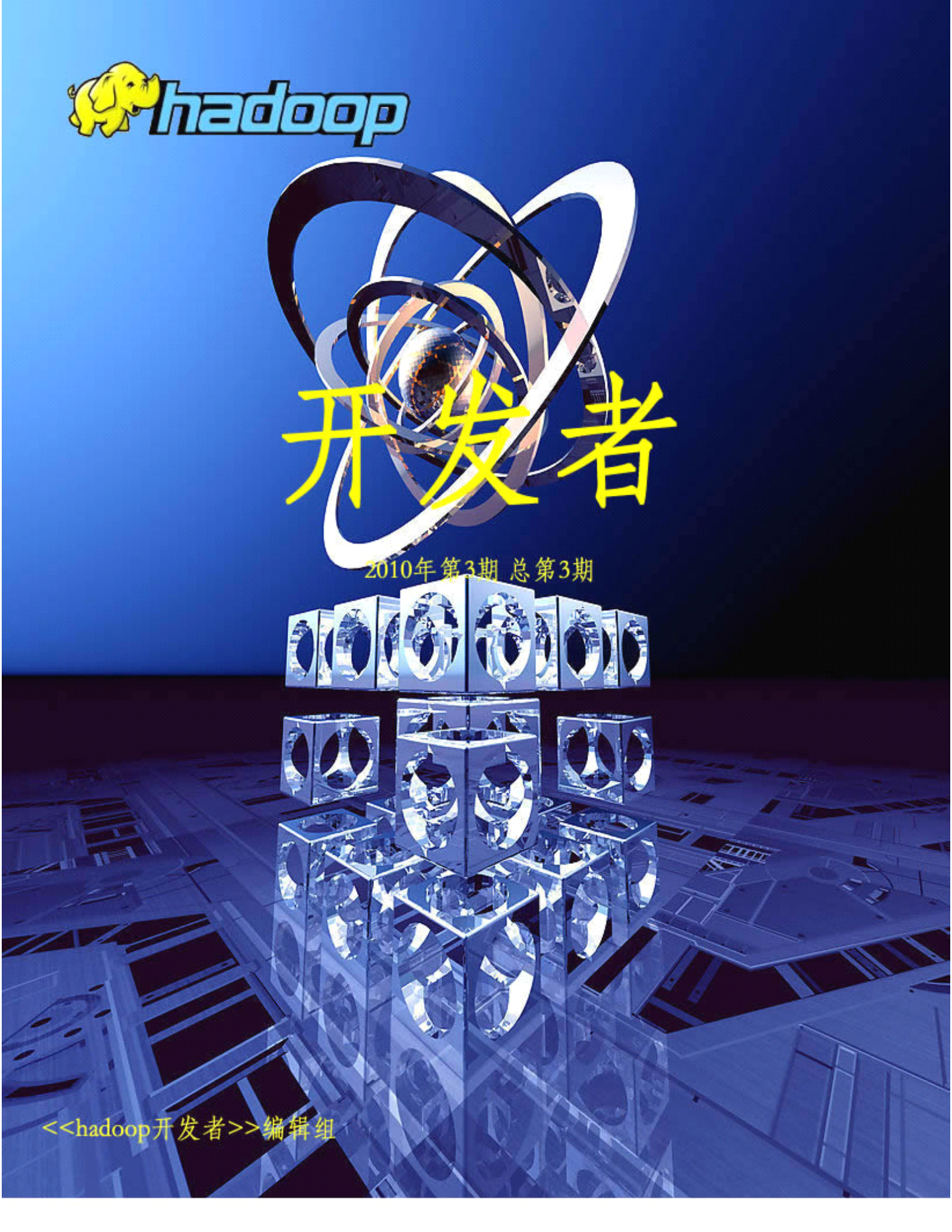
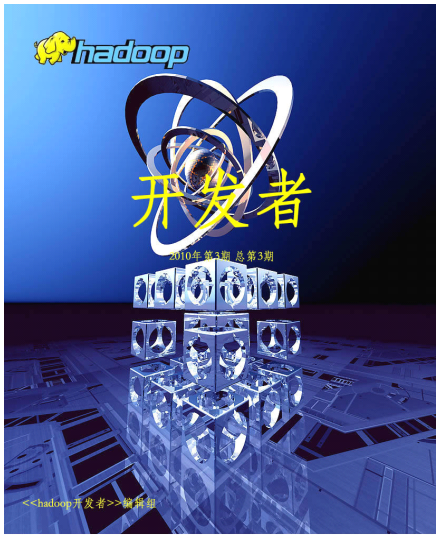

hadoop
开发者
2010年 第3期 总 第3期
<<hadoop开发者>>编辑组



<<Hadoop 开发者>>第三期

2010 年 6 月 20 日发布

欢迎投稿

出品

Hadoop 技术论坛

总编辑

易剑(一剑)

副总编辑

Barry(beyi) 代志远(国宝)

本期执行主编

代志远(国宝)

编辑

皮冰峰(若冰) 易剑(一剑) Barry(beyi) 贺湘辉(小米) Barry(beyi)
代志远(国宝) 柏传杰(飞鸿雪泥) 何忠育(spork) 秘中凯 陈炬

排版/美工/封面设计

代志远(国宝)

网址

http://www.hadoopor.com

投稿信箱

hadoopor@foxmail.com

刊首语

新一期的<<Hadoop 开发者>>与大家见面了。不知不觉 Hadoop 开发者已经与大家携手走过半年多了，感谢各位长久以来多 Hadoop 开发者的支持与鼓励。

Hadoop 是大家目前最热衷的技术话题之一，继 09 年评出国际十大影响力的热门技术榜首的 MapReduce 编程，基于 MapReduce 实现的 Hadoop 开源框架逐渐在被大家所接纳和使用，成为了目前市场中最热门的技术之一。

应运而生的 Hadoop 开发者致力于提供给大家最好的 Hadoop 咨询，让大家更方便的学习 Hadoop 的应用，但是由于人力有限，在每期中所能提供的文章质量难以达到最佳。需要让 Hadoop 开发者办的更好，就需要大家的努力和共同的提高，集思广益，需要 Hadoop 爱好者共同的参与。

希望大家多多投稿，多提意见。

<<Hadoop 开发者>>编辑组

本期执行主编：国宝

2010-6-20

目录

Hadoop 中的数据库访问

作者（飞鸿雪泥） E-mail:jaguar13@yahoo.cn

Hadoop 主要用来对非结构化或半结构化（HBase）数据进行存储和分析，而结构化的数据则一般使用数据库来进行存储和访问。本文的主要内容则是讲述如何将 Hadoop 与现有的数据库结合起来，在 Hadoop 应用程序中访问数据库中的文件。

一. DBInputFormat

DBInputFormat 是 Hadoop 从 0.19.0 开始支持的一种输入格式，包含在包 org.apache.hadoop.mapred.lib.db 中，主要用来与现有的数据库系统进行交互，包括 MySQL、PostgreSQL、Oracle 等几个数据库系统。DBInputFormat 在 Hadoop 应用程序中通过数据库供应商提供的 JDBC 接口来与数据库进行交互，并且可以使用标准的 SQL 来读取数据库中的记录。在使用 DBInputFormat 之前，必须将要使用的 JDBC 驱动拷贝到分布式系统各个节点的$HADOOP_HOME/lib/目录下。

在 DBInputFormat 类中包含以下三个内置类：

1．protected class **DBRecordReader** implements RecordReader<LongWritable, T>：用来从一张数据库表中读取一条条元组记录。

2．public static class **NullDBWritable** implements DBWritable, Writable：主要用来实现 DBWritable 接口。

3．protected static class DBInputSplit implements InputSplit：主要用来描述输入元组集合的范围，包括 start 和 end 两个属性，start 用来表示第一条记录的索引号，end 表示最后一条记录的索引号。

其中 DBWritable 接口与 Writable 接口比较类似，也包含 write 和 readFields 两个函数，只是函数的参数有所不同。DBWritable 中的两个函数分别为：

```
   public void write(PreparedStatement statement) throws
SQLException;
   public void readFields(ResultSet resultSet) throws SQLException;
```

这两个函数分别用来给 java.sql.PreparedStatement 设置参数，以及从 java.sql.ResultSet 中读取一条记录，熟悉 Java JDBC 用法的应该对这两个类的用法比较了解。

二．使用 DBInputFormat 读取数据库表中的记录

上文已经对 DBInputFormat 以及其中的相关内置类作了简单介绍，下面对怎样使用 DBInputFormat 读取数据库记录进行详细的介绍，具体步骤如下：

1．　使用 DBConfiguration.configureDB (JobConf job, String driverClass, String dbUrl, String userName, String passwd)函数配置

JDBC 驱动，数据源，以及数据库访问的用户名和密码。例如 MySQL 数据库的 JDBC 的驱动为“com.mysql.jdbc.Driver”，数据源可以设置为“jdbc:mysql://localhost/mydb”，其中 mydb 可以设置为所需要访问的数据库。

2. 使用 DBInputFormat.setInput(JobConf job, Class<? extends DBWritable> inputClass, String tableName, String conditions, String orderBy, String... fieldNames)函数对要输入的数据进行一些初始化设置，包括输入记录的类名（必须实现了 DBWritable 接口）、数据表名、输入数据满足的条件、输入顺序、输入的属性列。也可以使用重载的函数 setInput(JobConf job, Class<? extends DBWritable> inputClass, String inputQuery, String inputCountQuery)进行初始化，区别在于后者可以直接使用标准 SQL 进行初始化，具体可以参考 Hadoop API 中的讲解。

3. 按照普通 Hadoop 应用程序的格式进行配置，包括 Mapper 类、Reducer 类、输入输出文件格式等，然后调用 JobClient.runJob(conf)。

三．使用示例

假设 MySQL 数据库中有数据库 school，其中的 teacher 数据表定义如下：

```
DROP TABLE IF EXISTS `school`.`teacher`;
CREATE TABLE  `school`.`teacher` (
  `id` int(11) default NULL,
  `name` char(20) default NULL,
  `age` int(11) default NULL,
  `departmentID` int(11) default NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

首先给出实现了 DBWritable 接口的 TeacherRecord 类:

```
public class TeacherRecord implements Writable, DBWritable{
   int id;
   String name;
   int age;
   int departmentID;

