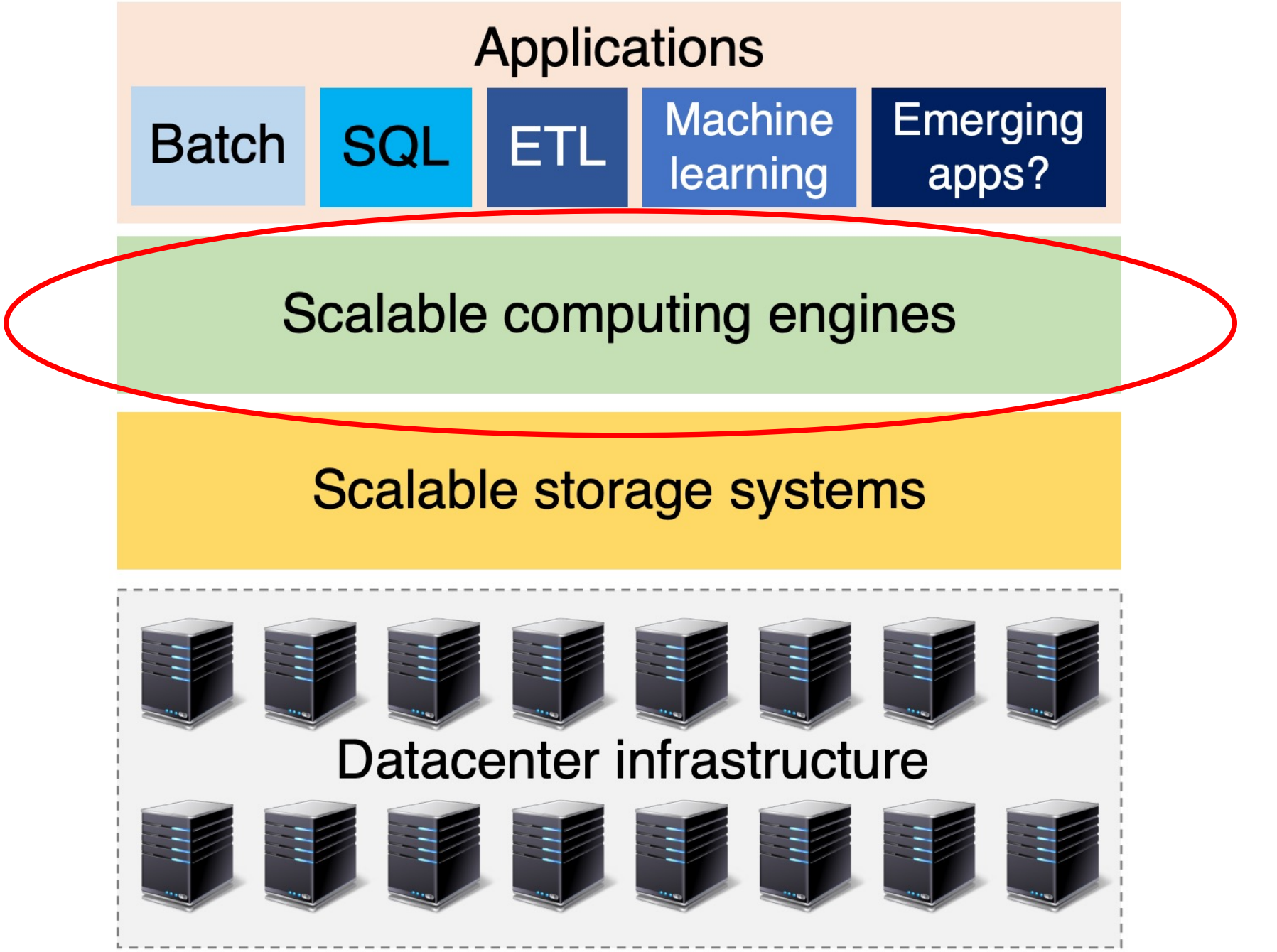
Cloud Computing
MapReduce

Minchen Yu
SDS@CUHK-SZ
Fall 2024

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

SCHOOL OF
DATA SCIENCE
數據科學學院


Applications
Batch
SQL
ETL
Machine learning
Emerging apps?
Scalable computing engines
Scalable storage systems
Datacenter infrastructure

The big picture

Datasets are too big to process using a single computer

数据集太大，无法使用单台计算机进行处理

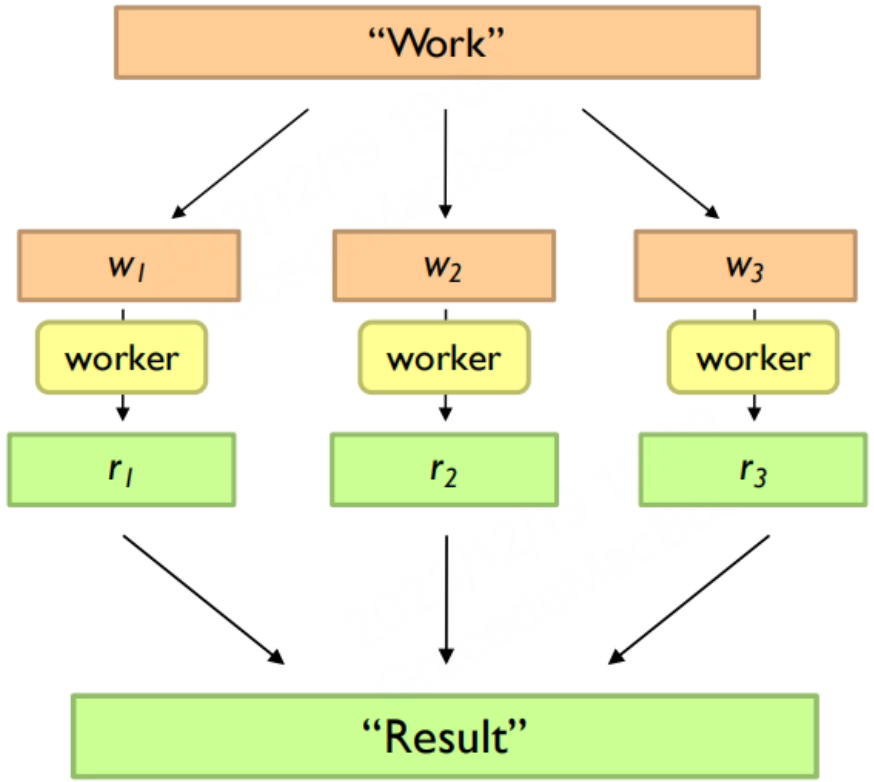
Good parallel processing engines are rare (back then in the late 90s)

良好的并行处理引擎非常罕见（90年代末）

Want a general and easy-to-use parallel processing framework

想要一个通用且易于使用的并行处理框架

Divide and conquer 分而治之



Partition

Combine

Parallelization challenges

How do we assign work units to workers? 我们如何为工人分配工作？

What if we have more work units than workers? 如果我们的工作比工人多怎么办？

What if workers need to share partial results? 如果工人需要分享部分结果怎么办？

How do we aggregate partial results? 我们如何汇总部分结果？

How do we know all the workers have finished? 我们怎么知道所有的工人都已经完成？

What if workers die? 如果工人死亡怎么办？

What's the common theme of all of these problems?

Common theme?

Parallelization problem arise from

工人之间的沟通（例如，状态交换）

- communication between workers (e.g., state exchange)
- access to shared resources (e.g., data)

访问共享资源（例如数据）

Thus, we need a synchronization mechanism

因此我们需要一个同步机制

Source: Ricardo Guimarães Herrmann

Managing multiple workers 管理多名工人

Very hard! 非常难

- don’t know the order in which workers run
- don’t know when workers interrupt each other
- don’t know when workers need to communicate partial results
- don’t know the order in which workers access shared data

不知道工人运行的顺序、工人何时会互相打扰、员工何时需要通报部分结果、工作人员访问共享数据的顺序

Managing multiple workers

Thus, we need 我们需要：

- semaphores (lock, unlock) 信号量（锁定、解锁）
- conditional variables (wait, notify, broadcast) 条件变量（等待、通知、广播）
- barriers (a job cannot start until its prerequisites have completed)
  障碍（一个工作在其先决条件完成之前无法开始）

But still… 但仍然…

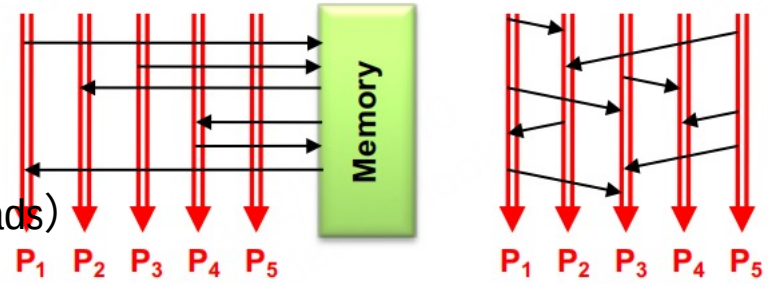
- deadlock, race conditions… 死锁、竞争条件……
- dinning philosophers, sleeping barbers…

Current tools 当前工具

Programming models 编程模型

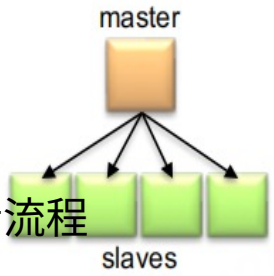
- shared memory (pthreads)
- message passing (MPI)

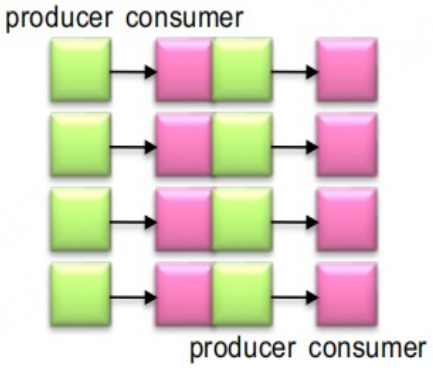
共享内存（pthreads）
消息传递（MPI）

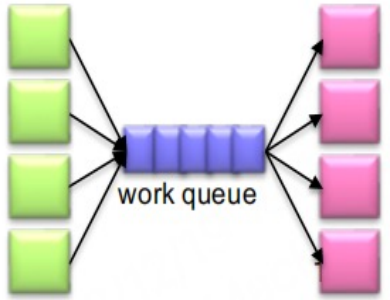


Design Patterns 设计模式

- master-workers 管理者 - 工人
- producer-consumer flows 生产者 - 消费者流程
- shared work queues 共享工作队列







Put into practice 付诸实践

Concurrency is difficult to reason about

并发性很难推理，出现下列情况时更是如此：

And it is even more so

- at the scale of datacenters 数据中心规模
- in the presence of failures 出现故障时
- in terms of multiple interacting services 就多种交互服务而言

Not to mention debugging… 更不用说调试了……

Put into practice

The reality:

现实情况是：

- lots of one-off solutions, workaround, custom code

大量一次性解决方案、变通方法和自定义代码

- write your own dedicated library, then code with it

编写您自己的专用库，然后使用它进行编码

- burden on the programmer to explicitly manage everything

程序员需要明确管理所有事情

- …


MapReduce

Typical big data problems

典型大数据问题

Log analysis: 日志分析：上周收到了多少条警告信息？

- how many warning messages were received last week?
  - e.g., **[Warning] 21:07/01/04/2017 Low memory!**

Web mining: 网络挖掘：

- which wiki pages about Donald Trump have been viewed the most times during the 2016 US Election?
- How many tweets mentioned the word “terrorism” yesterday?

