DATABASE MANAGEMENT SYSTEMS
LOGICAL DATABASE DESIGN

ER Diagram to Relational Model Conversion

The ER model is convenient for representing an initial, high-level database design. First step of any relational database design is to make ER Diagram for it and then convert it into relational Model.

Given an ER diagram describing a database, there is a standard approach to generating a relational database schema that closely approximates the ER design.

What is relational model?

Relational Model represents how data is stored in database in the form of table

.

Lets learn step by step how to convert ER diagram into relational model.

Logical database design is the process of mapping or translating the conceptual model (ER Diagrams) into a relational model (table). i.e. deciding how to arrange the attributes of the entities in a given business environment into database structures, such as the tables of a relational database.

Let us see how to translate an ER diagram into a collection of tables with associated constraints, i.e., a relational database schema.

Three basic rules to convert ER into tables or relations:

Rule 1: *Entity:* Entity Names will automatically be table names

Rule 2: *Mapping of attributes*: attributes will be columns of the respective tables.

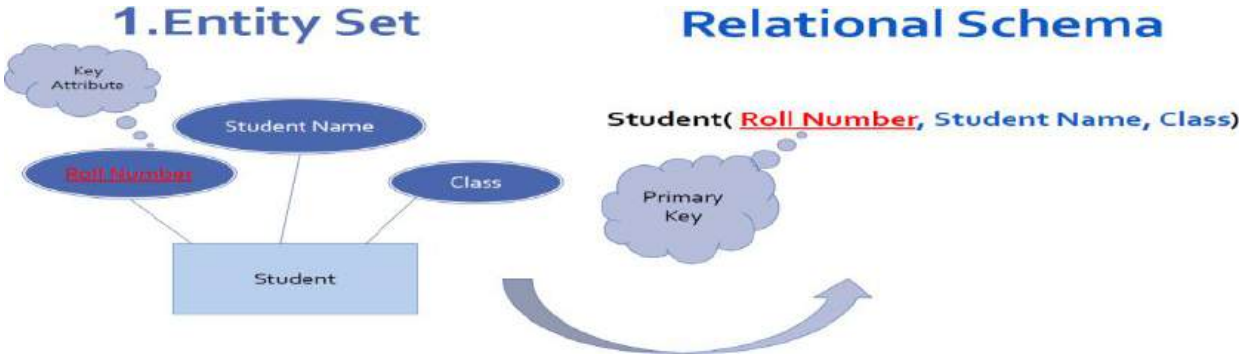
Rule 3: *Relationships*: relationship will be mapped by using a foreign key attribute. Foreign key is a primary or candidate key of one relation used to create association between tables.

1. Entity Set to Table:

Consider we have entity STUDENT in ER diagram with attributes Roll Number, Student Name and Class.

<u>To convert this entity set into relational schema</u>

1. Entity is mapped as relation in Relational schema
2. Attributes of Entity set are mapped as columns for that Relation.
3. Key attribute of Entity becomes Primary key for that Relation.



2. Entity Set with Multivalued Attributes:

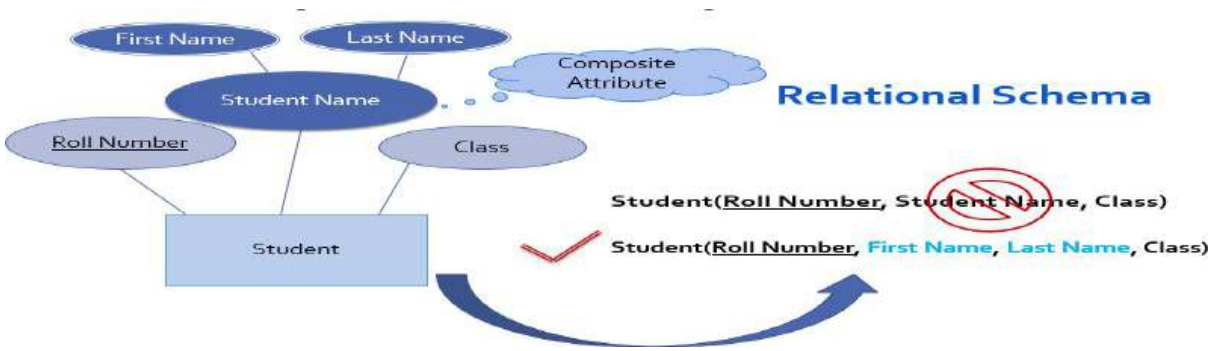
An entity may have several attributes which can take more than one value (eg: mobile number, hobby etc.). As it is not possible to represent multiple values in a single column in a relation, separate relation is created for multivalued attribute along with the key attribute of the relation.

1. Key attribute and multivalued attribute of entity set becomes composite key of new relation.
2. Separate relation employee is created with remaining attributes.

   This ensures instead of repeating all attributes of entity (for every value of the multivalued attribute) now only one attribute is need to repeat and that is the key attribute.

3. Entity set with Composite attribute:

Consider entity set student with attributes Roll Number, Student Name and Class. Here student name is composite attribute as it has further divided into First name, last name.
In this case to convert entity into relational schema,
composite attribute student name should not be include in relation but all parts of composite attribute are mapped as simple attributes for relation.



Relationship Sets (without Constraints) to Tables

A relationship set, like an entity set, is mapped to a relation in the relational model.