   @Override
   public void readFields(DataInput in) throws IOException {
         // TODO Auto-generated method stub
         this.id = in.readInt();
         this.name = Text.readString(in);
         this.age = in.readInt();
         this.departmentID = in.readInt();
   }

   @Override
   public void write(DataOutput out) throws IOException {
         // TODO Auto-generated method stub
         out.writeInt(this.id);
         Text.writeString(out, this.name);
         out.writeInt(this.age);
         out.writeInt(this.departmentID);
   }
```

```
	@Override
	public void readFields(ResultSet result) throws SQLException {
		// TODO Auto-generated method stub
		this.id = result.getInt(1);
		this.name = result.getString(2);
		this.age = result.getInt(3);
		this.departmentID = result.getInt(4);
	}

	@Override
	public void write(PreparedStatement stmt) throws SQLException
{
		// TODO Auto-generated method stub
		stmt.setInt(1, this.id);
		stmt.setString(2, this.name);
		stmt.setInt(3, this.age);
		stmt.setInt(4, this.departmentID);
	}

	@Override
	public String toString() {
		// TODO Auto-generated method stub
		return new String(this.name + " " + this.age + " " +
this.departmentID);
	}
   }
```

利用 DBAccessMapper 读取一条条记录：

```
public class DBAccessMapper extends MapReduceBase implements
Mapper<LongWritable, TeacherRecord, LongWritable, Text> {

  @Override
  public void map(LongWritable key, TeacherRecord value,
              OutputCollector<LongWritable, Text> collector,
Reporter reporter)
              throws IOException {
        // TODO Auto-generated method stub
          collector.collect(new LongWritable(value.id),
           new Text(value.toString()));
  }
}
```

Main 函数如下：

```
public class DBAccess {
  public static void main(String[] args) throws IOException {
        JobConf conf = new JobConf(DBAccess.class);
        conf.setOutputKeyClass(LongWritable.class);
        conf.setOutputValueClass(Text.class);

        conf.setInputFormat(DBInputFormat.class);
        FileOutputFormat.setOutputPath(conf, new
Path("dboutput"));

        DBConfiguration.configureDB(conf,"com.mysql.jdbc.Driver",

"jdbc:mysql://localhost/school","root","123456");

        String [] fields = {"id", "name", "age", "departmentID"};
        DBInputFormat.setInput(conf, TeacherRecord.class,
```

```
"teacher",
                    null, "id", fields);

            conf.setMapperClass(DBAccessMapper.class);
            conf.setReducerClass(IdentityReducer.class);

            JobClient.runJob(conf);
      }
   }
```

该示例从 teacher 表中读取所有记录，并以 TextOutputFormat 的格式输出到 dboutput 目录下，输出格式为<”id”，“name age departmentID”>。

四．使用 DBOutputFormat 向数据库中写记录

DBOutputFormat 将计算结果写回到一个数据库，同样先调用 DBConfiguration.configureDB（）函数进行数据库配置，然后调用函数 DBOutputFormat.setOutput (JobConf job, String tableName, String... fieldNames)进行初始化设置，包括数据库表名和属性列名。同样，在将记录写回数据库之前，要先实现 DBWritable 接口。每个 DBWritable 的实例在传递给 Reducer 中的 OutputCollector 时都将调用其中的 write(PreparedStatement stmt)方法。在 Reduce 过程结束时，PreparedStatement 中的对象将会被转化成 SQL 语句中的 INSERT 语句，从而插入到数据库中。

五．总结

DBInputFormat 和 DBOutputFormat 提供了一个访问数据库的简单接口，虽然接口简单，但应用广泛。例如，可以将现有数据库中的数据转储到 Hadoop 中，由 Hadoop 进行分布式计算，通过 Hadoop 对海量数据进行分析，然后将分析后的结果转储到数据库中。在搜索引擎的实现中，可以通过 Hadoop 将爬行下来的网页进行链接分析，评分计算，建立倒排索引，然后存储到数据库中，通过数据库进行快速搜索。虽然上述的数据库访问接口已经能满足一般的数据转储功能，但是仍然存在一些限制不足，例如并发访问、数据表中的键必须要满足排序要求等，还需 Hadoop 社区的人员进行改进和优化。

MapReduce 中多文件输出的使用

作者（皮冰锋） Email:pi.bingfeng@gmail.com

Mapreduce 的程序设计中，job 定义的 FileOutputFormat 默认只有一个输出，如果是多机实现，该目录下包含多个类似 part-000xx 的文件。因为一个 reducer 对应一个 RecordWriter，而 RecordWriter 负责最后的数据写入，所以 part-000xx 的文件个数与 job 指定的 reducer 个数一致。

那如果要输出为多个文件，我们该如何定义呢？本文总结了 3 种多文件输出的方法。

我们以投票系统作为例子分析，输入为每个人的投票记录(即<people,vote>)，要求将各个投票信息分开存放，即 vote1、vote2、vote3 的信息分开记录。

输入示例：

people1 vote2

people2 vote1

people3 vote2

people4 vote3

people5 vote2

people6 vote1

people7 vote3

people8 vote2

people9vote2

方法一：使用 MultipleOutputFormat

MultipleOutputFormat 只是一个抽象的类，MultipleTextOutputFormat 和 MultpleSequenceFileOutputFormat 是它的两个具体实现。顾名思义，MultipleTextOutputFormat 的多个文件输出都是 TextOutputFormat，即一行一对的文本格式，而 MultpleSequenceFileOutputFormat 的多个文件输出都是 SequenceFile，即二进制文件格式。

具体的实现方法如下：

1. 先要定义一个具体的类扩展 MultipleOutputFormat，在其内部实现中只重载 generateFileNameForKeyValue 方法：

```
    static class VoteNameMultipleTextOutputFormat extends
MultipleTextOutputFormat<Text, Text>{
    protected String generateFileNameForKeyValue (Text vote,
    Text people, String name){
            //原来为 return name, name 为各个不同的 part 分割，如 part-000xx
            //这里修改为 vote/part-000xx
            return new Path(vote.toString(), name).toString();
      }
    }
```

2. 在 Job configure 中设置输出格式为前面自定义的 MultipleOuputFormat：

conf.setOutputFormat(VoteNameMultipleTextOutputFormat.class);

实现成功后，输出文件 output 目录下生成 3 个 vote 目录，分别存放各自投票的人员信息，我这里实现的是单机版，测试结果如下：

```
F:\hadoop相关\output>ls *
vote1:
part-00000

vote2:
part-00000

vote3:
part-00000

F:\hadoop相关\output>more vote1\part-00000
vote1    people2
vote1    people6

F:\hadoop相关\output>more vote2\part-00000
vote2    people1
vote2    people3
vote2    people5
vote2    people8
vote2    people9

F:\hadoop相关\output>more vote3\part-00000
vote3    people4
vote3    people7
```

这里使用的 MultipleOutputFormat 实现的多文件输出，实际上是在 reducer 过程中写入<key, value> pair 的时候，根据 generateFileNameForKeyValue 定义的文件命名规则，将数据写入不同的文件，从而实现多文件输出。

方法二：使用 MultipleOutputs

MultipleOutputs 是在 job 指定的 output 输出的基础上，新增加一些额外的输出，与 MulitpleOutputFormat 相比，它才是真正意义上的多文件输出。

这里分两种情况：

附加单个输出文件：

如果调用 MultipleOutputs 的 addNamedOutput 方法，则是添加一个附加文件，OutputFormat 格式及 key, value 类型都可以自定义；

调用方式为：

在 job 配置阶段添加一个 namedoutput：

```
    MultipleOutputs.addNamedOutput (conf, "vote",
                                TextOutputFormat.class,
NullWritable.class, Text.class);
```

在 reducer 阶段多加一些输出(请自行配置 configure 及 close)：

```
    OutputCollector collector =
     multipleOutputs.getCollector ("vote", reporter);
       while (values.hasNext()){
             Text value = values.next();
             collector.collect(NullWritable.get(), value);
             output.collect(NullWritable.get(), value);
    }
```

附加一批输出文件:

如果调用 MultipleOutputs 的 addMultiNameOutput 方法，则是添加一批附加文件，这些附加文件的 OutputFormat 类型及 key, value 类型必须统一，但可以跟 Job 指定的类型不一致。

附加一批输出的使用方式：

在 job 配置阶段添加一个 addMultiNamedOutput：

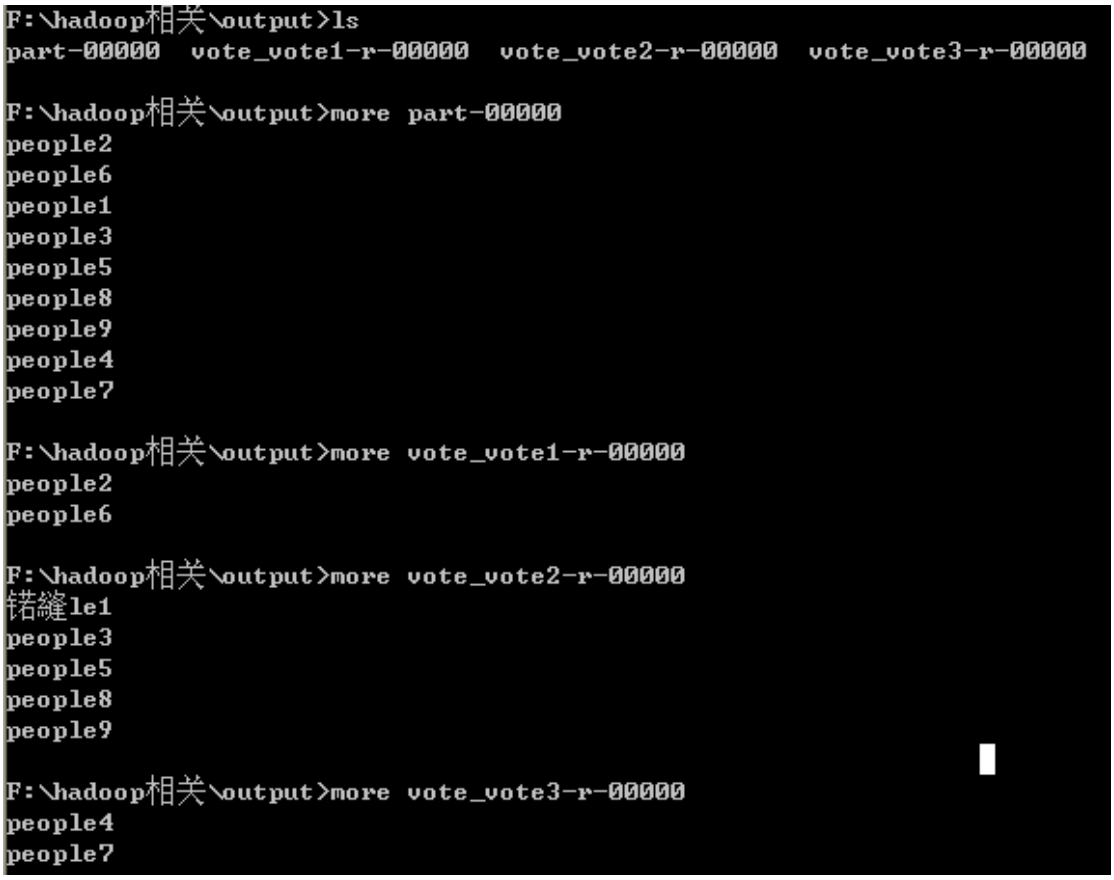
```
    MultipleOutputs.addMultiNamedOutput(conf, "vote",
                                TextOutputFormat.class,
NullWritable.class, Text.class);
```

在 reducer 阶段同样附加一些输出：

//这里我根据不同的 key(即 vote)来分割附加输出，实际可根据需要自行定义

