分布式编程环境MapReduce

内容

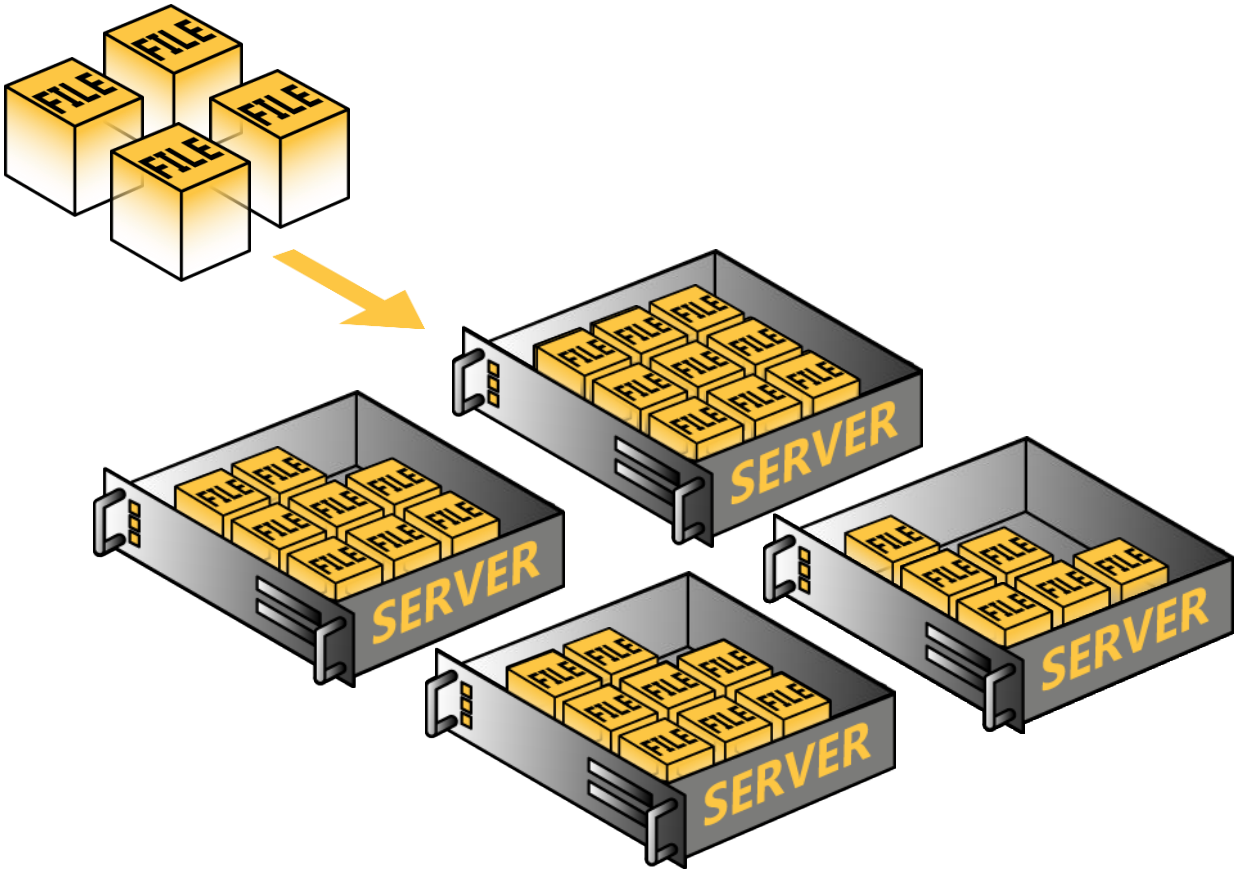
Hadoop架构示意

MapReduce原理

Hadoop实现

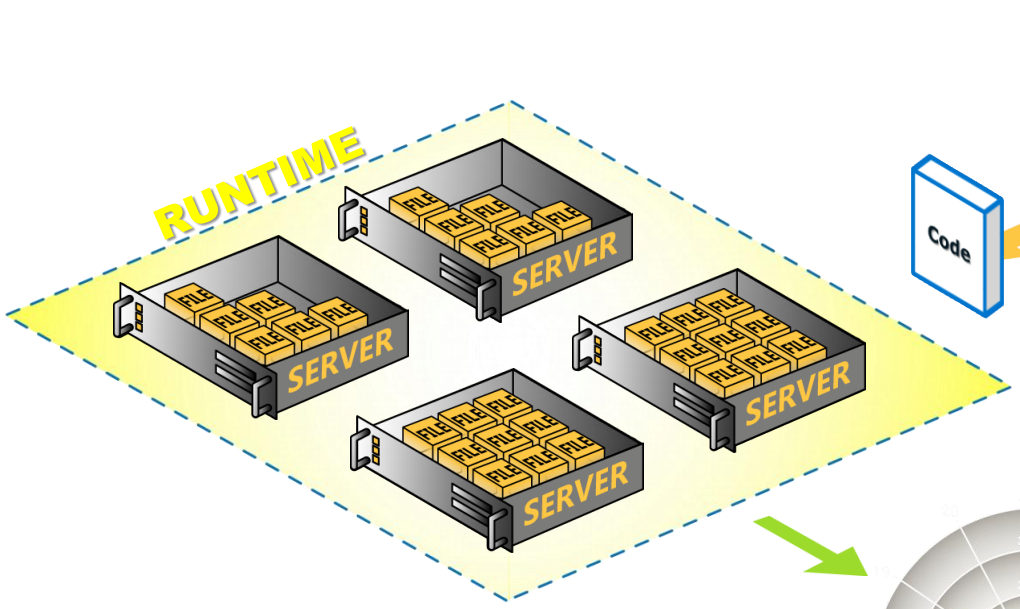
MapReduce程序调优

Hadoop示意
FIRST, STORE THE DATA



Hadoop示意

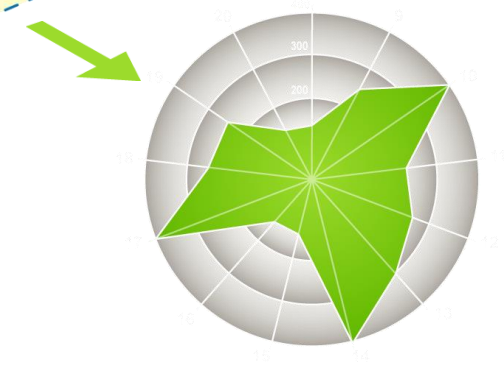
SECOND, TAKE THE PROCESSING TO THE DATA



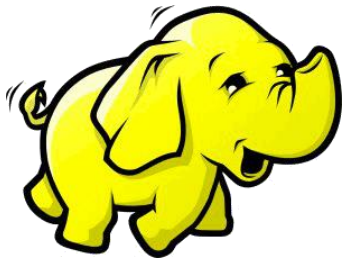

```
// Map Reduce function in
JavaScript

var map = function (key,
value, context) {
var words =
value.split(/[^a-zA-Z]/);
for (var i = 0; i <
words.length; i++) {
                    if
(words[i] !== "")
{context.write(words[i].to
LowerCase(), 1);}
}};

var reduce = function
(key, values, context) {
var sum = 0;
while (values.hasNext()) {
sum +=
parseInt(values.next());
                    }
context.write(key, sum);
};
```

