构建大规模信息检索系统中的挑战

Jeff Dean
Google Fellow
jeff@google.com

翻译：银杏泰克有限公司 郝培强
Tinyfool@gmail.com
http://www.tinydust.net/dev

为什么在信息检索公司工作

- 既有科学又有技术方面的挑战
  - 许多有趣的未解的问题
  - 涵盖计算机科学的诸多领域：
    - 架构，分布式系统，算法，压缩，信息检索，机器学习，界面，等等。
  - 规模远大于大多数其他系统
- 小团队就可以构建上亿用户使用的系统

信息检索系统的尺寸

- 必须在下列工程参数之间平衡取舍：
  - 索引的文档数
  - 每秒查询数
  - 索引的新鲜程度/更新率
  - 查询延迟
  - 每个文档保存的信息
  - 评分/检索算法的复杂度和开销
- 工程难度大致等于这些参数的乘积
- 以上这些参数影响整体的性能，以及单位成本下的性能

1999 和 2009

- 文档数: 约7000万到数百亿 约100倍 约1000倍
- 每日处理查询数: 约3倍
- 索引中每个文档的信息数: 约10000倍
- 更新延迟:从月到分钟 约5倍
- 平均查询延迟: 小于1秒到小于0.2秒 约1000倍
- 更多机器*更快的机器:

唯一不变的是改变本身

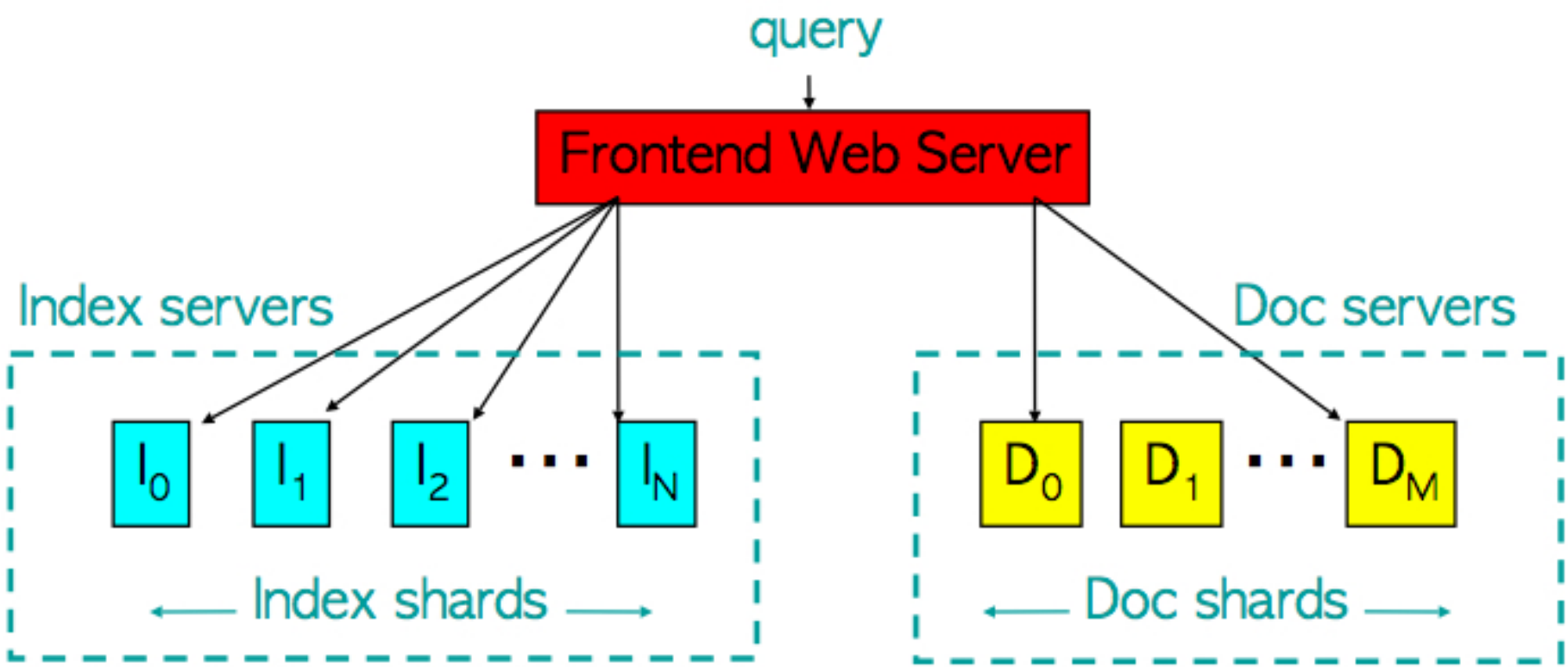
- 参数随时在变
  - 常常是几个数量级的改变
  - 在X规模下正确的设计在10X或100X规模下可能完全是错的
  - …以10倍规模的增长设计系统，在100倍增长之前计划重写系统
- 持续进化：
  - 10年间7个重大版本
  - 经常推出新版本，但是用户完全不知道我们做出了重大改变

