Google云计算原理

第3章

武汉大学 计算机学院 软件工程系

Google搜索引擎

- 全球最大搜索引擎、Google Maps、Google Earth、Gmail、YouTube等。这些应用的共性在于数据量巨大，且要面向全球用户提供实时服务。
- 关键字搜索
- 语义搜索：知识图谱可以用于诸如问答、语义搜索和自然语言处理等应用程序。知识图谱是由人工智能和自然语言处理技术提供支持的，并且它们通常是基于开放标准构建的，例如RDF和OWL。
- ChatGPT：人工智能技术驱动的自然语言处理工具。

一般架构和算法

Google云计算的三大核心技术：

- GFS
- MapReduce
- BigTable

互联网

用户

网络爬虫

网页去重

反作弊

倒排索引

链接关系

云存储与计算平台

内容相关性

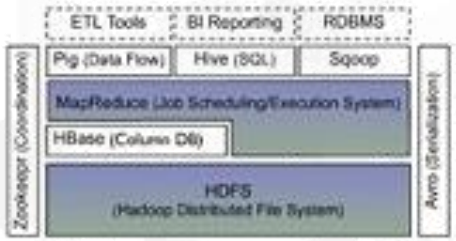
链接分析

查询分析

Cache系统

网页排序

The Hadoop Ecosystem

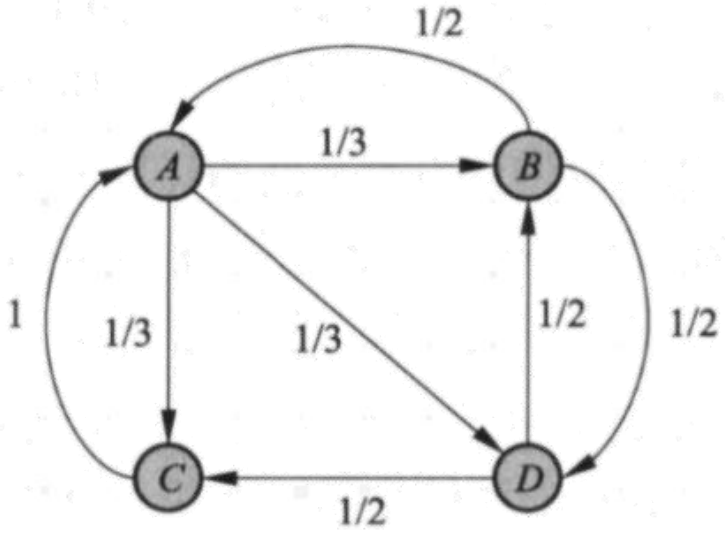


一般架构和算法

- **Bloom Filter**：是一种空间效率高、常用于判定元素是否存在于一个集合中的数据结构。
- **PageRank**算法：由Google公司的创始人之一发明，用于评估网页的重要性和排序，是现代搜索引擎排名算法的基础。

  TF-IDF算法：TF代表词频（Term Frequency），IDF代表逆文档频率，通过计算单词在文档中出现的频率和单词在整个文档集合中出现的频率，评估文档与查询的相关性。

  BM25算法：考虑了词项的权重、文档长度、查询长度等因素，以更准确地计算文档与查询的匹配程度。
- LSI（Latent Semantic Indexing）算法：基于语义相似性来计算文档之间的相关性，通过对文档进行降维和聚类，提高搜索结果的相关性和多样性。



