Date: 13-06-2025

SQL Practical Question Paper-1

Section A: Basics & Data Definition (10 Marks)

Q1. Differentiate between SQL and NoSQL. Provide two advantages and two disadvantages of each with real-world examples.

SQL is a structured relational DB which stores data in tables which consists of rows and columns.

Advantages of SQL:

➔ It follows an order to store data,

➔ Which helps us to find the required data faster and more efficiently.

Disadvantages of SQL:

➔ It is very time consuming to change the data schema.

➔ Limited Scalability.

NoSQL is an unstructured dynamic DB which stores values in random manner and has no fixed schema.

Advantages of NoSQL:

- We don’t have to follow any schemas to store data.
- We can insert data very easily.

Disadvantages of NoSQL:

- It is a time consuming process to find data compared to SQL.
- The data is not secure.

Q2.Given the below unnormalized data, convert it to 1NF, 2NF, and 3NF: Student (StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)

Q3. (a) Create a database named StudentDB. b) Create a table Students with fields: StudentID, Name, DOB, Email. c) Rename the table to Student_Info. d) Add a column PhoneNumber. e) Drop the table.

first normal form (1nf):
- remove repeating groups by separating each course into a different row for the same student.
- ensure each cell contains atomic (single) values.

second normal form (2nf):
- identify the composite primary key (student_id, courseid).
- move attributes that depend only on part of the primary key into separate tables.
- create:
  - student(student_id, name)
  - course(courseid, coursename, instructorname, instructorphone)
  - enrollment(student_id, courseid)

third normal form (3nf):
- remove transitive dependencies.
- instructorphone depends on instructorname, not directly on courseid.
- further split:
  - student(student_id, name)
  - course(courseid, coursename, instructorname)
  - instructor(instructorname, instructorphone)
  - enrollment(student_id, courseid)

q3. perform the following sql operations:
a) create a database:
create database student_db;
b) create a table named students:
create table students (
 student_id int primary key,
 name varchar(100),
dob date,

email varchar(100)
);
c) rename the table to student_info:
rename table students to student_info;
d) add a new column for phone number:
alter table student_info add phonenumber varchar(15);
e) drop the table:
drop table student_info;

q4. dml operations on student_info table
a) insert 3 student records into student_info:
insert into student_info (student_id, name, dob, email, phonenumber)
values
(1, 'sai', '2001-03-15', 'sai@gmail.com', '9876543210'),
(2, 'sunny', '1999-11-21', 'sunny@yahoo.com', '9123456780'),
(3, 'kumar', '2002-06-30', 'kumar@gmail.com', '9988776655');
b) update one student's phone number:
update student_info set phonenumber = '9999999999' where student_id = 1;
c) delete one student whose email ends with @gmail.com:
delete student_info where email like '%@gmail.com';
d) retrieve names and emails of students born after 2000:
select name, email from student_info where year(dob) > 2000;
e) retrieve distinct domain names from email column:
select distinct substring_index(email, '@', -1) as domain from student_info;

q5. advanced filtering with where
a) retrieve students with names starting with 'a':
select * from student_info where name like 'a%';
b) retrieve students with phone number between 9000000000 and 9999999999:
select * from student_info where phonenumber between 9000000000 and 9999999999;
c) retrieve students using in operator on city names (assuming city column):
select * from student_info where city in ('delhi', 'mumbai', 'chennai');

d) use and, or to filter students by age and email provider:
select * from student_info where year(curdate()) - year(dob) > 20 and email like '%@gmail.com';
e) use table and column aliasing to get names and dobs:
select s.name as studentname, s.dob as birthdate from student_info s;

q6. table marks and data analysis
create table and insert rows:
create table marks (student_id int, subject varchar(50), marks int);
insert into marks values
(1, 'maths', 85), (2, 'science', 75), (3, 'english', 65);
a) display student ids and their subjects where marks > 70:
select student_id, subject from marks where marks > 70;
b) display subjects with average marks:
select subject, avg(marks) as average_marks from marks group by subject;
c) filter subjects with average marks between 60 and 90:
select subject from marks group by subject having avg(marks) between 60 and 90;

q7. sql functions on student and marks data
a) get the current date formatted as 'yyyy-mm-dd':
select date_format(curdate(), '%y-%m-%d') as currentdate;
b) extract month and year from dob column:
select month(dob) as birthmonth, year(dob) as birthyear from student_info;
c) convert student's name to uppercase:
select upper(name) as uppername from student_info;
d) round off marks to 2 decimal places:
select round(marks, 2) as rounded_marks from marks;
e) use system function to return user or current database:
select user() as currentuser;

q8. aggregate and grouping operations

a) display total marks of each student:

```
select student_id, sum(marks) as totalmarks from marks group by student_id;
```

b) display subject-wise highest mark:

```
select subject, max(marks) as highest_marks from marks group by subject;
```

