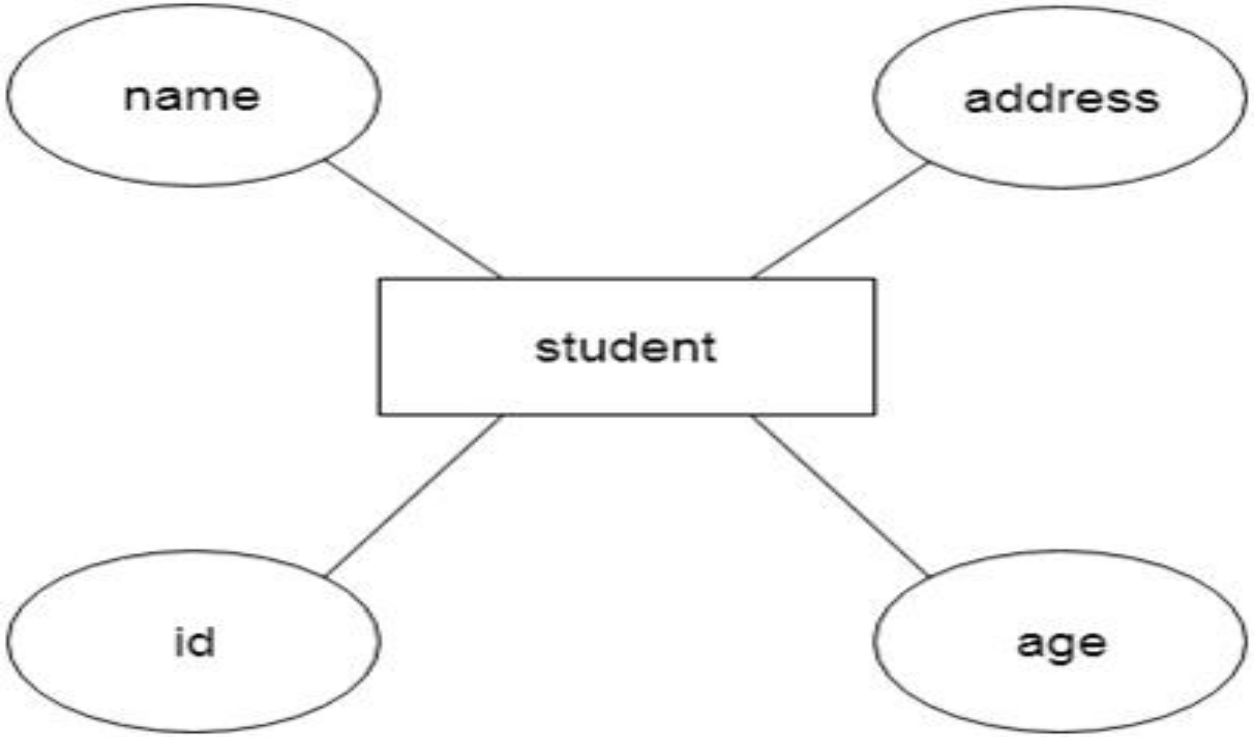
What is Entity Relationship Diagram?

Entity Relationship Diagram (ER Diagram)

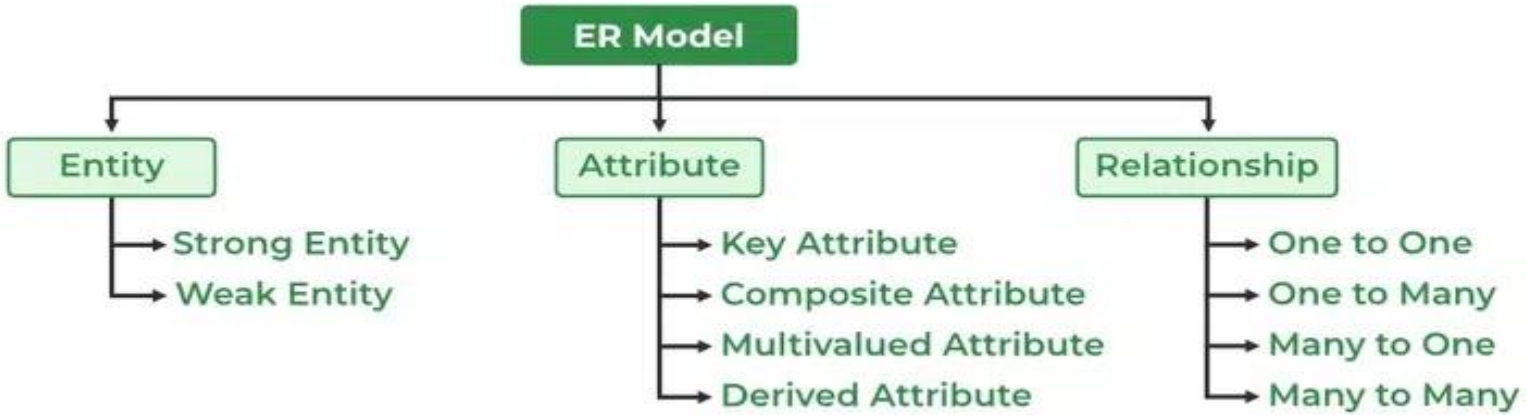
Use E-R model to get a high-level graphical view to describe the **"ENTITIES"** and their **"RELATIONSHIP"**

It is a **high-level data model**. This model is used to define the data elements and relationship for a specified system



Component of ER Diagram

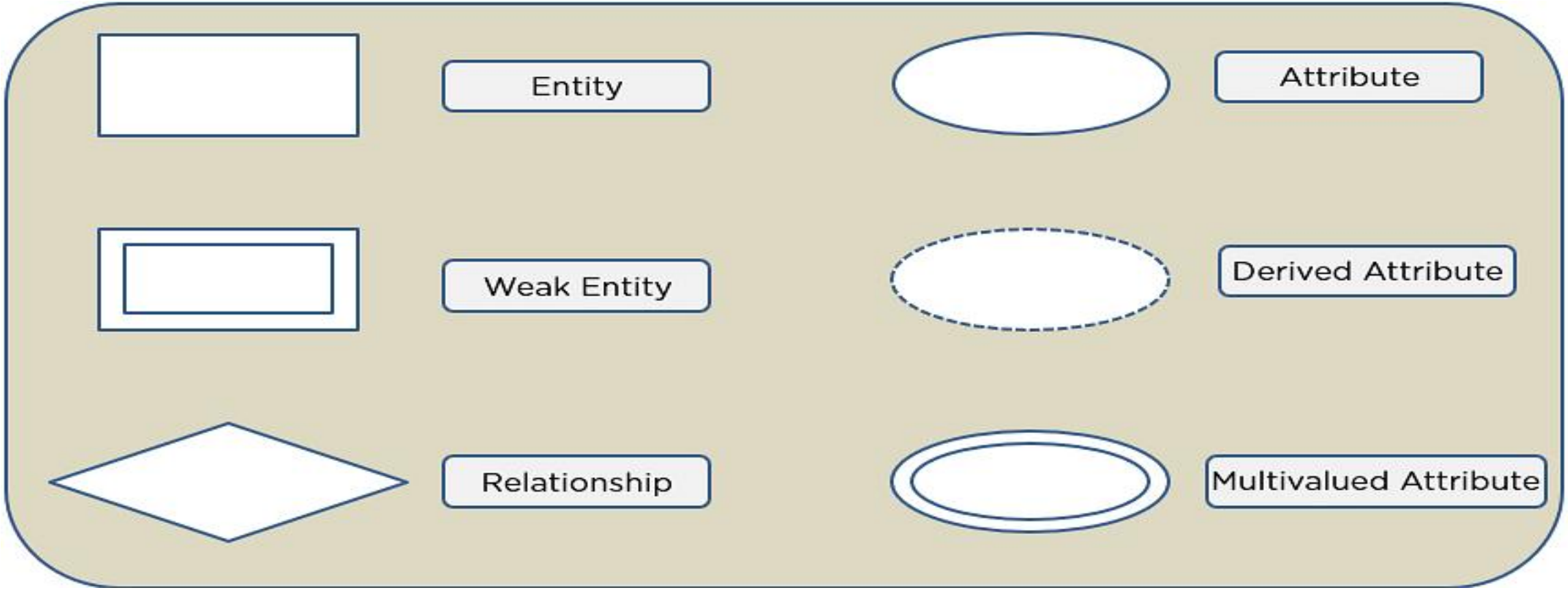
The basic constructs/components of ER Model are **Entity**, **Attributes** and **Relationships**.



Why Use ER Diagrams

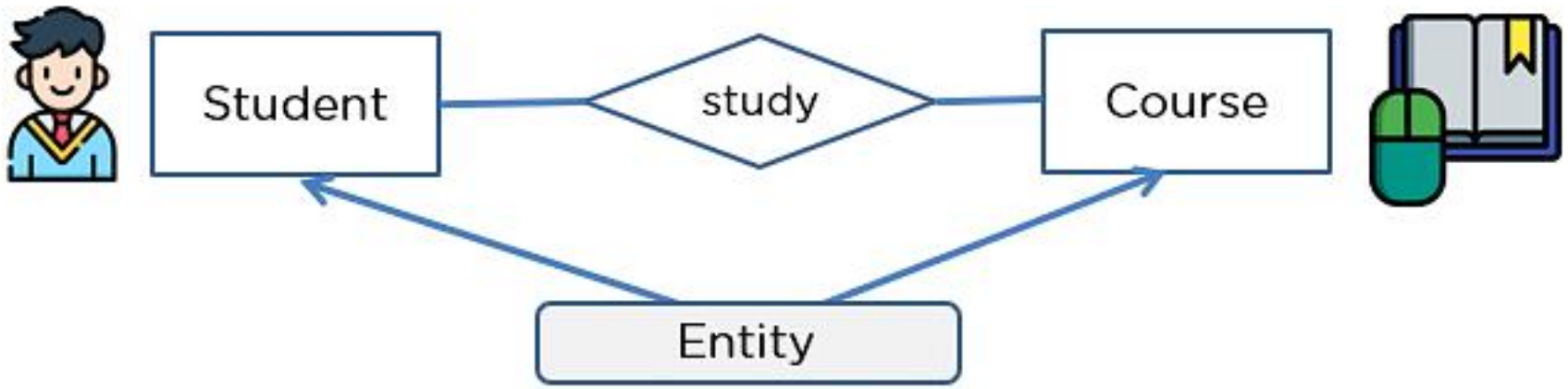
- ER Diagram helps you conceptualize the database and lets you know which fields need to be embedded for a particular entity
- ER Diagram gives a better understanding of the information to be stored in a database
- It reduces complexity and allows database designers to build databases quickly
- It helps to describe elements using Entity-Relationship models
- It allows users to get a preview of the logical structure of the database

Entity Relationship Diagram Symbols



An entity can be a real-world object or can be a living or non-living component.

What is Entity?

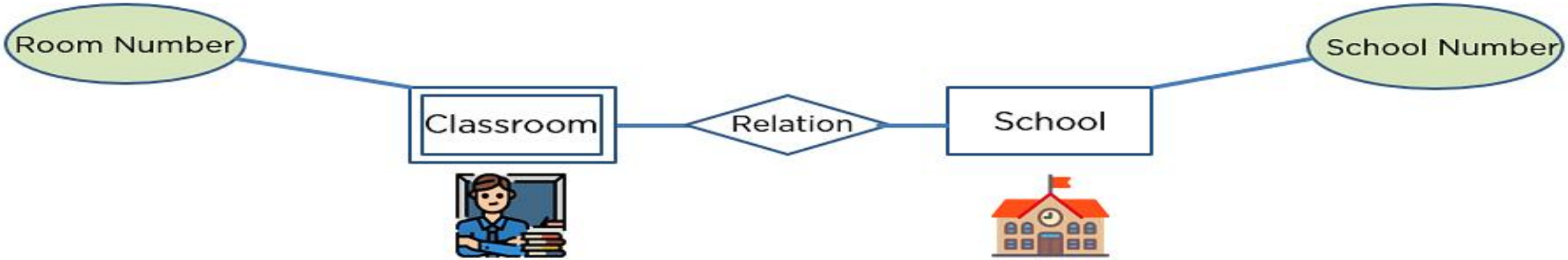


strong and weak entity

An entity may participate in a relation either totally or partially.

Strong Entity: A strong entity is not dependent on any other entity in the schema. A strong entity will always have a primary key. Strong entities are represented by a single rectangle.

Weak Entity: A weak entity is dependent on a strong entity to ensure its existence. Unlike a strong entity, a weak entity does not have any primary key. A weak entity is represented by a double rectangle. The relation between one strong and one weak entity is represented by a double diamond. This relationship is also known as identifying relationship.



entity

In relation to a database , an entity is a

- Person(student, teacher, employee, department, …)
- Place(classroom, building, …) --a particular position or area
- Thing(computer, lab equipment, …) --an object that is not named
- Concept(course, batch, student's attendance, …) -- an idea,

about which data can be stored. All these entities have some **attributes** or **properties** that give them their **identity**.

Every entity has its own characteristics.

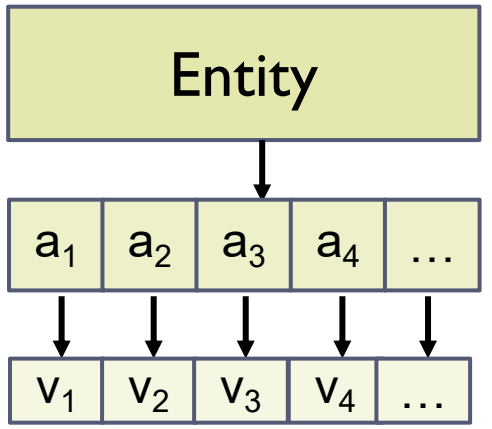
When you are designing attributes for your entities, **you will sometimes find that an attribute does not have a value**. For example, you might want an attribute for a person's middle name, but you can't require a value because some people have no middle name. **For these, you can define the attribute so that it can contain null values.**

In database management systems, **null** is used to represent missing or unknown data in a table column.

What is an Attribute?

