Software Development (JAVA) mthree

DAY 1: 16/09/2024

# Software Development Life Cycle (SDLC)

* SDLC is the software development process used for development of software, from inception to product release.
* The purpose of SDLC is to produce fully functional, effective, easy to use and easily maintainable software within estimated time and cost.

## SDLC Phases:

There are following phases in the Software Development Life Cycle: Planning, Analysis, Design, Development, and Maintenance.

* Requirement Gathering and Analysis: In the Requirements Gathering phase, the customer's needs are collected to define the product clearly. Any unclear points are discussed, and the project’s feasibility is checked. The finalised requirements are written in an SRS document, which is reviewed by both developers and the customer for understanding.
* Design : In this phase, the requirement gathered in the SRS document is used as an input and software architecture that is used for implementing system development is derived.
* Implementation : In this phase, the requirement gathered in the SRS document is used as an input and software architecture that is used for implementing system development is derived.
* Verification : Testing is done to verify whether the application behaves as expected and according to what was documented in the requirements analysis phase.
* Deployment and Maintenance: After thorough testing and resolution issues, the software is deployed for customer use. Following its release, a maintenance team handles any post-production issues to ensure smooth operation and address any bugs or problems that arise after the software is in use.

DAY 6: 23/09/2024

# Hashmap

A HashMap in Java is a collection class that implements the Map interface, allowing you to store key-value pairs. It is part of the java.util package and is widely used for its ability to provide fast access to data using keys.

Program:

package com.cache;

import java.util.HashMap;

import java.util.Map;

public class SimpleCache<K,V> {

private final Map<K,V> cache;

public SimpleCache(){

this.cache = new HashMap<>();

}

public void put(K key, V value){

cache.put(key, value);

}

public V get(K key){

return cache.get(key);

}

public void remove(K key){

cache.remove(key);

}

public void clear(){

cache.clear();

}

public int size(){

return cache.size();

}

public static void main(String[] args) {

SimpleCache<String, String> cache = new SimpleCache<>();

cache.put("key1", "value1");

cache.put("key2", "value2");

cache.put("key3", "value3");

System.out.println(cache.get("key1"));

cache.remove("key2");

System.out.println(cache.size());

cache.clear();

System.out.println(cache.size());

}

}

## Linked Hashmap

A Hashmap in Java is a type of Map that combines the features of both a HashMap and a LinkedList. It provides the same basic functionality as a HashMap, but with an additional feature: it maintains the order of elements based on the order in which they were inserted into the map (insertion order).

Key Features:

* Insertion order
* Access order
* Performance
* Non-Synchronized

Program:

package com.cache;

import java.util.LinkedHashMap;

import java.util.Map;

public class LRUCache<K,V> extends LinkedHashMap<K,V> {

private final int capacity;

public LRUCache(int capacity){

super(capacity, 0.75f, true);

this.capacity = capacity;

}

@Override

protected boolean removeEldestEntry(Map.Entry<K,V> eldest){

return size() > capacity;

}

public static void main(String[] args) {

LRUCache<String, String> cache = new LRUCache<>(3);

cache.put("key1", "value1");

cache.put("key2", "value2");

cache.put("key3", "value3");

System.out.println(cache);

cache.get("key2");

cache.put("key4", "value4");

cache.put("key5", "value5");

System.out.println(cache);

}

}

## Guava Java Library by Google

<dependency>

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

<version>33.3.0-jre</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>2.0.16</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>2.0.16</version>

</dependency>

## Cache Builder

package com.cache;

import com.google.common.cache.\*;

import java.util.Random;

import java.util.concurrent.TimeUnit;

public class GauvaCacheExample {

public static void main(String[] args) {

LoadingCache<String, String> cache = CacheBuilder.newBuilder()

.maximumSize(200000)

.expireAfterWrite(10, TimeUnit.MINUTES)

.build(new CacheLoader<String, String>() {

@Override

public String load(String key) throws Exception {

return "Value for " + key;

}

});

try{

String[] genres= {"Action", "Comedy", "Drama", "Horror", "Romance", "Sci-Fi", "Thriller"};

//Write code to generate 1000 movies with random genres and push in popularMoviewCache

for(int i=0; i<200000; i++){

String movie = "Movie" + i;

String genre = genres[new Random().nextInt(genres.length)];

cache.put(movie, genre);

}

//Thread.sleep(1000);

for(int i=0;i<100;i++){

long startTime = System.nanoTime();

String movie = cache.get("Movie92600");

long endTime = System.nanoTime();

System.out.println("Time taken to fetch the movie: " + (endTime - startTime) + " nanoseconds");

}

// It would be retrieved from cache

}catch(Exception e){

e.printStackTrace();

}

}

}

DAY 7: 24/09/2024

# Concurrent Hashmap

A ConcurrentHashMap is a thread-safe variant of HashMap in Java that allows multiple threads to read and write data concurrently without the need for external synchronisation. It is part of the java.util.concurrent package, which provides various thread-safe collections designed to handle concurrency effectively.

Use Cases:

* Multi-threaded Application
* High-concurrency

Program:

package com.cache.booksystem;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.ConcurrentMap;

import java.util.\*;

public class Book LibraryCache {

private final Map<String,Book> bookDatabase=new HashMap<>();

//Our cache

private final Map<String,Book> cache=new HashMap<>();

private final int CACHE\_SIZE=5;

private int cacheHits=0;

private int cacheMisses=0;

public Book Library Cache(){

bookDatabase.put("1",new Book("1","Book1","Author1"));

bookDatabase.put("2",new Book("2","Book2","Author2"));

bookDatabase.put("3",new Book("3","Book3","Author3"));

bookDatabase.put("4",new Book("4","Book4","Author4"));

bookDatabase.put("5",new Book("5","Book5","Author5"));

bookDatabase.put("6",new Book("6","Book6","Author6"));

}

public Book getBook(String bookId){

Book book=cache.get(bookId);

if(book!=null){

cacheHits++;

System.out.println("Cache Hit for bookId: "+bookId);

return book;

}

else{

cacheMisses++;

System.out.println("Cache Miss for bookId: "+bookId);

book=bookDatabase.get(bookId);

if(book!=null){

cache.put(bookId,book);

}

return book;

}

}

private void addToCache(String bookId,Book book){

if(cache.size()>CACHE\_SIZE){

String keyToRemove=cache.keySet().iterator().next();

cache.remove(keyToRemove);

System.out.println("Cache is full. Removing least recently used bookId: "+keyToRemove);

}

cache.put(bookId,book);

System.out.println("Added to cache: "+bookId);

}

public void printCacheStatistics(){

System.out.println("Cache Hits: "+cacheHits);

System.out.println("Cache Misses: "+cacheMisses);

System.out.println("Cache Hit Ratio: "+((double)cacheHits/(cacheHits+cacheMisses)));

System.out.println("Current Cache Size: "+cache.size());

System.out.println("Books in Cache: "+cache.keySet());

}

private static class Book{

private String id;

private String title;

private String author;

public Book(String id,String title,String author){

this.id=id;

this.title=title;

this.author=author;

}

@Override

public String toString(){

return "Book{id='"+id+"', title='"+title+"', author='"+author+"'}";

}

}

public static void main(String[] args){

BookLibrary Cache cache=new Book LibraryCache();

String []requestedBooks={"1","2","3","4","5","6","1","2","3","4","5","6","5","4","3","9"};

for(String bookId:requestedBooks){

Book book=cache.getBook(bookId);

if(book!=null){

System.out.println("retrieved book: "+book);

}

else{

System.out.println("Book not found : "+book);

}

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

}

cache.printCacheStatistics();

}

}

# Transient Keyword

The transient keyword in Java is used to tell the system not to save a specific field of an object when that object is converted into a stream of data (serialised).

Example: Imagine you have a user object with a username and password. If you mark the password field as transient, when you save (serialise) the user object, the password won't be saved. Later, when you load (deserialize) the object, the password will be empty or set to a default value, like null.

### **Why Use transient?**

* **Skip Sensitive Data**: If a field contains sensitive information (like a password), marking it as transient ensures that it won’t be saved when the object is serialised.
* **Ignore Unnecessary Data**: If a field's data is temporary or can be easily recreated, you can mark it as transient so it doesn't get stored.

# Executor Service

The ExecutorService in Java is part of the java.util.concurrent package and provides a high-level API for managing and controlling a pool of threads. Instead of manually creating and managing threads, ExecutorService allows to submit tasks that are executed by a pool of threads. This makes it easier to handle multiple tasks concurrently, efficiently utilising system resources.

package com.cache.booksystem;

import java.util.\*;

import java.io.\*;

import java.util.concurrent.\*;

public class Book Library System {

public static void main(String[] args) {

BookLibrary bookLibrary=new BookLibrary();

bookLibrary.addBook(new Book("123","Harry Potter","J.K.Rowling"));

bookLibrary.addBook(new Book("456","To Kill a Mockingbird","Harper Lee"));

bookLibrary.addBook(new Book("789","1984","George Orwell"));

bookLibrary.addBook(new Book("101","Pride and Prejudice","Jane Austen"));

bookLibrary.addBook(new Book("102","The Great Gatsby","F. Scott Fitzgerald"));

bookLibrary.addBook(new Book("103","Moby Dick","Herman Melville"));

bookLibrary.addBook(new Book("104","War and Peace","Leo Tolstoy"));

bookLibrary.addBook(new Book("105","Hamlet","William Shakespeare"));

bookLibrary.addBook(new Book("106","Macbeth","William Shakespeare"));

System.out.println(bookLibrary.getAllBooks());

System.out.println(bookLibrary.getAuthors());

System.out.println(bookLibrary.getBookCountByAuthor());

//How Future is used in Asynchronous programming

//ExecutorService, Callable and Future

//Explain the difference between Future and ExecutorService

//Explain the difference between Future and Thread

//Explain the difference between Future and Runnable

Future<Book> futureMostPopularBook=bookLibrary.getMostPopularBookAsync();

try{

Book mostPopularBook=futureMostPopularBook.get();

System.out.println("Most popular book: "+mostPopularBook.toString());

}catch(InterruptedException | ExecutionException e){

e.printStackTrace();

}finally{

bookLibrary.executorService.shutdown();

}

System.out.println("Program continues while waiting for the most popular book");

}

static class BookLibrary{

private List<Book> books = new ArrayList<>();

private Set<String> authors = new HashSet<>();

private Map<String,Integer> bookCountByAuthor = new HashMap<>();

private ExecutorService executorService = Executors.newSingleThreadExecutor();

BookLibrary(){

books=new ArrayList<>();

authors=new HashSet<>();

bookCountByAuthor=new HashMap<>();

executorService=Executors.newSingleThreadExecutor();

}

public void addBook(Book book){

books.add(book);

authors.add(book.author);

bookCountByAuthor.put(book.author,bookCountByAuthor.getOrDefault(book.author,0)+1);

}

public List<Book> getAllBooks(){

return new ArrayList<>(books);

}

public Set<String> getAuthors(){

return new HashSet<>(authors);

}

public Map<String,Integer> getBookCountByAuthor(){

return new HashMap<>(bookCountByAuthor);

}

//Asynchronous programming

public Future<Book> getMostPopularBookAsync(){

return executorService.submit(()->{

Thread.sleep(2000);

return books.isEmpty()?null:getMostPopularBook();

});

}

private Book getMostPopularBook(){

return books.get(0);

}

}

static class Book implements Serializable{

private String isbn;

private String title;

private String author;

public Book(String isbn,String title,String author){

this.isbn=isbn;

this.title=title;

this.author=author;

}

// transient field, it would not be serialised

transient int currentPage=0;

@Override

public String toString(){

return "Book{"+"isbn='"+isbn'\''+", title='"+title+'\''+", author='"+author+'\''+'}';

}

}

}

# Group Activity: Banking System Application Project

Github link: <https://github.com/Anushka-Bajpai/Banking-System-Project>

DAY 8 : 25/09/2024

Data Structures and Algorithms (DSA)

### What is Data Structure?

A data structure is a way of storing and organising data in our devices to use the data efficiently and effectively. The main idea behind data structures is to minimise the time and space complexities. An efficient data structure takes minimum memory space and requires minimum data execution time.

### What is an algorithm?

The algorithm is defined as a process or set of well-defined instructions that are typically used to solve a particular group of problems or perform a specific type of calculation. To explain it in simpler terms, it is a set of operations performed in a step-by-step manner to execute a task.

### Following are the examples of DSA:

### Arrays, Strings, Tree, Graph, Stacks, Queues, Linkedlist, Hashmap, etc.

# Sorting Techniques

## Selection Sort:

The algorithm repeatedly selects the smallest(or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

Selection Code Program:

import java.util.\*;

public class SelectSort {

static void selection\_sort(int arr[], int n) {

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

int mini = i;

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

if (arr[j] < arr[mini]) {

mini = j;

}

}

//swap

int temp = arr[mini];

arr[mini] = arr[i];

arr[i] = temp;

}

System.out.println("After selection sort:");

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

System.out.print(arr[i] + " ");

}

System.out.println();

}

public static void main(String args[]) {

int arr[] = {13, 46, 24, 52, 20, 9};

int n = arr.length;

System.out.println("Before selection sort:");

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

System.out.print(arr[i] + " ");

}

System.out.println();

selection\_sort(arr, n);

}

}

Output:

Before selection sort:

13 46 24 52 20 9

After selection sort:

9 13 20 24 46 52

Time Complexity:

Best Case Complexity: O(N^2)

Average Case Complexity: O(N^2)

Worst Case Complexity: O(N^2)

## Bubble Sort:

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through a list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

How Bubble Sort Works**:**

1. Compare Adjacent Elements: Starting from the first element, compare it with the next one.
2. Swap if Necessary: If the first element is greater than the second, swap them.
3. Move to the Next Pair: Move to the next pair of elements and repeat the comparison and swap if necessary.
4. Repeat: The process continues for each element in the list.
5. Next Pass: After each complete pass through the list, the largest element among the unsorted elements "bubbles up" to its correct position. The algorithm then repeats the process for the remaining unsorted elements.
6. Stop Condition: The sorting is complete when no swaps are needed in a pass, indicating that the list is sorted.