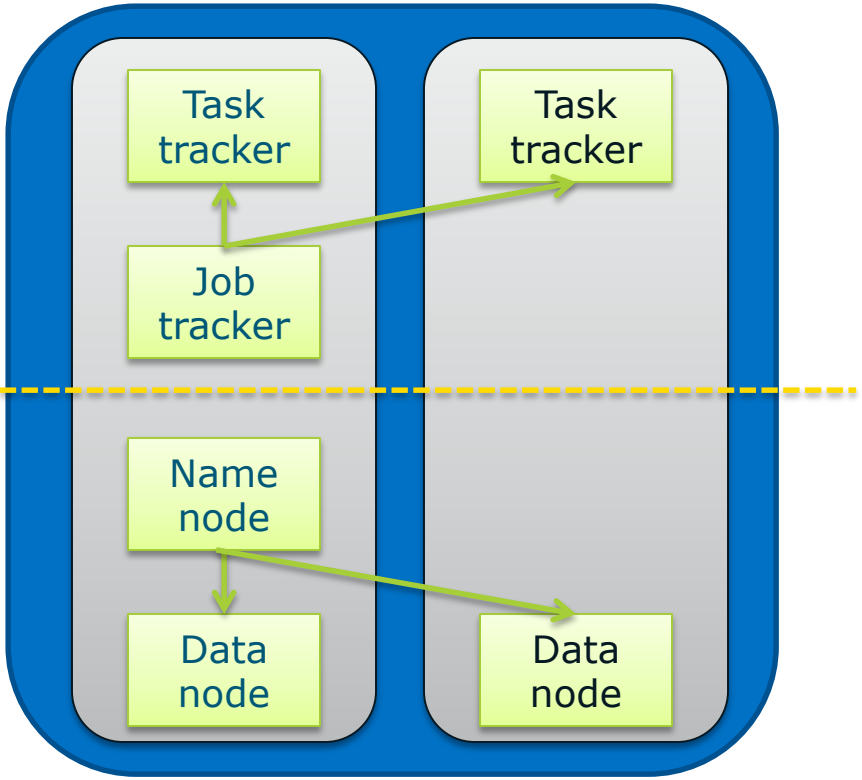


Hadoop 架构



Map Reduce Layer

HDFS Layer



Reference:
http://en.wikipedia.org/wiki/File:Hadoop_1.png

MapReduce编程框架

高效处理大量的数据

- TB级的数据很常见

将任务分布到集群中所有节点上，Share Nothing

让计算靠近数据

- 尽可能的在数据所在节点上进行相应计算

提供简明的编程接口

- 2个用户控制的阶段
  - Map
  - Reduce
- 两者之间是框架控制的Shuffle和Sort

MapReduce程序的特征

自动并行化以及分布式

错误容忍

提供状态以及监控工具

对于程序员提供清晰的编程接口

- 可使用Java实现
- 也可用Hadoop Streaming用任何脚本语言实现

隐藏了大量的实现细节

- 程序员只要专注于用map和reduce函数实现应用逻辑

MapReduce程序的编程模型

只需要编写两个函数即可，分别是Map函数以及Reduce函数

- `void map(MK key, MV value,`
  `Context context)`

- `void reduce(RK key,`
  `Iterable<RV> values,`
  `Context context)`

MapReduce的例子—WordCount

WordCount是MapReduce的HelloWorld程序

程序的目的是统计大量文档中每个单词出现的次数

程序具有实际的意义，即可以统计单词在文档集合中出现的频率，这是进行词语重要程度的基本统计

请想一下一个串行的程序如何完成这样的工作

- 读取每一个文件
- 单词拆分
- 使用哈希表进行统计
- 如果文件太大怎么办？

运行Word Count

运行hadoop自带的wordcount程序：

hadoop jar /usr/lib/hadoop/hadoop-examples.jar wordcount /user/train/test/overview.txt /user/train/output

