UTM

UNIVERSITI TEKNOLOGI MALAYSIA

FACULTY OF COMPUTING

SECP2623-01 DATABASE PROGRAMMING

SEMESTER 5

2025/2026

PROJECT II

TITLE: COURT BOOKING SYSTEM

LINK PRESENTATION: https://www.youtube.com/watch?v=3YgGea3FnAI

LECTURER: DR. ROZILAWATI BINTI DOLLAH @ MD. ZAIN

NAME	MATRIC NO
ANIS SAFIYYA BINTI JANAI	A23CS0049
NURUL ASYIKIN BINTI KHAIRUL ANUAR	A23CS0162
NABIL AFLAH BOO BINTI MOHD YOSUF BOO YONG CHONG	A23CS0252
AFIF SHAQIR IRFAN BIN AQRAM	A23CS0204

TABLE OF CONTENTS

1.0 INTRODUCTION

This project focuses on the development of a Court Booking System that manages court reservations and booking records efficiently. The system uses MongoDB, a NoSQL database to store and manage user, court and booking data using a document-based data format. Using MongoDB, the system provides adaptive data storage and efficient database operations for a real-world booking environment.

1.1 Domain Description

This project is about a Court Booking System, which allows individuals to book sports courts such as basketball, badminton and futsal. The system allows users to reserve courts, pay for it and track their bookings. Through this system, users can manage their court reservations in a more organized and efficient manner. MongoDB is a NoSQL database that stores and maintains user, court and booking data on the backend. It accomplishes this via a document-based data model. This document-based solution is flexible in processing booking information and meets the needs of a real-world court booking system.

1.2 Project Objectives

The project's objectives are to manage a court booking system by utilizing MongoDB and its document-based data model to apply NoSQL database approach. The project aims to produce and arrange a number of collections that accurately represent real-world entities such as courts, reservations and users. The project also focuses on developing complete CRUD operations to provide effective data management and database manipulation. Lastly, the project demonstrates how to apply MongoDB query features such as filtering, sorting and update operations to retrieve and handle data depending on the certain conditions.

1.3 Team Members and Roles

This project was completed by a team of members, where each member was responsible for specific tasks. The allocated tasks helped ensure that the project ran smoothly and efficiently.

Name	Role	Responsibilities
Anis Safiyya Binti Janai	Indexing Specialist	Implemented single-field and compound indexes, analyzed query performance and documented optimization strategies.
Nurul Asyikin Binti Khairul Anuar	Database Developer	Designed database structure, implemented CRUD operations, handled data insertion and updates, and prepared documentation for CRUD implementation.
Nabil Aflah Boo Binti Mohd Yosuf Boo Yong	Optimization Specialist	Developed advanced sorting operations, optimized query outputs

Chong		and ensured data clarity using projections.
Afif Shaqir Irfan Bin Aqram	Aggregation Analyst	Designed and implemented aggregation pipelines and interpreted aggregation results for reporting.Created the NOSQL Data Model reporting.

2.0 NOSQL DATA MODEL

2.1 List and explanation of collections

- **2.1.1 Users Collection**

  This collection stores details for all registered customers. We keep user profiles separate so we don't have to repeat their personal info in every single booking. This way, if a user changes their phone number, we only update it in one place, and it stays consistent everywhere.

- **2.1.2 Courts Collection**

  This collection holds facility details like the court type (e.g., Badminton), location, and hourly rates. We keep this independent so admins can update prices easily without affecting historical booking records. It acts as the "master list" for our facilities.

- **2.1.2 Bookings Collection (bookings)**

  This is the main collection that records every reservation. It links a User to a Court. We decided to embed payment details directly here because payments are always checked together with the booking. This ensures that even if court prices change later, the original payment record remains accurate.

2.2 Example of JSON Documents

- **2.2.1 Users Collection**

  Figure 2.2.1 illustrates a document from the users collection representing a customer profile. It stores essential personal information for "Nurul Asyikin", including her contact number and role, identified by the unique user_id "U001".

