**Data Modeling**

**1. Embedding and Referencing**

MongoDB allows two main ways to represent relationships between data:

* **Embedding (Denormalization):**
  + Store related data **in the same document**.
  + Example:
  + {
  + "name": "John",
  + "address": {
  + "street": "123 Main St",
  + "city": "Chennai"
  + }
  + }
  + **Pros:** Faster reads, fewer joins.
  + **Cons:** Data duplication, harder updates if reused elsewhere.
* **Referencing (Normalization):**
  + Store related data **in separate collections** and use references.
  + Example:
  + {
  + "name": "John",
  + "address\_id": ObjectId("abc123")
  + }
  + {
  + "\_id": "abc123",
  + "street": "123 Main St",
  + "city": "Chennai"
  + }
  + **Pros:** Less duplication, easier to update.
  + **Cons:** Requires additional queries ($lookup) to join data.

**2. Schema Design Considerations**

MongoDB is schema-flexible, but good design improves performance:

* **Access patterns:** Design collections based on how data will be queried, not just relationships.
* **Document size:** Keep documents below **16 MB**.
* **Cardinality:**
  + Use **embedding** for one-to-few relationships.
  + Use **referencing** for one-to-many or many-to-many.
* **Atomic operations:** Updates on single documents are atomic—embed fields that must change together.
* **Indexing:** Plan indexes for frequently queried fields to improve performance.

**3. Data Normalization vs. Denormalization**

* **Normalization:**
  + Split data into multiple collections to reduce redundancy.
  + Similar to relational databases.
  + Example: separate customers and orders collections.
  + Reduces data redundancy.
  + Slower reads, requires $lookup.
* **Denormalization:**
  + Combine related data into a single collection.
  + Faster reads, fewer joins.
  + More storage, harder updates.

**Rule of thumb:**

* Normalize for **write-heavy applications** (frequent updates).
* Denormalize for **read-heavy applications** (frequent queries).

**Querying**

**4. Simple Queries**

**Document –name,age ,department**

Used to retrieve data using the find() method.

**db.students.find({ age: 21 });**

Operators:

* { field: value } — Match exact value.
* { field: { $gt: value } } — Greater than.
* { field: { $in: [val1, val2] } } — Match any of several values.

Some user with the same age also

Example:

db.products.find({ price: { $lt: 1000 }, category: "Electronics" });

**5. Advanced Query Operators**

MongoDB supports rich query operators:

* **Comparison:** $eq, $ne, $gt, $lt, $gte, $lte
* **Logical:** $and, $or, $not, $nor
* **Element:** $exists, $type
* **Array:** $all, $elemMatch, $size
* **Evaluation:** $regex, $expr

Example:

db.students.find({ $or: [{ marks: { $gt: 90 } }, { grade: "A" }] });

**db.employees.find({ name: { $regex: /^R/, $options: 'i' } }) –name with capital ‘R’ as well as ‘R’**

**db.employees.find({ bonus: { $exists: true } }) ---emp1 ,emp3**

**db.employees.find({ bonus: { $exists: false } }) ---emp2**

**--empid empname salary bonus ---emp1**

**empid empname salary ---emp2**

**empid empname salary bonus ---emp3**

**6. Projections**

**Select empid,empname,salary from employee;**

**Find()---all the entire**

Used to control which fields appear in the output.

db.students.find({}, { name: 1, age: 1, \_id: 0 });

* 1 → include field
* 0 → exclude field
* \_id is included by default unless explicitly excluded.

**db.employees.find({}, { name: 1, salary: 1 })**

**db.employees.find({}, { salary: 0 })**

**projection +condition**

**db.employees.find(**

**{ department: "IT" },**

**{ \_id: 0, name: 1, city: 1 }**

**)**

**db.students.find({}, { \_id: 0, name: 1, "marks.math": 1 })**

**db.posts.insertOne({**

**title: "MongoDB Basics",**

**comments: ["Good", "Very Helpful", "Excellent", "Nice Post"]**

**})**

**db.posts.find({}, { comments: { $slice: 2 } })**

**7. Sort**

Sorts query results in ascending or descending order.

db.products.find().sort({ price: 1 }); // ascending

db.products.find().sort({ price: -1 }); // descending

Multiple fields can be sorted:

db.students.find().sort({ grade: 1, age: -1 });

**8. Skip**

Skips a specified number of documents in the result set (used for pagination).

50 records ----pagination –per page 10 records ---

5 pages 10 records in each page

db.students.find().skip(5);

— Skips the first 5 documents.

db.students.find().skip(2).limit(2)

**9. Limit**

Limits the number of documents returned.

db.students.find().limit(10);

Often combined with skip() for pagination:

db.students.find().skip(10).limit(5);

**Example Pagination Query**

db.products.find()

.sort({ price: 1 })

.skip(20)

.limit(10);

→ Retrieves the 3rd page of results (10 per page, sorted by price).

An **index** in MongoDB improves the **speed of data retrieval**.  
Without an index, MongoDB must perform a **collection scan** — checking every document to find matches.

Indexes work like a **book’s table of contents** — they help MongoDB quickly locate the data instead of scanning every page (document).

Id name email salary

db.employees.createIndex({ email: 1 })

db.employees.find({ name: "Ravi" }) -COLLSCAN (collection scan -without index)

IXSCAN ---if I apply index in my collection ---

db.employees.find({ name: "Ravi" }).explain("executionStats") ---output

Unique index

Single field index

db.employees.createIndex({ emp\_id: 1 }, { unique: true })

Compound index

Db.employees.createIndex({department:1,salary:1})

Department+salary

Multikey index

db.products.insertMany([

{ name: "Laptop", tags: ["electronics", "tech"] },

{ name: "Shirt", tags: ["clothing", "fashion"] }

])

db.products.createIndex({ tags: 1 })

db.products.find({tags:tech})

Text index

Content -field ---

Db.articles.createIndex({content:”text”})

Db.articles.find({$text:{$search :”database performance”}})

All the documents where content contains the word database or performance

Hashed Index ----hash based lookup

Db.users.createIndex({user\_id:”hashed”})

Shared cluster ---even data distribution

Geospatial index

Location based queries

Db.user.createIndex({location:”2dsphere”})

db.places.find({

location: {

$near: {

$geometry: { type: "Point", coordinates: [77.5946, 12.9716] },

$maxDistance: 5000

}

}

})

Return all locations with 5KM

Index Management and monitoring

Db.employees.getIndexes() ---list all the indexesin the collection

Db.employees.dropIndex({name:1})

Db.employees.dropIndexes() ----delete all the index in ur collection

Db.employees.find({salary:{$gt:50000}}.explain(“executionStats”)

Stage :IXSCAN ---using your index

Stage : COLLSCAN -----MongoDB will scanning the whole collection

ORM ---hibernate

Inserting into database in the form of object

Retrieve in the form of o

ODM ---Object Document Mapping