Parallelization challenges

迭代大量记录

Iterate over a large number of records

从每个中提取一些有趣的东西

Extract something of interest from each

MAP

Shuffle and sort intermediate results

打乱并排序中间结果

Aggregate intermediate results

汇总中间结果

Generate final output

生成最终输出

REDUCE

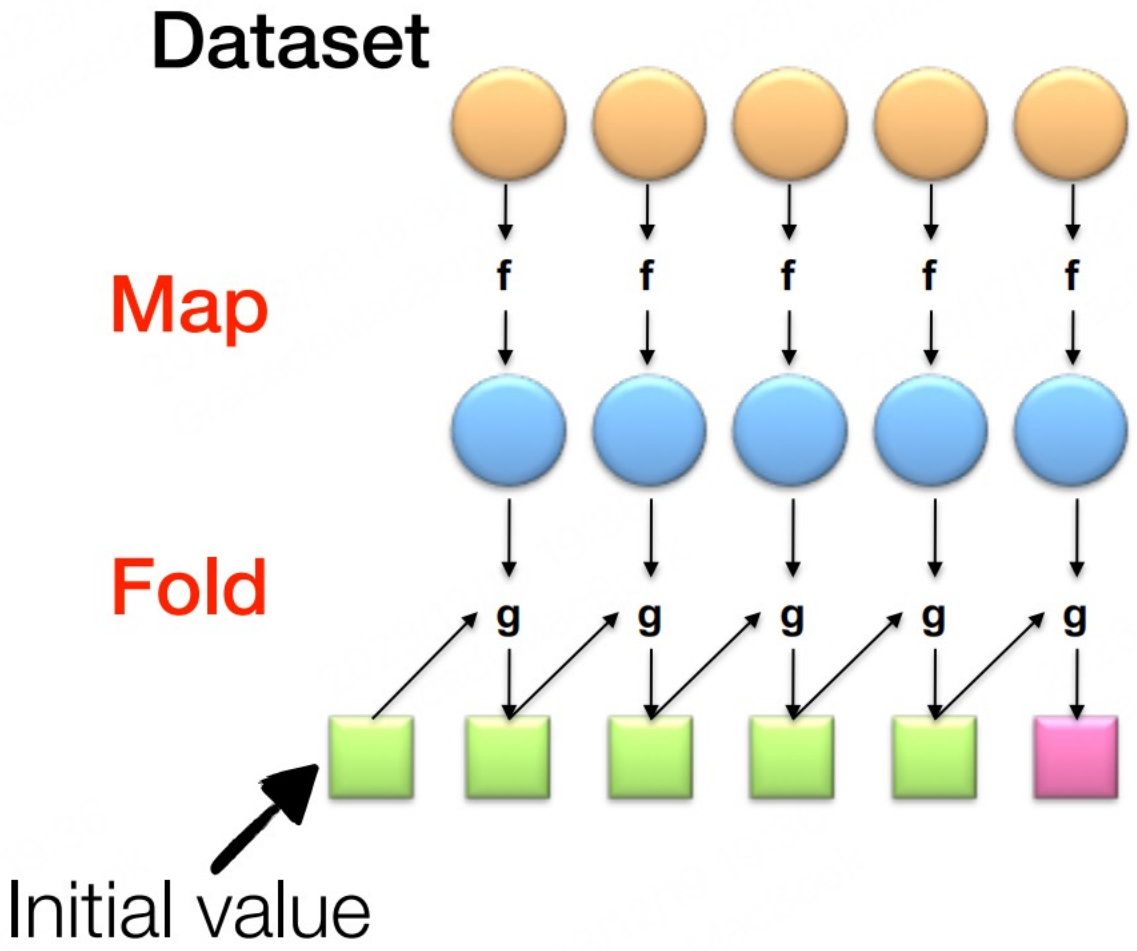
Key idea: provide a functional abstraction for these two operations

关键思想：为这两个操作提供功能抽象

(Dean and Ghemawat, OSDI 2004)

Roots in functional programming

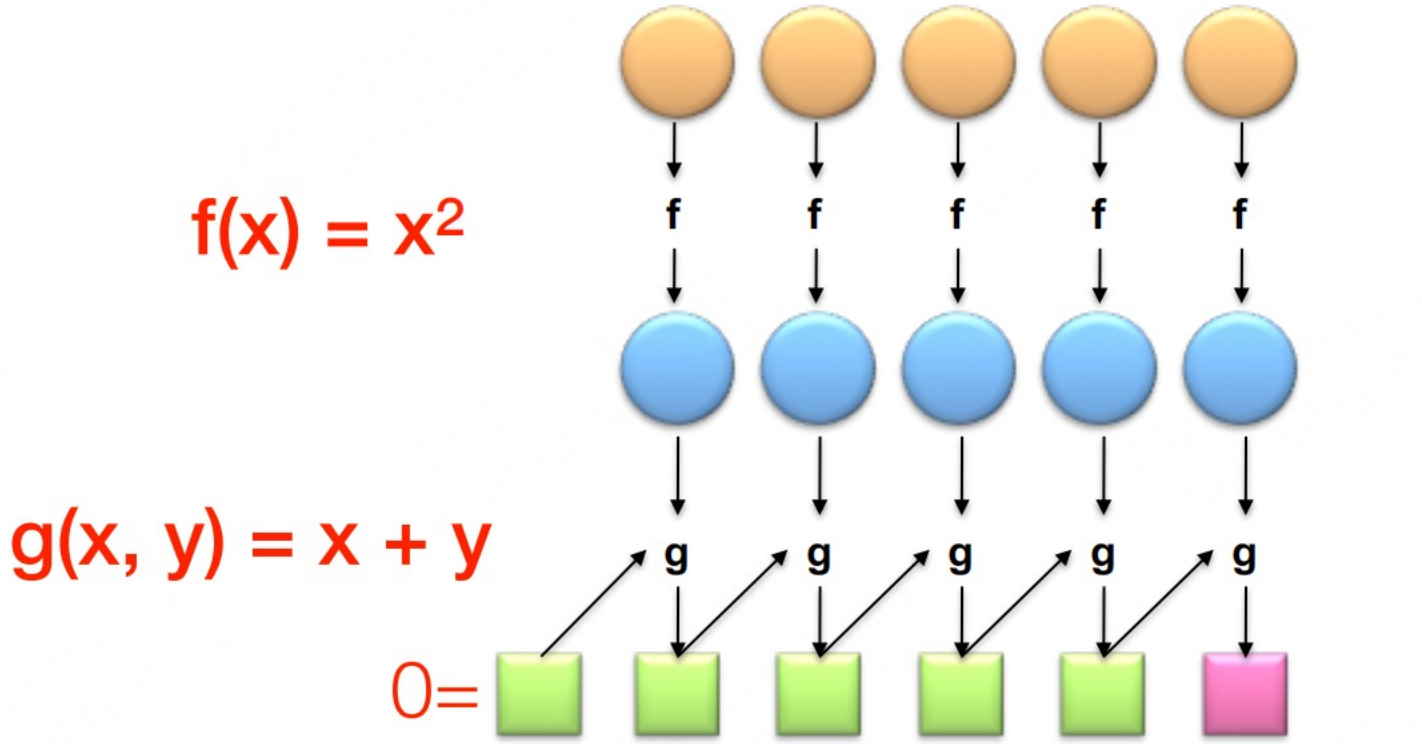
函数式编程的根源



Functional programming

给定一个数据集，计算平方和。

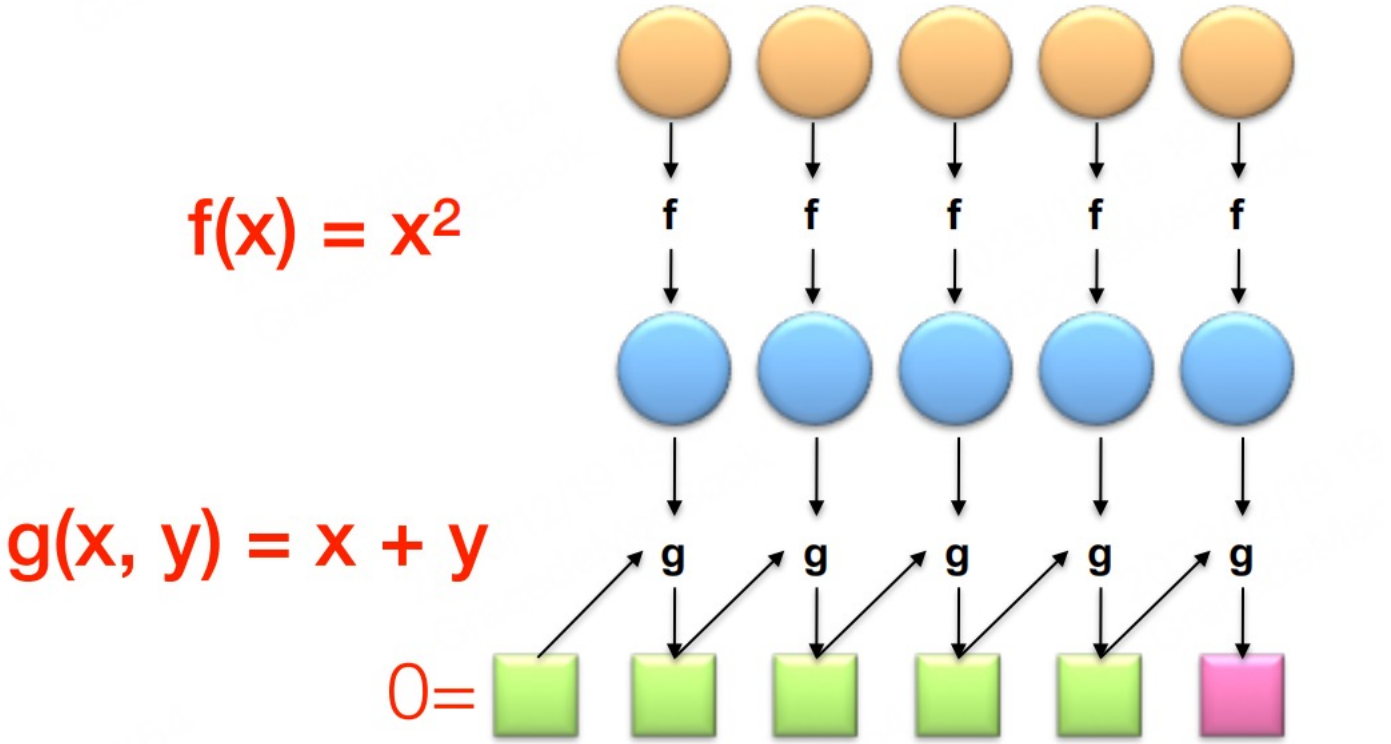
Given a dataset $X = [x_1, \ldots, x_n]$, compute the square sum $\sum_i x_i^2$



Functional programming 函数式编程

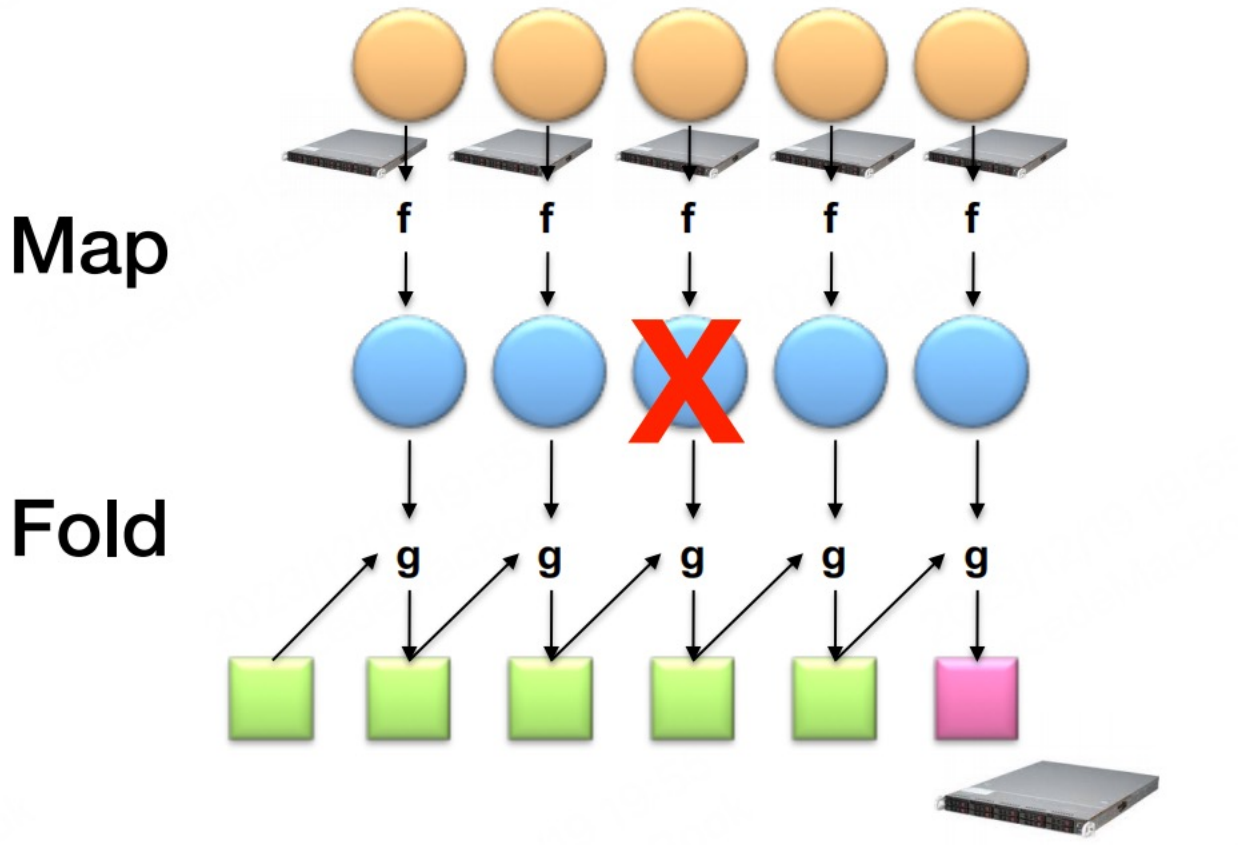
Functional operations never modify existing datasets, but they **create new ones**

