**Question 3**

**Question 3.1**

Graph databases can be used for these scenarios because it natively models and efficiently traverses deep, variable‑length relationships without expensive JOINs or denormalization.

**Schema:**

* **Node labels & key properties**
  + :User {userId (PK), name, email}
  + :Product {productId (PK), name, price}
  + :Category {categoryId (PK), title}
  + :Order {orderId (PK), orderDate}
  + :Location {locationId (PK), name, type}
* **Relationship types & properties**
  + (u:User)-[:PURCHASED {at: datetime, quantity: int}]->(p:Product)
  + (p:Product)-[:IN\_CATEGORY]->(c:Category)
  + (u1:User)-[:FRIEND\_WITH {since: date}]->(u2:User)
  + (u:User)-[:PLACED\_ORDER]->(o:Order)
  + (o:Order)-[:CONTAINS {quantity: int}]->(p:Product)
  + (u:User)-[:LOCATED\_IN]->(l:Location)
* **Indexes & constraints**
  + Uniqueness constraints on each primary key property (userId, productId, etc.)
  + Indexes on frequently queried fields: :User(email), :Product(name), :Order(orderDate)

**Reason for Selection:**

* **Traversal performance:** index‑free adjacency makes each hop O(1), so multi‑step queries (friends‑of‑friends, recommendation paths) run in time proportional to path length rather than number of joins.
* **Flexibility:** adding new node or relationship types requires no table migrations.
* **Expressiveness:** relationship properties capture context (purchase time, quantity, friendship date) directly on edges, simplifying query patterns.

**Question 3.2**

**SQL Queries  
  
1. Find all friends of user 'alice'**

SELECT u2.user\_id, u2.name

FROM friendships f

JOIN users u2 ON f.friend\_id = u2.user\_id

WHERE f.user\_id = 'alice';

**2. Find friends of 'alice' who purchased product 'X' in the last month**

SELECT u2.user\_id, u2.name, p.at, p.quantity

FROM friendships f

JOIN users u2 ON f.friend\_id = u2.user\_id

JOIN purchases p ON u2.user\_id = p.user\_id

WHERE f.user\_id = 'alice'

AND p.product\_id = 'X'

AND p.at >= DATE\_SUB(NOW(), INTERVAL 1 MONTH);

**3. Recommend top 10 products based on what alice’s friends bought (excluding alice’s own purchases)**

SELECT p.product\_id,

COUNT(\*) AS recommend\_score

FROM friendships f

JOIN purchases p ON f.friend\_id = p.user\_id

LEFT JOIN purchases pa

ON pa.user\_id = 'alice'

AND pa.product\_id = p.product\_id

WHERE f.user\_id = 'alice'

AND pa.user\_id IS NULL

GROUP BY p.product\_id

ORDER BY recommend\_score DESC

LIMIT 10;

**4. List all products in category 'Electronics'**

SELECT pr.product\_id, pr.name, pr.price

FROM products pr

JOIN product\_categories pc ON pr.product\_id = pc.product\_id

JOIN categories c ON pc.category\_id = c.category\_id

WHERE c.title = 'Electronics';

**5. Find all orders that contain product 'X'**

SELECT o.order\_id, o.user\_id, oi.quantity

FROM order\_items oi

JOIN orders o ON oi.order\_id = o.order\_id

WHERE oi.product\_id = 'X';

**NoSQL Queries**

// MongoDB (NoSQL) examples

// Assume collections:

// users: { \_id, name, email, friends: [ObjectId,...] }

// products:{ \_id, name, price, categories: [ObjectId,...] }

// categories:{ \_id, title }

// orders: { \_id, userId, orderDate, items: [ { productId, quantity } ] }

// 1. Find all friends of 'alice'

db.users.aggregate([

{ $match: { \_id: ObjectId("alice") } },

{ $unwind: "$friends" },

{ $lookup: {

from: "users",

localField: "friends",

foreignField: "\_id",

as: "friendInfo"

}

},

{ $unwind: "$friendInfo" },

{ $project: { \_id: 0, friendId: "$friendInfo.\_id", name: "$friendInfo.name" } }

]);

// 2. Friends of alice who purchased product 'X' in last 30 days

db.users.aggregate([

{ $match: { \_id: ObjectId("alice") } },

{ $unwind: "$friends" },

{ $lookup: {

from: "orders",

let: { fid: "$friends" },

pipeline: [

{ $match: {

$expr: { $eq: ["$userId", "$$fid"] },

orderDate: { $gte: new Date(new Date() - 1000\*60\*60\*24\*30) }

}

},

{ $unwind: "$items" },

{ $match: { "items.productId": "X" } },

{ $project: { \_id: 0, friendId: "$$fid", at: "$orderDate", quantity: "$items.quantity" } }

],

as: "recentPurchases"

