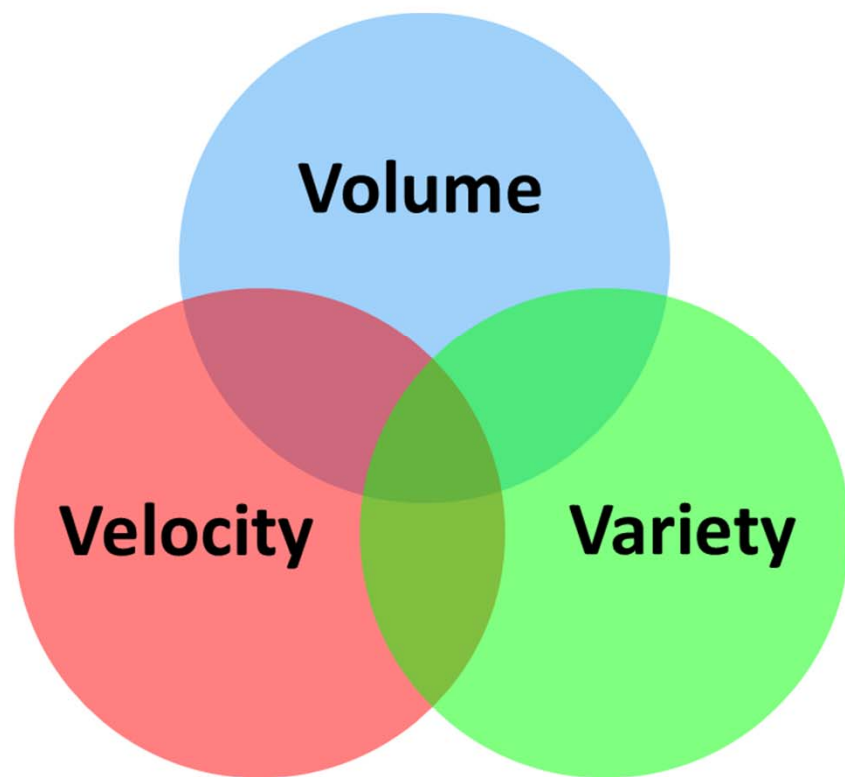


大数据系统与大规模数据分析

# 大数据运算系统 (1)



陈世敏

中科院计算所  
计算机体系结构  
国家重点实验室

©2015-2021 陈世敏

# 作业时间安排

周次	内容	作业
第4周, 3/31	大数据存储系统1: 基础, 文件系统, HDFS	作业1布置
第5周, 4/7	大数据存储系统2: 键值系统	
第6周, 4/14	大数据存储系统3: 图存储, document store	
第7周, 4/21	大数据运算系统1: MapReduce, 图计算系统	作业1提交 作业2布置
第8周, 4/28	大数据运算系统2: 图计算系统, MR+SQL	
第9周, 5/5	大数据运算系统3: 内存计算系统	大作业布置 (系统,6人/组)
第10周, 5/12	分布式哈希表, 区块链技术中的加密算法	作业2提交
第11周, 5/19	最邻近搜索和位置敏感 (LSH) 算法	作业3
第12周, 5/26	奇异值分解与数据空间的维度约化	大作业布置 (分析,3人/组)
第13周, 6/2	推荐系统	大作业 仅选1个
第14周, 6/9	流数据采样与估计、流数据过滤与分析	
第15周, 6/16	期末考试	
第16周, 6/23	大作业验收报告 (上下午, 教1-208)	大作业验收

# Outline

- MapReduce/Hadoop
  - 编程模型
  - 系统实现
  - 典型算法
- Microsoft Dryad
- 同步图计算系统

# MapReduce/Hadoop 简介

- MapReduce是目前云计算中最广泛使用的计算模型

- 由Google于2004年提出
- “*MapReduce: Simplified Data Processing on Large Clusters*”. Jeffrey Dean and Sanjay Ghemawat (Google). **OSDI 2004**.

- Hadoop是MapReduce的一个开源实现



- 2005年由Doug Cutting and Mike Cafarella 开始了Hadoop项目
- 2006年Cutting成为Yahoo Lab的员工
- Hadoop的开发主要由Yahoo Lab推动，后来成为Apache开源项目
- 基于Java
- 已经被广泛使用：Yahoo, Facebook, Twitter, Linkedin, Ebay, AOL, Hulu, 百度, 腾讯, 阿里, 天涯社区, .....

# MapReduce编程模型

- 整体思路
- 数据模型
- Map-shuffle-Reduce
- Word count举例
- 与SQL Select语句的关系

# 整体思路

- 并行分布式程序设计不容易
  - ❑ Multi-threading
  - ❑ Socket programming
  - ❑ Data distribution
  - ❑ Job distribution, coordination, load balancing
  - ❑ Fault tolerance
  - ❑ Debugging
- 需要有经验的程序员+编程调试时间
  - ❑ 上面每个方面都需要学习和经验积累
  - ❑ 调试分布式系统很花时间和精力

# 整体思路

- 解决思路

- 程序员写串行程序
- 由系统完成并行分布式地执行

在本课中，程序员是指使用大数据平台实现上层应用功能的人员

串行程序

Map

Reduce

并行分布式执行

MapReduce 系统

# 整体思路

- 解决思路
  - 程序员写串行程序
  - 由系统完成并行分布式地执行
- 程序员保证串行程序的正确性
  - 编程序时不需要思考并行的问题
  - 调试时只需要保证串行执行正确
- 系统负责并行分布执行的正确性和效率
  - Multi-threading, Socket programming, Data distribution, Job distribution, coordination, load balancing, Fault tolerance
- 有什么问题吗？



# 整体思路

- 有什么问题吗？
- 牺牲了程序的功能！
  - 直接进行并行分布式编程，可以完成各种各样丰富的功能
  - 而一个编程模型实际上是限定了程序的功能类型
- 推论1：系统的编程模型必须有代表性
  - 必须代表一大类重要的应用才有生命力
- 推论2：一个成功的模型也就不可避免地被人们应用于原本不适合的情形
  - 需要扩展，需要新的模型
  - 这是后话

# MapReduce的数据模型

- <key, value>

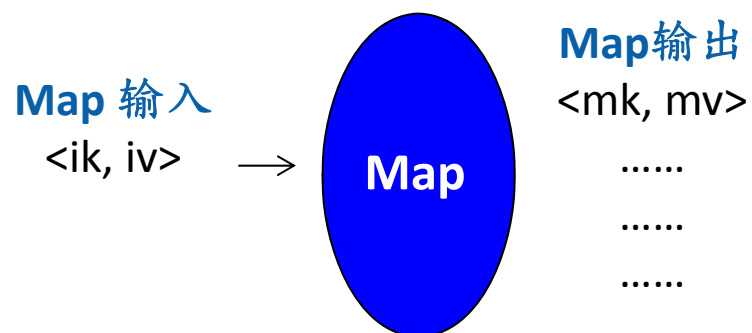
- 数据由一条一条的记录组成
- 记录之间是无序的
- 每一条记录有一个key，和一个value
- key: 可以不唯一
- key与value的具体类型和内部结构由程序员决定，系统基本上把它们看作黑匣

# MapReduce

$\text{Map}(ik, iv) \rightarrow \{ \langle mk, mv \rangle \}$

$\text{Reduce}(mk, \{mv\}) \rightarrow \{ \langle ok, ov \rangle \}$

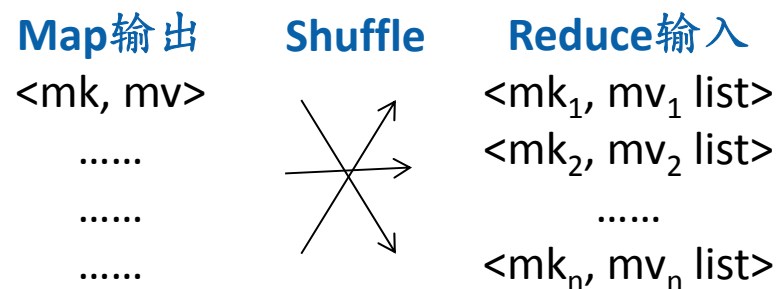
# Map函数



$\text{Map}(\text{ik}, \text{iv}) \rightarrow \{<\text{mk}, \text{mv}>\}$

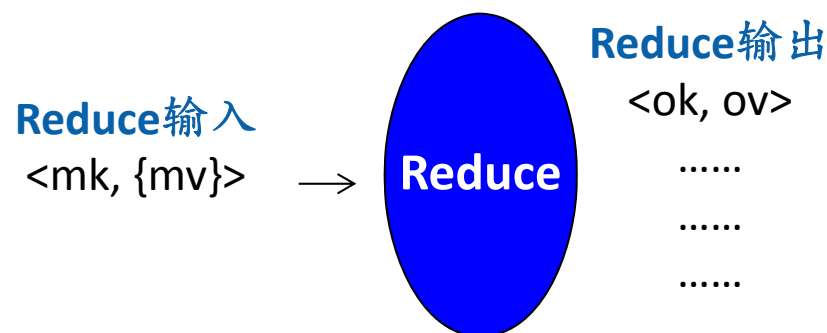
- 输入是一个key-value记录: <ik, iv>
  - 我们用'i'代表input
- 输出是0~多个key-value记录: <mk, mv>
  - 我们用'm'代表intermediate
- 注意: mk与ik很可能完全不同

# Shuffle (由系统完成)



- Shuffle = group by mk
- 对于所有的map函数的输出，进行group by
- 将相同mk的所有mv都一起提供给Reduce

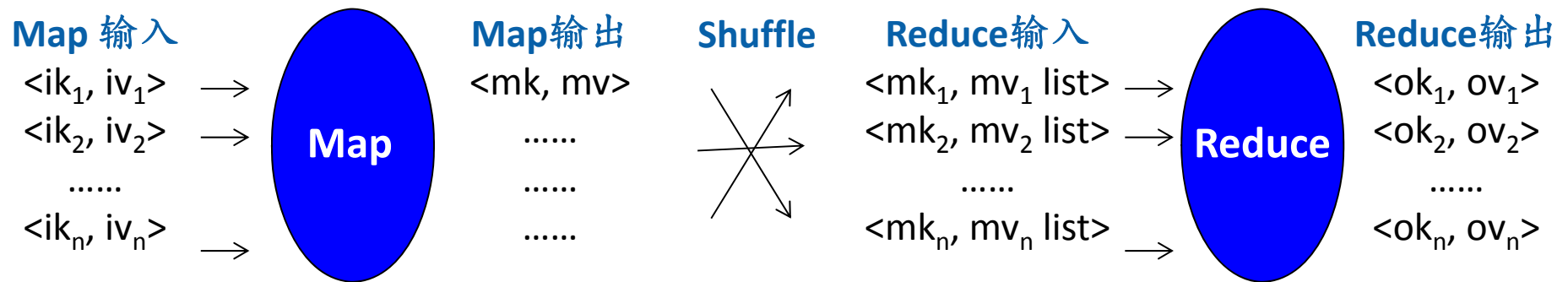
# Reduce函数



$\text{Reduce}(mk, \{mv\}) \rightarrow \{\langle ok, ov \rangle\}$

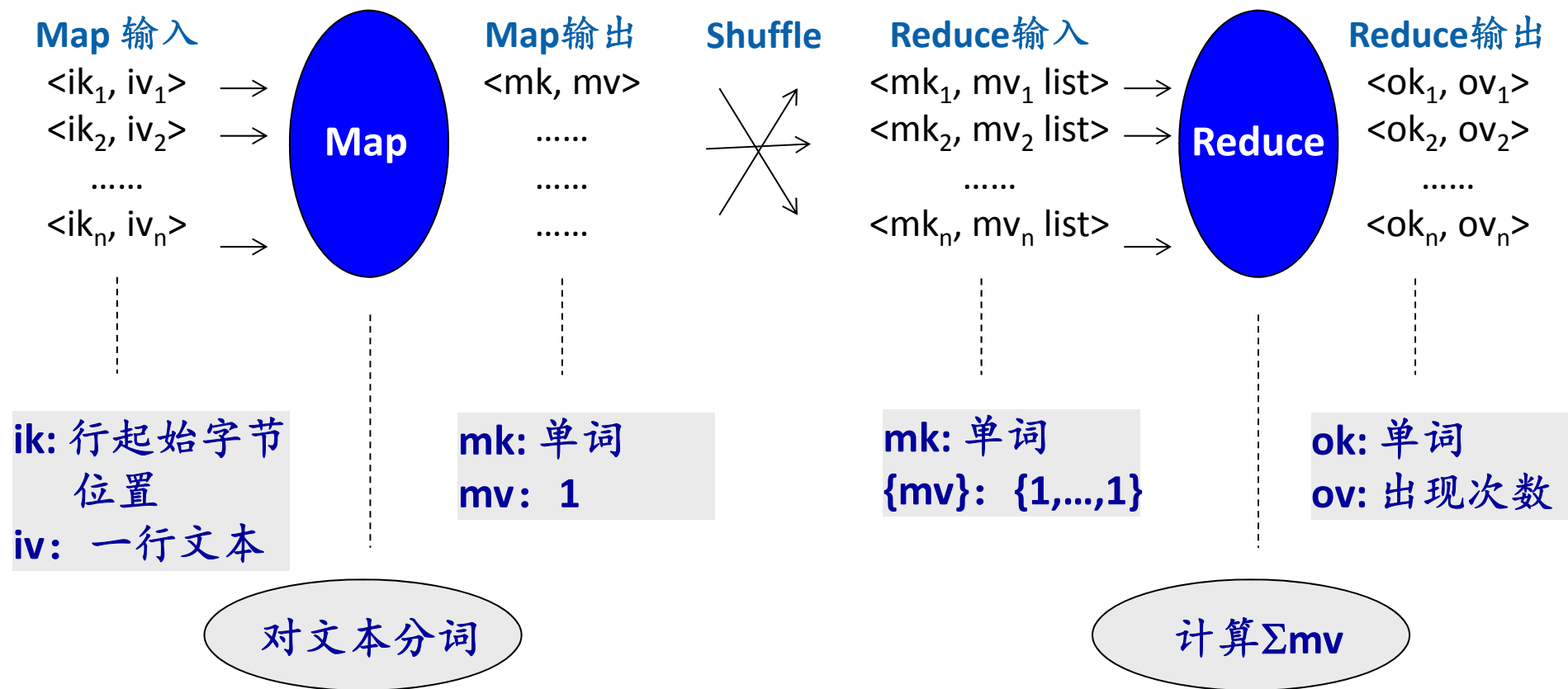
- 输入是一个mk和与之对应的所有mv
- 输出是0~多个key-value记录:  $\langle ok, ov \rangle$ 
  - 我们用'o'代表output
- 注意: ok与mk可能不同

# Map-shuffle-Reduce



- 程序员编制串行的Map函数和Reduce函数
- 系统完成shuffle功能
  - shuffle = group by mk

# Map-shuffle-Reduce: word count 举例



Word count: 统计文本中每个单词出现的次数



# Hadoop Mapper.java

(简化了 exception handling)

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    protected void setup(Context context) {
        // NOTHING
    }
    protected void map(KEYIN key, VALUEIN value, Context context){
        context.write((KEYOUT) key, (VALUEOUT) value);
    }
    protected void cleanup(Context context) {
        // NOTHING
    }
    public void run(Context context) { //Hadoop调用run
        setup(context);
        while (context.nextKeyValue()) {
            map(context.getCurrentKey(), context.getCurrentValue(), context);
        }
        cleanup(context);
    }
}
```

# Hadoop Reducer.java

(简化了 exception handling)

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    protected void setup(Context context) {
        // NOTHING
    }
    protected void reduce(
        KEYIN key, Iterable<VALUEIN> vlist, Context context) {
        for(VALUEIN v: vlist) context.write((KEYOUT)key, (VALUEOUT)v);
    }
    protected void cleanup(Context context) {
        // NOTHING
    }
    public void run(Context context) { //Hadoop调用run
        setup(context);
        while (context.nextKey()) {
            reduce(context.getCurrentKey(), context.getValues(), context);
        }
        cleanup(context);
    }
}
```