MapReduce下的WordCount程序

```
public void map(Object key, Text value, Context context) {
    Text word = new Text();
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens())
        context.write(new Text(itr.nextToken()),
                      new IntWritable(1));
 }

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

举例说明MapReduce程序的执行过程
下面是数据集

File 1: the weather is good

File 2: today is good

File 3: good weather is good.

每一个Map函数的输出

- Worker 1:
  - (the 1), (weather 1), (is 1), (good 1).
- Worker 2:
  - (today 1), (is 1), (good 1).
- Worker 3:
  - (good 1), (weather 1), (is 1), (good 1).

思考：Map任务的输出到什么地方去了？

每一个Reduce函数的输入

- Worker 1:
  - (the 1)
- Worker 2:
  - (is 1), (is 1), (is 1)
- Worker 3:
  - (weather 1), (weather 1)
- Worker 4:
  - (today 1)
- Worker 5:
  - (good 1), (good 1), (good 1), (good 1)

思考：在Map函数执行完毕，Reduce函数开始之前发生了什么？

程序的最终结果

- Worker 1:
  - (the 1)
- Worker 2:
  - (is 3)
- Worker 3:
  - (weather 2)
- Worker 4:
  - (today 1)
- Worker 5:
  - (good 4)

再次思考一下这个程序的执行过程，并对照上面的源代码进行程序执行过程的模拟

MapReduce的基本概念

一个作业(Job)被划分成很多个map任务(mapper)和reduce任务(reducer)

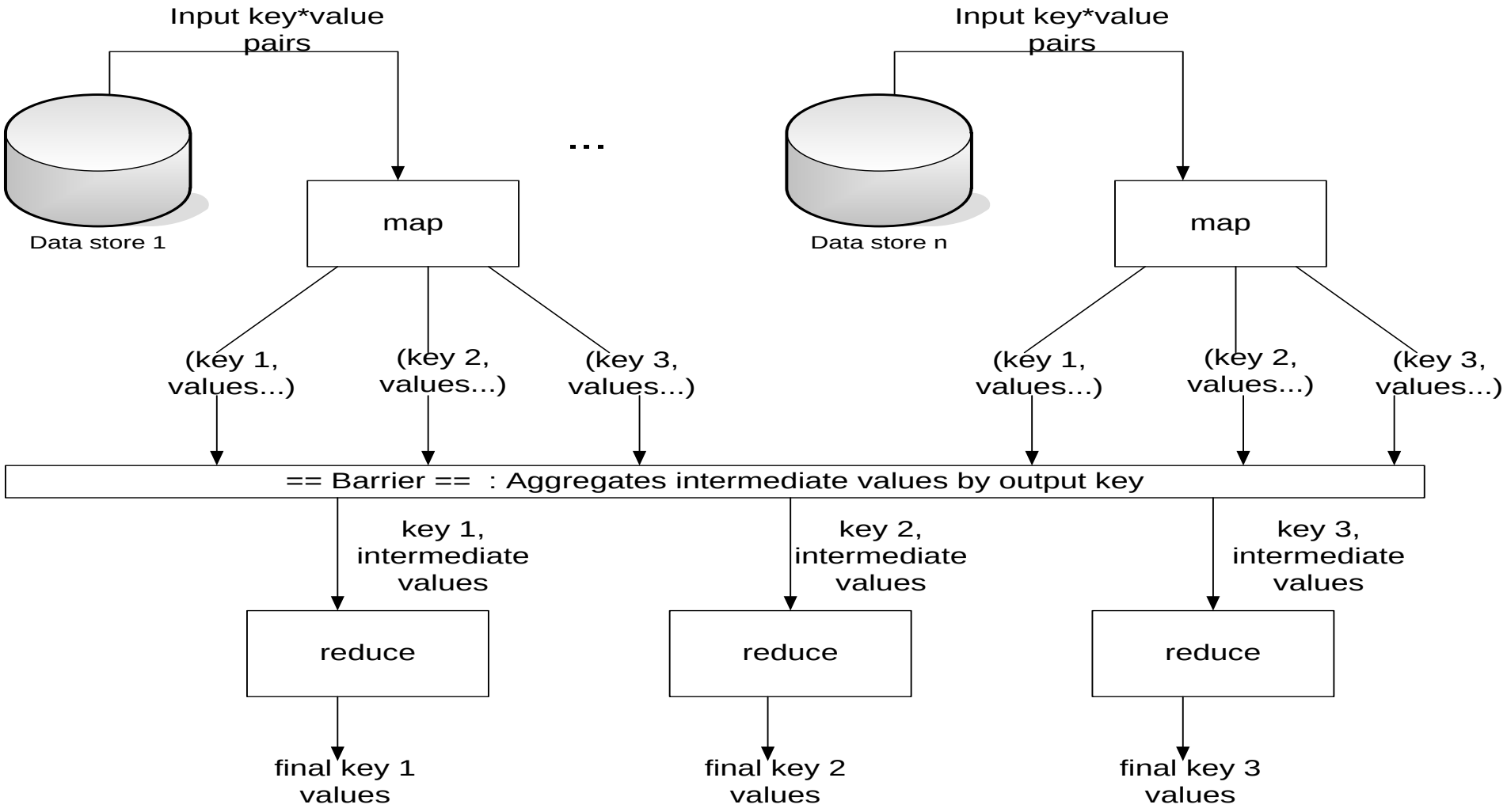
Mapper

- Job的收入数据被切分成很多片（split），每个split对应一个Mapper。所以mapper数量由输入决定的
- Mapper内部循环调用map函数，每次处理一**条**记录（key-value对），可生成一条或者多条中间记录

Reducer

- 用户指定Reducer数量
- Reducer内部循环调用Reduce函数，每次处理一**组**记录（从mapper来）
