Summary

17.02.2023 to 24.02.2023

Notes

Types of SQL Commands

DDL, DML,DCL & DQL

1- DDL: Stands for Data Definition Language;related to the structure of the database.

Create Statement &Using Databases	Create tables	Alter Statement (to modify columns)	Truncate	Drop

**create database database_name;
use database_name;**
	**create table student (
	id int,
	username varchar(100));**
	
	**create table student1 (
	id int primary key,
	username varchar(100));**
	#Primary key added
	
	**create table student5 (
	id int primary key,
	username varchar(100),
	gender char(1)
	CHARACTER SET ASCII);**
	# TO SPECIFY THE CHAR TO BE IN ASCII
	
	**create table student6 (
	id int primary key
	auto_increment,
	username varchar(100) not null,
	gender char(1)
	CHARACTER SET ASCII,
	date1 datetime,
	date2 timestamp);**
	**# Specifying the dates and time as a data type.
	#timestamp gives UTC.**
		**alter table student
		rename column id to user_id;**
		# Rename a column
		
		alter table table_name change column col_name_old col_name_new varchar(200);
		#to change the name of the column and also the data type of it.
		
		alter table vehicle add column description varchar(200) not null after vehicle_no;
		#to add a column in a certain position in the table
		
		alter table vehicle add column Vehicle_color varchar(200) not null first;
		#to add a column in the first position/column in the table
			truncate table student3;
			# removes the data in the rows
				drop table student3;
				# removes the whole table

	create table student7 (id int primary key auto_increment, username varchar(100) not null, gender char(1) CHARACTER SET ASCII, salary decimal(10,2)); **#Specifying the Decimal as data type as it allows here up to 10 char and 2 decimal points** **create table student11 select * from student6;** **# to copy a table into a new one;it gives a new table student11 that has the same data in student6**	**alter table vehicle alter description set default 'car is superb'; describe vehicle;** **#to change the default value of specific column if it is NULL to 'car is superb' for ex.**		

2- DCL:Stands for Data control language;related to grant and revoke access to the database objects which helps in the data security

- Grant
- Revoke

<u>Creating a User in SQL using the CREATE USER Statement</u>

/*

```
CREATE USER username@hostname IDENTIFIED BY 'pw';
grant all on *.* to username@hostname;
# using GRANT Statement to grant the user all privileges and permissions to access
and work within specific database (database_name.*) or all the databases(*.*)
GRANT SELECT, INSERT, UPDATE ON *.* TO username@hostname;
# or using this statement to grant the user to only work within select,insert and
update within the database specified
show grants for username@hostname;
# to show all the grants given to this user
select user,host,account_locked,password_expired from mysql.user;
select user,host from mysql.user;
# it gives the current status of all the users on MySQL server
select user,host,db,command from information_schema.processlist;
#it gives the current running process on MySQL server including the database used
and the type of command used in this db
select user();
#returns the username and hostname of the currently logged-in user
```

Drop user username@hostname;
#removes the user
UPDATE mysql.user SET Password=PASSWORD('rovan') WHERE User='rovantest';
SET PASSWORD FOR rovantest@localhost = PASSWORD('rovan');
# to change the password of user; but it is not working as it works only for older versions of MySQL
FLUSH PRIVILEGES;
# to make sure the password or any privilege is updated and executed immediately in MySQL server
/*

3- DML:Stands for Data Manipulation Language;related to manipulating the data in the database

Insert Statement	Update Statement	Delete Statement

insert into student (id,username) values (55,'rovan');
 	
insert into student6 (username,gender,date1,date2) values ('rovan','f',now(),now());
# using now() to set the current data and time 	
	update student set id=88 where username='rovan';
	# it will give error at first because the safety mode is on,you can disable from edit-preferences-sql editor-disable it
		delete from student where id=1;

4- DQL:Stands for Data Query Language;related to Select Query to retrieve data from one or more tables

SELECT Statement

SELECT * from student;
# key words can be used as capital or small letter 	
 	
select '123' * 2;
# also it is a character but it is doing the mathematical operation anyway 	
 	
select '123abc' * 2;
# also works as it can be used if there is a currency attached in the salary and you want to compute it anyway 	
 	
select 'abc123' *2;
# doesn’t work because the characters are in the first 	
 	
select table_name,engine from information_schema.tables;

#Retrieves the name of the tables and its current storage engine in the database used

Keywords

- **show tables;** # it shows all the tables created in this database being used.
- **describe table_name;** or **show columns from table_name;** # to show the column name,type,null,key,default,extra which is used for an auto-increment attribute.
- **show full columns from table_name;** # here it shows beside the previous ones, the privileges
- **show columns from table_name like 'v%';** #show the columns in a table that start with..
- **show character set like 'ascii%';** # it shows the ascii definition/description
- **SHOW CHARACTER SET WHERE Charset = 'ascii';** # same result
- **show engines;** # it shows all the engines that can be used for storage
- **Show table status;** # it shows all the status of all the tables including the Engine used in the database used
- **lock table table_name read;** #to lock a table to be only-read;can't execute insert,update,delete statements
- **unlock tables;** # to unlock the table
- **select connection_id();** # to check the status of the connection as it is useful to know which user is clogging up the connection for others
- **select user,host,db,command from information_schema.processlist;** #is used after knowing the connection id to see which command is in the query state or from client connections under Server Tab
- **select current_timestamp();** # timestamp is in UTC timezone if you want to retrieve the data later according to any timezone
- **Select last_insert_id();** # it shows the last id for the PK in the table
- **show create table sales**; # it shows the partitioning information about a table
- **SET SQL_SAFE_UPDATES = 0;**# to turn off the safety mood for update/delete
- **SET SQL_SAFE_UPDATES = 1;**#to turn on again

Changing the Engine for specific tables to improve the database performance (for Administrators/Developers)

select table_name,engine from information_schema.tables where table_name='table_name';
Show table status;
#Retrieves the metadata about this table and the current engine used for storage
alter table vehicle ENGINE='MyISAM';
#for example to change the engine used in this table to another engine like MyISAM
repair table table_name;
#to repair a table that has issue with data accessibility or crashing.
repair table table_name quick extended;
# to repair this table and all the table related to it

CREATE VIEW Statement to Create Virtual Table to avoid running the same command line multiple times-specially with enormous data

```
create view new_table_name as select col1,col2 from table_name where col3=' ';
select * from new_table_namer;
```

SQL Order Execution

For readability clauses are interpreted from left to right, but SQL executes them usually in a different order.
The order in which the clauses in queries are executed is as follows:
1. FROM/JOIN: The FROM and/or JOIN clauses are executed first to determine the data of interest.
2. WHERE: The WHERE clause is executed to filter out records that do not meet the constraints.
3. GROUP BY: The GROUP BY clause is executed to group the data based on the values in one or more columns.
4. HAVING: The HAVING clause is executed to remove the created grouped records that don’t meet the constraints.
5. SELECT: The SELECT clause is executed to derive all desired columns and expressions.
6. ORDER BY: The ORDER BY clause is executed to sort the derived values in ascending or descending order.
7. LIMIT/OFFSET: Finally, the LIMIT and/or OFFSET clauses are executed to keep or skip a specified number of rows.