Figure 2.2.1

- **2.2.2 Courts Collection**

  Figure 2.2.2 displays a document from the courts collection used to manage facility data. It defines a specific "Badminton" court located at "Block A" with a set hourly rate (price_per_hour) of 25, which allows the system to standardize pricing for bookings.

Figure 2.2.2

- **2.2.3 Bookings collection**
  Figure 2.2.3 shows a transaction record from the bookings collection (Booking ID "B006"). This document demonstrates a hybrid data model: it uses references to link the User ("U002") and Court ("C001"), while embedding the payment details directly within the document to ensure efficient data retrieval.

```
> db.bookings.findOne({ "booking_id": "B006" })
< {
    _id: ObjectId('696b0a99d614368b090374e7'),
    booking_id: 'B006',
    user_id: 'U002',
    court_id: 'C001',
    court_type: 'Badminton',
    booking_date: 2026-01-16T00:00:00.000Z,
    payment: [
      {
        amount: 20,
        status: 'Paid'
      }
    ],
    booking_status: 'Confirmed'
  }
```

Figure 2.2.3

2.3 Explanation of Embedding vs Referencing

We embedded payments inside bookings because they belong together. When you check a booking, you almost always need to see if it's paid. Keeping them in one document makes the system faster since we don't have to search another collection.

We used referencing for Users and Courts. Since one user can make many bookings, it doesn't make sense to copy their full profile every time. By just using user_id, we save space and avoid duplicate data.

2.4 Data Types Used

Data Type	Description	Example
ObjectId	A unique 12-byte identifier automatically generated by MongoDB.	Unique ID for every documents
String	Used for text-based data,	"U001", "Badminton",

	codes, and identifiers.	"Nurul Asyikin", "Completed"
Number	Used for numerical values involving calculations or prices.	price_per_hour: 20, amount: 160, price_per_hour: 80
Date	Used to store specific dates and times for scheduling.	new Date("2026-01-10"), created_at, booking_date
Array	A list of values or objects stored within a single field.	payment: [...] (List of payment records)
Embedded Document	A JSON object stored inside a parent document (often inside an array).	{ amount: 40, status: "Paid" } (Inside the payment array)

3.0 CRUD IMPLEMENTATION SUMMARY

This section demonstrates the implementation of CRUD operations for the Book Court Management System using the MongoDB shell on the database. The operations are executed on three main collections which are users, courts and bookings.

3.1 Create (Insert Operations)

The Create operation is used to insert new documents into the database collections.This operation ensures that the system can record new user bookings accurately and supports the subsequent retrieval and updates. The following command inserts a completed badminton court booking into the bookings collection.

```
> db.bookings.insertMany([
    {
      booking_id: "B001",
      user_id: "U001",
      court_id: "C001",
      court_type: "Badminton",
      booking_date: new Date("2026-01-10"),
      payment: [
        {
          amount: 40,
          status: "Paid"
        }
      ],
      booking_status: "Completed"
    },
```

Figure 3.1.1: Insert Booking Command in MongoDB Shell

Based on the output illustrated Figure 3.1.2 confirms that the document was successfully entered into the booking collection. The presence of an autogenerated _id indicates that MongoDB properly stored the document.

Figure 3.1.2: Display The Acknowledgement Output for The Successful Insert Operation

3.2 Read Operation

This Read operation retrieves records based on specific conditions or filters. MongoDB query operators allow to filter documents, including data contained in embedded arrays. The query below retrieves bookings with a payment amount larger than RM60. This example shows the use of the $gt operator to filter nested array fields.

Figure 3.2: Execution and Output of Booking Query

As shown in Figure 3.2, only bookings that meet the stated criteria are returned. This demonstrates MongoDB's ability to efficiently filter nested fields within array structures.

