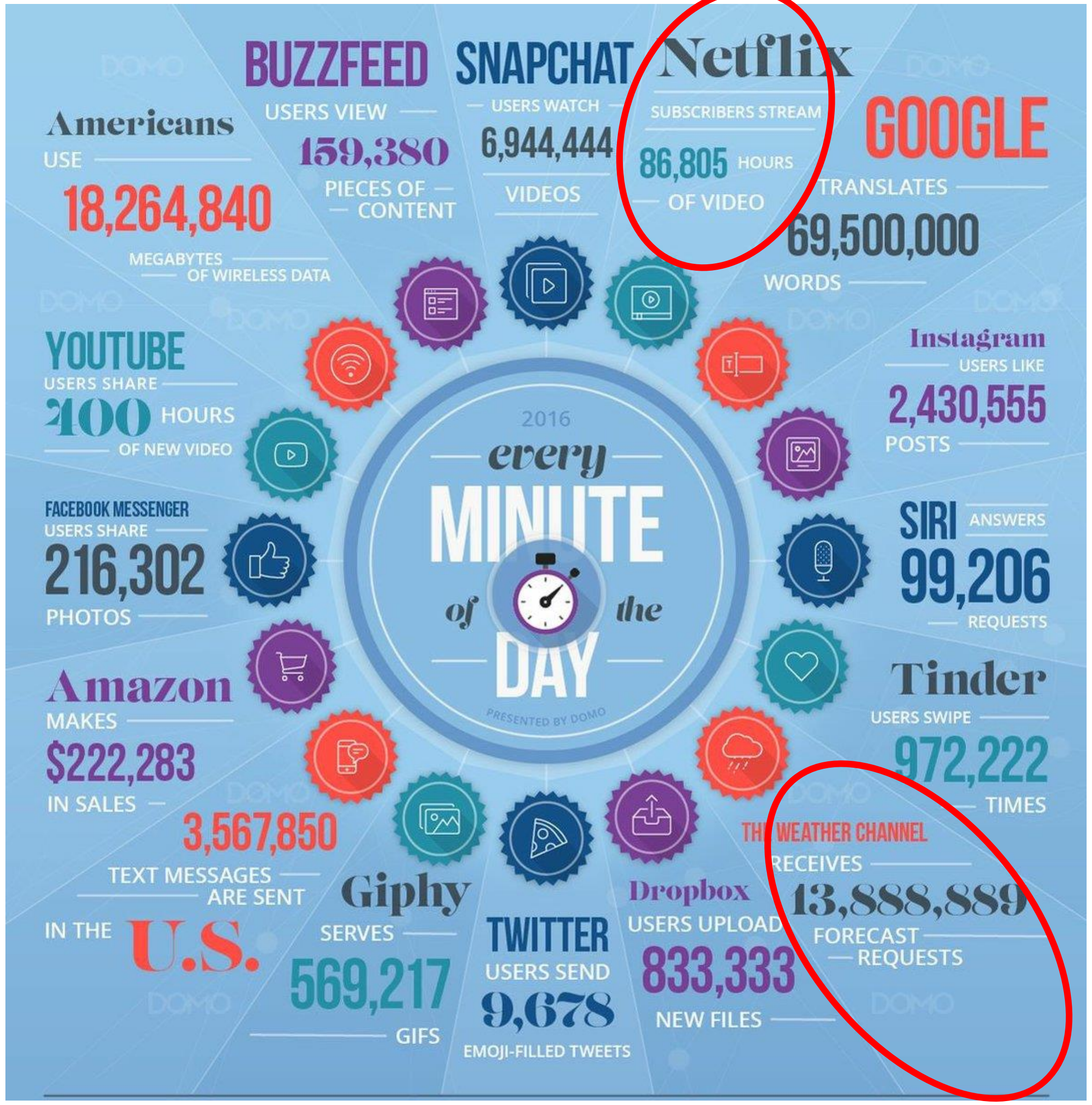
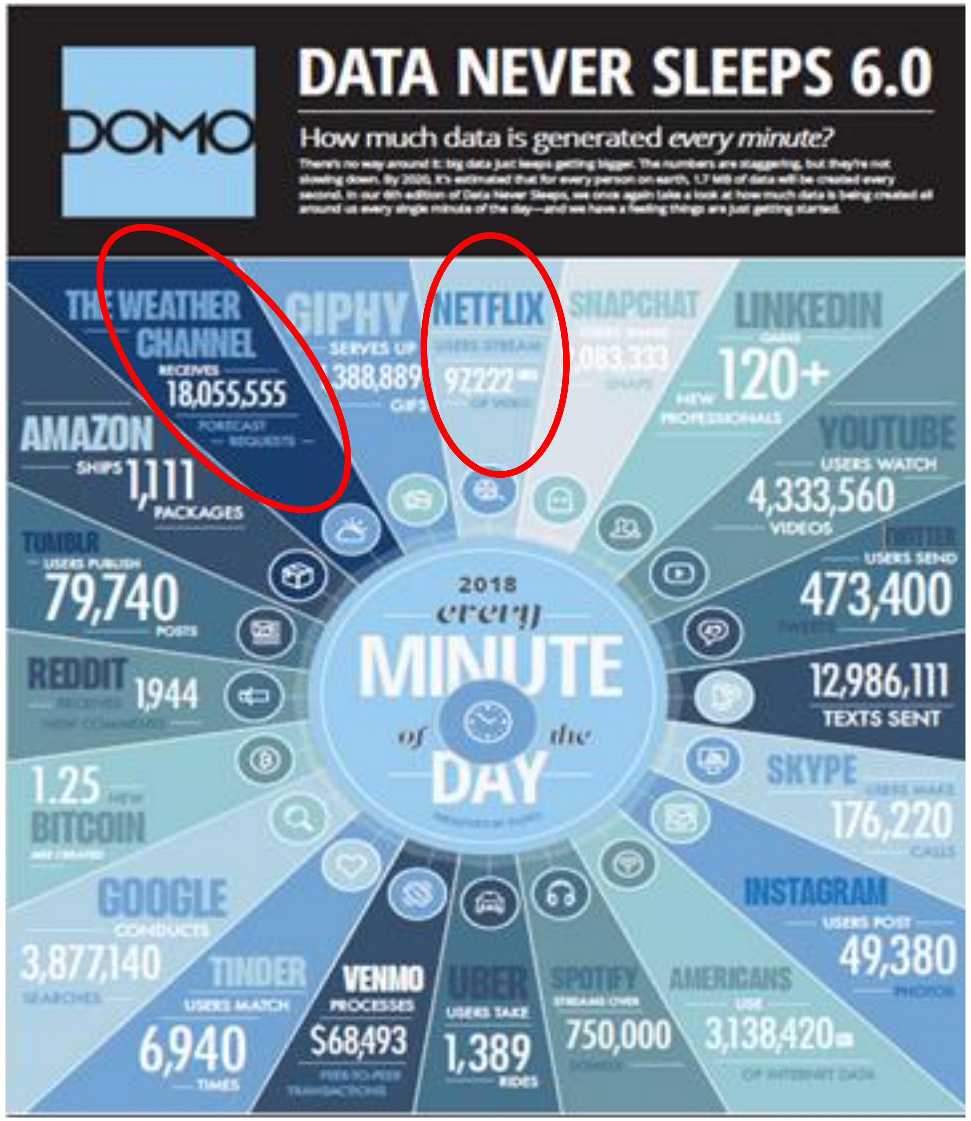
The Functional Programming Model

(MapReduce)

目录

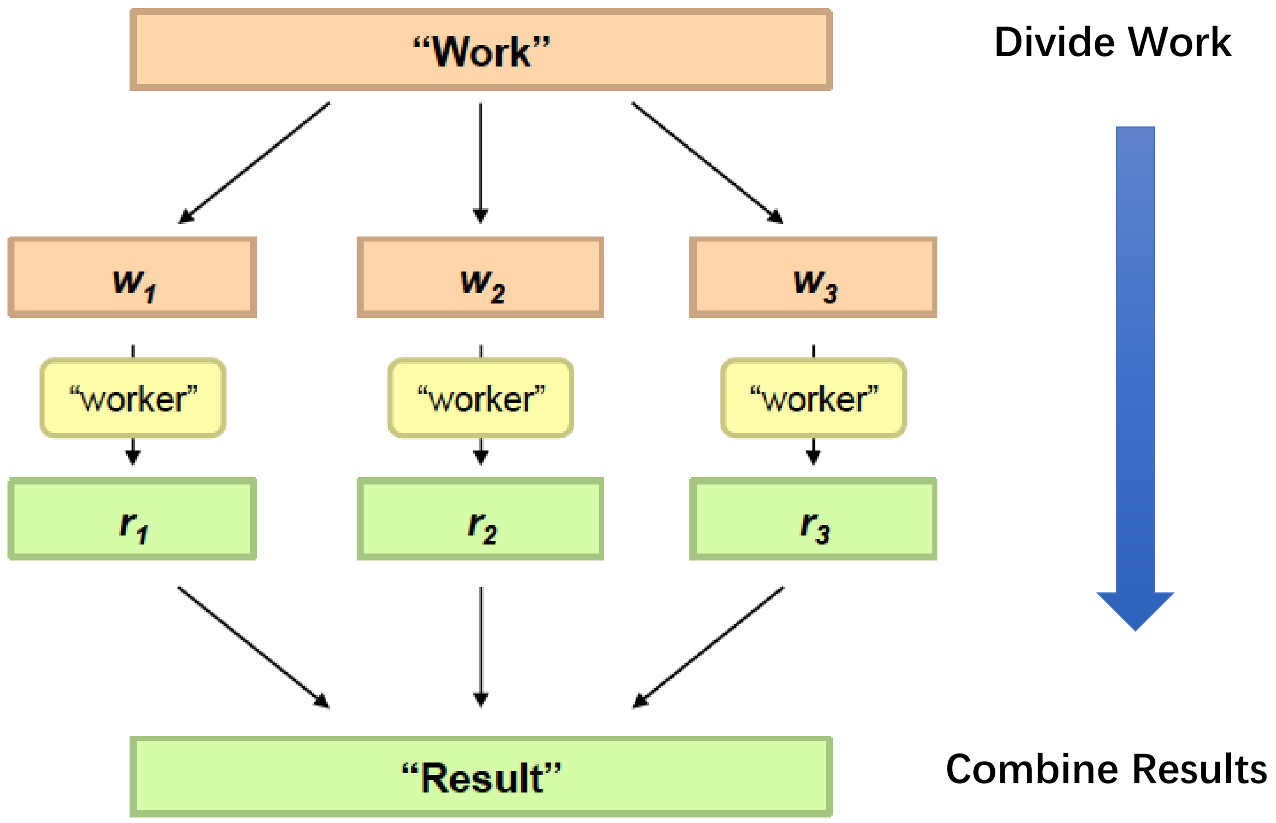
缘起


BUZZFEED
USERS VIEW
159,380
PIECES OF CONTENT
SNAPCHAT
USERS WATCH
6,944,444
VIDEOS
Netflix
SUBSCRIBERS STREAM
86,805 HOURS
OF VIDEO
GOOGLE
TRANSLATES
69,500,000
WORDS
Americans
USE
18,264,840
MEGABYTES
OF WIRELESS DATA
YOUTUBE
USERS SHARE
400 HOURS
OF NEW VIDEO
FACEBOOK MESSENGER
USERS SHARE
216,302
PHOTOS
Amazon
MAKES
$222,283
IN SALES
3,567,850
TEXT MESSAGES
ARE SENT
IN THE U.S.
Giphy
SERVES
569,217
GIFS
TWITTER
USERS SEND
9,678
EMOJI-FILLED TWEETS
Dropbox
USERS UPLOAD
833,333
NEW FILES
THE WEATHER CHANNEL
RECEIVES
13,888,889
FORECAST
REQUESTS
Tinder
USERS SWIPE
972,222
TIMES
SIRI
ANSWERS
99,206
REQUESTS
Instagram
USERS LIKE
2,430,555
POSTS
2016
every
MINUTE
of the
DAY
PRESENTED BY DOMO


DOMO
DATA NEVER SLEEPS 6.0
How much data is generated every minute?
THE WEATHER CHANNEL
RECEIVES
18,055,555
FORECAST REQUESTS
GIPHY
SERVES UP
388,889
GIFS
NETFLIX
USERS STREAM
97,222
SNAPCHAT
2,083,333
SNAPS
LINKEDIN
120+
NEW PROFESSIONALS
AMAZON
SHIPS 1,111
PACKAGES
YOUTUBE
USERS WATCH
4,333,560
VIDEOS
TUMBLR
USERS PUBLISH
79,740
POSTS
TWITTER
USERS SEND
473,400
TWEETS
REDDIT
1944
12,986,111
TEXTS SENT
1.25
BITCOIN
SKYPE
176,220
CALLS
INSTAGRAM
USERS POST
49,380
PHOTOS
GOOGLE
CONDUCTS
3,877,140
SEARCHES
TINDER
USERS MATCH
6,940
TIMES
VENMO
PROCESSES
$68,493
PEER-TO-PEER TRANSACTIONS
UBER
USERS TAKE
1,389
RIDES
SPOTIFY
750,000
AMERICANS
USE
3,138,420
OF INTERNET DATA
2018
every
MINUTE
of the
DAY

分治法(Divide and Conquer)

- 处理大数据问题的一种可行方法
  - 分解大问题成为若干小问题
  - worker并行处理小问题
  - 合并worker得出的中间结果
- Worker可以是
  - 处理器核中的Thread
  - 多核处理器中的核
  - 机器中的多处理器
  - 集群中的机器



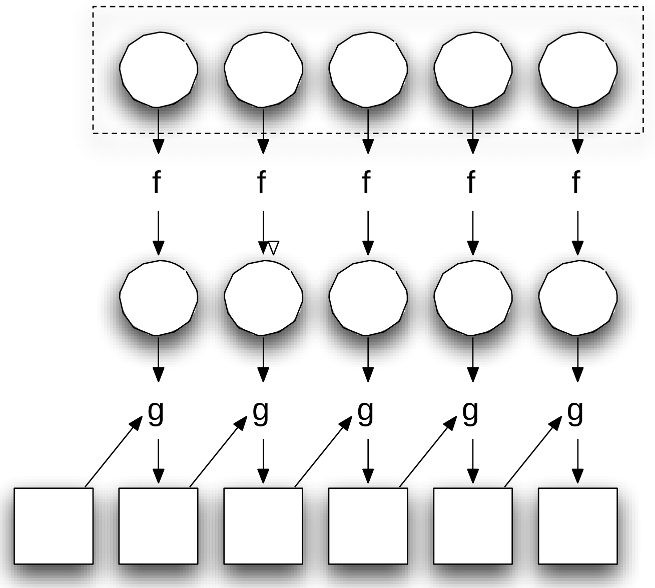
分治法(Divide and Conquer)

- 分解原始问题成为小并行任务，将任务调度到集群上的worker执行
  - 数据局部性
  - 资源可用性
- 确保worker获得它所需要的数据
- 协调、同步worker，共享部分结果
- 处理失效

MapReduce Approach

- OpenMP, MPI, …
  - 开发人员需要关注几乎所有方面，如：同步、并发；资源分发
- MapReduce:
  - 绝大多数以上问题开发人员不用关心
  - 开发人员需要重点关注：问题分解和部分结果分享
  - 优化内存和网络使用

Functional Programming



- **higher order functions**
  - 函数接收其它函数作为参数
  - **Map、Fold**
- map phase:
  - Map将列表作为f的参数，函数f对列表的每一元素处理
- fold phase:
  - fold 将列表作为函数g的参数之一及初始值
  - g首先作用于初始值及列表中的第一项
  - 结果存储在中间变量里，然后，它和列表的第2项作为g的输入
  - 这个过程重复直到列表中的所有项都处理完毕

Functional Programming

- 函数操作不会修改数据结构，而是创建新的
  - 原数据始终不会更改
- 程序设计中数据流不明确
- 操作顺序不重要

Functional Programming和MapReduce

- MapReduce和Functional Programming是等价的
  - MapReduce 实现了functional programming一子集

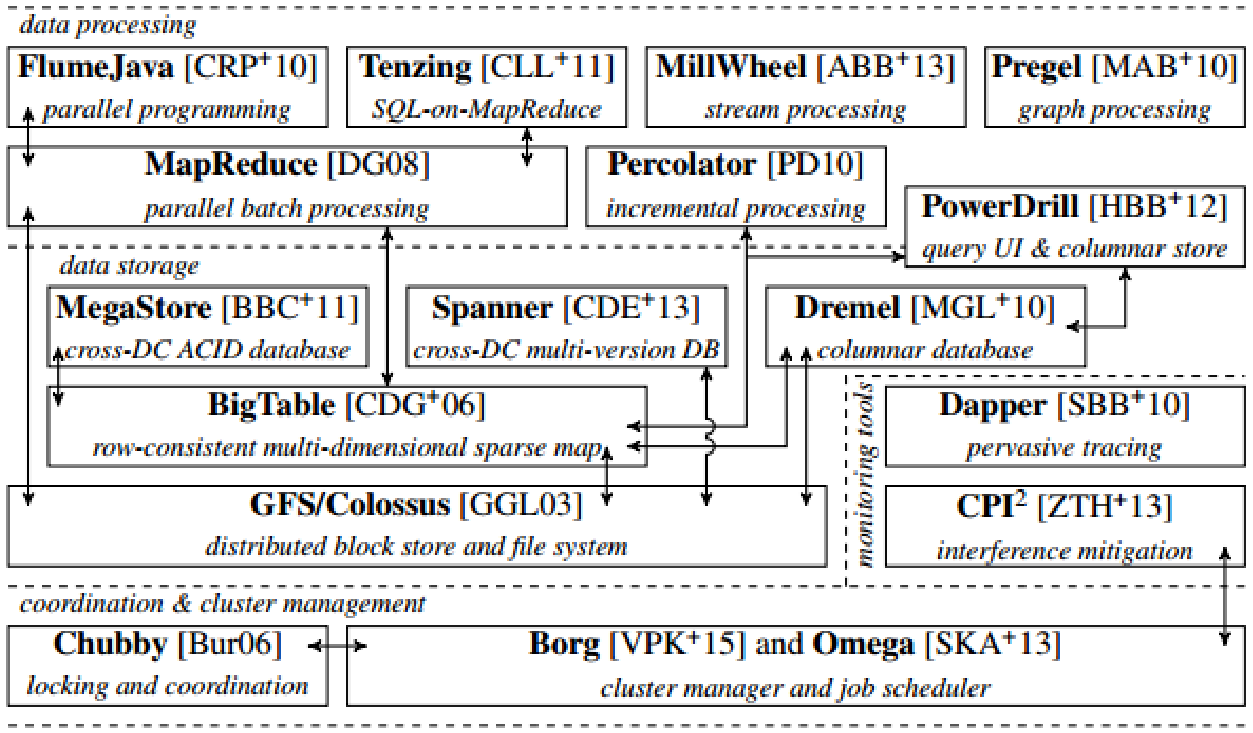
- 框架协调map和reduce phases:
  - 并行归并中间结果
- 用户指定的计算并行应用于所有输入记录
- 中间结果由另一用户给定的计算汇总

MapReduce can refer to…

- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

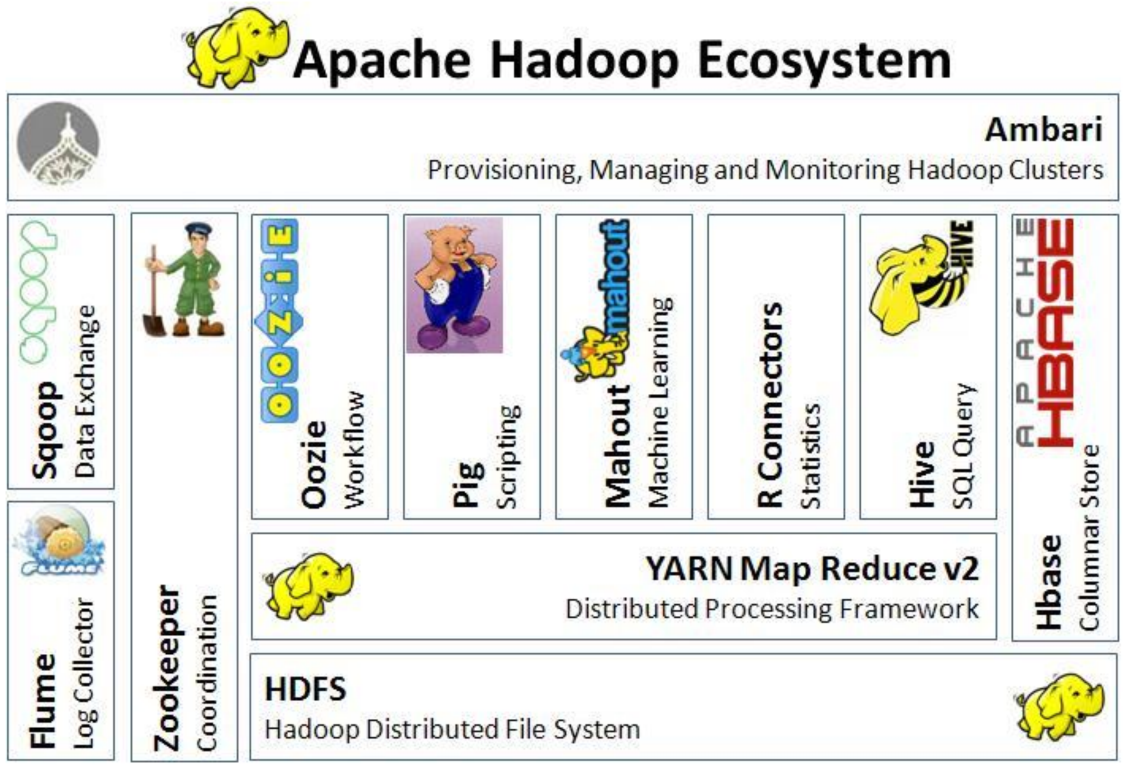
Usage is usually clear from context!

Google大数据处理技术和Hadoop



data processing
FlumeJava [CRP+10] parallel programming
Tenzing [CLL+11] SQL-on-MapReduce
MillWheel [ABB+13] stream processing
Pregel [MAB+10] graph processing
MapReduce [DG08] parallel batch processing
Percolator [PD10] incremental processing
PowerDrill [HBB+12] query UI & columnar store
data storage
MegaStore [BBC+11] cross-DC ACID database
Spanner [CDE+13] cross-DC multi-version DB
Dremel [MGL+10] columnar database
BigTable [CDG+06] row-consistent multi-dimensional sparse map
GFS/Colossus [GGL03] distributed block store and file system
monitoring tools
Dapper [SBB+10] pervasive tracing
CPI2 [ZTH+13] interference mitigation
coordination & cluster management
Chubby [Bur06] locking and coordination
Borg [VPK+15] and Omega [SKA+13] cluster manager and job scheduler

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113.