函数操作永远不会修改现有的数据集，但它们会创建新的数据集。



Ideal for parallelization

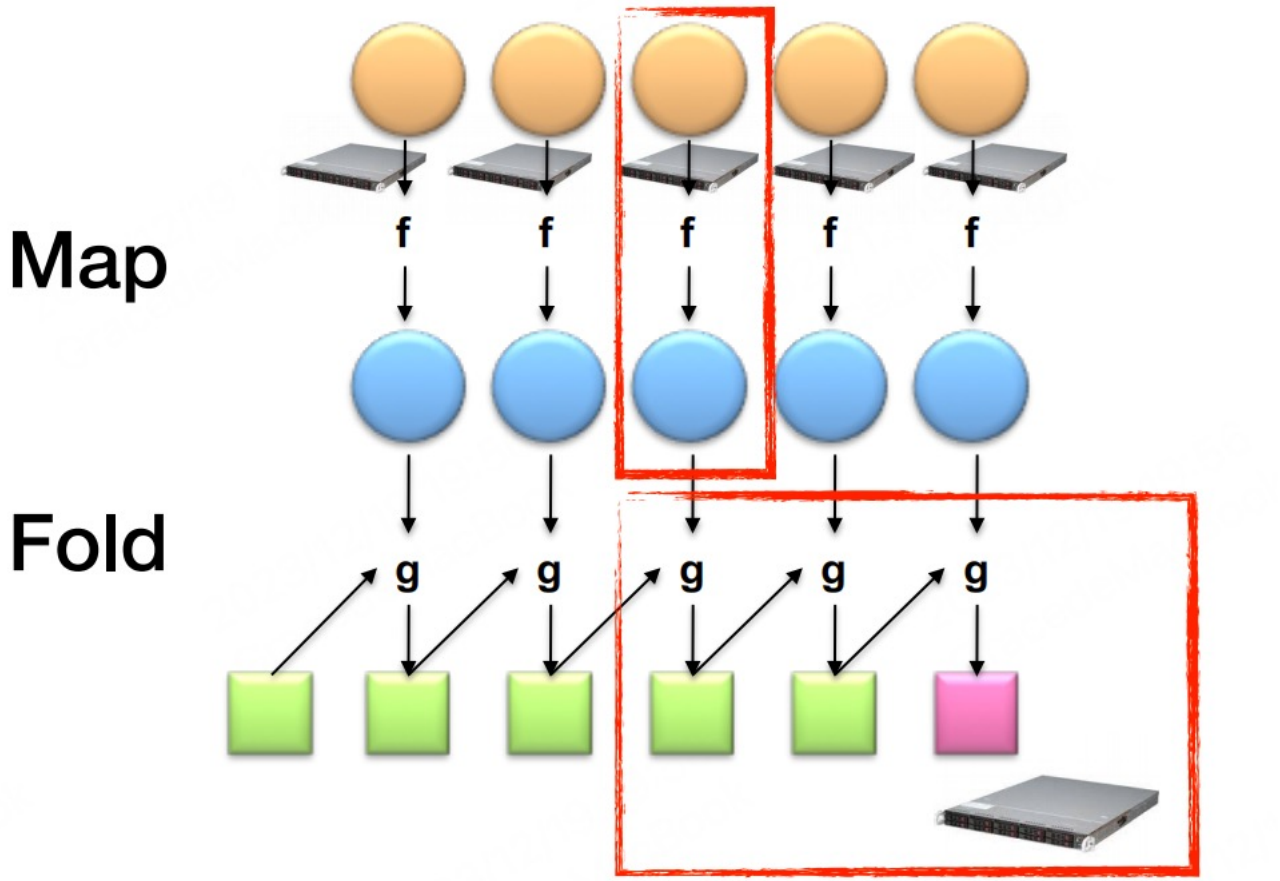
What if a worker fails? 如果工人失效了怎么办？



Ideal for parallelization

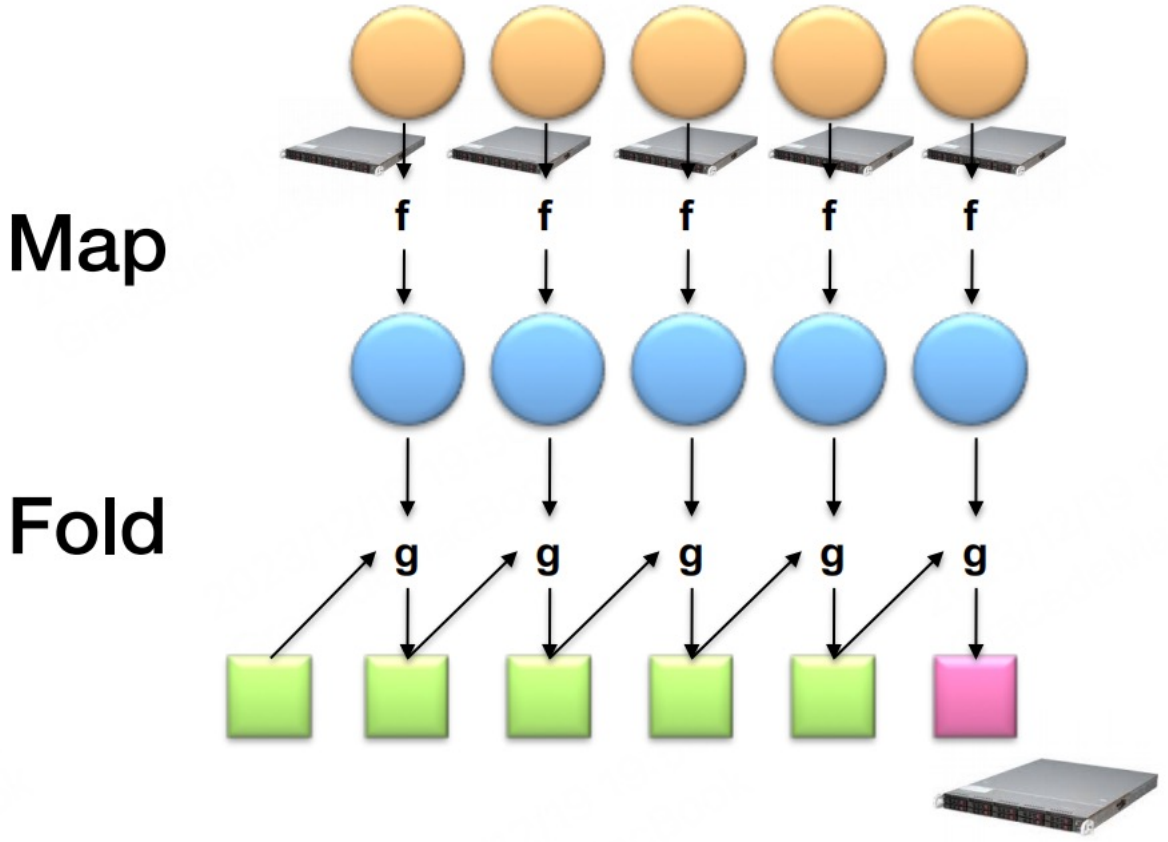
在其他机器上再次执行工作。

Do the work again, on some other machines



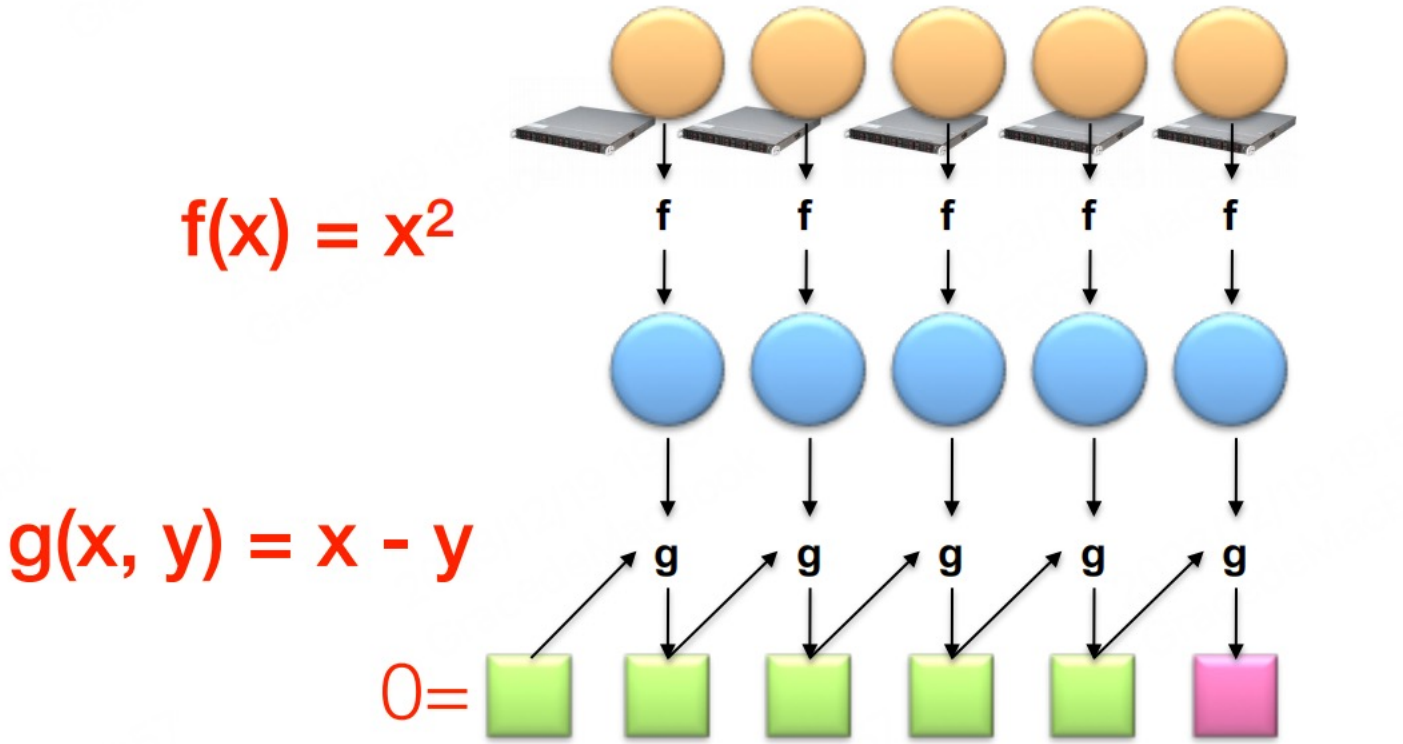
Can we apply any function?

我们可以应用任何函数吗？



Can we apply any function?

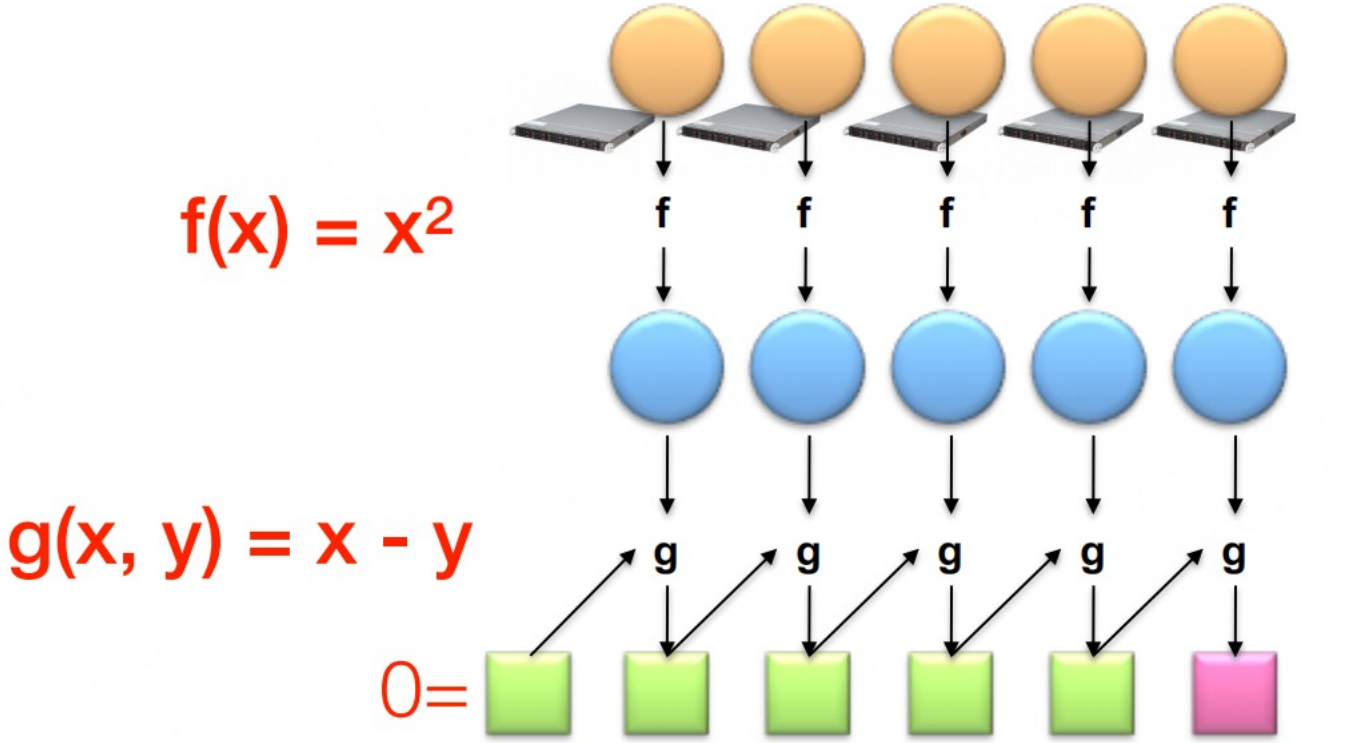
What if g(x, y) = x - y?



