Distributed Sorting

Progress Presentation

20190629 김균서
20150271 정용준

Weekly Progress - Week1

- Create Github Repo for project
- Understand project goal and objectives & Identify challenges
- Overall plan for each week
- Weekly meeting schedule with exact date

Week 1: 프로젝트 이해, Git repo 생성 및 팀원과의 일정 조율
Week 2, 3: 전반적인 프로그램 구조 디자인, 구현 방법 구체화, 핵심 내용은 비워둔 채 전체적인 틀 형성
Week 4: 각각의 세부 코딩 영역 역할 분배 후 코드 작성
Week 5: 각자 상대가 구현한 코드에 대한 이해 및 교차 테스트, 디버깅
Week 6, 7: 전체 프로그램에 대한 테스트, 디버깅
Week 8: 일정 지연에 대한 대비 기간이자 최종 프레젠테이션 준비 기간

< Meetings >

- 10.19.(Sat) : 프로젝트 주제 인식, 향후 계획 작성
- 10.25.(Fri) : 프로그래밍 환경 구축, 프로그램 구조 디자인
- 10.29.(Tue) : 프로그램 구조 디자인
- 11.01.(Sat) : 사용 라이브러리 선정 등 구현 방법 구체화, 클래스 설계
- 11.05.(Tue) : 코드 영역 분담 및 작성
- 11.09.(Sat) : 코드 작성, 진행 상황 확인
- 11.12.(Tue) : 서로의 구현 코드 설명, 이해 및 테스트
- 11.16.(Sat) : 코드 수정/테스트/디버깅, 중간 발표 준비
- 11.19.(Tue) : 중간 발표 준비
- 11.23.(Sat) : 중간 발표 피드백 수용, 코드 병합 후 전체 프로그램 테스트
- 11.26.(Tue) : 테스트 및 디버깅, 필요 시 재구현
- 11.30.(Sat) : 테스트 및 디버깅
- 12.03.(Tue) : 테스트 및 디버깅, 최종 발표 준비

Week 1

프로젝트 목표 인식

1. Project goal

이 프로젝트의 목표는 여러 대의 머신에 분산되어 저장된 키/값 레코드를 정렬하는 것이다. 기본적으로 주어진 데이터는 디스크에 저장된 100바이트의 레코드로 이루어져 있으며, 첫 10바이트가 키로 사용되고 나머지 90바이트는 정렬 과정에서는 사용되지 않는 값이다. 목표는 이러한 레코드를 다음과 같은 환경에서 정렬하는 것이다:

> • 단일 디스크에 저장된 데이터를 정렬하는 것이 초기 목표이다. 메모리의 크기가 제한된 상황에서, 예를 들어 50GB 데

2. Challenges

1. **메모리 제한 문제**
   정렬하려는 데이터가 메모리 크기(8GB)를 초과하는 상황에서 전체 데이터를 메모리로 가져와 정렬하는 것이 불가능
   이를 해결하기 위해 디스크 기반 병합 정렬을 사용해야 한다. (데이터를 작은 청크로 나누어 각각을 디스크에서 처리
   병합)

2. **디스크 용량 문제**
   데이터가 디스크에 모두 저장되지 않는 경우. 한 예시로 10TB의 데이터를 1TB 용량의 디스크에 저장해야 하는 상황

Weekly Progress - Week2

- Brainstorming idea about system architecture
- Break down into 3 key parts: Master / Worker / Communication
- Very brief skeleton code with the help of ChatGPT

1. System Architecture

1. Master Node

master가 수행하는 작업들은 다음과 같다.

- Track and communicate with workers
- Distribute sorting tasks
- Collect results and determine ordering
- Determine the final ordering of data blocks

2. Worker Nodes

worker가 수행하는 작업들은 다음과 같다.

