# Advance Database Management (315324)

Model Question Paper — Set C · Model Answers

**About the code in these answers:** every query and command was actually executed before printing, and the outputs shown are copied from those runs. **SQL** — PostgreSQL 16 (object-relational features: composite/structured types, table inheritance, arrays); where the textbook SQL:1999 syntax (e.g. `UNDER`, `MULTISET`) is not accepted by PostgreSQL, the standard syntax to write in the exam is given and the PostgreSQL equivalent is shown with its real output. **MongoDB** — shell commands were run on an in-memory MongoDB-compatible engine (mongomock 4.3) and printed in mongosh style; ObjectId values will be different on your computer. Plain JavaScript expressions (as typed in mongosh) were run on Node.js (the same V8 JavaScript engine used by mongosh), time zone IST. **XML** — XQuery / XPath on Saxon-HE 13; XML Schema / DTD validation with libxml2 (lxml). Commands are shown in **bold**, results below them. Sample data is assumed wherever needed.

## Q.1

**a) Two advantages of concurrency control.** 2 M

1. **Maintains consistency and isolation:** prevents problems of concurrent execution such as lost update, dirty read and unrepeatable (inconsistent) read; the result is serializable.
2. **Better performance:** many transactions run at the same time — higher throughput and resource (CPU, disk) utilisation, and reduced waiting / response time.

*(Any two, 1 M each)*

**b) Two phases of 2PL.** 2 M

1. **Growing phase:** a transaction may obtain (acquire) locks but may not release any lock.
2. **Shrinking phase:** a transaction may release locks but may not obtain any new lock.

*(Each phase 1 M)*

**c) Centralized database system; two types.** 2 M

**Centralized database system:** a database system that runs on a single computer system and does not interact with other computer systems; all data and DBMS processing are at one place.

1. **Single-user system** — desktop / personal computer used by one person at a time (e.g. MS Access on a PC).
2. **Multi-user system** — a server with many CPUs/terminals serving many users concurrently.

*(Definition 1 M, types 1 M)*

**d) Syntax — multiset type attribute.** 2 M

```
attribute_name  element_type MULTISET

-- example
CREATE TYPE Book AS (
   title       VARCHAR(20),
   keyword_set VARCHAR(20) MULTISET );

INSERT INTO books VALUES ('DBMS', MULTISET['SQL', 'ER model', 'SQL']);
```

A multiset is an unordered collection that may contain duplicates.

*(Syntax 1 M, example 1 M)*

**e) Document database.** 2 M

**Document database:** a NoSQL database that stores data as self-describing **documents** (JSON / BSON / XML) grouped into collections; each document has a unique key and may have a different structure. Example: MongoDB, CouchDB.

```
{ _id: 1, name: "Asha", branch: "IF",
  marks: [78, 85], address: { city: "Pune" } }
```

*(Definition 1 M, example 1 M)*

**f) Data mining; two techniques.** 2 M

**Data mining:** the process of semi-automatically analysing large databases to discover useful patterns, trends and rules (knowledge discovery in databases).

**Techniques:** classification, clustering, association rules, regression (any two).

*(Definition 1 M, two techniques 1 M)*

**g) XML vs HTML.** 2 M

| XML | HTML |
|---|---|
| Designed to store and transport data. | Designed to display data in a browser. |
| User-defined tags. | Predefined tags (<p>, <table>). |
| Case-sensitive; every tag must be closed and properly nested. | Not strict; some tags need not be closed. |

*(Any two, 1 M each)*

## Q.2

**a) Two-tier and three-tier client-server models.** 4 M

In a **client-server** system the database functionality is divided into a **front end** (client: user interface, forms, report writers) and a **back end** (server: query evaluation, optimisation, concurrency control, recovery).

