MongoDB Mastery

Content

- Relationship & Advanced Data Modeling
- Aggregation Pipeline
- Indexing for Performance
- Database Transactions

Relationship Modeling

Relationship Modeling

- **What Is Relationship Modeling?**
  - In relational databases, relationships are defined via foreign keys
  - In MongoDB, relationships are created using document structure
    → Embedded documents or referenced documents
  - MongoDB is flexible — choose what's best for the read/write pattern, data size, and access frequency.

Relationship Modeling

mongoDB

Types of Relationships

Type	Example	Strategy
One-to-One	User ↔ Profile	Embed or Reference
One-to-Many	BlogPost ↔ Comments	Embed or Reference
Many-to-Many	Users ↔ Roles / Products ↔ Tags	Reference only

Relationship Modeling

- How to implement relationships in mongoose? :
  - There are two ways of implementing relationships:
    - Embedding:
      - storing related data inside the same document, as a sub-document or an array of sub-documents.
    - Referencing:
      - storing related data in separate documents, and connecting them using ObjectIdreferences.

Embedding

- How does embedding documents work :
    - Instead of creating separate collections (tables in a relational database) for relatedentities and linking them with references (like foreign keys), you embed the "child"document(s) directly within the "parent" document.

```
{
  "_id": "u1",
  "name": "Ali",
  "email": "ali@example.com",
  "profile": {
    "bio": "Backend developer",
    "avatar": "avatar.jpg"
  }
}
```

Embedding

- **Benefits of Embedding:**
  - **Faster reads:** One document = one read from DB
  - **Simpler data model:** Easy to understand and access
  - **Atomic updates:** Updating the full document is safe
  - **Less joins/code:** No .populate() or $lookup needed

Embedding

- When to Embed documents:
  - When the embedded data is only used with the parent
  - When the data is not reused elsewhere
  - When you don't need to query the embedded data directly
  - When the embedded array is not very large (keep under ~100 subdocs ideally)
- Avoid Embedding When:
  - The sub-document grows indefinitely (e.g. chat messages)
  - You need to query/update sub-documents independently
  - The sub-document is shared across multiple parents

Referencing

- Referencing **means storing related data in separate** documents**, and connecting them using ObjectId references.**
- **Think of it like creating a "foreign key" in relational databases.**

```
{
  "_id": "u1",
  "name": "Ali",
  "email": "ali@example.com",
  "profile": "p1"  // Reference to Profile._id
}

```

Referencing

- When to Use Referencing
  - Use referencing when:
    - The related data grows large (e.g., comments, orders)
    - You need to query/update sub-data independently
    - The related data is shared across multiple documents
    - You want to normalize your data
  - Avoid referencing when:
    - The data is always used together
    - You want fast, single-document reads

One-to-One Relationships

mongoDB

- **What Is a One-to-One Relationship?**
  - A One-to-One (1:1) relationship means that:
  - One document in a collection is linked to exactly one document in another collection.
  - For example:
    - **A User has one Profile**
    - A Car has one Engine
    - **A Company has one Address**
    - **So, for each User → there is only one Profile.**

One-to-One Relationships

- **When Should You Use a One-to-One Relationship?**

  - Use a one-to-one relationship when:
  - You want to separate optional or sensitive information
  - The second document might grow large
  - You want to modularize your data (store it in its own collection)

Embedding

How to implement 1:1 embedding with mongoose :

```
1  const ProfileSchema = new mongoose.Schema({
2    bio: String,
3    avatar: String
4  });
5
6  const UserSchema = new mongoose.Schema({
7    name: String,
8    email: String,
9    profile: ProfileSchema
10 });
11
12 const User = mongoose.model('User', UserSchema);
13 export default User;
```

mongoDB

Embedding

Then you can create users with profiles using:

```
await User.create({
  name: 'Ali',
  email: 'ali@example.com',
  profile: {
    bio: 'Node.js backend developer',
    avatar: 'ali.jpg'
  }
});

const user = await User.findOne({ email: 'ali@example.com' });
console.log(user.profile.bio);

```

Referencing

How to implement 1:1 referencing with mongoose :

```
1  const ProfileSchema = new mongoose.Schema({
2    bio: String,
3    avatar: String,
4    social: {
5      twitter: String,
6      github: String
7    }
8  });
9  const UserSchema = new mongoose.Schema({
10   name: String,
11   email: String,
12   profile: {
13     type: mongoose.Schema.Types.ObjectId, ref: 'Profile'
14   }
15 });
```

Referencing

Then you can create a user profile and link it to the user using:

```
const profile = await Profile.create({
  bio: 'I love Node.js!',
  avatar: 'avatar.png',
  social: {
    twitter: '@mamdouh',
    github: 'mamdouhdev'
  }
});

const user = await User.create({
  name: 'Mamdouh',
  email: 'mamdouh@example.com',
  profile: profile._id
});
```

mongoDB

Referencing

Now when we want to retrieve a user with their profile, we use the .populate() method to join tables:

```
console.log(await User.findById(user._id).populate('profile'));
{
  "_id": "...",
  "name": "Mamdouh",
  "email": "mamdouh@example.com",
  "profile": {
    "_id": "...",
    "bio": "I love Node.js!",
    "avatar": "avatar.png",
    "social": {
      "twitter": "@mamdouh",
      "github": "mamdouhdev"
    }
  }
}
```

mongoDB

One-to-Many Relationships

- **What Is a One-to-Many Relationship?**
  - A One-to-Many relationship means that one document is related to many other documents.
  - Real-world examples:
    - One user has many orders
    - One post has many comments
    - One category has many products
    - So, 1 "parent" → many "children"