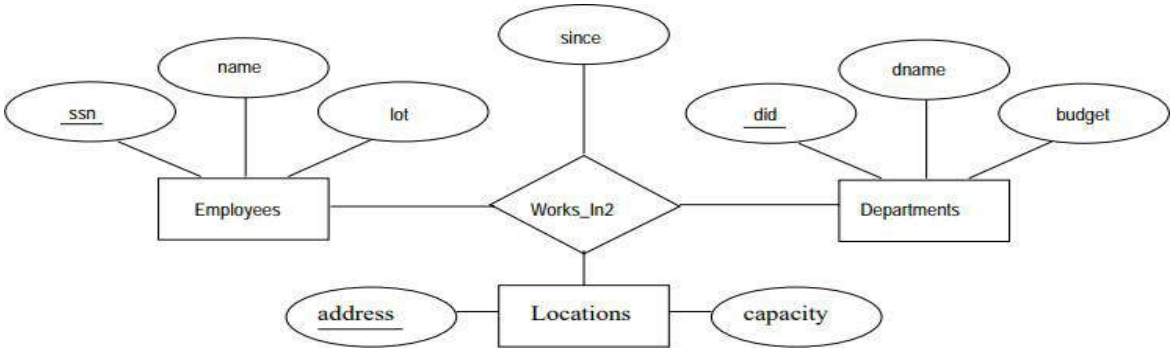
To represent a relationship, we must be able to identify each participating entity and give values to the descriptive attributes of the relationship.

Thus, the attributes of the relation include:

The primary key attributes of each participating entity set, as foreign key fields.

The descriptive attributes of the relationship set.

Example 1: Consider the ERD in the following figure. All the available information about the Works In2 table is captured by the following SQL definition:

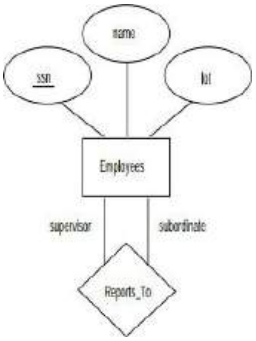


Create Tables for the participating entities Employee and Department with their respective attributes.

Create the table for the **Works_In2** relationship set as below

CREATE TABLE Works_In2 (ssn CHAR(11), did INTEGER, address CHAR(20), since DATE,
PRIMARY KEY (ssn, did, address),
FOREIGN KEY (ssn) REFERENCES Employees,
FOREIGN KEY (address) REFERENCES Locations,
FOREIGN KEY (did) REFERENCES Departments);

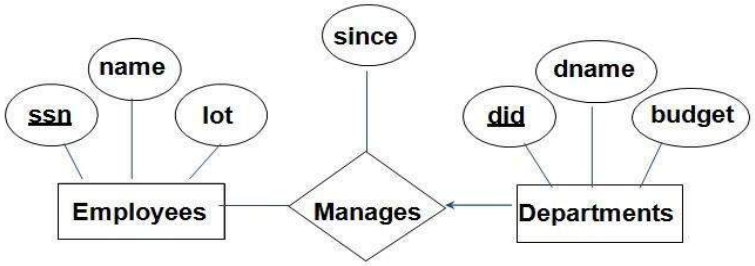
Example 2:
Consider another example:
Here the role indicators (supervisor/subordinate) are created as columns



CREATE TABLE Reports To (
Supervisor_ssn CHAR(11), subordinate_ssn CHAR(11),
PRIMARY KEY (supervisor ssn, subordinate ssn),
FOREIGN KEY (supervisor ssn) REFERENCES Employees(ssn),
FOREIGN KEY (subordinate ssn) REFERENCES Employees(ssn))

Observe that we need to explicitly name the referenced field of Employees because the field name differs from the name(s) of the referring field(s).

Relationship Sets (with Key Constraints) to Tables



Because each department has at most one manager, no two tuples can have the same *did* value but differ on the *ssn* value.

A consequence of this observation is that *did* is itself a key for Manages

The Manages relation can be defined using the following SQL statement:

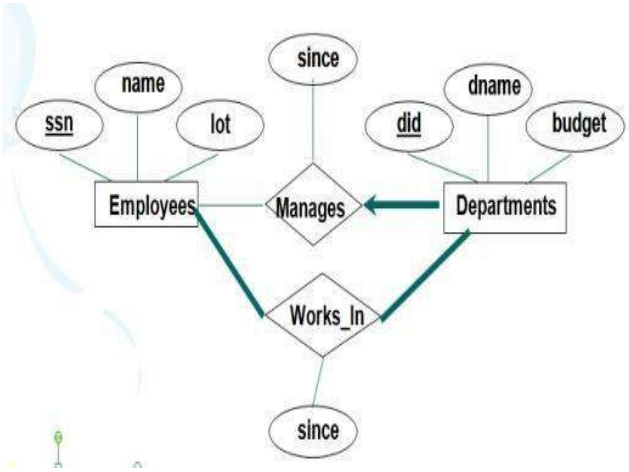
CREATE TABLE Manages(
ssn CHAR(11), did INTEGER, since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn) REFERENCES Employees (ssn),
FOREIGN KEY (did) REFERENCES Departments(did));

- **A second approach:** Since each department has a unique manager, we could instead combine Manages and Departments.
- This approach eliminates the need for a separate Manages relation, and queries asking for a department's manager can be answered without combining information from two relations.
- The only drawback to this approach is that space could be wasted if several departments have no managers.
- In this case the added fields would have to be filled with *null* values.

The Dept_Mgr relation can be defined using the following SQL statement:

CREATE TABLE Dept_Mgr (
did INTEGER, dname CHAR(20), budget REAL, ssn CHAR(11), since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn) REFERENCES Employees(ssn))

Translating Relationship Sets with Participation Constraints:
Total participation constraint insists that every department is required to have a manager, and key constraint says every department will have at most one manager. Participation constraint is implemented using NOT NULL constraint



CREATE TABLE
Dept_Mgr(did INTEGER,
dname CHAR(20),
budget REAL,
ssn CHAR(11) NOT NULL,
since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn) REFERENCES Employees(ssn),
ON DELETE NO ACTION));

