VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Database Management Systems (23CS3PCDBM)

Submitted by

Neha Karthik(1BM24CS184)

in partial fulfilment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2025 to Jan-2026

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Database Management Systems (23CS3PCDBM)" carried out by **Neha Karthik(1BM24CS184),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2022. The Lab report has been approved as it satisfies the academic requirements in respect of a Database Management Systems (23CS3PCDBM) work prescribed for the said degree.

Divya Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

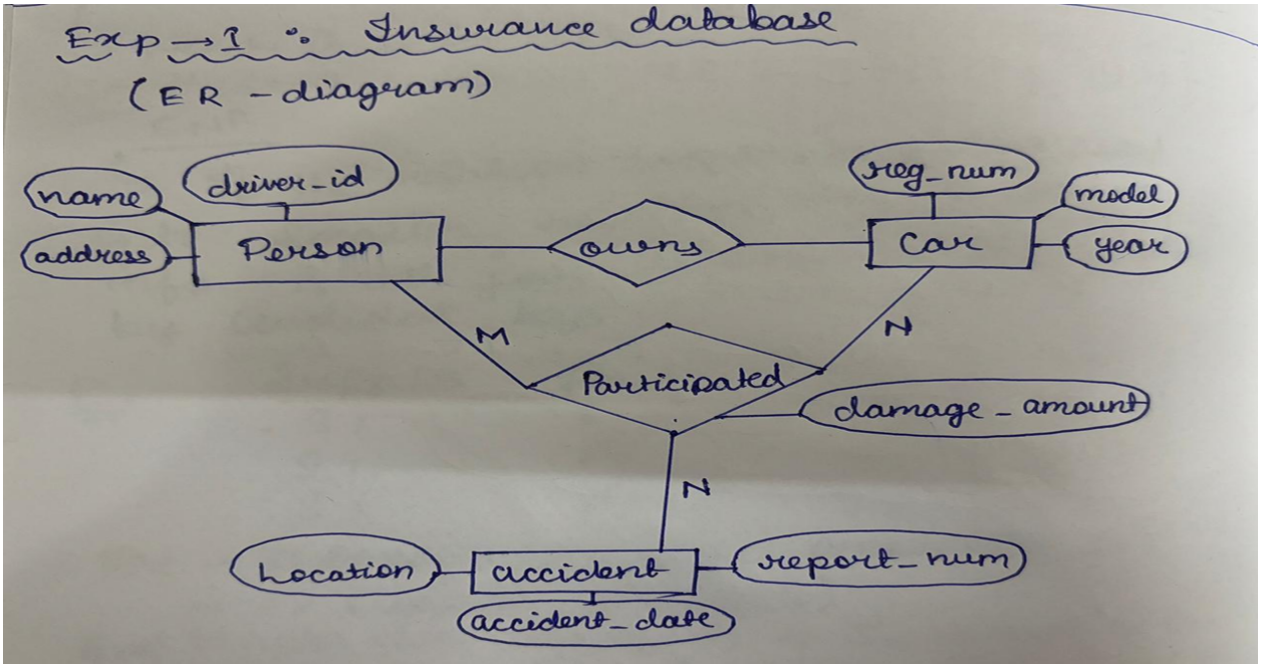
Index

Experiment 1: Insurance Database

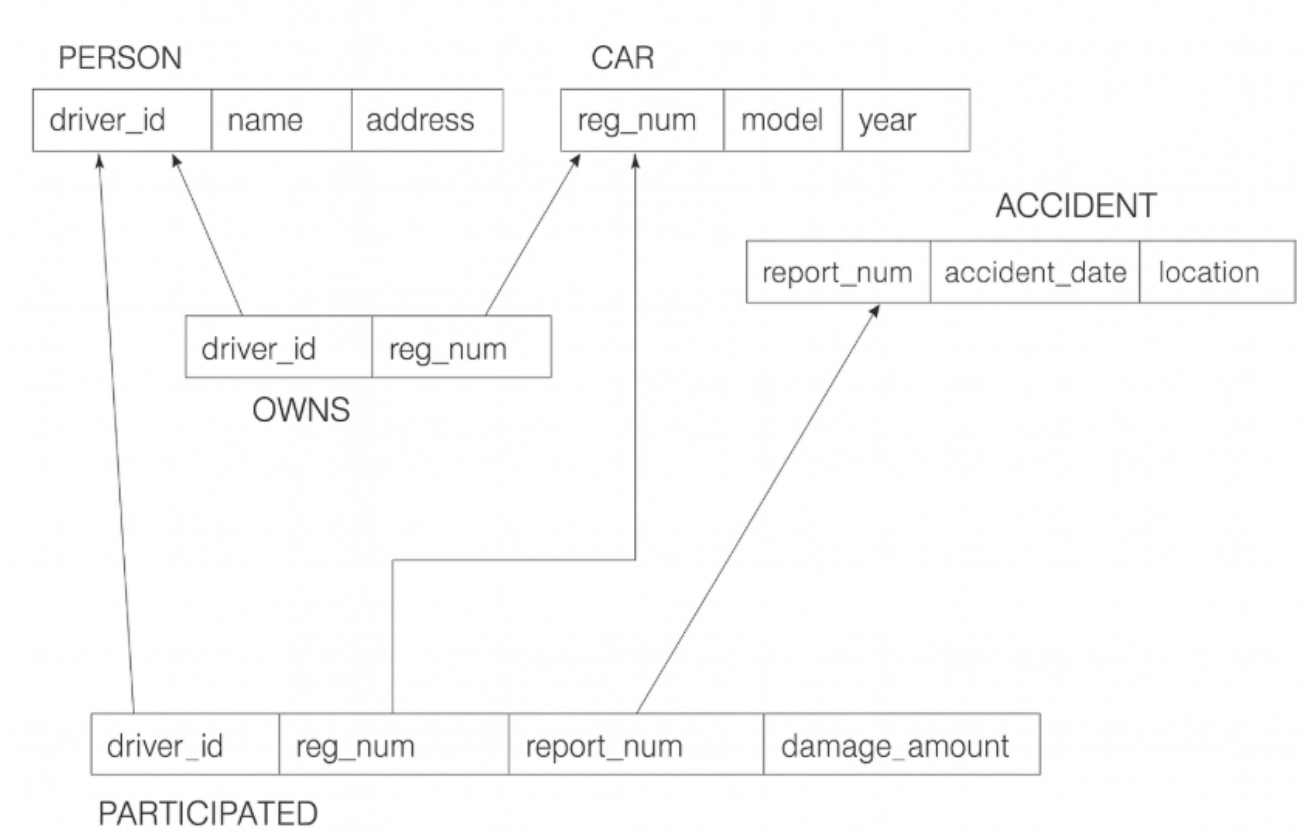
Specification of Insurance Database Application

The insurance database must maintain information about drivers, the cars they own, the accidents reported, and the participation of each driver and car in those accidents. Each driver in the system is uniquely identified by a driver ID, along with their name and address, and each car is uniquely identified by its registration number together with details such as model and manufacturing year. The system must allow storing ownership information that links a driver to one or more cars, while also allowing a car to be linked to one or more drivers if shared ownership occurs; duplicate ownership records for the same driver and car must not exist. Accident information must be stored using a unique report number assigned to each accident, along with the date on which the accident occurred and the location where it happened. Every accident reported in the system must have at least one participating driver and car, and this participation is recorded by linking the driver, the involved car, and the accident report together with the corresponding damage amount for that particular involvement. A participation record must reference an existing driver, an existing car, and an existing accident, and no two participation entries may repeat the same combination of driver, car, and accident report. The database must ensure that damage amounts are non-negative, accident dates are valid calendar dates, and car manufacturing years fall within reasonable limits. It must also preserve referential integrity so that ownership or participation entries cannot exist without valid driver, car, and accident information already present in the system. Deletion policies must prevent removal of drivers or cars that appear in past accident participation records unless historical consistency is preserved through controlled deletion rules or archival mechanisms. The system should maintain accurate links between drivers, cars, and accidents at all times, ensuring reliable retrieval of ownership histories, accident histories, and damage information for administrative, legal, and insurance-related purposes.

Entity Relationship Diagram



Schema Diagram



- PERSON (driver_id: String, name: String, address: String)
- CAR (reg_num: String, model: String, year: int)
- ACCIDENT (report_num: int, accident_date: date, location: String)
- OWNS (driver_id: String, reg_num: String)
- PARTICIPATED (driver_id: String,reg_num: String, report_num: int, damage_amount:

int)

- Create the above tables by properly specifying the primary keys and the foreign keys. -

Enter at least five tuples for each relation

Create database

```
create database insurance;

use insurance;
```

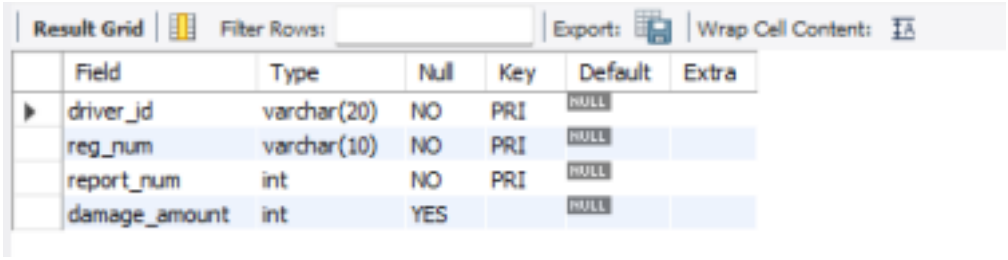
Create table

```
create table insurance.person(

driver_id varchar(20),

name varchar(30),

address varchar(50),

PRIMARY KEY(driver_id)

);

create table insurance.car(

reg_num varchar(15),

model varchar(10),
year int,

PRIMARY KEY(reg_num)

);

create table insurance.owns(

driver_id varchar(20),

reg_num varchar(10),

PRIMARY KEY(driver_id, reg_num),

FOREIGN KEY(driver_id) REFERENCES person(driver_id),

FOREIGN KEY(reg_num) REFERENCES car(reg_num)

);

create table insurance.accident(

report_num int,

accident_date date,

location varchar(50),

PRIMARY KEY(report_num)
```

```
);
create table insurance.participated(
driver_id varchar(20),
reg_num varchar(10),
report_num int,
damage_amount int,
PRIMARY KEY(driver_id,reg_num,report_num),
FOREIGN KEY(driver_id) REFERENCES person(driver_id),
FOREIGN KEY(reg_num) REFERENCES car(reg_num),
FOREIGN KEY(report_num) REFERENCES accident(report_num)
);
```

Structure of the table

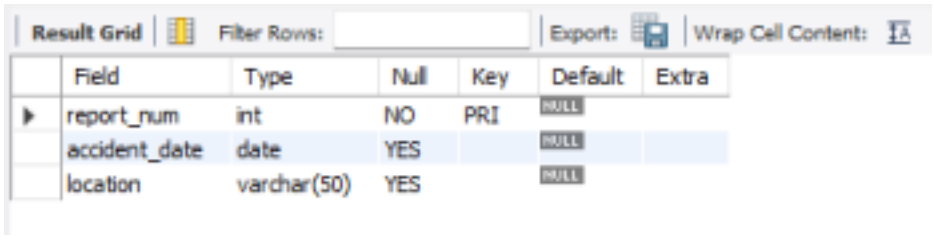
desc person;



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
driver_id	varchar(20)	NO	PRI	NULL	
reg_num	varchar(10)	NO	PRI	NULL	
report_num	int	NO	PRI	NULL	
damage_amount	int	YES		NULL	

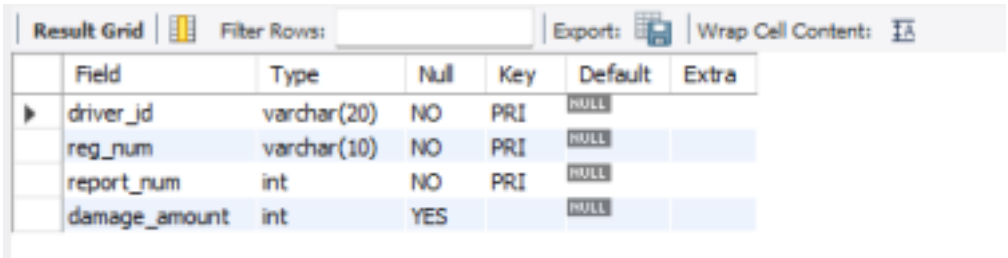
desc accident;



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
report_num	int	NO	PRI	NULL	
accident_date	date	YES		NULL	
location	varchar(50)	YES		NULL	

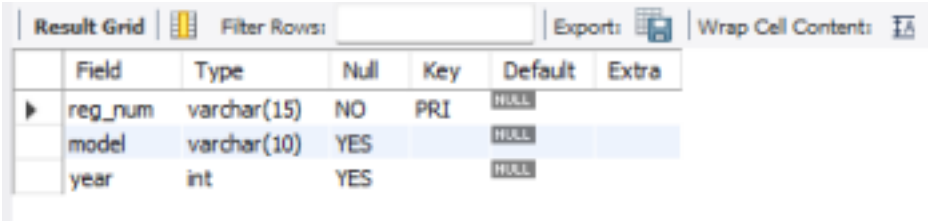
desc participated;



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

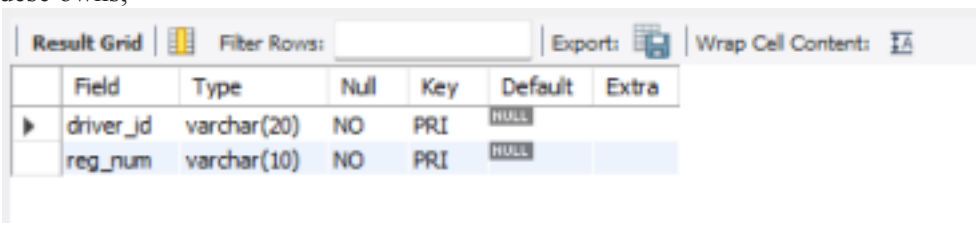
Field	Type	Null	Key	Default	Extra
driver_id	varchar(20)	NO	PRI	NULL	
reg_num	varchar(10)	NO	PRI	NULL	
report_num	int	NO	PRI	NULL	
damage_amount	int	YES		NULL	

desc car;



