VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Database Management Systems (23CS3PCDBM)

Submitted by

Deepak S (1BM25CS435)

in partial fulfillment 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 **Deepak S (1BM25CS435),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment 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.

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

Index

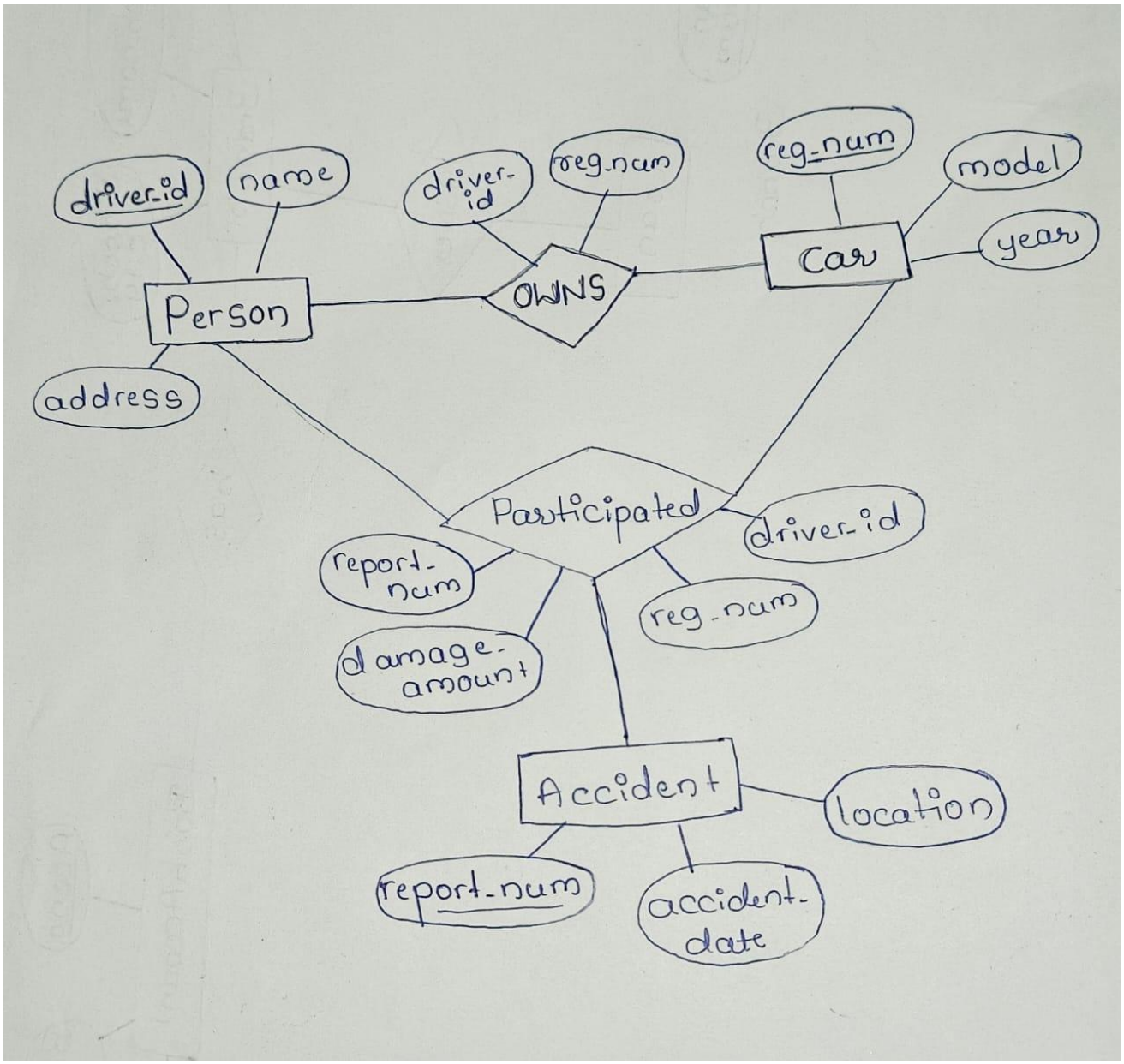
Sl. No.	Date	Experiment Title	Page No.
1	9-10-2025	Insurance Database	4-10
2	9-10-2025	More Queries on Insurance Database	11-11
3	16-10-2025	Bank Database	12-19
4	16-10-2025	More Queries on Bank Database	20-22
5	6-11-2025	Employee Database	23-30
6	13-11-2025	More Queries on Employee Database	31-33
7	20-11-2025	Supplier Database	34-39
8	27-11-2025	More Queries on Supplier Database	40-43
9	11-12-2025	NOSQL Installation in Cloud	44-44
10	11-12-2025	NO SQL - Student Database	45-47
11	11-12-2025	NO SQL - Customer Database	48-51
12	11-12-2025	NO SQL – Restaurant Database	52-59
13	20-12-2025	LeetCode Practice	60-60
14	20-12-2025	LeetCode Practice	61-62

Experiment 1: Insurance Database

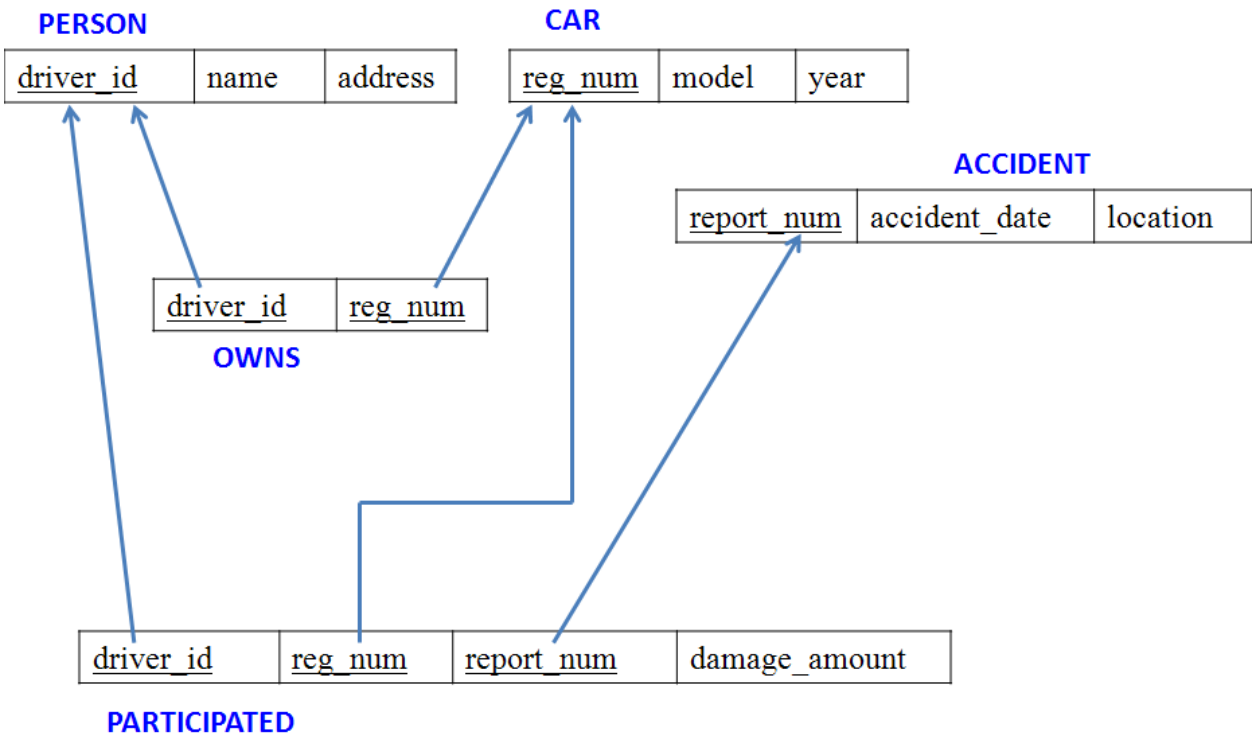
Specification of Insurance Database Application

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 person(
driver_id varchar(10) primary key,
name varchar(20),
address varchar(30),

create table cars (
reg_num varchar(10),
model varchar(20),
year int);

create table owners(
driver_id varchar(10),
reg_num varchar(10),
foreign key(driver_id) references person1(driver_id),
foreign key(reg_num)references cars(reg_num));

create table participated(
driver_id varchar (10),
reg_num varchar(10),
report_num int primary key,
damage_ammount int,foreign key(driver_id) references person1(driver_id),
foreign key(reg_num)references cars(reg_num));
```

create table accident(report_num int,

accident_date date,

location varchar(20),

foreign key(report_num) references participated(report_num));

Structure of the table

desc person;

	Field	Type	Null	Key	Default	Extra
▶	driver_id	varchar(10)	NO	PRI	NULL	
	name	varchar(20)	YES		NULL	
	address	varchar(30)	YES		NULL	

desc accident;

	Field	Type	Null	Key	Default	Extra
▶	report_num	int	YES	MUL	NULL	
	accident_date	date	YES		NULL	
	location	varchar(20)	YES		NULL	

desc participated;

	Field	Type	Null	Key	Default	Extra
▶	driver_id	varchar(10)	YES	MUL	NULL	
	reg_num	varchar(10)	YES	MUL	NULL	
	report_num	int	NO	PRI	NULL	
	damage_ammount	int	YES		NULL	

desc car;

	Field	Type	Null	Key	Default	Extra
▶	reg_num	varchar(15)	NO	PRI	NULL	
	model	varchar(20)	YES		NULL	
	year	int	YES		NULL	

desc owns;

	Field	Type	Null	Key	Default	Extra
▶	driver_id	varchar(10)	YES	MUL	NULL	
	reg_num	varchar(10)	YES	MUL	NULL	

Inserting Values to the table

insert into person1 values("A01","richard","srinivas nagar");

insert into person1 values("A02","pradeep","rajaji nagar");

insert into person1 values("A03","smith","ashok nagar");

insert into person1 values("A04","venu","nr colony");

insert into person1 values("A05","jhon","hanumanth nagar");

select * from person1;

driver_id	name	address
A01	richard	srinivas nagar
A02	pradeep	rajaji nagar
A03	smith	ashok nagar
A04	venu	nr colony
A05	jhon	hanumanth nagar
NULL	NULL	NULL

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;

reg_num	model	year
KA031181	lancer	1957
KA041702	audi	2005
KA05225	indica	1990
KA053408	honda	2008
KA095477	toyota	1998
NULL	NULL	NULL

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
▶	A01	KA05225
	A02	KA053408
	A03	KA031181
	A04	KA095477
	A05	KA041702