  - 分组的依据是有相同的key。框架保证相同key的记录必然由同一个reducer处理

MapReduce系统的系统结构



reduce阶段只有在所有的map阶段完全完成之后才能够开始。为什么？

Partitioner

MapReducer框架如何知道应该将Map函数的输出交由哪个Reduce函数来执行呢？

答案是：Partitioner。Partitioner将每个Map函数的输出定位到某个Reduce任务

缺省的Partitioner是HashPartitioner。可以定制Partitioner来实现自己的划分策略

每个Map任务的所有输出会被划分成若干分区（partition），每个分区对应一个Reduce任务。分区内部是排序的

Shuffle

Shuffle就是Mapper的输出传给Reducer的过程

MapReduce框架确保Reducer的输入都是按照键值排序的

Shuffle分为：

- Map端shuffle
- Reduce端shuffle

Shuffle: Map端

每个Map任务有块内存（io.sort.mb）存放map函数的输出

- Map任务的输出可能会大于这块内存；如放不下，则将其写到（spill）本地磁盘上。
- 在写磁盘前，先分区，然后分区内部排序

最后，在map任务结束前，将内存和所有的spill文件合并成一个已分区且排序的文件作为map的输出

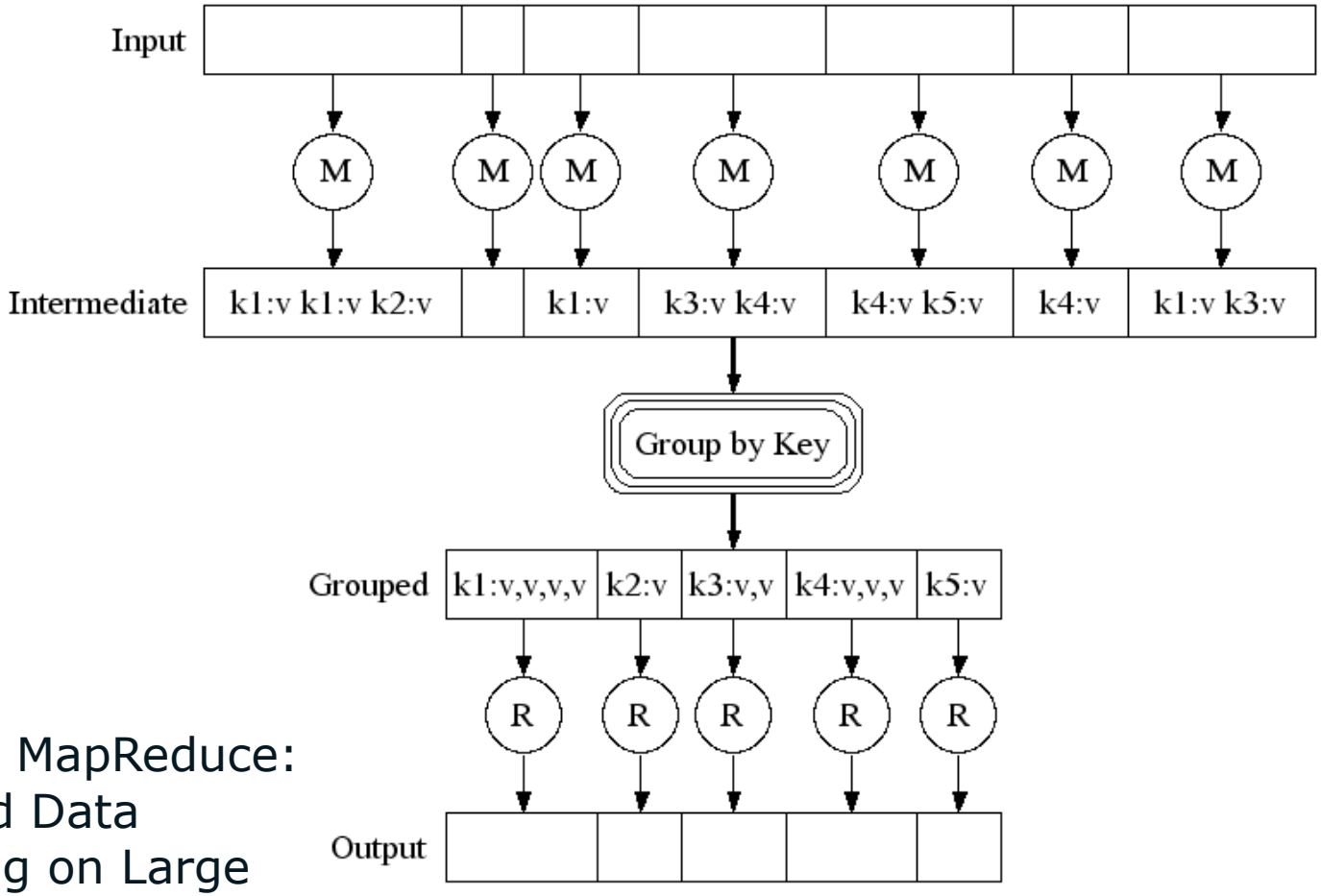
Shuffle: Reduce端

Reduce任务会在所有的Map任务完成前就启动，进入复制阶段（copy phase）

- 获取heartbeat中传过来的已经完成的Map任务列表
- 从map任务那里通过HTTP并行的取得属于这个reducer的map输出
- 合并不同map输出成一个有序的文件

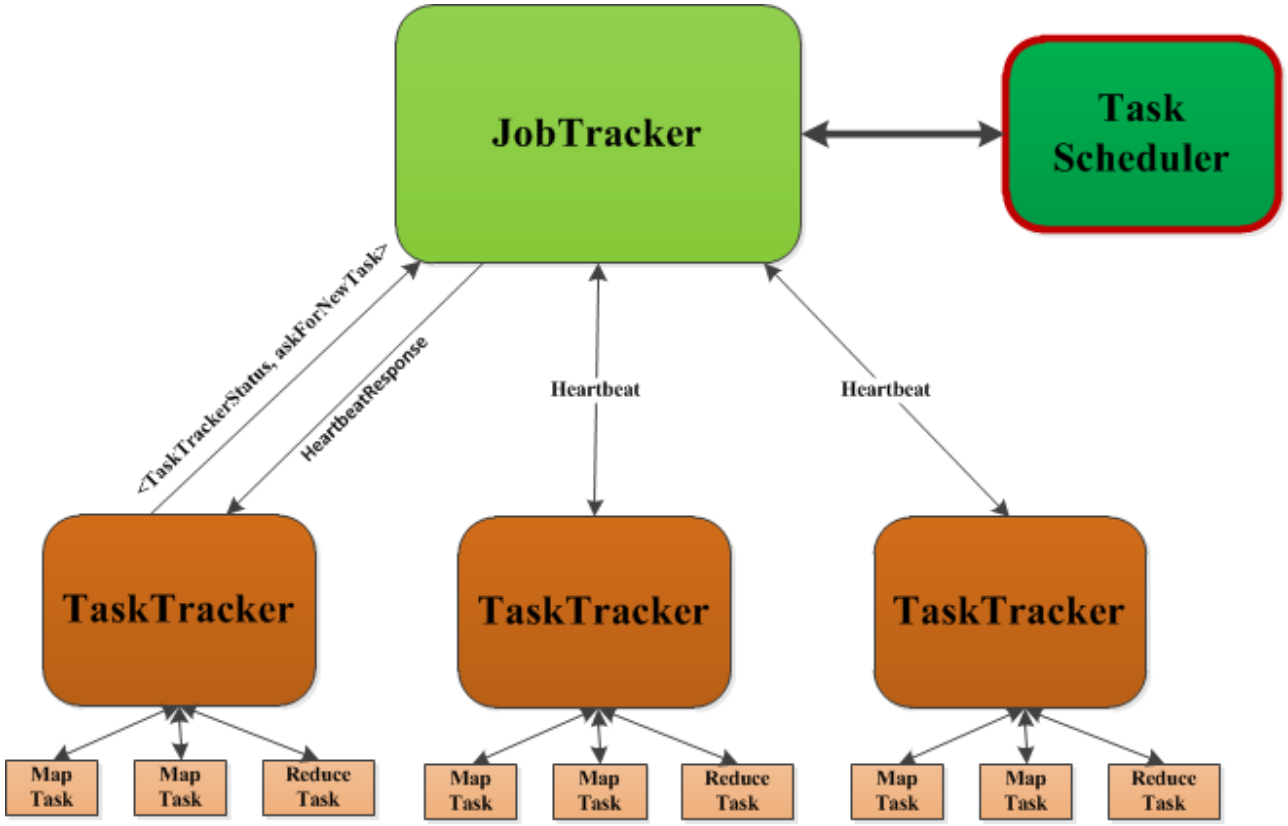
当所有的map输出都被合并成一个有序文件后，reduce函数启动

MapReduce程序的运行过程



参见论文：MapReduce: Simplified Data Processing on Large Clusters， OSDI2004

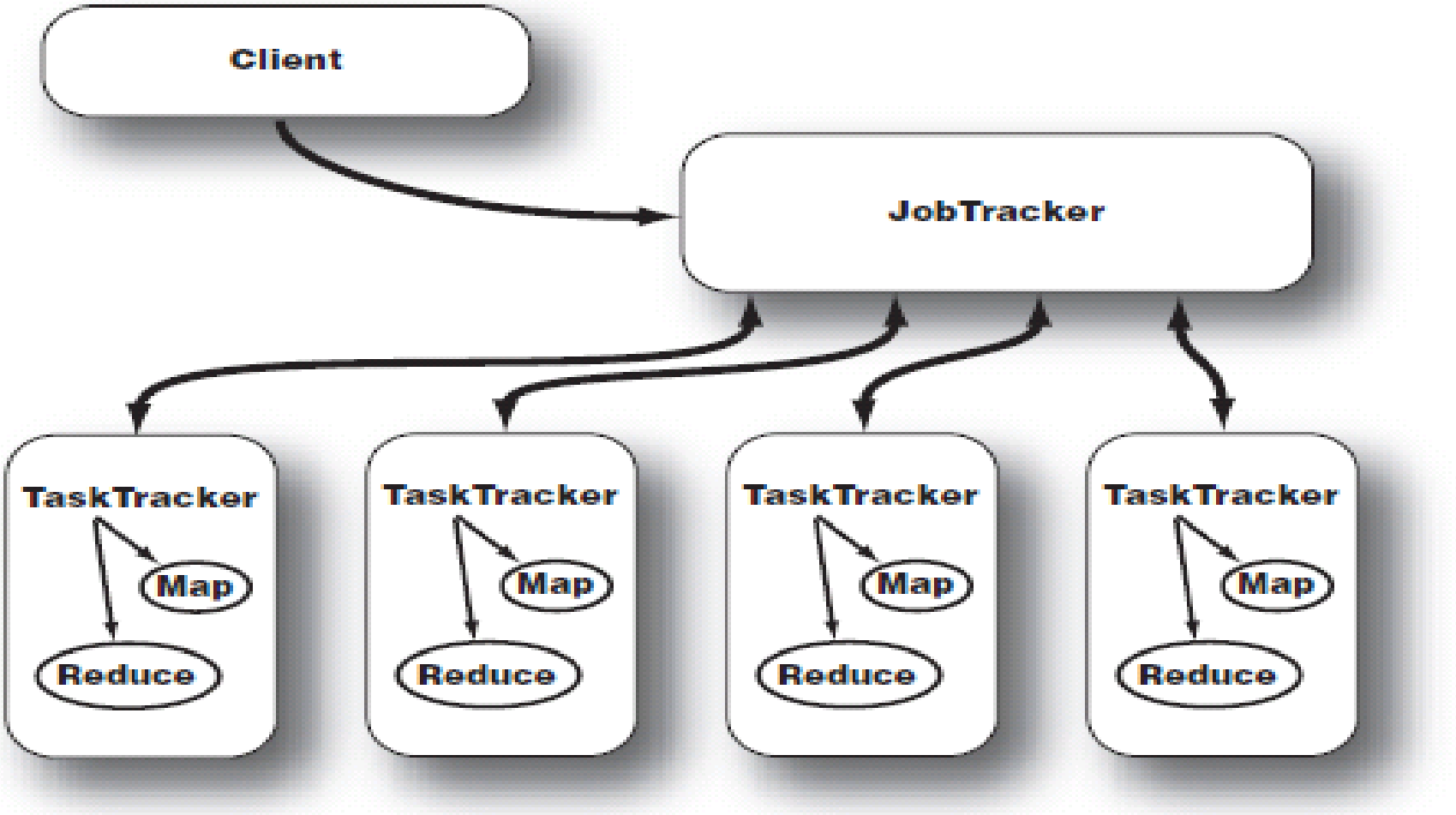
M/R的任务调度


JobTracker
Task Scheduler
<TaskTrackerStatus, askForNewTask>
HeartbeatResponse
Heartbeat
Heartbeat
TaskTracker
TaskTracker
TaskTracker
Map Task
Map Task
Reduce Task
Map Task
Map Task
Reduce Task
Map Task
Map Task
Reduce Task

M/R的任务调度

- **heartbeat**
  - TaskTracker周期性（默认为3s）调用RPC向JobTracker汇报信息，形成heartbeat
