

Design and develop SQL DDL Statements which demonstrate the use of SQL objects such as table,view,index,sequence ,synonym,different constraints



Here's an example of SQL DDL statements that create various SQL objects, demonstrating the use of tables, views, indexes, sequences, synonyms, and constraints.

1. Create Table with Constraints

sql

```
CREATE TABLE employees (  
    employee_id NUMBER(5) PRIMARY KEY,  
    first_name VARCHAR2(50) NOT NULL,  
    last_name VARCHAR2(50) NOT NULL,  
    email VARCHAR2(100) UNIQUE,  
    phone_number VARCHAR2(15),  
    hire_date DATE NOT NULL,  
    job_id VARCHAR2(10) NOT NULL,  
    salary NUMBER(10, 2) CHECK (salary > 0),  
    manager_id NUMBER(5),  
    department_id NUMBER(5),  
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id),  
    FOREIGN KEY (department_id) REFERENCES departments(department_id)  
);  
  
CREATE TABLE departments (  
    department_id NUMBER(5) PRIMARY KEY,  
    department_name VARCHAR2(100) NOT NULL,  
    location_id NUMBER(5)  
);
```

2. Create a View

Create a view to simplify the query of employee details:

sql

```
CREATE VIEW employee_details AS
SELECT e.employee_id, e.first_name, e.last_name, e.email,
d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

3. Create an Index

Create an index on the **salary** column in the **employees** table to speed up searches by salary:

sql

```
CREATE INDEX idx_employee_salary ON employees(salary);
```

4. Create a Sequence

Create a sequence to generate unique employee IDs:

sql

```
CREATE SEQUENCE emp_id_seq  
START WITH 1  
INCREMENT BY 1  
NOCACHE  
NOCYCLE;
```

You can use this sequence to automatically populate `employee_id` values in inserts.

5. Create a Synonym

Create a synonym to make it easier to refer to the `employee_details` view:

sql

```
CREATE SYNONYM emp_details FOR employee_details;
```

Now, `emp_details` can be used in place of `employee_details` in queries.

6. Demonstrate Different Constraints

The `employees` table includes a variety of constraints:

- **PRIMARY KEY** (`employee_id`): Uniquely identifies each employee.
- **UNIQUE** (`email`): Ensures that no two employees have the same email.
- **CHECK** (`salary > 0`): Ensures the salary is positive.
- **NOT NULL** (e.g., `first_name`, `last_name`, `hire_date`, `job_id`): Ensures these fields must be filled in.
- **FOREIGN KEY** (`manager_id`, `department_id`): Links `employees` to other tables.

7. Insert Data Using Sequence

Here's how to use the sequence in an insert statement:

sql

```
INSERT INTO employees (employee_id, first_name, last_name, email,  
phone_number, hire_date, job_id, salary, manager_id, department_id)  
VALUES (emp_id_seq.NEXTVAL, 'John', 'Doe', 'johndoe@example.com', '123-  
456-7890', SYSDATE, 'DEV', 75000, NULL, 1);
```

Putting It All Together

These DDL statements provide a foundational database structure using SQL objects like tables, views, indexes, sequences, synonyms, and constraints to enforce data integrity and optimize performance.

write at least 10 sqlqueries on thesuitable database application using
sql dml statements



Here are 10 SQL DML queries that perform various operations on the **employees** and **departments** tables from the previous example. These queries will demonstrate typical operations like inserting, updating, deleting, and retrieving data, as well as joining tables and using aggregate functions.

1. Insert Data into the **departments** Table

sql

```
INSERT INTO departments (department_id, department_name, location_id)  
VALUES (1, 'Sales', 101);
```

2. Insert Data into the **employees** Table Using Sequence

sql

```
INSERT INTO employees (employee_id, first_name, last_name, email,
phone_number, hire_date, job_id, salary, manager_id, department_id)
VALUES (emp_id_seq.NEXTVAL, 'Jane', 'Smith', 'janesmith@example.com',
'987-654-3210', SYSDATE, 'HR', 60000, NULL, 1);
```

3. Update an Employee's Salary

Increase the salary of the employee with `employee_id = 1` by 10%.

sql