```
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	2005-03-04	kanakpura road

```
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_ammount
▶	A01	KA05225	11	10000
	A02	KA053408	12	25000
	A03	KA031181	13	25000
	A04	KA095477	14	3000
	A05	KA041702	15	5000
*	NULL	NULL	NULL	NULL

Experiment 2:More Queries on Insurance Database

Queries (Questions and Output)

- **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_ammount=25000

  where report_num=12;

  select *from participated;

	driver_id	reg_num	report_num	damage_ammount
▶	A01	KA05225	11	10000
	A02	KA053408	12	25000
	A03	KA031181	13	25000
	A04	KA095477	14	3000
	A05	KA041702	15	5000
*	NULL	NULL	NULL	NULL

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

  select count(*) from participated p,
  accident a where p.report_num=a.report_num and
  a.accident_date like "%2008-02-17";

	count(*)
▶	1

- **Find the average of damage Ammount from participated.**

  select avg(damage_ammount) from participated;

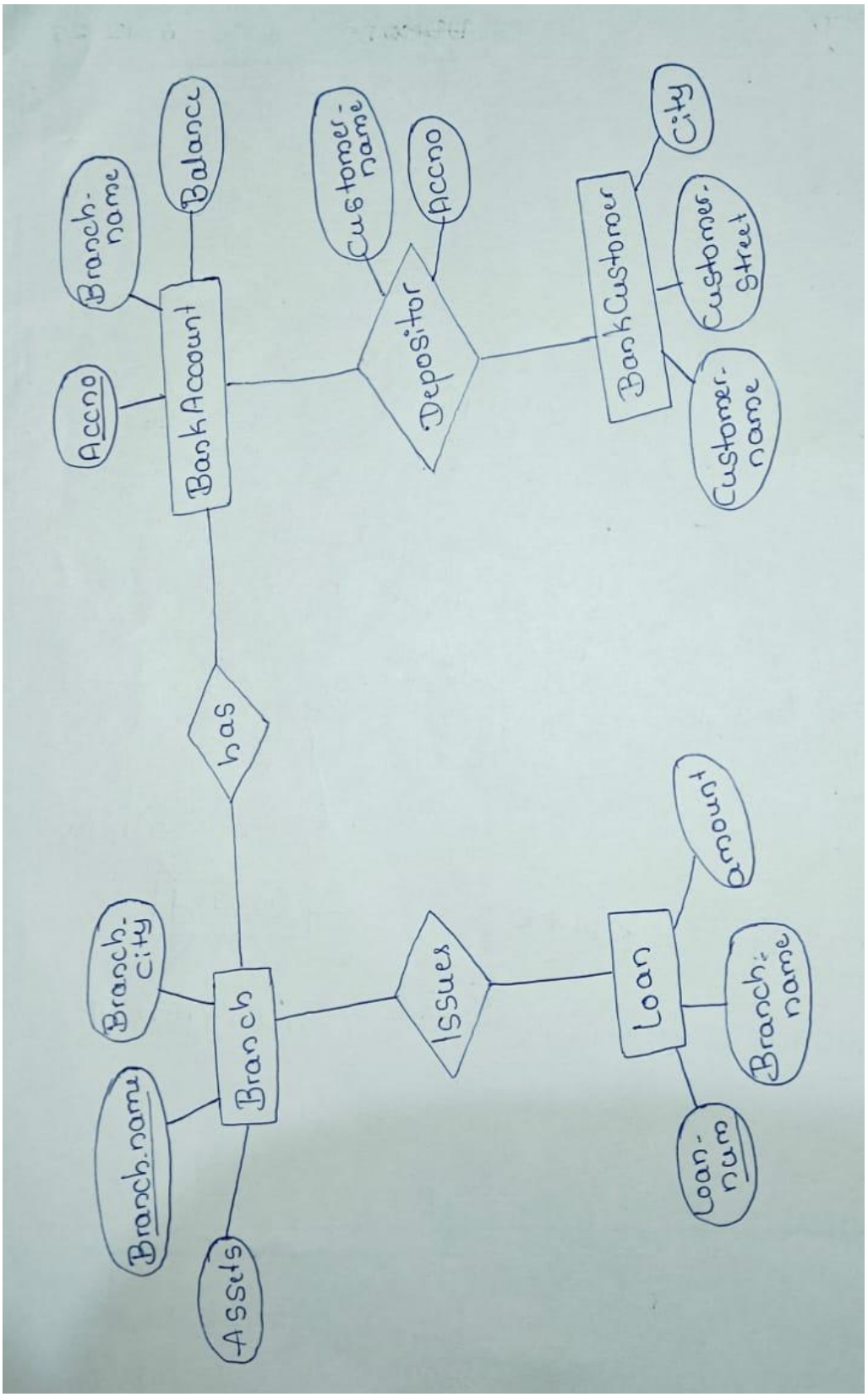
	avg(damage_ammount)
▶	13600.0000

Experiment 3: Bank Database

Specification of Insurance Database Application

The banking system must store information about branches, bank accounts, customers, deposit relationships, and loans so that branch details (identified by branch name together with city and total assets) are linked to accounts and loans, each account (identified by an account number) records the branch it belongs to and the current balance, customers are recorded with their name, street and city, and a depositor relationship associates a customer with an account; loans are recorded by a unique loan number together with the branch name that issued the loan and the loan amount. Account numbers and loan numbers must be unique identifiers, branch names are used to associate accounts and loans to a branch, and customer names (as modelled) are used to identify customers referenced by depositor entries; every depositor entry must reference an existing customer and an existing account so that ownership and access relationships are always valid, and duplicate depositor records linking the same customer and account are disallowed. The system must maintain referential integrity so accounts cannot reference a non-existent branch, depositor rows cannot reference missing customers or accounts, and loans must reference an existing branch; deletion of a branch, account, or customer that is referenced by dependent records should be controlled (either disallowed or handled by archival/controlled reassignment) to preserve historical transaction and loan consistency. Numeric and temporal constraints must be enforced: account balances should be constrained to valid values (for example non-negative WHERE overdraft is not allowed), branch assets and loan amounts must be non-negative and within specified business limits, and UPDATEs to balance or loan amounts should be auditable. Cardinality rules implied by the schema are enforced: a branch may host many accounts and issue many loans, an account belongs to exactly one branch, a customer may be linked to many accounts through depositor relationships, and an account may have many depositors if joint accounts are permitted by policy. Implementation must prevent orphaned records, ensure uniqueness WHERE required, and rely on application logic or database-level triggers to enforce complex rules such as cascading effects on deletion, business rules about allowed balance operations or overdrafts, and any required validation when transferring accounts between branches or when converting a customer's identifying details; the database should thus reliably support queries for branch-wise account lists, customer account ownership, account balances, and loan portfolios while preserving historical and referential integrity for auditing and regulatory reporting.

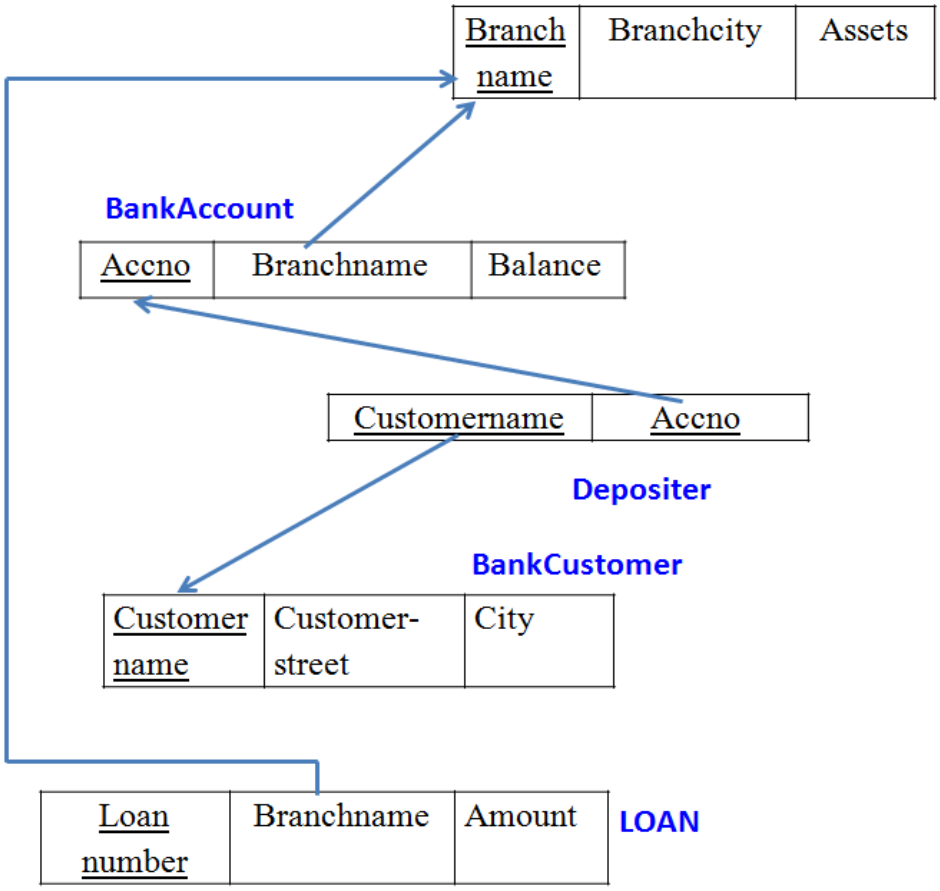
Entity Relationship Diagram



Consider the following database for a banking enterprise.

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)
Loan (loan-number: int, branch-name: String, amount: real)

Schema Diagram



Create database