倒排索引（Invert Index）

ID	Col_0	Col_1	Col_2
0	0	a	b
1	1	b	a
2	1	b	b

Col_0	ID
0	0
1	1, 2

Col_1	ID
a	0
b	1, 2

Bloom Filter

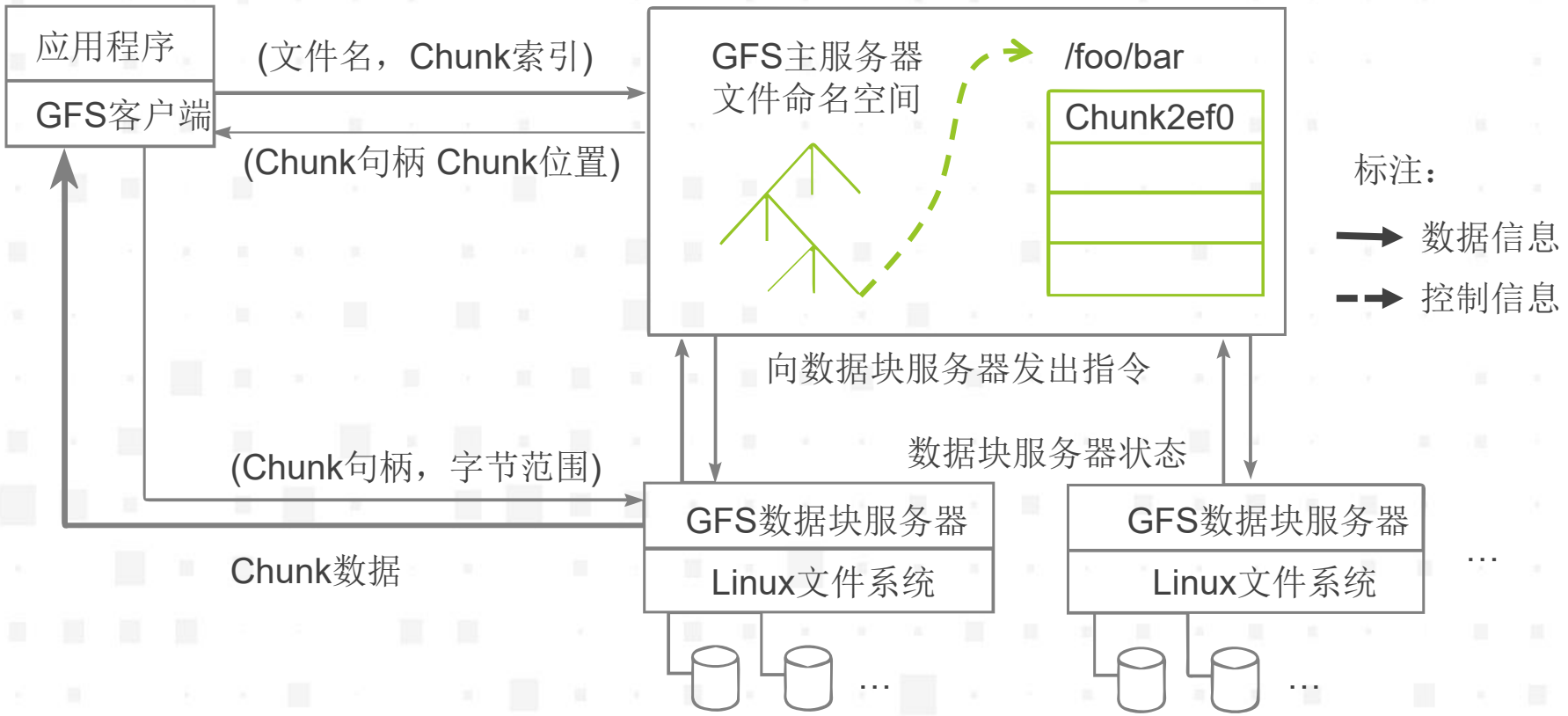
目 录

3.1 Google文件系统GFS

- 2.1.1 系统架构
- 2.1.2 容错机制
- 2.1.3 系统管理技术

3.1 Google文件系统

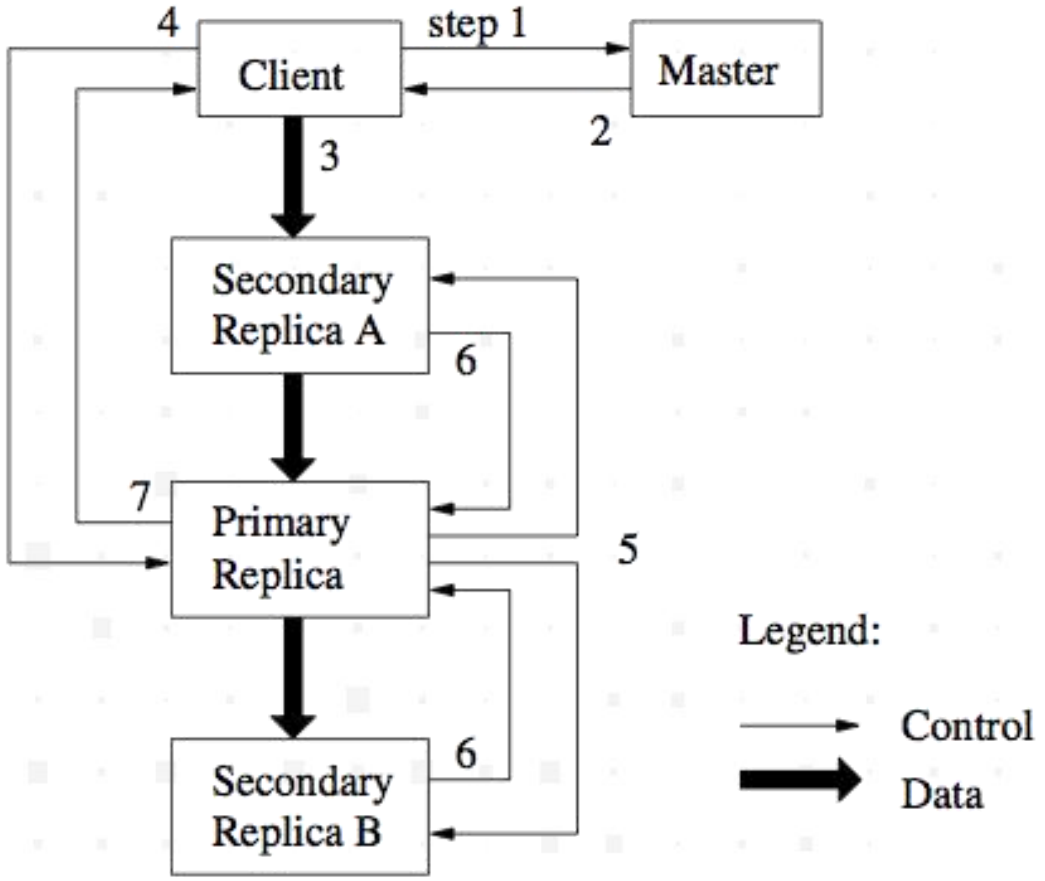
GFS的系统架构



3.1 Google文件系统

GFS的系统架构

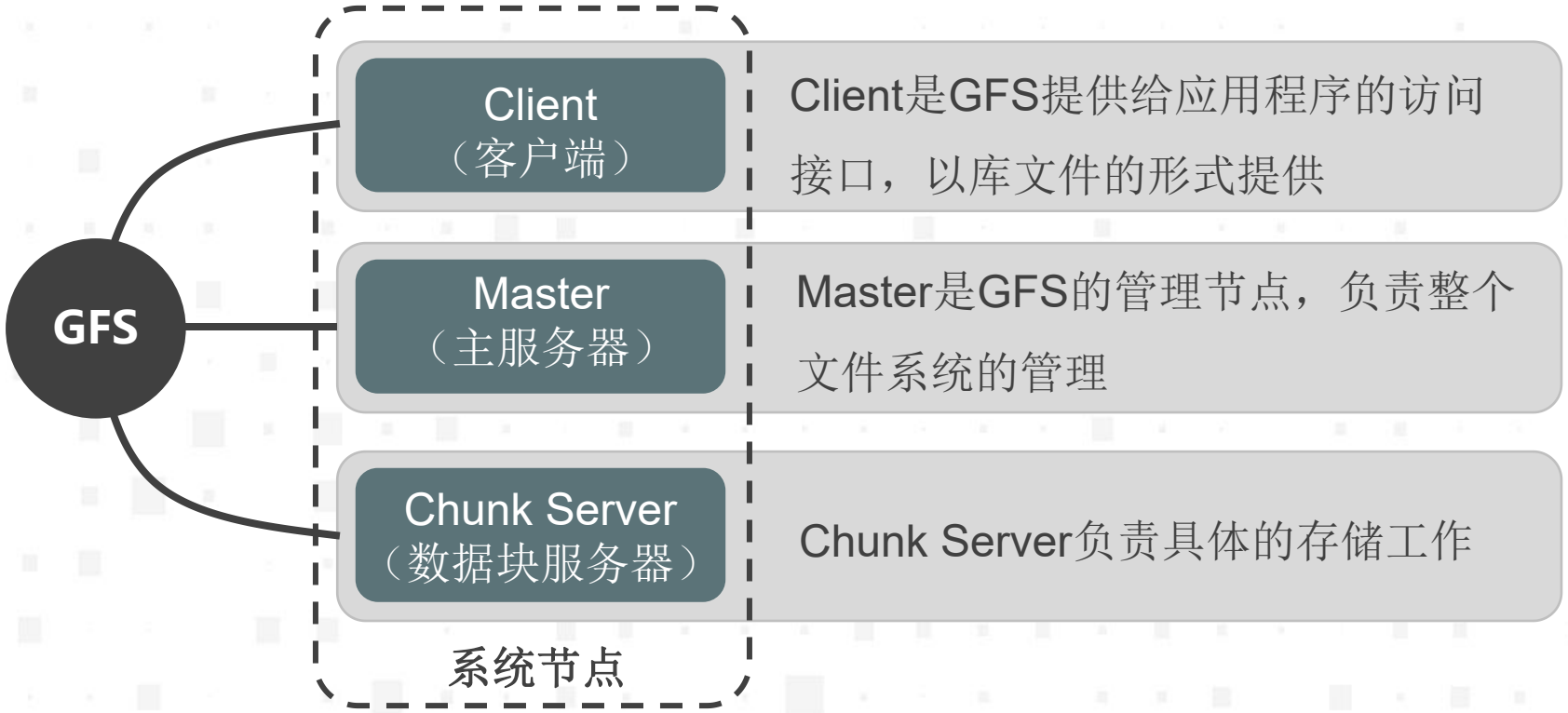
- GFS 使用租约机制来保障在跨多个副本的数据写入中保持顺序一致性。
- GFS Master 将 chunk 租约发放给其中一个副本，这个副本我们就称为主副本，其他副本称为次副本。
- 由主副本来确定一个针对该 chunk 的写入顺序，次副本则遵守这个顺序，这样就保障了全局顺序一致性。
- chunk 租约机制的设计主要是为了减轻 Master 的负担，由主副本所在的 chunkserver 来承担流水线顺序的安排。



分布式事务

3.1 Google文件系统

- GFS将整个系统节点分为三类角色



3.1 Google文件系统

- **GFS的实现机制**

