

Database Management System (313302)

Model Question Paper — Set B · Model Answers

About the SQL and PL/SQL in these answers: All code is written in **Oracle SQL / PL/SQL** (as used in the MSBTE syllabus and practicals). Every query and program was actually executed before printing — on PostgreSQL 16, with Oracle data types and PL/SQL syntax converted automatically — and the result tables and outputs shown are from those runs, displayed in SQL*Plus style (headings in capitals, Oracle feedback messages such as “1 row updated.”). Sample data is assumed wherever a question says “assume suitable data”.

Q.1

a) Data abstraction and instance. 2 M

i) Data abstraction: Hiding the complex details of how data is stored and maintained, and showing users only the required (relevant) data. DBMS provides three levels of abstraction — physical, logical (conceptual) and view (external) level.

ii) Instance: The collection of data (information) stored in the database at a particular moment of time is called an instance (or snapshot / state) of the database. It changes whenever data is inserted, deleted or updated.

(Each definition 1 M)

b) Any two Codd’s rules. 2 M

- Rule 8 — Physical data independence:** Application programs and user queries must remain unaffected when the physical storage or access methods (files, indexes) are changed.
- Rule 9 — Logical data independence:** Changes in the logical structure (e.g. adding a column or splitting a table) should not require changes in application programs.

(Other examples: Rule 1 Information rule, Rule 2 Guaranteed access, Rule 3 Systematic treatment of NULL.)

(Any two rules, 1 M each)

c) Normalization and normal forms. 2 M

Normalization: Normalization is the process of organizing data in a database by decomposing large relations into smaller well-structured relations, based on functional dependencies, so as to reduce data redundancy and remove insertion, update and deletion anomalies.

Normal forms: 1NF (First Normal Form), 2NF (Second), 3NF (Third), BCNF (Boyce-Codd Normal Form), 4NF and 5NF.

(Definition 1 M, list 1 M)

d) Four string functions. 2 M

- UPPER(str)** — converts the string to upper case.
- LOWER(str)** — converts the string to lower case.
- LENGTH(str)** — returns the number of characters.
- SUBSTR(str, m, n)** — returns n characters starting from position m.
- INITCAP(str)** — first letter of each word in capital.

Also: LTRIM, RTRIM, TRIM, LPAD, RPAD, CONCAT, REPLACE, INSTR.

```
SELECT UPPER('nashik'), LOWER('PUNE'), LENGTH('Database'), SUBSTR('Database',1,4),
INITCAP('ananda tech') FROM dual;
```

Output:

UPPER('NASHIK')	LOWER('PUNE')	LENGTH('DATABASE')	SUBSTR('DATABASE' ,1,4)	INITCAP('ANANDATECH')
NASHIK	pune	8	Data	Ananda Tech

(Any four, ½ M each)

e) Advantages of PL/SQL.

2 M

1. **Block structure:** Code is written in blocks; a whole block is sent to the server at once, which reduces network traffic and improves performance.
2. **Procedural capabilities:** Supports variables, conditions (IF), loops, along with SQL statements.
3. **Exception handling:** Run-time errors can be trapped and handled in the EXCEPTION section.
4. **Reusability:** Stored procedures, functions, packages and triggers are stored in the database and reused by many applications.

(Also: portability, tight integration with SQL, security through stored subprograms.)

(Any four, ½ M each)

f) Sequence — definition and syntax.

2 M

Sequence: A sequence is a database object that generates unique integer numbers in ascending or descending order automatically; it is commonly used to generate primary key values.

```
CREATE SEQUENCE sequence_name
[START WITH n]
[INCREMENT BY n]
[MINVALUE n | NOMINVALUE]
[MAXVALUE n | NOMAXVALUE]
[CYCLE | NOCYCLE]
[CACHE n | NOCACHE];
```

(Definition 1 M, syntax 1 M)

g) Causes of database failure.

2 M

1. **Transaction failure:** logical errors (bad input, data not found) or system errors such as deadlock force a transaction to abort.
2. **System crash:** power failure, hardware fault, or operating-system / DBMS software bug; contents of main memory (buffers) are lost.
3. **Disk (media) failure:** head crash or bad sectors destroy data stored on disk.
4. **Natural / physical disasters and human errors:** fire, flood, theft, sabotage, accidental deletion of data by a user.

(Any four, ½ M each)

Q.2

a) Network vs hierarchical data model.

