

华中科技大学

课程实验报告

课程名称： 大数据处理

专业班级：

学 号：

姓 名：

指导教师： 刘海坤

报告日期： 2024年6月6日

计算机科学与技术学院

目录

实验 1: MapReduce 实验.....3

实验 2: Spark 实验.....17

实验 3: 图计算实验.....23

实验总结.....28

实验 1: MapReduce 实验

1.1 实验概述

本实验旨在通过编写和执行基于 MapReduce 编程模型的 PageRank 程序，帮助学生深入理解 MapReduce 的工作原理，并学会使用 Hadoop 框架进行大规模数据处理。通过此实验，学生将能够掌握 MapReduce 编程的基本概念、编写简单的 MapReduce 程序以及在分布式环境中运行它们。

本实验的环境要求包括 Linux 操作系统 (Ubuntu)、Java JDK 1.8 和 Hadoop 3.1.3。建议至少准备一台计算机或虚拟机，配置至少 4GB 内存和 100GB 的硬盘空间用于安装 Hadoop。同时需要互联网连接，用于下载所需的软件和文档。

实验数据集采用了 SNAP-Stanford 数据集，包含 281,903 个顶点和 2,312,497 条边。

通过完成本实验，我将深入理解 MapReduce 编程模型的工作原理，并掌握基本的 MapReduce 编程概念和技巧，同时能够在分布式环境中使用 Hadoop 框架进行大规模数据处理。

1.2 实验内容

本实验的主要内容是在开源系统 Hadoop 上实现 PageRank 算法，通过这一过程进一步理解 MapReduce 的原理。

PageRank 算法是搜索引擎发展的重要成果之一，其核心思想是基于从许多优质网页链接过来的网页，必定还是优质网页的假设。为了区分网页之间的优劣，PageRank 引入了一个值来评估网页的受欢迎程度，即 PR 值。PR 值越高，说明该网页的受欢迎程度越高。

算法开始时，将所有网页的 PR 值设定为相同的初始值，通常将初始 PR 值设置为 $1/N$ ，其中 N 表示网页的总数。然后，通过迭代计算，对所有网页的 PR 值进行更新。

迭代计算使用了以下公式：

$$PR(p_i) = (1 - d) / N + d * \sum (PR(p_j) / L(p_j))$$

在公式中， N 表示网页的总数， d 是阻尼因子，通常设为 0.85。 $PR(p_i)$ 表示网页 p_i 的 PR 值， $L(p_j)$ 表示网页 p_j 链出的网页数目，也称为出度。

在有限次迭代后，所有网页的 PR 值会收敛到一个固定的值。当两次迭代之间 PR 值的改变量小于一个设定的阈值时，算法结束。

1.2.1 阶段 1 图解析

PageRankJob1Mapper 类是一个映射器类，扩展了 Hadoop 的 Mapper 类。在 map 方法中，它解析输入图的每一行，并创建键-值对。输入格式为<nodeA> <nodeB>，表示从 nodeA 到 nodeB 的边。在映射过程中，它会跳过以#字符开头的注释行。然后，它将 nodeA 作为键，nodeB 作为值进行输出。另外，它还将 nodeA 和 nodeB 添加到 PageRank.NODES 列表中，以便在后续作业中计算图中节点的总数。

PageRankJob1Reducer 类是一个归约器类，扩展了 Hadoop 的 Reducer 类。在 reduce 方法中，它对给定的键和值进行归约操作。归约过程中，它遍历所有与给定键相关联的节点，并构建一个逗号分隔的值列表。它还为给定节点初始化了 PageRank 值，该值的初始值为 DAMPING FACTOR / TOTAL NODES。然后，它将键和值列表作为输出进行写入。

这些类是 PageRank 算法第一个作业的关键组成部分，它们负责解析输入图的边和节点，以及计算节点的初始 PageRank 值。这些结果将在后续的作业中用于迭代计算和更新 PageRank 值。

```
class PageRankJob1Mapper extends Mapper<LongWritable, Text, Text, Text>
{
    @Override
    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        if (value.charAt(0) != '#') {

            int tabIndex = value.find("\t");
            String nodeA = Text.decode(value.getBytes(), 0, tabIndex);
            String nodeB = Text.decode(value.getBytes(), tabIndex + 1,
value.getLength() - (tabIndex + 1));
            context.write(new Text(nodeA), new Text(nodeB));

            // add the current source node to the node list so we can
            // compute the total amount of nodes of our graph in Job#2
            PageRank.NODES.add(nodeA);
            // also add the target node to the same list: we may have a target
node
            // with no outlinks (so it will never be parsed as source)
            PageRank.NODES.add(nodeB);

        }

    }
}
```

```

class PageRankJob1Reducer extends Reducer<Text, Text, Text, Text> {

    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

        boolean first = true;
        String links = (PageRank.DAMPING / PageRank.NODES.size()) + "\t";

        for (Text value : values) {
            if (!first)
                links += ",";
            links += value.toString();
            first = false;
        }

        context.write(key, new Text(links));
    }
}

```

1.2.2 阶段 2 PageRank 计算

PageRankJob2Mapper 类是 Mapper 类的子类，它将输入数据解析为页面、页面排名和页面链接。输入数据的格式为：

<title> <page-rank> <link1>, <link2>, <link3>, ...<linkN>

Mapper 类的 map 方法首先通过制表符找到输入数据中的分隔位置，然后将输入数据分割为页面、页面排名和页面链接三个部分。接下来，它通过逗号分割页面链接，遍历所有的链接，并为每个链接创建一个新的键值对。键是链接，值是由页面排名和页面链接数组组成的文本。这样做是为了在计算过程中传递页面排名和页面链接数信息。最后，Mapper 类还会将原始的页面链接输出，以便 Reducer 类能够正确地重构输入数据。