  - 汇报信息包括TaskTracker状态信息、Task运状况等
- **Slot**
  - 资源划分单位
  - 分为map slot和reduce slot两种
  - 由参数mapred.tasktracker.[map|reduce].tasks.maximum设置

Hadoop实现


Client
JobTracker
TaskTracker
Map
Reduce
TaskTracker
Map
Reduce
TaskTracker
Map
Reduce
TaskTracker
Map
Reduce

Hadoop实现（2）

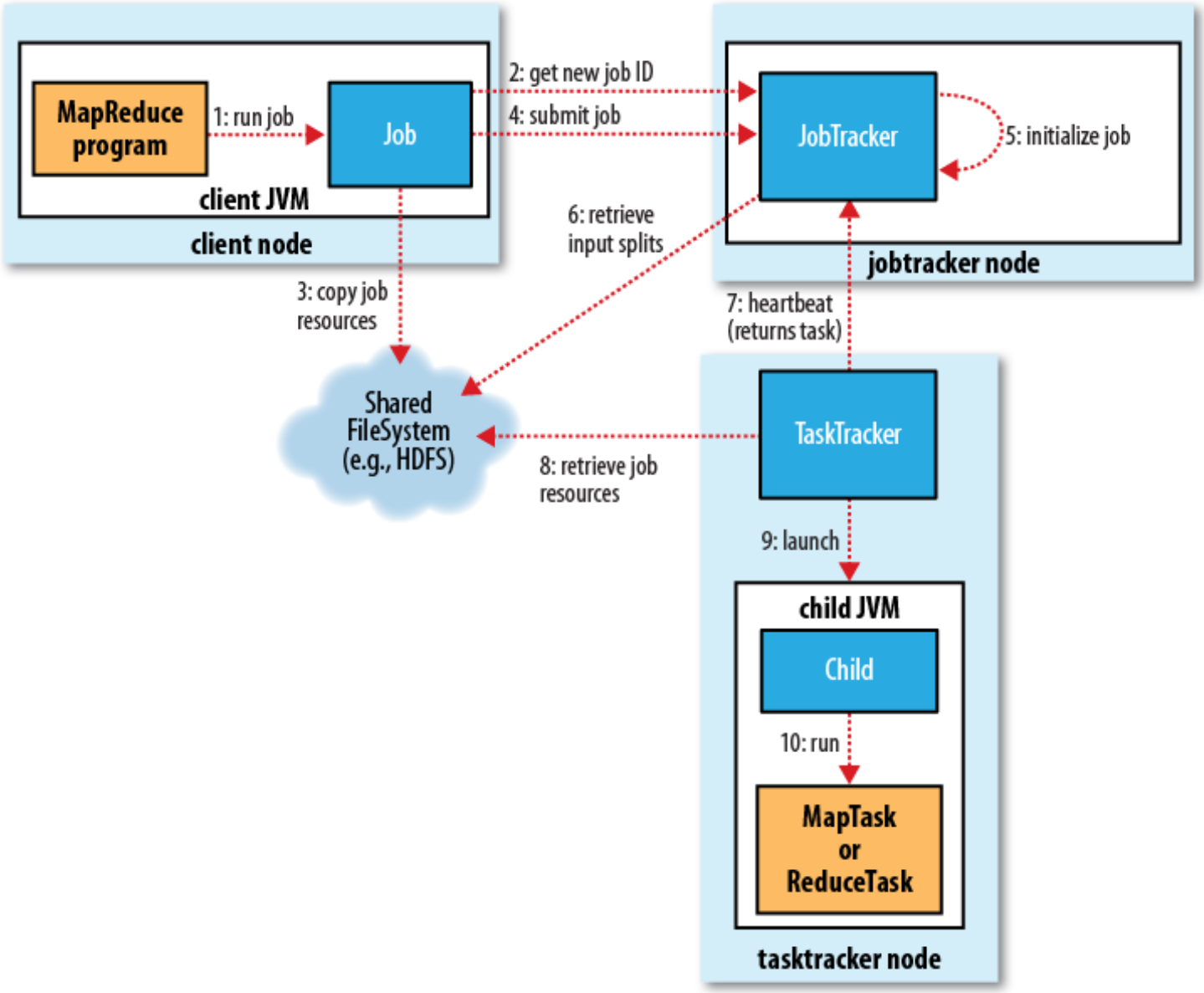
JobTracker: 管理MapReduce集群

TaskTracker: 管理单个节点上的资源和任务执行

每个节点上静态设置若干个Map Slot和Reduce Slot

- map任务或reduce任务只有被分配到某个slot才能执行
- Map slot和reduce slot的比例要看具体应用，一般2：1
- Slot的内存大小和参数静态可配置
- Slot的总数取决于节点能力
  - 内存、CPU、网络

Hadoop MapReduce的执行流程



Input

InputFormat

- 定义了数据在哪里，如何分片(InputSplit)，如何解析(RecordReader)

输入文件一般保存在HDFS中

- 可以是文本文件，也可以是其它形式的文件
- 文件经常很大，甚至有几十个GB

FileInputFormat

- 所有基于文件的InputFormat都从这个类继承
- 当启动MapReduce Job时，FileInputFormat会从用户指定的一个目录中读取所有文件。然后将这些文件分割为一个或者多个InputSplit（缺省基于数据块大小）

预定义的文件输入格式

InputFormat	描述	Key	Value
TextInputFormat	缺省；从文本文件中读取每行	该行的偏移	行内容
KeyValueTextInputFormat	将每行分割成键值对	第一个tab符之前的内容	该行剩余内容
SequenceFileInputFormat	从SequenceFile读取	SequenceFile的键	SequenceFile的值
NLineInputFormat	从文本文件中读取每行，split份数由用户指定	该行的偏移	行内容

OutputFormat

OutputFormat定义怎么保存MapReduce的输出

JobConf.setOutputFormat()

写入到HDFS的所有OutputFormat都继承自FileOutputFormat

每一个Reducer都写一个文件到一个共同的输出目录，文件名是part-nnnnn，其中nnnnn是与reducer相关的一个号

FileOutputFormat.setOutputPath()

Output Format

OutputFormat	描述