2005: Doug Cutting and Michael J. Cafarella developed Hadoop to support distribution for the Nutch search engine project(invert index)

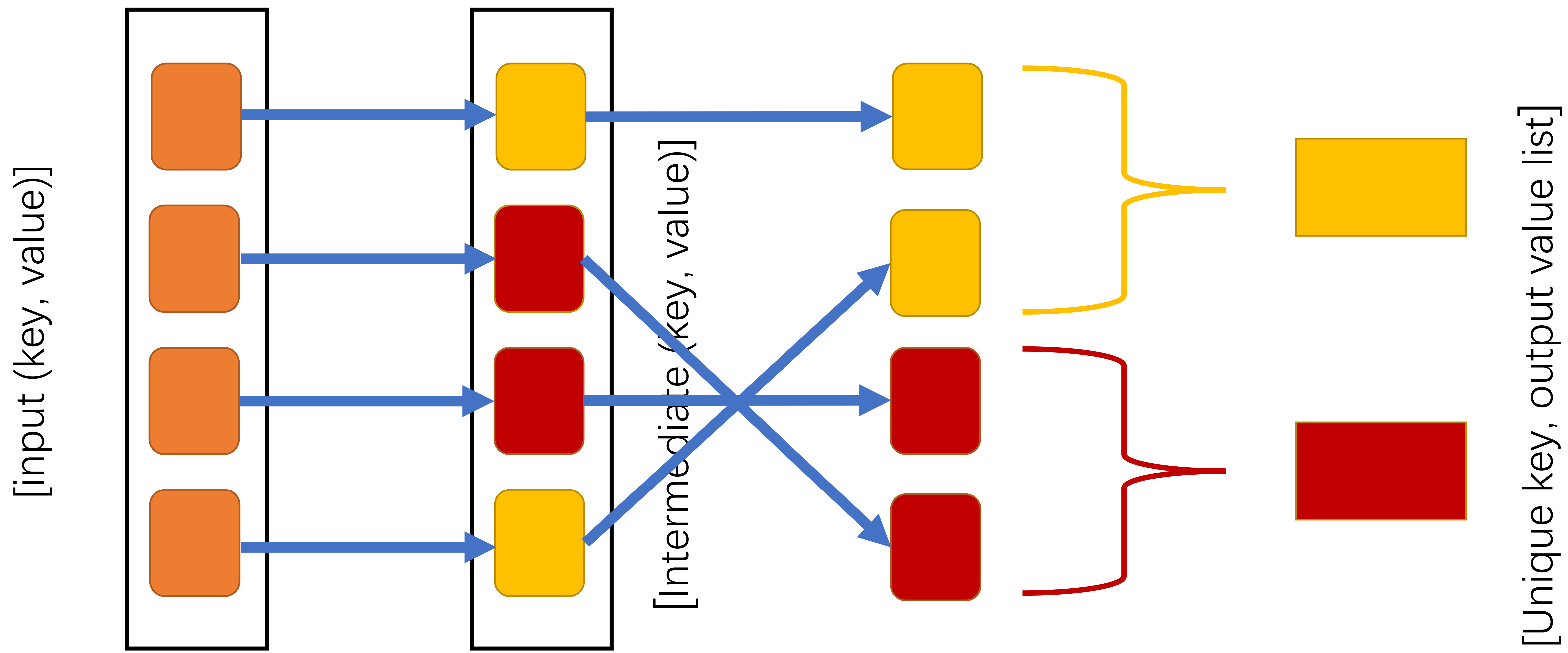
MR model

- Map()
  - 处理key/value，产生中间key/value对
- Reduce()
  - 根据key归并所有中间值
- 用户实现两基本方法:
  - 1. Map: (key1, val1) → (key2, val2)
  - 2. Reduce: (key2, [val2]) → [val3]
- *Map - clause group-by (for Key) of an aggregate function* of SQL
- Reduce - *aggregate* function (e.g., average) that is computed over all the rows with the same group-by attribute (key).

执行框架处理所有其它事项

- **执行框架**
  - 调度: 分配任务给map和reduce任务
  - 数据分发: 将处理靠近数据
  - Synchronization: gather, sort, 和 shuffles intermediate data
  - 容错: 检测worker失效并重启任务
- **对数据流和执行流的控制有限**
  - 所有算法须用m, r, c, p表达

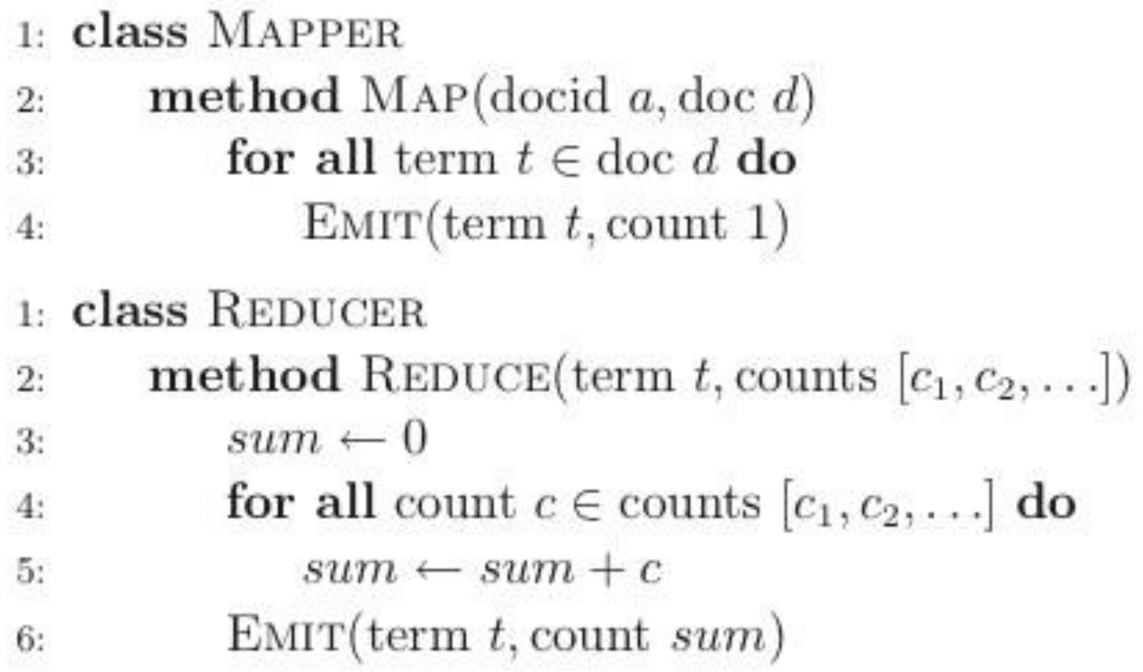
Programming Model



Map Function   Shuffle (merge sort by key)   Reduce function

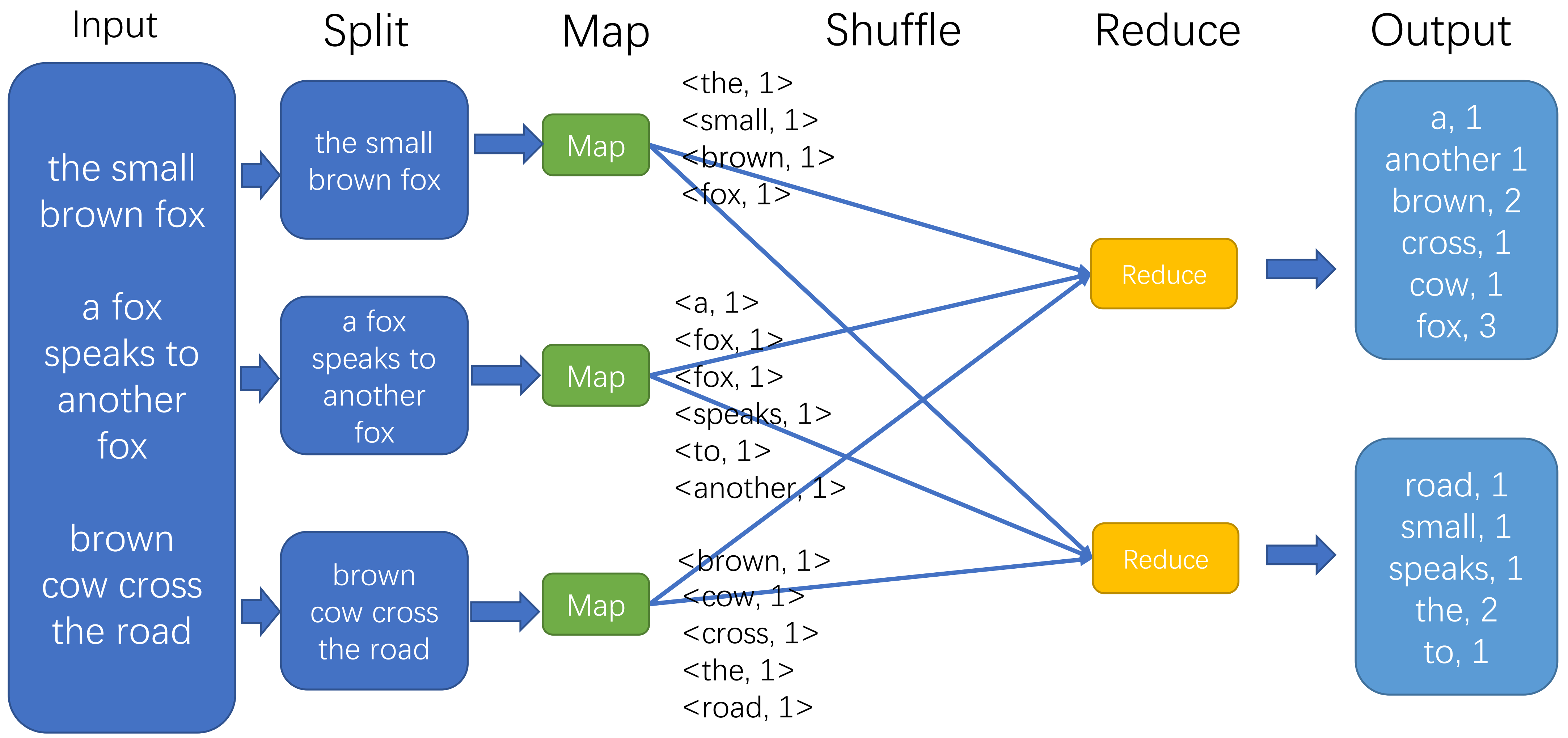
Word count

- 统计文档中词汇出现的次数



```
class Mapper
    method Map(docid $a$, doc $d$)
        for all term $t \in$ doc $d$ do
            Emit(term $t$, count 1)

class Reducer
    method Reduce(term $t$, counts $[c_1, c_2, \ldots]$)
        $sum \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
            $sum \leftarrow sum + c$
        Emit(term $t$, count $sum$)
```

示例: WordCount



术语

- MapReduce:
  - Job: 为处理一数据集，程序（ Mapper和Reducer ）的一次运行
  - Task: 一Mapper或Reducer处理一数据块
  - Task Attempt: instance of an attempt to execute a task
    - Task attempted at least once, possibly more
    - Multiple crashes on input imply discarding it
    - Multiple attempts may occur in parallel (speculative execution)
- 例子:
  - Job：运行 “Word Count”程序处理20个文件
  - 20个文件映射到 = 20 map tasks + 一定数量的reduce任务
  - 至少20次尝试。如果机器崩溃，会有更多尝试

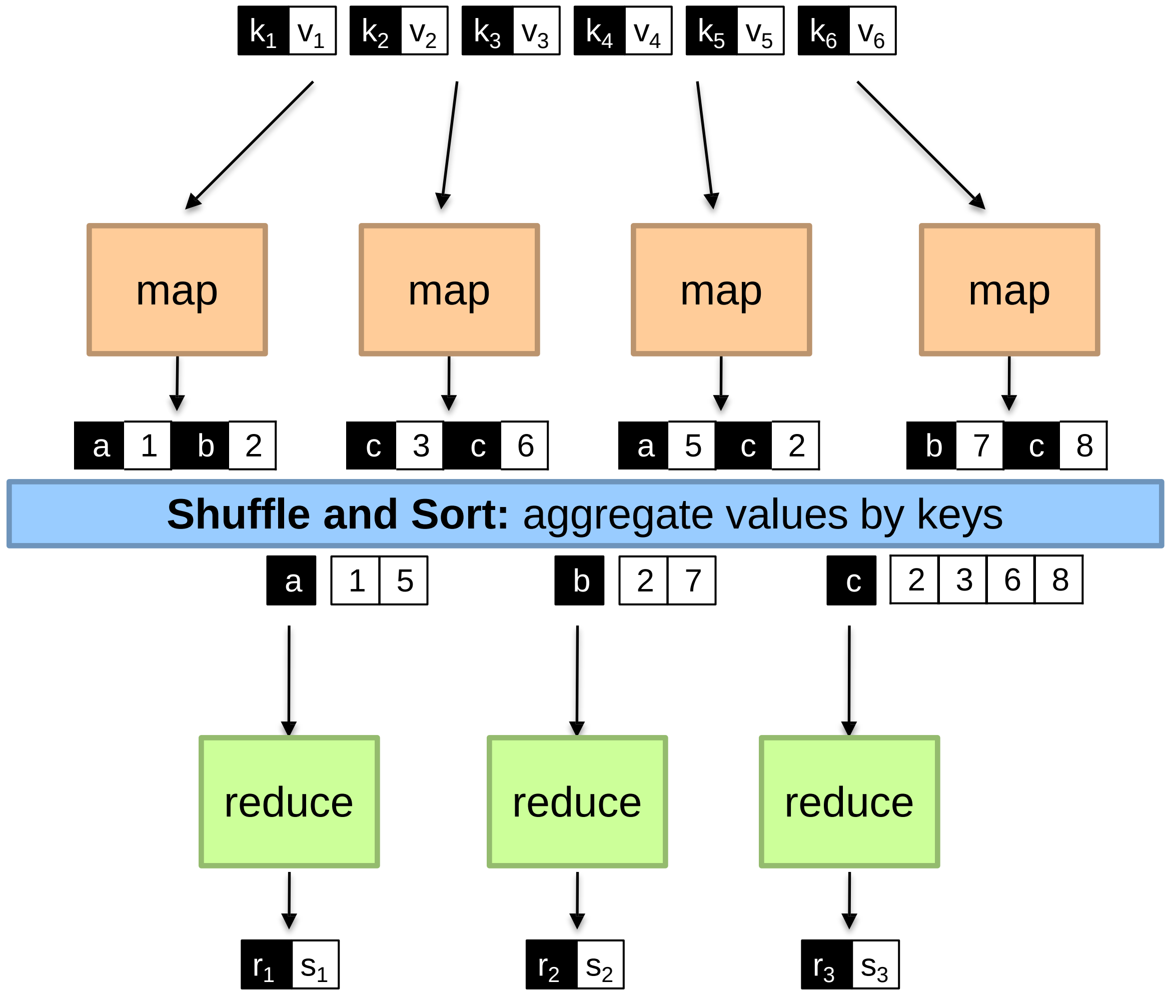
数据结构

- Key-value pairs是MapReduce基本数据结构
  - Key和value可以是: 整数、浮点数, 字符串, raw bytes
  - They can also be arbitrary data structures
- MapReduce算法:
  - 将key-value structure用于表达任意数据集
    - E.g.: 一组网页，其中key可以是URL，value是网页内容
  - 有的算法里key用于唯一识别一记录，也有的算法里key未使用
  - Key可以多种方式组合，以适应算法的需要

A MapReduce job

- Mapper和reducer:
    - map: (k1, v1) → [(k2, v2)]
    - reduce: (k2, [v2]) → [(k3, v3)]
- MapReduce job包含:
    - 存储在分布式文件系统的一数据集。它被分解成一组文件
    - Mapper处理每一输入key-value pair，输出中间key-value pairs
    - Reducer处理同一中间key的所有value，输出key-value pairs

MapReduce简单视图



k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

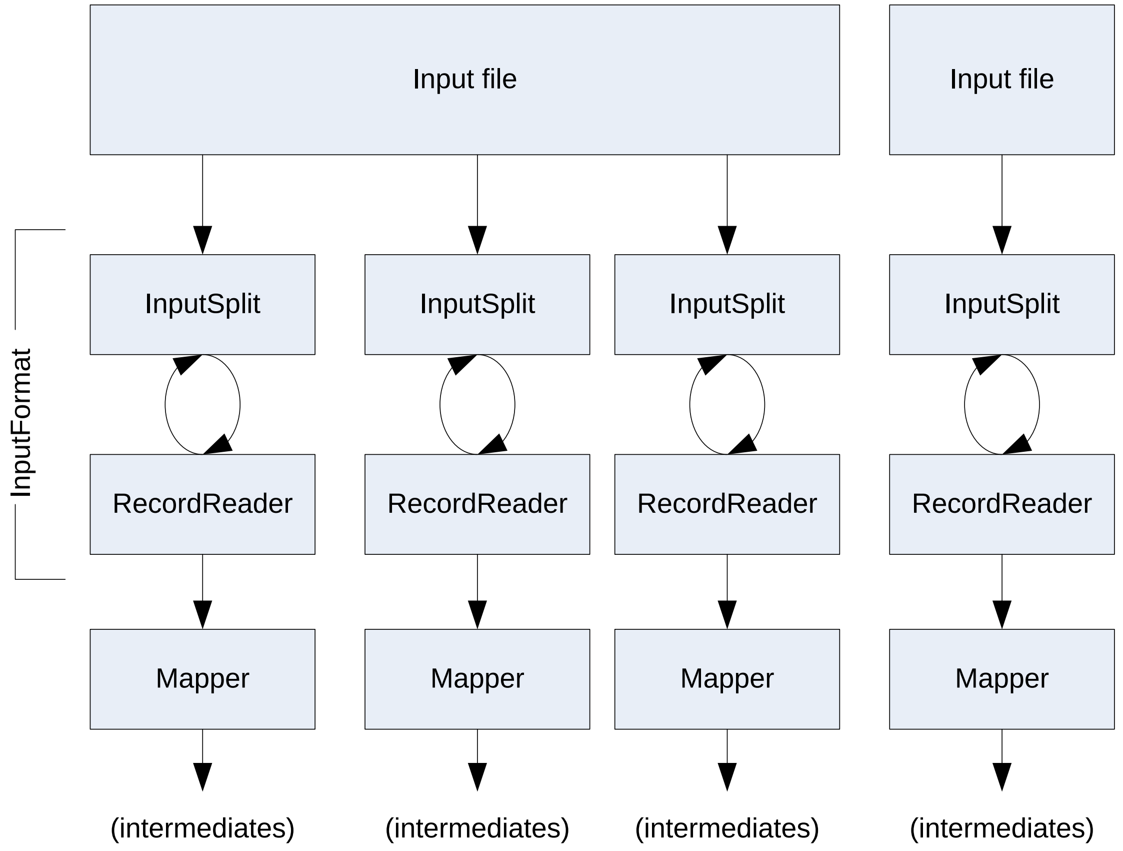
- 应用开发人员指定：函数***Map***和***Reduce***以及一组输入文件，并提交任务
- 工作流
  - *Input* phase从输入文件产生大量***FileSplit*** (每一Map任务一个)
  - *Map* **phase** 执行一用户函数，将输入kev-pair变换成一组新的kev-pair
  - 框架sorts & ***Shuffles*** the kev-pairs to output nodes
  - ***Reduce*** phase根据key合并所有kev-pair，形成新的kevpair
  - output phase将结果pairs写入文件
- **所有phase都有许多任务**
  - 框架在集群上调度任务
  - 当节点失效时，框架负责恢复

任务分配及数据分发

- 分配任务
  - 启动用户程序的多个实例，其中一实例成为Master
  - Master找到空闲机器，并分配任务给它
  - 利用数据局部性，在拥有数据的机器上执行map任务
- 数据分发
  - 输入：在分布式文件系统里，输入文件分成片，如 128 MB blocks
  - 在map和reduce阶段有一个对中间结果（中间key）分布式 “group by”操作
    - 中间结果根据key排序分发给reducer
    - reducer之间key无序
  - 从map任务产生的中间结果没有保存在分布式文件系统，而是临时写在机器的磁盘里
  - 从reducer输出的key写回分布式文件系统
    - 输出可能有r个文件，每个文件对应一reducer
    - 该输出可以是下一MapReduce阶段的输入

Input Splitter

- 负责将输入分割成多个chunk
- Chunk然后输入到mapper
- Chunk缺省大小是64MB
- 通常使用内置的splitter, 也可用自己的splitter来处理特定格式的文件



Mapper

- 读输入<K,V>pair
- 输出<K’, V’>对

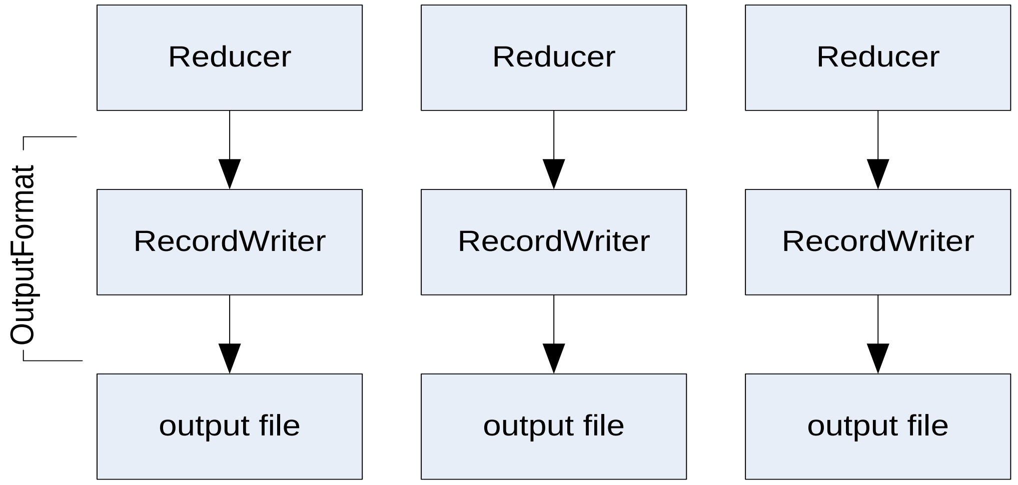
- 在词统计例子，假设输入：“The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.”
- 输出:
  - <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>

Reducer

- 接收Mapper的输出，根据key汇集值
  - 同一key的输入须送往同一reducer
- 如在词统计例子，假设reducer的输入:
  - <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>
- 输出:
  - <The, 6> <teacher, 1> <went, 1> <to, 1> <store, 3> <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

Output Committer

- 负责将reducer的输出写入文件
- 通常它需要相应的input splitter (以便其它作业能够将它作为输入读入)
- 如果无特殊需求，可以使用内置的splitter

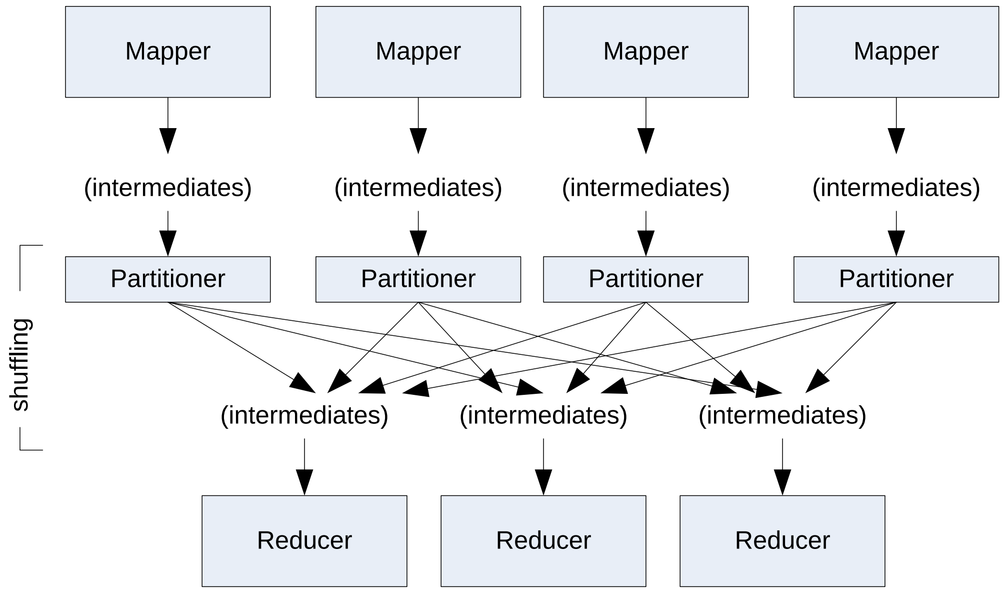


Writables

- 可序列化/反序列化的类型
- input/output class要求是writable, 以便framework序列化数据写入磁盘
  - 基本数据类型，如String, Integer, Long, etc.
  - 集合数据类型，如array, map, etc.
- 用户需在自己定义的数据类型里实现该接口，然后才能序列化数据
- 一个应用需要至少6个writables
  - 2个用于input
  - 2个用于中间结果 (Map <-> Reduce)
  - 2 个用于输出

Partitioner

- Partitioner负责:
  - 按照key划分中间结果
  - 指定中间key-value pair须拷贝给哪一reducer任务
- Hash-based partitioner
  - Default：Key.hashCode() % numOfReducers
  - 确保按照key大致均衡
    - 忽略value可能导致reducer处理数据时不均衡
  - 当处理复杂的key或其它需求时, partitioner可能需要定制
    - 缺省的较简单，当需要更加平衡的负载时
    - 例如：当计算词对<W1, W2>的概率时，所有W1须送往同一reducer



