Name:FILES_01

Q1. Explain the differences between SQL and NoSQL databases and provide use cases for each.

Answer: The choice between SQL and NoSQL databases is a fundamental decision in system design. SQL databases, also known as Relational Databases (RDBMS), use a structured schema and represent data in tables. They are vertically scalable, meaning you increase the capacity of a single server. SQL databases are ideal for applications requiring complex queries and multi-row transactions, such as accounting systems or legacy ERP applications, where data consistency is critical. On the other hand, NoSQL databases are non-relational and can be document-based, key-value pairs, or graph-based. They have dynamic schemas for unstructured data and are horizontally scalable, meaning they handle increased traffic by adding more servers to the cluster. NoSQL is preferred for big data applications, real-time web apps, and content management systems where the data structure may change rapidly and high availability is prioritized over strict consistency. For instance, a social media platform might use NoSQL to store diverse user-generated content, while a banking application would rely on SQL for its transactional integrity.

Q2. Discuss the concept of Microservices Architecture and its advantages over Monolithic Architecture.

Answer: Microservices architecture is a design approach where a single application is composed of many small, independent services that communicate over a network, typically using APIs. Each service is built around a specific business capability and can be developed, deployed, and scaled independently. This is a stark contrast to Monolithic architecture, where all components of the application—such as the user interface, business logic, and database access—are bundled into a single unit. The main advantage of microservices is that it allows for faster deployment cycles and continuous integration. If one service needs an update, only that service is redeployed without affecting the rest of the system. It also allows teams to use different technologies and programming languages suited for specific tasks. However, microservices introduce complexity in terms of network latency, data consistency, and service orchestration. In a monolith, communication is fast because it happens within the same memory space, but scaling is difficult because the entire app must be replicated. Microservices are ideal for large-scale, evolving projects, whereas monoliths are often sufficient for small, simple applications.

Q3. Describe the process of Memory Management in modern Operating Systems, including Paging and Segmentation.

Answer: Software Architecture refers to the high-level structure of a software system, defining the components, their relationships, and the principles guiding its design. One of the most common architectural patterns is the Layered or N-Tier architecture. In this pattern, the application is organized into horizontal layers, each with a specific responsibility. The most common layers are the Presentation Layer (User Interface), Business Logic Layer (Processing), and Data Access Layer (Database). Each layer only communicates with the layer immediately below it. This separation of concerns makes the system easier to maintain and test. For example, the user interface can be updated without changing the underlying business logic. N-Tier architecture is widely used in enterprise web applications where scalability and security are important. By separating the layers, different tiers can be hosted on different physical servers, allowing for independent scaling of the

web server, application server, and database server. While this adds some latency due to network communication between tiers, the benefits of modularity and maintainability usually outweigh the performance costs in large systems.

.

Q4. Explain the principles of Object-Oriented Programming (OOP) and their importance in software development.

Answer: Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data and code. The four fundamental principles of OOP are Encapsulation, Abstraction, Inheritance, and Polymorphism. Encapsulation involves bundling data particularly useful for managing large, complex software projects where multiple developers work on different modules, as it provides a clear structure and reduces redundancy across the codebase.

Q5. What is Software Architecture? Explain the Layered (N-Tier) architecture and its practical applications.

Answer: Software Architecture refers to the high-level structure of a software system, defining the components, their relationships, and the principles guiding its design. One of the most common architectural patterns is the Layered or N-Tier architecture. In this pattern, the application is organized into horizontal layers, each with a specific responsibility. The most common layers are the Presentation Layer (User Interface), Business Logic Layer (Processing), and Data Access Layer (Database). Each layer only communicates with the layer immediately below it. This separation of concerns makes the system easier to maintain and test. For example, the user interface can be updated without changing the underlying business logic. N-Tier architecture is widely used in enterprise web applications where scalability and security are important. By separating the layers, different tiers can be hosted on different physical servers, allowing for independent scaling of the web server, application server, and database server. While this adds some latency due to network communication between tiers, the benefits of modularity and maintainability usually outweigh the performance costs in large systems.

SECTION 2: CODING / LOGIC QUESTIONS

Q6. Write a Python program to implement a Binary Search Tree (BST) with insertion and in-order traversal logic.

Answer: class BSTNode: def **init**(self, val): self.left = None self.right = None self.value = val

def insert(root, key): if root is None: return BSTNode(key) else: if root.value == key: return root elif root.value < key: root.right = insert(root.right, key) else: root.left = insert(root.left, key) return root

def inorder_traversal(root): res = [] if root: res = inorder_traversal(root.left) res.append(root.value) res = res + inorder_traversal(root.right) return res

Testing the BST

root_node = BSTNode(50) root_node = insert(root_node, 30) root_node = insert(root_node, 20) root_node = insert(root_node, 40) root_node = insert(root_node, 70) root_node = insert(root_node, 60) root_node = insert(root_node, 80)

print("In-order traversal of the BST:") print(inorder_traversal(root_node))

Expected output: [20, 30, 40, 50, 60, 70, 80]

Q7. Implement a Python script that fetches data from a mock REST API using the requests library and handles edge cases.