其余要讨论的

- Google搜索系统的演化
  - 几代 抓取/索引/服务 系统
  - 底层架构的简要描述
  - 许多许多人一起工作
- 有趣的方向和挑战

1997年的Google(google.stanford.edu)

研究项目，1997年


query
Frontend Web Server
Index servers
Doc servers
I_0
I_1
I_2
I_N
D_0
D_1
D_M
Index shards
Doc shards

索引的分割方法

- 按文档分割：每个分片包含索引的全部文档的一个子集
  - 利：每个分片可以独立处理查询
  - 利：便于保存每个文档的格外信息
  - 利：网络传输量（请求和响应）少
  - 弊：必须在每个分片上执行查询
  - 弊：N个分片，K个词的查询需要O(K*N)次的磁盘寻道
- 按词分割：每个分片包含文档中所有词的一个子集
  - 利：K个词的查询=>最多需要K个分片处理
  - 利：K个词的查询需要O(K)次的磁盘寻道
  - 弊：需要更高的网络带宽
    - 每个匹配文档的每个词的数据必须集合到一起（译注：进行处理）
  - 弊：难以保持基于文档的信息

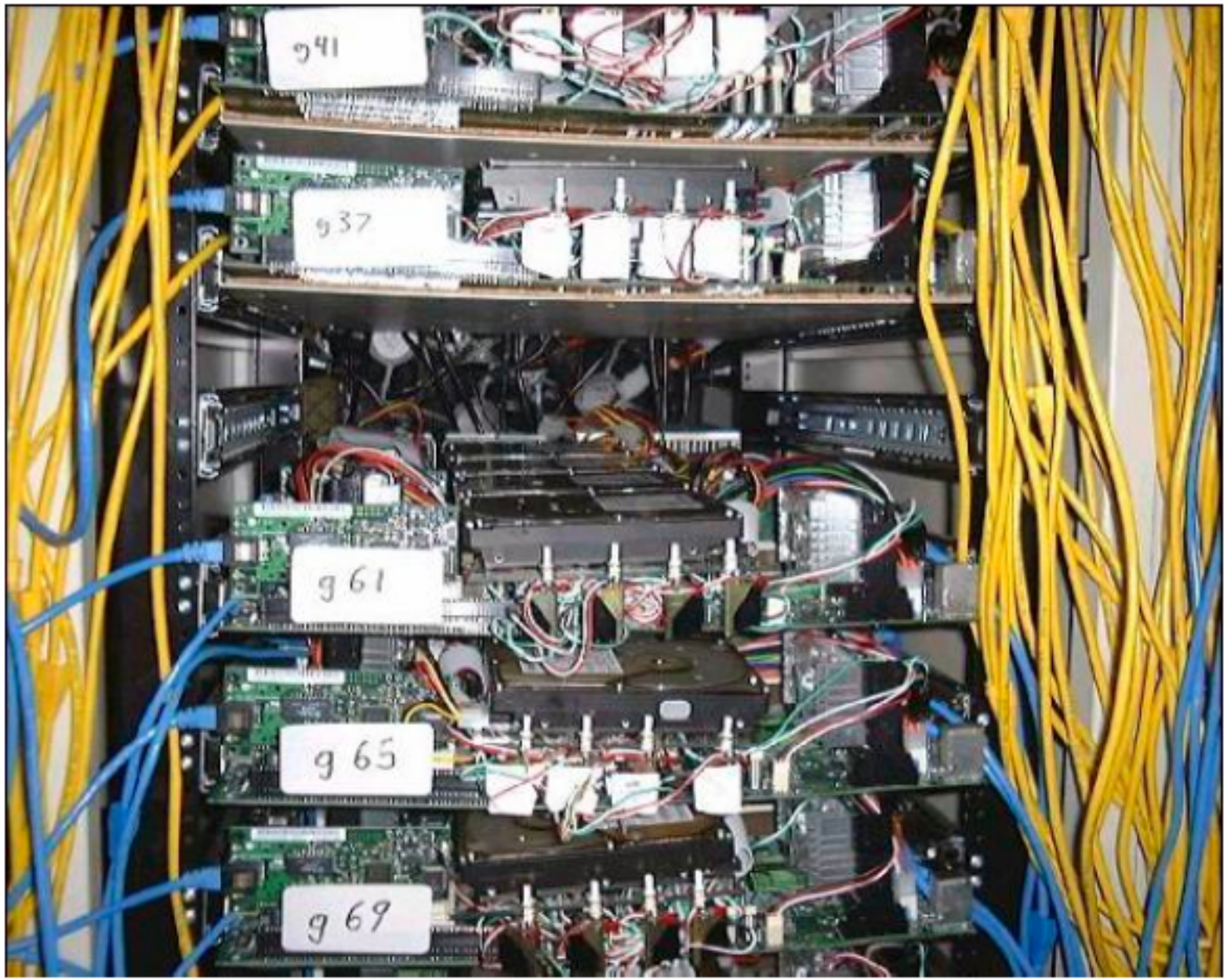
索引的分割方法

针对我们的计算环境，按文档分割更加靠谱