```
UPDATE employees
SET salary = salary * 1.10
WHERE employee_id = 1;
```

4. Delete an Employee Record

Delete the employee with `employee_id = 2` from the `employees` table.

sql

```
DELETE FROM employees
WHERE employee_id = 2;
```

5. Retrieve All Employees with a Specific Job ID

Get details of all employees with the job ID "HR".

sql

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE job_id = 'HR';
```

6. Count the Number of Employees in Each Department

Count the total number of employees in each department.

sql

```
SELECT department_id, COUNT(*) AS total_employees
FROM employees
GROUP BY department_id;
```

7. Retrieve Employees with Salary Greater Than the Average

Get details of employees who earn more than the average salary in the company.

sql

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

8. Find the Highest Paid Employee in Each Department

Retrieve the highest salary for each department along with the employee details.

sql

```
SELECT e.employee_id, e.first_name, e.last_name, e.salary,
e.department_id
FROM employees e
JOIN (
    SELECT department_id, MAX(salary) AS max_salary
    FROM employees
    GROUP BY department_id
) dept_max ON e.department_id = dept_max.department_id AND e.salary =
dept_max.max_salary;
```

9. Retrieve Department Names with Employee Counts

Get a list of all departments and the number of employees in each, including departments with no employees.

sql

```
SELECT d.department_name, COUNT(e.employee_id) AS employee_count
FROM departments d
LEFT JOIN employees e ON d.department_id = e.department_id
GROUP BY d.department_name;
```

10. Retrieve Employee Details Using a Synonym

Retrieve employee details from the `employee_details` view (or `emp_details` synonym).

sql

```
SELECT * FROM emp_details;
```

These queries cover a range of operations, including data manipulation, conditional filtering, joins, aggregations, and subqueries, providing practical examples for managing and analyzing data in a relational database.

writre about ddl,dml,tql



In SQL (Structured Query Language), commands are divided into several categories based on their functionality. The primary categories are:

1. **DDL (Data Definition Language)**
2. **DML (Data Manipulation Language)**
3. **TQL (Transaction Query Language)**

Here's a detailed explanation of each:

1. DDL (Data Definition Language)

DDL is used to define and manage the structure of database objects like tables, indexes, views, and more. DDL commands create, modify, and delete these database structures.

Common DDL Commands:

- **CREATE:** Creates a new database object, like a table, view, or index.
Example: `CREATE TABLE employees (employee_id NUMBER, first_name VARCHAR2(50), salary NUMBER);`
- **ALTER:** Modifies an existing database object by adding or changing columns, constraints, etc.
Example: `ALTER TABLE employees ADD department_id NUMBER;`
- **DROP:** Deletes a database object, such as a table or view.
Example: `DROP TABLE employees;`
- **TRUNCATE:** Removes all records from a table but keeps its structure intact.
Example: `TRUNCATE TABLE employees;`
- **RENAME:** Renames an existing database object.
Example: `RENAME employees TO staff;`

DDL commands are **auto-committed**, meaning any changes made by DDL statements are automatically saved in the database.

2. DML (Data Manipulation Language)

DML is used to work with data inside the database. It allows you to insert, update, delete, and retrieve data from tables.

Common DML Commands:

- **INSERT:** Adds new records to a table.
Example: `INSERT INTO employees (employee_id, first_name, salary) VALUES (1, 'John', 50000);`
- **UPDATE:** Modifies existing records in a table.
Example: `UPDATE employees SET salary = salary * 1.10 WHERE employee_id = 1;`
- **DELETE:** Removes records from a table based on specified conditions.
Example: `DELETE FROM employees WHERE employee_id = 1;`
- **SELECT:** Retrieves data from one or more tables.
Example: `SELECT * FROM employees WHERE salary > 40000;`

Unlike DDL commands, DML commands **require an explicit commit** (if within a transaction) to save changes permanently.

3. TQL (Transaction Query Language)

TQL, often referred to as **TCL (Transaction Control Language)**, is used to manage database transactions. These commands control the changes made by DML commands, ensuring the integrity and consistency of data.

