

Advance Database Management (315324)

Model Question Paper — Set B · 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, MULTISSET) 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) Lock and lock-compatibility matrix.

2 M

Lock: A lock is a mechanism (variable associated with a data item) used to control concurrent access to the data item; a transaction must hold an appropriate lock before reading or writing the item. Two modes: shared (S) and exclusive (X).

Lock requested ↓ / Lock held →	S (shared)	X (exclusive)
S (shared)	true (granted)	false (wait)
X (exclusive)	false (wait)	false (wait)

Only S–S is compatible; any combination involving X is incompatible, so the requesting transaction must wait.

(Definition 1 M, matrix 1 M)

b) Throughput and response time.

2 M

- Throughput:** the number of tasks (transactions / queries) that can be completed in a given time interval, e.g. transactions per second. Parallelism among many small transactions increases throughput.
- Response time:** the amount of time taken to complete a single task from the time it is submitted. Parallelism within a large query reduces its response time.

(Each definition 1 M)

c) XPath with two expressions.

2 M

XPath is a language for addressing (selecting) parts of an XML document using **path expressions**; the document is viewed as a tree of nodes. It is used inside XQuery and XSLT.

- /college/student/name — selects the name elements of all students (children of student, which are children of the root college).
- //student[@branch="IF"] — selects student elements anywhere in the document whose attribute branch is IF.

(Definition 1 M, two expressions 1 M)

d) Four benefits of NoSQL.

2 M

1. **Flexible schema** — new fields can be added without altering a table.
2. **Horizontal scalability** — data is spread (sharded) over many cheap servers.
3. **High availability** — automatic replication; no single point of failure.
4. **Handles big and unstructured data with high performance** — suitable for real-time web, IoT and social-media apps.

(Any four, ½ M each)

e) Syntax of find().

2 M

```
db.collection.find( query, projection )
```

1. **query** (optional) — a document giving the selection criteria, e.g. { marks: { \$gt: 60 } }.
Omitted or {} → all documents.
2. **projection** (optional) — a document saying which keys to return: 1 = include, 0 = exclude, e.g. { name: 1, _id: 0 }.

find() returns a cursor; methods like .sort(), .limit(), .count() can be chained.

(Syntax 1 M, parameters 1 M)

f) ETL.

2 M

ETL = Extract, Transform, Load — the warehouse tools that bring data from sources into the data warehouse.

1. **Extract:** read / collect the required data from different operational sources.
2. **Transform:** clean (remove errors, duplicates), convert formats/units and integrate data into the warehouse schema.
3. **Load:** store the transformed data into the warehouse and refresh it periodically.

(Expansion ½ M, purpose 1½ M)

g) Distributed database; two types.

2 M

Distributed database: a collection of multiple, logically interrelated databases stored at different sites (computers) connected by a network, managed by a DDBMS so that it appears to users as one database.

Types: i) Homogeneous distributed database ii) Heterogeneous distributed database.

(Definition 1 M, types 1 M)

Q.2

a) 2PL; need of strict and rigorous 2PL.

4 M

Two-phase locking protocol: each transaction issues lock and unlock requests in two phases —

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

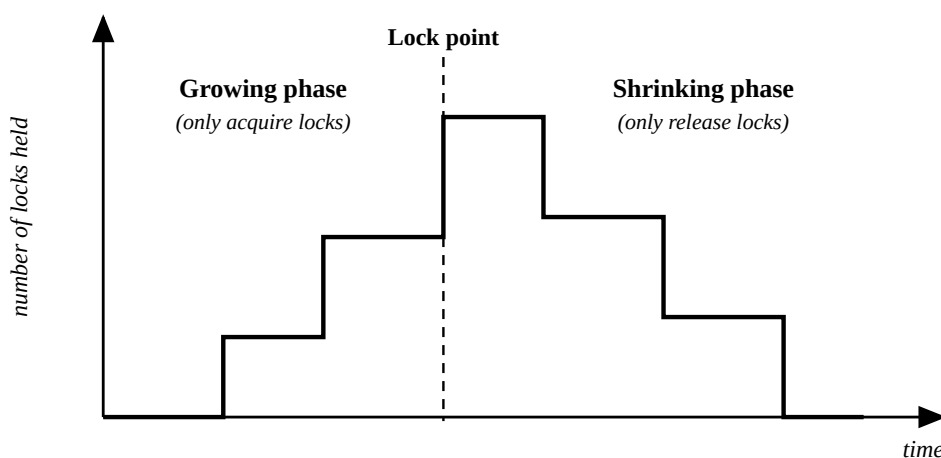


Fig.: Two-phase locking — growing and shrinking phases

Example (checked by program):