- Receive tasks from the master
- Sort local data blocks
- Shuffle sorted datat to appropriate workers
- Merge sorted blocks

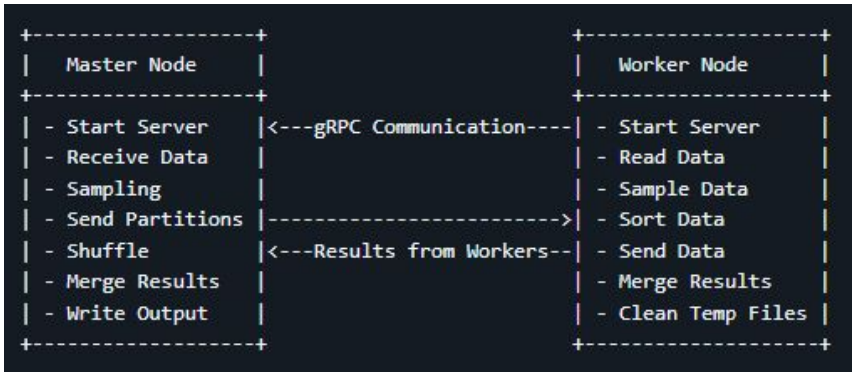
```
syntax = "proto3";

service SortService {
  rpc DistributeTasks (TaskRequest) returns (TaskResponse);
  rpc CollectResults (ResultRequest) returns (ResultResponse);
}

message TaskRequest {
  string workerId = 1;
  string inputBlock = 2;
}

message TaskResponse {
  string status = 1;
}

message ResultRequest {
  string workerId = 1;
  string sortedBlock = 2;
}

message ResultResponse {
  string status = 1;
}
```

```
class Master(numWorkers: Int) {
  val workers = new Array[String](numWorkers) // stores IPs of workers

  // Start master node
  def startMaster(): Unit = {
    // implement network code here to handle worker connections
  }

  // Distribute sorting tasks to workers
  def distributeTasks(): Unit = {
    // logic to distribute input blocks
  }

  // Collect results and determine ordering
  def collectResults(): Unit = {
    // logic to gather sorted blocks and determine final ordering
  }

  // Other master methods...
}
```

Weekly Progress - Week3

- Progress delayed, no meetings
- Prepare VM cluster server access
- Personal study on weakness in Scala syntax, library usage, etc.
- Master ↔ Worker structure remind

- Sampling:
  마스터는 각 워커로부터 샘플 데이터를 수집하여 전체 데이터의 분포를 분석합니다.
  이를 위해 각 워커에서 일정 비율로 데이터를 샘플링합니다.
- Sort/Partition:
  각 워커는 자신의 데이터 파티션을 정렬합니다.
  정렬 후, 각 워커는 정렬된 데이터의 키를 기준으로 분할합니다.
- Shuffle:
  마스터는 각 워커에게 정렬된 데이터를 기반으로 데이터를 다른 워커로 전송하도록 지시합니다.
  데이터를 송신할 워커와 수신할 워커의 매핑을 관리합니다.
- Merge:
  각 워커는 수신한 데이터를 병합하고 최종 정렬된 결과를 생성합니다.
  결과를 지정된 출력 디렉토리에 저장합니다.



Weekly Progress - Week4

- Access to test cluster
- test connectivity and status check with master and worker machines
- create dummy dataset by using gensort

```
System load:  0.0                Processes:             210
Usage of /:   1.2% of 475.07GB   Users logged in:       1
Memory usage: 2%                 IP address for ens160: 2.2.2.101
Swap usage:   0%

purple@vm01:~$ ls -al
total 40
drwxr-xr-x  5 purple purple 4096 Nov 10 22:08 .
drwxr-xr-x 14 root   root   4096 Oct 22 20:45 ..
-rw-------  1 purple purple  165 Oct 22 15:19 .bash_history
-rw-r--r--  1 purple purple  220 Apr  5  2018 .bash_logout
-rw-r--r--  1 purple purple 3908 Oct 22 20:42 .bashrc
drwxrwxr-x  3 purple purple 4096 Nov 10 22:08 .cache
drwx------  3 purple purple 4096 Nov 10 22:08 .gnupg
-rw-r--r--  1 purple purple  807 Apr  5  2018 .profile
drwxrwxr-x  3 purple purple 4096 Oct 22 14:37 .sbt
-rw-------  1 purple purple   50 Nov 10 22:08 .Xauthority
```