  - 客户端首先访问Master节点，获取交互的Chunk Server信息，然后访问这些Chunk Server，完成数据存取工作。这种设计方法实现了控制流和数据流的分离。
  - Client与Master之间只有控制流，而无数据流，极大地降低了Master的负
  - Client与Chunk Server之间直接传输数据流，同时由于文件被分成多个Chunk进行分布式存储，Client可以同时访问多个Chunk Server，从而使得整个系统的I/O高度并行，系统整体性能得到提高。

3.1 Google文件系统

- GFS的特点

1 采用中心服务器模式

- 可以方便地增加Chunk Server
- Master掌握系统内所有Chunk Server的情况，方便进行负载均衡
- 不存在元数据的一致性问题

3.1 Google文件系统

- **GFS的特点**

2

不缓存数据

- 文件操作大部分是流式读写，不存在大量重复读写，使用Cache对性能提高不大
- Chunk Server上数据存取使用本地文件系统从可行性看，Cache与实际数据的一致性维护也极其复杂

3.1 Google文件系统

- GFS的特点

3

在用户态下实现

- 利用POSIX编程接口存取数据降低了实现难度，提高通用性
- POSIX接口提供功能更丰富
- 用户态下有多种调试工具
- Master和Chunk Server都以进程方式运行，单个进程不影响整个操作系统
- GFS和操作系统运行在不同的空间，两者耦合性降低

