云计算技术与工程

大数据处理技术

徐敏贤

副研究员
云计算研究中心
中国科学院深圳先进技术研究院
http://www.minxianxu.info/vcc

鱼水三顾合，风云四海生。 武侯立岷蜀，壮志吞咸京。
——（唐）李白

内容回顾

- Q1: Ceph基于对象的存储架构有哪些核心设计?

- 核心是将数据通路（数据读或写）和控制通路（元数据）分离，并且基于对象存储设备（Object-based Storage Device，OSD）构建存储系统，**每个对象存储设备具有一定的智能**，能够自动管理其上的数据分布。对象存储结构由对象、对象存储设备、元数据服务器、对象存储系统的客户端四部分组成。
- OSD管理存储在其上**对象的元数据**，该元数据与传统的inode元数据相似，通常包括对象的数据块和对象的长度。对象存储架构将系统中主要的元数据管理工作由OSD来完成，降低了Client的开销。

内容回顾

- Q2: HBase解决了传统关系型数据库（RDBMS）的哪些问题？它的特点有哪些？

RDBMS处理海量数据的问题：

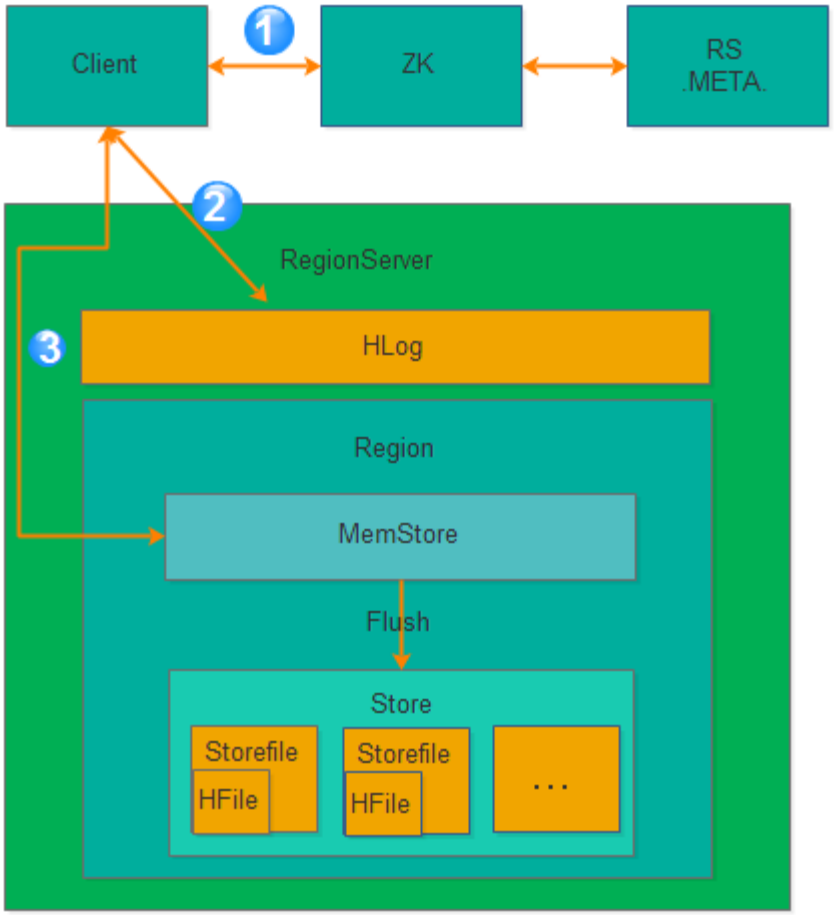
- 可扩展性困难
  - Scale up: scale vertically → costly
  - Scale out: scale horizontally → difficult
- 低可用性
  - 索引维护开销巨大
  - “链接 Join”操作开销巨大

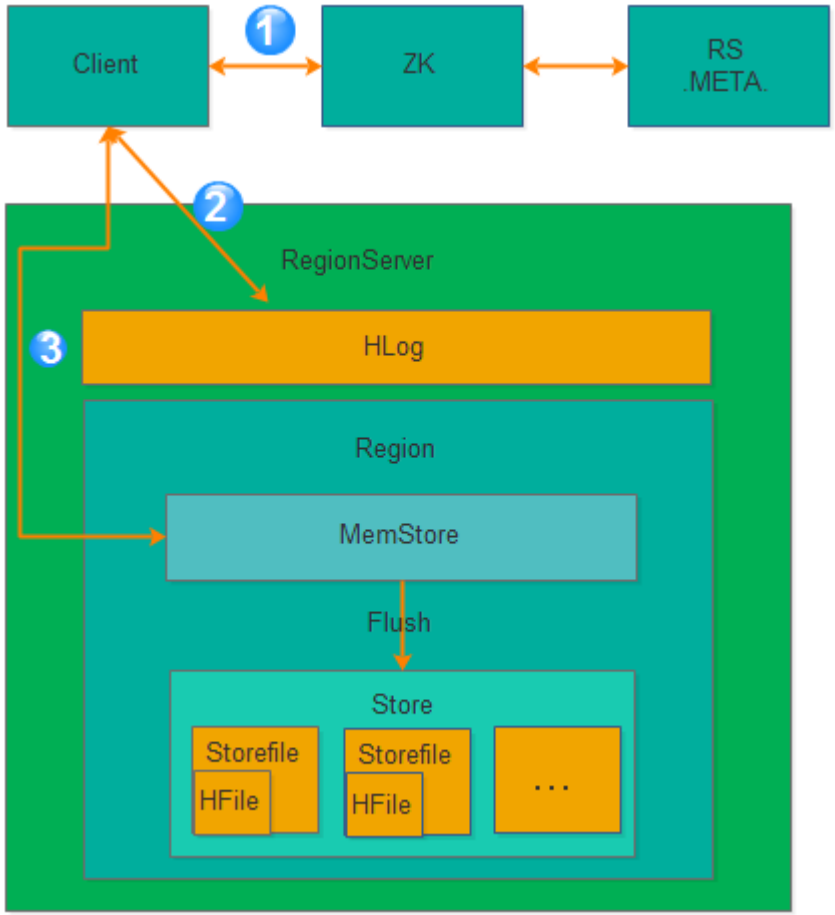
- HBase – Hadoop Database，是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用HBase技术可在廉价PC Server上搭建起大规模结构化存储集群

- 为什么需要设计HBase?
  - 如何存储种类总多的数据，如网页，邮件……
  - 如何使用统一的方式存储各类数据?
  - 如何快速地从海量信息中寻找需要的数据?
- 数据存储可靠性
- 存储海量的记录（若干TB），便于压缩
- 可以保存记录的多个版本
- 与写操作相比，数据记录读操作占大多数工作负载
- 单个节点故障损坏是常见的，需要高可用性
- 高可扩展性

内容回顾

- Q3: 根据下图介绍一下Hbase的写入逻辑





①Client获取数据写入的Region所在的RegionServer

②请求写Hlog

③请求写MemStore。只有当写Hlog和写MemStore都成功了才算请求写入完成。MemStore后续会逐渐刷到HDFS中。

Hlog存储在HDFS，当RegionServer出现异常，需要使用Hlog来恢复数据。

内容回顾

- Q4: Hive具有高可扩展性、延展性、容错性等优点，但存在哪些缺点?

（+） **可扩展性,横向扩展**，Hive 可以自由的扩展集群的规模，一般情况下不需要重启服务 横向扩展：通过分担压力的方式扩展集群的规模 纵向扩展：一台服务器cpu i7-6700k 4核心8线程，8核心16线程，内存64G => 128G

（+） **延展性**，Hive 支持自定义函数，用户可以根据自己的需求来实现自己的函数

（+） **良好的容错性**，可以保障即使有节点出现问题，SQL 语句仍可完成执行

（-） **Hive 不支持记录级别的增删改操作**，但是用户可以通过查询生成新表或者将查询结 果导入到文件中

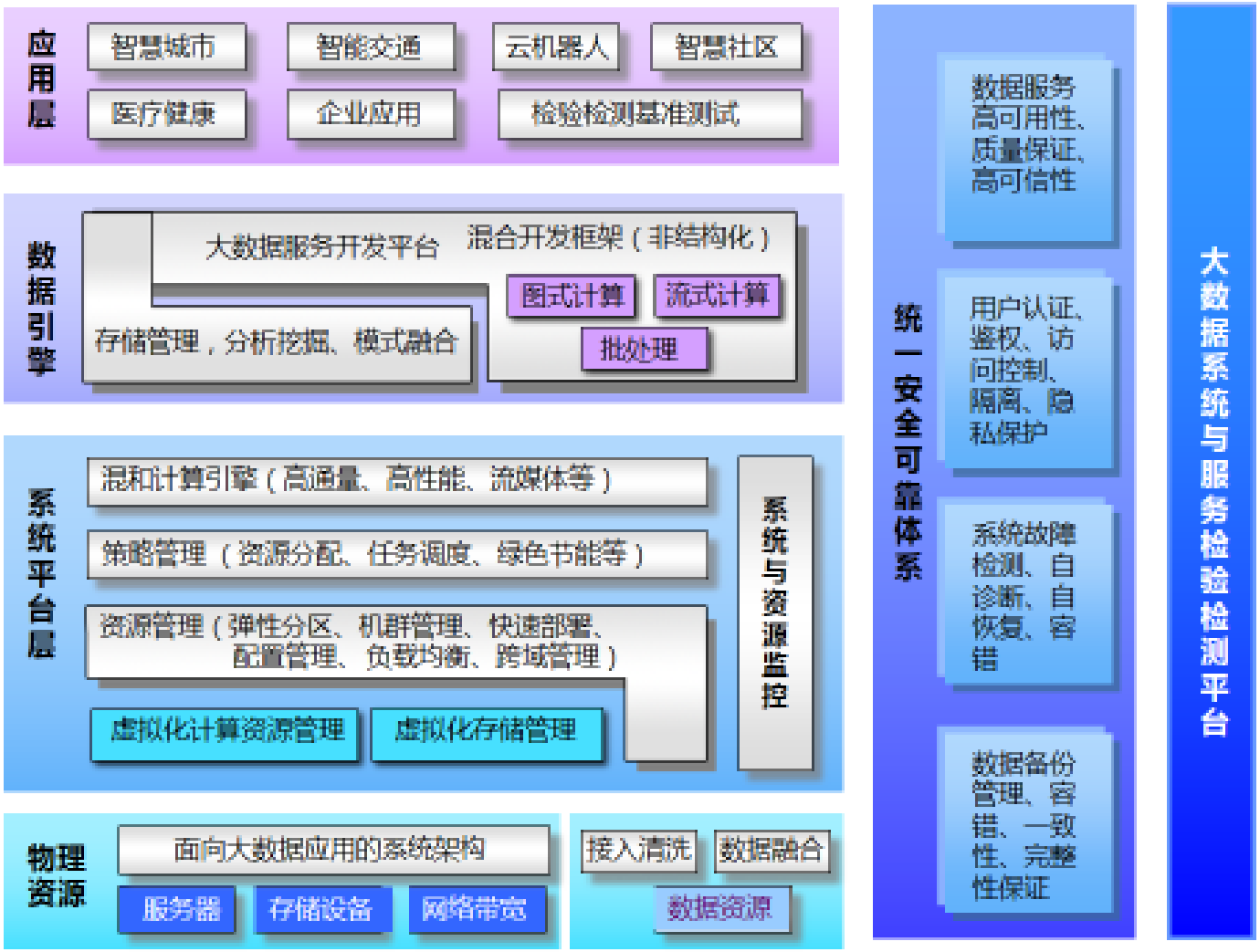
（-） **Hive 的查询延时很严重**，因为 MapReduce Job 的启动过程消耗很长时间，所以不能用在交互查询系统中。

（-） **Hive 不支持事务**（因为没有增删改，所以主要用来做 OLAP（联机分析处理），而 不是 OLTP（联机事务处理），这就是数据处理的两大级别）。

目录 Contents

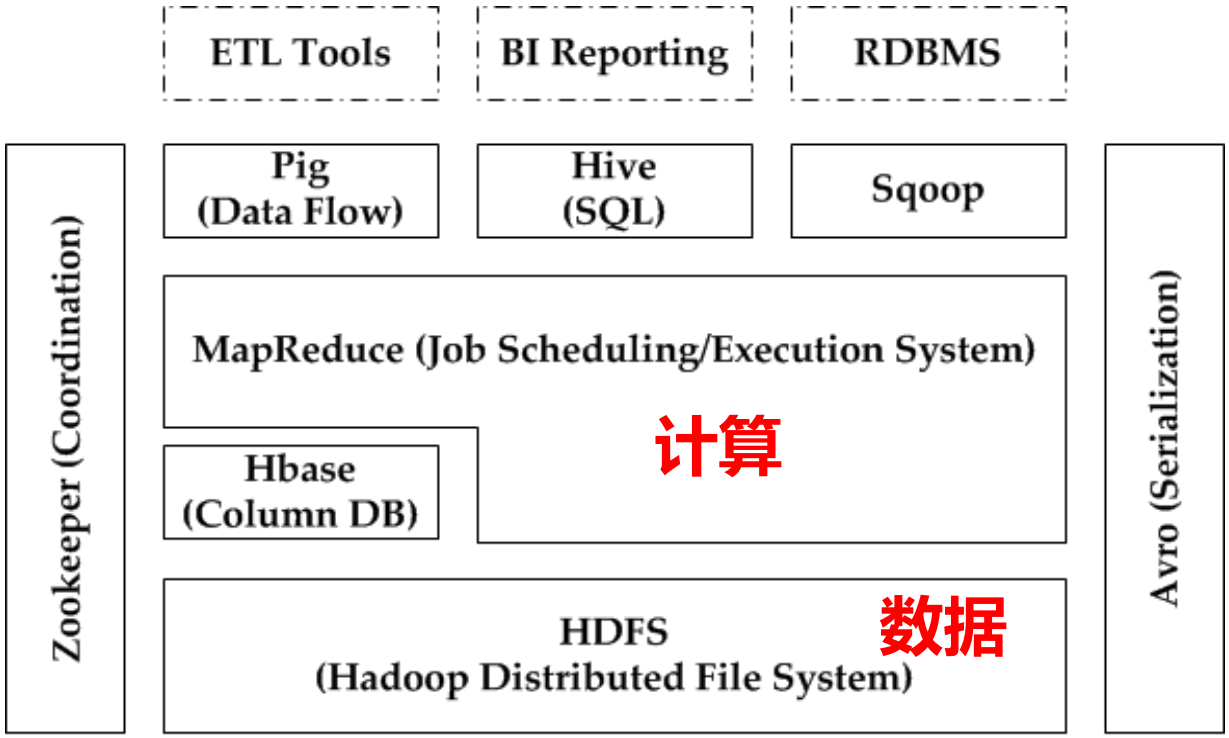
注：本章部分内容由王洋教授提供

大数据框架研讨

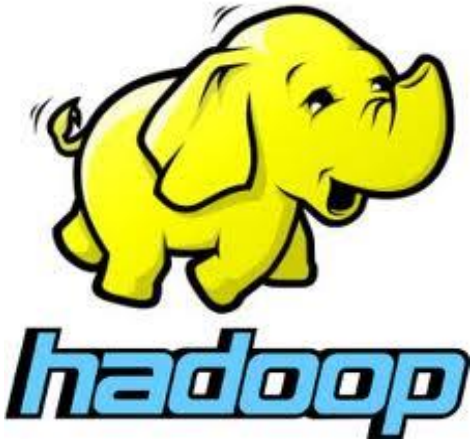

应用层
智慧城市
智能交通
云机器人
智慧社区
医疗健康
企业应用
检验检测基准测试
数据引擎
大数据服务开发平台
混合开发框架（非结构化）
图式计算
流式计算
批处理
存储管理，分析挖掘、模式融合
系统平台层
混和计算引擎（高通量、高性能、流媒体等）
策略管理（资源分配、任务调度、绿色节能等）
资源管理（弹性分区、机群管理、快速部署、配置管理、负载均衡、跨域管理）
虚拟化计算资源管理
虚拟化存储管理
系统与资源监控
物理资源
面向大数据应用的系统架构
服务器
存储设备
网络带宽
接入清洗
数据融合
数据资源
统一安全可靠体系
数据服务高可用性、质量保证、高可信性
用户认证、鉴权、访问控制、隔离、隐私保护
系统故障检测、自诊断、自恢复、容错
数据备份管理、容错、一致性、完整性保证
大数据系统与服务检验检测平台

