Section 1 :----java Coding :--

=================================================================================

**1:- Sort a list of Employees by name and then by salary using:Java Streams.**

// Sort by name first, then by salary

List<Employee> sortedEmployees = employees.stream()

.sorted(Comparator.comparing(Employee::getName)

.thenComparing(Employee::getSalary))

.collect(Collectors.toList());

// Print the sorted list

sortedEmployees.forEach(System.out::println);

=======================================================================================

**2:- Group Employees by departments using:Java Streams.**

Map<String, List<Employee>> collect = emp.stream().collect(Collectors.groupingBy(Employee::getDepartment));

collect.forEach((e, d) -> System.out.println(e + " " + d));

sql :-

SELECT department,

AVG(salary) AS average\_salary

FROM employees

GROUP BY department;

========================================================================================

**3:- Write a Java program for:Returning the top 2 students in each subject (marks are subject-wise).**

import java.util.\*;

import java.util.stream.Collectors;

class Student {

private String name;

private int marks;

// Constructor

public Student(String name, int marks) {

this.name = name;

this.marks = marks;

}

// Getters

public String getName() {

return name;

}

public int getMarks() {

return marks;

}

@Override

public String toString() {

return name + " (" + marks + " marks)";

}

}

public class TopStudentsInSubjects {

public static void main(String[] args) {

// Create a map where the key is the subject and value is a list of students and their marks

Map<String, List<Student>> subjectToStudentsMap = new HashMap<>();

// Adding students and their marks in each subject

subjectToStudentsMap.put("Math", Arrays.asList(

new Student("Alice", 95),

new Student("Bob", 85),

new Student("Charlie", 90),

new Student("David", 80)

));

subjectToStudentsMap.put("Science", Arrays.asList(

new Student("Alice", 88),

new Student("Bob", 92),

new Student("Charlie", 85),

new Student("David", 94)

));

subjectToStudentsMap.put("English", Arrays.asList(

new Student("Alice", 91),

new Student("Bob", 78),

new Student("Charlie", 87),

new Student("David", 93)

));

// Process each subject to find top 2 students

subjectToStudentsMap.forEach((subject, students) -> {

List<Student> topStudents = students.stream()

.sorted((s1, s2) -> Integer.compare(s2.getMarks(), s1.getMarks())) // Sort by marks in descending order

.limit(2) // Get top 2 students

.collect(Collectors.toList());

// Print top 2 students for the subject

System.out.println("Top 2 students in " + subject + ":");

topStudents.forEach(System.out::println);

System.out.println();

});

}

}

===============================================================================================================

**4:- Finding the second largest number in a list efficiently.**

public static void main(String[] args) {

List<Integer> list = Arrays.asList(1, 2, 3, 9, 8);

int largest = Integer.MIN\_VALUE;

int secondLagest = Integer.MIN\_VALUE;

for (int num : list) {

if (num > largest) {

secondLagest = largest;

largest = num;

} else if (num > secondLagest && num < largest) {

secondLagest = num;

}

}

System.out.println(secondLagest);

}

Time Complexity:

O(n): We only need one iteration through the list, so it’s linear in terms of the number of elements in the list.

Space Complexity:

O(1): Only a fixed amount of extra space is used (largest and secondLargest variables), so it’s constant space.

using java 8 :-

List<Integer> list = Arrays.asList(1, 2, 3, 9, 8);

Integer secondMax = list.stream().sorted((n1, n2) -> n2.compareTo(n1)).skip(1).findFirst().get();

System.out.println(secondMax);

=================================================================================================================

**5:- Filtering unique elements from a list using Java 8.**

import java.util.\*;

import java.util.stream.\*;

public class UniqueElements {

public static void main(String[] args) {

// Create a list with some duplicate elements

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 3, 5, 6, 2, 7, 5);

// Using Java 8 Streams to get unique elements

List<Integer> uniqueNumbers = numbers.stream()

.distinct() // Filter out duplicates

.collect(Collectors.toList()); // Collect the results in a List

// Print the unique elements

System.out.println("Unique elements: " + uniqueNumbers);

}

}

=========================================================================================

6:----Finding the first non-repetitive character in a string.

import java.util.\*;

import java.util.stream.\*;

public class FirstNonRepeatingCharacter {

public static void main(String[] args) {

String input = "geeksforgeeks";

// Create a frequency map using streams

LinkedHashMap<Character, Long> freMap = input.chars()

.mapToObj(ch -> (char) ch)

.collect(Collectors.groupingBy(

x -> x,

LinkedHashMap::new,

Collectors.counting()));

// Find the first non-repeating character

Optional<Map.Entry<Character, Long>> entry = freMap.entrySet().stream()

.filter(entry1 -> entry1.getValue() == 1)

.findFirst();

// Output the result

if (entry.isPresent()) {

System.out.println("The first non-repeating character is: " + entry.get().getKey());

} else {

System.out.println("No non-repeating character found.");

}

}

}

=======================================================================-------------------------------------------------------------------------

**8:- Generating random numbers, sorting them, and finding their index.**

import java.util.\*;

public class RandomNumbersSorting {

public static void main(String[] args) {

// Step 1: Generate random numbers

int size = 10; // Size of the array

Random rand = new Random();

int[] numbers = new int[size];

// Populate the array with random numbers

for (int i = 0; i < size; i++) {

numbers[i] = rand.nextInt(100); // Random numbers between 0 and 99

}

// Print the original random numbers

System.out.println("Original Array: " + Arrays.toString(numbers));

// Step 2: Manually sort the numbers (using bubble sort)

bubbleSort(numbers);

// Print the sorted array

System.out.println("Sorted Array: " + Arrays.toString(numbers));

// Step 3: Find the index of a specific number (e.g., number 50)

int target = 50;

int index = findIndex(numbers, target);

// Step 4: Print the result

if (index != -1) {

System.out.println("The index of " + target + " is: " + index);

} else {

System.out.println(target + " is not found in the sorted array.");

}

}

// Bubble Sort implementation

public static void bubbleSort(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - 1 - i; j++) {

if (arr[j] > arr[j + 1]) {

// Swap elements if they are in the wrong order

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

// Find the index of the target number

public static int findIndex(int[] arr, int target) {

for (int i = 0; i < arr.length; i++) {

if (arr[i] == target) {

return i; // Return the index if the target is found

}

}

return -1; // Return -1 if the target is not found

}

}

// output

Original Array: [75, 23, 50, 10, 89, 61, 98, 2, 35, 65]

Sorted Array: [2, 10, 23, 35, 50, 61, 65, 75, 89, 98]

The index of 50 is: 4

Time Complexity:

Bubble Sort: O(n^2), where n is the number of elements in the array. This is because Bubble Sort compares each pair of adjacent elements.

Finding Index: O(n), because we are doing a linear search through the array.

Space Complexity:

O(n): We use a temporary array to store the sorted numbers, so the space complexity is proportional to the number of elements in the array.

Summary:

This approach provides a more "manual" way of solving the problem using basic sorting and search algorithms.

While Bubble Sort is not the most efficient sorting algorithm for large datasets (since it has O(n²) time complexity),

it is simple and works well for small arrays.

You can replace it with more efficient algorithms like QuickSort or MergeSort if performance becomes a concern for large datasets.

=================================================================================

**9:-- Converting a list of objects (e.g., Person with name and age) to a map.**

// Convert the list of Person objects to a map where the key is the name and the value is the age

Map<String, Integer> collect = emp.stream().collect(Collectors.toMap(Employee::getName, Employee::getAge));

collect.forEach((name, age) -> {

System.out.println(name + " " + age);

});

================================================================================================

**10:- Implementing multithreading with multiple lists.**

import java.util.\*;

import java.util.concurrent.\*;

class SumTask implements Callable<Integer> {

private List<Integer> list;

public SumTask(List<Integer> list) {

this.list = list;

}

@Override

public Integer call() throws Exception {

int sum = 0;

for (int num : list) {

sum += num;

}

return sum;

}

}

public class MultithreadingWithMultipleLists {

public static void main(String[] args) throws InterruptedException, ExecutionException {

// Create multiple lists

List<Integer> list1 = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> list2 = Arrays.asList(6, 7, 8, 9, 10);

List<Integer> list3 = Arrays.asList(11, 12, 13, 14, 15);

// Create an ExecutorService to manage threads

ExecutorService executorService = Executors.newFixedThreadPool(3);

// Submit tasks for each list

Future<Integer> future1 = executorService.submit(new SumTask(list1));

Future<Integer> future2 = executorService.submit(new SumTask(list2));

Future<Integer> future3 = executorService.submit(new SumTask(list3));

// Get the results from each task

int sum1 = future1.get(); // Waits for completion and gets the result

int sum2 = future2.get(); // Waits for completion and gets the result

int sum3 = future3.get(); // Waits for completion and gets the result

// Print the results

System.out.println("Sum of list 1: " + sum1);

System.out.println("Sum of list 2: " + sum2);

System.out.println("Sum of list 3: " + sum3);

// Shutdown the executor service

executorService.shutdown();

}

}

===================================================================================================

=============================================================================================

**12:-- Write hashCode and equals methods for a custom Employee class.**

import java.util.Objects;

public class Employee {

private int id;

private String name;

private double salary;

// Constructor

public Employee(int id, String name, double salary) {

this.id = id;

this.name = name;

this.salary = salary;

}

// Getter and Setter methods

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

// Overriding equals method

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Employee employee = (Employee) o;

return id == employee.id &&

Double.compare(employee.salary, salary) == 0 &&

Objects.equals(name, employee.name);

}

// Overriding hashCode method

@Override

public int hashCode() {

return Objects.hash(id, name, salary);

}

@Override

public String toString() {

return "Employee{id=" + id + ", name='" + name + "', salary=" + salary + '}';

}

public static void main(String[] args) {

Employee employee1 = new Employee(1, "Alice", 50000);

Employee employee2 = new Employee(1, "Alice", 50000);

Employee employee3 = new Employee(2, "Bob", 60000);

System.out.println("Employee 1 hashCode: " + employee1.hashCode());

System.out.println("Employee 2 hashCode: " + employee2.hashCode());

System.out.println("Employee 3 hashCode: " + employee3.hashCode());

System.out.println("Employee 1 equals Employee 2: " + employee1.equals(employee2));

System.out.println("Employee 1 equals Employee 3: " + employee1.equals(employee3));

}

}

Breakdown of hashCode() in this Example:

Objects.hash(id, name, salary):

The hashCode() method combines the hash codes of the id, name, and salary fields.

id is an int, so its hash code is just its value.

name is a String, so name.hashCode() is computed using the string's hashCode() method.

salary is a double, so Double.hashCode(salary) is used to compute its hash code.

