MapReduce去重

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
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.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class Dedup {
    //map将输入中的value复制到输出数据的key上，并直接输出
    public static class Map extends Mapper<Object, Text, Text, Text> {
        private static Text line = new Text();

        //实现map函数
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            line = value;
            context.write(line, new Text(""));
        }
    }

    //reduce将输入中的key复制到输出数据的key上，并直接输出
    public static class Reduce extends Reducer<Text, Text, Text, Text> {
        //实现reduce函数
        public void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {
            context.write(key, new Text(""));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
args).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: Data Deduplication <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "Data Deduplication");
        job.setJarByClass(Dedup.class);
//设置Map、Combine和Reduce处理类
        job.setMapperClass(Map.class);
        job.setCombinerClass(Reduce.class);
        job.setReducerClass(Reduce.class);
//设置输出类型
        job.setOutputKeyClass(Text.class);
```

```
        job.setOutputValueClass(Text.class);
        //设置输入和输出目录
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

MapReduce排序

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
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.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class Sort {
    //map将输入中的value转化成IntWritable类型，作为输出的key
    public static class Map extends Mapper<Object, Text, IntWritable,
IntWritable> {
        private static IntWritable data = new IntWritable();

        //实现map函数
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            String line = value.toString();
            data.set(Integer.parseInt(line));
            context.write(data, new IntWritable(1));
        }
    }

    //reduce将输入中的key复制到输出数据的key上，
//然后根据输入的value-list中元素的个数决定key的输出次数
//用全局linenum来代表key的位次
    public static class Reduce extends
            Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {
        private static IntWritable linenum = new IntWritable(1);

        //实现reduce函数
        public void reduce(IntWritable key, Iterable<IntWritable> values,
Context context)
                throws IOException, InterruptedException {
            for (IntWritable val : values) {
                context.write(linenum, key);
                linenum = new IntWritable(linenum.get() + 1);
            }
        }
```

```
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
args).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: Data Sort <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "Data Sort");
        job.setJarByClass(Sort.class);
//设置Map和Reduce处理类
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
//设置输出类型
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);
//设置输入和输出目录
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

HBase1.1版本Java编程源代码.doc

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;

import java.io.IOException;

public class ExampleForHbase {
    public static Configuration configuration;
    public static Connection connection;
    public static Admin admin;

    //主函数中的语句请逐句执行，只需删除其前的//即可，如：执行insertRow时请将其他语句注释
    public static void main(String[] args) throws IOException {
        //创建一个表，表名为Score，列族为sname,course
        createTable("Score", new String[]{"sname", "course"});

        //在Score表中插入一条数据，其行键为95001,sname为Mary（因为sname列族下没有子列所以
第四个参数为空）
        //等价命令：put 'Score','95001','sname','Mary'
        insertRow("Score", "95001", "sname", "", "Mary");
        //在Score表中插入一条数据，其行键为95001,course:Math为88（course为列族，Math为
course下的子列）
        //等价命令：put 'Score','95001','score:Math','88'
        insertRow("Score", "95001", "course", "Math", "88");
        //在Score表中插入一条数据，其行键为95001,course:English为85（course为列族，
English为course下的子列）
        //等价命令：put 'Score','95001','score:English','85'
        insertRow("Score", "95001", "course", "English", "85");
```

```
        //1、删除Score表中指定列数据，其行键为95001,列族为course，列为Math
        //执行这句代码前请deleteRow方法的定义中，将删除指定列数据的代码取消注释注释，将删除制定列族的代码注释
        //等价命令：delete 'Score','95001','score:Math'
        //deleteRow("Score", "95001", "course", "Math");

        //2、删除Score表中指定列族数据，其行键为95001,列族为course（95001的Math和English的值都会被删除）
        //执行这句代码前请deleteRow方法的定义中，将删除指定列数据的代码注释，将删除制定列族的代码取消注释
        //等价命令：delete 'Score','95001','score'
        //deleteRow("Score", "95001", "course", "");