处理架构

- Yahoo, Facebook, 阿里云,百度：Hadoop架构；

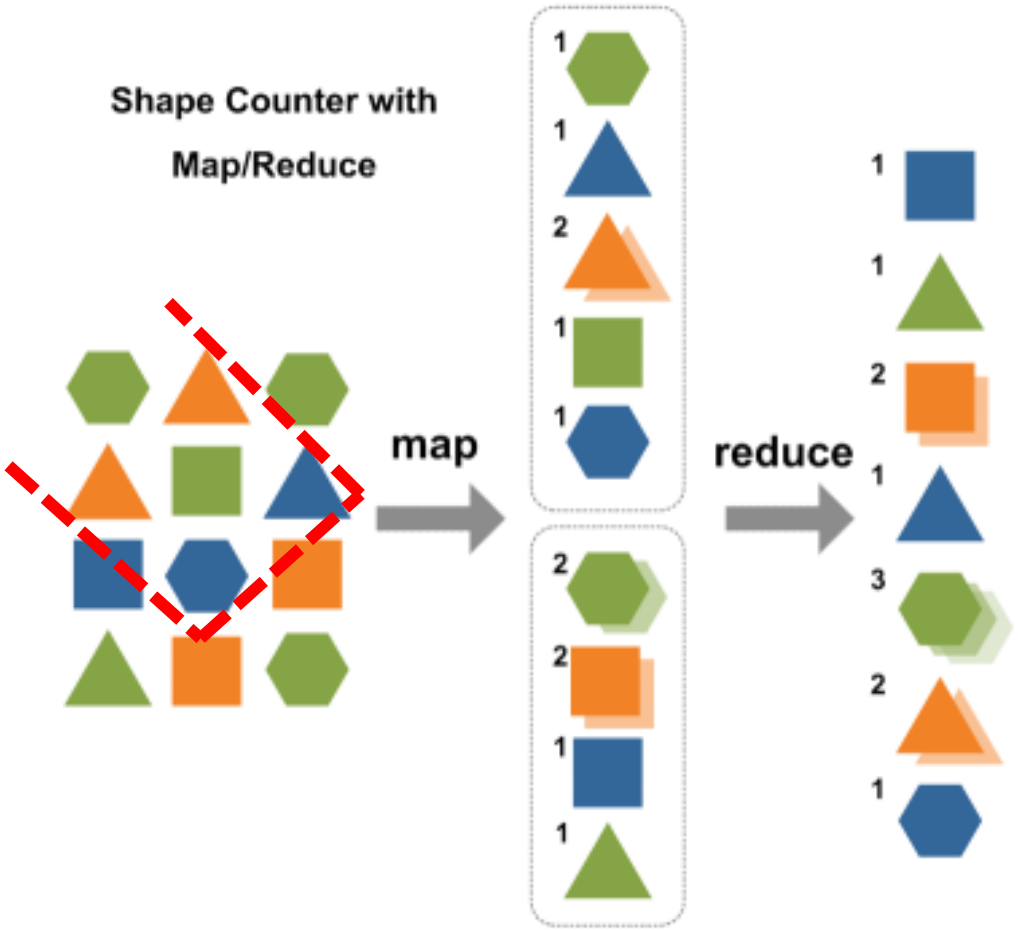


The Hadoop Ecosystem

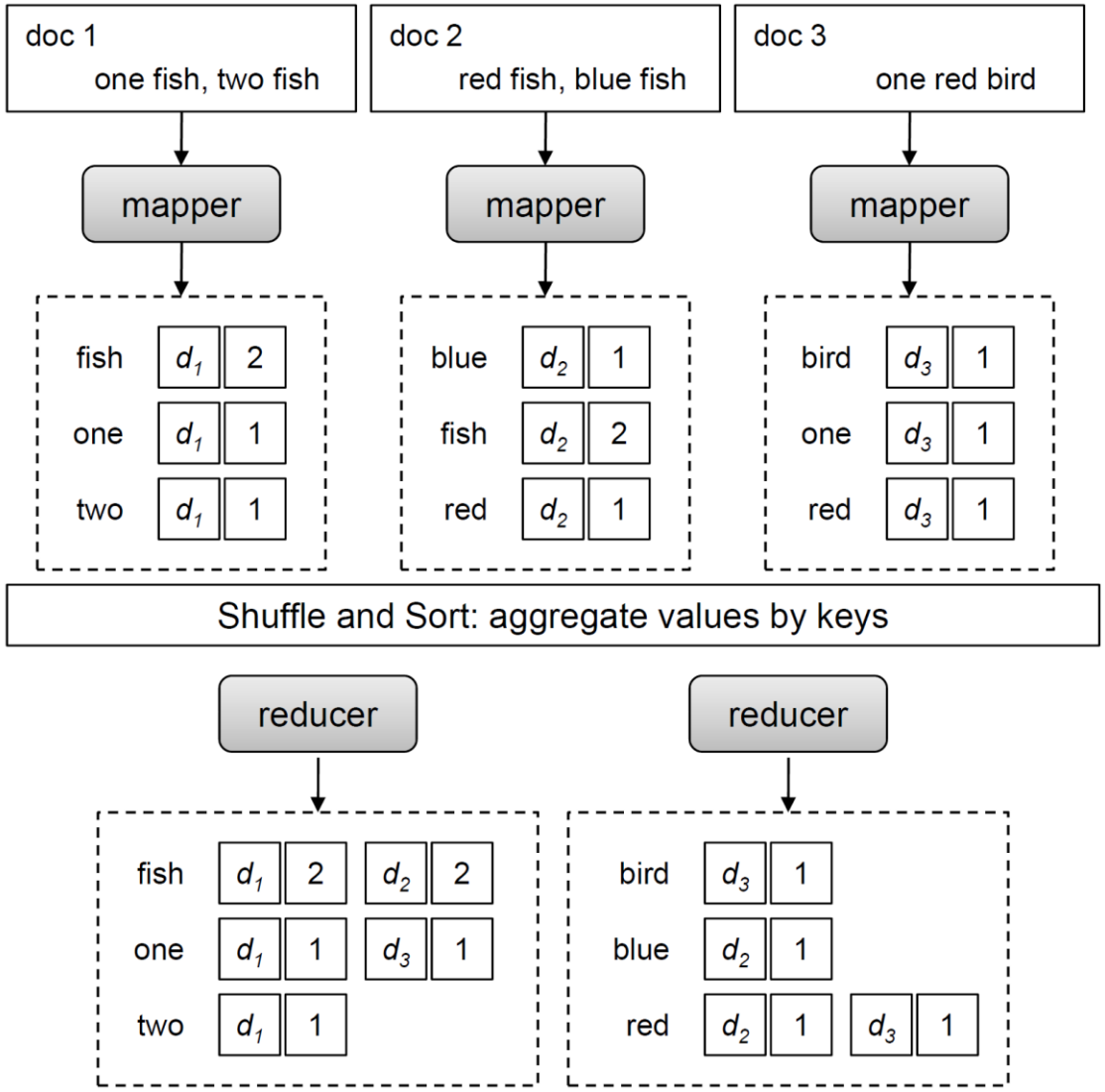


MapReduce简介

- Map阶段
  - 本地计算
- Reduce阶段
  - 全局计算



MapReduce

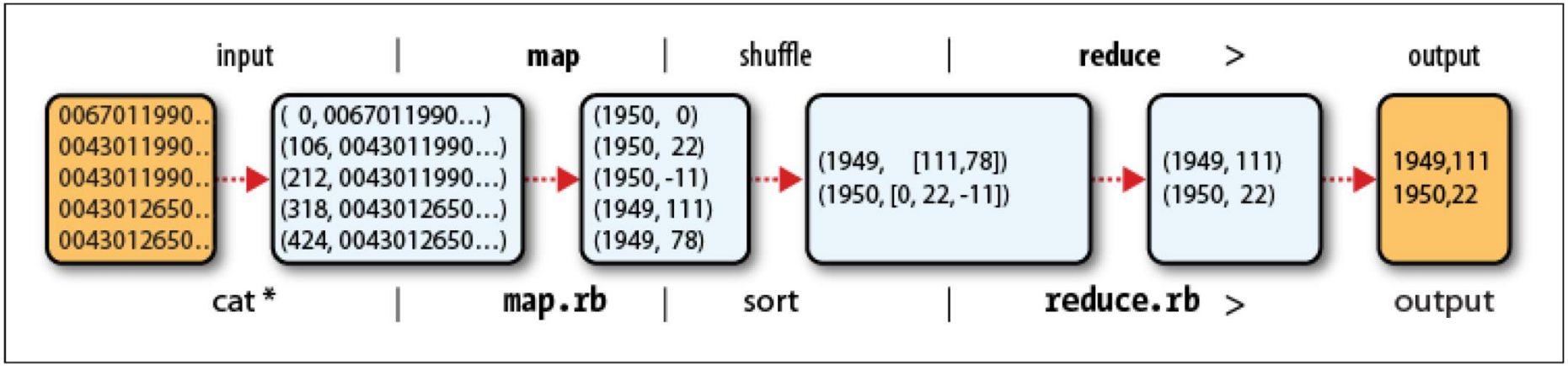

doc 1
one fish, two fish
doc 2
red fish, blue fish
doc 3
one red bird
mapper
mapper
mapper
fish d_1 2
one d_1 1
two d_1 1
blue d_2 1
fish d_2 2
red d_2 1
bird d_3 1
one d_3 1
red d_3 1
Shuffle and Sort: aggregate values by keys
reducer
reducer
fish d_1 2 d_2 2
one d_1 1 d_3 1
two d_1 1
bird d_3 1
blue d_2 1
red d_2 1 d_3 1

MapReduce简介

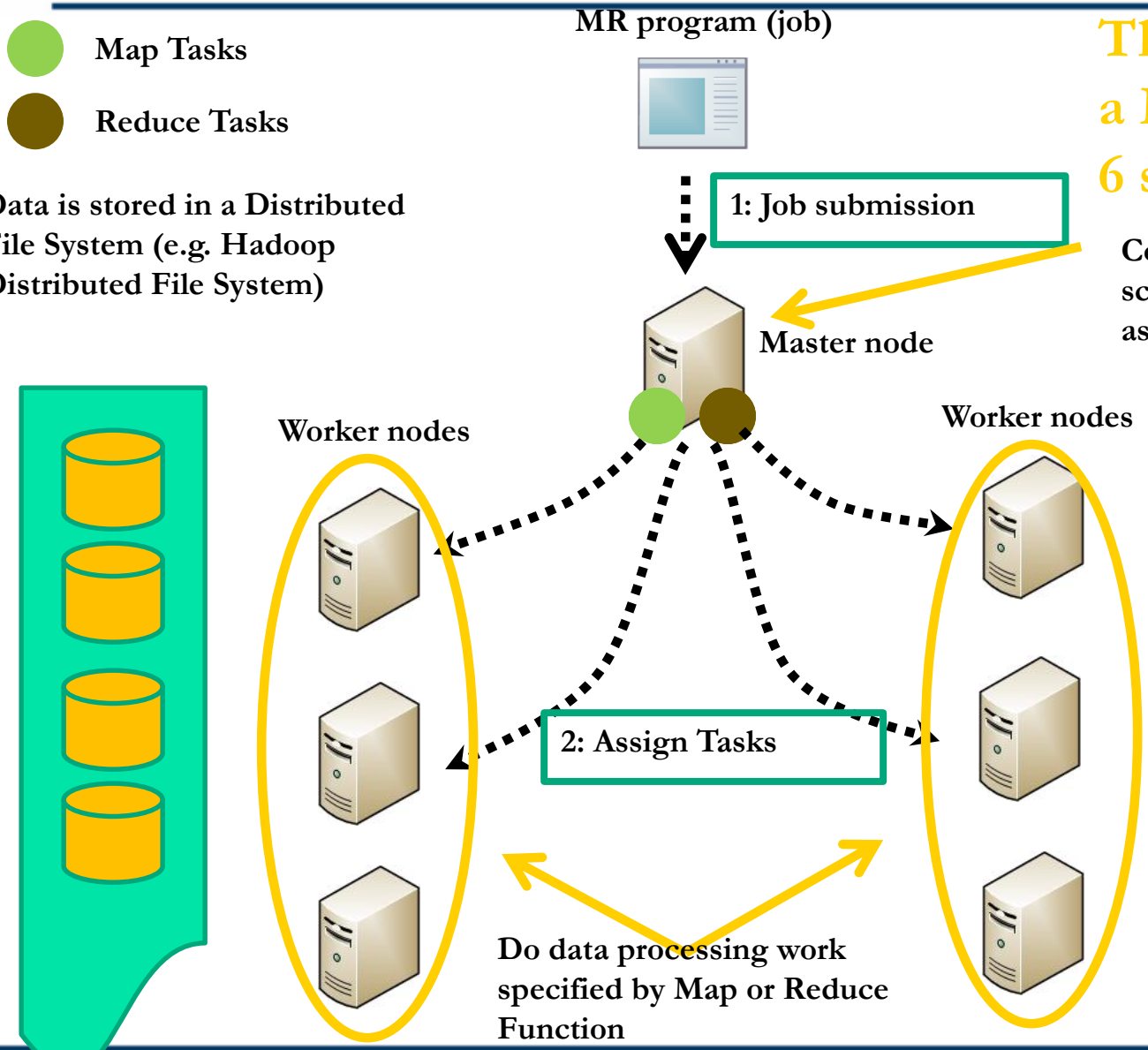
Map和Reduce函数格式：

Map: (K1, V1) -> list(K2, V2)

reduce: (K2, list(V2)) -> list(K3, V3)



MapReduce过程

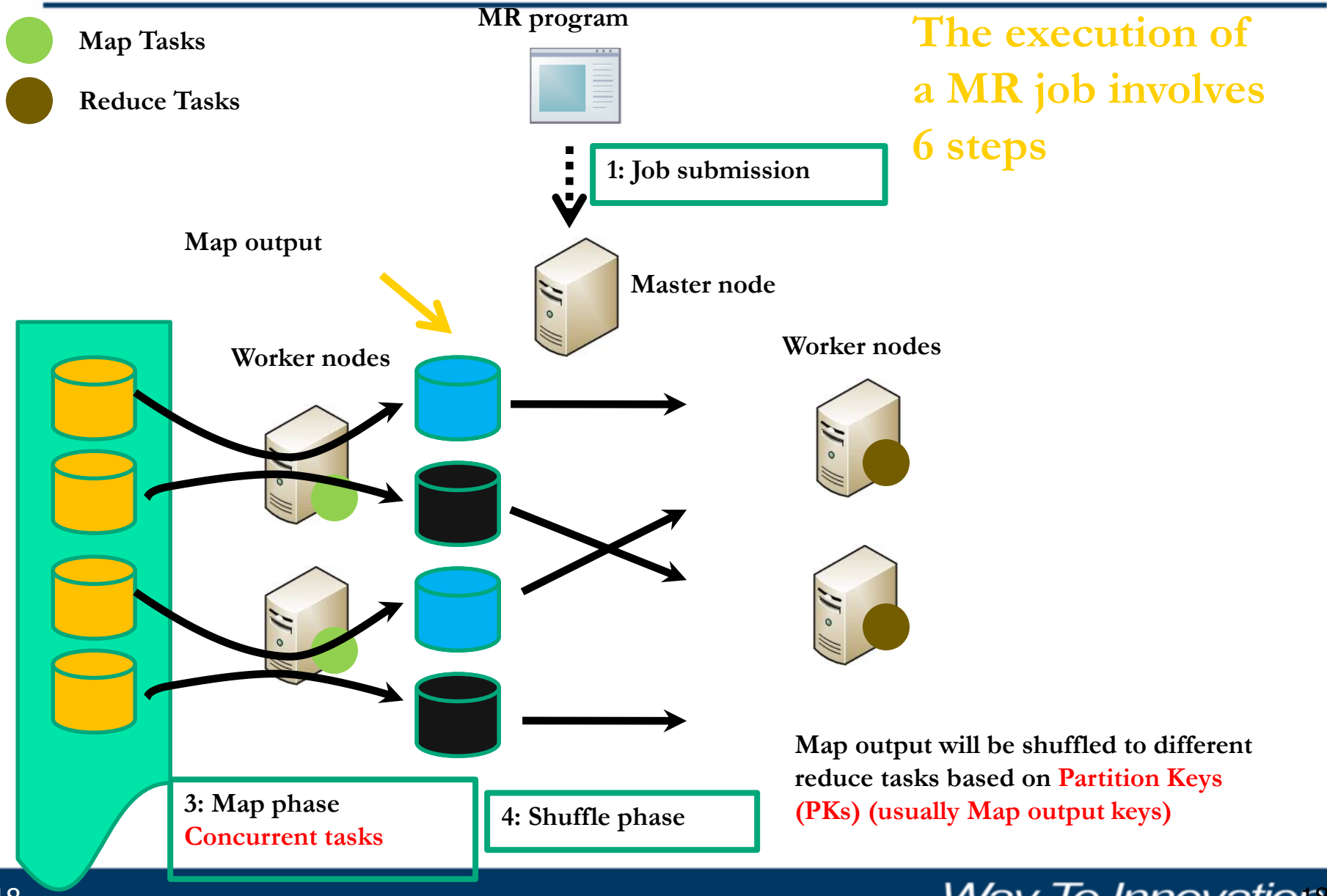


Data is stored in a Distributed File System (e.g. Hadoop Distributed File System)

The execution of a MR job involves 6 steps

Control level work, e.g. job scheduling and task assignment

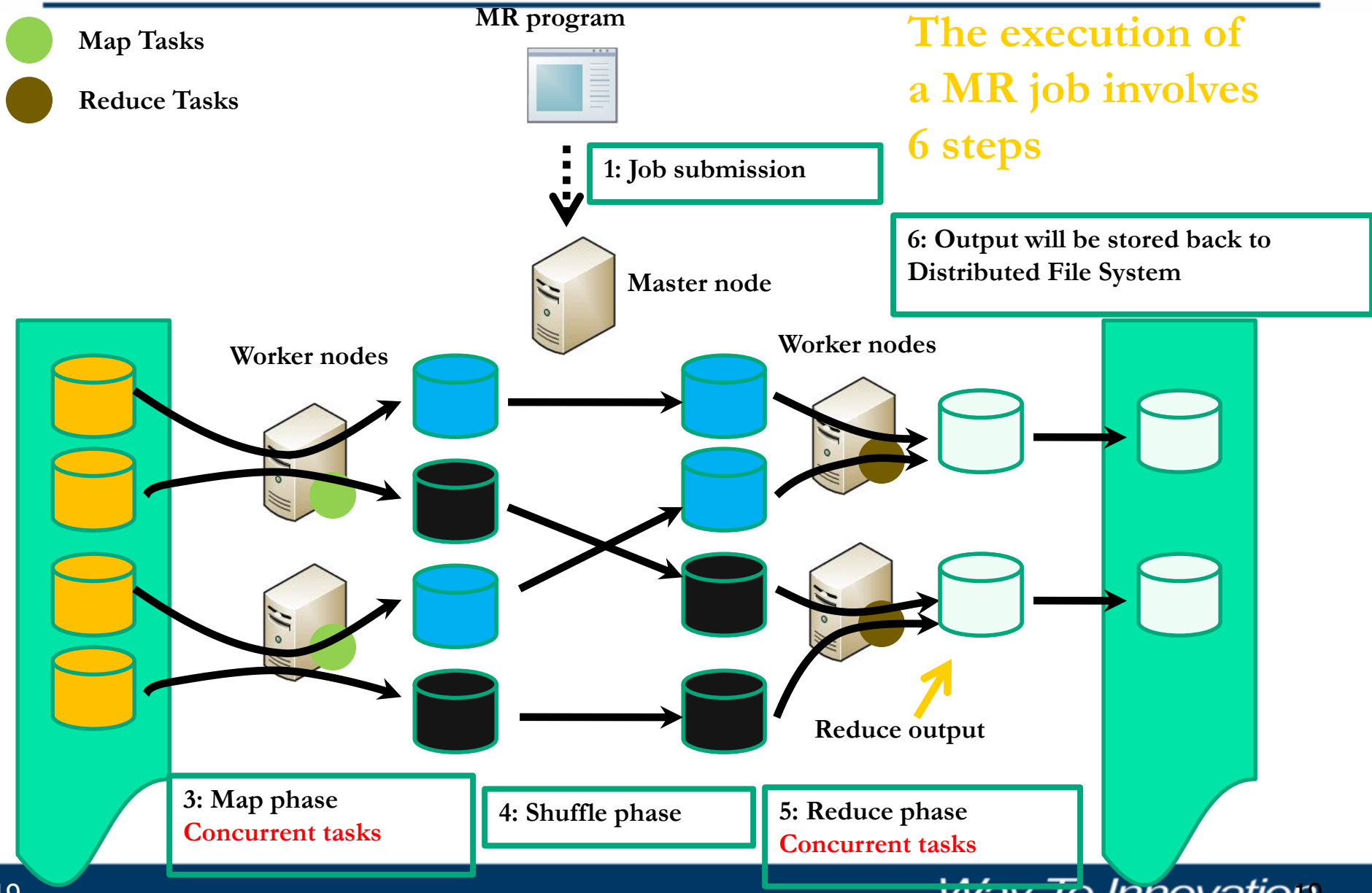
MapReduce过程



The execution of a MR job involves 6 steps

Map output will be shuffled to different reduce tasks based on Partition Keys (PKs) (usually Map output keys)

MapReduce过程



19

MR vs. RDBMS

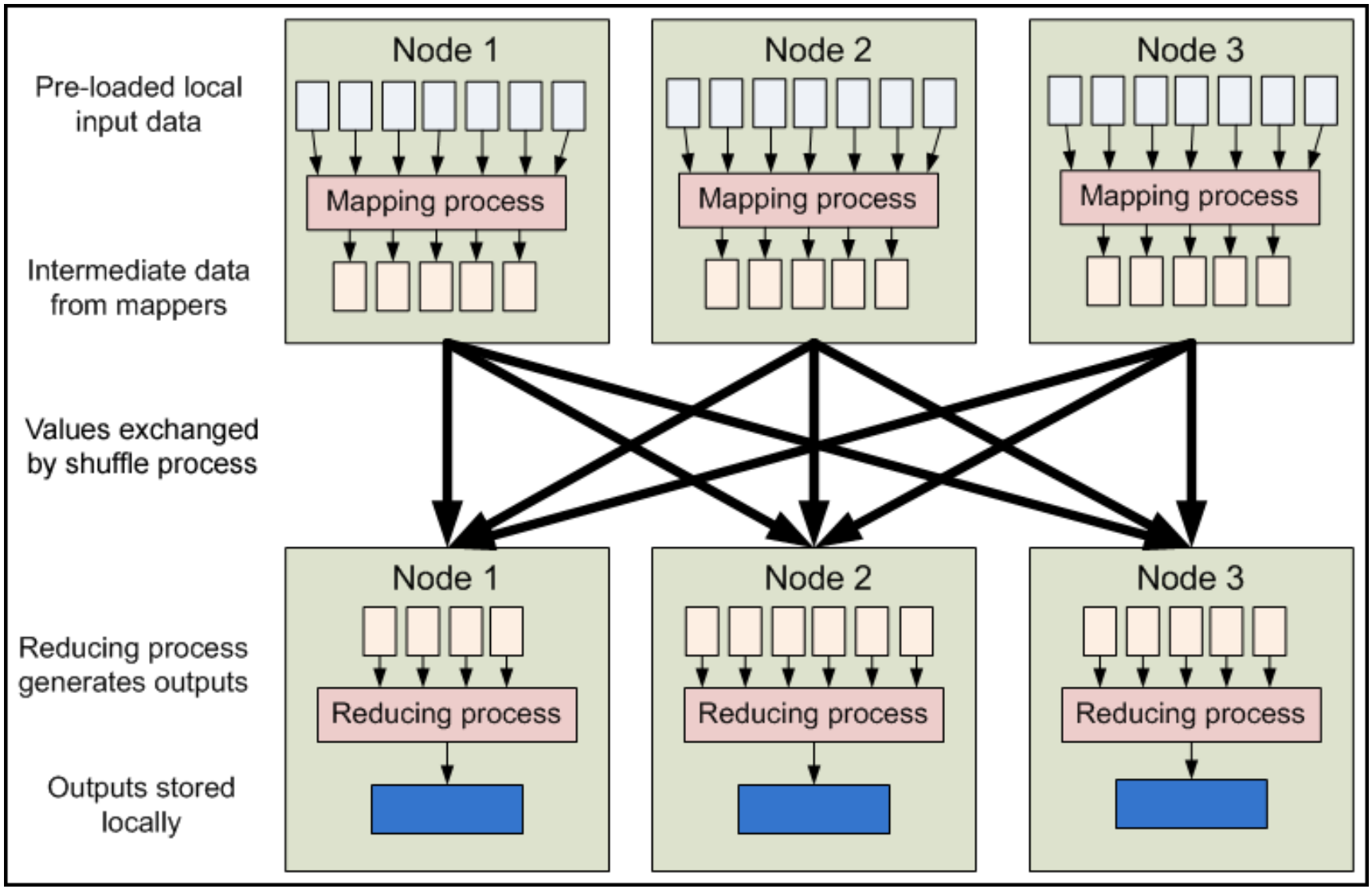
- **RDBMS 和MapReduce 技术的竞争与相互渗透**

	RDBMS	MapReduce
模式	内部支持	外部附加
索引	内部支持	编程实现
数据类型	结构化数据	非结构化、半结构化、结构化数据
编程模型	声明性语言 SQL	过程性语言
灵活性	有限	大
扩展性	上百节点	上千节点
容错性	低,查询重启	高,子任务重新执行
性能	高	比 RDBMS 低 [1]
应用范围	在线事务处理 在线分析处理	批量处理 [2] 深度分析