Combining the Hash Codes:

The Objects.hash() method combines the hash codes of these fields and returns a single hash code for the Employee object.

Result:

Each Employee object will have a unique hash code based on its id, name, and salary fields.

Employee 1 hashCode: 138103763

Employee 2 hashCode: 138103763

Employee 3 hashCode: 1716547229

Employee 1 equals Employee 2: true

Employee 1 equals Employee 3: false

=======================================================================================

**13:---Explanation and behavior of:String s = "abc";**

String p = "abc";

String r = new String("abc");

s == p, s == r, s.equals(p), s.equals(r.)

Summary of Comparison Results:

Expression Result

s == p true

s == r false

s.equals(p) true

s.equals(r) true

========================================================================================== **Section 2 : sql queries**

**1:---- Queries:Get the 3rd highest salary.**

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 2;

LIMIT 1 OFFSET 2: Limits the result to one row and skips the first two rows, effectively retrieving the third highest salary.

=======================================================================

**2-------------Count employees per department and location.**

SELECT department, location, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department, location;

=====================================================================

**3:-----Get the top 5 highest-paid employees per department.**

WITH RankedEmployees AS (

SELECT employee\_id, department, salary,

ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rank

FROM employees

)

SELECT employee\_id, department, salary

FROM RankedEmployees

WHERE rank <= 5

ORDER BY department, rank;

Explanation:

The ROW\_NUMBER() function assigns a rank based on salary within each department (PARTITION BY department).

The WHERE rank <= 5 limits the results to the top 5 highest-paid employees in each department.

The ORDER BY department, rank ensures the results are sorted by department and rank.

=========================================================================

4:- -----------Use joins to fetch employee and department names where salary is the highest in a department.

SELECT e.employee\_name, d.department\_name

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id

WHERE e.salary = (

SELECT MAX(salary)

FROM employees

WHERE department\_id = e.department\_id

);

===================================================================================

**5:----------- Group employees by age or other attributes.**

SELECT age, COUNT(\*) AS number\_of\_employees

FROM employees

GROUP BY age

ORDER BY age;

====================================================================

**6:----------------Write a query to pull cities with more than 10,000 employees.**

SELECT city, COUNT(\*) AS employee\_count

FROM employees

GROUP BY city

HAVING COUNT(\*) > 10000

ORDER BY employee\_count DESC;

=================================================================

7-----------Difference between:Inner and Outer joins.

INNER JOIN: Only returns rows where there is a match in both tables.

OUTER JOIN: Returns all rows from one table and the matching rows from the other table, with NULL values for non-matching rows.

===========================================================

**8:------------ Function and Stored Procedure.**

Function:

Purpose: Primarily used to compute and return a single value (like a calculation or transformation).

Return Type: Always returns a value (scalar or table).

Usage: Can be used in queries (e.g., SELECT, WHERE, ORDER BY).

Side Effects: Cannot modify database state (e.g., cannot use INSERT, UPDATE, DELETE).

Invocation: Can be called directly in SQL statements.

Example:

sql

Copy

CREATE FUNCTION GetEmployeeSalary (employee\_id INT)

RETURNS DECIMAL

AS

BEGIN

RETURN (SELECT salary FROM employees WHERE id = employee\_id);

END;

Stored Procedure:

Purpose: Used to perform a series of operations, like executing multiple queries or updating data.

Return Type: Does not necessarily return a value, but can return multiple result sets or status codes (via OUT parameters).

Usage: Typically invoked using EXEC or CALL in SQL, not within a query.

Side Effects: Can modify the database (e.g., INSERT, UPDATE, DELETE).

Invocation: Called as a separate statement (e.g., EXEC or CALL).

Example:

sql

Copy

CREATE PROCEDURE UpdateEmployeeSalary (employee\_id INT, new\_salary DECIMAL)

AS

BEGIN

UPDATE employees

SET salary = new\_salary

WHERE id = employee\_id;

END;

Key Differences:

Aspect Function Stored Procedure

Return Type Always returns a value Can return multiple result sets or status

Use in Queries Can be used in SELECT, WHERE, etc. Cannot be used directly in queries

Side Effects Cannot modify data (read-only) Can modify data (e.g., INSERT, UPDATE)

Invocation Used in expressions and queries Invoked using EXEC or CALL

In summary:

Function is for calculations or transformations with no side effects.

Stored Procedure is for executing complex logic, including data modifications, with possible side effects.

============================================================================================

**9 :-----Importance and criteria for Indexing and Partitioning.**

Indexing in SQL

Importance of Indexing:

Performance Boost: Indexes speed up data retrieval operations, making queries faster (especially for large tables).

Efficient Searching: With an index, the database engine can avoid scanning the entire table, allowing faster searches for specific values.

Improved Sorting and Filtering: Queries that involve ORDER BY, GROUP BY, and WHERE clauses benefit significantly from indexes.

Reduced I/O: Reduces the amount of data read from disk, especially in large datasets.

Criteria for Indexing:

Frequently Queried Columns: Index columns that are often used in WHERE, JOIN, ORDER BY, or GROUP BY clauses.

Unique or Primary Key Columns: These are ideal candidates for indexing to ensure data uniqueness and fast retrieval.

Large Tables: Tables with a large number of rows benefit from indexing as it helps in fast searching.

Foreign Keys: Columns frequently used as foreign keys should be indexed to speed up joins.

Example of Creating an Index:

sql

Copy

-- Create an index on the 'email' column of the 'employees' table

CREATE INDEX idx\_email ON employees(email);

This index will speed up any search queries that involve the email column, such as:

sql

Copy

SELECT \* FROM employees WHERE email = 'example@example.com';

Partitioning in SQL

Importance of Partitioning:

Improved Query Performance: Partitioning splits a large table into smaller, more manageable pieces, improving query performance

by scanning only relevant partitions.

Faster Data Management: You can manage partitions independently (e.g., backup, load, or delete specific partitions).

Efficient Data Storage: Helps in distributing data across different physical storage locations for better performance and scalability.

Parallel Processing: Queries can be processed in parallel across partitions, improving performance in large-scale databases.

Criteria for Partitioning:

Large Tables: When a table grows significantly (millions of rows), partitioning can help in managing and querying data efficiently.

Data Distribution: Partitioning works well when the data can be logically divided (e.g., by date, region, etc.).

Frequent Range Queries: Tables queried by ranges (like date ranges) benefit from partitioning, as only relevant partitions are scanned.

Example of Partitioning by Range (e.g., Date):

sql

Copy

-- Partition the 'sales' table by year (assuming 'sale\_date' is a date column)

CREATE TABLE sales (

sale\_id INT,

sale\_date DATE,

amount DECIMAL

)

PARTITION BY RANGE (YEAR(sale\_date)) (

PARTITION p\_2020 VALUES LESS THAN (2021),

PARTITION p\_2021 VALUES LESS THAN (2022),

PARTITION p\_2022 VALUES LESS THAN (2023)

);

Here, the sales table is partitioned by the year of the sale\_date. Each partition will contain the sales for a specific year,

improving the performance of queries that filter by year.

Key Differences Between Indexing and Partitioning:

Aspect Indexing Partitioning

Purpose Improves search, sorting, and filtering speeds Divides large tables into smaller, manageable pieces

How It Works Creates a data structure (index) for fast lookup Divides a table’s data based on a criterion (e.g., range)

Use Case Frequently queried columns (e.g., for searching) Large tables with logical data ranges (e.g., dates)

Impact on Query Reduces data scanning for retrieval Reduces scanning by focusing on relevant partitions

Storage Consideration Increases storage requirements for the index Can improve storage management and scalability

Summary:

Indexing: Helps in quick data retrieval by creating a fast lookup mechanism, especially for frequently queried columns.

Partitioning: Helps in managing and optimizing large datasets by dividing a table into smaller parts,

improving performance for specific types of queries (like date-based ranges).

=========================================================

**10 :----Rules for using the GROUP BY clause.**

here are the simple rules for using the GROUP BY clause:

Use with Aggregate Functions: It’s used with aggregate functions like COUNT(), SUM(), AVG(), etc.

Columns in SELECT: All non-aggregated columns in the SELECT clause must be included in the GROUP BY clause.

Order of Clauses: GROUP BY comes after WHERE but before HAVING and ORDER BY.

Filtering Groups: Use HAVING to filter groups after the aggregation, and WHERE to filter individual rows before grouping.

