NAME:- DUBEY KARAN SANJEEV
CLASS:- B.E - 4
ROLL NO:- 04
BATCH:- A

Experiment No-8

AIM : To implement Sorting and Searching using Map-Reduce.

THEORY:

In Hadoop, the process by which the intermediate output from mappers is transferred to the reducer is called Shuffling. Reducer gets 1 or more keys and associated values on the basis of reducers. Intermediated key-value generated by mapper is sorted automatically by key.

In this blog, we will discuss in detail about shuffling and Sorting in Hadoop MapReduce. Here we will learn what is sorting in Hadoop, what is shuffling in Hadoop, what is the purpose of Shuffling and sorting phase in MapReduce, how MapReduce shuffle works and how MapReduce sort works.

Shuffle phase in Hadoop transfers the map output from Mapper to a Reducer in MapReduce. Sort phase in MapReduce covers the merging and sorting of map outputs. Data from the mapper are grouped by the key, split among reducers and sorted by the key. Every reducer obtains all values associated with the same key. Shuffle and sort phase in Hadoop occur simultaneously and are done by the MapReduce framework.

1. Shuffling in MapReduce

The process of transferring data from the mappers to reducers is known as shuffling i.e. the process by which the system performs the sort and transfers the map output to the reducer as input. So, MapReduce shuffle phase is necessary for the reducers, otherwise, they would not have any input (or input from every mapper). As shuffling can start even before the map phase has finished so this saves some time and completes the tasks in lesser time.

2. Sorting in MapReduce

The keys generated by the mapper are automatically sorted by MapReduce Framework, i.e. Before starting of reducer, all intermediate key-value pairs in MapReduce that are generated by mapper get sorted by key and not by value. Values passed to each reducer are not sorted; they can be in any order.

Sorting in Hadoop helps reducer to easily distinguish when a new reduce task should start. This saves time for the reducer. Reducer starts a new reduce task when the next key in the sorted input data is different than the previous. Each reduce task takes key-value pairs as input and generates key-value pair as output.

The sorting of large amount of data is done in shuffle-sort phase of a MapReduce task. In Mapper class, the Map function that reads the input rows and produces an output pair with key and value. Key is represented in second column and value is represented in first column. After the Map phase is completed, the key-value pairs are sorted by keys using shuffle-sort phase. Sorting of data automatically occurs between the map and reduce phase during which the comparator is used to determine the order of keys and their corresponding lists of values
and the sorted key value pairs are fed into the reduce function. The reduce function aggregates the input data and writes into output directory.

CONCLUSION: In conclusion, Shuffling-Sorting occurs simultaneously to summarize the Mapper intermediate output. Shuffling and sorting in Hadoop MapReduce are not performed at all if you specify zero reducers.

Program formation/ Execution/ ethical practices (06)	**Timely Submission and Documentation (02)**	**Viva Answer (02)**	**Experiment Marks (10)**	**Teacher Signature with date**

```
//package com.my.cert.example;
package Sortf1;
import java.nio.ByteBuffer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.IntWritable.Comparator;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparator;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import
org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class Sortf2 {
 public static void main(String[] args) throws Exception {

 //Path inputPath = new Path("\\user\\training\\test.txt");
//Path outputDir = new Path("\\user\\training\\out123");

 Path inputPath = new Path(args[0]);
 Path outputDir = new Path(args[1]);

 // Create configuration
 Configuration conf = new Configuration(true);

 // Create job
 Job job = new Job(conf, "Test HIVE commond");
 job.setJarByClass(Sortf2.class);

 // Setup MapReduce
 job.setMapperClass(Sortf2.MapTask.class);
 job.setReducerClass(Sortf2.ReduceTask.class);
 job.setNumReduceTasks(1);

 // Specify key / value
 job.setMapOutputKeyClass(IntWritable.class);
 job.setMapOutputValueClass(IntWritable.class);
 job.setOutputKeyClass(IntWritable.class);
 job.setOutputValueClass(IntWritable.class);
 job.setSortComparatorClass(IntComparator.class);
 // Input
 FileInputFormat.addInputPath(job, inputPath);
 job.setInputFormatClass(TextInputFormat.class);

 // Output
 FileOutputFormat.setOutputPath(job, outputDir);
 job.setOutputFormatClass(TextOutputFormat.class);
 /*
 * // Delete output if exists FileSystem hdfs = FileSystem.get(conf); if
* (hdfs.exists(outputDir)) hdfs.delete(outputDir, true);  *
 * // Execute job int code = job.waitForCompletion(true) ? 0 : 1;
 * System.exit(code);
 */

 // Execute job
 int code = job.waitForCompletion(true) ? 0 : 1;
 System.exit(code);

 }

 public static class IntComparator extends WritableComparator {

 public IntComparator() {
 super(IntWritable.class);
 }

 @Override
 public int compare(byte[] b1, int s1, int l1,
 byte[] b2, int s2, int l2) {

 Integer v1 = ByteBuffer.wrap(b1, s1, l1).getInt();
 Integer v2 = ByteBuffer.wrap(b2, s2, l2).getInt();

 return v1.compareTo(v2) * (-1);
 }
 }

 public static class MapTask extends
 Mapper<LongWritable, Text, IntWritable, IntWritable> {
 public void map(LongWritable key, Text value, Context context)
throws java.io.IOException, InterruptedException {
 String line = value.toString();
 String[] tokens = line.split(","); // This is the delimiter between
int keypart = Integer.parseInt(tokens[0]);
 int valuePart = Integer.parseInt(tokens[1]);
 context.write(new IntWritable(valuePart), new IntWritable(keypart));

 }
 }

 public static class ReduceTask extends
 Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {  public void
reduce(IntWritable key, Iterable<IntWritable> list, Context context)  throws
java.io.IOException, InterruptedException {

 for (IntWritable value : list) {

 context.write(value,key);

 }

 }
```

```
    }
  }
```

INPUT:

```
21
36
8
12
35
14
95
75
26
23
69
51
85
97
24
65
18
3
```

OUTPUT:

```
3
8
12
14
18
21
23
24
26
35
36
51
65
69
75
85
95
97
```