4 M

Network data model	Hierarchical data model
Data is represented as records connected by links, forming a graph.	Data is represented as records arranged in a tree (parent–child).
A child (member) record can have many parent (owner) records.	A child record can have only one parent record.
Supports 1:1, 1:N and M:N relationships directly.	Supports 1:1 and 1:N only; M:N needs duplicate records.
Access to a record is possible through many paths.	Access is only from the root, through the parent path.
Less redundancy than hierarchical model.	More redundancy when one child is related to many parents.
Insertion / deletion is easier, but structure is complex.	Simple structure, but insertion of a child needs its parent; deleting a parent deletes its children.
Example: IDMS (CODASYL DBTG).	Example: IBM IMS.

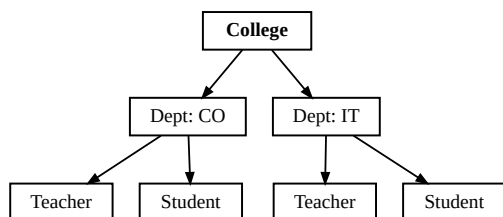


Fig. (a): Hierarchical model (tree: one parent per child)

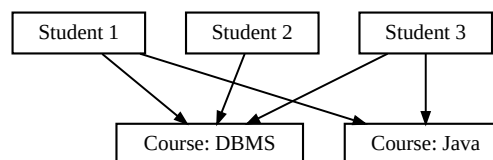


Fig. (b): Network model (graph: a child may have many parents)

(Any four points, 1 M each)

b) Composite, multivalued and derived attributes.

4 M

- Composite attribute:** An attribute that can be divided into smaller sub-parts, each having its own meaning. Example: *name* → first_name, middle_name, last_name; *address* → street, city, pin. Drawn as an ellipse connected to ellipses of its parts.
- Multivalued attribute:** An attribute that can have more than one value for the same entity. Example: a student may have many *mobile_no* values. Drawn as a **double ellipse**.
- Derived attribute:** An attribute whose value is calculated from other stored attributes and need not be stored. Example: *age* is derived from *dob* and today's date. Drawn as a **dashed ellipse**.

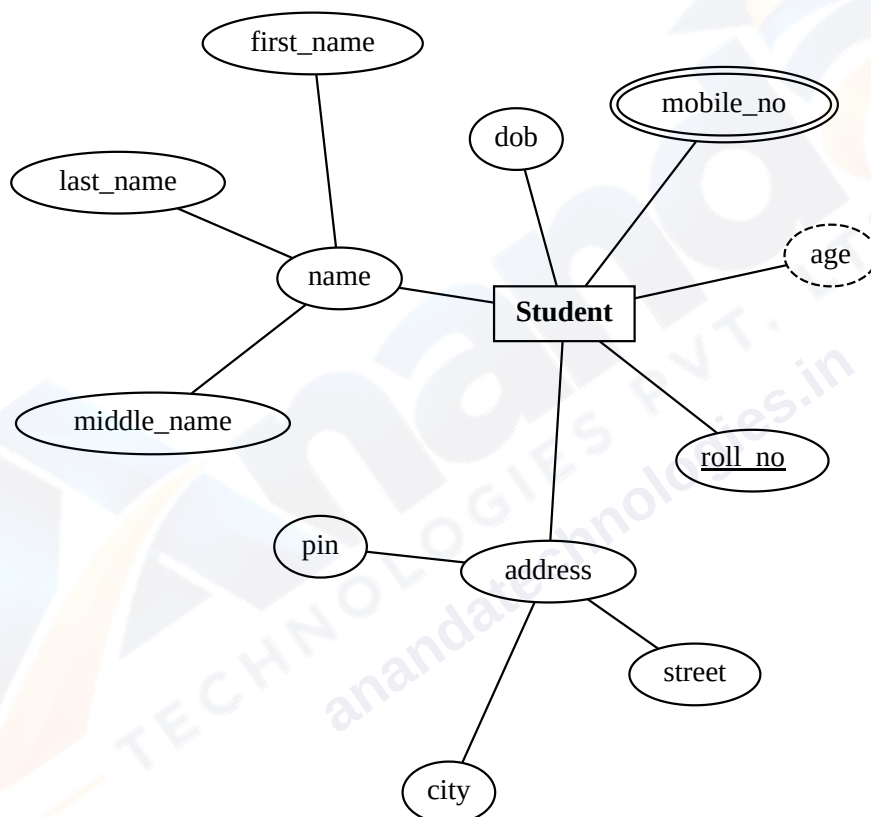


Fig.: Student entity showing composite, multivalued and derived attributes

(Each attribute type with example 1 M, diagram 1 M)

c) PL/SQL — largest of three numbers.

