Big Data Assignment

Q1) Explain at least 5 trade-offs we discussed in the class.

1. Trade-Offs (10 Marks)

In any decision, gaining one advantage usually comes at the cost of another. Here are five of them, along with my understanding of how they impact real-world applications:

1. Distributed Computing vs. Centralized Systems

In a **centralized system**, everything—computation and data storage—happens on a single machine or within a tightly managed setup. This makes things easier to handle, but there's a big catch: it can't scale well. As data and workload grow, the system struggles to keep up.

On the other hand, **distributed computing** spreads tasks across multiple machines (or nodes), making it more resilient to failures and capable of handling massive amounts of data. But it's not all smooth sailing—managing multiple machines brings challenges like keeping data consistent, ensuring smooth communication between nodes, and avoiding synchronization issues.

Example: Running everything on one computer (**centralized system**) is simple but struggles as workloads grow. Think of a small bakery tracking orders in one notebook—easy at first, but chaotic as orders pile up.

Big companies like Amazon use **distributed systems**, where multiple machines share the workload. If one fails, others take over. But keeping everything in sync—like inventory or customer orders—adds complexity. It's a trade-off between simplicity and scalability.

2. Scaling Vertically vs. Horizontally

When a system starts experiencing high traffic, we need to decide how to scale it.

- **Vertical scaling** means upgrading a single machine's resources (CPU, RAM, SSD). This is straightforward but has a limit—there's only so much we can upgrade before hitting hardware restrictions.
- **Horizontal scaling** involves adding more machines to share the load. While this approach provides infinite scalability, it requires a more complex architecture, including load balancers and distributed databases.

Example: Imagine running an online store. At first, upgrading a single server (vertical scaling) works, but there's a limit. When traffic grows too much, you switch to horizontal scaling—adding multiple servers to share the load.

This allows unlimited growth but requires extra setup, like load balancers and distributed databases.

It's like a restaurant:

Vertical scaling = Hiring a better chef.

Horizontal scaling = Opening more kitchens to serve more customers.

3. Batch Processing vs. Stream Processing

When dealing with Big Data, we need to decide how to process it:

Batch Processing: Data is collected over time and processed periodically. This is efficient for large volumes of data but results in delays (latency).

Stream Processing: Data is processed as it arrives, allowing real-time insights. However, this requires high computational power and is harder to implement efficiently.

Example:

Batch Processing Example**: A bank processes all its transactions overnight and updates account balances in bulk.**

Stream Processing Example**: A stock trading platform processes market data in real-time to update prices and execute trades instantly.**

4. Monolithic vs. Microservices Architecture

This trade-off is common in software development:

Monolithic Architecture**: In this setup, the entire application is built as one large unit. It's simpler to start with, but as the application grows, it becomes harder to manage and update.**

Microservices Architecture**: Here, the application is split into smaller, independent services that work together. This makes it easier to scale and update parts of the app independently, but it can get more complicated because the services need to communicate with each other.**

Example:

Monolithic Architecture Example**: A simple blogging app where all features (posts, comments, user login) are part of one system.**

Microservices Architecture Example**: An online store like Amazon, where user login, product catalog, and payment are separate services that work together.**

5. SQL vs. NoSQL Databases

Databases are crucial when handling Big Data, and deciding between SQL and NoSQL is an important choice:

SQL (Relational Databases)**: These databases offer strong consistency and are great for structured data with well-defined relationships. They also have powerful query capabilities. However, they can struggle when you need to scale up to handle large amounts of data.**

NoSQL (Non-relational Databases)**: NoSQL databases are more flexible, allowing for dynamic data structures and high scalability. They perform well in distributed systems and can handle large volumes of data. The trade-off is that they often don't guarantee the same level of consistency as SQL databases.**

Example:

SQL Example**: A library system where books, patrons, and loans are all related and require strict data consistency (using MySQL or PostgreSQL).**

NoSQL Example**: A messaging app like WhatsApp, where messages and user profiles change frequently and need to scale quickly (using MongoDB or Cassandra).**

Q2) You have a huge data set and it is in unsorted order. Which search do you prefer?

Hint: This is a trade-off question

Ans) Since the dataset is unsorted**, binary search is not an option. The best choice is** linear search**, as it does not require sorting. However, if sorting is feasible as a pre-processing step, binary search could be considered for repeated queries.**

The choice of search algorithm is a classic trade-off: do we focus on efficiency, or do we prioritize simplicity? Let's explore the possible options.

Q3) Write a function which generates 100 random numbers. Use both return and yield, explain what you observe?

Ans) import random

```
# Using return
def generate_numbers_return():
    numbers = [random.randint(1, 100) for _ in range(100)]
    return numbers  # Returns the entire list at once


# Using yield
def generate_numbers_yield():
    for _ in range(100):
        yield random.randint(1, 100)  # Yields one value at a time


# Observations:
# - `return` stores all numbers in memory before returning, leading to higher memory usage.
# - `yield` generates numbers one by one, which is more memory-efficient for large datasets.
```

Q4) 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 ?

Ans)

```
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])
```

Refrences – For first 3 questions I have referred from the pdf and classes.

For 4th question I have taken some help from the internet and applied here.