        //3、删除Score表中指定行数据，其行键为95001
        //执行这句代码前请deleteRow方法的定义中，将删除指定列数据的代码注释，以及将删除制定列族的代码注释
        //等价命令：deleteall 'Score','95001'
        //deleteRow("Score", "95001", "", "");

        //查询Score表中，行键为95001，列族为course，列为Math的值
        //getData("Score", "95001", "course", "Math");
        //查询Score表中，行键为95001，列族为sname的值（因为sname列族下没有子列所以第四个参数为空）
        getData("Score", "95001", "sname", "");

        //删除Score表
        //deleteTable("Score");
    }

    //建立连接
    public static void init() {
        configuration = HBaseConfiguration.create();
        configuration.set("hbase.rootdir", "hdfs://localhost:9000/hbase");
        try {
            connection = ConnectionFactory.createConnection(configuration);
            admin = connection.getAdmin();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    //关闭连接
    public static void close() {
        try {
            if (admin != null) {
                admin.close();
            }
            if (null != connection) {
                connection.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
    /**
     * 建表。HBase的表中会有一个系统默认的属性作为主键，主键无需自行创建，默认为put命令操作中表名后第一个数据，因此此处无需创建id列
     *
     * @param myTableName 表名
     * @param colFamily   列族名
     * @throws IOException
     */
    public static void createTable(String myTableName, String[] colFamily) throws IOException {

        init();
        TableName tableName = TableName.valueOf(myTableName);

        if (admin.tableExists(tableName)) {
            System.out.println("talbe is exists!");
        } else {
            HTableDescriptor hTableDescriptor = new HTableDescriptor(tableName);
            for (String str : colFamily) {
                HColumnDescriptor hColumnDescriptor = new HColumnDescriptor(str);
                hTableDescriptor.addFamily(hColumnDescriptor);
            }
            admin.createTable(hTableDescriptor);
            System.out.println("create table success");
        }
        close();
    }

    /**
     * 删除指定表
     *
     * @param tableName 表名
     * @throws IOException
     */
    public static void deleteTable(String tableName) throws IOException {
        init();
        TableName tn = TableName.valueOf(tableName);
        if (admin.tableExists(tn)) {
            admin.disableTable(tn);
            admin.deleteTable(tn);
        }
        close();
    }

    /**
     * 查看已有表
     *
     * @throws IOException
     */
    public static void listTables() throws IOException {
        init();
        HTableDescriptor hTableDescriptors[] = admin.listTables();
        for (HTableDescriptor hTableDescriptor : hTableDescriptors) {
```

```
            System.out.println(hTableDescriptor.getNameAsString());
        }
        close();
    }

    /**
     * 向某一行的某一列插入数据
     *
     * @param tableName 表名
     * @param rowKey    行键
     * @param colFamily 列族名
     * @param col       列名（如果其列族下没有子列，此参数可为空）
     * @param val       值
     * @throws IOException
     */
    public static void insertRow(String tableName, String rowKey, String
colFamily, String col, String val) throws IOException {
        init();
        Table table = connection.getTable(TableName.valueOf(tableName));
        Put put = new Put(rowKey.getBytes());
        put.addColumn(colFamily.getBytes(), col.getBytes(), val.getBytes());
        table.put(put);
        table.close();
        close();
    }

    /**
     * 删除数据
     *
     * @param tableName 表名
     * @param rowKey    行键
     * @param colFamily 列族名
     * @param col       列名
     * @throws IOException
     */
    public static void deleteRow(String tableName, String rowKey, String
colFamily, String col) throws IOException {
        init();
        Table table = connection.getTable(TableName.valueOf(tableName));
        Delete delete = new Delete(rowKey.getBytes());
        //删除指定列族的所有数据
        //delete.addFamily(colFamily.getBytes());
        //删除指定列的数据
        //delete.addColumn(colFamily.getBytes(), col.getBytes());

        table.delete(delete);
        table.close();
        close();
    }

    /**
     * 根据行键rowkey查找数据
     *
     * @param tableName 表名
     * @param rowKey    行键
```