4 M

```

SET SERVEROUTPUT ON;
DECLARE
  a NUMBER := &a;
  b NUMBER := &b;
  c NUMBER := &c;
BEGIN
  IF a >= b AND a >= c THEN
    DBMS_OUTPUT.PUT_LINE('Largest number is ' || a);
  ELSIF b >= a AND b >= c THEN
    DBMS_OUTPUT.PUT_LINE('Largest number is ' || b);
  ELSE

```

```
DBMS_OUTPUT.PUT_LINE('Largest number is ' || c);
END IF;
END;
/
```

Output (run 1):

```
Enter value for a: 45
Enter value for b: 78
Enter value for c: 23
Largest number is 78
PL/SQL procedure successfully completed.
```

Output (run 2):

```
Enter value for a: 12
Enter value for b: 9
Enter value for c: 31
Largest number is 31
PL/SQL procedure successfully completed.
```

(Logic 3 M, syntax 1 M)

d) CREATE, ALTER, DROP and TRUNCATE.

4 M

1) CREATE — creates a new table (or other object).

Syntax: CREATE TABLE table_name (column1 datatype(size) [constraint], column2 ...);

2) ALTER — modifies the structure of an existing table: add a column, modify a column, drop a column, add a constraint.

Syntax: ALTER TABLE table_name ADD (column datatype); / MODIFY (column datatype); / DROP COLUMN column;

3) TRUNCATE — removes all rows from a table permanently but keeps the table structure (cannot be rolled back).

Syntax: TRUNCATE TABLE table_name;

4) DROP — removes the table structure along with all its data, indexes and constraints.

Syntax: DROP TABLE table_name;

Example:

```
CREATE TABLE course (cid NUMBER(3) PRIMARY KEY, cname VARCHAR2(20), fees NUMBER(6));
```

Table created.

```
ALTER TABLE course ADD (duration NUMBER(2));
```

Table altered.

```
ALTER TABLE course MODIFY (cname VARCHAR2(30));
```

Table altered.

```
INSERT INTO course VALUES (1, 'DBMS', 4000, 6);
```

1 row created.

```
TRUNCATE TABLE course;
```

Table truncated.

```
SELECT * FROM course;
```

Output:

no rows selected

```
DROP TABLE course;
```

Table dropped.

(Each command with syntax / example 1 M)

Q.3

a) Types of database users.

4 M

1. **Naive (end) users:** Unsophisticated users who interact with the database through ready-made application programs / forms, without knowing SQL. Example: bank clerk, ticket booking clerk, ATM user.
2. **Application programmers:** Computer professionals who write application programs (in Java, PHP, C#, etc. with embedded SQL) through which naive users access the database.
3. **Sophisticated users:** Users such as analysts, engineers and scientists who interact with the database directly by writing queries in SQL or using query / reporting tools, without writing programs.
4. **Specialized users:** Sophisticated users who write special database applications such as CAD systems, expert systems, and multimedia / GIS applications that store complex data types.
5. **Database Administrator (DBA):** The person having central control of the database. Functions: schema definition, storage structure and access-method definition, granting authorization (users and privileges), backup and recovery, performance monitoring, routine maintenance.

(Any four types, 1 M each)

b) Strong and weak entity set.

4 M

Strong entity set: An entity set that has a primary key of its own, so each entity can be identified independently. Drawn as a single rectangle; its key is underlined. Example: *Employee* (emp_id).

Weak entity set: An entity set that does not have sufficient attributes to form a primary key. It depends on a strong (owner / identifying) entity set and is identified by the owner's primary key + its own **partial key (discriminator)**. Drawn as a double rectangle, linked to the owner through a double diamond (identifying relationship), with total participation (double line). Example: *Dependent* (dep_name) of an employee.

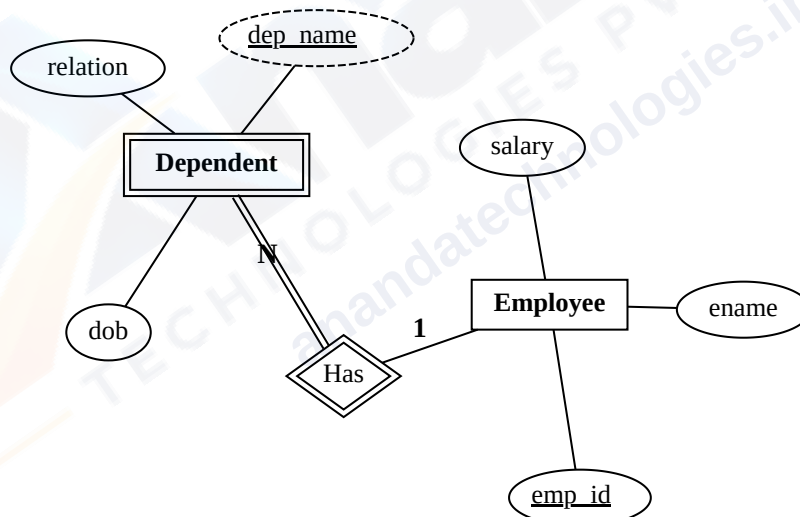


Fig.: Employee (strong) and Dependent (weak) entity sets

Primary key of Dependent = {emp_id, dep_name}. Partial key dep_name is shown with a dashed underline (here: underlined name in a dashed ellipse).