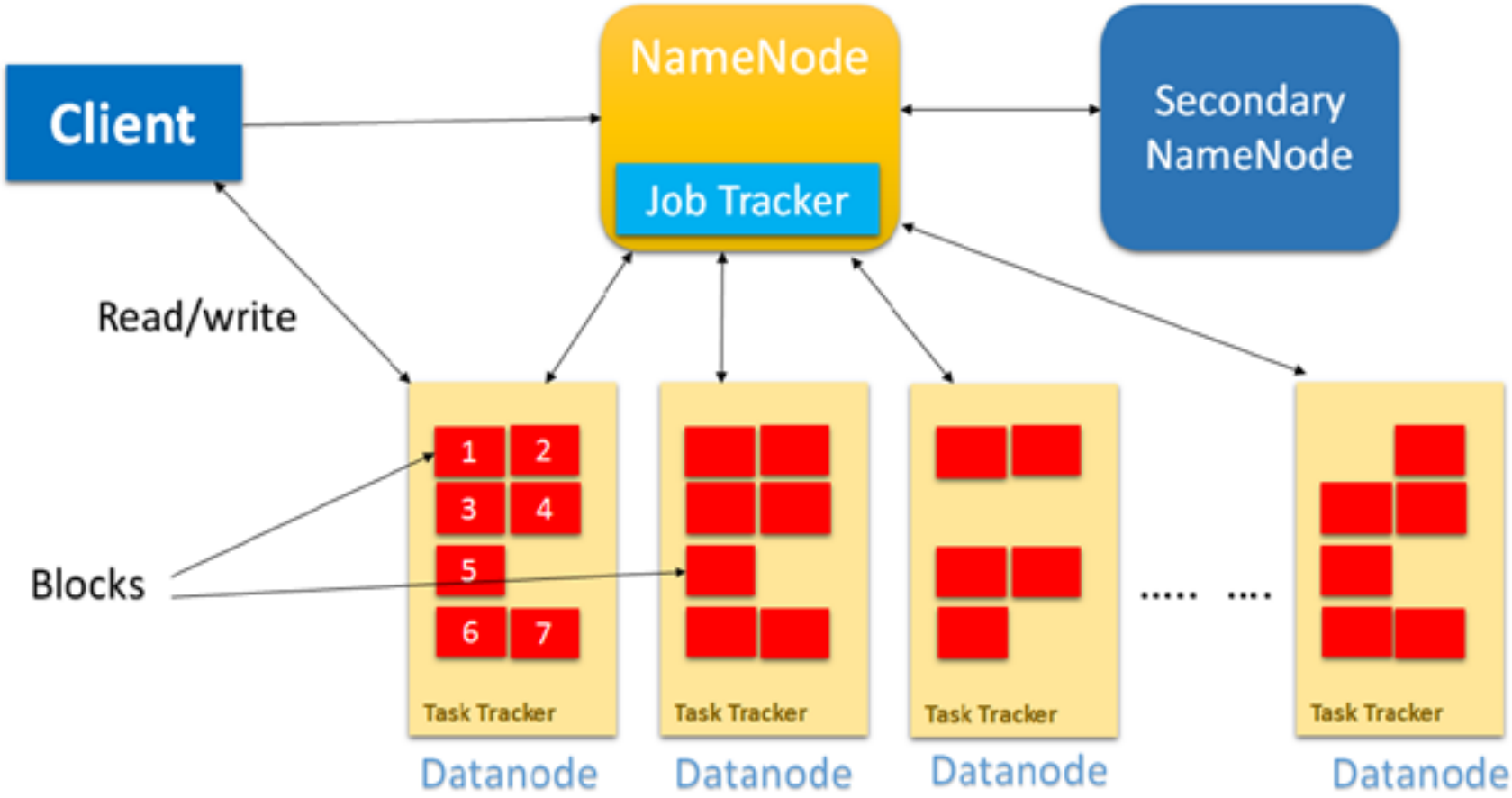
MapReduce特点

- 处理海量数据（>1TB）
- 上百/上千CPU实现并行处理
- 移动计算比移动数据更划算
- 自动实现分布式并行计算
- 模型抽象简洁，程序员易用
- 提供状态监控工具
- 容错性

MapReduce – 数据流

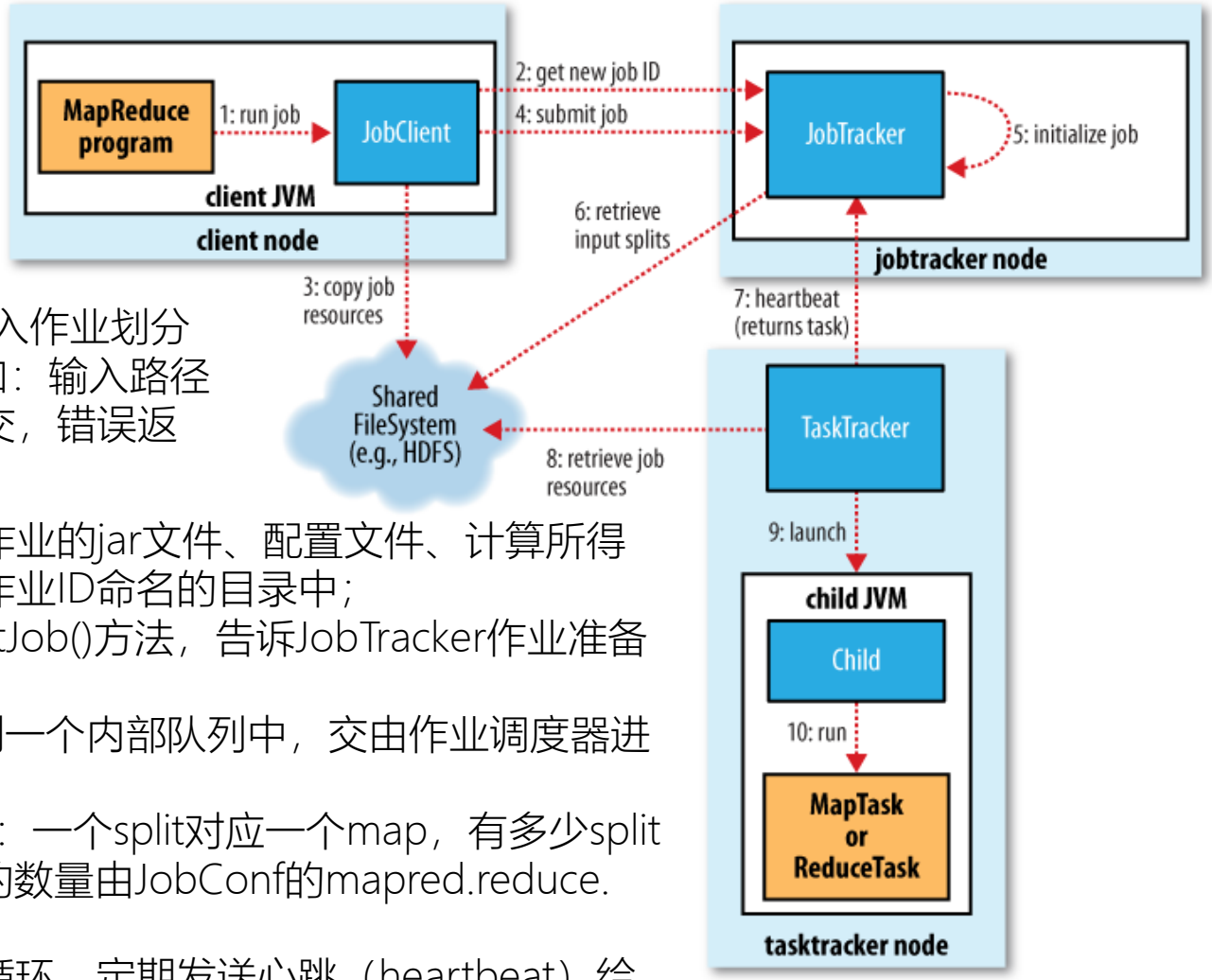


MapReduce架构



MapReduce工作流

① map/reduce程序通过runJob()方法新建一个JobClient实例;
② 向JobTracker请求一个新jobID，通过JobTracker的getNewJobId()获取;
③ 检查作业输入输出说明， 输入作业划分split，如果划分无法计算（如：输入路径不存在），作业将不会被提交，错误返回给map/reduce程序。



④ 将运行作业所需要的资源（作业的jar文件、配置文件、计算所得的输入划分）复制到一个以作业ID命名的目录中;
⑤ 通过调用JobTracker的submitJob()方法，告诉JobTracker作业准备提交;
⑥ JobTracker将提交的作业放到一个内部队列中，交由作业调度器进行调度，并对其进行初始化。
⑦ 创建Map任务、Reduce任务：一个split对应一个map，有多少split就有多少map; Reduce任务的数量由JobConf的mapred.reduce.tasks属性决定
⑧ TaskTracker执行一个简单的循环，定期发送心跳（heartbeat）给JobTracker

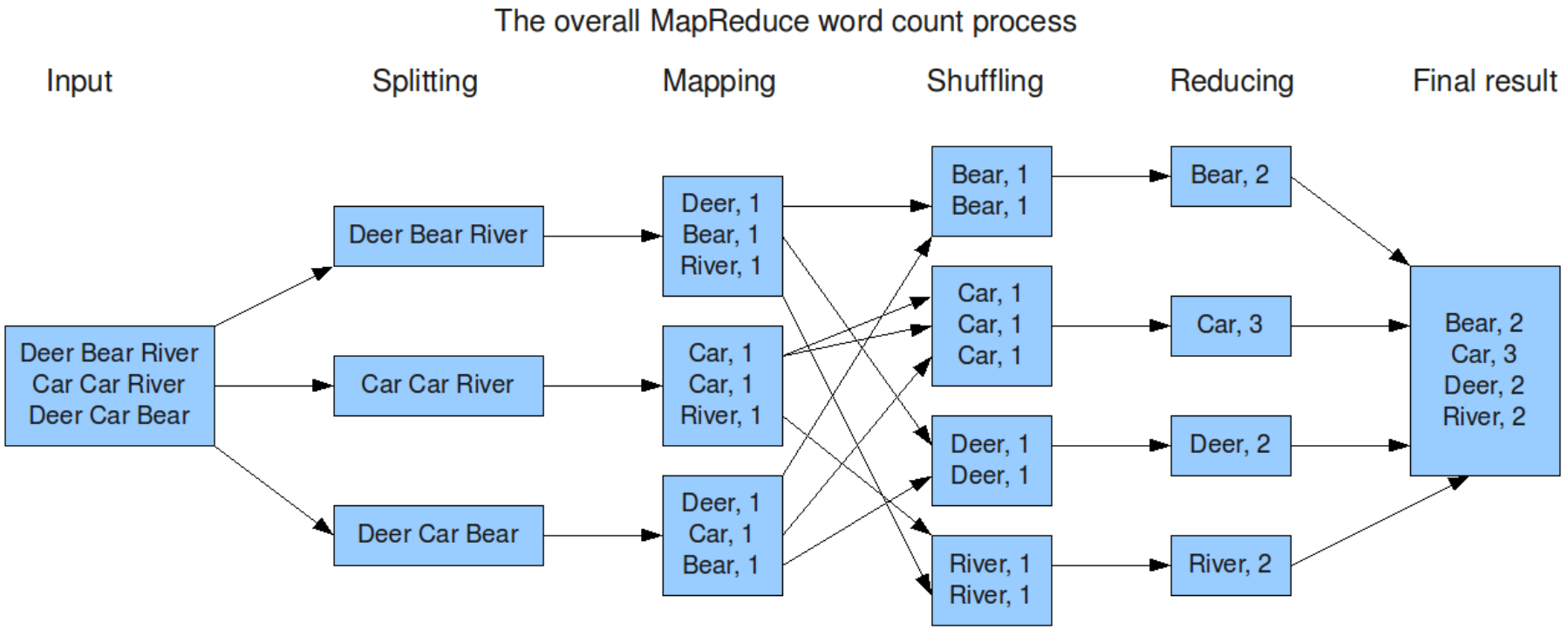
MapReduce – 特点

- 细粒度Map 和 Reduce 任务
  - 提示负载平衡
  - 快速从失败任务恢复
- 失败后自动从新执行
  - Framework re-executes failed tasks
- 局部性优化
  - 对大数据集，带宽是个问题
  - MapReduce + HDFS 有效的方法
  - MapReduce 查询 HDFS 定位输入数据
  - Map 尽可能调度到输入数据附近的位置

Word Count 示例

- Mapper
  - Input: value: lines of text of input
  - Output: key: word, value: 1
- Reducer
  - Input: key: word, value: set of counts
  - Output: key: word, value: sum
- Launching program
  - Defines this job
  - Submits job to cluster

Word Count数据流



Word Count Mapper

```
public static class Map extends MapReduceBase implements
    Mapper<LongWritable,Text,Text,IntWritable> {
  private static final IntWritable one = new IntWritable(1);
  private Text word  = new Text();

  public static void map(LongWritable key, Text value,
   OutputCollector<Text,IntWritable> output, Reporter reporter)
   throws IOException {
    String line = value.toString();
    StringTokenizer = new StringTokenizer(line);
    while(tokenizer.hasNext()) {
      word.set(tokenizer.nextToken());
      output.collect(word,one);
      }
    }
}
```

Word Count Reducer

```
public static class Reduce extends MapReduceBase implements
    Reducer<Text,IntWritable,Text,IntWritable> {
public static void reduce(Text key, Iterator<IntWritable> values,
    OutputCollector<Text,IntWritable> output, Reporter reporter)
    throws IOException {
        int sum = 0;
        while(values.hasNext()) {
           sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
 }
```

Word Count 其他

- 通过配置***JobConfs*** 控制Job
- JobConfs 是一个从属性名到字符串值的映射集
- 框架定义了属性控制Job如何被执行
  - conf.set(“mapred.job.name”, “MyApp”);
- 应用可以给 JobConf加入任意值
  - conf.set(“my.string”, “foo”);
  - conf.set(“my.integer”, 12);
- JobConf 对所有的tasks起作用

总结

- 为你的应用建立一个发射程序
- 发射程序要配置:
  - 用到的*Mapper* 和 *Reducer*
  - output key 和 value types (input types *InputFormat*推导出来)
  - input 和 output 位置
- 发射程序然后提交 job 然后等待其运行完毕

总结

```
JobConf conf = new JobConf(WordCount.class);
conf.setJobName( “wordcount” );

conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(Map.class);
conf.setCombinerClass(Reduce.class);
conf.setReducer(Reduce.class);

conf.setInputFormat(TextInputFormat.class);
Conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));

JobClient.runJob(conf);
```

Input and Output格式

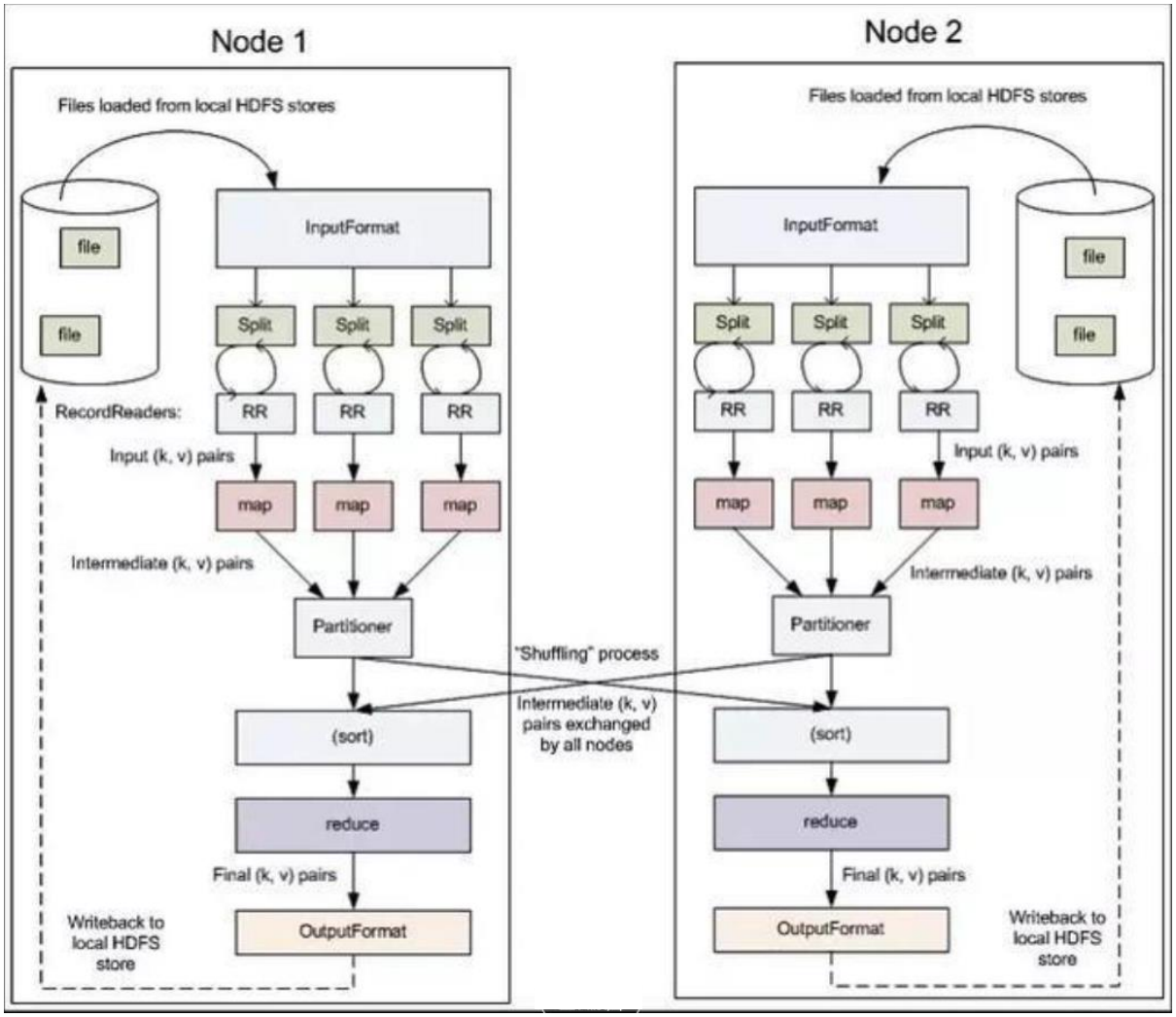
- A Map/Reduce 可利用*InputFormat* 说明其input将要被如何读入
- A Map/Reduce可利用*outputFormat* 说明其put将要被如何写出
- *TextInputFormat* 和*TextOutputFormat*是缺省情况
- 另一个共同选择是针对任意数据的 *SequenceFileInputFormat* 和 *SequenceFileOutputFormat*
- 这些是基于文件的, 但并不要求一定这样

多少个Maps and Reduces

- Maps
  - 通常有多少HDFS blocks就有多少（缺省）
  - 否则 maps 个数can be specified as a hint
  - Maps个数 can also be controlled by specifying the *minimum split size*
  - Maps的输入大小有下式计算:
    - *max(min(block_size,data/#maps), min_split_size)*
- Reduces
  - 除非被处理的数据量很小
    *0.95*num_nodes*mapred.tasktracker.tasks.maximum*