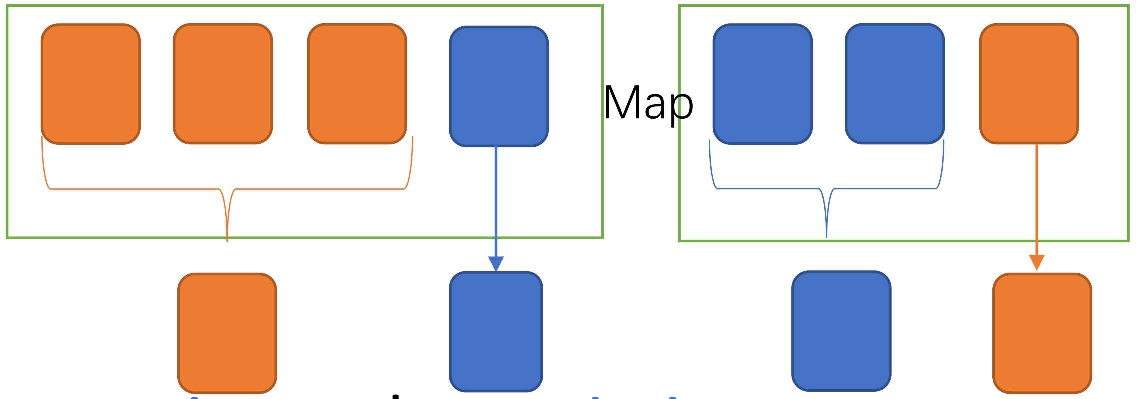
Combiner

- Combiner本质上是中间结果的reducer（可选）：
  - 在“shuffle and sort”阶段之前，允许在本地汇集中间结果
  - 每一combiner独立运行
- 减少来自mapper的输出，因此减少了带宽需求和排序开销
- 不会改变输入的类型

Combiner Function

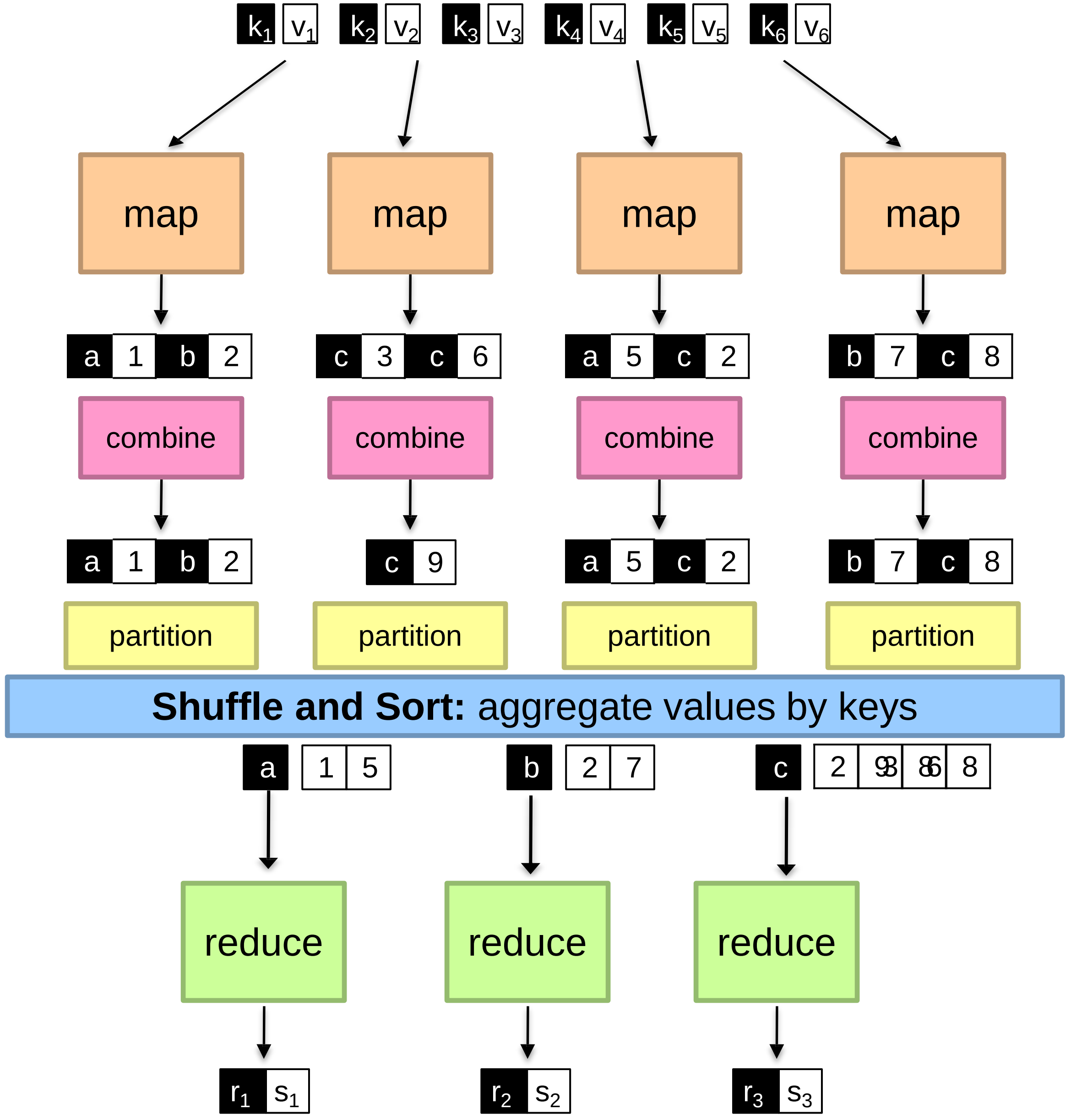
- Local reducer at the map worker
- Can save network time by pre-aggregating at mapper

combine(k1, list(v1))



- Works only if reduce function is commutative and associative

Partitioners and Combiners



Complete view of MapReduce illustrating combiners and partitioners.

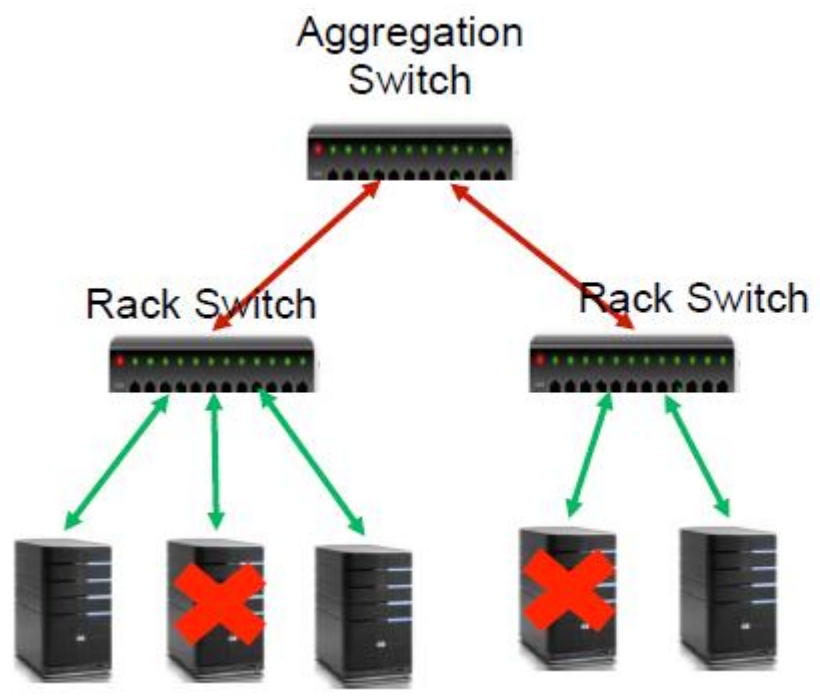
Master

- 负责调度和管理作业
  - 计算应尽可能接近数据
    - 带宽贵且慢
    - GFS/HDFS可以复制数据到本地
- 如果任务未能报告进展 (如reading input, writing output, etc), crashes, the machine goes down, etc, 可认为任务失败，kill任务，然后用同一输入重启该步骤
  - Master可以重启失败节点
    - Nodes should have no side effects!
  - 如果节点是最后一步，且运行慢，master可以启动该节点一副本
    - 由于硬件、网络原因等.
    - 一旦有节点首先完成，其它立即终止

Master

- Master的数据结构
  - Task status: (idle, in-progress, completed)
    - 当worker可用时，Idle task可以调度去执行
  - 当一map任务完成时， 它发送给master：R个中间文件的大小及位置(one for each reducer)
  - Master逐渐将这些push到reducer
- Master周期性ping worker来检查失效
- Master由framework提供, 不需要用户代码

Fault Tolerance

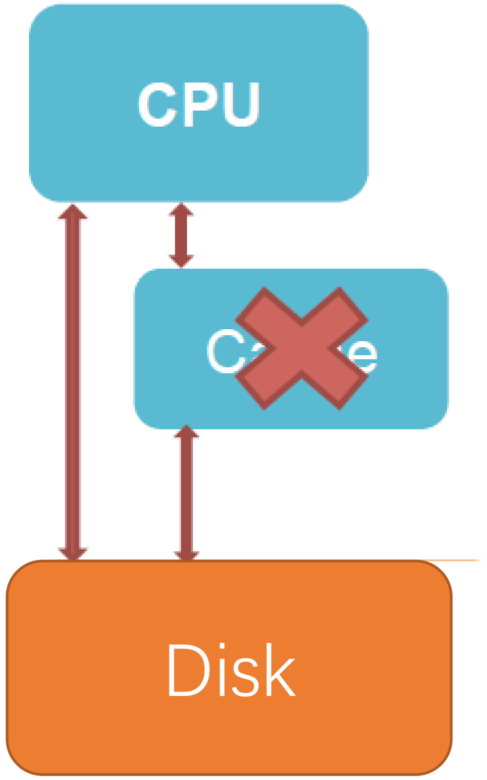


- Map worker failure
  - Completed or in-progress tasks are reset to idle
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce Task is aborted and client is notified

- Reset tasks are rescheduled on another machine

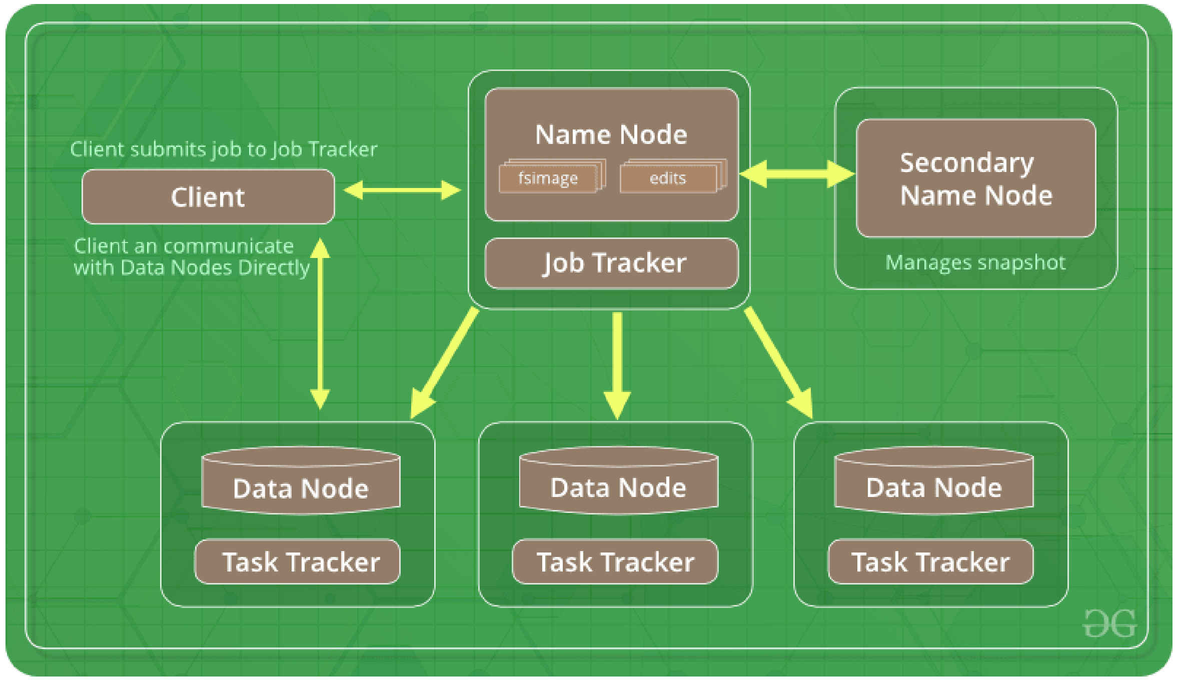
Backup Tasks

- Slow worker delay completion time
  - Processor's cache being disabled
  - Bad disk with soft errors
- Start back-up tasks, for those in-progress as the job nears completion
  - First task to complete is considered

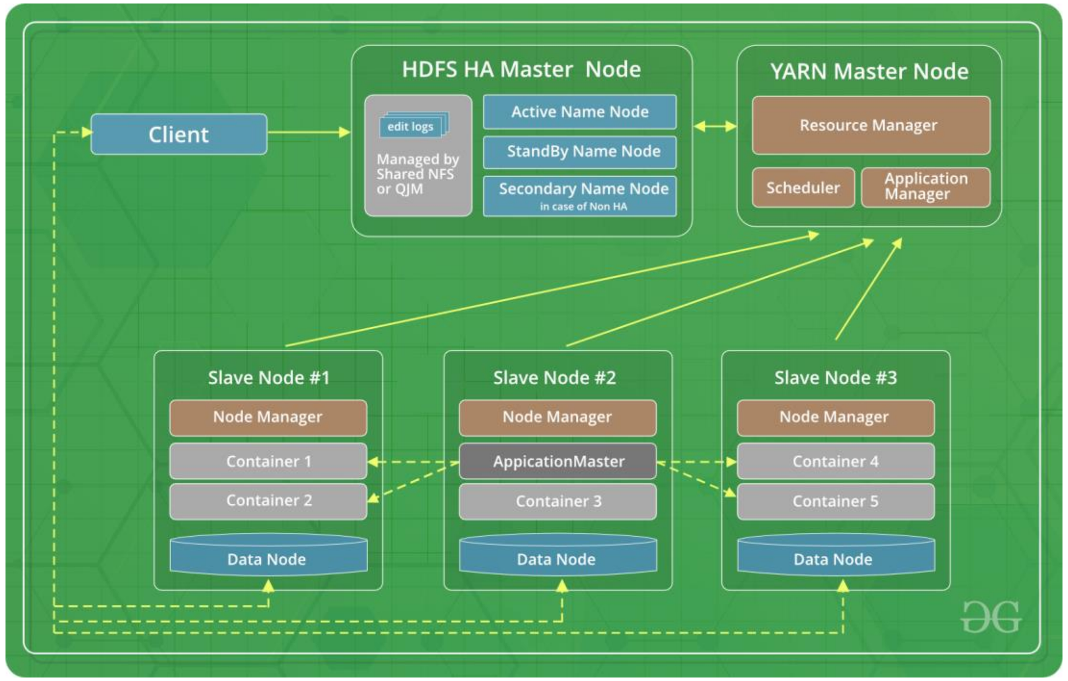


Hadoop MapReduce工作过程

Hadoop发展



Hadoop 1



Hadoop 2

```
public class WordCount {

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

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

  public static class Reduce extends MapReduceBase implements
              Reducer<Text, IntWritable, Text, IntWritable> {
    public 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));
  }}

  public static void main(String[] args) throws Exception {
    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.setReducerClass(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);
  }}
```

Mapper

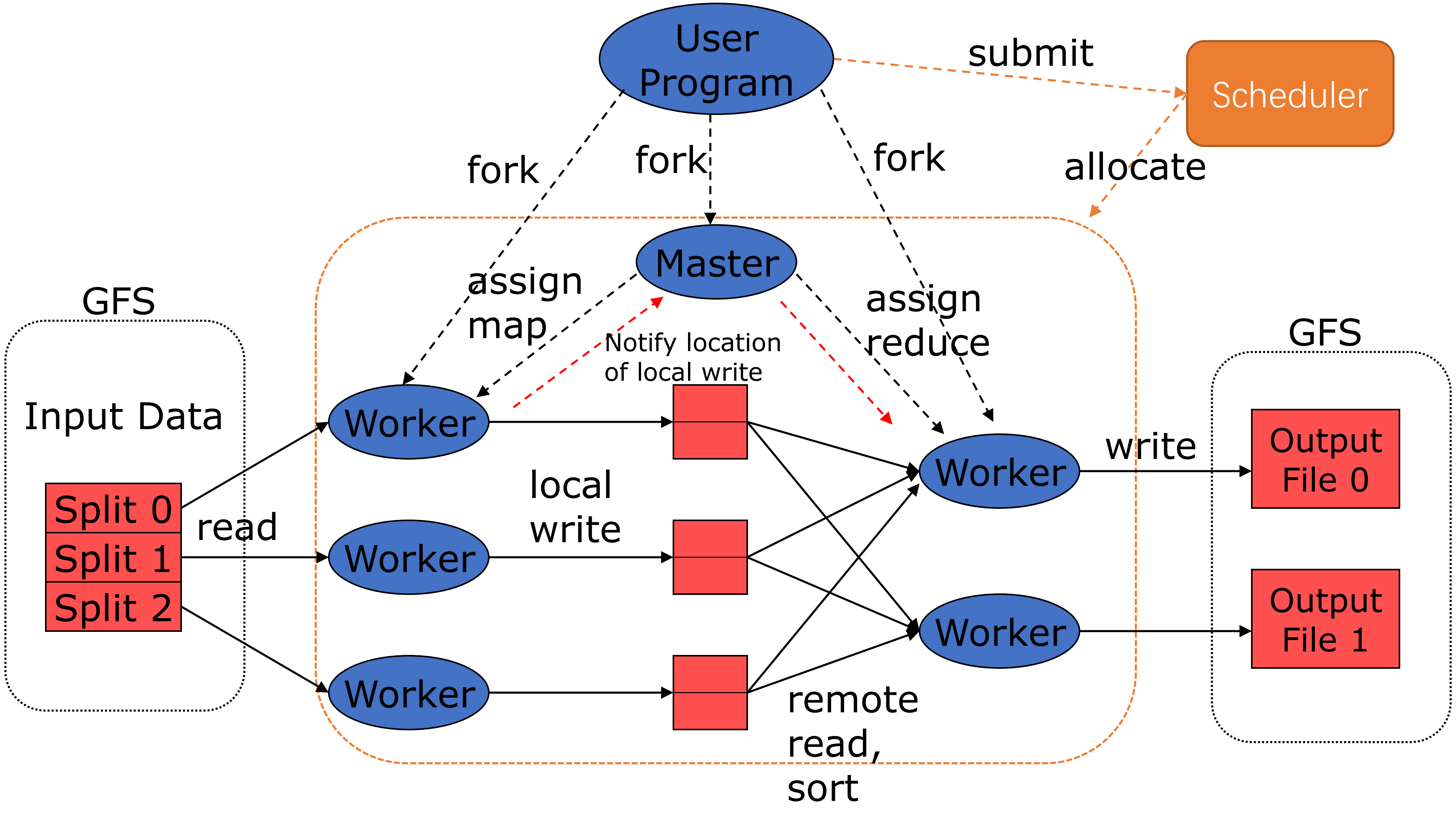
Reducer

Run this program as a MapReduce job

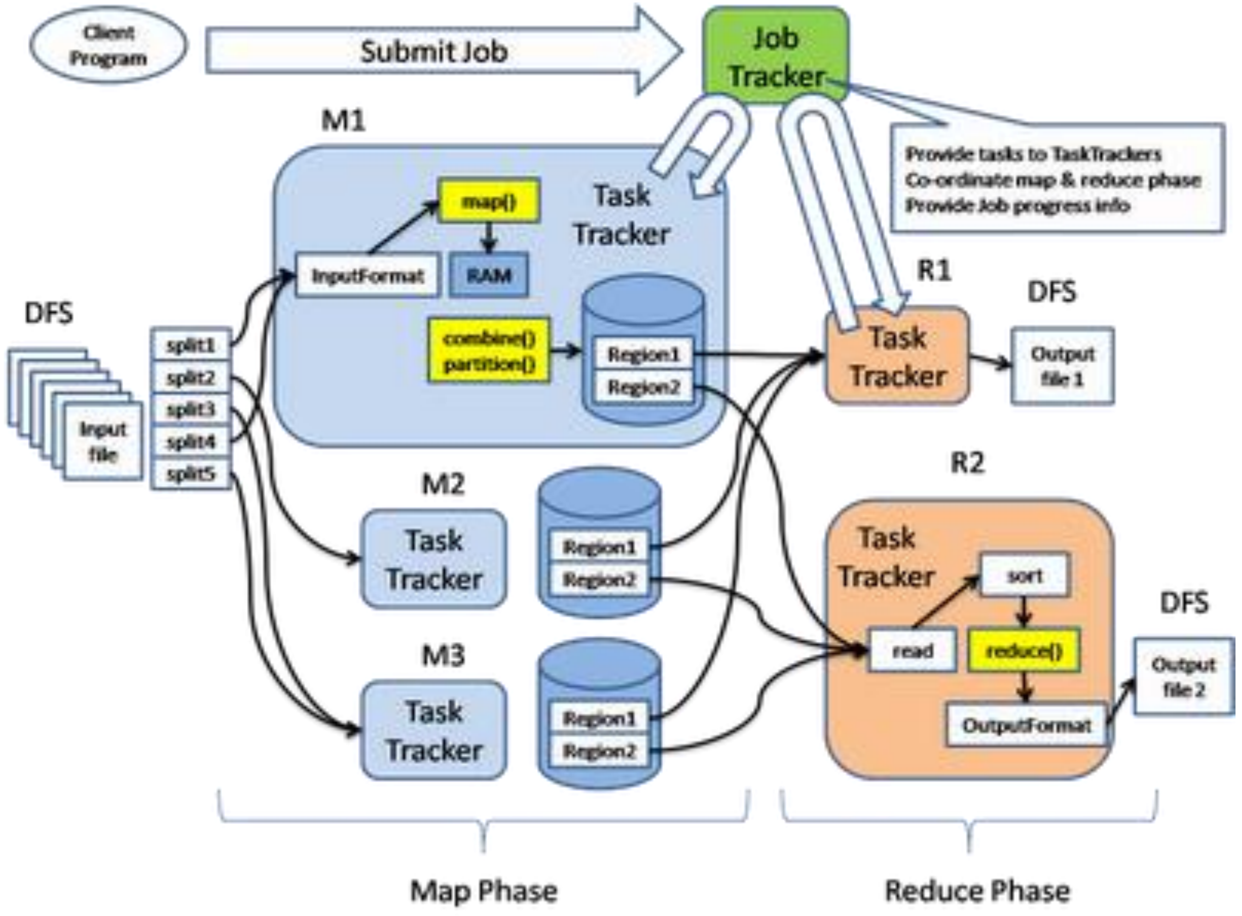
The Execution Framework

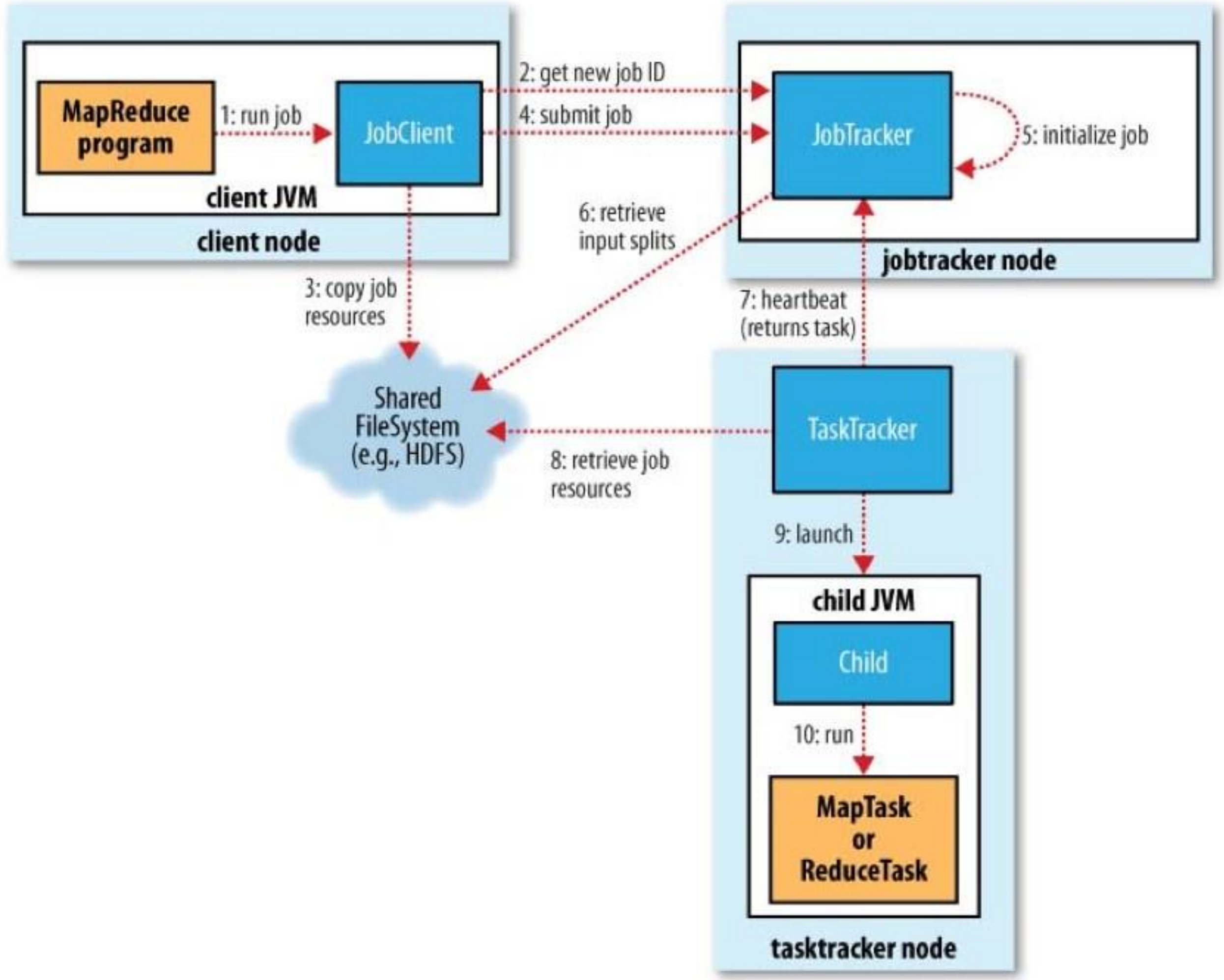
- MapReduce程序，即作业:
  - Code of mappers and reducers
  - Code for combiners and partitioners (optional)
  - Configuration parameters
- 一MapReduce作业提交给集群
  - 框架负责其它所有事