(Strong 1 M, weak 1 M, diagram 2 M)

c) Types of joins.

4 M

A **join** combines rows of two or more tables based on a related column.

Sample data (assumed) — table **EMP**: Sample data (assumed) — table **DEPT**:

EMPNO	ENAME	DEPTNO
1	Amit	10
2	Neha	20
3	Ravi	10
4	Sara	

DEPTNO	DNAME
10	Sales
20	HR
30	IT

1) Inner join (equi join) — only matching rows of both tables:

```
SELECT e.ename, d.dname FROM emp e INNER JOIN dept d ON e.deptno = d.deptno;
```

Output:

ENAME	DNAME
Amit	Sales
Neha	HR
Ravi	Sales

2) Left outer join — all rows of left table + matching rows of right (NULL if no match):

```
SELECT e.ename, d.dname FROM emp e LEFT OUTER JOIN dept d ON e.deptno = d.deptno;
```

Output:

ENAME	DNAME
Amit	Sales
Neha	HR
Ravi	Sales
Sara	

3) Right outer join — all rows of right table + matching rows of left:

```
SELECT e.ename, d.dname FROM emp e RIGHT OUTER JOIN dept d ON e.deptno = d.deptno;
```

Output:

ENAME	DNAME
Amit	Sales
Neha	HR
Ravi	Sales
	IT

4) Full outer join — all rows of both tables:

```
SELECT e.ename, d.dname FROM emp e FULL OUTER JOIN dept d ON e.deptno = d.deptno;
```

Output:

ENAME	DNAME
Amit	Sales
Neha	HR
Ravi	Sales
Sara	
	IT

Other joins: **cross join** (Cartesian product, every row with every row), **self join** (table joined with itself, e.g. employee and manager), **natural join** (equi join on same-named columns), **non-equij join** (condition other than =).

(Any four joins with example, 1 M each)

d) Trigger — salary not less than 5000.

4 M

Trigger: A trigger is a named PL/SQL block stored in the database that is executed (fired) **automatically** when a specified event — INSERT, UPDATE or DELETE on a table — occurs. It can be BEFORE / AFTER and row-level (FOR EACH ROW) or statement-level. Uses: enforcing business rules, auditing, maintaining derived data.

```
CREATE OR REPLACE TRIGGER trg_check_sal
BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW
BEGIN
    IF :NEW.sal < 5000 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Salary must not be less than 5000');
    END IF;
END;
/
INSERT INTO emp VALUES (2, 'Rita', 18000);
INSERT INTO emp VALUES (3, 'Mohan', 4500);
UPDATE emp SET sal = 3000 WHERE empno = 1;
SELECT * FROM emp;
```

Output (Emp initially has 1, Ajay, 12000):

```
Trigger created.
1 row created.
ORA-20001: Salary must not be less than 5000
ORA-20001: Salary must not be less than 5000

EMPNO  ENAME  SAL
-----
1      Ajay   12000
2      Rita   18000
```

:NEW refers to the new values of the row being inserted / updated. RAISE_APPLICATION_ERROR stops the statement, so salaries 4500 and 3000 are rejected.

(Definition 1 M, trigger 3 M)

Q.4

a) Instance vs schema; data independence.

4 M

Schema	Instance
Overall logical design / structure of the database.	Actual data stored in the database at a particular moment.
Defined using DDL, e.g. Student(rollno, name, marks).	Rows present at a time, e.g. (1, Asha, 78), (2, Ravi, 81).
Changes rarely.	Changes frequently with every insert, update, delete.
Like a variable declaration (type) in a program.	Like the value of the variable at a given time.

Data independence: The ability to change the schema at one level without having to change the schema at the next higher level.

- Physical data independence:** Ability to change the physical (internal) schema — storage devices, file organization, indexes — without changing the conceptual schema or application programs. Example: adding an index on *rollno* does not change any query.
- Logical data independence:** Ability to change the conceptual (logical) schema — adding / removing attributes or tables — without changing external views or application programs. Example: adding a column *email* to Student does not affect a view that shows only *rollno* and *name*. It is harder to achieve than physical independence.

