes** information given.

B-Tree

- It is like **binary search tree**.

Binary Search Tree

- We can do **searching** here.

- Left -> Smaller values
- Right -> Bigger values.

- We have to do **indexing** in such a way that we have to take the help of **BST(Binary search tree)**.

- In **BST**, the **min time for searching** is **Log n**, if we have **n** nodes in the tree.

- As **no. of elements** in **BST increases** -> Tree grows **vertically** and it's height increases.
- Hence even for **searching** when **log n base 2** time required that too becomes **very large**.
- **Solution** -> Make a tree which can grow **horizontally and vertically** also.
- We have to make a tree where in **every node** we can keep **multiple keys(more than 1 key)**.
- The **advantage** is that when we insert an element or when we increase the no. of elements, then the tree will grow **horizontally and vertically** as well instead of **horizontally only** in **BST**.
- That is **B-Tree**.
- Binary Tree -> From **one node** we get **two pointers or children**.
- 3-ary tree -> Each node can have **3 children**.
- For **3-ary tree**, we have **3 children** we can have **2 keys**.
- For **binary tree**, we have **2 children** we can have **1 keys**.

- We will grow the tree in every direction.

- **Height** will be not that much and for that **searching** will take **less no. of block access**.

- Atmost -> Maximum
- **Max** is **p** pointers and **keys** is **(p - 1)**.

- Ceil value.

- Keys -> Values.

- **Except root on any node**, we have to apply the **min and max** rule of **keys**.

- For later.
- Leaves -> Leaf nodes.

- Yes, **balanced tree** always.
- All **leaves** appear on the **same level**.

- Remember the **formula** [**IMPORTANT**]
- Why learning **B-Tree**?

> We want to keep the **indexing** in **B-Tree**.

- **Every node** has **two pointers**. **One pointer** is of the **record** and the **other pointer** is for the **tree**.
- Rec pointer -> Record pointer.
- **Record** means the **row/tuple**.

- Yes.

- We have kept **max of '4' keys**.
- We have to keep **'4' record pointers** for **4** keys.
- We have to keep **'5' tree pointers** for **4** keys.

- Insertion in **B-Tree**.
- **Insrtion** is always done on **leaf node**.

- We can keep **2 keys** on a **single node**.

- We had to make **2 extra nodes**.

- Question.

- Yes.

- Yes.

- Good point, we will see **tomorrow**.

file-organization-and-indexing-part-iv (33) [8th Aug 2023]

- When the order is **even numbers like 4,6** then there is confusion. We don't know which element to send **above** during **node splitting**.
- We can have **two option**.

- Right biasing
- Left biasing

- Node split in **left or right biasing**.

- Continue with **Left biasing**.

- Assume a B-Tree of **order-p**.
- 'p' -> even and p > 2.
- When during an insertion a node is splitted then the no. of kyes in 2 splitted nodes should be?

- Question.

- Will be mentioned in the question.

- Yes.

- Question.
- Degree -> No. of children -> Pointer.

- We are not **concluding** yet.

- Option **B** is **correct** only.

- Question.

- Question.

- Question.

- In general, it is **no. of nodes + 1**. As **maximum** no. of nodes are asked.
- No. of splits -> 4 [Above question]
- No. of splits in the question = No. of levels.

- Yes.

- Practical Imlementation of node on blocks.

- B8
- root(B8).
- Idea to store B-Tree on dissk -> Keep one node of tree on a block.

- Rec pointer -> Record pointer
- Tree pointers.

- Max keys -> (p - 1)
- Max tree pointers -> p
- Max record pointers -> (p - 1)

- **One complete node's** size.
- We have to keep the **complete node** within **one block**.

- Tree pointer or block no or block pointer.
- We want to do **min. internal fragmentation**.

- Question.

- yes.

- Question.

- Height of **B-Tree**.

Doubts

- Option **B**.

- Yes.

file-organization-and-indexing-part-v (34) [8th Aug 2023]

- select * from table where key = 20;
- To search into index, 2 blocks are accessed.

- To access a specific index, **max no. of blocks** accessed = **no. of levels** in **B-Tree**.

- **4 blocks** accessed.

- In given tree, select * from table where ke between 4 and 30.
- No. of blocks accessed -> **4** blocks.

- **Deletion** in B-Tree.

- **No changes** after **deletion**.

- Question [**IMPORTANT**]

- Yes.

- No violation so **no changes** afterwards.

- Because of **min. key violation** we cannot borrow.

- Will be mentioned in the question.

- Inorder successor or predecessor will always be on **leaf node**.
- Will be mentioned in the question, whether we have to use **successor or predecessor**.

Doubts

- COA.

file-organization-and-indexing-part-vi (35) [16th Aug 2023]

- Insertion and deletion.
- Writing the **range query** there are problems. Accessing all of the values creates a problem.
- For that **reason**, the **efficiency** of the **B-Tree** is not that good.
- To support the **range queries**, the **B+ tree** had been introduced.
- **B+ tree** says that if we want to access things in the **range** then we can use **B+ tree**.
- In **B+ tree**, **all of the keys** are present in the **leaf node**.
- In **B tree**, some keys are present in the **internal node**, some keys are present at the **leaf node**.

