第三章　批处理系统MapReduce

徐 辰
cxu@dase.ecnu.edu.cn
華東師範大學

DaSE Data Science & Engineering

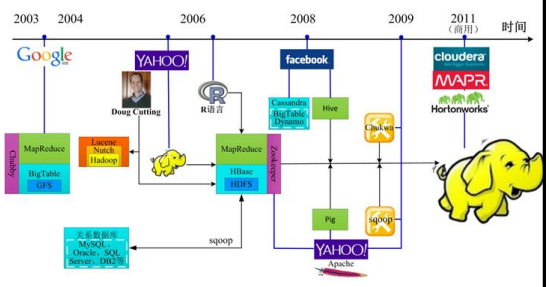
Hadoop简介

2

- Hadoop是Apache软件基金会旗下的一个开源分布式计算平台，为用户提供了系统底层细节透明的分布式基础架构
- Hadoop是基于Java语言开发的，具有很好的跨平台特性，并且可以部署在廉价的计算机集群中
- Hadoop的核心是分布式文件系统HDFS（Hadoop Distributed File System）和MapReduce
- Hadoop被公认为行业大数据标准开源软件，在分布式环境下提供了海量数据的处理能力

Hadoop生态圈发展路线

3



大纲

4

- 设计思想
  - MPI与MapReduce
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

MPI(Message Passing Interface)编程简介

5

- MPI是一个信息传递应用程序接口，包括协议和和语义说明
- 常用接口
  - MPI_Init(…) 并行环境初始化
  - MPI_Comm_size(…) 获得进程个数 size
  - MPI_Comm_rank(…) 获取进程的rank值
  - MPI_Send(…) 发送消息
  - MPI_Recv(…) 获取消息
  - MPI_Finalize() 退出MPI环境

MPI编程举例

6

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
void main(int argc, char* argv[])
{
    int numprocs, myid, source;
    MPI_Status status;
    char message[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    if (myid != 0) {  //非0号进程发送消息
        strcpy(message, "Hello World!");
        MPI_Send(message, strlen(message) + 1, MPI_CHAR, 0, 99,
            MPI_COMM_WORLD);
    }
    else {  // myid == 0，即0号进程接收消息
        for (source = 1; source < numprocs; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, 99,
                MPI_COMM_WORLD, &status);
            printf("接收到第%d号进程发送的消息：%s\n", source, message);
        }
    }
    MPI_Finalize();
} /* end main */
```

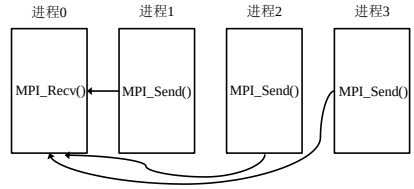
运行MPI程序

7

```
PS F:\并行计算学习\pi\x64\Debug> mpiexec.exe -n 4 .\send_recv.exe
接收到第1号进程发送的消息，Hello World!
接收到第2号进程发送的消息，Hello World!
接收到第3号进程发送的消息，Hello World!
PS F:\并行计算学习\pi\x64\Debug> mpiexec.exe -n 8 .\send_recv.exe
接收到第1号进程发送的消息，Hello World!
接收到第2号进程发送的消息，Hello World!
接收到第3号进程发送的消息，Hello World!
接收到第4号进程发送的消息，Hello World!
接收到第5号进程发送的消息，Hello World!
接收到第6号进程发送的消息，Hello World!
接收到第7号进程发送的消息，Hello World!
PS F:\并行计算学习\pi\x64\Debug>
```

MPI程序工作示意

8



MPI的局限性

9

- 从用户编程的角度来看，程序员需要考虑到进程之间的并行问题，并且进程之间的通信需要用户在程序中显式地表达，这无疑增加了程序员编程的复杂性。
- 从系统实现的角度来看，MPI程序是以多进程方式运行的。如果在运行过程中某一进程因故障导致崩溃，那么除非用户在编写程序时添加了故障恢复的功能，否则MPI编程框架本身并不能提供容错能力。

大纲

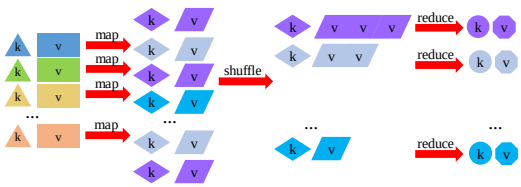
10

- 设计思想
  - MPI与MapReduce
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

数据模型

11

- 将数据抽象为一系列键值对，在处理过程中对键值对进行转换



大纲

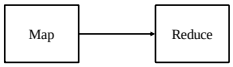
12

- 设计思想
  - MPI与MapReduce
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

逻辑计算模型

13

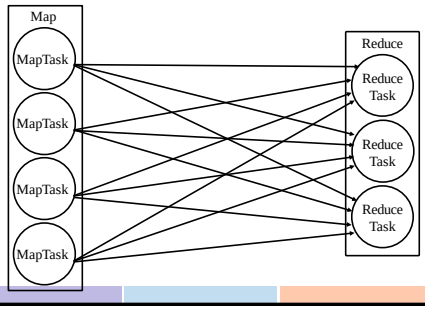
- 抽象为Map和Reduce两个过程
  - Map的过程将输入键值对进行一次变换，产生若干个新的键值对，Map转换前后的键值对的内容通常都会不同
  - Reduce过程会对相同键的键值对进行计算，并可根据需要将计算结果进行一次键值对转换后输出
- 有向无环图（Directed Acyclic Graph, DAG）



物理计算模型

14

- 采用“分而治之”策略，由多个任务并行处理



用户编程容易

15

- 用户不需要掌握分布式并行编程细节