Nope…

The order matters, making the results indefinite and hard to reason about!

如果这样）那么顺序很重要，这使得结果不确定且难以推理！



Thus, we require…

Commutativity

变顺序不影响结果

- ✓ $g(x, y) = g(y, x)$
- ✓ e.g., $x + y = y + x$

Associativity

- ✓ $g(g(x, y), z) = g(x, g(y, z))$
- ✓ e.g., $(x + y) + z = x + (y + z)$

The programming model of MapReduce borrows from functional programming

MapReduce 的编程模型借鉴了函数式编程。

Records from the data source are fed as **key-value pairs**, e.g., **[<filename, line>]**

来自数据源的记录以键值对的形式提供，例如[<filename, line>]

MapReduce

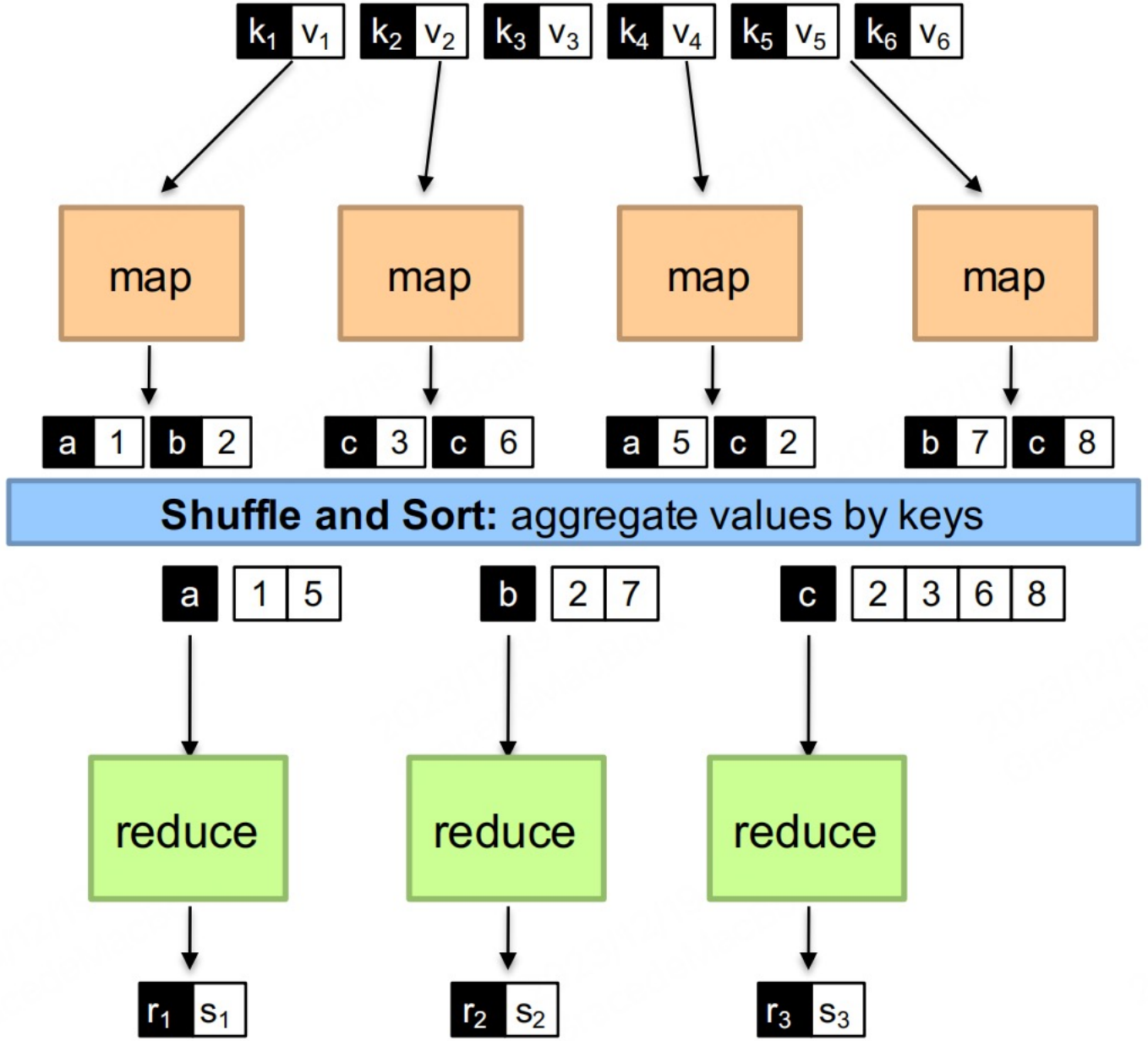
Programmers specify two functions 程序员指定两个函数：

- **map** (k, v) → [<k2, v2>]
- **reduce** (k2, [v2]) → [<k3, v3>]

All values with the same key are sent to the same reducer

所有具有相同键的值都将发送到同一个 Reducer，执行框架处理其他一切事情…

The execution framework handles everything else…

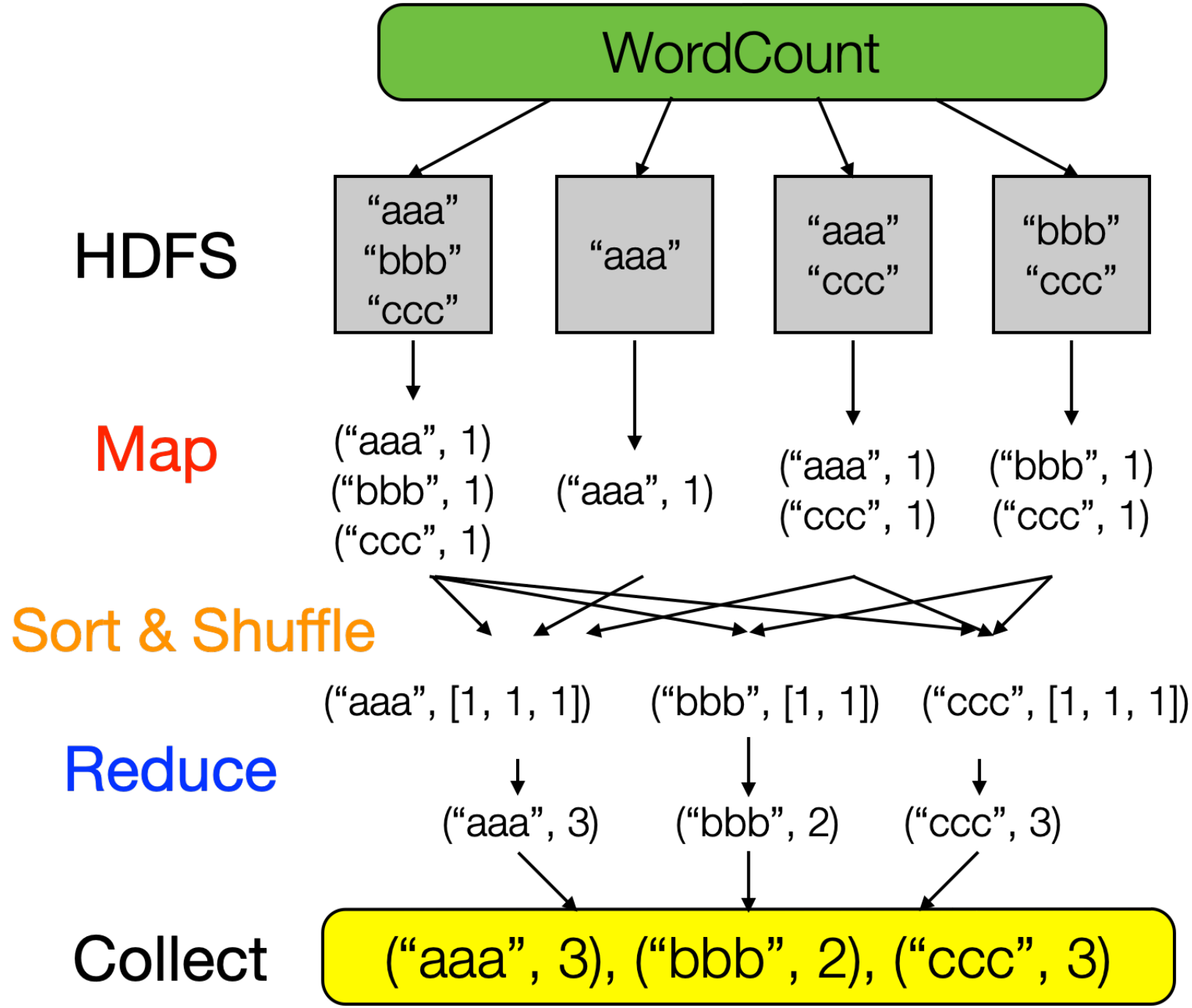

k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

WordCount: a “Hello World” from MapReduce

WordCount

Count the occurrence of each word in a large document

统计大型文档中每个单词的出现次数。


WordCount
HDFS
“aaa”
“bbb”
“ccc”
“aaa”
“aaa”
“ccc”
“bbb”
“ccc”
Map
(“aaa”, 1)
(“bbb”, 1)
(“ccc”, 1)
(“aaa”, 1)
(“aaa”, 1)
(“ccc”, 1)
(“bbb”, 1)
(“ccc”, 1)
Sort & Shuffle
(“aaa”, [1, 1, 1])
(“bbb”, [1, 1])
(“ccc”, [1, 1, 1])
Reduce
(“aaa”, 3)
(“bbb”, 2)
(“ccc”, 3)
Collect
(“aaa”, 3), (“bbb”, 2), (“ccc”, 3)

A tale of two functions…

What to emit?

```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
    Emit(term, sum)
```

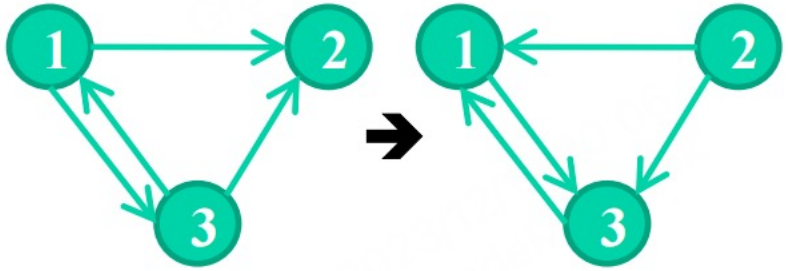
How to reduce?