TextOutputFormat	缺省；将每行以”键\t值”的形式写text文件
SequenceFileOutputFormat	将每个键值写入SequenceFile
NullOutputFormat	丢弃输出

任务调度器（Scheduler）

FIFO

- 不支持抢占

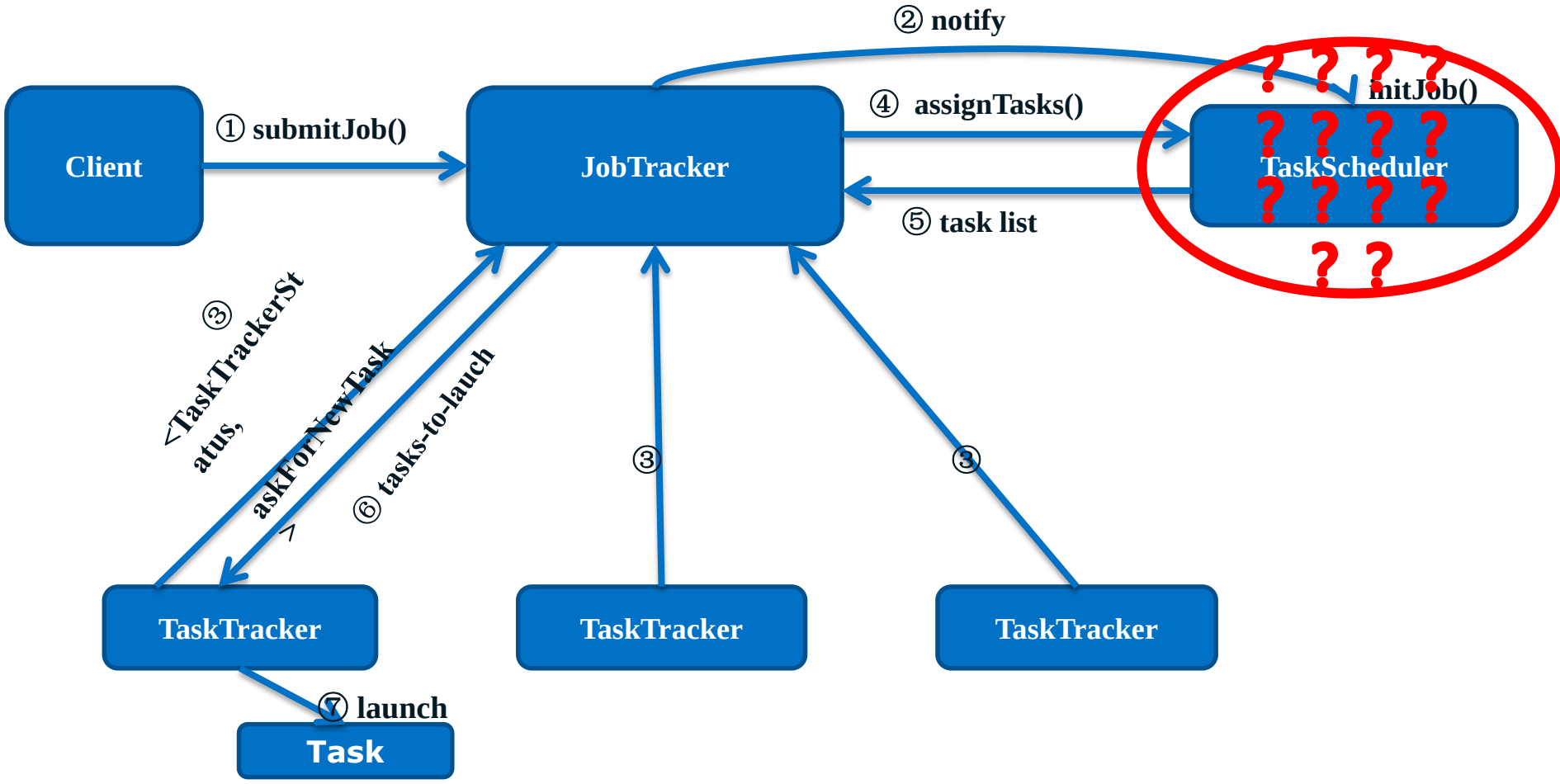
Fair Scheduler

- 如果只有一个Job，可使用集群所有资源
- 如有多个Job，空闲的slot会以“让每个用户公平共享集群”的方式分配。每个用户都有自己的Job pool
- 支持抢占

Capacity Scheduler

- 多个job queue。每个queue设置可占用资源比例和最大比例
- Queue内部FIFO调度。

Hadoop调度示意


② notify
Client
① submitJob()
JobTracker
④ assignTasks()
initJob()
TaskScheduler
⑤ task list
③
<TaskTrackerSt
atus,
askForNewTask
⑥ tasks-to-lauch
③
③
TaskTracker
TaskTracker
TaskTracker
⑦ launch
Task

Hadoop三级调度

所有调度器实际上均采用了三级调度策略，即为空闲的slot依次选择一个队列、作业和任务。

- **队列（queue）**
  - ✓ 用户被划分到某个队列
  - ✓ 每个队列分配一定量的资源

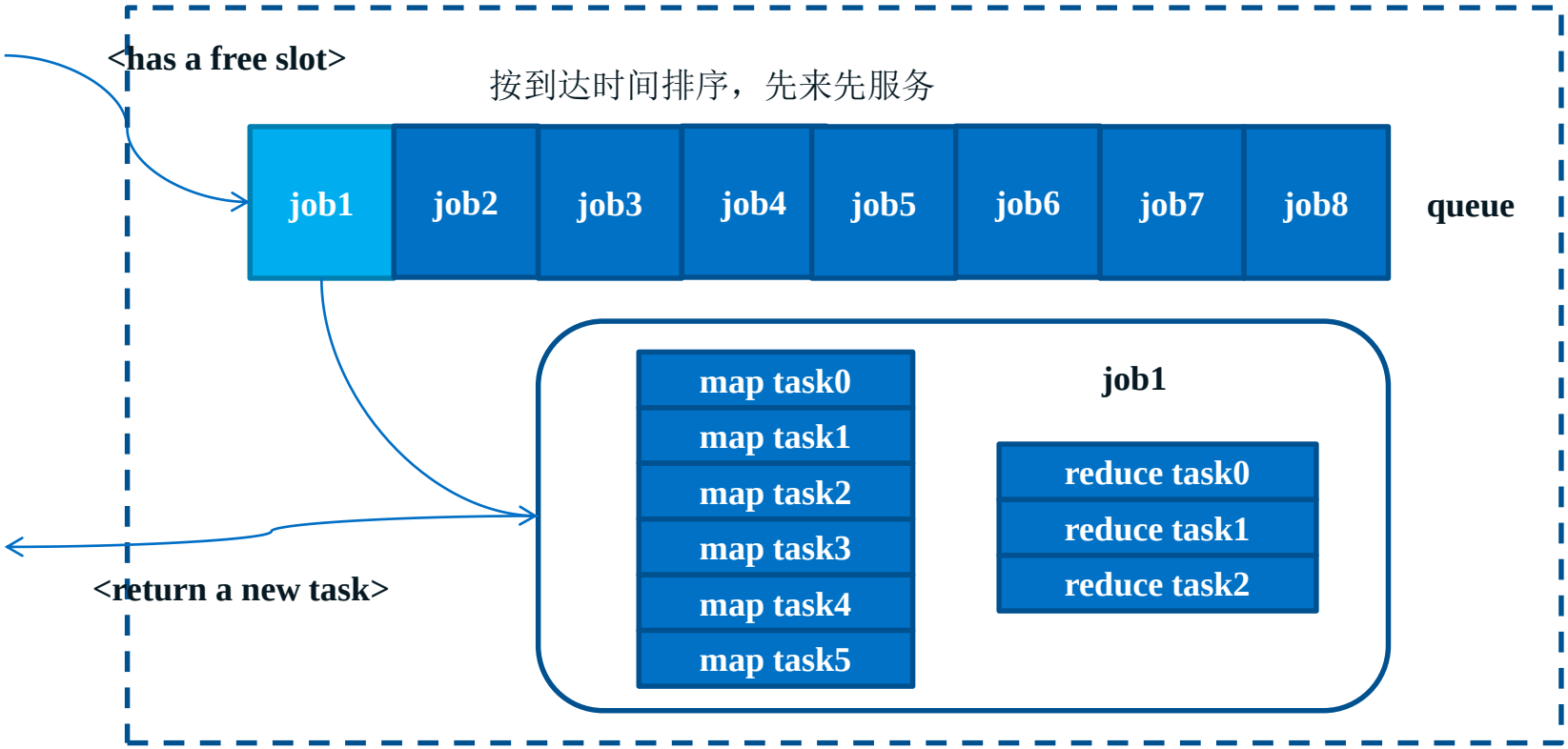
不同调度器，采用策略不同

- **作业（job）**
  - ✓ 提交时间
  - ✓ 优先级（5个优先级：VERY_HIGH,HIGH, NORMAL, LOW, VERY_LOW）

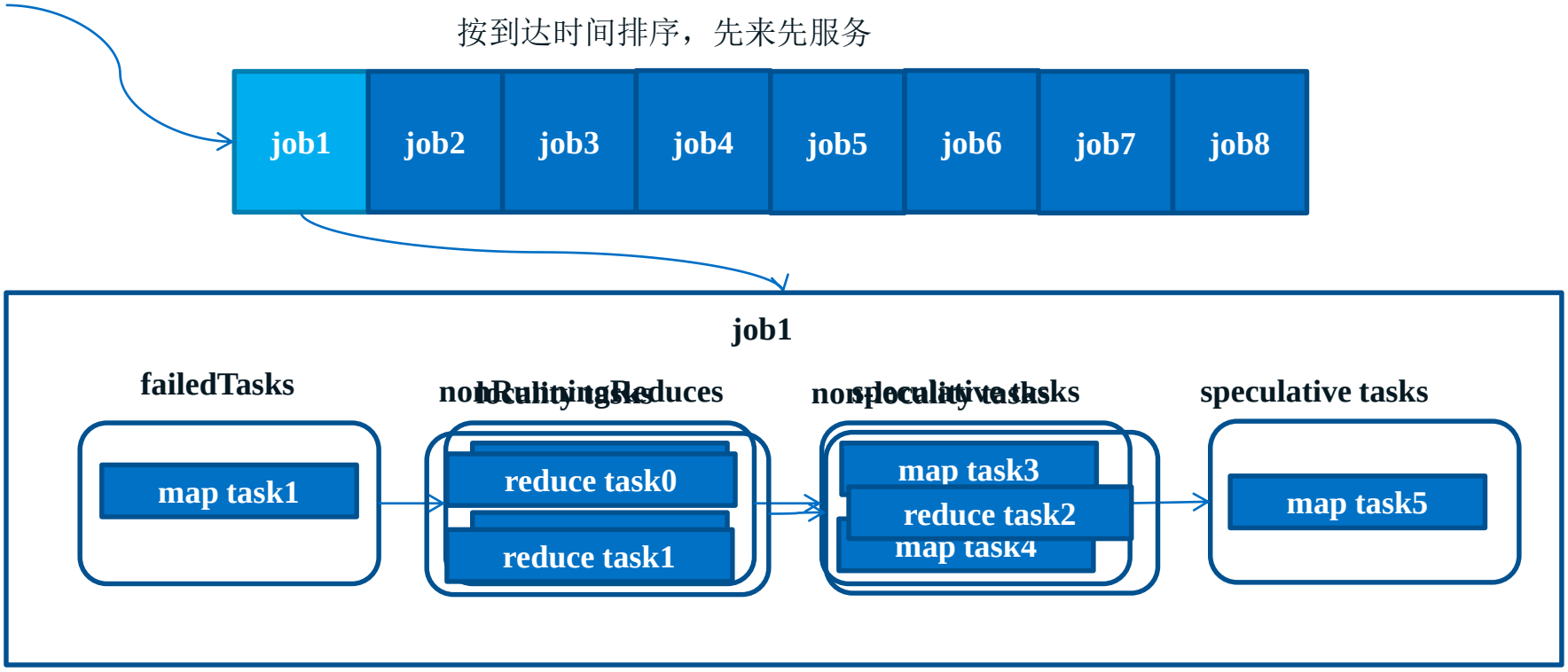
- **任务（task）**
  - ✓ 本地性（node locality, rack locality）

不同调度器，采用策略相同

Hadoop现有调度器(FIFO)



Hadoop现有调度器(FIFO)



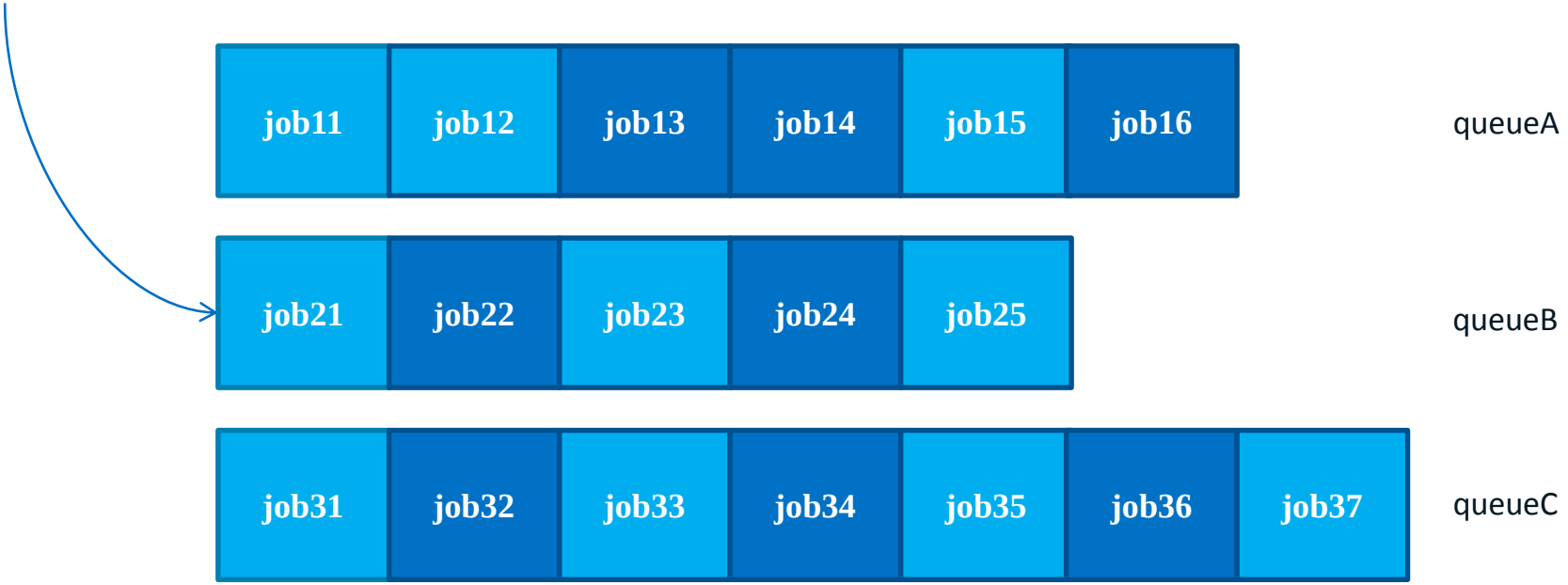
Hadoop现有调度器 (Capacity Scheduler)

Hadoop现有调度器 (Capacity Scheduler)

- 由Yahoo开源，共享集群调度器
- 以队列方式组织作业
- 每个队列内部采用FIFO调度策略
- 每个队列分配一定比例资源
- 可限制每个用户使用资源量

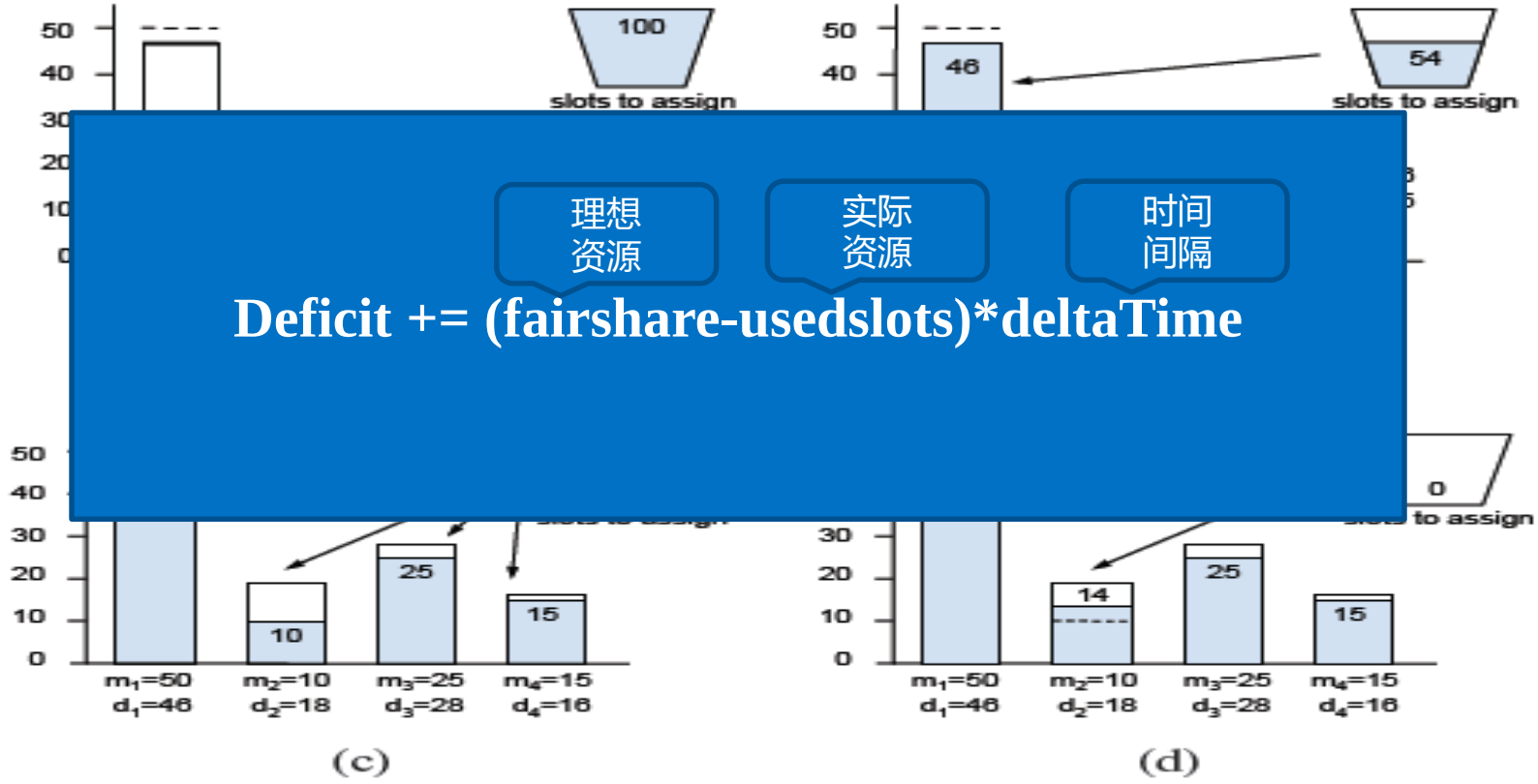
Hadoop现有调度器 (Fair Scheduler)

按缺额排序，缺额大者优先



Hadoop现有调度器 (Fair Scheduler)

- 由Facebook开源的，共享集群调度器
- 以队列方式组织作业
- 基于最小资源量(min share)与公平共享量(fair share)进行调度