Step	Txn	Operation	Lock manager	Locks held after step	Phase of txn
1	T1	lock-S(X)	granted	T1:S(X)	growing
2	T1	read(X)	—	T1:S(X)	growing
3	T1	lock-X(Y)	granted	T1:S(X), T1:X(Y)	growing
4	T1	read(Y)	—	T1:S(X), T1:X(Y)	growing
5	T1	write(Y)	—	T1:S(X), T1:X(Y)	growing
6	T1	unlock(X)	released	T1:X(Y)	shrinking
7	T2	lock-X(X)	granted	T2:X(X), T1:X(Y)	growing
8	T1	unlock(Y)	released	T2:X(X)	shrinking
9	T2	read(X)	—	T2:X(X)	growing
10	T2	write(X)	—	T2:X(X)	growing
11	T2	unlock(X)	released	—	shrinking

T1 obtains S(X), X(Y) [lock point at step 3] and then releases; T2 locks X only after T1 released it

2PL guarantees **conflict serializability**, but (i) it does not prevent **deadlock** and (ii) it allows **cascading rollback**: in the example T1 releases Y before committing; if another transaction reads Y and T1 later aborts, that transaction must also roll back.

- **Strict 2PL**: all *exclusive* locks are held until the transaction commits or aborts → uncommitted data cannot be read by others, so cascading rollbacks are avoided.
- **Rigorous 2PL**: all locks (S and X) are held until commit/abort → transactions are serializable in the order in which they commit. Most commercial DBMSs use strict / rigorous 2PL.

(2PL 2 M, strict and rigorous 2 M)

b) Parallel vs distributed database.

4 M

Point	Parallel database	Distributed database
Location	Processors/disks are in one place (same building / machine room).	Sites are geographically separated (different cities / branches).
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.
Query execution	A single query is split and executed in parallel (intra-query parallelism).	A query may need data from remote sites; local and global (distributed) transactions.
Examples	Teradata, Oracle RAC, parallel query in PostgreSQL.	Bank branches database, Google Spanner, Apache Cassandra cluster across regions.

(Any four points, 1 M each)

c) Structured types and type inheritance in SQL.

4 M

Structured (user-defined) types allow composite attributes to be defined and used as column types. **Type inheritance** allows a new type to be defined as a sub-type of an existing type: it inherits all attributes and methods of the super-type and may add its own.

```
-- structured types (SQL:1999)
CREATE TYPE Name AS (first_name VARCHAR(20), last_name VARCHAR(20)) FINAL;
CREATE TYPE Address AS (street VARCHAR(20), city VARCHAR(20), pin VARCHAR(6))
NOT FINAL;
CREATE TABLE person (name Name, address Address, dob DATE);

-- type inheritance
CREATE TYPE Person AS (name VARCHAR(20), address VARCHAR(20)) NOT FINAL;
CREATE TYPE Student UNDER Person (degree VARCHAR(20), department VARCHAR(20));
CREATE TYPE Teacher UNDER Person (salary INTEGER, department VARCHAR(20));
```

- FINAL — no sub-types can be created from the type; NOT FINAL — sub-types are allowed.
- Student and Teacher inherit name and address from Person; components are accessed by dot notation, e.g. address.city.
- SQL does not support multiple inheritance of types (a type cannot be UNDER two types).

Executed example (PostgreSQL supports structured/composite types; it has no UNDER for types, so sub-typing is shown with the equivalent table inheritance):

```
adm_mp=# CREATE TYPE name_t AS (first_name VARCHAR(20), last_name VARCHAR(20));
CREATE TYPE
adm_mp=# CREATE TYPE address_t AS (street VARCHAR(20), city VARCHAR(20), pin
VARCHAR(6));
CREATE TYPE
adm_mp=# CREATE TABLE person (name name_t, address address_t, dob DATE);
CREATE TABLE
adm_mp=# INSERT INTO person VALUES (ROW('Meera', 'Kulkarni'), ROW('MG Road',
'Pune', '411001'), '2007-03-14');
INSERT 0 1
adm_mp=# SELECT (name).last_name, (address).city FROM person;
last_name | city
```

-----+-----
Kulkarni | Pune
(1 row)

(Explanation 2 M, example 2 M)

d) Structured, semi-structured, unstructured data.

4 M

Point	Structured data	Semi-structured data	Unstructured data
Format	Fixed format — rows and columns with a schema.	Self-describing tags / keys; flexible schema.	No predefined format or model.
Storage	Relational DBMS (Oracle, MySQL).	XML / JSON files, MongoDB.	Files, data lakes, NoSQL / object storage.
Searching	Easy with SQL.	Using XPath / XQuery / JSON queries.	Difficult; needs text mining, AI, image processing.
Examples	Student table, bank transactions, sensor readings in tables.	XML invoice, JSON API response, e-mail headers.	Images, videos, audio, PDF files, social-media posts.

- **Structured data** has a fixed schema; each record has the same fields of fixed type — e.g. Student(roll_no, name, marks) in MySQL.
- **Semi-structured data** carries its structure with it through tags/keys; records may have different fields — e.g. { "name": "Asha", "skills": ["SQL", "Python"] }.
- **Unstructured data** has no data model — about 80% of today's data (photos, CCTV video, voice notes, PDFs); NoSQL stores and data lakes are used for it.

(Each type with example 1½ M)

Q.3

a) Transaction server vs data server.