```
create database Bank;

use Bank;
```

Create table

```
create table branch(
branchname varchar(50) primary key,
branchcity varchar(50),
assets int);

create table bankaccount(
accno int primary key,
branchname varchar(50),
balance int,
foreign key(branchname) references branch (branchname));

create table bankcustomer (
customername varchar(20) primary key,
customerstreet varchar(50),
city varchar(50));

create table depositer(
customername varchar(20),
accno int,
foreign key(customername)references bankcustomer(customername),
foreign key(accno)references bankaccount(accno));

create table loan (
loannumber int,
branchname varchar(50),
amount int,
foreign key(branchname) references branch(branchname));
```

Structure of the table

Desc branch;

	Field	Type	Null	Key	Default	Extra
▶	branchname	varchar(50)	NO	PRI	NULL	
	branchcity	varchar(50)	YES		NULL	
	assets	int	YES		NULL	

Desc bankaccount;

	Field	Type	Null	Key	Default	Extra
▶	accno	int	NO	PRI	NULL	
	branchname	varchar(50)	YES	MUL	NULL	
	balance	int	YES		NULL	

Desc bankcustomer;

	Field	Type	Null	Key	Default	Extra
▶	customername	varchar(20)	NO	PRI	NULL	
	customerstreet	varchar(50)	YES		NULL	
	city	varchar(50)	YES		NULL	

Desc depositer;

	Field	Type	Null	Key	Default	Extra
▶	customername	varchar(20)	YES	MUL	NULL	
	accno	int	YES	MUL	NULL	

Desc loan;

	Field	Type	Null	Key	Default	Extra
▶	loannumber	int	YES		NULL	
	branchname	varchar(50)	YES	MUL	NULL	
	amount	int	YES		NULL	

Inserting Values to the table

```
insert into branch values("SBI_chamrajpet","bengaluru",50000);
insert into branch values("SBI_residencyroad","bengaluru",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;
```

branchname	branchcity	assets
SBI_chamrajpet	bengaluru	50000
SBI_jantarmantar	delhi	20000
SBI_parlimentroad	delhi	10000
SBI_residencyroad	bengaluru	10000
SBI_shivajiroad	bombay	20000
NULL	NULL	NULL

```
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;
```

accno	branchname	balance
1	SBI_chamrajpet	2100
2	SBI_residencyroad	5250
4	SBI_parlimentroad	9450
5	SBI_jantarmantar	8400
8	SBI_residencyroad	4200
9	SBI_parlimentroad	3150
10	SBI_residencyroad	5250
11	SBI_jantarmantar	2100
NULL	NULL	NULL

```
insert into bankcustomer values("avinash","bull_temple_road","bengaluru");
insert into bankcustomer values("dinesh","bannergatt_road","bengaluru");
insert into bankcustomer values("mohan","nationalcollege_road","bengaluru");
insert into bankcustomer values("nikil","akbar_road","delhi");
insert into bankcustomer values("ravi","prithviraj_road","delhi");
select*from bankcustomer;
```

customername	customerstreet	city
avinash	bull_temple_road	bengaluru
dinesh	bannergatt_road	bengaluru
mohan	nationalcollege_road	bengaluru
nikil	akbar_road	delhi
ravi	prithviraj_road	delhi
NULL	NULL	NULL

```
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;
```

customername	accno
avinash	1
dinesh	2
nikil	4
ravi	5
avinash	8
nikil	9
dinesh	10
nikil	11

```
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;
```

loannumber	branchname	amount
1	SBI_chamrajpet	1000
2	SBI_residencyroad	2000
3	SBI_shivajiroad	3000
4	SBI_parlimentroad	4000
5	SBI_jantarmantar	5000

Experiment 4: More Queries on Bank Database

Queries (Questions and Output)

- **Find all the customers who have at least two deposits at the same branch (Ex. 'SBI_ResidencyRoad').**

  select customername from depositer d,bankaccount a
  where d.accno=a.accno and a.branchname="SBI_residencyroad"
  group by d.customername having count(d.customername)>=2;

	customername
▶	dinesh

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

  select customername,accno from depositer
  where accno in (select accno from bankaccount a,branch b
  where a.branchname=b.branchname and b.branchcity="delhi");

	customername	accno
▶	ravi	5
	nikil	11
	nikil	4
	nikil	9

- **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");

  select*from bankaccount;

	accno	branchname	balance
▶	1	SBI_chamrajpet	2100
	2	SBI_residencyroad	5250
	4	SBI_parlimentroad	9450
	5	SBI_jantarmantar	8400
	8	SBI_residencyroad	4200
	9	SBI_parlimentroad	3150
	10	SBI_residencyroad	5250
	11	SBI_jantarmantar	2100
*	NULL	NULL	NULL

- **CREATE A VIEW WHICH GIVES EACH BRANCH THE SUM OF THE AMOUNT OF ALL THE LOANS AT THE BRANCH.**

```
create view v2 as
select branchname,sum(balance) from bankaccount group by branchname;
select *from v2;
```

	branchname	sum(balance)
▶	SBI_chamrajpet	2100
	SBI_jantarmantar	10500
	SBI_parlimentroad	12600
	SBI_residencyroad	14700

- **THE ANNUAL INTEREST PAYMENTS ARE MADE AND ALL BRANCHES ARE TO BE INCREASED BY 5%.**

```
update bankaccount set balance=balance*1.05;
set sql_safe_updates=0;
select *from bankaccount;
```

	accno	branchname	balance
▶	1	SBI_chamrajpet	2100
	2	SBI_residencyroad	5250
	4	SBI_parlimentroad	9450
	5	SBI_jantarmantar	8400
	8	SBI_residencyroad	4200
	9	SBI_parlimentroad	3150
	10	SBI_residencyroad	5250
	11	SBI_jantarmantar	2100
*	NULL	NULL	NULL

- **FIND ALL CUSTOMERS HAVING A LAON, AN ACCOUNT OR BOTH AT THE BANK**

```
select distinct customername from depositer
union(select customername from bankcustomer);
```

	customername
▶	avinash
	dinesh
	nikil
	ravi
	mohan

- **LIST THE ENTIRE LOAN RELATION IN THE DESCENDING ORDER OF AMOUNT**