Common TQL Commands:

- **COMMIT**: Saves all changes made in the current transaction to the database permanently.
Example: `COMMIT;`
- **ROLLBACK**: Undoes all changes made in the current transaction.
Example: `ROLLBACK;`
- **SAVEPOINT**: Sets a point within a transaction that you can later roll back to.
Example: `SAVEPOINT save1;`
- **SET TRANSACTION**: Sets transaction properties, like setting the isolation level.
Example: `SET TRANSACTION READ ONLY;`

TQL commands are essential in ensuring **atomicity and consistency** in databases, particularly in complex operations involving multiple DML statements.

Summary

- **DDL** focuses on defining and altering the structure of the database (e.g., creating tables).
- **DML** deals with manipulating the data within these structures (e.g., adding or updating records).
- **TQL** (or TCL) manages transactions, providing control over saving and discarding changes to the data.

Each plays a critical role in ensuring that data is both accurately stored and safely managed within a relational database system.

wrire abut dcl



DCL (Data Control Language) is a category of SQL commands focused on managing access to the database. DCL commands are used to control the permissions of users, ensuring only authorized users can access, modify, or perform specific actions on the database objects.

Common DCL Commands

1. **GRANT:** Gives specific permissions to a user or role. This allows the user to perform actions such as querying tables, modifying data, or executing procedures, depending on the granted privileges.

Example:

```
sql
```

```
GRANT SELECT, INSERT ON employees TO john_doe;
```

This command grants the user `john_doe` permission to select and insert data into the `employees` table.

2. **REVOKE:** Removes specific permissions from a user or role. This command restricts users from performing certain actions on database objects if they no longer have the necessary permissions.

Example:

```
sql
```

```
REVOKE INSERT ON employees FROM john_doe;
```

This command removes the `INSERT` privilege on the `employees` table from the user `john_doe`, preventing them from adding new records to the table.

Common Privileges in DCL

DCL commands grant or revoke a variety of privileges, including:

- **Object Privileges:** Specific to database objects, such as tables or views. These include privileges like:
 - **SELECT:** Permission to read data from a table or view.
 - **INSERT:** Permission to add new rows to a table.
 - **UPDATE:** Permission to modify existing rows in a table.
 - **DELETE:** Permission to remove rows from a table.
 - **EXECUTE:** Permission to run a stored procedure or function.
- **System Privileges:** Broader permissions that apply to actions across the database, such as:
 - **CREATE SESSION:** Allows a user to connect to the database.
 - **CREATE TABLE:** Allows a user to create tables in the database.
 - **CREATE VIEW:** Allows a user to create views.

Examples of Using DCL for Security and Access Control

- **Granting All Privileges on a Table to a User:**

sql

```
GRANT ALL PRIVILEGES ON employees TO admin_user;
```

This grants full control over the **employees** table to the user **admin_user**.

- **Revoking Access to a Database Object:**

sql

```
REVOKE SELECT, UPDATE ON departments FROM user123;
```

This revokes the **SELECT** and **UPDATE** privileges on the **departments** table from **user123**.

Importance of DCL in Database Security

DCL plays a critical role in enforcing **data security and access control** by:

- Ensuring that only authorized users can access or modify data.
- Minimizing the risk of unauthorized data exposure or alteration.
- Allowing fine-grained control over each user's capabilities, limiting potential errors or misuse.

Key Points to Remember

- **DCL commands are auto-committed:** Changes made with **GRANT** or **REVOKE** are saved automatically and cannot be rolled back.
- DCL enhances security by controlling who has access to sensitive information and database objects.

DCL is essential for maintaining robust security within any SQL-based database system, especially in multi-user environments where access to data must be carefully managed.

2/2

Write a PL/SQL code block to calculate the area of a circle for a value of radius varying from 5 to a specified value. Store the radius and the corresponding values of calculated area in an empty table named **areas**, consisting of two columns, radius and area.



Here's a PL/pgSQL (PostgreSQL) code block to calculate the area of a circle for radius values ranging from 5 to a specified value, and store the results in an **areas** table.

