**Question 2 – MongoDB JSON Schema: Elements & Example**

1. *Explain the elements of the JSON Schema in MongoDB and exemplify its used.*

**Schema Elements**

**bsonType**

* Specifies the expected BSON type (**“object”, ”string”, ”int”, “decimal”, “array”,** etc.)

**required**

* An array of field names that must be present

**properties**

* Defines sub-schemas for individual fields:
  + **bsonType**
  + Numeric constants (**minimum, maximum**)
  + String constraints (**pattern**)
  + Enumerations (**enum**)
  + Nested **properties** or **items** (for arrays)

**additionalPropperties** (optional)

* Whether fields **not** under **properties** are allowed

**Items**

* For arrays: defines the schema each element must match

**Example from Assignment 2**

* Enforced that every **books** document has a top-level **price: decimal** and an ISBN string, and that **authors** is an array of **ObjectId** references

1. *Compare realizing database models in SQL, document-based, and graph databases using your course assignments. Include mapping to a logical model where appropriate.*

**Comparing SQL vs. Document-Based vs. Graph Modelling**

**SQL (Assignment 1)**

Conceptual → Logical

* **ER Model** → Normalize → Tables + FKs + Join tables

1-N & N-M modeling

* 1–N via FK in child table
* N–M via explicit join-tables

Hierarchies

* Recursive FK (general example **Category.parent\_id**) CTEs (Common Table Expression temporary result set you can reference within a single SQL statement) for traversal

Polymorphic/Enum

* Single table and discriminator column or class/table inheritance

Invalid Data Guardrails

* RDBMS (Relational Database Management System) constraints (**NOT NULL, CHECK, UNIQUE**)

Joins vs. Lookups

* **JOIN** operations across tables

Performance & Trade-offs

* Strict normalization avoids redundancy, but many joins can be slow

Evolution & Flexibility

* Schema migrations required for new columns or relations, downtime.

**Document (Assignment 2)**

Conceptual → Logical

* **ER Model** → Denormalize rules (“no joins”) → JSON collections with embed vs. ref decisions

1-N & N-M modeling

* 1–N via embedding small sub-docs (e.g. **orders.order\_details**)
* N–M via arrays of ObjectIds (e.g. **book.authors**)

Hierarchies

* Recursive via **parentId** refs & **$graphLookup** for deep queries

Polymorphic/Enum

* **Polymorphic Pattern** in **book\_copies.type** enum (**Hardcover/Paperback/E-book**)

Invalid Data Guardrails

* **JSON Schema** validators on collections to require **email** on customers, **price** on books, etc.

Joins vs. Lookups

* **$lookup/$graphLookup** in aggregation pipeline to “join” collections

Performance & Trade-offs

* Embedding boosts reading performance for common patterns
* referencing avoids large doc size
* denormalized **books.price** (Extended Reference) balances read vs. update cost

Evolution & Flexibility

* Schemaless by default
* Schema enforcement optional
* easy to add fields (**rating, page\_count**) without migrations

**Graph (Assignment 3)**

Conceptual → Logical

* **ER/Domain Model** → Nodes & Relationships → Property Graph schema

1-N & N-M modeling

* 1–N via singe OUT edge
* N–M via multiple edges (e.g. **(Book)-[:WRITTEN\_BY]->(Author)**)

Hierarchies

* Native: use variable-length incoming edges (**[:SUBCATEGORY\_OF\*]**) in Cypher

Polymorphic/Enum

* Node labels or relationship types (e.g. **(Copy:Hardcover), (Copy:EBook)**)

Invalid Data Guardrails

* Schema is optional
* Validation at application layer or via GraphQL type checks

Joins vs. Lookups

* Traversal via native graph pattern matching (**MATCH (c)-[:PLACED]->(o)**)

Performance & Trade-offs

* Most queries are local traversals
* Graph indexes on node properties
* Deep joins are very cheap

Evolution & Flexibility

* Flexible: add new node labels, relationship types
* Evolve GraphQL SDL (Schema Definition Language) or Cypher procedures

**Mapping to Logical Model**

* **SQL**: defined **Books(id, isbn, title, price)** + **Authors(id, name)** + join tables for many-to-many.
* **Document**: You mapped **Book → books** collection, **Customer → customers** with embedded address, **Order → orders** with embedded **order\_details**.
* **Graph**: You mapped entities to nodes **(:Book, :Customer, :Order**), and relationships **(:WRITTEN\_BY, :PLACED, :CONTAINS**) in Neo4j via Cypher and GraphQL directives.