Name:- Sayali Dhadge

Emp_ID:- HS2632

Test 1

---

1) MongoDB:

Case Study: Real-Time Analytics Dashboard

Scenario:

You are working on a project to develop a real-time analytics dashboard for a web application. The dashboard will provide insights into user behavior, traffic patterns, and system performance. The application generates a large volume of events that need to be processed and stored efficiently for real-time analysis.

Requirements:

1. The system should be able to handle a high volume of incoming events in real-time.

2. Each event contains information such as timestamp, user ID, event type, and additional data specific to the event.

3. The events need to be stored in a database for further analysis and reporting.

4. The database should support fast querying and aggregation of event data.

5. Implement a feature to visualize the analytics data using charts and graphs on the dashboard.

6. The database should provide horizontal scalability to handle increasing event volumes.

7. Ensure data durability and availability in case of system failures or crashes.

Tasks:

1. Design the MongoDB database schema to store the event data efficiently for real-time analytics.

2. Implement MongoDB queries for the following operations:

a. Insert a new event into the database.

```
> db.events.insertOne({
    timestamp: new Date(),
    userId: "10002",
    eventType: "serial",
    title: "heraferi",
    location: "Kotor",
    cost: "$883442.70"
  });
< {
    acknowledged: true,
    insertedId: ObjectId("64b225dc475a12923b260da0")
  }
```

b. Retrieve events within a specific time range, filtered by event type.

```
db.events.find({
 timestamp: {
  $gte: new Date("2023-07-14T05:04:10.489Z"),
  $lte: new Date("2023-07-15T05:04:20.489Z")
 },
 eventType: "event6"
```

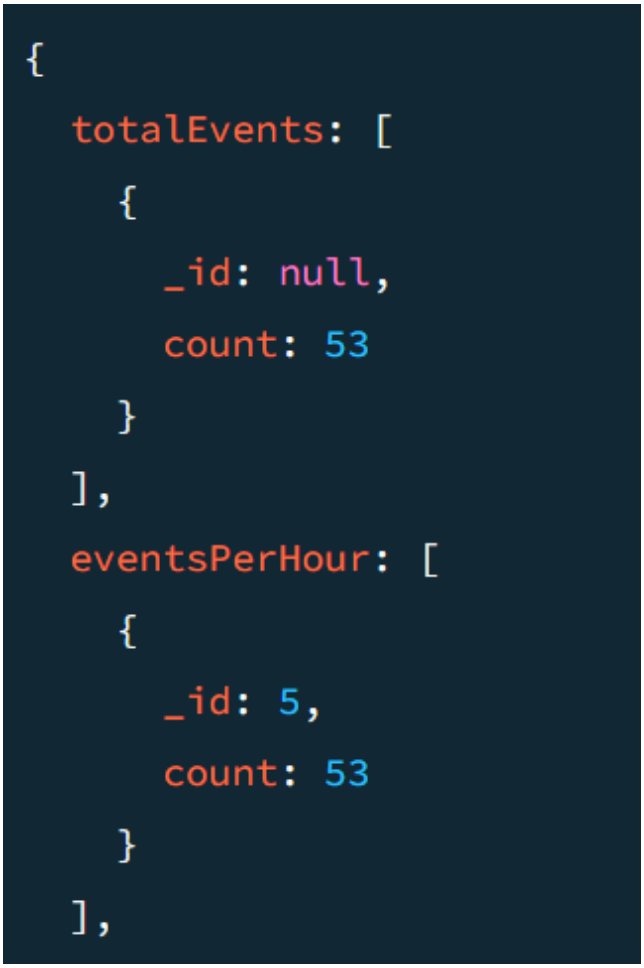
```
< {
    _id: ObjectId("64b228d4475a12923b260da9"),
    timestamp: 2023-07-15T05:04:20.489Z,
    userId: 'user6',
    eventType: 'event6',
    title: 'Title 6',
    location: 'Location 6',
    cost: '$343.13'
  }
  {
    _id: ObjectId("64b229e7475a12923b260dd6"),
    timestamp: 2023-07-15T05:04:20.489Z,
    userId: 'user6',
    eventType: 'event6',
    title: 'Title 6',
    location: 'Location 6',
    cost: '$343.13'
```

});

c. Perform aggregation queries to calculate metrics like total events, events per hour, and user activity.

```
db.events.aggregate([
 {
  $facet: {
   totalEvents: [
    {
     $group: {
      _id: null,
      count: { $sum: 1 }
     }
    }
   ],
   eventsPerHour: [
    {
     $group: {
      _id: { $hour: "$timestamp" },
      count: { $sum: 1 }
     }
    },
    {
     $sort: { _id: 1 }
    }
   ],
   userActivity: [
    {
     $group: {
      _id: "$userId",
      count: { $sum: 1 }
```

```
      }
    },
    {
      $sort: { count: -1 }
    }
  ]
 }
}
]);
```



d. Retrieve the top N users based on the number of events generated.

```
db.events.aggregate([
 {
  $group: {
   _id: "$userId",
   eventCount: { $sum: 1 }
  }
 },
 {
  $sort: { eventCount: -1 }
 },
 {
  $limit: 2
 }
]);
```

3. Discuss the benefits of using MongoDB's flexible schema for handling diverse event data.

MongoDB's flexible schema provides the ability to adapt to changing event data structures, handle different event types in a single collection, support agile development practices, avoid data duplication, improve performance, and simplify data integration.

