大数据算法

第七讲 基于MapReduce的并行算法设计

哈尔滨工业大学
王宏志
wangzh@hit.edu.cn

本讲内容

7.1 MapReduce概述

7.2 字数统计

7.3 平均数计算

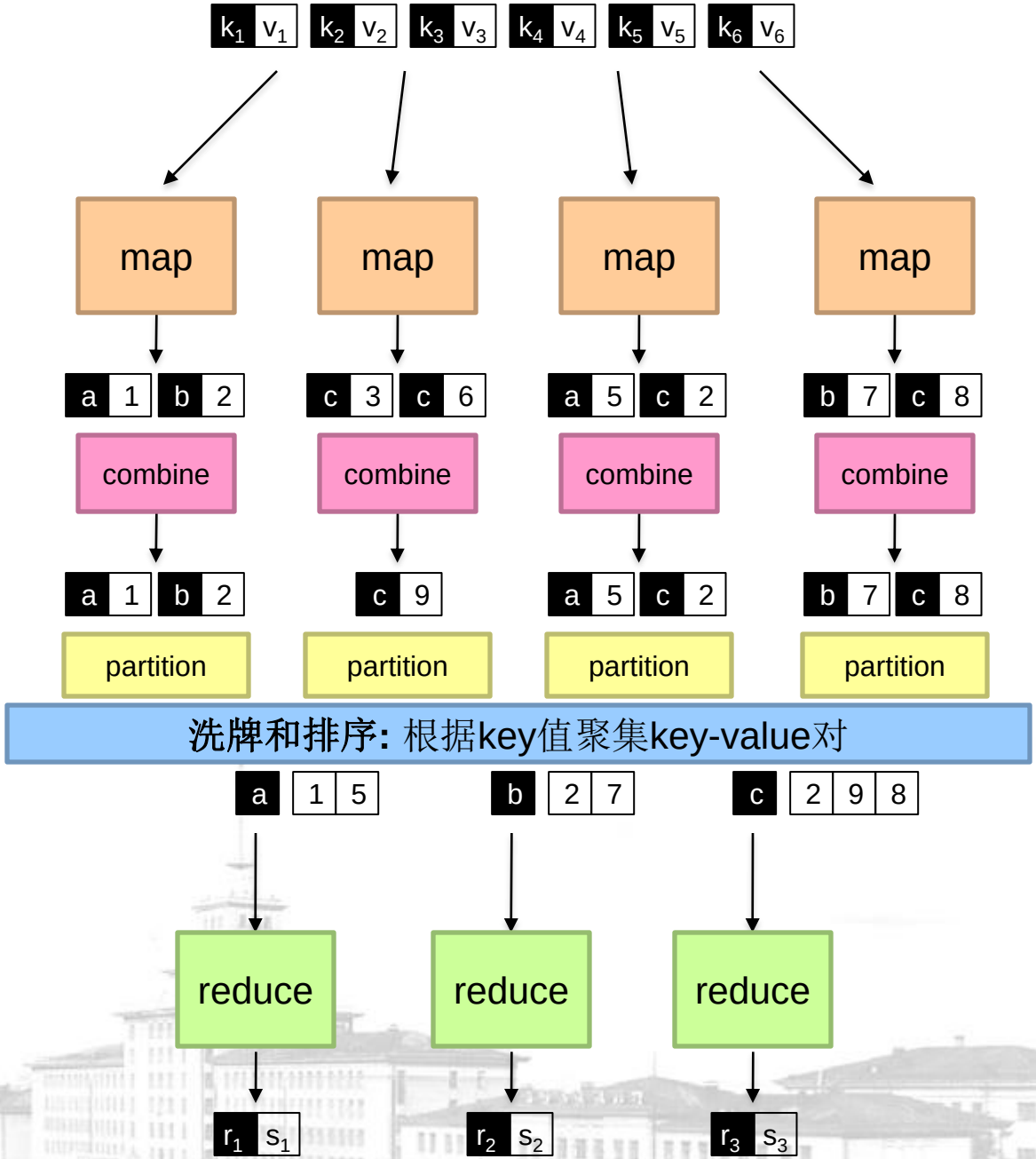
7.4 单词共现矩阵的计算

MapReduce

- MapReduce 是由Google公司的Jeffrey Dean 和 Sanjay Ghemawat 开发的分布式编程模型。
- MapReduce实现了两个主要功能
  - Map把一个函数应用于集合中的所有成员，然后返回一个基于这个处理的结果集。
  - Reduce是把从两个或更多个Map中，通过多个线程，进程或者独立系统并行执行处理的结果集进行分类和归纳。
  - Map() 和 Reduce() 两个函数可能会并行运行，即使不是在同一的系统的同一时刻。