One-to-Many Relationships

- **When Should You Use a One-to-Many Relationship?**
  - Use it when:
    - You have a clear “parent-child” relationship
    - The "many" side is logically grouped with the "one"
    - You want to store related but independent documents

Embedding

How to implement 1:M embedding with mongoose :

```
1  const CommentSchema = new mongoose.Schema({
2    text: String,
3    user: String
4  });
5
6  const PostSchema = new mongoose.Schema({
7    title: String,
8    content: String,
9    comments: [CommentSchema]
10 });
11
12 const Post = mongoose.model('Post', PostSchema);
```

mongoDB

Embedding

Then you can create comments for a post using:

```
// Find the post
const post = await Post.findById(post._id);

// Add new comment
post.comments.push({
  text: 'Very helpful!',
  user: 'Mamdouh'
});

// Save the updated post
await post.save();
```

mongoDB

Referencing

How to implement 1:M referencing with mongoose :

```
1  const PostSchema = new mongoose.Schema({
2    title: String,
3    content: String
4  });
5
6  const Post = mongoose.model('Post', PostSchema);
7
8  const CommentSchema = new mongoose.Schema({
9    text: String,
10   user: String,
11   postId: { type: mongoose.Schema.Types.ObjectId, ref: 'Post' }
12 });
13
14 const Comment = mongoose.model('Comment', CommentSchema);
```

Referencing

Then you can create comments for a post using:

```
// 1. Create a post
const post = await Post.create({
  title: 'Mongo Relationships',
  content: 'Let's learn about 1:M referencing!'
});

// 2. Create comments for that post (referencing it)
const comment1 = await Comment.create({
  text: 'This is very helpful!',
  user: 'Sara',
  postId: post._id
});
```

Referencing

Finally, you could retrieve all post comments using:

```
const { text } = require("express");

const comments = await Comment.find({ postId: post._id });
console.log(comments);

// Output:

[
    {
        _id: '60c72b2f9b1e8c001c8e4d5a',
        postId: '60c72b2f9b1e8c001c8e4d5a',
        user: 'Sara',
        text: 'this is very helpful'
    }
    ....
]
```

mongoDB

Many-to-Many Relationships

- **What Is a Many-to-Many Relationship?**
  - A Many-to-Many relationship means:
    - A book can belong to many categories.
    - A category can contain many books.
    - This is different from One-to-Many (1:M) where only one side holds multiple references.

Many-to-Many Relationships

- **When Should You Use Many-to-Many?**
  - You should use M:M relationships when:
    - You need cross-linking between entities.
    - Both sides need to be aware of each other.
    - The relationship is flexible and dynamic (a book may be in several genres, and genres may change over time).
  - Real-world Examples:
    - Books and Categories
    - Students and Courses
    - Users and Roles
    - Posts and Tags

Referencing

- You can’t embed schemas recursively because it leads to a circular dependency
- We can only use referencing if we want true, bi-directional M:M relationship:

```
1  const CategorySchema = new mongoose.Schema({
2    name: String,
3    books: [{
4      type: mongoose.Schema.Types.ObjectId,
5      ref: 'Book'
6    }]
7  });
8
9  const BookSchema = new mongoose.Schema({
10   title: String,
11   author: String,
12   categories: [{
13     type: mongoose.Schema.Types.ObjectId,
14     ref: 'Category'
15   }]
16 });
```

mongoDB

Referencing

Then you can create books and categories bi-directionally using:

```
1  const fiction = await Category.create({ name: 'Fiction' });
2  const history = await Category.create({ name: 'History' });
3
4  const book = await Book.create({
5    title: 'World Stories',
6    author: 'John Smith',
7    categories: [fiction._id, history._id]
8  });
```

Referencing

But because MongoDB is a non-relational database, we have to update our categories to include the new books too :

```
1  const fiction = await Category.create({ name: 'Fiction' });
2  const history = await Category.create({ name: 'History' });
3
4  const book = await Book.create({
5    title: 'World Stories',
6    author: 'John Smith',
7    categories: [fiction._id, history._id]
8  });
9
10 fiction.books.push(book._id);
11 history.books.push(book._id);
12 await fiction.save();
13 await history.save();
```

mongoDB

Populating Responses

- **What Is .populate()?**
  - **.populate()** is a Mongoose method used to automatically replace referenced ObjectIds in a document with the actual documents from the referenced collection.
  - This is a crucial feature for handling relationships between data in MongoDB, where relationships are often represented by storing the _id of a related document.

populate

- How it works:

    1. Referencing: In your Mongoose schema, you define a field that references another model using type: Schema.Types.ObjectId and the ref option, which specifies the name of the referenced model.

```
const storySchema = new mongoose.Schema({
    author: { type: mongoose.Schema.Types.ObjectId, ref: 'Person' },
    title: String,
});
```

populate

- How it works:

  2. Population: When you query for documents from the first collection, you can use .populate() on the query to replace the ObjectId in the referenced field with the actual document data from the related collection.

```
1  Story.findOne({ title: 'My Awesome Story' })
2      .populate('author')
3      .exec((err, story) => {
4          if (err) return handleError(err);
5          console.log('The author is %s', story.author.name);
6      });
```

mongoDB

Populating Responses

- **Key features and uses:**
  - **Simplifies data retrieval:**
    - Eliminates the need for manual lookups or multiple queries to fetch related data.
  - **Populate single or multiple documents:**
    - Can be used to populate a single field or an array of referenced documents.
  - **Populate multiple paths:**
    - You can chain .populate() calls to populate multiple fields in a single query.

Populating Responses

- **Key features and uses:**
  - **Select specific fields:**
    - You can specify which fields from the populated document you want to include using the select option.
  - **Query conditions within population:**
    - You can apply conditions and options to the populated documents themselves, such as limiting the number of sub-documents populated from an array.

Populating Responses

Parameters

Parameter	Type	Description
path	String	Field to populate (must be a reference)
select	String or Object	Fields to include/exclude ('name email' or '-password')
model	String	Explicit model name (optional)
match	Object	Filter criteria for populated documents
options	Object	Pagination/sorting: { limit, skip, sort }

Populating Responses

- **Examples:**
  - Basic populate: **Post.find().populate('author');**
  - Select specific fields: **Post.find().populate('author', 'name email -_id');**
  - Filtering referenced documents:

```
Post.find().populate({ path: 'author', match: { name: /john/i }});
```

  - **Sorting and limiting populated documents:**

```
Post.find().populate({path: 'comments', options: { sort: { createdAt: -1 },
limit: 5 }});
```

Aggregation Pipeline

Aggregation Pipeline

- **What Is the Aggregation Pipeline?**
  - The Aggregation Pipeline in MongoDB is a framework to process data records, transform them, and return computed results — all within the database, not in your app.
  - Each stage performs a specific operation, like filtering, grouping, or sorting, and the output of one stage becomes the input for the next.
  - This framework provides a powerful and flexible way to work with data in MongoDB.

Aggregation Pipeline

- **Core Concepts:**
  - **Stages:**
    - The pipeline is composed of multiple stages, each performing a specific operation on the documents.
  - **Data Flow:**
    - Documents enter the pipeline and are transformed by each stage in sequence.
  - **Result:**
    - The final output of the pipeline is the result of all the operations performed on the input documents.

Aggregation Stages

- $match: Filters documents based on specified criteria.
- $group: Groups documents based on a key and performs calculations on the groups.
- $sort: Sorts documents based on specified fields.
- $project: Reshapes the structure of documents by including or excluding fields, or adding new fields.
- $unwind: Deconstructs an array field from an input document into multiple documents, one for each element in the array.

Aggregation Stages

- $limit: Limits the number of documents passed to the next stage.
- $skip: Skips a specified number of documents before passing the remainingones to the next stage.
- $lookup: Performs a left outer join with another collection.
- $addFields: Adds new fields to documents.
- $out: Writes the results of the aggregation to a new collection.

$sort

- What It Does:
  - $sort sorts the documents that pass through the aggregation pipeline based on specified fields, either in ascending (1) or descending (-1) order.
  - Equivalent to **ORDER BY** price **DESC** in **SQL**

```
{ name: "Book A", price: 20 }
{ name: "Book B", price: 10 }
{ name: "Book C", price: 30 }

db.products.aggregate([
  { $sort: { price: -1 } }
]);
```

Output

```
// output
{ name: "Book C", price: 30 }
{ name: "Book A", price: 20 }
{ name: "Book B", price: 10 }
```

Aggregation Stages

- Tips & Notes:
  - **$sort** must appear after **$match**, **$project**, or **$group** if you're using them. sort operates on the current pipeline result.
  - Sorting large datasets may impact performance if no proper index exists.
  - Can sort by multiple fields: **{ price: 1, name: -1 }.**

$group

- $group groups documents by a specified key and lets you perform aggregate operations like:
  - sum, avg, min, max
  - push, addToSet
  - first, last, etc.
- Each output document represents one group.

```
// Sample data for the sales collection
{ item: "Book", quantity: 5, price: 10 }
{ item: "Pen", quantity: 10, price: 2 }
{ item: "Book", quantity: 2, price: 10 }

db.sales.aggregate([
  {
    $group: {
      _id: "$item",
      totalQuantity: { $sum: "$quantity" }
    }
  }
]);
```

Output

```
// Output:
{ _id: "Book", totalQuantity: 7 }
{ _id: "Pen", totalQuantity: 10 }
```