Answer: import requests import time

def fetch_user_data(user_id): url = f"https://jsonplaceholder.typicode.com/users/{user_id}" try: response = requests.get(url, timeout=5)

Check if the request was successful

if response.status_code == 200:

}

elif response.status_code == 404:

return {"success": False, "error": "User not found"}

else:

return {"success": False, "error": f"Server error: {response.status_code}"}

except requests.exceptions.Timeout:

return {"success": False, "error": "The request timed out"}

except requests.exceptions.ConnectionError:

return {"success": False, "error": "Failed to connect to the server"}

except Exception as e:

return {"success": False, "error": str(e)}

Running test cases

for i in [1, 999]: # 1 exists, 999 does not print(f"Fetching data for User ID {i}...") result = fetch_user_data(i) print(result) time.sleep(1)

Q8. Write a Python implementation of the Bubble Sort algorithm with an optimization to stop early if the list is sorted.

Answer: def optimized_bubble_sort(arr): n = len(arr) for i in range(n): # Flag to check if any swapping happened swapped = False

Last i elements are already in place

```
    for j in range(0, n - i - 1):
        if arr[j] > arr[j + 1]:
            # Swap if the element found is greater than the next
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            swapped = True

    # If no two elements were swapped by inner loop, then break
    if not swapped:
        print(f"Array sorted early at iteration {i+1}")
        break
return arr
```

Testing with different cases

unsorted_list = [64, 34, 25, 12, 22, 11, 90] nearly_sorted = [1, 2, 3, 5, 4, 6, 7]

print("Sorting unsorted list:", optimized_bubble_sort(unsorted_list)) print("Sorting nearly sorted list:", optimized_bubble_sort(nearly_sorted))

Q9. Create a Python function to read a large log file and count the occurrences of specific keywords using a dictionary.

Answer: import os

def analyze_log_file(file_path, keywords): counts = {key: 0 for key in keywords}

```
if not os.path.exists(file_path):
    print("Log file not found.")
    return None

try:
    with open(file_path, 'r') as file:
        for line_num, line in enumerate(file, 1):
            clean_line = line.lower()
            for key in keywords:
                if key.lower() in clean_line:
                    counts[key] += 1

    return counts
```

except Exception as e:

print(f"An error occurred: {e}")

return None

Simulation: Creating a dummy log file

dummy_log = "app.log" with open(dummy_log, "w") as f: f.write("ERROR: Database connection failed\n") f.write("INFO: User logged in\n") f.write("DEBUG: Query executed in 10ms\n") f.write("ERROR: Timeout reached\n")

search_terms = ["ERROR", "INFO", "DEBUG"] report = analyze_log_file(dummy_log, search_terms) print("Log Analysis Report:", report)

Q10. Implement a Thread-safe Singleton pattern in Python using a metaclass.

Answer: import threading

class SingletonMeta(type): _instances = {} _lock: threading.Lock = threading.Lock()

def __call__(cls, *args, **kwargs):

Double-checked locking to ensure thread safety

if cls not in cls._instances:

with cls._lock:

if cls not in cls._instances:

instance = super().__call__(*args, **kwargs)

cls._instances[cls] = instance

return cls._instances[cls]

class DatabaseConnector(metaclass=SingletonMeta): def **init**(self): self.connection_string = "db://localhost:5432" print("Initializing Database Connection...")

Testing Singleton across threads

def test_singleton(): db1 = DatabaseConnector() print(f"DB1 Instance ID: {id(db1)}")

thread1 = threading.Thread(target=test_singleton) thread2 = threading.Thread(target=test_singleton)

thread1.start() thread2.start() thread1.join() thread2.join()

SECTION 3: PROGRAMMING LANGUAGE QUESTIONS

Q11. What are Python List Comprehensions and why are they used?

Answer: The GIL is a mutex that protects access to Python objects, preventing multiple native threads from executing Python bytecodes at once in a single process. This means that even on multi-core systems, a multi-threaded Python program will only run on one CPU core at a time for CPU-bound tasks. To achieve true parallelism for CPU-heavy operations, developers use the multiprocessing module instead of threading.

.

Q12. Explain the difference between __init__ and __new__ in Python.

Answer: __new__ is the method responsible for creating a new instance of a class. it is a static method that returns the instance itself. __init__ is the initializer method that takes the instance created by __new__ and sets up its initial state or attributes. In most cases, you only need to override __init__, but __new__ is used in special cases like creating Singletons or inheriting from immutable types like int or str.

Q13. What is the purpose of the with statement in Python?

Answer: The with statement is used for resource management and exception handling. It simplifies the process of working with resources like files, network connections, or locks by ensuring they are properly closed or released after use, even if an exception occurs. It works by utilizing "Context Managers" which define __enter__ and __exit__ methods.

Q14. How does Python's Global Interpreter Lock (GIL) affect multi-threading?

Answer: List comprehensions provide a concise way to create lists in Python. They consist of brackets containing an expression followed by a for clause, then zero or more for or if clauses. They are used because they are more readable and often faster than traditional for loops using append(). For example: [x**2 for x in range(10) if x % 2 == 0] creates a list of squares for even numbers

Q15. Explain the difference between is and == in Python.

Answer: The == operator compares the values of two objects to see if they are equal (equality). The is operator compares the memory addresses of two objects to see if they are the exact same object in memory (identity). For example, two different lists with the same elements will return True for == but False for is.