Control Flow and data flow



MapReduce Job Run示意





Job Submission

- JobClient class
    - runJob()方法创建JobClient的一新实例
    - 然后用该实例调用submitJob()

Job Initialization

- JobTracker负责:
  - 为作业创建对象
  - 封装它的任务
  - 记录任务status和progress
- 调度
  - JobTracker维护一调度队列
  - 可添加队列规则
- Compute mappers and reducers
  - JobTracker retrieves input splits (computed by JobClient)
  - 根据input split的数量确定Mapper的数量
  - 读配置文件，设置Reducer的数量

Task Assignment

- 选择一任务
  - JobTracker首先需要选择一作业 (i.e. scheduling)
  - TaskTrackers有固定数量的slots for map and reduce tasks
  - JobTracker给map任务优先级
- Data locality
  - MapReduce里，数据如何喂给任务跟调度及分布式文件系统交织在一起
  - JobTracker是拓扑感知
    - 对map任务有用，对reduce任务无用
  - 当节点具有数据时，调度器在该节点启动相应任务
  - 如果不行，任务在它处启动，数据将传输过去
  - 距离将是考量因素，有助于降低带宽消耗
    - Same rack scheduling
- 心跳机制（Hearbeat-based mechanism）
  - TaskTracker周期性发送hearbeat给JobTracker
    - 表明TaskTracker“活”着
    - Heartbeat包含TaskTrackers的可用信息
  - 如果TaskTracker可用，JobTracker piggybacks a task

Task Execution

- Task Assignement之后，TaskTrackers可以执行
  - Copy the JAR from the HDFS
  - Create a local working directory
  - Create an instance of TaskRunner
- TaskRunner启动一child JVM
  - 这是为了防止bug让TaskTracker停止运行
  - 为每一InputSplit创建一child JVM
- 用户定义的map和reduce方法可以不用Java实现，可用C++或python

Scheduling

- 每一作业分解成任务
  - Map任务处理数据集的一部分，该数据集通常跟底层分布式文件系统有关
  - Reduce任务处理中间结果，并写回分布式文件系统
- 任务数量可能超过机器的数量
  - 调度器维护任务队列，当机器空闲时，分配任务过去
- 作业同样需要调度
  - 用户们提交的作业数量较多
  - 作业的复杂度不一样
  - 公平性

Scheduling

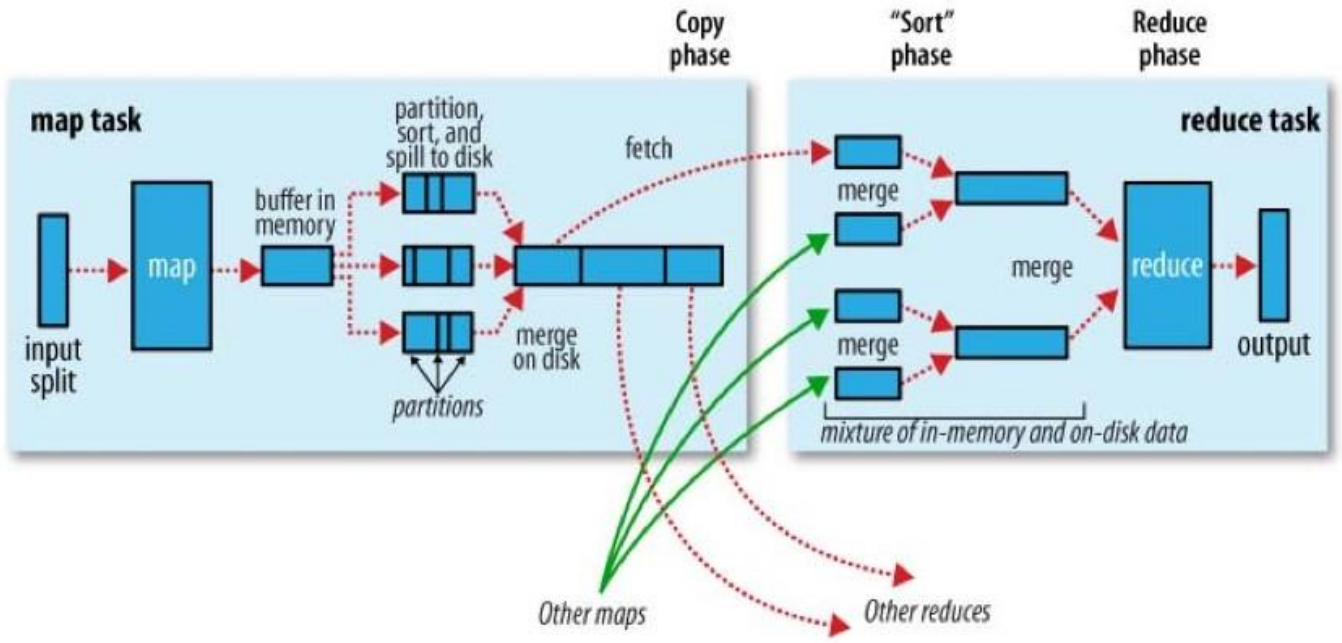
- Dealing with stragglers
  - 作业执行时间取决于最慢的map和reduce任务
  - 猜测执行（Speculative execution）对运行速度慢的机器可能有帮助
- 数据值分布偏斜
  - 例如: 传感器采集的数据，如温度
  - 此时调度可能无法解决，可以用定制的划分方法或采样技术解决

Scheduling

- 调度器可以定制
    - Hadoop有多个scheduler
- FIFO Scheduler (default behavior)
    - 每一作业使用整个集群：长作业独占机器，短作业无法保证执行时间，面临饿死
- Fair Scheduler
    - 每一用户分配一定份额的集群计算能力（时间、机器等）
    - 作业都放到池子里，每一用户一个池子
        - 提交更多作业的用户并不比其他用户能获得更多资源
        - 确保每个池子获得最小的计算能力
    - 支持剥夺（preemption）
- Capacity Scheduler
    - Hierarchical queues (mimic an oragnization)
    - FIFO scheduling in each queue
    - Supports priority

Shuffle and Sort

- 排序并转换map输出的过程：MapReduce框架确保每一reducer的输入都按照key排序
- Map Side
  - Map任务输出不是简单写入磁盘，而是在内存缓存，预排序
  - Circular memory buffer （缺省100 MB）
    - Threshold based mechanism to spill buffer content to disk
    - Map输出写入，同时溢出到磁盘(spilling to disk)
    - 当buffer溢出时写满(buffer fills up while spilling)，map任务阻塞
  - Disk spills
    - Written in round-robin to a local dir
    - 根据所要送往的reducer划分输出数据
    - 每一划分内的数据排序 (in-memory)
    - 如果存在一combiner, 排序阶段之后将执行combiner



Shuffle and Sort: the Reduce Side

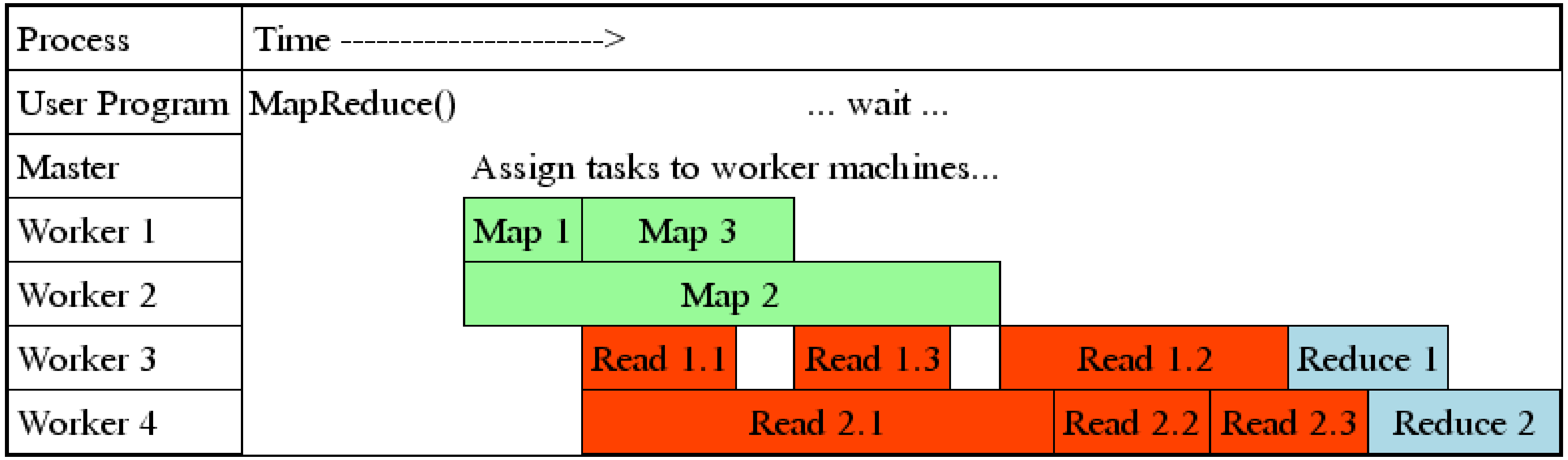
- Map输出文件放在tasktracker的本地磁盘里
- 另一tasktracker (负责一reduce任务)从其它TaskTracker (完成了它们的map任务)获得输入
  - Reducers如何知道从哪些tasktracker获取map输出?
    - 当一map任务完成，它通知父tasktracker
    - Tasktracker通知jobtracker (用heartbeat mechanism)
    - Reducer的一线程周期性 poll the jobtracker
    - Tasktrackers不会马上删除本地map输出，尽管一reduce任务已经取到了输出
- Copy phase: a pull approach
  - 少量拷贝线程 (5)并行地取map输出

Shuffle and Sort: the Reduce Side

- Map的输出拷贝给运行reducer的trasktracker (in memory)
  - 否则，拷贝到磁盘
- Input consolidation
  - 后台线程合并所有partial input到一个大的排序文件里
  - 如果用了压缩(for map outputs to save bandwidth), 在内存里解压
- 排序输入
  - 当所有map输出已拷贝，合并阶段开始
  - 所有map输出排序

Synchronization

- 在MapReduce里, synchronization通过“shuffle and sort”路障实现
  - Intermediate key-value pairs按照key归类
  - 这需要分布式排序，涉及所有mappers, 同时考虑所有reducers
  - 若有m mappers和 r reducers，至多需要m × r拷贝操作
- 可将shuffling和map执行流水



Handling Failures

- Task Failure
  - Case 1: map或reduce任务抛出异常
    - child JVM向父TaskTracker报告
    - TaskTracker记录错误，标记此TaskAttempt失败
    - TaskTracker释放slot，运行另一任务
  - Case 2: Hanging tasks
    - 长时间没有向TaskTracker更新进展（no progress updates，timeout = 10 minutes)
    - TaskTracker kills the child JVM
  - 告知JobTracker，一任务失败
    - 避免在同一TaskTracker重新调度该任务
    - 如果一任务失败4次，该任务将不再调度
    - Default behavior: if any task fails 4 times, the job fails

Handling Failures

- TaskTracker Failure
  - Types: crash, running very slowly
  - 没有发送Heartbeat给JobTracker
  - JobTracker等待超时 (10 minutes)，从调度池删除该TaskTracker
  - JobTracker需要重新调度已完成任务、进行中任务
  - 如果太多任务失败，JobTracker可能将一TaskTracker放入黑名单
- JobTracker Failure

Example: Sparse Matrices with Map/Reduce

$$\overset{A}{\begin{bmatrix} 10 & & 20 \\ & 30 & 40 \\ 50 & 60 & 70 \end{bmatrix}} \times \overset{B}{\begin{bmatrix} -1 & \\ -2 & -3 \\ & -4 \end{bmatrix}} = \overset{C}{\begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}}$$

- Task: 计算C = A·B
- A、B是稀疏矩阵

计算稀疏矩阵积

- 矩阵表示成⟨**row, col, value, matrixID**⟩
- 执行
  - **Phase 1: 计算$a_{i,k} \cdot b_{k,j}$**
  - **Phase 2: 对每一(i,j)求和**
  - **每一阶段涉及 Map/Reduce**

A

$$\begin{bmatrix} 10 & & 20 \\ & 30 & 40 \\ 50 & 60 & 70 \end{bmatrix}$$

$1 \xrightarrow[A]{10} 1$

$1 \xrightarrow[A]{20} 3$

$2 \xrightarrow[A]{30} 2$

$2 \xrightarrow[A]{40} 3$

$3 \xrightarrow[A]{50} 1$

$3 \xrightarrow[A]{60} 2$

$3 \xrightarrow[A]{70} 3$

B

$$\begin{bmatrix} -1 & \\ -2 & -3 \\ & -4 \end{bmatrix}$$

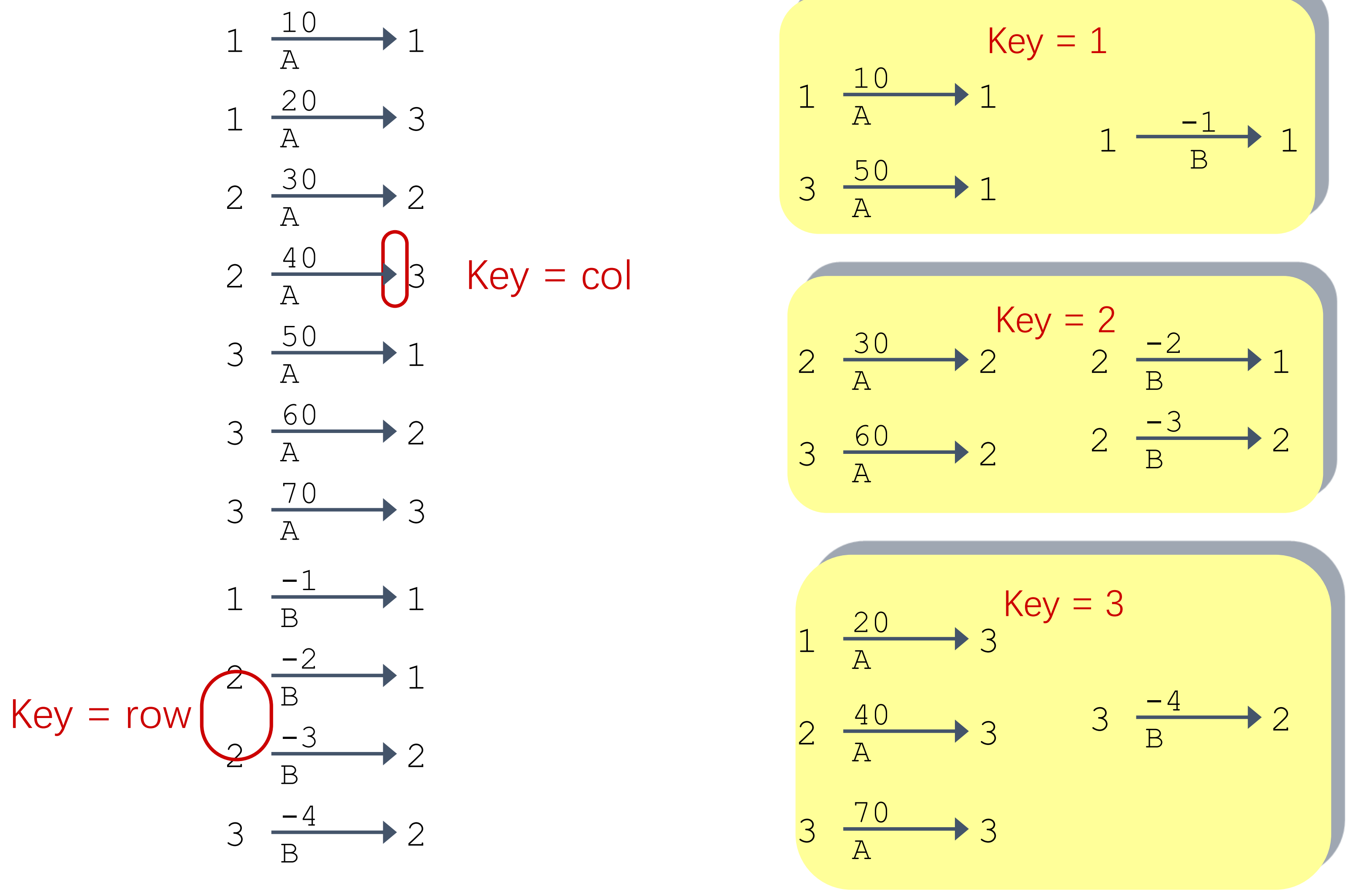
$1 \xrightarrow[B]{-1} 1$

$2 \xrightarrow[B]{-2} 1$

$2 \xrightarrow[B]{-3} 2$

$3 \xrightarrow[B]{-4} 2$

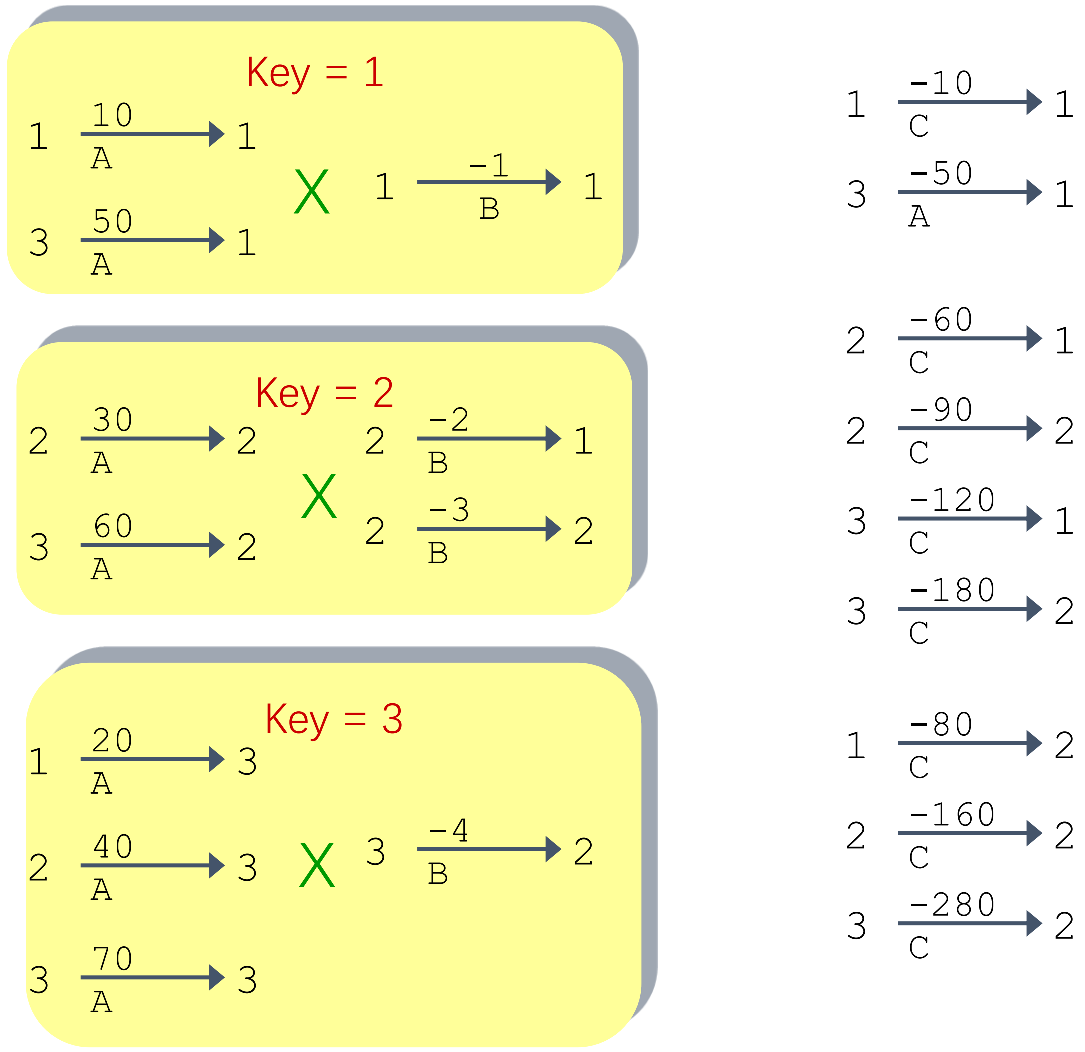
Phase 1 Map of Matrix Multiply



$1 \xrightarrow[A]{10} 1$
$1 \xrightarrow[A]{20} 3$
$2 \xrightarrow[A]{30} 2$
$2 \xrightarrow[A]{40} 3$
Key = col
$3 \xrightarrow[A]{50} 1$
$3 \xrightarrow[A]{60} 2$
$3 \xrightarrow[A]{70} 3$
$1 \xrightarrow[B]{-1} 1$
$2 \xrightarrow[B]{-2} 1$
Key = row
$2 \xrightarrow[B]{-3} 2$
$3 \xrightarrow[B]{-4} 2$
Key = 1
$1 \xrightarrow[A]{10} 1$
$3 \xrightarrow[A]{50} 1$
$1 \xrightarrow[B]{-1} 1$
Key = 2
$2 \xrightarrow[A]{30} 2$
$3 \xrightarrow[A]{60} 2$
$2 \xrightarrow[B]{-2} 1$
$2 \xrightarrow[B]{-3} 2$
Key = 3
$1 \xrightarrow[A]{20} 3$
$2 \xrightarrow[A]{40} 3$
$3 \xrightarrow[A]{70} 3$
$3 \xrightarrow[B]{-4} 2$

- Group values $a_{i,k}$ and $b_{k,j}$ according to key k

Phase 1 “Reduce” of Matrix Multiply



- Generate all products $a_{i,k} \cdot b_{k,j}$

Matrix Multiply Phase 1 Mapper

```
public class P1Mapper extends MapReduceBase implements Mapper {

    public void map(WritableComparable key, Writable values,
                OutputCollector output, Reporter reporter) throws
IOException {
      try {
          GraphEdge e = new GraphEdge(values.toString());
          IntWritable k;
          if (e.tag.equals("A"))
            k = new IntWritable(e.toNode);
          else
            k = new IntWritable(e.fromNode);
          output.collect(k, new Text(e.toString()));
      } catch (BadGraphException e) {}
    }
}
```