==========================================================================

**11:- SQL Triggers.**

A SQL trigger is a database object that automatically executes a specified set of SQL statements in response

to certain events on a table or view (such as INSERT, UPDATE, or DELETE). It is used to enforce business rules,

audit data changes, or maintain data integrity.

Key Points:

Automatic Execution: Triggers fire automatically based on specified events.

Types: Common types include BEFORE, AFTER, and INSTEAD OF triggers.

Events: Triggered by data modifications like INSERT, UPDATE, or DELETE.

Usage: Enforce constraints, audit logging, or update related tables.

Example:

sql

Copy

CREATE TRIGGER update\_salary

AFTER UPDATE ON employees

FOR EACH ROW

BEGIN

INSERT INTO salary\_audit (employee\_id, old\_salary, new\_salary)

VALUES (OLD.employee\_id, OLD.salary, NEW.salary);

END;

This trigger records salary changes in an audit table after an UPDATE on the employees table.

==================================================================================================================

**Section 3 :------------kafka**

**1- What is the default Kafka retention period?**

The default Kafka retention period for messages is 7 days (168 hours). This means that by default,

Kafka will retain messages in a topic for 7 days before they are eligible for deletion, unless the retention period is configured differently.

The retention period can be configured per topic using the log.retention.hours parameter or its equivalent

in newer versions, like log.retention.ms, where you can specify the retention in milliseconds.

==========================================================================================

**2--- How do you read Kafka messages again after they’ve been read once?**

To read Kafka messages again after they've been read once, you can:

Reset the Consumer Offset: Use the kafka-consumer-groups command or programmatically reset the offset to an earlier point,

such as the beginning, using --to-earliest.

Create a New Consumer Group: A new group will start reading from the earliest or latest message, depending on your configuration.

Adjust Retention Settings: Ensure messages are still available by extending the retention period if necessary.

// Reset offset to the beginning (first message) for all partitions

consumer.seekToBeginning(partitions);

// Alternatively, you can reset to the end (latest message)

// consumer.seekToEnd(partitions);

============================================================================================================

**3---------------What configurations are needed to use Kafka?**

To use Kafka, several key configurations are required, depending on your setup and use case. '

Here’s a list of essential configurations for both the Kafka broker and Kafka consumer/producer:

a)

Kafka Broker Configurations:

listeners: Defines the address and port where the Kafka broker will listen for incoming connections.

Example: listeners=PLAINTEXT://localhost:9092

log.dirs: Specifies the directory where Kafka stores its log files (data for topics).

Example: log.dirs=/var/lib/kafka/data

zookeeper.connect: Kafka relies on ZooKeeper for coordination (in older versions). Specifies ZooKeeper’s address.

Example: zookeeper.connect=localhost:2181

log.retention.hours or log.retention.ms: Defines the retention period for Kafka topics (messages in topics are deleted after this time).

Example: log.retention.ms=86400000 (1 day)

num.partitions: Default number of partitions per topic.

Example: num.partitions=3

broker.id: Unique identifier for each broker in the Kafka cluster.

Example: broker.id=1

b)

Kafka Producer Configurations:

bootstrap.servers: A comma-separated list of Kafka broker addresses.

Example: bootstrap.servers=localhost:9092

key.serializer: The serializer for the key of the message.

Example: key.serializer=org.apache.kafka.common.serialization.StringSerializer

value.serializer: The serializer for the value of the message.

Example: value.serializer=org.apache.kafka.common.serialization.StringSerializer

acks: The acknowledgment level the producer waits for (0, 1, or all).

Example: acks=all (waits for all replicas to acknowledge)

c)

Kafka Consumer Configurations:

bootstrap.servers: A comma-separated list of Kafka broker addresses.

Example: bootstrap.servers=localhost:9092

group.id: The unique consumer group ID used for offset tracking.

Example: group.id=my-consumer-group

key.deserializer: The deserializer for the key of the message.

Example: key.deserializer=org.apache.kafka.common.serialization.StringDeserializer

value.deserializer: The deserializer for the value of the message.

Example: value.deserializer=org.apache.kafka.common.serialization.StringDeserializer

auto.offset.reset: What to do when there is no initial offset or the offset is out of range (earliest, latest).

Example: auto.offset.reset=earliest

============================================================================

**4:---------------Kafka architecture.**

Kafka's architecture is designed around a distributed and scalable message broker system.

Below is a simplified overview of Kafka architecture components, followed by a diagram to visualize how they work together:

Kafka Architecture Components:

Producer:

The producer sends messages (events) to Kafka topics.

Producers push data to Kafka brokers.

Kafka Brokers:

Kafka brokers are the core component where data is stored. A Kafka cluster consists of multiple brokers.

Brokers store partitions of topics and handle incoming data.

Topic:

A topic is a logical channel to which producers send messages.

Kafka topics are divided into partitions for scalability and parallelism.

Partition:

Each topic is split into partitions, which allow Kafka to scale and distribute data across brokers.

Each partition is an ordered, immutable sequence of messages, and it allows for parallel data processing.

Consumer:

Consumers subscribe to Kafka topics and read messages.

They maintain offsets to track which message they have consumed.

Consumer Group:

A consumer group is a group of consumers that share the load of reading data from Kafka topics.

Each message in a topic partition is consumed by only one consumer within the group, enabling parallel consumption.

Zookeeper (in older Kafka versions, but now being replaced by KRaft):

Zookeeper manages metadata and broker coordination, leader election, and overall Kafka cluster management.

It helps maintain consistency across distributed brokers.

Kafka Architecture Diagram:

+----------------------------------------------------+

| Kafka Cluster |

| +------------+ +------------+ +------------+ |

| | Broker 1 | | Broker 2 | | Broker 3 | |

| | |<---->| |<-->| | |

| | Partitions | | Partitions | | Partitions | |

| +------------+ +------------+ +------------+ |

| ^ ^ ^ |

| | | | |

+--------+------------------+-------------------+-------+

| | |

+--------------------+ +--------------------+ +--------------------+

| Producer | | Producer | | Producer |

+--------------------+ +--------------------+ +--------------------+

| | |

+------------------>+-------------------+-------------------+

| |

+-----------------------------+

| Consumer Group |

| +---------------------+ |

| | Consumer 1 | |

| | Consumer 2 | |

| | Consumer N | |

+-----------------------------+

Key Points:

Scalability: Kafka scales horizontally by adding more brokers and partitions.

Fault Tolerance: Kafka replicates partitions across brokers to ensure data is available even if a broker fails.

Parallelism: Kafka enables parallel data consumption by distributing topic partitions across consumer groups.

This architecture is designed for high throughput, low latency, and fault tolerance, making Kafka a popular choice for

real-time data streaming applications.

==========================================================

**5--------- why kafka perfer over sqs**

High Throughput: Kafka supports very high message throughput, making it suitable for real-time data streaming

with millions of messages per second, while SQS has a lower throughput compared to Kafka.

Durability & Persistence: Kafka persists messages on disk and allows configurable retention times (days, weeks),

enabling consumers to read messages at any time. SQS does not offer long-term storage and deletes messages once

consumed unless explicitly retained.

Message Ordering: Kafka guarantees message ordering within partitions,

which is crucial for many real-time applications. SQS, on the other hand, doesn’t guarantee order unless using FIFO queues.

========================================================================================================================================

**Section 4: Spring framework**

**1:----Spring annotations and their uses.**

Here are some important Spring annotations and their uses:

@Component: Marks a class as a Spring-managed bean. It’s a general-purpose annotation that can be used for any bean.

Example:

@Component is used for simple beans, like services or repositories.

@Autowired: Automatically injects dependencies into a class, constructor, or method.

Example: @Autowired private MyService myService;

@Service: A specialization of @Component used to mark service classes in the service layer.

Example: Used to indicate business logic in your application.

@Repository: A specialization of @Component used to define a Data Access Object (DAO) class, often for database interactions.

Example: @Repository helps with exception translation in persistence layers.

@Controller: Marks a class as a Spring MVC controller that handles HTTP requests.

Example: Used in Spring MVC to handle web requests.

@RestController: A specialization of @Controller that combines @ResponseBody,

meaning the return values of methods will be directly written to the HTTP response body.

Example: Used in RESTful web services to return JSON/XML responses.

@RequestMapping: Specifies a URL pattern to map web requests to specific methods in controllers.

Example: @RequestMapping("/home") maps the /home URL to the method.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping: Specialized forms of @RequestMapping to handle

HTTP GET, POST, PUT, and DELETE requests, respectively.

Example: @GetMapping("/users") maps to a GET request for the users resource.

@Value: Injects values into fields, typically from configuration files (e.g., application.properties).

Example: @Value("${app.name}") private String appName;

@Configuration: Marks a class as a configuration class, indicating it contains bean definitions using @Bean methods.

Example: Used to define beans in Java-based configuration.

@Bean: Defines a bean within a @Configuration class.

Example: @Bean public MyService myService() { return new MyService(); }

@EnableAutoConfiguration: Tells Spring Boot to automatically configure the application based on the dependencies present in the classpath.

Example: Automatically configures the application’s components like data sources.

@Qualifier: Used with @Autowired to specify which bean to inject when there are multiple candidates.

Example: @Autowired @Qualifier("myService") private MyService service;

@Scope: Specifies the scope of a Spring bean (e.g., singleton, prototype).

Example: @Scope("prototype") makes the bean a new instance every time it is requested.

@Transactional: Marks a method or class for transaction management.

Example: Used for managing database transactions in service layer methods.

These annotations enable key functionalities in the Spring Framework, such as dependency injection, transaction management,

web handling, and more, providing a robust foundation for building

Java applications.