Field	Type	Null	Key	Default	Extra
reg_num	varchar(15)	NO	PRI	NULL	
model	varchar(10)	YES		NULL	
year	int	YES		NULL	

desc owns;



Field	Type	Null	Key	Default	Extra
driver_id	varchar(20)	NO	PRI	NULL	
reg_num	varchar(10)	NO	PRI	NULL	

Inserting Values to the table

insert into person values("A01","Richard", "Srinivas nagar");

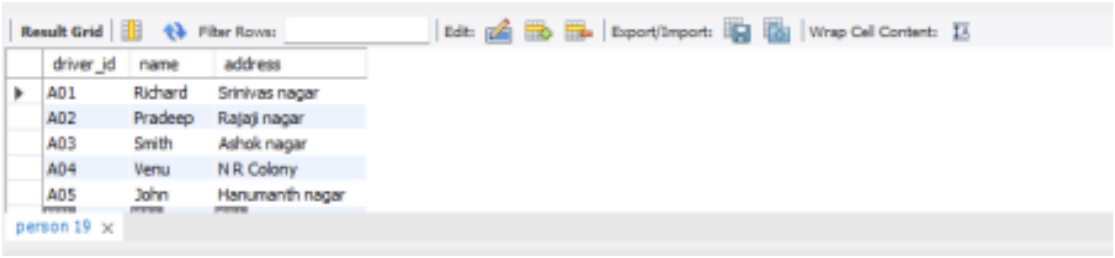
insert into person values("A02","Pradeep", "Rajaji nagar");

insert into person values("A03","Smith", "Ashok nagar");

insert into person values("A04","Venu", "N R Colony");

insert into person values("A05","John", "Hanumanth nagar");

select * from person;



driver_id	name	address
A01	Richard	Srinivas nagar
A02	Pradeep	Rajaji nagar
A03	Smith	Ashok nagar
A04	Venu	N R Colony
A05	John	Hanumanth nagar

person 19

insert into car values("KA052250","Indica", "1990");
insert into car values("KA031181","Lancer", "1957");

insert into car values("KA095477","Toyota", "1998");

insert into car values("KA053408","Honda", "2008");

insert into car values("KA041702","Audi", "2005");

select * from car;

insert into owns values("A01","KA052250");

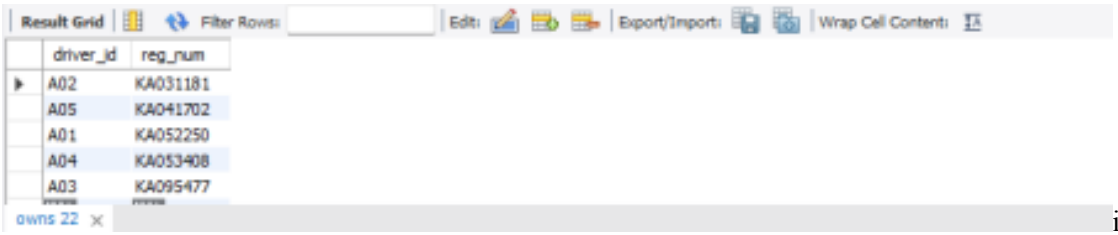
insert into owns values("A02","KA031181");

insert into owns values("A03","KA095477");

insert into owns values("A04","KA053408");

insert into owns values("A05","KA041702");

select * from owns;



driver_id	reg_num
A02	KA031181
A05	KA041702
A01	KA052250
A04	KA053408
A03	KA095477

owns 22

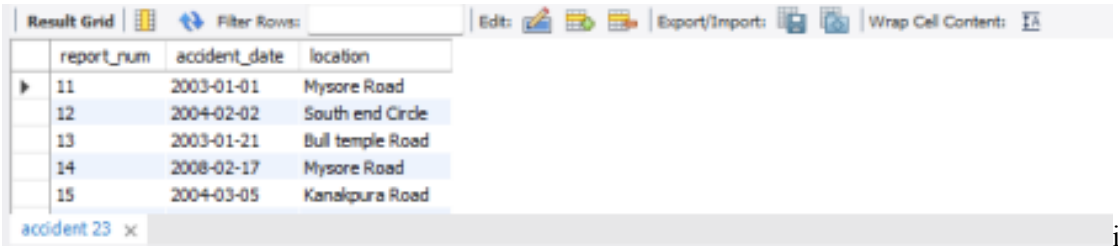
insert into accident values(11,'2003-01-01',"Mysore Road");
insert into accident values(12,'2004-02-02',"South end Circle");

insert into accident values(13,'2003-01-21',"Bull temple Road");

insert into accident values(14,'2008-02-17',"Mysore Road");

insert into accident values(15,'2004-03-05',"Kanakpura Road");

select * from accident;



report_num	accident_date	location
11	2003-01-01	Mysore Road
12	2004-02-02	South end Circle
13	2003-01-21	Bull temple Road
14	2008-02-17	Mysore Road
15	2004-03-05	Kanakpura Road

accident 23

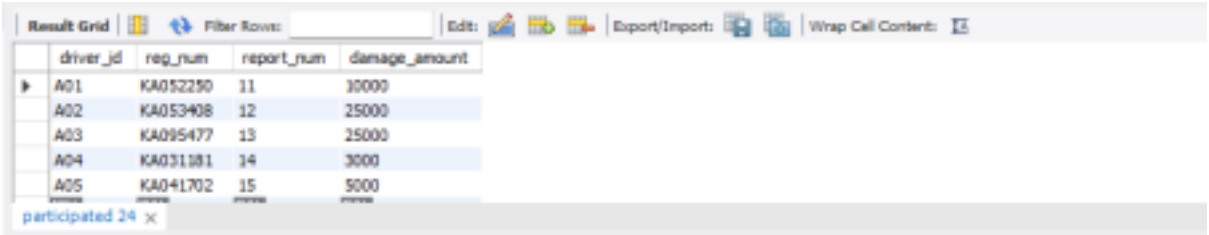
insert into participated values("A01","KA052250",11,10000);
insert into participated values("A02","KA053408",12,50000);

insert into participated values("A03","KA095477",13,25000);

insert into participated values("A04","KA031181",14,3000);

insert into participated values("A05","KA041702",15,5000);

select * from participated;



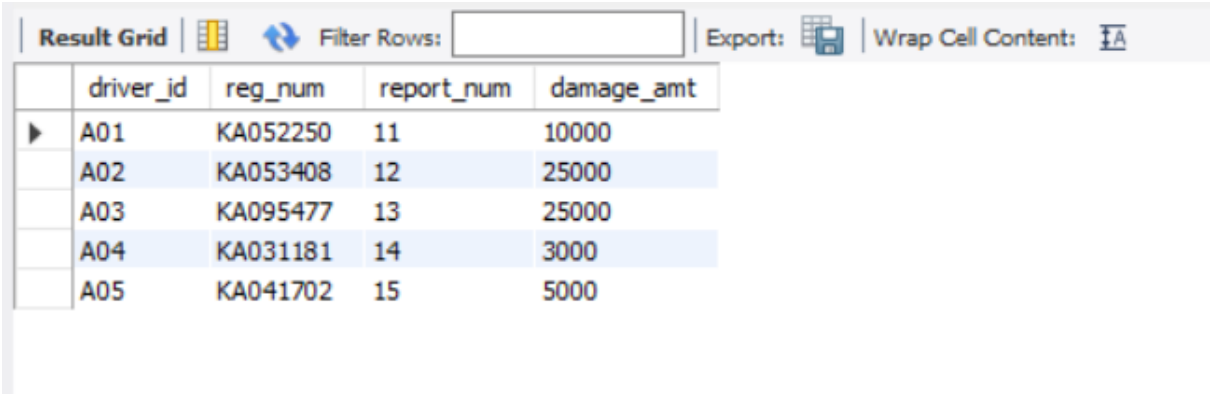
driver_id	reg_num	report_num	damage_amount
A01	KA052250	11	10000
A02	KA053408	12	25000
A03	KA095477	13	25000
A04	KA031181	14	3000
A05	KA041702	15	5000

participated 24

QUERY 1:

1. **Update the damage amount to 25000 for the car with a specific reg-num (example 'KA053408') for which the accident report number was 12.**

update participated set damage_amt=25000 where reg_num='KA053408' and report_num=12;

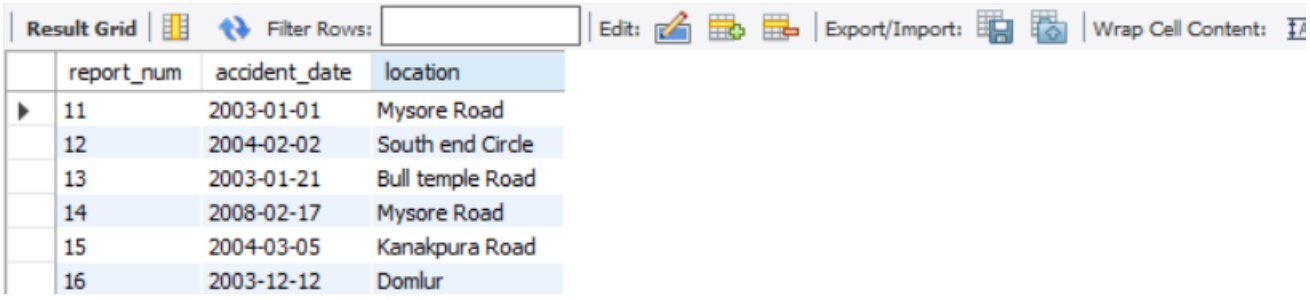


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

driver_id	reg_num	report_num	damage_amt
A01	KA052250	11	10000
A02	KA053408	12	25000
A03	KA095477	13	25000
A04	KA031181	14	3000
A05	KA041702	15	5000

2. **Add a new accident to the database.**

insert into accident values(16,"2003-12-12","Domlur");



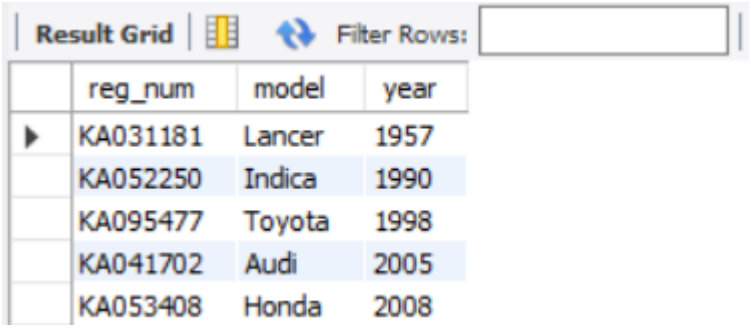
Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

report_num	accident_date	location
11	2003-01-01	Mysore Road
12	2004-02-02	South end Circle
13	2003-01-21	Bull temple Road
14	2008-02-17	Mysore Road
15	2004-03-05	Kanakpura Road
16	2003-12-12	Domlur

QUERY 2:

Display the entire CAR relation in the ascending order of manufacturing year.

```
select * from car order by year asc;
```



Result Grid | Filter Rows:

reg_num	model	year
KA031181	Lancer	1957
KA052250	Indica	1990
KA095477	Toyota	1998
KA041702	Audi	2005
KA053408	Honda	2008

QUERY 3:

Find the number of accidents in which cars belonging to a specific model (Example “Lancer) were involved.

```
select count(report_num) CNT from car c,participated p where c.reg_num=p.reg_num
and model="Lancer";
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

CNT
1

QUERY 4:

Find the total number of people who owned cars that involved in accidents in 2008.