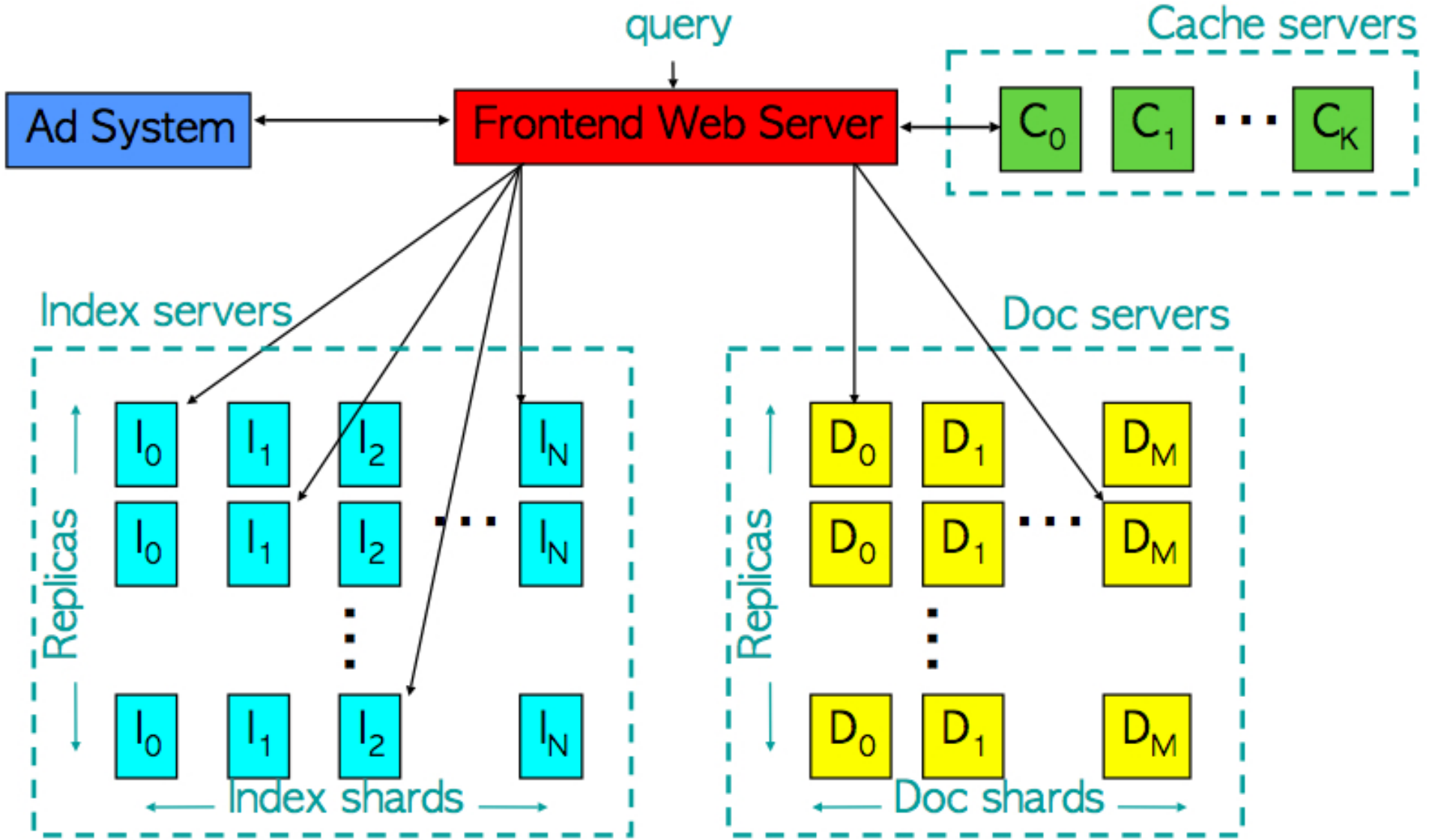
基本原则

- 被赋予较小整数编号(docids)的文档
  - 最好令高质量/重要的文档的编号更小
- 索引服务器
  - 针对给定(查询)返回排序后的(分数，文档编号)的列表
  - 按文档分片
  - 为了承载能力对索引分片进行复制
  - 开销是O(查询数*索引中的文档数)
- 文档服务器
  - 针对给定(文档编号，查询)生成(标题，摘要)
  - 从文档编号映射到硬盘上的文档全文
  - 也按文档分片
  - 开销是O(查询数)

板子(1999)



1999年的服务系统


query
Cache servers
Ad System
Frontend Web Server
C_0 C_1 ... C_K
Index servers
Doc servers
Replicas
I_0 I_1 I_2 ... I_N
D_0 D_1 ... D_M
Index shards
Doc shards

缓存

- 缓存服务器：
  - 既缓存索引返回的结果，也缓存摘要
  - 命中率基本上在30%到60%之间
    - 取决于索引更新的频率，查询的交叉程度，个性化的级别，等等
- 主要的好处：
  - 性能！10个机器做100个或者1000个机器的工作
  - 对于命中的查询，减少了查询延迟