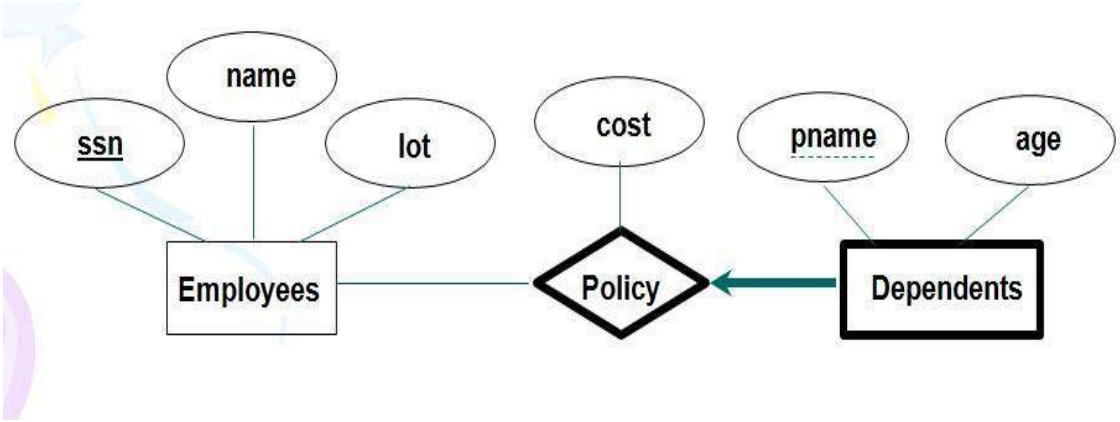
- It also captures the participation constraint that every department must have a manager:
- Because *ssn* cannot take on *null* values, each tuple of Dept_Mgr identifies a tuple in Employees (who is the manager).
- The NO ACTION specification, which is the default and need not be explicitly specified, ensures that an Employees tuple cannot be deleted while it is pointed to by a Dept_Mgr tuple.
- If we wish to delete such an Employees tuple, we must first change the Dept_Mgr tuple to have a new employee as manager.

Translating a weak entity set:

A weak entity set always participates in a one-to-many binary relationship and has a key constraint and total participation.

A Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity.

The Dependents entity must be deleted if the owning Employees entity is deleted.



Weak entity set and identifying relationship set are translated into a single table.

When the owner entity is deleted, all owned weak entities must also be deleted.

```
CREATE TABLE Dep_Policy
( Pname       CHAR(20),
age      INTEGER,
cost     REAL,
ssn      CHAR(11) NOT NULL,
PRIMARY KEY  (pname, ssn),
FOREIGN KEY (ssn) REFERENCES Employees(ssn), ON DELETE CASCADE) )
```

Translating ER Diagrams with Aggregation:

- For the ER diagram shown in Figure, the Employees, Projects, and Departments entity sets and the Sponsors relationship set are mapped as explained before.
- For the **Monitors relationship set**, we create a relation with the following attributes: *the key attributes of Employees (ssn), the key attributes of Sponsors (did, pid), and the descriptive attributes of Monitors (until).*

DATABASE MANAGEMENT SYSTEMS
TYPES OF KEYS

A key in a Database Management System (DBMS) is a set of one or more attributes that uniquely identifies a record within a table. Keys are crucial in ensuring that each record within a table is distinct and they also help in establishing relationships between tables.

Keys are the foundation of data integrity and play a pivotal role in defining how data is stored, accessed, and related across tables in a database.

They ensure that relationships between tables are logical, reliable, and enforce the rules that make databases accurate, organized, and meaningful.

Types of Keys in the Relational Model

- Primary Key
- Candidate Key
- Super Key
- Foreign Key
- Composite Key
- Alternate Key
- Unique Key

Primary Key:

A primary key is a unique identifier for each record in a table. It is a single attribute or a combination of attributes that ensures that no two rows in the table have the same value in the primary key column(s).

Characteristics:

Uniqueness: No two rows can have the same primary key value.
Non-nullability: A primary key value cannot be NULL.

Table : Student

Student ID	Name	Age
101	Asha	22
102	Usha	23
103	Nisha	22

Primary Key : Student-ID

Candidate Key:

A candidate key is a minimal set of attributes that can uniquely identify a record in a table. It has similar properties as that of a primary key.

It is also known as a minimal super key. A table can have multiple candidate keys, but only one can be selected as the primary key.

Table : Student

Student_ID	Email	Name	Age
101	asha@example.com	Asha	22
102	usha@example.com	Usha	23
103	nisha@example.com	Nisha	22

Candidate Key : Student-ID, Email

Super Key:

A super key is a set of one or more attributes that, when combined, can uniquely identify a record in a table. A super key can include the primary key as well as any additional attributes.

A super key may contain extraneous attributes.

A super key is a super set of Candidate key. i.e when additional attributes are combined with a candidate key, it makes it a super key.

Table : Student

Student_ID	Email	Name	Age
101	asha@example.com	Asha	22
102	usha@example.com	Usha	23
103	nisha@example.com	Nisha	22

Super Keys: {Student_ID}, {Email}, (Student_ID,Name}, {Email,Name}, (Email,age}

Foreign Key:

A foreign key is an attribute or a set of attributes in one table that refers to the primary key in another table. It is used to establish and enforce a link between the data in the two tables.

The foreign key would require every value present in a column/set of columns to match the referential table's primary key.

Table : Student

Student ID	Name	Age
101	Asha	22
102	Usha	23
103	Nisha	22

Table : Enrollments

Enrollment_ID	Student_ID	Course_ID
1001	101	CSE101
1002	102	CSE102
1003	101	CSE103

Foreign Key: Student_ID in Enrollments referencing Student_ID in Students

Composite Key:

A composite key is a combination of two or more attributes that together uniquely identify a row in a table. None of the individual attributes can uniquely identify the record by itself.

The attributes present in a set may not uniquely identify whenever we consider them separately. Thus, when we take them all together, it will ensure total uniqueness.