```
select count(distinct driver_id) cnt from participated,accident where
participated.report_num = accident.report_num and accident.accident_date like
'2008%';
```

	CNT
▶	1

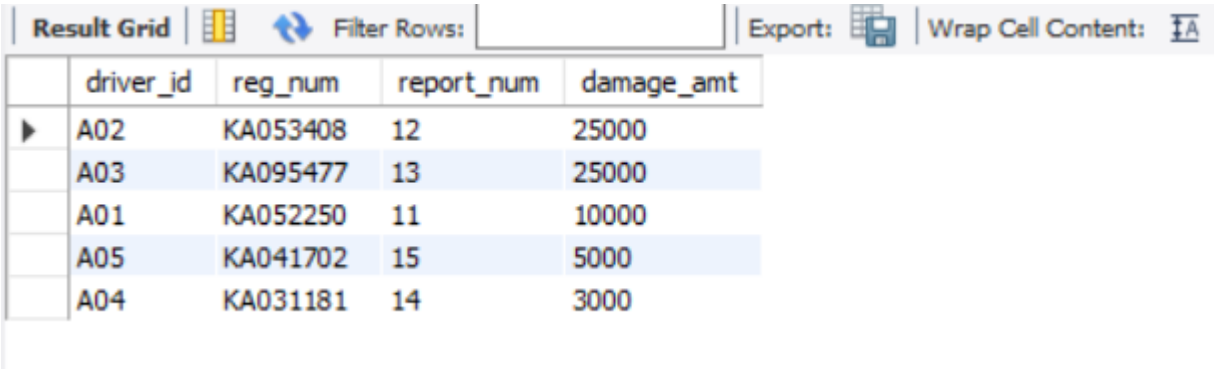
Experiment 2: More Queries on Insurance Database

Queries (Questions and output)

QUERY 1:

List the entire participated relation in the descending order of damage amount.

SELECT * FROM PARTICIPATED ORDER BY DAMAGE_AMT DESC;



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

driver_id	reg_num	report_num	damage_amt
A02	KA053408	12	25000
A03	KA095477	13	25000
A01	KA052250	11	10000
A05	KA041702	15	5000
A04	KA031181	14	3000

QUERY 2:

Find the average damage amount.

SELECT AVG(DAMAGE_AMT) FROM PARTICIPATED;

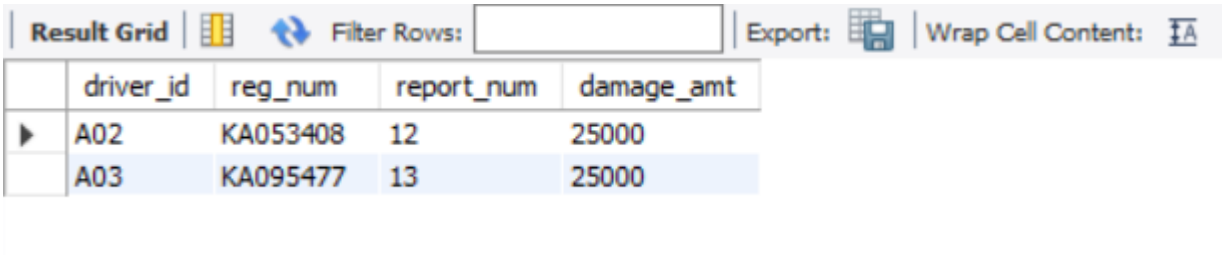
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

AVG(DAMAGE_AMT)
13600.0000

QUERY 3:

Delete the tuple from participated relation whose damage amount is below the average damage amount.

DELETE FROM PARTICIPATED WHERE DAMAGE_AMOUNT<(SELECT AVG (DAMAGE_AMOUNT) FROM PARTICIPATED);



driver_id	reg_num	report_num	damage_amt
A02	KA053408	12	25000
A03	KA095477	13	25000

QUERY 4:

List the name of drivers whose damage is greater than the average damage amount.

SELECT NAME FROM PERSON A, PARTICIPATED B WHERE A.DRIVER_ID = B.DRIVER_ID AND DAMAGE_AMOUNT>(SELECT AVG(DAMAGE_AMOUNT) FROM PARTICIPATED);

name
Pradeep
Smith

Experiment 3: Bank Database

Specification of Bank Database Application

The bank database must maintain information about bank branches, customer accounts, bank customers, and the relationship between customers and the accounts they hold. Each branch in the system is uniquely identified by its branch name and stores additional information such as the city in which the branch is located and the total assets held by the branch. Branch asset values must be non-negative real numbers.

Each bank account in the system is uniquely identified by an account number (accno) and is associated with exactly one branch. The account stores information such as the current balance, which must be a non-negative real value. Every account record must reference an existing branch, ensuring that no account can exist without a valid branch.

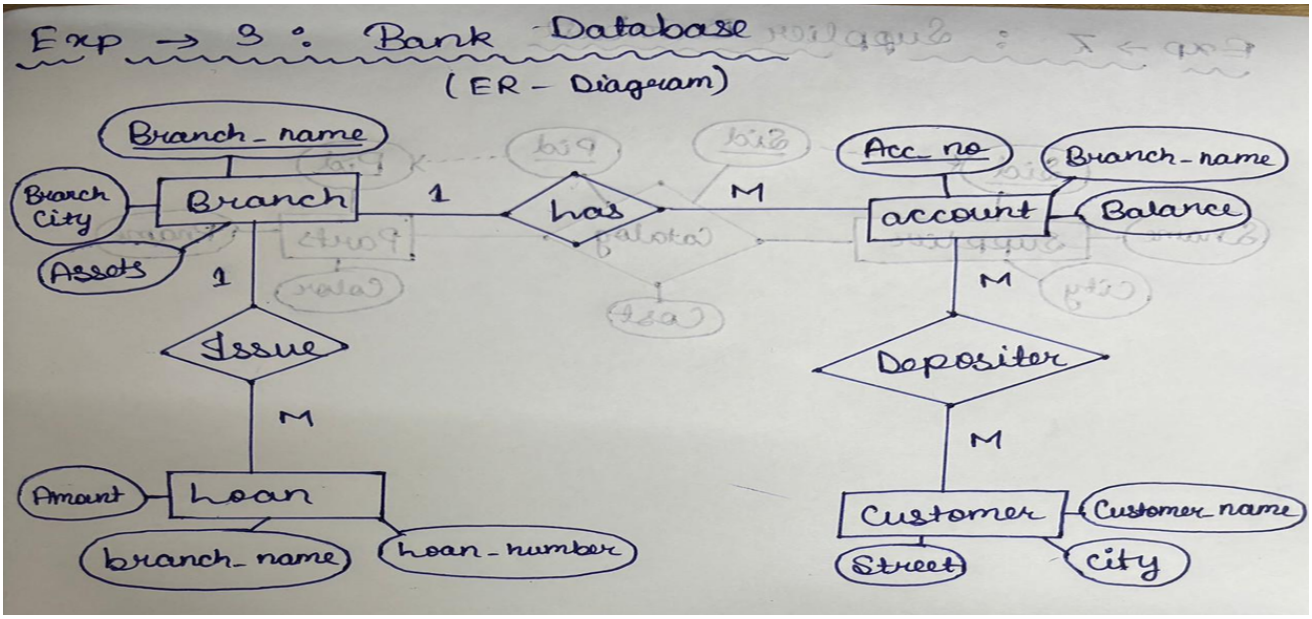
The database must also maintain customer information. Each customer is uniquely identified by the customer name and includes details such as street and city of residence. Customer records must exist independently of account records.

The relationship between customers and accounts is captured through the DEPOSITER relation. This relation links a customer to a bank account, representing ownership or deposit rights. A customer may hold multiple accounts, and an account may be jointly held by multiple customers. Duplicate depositer records for the same customer–account combination must not exist.

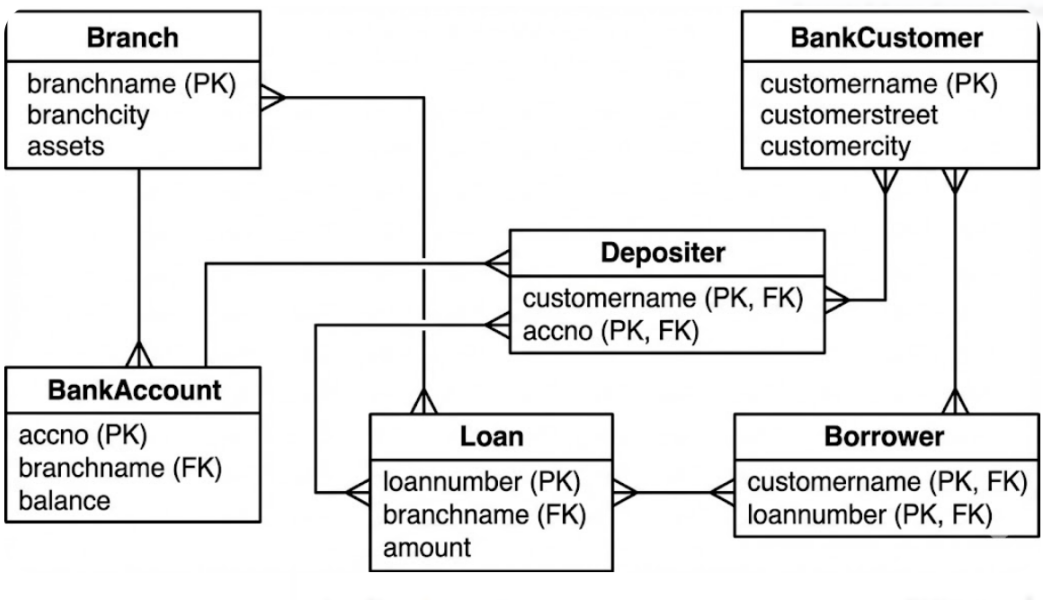
The database must enforce referential integrity so that depositer entries cannot exist unless the referenced customer and account records are already present. Deletion of customers or accounts that are referenced in depositer records must be restricted or handled using controlled cascading or archival policies to preserve account ownership history.

Overall, the system must maintain accurate and consistent relationships between branches, accounts, and customers, allowing reliable retrieval of branch details, customer information, account balances, and ownership relationships for banking operations, auditing, and customer services.

Entity Relationship Diagram



Schema Diagram:



Branch (branch-name: String, branch-city: String, assets: real)
BankAccount(accno: int, branch-name: String, balance: real)
BankCustomer (customer-name: String, customer-street: String, customer-city: String) Depositer(customer-name: String, accno: int)

Database Creation

```
create database if not exists bank;
use bank;
```

Table Creation

```
CREATE TABLE BankAccount (
    accno INT PRIMARY KEY,
    branchname VARCHAR(30),
    balance REAL,
    FOREIGN KEY (branchname) REFERENCES Branch(branchname) ON DELETE CASCADE
);

CREATE TABLE BankCustomer (
    customername VARCHAR(30) PRIMARY KEY,
    customerstreet VARCHAR(30),
    customercity VARCHAR(30)
);
CREATE TABLE Depositer (
    customername VARCHAR(30),
    accno INT,
    PRIMARY KEY (customername, accno),
    FOREIGN KEY (customername) REFERENCES BankCustomer(customername) ON DELETE
CASCADE,
    FOREIGN KEY (accno) REFERENCES BankAccount(accno) ON DELETE CASCADE
);

CREATE TABLE Loan (
    loannumber INT PRIMARY KEY,
    branchname VARCHAR(30),
    amount REAL,
    FOREIGN KEY (branchname) REFERENCES Branch(branchname) ON DELETE CASCADE
);

CREATE TABLE Borrower (
    customername VARCHAR(30),
    loannumber INT,
    PRIMARY KEY (customername, loannumber),
    FOREIGN KEY (customername) REFERENCES BankCustomer(customername) ON DELETE
CASCADE,
    FOREIGN KEY (loannumber) REFERENCES Loan(loannumber) ON DELETE CASCADE
);
```

Structure of the table

desc BankAccount;

cid	name	type	notnull	dflt_value	pk
0	accno	INT	0		1
1	branchname	VARCHAR(30)	0		0
2	balance	REAL	0		0

desc BankCustomer;

cid	name	type	notnull	dflt_value	pk
0	customername	VARCHAR(30)	0		1
1	customerstreet	VARCHAR(30)	0		0
2	customercity	VARCHAR(30)	0		0

desc depositer;

cid	name	type	notnull	dflt_value	pk
0	customername	VARCHAR(30)	0		1
1	accno	INT	0		2

desc Loan;

cid	name	type	notnull	dflt_value	pk
0	loannumber	INT	0		1
1	branchname	VARCHAR(30)	0		0
2	amount	REAL	0		0

desc borrower;

cid	name	type	notnull	dflt_value	pk
0	customername	VARCHAR(30)	0		1
1	loannumber	INT	0		2

Inserting Values to the table

INSERT INTO Branch VALUES('SBI_Chamrajpet', 'Bangalore', 50000);
INSERT INTO Branch VALUES('SBI_ResidencyRoad', 'Bangalore', 10000);
INSERT INTO Branch VALUES('SBI_ShivajiRoad', 'Bombay', 20000);
INSERT INTO Branch VALUES('SBI_ParlimentRoad', 'Delhi', 10000);
INSERT INTO Branch VALUES('SBI_Jantarmantar', 'Delhi', 20000);
select * from branch;

Branch

branchname	branchcity	assets
SBI_Chamrajpet	Bangalore	50000
SBI_ResidencyRoad	Bangalore	10000
SBI_ShivajiRoad	Bombay	20000
SBI_ParlimentRoad	Delhi	10000
SBI_Jantarmantar	Delhi	20000

INSERT INTO BankCustomer VALUES('Avinash', 'Bull_Temple_Road', 'Bangalore');
INSERT INTO BankCustomer VALUES('Dinesh', 'Bannergatta_Road', 'Bangalore');
INSERT INTO BankCustomer VALUES('Mohan', 'NationalCollege_Road', 'Bangalore');
INSERT INTO BankCustomer VALUES('Nikil', 'Akbar_Road', 'Delhi');
INSERT INTO BankCustomer VALUES('Ravi', 'Prithviraj_Road', 'Delhi');
select * from bankcustomer;

BankCustomer

customername	customerstreet	customercity
Avinash	Bull_Temple_Road	Bangalore
Dinesh	Bannergatta_Road	Bangalore
Mohan	NationalCollege_Road	Bangalore
Nikil	Akbar_Road	Delhi
Ravi	Prithviraj_Road	Delhi

```
INSERT INTO BankAccount VALUES(1, 'SBI_Chamrajpet', 2000);
INSERT INTO BankAccount VALUES(2, 'SBI_ResidencyRoad', 5000);
INSERT INTO BankAccount VALUES(3, 'SBI_ShivajiRoad', 6000);
INSERT INTO BankAccount VALUES(4, 'SBI_ParlimentRoad', 9000);
INSERT INTO BankAccount VALUES(5, 'SBI_Jantarmantar', 8000);
INSERT INTO BankAccount VALUES(6, 'SBI_ShivajiRoad', 4000);
INSERT INTO BankAccount VALUES(8, 'SBI_ResidencyRoad', 4000);
INSERT INTO BankAccount VALUES(9, 'SBI_ParlimentRoad', 3000);
INSERT INTO BankAccount VALUES(10, 'SBI_ResidencyRoad', 5000);
INSERT INTO BankAccount VALUES(11, 'SBI_Jantarmantar', 2000);
select * from bankaccount;
```

BankAccount

accno	branchname	balance
1	SBI_Chamrajpet	2000
2	SBI_ResidencyRoad	5000
3	SBI_ShivajiRoad	6000
4	SBI_ParlimentRoad	9000
5	SBI_Jantarmantar	8000
6	SBI_ShivajiRoad	4000
8	SBI_ResidencyRoad	4000
9	SBI_ParlimentRoad	3000
10	SBI_ResidencyRoad	5000
11	SBI_Jantarmantar	2000

```
INSERT INTO Loan VALUES(1, 'SBI_Chamrajpet', 1000);
INSERT INTO Loan VALUES(2, 'SBI_ResidencyRoad', 2000);
INSERT INTO Loan VALUES(3, 'SBI_ShivajiRoad', 3000);
INSERT INTO Loan VALUES(4, 'SBI_ParlimentRoad', 4000);
INSERT INTO Loan VALUES(5, 'SBI_Jantarmantar', 5000);
select * from loan;
```

Loan

loannumber	branchname	amount
1	SBI_Chamrajpet	1000
2	SBI_ResidencyRoad	2000
3	SBI_ShivajiRoad	3000
4	SBI_ParlimentRoad	4000
5	SBI_Jantarmantar	5000

```
INSERT INTO Depositer VALUES('Avinash', 1);
INSERT INTO Depositer VALUES('Dinesh', 2);
INSERT INTO Depositer VALUES('Nikil', 4);
INSERT INTO Depositer VALUES('Ravi', 5);
INSERT INTO Depositer VALUES('Avinash', 8);
INSERT INTO Depositer VALUES('Nikil', 9);
INSERT INTO Depositer VALUES('Dinesh', 10);
INSERT INTO Depositer VALUES('Nikil', 11);
select * from depositer;
```

Depositer

customername	accno
Avinash	1
Dinesh	2
Nikil	4
Ravi	5
Avinash	8
Nikil	9
Dinesh	10
Nikil	11

Queries:

Query 1:

Display the branch name and assets from all branches in lakhs of rupees and rename the assets column to 'assets in lakhs'.

select branch_name, (assets / 100000) as `assets in lakhs` from branch;

branchname	assets in lakhs
SBI_Chamrajpet	0.5
SBI_ResidencyRoad	0.1
SBI_ShivajiRoad	0.2
SBI_ParlimentRoad	0.1
SBI_Jantarmantar	0.2

Query 2:

Find all the customers who have at least two accounts at the same branch (ex. SBI_ResidencyRoad).

SELECT d.customername, b.branchname, COUNT(*) AS num_accounts FROM Depositer d JOIN BankAccount b ON d.accno = b.accno GROUP BY d.customername, b.branchname HAVING COUNT(*) >= 2;

customername	branchname	num_accounts
Dinesh	SBI_ResidencyRoad	2
Nikil	SBI_ParlimentRoad	2

Query 3:

Create a view which gives each branch the sum of the amount of all the loans at the branch.

CREATE VIEW Branch_Total_Loan AS SELECT branchname, SUM(amount) AS Total_Loan FROM Loan
GROUP BY branchname;
SELECT * FROM Branch_Total_Loan ;

branchname	Total_Loan
SBI_Chamrajpet	1000
SBI_Jantarmantar	5000
SBI_ParlimentRoad	4000
SBI_ResidencyRoad	2000
SBI_ShivajiRoad	3000

Experiment 3: More Queries on Bank Database

Queries:

Query 1:

Find all the customers who have an account at all the branches located in a specific city (Ex. Delhi).

