Trade offs (10 Marks)
Explain atleast 5 trade offs we discussed in the class.

1. Distributed Computing vs Centralized Systems

Distributed Computing: Involves multiple independent systems working together.

Centralized Systems: All processes are managed by a single system.

Example: In a distributed computing setup, like a cloud-based file storage service such as Google Drive, multiple independent servers work together to store and manage data, ensuring scalability and fault tolerance. This means that as more users upload files, the system can handle the increased load by spreading it across several servers. In contrast, a centralized system, like an on-premise file server in a small office, relies on a single machine to store all the data. While easier to manage and secure, this setup can become a bottleneck when multiple users access the server simultaneously, leading to slower performance and a single point of failure.

2. Scaling Horizontally vs Vertically

Horizontal Scaling: Involves adding more machines (servers) to handle increased load, which can be more cost-effective and provide better fault tolerance.

Vertical Scaling: Involves upgrading a single machine (e.g., more CPU, memory). It's simpler and easier to implement, but it has limitations in terms of hardware capacity and cost efficiency.

Example: When a social media platform like Facebook experiences growth, it scales horizontally by adding more servers to distribute the increasing load of user posts, comments, and media. This method allows the system to scale flexibly as the user base grows. On the other hand, a small business website may scale vertically by upgrading a single server with more RAM or CPU power to handle increased traffic, but it’s limited in terms of how much it can grow and can become costlier over time compared to adding more servers.

3. Batch Processing vs Stream Processing

Batch Processing: Data is processed in large chunks or batches at scheduled intervals. It’s efficient for large volumes of data, but there's a delay between data generation and processing.

Stream Processing: Processes data in real-time as it is generated. While it’s great for timely insights and responses, it requires more resources and complexity to handle high volumes and ensure reliability.

Example: A bank’s end-of-day transaction processing system uses batch processing to process all the transactions from the day in one go overnight, updating account balances and generating reports. This is

efficient for large volumes of data but doesn't handle real-time requirements. On the other hand, a stock trading platform uses stream processing to monitor and react to market data in real-time, allowing traders to buy and sell stocks instantly based on live data, which requires more complex systems but provides immediate action.

4. Monolithic vs Microservices

Monolithic: All components of an application are tightly integrated into a single unit. It's easier to develop and test initially but becomes hard to scale, maintain, and update as the system grows.

Microservices: Breaks the application into smaller, independent services that can be developed, deployed, and scaled independently. This offers flexibility, scalability, and resilience but increases the complexity in terms of service communication, data consistency, and deployment.

Example: An e-commerce website might start as a monolithic application where the front-end, back-end, and database are all tightly coupled into a single codebase. This simplifies initial development but makes scaling and maintaining the system difficult as it grows. As the platform scales, it might transition to a microservices architecture, where different functions like payment processing, product catalog management, and user authentication are handled by separate services, allowing independent scaling and easier updates without affecting the entire system.

5. Programming Languages

Low-Level Languages : These languages give you fine-grained control over system resources and offer faster execution times but require more effort to develop and maintain due to their complexity and potential for bugs.

High-Level Languages : These languages are easier to use, more abstract, and reduce the development time but tend to be slower in execution compared to low-level languages due to their abstraction and reliance on runtime environments.

Example: A real-time embedded system used in an airplane's navigation system might be built with low-level languages like C or C++ to ensure high performance and low latency, allowing the system to respond instantly to changes in flight data. In contrast, a data analysis script used by a data scientist to process and visualize large datasets would likely be written in high-level languages like Python, which offers ease of development and readability, though it might be slower than the performance-oriented low-level approach.

Reference: Gradskey Zoom Meeting on 25th Jan

Search (15 Marks)

You have a huge data set and it is in unsorted order. Which search do you prefer ?

Hint : This is a trade off question

If you have a huge dataset that is unsorted, **Linear Search** is generally preferred because it doesn't require any preprocessing. It simply checks each element one by one until it finds the target or exhausts the dataset, making it a straightforward and direct approach with a time complexity of **O(n)**. However, if you were to sort the data first, you could use **Binary Search**, which is much faster with a time complexity of **O(log n)**. But sorting itself would take **O(n log n)** time, making it inefficient for a one-time search, especially with large unsorted datasets.

Thus, if you only need to search once, Linear Search is typically the better choice. However, if you plan to perform multiple searches on the same dataset, it may make sense to sort the data first and then apply Binary Search*for faster subsequent lookups, despite the initial sorting cost. It's a trade-off between initial sorting overhead and long-term search efficiency.

Return vs Yield (15 Marks)

Write a function which generates 100 random numbers. Use both return and yield, explain what you observe ?

```
import random

def generate_random_numbers():
    random_numbers = []
    for _ in range(100):
        random_numbers.append(random.randint(1, 100))
    return random_numbers
def generate_random_numbers_yield():
    for _ in range(100):
        yield random.randint(1, 100)
random_numbers = generate_random_numbers()
print(random_numbers[:10])
random_numbers_gen = generate_random_numbers_yield()
print([next(random_numbers_gen) for _ in range(10)])
```

When using return, all values are stored and returned as a list, which can consume a lot of memory for large datasets.

With yield, the values are produced one at a time and can be consumed without holding all values in memory at once, which makes it much more efficient for handling large streams of data.

MergeSort (25 Marks)

Use the above function and store the 100 numbers in a list.

- **Perform merge sort as usual**
- **Use Batch Processing we did in the above exercises**
- **Can you try to attempt MapReduce Paradigm for this ?**

```
import random
from functools import reduce


def generate_random_numbers():
    random_numbers = []
    for _ in range(100):
        random_numbers.append(random.randint(1, 100))
    return random_numbers


def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
```

```
    return merge(left, right)

def merge(left, right):
    sorted_arr = []
    while left and right:
        if left[0] < right[0]:
            sorted_arr.append(left.pop(0))
        else:
            sorted_arr.append(right.pop(0))
    sorted_arr.extend(left)
    sorted_arr.extend(right)
    return sorted_arr

def batch_process(arr, batch_size=25):
    batches = [arr[i:i + batch_size] for i in range(0, len(arr), batch_size)]
    sorted_batches = [merge_sort(batch) for batch in batches]
    return sorted_batches

def map_phase(arr, batch_size=25):
    batches = [arr[i:i + batch_size] for i in range(0, len(arr), batch_size)]
    return [merge_sort(batch) for batch in batches]

def reduce_phase(sorted_batches):
    return reduce(lambda x, y: merge(x, y), sorted_batches)
```

```
random_numbers = generate_random_numbers()

sorted_numbers = merge_sort(random_numbers)

batches = batch_process(random_numbers)

mapped_batches = map_phase(random_numbers)

sorted_numbers_mapreduce = reduce_phase(mapped_batches)


print("Unsorted numbers:", random_numbers[:10])

print("Sorted numbers:", sorted_numbers[:10])

print("Sorted batches:", batches[:2])

print("Sorted numbers (MapReduce):", sorted_numbers_mapreduce[:10])
```