PageRankJob2Reducer 类是 Reducer 类的子类，它接收 Mapper 类输出的键值对，并根据输入数据格式进行处理。输入数据有两种记录类型，分别是页面链接集合和链接页面的排名、源页面的页面排名和源页面的出链总数。Reducer 类通过迭代 Iterable 对象中的所有值，并根据记录类型执行不同的操作。

对于页面链接集合记录类型，Reducer 类会将链接添加到字符串 links 中，以便在后续使用中能够重构输入数据。

对于链接页面的排名记录类型，Reducer 类会提取页面排名和页面链接数，并计算其他页面对当前页面的贡献。然后，它根据 PageRank 算法的公式计算新

的页面排名，并将结果作为键值对输出。

总体而言，这段代码实现了 PageRank 算法的第二个 MapReduce 作业的 Mapper 和 Reducer 类。Mapper 类负责解析输入数据并生成中间结果，而 Reducer 类负责根据中间结果计算新的页面排名。

```
class PageRankJob2Mapper extends Mapper<LongWritable, Text, Text, Text>
{
    @Override
    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        int tIdx1 = value.find("\t");
        int tIdx2 = value.find("\t", tIdx1 + 1);

        // extract tokens from the current line
        String page = Text.decode(value.getBytes(), 0, tIdx1);
        String pageRank = Text.decode(value.getBytes(), tIdx1 + 1, tIdx2
- (tIdx1 + 1));
        String links = Text.decode(value.getBytes(), tIdx2 + 1,
value.getLength() - (tIdx2 + 1));

        String[] allOtherPages = links.split(",");
        for (String otherPage : allOtherPages) {
            Text pageRankWithTotalLinks = new Text(pageRank + "\t" +
allOtherPages.length);
            context.write(new Text(otherPage), pageRankWithTotalLinks);
        }

        // put the original links so the reducer is able to produce the
correct output
        context.write(new Text(page), new Text(PageRank.LINKS_SEPARATOR
+ links));
    }
}

class PageRankJob2Reducer extends Reducer<Text, Text, Text, Text> {
    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException,
```

```

InterruptedException {

    String links = "";
    double sumShareOtherPageRanks = 0.0;

    for (Text value : values) {

        String content = value.toString();

        if (content.startsWith(PageRank.LINKS_SEPARATOR)) {
            // if this value contains node links append them to the
            'links' string
            // for future use: this is needed to reconstruct the input
            for Job#2 mapper
            // in case of multiple iterations of it.
            links +=
            content.substring(PageRank.LINKS_SEPARATOR.length());
        } else {

            String[] split = content.split("\\t");

            // extract tokens
            double pageRank = Double.parseDouble(split[0]);
            int totalLinks = Integer.parseInt(split[1]);

            // add the contribution of all the pages having an outlink
            pointing
            // to the current node: we will add the DAMPING factor later
            when recomputing
            // the final pagerank value before submitting the result
            to the next job.
            sumShareOtherPageRanks += (pageRank / totalLinks);
        }
    }

    double newRank = PageRank.DAMPING * sumShareOtherPageRanks + (1
- PageRank.DAMPING);
    context.write(key, new Text(newRank + "\\t" + links));

}

}

```

1.2.3 阶段3 排序

PageRankJob3Mapper 类是 Mapper 类的子类，它将输入数据解析为页面和页面排名，并将页面排名作为键，页面作为值进行输出。

Mapper 类的 map 方法首先通过制表符找到输入数据中的分隔位置，然后将输入数据分割为页面和页面排名两个部分。接下来，它将页面排名解析为浮点数，并将页面排名作为键，页面作为值进行输出。这样做是为了利用 Hadoop 对键进行排序的功能。由于这个作业不需要 Reducer 类，仅需进行键的映射和排序，因此只有 Mapper 类被实现。

总体而言，这段代码实现了 PageRank 算法的第三个 MapReduce 作业的 Mapper 类，它负责解析输入数据并将页面排名作为键进行输出，以便进行排序操作。

```
class PageRankJob3Mapper extends Mapper<LongWritable, Text, DoubleWritable, Text> {

    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

        int tIdx1 = value.find("\t");
        int tIdx2 = value.find("\t", tIdx1 + 1);

        // extract tokens from the current line
        String page = Text.decode(value.getBytes(), 0, tIdx1);
        float pageRank = Float.parseFloat(Text.decode(value.getBytes(), tIdx1 + 1, tIdx2 - (tIdx1 + 1)));

        context.write(new DoubleWritable(pageRank), new Text(page));

    }

}
```

1.2.4 阶段4 PageRank 算法实现

首先代码中的 main 方法是程序的入口点，它接受命令行参数并解析它们。参数包括输入路径、输出路径、阻尼系数（damping factor）和迭代次数等。通过解析参数，可以设置算法的配置值。然后，代码会检查参数的有效性，并删除输出路径中已存在的文件。

接下来，代码会打印当前的配置值，并等待 1 秒钟。然后，它会依次运行三个作业（Job）：图解析作业（Job #1）、PageRank 计算作业（Job #2）和排序作

业 (Job #3)。每个作业都是一个独立的 MapReduce 作业。

job1 方法运行图解析作业，它将输入数据解析为图形表示，并初始化页面排名 (PageRank)。这个方法设置了作业的输入路径、输出路径、输入格式、输出格式、Mapper 类和 Reducer 类等。

job2 方法运行 PageRank 计算作业，它根据当前的页面排名计算新的排名，并生成与输入相同格式的输出。这个方法也设置了作业的输入路径、输出路径、输入格式、输出格式、Mapper 类和 Reducer 类等。