```
select *from loan order by amount desc;
```

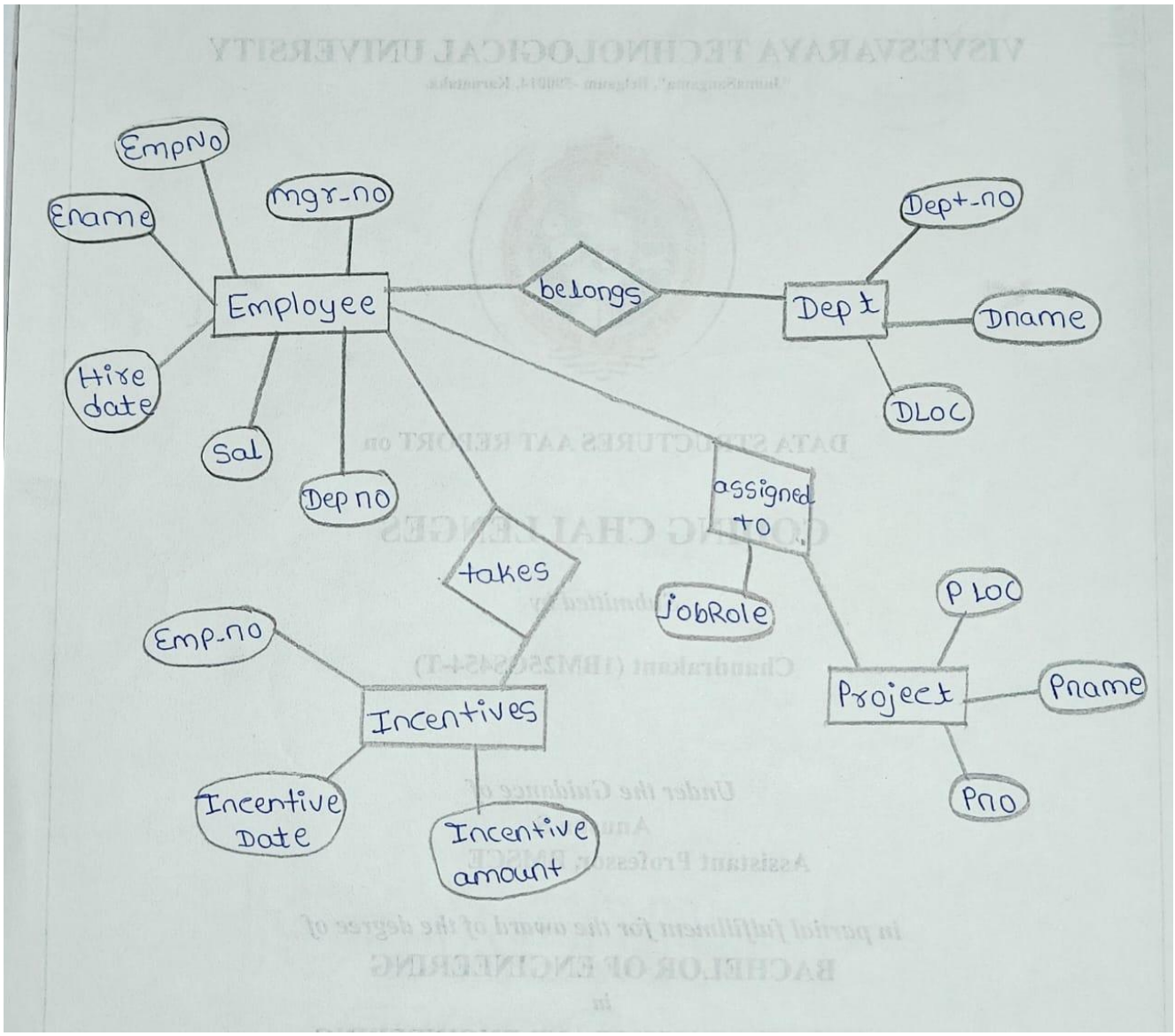
	loannumber	branchname	amount
▶	5	SBI_jantarmantar	5000
	4	SBI_parlimentroad	4000
	3	SBI_shivajiroad	3000
	2	SBI_residencyroad	2000
	1	SBI_chamrajpet	1000

Experiment 5: Employee Database

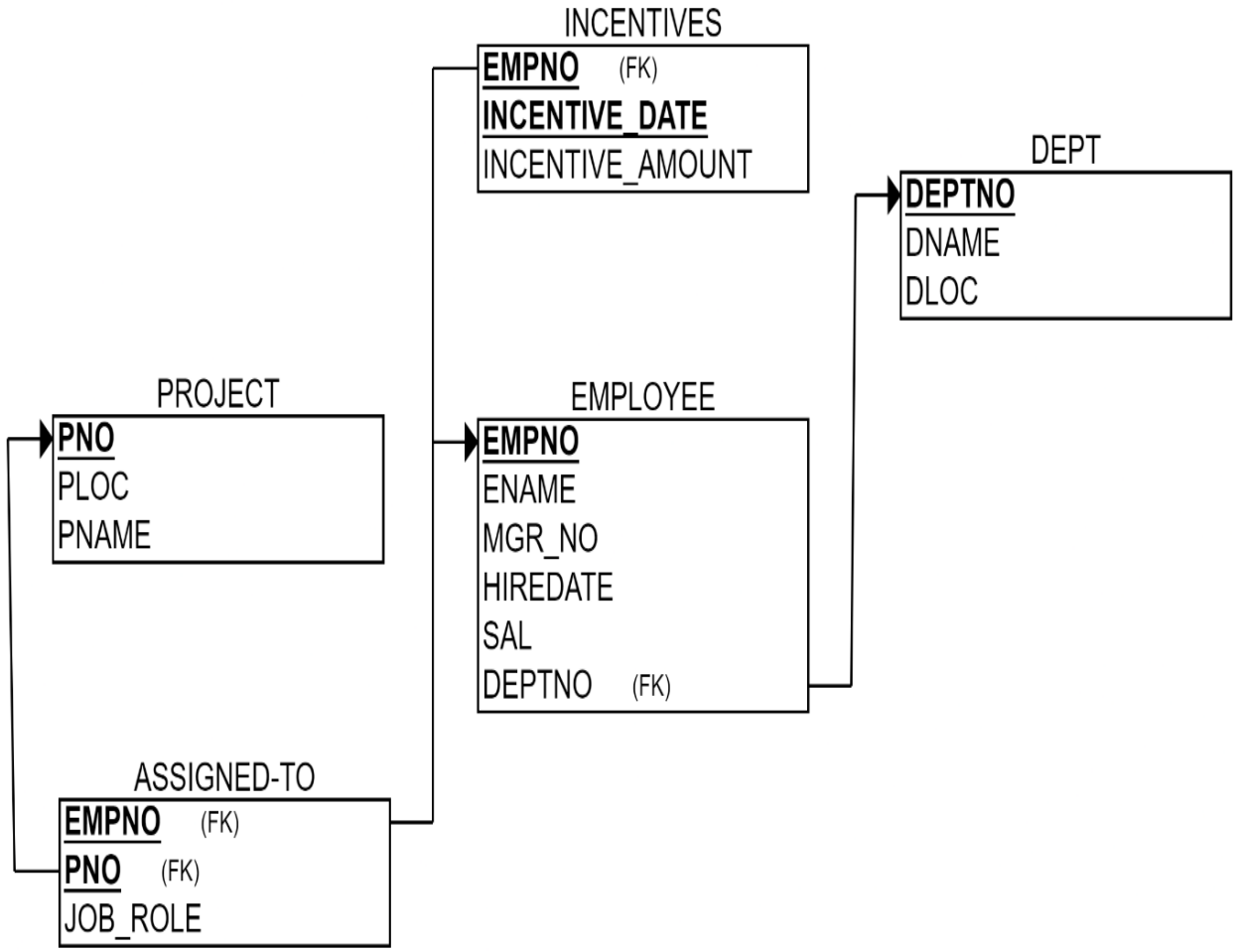
Specification of Insurance Database Application

The employee database must record each employee's identifying number, name, manager reference, hire date, salary, and department affiliation while also tracking departmental details, project assignments (including the role an employee plays on a project), and any incentive payments given to employees. Every employee is represented by a unique employee number and has a hire date and salary that must be valid; the manager field is a self-referencing link that must, if present, point to an existing employee and must never create a circular management chain or reference the employee themself. Departments are identified by a unique department number and include a department name and location; every department referenced by an employee or by other structures must exist in the department table, and departments may contain zero or many employees. Projects are recorded with a unique project number, project name and project location; employees may be assigned to multiple projects and each project may have many employees, with each assignment carrying the employee's job role for that project — duplicate assignments of the same employee to the same project are disallowed. Incentive payments are recorded with the employee reference, the incentive date and the incentive amount; an incentive entry must reference an existing employee and incentive amounts must be non-negative and dated on or after the employee's hire date. Referential integrity must be enforced so that employee records cannot reference non-existent departments, projects, or managers, and assignment and incentive records cannot exist without corresponding employee, project, or department records as appropriate. Salary, incentive amounts, and any monetary fields must be constrained to valid numeric ranges and hire/ incentive dates must be valid calendar dates (and typically not future-dated unless business rules permit). Deletion and update policies must preserve historical consistency: deleting an employee who appears as a manager, as a project assignee, or in incentive records should be prevented or should be handled via controlled archival, reassignment, or soft-delete flags rather than hard deletion to preserve audit trails; similarly, changing a department or project identifier must either be disallowed if it would orphan historical records or handled by introducing immutable surrogate keys. Business rules include preventing circular manager chains, ensuring an employee's manager (if specified) cannot be the employee themself, disallowing duplicate project-assignments, requiring that incentive dates fall within the employee's employment window, and optionally requiring at least one project assignment or at least one incentive record depending on policy for reporting. Implementation should use primary-key and foreign-key constraints for identity and linkage, unique constraints to prevent duplicate assignments, check constraints for monetary and date ranges, and application logic or triggers for complex temporal or graph constraints (like cycle detection in management relationships and enforcing non-overlap or other schedule-related rules if assignments gain temporal attributes later). The system must therefore reliably support queries such as employee reporting lines, department staffing lists, project rosters with job roles, incentive payment histories, salary analyses, and audit reports while maintaining data integrity, preventing inconsistent deletions, and preserving a complete historical record for HR and compliance needs.

Entity Relationship Diagram



Department (Dept_no;int, dname;String, dloc;String)
Employee (Emp_no;int, ename;String, mgr_no;String, hire_date;date, salary;double, dept_no;int)
Intentives (emp_no;int, incentive_date;date, incentive_ammount;double)
Project (pno;int, pname;String, ploc;String)
Assigned_To (emp_no;int, pno;int, jobrole;string)



Create database

```
create database Employee;

use Employee;
```

Create table

```
create table department(
dept_no decimal(2,0) primary key,
dname varchar(20),
dloc varchar(20));

create table employee (
emp_no decimal (4,0) primary key,
ename varchar(20),
mgr_no decimal(4,0),
hire_date date,
sallary decimal(7,2),
dept_no decimal(2,0),
foreign key (dept_no) references department (dept_no) on delete cascade on update cascade);

create table incentive(
emp_no decimal(4,0) references employee (emp_no) on delete cascade on update cascade,
incentive_date date,
incentive_amount decimal(10,2),
primary key (emp_no,incentive_date));

create table project (
pno int primary key,
pname varchar(30)not null,
ploc varchar(30));

create table assigned_to(
emp_no decimal(4,0) references employee(emp_no) on delete cascade on update cascade,
pno int references project(pno) on delete cascade on update cascade,
job_role varchar(30),
primary key (emp_no,pno));
```

Structure of the table

Desc department

	Field	Type	Null	Key	Default	Extra
▶	dept_no	decimal(2,0)	NO	PRI	NULL	
	dname	varchar(20)	YES		NULL	
	dloc	varchar(20)	YES		NULL	

Decs employee

	Field	Type	Null	Key	Default	Extra
▶	emp_no	decimal(4,0)	NO	PRI	NULL	
	ename	varchar(20)	YES		NULL	
	mgr_no	decimal(4,0)	YES		NULL	
	hire_date	date	YES		NULL	
	sallary	decimal(7,2)	YES		NULL	
	dept_no	decimal(2,0)	YES	MUL	NULL	

Desc incentives

	Field	Type	Null	Key	Default	Extra
▶	emp_no	decimal(4,0)	NO	PRI	NULL	
	incentive_date	date	NO	PRI	NULL	
	incentive_amount	decimal(10,2)	YES		NULL	

Desc project

	Field	Type	Null	Key	Default	Extra
▶	pno	int	NO	PRI	NULL	
	pname	varchar(30)	NO		NULL	
	ploc	varchar(30)	YES		NULL	

Desc assigned_to

	Field	Type	Null	Key	Default	Extra
▶	emp_no	decimal(4,0)	NO	PRI	NULL	
	pno	int	NO	PRI	NULL	
	job_role	varchar(30)	YES		NULL	

Inserting Values to the table

```
insert into department values("10","research","bengaluru");
insert into department values("20","accounting","mumbai");
insert into department values("30","sales","delhi");
insert into department values("40","operation","chennai");
select *from department;
```

dept_no	dname	dloc
10	research	bengaluru
20	accounting	mumbai
30	sales	delhi
40	operation	chennai
NULL	NULL	NULL

```
insert into project values(101,"AI project","bengaluru");
insert into project values(102,"IOT","hyderabad");
insert into project values(103,"BLOCKCHAIN","bengaluru");
insert into project values(104,"DATA SCIENCE","mysuru");
insert into project values(105,"AUTONOMUS SYSTEMS","pune");
select *from project;
```

pno	pname	ploc
101	AI project	bengaluru
102	IOT	hyderabad
103	BLOCKCHAIN	bengaluru
104	DATA SCIENCE	mysuru
105	AUTONOMUS SYSTEMS	pune
NULL	NULL	NULL