```
SELECT DISTINCT d.customername FROM Depositer d WHERE NOT EXISTS (
SELECT b.branchname FROM Branch b WHERE b.branchcity = 'Delhi' AND b.branchname
NOT IN ( SELECT a.branchname FROM Depositer d2 JOIN BankAccount a ON d2.accno =
a.accno WHERE d2.customername = d.customername ));
```

customername
Nikil

Query 2:

Find all customers who have a loan at the bank but do not have an account.

```
SELECT customername FROM Borrower WHERE customername NOT IN (
SELECT customername FROM Depositer );
```

customername
Mohan

Query 3:

Find all customers who have both an account and a loan at the Bangalore branch.

```
SELECT DISTINCT d.customername FROM Depositer d JOIN BankAccount a ON d.accno
= a.accno JOIN Borrower b ON d.customername = b.customername JOIN Loan l ON
b.loannumber = l.loannumber WHERE a.branchname IN ( SELECT branchname FROM
Branch WHERE branchcity = 'Bangalore' ) AND l.branchname IN ( SELECT branchname
FROM Branch WHERE branchcity = 'Bangalore' );
```

customername
Avinash
Dinesh

Query 4:

Find the names of all branches that have greater assets than all branches located in Bangalore.

```
SELECT branchname FROM Branch WHERE assets > ALL (
SELECT assets FROM Branch WHERE branchcity = 'Bangalore' );
```

branchname

Query 5:

Demonstrate how you delete all account tuples at every branch located in a specific city (Ex. Bombay)

```
DELETE FROM BankAccount WHERE branchname IN ( SELECT branchname FROM
Branch WHERE branchcity = 'Bombay' );
```

Query 6:

Update the Balance of all accounts by 5%

```
update bankaccount set balance = balance * 1.05;
```

Experiment 5: Employee Database

Specification of Employee Database Application

The employee database must maintain comprehensive information about employees, the departments they belong to, the projects they work on, and the incentives they receive. Each employee in the system is uniquely identified by an employee number (EMPNO), along with associated attributes such as employee name, hire date, salary, and department number. The system must also support a managerial hierarchy where an employee may act as a manager for other employees, represented through a recursive relationship in which the manager number (MGR_NO) references another existing employee.

Each department in the system is uniquely identified by a department number (DEPTNO) and stores additional information such as department name and department location. Every employee must be associated with a valid department, ensuring that no employee record can exist without a corresponding department already defined in the system. Referential integrity must be enforced so that department information cannot be removed while employees are still assigned to it.

The database must store project-related information, where each project is uniquely identified by a project number (PNO), along with its name and location. Employees may be assigned to one or more projects, and each project may have one or more employees working on it. This many-to-many relationship is represented through an ASSIGNED-TO association, which records the employee number, project number, and the job role performed by the employee in that project. Duplicate assignment records for the same employee–project combination must not exist.

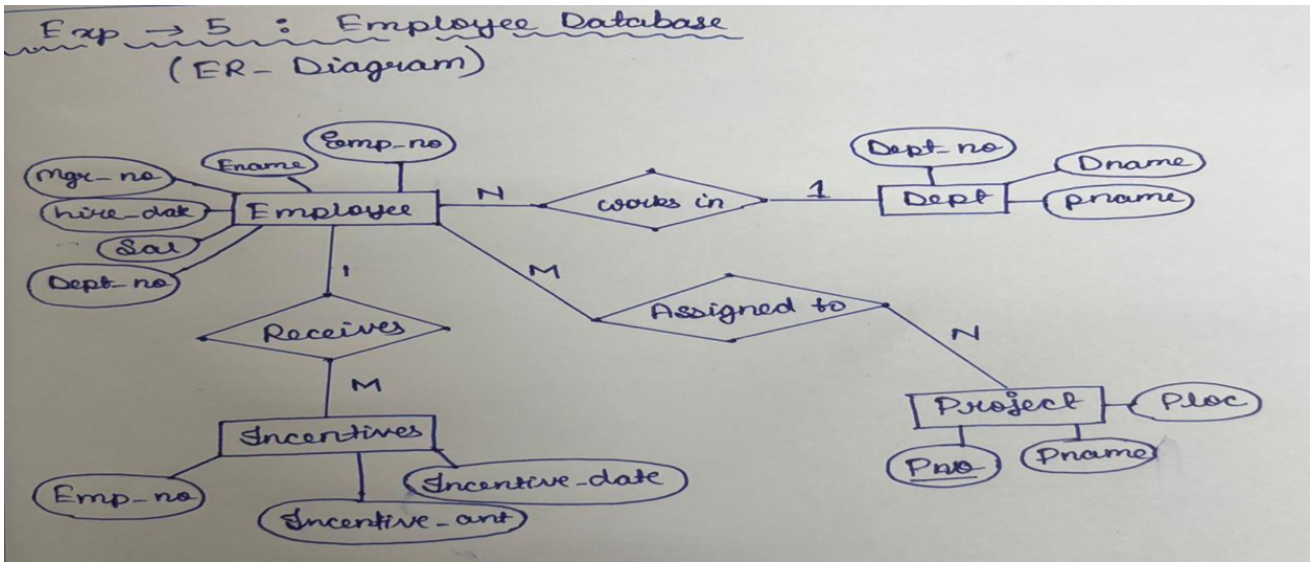
Incentive information must also be maintained for employees. Each incentive record is associated with a valid employee and is uniquely identified by the combination of employee number and incentive date. Additional details such as the incentive amount are stored. An employee may receive multiple incentives over time, but every incentive record must reference an existing employee. Incentive amounts must be non-negative, and incentive dates must be valid calendar dates.

The database must ensure that all foreign key relationships are properly enforced. Assignment records cannot exist unless both the referenced employee and project are present in the system, and incentive records cannot exist without a valid employee. Similarly, employees must reference valid departments, and managerial relationships must reference valid employee records.

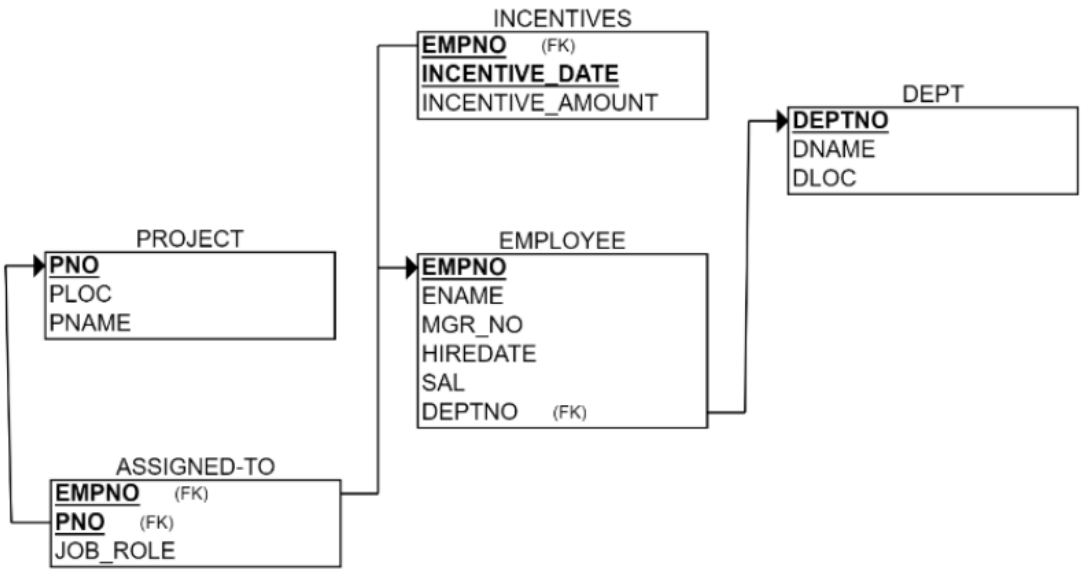
Deletion policies must preserve data consistency and historical correctness. Employees who are referenced as managers, assigned to projects, or associated with incentive records should not be deleted unless appropriate cascading rules or archival mechanisms are applied. Projects and departments should not be removed while they are still referenced by assignment or employee records, respectively.

Overall, the system must maintain accurate and consistent relationships between employees, departments, projects, and incentives, enabling reliable retrieval of employee details, departmental structures, project assignments, managerial hierarchies, and incentive histories for administrative, organizational, and payroll-related purposes.

ER Diagram


Exp → 5 : Employee Database
(ER - Diagram)
Emp_no
Ename
Mgr_no
hire_dat
Sal
Dept_no
Employee
N
works in
1
Dept_no
Dname
Dept
pname
1
M
Receives
Assigned to
N
M
Incentives
Project
Ploc
Pno
Pname
Emp_no
Incentive_date
Incentive_amt

Schema Diagram



EMPLOYEE (empno: int, ename: String, mgr_no: int, hiredate: date, sal: int, deptno: int)
DEPT (deptno: int, dname: String, dloc: String)
PROJECT (pno: int, pname: String, ploc: String)
INCENTIVES (empno: int, incentive_date: date, incentive_amount: int)
ASSIGNED_TO (empno: int, pno: int, job_role: String)

Create the above tables by properly specifying the primary keys and the foreign keys.
Enter at least five tuples for each relation.

Create Database

```
create database employee;
use EMPLOYEE;
```

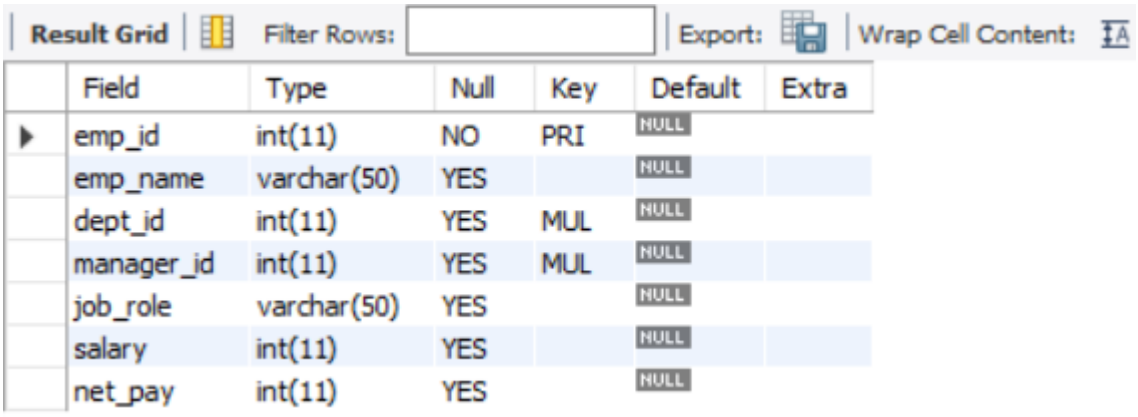
Table Creation

```
create table department (
    dept_id int primary key,
    dept_name varchar(50),
    dept_location varchar(50),
    top_manager_id int
);

create table employee (
    emp_id int primary key,
    emp_name varchar(50),
    dept_id int,
    manager_id int,
    job_role varchar(50),
    salary int,
    net_pay int,
    foreign key (dept_id) references department(dept_id),
    foreign key (manager_id) references employee(emp_id)
);

create table project (
    project_id int primary key,
    project_name varchar(50),
    project_location varchar(50)
);
create table assigned_to (
    emp_id int,
    project_id int,
    job_role varchar(50),
    primary key (emp_id, project_id),
    foreign key (emp_id) references employee(emp_id),
    foreign key (project_id) references project(project_id)
);
create table incentives (
    emp_id int,
    incentive_amount int,
    incentive_date date,
```