job3 方法运行排序作业，它根据页面排名对文档进行排序。这个方法同样设置了作业的输入路径、输出路径、输入格式、输出格式、Mapper 类等。

最后，代码打印完成消息并退出程序。

总体而言，这段代码实现了一个使用 Hadoop 集群运行 PageRank 算法的框架，它将输入数据解析为图形表示，并通过多次迭代计算页面排名，最后对结果进行排序。

```
public class PageRank {

    // args keys
    private static final String KEY_DAMPING = "--damping";
    private static final String KEY_DAMPING_ALIAS = "-d";

    private static final String KEY_COUNT = "--count";
    private static final String KEY_COUNT_ALIAS = "-c";

    private static final String KEY_INPUT = "--input";
    private static final String KEY_INPUT_ALIAS = "-i";

    private static final String KEY_OUTPUT = "--output";
    private static final String KEY_OUTPUT_ALIAS = "-o";

    private static final String KEY_HELP = "--help";
    private static final String KEY_HELP_ALIAS = "-h";

    // utility attributes
    public static NumberFormat NF = new DecimalFormat("00");
    public static Set<String> NODES = new HashSet<String>();
    public static String LINKS_SEPARATOR = "|";

    // configuration values
    public static Double DAMPING = 0.85;
    public static int ITERATIONS = 2;
    public static String IN_PATH = "";
    public static String OUT_PATH = "";
```

```

/**
 * This is the main class run against the Hadoop cluster.
 * It will run all the jobs needed for the PageRank algorithm.
 */
public static void main(String[] args) throws Exception {

    try {

        // parse input parameters
        for (int i = 0; i < args.length; i += 2) {

            String key = args[i];
            String value = args[i + 1];

            // NOTE: do not use a switch to keep Java 1.6 compatibility!
            if (key.equals(KEY_DAMPING) ||
key.equals(KEY_DAMPING_ALIAS)) {
                // be sure to have a damping factor in the interval [0:1]
                PageRank.DAMPING =
Math.max(Math.min(Double.parseDouble(value), 1.0), 0.0);
            } else if (key.equals(KEY_COUNT) ||
key.equals(KEY_COUNT_ALIAS)) {
                // be sure to have at least 1 iteration for the PageRank
algorithm
                PageRank.ITERATIONS =
Math.max(Integer.parseInt(value), 1);
            } else if (key.equals(KEY_INPUT) ||
key.equals(KEY_INPUT_ALIAS)) {
                PageRank.IN_PATH = value.trim();
                if (PageRank.IN_PATH.charAt(PageRank.IN_PATH.length()
- 1) == '/')
                    PageRank.IN_PATH = PageRank.IN_PATH.substring(0,
PageRank.IN_PATH.length() - 1);
            } else if (key.equals(KEY_OUTPUT) ||
key.equals(KEY_OUTPUT_ALIAS)) {
                PageRank.OUT_PATH = value.trim();
                if
(PageRank.OUT_PATH.charAt(PageRank.OUT_PATH.length() - 1) == '/')
                    PageRank.OUT_PATH = PageRank.OUT_PATH.substring(0,
PageRank.IN_PATH.length() - 1);
            } else if (key.equals(KEY_HELP) ||
key.equals(KEY_HELP_ALIAS)) {

```

```

        printUsageText(null);
        System.exit(0);
    }
}

} catch (ArrayIndexOutOfBoundsException e) {
    printUsageText(e.getMessage());
    System.exit(1);
} catch (NumberFormatException e) {
    printUsageText(e.getMessage());
    System.exit(1);
}

// check for valid parameters to be set
if (PageRank.IN_PATH.isEmpty() || PageRank.OUT_PATH.isEmpty()) {
    printUsageText("missing required parameters");
    System.exit(1);
}

// delete output path if it exists already
FileSystem fs = FileSystem.get(new Configuration());
if (fs.exists(new Path(PageRank.OUT_PATH)))
    fs.delete(new Path(PageRank.OUT_PATH), true);

// print current configuration in the console
System.out.println("Damping factor: " + PageRank.DAMPING);
System.out.println("Number of iterations: " +
PageRank.ITERATIONS);
System.out.println("Input directory: " + PageRank.IN_PATH);
System.out.println("Output directory: " + PageRank.OUT_PATH);
System.out.println("-----");

Thread.sleep(1000);

String inPath = null;;
String lastOutPath = null;
PageRank pagerank = new PageRank();

System.out.println("Running Job#1 (graph parsing) ...");
boolean isCompleted = pagerank.job1(IN_PATH, OUT_PATH + "/iter00");
if (!isCompleted) {
    System.exit(1);
}

```

```

        for (int runs = 0; runs < ITERATIONS; runs++) {
            inPath = OUT_PATH + "/iter" + NF.format(runs);
            lastOutPath = OUT_PATH + "/iter" + NF.format(runs + 1);
            System.out.println("Running Job#2 [" + (runs + 1) + "/" +
PageRank.ITERATIONS + "] (PageRank calculation) ...");
            isCompleted = pagerank.job2(inPath, lastOutPath);
            if (!isCompleted) {
                System.exit(1);
            }
        }

        System.out.println("Running Job#3 (rank ordering) ...");
        isCompleted = pagerank.job3(lastOutPath, OUT_PATH + "/result");
        if (!isCompleted) {
            System.exit(1);
        }

        System.out.println("DONE!");
        System.exit(0);
    }

    /**
     * This will run the Job #1 (Graph Parsing).
     * Will parse the graph given as input and initialize the page rank.
     *
     * @param in the directory of the input data
     * @param out the main directory of the output
     */
    public boolean job1(String in, String out) throws IOException,
                                                                ClassNotFoundException,
                                                                InterruptedException {

        Job job = Job.getInstance(new Configuration(), "Job #1");
        job.setJarByClass(PageRank.class);

        // input / mapper
        FileInputFormat.addInputPath(job, new Path(in));
        job.setInputFormatClass(TextInputFormat.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setMapperClass(PageRankJob1Mapper.class);

        // output / reducer
        FileOutputFormat.setOutputPath(job, new Path(out));
    }