一些很方便的工具

- Partitioners
- Combiners
- Compression
- Counters
- Zero Reduces
- Distributed File Cache
- Tool



Partitioners

- Partitioners 是一段刻画keys如何在reduces中分配的应用程序代码
- 缺省partitioning 将keys随机均匀分布
  - Uses *key.hashCode() % num_reduces*
- 通常需要Custom partitioning, 例如, to produce a total order in the output
  - 须实现*Partitioner* interface
  - 调用*conf.setPartitionerClass(MyPart.class)进行设置*
  - To get a total order, sample the map output keys and pick values to divide the keys into roughly equal buckets and use that in your partitioner

Combiners

- When *maps* produce many repeated keys
  - 通常用于跟在map后面做局部聚合
  - Done by specifying a *Combiner*
  - 目标是减小需要传输数据的大小
  - Combiners 与 Reduces 有相同的 interface 通常是同一个 class
  - Combiners 不能有副作用
  - In *WordCount*,
    `conf.setCombinerClass(Reduce.class);`

Compression

- 压缩outputs and intermediate data 通常能获得巨大性能收益
  - 通过配置文件或进行程序设计设置
  - Set *mapred.output.compress* to *true* 去压缩 job output
  - Set *mapred.compress.map.output* to *true* 去压缩 map outputs
- 压缩类型 *(mapred(.map)?.output.compression.type)*
  - “block” - Group of keys and values are compressed together
  - “record” - Each value is compressed individually
  - Block compression is almost always best
- 压缩代码 *(mapred(.map)?.output.compression.codec)*
  - Default (zlib) - slower, but more compression
  - LZO - faster, but less compression

Speculative execution

- The framework can run multiple instances of slow tasks
  - Output from instance that finishes first is used
  - Controlled by the configuration variable *mapred.speculative.execution*
  - Can dramatically bring in long tails on jobs

Zero Reduces

- Frequently, we only need to run a filter on the input data
  - No sorting or shuffling required by the job
  - Set the number of reduces to 0
  - Output from maps will go directly to OutputFormat and disk

Distributed File Cache

- 有时在本地计算机上需要一些只读的数据拷贝
  - 为每一个Mapper下载 1GB 的数据是很昂贵的
- 在JobConf中定义一个希望下载的文件列表
- 文件为每个机器下载一次
- 加到发射程序:

```
DistributedCache.addCacheFile(new
  URI("hdfs://nn:8020/foo"), conf);
```

- 加到task:

```
Path[] files = DistributedCache.getLocalCacheFiles(conf);
```

DistributedCache can be used to distribute simple, read-only data/text files and/or more complex types such as archives, jars etc

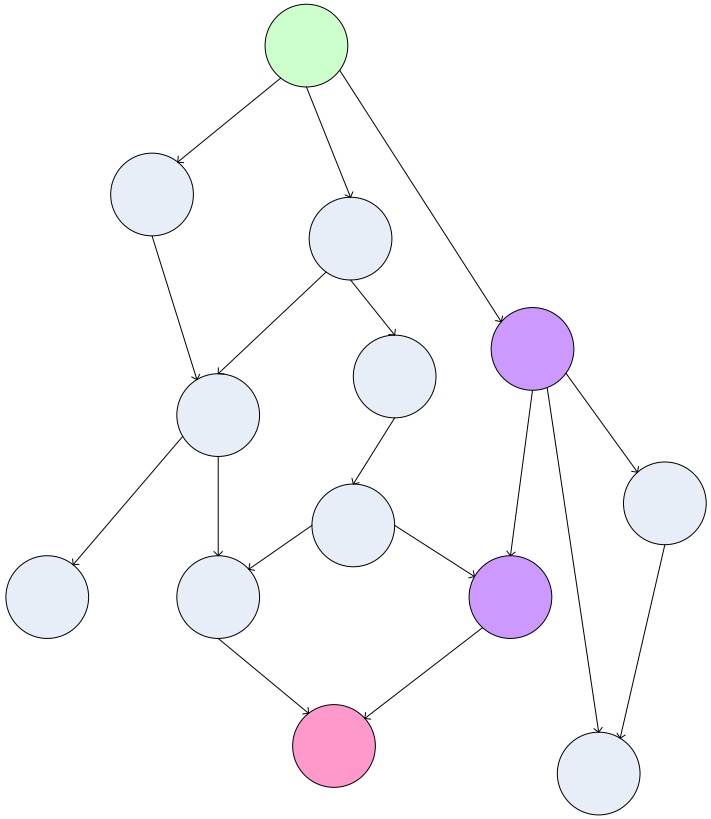
Tool

- 处理“标准”的Hadoop命令行选项
  - -conf file - load a configuration file named file
  - -D prop=value - define a single configuration property prop
  - Tool, is the standard for any Map-Reduce tool/application. The tool/application should delegate the handling of standard command-line options to ToolRunner.run(Tool, String[]) and only handle its custom arguments.
- Class looks like:

```
public class MyApp extends Configured implements Tool {
 public static void main(String[] args) throws Exception {
  System.exit(ToolRunner.run(new Configuration(),
             new MyApp(), args));
}
public int run(String[] args) throws Exception {
  …. getConf() ….
  }
}
```

最短路应用

- 一个常见的图应用是从一个起始节点到一个或多个目标节点的最短
- 在单机上通常用*Dijkstra's Algorithm*
- 我们可以通过MapReduce使用 BFS算法完成这个计算吗?



最短路应用: Intuition

- 我们可以归纳地定义这个问题的解决思路
  - DistanceTo(startNode) = 0
  - For all nodes ***n*** directly reachable from startNode, DistanceTo(n) = 1
  - For all nodes ***n*** reachable from some other set of nodes S,

    DistanceTo(n) = 1 + min(DistanceTo(m), $m \in S$)

从Intuition 到算法

- A map task receives a node ***n*** as a key, and (*D*, *points-to*) as its value
  - *D* 是n到起始点的距离
  - *points-to*是n的可达节点表
- ∀p ∈ points-to, emit (p, D+1)
- Reduces task gathers possible distances to a given *p* and selects the minimum one as D

一些观察

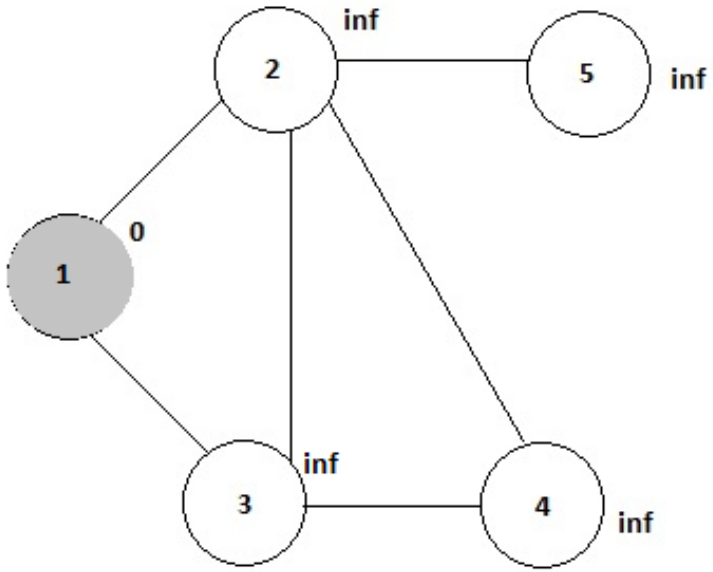
- This MapReduce task can advance the known frontier by one hop
- To perform the whole BFS, a non-MapReduce component then feeds the output of this step back into the MapReduce task for another iteration
  - Problem: Where'd the *points-to* list go?
  - Solution: Mapper emits (*n*, *points-to*) as well

算法的终止

- This algorithm starts from one node
- Subsequent iterations include many more nodes of the graph as the frontier advances
- 算法会终止吗?
  - Yes! Eventually, routes between nodes will stop being discovered and no better distances will be found. When distance is the same, we stop
  - Mapper should **emit** ***(n,D)*** to ensure that "current distance" is carried into the reducer

MapReduce实现BFS(1)

假设有一个无向图，如下：



每一条边的权重设为1
根据上图，输入数据为：

1<tab>2,3|**0**|**GRAY**|source
2<tab>1,3,4,5|**inf**|WHITE|null
3<tab>1,4,2|**inf**|WHITE|null
4<tab>2,3|**inf**|WHITE|null
5<tab>2|**inf**|WHITE|null

source<tab>adjacency_list|**distance_from_the_source**|color|parentNode

MapReduce实现BFS(2)

Intermediate output 1:

Reducer 1: (part-r-00000)

2<tab>1,3,4,5,|**1**|**GRAY**|1

5<tab>2,|inf|WHITE|null

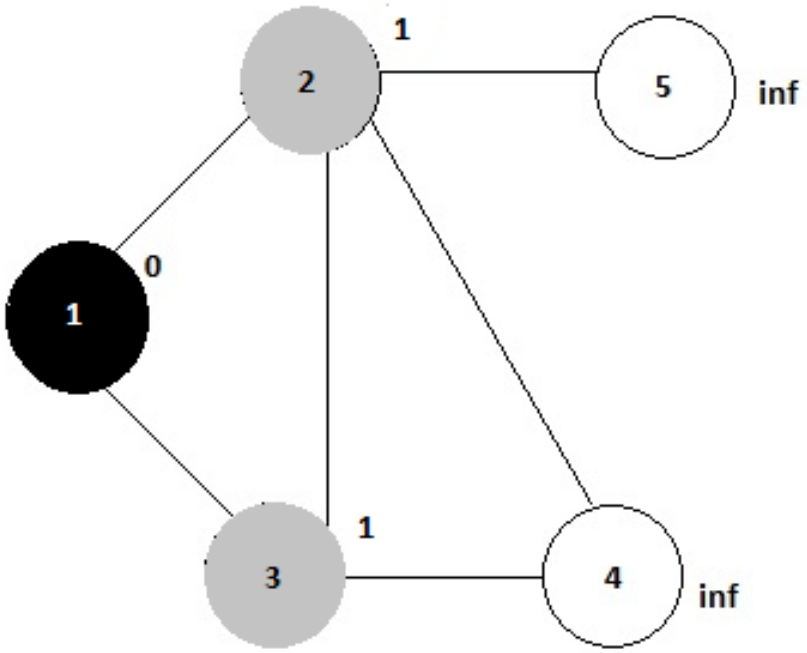
Reducer 2: (part-r-00001)

3<tab>1,4,2,|**1**|**GRAY**|1

Reducer 3: (part-r-00002)

1<tab>2,3,|0|**BLACK**|source

4<tab>2,3,|inf|WHITE|null



49

MapReduce实现BFS(3)

Intermediate output 2:

Reducer 1: (part-r-00000)

2<tab>1,3,4,5,|1|**BLACK**|1

5<tab>2,|2|**GRAY**|2

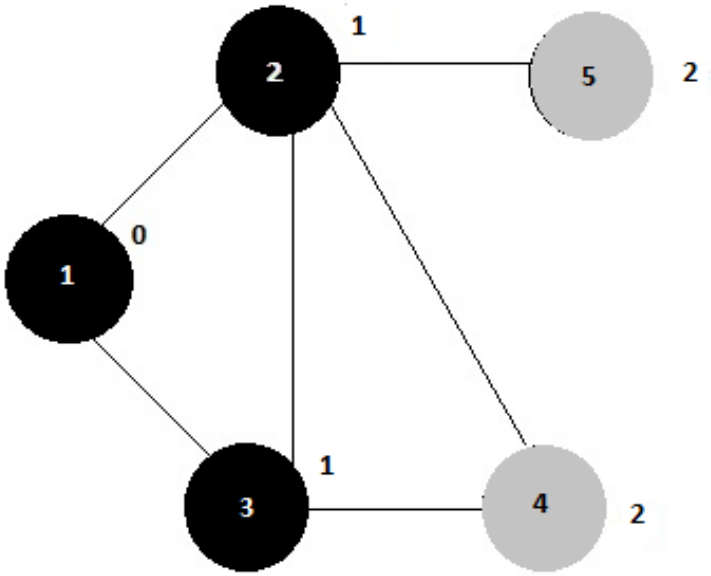
Reducer 2: (part-r-00001)

3<tab>1,4,2,|1|**BLACK**|1

Reducer 3: (part-r-00002)

1<tab>2,3,|0|**BLACK**|source

4<tab>2,3,|2|**GRAY**|2



MapReduce实现BFS(4)

Final output:

Reducer 1: (part-r-00000)

2<tab>1,3,4,5,|1|**BLACK**|1

5<tab>2,|2|**BLACK**|2

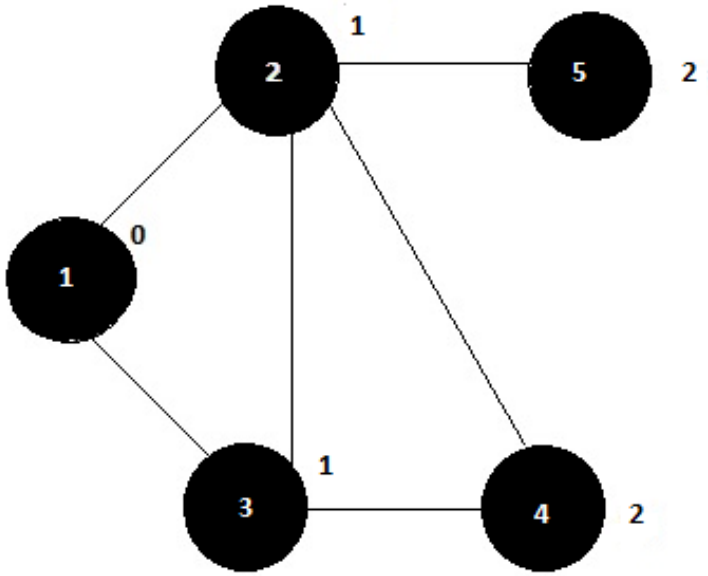
Reducer 2: (part-r-00001)

3<tab>1,4,2,|1|**BLACK**|1

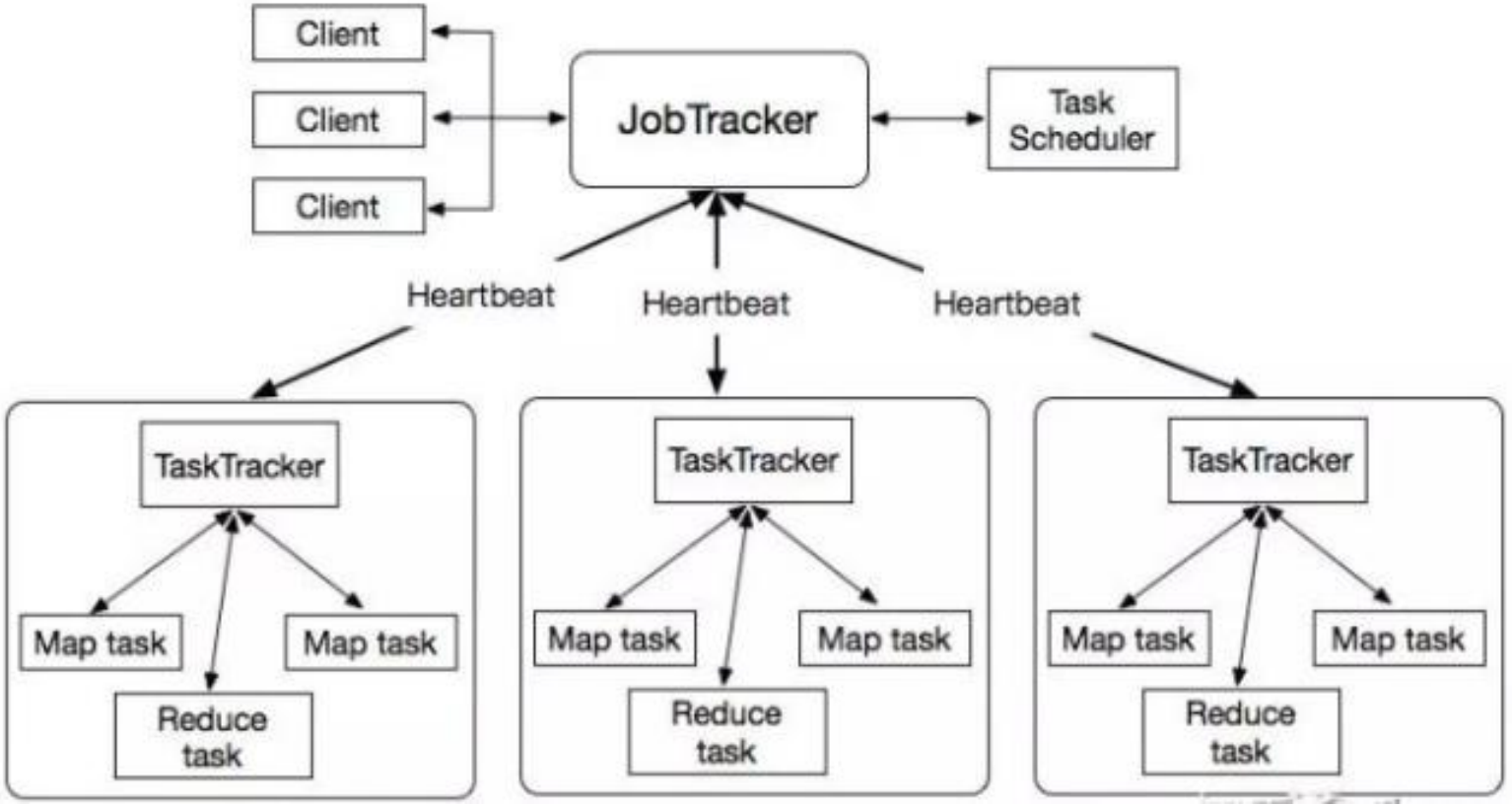
Reducer 3: (part-r-00002)

1<tab>2,3,|0|**BLACK**|source

4<tab>2,3,|2|**BLACK**|2



51


Client
Client
Client
JobTracker
Task Scheduler
Heartbeat
Heartbeat
Heartbeat
TaskTracker
Map task
Map task
Reduce task
TaskTracker
Map task
Map task
Reduce task
TaskTracker
Map task
Map task
Reduce task

当前MapReduce的局限

- 可扩展性
  - Maximum Cluster Size – 4000 Nodes
  - Maximum Concurrent Tasks – 40000
  - Coarse synchronization in Job Tracker
- 单点失效
  - Failure kills all queued and running jobs
  - Jobs need to be resubmitted by users
- 重启困难

YARN

YARN

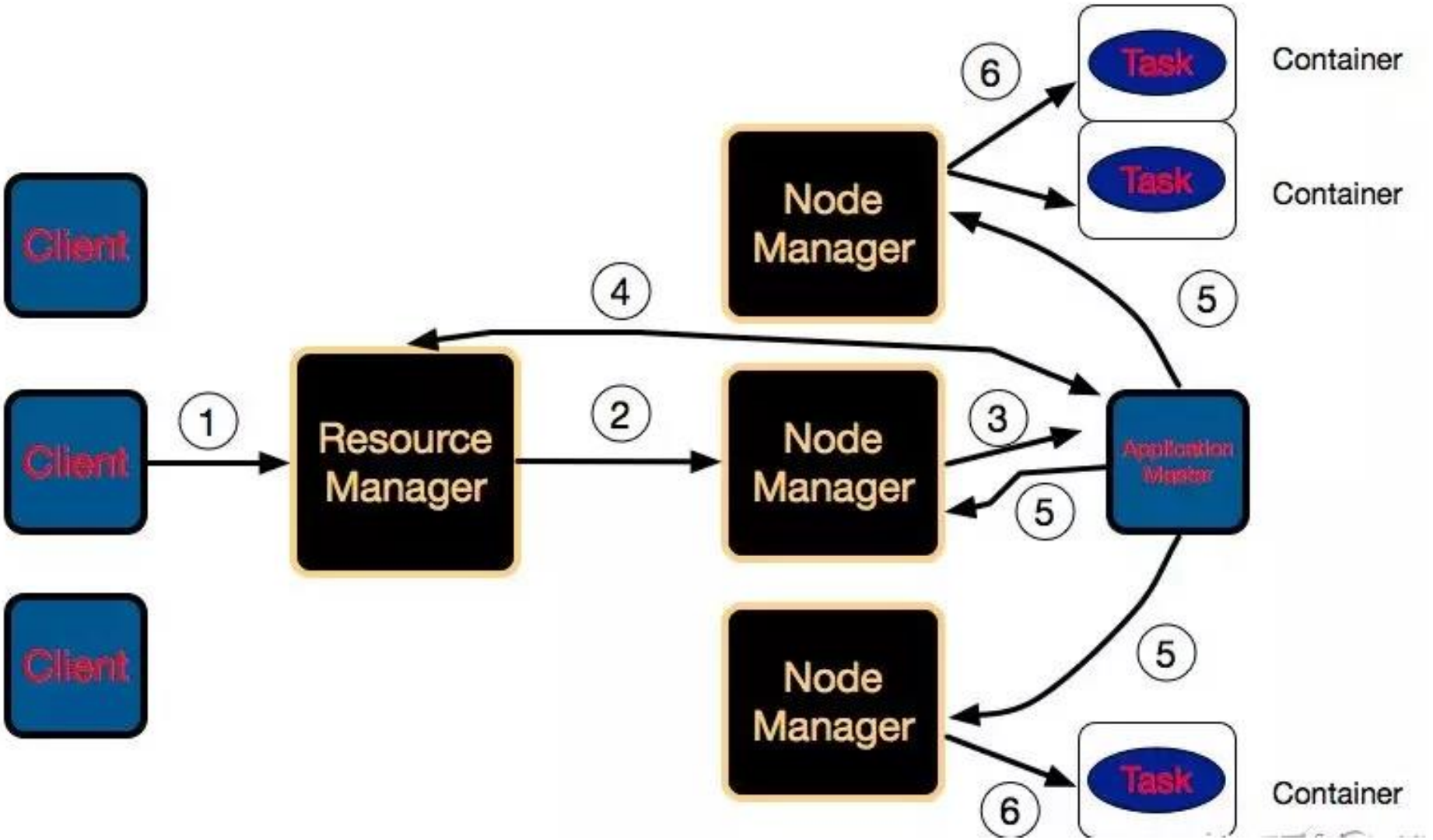
YARN基本设计思想是将原MapReduce架构中JobTracker的两个主要功能，即资源管理和作业调度/监控分成两个独立组件，即全局的ResourceManager和与每个应用相关的ApplicationMaster。

- 将JobTracker的两个主要功能分开
  - Global Resource Manager – 集群资源管理
  - Application Master – 作业调度与监控(one per application).
- Tasktracker
  - NodeManager (NM)
- YARN 维持了与现存 MapReduce 应用的兼容性.