# WordCount.java

```
public class WordCount {  
  
    public static class TokenizerMapper  
        extends Mapper<Object, Text, Text, IntWritable>{  
  
        private final static IntWritable one = new IntWritable(1);  
        private Text word = new Text();  
  
        public void map(Object key, Text value, Context context  
                        ) throws IOException, InterruptedException {  
            StringTokenizer itr = new StringTokenizer(value.toString());  
            while (itr.hasMoreTokens()) {  
                word.set(itr.nextToken());  
                context.write(word, one);  
            }  
        }  
    }  
}
```

# WordCount.java

```
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
        Context context
        ) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

# WordCount.java

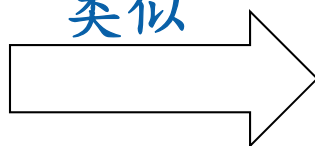
```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf,
                                                    args).getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: wordcount <in> <out>");
        System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

# 比较

## MapReduce

- Map
- Shuffle
- Reduce
- 选择的功能更加丰富
  - 程序实现的
  - 类似最简单的SQL select
  - 但不支持join

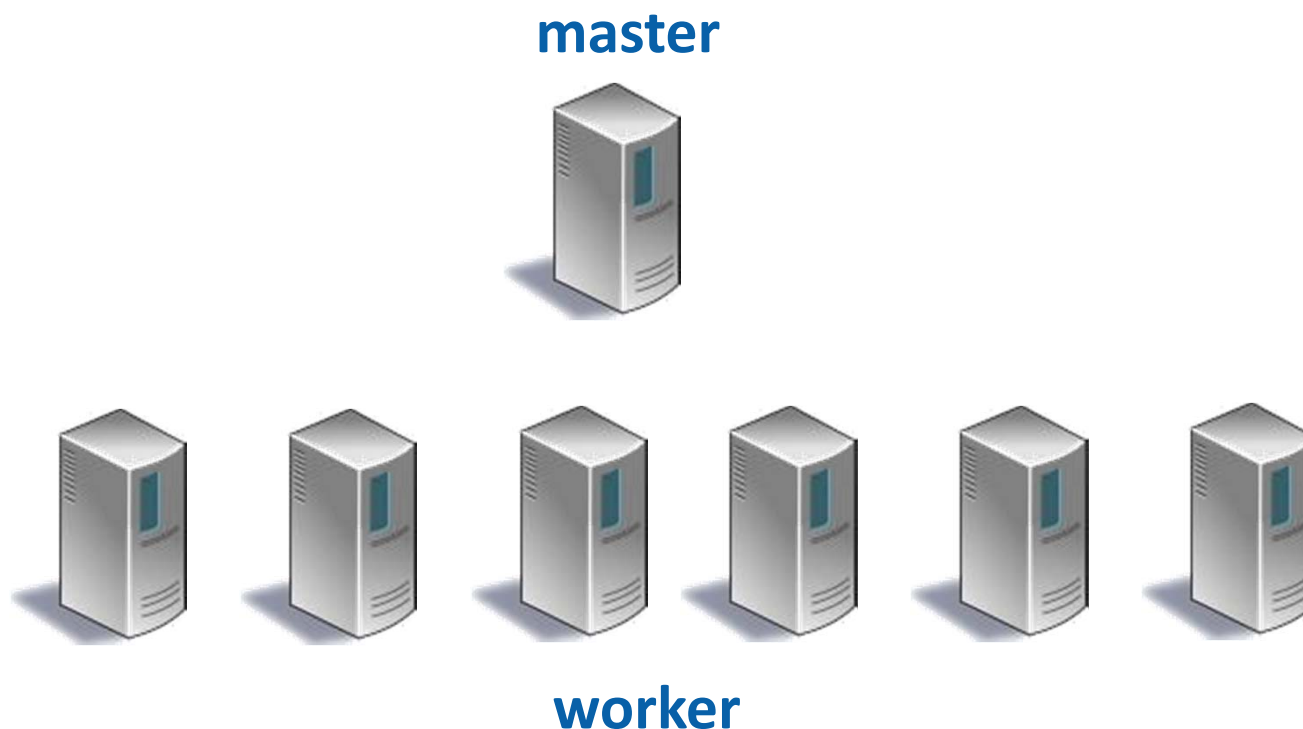
类似



## SQL Select

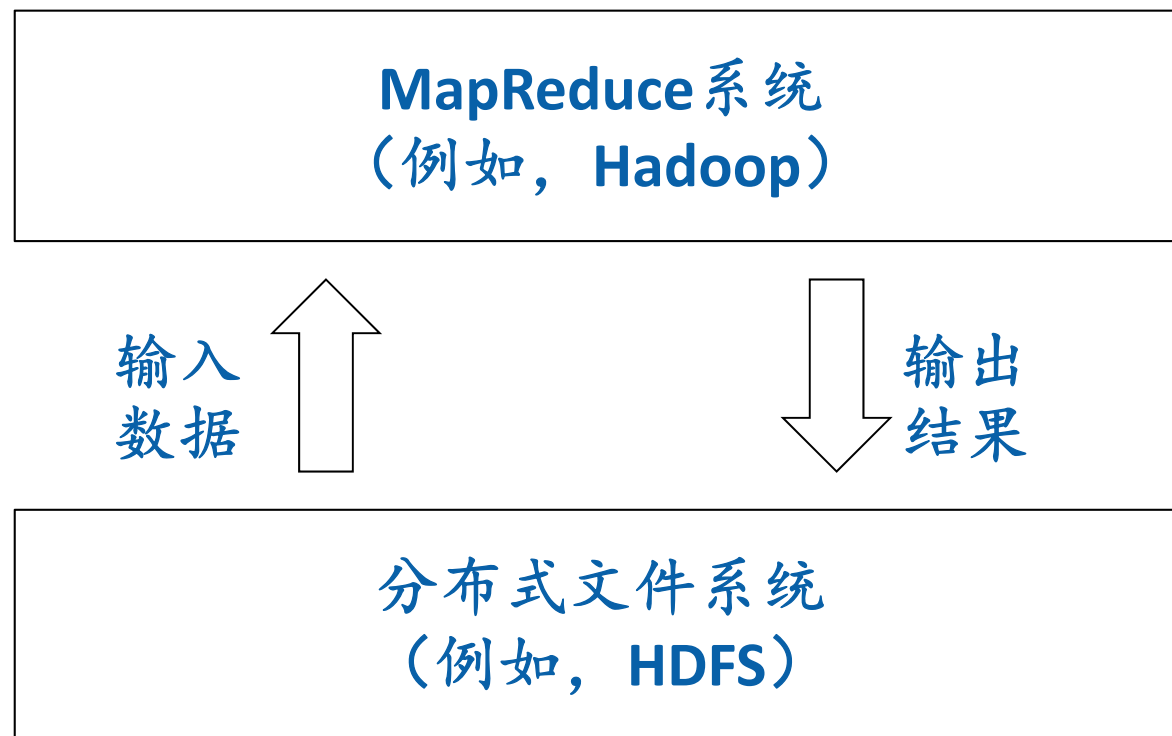
- Selection/projection
- Group by
- Aggregation, Having
- 功能由数据类型和SQL语言标准定义
  - 有UDF: user defined function
  - 但支持得不好

# MapReduce系统架构



在OSDI'04文章中，基本上是1个master对应  
100~1000数量级的workers

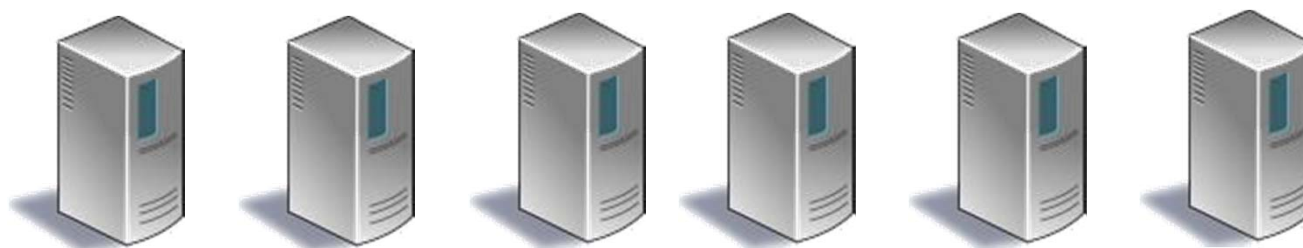
# 系统架构





# MapReduce / Hadoop 系统架构

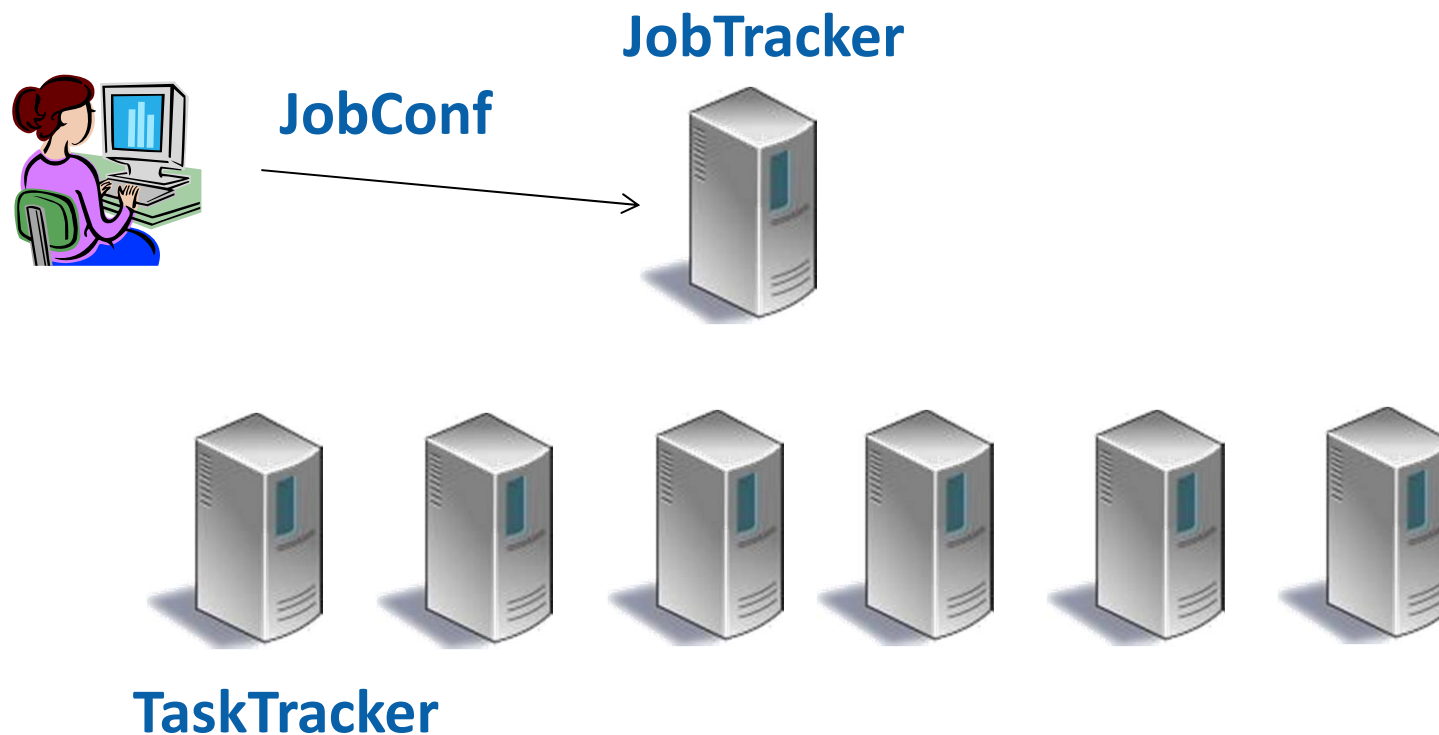
**JobTracker** 控制协调作业的运行



**TaskTracker:** 执行Map Task 或Reduce Task

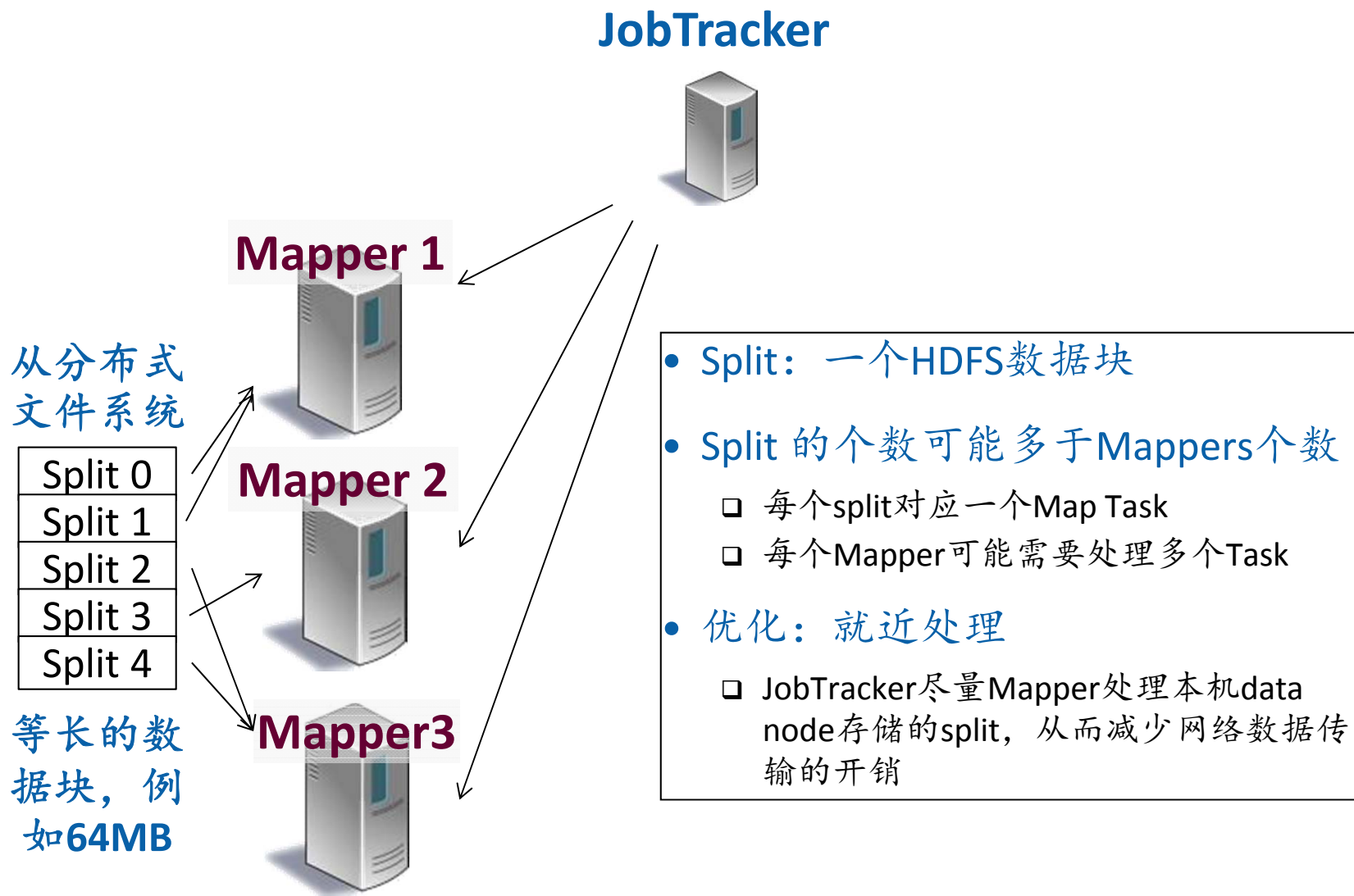
- 注意：JobTracker, TaskTracker, Name Node, Data Node都是进程，所以可以在一台机器上同时运行JobTracker/Name Node, TaskTracker/Data Node
- Hadoop 2.x采用YARN代替了JobTracker，但功能大同小异