We'll first create the **areas** table, then write a PL/pgSQL block to populate it with the radius and corresponding area for each radius value.

Step 1: Create the **areas** Table

sql

```
CREATE TABLE areas (  
    radius NUMERIC,  
    area NUMERIC  
);
```

Step 2: Write the PL/pgSQL Block

This code block calculates the area of a circle for radius values from 5 up to a specified value and inserts the radius and area into the **areas** table.

plpgsql

```
DO  
$$  
DECLARE  
    r NUMERIC := 5;          -- Starting radius  
    max_radius NUMERIC := 10; -- Replace 10 with your desired maximum  
radius  
    calculated_area NUMERIC;  
BEGIN  
    -- Loop through radius values from 5 to the maximum radius  
    WHILE r <= max_radius LOOP  
        -- Calculate the area of the circle  
        calculated_area := 3.14159 * r * r;  
  
        -- Insert the radius and area into the areas table  
        INSERT INTO areas (radius, area) VALUES (r, calculated_area);  
  
        -- Increment the radius for the next iteration  
        r := r + 1;  
    END LOOP;  
END  
$$;
```

Explanation

- The **DO** block allows us to execute anonymous PL/pgSQL code.
- We initialize the radius **r** at 5 and define a **max_radius** for the upper limit.
- The **WHILE** loop iterates from **r = 5** to **max_radius**.
- Within the loop:
 - We calculate the area using the formula πr^2 .
 - Insert the **radius** and **calculated_area** into the **areas** table.
 - Increment **r** by 1 to continue until reaching **max_radius**.

This code will populate the **areas** table with radius values and corresponding area calculations for each radius from 5 to **max_radius**. Adjust **max_radius** as needed to extend or reduce the range.

Write a PL/SQL block to use procedure created with above requirement.

Stud_Marks(name, total_marks) Class)



To achieve the student categorization based on marks in PL/SQL, we'll follow these steps:

1. Create a table **Stud_Marks** to store the student's name, marks, and category.
2. Create a procedure named **proc_rate** that categorizes students based on their **total_marks** and updates the category in the **Stud_Marks** table.
3. Write a PL/SQL block to insert sample data into **Stud_Marks** and then call the **proc_rate** procedure to categorize each student.

Step 1: Create the **Stud_Marks** Table

sql

```
CREATE TABLE Stud_Marks (
  name VARCHAR2(50),
  total_marks NUMBER,
  category VARCHAR2(30)
);
```

Step 2: Create the **proc_rate** Procedure

This procedure will determine the category of each student based on their **total_marks** and update the **category** column accordingly.

sql

```
CREATE OR REPLACE PROCEDURE proc_rate IS
BEGIN
    -- Loop through each student in the table and assign a category
    based on total_marks
    FOR student IN (SELECT name, total_marks FROM Stud_Marks) LOOP
        -- Determine category based on marks and update the table
        UPDATE Stud_Marks
        SET category = CASE
            WHEN student.total_marks > 990 THEN 'Distinction'
            WHEN student.total_marks BETWEEN 900 AND 989 THEN 'First
Class'
            WHEN student.total_marks BETWEEN 825 AND 899 THEN 'Higher
Second Class'
            ELSE 'No Category'
        END
        WHERE name = student.name;
    END LOOP;
END;
```

Step 3: PL/SQL Block to Populate Data and Use the **proc_rate** Procedure

sql

```
BEGIN
  -- Insert sample data into Stud_Marks table
  INSERT INTO Stud_Marks (name, total_marks) VALUES ('Alice', 995);
  INSERT INTO Stud_Marks (name, total_marks) VALUES ('Bob', 920);
  INSERT INTO Stud_Marks (name, total_marks) VALUES ('Charlie', 870);
  INSERT INTO Stud_Marks (name, total_marks) VALUES ('David', 810);

  -- Commit the inserts
  COMMIT;

  -- Call the proc_rate procedure to categorize students
  proc_rate;

  -- Display the results
  FOR record IN (SELECT * FROM Stud_Marks) LOOP
    DBMS_OUTPUT.PUT_LINE('Name: ' || record.name ||
                          ', Total Marks: ' || record.total_marks ||
                          ', Category: ' || record.category);
  END LOOP;
END;
/
```