3.3 Update Operation

The Update operation updates current records with adjustments such as payment confirmation or booking approval. The following update operation changes all pending bookings to confirmed and updates their payment status to paid. The results in Figure 3.3 shows multiple documents being matched and successfully updated.

```
> db.bookings.updateMany(
    { booking_status: "Pending" },
    { $set: { "payment.$[].status": "Paid", booking_status: "Confirmed" } }
  )
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 2,
    modifiedCount: 2,
    upsertedCount: 0
  }
> db.bookings.find()
< {
    _id: ObjectId('6968c750ef5da818f266150d'),
    booking_id: 'B001',
    user_id: 'U001',
    court_id: 'C001',
    court_type: 'Badminton',
    booking_date: 2026-01-10T00:00:00.000Z,
    payment: [
      {
        amount: 40,
        status: 'Paid'
      }
    ],
    booking_status: 'Completed'
  }
```

Figure 3.3: Update Operation and Result For Pending Bookings

3.4 Delete Operation

The Delete operation removes unnecessary or invalid records from the database to maintain data accuracy. The following command deletes a cancelled booking record from the system. As presented in Figure 3.4, MongoDB confirms that one document has been deleted. This ensures that cancelled bookings do not remain in the database.

Figure 3.4: Delete Command Execution and Acknowledgement Output

4.0 ADVANCED OPERATIONS

This section shows the implementation of the advanced MongoDB operation in the court booking system. The advanced operations include index creation, aggregation pipelines, and sorting. The performance improvement and challenges encountered during the implementation are also discussed in this section.

4.1 Index Creation

Indexing improves query performance by locating the required data rather than going through each document in a collection. There are two types of index, which are single-field indexes and compound indexes. Figure 4.1.0 shows the creation of a single-field index on the booking_status field.

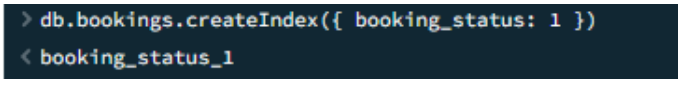

Figure 4.1.0: Single Field Index Creation

This index gives an efficient sorting of the booking collection that is filtered by the status, whether it is Confirmed, Completed, or Cancelled. The sorting of the booking are demonstrated in Figure 4.1.1, where the bookings are filtered on Confirmed status.

Figure 4.1.1: Single Field Index Usage Example

Next, the compound index involves indexing multiple fields, as shown in Figure 4.1.2. This project implements indexing on the booking_date field with the user_id and court_type fields in the bookings collection.

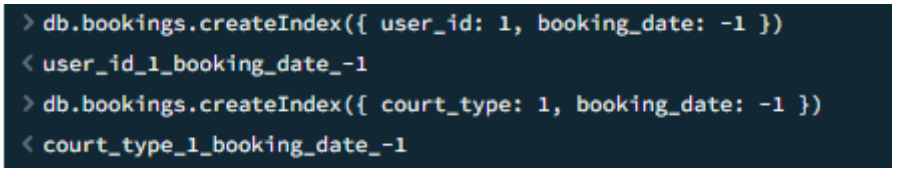

Figure 4.1.2: Compound Index Creation

Figure 4.1.3 illustrates how the index on user_id and booking_date fields optimized queries as it supports both find and sort operations. This allows data retrieval of a user's booking history that can be sorted by date of the bookings.

Figure 4.1.3: Compound Index Usage Example

As shown in Figure 4.1.4, the index on the court_type and booking_date fields eases the process of finding bookings by court types and date. For instance, this index can find all Futsal court bookings within a specific date range or sort all of them by date. This index enhanced support for the bookings collection, which perform frequents searches and monitoring.

Figure 4.1.4: Compound Index Usage Example

4.2 Aggregation Pipeline

Aggregation pipelines process data in stages to perform grouping, calculations, and transformations, similar to SQL GROUP BY .aggregate() is used to perform data processing and analysis andIt works using a pipeline, where each stage processes the output of the previous stage.

In this section, we implemented Aggregation Pipelines to perform advanced data analysis on the bookings collection. As per the project requirements, each query utilizes at least two pipeline stages such as $match, $group, $sort, and $limit to transform the raw data into meaningful insights.

The following four examples demonstrate different types of aggregation outputs, including total counts, grouping by category, top N items, and financial summaries (averages and totals).