Aggregation Stages

Grouping Common Accumulators

Accumulator	What It Does
$sum	Total sum of values
$avg	Average value
$min, $max	Minimum or maximum value
$first, $last	First/last value by sort order
$push	Adds values to array (duplicates ok)
$addToSet	Adds values to array (unique only)

Aggregation Stages

- **Tips & Notes:**
  - The _id is required, it defines how you group.
    - Use null if you want one group for everything.
  - You can group by multiple fields using an object:

    _id: { item: "$item", category: "$category" }
  - Use **$project** before **$group** if you want to reshape documents first.
  - SQL Equivalent: **SELECT** item, **SUM**(quantity)
    FROM sales
    GROUP BY item;

$project

- $project allows you to reshape a document by:
  - Including or excluding specific fields.
  - Renaming fields.
  - Adding new fields by computing values.
  - Control document structure for the next pipeline stage.
  - SQL Equivalent of: **SELECT** salary * 1.1 **AS** bonusSalary **FROM** employees

```
{
  "_id": ObjectId("..."),
  "firstName": "John",
  "lastName": "Doe",
  "age": 60,
  "salary": 5000
}
db.employees.aggregate([
  {
    $project: {
      _id: 0,
      name: 1,
      age: 1,
      isSenior: { $gte: ["$age", 50] },
      fullName: { $concat: ["$firstName", " ", "$lastName"] },
      bonusSalary: { $multiply: ["$salary", 1.1] }
    }
```

Output

```
{
  "name": "John",
  "age": 60,
  "isSenior": true,
  "fullName": "John Doe",
  "bonusSalary": 5500
}
```

Aggregation Stages

Common $project Operators

Operator	Description
1 / 0	Include (1) or exclude (0) a field
$add, $subtract, $multiply, $divide	Math operations
$concat	Join strings "concatenate"
$toUpper, $toLower	Change string case
$gte, $lte, $eq, $cond	Logical comparisons and conditions
$arrayElemAt	Get specific index from array
$dateToString	Format date fields

$unwind

- Takes an array field from input documents and outputs a separate document for each element in the array.

```
1  const employees = [
2    { name: "Alice", skills: ["JavaScript", "React"]
3    { name: "David", department: "Design" },
4  ];
5
6  Employee.aggregate([
7    {
8      $unwind: {
9        path: "$skills",
10       includeArrayIndex: "skillIndex",
11       preserveNullAndEmptyArrays: false
12     }
13   }
14 ]);
```

Output

```
// output
[
  {
    _id: new ObjectId('687b0dae9aba05bc108b4e8e'),
    name: 'Alice',
    skills: 'JavaScript',
    department: 'Engineering',
    __v: 0,
    skillIndex: 0
  },
  {
    _id: new ObjectId('687b0dae9aba05bc108b4e8e'),
    name: 'Alice',
    skills: 'React',
    department: 'Engineering',
    __v: 0,
    skillIndex: 1
  },
]
```

$limit

- The **$limit** stage limits the number of documents that pass through the aggregation pipeline to a specified number.
- Usually comes after sorting. Ex. Limit for the top five most expensive items

```
1  await Product.insertMany([
2    { name: 'Laptop', price: 1200 },
3    { name: 'Smartphone', price: 800 },
4    { name: 'Monitor', price: 300 },
5    { name: 'Keyboard', price: 100 },
6    { name: 'Tablet', price: 400 },
7    { name: 'Smartwatch', price: 250 },
8    { name: 'Desk Lamp', price: 60 },
9  ]);
10
11 const topProducts = await Product.aggregate([
12   { $sort: { price: -1 } },
13   { $limit: 5 }
14 ]);
```

Output

```
Top 5 Most Expensive Products:
[
  {
    _id: new ObjectId('687c45df56d122b046911d6d'),
    name: 'Laptop',
    price: 1200,
    __v: 0
  },
  {
    _id: new ObjectId('687c45df56d122b046911d6e'),
    name: 'Smartphone',
    price: 800,
    __v: 0
  },
  {
    _id: new ObjectId('687c45df56d122b046911d71'),
    name: 'Tablet',
    price: 400,
    __v: 0
  },
  {
    _id: new ObjectId('687c45df56d122b046911d6f'),
    name: 'Monitor',
    price: 300,
    __v: 0
  },
  {
    _id: new ObjectId('687c45df56d122b046911d72'),
    name: 'Smartwatch',
    price: 250,
    __v: 0
  }
]
```

$skip

- $skip is a stage in the MongoDB Aggregation Pipeline that skips a specified number of documents from the input and passes the remaining documents to the next stage.
- Think of it like the SQL **OFFSET** keyword or **LIMIT** x **OFFSET** y

```
1  await Product.insertMany([
2    { name: 'Laptop', price: 1200 },
3    { name: 'Phone', price: 800 },
4    { name: 'Tablet', price: 600 },
5    { name: 'Monitor', price: 300 },
6    { name: 'Keyboard', price: 100 },
7    { name: 'Mouse', price: 50 },
8    { name: 'Printer', price: 400 },
9    { name: 'Speaker', price: 200 },
10   { name: 'Webcam', price: 150 },
11   { name: 'Microphone', price: 250 }
12 ]);
13
14 const result = await Product.aggregate([
15   { $sort: { price: -1 } },
16   { $skip: 5 },
17   { $limit: 5 }
18 ]);
```