```
Class X {
    map() {              //map方法的实现
      …
    }

    reduce() {           //reduce方法的实现

    }

    main(){
       Job job = …       //定义分布式作业
       job.config =      //作业参数设置
    }
}
```

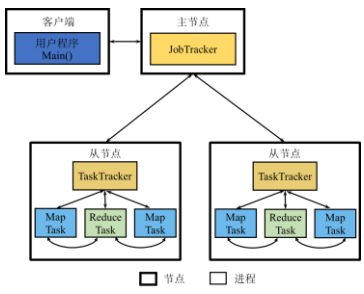
大纲

16

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

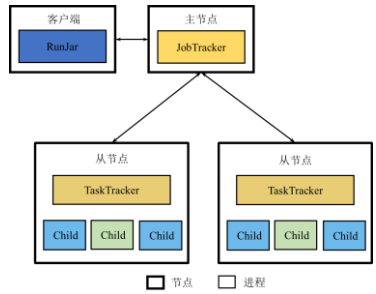
抽象架构图

17



Hadoop MapReduce架构图

18



JobTracker

19

- 资源管理
  - 通过监控TaskTracker来管理系统拥有的计算资源
- 作业管理
  - 负责将作业（Job）拆分成任务（Task），并进行任务调度以及跟踪任务的运行进度、资源使用量等信息

TaskTracker

20

- 管理本节点的资源
  - TaskTracker使用slot等量划分本节点上的资源量（CPU、内存等）
- 执行JobTracker的命令
  - 接收JobTracker发送过来的命令并执行（如启动新Task、杀死Task等）
- 向JobTracker汇报情况
  - 通过心跳将本节点上资源使用情况和任务运行进度汇报给JobTracker

Task

21

- 任务执行
  - JobTracker根据TaskTracker汇报的信息进行调度，命令存在空闲slot的TaskTracker启动Task进程执行map或reduce任务
  - 在Hadoop MapReduce的实现中该进程的名称为Child

Client

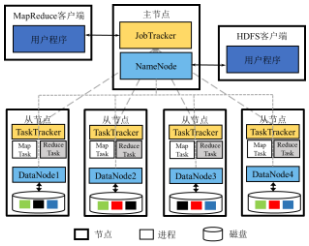
22

- 提交作业
  - 用户编写的MapReduce程序通过Client提交到JobTracker
  - 用户可通过Client提供的一些接口查看作业运行状态
  - 在Hadoop MapReduce的实现中，该进程的名称为RunJar

MapReduce与HDFS关系

23

- 计算与存储相分离
- 计算向数据靠拢，而不是数据向计算靠拢



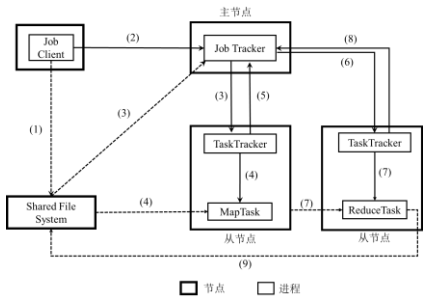
大纲

24

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

应用程序执行流程

25



应用程序执行流程

26

① Client将用户编写的MapReduce作业的配置信息、jar包等信息上传到共享文件系统，通常是HDFS。
② Client提交作业给JobTracker，即告知作业信息的位置。
③ JobTracker读取作业的信息，生成一系列Map和Reduce任务，调度给拥有空闲slot的TaskTracker。
④ TaskTracker根据JobTacker的指令启动Child进程执行Map任务，Map任务将从共享文件系统读取输入数据。
⑤ JobTracker从TaskTracker处获得Map任务进度信息。
⑥ 一旦Map任务完成后，JobTacker将Reduce任务分发给TaskTracker。
⑦ TaskTracker根据JobTacker的指令启动Child进程执行Reduce任务，Reduce任务将从Map任务所在节点的本地磁盘拉取Map的输出结果。
⑧ JobTracker从TaskTracker处获得Reduce任务进度信息。
⑨ 当Reduce任务运行结束并将结果写入共享文件系统，则意味着整个作业执行完毕。

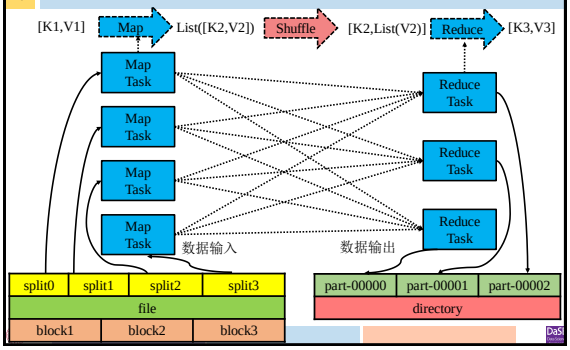
大纲

27

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Map-Shuffle-Reduce

28



大纲

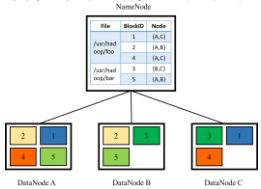
29

- 设计思想
- 体系架构
- 工作原理
  - 数据输入
  - Map阶段
  - Shuffle阶段
  - Reduce阶段
  - 数据输出
- 容错机制
- 编程示例

数据输入

30

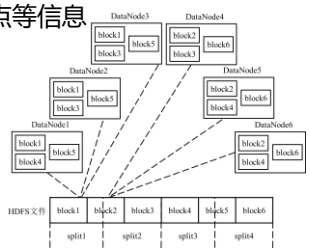
- 从存储系统中文件与Map任务可处理的键值对记录之间的映射
  - 输入文件的格式
- 问题：文件分块存储，可能存在跨块记录



Split vs. Block

31

- Split相对block而言是逻辑概念，包含一些元信息，如数据起始位置、数据长度、数据所在节点等信息



InputFormat

32

- 数据逻辑划分
  - InputFormat根据预定义格式将输入数据在逻辑上划分为若干个Split（切片）
  - Map任务读取的单位是Split，而不是物理的文件块
  - Split的数量往往决定了Map任务的个数，一个Split的数据一般由一个Map任务来处理
- 键值对解析
  - 给定一个Split，InputFormat将根据分隔符、大小等元信息将Split中的数据解析为相应键值对

常见的InputFormat

33

- TextInputFormat
- KeyValueTextInputFormat
- NLineInputFormat
- CombineTextInputFormat
- …
- 自定义InputFormat

大纲

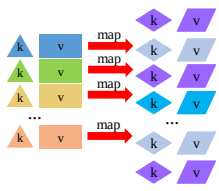
34

- 设计思想
- 体系架构
- 工作原理
  - 数据输入
  - Map阶段
  - Shuffle阶段
  - Reduce阶段
  - 数据输出
- 容错机制
- 编程示例

Map逻辑过程

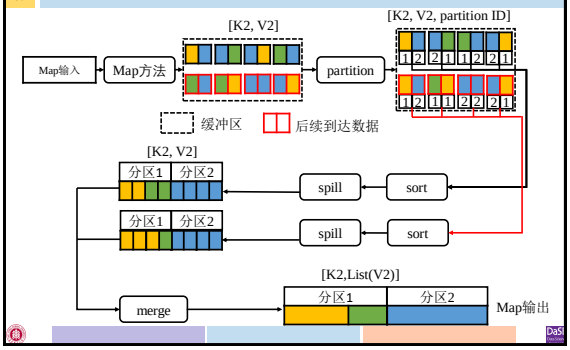
35

- [k1, v1] → List([k2, v2])



Map物理过程

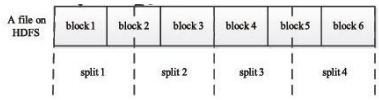
36



Map任务的数量

37

- MapReduce为每个split创建一个Map任务，split 的多少决定了Map任务的数目
- mapred.map.tasks设置程序员期望的map个数



大纲

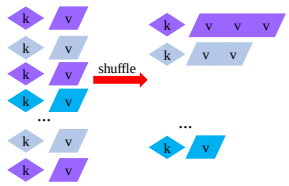
38

- 设计思想
- 体系架构
- 工作原理
  - 数据输入
  - Map阶段
  - Shuffle阶段
  - Reduce阶段
  - 数据输出
- 容错机制
- 编程示例

Shuffle逻辑过程

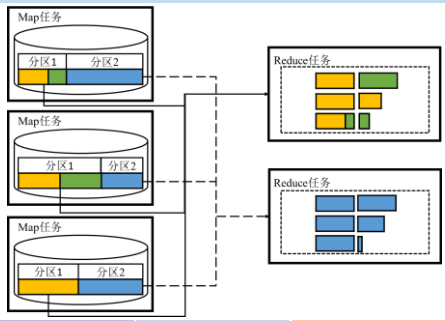
39

- List([k2, v2]) → [k2, List(v2)]



Shuffle物理过程

40



何时Shuffle？

41

- 当系统中的Map任务完成率达到设定阈值时，系统将启动Reduce任务
  - 例如，阈值设定为60%意味着如果系统中共有100个Map任务，那么一旦有60个Map任务已经完成了就可以启动Reduce任务，而不必等到这100个Map任务全部完成
- Reduce任务不会等到所有的Map任务执行结束才拉取Map任务的输出结果，但是拉取的数据必然来自于已经完成运行的Map任务，即已经保存在磁盘上的文件

大纲

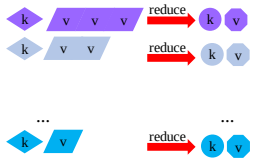
42

- 设计思想
- 体系架构
- 工作原理
  - 数据输入
  - Map阶段
  - Shuffle阶段
  - Reduce阶段
  - 数据输出
- 容错机制
- 编程示例

Reduce逻辑过程

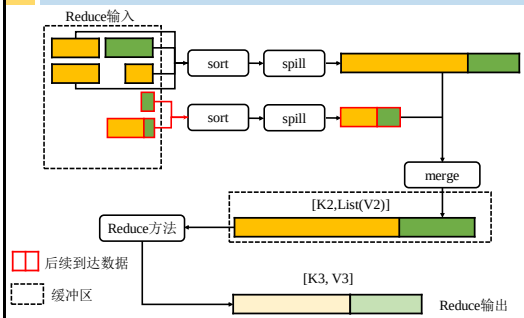
43

- [k2, List(v2)] → [k3, v3]



Reduce物理过程

44



Reduce任务的数量

45

- 程序指定
- 最优的Reduce任务个数取决于集群中可用的reduce任务槽(slot)的数目
- 通常设置比reduce任务槽数目稍微小一些的Reduce任务个数

大纲

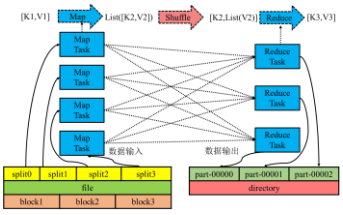
46

- 设计思想
- 体系架构
- 工作原理
  - 数据输入
  - Map阶段
  - Shuffle阶段
  - Reduce阶段
  - 数据输出
- 容错机制
- 编程示例

数据输出

47

- 每个Reduce任务的输出结果将以一个文件的形式保持到指定的目录当中
  - MapReduce输出结果是一组文件



OutputFormat

48

- 与数据输入阶段相反，MapReduce需要定义输出文件的格式，即OutputFormat
  - 包括分隔符等元信息
  - 从MapReduce程序处理的逻辑键值对数据到物理存储之间的映射
  - MapReduce系统将Reduce任务处理产生的结果按OutputFormat定义的格式写入HDFS等

常见的OutputFormat

49

- TextOutputFormat
- NullOutputFormat
- LazyOutputFormat
- …
- 自定义OutputFormat

大纲

50

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

MapReduce故障类型

51

- 主节点故障
  - JobTracker故障：如宕机引起
- 从节点故障
  - TaskTracker故障：如节点宕机引起
  - Task故障: 如JVM内存不够退出
- MapReduce容错和HDFS容错是两回事

大纲

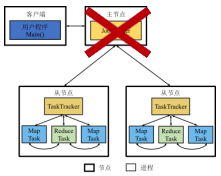
52

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - JobTracker故障
  - TaskTracker故障
  - Task故障
- 编程示例

JobTracker故障

53

- 对于MapReduce 1.0的架构，JobTracker故障意味着所有作业需要重新执行
- MapReduce 1.0没有处理JobTracker故障的机制，因而成为单点瓶颈



大纲

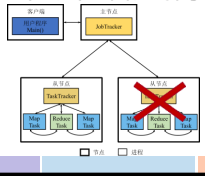
54

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - JobTracker故障
  - TaskTracker故障
  - Task故障
- 编程示例

TaskTracker故障

55

- JobTracker不会接收到“心跳”
- JobTracker会安排其他TaskTracker重新运行失败TaskTracker的任务
- 这个过程对于用户来说是透明的，只会感觉到该作业在执行某段时间里变慢了而已



大纲

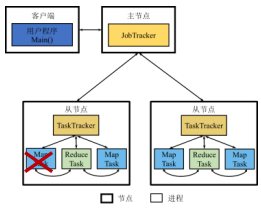
56

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - JobTracker故障
  - TaskTracker故障
  - Task故障
- 编程示例

Task故障

57

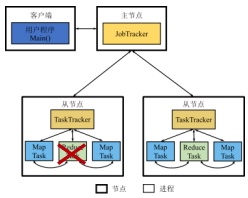
- Map Task故障
  - 重新执行Map任务
  - 去HDFS重新读入数据



Task故障

58

- Reduce Task故障
  - 重新执行Reduce任务
  - 去哪里重新读入数据?



Task故障

59

- 典型例子
  - Map任务或Reduce任务代码异常
- 当一个任务经过最大尝试次数运行后仍然失败，那么整个作业将被标记为失败

大纲

60

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Map方法框架

61

TextInputFormat→[LongWritable, Text]

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

/* 步骤1：确定输入键值对[K1,V1]和输出键值对[K2,V2]的数据类型 */
public class CustomMapper extends Mapper<K1的数据类型, V1的数据类型, K2的数据类型, V2的数
    据类型> {

  @Override
  protected void map(K1的数据类型 key, V1的数据类型 value, Context
      context) throws IOException, InterruptedException {
    /* 步骤2：编写处理逻辑将[K1,V1]转换为[K2,V2]对并输出 */
    ......
    context.write(K2, V2)
  }
}
```