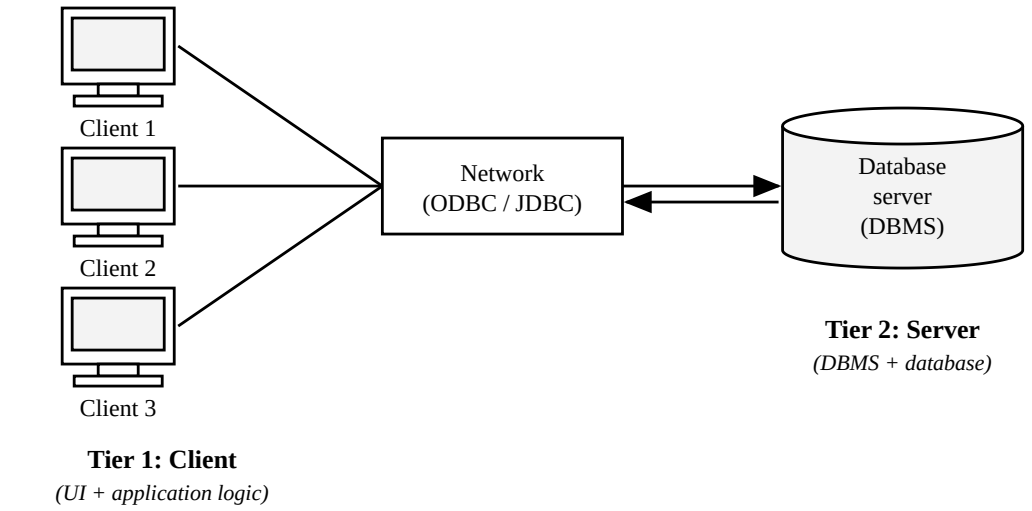


*Fig. (a): Two-tier client-server architecture*

- **Two-tier:** the application runs on the client machine and directly calls the database server through ODBC / JDBC.
- Simple and fast for small numbers of users (e.g. a desktop billing program connected to a MySQL server); but business logic is on every client, so maintenance and security are harder and it does not scale to many users.

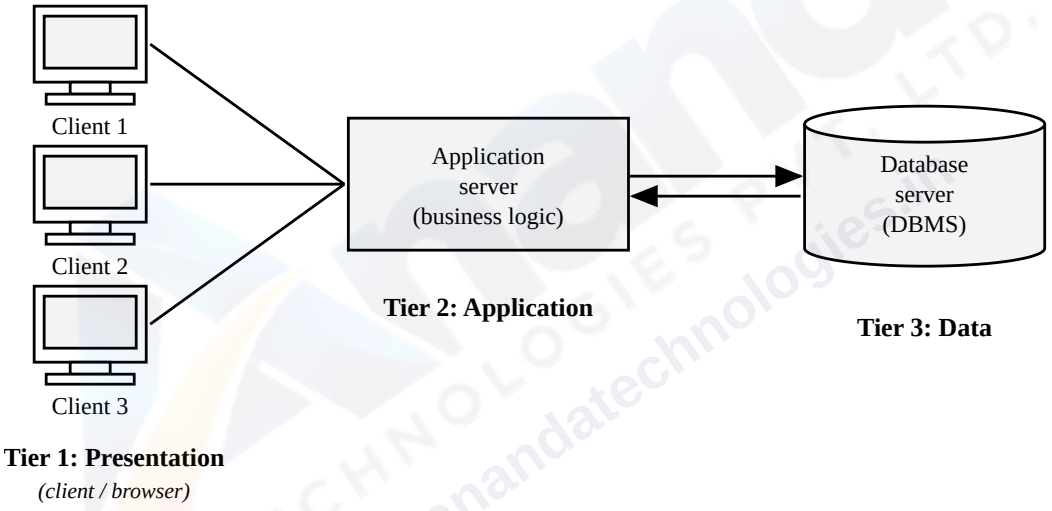


*Fig. (b): Three-tier client-server architecture*

- **Three-tier:** the client is only a front end (often a web browser); it communicates with an **application server** which contains the business logic and in turn talks to the database server.
- More secure and scalable, easier to maintain (logic changed at one place) — used by web applications and large systems (e.g. online railway reservation).

*(Each model: diagram 1 M, explanation 1 M)*

**b) Four benefits of distributed database system.** 4 M

1. **Sharing of data:** users at one site can access data stored at other sites (e.g. a branch can see head-office data).
2. **Local autonomy:** each site controls its own data and local users.

3. **Reliability and availability:** if one site fails, other sites keep working; with replication, data is still available.
4. **Faster response:** most queries use local data, so they are answered quickly with less network traffic.
5. **Modular (incremental) growth:** a new site can be added without disturbing other sites.
6. **Lower communication cost and parallel processing:** data is kept near its users; a big query can be split among sites.

*(Any four benefits, 1 M each)*

**c) Complex data types in object-based databases.** 4 M

Traditional relational databases require atomic attribute values (first normal form). Object-based databases allow **complex data types** so that real-world objects can be represented directly:

1. **Structured (composite) types:** an attribute made of components, e.g. name = (first_name, last_name), address = (street, city, pin).
2. **Collection types:** multivalued attributes stored as **arrays** (ordered) or **multisets** (unordered, duplicates allowed), e.g. phone numbers, keywords.
3. **Reference types:** an attribute that refers to (points to) an object of another table — `ref(Person)`.
4. **Nested relations / user-defined types with methods** — relations within relations; objects with behaviour (non-1NF model).