```

```

        job.setOutputFormatClass(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.setReducerClass(PageRankJob1Reducer.class);

        return job.waitForCompletion(true);

    }

    /**
     * This will run the Job #2 (Rank Calculation).
     * It calculates the new ranking and generates the same output format
as the input,
     * so this job can run multiple times (more iterations will increase
accuracy).
     *
     * @param in the directory of the input data
     * @param out the main directory of the output
     */
    public boolean job2(String in, String out) throws IOException,
                                                                    ClassNotFoundException,
                                                                    InterruptedException {

        Job job = Job.getInstance(new Configuration(), "Job #2");
        job.setJarByClass(PageRank.class);

        // input / mapper
        FileInputFormat.setInputPaths(job, new Path(in));
        job.setInputFormatClass(TextInputFormat.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setMapperClass(PageRankJob2Mapper.class);

        // output / reducer
        FileOutputFormat.setOutputPath(job, new Path(out));
        job.setOutputFormatClass(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.setReducerClass(PageRankJob2Reducer.class);

        return job.waitForCompletion(true);

    }

```

```

/**
 * This will run the Job #3 (Rank Ordering).
 * It will sort documents according to their page rank value.
 *
 * @param in the directory of the input data
 * @param out the main directory of the output
 */
public boolean job3(String in, String out) throws IOException,
                                                ClassNotFoundException,
                                                InterruptedException {

    Job job = Job.getInstance(new Configuration(), "Job #3");
    job.setJarByClass(PageRank.class);

    // input / mapper
    FileInputFormat.setInputPaths(job, new Path(in));
    job.setInputFormatClass(TextInputFormat.class);
    job.setMapOutputKeyClass(DoubleWritable.class);
    job.setMapOutputValueClass(Text.class);
    job.setMapperClass(PageRankJob3Mapper.class);

    // output
    FileOutputFormat.setOutputPath(job, new Path(out));
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setOutputKeyClass(DoubleWritable.class);
    job.setOutputValueClass(Text.class);

    return job.waitForCompletion(true);

}

/**
 * Print the main an only help text in the System.out
 *
 * @param err an optional error message to display
 */
public static void printUsageText(String err) {

    if (err != null) {
        // if error has been given, print it
        System.err.println("ERROR: " + err + ".\n");
    }

    System.out.println("Usage: pagerank.jar " + KEY_INPUT + " <input>

```

```

" + KEY_OUTPUT + " <output>\n");
    System.out.println("Options:\n");
    System.out.println("    " + KEY_INPUT + "    (" + KEY_INPUT_ALIAS
+ ")    <input>    The directory of the input graph [REQUIRED]");
    System.out.println("    " + KEY_OUTPUT + "    (" + KEY_OUTPUT_ALIAS
+ ")    <output>    The directory of the output result [REQUIRED]");
    System.out.println("    " + KEY_DAMPING + "    (" + KEY_DAMPING_ALIAS
+ ")    <damping>    The damping factor [OPTIONAL]");
    System.out.println("    " + KEY_COUNT + "    (" + KEY_COUNT_ALIAS
+ ")    <iterations> The amount of iterations [OPTIONAL]");
    System.out.println("    " + KEY_HELP + "    (" + KEY_HELP_ALIAS +
")    Display the help text\n");
}

}

```

1.3 实验小结

PageRank 算法是由谷歌公司提出的用于评估网页重要性的算法。它基于图论的思想，将网页之间的链接关系建模为一个有向图，并通过迭代计算每个页面的重要性指标（即 PageRank 值）。PageRank 算法的核心思想是一个页面的重要性取决于指向它的页面的重要性。算法通过不断迭代计算，直到收敛为止，得到每个页面的 PageRank 值。

本次实验使用了 MapReduce 框架来实现 PageRank 算法。MapReduce 是一种用于大规模数据处理的编程模型，它将任务分割为多个并行的 Map 和 Reduce 步骤，通过分布式计算来加速处理过程。MapReduce 框架提供了自动的任务调度、数据分片和结果合并等功能，适合处理大规模数据集。

实验中采用了三个 MapReduce 作业来实现 PageRank 算法的计算过程。第一个作业用于初始化页面的排名值和链接信息，第二个作业用于根据链接关系计算页面的重要性，第三个作业用于对页面进行排序。每个作业都包含了 Mapper 和 Reducer 类，通过适当的数据解析、计算和输出来完成相应的任务。

实验的最终结果是按照页面排名对页面进行排序，并输出排序结果。通过 PageRank 算法的迭代计算和排序操作，可以得到每个页面的 PageRank 值和按照重要性排名的页面列表。

通过本次实验，我获得了以下收获：

- 熟悉了 PageRank 算法的原理和实现过程，加深了对网页重要性评估的理解。
- 熟悉了 MapReduce 框架的使用，掌握了大规模数据处理的基本方法。
- 理解了分布式计算的优势，可以通过并行处理大规模数据提高计算效率。

- 学会了使用 Mapper 和 Reducer 类来实现具体的计算任务，并根据需求进行数据解析和结果输出。
- 通过实验的实际操作，加深了对算法和框架的理论知识的理解，并提升了实际编程和调试的技巧。

实验 2: Spark 实验

2.1 实验概述

WordCount 实验的目的是通过编写和执行基于 Spark 的 WordCount 程序, 让学生掌握 Spark 编程的基本概念和函数式编程思想。实验内容包括使用 Scala 编写 WordCount 程序、在命令行中执行程序、使用 Eclipse 编译和打包程序, 以及查看程序执行结果。实验环境要求使用 Linux (Ubuntu) 操作系统, 安装 Java JDK 1.8 和 Spark 3.4, 并建议配置至少 4GB 内存和 100GB 硬盘空间。

Spark Streaming 实验的目的是通过编写和执行基于 Spark Streaming 的 WordCount 程序, 让学生了解 Spark Streaming 的工作原理。实验内容包括基于 Spark Streaming 编程模型实现 WordCount 程序的基本步骤, 例如创建 SparkSession 实例、创建 DataFrame 表示输入数据、进行 DataFrame 转换操作, 并创建 StreamingQuery 开启流查询。实验环境要求与 WordCount 实验相同。