```
    foreign key (emp_id) references employee(emp_id)
);
```

Structure of the Table

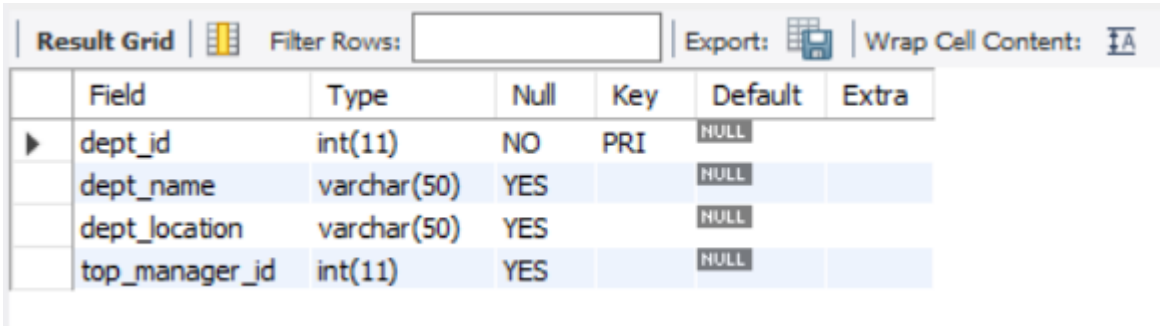
DESC EMPLOYEE;



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
emp_id	int(11)	NO	PRI	NULL	
emp_name	varchar(50)	YES		NULL	
dept_id	int(11)	YES	MUL	NULL	
manager_id	int(11)	YES	MUL	NULL	
job_role	varchar(50)	YES		NULL	
salary	int(11)	YES		NULL	
net_pay	int(11)	YES		NULL	

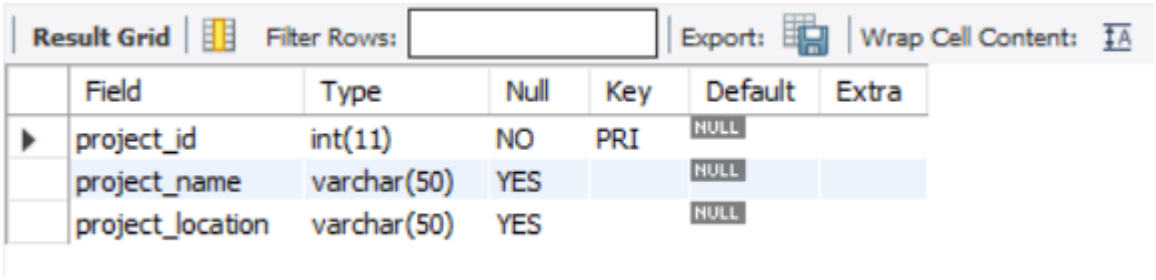
DESC DEPARTMENT;



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
dept_id	int(11)	NO	PRI	NULL	
dept_name	varchar(50)	YES		NULL	
dept_location	varchar(50)	YES		NULL	
top_manager_id	int(11)	YES		NULL	

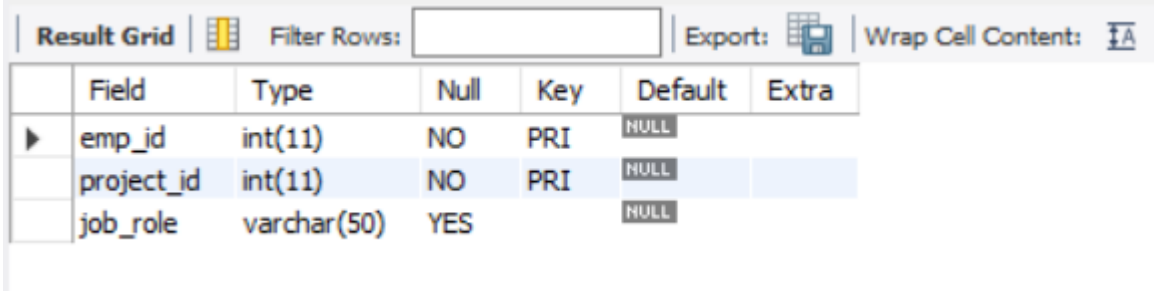
DESC PROJECT;



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
project_id	int(11)	NO	PRI	NULL	
project_name	varchar(50)	YES		NULL	
project_location	varchar(50)	YES		NULL	

DESC ASSIGNED_TO;



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
emp_id	int(11)	NO	PRI	NULL	
project_id	int(11)	NO	PRI	NULL	
job_role	varchar(50)	YES		NULL	

DESC INCENTIVES;

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
emp_id	int(11)	YES	MUL	NULL	
incentive_amount	int(11)	YES		NULL	
incentive_date	date	YES		NULL	

Inserting Values to the table

insert into department values (10, 'HR', 'Bengaluru', 1), (20, 'Finance', 'Mumbai', 2),
(30, 'Sales', 'Hyderabad', 3), (40, 'IT', 'Mysuru', 4), (50, 'Admin', 'Delhi', 5);
select * from department;

Result Grid | Filter Rows: | Edit:

dept_id	dept_name	dept_location	top_manager_id
10	HR	Bengaluru	1
20	Finance	Mumbai	2
30	Sales	Hyderabad	3
40	IT	Mysuru	4
50	Admin	Delhi	5

insert into employee values
(1, 'Anish', 10, null, 'Manager', 70000, 68000),
(2, 'Rahul', 10, 1, 'Executive', 40000, 38000),
(3, 'Sita', 20, 2, 'Clerk', 25000, 24000),
(4, 'Karan', 10, 1, 'Analyst', 42000, 41000),
(5, 'Pooja', 30, 3, 'Associate', 28000, 27000);

Result Grid | Filter Rows: | Edit: | Export/Import:

emp_id	emp_name	dept_id	manager_id	job_role	salary	net_pay
1	Anish	10	NULL	Manager	70000	68000
2	Rahul	10	1	Executive	40000	38000
3	Sita	20	2	Clerk	25000	24000
4	Karan	10	1	Analyst	42000	41000
5	Pooja	30	3	Associate	28000	27000

insert into project values
(101, 'Alpha', 'Bengaluru'),
(102, 'Beta', 'Mumbai'),
(103, 'Gamma', 'Hyderabad'),
(104, 'Delta', 'Mysuru'),
(105, 'Omega', 'Delhi');

Result Grid | Filter Rows: | Edit:

project_id	project_name	project_location
101	Alpha	Bengaluru
102	Beta	Mumbai
103	Gamma	Hyderabad
104	Delta	Mysuru
105	Omega	Delhi

```
insert into assigned_to values
(1, 101, 'Lead'),
(2, 101, 'Support'),
(3, 102, 'Assistant'),
(4, 101, 'Analyst'),
(5, 103, 'Coordinator');
```

Result Grid | Filter Rows:

emp_id	project_id	job_role
1	101	Lead
2	101	Support
3	102	Assistant
4	101	Analyst
5	103	Coordinator

```
insert into incentives values
(1, 5000, '2019-02-10'),
(2, 3000, '2019-02-12'),
(3, 7000, '2019-02-15'),
(4, 4500, '2019-02-20'),
(5, 2000, '2019-03-10');
```

Result Grid | Filter Rows:

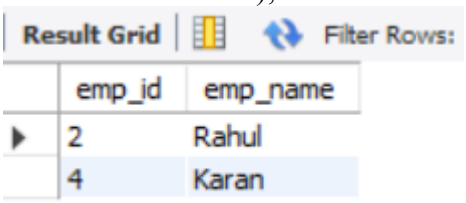
emp_id	incentive_amount	incentive_date
1	5000	2019-02-10
2	3000	2019-02-12
3	7000	2019-02-15
4	4500	2019-02-20
5	2000	2019-03-10

Queries

Query 1:

Retrieve the employee numbers of all employees who work on project located in Bengaluru, Hyderabad, or Mysuru

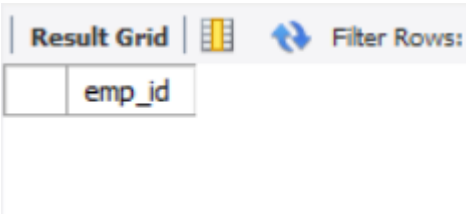
```
select distinct a.emp_id from assigned_to a join project p on a.project_id =
p.project_id where p.project_location in ('BENGALURU', 'HYDERABAD',
'MYSURU');
```



Query 2:

Get Employee ID's of those employees who didn't receive incentives.

```
select e.emp_id from employee e left join incentives i on e.emp_id = i.emp_id where
i.emp_id is null;
```



Query 3:

Write a SQL query to find the employees name, number, dept, job_role, department location and project location who are working for a project location same as his/her department location.

```
select e.emp_name, e.emp_id, e.dept_id, e.job_role, d.dept_location,
p.project_location from employee e join department d on e.dept_id =
d.dept_id join assigned_to a on e.emp_id = a.emp_id join project p on
a.project_id = p.project_id where d.dept_location = p.project_location;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

emp_name	emp_id	dept_id	job_role	dept_location	project_location
Anish	1	10	Manager	Bengaluru	Bengaluru
Rahul	2	10	Executive	Bengaluru	Bengaluru
Sita	3	20	Clerk	Mumbai	Mumbai
Karan	4	10	Analyst	Bengaluru	Bengaluru
Pooja	5	30	Associate	Hyderabad	Hyderabad

Experiment 6: More Queries on Employee Database

Query 1:

List the name of the managers with the maximum employees

```
select m.emp_name from employee m join employee e on m.emp_id = e.manager_id group by m.emp_id, m.emp_name order by count(e.emp_id);
```

Result Grid | Filter Rows:

emp_name
Rahul
Sita
Anish

Query 2:

Display those managers name whose salary is more than average salary of his Employee.

```
select m.emp_name from employee m join employee e on m.emp_id = e.manager_id group by m.emp_id, m.emp_name order by count(e.emp_id);
```

Result Grid | Filter Rows:

emp_name
Anish
Rahul

Query 3:

SQL Query to find the name of the top level manager of each department

```
select m.emp_name from employee m join employee e on m.emp_id =
e.manager_id group by m.emp_id, m.emp_name, m.salary
having m.salary > avg(e.salary);
```

Result Grid | Filter Rows:

	emp_name	dept_id
▶	Anish	10

Query 4:

SQL Query to find the employee details who got second maximum incentive in January 2019.

```
select e.* from employee e join incentives i on e.emp_id =
i.emp_id where month(i.incentive_date) = 1 and
year(i.incentive_date) = 2019 order by i.incentive_amount desc
limit 1 offset 1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Conte

	emp_id	emp_name	dept_id	manager_id	job_role	salary	net_pay
▶	1	Anish	10	NULL	Manager	70000	68000

Query 5:

Display those employees who are working in the same dept where his manager is work.

```
select e.emp_id, e.emp_name from employee e join employee m on e.manager_id =
m.emp_id where e.dept_id = m.dept_id;
```

Result Grid | Filter Rows:

	emp_id	emp_name
▶	2	Rahul
	4	Karan

Experiment 7: Supplier Database

Specification of Supplier Database Application

The supplier–parts catalog database must maintain information about suppliers, the parts they supply, and the prices charged by each supplier for specific parts. Each supplier in the system is uniquely identified by a supplier ID (sid), along with additional details such as supplier name and address. Each part in the system is uniquely identified by a part ID (pid) and includes descriptive attributes such as part name and color.

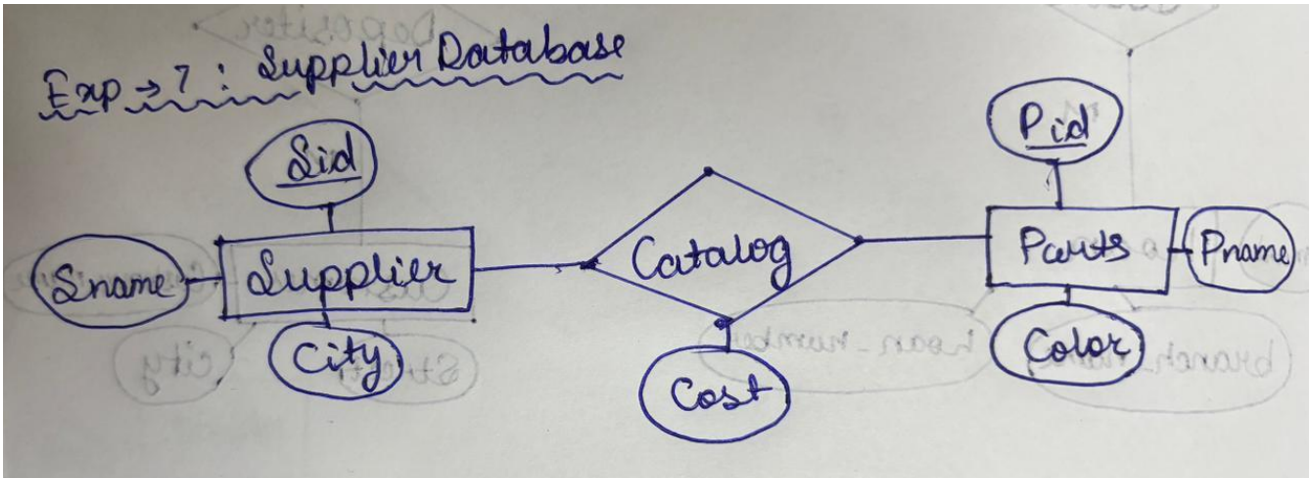
The database must record pricing information through a catalog relationship that links suppliers and parts. This relationship captures the cost at which a particular supplier supplies a specific part. A supplier may supply multiple parts, and a part may be supplied by multiple suppliers, resulting in a many-to-many relationship between suppliers and parts. This relationship is resolved using the CATALOG relation.

Each catalog entry must reference an existing supplier and an existing part. The combination of supplier ID and part ID must be unique in the catalog, ensuring that no duplicate price entries exist for the same supplier–part combination. The cost attribute must be a non-negative real value representing the price charged by the supplier for that part.

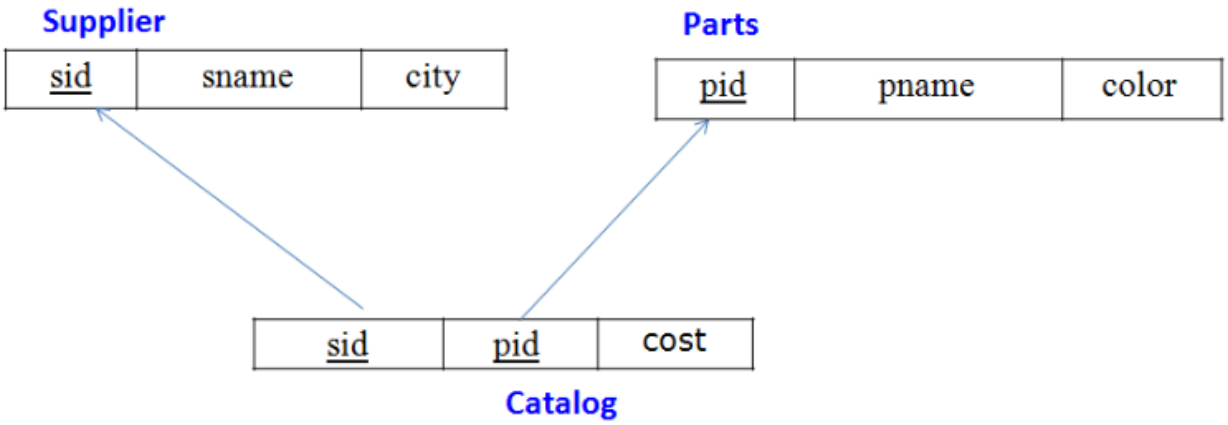
The database must enforce referential integrity such that no catalog record can exist unless the corresponding supplier and part records are already present in the system. Deletion of supplier or part records that are referenced in the catalog must be restricted or handled using controlled cascading or archival mechanisms to preserve historical pricing consistency.

Overall, the system must maintain accurate and consistent relationships between suppliers, parts, and catalog entries, enabling reliable retrieval of supplier details, part information, and pricing data for procurement, inventory management, and administrative purposes.

Entity Relationship Diagram



Schema Diagram



SUPPLIERS(sid: integer, sname: string, address: string)
PARTS(pid: integer, pname: string, color: string)
CATALOG(sid: integer, pid: integer, cost: real)
The Catalog relation lists the prices charged for parts by Suppliers.

Create database

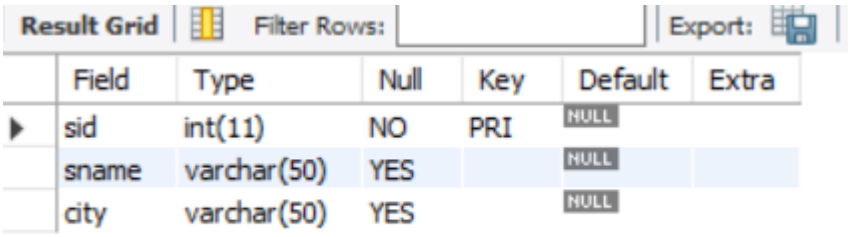
```
create database supplierdb;
use supplierdb;
```

Create Tables:

```
create table suppliers (
    sid int primary key,
    sname varchar(50),
    city varchar(50)
);

create table parts (
    pid int primary key,
    pname varchar(50),
    color varchar(20)
);

create table catalog (
    sid int,
    pid int,
    cost int,
    foreign key (sid) references suppliers(sid),
    foreign key (pid) references parts(pid)
);
```

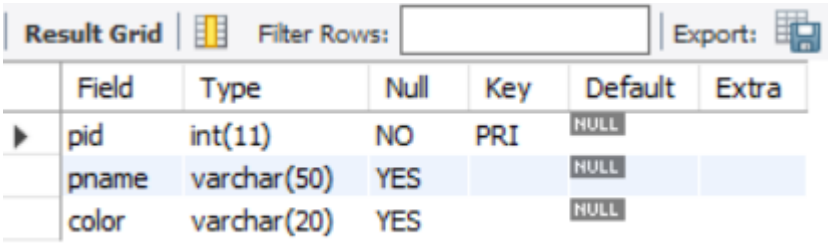
Structure of the table

desc suppliers;



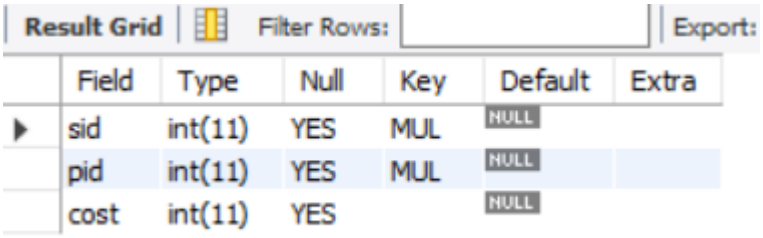
Field	Type	Null	Key	Default	Extra
sid	int(11)	NO	PRI	NULL	
sname	varchar(50)	YES		NULL	
city	varchar(50)	YES		NULL	

desc parts;



Field	Type	Null	Key	Default	Extra
pid	int(11)	NO	PRI	NULL	
pname	varchar(50)	YES		NULL	
color	varchar(20)	YES		NULL	

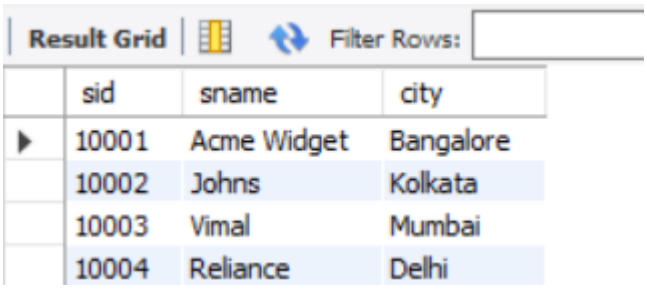
desc catalog;



Field	Type	Null	Key	Default	Extra
sid	int(11)	YES	MUL	NULL	
pid	int(11)	YES	MUL	NULL	
cost	int(11)	YES		NULL	

Inserting Values to the table

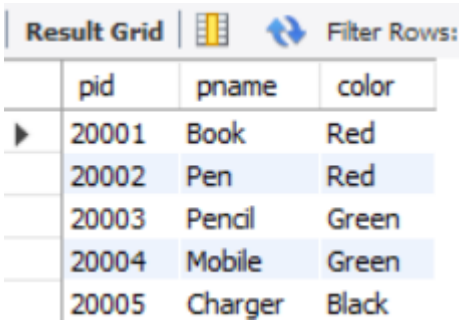
insert into suppliers (sid, sname, city) values
(10001, 'Acme Widget', 'Bangalore'),
(10002, 'Johns', 'Kolkata'),
(10003, 'Vimal', 'Mumbai'),
(10004, 'Reliance', 'Delhi');
select * from suppliers;



sid	sname	city
10001	Acme Widget	Bangalore
10002	Johns	Kolkata
10003	Vimal	Mumbai
10004	Reliance	Delhi

insert into parts (pid, pname, color) values
(20001, 'Book', 'Red'),

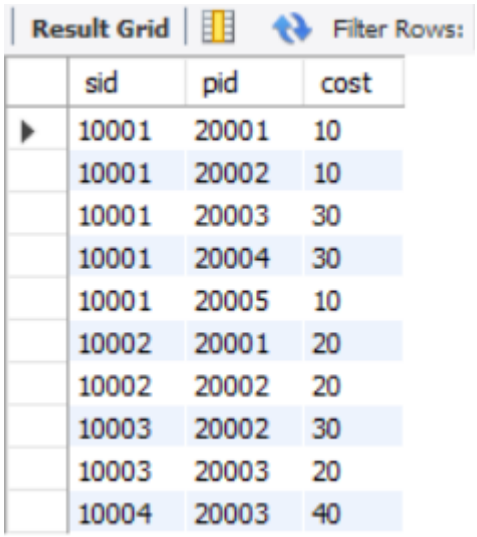
(20002, 'Pen', 'Red'),
(20003, 'Pencil', 'Green'),
(20004, 'Mobile', 'Green'),
(20005, 'Charger', 'Black');
select * from parts;



Result Grid | Filter Rows:

pid	pname	color
20001	Book	Red
20002	Pen	Red
20003	Pencil	Green
20004	Mobile	Green
20005	Charger	Black

insert into catalog (sid, pid, cost) values
(10001, 20001, 10),
(10001, 20002, 10),
(10001, 20003, 30),
(10001, 20004, 30),
(10001, 20005, 10),
(10002, 20001, 20),
(10002, 20002, 20),
(10003, 20002, 30),
(10003, 20003, 20),
(10004, 20003, 40);



Result Grid | Filter Rows:

sid	pid	cost
10001	20001	10
10001	20002	10
10001	20003	30
10001	20004	30
10001	20005	10
10002	20001	20
10002	20002	20
10003	20002	30
10003	20003	20
10004	20003	40

Queries:

Query 1:

Find the pnames of parts for which there is some supplier.

```
select distinct p.pname from parts p join catalog c on p.pid = c.pid;
```

Result Grid | Filter Rows:

pname
Book
Pen
Pencil
Mobile
Charger

Query 2:

Find the snames of suppliers who supply every part.

```
select s.sname from suppliers s join catalog c on s.sid = c.sid group by s.sid, s.sname
having count(distinct c.pid) = (select count(*) from parts);
```

Result Grid | Filter Rows:

sname
Acme Widget

Query 3:

Find the snames of suppliers who supply every red part.

```
select s.sname from suppliers s join catalog c on s.sid = c.sid where c.pid in (select pid
from parts where color = 'Red') group by s.sid, s.sname having count(distinct c.pid) =
(select count(*) from parts where color = 'Red');
```

Result Grid | Filter Rows:

sname
Acme Widget
Johns

Query 4:

Find the pnames of parts supplied by Acme Widget Suppliers and by no one else.

```
select p.pname from parts p where p.pid in (select pid from catalog where sid =
10001and pid not in (select pid from catalog where sid <> 10001));
```

Result Grid

pname
Mobile
Charger

Query 5:

Find the sids of suppliers who charge more for some part than the average cost of that part (averaged over all the suppliers who supply that part).

```
select distinct c.sid from catalog c join (select pid, avg(cost) as avg_cost
from catalog group by pid) x on c.pid = x.pid where c.cost > x.avg_cost;
```

Result Grid

sid
10002
10003
10004

Query 6:

For each part, find the sname of the supplier who charges the most for that part.

```
select p.pname, s.sname from parts p join catalog c on p.pid = c.pid
join suppliers s on c.sid = s.sid where (c.pid, c.cost) in (select pid, max(cost)
from catalog group by pid);
```

Result Grid Filter Rows:

pname	sname
Mobile	Acme Widget
Charger	Acme Widget
Book	Johns
Pen	Vimal
Pencil	Reliance

Experiment 8: More Queries on Supplier Database

Queries:

Query 1:

Find the most expensive part overall and the supplier who supplies it.

```
select s.sname, p.pname, c.cost from catalog c join suppliers s on c.sid = s.sid join parts p on c.pid = p.pid where c.cost = (select max(cost) from catalog);
```

Result Grid | Filter Rows:

sname	pname	cost
Reliance	Pencil	40

Query 2:

Find suppliers who do NOT supply any red parts.

```
select s.* from suppliers s where s.sid not in (select c.sid from catalog c join parts p on c.pid = p.pid where p.color = 'Red');
```

Result Grid | Filter Rows:

sid	sname	city
10004	Reliance	Delhi

Query 3:

Show each supplier and total value of all parts they supply.

```
select s.sname, sum(c.cost) as totalvalue from suppliers s join catalog c on s.sid = c.sid group by s.sid;
```

Result Grid | Filter Rows:

sname	totalvalue
Acme Widget	90
Johns	40
Vimal	50
Reliance	40

Query 4:

Find suppliers who supply at least 2 parts cheaper than ₹20.

select s.sid, s.sname from suppliers s join catalog c on s.sid = c.sid where c.cost < 20 group by s.sid having count(c.pid) >= 2;

Result Grid | Filter Rows:

sid	sname
10001	Acme Widget

Query 5:

List suppliers who offer the cheapest cost for each part.

select s.sname, p.pname, c.cost from catalog c join suppliers s on c.sid = s.sid join parts p on c.pid = p.pid where c.cost = (select min(c2.cost) from catalog c2 where c2.pid = c.pid);

Result Grid | Filter Rows:

sname	pname	cost
Acme Widget	Book	10
Acme Widget	Pen	10
Acme Widget	Mobile	30
Acme Widget	Charger	10
Vimal	Pencil	20

Query 6:

Create a view showing suppliers and the total number of parts they

supply.

create view supplier_part_count as select s.sid, s.sname, count(c.pid) as totalparts from suppliers s left join catalog c on s.sid = c.sid group by s.sid;

select * from supplier_part_count;

Result Grid | Filter Rows:

sid	sname	totalparts
10001	Acme Widget	5
10002	Johns	2
10003	Vimal	2
10004	Reliance	1

Query 7:

Create a view of the most expensive supplier for each part.

```
create view most_expensive_supplier as select s.sname, p.pname, c.cost from catalog c join suppliers s on c.sid = s.sid join parts p on c.pid = p.pid where c.cost = (select max(c2.cost) from catalog c2 where c2.pid = c.pid);
```

Result Grid | Filter Rows:

sname	pname	cost
Acme Widget	Mobile	30
Acme Widget	Charger	10
Johns	Book	20
Vimal	Pen	30
Reliance	Pencil	40

Query 8:

Create a Trigger to prevent inserting a Catalog cost below 1.

