



中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

SSE316 : 云计算技术 Cloud Computing Technology

陈壮彬

软件工程学院

<https://zbchern.github.io/sse316.html>



云资源管理

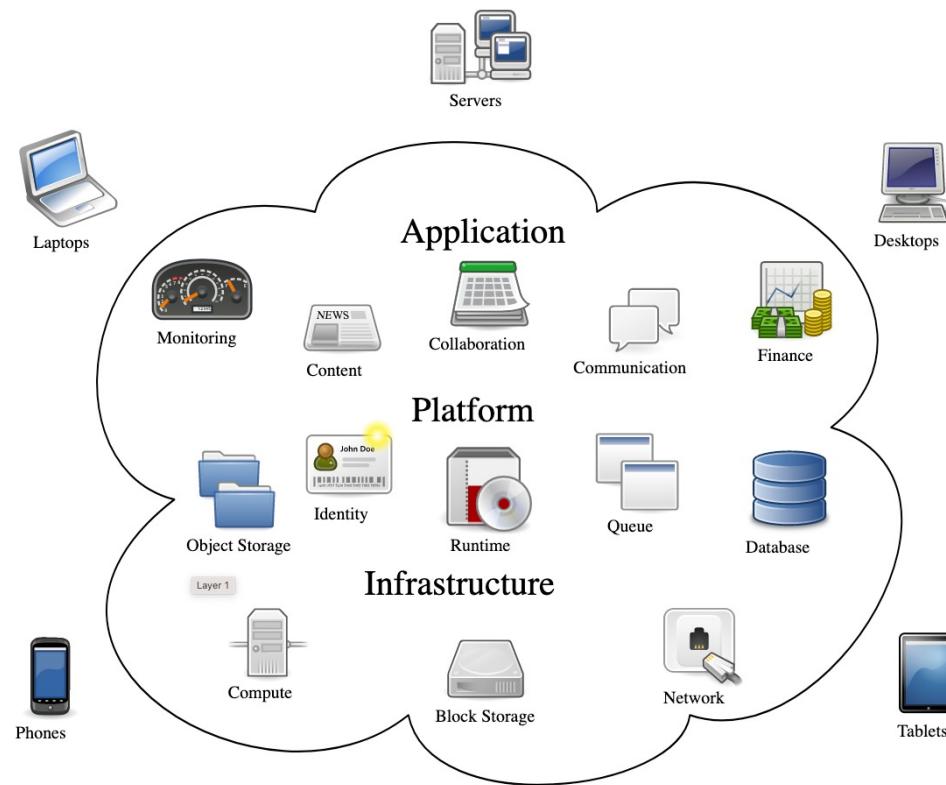
- ❖ 负载均衡策略
- ❖ 仓库级计算机
- ❖ 集群资源调度



云资源管理

- ❖ 负载均衡策略
- ❖ 仓库级计算机
- ❖ 集群资源调度

云IT资源管理



由于**云基础设施的规模**以及**云系统与大量用户之间不可预测的交互**，**云资源管理非常具有挑战性。**

云资源管理策略



1. 准入控制

2. 资源弹性伸缩

3. 负载均衡

4. 能源优化

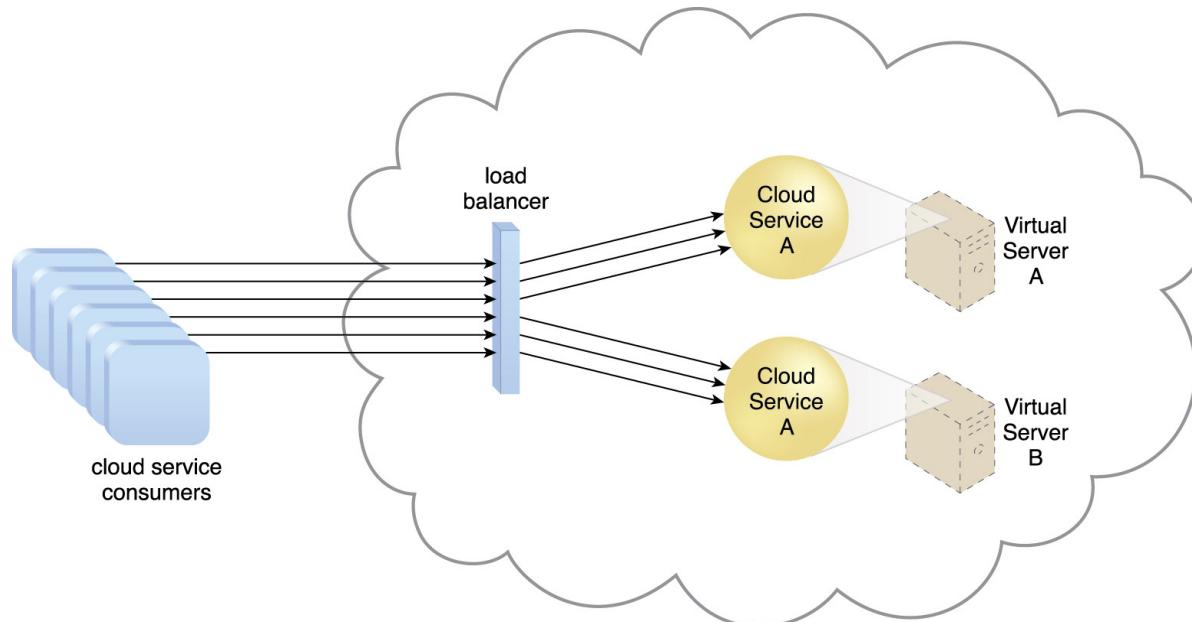
5. QoS保证



什么是负载均衡？



负载均衡 (load balancing) 通常作为一种服务提供，用于管理来自用户的请求并将它们分配给多个服务器，由云服务提供商或企业内部搭建。



负载均衡的优势



提高可用性

自动检测故障并重定向流量来提高系统的容错能力。



提高可扩展性

负载均衡器可以通过增加服务器数量来扩展系统的处理能力。



提升性能

通过减少每个服务器的负载，提高系统的性能和响应速度。



提高安全性

在负载均衡器前添加防火墙、入侵检测系统等安全设备，隐藏服务器细节等。



能耗与能效



- 是不是只需要考虑将工作负载尽量均衡就好？

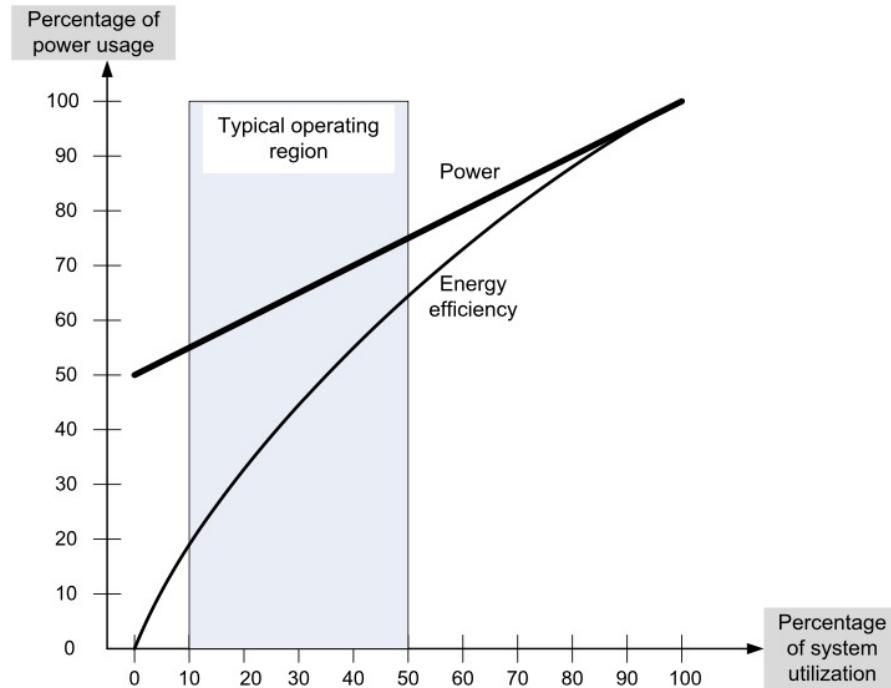


FIGURE 9.1

Even when power requirements scale linearly with the load, the energy efficiency of a computing system is not a linear function of the load. When idle, a system may use 50% of the power corresponding to the full load. Data collected over a long period of time shows that the typical operating region for the servers at a data center is the range 10% to 50% of system utilization [55].