Attributes are the properties that define a relation.
e.g. student(ID, firstName, middleName, lastName, city)

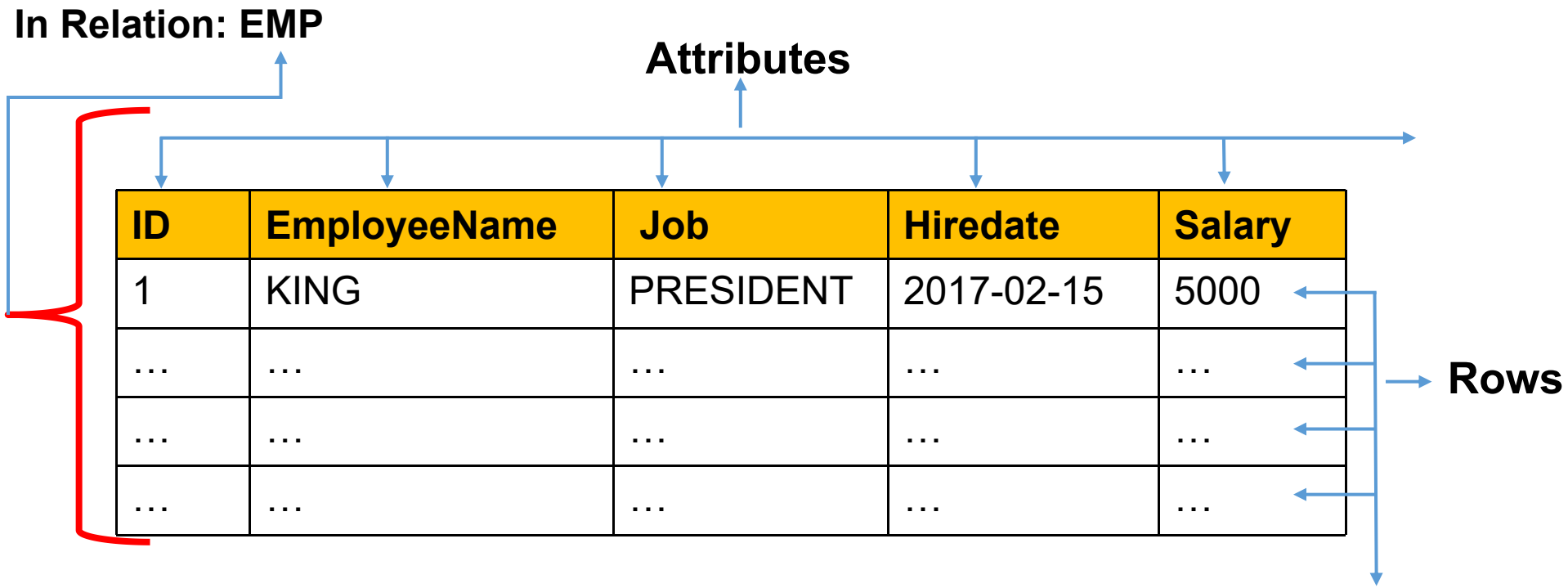
In some cases, you might not want a specific attribute to contain a null value, but you don't want to require that the user or program always provide a value. In this case, a default value might be appropriate. **A default value is a value that applies to an attribute if no other valid value is available**.



attributes

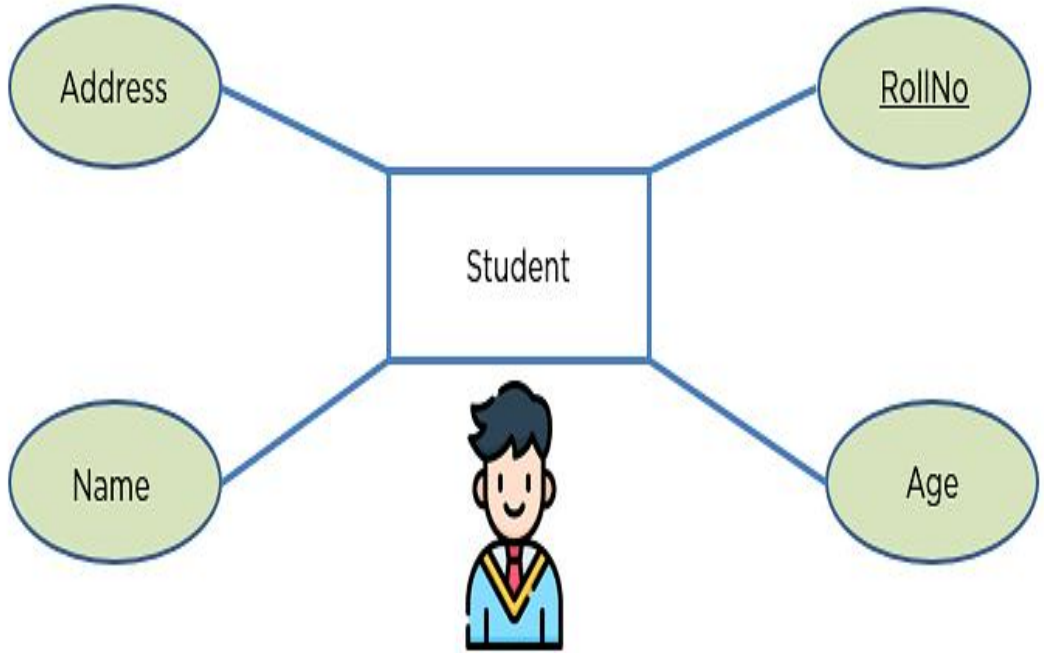
A table has rows and columns

In RDBMS, a table organizes data in rows and columns. The **COLUMNS** are known as **ATTRIBUTES / FIELDS** whereas the **ROWS** are known as **RECORDS / TUPLE**.



In Entity Relationship(ER) Model attributes can be classified into the following types.

- Simple/Atomic and Composite Attribute
- Single Valued and Multi Valued attribute
- Stored and Derived Attributes
- Complex Attribute
- Key Attribute and Non-Key Attribute



Remember:

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in different relations.

attributes

- **Simple / Atomic Attribute** --VS-- **Composite Attribute**
  (Can't be divided further) (Can be divided further)
- **Single Value Attribute** --VS-- **Multi Valued Attribute**
  (Only One value) (Multiple values)
- **Stored Attribute** --VS-- **Derived Attribute**
  (Only One value) (Virtual)
- **Complex Attribute**
  (Composite & Multivalued)

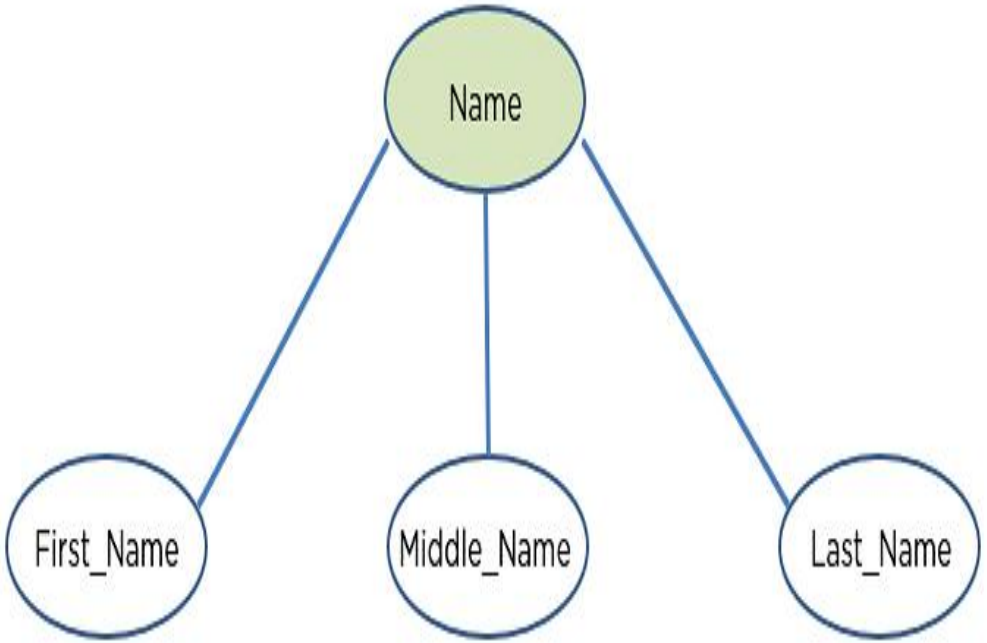
Employee ID: An employee ID can be a composite attribute, which is composed of sub-attributes such as department code, job code, and employee number.

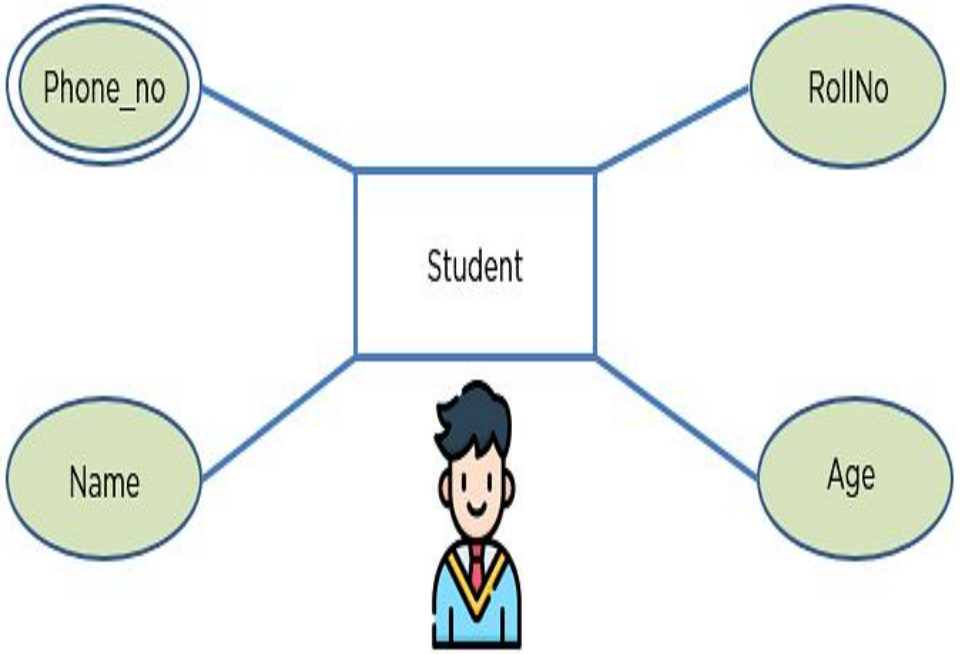
- **Atomic Attribute:** An attribute that cannot be divided into smaller independent attribute is known as atomic attribute.
  e.g. ID's, PRN, age, gender, zip, marital status cannot further divide.

- **Single Value Attribute:** An attribute that has only single value is known as single valued attribute.
  e.g. manufactured part can have only one serial number, voter card, blood group, price, quantity, branch can have only one value.

- **Stored Attribute:** The stored attribute are such attributes which are already stored in the database and from which the value of another attribute is derived.
  e.g. (HRA, DA…) can be derive from salary, age can be derived from DoB, total marks or average marks of a student can be derived from marks.

Composite VS Multi Valued Attribute



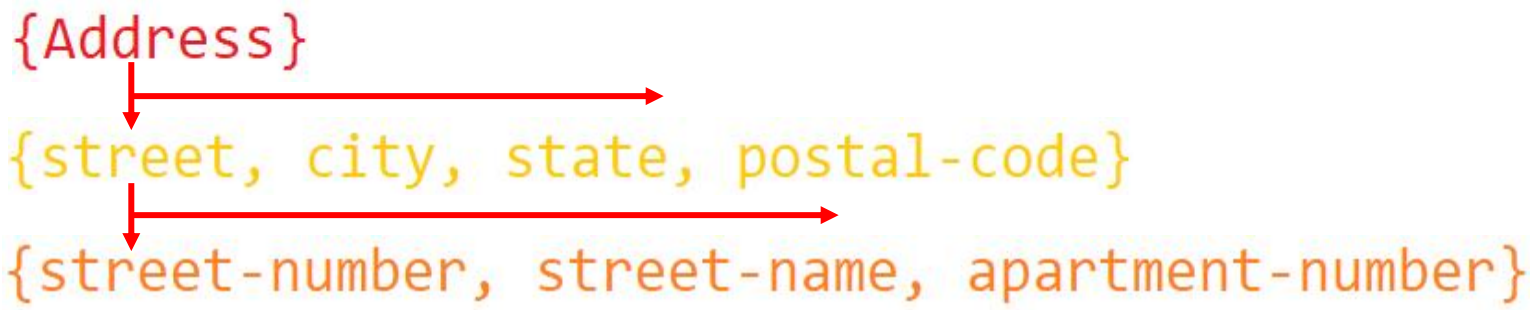


composite / multi valued attributes

Composite Attribute

Person Entity

- *Name* attribute: (firstName, middleName, and lastName)
- *PhoneNumber* attribute: (countryCode, cityCode, and phoneNumber)

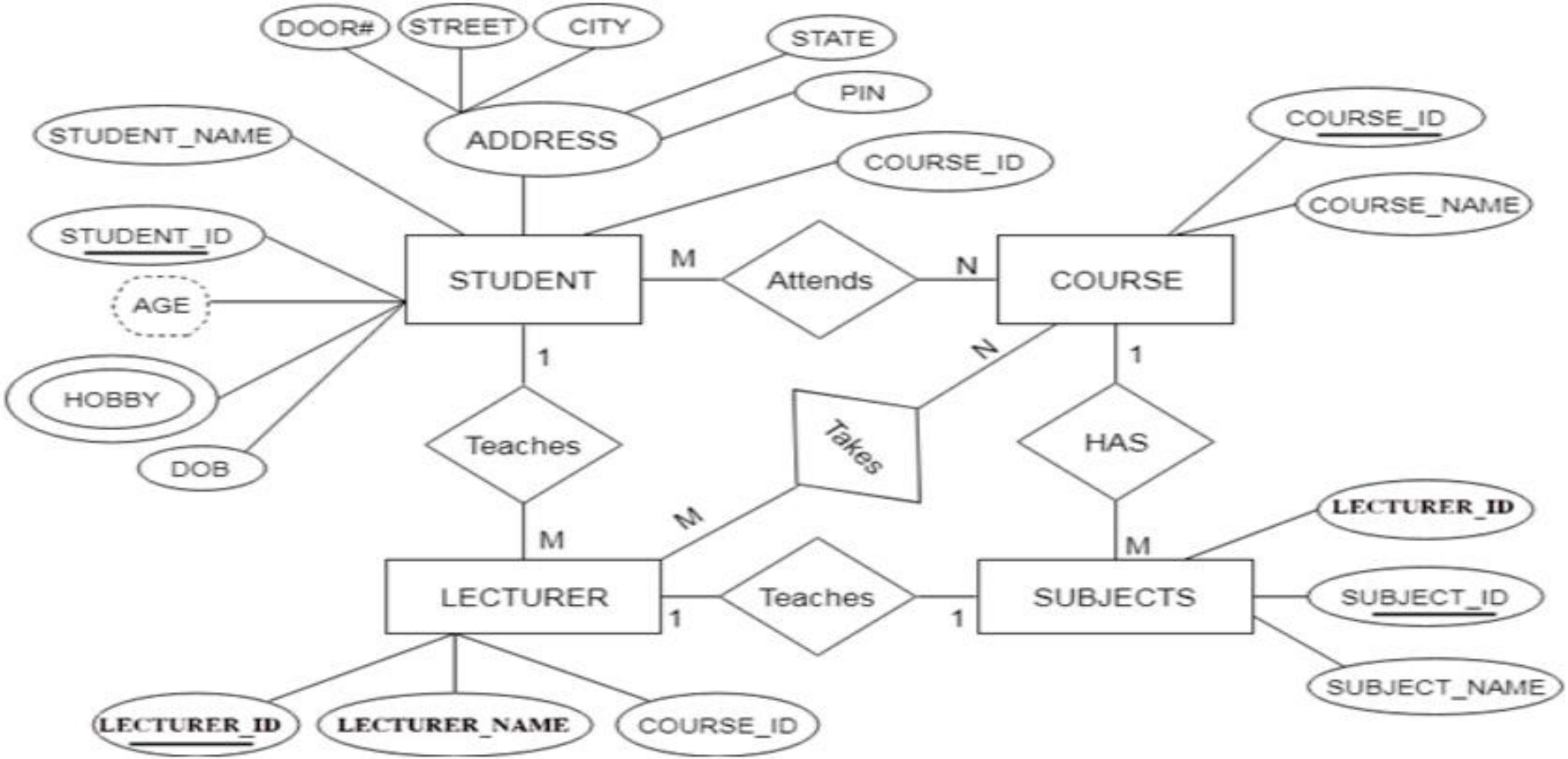


Multi Valued Attribute

Person Entity

- *Hobbies* attribute: [reading, hiking, hockey, skiing, photography, . . .]
- *SpokenLanguages* attribute: [Hindi, Marathi, Gujarati, English, . . .]
- *Degrees* attribute**:** [10^{th} , 12^{th}, BE, ME, PhD, . . .]
- *emailID* attribute**:** [saleel@gmail.com, salil@yahoomail.com, . . .]