通过完成这两个实验, 学生将能够熟悉 Spark 编程的基本概念, 掌握大规模数据处理的技巧, 并在分布式环境中运行 Spark 程序。这些实验还要求学生具备一台计算机或虚拟机, 并保证网络连接畅通以便下载所需的软件和文档。

总之, 这些实验旨在提供一个基于 Spark 的学习平台, 帮助学生深入理解和应用大数据处理框架, 为他们在数据分析和处理领域的学习和研究打下坚实的基础。

2.2 实验内容

本实验报告介绍了两个实验: WordCount 实验和 Spark Streaming 实验。在 WordCount 实验中, 我们使用 Scala 编写了一个 WordCount 程序, 通过掌握 RDD 和函数式编程思想, 实现了对文本数据的统计和计数。我们通过命令行执行该程序, 并使用 Eclipse 进行编译和打包操作。最后, 我们查看了程序的执行结果, 确保其按预期工作。

在 Spark Streaming 实验中, 基于 Spark Streaming 编程模型, 我们实现了一个 WordCount 程序。编写 Spark Streaming 程序的基本步骤包括创建 SparkSession 实例、创建 DataFrame 表示输入数据、进行 DataFrame 转换操作,

然后创建 `StreamingQuery` 开启流查询。最后，我们调用 `StreamingQuery.awaitTermination()` 方法等待流查询结束。

2.2.1 阶段1 WordCount 实验

本实验主要围绕 WordCount 实验展开，通过使用命令行和 Scala 代码来实现不同的 WordCount 场景，统计文本中单词的个数。

在第一部分，我们使用命令行执行 Scala 代码，通过 `spark-shell` 读取 Linux 系统本地文件“`file:///home/stu/software/hadoop/README.txt`”，并统计单词“Hadoop”的个数。我们使用 `SparkConf` 和 `SparkContext` 来创建 Spark 应用程序的配置和上下文，然后使用 `textFile` 方法读取文件内容，并使用 `count` 方法统计行数。通过 `filter` 和 `count` 方法，我们筛选出包含“Hadoop”的行，并统计它们的数量。

```
import org.apache.spark.{SparkConf, SparkContext}

<!-- import org.apache.spark.{SparkConf, SparkContext} -->

val sparkConf = new SparkConf().setAppName("MyApp").setMaster("local")
val sc = SparkContext.getOrCreate(sparkConf)
val textFile = sc.textFile("/data/ywxia/BigDataProcessingSystems-Experiments/lab2-spark/exp/exp2/README.md")
textFile.count()

val linesCountWithHadoop = textFile.filter(line => line.contains("Hadoop")).count()
```

图 2-1

在第二部分，我们在 `spark-shell` 中读取 HDFS 系统中的文件“`/user/hadoop/test.txt`”，并统计文件的行数。我们创建 `SparkConf` 和 `SparkContext`，并使用 `textFile` 方法读取文件内容，然后使用 `count` 方法统计行数。

```
import org.apache.spark.{SparkConf, SparkContext}

<!-- import org.apache.spark.{SparkConf, SparkContext} -->

val sparkConf = new SparkConf().setAppName("MyApp").setMaster("local")
val sc = SparkContext.getOrCreate(sparkConf)

val lines = sc.textFile("hdfs://localhost:9000/user/xyw/test.txt")
lines.count()
```

图 2-2

在第三部分，我们使用命令行实现 WordCount，即统计文件中各个单词的计数。我们创建 `SparkConf` 和 `SparkContext`，并使用 `textFile` 方法读取文件内容。

然后，我们使用 flatMap 方法将每行拆分为单词，并使用 map 方法将每个单词映射为键值对（单词，1）。最后，我们使用 reduceByKey 方法将相同单词的计数进行累加，并将结果保存到输出文件中。

```
import org.apache.spark.{SparkConf, SparkContext}
import java.io.PrintWriter

<!-- import org.apache.spark.{SparkConf, SparkContext} -->

val sparkConf = new SparkConf().setAppName("MyApp").setMaster("local")
val sc = SparkContext.getOrCreate(sparkConf)

val textFile = sc.textFile("/data/ywxia/BigDataProcessingSystems-Experiments/lab2-spark/exp/exp2/AChristmasCarol_CharlesDickens_Dutch.txt")
val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey((a,b) => a+b)

val writer = new PrintWriter("/data/ywxia/BigDataProcessingSystems-Experiments/lab2-spark/exp/exp2/output.txt")
wordCounts.collect().foreach(writer.println)
writer.close()
```

图 2-3

在第四部分，我们使用 Scala 语言实现 WordCount。我们创建一个名为 WordCountApp 的对象，并在 main 方法中编写实现代码。我们使用 Source.fromFile 方法打开文件，使用 getLines 方法按行读取文件内容。然后，我们使用 flatMap 方法将每行拆分为单词，并使用 toList 方法将结果转换为列表。接下来，我们使用 map 方法给每个单词映射一个初始计数值 1，并使用 groupBy 方法按单词进行分组。最后，我们使用 mapValues 方法将每个单词的计数值统计出来。最终，我们得到了每个单词及其对应的计数值。

```
import java.io.File
import scala.io.Source

object WordCountApp {

  def main(args: Array[String]): Unit = {
    //文件路径
    val filePath = "/data/ywxia/BigDataProcessingSystems-Experiments/lab2-spark/exp/exp2/AChristmasCarol_CharlesDickens_Dutch.txt"
    val codec = "utf-8"
    //打开文件
    val file = Source.fromFile(filePath, codec)
    val wc = file.getLines().flatMap(_.split("\t")).toList.map((_, 1)).groupBy((_,_)).mapValues(_.size)
    println(wc)
    // 关闭文件
    file.close()
  }
}
```