Table : Enrollments

Enrollment_ID	Student_ID	Course_ID
1001	101	CSE101
1002	102	CSE102
1003	101	CSE103

Composite Key: {Student_ID, Course_ID}

Alternate Key:

An alternate key is a candidate key that is not chosen as the primary key. It is an alternative way to uniquely identify records in the table.

Table : Student

Student_ID	Email	Name	Age
101	asha@example.com	Asha	22
102	usha@example.com	Usha	23
103	nisha@example.com	Nisha	22

Alternate Key: {Email}

Unique Key:

A unique key is similar to a primary key, but it allows one NULL value and ensures that all non-null values are unique.

Unique key is not a Primary key as it can have one NULL values.

Table : Enrollments

Employee_ID	Email	Name
201	seeta@example.com	Seeta
202	geeta@example.com	Geeta
203	NULL	Reeta

Unique Key: {Email}

DATABASE MANAGEMENT SYSTEMS
INTRODUCTION TO VIEWS

Views in SQL

- A view is a table whose rows are not explicitly stored in the database but are computed as needed from a view definition.
  - A relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a view.
  - Views in SQL are considered as a virtual table. A view also contains rows and columns.
  - To create the view, we can select the fields from one or more tables present in the database.
  - A view can either have specific rows based on certain condition or all the rows of a table.
  - Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s) when data redundancy is to be kept minimum while maintaining security

1. Creating view

A view can be created using the **CREATE VIEW** statement. We can create a view from a single table or multiple tables.

Syntax:

```
CREATE VIEW view_name AS
    SELECT column1, column2.....
    FROM table_name
    WHERE condition;
```

Note: view_name: Name for the View,
table_name: Name of the table,
condition: Condition to select rows

Sample tables:
Student_Detail

STU_ID	NAME	ADDRESS
1	Stephan	Delhi
2	Kathrin	Noida
3	David	Ghaziabad
4	Alina	Gurugram

Student_Marks

STU_ID	NAME	MARKS	AGE
1	Stephan	97	19
2	Kathrin	86	21
3	David	74	18
4	Alina	90	20
5	John	96	18

Example : Creating View from a single table

In this example, we create a View named DetailsView from the table Student_Detail.

Query:

```
CREATE VIEW DetailsView AS
    SELECT NAME, ADDRESS
    FROM StudentDetails
    WHERE S_ID < 4;
```

Just like table query, we can query the view to view the data.

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Stephan	Delhi
Kathrin	Noida
David	Ghaziabad

Example : Creating View from multiple tables

View from multiple tables can be created by simply including multiple tables in the SELECT statement.

Syntax:

```
CREATE VIEW MarksView AS
SELECT Table1.colname1, Table1.colname2, Table2.colname3
FROM Table1,Table2
WHERE table1.colname4=table2.colname2;
```

In the given example, a view is created named MarksView from two tables Student_Detail and Student_Marks.

Query:

```
CREATE VIEW MarksView AS
  SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
  StudentMarks.MARKS
  FROM StudentDetails, StudentMarks
  WHERE StudentDetails.NAME = StudentMarks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

Output:

NAME	ADDRESS	MARKS
Stephan	Delhi	97
Kathrin	Noida	86
David	Ghaziabad	74
Alina	Gurugram	90

2. Update SQL View:

The SQL view created can also be modified. We can do the following operations with the SQL VIEW.
But, **all views are not updatable.** A SQL view can be updated if the following conditions are satisfied.

1. The view is defined based on only one table.
2. The view should not have any field which is made of an **aggregate** function.
3. The view must not have GROUP BY, HAVING or DISTINCT clause in its definition.
4. The view should not be created using any nested query.
5. The selected output fields of the view must not use constants, string or value expressions.
6. If you want to update a view based on another view then that view should be updatable.

Updating a SQL View
We can use the **CREATE OR REPLACE VIEW** statement to **modify** the SQL view.

Syntax:

```
CREATE OR REPLACE VIEW view_name AS
    SELECT column1,column2,..
    FROM table_name
    WHERE condition;
```

Example: If we want to update the view marksview1 and remove the attribute "*Age*" from in the view then the query would be:

Query

```
mysql> create or replace view marksview1 as select student_detail.name,
    student_marks.marks from student_detail, student_marks where
    student_detail.std_id=student_marks.std_id;
    Query OK, 0 rows affected (0.03 sec)
```

The above **CREATE OR REPLACE VIEW statement** would create a virtual table based on the result of the SELECT statement. Now, you can query the SQL VIEW as follows to see the output:

Output

```
mysql> select * from marksview1;
+---------+-------+
| name    | marks |
+---------+-------+
| stephan |    97 |
| kathrin |    86 |
| david   |    74 |
| alina   |    90 |
+---------+-------+
4 rows in set (0.03 sec)
```

Note:;

- Views are dynamically generated, hence everytime user queries the view it fetches the information from the original table.
- Rows inserted into an updatable view will insert the corresponding row into the original table.
- Attempts to insert rows through a view that doesnot contain Primary Key will be rejected as PK cannot take Null values.
- When views are created based on selection, the insertions may reflect in base table but may not be present in the view.

Deleting View

A view can be deleted using the Drop View statement.

Syntax

DROP VIEW view_name;

View_name: Name of the View which we want to delete.

Example:

If we want to delete the View **MarksView**, we can do this as:

EX: DROP VIEW MarksView;

NOTE: A view will also be dropped if a table is dropped using the CASCADE option.

Example : Drop table Student CASCADE;

However, a drop table with restrict can be used to prevent accidental deletions.

Drop table Student RESTRICT

will not drop student table if there is a view or integrity constraint refers to Student.

Executed Queries on Views:

Creating and inserting data into student_details table:

```
mysql> create table student_detail(std_id int,name varchar(20),address varchar(20));
Query OK, 0 rows affected (0.33 sec)

    mysql> insert into student_detail(std_id,name,address) values
    (1,'stephan','delhi'),(2,'kathrin','noida'),(3,'david','ghaziabad'),(4,'alina','gurugram');
    Query OK, 4 rows affected (0.11 sec)
    Records: 4 Duplicates: 0 Warnings: 0
```

Creating and inserting data into student_marks table:

```
mysql> create table student_marks(std_id int,name varchar(20),marks int,age int);
Query OK, 0 rows affected (0.11 sec)

    mysql> insert into student_marks(std_id,name,marks,age) values
    (1,'stephan',97,19),(2,'kathrin',86,21),(3,'david',74,18),(4,'alina',90,19),(5,'john',96,18);
    Query OK, 5 rows affected (0.02 sec)
    Records: 5 Duplicates: 0 Warnings: 0
```

Creating view detailsview on student_detial:

```
mysql> create view detailsview as select name,address from student_detail where std_id<4;
Query OK, 0 rows affected (0.17 sec)

mysql> select * from detailsview;
+---------+-----------+
| name    | address   |
+---------+-----------+
| stephan | delhi     |
| kathrin | noida     |
| david   | ghaziabad |
+---------+-----------+
3 rows in set (0.08 sec)
```

Creating view marksview on student_detial and s=student_marks:

```
mysql> create view marksview as select student_detail.name, student_detail.address,
student_marks.marks from student_detail, student_marks where
student_detail.name=student_marks.name;
Query OK, 0 rows affected (0.09 sec)

mysql> select * from marksview;
+---------+-----------+-------+
| name    | address   | marks |
+---------+-----------+-------+
| stephan | delhi     |    97 |
| kathrin | noida     |    86 |
| david   | ghaziabad |    74 |
| alina   | gurugram  |    90 |
+---------+-----------+-------+
4 rows in set (0.00 sec)
```

DATABASE MANAGEMENT SYSTEMS
AGGREGATE OPERATORS

In addition to simply retrieving data, we often want to perform some computation or summarization.
SQL allows a powerful class of constructs for computing aggregate values such as COUNT, MIN etc.
Aggregate functions are used in MySQL to calculate a set of values and return a single value as a result.
These functions are useful when we want to analyze data stored in a table and generate summary information, such as the total number of rows, the sum of values in a column, or the average of a set of values.
Using aggregate functions, we can filter data, calculate statistics, and improve performance (by limiting the data that needs to be processed).

SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. **COUNT ([DISTINCT] A):** The number of (unique) values in the A column.
2. **SUM ([DISTINCT] A):** The sum of all (unique) values in the A column.
3. **AVG ([DISTINCT] A):** The average of all (unique) values in the A column.
4. **MAX (A):** The maximum value in the A column.
5. **MIN (A):** The minimum value in the A column

GROUP BY():

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions to group the result-set by one or more columns.

HAVING CONDITION:

The HAVING clause was added to SQL to check the conditions with aggregate functions.

Syntax:

```
SELECT column_name(s)
FROM table_name      WHERE condition
GROUP BY column_name(s) HAVING condition
ORDER BY column_name(s);
```

Use this sample EMP table for illustrating the group functions

EMP TABLE

empno	ename	job	mgr	hiredate	sal	comm	deptno
7369	SMITH	CLERK	7902	1980-12-17	800	NULL	20
7499	ALLEN	SALESMAN	7698	1981-02-20	1600	300	30
7521	WARD	SALESMAN	7698	1981-02-22	1250	500	30
7566	JONES	MANAGER	7839	1981-04-02	2975	NULL	20
7654	MARTIN	SALESMAN	7698	1981-09-28	1250	1400	30
7698	BLAKE	MANAGER	7839	1981-05-01	2850	NULL	30
7782	CLARK	MANAGER	7839	1981-06-09	2450	NULL	10
7788	SCOTT	ANALYST	7566	1982-12-09	30000	NULL	20
7839	KING	PRESIDENT	NULL	1981-11-17	25000	NULL	10
7844	TURNER	SALESMAN	7698	1981-09-08	45500	0	30
7876	ADAMS	CLERK	7788	1983-01-12	21100	NULL	20
7900	JAMES	CLERK	7698	1981-12-03	25950	NULL	30
7902	FORD	ANALYST	7566	1981-12-03	63000	NULL	20
7934	MILLER	CLERK	7782	1982-01-23	75300	NULL	10

COUNT():

COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table.
COUNT(*) considers duplicate and Null.
When we apply COUNT() on a column then NULL values are ignored.

SYNTAX:

SELECT COUNT(*) FROM table_name WHERE condition;

OR

SELECT COUNT([ALL|DISTINCT] column_name) FROM table_name WHERE condition;

Examples:

select count(*) from emp;	14
select count(comm) from emp;	4
select count(job) from emp;	14
select count(distinct job) from emp;	5
select count(ename) from emp where deptno=30;	6

select count(ename) "No. of Employees", deptno from emp group by deptno;

Empl	DeptNo
3	10
5	20
6	30

select count(ename) "No. of Employees", deptno from emp group by deptno having count(ename)>3;

Empl	DeptNo
5	20
6	30

SUM()

The SUM() function returns the total sum of a numeric column.

SYNTAX:

SELECT SUM(COL_NAME) FROM table_name

[WHERE CONDITION]

[GROUP BY COL_NAME] [HAVING CONDITION];

Example:

select sum(sal) from emp;	299025.00
select sum(sal) from emp where job='analyst';	93000.00
select sum(sal) from emp where deptno=10;	102750.00

select job, sum(sal) from emp group by job;

```
mysql> select job, sum(sal) from emp group by job;
+-----------+------------+
| job       | sum(sal)   |
+-----------+------------+
| CLERK     | 123150.00  |
| SALESMAN  |  49600.00  |
| MANAGER   |   8275.00  |
| ANALYST   |  93000.00  |
| PRESIDENT |  25000.00  |
+-----------+------------+
5 rows in set (0.00 sec)
```

select deptno, sum(sal) from emp group by deptno having sum(sal)>100000;