(Difference 2 M, data independence 2 M)

b) ACID properties.

4 M

A transaction must satisfy four properties (ACID) to keep the database correct. Example transaction T: transfer ₹1000 from account A to account B.

1. **Atomicity (all or nothing):** Either all operations of T are reflected in the database or none. If T fails after debiting A, the debit is undone (ROLLBACK).
2. **Consistency:** Execution of T alone keeps the database consistent — the sum of balances of A and B remains the same before and after T (integrity constraints remain satisfied).
3. **Isolation:** Even if many transactions run concurrently, each transaction is unaware of the others; the result is the same as if they were executed one after another (serially).
4. **Durability:** After T commits successfully, its changes persist in the database even if a system failure occurs later.

Demonstration of atomicity using TCL:

```
UPDATE account SET balance = balance - 1000 WHERE acc_no = 1001;
```

1 row updated.

```
ROLLBACK;
```

Rollback complete.

```
SELECT * FROM account;
```

Output:

ACC_NO	NAME	BALANCE
1001	A	5000
1002	B	3000

```
UPDATE account SET balance = balance - 1000 WHERE acc_no = 1001;
```

1 row updated.

```
UPDATE account SET balance = balance + 1000 WHERE acc_no = 1002;
```

1 row updated.

```
COMMIT;
```

Commit complete.

```
SELECT * FROM account ORDER BY acc_no;
```

Output:

ACC_NO	NAME	BALANCE
1001	A	4000
1002	B	4000

The first debit was rolled back (balance stays 5000); the complete transfer was committed together (4000 + 4000 = 8000, consistent).

(Each property 1 M)

c) Set operators.

4 M

Set operators combine the results of two SELECT queries. Both queries must have the same number of columns with compatible data types.

Sample data (assumed) — table **SPORTS**: Sample data (assumed) — table **CULTURAL**:

NAME
Amit
Neha
Ravi

NAME
Neha
Pooja
Ravi

1) UNION — all distinct rows from both queries (duplicates removed):

```
SELECT name FROM sports UNION SELECT name FROM cultural;
```

Output:

NAME
Ravi
Neha
Pooja
Amit

2) UNION ALL — all rows including duplicates:

```
SELECT name FROM sports UNION ALL SELECT name FROM cultural;
```

Output:

NAME
Amit
Neha
Ravi
Neha
Pooja
Ravi

3) INTERSECT — only rows common to both queries:

```
SELECT name FROM sports INTERSECT SELECT name FROM cultural;
```

Output:

NAME
Neha
Ravi

4) MINUS — rows of first query that are not in the second:

```
SELECT name FROM sports MINUS SELECT name FROM cultural;
```

Output:

NAME
Amit

(Each operator with example 1 M)

d) View Emp_D10.

4 M

Sample data (assumed) — table **EMPLOYEE**:

EID	ENAME	JOB	SALARY	DEPTNO
1	Sunil	Manager	62000	10
2	Asha	Clerk	28000	10
3	Rakesh	Analyst	45000	20
4	Meera	Analyst	41000	10
5	Tushar	Clerk	26000	30

i) Create the view:

```
CREATE VIEW emp_d10 AS
  SELECT eid, ename, salary FROM employee WHERE deptno = 10;
```

View created.

```
SELECT * FROM emp_d10;
```

Output:

EID	ENAME	SALARY
1	Sunil	62000
2	Asha	28000
4	Meera	41000

ii) Employees of the view with salary > 40000:

```
SELECT * FROM emp_d10 WHERE salary > 40000;
```

Output:

EID	ENAME	SALARY
1	Sunil	62000
4	Meera	41000

iii) Remove the view:

```
DROP VIEW emp_d10;
```

View dropped.

(i) 2 M, ii) 1 M, iii) 1 M)

e) Explicit cursor — names and salaries.

4 M

```
SET SERVEROUTPUT ON;
DECLARE
    CURSOR c_emp IS SELECT ename, salary FROM employee;
    v_name employee.ename%TYPE;
    v_sal employee.salary%TYPE;
BEGIN
    OPEN c_emp;
    LOOP
        FETCH c_emp INTO v_name, v_sal;
        EXIT WHEN c_emp%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(RPAD(v_name, 10) || v_sal);
    END LOOP;
    CLOSE c_emp;
END;
/
```

Output (Employee table as in Q.4 d):

```
Sunil      62000
Asha       28000
Rakesh     45000
Meera      41000
Tushar     26000
PL/SQL procedure successfully completed.
```

The cursor is declared, opened, each row is fetched in a loop until %NOTFOUND becomes TRUE, and then the cursor is closed.