# MR运行：提交作业

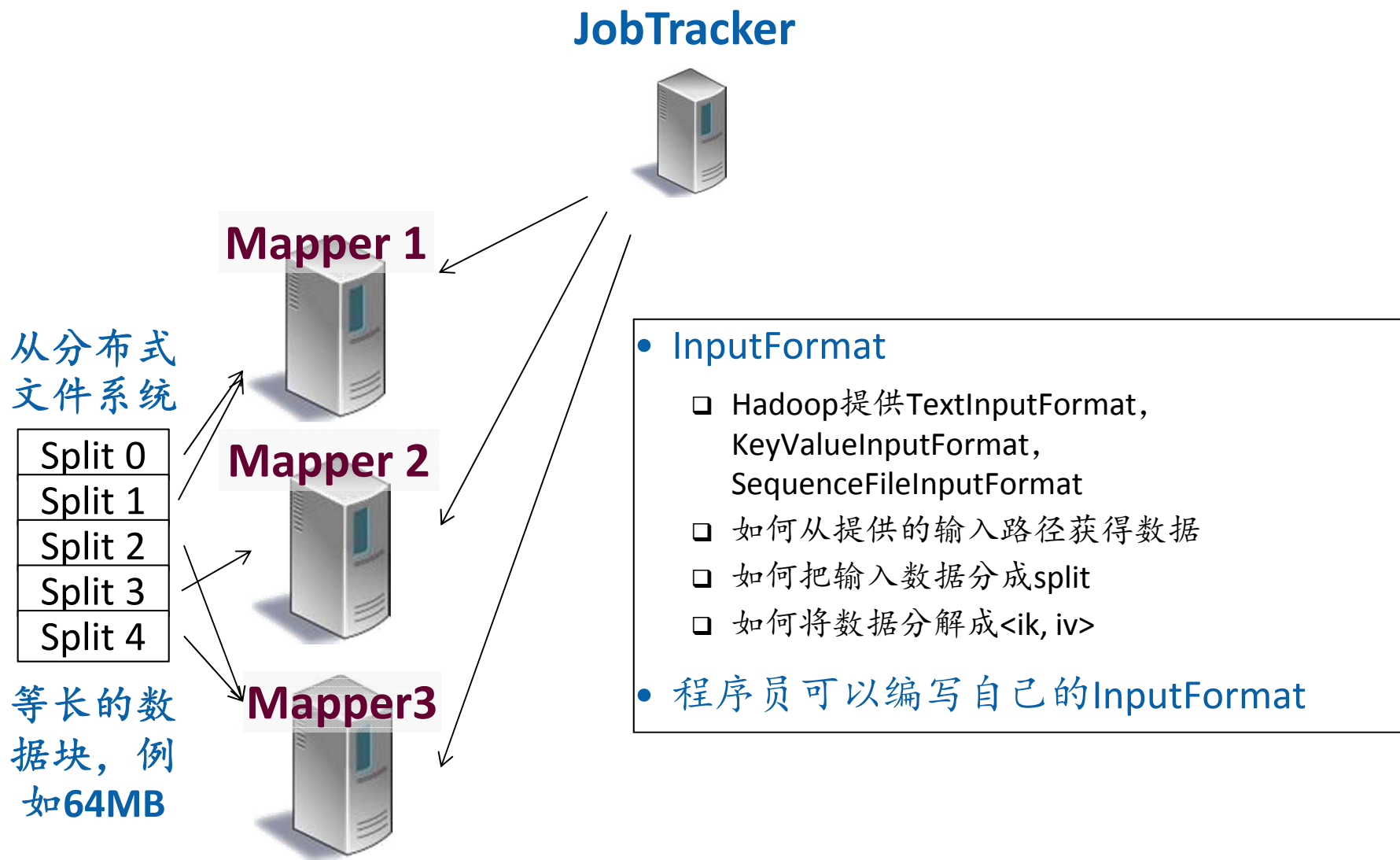


- 包括Map函数、Reduce函数(Jar)、配置信息(例如，几个Mappers，几个Reducers)、输入路径、输出路径等

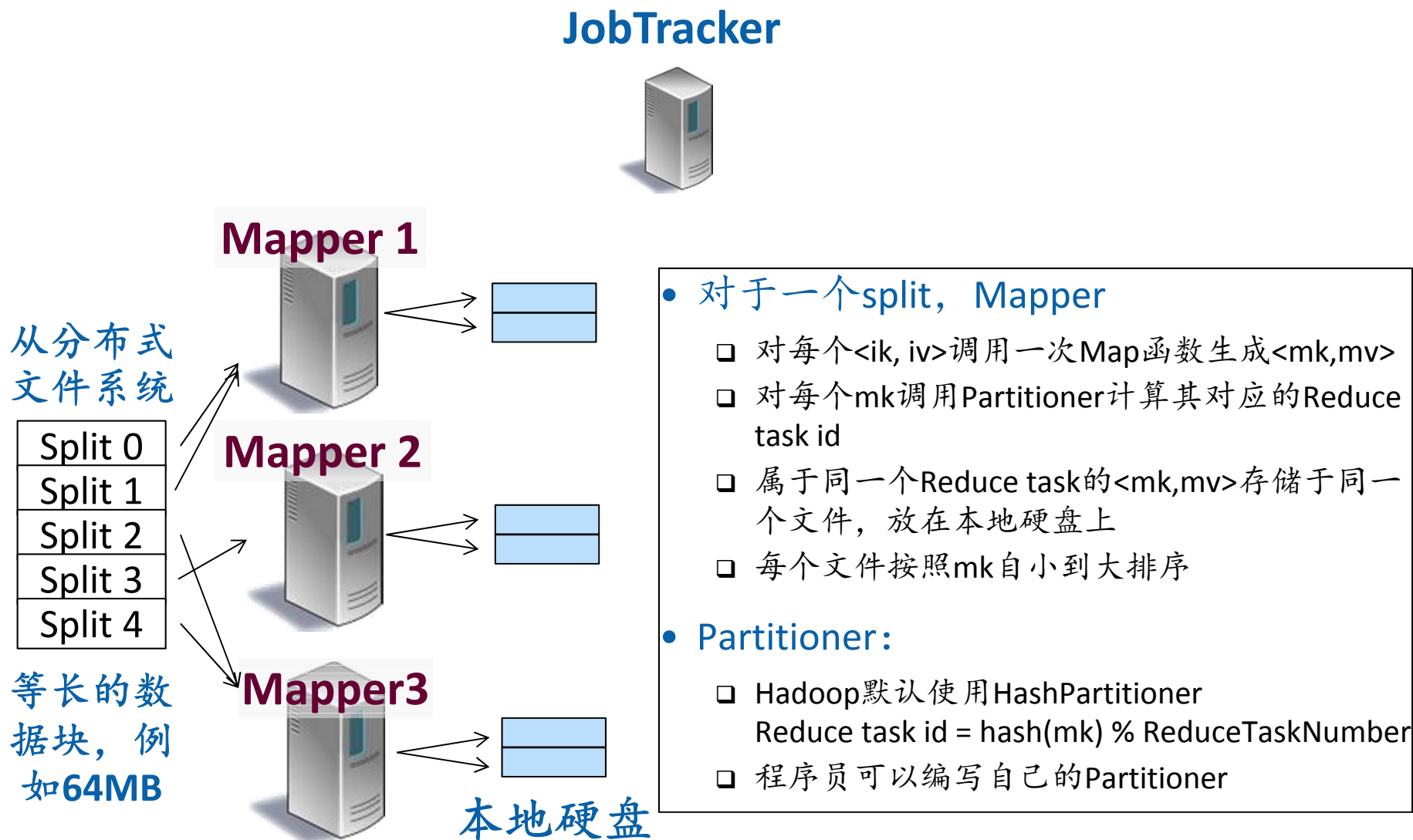
# MR运行：Map Task 读数据



# MR运行：Map Task 读数据



# MR运行：Map Task 执行

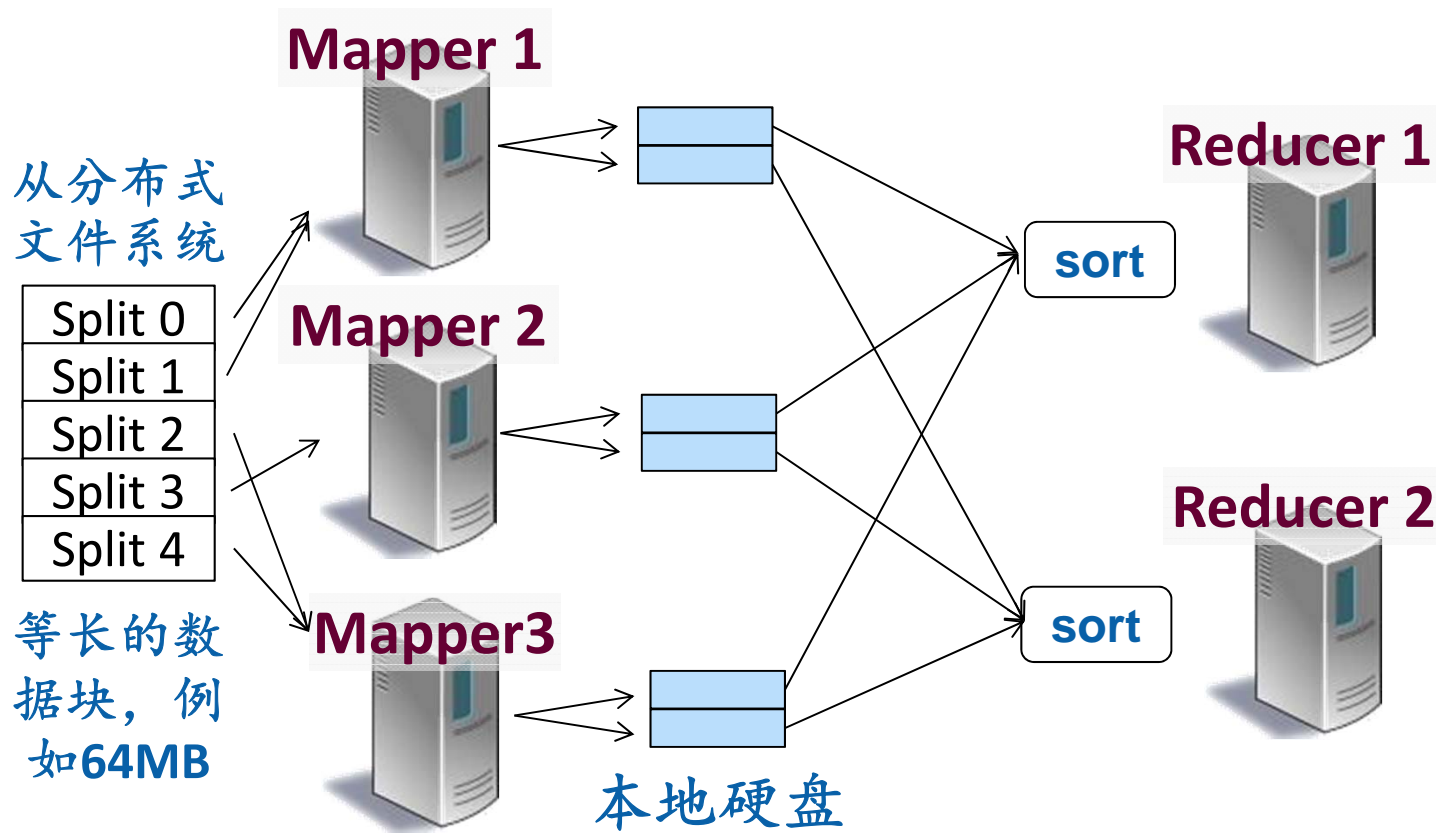


# MR运行：Shuffle

JobTracker



- Reducer从每个Map task传输中间结果文件
  - 每个文件本身已经排好序了
- 对多个结果文件进行归并，从而实现group by

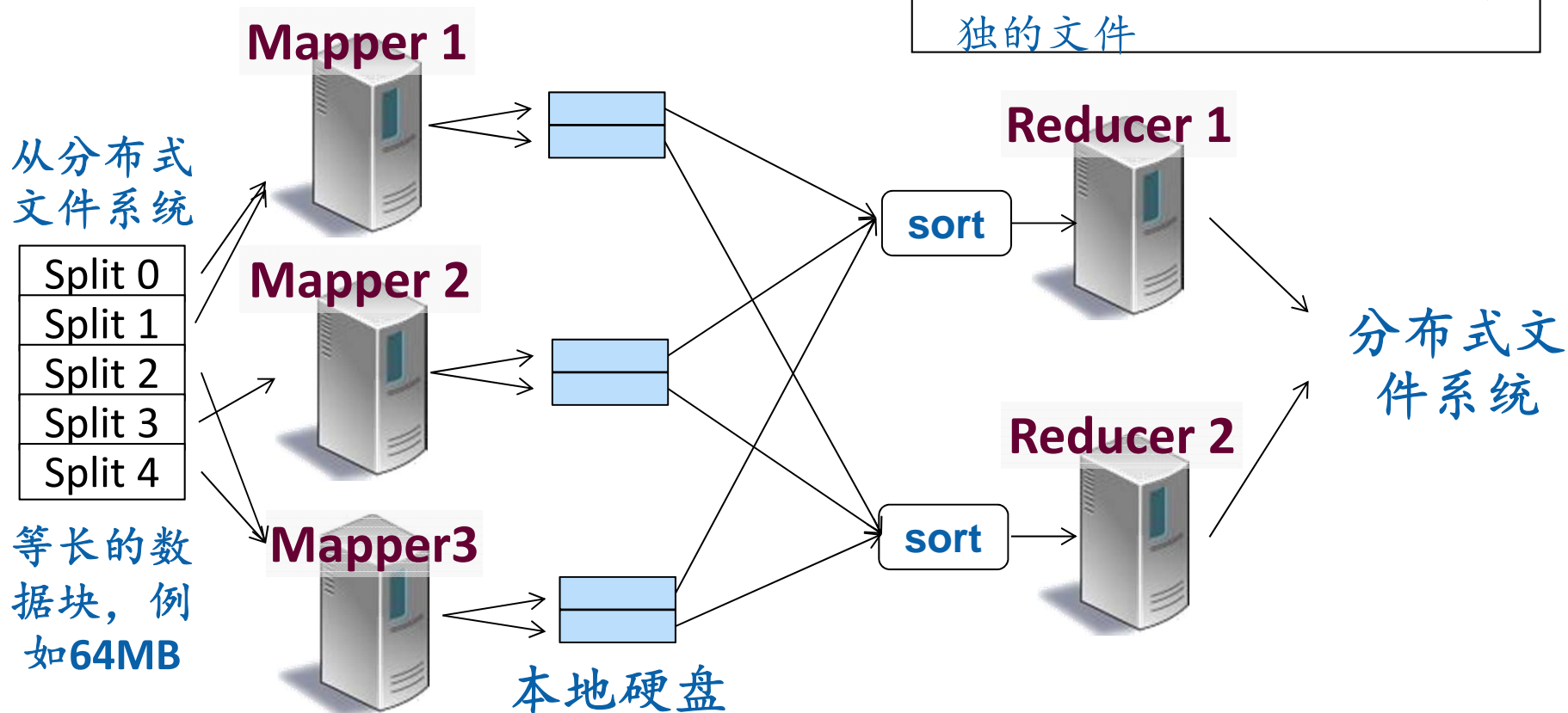


# MR运行：Reduce

JobTracker



- 对于每个<mk,{mv}>调用一次 Reduce函数
- 产生的<ok, ov>写入输出文件
- OutputFormat
- 每个Reduce task 产生一个单独的文件



# Combiner

- Combiner = partial reducer

- $\text{Combiner}(\text{mk}, \{\text{mv}\}) \rightarrow \langle \text{mk}, \text{mv}' \rangle$

- 回顾 word count

- 传输很多  $\langle \text{mk}, 1 \rangle$  似乎有些浪费

- 如果在 Mapper 一侧，一个 mk 出现多次，完全可以进行本地的累计，这样只需要传输一个  $\langle \text{mk}, \text{本地次数} \rangle$