Aggregation 1 : Top 3 Active Customers

First, we started by wanting to find out who our most frequent players are. We grouped the data by User ID to count how many bookings each person has made. We then limited the result to display only the top 3 users, which is useful for identifying regular customers.

Figure 4.2.1: Aggregation Top 3 Active Customers

The output shows the IDs of the customers with the highest number of bookings, proving that the system can successfully track user activity.

Aggregation 2 : Total Bookings by Status

For the second analysis, we looked at the overall status of bookings in the system. We grouped the bookings to count how many are currently 'Confirmed', 'Pending', or 'Cancelled'. This gives a quick snapshot of the system operation.

```
> db.bookings.aggregate([

    { $group: { _id: "$booking_status", total: { $sum: 1 } } },

    { $sort: { total: -1 } }

  ])
< {
    _id: 'Confirmed',
    total: 5
  }
```

Figure 4.2.2: Aggregation Total Bookings by Status

This result provides a summary breakdown of all bookings. A high number of confirmed bookings indicates good utilization, while many cancellations might require further investigation.

Aggregation 3 : Total Revenue by Court Type

For the third, calculating the total earnings for each type of sport. This analysis uses $match to filter for paid bookings and $group to sum up the revenue. Since the payment details are stored inside an array, we also included $unwind to process the amounts correctly.

```
> db.bookings.aggregate([

    { $match: { "payment.status": "Paid" } },

    { $unwind: "$payment" },

    { $group: { _id: "$court_type", totalRevenue: { $sum: "$payment.amount" } } },

    { $sort: { totalRevenue: -1 } }

  ])
< {
    _id: 'Futsal',
    totalRevenue: 160
  }
  {
    _id: 'Basketball',
    totalRevenue: 120
  }
  {
    _id: 'Badminton',
    totalRevenue: 60
  }
```

Figure 4.2.3: Aggregation Total Revenue by Court Type

The result lists the court types sorted by total revenue. This allows the management to clearly see which sports facility is the most profitable.

4.3 Sorting

Sorting plays an important role in transforming a raw list of database records into meaningful information by arranging the data in a specific order whether it is ascending or descending. In the Court Booking System, sorting helps to improve the user experience by allowing the customers to find the courts based on their preferences and making it easier for the administrators to prioritize their daily tasks by focusing on the urgent bookings.

Figure 4.3.1 shows the implementations of compound sort to organize the court list into a user friendly catalog. The primary sort that is applied to court_type is sorted in ascending order by grouping all courts of the same sport alphabetically. The secondary sort is also an ascending sort which is applied to the price_per_hour field to rank the courts within each category from the cheapest to the most expensive. This helps the customers to browse the available options by their favourite sports while identifying the most budget friendly court.

Figure 4.3.1: Compound Sort for Courts and Price Per Hour

Figure 4.3.2 is implemented to improve the administrative workflow by focusing on the priority tasks. By sorting the booking_status in descending order, the system will place active statuses like confirmed at the top of the list. Besides, to ensure the current booking information, a secondary descending sort is applied to the booking_date. This ensures that the newest and urgent booking requests are always visible first.

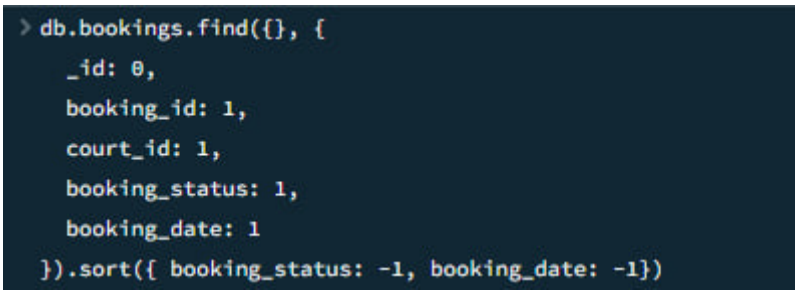

Figure 4.3.2: Compound Sort for Booking Status and Booking Date

Figure 4.3.3: Output of Compound Sort for Booking Status and Booking Date

Figure 4.3.4 focuses the sort implementation by organizing all booking records around individual customer activity. The primary sort uses user_id in ascending order to group every transaction that has been made by a specific user together. The secondary descending sort is then applied to the booking_date to display their most recent booking at the top. This allows the staff to quickly review customers' recent visits or sport preferences to provide a more personalized service.