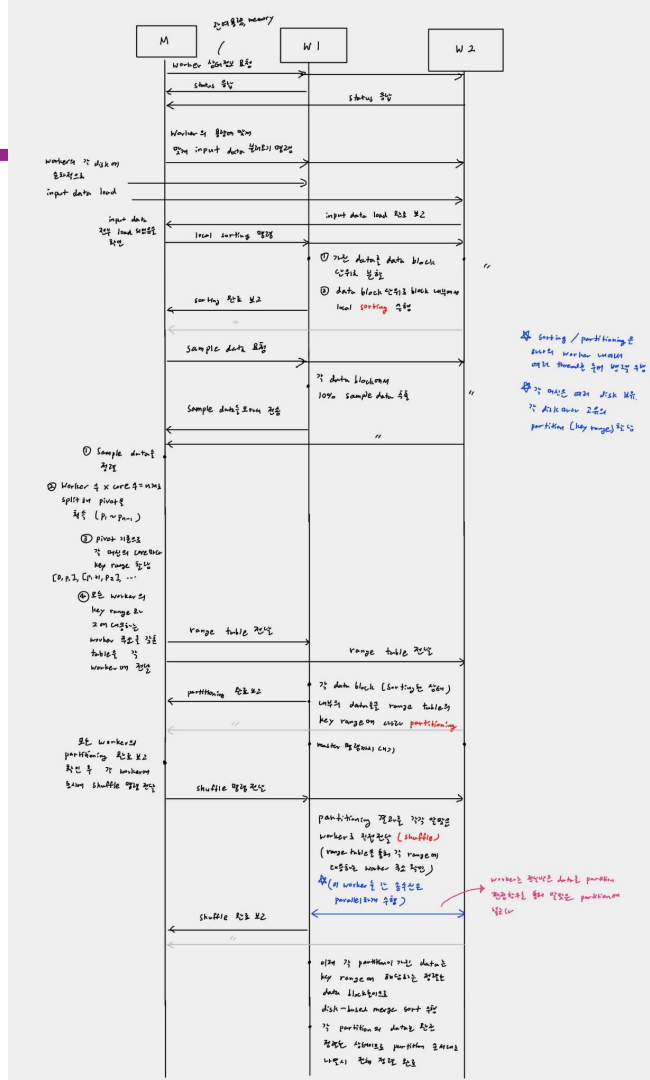
```
[purple@vm-1-master ~]$ gensort -b 1000 ./test/partition1
[purple@vm-1-master ~]$ dir partition1
dir: cannot access partition1: No such file or directory
[purple@vm-1-master ~]$ cd test/
[purple@vm-1-master test]$ ls
partition1
[purple@vm-1-master test]$ dir partition1
partition1
[purple@vm-1-master test]$ head partition1
```

```
[purple@vm-1-master test]$ gensort -a 1000 partition2
[purple@vm-1-master test]$ ls
partition1  partition2
[purple@vm-1-master test]$ head partition2
AsfAGHM5om  00000000000000000000000000000000  00002222000022220000222200002222000022220000000
~sHd0jDv6X  00000000000000000000000000000001  77779999444488885555CCCC777755555555BBBB6666444
uI^EYm8s=|  00000000000000000000000000000002  CCCCFFFF777799995555FFFF11112222999988884444DDDD
Q)JN)R9z-L  00000000000000000000000000000003  FFFF1111000000000006666888BBBB33333333AAAA1111
o4FoBkqERn  00000000000000000000000000000004  7777AAAABBBBBBBB22224444444499995555BBBB1111888
*}-Wz1;TD-  00000000000000000000000000000005  AAAA88883333BBBB888888884444777722227779999000
```