- 这可以用 Combiner 实现

ik: 行起始字节  
位置  
iv: 一行文本

mk: 单词  
mv: 1

mk: 单词  
{mv}: {1,...,1}

ok: 单词  
ov: 出现次数

对文本分词

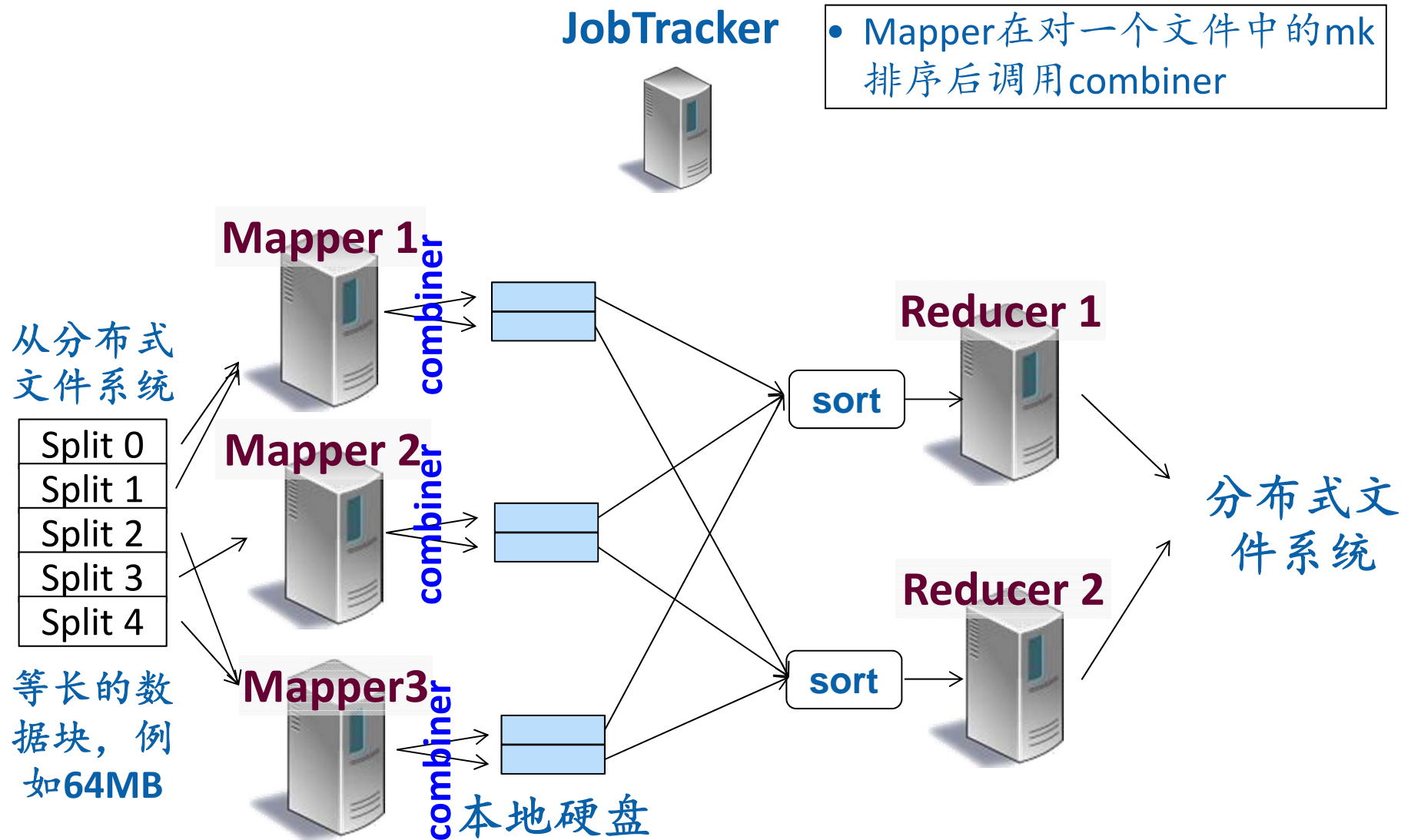
Map

计算  $\sum \text{mv}$

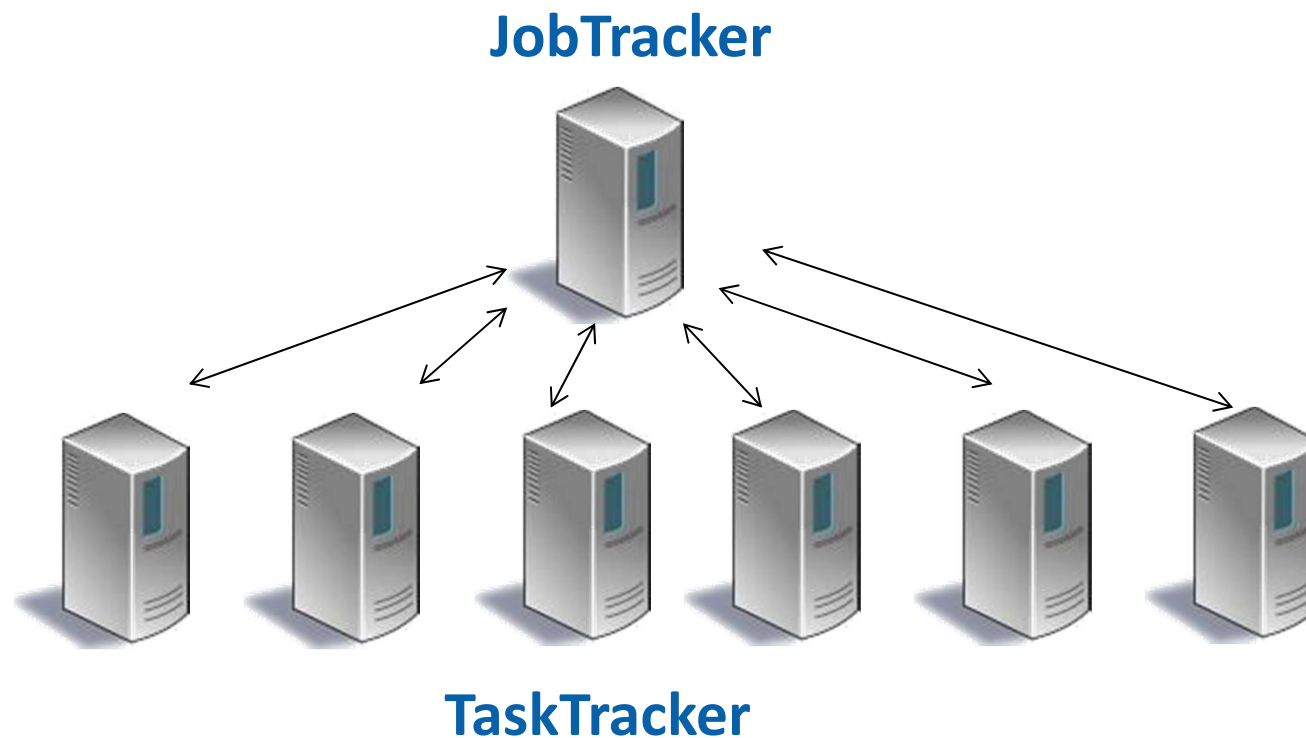
Reduce



# MR运行：Combiner



# MR: Fault Tolerance (容错)



- HeartBeat(心跳)消息
  - 定期发送，向JobTracker汇报进度
- JobTracker可以及时发现不响应的机器或速度非常慢的机器
  - 这些异常机器被称作Stragglers

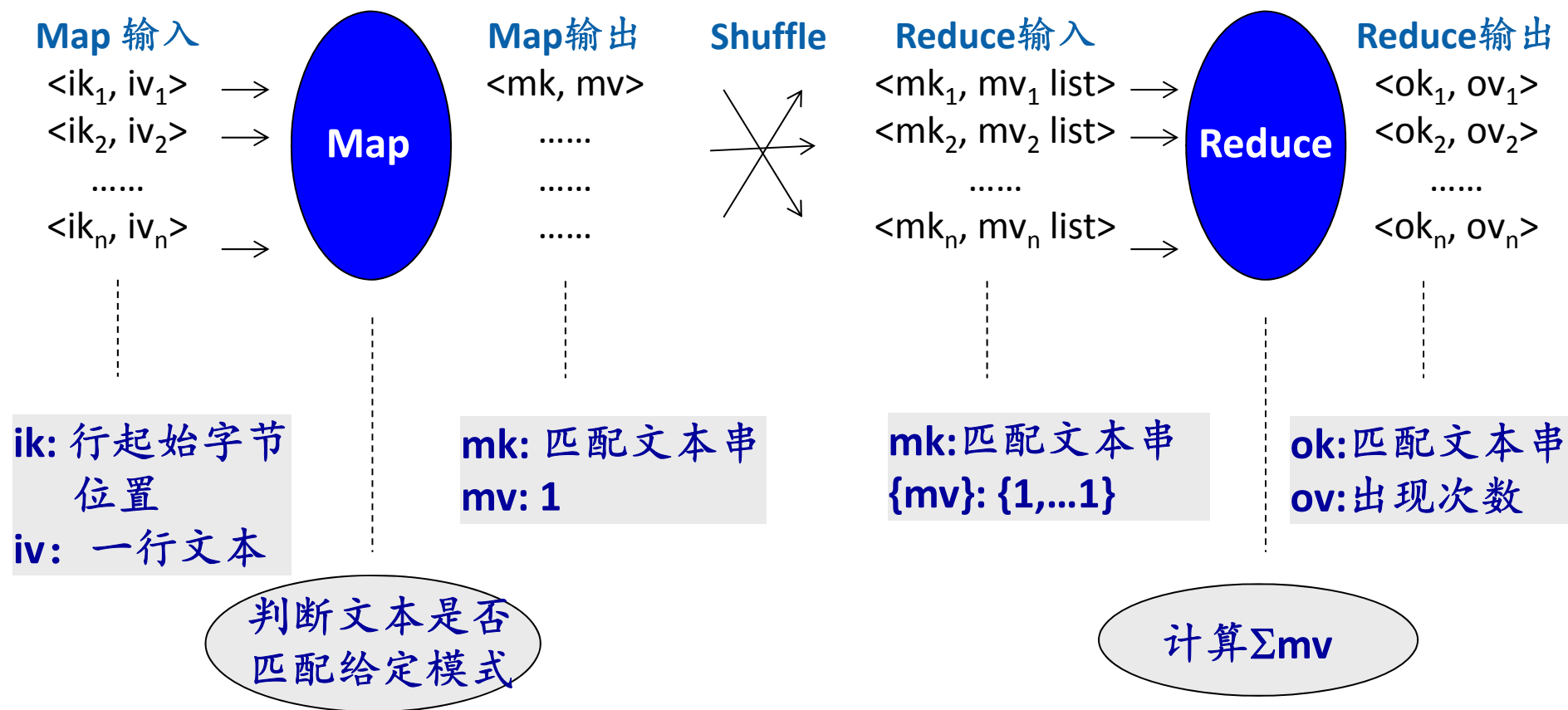
# MR: Fault Tolerance (容错)

- 一旦发现Straggler
  - JobTracker就将它需要做的工作分配给另一个worker
- Straggler是Mapper，将所对应的splits分配给其它的Mapper
  - 输入数据是分布式文件，所以不需要特殊处理
  - 通知所有的Reducer这些splits的新对应Mapper
  - Shuffle时从新对应的Mapper传输数据
- Straggler是Reducer，在另一个TaskTracker执行这个Reducer
  - 这个Reducer需要重新从Mappers传输数据
  - 注意：因为Mapper的输出是在本地文件中的，所以可以多次传输