3.1 Google文件系统GFS

3.1.1 系统架构

3.1.2 容错机制

3.1.3 系统管理技术

3.1 Google文件系统

- **Master容错**

当Master发生故障时，在磁盘数据保存完好的情况下，可以迅速恢复以上元数据

为了防止Master彻底死机的情况，GFS还提供了Master远程的实时备份

3.1 Google文件系统

- **Chunk Server容错**

GFS采用副本的方式实现Chunk Server的容错

每一个Chunk有多个存储副本（默认为三个）

对于每一个Chunk，必须将所有的副本全部写入成功，才视为成功写入

相关的副本出现丢失或不可恢复等情况，Master自动将该副本复制到其他Chunk Server

GFS中的每一个文件被划分成多个Chunk，Chunk的默认大小是**64MB**

每一个Chunk以Block为单位进行划分，大小为64KB，每一个Block对应一个32bit的校验和

3.1 Google文件系统GFS

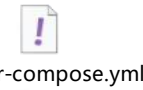
3.1.1 系统架构

3.1.2 容错机制

▶ 3.1.3 系统管理技术

3.1 Google文件系统

- 系统管理技术



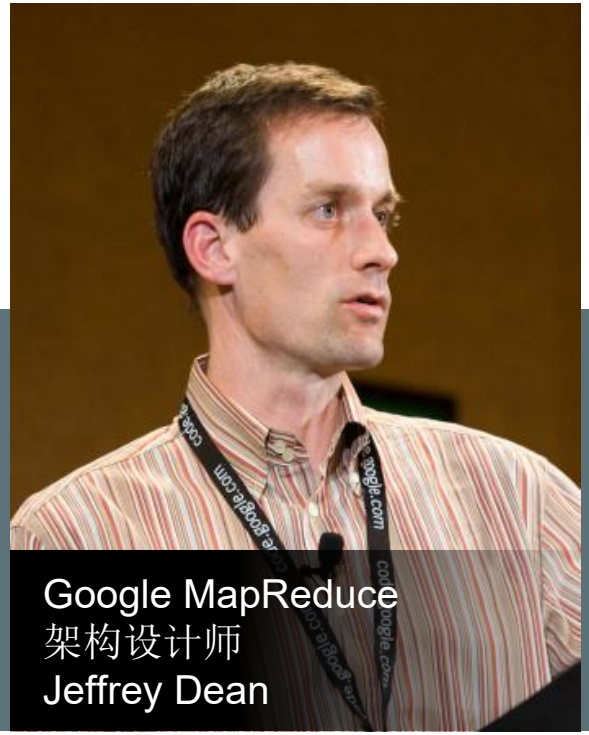
目 录

3.2 分布式数据处理MapReduce

- 3.2.1 产生背景
- 3.2.2 编程模型
- 3.2.3 实现机制
- 3.2.4 案例分析

3.2 分布式数据处理MapReduce

- 产生背景



Google MapReduce
架构设计师
Jeffrey Dean

Jeffery Dean设计一个新的抽象模型， 封装并行处理、容错处理、本地化计算、负载均衡的细节，还提供了一个简单而强大的接口。

这就是MapReduce

3.2 分布式数据处理MapReduce

- 产生背景

MapReduce这种**并行编程模式**思想最早是在**1995年**提出的。

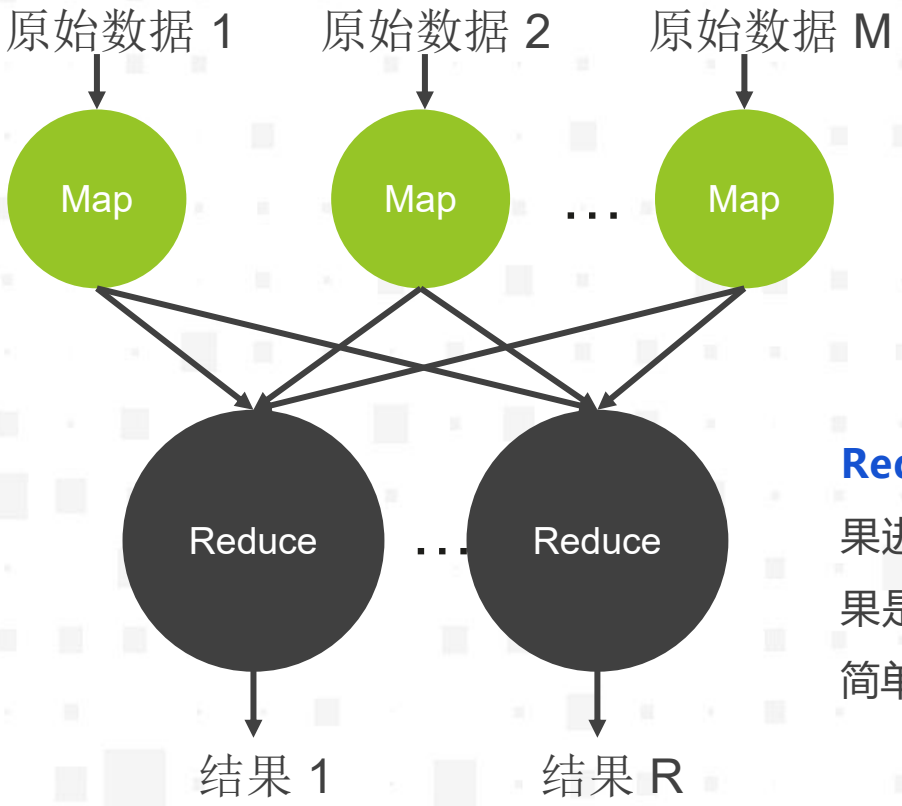
与传统的分布式程序设计相比，MapReduce封装了**并行处理、容错处理、本地化计算、负载均衡等细节，还提供了一个简单而强大的接口**。

MapReduce把对数据集的大规模操作，**分发给一个主节点管理下的各分节点**共同完成，通过这种方式**实现任务的可靠执行与容错机制**。

3.2 分布式数据处理MapReduce

3.2 分布式数据处理MapReduce

- 编程模型



Map函数——对一部分原始数据进行指定的操作。每个Map操作都针对不同的原始数据，因此Map与Map之间是互相独立的，这使得它们可以充分并行化。

Reduce操作——对每个Map所产生的一部分中间结果进行合并操作，每个Reduce所处理的Map中间结果是互不交叉的，所有Reduce产生的最终结果经过简单连接就形成了完整的结果集。

3.2 分布式数据处理MapReduce

- 编程模型

Map: (in_key, in_value) → {(key_j, $value_j$) | j = 1...k}

Reduce: (key, [$value_1$,...,$value_m$]) → (key, final_value)