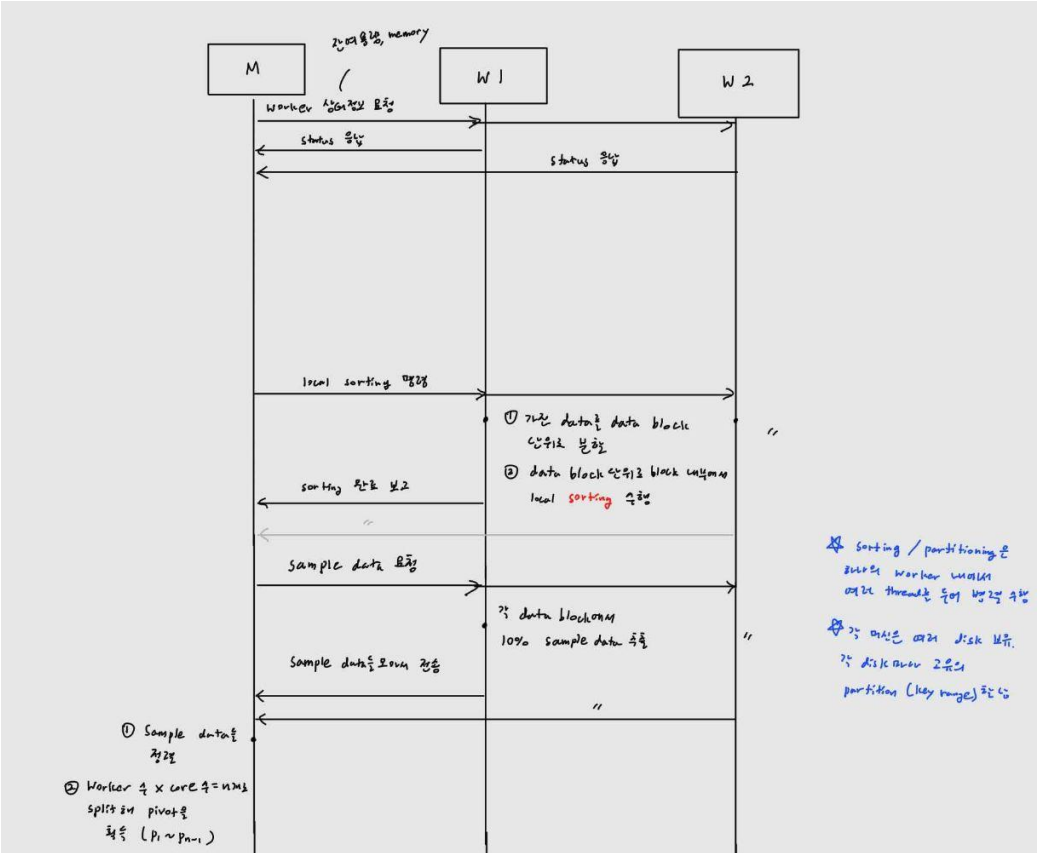
Weekly Progress - Week5

- Fully define master/worker role concept
- Organize necessary communications, functions
  by a single timeline diagram