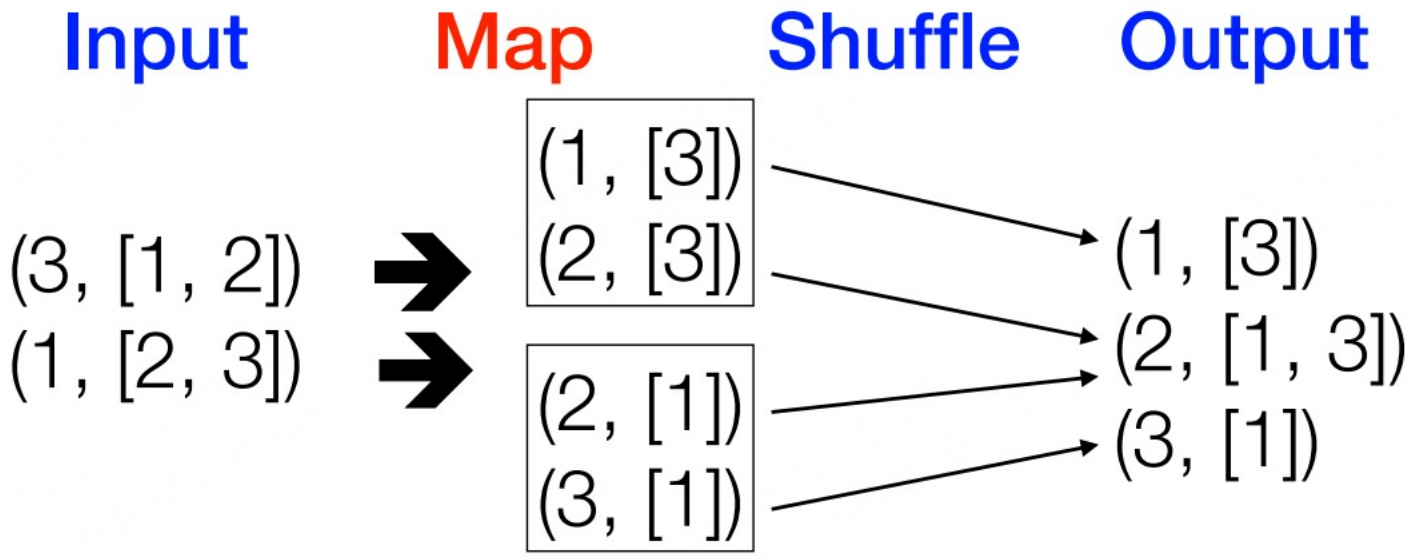
发射什么？
如何减少？

Reverse graph edge directions 反转图形边缘方向

图的邻接表（3 个节点和 4 条边）。

Adjacency list of graph (3 nodes and 4 edges)





Problems solved by MapReduce

Read a lot of data 读取大量数据

Map: (k1, v1) → [<k2, v2>]

- extract something you care about from each record
  从每条记录中提取您关心的内容

Shuffle and sort 随机播放并排序

Reduce: (k2, [v2]) → [<k3, v3>]

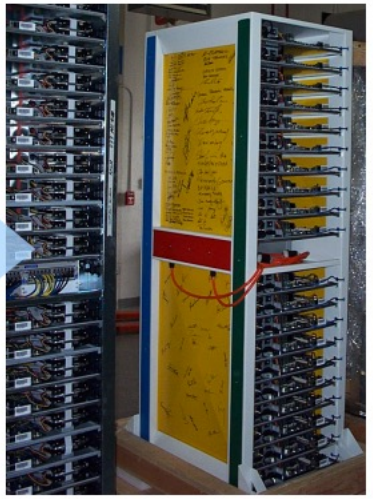
- aggregate, summarize, filter, or transform
  聚合、汇总、过滤或转换

Write the results 写结果

Google Map

Geographic Data

Index Files



Datacenter

Input

Feature List

1: <type=Road>, <intersections=(3)>, <geom>, …

2: <type=Road>, <intersections=(3)>, <geom>, …

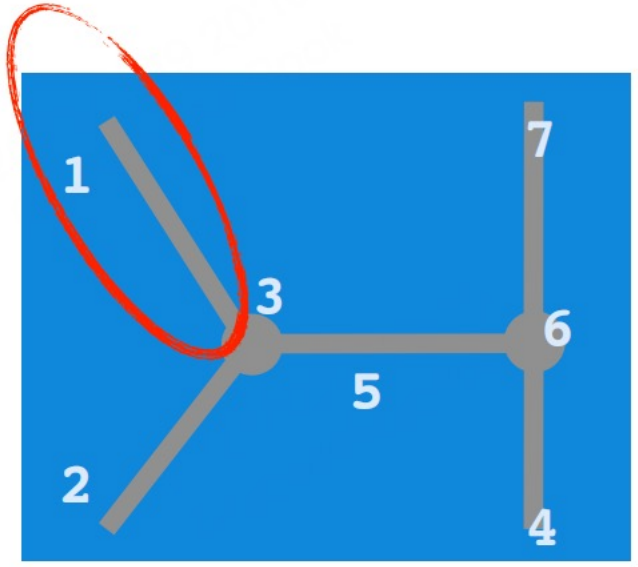
3: <type=Intersection>, <roads=(1,2,5)>, …

4: <type=Road>, <intersections=(6)>, <geom>, …

5: <type=Road>, <intersections=(3, 6)>, <geom>, …

6: <type=Intersection>, <roads=(5,6,7)>, …

7: <type=Road>, <intersections=(6)>, <geom>, …



Input

Feature List

1: <type=Road>, <intersections=(3)>, <geom>, …

2: <type=Road>, <intersections=(3)>, <geom>, …

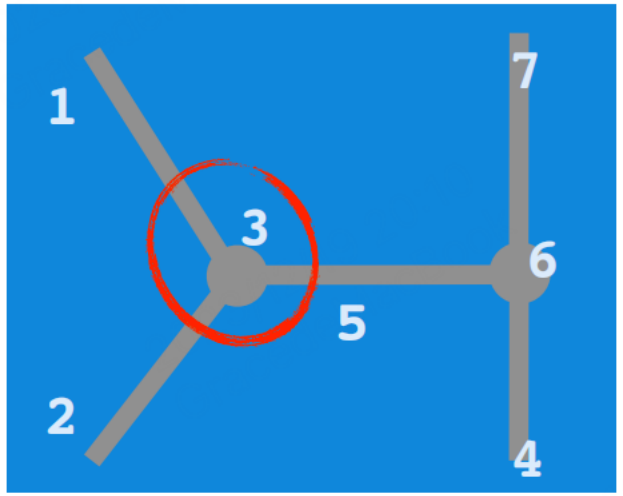
3: <type=Intersection>, <roads=(1,2,5)>, …

4: <type=Road>, <intersections=(6)>, <geom>, …

5: <type=Road>, <intersections=(3, 6)>, <geom>, …

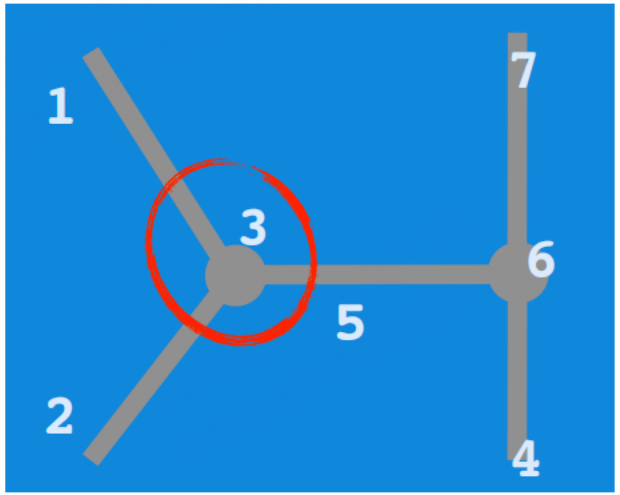
6: <type=Intersection>, <roads=(5,6,7)>, …

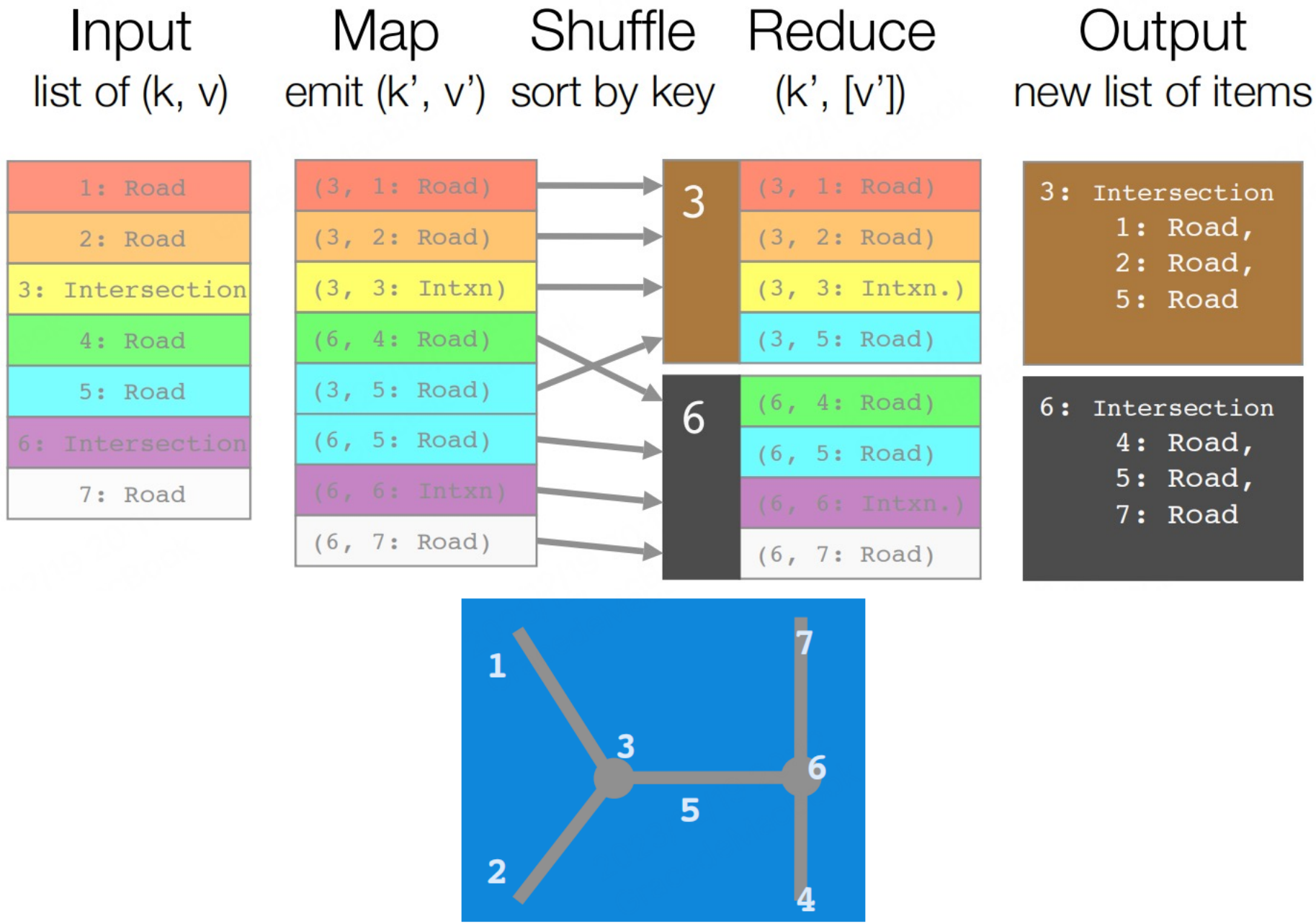
7: <type=Road>, <intersections=(6)>, <geom>, …



Output

```
Intersection List

  3: <type=Intersection>, <roads=(

      1: <type=Road>, <geom>, <name>, …

      2: <type=Road>, <geom>, <name>, …

      5: <type=Road>, <geom>, <name>, …)>, …

  6: <type=Intersection>, <roads=(

      4: <type=Road>, <geom>, <name>, …

      5: <type=Road>, <geom>, <name>, …

      7: <type=Road>, <geom>, <name>, …)>, …
```




Input
list of (k, v)
Map
emit (k', v')
Shuffle
sort by key
Reduce
(k', [v'])
Output
new list of items
1: Road
2: Road
3: Intersection
4: Road
5: Road
6: Intersection
7: Road
(3, 1: Road)
(3, 2: Road)
(3, 3: Intxn)
(6, 4: Road)
(3, 5: Road)
(6, 5: Road)
(6, 6: Intxn)
(6, 7: Road)
3
(3, 1: Road)
(3, 2: Road)
(3, 3: Intxn.)
(3, 5: Road)
6
(6, 4: Road)
(6, 5: Road)
(6, 6: Intxn.)
(6, 7: Road)
3: Intersection
1: Road,
2: Road,
5: Road
6: Intersection
4: Road,
5: Road,
7: Road
1
2
3
4
5
6
7

MapReduce

Programmers specify two functions

- **map** (k, v) → [<k2, v2>]
- **reduce** (k2, [v2]) → [<k3, v3>]

All values with the **same key** are sent to the **same reducer**

所有具有相同键的值都将发送到同一个 Reducer，执行框架处理其他一切事情…

The execution framework handles **everything else**…

What's "everything else"?