4. Explain how MongoDB's replica sets and automatic failover can ensure high availability and data durability in the analytics system.

MongoDB's replica sets and automatic failover play a crucial role in ensuring high availability and data durability in an analytics system.

1. Replica Sets:

- Replica sets are a group of MongoDB instances that store the same data. They consist of a primary node and one or more secondary nodes.

- The primary node is responsible for handling all write operations and is the only node that can accept write requests from clients.

- Secondary nodes replicate data from the primary node and serve read requests, providing redundancy and scalability.

- Replica sets are designed to ensure data availability and prevent a single point of failure.

2. Automatic Failover:

- Automatic failover is the process of automatically promoting a secondary node to become the new primary node in the event of a primary node failure.

- When the primary node becomes unavailable, replica sets use an election process to select a new primary node from the available secondary nodes.

- Once a new primary is elected, the replica set continues to operate, accepting write operations and serving read operations.

- The failover process is automatic and transparent to the application, ensuring continuous availability and minimal downtime.

Q4) Implement a stack using linked lists.

A stack is a data structure that follows the LIFO (last in, first out) principle. This means that the last item added to the stack is the first item removed. A linked list is a data structure that consists of a series of nodes, each of which contains data and a pointer to the next node.

To implement a stack using linked lists, you would need to create a linked list node class that stores the data and a pointer to the next node. You would then need to create a stack class that contains a pointer to the top node of the stack. The stack class would also need to have methods for adding and removing items from the stack.

Ans:-

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}
class Stack {
  constructor() {
    this.top = null;
    this.size = 0;
  }


  // Add an item to the stack
  push(data) {
```

```
  const newNode = new Node(data);
  newNode.next = this.top;
  this.top = newNode;
  this.size++;
}

// Remove and return the top item from the stack
pop() {
  if (!this.top) {
    return null; // Stack is empty
  }
  const removedNode = this.top;
  this.top = this.top.next;
  this.size--;
  return removedNode.data;
}

// Return the top item from the stack without removing it
peek() {
  if (!this.top) {
    return null; // Stack is empty
  }
  return this.top.data;
}

// Check if the stack is empty
isEmpty() {
```

```
    return this.top === null;
  }

  // Return the number of items in the stack
  getSize() {
    return this.size;
  }

  // Clear the stack
  clear() {
    this.top = null;
    this.size = 0;
  }
}
const stack = new Stack();

// Pushing items to the stack
stack.push(10);
stack.push(20);
stack.push(30);

console.log("top element: " + stack.peek());

console.log("pop : " + stack.pop());
```

```
stack.push(40);


console.log("size of stack:  " + stack.getSize());


console.log("checking if stack is empty : " + stack.isEmpty());
console.log("popping elements:");
console.log(stack.pop());
console.log(stack.pop());
console.log(stack.pop());
console.log(stack.pop());


console.log("size of stack:  " + stack.getSize());