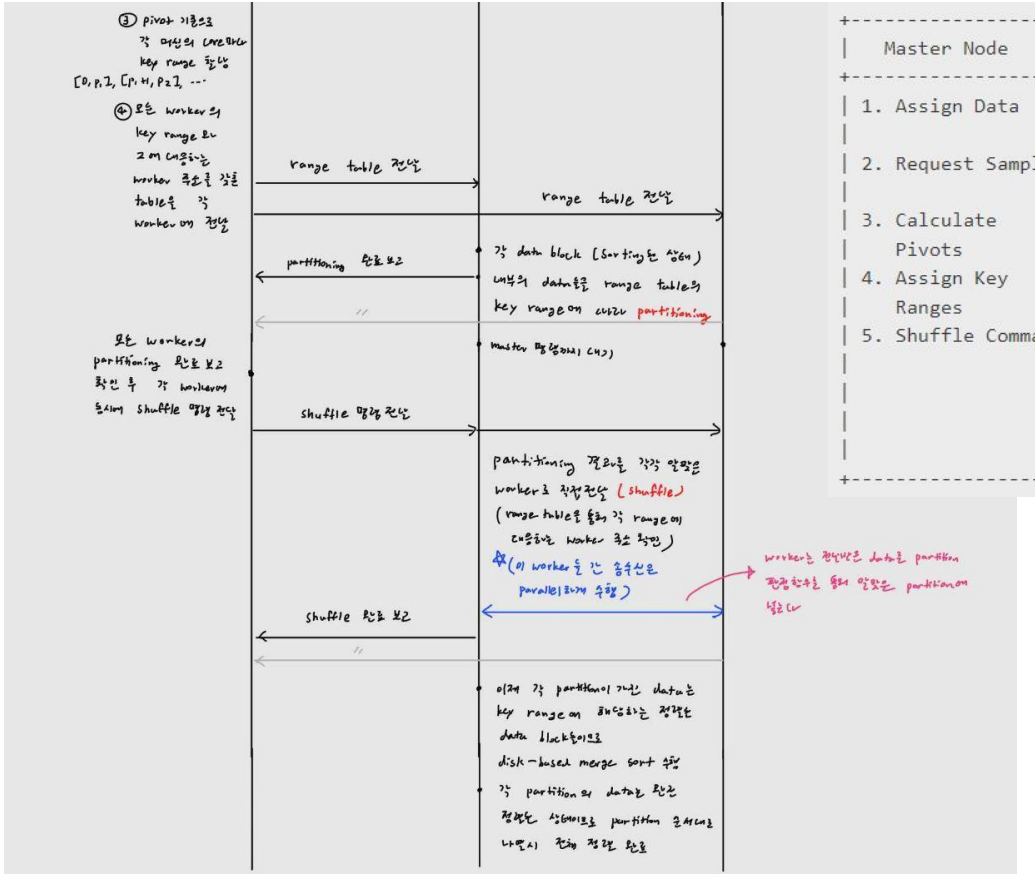
Distributed Sorting System
- Master
  - 작업 관리 및 분배
    - 워커에 작업 할당
    - 데이터 블록 나누기 (샘플링 및 파티셔닝을 위한 블록 단위 분할)
  - 네트워크 통신
    - 워커와 상태 통신 (작업 진행 및 오류 보고)
    - gRPC 및 Protobuf 사용 (샘플 수집, 데이터 분배 등)
  - 샘플링 및 파티셔닝
    - 각 워커로부터 샘플 수집
    - pivot 값 선택 (기본 및 정밀 방법)
    - 파티셔닝 계획 수립 및 전달
  - 셔플 및 병합 관리
    - 정렬된 데이터 블록 수집 (큐 기반 데이터 관리)
    - 워커 간 데이터 병합
    - 최종 병합 수행 및 정렬 결과 생성
  - 작업 상태 모니터링
    - 각 워커의 작업 상태 모니터링
    - 오류 및 지연 처리 (재전송 및 예외 관리)
    - 정렬 완료 상태 수집
- Worker
  - 데이터 수신 및 처리
    - 마스터로부터 데이터 수신 (파티셔닝 정보 및 정렬 작업)
    - 샘플링 및 전송 (마스터에게 샘플 데이터 전송)
    - 데이터 블록 정렬 수행
  - 네트워크 통신
    - 작업 상태 보고 (작업 진행 상황 및 오류 보고)
    - 정렬된 데이터 전송 (다른 워커 또는 마스터로 전송)
    - 다른 워커와의 데이터 병합 및 전송 (워커 간 통신)
  - 정렬 작업 수행
    - 데이터 블록 로드 및 정렬
    - 메모리 및 디스크 활용 정렬 (메모리 부족 시 디스크 사용)
    - 병합 정렬 진행 (다른 워커와의 데이터 병합)
  - 병렬 처리 (멀티코어 활용)
    - 멀티코어 활용 정렬 (병렬 처리로 정렬 속도 향상)
    - 병합 정렬 시 여러 스레드 활용
  - 임시 파일 관리
    - 정렬 과정에서 생성되는 임시 파일 관리
    - 최종적으로 임시 파일 삭제 (정렬 완료 후 클린업)
- Master-Worker Interaction
  - 작업 할당
    - 마스터가 데이터 블록을 나누어 워커에게 할당
  - 상태 보고
    - 워커가 작업 진행 상황을 마스터에 보고
  - 샘플링 및 파티셔닝
    - 각 워커가 샘플 데이터를 마스터에 전송



Design


M
W 1
W 2
Worker 상태정보 요청
status 응답
status 응답
local sorting 명령
① 가진 data를 data block 단위로 분할
② data block 단위로 block 내부에서 local sorting 수행
sorting 완료 보고
sample data 요청
각 data block에서 10% sample data 추출
sample data들 모아서 전송
① Sample data들 정렬
② Worker 수 x core 수 = n개로 split 해 pivot을 획득 (p1 ~ pn-1)
★ sorting / partitioning은 해당 worker 내에서 여러 thread를 통해 병렬 수행
★ 각 머신은 여러 disk 보유. 각 disk마다 고유의 partition (key range) 할당