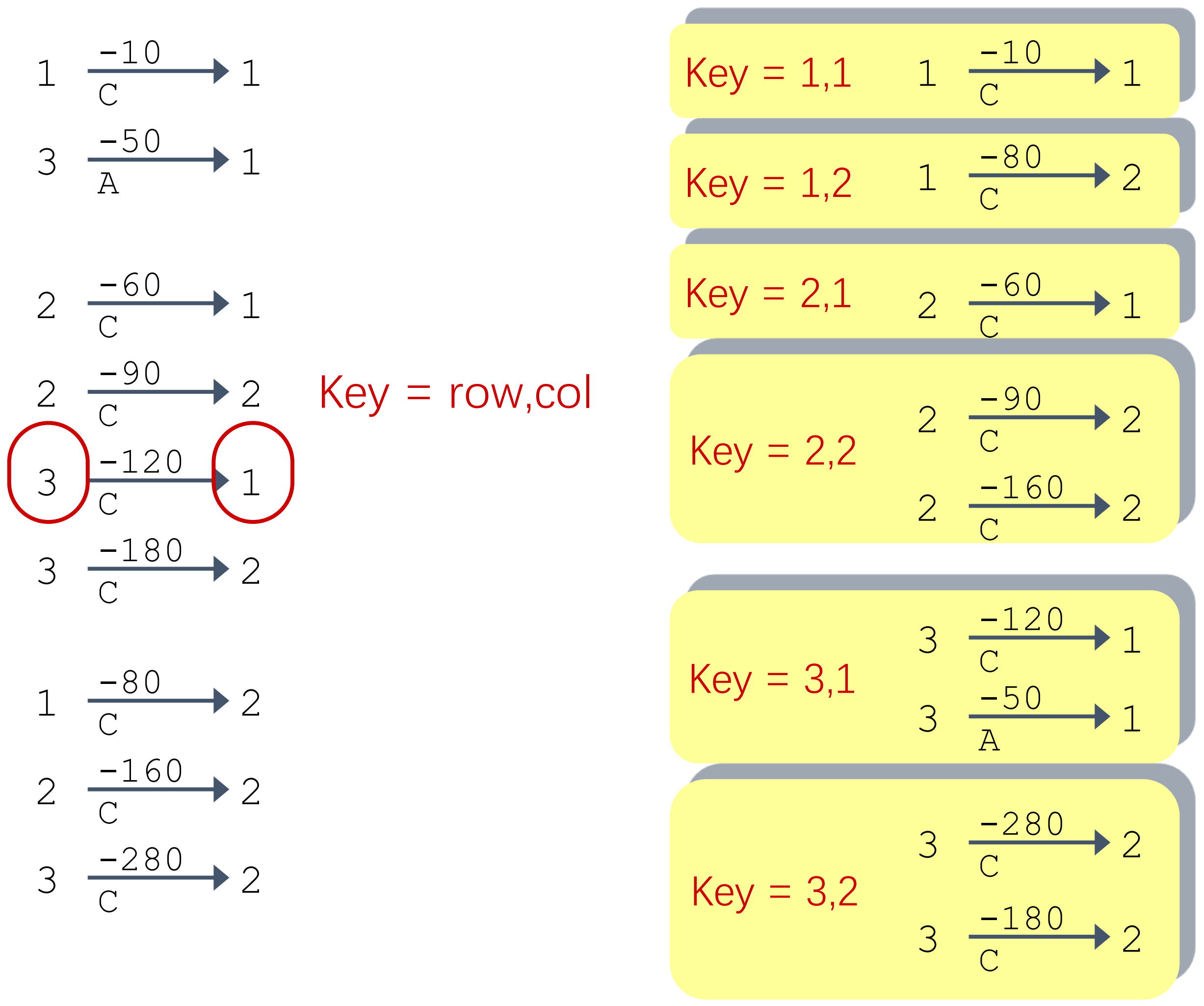
Matrix Multiply Phase 1 Reducer

```
public class P1Reducer extends MapReduceBase implements Reducer {

        public void reduce(WritableComparable key, Iterator values,
                        OutputCollector output, Reporter reporter)
                        throws IOException
    {
        Text outv = new Text(""); // Don't really need output values
        /* First split edges into A and B categories */
        LinkedList<GraphEdge> alist = new LinkedList<GraphEdge>();
        LinkedList<GraphEdge> blist = new LinkedList<GraphEdge>();
        while(values.hasNext()) {
                try {
                    GraphEdge e =
                        new GraphEdge(values.next().toString());
                    if (e.tag.equals("A")) {
                        alist.add(e);
                    } else {
                        blist.add(e);
                    }
                } catch (BadGraphException e) {}
        }
        // Continued
```

MM Phase 1 Reducer (cont.)

```
            // Continuation

            Iterator<GraphEdge> aset = alist.iterator();
            // For each incoming edge
            while(aset.hasNext()) {
                GraphEdge aedge = aset.next();
                // For each outgoing edge
                Iterator<GraphEdge> bset = blist.iterator();
                while (bset.hasNext()) {
                    GraphEdge bedge = bset.next();
                    GraphEdge newe = aedge.contractProd(bedge);
                    // Null would indicate invalid contraction
                    if (newe != null) {
                        Text outk = new Text(newe.toString());
                        output.collect(outk, outv);
                    }
                }
            }
    }
}
```

Phase 2 Map of Matrix Multiply



- Group products $a_{i,k} \cdot b_{k,j}$ with matching values of i and j

Phase 2 Reduce of Matrix Multiply

Key = 1,1 $1 \xrightarrow[C]{-10} 1$

Key = 1,2 $1 \xrightarrow[C]{-80} 2$

Key = 2,1 $2 \xrightarrow[C]{-60} 1$

Key = 2,2 $2 \xrightarrow[C]{-90} 2$ $2 \xrightarrow[C]{-160} 2$

Key = 3,1 $3 \xrightarrow[C]{-120} 1$ $3 \xrightarrow[A]{-50} 1$

Key = 3,2 $3 \xrightarrow[C]{-280} 2$ $3 \xrightarrow[C]{-180} 2$

$1 \xrightarrow[C]{-10} 1$

$1 \xrightarrow[C]{-80} 2$

$2 \xrightarrow[C]{-60} 1$

$2 \xrightarrow[C]{-250} 2$

$3 \xrightarrow[C]{-170} 1$

$3 \xrightarrow[C]{-460} 2$

C

$$\begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}$$

- Sum products to get final entries

Matrix Multiply Phase 2 Mapper

```
public class P2Mapper extends MapReduceBase implements Mapper {

    public void map(WritableComparable key, Writable values,
                    OutputCollector output, Reporter reporter)
                        throws IOException {
        String es = values.toString();
        try {
            GraphEdge e = new GraphEdge(es);
            // Key based on head & tail nodes
            String ks = e.fromNode + " " + e.toNode;
            output.collect(new Text(ks), new Text(e.toString()));
        } catch (BadGraphException e) {}

    }
}
```

Matrix Multiply Phase 2 Reducer

```
public class P2Reducer extends MapReduceBase implements Reducer {
        public void reduce(WritableComparable key, Iterator values,
                OutputCollector output, Reporter reporter)throws IOException {
        GraphEdge efinal = null;
        while (efinal == null && values.hasNext()) {
            try {
                efinal = new GraphEdge(values.next().toString());
            } catch (BadGraphException e) {}
        }
        if (efinal != null) {
            while(values.hasNext()) {
                try {
                    GraphEdge eother = new GraphEdge(values.next().toString());
                    efinal.weight += eother.weight;
                } catch (BadGraphException e) {}
            }
            if (efinal.weight != 0)
                output.collect(new Text(efinal.toString()),
                        new Text(""));
        }
    }
}
```

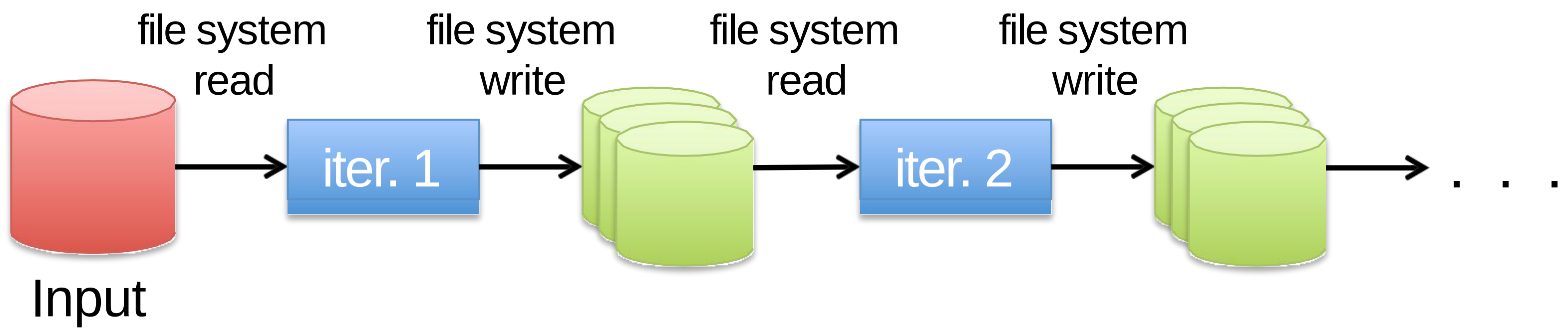
Spark

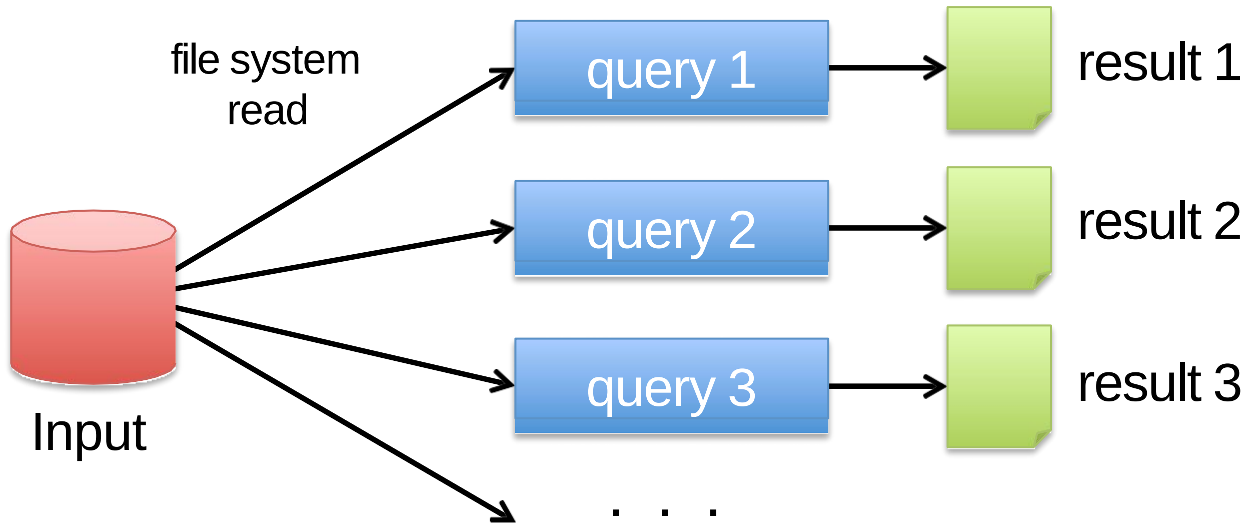
- Limitations of MapReduce
- Spark简介
- Example
- Spark Streaming

Limitations of MapReduce

- MapReduce is great at one-pass computation, but inefficient for multi-pass algorithms
- No efficient primitives for data sharing
  - State between steps goes to distributed file system
  - Slow due to replication & disk storage

Example: Iterative Apps



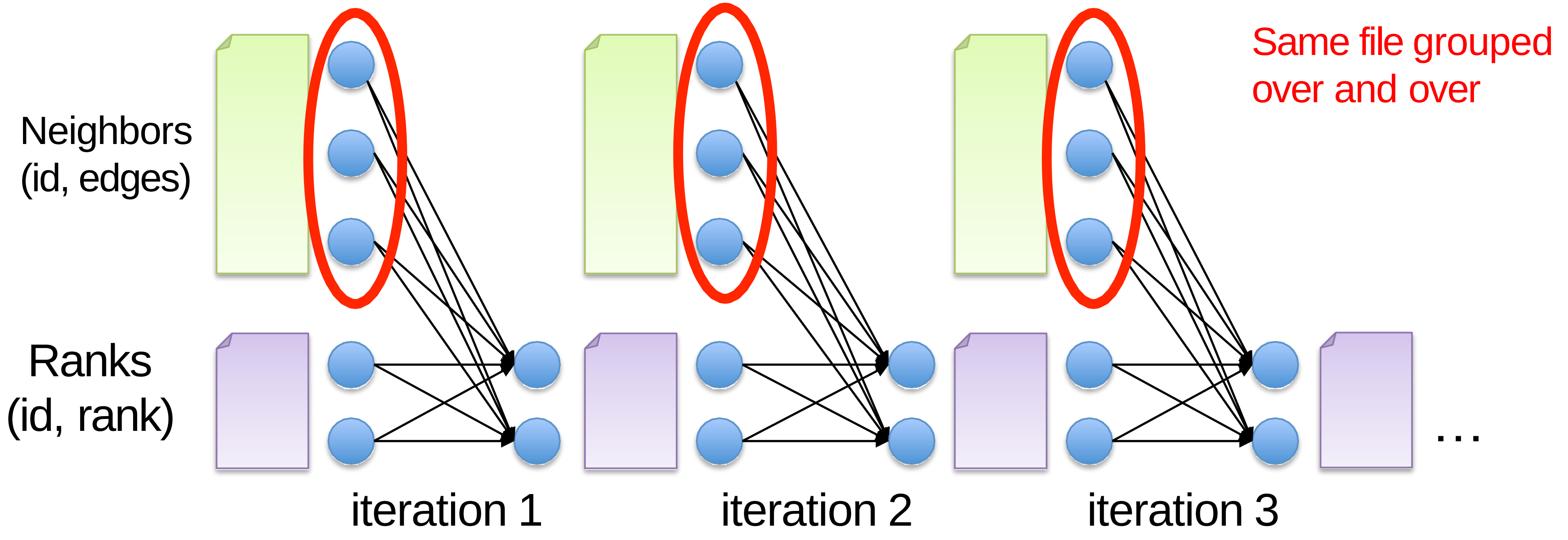


Commonly spend 90% of time doing I/O

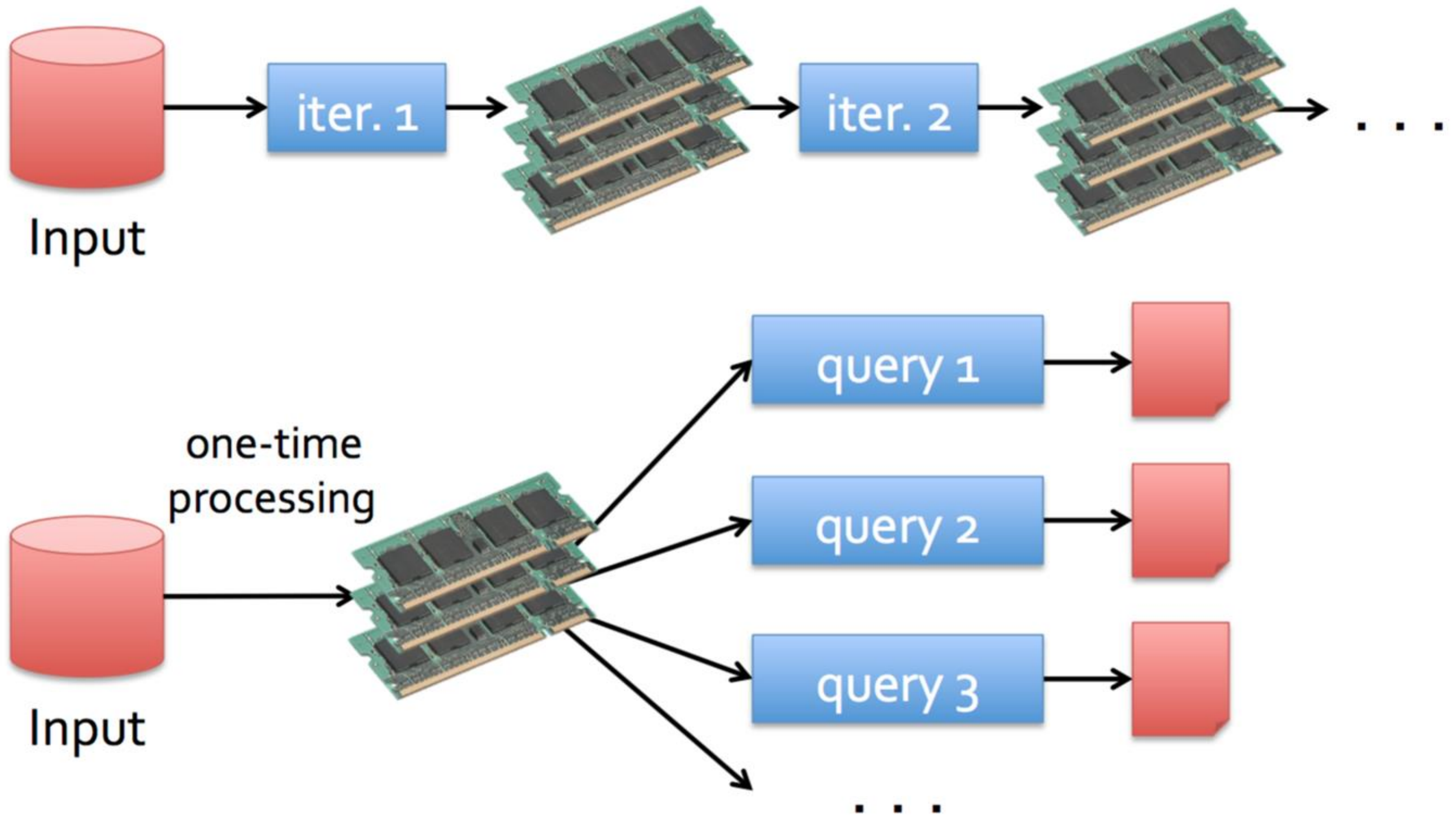
Example: PageRank

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

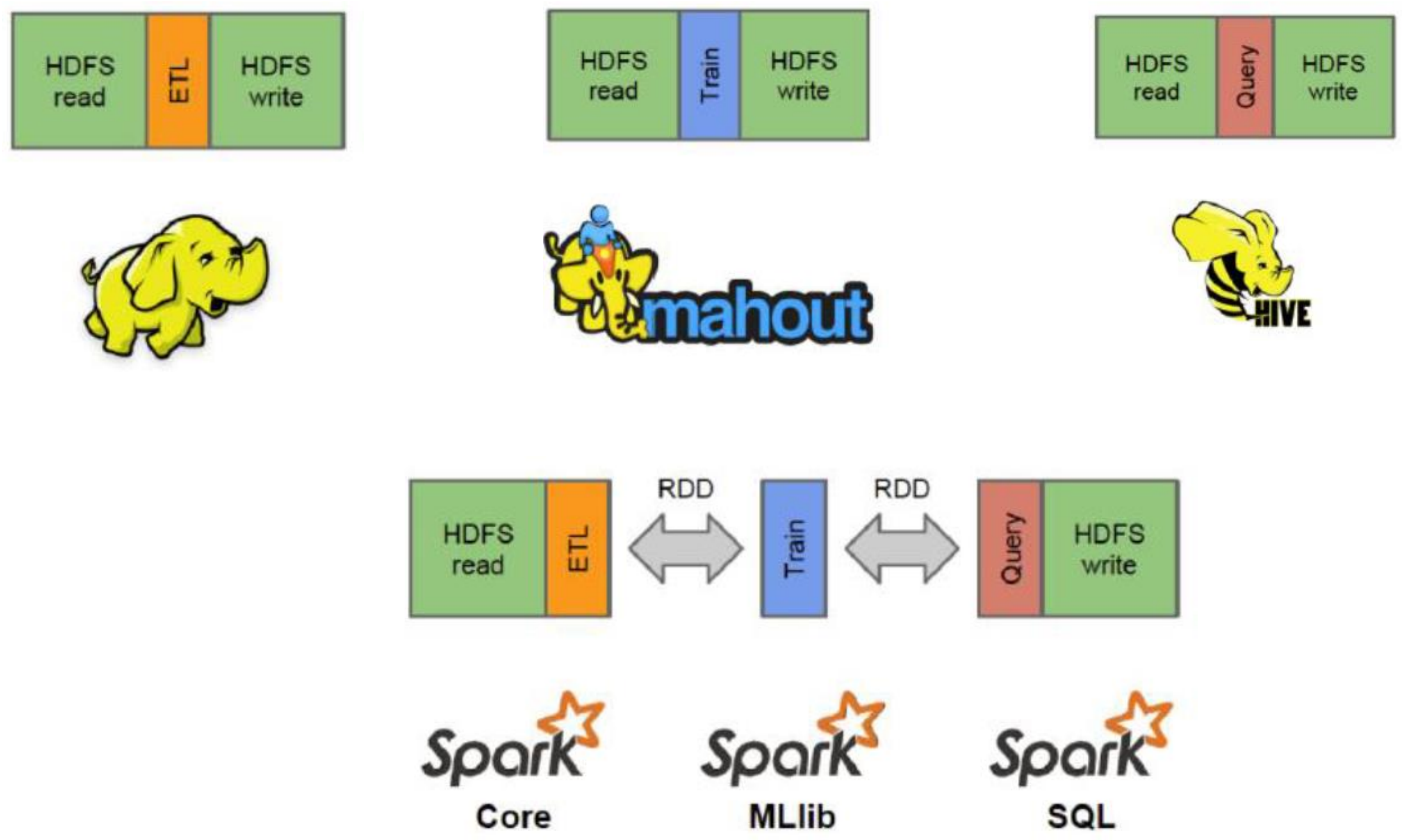
Repeatedly multiply sparse matrix and vector
Requires repeatedly hashing together page adjacency lists and rank vector



