ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

WORKING WITH INDEXES & PERFORMANCE OPTIMIZATION (WEEKS 7-8)

TYPES OF INDEXES IN MONGODB: SINGLE FIELD, COMPOUND, MULTIKEY

CHAPTER 1: INTRODUCTION TO INDEXING IN MONGODB

1.1 What is Indexing in MongoDB?

Indexing in **MongoDB** is a technique that **improves query performance** by reducing the amount of data that needs to be scanned. Without an index, MongoDB performs a **collection scan**, meaning it checks **every document** in a collection, which is slow for large datasets.

✓ **Indexes improve search speed** by organizing data efficiently.
✓ **Indexes reduce query execution time** by allowing MongoDB to locate documents faster.
✓ **Indexes work similarly to an index in a book**, helping locate information quickly.

1.2 How Indexes Work?

MongoDB indexes are stored as **B-Trees,** making lookups **faster and scalable**. When a query is executed:

1. MongoDB checks if an **index exists** for the queried field.
2. If an index is present, it **retrieves the documents quickly**.
3. If no index is found, MongoDB **scans the entire collection,** slowing performance.

Example: Finding a user by email **without an index**

```
db.users.find({ email: "alice@example.com" });
```

✓ If **no index exists,** MongoDB **checks every document,** making queries slow.

CHAPTER 2: SINGLE FIELD INDEX

2.1 What is a Single Field Index?

A **single field index** creates an index on **one specific field,** optimizing queries that filter or sort using that field.

✓ Best for **queries filtering or sorting by a single field**.
✓ Works well when **a field is frequently queried**.
✓ Improves **read performance but increases write overhead**.

2.2 Creating a Single Field Index

To create an index on the email field:

```
db.users.createIndex({ email: 1 });
```

✓ 1 means **ascending order** (-1 would create a **descending index**).

2.3 Using an Index in a Query

If the index exists, MongoDB will use it:

```
db.users.find({ email: "alice@example.com"
}).explain("executionStats");
```

✓ The query planner will show **IXSCAN** instead of **COLLSCAN**, confirming that the index is used.

2.4 Sorting with a Single Field Index

Indexes **optimize sorting operations**:

```
db.users.find().sort({ email: 1 });
```

✓ Without an index, sorting **scans the entire collection**.
✓ With an index, sorting **uses the pre-sorted index data**, improving efficiency.

CHAPTER 3: COMPOUND INDEX

3.1 What is a Compound Index?

A **compound index** creates an index on **multiple fields**, optimizing queries that filter or sort by multiple fields.

✓ Best for queries that **filter by multiple fields**.
✓ Works well when queries **sort data using multiple fields**.
✓ Improves **query execution time** when combining multiple conditions.

3.2 Creating a Compound Index

To create an index on firstName and lastName:

db.users.createIndex({ firstName: 1, lastName: 1 });

✓ Queries filtering by **firstName and lastName** will use this index.

3.3 How Compound Indexes Work?

A compound index { firstName: 1, lastName: 1 } supports:

✓ Queries that filter by **firstName** alone.
✓ Queries that filter by **both firstName and lastName**.
❌ **Does NOT** support queries filtering by lastName alone.

Example: Using a compound index efficiently

db.users.find({ firstName: "Alice", lastName: "Johnson" });

✓ This query **uses the index** because it matches the indexed fields.

3.4 Sorting with a Compound Index

db.users.find().sort({ firstName: 1, lastName: -1 });

✓ Uses the **compound index** to optimize sorting.

CHAPTER 4: MULTIKEY INDEX

4.1 What is a Multikey Index?

A **multikey index** is used for **indexing arrays**, allowing searches within an array field.

✓ Best for **fields storing multiple values** (e.g., tags, categories).
✓ Allows **efficient queries on array elements**.
✓ Automatically created when **indexing an array field**.

4.2 Creating a Multikey Index

If tags is an **array field,** create a multikey index:

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

✓ MongoDB **automatically detects** arrays and creates a **multikey index**.

4.3 Querying an Array Field with a Multikey Index

Find articles where tags include "MongoDB":

```
db.articles.find({ tags: "MongoDB" });
```

✓ Uses the **multikey index** to efficiently filter articles.

4.4 Limitations of Multikey Indexes

❌ **Multikey indexes cannot be used in compound indexes** with **multiple arrays**.
❌ **Sorting on a multikey index is limited,** requiring careful query design.

CHAPTER 5: COMPARING INDEX TYPES

Index Type	Best For	Example Use Case
Single Field	Queries filtering by **one field**	Searching users by email
Compound	Queries filtering by **multiple fields**	Searching users by firstName + lastName

Multikey	Queries searching inside **array fields**	Finding articles with tags: ["MongoDB", "NoSQL"]

Case Study: How Twitter Uses Indexing for Fast Search

Background

Twitter processes **millions of tweets per second,** requiring **high-speed indexing** for search and recommendations.

Challenges

- **Efficiently searching tweets by hashtags and users**.
- **Handling high-speed queries for trending topics**.
- **Balancing indexing performance with real-time data updates**.

Solution: Implementing Indexed Search

✓ **Used compound indexes** for querying tweets by userId and createdAt.
✓ **Applied multikey indexes** for hashtags (tags array field).
✓ **Created single-field indexes** for user searches (username).

By optimizing indexing strategies, Twitter **reduced search latency by 80%,** improving real-time performance.

