# 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 centralized systems, all the computation and data storage happen on a single machine or within a tightly controlled environment. This setup is simpler to manage but comes with major scalability limitations. On the other hand, distributed computing spreads tasks across multiple machines (nodes), improving fault tolerance and allowing systems to handle large-scale data. However, this introduces complexities like network communication, data consistency, and synchronization issues.

**Example:** Google's search engine wouldn't work efficiently if it relied on a single powerful server. Instead, it distributes the workload across thousands of machines to process web pages faster. However, this also requires robust distributed algorithms to manage data consistency across all nodes.

### 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:** A start-up might begin with vertical scaling (buying a more powerful server), but as its user base grows, it will eventually have to shift to horizontal scaling, adding multiple servers to handle traffic.

### 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:**

- A company analysing customer purchase trends at the end of each day might use batch processing.
- A stock market trading platform that needs real-time price updates must rely on stream processing.
-

**4. Monolithic vs. Microservices Architecture**

This trade-off is common in software development:

- **Monolithic Architecture**: The entire application is built as a single unit. It's easier to develop initially but becomes difficult to maintain as it grows.
- **Microservices Architecture**: The application is broken into independent services that communicate with each other. This improves scalability and flexibility but increases complexity due to inter-service communication.

**Example:** A small e-commerce website might start with a monolithic design. But as it expands, it may adopt microservices, separating components like payment processing, inventory management, and user authentication.

**5. SQL vs. NoSQL Databases**

Databases play a key role in Big Data, and choosing between SQL and NoSQL is a critical decision:

- **SQL (Relational Databases)**: Offer strong consistency, structured schemas, and powerful query capabilities. However, they struggle with high scalability.
- **NoSQL (Non-relational Databases)**: Provide flexible schemas, high scalability, and better performance for distributed systems, but they often sacrifice strict consistency.

**Example:**

- A banking system requires strong consistency, making SQL a better choice.
- A social media platform dealing with unstructured data (posts, comments, likes) benefits from NoSQL's scalability.

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
```

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])
```

Reference : - Q1,Q2,Q3 the reference is taken from the pdf provided and also the teaching from the Lokesh sir

About Q4 some part I have done and the rest I have taken help from internet after understanding clearly I have applied it.