c) use group by and having to display subjects with average marks > 75:

```
select subject, avg(marks) as average_marks from marks group by subject having
avg(marks) > 75;
```

q9. sql joins

assume a courses table: courses(courseid, coursename, student_id)

a) inner join to retrieve students and their courses:

```
select s.name, c.coursename
from student_info s
inner join courses c on s.studentid = c.studentid;
```

b) left join to get all students even if not enrolled:

```
select s.name, c.coursename
from student_info s
left join courses c on s.studentid = c.studentid;
```

c) right join to get all courses even if no students:

```
select s.name, c.coursename
from student_info s
right join courses c on s.studentid = c.studentid;
```

d) full outer join equivalent using union:

```
select s.name, c.coursename
from student_info s
left join courses c on s.studentid = c.studentid
union
select s.name, c.coursename
from student_info s
right join courses c on s.studentid = c.studentid;
```

e) cross join to show all combinations:

```
select s.name, c.coursename
from student_info s
```

```
cross join courses c;
```

q10. subqueries

a) students who scored more than average in 'maths':

```
select student_id
from marks
where subject = 'maths'
and marks > (select avg(marks) from marks where subject = 'maths');
```

b) students not in the marks table:

```
select * from student_info
where student_id not in (select distinct student_id from marks);
```

c) use exists to get students with at least one subject:

```
select * from student_info s
where exists (select 1 from marks m where s.studentid = m.studentid);
```

d) use all to find those scoring more than all in 'science':

```
select student_id
from marks
where subject = 'science'
and marks > all (select marks from marks where subject = 'science');
```

e) use any for students scoring better than some in 'english':

```
select student_id
from marks
where subject = 'english'
and marks > any (select marks from marks where subject = 'english');
```

q11. set operations and correlated subqueries

assume another table: backup_students(studentid, name)

a) union of student names from two tables:

```
select name from student_info
union
select name from backup_students;
```

b) intersect to find common students (postgresql syntax):

```
select name from student_info
intersect
select name from backup_students;
```

c) except to list students in student_info but not in marks:

```
select student_id from student_info
except
select student_id from marks;
```

d) merge concept simulated using update + insert:

```
update student_info
set phonenumber = '1111111111'
where student_id = 1;
```

```
insert into student_info (student_id, name, dob, email, phonenumber)
select 5, 'new student', '2003-08-01', 'new@student.com', '9990000000'
where not exists (select 1 from student_info where student_id = 5);
```

e) correlated subquery to list students with above average per subject:

```
select distinct m1.studentid
from marks m1
where m1.marks > (
select avg(m2.marks) from marks m2 where m2.subject = m1.subject
);
```

PART -2

q1. when nosql is preferred over sql

nosql is preferred in the following scenarios
when handling large volumes of unstructured or semi-structured data
when scalability and high performance are essential (horizontal scaling)

types of nosql databases and real-time examples
document-based: used in content management systems like medium
key-value stores: used for caching in e-commerce platforms

q2. normalization of retail store data to bcnf

unnormalized table
customer(customerid, name, orders(orderid, productid, quantity, productname))

1nf
customer(customerid, name)
orders(orderid, customerid, productid, quantity, productname)

2nf
customer(customerid, name)
orders(orderid, customerid)
orderdetails(orderid, productid, quantity, productname)

3nf
customer(customerid, name)
orders(orderid, customerid)
products(productid, productname)
orderdetails(orderid, productid, quantity)

bcnf
already in bcnf after removing transitive dependencies and ensuring keys

q3. complex ddl and constraints

a) create a database retail_db and design schema for customers, orders, and products

```
create database retail_db;
use retail_db;

create table customers (
customerid int primary key,
name varchar(100),
email varchar(100)
);

create table products (
productid int primary key,
productname varchar(100),
category varchar(50),
price decimal(10,2)
);

create table orders (
orderid int primary key,
customerid int,
productid int,
quantity int,
orderdate date,
foreign key (customerid) references customers(customerid),
foreign key (productid) references products(productid)
);
```

b) implement a check constraint on quantity (> 0) in orders

```
alter table orders
add constraint chk_quantity check (quantity > 0);
```

c) alter the products table to add a 'discount' column and update some values

```
alter table products add discount decimal(5,2);
```

```
update products set discount = 10 where category = 'electronics';
update products set discount = 5 where category = 'books';
```

q4. dml operations

a) insert 3 sample orders per customer

```
insert into orders values (1, 101, 201, 2, curdate());
insert into orders values (2, 101, 202, 1, curdate());
insert into orders values (3, 101, 203, 3, curdate());
insert into orders values (4, 102, 204, 1, curdate());
insert into orders values (5, 102, 205, 2, curdate());
insert into orders values (6, 102, 206, 4, curdate());
```

b) update prices with 10 percent increase where quantity sold > 5

```
update products
set price = price * 1.10
where productid in (
select productid
from orders
group by productid
having sum(quantity) > 5
);
```