负载均衡算法（1）



静态算法

在负载均衡器预先配置的算法，不随着系统负载的变化而自动调整

- ✓ 随机算法 (Random) : 随机选择一个后端服务器来处理请求
- ✓ 轮询算法 (Round Robin) : 将请求依次分配给后端服务器，每个服务器处理相同数量的请求
- ✓ 加权轮询算法 (Weighted Round Robin) : 与轮询算法类似，但是每个服务器有一个权重值，表示每个服务器处理请求的能力

负载均衡算法（2）

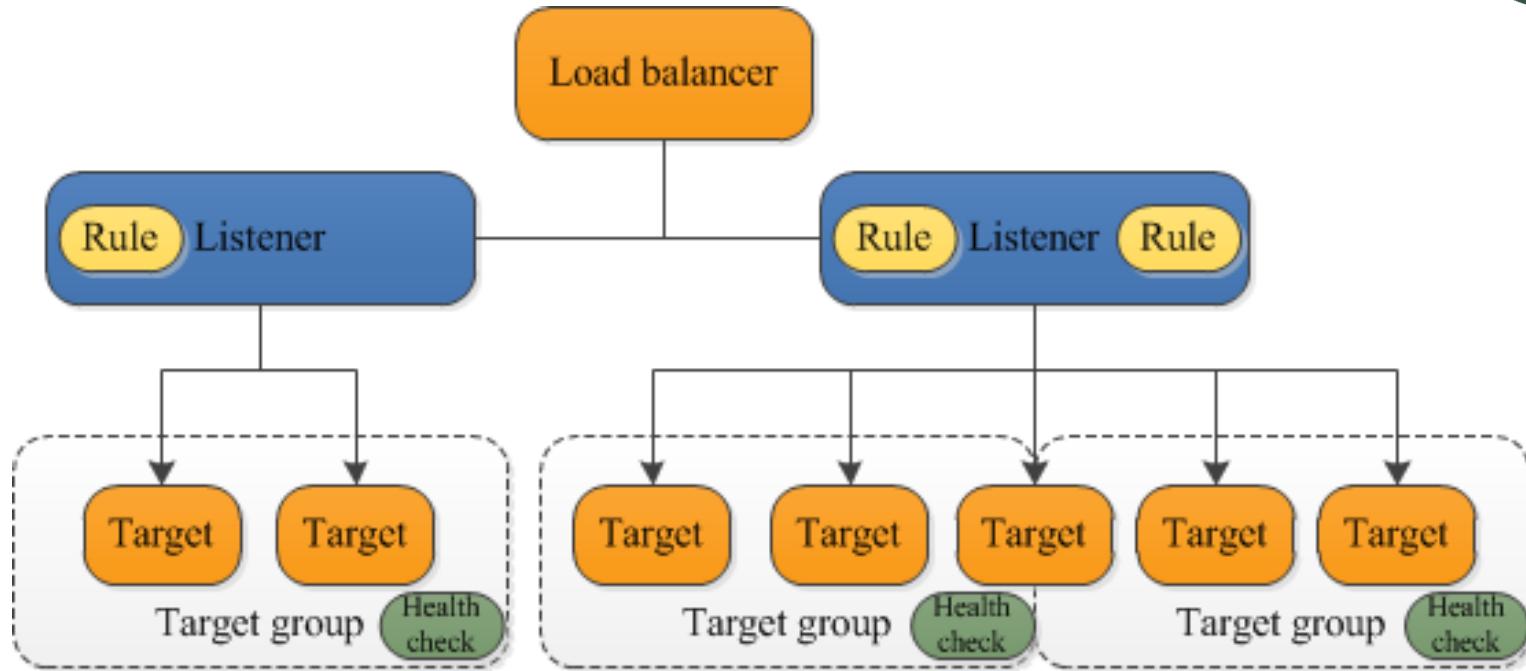


动态算法

实时监测后端服务器的负载情况动态地选择服务器来处理请求

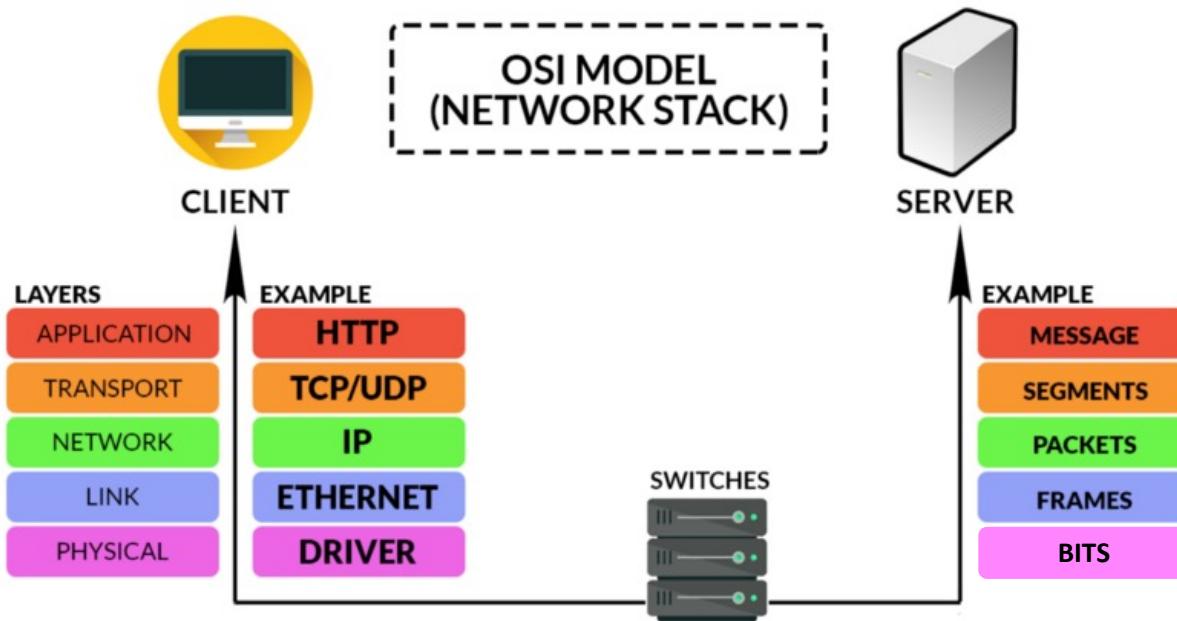
- ✓ 最小连接数算法 (Least Connections) : 将请求分配给当前连接数最少的服务器
- ✓ 最短响应时间算法 (Shortest Response Time) : 将请求分配给响应时间最短的服务器
- ✓ 基于哈希的算法 (Hash-based) : 根据请求的哈希值将请求分配给特定的服务器，确保相同请求总是被分配到同一个服务器上

负载均衡技术



Targets 可以是物理/虚拟服务器、容器和 IP 地址等，横跨单个或多个可用区。

负载均衡的种类



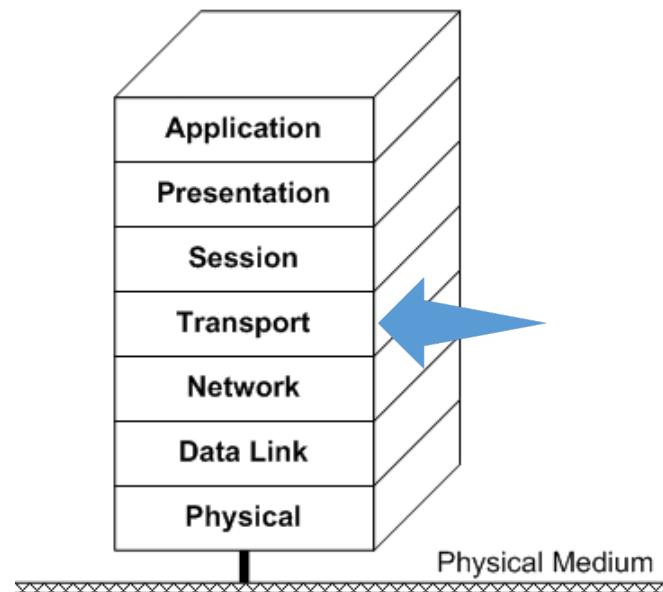
Listener 可以设置在整个请求链路中的不同节点，包括：

- 网络负载均衡 (Network load balancing)
- 网关负载均衡 (Gateway load balancing)
- 应用负载均衡 (Application load balancing)
-

网络负载均衡



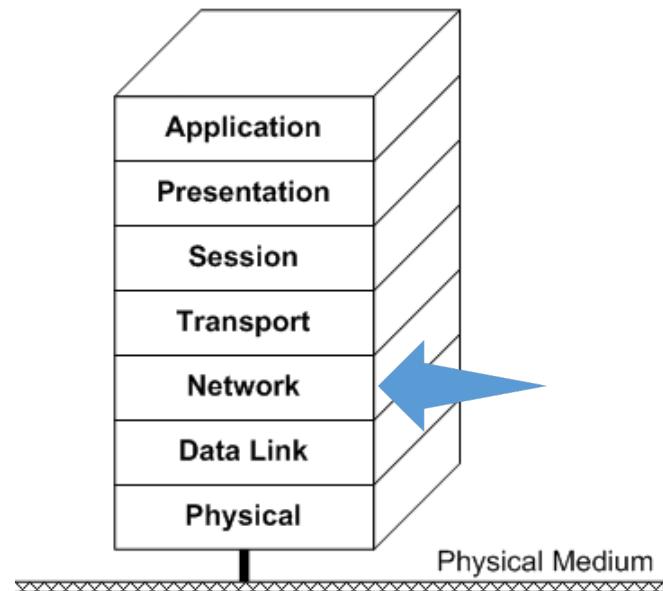
- 网络负载均衡主要针对网络流量进行平衡，
主要在传输层（Layer 4）上，如TCP和UDP
协议
- 通过IP地址、端口号、协议、请求类型等进
行负载均衡计算
- 适合大规模、无状态的Web应用程序、数据
库、DNS等服务



网关负载均衡



- 将负载均衡功能集成到网关设备中，主要在 **The OSI Reference Model** 网络层 (Layer 3) 上
- 可通过硬件设备（如路由器、交换机、防火墙等）或软件（如代理服务器、网关服务器等）实现
- 根据源和目的IP地址进行负载均衡，有些网关也使用端口号

