**Question 1 – MongoDB Modelling**

1. *Explain the options for modelling data in MongoDB. Include modelling patterns.*

**Modelling Options for Relationships**

Data that’s read together should be stored together. Data that’s not read together should be stored separately. Everything is a trade-off driven by the workload.

1-1 Relationships

1. Embed one object in the other

* Store the entire related document inline when the two are always accessed as a pair.
  + (e.g. every User has exactly one Profile).
* **Pros:** Zero lookups, atomic updates (write operation that touches one document will either **fully succeed** or **fully fail**; never “half applied” change on that document).
* **Cons:** Slight duplication if the embedded piece is also queried on its own (rare case).

1-N Relationships

1. **Child → Parent Reference**

* Each stores the parent’s **\_id**.
* **Use if:** The child grows unbounded or is updated independently from its parent.

1. **Parent → Child References**

* Parent keeps an array of child **\_id**s.
* **Use if:** we need fast lookups of a parent’s children but want to cap document growth.

1. **Embed Parent in Child**

* Copy parent fields into each child.
* **Use if:** we read child data with a snapshot of parent fields and parent updates are infrequent.

1. **Embed Children in Parent**

* Parent contains an array of sub-documents.
* **Use if:** The number of children is small, or you only ever query the latest N (e.g. only show the 10 most recent comments of a blog post) (otherwise hit 16 MB limit)

N-N Relationships

1. **Array of foreign-key references**

* Each side keeps an array of the other side’s **\_id**s.
* **Use if:** Sets are bounded, or we query bi-directionally often.

1. **Embed related objects**

* Duplicate the full sub-document on each side.
* **Use if:** Each side’s related set is very small and mostly read together.

**Key Trade-Offs**

* **Embed** → fewer round-trips (request → response), faster reads, atomic updates; but risks duplication, document growth, 16 MB cap.
* **Reference** → bounded docs, normalized updates; but requires $lookup or multiple queries.
* Always match your choice to **query patterns** (frequency, cardinality, and atomicity needs).

**Design Patterns**

Embedding

* Inline subordinated data in the parent document to avoid joins and optimize co-reads.
* **Pros**: no joins, fast reads.
* **Cons**: duplication, larger docs, 16 MB limit.

Attribute Pattern (when field sets vary widely)

* Replace many parallel fields with an array of {key, value} documents to handle dynamic attribute sets.

Computed Pattern (for read-heavy stats)

* Maintain derived or aggregate values (e.g. running totals, averages) at write time instead of recalculating on reads

Extended Reference Pattern

* In addition to storing a foreign **\_id**, duplicate a small subset of the referenced document’s fields to speed lookups.

Subset Pattern

* Embed only a “hot” subset of related elements (e.g. most recent N); archive or link out to the full set elsewhere.

Polymorphic Pattern

* Store multiple sub-types in one collection by using discriminators, each with its own embedded sub-document for subtype-specific fields.

Recursive Relationships

* Model self-referencing links, e.g. each **Employee** has a **manager\_id** pointing at another **Employee** document.

Tree Pattern

* Precompute and store the full ancestry chain (an array of all ancestor IDs or names) in each node to enable fast tree traversals without recursive queries.

1. *Compare database modelling in SQL, document-based, and graph databases using your course assignments.*

**Comparing SQL vs. Document vs. Graph**

**SQL (Assignment 1)**

Schema & Entities

* Rigid tables (**book**, **author**, **category**, **order**, **order\_details**, plus join-tables for N–M) **with parent\_category\_id** FK for hierarchies.

1-N & N-M modeling

* 1–N via FK columns; N–M via explicit join-tables (e.g. **book\_genre\_pair**).

Hierarchies

* Self-referencing FK + recursive CTE (**WITH RECURSIVE**) on **category.parent\_category\_id**.

Performance & Trade-offs

* Joins are powerful but can become costly with many tables and deep hierarchies.

Evolution & Flexibility

* Schema changes require ALTER TABLE, migrations, downtime.

**Document (Assignment 2)**

Schema & Entities

* Flexible collections (**books**, **customers**, **orders**, etc.). Modelled the same entities as BSON documents.

1-N & N-M modeling

* 1–N by embedding small arrays (e.g. **orders.items**), referencing others by ObjectId. N–M via arrays of ObjectIds or simple values.

Hierarchies

* Parent–child via ObjectId refs plus occasional **$graphLookup** for deep traversals.

Performance & Trade-offs

* Reads can be faster (embedded docs), but updates to nested data can be more complex. Applied the **Extended Reference** and **Subset** patterns to balance this.

Evolution & Flexibility

* Schemaless by default—can enforce via JSON Schema if desired. Easy to add new fields or sub-documents.

**Graph (Assignment 3)**

Schema & Entities

* Nodes for every entity **(:Book, :Author, :Order, :Category**, etc.) with labels and properties.

1-N & N-M modeling

* Every relationship is a first-class edge (e.g. **(:Order)-[:CONTAINS]->(:BookCopy), [:BELONGS\_TO\_GENRE]**).

Hierarchies

* Native variable-length paths: **MATCH (c)-[:SUBCATEGORY\_OF\*0..]->(sub) RETURN sub**.

Performance & Trade-offs

* Traversals are O(1) per hop, ideal for highly connected queries—but the schema is fully connected, which can make bulk analytics harder.

Evolution & Flexibility

* Fully schemaless; adding new node/relationship types is trivial, though tooling is less mature.