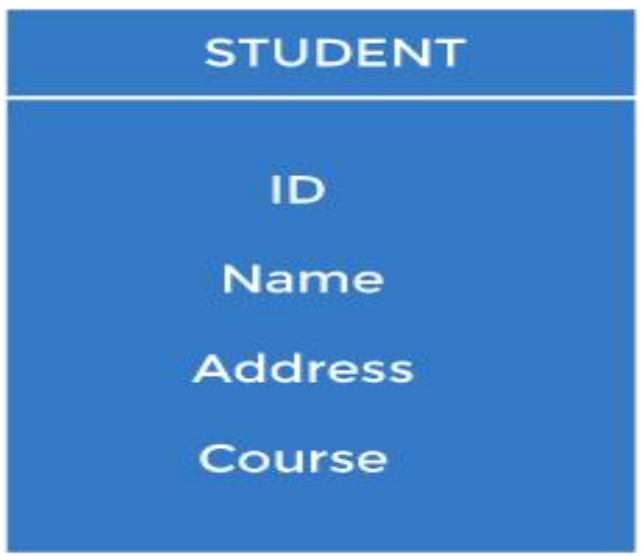
entity relationship diagram

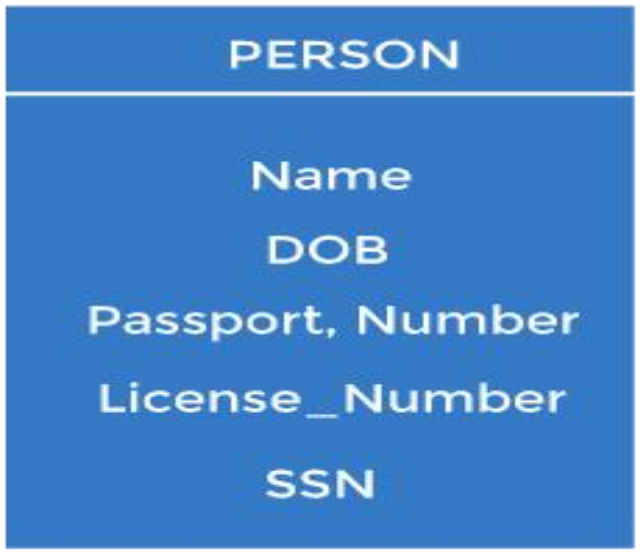


Keys

It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.

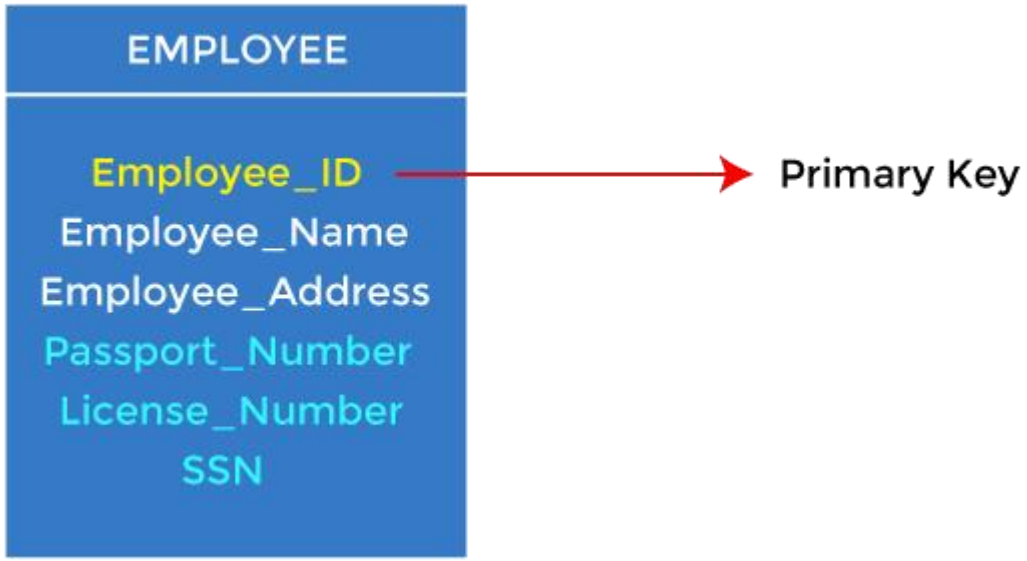
For example,
ID is used as a key in the Student table because it is unique for each student. In the PERSON table, passport_number, license_number, SSN are keys since they are unique for each person.





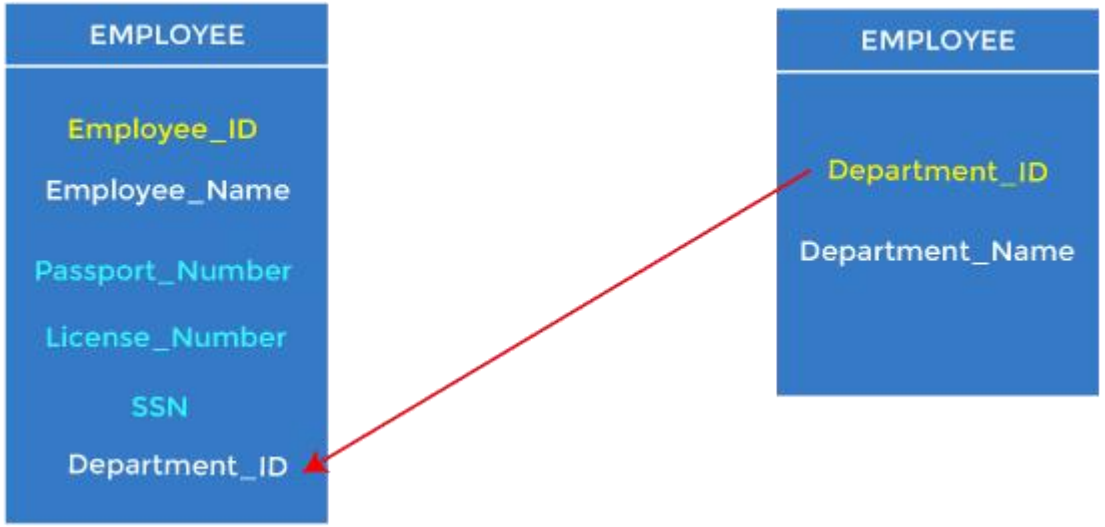
1. Primary key

- It is the first key used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys, as we saw in the PERSON table. The key which is most suitable from those lists becomes a primary key.
- In the EMPLOYEE table, ID can be the primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License_Number and Passport_Number as primary keys since they are also unique.
- For each entity, the primary key selection is based on requirements and developers.

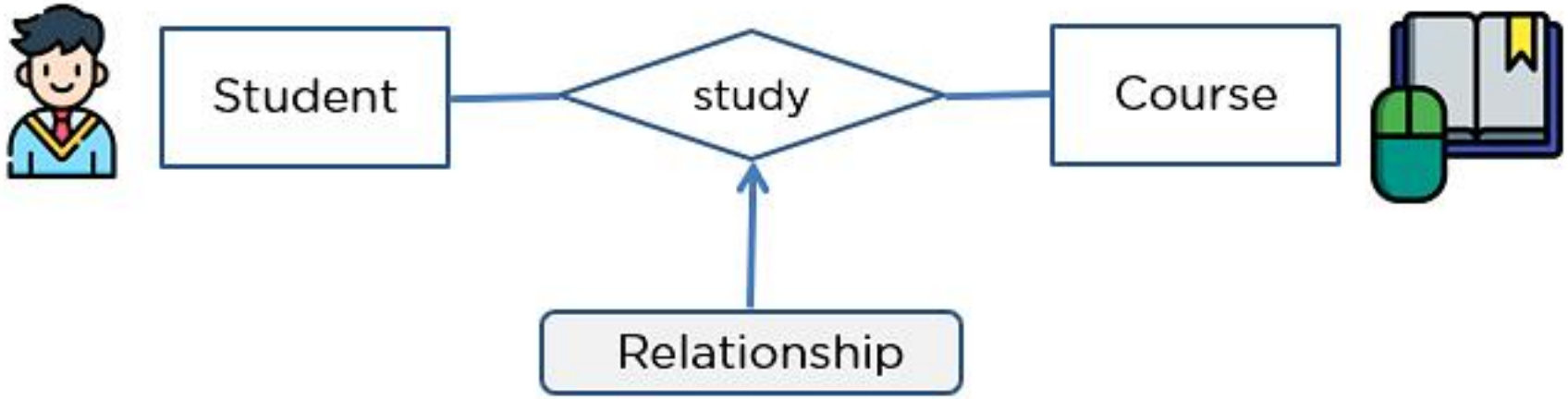


Foreign key

- Foreign keys are the column of the table used to point to the primary key of another table.
- Every employee works in a specific department in a company, and employee and department are two different entities. So we can't store the department's information in the employee table. That's why we link these two tables through the primary key of one table.
- We add the primary key of the DEPARTMENT table, Department_Id, as a new attribute in the EMPLOYEE table.
- In the EMPLOYEE table, Department_Id is the foreign key, and both the tables are related.



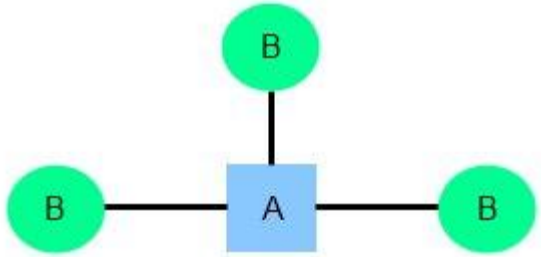
Common relationships



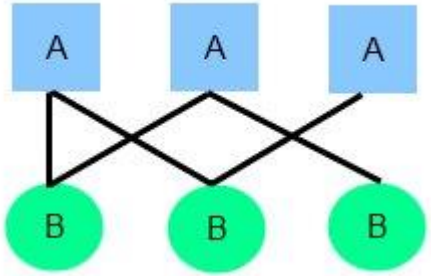
Common relationship

one-to-one (1:1)

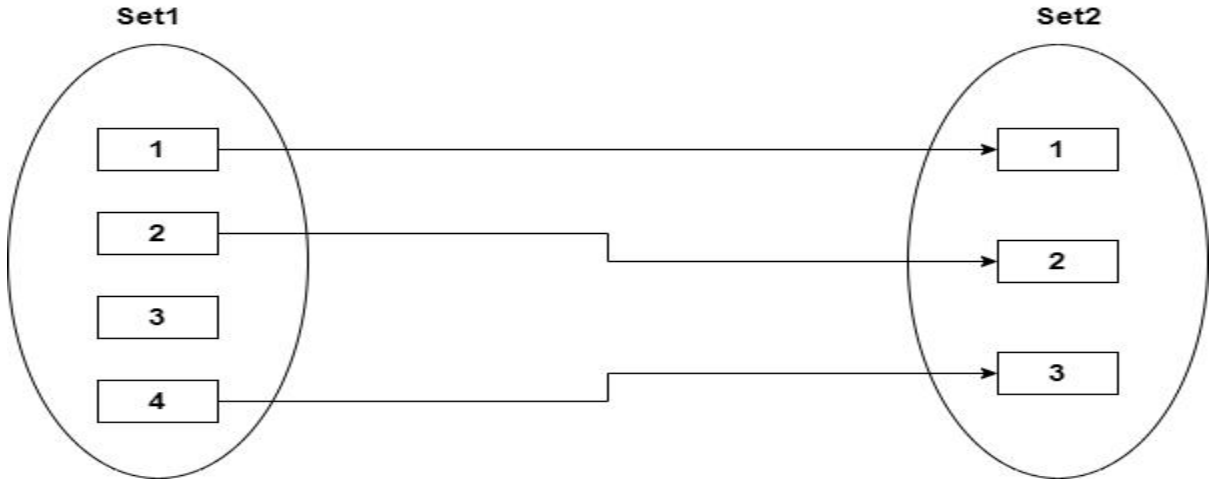
one-to-many (1:M)

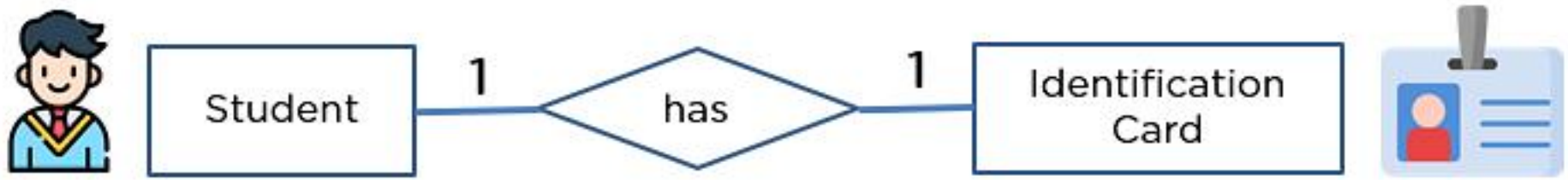


many-to-many (M:N)



one-to-one relationship

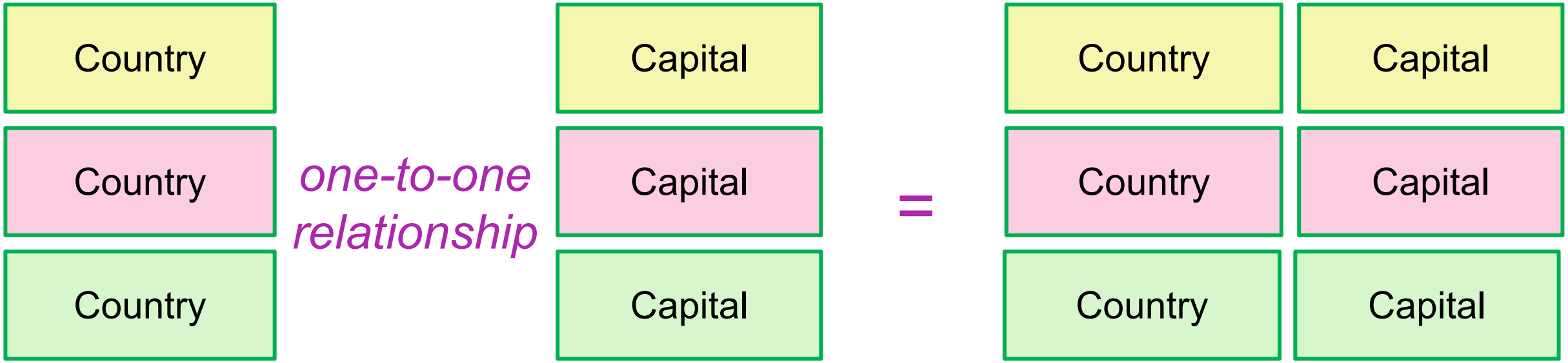




one-to-one relationship

A *one-to-one* relationship between two tables means that a row in one table can only relate to zero/one row in the table on the other side of their relationship. This is the least common database relationship.