MapReduce 模型

- 用户定义的**Map**和**Reduce**函数（**无状态**）
- 输入: 一个key/value对元组的列表 (k1/v1)
  - 用户的map函数被应用于每个key/value对
  - 产生中间key/value对列表
- 输出: 一个key/value对元组的列表 (k2/v2)
  - 中间值基于key值分组
  - 用户的reduce函数被应用于每个组
- **每个元组都是独立的**
  - 可以用分布式大规模并行的方式进行处理
  - 总输入能远大于工人的内存


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
洗牌和排序: 根据key值聚集key-value对
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce:编程重点

- 程序员必须指定:
  map (k, v) → <k', v'>*
  reduce (k', v') → <k', v'>*
  - 所有具有相同key的value被聚集到一起
- 可选的操作:
  partition (k',划分数) → k'的划分
  - 往往使用key的一个简单散列函数, e.g., hash(k') mod n
  - 为并行reduce操作划分key空间
  combine (k', v') → <k', v'>*
  - map后的阶段运行的小reducer
  - 用作减少网络流量的优化器
- 执行框架处理其他的一切…

“其他的一切”

- **执行框架处理一切**
  - **调度**:为map和reduce分配工人
  - **“数据分布”**:将过程移动到数据
  - **同步**:中间数据进行聚集，排序或洗牌
  - **错误处理**:检测工人失败和重新启动
- **人的工作**
  - 所有算法都必须用m,r(c，p)表达
- **不知道的**:
  - map和reduce在哪里运行
  - mapper或reducer何时结束
  - 一个特定的mapper正在处理哪个输入
  - 一个特定的reducer正在处理哪个特定中间键值

同步工具

- 聪明构建数据结构
  - 将部分结果联系在一起
- 中间键的排序顺序
  - 控制reducer处理键的顺序
- partitioner
  - 控制哪些reducer处理哪些键
- 在mapper和reducer中保持状态
  - 捕获多个键和值的关系

可扩展的MapReduce算法的实现

- **避免创建对象**
  - 昂贵的操作
  - 垃圾收集
- **避免缓冲**
  - 有限的堆
  - 适用于小数据集，但难以扩展！
- **避免Mapper和Reducer间的全局变量传递**
  - 难以通信

本地聚合的重要性

- 理想的可扩展性:
  - 数据加倍,运行时间加倍
  - 资源加倍,运行时间减半
- 为什么我们不能做到这一点呢?
  - 同步需要通信
  - 通信影响性能
- 因此…避免通信!
  - 通过本地聚合减少中间数据
  - 有效利用combiner

本讲内容

7.1 MapReduce概述

7.2 字数统计

7.3 平均数计算

7.4 单词共现矩阵的计算

字数统计:基准算法

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ 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 ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```

combiner有何影响?

字数统计: 版本 1

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

是否仍然需要combiner?

字数统计:版本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                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

关键: 保持输入key-value对的状态

还需要combiner吗?

本地聚合的设计模式

- “In-mapper聚合”
  - 保持多个map调用中的状态，将combiner的功能集成到mapper中
- 优点
  - 速度
  - 为什么这个快于实际的combiner?
- 劣势
  - 需要显式的内存管理

本讲内容

7.1 MapReduce概述
7.2 字数统计
7.3 平均数计算
7.4 单词共现矩阵的计算

combiner设计

- combiner和reducer共享相同的方法
  - 有时reducer可以用作combiner
  - 大部分时候不行…
- 记住:combiner是可选的优化
  - 不应该影响算法的正确性
  - 可能运行0,1次或者多次
- **例子**:找到与相同的键值相关联的所有整数的平均数

计算平均数: 版本 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 [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

为什么我们不能用reducer代替combiner?