# 典型算法

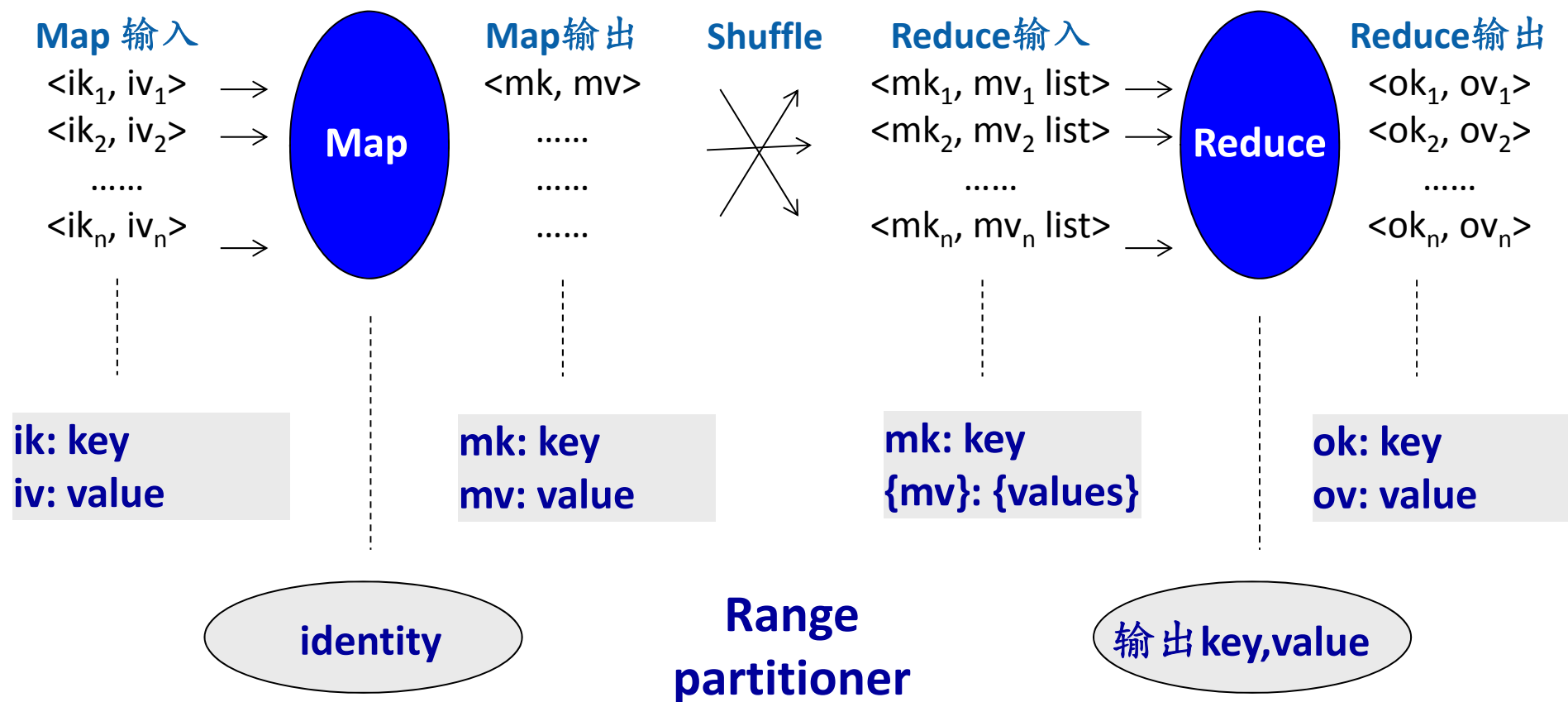
- Grep
- Sorting
- Join

# 举例：Grep (找到符合特定模式的文本)



与 word count 类似

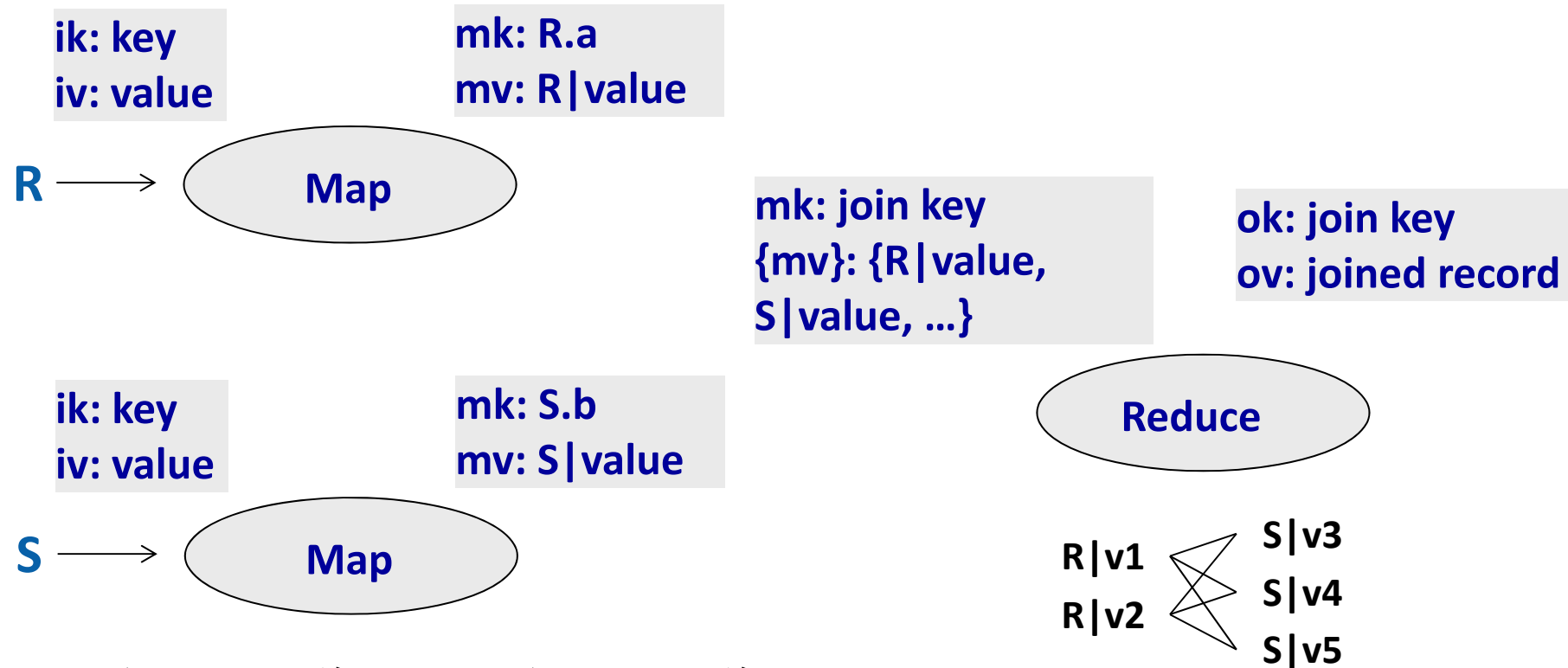
# 举例：Sorting



- 利用MapReduce系统的shuffle/sort功能完成sorting
- identity指将输入拷贝到输出

# 举例：Equi-Join

$$R \bowtie_{R.a = S.b} S$$



- 一组Mapper处理R，一组Mapper处理S
- 利用shuffle/group by把匹配的record放到一起
- Reduce调用时， $\{mv\}$ 包含对应同一个join key的所有匹配的R和S记录，于是产生每一对R记录和S记录的组合（笛卡尔积），并输出
- 需要传输整个R与S会产生比较大的代价

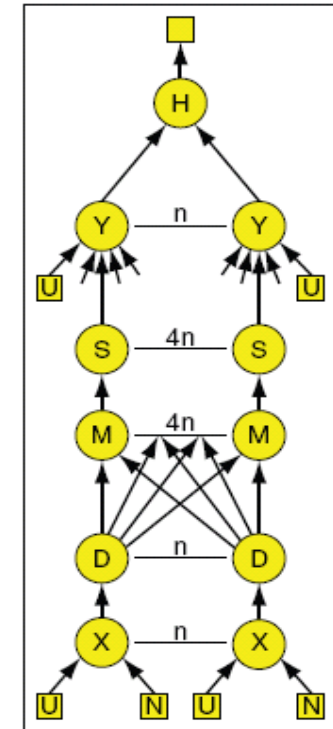
# Outline

- MapReduce/Hadoop
  - 编程模型
  - 系统实现
  - 典型算法
- Microsoft Dryad
- 同步图计算系统



# Microsoft Dryad

- Dryad是对MapReduce模型的一种扩展
  - ❑ 组成单元不仅是Map和Reduce，可以是多种节点
  - ❑ 节点之间形成一个有向无环图DAG(Directed Acyclic Graph)，以表达所需要的计算
  - ❑ 节点之间的数据传输模式更加多样
    - 可以是类似Map/Reduce中的shuffle
    - 也可以是直接1:1、1:多、多:1传输
  - ❑ 比MapReduce更加灵活，但也更复杂
    - 需要程序员规定计算的DAG
- Microsoft内部云计算系统Cosmos基于Dryad



Dryad paper  
[Eurosys'07]

# Outline

- MapReduce/Hadoop

- 编程模型
- 系统实现
- 典型算法

- Microsoft Dryad

- 同步图计算系统

- 图算法
- 同步图计算
- 图计算编程
- 系统实现

# 图(Graph)的概念

- $G=(V, E)$

- $V$ : 顶点(Vertex)的集合

- $E$ : 边(Edge)的集合

- 边  $e=(u,v), u \in V, v \in V$

- 有向图 (directed graph)

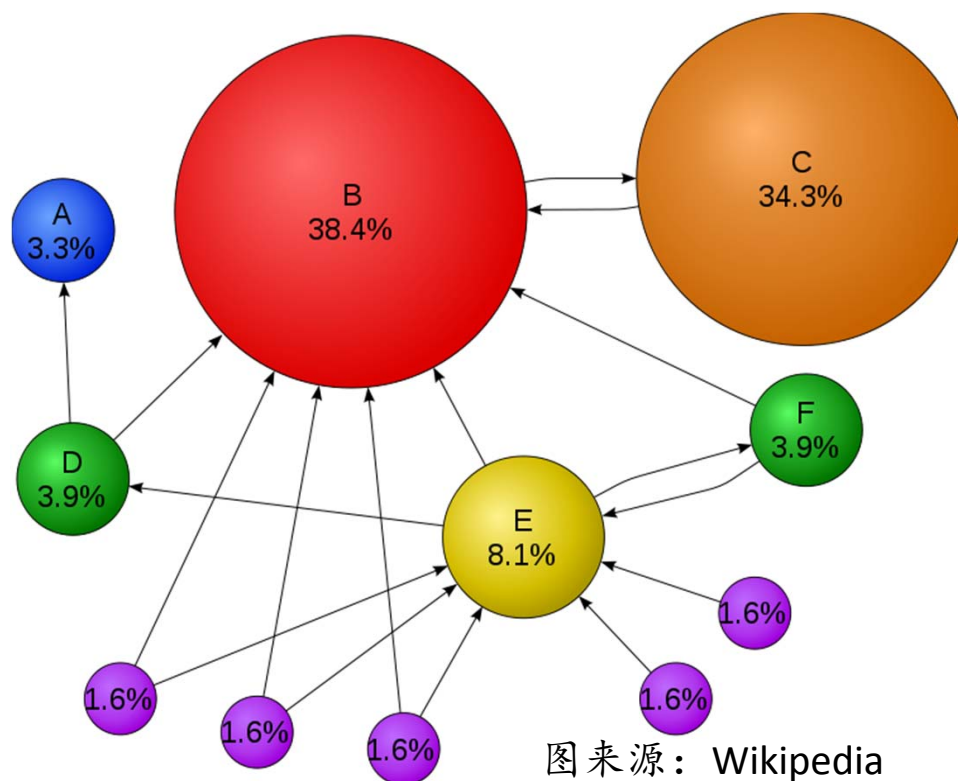
- 边有方向

- 无向图

- 边没有方向

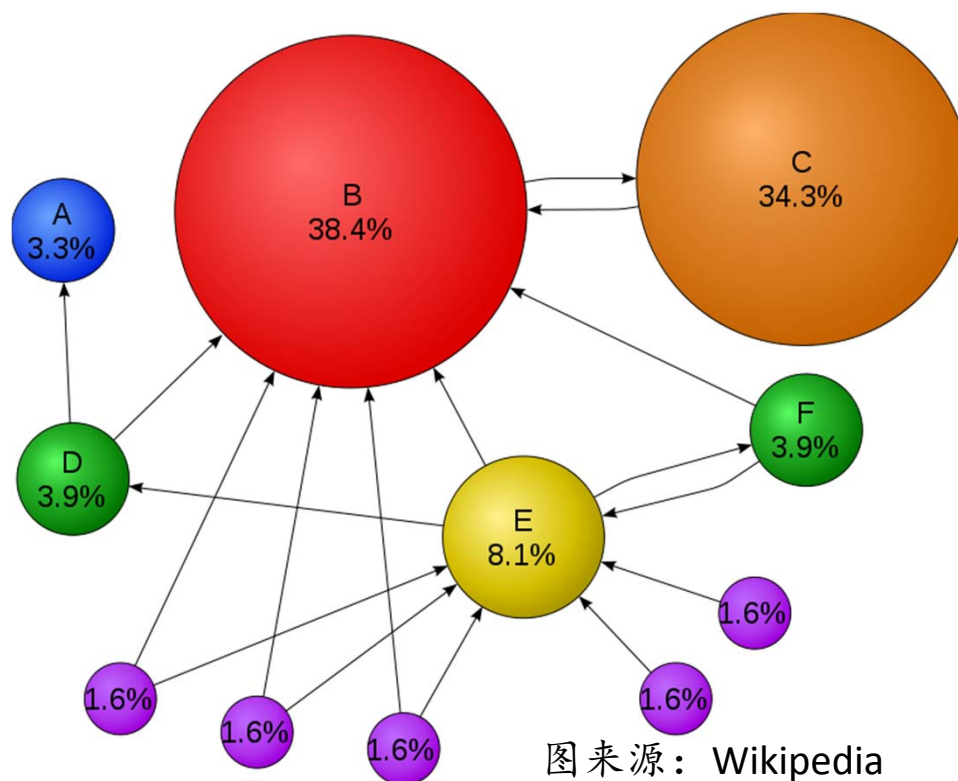
- 可以用有向图表达无向图: 每条无向边  $\rightarrow$  2条有向边

# 图算法举例：PageRank



- Google用于对网页重要性打分的算法
- 上图简单示意了PageRank在一个图上的运行结果
  - 顶点：网页
  - 边：超链接

# 图算法举例：PageRank



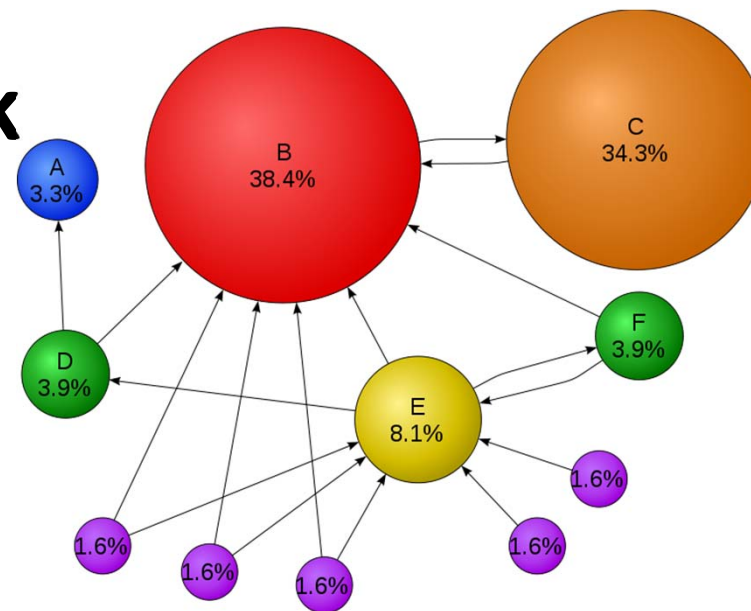
如果没有这种随机跳转, 进入A,B,C后就出不来了

- 用户浏览一个网页时，有85%的可能性点击网页中的超链接，有15%的可能性转向任意的网页
  - PageRank算法就是模拟这种行为
  - $d=85\%$  (damping factor)

# 图算法举例：PageRank

- $$R_u = \frac{1-d}{N} + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

- $R_v$ : 顶点v的PageRank
- $L_v$ : 顶点v的出度（出边的条数）
- $B(u)$ : 顶点u的入邻居集合
- d: damping factor
- N: 总顶点个数



图来源：Wikipedia

## • 计算方法

- 初始化：所有的顶点的PageRank为 $\frac{1}{N}$
- 迭代：用上述公式迭代直至收敛

# 图算法举例：PageRank

- $R_u = \frac{1-d}{N} + d \sum_{v \in B(u)} \frac{R_v}{L_v}$

问题：N非常大时，数据精度可能不够？

- $NR_u = 1 - d + d \sum_{v \in B(u)} \frac{NR_v}{L_v}$

- 设  $R'_u = NR_u$

- $R'_u$  初始化为1

- $R'_u = 1 - d + d \sum_{v \in B(u)} \frac{R'_v}{L_v}$

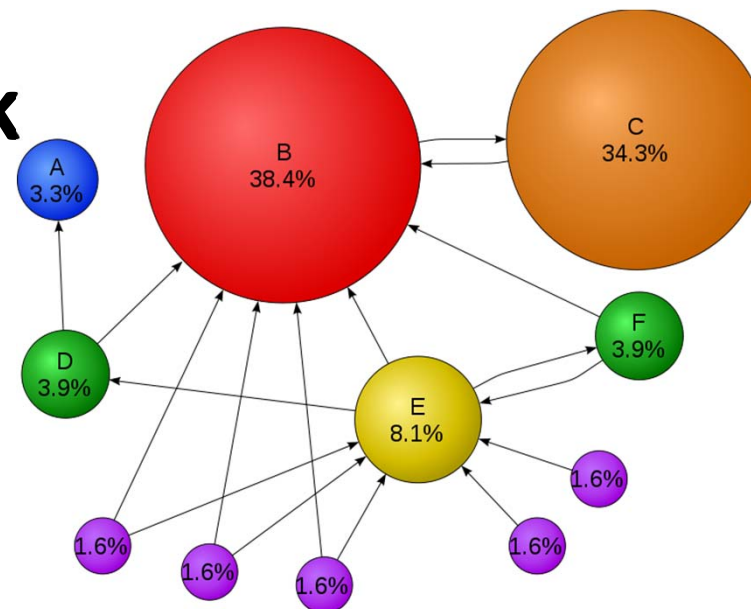
# 图算法举例：PageRank

- $R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$

- $R_v$ : 顶点v的PageRank\*N
- $L_v$ : 顶点v的出度（出边的条数）
- $B(u)$ : 顶点u的入邻居集合
- d: damping factor
- N: 总顶点个数

- 计算方法

- 初始化：所有的顶点的PageRank为1
- 迭代：用上述公式迭代直至收敛



图来源：Wikipedia



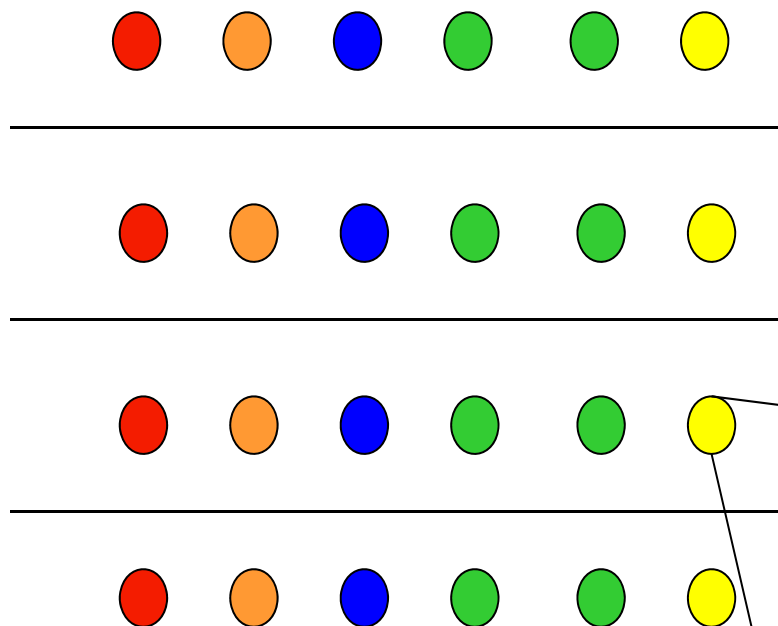
# 同步图运算系统

- “*Pregel: a system for large-scale graph processing.*”  
Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, et al.  
SIGMOD 2010.
- 开源实现: Apache Giraph, Apache Hama
- 我们的实现: GraphLite