Bubble Code Program:

import java.util.\*;

public class sort {

static void bubble\_sort(int[] arr, int n) {

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

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

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

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

System.out.println("After bubble sort: ");

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

System.out.print(arr[i] + " ");

}

System.out.println();

}

public static void main(String args[]) {

int arr[] = {13, 46, 24, 52, 20, 9};

int n = arr.length;

System.out.println("Before Using Bubble Sort: ");

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

System.out.print(arr[i] + " ");

}

System.out.println();

bubble\_sort(arr, n);

}

}

### Output:

Before Using Bubble Sort:

13 46 24 52 20 9

After bubble sort:

9 13 20 24 46 52

Time Complexity:

Best Case Complexity: O(N)

Average Case Complexity: O(N^2)

Worst Case Complexity: O(N^2)

## 3. Insertion Sort

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a stable sorting algorithm, meaning that elements with equal values maintain their relative order in the sorted output.

Following are the steps for the insertion sorting:

* We start with the second element of the array as the first element in the array is assumed to be sorted.
* Compare the second element with the first element and check if the second element is smaller then swap them.
* Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
* Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.
* Repeat until the entire array is sorted.

Selection Sort Program :

import java.util.\*;

public class Main {

static void insertion\_sort(int[] arr, int n) {

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

int j = i;

while (j > 0 && arr[j - 1] > arr[j]) {

int temp = arr[j - 1];

arr[j - 1] = arr[j];

arr[j] = temp;

j--;

}

}

System.out.println("After insertion sort: ");

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

System.out.print(arr[i] + " ");

}

System.out.println();

}

public static void main(String args[]) {

int arr[] = {13, 46, 24, 52, 20, 9};

int n = arr.length;

System.out.println("Before Using insertion Sort: ");

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

System.out.print(arr[i] + " ");

}

System.out.println();

insertion\_sort(arr, n);

}

}

Output:

Before Using insertion Sort:

13 46 24 52 20 9

After insertion sort:

9 13 20 24 46 52

### Time Complexity:

Best Case Complexity: O(N)

Average Case Complexity: O(N^2)

Worst Case Complexity: O(N^2)

Day 9: 26/09/2024

## Arrays Data Structure

An array is a collection of items of the same variable type that are stored at contiguous memory locations. It’s one of the most popular and simple data structures and is often used to implement other data structures. Each item in an array is indexed starting with 0 . Each element in an array is accessed through its index.

Need of Array Data Structure:

Arrays are a fundamental data structure in computer science. They are used in a wide variety of applications, including:

* Storing data for processing
* Implementing data structures such as stacks and queues
* Representing data in tables and matrices
* Creating dynamic data structures such as linked lists and trees

Types of Arrays:

1. 1-Dimensional Array: These arrays store a single row of elements.
2. Multi-Dimensional Array: These arrays store multiple rows of elements.

Arrays Operations:

1. Insertion
2. Deletion
3. Traversal
4. Searching

## Structured Query Language (SQL)

Structured query language (SQL) is a programming language for storing and processing information in a relational database. A relational database stores information in tabular form, with rows and columns representing different data attributes and the various relationships between the data values. We can use SQL statements to store, update, remove, search, and retrieve information from the database. We can also use SQL to maintain and optimise database performance.

It is used to perform CRUD operations :

* Create
* Rename
* Update
* Delete

### SQL Commands:

There are following types of SQL commands:

1. Data Definition Language : Create, Alter, Rename, Truncate & Drop.
2. Data Manipulation Language : Select, Insert, Update, Delete.
3. Data Query Language: Select
4. Data Control Language : Grant & Revoke permission to users
5. Transaction Control Language: Start transaction, commit, rollback etc.

#### DAY 11: 30/09/2024

# BINARY SEARCH

Binary search is a search algorithm used to find the position of a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty. The search interval is halved by comparing the target element with the middle value of the search space.To apply binary search, array must be sorted.

## Binary Search using Iterative Approach:

public class BinarySearch {

// Iterative binary search method

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

int low = 0;

int high = arr.length - 1;

// Repeat until the pointers low and high meet

while (low <= high) {

// Find the middle index

int mid = low + (high - low) / 2;

// Check if target is present at mid

if (arr[mid] == target) {

return mid; // Target found, return the index

}

// If target is greater, ignore the left half

if (arr[mid] < target) {

low = mid + 1;

}

// If target is smaller, ignore the right half

else {

high = mid - 1;

}

}

// Target not found

return -1;

}

public static void main(String[] args) {

int[] arr = {2, 3, 4, 10, 40};

int target = 10;

int result = binarySearch(arr, target);

if (result == -1) {

System.out.println("Element not present");

} else {

System.out.println("Element found at index " + result);

}

}

}

OUTPUT:

Element found at index 3

## Binary Search using Recursive Approach:

public class BinarySearchRecursive {

// Recursive binary search method

public static int binarySearch(int[] arr, int low, int high, int target) {

if (low <= high) {

// Find the middle index

int mid = low + (high - low) / 2;

// Check if target is present at mid

if (arr[mid] == target) {

return mid;

}

// If target is smaller, search the left half

if (arr[mid] > target) {

return binarySearch(arr, low, mid - 1, target);

}

// If target is larger, search the right half

return binarySearch(arr, mid + 1, high, target);

}

// Target not found

return -1;

}

public static void main(String[] args) {

int[] arr = {2, 3, 4, 10, 40};

int target = 10;

int n = arr.length;

int result = binarySearch(arr, 0, n - 1, target);

if (result == -1) {

System.out.println("Element not present");

} else {

System.out.println("Element found at index " + result);

}

}

}

OUTPUT:

Element found at index 3

## TRIE Data Structure

* The Trie data structure is a tree-like data structure used for storing a dynamic set of strings. It is commonly used for efficient retrieval and storage of keys in a large dataset. The structure supports operations such as insertion, search, and deletion of keys, making it a valuable tool in fields like computer science and information retrieval. In this article we are going to explore insertion and search operations in Trie Data Structure.
* A Trie is an advanced data structure that is sometimes also known as a prefix tree.
* The longest common prefix, pattern searching, autocomplete and implementation of the dictionary are some of the common applications of a Trie Data Structure.

### **Basic Operations on Trie Data Structure:**

1. Insertion : This operation is used to insert new strings into the Trie data structure.
2. Search : This operation is used to search whether a string is present in the Trie data structure or not
3. Deletion : This operation is used to delete strings from the Trie data structure.

Trie Implementation:

import java.util.\*;

// TrieNode class represents a single node in the Trie

class TrieNode<T> {

// Map to store child nodes, where the key is the character and value is the child TrieNode

Map<Character, TrieNode<T>> children;

// Value associated with this node (null if this node doesn't represent the end of a word)

T value;

// Flag to indicate if this node represents the end of a word

boolean isEndOfWord;

// Constructor to initialize a new TrieNode

public TrieNode() {

this.children = new HashMap<>();

this.value = null;

this.isEndOfWord = false;

}

}

// Trie class implements the main functionality of the Trie data structure

class Trie<T> {

// Root node of the Trie

private TrieNode<T> root;

// Constructor to initialize an empty Trie

public Trie() {

this.root = new TrieNode<>();

}

// Method to insert a word and its associated value into the Trie

public void insert(String word, T value) {

TrieNode<T> current = root;

// Iterate through each character in the word

for (char c : word.toCharArray()) {

// If the current character doesn't exist as a child, create a new node

current = current.children.computeIfAbsent(c, k -> new TrieNode<>());

}

// Mark the last node as the end of a word and set its value

current.isEndOfWord = true;

current.value = value;

}

// Method to search for a word in the Trie

public T search(String word) {

TrieNode<T> node = findNode(word);

// Return the value if the word exists and is marked as end of word, otherwise null

return (node != null && node.isEndOfWord) ? node.value : null;

}

// Method to check if any word in the Trie starts with the given prefix

public boolean startsWith(String prefix) {

// If we can find a node for this prefix, it exists in the Trie

return findNode(prefix) != null;

}

// Helper method to find a node corresponding to a given word or prefix

private TrieNode<T> findNode(String str) {

TrieNode<T> current = root;

// Traverse the Trie following the characters in the string

for (char c : str.toCharArray()) {

current = current.children.get(c);

// If at any point we can't find a child node, the word/prefix doesn't exist

if (current == null) return null;

}

// Return the final node we reached

return current;

}

// Method to get all words in the Trie with a given prefix

public List<String> getWordsWithPrefix(String prefix) {

List<String> result = new ArrayList<>();

TrieNode<T> prefixNode = findNode(prefix);

// If the prefix exists in the Trie, collect all words starting from its last node

if (prefixNode != null) {

collectWords(prefixNode, prefix, result);

}

return result;

}

// Recursive helper method to collect all words from a given node

private void collectWords(TrieNode<T> node, String currentPrefix, List<String> result) {

// If this node is marked as end of word, add the current prefix to results

if (node.isEndOfWord) {

result.add(currentPrefix);

}

// Recursively explore all child nodes

for (Map.Entry<Character, TrieNode<T>> entry : node.children.entrySet()) {

collectWords(entry.getValue(), currentPrefix + entry.getKey(), result);

}

}

}

// Main class to demonstrate the usage of the Trie

public class TrieDemo {

public static void main(String[] args) {

Trie<Integer> trie = new Trie<>();

// Insert some words with associated values

trie.insert("apple", 1);

trie.insert("app", 2);

trie.insert("application", 3);

trie.insert("banana", 4);

// Demonstrate search functionality

System.out.println("Value for 'apple': " + trie.search("apple")); // Output: 1

System.out.println("Value for 'app': " + trie.search("app")); // Output: 2

System.out.println("Value for 'appl': " + trie.search("appl")); // Output: null

// Demonstrate prefix checking

System.out.println("Starts with 'app': " + trie.startsWith("app")); // Output: true

System.out.println("Starts with 'ban': " + trie.startsWith("ban")); // Output: true

System.out.println("Starts with 'car': " + trie.startsWith("car")); // Output: false

// Demonstrate getting words with a prefix

System.out.println("Words with prefix 'app': " + trie.getWordsWithPrefix("app"));

// Output: [app, apple, application]

}

}

OUTPUT:

Value for 'apple': 1

Value for 'app': 2

Value for 'appl': null

Starts with 'app': true

Starts with 'ban': true

Starts with 'car': false

Words with prefix 'app': [app, apple, application]

#### 

#### 

### Day 12: 01/09/2024

# Merge Sort

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves. This process is repeated until the entire array is sorted.

Merge Sort Algorithm Implementation:

import java.util.\*;

class Solution {

private static void merge(int[] arr, int low, int mid, int high) {

ArrayList<Integer> temp = new ArrayList<>(); // temporary array

int left = low; // starting index of left half of arr

int right = mid + 1; // starting index of right half of arr

//storing elements in the temporary array in a sorted manner//

while (left <= mid && right <= high) {

if (arr[left] <= arr[right]) {

temp.add(arr[left]);

left++;

} else {

temp.add(arr[right]);

right++;

}

}

// if elements on the left half are still left //

while (left <= mid) {

temp.add(arr[left]);

left++;

}

// if elements on the right half are still left //

while (right <= high) {

temp.add(arr[right]);

right++;

}

// transfering all elements from temporary to arr //

for (int i = low; i <= high; i++) {

arr[i] = temp.get(i - low);

}

}

public static void mergeSort(int[] arr, int low, int high) {

if (low >= high) return;

int mid = (low + high) / 2 ;

mergeSort(arr, low, mid); // left half

mergeSort(arr, mid + 1, high); // right half

merge(arr, low, mid, high); // merging sorted halves

}

}

public class Sort {

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

int n = 7;

int arr[] = { 9, 4, 7, 6, 3, 1, 5 };

System.out.println("Before sorting array: ");

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

System.out.print(arr[i] + " ");

}

System.out.println();

Solution.mergeSort(arr, 0, n - 1);

System.out.println("After sorting array: ");

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

System.out.print(arr[i] + " ");

}

System.out.println();

}

}

OUTPUT:

Before sorting array:

9 4 7 6 3 1 5

After sorting array:

1 3 4 5 6 7 9

**Time complexity:** O(nlogn)

# Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does the Quicksort Algorithm work?

There are three main steps in the algorithm:

1. Choose a pivot

2. Partition the array around the pivot. After partition, it is ensured that all elements are smaller than all right and we get the index of the endpoint of smaller components. The left and right may not be sorted individually.

3. Recursively call for the two partitioned left and right subarrays.

4. We stop recursion when there is only one element left.

Quick Sort Algorithm Implementation:

import java.util.\*;

class Solution {

static int partition(List<Integer> arr, int low, int high) {

int pivot = arr.get(low);

int i = low;

int j = high;

while (i < j) {

while (arr.get(i) <= pivot && i <= high - 1) {

i++;

}

while (arr.get(j) > pivot && j >= low + 1) {

j--;

}

if (i < j) {

int temp = arr.get(i);

arr.set(i, arr.get(j));

arr.set(j, temp);

}

}

int temp = arr.get(low);

arr.set(low, arr.get(j));

arr.set(j, temp);

return j;

}

static void qs(List<Integer> arr, int low, int high) {

if (low < high) {

int pIndex = partition(arr, low, high);

qs(arr, low, pIndex - 1);

qs(arr, pIndex + 1, high);

}

}

public static List<Integer> quickSort(List<Integer> arr) {

// Write your code here.

qs(arr, 0, arr.size() - 1);

return arr;

}

}

public class Sort {

public static void main(String args[]) {

List<Integer> arr = new ArrayList<>();

arr = Arrays.asList(new Integer[] {4, 6, 2, 5, 7, 9, 1, 3});

int n = arr.size();

System.out.println("Before Using Quick Sort: ");

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

System.out.print(arr.get(i) + " ");

}

System.out.println();

arr = Solution.quickSort(arr);

System.out.println("After Quick sort: ");

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

System.out.print(arr.get(i) + " ");

}

System.out.println();

}

}

OUTPUT:  
Before Using Quick Sort:

4 6 2 5 7 9 1 3

After Quick sort:

1 2 3 4 5 6 7 9

# BINARY TREE

The binary tree is a tree data structure(non-linear) in which each node can have at most two children which are referred to as the left child and the right child.