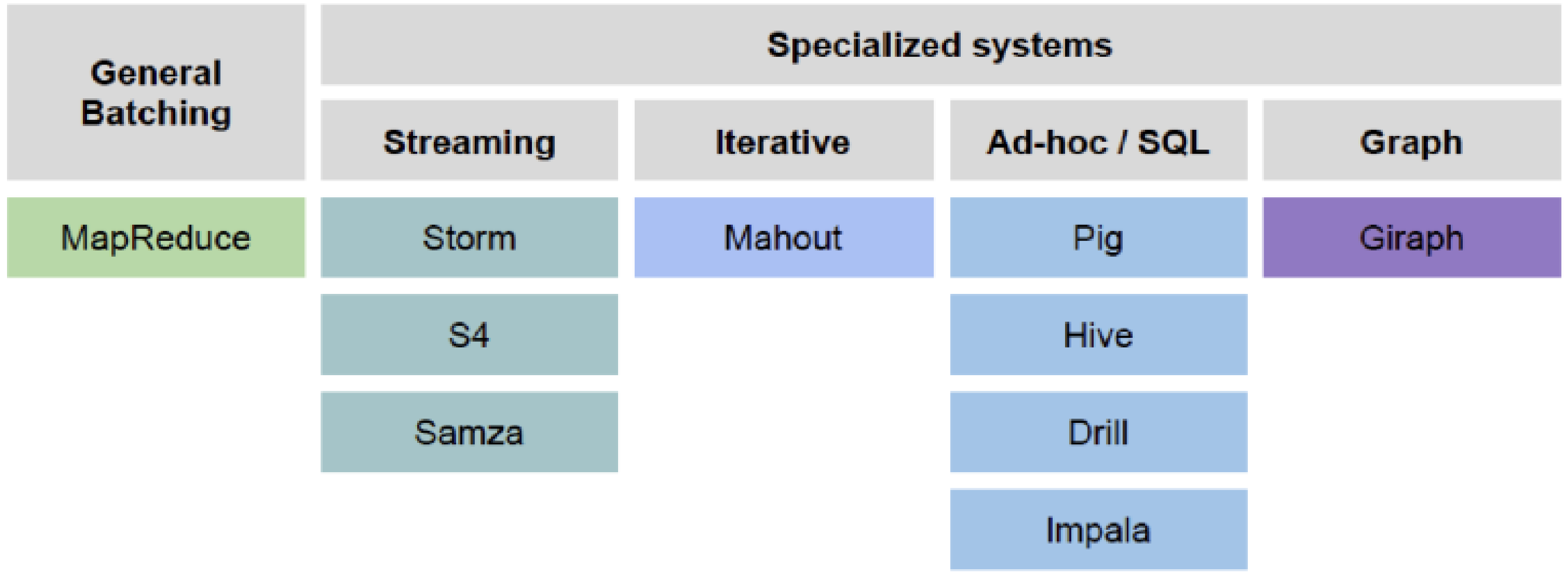
Goal: In-Memory Data Sharing



Spark vs MapReduce: Data Flow



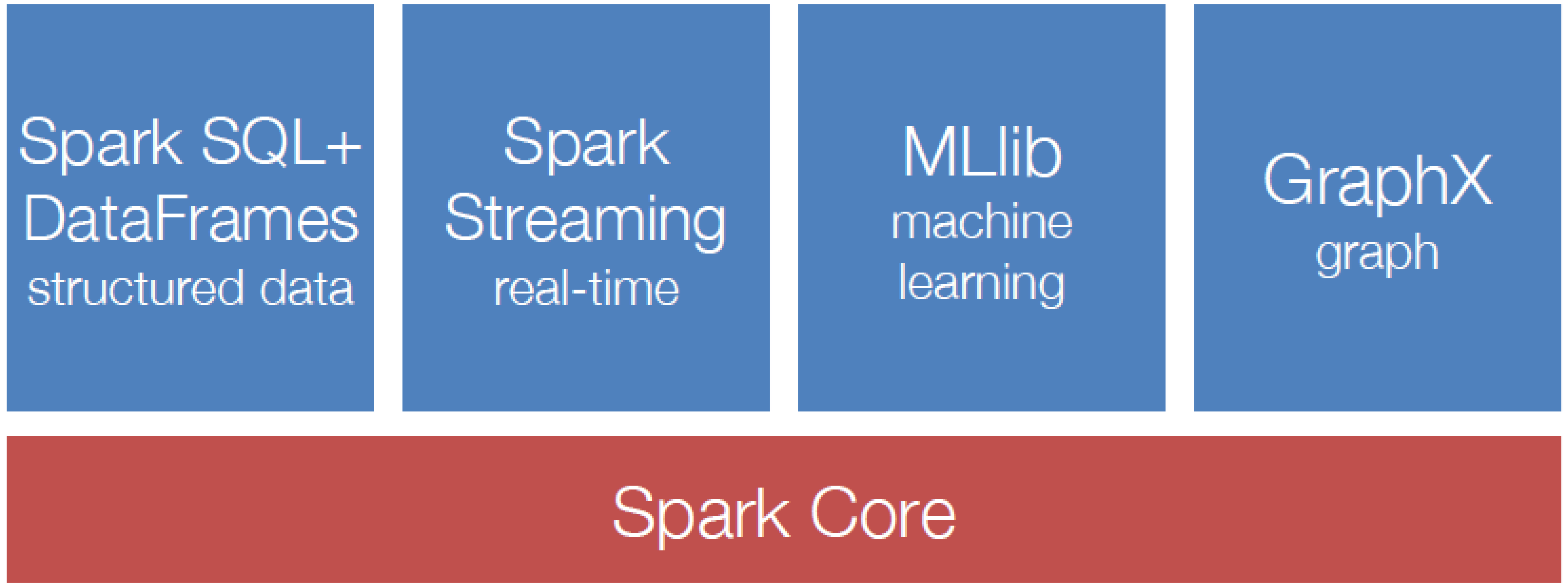
Apache Hadoop: No Unified Vision



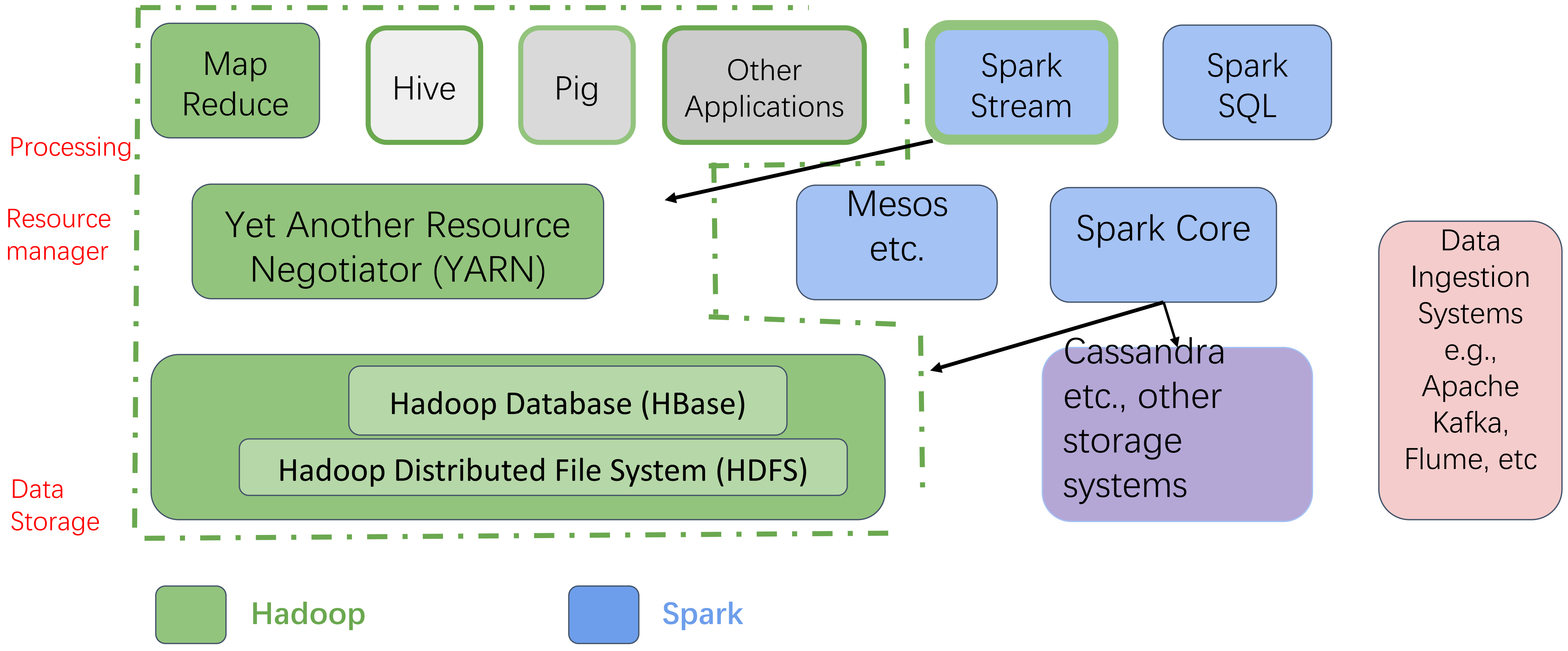
General Batching	Specialized systems			
	Streaming	Iterative	Ad-hoc / SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	

- Sparse Modules
- Diversity of APIs
- Higher Operational Costs

Spark Ecosystem: A Unified Pipeline


Spark SQL+ DataFrames structured data
Spark Streaming real-time
MLlib machine learning
GraphX graph
Spark Core

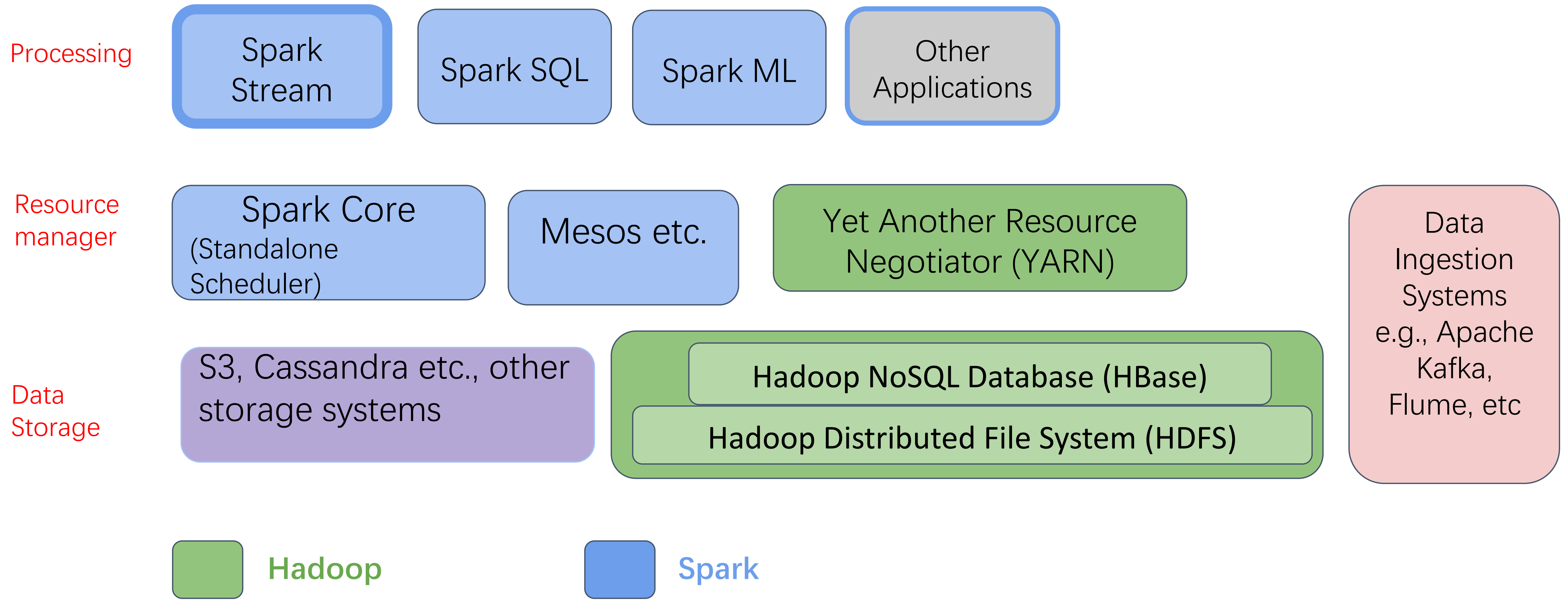
Apache Hadoop & Apache Spark



Processing
Resource manager
Data Storage
Map Reduce
Hive
Pig
Other Applications
Spark Stream
Spark SQL
Yet Another Resource Negotiator (YARN)
Mesos etc.
Spark Core
Data Ingestion Systems e.g., Apache Kafka, Flume, etc
Hadoop Database (HBase)
Hadoop Distributed File System (HDFS)
Cassandra etc., other storage systems
Hadoop
Spark

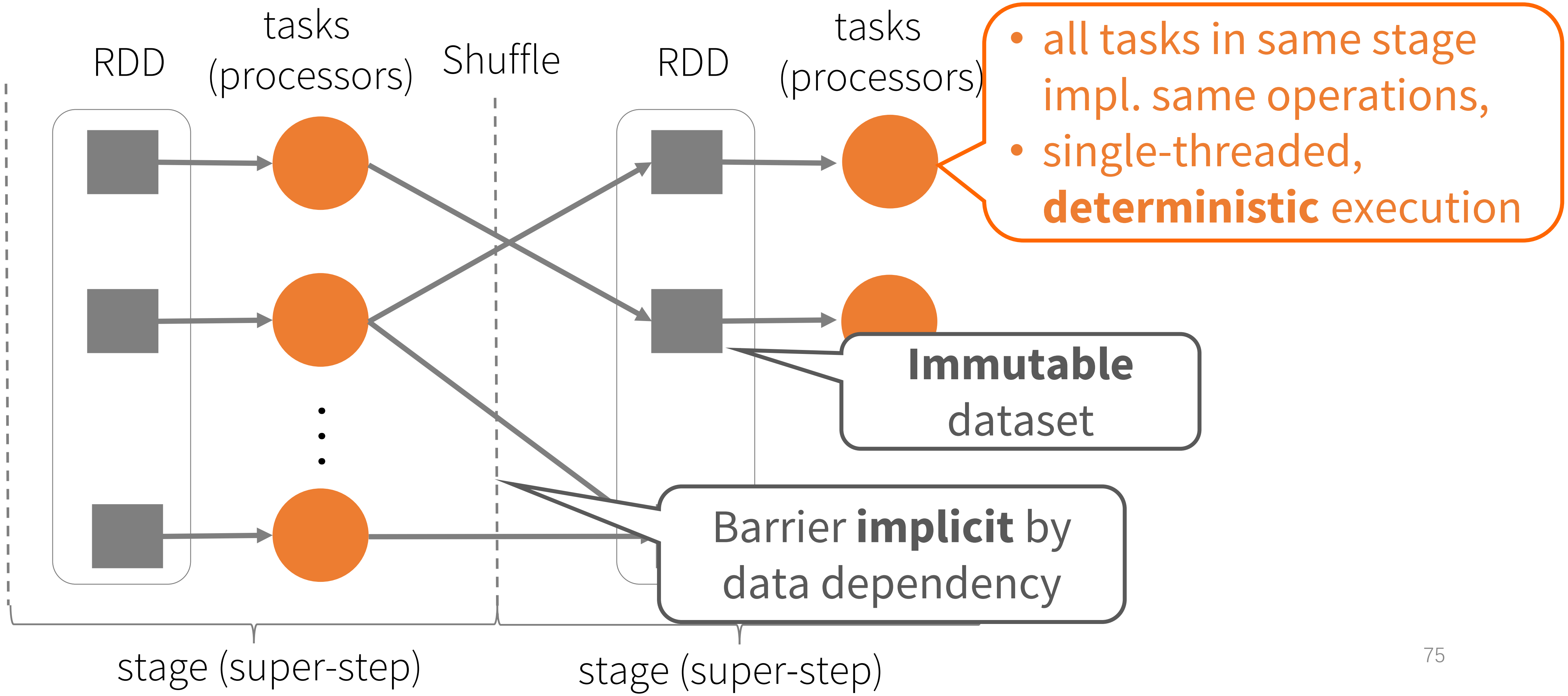
Apache Spark

** Spark can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos or YARN)


Processing
Spark Stream
Spark SQL
Spark ML
Other Applications
Resource manager
Spark Core (Standalone Scheduler)
Mesos etc.
Yet Another Resource Negotiator (YARN)
Data Ingestion Systems e.g., Apache Kafka, Flume, etc
Data Storage
S3, Cassandra etc., other storage systems
Hadoop NoSQL Database (HBase)
Hadoop Distributed File System (HDFS)
Hadoop
Spark

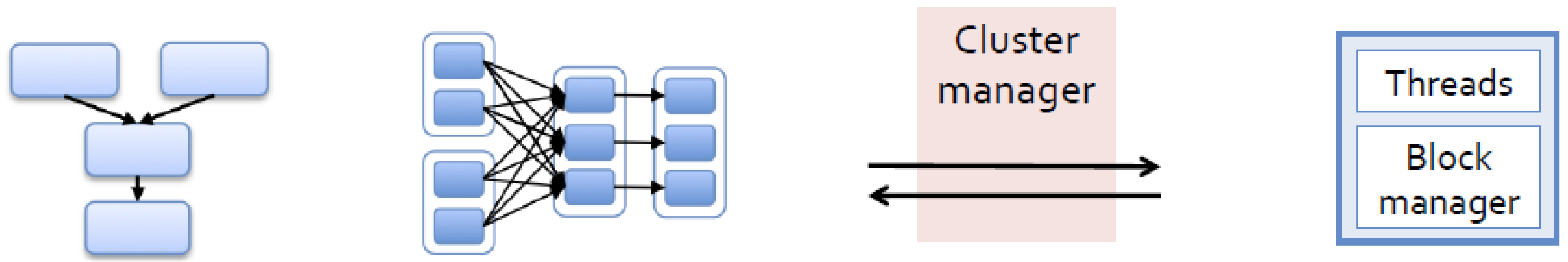
Spark

Extends a programming language with a distributed collection data-structure:
"Resilient distributed datasets" (RDD)


RDD
tasks (processors)
Shuffle
RDD
tasks (processors)
• all tasks in same stage impl. same operations,
• single-threaded, **deterministic** execution
Immutable dataset
Barrier **implicit** by data dependency
stage (super-step)
stage (super-step)

Lifetime of a Job in Spark

RDD Objects | **DAG Scheduler** | **Task Scheduler** | **Worker**



RDD Objects

```
 rdd1.join(rdd2)
.groupBy(...)
.filter(...)
```

Build the operator DAG

DAG Scheduler

Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

Task Scheduler

Launch tasks via Master

Retry failed and straggler tasks

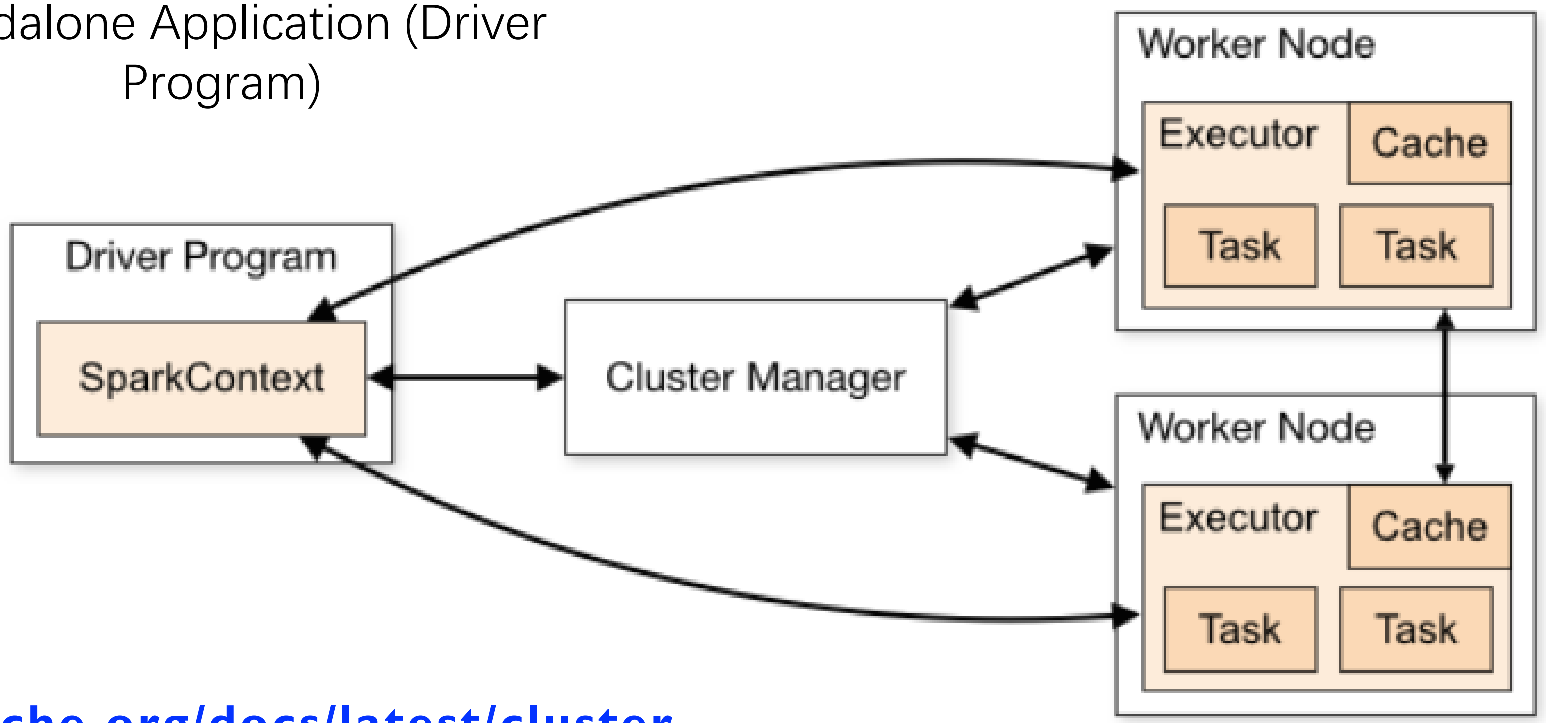
Worker

Execute tasks

Store and serve blocks

Spark程序执行

Standalone Application (Driver Program)



spark.apache.org/docs/latest/cluster-overview.html

Spark, a generalization of MapReduce

- DAG computation model vs two stage computation model (Map and Reduce)

- Disk-based vs. memory-optimized

Spark Essentials: SparkContext

- First thing that a Spark program does is create a `SparkContext` object, which tells Spark how to access a cluster
  - Since Spark 2.0, mostly use SparkSession as most of the methods available in SparkContext are also present in SparkSession
- In the shell for either Scala or Python, this is the `sc` variable, which is created automatically
- Other programs must use a constructor to instantiate a new `SparkContext`
- Then in turn `SparkContext` gets used to create other variables

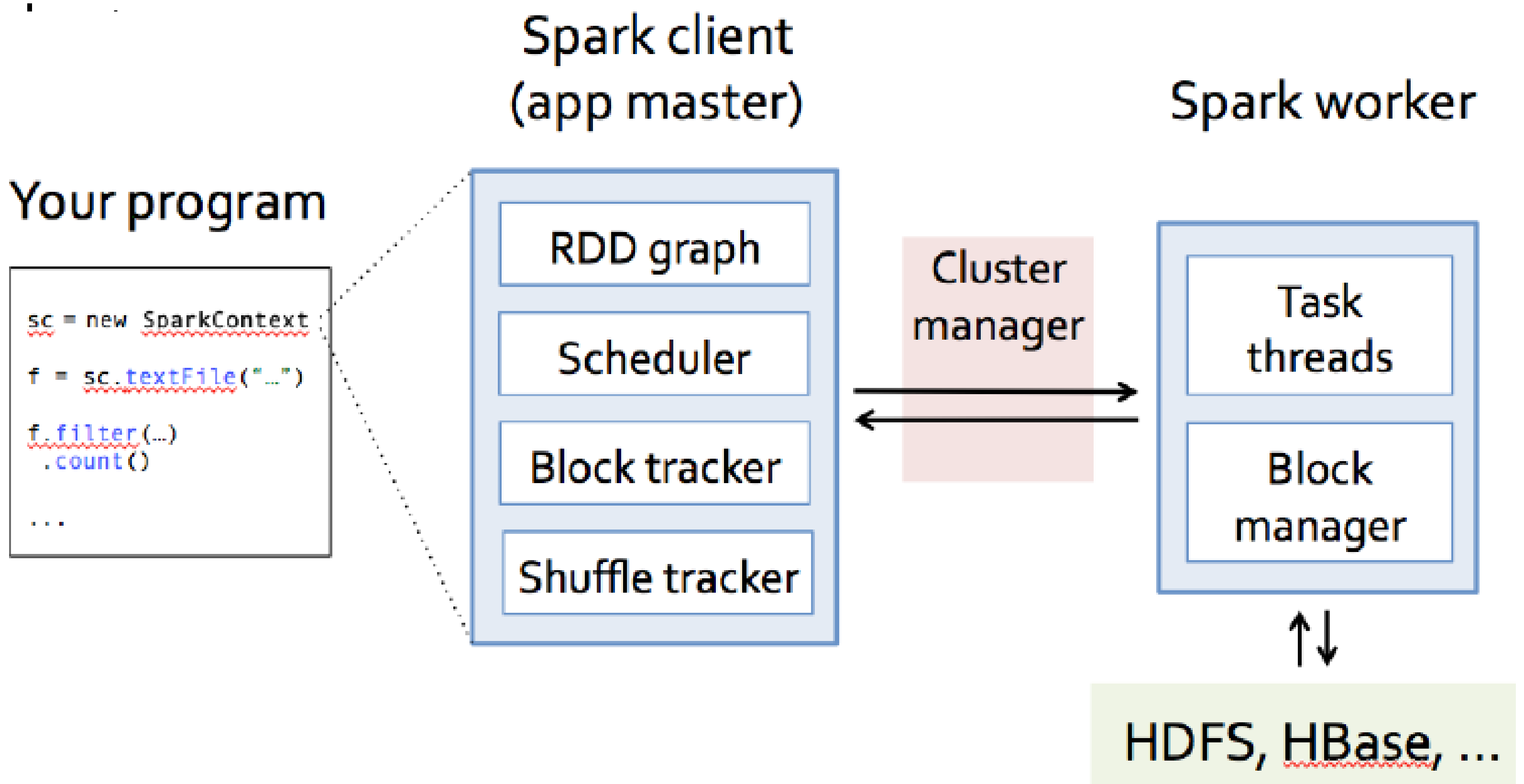
Spark Essentials: Master

- The `master` parameter for a `SparkContext` determines which cluster to use

master	***description***
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to # cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

Spark Essentials: SparkContext

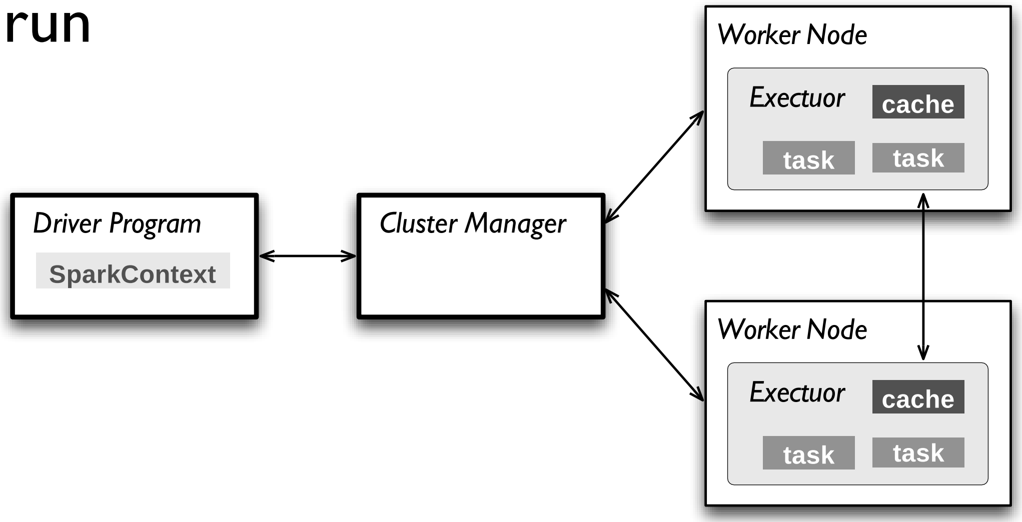
- Standalone applications → Driver code → Spark Context
- Spark context works as a client and represents connection to a Spark



```
sc = new SparkContext
f = sc.textFile("…")
f.filter(…)
 .count()
...
```