4 M

Point	Transaction server	Data server
Unit of interaction	Client sends a whole request (SQL query / remote procedure call).	Client requests data items — files, pages or objects.
Where processing happens	Query processing and transaction execution on the server.	Most processing on the client; server only supplies and stores data.
Data shipped	Only the result of the query is sent to client.	Pages / objects are shipped to client and cached there.
Used in	Relational databases (Oracle, SQL Server) with ODBC/JDBC clients.	Object-oriented databases, CAD systems, file servers.
Issues	Server load; concurrency among server processes (shared memory, lock table).	Page shipping vs item shipping, lock granularity, data and lock caching on clients.

(Any four points, 1 M each)

b) Distributed database system architecture.

4 M

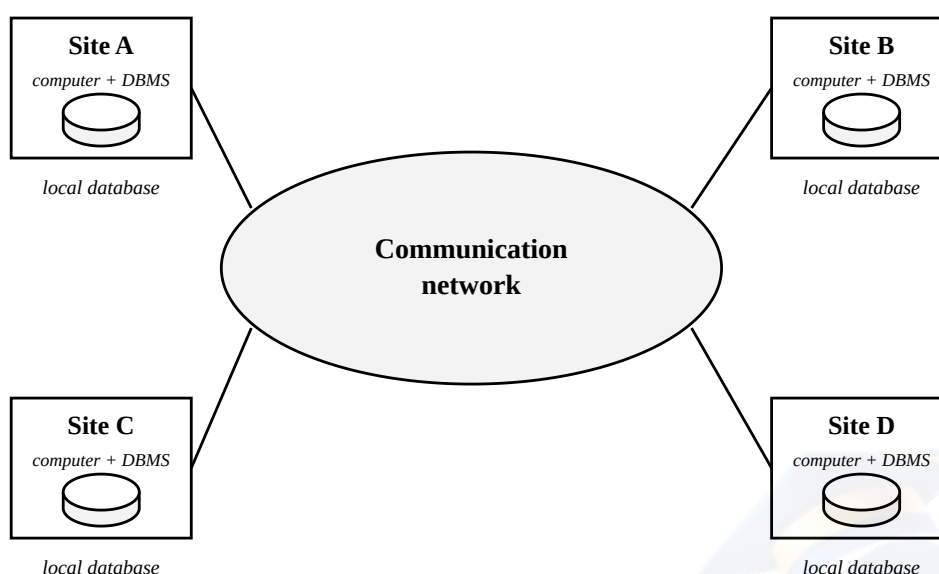


Fig.: Distributed database system architecture

1. The database is stored on several computers (**sites / nodes**) located at different places and connected by a **communication network** (LAN / WAN).
2. Each site has its own local DBMS and local database and can process **local transactions** (using only local data) independently — local autonomy.
3. A **global transaction** accesses data at several sites; a transaction manager at each site and a transaction coordinator manage it (commit protocols such as 2PC).
4. Data is stored by **replication** (copies at several sites — availability) and/or **fragmentation** (horizontal: subsets of rows; vertical: subsets of columns), giving location and fragmentation transparency to users.

(Diagram 2 M, explanation 2 M)

c) insertMany() and find() on employees.

4 M

```

test> use company
switched to db company
company> db.employees.insertMany([
  { name: "Ajay", dept: "Sales", salary: 38000 },
  { name: "Neha", dept: "IT", salary: 52000 },
  { name: "Farhan", dept: "HR", salary: 41000 },
  { name: "Pooja", dept: "IT", salary: 35000 }
])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('6720a1b2c3d4e5f6000000001'),
    '1': ObjectId('6720a1b2c3d4e5f6000000002'),
    '2': ObjectId('6720a1b2c3d4e5f6000000003'),
    '3': ObjectId('6720a1b2c3d4e5f6000000004')
  }
}
company> db.employees.find({ salary: { $gt: 40000 } })
[

```

```
{
  _id: ObjectId('6720a1b2c3d4e5f6000000002'),
  name: 'Neha',
  dept: 'IT',
  salary: 52000
},
{
  _id: ObjectId('6720a1b2c3d4e5f6000000003'),
  name: 'Farhan',
  dept: 'HR',
  salary: 41000
}
]
```

`insertMany()` takes an array of documents and returns the generated `_id` values (ObjectIds); `$gt` selects `salary > 40000` — 2 of 4 employees are displayed.

(Insert 2 M, find 2 M)

d) Data lakehouse and its components.

4 M

A **data lakehouse** is a modern data-management architecture that combines the low-cost, flexible storage of a **data lake** (all types of raw data in open formats) with the data management features of a **data warehouse** (ACID transactions, schema, indexing, fast SQL/BI). A single copy of data serves BI reports as well as data science / machine learning. Examples: Databricks Delta Lake, Apache Iceberg, Apache Hudi.