# 图计算模型

运算  
分成  
多个  
超步

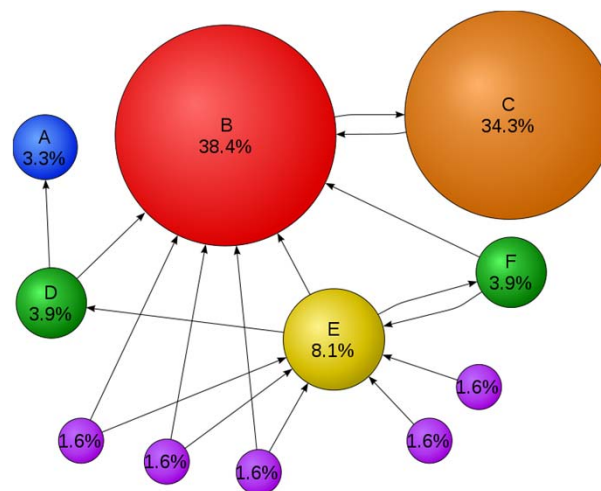
超步内，并行执行每个顶点



超步间全局同步

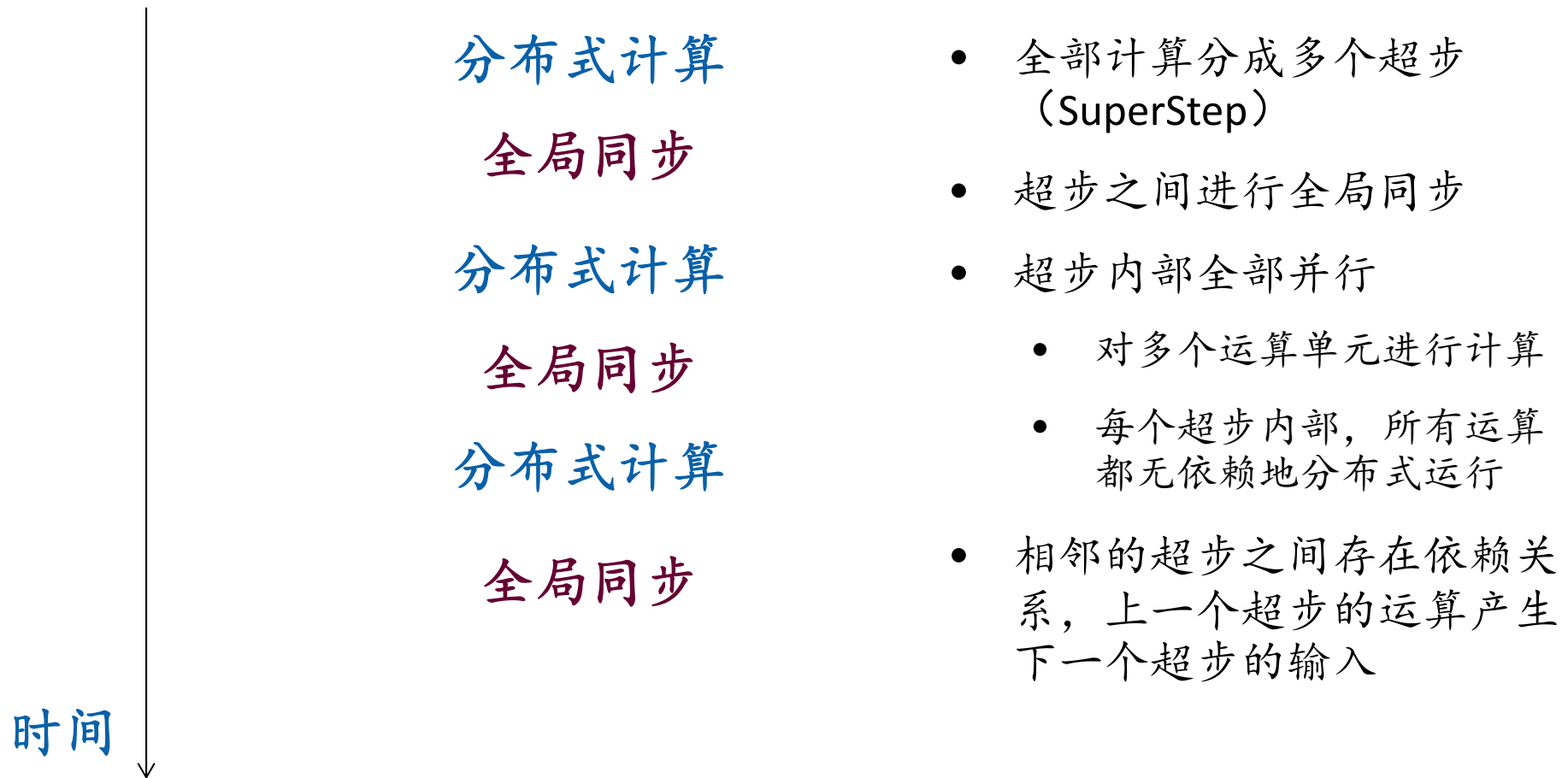
## 顶点算法通常步骤

- 1) 接收上个超步发出的 in-neighbor 的消息
- 2) 计算当前顶点的值
- 3) 向 out-neighbor 发消息

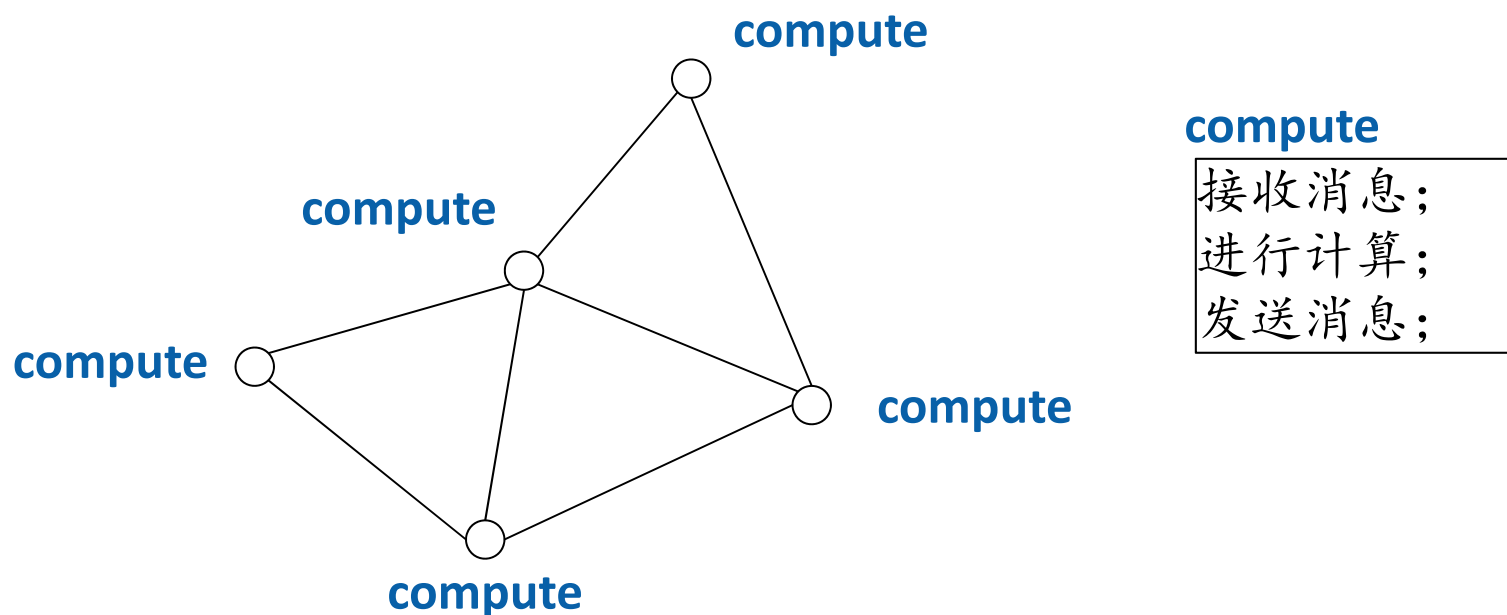


# 特点1: BSP模型

- BSP: Bulk Synchronous Processing

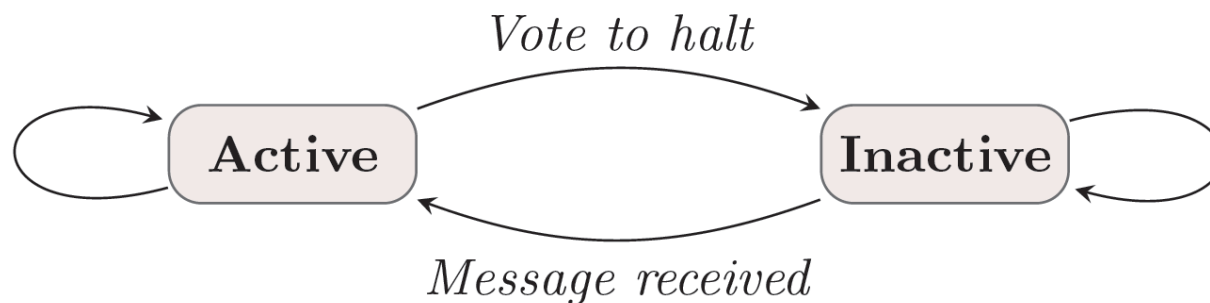


## 特点2：基于顶点的编程模型



- 每个顶点有一个value
- 顶点为中心的运算
  - 程序员可以实现一个Compute函数
  - 在每个超步中，同步图系统对每个顶点调用一次Compute
  - Compute通常接收消息，计算，然后发送消息

# 图运算如何结束？



- 顶点的两种状态

- 活跃态Active：图系统只对活跃顶点调用compute
  - 顶点初始状态都是活跃态
- 非活跃态Inactive：compute调用Vote to halt时，顶点变为非活跃态
  - 注意：非活跃的顶点也可以重新变得活跃

- 上图是顶点状态的转化图

- 当所有的顶点都处于非活跃状态时，图系统结束本次图运算

# GraphLite

- 我们下面以GraphLite为例介绍同步图编程
- GraphLite实现了Pregel论文中定义的API
- GraphLite是C/C++实现的

<https://github.com/schencoding/GraphLite>

# 图计算编程

- 数据

- 顶点?

- 边?

- 消息?

- 运算

- Compute?

# GraphLite编程

- 继承class Vertex, 实现一个子类
- 可以定义
  - 顶点值、边值、消息值的类型
  - 实现Compute函数



# Class Vertex

顶点值Type      边值Type      消息值Type



```
template<typename V, typename E, typename M>
class Vertex : public VertexBase {

public:

    void compute(MessageIterator* msgs) { ... }

    用户实现图算法

}
```

# 比照顶点算法通常步骤

```
void compute(MessageIterator* msgs) { ... }
```

当前顶点接收到的上个超步发来的所有的消息

```
for(; !msgs->done(); msgs->next()){
```

使用msgs->getValue()获得消息值

```
}
```

## 顶点算法通常步骤

- 1) 接收上个超步发出的in-neighbor的消息
- 2) 计算当前顶点的值
- 3) 向out-neighbor发消息

# 比照顶点算法通常步骤

## const V & getValue()

- V是用户自定义的顶点值类型
- 读取当前顶点值

## V \* mutableValue()

- 获得当前顶点值的地址
- 于是可以修改
  - \*mutableValue() = 赋值

### 顶点算法通常步骤

- 1) 接收上个超步发出的in-neighbor的消息
- 2) 计算当前顶点的值
- 3) 向out-neighbor发消息

# 比照顶点算法通常步骤

**OutEdgeIterator getOutEdgeIterator()**

- 得到一个iterator，可以循环访问每条出边
- 出边包含出邻ID等信息

**void sendMessageTo(const int64\_t& dest\_vertex,  
const M & msg)**

**void sendMessageToAllNeighbors(const M & msg)**

- dest\_vertex是目标顶点的ID
- M是用户自定义的消息值的类型

## 顶点算法通常步骤

- 1) 接收上个超步发出的in-neighbor的消息
- 2) 计算当前顶点的值
- 3) 向out-neighbor发消息

# GraphLite 系统提供的函数

系统提供的，可以在Compute中调用的

- `getValue()`, `mutableValue()`
  - 顶点Value的读和写
- `getOutEdgeIterator()`  
`sendMessageTo()`, `sendMessageToAllNeighbors()`
  - 访问出边，发送消息
- `superstep()`, `voteToHalt()`
  - 判断当前超步序号，将当前顶点设为Inactive状态
- `accumulate()`, `getAggregate()`等
  - 全局Aggregator

## 举例：PageRank实现

$$R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

```
class PageRankVertex: public Vertex<double, double, double>
{
    public:
        void compute(MessageIterator* msgs) { ... }
}
```

顶点值、边值和消息值的类型全为double

# 举例：PageRank实现

$$R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

```
void compute(MsgIterator * msgs)
{
    double val;
    if (superstep() == 0) {
        val = 1.0; // initial value
    }
    else {
        // compute pagerank
        double sum = 0.0;
        for (; !msgs->done(); msgs->next()) {
            sum += msgs->getValue();
        }
        val = 0.15 + 0.85 * sum;
    }
    // set new pagerank value and propagate
    *mutableValue() = val;
    int64_t n = getOutEdgeIterator().size();
    sendMessageToAllNeighbors(val / n);
}
```

# 举例：PageRank实现

$$R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

```
void compute(MsgIterator * msgs)
{
    double val;
    if (superstep() == 0) {
        val = 1.0; // initial value
    }
    else {
        // compute pagerank
        double sum = 0.0;
        for (; !msgs->done(); msgs->next()) {
            sum += msgs->getValue();
        }
        val = 0.15 + 0.85 * sum;
    }
    // set new pagerank value and propagate
    *mutableValue() = val;
    int64_t n = getOutEdgeIterator().size();
    sendMessageToAllNeighbors(val / n);
}
```