Join Function
(inner join=join,natural join,left join=left outer join,right join-right outer join,cross join,full joinunion)

```
create table customer_new(customer_id int, customer_name varchar(250) , address varchar(400))
create table order_new(order_id int, order_date date, customer_id int, shipper_id int)

insert into customer_new values(1,'agsagdhas', 'ashgdhagdhgagd cgda'),
(2,'ssdagsagdhas', 'ashgdhagdhgagd cgda'),
(3,'h5agegsagdhas', 'ashgdhagdhg agd cgda'),
(4,'qwagsagdhas', 'ashgdhagdhgagd cgda'),
(5,'bbragsagdhas', 'ashgdhagdhgagd cgda')


insert into order_new values(1,'2022-05-05',1,2),
(2,'2022-05-05',6,2),
(3,'2022-05-05',7,2),
(4,'2022-05-05',2,2),
(5,'2022-05-05',3,2)

select distinct o.order_id as OrderNumber, c.customer_name as CustomerName
from order_new o inner join customer_new c on c.customer_id = o.customer_id;

select distinct o.order_id as OrderNumber, c.customer_name as CustomerName
from order_new o left outer join customer_new c on c.customer_id = o.customer_id;

select o.order_id as OrderNumber, c.customer_name as CustomerName
from order_new o left join customer_new c using (customer_id);

select o.order_id as OrderNumber, c.customer_name as CustomerName
from order_new o left outer join customer_new c on c.customer_id = o.customer_id
union
select o.order_id as OrderNumber, c.customer_name as CustomerName
from order_new o right join customer_new c using (customer_id);

select  o.order_id as OrderNumber, c.customer_name as CustomerName
from order_new o inner join customer_new c
on c.customer_id = o.customer_id;

select  o.order_id as OrderNumber, c.customer_name as CustomerName
from order_new o cross join customer_new c
on c.customer_id = o.customer_id;

create table employee(eid int, emp_name varchar(250),  income int)
create table employee1(eid int, emp_name varchar(250),  income int, city varchar(236))
insert into employee1 values(1,'asjhdskj',7163,'a'),
(2,'asjhdskj',2734163,'a'),
(3,'asjhdskj',73,'b'),
(4,'asjhdskj',7161233,'a'),
(10,'asjhdskj',716332,'b')

select city,min(income) as "Minimum income"
from employee1
group by city
having min(income)>73

create table emp_business (e_id int, e_name varchar(250), occupation
varchar(250),working_date date, working_hours int)

insert into emp_business values(1,'aksjdhgkjsad','IT_DB',null, 5),
(2,'aksdjdhgkjsad','IT_Com',null, 7),
(3,'aksjdhgkjsad','IT_tech',null, 3),
(4,'aksjdhgkjsad','IT_HR',null, 7),
```

(5,'aksjdhgkjsad','IT_innovat',null, 6),
(6,'aksjdhgkjsad','IT_resarch',null, 4)

select occupation ,sum(working_hours) as "total working hours" from emp_business group by occupation having sum(working_hours)>6
select occupation ,avg(working_hours) as "total working hours" from emp_business group by occupation having avg(working_hours)>6

Using ALL:
Select * from products where price >= ALL(select min(price) from products where category='electronics');

REGULAR EXPRESSION

^: indicates starting beginning of string
$: indicates end of string
.: indicates any single character except new line
\n: stored new space
\t: space or tab
[abc]: any character in the square bracket
[^abc]: any character is not in the square bracket.
*: indicates zero or more string or character
+: indicates one or more string or character
{n}: indicates match n instances preceding element
[A-Z]:: any upper case
[a-z]: any small case
[0-9]: from 0 to 9

Using regexp/rlike:

```
select ('B' regexp '[A-Z]') as match_; # gives 1
select ('B' not regexp '[A-Z]') as not_match_; # gives 0
select ('B' regexp '[A-Z]') as matching; # gives 1
#using RLIKE with regexp
select ('B' RLIKE '[C-Z]') as not_match_; # gives 0
select ('B'  like '[A-Z]') as match_; # gives 0 because the two values are not the same
select ('B' not like '[A-Z]') as match_;   # gives 1 because of 'NOT' operator
```

```
# using regexp_like :
select regexp_like('a','[A-Z]','c') as check_; # c means case sensitive; to match the exact case
select regexp_like('a','[A-Z]','i') as check_; # i means ignore; to ignore the case sensitive
```

Indexing & Hashing

Hashing is the process to transform data of any size into a fixed-size value(Hash value/code), typically a numerical value, that represents the original data.

Given the same input, it will always produce the same hash value.

Hashing uses: cryptography, data structures, and databases.

- cryptography, hashing is used to securely store passwords and other sensitive information.
- data structures, hashing is used to quickly look up data in a table or dictionary.
- databases, hashing is used to speed up searches and queries, and to ensure data integrity and security.

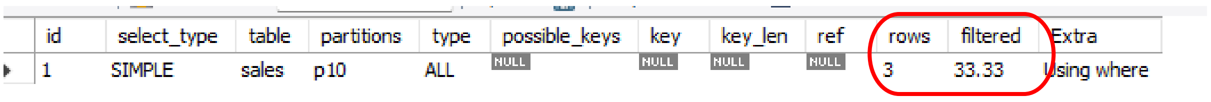
Note:Changing the Engine ,changes the index type.

Partition
(for improving the database performance even after indexing)

The partition creating should be made when CREATE TABLE statement
Example: for ex.we have already a table ‘sales1’ with data inserted already and we want to execute the partition process to it. So we copy the data in ‘sales1’ table to the new created table which is used for the partition.
Why that?
Because the partition process should be executed within the CREATE TABLE query.

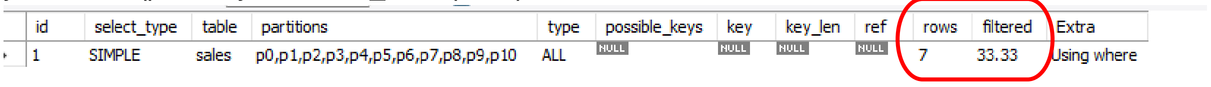
```
create table sales1(id int not null auto_increment,
sale_date date not null,
amount DECIMAL(10,2) not null,
PRIMARY KEY(id,sale_date));
INSERT INTO sales1 (id, sale_date, amount) VALUES
(1, '2014-11-03', 54353),
(2, '2017-01-04', 2576727),
(3, '2016-08-08', 757654),
(4, '2014-04-06', 324235),
(5, '2023-05-03', 23334),
(6, '2019-12-01', 2342134),
(7, '2030-11-03', 54353));
```

create table sales(id int not null auto_increment,
sale_date date not null,
amount DECIMAL(10,2) not null,
PRIMARY KEY(id,sale_date))
partition by range(year(sale_date))
(
partition p0 values less than (2010),
partition p1 values less than (2011),
partition p2 values less than (2012),
partition p3 values less than (2013),
partition p4 values less than (2014),
partition p5 values less than (2015),
partition p6 values less than (2016),
partition p7 values less than (2017),
partition p8 values less than (2018),
partition p9 values less than (2019),
partition p10 values less than MAXVALUE
);
insert into sales (id,sale_date,amount) select id,sale_date,amount from sales1; # to copy the data in a table to another table
select * from sales where year(sale_date)>2023;
explain select * from sales where sale_date='2019-12-01'; # it scans 3 rows only in p10 and retrieves one row of the 3 rows,so it filters 1 of 3 x 100



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	sales	p10	ALL	NULL	NULL	NULL	NULL	3	33.33	Using where

explain select * from sales where sale_date<'2019-12-01'; # returns all the partitions for year<2019 (p10 for year < max_value(2030)



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	sales	p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10	ALL	NULL	NULL	NULL	NULL	7	33.33	Using where

<u>Using Rank to rank specific columns like grades,score..</u>

create table student_score(name varchar(50),score int);

insert into student_score
values ('Alice',90),
('Bob',85),
('carl',75),
('Dave',80),
('eric',90);

select Name, score, rank() OVER (ORDER BY SCORE DESC) AS SCHOOL_RANK from student_score ;
select name, score from(select name,score,DENSE_rank() OVER (ORDER BY SCORE DESC) AS SCHOOL_RANK from student_score) as ranked_students where SCHOOL_RANK=3; # to get the name & score for the 3rd rank
select Name, score, DENSE_rank() OVER (ORDER BY SCORE DESC) AS SCHOOL_RANK from student_score;

Notes on ACID properties:

- Atomicity: either executes everything within the query or nothing
- Consistency: if a specific primary key is deleted in the main table then we should delete it also in the table having this PK as a FK.