    - 命中缓存的查询，一般比较流行而且计算成本高昂（常用词，很多文档需要评分，等等）
- 注意：索引更新和缓存重建的时候，产生延迟高峰以及性能的下降

爬虫(1998-1999)

- 简单的批处理爬虫系统
  - 从几个链接开始
  - 爬取页面
  - 提取链接，加入队列
  - 得到足够多的页面后停止
- 注意:
  - 不要太频繁的连接任何网站
  - 为没有爬取的页面定优先级
    - 一种办法：在变化中的网站链接图上计算PageRank
  - 高效地维护未爬链接队列
    - 一种办法：保存在分片的数个服务器上
  - 处理

索引(1998-1999)

- 简单的批处理索引系统
  - 基于简单的Unix工具
  - 没有真正的检查点，所以机器故障会令人很痛苦
  - 原始数据没有校验，所以硬件位错误会造成问题
    - 早期机器没有ECC没有奇偶校验使问题更加恶化
    - 没有奇偶校验对1TB数据进行排序：结果是“部分排序”
    - 再次排序:另一个形式的“部分排序”
- “在内存的干扰下编程”
  - 让我们开发一种抽象文件，保存小纪录的校验和，当记录损坏的时候可以跳过问题恢复数据

索引更新(1998-1999)

- 1998-1999:索引更新(一月一次)
  - 等到负载低的时候
  - 令某些副本下线
  - 把新的索引复制到这些副本
  - 启动一些新的前端指向已更新的索引，服务与一些负载