```
    OutputCollector collector =
    multipleOutputs.getCollector("vote", key.toString(), reporter);
      while(values.hasNext()){
                        Text value = values.next();
                        collector.collect(NullWritable.get(), value);
                        output.collect(NullWritable.get(), value);
      }
```

因为本例中需要输出是多个 vote 结果情况，所以采用 addMultiNamedOutput 方法，单机测试结果如下：



如果不需要输出 output，只需要 namedOutput，可以在 Job 定义时设置 OutputFormat 格式为 NullOutputFormat,并去掉 reduce 的 output.collect 方法，只保留 namenode 的 outputcollector 即可。

前面介绍了 MultipleOutputs 在 reducer 中的应用，其实它也可以应用在 mapper 过程中，具体方法与之类似，但是在 mapper 的输出中，只有 output 输出被发送到 reducer 阶段，作为 reducer 的输入，namedOutput 不会参与。

方法三：我把它称为 Writable 方法

这个方法我在《hadoop 开发者第二期》中“Nutch 中 mapreduce 应用的几个特殊点”中提到过，即 Nutch 的 FetcherOutputFormat 展示给我们的方法。

FetcherOutputFormat 的要求是在 reducer 之后分开存储 3 种不同的数据结构：Content, CrawlDatum 即 ParseImpl，但传统的 MapReduce 的 job 输出不能实现这个要求，所以就借用了 NutchWritable 这个统一的对象，以此来囊括前面 3 个对象：

```
public class NutchWritable extends GenericWritableConfigurable {
  private static Class<? extends Writable>[] CLASSES = null;
  static {
    CLASSES = (Class<? extends Writable>[]) new Class[] {
      org.apache.hadoop.io.NullWritable.class,
   ……
      org.apache.nutch.crawl.CrawlDatum.class,
   ……
      org.apache.nutch.parse.ParseImpl.class,
   ……
    };
  }
  public NutchWritable() { }

  public NutchWritable(Writable instance) {
    set(instance);
  }
}
```

在真正的 RecordWriter 写数据的时候，在将上述封装的 NutchWritable 还原成封装之前的 Content, CrawlDatum, ParseImpl 对象，并根据对象的不同写到不同的文件中去，从而实现多文件的写入。

```
    public void write(Text key, NutchWritable value)
            throws IOException {

            Writable w = value.get();

            if (w instanceof CrawlDatum)
              fetchOut.append(key, w);

            else if (w instanceof Content)
              contentOut.append(key, w);

            else if (w instanceof Parse)
              parseOut.write(key, (Parse)w);
    }
```

由于本文前面举的例子中的输入数据结构单一，Writable 方法不太适合，所以就没有实验结果。

最后我们借鉴《Hadoop, the Definite Guide》中的一个表格区分 3 种方法的不同：

	MultipleOutputFormat	MultipleOuputs	Writable
可灵活设置输出文件名称	Yes	No	Yes
输出类型可不一致	No	Yes	Yes
可以用在 mapper 或 reducer 过程中	No	Yes	No
OutputFormat 有各	No,除了	Yes	Yes

种格式	TextOutputFormat 及 SequenceOutputFormat，其他的需要自定义		
每个 record 都有多个输出	No，实际上是对 record 的分割	Yes	No

Zookeeper 使用与分析

国宝

分布式这个概念并不陌生，但真正实战开始还要从 google 说起，很早以前在实验室中分布式被人提出，可是说是计算机内入行较为复杂学习较为困难的技术，并且市场也并不成熟，因此大规模的商业应用一直未成出现，但从 Google 发布了 MapReduce 和 DFS 以及 Bigtable 的论文之后，分布式在计算机界的格局就发生了变化，从架构上实现了分布式的难题，并且成熟的应用在了海量数据存储和计算上，其集群的规模也是当前世界上最为庞大的。以 DFS 为基础的分布式计算框架和 key、value 数据高效的解决运算的瓶颈，而且开发人员不用再写复杂的分布式程序，只要底层框架完备开发人员只要用较少的代码就可以完成分布式程序的开发，这使得开发人员只需要关注业务逻辑的即可。Google 在业界技术上的领军地位，让业界望尘莫及的技术实力，IT 因此也是对 Google 所退出的技术十分推崇。在最近几年中分布式则是成为了海量数据存储以及计算、高并发、高可靠性、高可用性的解决方案。

众所周知通常分布式架构都是中心化的设计，就是一个主控机连接多个处理节点。问题可以从这里考虑，当主控机失效时，整个系统则就无法访问了，所以保证系统的高可用性是非常关键之处，也就是要保证主控机的高可用性。分布式锁就是一个解决该问题的较好方案，多主控机抢一把锁。在这里我们就谈入我们的重点 Zookeeper，

Zookeeper 是什么，chubby 我想大家都不会陌生的，chubby 是实现 Google 的一个分布式锁的实现，运用到了 paxos 算法解决的一个分布式事务管理的系统。Zookeeper 就是雅虎模仿强大的 Google chubby 实现的一套分布式锁管理系统，用于高可靠的的维护元数据，目前该代码开源在 Apache 的 hadoop 项目中，可以下到全部的代码。代码量虽然不多，但实现的结构和投票算法还是蛮复杂的。

下面我们将分以下几个结构来详细分析下 Zookeeper 的实现。

1 应用

1.1 集群模式

集群模式下配置多个 Zookeeper 节点，启动 Zookeeper 集群，Zookeeper 会根据配置投票选举一个节点获得分布式锁，关键配置举例：

```
# The Cluster servers
#server.1=192.168.1.10:2887:3887
#server.2=192.168.1.11:2888:3888
#server.3=192.168.1.12:2889:3889
```

以上配置的几个 Zookeeper 节点会相互投票，直到选举出一个 leader，其他节点则为 follower。

1.2 单机模式

不配置该信息，启动时会自动选取当前节点为 Zookeeper 主控节点。

使用 Zookeeper：

获取 Zookeeper 客户端，zookeeper 提供了同时提供了 c 与 java 的客户端访问接口，但主要框架是由 java 语言实现的，这里就用 java 客户端的使用举例：

调用程序需要指定 zookeeper 的连接地址和端口号，实例化时客户端会自动创建 session 并连接，zookeeper 集群，代码如下

```
ZooKeeper zooKeeper = new ZooKeeper(ip:port, timeout, null);
```

通过以上的方式获取到 zookeeper 的客户端就可以进行 zookeeper 的操作了，示例操作检查节点是否存在，不存在则创建该节点并赋值：

```
Stat stat = zooKeeper.exists(path, false);
byte[] bytes = values;
```

```
    if (stat == null) {
       zooKeeper.create(path,bytes,Ids.OPEN_ACL_UNSAFE,
CreateMode.PERSISTENT);
    } else {
       zooKeeper.setData(path, bytes, -1);
    }
```

删除某个节点:

```
    zooKeeper.delete(path, version);
```

更新某个节点数据:

```
    byte[] bytes = values;
    zooKeeper.setData(path, bytes, -1);
```

2 投票算法

集群模式下有如下配置:

The Cluster servers

#server.1=192.168.1.10:2887:3887

#server.2=192.168.1.11:2888:3888

#server.3=192.168.1.12:2889:3889

每个 zookeeper 节点上都一个唯一的 id，投票算法可以在配置文件中自行指定采用哪种投票算法，选择的优先级最高的是先比较事务序列，向其他节点发送投票时是需要将投票的 id 与事务序列发送出去，先比较事务序列，事务序列最大的首先被选择，如果事务序列相同则考虑 id 中最大的选取出来作为 leader，其他节点被选为 follower，操作数据会以 leader 为主，其他节点将数据从 leader 中 merge 过来。

关键代码片段为：