Spark Essentials: Master

1. connects to a *cluster manager* which allocate resources across applications
2. acquires *executors* on cluster nodes – worker processes to run computations and store data
3. sends *app code* to the executors
4. sends *tasks* for the executors to run



Spark Essentials: Resilient Distributed Dataset

RDD is the fundamental unit of data in Spark**:** An *Immutable* collection of objects (or records, or elements) that can be operated on "in parallel" (spread across a cluster)

Resilient -- if data in memory is lost, it can be recreated

- Recover from node failures
- An RDD keeps its lineage information → it can be recreated from parent RDDs

Distributed -- processed across the cluster

- Each RDD is composed of one or more partitions → (more partitions – more parallelism)

Dataset -- initial data can come from a file or be created

RDD

Key Idea: Write applications in terms of transformations on distributed datasets

- Collections of objects spread across a Memory caching layer(cluster) that stores data in a distributed, fault-tolerant cache
- Can fall back to disk when dataset does not fit in memory
- Built through parallel transformations (map, filter, group-by, join, etc)
- Automatically rebuilt on failure
- Controllable persistence (e.g. caching in RAM)

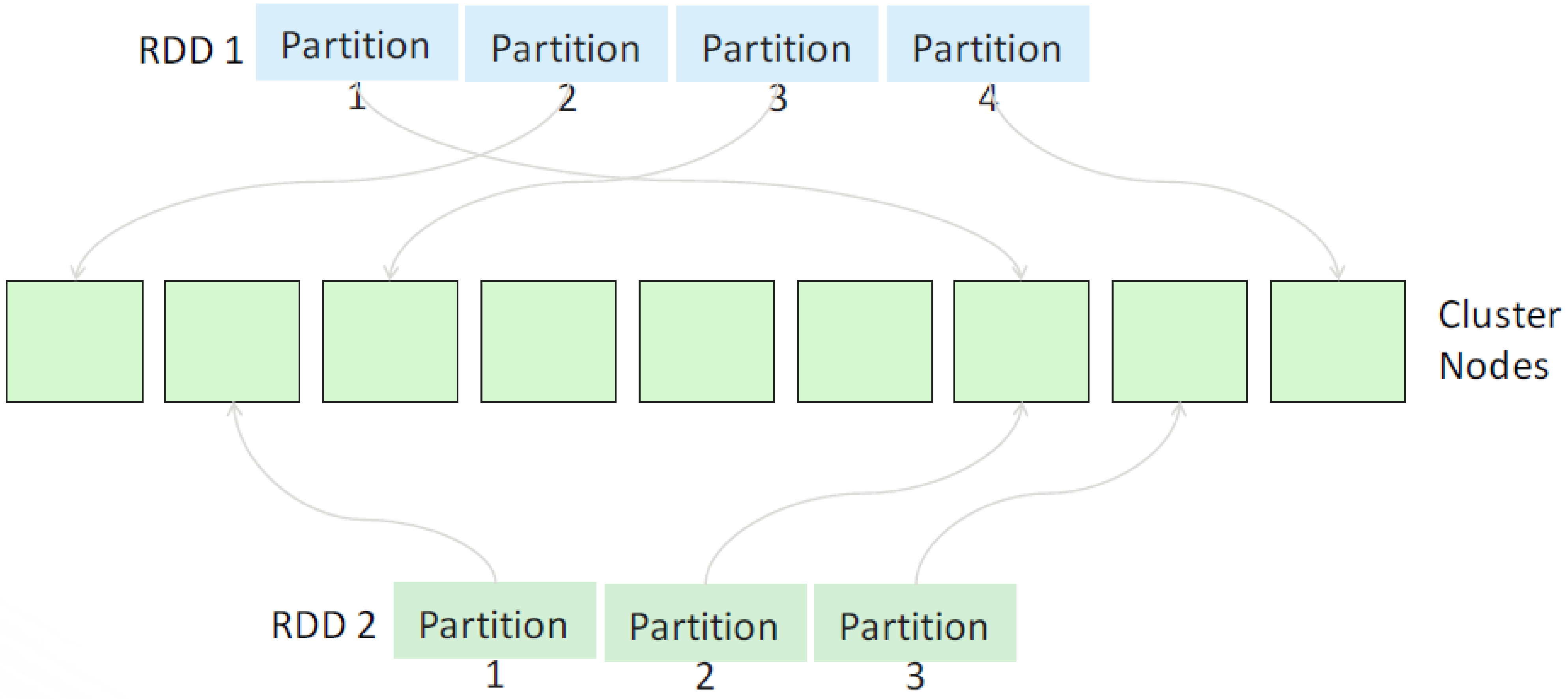
RDDs -- Immutability

- Immutability → lineage information → can be recreated at any time → Fault-tolerance
- Avoids data inconsistency problems ← no simultaneous updates → Correctness
- Easily live in memory as on disk → Caching → Safe to share across processes/tasks → Improves performance
- Tradeoff: (**Fault-tolerance & Correctness**) vs (**Disk Memory & CPU**)

Spark Essentials: RDD

- 两种类型:
  - *parallelized collections* – take an existing Scala collection and run functions on it in parallel
  - *Hadoop datasets* – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop
- Spark可以从如下数据源创建RDD：HDFS文件；Hadoop支持的其它存储系统，如：本地文件系统，Amazon S3，Hypertable，Hbase等.
- Spark支持文本文件，SequenceFiles, 任何Hadoop `InputFormat`，目录或glob (e.g. `/data/201404*`)

RDD and Partitions



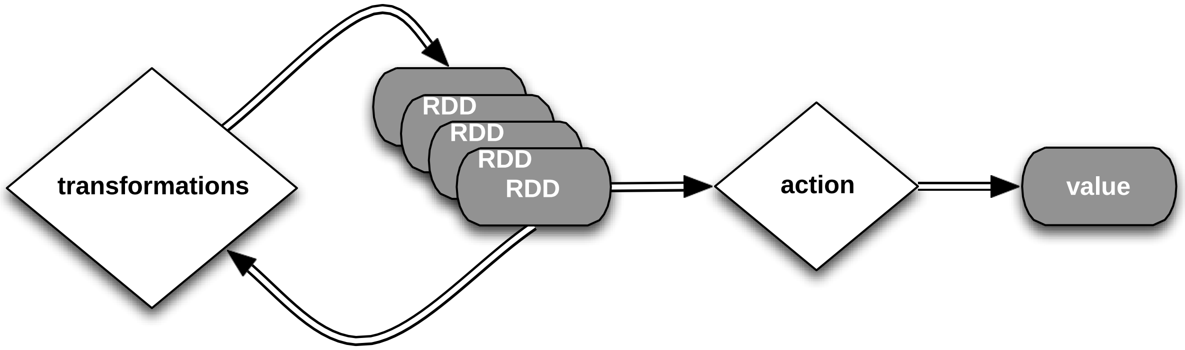
Spark Programming

Creating RDDs

```
# Turn a Python collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Use existing Hadoop InputFormat (Java/Scala only)
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Spark Essentials: Transformations

- 对RDD变换的一组操作。
- 如同关系代数，变换创建新的RDD
- Spark的所有变换都是延迟执行：并不会立即执行，它们记录对同一数据集要做的变换
  - 优化计算
- Examples: map(), filter(), groupByKey(), sortByKey(), etc.



Spark Essentials: Transformations

transformation	*description*
map(*func***)**	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func***)**	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func***)**	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed***)**	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset***)**	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks***]))**	return a new dataset that contains the distinct elements of the source dataset

Spark Essentials: Transformations

transformation	*description*
groupByKey([*numTasks***])**	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey(*func***, [***numTasks***])**	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending***], [***numTasks***])**	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset***, [***numTasks***])**	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cogroup(*otherDataset***, [***numTasks***])**	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
cartesian(*otherDataset***)**	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

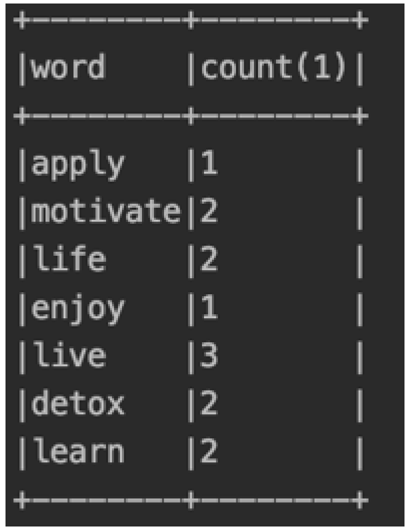
Spark Programming

Basic Transformations

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x) // {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) // {4}
```

Word Count

```
val conf= new SparkConf().setAppName("test").setMaster("local")
val sc =new SparkContext(conf)
val spark=SparkSession.builder().config(conf).getOrCreate()
sc.setLogLevel("ERROR")
val line1 = "live life enjoy detox"
val line2="learn apply live motivate"
val line3="life detox motivate live learn"
val rdd =sc.parallelize(Array(line1,line2,line3))
val rdd1 = rdd.flatMap(=>x.split(" "))
import  spark.implicits x._
val df = rdd1.toDF("word")
df.createOrReplaceTempView("tempTable")
val rslt=spark.sql("select word,COUNT(1) from tempTable GROUP BY word ")
rslt.show(1000,false)
```



```
+--------+--------+
|word    |count(1)|
+--------+--------+
|apply   |1       |
|motivate|2       |
|life    |2       |
|enjoy   |1       |
|live    |3       |
|detox   |2       |
|learn   |2       |
+--------+--------+
```

RDD Actions

- Apply transformation chains on RDDs, eventually performing some additional operations (e.g., counting)
- Some actions only store data to an external data source (e.g. HDFS), others fetch data from the RDD (and its transformation chain) upon which the action is applied, and convey it to the driver
- Some common actions
  - count() – return the number of elements
  - take(*n*) – return an array of the first *n* elements
  - collect()– return an array of all elements
  - saveAsTextFile(*file*) – save to text file(s)

Spark Essentials: Actions

action	***description***
`reduce(`*`func`***`)`**	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
`collect()`	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
`count()`	return the number of elements in the dataset
`first()`	return the first element of the dataset – similar to *take(1)*
`take(`*`n`***`)`**	return an array with the first `n` elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
`takeSample(`*`withReplacement`***`,`** ***`fraction`***`,` ***`seed`***`)`	return an array with a random sample of `num` elements of the dataset, with or without replacement, using the given random number generator seed

Spark Essentials: Actions

action	*description*
`saveAsTextFile(``path`**`)`**	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(``path`**`)`**	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a \`Map\` of `(K, Int)` pairs with the count of each key
`foreach(``func`**`)`**	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Spark Programming

Basic Actions

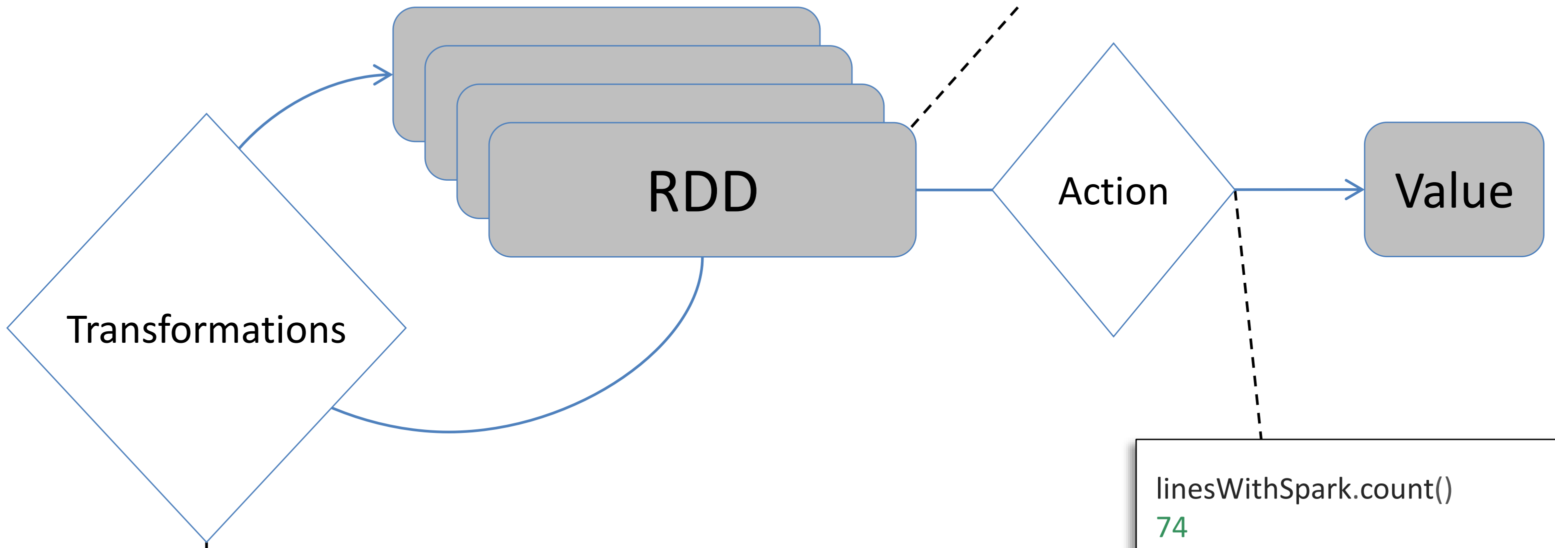
```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local
collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2) # => [1, 2]

# Count number of elements
nums.count() # => 3

# Merge elements with an associative
function
nums.reduce(lambda x, y: x + y) # => 6
```

Some Key-Value Operations

```
pets = sc.parallelize([(“cat”, 1), (“dog”, 1),
(“cat”, 2)])

pets.reduceByKey(lambda x, y: x + y) # =>
{(cat, 3), (dog, 1)}

pets.groupByKey() # => {(cat, [1, 2]), (dog,
[1])}

pets.sortByKey() # => {(cat, 1), (cat, 2),
(dog, 1)}
```

Working With RDDs

```
textFile = sc.textFile("SomeFile.txt")
```



```
linesWithSpark.count()
74

linesWithSpark.first()
# Apache Spark
```

```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

```scala
package com.sparkbyexamples.spark.rdd
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.SparkSession

object WordCountExample {
  def main(args:Array[String]): Unit = {
    val spark:SparkSession = SparkSession.builder()
      .master("local[3]")
      .appName("SparkByExamples.com")
      .getOrCreate()
    val sc = spark.sparkContext
    val rdd:RDD[String] = sc.textFile("src/main/resources/test.txt")
    println("initial partition count:"+rdd.getNumPartitions)
    val reparRdd = rdd.repartition(4)
    println("re-partition count:"+reparRdd.getNumPartitions)
    val rdd2 = rdd.flatMap(f=>f.split(" "))
    //Create a Tuple by adding 1 to each word
    val rdd3:RDD[(String,Int)]= rdd2.map(m=>(m,1))
    val rdd4 = rdd3.filter(a=> a._1.startsWith("a"))
    val rdd5 = rdd3.reduceByKey(_ + _)
    //Swap word,count and sortByKey transformation
    val rdd6 = rdd5.map(a=>(a._2,a._1)).sortByKey()
    rdd6.foreach(println)
    println("Count : "+rdd6.count())
    val firstRec = rdd6.first()
    println("First Record : "+firstRec._1 + ","+ firstRec._2)
    val datMax = rdd6.max()
    println("Max Record : "+datMax._1 + ","+ datMax._2)
    val totalWordCount = rdd6.reduce((a,b) => (a._1+b._1,a._2))
    println("dataReduce Record : "+totalWordCount._1)
    val data3 = rdd6.take(3)
    data3.foreach(f=>{
      println("data3 Key:"+ f._1 +", Value:"+f._2)
    })

    val data = rdd6.collect()
    data.foreach(f=>{
      println("Key:"+ f._1 +", Value:"+f._2)
    })
    rdd5.saveAsTextFile("c:/tmp/wordCount")
  }
}
```

Spark Essentials: Persistence

- Spark can *persist* (or cache) a dataset in memory across operations
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster
- The cache is *fault-tolerant:* if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

Spark Essentials: Persistence

transformation	*description*
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

Spark Essentials: Broadcast Variables

- Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

For example, to give every node a copy of a large input dataset efficiently

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

Spark Essentials: Accumulators

- Accumulators are variables that can only be “added” to through an *associative* operation

- Used to implement counters and sums, efficiently in parallel

- Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

- Only the driver program can read an accumulator’s value, not the tasks

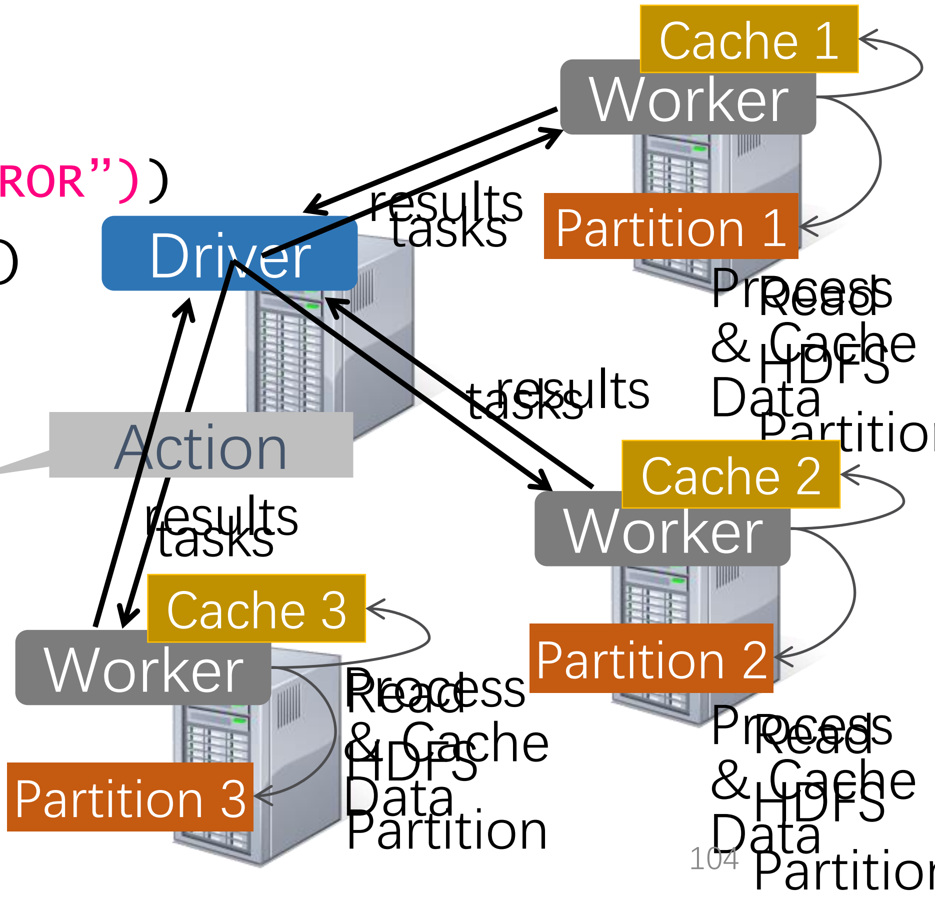
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Base RDD sformed RDD

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```

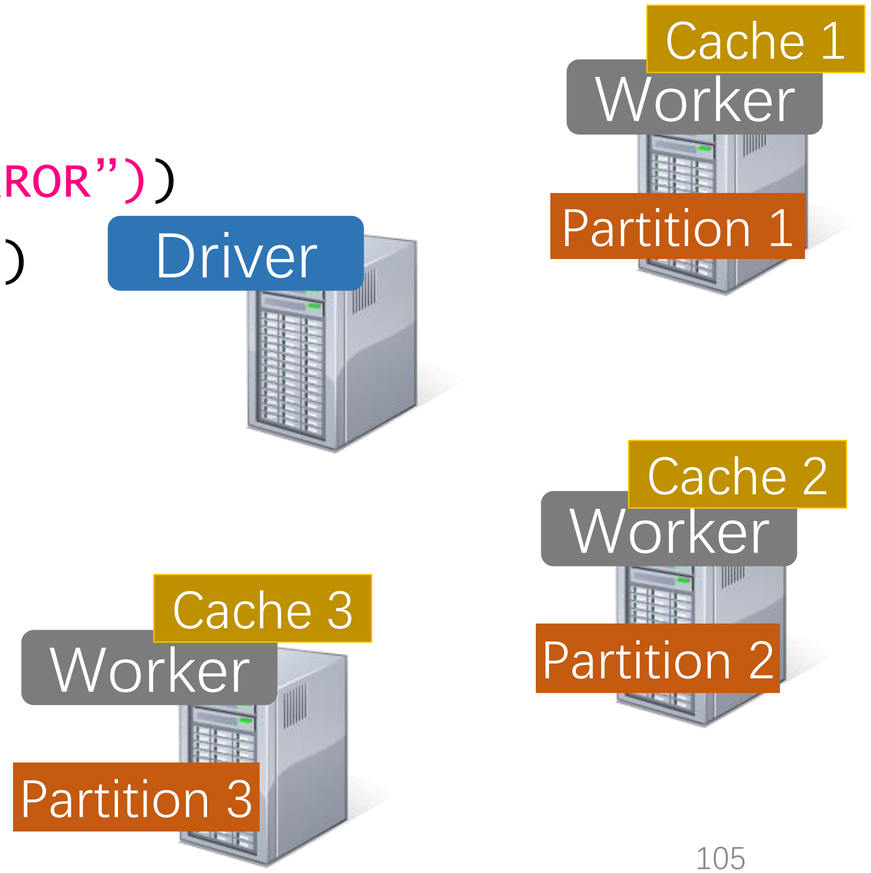
Action



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

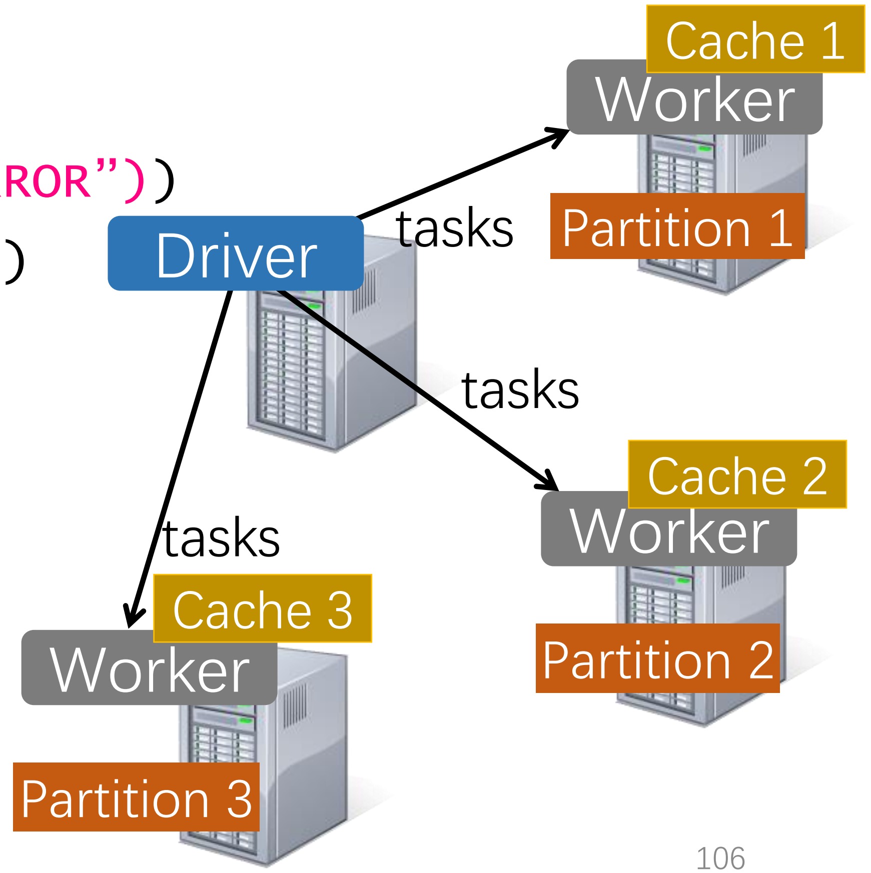
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

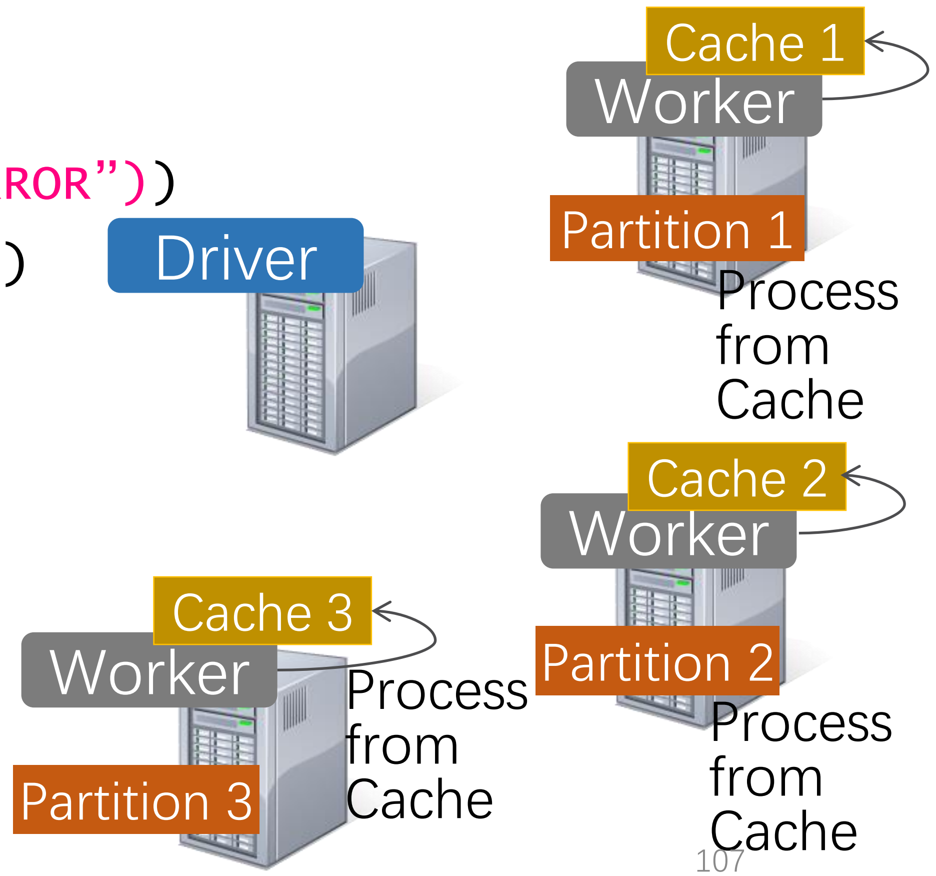
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

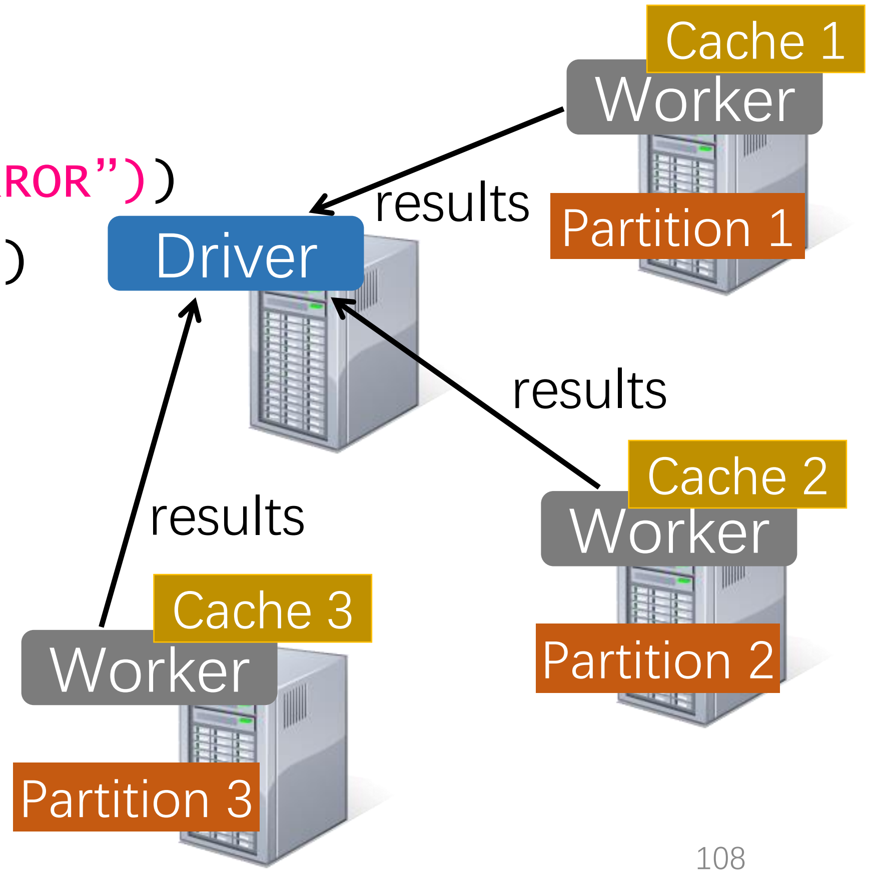
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Example: Log Mining

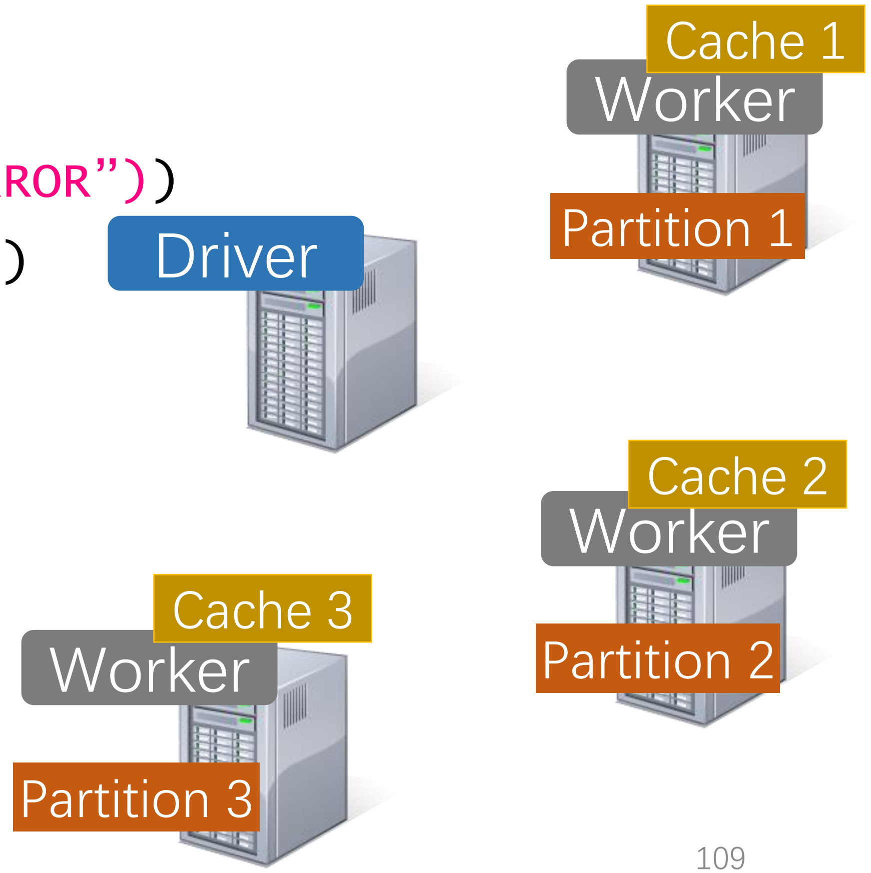
Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

Cache your data → Faster Results

Full-text search of Wikipedia

- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk



Language Support

Python