```
mysql> select deptno, sum(sal) from emp group by deptno having sum(sal)>100000;
+--------+-----------+
| deptno | sum(sal)  |
+--------+-----------+
|     10 | 102750.00 |
|     20 | 117875.00 |
+--------+-----------+
2 rows in set (0.00 sec)
```

AVG()

The AVG() function returns the average value of a numeric column.

SYNTAX:

SELECT AVG(COL_NAME) FROM table_name
[WHERE CONDITION]
[GROUP BY COL_NAME] [HAVING CONDITION];

Example:

select avg(sal) from emp; 21358.92

select avg(sal) from emp where job='analyst'; 123150.00

select deptno, avg(sal) from emp group by deptno having avg(sal)>15000;

MAX() and MIN()

The MIN() function returns the smallest value of the selected column.
The MAX() function returns the largest value of the selected column.

SYNTAX:

SELECT MIN(COL_NAME) FROM table_name
[WHERE CONDITION]
[GROUP BY COL_NAME] [HAVING CONDITION];

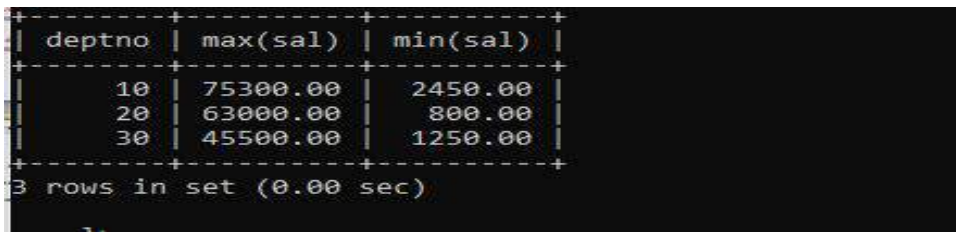
SELECT MAX(COL_NAME) FROM table_name
[WHERE CONDITION]
[GROUP BY COL_NAME] [HAVING CONDITION];

Example:

select min(sal) from emp; 800.00

select max(sal) from emp where job='analyst'; 63000.00

select deptno, max(sal), min(sal) from emp group by deptno;