**Example** (executed on PostgreSQL):

```
adm_mp=# CREATE TYPE name_t AS (first_name VARCHAR(15), last_name VARCHAR(15));
CREATE TYPE
adm_mp=# CREATE TYPE address_t AS (street VARCHAR(20), city VARCHAR(15));
CREATE TYPE
adm_mp=# CREATE TABLE customer (cid INT, name name_t, address address_t, phones
VARCHAR(10)[]);
CREATE TABLE
adm_mp=# INSERT INTO customer VALUES (1, ROW('Vikas', 'More'), ROW('Station
Road', 'Latur'), ARRAY['9850012345', '9850067890']);
INSERT 0 1
adm_mp=# SELECT cid, (name).first_name, (address).city, cardinality(phones) AS
no_of_phones FROM customer;
 cid | first_name | city  | no_of_phones
-----+------------+-------+--------------
   1 | Vikas      | Latur |            2
(1 row)
```

*(Explanation 2 M, example 2 M)*

**d) SQL vs NoSQL.** 4 M

| Point | SQL database | NoSQL database |
|---|---|---|
| **Schema** | Fixed, predefined schema; must be altered to add columns. | Dynamic / flexible schema; each document can have different fields. |
| **Query language** | Standard SQL (declarative). | No single standard; database-specific APIs, e.g. MongoDB find(). |
| **Joins** | Complex joins are easy. | Joins are limited; data is usually denormalised / embedded. |
| **Best suited for** | Structured data and complex transactions (banking, ERP). | Big data, semi-structured / unstructured data, real-time web apps. |

*(Any four points, 1 M each)*

## Q.3

**a) Types of concurrency control protocols.** 4 M

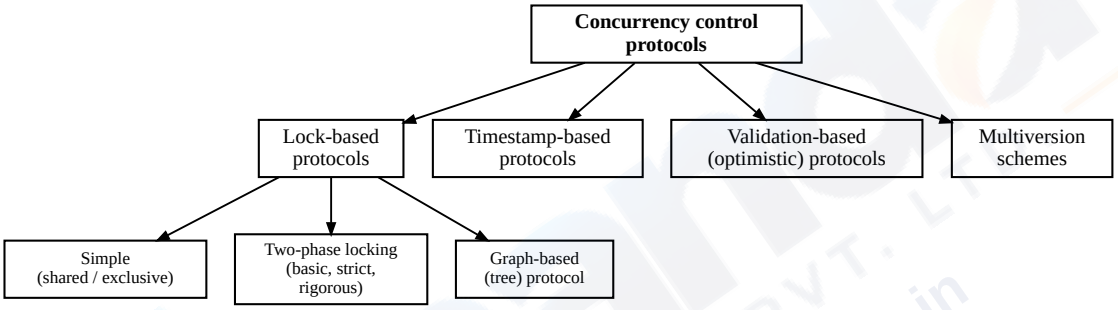


*Fig.: Classification of concurrency control protocols*

1. **Lock-based protocols:** a transaction must obtain a lock (shared / exclusive) on a data item before using it. Example: **Two-Phase Locking** (basic, strict, rigorous) and graph-based (tree) protocols.
2. **Timestamp-based protocols:** each transaction gets a unique timestamp TS(T) when it starts; conflicting operations are executed in timestamp order. Each item keeps W-timestamp and R-timestamp; an operation that arrives too late causes the transaction to roll back. Deadlock-free.
3. **Validation-based (optimistic) protocols:** transactions execute in three phases — read, validation, write. Updates are made on local copies; at commit time validation checks for conflicts. Good when conflicts are rare.
4. **Multiversion schemes:** each write creates a new version of the item; a read gets a suitable old version, so reads never wait (multiversion timestamp ordering, multiversion 2PL, snapshot isolation).

*(Any four types, 1 M each)*

**b) Throughput and response time; numerical.** 4 M

1. **Throughput:** number of tasks completed in a given time. Increased by processing many small tasks in parallel (inter-query parallelism) — important for OLTP systems.

2. **Response time:** time to complete a single task from its submission. Reduced by dividing a large task into sub-tasks executed in parallel (intra-query parallelism) — important for decision-support queries.
3. Performance of parallel systems is judged by **speedup** (TS/TL for the same task on N-times larger system) and **scaleup** (N-times larger task on N-times larger system).

**Numerical:**

Throughput = transactions / time = 900 / (3 × 60 s) = 900 / 180 = **5 transactions per second**. Speedup = TS / TL = 12 / 4 = **3**; with N = 4 processors, speedup < N, so it is **sublinear** (efficiency 75%). The response time of the query has reduced from 12 s to 4 s.

*(Definitions 2 M, calculations 2 M)*

### c) XQuery FLWOR with sorting and function. 4 M

*XML document **result.xml**:*

```
<result>
  <student roll="1"><name>Asha</name><branch>IF</branch><marks>82</marks>
</student>
  <student roll="2"><name>Rohan</name><branch>AN</branch><marks>71</marks>
</student>
  <student roll="3"><name>Sneha</name><branch>IF</branch><marks>91</marks>
</student>
  <student roll="4"><name>Imran</name><branch>IF</branch><marks>64</marks>
</student>
  <student roll="5"><name>Kavya</name><branch>AN</branch><marks>88</marks>
</student>
</result>
```

*XQuery:*

```
let $s := doc("result.xml")/result/student[branch = "IF"]
return
<if_students average="{ avg($s/marks) }">{
  for $x in $s
  order by xs:integer($x/marks) descending
  return <student marks="{$x/marks}">{ $x/name/text() }</student>
}</if_students>
```

*Output:*

```
<if_students average="79">
   <student marks="91">Sneha</student>
   <student marks="82">Asha</student>
   <student marks="64">Imran</student>
</if_students>
```

`let` binds all IF students once; `for … order by … descending` sorts them on marks; `avg()` is a built-in XQuery function ((82 + 91 + 64) / 3 = 79). `xs:integer()` converts text to a number so that sorting is numeric.