Design




```
+--------------------+                          +--------------------+
|   Master Node      |                          |   Worker Node      |
+--------------------+                          +--------------------+
| 1. Assign Data     |                          | 1. Split Data      |
|                    |                          | 2. Local Sort      |
| 2. Request Sample  |----Request Sample Data------>| 3. Extract Sample  |
|                    |                          | 4. Send Sample     |
| 3. Calculate       |<----Receive Sample Data------|                    |
|    Pivots          |                          |                    |
| 4. Assign Key      |----Send Key Ranges---------->| 5. Partition Data  |
|    Ranges          |                          |                    |
| 5. Shuffle Command |                          | 6. Shuffle Data    |
|                    |<----Send Shuffled Data-------|                    |
|                    |                          | 7. Merge Locally   |
|                    |                          | 8. Disk-Based      |
|                    |                          |    Merge Sort      |
+--------------------+                          +--------------------+
```

Details

- Programming environment
  - CentOS7 3.10.0-1160.59.1.el7.x86_64
  - OpenJDK 1.8.0_422-422
  - Scala version: 2.12.19
  - sbt version: 1.10.2
  - IDE: VSCode
  - Version Control: Github

- Libraries
  - gRPC with Protobuf (ScalaPB)
  - Spark
  - Apache Commons IO

- Logging

  Haven’t used yet, but planning

- Testing framework
  - ScalaTest
  - JUnit

Logistics

- Communication error handling:
  two-member team, frequent meeting & short review at the beginning
- Used KakaoTalk for quick updates
- Clear role division:
  Reminded each other after every meeting.
- Role
  김균서 : progress report, code(master), git repository, network
  정용준 : schedule, milestone, code(worker), network

Implementation - Worker

Worker(revised)

1. 각 Worker에서 data를 data block 단위로 분할하고, 내부에서 local sorting 수행

```scala
import java.io._
import scala.io.Source

def localSorting(filePath: String, blockSize: Int): Unit = {
  val source = Source.fromFile(filePath)
  val dataBlock = source.getLines().take(blockSize).toList.sorted
  source.close()

  val outputFile = new File("sorted_block.txt")
  val writer = new BufferedWriter(new FileWriter(outputFile))
  dataBlock.foreach(line => writer.write(line + "\n"))
  writer.close()

  reportSortingCompletionToMaster()
}

//Worker에서 master에게 sorting 완료 보고
def reportSortingCompletionToMaster(): Unit = {
  val message = "Sorting complete"
  sendToMaster(message)
}
```

2. 각 Worker에서 data block의 10% sample data 추출 후 master에 전송

```
import scala.util.Random
import scala.io.Source

def sampleAndSendData(filePath: String): Unit = {
  val source = Source.fromFile(filePath)
  val dataBlock = source.getLines().toList
  source.close()

  val sampleSize = (dataBlock.size * 0.1).toInt
  val sampleData = Random.shuffle(dataBlock).take(sampleSize)

  sendToMaster("sample_data", sampleData)
}

def sendToMaster(messageType: String, data: List[String]): Unit = {
  val dataPacket = Map("message_type" -> messageType, "data" -> data)
  val serializedData = ujson.write(dataPacket)

  // 소켓을 사용해 마스터에 데이터 전송
```

3. Master에서 pivot 선택 및 key range 할당 후 worker에게 전달

```
def determineAndSendKeyRanges(sampleData: List[String], workers: Int, cores: Int): Unit =
  val sortedSample = sampleData.sorted
  val pivots = (1 until workers * cores).map { i =>
    sortedSample((i * sortedSample.size) / (workers * cores))
  }

  val keyRangeTable = pivots.sliding(2).zipWithIndex.map {
    case (List(min, max), index) =>
      (s"worker_${index / cores + 1}_core_${index % cores + 1}", (min, max))
  }.toMap

  // 각 worker에 key range 전송
  keyRangeTable.foreach { case (workerCore, range) =>
    sendToWorker(workerCore, range)
  }
}
```