```
DELIMITER //
create trigger prevent_low_cost
before insert on catalog
for each row
begin
   if new.cost < 1 then
      signal sqlstate '45000' set message_text = 'Cost must be at least 1';
   end if;
end;

//
DELIMITER ;
```

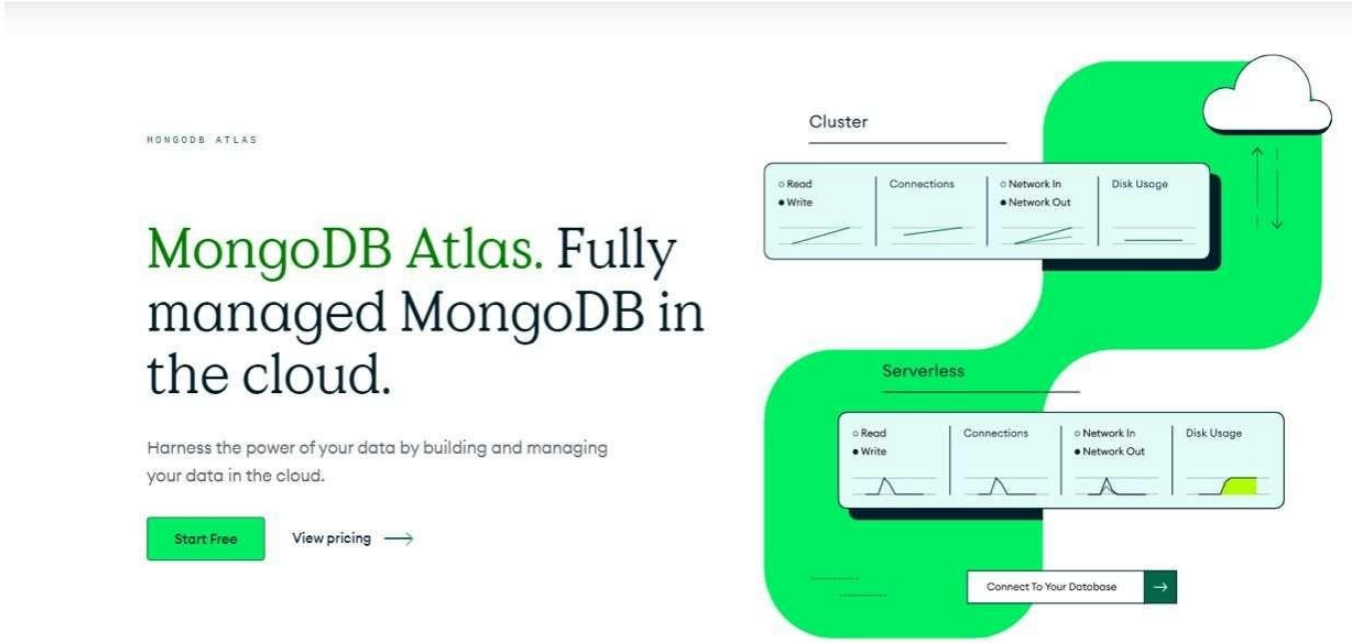
Query 9:

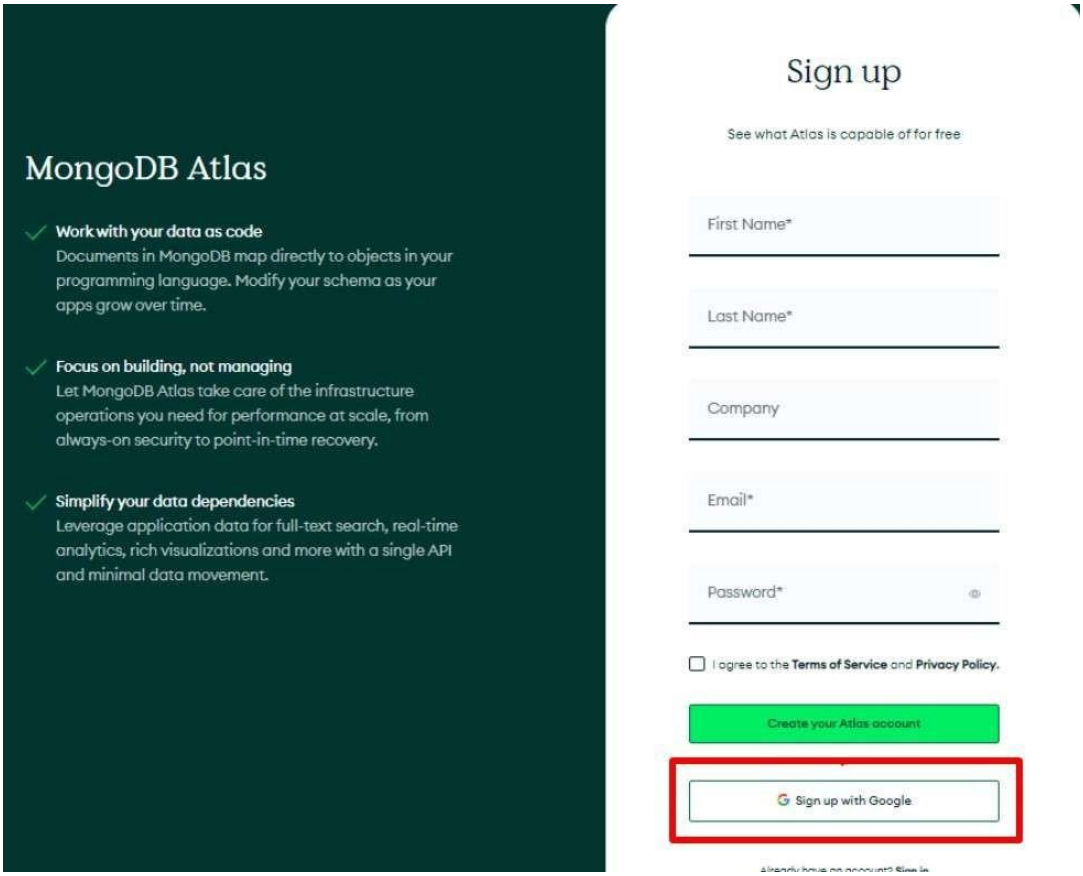
Create a trigger to set to default cost if not provided.

```
DELIMITER //
create trigger set_default_cost
before insert on catalog
for each row
begin
    if new.cost is null then
        set new.cost = 100;
    end if;
end;

//
DELIMITER ;
```

NOSQL Installation in Cloud

NoSQL installation in the cloud with reference to MongoDB is commonly done using a managed cloud service such as MongoDB Atlas, which simplifies deployment and maintenance. In this approach, MongoDB is hosted on cloud platforms like AWS, Microsoft Azure, or Google Cloud, eliminating the need for manual installation and server configuration. The user creates a cluster through a web interface, selects the cloud provider and region, and configures basic security settings such as database users and network access. Once the cluster is created, MongoDB provides a secure connection string that allows applications to connect to the database from anywhere in the cloud or on-premises systems. This cloud-based installation supports automatic scaling, backups, and replication, ensuring high availability and reliability. By using MongoDB in the cloud, organizations benefit from reduced administrative overhead, improved performance, and the ability to easily manage large volumes of unstructured or semi-structured data.



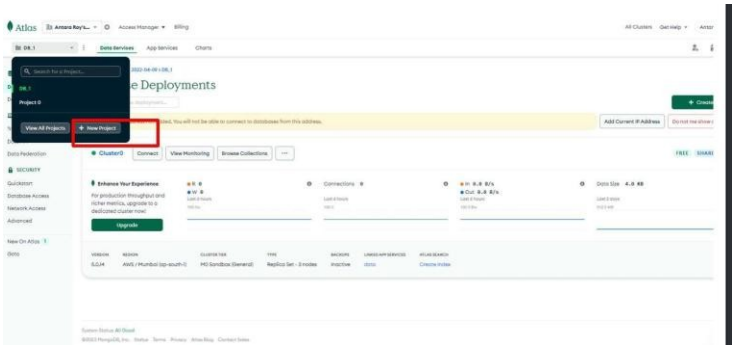


Steps to Install and Use NoSQL Database in Cloud

Account Creation

1. Open the MongoDB Atlas website.

2. Register using email or Google authentication.
3. Verify the account and log in



Projec tCreation

1. Click **New Project** from the dashboard.
2. Enter a project name (e.g., *CloudNoSQLProject*).
3. Create the project.

Database User Configuration

1. Navigate to **Database Access**.
2. Add a new database user.
3. Provide username and password.
4. Assign **read and write permissions**.
5. Save the configuration.

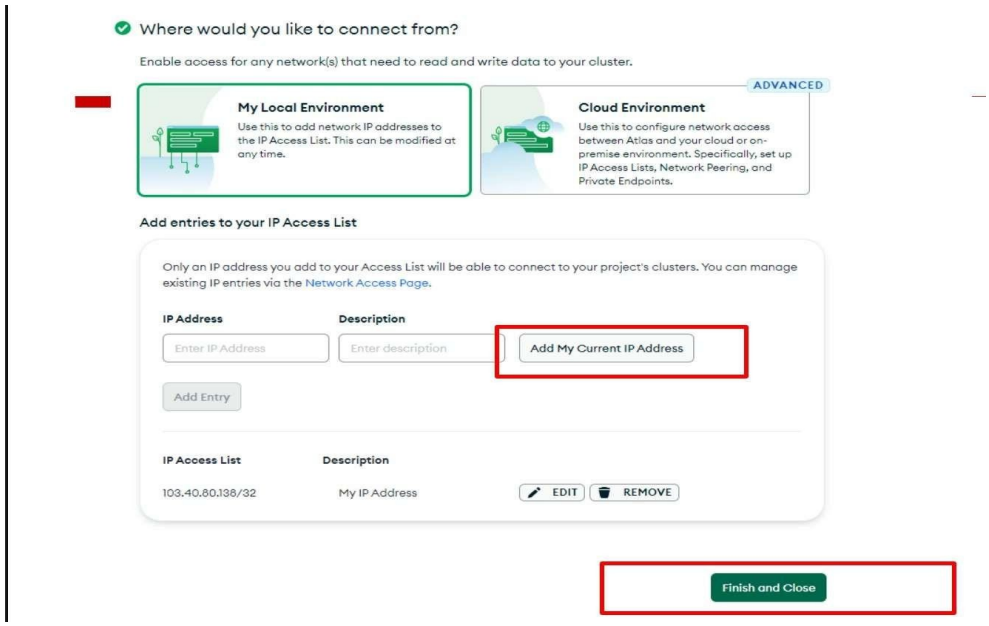
Network Access Configuration

1. Go to **Network Access**.
2. Add a new IP address.
3. Allow access from all IP addresses (`0.0.0.0/0`) for development.
4. Confirm the changes.

Database Connection

1. Select **Connect** from the dashboard.
2. Choose **Connect using MongoDB Shell**.
3. Copy the connection string.
4. Replace the username and password.

Connect to the database using the MongoDB shell



Database Creation

After successful connection, create a database:

```
use StudentDB
```

This command creates and switches to the database if it does not already exist.

Experiment 9: NOSQL - Student Database

Specification of Student Database Application

The **Student database** is designed to store and manage basic information related to students using a NoSQL (MongoDB) environment. Each student in

the system is uniquely identified by a **Roll Number (Rollno)**. Along with the roll number, the database maintains essential student details such as **age**, **contact number**, and **email ID**, which are required for identification and communication purposes.

The system must support the insertion of new student records and allow modification of existing records. It should provide update operations to change specific attributes such as the **email ID** of a student identified by a given roll number, as well as replacement of attribute values like updating a student's name when required.

The database should allow exporting the stored student records to the local file system in standard formats such as JSON or CSV for backup, reporting, or data migration. Similarly, it must support importing student data from an external CSV file into the MongoDB collection while maintaining data consistency.

Proper deletion mechanisms must be available to remove the student collection when required. The system should ensure that all stored values are valid, such as maintaining reasonable age values and correctly formatted contact numbers and email IDs.

Overall, the Student database must ensure efficient storage, retrieval, update, export, and import of student information, supporting reliable student data management using MongoDB.

i) Create database

db.createCollection(‘student')

use Student

ii. Create table And insert appropriate values

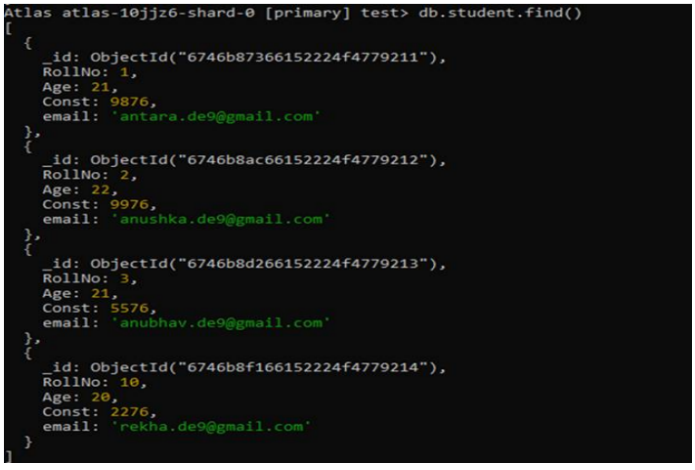
db.student.insertMany([

{Rollno: 10, Name: "ABC", Age: 20, ContactNo: "9876543210", EmailId: “abc@gmail.com"},

{Rollno: 11, Name: "ABC", Age: 21, ContactNo: "9123456780", EmailId: "abc11@gmail.com"},

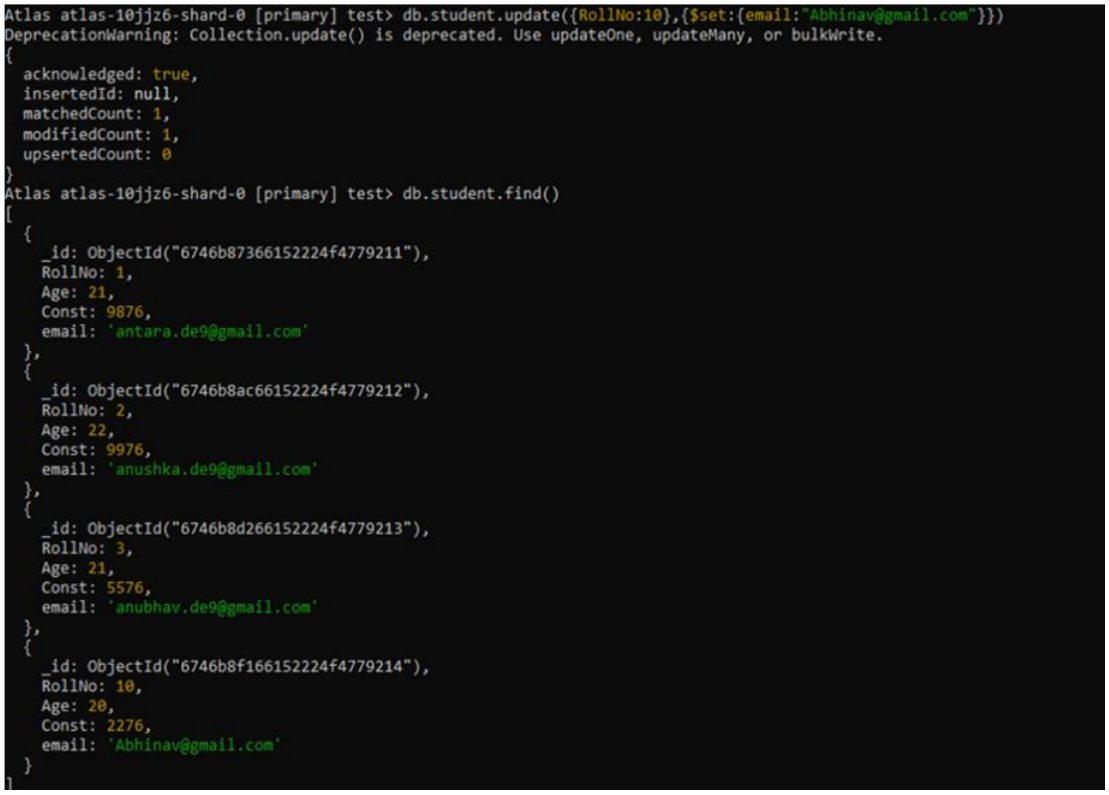
{Rollno: 12, Name: "XYZ", Age: 22, ContactNo: "9988776655", EmailId: "xyz@gmail.com"}

])



iii.Write query to update Email-Id of a student with rollno 10.

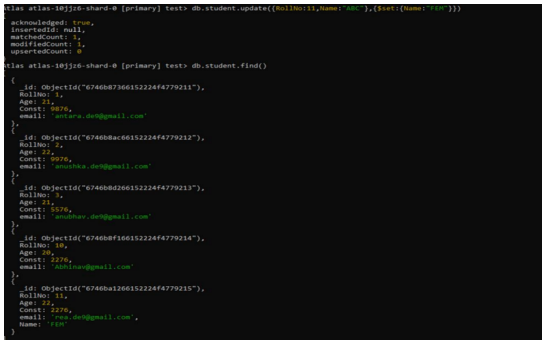
```
db.student.updateOne(
  {Rollno: 10},
  {$set: {EmailId: "newemail10@gmail.com"}}
)
```



iv. Replace the student name from “ABC” to “FEM” of rollno 11