Reduce方法框架

62

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

/* 步骤1：确定输入键值对[K2,List(V2)]和输出键值对[K3,V3]的数据类型 */
public class CustomReducer extends Reducer<K2的数据类型, V2的数据类型, K3的数据类型, V3的
    数据类型> {

  @Override
  protected void reduce(K2的数据类型 key, Iterable<V2的数据类型> values, Context
      context) throws IOException, InterruptedException {
    /* 步骤2：编写处理逻辑将[K2,List(V2)]转换为[K3,V3]对并输出 */
    ......
    context.write(K3, V3)
  }
}
```

主方法框架

63

```
public int run(String[] args) throws Exception {
  /* 步骤1：设置作业的信息 */
  ......
  Job job = Job.getInstance(getConf(), getClass().getSimpleName());
  // 设置程序的类名
  job.setJarByClass(getClass());

  // 设置数据的输入输出路径
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  // 设置数据的输入输出格式
  job.setInputFormatClass(输入格式类);
  job.setOutputFormatClass(输出格式类);

  // 设置map和reduce方法
  job.setMapperClass(CustomMapper.class);
  job.setReducerClass(CustomReducer.class);

  // 设置map方法的输出键值对数据类型
  job.setMapOutputKeyClass(map方法的输出键数据类型);
  job.setMapOutputValueClass(map方法的输出值数据类型);
  // 设置reduce方法的输出键值对数据类型
  job.setOutputKeyClass(reduce方法的输出键数据类型);
  job.setOutputValueClass(reduce方法的输出值数据类型);
  ......

