Question 1:

1.1. Traditionally, transferring data from the disk to main memory was the major bottleneck in computing. Now, with memory data processing, what is the new bottleneck?

Ans 1.1

Memory access can become a bottleneck: CPU waiting to process data in main memory becomes major bottleneck.As physical memory(also called main memory) is slower than the speed of CPU cache, hence the program can only be executed as fast as the main memory is accessed.When the CPU has to wait for data to be fetched from the slower RAM, it causes a bottleneck. This waiting time affects the overall speed at which computations can be performed, limiting the system's performance.

1.2. Garbage collection has the potential to substantially impact the performance of a system. What is one workaround to reduce the overhead of garbage collection?

Ans 1.2.

1) **Reuse Objects:** By reusing objects through object pooling can avoid the object getting GCed. Instead of a new object getting created every time, the objects are reused from the pool of objects. By doing this, the object becomes old and therefore is not collected as garbage.
2) **Use Primitive Data Types (Java):** By using primitive data types, we can avoid the creation of objects. For eg. using int instead of Integer or int[] instead of ArrayList<Integer>, can reduce the number of objects created. Because overhead created by primitive data types is less than their objects counterparts, this could be a good workaround to reduce overhead for GC.
3) **Specialized Libraries (fastutil, koloboke, ...):** We can obtain more memory-efficient data structures helpful for specific use cases by using specialised libraries such as fastutil or koloboke. These libraries frequently optimise memory consumption and offer alternatives to conventional Java collections, hence reducing the number of objects created and perhaps minimising garbage collection overhead.

1.3. Why is parallelism more important now than it was 30 years ago?

Ans 1.3.

As the two major “laws” are collapsing, parallelism is gaining more prominence:

1. Moore’s law
2. Dennard scaling

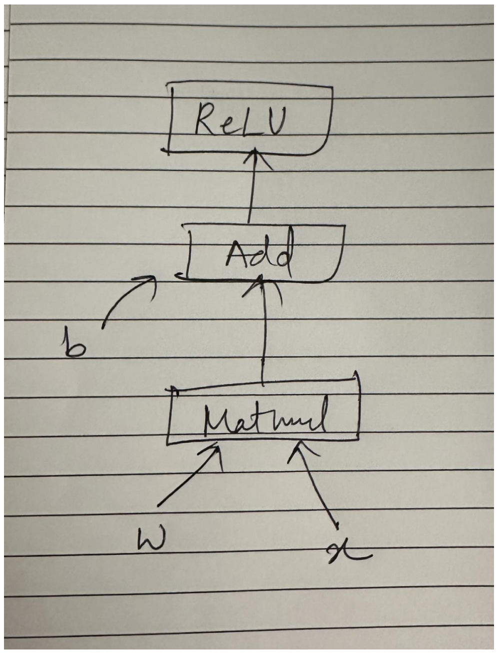
1) **Stagnation of Maximum Clock Rate**: The maximum clock rate of processors has stopped increasing significantly, limiting the performance improvements that can be achieved through higher clock speeds.
2) **End of Moore's Law and Dennard Scaling** : The slowing of Moore's Law and the end of Dennard Scaling indicate we are reaching the physical limits of transistor miniaturization and power efficiency in traditional single-core processors.
3) **Rise of Multi-Core Processors**: To overcome these limitations, there has been a shift towards multi-core architectures. By using multiple cores, computers can perform more tasks simultaneously, offering a new pathway for performance improvement.

These factors collectively underscore why parallelism in computing has gained significantly more importance today compared to the computing landscape 30 years ago.

Question 4:

4.1. What is the key idea of TensorFlow?

Ans 4.1.



TensorFlow is an end to end open source Machine learning platform for everyone. The keys ideas of Tensorflow are:

Computational Graph: The primary feature of TensorFlow is its ability to express computations using a computational graph. The mathematical processes represented by nodes in this graph are connected by edges, which show the tensor flow between them. This graph-based method offers a productive and adaptable way to specify and arrange intricate calculations.