Everything else…

Handles scheduling

- assigns workers to map and reduce tasks
- load balancing

处理调度

分配工作人员进行映射和减少任务

负载平衡

Handles “data distribution”

- move processes to data
- automatic parallelization

处理 “ 数据分发 ”

将流程转移到数据

自动并行化

Everything else…

Handles synchronization

处理同步

- gathers, sorts, and shuffles intermediate data

收集、分类和整理中间数据

- network and disk transfer optimization

网络和磁盘传输优化

Handles errors and faults

处理错误和故障检测

- detects worker failures and restarts

工作器故障并重新启动

Everything happens on top of a distributed filesystem

一切都发生在分布式文件系统之上。

MapReduce refinement

Programmers specify two functions

- **map** (k, v) → [<k2, v2>]
- **reduce** (k2, [v2]) → [<k3, v3>]
- All values with the **same key** are reduced together

所有具有相同键的值一起减少。

Not quite… usually, programmers also specify combiner and partitioner

不完全是……通常程序员还会指定组合器和分区器。

Combiner and partitioner

combine (k, [v]) → <k, v>

- mini-reducers that run in memory after the map phase
- used as an optimization to reduce network traffic

在 map 阶段之后在内存中运行的小型减少

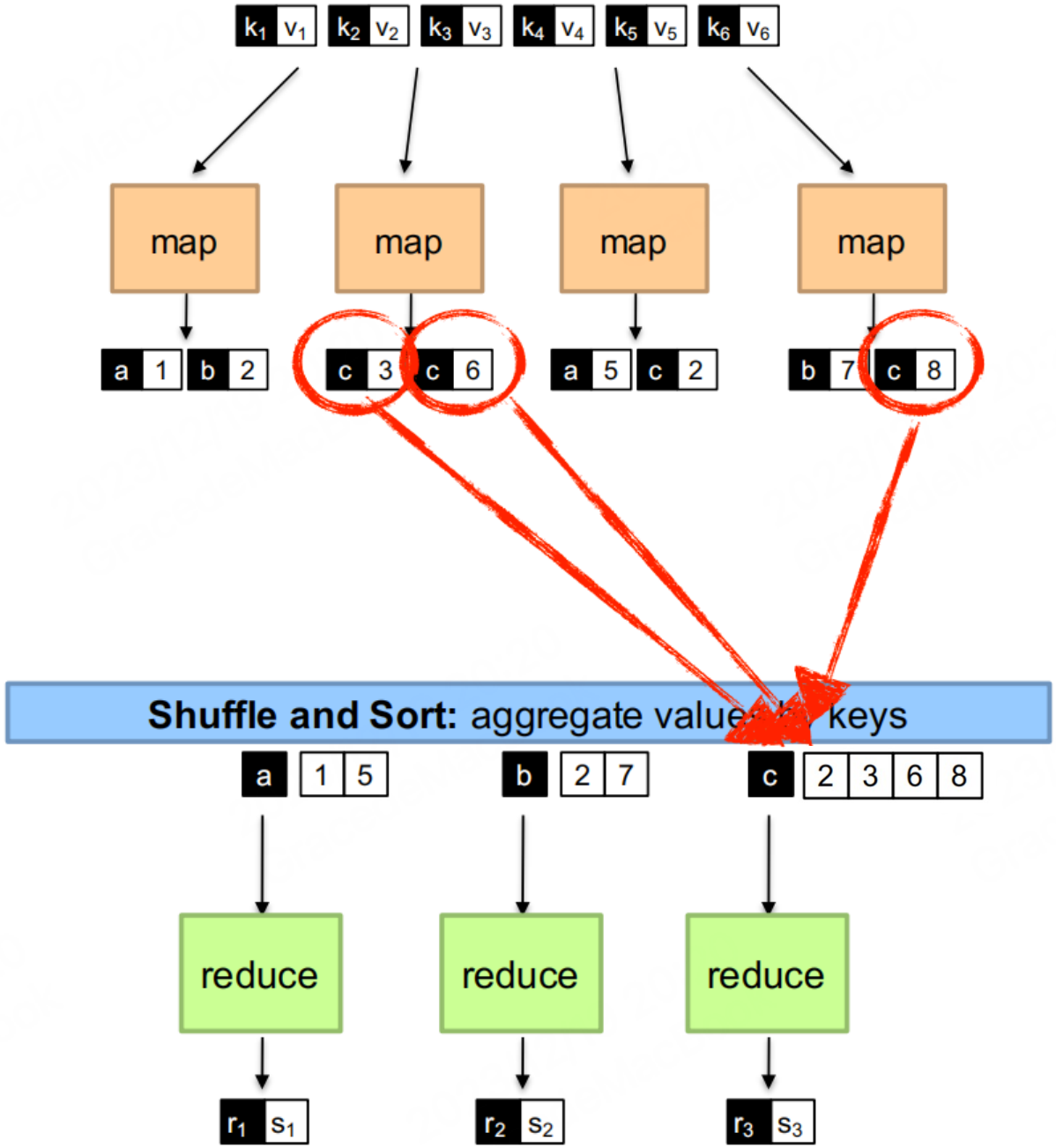
用作减少网络流量的优化手段

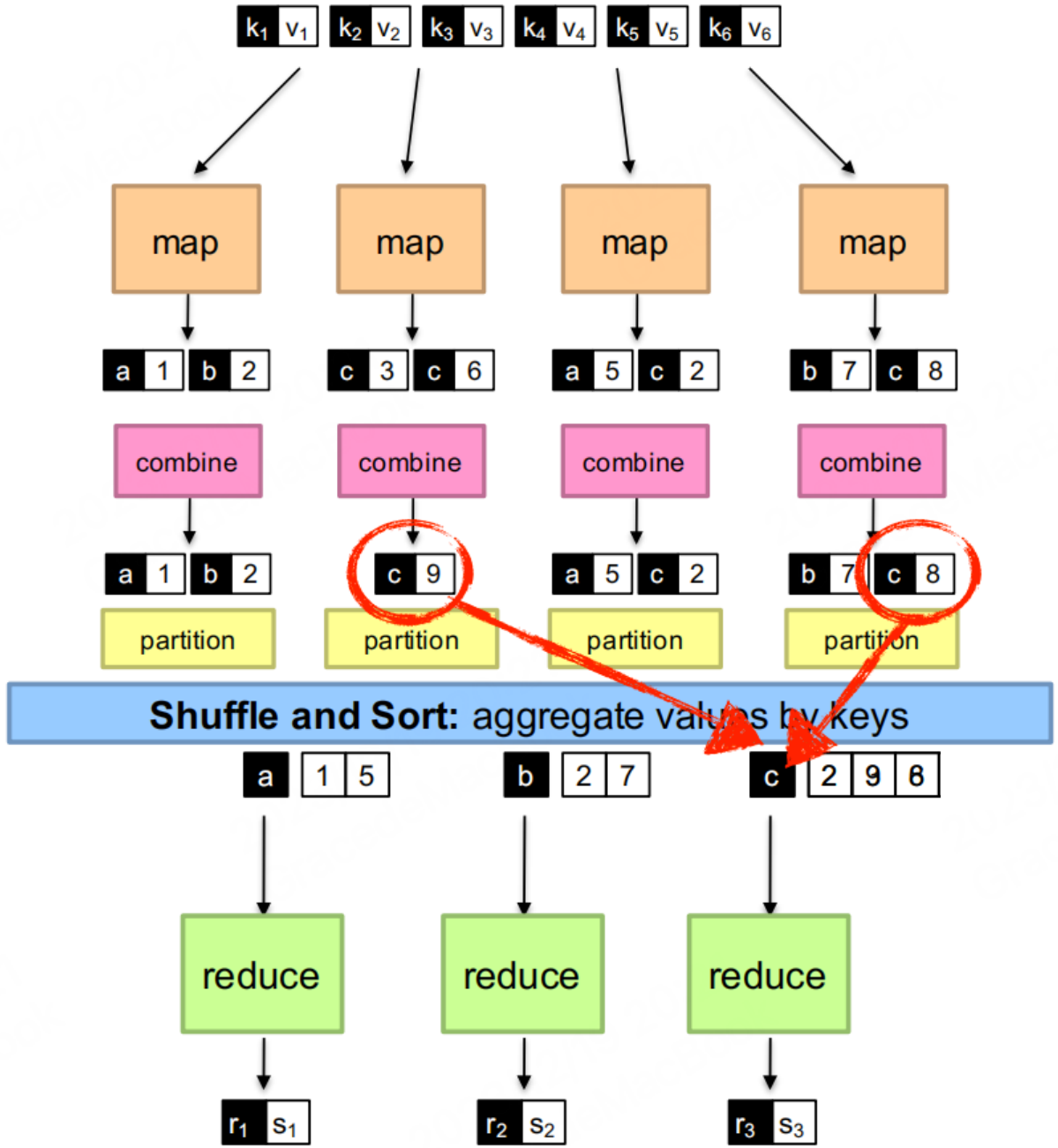
partition (k, # of partitions) → partition for k

- divides up key spaces for parallel reduce operations
- often a simple hash of the key, e.g., hash(k) mod

划分键空间以进行并行 Reduce 操作

通常是密钥的简单哈希，例如 hash(k) mod


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
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
r1 s1
r2 s2
r3 s3


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce

Programmers specify:

- **map** (k1, v1) → [<k2, v2>]
- **combine** (k2, [v2]) → <k2, v2>
- **partition** (k2, # of partitions) → partition for k2
- **reduce** (k2, [v2]) → [<k3, v3>]

All values with the **same key** are sent to the **same reducer**

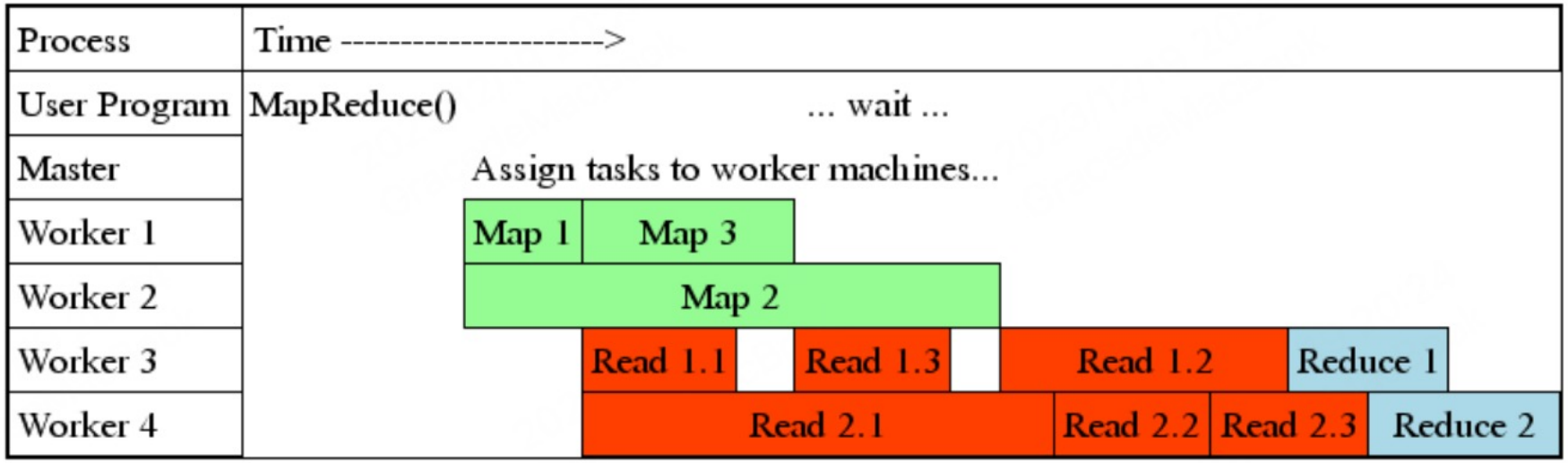
The execution framework handles **everything else**…

所有具有相同键的值都将发送到同一个 Reducer，执行框架处理其他一切事情……

Two more details…

Barrier between map and reduce phases　map 和 reduce 阶段之间的障碍

- no reduce can start until map is complete　在 map 完成之前，reduce 无法启动
- but we can begin transferring intermediate data earlier to **pipeline** shuffling with map execution　但我们可以尽早开始将中间数据传输到管道，使用 map 执行进行改组。



Two more details…

Barrier between map and reduce phases

- no reduce can start until map is complete
- but we can begin transferring intermediate data earlier to **pipeline** shuffling with map execution