```
   if ((newZxid > curZxid) || ((newZxid == curZxid) && (newId >
curId)))
      return true;
   else
      return false;
```

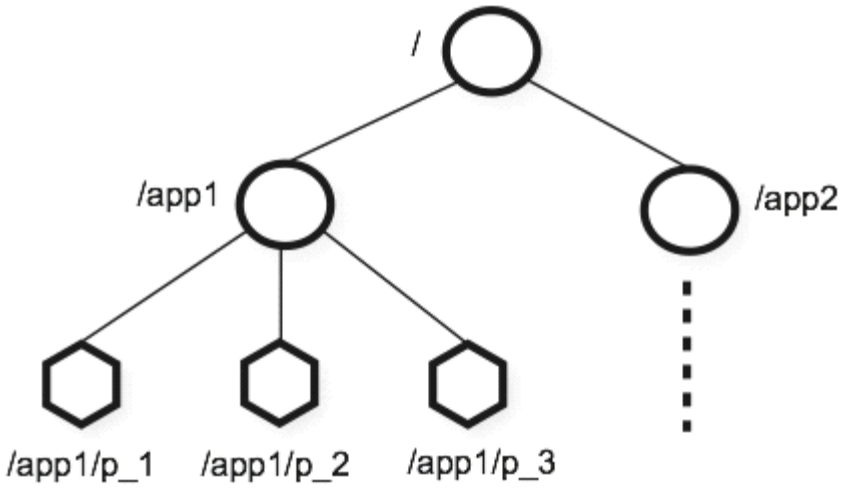
启动时：

>1 每个节点向集群中所有节点包括发送节点自身发送投票，投票推荐自己为主控机

>2 将接受到的票中最大 id 的选出推荐为自己节点的投票本地保存各个节点发来的投票，并计算，将 id 最大的挑出，计算投该 id 的票数，如果超过半数则终止投票选举此 id 为主控，并向其他节点发送通知，如果没有超过半数则向各个节点发送自己推荐的投票并重复第二步操作。

3 元数据

3.1 元数据数据模型



3.2 操作

Zookeeper 中元数据的组织结构是树形的，用户使用可以使用 /a/b/c 的形式来使用元数据，他提供了 create,exsits,delete,getData,setData,getChildren。

create:创建节点

exsits:检查是否有指定名称的节点

delete：删除某个节点

getData：获取某个节点上的值，返回的是字节流

setData：给某个节点上赋值，参数是字节流

getChildren:得到指定节点下的子节点

3.3 元数据的存储

从整体架构上看，zookeeper 存储元数据上就是一个分布式文件系统，但是有不同之处，zookeeper 的每个节点上保存的数据都是整个系统所有数据独立完整的一份，可以说是 follower 节点上的数据都是 leader 节点上的数据备份。

在确定了 leader 与 follower 之后，数据都去操作 leader，并同时将数据备份到 follower 中，这样一来即使 leader 挂掉，可以再次自动投票从 follower 中选出 leader 来，同时数据也是最新的，最大可能的保证了可用性。

4 通信协议

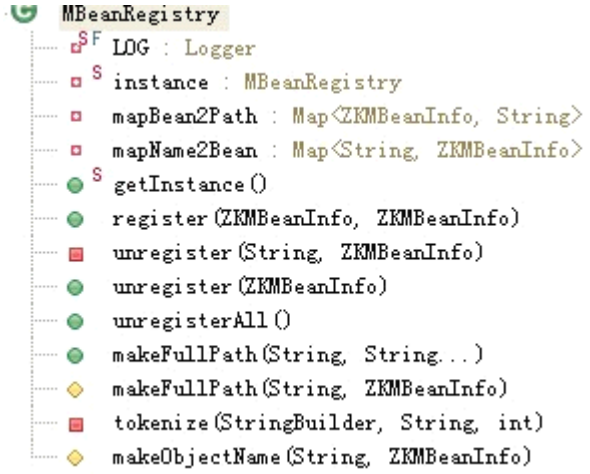
Zookeeper 中投票时需要不同节点之间进行通信投票，而投票不需要确认对方是否一定每个投票都受到，并且为了提高投票的效率节约时间，投票通信时采用的通信方式是 udp，示例如下：

```
    byte responseBytes[] = new byte[48];
    ByteBuffer responseBuffer = ByteBuffer.wrap(responseBytes);
    DatagramPacket responsePacket = new DatagramPacket(responseBytes,
responseBytes.length);
```

```
    mySocket = new DatagramSocket(port);
```

5 zookeeper 节点配置信息维护-JMX

为了方便 zookeeper 节点管理，zookeeper 的节点管理采用了 JMX，示例：



在 MBeanRegistry 注册节点：注册节点实现 ZKMBeanInfo 接口