  return job.waitForCompletion(true) ? 0 : 1;
}

public static void main(String[] args) throws Exception {
  /* 步骤2：运行作业 */
  ......
  int exitCode = ToolRunner.run(new CustomJob(), args);
  System.exit(exitCode);
}
```

大纲

64

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 关系表自然连接及其优化
  - 网页链接排名
  - K均值聚类

词频统计

65

- 输入：一个包含大量单词的文本文件
- 输出：
  - 文件中每个单词及其出现次数（频数）
  - 每个单词和其频数占一行，单词和频数之间有间隔

输入	输出
An An My Me An An My He My My An My	An 5 He 1 Me 1 My 5

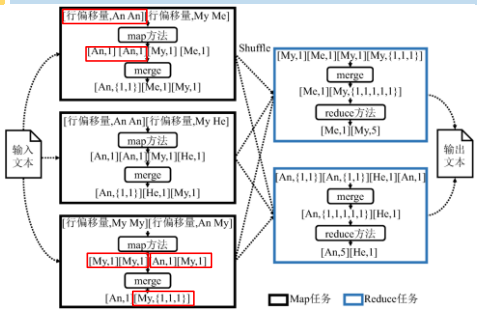
解决方案

66

- Map过程：
  - 把文本的每行内容转换为键值对[单词,1]
- Reduce过程：
  - 单词相同的键值对被发送到同一个Reduce中
  - 对单词相同的键值对进行计数
  - 输出计数后的结果[单词，频数]

词频统计运行过程

67



编写map方法

68

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
......

/* 步骤1：确定输入键值对[K1, V1]的数据类型为[LongWritable, Text]，输出键值对[K2, V2]的数据
    类型为[Text, IntWritable] */
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    /* 步骤2：编写处理逻辑将[K1, V1]转换为[K2, V2]并输出 */
    // 以空格作为分隔符拆分成单词
    String[] words = value.toString().split(" ");
    for (String word : words) {
      // 输出分词结果
      context.write(new Text(word), new IntWritable(1));
    }
  }
}
```

编写reduce方法

69

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

/* 步骤1：确定输入键值对[K2, List(V2)]的数据类型为[Text, IntWritable]，输出键值对[K3,
    V3]的数据类型为[Text, IntWritable] */
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
  @Override
  protected void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
    /* 步骤2：编写处理逻辑将[K2, List(V2)]转换为[K3, V3]并输出 */
    int sum = 0;
    // 遍历累加求和
    for (IntWritable value : values) {
      sum += value.get();
    }
    // 输出计数结果
    context.write(key, new IntWritable(sum));
  }
}
```

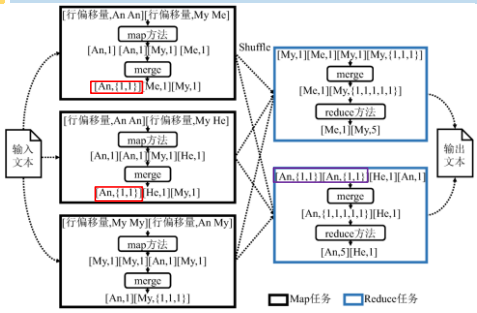
编写主方法

70

```
public class WordCount extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    /* 步骤1：设置作业的信息 */
    Job job = Job.getInstance(getConf(), getClass().getSimpleName());
    // 设置程序的类名
    job.setJarByClass(getClass());

    // 设置数据的输入输出路径
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    // 设置Map和Reduce方法
    job.setMapperClass(WordCountMapper.class);
    job.setReducerClass(WordCountReducer.class);

    // 设置Map方法的输出键值对数据类型
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    // 设置Reduce方法的输出键值对数据类型
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    /* 步骤2：运行作业 */
    int exitCode = ToolRunner.run(new WordCount(), args);
    System.exit(exitCode);
  }
}
```

词频统计运行过程

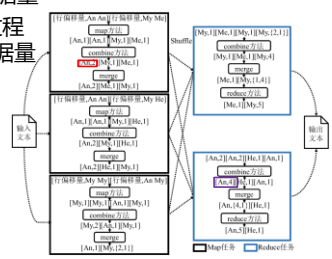
71



优化方案

72

- 使用combine方法
  - 减少Shuffle数据量
  - 减少Reduce过程需要处理的数据量



编写combine方法

73

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

/* 步骤1：确定输入键值对[K2, List(V2)]的数据类型为[Text, IntWritable]，输出键值对[K3,
   V3]的数据类型为[Text, IntWritable] */
public class WordCountCombiner extends Reducer<Text, IntWritable, Text, IntWritable> {
  @Override
  protected void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
    /* 步骤2：将[K2, List(V2)]合并为[K3, V3]并输出 */
    int sum = 0;
    // 进行合并操作
    for (IntWritable value : values) {
      sum += value.get();
    }
    // 输出合并的结果
    context.write(key, new IntWritable(sum));
  }
}
```

修改主方法

74

```
......
public class WordCount extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    ......
    job.setCombinerClass(WordCountCombiner.class);
    ......
  }

  ......
}
```

设置combine方法

大纲

75

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 关系表自然连接及其优化
  - 网页链接排名
  - K均值聚类

关系表的自然连接

76

- 输入：两个CSV文件，分别保存雇员表和部门表
- 输出：雇员表和部门表的自然连接结果

雇员表

Name	EmpId	DeptName
Harry	3415	会计
Sally	2241	销售
George	3401	会计
Harriet	2202	销售
Bart	6077	会计
Elise	9263	销售
Gemma	3870	会计
Tyler	6236	销售
Camden	4527	销售
Sofia	2035	会计

部门表

DeptName	Manager
会计	George
销售	Harriet

雇员表⋈部门表

Name	EmpId	DeptName	Manager
Sofia	2035	会计	George
Gemma	3870	会计	George
Bart	6077	会计	George
George	3401	会计	George
Harry	3415	会计	George
Camden	4527	销售	Harriet
Tyler	6236	销售	Harriet
Elise	9263	销售	Harriet
Harriet	2202	销售	Harriet
Sally	2241	销售	Harriet

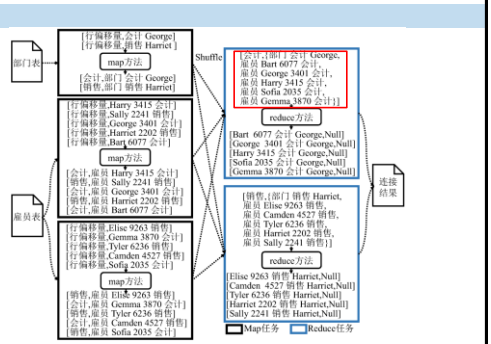
解决方案

77

- Map过程：　标记来自哪个关系表
  - 把来自雇员表的每个元组A转换成键值对[DeptName, 雇员 A]
  - 把来自部门表的每个元组B转换成键值对[DeptName, 部门 B]
- Reduce过程：
  - 具有相同DeptName值的元组被发送到同一Reduce中
  - 来自雇员表和部门表的具有相同DeptName值的元组进行连接
  - 输出连接后的元组