Keys arrive at each reducer in **sorted order**

- no enforced ordering across reducers

映射和缩小阶段之间的障碍

在地图完成之前，不能开始降低

但我们可以用地图执行更早地开始，将中间数据转移到管道调用

密钥按排序顺序到达每个减速

无需在减速机之间强制下单

MapReduce can refer to…

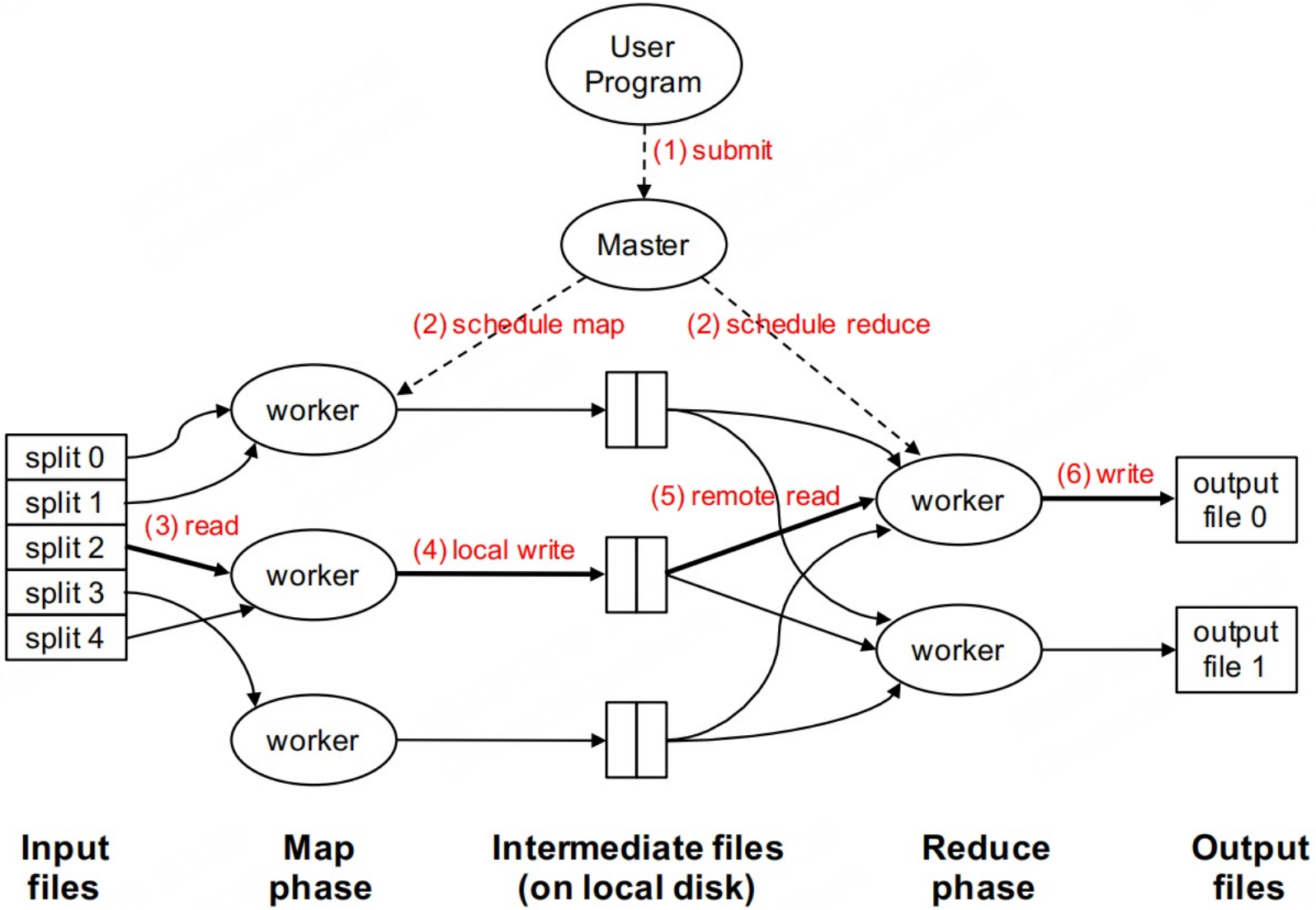
The programming model

The execution framework (aka “runtime”)

The specific implementation

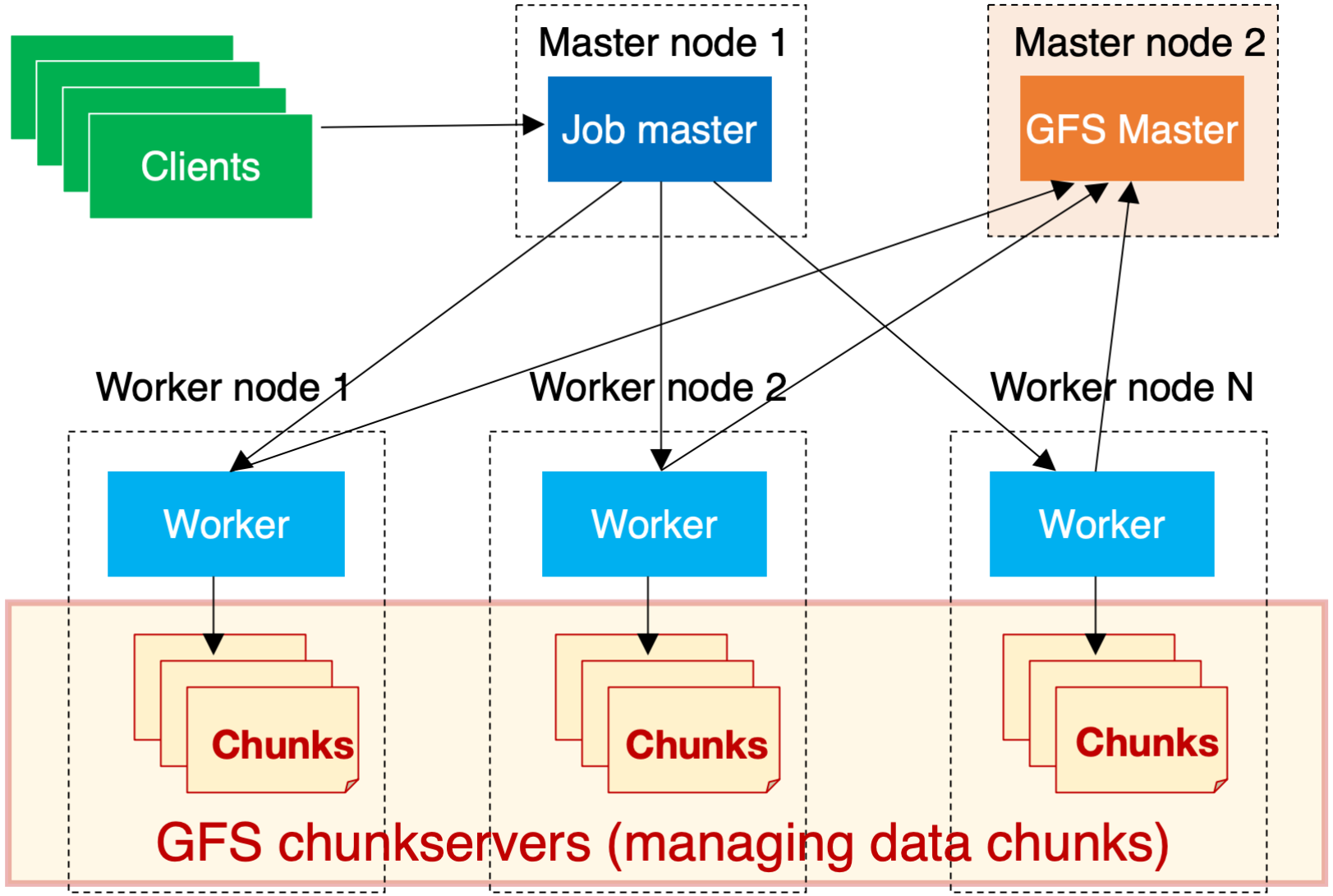
编程模型、执行框架（又称“运行时”）以及具体实现。

Usage is usually clear from context!



Adapted from (Dean and Ghemawat, OSDI 2004)

MapReduce + GFS: put every together



MapReduce Algorithm Design

MapReduce: Recap

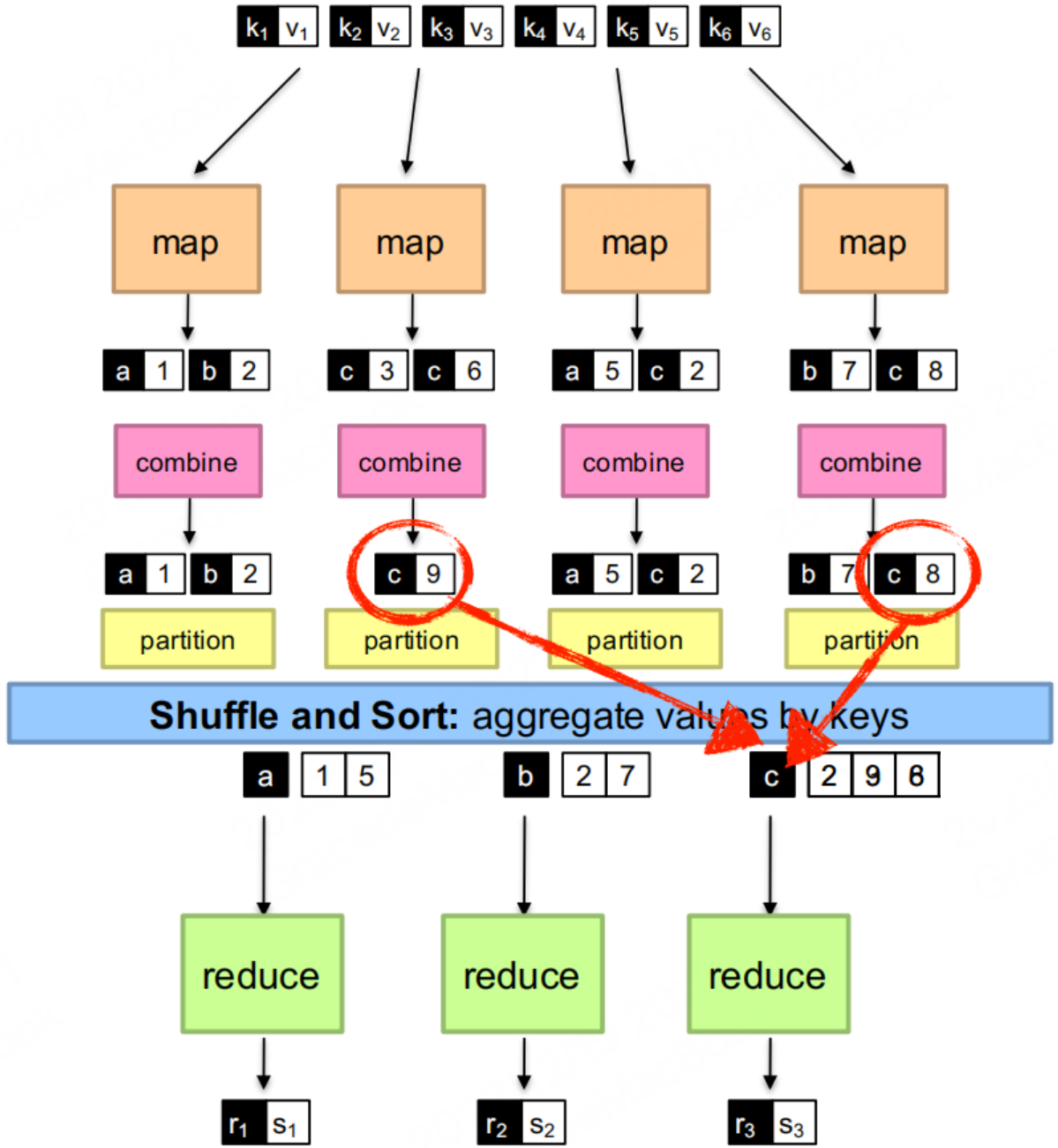
Programmers specify two functions

- **map** (k, v) → [<k2, v2>]
- **reduce** (k2, [v2]) → [<k3, v3>]
- all values with the same key are reduced together

Optionally also:

- **combine** (k, [v]) → <k, v>
- **partition** (k, # of partitions)

The execution framework handles everything else…


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

“Everything else”

Scheduling

调度：分配工作人员执行 map 和 reduce 任务

- assigns workers to map and reduce tasks

Data distribution

数据分发：将流程转移到数据

- moves process to data

Synchronization

同步：收集、排序和交换中间数据

- gathers, sorts, and shuffles intermediate data

Errors and faults

错误和故障：检测工作人员故障并重新启动

- detects worker failures and restarts

Limited control

控制有限

All algorithms must be expressed in m, r, c, p

所有算法都必须用 m、r、c、p 来表达

You don’t know

- where mappers and reducers run
- when a mapper or reducer begins or finishes
- which input a particular mapper is processing
- which intermediate key a particular reducer is processing

