**Assignment Name: Big data assignment 1**

**Q1) Explain atleast 5 trade offs we discussed in the class.**

1**.** Distributed Computing vs. Centralized Systems

* Distributed Computing: Multiple machines (nodes) work together to perform tasks, sharing workloads.
  + Pros: Scalability, fault tolerance, better performance under high loads.
  + Cons: Increased complexity, network latency, potential data inconsistency.
* Centralized Systems: A single system or server handles all tasks.
  + Pros: Easier to manage, lower latency (no network communication needed), strong consistency.
  + Cons: Limited scalability, single point of failure.

Example:

* A cloud-based database (e.g., Google Spanner) is a distributed system.
* A traditional relational database running on a single server (e.g., MySQL on a local machine) is centralized.

2. Scaling Vertically vs. Horizontally

* Vertical Scaling (Scaling Up): Increasing the power of a single machine (more CPU, RAM, etc.).
  + Pros: Simple, no changes in application architecture.
  + Cons: Expensive, limited by hardware capabilities.
* Horizontal Scaling (Scaling Out): Adding more machines to distribute the load.
  + Pros: More scalable, cost-effective in the long run.
  + Cons: More complex (requires load balancers, distributed databases, etc.).

Example:

* Vertical Scaling: Upgrading a server from 16GB RAM to 64GB RAM.
* Horizontal Scaling: Adding more servers and distributing requests using a load balancer.

3. Batch Processing vs. Stream Processing

* Batch Processing: Data is collected over time and processed in bulk.
  + Pros: Efficient for large datasets, optimized for throughput.
  + Cons: High latency, not suitable for real-time applications.
* Stream Processing: Data is processed in real time as it arrives.
  + Pros: Low latency, real-time insights.
  + Cons: More complex, may require fault-tolerant architecture.

Example:

* Batch Processing: Running a payroll system at the end of each month.
* Stream Processing: Fraud detection in financial transactions (analyzing transactions in real time).

4. Monolithic vs. Microservices

* Monolithic Architecture: A single, tightly integrated codebase.
  + Pros: Easier to develop and deploy initially, simpler debugging.
  + Cons: Hard to scale, difficult to update parts independently.
* Microservices Architecture: The application is broken down into smaller, independent services.
  + Pros: Easier to scale and maintain, independent updates.
  + Cons: More complex (requires APIs, service discovery, inter-service communication).

Example:

* Monolithic: A single Java Spring Boot application managing everything (user auth, payments, etc.).
* Microservices: Separate services for authentication, payments, and orders, communicating via APIs.

5. Inter-Service Communication (EDA vs. REST API)

* Event-Driven Architecture (EDA): Services communicate asynchronously by publishing/subscribing to events.
  + Pros: Scalable, loosely coupled, better fault tolerance.
  + Cons: Harder to debug, increased complexity.
* REST API: Services communicate via HTTP requests (synchronous).
  + Pros: Simpler to understand, widely used.
  + Cons: Tightly coupled, may lead to performance issues under heavy loads.

Example:

* EDA: A payment service publishes an event “Order Paid,” and a shipping service listens for that event to start shipping.
* REST API: A frontend calls the backend API to fetch user details synchronously.

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

**Ans)** When dealing with a huge unsorted dataset in the context of Big Data, the preferred search method depends on the structure and access patterns of the data. Here are the best options:

1. NoSQL-Based Search (Indexing for Fast Lookups)

* Use Case: If the data is stored in a NoSQL database (e.g., Elasticsearch, MongoDB, Cassandra), indexing can significantly speed up searches.
* Why? These databases use hash-based or tree-based indexing to optimize lookups.
* Example: Searching for a user's record in a MongoDB collection with an indexed field.

2. Parallel and Distributed Search (MapReduce & Spark)

* Use Case: If the dataset is massive and stored across multiple nodes (e.g., in HDFS or a cloud storage system), distributed computing is the best approach.
* Why? Apache Spark and Hadoop MapReduce distribute the search workload across multiple machines.
* Example: Running a query on an unindexed dataset stored in Amazon S3, HDFS, or a data lake.

3. Linear Search (if no indexing is available)

* Use Case: If the dataset is completely unstructured and does not have indexing or a distributed framework.
* Why? Sorting a huge dataset before searching may be expensive, so a sequential scan (brute force) may be the only option.
* Example: Searching for a specific record in raw log files.

Best Choice?

* If the dataset is stored in a NoSQL database → Use index-based search (like Elasticsearch).
* If the dataset is distributed (HDFS, S3, or cloud) → Use Apache Spark or MapReduce.
* If there is no structure and no indexing → Use linear search (full scan) as a last resort.

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

**A screenshot of a computer program

Description automatically generated**

**Observations**

1. **return returns the entire list at once**
   * The function generate\_numbers\_return() returns **all 100 numbers in one go**.
   * This consumes **more memory** because all numbers are stored in a list.
   * Useful when you need the entire dataset at once.
2. **yield generates numbers one at a time (lazy evaluation)**
   * The function generate\_numbers\_yield() returns a **generator object** instead of a list.
   * It **does not store all numbers in memory**; instead, it generates them on demand.
   * More **memory efficient** and useful for large datasets.

**Key Takeaways**

* **Use return** when you need all values at once.
* **Use yield** when working with large data streams to save memory.
* Generators (yield) are **lazy** and efficient for handling Big Data processing.

**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)** Observations

1. Merge Sort (Regular)
   * Works well but processes all data at once.
   * Can be slow for large datasets.
2. Batch Processing
   * Processes data in smaller chunks, making it more manageable.
   * Suitable when memory is a constraint.
3. MapReduce
   * Uses parallel computing to speed up sorting.
   * Efficient for large-scale distributed sorting (e.g., sorting big data in Hadoop/Spark).

Conclusion

* If dealing with a small dataset, regular merge sort is fine.
* For medium datasets, batch processing optimizes memory usage.
* For huge datasets (Big Data), MapReduce is the best choice as it distributes the workload.

**GitHub Link For all coding part and question**

* [**https://github.com/Abhinavkrtiwari/Big\_Data\_Assignment**](https://github.com/Abhinavkrtiwari/Big_Data_Assignment)