Explanation

- **Table Creation:** `Stud_Marks` table has columns for the student's `name`, `total_marks`, and `category`.
- **Procedure (`proc_rate`):** Loops through each student in `Stud_Marks` and categorizes them based on `total_marks`. The `UPDATE` statement uses a `CASE` expression to determine and set the correct `category`.
- **PL/SQL Block:** Inserts sample data, commits it, calls `proc_rate` to assign categories, and finally displays each student's details using `DBMS_OUTPUT`.

After running this, each student in the `Stud_Marks` table will have an assigned category based on their `total_marks`. The output will display the categorized records for each student.

Cursors: (All types: Implicit, Explicit, Cursor Loop, Parameterized Cursor)
Write a PL/SQL block of code using parameterized cursor that will merge the data available in the newly created table `N_RollCall` with the data

available in the table O_RollCall. If the the first table already exist in thc second table then that data should be skipped.



In PL/SQL, we can use a parameterized cursor to merge data from N_RollCall (new table) with O_RollCall (old table). Here, the objective is to insert records from N_RollCall into O_RollCall only if they do not already exist.

Let's assume both tables have the same structure with columns student_id and student_name.

Step 1: Create the Tables

sql

```
CREATE TABLE O_RollCall (  
    student_id NUMBER PRIMARY KEY,  
    student_name VARCHAR2(50)  
);  
  
CREATE TABLE N_RollCall (  
    student_id NUMBER PRIMARY KEY,  
    student_name VARCHAR2(50)  
);
```

Step 2: Insert Sample Data into O_RollCall and N_RollCall

sql

```
-- Insert sample data into O_RollCall
INSERT INTO O_RollCall (student_id, student_name) VALUES (1, 'Alice');
INSERT INTO O_RollCall (student_id, student_name) VALUES (2, 'Bob');

-- Insert sample data into N_RollCall
INSERT INTO N_RollCall (student_id, student_name) VALUES (2, 'Bob'); --
Duplicate
INSERT INTO N_RollCall (student_id, student_name) VALUES (3, 'Charlie');
INSERT INTO N_RollCall (student_id, student_name) VALUES (4, 'David');

COMMIT;
```

Step 3: Create a PL/SQL Block with a Parameterized Cursor

In this block, a parameterized cursor will be used to check each `student_id` in `N_RollCall`. If a student ID does not exist in `O_RollCall`, it will be inserted; otherwise, it will be skipped.

plpgsql

```
DECLARE
    -- Define the parameterized cursor to retrieve student data from
    N_RollCall
    CURSOR new_students(p_student_id NUMBER) IS
        SELECT student_id, student_name
        FROM N_RollCall
        WHERE student_id = p_student_id;

    -- Variables to hold data from the cursor
    v_student_id NUMBER;
    v_student_name VARCHAR2(50);

BEGIN
    -- Loop through each student in N_RollCall
    FOR student IN (SELECT student_id, student_name FROM N_RollCall)
    LOOP
        -- Open the cursor with the current student_id as a parameter
        OPEN new_students(student.student_id);

        -- Fetch the data into variables
        FETCH new_students INTO v_student_id, v_student_name;

        -- Check if the student_id already exists in O_RollCall
        IF NOT EXISTS (SELECT 1 FROM O_RollCall WHERE student_id =
v_student_id) THEN
            -- Insert into O_RollCall if student_id does not exist
            INSERT INTO O_RollCall (student_id, student_name)
            VALUES (v_student_id, v_student_name);
        END IF;

        -- Close the cursor
        CLOSE new_students;
    END LOOP;

    -- Commit the changes to O_RollCall
    COMMIT;
END;
/
```

Explanation

1. **Parameterized Cursor:** `new_students` takes a parameter `p_student_id` and retrieves data for each student ID in `N_RollCall`.