```
+--------+----------+----------+
| deptno | max(sal) | min(sal) |
+--------+----------+----------+
|     10 | 75300.00 |  2450.00 |
|     20 | 63000.00 |   800.00 |
|     30 | 45500.00 |  1250.00 |
+--------+----------+----------+
3 rows in set (0.00 sec)
```

DATABASE MANAGEMENT SYSTEMS
UNIT II – TOPIC 3
INTEGRITY CONSTRAINTS

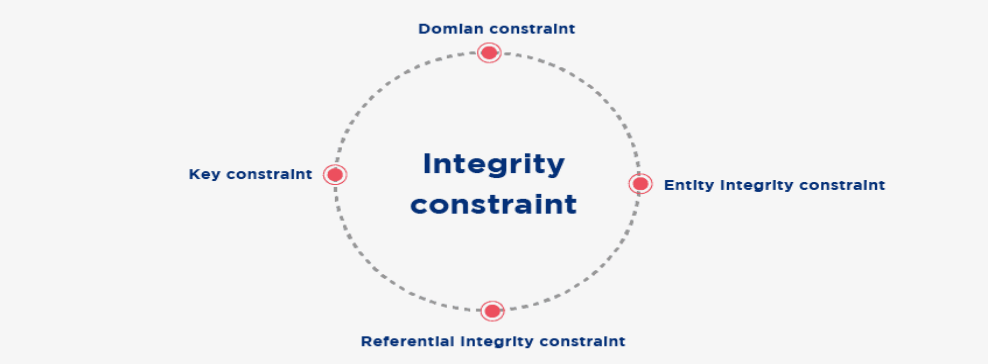
Integrity constraints are the set of predefined rules in a database to maintain data accuracy, consistency and reliability. In short, it is used to maintain the quality of information. They can be used to enforce business rules or to ensure that data is entered correctly. Integrity constraints can also be used to enforce relationships between tables. They ensure that the data in the database adheres to a set of rules, which can help prevent errors and inconsistencies.

Integrity Constraints act as guidelines ensuring that data in the database remain accurate and consistent. They guard against accidental damage to the database.

Integrity constraints in SQL can be either enforced by the database system or by application code. Enforcing them at the database level can help ensure that the rules are always followed, even if the application code is changed. However, enforcing them at the application level can give the developer more flexibility in how the rules are enforced.

Types of Integrity Constraint

There are four main types of integrity constraints: domain, entity, referential, and key.



1. Domain constraints

Domain constraints can be defined as the definition of a valid set of values for an attribute. The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

Example:

Eid	Ename	Job	Sal	Location	Deptno
100	Rama	Scientist	102000	Hyd	12
101	Srikar	Manager	80300	Hyd	20
102	Krishana	Analyst	2Lakhs	Sec	24

Here for Eid 102, Salary attribute violates the domain constraint as it is allowed to have only numeric values.

2. Entity integrity constraints

An entity integrity constraint is a restriction on null values. Null values are values that are unknown or not applicable, and they can be problematic because they can lead to inaccurate results. This constraint serves unique identification of each tuple in the relation.

The primary key attribute is used to identify individual rows in relation and entity integrity constraint states that it cannot be NULL. However a table can contain a NULL value in other fields / attributes.

In the employee table below Eid is primary key and can't contain NULL value.

Eid	Ename	Job	Sal	Location	Deptno
100	Rama	Scientist	102000	Hyd	12
101	Srikar	Manager	80300	Hyd	20
	Krishana	Analyst	95600	Pune	24

3. Referential Integrity Constraints (Foreign Key)

A referential integrity constraint is be used to enforce relationships between tables. The foreign key a constraint is a column or list of columns that points to the primary key column of another table.

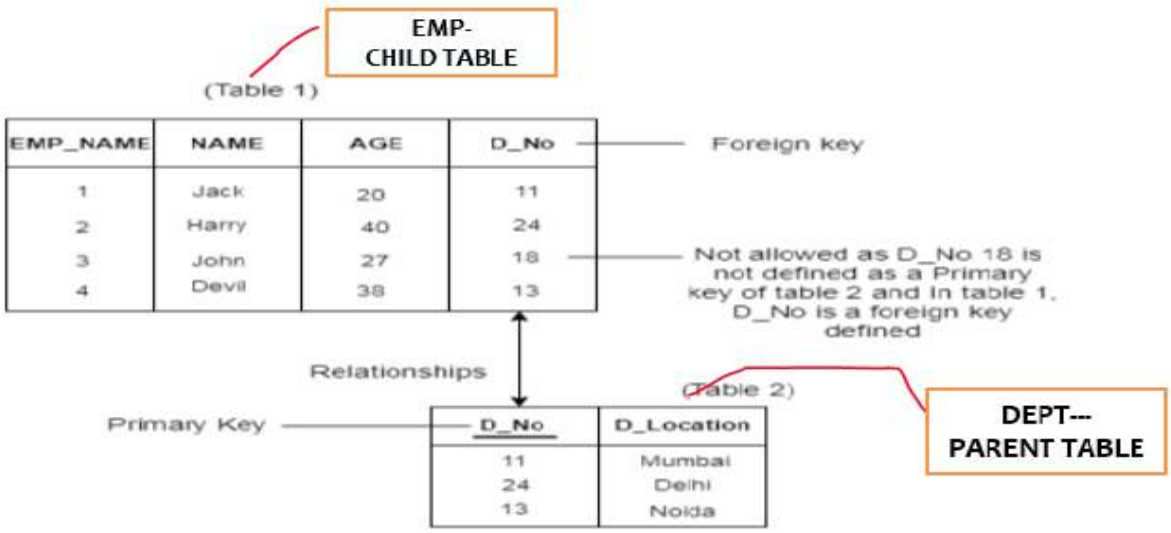
In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

The rules are:

- can't delete a record from a primary table if matching records exist in a related table.
- can't change a primary key value in the primary table if that record has related records.
- can't enter a value in the foreign key field of the related table that doesn't exist in the primary key of the primary table.
- However, you can enter a Null value in the foreign key, specifying that the records are unrelated.

Example:



4. **Key Constraints**

Key constraints in DBMS are a restriction on duplicate values. A key is composed of one or more columns whose values uniquely identify each row in the table. These are called uniqueness constraints since it ensures that every tuple in the relation should be unique.

Keys are the entity set that is used to identify an entity within its entity set uniquely.
A entity set can have multiple keys or candidate keys (minimal superkey), out of which we choose one of the keys as the primary key.
Primary key is always unique and cannot have null values in the relational table.

Example:

In the below example Eid is primary key but Eid have the same value 100. so, it is violating the key constraint

Eid	Ename	Job	Sal	Location	Deptno
100	Rama	Scientist	102000	Hyd	12
101	Srikar	Manager	80300	Hyd	20
100	Krishana	Analyst	95600	Pune	24

Other Key Constraints:

1. **NOT NULL Constraints**
2. **NULL Constraints**
3. **PRIMARY KEY Constraints**
4. **FOREIGN KEY Constraints**
5. **UNIQUE Constraints**
6. **CHECK Constraints**

NOT NULL Constraint:

The NOT NULL constraint in SQL is used to ensure that a column in a table doesn't contain NULL (empty) values, and prevent any attempts to insert or update rows with NULL values.

The NOT NULL constraint enforces a column to NOT accept NULL values, which means that you cannot insert a new record, or update a record without adding a value to this field.

NULL represents a record where data may be missing data or data for that record may be optional. By default a column is allowed to have NULL values, but the NOT NULL constraint enforces a column to NOT accept NULL values

A not-null constraint cannot be applied at table level

NOT NULL on CREATE TABLE

```
CREATE TABLE Persons (
   PID int (3) NOT NULL,
   PName varchar (10) NOT NULL,
   PADD varchar (20) NOT NULL,
   PAge int (2)
 );
```

NULL Constraint:

- By default, a column can hold NULL values

The term **NULL** in SQL is used to specify that a data value does not exist in the database. It is not the same as an empty string or a value of zero, and it signifies the absence of a value or the unknown value of a data field.

Some common reasons why a value may be NULL −

- The value may not be provided during the data entry.
- The value is not yet known.

It is important to understand that you cannot use comparison operators such as "=", "<", or ">" with NULL values. This is because the NULL values are unknown and could represent any value. Instead, you must use "IS NULL" or "IS NOT NULL" operators to check if a value is NULL

Syntax

The basic syntax of **NULL** while creating a table.

```
CREATE TABLE CUSTOMERS (
        ID   INT(3)          NOT NULL,
        NAME VARCHAR (20)    NOT NULL,
        AGE  INT(2)          NOT NULL,
        ADDRESS  VARCHAR (25) ,
        SALARY   FLOAT(6, 2),
   );
```

In the above example SALARY and ADDRESS are NULL constraints.

PRIMARY KEY Constraint :

The PRIMARY KEY constraint uniquely identifies each record in a table.

- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

```
CREATE TABLE Persons(
    PID int(3) NOT NULL,
    PName varchar(10) NOT NULL,
    PADD varchar(20) NOT NULL,
    PAge int (2),
    PRIMARY KEY (PID)
);
```

FOREIGN KEY Constraint :

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.

FOREIGN KEY constraints enforce referential integrity.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

The rules are:

- can't delete a record from a primary table if matching records exist in a related table.
- can't change a primary key value in the primary table if that record has related records.
- can't enter a value in the foreign key field of the related table that doesn't exist in the primary key of the primary table.
- However, you can enter a Null value in the foreign key, specifying that the records are unrelated.