The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves. A binary tree can be visualised as a hierarchical structure with the root at the top and the leaves at the bottom.

Terminologies in Binary Tree:

* **Nodes:** The fundamental part of a binary tree, where each node contains **data** and links to two child nodes.
* **Root**: The topmost node in a tree is known as the root node. It has no parent and serves as the starting point for all nodes in the tree.
* **Parent Node**: A node that has one or more child nodes. In a binary tree, each node can have at most two children.
* **Child Node**: A node that is a descendant of another node (its parent).
* **Leaf Node**: A node that does not have any children or both children are null.
* **Internal Node**: A node that has at least one child. This includes all nodes except the rootand the leaf nodes.
* **Depth of a Node**: The number of edges from a specific node to the root node. The depth of the root node is zero.
* **Height of a Binary Tree**: The number of nodes from the deepest leaf node to the root node.

public class BinaryTree {

// Node class representing each node in the binary tree

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

// Root of Binary Tree

Node root;

// Constructor

BinaryTree() {

root = null;

}

// Method to insert a new node

void insert(int key) {

root = insertRec(root, key);

}

// A recursive function to insert a new key in BST

Node insertRec(Node root, int key) {

// If the tree is empty, return a new node

if (root == null) {

root = new Node(key);

return root;

}

// Otherwise, recur down the tree

if (key < root.data)

root.left = insertRec(root.left, key);

else if (key > root.data)

root.right = insertRec(root.right, key);

// Return the (unchanged) node pointer

return root;

}

// Method to do inorder traversal of tree

void inorder() {

inorderRec(root);

}

// A utility function to do inorder traversal of BST

void inorderRec(Node root) {

if (root != null) {

inorderRec(root.left);

System.out.print(root.data + " ");

inorderRec(root.right);

}

}

// Method to search a key in BST

boolean search(int key) {

return searchRec(root, key);

}

// A recursive function to search a given key in BST

boolean searchRec(Node root, int key) {

// Base Cases: root is null or key is present at root

if (root == null || root.data == key)

return root != null;

// Key is greater than root's key

if (root.data < key)

return searchRec(root.right, key);

// Key is smaller than root's key

return searchRec(root.left, key);

}

// Depth First Search (DFS) implementation

void depthFirstSearch() {

System.out.println("Depth First Search (Pre-order traversal):");

dfsRec(root);

System.out.println();

}

// A utility function to do DFS of BST

void dfsRec(Node node) {

if (node == null)

return;

// First print data of node

System.out.print(node.data + " ");

// Then recur on left subtree

dfsRec(node.left);

// Now recur on right subtree

dfsRec(node.right);

}

// Driver Code

public static void main(String[] args) {

BinaryTree tree = new BinaryTree();

/\* Let's create following BST

50

/ \

30 70

/ \ / \

20 40 60 80 \*/

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

// Print inorder traversal of the BST

System.out.println("Inorder traversal of the binary tree:");

tree.inorder();

System.out.println();

// Search for a key

int key = 60;

System.out.println("\nSearching for key " + key + ": " +

(tree.search(key) ? "Found" : "Not Found"));

// Perform DFS

tree.depthFirstSearch();

}

}

OUTPUT:  
Inorder traversal of the binary tree:

20 30 40 50 60 70 80

Searching for key 60: Found

#### Depth First Search (Pre-order traversal):

50 30 20 40 70 60 80

#### Day 12: 03/10/24

## Dynamic Programming:

Dynamic Programming is a method used in mathematics and computer science to solve complex problems by breaking them down into simpler subproblems. By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

### Approaches of Dynamic Programming:

1. Top-Down Approach (Memoization) : In the top-down approach, also known as memoization, we start with the final solution and recursively break it down into smaller subproblems. To avoid redundant calculations, we store the results of solved subproblems in a memoization table.
2. Bottom-Up Approach (Tabulation) : In the bottom-up approach, also known as tabulation, we start with the smallest subproblems and gradually build up to the final solution. We store the results of solved subproblems in a table to avoid redundant calculations.

Problems on Dynamic Programming (DP):

1. Coin Change Problem

DAY 16 : 07/10/24

## JOINS

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Different Types of SQL JOINs:

1. Inner Join
2. Left Join
3. Right Join
4. Outer Join

1. INNER JOIN : The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

Syntax:

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

INNER JOIN table2

ON table1.matching\_column = table2.matching\_column;

2. LEFT JOIN : LEFT JOIN returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain null. LEFT JOIN is also known as LEFT OUTER JOIN.

Syntax:

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

LEFT JOIN table2

ON table1.matching\_column = table2.matching\_column;

3. RIGHT JOIN : RIGHT JOIN returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join.It is very similar to LEFT JOIN For the rows for which there is no matching row on the left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

RIGHT JOIN table2

ON table1.matching\_column = table2.matching\_column;

4. FULL JOIN : In SQL, the FULL JOIN (or FULL OUTER JOIN) is a powerful technique used to combine records from two or more tables. Unlike an INNER JOIN, which only returns rows where there are matches in both tables, a FULL JOIN retrieves all rows from both tables, filling in NULL values where matches do not exist.

Syntax:

SELECT columns

FROM table1

FULL JOIN table2

ON table1.column = table2.column;

*QUERY 1.* CREATE A TABLE TO STORE EMPLOYEES RECORD.

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

manager\_id INT,

FOREIGN KEY (manager\_id) REFERENCES employees(employee\_id)

);

INSERT INTO employees (employee\_id, first\_name, last\_name, manager\_id) VALUES

(1, 'John', 'Doe', NULL),

(2, 'Jane', 'Smith', 1),

(3, 'Bob', 'Johnson', 1),

(4, 'Alice', 'Williams', 2),

(5, 'Charlie', 'Brown', 2),

(6, 'David', 'Lee', 3);

*QUERIES:*

**-- Create tables**

CREATE TABLE departments (

department\_id INT PRIMARY KEY,

department\_name VARCHAR(50),

location\_id INT

);

CREATE TABLE locations (

location\_id INT PRIMARY KEY,

city VARCHAR(50)

);

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

email VARCHAR(100),

phone\_number VARCHAR(20),

hire\_date DATE,

job\_id VARCHAR(10),

salary DECIMAL(10, 2),

department\_id INT,

FOREIGN KEY (department\_id) REFERENCES departments(department\_id)

);

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

customer\_name VARCHAR(100),

email VARCHAR(100)

);

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

customer\_id INT,

order\_date DATE,

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

category VARCHAR(50),

price DECIMAL(10, 2)

);

CREATE TABLE discontinued\_products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

category VARCHAR(50)

);

**-- Insert sample data**

INSERT INTO departments (department\_id, department\_name, location\_id) VALUES

(10, 'Administration', 1),

(20, 'Marketing', 2),

(30, 'Purchasing', 1),

(40, 'Human Resources', 2),

(50, 'Shipping', 3);

INSERT INTO locations (location\_id, city) VALUES

(1, 'New York'),

(2, 'Los Angeles'),

(3, 'Chicago');

INSERT INTO employees (employee\_id, first\_name, last\_name, email, phone\_number, hire\_date, job\_id, salary, department\_id) VALUES

(1, 'John', 'Doe', 'john.doe@example.com', '515.123.4567', '2019-06-17', 'AD\_PRES', 24000.00, 10),

(2, 'Jane', 'Smith', 'jane.smith@example.com', '515.123.4568', '2020-02-20', 'AD\_VP', 17000.00, 10),

(3, 'Alice', 'Johnson', 'alice.johnson@example.com', '515.123.4569', '2020-08-11', 'MK\_MAN', 9000.00, 20),

(4, 'Bob', 'Brown', 'bob.brown@example.com', '515.123.4560', '2021-03-05', 'HR\_REP', 6000.00, 40),

(5, 'Charlie', 'Davis', 'charlie.davis@example.com', '515.123.4561', '2021-11-30', 'SH\_CLERK', 3000.00, 50);

INSERT INTO customers (customer\_id, customer\_name, email) VALUES

(1, 'Acme Corp', 'contact@acmecorp.com'),

(2, 'GlobalTech', 'info@globaltech.com'),

(3, 'Local Shop', 'owner@localshop.com');

INSERT INTO orders (order\_id, customer\_id, order\_date) VALUES

(1, 1, '2023-01-15'),

(2, 1, '2023-02-20'),

(3, 2, '2023-02-22');

INSERT INTO products (product\_id, product\_name, category, price) VALUES

(1, 'Laptop', 'Electronics', 999.99),

(2, 'Smartphone', 'Electronics', 699.99),

(3, 'Desk Chair', 'Furniture', 199.99);

INSERT INTO discontinued\_products (product\_id, product\_name, category) VALUES

(101, 'Old Laptop Model', 'Electronics'),

(102, 'Discontinued Phone', 'Electronics');-- Create tables

CREATE TABLE departments (

department\_id INT PRIMARY KEY,

department\_name VARCHAR(50),

location\_id INT

);

CREATE TABLE locations (

location\_id INT PRIMARY KEY,

city VARCHAR(50)

);

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

email VARCHAR(100),

phone\_number VARCHAR(20),

hire\_date DATE,

job\_id VARCHAR(10),

salary DECIMAL(10, 2),

department\_id INT,

FOREIGN KEY (department\_id) REFERENCES departments(department\_id)

);

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

customer\_name VARCHAR(100),

email VARCHAR(100)

);

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

customer\_id INT,

order\_date DATE,

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

category VARCHAR(50),

price DECIMAL(10, 2)

);

CREATE TABLE discontinued\_products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

category VARCHAR(50)

);

**-- Insert sample data**

INSERT INTO departments (department\_id, department\_name, location\_id) VALUES

(10, 'Administration', 1),

(20, 'Marketing', 2),

(30, 'Purchasing', 1),

(40, 'Human Resources', 2),

(50, 'Shipping', 3);

INSERT INTO locations (location\_id, city) VALUES

(1, 'New York'),

(2, 'Los Angeles'),

(3, 'Chicago');

INSERT INTO employees (employee\_id, first\_name, last\_name, email, phone\_number, hire\_date, job\_id, salary, department\_id) VALUES

(1, 'John', 'Doe', 'john.doe@example.com', '515.123.4567', '2019-06-17', 'AD\_PRES', 24000.00, 10),

(2, 'Jane', 'Smith', 'jane.smith@example.com', '515.123.4568', '2020-02-20', 'AD\_VP', 17000.00, 10),

(3, 'Alice', 'Johnson', 'alice.johnson@example.com', '515.123.4569', '2020-08-11', 'MK\_MAN', 9000.00, 20),

(4, 'Bob', 'Brown', 'bob.brown@example.com', '515.123.4560', '2021-03-05', 'HR\_REP', 6000.00, 40),

(5, 'Charlie', 'Davis', 'charlie.davis@example.com', '515.123.4561', '2021-11-30', 'SH\_CLERK', 3000.00, 50);

INSERT INTO customers (customer\_id, customer\_name, email) VALUES

(1, 'Acme Corp', 'contact@acmecorp.com'),

(2, 'GlobalTech', 'info@globaltech.com'),

(3, 'Local Shop', 'owner@localshop.com');

INSERT INTO orders (order\_id, customer\_id, order\_date) VALUES

(1, 1, '2023-01-15'),

(2, 1, '2023-02-20'),

(3, 2, '2023-02-22');

INSERT INTO products (product\_id, product\_name, category, price) VALUES

(1, 'Laptop', 'Electronics', 999.99),

(2, 'Smartphone', 'Electronics', 699.99),

(3, 'Desk Chair', 'Furniture', 199.99);

INSERT INTO discontinued\_products (product\_id, product\_name, category) VALUES

(101, 'Old Laptop Model', 'Electronics'),

(102, 'Discontinued Phone', 'Electronics');

DAY 17 : 08/10/24

*QUERY 2.* write an sql query to get the average salary from your dep

select

d.department\_name,

e.first\_name,

e.last\_name,

e.salary,

(select avg(salary) from employees where department\_id = e.department\_id) as dept\_avg\_salary

from

employees e

inner join

departments d

on e.department\_id =d.department\_id

where e.salary > (select avg(salary) from employees )

order by d.department\_name , e.salary desc

Example 1: Flight Connections

CREATE TABLE airports (

airport\_code CHAR(3) PRIMARY KEY,

airport\_name VARCHAR(100),

city VARCHAR(50),

country VARCHAR(50)

);

CREATE TABLE flights (

flight\_id INT PRIMARY KEY,

departure\_airport CHAR(3),

arrival\_airport CHAR(3),

departure\_time TIME,

arrival\_time TIME,

FOREIGN KEY (departure\_airport) REFERENCES airports(airport\_code),

FOREIGN KEY (arrival\_airport) REFERENCES airports(airport\_code)

);

INSERT INTO airports (airport\_code, airport\_name, city, country) VALUES

('JFK', 'John F. Kennedy International Airport', 'New York', 'USA'),

('LAX', 'Los Angeles International Airport', 'Los Angeles', 'USA'),

('LHR', 'London Heathrow Airport', 'London', 'UK'),

('CDG', 'Charles de Gaulle Airport', 'Paris', 'France'),

('NRT', 'Narita International Airport', 'Tokyo', 'Japan');

INSERT INTO flights (flight\_id, departure\_airport, arrival\_airport, departure\_time, arrival\_time) VALUES

(1, 'JFK', 'LAX', '08:00', '11:00'),

(2, 'LAX', 'NRT', '13:00', '17:00'),

(3, 'NRT', 'LHR', '19:00', '23:00'),

(4, 'LHR', 'CDG', '09:00', '10:30'),

(5, 'CDG', 'JFK', '12:00', '14:00'),

(6, 'JFK', 'LHR', '18:00', '06:00'),

(7, 'LHR', 'LAX', '10:00', '13:00');

-- Find all possible one-stop flights from JFK to NRT

ANSWER QUERY:

select

f1.flight\_id,

a1.city as departure\_city,

a2.city as layover\_city,

a3.city as arrival\_city,

f1.departure\_time,

f1.arrival\_time as layover\_arrival,

f2.departure\_time as layover\_departure,

f2.arrival\_time

from flights f1

join flights f2 on f1.arrival\_airport =f2.departure\_airport

join airports a1 on f1.departure\_airport = a1.airport\_code

join airports a2 on f1.arrival\_airport = a2.airport\_code

join airports a3 on f2.arrival\_airport = a3.airport\_code

where f1.departure\_airport ='JFK'

and f2.arrival\_airport ='NRT'

and f2.departure\_time> f1.arrival\_time;

