# Software Engineering

**Submitted by:**

**Muhammad Awais (FA23-BCS-103)**

**Section:** **FA23-BCS_B**

**Submitted on:**

**October 11, 2025**

**Department of Computer Science**

**COMSATS University Islamabad**

# Database Schema (MongoDB)

We are using MongoDB because the data is mostly read-heavy, large, and semi-structured.

## Main Collections:

### books

**Fields:**
¬ _id (ObjectId)
¬ book_id (Integer)
¬ goodreads_book_id (Integer)
¬ title (String)
¬ authors (String)
¬ original_publication_year (Integer)
¬ average_rating (Float)
¬ ratings_count (Integer)
¬ image_url (String)
¬ small_image_url (String).

## Indexes Used:

Indexes improve reading speed.

### Books Collection Indexes:

1. **{title: 1, authors: 1}**
   Used for searching by title and author.
2. **{average_rating: -1}**
   Used to quickly get books with highest ratings.
3. **{book_id: 1}**
   Used for direct book lookups.

### Ratings Collection Indexes:

1. **{book_id: 1}**
   Used for finding all ratings of a book.
2. **{user_id: 1, book_id: 1}** (unique)
   Used for user-specific book ratings.

### Tags Collection Indexes:

1. **{tag_id: 1}**
   Used for tag lookups.
2. **{tag_name: 1}**
   Used for tag name searches.

### Book Tags Collection Indexes:

1. **{tag_id: 1}**
   Used for finding books by tag.
2. **{goodreads_book_id: 1}**
   Used for finding tags by book.

### To-Read Collection Indexes:

1. **{user_id: 1, book_id: 1}** (unique)
   Used for user's reading list.

These indexes reduce query time but increase storage a little.

## Why these Indexes?

¬ Title and author search becomes fast
¬ Rating queries return faster
¬ High-rating sorting becomes efficient
¬ User-specific operations are optimized
¬ Better response performance for REST API
¬ Tag-based filtering works efficiently

Without indexes, MongoDB would scan the whole database every time.

## Trade-offs / Limitations:

### Pros:

¬ Fast read performance for common queries
¬ Easy scaling with sharding
¬ Flexible schema for evolving data
¬ Good for complex aggregations
¬ Efficient for high-volume reads

### Cons:

¬ Indexes take extra storage
¬ Write performance becomes slightly slower
¬ Data is normalized across collections
¬ Some queries require joins between collections
¬ More complex query patterns needed for related data