2. **Loop through N_RollCall:** For each student in `N_RollCall`, the cursor checks if the student's ID exists in `O_RollCall`.
3. **Condition Check:** If the `student_id` does not exist in `O_RollCall`, it inserts the student's data into `O_RollCall`; otherwise, it skips the entry.
4. **Commit:** After looping through all entries, the changes are committed to ensure data persistence.

Result

After running this code, `O_RollCall` will contain only unique `student_id` entries from both tables, avoiding duplicates.

rite a database trigger on Library table. The System should keep track Of the records that are eing updated or deleted, The old value of updated or deleted records shouldbe added in .library Audit table.



To create a database trigger that tracks updates and deletions on the `Library` table, we'll use an `AFTER UPDATE OR DELETE` trigger. This trigger will log the old values of the updated or deleted records into a `Library_Audit` table.

Step 1: Create the `Library` Table and `Library_Audit` Table

The `Library` table stores the original records, while `Library_Audit` keeps a log of any records that are updated or deleted, along with a timestamp.

Library Table

Let's assume the `Library` table has columns like `book_id`, `title`, `author`, and `category`.

sql

```
CREATE TABLE Library (  
    book_id NUMBER PRIMARY KEY,  
    title VARCHAR2(100),  
    author VARCHAR2(100),  
    category VARCHAR2(50)  
);
```

Library_Audit Table

The **Library_Audit** table will store old values of records when they are updated or deleted. It will include columns for all original data fields, along with the operation type and a timestamp.

sql

```
CREATE TABLE Library_Audit (  
    audit_id NUMBER PRIMARY KEY,  
    book_id NUMBER,  
    title VARCHAR2(100),  
    author VARCHAR2(100),  
    category VARCHAR2(50),  
    operation_type VARCHAR2(10),  
    operation_time TIMESTAMP  
);
```

Step 2: Create the Trigger on **Library** Table

This trigger will fire after an **UPDATE** or **DELETE** operation on the **Library** table and insert the old values into the **Library_Audit** table.

sql

```
CREATE OR REPLACE TRIGGER trg_library_audit
AFTER UPDATE OR DELETE ON Library
FOR EACH ROW
BEGIN
    -- Insert old values into Library_Audit when a row is updated or
    deleted
    INSERT INTO Library_Audit (
        audit_id,
        book_id,
        title,
        author,
        category,
        operation_type,
        operation_time
    )
    VALUES (
        -- Assume we have a sequence named audit_seq for generating
        audit_id
        audit_seq.NEXTVAL,
        :OLD.book_id,
        :OLD.title,
        :OLD.author,
        :OLD.category,
        CASE
            WHEN DELETING THEN 'DELETE'
            WHEN UPDATING THEN 'UPDATE'
        END,
        SYSTIMESTAMP
    );
END;
/
```

Explanation

- **Trigger Type:** **AFTER UPDATE OR DELETE** ensures that the trigger activates only after the record is updated or deleted from the **Library** table.
- **FOR EACH ROW:** Specifies that the trigger should execute for each affected row.
- **Audit Insertion:**
 - **:OLD** is used to refer to the values before the update or delete operation.
 - The **operation_type** field is populated based on the type of operation (**UPDATE** or **DELETE**).
 - **SYSTIMESTAMP** logs the exact time of the operation.

Step 3: Test the Trigger

1. Insert sample data into the **Library** table.
2. Perform **UPDATE** and **DELETE** operations on **Library** to see entries in **Library_Audit**.

Sample Testing

sql

```
-- Insert data into Library
INSERT INTO Library (book_id, title, author, category) VALUES (1, 'The
Great Gatsby', 'F. Scott Fitzgerald', 'Fiction');
INSERT INTO Library (book_id, title, author, category) VALUES (2,
'1984', 'George Orwell', 'Dystopian');
COMMIT;

-- Update a record in Library
UPDATE Library SET title = 'The Great Gatsby - Revised Edition' WHERE
book_id = 1;

-- Delete a record from Library
DELETE FROM Library WHERE book_id = 2;

-- View the contents of Library_Audit
SELECT * FROM Library_Audit;
```

Result

After the update and delete operations, **Library_Audit** will contain records showing the old values from **Library** along with the operation type and timestamp. This setup allows tracking of historical changes in the **Library** table.