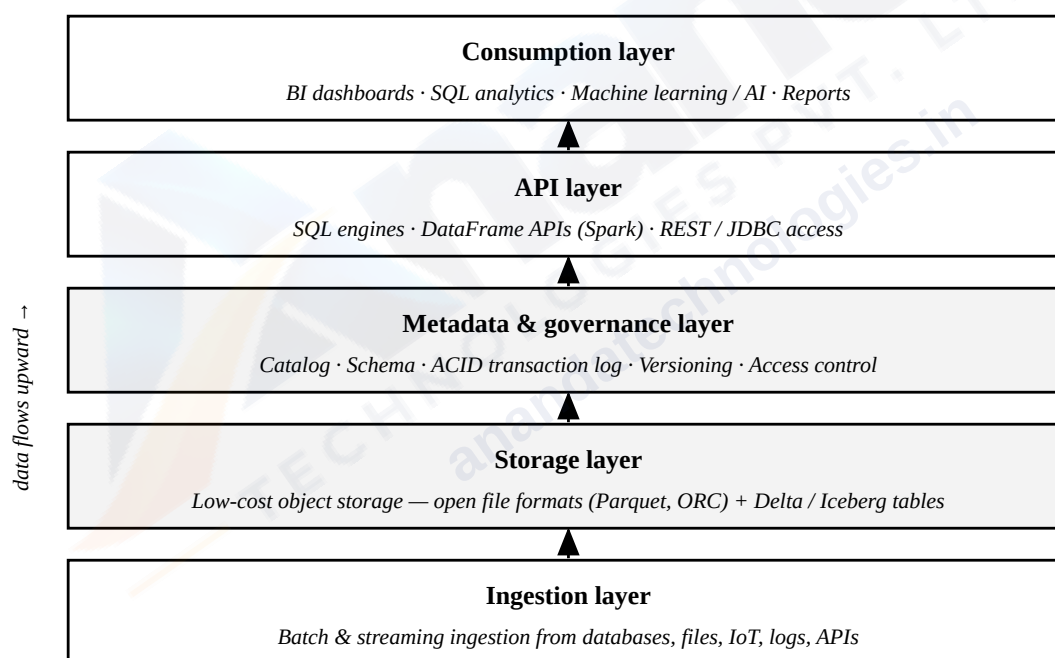


Fig.: Layers (components) of a data lakehouse

- Ingestion layer:** brings batch and streaming data from databases, files, IoT devices, logs into the lakehouse.
- Storage layer:** cheap cloud object storage (S3, ADLS) holding data in open columnar file formats such as Parquet / ORC.
- Metadata layer:** catalog of tables and schemas, ACID transaction log, versioning (time travel), schema enforcement, indexing and caching — this is what turns a lake into a lakehouse.

4. **API layer:** SQL engines and DataFrame APIs so that tools can read data directly and efficiently.
5. **Consumption layer:** BI dashboards, reporting, SQL analytics and ML / AI tools.

(Definition 1 M, components 3 M)

Q.4

a) Clauses of FLWOR expression.

4 M

1. **for** — binds a variable to each item of a sequence in turn (like FROM of SQL).
2. **let** — binds a variable to a whole value / sequence (no iteration).
3. **where** — filters the tuples using a condition (like WHERE).
4. **order by** — sorts the results (ascending / descending).
5. **return** — constructs the result for each tuple, often as new XML elements (like SELECT).

Example — names and annual salary of employees earning more than 40000, highest salary first:

XML document *company.xml*:

```
<company>
  <emp id="E1"><name>Ajay</name><dept>Sales</dept><salary>38000</salary></emp>
  <emp id="E2"><name>Neha</name><dept>IT</dept><salary>52000</salary></emp>
  <emp id="E3"><name>Farhan</name><dept>HR</dept><salary>41000</salary></emp>
  <emp id="E4"><name>Pooja</name><dept>IT</dept><salary>35000</salary></emp>
</company>
```

XQuery:

```
for $e in doc("company.xml")/company/emp
let $annual := xs:decimal($e/salary) * 12
where xs:decimal($e/salary) > 40000
order by xs:decimal($e/salary) descending
return <employee name="{ $e/name }" annual="{ $annual }"/>
```

Output:

```
<employee name="Neha" annual="624000"/>
<employee name="Farhan" annual="492000"/>
```

(Clauses 2 M, example 2 M)

b) Four MongoDB data types.

4 M

MongoDB stores documents in **BSON** (binary JSON), which supports more types than JSON:

1. **String** — UTF-8 text: name: "Asha".
2. **Number** — 64-bit floating point by default (marks: 82.5); NumberInt() / NumberLong() for integers.
3. **Boolean** — true / false; **null** — remark: null.
4. **Date** — stored as milliseconds since 1-Jan-1970 (UTC): new Date().
5. **Array** — ordered list of values of any type: skills: ["SQL", "Java"].
6. **Embedded document** — a document inside a document: address: { city: "Pune", pin: "411001" }.
7. **ObjectId** — 12-byte unique id automatically given to _id.

```
test> use college
switched to db college
college> db.students.insertOne({ _id: 101, name: "Asha", marks: 82.5, passed:
true, remark: null, dob: new Date("2008-04-15"), skills: ["SQL", "Java"],
address: { city: "Pune", pin: "411001" } })
{ acknowledged: true, insertedId: 101 }
college> db.students.find({ "address.city": "Pune", skills: "SQL" })
[
  {
    _id: 101,
    name: 'Asha',
    marks: 82.5,
    passed: true,
    remark: null,
    dob: ISODate('2008-04-15T00:00:00.000Z'),
    skills: [ 'SQL', 'Java' ],
    address: { city: 'Pune', pin: '411001' }
  }
]
```

Embedded fields are queried with dot notation ("address.city") and an array field matches if any element matches.