```
insert into employee values(7369,"adarsh",7902,'2012-12-17', 80000,20);
insert into employee values(7499,"shruthi",7698,'2013-02-20', 16000,30);
insert into employee values(7521,"anvitha",7698,'2015-02-22', 12500,30);
insert into employee values(7566,"tanvir",7839,'2008-04-02', 29750,20);
insert into employee values(7654,"ramesh",7698,'2014-09-28', 12500,30);
insert into employee values(7698,"kumar",7839,'2015-05-01', 28500,30);
insert into employee values(7782,"clark",7839,'2017-06-09', 24500,10);
insert into employee values(7788,"scott",7566,'2010-12-09', 30000,20);
insert into employee values(7839,"king",null,'2009-11-17', 99999.99,10);
insert into employee values(7844,"turner",7698,'2010-09-08', 15000,30);
insert into employee values(7876,"adams",7788,'2013-01-12', 11000,20);
insert into employee values(7900,"james",7698,'2017-12-03', 9500,30);
insert into employee values(7902,"ford",7566,'2010-12-03', 30000,20);
select *from employee;
```

emp_no	ename	mgr_no	hire_date	sallary	dept_no
7369	adarsh	7902	2012-12-17	80000.00	20
7499	shruthi	7698	2013-02-20	16000.00	30
7521	anvitha	7698	2015-02-22	12500.00	30
7566	tanvir	7839	2008-04-02	29750.00	20
7654	ramesh	7698	2014-09-28	12500.00	30
7698	kumar	7839	2015-05-01	28500.00	30
7782	clark	7839	2017-06-09	24500.00	10
7788	scott	7566	2010-12-09	30000.00	20
7839	king	NULL	2009-11-17	99999.99	10
7844	turner	7698	2010-09-08	15000.00	30
7876	adams	7788	2013-01-12	11000.00	20
7900	james	7698	2017-12-03	9500.00	30
7902	ford	7566	2010-12-03	30000.00	20
NULL	NULL	NULL	NULL	NULL	NULL

```
insert into incentive values(7499,'2019-02-01', 5000);
insert into incentive values(7521,'2019-02-01', 8000);
insert into incentive values(7521,'2019-03-01', 2500);
insert into incentive values(7566,'2022-02-01', 5070);
insert into incentive values(7654,'2020-02-01', 2000);
insert into incentive values(7654,'2022-02-01', 879);
insert into incentive values(7698,'2019-03-01', 500);
insert into incentive values(7698,'2020-03-01', 9000);
insert into incentive values(7698,'2022-04-01', 4500);
select*from incentive;
```

	emp_no	incentive_date	incentive_amount
▶	7499	2019-02-01	5000.00
	7521	2019-02-01	8000.00
	7521	2019-03-01	2500.00
	7566	2022-02-01	5070.00
	7654	2020-02-01	2000.00
	7654	2022-02-01	879.00
	7698	2019-03-01	500.00
	7698	2020-03-01	9000.00
	7698	2022-04-01	4500.00
*	NULL	NULL	NULL

```
insert into assigned_to values(7499,101,"software engineer");
insert into assigned_to values(7499,102,"software engineer");
insert into assigned_to values(7521,101,"software architect");
insert into assigned_to values(7521,102,"software engineer");
insert into assigned_to values(7566,101,"project manager");
insert into assigned_to values(7654,102,"sales");
insert into assigned_to values(7654,103,"cyber security");
insert into assigned_to values(7698,104,"software engineer");
insert into assigned_to values(7839,104,"general manager");
insert into assigned_to values(7900,105,"software engineer");
select *from assigned_to;
```

	emp_no	pno	job_role
▶	7499	101	software engineer
	7499	102	software engineer
	7521	101	software architect
	7521	102	software engineer
	7566	101	project manager
	7654	102	sales
	7654	103	cyber security
	7698	104	software engineer
	7839	104	general manager
	7900	105	software engineer
*	NULL	NULL	NULL

Experiment 6:More Queries on Employee Database

Queries (Questions and Output)

- **Display those managers name whose salary is more than average salary of his employee**

```
select emp_no,ename,sallary from employee e
where e.emp_no in
(select mgr_no from employee) and e.sallary>
(select avg(sallary)from employee m where m.mgr_no=e.emp_no);
```

	emp_no	ename	sallary
▶	7698	kumar	28500.00
	7839	king	99999.99
	7788	scott	30000.00
*	NULL	NULL	NULL

- **List the name of the managers with the most employees**

```
select m.ename ,count(*) from employee e,employee m
where e.mgr_no=m.emp_no group by m.ename having count(*)=
(select max(mycount)from (select count(*) mycount from employee group by mgr_no)a);
```

	ename	count(*)
▶	kumar	5

- **Display those managers name whose salary is more than average salary of his employee**

```
select emp_no,ename,sallary from employee e
where e.emp_no in
(select mgr_no from employee) and e.sallary>
(select avg(sallary)from employee m where m.mgr_no=e.emp_no);
```

	emp_no	ename	sallary
▶	7698	kumar	28500.00
	7839	king	99999.99
	7788	scott	30000.00
*	NULL	NULL	NULL

- **SQL Query to find the employee details who got second maximum incentive in Febrauary 2019.**

```
select *from employee e,incentive i
where e.emp_no=i.emp_no and 2 = ( select count(*)
from incentive j
where i.incentive_amount <= j.incentive_amount );
```

	emp_no	ename	mgr_no	hire_date	sallary	dept_no	emp_no	incentive_date	incentive_amount
▶	7521	anvitha	7698	2015-02-22	12500.00	30	7521	2019-02-01	8000.00

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

```
SELECT e.emp_no
FROM employee e, assigned_to a, project p
WHERE e.emp_no = a.emp_no AND a.pno = p.pno
AND p.ploc IN ('BENGALURU', 'HYDERABAD', 'MYSURU');
```

	emp_no
▶	7499
	7499
	7521
	7521
	7566
	7654
	7654
	7698
	7839

- **Write a SQL query to find those employees whose net pay are higher than or equal to the salary of any other employee in the company**