*(FLWOR with order by 3 M, average 1 M)*

### d) Virtual warehouse; functions of warehouse tools. 4 M

**Virtual warehouse:** a set of **views** defined over the operational databases. Data is not copied into a separate store; when a query is asked, it is answered from the source databases through the views.

It is easy to build, but needs spare capacity on the operational servers and gives slower answers; only some summary views may be materialised.

**Functions of data-warehouse (back-end) tools:**

1. **Data extraction:** gathers data from multiple, heterogeneous and external sources.
2. **Data cleaning:** detects and corrects errors (wrong spellings, missing values, duplicates — e.g. fuzzy lookup, merge-purge).
3. **Data transformation:** converts data from source format to warehouse format (units, codes, schema integration).
4. **Load:** sorts, summarises, consolidates, computes views, builds indexes and partitions.
5. **Refresh:** propagates updates from the sources to the warehouse periodically.

*(Virtual warehouse 1 M, tool functions 3 M)*

## Q.4

**a) Transaction processing in a parallel database system.** 4 M

1. **Shared-memory and shared-disk systems:** transactions are processed almost as in a centralized system — a common lock table and log are used. In shared-disk systems the locking must also keep processor caches coherent: before reading a page a node gets a lock and reads the latest copy from disk; before releasing an X-lock the page is flushed (cache-coherency protocol).
2. **Shared-nothing systems:** data is **partitioned** across nodes (round-robin, hash, range partitioning). A transaction is executed at the node(s) holding its data; each node has its own lock manager and log. A transaction that updates data at many nodes is a **distributed transaction** and needs an atomic commit protocol such as **two-phase commit**.
3. **Inter-query parallelism:** different transactions run in parallel on different processors → higher throughput (main use in transaction processing).
4. **Intra-query parallelism:** one large query is divided into parts executed in parallel (intra-operation / inter-operation) → lower response time.
5. Concurrency control (2PL) and recovery (logging) must be coordinated across processors; deadlocks may occur across nodes and are detected globally.

*(Any four points, 1 M each)*

**b) Table inheritance; consistency requirements.** 4 M

Table inheritance lets a **subtable** be declared `UNDER` a **supertable**; the subtable inherits all attributes of the supertable, and every row of the subtable is also visible in the supertable.

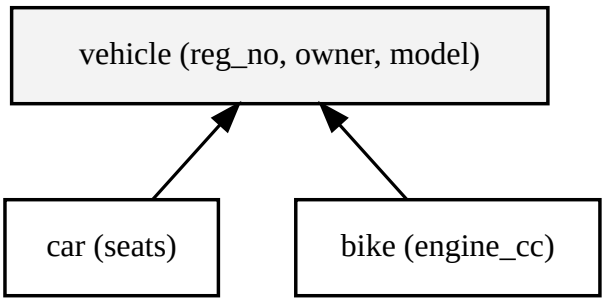


*Fig.: Supertable vehicle with subtables car and bike*

```
-- SQL:1999
CREATE TABLE vehicle OF Vehicle;
CREATE TABLE car  OF Car  UNDER vehicle;
CREATE TABLE bike OF Bike UNDER vehicle;
```

**Executed (PostgreSQL):**

```
adm_mp=# CREATE TABLE vehicle (reg_no VARCHAR(12), owner VARCHAR(15), model
VARCHAR(15));
CREATE TABLE
adm_mp=# CREATE TABLE car (seats INT) INHERITS (vehicle);
CREATE TABLE
adm_mp=# CREATE TABLE bike (engine_cc INT) INHERITS (vehicle);
CREATE TABLE
adm_mp=# INSERT INTO car VALUES ('MH26-AB-1111', 'Patil', 'Swift', 5);
INSERT 0 1
adm_mp=# INSERT INTO bike VALUES ('MH24-C-2222', 'Shaikh', 'Pulsar', 150);
INSERT 0 1
adm_mp=# SELECT reg_no, owner, model FROM vehicle;
    reg_no    | owner  | model
--------------+--------+--------
 MH26-AB-1111 | Patil  | Swift
 MH24-C-2222  | Shaikh | Pulsar
(2 rows)
```

**Consistency requirements on subtables:**

1. Each tuple of the supertable can correspond to **at most one tuple in each** of its immediate subtables.
2. All tuples corresponding to each other (in supertable and subtables) must be derived from **one tuple** inserted into one table — so a tuple cannot be in two sibling subtables (a vehicle cannot be both car and bike) unless a common subtable exists.