Exercise

1. Create a **single field index** on the email field in a users collection.
2. Create a **compound index** on category and price for a products collection.

3. Implement a **multikey index** on an articles collection for a tags array.
4. Use .explain("executionStats") to verify that a query is using an index.

Conclusion

In this section, we explored:

✓ **Single field indexes for optimizing queries by one field**.

✓ **Compound indexes for improving performance in multi-field queries**.

✓ **Multikey indexes for indexing array fields efficiently**.

✓ **How Twitter uses indexing to enhance search speed**.

UNDERSTANDING INDEXING PERFORMANCE WITH EXPLAIN() IN MONGODB

CHAPTER 1: INTRODUCTION TO INDEXING PERFORMANCE IN MONGODB

1.1 What is Indexing in MongoDB?

Indexes are **special data structures** in MongoDB that **improve query performance** by allowing the database to find matching documents **faster**. Without indexes, MongoDB **scans the entire collection** (a full collection scan) to locate the desired records, which can lead to **slow queries**.

✓ **Indexes reduce query execution time** by minimizing the number of documents scanned.
✓ **They speed up read operations** (e.g., find(), sort(), count()).
✓ **They are essential for optimizing large datasets** with millions of records.

1.2 The Role of explain() in Indexing

MongoDB provides the explain() method, which helps **analyze query execution** and determine whether an index is being used efficiently.

✓ **Shows query execution details** – Displays whether an index or full collection scan is used.
✓ **Helps optimize query performance** – Identifies inefficient queries.
✓ **Provides execution time statistics** – Measures the time taken to fetch results.

Example: Checking Query Execution Plan

```
db.users.find({ email: "john@example.com"
}).explain("executionStats")
```

✓ This **analyzes the query execution,** showing whether an **index or collection scan** was used.

CHAPTER 2: USING EXPLAIN() TO ANALYZE QUERIES

2.1 The Three Modes of explain()

MongoDB supports three execution modes for explain():

Mode	Description
"queryPlanner"	Shows the **planned query execution** without running it.
"executionStats"	Runs the query and **returns execution details**.
"allPlansExecution"	Returns **all execution plans,** useful for performance tuning.

Example: Running explain() in Different Modes

```
db.orders.find({ customerId: "12345" }).explain("queryPlanner")
```

✓ Returns the **query execution plan,** but does **not execute the query**.

```
db.orders.find({ customerId: "12345" }).explain("executionStats")
```

✓ Runs the query and returns **execution time, scanned documents, and index usage**.

CHAPTER 3: INTERPRETING EXPLAIN() OUTPUT

3.1 Key Metrics in explain()

The output of explain() contains various fields that help analyze query performance.

Field Name	Description
queryPlanner.winningPlan	Shows the query execution plan used.
executionStats.nReturned	Number of documents returned by the query.
executionStats.executionTimeMillis	Total time (in milliseconds) to execute the query.
executionStats.totalDocsExamined	Number of documents scanned before returning results.

Example: Analyzing explain() Output

```
db.users.find({ age: { $gt: 25 } }).explain("executionStats")
```

📌 **Sample Output (Simplified)**

```
{
 "executionStats": {
  "nReturned": 50,
  "executionTimeMillis": 12,
  "totalDocsExamined": 1000,
  "indexName": "age_1"
 }
}
```

✓ **nReturned** → The query returned **50 matching documents**.
✓ **executionTimeMillis** → The query took **12 milliseconds** to execute.
✓ **totalDocsExamined** → **1000 documents** were scanned before finding the 50 results.
✓ **indexName** → The query used the **age_1 index,** improving performance.

CHAPTER 4: OPTIMIZING QUERY PERFORMANCE USING INDEXES

4.1 Identifying Queries That Need Indexes

A **slow query** is typically identified when:

✓ **totalDocsExamined is much higher than nReturned**.
✓ **executionTimeMillis is high** (indicating long query execution).
✓ **The query is performing a full collection scan (COLLSCAN)** instead of an indexed scan (IXSCAN).

Example: A Query Performing a Full Collection Scan

```
db.products.find({ price: { $gt: 500 } }).explain("executionStats")
```

📌 **Sample Output (Before Indexing)**

```
{
 "executionStats": {
  "executionStages": {
   "stage": "COLLSCAN"
  },
  "totalDocsExamined": 100000,
  "nReturned": 500,
```

"executionTimeMillis": 200

}

}

🚨 **Warning Signs:**

- **stage: "COLLSCAN"** → A full collection scan is being performed.
- **totalDocsExamined is 100,000** → The database had to scan **all documents**.
- **executionTimeMillis is high** → Query execution took **200 milliseconds**.

4.2 Creating an Index to Improve Performance

To optimize the query, create an **index on the price field**:

```
db.products.createIndex({ price: 1 })
```

✓ This enables **indexed lookups,** reducing scan time.

Example: Query Performance After Indexing

```
db.products.find({ price: { $gt: 500 } }).explain("executionStats")
```

📌 **Sample Output (After Indexing)**

{

"executionStats": {

"executionStages": {

"stage": "IXSCAN"

},

"totalDocsExamined": 500,

```
  "nReturned": 500,
  "executionTimeMillis": 10
 }
}
```


🚀 **Improvements:**

- **stage: "IXSCAN"** → The query now uses an **index scan** instead of a collection scan.
- **totalDocsExamined dropped from 100,000 to 500**.
- **executionTimeMillis reduced from 200ms to 10ms**.