```
SELECT distinct e.ename
FROM employee e,incentive i
WHERE (SELECT max(sallary+incentive_amount)
FROM employee,incentive) >= ANY
(SELECT sallary
FROM employee e1
where e.dept_no=e1.dept_no);
```

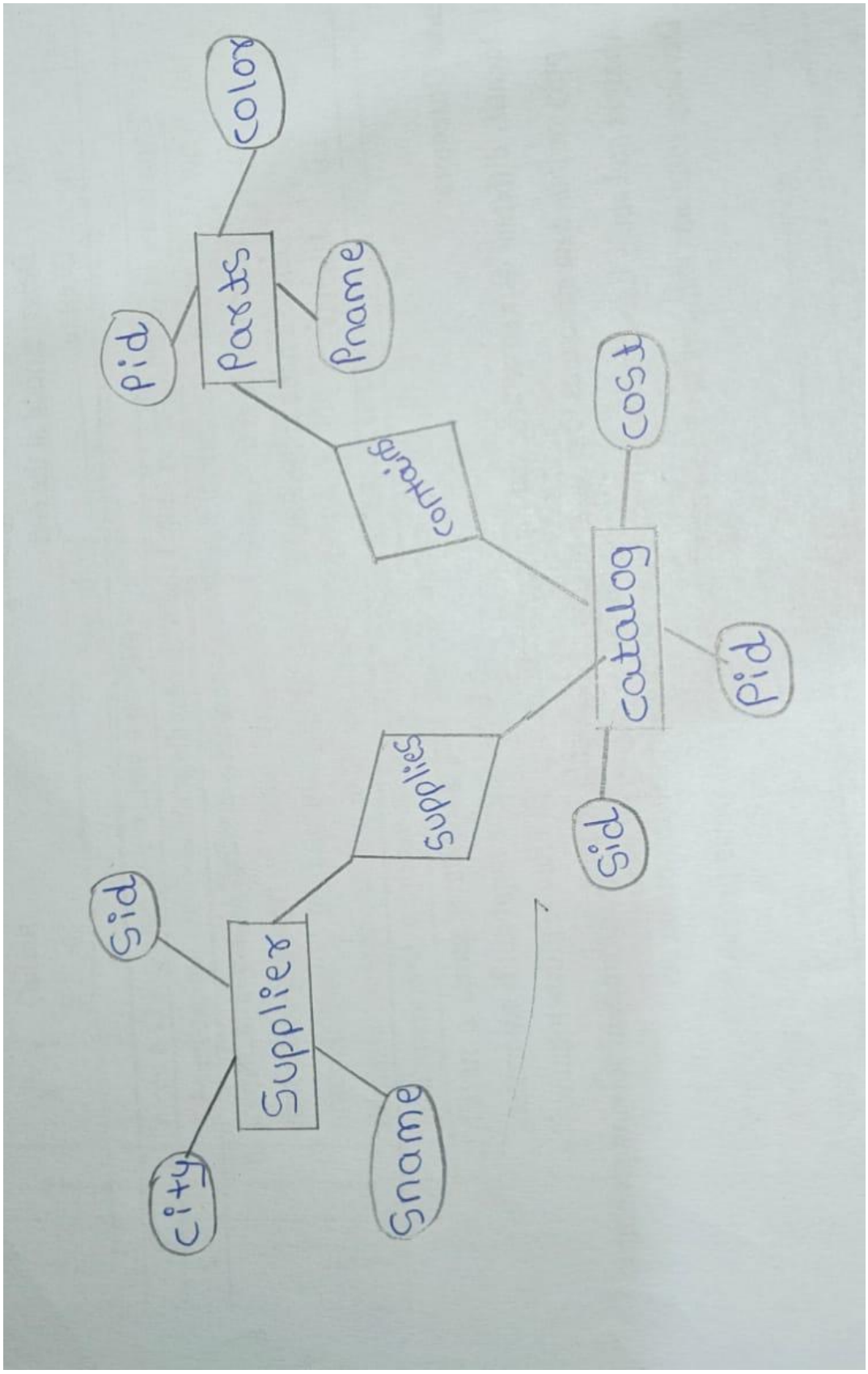
ename
adams
adarsh
anvitha
dark
ford
james
king
kumar
ramesh
scott
shruthi
tanvir
turner

Experiment 7: Supplier Database

Specification of Insurance Database Application

The supplier database must store information about suppliers, the parts they provide, and the prices at which each part is offered so that purchasing, analysis, and reporting can be done accurately. Each supplier is uniquely identified by a supplier ID and is recorded with a name and the city in which the supplier is located; each part is uniquely identified by a part ID and includes a part name and a colour. The system must maintain a catalog that links suppliers to the parts they supply and records the cost at which a given supplier sells a given part. Every catalog entry must reference an existing supplier and an existing part, and there must be no duplicate entries for the same combination of supplier and part, so that at most one current price record exists per supplier–part pair. Costs must be valid numeric values and strictly non-negative, and business rules may specify upper limits or currency formats that must be enforced consistently. The data model must support the possibility that a supplier can provide many different parts, that a part can be supplied by many different suppliers, and that some suppliers or parts may temporarily have no catalog entries if they are inactive or not currently traded. Referential integrity must be enforced so that a supplier or part cannot be deleted while still referenced in the catalog unless such deletion is handled by controlled archival or cascade rules that preserve historical price information; in general, historical catalog data should not be lost, as it may be required for audits or trend analysis. The system should allow queries such as “find all suppliers for a given part,” “list all parts provided by a given supplier,” “retrieve the cheapest supplier for each part,” and “analyse supplier coverage by city,” and must therefore guarantee that identifiers are unique, relationships between suppliers, parts, and catalog entries are consistent, and price information is accurate and reliably maintained over time.

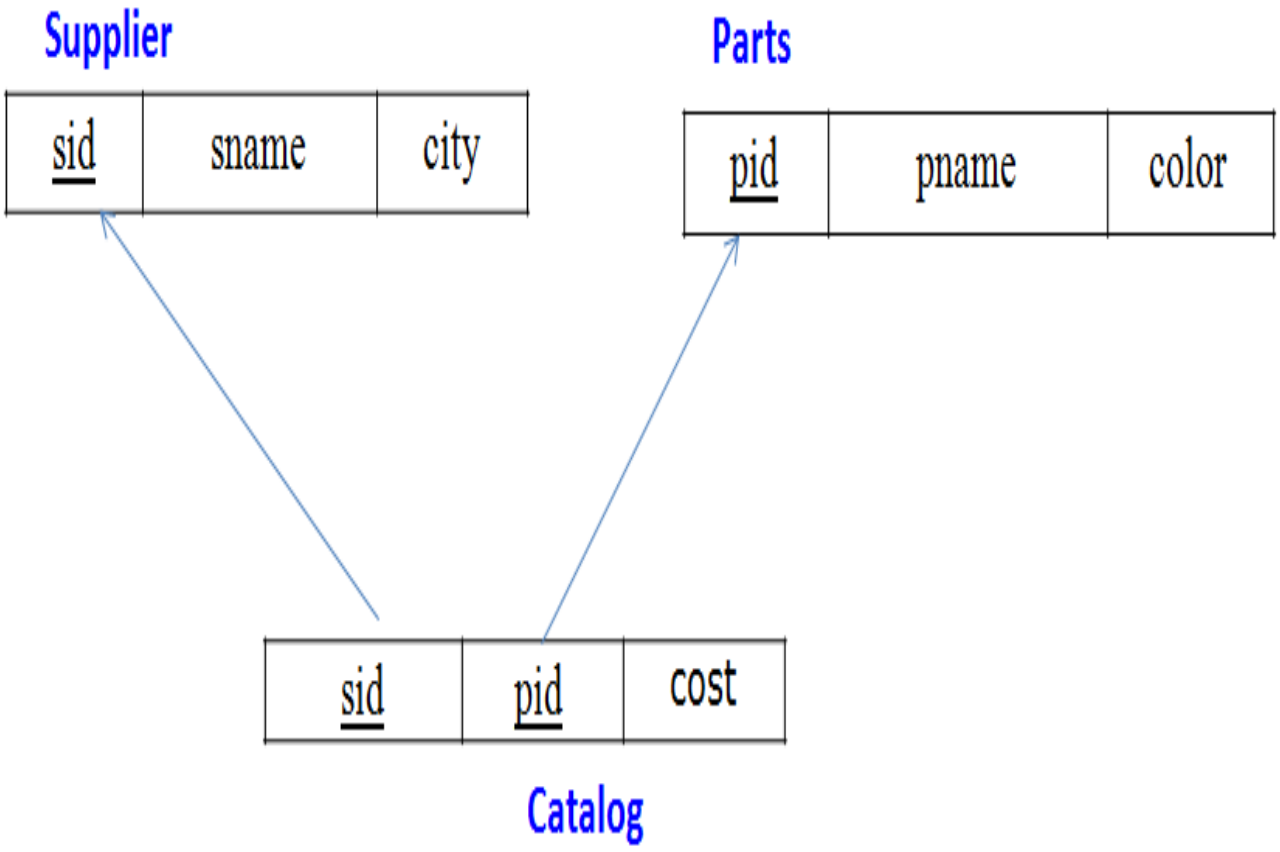
Entity Relationship Diagram



supplier (sid;int, sname;string, city;string)

parts (pid;int, pname;string, color;string)

catalog (sid;int, pid;int, cost;double)



Create database

```
create database supplier;

use supplier;
```

Create table

```
create table SUPPLIERS(
sid integer(5) primary key,
sname varchar(20),
city varchar(20));

create table PARTS(
pid integer(5) primary key,
pname varchar(20),
color varchar(10));

create table CATALOG(
sid integer(5),
pid integer(5),
foreign key(sid) references SUPPLIERS(sid),
foreign key(pid) references PARTS(pid),
cost float(6),
primary key(sid, pid));
```

Structure of the table

Desc supplier;

Field	Type	Null	Key	Default	Extra
sid	int	NO	PRI	NULL	
sname	varchar(20)	YES		NULL	
city	varchar(20)	YES		NULL	

Desc parts;

Field	Type	Null	Key	Default	Extra
pid	int	NO	PRI	NULL	
pname	varchar(20)	YES		NULL	
color	varchar(10)	YES		NULL	

Desc catalog;

Field	Type	Null	Key	Default	Extra
sid	int	NO	PRI	NULL	
pid	int	NO	PRI	NULL	
cost	float	YES		NULL	

Inserting Values to the table

```
insert into SUPPLIERS values(10001,'acme widget','bangolore');
insert into SUPPLIERS values(10002,'johns','kolkota');
insert into SUPPLIERS values(10003,'vimal','mumbai');
insert into SUPPLIERS values(10004,'reliance','delhi');
select *from SUPPLIERS;
```

	sid	sname	city
▶	10001	acme widget	bangolore
	10002	johns	kolkota
	10003	vimal	mumbai
	10004	reliance	delhi
*	NULL	NULL	NULL

```
insert into PARTS values(20001,'book','red');
insert into PARTS values(20002,'pen','red');
insert into PARTS values(20003,'pencil','green');
insert into PARTS values(20004,'mobile','green');
insert into PARTS values(20005,'charger','black');
select *from PARTS;
```

	pid	pname	color
▶	20001	book	red
	20002	pen	red
	20003	pencil	green
	20004	mobile	green
	20005	charger	black
*	NULL	NULL	NULL

```
insert into CATALOG values(10001,20001,10);
insert into CATALOG values(10001,20002,10);
insert into CATALOG values(10001,20003,30);
insert into CATALOG values(10001,20004,10);
insert into CATALOG values(10001,20005,10);
insert into CATALOG values(10002,20001,10);
insert into CATALOG values(10002,20002,20);
insert into CATALOG values(10003,20003,30);
insert into CATALOG values(10004,20003,40);
select *from CATALOG;
```

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

Experiment 8: More Queries on Supplier Database

Queries (Questions and Output)

- **Find the pnames of parts for which there is some supplier**

```
SELECT DISTINCT
 P.pname
FROM Parts P, Catalog C
WHERE P.pid = C.pid;.
```

pname
book
pen
pencil
mobile
charger

- **Find the snames of suppliers who supply every part.**

```
SELECT S.sname FROM Suppliers S
WHERE (( SELECT count(P.pid)
 FROM Parts P ) =( SELECT count(C.pid)
 FROM Catalog C WHERE C.sid = S.sid ));
```

sname
acme widget

- **Find the snames of suppliers who supply every red part.**

```
SELECT S.sname FROM Suppliers S
WHERE(( SELECT count(P.pid)
FROM Parts P where color='red' ) =
(SELECT count(C.pid)
FROM Catalog C, Parts P
WHERE C.sid = S.sid AND
C.pid = P.pid AND P.color = 'red' ));
```

	sname
▶	acme widget
	johns

- **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
 WHERE C.cost >( SELECT AVG (C1.cost)
FROM CATALOG C1 WHERE C1.pid = C.pid );
.
```

	sid
▶	10002
	10004

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

```
SELECT P.pid, S.sname
FROM Parts P, Suppliers S, Catalog C
WHERE C.pid = P.pid
AND C.sid = S.sid
AND C.cost = (SELECT MAX(C1.cost)
FROM Catalog C1
WHERE C1.pid = P.pid);
```

	pid	sname
▶	20001	acme widget
	20004	acme widget
	20005	acme widget
	20001	johns
	20002	johns
	20005	johns
	20003	reliance

- **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(DISTINCT C.pid) AS total_parts
FROM Suppliers S
LEFT JOIN Catalog C ON S.sid = C.sid
GROUP BY S.sid, S.sname;

SELECT * FROM Supplier_Part_Count;
```

	sid	sname	total_parts
▶	10001	acme widget	5
	10002	johns	3
	10003	vimal	1
	10004	reliance	1

- **Create a view of the most expensive supplier for each part**

```
CREATE VIEW Most_Expensive_Supplier_Per_Part AS
SELECT P.pid, P.pname, S.sid, S.sname, C.cost
FROM Parts P
JOIN Catalog C ON P.pid = C.pid
JOIN Suppliers S ON S.sid = C.sid
WHERE C.cost = (
        SELECT MAX(C1.cost)
        FROM Catalog C1
       WHERE C1.pid = P.pid);