=======================================================================================================

**2:- What is the Dispatcher Servlet in Spring?**

The DispatcherServlet in Spring is the central component of the Spring MVC framework,

responsible for handling HTTP requests and routing them to appropriate handlers (controllers).

Key Roles of the DispatcherServlet:

Request Handling: It intercepts incoming HTTP requests and forwards them to the appropriate controller based on the request mapping.

Controller Invocation: After receiving a request, it delegates the request to the relevant controller method (using handler mappings).

Model and View Resolution: It retrieves the ModelAndView object from the controller, which contains the model data and the logical view name,

and then resolves it to the actual view (JSP, Thymeleaf, etc.).

View Rendering: It uses a view resolver to render the view (such as HTML or JSON) and send the response back to the client.

How it Works:

DispatcherServlet listens to all incoming HTTP requests and maps them to appropriate controller methods via @RequestMapping

or other specialized annotations (like @GetMapping).

The controller processes the request and returns a ModelAndView or a response body.

The View Resolver resolves the view and renders it as the final HTTP response.

Lifecycle:

Initialization: When the application starts, DispatcherServlet is initialized and configured in the web.xml

(or via Java-based configuration in Spring Boot).

Request Handling: For each incoming request, DispatcherServlet dispatches it to the appropriate controller.

Response: It sends the response back to the client after processing.

==============================================================================================================

**3------Starting a Spring Boot application:Use of @Configuration, component scan, property file precedence, Spring profiles**

a)@Configuration

@Configuration:

Purpose: Marks a class as a configuration class that contains bean definitions.

In Spring Boot, you generally don’t need to manually use @Configuration because Spring Boot automatically configures beans

through auto-configuration.

However, you can still use it for custom configurations.

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyServiceImpl();

}

}

This is typically used for Java-based configuration.

b: component scan

Component Scanning:

Purpose: Spring Boot automatically scans for annotated components (like @Component, @Service, @Repository, @Controller, etc.)

in the base package and its sub-packages.

This is done using @SpringBootApplication, which is a combination of:

@Configuration

@EnableAutoConfiguration

@ComponentScan

Default Behavior: Spring Boot scans the package where the main application class (annotated with @SpringBootApplication) is located.

Example:

@SpringBootApplication

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

This will trigger component scanning starting from com.example (the package of the main class) and scan all sub-packages.

Custom Component Scan: You can customize the base package to scan:

@ComponentScan(basePackages = "com.example.custom")

public class AppConfig {

// Custom configurations

}

c: . Property File Precedence:

Spring Boot supports multiple ways to externalize configuration properties, and these properties can come from different sources.

The order of precedence determines which properties are used.

Precedence Order:

Command-line arguments: Properties passed via the command line (e.g., --server.port=8081).

application.properties or application.yml: Files located in src/main/resources.

Profile-specific files: application-{profile}.properties (e.g., application-dev.properties).

Environment variables: Properties set in the environment, with the option to prefix them (e.g., SPRING\_DATASOURCE\_URL).

JVM System properties: Properties passed to the JVM (e.g., -Dserver.port=8081).

Default properties: Built-in defaults provided by Spring Boot.

d: Spring Profiles:

Purpose: Spring Profiles are used to specify different configurations for different environments (e.g., development, production, test).

You can activate profiles via properties or annotations.

using profiles in code

@Configuration

@Profile("dev")

public class DevConfig {

// Development-specific beans

}

@Configuration

@Profile("prod")

public class ProdConfig {

// Production-specific beans

}

Activating Profiles:

Via application.properties

spring.profiles.active=dev

Via command-line argument

java -jar myapp.jar --spring.profiles.active=prod

Summary:

@Configuration: Marks a class as a source of bean definitions.

Component Scanning: Automatically detects Spring components in the base package and its sub-packages.

Property File Precedence: Spring Boot loads properties in a defined order,

giving higher precedence to command-line arguments, environment variables, etc.

Spring Profiles: Allows different configurations for different environments (development, production, etc.),

activated via property files or programmatically.

=======================================================================================

**4:-------------Adding Spring Security to a project.**

Basic Authentication (Default)

By default, Spring Security secures your application with HTTP Basic Authentication.

This means it requires a username and password to access the resources.

When you add spring-boot-starter-security:

Spring Security automatically secures all HTTP endpoints.

The default user credentials are generated as user and a random password, which can be found in the console logs when the application starts.

You can customize this default user by defining your own credentials in application.properties:

spring.security.user.name=admin

spring.security.user.password=admin123

. Password Encoding

It’s a good practice to encode user passwords for security. Spring Security supports password encoding via PasswordEncoder.

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.inMemoryAuthentication()

.withUser("admin").password(passwordEncoder().encode("admin123")).roles("ADMIN")

.and()

.withUser("user").password(passwordEncoder().encode("user123")).roles("USER");

}

Summary

Add the spring-boot-starter-security dependency to your project.

Configure authentication:

Use in-memory authentication, a database-backed user store, or custom authentication mechanisms.

Customize HTTP security:

Secure routes with roles, create custom login forms, and manage access to different parts of your application.

Secure REST APIs with token-based authentication like JWT (if applicable).

Add password encoding for security and use BCryptPasswordEncoder in production.

Test security using Spring's testing features (@WithMockUser).

=========================================================================================================

**5:---Benefits of JPA over JDBC.**

Object-Relational Mapping (ORM): JPA maps Java objects to database tables, making it easier to work with entities and

reducing boilerplate code for CRUD operations.

Reduced Boilerplate Code: With JPA, you don’t need to manually write SQL queries or handle connection management.

It automates object persistence, transactions, and query execution.

Lazy Loading: JPA supports lazy loading, enabling the fetching of related entities only when needed, reducing unnecessary database calls.

Caching: JPA provides first-level and optional second-level caching, improving performance by reducing redundant database queries.

Declarative Transactions: JPA integrates seamlessly with Spring's transaction management,

enabling declarative transaction handling without manual transaction management in JDBC.

Advanced Query Support: JPA provides JPQL (Java Persistence Query Language), which is object-oriented and more

flexible than SQL in JDBC for complex queries.

Automatic Relationship Handling: JPA simplifies handling relationships between entities (e.g., one-to-many, many-to-many)

with automatic foreign key management, reducing manual effort in JDBC.

====================================================================================================================

6:- Handling XML-based configuration in Spring Boot.

In Spring Boot, XML-based configuration can be handled by using the @ImportResource annotation to load XML configuration files.

However, Spring Boot prefers Java-based configuration due to its simplicity and ease of use.

Steps for Handling XML-based Configuration:

Create XML Configuration File:

Define beans in applicationContext.xml (or any custom name).

Example (applicationContext.xml):

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">

<bean id="myBean" class="com.example.MyBean"/>

</beans>

Import XML Configuration in Spring Boot:

Use @ImportResource to load the XML configuration in your @SpringBootApplication or configuration class.

Example:

@SpringBootApplication

@ImportResource("classpath:applicationContext.xml")

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

Access Beans Defined in XML:

Beans defined in the XML file can now be accessed like any other Spring bean.

Notes:

XML-based configuration in Spring Boot is supported but less common compared to Java-based configuration.

It’s recommended to use @Configuration and @Bean for simplicity and flexibility in Spring Boot.

===================================================================================================

**7:---Monitoring application health in Spring.**

In Spring Boot, you can monitor application health using Spring Boot Actuator.

It provides various endpoints to check the health and metrics of the application.

Steps to Monitor Application Health:

Add Spring Boot Actuator Dependency: Add the following dependency in your pom.xml (for Maven) or build.gradle (for Gradle).

Maven:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

Enable Health Check Endpoint:

By default, Spring Boot Actuator exposes a /actuator/health endpoint that shows the health status of the application.

Access the Health Endpoint: After starting the application, you can access the health status via:

http://localhost:8080/actuator/health

Customize Health Checks: You can customize the health check to include additional details or manage components like database, disk space, etc.

Example configuration in application.properties:

management.endpoints.web.exposure.include=health,info

Additional Monitoring: Spring Boot Actuator also provides other endpoints like /metrics, /env, and /info for additional monitoring and information.

Summary:

Spring Boot Actuator simplifies monitoring by providing built-in health checks and various endpoints to monitor application status and metrics.

You can easily enable it with a dependency and customize it via configuration.

Summary:

/health: Shows the overall health status of the application (e.g., "UP").

/metrics: Exposes performance metrics like memory usage, HTTP requests, and more.

/env: Provides environment properties and configuration values.

/info: Displays custom application information like version, name, etc.

=======================================================================================

**8:------------------ Custom exceptions in Spring Boot.**

Creating custom exceptions in Spring Boot allows you to handle specific error conditions more effectively.

Spring provides robust support for handling exceptions using @ControllerAdvice and @ExceptionHandler annotations.

Here's how to create and use custom exceptions in a Spring Boot application.

Create Custom Exception Class

Create a custom exception class that extends RuntimeException or Exception.

package com.example.demo.exception;

public class ResourceNotFoundException extends RuntimeException {

private String resourceName;

private String fieldName;

private Object fieldValue;

public ResourceNotFoundException(String resourceName, String fieldName, Object fieldValue) {

super(String.format("%s not found with %s: '%s'", resourceName, fieldName, fieldValue));

this.resourceName = resourceName;

this.fieldName = fieldName;

this.fieldValue = fieldValue;

}

public String getResourceName() {

return resourceName;

}

public String getFieldName() {

return fieldName;

}

public Object getFieldValue() {

return fieldValue;

}

}

In this example, the ResourceNotFoundException takes the resource name, field name, and field value to provide a meaningful error message.