Either by Trigger or DELETE ON CASCADE (not preferred)

- Isolation:if two users are using the same table at the same time;one is inserting new records and the other is reading data from the table

Here the DBMS will hold the reading process for the second user until the first user finishes his updates.

Meaning that the transactions are isolated and are executed separately to prevent data corruption/interference between transactions also the other transactions can't be read until the first transaction is done.

For example: transaction of money in a bank or transferring money between two banks during the weekend so you will not be able to see the process update immediately.

- Durability: if the power is off or software crashes suddenly the data that was committed in a database, it is guaranteed to be saved on Disk/backup and once the power is back it re-executes the data.
- Ex: using Whatsapp during bad internet, it saves the data in ur mobile phone until the internet is good.

<u>Delimiters&Triggers</u>
<u>Used for Data Validation</u>

Create Trigger Trigger_name
before/after insert on table_name for each row
begin
#any sql query you want to do
end;

MyResearch Work

DDL,DML,DCL & DQL
(Create,Alter,Truncate,Drop,Insert,Delete,Update,Select)

Example 1: Creating a table 'student' which has the student id and student name

create table student (
id int,
username varchar(100));

inserting options:
insert into student (id,username) values (55,'rovan');
insert into student (id) values (2);
#inserting a value for one column only specified: it will insert the data into one column only and the other is null because only one column is specified
insert into student values (6); # it will give error because at least one column name for the whole row must be specified
insert into student values (6,'rov.'); # it will succeed because all the columns is specified matching the values
insert into student3 (id,username) values (1,'rovan');
insert into student3 (id,username) values (1,'rovan'); # it will give an error because the id(which is PK) value already exists; PK is unique by Default.
insert into student (user_name) values ('r'); # it works when the PK is auto-incremented

Updating:
update student set id=88 where username='rovan';
update student set id=44,username='Nick' where username='rovan'; # to update more than one column for one row
Update student set gender='f' where id=55; #after adding column 'gender' we update it with values using WHERE clause

Altering:
alter table student
modify column user_name varchar(20); # to change the data type of column
describe student; # it shows the column and its data type
alter table student add primary key (id); # to modify one column as primary key
alter table student modify column user_id int auto_increment; # to make the PK auto increment
Alter table student add column gender char(1) CHARACTER SET ASCII; # TO Specify the Char to be in ASCII code

Truncate:

truncate table student; # removes the data in the rows

Deleting:
delete from student where user_id=null;

Select:
select max(user_id) from student; # to know the last value for the PK if we have to insert it manually; not auto-incremented

Example2: Creating a table for which student is enrolled in a course that has a start/end date (using datetime & timestamp data types)

create table student6 (
id int primary key auto_increment,
username varchar(100) not null,
gender char(1) CHARACTER SET ASCII,
Course Varchar(20),
date1 datetime,
date2 timestamp); # timestamp is UTC

Inserting:
insert into student6 (username,gender,course,date1,date2) values ('rovan','f',’eng’,now(),now());
insert into student6 (username,gender,date1,date2) values ('erf','m',’math’,'2022-01-29 10:40:01',now());
insert into student6 (username,gender,date1,date2) values ('erf','m',’programming’,now(),now());

Select:
select date_add('2022-12-01', interval 1 day); # to add a period(DAYS,YEARS,MONTHS,HH,MM OR SS) to certain date
select date_add('2022-12-01 17:30:01', interval '2:30' hour_minute);
select datediff(curdate(),date1) from student6; # it gives the period in days
select datediff(now(),date1) from student6; # using curdate() or now()
select datediff(20220301,20220101) ; # gives the period in days
SELECT
TIMESTAMPDIFF(YEAR, date1, NOW()) AS years_diff,
TIMESTAMPDIFF(MONTH, date1, NOW()) AS months_diff,
TIMESTAMPDIFF(DAY, date1, NOW()) AS days_diff,
TIMESTAMPDIFF(HOUR, date1, NOW()) AS hours_diff,
TIMESTAMPDIFF(MINUTE, date1, NOW()) AS minutes_diff,
TIMESTAMPDIFF(SECOND, date1, NOW()) AS seconds_diff
FROM student6; # it calculates the period from certain date in years,month,days,hours,minutes or seconds

select timediff(now(),date1) from student6; # it gives the period in H minutes seconds
select period_diff(202203,202201); # to calculate the period between two dates of YYYYMM format only

Example3: The difference between the data types **Decimal and Float**

create table student7 (
id int primary key auto_increment,
username varchar(100) not null,
gender char(1) CHARACTER SET ASCII,
Salary_Before_Taxes decimal(10,2)
Salary_After_Taxes float
);

inserting:
insert into student7
(username,gender,salary_Before_Taxes,Salary_After_Taxes) values
('rovan','f',5000.533,4000.539); # decimal value: it will remove the last decimal point because it should show 2 decimal points only as specified without approximation (5000.53) ; which is better to use if we want precise value.
#float value: it will give the approximate value (4000.54) ; gives less precise value.

Example4: Using ALTER statement for different purposes

create table vehicle(
vehicle_no varchar(20) primary key,
Description varchar(20) not null,
model_name varchar(45),
price decimal(10,2),
sell_price decimal(10,2));

insert into vehicle(vehicle_no,model_name,price,sell_price)
values('ssajd314231','mercedes',276473,18797),
('ssajd3142328438','mercedes1',47647,48797),
('ssajd31423341','mercedes2',67647,98797);

Alter:

Changing the data type:
# Before we change the data type, we should do the **' show table; '** command line to know what is the data type for this column first and to re-add the data types we want to keep in the command line besides the new ones.
alter table vehicle modify description varchar(20) default 'car is superb';
alter table vehicle modify description varchar(20); # this will delete the default set earlier ' car is superb' ot be 'null'

Renaming a column or more than one column:
alter table vehicle rename column description to car_description;
alter table vehicle rename column description to car_description,rename column vehicle_color to color;

Renaming a column and changing the data type:
alter table vehicle change column car_description description varchar(200);

Showing Column names starts with…:

show columns from vehicle like 'v%';
show columns from vehicle like '%i%';
show columns from student like '%_i%';
show COLUMNS FROM VEHICLE LIKE '%d' or 'v%'; # doesn't work
show full columns from vehicle like '_e%'; # _ is used as a wild character instead of %;means that before the letter 'e',there is one character that only existed.
# %: is a wild character used to describe many characters
# _: is a wild character used to describe one character

Locking and Unlocking Tables (Read or Write)
lock table vehicle read; #locks the table to be read only
show full columns from vehicle; # it shows the privilege that we can write the statements(insert,update,delete..) but it can't be executed
lock table vehicle write; # locks the tables to be read or written into
unlock tables;

SELECT CASE STATEMENT
used to perform conditional logic in SQL

```
SELECT
  CASE expression
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    ...
    ELSE default_result
  END
FROM table_name;
```

Example: Creating table employees to show whether they are active or not within the company

```
CREATE TABLE employees (     # research of active or not employee using boolean values
as tinyint(1)
  id INT PRIMARY KEY,
  name VARCHAR(50),
  is_active TINYINT(1)
);

INSERT INTO employees (id, name, is_active)
VALUES (1, 'John Smith', 1),
       (2, 'Jane Doe', 0),
       (3, 'Bob Johnson', 1);
```

select name, case when is_active is true then name else 'no' end from employees;

Join Function

to combine rows from two or more tables based on a related column between them.
(Inner join,left join,right join)

Example: Creating two tables Customer and Purchase;Shows each purchase for each customer

