MapReduce Algorithm Assignment

Solution of Q1:

```
Map(string TableName, string record)
{
   // Split the record by spaces (handle multiple spaces)
   string[] fields = record.split("\\s+");


   if (TableName.equals("Table1")) {
      string studentId = fields[0];
      string studentName = fields[1];
      string major = fields[2];

      // emit studentId as key, and "Table1" + student details as value
      emit(studentId, "Table1," + studentName + "," + major);
   }
   else {
      // Second table: Student Address, Student Id, Year
      string address = fields[0];
      string studentId = fields[1];
      string year = fields[2];

      // Emit studentId as key, and "Table2" + address and year as value
      emit(studentId, "Table2," + address + "," + year);
   }
}

Reduce(string studentId, List<string> records)
{
   string studentName = " ";
   string major = " ";
   string address = " ";
   string year = " ";

   // Iterate through records to separate data from Table1 and Table2
   for (string record : records) {
      string[] fields = record.split(",");

      if (fields[0].equals("Table1")) {
         studentName = fields[1];
         major = fields[2];
      }
      else if (fields[0].equals("Table2")) {
            address = fields[1];
            year = fields[2];
        }
    }

    // Emit the joined result
    emit(studentId, studentId + " " + studentName + " " + major + " " + address + " " + year);
}
```

Scalability:

The algorithm will scale when I use clusters with 2 compute nodes and 4 compute nodes.

- **With 2 Compute Nodes**:
  Scalability is limited because each node must handle 50 GB of input data (assuming the two tables are each 50 GB). The higher per-node workload (joining) may lead to longer execution times due to increased network traffic and processing demands.

- **With 4 Compute Nodes**:
  The algorithm will scale better as the per-node workload is reduced to 25 GB. Network traffic and processing demands are more evenly distributed across nodes, leading to faster execution.

In general, scaling improves with more nodes, as the workload is evenly distributed, especially when keys are unique.

Description of how a MapReduce implementation will execute the algorithm:

The MapReduce implementation for joining two large tables on studentId proceeds as follows:

1. **Input Splitting**:
   - The 100 GB data (50 GB per table) is divided into chunks (e.g., 64 MB) and distributed to mappers.
2. **Map Phase**:
   - Each mapper processes its chunk, splits records, and emits intermediate key-value pairs:
     - **Key**: studentId
     - **Value**: Record details prefixed with the table name (e.g.,"Table1,Tom,CS" or "Table2,Chicago,3rd").
3. **Shuffle and Sort Phase**:
   - Intermediate pairs are shuffled, grouped by studentId, and sorted for reducer input.Example:
   - Mapper 1 emits: (112, "Table1,Tom,CS") ,(124, "Table1,John,Math")

- Mapper 2 emits:
  (124, "Table2,Milwaukee,1st"), (134, "Table2,Chicago,3rd").

After shuffling, Reducer 1 gets:
Key = 124, Values = ["Table1,John,Math", " Table2,Milwaukee,1st "].

Then, keys (studentId) are sorted to facilitate efficient processing in the reduce phase.

4. **Reduce Phase**:
   - Reducers process grouped keys, join records from both tables, and emit the final joined results:
     - Output: studentId, studentName, major, address, year.

Solution to Question 2:

```
// Mapper function to emit minimum vote count per state
Map(string lineno, string record)
{
   // Split the record by commas
   string[] fields = record.split(",");

   // Extract Party_Name and State fields
   string partyName = fields[1].trim();
   string state = fields[3].trim();

   // Check if the party is Democratic
   if (partyName.equals("Democratic")) {
      emit(state, 1);  // Emit state as key with 1 as the vote count
   }
}

// Reducer function to find vote count across states
Reduce(string state, List<integer> counts)
{
   int totalVotes = sum(counts);  // Sum up all vote counts for the state
   emit( state , totalVotes);    // Emit the total votes for Democratic Party in each state
}
```

Solution to Question 3:

```
// Mapper function
Map(string lineno, string record)
{
   string[] fields = record.split("\\s+");
   int lineNumber = integer.parseInt(fields[0]);
   Line line = new Line(
      integer.parseInt(fields[1]),
      integer.parseInt(fields[2]),
      integer.parseInt(fields[3]),
      integer.parseInt(fields[4])
   );

   // Emit key-value pairs for each unique pair of lines
   for (int i = 1; i <= N; i++)
   {
      if (i != lineNumber)
      {
         string key;

         // Ensure the smaller line number comes first in the key
         if (i < lineNumber) {
            key = i + "-" + lineNumber;
         } else {
            key = lineNumber + "-" + i;
         }

         emit(key, line);

         // Sample emit (1-2, line1) , (1-3,line1) , (1-4,line1) .. (1-N,line1)
         // (1-2, line2) , (2-3, line2) , (2-4, line2) .. (2-N, line2)
         // So for each pair there will be two emit.
      }
   }
}

// Reducer function
Reduce(string linePair, List<Line> lines)
{
   // Proceed only if exactly two lines are present for the pair
   if (lines.size() == 2) {
      Line l1 = lines.get(0);
      Line l2 = lines.get(1);
```

```
        // Check if the two lines intersect
        if (isIntersecting(l1, l2)) {
            // Get the intersection point and emit it
            Point intersection = getIntersection(l1, l2);
            emit("Intersection", intersection);
        }
    }
}
```