```
     * @param colFamily 列族名
     * @param col       列名
     * @throws IOException
     */
    public static void getData(String tableName, String rowKey, String
colFamily, String col) throws IOException {
        init();
        Table table = connection.getTable(TableName.valueOf(tableName));
        Get get = new Get(rowKey.getBytes());
        get.addColumn(colFamily.getBytes(), col.getBytes());
        Result result = table.get(get);
        showCell(result);
        table.close();
        close();
    }

    /**
     * 格式化输出
     *
     * @param result
     */
    public static void showCell(Result result) {
        Cell[] cells = result.rawCells();
        for (Cell cell : cells) {
            System.out.println("RowName:" + new String(CellUtil.cloneRow(cell))
+ " ");
            System.out.println("Timetamp:" + cell.getTimestamp() + " ");
            System.out.println("column Family:" + new
String(CellUtil.cloneFamily(cell)) + " ");
            System.out.println("Column Name:" + new
String(CellUtil.cloneQualifier(cell)) + " ");
            System.out.println("value:" + new String(CellUtil.cloneValue(cell))
+ " ");
        }
    }
}
```

注：

1、新建Java Project：JRE选择：use an execution environment JRE

2、导入Jar包：usr/local/hbase/lib下除ruby外的所有Jar包

RDD编程

案例1、求TOP值

任务描述：

orderid,userid,payment,productid

file1.txt

```
1,1768,50,155
2,1218, 600,211
3,2239,788,242
4,3101,28,599
5,4899,290,129
6,3110,54,1201
7,4436,259,877
8,2369,7890,27
```

file2.txt

```
100,4287,226,233
101,6562,489,124
102,1124,33,17
103,3267,159,179
104,4569,57,125
105,1438,37,116
```

求Top N个payment值

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

object TopN {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("TopN").setMaster("local")
    val sc = new SparkContext(conf)
    sc.setLogLevel("ERROR")
    val lines =
sc.textFile("hdfs://localhost:9000/user/hadoop/spark/mycode/rdd/examples", 2)
    var num = 0;
    val result = lines.filter(line => (line.trim().length > 0) &&
(line.split(",").length == 4))
      .map(_.split(",")(2))
      .map(x => (x.toInt, ""))
      .sortByKey(false)
      .map(x => x._1).take(5)
      .foreach(x => {
        num = num + 1
        println(num + "\t" + x)
      })
  }
}
```

案例2、求最大最小值

任务描述：求出多个文件中数值的最大、最小值

file1.txt

```
129
54
167
324
111
54
26
697
4856
3418
```

file2.txt

```
5
329
14
4567
2186
457
35
267
```

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

object MaxAndMin {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName(
    “MaxAndMin
    “).setMaster("local")
    val sc = new SparkContext(conf)
    sc.setLogLevel("ERROR")
    val lines = sc.textFile("hdfs://localhost:9000/user/hadoop/spark/chapter5",
2)
    val result = lines.filter(_.trim().length > 0).map(line => ("key",
line.trim.toInt)).groupByKey().map(x => {
      var min = Integer.MAX_VALUE
      var max = Integer.MIN_VALUE
      for (num <- x._2) {
        if (num > max) {
          max = num
        }
        if (num < min) {
          min = num
        }
      }
      (max, min)
    }).collect.foreach(x => {
      println("max\t" + x._1)
      println("min\t" + x._2)
    })
  }
}
```

案例3、文件排序

任务描述：
有多个输入文件，每个文件中的每一行内容均为一个整数。要求读取所有文件中的整数，进行排序后，输出到一个新的文件中，输出的内容个数为每行两个整数，第一个整数为第二个整数的排序位次，第二个整数为原待排序的整数

输入文件

file1.txt	file2.txt	file3.txt
33	4	1
37	16	45
12	39	25
40	5	

输出文件

1	1
2	4
3	5
4	12
5	16
6	25
7	33
8	37
9	39
10	40
11	45

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.HashPartitioner

object FileSort {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("FileSort")
    val sc = new SparkContext(conf)
    val dataFile = "file:///usr/local/spark/mycode/rdd/data"
    val lines = sc.textFile(dataFile, 3)
    var index = 0
    val result = lines.filter(_.trim().length > 0).map(n => (n.trim.toInt,
"")).partitionBy(new HashPartitioner(1)).sortByKey().map(t => {
                index += 1
      (index, t._1)
    })
    result.saveAsTextFile("file:///usrl/local/spark/mycode/rdd/examples/result")
  }
}
```