*(Explanation and example 2 M, consistency requirements 2 M)*

### c) Arrays and embedded documents in MongoDB. 4 M

1. **Array:** an ordered list of values stored in one field; values may be of any type, even documents. Queries: a scalar value matches any element; `$all` — must contain all values; `$size` — array length; index position with dot notation (`"marks.0"`).
2. **Embedded document:** a document used as the value of a key — keeps related data together (no join needed). Inner fields are queried with **dot notation**, e.g. `"address.city"`.

```
test> use college
switched to db college
college> db.students.insertMany([
```

```
  { _id: 1, name: "Asha", subjects: ["ADM", "SE", "AJP"], address: { city:
"Pune", pin: "411001" } },
  { _id: 2, name: "Ravi", subjects: ["ADM", "OS"], address: { city: "Nanded",
pin: "431601" } }
])
{ acknowledged: true, insertedIds: { '0': 1, '1': 2 } }
college> db.students.find({ subjects: "OS" }, { name: 1 })
[ { _id: 2, name: 'Ravi' } ]
college> db.students.find({ subjects: { $all: ["ADM", "SE"] } }, { name: 1 })
[ { _id: 1, name: 'Asha' } ]
college> db.students.find({ subjects: { $size: 2 } }, { name: 1 })
[ { _id: 2, name: 'Ravi' } ]
college> db.students.find({ "address.city": "Nanded" }, { name: 1, address: 1 })
[ { _id: 2, name: 'Ravi', address: { city: 'Nanded', pin: '431601' } } ]
```

*(Array 2 M, embedded document 2 M)*

### d) find() with $or, $nin, missing field, regex.

4 M

Sample data (assumed):

```
use library
db.books.insertMany([
  { title: "Database System Concepts", author: "Korth", category: "Textbook",
price: 850, discount: 10 },
  { title: "Data Mining Basics", author: "Han", category: "Textbook", price: 640
},
  { title: "Malgudi Days", author: "R. K. Narayan", category: "Fiction", price:
250, discount: 5 },
  { title: "Tinkle Digest", author: "Anant Pai", category: "Comics", price: 120
},
  { title: "Wings of Fire", author: "A. P. J. Abdul Kalam", category:
"Biography", price: 290 }
])
```

```
library> db.books.find({ $or: [ { author: "Korth" }, { price: { $lt: 300 } } ]
}, { _id: 0, title: 1 })
[
  { title: 'Database System Concepts' },
  { title: 'Malgudi Days' },
  { title: 'Tinkle Digest' },
  { title: 'Wings of Fire' }
]
library> db.books.find({ category: { $nin: ["Fiction", "Comics"] } }, { _id: 0,
title: 1, category: 1 })
[
  { title: 'Database System Concepts', category: 'Textbook' },
  { title: 'Data Mining Basics', category: 'Textbook' },
  { title: 'Wings of Fire', category: 'Biography' }
]
library> db.books.find({ discount: { $exists: false } }, { _id: 0, title: 1 })
[
  { title: 'Data Mining Basics' },
  { title: 'Tinkle Digest' },
  { title: 'Wings of Fire' }
]
library> db.books.find({ title: /^Data/ }, { _id: 0, title: 1 })
[
  { title: 'Database System Concepts' },
```

```
  { title: 'Data Mining Basics' }
]
```

(i) `$or` takes an array of conditions (OR query). (ii) `$nin` = not in the list. (iii) `$exists: false` selects documents without the key (`{discount: null}` would also match these, since null matches a missing key). (iv) A regular expression `/^Data/` matches titles beginning with "Data" (type-specific querying).

*(Each query 1 M)*

**e) Data mining vs data warehousing.** 4 M

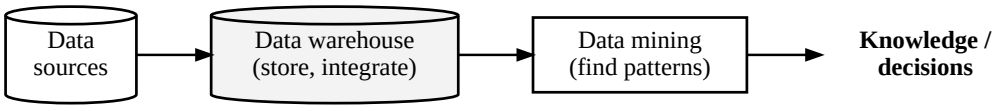


*Fig.: Data warehouse feeds data mining*

| Point | Data mining | Data warehousing |
|---|---|---|
| **Meaning** | Process of extracting hidden patterns / knowledge from large data. | Central repository that stores integrated historical data for analysis. |
| **Nature** | An analysis technique (process). | A storage system / database (environment). |
| **Purpose** | Prediction and discovery — e.g. which customers may leave. | Reporting and querying — e.g. total sales per region per year. |
| **Techniques / tools** | Classification, clustering, association rules, regression. | ETL tools, OLAP, star / snowflake schema, data marts. |
| **Order** | Comes after warehousing. | Comes first; supplies cleaned data for mining. |

*(Any four points, 1 M each)*

## Q.5

**a) 2PL and deadlock.** 6 M

**Two-Phase Locking:** every transaction obtains and releases locks in two phases — a **growing phase** (only acquire locks) followed by a **shrinking phase** (only release locks). The point at which the last lock is acquired is the **lock point**. 2PL ensures conflict serializability (serial order = order of lock points).