2. Create Exception Handler with @ControllerAdvice

@ControllerAdvice is used globally to handle exceptions in your application. It allows you to handle exceptions across all controllers.

package com.example.demo.exception;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.context.request.WebRequest;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<?> handleResourceNotFoundException(ResourceNotFoundException ex, WebRequest request) {

ErrorDetails errorDetails = new ErrorDetails(

ex.getMessage(),

request.getDescription(false)

);

return new ResponseEntity<>(errorDetails, HttpStatus.NOT\_FOUND);

}

// Global exception handler for all other exceptions

@ExceptionHandler(Exception.class)

public ResponseEntity<?> handleGlobalException(Exception ex, WebRequest request) {

ErrorDetails errorDetails = new ErrorDetails(

ex.getMessage(),

request.getDescription(false)

);

return new ResponseEntity<>(errorDetails, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

@ExceptionHandler(ResourceNotFoundException.class) ensures that whenever a ResourceNotFoundException is thrown, it is caught by this method.

The ErrorDetails class is a custom class used to define the error response.

3. Create ErrorDetails Class

This class is used to structure the error response.

package com.example.demo.exception;

public class ErrorDetails {

private String message;

private String details;

public ErrorDetails(String message, String details) {

this.message = message;

this.details = details;

}

public String getMessage() {

return message;

}

public String getDetails() {

return details;

}

}

This is a simple POJO that holds the error message and details of the error, like the request URL or other details.

4. Using Custom Exception in Controller

In your controller, you can now throw the ResourceNotFoundException when a resource is not found.

package com.example.demo.controller;

import com.example.demo.exception.ResourceNotFoundException;

import com.example.demo.model.User;

import com.example.demo.service.UserService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/{id}")

public User getUserById(@PathVariable(value = "id") Long userId) {

return userService.findUserById(userId)

.orElseThrow(() -> new ResourceNotFoundException("User", "id", userId));

}

}

If a user is not found, the ResourceNotFoundException is thrown.

This exception is handled globally by GlobalExceptionHandler.

Summary

Custom Exception: ResourceNotFoundException is created to represent a specific error (user not found).

Global Exception Handler: GlobalExceptionHandler handles the exception and formats a custom error response.

Controller: Throws the custom exception when a resource is not found.

Error Response: ErrorDetails is used to return structured error information in JSON format.

========================================================================================================

**9:---- Difference between Spring and Spring Boot.**

Configuration:

Spring: Requires extensive configuration (XML or Java-based).

Spring Boot: Provides auto-configuration, minimizing the need for manual setup.

Project Setup:

Spring: Developers must set up the entire application infrastructure, including web servers and configurations.

Spring Boot: Simplifies project setup with default configurations and embedded servers (e.g., Tomcat, Jetty).

Boilerplate Code:

Spring: Requires a lot of boilerplate code for tasks like setting up application contexts and configuring beans.

Spring Boot: Reduces boilerplate code with features like auto-configuration and built-in support for common tasks (e.g., logging, security).

Dependency Management:

Spring: Dependency management is manually handled by the developer.

Spring Boot: Manages dependencies through the spring-boot-starter mechanism, offering pre-configured dependencies.

Deployment:

Spring: Requires external server setups like Tomcat or Jetty for deployment.

Spring Boot: Can be run as a standalone application (embedded server), simplifying deployment.

Purpose:

Spring: Provides a comprehensive framework for building Java-based enterprise applications.

Spring Boot: Aimed at simplifying Spring application development by reducing configuration and enhancing productivity.

In summary, Spring Boot simplifies and accelerates the development process compared to Spring, which requires more manual setup and configuration.

==========================================================================================================================

**10:----Dependency Injection and Inversion of Control.**

Inversion of Control (IoC): A design principle where the control of object creation and management is transferred from the

application to the framework

or container (e.g., Spring). The framework controls the flow and object lifecycle, instead of the application controlling it.

Dependency Injection (DI): A type of IoC where dependencies (objects or services) are provided (injected) into a class instead of the

class creating them itself.

Summary:

IoC: The framework (e.g., Spring) controls object creation and lifecycle.

DI: The framework provides dependencies to classes, instead of classes creating their own.

=====================================================================

**11: spring bean life cycle**

Summary of Interfaces/Methods:

Aware Interfaces: BeanNameAware, BeanFactoryAware, etc.

Lifecycle Interfaces: InitializingBean, DisposableBean

Lifecycle Annotations: @PostConstruct, @PreDestroy

Customization: init-method and destroy-method in XML/Java configuration.

==============================================================================

**12::-----------------------Difference between Controller and RestController.**

Purpose:

@Controller: Used for traditional Spring MVC controllers that return views (HTML pages).

@RestController: Used for RESTful web services, returning data (e.g., JSON or XML) instead of views.

Response Handling:

@Controller: Typically returns a view name that resolves to a view (e.g., JSP, Thymeleaf).

@RestController: Automatically applies @ResponseBody to methods, returning the response body directly (e.g., JSON).

Use Case:

@Controller: Suitable for MVC applications where the response is a view (UI).

@RestController: Suitable for REST APIs where the response is usually in JSON format

Summary:

@Controller: Used for MVC views, returns HTML.

@RestController: Used for REST APIs, returns JSON/XML data directly.

======================================================

**13:----Transaction management in Spring.**

either transaction should commit completely or rollback ( all data should be saved in different table)

ex: customer , address ( one to one mapping)

so customer data

address data :

address object is null the customer data will saved but address data will give null pointer exception

so to solve this problem transaction management come in picture

annotate method :

@EnableTransactionManagement (added in the main class)

@Transactional

either all the transaction commit in database or any disturbance occur it will be rollback

==========

How the Transaction Works Internally

When a method annotated with @Transactional is called (like createUser or createUserWithException),

Spring creates a proxy around the UserService bean.

The proxy intercepts the method call, starts a transaction by calling the PlatformTransactionManager,

and commits or rolls back the transaction depending on the method outcome.

If an exception occurs in the method, Spring will roll back the transaction (if the exception type matches the configuration, e.g., rollbackFor).

After the method completes, Spring commits the transaction if no errors were encountered.

======================

Propagation: Defines how a transaction should behave when there is an existing transaction.

REQUIRED (default): Use the current transaction if exists, or create a new one.

REQUIRES\_NEW: Always create a new transaction.

MANDATORY: Require an existing transaction.

NESTED: Execute within a nested transaction (only supported by some databases).

Isolation: Defines the level of isolation for the transaction to prevent issues like dirty reads, non-repeatable reads, and phantom reads.

DEFAULT

READ\_UNCOMMITTED

READ\_COMMITTED

REPEATABLE\_READ

SERIALIZABLE

Rollback: Defines the conditions under which a transaction should be rolled back. By default, Spring will roll back on unchecked exceptions

(e.g., RuntimeException) but can be configured to roll back on checked exceptions as well.

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

@Transactional

public void createUser(User user) {

// Transaction will start here

userRepository.save(user);

// Additional business logic can be added here

}

@Transactional(rollbackFor = Exception.class)

public void createUserWithException(User user) throws Exception {

userRepository.save(user);

// Simulating an exception

if (user.getAge() < 18) {

throw new Exception("Age must be 18 or above");

}

}

}

===========

1. Dirty Read

A dirty read occurs when a transaction reads data that has been written by another transaction

but has not yet been committed. Since the other transaction could still be rolled back,

the data read by the first transaction might never actually be saved, leading to inconsistent or incorrect results.

Example:

Transaction 1 updates a record (e.g., the balance of a bank account) but has not yet committed the changes.

Transaction 2 reads the updated balance before Transaction 1 commits or rolls back.

If Transaction 1 later rolls back, Transaction 2 has read data that was never actually saved to the database — this is a dirty read.

2. Non-Repeatable Read

A non-repeatable read happens when a transaction reads the same data twice, but the data has been modified (committed)

by another transaction between the two reads. This can lead to inconsistent results because the data may have changed

after the first read, which can be problematic if you expect the data to remain the same throughout the transaction.

Example:

Transaction 1 reads a record from the database.

Transaction 2 modifies and commits changes to the same record.

Transaction 1 reads the same record again and sees the updated value, even though it was not expecting that change.

Phantom Read

A phantom read occurs when a transaction reads a set of rows that match a certain condition, but another transaction inserts,

deletes, or updates rows that would affect the result set. When the first transaction re-queries the database,

the result set has changed, which is unexpected behavior.

Example:

Transaction 1 performs a query to retrieve a set of rows based on certain criteria.

Transaction 2 inserts, updates, or deletes rows that match the criteria.

Transaction 1 performs the same query again and sees a different result set because the rows have changed

(i.e., "phantom" rows have appeared or disappeared).

======

Transaction Isolation Levels and Their Effect on These Anomalies

These anomalies are directly related to the isolation level of the transaction. The isolation level controls how

much one transaction is isolated from the others. The following are the four standard isolation levels

defined by the SQL standard, listed from least to most restrictive:

READ UNCOMMITTED:

Dirty Reads: Allowed

Non-Repeatable Reads: Allowed

Phantom Reads: Allowed

This is the least restrictive isolation level. Transactions can see uncommitted changes made by other transactions.

READ COMMITTED (default for many databases):

Dirty Reads: Not Allowed

Non-Repeatable Reads: Allowed

Phantom Reads: Allowed

This isolation level ensures that a transaction will only read committed data. However,

it still allows non-repeatable reads (i.e., if a row is modified by another transaction after it’s read).

REPEATABLE READ:

Dirty Reads: Not Allowed

Non-Repeatable Reads: Not Allowed

Phantom Reads: Allowed

This level guarantees that once a transaction reads a row, that row cannot be changed by another transaction.