计算平均数: 版本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 [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         Emit(string t, pair (sum, cnt))          ▷ Separate sum and count

1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

为何无效?

计算平均数：版本3

```
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 [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] 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 [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, pair (r_avg, cnt))
```

正确否?

计算平均数：版本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 ∈ S do
10:            Emit(term t, pair (S{t}, C{t}))
```

还需要combiner吗?

本讲内容

7.1 MapReduce概述
7.2 字数统计
7.3 平均数计算
7.4 单词共现矩阵的计算

算法设计:运行示例

- 计算文本集合中词的共现矩阵
  - M = N x N 矩阵(N =词数)
  - M_{ij}: *i* 和 *j*出现在同一个上下文的次数 (具体的说,让上下文=句子)
- 为什么?
  - 作为一种测量语义距离的方法
  - 语义距离可用于许多语言处理任务

大规模计数问题

- 一个文本集合的单词共现矩阵——计数问题的实例
  - 一个大事件空间(单词数)
  - 大量的观测值(单词集合)
  - 目标：计算对事件的统计
- 基本方法
  - Mapper生成部分计数
  - Reducer聚合部分计数

如何高效聚合部分计数?

首次尝试："词对法"

- 每个Mapper处理一个句子:
  - 生成所有共现的词对
  - 对于所有的词对, emit (a,b) → 计数
- Reducer将这些词对对应的计数加和