关系表自然连接的运行过程

78



编写自定义ReduceJoinWritable

79

```
public class ReduceJoinWritable implements Writable {
  // 保存雇员表或部门表元组
  private String data;
  // 标识当前对象保存的元组来自雇员表还是部门表
  private String tag;

  // 用于标识的常量
  public static final String EMPLOYEE = "1";
  public static final String DEPARTMENT = "2";

  @Override
  public void write(DataOutput dataOutput) throws IOException {
    dataOutput.writeUTF(tag);
    dataOutput.writeUTF(data);
  }

  @Override
  public void readFields(DataInput dataInput) throws IOException {
    tag = dataInput.readUTF();
    data = dataInput.readUTF();
  }

  // get和set方法
  ......
}
```

自定义数据类型来保存标识和元组两类信息

编写map方法

80

```
/* 步骤1：确定输入键值对[K1, V1]的数据类型为[LongWritable, Text]，确定输出键值对[K2, V2]的
   数据类型为[Text, ReduceJoinWritable] */
public class ReduceJoinMapper extends Mapper<LongWritable, Text, Text,
    ReduceJoinWritable> {

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    /* 步骤2：编写处理逻辑将[K1, V1]转换为[K2, V2]并输出 */
    // 获取输入键值对所属的split
    FileSplit split = (FileSplit) context.getInputSplit();
    // 通过Split获取键值对所属的文件路径
    String path = split.getPath().toString();
    ReduceJoinWritable writable = new ReduceJoinWritable();
    // 用writable的data属性记录
    writable.setData(value.toString());
    // 以制表符为分隔符切分元组，方便获取DeptName属性值
    String[] datas = value.toString().split("\t");
    // 通过输入数据所属文件路径判断datas的来源并进行分类处理
    if (path.contains(Constants.EMPLOYEE)) {
      // 标识data保存的元组来自雇员表
      writable.setTag(ReduceJoinWritable.EMPLOYEE);
      // 用DeptName属性值为键输出结果
      context.write(new Text(datas[2]), writable);
    } else if (path.contains(Constants.DEPARTMENT)) {
      // 标识data中保存的元组来自部门表
      writable.setTag(ReduceJoinWritable.DEPARTMENT);
      // 用DeptName属性值为键输出结果
      context.write(new Text(datas[0]), writable);
    }
  }
}
```

获取键值对所属的文件路径，并利用路径对键值对进行分类处理

编写reduce方法

81

```
/* 步骤1：确定输入键值对[K2, List(V2)]的数据类型为[Text, ReduceJoinWritable]，确定输出键值
   对[K3, V3]的数据类型为[Text, NullWritable] */
public class ReduceJoinReducer extends Reducer<Text, ReduceJoinWritable, Text,
    NullWritable> {

  @Override
  protected void reduce(Text key, Iterable<ReduceJoinWritable> values, Context context)
      throws IOException, InterruptedException {
    /* 步骤2：编写处理逻辑将输入键值对[K2, List(V2)]转换为[K3, V3]并输出*/
    List<String> employees = new ArrayList<>();
    List<String> departments = new ArrayList<>();
    // 分离values集合中雇员表和部门表元组
    for (ReduceJoinWritable value : values) {
      //获取ReduceJoinWritable对象的标识
      String tag = value.getTag();
      if (tag.equals(ReduceJoinWritable.EMPLOYEE)) {
        employees.add(value.getData());
      } else if (tag.equals(ReduceJoinWritable.DEPARTMENT)) {
        departments.add(value.getData());
      }
    }

    // 进行连接操作并输出连接结果
    for (String employee : employees) {
      for (String department : departments) {
        String[] datas = department.split("\t");
        // 不重复输出DeptName属性值
        String result = employee + "\t" + datas[1];
        context.write(new Text(result), NullWritable.get());
      }
    }
  }
}
```

从输入值中分离元组，并执行连接操作

编写主方法

82

```
public class ReduceJoin extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    /* 步骤1：设置作业的信息 */
    Job job = Job.getInstance(getConf(), getClass().getSimpleName());
    // 设置程序的类名
    job.setJarByClass(getClass());

    // 设置数据的输入输出路径
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    // 设置map和reduce方法
    job.setMapperClass(ReduceJoinMapper.class);
    job.setReducerClass(ReduceJoinReducer.class);

    // 设置map方法的输出键值对数据类型
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(ReduceJoinWritable.class);
    // 设置reduce方法的输出键值对数据类型
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(NullWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }
  public static void main(String[] args) throws Exception {
    /* 步骤2：运行作业 */
    int exitCode = ToolRunner.run(new ReduceJoin(), args);
    System.exit(exitCode);
  }
}
```

大表⋈小表

83

- 假如部门表比较小，雇员表非常大

雇员表

Name	EmpId	DeptName
Harry	3415	会计
Sally	2241	销售
George	3401	会计
Harriet	2202	销售
Bart	6077	会计
Elise	9263	销售
Gemma	3870	会计
Tyler	6236	销售
Camden	4527	销售
Sofia	2035	会计

.....

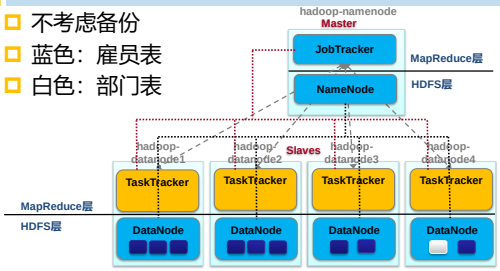
部门表

DeptName	Manager
会计	George
销售	Harriet

数据分布示列

84

- 不考虑备份
- 蓝色：雇员表
- 白色：部门表

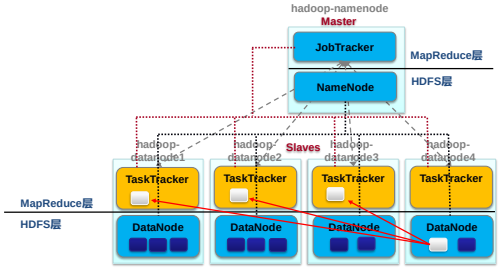


- 若数据本身无序，连接将有大量的数据移动

优化方案

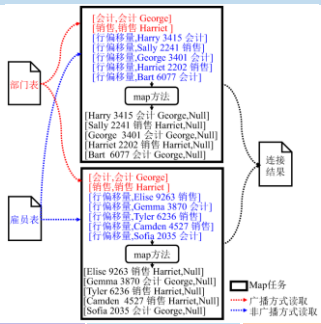
85

- 编程时将“小表”广播出去



优化方案

86



编写map方法

87

```
/* 步骤1：确定输入键值对[K1, V1]的数据类型为[LongWritable, Text]，确定输出键值对[K2, V2]的
   数据类型为[Text, NullWritable] */
public class MapJoinMapper extends Mapper<LongWritable, Text, Text, NullWritable> {
 private Map<String, String> departmentsTable = new HashMap<>();

 @Override
 protected void map(LongWritable key, Text value, Context context)
   throws IOException, InterruptedException {
  /* 步骤2：编写处理逻辑得[K1, V1]转换为[K2, V2]并输出 */
  if (departmentsTable.isEmpty()) {
   URI uri = context.getCacheFiles()[0];
   FileSystem fs = FileSystem.get(uri, new Configuration());
   BufferedReader reader = new BufferedReader(new InputStreamReader(fs.open(new
       Path(uri))));
   String content;
   // 载入部门表
   while ((content = reader.readLine()) != null) {
    // 以制表符为分隔符对部门表元组进行切分，以便获取DeptName属性值
    String[] datas = content.split("\t");
    // 以DeptName属性值为键将元组保存在集合中
    departmentsTable.put(datas[0], datas[1]);
   }
  }
  // 以制表符为分隔符切分雇员表元组
  String[] datas = value.toString().split("\t");
  // 获取DeptName的属性值
  String deptName = datas[2];
  // 进行连接操作并输出
  if (departmentsTable.containsKey(deptName)) {
   context.write(new Text(value.toString() + "\t" + departmentsTable.get(deptName)),
       NullWritable.get());
  }
 }
}
```