结束条件



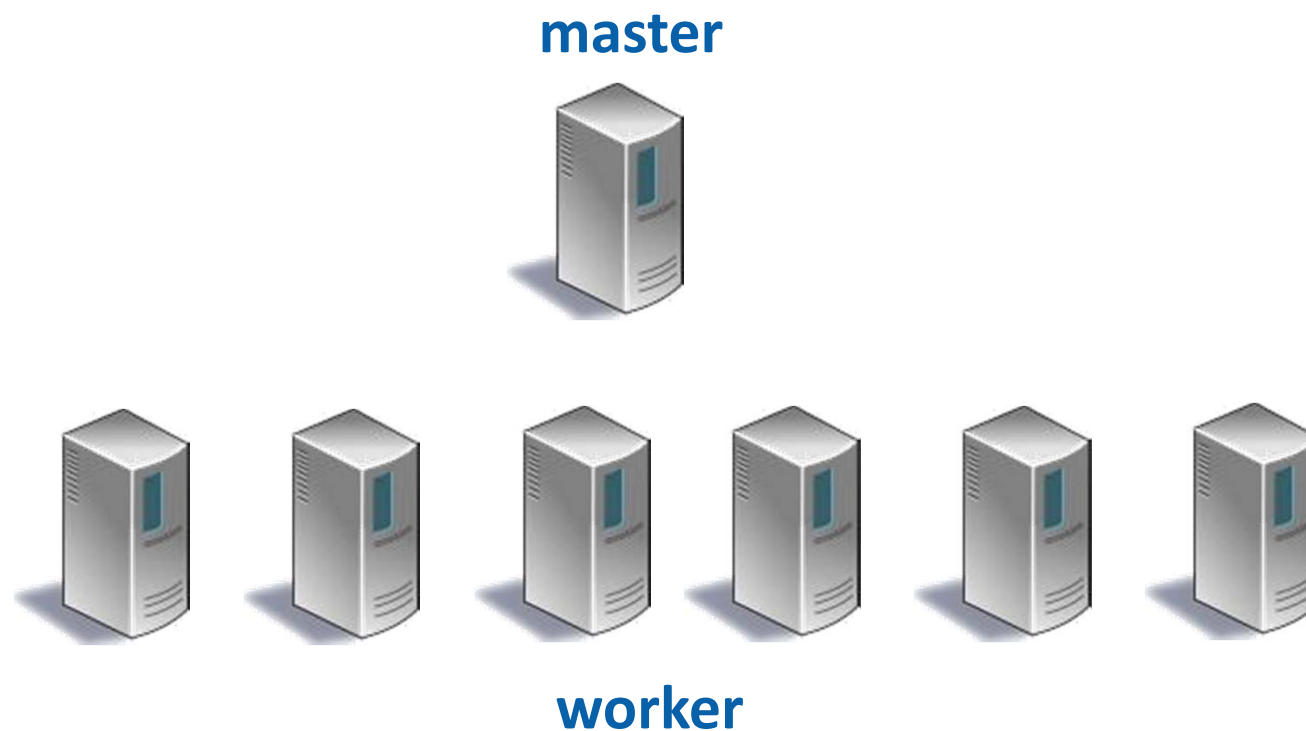
# 举例：PageRank实现

$$R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

## 结束条件

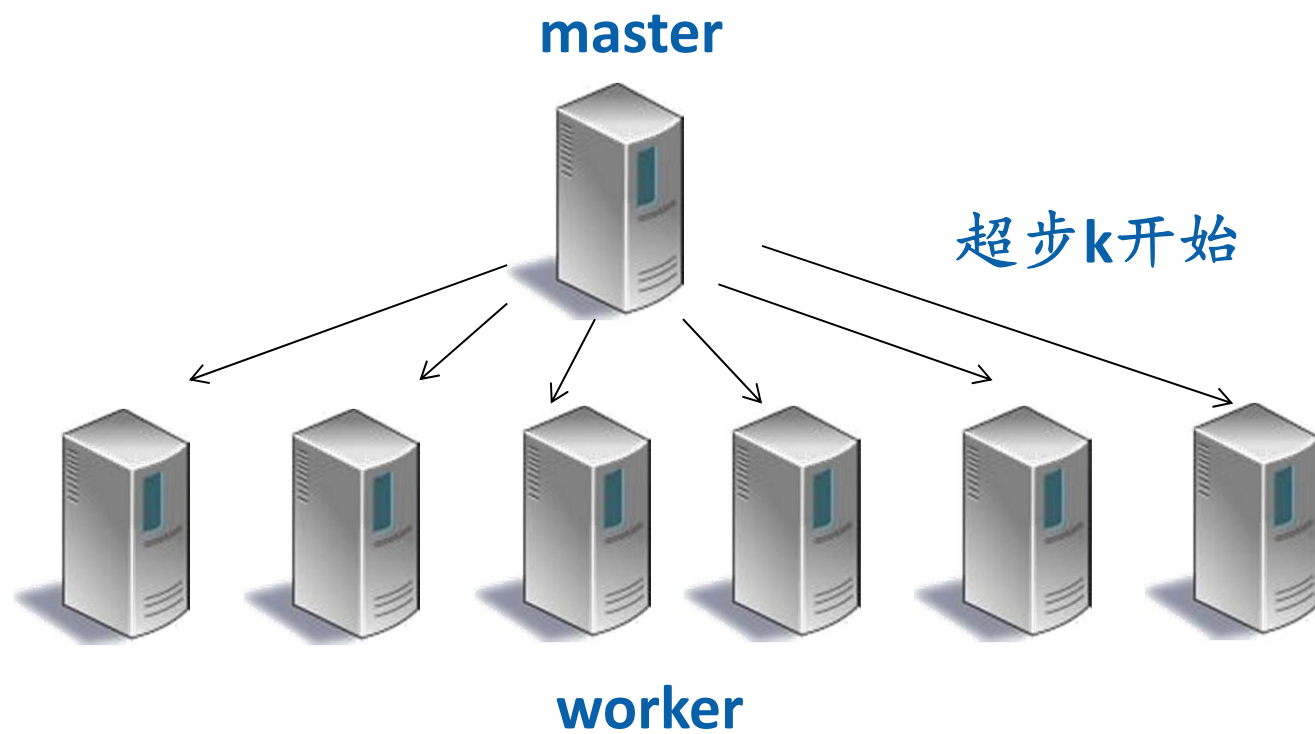
```
else {  
    // check if converged  
    if (superstep() >= 2 &&  
        *(double *)getAggregate(AGGERR) < TH) {  
        voteToHalt(); return;  
    }  
    // compute pagerank  
    double sum= 0.0;  
    for (; !msgs->done(); msgs->next()) {  
        sum += msgs->getValue();  
    }  
    val = 0.15 + 0.85 * sum;  
    // accumulate delta pageranks  
    double acc = fabs(getValue() - val);  
    accumulate(&acc, AGGERR);  
}
```

# 同步图运算系统的系统架构

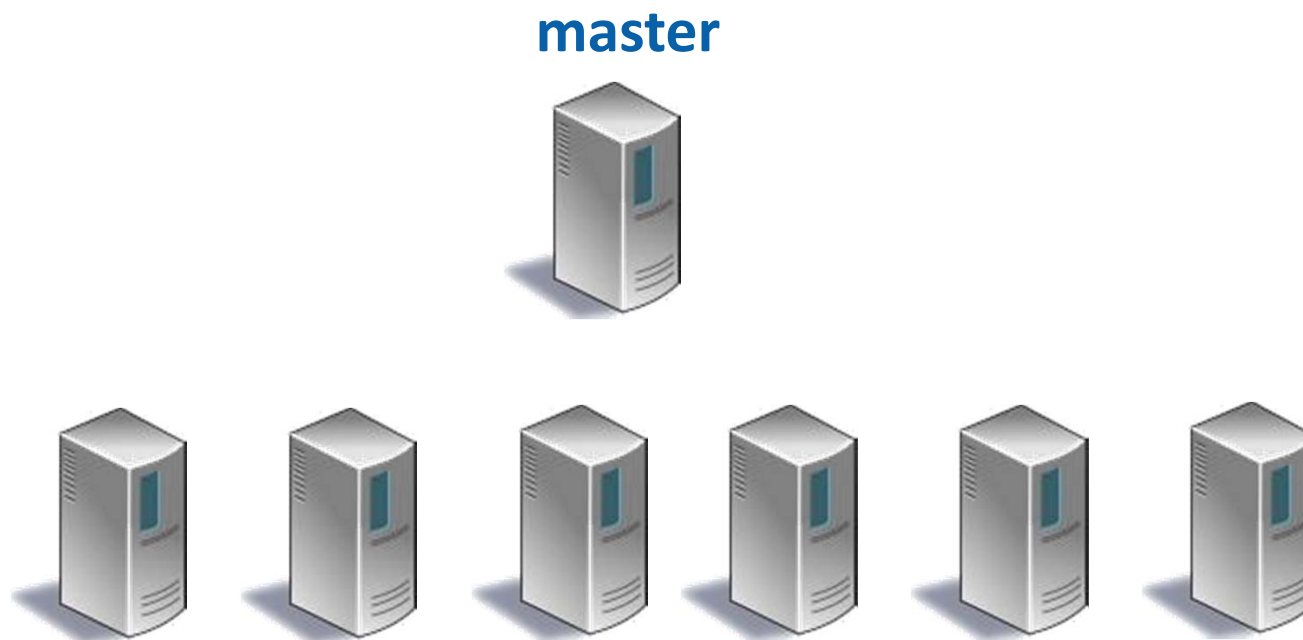


- 每个worker对应一个graph partition
- 例如：hash partition
  - $\text{Partition id} = \text{hash}(\text{vertex\_id}) \% \text{WorkerNumber}$