大数据的特点

数据量大、数据类型繁多、处理速度快、价值密度低

舍恩伯格对大数据分析的观点

大数据是指不用随机分析法这样的捷径，而是采用所有数据的方法。大数据中的“大”非绝对意义的大，指全体数据，有时并非真的“大”。

随着数据量的增加，数据错误率也增加，格式也存在不一致。只有5%的数据是结构化且适用传统统计方法，95%的数据是非结构化。只有接受不精确性才能利用这些大量的数据。

大数据的核心：建立在相关关系分析基础上的预测。相关关系是：A与B经常一起发生。只要注意到B发生，就能预测A的发生。

大数据环境下的隐私问题

•大量数据的集中存储增加了其泄露的风险；

•一些敏感数据的所有权和使用权并没有清晰界定。

大数据的两大核心技术

分布式存储、分布式处理（有一说是Hadoop、Spark）

Hadoop运行模式

本地模式、伪分布式集群、完全分布式集群

Hadoop的配置文件放在什么目录

$HADOOP_HOME/share/hadoop

Hadoop的生态系统包括哪些组件

1、HDFS；2、MapReduce；3、YARN；4、Hive；5、Pig；6、HBase；7、HCatalog；8、Avro；9、Thrift；10、Drill；11、Mahout；12、Sqoop；13、Flume；14、Ambari；15、Zookeeper。

HDFS的结构模型

主从（Master/Slave）结构模型

NameNode、DataNode的作用，NameNode包含的核心数据结构

P52

NameNode：在内存中保存着整个文件系统的名称空间和文件数据块的地址映射

DataNode：提供真实文件数据的存储服务

NameNode包含的核心数据结构：EditLog, FsImage

HDFS-site.xml的几个属性的功能

参数	作用
dfs.namenode.name.dir	名称节点本地文件系统中存放元数据文件表(fsimage文件)的目录。这个文件中存储的是HDFS元数据的最近快照。如果该属性值是一个逗号分隔的目录列表，文件会被复制到所有的目录中，用作数据冗余。该属性默认值为:file://${hadoop.tmp.dir}/dfs/name。
dfs.namenode.edits.dir	名称节点本地文件系统中存储元数据事务处理文件(edits文件)的目录。如果该属性值是一个逗号分隔的目录列表，该事务处理文件会被复制到所有的目录中，用作数据冗余。其默认值与dfs.namenode.name.dir相同
dfs.namenode.checkpoint.dir	该属性决定了辅助名称节点中存放临时fsimage文件的目录，该临时fsimage文件用来在名称节点可访问的本地/网络文件系统中进行合并，该文件用来与从名称节点拷贝过来的edits文件合并。如果这是一个以逗号分隔的目录列表，镜像文件会复制到所有目录中，用作数据冗余。该属性的默认值为file://${hadoop.tmp.dir}/dfs/namesecondary
dfs.namenode.checkpoint.edits.dir	该属性决定了辅助名称节点中存放从名称节点拷贝过来的edits文件的目录，该文件用来与已经拷贝到由dfs.namenode.checkpoint.dir属性决定的目录中的fsimage文件合并，该合并在辅助名称节点可访问的本地/网络文件系统中进行。如果这是一个以逗号分隔的目录列表，edits文件会复制到所有目录中，用作数据冗余。该属性的默认值与dfs.namenode.checkpoint.dir相同
dfs.namenode.checkpoint.period	两个检查点之间的间隔秒数。经过该属性配置的时间之后，检查点操作就开始执行，该操作会合并从名称节点拷贝过来的edits文件和fsimage文件
dfs.blocksize	指定新文件的默认数据块大小，单位是字节，其默认值是128MB.需要注意的是数据块的大小不是一个系统全局参数，这个参数可以针对单个文件指定
dfs.replication	默认的数据块备份数量。该参数也可以针对单个文件纪念性指定，如果没有特殊指定，就会以参数值作为文件的备份数量。其默认值为3
dfs.namenode.handler.count	该参数决定了名称节点与数据节点通信的服务器线程数。其默认值为10，但是推荐其值为集群节点数量的10%。最小值为10.如果该值设置过低，会在数据节点的日志中发现很多告警信息，这些告警信息显示了当数据节点与名称节点进行心跳信息通信的时候被拒绝了
dfs.datanode.du.reserved	该参数为每卷磁盘中的保留存储空间(单位为字节)，该存储空间保留供非HDFS使用。其默认值为0，但是建议其值为10GB和整个磁盘空间大小的25%两者之间的较小值
dfs.hosts	该属性指定了指向一个文件的完整路径名，指向的文件包含了一个允许与名称节点通信的主机列表。如果没有设置该属性，集群中的所有节点都允许与名称节点通信