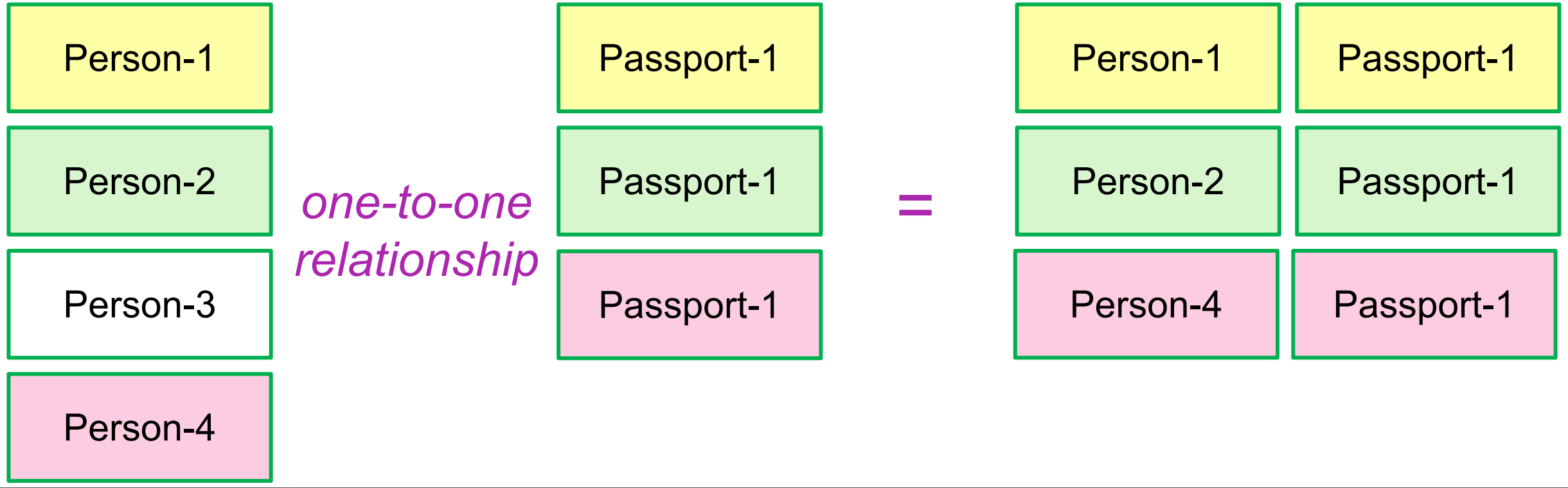
A *one-to-one* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which one element of entity ***R*** may only be linked to zero/one element of entity ***S***, and vice versa.



one-to-one relationship

A *one-to-one* relationship between two tables means that a row in one table can only relate to zero/one row in the table on the other side of their relationship. This is the least common database relationship.

A *one-to-one* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which one element of entity ***R*** may only be linked to zero/one element of entity ***S***, and vice versa.



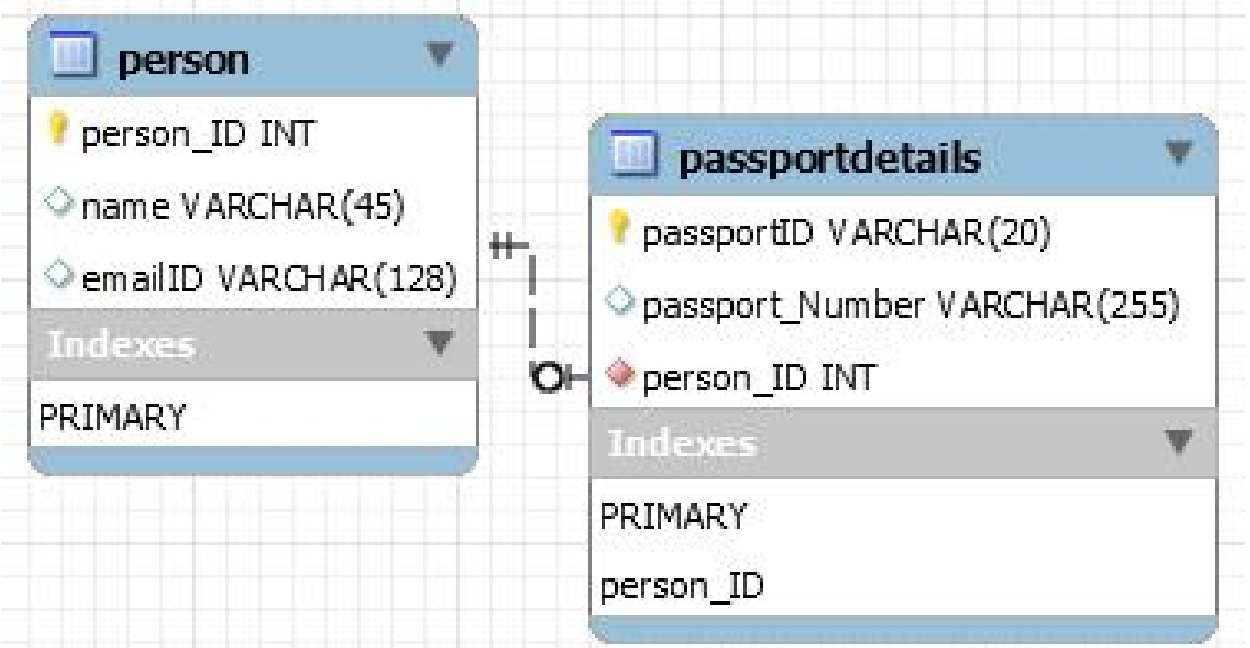
how to create one-to-one relationship

```
CREATE TABLE person (
    person_ID INT PRIMARY KEY ,
    name VARCHAR(45),
    emailID VARCHAR(128)
);
```

person_ID	name	emailID
1	Ramesh	ramesh@gmail.com
2	Rajan	rajan.hotmail.com
3	Kumar	kumer112@yahoomail.com
4	Suraj	suran@gmail.com
NULL	NULL	NULL

```
CREATE TABLE passportDetails (
    passportID VARCHAR(20) PRIMARY KEY,
    passport_Number VARCHAR(255),
    person_ID INT UNIQUE,
    FOREIGN KEY(person_ID) REFERENCES
person(person_ID)
);
```

passportID	passport_Number	person_ID
IN-zx001	IN-XAJS1028S	1
IN-zx002	IN-XBDH1738S	2
IN-zx003	IN-XCKE1933S	3
NULL	NULL	NULL



one-to-many relationship

one-to-many relationship

A *one-to-many* relationship between two tables means that a row in one table can have zero or more row in the table on the other side of their relationship.

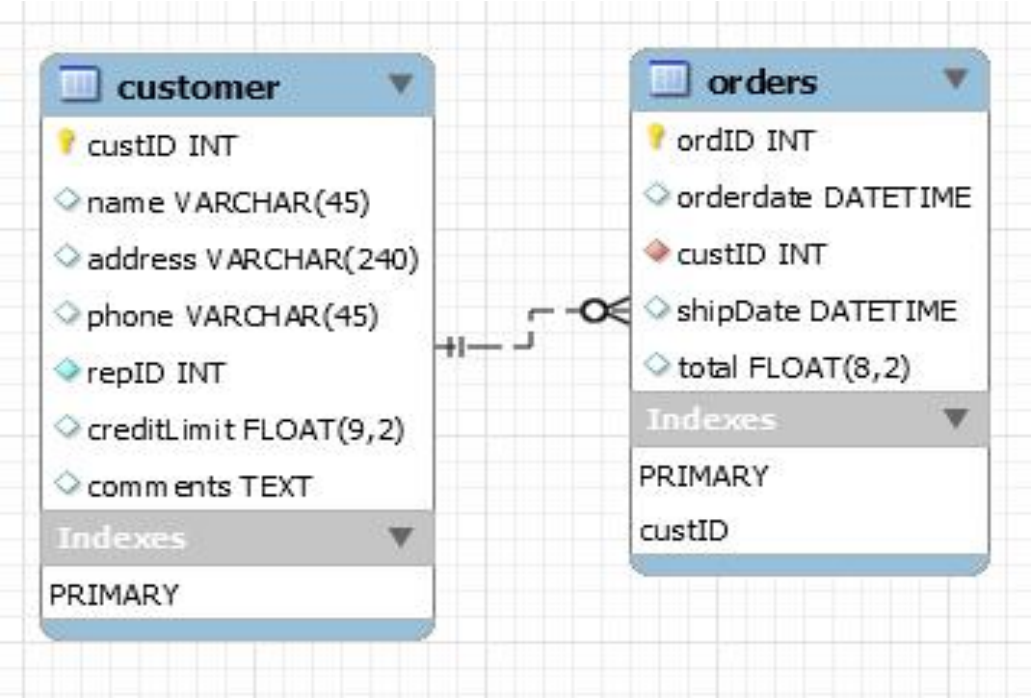
a *one-to-many* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which an element of ***R*** may be linked to many elements of ***S***, but a member of ***S*** is linked to only one element of ***R***.

Customer-1		Order-1	=	Customer-1	Order-1
Customer-2		Order-1		Customer-2	Order-1
Customer-3	*one-to-many relationship*	Order-2		Customer-2	Order-2
Customer-4		Order-1		Customer-3	Order-1
Customer-5		Order-2		Customer-3	Order-2
		Order-3		Customer-3	Order-3
		Order-1		Customer-4	Order-1

how to create one-to-many relationship

```
CREATE TABLE customer (
    custID INT PRIMARY KEY,
    name VARCHAR(45),
    address VARCHAR(240),
    phone VARCHAR(45),
    repID INT NOT NULL,
    creditLimit FLOAT(9,2),
    comments TEXT,
    constraint custid_zero CHECK(custID > 0)
);
```

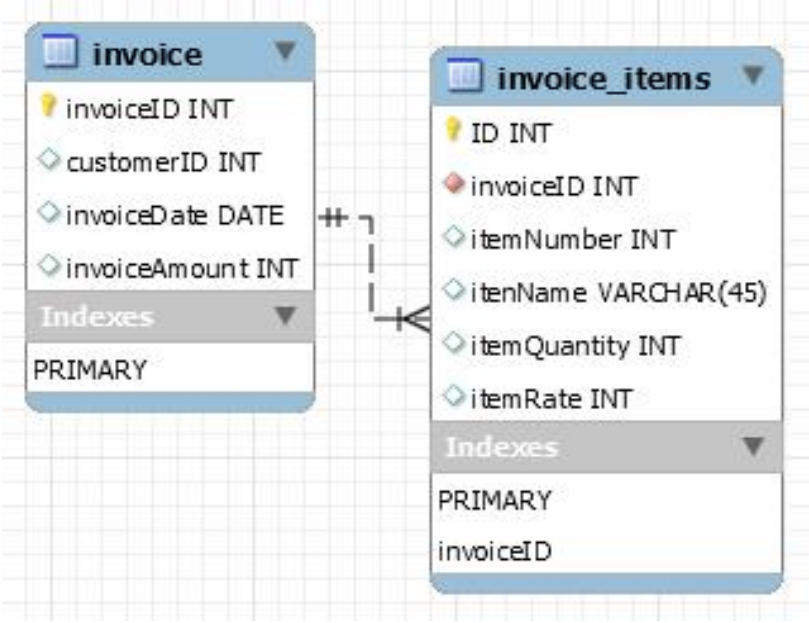
```
CREATE TABLE orders (
    ordID INT PRIMARY KEY,
    orderdate DATETIME,
    custID INT,
    shipDate DATETIME,
    total FLOAT(8,2),
    FOREIGN KEY(custID) REFERENCES customer(custID),
    constraint total_greater_zero CHECK(total >= 0)
);
```



how to create one-to-many relationship

```
CREATE TABLE invoice (
   invoiceID INT PRIMARY KEY,
   customerID INT,
   invoiceDate DATE,
   invoiceAmount INT
);
```

invoiceID	customerID	invoiceDate	invoiceAmount
1	235	2020-01-13	1750
2	235	2020-02-28	5000
3	778	2020-03-10	2000
4	778	2020-03-16	2300
NULL	NULL	NULL	NULL



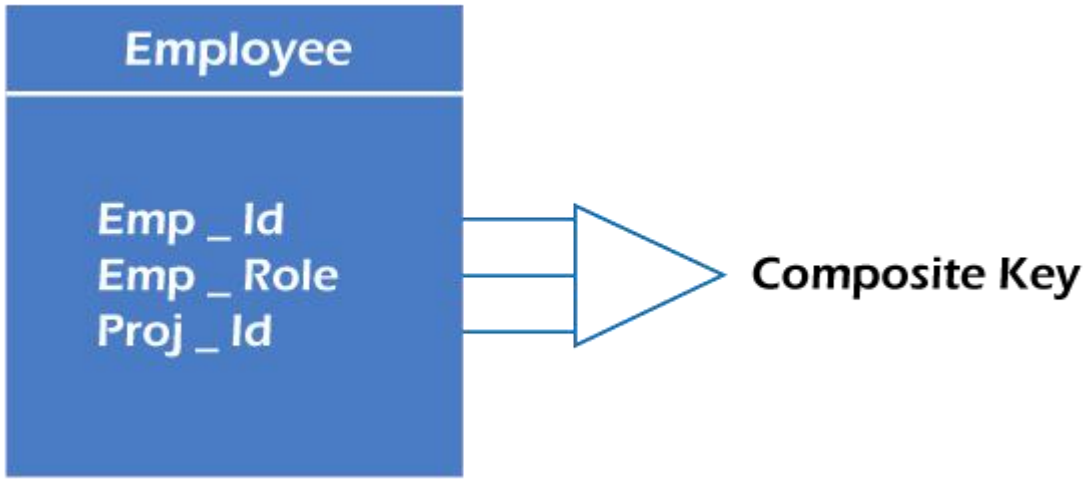

```
CREATE TABLE invoice_items (
   invoiceID INT,
   itemID INT,
   itenName VARCHAR(45),
   itemQuantity INT,
   itemRate INT,
   PRIMARY KEY(invoiceID, itemID),
   FOREIGN KEY(invoiceID) REFERENCES
invoice(invoiceID)
);
```

```
CREATE TABLE invoice_items (
   invoiceID INT NOT NULL,
   itemID INT NOT NULL,
   itenName VARCHAR(45),
   itemQuantity INT,
   itemRate INT,
   UNIQUE(invoiceID, itemID),
   FOREIGN KEY(invoiceID) REFERENCES
invoice(invoiceID)
);
```