读取广播的部门表，并与输入的雇员表执行连接操作

修改主方法

88

```
public class MapJoin extends Configured implements Tool {

 @Override
 public int run(String[] args) throws Exception {

  /* 步骤1：设置作业的信息 */
  Job job = Job.getInstance(getConf(), getClass().getSimpleName());
  // 设置程序的类名
  job.setJarByClass(getClass());

  // 设置数据的输入输出路径
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  // 设置map方法及其输出键值对数据类型
  job.setMapperClass(MapJoinMapper.class);
  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(NullWritable.class);
  // 设置Reduce任务数量为0
  job.setNumReduceTasks(0);

  // 将部门表通过分布式缓存广播出去
  job.addCacheFile(new URI(args[2]));

  return job.waitForCompletion(true) ? 0 : 1;
 }

 public static void main(String[] args) throws Exception {
  /* 步骤3：运行作业 */
  int exitCode = ToolRunner.run(new MapJoin(), args);
  System.exit(exitCode);
 }
}
```

将Reduce任务数设置为0，并广播部门表

大纲

89

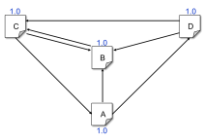
- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 关系表自然连接及其优化
  - 网页链接排名
  - K均值聚类

网页链接排名

90

- 输入：保存在文本文件中，一行为一项网页信息
  - 网页信息：(网页名 网页排名值 (出站链接 出站链接的权重...))
- 输出：网页名及其排名值

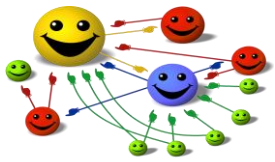
输入	输出
A 1.0 B 1.0 D 1.0 B 1.0 C 1.0 C 1.0 A 1.0 B 1.0 D 1.0 B 1.0 C 1.0	A 0.21436 B 0.36332 C 0.40833 D 0.13027



网页链接排名

91

- 算法思路
  - 许多网页链向该网页，则该网页排名高
  - 有一个高排名值的网页链向该网页，则该网页排名高

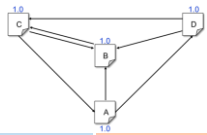


图片来源: en.wikipedia.org/wiki/File:PageRank-hi-res-2.png

网页链接排名

92

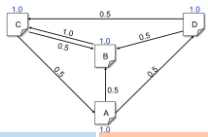
- 算法执行过程
  - 初始时，每个网页的排名值为1



网页链接排名

93

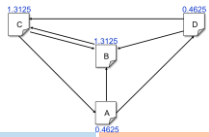
- 算法执行过程
  - 初始时，每个网页的排名值为1
  - 每一步迭代，网页p_i对其链向的网页的排名值贡献$PR(p_j)/L(p_j)$



网页链接排名

94

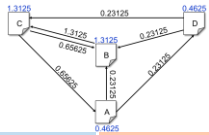
- 算法执行过程
  - 初始时，每个网页的排名值为1
  - 每一步迭代，网页p_i对其链向的网页的排名值贡献$PR(p_j)/L(p_j)$
  - 每个网页p_i累加所有链向p_i的网页$M(p_i)$的贡献值，然后更新排名值为$0.15/N + 0.85 * \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$



网页链接排名

95

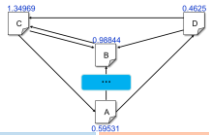
- 算法执行过程
  - 初始时，每个网页的排名值为1
  - 每一步迭代，网页p_i对其链向的网页的排名值贡献$PR(p_j)/L(p_j)$



网页链接排名

96

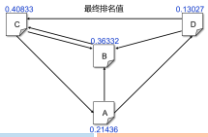
- 算法执行过程
  - 初始时，每个网页的排名值为1
  - 每一步迭代，网页p_i对其链向的网页的排名值贡献$PR(p_j)/L(p_j)$
  - 每个网页p_i累加所有链向p_i的网页$M(p_i)$的贡献值，然后更新排名值为$0.15/N + 0.85 * \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$



网页链接排名

97

- 算法执行过程
  - 初始时，每个网页的排名值为1
  - 每一步迭代，网页p_i对其链向的网页的排名值贡献$PR(p_j)/L(p_j)$
  - 每个网页p_i累加所有链向p_i的网页$M(p_i)$的贡献值，然后更新排名值为$0.15/N + 0.85 * \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$
  - 最终收敛



迭代计算

98

- 迭代计算的特征
  - 由一系列迭代步骤(Step)的循环构成，每个步骤执行的操作完全相同，直到最大迭代次数
  - 一个步骤的输出是下一个步骤的输入
- 迭代计算在MapReduce中的实现
  - 一个步骤的计算过程由一个MapReduce作业来实现，迭代次数决定了MapReduce作业的个数
  - 每一步骤结束时将结果写入HDFS，下一步将该结果再次从HDFS读出

解决方案

99

- 算法执行过程
  - 初始时，每个网页的排名值为1
  - 每一步迭代，网页p_i对其链向的网页的排名值贡献$PR(p_j)/L(p_j)$
  - 每个网页p_i累加所有链向p_i的网页$M(p_i)$的贡献值，然后更新排名值为$0.15/N + 0.85 * \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$
  - 最终收敛

一次迭代一个MapReduce作业

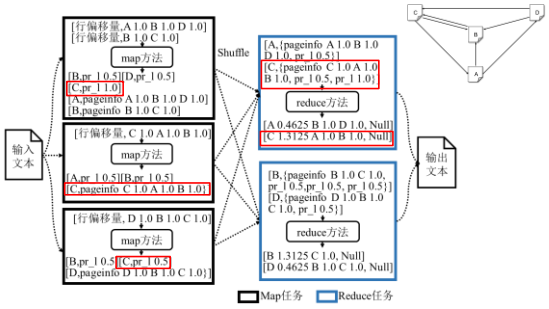
解决方案

100

- Map过程：区分网页信息和贡献值
  - 把每项网页信息转换为键值对[网页名，{PAGE_INFO，网页信息}]
  - 计算每个网页对其出站链接的贡献值，即将每项网页信息转换成键值对[出站链接，{PR_L,贡献值}]
- Reduce过程：
  - 计算每个网页的排名值
  - 如果不是最后一次迭代，则输出更新排名值后的网页信息。否则，输出网页名及其排名值