Variable Operators: Stateful nodes in the computational graph are called variable operators. They don't change while the graph is being executed. Variable operators such as W and b are examples of parameters. These parameters output their current values, which are learned throughout the training process.

Placeholders: In the computational graph, placeholders are nodes that act as inputs when the calculation is being done. They permit the dynamic input value feeding. For example, 'x' can be used as a placeholder for input data. Several datasets can be fed into the same computational graph structure by using placeholders.

Mathematical Operators: Numerous mathematical operators are available in TensorFlow, and they can be applied to variables and placeholders in the computational graph. Matrix multiplication (matmul), Add, and Rectified Linear Unit (Relu) are a few examples. The mathematical transformations that are done to the data as it moves across the graph are specified by these operators.

Formula :
H = ReLU(Wx + b)

4.2. What is the execution style of TensorFlow 2.x: eager or lazy? Why is this a benefit?

Ans 4.2.
TensorFlow 2.x follows the Eager Execution style. This is a departure from the previous approach, where static computational graphs were built before execution. Now the operations are done immediately without building static computational graphs.

This is a benefit due to several reasons:

Easier Debugging
Eager execution facilitates faster debugging by executing actions instantly. It is easier to find and track errors step-by-step when developers are able to print and examine intermediate values at various points during the computing process.

Compatibility with Python Ecosystem
TensorFlow's eager execution connects it more closely with the larger Python environment. It becomes more accessible and usable in a manner comparable to other Python libraries since operations are executed quickly. This interoperability facilitates TensorFlow integration into current Python workflows and promotes a more intuitive coding experience.

In conclusion, TensorFlow's adoption of eager execution improves the user experience by providing immediate feedback during code execution, improving compatibility with the Python ecosystem, and making TensorFlow more accessible and intuitive for a broader range of users, bridging the gap with other popular deep learning frameworks such as PyTorch.

4.3. What are Tensors in the TensorFlow context? Are Tensors dense or sparse?

Ans 4.3.
Tensors are the edges in the computation graph of operations in TensorFlow. They act as computation inputs and outputs, representing data flowing between graph nodes. Tensors are n-dimensional arrays that can carry a variety of primitive data types such as integers, floating-point numbers, and byte arrays. Tensors may represent a wide range of data, from simple numerical values to more complicated structures, thanks to their adaptability.

TensorFlow operations can be done on tensors to build new tensors, resulting in a dynamic and flexible system for designing and executing computations.

Tensors in TensorFlow are typically dense, which means that all of the tensor's elements are explicitly recorded in memory. This feature ensures that the value of each element is saved, making it easy to access and change the data. However, while TensorFlow tensors are dense by default, users must develop efficient ways to represent and operate with sparse data when necessary.

Stating the properties of the dense tensors:

- In memory, all elements are explicitly represented.
- Allows direct access to the value of each element.
- Suitable for instances when the majority of the elements have non-zero values.

2.1. What are the three design goals of the Google File System?

Ans 2.1.
The three primary design goals of the Google File System (GFS) are:

1) **Handling Huge Files**: GFS is designed to handle large files, which can span numerous servers. To do this, GFS employs a coarse granularity for blocks in order to keep metadata reasonable. GFS can efficiently transport and manage data across various servers in the system by splitting huge files into smaller, manageable blocks.

2) **Reduces failure points:** GFS allows files to span across multiple servers. By distributing files over different servers, Google File System (GFS) reduces failure spots. A single master server controls metadata, and if it fails, a backup master takes over

immediately, recreating metadata for smooth continuity. This design improves system fault tolerance and robustness.

3) **Workload :** GFS is designed to handle a specific workload pattern that includes concurrent append-only writes and primarily sequential reads. Because of the nature of web crawling, this task is frequent in a search engine context. Web crawling involves adding new content to an index on a continual basis, whereas batch processing and sequential scans are employed to develop and update the search index.