Output

```
Products after skipping top 5 expensive ones:
[
  {
    _id: new ObjectId('687c49221e4da9d4c480b2b5'),
    name: 'Microphone',
    price: 250,
    __v: 0
  },
  {
    _id: new ObjectId('687c49221e4da9d4c480b2b3'),
    name: 'Speaker',
    price: 200,
    __v: 0
  },
  {
    _id: new ObjectId('687c49221e4da9d4c480b2b4'),
    name: 'Webcam',
    price: 150,
    __v: 0
  },
  {
    _id: new ObjectId('687c49221e4da9d4c480b2b0'),
    name: 'Keyboard',
    price: 100,
    __v: 0
  },
  {
    _id: new ObjectId('687c49221e4da9d4c480b2b1'),
    name: 'Mouse',
    price: 50,
    __v: 0
  }
]
```

mongoDB

$lookup

- $lookup allows you to join data from another collection, similar to JOIN in SQL.
- It performs a left outer join, which means:
  - All documents from the “left” collection (main one) will be returned.
  - Matching documents from the “right” collection (foreign) will be included if they exist.
  - If no match is found, an empty array is returned in the field.

$lookup

Important Notes:

- from must be the collection name, not the model name (e.g., 'courses', not 'Course')
- Only works across collections in the same database
- **$lookup** adds an array, even if only one document is matched
- Combine with **$unwind** if you want to flatten single-value arrays
- Cannot populate using **$lookup**; use one or the other
- Heavy **$lookups** on large datasets can be slow – use wisely

$lookup

$lookup vs .populate() vs SQL JOIN

Feature	$lookup	.populate()	SQL JOIN
Where it runs	DB-level (fast)	App-level (JS)	DB-level (fast)
Returns	Array (always)	Object or Array	Rows
Flexibility	Very high (can filter, reshape, etc.)	Low (automatic only)	High
Use case	Reports, stats, dashboards	Normal app behavior	Complex queries

```javascript
await Student.insertMany([
  { name: 'Alice', courseIds: [course1._id, course2._id] }
  { name: 'Charlie', courseIds: [] }
]);

const result = await Student.aggregate([
  {

    $lookup: {
      from: 'courses',
      localField: 'courseIds',
      foreignField: '_id',
      as: 'enrolledCourses'
    },

  },
  {
  $project: {
      name: 1,
      enrolledCourses: 1
    }
    }
]);
```

Output

```javascript
[
  {
    "_id": "687c4b81c5492eef52b698f2",
    "name": "Alice",
    "courseIds": [
      "687c4b81c5492eef52b698ec",
      "687c4b81c5492eef52b698ee"
    ],
    "__v": 0,
    "enrolledCourses": [
      {
        "_id": "687c4b81c5492eef52b698ec",
        "title": "Math",
        "__v": 0
      },
      {
        "_id": "687c4b81c5492eef52b698ee",
        "title": "Physics",
        "__v": 0
      }
    ]
  },
  {
    "_id": "687c4b81c5492eef52b698f4",
    "name": "Charlie",
    "courseIds": [],
    "__v": 0,
    "enrolledCourses": []
  }
]
```

mongoDB

$addFields

- **$addFields** adds new fields to each document in the pipeline.
- It can also modify existing fields.
- It's non-destructive; Existing fields stay unless you overwrite them.
- It’s like adding a computed column in a SELECT query:

```
SELECT name, salary, salary * 0.1 AS bonus FROM employees;
```

$addFields

Use Cases:

- Computed fields (discounts, totals, etc.)
- Text transformations ($toUpper, **$concat**)
- Extracting from arrays (**$arrayElemAt**)
- Combining with $project for cleaner output
- Don’t use it if you only want to remove or rename fields; Use **$project** for that.

$addFields vs $project

mongoDB

Feature	$addFields	$project
Adds new fields	Yes	Yes
Removes fields	No (keeps all existing fields)	Yes (explicitly include/exclude fields)
Modifies fields	Yes	Yes (can override fields with new expressions)
Default behavior	Keeps all existing fields	Excludes all unless you include them manually
Best used for	Calculating and appending new data	Shaping final output (like SELECT in SQL)

```
wait Product.insertMany([
 { name: 'Laptop', price: 1000, tags: ['electronics', 'computing'
 { name: 'Phone', price: 500, tags: ['electronics', 'mobile'] },
 { name: 'Book', price: 40, tags: ['reading', 'education'] }
);

onst result = await Product.aggregate([
 {
   $addFields: {
     discountPrice: { $multiply: ['$price', 0.8] },
     upperCaseName: { $toUpper: '$name' },
     firstTag: { $arrayElemAt: ['$tags', 0] }
   }
 },
 {
   $project: {
     _id: 0,
     name: 1,
     price: 1,
     discountPrice: 1,
     upperCaseName: 1,
     firstTag: 1
   }
 }
);
```