- 作业优先级越高，分配到的资源越多


理想
资源
实际
资源
时间
间隔
Deficit += (fairshare-usedslots)*deltaTime
100
slots to assign
54
slots to assign
0
to assign
50
40
30
20
10
0
46
10
25
15
14
m_1=50
d_1=46
m_2=10
d_2=18
m_3=25
d_3=28
m_4=15
d_4=16
(c)
(d)

编写自己的Hadoop调度器

步骤1 编写JobInProgressListener

步骤2 编写调度器类，继承抽象类TaskScheduler

步骤3 配置并启用Hadoop调度器

让计算靠近数据

对于每个任务，理论上可以在任何一个节点上执行。但如果不进行局部性考虑的话，节点之间的带宽会成问题

MapReduce的思想：让计算靠近数据

调度器将尽量将任务和数据调度到同一个物理节点，或者至少是一个相同的机架

- 如果 Map的输入可以提供位置信息的话
  - HDFS的输入数据块存放在哪几个节点

使用Combiner函数进行优化

Combiner函数是一个在本地执行的reduce函数，在进行数据交换前在本地先做一次排序或者过滤

能够节省带宽

一般直接用Reduce函数做combiner

使用Combiner函数是否会改变程序的结果？WordCount程序呢？

任务JVM重用

每个任务都有一个独立的JVM来运行

启动JVM是一个不小的开销（~1s）

Hadoop允许在任务之间复用JVM
(mapred.job.reuse.jvm.num.tasks)

复用可能会引起内存等方面的问题，默认不复用

任务执行

Map或者Reduce任务的每次执行称为一次尝试(attempt)

尝试可以失败

- 失败的话，会被JobTracker调度到其他TaskTracker上重新执行
- 如果一个任务失败超过4次的话，整个Job会被认为失败

同一个任务可以有超过一个的尝试在同时执行（推测执行）

- 互不影响。当一个成功时，JobTracker将另一个杀死
- 好处：防止某个attempt拖慢Job
- 坏处：资源竞争，可能会拖慢Job

出错恢复

任务进程定期向TaskTracker发送心跳

- 如果10分钟无心跳，则认为死亡。其JVM被TaskTracker杀死

如果任务出现Exception，那么被认为死亡

TaskTracker定期向JobTracker发送心跳

- 这是两者间惟一的通信机制
- TaskTracker将完成的attempt，出错的attempt汇报给JobTracker
- JobTracker在心跳的reply中通知TaskTracker执行相应的动作
- 如果10分钟无心跳，则认为死亡。其正执行的所有任务被重新调度

如某个TaskTracker出现过高的任务失败比率，则这个TaskTracker会被JobTracker列入黑名单

MapReduce中的计数器

http://jobtracker:50030