2.2. Why is a workload of concurrent append-only writes and mostly sequential reads common in a search engine?

Ans 2.2.
In a search engine, the common workload of concurrent append-only writes and mostly sequential reads is driven by the nature of web crawling and search index creation.
Web crawling involves regularly adding new content to an existing dataset, allowing for real-time updates. Concurrent writes are distributed properly across various nodes for aiding scalability.
To efficiently create or update the search index, a batch processing approach is employed, where mostly sequential reads are performed. This design is specifically tailored to the search engine's need for dynamically incorporating new information through concurrent writes during web crawling while optimizing the efficiency of creating and updating the search index through sequential reads during batch processing. Essentially, this workload structure aligns with the demands of a search engine, allowing it to keep pace with the ever-evolving content on the web and deliver fast, relevant results to users.

2.3. What data is communicated between the GFS client and the GFS chunkserver?

Ans 2.3.

In the GFS system, the data communicated between the GFS client and the GFS chunkserver is in the following way:
From GFS Client to GFS Chunkserver:

- Chunk Data: When writing, the client transmits to the chunkserver the actual data to be saved in a chunk.
- Chunk Handle and Byte Range: The client sends a request to the chunkserver that includes the chunk handle (a unique identifier for the chunk) and the byte range for the operation for both read and write operations

From GFS Chunkserver to GFS Client:

- It may also send acknowledgments or error messages for write operations.

Although it is not directly involved in the data flow between the client and chunkserver, the GFS master plays an important role in this process. After the client asks access to a file, the master delivers the chunk handle and locations, allowing the client to interface directly with the chunkserver for real data transmission.

Question 3:

3.1. What is the limitation of MapReduce that Spark aims to address?

Ans 3.1.
Spark aims to address several limitations of MapReduce, including:

1. **Stragglers and Global Synchronization Barrier:** The global synchronisation barrier in MapReduce might result in stragglers, causing job completion delays. Spark attempts to address this by implementing more effective scheduling techniques and reducing the impact of stragglers on overall performance.

2. **Iterative Workloads and Multiple Map-Reduce Iterations:** MapReduce is not well-suited for iterative workloads involving several map-reduce iterations, such as machine learning algorithms. Spark is designed to effortlessly handle iterative workloads, resulting in a more intuitive and efficient environment for iterative calculations.

3. **State Preservation Across Iterations:** In MapReduce, preserving state across iterations can be challenging. Spark addresses this limitation by allowing the efficient reuse of intermediate results across multiple iterations, eliminating the need to write intermediate data to disk and enhancing performance for iterative algorithms.

4. **Expensive Writing of Intermediate Results to Disk:** The expense of writing intermediate results to disc, which can be computationally expensive, is frequently incurred by MapReduce. Spark optimises this process by allowing the reuse of intermediate results without the need for disc writing, decreasing I/O cost and enhancing efficiency.

5. **Low-Level Programming Interface:** Because MapReduce requires a low-level programming interface, it is difficult for developers to explain certain computations succinctly. Spark offers a higher-level programming interface that simplifies development and improves the expressiveness of complex computations.

In order to achieve these goals, Spark includes capabilities such as iterative task support, the ability to reuse intermediate results without disc I/O, lineage-based fault tolerance, and a programming interface that is more developer-friendly than MapReduce's low-level nature. With these enhancements, Spark becomes a more versatile and efficient framework for a wider range of data processing applications.

3.2. Narrow and wide dependency in Spark. How do narrow dependencies generally perform compared to wide dependencies? What is one example of each type of dependency?
Ans 3.2.
Dependencies between transformations are classified as either narrow or wide in Apache Spark, and this classification has a substantial impact on how Spark executes these transformations.

1) **Narrow Dependency:** In a narrow dependency, each partition of the parent RDD (Resilient Distributed Dataset) is used by at most one partition of the child RDD.
   a) **Performance**: Because minimal dependencies enable Spark to execute transformations on the same partition (or thread), they can be pipelined locally, resulting in improved performance by reducing data shuffling.
   b) **Example**: A sequence of'map' transformations followed by 'filter' transformations is an example of a limited dependency. The output of the'map' transformation is sent straight into the 'filter' transformation in this case, and both can occur in the same stage without the need for a shuffle.
2) **Wide Dependency:** In a wide dependency, multiple parent partitions contribute to at least one child partition.
   a) **Performance**: Wide dependencies are less efficient compared to narrow dependencies. They mark the end of a stage in Spark's computational pipeline and require a shuffle of the data across the network.
   b) **Example**: A'map' transformation followed by a 'groupByKey' operation is a classic example of a wide dependency (provided the data isn't partitioned by the key used in 'groupByKey'). The 'groupByKey' function requires aggregating data with the same key over many partitions, resulting in a shuffle.