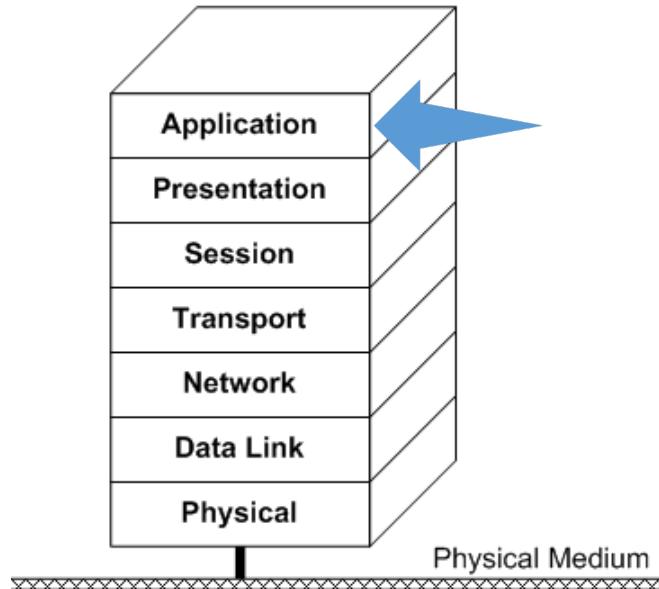


应用负载均衡



- 应用层负载均衡是一种将负载均衡应用到应用层的负载均衡方法
- 更精细地控制请求的负载均衡，可根据请求的内容、用户的地理位置、请求的优先级等
- 应用层负载均衡器通常在请求的首部或者主体中查找关键字，常见的应用层负载均衡协议包括HTTP、HTTPS、SMTP等

The OSI Reference Model

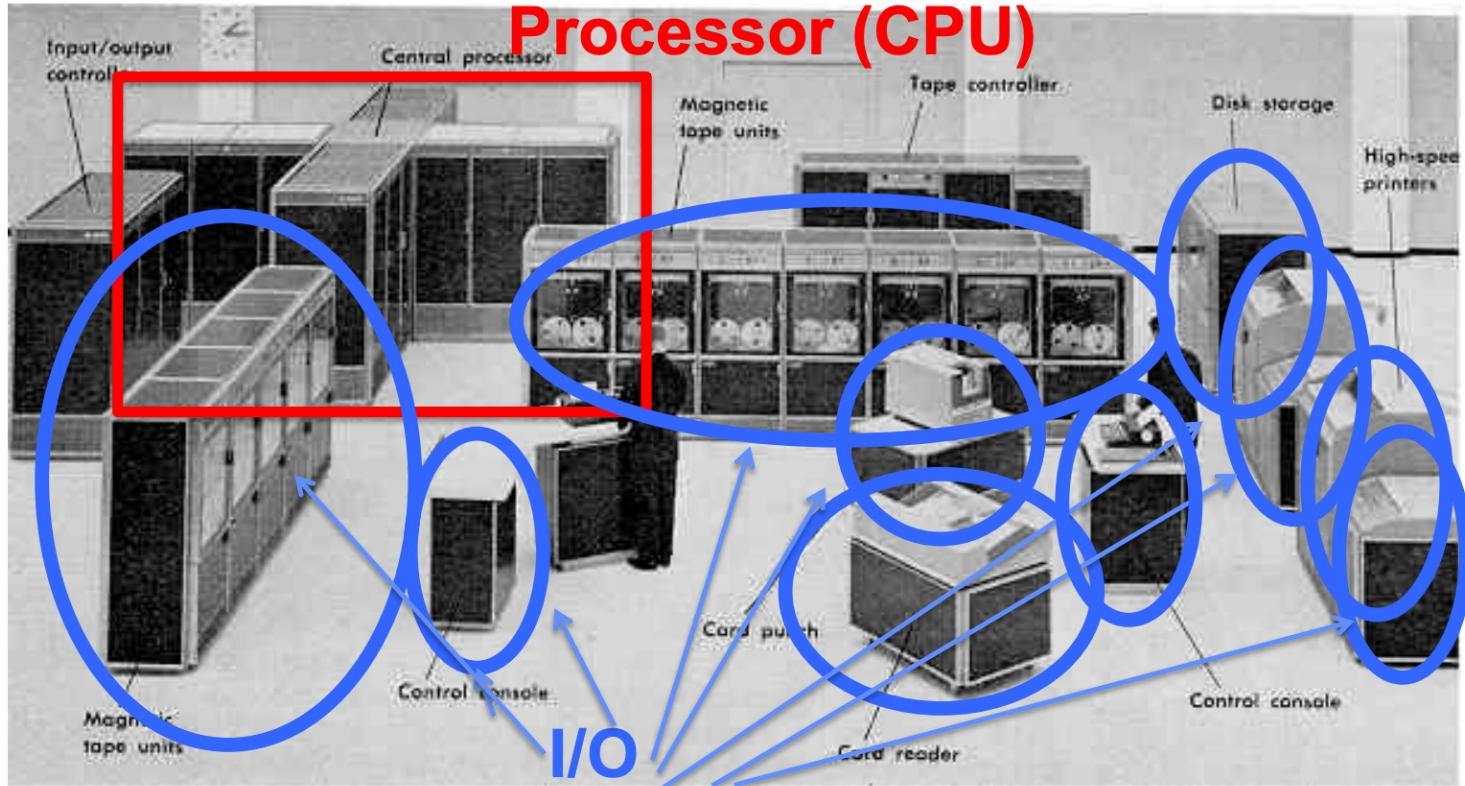




云资源管理

- ❖ 负载均衡策略
- ❖ 仓库级计算机
- ❖ 集群资源调度

大型机时代：1950s-60s



“Big Iron”: IBM, UNIVAC, ... build \$1M computers
for businesses → COBOL, Fortran, timesharing OS

迷你计算机时代：1970s



**Using integrated circuits, Digital, HP... build \$10k
computers for labs, universities → C, UNIX OS**

个人计算机时代：1980s-2000s



Using microprocessors, Apple, IBM, ... build \$1k computer
for 1 person → **Basic, Java, Windows OS**

云计算时代：2000s至今



Personal Mobile Devices (PMD):
Relying on wireless networking, Apple, Nokia, ... build \$500 smartphone and tablet computers for individuals
→ Objective C, Java, Android OS + iOS

Cloud Computing:
Using Local Area Networks, Amazon, Google, ... build \$200M

Warehouse Scale Computers
with 100,000 servers for Internet Services for PMDs

→ MapReduce/Spark, Ruby on Rails





为什么需要仓库级计算机？

- **数据规模的增长**

- ✓ 现代社会产生了大量数据，特别是互联网、物联网等技术的普及，数据量呈爆炸式增长

- **处理能力的需求**

- ✓ 对于很多数据密集型应用（比如机器学习、数据挖掘等）需要高效的计算能力和海量的存储空间

- **云提供商低成本考虑**

- ✓ 更快的应用开发
 - 管理系统配置
 - 简化的系统升级和维护
 - 存储和其他资源的单一系统视图
 - ✓ 通过用户间共享硬件降低硬件成本
 - ✓ 通过摊销硬件和存储管理成本降低成本