(Any four types with example, 1 M each)

c) Support and confidence; numerical.

4 M

An **association rule** $A \rightarrow B$ (e.g. bread $\rightarrow$ butter) says that customers who buy A also tend to buy B. Its strength is measured by:

1. **Support** = fraction of all transactions that contain both A and B = (no. of transactions containing $A \cup B$) / (total transactions). Low support means the rule is not common.
2. **Confidence** = fraction of transactions containing A that also contain B = $\text{support}(A \cup B) / \text{support}(A)$. It measures how often the rule is true.

TID	Items	bread?	bread & butter?
T1	bread, milk	yes	—
T2	bread, butter, milk	yes	yes
T3	bread, butter	yes	yes
T4	milk, tea	—	—
T5	bread, butter, milk, tea	yes	yes

Total transactions = 5; containing bread = 4; containing bread and butter = 3.

Support(bread $\rightarrow$ butter) = $3/5 = 0.60$ (60%)

Confidence(bread $\rightarrow$ butter) = $3/4 = 0.75$ (75%)

So 60% of all baskets contain bread and butter together, and 75% of customers who buy bread also buy butter. (Computed by program.)

(Definitions 2 M, calculation 2 M)

d) Shared-nothing and hierarchical architectures.

4 M

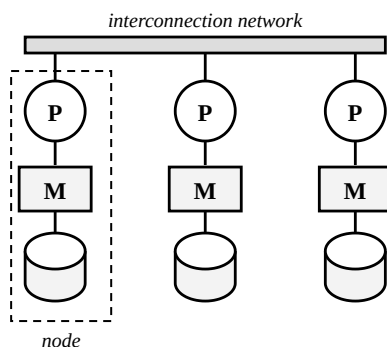


Fig. (c): Shared nothing

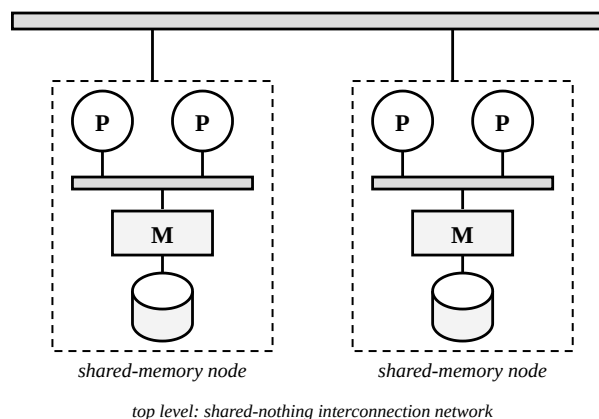


Fig. (d): Hierarchical

P = processor, M = memory, cylinder = disk

Shared nothing: each node has its own processor, memory and disk(s); nodes communicate only through a high-speed interconnection network. A node acts as the server for the data on its own disk.

- Most scalable — thousands of processors; no memory or disk bottleneck; used by Teradata, Greenplum, Hadoop / NoSQL clusters.
- Cost of communication and of non-local disk access is higher (data must be sent through the network).

Hierarchical: combines the above. At the top level, nodes are connected by an interconnection network and share nothing; each node itself is a shared-memory system (or shared-disk) with a few processors.

- Combines fast communication within a node and the scalability of shared nothing at the top level; most large parallel systems today are built this way.

(Each architecture: diagram 1 M, explanation 1 M)

e) Object-oriented vs object-relational database.

4 M

Point	Object-oriented database (OODB)	Object-relational database (ORDB)
Basis	Built on an object-oriented programming language (C++, Java, Smalltalk) — persistent objects.	Built on the relational model, extended with object features (SQL:1999).
Data stored as	Objects with object identifiers (OIDs), classes and methods.	Tables (relations) whose columns may have user-defined / complex types.
Query language	OQL (ODMG) or navigation through programming language — no widely used standard.	Standard SQL extended with types, inheritance, arrays, references.
Data access	Mostly navigational (follow pointers).	Mostly declarative (set-oriented queries).
Examples	db4o, ObjectDB, Versant, GemStone/S.	Oracle, PostgreSQL, IBM DB2, Informix.

(Any four points, 1 M each)

Q.5

a) Lock-based protocols: modes, compatibility, granting of locks.

6 M

In lock-based protocols a transaction must hold a lock on a data item before accessing it; locks are requested from and granted by the **concurrency-control manager (lock manager)**.

1) Modes of locks

1. **Shared (S):** lock-S(Q) — the holder can read but not write Q.
2. **Exclusive (X):** lock-X(Q) — the holder can read and write Q. Released by unlock(Q).