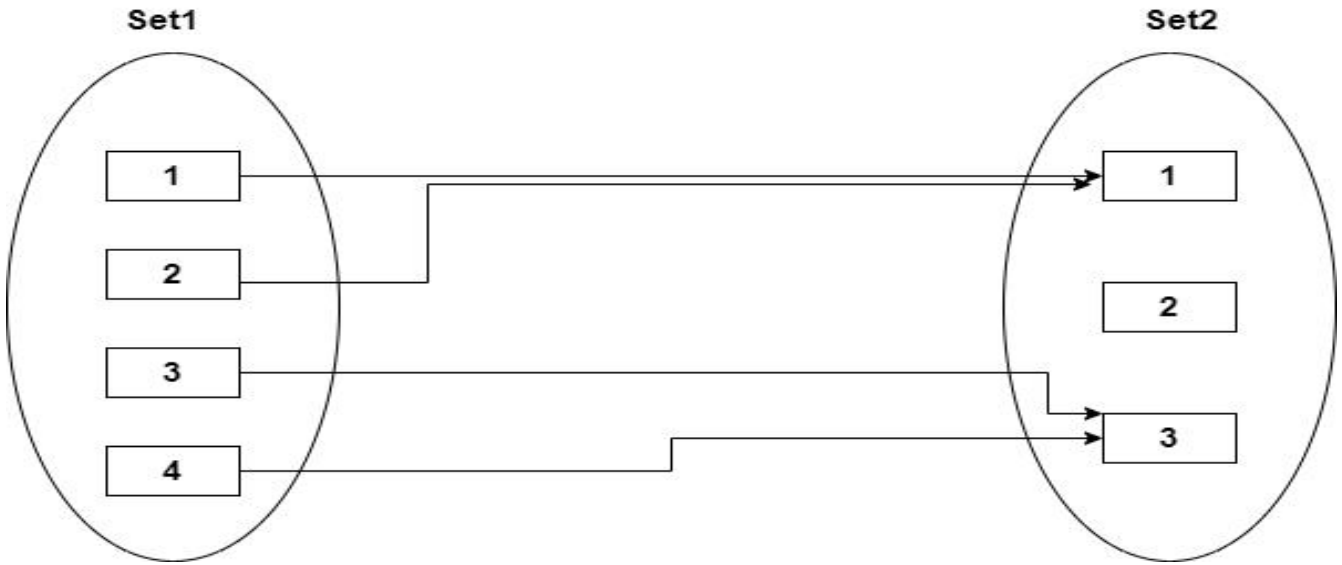
Composite key

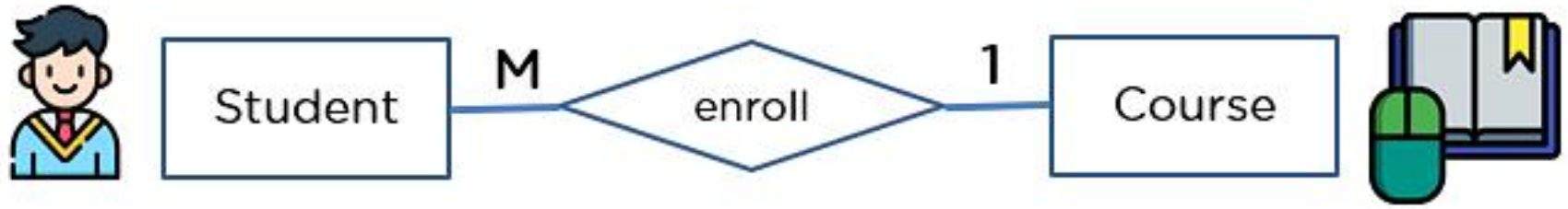
Whenever a primary key consists of more than one attribute, it is known as a composite key. This key is also known as Concatenated Key.

For example,
in employee relations, we assume that an employee may be assigned multiple roles, and an employee may work on multiple projects simultaneously. So the primary key will be composed of all three attributes, namely Emp_ID, Emp_role, and Proj_ID in combination. So these attributes act as a composite key since the primary key comprises more than one attribute.



many-to-one relationship

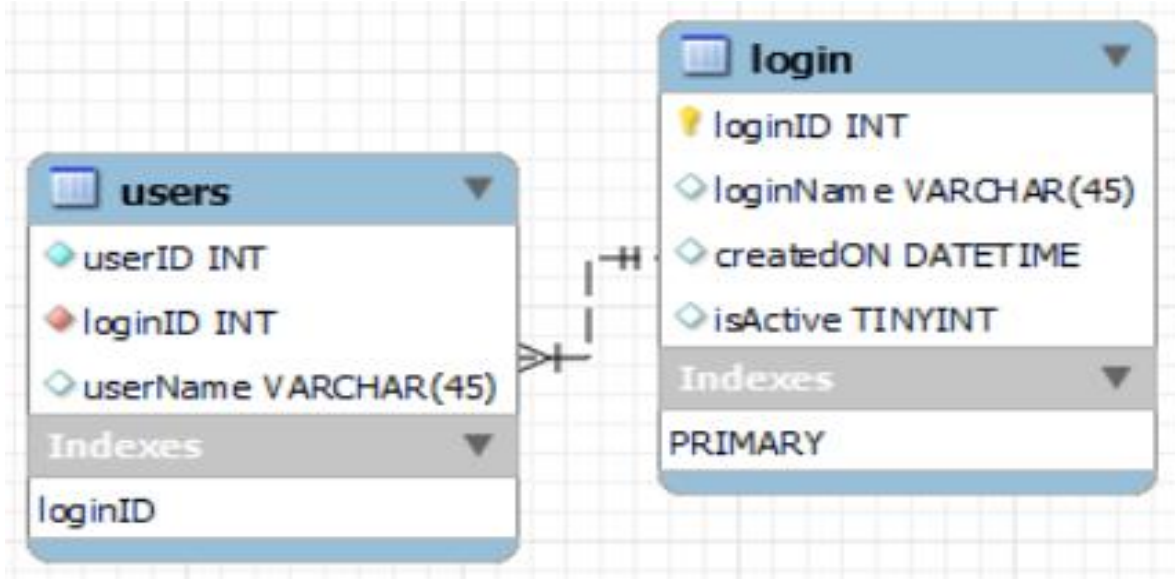




many-to-one relationship

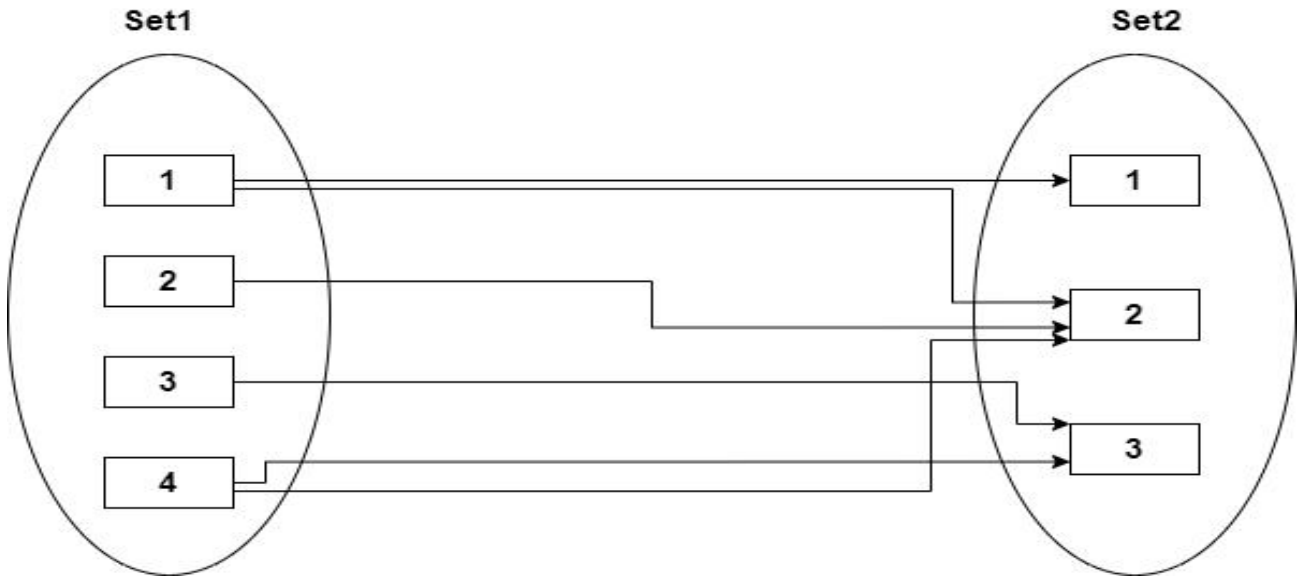
```
CREATE TABLE users (
   userID INT,
   loginID INT,
   userName VARCHAR(45),
   PRIMARY KEY(loginID, userID),
   constraint fk_users_login_loginID1 FOREIGN
KEY(loginID)
   REFERENCES login(loginID)
);
```

```
CREATE TABLE users (
   userID INT NOT NULL,
   loginID INT NOT NULL,
   userName VARCHAR(45),
   UNIQUE(loginID, userID),
   constraint fk_users_login_loginID2 FOREIGN
KEY(loginID)
   REFERENCES login(loginID)
);
```



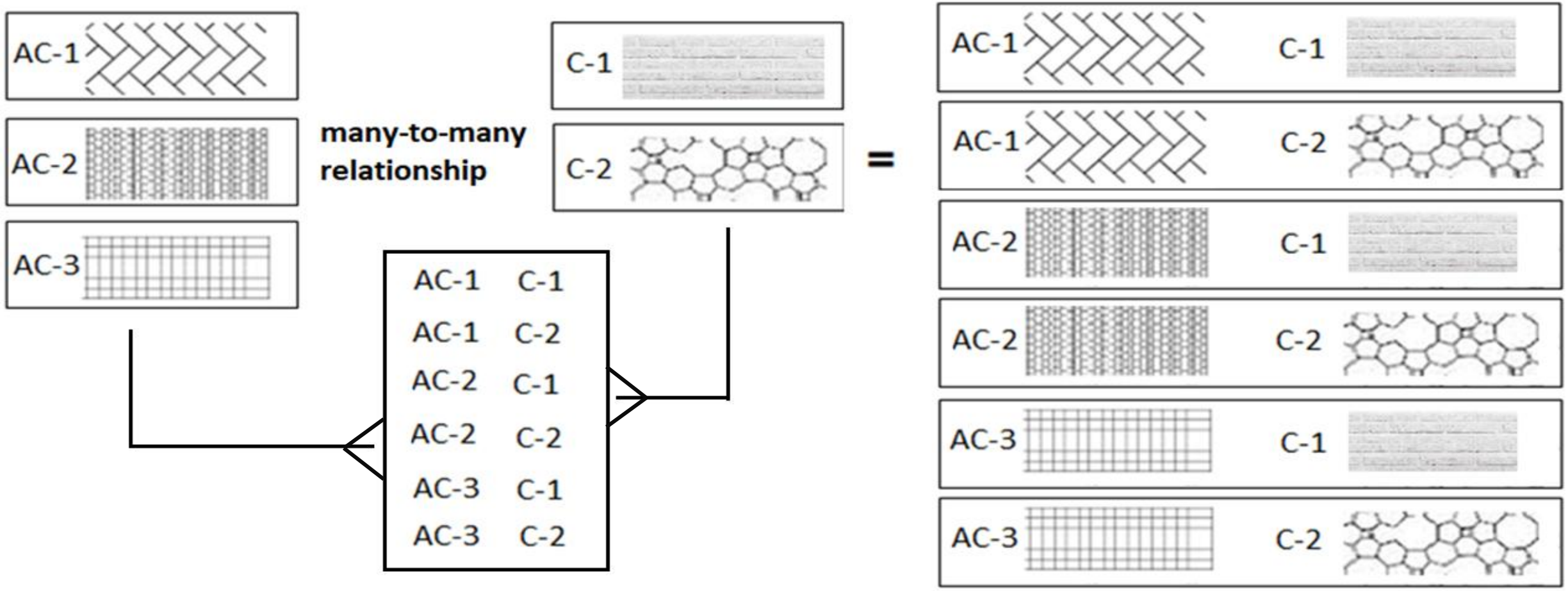

```
CREATE TABLE login (
   loginID INT,
   loginName VARCHAR(45),
   createdON DATETIME,
   isActive TINYINT,
   PRIMARY KEY(loginID)
);
```

many-to-many relationship



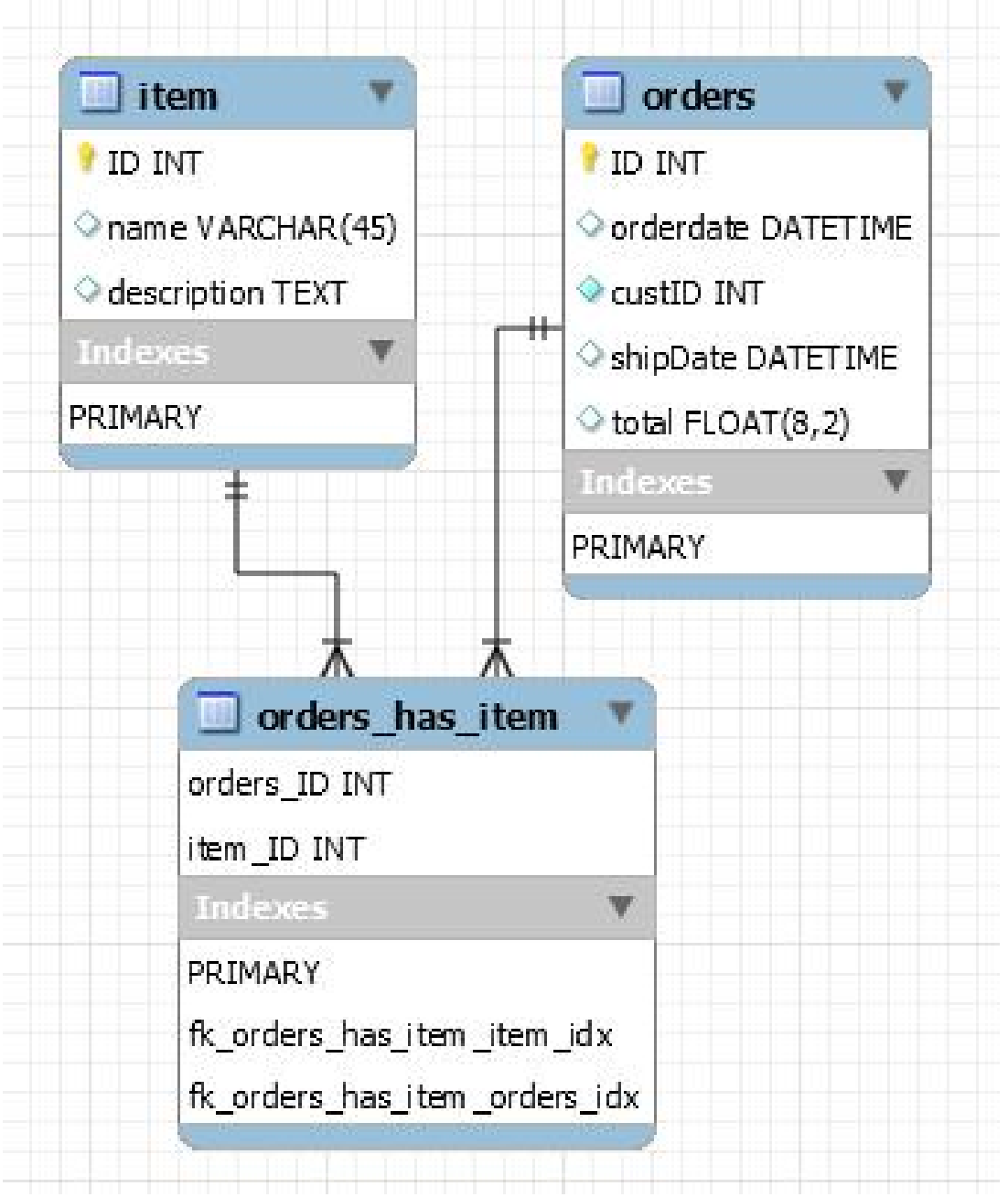
many-to-many relationship

A *many-to-many* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which ***R*** may contain a parent instance for which there are many children in ***S*** and vice versa.