🔹 **Indexing improved performance by 95%!**

CHAPTER 5: TYPES OF INDEXES FOR PERFORMANCE OPTIMIZATION

5.1 Common Index Types and Their Use Cases

Index Type	Description	Use Case
Single-Field Index	Indexes a single field.	Fast lookups on a single column (email).
Compound Index	Indexes multiple fields together.	Optimizes searches involving **multiple criteria**.
Text Index	Enables full-text search on text fields.	Searching blogs, articles, and product descriptions.
TTL Index	Automatically deletes documents after a time.	Temporary data like **session logs**.

Geospatial Index	Supports location-based queries.	Mapping and **nearby searches**.

5.2 Example: Creating a Compound Index

If queries filter by both category and price, create a **compound index**:

```
db.products.createIndex({ category: 1, price: -1 })
```

✓ **Speeds up searches** filtering by category and sorting by price.

Case Study: How a Travel Booking Website Optimized Queries Using Indexes

Background

A **travel booking website** faced **slow search performance** when users filtered hotels by **location, price, and rating**.

Challenges

✓ **Full collection scans** slowed down hotel searches.
✓ **High query execution times** during peak traffic.
✓ **Poor user experience** due to long response times.

Solution: Using Indexing and explain() for Optimization

The team used explain() to analyze queries and:

✓ **Created compound indexes** on location, price, and rating.
✓ **Replaced collection scans with indexed lookups (IXSCAN)**.
✓ **Reduced query execution time from 500ms to 30ms**.

Results

🚀 **Faster search queries** led to **higher user engagement**.
⚡ **Database load reduced by 80%,** improving scalability.
🔍 **Real-time insights** into slow queries using explain().

By leveraging **MongoDB indexing and performance analysis**, the travel website achieved **faster, more efficient searches**.

Exercise

1. Run explain("executionStats") on a query that filters **users by age**.
2. Identify whether it uses **COLLSCAN or IXSCAN**.
3. Create an **index on the age field** and analyze performance again.
4. Optimize a **product search query** using a **compound index** on category and price.

Conclusion

In this section, we explored:
✓ **How explain() helps analyze query execution performance**.
✓ **How to identify slow queries and optimize them with indexes**.
✓ **How different types of indexes improve query efficiency**.

IMPLEMENTING PARTIAL & SPARSE INDEXES IN MONGODB

CHAPTER 1: INTRODUCTION TO INDEXING IN MONGODB

1.1 Understanding Indexes in MongoDB

An **index** in MongoDB is a data structure that improves the speed of **query operations** on a collection by reducing the number of documents that must be scanned.

Indexes help in:

✓ **Faster queries** – Reducing query execution time.
✓ **Optimized read performance** – Especially for large datasets.
✓ **Reduced I/O operations** – Less data scanned improves efficiency.

MongoDB automatically creates an **index on the _id field,** but additional indexes must be created manually for optimized queries.

Types of indexes include:

- **Single Field Indexes** – Index on a single field.
- **Compound Indexes** – Index on multiple fields.
- **Text Indexes** – Used for text search.
- **Partial Indexes** – Indexes only a subset of documents.
- **Sparse Indexes** – Indexes only documents that have the indexed field.

In this chapter, we focus on **Partial and Sparse Indexes,** which optimize storage and query performance by reducing unnecessary indexing.

CHAPTER 2: UNDERSTANDING PARTIAL INDEXES IN MONGODB

2.1 What is a Partial Index?

A **Partial Index** creates an index **only on documents that meet a specified condition**. This is useful for indexing frequently queried subsets of data while **reducing index size**.

✓ **Smaller index size** – Saves storage space.
✓ **Faster queries** – Filters unnecessary documents.
✓ **Improves performance** – Queries match fewer indexed values.

2.2 Creating a Partial Index

A **partial index** is created using the { partialFilterExpression: { condition } } option.

Example: Index Only Active Users

```
db.users.createIndex(
  { email: 1 },
  { partialFilterExpression: { status: "active" } }
);
```

✓ **Only users with status: "active" are indexed**.
✓ Queries for **inactive users won't use the index,** reducing overhead.

To check the index:

```
db.users.getIndexes();
```

2.3 Querying with a Partial Index

Optimized Query Using the Partial Index

```
db.users.find({ status: "active", email: "alice@example.com" }).explain("executionStats");
```

✓ The query uses the **partial index** if it matches the filter condition (status: "active").

2.4 Use Cases for Partial Indexes

Partial indexes are useful when:

✓ **Filtering active records** – Index only active users, available products, etc.
✓ **Indexing recent data** – Example: Orders from the last year.
✓ **Conditional indexing** – Example: Index only users with verified emails.

Example: Indexing Only Orders in the Last 12 Months

```
db.orders.createIndex(
  { orderDate: 1 },
  { partialFilterExpression: { orderDate: { $gte: new Date(new Date().setFullYear(new Date().getFullYear() - 1)) } } }
);
```

✓ Optimizes queries for **recent orders**, avoiding indexing old data.

CHAPTER 3: UNDERSTANDING SPARSE INDEXES IN MONGODB

3.1 What is a Sparse Index?

A **Sparse Index** only includes documents that have the indexed field. If a document **does not contain the field**, it is **not included in the index**.

✓ **Saves storage space** – Unindexed documents reduce index size.
✓ **Speeds up queries** – Avoids indexing unnecessary documents.
✓ **Prevents duplicate keys** – When a unique index is sparse.