However, it still allows new rows to be inserted that would match the query's condition.

SERIALIZABLE:

Dirty Reads: Not Allowed

Non-Repeatable Reads: Not Allowed

Phantom Reads: Not Allowed

This is the highest isolation level and prevents all anomalies. It ensures complete isolation between

transactions by locking the data and ensuring that no other transactions can modify, insert,

or delete the data that the current transaction is working with.

import org.springframework.transaction.annotation.Isolation;

import org.springframework.transaction.annotation.Transactional;

@Transactional(isolation = Isolation.SERIALIZABLE)

public void myMethod() {

// business logic here

}

Conclusion

To summarize:

Dirty Reads: A transaction reads uncommitted data, potentially leading to inconsistencies.

Non-Repeatable Reads: A transaction reads the same data twice, but the data has changed due to another transaction.

Phantom Reads: A transaction's result set changes due to inserts, deletes, or updates in another transaction.

These anomalies can be controlled by adjusting the transaction isolation level, with SERIALIZABLE providing the highest level

of consistency and isolation, and READ UNCOMMITTED being the least restrictive.

======================================================================================================================

Section 5: ----------- Java concept

**1:--- Difference between Abstract Class and Interface.**

. Abstract Class

Definition: An abstract class is a class that cannot be instantiated on its own and may have both abstract (without implementation)

and non-abstract (with implementation) methods.

Usage: Use an abstract class when you want to provide a common base class with shared implementation for related classes

and when you want to prevent direct instantiation.

Key Points:

Can have both abstract and concrete methods.

Can have fields (instance variables).

Supports constructor.

A class can inherit from only one abstract class (single inheritance).

abstract class Animal {

abstract void makeSound(); // abstract method

void breathe() { // concrete method

System.out.println("Breathing...");

}

}

class Dog extends Animal {

void makeSound() {

System.out.println("Bark");

}

}

When to Use:

When you need a base class with shared functionality and some methods that must be implemented by subclasses.

2. Interface

Definition: An interface is a contract that defines methods without implementation. A class that implements the interface must provide

concrete implementations for all the methods.

Usage: Use an interface when you want to define a contract for unrelated classes, or when a class needs to implement multiple behaviors.

Key Points:

Can only have abstract methods (prior to Java 8).

Cannot have fields (only constants).

No constructors.

A class can implement multiple interfaces (multiple inheritance).

interface Animal {

void makeSound(); // abstract method

}

class Dog implements Animal {

public void makeSound() {

System.out.println("Bark");

}

}

When to Use:

When you need to define a common contract for classes from different inheritance hierarchies (polymorphism across unrelated classes).

==========================================================

**2:-What is a Functional Interface?Can it contain default methods?**

Key Points:

Contains one abstract method.

Can have multiple default and static methods.

Can be annotated with @FunctionalInterface (optional but recommended for clarity).

Can It Contain Default Methods?

Yes, a functional interface can contain default methods. These methods provide a default

implementation, and the abstract method remains the only method that must be implemented by the class or lambda expression.

=====================================================================

**3:--- What are its predefined types?**

In Java, functional interfaces are often used with lambda expressions and streams. Java provides several predefined functional interfaces

in the java.util.function package that cover common use cases. Here are some of the most commonly used predefined functional interfaces:

1. Predicate<T>

Definition: Represents a function that takes a single argument and returns a boolean value.

Used for: Conditional checks or filtering.

Method: boolean test(T t);

Example:

Predicate<Integer> isEven = x -> x % 2 == 0;

System.out.println(isEven.test(4)); // true

2. Function<T, R>

Definition: Represents a function that takes one argument of type T and returns a result of type R.

Used for: Transforming data or applying a function.

Method: R apply(T t);

Example:

Function<String, Integer> length = str -> str.length();

System.out.println(length.apply("Hello")); // 5

3. Consumer<T>

Definition: Represents an operation that takes a single argument of type T and returns no result (void).

Used for: Performing actions, such as printing or modifying data.

Method: void accept(T t);

Example:

Consumer<String> print = str -> System.out.println(str);

print.accept("Hello, World!"); // Prints: Hello, World!

4. Supplier<T>

Definition: Represents a function that takes no arguments and returns a result of type T.

Used for: Generating values or providing data.

Method: T get();

Example:

Supplier<Double> random = () -> Math.random();

System.out.println(random.get()); // Generates a random number

5. UnaryOperator<T>

Definition: A special case of Function where the input and output types are the same.

Used for: Operations that modify a value of type T and return a result of the same type.

Method: T apply(T t);

Example:

UnaryOperator<Integer> square = x -> x \* x;

System.out.println(square.apply(5)); // 25

6. BinaryOperator<T>

Definition: A special case of BiFunction where both arguments and the result are of the same type T.

Used for: Operations involving two arguments of the same type and returning a result of the same type (e.g., addition, multiplication).

Method: T apply(T t1, T t2);

Example:

BinaryOperator<Integer> add = (a, b) -> a + b;

System.out.println(add.apply(3, 4)); // 7

7. BiFunction<T, U, R>

Definition: Represents a function that takes two arguments of types T and U, and returns a result of type R.

Used for: Functions that operate on two different types and return a result.

Method: R apply(T t, U u);

BiFunction<Integer, Integer, String> sumToString = (a, b) -> "Sum: " + (a + b);

System.out.println(sumToString.apply(2, 3)); // "Sum: 5"

8. BiPredicate<T, U>

Definition: A functional interface that takes two arguments and returns a boolean value.

Used for: Conditional checks or filters on two arguments.

Method: boolean test(T t, U u);

Example:

BiPredicate<Integer, Integer> isGreaterThan = (a, b) -> a > b;

System.out.println(isGreaterThan.test(5, 3)); // true

=====================================================

**4---------------Difference between Runnable and Callable.**

Both Runnable and Callable are functional interfaces used to represent tasks that can be executed by multiple threads in Java

Runnable Interface

Definition: Represents a task that can be executed by a thread. It is designed for tasks that do not return a result

and cannot throw checked exceptions.

Method: void run()

Use Case: Ideal for tasks that perform an operation without needing to return a result.

class MyRunnable implements Runnable {

@Override

public void run() {

System.out.println("Task is running");

}

}

public class RunnableExample {

public static void main(String[] args) {

Runnable myRunnable = new MyRunnable();

Thread thread = new Thread(myRunnable);

thread.start(); // Starts the thread and executes the run() method

}

}

. Callable Interface

Definition: Similar to Runnable, but it allows the task to return a result and handle checked exceptions.

It is part of the java.util.concurrent package.

Method: V call() throws Exception

Use Case: Ideal for tasks that need to return a result or throw checked exceptions during execution.

import java.util.concurrent.\*;

class MyCallable implements Callable<Integer> {

@Override

public Integer call() throws Exception {

System.out.println("Task is running in Callable");

return 42; // Return a result

}

}

public class CallableExample {

public static void main(String[] args) throws Exception {

ExecutorService executor = Executors.newSingleThreadExecutor();

Future<Integer> future = executor.submit(new MyCallable());

System.out.println("Result from Callable: " + future.get()); // Retrieves the result

executor.shutdown();

}

}

======================================================================

5:---Internal working of HashMap.Default capacity and load factor.

Internal Working of HashMap in Java

HashMap in Java is part of the java.util package and provides a key-value mapping. Internally, HashMap uses an array of buckets to store data,

where each bucket contains a linked list or tree of entries (depending on the size of the bucket) for handling hash collisions.

Key Components:

Buckets: The underlying data structure is an array of buckets. Each bucket holds a linked list or tree of entries (which are key-value pairs).

Hashing: The HashMap computes a hash code for each key, and the hash code determines the index (bucket) where the key-value pair is stored.

This is done via the hash() method.

Collisions: If two keys have the same hash code (or hash to the same bucket), a collision occurs. HashMap handles collisions using linked

lists (before Java 8) or balanced trees (since Java 8 when the list size exceeds 8).

Entry: The basic unit in a HashMap is an entry, which consists of the key, value, hash code, and a reference to the next entry

(in case of a collision).

Basic Working:

Putting a value:

HashMap computes the hash code of the key.

The hash code is then used to determine the index of the bucket.

If the bucket is empty, the key-value pair is added directly.

If the bucket contains other entries (due to collisions), a linked list or tree structure is used to store the new entry.

Getting a value:

HashMap computes the hash code of the key.

It finds the corresponding bucket and checks for the key in the linked list or tree.

Removing a value:

HashMap computes the hash code of the key, locates the bucket, and removes the key-value pair from the list or tree.

Default Capacity and Load Factor

Default Capacity:

The default capacity of a HashMap is 16. This is the initial size of the underlying array (number of buckets).

Default Load Factor:

The default load factor is 0.75. The load factor determines when the HashMap should resize its internal array

to accommodate more entries. It defines the threshold at which the capacity will be increased.

Formula for resizing:

The HashMap will resize when the number of entries exceeds the product of the current capacity and the load factor:

Threshold

=

capacity

×

load factor

Threshold=capacity×load factor

For example, if the capacity is 16 and the load factor is 0.75, the threshold for resizing will be:

16

×

0.75

=

12

16×0.75=12

So, once 12 entries are inserted into the HashMap, it will resize (double its capacity).

============================================================

6:----Multithreading concepts:Executor service.

Executor Service in Java (Multithreading)

ExecutorService is a higher-level replacement for the traditional way of managing threads in Java.

It is part of the java.util.concurrent package and provides an easier and more efficient way to manage and control thread execution.

Key Concepts:

Executor: The basic interface for managing tasks.

