**Scenario 1: Slow Query Performance**

**Question:**

You have a collection orders with millions of documents. A frequent query filters orders by customerId and orderDate. The query takes too long to execute. What indexing strategy should you use?

**Answer:**

Create a **compound index** on { customerId: 1, orderDate: -1 }. This will speed up queries filtering by customerId and sorting by orderDate in descending order.

db.orders.createIndex({ customerId: 1, orderDate: -1 })

This will optimize queries like:

db.orders.find({ customerId: 123 }).sort({ orderDate: -1 })

Without an index, MongoDB performs a **collection scan**, which is slow.

**Scenario 2: Unique Constraints**

**Question:**

You need to ensure that each user has a unique email in the users collection. How do you enforce this at the database level?

**Answer:**

Use a **unique index** on the email field.

db.users.createIndex({ email: 1 }, { unique: true })

This prevents duplicate emails from being inserted.

**Scenario 3: Index Selection Issue**

**Question:**

A collection has multiple indexes, but your query is still slow. How do you check which index MongoDB is using?

**Answer:**

Use the **explain()** method with .executionStats to analyze index usage.

db.orders.find({ customerId: 123 }).explain("executionStats")

* If the query does not use the expected index, consider **creating a more efficient index** or using **index hints**.
* To force a specific index:
* db.orders.find({ customerId: 123 }).hint({ customerId: 1 })

**Scenario 4: Large Text Searches**

**Question:**

Your application allows searching products by name and description. What indexing strategy should you use?

**Answer:**

Use a **text index** for efficient full-text search.

db.products.createIndex({ name: "text", description: "text" })

Search query:

db.products.find({ $text: { $search: "laptop" } })

This enables keyword-based searching across multiple fields.

**Scenario 5: Index Impact on Write Performance**

**Question:**

Your users collection has **5 indexes**, and inserts are slow. Why is this happening?

**Answer:**

Each insert operation must update all associated indexes, slowing down write performance. Solutions:

1. **Reduce unnecessary indexes** (keep only essential ones).
2. **Use write-optimized storage options**.
3. **Batch insertions** instead of single inserts.

**Scenario 6: Geospatial Queries**

**Question:**

You need to find the nearest restaurants to a user’s location stored as { latitude, longitude }. How do you index this?

**Answer:**

Use a **2dsphere index** for geospatial queries.

db.restaurants.createIndex({ location: "2dsphere" })

Find nearby restaurants:

db.restaurants.find({

location: {

$near: {

$geometry: { type: "Point", coordinates: [longitude, latitude] },

$maxDistance: 5000 // 5km radius

}

}

})

**Scenario 7: Indexing an Array Field**

**Question:**

Your articles collection has a tags field (array). How do you efficiently query articles that contain a specific tag?

**Answer:**

Use a **multikey index**.

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

Query:

db.articles.find({ tags: "mongodb" })

MongoDB automatically creates a **multikey index** for array fields.

**Scenario 8: Sorting Without an Index**

**Question:**

A query sorts results by createdAt, but performance is poor. How can you improve it?

**Answer:**

Create an index on createdAt:

db.posts.createIndex({ createdAt: -1 })

Sorting without an index requires MongoDB to **load all documents into memory** before sorting.

**Scenario 9: Index Too Large**

**Question:**

Your collection has a large index, consuming too much RAM. How can you optimize it?

**Answer:**

* **Use smaller index keys** (avoid large strings).
* **Drop unnecessary indexes**.
* **Use Partial or Sparse Indexes** to index only relevant documents.

Example: Index only active users

db.users.createIndex({ lastLogin: 1 }, { partialFilterExpression: { status: "active" } })

**Scenario 10: TTL Index for Auto-Deleting Data**

**Question:**

You need to automatically delete logs after 30 days. How can you do this?

**Answer:**

Use a **TTL (Time-To-Live) index**.

db.logs.createIndex({ createdAt: 1 }, { expireAfterSeconds: 2592000 }) // 30 days

This will automatically delete documents **30 days after createdAt**.