3.2 Creating a Sparse Index

A sparse index is created using { sparse: true }.

Example: Index Only Users with an Email Field

```
db.users.createIndex(
  { email: 1 },
  { sparse: true }
);
```

✓ **Documents without an email field are not indexed**.
✓ Saves **storage** and improves query efficiency.

3.3 Querying with a Sparse Index

Optimized Query Using the Sparse Index

```
db.users.find({ email: "alice@example.com"
}).explain("executionStats");
```

✓ Uses the sparse index **only for documents that have an email**.
✓ Documents **without an email are skipped,** reducing index size.

3.4 Use Cases for Sparse Indexes

Sparse indexes are useful when:

✓ **Fields are optional** – Example: Indexing users **only if they have an email**.
✓ **Saving disk space** – Indexing only present fields reduces storage.
✓ **Handling unique constraints** – Avoids duplicate keys in optional fields.

Example: Unique Sparse Index for Optional Fields

```
db.users.createIndex(
  { phone: 1 },
  { unique: true, sparse: true }
);
```

✓ Ensures **phone numbers are unique**.
✓ Allows **documents without a phone number,** avoiding duplicate key errors.

CHAPTER 4: COMBINING PARTIAL AND SPARSE INDEXES

4.1 Using Both Partial & Sparse Indexes Together

Partial and sparse indexes can be combined to further **optimize performance**.

Example: Index Verified Users Who Have an Email

```
db.users.createIndex(
  { email: 1 },
  {
    sparse: true,
    partialFilterExpression: { verified: true }
```

```
  }
);
```

✓ **Indexes only verified users who have an email**.
✓ Reduces **index size** and speeds up queries.

Case Study: How a Job Portal Improved Performance Using Partial & Sparse Indexes

Background

A job portal stored **millions of user profiles,** but not all users had verified emails or phone numbers.

Challenges

- **Slow searches** due to indexing unnecessary fields.
- **Large index sizes,** increasing database load.
- **Duplicate key errors** when applying unique constraints on phone numbers.

Solution: Implementing Partial & Sparse Indexes

✓ **Sparse index on phone field** – Only indexed users with a phone number.
✓ **Partial index on email field** – Indexed only **verified** users.
✓ **Reduced index size,** making queries **60% faster**.

Results

✓ **Faster search queries,** reducing response time by **40%**.
✓ **Saved storage,** cutting index size by **50%**.
✓ **Better user experience,** with instant profile searches.

This case study highlights how **partial and sparse indexes significantly optimize query performance**.

Exercise

1. Create a **partial index** on a products collection for items that are in stock (stock > 0).
2. Create a **sparse index** on a users collection for the phone field.
3. Write a **query using explain()** to confirm that the index is used efficiently.

Conclusion

✓ **Partial Indexes** optimize queries by indexing only relevant documents.
✓ **Sparse Indexes** save space by excluding documents without the indexed field.
✓ **Combining both** improves performance and **reduces unnecessary indexing overhead**.

UNDERSTANDING QUERY OPTIMIZATION & CACHING

CHAPTER 1: INTRODUCTION TO QUERY OPTIMIZATION AND CACHING

1.1 What is Query Optimization?

Query optimization is the process of improving **database query performance** to ensure efficient data retrieval. Optimized queries execute **faster, consume fewer resources,** and enhance the overall performance of an application.

When databases handle **millions of queries per second,** poorly optimized queries can lead to **slow response times, high server load, and degraded user experience**. Query optimization techniques help minimize execution time and improve scalability.

1.2 What is Caching?

Caching is the practice of **storing frequently accessed data** in memory, reducing the need to repeatedly query the database. A cache **improves performance** by serving data instantly instead of retrieving it from disk storage every time.

Key Benefits of Caching:

✓ Reduces **database load** by avoiding unnecessary queries.
✓ Speeds up **response time,** improving user experience.
✓ Saves **computation resources,** optimizing cost efficiency.

Where is Caching Used?

- **Web applications** – Caching user sessions, API responses.
- **E-commerce platforms** – Caching product catalogs, pricing.

- **Streaming services** – Caching user preferences, recommendations.

CHAPTER 2: QUERY OPTIMIZATION TECHNIQUES

2.1 Using Indexing for Faster Query Execution

Indexes allow databases to **find data quickly** without scanning the entire table.

Example: Query Without an Index (Slow Execution)

```
SELECT * FROM users WHERE email = 'john@example.com';
```

- If the users table has **millions of records,** this query **scans all rows**.
- This leads to **high CPU and memory usage**.

Example: Creating an Index (Optimized Query Execution)

```
CREATE INDEX idx_email ON users(email);
```

- Now, the database **searches through the index** instead of scanning all records, **speeding up the query**.

2.2 Avoiding SELECT *

Using SELECT * retrieves **all columns,** increasing query execution time.

Bad Practice (Slow Query Execution)

```
SELECT * FROM orders WHERE customer_id = 101;
```

- This **fetches all columns,** even if **only one or two fields** are needed.

Good Practice (Optimized Query Execution)

SELECT order_id, amount FROM orders WHERE customer_id = 101;

✓ Fetches only **necessary columns**, improving efficiency.

2.3 Optimizing WHERE Clauses and Joins

Poorly written **WHERE** clauses and unnecessary **JOINs** can slow down queries.