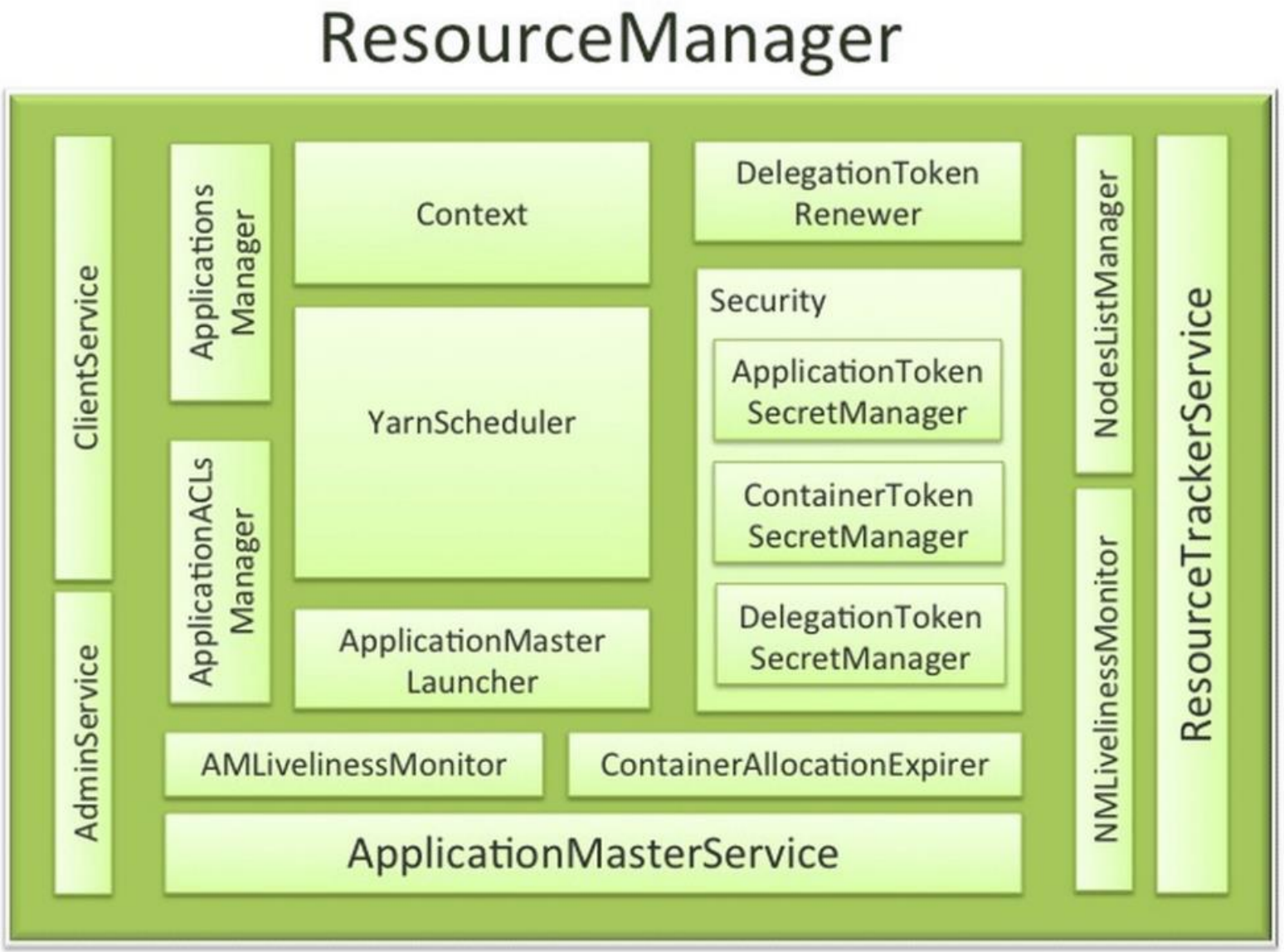
- ResourceManager：全局的资源管理者，整个集群只有唯一的一个，负责集群的资源统一管理和分配调度：
  ① 处理客户端的请求
  ② 启动监控 ApplicationMaster
  ③ 监控 NodeManager
  ④ 资源分配调度
- NodeManager：整个集群中有多个，负责自己本身节点的资源管理使用
  ① 定时向RM汇报本节点的资源情况
  ② 单个节点上的资源任务管理
  ③ 执行 NodeManager 的命令：启动 Container
  ④ 协助 ApplicationMaster处理任务
  ⑤ 管理着抽象的资源容器，这些容器代表着一个应用程序针对每个节点上的资源

- ApplicationMaster：管理应用程序在yarn上的每个实例：
  ① 每个应用程序对应一个：MR、Spark，负责应用程序的管理
  ② 为应用程序向RM申请资源（core、memory），分配给内部task
  ③ 需要与 NM 通信：启动/停止 task，task 是运行在 container 里面，AM 也是运行在 container 里面
- Container：YARN 中的资源抽象，封装了该节点上的多维度资源：
  ① 封装了 CPU、Memory 等资源的一个容器
  ② 是一个任务运行环境的抽象，且该任务只能使用该 container 中描述的资源

YARN的工作流


Client
Client
Client
1
Resource Manager
2
Node Manager
Node Manager
Node Manager
3
4
5
5
5
6
6
Application Master
Task
Task
Task
Container
Container
Container

YARN资源调度器



59

YARN资源调度器模型

- 在MapReducev1和YARN中，资源调度器的实现原理基本一致，不同的是YARN采用了事件驱动的编程模型和独特的资源表示模型

➢ 事件驱动模型

YARN的资源调度器采用事件驱动的编程模型。它实际上成为一个事件处理器，需要处理来自外部的6种类型的事件：

表2 资源调度器需要处理的事件

事件	事件含义	资源调度器进行的处理
NODE_REMOVED	集群中移除一个计算节点	从分配资源总量中移除相应的资源量
NODE_ADDED	集群中增加一个计算节点	将新增资源量添加到可分配资源总量中
APPLICATION_ADDED	RM收到一个新的AM	将该应用程序添加到相应的队列中
APPLICATION_REMOVED	一个应用程序运行结束	将该应用程序从相应的队列中清除
CONTAINER_EXPIRED	分配给AM的Container失效	对该Container进行回收再分配
NODE_UPDATE	RM收到NM的心跳信息	如果有Container得到释放，则会触发资源分配

➢ 资源表示模型

YARN同MRv1一样，采用动态资源分配机制。不同的是，YARN中资源的表示方式是容器(Container)，而不是槽位(Slot)。

Container是YARN中资源抽象，封装了某个节点上一定的资源。与资源量固定了slot 相比，Container资源量动态可变，更加灵活。

YARN资源调度器的机制

- 双层调度机制（核心）
  - ① 资源分配给应用程序（RMS）
  - ② 应用程序将资源分配给任务（AM）
- 层级队列管理机制

  队列组织，可嵌套，最小最大容量
- 资源保证与抢占机制
  - ① 资源保证(**增量**或一次性)
  - ② 资源抢占

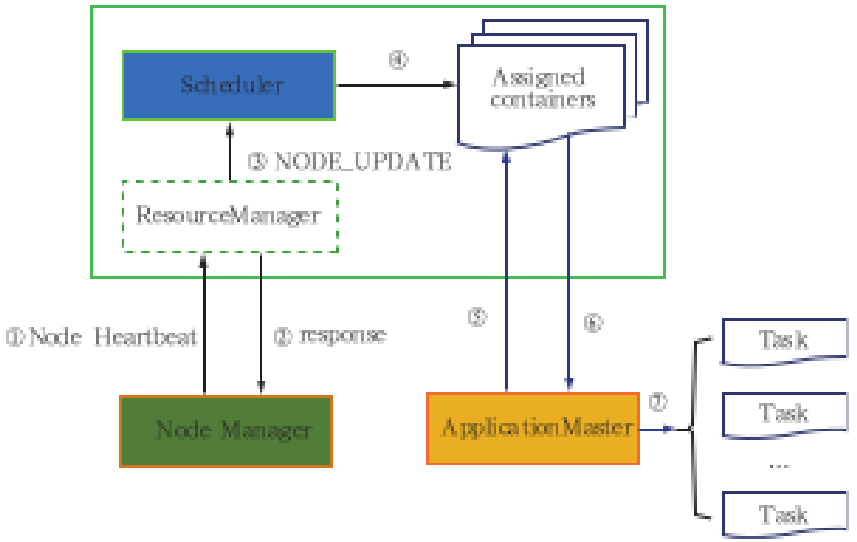
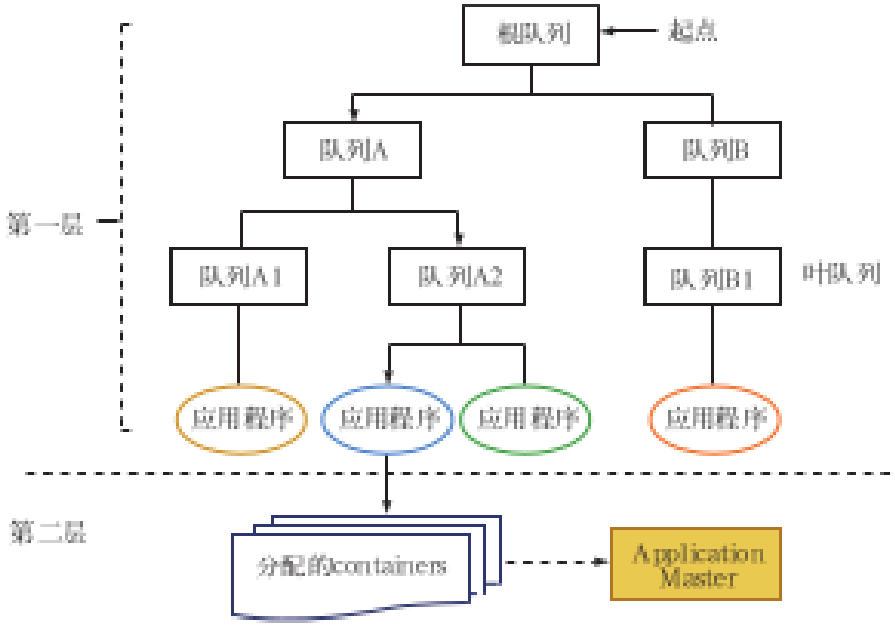


图4 YARN的资源分配过程



Google把资源调度器分为三代
第一代是独立的集群调度，
第二代是双层调度(Mesos,YARN)
第三代是共享状态调度(Omega)。
YARN是第二代： FIFO，Capacity
Fair三种调度器

YARN容量调度器（1）

- Yahoo开发的多用户调度器
- 以队列为单位划分资源，每个队列可设定资源最低保证和使用上限。
- 当队列的资源有剩余时，可暂时将剩余资源共享。
- Capacity Scheduler主要有以下特点：

✓ 容量保证
✓ 灵活性
✓ 多重租赁
✓ 安全保证
✓ 动态更新配置文件等。

YARN容量调度器（2）

- 启用容量调度器

  - 通过设置ResourceManager平台中的/etc/hadoop/conf/yarn-site.xml（HDP或CDH平台）文件属性来启用容量调度器。

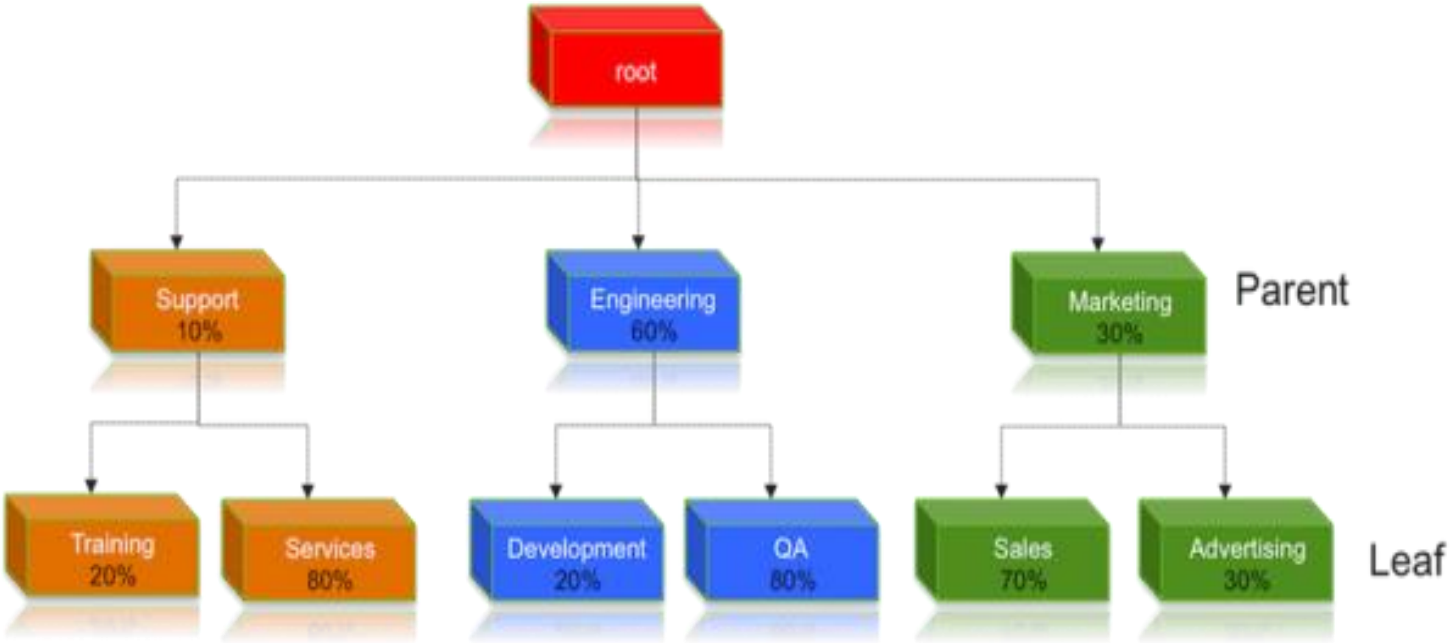
  - 容量调度器的设置包含在YARN RM主机的/etc/hadoop/conf/capacity-scheduler.xml中，容量调度器在启动时读取这个文件，或者在管理员修改这个文件后通过下面命令重新载入配置，管理员的权限配置由RM的yarn.admin.acl属性配置。

    yarn rmadmin -refreshQueues

YARN容量调度器（3）

- 设定队列

YARN中基础调度单元是队列，每一个队列的容量指定了提交到该队列的应用的可用集群资源百分比。队列可以用反映使用资源的不同组织、组和用户的数据库结构、资源需求和访问限制的层级来表示。



YARN容量调度器（4）

- 层级别列特征

- ✓ 队列分两种：叶子(leaf)队列和父(parent)队列
- ✓ 父队列用于组织层级之间的管理. 他们可以包含更多的父队列和叶子队列，他们自己不会直接接受应用的提交。
- ✓ 叶子队是居于父队列之下并接受应用。叶子队列不会包含其他子队列。
- ✓ 顶层的父队列root不属于任何组织，它代表了集群自身。
- ✓ 通过父队列和叶子队列，管理员可以为不同的组织层级指定容量分配

65

YARN容量调度器（5）

- 队列调度

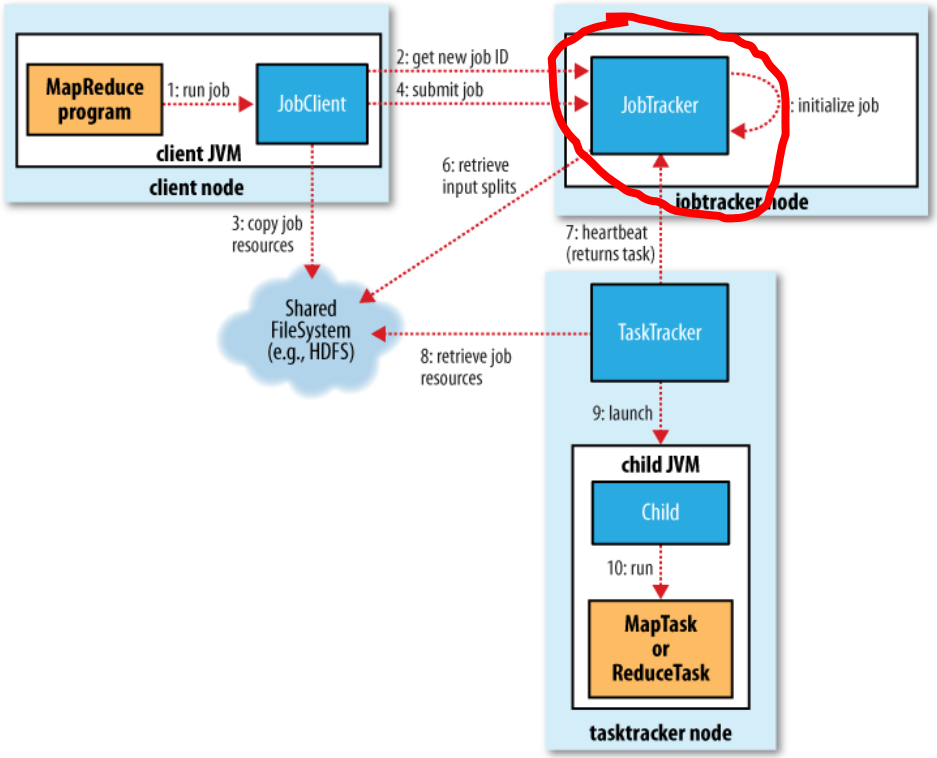
- 层级队列确保了资源会首先在组织子队列之间分享，然后才会将剩余的资源在其他组织之间分享。这会确保每一个组织可以控制资源的使用。
- 每一个层级、每一个父队列保证了它的子队列以一个按需的方式排序。队列的排序由每一个当前使用的队列容量决定（或者当两个队列的预留容量相同时由队列路径全名决定）
- root队列会知道集群容量如何在一级父队列之间分布式共享，并在它们的子队列中启用调度。
- 每一个父队列会将容量的限制应用到子队列之中。
- 叶子队列维持一个活跃应用的列表（可能来自多个用户)并以FIFO的方式调度资源，并受制于分配给用户的资源容量限制。