- 使用combiner!

词对法: 伪代码

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence

1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                              ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

词对法：分析

- 优点
  - 易于实现，易懂
- 缺点
  - 排序和洗牌代价高
  - combiner没有多少机会起作用

另外的尝试：条纹法

- 想法:将词对聚集到相关数组中

(a, b) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
(a, f) → 2

a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

- 每个mapper处理一个句子:
  - 生成所有共现的词对
  - 对每个词，传递a → { b:$count_b$, c:$count_c$, d:$count_d$ … }
- Reducer处理相应数组中对应元素的频率

```
    a → { b: 1,       d: 5, e: 3 }
+   a → { b: 1, c: 2, d: 2,       f: 2 }
    a → { b: 2, c: 2, d: 7, e: 3, f: 2 }
```

关键点：聪明地设计数据结构来集成部分结果.

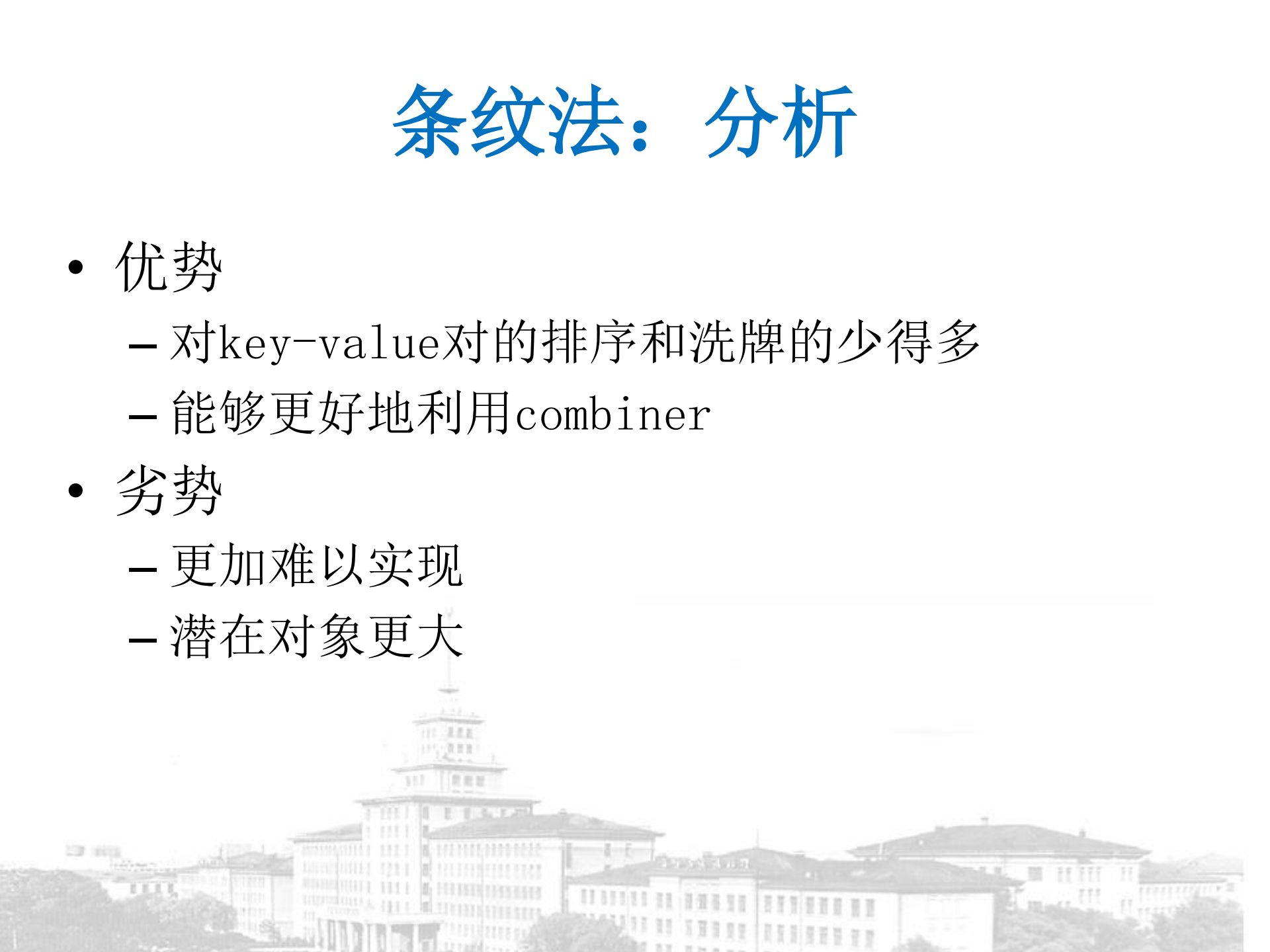
条纹法: 伪代码

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

1: class Reducer
2:     method Reduce(term w, stripes [H_1, H_2, H_3, ...])
3:         H_f ← new AssociativeArray
4:         for all stripe H ∈ stripes [H_1, H_2, H_3, ...] do
5:             Sum(H_f, H)                        ▷ Element-wise sum
6:         Emit(term w, stripe H_f)
```