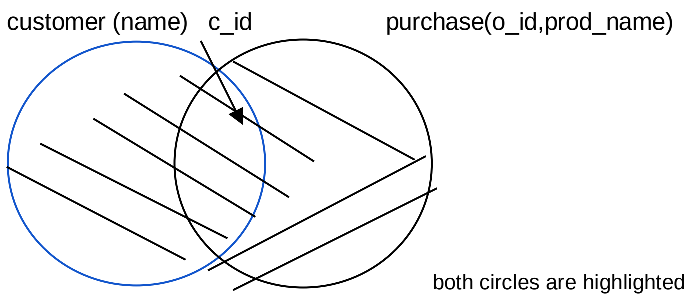
```
create table customer(c_id int not null, name varchar(250));
create table purchase (o_id int , prod_name varchar(250), c_id int);
insert into customer(c_id,name) values(1,'jieshksjd'),
(2,'sdvsdvjieshksjd'),
(3,'wejieshksjd'),
(4,'vsdjiesvsshksjd'),
(13,'jxcieshksjd') ;
insert into purchase (o_id,prod_name,c_id) values(231,'hsagdhsagdhagsd',13),
(322,'fdfadfhsagdhsagdhagsd',23),
(233,'faadfhsagdhsagdhagsd',33),
(234,'dafdfhsagdhsagdhagsd',432),
(236,'jhghfdshsagdhsagdhagsd',2),
```

(235,'sdfsdfdshsagdhsagdhagsd',53);

#to get the customers who have orders in the purchase table using EXISTS.
select name from customer where exists (select * from purchase where customer.c_id=purchase.c_id);

#or using inner join/join & natural join
select name from customer inner join purchase where customer.c_id=purchase.c_id;
select name from customer join purchase where customer.c_id=purchase.c_id;
 # gives the same result inner join or join
select purchase.c_id,o_id from purchase inner join customer where customer.c_id=purchase.c_id; # we should mention the table name when selecting c_id to specify which c_id we need the one in table purchase or table customer
select p.c_id as Customer_Number ,p.o_id as order_Number from purchase p inner join customer c where c.c_id=p.c_id; # define each table using ‘AS’ or without ‘AS’ to make it easier when referring to this table in columns’ names.
select p.c_id as Customer_Number ,p.o_id as order_Number from purchase p inner join customer c on c.c_id=p.c_id; # using inner join table_name **on** instead **where**.
select distinct p.c_id as Customer_Number ,p.o_id as order_Number from purchase p join using customer as c on c.c_id=p.c_id; # using inner join table_name
select c.name as CustomerName ,p.o_id as order_Number from purchase p join customer as c using (c_id); # but we’ve to make sure that column name is the same in both tables for c_id
# natural join used with 2 tables only not more and we've to make sure that the two columns names (C_id) in both tables are matching.(it is based on matching column names)
select c.name as CustomerName ,p.o_id as order_Number from purchase p natural join customer as c; # same result but The natural join eliminates duplicate columns that have matching names in both tables.
Natural Join can also be risky as they can lead to unexpected results if column names are changed or new columns are added to the tables being joined.

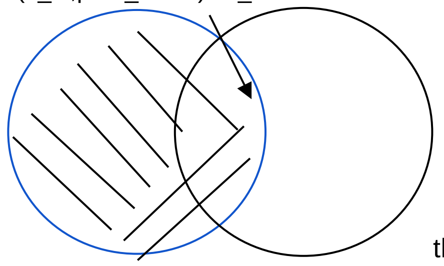
# cross join:matches each row of the first table with each row of the second table resulting in a new table that has all the possible combinations between the two tables.



both circles are highlighted as it gives combinations between the rows and each other (o_id,prod_name,c_id,c_id,name)

select * from purchase cross join customer;
select p.o_id as purchaseNumber,c.name as customername from purchase p cross join customer c on c.c_id=p.c_id; # here it is specified to match the rows with p.c_id=c.c_id only

#using left join or left outer join: it gives all the rows from the left table & matching all the rows in the right table

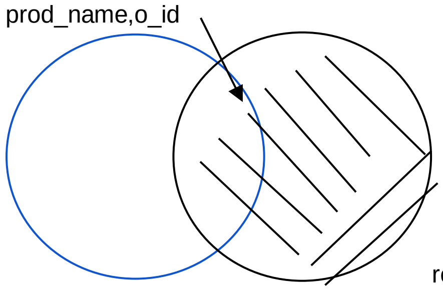


the highlighted part is the ‘name’ column from table customer

Select * from purchase left join customer using (c_id); #it gives all the rows from the table purchase matching the Name column from table customer; either name or NULL.

select c.name as CustomerName ,p.o_id as order_Number from purchase p left join customer as c on c.c_id=p.c_id;
select c.name as CustomerName ,p.o_id as order_Number from purchase p left join customer c using (c_id); # same result

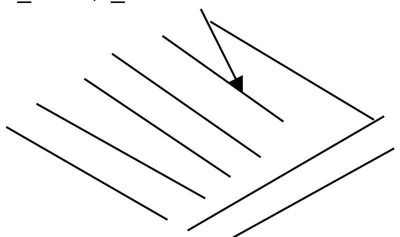
#using right or right outer join;

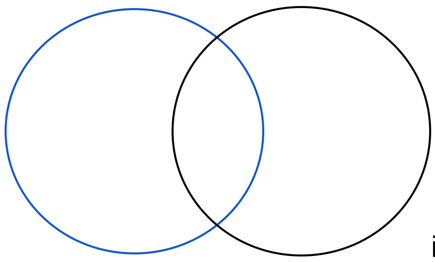


returning all the rows from the right table t & matching the rows in the left table

Select * from purchase right join customer; # it gives customer.c_id,name,o_id,prod_name
select * from customer right join purchase using (c_id);#it gives p.c_id,o_id,prod_name,customer_name
select p.c_id as Customer_Number ,p.o_id as order_Number from purchase p right join customer as c on c.c_id=p.c_id;#all the rows of customer table matches &check the rows of the purchase table

# full join is not supported i guess so we use Union between left join and right join




it gives the total of right join and left join with removing the duplicates automatically

select distinct c.name as CustomerName ,p.o_id as order_Number from purchase p left outer join customer as c on c.c_id=p.c_id union select distinct c.name as CustomerName ,p.o_id as order_Number from purchase p right join customer as c on c.c_id=p.c_id;

select c.name as CustomerName ,p.o_id as order_Number from purchase p left outer join customer as c on c.c_id=p.c_id union all select c.name as CustomerName ,p.o_id as order_Number from purchase p right join customer as c on c.c_id=p.c_id;
# 'union all' doesn't remove the duplicates, it just combines both data.

#Using Self join & GROUP BY:
#SELF JOIN :Showing DUPLICATES FROM SAME TABLE ex: if we want to know in a student table if someone paid twice .
select a.name,b.name from customer a,customer b where a.name=b.name and a.c_id<>b.c_id;# gives two columns name,name.
# we can also use group by instead of self join to know the duplicates.
select user_name,count(*) from student group by user_name; # IT COUNTS THE NAMES IN THE TABLE. IF TWO NAMES ARE THE SAME ,THE COUNT INCREASED FOR THIS NAME
select name,count(name) from customer group by name;

DELETE USING JOIN:

delete customer,purchase from purchase p inner join customer c on p.c_id=c.c_id;
# error here is mentioning the table name instead of the ALIAS.
delete c,p from purchase p inner join customer c on p.c_id=c.c_id;

#using a join to delete data from both the "purchase" and "customer" tables based on a common column, which is the "c_id" column in this case.

#Specifically, the query is deleting all the rows from the "customer" table and the "purchase" table where the "c_id" value in the "customer" table matches the "c_id" value in the "purchase" table.

#This means that any customer who has made a purchase will be deleted from the "customer" table, along with all the corresponding purchase records from the "purchase" table.

