BT:

1) a) The main advantages: Improve access speed, Enhance reliability, Load balancing
b) The potential drawbacks: Data consistency issues, Increased storage costs, Increased management complexity, Security risks
2)a) The main disadvantage is the high network transmission overhead. Transferring a very large file from remote node B to node A can be significant and can take a long time, especially if the network is slow or unstable, which can lead to inefficiencies in the process. Moreover, if a network failure occurs during transmission, it may need to be retransmitted, further increasing the time cost and uncertainty.
b) A better approach is to move the processing as close to the data as possible, i.e. to filter on node B where the file is stored. You can run a filter on Node B to transfer the filtered results back to Node A. Doing so reduces the amount of data transferred, as only the filtered results need to be transferred, not the entire large file. If the amount of filtered result data is relatively small, the network transmission time will be greatly reduced and the overall efficiency will be significantly improved.
c) Data locality refers to the tendency of a program to access data during the execution of a program in a computer system, that is, a tendency to access adjacent data. It is mainly divided into temporal locality and spatial locality. Temporal locality means that if a data item is accessed, it is likely to be accessed again in the near future. For example, in a loop, loop variables and array elements are accessed repeatedly with each iteration of the loop. Spatial locality means that if a data item is accessed, then the data item adjacent to its address is likely to be accessed in the near future. This is usually because the program stores and accesses data sequentially in memory, for example, array elements are stored contiguous in memory, and when an array element is accessed, it is likely that its neighbors will be accessed. In system design and optimization, data locality can effectively reduce data access latency and improve system performance.
3) i) Sequential Processing refers to a processing method in which a computer executes tasks one after another in the order of instructions. In this mode, tasks are processed one by one, and the next task will only start after the previous one is completed. For example, in a simple program with three steps: reading a file, processing data, and saving the results. If it is sequential processing, the program will first complete the operation of reading the file, then start processing the data, and only after the data processing is completed will it proceed to save the results. The advantages of sequential processing are its simple logic, ease of understanding and implementation, and it is suitable for simple tasks or scenarios where real-time requirements are not high. However, its efficiency may be limited when dealing with complex and large tasks because it cannot utilize multiple computing resources simultaneously to accelerate the processing speed.
ii) Parallel / Distributed Processing refers to a computing method that uses multiple processors or computing units to process different parts of a task simultaneously, thereby increasing the processing speed. These multiple processors can be in the same computer (such as a multi-core processor) or distributed on different computers (connected via a network), and the latter is also known as distributed processing. Taking image processing as an example, a high-resolution image can be divided into multiple small pieces, and each small

piece can be assigned to different processors for processing, such as edge detection, color correction, etc. Parallel / Distributed processing can significantly reduce the time required to process large tasks and make full use of the system's computing resources. However, it also faces some challenges, such as the rationality of task partitioning, communication overhead between processors, and maintaining data consistency.

4)a）Calculating a+b and calculating c/d can be executed in parallel.

Calculating e*f and calculating g/h can be executed in parallel.

b) Yes, the limitations include: Resource competition, Synchronization issues and Data dependency

c) Limitations of executing sections on different computer nodes: Network latency, Communication overhead, Data consistency and Fault handling

5)1. Data division

Divide a large file into 50 smaller parts so that each computer node can work on one of them. It can be evenly divided based on file size and number of nodes, ensuring that each node processes approximately the same amount of data. For example, if the total file size is 1GB, you can roughly divide it into about 20MB in size for each part, and distribute it to individual nodes.

2. Node processing

Each computer node independently processes the portion of the file assigned to itself, determining the maximum number in that section. You can use a simple traversal algorithm to compare each number in turn, recording the largest number you have encountered.

3. Summary of results

Round 1 Summary:

A node can be randomly selected as the master, and the other nodes send the maximum number of their own processing parts to the master.

The master node receives the maximum number sent by each node and compares it to record the maximum value.

The second round of summary (optional, decide whether to need it based on the actual situation):

If the results after the first round of aggregation are still large, they can be summarized again for the sake of accuracy and reliability.

The maximum number after the first round of aggregation is sent to a subset of nodes, which again compare and select the maximum value, and then return the result to the master node.

4. The final result

The final number given by the master node is the largest number in the entire file.

This distributed/parallel approach can greatly improve the efficiency of processing large files, reduce processing time, and make full use of the computing resources of multiple computer nodes. However, it is also necessary to consider issues such as network communication overhead and node fault handling.

6)1. Data and the nature of the task

Data type: In the example of determining the maximum number of a file, the data is homogeneous, all numeric and relatively single. For the example of a mathematical formula, the data is made up of different variables, and there may be different data types involved in

the operation.

Task complexity: The task of determining the maximum number of files is relatively simple and straightforward, with the goal of finding a single maximum value. The task of working with mathematical formulas is more complex and involves several different arithmetic operations that need to be coordinated and combined to arrive at the final result.

2.the processing steps

Data Division: When determining the maximum number of files, the main thing is to divide the large file evenly into several smaller parts and distribute them to different computer nodes for processing.

For mathematical formulas, the partition may be based more on the logical structure of the operation, determining which parts can be computed independently and in parallel.