(Declare 1 M, open/fetch loop 2 M, close 1 M)

Q.5

a) E-R diagram — Hospital Management System.

6 M

Assumptions: A doctor treats many patients and a patient may be treated by many doctors (M : N). Many patients are admitted in one ward (N : 1). A bill cannot exist without a patient, so Bill is a weak entity identified by patient_id + bill_no.

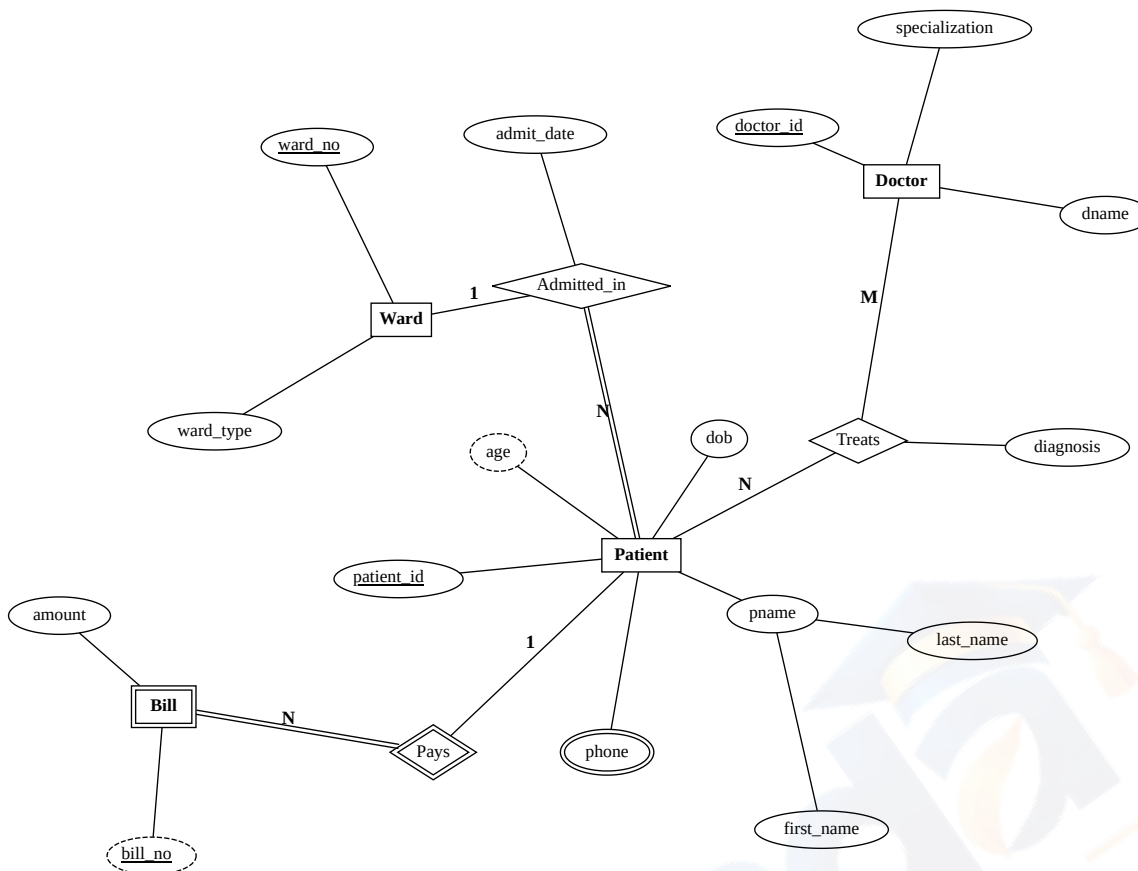


Fig.: E-R diagram for Hospital Management System

Rectangles: entities; double rectangle: weak entity (Bill); double diamond: identifying relationship (Pays); underlined: keys; pname: composite; phone: multivalued; age: derived; 1 / M / N: cardinality; double lines: total participation.

(Entities 2 M, relationships and cardinality 2 M, attributes 2 M)

b) SQL queries on Employee table.