Map输入参数：in_key和in_value，它指明了Map需要处理的原始数据

Map输出结果：一组<key,value>对，这是经过Map操作后所产生的中间结果

Reduce输入参数：
（key,[$value_1$,...,$value_m$]）

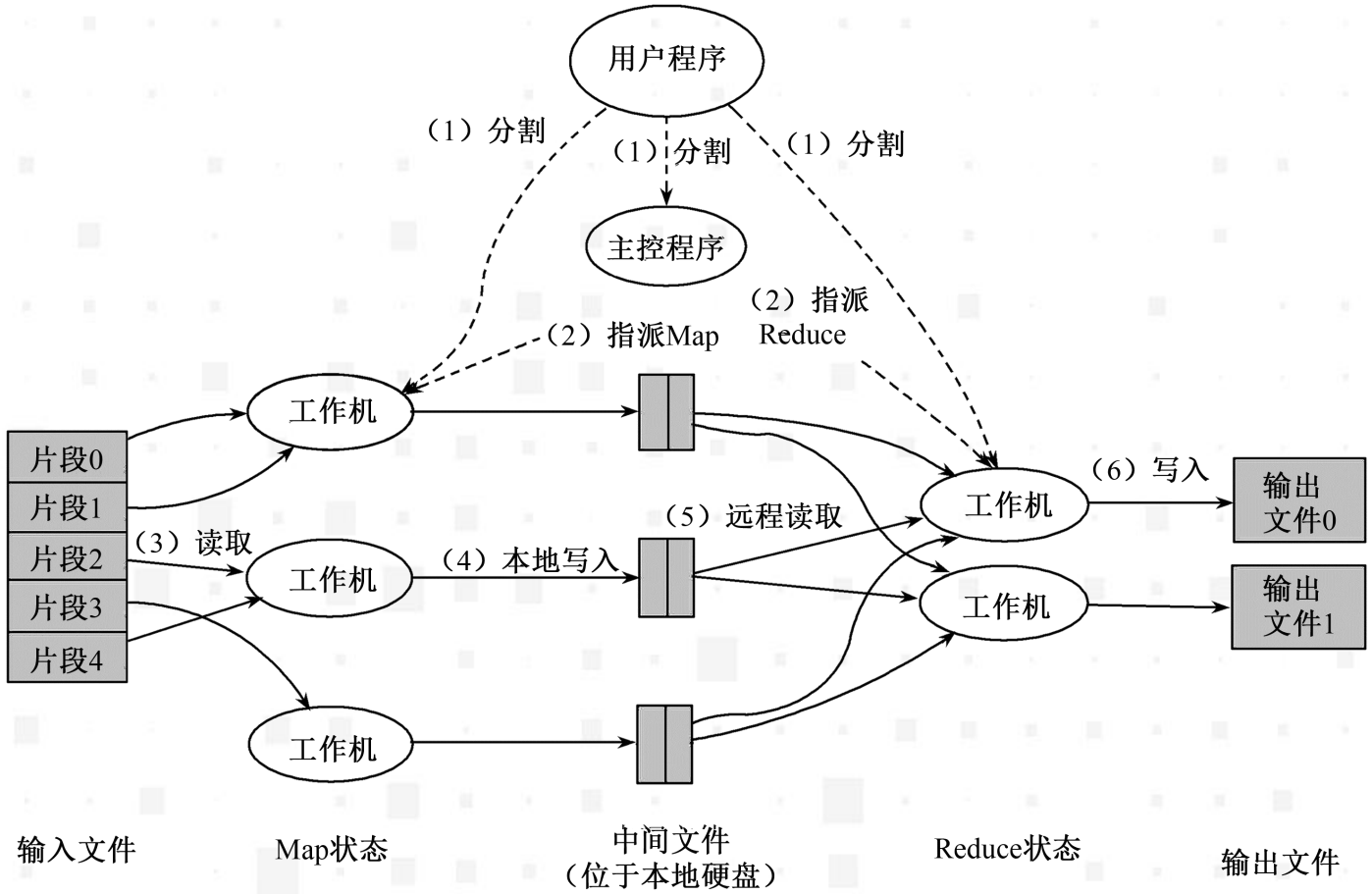
Reduce工作：
对这些对应相同key的value值进行归并处理

Reduce输出结果：
（key, final_value），所有Reduce的结果并在一起就是最终结果

3.2 分布式数据处理MapReduce

3.2 分布式数据处理MapReduce

- 实现机制



3.2 分布式数据处理MapReduce

- 实现机制

（1）MapReduce函数首先把**输入文件分成M块**

（2）分派的执行程序中有一个**主控程序Master**

（3）一个被分配了Map任务的Worker读取并处理相关的输入块

（4）这些缓冲到内存的中间结果将被定时写到本地硬盘，这些**数据通过分区函数分成R个区**

（5）当Master通知执行Reduce的Worker关于中间<key,value>对的位置时，它调用远程过程，从Map Worker的本地硬盘上读取缓冲的中间数据

（6）Reduce Worker根据每一个唯一中间key来遍历所有的排序后的中间数据，并且把key和相关的中间结果值集合传递给用户定义的Reduce函数

（7）当所有的Map任务和Reduce任务都完成的时候，Master激活用户程序

3.2 分布式数据处理MapReduce

- 容错机制

由于MapReduce在成百上千台机器上处理海量数据，所以容错机制是不可或缺的。总的来说，MapReduce通过重新执行失效的地方来实现容错。

Master失效

Master会周期性地设置检查点（checkpoint），并导出Master的数据。一旦某个任务失效，系统就从最近的一个检查点恢复并重新执行。

由于只有一个Master在运行，如果Master失效了，则只能终止整个MapReduce程序的运行并重新开始。

Worker失效

Master会**周期性地给Worker发送ping**命令，如果没有Worker的应答，则Master认为Worker失效，终止对这个Worker的任务调度，把失效Worker的任务调度到其他Worker上重新执行。