Example: Inefficient Query with Unnecessary JOINs

SELECT customers.name, orders.amount

FROM customers, orders

WHERE customers.id = orders.customer_id;

- Without indexing, this performs a **full-table scan** on both tables.

Optimized Query Using Indexed JOINs

SELECT c.name, o.amount

FROM customers c

JOIN orders o ON c.id = o.customer_id;

✓ Uses an **indexed JOIN**, reducing unnecessary scans.

2.4 Using Query Execution Plans to Analyze Performance

Database systems provide **execution plans** to analyze query performance and suggest optimizations.

MySQL: Explain Query Plan

```
EXPLAIN SELECT * FROM users WHERE email =
'john@example.com';
```

✓ Shows how the query executes and whether **indexes are being used**.

MongoDB: Explain Query Plan

```
db.users.find({ email: "john@example.com"
}).explain("executionStats");
```

✓ Displays query execution statistics in MongoDB.

CHAPTER 3: CACHING STRATEGIES FOR PERFORMANCE IMPROVEMENT

3.1 Types of Caching

Caching Type	Description	Example Use Case
Application Cache	Stores frequently accessed data in memory	User sessions, API responses
Database Cache	Stores query results to reduce repeated database hits	E-commerce product listings
CDN (Content Delivery Network)	Caches static assets closer to users	Images, videos, CSS files

---

3.2 Implementing Caching in Node.js

1. Using Redis for Database Query Caching

Redis is an **in-memory database** that speeds up queries by storing frequently requested data.

Install Redis and the Redis Client for Node.js:

npm install redis

Connect Redis to Your Node.js Application:

```
const redis = require('redis');

const client = redis.createClient();


client.on('connect', () => {

   console.log('Connected to Redis');

});
```

3.3 Implementing Caching for API Responses

When a user requests data, **check if it exists in the cache** before querying the database.

Example: API Route with Redis Caching

```
const express = require('express');

const redis = require('redis');

const app = express();

const client = redis.createClient();

const db = require('./database'); // Simulated database module


app.get('/products/:id', async (req, res) => {
```

```
  const productId = req.params.id;


  // Check if data exists in Redis cache
  client.get(productId, async (err, cachedData) => {
    if (cachedData) {
      return res.json(JSON.parse(cachedData)); // Return cached data
    }

    // Fetch from database if not in cache
    const product = await db.getProductById(productId);
    client.setex(productId, 3600, JSON.stringify(product)); // Cache for 1 hour
    res.json(product);
  });
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

✓ Reduces **database queries** and speeds up API responses.
✓ Stores **cached results for 1 hour** to maintain up-to-date data.

3.4 Implementing Query Caching in MongoDB

MongoDB allows **query result caching** using **$cacheStage** in aggregation pipelines.

Example: Caching Aggregation Queries in MongoDB

```
db.orders.aggregate([
  { $match: { status: "completed" } },
  { $group: { _id: "$customer_id", totalSpent: { $sum: "$amount" } } },
  { $cacheStage: { cacheKey: "completed_orders_cache", ttl: 3600 } }
]);
```

✓ Stores **frequent aggregation results**, improving query speed.

Case Study: How Facebook Uses Caching for Real-Time Performance

Background

Facebook handles **billions of daily active users** with real-time interactions like **news feeds, notifications, and messaging**.

Challenges

- **Reducing database load** from frequent user requests.
- **Delivering real-time content updates** efficiently.
- **Minimizing server costs** while handling **massive traffic**.

Solution: Using Multi-Layered Caching

✓ **Redis for user sessions** – Speeds up login and profile data retrieval.
✓ **Memcached for API calls** – Caches frequently requested API responses.
✓ **Edge Caching (CDN)** – Delivers static files (images, videos) faster.

Results

- **40% faster page loads** across mobile and web apps.
- **75% reduction in database queries,** optimizing server performance.
- **Real-time notifications and chat updates** with near-zero latency.

This case study highlights **how caching significantly boosts performance and scalability** in high-traffic applications.

Exercise

1. What is the difference between **query optimization** and **caching**?
2. Write an **optimized SQL query** to find users who made purchases in the last 30 days.
3. Implement **Redis caching in a Node.js API route** for fetching product details.

Conclusion

In this section, we explored:

✓ **Query optimization techniques** like indexing and query execution plans.

✓ **How caching reduces database load** and speeds up application performance.

✓ **Real-world case studies on how large companies use caching for scalability**.

MANAGING LARGE DATASETS WITH SHARDING & REPLICATION IN MONGODB

CHAPTER 1: INTRODUCTION TO LARGE DATASET MANAGEMENT

1.1 Why Managing Large Datasets Matters?

As applications grow, databases must handle **large amounts of data efficiently** while ensuring **high availability and fast performance**. In **MongoDB**, two key techniques help manage large datasets:

✓ **Sharding** – Distributes data across multiple servers to improve **scalability**.
✓ **Replication** – Duplicates data across multiple servers to ensure **high availability** and **fault tolerance**.

1.2 Challenges of Large Datasets

1. **Slow Query Performance** – Large datasets lead to longer query times.
2. **Storage Limitations** – Single servers have limited storage and processing power.
3. **Single Point of Failure** – If one server crashes, data loss can occur.
4. **High Read/Write Load** – Applications with high traffic can overload a single database.

MongoDB provides **Sharding** and **Replication** to solve these issues, making databases **scalable, distributed, and highly available**.

CHAPTER 2: UNDERSTANDING SHARDING IN MONGODB

2.1 What is Sharding?

Sharding is a method of **splitting large datasets** across **multiple servers (shards)** to distribute the data evenly.

✓ **Improves Query Performance** – Queries execute faster by reducing load on each server.
✓ **Handles Large Data** – Sharding allows storing **terabytes of data** across many machines.
✓ **Increases Write Scalability** – Multiple shards handle concurrent write operations.

2.2 How Sharding Works?

A **sharded cluster** consists of:

- **Shards** – Data storage servers that hold a portion of the data.
- **Query Routers (mongos)** – Directs client requests to the correct shard.
- **Config Servers** – Stores metadata about shard locations.

2.3 Enabling Sharding in MongoDB

Step 1: Enable Sharding for a Database