ExecutorService: Extends Executor and adds lifecycle management methods such as shutdown(), submit(), etc.

Thread Pool: Executor manages a pool of threads to execute submitted tasks, which helps reuse threads and avoid the overhead of creating a

new thread for every task.

public class Test {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService es = Executors.newFixedThreadPool(2);

Callable<Integer> task = () -> {

Thread.sleep(5000);

return 42;

};

Future<Integer> submit = es.submit(task);

Integer result = submit.get(); // This will block until the task completes

System.out.println("Task result: " + result); // Output: Task result: 42

// Shutdown the executor

es.shutdown();

}

}

===========================================================================================

**7:--- Volatile, atomic variables, locks, and semaphores.**

volatile variable

A volatile variable is one where changes to the variable are immediately visible to all threads.

This ensures that the value of the variable is not cached or optimized by the compiler.

When to use: Use volatile when you need a simple flag or variable that multiple threads need to access,

and you want changes to that variable to be immediately visible across all threads.

It's typically used for flags or state variables that are being shared between threads.

lock :-

Locks

A Lock is a more flexible and powerful way of controlling access to shared resources compared to the synchronized keyword.

Java provides classes like ReentrantLock that allow locking with additional features such as try-lock and timed lock.

When to use: Use locks when you need more control over synchronization, such as being able to interrupt threads waiting for a lock,

or when you want to avoid deadlocks by acquiring multiple locks in a specific order.

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class LockExample {

private Lock lock = new ReentrantLock();

public void criticalSection() {

lock.lock(); // Acquiring the lock

try {

// Critical section of code

System.out.println(Thread.currentThread().getName() + " is inside critical section");

} finally {

lock.unlock(); // Releasing the lock

}

}

public static void main(String[] args) throws InterruptedException {

LockExample example = new LockExample();

// Create multiple threads that will try to access the critical section

Thread[] threads = new Thread[5];

for (int i = 0; i < 5; i++) {

threads[i] = new Thread(example::criticalSection);

threads[i].start();

}

// Wait for all threads to finish

for (Thread t : threads) {

t.join();

}

}

}

**semaphores**

. Semaphores

A Semaphore is a signaling mechanism used to manage access to a finite set of resources.

It maintains a count that represents the number of available permits.

Threads can acquire permits before accessing a resource, and they must release the permit when done.

When to use:

Use semaphores when you want to limit the number of threads accessing a particular resource or service,

such as controlling a pool of database connections or limiting the number of concurrent tasks.

import java.util.concurrent.Semaphore;

public class SemaphoreExample {

private static final Semaphore semaphore = new Semaphore(3); // Allows 3 threads at a time

public void accessResource() {

try {

semaphore.acquire(); // Acquire a permit

System.out.println(Thread.currentThread().getName() + " is accessing the resource");

Thread.sleep(2000); // Simulating work

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

semaphore.release(); // Release the permit

System.out.println(Thread.currentThread().getName() + " has released the resource");

}

}

public static void main(String[] args) throws InterruptedException {

SemaphoreExample example = new SemaphoreExample();

// Create 5 threads, but only 3 can access the resource at once

Thread[] threads = new Thread[5];

for (int i = 0; i < 5; i++) {

threads[i] = new Thread(example::accessResource);

threads[i].start();

}

// Wait for all threads to finish

for (Thread t : threads) {

t.join();

}

}

}

===============================================================

**8---------------------Waiting for threads to complete in Executor Framework.**

shutdown() is called to prevent further tasks from being submitted.

awaitTermination() is used to block the main thread until all tasks are finished or the specified timeout (60 seconds in this case) is reached.

2. Using Future.get() for Waiting on Callable Tasks

If you're submitting tasks that return a result (using Callable), you can wait for their completion by calling Future.get().

The Future object is returned when you submit a task,

and calling get() will block the main thread until the task completes, returning the result or throwing an exception if the task failed.

**Section 6:--------------------------- AWS**

**1:- AWS services used in the project:EC2 and its role in deployment.**

In an AWS deployment project, Amazon EC2 (Elastic Compute Cloud) plays a crucial role as the compute resource to run your application.

Here's a concise overview of its role:

Provisioning Virtual Servers: EC2 provides scalable virtual machines (instances) that you can use to host your application,

whether it's a web server, backend service, or database.

Custom Configuration: You can choose the instance type (CPU, memory, storage) and operating system based on your application requirements.

Deployment: EC2 instances are commonly used to deploy applications that require full control over the server environment,

such as custom applications or services.

Scaling: EC2 integrates with Auto Scaling to automatically adjust the number of instances based on traffic or workload,

ensuring availability and cost-efficiency.

Security: EC2 instances are secured using Security Groups, Key Pairs for SSH access, and IAM roles for permissions to access other AWS services.

Integration: EC2 instances can work with other AWS services like RDS for databases, S3 for storage, and ECR for Docker containers.

In summary, EC2 provides the compute power needed to run and scale applications, offering flexibility, control, and integration with

other AWS services.

=====================================================================

**2------------ S3 usage, data archiving, and retention policies.**

Data archiving refers

to the process of moving data that is no longer actively used or frequently accessed to a long-term storage solution

for preservation.The archived data is stored in a secure, cost-effective way, with the understanding that it may need to be retrieved later

for reference, compliance, or historical purposes.

Amazon S3 Usage:

Data Storage: Store files, backups, logs, media, and application data.

Static Website Hosting: Host static content like HTML, CSS, and JavaScript files.

Scalable and Secure: Easily scale storage needs and secure data with encryption and access controls.

Data Archiving in S3:

S3 Glacier: Low-cost storage for long-term archiving, with retrieval times ranging from minutes to hours.

S3 Glacier Deep Archive: Even cheaper than Glacier, ideal for rarely accessed data that requires long-term retention.

Retention Policies:

Lifecycle Policies: Automatically transition data between storage classes (e.g., from S3 Standard to Glacier) or delete data after a set period.

Versioning: Retain previous versions of objects for recovery.

Object Lock: Prevent deletion or modification of data for compliance (immutability).

==================================================================

**3:-- Lambda expressions in AWS.**

AWS Lambda Expressions refer to the use of Lambda functions in AWS for serverless computing.

A Lambda function is a small, event-driven piece of code that runs in response to triggers

(such as HTTP requests, file uploads to S3, or changes in DynamoDB).

Key Points:

Event-Driven: Lambda runs in response to specific events (e.g., API Gateway, S3, CloudWatch).

Serverless: No need to provision or manage servers; AWS automatically scales the execution.

Short-lived: Ideal for tasks that run for a short duration, with automatic scaling and cost based on execution time.

Supported Languages: Supports Node.js, Python, Java, Go, C#, and more.

Lambda simplifies backend development by handling infrastructure, allowing developers to focus solely on code logic.

Example:

Suppose you want to automatically resize images when they're uploaded to an S3 bucket.

Upload Image to S3: When a user uploads an image to your S3 bucket, it triggers the Lambda function.

Lambda Function: The Lambda function processes the image (e.g., resizing it).

Store Resized Image: After processing, the resized image is saved to another S3 bucket or directory.

Flow:

Event: Image uploaded to S3.

Lambda Function: Triggered to resize the image.

Action: Store the resized image back in S3.

This process is fully serverless, meaning you don’t need to manage any infrastructure—AWS handles scaling and execution based on the event.

Benefits:

No server management: You only focus on the code.

Automatic scaling: Lambda scales automatically with the number of events.

Cost-efficient: Pay only for the compute time your function uses.

============================================================

**4:---Handling errors during deployment on Docker and Kubernetes.**

Key Differences:

Liveness: Checks if the application is running. If it fails, the container is restarted.

Readiness: Checks if the application is ready to handle traffic. If it fails, traffic is stopped, but the container is not restarted.

Docker:

Logs: Use docker logs <container\_id> to check container logs for errors.

Container Health: Ensure health checks are defined in Dockerfile or docker-compose.yml to automatically restart failing containers.

Error Codes: Pay attention to exit codes (e.g., non-zero exit codes indicate failure).

Docker Events: Use docker events to get a real-time stream of events and errors.

Kubernetes:

Pod Logs: Use kubectl logs <pod\_name> to view logs of failing pods.

Pod Status: Use kubectl describe pod <pod\_name> to get detailed status and error messages (e.g., CrashLoopBackOff).

Health Checks: Implement liveness and readiness probes in the pod definition to detect and handle failures (restarts or traffic rerouting).

Rolling Updates: Use rolling updates to deploy updates with minimal disruption, allowing rollback on failure.

Resource Limits: Define proper resource requests and limits to prevent pods from crashing due to resource exhaustion.

Summary:

Docker: Use logs, health checks, and exit codes to identify and resolve issues.

Kubernetes: Leverage logs, describe pod, probes, and rolling updates for error handling and recovery during deployment.

The CrashLoopBackOff :----------------

is a status in Kubernetes that indicates a pod is repeatedly crashing and failing to start successfully.

It means that a container inside a pod is failing to start, and Kubernetes is trying to restart it,

but the container keeps crashing each time it attempts to start. After several retries, Kubernetes puts the pod into a "back-off" state,

where it will pause for a certain amount of time before trying again.

Key Concepts:

CrashLoopBackOff State:

Crash: The container inside the pod is failing due to some error (e.g., application crash, configuration error, or missing dependency).

Loop: Kubernetes keeps attempting to restart the container because it doesn't reach a healthy state.

BackOff: After several failed attempts to restart the container, Kubernetes will enter a back-off mode,

where it waits longer before each subsequent restart attempt.

Common Causes of CrashLoopBackOff:

Application errors: Application exceptions or unhandled errors causing the container to exit.