3.2 分布式数据处理MapReduce

3.2.1 产生背景

3.2.2 编程模型

3.2.3 实现机制

▶ 3.2.4 案例分析

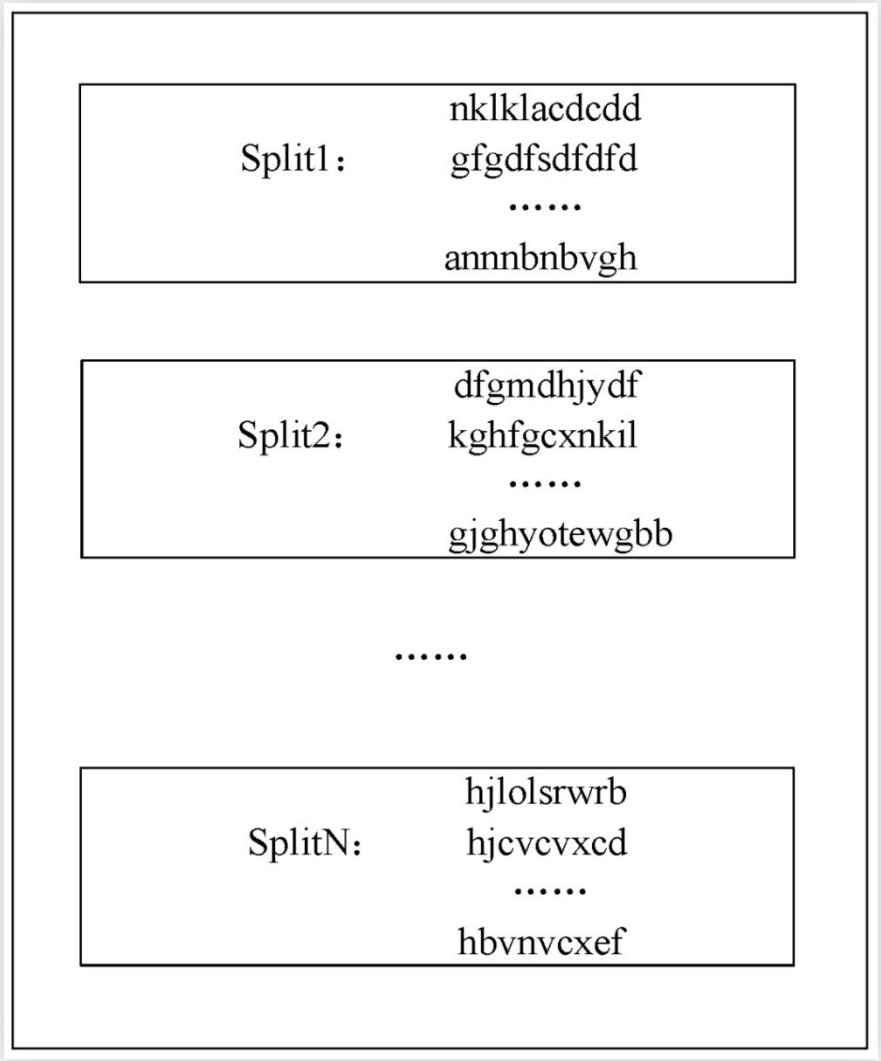
3.2 分布式数据处理MapReduce

怎样通过MapReduce完成排序工作，使其有序（字典序）呢?