4. Worker에서 sorting된 data를 key range에 따라 partitioning 후 master에 완료 보고

```
def partitionData(filePath: String, keyRangeTable: Map[String, (String, String)]): Map[Str
  val source = Source.fromFile(filePath)
  val dataBlock = source.getLines().toList
  source.close()

  val partitions = keyRangeTable.map {
    case (workerCore, (min, max)) =>
      val partition = dataBlock.filter(record => record >= min && record <= max)
      workerCore -> partition
  }

  reportPartitioningCompletionToMaster()
  partitions
}

def reportPartitioningCompletionToMaster(): Unit = {
  sendToMaster("Partitioning complete")
}
```

5. Master에서 모든 worker의 partitioning 완료 보고를 확인한 후 각 worker에 shuffle 명령 전달

```
def sendShuffleCommand(): Unit = {
  val message = "Shuffle data now"
  sendToWorkers(message)
}

def sendToWorker(targetWorker: String, data: List[String]): Unit = {
  val dataPacket = Map("worker" -> targetWorker, "data" -> data)
  val serializedData = ujson.write(dataPacket)

  // 예시로 소켓을 통해 다른 Worker로 전송
  val clientSocket = new java.net.Socket(targetWorker, 12345)
  val out = new java.io.PrintWriter(clientSocket.getOutputStream, true)
  out.println(serializedData)
  clientSocket.close()
}

def reportShuffleCompletionToMaster(): Unit = {
  sendToMaster("Shuffle complete")
}
```

6. Worker에서 partition을 알맞은 worker에 전달(shuffle)

```
def shuffleData(partitions: Map[String, List[String]]): Unit = {
  partitions.foreach {
    case (workerAddress, data) =>
      sendToWorker(workerAddress, data)
  }
  reportShuffleCompletionToMaster()
}

def reportShuffleCompletionToMaster(): Unit = {
  val message = "Shuffle complete"
  sendToMaster(message)
}
```

7. 각 partition이 가진 data에 대해 disk-based merge sort 수행

```
def diskBasedMergeSort(sortedPartitions: List[String]): Unit = {
  val mergedData = sortedPartitions.sorted
  val outputFile = new File("fully_sorted_data.txt")
  val writer = new BufferedWriter(new FileWriter(outputFile))
  mergedData.foreach(line => writer.write(line + "\n"))
  writer.close()
}
```

Implementation - Worker

```scala
import java.net.{Socket, ServerSocket}
import scala.collection.mutable
import scala.io.Source
import scala.util.control.NonFatal
import play.api.libs.json._

object DataPartitioner {

  // 1. 레코드가 어느 워커에 속하는지 결정하는 함수
  def findWorkerPartition(record: Int, keyRangeTable: Map[String, Map[String, Int]]): Stri
    keyRangeTable.foreach { case (workerId, keyRange) =>
      if (keyRange("min") <= record && record <= keyRange("max")) {
        return workerId
      }
    }
    throw new IllegalArgumentException(s"Record $record does not fit into any worker's key
  }

  // 2. 레코드가 worker 내의 어느 코어에 속하는지 결정하는 함수
  def findCorePartition(record: Int, workerRangeTable: Map[String, Map[String, Int]]): Str
    workerRangeTable.foreach { case (coreId, keyRange) =>
      if (keyRange("min") <= record && record <= keyRange("max")) {
        return coreId
      }
    }
    throw new IllegalArgumentException(s"Record $record does not fit into any core's key r
  }

  // 3. 데이터를 worker에 전송하는 함수
  def sendToWorker(targetWorker: String, partition: List[Int], port: Int): Unit = {
    try {
      val socket = new Socket(targetWorker, port)
      val out = socket.getOutputStream

      val serializedData = Json.stringify(Json.toJson(partition))
      out.write(serializedData.getBytes)
      out.flush()
      socket.close()

      println(s"Sent partition to worker: $targetWorker")
    } catch {
      case NonFatal(e) =>
        println(s"Error sending data to worker $targetWorker: ${e.getMessage}")
    }
  }
```

```scala
// 4. worker가 받은 데이터를 각 코어로 분배하는 함수
def allocateToCore(dataBlock: List[Int], workerRangeTable: Map[String, Map[String, Int]
  val coresPartitions = mutable.Map[String, List[Int]]().withDefaultValue(List())

  dataBlock.foreach { record =>
    val coreId = findCorePartition(record, workerRangeTable)
    coresPartitions(coreId) = coresPartitions(coreId) :+ record
  }

  coresPartitions.toMap
}

// 5. 전체적인 데이터 분류 및 전송 로직
def partitionAndSendData(sortedBlock: List[Int], keyRangeTable: Map[String, Map[String,
  val partitions = mutable.Map[String, List[Int]]().withDefaultValue(List())

  // 1. 각 레코드를 해당하는 워커로 분류
  sortedBlock.foreach { record =>
    val targetWorker = findWorkerPartition(record, keyRangeTable)
    partitions(targetWorker) = partitions(targetWorker) :+ record
  }

  // 2. 분류된 파티션을 각 워커에 전송
  partitions.foreach { case (workerId, partition) =>
    sendToWorker(workerId, partition, 9000) // Port는 예시로 9000으로 설정
  }

  // 3. 각 worker는 받은 데이터를 코어에 분배
  partitions.foreach { case (workerId, partition) =>
    val workerRangeTable = workerRangeTables(workerId)
    val corePartitions = allocateToCore(partition, workerRangeTable)
    println(s"Worker $workerId allocated data to cores: $corePartitions")
  }

  // 파티셔닝 작업 완료를 마스터에 보고
  reportPartitioningCompletionToMaster()
}
```