单次迭代运行过程

101



编写ReducePageRankWritable

102

```
public class ReducePageRankWritable implements Writable {

  // 保存贡献值或网页信息
  private String data;
  // 标识data保存的是贡献值还是网页信息
  private String tag;

  // 用于标识的常量
  public static final String PAGE_INFO = "1";
  public static final String PR_L = "2";

  ......
}
```

与关系表中类似，自定义类型以对不同信息进行标识

编写map方法

103

```
/* 步骤1：确定输入键值对[K1, V1]的数据类型为[LongWritable, Text]，确定输出键值对[K2, V2]的
   数据类型为[Text,ReducePageRankWritable] */
public class PageRankMapper extends Mapper<LongWritable, Text, Text,
    ReducePageRankWritable> {

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    /* 步骤2：编写处理逻辑将[K1, V1]转换为[K2, V2]并输出 */
    // 以空格为分隔符切分
    String[] pageInfo = value.toString().split(" ");
    // 网页的排名值
    double pageRank = Double.parseDouble(pageInfo[1]);
    // 网页的出站链接数
    int outLink = (pageInfo.length - 2) / 2;
    ReducePageRankWritable writable;
    writable = new ReducePageRankWritable();
    // 计算贡献值并保存
    writable.setData(String.valueOf(pageRank / outLink));
    // 设置对应标识
    writable.setTag(ReducePageRankWritable.PR_L);
    // 对于每一个出站链接，输出贡献值
    for (int i = 2; i < pageInfo.length; i += 2) {
      context.write(new Text(pageInfo[i]), writable);
    }
    writable = new ReducePageRankWritable();
    // 保存网页信息并标识
    writable.setData(value.toString());
    writable.setTag(ReducePageRankWritable.PAGE_INFO);
    // 以输入的网页信息的网页名称为键进行输出
    context.write(new Text(pageInfo[0]), writable);
  }
}
```

1. 计算当前网页对出站链接的贡献值，并以出站链接的网页名为键进行输出
2. 以输入的网页信息的网络名称为键输出网页信息

编写reduce方法

104

```
/* 步骤1：确定输入键值对[K2, V2]的数据类型为[Text, ReducePageRankWritable]，确定输出键值
   对[K3, V3]的数据类型为[Text, NullWritable] */
public class PageRankReducer extends Reducer<Text, ReducePageRankWritable, Text,
    NullWritable> {

  // 阻尼系数
  private static final double D = 0.85;

  @Override
  protected void reduce(Text key, Iterable<ReducePageRankWritable> values, Context
      context)
      throws IOException, InterruptedException {
    /* 步骤2：编写处理逻辑将[K2, V2]转换为[K3, V3]并输出 */
    String[] pageInfo = null;
    // 从配置项中读取网页的总数
    int totalPage = context.getConfiguration().getInt(PageRank.TOTAL_PAGE, 0);
    // 从配置项中读取当前迭代次数
    int iteration = context.getConfiguration().getInt(PageRank.ITERATION, 0);
    double num = 0;
    for (ReducePageRankWritable value : values) {
      String tag = value.getTag();
      // 如果是贡献值则进行求和，否则记录除与分隔符切分后保存到pageInfo
      if (tag.equals(ReducePageRankWritable.PR_L)) {
        num += Double.parseDouble(value.getData());
      } else if (tag.equals(ReducePageRankWritable.PAGE_INFO)) {
        pageInfo = value.getData().split(" ");
      }
    }
    // 根据公式计算排名值
    double pageRank = (1 - D) / totalPage + D * num;
    // 更新网页信息中的排名值
    pageInfo[1] = String.valueOf(pageRank);
    // 最后一次迭代输出网页信息排名值，否则只输出网页名称
    StringBuilder result = new StringBuilder();
    if (iteration == (PageRank.MAX_ITERATION - 1)) {
      result.append(pageInfo[0].append(" ").append(String.format("%.8f", pageRank));
    } else {
      for (String data : pageInfo) {
        result.append(data).append(" ");
      }
    }
    context.write(new Text(result.toString()), NullWritable.get());
  }
}
```

从输入值中分离网页信息和贡献值，计算排名值并更新网页信息

编写主方法

105

```
public class PageRank extends Configured implements Tool {
  // 最大的迭代次数
  public static final int MAX_ITERATION = 20;
  // 总的网页记录当前迭代次数
  private static int iteration = 0;
  // 配置项中用于记录网页总数的键
  public static final String TOTAL_PAGE = "1";
  // 配置项中用于记录当前迭代次数的键
  public static final String ITERATION = "2";

  @Override
  public int run(String[] args) throws Exception {
    /* 步骤1：设置作业的信息 */
    int totalPage = Integer.parseInt(args[2]);
    getConf().setInt(PageRank.ITERATION, iteration);
    getConf().setInt(PageRank.TOTAL_PAGE, totalPage);

    Job job = Job.getInstance(getConf(), getClass().getSimpleName());
    // 设置程序的类名
    job.setJarByClass(getClass());

    // 设置数据的输入路径
    if (iteration == 0) {
      FileInputFormat.addInputPath(job, new Path(args[0]));
    } else {
      // 将上一次迭代的输出设置为输入
      FileInputFormat.addInputPath(job, new Path(args[1] + (iteration - 1)));
    }
    // 设置数据的输出路径
    FileOutputFormat.setOutputPath(job, new Path(args[1] + iteration));

    // 设置map方法及其输出键值对的数据类型
    job.setMapperClass(PageRankMapper.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(ReducePageRankWritable.class);

    // 设置reduce方法及其输出键值对的数据类型
    job.setReducerClass(PageRankReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(NullWritable.class);

    return job.waitForCompletion(true) ? 0 : -1;
  }

  public static void main(String[] args) throws Exception {
    /* 步骤1：设计作业 */
    int exitCode = 0;
    while (iteration < MAX_ITERATION) {
      exitCode = ToolRunner.run(new PageRank(), args);
      if (exitCode != 0) {
        break;
      }
      iteration++;
    }
    System.exit(exitCode);
  }
}
```

将每一次迭代的输出设置为下一次迭代的输入，循环提交作业以执行迭代运算

大纲

106

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 关系表自然连接及其优化
  - 网页链接排名
  - K均值聚类

K均值聚类

107

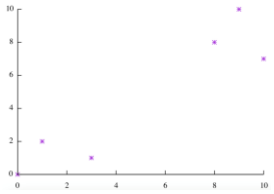
- 输入：两个文本文件，分别保存数据集和聚类中心集
  - 数据集：每行为一个二维数据点及其类别标签
  - 聚类中心集：每行为一个二维数据点
- 输出：数据点及其类别标签

数据集	聚类中心集	聚类结果
0,0 -1 1,2 -1 3,1 -1 8,8 -1 9,10 -1 10,7 -1	1,2 3,1	0,0 1.0 1,2 1.0 10,7 2.0 3,1 1.0 8,8 2.0 9,10 2.0