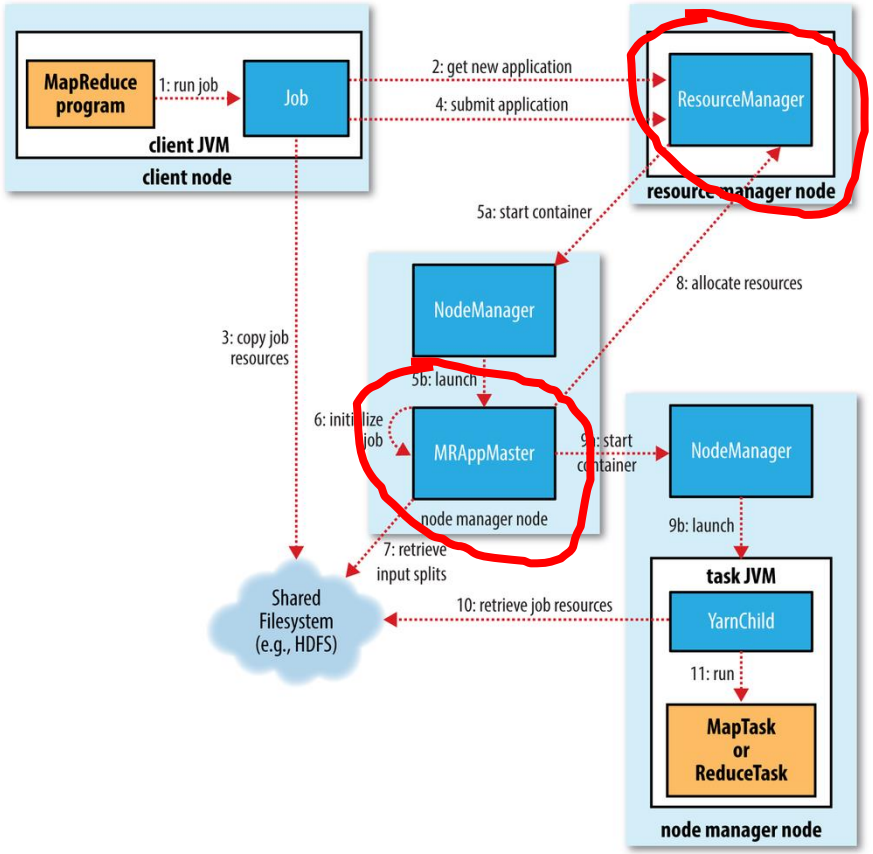
YARN容量调度器（6）

- 通过ACL来限制队列的访问
  - ✓ 访问控制列表用于限制用户和管理员来访问队列。应用只能提交到叶子队列上，但是ACL限制可以设在父队列上，并应用到它所有的子队列中。

- 通过队列管理集群的容量资源
  - ✓ 容量调度器被设计用来允许组织间以非常类似FIFO队列的方式来共享集群资源。
  - ✓ 调度是一个匹配资源需求的过程—来自多个用户的不同应用，提交到队列体系中的不同层级的不同队列之中，并使用集群节点中的可用容量资源进行分配。由于集群容量资源的总量会变化，容量属性值可以为百分数。

MapReduce和YARN对比





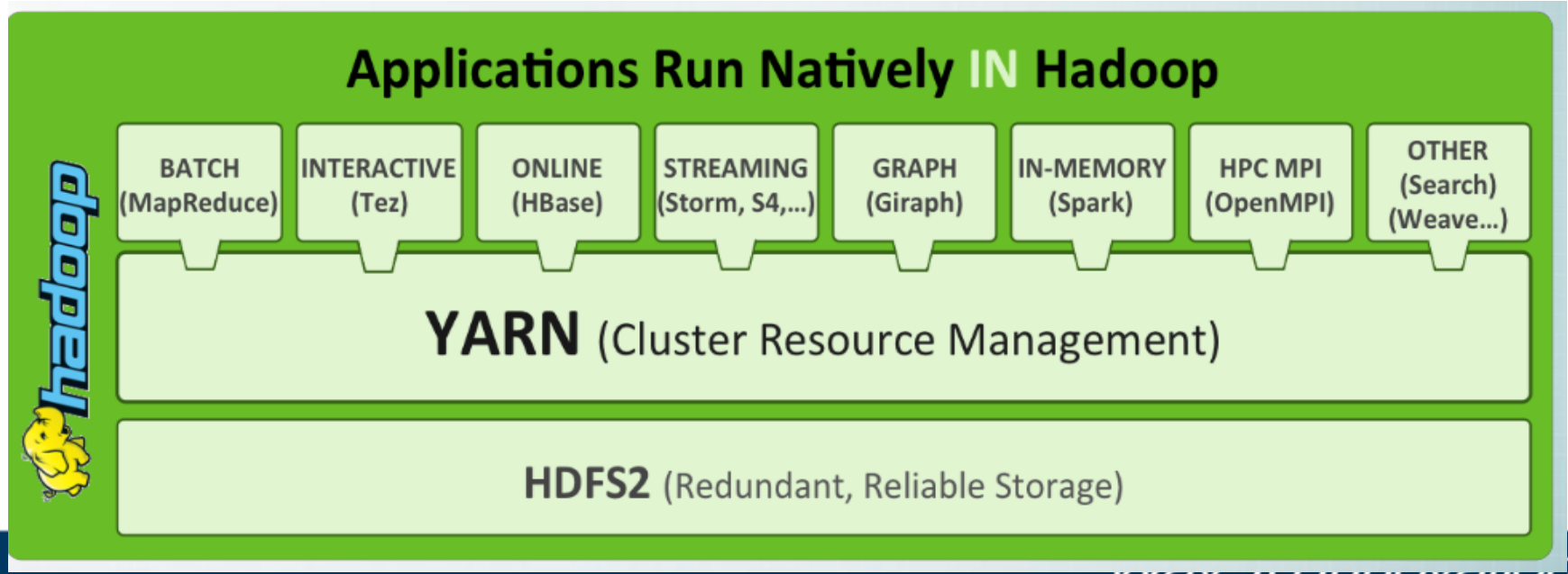
68

MapReduce和YARN对比

- Fault Tolerance and Availability
  - Resource Manager
    - No single point of failure – state saved in ZooKeeper
    - Application Masters are restarted automatically on RM restart
  - Application Master
    - Optional failover via application-specific checkpoint
    - MapReduce applications pick up where they left off via state saved in HDFS
- Wire Compatibility
  - Protocols are wire-compatible
  - Old clients can talk to new servers
  - Rolling upgrades

YARN平台支持

- **支持除MapReduce外的多个程序设计模式**
  - Tez – Generic framework to run a complex DAG
  - HBase on YARN (HOYA)
  - Machine Learning: Spark
  - Graph processing: Giraph
  - Real-time processing: Storm
  - *Run all on the same Hadoop cluster!*



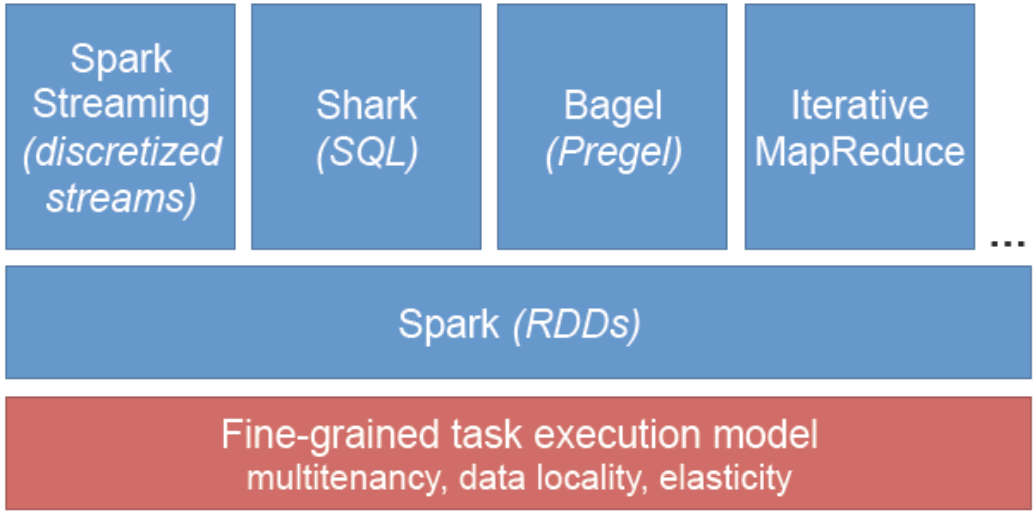
SPARK

Spark是什么

- Spark是UC Berkeley AMP lab所开源的类Hadoop MapReduce 的通用的并行计算框架;
- Spark 基于 map-reduce 算法实现的分布式计算，拥有 Hadoop MapReduce所具有的优点，并在**数据模型上做了扩展（支持迭代、交互、流式查询）**，核心思想是提供高效的数据共享抽象;
- Spark不同于MapReduce的是Job中间输出和结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更 好地适用于数据挖掘与机器学习等需要迭代的map reduce的算法。

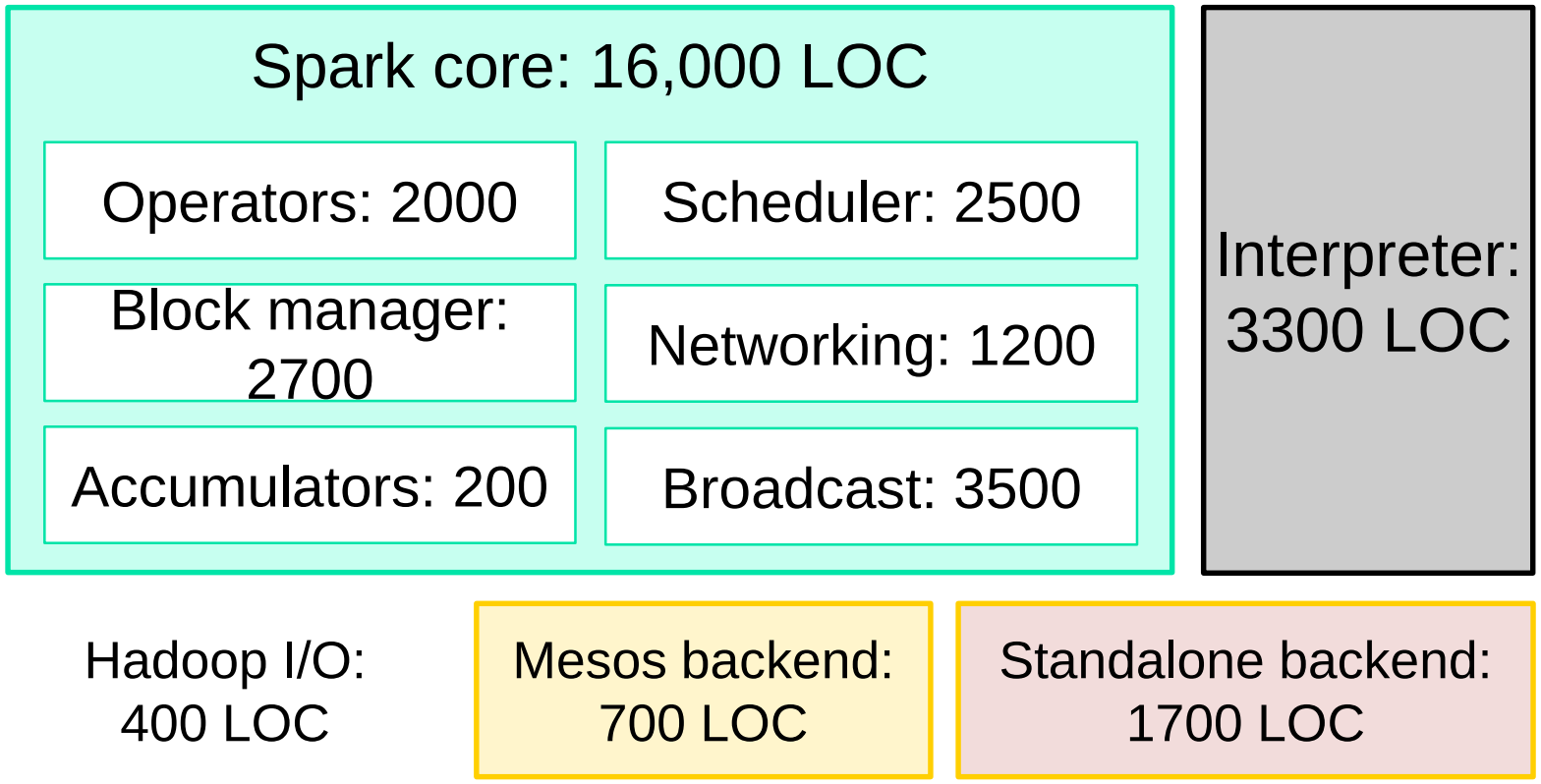
Project Goals

- Generality : diverse workloads, operators, job sizes
- Low latency : sub-second
- Fault tolerance : faults shouldn't be special case
- Simplicity: often comes from generality

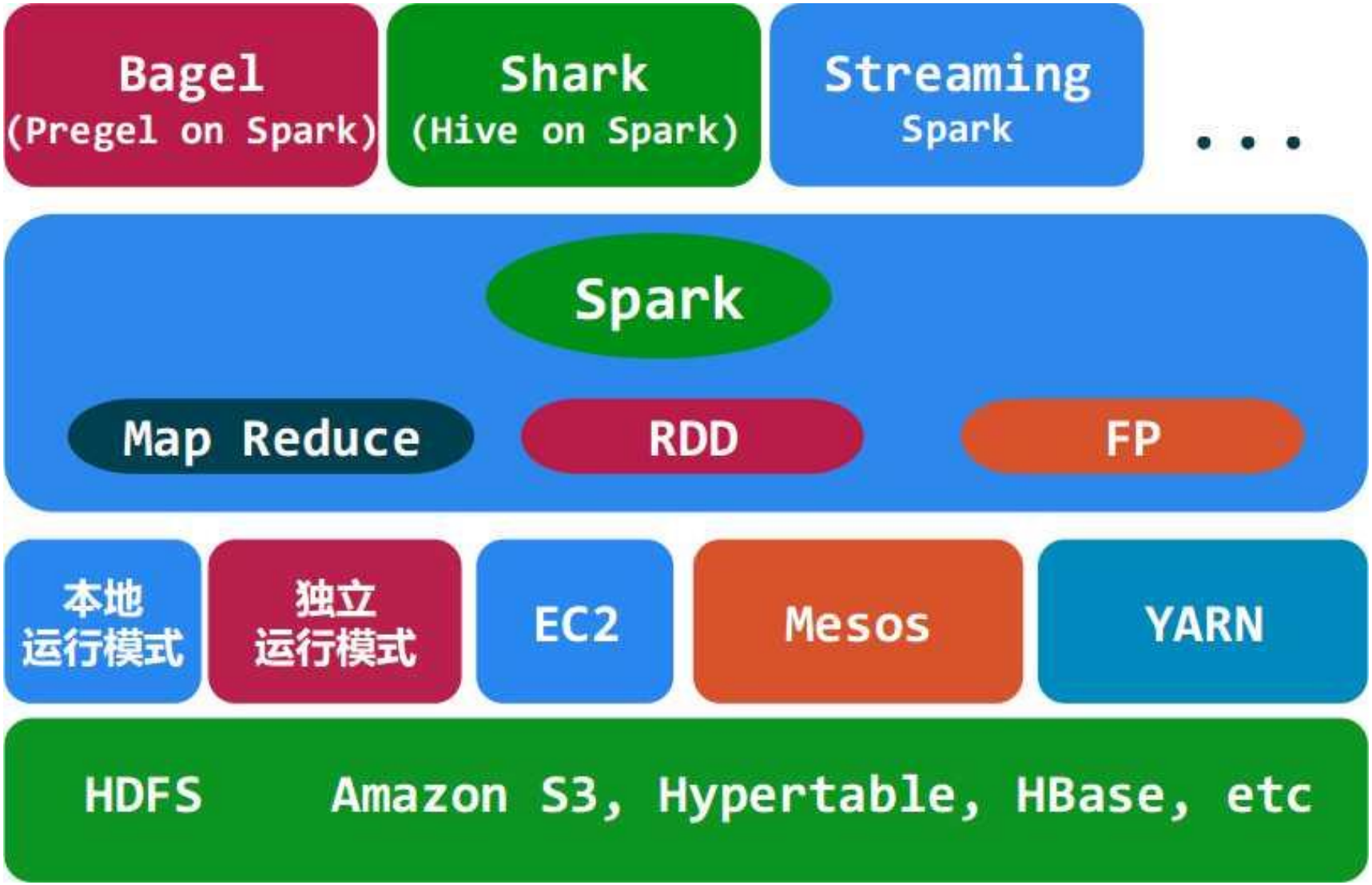


Spark: 20,000 LOC
Hadoop 1.0: 90,000 LOC
Hadoop 2.0: 220,000 LOC

Codebase Details



Spark生态系统


Bagel
(Pregel on Spark)
Shark
(Hive on Spark)
Streaming
Spark
. . .
Spark
Map Reduce
RDD
FP
本地
运行模式
独立
运行模式
EC2
Mesos
YARN
HDFS
Amazon S3, Hypertable, HBase, etc

Spark生态系统

- **Shark (Hive on Spark)**

  1） Shark基本上就是在Spark的框架基础上提供和Hive一样的HiveQL命令接口，Shark使用了Hive的API来实现Query Parsing和 Logic Plan Generation，

  2）最后的PhysicalPlan execution阶段用Spark代替Hadoop MapReduce。

  3）通过配置Shark参数，Shark可以自动在内存中缓存特定的RDD，实现数据重用，进而加快特定数据集的检索。

  4）Shark 通过UDF用户自定义函数实现特定的数据分析学习算法，使得SQL数据查询和运算分析能结合在一起，最大化RDD的重复使用。

- Spark streaming: 构建在Spark上处理Stream数据的框架，

  1）将Stream数据分成小的时间片断（几秒），以类似batch批量处理的方式来处理这小部 分数据。

  2）Spark Streaming构建在Spark上，一方面是因为Spark的低延迟执行引擎（100ms+）可以用于实时计算，另一方面相比基于Record的其它处理框架（如Storm），Spark数据集更容易做高效的容错处理。

  3）小批量处理的方式使得它可以同时兼容批量和实时数据处理的逻辑和算法。方便了一些需要历史数据和实时数据联合分析的特定应用场合。

Spark核心概念

Resilient Distributed Dataset (RDD)弹性分布数据集

- RDD是Spark的最基本抽象，是对分布式内存的抽象使用，实现了以操作本地数据集合的方式来操作分布式数据集。
- RDD是Spark最核心的概念，它表示已被分区，不可变的并能够被并行操作的数据集合，不同的数据集格式对应不同的RDD实现。RDD必须是可序列化的。
- RDD可以cache到内存中，每次对RDD数据集的操作之后的结果，都可以存放到内存中，下一个操作可以直接从内存中输入，省去了MapReduce大量的磁盘IO操作。

RDD特点与优势

- 它是在集群节点上的不可变的、已分区的集合对象。
- 通过并行转换的方式来创建如(map, filter, join, etc)。
- 失败自动重建。
- 可以控制存储级别（内存、磁盘等）来进行重用。
- 是静态类型的。

- RDD只能从持久存储或通过Transformations操作产生，对于丢失部分数据分区可根据它的lineage重新计算出来。
- RDD API是函数式的，强调*不变性*，在大部分场景下倾向于创建新对象而不是修改老对象。
- RDD的数据分区特性使其可通过数据的本地性来提高性能，这与Hadoop MapReduce是一样的。
- RDD是可序列化的，在内存不足时可自动降级为磁盘存储。

RDD的生成

RDD有两种创建方式：

- 从Hadoop文件系统（或与Hadoop兼容的其它存储系统）输入（例如HDFS）创建。
- 从父RDD转换得到新RDD。

下面来看一从Hadoop文件系统生成RDD的方式，如：val file = spark.textFile("hdfs://...")，file变量就是RDD（实际是HadoopRDD实例）RDD的转换与操作