INSERT INTO employees (employee\_id, first\_name, last\_name, manager\_id) VALUES

(1, 'John', 'Doe', NULL), -- CEO

(2, 'Jane', 'Smith', 1), -- Reports to John

(3, 'Bob', 'Johnson', 1), -- Reports to John

(4, 'Alice', 'Williams', 2), -- Reports to Jane

(5, 'Charlie', 'Brown', 2), -- Reports to Jane

(6, 'David', 'Lee', 3), -- Reports to Bob

(7, 'Eva', 'Garcia', 3), -- Reports to Bob

(8, 'Frank', 'Lopez', 4), -- Reports to Alice

(9, 'Grace', 'Kim', 6), -- Reports to David

(10, 'Henry', 'Chen', 7); -- Reports to Eva

with recursive employee\_hierarchy as (

select employee\_id, first\_name,last\_name , manager\_id,0 as level

from employees

where manager\_id is NULL

union all

select e.employee\_id, e.first\_name,e.last\_name,e.manager\_id, 1

from employees e

join employees eh on eh.manager\_id =eh.employee\_id

)

select

employee\_id,

concat(first\_name, ' ', last\_name) as employee\_name,

level

from employee\_hierarchy

order by level , employee\_id

QUERY EXAMPLE:

-- Sample Data Setup

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

department VARCHAR(50),

salary DECIMAL(10, 2),

hire\_date DATE

);

INSERT INTO employees VALUES

(1, 'John', 'Doe', 'IT', 75000, '2020-01-15'),

(2, 'Jane', 'Smith', 'HR', 65000, '2019-05-11'),

(3, 'Bob', 'Johnson', 'IT', 80000, '2018-03-23'),

(4, 'Alice', 'Williams', 'Finance', 72000, '2021-09-30'),

(5, 'Charlie', 'Brown', 'IT', 68000, '2022-02-14'),

(6, 'Eva', 'Davis', 'HR', 61000, '2020-11-18'),

(7, 'Frank', 'Miller', 'Finance', 79000, '2017-07-12'),

(8, 'Grace', 'Taylor', 'IT', 77000, '2019-04-22'),

(9, 'Henry', 'Anderson', 'Finance', 71000, '2021-01-05'),

(10, 'Ivy', 'Thomas', 'HR', 63000, '2022-06-30');

CREATE TABLE projects (

project\_id INT PRIMARY KEY,

project\_name VARCHAR(100),

start\_date DATE,

end\_date DATE

);

INSERT INTO projects VALUES

(1, 'Database Migration', '2023-01-01', '2023-06-30'),

(2, 'New HR System', '2023-03-15', '2023-12-31'),

(3, 'Financial Reporting Tool', '2023-02-01', '2023-11-30'),

(4, 'IT Infrastructure Upgrade', '2023-05-01', '2024-04-30');

CREATE TABLE employee\_projects (

employee\_id INT,

project\_id INT,

role VARCHAR(50),

FOREIGN KEY (employee\_id) REFERENCES employees(employee\_id),

FOREIGN KEY (project\_id) REFERENCES projects(project\_id)

);

INSERT INTO employee\_projects VALUES

(1, 1, 'Project Lead'),

(2, 2, 'Project Manager'),

(3, 1, 'Database Admin'),

(4, 3, 'Financial Analyst'),

(5, 4, 'Network Engineer'),

(6, 2, 'HR Specialist'),

(7, 3, 'Data Analyst'),

(8, 4, 'Systems Architect'),

(1, 4, 'Security Consultant'),

(3, 4, 'Software Developer');

-- Questions

-- 1. Write a query to find the top 3 highest paid employees in each department.

ANSWER:

3 Highest paid employees:

select department,first\_name,last\_name,salary, rankno

from (

select department, first\_name,last\_name,salary, dense\_rank() over (partition by department order by salary desc) as rankno

from employees

)ranked\_employees

where rankno <=3

order by department , rankno;

select department,first\_name,last\_name,salary,hire\_date,

sum(salary) over( partition by department order by hire\_date rows between unbounded preceding and current row) as running\_total

from employees

order by department, hire\_date

-- 2. Calculate the running total of salaries in each department, ordered by hire date.

-- 3. Find employees who are working on more than one project, along with the count of projects they're involved in.

ANSWER:

select

e.employee\_id,

e.first\_name,

e.last\_name,

count(ep.project\_id) as project\_count

from employees e

join employee\_projects ep on e.employee\_id = ep.employee\_id

group by e.employee\_id,e.first\_name,e.last\_name

having count(ep.project\_id)>1

order by project\_count desc,e.last\_name,e.first\_name

-- 4. Identify projects that have team members from all departments.

ANSWER:

WITH DepartmentCounts AS (

-- Step 1: Count total unique departments

SELECT COUNT(DISTINCT department) AS total\_departments

FROM employees

),

ProjectDepartmentCounts AS (

-- Step 2: For each project, count the unique departments involved

SELECT ep.project\_id, p.project\_name, COUNT(DISTINCT e.department) AS project\_department\_count

FROM employee\_projects ep

JOIN employees e ON ep.employee\_id = e.employee\_id

JOIN projects p ON ep.project\_id = p.project\_id

GROUP BY ep.project\_id, p.project\_name

)

SELECT pdc.project\_name

FROM ProjectDepartmentCounts pdc

JOIN DepartmentCounts dc ON pdc.project\_department\_count = dc.total\_departments;

-- 5. Calculate the average salary for each department, but only include employees hired in the last 3 years.

SELECT

department,

AVG(salary) as avg\_salary

FROM

employees

WHERE

hire\_date > CURRENT\_DATE - INTERVAL '3 years'

GROUP BY

department

ORDER BY

avg\_salary DESC;

-- 6. Create a pivot table showing the count of employees in each department, with columns for different salary ranges (e.g., <65000, 65000-75000, >75000).

-- 7. Find the employee(s) with the highest salary in their respective departments, who are also working on the longest-running project.

-- 8. Calculate the percentage of each department's salary compared to the total salary of the company.

-- 9. Identify employees who have a higher salary than their department's average, and show by what percentage their salary exceeds the average.

-- 10. Create a query that shows a hierarchical view of employees and their projects, with multiple levels of projects if an employee is in more than one.

-- Bonus Challenge:

-- 11. Implement a query to find the "Kevin Bacon Number" equivalent for projects.

-- (i.e., for each pair of employees, find the shortest connection through shared projects)

NOTES:

EXPLAIN output:

1. id:

- Identifies each SELECT in the query

- A sequential number for simple queries

- Can be the same for JOINed tables or subqueries

2. select\_type:

- SIMPLE: Simple SELECT (no subqueries or UNIONs)

- PRIMARY: Outermost SELECT

- SUBQUERY: First SELECT in a subquery

- DERIVED: SELECT in a derived table (subquery in FROM clause)

- UNION: Second or later SELECT in a UNION

- UNION RESULT: Result of a UNION

3. table:

- The table this row refers to

- Can be a derived table name for subqueries

4. partitions:

- Partitions read, if the table is partitioned

5. type:

- Join type, crucial for optimization

- Common values:

- system: Table has only one row

- const: Matching one row, very fast

- eq\_ref: One row read per combination of rows from previous tables

- ref: All matching rows from an index read for each combination

- range: Index used to retrieve rows within a range

- index: Full index scan

- ALL: Full table scan (slowest)

6. possible\_keys:

- Indexes that could be used

- NULL if no index could be used

7. key:

- The index actually chosen

- NULL if no index was used

8. key\_len:

- Length of the chosen key

9. ref:

- Columns or constants used with the key

10. rows:

- Estimated number of rows to examine

11. filtered:

- Estimated percentage of rows filtered by table condition

12. Extra:

- Additional information like "Using index", "Using temporary", "Using filesort"

Example:

Let's say we have this query:

sql

EXPLAIN SELECT e.first\_name, d.department\_name

FROM employees e

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

WHERE e.salary > 50000;

A possible EXPLAIN output might look like:

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

| id | select\_type | table | partitions | type | possible\_keys | key | key\_len | ref | rows | filtered | Extra |

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

| 1 | SIMPLE | e | NULL | ALL | NULL | NULL | NULL | NULL | 1000 | 33.33 | Using where |

| 1 | SIMPLE | d | NULL | eq\_ref | PRIMARY | PRIMARY | 4 | employees.department\_id | 1 | 100.00 | NULL |

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

Interpreting this:

1. It's a SIMPLE query (no subqueries or UNIONs).

2. It's scanning the entire employees table (type: ALL) which is inefficient.

3. It's using an index lookup on departments (type: eq\_ref).

4. It estimates examining 1000 rows from employees.

5. The WHERE condition is filtering about 33.33% of the rows.

6. It's using a WHERE clause on the employees table.

This output suggests that adding an index on the salary column in the employees table could potentially improve the query's performance.

Understanding EXPLAIN output helps in identifying performance bottlenecks and guides index creation and query rewriting for optimization.

1. id:

- Purpose: Identifies each SELECT in the query.

- Example:

sql

EXPLAIN

SELECT \* FROM employees

UNION

SELECT \* FROM retired\_employees;

Output:

+----+-------------+-------------------+...

| id | select\_type | table |...

+----+-------------+-------------------+...

| 1 | PRIMARY | employees |...

| 2 | UNION | retired\_employees |...

| 3 | UNION RESULT | <union1,2> |...

+----+-------------+-------------------+...

- Explanation: Here, 1 and 2 represent the two SELECT statements, while 3 is the result of the UNION.

2. select\_type:

- Purpose: Indicates the type of SELECT statement.

- Examples:

a) SIMPLE:

sql

EXPLAIN SELECT \* FROM employees WHERE salary > 50000;

b) SUBQUERY:

sql

EXPLAIN SELECT \* FROM employees WHERE department\_id IN (SELECT id FROM departments WHERE location = 'New York');

c) DERIVED:

sql

EXPLAIN SELECT \* FROM (SELECT id, name FROM employees WHERE salary > 50000) AS high\_paid\_employees;

3. table:

- Purpose: Shows which table the row refers to.

- Example:

sql

EXPLAIN SELECT e.name, d.name

FROM employees e

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

Output might show 'e' and 'd' in the table column.

4. partitions:

- Purpose: Indicates which partitions are being accessed.

- Example (assuming 'employees' is partitioned by year):

sql

EXPLAIN SELECT \* FROM employees WHERE hire\_date BETWEEN '2020-01-01' AND '2020-12-31';

Might show 'p2020' in the partitions column.

5. type:

- Purpose: Shows the join type, crucial for understanding query efficiency.

- Examples:

a) const:

sql

EXPLAIN SELECT \* FROM employees WHERE id = 1;

b) ref:

sql

EXPLAIN SELECT \* FROM employees WHERE department\_id = 5;

(Assuming department\_id is indexed but not unique)

c) range:

sql

EXPLAIN SELECT \* FROM employees WHERE salary BETWEEN 50000 AND 60000;

(Assuming salary is indexed)

6. possible\_keys:

- Purpose: Lists indexes that could potentially be used.

- Example:

sql

EXPLAIN SELECT \* FROM employees WHERE department\_id = 5 AND salary > 50000;

Might show 'idx\_dept\_id, idx\_salary' if both columns are indexed.

7. key:

- Purpose: Shows the actual index used.

- Example: In the above query, it might show 'idx\_dept\_id' if MySQL decides this is more selective.

8. key\_len:

- Purpose: Indicates how much of the index is being used.

- Example: If department\_id is an INT (4 bytes), key\_len might show '4'.

9. ref:

- Purpose: Shows which columns or constants are compared to the index.

- Example:

sql

EXPLAIN SELECT \* FROM employees e JOIN departments d ON e.department\_id = d.id WHERE d.name = 'IT';

Might show 'const' for the departments table and 'd.id' for the employees table.

10. rows:

- Purpose: Estimates the number of rows examined.

- Example: In a query on a large table, this might show a high number like 10000, indicating potential for optimization.

11. filtered:

- Purpose: Estimates the percentage of rows filtered by table conditions.

- Example: In a query with WHERE salary > 50000, if 30% of employees meet this condition, it might show 30.00.

12. Extra:

- Purpose: Provides additional information about how MySQL executes the query.

- Examples:

a) "Using index" (good):

sql

EXPLAIN SELECT id, name FROM employees WHERE id > 1000;

b) "Using temporary" (potential for optimization):

sql

EXPLAIN SELECT department\_id, AVG(salary) FROM employees GROUP BY department\_id ORDER BY AVG(salary);

c) "Using filesort" (potential for optimization):

sql

EXPLAIN SELECT \* FROM employees ORDER BY last\_name;

(Assuming no index on last\_name)

Understanding these elements helps in query optimization. For instance, seeing 'ALL' in the type column or high numbers in the rows column often indicates areas for improvement, potentially by adding indexes or rewriting the query.

DAY 18 : 09/10/24

1. Primary Keys

A primary key is a column or set of columns in a table that uniquely identifies each row.

Key characteristics:

- Must be unique

- Cannot contain NULL values

- Should be immutable (not change over time)

Example:

Let's create a simple "Students" table:

sql

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

DateOfBirth DATE

);

Here, StudentID is the primary key.

2. Foreign Keys

A foreign key is a column or set of columns in one table that refers to the primary key in another table. It establishes a link between two tables.

Key characteristics:

- Creates a relationship between tables

- Ensures referential integrity

- Can contain NULL values (unless specified otherwise)

Example:

Let's create a "Courses" table and an "Enrollments" table:

sql

CREATE TABLE Courses (

CourseID INT PRIMARY KEY,

CourseName VARCHAR(100)

);

CREATE TABLE Enrollments (

EnrollmentID INT PRIMARY KEY,

StudentID INT,

CourseID INT,

EnrollmentDate DATE,

FOREIGN KEY (StudentID) REFERENCES Students(StudentID),

FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)

);

Here, StudentID and CourseID in the Enrollments table are foreign keys referencing the Students and Courses tables, respectively.