```
sh.enableSharding("ecommerceDB");
```

✓ Allows ecommerceDB to be **sharded**.

Step 2: Shard a Collection Based on a Key

```
sh.shardCollection("ecommerceDB.orders", { userId: "hashed" });
```

✓ Distributes orders based on the **userId field**.

2.4 Choosing a Shard Key

A **shard key** determines how data is distributed. **Good shard keys:**

✓ **High Cardinality** – Many unique values (e.g., userId, email).
✓ **Even Distribution** – Avoids hotspots on a single shard.

Example of a Bad Shard Key:
❌ **Boolean fields (isActive: true/false)** – Leads to uneven distribution.

CHAPTER 3: UNDERSTANDING REPLICATION IN MONGODB

3.1 What is Replication?

Replication **copies data across multiple servers**, ensuring **high availability and failover protection**.

✓ **Ensures Data Redundancy** – Data is copied to multiple nodes.
✓ **Provides High Availability** – Prevents downtime during failures.
✓ **Supports Read Scaling** – Read operations can be distributed among replica nodes.

3.2 How Replication Works?

A **replica set** consists of:

- **Primary Node** – Handles all writes and reads by default.
- **Secondary Nodes** – Copies data from the primary node.
- **Arbiter (Optional)** – Helps elect a new primary node if failure occurs.

3.3 Setting Up a Replica Set

Step 1: Start MongoDB with Replica Set Mode

```
mongod --replSet "myReplicaSet"
```

✓ Starts MongoDB in **replication mode**.

Step 2: Initialize the Replica Set

```
rs.initiate({
 _id: "myReplicaSet",
 members: [
  { _id: 0, host: "server1:27017" },
  { _id: 1, host: "server2:27017" },
  { _id: 2, host: "server3:27017" }
 ]
});
```

✓ Defines **three nodes** in the replica set.

3.4 Checking Replica Set Status

To verify replication is working:

```
rs.status();
```

✓ Displays **health and sync status** of replica nodes.

CHAPTER 4: COMPARING SHARDING & REPLICATION

Feature	Sharding	Replication
Purpose	Distributes data	Duplicates data
Scaling	Horizontal (more servers)	Read scaling, failover protection
Used For	Large datasets	High availability
Failure Handling	Reduces load on individual servers	Automatically elects a new primary

---

4.1 When to Use Sharding vs. Replication?

Use Case	Best Approach
Large datasets (>1TB)	✅ Sharding
Disaster Recovery & Failover	✅ Replication
High Read Traffic	✅ Replication
Scalable Write Operations	✅ Sharding

Case Study: How Instagram Uses MongoDB for Large-Scale Data Management

Background

Instagram handles **billions of images, comments, and user interactions** daily.

Challenges

- **Efficiently storing user-generated content.**
- **Handling high read and write operations.**

- **Ensuring system uptime without failures.**

Solution: Implementing Sharding & Replication

✓ **Used sharding** to distribute images and user activity across multiple servers.
✓ **Implemented replication** for **high availability,** ensuring no downtime.
✓ **Optimized indexing and caching** to improve query speed.

This approach **improved Instagram's scalability,** enabling it to handle **millions of active users per second**.

Exercise

1. Enable **sharding** on a database named "myAppDB".
2. Set up a **shard key** for a "customers" collection using customerId.
3. Configure a **replica set** with three nodes.
4. Check the **replication status** in MongoDB.

Conclusion

In this section, we explored:
✓ **How Sharding distributes large datasets across multiple servers**.
✓ **How Replication ensures data redundancy and high availability**.
✓ **The differences between Sharding and Replication**.
✓ **How Instagram uses these techniques for scalability**.

TROUBLESHOOTING PERFORMANCE BOTTLENECKS IN MONGODB

CHAPTER 1: INTRODUCTION TO PERFORMANCE BOTTLENECKS IN MONGODB

1.1 Understanding Performance Bottlenecks

A **performance bottleneck** occurs when a part of the database system becomes a limiting factor, slowing down the entire application. MongoDB, while highly scalable and efficient, can experience **slow queries, high CPU usage, and excessive memory consumption** due to:

✓ **Poorly optimized queries** – Scanning too many documents.
✓ **Missing or inefficient indexes** – Increasing search time.
✓ **High write operations** – Causing excessive disk I/O.
✓ **Inefficient schema design** – Leading to unnecessary data duplication.

Identifying and resolving these bottlenecks **improves application performance, reduces costs, and enhances user experience**.

CHAPTER 2: IDENTIFYING PERFORMANCE BOTTLENECKS USING EXPLAIN()

2.1 Using explain() to Analyze Query Performance

MongoDB's explain() method provides insights into **query execution plans** to determine if a query is causing slowdowns.

Example: Identifying a Slow Query