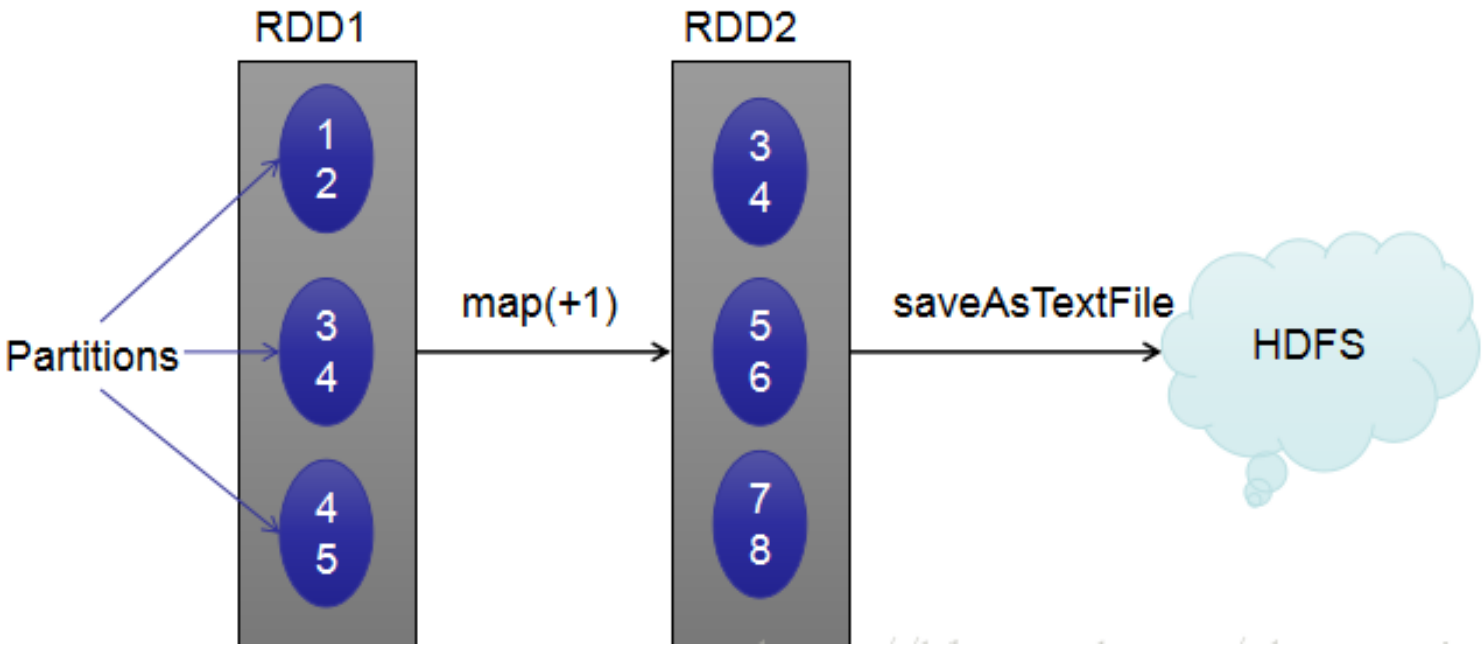
RDD有两种计算方式：

- 转换操作(Transformations) (如：map, filter, groupBy, join等)是Lazy的，也就是说从一个RDD转换生成另一个RDD的操作不是马上执行，Spark在遇到转换操作时只会记录需要这样的操作，并不会去执行，需要等到有Actions操作的时候才会真正启动计算过程进行计算。
- 操作(Actions) (如：count, collect, save等)，Actions操作会返回结果或把RDD数据写到存储系统中。Actions是触发Spark启动计算的动因。

主要的Transformation和Acion如下表所示：

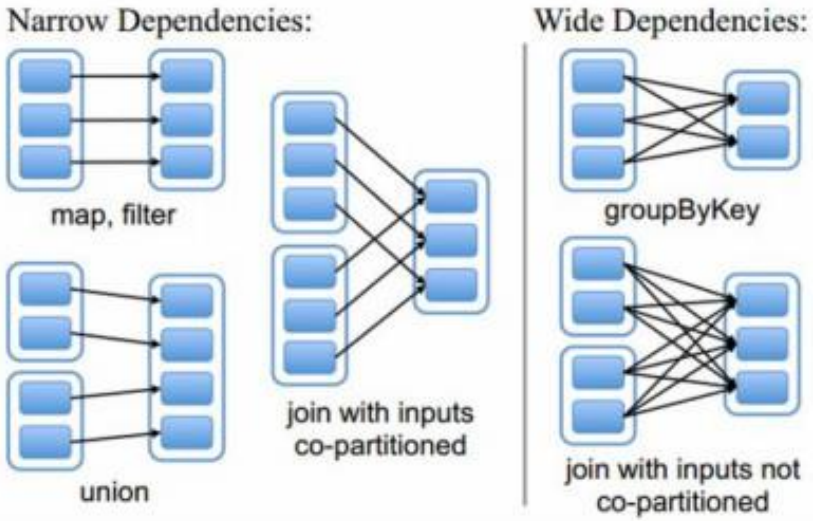
Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

作用在RDD上的操作（算子）



RDD依赖关系

不同的操作依据其特性，可能会产生不同的依赖，RDD之间的依赖关系有以下两种：



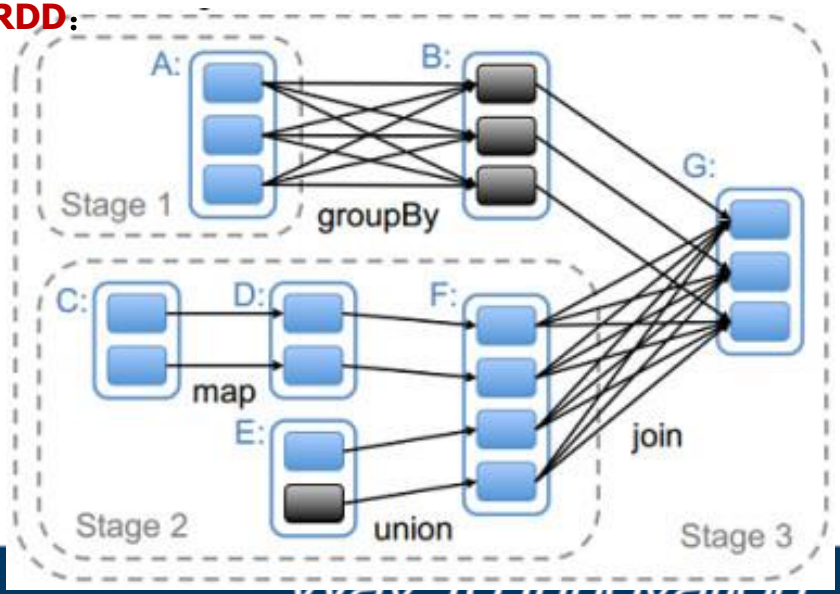
➢窄依赖(Narrow Dependencies)

一个父RDD分区最多被一个子RDD分区引用，表现为一个父RDD的分区；对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区，如map、filter、union等操作则产生窄依赖；

➢宽依赖(Wide Dependencies)

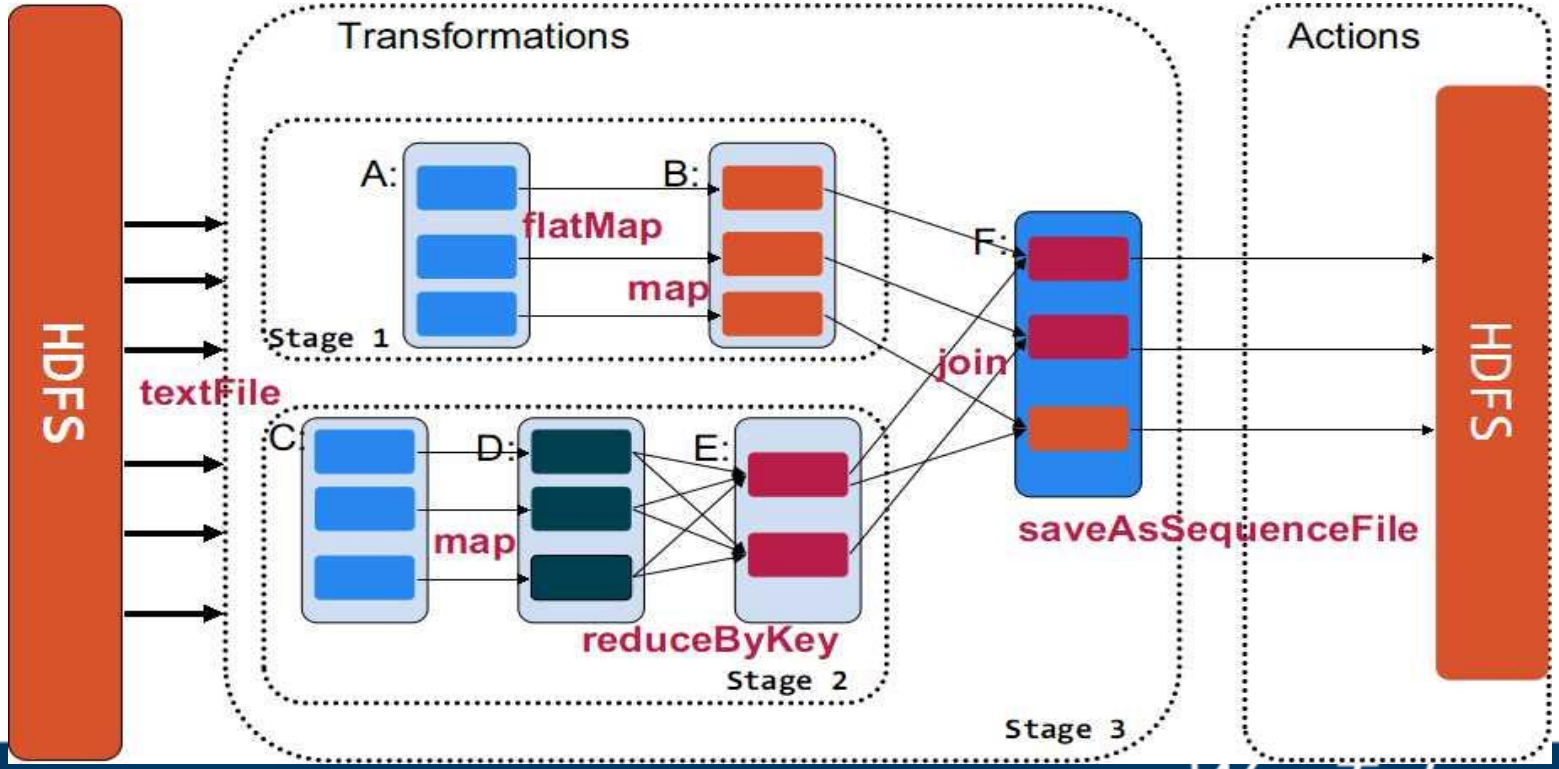
一个子RDD的分区依赖于父RDD的多个分区或所有分区，也就是说存在一个父RDD的一个分区对应一个子RDD的多个分区，如groupByKey等操作则产生宽依赖操作；

蓝色实心方框代表一个**partition**，蓝边矩形框代表一个**RDD**：



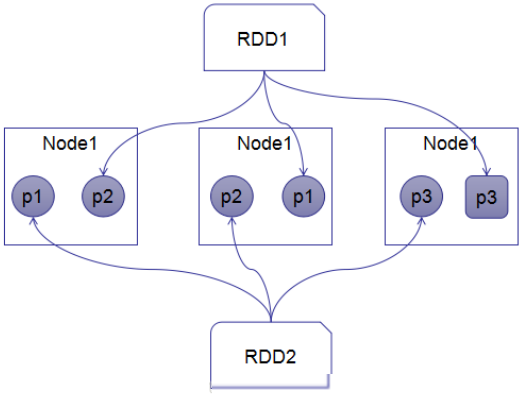
RDD Stage

- spark提交Job之后会把Job生成多个Stage，多个Stage之间是有依赖的，Stage之间的依赖关系就构成了DAG（有向无环图）。
- 对于窄依赖，Spark会尽量多地将RDD转换放在同一个Stage中；而对于宽依赖，但大多数时候是shuffle操作，Spark通常将shuffle操作定义为stage的边界。



RDD数据存储管理

- RDD可以被抽象地理解为一个大的数组（Array），但是这个数组是分布在集群上的。逻辑上RDD的每个分区叫一个Partition。
- 在 Spark的执行过程中，RDD经历一个个的Transfomation算子之后，最后通过Action算子进行触发操作。



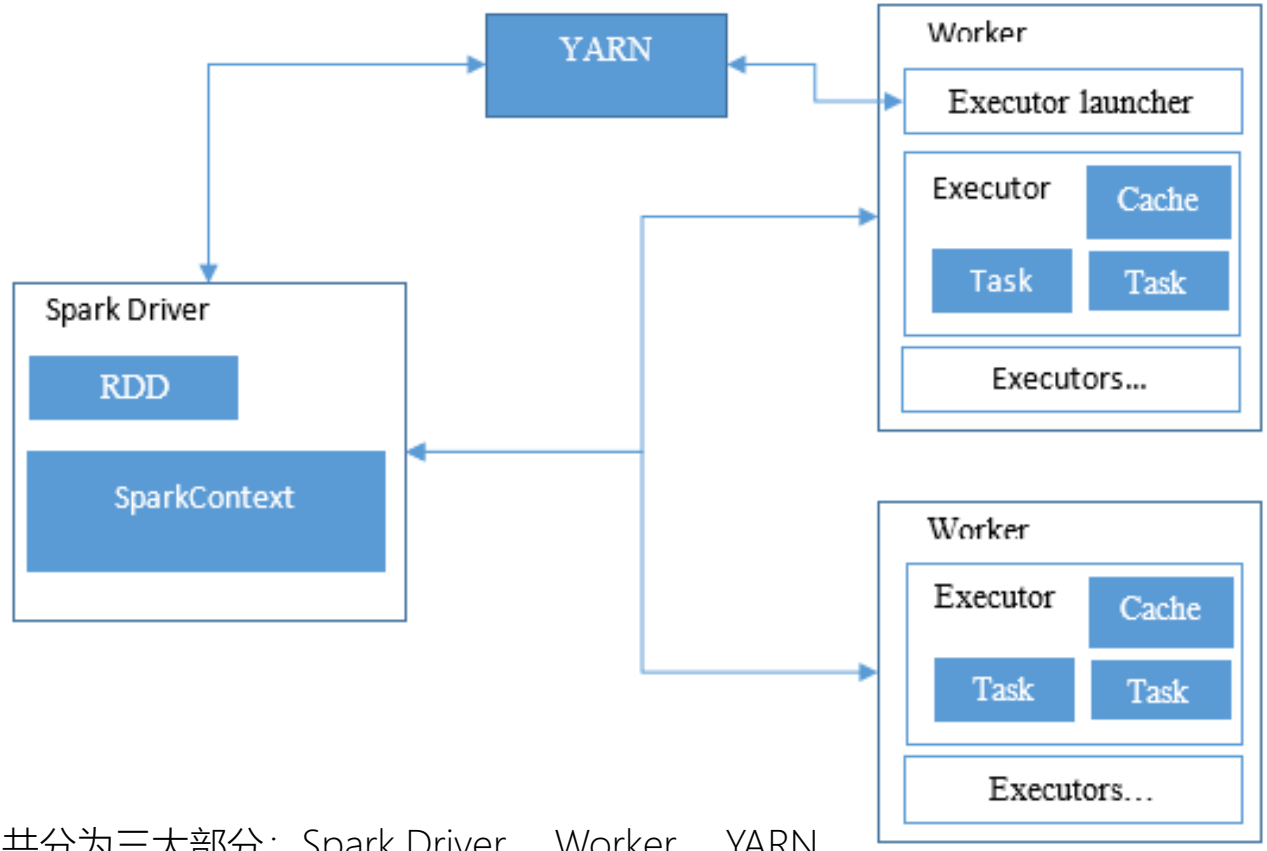
- 逻辑上每经历一次变换，就会将RDD转换为一个新的RDD，RDD之间通过Lineage产生依赖关系。变换的输入和输出都是RDD。
- RDD会被划分成很多的分区分布到集群的多个节点中。分区是个逻辑概念，变换前后的新旧分区在物理上可能是同一块内存存储。这是很重要的优化，以防止函数式数据不变性（immutable）导致的内存需求无限扩张。
- 有些RDD是计算的中间结果，其分区并不一定有相应的内存或磁盘数据与之对应，如果要迭代使用数据，可以调cache()函数缓存数据。

RDD数据存储管理

- 在物理上，RDD对象实质上是一个元数据结构，存储着Block、 Node等的映射关系，以及其他的元数据信息。
- 一个RDD是一组分区，在物理数据存储上，RDD的每个分区对应的就是一个Block，Block可以存储在内存，当内存不够时可以存储到磁盘上。每个Block中存储着RDD所有数据项的一个子集，暴露给用户的可以是一个Block的迭代器，也可以就是一个数据项（例如，通过map函数对每个数据项并行计算）。
- 如果是从HDFS等外部存储作为输入数据源，数据按照HDFS中的数据分布策略进行数据分区，HDFS中的一个Block对应Spark的一个分区。
- Spark支持重分区，数据通过Spark默认的或者用户自定义的分区器决定数据块分布在哪些节点。例如，支持Hash分区（按照数据项的Key值取Hash值，Hash值相同的元素放入同一个分区之内）和Range分区（将属于同一数据范围的数据放入同一分区）等分区策略。

Spark系统结构

Spark自身只对计算负责，其计算资源的管理和调度由第三方框架来实现。常用的框架有YARN和Mesos，Spark on YARN的系统结构图：



共分为三大部分：Spark Driver， Worker， YARN 。其中Driver program负责将RDD转换为任务，并进行任务调度。Worker负责任务的执行。YARN负责计算资源的维护和分配。

从RDD到输出结果的整个Spark计算过程：

1.RDD构建：构建RDD之间的依赖关系，将RDD转换为阶段的有向无环图。

2.任务调度：根据空闲计算资源情况进行任务提交，并对任务的运行状态进行监测和处理。

3.任务计算：搭建任务运行环境，执行任务并返回任务结果。

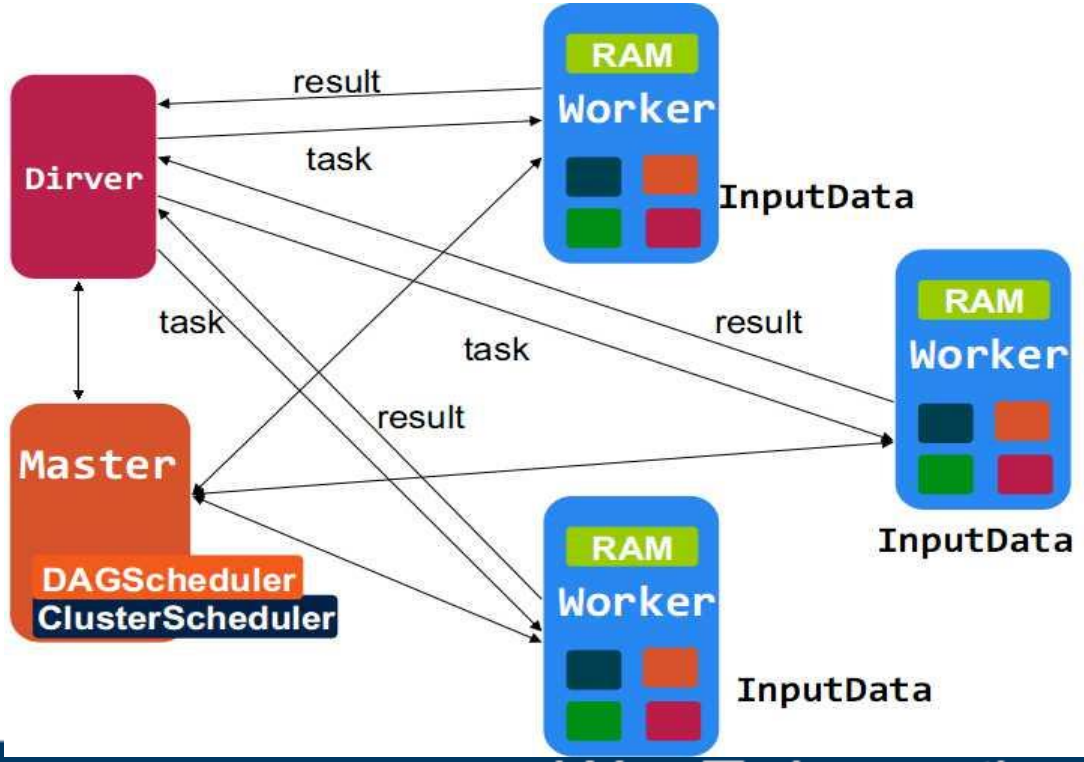
4.Shuffle过程：两个阶段之间有宽依赖时，需要进行Shuffle操作。

5.计算结果收集：从每个任务收集并汇总结果。

编程接口

- Spark通过与编程语言集成的方式包装RDD的操作，每个数据集都表示为RDD对象，对数据集的操作就表示成对RDD对象的操作。
- Spark和Hadoop MapReduce类似，由Master(类似于MapReduce的Jobtracker)和Workers(Spark的Slave工作节点)组成。用户编写的Spark程序被称为Driver程序，Dirver程序会连接master并定义了对各RDD的转换与操作.