3. Normalization

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. There are several normal forms, but we'll focus on the first three, which are the most commonly used.

First Normal Form (1NF):

- Each column contains atomic (indivisible) values

- No repeating groups

Example:

Consider this unnormalized table:

Students:

StudentID | Name | Courses

1 | John Doe | Math, Physics, Chemistry

2 | Jane Smith | Biology, Chemistry

To bring it to 1NF:

sql

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(100)

);

CREATE TABLE StudentCourses (

StudentID INT,

Course VARCHAR(50),

FOREIGN KEY (StudentID) REFERENCES Students(StudentID)

);

Second Normal Form (2NF):

- Must be in 1NF

- All non-key attributes are fully functionally dependent on the primary key

Example:

Consider this table that's in 1NF but not 2NF:

Orders:

OrderID | ProductID | ProductName | Quantity | CustomerID | CustomerName

To bring it to 2NF:

sql

CREATE TABLE Orders (

OrderID INT,

ProductID INT,

Quantity INT,

CustomerID INT,

PRIMARY KEY (OrderID, ProductID)

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100)

);

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100)

);

Third Normal Form (3NF):

- Must be in 2NF

- No transitive dependencies (non-key columns depend only on the primary key)

Example:

Consider this table that's in 2NF but not 3NF:

Employees:

EmployeeID | Name | DepartmentID | DepartmentName | ManagerID

To bring it to 3NF:

sql

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100),

DepartmentID INT,

ManagerID INT,

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)

);

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(100)

);

These normalised structures reduce data redundancy and improve data integrity. Each piece of information is stored in only one place, making updates and maintenance easier and reducing the risk of data inconsistencies.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EXAMPLE :

Let's start with an unnormalized table for a small bookstore:

BookOrders:

OrderID | CustomerName | CustomerEmail | BookTitle | Author | Price | OrderDate | ShippingAddress

Step 1: First Normal Form (1NF)

To achieve 1NF, we need to ensure that:

1. Each column contains atomic values

2. There are no repeating groups

Our table is almost in 1NF, but let's assume that sometimes multiple books are ordered together. We'll split this into two tables:

sql

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerName VARCHAR(100),

CustomerEmail VARCHAR(100),

OrderDate DATE,

ShippingAddress VARCHAR(200)

);

CREATE TABLE OrderDetails (

OrderDetailID INT PRIMARY KEY,

OrderID INT,

BookTitle VARCHAR(200),

Author VARCHAR(100),

Price DECIMAL(10, 2),

FOREIGN KEY (OrderID) REFERENCES Orders(OrderID)

);

Now we're in 1NF. Each column contains atomic values, and we've eliminated the repeating group (multiple books per order).

Step 2: Second Normal Form (2NF)

To achieve 2NF:

1. The table must be in 1NF

2. All non-key attributes must be fully functionally dependent on the primary key

Our Orders table is already in 2NF because all attributes depend on the primary key OrderID.

However, in the OrderDetails table, BookTitle, Author, and Price don't depend on the full primary key (OrderDetailID), but only on part of it (BookTitle). Let's fix this:

sql

CREATE TABLE Books (

BookID INT PRIMARY KEY,

BookTitle VARCHAR(200),

Author VARCHAR(100),

Price DECIMAL(10, 2)

);

CREATE TABLE OrderDetails (

OrderDetailID INT PRIMARY KEY,

OrderID INT,

BookID INT,

Quantity INT,

FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),

FOREIGN KEY (BookID) REFERENCES Books(BookID)

);

Now we're in 2NF. All non-key attributes in each table fully depend on their respective primary keys.

Step 3: Third Normal Form (3NF)

To achieve 3NF:

1. The table must be in 2NF

2. There should be no transitive dependencies (non-key columns should depend only on the primary key)

In our current structure, we have a transitive dependency in the Orders table: CustomerEmail depends on CustomerName, which depends on OrderID. Let's resolve this:

sql

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100),

CustomerEmail VARCHAR(100)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

ShippingAddress VARCHAR(200),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

-- Books and OrderDetails tables remain the same

Now we're in 3NF. All non-key attributes in each table depend only on the primary key, and we've eliminated transitive dependencies.

Final 3NF Structure:

sql

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100),

CustomerEmail VARCHAR(100)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

ShippingAddress VARCHAR(200),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Books (

BookID INT PRIMARY KEY,

BookTitle VARCHAR(200),

Author VARCHAR(100),

Price DECIMAL(10, 2)

);

CREATE TABLE OrderDetails (

OrderDetailID INT PRIMARY KEY,

OrderID INT,

BookID INT,

Quantity INT,

FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),

FOREIGN KEY (BookID) REFERENCES Books(BookID)

);

This structure in 3NF offers several benefits:

1. Reduced data redundancy: Customer and book information is stored only once.

2. Improved data integrity: Updating customer or book information only needs to happen in one place.

3. Easier data maintenance: Adding or modifying data is simpler and less error-prone.

4. Flexible querying: It's easier to query and analyze data across different aspects of the business.

Here's a summary of the changes made at each step:

1NF: Split the single table into Orders and OrderDetails to handle multiple books per order.

2NF: Extracted book information into a separate Books table to remove partial dependencies.

3NF: Created a separate Customers table to eliminate transitive dependencies.

This example demonstrates how normalisation progressively organizes data to reduce redundancy and improve data integrity. Each step builds on the previous one, resulting in a well-structured, efficient database design.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PRACTISE:

-- Create Tables

CREATE TABLE customers (

customer\_id INT PRIMARY KEY AUTO\_INCREMENT,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

email VARCHAR(100) UNIQUE,

registration\_date DATE

);

CREATE TABLE products (

product\_id INT PRIMARY KEY AUTO\_INCREMENT,

product\_name VARCHAR(100),

category VARCHAR(50),

price DECIMAL(10, 2),

stock\_quantity INT

);

CREATE TABLE orders (

order\_id INT PRIMARY KEY AUTO\_INCREMENT,

customer\_id INT,

order\_date DATETIME,

total\_amount DECIMAL(10, 2),

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

CREATE TABLE order\_items (

order\_item\_id INT PRIMARY KEY AUTO\_INCREMENT,

order\_id INT,

product\_id INT,

quantity INT,

price\_per\_unit DECIMAL(10, 2),

FOREIGN KEY (order\_id) REFERENCES orders(order\_id),

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

CREATE TABLE product\_reviews (

review\_id INT PRIMARY KEY AUTO\_INCREMENT,

product\_id INT,

customer\_id INT,

rating INT CHECK (rating BETWEEN 1 AND 5),

review\_text TEXT,

review\_date DATE,

FOREIGN KEY (product\_id) REFERENCES products(product\_id),

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

-- Advanced SQL Practice Questions

-- Question 1: Find the top 3 customers who have spent the most money,

-- along with their total spend and the number of orders they've made.

-- Question 2: Calculate the average rating for each product category,

-- but only include categories with at least 2 reviews.

-- Question 3: Find products that have never been ordered.

-- Question 4: For each customer, find the time difference between their

-- registration date and their first order date.

-- Question 5: Create a report showing the total revenue for each month,

-- along with a running total of revenue throughout the year.

-- Question 6: Identify customers who have made a purchase but have never left a product review.

-- Question 7: Find the product that has been ordered the most times (by quantity).

-- Question 8: Calculate the percentage of total revenue that each product category contributes.

-- Question 9: For each customer, find their most frequently purchased product category.

-- Question 10: Create a query to show the distribution of ratings (count of 1-star, 2-star, etc.) for each product.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Optimized 3NF Schema**

CREATE TABLE customers (

customer\_id INT PRIMARY KEY AUTO\_INCREMENT,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

email VARCHAR(100) UNIQUE,

registration\_date DATE

);

CREATE TABLE categories (

category\_id INT PRIMARY KEY AUTO\_INCREMENT,

category\_name VARCHAR(50) UNIQUE

);

CREATE TABLE products (

product\_id INT PRIMARY KEY AUTO\_INCREMENT,

product\_name VARCHAR(100),

category\_id INT,

price DECIMAL(10, 2),

stock\_quantity INT,

FOREIGN KEY (category\_id) REFERENCES categories(category\_id)

);

CREATE TABLE orders (

order\_id INT PRIMARY KEY AUTO\_INCREMENT,

customer\_id INT,

order\_date DATETIME,

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

CREATE TABLE order\_items (

order\_item\_id INT PRIMARY KEY AUTO\_INCREMENT,

order\_id INT,

product\_id INT,

quantity INT,

price\_per\_unit DECIMAL(10, 2),

FOREIGN KEY (order\_id) REFERENCES orders(order\_id),

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

CREATE TABLE product\_reviews (

review\_id INT PRIMARY KEY AUTO\_INCREMENT,

product\_id INT,

customer\_id INT,

rating INT CHECK (rating BETWEEN 1 AND 5),

review\_text TEXT,

review\_date DATE,

FOREIGN KEY (product\_id) REFERENCES products(product\_id),

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

-- SQL Queries for Problem Statements

-- 1. Find the top 3 customers who have spent the most money,

-- along with their total spend and the number of orders they've made.

SELECT

c.customer\_id,

c.first\_name,

c.last\_name,

COUNT(DISTINCT o.order\_id) AS total\_orders,

SUM(oi.quantity \* oi.price\_per\_unit) AS total\_spend

FROM

customers c

JOIN

orders o ON c.customer\_id = o.customer\_id

JOIN

order\_items oi ON o.order\_id = oi.order\_id

GROUP BY

c.customer\_id, c.first\_name, c.last\_name

ORDER BY

total\_spend DESC

LIMIT 3;

-- 2. Calculate the average rating for each product category,

-- but only include categories with at least 2 reviews.

SELECT

c.category\_name,

AVG(pr.rating) AS avg\_rating,

COUNT(pr.review\_id) AS review\_count

FROM

categories c

JOIN

products p ON c.category\_id = p.category\_id

JOIN

product\_reviews pr ON p.product\_id = pr.product\_id

GROUP BY

c.category\_id, c.category\_name

HAVING

COUNT(pr.review\_id) >= 2

ORDER BY

avg\_rating DESC;

-- 3. Find products that have never been ordered.

SELECT

p.product\_id,

p.product\_name

FROM

products p

LEFT JOIN

order\_items oi ON p.product\_id = oi.product\_id

WHERE

oi.order\_item\_id IS NULL;

-- 4. For each customer, find the time difference between their

-- registration date and their first order date.

SELECT

c.customer\_id,

c.first\_name,

c.last\_name,

c.registration\_date,

MIN(o.order\_date) AS first\_order\_date,

DATEDIFF(MIN(o.order\_date), c.registration\_date) AS days\_to\_first\_order

FROM

customers c

LEFT JOIN

orders o ON c.customer\_id = o.customer\_id

GROUP BY

c.customer\_id, c.first\_name, c.last\_name, c.registration\_date

ORDER BY

days\_to\_first\_order;

-- 5. Create a report showing the total revenue for each month,

-- along with a running total of revenue throughout the year.

SELECT

DATE\_FORMAT(o.order\_date, '%Y-%m') AS month,

SUM(oi.quantity \* oi.price\_per\_unit) AS monthly\_revenue,

SUM(SUM(oi.quantity \* oi.price\_per\_unit)) OVER (

ORDER BY DATE\_FORMAT(o.order\_date, '%Y-%m')

) AS running\_total

FROM

orders o

JOIN

order\_items oi ON o.order\_id = oi.order\_id

GROUP BY

DATE\_FORMAT(o.order\_date, '%Y-%m')

ORDER BY

month;

-- 6. Identify customers who have made a purchase but have never left a product review.

SELECT DISTINCT

c.customer\_id,

c.first\_name,

c.last\_name

FROM

customers c

JOIN

orders o ON c.customer\_id = o.customer\_id

LEFT JOIN

product\_reviews pr ON c.customer\_id = pr.customer\_id

WHERE

pr.review\_id IS NULL;

-- 7. Find the product that has been ordered the most times (by quantity).

SELECT

p.product\_id,

p.product\_name,

SUM(oi.quantity) AS total\_quantity\_ordered

FROM

products p

JOIN

order\_items oi ON p.product\_id = oi.product\_id

GROUP BY

p.product\_id, p.product\_name

ORDER BY

total\_quantity\_ordered DESC

LIMIT 1;

-- 8. Calculate the percentage of total revenue that each product category contributes.

WITH category\_revenue AS (

SELECT

c.category\_id,

c.category\_name,

SUM(oi.quantity \* oi.price\_per\_unit) AS revenue

FROM

categories c

JOIN

products p ON c.category\_id = p.category\_id

JOIN

order\_items oi ON p.product\_id = oi.product\_id

GROUP BY

c.category\_id, c.category\_name

),

total\_revenue AS (

SELECT SUM(revenue) AS total FROM category\_revenue

)

SELECT

cr.category\_name,

cr.revenue,

(cr.revenue / tr.total) \* 100 AS percentage\_of\_total

FROM

category\_revenue cr

CROSS JOIN

total\_revenue tr

ORDER BY

percentage\_of\_total DESC;

-- 9. For each customer, find their most frequently purchased product category.

WITH customer\_category\_purchases AS (

SELECT

o.customer\_id,

p.category\_id,

COUNT(\*) AS purchase\_count,

ROW\_NUMBER() OVER (

PARTITION BY o.customer\_id

ORDER BY COUNT(\*) DESC

) AS rn

FROM

orders o

JOIN

order\_items oi ON o.order\_id = oi.order\_id

JOIN

products p ON oi.product\_id = p.product\_id

GROUP BY

o.customer\_id, p.category\_id

)

SELECT

c.customer\_id,

c.first\_name,

c.last\_name,

cat.category\_name AS most\_frequent\_category,

ccp.purchase\_count

FROM

customer\_category\_purchases ccp

JOIN

customers c ON ccp.customer\_id = c.customer\_id

JOIN

categories cat ON ccp.category\_id = cat.category\_id

WHERE

ccp.rn = 1;

-- 10. Create a query to show the distribution of ratings (count of 1-star, 2-star, etc.) for each product.