6 M

Sample data (assumed) — table **EMPLOYEE**:

EMPID	ENAME	SALARY	DESIGNATION	CITY
201	Sachin	30000	Clerk	Pune
202	Mansi	52000	Manager	Thane
203	Sagar	41000	Analyst	Mumbai
204	Ketan	22000	Clerk	Thane
205	Leena	27500	Clerk	Nashik

i) Increase salary of Clerks by 10%:

```
UPDATE employee SET salary = salary + salary * 0.10 WHERE designation = 'Clerk';
```

3 rows updated.

ii) Delete employees staying in Thane:

```
DELETE FROM employee WHERE city = 'Thane';
```

2 rows deleted.

iii) Salary between 25000 and 45000:

```
SELECT empid, ename FROM employee WHERE salary BETWEEN 25000 AND 45000 ORDER BY empid;
```

Output:

EMPID	ENAME
201	Sachin
203	Sagar
205	Leena

iv) Insert one record:

```
INSERT INTO employee VALUES (206, 'Rohit', 35000, 'Analyst', 'Nagpur');
```

1 row created.

v) Add column email:

```
ALTER TABLE employee ADD (email VARCHAR2(30));
```

Table altered.

vi) Names starting with S:

```
SELECT ename FROM employee WHERE ename LIKE 'S%' ORDER BY ename;
```

Output:

ENAME
Sachin
Sagar

(Queries are run one after another on the same table, so iii) uses the salaries updated in i.)

(Each query 1 M)

c) PL/SQL — withdrawal with user-defined exception.

6 M

Table: Account (acc_no, name, balance) with rows (1001, Ravi, 5000) and (1002, Sima, 3000).

```
SET SERVEROUTPUT ON;
DECLARE
    v_acc account.acc_no%TYPE := &v_acc;
    v_amt NUMBER := &v_amt;
    v_bal account.balance%TYPE;
    insufficient_balance EXCEPTION;
BEGIN
    SELECT balance INTO v_bal FROM account WHERE acc_no = v_acc;
    IF v_bal < v_amt THEN
        RAISE insufficient_balance;
    END IF;
    UPDATE account SET balance = balance - v_amt WHERE acc_no = v_acc;
    DBMS_OUTPUT.PUT_LINE('Withdrawal successful. New balance = ' || (v_bal - v_amt));
EXCEPTION
    WHEN insufficient_balance THEN
        DBMS_OUTPUT.PUT_LINE('Insufficient balance! Available balance = ' || v_bal);
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Invalid account number');
END;
/
SELECT * FROM account ORDER BY acc_no;
```

Output (run 1):

```
Enter value for v_acc: 1001
Enter value for v_amt: 2000
Withdrawal successful. New balance = 3000
PL/SQL procedure successfully completed.
```

ACC_NO	NAME	BALANCE
1001	Ravi	3000
1002	Sima	3000

Output (run 2):

```
Enter value for v_acc: 1002
Enter value for v_amt: 9000
Insufficient balance! Available balance = 3000
PL/SQL procedure successfully completed.
```

ACC_NO	NAME	BALANCE
1001	Ravi	5000
1002	Sima	3000

Output (run 3):

```
Enter value for v_acc: 1009
Enter value for v_amt: 500
Invalid account number
PL/SQL procedure successfully completed.
```

ACC_NO	NAME	BALANCE
1001	Ravi	5000
1002	Sima	3000

The user-defined exception is declared in DECLARE, raised with RAISE when the balance is not enough, and handled in the EXCEPTION section; the predefined NO_DATA_FOUND handles a wrong account number.

(Declaration 1 M, logic 3 M, exception handling 2 M)

Q.6

a) Parent–child relationship and referential integrity.

6 M

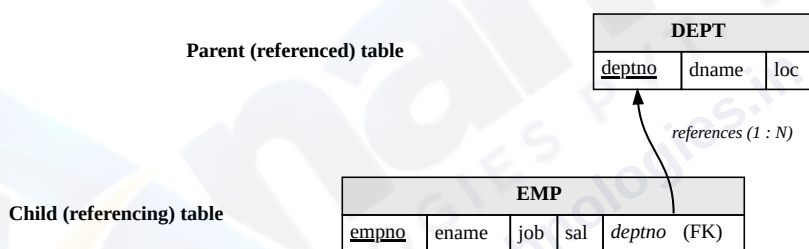


Fig.: DEPT (parent) — EMP (child) relationship through deptno

Explanation: One department has many employees, so DEPT and EMP have a one-to-many (1 : N) relationship. DEPT is the **parent (master / referenced)** table — its primary key *deptno* is referenced. EMP is the **child (detail / referencing)** table — its column *deptno* is a foreign key.