Second NameNode的作用

1. SecondNamenode是对主Namenode的一个补充,对内存的需求和Namenode相同
2. SecondNamenode会周期地进行fsimage文件的合并,防止edits文件过大,导致Namenode启动时间过长,
3. 应该与Namenode部署到不同节点上

HDFS默认的Block Size的大小

64MB

HDFS的读写特征

P65

启动HDFS的命令

/hadoop/sbin/start-dfs.sh

/hadoop/sbin/stop-dfs.sh

Linux命令、HDFS基本的shell 命令

P59

Map Reduce的优势

易于编程,良好的扩展性,高容错性,适合海量数据计算

Map Reduce采用的策略

分而治之

Map Reduce体系结构

主要由四个部分组成，分别是：Client、JobTracker、TaskTracker以及Task。
1、Client：程序通过Client提交到JT端，可以通过Cilent提供的接口查看作业运行状态。
2、JobTracker： 监控资源、调度作业，监控所有的TT和Job的健康，一旦发现失败，就会将任务转移到其他节点。
3、TaskTracker： 想JT汇报资源使用情况和作业运行情况，接受JT的命令并执行。
4、Task： Task 分为Map Task 和Reduce Task 两种，均由TaskTracker 启动。

Map Reduce的工作流程

P133

shuffle的过程

P135

典型的NoSQL数据库包括哪些

Redis、Memcache、MongoDb

HBase属于什么数据库

分布式数据库

HBase的消息通信机制靠什么

Zookeeper

HBase表的索引

1.全局索引；2.覆盖索引；3.本地索引

HBase采用列式存储

P73

Region是什么的基本单位

HBase数据管理的基本单位

-ROOT-表的位置信息保存在哪里

Zookeeper

HBase依靠Map Reduce提供强大的计算能力

HBase中Master服务器及Region服务器的作用

P79

HBase的系统架构

P78

HBase的基本的shell命令

P82

Spark的特点

P193

Spark的生态系统，包含哪些组件

P195、196

Spark运行架构

P197

窄依赖、宽依赖的分类

P205

文件数据读写，可通过sc.textFile读哪些类型的文件

txt文件、json文件、md文件（本地文件、HBase文件）

RDD的概念

P200

RDD的转换操作API

P211、212

RDD的行动操作API

P211、212

Scala语言的控制结构

```
def compare(a:Int,b:Int):Int={

  if(a==b) println("1")

  else if(a<b) println("2")

  else println("3")

}
```