SELECT

p.product\_id,

p.product\_name,

SUM(CASE WHEN pr.rating = 1 THEN 1 ELSE 0 END) AS one\_star,

SUM(CASE WHEN pr.rating = 2 THEN 1 ELSE 0 END) AS two\_star,

SUM(CASE WHEN pr.rating = 3 THEN 1 ELSE 0 END) AS three\_star,

SUM(CASE WHEN pr.rating = 4 THEN 1 ELSE 0 END) AS four\_star,

SUM(CASE WHEN pr.rating = 5 THEN 1 ELSE 0 END) AS five\_star

FROM

products p

LEFT JOIN

product\_reviews pr ON p.product\_id = pr.product\_id

GROUP BY

p.product\_id, p.product\_name

ORDER BY

p.product\_id;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**EXAMPLE : Create a sample table for stock prices**

CREATE TABLE stock\_prices (

date DATE,

stock\_symbol VARCHAR(10),

closing\_price DECIMAL(10, 2)

);

-- Insert sample data

INSERT INTO stock\_prices (date, stock\_symbol, closing\_price) VALUES

('2023-01-01', 'AAPL', 150.00),

('2023-01-02', 'AAPL', 152.50),

('2023-01-03', 'AAPL', 151.75),

('2023-01-04', 'AAPL', 153.00),

('2023-01-05', 'AAPL', 155.25),

('2023-01-01', 'GOOGL', 2800.00),

('2023-01-02', 'GOOGL', 2825.00),

('2023-01-03', 'GOOGL', 2815.50),

('2023-01-04', 'GOOGL', 2830.00),

('2023-01-05', 'GOOGL', 2850.75);

select date,

stock\_symbol,

closing\_price,

lead(closing\_price,2) over (partition by stock\_symbol order by date) as next\_day\_price

from stock\_prices

order by stock\_symbol, date;

select date,

stock\_symbol,

lag(closing\_price,1) over (partition by stock\_symbol order by date) as previous\_day\_price

,closing\_price

from stock\_prices

order by stock\_symbol, date;

select date,

stock\_symbol,

closing\_price,

lag(closing\_price,1) over (partition by stock\_symbol order by date) as previous\_day\_price

,closing\_price-LAG(closing\_price,1) over (partition by stock\_symbol order by date) as price\_change

from stock\_prices

order by stock\_symbol, date;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**PRACTICE EXERCISE:**

-- Create tables

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

category VARCHAR(50),

unit\_price DECIMAL(10, 2)

);

CREATE TABLE sales (

sale\_id INT PRIMARY KEY,

product\_id INT,

sale\_date DATE,

quantity INT,

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

-- Insert sample data

INSERT INTO products (product\_id, product\_name, category, unit\_price) VALUES

(1, 'Laptop', 'Electronics', 999.99),

(2, 'Smartphone', 'Electronics', 599.99),

(3, 'Tablet', 'Electronics', 399.99),

(4, 'Desk Chair', 'Furniture', 149.99),

(5, 'Coffee Table', 'Furniture', 199.99),

(6, 'Bookshelf', 'Furniture', 89.99),

(7, 'Running Shoes', 'Apparel', 79.99),

(8, 'T-shirt', 'Apparel', 19.99),

(9, 'Jeans', 'Apparel', 59.99);

INSERT INTO sales (sale\_id, product\_id, sale\_date, quantity) VALUES

(1, 1, '2023-01-15', 2),

(2, 2, '2023-01-16', 3),

(3, 4, '2023-01-17', 1),

(4, 7, '2023-01-18', 4),

(5, 3, '2023-02-01', 2),

(6, 5, '2023-02-02', 1),

(7, 8, '2023-02-03', 5),

(8, 1, '2023-02-15', 1),

(9, 6, '2023-02-16', 2),

(10, 2, '2023-02-17', 2),

(11, 9, '2023-03-01', 3),

(12, 4, '2023-03-02', 2),

(13, 7, '2023-03-03', 3),

(14, 3, '2023-03-15', 1),

(15, 5, '2023-03-16', 1);

-- Problem Statement:

-- Analyze the sales data to find the top-performing product category for each month of 2023.

-- The performance should be based on total revenue (quantity sold \* unit price).

-- Your query should return the month, the top-performing category, and its total revenue.

-- In case of a tie, include all categories with the same top performance.

-- Expected output format:

-- | month | top\_category | total\_revenue |

-- |-------|--------------|---------------|

-- | 2023-01 | Electronics | 2799.97 |

-- | 2023-02 | Furniture | 399.98 |

-- | 2023-03 | Electronics | 999.98 |

DAY 19 : 10/10/24

## **Basic Query Processing Steps**

1. Parsing (Parse Tree)

· Breaks query into tokens.

· Builds a parse tree to check syntax.

2. Semantic Analysis

· Validates query against the database schema.

· Ensures data types and references are correct.

3. Query Optimization

· Analyzes different execution strategies.

· Chooses the most efficient execution plan.

4. Execution

· Executes the optimized plan.

· Retrieves and returns the query results.

Query Optimization:

1. Statistical Analysis

· Analyzes statistics (e.g., table size, index usage) to understand data distribution.

· Helps the optimizer estimate the cost of various operations.

2. Cost Estimation

· Calculates the resource cost (e.g., CPU, memory, disk I/O) for different query execution strategies.

· Helps compare different possible query plans.

3. Plan Generator

· Generates multiple potential execution plans (e.g., using different join methods or indexes).

· Considers various strategies like nested loops, hash joins, or merge joins.

4. Plan Selection

· Selects the optimal query execution plan based on cost estimation.

· The chosen plan minimizes resource usage and maximizes performance.

**Storage Engines:**

Storage engines are the components in a database that handle how data is stored, retrieved, and managed.

Common Storage Engines:

InnoDB : The default storage engine for MySQL, designed for transactional and high-performance operations.

· Transactions: Supports safe operations like commit and rollback.

· Row locking: Locks only specific rows being updated, which allows more users to work at the same time.

· Foreign keys: Maintains relationships between tables, ensuring data consistency.

MyISAM: An older storage engine, mainly used for databases with heavy read operations.

· Great for reading: Best for databases with lots of reads, like reporting or archives.

· Table locking: Locks the whole table for changes, which can slow down when multiple users try to write at once.

· No transactions or foreign keys, but fast for simpler databases with mostly reads.

Joins and Performance Tips:

1. Joins – Trim on Indexed Columns

o What it means: Use indexed columns in join conditions for faster lookups.

o Tip: Make sure the columns being compared in joins (like ON conditions) are indexed to speed up query performance.

2. Larger Table: Hash Join

o What it means: For large tables, a hash join is efficient.

o How it works: It builds a hash table for one of the tables (usually the smaller one) and then matches rows from the larger table using the hash values.

3. Sorted Sort Merge Join

o What it means: This join is used when the data is already sorted.

o How it works: The two tables are merged by sorting the rows, then matching the sorted rows directly, which is efficient for large, sorted datasets.

**Procedures:**

A procedure (or stored procedure) is a set of SQL statements that can be stored in the database and executed later as a single unit. It allows for reusable code, which can perform complex operations like inserts, updates, or calculations.

1. Delimiter //

When creating stored procedures or functions in MySQL, we change the default delimiter (;) to another symbol (e.g., //). This allows the use of semicolons inside the procedure without MySQL interpreting them as the end of the statement. Once the procedure is defined, the delimiter is switched back to ;.

2. Creating a Procedure

A stored procedure in MySQL allows you to define reusable SQL code. In the example, the insert\_rec() procedure inserts 10,000 records into large\_table using a loop. Each record is given a name (Name0, Name1, etc.) and a random value. Procedures can be called as needed to automate tasks like bulk inserts.

Delimiter //

create procedure

insert\_rec()

begin

declare i int default 0;

while i<10000 do

insert into large\_table(name,value)

values(concat('Name',i),floor(1+rand()\*10000));

set i=i+1;

end while;

end //

Delimiter ;

call insert\_rec();

3. Show Procedure Status

The show procedure status command provides information on all the stored procedures in a database. You can use it to verify whether your stored procedure was successfully created and check details like when it was last modified or executed.

show procedure status where db='test\_db\_poc';

4. Count Records in Table

You can use select count(\*) to count the number of rows in a table before and after calling a procedure like insert\_rec(). This helps in verifying that the procedure worked correctly and added the expected number of records.

select count(\*) from large\_table;

5. SQL\_NO\_CACHE

The sql\_no\_cache option forces MySQL to execute the query without using any cached results. This is useful for cases where you want the most up-to-date data, ensuring that the sum or any other calculation reflects the current state of the table, not an older cached result.

select sql\_no\_cache sum(value) from large\_table;

6. Creating an Index

Creating an index on a column (e.g., value) improves the performance of search queries involving that column. Indexes allow faster lookups by keeping a sorted data structure, but they can slightly slow down operations like inserts and updates due to the need to maintain the index.

create index idx\_val on large\_table(value);

7. Dropping an Index

If an index is no longer needed or is slowing down certain operations, it can be dropped using alter table drop index. Removing the index makes insert and update operations faster but can slow down searches on the indexed column.

alter table large\_table drop index idx\_val;

8. Cross Join

A cross join combines every row from one table with every row from another table, producing a Cartesian product. In the example, the query compares values from two instances of large\_table, finding rows where a.value equals b.value \* 100. Cross joins are powerful but can be resource-intensive, especially on large tables.

select a.value from large\_table a

cross join large\_table b

on a.value=b.value\*100;

**StoredProcedureConnection using JAVA**

This Java program is designed to interact with a MySQL database. It connects to the database, executes a stored procedure, retrieves data, and manages resources efficiently. This program is a practical example of how to use JDBC (Java Database Connectivity) for interacting with MySQL databases to execute stored procedures and query data.

package com.Database;

import java.sql.\*;

public class StoredProcedureConnection {

private static final String URL = "jdbc:mysql://localhost:3306/day4db";

private static final String USERNAME = "root";

private static final String PASSWORD = "root";

private static Connection connection;

private static Statement statement;

private static ResultSet resultSet;

private static CallableStatement callableStatement;

public static void main(String[] args) {

try {

connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);

System.out.println("Connection established successfully!");

callInsertProcedure();

countRecords();

} catch (SQLException e) {

e.printStackTrace();

} finally {

closeResources();

}

}

private static void callInsertProcedure() throws SQLException {

String procedureCall = "{ CALL INSERT\_MILLION\_RECORDS() }";

callableStatement = connection.prepareCall(procedureCall);

callableStatement.execute();

System.out.println("Procedure called successfully.");

}

private static void countRecords() throws SQLException {

String query = "SELECT COUNT(\*) AS total FROM large\_table";

statement = connection.createStatement();

resultSet = statement.executeQuery(query);

if (resultSet.next()) {

int count = resultSet.getInt("total");

System.out.println("Total records in large\_table: " + count);

}

}

private static void closeResources() {

try {

if (resultSet != null) resultSet.close();

if (statement != null) statement.close();

if (callableStatement != null) callableStatement.close();

if (connection != null) connection.close();

System.out.println("Resources closed.");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Key Steps in the Program

1. Database Connection:

The program begins by establishing a connection to the MySQL database using the

credentials (database URL, username, and password). A successful connection is confirmed

with a print statement.

2. Executing a Stored Procedure:

A method is used to call a stored procedure named insert\_rec(). This procedure inserts

records into a table called large\_table in the database. Using a CallableStatement, Java can

execute stored procedures stored in the database, allowing for more complex operations to

be encapsulated and reused.

3. Querying Data:

After executing the stored procedure, the program runs a SQL query to count the total

number of records in the large\_table. The result is retrieved and displayed to show the

impact of the stored procedure.

4. Resource Management:

Proper resource management is crucial when working with databases. The program ensures

that all connections, statements, and result sets are closed after use. This prevents potential

memory leaks and improves the program's efficiency.

### Limit of System:

The SHOW VARIABLES LIKE 'max%' command in MySQL displays system limits related to

resources like connections, packet size, and memory usage. Key variables include

max\_connections(maximum client connections), max\_allowed\_packet (largest packet

size),max\_heap\_table\_size (in memory table size), max\_sort\_length (maximum string length for table).

### QUERY HINTING

Query hinting in SQL refers to giving explicit instructions to the query optimizer on how to execute a query. Normally, the optimizer decides the best way to execute a query based on indexes, statistics,and other factors. However, in some cases, you might want to force the optimizer to follow a specific path, particularly when the automatic optimization isn't producing the desired result.

DAY 20 : 11/10/24

Dimensional Modelling

Dimensional modelling is a data modelling technique used in data warehouses to optimise query performance and data retrieval for analytics. It is different from traditional relational database design(normalisation) as it focuses on the end user's perspective, making it easier to extract insights. The Key components of dimensional modelling are fact tables and dimension tables

Components of Dimensional Modelling:

1. Fact Table:

* A central table in a star or snowflake schema.
* Stores quantitative data for analysis, usually numerical values (e.g., sales amount, units sold).
* It contains foreign keys that refer to the dimension tables.
* Fact tables are usually denormalized.

2. Dimension Table:

* Contains descriptive attributes (textual or categorical) related to dimensions, which describe the facts.
* For example, in a sales scenario, dimensions could be Product, Time, Location, etc.
* Dimension tables are typically denormalized to support fast query performance.

Types of Schemas:

1. Star Schema:

* A simple dimensional model where a fact table is connected to multiple dimension tables directly.
* The structure resembles a star, with the fact table at the centre and dimensions surrounding it.
* Easier to understand and query, but less normalized.

2. Snowflake Schema:

* An extension of the star schema where dimension tables are normalized in multiple related tables.
* Results in more complex joins but can reduce data redundancy.

3. Galaxy Schema (or Fact Constellation):

* A more complex design where multiple fact tables share dimension tables.
* Useful for representing different business processes.

### DATA WAREHOUSE

A data warehouse is a central repository of information that can be analysed to make more informed decisions.

• All Quantitative data will be stored in the fact table

•All descriptive data will be stored in the dimension table

Normalization: Reduces repetition of data.

Denormalization:

* Denormalization is a database optimization technique in which we add redundant data to one or more tables.
* This can help us avoid costly joins in a relational database.

Star Schema: In a star schema, the fact table holds measurable data (like sales amounts), while dimension tables store descriptive data (like product names or customer details). The fact table is directly connected to each dimension table, which makes the schema simple and fast to query.