}

},

{ $unwind: "$recentPurchases" },

{ $replaceRoot: { newRoot: "$recentPurchases" } }

]);

// 3. Recommend top 5 products bought by alice’s friends (excluding alice’s own buys)

db.orders.aggregate([

{ $match: { userId: { $in: db.users.findOne({\_id:ObjectId("alice")}).friends } } },

{ $unwind: "$items" },

{ $group: {

\_id: "$items.productId",

count: { $sum: 1 }

}

},

{ $match: {

\_id: { $nin: db.orders

.find({ userId: "alice" })

.map(o => o.items.map(i => i.productId))

.flat()

}

}

},

{ $sort: { count: -1 } },

{ $limit: 5 }

]);

**Question 3.3**

**Use EXPLAIN/ANALYZE**By examining the execution plan (EXPLAIN, EXPLAIN ANALYZE, or your engine’s profile tool) to identify scans, joins, and bottlenecks.

**Partitioning & Sharding**  
Horizontally partition large tables or shard documents/graphs by date, tenant, or region so queries scan only relevant subsets.

**Avoid SELECT \***  
Fetch only the columns you need to reduce I/O, network transfer, and memory usage.

**Continuous Monitoring & Maintenance**  
Automate alerts on latency or index bloat, drop unused indexes, archive stale data, and benchmark under realistic loads whenever schema, index, or engine versions change.

**Keep Transactions Short & Pool Connections**

* By breaking large operations into smaller batches to avoid long‑running locks.
* Using a connection pool to recycle database connections and reduce overhead.

**Cache Frequent Queries**  
By leveraging in‑memory caches or materialized views for read‑heavy, repeatable queries.

**Use Parameterized/Prepared Statements**  
Let the database cache and reuse execution plans instead of reparsing and planning ad hoc queries.

**Limit Result Sets**  
Use LIMIT/OFFSET (or keyset pagination) to prevent queries from returning excessive rows at once.

**Keep Statistics Up‑to‑Date**  
Regularly running ANALYZE (PostgreSQL), UPDATE STATISTICS (SQL Server), or the equivalent so the optimizer has accurate data distributions.

**Question 3.4**

**First‑class relationships:** Nodes (entities) and edges (relationships) are stored natively, so traversals (e.g. “friends → friends → purchases”) are O(1) per hop rather than costly JOINs.

**Schema flexibility:** Can add new node or relationship types without downtime or table migrations.

**Relationship‑centric queries:** Queries that traverse many hops (recommendations, path‑finding, network analysis) run orders of magnitude faster.

**Rich analytics:** Built‑in graph algorithms (shortest path, centrality, community detection) operate directly on live data model.

**Querying the Data**

**1. Find all direct friends of user “alice”:**

MATCH (a:User {userId:'alice'})-[:FRIEND\_WITH]->(f:User)

RETURN f.userId AS friendId, f.name;

**2. Friends of “alice” who purchased product “X” in the last month:**

MATCH (a:User {userId:'alice'})-[:FRIEND\_WITH]->(f)-[p:PURCHASED]->(prod:Product {productId:'X'})

WHERE p.at >= date().minusMonths(1)

RETURN f.userId AS friendId, p.at AS purchasedAt, p.quantity;  
  
**3. Recommend top 5 products based on friends’ purchases (excluding alice’s own):**MATCH (a:User {userId:'alice'})-[:FRIEND\_WITH]->(f)-[:PURCHASED]->(prod:Product)

WHERE NOT (a)-[:PURCHASED]->(prod)

WITH prod, count(\*) AS score

RETURN prod.productId AS productId, prod.name AS name, score

ORDER BY score DESC

LIMIT 5;  
  
**4. Shortest social path between two users:**MATCH path = shortestPath(

(u1:User {userId:'alice'})-[:FRIEND\_WITH\*]-(u2:User {userId:'bob'})

)

RETURN [n IN nodes(path) | n.userId] AS chain;  
  
**5. Count how many users are in each location:**

MATCH (u:User)-[:LOCATED\_IN]->(loc:Location)

RETURN loc.name AS location, count(u) AS userCount

ORDER BY userCount DESC;

**Question 3.5**

|  |  |  |
| --- | --- | --- |
| Aspect | SQL | NoSQL |
| Schema | Fixed tables & columns; migrations needed for changes. | Schema‑on‑read; documents, key‑values, graphs, etc., with fields that can vary. |
| Query Language | Standard SQL (SELECT, JOIN, etc.). | Database‑specific (e.g. JSON APIs, CQL, Cypher). |
| Transactions | Strong ACID across multiple tables. | Often eventual consistency: some offer tunable or limited ACID. |
| Scalability | Scale up (bigger server); sharding is complex. | Scale out (add servers) with built‑in sharding. |
| Relationships | Native JOINs & foreign keys. | Typically, denormalize or handle app logic (except graph DBs). |