Invalid configuration: Missing or incorrect environment variables, configuration files, or arguments.

Out of resources: The container might be using too much CPU or memory, causing it to be killed by Kubernetes.

Failed health checks: Liveness/readiness probes failing because the application is not responding in time.

Incorrect Docker image or entrypoint: The Docker image might be missing dependencies or configured incorrectly.

========================================

**5:- Logging in Kubernetes.**

Centralized Logging:

Use logging solutions like Elasticsearch, Fluentd, and Kibana (EFK) stack or Prometheus and Grafana to aggregate and analyze logs

from multiple pods.

Log Retention and Management:

Kubernetes itself doesn’t store logs long-term, so integrate with external systems like CloudWatch,

Elasticsearch, or Stackdriver for centralized log collection and retention.

Summary:

kubectl logs: View container logs.

==========================================================================================================================================

**Section 7 :------------------- Miscellaneous**

1:---Maven and Build Tools:Transitive dependencies.

In Maven, transitive dependencies refer to dependencies that are required by your direct dependencies.

When you include a dependency in your project,

Maven automatically resolves and includes its dependencies, recursively, into your project.

For example,

if your project depends on library A, and A depends on library B, Maven will also include library B in your project,

even though you didn't explicitly declare it.

This process is managed through dependency scopes (e.g., compile, test, runtime), ensuring that only the relevant dependencies

are included in the appropriate phases of the build lifecycle.

=====================================

**2:-------Resolving dependency conflicts.**

Maven's Conflict Resolution Strategy:

Nearest wins: If two versions of the same dependency are found, Maven picks the one nearest to your project in the dependency tree.

Exclusion: You can exclude a version of a dependency to prevent it from being included in your project.

<dependency>

<groupId>com.example</groupId>

<artifactId>library-B</artifactId>

<version>1.0.0</version>

<exclusions>

<exclusion>

<groupId>com.example</groupId>

<artifactId>library-X</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependencies>

<dependency>

<groupId>com.example</groupId>

<artifactId>library-A</artifactId>

<version>1.0.0</version>

</dependency>

<dependency>

<groupId>com.example</groupId>

<artifactId>library-B</artifactId>

<version>1.0.0</version>

</dependency>

</dependencies>

dependency tree

[Your Project]

├── library-A (depends on library-X:1.0.0)

└── library-B (depends on library-X:2.0.0)

Conflict:

library-A pulls in library-X version 1.0.0.

library-B pulls in library-X version 2.0.0.

Maven will resolve this conflict and choose one version of library-X.

===================================================

**3:-Use of plugins in Maven.**

Plugins in Maven are essential tools that help automate and customize various tasks in your build lifecycle,

and they are easy to configure within the pom.xml.

In Maven, plugins are used to extend Maven's capabilities and automate tasks in the build lifecycle,

such as compiling code, running tests, packaging, and deploying artifacts.

Key Uses of Plugins:

Compiling code (e.g., maven-compiler-plugin).

Running tests (e.g., maven-surefire-plugin).

Packaging the project (e.g., maven-jar-plugin).

Deploying (e.g., maven-deploy-plugin).

==================================================================================

**4:- Debugging scenarios:Resolving "no bean found" errors.**

Ensure the class is annotated with @Component, @Service, @Repository, or @Controller to define it as a Spring bean.

Verify that the package containing the class is included in the component scanning configuration.

Use @Qualifier to resolve ambiguity if multiple beans of the same type exist.

Check Java-based or XML configuration for correct bean definitions, especially in non-auto-scanned beans.

Confirm the active Spring profile if the bean is profile-specific, using spring.profiles.active.

===================================================================================================

5:---How web.xml works in a Spring application.

In a Spring application, web.xml is used for configuring the Servlet container and initializing the Spring framework before handling HTTP requests.

Key Functions:

Servlet Configuration: It defines the DispatcherServlet, which acts as the front controller for Spring MVC.

Context Configuration: It configures the Spring ApplicationContext to load beans and resources.

Servlet Mappings: It maps the DispatcherServlet to specific URL patterns (e.g., / or /app/\*)

=======================================================================

**6:----How to analyze and improve query performance in a database.**

The EXPLAIN output helps you understand whether your query is optimized,

how it accesses data (e.g., using indexes or scanning the table), and guides you in making optimizations (like adding indexes).

+----+-------------+-----------+------------+------+---------------+---------+---------+-------+------+----------+--------------------------+

| id | select\_type | table | type | key | key\_len | ref | rows | Extra |

+----+-------------+-----------+------------+------+---------------+---------+---------+-------+------+----------+--------------------------+

| 1 | SIMPLE | employees | range | idx\_dept\_id | 4 | NULL | 50 | Using where |

+----+-------------+-----------+------------+------+---------------+---------+---------+-------+------+----------+--------------------------+

1. Check Query Execution Plan

Why: The execution plan shows how the database executes a query and helps identify inefficiencies.

How: Use commands like EXPLAIN or EXPLAIN ANALYZE in SQL to view the query plan.

Example:

EXPLAIN SELECT \* FROM employees WHERE department\_id = 10;

This will show if the database is using an efficient index or scanning the entire table.

2. Use Indexes

Why: Indexes speed up data retrieval by providing fast access paths.

How: Add indexes on columns that are frequently used in WHERE, JOIN, or ORDER BY clauses

CREATE INDEX idx\_department\_id ON employees(department\_id);

3. Avoid SELECT \* (Select Specific Columns)

Why: Fetching unnecessary columns increases data transfer and processing time.

How: Select only the columns you need.

Example:

SELECT employee\_id, employee\_name FROM employees WHERE department\_id = 10;

.4 Optimize Joins

Why: Joining large tables can be slow if not optimized.

How: Use appropriate indexes and filter rows early with WHERE conditions before joining.

Example:

SELECT e.employee\_name, d.department\_name

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id

WHERE d.department\_name = 'Sales';

5 Limit the Result Set

Why: Returning too many rows can slow down performance, especially for large datasets.

How: Use LIMIT or TOP to return only a subset of rows.

SELECT employee\_name FROM employees LIMIT 10;

6 Use Caching:

Cache frequently used query results, reducing the need to run the same query multiple times.

7--Batch Updates/Inserts:

For large data modifications, use batch processing to reduce overhead caused by individual row updates or inserts.

8 Use Proper Data Types:

Ensure that columns and parameters use the correct data types to reduce unnecessary casting and improve query performance.

Conclusion:

By analyzing the query execution plan, using proper indexes, avoiding unnecessary columns,

optimizing joins, limiting results, and updating statistics, you can improve database query performance significantly.

===================================================================================================================**ection 8 : Problem solving scenario**

**1:--- Efficient searching in large datasets:Deciding the right data structure based on requirements.**

Choosing the Right Data Structure for Search

For Exact Matches:

Use Hash Tables when fast lookups are required, especially for key-value pairs.

Use Binary Search Trees or Self-Balancing Trees for ordered data with efficient searches.

For Prefix or String Matching:

Use a Trie for fast prefix searches, auto-completion, or dictionary lookups.

For Range Queries:

Use Binary Search Trees, B+ Trees, or AVL Trees where you need to find all values within a given range.

For Efficient Top N Queries:

Use Heaps (Priority Queue) to find the largest or smallest elements efficiently.

For Large Datasets with Disk Storage:

Use B-Trees or B+ Trees for databases and file systems, as they are optimized for disk-based operations.

For Membership Testing:

Use Bloom Filters when you need a very space-efficient solution with acceptable false positives.

Conclusion:

The right data structure for efficient searching depends on your specific use case:

For general-purpose searching, a Hash Table is usually the fastest.

For ordered searches or range queries, consider using a Binary Search Tree or B+ Tree.

For prefix-based searches, use a Trie.

If space efficiency is paramount and false positives are acceptable, use a Bloom Filter.

============================================================================================================

2:----Maintain a recently searched item list in a collection.

To maintain a recently searched item list in a collection, we can use a queue or deque (double-ended queue),

as these allow us to efficiently manage the order of elements, ensuring that the most recent items are easily

accessible while removing older items when the collection exceeds a set limit (e.g., a fixed number of most recent searches).

A Deque is ideal because it allows adding elements at both ends efficiently and can help us implement a Least Recently Used

(LRU) cache-like behavior if needed.

Here's how we can do it in Java:

Alternative Data Structures:

If you don't need to maintain the order, you could use other structures like:

LinkedHashSet: To maintain the order of insertion but automatically eliminate duplicates

(although in this case, we are adding recent searches and may not require removing duplicates).

===================================================================

3-------------Buy and sell stocks for maximum profit in a week.

public class StockProfit {

// Function to calculate maximum profit

public static int maxProfit(int[] prices) {

if (prices == null || prices.length < 2) {

return 0;

}

// Initialize variables

int minPrice = Integer.MAX\_VALUE; // To track the minimum price

int maxProfit = 0; // To track the maximum profit

// Traverse through the array of stock prices

for (int price : prices) {

// Update minPrice if the current price is lower than the previously tracked minPrice

if (price < minPrice) {

minPrice = price;

}

// Calculate the profit if we sold at the current price

int profit = price - minPrice;

// Update maxProfit if the current profit is higher than previously tracked maxProfit

if (profit > maxProfit) {

maxProfit = profit;

}

}

// Return the maximum profit found

return maxProfit;

}

public static void main(String[] args) {

// Example stock prices for a week (7 days)

int[] prices = {7, 1, 5, 3, 6, 4};

// Call the maxProfit function and print the result

System.out.println("Maximum Profit: " + maxProfit(prices));

}

}