In short, limited dependencies are often more performant because they allow transformations to be pipelined locally without the requirement for data moving across partitions. While wide dependencies are necessary for certain sorts of data processing jobs, they bring additional complexity due to the required shuffle operations.

3.3. How is an RDD preserved for short-running jobs?

Ans 3.3.

RDDs (Resilient Distributed Datasets) are a fundamental data structure that was created with fault tolerance in mind. They can be preserved in a variety of methods, including:

1) **Fault tolerance** is essential in all vocations. Lineage information is used by RDDs to achieve fault tolerance. If an RDD partition is lost, it can be recomputed using this

lineage information instead of replicating the data over many nodes. This is not the case in long-running jobs because lineage is expensive, and they are used for checkpointing.

2) **RDDs are mostly kept in memory**. This enables for quick access and processing, which is very useful for short-run activities.
3) **We can cache the RDD** on RAM or disc in Spark to give faster access.
4) **RDDs are evaluated lazily** in Spark. This means they are not computed until an action (such as 'collect', 'count', and so on) is done on them. This helps to optimise the overall execution plan and reduces superfluous computation for short-running activities.

5.1. Explain the difference between training and inference? This can be done in one short sentence each.

Ans 5.1.
Training:
The phase in which a machine learning model learns from provided data. It involves modifying the model's parameters to reduce the gap between the model's predictions and the actual results.
This step is computationally intensive because it involves processing big datasets and updating the model's parameters periodically. Because the primary purpose is to learn from data rather than make instant predictions, it often has more flexible performance requirements.

Inference:
Inference is the process of utilising a trained model to generate predictions or judgements based on previously unknown data. The learned parameters of the model are used to create predictions about this data.
Inference is typically less computationally demanding than training. Because it is frequently employed in real-world applications where timely and precise forecasts are crucial, it has tougher performance requirements, notably in terms of speed and accuracy.

5.2. Why would a model be quantized before serving it?

Ans 5.2.
We aim to optimise the models to reduce latency and inference costs so that they may be deployed on low-resource devices. One of the model compression strategies is quantization. Prior to serving the model, quantization is performed to optimise the model's performance and efficiency. Here are the main reasons for doing so.

Quantization is the process of reducing the amount of bits used to express a number.

32 floating point → 8 bit integer
As a result, the model size is lowered. A smaller model requires less storage space and is easier to load and distribute, which is especially useful when resources are limited.

In addition, because integer computations are faster than floating point computations, the model is optimised, becomes less complex, and can run faster. Because lower-precision procedures are less computationally expensive, more operations can be performed in the same amount of time. This acceleration is crucial for real-time applications such as autonomous vehicles or augmented reality, where input must be processed quickly.

Quantized models consume less energy, which is advantageous for deployment on battery-powered devices such as smartphones and IoT devices. Reducing computing resources not only saves battery life but also helps the gadget last longer.

5.3. What are two challenges of serving a model on the edge instead of in the cloud?

Ans 5.3.
The two challenges of serving a model on the edge instead of in the cloud are:

1) **Limited Memory and Computing Power:** Edge devices encounter hurdles when deploying complicated machine learning models due to their limited processing power, memory, and storage when compared to cloud servers. It is critical for efficient edge deployment to strike a balance between model complexity and available resources.
2) **Energy Consumption:** Energy efficiency is a critical issue. Running computationally intensive models can result in quick battery depletion, which is undesirable for devices that are anticipated to operate for lengthy periods of time without being recharged frequently. Managing machine learning models' energy consumption while preserving performance becomes a critical challenge in edge deployment.Model pruning, the use of lightweight model structures, and the optimisation of computing activities are all necessary to assure the model's sustainability in energy-constrained contexts.