Output

```
1  //With Added Fields:
2  [
3    {
4      name: 'Laptop',
5      price: 1000,
6      discountPrice: 800,
7      upperCaseName: 'LAPTOP',
8      firstTag: 'electronics'
9    },
10   {
11     name: 'Phone',
12     price: 500,
13     discountPrice: 400,
14     upperCaseName: 'PHONE',
15     firstTag: 'electronics'
16   },
17   {
18     name: 'Book',
19     price: 40,
20     discountPrice: 32,
21     upperCaseName: 'BOOK',
22     firstTag: 'reading'
23   }
24 ]
25
```

mongoDB

$out

- **$out** writes the final result of the aggregation pipeline to a collection (either replacing it or creating a new one).
- Think of it like saving the result of a complex query into its own collection.
- Must be the last stage in the pipeline.
- It will replace the contents of the target collection.
- When to use **$out**:
  - Caching expensive aggregation results.
  - Generating reports.
  - Archiving filtered data.
  - Creating flattened/denormalized views of data.

```javascript
await Product.insertMany([
  { name: 'Laptop', price: 1000, tags: ['electronics', 'computing'] },
  { name: 'Phone', price: 500, tags: ['electronics', 'mobile'] },
  { name: 'Book', price: 40, tags: ['reading', 'education'] }
]);

await Product.aggregate([
  {
    $addFields: {
      discountPrice: { $multiply: ['$price', 0.8] },
      upperCaseName: { $toUpper: '$name' },
      firstTag: { $arrayElemAt: ['$tags', 0] }
    }
  },
  {
    $project: {
      _id: 0,
      name: 1,
      price: 1,
      discountPrice: 1,
      upperCaseName: 1,
      firstTag: 1
    }
  },
  {
    $out: 'discounted_products'
  }
]);
```

Output

```javascript
// Discounted Products:
[
  {
    _id: new ObjectId('687c58182ca71a8e723d25c3'),
    name: 'Laptop',
    price: 1000,
    discountPrice: 800,
    upperCaseName: 'LAPTOP',
    firstTag: 'electronics'
  },
  {
    _id: new ObjectId('687c58182ca71a8e723d25c4'),
    name: 'Phone',
    price: 500,
    discountPrice: 400,
    upperCaseName: 'PHONE',
    firstTag: 'electronics'
  },
  {
    _id: new ObjectId('687c58182ca71a8e723d25c5'),
    name: 'Book',
    price: 40,
    discountPrice: 32,
    upperCaseName: 'BOOK',
    firstTag: 'reading'
  }
]
```

mongoDB

$merge

- **$merge** writes the output of an aggregation pipeline to a collection, just like $out, but with more control over what happens when documents with matching _id already exist.
- Can output to a collection in the same or different database.
- Can output to the same collection that is being aggregated.
- Creates a new collection if the output collection does not already exist.
- Can incorporate results (insert new documents, merge documents, replace documents, keep existing documents)

```
await Product.aggregate([
    {
        $match: { hasDiscount: true }
    },
    {
        $addFields: {
            discountPrice: {
                $subtract: [
                    '$price',
                    { $multiply: ['$price', '$discountRate'] }
                ]
            },
            uppercaseName: { $toUpper: '$name' },
            firstTag: { $arrayElemAt: ['$tags', 0] }
        }
    },
    {
        $project: {
            _id: 1,
            name: 1,
            price: 1,
            discountRate: {
                $concat: [
                    { $toString: { $multiply: [100, '$discountRate'] } }, '%']
            },
            discountPrice: 1,
            uppercaseName: 1,
            firstTag: 1
        }
    },

    {
        $merge: {
            into: 'discounted_products',
            whenMatched: 'merge',
            whenNotMatched: 'insert'
        }
    }
]);
```

Output

```
//Discounted Products:
 [
  {
    _id: new ObjectId('687c6553e5f8e8623dc7a5fb'),
    discountPrice: 800,
    discountRate: '20%',
    firstTag: 'electronics',
    name: 'Laptop',
    price: 1000,
    uppercaseName: 'LAPTOP'
  },
  {
    _id: new ObjectId('687c6553e5f8e8623dc7a5fd'),
    discountPrice: 36,
    discountRate: '10%',
    firstTag: 'education',
    name: 'Book',
    price: 40,
    uppercaseName: 'BOOK'
  },
  {
    _id: new ObjectId('687c6553e5f8e8623dc7a5fe'),
    discountPrice: 170,
    discountRate: '15%',
    firstTag: 'furniture',
    name: 'Desk',
    price: 200,
    uppercaseName: 'DESK'
  }
]
```

mongoDB

Indexing For Performance

Indexing

- Definition:
  - Indexing is a special data structure that MongoDB uses to make finding documents faster. Instead of scanning every document, MongoDB uses an index to quickly locate the data you need.
- Why it matters:
  - Without an index, MongoDB must scan every single document to find results (called a collection scan).
  - With an index, it can jump straight to the matching documents, saving time and resources.