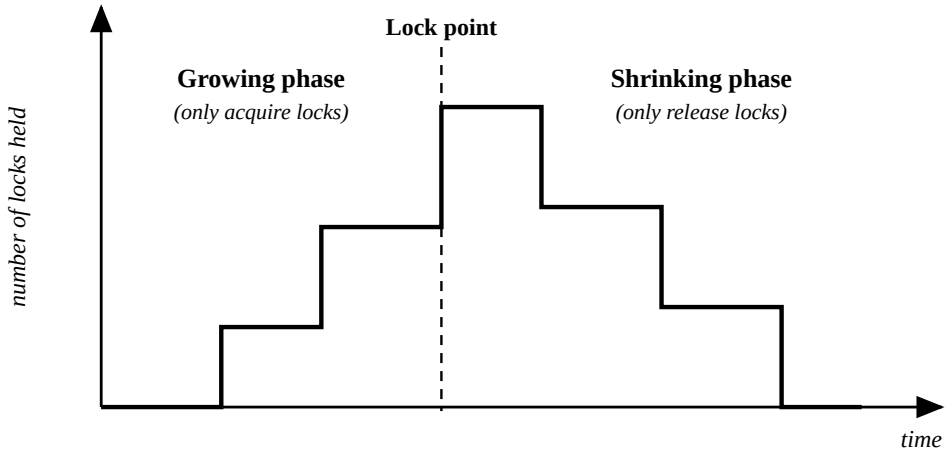


*Fig.: Two-phase locking — growing and shrinking phases*

**2PL does not prevent deadlock** — example (lock-manager decisions computed by program):

| Step | Txn | Operation | Lock manager | Locks held after step | Phase of txn |
|---|---|---|---|---|---|
| 1 | T1 | lock-X(A) | granted | T1:X(A) | growing |
| 2 | T1 | write(A) | — | T1:X(A) | growing |
| 3 | T2 | lock-X(B) | granted | T1:X(A), T2:X(B) | growing |
| 4 | T2 | write(B) | — | T1:X(A), T2:X(B) | growing |
| 5 | T1 | lock-X(B) | WAIT (held by T2:X) | T1:X(A), T2:X(B) | growing |
| 6 | T2 | lock-S(A) | WAIT (held by T1:X) | T1:X(A), T2:X(B) | growing |

*Both transactions are still in their growing phase and each waits for the other*

T1 holds X(A) and waits for B; T2 holds X(B) and waits for A. Both follow 2PL (no lock has been released), yet neither can proceed — a **deadlock**. The wait-for graph has the cycle T1 → T2 → T1, so the system must roll back one of them (victim) — or use prevention schemes such as wait-die / wound-wait or lock timeouts.

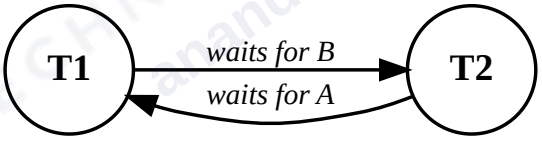


*Fig.: Wait-for graph with a cycle (deadlock)*

Variants: **strict 2PL** (X-locks held till commit — avoids cascading rollback) and **rigorous 2PL** (all locks till commit). Neither removes deadlocks.

*(2PL 2 M, example 3 M, conclusion 1 M)*

**b) XQuery — join and nested queries.** 6 M

Assumed XML data:

```
<customers>
  <customer><cid>C1</cid><cname>Sagar</cname><city>Pune</city></customer>
  <customer><cid>C2</cid><cname>Ayesha</cname><city>Nanded</city></customer>
  <customer><cid>C3</cid><cname>Vikram</cname><city>Latur</city></customer>
</customers>
```

```
<orders>
  <order><oid>O1</oid><cid>C1</cid><item>Laptop</item><amount>52000</amount>
</order>
  <order><oid>O2</oid><cid>C2</cid><item>Printer</item><amount>9500</amount>
</order>
  <order><oid>O3</oid><cid>C1</cid><item>Mouse</item><amount>600</amount>
</order>
  <order><oid>O4</oid><cid>C2</cid><item>Router</item><amount>2100</amount>
</order>
</orders>
```

**i) Join** — customer name, item and amount for orders above 1000:

*XQuery:*

```
for $c in doc("customers.xml")/customers/customer,
    $o in doc("orders.xml")/orders/order
where $c/cid = $o/cid and xs:decimal($o/amount) > 1000
return <purchase>{ $c/cname, $o/item, $o/amount }</purchase>
```

*Output:*

```
<purchase>
   <cname>Sagar</cname>
   <item>Laptop</item>
   <amount>52000</amount>
</purchase>
<purchase>
   <cname>Ayesha</cname>
   <item>Printer</item>
   <amount>9500</amount>
</purchase>
<purchase>
   <cname>Ayesha</cname>
   <item>Router</item>
   <amount>2100</amount>
</purchase>
```