2) Lock-compatibility: a lock is granted only if the requested mode is compatible with the modes of locks already held on the item by other transactions:

Lock requested ↓ / Lock held →	S (shared)	X (exclusive)
S (shared)	true (granted)	false (wait)
X (exclusive)	false (wait)	false (wait)

3) Granting of locks: when T_i requests a lock on Q in mode M, the lock manager grants it if

1. no other transaction holds a lock on Q in a mode that conflicts with M, and
2. no other transaction is **waiting** for a lock on Q that made its request earlier than T_i .

The second condition prevents **starvation** (e.g. a transaction waiting for X-lock while a stream of transactions keeps getting S-locks). Otherwise the requester waits until the conflicting locks are released.

Example (lock manager decisions computed by program):

Step	Txn	Operation	Lock manager	Locks held after step	Phase of txn
1	T1	lock-X(B)	granted	T1:X(B)	growing
2	T1	read(B)	—	T1:X(B)	growing
3	T2	lock-S(A)	granted	T2:S(A), T1:X(B)	growing
4	T2	read(A)	—	T2:S(A), T1:X(B)	growing
5	T1	write(B)	—	T2:S(A), T1:X(B)	growing
6	T2	lock-S(B)	WAIT (held by T1:X)	T2:S(A), T1:X(B)	growing
7	T1	unlock(B)	released	T2:S(A)	shrinking
8	T2	lock-S(B)	granted	T2:S(A), T2:S(B)	growing
9	T2	read(B)	—	T2:S(A), T2:S(B)	growing
10	T3	lock-S(A)	granted	T2:S(A), T3:S(A), T2:S(B)	growing
11	T3	read(A)	—	T2:S(A), T3:S(A), T2:S(B)	growing
12	T2	unlock(A)	released	T3:S(A), T2:S(B)	shrinking
13	T2	unlock(B)	released	T3:S(A)	shrinking
14	T3	unlock(A)	released	—	shrinking

T2's request for S(B) waits while T1 holds X(B); it is granted after T1 unlocks B. T2 and T3 share S(A).

(Lock modes 1 M, compatibility 2 M, granting rules 1 M, example 2 M)

b) SQL — structured type, array and multiset columns.

6 M

SQL:1999 / 2003 form (to write in the exam):

```
CREATE TYPE address_t AS (street VARCHAR(20), city VARCHAR(15), pin CHAR(6));

CREATE TABLE employee (
    eid      INT PRIMARY KEY,
    ename    VARCHAR(20),
    address  address_t,
    phones   VARCHAR(12) ARRAY[3],
    skills   VARCHAR(15) MULTISET );

INSERT INTO employee VALUES (1, 'Rahul', address_t('FC Road', 'Pune', '411004'),
    ARRAY['9822011111', '9011122233'], MULTISET['Java', 'Python', 'SQL']);

SELECT ename FROM employee WHERE address.city = 'Pune';
SELECT ename, phones[1] FROM employee;
SELECT ename FROM employee WHERE 'Python' MEMBER OF skills;
```

Executed on PostgreSQL (composite-type field written as (address).city; the multiset is stored as an array and searched with = ANY):

```
adm_mp=# CREATE TYPE address_t AS (street VARCHAR(20), city VARCHAR(15), pin
CHAR(6));
CREATE TYPE
adm_mp=# CREATE TABLE employee (
...     eid      INT PRIMARY KEY,
...     ename    VARCHAR(20),
...     address  address_t,
...     phones   VARCHAR(12)[3],
...     skills   VARCHAR(15)[]           -- multiset of skills (PostgreSQL: array)
... );
CREATE TABLE
adm_mp=# INSERT INTO employee VALUES (1, 'Rahul', ROW('FC Road', 'Pune',
'411004'), ARRAY['9822011111', '9011122233'], ARRAY['Java', 'Python', 'SQL']);
INSERT 0 1
adm_mp=# INSERT INTO employee VALUES (2, 'Sana', ROW('Shivaji Nagar', 'Nanded',
'431602'), ARRAY['9422033344'], ARRAY['PHP', 'SQL']);
INSERT 0 1
adm_mp=# SELECT ename FROM employee WHERE (address).city = 'Pune';
    ename
-----
    Rahul
(1 row)
adm_mp=# SELECT ename, phones[1] AS first_phone FROM employee;
    ename | first_phone
-----+-----
    Rahul | 9822011111
    Sana  | 9422033344
(2 rows)
adm_mp=# SELECT ename FROM employee WHERE 'Python' = ANY (skills);
    ename
-----
    Rahul
(1 row)
```

(Type 1 M, table 1 M, insert 1 M, three queries 3 M)

c) Output of commands in MongoDB shell.