图 2-4

通过这些实验，我们掌握了使用命令行和 Scala 代码实现 WordCount 的方法。我们学会了使用 Spark 的 API 来读取文件、进行数据转换和聚合操作，以及将结果保存到输出文件中。这些技能对于大规模数据处理和分析非常重要，为学生在实际项目中应用 Spark 提供了基础。

2.2.2 阶段 2 Spark Streaming 实验

本实验主要涵盖了使用 Spark Streaming 编写并执行 WordCount 程序的步骤。我们将 WordCount 程序写成一个独立的 Scala 文件，并通过 Spark 进行执行。

首先，在终端中进入指定目录并创建相应的文件和文件夹结构。我们使用以下命令创建了 /usr/local/spark/mycode/streaming 目录，并在 src/main/scala 目录下创建了 TestStructuredStreaming.scala 文件。

然后，我们在 TestStructuredStreaming.scala 文件中编写了 WordCount 程序的代码。程序首先创建了 SparkSession 对象并设置应用程序的名称。然后，使用 spark.readStream 方法从指定的主机和端口读取实时数据流。我们将流数据按空格拆分成单词，并使用 groupBy 和 count 方法对单词进行分组和计数。最后，使用 writeStream 方法将计数结果输出到控制台，并调用 awaitTermination 方法等待流查询的结束。

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.SparkSession

object WordCountStructuredStreaming{
  def main(args: Array[String]){
    val spark = SparkSession.builder.appName("StructuredNetworkWordCount").getOrCreate()
    import spark.implicits._
    val lines = spark.readStream.format("socket").option("host","localhost").option("port",9998).load()

    val words = lines.as[String].flatMap(_.split(" "))
    val wordCounts = words.groupBy("value").count()

    val query = wordCounts.writeStream.outputMode("complete").format("console").start()
    query.awaitTermination()
  }
}
```

图 2-5

接下来，我们在 /usr/local/spark/mycode/streaming 目录下创建了 sbt 构建文件 simple.sbt，并添加了所需的依赖库。

完成构建文件后，我们使用 sbt package 命令执行打包和编译操作，生成可执行的 JAR 文件。

最后，我们使用 spark-submit 命令提交 Spark 应用程序，并指定主类和 JAR 文件的路径。执行该命令后，Spark 应用程序开始监听输入流，并在终端上输出“hello”和“world”单词的出现次数。

```
hadoop@LAPTOP-MLOEKGFH:~/BigDataProcessingSystems-Experiments-main/lab2-spark/exp/exp3/streaming$ nc -lk 9998
hello world
test hello world one
hello hello hello hello hello world one test
```

```
Batch: 4
```

| value | count |
|-------|-------|
| one | 2 |
| hello | 7 |
| world | 3 |
| | 1 |
| test | 2 |

图 2-6

通过这个实验，我们学习了使用 Spark Streaming 编写和执行实时数据处理程序的过程。我们使用 SparkSession 对象读取和处理实时数据流，并对数据进行分组和计数操作。这样的实时数据处理技术在大数据领域具有重要的应用价值，可以帮助我们对数据进行实时分析和决策。

2.3 实验小结

我学习了 Spark 和 Spark Streaming 的基本概念和原理。Spark 是一个快速、通用的大数据处理框架，通过内存计算和并行处理提供高效的数据处理能力。Spark Streaming 是 Spark 的流式处理模块，用于处理实时数据流和流式数据处理任务。

我学会了使用 Spark 的 API 进行大规模数据处理和分析。通过 Spark 的核心概念和 API，如 SparkConf、SparkContext、RDD 和 DataFrame，我们可以读取、转换和操作数据，进行各种数据处理和分析任务。

在 WordCount 实验部分，我们学会了使用命令行和 Scala 代码实现单词计数功能。通过使用 Spark 的 API，我们可以轻松地读取文本文件、拆分单词、进行聚合操作，并输出结果。

在 Spark Streaming 实验部分，我们学会了使用 Spark Streaming 实现实时数据处理任务。通过编写 Scala 代码，我们可以读取实时数据流、进行实时的分组和计数操作，并实时输出结果。

通过实验，我们成功实现了 WordCount 和实时数据处理的功能。我们能够统计文本中单词的个数，并实时处理和分析输入的数据流，输出相应的结果。这些结果为我们提供了对大规模数据进行处理和分析的能力，以及实时数据处理的解

决方案。

通过本次实验，我获得了以下收获：

- 掌握了使用 Spark 进行大规模数据处理和分析的基本技能。我们了解了 Spark 的核心概念和 API，学会了使用 Spark 进行数据读取、转换和聚合操作，以及将结果输出到不同的存储介质中。
- 理解了 Spark Streaming 的基本原理和使用方法。我们学会了使用 Spark Streaming 读取和处理实时数据流，进行实时的数据转换和计算，并输出实时结果。
- 增强了对大数据处理和实时数据处理的理解。通过实际操作和编码实现，我们对大数据处理和实时数据处理的流程、技术和挑战有了更深入的了解。
- 培养了独立解决问题的能力。通过实验，我们需要独立思考、编写代码并调试，从而培养了解决问题和开发实际项目的能力。

实验 3：图计算实验

3.1 实验概述

本实验旨在通过使用 GraphX 图计算系统，帮助学生深入理解图计算的工作原理，并掌握使用 GraphX 进行大规模图数据分析和处理的能力。实验环境为 Linux (Ubuntu) 操作系统，使用 JAVA 1.8 作为软件环境，并建议配置至少 4GB 内存和 100GB 的硬盘空间安装 Hadoop。实验要求具备互联网连接，以便下载所需软件和文档。

3.2 实验内容

GraphX 是 Apache Spark 的图计算库，可以实现并行化的图计算和图分析任务。通过本实验，学生将学习编写基于 GraphX 的 DFS（深度优先搜索）和 SCC（强连通分量）程序，并在 GraphX 系统中执行这些算法。DFS 算法用于图遍历和寻找连通分量，而 SCC 算法用于识别强连通分量。这些算法是图计算中常用的基本算法，对于图数据的分析和处理具有重要意义。

通过实验，学生将在实践中加深对图计算系统的理解，并能够将图算法应用于实际问题中。实验结果将提供宝贵的经验和技能，使学生能够应对大规模数据分析的挑战。实验报告将详细介绍实验步骤、方法和结果，并对实验结果进行分析和结论。

3.2.1 阶段 1 DFS

任务 1 旨在实现基于 GraphX 的深度优先搜索（DFS）算法。给定一个图，从指定的起始顶点开始，DFS 算法通过遍历图中的节点和边来探索整个图，并输出每个节点到起始顶点的距离。