机器学习应用

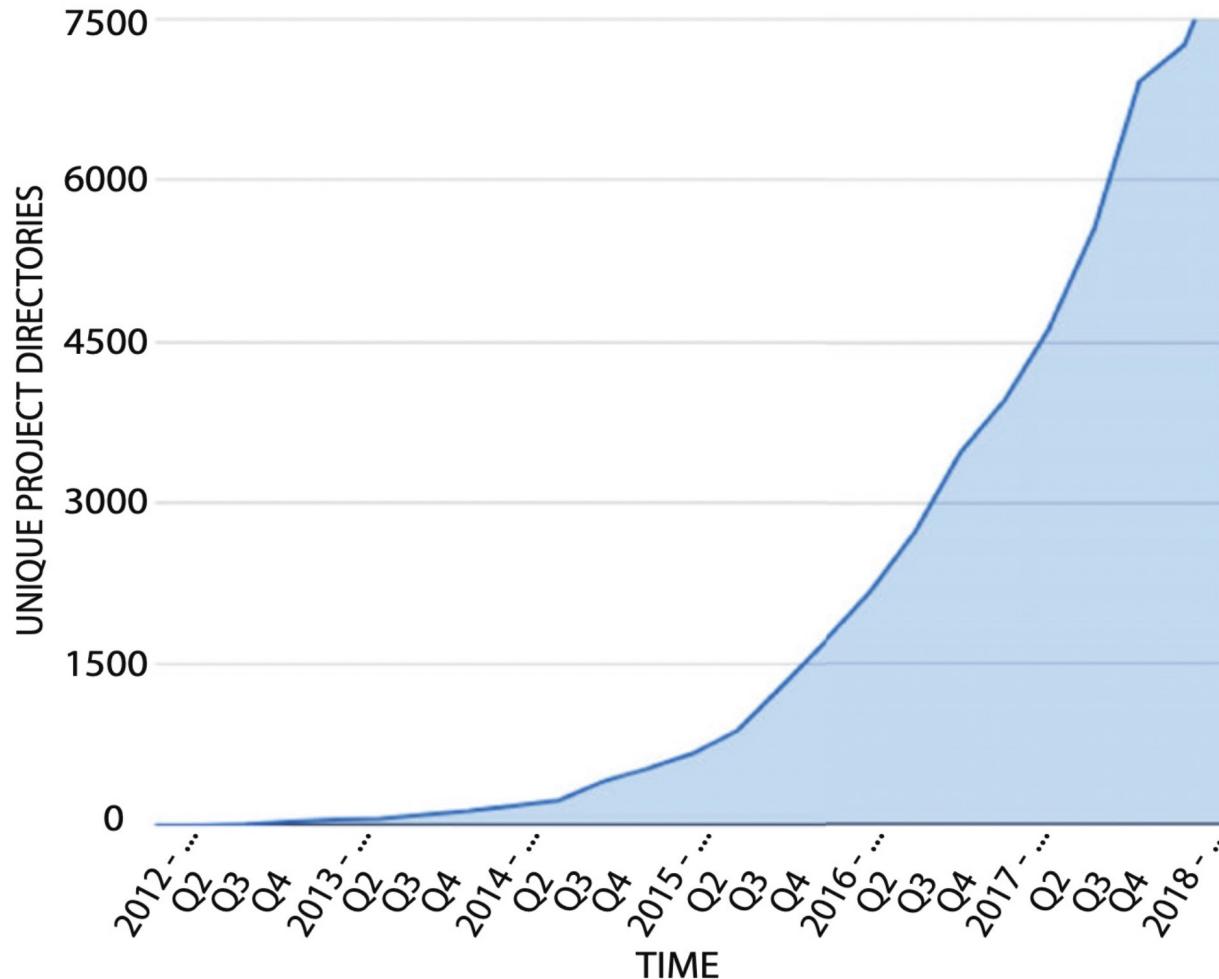
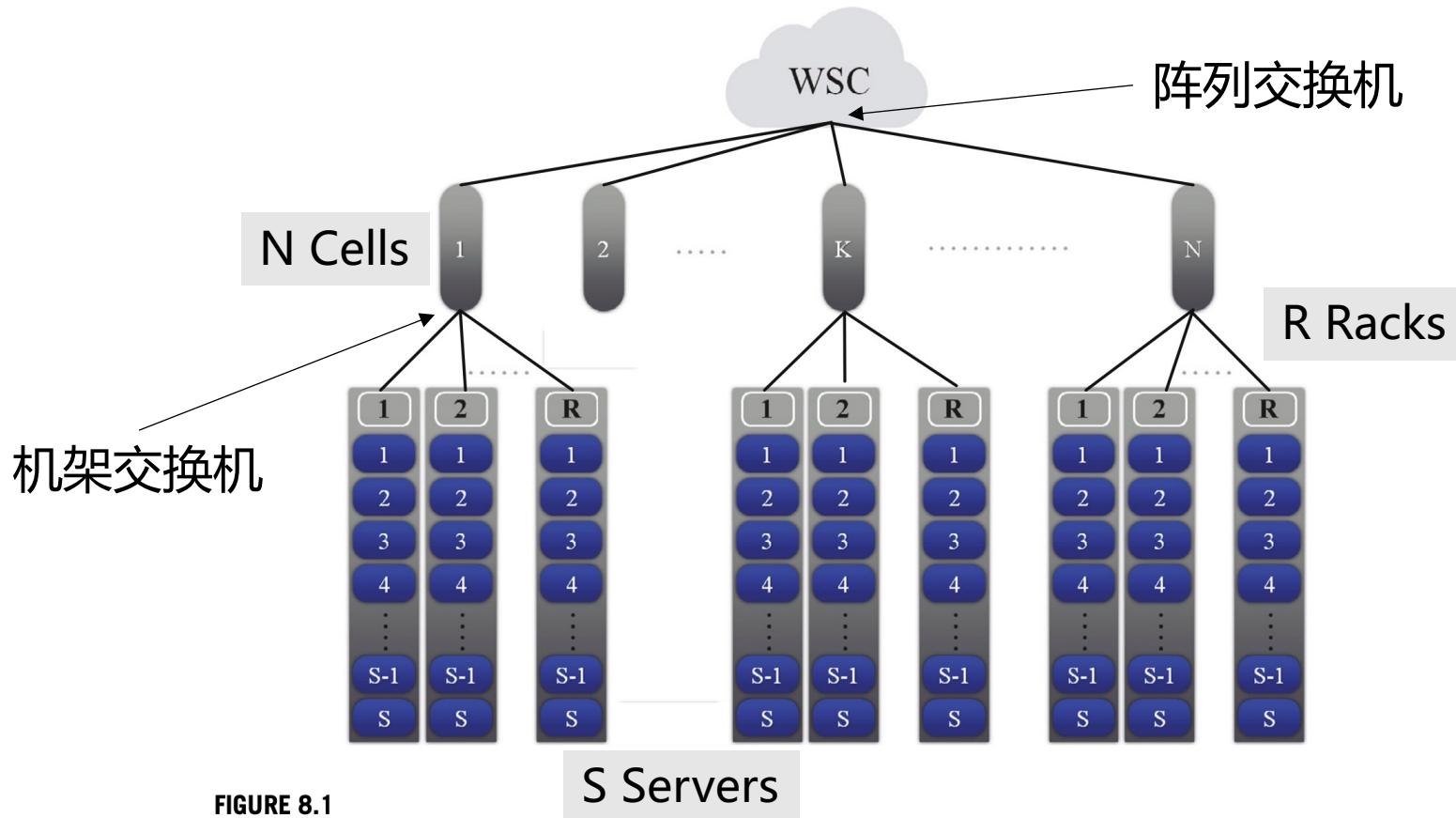


Figure 2.6: Growth of machine learning at Google.

仓库级计算机架构



- 是一种网络层次结构，将服务器（ Server ）、机架（ Rack ）和单元（ Cell ）连接在一起



The organization of a WSC with N cells, R racks, and S servers per rack.

WSC应用的特点



- 数据密集型
 - ✓ 仓库级计算机的主要任务是处理大量的数据
 - ✓ 需要处理大量的数据输入和输出
- 并行计算
 - ✓ 仓库级计算机通常包含大量的处理器和内存，可以支持大规模的并行计算，以提高处理效率
- 分布式存储
 - ✓ 由于需要处理大量的数据，仓库级计算机通常采用分布式存储系统，将数据存储在多个磁盘上



云资源管理

- ❖ 负载均衡策略
- ❖ 仓库级计算机
- ❖ 集群资源调度

计算机云的调度



调度是计算资源管理的一个关键组件，负责在多个级别上进行资源共享/多路复用。

- 一个服务器可以在多个虚拟机之间共享
- 一个虚拟机可以支持多个应用程序
- 每个应用程序可以由多个线程组成

计算机云调度的需求

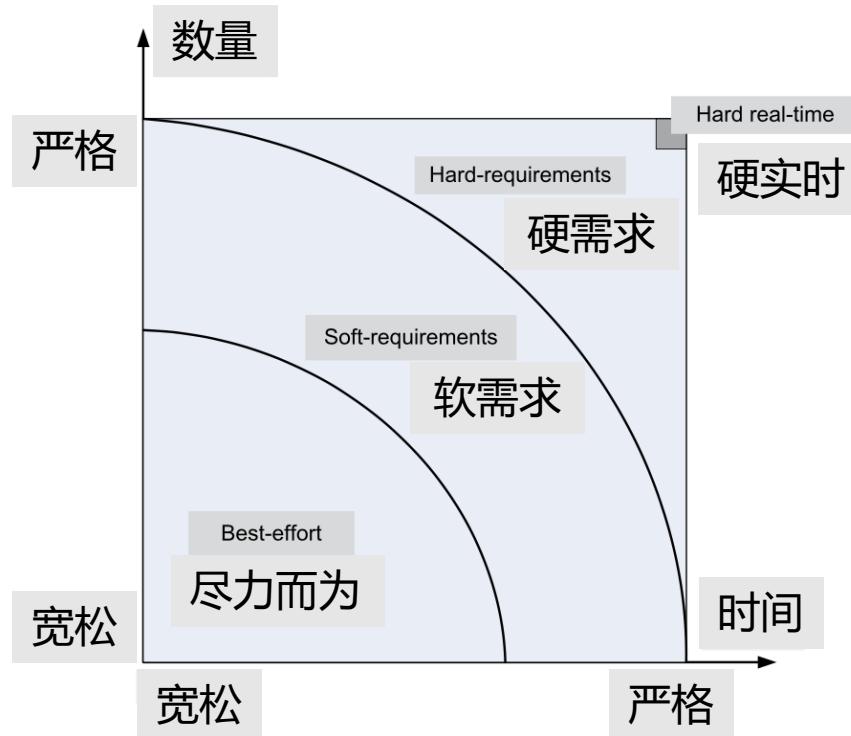
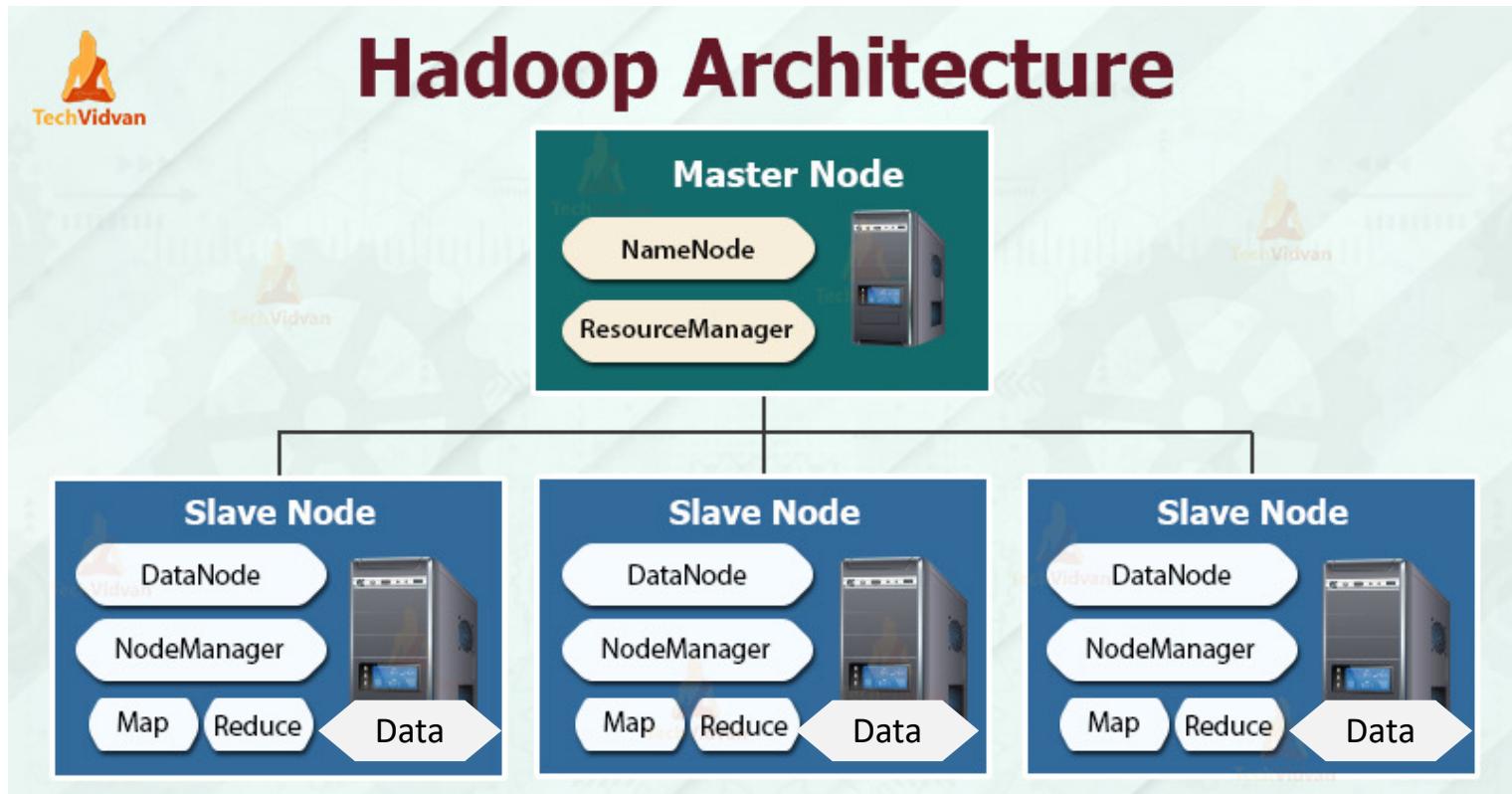