```
public interface ZKMBeanInfo {
    /**
     * @return a string identifying
     */
    public String getName();
    /**
     * If isHidden returns true, the
     * and thus won't be available f
     * @return true if the MBean is
     */
    public boolean isHidden();
}
```

当需要修改或调用节点相关的信息可以直接冲本地 JMX 中取到。

浅析一种分类数据模型

大卫@火星 zengzhongyi@ustc.mstechclub.cn

引子

互联网数据每天增长速度非常地快，从宏观上看，目前没有哪一种数据模型能够适应这种速度，国内研究互联网数据管理模型的单位并不多，笔者有幸在中科院计算所知识网格组研究互联网资源管理方面的问题，知识网格组于 2002 年左右提出互联网资源空间模型，将分类的观点引入互联网数据管理模型，本文使用到了分类数据空间，但模型与分类数据空间不同。

什么是数据模型

什么是数据？数据是指人看待信息的方式，比如记录,各种序列化的对象等。什么是数据模型？数据模型是指用于组织，管理，操纵数据的存储结构。

分类数据模型

当我们要在一个分类存储系统中构建一个库用于存放相片时，首先我们需要一个模式，这个模式用一棵分类树表示，不鼓励设计出庞大的分类树，如图 1：

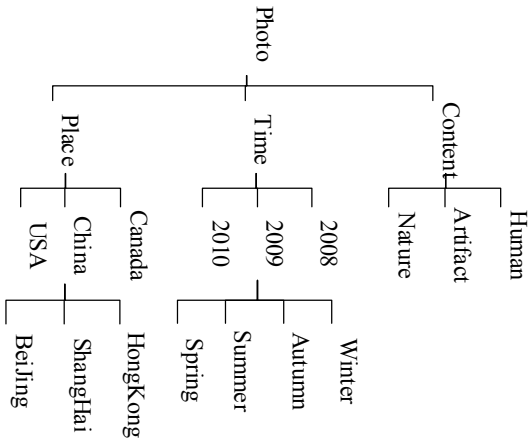


图 1:照片的分类模式树

层次分类模型也是一棵树，如图 3，树中的节点叫数据抽屉，我们都知道，一个抽屉可以存放东西，假定这些东西就是数据，但如果你有非常多的照片，比如 1000 万张，这个时候如果你将它们全部塞进一个抽屉，虽然抽屉能装下这些照片，但是当你想在这个抽屉中准确而快速地找到一张特定的照片却并非易事，然而如果能够在这个抽屉中按照一定的规律对它的空间进行分割，并在抽屉中每一个细小的空间上贴上一个标签，那么当我们在抽屉中寻找某张特定的照片时，我们只需要根据对抽屉进行分割的规律进行寻找，这样似乎很有道理，嗯，那我们就开始构造第一个抽屉，主要任务是如何对这个抽屉进行分割，如图 2，这棵层次树的根节点是一个数据抽屉，该抽屉被分割成 27 个小的空间，P 空间被贴上标签（2009, china, artifact). P 空间是数据抽屉中的一部分空间。根据标签我们知道，P 空间中存储的所有数据都是 2009 年在中国照的人物类的照片。同样的道理，如果把 P 这个小空间也看成是一个数据抽屉的话，且存放在 P 这个抽屉里的照片数目也非常的多，比如达到了 10000 张，这个时候你如果想找到一张是 2009 年春天照的相片，这对于人来说也不是一件非常轻松的事，但如果这部分空间也能按照一定的规律进行分割，是不是会达到好的效果呢？如图 2，数据抽屉 P 依照分类模式树可被分割的成 12 个小空间。于是我们可以想像，随着照片的动态加入与删除，数据抽屉中会嵌套着小的数据抽屉，我们可以把它想像成一棵层次树。

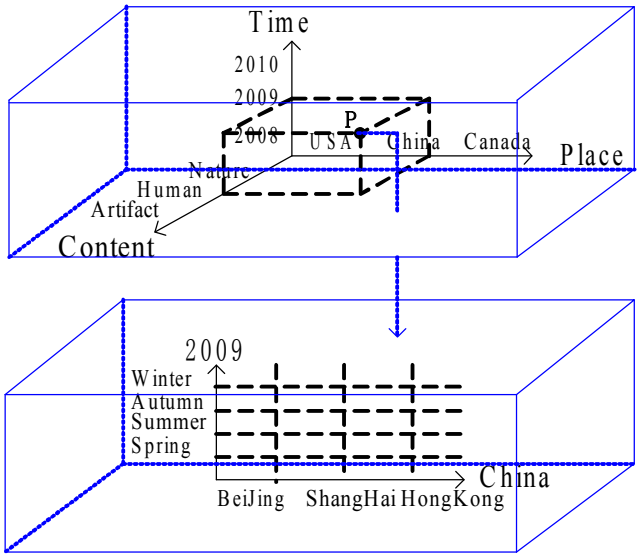


图 2:层次分类数据模型

重要的证明

定理.如果分类模式树的高度为 h,那么分类数据模型的最大高度为 h-2.

证明:(1).当分类模型树的高度为 3 时,分类数据模型至多为 1.(2).假定分类模式树的高度为 h-1 时，分类数据模型至多为 h-3.(3).假定 Ni 是分类模式树中第 h-1 层上的节点，且 Ni 有孩子，根据分类数据模型的构造过程可知，Ni 可能是分类数据模型第 h-2 层上数据抽屉的某一个维。因此其最大高度为 h-2.

这个定理能够说明的道理是分类数据模型是一棵高度十分有限的树。

系统结构

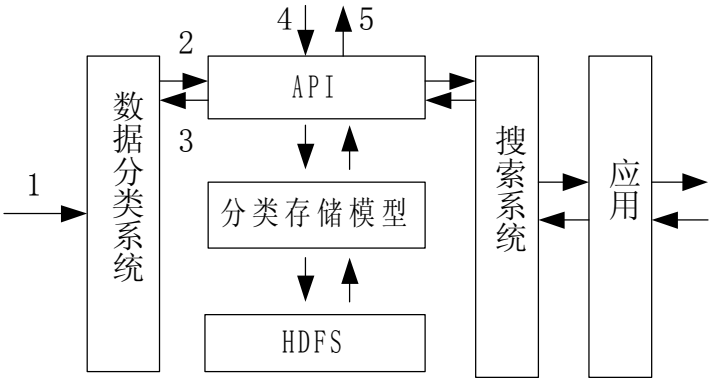


图 3:分类存储系统

数据分类系统模块：该模块有两个输入，一个输出，箭头 1 所代表的数据是待分类数据，在我们的例子中的相片，箭头 3 代表从分类数据模型获取分类模式树，用作数据分类系统的监督者，箭头 2 代表数据经分类系统打过标签后可直接存入分类存储模块。

分类存储模块：该模块实现了分类层次模型，调用 HDFS 提供的文件操纵函数实现与磁盘的交互。并提供一整套的 API 供其它模块调用，箭头 4 是指用户可以将数据通过人工分类的方式存入到分类存储系统，箭头 5 表示用户可直接从分类存储系统取数据。

应用

目前开发了一些原型，但还存在许多问题没有解决，本人编程水平有限，但从某种角度来看这个模型有一定的道理，这个分类层次模型与 trie 树有点像,也是一种以空间换时间的树。写这篇文章，希望能够抛砖引玉，看到真牛人提出数据管理的新方法，新理论。

Sector 框架分析

李海波

1 概述：

云计算被证明在处理大的数据集的时候是非常有效的编程架构。包括谷歌在内的 GFS/MR 和 hadoop，都开始去存储和处理海量数据集，特别是和 WEB 相关的应用。在这片论文李，我们提出一个新的云计算模型，包括云端存储、云端计算。相对于现存的云计算模型，云端存储提供的不仅仅是存储的数据中心，也提供数据的分布式垮广域网传输。另一方面，云端计算提供了一种流处理规则，去支持数据的密集应用。他支持所有的能够用 MR 解决的应用，但是更简单、更直接的去使用，并且 2 倍以 hadoop 的速度运行。我们发布了 ss 的开源版本并进行真实世界的各种使用。

介绍：

提到云，我们马上会联想到这是一个提供按需分配资源的或者服务的互联网上的基础设施，一般都是依托大规模的可靠的数据中心。云端存储提供存储服务（块 I/O，或者文件服务），数据云端提供数据管理服务（基于记录的、基于列的或者基于对象的），云端计算提供计算服务。这些组合起来组成云计算的平台，去开发云级别的应用。

例如谷歌的 GFS，BigTable 和 MR，Amazon 的 s3，simpleDB ，和 EC2，还有 hadoop 系统，HDFS，hadoop 的 MR，和 HBase，一个 BigTable 的实现。

简单的推理：高性能的计算系统中处理器是一个稀缺资源，因此是共享的。当处理器可用，数据迁移到处理器。简单的说，这就是超级云计算的模型。另外一个方法是存储数据并找出数据的共同计算。简单的来说，这就是数据中心模型。

云计算平台（GFS,MR,BigTable，Hadoop）有两个重要的限制：第一个是云计算系统假设云端的所有的节点都在同一节点，要么是同一数据中心，要么是相对小的贷款在地理上分散的包含数据的集群间。

第二个，这些云假设单个的输入或者输出是非常小的，尽管聚集数据管理和处理非常的大。这样可以理解因为大部分云应用都是处理相对小的 web 网页，将 web 网页作为输入的收集和处理，输出包括查询和返回相对小的数据集。尽管有些电子科学应用拥有这些特色，但是其他必摄取相对大的数据集作为返回。

相对的，我们假设，这里有高速的网络（10Gbps 或者更高）连接各种各样的地理上分散的集群，这里的云必须支持摄取和返回交大的数据集。

这篇论文里，我们描述一个云端存储系 sector 和云端计算 sphere。他们的开源地址在：http://sector.sourceforge.net/

Sector 是一个分布式存储系统，能够应用在广域网环境下，并且允许用户以高速度从任何地理上分散的集群间摄取和下载大的数据集。另外，Sector 自动的复制文件有更高的可靠性、方便性和访问吞吐率。Sector 已经被分布式的 Sloan Digital Sky Survey 数据系统所使用。

Sphere 是一个计算服务构建在 sector 之上，并为用户提供简单的编程接口去进行分布式的密集型数据应用。Sphere 支持流操作语义，这通常被应用于 GPU 何多核处理器。流操作规则能够实现在支持 MR 计算的应用系统上。

本文的其余部分将分别描述 sector 和 sphere 在第 2 和第 3 节。第 4 节介绍了一些实验研究。第 5 节介绍有关的工作和第 6 条的摘要和结论。

这是一个扩大版的会议论文相对于[22]文件：1）描述了一个更高版本的 sector，包括安全，b）包括其他信息如 sector 如何工作，包括如何设计安全和调度设计，及 c）描述了新的实验研究。

2 Sector

2.1 概述

Sector 是一个云存储系统。具体的，sector 提供一个高并行性和可靠性的穿越互联网的数据中心提供存储服务，sector 有三个条件：
1）首先，sector 假设他要访问大量的计算机集群（也叫做节点），这些节点可能在数据中心里面，也可能跨数据中心存在。
2）第二，sector 假设这里有一个高速网络连接各个节点。例如，试验数据表明：节点在 1 rack 的网络连接速度是 1Gbps，而 2 racks 的一个数据中心则是 10 Gbps，那么，两个不同的数据中心共同连接 10Gbps。（没太懂，原文如下：For example, in the experimental studies described below, the nodes within a rack are connected by 1 Gbps networks, two racks within a data center are connected by 10 Gbps networks, and two different data centers are connected by 10 Gbps networks.）
3）sector 假设数据集存储在一个或者多个分离的文件中，这叫做 sector 分片。不同的文件组成一个数据集被复制或者分散在多个节点上，但是统一被 sector 管理。例如，一个呗 sector 管理的文件，试验研究将一个 1.3TB 的数据集分片成 64 个文件，每一个大约 20..3GB。

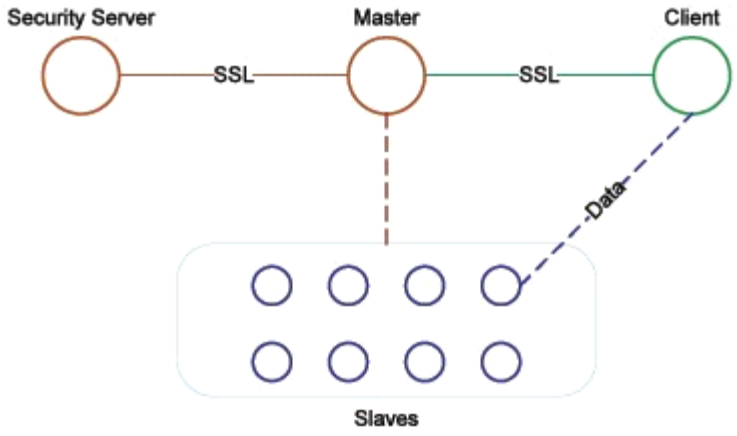


Figure 1. The Sector system architecture.

图 1 sector 系统的架构

图 1 展示了 sector 的总体架构。安全服务器包括用户账户、用户密码和文件访问信息。他还包括一系列奴机 slave 的 IP 地址，所以非法主机不能够假如到系统中来并发送信息来干扰系统。

主机 master 维护文件的元数据信息，控制所有运行的 slave 节点。并且回应客户的请求。Master 同安全服务器交互去认证 slaves、clients 和用户。

Slaves 是存储文件的节点，并且处理数据应答其上的 sector client。

Slaves 通常运行在

一个或者多个数据中心的一组计算机集群上。

2.2 文件系统管理

Sector 并不是本地的文件系统，另外，他依赖于每一个 slave 节点的文帝文件系统去存储分片。设计 sector 的一个关键因素是每一个 sector 分片都存储在一个本地文件系统的单独一个文件中去。这就是说，sector 并不将 sector 分片继续进行分片。这种设计极大的简化了 sector 系统并提供几个优势：首先，这种方法，sector 可以检索所有的元数据，通过简单扫描每一台节点的数据目录。第二个，sector 能够连接到单一的从机节点去下载或者上传文件。相反，如果云存储管理在块级水平，那么，一个用户必须连接存储一个文件的分块的所有从机，hadoop 系统就是这样的一个例子文件管理在块级水平。第三，sector 可以操纵本地文件系统。

这种方法的缺点就是他不允许用户将大的数据集撕裂为多个文件或者用一个 utility 去实现这个。Sector 假设任何用户能够开发足够的代码去应付大数据集并分类为多个文件。

Master 维护元数据索引，这些元数据可能随后被 sector 或者支持的文件查询所访问，例如文件查找和目录服务。

Master 维护所有 slaves 的信息和系统的拓扑，这是为了更好的性能和资源利用。

目前的实现假设 sector 装载一个分层拓扑上，计算节点在多个数据中心的机架内，该拓扑结构手动指定一个主配置文件的服务器。

Master 周期性的检查每个文件的副本。如果数量低于一个门限，则 master 选择一个 slave 创建一个新的文件的副本。新的文件副本位置根据 slaves 的网络拓扑来存放。当一个 client 请求一个文件，master 会选择一个离 client 最近而且不忙的 slave 给 client 使用。

2.2 安全性

Sector 运行一个独立的安全服务器。这种设计允许不同的安全服务提供商进行配置（例如 LDAP 和 Kerberos）。另外，多个 sector masters（为了更高的可靠性和可用性）能够使用安全服务。

Client 登陆 master 服务器是经由 ssl 连接的。用户名、密码送到 master。Master 于是建立 ssl 连接给安全服务器，并去认证 client 的信任度。安全认证服务器检查它的用户数据库并且发送结果反馈给 master，一同的还有一个独一无二的 session ID 和 client 的文件访问权限。除了密码之外，用户的的 client 的 IP 地址也是要经过检查的。SSL 连接也需要凭证和认证。

如果一个 client 需要访问文件，那么 master 会检查哪一个用户有访问那个文件的权限。如果允许，则 master 选择一个 slave 节点为 client 提供服务。Slave 和 client 建立一个独一无二的数据连接，中间是要经过 master 协调的。目前，数据连接并不是加密的，但是我们将来会增加加密功能。

Sector 的 slave 节点仅仅允许 sector 的命令。Sector clients 和其他 slave 节点都不能发送命令直接给 slave。所有的主从节点和从机-从机节点数据传输都必须经过 master 的协调。

最后，安全服务控制是否一个 slave 可以加入到系统。安全服务器支持 IP 列表和 IP 段控制，在这个范围内的计算机才能够加入变成 slave。

2.4 管理和数据传输

Sector 使用 udp 作为信息（message）的传输，而是用 udt 作为数据的传输。Udp 传输速度高于 tcp 因为他不需要简历连接。我们开发一个可靠的信息传输库叫做 GMP（group messaging protocol）用在了 sector 中。对于数据传输，一个 sector slave 会同 client 建立一个 udt 连接，这个 udt 连接使用会和连接模式：rendezvous，并且通过 master 进行协调。Udt 是一个高性能的数据传输协议，在高带宽长距离连接上优于 TCP。

一个单独的 udp 端口使负责传输 message，其他的 udp 端口负责所有的数据连接。一定数量的线程处理 udp 数据包，独立于连接，这使得系统的可扩展性很强。（此处不是很通）。

3 SPHERE

3.1 概述

回忆一下 sphere 是一个计算云覆盖在 sector 的上面。为了介绍 sphere，考虑下面的例子。假设我们有 1 亿 billion 的太空图像，来自 sloan digital sky surver(sdds)，目标是找出褐矮行星。假设平均的图片大小是 1MB 所以总体数据 1TB。SDSS 数据集被存储在 64 个文件中，命名为 SDSS1.DAT，……SDDS64.DAT,每一个包括一个或者多个图片。

为了去随即访问一个数据集中的图片（一共 64 个文件），我们建立一个索引文件为每一个文件。索引文件指示着文件的偏移量：起始点和终止点。索引文件命名规则：数据文件名后面加上 idx，如：SDSS1.DAT.IDX，等。

去使用 sphere，用户写一个函数”findBrownDwarf”去发现褐矮星，从所有图片里面查找。这个函数，输入是一个图片，输出是褐矮星。

findBrownDwarf(input, output);

一个标准的穿行程序可能如下：