Regular Expression
Using 'regexp' function

Example: Creating a table with employee names
CREATE table employee1 (eid int,emp_name varchar(250),income int,city varchar(236));
insert into employee1
values(1,'bfbgcx',7163,'a'),(2,'dsfz',56254,'b'),(3,'zefsf',54545,'a'),(4,'fgfd',5467,'b');

select * from employee1 where emp_name regexp '^[dfg]'; # it means any name that starts with d or f or g
select * from employee1 where emp_name regexp '.*f'; # to give any name that contain 'f'
select * from employee1 where emp_name regexp 'f'; # same result
select * from employee1 where emp_name regexp '^.{5}'; # any name with 5 letters
select * from employee1 where emp_name regexp 'n$'; # any name ends with n
select * from employee1 where emp_name not regexp 'n$'; # it will give all the names that don't end with 'n'

#using or sign as |
select * from employee1 where emp_name rlike 'x$|f$'; # it gives any name ends with x or f

#Using regexp_replace & regexp_substr:

1-first argument:the input string
2-second argument: the regexp specified
3-third argument:the beginning of the search in the string specified
4-forth argument:the occurrence of the pattern
select regexp_substr('roov a n','[a-z]+') as substring; # gives roov
select regexp_substr('roov a n','[a-z]+',1) as substring; # gives roov
select regexp_substr('roov a n','[a-z]+',1,3) as substring; # gives n

Indexing (using previous example)

Indexing: is to create a specific data structure to improve the performance of the data retrieval for quick look ups (ex: B-tree data structure). **How??**
By organising the values in a column or set of columns;called indexed columns.
When retrieving specific data or specific range of data, **the engine uses this index to scan only the rows within that range without having to scan all the rows in the table which will increase the speed of giving the desired output.**

Type 1: B-tree: the column data is stored in a tree-data structure for quick access.

emp_name

key	**Row pointer**
bfbgcx	
dsfz	
zefsf	
fgfd	

Index column- B-tree index with unique values

eid	**emp_name**	**income**	**city**
1	**bfbgcx**	**7163**	**a**
2	**dsfz**	**56254**	**b**
3	**zefsf**	**54545**	**a**
4	**fgfd**	**5467**	**b**

Main table 'employees1'

Here the index column stores the unique values of the rows in 'key' along with a row pointer to the main table to scan.

# INDEXING

show index from employee1 where column_name='emp_name'; # first thing to do before retrieving any data for any organisation

create index name on employee1(emp_name); # creating index of one column to increase the speed process of retrieving the data from this column

create index name_index on employee1(emp_name); # creating index for the same column; however it is not recommended as it takes from the disk space

select * from employee1 where emp_name='dsfz'; # here the engine instead of scanning all the rows in the main table,it will scan only the key 'd to e'

ALTER TABLE employee1 DROP INDEX name;

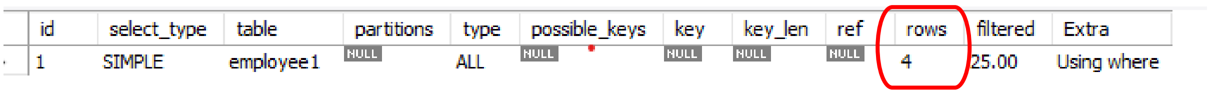
ALTER TABLE employee1 DROP INDEX name_index;

or
Drop index name on employee1;

‘EXPLAIN’ STATEMENT
show information about how MySQL processes the SELECT query used. This information can be useful for optimizing the query and improving its performance

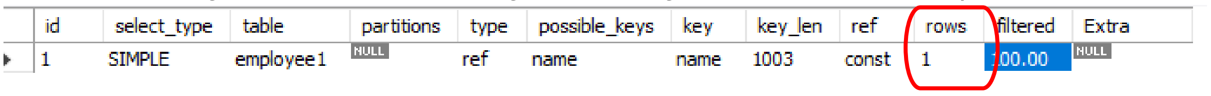
#using EXPLAIN Statement to see how SQL processes the query
EXPLAIN select * from employee1 where emp_name='dsfz'; # to monitor the performance of the database with or without the indexes

When Retrieving the data Before indexing/Partitioning the column,it scans all the row in this column before giving the data



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee1	NULL	ALL	NULL	NULL	NULL	NULL	4	25.00	Using where

When Retrieving the data After indexing/Partitioning the column,it scans specific rows



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee1	NULL	ref	name	name	1003	const	1	100.00	NULL

Note: WE consider indexing columns according to the time the query takes to retrieve the data ; as it should take at most 30sec for retrieving otherwise the column will need to have index.

Type2: Hash index
Used to transform the data into a fixed-size value; numeric value to improve the performance of data retrieving.
Used more with tables containing duplicate values.

Partitioning

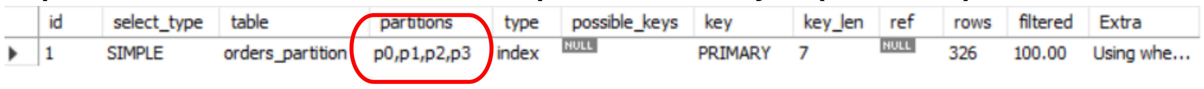
Partitioning is used to optimise the performance of the queries even after indexing.
Example: We have table name ‘orders’ which contain (orderNumber PK,orderDate,requiredDate,shippedDate,status,comments,customerNumber FK)
We need to partition the orderDate according to the years.

select min(orderDate) as min ,max(orderDate) as max from orders; # to check the oldest and newest date of a table for partitioning

```
create table orders_partition
( orderNumber int ,orderDate date,requiredDate date ,
shippedDate date ,
status varchar(15) ,
comments text ,
customerNumber int,primary key(orderNumber,orderDate))
partition by range(year(orderDate))
(
partition p0 values less than (2003),
partition p1 values less than (2004),
partition p2 values less than(2005),
partition p3 values less than (maxvalue)  # maxvalue is the highest possible value for the
partitioning expression,this partition will include all values equal or greater than the partitioning
expression (equal or greater than 2005)
);
insert into orders_partition select * from orders;
describe table orders_partition;
show create table orders_partition; # describe how the partition is made while creating this
table
drop table orders_partition; # to delete the table partitioned.
```

Explain Statement

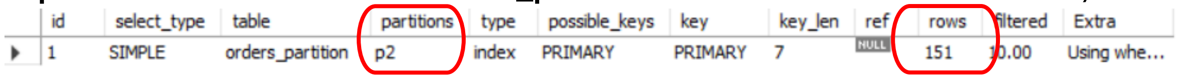
Explain select orderDate from orders_partition where year(orderDate)=2005;



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	orders_partition	p0,p1,p2,p3	index	NULL	PRIMARY	7	NULL	326	100.00	Using whe...

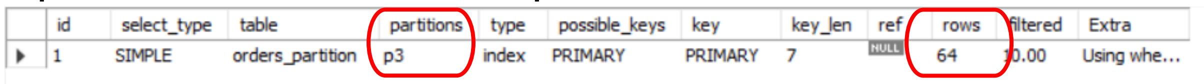
Here because the where clause doesn't match the data type of orderDate which is Date

explain select orderDate from orders_partition where orderDate= '2004-10-01';



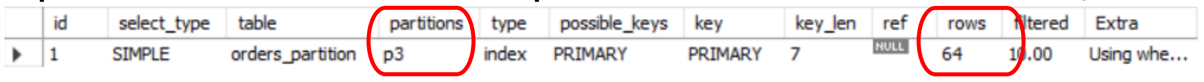
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	orders_partition	p2	index	PRIMARY	PRIMARY	7	NULL	151	10.00	Using whe...

explain select orderDate from orders_partition where orderDate= '2005-10-01';



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	orders_partition	p3	index	PRIMARY	PRIMARY	7	NULL	64	10.00	Using whe...

explain select orderDate from orders_partition where orderDate= '2006-10-01';



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	orders_partition	p3	index	PRIMARY	PRIMARY	7	NULL	64	10.00	Using whe...