) MongoDB Queries: Design and Develop MongoDB Queries using CRUD operations. (Use CRUD operations, SAVE method, logical operators etc.)



Here's a collection of MongoDB queries demonstrating CRUD operations (Create, Read, Update, Delete), the `save()` method, and logical operators on a sample `Library` collection.

Step 1: Sample Document Structure

Suppose our `Library` collection stores documents with the following structure:

json

```
{
  "_id": ObjectId("..."),
  "book_id": 1,
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "category": "Fiction",
  "available_copies": 5,
  "publication_year": 1925
}
```

MongoDB CRUD Operations and Queries

1. Create Documents

- Insert a single document:

javascript

```
db.Library.insertOne({
  book_id: 1,
  title: "The Great Gatsby",
  author: "F. Scott Fitzgerald",
  category: "Fiction",
  available_copies: 5,
  publication_year: 1925
});
```

- Insert multiple documents:

javascript

```
db.Library.insertMany([
  {
    book_id: 2,
    title: "1984",
    author: "George Orwell",
    category: "Dystopian",
    available_copies: 8,
    publication_year: 1949
  },
  {
    book_id: 3,
    title: "To Kill a Mockingbird",
    author: "Harper Lee",
    category: "Fiction",
    available_copies: 7,
    publication_year: 1960
  }
]);
```

2. Read Documents

- Find all books:

javascript

```
db.Library.find({});
```

- Find a book by title:

javascript

```
db.Library.find({ title: "1984" });
```

- Use logical operators to find books in the "Fiction" category and published before 1960:

javascript

```
db.Library.find({
  $and: [
    { category: "Fiction" },
    { publication_year: { $lt: 1960 } }
  ]
});
```

- Find books that are either "Fiction" or have more than 6 available copies:

javascript

```
db.Library.find({
  $or: [
    { category: "Fiction" },
    { available_copies: { $gt: 6 } }
  ]
});
```

3. Update Documents

- Update a specific book's available copies using `updateOne`:

javascript

```
db.Library.updateOne(  
  { book_id: 1 },  
  { $set: { available_copies: 4 } }  
);
```

- Update multiple documents to change all "Dystopian" category books to "Classic Dystopian":

javascript

```
db.Library.updateMany(  
  { category: "Dystopian" },  
  { $set: { category: "Classic Dystopian" } }  
);
```

- Using `save()` to replace a document:

javascript

```
let book = db.Library.findOne({ book_id: 2 });  
book.title = "1984 - Revised Edition";  
db.Library.save(book);
```

4. Delete Documents

- Delete a single book by `book_id`:

javascript

```
db.Library.deleteOne({ book_id: 3 });
```

- Delete all books in the "Dystopian" category:

javascript

```
db.Library.deleteMany({ category: "Classic Dystopian" });
```

5. Additional Queries with Logical Operators

Find books with more than 5 copies that are either by "George Orwell" or "Fiction" category:

javascript

```
db.Library.find({
  $and: [
    { available_copies: { $gt: 5 } },
    { $or: [ { author: "George Orwell" }, { category: "Fiction" } ] }
  ]
});
```

Summary of MongoDB CRUD Operations

- **Create:** `insertOne()`, `insertMany()`
- **Read:** `find()`, `findOne()`
- **Update:** `updateOne()`, `updateMany()`, `save()`
- **Delete:** `deleteOne()`, `deleteMany()`

These examples cover basic CRUD operations, the use of logical operators (`$and`, `$or`), and the `save()` method for replacing documents in MongoDB.

MongoDB - Aggregation and Indexing: Design and Develop MongoDB
Queries using aggregation and indexing with suitable example using
MongoDB,



Here's an overview of MongoDB queries using **aggregation** and **indexing** with examples on a sample **Library** collection.