FIGURE 9.7

Resource requirement policies. *Best-effort* policies do not impose requirements regarding either the amount of resources allocated to an application, or the timing when an application is scheduled. *Soft-requirements* policies require statistically guaranteed amounts of resources and timing constraints. *Hard-requirements* policies demand strict timing and precise amounts of resources.

Hadoop架构



Hadoop中的slot



- 在 Hadoop 中，一个 “slot”指的是可用于运行任务或作业的处理单元
- 每个 slave 节点都有固定数量的 map slot 和 reduce slot 可用于运行 MapReduce 作业
- NodeManager 每隔几秒通过 heartbeat 通知 Master 节点 Slave 节点的 slot 剩余情况

Hadoop运行大数据应用



- 用户提交 job，包含一个 map 函数和一个 reduce 函数
- Hadoop 将一个 job 拆成多个 task，每个 map task 处理一个 input 数据块，产生中间结果，以 key-value 的形式保存
- Reduce task 基于 map task 产生的中间结果生产最终的输出

多租户模式



- 像MapReduce 和 Dryad 这样的集群计算系统最初是针对web 索引等批处理作业进行优化的
- 然而，最近出现了另一个用例：在多个用户之间共享集群，这些 用户在**公共数据集**上混合运行长批处理作业和短交互查询
- 当某个 slave 节点有空闲的 slot，master 节点会分配给某个 job 中的某个 task

如何在多租户之间调度集群的计算资源？



传统处理器调度策略



- 先到先得 (First Come First Served)
- 最短任务优先 (Shortest Job First)
- 最短完成时间任务优先 (Shortest Time-to-Completion First)
- 时间片轮转 (Round Robin)

集群资源调度的两个目标



公平分享
Fair sharing

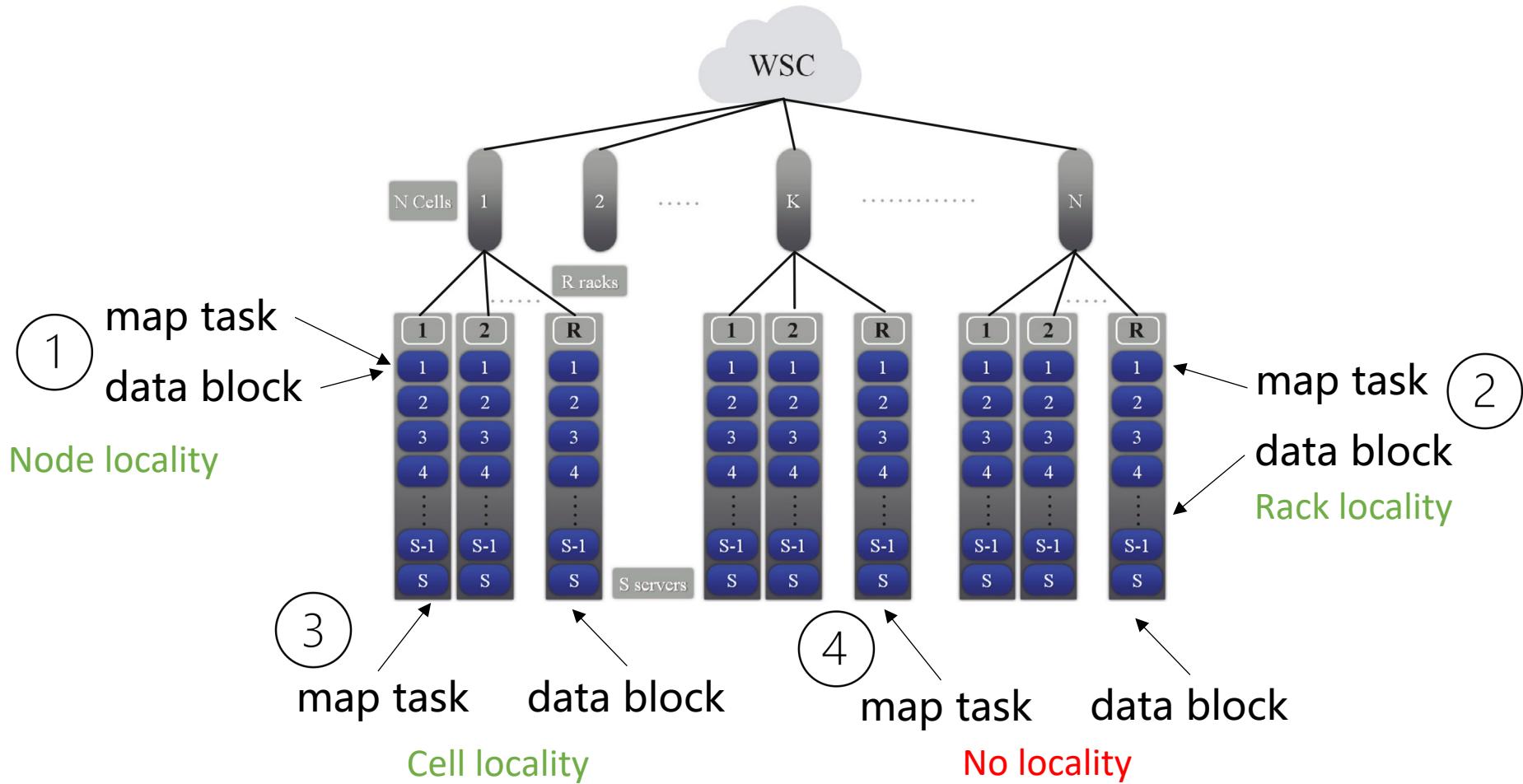
数据局部性
Data locality

最大-最小公平原则



- 最大-最小公平原则 (max-min fair share) : 满足有“小需求”的用户所需的资源，并将未使用的资源平均分配给有“大需求”的用户
 - ✓ 资源按需求增加的顺序分配
 - ✓ 没有用户的资源份额大于其需求
 - ✓ 需求未得到满足的用户可获得同等份额的剩余资源
- 例子：资源总量为10，四个用户的需求为2, 2.6, 4, 5
 - ✓ 将资源平均分成四份，每份2.5
 - ✓ 用户一仅需2，剩余0.5
 - ✓ 将0.5均匀分给其他三个用户，每份0.17
 - ✓ 用户二得2.67，仅需2.6，剩余0.07
 - ✓ 将0.07均分给其他两个用户，每份0.035
 - ✓ 最终四个用户分别得到：2, 2.6, 2.7, 2.7

数据局部性



在Hadoop中实现公平原则



- 总是将可用的 slot 分配给拥有最少 task 的 job
- 只要系统中的 slot 以足够快的速度变得可用，即可满足最大-最小公平原则

Algorithm 1 Naïve Fair Sharing

when a heartbeat is received from node n :

```
if  $n$  has a free slot then
    sort jobs in increasing order of number of running tasks
    for  $j$  in jobs do
        if  $j$  has unlaunched task  $t$  with data on  $n$  then
            launch  $t$  on  $n$ 
        else if  $j$  has unlaunched task  $t$  then
            launch  $t$  on  $n$ 
        end if
    end for
end if
```