Node processing content: In the example of the largest number of files, each node independently processes the part of the file to which it is assigned, finding the local maximum.

For mathematical formulas, each node is responsible for a specific part of the operation, such as calculating addition, multiplication, or division.

The results are summarized as follows: In the example of determining the maximum number of files, the local maximum value of each node is gradually summarized to the global maximum value through multiple rounds of result summarization.

For mathematical formulas, the results need to be combined and summarized according to the order and logical relationship of the operations, and a more complex synchronization mechanism may be required.

3. Communication and synchronization

Communication Frequency and Content: In the example of determining the maximum number of a file, the communication occurs mainly in the result aggregation phase, where the node sends a local maximum to the central node or other nodes, and the communication frequency is relatively low.

For the example of mathematical formulas, due to the complex data dependencies that may exist between the various parts of the operation, the communication may be more frequent, and intermediate results and control information need to be passed between different nodes.

Synchronization Requirements: Synchronization is relatively simple in the task of determining the maximum number of files, which is to ensure that the results are summarized at the right point in time to find the global maximum.

In the task of working with mathematical formulas, synchronization is more demanding, and it is necessary to ensure that the different parts of the operation are executed in the correct order to ensure the correctness of the final result.

4.Complexity

Overall Complexity: The method of determining the maximum number of files is usually simpler because of the relatively homogeneous tasks and data types.

Distributed/parallel methods for processing mathematical formulas are often more complex due to the complexity of the task and the diversity of the data.

Difficulty of troubleshooting: In the case of the largest number of files, a node failure may be relatively easy to deal with, requiring only the reassignment of tasks for that node or inference from the results of other nodes.

For examples of mathematical formulas, the failure of one node may have a greater impact

on the entire calculation process, requiring more complex fault recovery mechanisms to ensure the correctness of the results.

MT:

7) a) The NameNode is the master node in HDFS. It manages the file system namespace and keeps track of the metadata of all files and directories in the HDFS. It knows which blocks of a file are stored on which DataNodes. The DataNodes are the slave nodes that store the actual data blocks. They are responsible for serving read and write requests from clients. DataNodes report back to the NameNode periodically with information about their storage status and block lists.

In summary, the NameNode manages the metadata and makes decisions about data storage and access, while the DataNodes store the actual data and follow the instructions from the NameNode.

b)

1.Given a file size of 180MB and a block size of 64MB. The file can be divided into three splits: S1 of size 64MB, S2 of size 64MB, and S3 of size 52MB.

2.In HDFS, by default, a split (or block) has three replicas.

Replication is achieved by the NameNode instructing DataNodes to copy blocks to other DataNodes. When a client writes a file, the NameNode determines a set of DataNodes where the blocks should be placed. The first DataNode starts receiving the data and then writes a copy to another DataNode. That DataNode in turn writes a copy to yet another DataNode, ensuring that there are three copies of each block distributed across different DataNodes and racks to provide fault tolerance and high availability.

3.Let's assume split S1 is stored in Node1, Node3, and Node6. This configuration indicates that these three nodes have a copy of split S1.

4.When Node5 crashes and it was storing split S1 along with Node3 and Node7:

The reason NameNode needs to deal with this situation is to maintain the required level of redundancy and ensure data availability.

The NameNode detects the failure of Node5 through periodic heartbeats from the DataNodes. Once it detects the failure, it initiates the replication process. It selects a new set of DataNodes that have sufficient storage space and are not on the same rack as the existing copies (to ensure better fault tolerance). The NameNode instructs these new DataNodes to copy the block from the existing healthy replicas (in this case, from Node3 and Node7). This way, the system maintains the required number of replicas and ensures that the data remains available even in the face of node failures.

AT:

8) a) The Job Tracker is the master node in the Hadoop MapReduce framework and is responsible for coordinating and managing the execution of the entire MapReduce job. The Task Tracker is a slave node that runs on each data node and is responsible for executing the specific tasks assigned by the Job Tracker.

The Job Tracker receives MapReduce jobs submitted by clients, decomposes the jobs into multiple map tasks and reduce tasks, and assigns these tasks to available Task Trackers. The

Task Tracker regularly sends heartbeat messages to the Job Tracker to report its own status and task execution progress. The Job Tracker decides whether to reassign tasks or perform fault recovery based on the status information of the Task Tracker.

b) **Function of the map function:**

The map function is the core function of the mapping stage in the MapReduce framework. It takes a fragment of input data (such as a line of a file or a data block) as input.

The main task of the map function is to process the input data and generate a series of intermediate key-value pairs. For example, for a text processing task, the map function may split each line of text into words and output key-value pairs like (word, 1), indicating that this word appears once.

The map function usually executes in parallel on multiple nodes to improve processing efficiency.

Function of the reduce function:

The reduce function is the core function of the reduction stage in the MapReduce framework. It takes the intermediate key-value pairs generated by the map function as input.

The main task of the reduce function is to merge and process the intermediate values with the same key to generate the final output result. For example, for a word counting task, the reduce function will receive all key-value pairs with the same word, sum the values, and obtain the total number of occurrences of this word in the entire dataset.