Sample **Library** Collection Document Structure

json

```
{
  "_id": ObjectId("..."),
  "book_id": 1,
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "category": "Fiction",
  "available_copies": 5,
  "publication_year": 1925,
  "ratings": [
    { "user_id": 101, "rating": 4 },
    { "user_id": 102, "rating": 5 }
  ]
}
```

This collection stores data about books, including the `book_id`, `title`, `author`, `category`, `available_copies`, `publication_year`, and `ratings` (which is an array of ratings provided by users).

1. Aggregation Examples

Aggregation operations allow us to perform data analysis on MongoDB collections. Below are some common aggregation queries using the `Library` collection.

a) Calculate the Average Rating of Each Book

To calculate the average rating for each book, we can use the `$unwind` stage to flatten the `ratings` array, followed by `$group` to calculate the average.

javascript

```
db.Library.aggregate([
  { $unwind: "$ratings" },
  {
    $group: {
      _id: "$title",
      avg_rating: { $avg: "$ratings.rating" }
    }
  }
]);
```

b) Count Books in Each Category

To find the number of books in each category, we can use `$group` by `category` and then count the documents in each group.

javascript

```
db.Library.aggregate([
  {
    $group: {
      _id: "$category",
      total_books: { $sum: 1 }
    }
  }
]);
```

c) Find Top 3 Most Recently Published Books

To get the top 3 most recent books, we can use `$sort` on `publication_year` in descending order and then `$limit` to get only the top 3 results.

javascript

```
db.Library.aggregate([
  { $sort: { publication_year: -1 } },
  { $limit: 3 }
]);
```

d) Sum of Available Copies by Author

To calculate the total number of available copies for each author, we can use `$group` by `author` and sum up `available_copies`.

javascript

```
db.Library.aggregate([
  {
    $group: {
      _id: "$author",
      total_available_copies: { $sum: "$available_copies" }
    }
  }
]);
```

e) Filter Books Published After 1950 with Average Rating > 4

Here, we use `$match` to filter books published after 1950, `$unwind` to flatten `ratings`, and `$group` to calculate the average rating. Then, `$match` is used again to filter books with an average rating greater than 4.

javascript

```
db.Library.aggregate([
  { $match: { publication_year: { $gt: 1950 } } },
  { $unwind: "$ratings" },
  {
    $group: {
      _id: "$title",
      avg_rating: { $avg: "$ratings.rating" },
      publication_year: { $first: "$publication_year" }
    }
  },
  { $match: { avg_rating: { $gt: 4 } } }
]);
```

2. Indexing Examples

Indexes in MongoDB are used to optimize query performance. Below are some examples of creating indexes and the benefits they provide.

a) Create an Index on **title** for Faster Searches

Creating an index on **title** will speed up queries that search for specific book titles.

```
javascript
```

```
db.Library.createIndex({ title: 1 });
```

b) Compound Index on **category** and **publication_year**

A compound index on **category** and **publication_year** will improve performance for queries that filter by both fields.

```
javascript
```

```
db.Library.createIndex({ category: 1, publication_year: -1 });
```

This compound index will be useful for queries like:

```
javascript
```

```
db.Library.find({ category: "Fiction", publication_year: { $gt: 2000 }  
});
```

c) Text Index on **author** and **title** for Full-Text Search

A text index allows us to perform full-text search on string fields. Here, we create a text index on both `author` and `title` fields.

```
javascript
```

```
db.Library.createIndex({ author: "text", title: "text" });
```

With this text index, we can run full-text search queries:

```
javascript
```

```
db.Library.find({ $text: { $search: "Orwell" } });
```

d) Unique Index on `book_id`

A unique index on `book_id` ensures that each book has a unique ID, preventing duplicate entries.

```
javascript
```

```
db.Library.createIndex({ book_id: 1 }, { unique: true });
```

Summary

Using **Aggregation**, we can perform complex data analysis directly within MongoDB. Indexes help optimize query performance, especially for large datasets, by reducing the time required to search, filter, and sort data.

You've hit the Free plan limit for GPT-4o.

Responses will use another model until your limit resets tomorrow after 2:30AM.

mongoDB - Map reduces operations: Implement Map reduces operation with suitable example using MongoDB.

ChatGPT can make mistakes. Check important info.