然而，此方案下数据局部性难以保证。

公平原则与数据局部性的矛盾



- 进一步分析为什么Algorithm 1无法保证数据局部性

- ✓ Head-of-line Scheduling

- 当作业需要读取的数据块数量较少时，不太可能有数据局部性
 - 例如，一个在10%的节点上具有数据的作业只能实现10%的位置

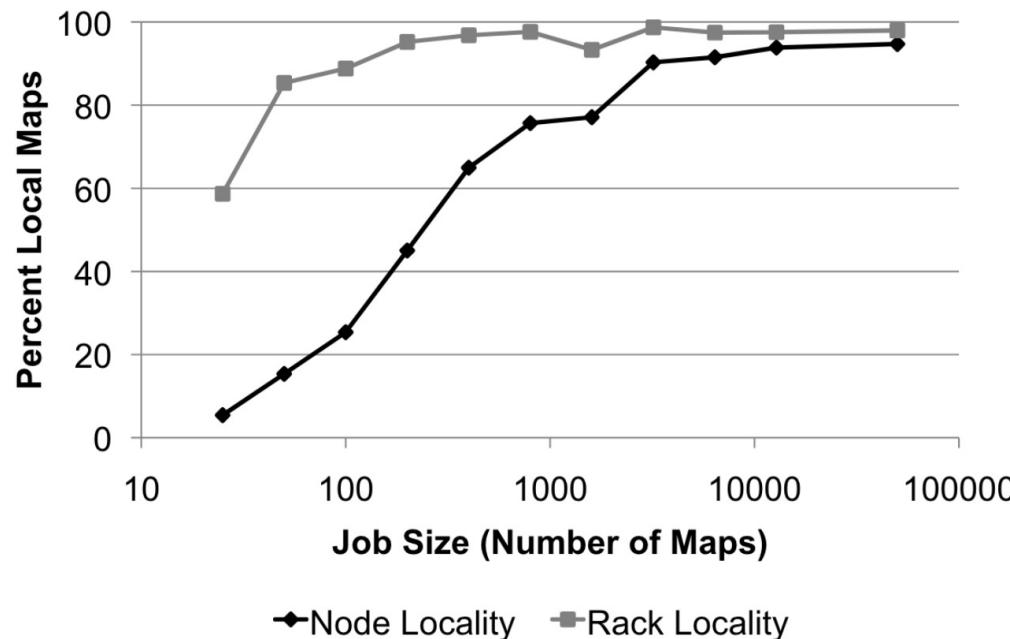


Figure 2: Data locality vs. job size in production at Facebook.

公平原则与数据局部性的矛盾



- 进一步分析为什么Algorithm 1无法保证数据局部性

- ✓ Sticky Slots

- 一个作业倾向于被重复分配到同一个节点，即使是大的 job 也会发生
 - 例如，假设有 10 个 job，每个作业有 10 个 task。假设作业 j 在节点 n 上完成了一项任务，节点 n 现在请求新的 task。此时，j 有 9 个正在运行的任务，而所有其他作业都有 10 个。因此，Algorithm 1 将再次运行 j 的 task

计算数据局部性的概率



- 假设每个 slave 节点有 L 个 slots
- Job j 有比例为 f 的机会使用集群（中的slots）
- 每个数据块有 R 个备份
- 集群中的某个 slot 不被 job j 使用的概率为
$$(1 - f)$$
- 给定任意一个 j 的数据块， j 不能实现数据局部性的概率为
$$(1 - f)^{RL}$$
因为总共有 RL 个 slots 能使 j 实现数据局部性

数据局部性概率分布

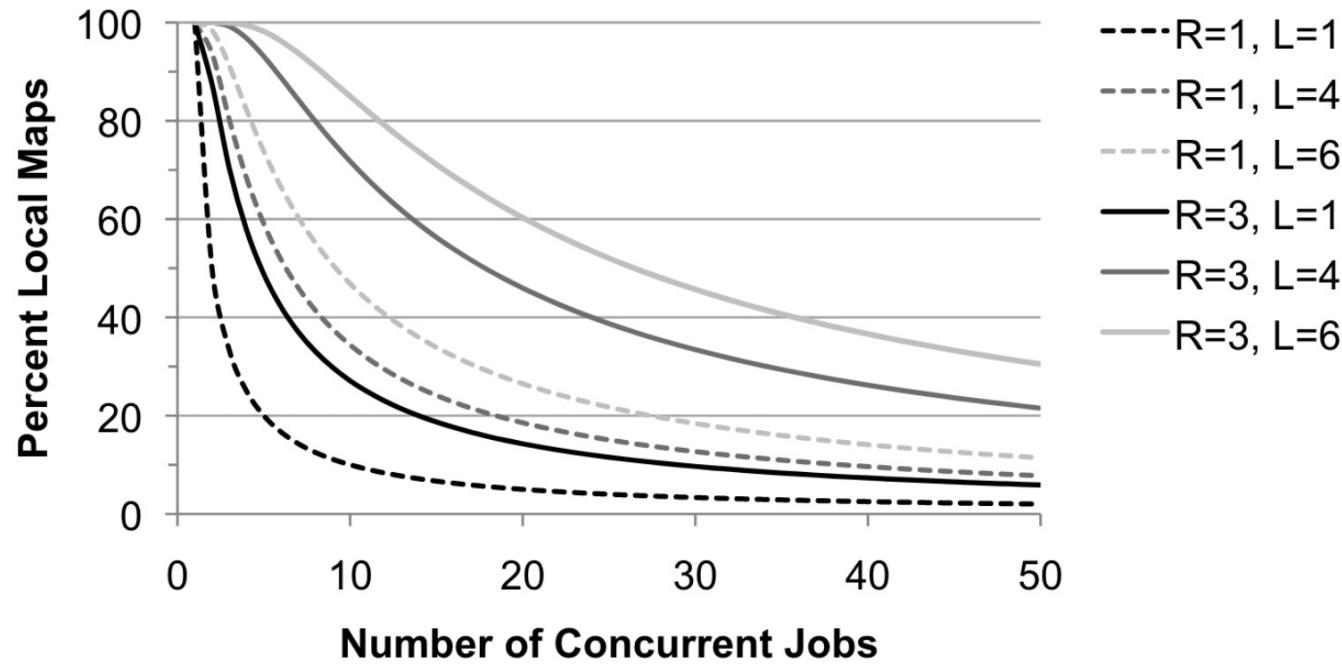


Figure 3: Expected effect of sticky slots on node locality under various values of file replication level (R) and slots per node (L).

Even with large R and L , locality falls below 80% for 15 jobs and below 50% for 30 jobs.

延迟调度算法 Delay Scheduling



Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling

Matei Zaharia

University of California, Berkeley

matei@berkeley.edu

Khaled Elmeleegy

Yahoo! Research

khaled@yahoo-inc.com

Dhruba Borthakur

Facebook Inc

dhruba@facebook.com

Scott Shenker

University of California, Berkeley

shenker@cs.berkeley.edu

Joydeep Sen Sarma

Facebook Inc

jssarma@facebook.com

Ion Stoica

University of California, Berkeley

istoica@cs.berkeley.edu

EuroSys 2010

延迟调度算法 Delay Scheduling



- 当节点请求 task 时，如果排在前面的 job 无法实现数据局部性，则跳过它并查看后续 job
- 如果一个 job 被跳过的时间足够长，则允许它启动非本地任务，以避免饥饿（starvation）

延迟调度的逻辑



- 在大规模集群中，task 以极快的速度完成，资源重新分配的时间比 job 持续的时间小得多
- 因此，我们可以放宽公平原则，当 task 无法实现数据局部性时等一等，数据局部性可期望在短时间内实现

等待对 task 的影响



- 假设一个 job j 需要 F 个 slots 才能完成，task 的平均时长为 T ，集群总共有 S 个 slots，因此，平均 T/S 秒就有一个 slot 可用
- Job j 需要 FT/S 时间获得所有 slots，假设 j 的运行时间为 J
- 则获取所需 slots 的时间可被忽略，当且仅当

$$J \gg \frac{F}{S}T$$

等待时间可忽略的条件



- 下列三种情况至少有一种满足即可忽略等待时间，即

$$J \gg \frac{F}{S} T$$

- ✓ Job 数量多，每个 job 分得的 slots 比例 $f = \frac{F}{S}$ 小
- ✓ Job 包含的 task 数量少，因而 f 也小
- ✓ Job 本身持续时间长

Facebook 中的工作负载情况

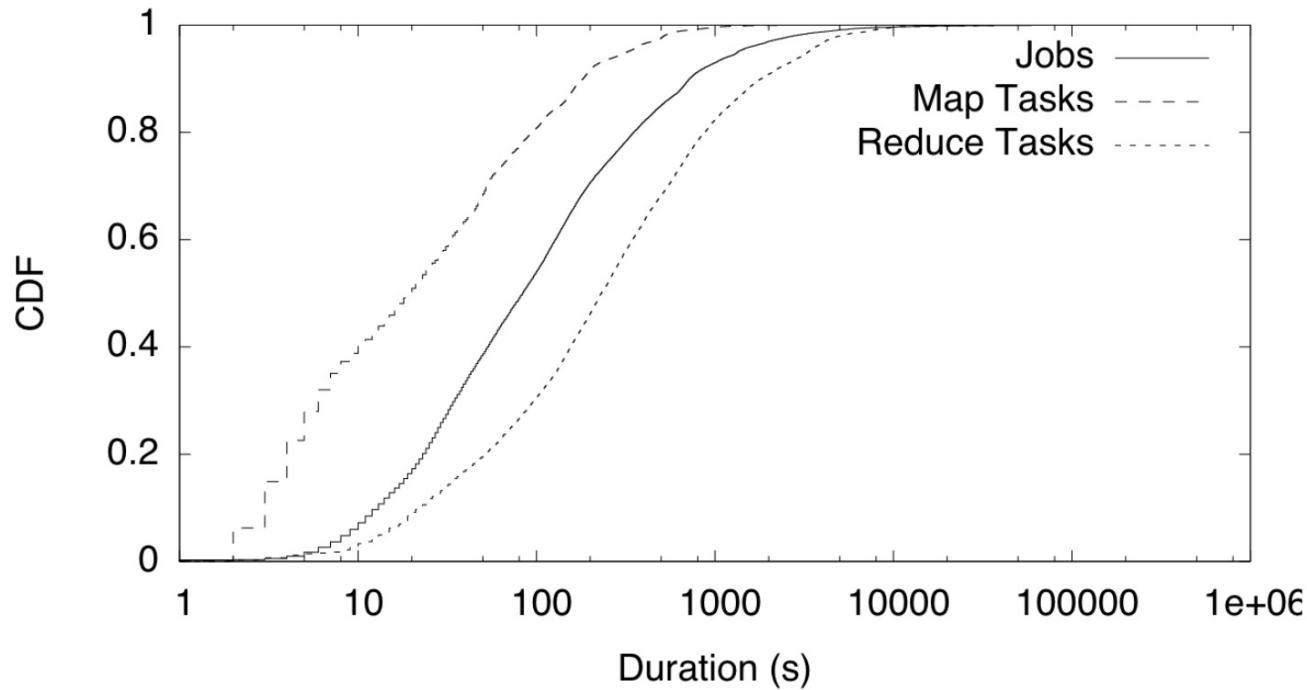


Figure 1: CDF of running times for MapReduce jobs, map tasks and reduce tasks in production at Facebook in October 2009.