```

def main(args: Array[String]): Unit = {
  // 设置 Spark 环境
  val conf = new SparkConf().setAppName("GraphDFS").setMaster("local")
  val sc = new SparkContext(conf)

  // 读取顶点和边的数据
  val vertices = sc.textFile("input/test-vertices.txt")
    .map { line =>
      val fields = line.split('\t')
      (fields(0).toLong, fields(1))
    }
  val edges = sc.textFile("input/test-edges.txt")
    .map { line =>
      val fields = line.split('\t')
      Edge(fields(0).toLong, fields(1).toLong, 0)
    }

  // 创建 Graph
  val graph = Graph(vertices, edges).mapVertices((id, label) => label.asInstanceOf[Any])

  // 执行深度优先搜索
  val startVertex = 0L
  val dfsResult = dfs(graph, startVertex)

  // 输出 DFS 结果
  dfsResult.foreach { case (vertex, distance) =>
    val distanceFromStart = if (vertex == startVertex) 0 else distance
    println(s"$vertex\t$distanceFromStart")
  }

  sc.stop()
}

def dfs(graph: Graph[Any, Int], startVertex: VertexId): Seq[(VertexId, Int)] = {
  var visited = Set[VertexId]()
  var stack = List[(VertexId, Int)]((startVertex, 0))
  var result = List[(VertexId, Int)]()

  while (stack.nonEmpty) {
    val (currentVertex, distance) = stack.head
    stack = stack.tail

    if (!visited.contains(currentVertex)) {
      visited += currentVertex
      val neighbors = graph.edges.filter(_.srcId == currentVertex).map(_.dstId).collect()
      stack = neighbors.map((_, distance + 1)).toList ::: stack
      result ::= (currentVertex, distance)
    }
  }

  result.reverse
}

```

图 3-1

首先，我们使用 SparkConf 和 SparkContext 设置 Spark 环境。

然后，我们读取输入文件中的顶点和边数据，将其转换为对应的数据结构。

接下来，通过调用 Graph 构造函数创建一个 Graph 对象，其中顶点使用字符串标识，并将边的权重初始化为 0。

在 DFS 函数中，我们使用一个栈来保存需要遍历的节点，初始时将起始顶点入栈。

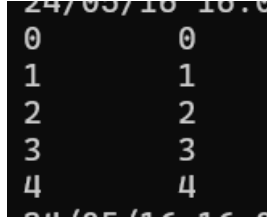
在每次迭代中，我们从栈中弹出一个节点，并检查它是否已被访问过。如果未被访问，则将其标记为已访问，并获取其邻居节点。

对于每个邻居节点，我们将其与起始顶点的距离加一，并将其加入栈中，以

便后续遍历。

同时，我们将当前节点与起始顶点的距离保存在结果列表中。

最后，返回结果列表，其中包含每个节点与起始顶点的距离。



| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |

图 3-2

根据 DFS 结果，我们可以看到每个节点与起始顶点的距离。距离表示从起始顶点到达该节点所需的最短路径长度。输出顺序表示 DFS 访问节点的顺序，通过分析距离，我们可以了解图的结构以及节点之间的关系。

3.2.2 阶段 2 SCC

任务 2 旨在实现基于 GraphX 的强连通分量（SCC）算法。给定一个图，SCC 算法将图中的节点划分为强连通分量，并输出每个强连通分量的节点列表。强连通分量是指在图中，任意两个节点之间都存在路径的节点子集。

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
import org.apache.spark.graphx.lib.StronglyConnectedComponents
import org.apache.spark.{SparkConf, SparkContext}

object GraphSCC {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("GraphSCC").setMaster("local")
    val sc = new SparkContext(conf)

    val vertices: RDD[(VertexId, String)] = sc.textFile("input/test-vertices.txt")
      .map { line =>
        val fields = line.split('\t')
        (fields(0).toLong, fields(1))
      }
    val edges: RDD[Edge[Int]] = sc.textFile("input/test-edges.txt")
      .map { line =>
        val fields = line.split('\t')
        Edge(fields(0).toLong, fields(1).toLong, 0)
      }

    val graph: Graph[String, Int] = Graph(vertices, edges)

    val sccResult: Graph[VertexId, Int] = graph.stronglyConnectedComponents(100)

    val sccCount: Long = sccResult.vertices.map { case (_, componentId) => componentId }.distinct().count()

    val group = sccResult.vertices.map {
      case (vertexId, minId) => (minId, vertexId)
    }.groupByKey()

    group.foreach(println)

    println(s"Number of strongly connected components: $sccCount")

    sc.stop()
  }
}
```