- All of the **leaf nodes** are connected through **linked list**, either singly or doubly linked list as per the **implementation**.
- At the end of the **leaf node** we will get the **address** of the **next leaf node**.

- **13** doesn't necessary need to be a record. It is just a **value**. It is just used for **searching**.
- **13** kept so that we can either go **right or left**. There is no compulsion that **13** needs to be a **record**.

- **13** -> Anchor key.

- [**IMPORTANT**]

- [**IMPORTANT**]

- Example.
- We don't have to **search** everytime from **root**.

- Order is **same**.
- We are doing **right biasing** here. **Left biasing** also possible.
- We have just copied **2's key** above. The **record pointer(RP)** didn't go above.
- **Record pointer(RP)** is with the **leaf node** only.

- The **node split** will happen just like in **B tree** when doing for the **internal nodes**.
- No need of copy.

- All keys are present on **leaf nodes** and **internal nodes** contain only **anchor keys**.

- Question.

- We can keey anything for **anchor keys**. We do not need copy of them.
- We are doing **right biasing** on these as well.

- [**IMPORTANT**]

- Question - aneves.

- We have changed the anchor.

- If we want to distribute the key to the **left side** then **10** will go to the **left side**.
- If we want to distribute the key to the **right side** then **21`** will go to the **right side**.
- It is the implementation detail than we want to distribute the key to the **left or right** side.

- When we inserted the **8** element, we cannot distribute **9** to the **right** as it(right) is already full. We cannot take any further than the **nearest neighbour**.
- If not possible then we have to do **node splitting**.

- Question.

- We will access **5 nodes**.

- Height of **B+ tree** can be more than that of **B-Tree** for a given no. of keys.

- Yes, to store the duplicates in **B+ Tree**.

- [**IMPORTANT**]

- Question.
- We can get more **100 internal nodes**.

- To access one record, the **max. no. of nodes** we need to access is **18**.
- It is **4** here.

- [**IMPORTANT**]

- Deletion in **B+ tree** (Tomorrow).

doubt-clearing-session (36) [17th Aug 2023]

- Insertion:-

1. Node split insertion
2. Using Key distribution

- If **key distribution** is not possible/does not happen then we have to do **node split insertion** only.

Deletion in B+ Tree

- If we delete a key then the **deetion** happens from the **leaf node** only. The keys are actually present in the **leaf nodes** only.
- Deletion is performed at the **leaf node** only.

- Yes, no problem.
- Anchor keys is for going **left or right**. It is not necessary that the key will become the **anchor key**.

- Delete **15** next.
- It is not **necessary** to change the **anchor keys** as long as the **left keys** are **smaller** and the **right keys** are **greater** than the **anchor key**.

- There is no violation of min. keys.
- There is **no problem**.
- No point in updating the **anchor**, it is waste of time.

- Example below.

- Violation happened when trying to **delete '5'**. We have to see if the **immediate siblings(left or right)** have **extra keys**.
- We transferred **15**, which is **borrow '15'** from the **neighbour**.

- If **delete 12** happened then there will be **min. key violation**. Min. 2 should be there but there is **only one**.
- We looked at the **neighbours** to check if they have **extra keys** or not, which is keys more than **min. keys**.

- Question.

- Yes, they are all in the **same level** only.

- Start

- As **borrow** not possible. We have to do **merge of two nodes**.

- **Merged**. We could have done the **merging** with the **right node** as well. Both are **fine**. Anyone works.

- Root node can have min. of one(1) key.

- We have to do **merging** for the anchor nodes.

- We have to follow a pattern always. First we will try **borrowing**. If **borrowing** is not possible then we will do **merge**. First borrowing we will do from the **left** if not possible from the **left** then we will go to the **right**. If borrow not possible from both the sides then we will do **merge**. In **merge** also, start from the **left** and if not possible in the **left** then go to the **right**.

- Question.

- There is also violation at the **anchor node(20)**. We have to bring **36** from the **neighbour**.

- Like a **B tree** deletion for the **Anchor keys** only.

- **Rotation** will happen, **36** will go to the **top** and **30** will come to the **left side**.

- If we do **merging** at the **internal nodes** then we will no **remove** the **anchor**. We will pull down the **anchor** or we will update the **anchor** for **merging** to happen.
- If we have **borrowed** from the neighbours of **internal nodes** then **borrow** will not happen directly. It will happen through **rotation**.

- [**IMPORTANT**]

- B+ tree.

- [**IMPORTANT**]

- Question.
- Non-leaf node -> Internal node.
- Tree pointer -> Block pointer

- Order(p) -> 51
- Max. no. of keys -> p - 1 -> 51 - 1 -> 50 [Answer].

- [**IMPORTANT**]

- Option **A**.

- Yes
- For **B Tree**, Option **A** as well.

8p -8 + 2p <= 512 [Non-leaf node]
10p <= 520
p <= 52

- Order -> 52 [Answer]

- [**IMPORTANT**]

- Good Point.

- Doubts.

Test Series

- Test Series suggestion.

Misplaced Images

Joke