Indexing

- Why Use Indexing?
  - Faster Read Performance
    - MongoDB **can return query results much** faster **using indexes.**
    - **Especially important as collections grow larger**.
  - More Efficient Queries
    - Useful when using .find(), .sort(), .limit(), and text search.
    - **Helps MongoDB avoid scanning the entire collection.**

Indexing

- Why Use Indexing?
  - Supports Advanced Features
    - Enables text search, geospatial queries, TTL (Time-To-Live) for expiring data, and more.
  - Optimized Sorting
    - Speeds up .sort() operations if the sort field is indexed.
  - ☐ Indexes don’t affect how you store or retrieve data, they just make searching smarter.

Indexing

- When Should You Use Indexes?
  - Use indexes when:
    - You're querying the same field often (e.g. usernames, emails).
    - You want to enforce uniqueness (e.g. no duplicate emails).
    - You're performing full-text search on article content.
    - You need to auto-delete documents (e.g. session data after 1 hour).
    - You're working with large collections and want to reduce response time.

Indexing

- Avoid over-indexing:
  - Indexes take up disk space.
  - Too many indexes can slow down insert and update operations.
  - Every time you write data, MongoDB must also update the indexes.
  - Only index the fields that your app searches or sorts on frequently.

Indexing

mongoDB

Types of Indexes in MongoDB

Index Type	Purpose & Use Case
Default (_id)	Created automatically for every document. Helps quickly find by ID.
Single Field	Indexes one field. Great for searching specific fields like email.
Compound	Combines multiple fields in one index (e.g. firstName, lastName).
Unique	Prevents duplicate values (e.g. unique usernames or emails).
Text	Enables full-text search within string fields like title or content.
TTL	Automatically deletes documents after a certain time (great for logs, tokens, etc).

Single Field Index

- **What is a Single Field Index?:**
  - A Single Field Index is the simplest type of index — it creates an index on one field only.
  - It helps MongoDB quickly search, filter, or sort based on that field.
- **Use it when:**
  - You often search for documents using one specific field
    → e.g. **find({ email: "user@example.com" })**
  - You want to speed up sorting on that field
    → e.g. **find().sort({ createdAt: -1 })**

Single Field Index

- **How to implement Single Field index in your node app:**

```
1  const userSchema = new mongoose.Schema({
2    name: String,
3    email: String,
4    age: Number
5  });
6
7  userSchema.index({ email: 1 }); // 1 for ascending order
8
```

Compound Index

- **What is a Compound Index?**
  - A Compound Index is an index on two or more fields together, in a specific order.
  - MongoDB uses it to speed up queries that involve multiple fields.
- **Use it when:**
  - You often filter or sort using multiple fields
    → e.g. **find({ name: "Aly", age: 21 })**
  - You want to support sorted results based on a field combination

Compound Index

- **How to implement Compound index in your node app:**

```
1  const userSchema = new mongoose.Schema({
2    name: String,
3    email: String,
4    age: Number
5  });
6
7  userSchema.index({ name: 1, age: -1 });
```

- Order matters: { name: 1, age: -1 } is not the same as { age: -1, name: 1 }.
- { name: 1, age: -1 } supports:
  - **find({ name: "Ali" })**
  - **find({ name: "Ali", age: 25 })**
  - But not **find({ age: 25 })** alone

Unique Index

- **What is a Unique Index?**
  - A Unique Index makes sure that no two documents in a collection have the same value for a specific field (or combination of fields).
  - It enforces data uniqueness, just like **PRIMARY KEY** in SQL.
- Why Use It?
  - To prevent duplicates (like duplicate emails or usernames)
  - To ensure data integrity
  - Saves time by automatically rejecting duplicates without extra code

Unique Index

- **How to implement unique index in your node app:**

```
const userSchema = new mongoose.Schema({
  email: {
    type: String,
    unique: true
  },
  name: String
});
```

- unique: true in Mongoose is a shortcut — it tells MongoDB to create the index, but doesn't guarantee it'll exist unless the index is built
- You can manually drop and re-create unique indexes if needed
- If duplicate data already exists, MongoDB will fail to create the index

Text Index

- **What is a Text Index?**
  - A text index allows MongoDB to perform full-text search on string fields like titles, descriptions, or content — it searches for words, not exact strings.
  - Perfect for apps with articles, books, blogs, or searchable content!
- **Why Use Text Indexes?**
  - Without text indexes:
    - Searching large text fields is slow
    - You can only use regular expressions (/word/) which don't scale
  - **With text indexes:**
    - MongoDB analyzes and indexes words
    - Searching is optimized, fast, and more relevant
    - Supports ranking results based on relevance

Text Index

- **How Does It Work Internally?**
  - MongoDB tokenizes strings → breaks content into words
  - It removes stop words (e.g., “the”, “is”, “and”)
  - It indexes the words, not full strings
  - Each word points to the documents where it appears
- **Tips:**
  - Only works on string fields
  - You can’t use regex and **$text** together in the same query
  - You can’t create another text index unless you drop the first
  - Make sure the fields you're indexing have meaningful, searchable content

Text Index

- How to implement text index in your node app:

```
const bookSchema = new mongoose.Schema({
  title: String,
  content: String,
  page: Number
});

bookSchema.index({ title: 'text', content: 'text' });
```

- How to Search with a Text Index:

```
await Book.find({ $text: { $search: "science" } })
```

- MongoDB will now quickly return documents where "science" appears in either title or content.