**ii) Nested query** — every customer with all his orders and total amount (a customer with no order appears with an empty list):

*XQuery:*

```
<report>{
  for $c in doc("customers.xml")/customers/customer
  let $ord := doc("orders.xml")/orders/order[cid = $c/cid]
  return <customer name="{$c/cname}" total="{ sum($ord/amount) }">{
           for $o in $ord return <item>{ $o/item/text() }</item>
         }</customer>
}</report>
```

*Output:*

```
<report>
   <customer name="Sagar" total="52600">
      <item>Laptop</item>
      <item>Mouse</item>
   </customer>
   <customer name="Ayesha" total="11600">
      <item>Printer</item>
      <item>Router</item>
   </customer>
   <customer name="Vikram" total="0"/>
</report>
```

In the join, the `where` clause equates cid of both documents (like an SQL equi-join). The nested query places a FLWOR expression inside the element constructor of the outer FLWOR — like an outer join with grouping.

*(Join 3 M, nested query 3 M)*

### c) Basic operations in MongoDB shell.

6 M

| Operation | Syntax |
|---|---|
| Create / switch database | `use <db_name>` |
| Create collection | `db.createCollection("name")` (or automatically on first insert) |
| Insert | `db.coll.insertOne({…})`, `db.coll.insertMany([{…}, …])` |
| Read | `db.coll.find(query, projection)`, `findOne()` |
| Update | `db.coll.updateOne(filter, {$set: {…}})`, `updateMany()` |
| Delete | `db.coll.deleteOne(filter)`, `deleteMany(filter)` |
| Drop | `db.coll.drop()`, `db.dropDatabase()` |

```
test> use hostel
switched to db hostel
hostel> db.createCollection("rooms")
{ ok: 1 }
hostel> db.rooms.insertMany([
  { room: 101, student: "Amit", fees: 30000 },
  { room: 102, student: "Bhavna", fees: 30000 },
  { room: 103, student: "Chetan", fees: 28000 }
])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('6720a1b2c3d4e5f600000001'),
    '1': ObjectId('6720a1b2c3d4e5f600000002'),
    '2': ObjectId('6720a1b2c3d4e5f600000003')
  }
}
hostel> db.rooms.find({ fees: 30000 }, { _id: 0 })
[
  { room: 101, student: 'Amit', fees: 30000 },
  { room: 102, student: 'Bhavna', fees: 30000 }
]
```

```
hostel> db.rooms.updateMany({ fees: 30000 }, { $inc: { fees: 2000 } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 2,
  upsertedCount: 0
}
hostel> db.rooms.deleteOne({ room: 103 })
{ acknowledged: true, deletedCount: 1 }
hostel> db.rooms.find({}, { _id: 0 })
[
  { room: 101, student: 'Amit', fees: 32000 },
  { room: 102, student: 'Bhavna', fees: 32000 }
]
hostel> db.rooms.drop()
true
hostel> show collections
```

`$inc` increases a numeric field; updateMany() changed 2 documents. After `drop()` the collection list is empty.

*(Each operation with syntax and example 1 M)*

## Q.6

**a) Parallel vs distributed database (six points).** 6 M

| Point | Parallel database | Distributed database |
|---|---|---|
| **Location** | Processors/disks are in one place (same building / machine room). | Sites are geographically separated (different cities / branches). |
| **Interconnection** | Very fast interconnection network (bus, switch, mesh). | Slower network — LAN / WAN / Internet. |
| **Main aim** | Performance — higher throughput and lower response time through parallelism. | Data sharing, local autonomy and high availability for users at many sites. |
| **Autonomy** | Nodes are not autonomous; whole system works as one DBMS. | Each site has local autonomy and its own local DBMS. |
| **Nodes / software** | Usually identical (homogeneous) processors and the same DBMS. | Sites may be heterogeneous (different hardware, OS, DBMS). |
| **Cost of communication** | Communication cost is negligible. | Communication cost is high; query plans try to reduce data transfer. |
| **Failure** | Failure of the machine room affects the whole system. | Failure of one site does not stop the other sites (replicated data remains available). |

*(Any six points, 1 M each)*

**b) Decision-tree classifier; validating a classifier.** 6 M

A **decision-tree classifier** is a tree in which each **internal node** tests an attribute, each **branch** is an outcome of the test and each **leaf** gives a class label. To classify a new item we start at the root and follow the branches matching its attribute values until a leaf is reached. The tree is built top-down (greedy): at each node the attribute that best separates the classes is chosen — measured by **information gain** (entropy) or Gini index — and the data is partitioned; partitioning stops when a node is pure (one class) or no attributes remain.