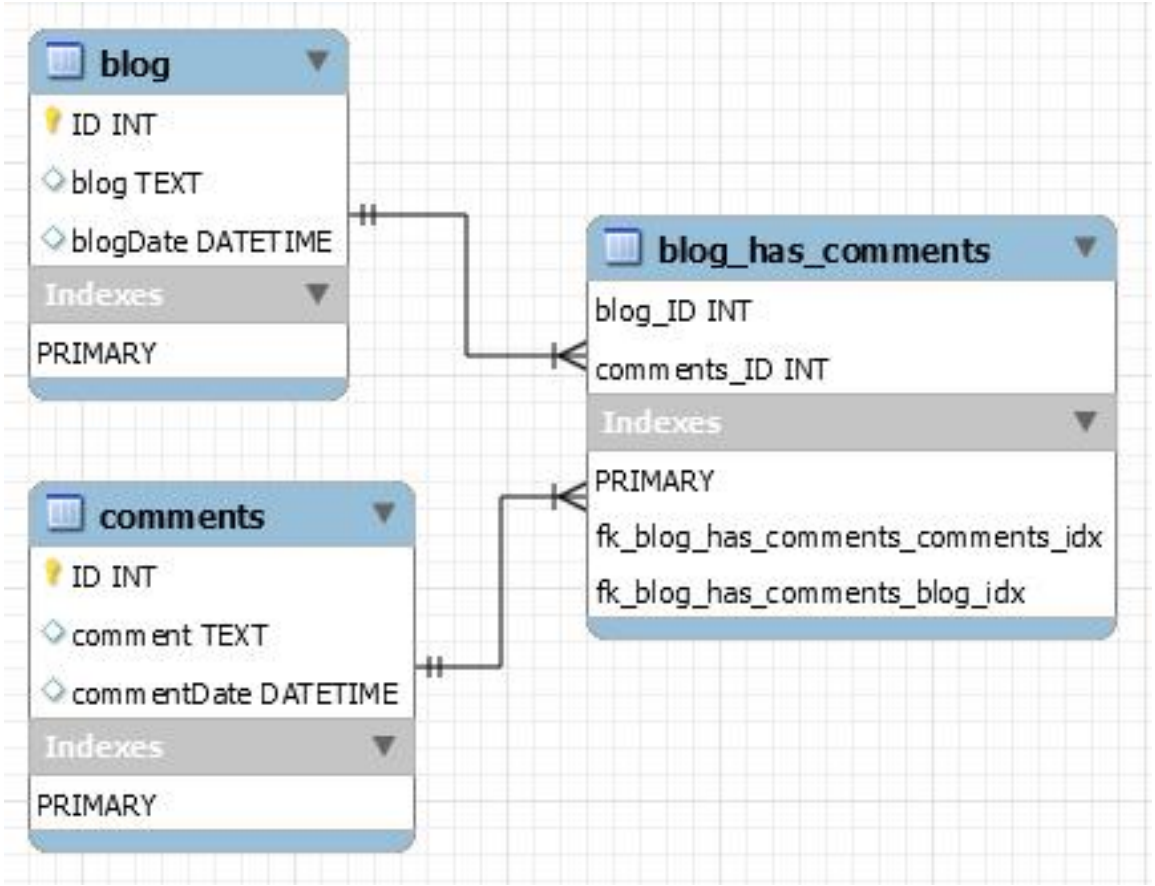
how to create many-to-many relationship

```
CREATE TABLE item (
    ID INT PRIMARY KEY,
    name VARCHAR(45),
    description TEXT
);

CREATE TABLE orders (
    ID INT PRIMARY KEY,
    orderdate DATETIME,
    custID INT NOT NULL,
    shipDate DATETIME,
    total FLOAT(8,2),
    constraint total_greater_zero CHECK(total >= 0)
);

CREATE TABLE orders_has_item (
   orders_ID INT NOT NULL,
   item_ID INT NOT NULL,
   PRIMARY KEY(orders_ID, item_ID),
   constraint fk_orders_has_item_orders FOREIGN
   KEY(orders_ID)
   REFERENCES orders(ID),
   constraint fk_orders_has_item_item1 FOREIGN KEY(item_ID)
   REFERENCES item(ID)
);
```



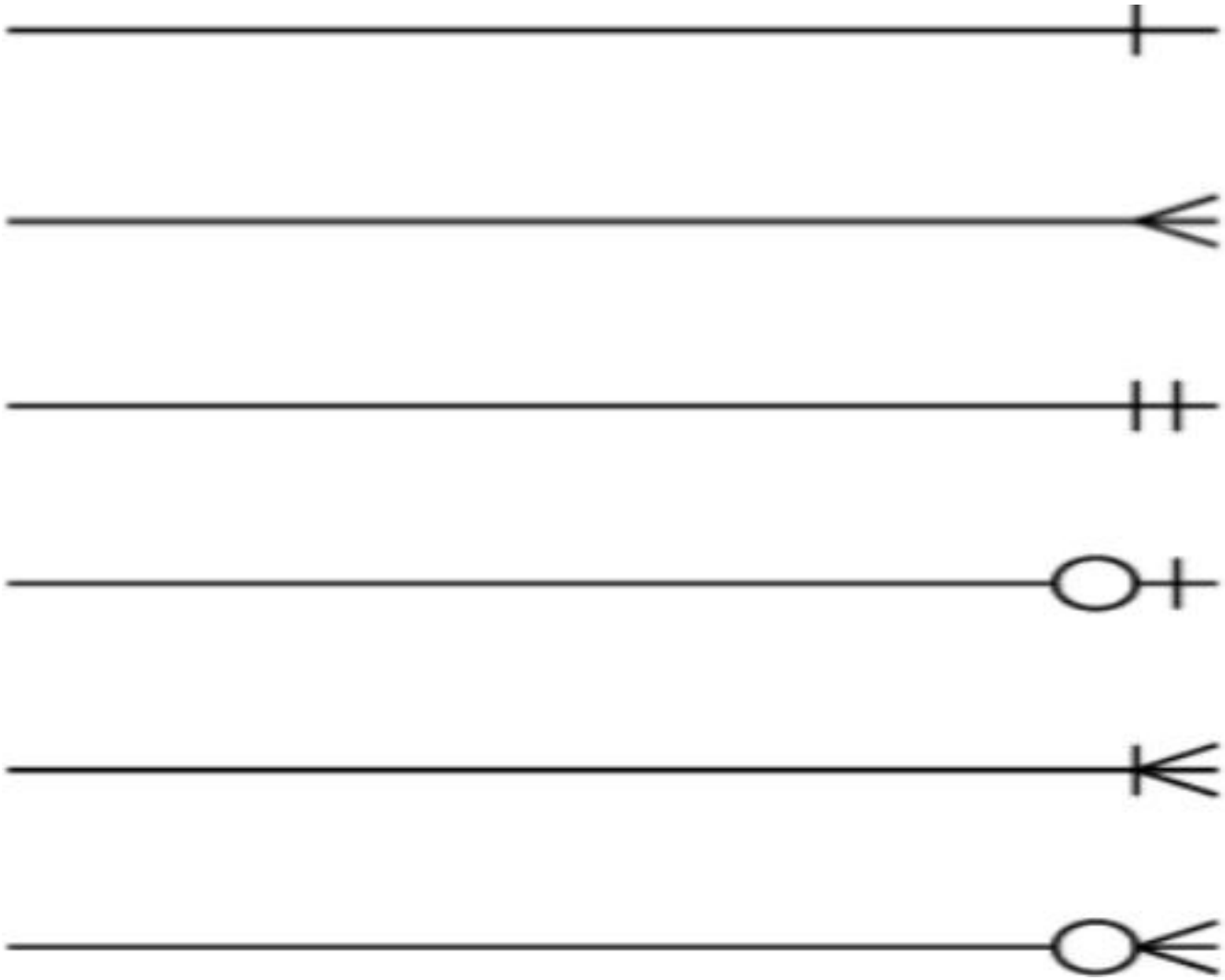
how to create many-to-many relationship

```
CREATE TABLE blog (
   ID INT PRIMARY KEY,
   blog TEXT,
   blogDate DATETIME
);

CREATE TABLE comments (
   ID INT PRIMARY KEY,
   comment TEXT,
   commentDate DATETIME
);

CREATE TABLE blog_has_comments (
    blog_ID INT,
    comments_ID INT,
    PRIMARY KEY(blog_ID, comments_ID),
    constraint fk_blog_has_comments_blog FOREIGN KEY(blog_ID)    REFERENCES blog(ID),
    constraint fk_blog_has_comments_comments FOREIGN KEY(comments_ID) REFERENCES
comments(ID)
);
```