条纹法：分析

- 优势
  - 对key-value对的排序和洗牌的少得多
  - 能够更好地利用combiner
- 劣势
  - 更加难以实现
  - 潜在对象更大

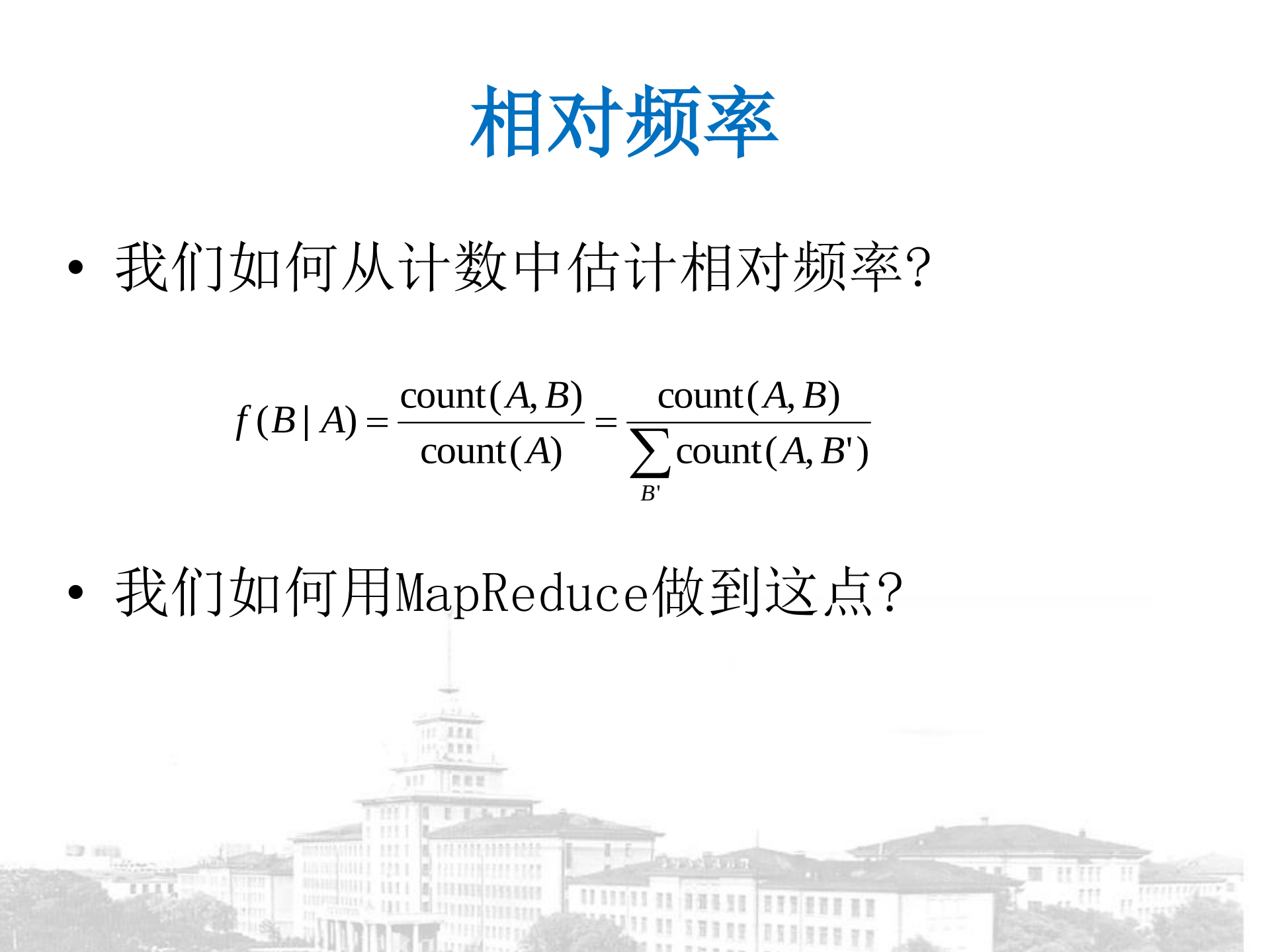


相对频率

- 我们如何从计数中估计相对频率?

$$f(B \mid A) = \frac{\text{count}(A, B)}{\text{count}(A)} = \frac{\text{count}(A, B)}{\sum_{B'} \text{count}(A, B')}$$

- 我们如何用MapReduce做到这点?



f(B|A): 条纹法

- 容易!
  - 通过扫描数据一遍计算（a, *)
  - 扫描数据另外一遍直接计算f(B|A)