```
CREATE TABLE dept (
        deptno int(3),
        dname varchar(20),
        location varchar(20),
        Primary key(deptno) );      -- PARENT TABLE OR Primary table

CREATE TABLE Emp (
        eid int(3),
        ename varchar(20),
        salary float(6,2),d
        no int(3),
        PRIMARY KEY  (Eid),
        FOREIGN KEY (dno) REFERENCES dept(deptno) );     -- CHILD TABLE
```

UNIQUE Constraint:

The UNIQUE constraint in SQL is used to ensure that no duplicate values will be inserted into a specific column or combination of columns that are participating in the UNIQUE constraint and not part of the PRIMARY KEY.
In other words, the index that is automatically created when you define a UNIQUE constraint will guarantee that no two rows in that table can have the same value for the columns participating in that index, with the ability to insert only NULL value to these columns.

The UNIQUE constraint ensures that all values in a column are different.
Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns
A PRIMARY KEY constraint automatically has a UNIQUE constraint.

Difference between UNIQUE constraints and PRIMARY KEY constraint is unique allows NULL values whereas primary key not allowed the NULL values.

```
CREATE TABLE Persons(
    PID int(3) UNIQUE,
    PName varchar(10) NOT NULL,
    PADD varchar(20),
    PAge int (2),
    Padhar varchar(12)
    PRIMARY KEY (Padhar) );
```

CHECK Constraint:

- The CHECK constraint is used to limit the value range that can be placed in a column.
- If you define a CHECK constraint on a column it will allow only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

```
CREATE TABLE Persons (
    PID int(3),
    PName varchar(10) NOT NULL,
    PAdd varchar(20) NOT NULL,
    PAge int (2),
    PRIMARY KEY(PID),
    CHECK (PAge>=18)  );
```

If we enter the Page below 18 then shows that CHECK Constraint is violated.

```
CREATE TABLE Emp(
    EID int(3),
    EName varchar(20) NOT NULL,
    Doj date,
    Mobile bigint CHECK(Length(Mobile)=10),
    PRIMARY KEY(PID) );
```

Check constraint here ensures that mobile number is exactly 10 digits.

DATABASE MANAGEMENT SYSTEMS
UNIT II – TOPIC 1
RELATIONAL MODEL

What is Relational Model?

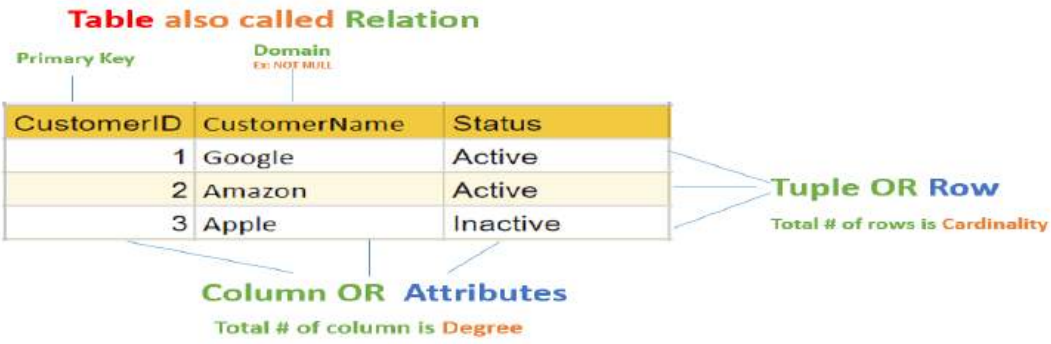
Relational model can represent as a table with columns and rows. Each row is known as a tuple. Each table of the column has a name or attribute.

Relational Model (RM) represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

The table name and column names are helpful to interpret the meaning of values in each row. The data are represented as a set of relations. In the relational model, data are stored as tables. However, the physical storage of the data is independent of the way the data are logically organized.

Relational Model – Keywords / Terminologies

1. **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student_Rollno, NAME, etc.
2. **Tables** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
3. **Tuple** – It is nothing but a single row of a table, which contains a single record.
4. **Relation Schema:** A relation schema represents the name of the relation with its attributes.
5. **Degree (Arity**): The total number of attributes which in the relation is called the degree of the relation.
6. **Cardinality:** Total number of rows present in the Table.
7. **Column:** The column represents the set of values for a specific attribute.
8. **Relation instance** – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.
9. **Relation key** - Every row has one, two or multiple attributes, which is called relation key.
10. **Attribute domain** – Every attribute has some pre-defined value and scope which is known as attribute domain
11. **Relational database** – A collection of relations with distinct relation names
12. **Relational database schema** - The collection of schemas for the relations in the database.



Relational Model Concepts:

1. **Data is Organized into Tables:** Data is organized into tables, also known as relations. Each table consists of rows and columns. Rows represent individual records, while columns represent attributes or fields.

2. **Tables Have Keys:** Each table has one or more columns that uniquely identify each row. These are called primary keys. Additionally, there can be foreign keys, which establish relationships between tables.

3. **Relationships Between Tables:** Tables can be related to each other through common attributes. These relationships can be one-to-one, one-to-many, or many-to-many.

4. **Data Integrity**: The relational model enforces data integrity through constraints such as primary key constraints, foreign key constraints, and other integrity rules. These constraints ensure that data remains consistent and accurate.

5. **Structured Query Language (SQL):** The relational model is typically manipulated using a special-purpose language called SQL (Structured Query Language). SQL provides a standardized way to perform operations such as querying, updating, and deleting data in relational databases.

6. **Normalization:** The process of organizing data to minimize redundancy and dependency is called normalization. It involves breaking down large tables into smaller ones and defining relationships between them to reduce redundancy and improve data integrity.

7. **Transactions:** Relational databases support transactions, which are units of work that must be performed atomically (all or nothing), consistently (in a valid state), isolated (separate from other transactions), and durably (persistently stored).