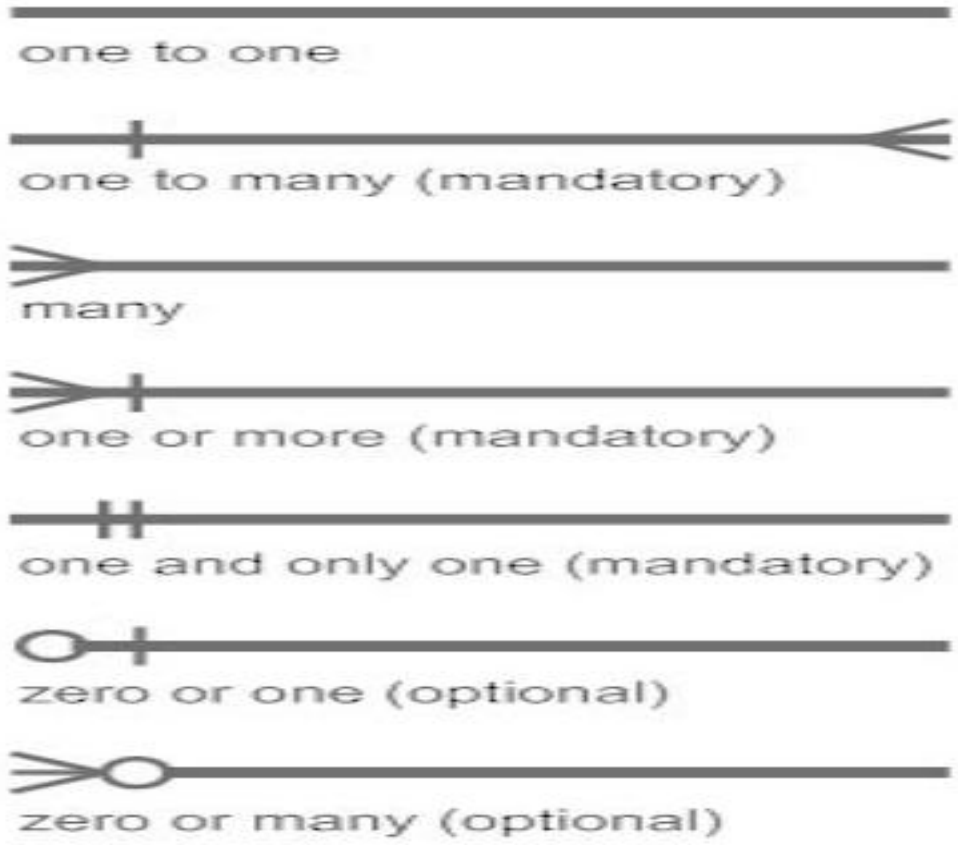


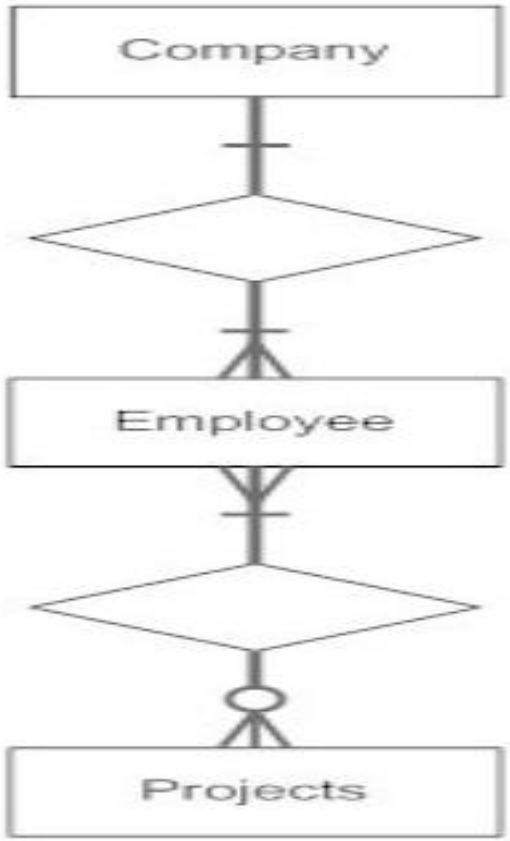
Add Example of many to many relation

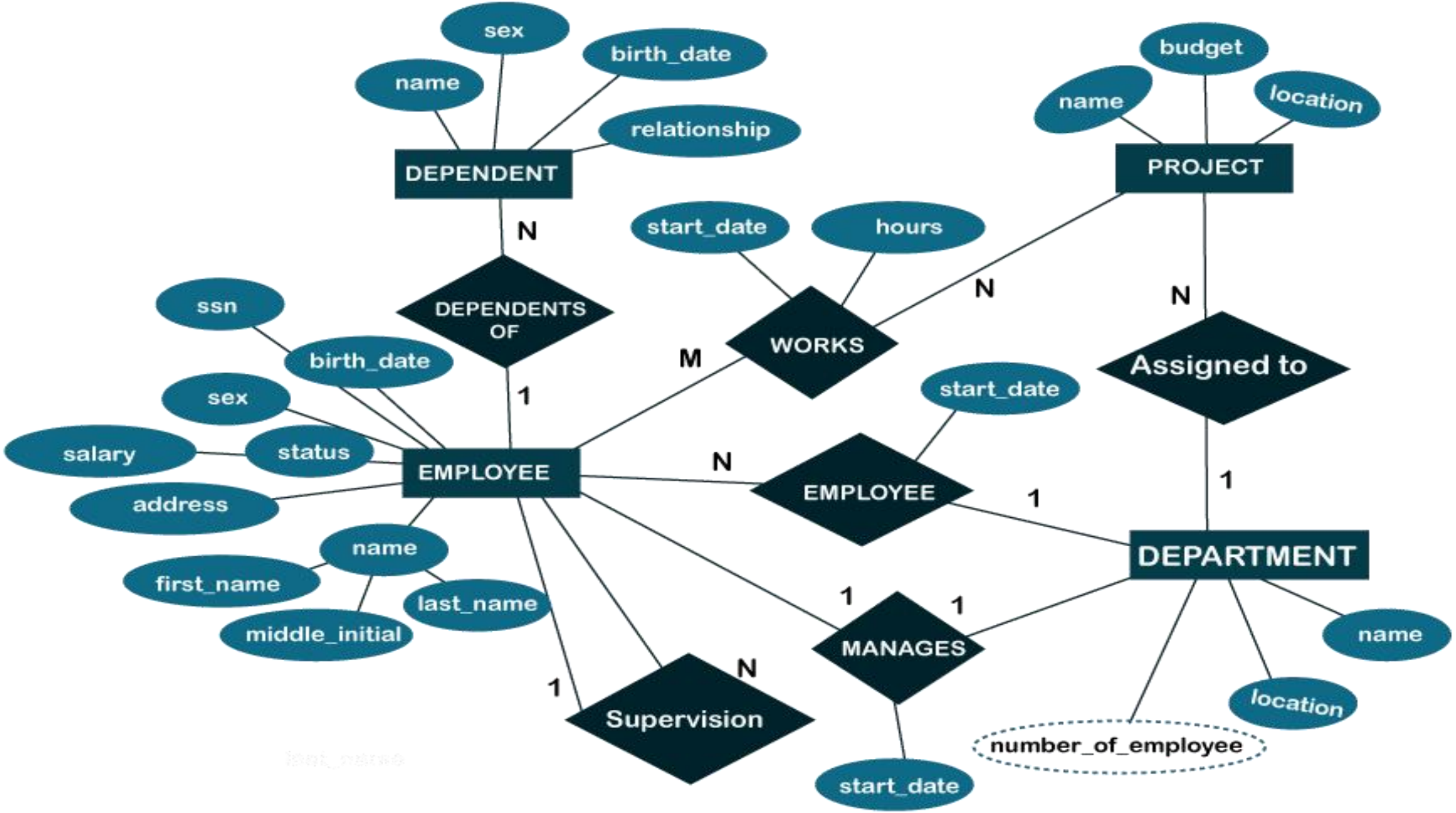
Notation of ER diagram



Notation of ER diagram






sex
birth_date
name
relationship
DEPENDENT
N
DEPENDENTS OF
1
budget
name
location
PROJECT
start_date
hours
N
N
WORKS
M
Assigned to
ssn
birth_date
sex
salary
status
address
EMPLOYEE
start_date
N
EMPLOYEE
1
1
DEPARTMENT
name
first_name
middle_initial
last_name
1
1
MANAGES
name
location
number_of_employee
1
N
Supervision
start_date

IS
NULL
NULL

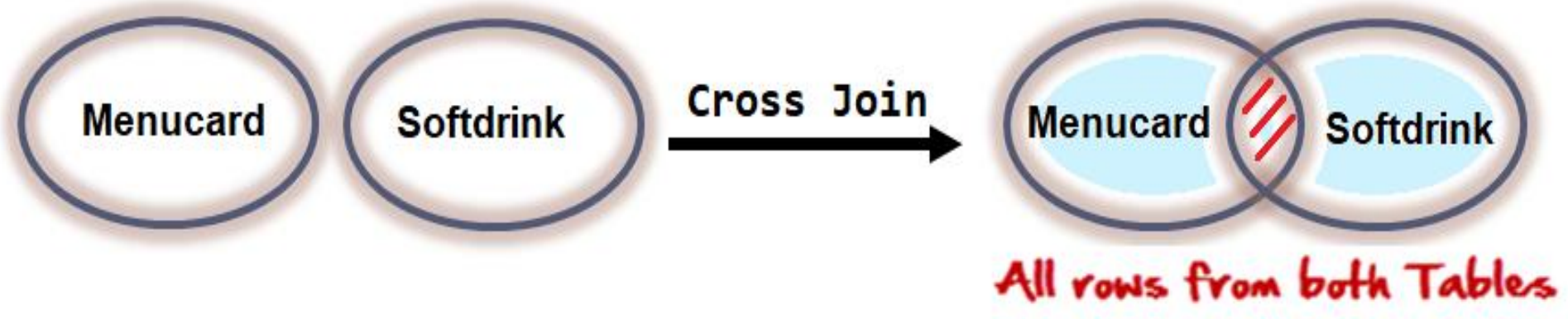
joins

JOINS are used to **retrieve data from multiple tables**.

JOIN is performed whenever **two or more tables** are joined in a SQL statement.

Type of JOINS

- Cartesian or Product Join – Cross Join
- Equijoin – Inner Join
- Natural Join
- Simple Join
- Outer Join – Right Outer Join, Left Outer Join
- Self Join



cartesian or product join

- **Cartesian/Product means** Number of Rows present in Table 1 Multiplied by Number of Rows present in Table 2.
- **Cross Join in MySQL** does not require any common column to **join** two table.

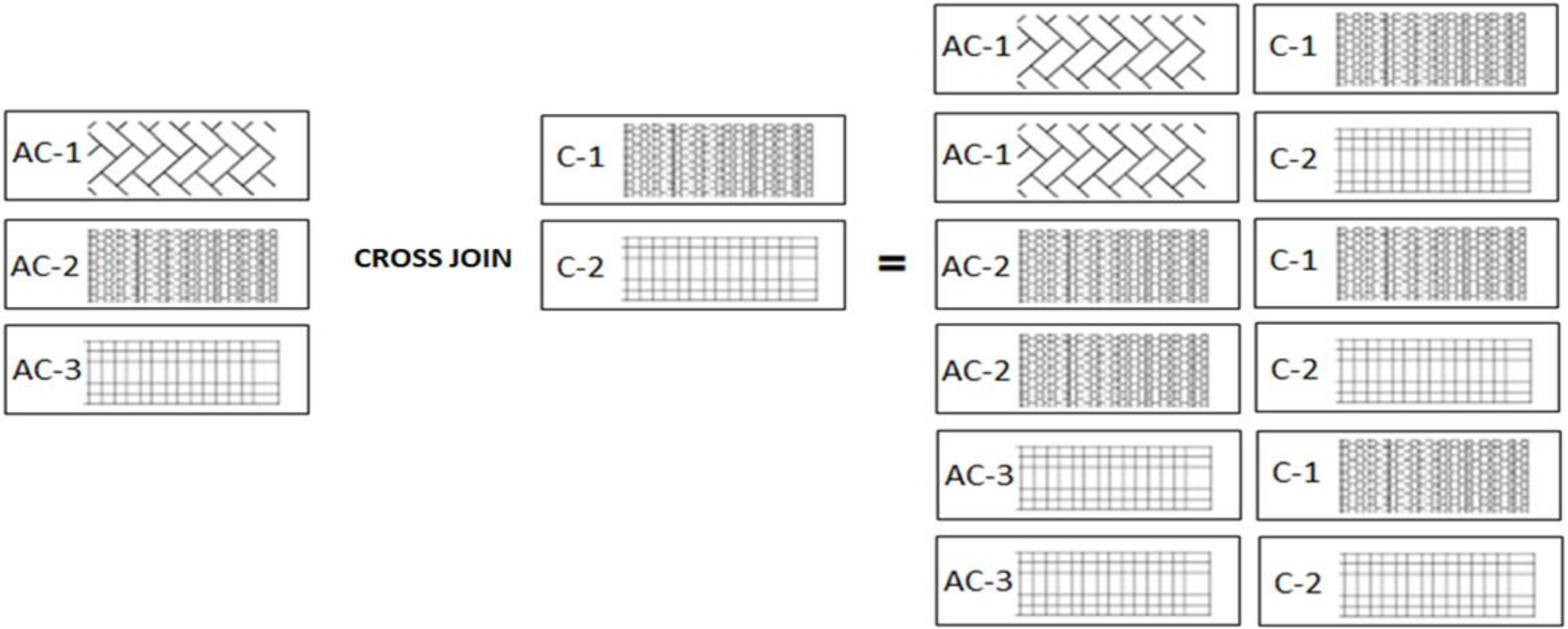
The result of $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m)$ is a relation Q with degree n + m attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, in that order.

Degree $d(R \times S) = d(R) + d(S)$.

Cardinality $|R \times S| = |R| * |S|$

joins - cartesian or product

The CROSS JOIN gets a row from the first table (r_1) and then creates a new row for every row in the second table (r_2). It then does the same for the next row for in the first table (r_1) and so on.



joins – cross join

The CROSS JOIN produced a result set which is the product of rows of two associated tables when no WHERE clause is used with CROSS JOIN. In this join, the result set appeared by multiplying each row of the first table with all rows in the second table if no condition introduced with CROSS JOIN.

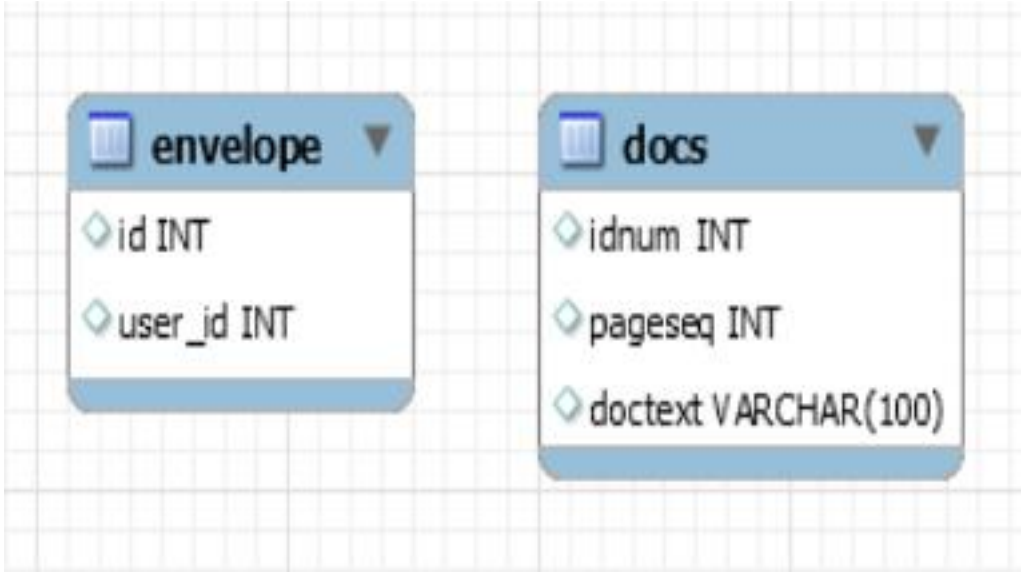
SELECT A_1, A_2, A_3, . . . FROM r_1 CROSS JOIN r_2, . . .

envelope
Table

	id	user_id
▶	1	1
	2	2
	3	3

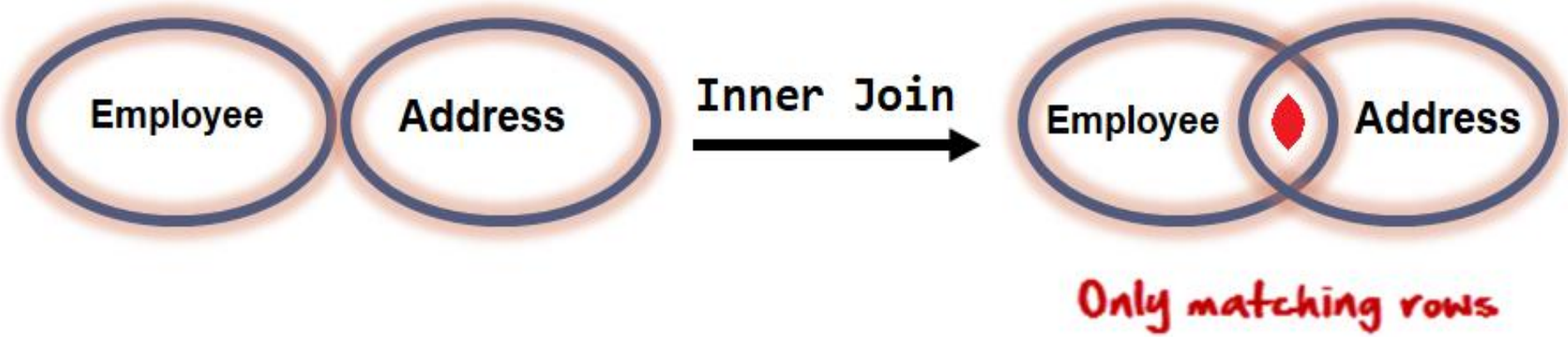
docs
Table

	idnum	pageseq	doctext
▶	1	5	NULL
	2	6	NULL
	NULL	0	NULL



- SELECT * FROM envelope CROSS JOIN docs;

	id	user_id	idnum	pageseq	doctext
▶	1	1	1	5	NULL
	2	2	1	5	NULL
	3	3	1	5	NULL
	1	1	2	6	NULL
	2	2	2	6	NULL
	3	3	2	6	NULL
	1	1	NULL	0	NULL
	2	2	NULL	0	NULL
	3	3	NULL	0	NULL