```
db.orders.find({ status: "shipped" }).explain("executionStats")
```

📌 Key Fields to Analyze in explain() Output

Field Name	Description
executionTimeMillis	Time taken to execute the query (lower is better).
totalDocsExamined	Number of documents scanned before returning results.
stage	Indicates whether an index was used (IXSCAN) or if a full collection scan occurred (COLLSCAN).

🚨 Warning Signs of a Slow Query:

✓ **executionTimeMillis is high** → Query takes too long.

✓ **totalDocsExamined is much higher than nReturned** → Scanning too many documents.

✓ **stage: "COLLSCAN"** → Full collection scan instead of using an index.

CHAPTER 3: RESOLVING QUERY PERFORMANCE ISSUES

3.1 Creating Indexes to Speed Up Queries

If a query is **performing a full collection scan (COLLSCAN),** create an **index** on frequently queried fields.

Example: Creating an Index for Faster Searches

```
db.orders.createIndex({ status: 1 })
```

After indexing, re-run explain("executionStats") to confirm it uses **IXSCAN** instead of **COLLSCAN**.

3.2 Optimizing Query Logic

Some queries may **fetch unnecessary data,** slowing down performance.

Example: Retrieving Only Required Fields

Instead of:

```
db.users.find({ age: { $gte: 18 } })
```

Use **projection ({ field: 1 })** to return only necessary fields:

```
db.users.find({ age: { $gte: 18 } }, { name: 1, email: 1 })
```

✓ Reduces **network bandwidth usage**.
✓ Fetches **only required data** for better performance.

3.3 Using Pagination with limit() and skip()

Fetching **too many documents at once** can slow down performance. Implement **pagination** using limit() and skip().

Example: Implementing Pagination for User Listings

```
db.users.find().sort({ name: 1 }).skip(10).limit(10)
```

✓ Retrieves only **10 users at a time,** skipping the first 10.
✓ **Reduces query execution time** for large datasets.

CHAPTER 4: MONITORING SYSTEM PERFORMANCE METRICS

4.1 Checking CPU and Memory Usage

If MongoDB **consumes too much CPU or RAM,** check system resource usage.

📌 **To monitor system performance, use:**

- **top or htop (Linux/macOS)** – Shows MongoDB's CPU and memory usage.
- **Task Manager (Windows)** – Displays running processes and resource usage.
- **MongoDB's serverStatus() command** – Provides real-time database statistics.

Example: Checking MongoDB Server Status

db.serverStatus()

✓ Returns **CPU, memory, and active connections** statistics.

4.2 Monitoring Query Performance with db.currentOp()

To check **long-running queries**, use:

db.currentOp()

✓ Identifies **queries consuming too much time or resources**.

4.3 Using Profiling to Track Slow Queries

MongoDB's **Database Profiler** helps log slow queries.

Enable Profiling for Slow Queries (Longer than 100ms)

db.setProfilingLevel(1, 100)

✓ Logs queries **taking longer than 100ms**.

To view slow queries:

db.system.profile.find().sort({ millis: -1 }).limit(5)

✓ Returns the **5 slowest queries**, sorted by execution time.

CHAPTER 5: SCALING MONGODB TO HANDLE HIGH TRAFFIC

5.1 Using Replication for High Availability

MongoDB **Replication** ensures data redundancy by storing **copies of data** across multiple servers.

✓ **Reduces downtime** in case of server failures.
✓ **Distributes read operations** across replica nodes.

To check **replica set status**:

rs.status()

✓ Displays information about **replica set members**.

5.2 Using Sharding for Large Databases

If **datasets grow too large, sharding** distributes data across **multiple servers**.

✓ Improves **write performance**.
✓ Balances **load across multiple servers**.

To **enable sharding,** use:

sh.enableSharding("myDatabase")

✓ Enables **horizontal scaling** for large applications.

Case Study: How a FinTech Company Optimized MongoDB Performance

Background

A **FinTech company** handling **millions of transactions** faced performance bottlenecks, causing:

✓ **Slow query responses** during peak hours.
✓ **High CPU and memory usage**, leading to system crashes.
✓ **Data inconsistency issues** due to write conflicts.

Challenges

- Query execution times **exceeded 1 second**, impacting user experience.
- The database **locked during write-heavy operations**.
- **Full collection scans** slowed down transactions.

Solution: Performance Optimization Strategy

The company implemented:

✓ **Indexing on frequently queried fields** (e.g., accountNumber, transactionDate).
✓ **Query optimization** by selecting only necessary fields.
✓ **Sharding for horizontal scaling**, distributing data across multiple servers.
✓ **Replication to improve read performance and fault tolerance**.

Results

🚀 **Query response time improved by 80%** (from 1s to 200ms).
⚡ **Database handled 5x more transactions** without performance drops.
🔍 **Real-time monitoring** helped detect and prevent performance issues.

By **optimizing queries, using indexes, and scaling MongoDB**, the company significantly **improved database efficiency**.

Exercise

1. Use explain("executionStats") on a **slow query** and analyze its performance.
2. Identify whether the query is **performing a full collection scan (COLLSCAN)**.
3. Create an **index on a frequently queried field** and re-run explain() to check performance improvements.
4. Use db.currentOp() to identify **long-running operations** in your database.

Conclusion

In this section, we explored:

✓ **How to identify performance bottlenecks using explain() and currentOp()**.

✓ **How to optimize query performance using indexes and pagination**.

✓ **How to monitor MongoDB system performance for high CPU and memory usage**.

✓ **How replication and sharding improve MongoDB scalability**.

ASSIGNMENT:

ANALYZE AN APPLICATION'S QUERY PERFORMANCE AND IMPLEMENT INDEXES FOR OPTIMIZATION

SOLUTION GUIDE: ANALYZING QUERY PERFORMANCE & IMPLEMENTING INDEXES FOR OPTIMIZATION

Step 1: Set Up a MongoDB Database & Sample Data

1.1 Configure MongoDB Connection

Create a **.env** file:

```
MONGO_URI=mongodb://localhost:27017/performanceDB
PORT=5000
```

Create **config/db.js**:

```
const mongoose = require('mongoose');
require('dotenv').config();

mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB Connected'))
.catch(err => console.error('MongoDB Connection Error:', err));

module.exports = mongoose;
```

1.2 Define a Sample Data Schema

Create **models/User.js**:

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  age: Number,
  city: String,
  isActive: Boolean,
  createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('User', userSchema);
```

✓ Stores user data with fields like name, email, age, city, and active status.

1.3 Insert Sample Data

Create **seed.js** to populate the database:

```
const mongoose = require('./config/db');
const User = require('./models/User');

const users = [];
```

```
for (let i = 1; i <= 50000; i++) {
  users.push({
    name: `User ${i}`,
    email: `user${i}@example.com`,
    age: Math.floor(Math.random() * 50) + 18,
    city: i % 2 === 0 ? "New York" : "Los Angeles",
    isActive: i % 5 !== 0
  });
}

const seedDatabase = async () => {
  try {
    await User.insertMany(users);
    console.log("50,000 sample users inserted!");
  } catch (error) {
    console.error("Error seeding database:", error);
  } finally {
    mongoose.connection.close();
  }
};

seedDatabase();
```

Run the script to insert **50,000 user records**:

node seed.js

Step 2: Analyzing Query Performance

2.1 Running a Slow Query Without Indexes

Execute the following query to find **active users in New York**:

```
db.users.find({ isActive: true, city: "New York"
}).explain("executionStats")
```

✓ **Without an index, MongoDB performs a "COLLSCAN" (collection scan), checking every document.**
✓ This results in **slow execution** due to scanning **all 50,000 records**.

Example Output (Slow Query Execution Stats)

```
{
  "executionStats": {
    "executionTimeMillis": 320,
    "totalDocsExamined": 50000,
    "nReturned": 20000
  }
}
```

✓ The query took **320 milliseconds** and scanned **50,000 documents** (inefficient).

Step 3: Implementing Indexes for Optimization

3.1 Creating an Index for Fast Filtering

Create an index on isActive and city to speed up lookups:

```
db.users.createIndex({ isActive: 1, city: 1 })
```

✓ The index **stores sorted values of isActive and city,** enabling faster lookups.

3.2 Running the Optimized Query with Index

Execute the query again:

```
db.users.find({ isActive: true, city: "New York"
}).explain("executionStats")
```

Example Output (Optimized Query Execution Stats)

```
{
  "executionStats": {
    "executionTimeMillis": 12,
    "totalDocsExamined": 20000,
    "nReturned": 20000
  }
}
```

✓ The **execution time reduced from 320ms to 12ms**.
✓ MongoDB **examined only 20,000 documents instead of 50,000,** improving performance.

Step 4: Implementing Additional Indexes for Optimization

4.1 Creating a Compound Index for Multi-Field Queries

To optimize searches by email and age:

```
db.users.createIndex({ email: 1, age: 1 })
```

✓ **Speeds up searches filtering users by email and age.**

Example: Fast Query Using Compound Index

```
db.users.find({ email: "user25000@example.com", age: 30 }).explain("executionStats")
```

✓ Uses the **compound index** instead of scanning the full collection.

4.2 Using Partial Indexes for Active Users

To index **only active users**, saving storage:

```
db.users.createIndex(
  { city: 1 },
  { partialFilterExpression: { isActive: true } }
);
```

✓ **Reduces index size** by ignoring inactive users.

Example: Querying Active Users in a City

```
db.users.find({ city: "New York", isActive: true }).explain("executionStats")
```

✓ **Faster execution, fewer indexed documents**.

Case Study: How an E-Commerce Platform Optimized Search Queries with Indexes

Background

An e-commerce platform with **1 million+ users** faced slow query performance.

Challenges

- **User search queries took over 2 seconds**.
- **Database scans increased server load**.
- **Growing data size slowed performance further**.

Solution: Implementing Indexing Strategy

✓ **Created compound indexes** for frequently queried fields (email, age).
✓ **Used partial indexes** to filter only **active users**.
✓ **Optimized indexing strategy** for high-traffic searches.

Results

- **Query execution time reduced from 2s to 50ms**.
- **Database load reduced by 60%**.
- **Improved user experience**, enabling instant search results.

This case study highlights **how indexing transforms database performance**.

Exercise

1. Create an index to **optimize searching users by name**.
2. Use explain() to compare performance **before and after indexing**.

3. Implement a **partial index** for users who registered in the last **6 months**.

Conclusion

✓ **Indexes significantly improve MongoDB query performance.**

✓ **Explain() helps analyze slow queries and optimize indexing.**

✓ **Using compound and partial indexes reduces storage and query time.**