f(B|A):词对法

(a, *) → 32

Reducer 将这个值保存在内存中

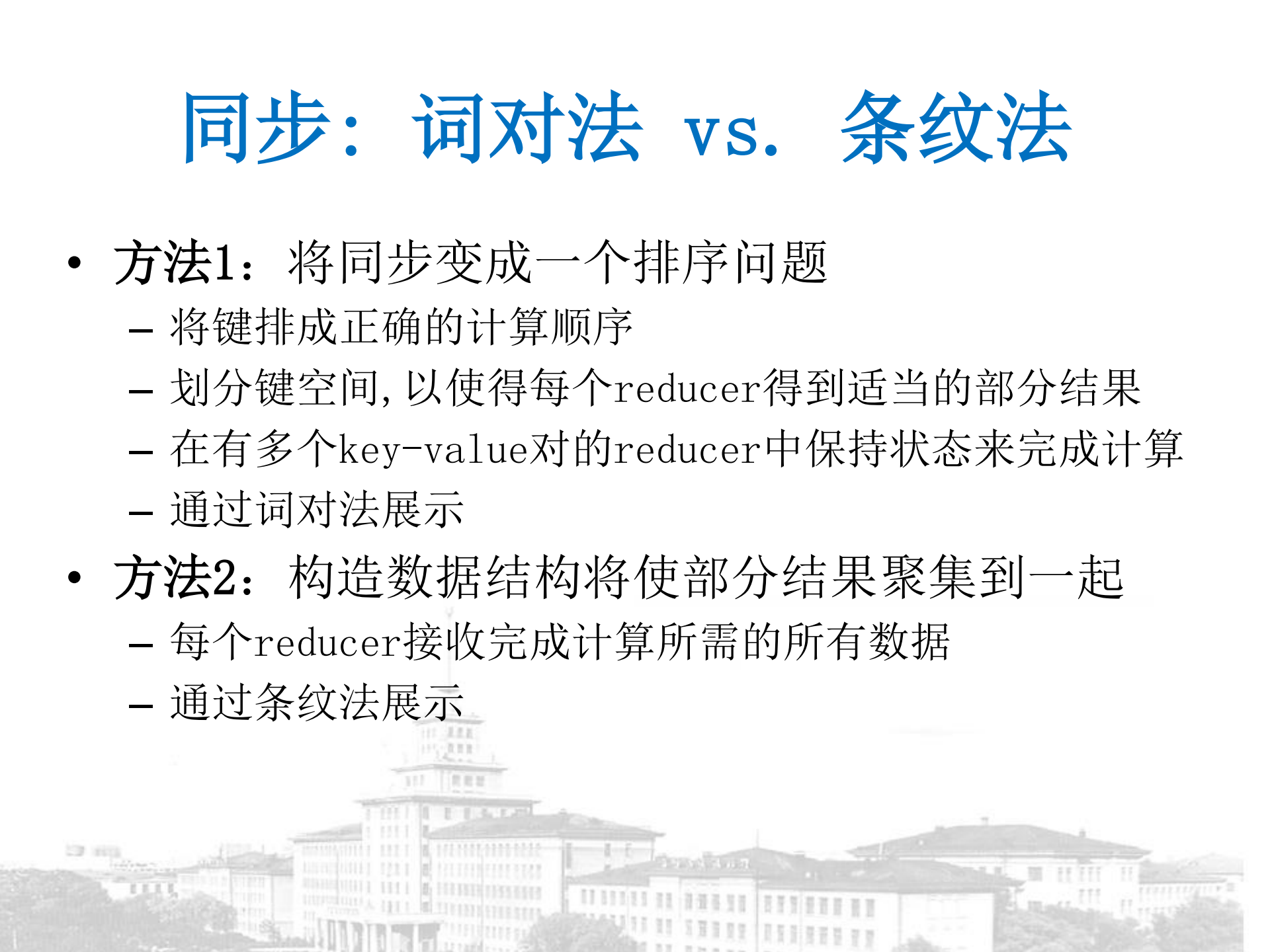
(a, b_1) → 3
(a, b_2) → 12
(a, b_3) → 7
(a, b_4) → 1
…

(a, b_1) → 3 / 32
(a, b_2) → 12 / 32
(a, b_3) → 7 / 32
(a, b_4) → 1 / 32
…

- 对于这个工作:
  - 必须为mapper中的每个b_n传递额外的 (a, *)
  - 必须确定所有a被传递到同一个reducer(使用partitioner)
  - 必须确定 (a, *) 先到达 (定义顺序)
  - 必须在涉及不同key-value对的reducer中保存状态

同步：词对法 vs. 条纹法

- **方法1**：将同步变成一个排序问题
  - 将键排成正确的计算顺序
  - 划分键空间,以使得每个reducer得到适当的部分结果
  - 在有多个key-value对的reducer中保持状态来完成计算
  - 通过词对法展示
- **方法2**：构造数据结构将使部分结果聚集到一起
  - 每个reducer接收完成计算所需的所有数据
  - 通过条纹法展示



概括:同步工具

- **聪明设计的数据结构**
  - 把数据放在一起
- **中间键的排序顺序**
  - 控制reducer处理键的顺序
- **划分器**
  - 控制哪个reducer处理哪个键
- **在mapper和reducer中保存状态**
  - 捕获多个键和值的依赖关系

话题和Tradeoff

- Key-value对的数量
  - 创建对象的开销
  - 对整个网络排序和洗牌的开销
- 每个key-value对的大小
  - 序列化/非序列化的代价
- 本地聚合
  - 执行本地聚合机会不同
  - Combiner造成很大不同
  - combiner vs. in-mapper组合
  - RAM vs. 磁盘 vs. 网络

致谢

- 本讲义部分内容来自于Jimmy Lin的讲义