Figure 4.3.4: Compound Sort for User Id and Booking Date

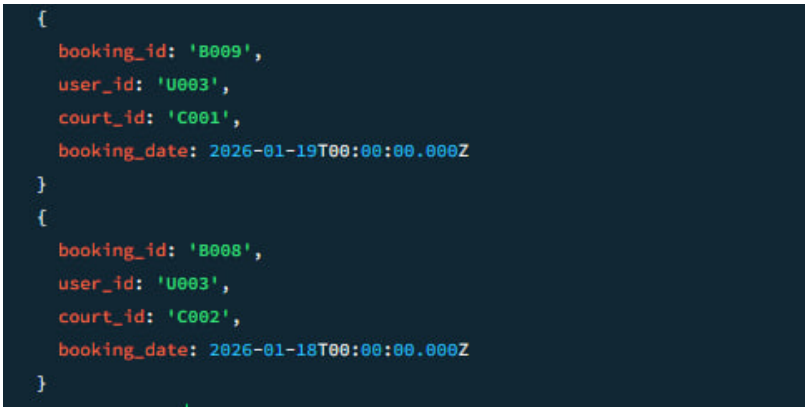

Figure 4.3.5: Output of Compound Sort for User Id and Booking Date

4.4 Performance Improvements

- **Improve the query efficiency**

By using single-field indexes and composite indexes, MongoDB can obtain relevant reservation records faster, thus avoiding searching the entire collection, especially some commonly used columns, such as reservation status, user ID and venue type.

- **Optimized Filtering and Sorting Operations**

Due to the establishment of a composite index between user_id and booking_date and court_type and booking_date, MongoDB can complete filtering and sorting in one operation.

This improves the response speed of users' booking history and venue-specific scheduling inquiries.

- **Enhance management ability.**

Administrators can quickly view reservations according to priority, date or venue type by indexing and sorting queries, so as to make wiser operational decisions.

- **Reduced Data Processing**

By eliminating redundant fields, the projection in the sorting query reduces the amount of data returned, improves query performance and reduces memory consumption.

4.5 Challenges Encountered

- **Effective composite index design**

In order to determine the correct field order of the composite index and match it with the actual query mode, careful planning must be carried out. If the fields are not arranged correctly, MongoDB may not be able to make full use of the index.

- **Blocked sorting may occur for fields that have not been indexed.**

As the size of the reservation collection grows, sorting unindexed fields may cause blocking the sorting, thus taking up more memory and processing time.

- **Balancing scalability and performance**

For example, whether adding a new index or changing an existing index, the system needs to strike a balance between maintaining the scalability of future data growth and improving the current query performance.

5.0 SECURITY & LIMITATIONS

This section outlines the measures that have been taken to protect the integrity of the Court Booking System and technical boundaries of current implementation.

1. **Basic Access Control**

   The access control in this system is managed through data visibility and most of the query in this system uses projections like { id: 0 } to hide internal MongoDB ObjectIDs. This ensures users and administrators can only see the relevant data by preventing the exposure of system metadata. The system also explicitly defines roles during user creation to ensure data has been categorized from the beginning. Administrative queries for revenue or booking history have been filtered to specific fields to ensure that all of the sensitive information is not exposed in general reports.

2. **Injection Risk**

   By using MongoDB operators such as $gt, $or, and $set, the system gets to avoid raw string concatenation. This helps to protect the database from NoSQL injection attacks if there is a user trying to bypass the logic system. Moreover, all sorting methods in the system use fixed BSON objects which prevents outsiders from manipulating the query order to reveal unauthorized data. The implementation of operators like $inc $push ensures that updates are performed on the intended fields and data types to reduce the risk of accidental data corruption.