Delimiters&Triggers

Delimiters: used to determine the end of the command line and the beginning of the next one using the sign ; by default.However, the user can create its own delimiter by using:
Delimiter // for ex.

Triggers: used for data validation and its types:

1- Audit trail or ‘AFTER INSERT’,’AFTER UPDATE’ AND ‘AFTER DELETE’ trigger: it logs the changes made in the specified table(inserting/updating or deleting data) into a trigger table for data validation.

Example1: We want to log the changes made to a table named ’employees’(id,name,is_Active).

Steps:

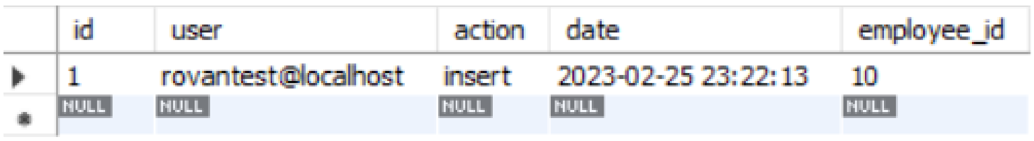
- we create a new table with any name ‘employees_log’ for ex.
- Create a trigger with the same new table name.

```
create table employees_log (
id serial primary key,
user varchar(50) not null,
action varchar(50) not null,
date timestamp not null,
employee_id integer not null);

CREATE TRIGGER employees_log
AFTER INSERT ON employees          # after update,after delete or after insert
FOR EACH ROW
INSERT INTO audit_trail
(user, action, date, employee_id)
VALUES (CURRENT_USER, 'insert', CURRENT_TIMESTAMP, NEW.id);

show tables; # to make sure the trigger table is created
show triggers; # to show all the triggers made

insert into employees values(10,'sarah',true);
Select * from audit_trail;
```



	id	user	action	date	employee_id
▶	1	rovantest@localhost	insert	2023-02-25 23:22:13	10
*	NULL	NULL	NULL	NULL	NULL

2-Referential integrity trigger or ’BEFORE UPDATE’ and ‘BEFORE DELETE’ trigger: it protects the data validation of the foreign key in a specific table by preventing updates or deletes the referenced primary key.

Example2: we have two tables:

table1 ‘Employees’ (employeeNumber PK,FirstName,LastName,Email)

table2 ‘Customers’ (‘CustomerNumber PK,CustomerName,Phone,CiPostalCode,SalesRepEmployeeNumber FK)

Here we want before deleting/updating the ‘employeeNumber’ row in the ‘Employees’ table, we need to check if this PK exists in the ‘Customers’ table for data validation.

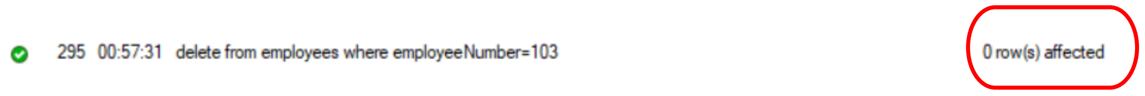
Steps:

- Using Delimiter //
- Creating the trigger to check on the table that has the FK.

NOTE: Delimiter // should be used if we are going to use semicolons within the trigger query.

delimiter //
create trigger check_customers
before delete on employees for each row
begin
   if (select count(*) from customers where salesRepEmployeeNumber=old.employeeNumber)>0
   then signal sqlstate '45000' set message_text= 'cannot delete employeeNumber because it has associated customers';
   END IF;
end //

Delimiter ; # to set back to semicolon ; for the end of the command line.
delete from employees where employeeNumber=103; # here this employeeNumber is existed in the 'Customers' table so it'll not be deleted.



```
295  00:57:31  delete from employees where employeeNumber=103          0 row(s) affected
```

3-Denormalization trigger or 'AFTER INSERT' and 'AFTER UPDATE': used to update the column in a table when a data is logged or changed in another table

Example: we have two tables 'orders1' and 'order_items'
Orders1 (order_id PK, total)
Order_items (order_items_id PK,order_id FK,product_nam,quantity,price)

```
CREATE TABLE orders1(
  order_id INT PRIMARY KEY,
  customer_name VARCHAR(255),
  total DECIMAL(10, 2)
);


CREATE TABLE order_items (
  order_item_id INT PRIMARY KEY,
  order_id INT,
  product_name VARCHAR(255),
  quantity INT,
  price DECIMAL(10, 2),
  CONSTRAINT order_items_ibfk_1 FOREIGN KEY (order_id) REFERENCES
orders1(order_id) ON DELETE CASCADE ON UPDATE CASCADE
);
```

#The ON DELETE CASCADE and ON UPDATE CASCADE clauses when a row in the parent table is deleted/updated,all related rows in the child table(s) should also be deleted/updated automatically.

create trigger orders1_update
after insert on order_items for each row
update orders1 set total= (select sum(quantity*price) from order_items where order_id=orders1.order_id) where order_id=new.order_id;

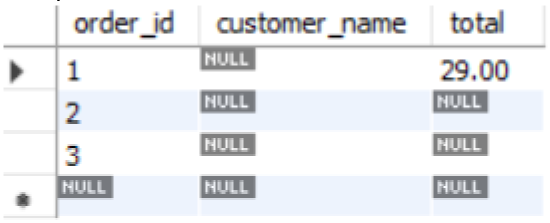
insert into order_items values(1,1,'fd',5,2); # it gives error because the order_id should be logged in the orders1 table first
insert into orders1 (order_id) values (1),(2),(3);

NOTE:

- order_items_id can’t be duplicated as it is PK
- Order_id data should be logged in the parent table first ;orders1 table

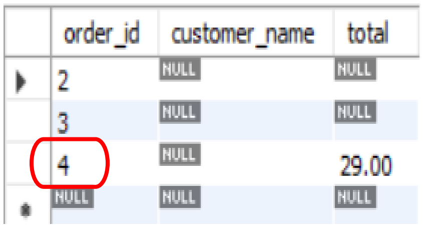
insert into order_items values(1,1,'fd',5,2),(2,1,'fd',2,2),(3,1,'k',5,3),(1,1,'ss',5,2);

#output in orders1 table



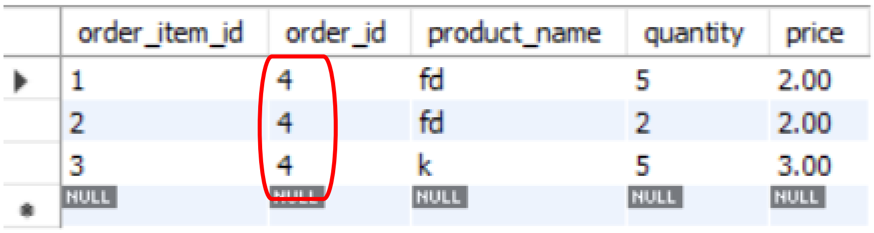
order_id	customer_name	total
1	NULL	29.00
2	NULL	NULL
3	NULL	NULL
NULL	NULL	NULL

#ON DELETE CASCADE ON UPDATE CASCADE
update orders1 set order_id=4 where order_id=1; # here the order_id in order_items table will be changed automatically to 2 because of on delete cascade on update cascade
#output in both tables



order_id	customer_name	total
2	NULL	NULL
3	NULL	NULL
4	NULL	29.00
NULL	NULL	NULL

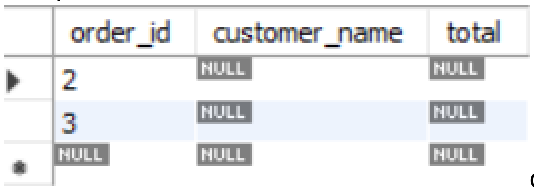
orders1 table



order_item_id	order_id	product_name	quantity	price
1	4	fd	5	2.00
2	4	fd	2	2.00
3	4	k	5	3.00
NULL	NULL	NULL	NULL	NULL