# 超步开始

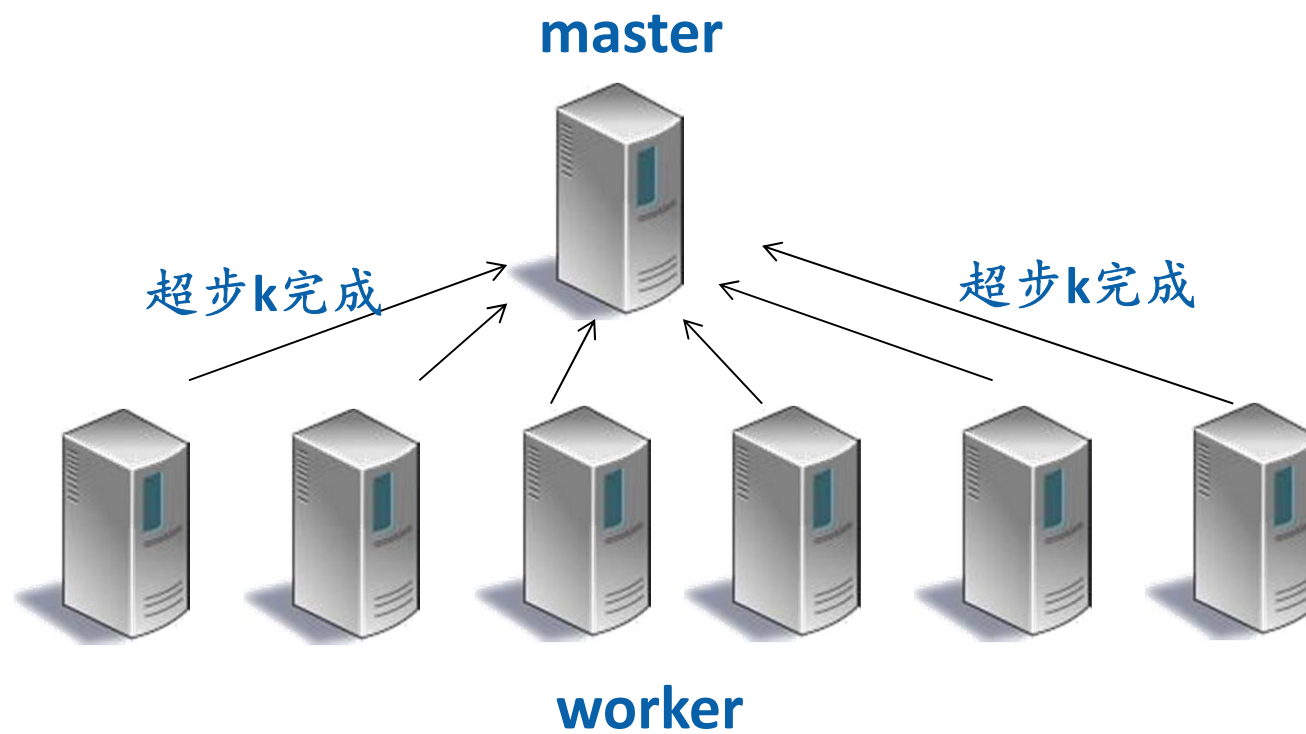


# 超步计算进行中

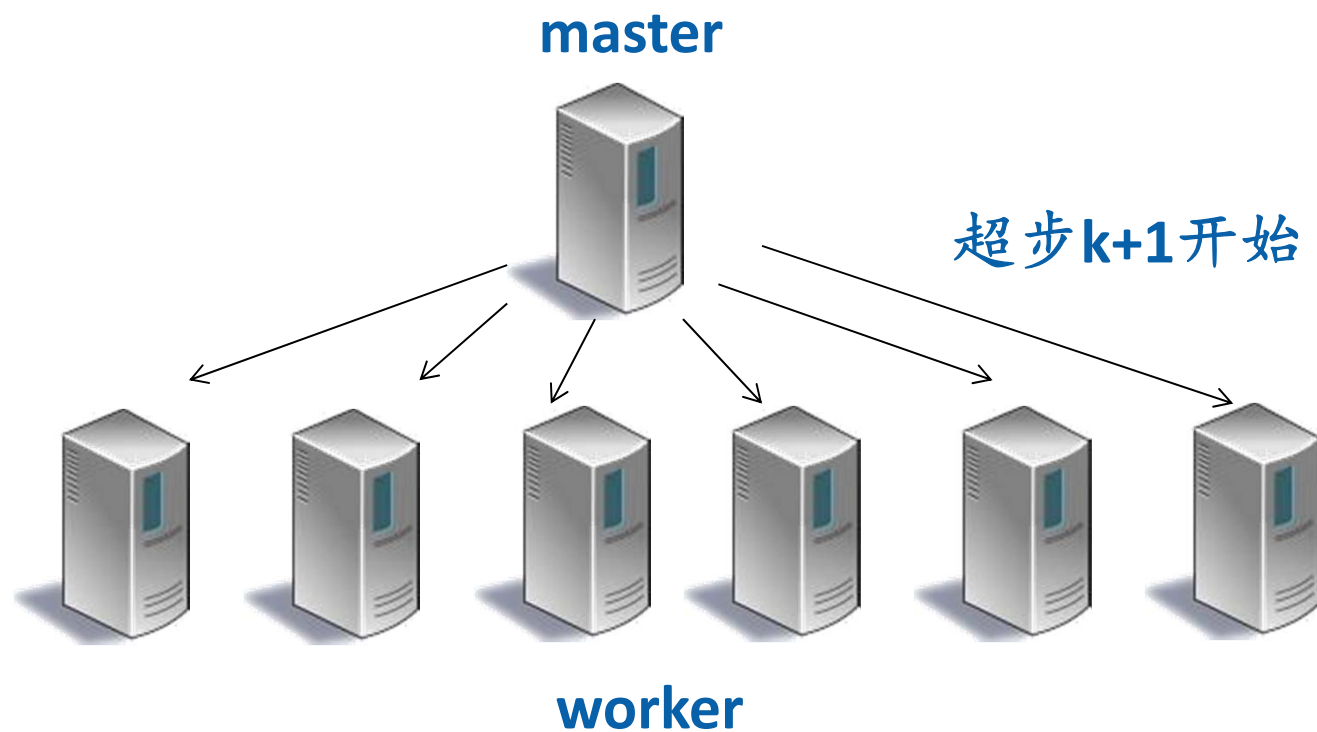


每个**worker**进行本地的计算，为本**partition**的每个顶点调用**compute**，收集顶点发送的信息，并发向对应的**worker**

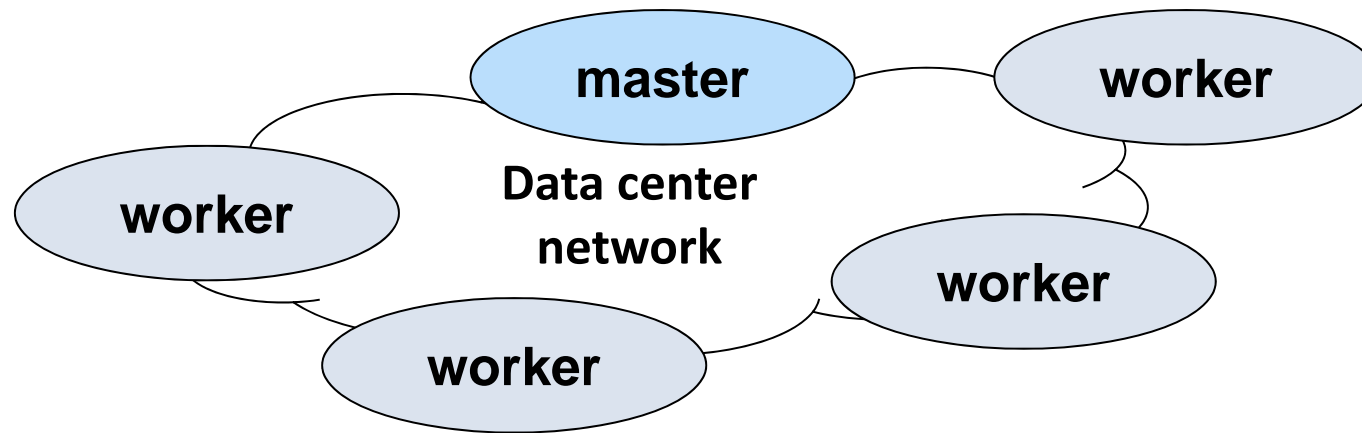
# 超步结束



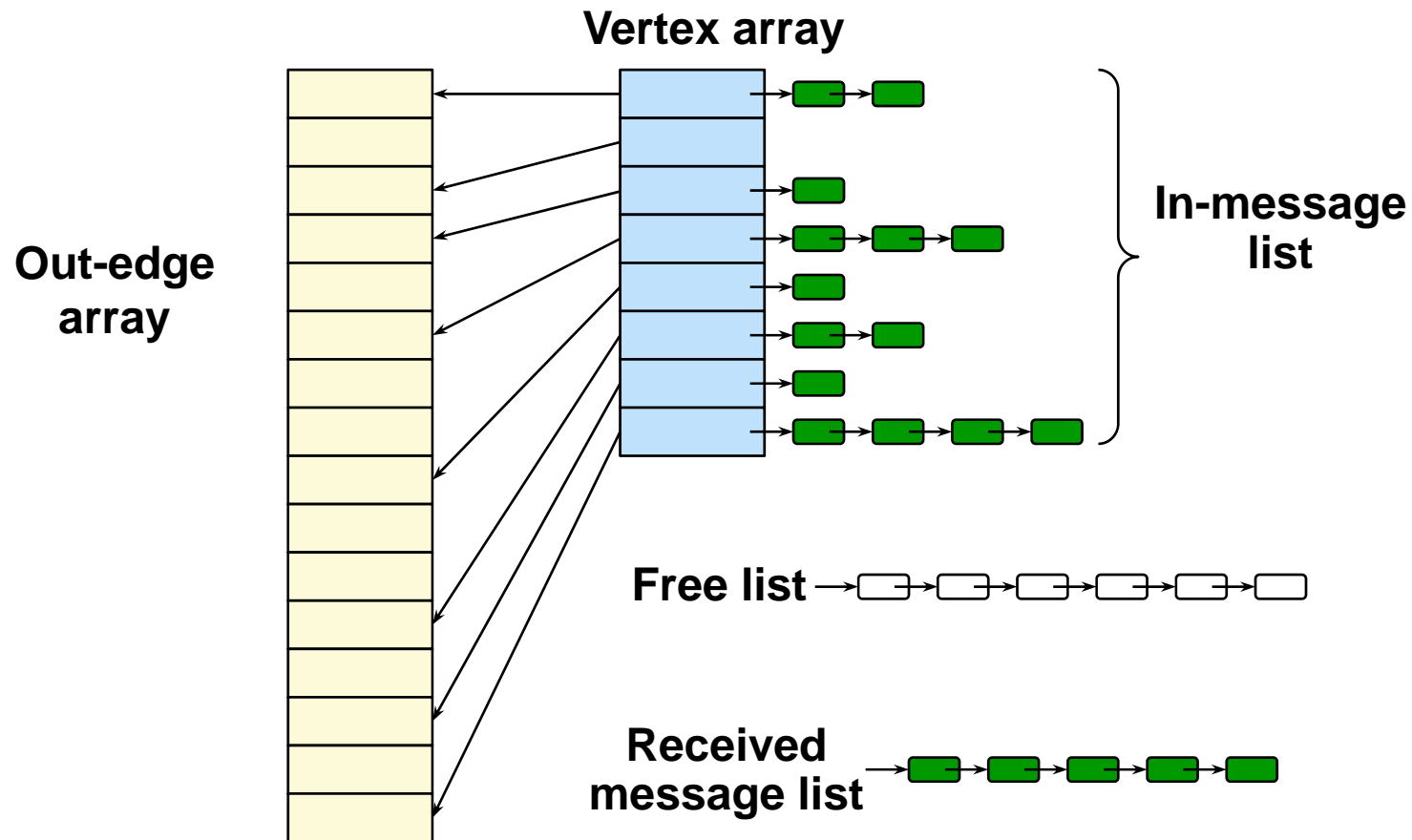
# 超步开始



# GraphLite



# GraphLite Worker

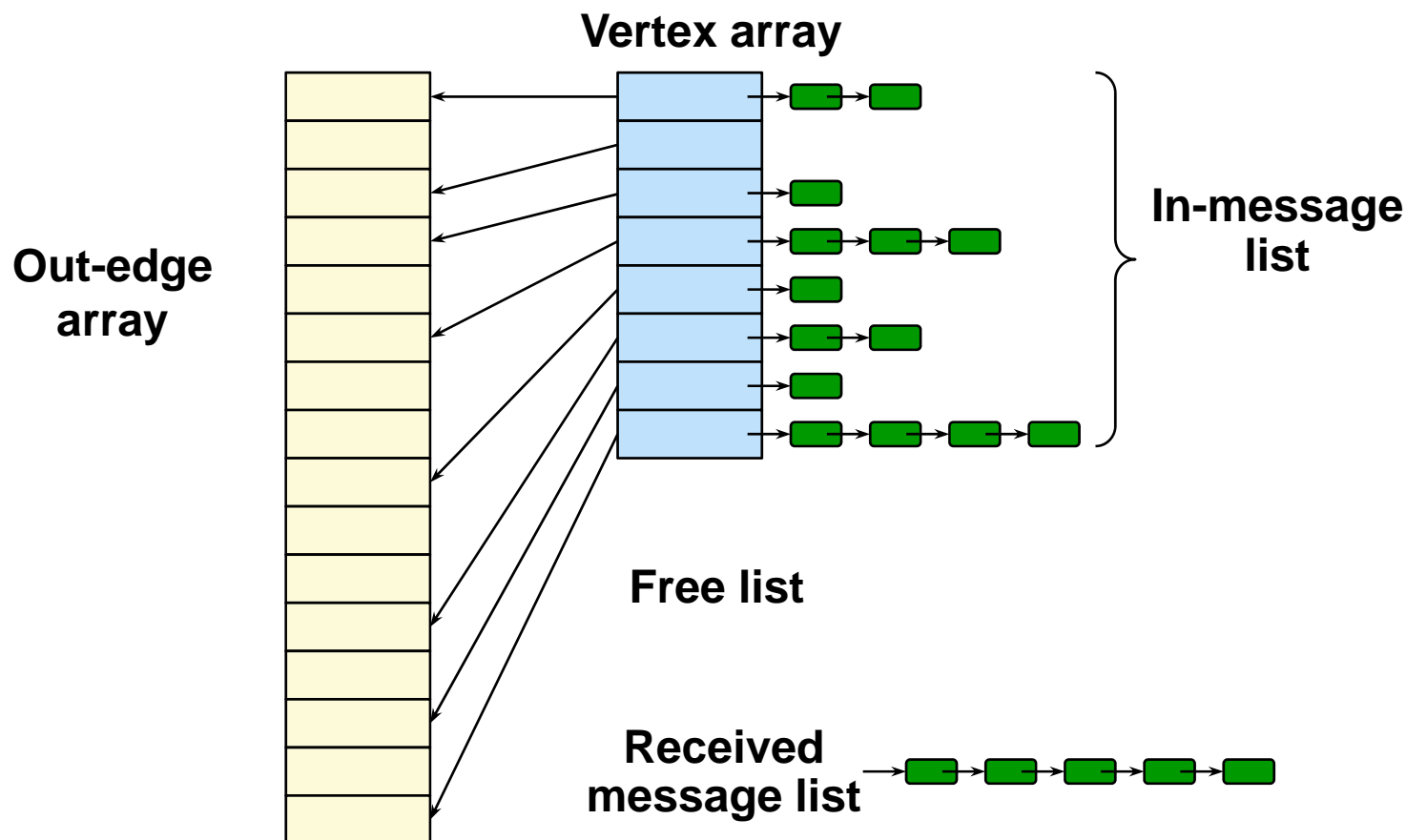


**Message: (source ID, target ID, message value, ptr)**



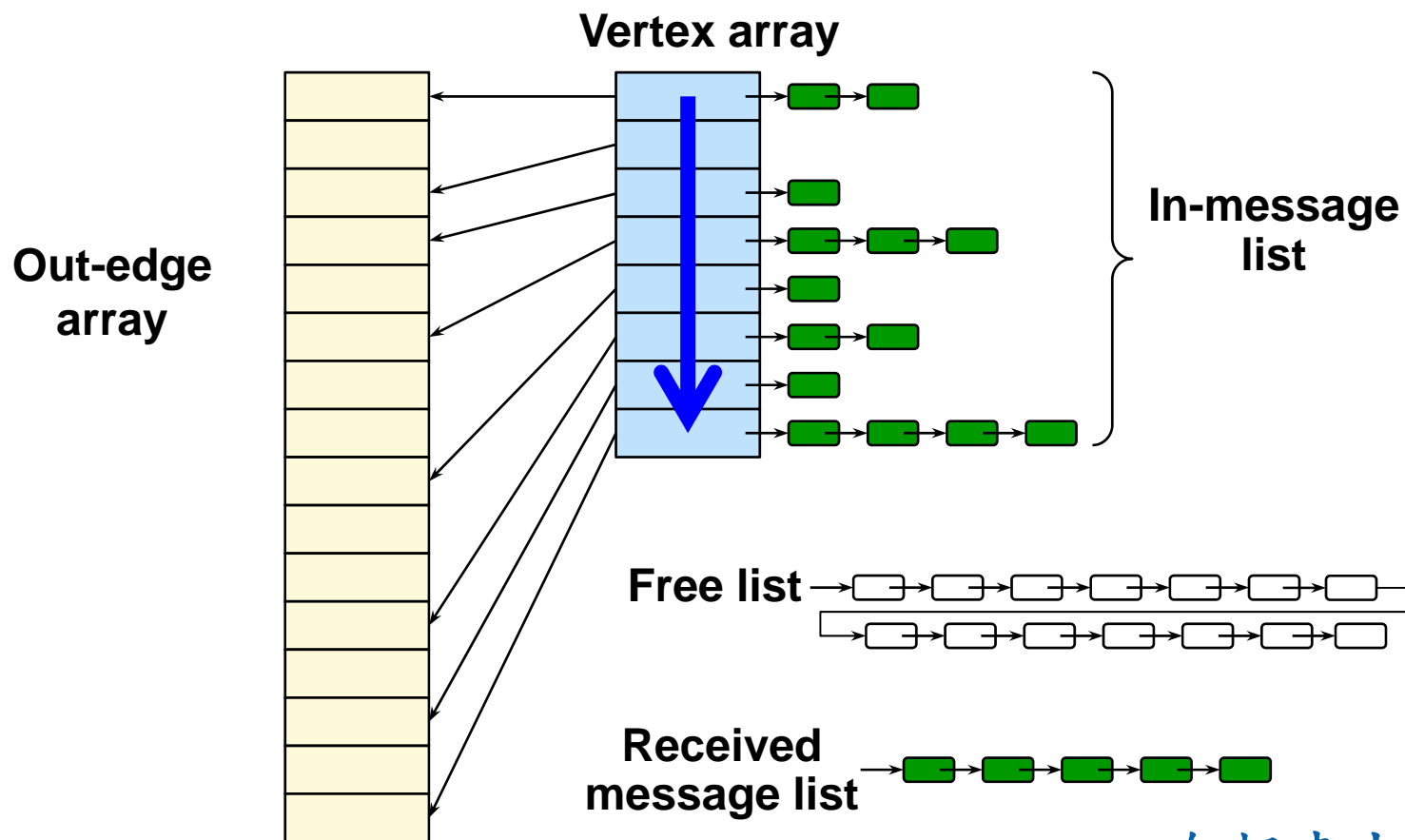
# 超步开始：分发message

把Received message list  
中的消息放入接收顶点的  
in-message list



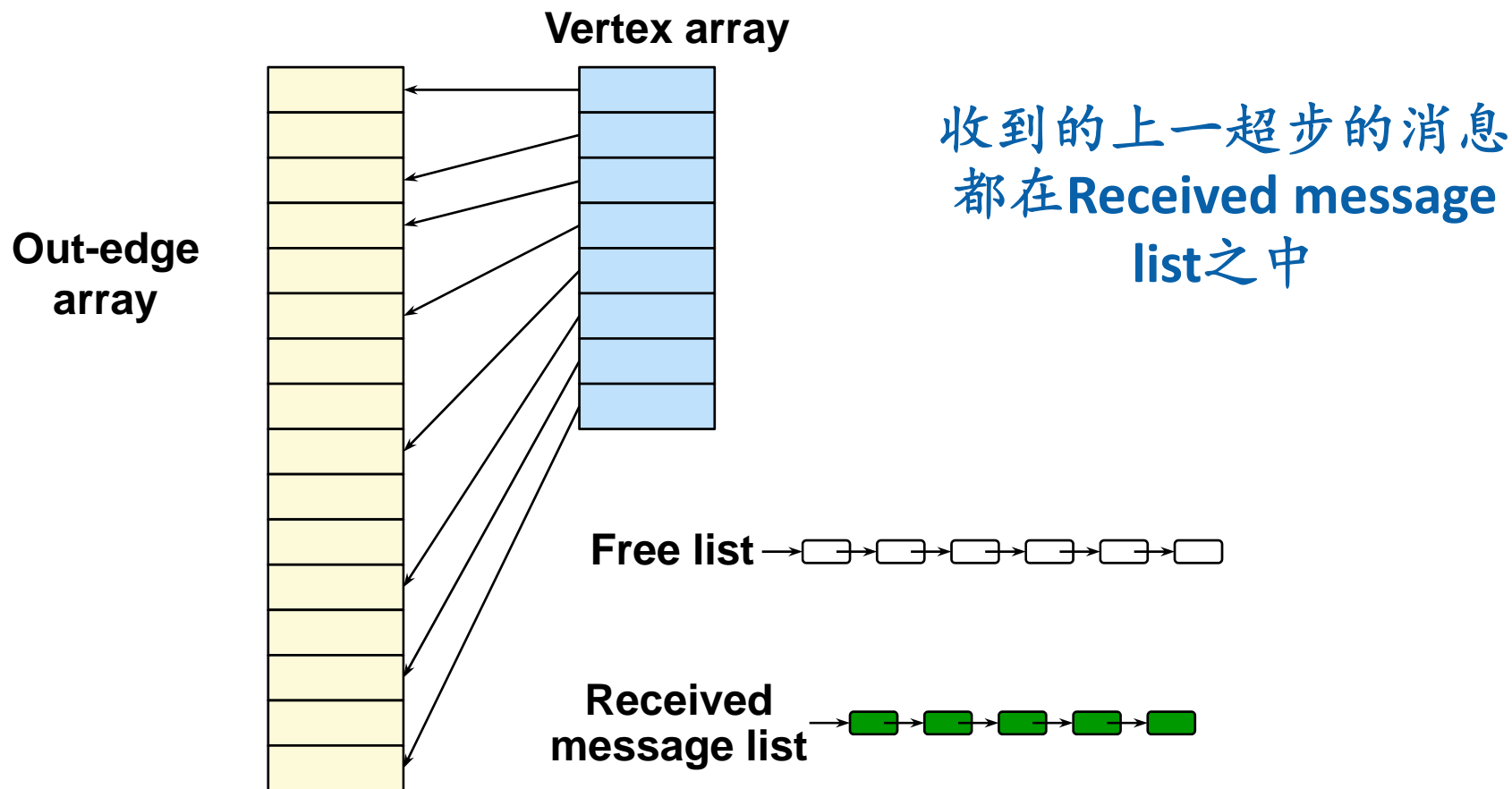
Message: (source ID, target ID, message value, ptr)

# 超步计算中：依次访问Vertex, 调用Compute



在超步中可能收到消息

# 超步结束时



Message: (source ID, target ID, message value, ptr)

# Aggregator全局统计量

- 每个超步内

- 每个Worker分别进行本地的统计: `accumulate()`

- 超步间，全局同步时

- Worker把本地的统计值发给master
  - Master进行汇总，计算全局的统计结果
  - Master把全局的统计结果发给每个Worker

- 下一个超步内

- Worker从Master处得到了上个超步的全局统计结果
    - Compute就可以访问上一超步的全局统计信息了
    - `getAggregate()`
  - 继续计算本超步的本地统计量

# 同步图运算系统小结

- 运算在内存中完成
- 基于BSP模型实现同步图运算
- 基于顶点的编程模型
- 容错依靠定期地把图状态写入硬盘生成检查点
  - 在一个超步开始时，master可以要求所有的worker都进行检查点操作

# 小结

- MapReduce/Hadoop

- 编程模型
- 系统实现
- 典型算法

- Microsoft Dryad

- 同步图计算系统

- 图算法
- 同步图计算
- 图计算编程
- 系统实现