```
for each file F in (SDSS slices)
for each image I in F
findBrownDwarf(I, …);
```

使用 sphere 的 client API，相应的伪代码如下：

```
SphereStream sdss;
sdss.init(/*list of SDSS slices*/);
SphereProcess myproc;
Myproc.run(sdss, "findBrownDwarf");
Myproc.read(result);
```

下面的伪代码片段，“sdss”是一个 sector 流数据结构，存储 sector 分片的元数据。应用通过一系列文件名进行流的初始化操作，sphere 从 sector 网络里自动检索元数据，最后三行简单的开始作业

并且等待输出结果。这些不需要用户区定位或者移动数据，也不需要他们去关心什么调度、信息处理和容错等。

3.2 计算模型

如同上面所说的那样，sphere 使用流处理计算模型。流处理模型是最为通用的方式，GPU 和多核处理器都在使用。在 sphere 中，每一个 slave 处理器都被考虑为 GPU 重的 ALU，或者一个多核中的一个 CPU。在流处理模型中，每一个输入数据流中的元素都是被多道计算单元以独立的方式用的同样的处理函数处理，这个模型也叫做 SPMD(单指令流多数据流)。

我们开始阐述 sphere 使用的 key：键值抽象。回忆 sector 数据集里面有无数的物理文件。一个流在 sphere 中式 iyge 抽象的存在并且很可能是一个或者几个数据集。Sphere 使用流作为输入并且使用流作为输出。一个 sphere 流包括多个数据段，这些数据段被 SPE(sphere processing engines)处理，一个 SPE 能够处理段的一个数据记录，一组数据记录，或者整个段。

图 2 说明了一个 sphere 如何处理流中的段。通常 SPE 中有很多数据段，这里会提供一个简单的机制维护加载平衡，较慢的 SPE 会处理少量的段。每一个 SPE 从流中接收一个短作为输入，产生一个段作为流的输出。这些输出段可以作为另外一个 SPE 的处理输入。

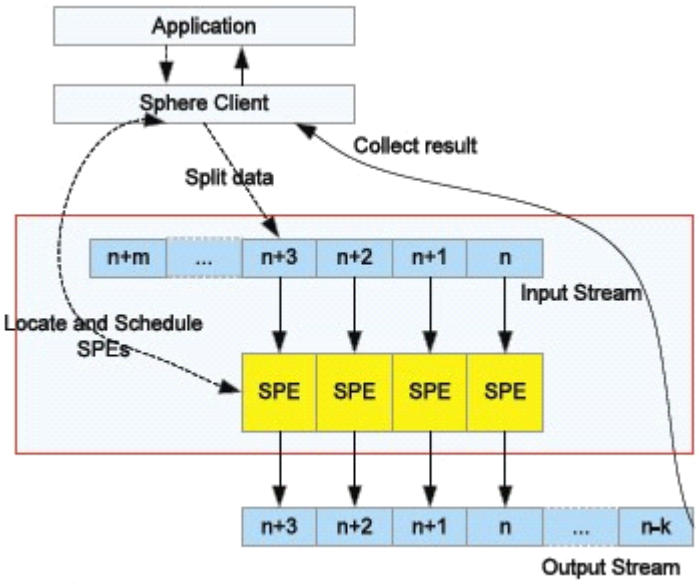


Figure 2. The computing paradigm of Sphere.

图 2 描绘了一个基本的 SPE 模型。Sphere 也支持这个模型的扩展。处理多道输入流：首先，多个输入流能够同时被处理（例如，操作 A[]+B[]）。注意这里并不是简单的扩展，他可以复杂的拆分输入流，并可以将段赋值给 SPE。

经过洗牌的输入流：第二，输出能够被发送到多个位置，而不是写道局部的硬盘。有些时候叫做洗牌。例如，用户定义一个函数被特化为一个 bucket ID，桶 ID（这里说的是目的文件，或者在局部或者在远程节点上）对每一个记录进行输出。Sphere 会发送 SPE’s 中的结果，以他们到达时候的次序将他们写道文件中去。也就是说，sphere 支持 MR 风格的计算。

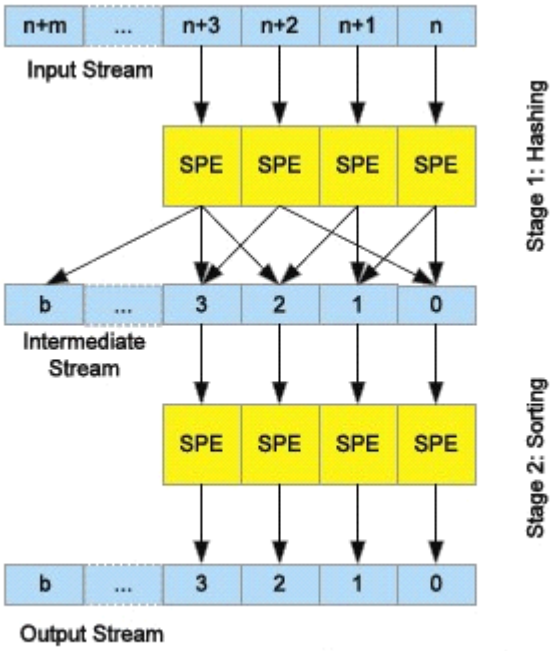


Figure 3: Sorting large distributed datasets with Sphere

图 3 现实了一个使用两个 sphere 处理（每一个处理成为 一个阶段 stage）的例子，第一个阶段将输入数据 hash 操作进入多个桶中，hash 函数扫描整个流并且将每一个元素放到一个合适的桶中。例如：如果被排序的数据是一个整数集合，那么 hash 函数够将所有小于

T0 的数据放入桶 B0 中，数据 T0-T1 之间的放入桶 B1 中，以此类推。

第二阶段，每一个桶（数据段）都被 SPE 所排序，注意经过阶段 2，整个数据集现在是被排序的了。这是因为所有桶中元素都都是有序的，并且桶之间也是有序的。

注意，阶段 2，SPE 处理整个数据段，而不是每一个数据记录。

SPE 可以处理记录或者记录的集合，这个是系统模型的第三个扩展。在 sphere 中，一个 SPE 在一定时间内，能够处理多个记录，一个记录，整个数据段，或者整个文件。

3.3 sphere 处理引擎

一旦 master 接受 client 的数据处理请求，那么，他发送一系列的 slave 节点给 client。Client 选择一些或者所有的 slave，并且请求 SPE 启动这些节点。于是 client 建立与 SPE 端的 udt 连接。流处理函数，以动态链接库的方式发送给各个 SPE 并且存储在 slave 的节点上。SPE 打开动态链接库并获取各种各样的处理函数。于是以下面四个循环开始运行：

1）SPE 从包含文件名的 client 接受新的数据段、偏移和需要处理的行数，还有各种各样的参数。

2）其次，SPE 接受新的数据段（还有相应的索引文件）从一个 slave 的硬盘到另外一个节点中去。

3）其次，S 流处理函数开始处理单一的数据记录，或者一组数据记录，或者整个段，并且将结果写回到合适的目的地。另外，SPE 周期性的发送认证给 client 以实时通告当前处理进展。

当数据段被完全处理，SPE 发送一个认证给 client，去规约被处理的当前数据段。

如果这里没有足够的数据段被处理，那么 client 关闭 SPE 的连接，那么 SPE 被释放。如果 client 被中断的话，那么，SPE 可能超时。

3.4 sphere client

Sphere client 提供一个 API 的集合：开发者可以利用此写分布式应用。开发者能够使用这些 API 初始化输入流，加载处理函数，开启 sphere 处理，并且读处理结果等。

Client 分裂输入流成为多个数据段，所以每一段都可以独立的被 SPE 处理。SPE 能够写本地磁盘，也能够返回处理状态或者返回自己的处理结果。Client 跟踪每一段的状态（例如段是否被处理）并且管理最后结果。

Client 负责管理每一个 sphere 进程的运行，sector/sphere 的一个设计原则就是：将更多的选择机会交给 client 去做，所以 secotr master 能够更为简洁。在 sphere 中，client 负责控制和调度进程的执行。

3.5 scheduler

3.5.1 数据分段和 SPE 初始化

Client 首先从 sector 中定位输入流中的数据文件，如果输入流是先前状态的输出流，那么信息已经在 sector 流结构内不需要继续分段。

总体数据大小和记录都是需要被计算，分片成为段。这是 sector 中基于数据文件的元数据索引。

Client 试图统一的计算输入流给可用的 SPE 单元，每一个 SPE 分配的数据是均衡的。然而，考虑到每一台 SPE 的内存限制，和事物的数据通信代价，sphere 限制数据分段的大小，用 Smin 和 Smax 来表示。（默认值是 8MB-128MB，用户可以自定义）。另外，调度器内存是段倍数的记录不能够被分割。调度器也支持上仅一个文件一个记录的数据。

作为特殊的例子，应用可能需要每一个数据文件以单独的段被处理，例如，现存应用仅仅涉及给一个处理文件。当数据文件没有索引记录的时候调度器也要进行处理。

3.5.2 SPE 调度

如果输入流被分段，那么 client 将每一个段分配给 SPE。下面是其规则：

1）每一个数据段被指派给同一节点的 SPE，假如此节点可用的话。

2）同一个文件的分段同时被处理，除非没有空闲的 SPE。

3）如果讲过 规则 1），2）以后仍然有空闲的 SPE，给他们一部分数据段去处理。

规则 1 试图运行 SPE 在同样的节点，条件是数据驻留，也就是利用数据的局部性。这个减少了网络流量并且获得更好的吞吐量。第二个规则改进了数据的同步访问因为 SPE 能够同时的独立的访问多个文件读取数据。

正如 section 3.3 说的那样，SPE 周期性的提供处理进程的回馈。如果一个 SPE 没有在超时范围内提供回馈，那么，client 将会抛弃这个 SPE。被抛弃的 SPE 的处理的数据段将会移交给其他的 SPE，如果这个 SPE 可用的话，或者已经返回给未赋值的 SPE 池中。这个机制就是 sphere 提供的容错机制。Sphere 并不在 SPE 中使用任何检查点，当处理一个数据段的错误，那么就必须更换另外的 SPE。