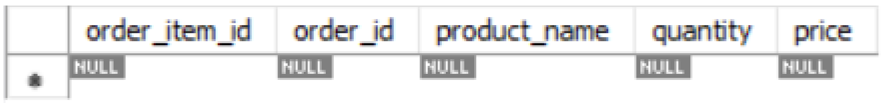
order_items table

delete from orders1 where order_id=4; # it will be deleted also from order_items automatically because of 'ON DELETE CASCADE ON UPDATE CASCADE'
#output for both tables



	order_id	customer_name	total
▶	2	NULL	NULL
	3	NULL	NULL
*	NULL	NULL	NULL

orders1 table



	order_item_id	order_id	product_name	quantity	price
*	NULL	NULL	NULL	NULL	NULL

order_items table

To cover later in this document:

- BCNF,4NF
- Like,not like,between and , any
- Exists case
- Using join for more than two tables
- Aggregation min,max,sum,avg with ‘having’ clause
- Strcmp
- Order by

Case Study

Subjects Discussed:

- ER Schemas & Relational Schemas
- Normalization

ER Schemas types:
One to one relationship
Diagram 1: Many to 1 relationship
Diagram 2: 1 to Many Relationship
Diagram 3: Many to Many Relationship

Normalization:
What is and why normalising databases?
Normalization is splitting complex relationships into simpler forms of data structure.
Normalisation forms (typically only 3 forms are needed): 1NF,2NF,3NF

- To avoid data redundancy
- To avoid alomanies

- To take up less storage space to speed up the retrieval of the data

1NF rules:

- Each cell has one single value
- No repeating groups
- Having a natural key or surrogate key to speed up the data retrieval.

2NF rules:

- Eliminating Partial dependencies;All non-key attributes must fully depend on the primary key of the table

3NF rules:

- all data in a given table is relevant to the object (or entity) described by the table.

Disadvantages of Normalisation:

- More tables mean more indexed keys stored in the memory for fast data retrieval.However many no./overflow of keys causes a full memory which goes later to store the keys in the hard drive which will be useless for an indexed key.
- Longer data retrieval stored in multiple tables
- Longer process of inserting new records as it has to go to multiple tables.

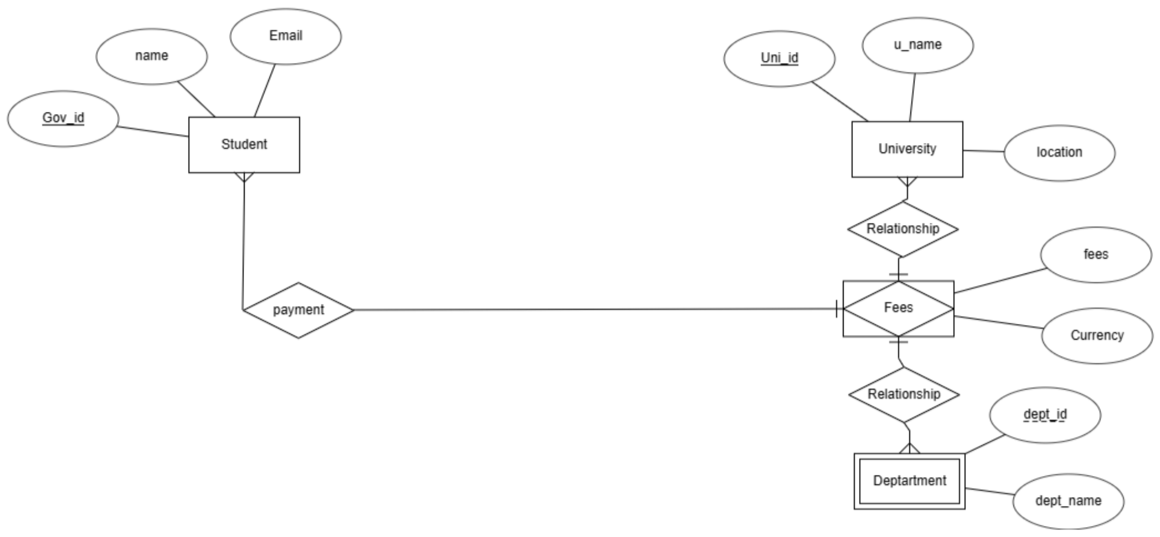
Solution? To go back to 1NF form which is called:

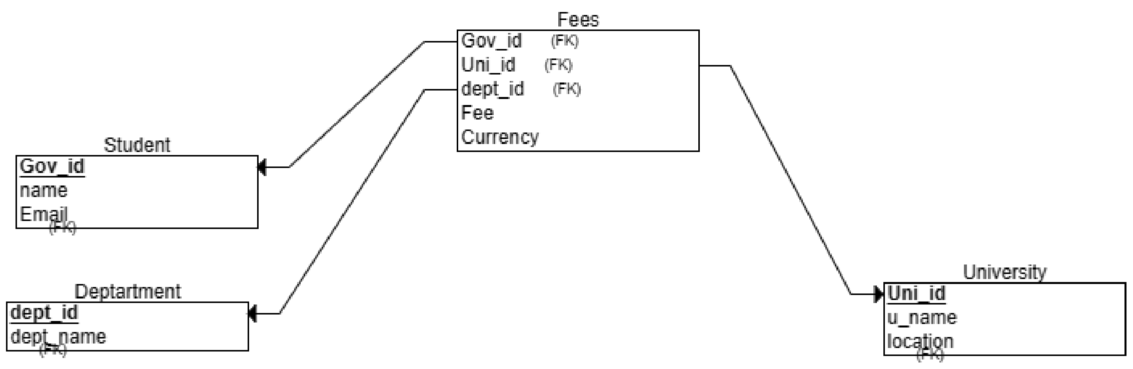
Denormalization

It's best to keep the data structure in both versions; normalized version and denormalized version

Diagram 1 (Many to 1 Relationship):

Case1: it shows each student with its university & Department he is enrolled into, and the fees required depending on the university and dept together.





Many to 1 Relationship Shows each student and his fees depending on the university&Dept he is enrolled into. Uni_id&Gov_id is a composite key;both determine the fees required. Currency(VARCHAR) in $,euro or pound

The main Table:

gov_id	StudentName	Email	University	Department	Fee	Currency
1234	A	abc@ffs	D	engineering	400	$
5678	B	hje@lkgr	E	Economics	300	euro
91011	C	esfk@ksr	j	engineering	500	$

Using Normalization

1NF rules: here the table is already in 1NF and gov_id is used as a natural key.
2NF rules: here the University,Department,Fee and Currency are not dependent fully on the gov_id

gov_id	StudentName	Email
1234	A	abc@ffs
5678	B	hje@lkgr
91011	C	esfk@ksr

Table:Student

id	University	Department	Fee	Currency
1	D	engineering	400	$
2	E	Economics	300	euro
3	j	engineering	500	$