- Workers存储着数据分块和享有集群内存，是运行在工作节点上的守护进程，当它收到对RDD的操作时，根据数据分片信息进行本地化数据操作，生成新的数据分片、返回结果或把RDD写入存储系统。



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

object WordCount {
  def main(args: Array[String]) {
    val conf = new SparkConf()
    conf.setAppName("WordCount")
    val sc = new SparkContext(conf)

    val file = "hdfs://127.0.0.1:9000/file.txt"
    val lines = sc.textFile(file)
    val words = lines.flatMap(_.split("\\s+"))
    val wordCount = words.countByValue()

    println(wordCount)
  }
}
```

运行

要运行Spark任务，首先要把代码打成JAR包。打包后，

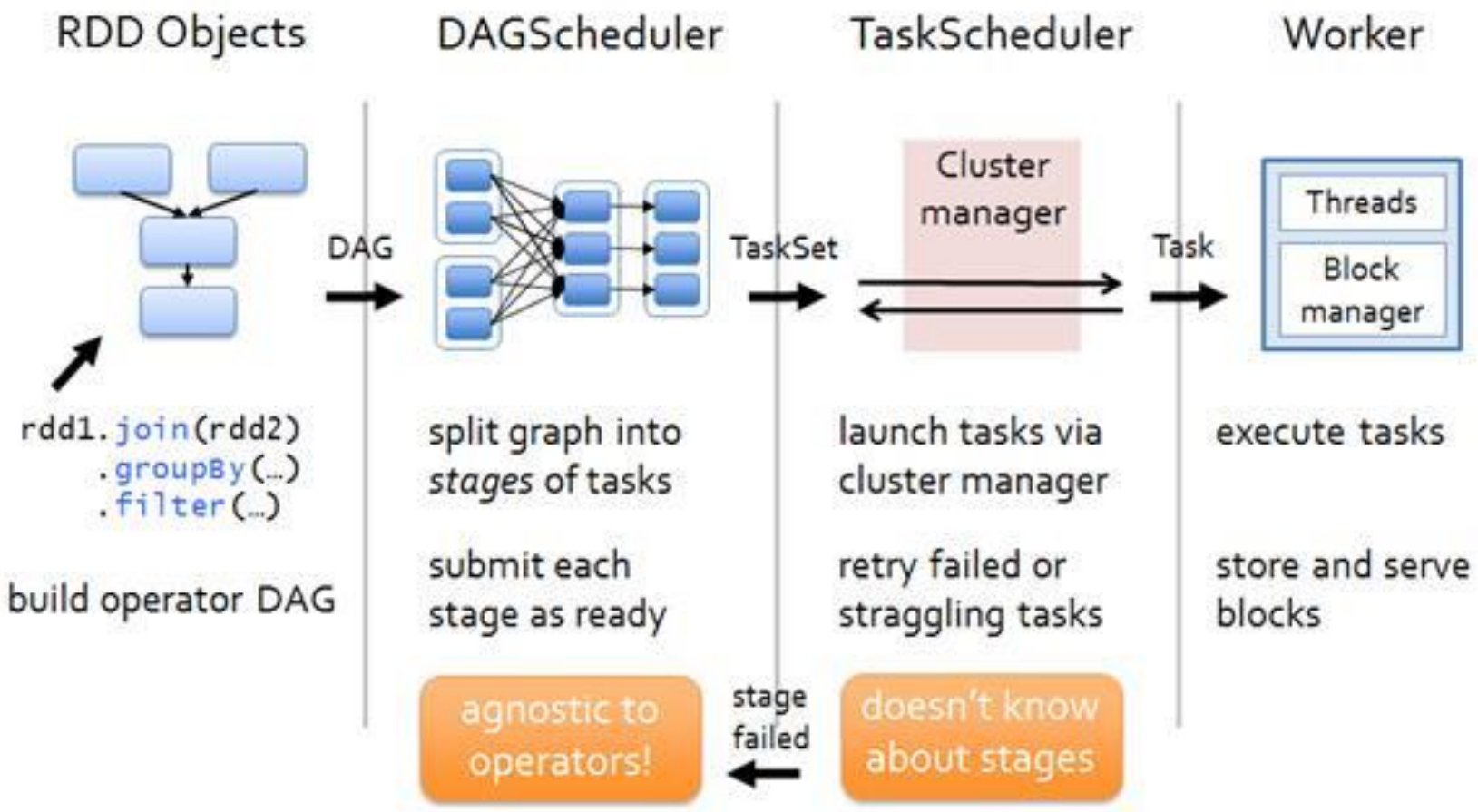
spark-submit --class "demo.WordCount" SparkDemo-1.0-SNAPSHOT.jar

其中demo.WordCount是main函数所在的object，而SparkDemo-1.0-SNAPSHOT.jar就是打出来的jar包。

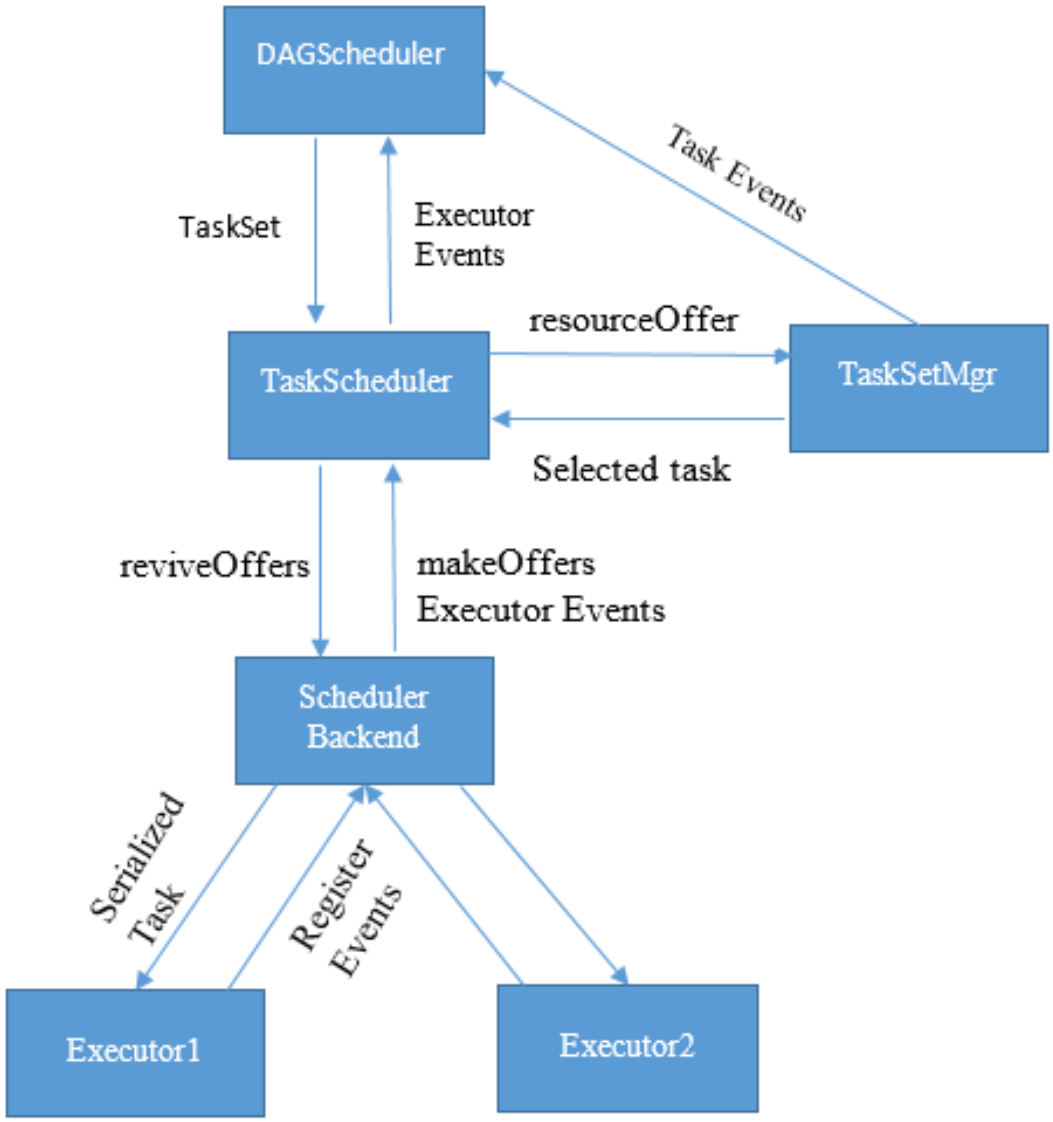
在这里我们用一个简洁的CharCount程序为例，这个程序把含有a-z字符的列表转化为RDD，对此RDD进行了Map和Reduce操作计算每个字母的频数，最后将结果收集。其代码如下：

```scala
package org.apache.spark

import scala.math.random

 * @author machong
object SparkCharCount {
  def main(args: Array[String]) {

    var partitionSize = 2;

    val conf = new SparkConf().setAppName("Spark CharCount")
    conf.setMaster("local[1]")
    val spark = new SparkContext(conf)

    val rdd = spark.parallelize(data, partitionSize).map{c =>
        (c, 1)
    };
    val rdd1 = new PairRDDFunctions(rdd).reduceByKey(_ + _)

    rdd1.collect

    spark.stop()
  }

  def data = {
    val dataSize = 100;

    val charTable = "abcdefghijklmnopqrstuvwxyz";
    val data = MutableList[String]()

    for (i <- 1 to dataSize) {
      val idx = (Math.random() * charTable.length).asInstanceOf[Int]
      data += (charTable.charAt(idx).toString)
    }
    data
  }
}
```

Spark工作流程

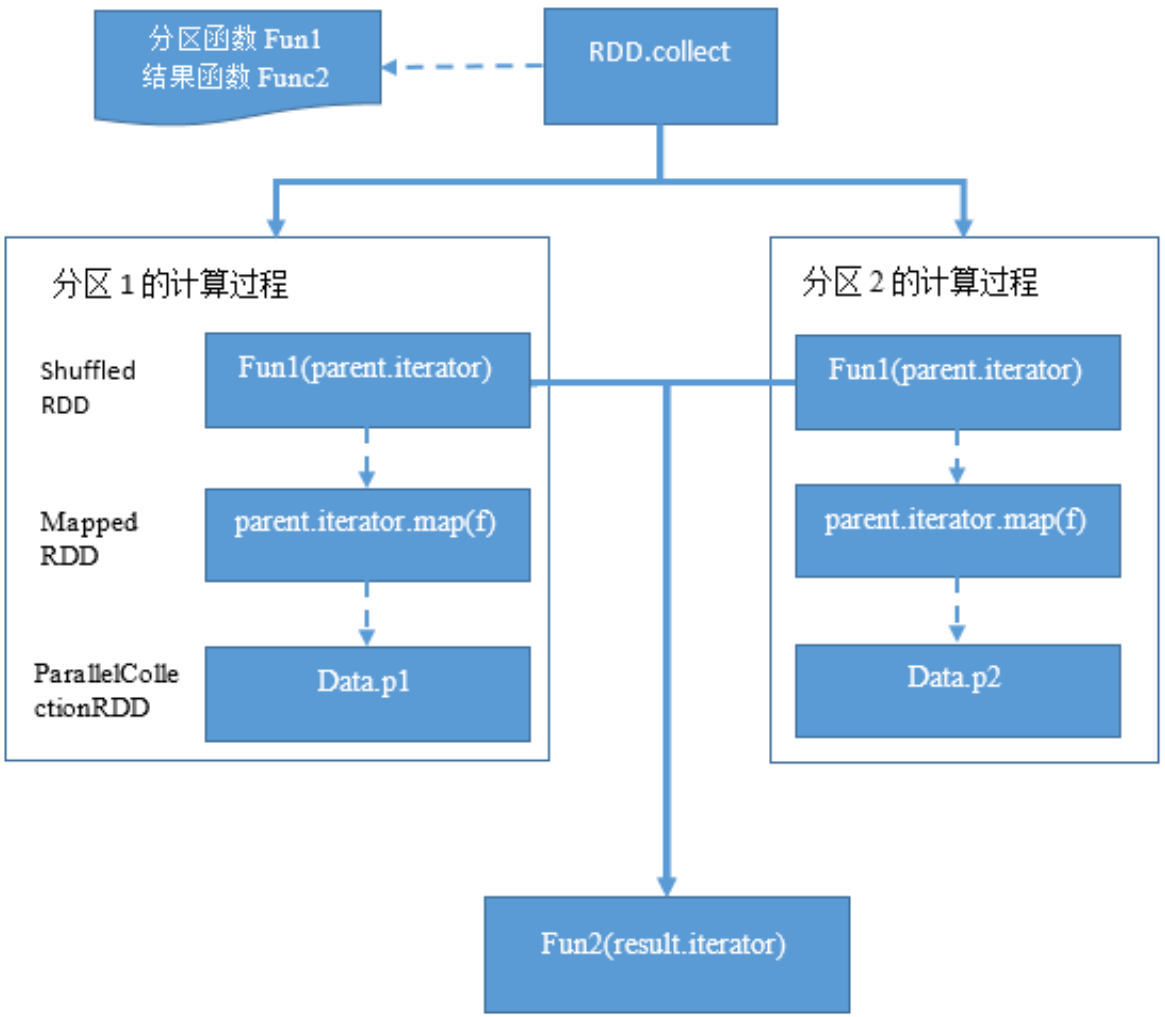

RDD Objects
DAGScheduler
TaskScheduler
Worker
DAG
TaskSet
Task
Cluster manager
Threads
Block manager
rdd1.join(rdd2)
.groupBy(…)
.filter(…)
build operator DAG
split graph into stages of tasks
submit each stage as ready
agnostic to operators!
launch tasks via cluster manager
retry failed or straggling tasks
doesn't know about stages
stage failed
execute tasks
store and serve blocks

任务调度



1. DAGScheduler负责总体的任务调度，
2. SchedulerBackend负责与Executors通信，维护计算资源信息，并负责将任务序列化并提交到Executor。
3. TaskSetManager负责对一个阶段的任务进行管理，其中会根据任务的数据本地性选择优先提交的任务。
4. TaskScheduler负责对TaskSet进行调度，通过调度策略确定TaskSet优先级。同时是一个中介者，其将DAGScheduler，SchedulerBackend和TaskSetManager联结起来，对Executor和Task的相关事件进行转发。

任务计算



任务的计算过程是在Executor上完成的，Executor监听来自SchedulerBackend的指令，接收到任务时会启动TaskRunner线程进行任务执行。

在TaskRunner中首先将任务和相关信息反序列化，然后根据相关信息获取任务所依赖的Jar包和所需文件，完成准备工作后执行任务的run方法，实际上就是执行ShuffleMapTask或ResultTask的run方法。

任务执行完毕后将结果发送给Driver进行处理。

Spark与Hadoop的对比

- Spark的中间数据放到内存中，对于迭代运算效率更高。
- Spark比Hadoop有更丰富的操作类型，因此更通用。

  1）Transformations 操作类型:

  map, filter, flatMap, sample, groupByKey, reduceByKey, union, join, cogroup, mapValues, sort,partionBy等多种操作类型，

  2）Actions 操作类型：

  Count, collect, reduce, lookup, save等多种actions操作。
- 各个处理节点之间的通信模型不再像Hadoop那样就是唯一的Data Shuffle一种模式。用户可以命名，物化，控制中间结果的存储、分区等，编程模型比Hadoop更灵活。
- 容错性。

  在分布式数据集计算时通过checkpoint来实现容错，而checkpoint有两种方式：

  1）checkpoint data

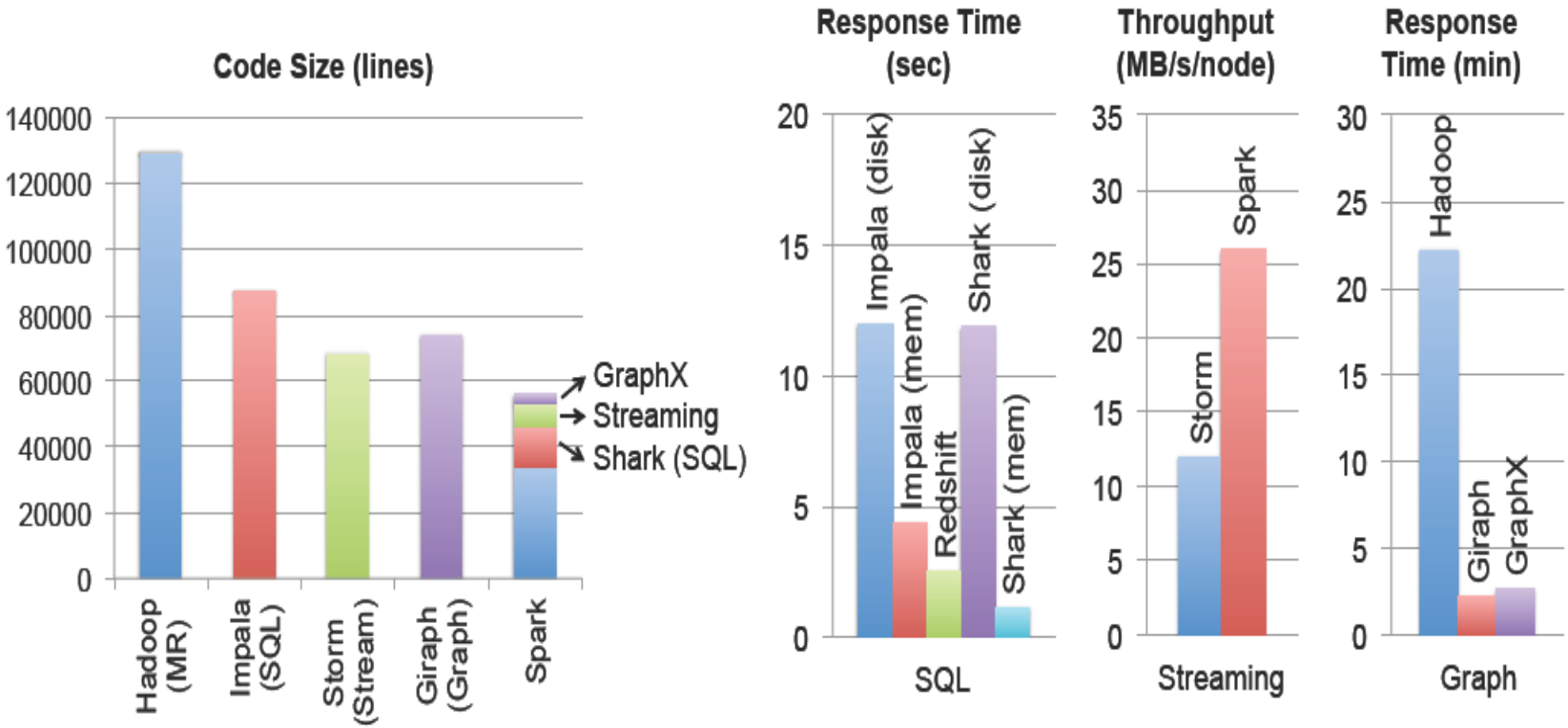
  2）logging the updates。

  用户可以控制采用哪种方式来实现容错。
- 可用性。

  Spark通过提供丰富的Scala, Java，Python API及交互式Shell来提高可用性。

Spark横向比较

和现有系统的比较



综合于一身

多功能

Spark的适用场景

- Spark是基于内存的迭代计算框架，适用于需要多次操作特定数据集的应用场合。需要反复操作的次数越多，所需读取的数据量越大，受益越大，数据量小但是计算密集度较大的场合，受益就相对较小。
- 由于RDD的特性，Spark不适用那种异步细粒度更新状态的应用，例如web服务的存储或者是增量的web爬虫和索引。就是对于那种增量修改的应用模型不适合。
- 总的来说Spark的适用面比较广泛且比较通用。