console.log("check id stack is empty: " + stack.isEmpty());
```

OUTPUT:-

```
[Running] node "d:\Test 1\Q4stack.js"
top element: 30
pop : 30
size of stack:  3
checking if stack is empty : false
popping elements:
40
20
10
null
size of stack:  0
check id stack is empty: true
```

Q5) Case Study 1: Phone Directory

Scenario:

You are tasked with designing a phone directory application that allows users to store and search for contact information. The application should support efficient insertion, deletion, and retrieval operations on a large number of contacts. Additionally, it should provide search functionality based on contact names and phone numbers.

Which data structure(s) would you choose to implement the phone directory application? Justify your choice(s) and discuss the time complexity of the key operations.

Ans:-

Given the requirement for efficient operations on a large number of contacts, a hash table is generally more suitable for this phone directory application due to its constant-time average case performance for insertion, deletion, and search by both name and phone number.

Time Complexity:

- Insertion: O(1) on average (can be O(n) in worst case with collisions)
- Deletion: O(1) on average (can be O(n) in worst case with collisions)
- Search by Name: O(1) on average (can be O(n) in worst case with collisions)
- Search by Phone Number: O(1) on average (can be O(n) in worst case with collisions)

Implement the data structure(s) you selected in question 1 and provide functions for inserting a contact, deleting a contact, and searching for a contact by name or phone number.

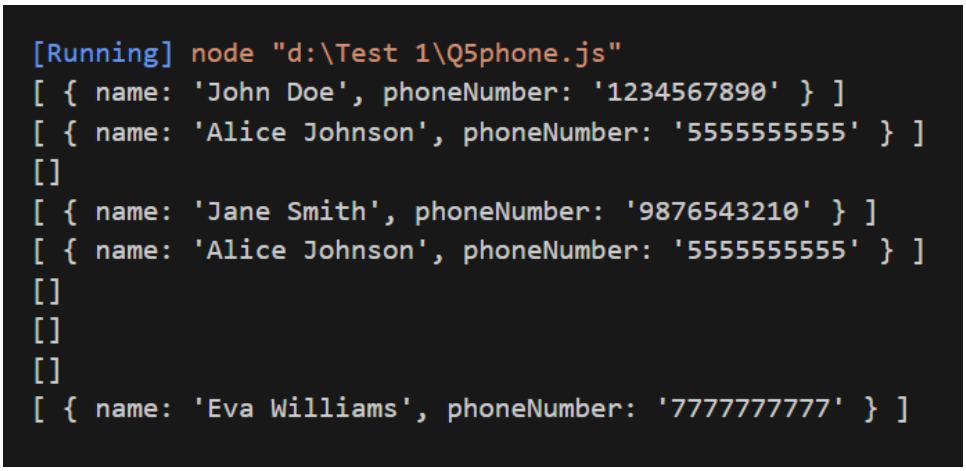
Code:-

```
class PhoneDirectory {
  constructor() {
    this.contactsByNumber = {};  // Hash table to store contacts by phone number
    this.contactsByName = {};    // Hash table to store contacts by name
  }

  // Insert a contact into the phone directory
  insertContact(name, phoneNumber) {
    const contact = { name, phoneNumber };
    this.contactsByNumber[phoneNumber] = contact;
```

```
    this.contactsByName[name] = contact;
  }

  // Delete a contact from the phone directory
  deleteContact(phoneNumber) {
    const contact = this.contactsByNumber[phoneNumber];
    if (contact) {
      delete this.contactsByNumber[phoneNumber];
      delete this.contactsByName[contact.name];
    }
  }

  // Search for contacts by name
  searchByName(name) {
    const contact = this.contactsByName[name];
    return contact ? [contact] : [];
  }

  // Search for contacts by phone number
  searchByNumber(phoneNumber) {
    const contact = this.contactsByNumber[phoneNumber];
    return contact ? [contact] : [];
  }
}
const phoneDirectory = new PhoneDirectory();


// Inserting contacts
phoneDirectory.insertContact('John Doe', '1234567890');
```

```
phoneDirectory.insertContact('Jane Smith', '9876543210');
phoneDirectory.insertContact('Alice Johnson', '5555555555');
phoneDirectory.insertContact('Bob Anderson', '9999999999');

// Searching contacts by name
console.log(phoneDirectory.searchByName('John Doe'));
console.log(phoneDirectory.searchByName('Alice Johnson'));
console.log(phoneDirectory.searchByName('David Smith'));

// Searching contacts by phone number
console.log(phoneDirectory.searchByNumber('9876543210'));
console.log(phoneDirectory.searchByNumber('5555555555'));
console.log(phoneDirectory.searchByNumber('1111111111'));

// Deleting a contact
phoneDirectory.deleteContact('9876543210');
console.log(phoneDirectory.searchByNumber('9876543210'));

// Searching deleted contact
console.log(phoneDirectory.searchByName('Jane Smith'));

// Inserting a new contact
phoneDirectory.insertContact('Eva Williams', '7777777777');

// Searching the newly inserted contact
console.log(phoneDirectory.searchByNumber('7777777777'));
```

OUTPUT:-



Analyze the time complexity of each operation in your implementation and suggest possible ways to optimize the data structure(s) or algorithms used to improve performance.

Time Complexity:

- Insertion: O(1) on average (can be O(n) in worst case with collisions)
- Deletion: O(1) on average (can be O(n) in worst case with collisions)
- Search by Name: O(1) on average (can be O(n) in worst case with collisions)
- Search by Phone Number: O(1) on average (can be O(n) in worst case with collisions)

Possible ways to optimize data structure:

- Hash Function Optimization:
- Handling Collisions:
- Load Factor Monitoring and Resizing

MYSQL

1.SELECT CONCAT('hotel_info', Hotel_name) AS hotel_info
FROM hotel_details
ORDER BY hotel_info DESC;2.SELECT car_id, car_name, owner_id
FROM cars
WHERE car_type IN ('Hatchback', 'SUV')
ORDER BY car_id;