K均值聚类

108

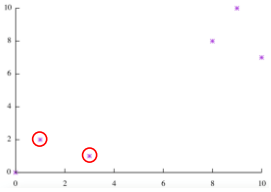
- 算法执行过程
  1. 设定聚类中心数k。例如，k=2



K均值聚类

109

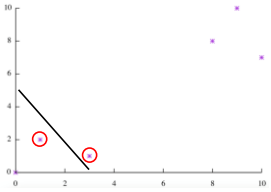
- 算法执行过程
  1. 设定聚类中心数k。例如，k=2
  2. 选取聚类中心



K均值聚类

110

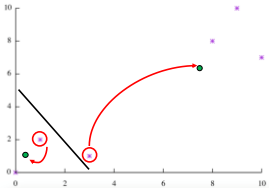
- 算法执行过程
  1. 设定聚类中心数k。例如，k=2
  2. 选取聚类中心
  3. 寻找每个数据点距离最近的聚类中心



K均值聚类

111

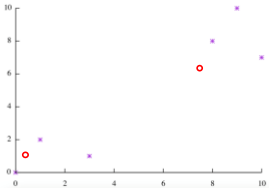
- 算法执行过程
  1. 设定聚类中心数k。例如，k=2
  2. 选取聚类中心
  3. 寻找每个数据点距离最近的聚类中心
  4. 计算同属一个聚类中心的数据点的新的聚类中心



K均值聚类

112

- 算法执行过程
  1. 设定聚类中心数k。例如，k=2
  2. 选取聚类中心
  3. 寻找每个数据点距离最近的聚类中心
  4. 计算同属一个聚类中心的数据点的新的聚类中心
  5. 重复3-4直到满足迭代终止条件



Naïve解决方案

113

- 算法执行过程
  1. 设定聚类中心数k。例如，k=2
  2. 选取聚类中心
  3. 寻找每个数据点距离最近的聚类中心 —一个MapReduce作业
  4. 计算同属一个聚类中心的数据点的新的聚类中心 —一个MapReduce作业
  5. 重复3-4直到满足迭代终止条件 —一次迭代两个MapReduce作业

Naïve解决方案

114

- 第一个MapReduce作业
  - Map过程：
    - 将数据点Hash到不同的Reduce任务，即将数据点转换为键值对[Reduce任务标识, point 数据点]
    - 将聚类中心发送到所有的Reduce任务，即将聚类中心转换为键值对[Reduce任务标识, center 聚类中心]
  - Reduce过程：
    - 分离数据点和聚类中心，计算每个数据点与聚类中心的距离，为数据点添加类别标签
    - 以[类别号, 数据点]形式的键值对输出聚类结果

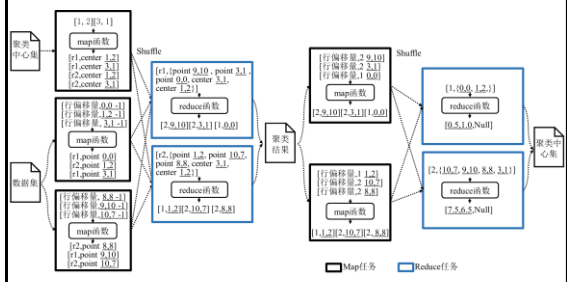
Naïve解决方案

115

- 第二个MapReduce作业
  - Map过程：
    - 读取第一个MapReduce作业产生的聚类结果，并将结果中的数据点转换为[类别号,数据点]的键值对
  - Reduce过程：
    - 计算新的聚类中心，并输出 [新聚类中心，空值]的键值对

Naïve解决方案的单次迭代运行过程

116

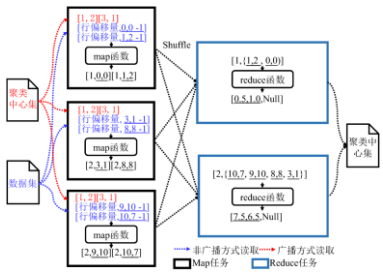


特点：每个数据点都需要与所有聚类中心进行计算

单次迭代运行过程

117

- “广播”聚类中心集



解决方案

118

- 算法执行过程
  1. 设定聚类中心数k。例如，k=2
  2. 选取聚类中心
  3. 寻找每个数据点距离最近的聚类中心 — 在Map阶段实现，其中采用广播方式读取聚类中心
  4. 计算同属一个聚类中心的数据点的新的聚类中心 — 在Reduce阶段实现
  5. 重复3-4直到满足迭代终止条件 — 一次迭代一个MapReduce作业

解决方案

119

- Map过程：
  - 广播方式读取聚类中心集
  - 计算每个数据点与各聚类中心之间的距离，为数据点添加类别标签，即将数据点转换为 [类别号, 数据点]的键值对
- Reduce过程：
  - 为属于同一聚类中心的数据点计算新的聚类中心，并输出[新聚类中心，空值]的键值对

编写map方法

120

读取广播的聚类中心集，为每个数据点添加类别标签

编写reduce方法

121

```java
/* 步骤1：确定输出键值对[K2, V2]的数据类型为[IntWritable, Text] ，确定输出键值对[K3, V3]的
   数据类型为[Text, NullWritable] */
public class KMeansReducer extends Reducer<Text, Text, Text, NullWritable> {

  @Override
  protected void reduce(IntWritable key, Iterable<Text> values, Context context)
      throws IOException, InterruptedException {
    /* 步骤2：编写处理逻辑将[K2, V2]转换为[K3, V3]并输出 */
    List<List<Double>> points = new ArrayList<>();
    // 解析数据点并保存到集合points
    for (Text text : values) {
      String value = text.toString();
      List<Double> point = new ArrayList<>();
      for (String s : value.split(",")) {
        point.add(Double.parseDouble(s));
      }
      points.add(point);
    }
    StringBuilder newCenter = new StringBuilder();
    // 计算每个维度的平均值从而得到新的聚类中心
    for (int i = 0; i < points.get(0).size(); i++) {
      double sum = 0;
      // 计算第i个维度值的和
      for (List<Double> data : points) {
        sum += data.get(i);
      }
      // 计算平均值得到新的聚类中心的第i个维度值并生成需要输出的数据
      newCenter.append(sum / points.size());
      newCenter.append(",");
    }

    context.write(new Text(newCenter.toString()), NullWritable.get());
  }
}
```

从输入值中解析数据点并计算聚类中心

编写主方法

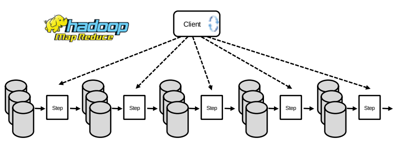
122

广播聚类中心集，并在最后一次迭代时将Reduce任务数设置为0

迭代计算的性能分析

123

- 一个步骤的计算过程由一个MapReduce作业来实现，迭代次数决定了MapReduce作业的个数
- 每一步骤结束时将结果写入HDFS，下一步将该结果再次从HDFS读出



课后阅读

124

- 论文
  - Dean, J., & Ghemawat, S. (2004). MapReduce : Simplified Data Processing on Large Clusters. In OSDI (pp. 137–149).

本章小结

125

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

谢谢！ Q&A