映射器和化简器在哪里运行
映射器或化简器何时开始或结束
特定映射器正在处理哪些输入
特定化简器正在处理哪些中间键

But still we can control

Cleverly-constructed data structures 巧妙构建的数据结构，将部分结果整合在一起

- bring partial results together

Sort order of intermediate keys 中间键的排序顺序，控制 Reducer 处理键的顺序

- control order in which reducers process keys

Partitioner 分区器，控制哪个 Reducer 处理哪个键

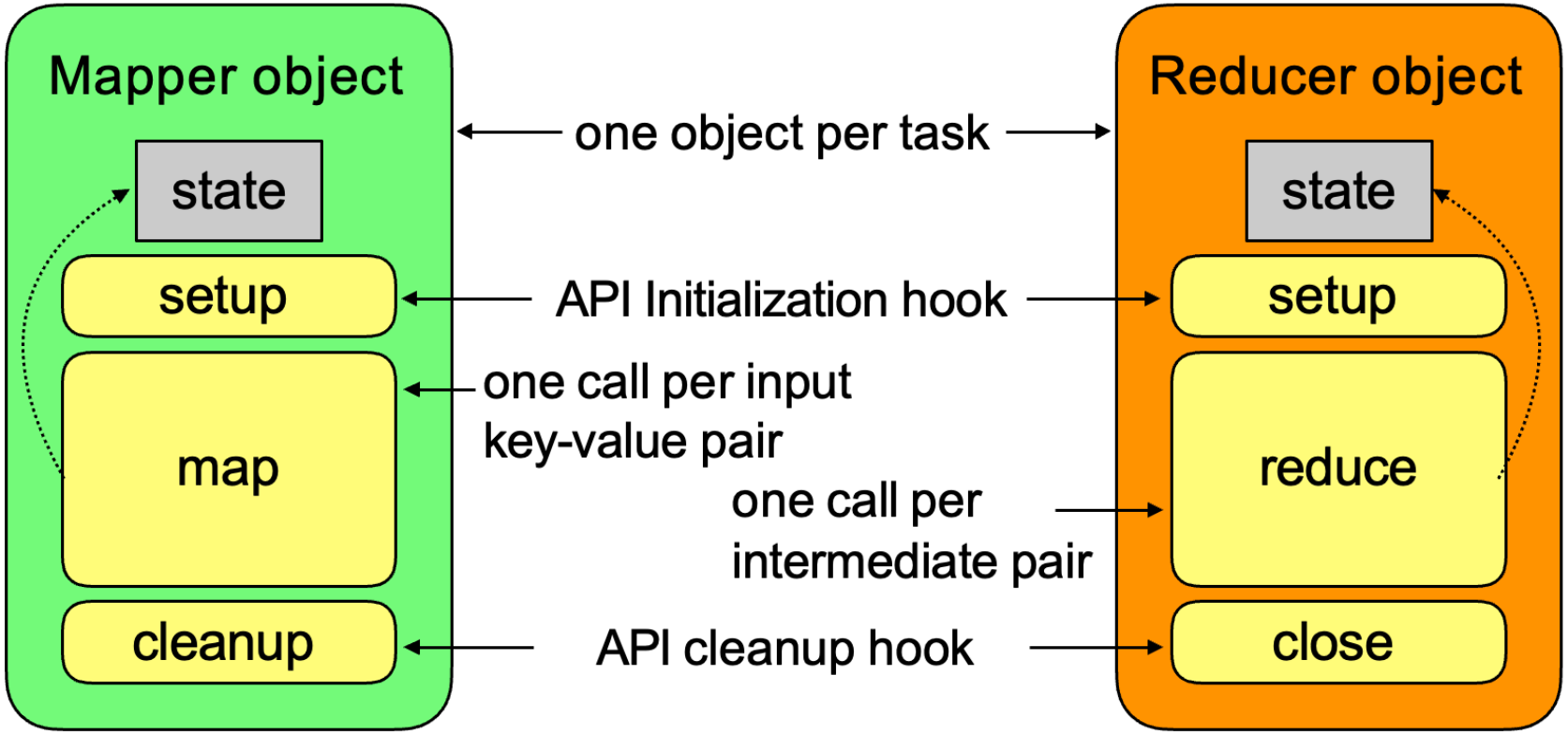
- control which reducer processes which keys

Preserving state in mappers and reducers

- capture dependencies across multiple keys and values

在映射器和化简器中保存状态，捕获跨多个键和值的依赖关系

Preserving state



Importance of local aggregation

理想的扩展特性：

Ideal scaling characteristics

- twice the data, twice the running time
- twice the resources, half the running time

两倍的数据，两倍的运行时间

两倍的资源，一半的运行时间

Why can't we achieve this? 为什么我们不能实现这个？

- synchronization requires communication
- communication kills performance

同步需要通信

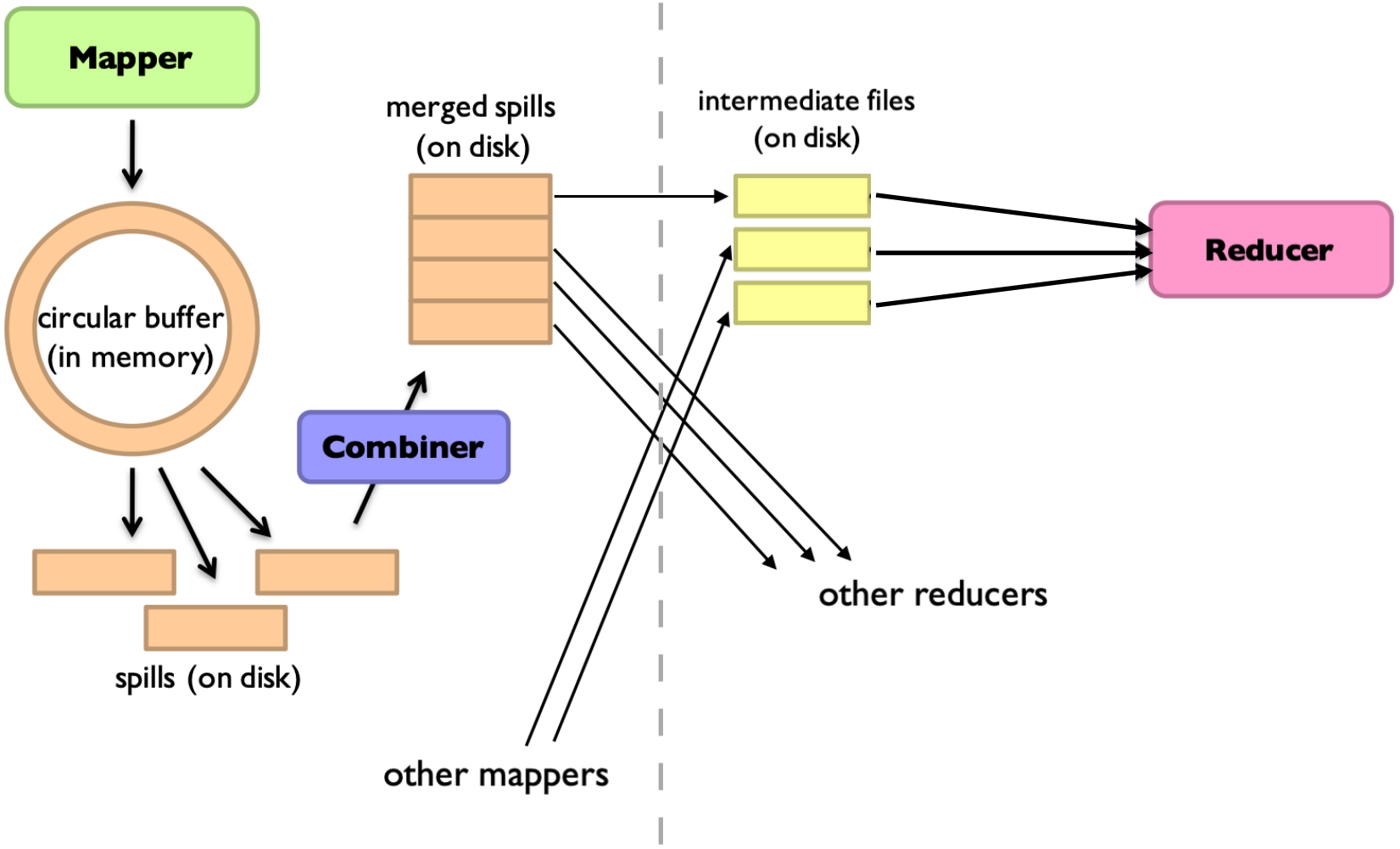
通信会降低性能

Thus… avoid communication, as much as possible! 因此……尽可能避免通信！

- reduce intermediate data via local aggregation
- combiners can help

通过本地聚合减少中间数据
组合器可以提供帮助

Network


Mapper
circular buffer
(in memory)
spills (on disk)
Combiner
merged spills
(on disk)
intermediate files
(on disk)
Reducer
other reducers
other mappers

WordCount: baseline

```
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         for all term t in doc d do
4:             EMIT(term t, count 1)

1: class REDUCER
2:     method REDUCE(term t, counts [c1, c2,...])
3:         sum → 0
4:         for all count c in counts [c1, c2,...] do
5:             sum → sum + c
6:         EMIT(term t, count sum)
```

What's the impact of combiners?

WordCount: version 1

```
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         H → new ASSOCIATIVEARRAY
4:         for all term t in doc d do
5:             H{t} → H{t} + 1
6:         for all term t in H do
7:             EMIT(term t, count H{t})
```

H{*t*}: a hash table

Do combiners still help?

WordCount: version 2

```
1: class Mapper
2:     method Initialize
3:         H → new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} → H{t} + 1
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

H{*t*}: a hash table

Key idea: preserve state across input key-value pairs!

Design pattern for local aggregation

映射器内合并

In-mapper combining

映射器内合并： 通过在多个映射调用中保存状态，将组合器的功能折叠到映射器中

- Fold the functionality of the combiner into the mapper by preserving state across multiple map calls

Advantages

优势：速度

- Speed

Disadvantages

缺点：需要显式内存管理

- Explicit memory management required

Combiner design

组合器设计

Combiners and reducers share same method signature

组合器和缩减器共享相同的方法签名

- sometimes, reducers can serve combiners
- often, not…

有时，缩减器可以用作组合器

Combiners are optional optimizations

组合器是可选的优化

- should not affect algorithm correctness
- may be run 0, 1, or multiple times (indefinite)

不应影响算法的正确性

Example: find the mean of integers associated with the same key

Computing the mean: version 1

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Reducer
2:     method Reduce(string t, integers [r1, r2, ...])
3:         sum → 0
4:         cnt → 0
5:         for all integer r in integers [r1, r2, ...] do
6:             sum → sum + r
7:             cnt → cnt + 1
8:         r_avg → sum/cnt
9:         Emit(string t, integer r_avg)
```

Why can't we use reducer as combiner?

Version 2

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)
1: class Combiner
2:     method Combine(string t, integers [r1, r2, . . .])
3:         sum → 0
4:         cnt → 0
5:         for all integer r in integers [r1, r2, . . .] do
6:             sum → sum + r
7:             cnt → cnt + 1
8:         Emit(string t, pair (sum, cnt))
1: class Reducer
2:     method Reduce(string t, pairs [(s1, c1), (s2, c2) . . .]) sum
3:         → 0
4:         cnt → 0
5:         for all pair (s, c) in pairs [(s1, c1), (s2, c2) . . .] do
6:             sum → sum + s
7:             cnt → cnt + c
8:         r_avg → sum/cnt
9:         Emit(string t, integer r_avg)
```

Why doesn't this work?

Version 3

Fixed?

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))

1: class Combiner
2:     method Combine(string t, pairs [(s1, c1), (s2, c2)...]) sum
3:         → 0
4:         cnt → 0
5:         for all pair (s, c) in pairs [(s1, c1), (s2, c2)...] do
6:             sum → sum + s
7:             cnt → cnt + c
8:         Emit(string t, pair (sum, cnt))

1: class Reducer
2:     method Reduce(string t, pairs [(s1, c1), (s2, c2)...])
3:         sum → 0
4:         cnt → 0
5:         for all pair (s, c) in pairs [(s1, c1), (s2, c2)...] do
6:             sum → sum + s
7:             cnt → cnt + c
8:         r_avg → sum/cnt
9:         Emit(string t, integer r_avg)
```

Version 4

```
1:  class Mapper
2:      method Initialize
3:          S ← new AssociativeArray
4:          C ← new AssociativeArray
5:      method Map(string t, integer r)
6:          S{t} ← S{t} + r
7:          C{t} ← C{t} + 1
8:      method Close
9:          for all term t in S do
10:             Emit(term t, pair (S{t}, C{t}))
```

Are combiners still needed?

Issues and tradeoffs

Local aggregation 本地聚合

- Opportunities to perform local aggregation varies
- Combiners make a big difference
- Combiners vs. in-mapper combining
- RAM vs. disk vs. network

执行本地聚合的机会各不相同
组合器有很大的不同
组合器与映射器内组合的方式
RAM、磁盘与网络之间的权衡

Debugging at scale 大规模调试

Work on small datasets, won't scale… why? 处理小型数据集时，无法扩展……为什么？

- memory management issues (buffering and object creation)
- too much intermediate data
- mangled input records

内存管理问题（缓冲和对象创建）
中间数据太多
输入记录混乱

现实世界的数据很混乱!

Real-world data is messy!

不存在 " 一致数据 "

- there's no such thing as "consistent data"
- watch out for corner cases

注意极端情况

- isolate unexpected behavior, bring local

隔离意外行为，将本地优化应用到全局

MapReduce implementation

Hadoop：Java 中 MapReduce 的开源实现

Hadoop: An open-source implementation of MapReduce in Java

- development led by Yahoo!, now an Apache project
- used in production at Yahoo!, Facebook, Twitter, LinkedIn, Netflix, …
- the de facto big data processing platform
- large and expanding software ecosystem
- lots of custom research implementations

事实上的大数据处理平台
大型且不断扩展的软件生态系统
大量定制研究实施

Credits

- Some slides adapted from course slides of COMP 4651 in HKUST
- Some slides adapted from course slides of DS5110 in UVA
- J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In USENIX OSDI, 2004.