3. **Database Constraints or Limitations**

   Sorting large datasets such as bookings collection will require a maximum of 100MB memory usage for in-memory sorts if the indexes have not been created. There are also nested array limitations in this system as the payments are stored as arrays which sort by its amount, it will rank the booking information based on individual elements in the array rather than a calculated total sum for that booking. Furthermore, for some of the fields that have not been indexed, the database needs to perform a blocking sort which will lead to latency as the volume of booking increases. While indexes improve search speed for “Confirmed” bookings or any specific “User IDs”, it will add a slight

delay to insertMany and updateMany operations as the index needs to update simultaneously with the data.

6.0 TEAMWORK

Anis Safiyya Binti Janai - Indexing Specialist

As an index specialist of this project, I am responsible for creating and implementing index algorithms to improve the efficiency of database queries in the court booking system. My main tasks include determining which fields in the booking collection have the highest query frequency and creating the necessary indexes to enhance typical queries. For example, in order to improve the query for obtaining booking information based on the current status, such as confirmed bookings, a single field index was added to the booking_status field. In order to facilitate the efficient screening and sorting of booking records, compound indexes are also created based on user_id and booking_date, as well as court_type and booking_date. These compound indexes aim to improve the efficiency of date-sorted queries and user-based booking history queries, and the applications are demonstrated through the search and sorting functions.

Nurul Asyikin Binti Khairul Anuar - Database Developer

As the project's database developer, I was in charge of designing and implementing the Court Booking System's core database functionality. My key contribution was to use the MongoDB shell to create and manage CRUD operations for users, courts and booking collections. I ensure that each operation functioned correctly and met the project requirement, especially when dealing with embedded documents and referenced fields. I also helped arrange the project documentation, particularly the CRUD implementation by selecting specific examples, explaining it and appropriately labelling screenshots.

Nabil Aflah Boo - Optimization Specialist

As an Optimization Specialist, I focused on implementing advanced sorting operations to ensure the database output was both professional and actionable for different types of users. I also developed a multi-level catalog sort for the courts collection that organized the courts by the sports type and its price which allowed the customers to easily identify the most budget friendly options within their preferred sports. In addition, I created an operational priority sort for the bookings collection that ranked all records by status and booking date, which assists

administrators to identify the urgent booking that require immediate attention. By applying projections to all sorting demonstrations, I also ensured that sensitive internal metadata will remain hidden to optimize the system for both security and data clarity.

Afif Shaqir Irfan Bin Aqram - Aggregation Analyst

My primary role was acting as the Aggregation Analyst and overseeing the NoSQL Data Modeling. I was responsible for structuring the database collections (Users, Courts, and Bookings) and deciding when to use embedding versus referencing to ensure system efficiency. I also developed the aggregation scripts to extract meaningful insights, such as calculating total revenue by court type and identifying top customers. Furthermore, I authored the technical documentation for Sections 2.0 and 4.0, explaining the logic behind our schema design and the results of our data analysis.

7.0 CONCLUSION

Through the development of the Court Booking System, each individual in the team gets to learn on how to design and manage a flexible NoSQL database that effectively handles diverse data types such as nested arrays, dates, and strings. We gained practical experience in transforming the raw data into useful insights for business intelligence by using aggregation pipelines such as calculating the total booking per user and tracking revenue by court types. We also learned the importance of using safe query behaviours by applying projections and the native BSON operators to avoid all chances of information leakage and to protect the database against the common security issues.

As a team, dealing with complex data structures and sorting logic proved to be a major problem in the project, especially when it came to technical difficulties. One of the challenges was sorting nested amounts of payment since the system orders documents in terms of specific components of the payment array rather than an overall amount. In addition, we need to construct the compound sort for bookings very carefully to ensure that active statuses were prioritized first, and this required precise control for descending orders.

To improve the system, we recommend using JSON schema validation in order to increase the restrictions of the data entry and prevent the collection of the wrong type of data. The real-time change streams could also improve the system by providing instant notification to

the users in case of an update on the status of a booking. Lastly, implementing a more advanced aggregation strategy for payments would allow the system to sort the results by the total calculated revenue per booking rather than individual payment entries, which provides more accurate financial reporting.