Table:University

3NF rules: -eliminating transitive dependencies
Here the Fees depends on the university and fee.
The currency is transitive dependency

id(PK)	University
1	D
2	E
3	j

Table:University

id(PK)	Department
1	engineering
2	Economics
3	Science

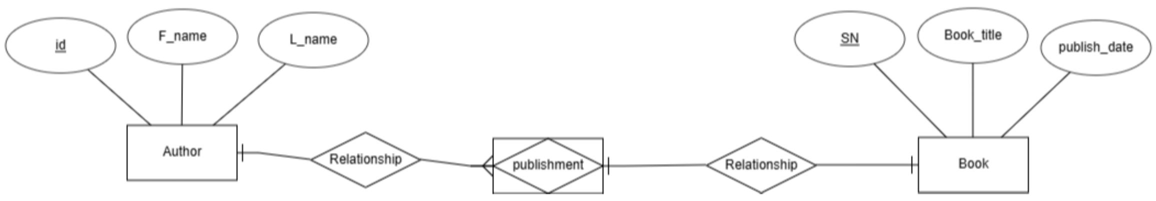
Table:Department

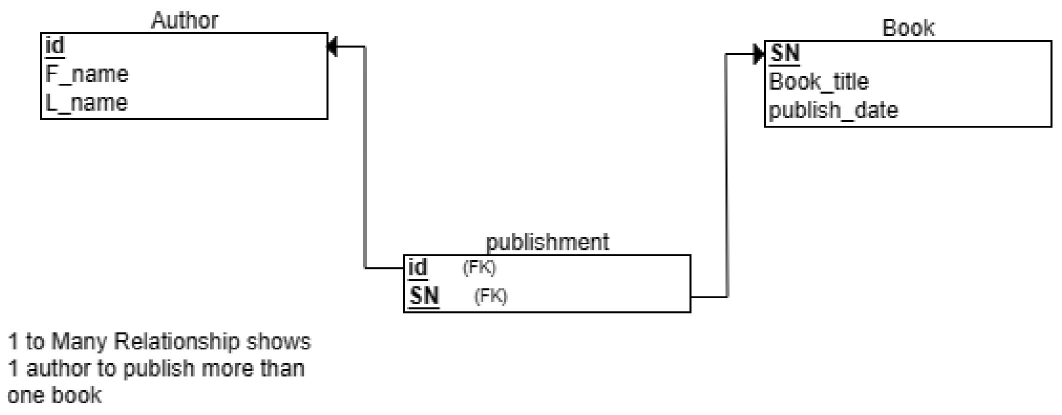
gov_id(FK)	University_id(FK)	Department_id(FK)	Fees	Currency
1	1	1	400	$
2	2	2	300	euro
3	3	1	500	$

Table:University_Department_Fees (university_id&Department_id is composite key)

Diagram 2 (1 to Many Relationship):

Case2: it shows one author to publish more than one book with its special serial number (can be used as PK) according to different publish dates





The main table:

AuthorFirstName	LastName	Book_Title	SerialNo	Publish date
A	C	X,Y,Z	1234,1244,1254	R,S,T
B	D	W,R,S	1235,1236,1247	U,V,H

Using Normalization

1NF rules:

ID (PK/Surrogate key)	AuthorFirstName	LastName	BookTitle	SerialNo	PublishDate
1	A	C	X	1234	R
2	A	C	Y	1244	S
3	A	C	Z	1254	T
4	B	D	W	1235	U
5	B	D	R	1236	V
6	B	D	S	1247	H

2NF rules:

Here the book_title,SerialNo and publish date is independent of the AuthorFirstName and LastName so the main table will be splitted into two tables.

ID(PK)	AuthorFirstName	LastName
1	A	C
2	B	D

Table: Author

SerialNo	BookTitle	PublishDate
1234	X	R
1244	Y	S
1254	Z	T
1235	W	U
1236	R	V
1247	S	H

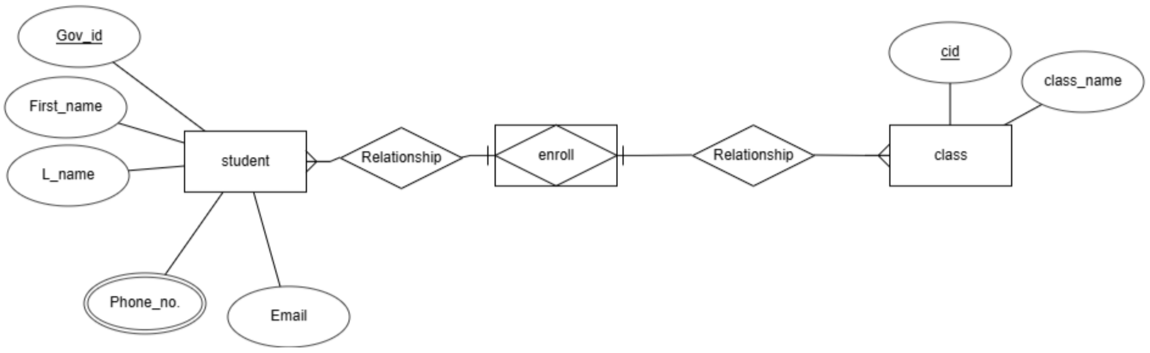
Table:Book

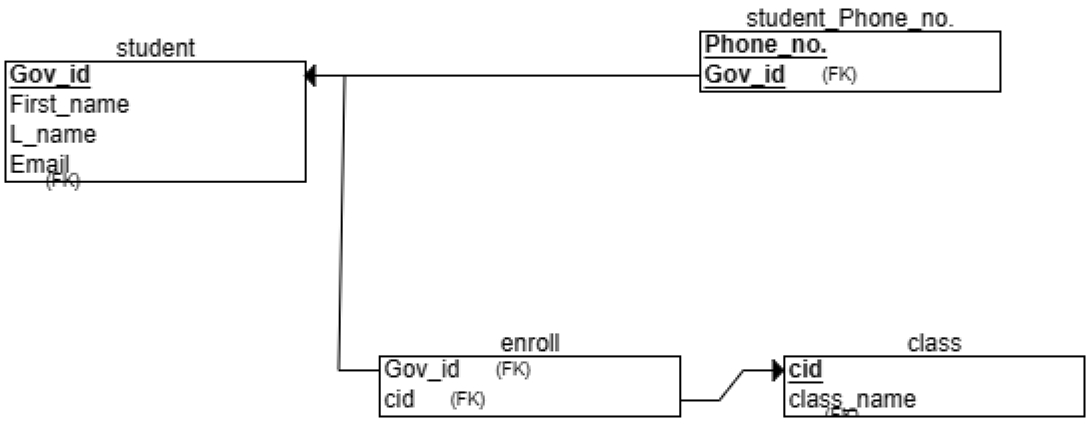
Here the PublishDate and BookTitle are fully dependent on the SerialNo which is considered UNIQUE (used as PK).

Author_id	SerialNo
1	1234
1	1244
1	1254
2	1235
2	1236
2	1247

Table:Author_Publishment

Diagram 3 (Many to Many Relationship):
Case3: it shows the number of students that each student can enroll into many classes.





Note: Phone no. can be Multi-valued and Gov_id used as PK.

The main table for the case:

Student_gov_id	student_name	Phone_no.	email	class_enrolled
12345	A	0171234,017363	aaa@bb	x,y,z
1236	B	017856,0178345	hhh@fde	w,x,y

Using Normalization

1NF rules:

- Each cell has one value
- Avoiding repeating rows

Here the PhoneNo Attribute will be separated with gov_id (that can be used as PK)

PhoneID	gov_id(FK)	PhoneNo
1	12345	0171234
2	12345	017363
3	1236	017856
4	1236	0178345

Table:Student_PhoneNo

Student_gov_id	student_name	email	class_name
12345	A	aaa@bb	x
12345	A	aaa@bb	y
12345	A	aaa@bb	z
1236	B	hhh@fde	w
1236	B	hhh@fde	x
1236	B	hhh@fde	y

2NF rules:

- The table must be in 1NF
- All the non-key attributes should fully depends on the primary key

Here gov_id(goverment id) can be primary key;unique.
As class_name will be put into a separated table,the main table will be split into 2 tables

Student_gov_id(PK)	student_name	email
12345	A	aaa@bb
1236	B	hhh@fde

Table : Student

Class_id (PK)	class_name
1	x
2	y
3	z
4	w

Table: Class

Then Relating the two tables to get the Student_enrollment table (Joint table):

Student_gov_id(FK)	Class_id (FK)
12345	1
12345	2
12345	3
1236	4
1236	1
1236	2

Table: Student_enrollment/enroll