c) delete orders where the product has never been sold

```
delete from orders
where productid in (
select productid
from products
where productid not in (select distinct productid from orders)
);
```

q5. querying and filtering

a) customers who ordered more than 3 different products

```
select customerid
from orders
group by customerid
having count(distinct productid) > 3;
```

b) products not ordered by any customer

```
select *
from products
where productid not in (select distinct productid from orders);
```

c) count of orders placed by each customer in the last 30 days

```
select customerid, count(*) as ordercount
from orders
where orderdate >= curdate() - interval 30 day
group by customerid;
```

q6. string date and system functions

a) use string functions to standardize and extract parts from customer email ids

```
select lower(email) as loweremail,
substring_index(email, '@', -1) as domain
from customers;
```

b) use date functions to compute days between order date and today

```
select orderid, datediff(curdate(), orderdate) as days_since_order
from orders;
```

c) use system functions to return current user and host

```
select current_user() as user,
@@hostname as host;
```

d) use nested functions to format a customer greeting string

```
select concat('hello, ', upper(substring(name, 1, 1)), lower(substring(name, 2))) as
greeting
from customers;
```

q7. aggregation and rollup

a) aggregate total revenue by product category

```
select category, sum(price * quantity) as revenue
from products p
join orders o on p.productid = o.productid
group by category;
```

b) use group by with rollup to compute subtotal and grand total sales

```
select category, sum(price * quantity) as revenue
from products p
join orders o on p.productid = o.productid
group by category with rollup;
```

c) use having clause to filter categories with revenue > 100000

```
select category, sum(price * quantity) as revenue
from products p
join orders o on p.productid = o.productid
group by category
having revenue > 100000;
```

q8. joins and cross product

a) self join to list customers referred by other customers

```
select a.name as referrer, b.name as referred
from customers a
join customers b on a.customerid = b.referredby;
```

b) equi join across orders and products

```
select o.orderid, p.productname, o.quantity
from orders o
join products p on o.productid = p.productid;
```

c) join customers and orders to display top 3 spenders using window function

```
select * from (
select c.customerid, c.name, sum(o.quantity * p.price) as totalspent,
rank() over (order by sum(o.quantity * p.price) desc) as rank
from customers c
join orders o on c.customerid = o.customerid
join products p on o.productid = p.productid
group by c.customerid
) ranked_customers
where rank <= 3;
```

d) left outer join with where null to identify inactive customers

```
select c.customerid, c.name
from customers c
left join orders o on c.customerid = o.customerid
where o.orderid is null;
```

e) cross join for all product combinations in a bundle offer

```
select p1.productname as product1, p2.productname as product2
from products p1
cross join products p2
where p1.productid < p2.productid;
```

q9. subqueries

a) correlated subquery to get customers whose order amount exceeds their average

```
select distinct c.customerid, c.name
from customers c
join orders o on c.customerid = o.customerid
where (o.quantity * o.price) > (
select avg(quantity * price)
from orders
where customerid = c.customerid
);
```

b) subquery using exists to find customers with at least 2 different products

```
select * from customers c
where exists (
select 1 from orders o
where o.customerid = c.customerid
group by o.customerid
having count(distinct o.productid) >= 2
);
```

c) use all to find customers who ordered more than every other customer

```
select customerid
from orders
group by customerid
```

```
having count(orderid) > all (
select count(orderid)
from orders
group by customerid
);
```

d) use any to find products costlier than some in category 'electronics'

```
select * from products
where price > any (
select price from products where category = 'electronics'
);
```

e) nested subquery to list top 3 best-selling products

```
select * from (
select productid, sum(quantity) as totalsold
from orders
group by productid
order by totalsold desc
limit 3
) topproducts;
```

q10. set operations merge

a) simulate intersect using inner join on two customer segments

```
select a.customerid, a.name
from region1_customers a
inner join region2_customers b on a.customerid = b.customerid;
```

b) use except to find products in inventory not yet ordered

```
select productid, productname
from products
where productid not in (select distinct productid from orders);
```

c) simulate merge if customer exists update else insert

```
update customers
set email = 'updated@example.com'
where customerid = 10;
```

```
insert into customers (customerid, name, email)
 select 10, 'new customer', 'updated@example.com'
 where not exists (
 select 1 from customers where customerid = 10
 );
```

d) use union to combine two regional customer tables

```
 select * from region1_customers
 union
 select * from region2_customers;
```

e) write a with cte that ranks customers by total spend and filters top 5

```
 with customerspending as (
 select c.customerid, c.name, sum(o.quantity * p.price) as totalspend,
 rank() over (order by sum(o.quantity * p.price) desc) as rank
 from customers c
 join orders o on c.customerid = o.customerid
 join products p on o.productid = p.productid
 group by c.customerid
 )
 select * from customerspending where rank <= 5;
```