Example: Imagine a retail store that tracks sales. You have the fact table for sales and several dimension tables for details like product, time, customer, and store.

• Fact Table:

Stores numbers (like total sales, number of units sold).

Also contains foreign keys pointing to each dimension table.

• Dimension Tables:

Hold descriptive information, like product names, customer details, etc.

Snowflake Schema: A snowflake schema is a variation of the star schema. In a snowflake schema, dimension tables are further normalized (broken down into more tables) to reduce redundancy.

Example: In the Product\_Dim table from the star schema, the product's category can be moved to a separateCategory\_Dim table. This way, you reduce duplicate information and make the schema look like a snowflake.

Fact Constellation (Galaxy Schema)

A fact constellation (or galaxy schema) is a more complex design where you have multiple fact tables sharing dimension tables. This is useful when you have different business processes to track,and each process has its own fact table.

Example: In addition to tracking Sales, you might also want to track Inventory. Both Sales\_Fact andInventory\_Fact can share the same dimensions, like product and store.

Sales\_Fact tracks sales-related data.

Inventory\_Fact tracks inventory-related data.

Both facts share the same Product\_Dim, Time\_Dim, and Store\_Dim, making it a constellation of fact.

Fact less table:

A fact-less fact table is a special type of fact table in a dimensional model that doesn't have any numeric or measurable data (like sales amount or units sold). Instead, it records events or relationships between dimension tables. These tables help track the occurrence of certain events or describe conditions that involve only dimensions, without any associated metrics.

Junk Dimension: A junk dimension is a dimension in a data warehouse that combines several low-cardinality (low distinct values) attributes, which do not belong to any specific dimension. By combining unrelated or unimportant attributes into a single junk Dimension: a Degenerate Dimension, often termed as a junk dimension, is a type of dimension table in a star schema of a data warehouse. It does not have its dimension table but resides in the fact table as a primary key.

Slowly Changing Dimensions (SCD): Slowly Changing Dimensions (SCD) are a concept in data warehousing that describe how to manage and track changes in dimension tables.

• SCD Type 1: Overwriting the DataIn SCD Type 1, the old data is overwritten with new data whenever changes occur. Nohistory of the changes is maintained, meaning that once the data is updated, the previous values are lost.

• SCD Type 2: Maintaining Historical DataIn SCD Type 2, changes are tracked by creating new rows in the dimension table to preserve the full history of data changes. Each version of a record is stored with an effective date range (or start/end date) or other mechanisms to differentiate between current and historical records.

ACTIVITY USE CASE: ANALYSIS ON FINANCIAL TRANSACTIONS

create DATABASE financial\_transactions\_db;

USE financial\_transactions\_db;

-- Enable Profiling

SET profiling = 1;

-- Create Customers Table

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY AUTO\_INCREMENT,

Name VARCHAR(100),

Segment VARCHAR(50)

);

-- Create Accounts Table

CREATE TABLE Accounts (

AccountID INT PRIMARY KEY AUTO\_INCREMENT,

CustomerID INT,

AccountType VARCHAR(50),

Balance DECIMAL(15, 2),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

-- Create Transactions Table

CREATE TABLE Transactions (

TransactionID INT PRIMARY KEY AUTO\_INCREMENT,

AccountID INT,

TransactionDate DATE,

Amount DECIMAL(15, 2),

TransactionType VARCHAR(50),

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

-- Insert data into Customers Table (1,000 customers)

INSERT INTO Customers (Name, Segment)

SELECT CONCAT('Customer', FLOOR(RAND() \* 100000)),

CASE WHEN RAND() > 0.5 THEN 'Retail' ELSE 'Corporate' END

FROM (SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 UNION SELECT 5) a

CROSS JOIN (SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 UNION SELECT 5) b;

-- Insert data into Accounts Table (10,000 accounts)

SET @customerCount = (SELECT COUNT(\*) FROM Customers);

INSERT INTO Accounts (CustomerID, AccountType, Balance)

SELECT FLOOR(RAND() \* @customerCount) + 1,

CASE WHEN RAND() > 0.5 THEN 'Savings' ELSE 'Checking' END,

ROUND(RAND() \* 10000, 2)

FROM (SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 UNION SELECT 5) a

CROSS JOIN (SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 UNION SELECT 5) b;

-- Insert data into Transactions Table (1 million transactions)

SET @accountCount = (SELECT COUNT(\*) FROM Accounts);

INSERT INTO Transactions (AccountID, TransactionDate, Amount, TransactionType)

SELECT FLOOR(RAND() \* @accountCount) + 1,

CURDATE() - INTERVAL FLOOR(RAND() \* 365) DAY, -- Random date in the past year

ROUND(RAND() \* 1000, 2), -- Random amount

CASE WHEN RAND() > 0.5 THEN 'Deposit' ELSE 'Withdrawal' END

FROM (SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 UNION SELECT 5) a

CROSS JOIN (SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 UNION SELECT 5) b

CROSS JOIN (SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 UNION SELECT 5) c

CROSS JOIN (SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 UNION SELECT 5) d;

--

--

--

-- Slow Query (Normalized Schema)

-- For each customer segment and account type, how much was deposited and withdrawn each year

explain SELECT c.Segment, a.AccountType, YEAR(t.TransactionDate) AS Year,

SUM(CASE WHEN t.TransactionType = 'Deposit' THEN t.Amount ELSE 0 END) AS TotalDeposits,

SUM(CASE WHEN t.TransactionType = 'Withdrawal' THEN t.Amount ELSE 0 END) AS TotalWithdrawals

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

JOIN Customers c ON a.CustomerID = c.CustomerID

GROUP BY c.Segment, a.AccountType, YEAR(t.TransactionDate)

ORDER BY Year, c.Segment, a.AccountType;

SHOW PROFILES;

-- Transition to Galaxy Schema (Dimensional Modeling)

-- Create DimDate Table

CREATE TABLE DimDate (

DateID INT PRIMARY KEY AUTO\_INCREMENT,

Date DATE,

Year INT,

Month INT

);

-- Create DimCustomer Table

CREATE TABLE DimCustomer (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

Segment VARCHAR(50)

);

-- Create DimAccount Table

CREATE TABLE DimAccount (

AccountID INT PRIMARY KEY,

AccountType VARCHAR(50)

);

-- Create FactTransactions Table

CREATE TABLE FactTransactions (

TransactionID INT PRIMARY KEY AUTO\_INCREMENT,

DateID INT,

CustomerID INT,

AccountID INT,

Amount DECIMAL(15, 2),

TransactionType VARCHAR(50),

FOREIGN KEY (DateID) REFERENCES DimDate(DateID),

FOREIGN KEY (CustomerID) REFERENCES DimCustomer(CustomerID),

FOREIGN KEY (AccountID) REFERENCES DimAccount(AccountID)

);

-- Create FactAccountBalances Table

CREATE TABLE FactAccountBalances (

BalanceID INT PRIMARY KEY AUTO\_INCREMENT,

DateID INT,

AccountID INT,

Balance DECIMAL(15, 2),

FOREIGN KEY (DateID) REFERENCES DimDate(DateID),

FOREIGN KEY (AccountID) REFERENCES DimAccount(AccountID)

);

-- Insert unique Date and Time details into DimDate

INSERT INTO DimDate (Date, Year, Month)

SELECT DISTINCT DATE(t.TransactionDate), YEAR(t.TransactionDate), MONTH(t.TransactionDate)

FROM Transactions t;

-- Insert data into DimCustomer from Customers table

INSERT INTO DimCustomer (CustomerID, Name, Segment)

SELECT CustomerID, Name, Segment FROM Customers;

-- Insert data into DimAccount from Accounts table

INSERT INTO DimAccount (AccountID, AccountType)

SELECT AccountID, AccountType FROM Accounts;

-- Insert data into FactTransactions

INSERT INTO FactTransactions (DateID, CustomerID, AccountID, Amount, TransactionType)

SELECT dd.DateID, a.CustomerID, a.AccountID, t.Amount, t.TransactionType

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

JOIN DimDate dd ON DATE(t.TransactionDate) = dd.Date;

--

--

--

-- Optimized Query (Galaxy Schema)

-- annual financial transaction summaries

explain SELECT dc.Segment, da.AccountType, dd.Year,

SUM(CASE WHEN ft.TransactionType = 'Deposit' THEN ft.Amount ELSE 0 END) AS TotalDeposits,

SUM(CASE WHEN ft.TransactionType = 'Withdrawal' THEN ft.Amount ELSE 0 END) AS TotalWithdrawals

FROM FactTransactions ft

JOIN DimDate dd ON ft.DateID = dd.DateID

JOIN DimCustomer dc ON ft.CustomerID = dc.CustomerID

JOIN DimAccount da ON ft.AccountID = da.AccountID

GROUP BY dc.Segment, da.AccountType, dd.Year

ORDER BY dd.Year, dc.Segment, da.AccountType;

SHOW PROFILES;

TRIGGERS

A trigger is a stored procedure in a database that automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when specific table columns are updated. In simple words, a trigger is a collection of SQL statements with particular names that are stored in system memory. It belongs to a specific class of stored procedures that are automatically invoked in response to database server events. Every trigger has a table attached to it.

Because a trigger cannot be called directly, unlike a stored procedure, it is referred to as a special procedure. A trigger is automatically called whenever a data modification event against a table takes place, which is the main distinction between a trigger and a procedure. On the other hand, a stored procedure must be called directly.

The following are the key differences between triggers and stored procedures:

* Triggers cannot be manually invoked or executed.
* There is no chance that triggers will receive parameters.
* A transaction cannot be committed or rolled back inside a trigger.

EXAMPLE:

create database trigger\_practise;

use trigger\_practise;

create table customers(id int auto\_increment primary key,

name varchar(100),

email varchar(100));

create table email\_changes\_log(

id int auto\_increment primary key,

customer\_id int,

old\_email varchar(100),

new\_email varchar(100),

changed\_at timestamp default current\_timestamp);

insert into customers(name,email) values('Auahdahd','dqhduiqwh@gmail.com');

select \* from customers;

DELIMITER //

CREATE TRIGGER log\_email\_changes

before update on customers

for each row

begin

if old.email!=new.email then

insert into email\_changes\_log(customer\_id,old\_email,new\_email)

values(old.id,old.email, new.email);

end if;

end//

delimiter ;

select \* from email\_changes\_log;

update customers set email='jdiejweiod@gmail.com' where id =1

select \* from customers

DAY 21 : 14/10/24

## SPRING

Spring Framework is one of the most popular and comprehensive frameworks in Java, designed to simplify enterprise-level applications by providing infrastructure support. It allows developers to create Java applications with high scalability, flexibility, and ease of maintenance. Here’s an overview of key concepts:

1. Inversion of Control (IoC)

Spring’s core feature is IoC which allows objects to be created and managed by the framework rather than by the developer. This is typically done using Dependency Injection (DI).

DI can be implemented in two ways: Constructor Injection and Setter Injection.

2. Spring Modules

Spring is modular, meaning you can use only the parts you need. Major modules include:

Spring Core: Provides the fundamental features like IoC, DI.

Spring AOP (Aspect-Oriented Programming): Enables separation of cross-cutting concerns (like logging, transaction management).

Spring Data: Facilitates interaction with databases using technologies like JDBC or Hibernate.

Spring MVC: A module for developing web applications based on the MVC pattern.

Spring Security: Provides security features such as authentication and authorization.

Spring Boot: Simplifies application development by providing default configurations for Spring applications.

3. Spring Boot

Spring Boot is a project that simplifies the Spring framework by providing pre-configured templates for application development.

It allows developers to create standalone applications with an embedded web server like Tomcat.

Spring Initializr is a tool that helps you quickly bootstrap a Spring Boot application.

4. Annotations

Spring makes extensive use of annotations to reduce XML configuration:

@Component, @Service, @Repository: For managing beans.

@Autowired: For injecting dependencies.

@Configuration, @Bean: For configuring Spring beans programmatically.

@RequestMapping, @GetMapping, @PostMapping: For mapping HTTP requests in Spring MVC.

5. Spring MVC

Spring MVC (Model-View-Controller) is used for building web applications.

It allows easy mapping of HTTP requests to handler methods using annotations like @Controller and @RequestMapping.

6. Spring Security

Provides a flexible security mechanism to secure applications.

Supports authentication (login mechanisms) and authorization (role-based access).

7. Spring Data

Simplifies database interactions.

Works with both relational databases (using JPA, Hibernate) and NoSQL databases (like MongoDB).

8. Transaction Management

Spring provides declarative transaction management using @Transactional.

It abstracts transaction management across different platforms (JDBC, JPA, etc.).

9. Benefits of Spring

Loose Coupling: Because of IoC, objects are loosely coupled, making the application more maintainable.

Comprehensive Ecosystem: Spring has a wide array of projects, such as Spring Cloud for microservices, Spring Batch for batch processing, etc.

Testability: Spring makes it easier to test Java applications using tools like JUnit and Mockito.

**# Understanding Dependency Injection and Inversion of Control in Spring**

## 1. Introduction to Dependency Injection (DI)

Dependency Injection is a design pattern that allows us to develop loosely coupled code. It's a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

### Step 1: Understanding the Problem DI Solves

Let's start with a simple example without DI:

java

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor() {

this.spellChecker = new SpellChecker();

}

}

In this case, TextEditor is tightly coupled to SpellChecker. If we want to use a different spell checker, we'd have to modify the TextEditor class.

### Step 2: Applying Dependency Injection

Now, let's refactor this using DI:

java

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

this.spellChecker = spellChecker;

}

}

Now, TextEditor is not responsible for creating the SpellChecker. It's "injected" from outside.

## 2. Types of Dependency Injection

Spring supports three types of dependency injection. Let's explore each one.

### Step 3: Constructor Injection

This is the most recommended form of DI in Spring.

java

@Component

public class TextEditor {

private final SpellChecker spellChecker;

@Autowired

public TextEditor(SpellChecker spellChecker) {

this.spellChecker = spellChecker;

}

}

Spring will automatically inject a SpellChecker bean when creating a TextEditor.