6 M

The MongoDB shell (mongosh) is a full **JavaScript interpreter**, so any JavaScript expression can be typed at the prompt and its value is printed.

```
test> new Date("2026/1/26")
ISODate('2026-01-25T18:30:00.000Z')
test> "Advance Database".toUpperCase()
'ADVANCE DATABASE'
test> "MongoDB".length
7
test> [5, 10, 15].concat([20])
[ 5, 10, 15, 20 ]
test> Math.max(12, 45, 7)
45
test> "MSBTE-K-Scheme".split("-")
[ 'MSBTE', 'K', 'Scheme' ]
```

1. `new Date("2026/1/26")` creates a Date object for 26-Jan-2026, 00:00 local time. mongosh displays dates in UTC as `ISODate()`; in India (UTC+5:30) midnight is 18:30 of the previous day.
2. `toUpperCase()` returns the string in capital letters.
3. `length` gives the number of characters (7).
4. `concat()` joins two arrays and returns a new array.
5. `Math.max()` returns the largest argument.
6. `split("-")` breaks the string at every “-” and returns an array of parts.

(Each output with explanation 1 M)

Q.6

a) Three-tier data-warehouse architecture.

6 M

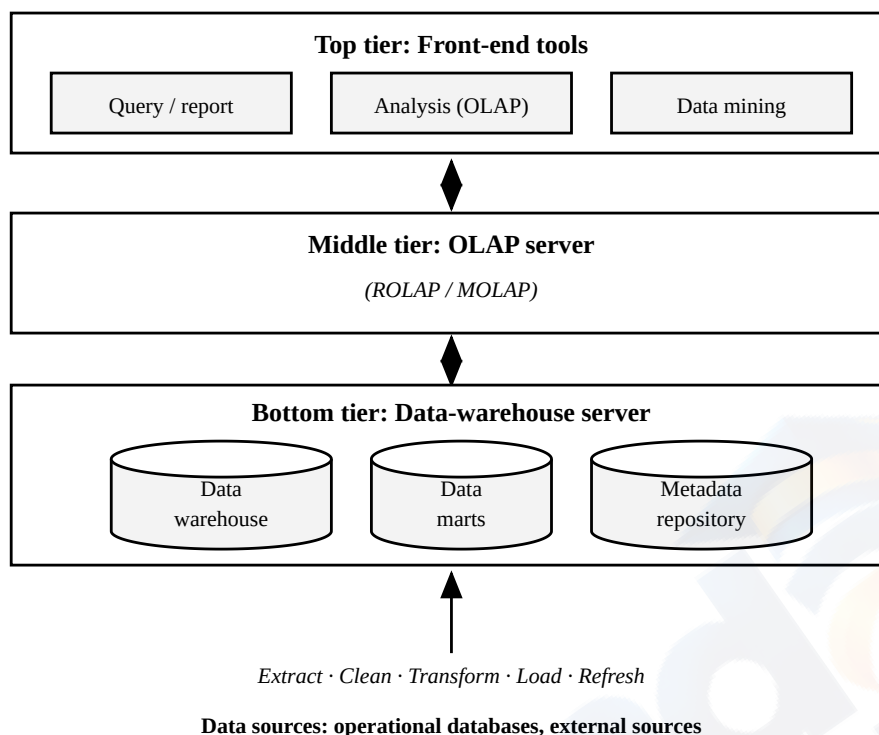


Fig.: Three-tier data-warehouse architecture

1) Bottom tier — data-warehouse server:

- A relational database system that stores the warehouse, data marts and the metadata repository.
- Back-end tools (ETL) extract data from operational databases and external sources, clean and transform it, load it, and refresh it periodically; gateways such as ODBC/JDBC are used.

2) Middle tier — OLAP server:

- Presents multidimensional views of data (cubes) to users.
- Implemented as **ROLAP** (relational OLAP — maps multidimensional operations to SQL on relational tables) or **MOLAP** (multidimensional OLAP — special array-based storage).

3) Top tier — front-end client tools:

- Query and reporting tools, analysis tools (slice, dice, roll-up, drill-down) and data-mining tools (trend analysis, prediction) used by managers and analysts.

(Diagram 2 M, each tier 4/3 M)

b) XML document schema — DTD and XML Schema.

6 M

An **XML document schema** specifies the structure that XML documents must follow — which elements and attributes may appear, their order, nesting, number of occurrences and (in XML Schema) data types. A document that follows its schema is called **valid**. Two mechanisms: **DTD** and **XML Schema (XSD)**.

1) Document Type Definition (DTD): uses `<!ELEMENT>` and `<!ATTLIST>` declarations. Content: `#PCDATA` (text), sub-elements, `*` (zero or more), `+` (one or more), `?` (optional), `|` (choice). Attribute

types: CDATA, ID, IDREF(S); #REQUIRED / #IMPLIED. Limitation: no data types (everything is text), not written in XML syntax.

```
<!ELEMENT student (roll_no, name, branch, marks+)>
<!ATTLIST student id ID #REQUIRED>
<!ELEMENT roll_no (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT branch (#PCDATA)>
<!ELEMENT marks (#PCDATA)>
```