SELECT * FROM Most_Expensive_Supplier_Per_Part;
```

	pid	pname	sid	sname	cost
▶	20001	book	10001	acme widget	10
	20004	mobile	10001	acme widget	10
	20005	charger	10001	acme widget	10
	20001	book	10002	johns	10
	20002	pen	10002	johns	20
	20005	charger	10002	johns	10
	20003	pencil	10004	reliance	40

Experiment 9: NOSQL Installation in Cloud

Aim:

To install and configure a NoSQL database in the cloud using MongoDB Atlas and perform basic database operations.

Software Requirements:

- Web browser (Chrome / Firefox)
- Internet connection
- MongoDB Atlas account

Procedure:

1. Open a web browser and visit https://www.mongodb.com/atlas.
2. Click on **Sign Up** and create a MongoDB Atlas account using email or Google login.
3. After successful login, create a **new project**.
4. Click on **Build a Database** and choose the **Free Tier (M0)** cluster.
5. Select the cloud provider and region (preferably nearest region).
6. Create the cluster and wait for deployment to complete.
7. Configure **Database Access** by creating a database user with username and password.
8. Configure **Network Access** by allowing access from the current IP address (or 0.0.0.0/0).
9. Click **Connect** and obtain the MongoDB connection string.
10. Use MongoDB Compass or MongoDB Shell to connect to the cloud database using the provided connection string.
11. Create a database and collections to perform operations such as insert, find, update, and delete.

Result:

MongoDB Atlas cloud database was successfully created and connected. Basic NoSQL operations were performed on the cloud-hosted database.

Conclusion:

Thus, a NoSQL database was successfully installed and configured in the cloud using MongoDB Atlas, demonstrating cloud-based database deployment and access.

Experiment 10: NoSQL – Student Database

Specification of NoSQL – Student Database Application:

The NoSQL student database must store and manage student information using a document-oriented data model in MongoDB. Each student record is uniquely identified by a roll number and includes attributes such as age, contact number, and email ID. The database must support insertion of appropriate student records and allow modification of existing data, including updating a student's email ID based on roll number and replacing a student name with a new value for a specified roll number. It should also support administrative operations such as exporting the student collection to the local file system for backup or data transfer purposes and importing data from external CSV files into MongoDB collections. Additionally, the system must allow safe deletion of collections when required. The database should ensure flexible schema handling, efficient document updates, and reliable data persistence while demonstrating fundamental NoSQL operations such as insert, update, replace, export, import, and drop.

Create Database:

```
> use Student;
< switched to db Student
```

Insert Appropriate Values:

Insert at least 4 documents each with RollNo, Name, Age, ContactNo and EmailID attributes.

```
> db.Student.insertMany ([
    {RollNo: 10, Name: "ABC", Age: 20, ContactNo: 1112223334, EmailID: "CSE10.cs24@bmsce.ac.in"},
    {RollNo: 11, Name: "DEF", Age: 20, ContactNo: 2223334445, EmailID: "CSE11.cs24@bmsce.ac.in"},
    {RollNo: 12, Name: "GHI", Age: 20, ContactNo: 3334445556, EmailID: "CSE12.cs24@bmsce.ac.in"},
    {RollNo: 13, Name: "JKL", Age: 20, ContactNo: 4445556667, EmailID: "CSE13.cs24@bmsce.ac.in"}
  ]);
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('693ff9c98e185b767bf90899'),
      '1': ObjectId('693ff9c98e185b767bf9089a'),
      '2': ObjectId('693ff9c98e185b767bf9089b'),
      '3': ObjectId('693ff9c98e185b767bf9089c')
    }
  }
```

Queries:

1. Write a query to update EmailID of a student with RollNo 10.

```
> db.Student.updateOne(
    {RollNo: 10},
    {$set: {EmailID: "CSE10Updated.cs24@bmsce.ac.in"}}
  );
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
```

2. Replace the student's name from "DEF" to "FEM" of RollNo 11.

```
> db.Student.updateOne (
    {RollNo: 11},
    {$set: {Name: "FEM"}}
  );
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
```

3. Export the created table into Local File System.

   - Open your Collections Table.
   - Click on Export Data -> Export Full Collection.
   - Export File Type → CSV.
   - A dialog box will open. Choose file destination and click save.

	A	B	C	D	E
1	RollNo	Name	Age	ContactNo	EmailID
2	10	ABC	20	1112223334	CSE10Updated.cs24@bmsce.ac.in
3	11	FEM	20	2223334445	CSE11.cs24@bmsce.ac.in
4	12	GHI	20	3334445556	CSE12.cs24@bmsce.ac.in
5	13	JKL	20	4445556667	CSE13.cs24@bmsce.ac.in

4. Drop the table.

```
> db.Student.drop();
< true
```

5. Import a given a CSV dataset from your Local File System into MongoDB collection.
   - Open your collections table.
   - Click Add Data → Import JSON or CSV file.
   - Select previously saved CSV file (students.csv).
   - Click Import.

Experiment 11: NoSQL – Customer Database

Specification of NoSQL – Customer Database Application:

The NoSQL customer database is designed to store and manage customer account information using a document-based data model in MongoDB. Each customer record is uniquely identified by a customer ID and includes attributes such as account balance and account type. The database must support insertion of multiple customer records and enable querying of customers whose total account balance exceeds a specified value for a given account type. It should also allow aggregation operations to determine the minimum and maximum account balances for each customer. Additionally, the system must support administrative operations such as exporting the customer collection to the local file system for backup purposes, importing customer data from external CSV files, and safely dropping collections when required. The database should demonstrate efficient data storage, flexible querying, and basic aggregation capabilities of NoSQL databases.

Create Database:

```
> use customer
< switched to db customer
```

Insert Appropriate Values:

Insert at least 4 documents each with Cust_id, Acc_Bal, Acc_Type attributes.

```
> db.customer.insertMany ([
    {Cust_id: 101, Acc_Bal: 1234, Acc_Type: "Z"},
    {Cust_id: 101, Acc_Bal: 777, Acc_Type: "Y"},
    {Cust_id: 103, Acc_Bal: 1634, Acc_Type: "Z"},
    {Cust_id: 104, Acc_Bal: 987, Acc_Type: "Z"},
    {Cust_id: 104, Acc_Bal: 1009, Acc_Type: "Y"}
  ]);
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('6940081e104a74526bf2021c'),
      '1': ObjectId('6940081e104a74526bf2021d'),
      '2': ObjectId('6940081e104a74526bf2021e'),
      '3': ObjectId('6940081e104a74526bf2021f'),
      '4': ObjectId('6940081e104a74526bf20220')
    }
  }
```

Queries:

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

```
> db.customer.aggregate ([
    {$match: {Acc_Type: 'Z'}},
    {$group: {
      _id: "$Cust_id",
      total_balance: {$sum: "$Acc_Bal"}
    }},
    {$match: {total_balance: {$gt: 1200}}}
  ]);
< {
    _id: 101,
    total_balance: 1234
  }
  {
    _id: 103,
    total_balance: 1634
  }
```

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

```
> db.customer.aggregate([
    {$group: {
        _id: "$Cust_id",
        min_balance: { $min: "$Acc_Bal" },
        max_balance: { $max: "$Acc_Bal" }
      }
    }
  ]);
< {
    _id: 104,
    min_balance: 987,
    max_balance: 1009
  }
  {
    _id: 101,
    min_balance: 777,
    max_balance: 1234
  }
  {
    _id: 103,
    min_balance: 1634,
    max_balance: 1634
  }
```

3. Export the created table into Local File System.
   - Open your Collections Table.
   - Click on Export Data → Export Full Collection.
   - Export File Type → CSV.
   - A dialog box will open. Choose file destination and click save.

	A	B	C
1	Cust_id	Acc_Bal	Acc_Type
2	101	1234	Z
3	101	777	Y
4	103	1634	Z
5	104	987	Z
6	104	1009	Y

4. Drop the table.

```
> db.customer.drop()
< true
```

5. Import a given a CSV dataset from your Local File System into MongoDB collection.
    - Open your collections table.
    - Click Add Data → Import JSON or CSV file.
    - Select previously saved CSV file (customer.csv).
    - Click Import.

Experiment 12: NoSQL – Restaurant Database

Specification of NoSQL – Restaurant Database Application:

The NoSQL restaurant database is designed to store and manage restaurant-related information using a document-oriented data model in MongoDB. Each restaurant document contains details such as restaurant ID, name, location information, cuisine type, and inspection scores. The database must support retrieval of all restaurant records and allow sorting of restaurant data based on specified attributes. It should enable selective querying to extract specific fields such as restaurant ID, name, town, and cuisine for restaurants meeting given score criteria. Additionally, the system must support aggregation operations to compute the average inspection score for each restaurant and perform pattern-based queries to identify restaurants located in areas with specific zip code formats. The database demonstrates flexible querying, sorting, filtering, and aggregation capabilities of MongoDB for real-world data analysis.

Create Database:

```
> use restaurant
< switched to db restaurant
```

Insert Appropriate Values:

Insert at least 4 documents each with Restaurant_id, Name, Cuisine, Address (Town, Zipcode), Score attributes.

```
> db.restaurant_details.insertOne([
   {restaurant_id: 1001,
    name: "Saffron & Sage",
    cuisine: "Modern Indian Fusion",
    address: {
      town: "Udaipur",
      zipcode: "101028"
    },
    score: 10.0
   }
 ]);
< {
   acknowledged: true,
   insertedId: ObjectId('69401559c7b448b946f298c3')
 }
```

```
> db.restaurant_details.insertOne([
   {restaurant_id: 1002,
    name: "The Gilded Noodle",
    cuisine: "Pan-Asian/Thai",
    address: {
      town: "Indore",
      zipcode: "452009"
    },
    score: 7.1
   }
 ]);
< {
   acknowledged: true,
   insertedId: ObjectId('69401566c7b448b946f298c4')
 }
```

```
> db.restaurant_details.insertOne([
   {restaurant_id: 1003,
    name: "Misty Malabar",
    cuisine: "South-Indian (Coastal)",
    address: {
      town: "Puducherry",
      zipcode: "105357"
    },
    score: 8.5
   }
 ]);
< {
   acknowledged: true,
   insertedId: ObjectId('6940156fc7b448b946f298c5')
 }
```

```
> db.restaurant_details.insertOne([
   {restaurant_id: 1004,
    name: "Velvet Vineyards",
    cuisine: "French-Italian",
    address: {
      town: "Mumbai",
      zipcode: "400001"
    },
    score: 18.5
   }
 ]);
< {
   acknowledged: true,
   insertedId: ObjectId('69401577c7b448b946f298c6')
 }
```

```
> db.restaurant_details.insertOne([
   {restaurant_id: 1005,
    name: "Indigo Masala",
    cuisine: "Indo-Mexican Fusion",
    address: {
      town: "Hyderabad",
      zipcode: "500108"
    },
    score: 12.5
   }
 ]);
< {
   acknowledged: true,
   insertedId: ObjectId('69401583c7b448b946f298c7')
 }
```

Queries:

1. Write a MongoDB query to display all the documents in the collection restaurants.