图 3-3

首先，我们使用 SparkConf 和 SparkContext 设置 Spark 环境。

然后，我们读取输入文件中的顶点和边数据，将其转换为对应的数据结构。

接下来，通过调用 Graph 构造函数创建一个 Graph 对象，其中顶点使用字符串标识，并将边的权重初始化为 0。

使用 Graph 对象的 stronglyConnectedComponents 方法执行 SCC 算法，并得到每个节点的强连通分量标识。

对于算法结果，我们计算强连通分量的数量，并输出每个强连通分量的节点列表。

```
(0,CompactBuffer(4, 0, 1, 3, 2))
(10,CompactBuffer(13, 19, 15, 16, 11, 14, 17, 12, 18, 10))
(5,CompactBuffer(6, 7, 9, 8, 5))
24/05/16 16:24:41 INFO Executor: Finished task 0.0 in stage
24/05/16 16:24:41 INFO TaskSetManager: Finished task 0.0 in
24/05/16 16:24:41 INFO TaskSchedulerImpl: Removed TaskSet 12
24/05/16 16:24:41 INFO DAGScheduler: ResultStage 1219 (forea
24/05/16 16:24:41 INFO DAGScheduler: Job 31 is finished. Can
24/05/16 16:24:41 INFO TaskSchedulerImpl: Killing all runnin
24/05/16 16:24:41 INFO DAGScheduler: Job 31 finished: foreac
Number of strongly connected components: 3
```

图 3-4

根据 SCC 算法的结果，我们可以看到每个强连通分量的节点列表和强连通分量的数量。每个强连通分量由一个标识符和节点列表组成，强连通分量的 ID 是其中包含的最小节点的 ID。通过分析结果，我们可以了解图中的强连通分量数量以及每个强连通分量中的节点。

3.3 实验小结

在本次实验中，我们使用了图计算库 GraphX 和 Apache Spark 来实现深度优先搜索（DFS）算法和强连通分量（SCC）算法。以下是对本次实验使用的理论、技术、方法和结果的总结，并描述了通过实验获得的收获。

DFS 算法：DFS 是一种用于图遍历的算法，通过深度优先搜索的方式遍历图中的节点。它是一种递归的算法，从起始顶点开始，沿着路径尽可能深入，直到达到不能继续前进的节点，然后回溯到上一个节点，并继续遍历其他路径。DFS 算法可以用于计算节点之间的距离和路径，以及判断图的连通性。

SCC 算法：SCC 算法用于将图中的节点划分为强连通分量。强连通分量是指在图中，任意两个节点之间都存在路径的节点子集。SCC 算法通过深度优先搜索

和 Tarjan 算法来识别和合并强连通分量。它可以用于分析图的结构和节点之间的关系。

GraphX: GraphX 是 Apache Spark 中的图计算库，提供了处理大规模图数据的功能。它基于分布式计算框架 Spark，使用分布式图模型来表示和处理图数据。GraphX 提供了一系列图算法和操作，包括 DFS 和 SCC 等，用于对图数据进行分析 and 处理。

DFS 算法的结果是每个节点与起始顶点的距离。它提供了从起始顶点到每个节点的最短路径长度。

SCC 算法的结果是将图中的节点划分为强连通分量。它提供了每个强连通分量的节点列表和强连通分量的数量。

通过完成本次实验，我获得了以下收获：

- 理解了 DFS 算法和 SCC 算法的原理和应用场景。DFS 算法可用于图的遍历和计算节点之间的距离，而 SCC 算法可用于识别图中的强连通分量。
- 学习了如何使用 GraphX 和 Spark 来处理大规模图数据。GraphX 提供了方便的图数据表示和图算法实现，并通过 Spark 的分布式计算能力加速了图计算的执行。
- 熟悉了使用 SparkConf 和 SparkContext 配置 Spark 环境的过程，以及读取和处理外部数据文件的方法。
- 掌握了基于 GraphX 的 DFS 算法和 SCC 算法的实现方式，并学会了分析和解释算法结果。
- 通过实验，加深了对图分析的理解，并了解了如何应用图计算算法来发现图数据中的模式和特征。

实验总结

本次实验涵盖了 MapReduce、Spark 和图计算三个方面的实验，旨在帮助学生深入理解大数据处理框架和技术，并为他们在数据分析和处理领域打下坚实的基础。通过完成这些实验，我获得了以下几点收获和体会。

首先，通过 MapReduce 实验，我深入理解了 MapReduce 编程模型的工作原理。通过编写和执行基于 MapReduce 的 PageRank 程序，我学会了如何使用 Hadoop 框架进行大规模数据处理。我掌握了 MapReduce 编程的基本概念和技巧，并熟悉了在分布式环境中运行 MapReduce 程序的流程和要点。

其次，Spark 实验让我对 Spark 编程有了更深入的了解。通过编写 WordCount 程序和 Spark Streaming 程序，我掌握了 Spark 编程的基本概念和函数式编程思想。我学会了使用 Scala 编写 Spark 程序，并通过命令行和 Eclipse 进行程序的执行、编译和打包操作。这些实验使我熟悉了大规模数据处理的技巧，以及在分布式环境中运行 Spark 程序的方法。

最后，图计算实验让我了解了 GraphX 图计算系统的工作原理。通过编写基于 GraphX 的 DFS 和 SCC 程序，我学会了使用图计算库进行并行化的图计算和图分析任务。我掌握了 DFS 算法和 SCC 算法的实现方法，以及在 GraphX 系统中执行这些算法的步骤和技巧。这些图计算实验对于图数据的分析和处理具有重要意义，并为我将图算法应用于实际问题提供了宝贵的经验和技能。

通过这些实验，我不仅提升了对 MapReduce、Spark 和图计算的理论理解，还在实践中加深了对这些大数据处理框架和技术的实际应用能力。我学会了如何处理大规模数据集，使用分布式计算框架进行高效的数据处理和分析。同时，我也意识到了大数据处理中的挑战和注意事项，比如数据集的规模、计算效率和结果可靠性等方面。

综上所述，这些实验为我提供了一个全面的学习平台，使我能够更深入地理解和应用大数据处理框架。我相信这些经验和技能将对我的未来学习和研究工作产生积极的影响，并为我数据分析和处理领域的发展奠定坚实的基础。我将继续学习相关的技术和工具，不断提升自己在大数据处理领域的能力和水平。