当 SPE 写结果给多个目的地的时候，容错模型是非常复杂的（例如使用桶的时候）。每一个 SPE 在讲结果返回给其他节点的桶的时候，倾斜结果给本地磁盘。以这这种方式，假如一个节点坏了，那么，结果必须送给同样的桶的其他节点。每一个桶保持输入结果的记录状态，因此假如一个 SPE 坏了，桶管理器能够继续从另外一个 SPE 接受正确的数据，并处理相同的数据段。

假如处理数据段的时候，输入数据或者用户的 bug 而出现错误，数据段不会被任何 SPE 所处理。错误报告会被返回给 client。因此应用要注意适合的操作。

在大多数情况下，数据段会超过 SPE 的数量，例如，数百台机器可能去处理 TG 的数据。因此，系统自然会均衡加载因为所有的 SPE 都保持繁忙在一段时间内。负载不均衡一般会出现在计算的边缘，这时候会有少量数据被处理，导致一些 SPE 空闲。

不同的 SPE 能够允许不同的时间去处理数据段。这里有几个理由：slaves 节点可能不是专用的，slaves 可能有不同的硬件配置（sector 肯以是同构的）；不同的数据段可能有不同的数据处理时间。计算的末端，空闲 SPE 但是有未完成的数据段的时候，每一个空闲 SPE 会分配一个未完成数据段。也就是说，剩余的数据段仍然

会运行在多个SPE上面，client首先从完成计算的SPE上收集计算结果。这样，sphere避免了SPE之间的进度的不协调。

3.6 与MR模型相对比

Sphere使用的流处理框架还是MR框架都可以是作为简单并行处理的一种方式。应用用户定义的函数（UDF）的方法给进行分段管理比MR的存储分段管理要更为通用（？不太通顺）。Sphere非常容易的具体化一个UDF并随后跟着一个规约UDF。我们下面更为详细的描述之：

一个MR的map处理能够直接被sphere通过输出流到局部存储去进行处理。MR的reduce可以简单的通过hash/桶操作交给sphere。MR模型中，在map阶段，slaves节点之间不可以有数据的交换，每一个reducer在规约阶段从所有的slaves中读数据。而sphere模型中，

第一个阶段会hash（key，value）对给每一个slave节点的桶，第二阶段，所有数据处理处理reduce阶段都在每一台slave本地处理。

我们用如何建立倒排索引的方法来说明MR和sphere的区别。输入是包含 词汇的web网页，而输出是是一系列的（词汇，pages）对。词汇是出现在page之中的，pages代表包含某一词汇的所有网页。

使用sphere建立倒排索引使用两个阶段。在第一个阶段，读入每一个web pages，词汇被检索出来，并且每一个词汇都hash进入一个桶中去。Sphere自动给每一个桶分配一个独立的slave节点进行处理。考虑到这个是hash或者洗牌阶段，那么。更为具体的，所有的以‘a’开始的单词都会被分配给 第0桶，而以b开头的分配给第 1桶，顺次上延。一个更为先进的hash技术是更为防范的分发单词。在第二个阶段，产生一个倒排索引。倒排索引包含多个文件由sector管理。

例如，假设这里有两个web网页：w1.html,w2.html。假设w1包含单词“bee”，“cow”，w2包含单词“bee”，“camel”..在sphere的第一个阶段，桶1会包含（bee，w1）,(bee,w2),桶2会包含

(cow,w1),(camel,w2)。在第二个阶段，每一个桶进行单独处理。桶 1 (bee,(w1,w2)),桶 2 依然没有变化。这种方式，倒排索引和结果已经处理好了，结果存储在多个文件中。

在 hadoop 的 MR 模型中，map 阶段能够产生四个中间文件包括（bee，w1）,(bee,w2), (cow,w1),(camel,w2)。在规约阶段，reducer 合并相同键值并产生三个数据项：(bee,(w1,w2))，(cow,w1),(camel,w2)。

4 试验研究

下面我们给出 2 个 sector/sphere 的应用并讨论他们的性能：

4.1 SDSS 数据分布

Sector 用来分布 SDSS 的数据结果。我们建立多个 sector 服务器在 Teraflow Testbed，在那里我们用来存储 SDSS 数据。我们存储 13TB 的 SDSS 数据版本 5（DR5）。包括 60 个目录文件，64 个目录文件以 EFG 的模式，和 257 个院士图片数据文件。….以下的没用不译。

我们加载 SDSS 文件在几个具体的位置目的是更好的覆盖北美和太平洋和欧洲。我们建立一个 web 站点（sdss。Ncdm。uic.edu）所以用户可以容易的获取 sector client 应用，并且 SDSS 文件能够被下载。每一个文件使用 MD5 校验。这些也在其中用户可以检验文件的完整性。

系统自从 2006 年 5 月就是在线的。过去的两年里，我们大约有 6000 次系统访问大约 250TB 的数据传输给用户终端。大约 80%的用户对目录文件干星期，每一个文件的 size 在 20GB~25GB 之间。

图 4 现实了一次试验中文件下载的平均性能。Clients 的连接数据率依然是 10Gbps。在这个试验中，瓶颈是 disk I/O 的速率。

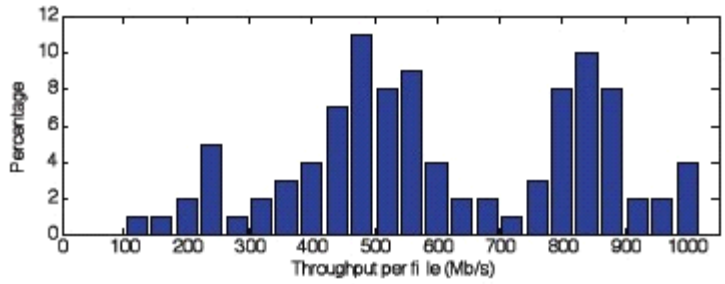


Figure 4. File downloading performance on TFT.

图 5 显示了数据传输的分布在过去十七八个月里实际转移到最终用户的吞吐量。大部分 SDSS 的下载里，瓶颈是网络连接到 Teraflow Testbed 的终端用户和简单的使用多道并行系统的下载。SDSS 下载分布如下：32 来自于美国，37.5 来自于欧洲，18.8 来自于亚洲，等等。用户的传输吞吐率从 8MB/s 到 900Mb/s，所有的都经过公共网络。更多的记录能够在网站上发现。

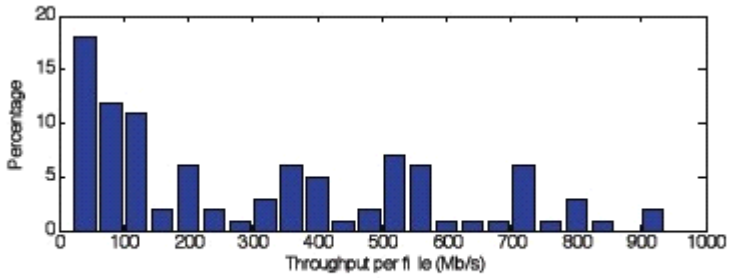


Figure 5. Performance of SDSS distribution to end users.

4.2 Terasort

我们实现一个 Terasort 的基准去评价 sphere 的性能。假设这里有 N 个节点，有 10GB 的文件在每一个节点上需要排序 N*10GB 的数据。每一个记录包含 10 字节键值和 90 字节的值。Sphere 实现的桶排序算法如下图 3。

试验总结见图 1。目前 testbed 包含 4 个机柜。每一有 32 个节点，包括 1 个 NFS 服务器，1 个头结点，和 30 个计算/slave 节点。头结点是 Dell1950……sector 和 hadoop 都配置在 120 个节点的广域网上。Sector 的 master 服务器和命名为 node/job tracker 的 hadoop 配置在一个或者多个首节点上。

图 1 显示了 Terasort 排序的基准性能。注意这里可以看见长时间的处理时间因为总体的数据一致在增加。

	Sector / Sphere	Hadoop 3 replicas	Hadoop 1 replica
UIC (1 location, 30 nodes)	1265	2889	2252
UIC + StarLight (2 locations, 60 nodes)	1361	2896	2617
UIC+StarLight+ CalIT2 (3 locations, 90 nodes)	1430	4341	3069
UIC+StarLight+ CalIT2+JHU (4 locations, 120 nodes)	1526	6675	3702

Table 1. The Terasort benchmark for Sector/Sphere and Hadoop. All time are in seconds.

Secot 和 hadoop 都是安全的复制数据。默认的复制策略都是产生三个副本。他们的复制策略部太相同。Hadoop 复制数据在初始的写阶段，sector 周期性的检查，如果没有足够数量的副本，那么就进行复制操作。见表 1。结果显示 sphere2 倍速度高于 hadoop。如果机柜数量增加，sphere 的优势会更大。

Run on Hadoop

Spork(何忠育)