3.2 分布式数据处理MapReduce

- 第一个步骤

对原始的数据进行分割（Split），得到N个不同的数据分块 。

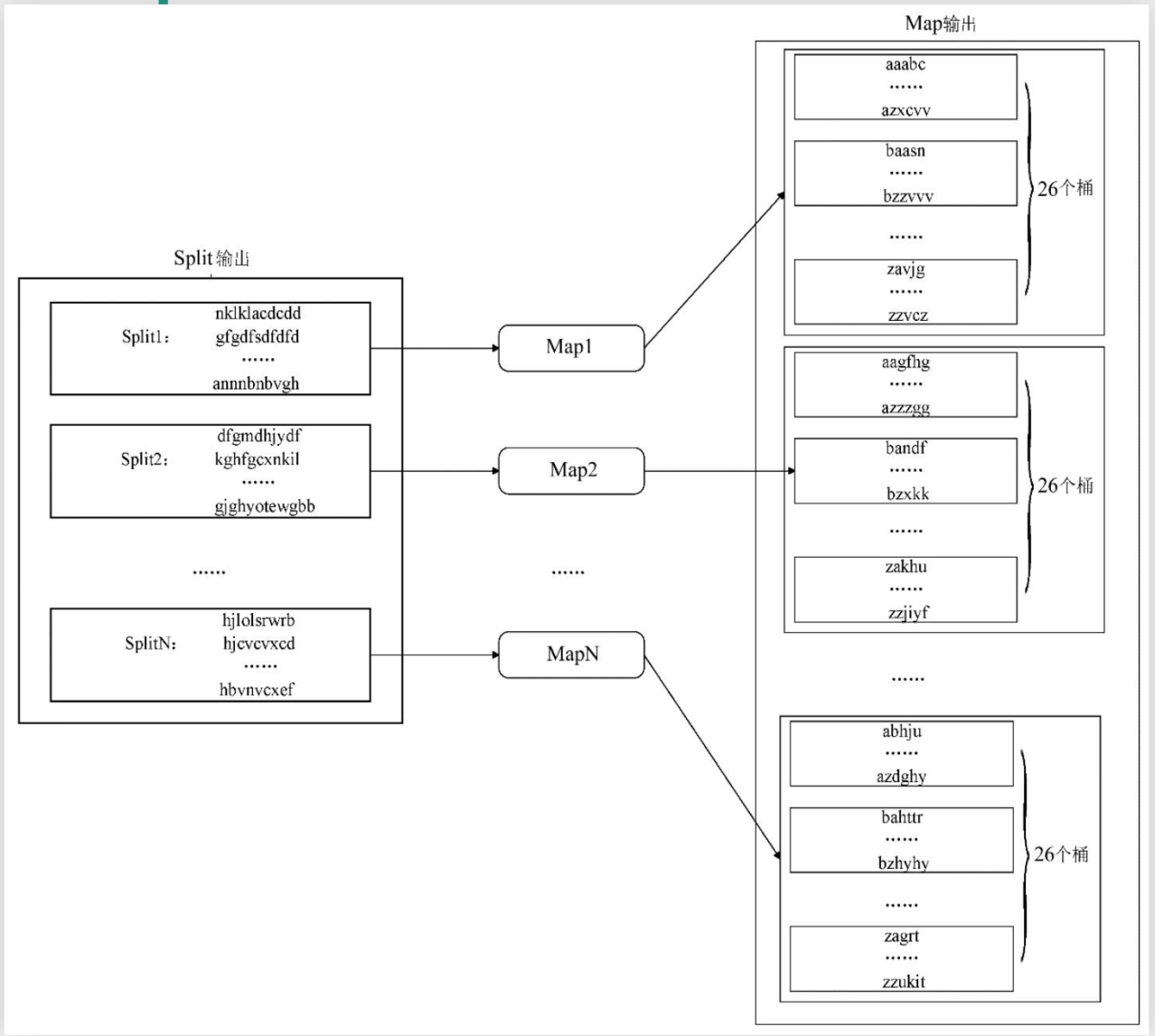


3.2 分布式数据处理MapReduce

- 第二个步骤

对每一个数据分块都启动一个Map进行处理。

采用桶排序的方法，每个Map中按照首字母将字符串分配到26个不同的桶中。

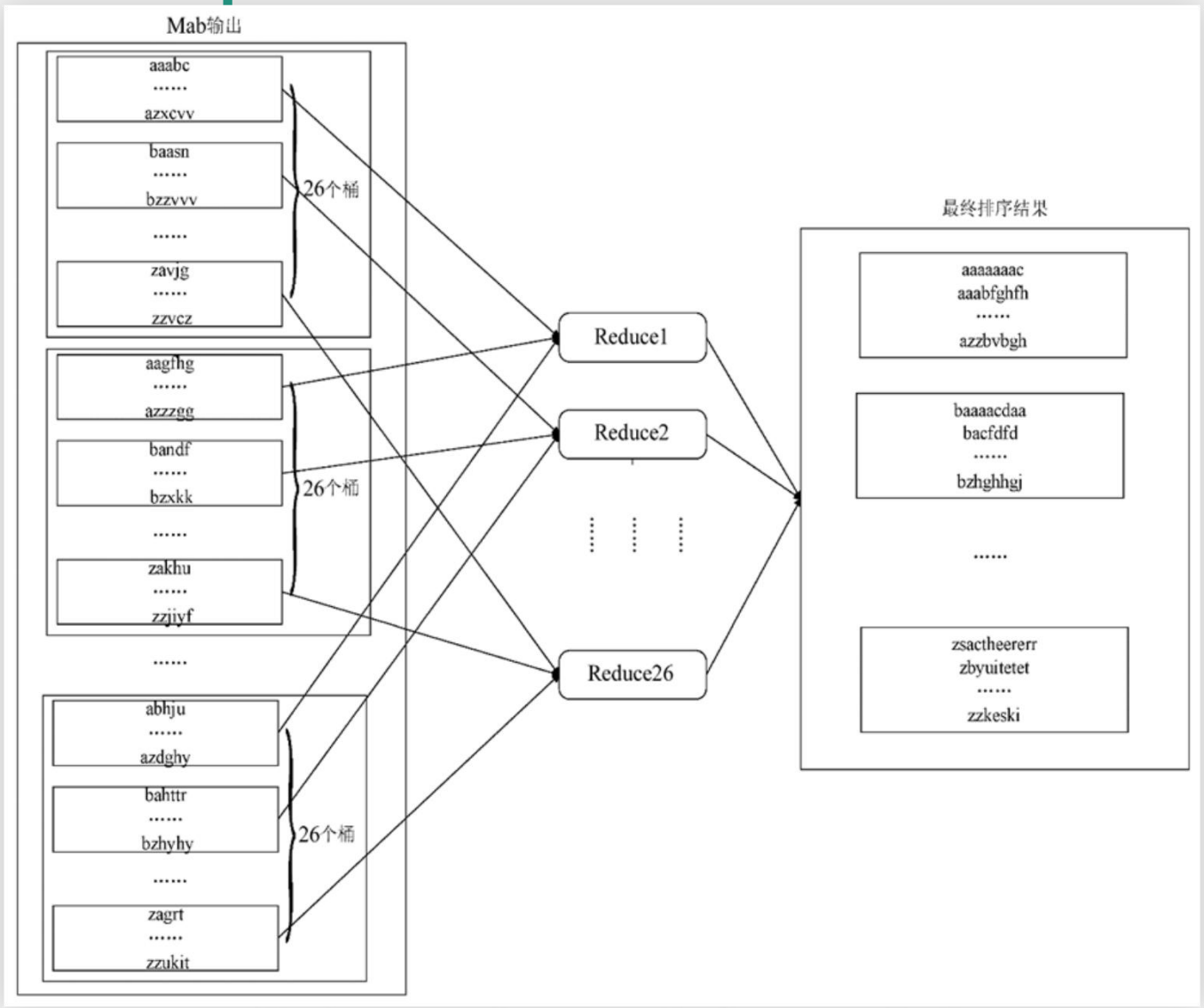


3.2 分布式数据处理MapReduce

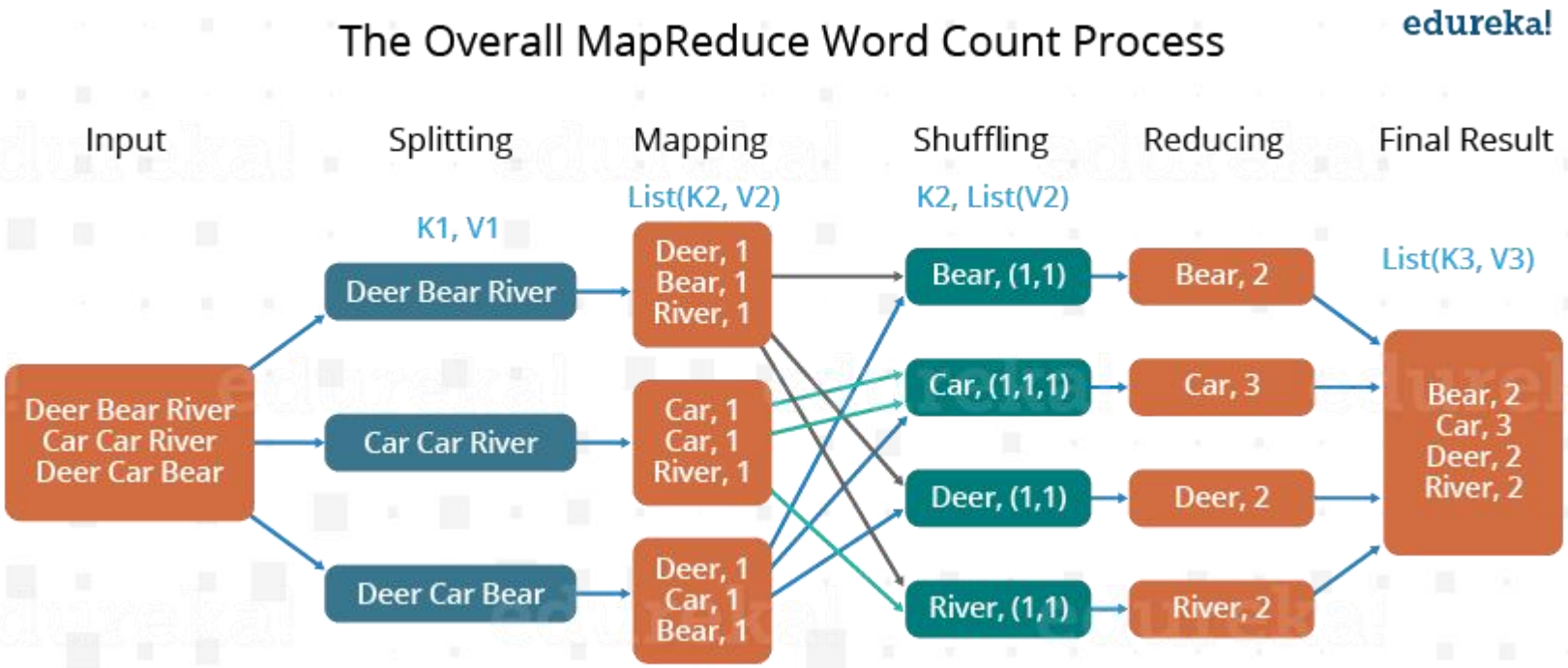
- 第三个步骤

对于Map之后得到的中间结果，启动26个Reduce。

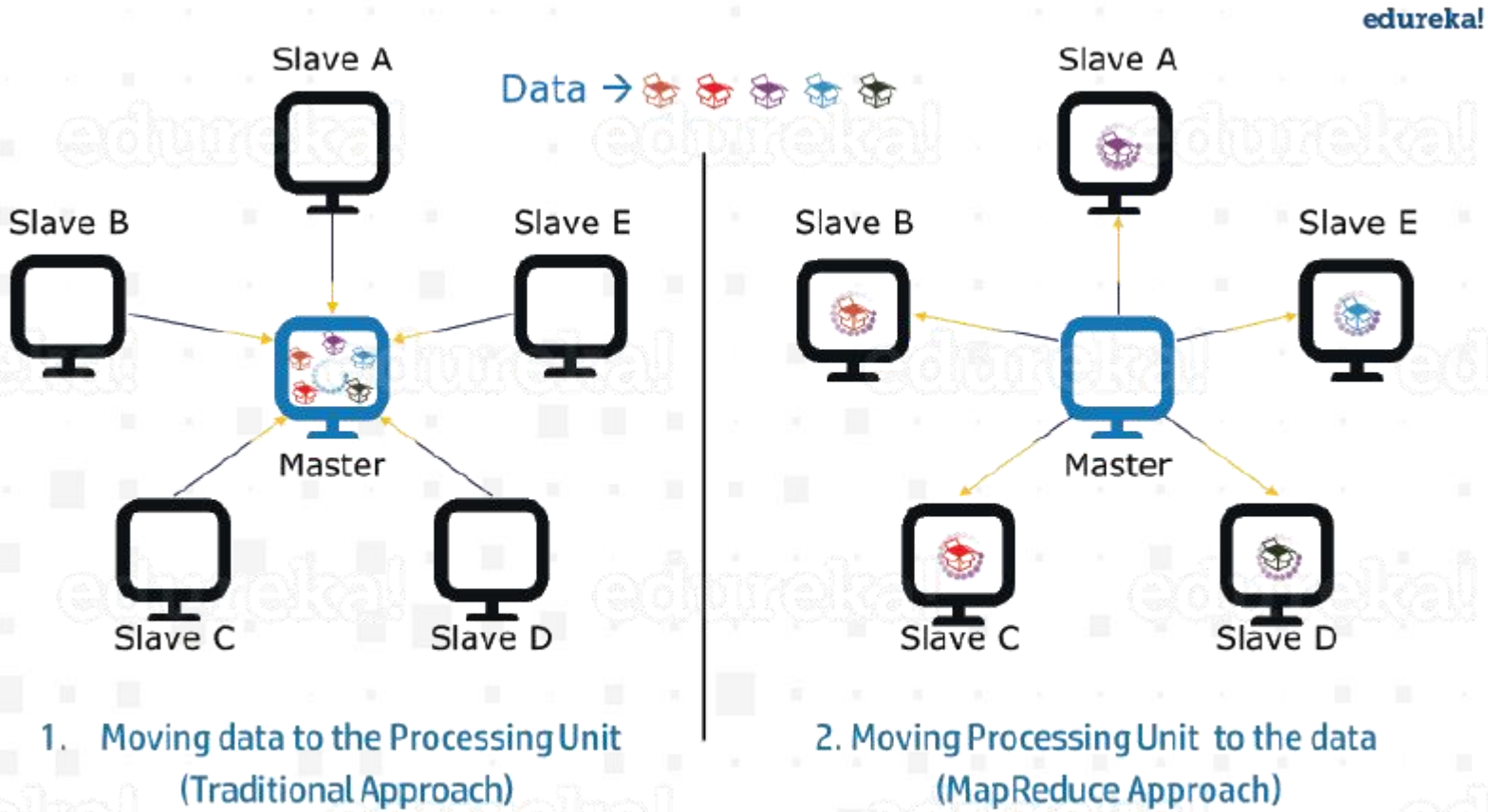
按照首字母将Map中不同桶中的字符串集合放置到相应的Reduce中进行处理。



3.2 分布式数据处理MapReduce

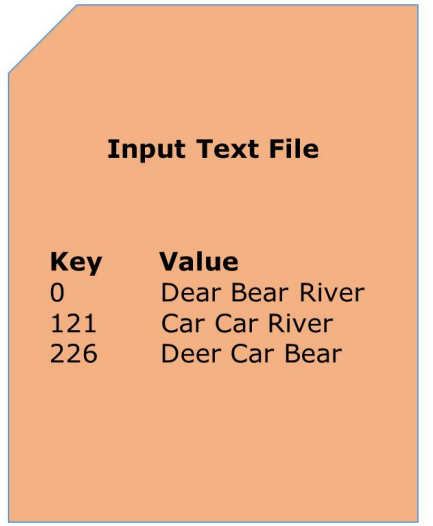


3.2 分布式数据处理MapReduce



3.2 分布式数据处理MapReduce

```
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {
    public void map(LongWritable key, Text value, Context context)
            throws IOException,InterruptedException      {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            value.set(tokenizer.nextToken());
            context.write(value, new IntWritable(1));
        }
    }
}
```



3.2 分布式数据处理MapReduce

```
public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable>{
     public void reduce(Text key, Iterable<IntWritable> values, Context context)
          throws IOException,InterruptedException {
               int sum=0;
               for(IntWritable x: values)
               {
                    sum+=x.get();
               }
               context.write(key, new IntWritable(sum));
          }
     }
}
```

3.2 分布式数据处理MapReduce

```
public class WordCount{
    public static class Map  ......
    public static class Reduce ......
    public static void main(String[] args) throws Exception {
        Configuration conf= new Configuration();
        Job job = new Job(conf,"My Word Count Program");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        Path outputPath = new Path(args[1]);
        //Configuring the input/output path from the filesystem into the job
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        //deleting the output path automatically from hdfs so that we don't have to delete it explicitly
        outputPath.getFileSystem(conf).delete(outputPath);
        //exiting the job only if the flag value becomes false
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

3.2 分布式数据处理MapReduce