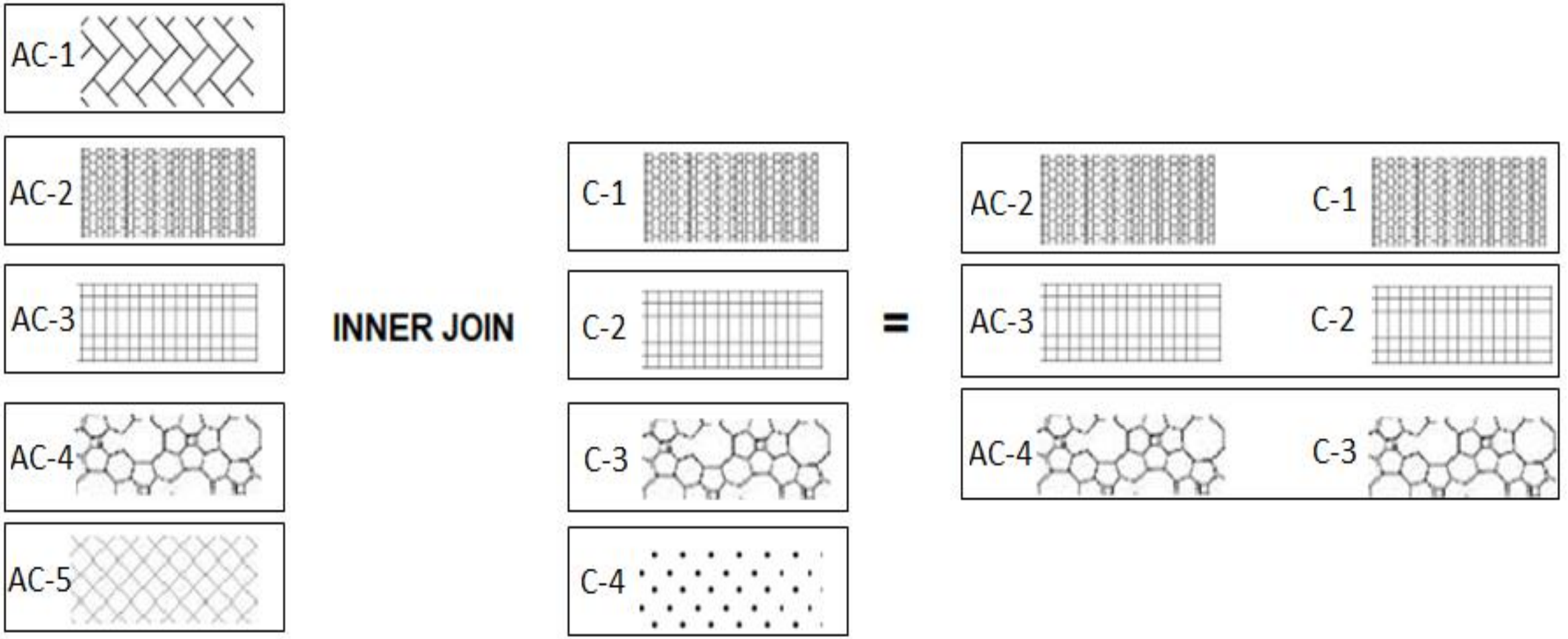
equi join

An **equi join / Inner Join** is a join with a join condition containing an equality operator.
An equijoin returns only those rows that have equivalent values for the specified columns. Rows that match remain in the result, those that don't are rejected. The match condition is commonly called the **join condition**. **equi join / Inner Join** returns rows when there is at least one match in both tables.

The result of $R(A_1, A_2, \ldots, A_n) \bowtie_{\text{<join condition>}} S(B_1, B_2, \ldots, B_m)$ is a relation Q with degree **n + m** attributes **$Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, in that order**. Q has one tuple for each combination of tuples—one from R and one from S—whenever the combination satisfies the join condition.

equi join example

The following table illustrates the inner join of two tables r_1(AC-1, AC-2, AC-3, AC-4, AC-5) and r_2(C-1, C-2, C-3, C-4). The result includes rows: (2,A), (3,B), and (4,C) as they have the same patterns.



inner join

The inner join is one of the most commonly used joins in SQL. The inner join clause allows you to query data from two or more related tables.

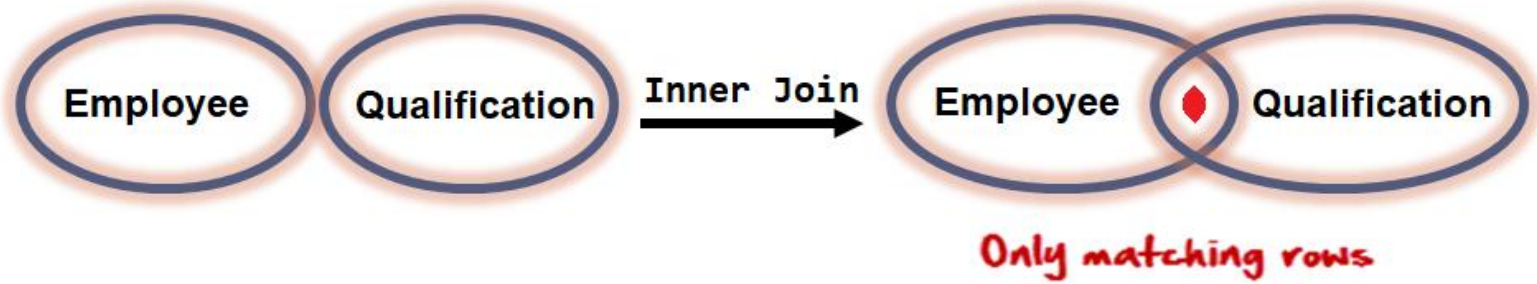
joins – inner join

INNER JOIN returns rows when there is at least one match in both tables.

The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

SELECT A_1, A_2, A_3, . . . FROM r_1 [INNER] JOIN r_2 ON r_1. A_1 = r_2. A_1

- SELECT * FROM employee emp INNER JOIN qualification quali ON emp.id = quali.employeeid;



natural join

Joins two tables based on common column names. Hence one must confirm the common columns before using a NATURAL JOIN

joins – natural join

The NATURAL JOIN is such a join that performs the same task as an INNER JOIN.

SELECT A_1, A_2, A_3, . . . FROM r_1 NATURAL [INNER] JOIN r_2 NATURAL [INNER] JOIN r_3 . . .

- SELECT * FROM emp NATURAL JOIN dept;

- The associated tables have one or more pairs of identically column-names.
- The columns must be of the same name.
- The columns datatype may differ.
- Don't use ON / USING clause in a NATURAL JOIN.
- When this join condition gets applied always the duplicates of the common columns get eliminated from the result.

A **NATURAL JOIN** can be used with a **LEFT OUTER** join, or a **RIGHT OUTER** join.

If the column-names are not same, then NATURAL JOIN will work as CROSS JOIN.

SELECT * FROM EMP
NATURAL JOIN DEPT

simple join

TODO

joins – simple join

The SIMPLE JOIN is such a join that performs the same task as an INNER JOIN.

SELECT A_1, A_2, A_3, . . . FROM r_1 SIMPLE JOIN r_2 USING (A_1, . . .)

- SELECT * FROM emp SIMPLE JOIN dept USING(deptno)

outer joins

In an outer join, along with rows that satisfy the matching criteria, we also include some or all rows that do not match the criteria.

joins – left outer join

The LEFT JOIN keyword returns all rows from the left table (r_1), with the matching rows in the right table (r_2).
The result is NULL in the right side when there is no match.

SELECT A_1, A_2, A_3, . . . FROM r_1 LEFT [OUTER] JOIN r_2 ON r_1. A_1 = r_2. A_1

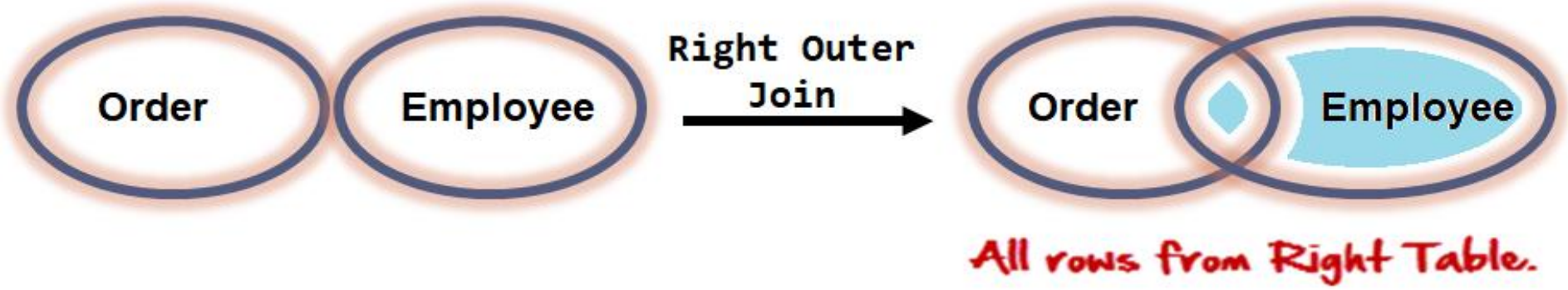
```
SELECT * FROM orders ord LEFT OUTER JOIN employee emp ON emp.id = ord.employeeid;
```

joins – right outer join

The RIGHT JOIN keyword returns all rows from the right table (r_2), with the matching rows in the left table (r_1).
The result is NULL in the left side table when there is no match.

SELECT A_1, A_2, A_3, . . . FROM r_1 RIGHT [OUTER] JOIN r_2 ON r_1. A_1 = r_2. A_1

```
SELECT * FROM orders ord RIGHT OUTER JOIN employee emp ON emp.id = ord.employeeid;
```



TODO

self joins

TODO

joins – self join

A SELF JOIN is a join in which a table is joined with itself (which is also called Unary relationships), especially when the table has a FOREIGN KEY which references its own PRIMARY KEY.

SELECT $r_x.A_1$, $r_x.A_2$, $r_y.A_1$, $r_y.A_2$, . . . FROM r_1 r_x, r_1 r_y WHERE $r_x.A_1 = r_y.A_1$

set operation in sql

Set operators are used to join the results of two (or more) SELECT statements.

Remember:

- The result set column names are taken from the column names of the first SELECT statement.
- SELECT statement should have the same data type. (Not in MySQL)

syntax

SELECT . . . UNION [ALL]
SELECT . . .

- SELECT * FROM books UNION SELECT * FROM newbooks;
- SELECT * FROM books UNION ALL SELECT * FROM newbooks;

syntax

SELECT . . . INTERSECT
SELECT . . .

- SELECT * FROM books INTERSECT SELECT * FROM newbooks;

syntax

SELECT . . . EXCEPT
SELECT . . .

EXCEPT returns rows from first dataset, which are not available in the second dataset.

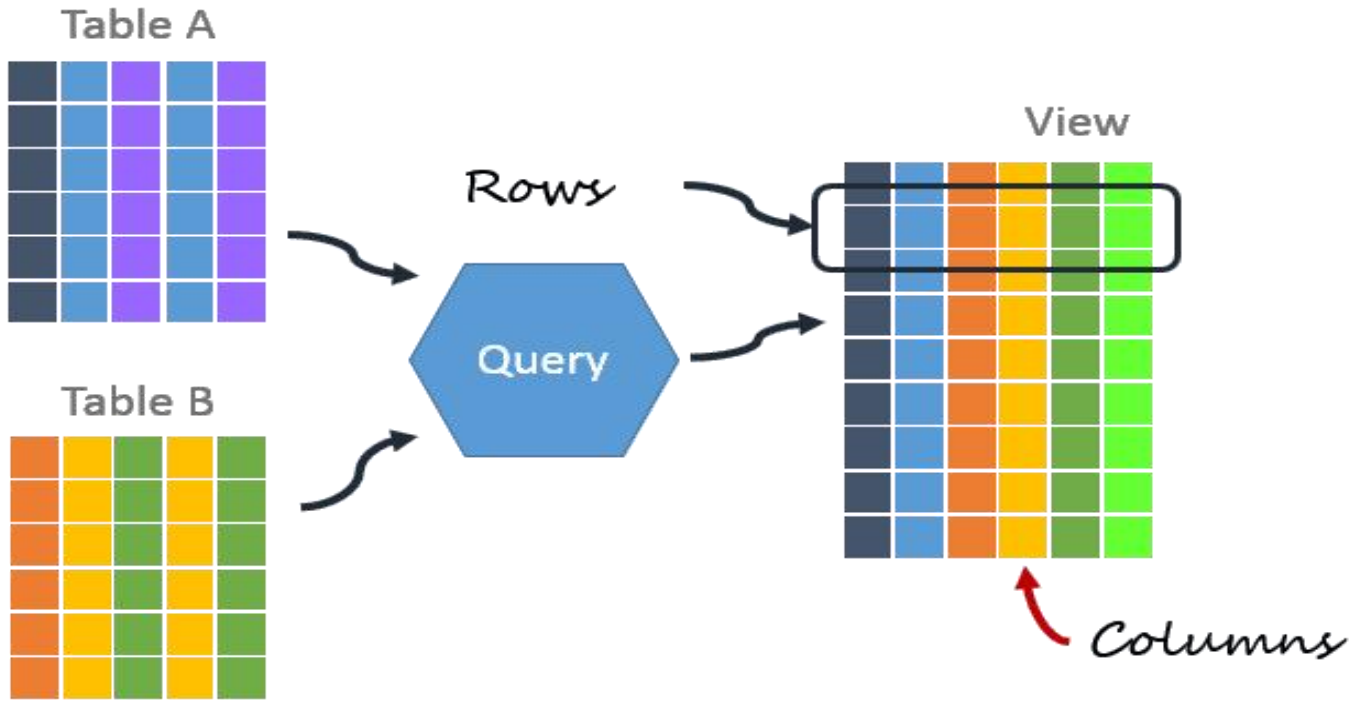
- SELECT * FROM books EXCEPT SELECT * FROM newbooks;

A ***VIEW*** in SQL as a logical subset of data from one or more tables. Views are used to restrict data access. A ***VIEW*** contains no data of its own but its like window through which data from tables can be viewed or changed. The table on which a View is based are called BASE Tables.

There are 2 types of Views in SQL:

- **Simple View** : Simple views can only contain a single base table.
- **Complex View** : Complex views can be constructed on more than one base table. In particular, complex views can contain: join conditions, a group by clause, a order by clause.

views



Views are not updatable in the following cases:

- A table in the FROM clause is reference by a subquery in the WHERE statement.
- There is a subquery in the SELECT clause.
- The SQL statement defining the view joins tables.
- One of the tables in the FROM clause is a non-updatable view.
- The SELECT statement of the view contains an aggregate function such as SUM(), COUNT(), MAX(), MIN(), and so on.
- The keywords DISTINCT, GROUP BY, HAVING clause, LIMIT clause, UNION, or UNION ALL appear in the defining SQL statement.

create view/ show create view

The select_statement is a SELECT statement that provides the definition of the view. The select_statement can select from base tables or other views.

```
CREATE [OR REPLACE]  VIEW view_name [(column_list)]
    AS select_statement  [WITH CHECK OPTION]
```

```
SHOW CREATE VIEW view_name
```

```
show create VIEW v1;
```

alter / drop view

This statement changes the definition of a view, which must exist.

```
ALTER VIEW view_name [(column_list)]
    AS select_statement
    [WITH CHECK OPTION]
```

e.g.

- ALTER VIEW studentview AS SELECT namefirst, namelast, emailid FROM student;

DROP VIEW removes one or more views.

```
DROP VIEW [IF EXISTS]
    view_name [, view_name] ...
```

e.g.

- DROP VIEW studentview;
- DROP VIEW studentid10view, studentviewwithcheck;
- DROP VIEW studentTotalMarksView, studentAddressView;