```
> db.restaurant_details.find();
< {
    _id: ObjectId('69401559c7b448b946f298c3'),
    restaurant_id: 1001,
    name: 'Saffron & Sage',
    cuisine: 'Modern Indian Fusion',
    address: {
      town: 'Udaipur',
      zipcode: '101028'
    },
    score: 10
  }
```

```
{
  _id: ObjectId('69401566c7b448b946f298c4'),
  restaurant_id: 1002,
  name: 'The Gilded Noodle',
  cuisine: 'Pan-Asian/Thai',
  address: {
    town: 'Indore',
    zipcode: '452009'
  },
  score: 7.1
}
```

```
{
  _id: ObjectId('6940156fc7b448b946f298c5'),
  restaurant_id: 1003,
  name: 'Misty Malabar',
  cuisine: 'South-Indian (Coastal)',
  address: {
    town: 'Puducherry',
    zipcode: '105357'
  },
  score: 8.5
}
```

```
{
  _id: ObjectId('69401577c7b448b946f298c6'),
  restaurant_id: 1004,
  name: 'Velvet Vineyards',
  cuisine: 'French-Italian',
  address: {
    town: 'Mumbai',
    zipcode: '400001'
  },
  score: 18.5
}
```

```
{
  _id: ObjectId('69401583c7b448b946f298c7'),
  restaurant_id: 1005,
  name: 'Indigo Masala',
  cuisine: 'Indo-Mexican Fusion',
  address: {
    town: 'Hyderabad',
    zipcode: '500108'
  },
  score: 12.5
}
```

2. Write a MongoDB query to arrange the name of the restaurants in descending along with all the columns.

```
> db.restaurant_details.find().sort({"name": -1});
< {
    _id: ObjectId('69401577c7b448b946f298c6'),
    restaurant_id: 1004,
    name: 'Velvet Vineyards',
    cuisine: 'French-Italian',
    address: {
      town: 'Mumbai',
      zipcode: '400001'
    },
    score: 18.5
  }
```

```
{
  _id: ObjectId('69401566c7b448b946f298c4'),
  restaurant_id: 1002,
  name: 'The Gilded Noodle',
  cuisine: 'Pan-Asian/Thai',
  address: {
    town: 'Indore',
    zipcode: '452009'
  },
  score: 7.1
}
```

```
< {
    _id: ObjectId('69401559c7b448b946f298c3'),
    restaurant_id: 1001,
    name: 'Saffron & Sage',
    cuisine: 'Modern Indian Fusion',
    address: {
      town: 'Udaipur',
      zipcode: '101028'
    },
    score: 10
  }
```

```
{
  _id: ObjectId('6940156fc7b448b946f298c5'),
  restaurant_id: 1003,
  name: 'Misty Malabar',
  cuisine: 'South-Indian (Coastal)',
  address: {
    town: 'Puducherry',
    zipcode: '105357'
  },
  score: 8.5
}
```

```
{
  _id: ObjectId('69401583c7b448b946f298c7'),
  restaurant_id: 1005,
  name: 'Indigo Masala',
  cuisine: 'Indo-Mexican Fusion',
  address: {
    town: 'Hyderabad',
    zipcode: '500108'
  },
  score: 12.5
}
```

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

```
> db.restaurant_details.find({score: {$lte: 10}});
< {
    _id: ObjectId('69401559c7b448b946f298c3'),
    restaurant_id: 1001,
    name: 'Saffron & Sage',
    cuisine: 'Modern Indian Fusion',
    address: {
      town: 'Udaipur',
      zipcode: '101028'
    },
    score: 10
  }
```

```
{
  _id: ObjectId('69401566c7b448b946f298c4'),
  restaurant_id: 1002,
  name: 'The Gilded Noodle',
  cuisine: 'Pan-Asian/Thai',
  address: {
    town: 'Indore',
    zipcode: '452009'
  },
  score: 7.1
}
```

```
{
  _id: ObjectId('6940156fc7b448b946f298c5'),
  restaurant_id: 1003,
  name: 'Misty Malabar',
  cuisine: 'South-Indian (Coastal)',
  address: {
    town: 'Puducherry',
    zipcode: '105357'
  },
  score: 8.5
}
```

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

```
> db.restaurant_details.aggregate([
    {
      $group: {
        _id: "$restaurant_id",
        avg_score: { $avg: "$score" }
      }
    }
  ]);
```

```
{
  _id: 1005,
  avg_score: 12.5
}
{
  _id: 1003,
  avg_score: 8.5
}
{
  _id: 1001,
  avg_score: 10
}
{
  _id: 1002,
  avg_score: 7.1
}
{
  _id: 1004,
  avg_score: 18.5
}
```

5. Write a MongoDB query to find the name and address of the restaurants that have a zip code that starts with '10'.

```
> db.restaurant_details.find(
    { "address.zipcode": { $regex: "^10" } },
    { name: 1, address: 1, _id: 0 }
  );
```

```
< {
    name: 'Saffron & Sage',
    address: {
      town: 'Udaipur',
      zipcode: '101028'
    }
  }
  {
    name: 'Misty Malabar',
    address: {
      town: 'Puducherry',
      zipcode: '105357'
    }
  }
```

Experiment 13: LeetCode Practice - I

Problem – 570. Managers with at least 5 direct reports:

570. Managers with at Least 5 Direct Reports

Medium | Topics | Companies | Hint

SQL Schema > Pandas Schema >

Table: `Employee`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| name        | varchar |
| department  | varchar |
| managerId   | int     |
+-------------+---------+
id is the primary key (column with unique values) for this table.
Each row of this table indicates the name of an employee, their department, and the id of
their manager.
If managerId is null, then the employee does not have a manager.
No employee will be the manager of themself.
```

Query:

Write a solution to find managers with at least five direct reports.

```
SELECT e.name
FROM Employee e, Employee m
WHERE e.id = m.managerId
GROUP BY m.managerId
HAVING COUNT(m.managerId) >= 5;
```

Input	Output

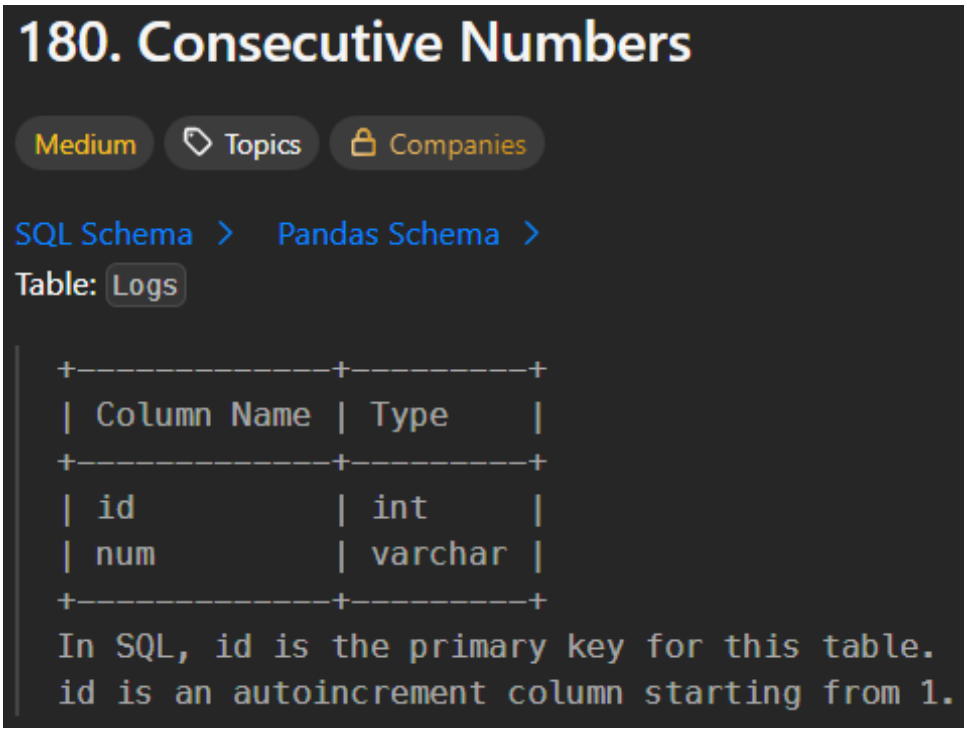
Employee =

id	name	department	managerId
101	John	A	null
102	Dan	A	101
103	James	A	101
104	Amy	A	101
105	Anne	A	101
106	Ron	B	101

name
John

Experiment 14: LeetCode Practice - II

Problem – 180. Consecutive Numbers:



Query:

Find all numbers that appear at least three times consecutively.

```
SELECT DISTINCT l1.num AS ConsecutiveNums
FROM Logs l1, Logs l2, Logs l3
WHERE l1.num = l2.num AND
        l2.num = l3.num AND
        l1.id = l2.id – 1 AND
        l2.id = l3.id - 1;
```

Input	Output
Logs = | id | num | | -- | --- | | 1 | 1 | | 2 | 1 | | 3 | 1 | | 4 | 2 | | 5 | 2 | | 6 | 2 | | 7 | 3 |	| ConsecutiveNums | | ------------- | | 1 | | 2 |