在Facebook的工作负载中，大多数 task 都很短，大多数 job 的 task 数量都很少，因此资源重新分配很快。

延迟调度算法实现



Algorithm 2 Fair Sharing with Simple Delay Scheduling

Initialize $j.\text{skipcount}$ to 0 for all jobs j .

when a heartbeat is received from node n :

if n has a free slot **then**

sort jobs in increasing order of number of running tasks

for j in jobs **do**

if j has unlaunched task t with data on n **then**

launch t on n

set $j.\text{skipcount} = 0$

else if j has unlaunched task t **then**

if $j.\text{skipcount} \geq D$ **then**

launch t on n

else

set $j.\text{skipcount} = j.\text{skipcount} + 1$

end if

end if

end for

end if

延迟调度策略的分析



1. 数据的非局部性随 D 的增大呈**指数减小**
2. 实现给定水平的局部性所需的等待量是平均 task 时长的一小部分，并且随着每个节点的 slot 数量**线性减少**

分析条件设定



- 假设集群中节点的数量为 M ，每个结点有 L 个 slot，则 slot 的总数 $S = ML$
- 记 P_j 为包含 job j 的数据块的节点集合
- 则 j 能够实现数据局部性的节点比例为 $p_j = \frac{|P_j|}{M}$
- 为简化分析，假设所有 task 的运行时间为 T

等待时间对数据局部性的影响



- job j 随机得到一个 slot 不可实现数据局部性的概率为

$$(1 - p_j)$$

- 则 job j 等待 D 个 slots 不可实现数据局部性的概率为

$$(1 - p_j)^D$$

- 举例而言，即便 job j 仅有 10% 的节点 ($p_j = 0.1$) 能够使其实现数据局部性，采用 $D = 10$ 就能使得数据局部性的概率为 65%， $D = 40$ 则概率为 99%

等待时长的分析



- 由于 slot 的总数为 S 且 task 时长为 T ，因此平均 T/S 秒就有一个 slot 可用
- 因此，若 job j 在队列头部，则需等待 DT/S 以实现数据局部性，此时间随 S 的增加而线性减小

计算等待时间 D (1)



- 计算实现数据局部性 λ 的期望等待时间 D
- 假设 Job j 的任务数量为 N ，集群的节点数为 M ，每个节点的 slot 数量为 L ，每个数据块的备份为 R
- Job j 数据局部性的期望概率为 j 剩下 $K = N, N - 1, \dots, 1$ 个 task 时概率的平均值



计算等待时间 D (2)

- 当 Job j 剩下 j 个 task 时，给定任意一个节点，其不包含 j 的数据块的概率为

$$\left(1 - \frac{K}{M}\right)^R$$

- 因此，能使 Job j 实现数据局部性的节点占比为

$$p_j = 1 - \left(1 - \frac{K}{M}\right)^R$$

- 若等待 D 时间，则 Job j 实现数据局部性的概率为

$$1 - (1 - p_j)^D = 1 - \left(1 - \frac{K}{M}\right)^{RD} \geq 1 - e^{-RDK/M}$$



计算等待时间 D (3)

- 对 $k = 1, \dots, N$ 取平均，可得当 job j 等待 D 时间，其取得的数据局部性的期望概率为

$$\begin{aligned} l(D) &= \frac{1}{N} \sum_{K=1}^N 1 - e^{-RDK/M} \\ &= 1 - \frac{1}{N} \sum_{K=1}^N e^{-RDK/M} \\ &\geq 1 - \frac{1}{N} \sum_{K=1}^{\infty} e^{-RDK/M} \\ &\geq 1 - \frac{e^{-RD/M}}{N(1 - e^{-RD/M})} \end{aligned}$$

- 求解 $l(D) \geq \lambda$ ，得

$$D \geq -\frac{M}{R} \ln\left(\frac{(1-\lambda)N}{1 + (1-\lambda)N}\right)$$

计算等待时间 D (4)



- 例如，当 $\lambda = 0.95$ ， $N = 20$ ， $R = 3$ 时， D 的值为

$$D \geq 0.23M$$

- 同时，一个 task 需等待的最长时间为

$$\frac{D}{S}T = \frac{D}{LM}T = \frac{0.23}{L}T$$

- 当时 $L = 8$ ，task 等待的时间仅为其执行时间的 **2.8%**



Rack Locality

- 当最佳数据局部性（即node locality）不可得时，可以追求下一级的数据局部性（即rack locality）
- 可通过设置两层的等待策略，等待 D_1 时间实现 node locality，之后等待 D_2 时间实现 rack locality，否则放弃数据局部性

Algorithm 2 Fair Sharing with Simple Delay Scheduling

Initialize $j.\text{skipcount}$ to 0 for all jobs j .

when a heartbeat is received from node n :

```
if  $n$  has a free slot then
    sort jobs in increasing order of number of running tasks
    for  $j$  in jobs do
        if  $j$  has unlaunched task  $t$  with data on  $n$  then
            launch  $t$  on  $n$ 
            set  $j.\text{skipcount} = 0$ 
        else if  $j$  has unlaunched task  $t$  then
            if  $j.\text{skipcount} \geq D$  then
                launch  $t$  on  $n$ 
            else
                set  $j.\text{skipcount} = j.\text{skipcount} + 1$ 
            end if
        end if
    end for
end if
```

等待 D_1 实现 node locality，
等待 D_2 实现 rack locality

实验结果 (1)



- 实验数据

Bin	# Maps	% Jobs at Facebook	# Maps in Benchmark	# Jobs in Benchmark
1	1	39%	1	38
2	2	16%	2	16
3	3–20	14%	10	14
4	21–60	9%	50	8
5	61–150	6%	100	6
6	151–300	6%	200	6
7	301–500	4%	400	4
8	501–1500	4%	800	4
9	> 1501	3%	4800	4

Table 2: Distribution of job sizes (in terms of number of map tasks) at Facebook and in our macrobenchmarks.

实验结果 (2)



- 运行时间 (IO-heavy workload)

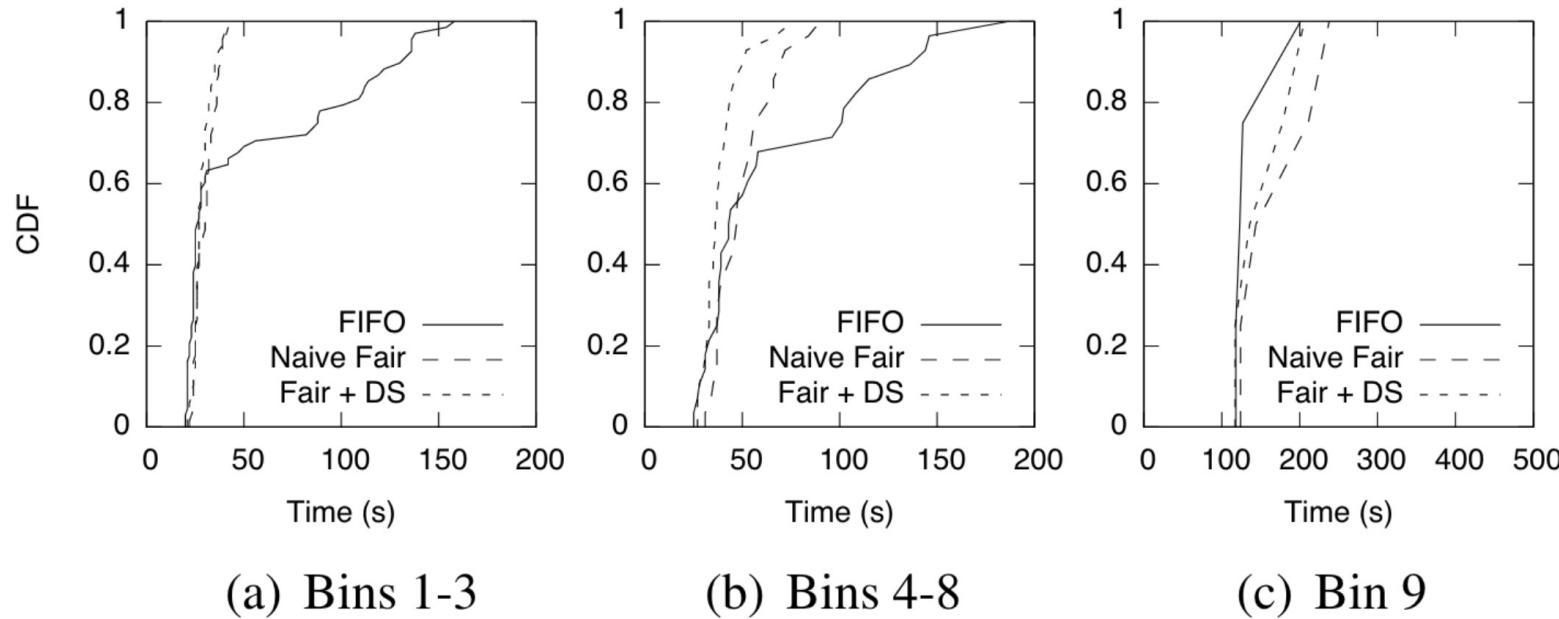


Figure 5: CDFs of running times of jobs in various bin ranges in the IO-heavy workload. Fair sharing greatly improves performance for small jobs, at the cost of slowing the largest jobs. Delay scheduling further improves performance, especially for medium-sized jobs.