**Example — predict result of a student** (training data, 11 students):

| # | Attendance | Assignments | Result |
|---|---|---|---|
| 1 | high | complete | pass |
| 2 | high | complete | pass |
| 3 | high | incomplete | pass |
| 4 | high | complete | pass |
| 5 | low | complete | pass |
| 6 | low | incomplete | fail |
| 7 | low | incomplete | fail |
| 8 | low | complete | pass |
| 9 | low | incomplete | fail |
| 10 | high | incomplete | pass |
| 11 | high | incomplete | pass |

Entropy(result) = 0.845; Gain(attendance) = 0.404; Gain(assignments) = 0.300. Attendance has the higher gain, so it is the root test; for low attendance the assignments attribute separates the classes completely. The tree built by the ID3 algorithm (computed by program):

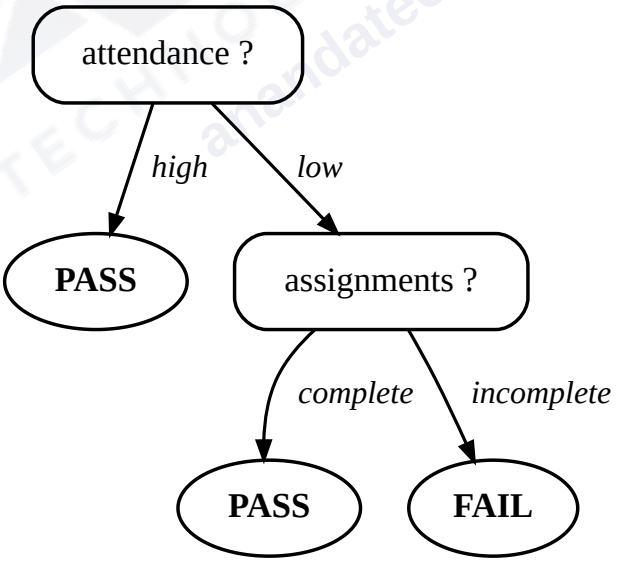


*Fig.: Decision tree for predicting result*

**Validating a classifier:** its quality is measured on **test data** not used for training (hold-out) or by **cross-validation** (data divided into k parts; each part is used once for testing). Measures are

computed from the confusion matrix — **accuracy** = (TP + TN) / total, **precision** = TP / (TP + FP), **recall** = TP / (TP + FN).

| Test students | Actual | Predicted |
|---|---|---|
| high, complete | pass | pass |
| low, incomplete | fail | fail |
| high, incomplete | fail | pass |
| low, complete | pass | pass |
| low, incomplete | pass | fail |
| high, complete | pass | pass |

With ‘pass’ as positive: TP = 3, TN = 1, FP = 1, FN = 1; accuracy = 4/6 = **67%**, precision = 3/4 = 0.75, recall = 3/4 = 0.75.

*(Explanation 2 M, example tree 2 M, validation 2 M)*

**c) XPath and XQuery; differences.** 6 M

**XPath** is a language for selecting nodes of an XML document using path expressions (`/` child, `//` descendant, `@` attribute, `[ ]` predicate, functions such as `count()`).

Document *library.xml*:

```
<library>
  <book id="B1" year="2019"><title>Database Concepts</title>
<author>Korth</author><price>850</price></book>
  <book id="B2" year="2021"><title>Learning XML</title><author>Ray</author>
<price>499</price></book>
  <book id="B3" year="2023"><title>NoSQL Distilled</title>
<author>Fowler</author><price>620</price></book>
</library>
```

| Requirement | XPath expression | Result (executed) |
|---|---|---|
| titles of all books | `/library/book/title` | `<title>Database Concepts</title> <title>Learning XML</title> <title>NoSQL Distilled</title>` |
| author of book B2 | `/library/book[@id="B2"]/author/text()` | `Ray` |
| titles of books costing less than 700 | `//book[price < 700]/title` | `<title>Learning XML</title> <title>NoSQL Distilled</title>` |

**XQuery** is the W3C query language for XML (like SQL for tables). It uses XPath to locate nodes and FLWOR expressions to filter, join, sort and construct new XML.

*XQuery:*

```
for $b in doc("library.xml")/library/book
where xs:integer($b/@year) >= 2020
order by $b/title
return <recent>{ $b/title/text() } ({ string($b/@year) })</recent>
```

*Output:*

```
<recent>Learning XML (2021)</recent>
<recent>NoSQL Distilled (2023)</recent>
```

| Point | XPath | XQuery |
|---|---|---|
| **Purpose** | Language to navigate an XML tree and select nodes. | Full query language to query, join and restructure XML data. |
| **Construct new XML** | Cannot create new elements; only selects existing nodes / values. | Can construct new XML elements and documents in the result. |
| **Joins / sorting** | No joins between documents; no ordering clause. | Supports joins, nested queries, sorting (order by), grouping. |
| **Relation** | XPath 2.0 is a subset of XQuery 1.0. | XQuery is a superset of XPath — like SQL for XML. |

*(XPath with example 1½ M, XQuery with example 1½ M, four differences 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