```
db.student.updateOne(
  {Rollno: 11},
```

{$set: {Name: "FEM"}}
)



v.Export the created table into local file system

mongoexport --db Student --collection student --out student.json

vi.Drop the table.
db.student.drop()

vii. Import a given csv dataset from local file system into mongodb collection.

mongoimport --db Student --collection student \
--type=csv --headerline --file student.csv

Experiment 10: NOSQL- Customer Database

Specification of Customer Database Application

The **Customer database** is designed to maintain information about customers and their bank accounts. Each customer in the system is uniquely identified by a **Customer ID (Cust_id)**. For every customer, the database stores details of one or more bank accounts held by the customer, including the **account balance (Acc_Bal)** and the **account type (Acc_Type)**.

The system must allow a customer to hold multiple accounts, possibly of the same or different account types, while ensuring that each account record is correctly associated with a valid customer. Duplicate account records for the same customer with identical account details must not exist. The **account balance** stored in the system must always be a **non- negative value**, and the **account type** must belong to a predefined set of valid account categories such as savings, current, or special types (for example, type ‘Z’).

The database should support queries that allow computation of the **total account balance per customer**, especially for customers holding accounts of a specific type. It must also allow analytical operations such as determining the **minimum and maximum account balances** maintained by each customer across all their accounts.

The system should provide mechanisms to **export customer and account data** to the local file system for backup, reporting, or data migration purposes, and similarly allow **importing data from external CSV files** into MongoDB collections while preserving data consistency. Proper deletion policies must be enforced so that customer records are removed only when it does not violate data consistency or application requirements.

Overall, the database must ensure accurate storage, retrieval, and aggregation of customer account information, supporting efficient financial analysis and reliable data management in a NoSQL (MongoDB) environment.

i) Create a collection named Customers

```
db.createCollection("Customers")
```

ii) Creating table and inserting the appropriate values

```
db.Customers.insertMany([
  {Cust_id: 101, Acc_Bal: 500, Acc_Type: "Z"},
  {Cust_id: 101, Acc_Bal: 800, Acc_Type: "Z"},
  {Cust_id: 102, Acc_Bal: 1500, Acc_Type: "Y"},
  {Cust_id: 103, Acc_Bal: 700, Acc_Type: "Z"},
  {Cust_id: 103, Acc_Bal: 600, Acc_Type: "Z"},
  {Cust_id: 104, Acc_Bal: 2000, Acc_Type: "Z"}
])
```

iii) Write a query to display those records whose total account balance is greater than 1200 of account type 'Z' for each customer_id.

```
db.Customers.aggregate([
  {$match: {Acc_Type: "Z"}},
  {$group: {
     _id: "$Cust_id",
     totalBalance: {$sum: "$Acc_Bal"}
  }},
  {$match: {totalBalance: {$gt: 1200}}}
])
```

iv. Determine Minimum and Maximum account balance for each customer_id.

```
db.Customers.aggregate([
  {$group: {_id: "$Cust_id", minBalance: {$min: "$Acc_Bal"}, maxBalance:
     {$max: "$Acc_Bal"}}}])
```

v. Export the created collection into local file system.

```
mongoexport --db CustomerDB --collection Customers --out Customers.json
```

vi. Drop the table.

```
db.Customers.drop()
```

vii. Import a given csv dataset from local file system into mongodb collection.

```
mongoimport --db CustomerDB --collection Customers \
--type=csv --headerline --file Customers.csv
```

Output:

```
switched to db student
{ "ok" : 1 }
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("693fedc1f35842b2df36f04c"),
                ObjectId("693fedc1f35842b2df36f04d"),
                ObjectId("693fedc1f35842b2df36f04e")
        ]
}
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
{
        "_id" : ObjectId("693fedc1f35842b2df36f04c"),
        "Rollno" : 10,
        "Name" : "ABC",
        "Age" : 20,
        "ContactNo" : "9876543210",
        "EmailId" : "newemail10@gmail.com"
}
{
        "_id" : ObjectId("693fedc1f35842b2df36f04d"),
        "Rollno" : 11,
        "Name" : "FEM",
        "Age" : 21,
        "ContactNo" : "9123456780",
        "EmailId" : "abc11@gmail.com"
}
{
        "_id" : ObjectId("693fedc1f35842b2df36f04e"),
        "Rollno" : 12,
        "Name" : "XYZ",
        "Age" : 22,
        "ContactNo" : "9988776655",
        "EmailId" : "xyz@gmail.com"
}
true
```

Experiment 11: NOSQL- Restaurant Database

Specification of Restaurant Database Application

The **Restaurant database** is designed to store and manage information related to restaurants and their inspection scores using a NoSQL (MongoDB) environment. Each restaurant in the system is uniquely identified by a **Restaurant ID**. Along with the identifier, the database maintains essential details such as the **name of the restaurant**, the **town in which it is located**, and the **type of cuisine** offered.

The system must also store inspection or review information in the form of **scores** assigned to restaurants. A restaurant may have one or more score entries recorded over time, and these scores are used to evaluate the overall quality and compliance of the restaurant. The database should support analytical queries to retrieve restaurant details based on score constraints, such as identifying restaurants whose scores do not exceed a specified value.

The database must provide querying capabilities to display all restaurant records, sort restaurant details based on attributes such as name, and project selected fields like restaurant ID, name, town, and cuisine. It should also support aggregation operations to compute statistical measures, including the **average score for each restaurant**, enabling performance analysis and comparison.

The system should ensure that score values are valid numeric entries and remain within acceptable limits. It must maintain consistency between restaurant details and their associated score records. Overall, the Restaurant database should allow efficient storage, retrieval, sorting, filtering, and aggregation of restaurant information, supporting reliable analysis and reporting in a MongoDB environment.

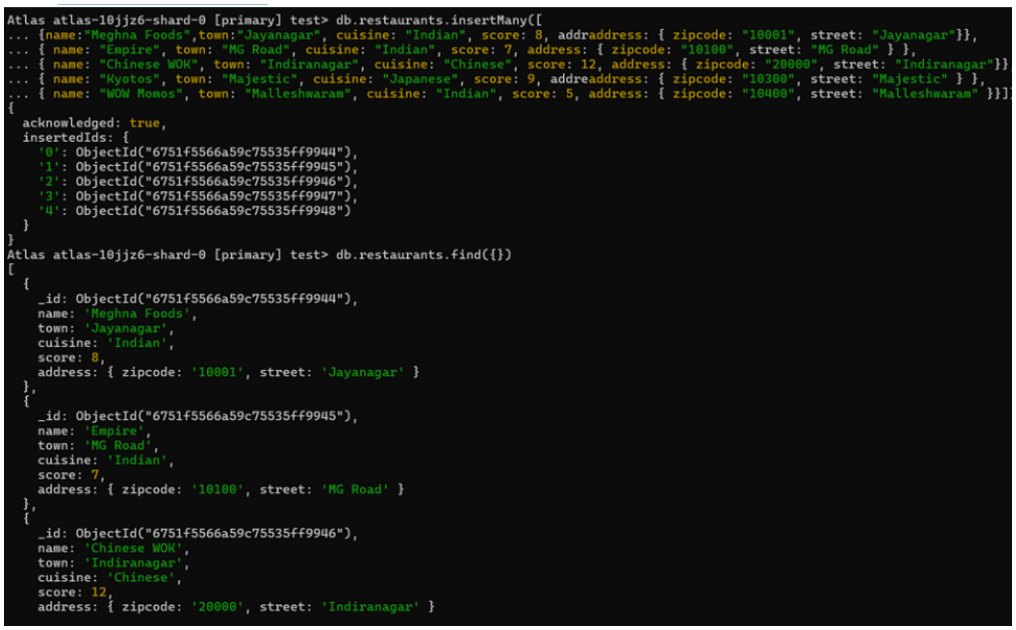
i) Create a collection named Restaurants

db.createCollection(“Restaurants”)

ii)Creating table and inserting the appropriate values

db.Restaurant.insertMany([{name:"Meghna Foods",town:"Jayanagar", cuisine: "Indian", score: 8, address: { zipcode: "10001", street: "Jayanagar"}}, { name: "Empire", town: "MG Road", cuisine: "Indian", score: 7, address: { zipcode: "10100", street: "MG Road" } }, { name: "Chinese WOK", town: "Indiranagar", cuisine: "Chinese", score: 12, address: { zipcode: "20000", street: "Indiranagar"}}, { name: "Kyotos", town: "Majestic", cuisine: "Japanese", score: 9, address: { zipcode: "10300", street: "Majestic" } }, { name: "WOW Momos", town: "Malleshwaram", cuisine: "Indian", score: 5, address: { zipcode: "10400", street: "Malleshwaram" }}])

db.Restaurant.find();



iii)Write a MongoDB query to find the restaurant Id, name, town and cuisine for those restaurants which a score which is not more than 10.

db.Restaurant.find({ "grades.score": { $lte: 10 } },{ _id: 1, name: 1, town: 1, cuisine: 1, restaurant_id: 1 });

```
  name: 'Empire',
  town: 'MG Road',
  cuisine: 'Indian',
  score: 7,
  address: { zipcode: '10100', street: 'MG Road' }
},
{
  _id: ObjectId("6751f5566a59c75535ff9946"),
  name: 'Chinese WOK',
  town: 'Indiranagar',
  cuisine: 'Chinese',
  score: 12,
  address: { zipcode: '20000', street: 'Indiranagar' }
}
]
Atlas atlas-10jjz6-shard-0 [primary] test> db.restaurants.find({ "score": { $lte: 10 } }, { _id: 1, name: 1, town: 1, cuisine: 1 })
[
  {
    _id: ObjectId("6751f5566a59c75535ff9944"),
    name: 'Meghna Foods',
    town: 'Jayanagar',
    cuisine: 'Indian'
  },
  {
    _id: ObjectId("6751f5566a59c75535ff9945"),
    name: 'Empire',
    town: 'MG Road',
    cuisine: 'Indian'
  },
  {
    _id: ObjectId("6751f5566a59c75535ff9947"),
    name: 'Kyotos',
    town: 'Majestic',
    cuisine: 'Japanese'
  },
  {
    _id: ObjectId("6751f5566a59c75535ff9948"),
    name: 'WOW Momos',
    town: 'Malleshwaram',
    cuisine: 'Indian'
  }
```

iv)Write a MongoDB query to find the average score for each restaurant.

```
db.restaurants.aggregate([ { $group: { _id: "$name", average_score: { $avg: "$score" }}}])
```

viii.) Write a MongoDB query to find the name and address of the restaurants that have a zipcode that starts with '10'.

```
db.restaurants.find({ "address.zipcode": /^10/ }, { name: 1,"address.street":1, _id: 0 })
```

```
Atlas atlas-10jjz6-shard-0 [primary] test> db.restaurants.aggregate([ { $group: { _id: "$name", average_score: { $avg: "$score" } } }
... ])
[
  { _id: 'Meghna Foods', average_score: 8 },
  { _id: 'WOW Momos', average_score: 5 },
  { _id: 'Chinese WOK', average_score: 12 },
  { _id: 'Kyotos', average_score: 9 },
  { _id: 'Empire', average_score: 7 }
]
Atlas atlas-10jjz6-shard-0 [primary] test> db.restaurants.find({ "address.zipcode": /^10/ }, { name: 1, "address.street": 1, _id: 0 })
[
  { name: 'Meghna Foods', address: { street: 'Jayanagar' } },
  { name: 'Empire', address: { street: 'MG Road' } },
  { name: 'Kyotos', address: { street: 'Majestic' } },
  { name: 'WOW Momos', address: { street: 'Malleshwaram' } }
]
```

LEETCODE PRACTICE 1

Table: `Products`

Code

MySQL Auto

```mysql
# Write your MySQL query statement below
SELECT sell_date,
       COUNT(DISTINCT product) AS num_sold,
       GROUP_CONCAT(DISTINCT product ORDER BY product ASC) AS products
FROM Activities
GROUP BY sell_date
ORDER BY sell_date;
```

```
| unit          | int     |
+---------------+---------+
This table may have duplicate rows.
product_id is a foreign key (reference column) to the Products table.
unit is the number of products ordered in order_date.
```

Write a solution to get the names of products that have at least `100` units ordered in **February 2020** and their amount.

Return the result table in **any order**.

OUTPUT

Code

MySQL Auto

```mysql
# Write your MySQL query statement below
SELECT sell_date,
       COUNT(DISTINCT product) AS num_sold,
       GROUP_CONCAT(DISTINCT product ORDER BY product ASC) AS products
FROM Activities
GROUP BY sell_date
ORDER BY sell_date;
```

LEETCODE PRACTICE 2

Table `Activities`:

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| sell_date   | date    |
| product     | varchar |
+-------------+---------+
```

There is no primary key (column with unique values) for this table. It may contain duplicates.
Each row of this table contains the product name and the date it was sold in a market.

Write a solution to find for each date the number of different products sold and their names.

The sold products names for each date should be sorted lexicographically.

Return the result table ordered by `sell_date`.

OUTPUT

Code

MySQL Auto

```mysql
# Write your MySQL query statement below
SELECT p.product_name, SUM(o.unit) AS unit
FROM Orders o
JOIN Products p ON o.product_id = p.product_id
WHERE o.order_date BETWEEN '2020-02-01' AND '2020-02-29'
GROUP BY o.product_id, p.product_name
HAVING SUM(o.unit) >= 100;
```