实验结果 (3)



- 运行时间 (CPU-heavy workload)

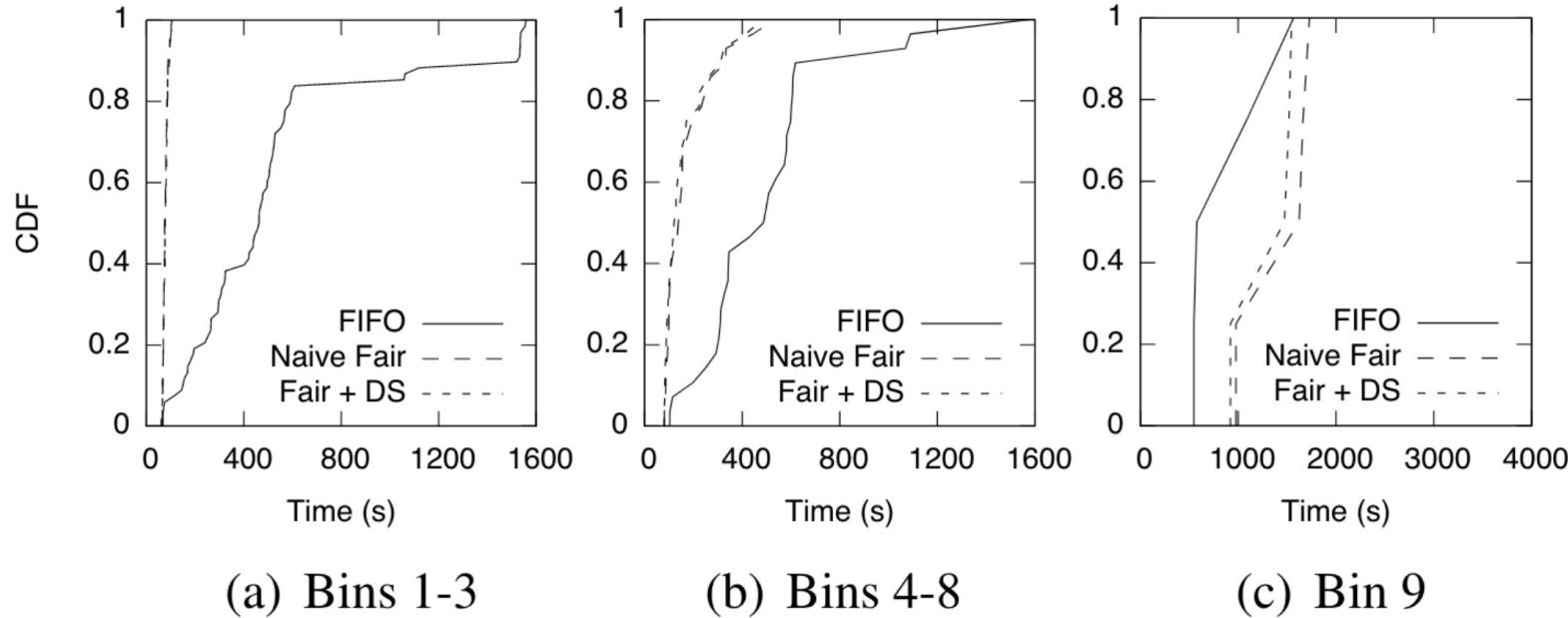


Figure 8: CDFs of running times of jobs in various bin ranges in the CPU-heavy workload. Fair sharing speeds up smaller jobs while slowing down the largest ones, but delay scheduling has little effect because the workload is CPU-bound.

实验结果 (4)



- 数据局部性

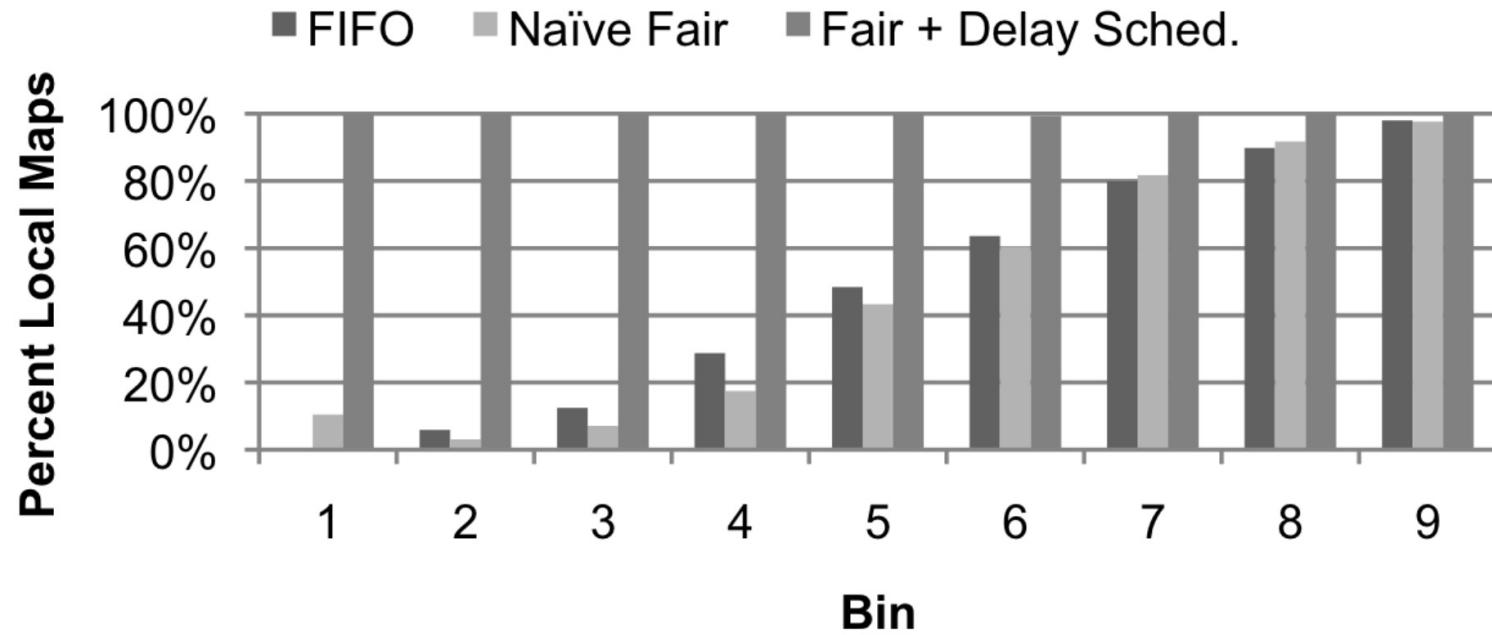


Figure 6: Data locality for each bin in the IO-heavy workload.

实验结果 (5)



- 平均提速

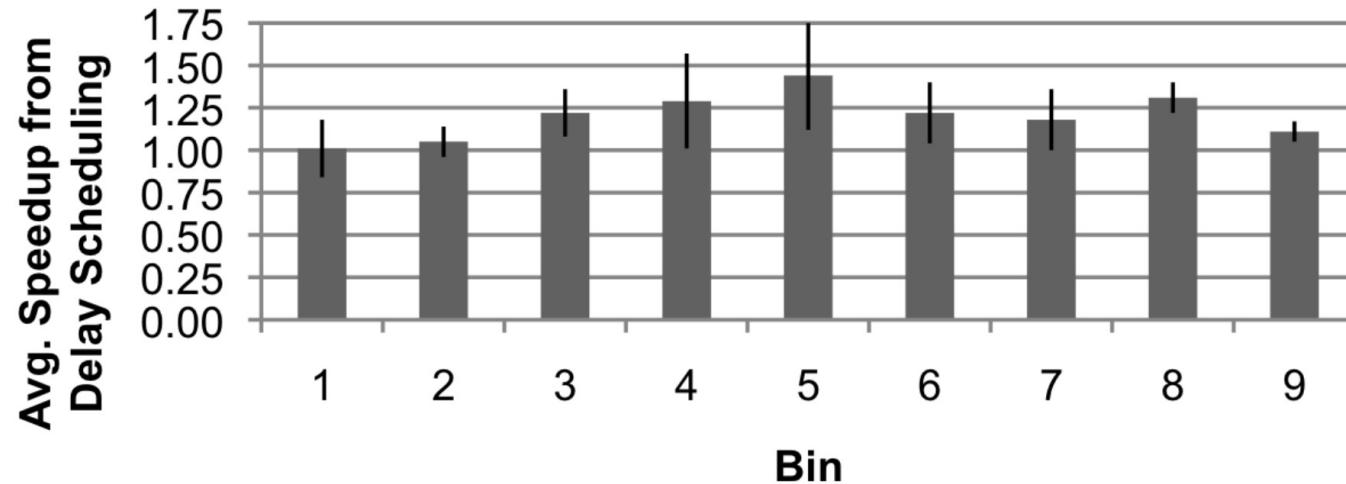


Figure 7: Average speedup of delay scheduling over naïve fair sharing for jobs in each bin in the IO-heavy workload. The black lines show standard deviations.



中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬
软件工程学院

<https://zbchern.github.io/sse316.html>