```
lines = sc.textFile(...)
lines.filter(lambda s: “ERROR” in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains(“ERROR”)).count()
```

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains(“error”);
  }
}).count();
```

Standalone Programs

- Python, Scala, & Java

Interactive Shells

- Python & Scala

Performance

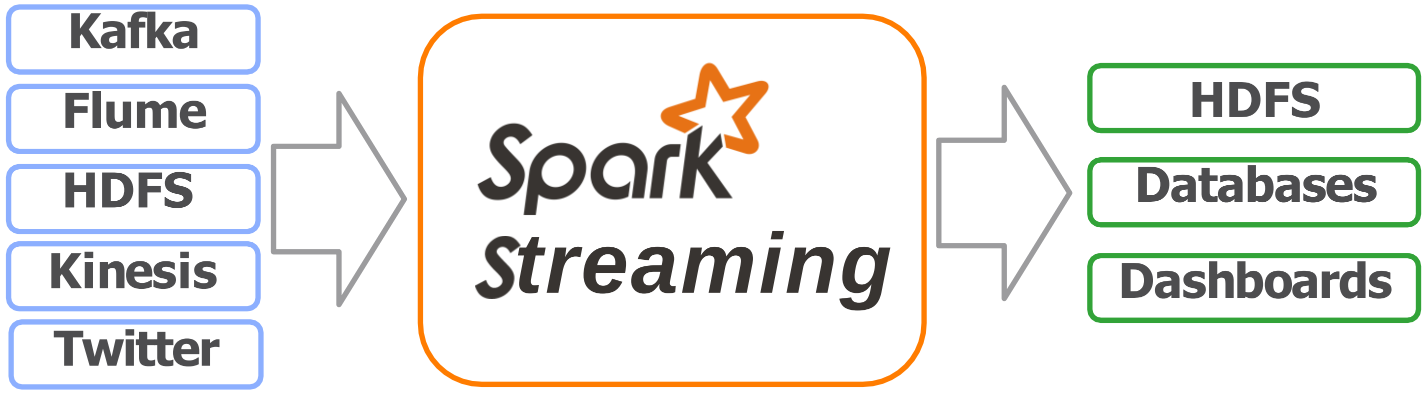
- Java & Scala are faster due to static typing
- …but Python is often fine

Spark 2.0

- Dataframes/datasets instead of RDDs
  - Like tables in SQL
  - Far more efficient:
    - Can directly access any field (dramatically reduce I/O and serialization/deserialization)
    - Can use column oriented access
- Dataframe APIs (e.g., Python, R, SQL) use same optimizer: Catalyst
- New libraries or old libraries revamped to use dataframe APIs
  - Spark Streaming → Structured Streaming
  - GraphX

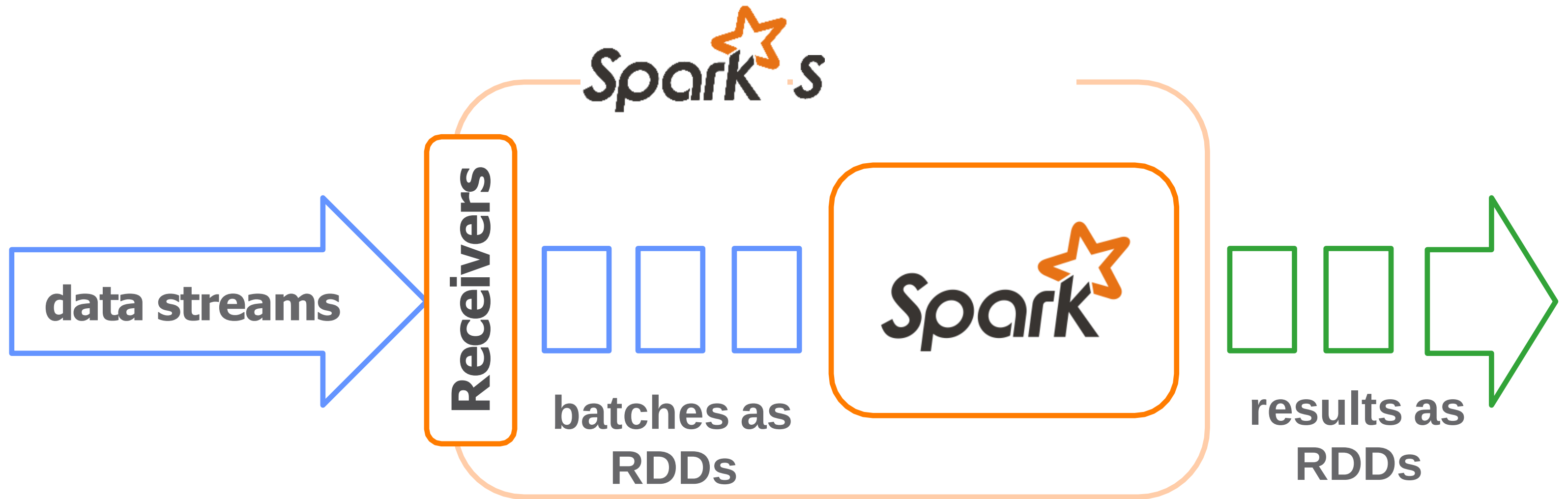
What is Spark Streaming?

- Receive data streams from input sources, process them in a cluster, push out to databases/ dashboards
- Scalable, fault-tolerant, second-scale latencies



How does Spark Streaming work?

- Chop up data streams into batches of few secs
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Processed results are pushed out in batches



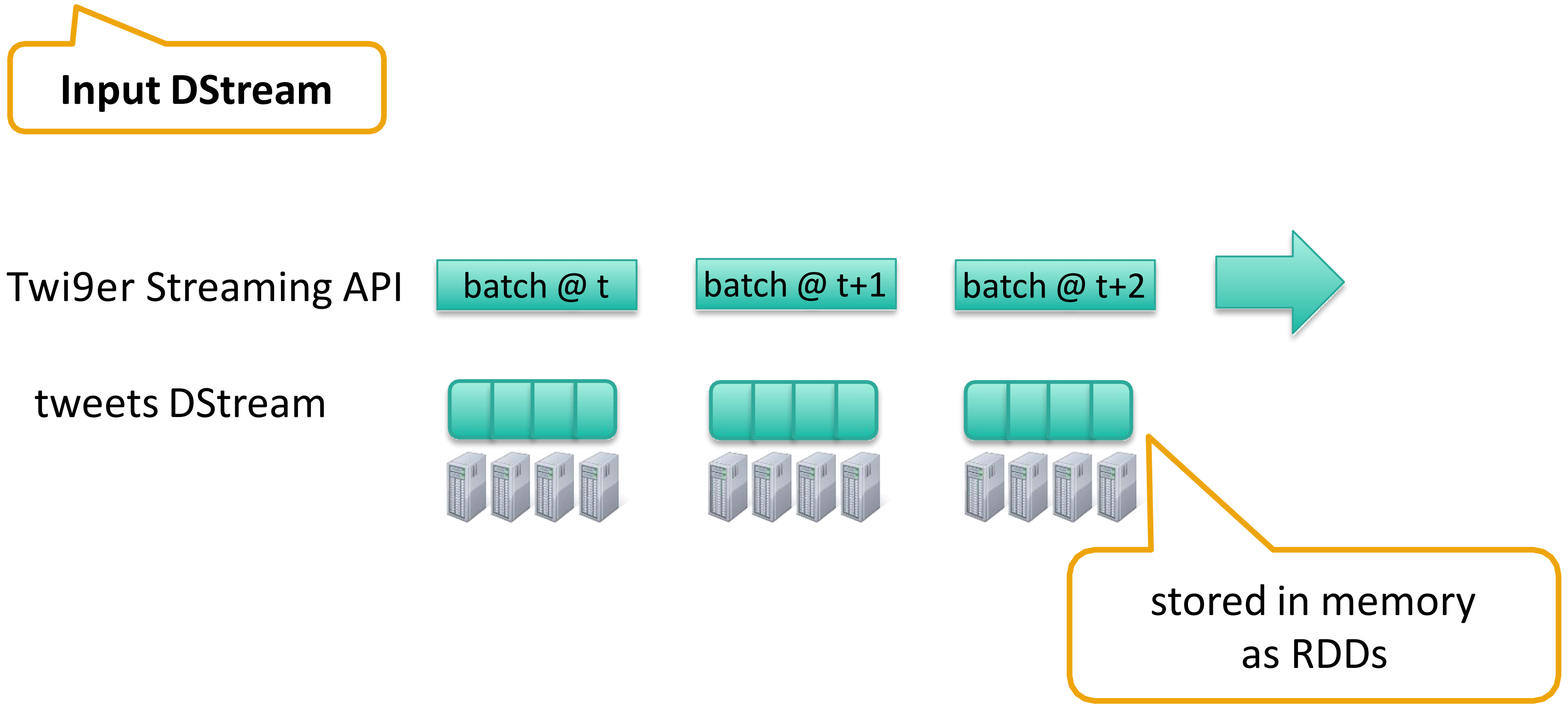
Spark Streaming Programming Model

- Discretized Stream (DStream)
    - Represents a stream of data
    - Implemented as a sequence of RDDs

- DStreams API very similar to RDD API
    - Functional APIs in Scala, Java
    - Create input DStreams from different sources
    - Apply parallel operations

Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
```

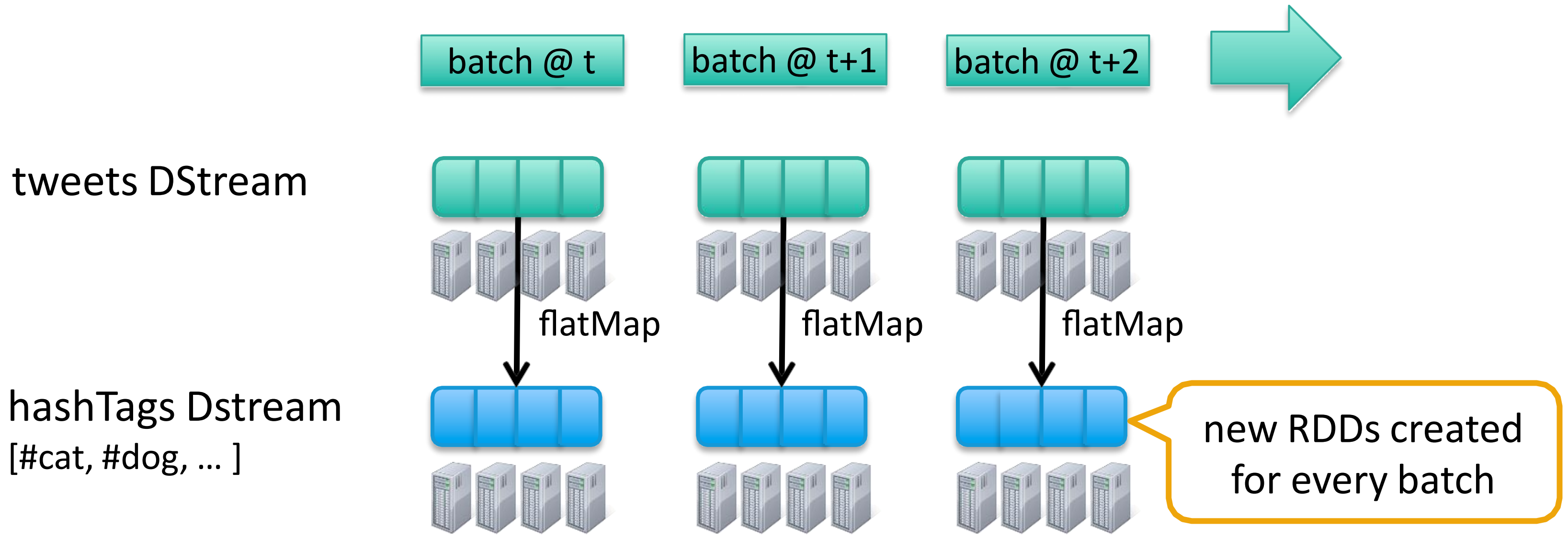


Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
```

transformed DStream

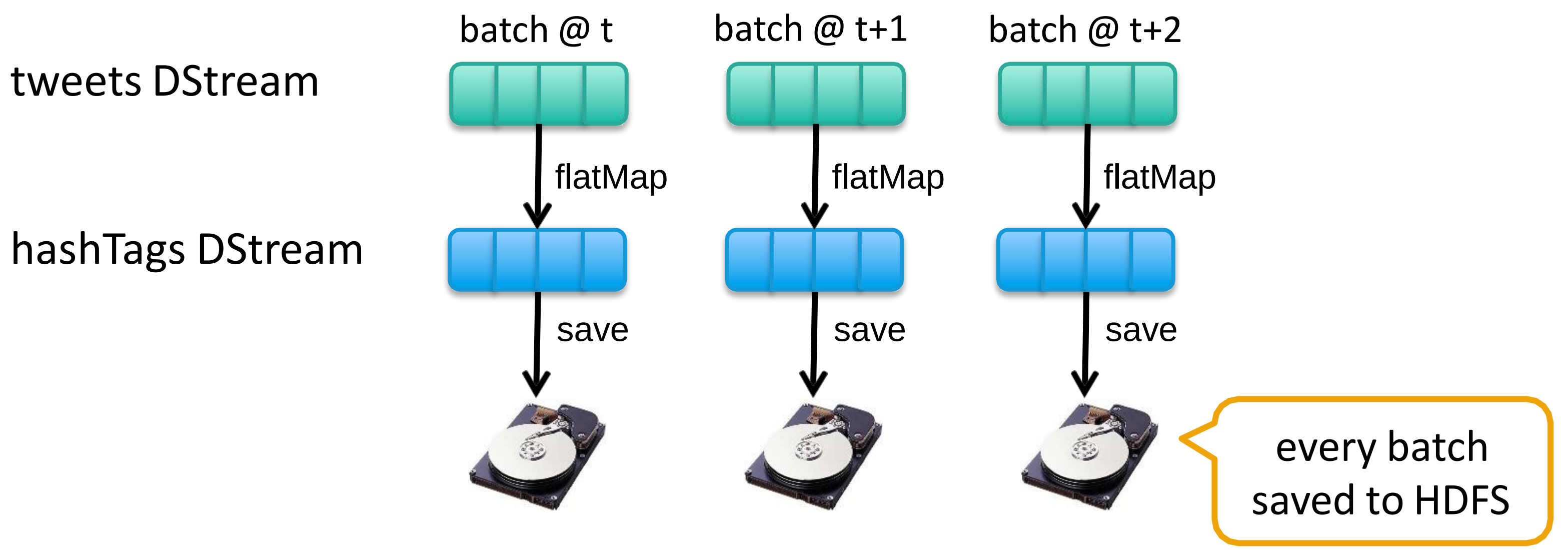
transforma0on: modify data in one DStream to create another DStream



Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

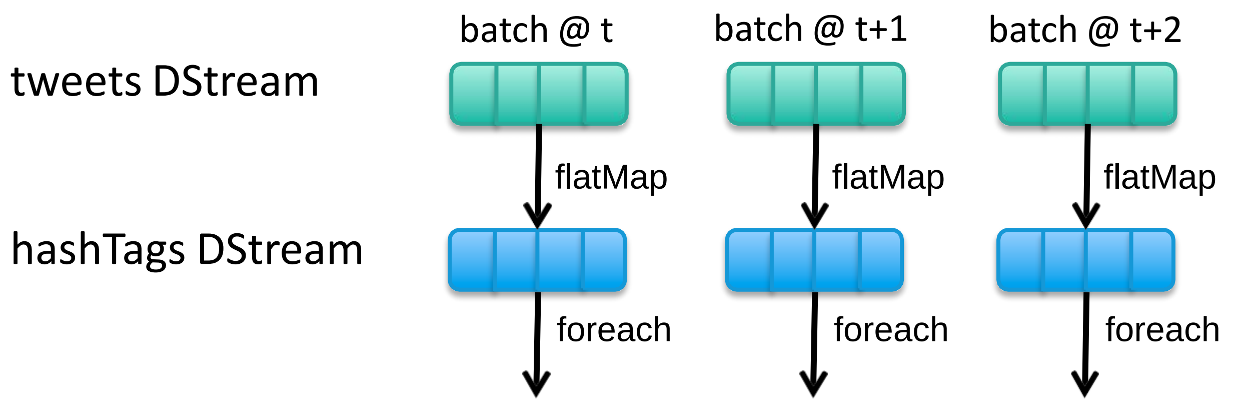
output opera0on: to push data to external storage



Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreachRDD(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



Write to a database, update analyOcs
UI, do whatever you want

Discussion

	OpenMP/Cilk	MPI	MapReduce / Spark
Environment, Assumptions	Single node, multiple core, shared memory	Supercomputers Sophisticate programmers High performance Hard to scale hardware	Commodity clusters Java programmers Programmer productivity Easier, faster to scale up cluster
Computation Model	Fine-grained task parallelism	Message passing	Data flow / BSP
Strengths	Simplifies parallel programming on multi-cores	Can write very fast asynchronous code	Fault tolerance
Weaknesses	Still pretty complex, need to be careful about race conditions	Fault tolerance Easy to end up with non-deterministic code (if not using barriers)	Not as high performance as MPI