TTL Index

- **What is a TTL Index?**
  - A TTL (Time-To-Live) index automatically deletes documents after a specified amount of time.
  - Useful for temporary data like sessions, OTPs, tokens, notifications, or logs.

- **Why Use TTL Indexes?**
  - No manual cleanup needed
  - Saves space by removing outdated data
  - Great for caching, user activity, or expiring messages
  - Keeps your collection lightweight and fast

TTL Index

- **How Does TTL Works Internally?**
  - You add a special Date field (e.g., createdAt)
  - Create a TTL index on that field
  - MongoDB checks the field’s value and automatically deletes the document when it expires
- **Tips:**
  - TTL only works on fields that are actual Date objects
  - MongoDB doesn't guarantee exact deletion time, it's approximate (~60s delay)
  - You can’t use TTL on multiple fields at once
  - Best used for auto-expiring documents, not scheduling tasks

Single Field Index

- **How to implement TTL index in your node app**

```
1  const messageSchema = new mongoose.Schema({
2    data: String,
3    createdAt: {
4      type: Date,
5      default: Date.now,
6      expires: 10
7    }
8  });
```

- Now messages will expire in 10 seconds after its creation

Database Transactions

Database Transactions

- **What Are Database Transactions?**
  - A transaction is a sequence of operations (like reads/writes) that are executed as a single unit of work.
  - It either completes fully or doesn’t apply any changes at all.
- **Use Cases:**
  - Money transfers between bank accounts
  - Order placement with stock updates
  - Updating multiple documents atomically
  - Avoiding partial updates or data corruption

Database Transactions

ACID Properties

Property	Description
Atomicity	All operations succeed or none do
Consistency	Maintains valid data before and after
Isolation	Concurrent transactions don't interfere
Durability	Data changes persist even after system failure

Database Transactions

- **Consider the following scenario:**
  - A user places an order
  - You deduct item from stock
  - Then an error happens before You insert the order in the orders collection
  - Order is not placed, but the stock is already reduced, which leads to Inconsistent data

Database Transactions

- **With a transaction:**
  - Begin transaction
  - Deduct item from stock
  - Insert the order
  - Commit transaction
  - If any step fails, all changes are rolled back and the stock returns to normal

MongoDB Transactions

- **What can you do with MongoDB Transactions? :**
  - Update multiple documents in multiple collections
  - Read + write in the same session
  - Roll back if any part fails
  - Combine complex business logic in one atomic operation

MongoDB Transactions

- **When Should You Use Transactions in MongoDB?**
  - Your operations span multiple documents/collections
  - You need strict consistency
  - You're simulating relational operations
  - You want fail-safe batch writes
- **Avoid When:**
  - You only modify one document (MongoDB is atomic at doc level)
  - You can use simpler patterns (like pre-save hooks or denormalization)
  - You're optimizing for performance over strict ACID

MongoDB Transactions

- **How to implement transactions in your app:**

```
const session = await mongoose.startSession();
session.startTransaction();

try {
  await User.updateOne({ _id: userId }, { $inc: { balance: -100 } }).session(session);
  await Order.create([{ userId, items }], { session });

  await session.commitTransaction();
  session.endSession();
} catch (err) {
  await session.abortTransaction();
  session.endSession();
  throw err;
}
```

MongoDB Transactions

- Key Notes on MongoDB Transactions
  - Only works with replica sets (or sharded clusters)
  - You must use sessions
  - Extra overhead, slower than normal operations
  - Good for rare, sensitive workflows, not everyday CRUD

MongoDB Replica Sets

- A replica set in MongoDB is a group of mongodb processes that maintain the same data set, providing high availability and data redundancy.

- Why use a Replica Set?

  - High Availability
    If the primary node fails, another node is automatically elected as the new primary—this avoids downtime.

  - Data Redundancy
    Data is replicated across multiple servers, reducing the risk of data loss.

  - Read Scalability (optional)
    You can distribute read operations across secondaries (with eventual consistency).

  - Transactions & Sessions Support
    Transactions across multiple documents (or collections) require replica sets.

MongoDB Replica Sets

- How it Works:
  - The primary node receives all writes.
  - Secondaries continuously replicate the data from the primary.
  - If the primary goes down, the replica set automatically elects a new primary.
- Example Use Case
  - To ensure no orders are lost if a server crashes, you:
    - Deploy 1 primary and 2 secondaries.
    - If the primary fails, a secondary takes over.
    - Writes and transactions continue with minimal interruption.

MongoDB Replica Sets

- How to initiate a replica set:
  - First, in your terminal, type the following command:

```
mongod --replSet rs0
```

  - **Then in a new terminal, start the mongosh shell and run:**

```
rs.initiate()
```

Any Questions?