```
bin/hadoop jar xxx.jar mainclass args
……
```

这样的命令，各位玩 Hadoop 的估计已经调用过 NN 次了，每次写好一个 Project 或对 Project 做修改后，都必须打个 Jar 包，然后再用上面的命令提交到 Hadoop Cluster 上去运行，在开发阶段那是极其繁琐的。程序员是“最懒”的，既然麻烦肯定是要想些法子减少无谓的键盘敲击，顺带延长键盘寿命。比如有的人就写了些 Shell 脚本来自动编译、打包，然后提交到 Hadoop。但还是稍显麻烦，目前比较方便的方法就是用 Hadoop Eclipse Plugin，可以浏览管理 HDFS，自动创建 MR 程序的模板文件，最爽的就是直接 Run on Hadoop 了，但版本有点跟不上 Hadoop 的主版本了，目前的 MR 模板还是 0.19 的，Run on Hadoop 也经常因为跟所用 Eclipse 版本的不匹配导致“哑火”。还有一款叫 Hadoop Studio 的软件，看上去貌似是蛮强大，但是没试过，这里不做评论。那么它们是怎么做到不用上面那个命令来提交作业的呢？是否我们可以自己搞个？不知道？没关系，开源的嘛，不懂得就直接看源码分析，这就是开源软件的最大利处。本想分析 Hadoop Eclipse Plugin 的，但由于 Hadoop Eclipse Plugin 里用到了很多 Eclipse 插件开发的包，我没用过，就懒得一个一个去查了。再说了，我们最好是在理解 Hadoop 作业提交过程的基础上去实现一个 Run on Hadoop，如果可以不借助 Eclipse 插件开发那就最好了，原生态一点，如让 Run as Java Application“变身”成 Run on Hadoop。

我们首先从 bin/hadoop 这个 Shell 脚本开始分析，看这个脚本内部到底做了什么，如何来提交 Hadoop 作业的。

因为是 Java 程序，这个脚本最终都是要调用 Java 来运行的，所以这个脚本最重要的就是添加一些前置参数，如 CLASSPATH 等。所以，我们直接跳到这个脚本的最后一行，看它到底添加了那些参数，然后再逐个分析（本文忽略了脚本中配置环境参数载入、Java 查找、cygwin 处理等的分析，基于 Cloudera Hadoop 0.20.1+152）。

```
# run it
```

```
    exec "$JAVA" $JAVA_HEAP_MAX $HADOOP_OPTS -classpath
"$CLASSPATH" $CLASS "$@"
```

从上面这行命令我们可以看到这个脚本最终添加了如下几个重要参数：JAVA_HEAP_MAX、HADOOP_OPTS、CLASSPATH、CLASS。鉴于篇幅原因，这里只对 CLASSPATH、CLASS 这两个重要参数进行说明。

CLASSPATH 参数是这个脚本的重点之一，主要负责添加配置文件目录和相应依赖库到 Java 的类路径。

```
    # 首先用 Hadoop 的配置文件目录初始化 CLASSPATH
    CLASSPATH="${HADOOP_CONF_DIR}"
    ……
    # 添加 Hadoop 核心 Jar 包和 libs 里的 Jar 包到 CLASSPATH
    for f in $HADOOP_HOME/hadoop-*-core.jar; do
    CLASSPATH=${CLASSPATH}:$f;
    done
    for f in $HADOOP_HOME/lib/*.jar; do
    CLASSPATH=${CLASSPATH}:$f;
    Done
```

CLASS 参数起的作用则是指向真正执行命令的类，它才是提交作业的实体。当你运行 Shell 脚本时，脚本会根据你输入的命令参数来设置 CLASS 和 HADOOP_OPTS，读者如果不知道命令行的某个命令是如何实现的，可以参照看对应的源码。这里我们只关心涉及到作业提交的命令，即“jar”对应的块。

```
    # 确定运行类
    ……
    elif [ "$COMMAND" = "jar" ] ; then
    CLASS=org.apache.hadoop.util.RunJar
    ……
```

嗯，了解了。"$COMMAND" = "jar"时对应的类是 org.apache.hadoop.util.RunJar，这个类等下我们继续分析，它是我们通向最终目标的下一个路口。

脚本在最后还设置了 hadoop.log.dir、hadoop.log.file 等 HADOOP_OPTS。接着，就利用 exec 命令带上刚才的参数调用 org.apache.hadoop.util.RunJar 提交任务了。

通过对上面的分析，我们大概知道了，如果想取代这个脚本，那就必须至少把 Hadoop 配置文件目录和依赖的库给加到类路径中（JAVA_HEAP_MAX 和 HADOOP_OPTS 不是必须的），然后调用 org.apache.hadoop.util.RunJar 类来提交作业到 Hadoop。

到这一步其实我们就可以写一个简单程序添加配置文件目录和依赖的库到类路径或者你直接在 CLASSPATH 环境变量里加入，然后调用 org.apache.hadoop.util.RunJar 类来提交作业。做一个类似 Cloudera Desktop 的 Web 应用，接受用户提交的 Jar，并在 Action 处理中提交到 Hadoop 中去运行，然后把结果返回给用户。鉴于篇幅本文不对这个做详述，详细的可在坛子里找到。

RunJar 是 Hadoop 中的一个工具类，结构很简单，只有两个方法：main 和 unJar。我们从 main 开始一步步分析。

main 首先检查传递参数是否符合要求，然后从第一个传递参数中获取 Jar 包的名字，并试图从 Jar 中包中获取 Manifest 信息，以查找 Mainclass Name。如果查找不到 Mainclass Name，则把传递参数中的第二个设为 Mainclass Name。

接下去，就是在"hadoop.tmp.dir"下创建一个临时文件夹，并挂载上关闭删除线程。这个临时文件夹用来放置解压后的 Jar 包内容。Jar 包的解压工作由 unJar 方法完成，通过 JarEntry 逐个获取 Jar 包内的内容，包括文件夹和文件，然后释放到临时文件夹中。

解压完毕后，开始做类路径的添加，依次把解压临时文件夹、传递进来的 Jar 包、临时文件夹内的 classes 文件夹和 lib 里的所有 Jar 包加入到类路径中。接着以这个类路径为搜索 URL 新建了一个 URLClassLoader（要注意这个类加载器的 parent 包括了刚才 bin/hadoop 脚本提交时给的类路径），并设置为当前线程的上下文类加载器。

最后，利用 Class.forName 方法，以刚才的那个 URLClassLoader 为类加载器，动态生成一个 mainclass 的 Class 对象，并获取它的 main 方法，然后以传递参数中剩下的参数作为调用参数来调用这个 main 方法，里面的代码才是最终的作业提交代码（以 WordCount 示例）。

```
Configuration conf = new Configuration();
……
Job job = new Job(conf, "word count");
job.setJarByClass(WordCountTest.class);
job.setMapperClass(TokenizerMapper.class);
job.setReducerClass(IntSumReducer.class);
……
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Shell 脚本通过反射技术动态执行 main，而我们想要的把 Run as Java Application“变身”成 Run on Hadoop 则是显式的执行包含 Mapper 和

Reducer 的 MR 类的 main 方法。所以我们必须模拟 RunJar 所做的事。主要包含 2 方面：

1、添加配置文件目录及作业依赖库到类路径

2、打包作业程序及相关资源

针对第一个，在 Eclipse 好解决，把需要的库及配置文件目录添加到 Build Path 即可。当然也可利用 Thread.currentThread().setContextClassLoader 来动态设置类路径。对于第二个，这里假设你启用了 Eclipse 的自动编译功能，我们可以在代码的开始阶段加入一段代码用来打包 bin 文件夹里的 class 文件为一个 Jar 包（可以根据自己情况再加入其它资源），然后再执行后面的常规操作。

```
  public static File createTempJar(String root) throws IOException
{
    if (!new File(root).exists()) {
      return null;
    }
    Manifest manifest = new Manifest();
    manifest.getMainAttributes().putValue("Manifest-Version",
"1.0");
    final File jarFile = File.createTempFile("EJob-", ".jar", new
File(System
        .getProperty("java.io.tmpdir")));

    Runtime.getRuntime().addShutdownHook(new Thread() {
      public void run() {
          jarFile.delete();
      }
    });

    JarOutputStream out = new JarOutputStream(new
FileOutputStream(jarFile),
        manifest);
    createTempJarInner(out, new File(root), "");
    out.flush();
    out.close();
    return jarFile;
  }

  private static void createTempJarInner(JarOutputStream out, File
f,
      String base) throws IOException {
    if (f.isDirectory()) {
      File[] fl = f.listFiles();
      if (base.length() > 0) {
          base = base + "/";
      }
      for (int i = 0; i < fl.length; i++) {
          createTempJarInner(out, fl[i], base + fl[i].getName());
```

```
      }
    } else {
      out.putNextEntry(new JarEntry(base));
      FileInputStream in = new FileInputStream(f);
      byte[] buffer = new byte[1024];
      int n = in.read(buffer);
      while (n != -1) {
          out.write(buffer, 0, n);
          n = in.read(buffer);
      }
      in.close();
    }
  }
```

这里的对外接口是 createTempJar，接收参数为需要打包的文件夹根路径，支持子文件夹打包。使用递归处理法，依次把文件夹里的结构和文件打包到 Jar 里。很简单，就是基本的文件流操作，陌生一点的就是 Manifest 和 JarOutputStream，查查 API 就明了。

还需要注意的一点是，在 0.20 接口以后，已使用 job.setJarByClass(ClassName.class)来设置作业 Jar 包。这个方法使用了 ClassName.class 的类加载器来寻找包含该类的 Jar 包，然后设置该 Jar 包为作业所用的 Jar 包。但是通过上述方法，我们的作业 Jar 包是在程序运行时才打包的，而 ClassName.class 的类加载器是 AppClassLoader，运行后我们无法改变它的搜索路径，所以使用 setJarByClass 是无法设置作业 Jar 包的。我们必须使用 JobConf 里的 setJar 来直接设置作业 Jar 包，像下面一样：

```
    ((JobConf)job.getConfiguration()).setJar(jarFile);
```

但是否可以预先设定一个目录，并把它预先加入 Project 的 Build Path 中，然后打包 Jar 时选定此目录作为目标目录，那是否就可以省去上面这句代码呢？毕竟 JobConf 是被 Deprecated 的了。这个笔者没有去试，就留给各位读者了。

下面举个实例来说明该方法的使用，还是以 WordCount 例。首先第一步要做的就是把配置文件目录及作业依赖库添加到 Project 的 Build Path，配置文件要和集群的一致。这里为了简便，未使用动态添加类路径的方法，想了解的可以在坛子里找，附带源码包里有使用动态类路径添加法。

```
    // Add this statements. XXX
    File jarFile = EJob.createTempJar("bin");
    Configuration conf = new Configuration();
    ……
    Job job = new Job(conf, "word count");
```

```
    // And add this statement. XXX
    ((JobConf)job.getConfiguration()).setJar(jarFile.toString());
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    ……
    System.exit(job.waitForCompletion(true) ? 0 : 1);
```

提交作业时，只用像平时运行一般 Java 程序一样 Run as Java Application 即可。这个“绿色”的 Run on Hadoop 方法使用简单，兼容性好，推荐一试。