Only Partitioning

```scala
// 6. 파티셔닝 작업 완료 보고 함수
def reportPartitioningCompletionToMaster(host: String = "localhost", port: Int = 9000):
  val message = "Partitioning complete"
  try {
    val socket = new Socket(host, port)
    val out = socket.getOutputStream

    out.write(message.getBytes)
    out.flush()
    socket.close()

    println(s"Reported partitioning completion to master: $message")
  } catch {
    case NonFatal(e) =>
      println(s"Error reporting partitioning completion to master: ${e.getMessage}")
  }
}

def main(args: Array[String]): Unit = {
  // 메인 함수에서는 예시 데이터를 사용하지 않고, 실제 작업 시 동적으로 설정된 데이터를 사용하도
  val sortedBlock = List(100, 150, 1200, 2500, 3000) // 예시 데이터 블록

  // keyRangeTable 및 workerRangeTables는 실제 작업 시 적절한 값으로 초기화되어야 합니다.
  val keyRangeTable = Map.empty[String, Map[String, Int]]
  val workerRangeTables = Map.empty[String, Map[String, Map[String, Int]]]

  // 데이터이 설정된 이후 partitionAndSendData 호출
  if (keyRangeTable.nonEmpty && workerRangeTables.nonEmpty) {
    partitionAndSendData(sortedBlock, keyRangeTable, workerRangeTables)
  } else {
    println("Key range table and worker range tables must be initialized before partitio
  }
}
}
```

Milestone

#1. Project Goal
- 전반적인 계획 설정
- 스케줄 관리

#2. Test cluster
- Master, Worker 실행 & 연결
- Master-Worker 통신에 사용할 network library 선정
- gensort sample data 생성

#3. Architecture & Design
- 프로그램 전체적인 구조 설정
- Master-Worker 간의 역할과 기능 공부
- 구현할 함수 설정
- skeleton code 작성

#4. Code - Worker
- local sorting / sample data / partitioning / shuffle / disk-based merge function 구현
- Master-worker network 코드
  sendToMaster, reportLoadCompletionToMaster, SampleAndSendData function 구현
- worker-worker network 코드
  sendToWorker

#5. Code - Master
- Pivot, sample data sorting, range table function 구현
- Master-worker network코드
  MasterServer, startServer

#6. Debugging & Testing
- 결과값 출력 후 기대치 비교
- 코드 수정 및 검토
- Time complexity 고려하여 코드 재수정

Future Plan

Week 6 : Implement Workers & Master functions (Now)
Week 7 : Implement network communication between Workers & Master / Testing
Week 8 : Testing & Debugging

Q&A

Any Question?