2) XML Schema (XSD): written in XML itself; supports data types (xs:string, xs:integer, xs:decimal, xs:date), user-defined types, minOccurs / maxOccurs, keys and namespaces.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="student">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="roll_no" type="xs:integer"/>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="branch" type="xs:string"/>
        <xs:element name="marks" type="xs:decimal" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:ID" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Sample document and validation results (checked with libxml2):

```
<student id="S101">
  <roll_no>12</roll_no>
  <name>Asha Patil</name>
  <branch>IF</branch>
  <marks>78</marks>
  <marks>85</marks>
</student>
```

```
<student id="S102">
  <roll_no>twelve</roll_no>
  <name>Rohan</name>
  <branch>AN</branch>
</student>
```

Schema	Document	Result
DTD	valid document	VALID
DTD	invalid document	INVALID — Element student content does not follow the DTD, expecting (roll_no , name , branch , marks+), got (roll_no name branch)
XML Schema	valid document	VALID
XML Schema	invalid document	INVALID — Element 'roll_no': 'twelve' is not a valid value of the atomic type 'xs:integer'.

The second document is rejected because *marks* (required, one or more) is missing; XML Schema also checks the data type of *roll_no*.

(Explanation 2 M, DTD 2 M, XML Schema 2 M)

c) Transaction processing in distributed database; 2PC.

In a distributed database there are two kinds of transactions: **local transactions** (access data only at the site where they start) and **global transactions** (access data at several sites). Each site has two subsystems:

1. **Transaction manager:** manages execution of the transactions (or sub-transactions) that access data at that site — maintains the log for recovery and participates in concurrency control.
2. **Transaction coordinator:** coordinates the transactions started at that site — starts the transaction, breaks it into sub-transactions and sends them to the appropriate sites, and coordinates its termination (commit at all sites or abort at all sites).

Failures special to distributed systems: site failure, loss of messages, communication-link failure, network partition. To keep atomicity, a **commit protocol** is used.

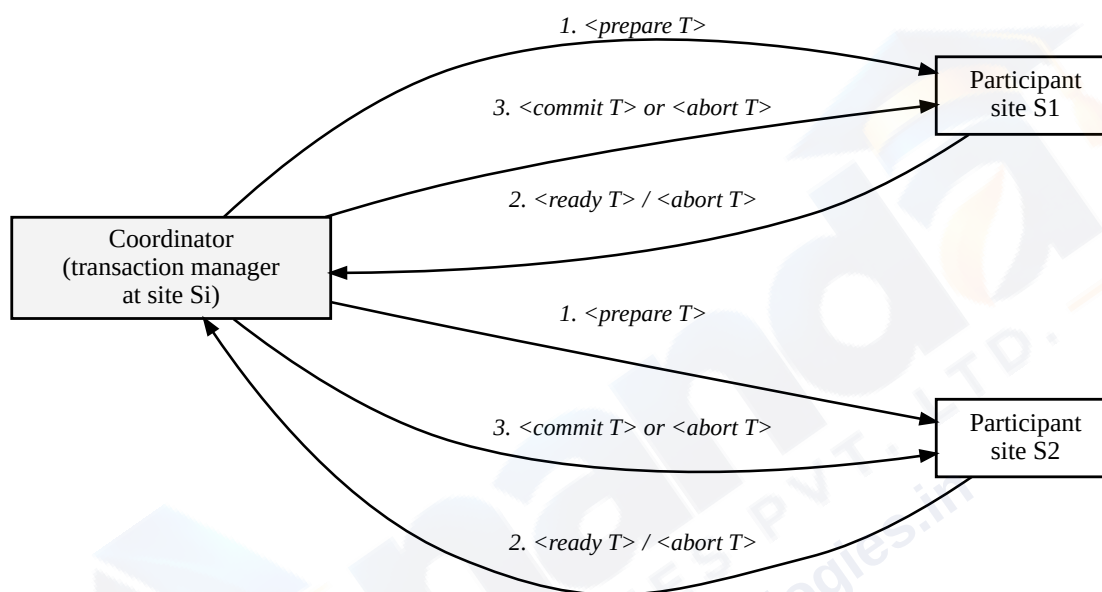


Fig.: Two-phase commit (2PC) protocol in a distributed transaction

Two-phase commit (2PC) — coordinator C_i of transaction T :

1. **Phase 1 (prepare / voting):** C_i writes $\langle \text{prepare } T \rangle$ in its log and sends $\text{prepare } T$ to all sites where T executed. Each site that can commit forces $\langle \text{ready } T \rangle$ to its log and replies $\text{ready } T$; otherwise it writes $\langle \text{no } T \rangle$ and replies $\text{abort } T$.
2. **Phase 2 (decision):** if C_i receives ready from all sites, it writes $\langle \text{commit } T \rangle$ and sends $\text{commit } T$ to all; if any site says abort (or times out), it writes $\langle \text{abort } T \rangle$ and sends $\text{abort } T$. Each participant records the decision and commits or aborts locally.

Thus either all sites commit or all abort. Drawback: **blocking** — if the coordinator fails after sites are ready, they must wait for it to recover (three-phase commit avoids this).

(Transaction / coordinator managers 2 M, diagram 1 M, 2PC phases 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