The reduce function usually executes on one or several nodes to summarize and integrate intermediate results.

c) **Mapper class**:

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class ColorCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String line = value.toString();
        String[] colors = line.split(" ");
        for (String color : colors) {
            context.write(new Text(color), new IntWritable(1));
        }
    }
}
```

Reducer class:

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.Reducer;

public class ColorCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

	@Override
	protected void reduce(Text key, Iterable<IntWritable> values, Context context)
			throws IOException, InterruptedException {
		int sum = 0;
		for (IntWritable value : values) {
			sum += value.get();
		}
		context.write(key, new IntWritable(sum));
	}
}
```

Driver class:

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class ColorCountDriver {

	public static void main(String[] args) throws Exception {
		if (args.length!= 2) {
			System.err.println("Usage: ColorCount <input path> <output path>");
			System.exit(-1);
		}

		Job job = Job.getInstance();
		job.setJarByClass(ColorCountDriver.class);
		job.setJobName("Color Count");

		FileInputFormat.addInputPath(job, new Path(args[0]));
		FileOutputFormat.setOutputPath(job, new Path(args[1]));

		job.setMapperClass(ColorCountMapper.class);
		job.setReducerClass(ColorCountReducer.class);

		job.setOutputKeyClass(Text.class);
		job.setOutputValueClass(IntWritable.class);

		System.exit(job.waitForCompletion(true)? 0 : 1);
```

```
    }
}
```

9) Mapper class:

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class EquipmentQuantityMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String orderData = value.toString();
        String[] orders = orderData.split("_id:");
        for (String order : orders) {
            if (!order.isEmpty()) {
                String[] fields = order.split(",");
                for (int i = 0; i < fields.length; i++) {
                    if (fields[i].contains("equip_name")) {
                        String equipName = fields[i].split(":")[1].trim().replaceAll("\"", "");
                        int j = i + 1;
                        while (!fields[j].contains("equip_name")) {
                            if (fields[j].contains("qty")) {
                                int quantity = Integer.parseInt(fields[j].split(":")[1].trim().replaceAll("\"", ""));
                                context.write(new Text(equipName), new IntWritable(quantity));
                                break;
                            }
                            j++;
                        }
                    }
                }
            }
        }
    }
}
```

Reducer class:

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.Reducer;

public class EquipmentQuantityReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{

	@Override
	protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
		int sum = 0;
		for (IntWritable value : values) {
			sum += value.get();
		}
		context.write(key, new IntWritable(sum));
	}
}
```

Driver class:

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class EquipmentQuantityDriver {

	public static void main(String[] args) throws Exception {
		if (args.length!= 2) {
			System.err.println("Usage: EquipmentQuantity <input path> <output path>");
			System.exit(-1);
		}

		Job job = Job.getInstance();
		job.setJarByClass(EquipmentQuantityDriver.class);
		job.setJobName("Equipment Quantity");

		FileInputFormat.addInputPath(job, new Path(args[0]));
		FileOutputFormat.setOutputPath(job, new Path(args[1]));

		job.setMapperClass(EquipmentQuantityMapper.class);
		job.setReducerClass(EquipmentQuantityReducer.class);

		job.setOutputKeyClass(Text.class);
		job.setOutputValueClass(IntWritable.class);
```

```
        System.exit(job.waitForCompletion(true)? 0 : 1);
    }
}
```

10) Mapper class:

```
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class StreamingMapper extends Mapper<LongWritable, Text, Text, Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        String line = value.toString();
        String[] parts = line.split("\\|");
        String accountNumber = parts[0].trim();
        String programName = parts[1].trim();
        context.write(new Text(programName), new Text(accountNumber));
    }
}
```

Reducer class:

```
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class StreamingReducer extends Reducer<Text, Text, Text, Text> {

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
        Set<String> uniqueAccounts = new HashSet<>();
        for (Text value : values) {
            uniqueAccounts.add(value.toString());
        }
        int streamCount = uniqueAccounts.size();
        double pricePerStream = extractPricePerStream(key.toString());
        double totalPayment = streamCount * pricePerStream;
        context.write(key, new Text(streamCount + " times streamed. Total payment: $" +
totalPayment));
    }
```

```
    private double extractPricePerStream(String programName) {
        // The price is hard-coded according to the program name, and the price information should be obtained from the configuration file or database in practice
        if (programName.equals("Aladdin")) return 0.05;
        else if (programName.equals("Game of Thrones S 1 Ep 2")) return 0.07;
        else if (programName.equals("Mick S 1 Ep 5")) return 0.02;
        else if (programName.equals("Avengers End Game")) return 0.1;
        else return 0;
    }
}
```

Driver class:

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class StreamingDriver {

    public static void main(String[] args) throws Exception {
        if (args.length!= 2) {
            System.err.println("Usage: StreamingDriver <input path> <output path>");
            System.exit(-1);
        }

        Job job = Job.getInstance();
        job.setJarByClass(StreamingDriver.class);
        job.setJobName("Streaming Service Calculation");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(StreamingMapper.class);
        job.setReducerClass(StreamingReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        System.exit(job.waitForCompletion(true)? 0 : 1);
    }
}
```