### Step 4: Setter Injection

Setter injection uses, well, setter methods:

java

@Component

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public void setSpellChecker(SpellChecker spellChecker) {

this.spellChecker = spellChecker;

}

}

### Step 5: Field Injection

Field injection injects dependencies directly into fields:

java

@Component

public class TextEditor {

@Autowired

private SpellChecker spellChecker;

}

Note: While convenient, field injection is generally discouraged as it makes testing more difficult.

## 3. Inversion of Control (IoC)

IoC is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework.

### Step 6: Understanding IoC

In traditional programming, our custom code makes calls to a library. IoC is the exact opposite - the framework makes calls to our custom code.

### Step 7: Spring IoC Container

Spring's IoC container is responsible for managing object creation, configuring these objects, and managing their lifecycle.

There are two types of IoC containers in Spring:

1. BeanFactory

2. ApplicationContext (a more advanced container and extension of BeanFactory)

### Step 8: Configuring Spring IoC

We can configure the Spring IoC container using XML, Java annotations, or Java code. Let's look at an example using Java annotations:

java

@Configuration

public class AppConfig {

@Bean

public SpellChecker spellChecker() {

return new SpellChecker();

}

@Bean

public TextEditor textEditor(SpellChecker spellChecker) {

return new TextEditor(spellChecker);

}

}

This configuration tells Spring how to create and wire our objects.

## 4. Putting It All Together

Let's see how DI and IoC work together in a Spring application.

### Step 9: Creating the Application

java

@SpringBootApplication

public class TextEditorApplication {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(TextEditorApplication.class, args);

TextEditor editor = context.getBean(TextEditor.class);

editor.spellCheck("Hello, wrld!");

}

}

In this example:

1. Spring Boot sets up the ApplicationContext (IoC container).

2. The container creates and configures all the beans.

3. We retrieve the TextEditor bean from the container.

4. We use the TextEditor, which internally uses the SpellChecker that was injected.

## 5. Benefits of DI and IoC

- Reduced dependency between classes

- Easier unit testing through mocking

- Greater modularity

- Increased code reusability

- More flexible, configurable applications

By leveraging DI and IoC, Spring allows us to write more modular, testable, and maintainable code.

**Design Considerations: @Autowired vs getBean() in Spring**

**1. @Autowired**

When to Use -

In Spring-managed beans (e.g., @Component, @Service, @Repository, @Controller)

For dependencies that are required throughout the lifecycle of a bean

When you want to leverage Spring's dependency injection fully

Design Advantages

Loose Coupling: @Autowired promotes loose coupling as the dependent class doesn't need to know about the creation and management of its dependencies.

javaCopy@Service

public class OrderService {

private final PaymentService paymentService;

@Autowired

public OrderService(PaymentService paymentService) {

this.paymentService = paymentService;

}

}

Testability: It's easier to mock dependencies in unit tests.

javaCopy@Test

public void testOrderService() {

PaymentService mockPaymentService = mock(PaymentService.class);

OrderService orderService = new OrderService(mockPaymentService);

// ... test orderService

}

Cleaner Code: Reduces boilerplate code for dependency lookup and instantiation.

Consistency: Ensures that the same instance of a dependency is used throughout the application (for singleton-scoped beans).

Lifecycle Management: Spring manages the complete lifecycle of autowired dependencies.

**2. getBean()**

When to Use

In classes not managed by Spring

When you need to choose between multiple implementations at runtime

For retrieving prototype-scoped beans multiple times

In factory patterns where bean selection happens dynamically

Design Advantages

Flexibility at Runtime: Allows for dynamic selection of dependencies.

javaCopy@Component

public class PaymentProcessor {

private final ApplicationContext context;

@Autowired

public PaymentProcessor(ApplicationContext context) {

this.context = context;

}

public void processPayment(String paymentType) {

PaymentService paymentService = context.getBean(paymentType + "PaymentService", PaymentService.class);

paymentService.process();

}

}

Lazy Loading: Can delay bean creation until it's actually needed, potentially improving startup time.

javaCopy@Component

public class HeavyResourceUser {

private final ApplicationContext context;

@Autowired

public HeavyResourceUser(ApplicationContext context) {

this.context = context;

}

public void useHeavyResource() {

HeavyResource resource = context.getBean(HeavyResource.class);

resource.performOperation();

}

}

Access to Multiple Implementations: Useful when you have multiple implementations of an interface and need to choose at runtime.

javaCopy@Component

public class ReportGenerator {

private final ApplicationContext context;

@Autowired

public ReportGenerator(ApplicationContext context) {

this.context = context;

}

public void generateReport(String reportType) {

ReportService reportService = context.getBean(reportType + "ReportService", ReportService.class);

reportService.generate();

}

}

Prototype Scope Handling: When you need a new instance of a prototype-scoped bean each time.

javaCopy@Component

public class PrototypeManager {

private final ApplicationContext context;

@Autowired

public PrototypeManager(ApplicationContext context) {

this.context = context;

}

public void doSomething() {

PrototypeBean prototypeBean = context.getBean(PrototypeBean.class);

prototypeBean.process();

}

}

**3. Design Patterns and Use Cases**

Dependency Injection Pattern

Prefer @Autowired for implementing the Dependency Injection pattern in Spring-managed beans.

It aligns well with the Inversion of Control principle.

Factory Pattern

Use getBean() when implementing a Factory pattern where the exact type of object is determined at runtime.

javaCopy@Component

public class AnimalFactory {

private final ApplicationContext context;

@Autowired

public AnimalFactory(ApplicationContext context) {

this.context = context;

}

public Animal createAnimal(String animalType) {

return context.getBean(animalType, Animal.class);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Component

public class DependencyValidator {

@Autowired(required = false)

private OptionalService optionalService;

@Autowired

private RequiredService requiredService;

@PostConstruct

public void validateDependencies() {

if (requiredService == null) {

throw new IllegalStateException("RequiredService is not injected!");

}

if (optionalService == null) {

System.out.println("OptionalService is not available, some features may be limited.");

}

}

}

**Key Differences from Constructor Initialization**

Timing: @PostConstruct methods are called after all dependency injection is complete, whereas constructors are called during bean instantiation.

Access: @PostConstruct methods have full access to dependency-injected fields, which might not be initialised in the constructor.

Inheritance: @PostConstruct methods in superclasses are called before those in subclasses.

Exception Handling: @PostConstruct methods can throw checked exceptions, unlike constructors.

**VIEW**

In SQL, a view is a virtual table created by a query. It allows you to store a query as a named object and retrieve data from it like a regular table. Views are used for simplifying complex queries, enhancing security by restricting data access, and abstracting underlying table structures.

Basic syntax for creating a view:

CREATE VIEW view\_name AS

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

Example of View Table:

Let's create a view based on an example employees table:

sqlCopy-- Create the employees table

CREATE TABLE employees (

id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

department VARCHAR(50),

salary DECIMAL(10, 2)

);

-- Insert some sample data

INSERT INTO employees VALUES

(1, 'John', 'Doe', 'IT', 75000),

(2, 'Jane', 'Smith', 'HR', 65000),

(3, 'Mike', 'Johnson', 'IT', 80000),

(4, 'Emily', 'Brown', 'Finance', 70000);

**Key Points about Views:**

* Virtual table: It does not store data physically; the data is fetched dynamically when queried.
* Read-only (by default): Updates to a view might not always be allowed unless certain conditions are met.
* Simplifies queries: Abstracts complex joins or aggregations.
* Security: Limits access to sensitive data by exposing only selected parts of tables.

**Materialized View**

A Materialized View in SQL is similar to a regular view, but with a key difference: it stores the result of the query physically, just like a table. This makes it faster to retrieve data from a materialized view since the query doesn't need to be executed every time the view is accessed. However, the stored data must be refreshed when the underlying data changes.

Key Differences Between a View and a Materialized View:

1. **View**: Does not store data physically; it runs the query each time you access it.
2. **Materialized View**: Stores the query result in a physical table for faster access.

Syntax to Create a Materialized View:

CREATE MATERIALIZED VIEW view\_name

AS

SELECT column1, column2, ...

FROM table\_name

WHERE condition

Advantages of Materialized Views:

* Improved performance: Since data is precomputed and stored, queries are faster.
* Data aggregation: Useful for precomputing and storing aggregated data.
* Offline access: Materialized views can work independently of the base tables after creation.

Drawbacks:

* Storage cost: It requires physical storage for the result set.
* Staleness: The data can become outdated until it’s refreshed.
* Materialized views are particularly useful in environment

Event Scheduler:

In SQL, the event scheduler is a feature primarily found in databases like MySQL that allows for the automated execution of tasks at specified times. These tasks, called events, can be scheduled to run at regular intervals or at a specific time. While not all SQL-based systems have built-in schedulers, here’s how scheduling works in MySQL using SQL queries:

1. Enabling the Event Scheduler:

To start using the event scheduler in MySQL, you need to ensure that it’s enabled. You can check the status and enable it if necessary.

* Check the status of the event scheduler:

**SHOW VARIABLES LIKE 'event\_scheduler';**

If the value is "OFF", you'll need to enable it.

* Enable the event scheduler:

**SET GLOBAL event\_scheduler = ON;**

2. Creating an Event:

To create a scheduled task (event) in MySQL, you use the CREATE EVENT statement. Events can be set to run once at a specific time or recur at regular intervals.

#### **Example 1: A One-Time Event**

This event will run only once at the specified date and time to delete old records from a table.

**CREATE EVENT delete\_old\_records**

**ON SCHEDULE AT '2024-10-31 23:59:00'**

**DO**

**DELETE FROM orders WHERE order\_date < NOW() - INTERVAL 1 YEAR;**

#### **Example 2: A Recurring Event**

This event will run daily at midnight to update a table:

**CREATE EVENT daily\_status\_update**

**ON SCHEDULE EVERY 1 DAY STARTS '2024-10-17 00:00:00'**

**DO**

**UPDATE orders SET status = 'archived' WHERE order\_date < NOW() - INTERVAL 1 YEAR;**

3. Modifying an Event

If you need to modify an existing event (e.g., to change the schedule or task), you can use the ALTER EVENT statement.

#### **Example: Modify an Event to Run Every Week**

**ALTER EVENT daily\_status\_update**

**ON SCHEDULE EVERY 1 WEEK;**

**Example Use Cases for SQL Event Schedulers:**

* Automating Data Cleanup: Automatically deleting old records from a log or transaction table.
* Generating Reports: Creating daily, weekly, or monthly reports automatically and storing them in a report table.
* Refreshing Data: Updating or recalculating values in a summary table or refreshing materialized views.
* Backup Automation: Scheduling regular database backups to be executed at off-peak hours.

**Key Points:**

* One-time events run only at the specified time, while recurring events run on a regular interval.
* Events are dependent on the event scheduler being enabled.
* Recurring events are particularly useful for routine maintenance and data management tasks.
* This is how event scheduling works in SQL databases like MySQL. If you are using a different RDBMS, the functionality may vary (for example, using pg\_cron in PostgreSQL or DBMS\_SCHEDULER in Oracle).

## SPRING BOOT DATA JPA

Spring Boot Data JPA is a part of the Spring Data family that simplifies working with relational databases using **Java Persistence API (JPA)**. It abstracts the complexities of interacting with databases by providing boilerplate methods for CRUD operations, custom queries, pagination, and more, allowing you to focus more on your application's business logic.

Key Features of Spring Data JPA:

* CRUD operations: Automatically provides methods to perform create, read, update, and delete operations.
* Query methods: Allows the creation of queries simply by defining method names.
* Pagination and Sorting: Provides built-in support for handling pagination and sorting.
* Custom queries: Allows for custom JPQL or native SQL queries.
* Entity relationships: Supports JPA features like entity relationships (one-to-many, many-to-many, etc.).

Getting Started with Spring Boot Data JPA

Add Dependencies: In a Maven project, add the required dependencies in pom.xml.

**SPRING SECURITY**

Spring Security is a powerful and customizable authentication and access control framework for securing Java applications. It is widely used in Spring-based applications to handle tasks such as user authentication, authorization, role-based access control, and securing web applications or REST APIs.

### Key Concepts of Spring Security:

**1. \*Authentication\*:** Verifying who the user is. Spring Security provides support for various authentication methods, including form-based login, basic authentication, OAuth, JWT, and custom authentication mechanisms.

**2. \*Authorization (Access Control)\*:** Determining what actions an authenticated user is allowed to perform. This is typically based on roles or permissions. Spring Security helps in defining role-based or permission-based access control to methods, endpoints, or resources.

**3. \*Security Filters\*:** Spring Security operates using a filter chain where multiple filters (e.g., for login, logout, authentication) intercept incoming requests and handle security concerns. These filters are customizable.

**4. \*CSRF Protection\*:** Cross-Site Request Forgery (CSRF) protection is enabled by default in Spring Security to prevent malicious actions being taken on a user's behalf without their consent.

**5. \*Password Encoding\*:** Spring Security provides support for password hashing and encoding mechanisms to securely store passwords. Passwords can be hashed using algorithms like BCrypt, Pbkdf2, etc.

**6. \*Method-Level Security\*:** Spring Security allows you to secure individual methods in your application, such as service or repository methods, using annotations like @Secured, @PreAuthorize, and @PostAuthorize.

**### How Spring Security is Typically Integrated:**

1. \*\*Dependency in pom.xml or build.gradle\*\*:

You need to include the Spring Security dependency in your project.

xml

<dependency>

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

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

</dependency>

2. \*Security Configuration\*: A custom configuration class is usually created to set up authentication and authorization rules.

java

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/public/\*\*").permitAll() // Allow public access

.anyRequest().authenticated() // Restrict other requests

.and()

.formLogin() // Enable form-based authentication

.loginPage("/login").permitAll()

.and()

.logout().permitAll();

}

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth

.inMemoryAuthentication() // In-memory user store

.withUser("user").password("{noop}password").roles("USER");

}

}

3. \*Common Annotations\*:

- @EnableWebSecurity: Enables Spring Security’s web security support.

- @PreAuthorize("hasRole('ROLE\_USER')"): Secures a method to only be accessible by users with a specific role.

4. \*JWT and OAuth2\*: For stateless authentication (e.g., APIs), Spring Security can integrate JWT (JSON Web Tokens) and OAuth2 for secure token-based authentication.