计数器Counters用以进行对工作过程的跟踪，反映当前的工作的进度

用户可以定义与自己应用相关的计数器，以对自己的应用进行详细的跟踪

Hadoop job_200709211549_0003 on localhost

User: hadoop
Job Name: streamjob34453.jar
Job File: /usr/local/hadoop-datastore/hadoop-hadoop/mapred/system/job_200709211549_0003/job.xml
Status: Succeeded
Started at : Fri Sep 21 16:07:10 CEST 2007
Finished at: Fri Sep 21 16:07:26 CEST 2007
Finished in: 16sec

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	3	0	0	3	0	0 / 0
	100.00%	1	0	0	1	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	map tasks	0	0	3
	Launched reduce tasks	0	0	1
	Data-local map tasks	0	0	3
Map-Reduce Framework	Map input records	77,637	0	77,637
	Map output records	103,909	0	103,909
	Map input bytes	3,659,910	0	3,659,910
	Map output bytes	1,083,767	0	1,083,767
	Reduce input groups	0	85,095	85,095
	Reduce input records	0	103,909	103,909
	Reduce output records	0	85,095	85,095

Change priority from NORMAL to: VERY_HIGH HIGH LOW VERY_LOW

MapReduce程序的调优

调优是一个实践过程，只有通过实验才能得到最好性能

一些常用的经验：

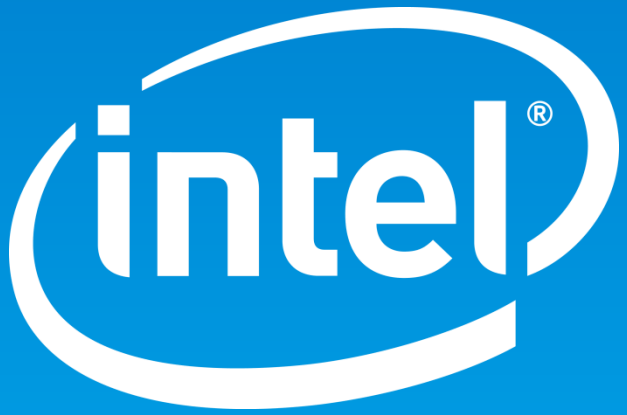
- 如果可能的话，不用Reduce
- 启用压缩（snappy）
- 使用combiner减少中间结果
- 每个Map任务处理的数据以128M~1G左右为宜。
- Reduce任务数量应适中，具体看reduce的任务性质和map输出的数据量
- 每个节点上的map slot + reduce slot个数应<2*CPU核数

MapReduce编程总结

MapReduce极大简化了大规模数据处理的编程方法

通过简单的抽象，用户可以只针对自己应用程序的本身进行编码，而不需要考虑底层的系统细节

MapReduce编程框架自动处理程序在大规模分布式系统中运行的细节，包括系统的扩展性以及可靠性


intel