Referential integrity constraint: every value of the foreign key EMP.deptno must either match an existing DEPT.deptno or be NULL. Hence (1) a child row with a non-existing deptno cannot be inserted, and (2) a parent row cannot be deleted while child rows refer to it (unless ON DELETE CASCADE is given).

```
CREATE TABLE dept (
  deptno NUMBER(2) CONSTRAINT pk_dept PRIMARY KEY,
  dname VARCHAR2(14),
  loc VARCHAR2(13) );
```

Table created.

```
CREATE TABLE emp (
  empno NUMBER(4) CONSTRAINT pk_emp PRIMARY KEY,
  ename VARCHAR2(10),
  job VARCHAR2(9),
  sal NUMBER(7,2),
  deptno NUMBER(2) CONSTRAINT fk_deptno REFERENCES dept(deptno) );
```

Table created.

Sample data:

Parent — table **DEPT**:

DEPTNO	DNAME	LOC
10	ACCOUNTING	MUMBAI
20	RESEARCH	PUNE

Child — table **EMP**:

EMPNO	ENAME	JOB	SAL	DEPTNO
7001	KIRAN	CLERK	18000	10
7002	MEGHA	ANALYST	42000	20
7003	JAYESH	MANAGER	56000	10

Checking the constraint:

```
INSERT INTO emp VALUES (7004, 'NILESH', 'CLERK', 15000, 40);
```

ERROR at line 1:

ORA-02291: integrity constraint (FK_DEPTNO) violated - parent key not found

```
DELETE FROM dept WHERE deptno = 10;
```

ERROR at line 1:

ORA-02292: integrity constraint (FK_DEPTNO) violated - child record found

```
DELETE FROM dept WHERE deptno = 30;
```

0 rows deleted.

Deleting department 30 deleted 0 rows (no such department). With **REFERENCES dept(deptno) ON DELETE CASCADE**, deleting a department would also delete its employees.

(Diagram 2 M, explanation 2 M, constraint 2 M)

b) Users, privileges and roles.

6 M

i)

```
CREATE USER sales_mgr IDENTIFIED BY sm2026;
```

User created.

ii)

```
GRANT CREATE SESSION, CREATE TABLE TO sales_mgr;
```

Grant succeeded.

iii)

```
GRANT SELECT, INSERT, UPDATE ON orders TO sales_mgr;
```

Grant succeeded.

iv)

```
CREATE ROLE report_role;
```

Role created.

```
GRANT SELECT ON orders TO report_role;
```

Grant succeeded.

v)

```
GRANT report_role TO sales_mgr;
```

Grant succeeded.

vi)

```
REVOKE UPDATE ON orders FROM sales_mgr;
```

Revoke succeeded.

A **role** is a named group of privileges; granting the role gives all its privileges to the user at once, which makes privilege management easy.

(Each command 1 M)

c) Procedure and function.

6 M

Table Emp (empno, ename, sal, deptno) with rows (1, Kiran, 20000, 10), (2, Megha, 42000, 20), (3, Jayesh, 36000, 10).

i) Procedure:

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE PROCEDURE raise_sal(p_no IN NUMBER, p_pct IN NUMBER) IS
BEGIN
    UPDATE emp SET sal = sal + sal * p_pct / 100 WHERE empno = p_no;
    DBMS_OUTPUT.PUT_LINE('Salary of employee ' || p_no || ' increased by ' || p_pct || '%');
END raise_sal;
/
EXEC raise_sal(1, 15);
SELECT * FROM emp WHERE empno = 1;
```

Output:

Procedure created.
Salary of employee 1 increased by 15%
PL/SQL procedure successfully completed.

EMPNO	ENAME	SAL	DEPTNO
1	Kiran	23000	10

ii) Function:

```
CREATE OR REPLACE FUNCTION dept_total(p_dno IN NUMBER) RETURN NUMBER IS
    v_total NUMBER;
BEGIN
    SELECT SUM(sal) INTO v_total FROM emp WHERE deptno = p_dno;
    RETURN v_total;
END dept_total;
/
DECLARE
    t NUMBER;
BEGIN
    t := dept_total(10);
    DBMS_OUTPUT.PUT_LINE('Total salary of dept 10 = ' || t);
END;
/
```

Output:

Function created.
Total salary of dept 10 = 56000
PL/SQL procedure successfully completed.

A procedure is called as a statement (EXEC / from a block) and does not return a value through RETURN; a function returns one value and is called within an expression.

(Procedure 3 M, function 3 M)

Note: This is a **model question paper prepared by Ananda Technologies** only for practice and exam preparation, based on the pattern of previous MSBTE question papers and the K-scheme syllabus. It is **not an official MSBTE paper**, and it does **not** mean that these questions will come in the exam. Answers follow standard textbook points; marks shown are indicative. Free for students — anandatechnologies.in