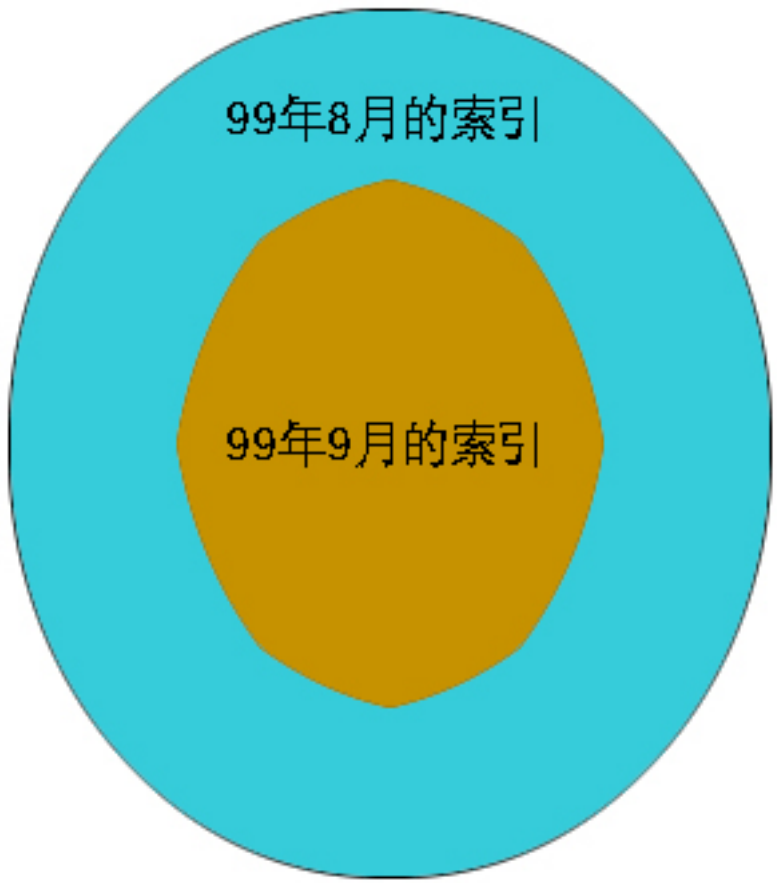
索引更新(1998-1999)

- 索引服务器硬盘：
  - 磁盘外圈的带宽比较高

索引更新(1998-1999)

- 索引服务器硬盘：
  - 磁盘外圈的带宽比较高

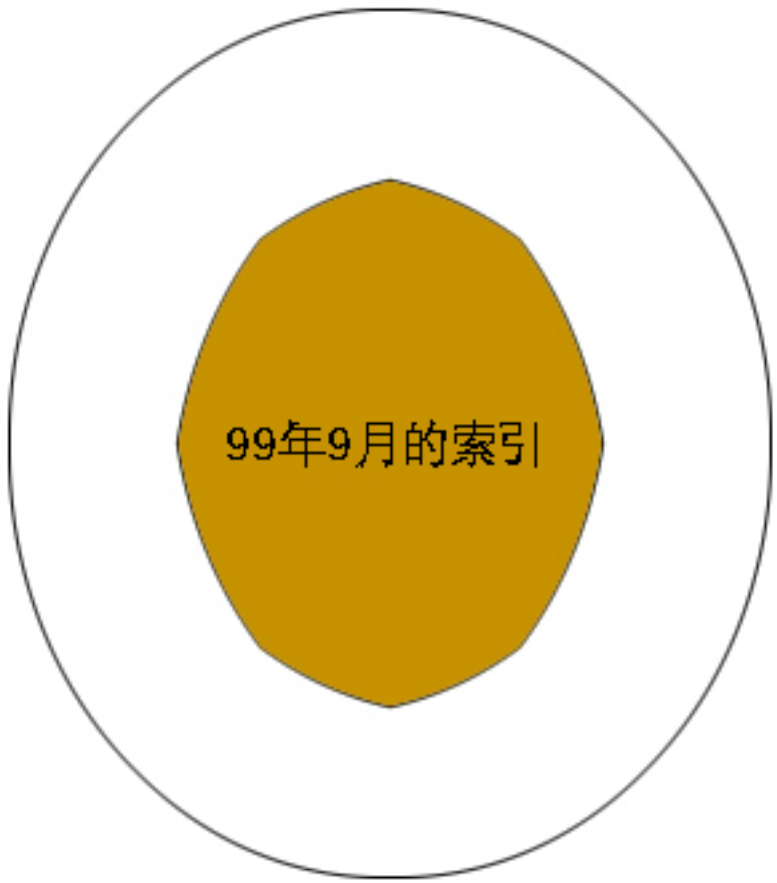
1. 把新的索引复制到磁盘内圈（老的索引继续工作）
2. 重新启动到使用新的索引



索引更新(1998-1999)

- 索引服务器硬盘：
  - 磁盘外圈的带宽比较高

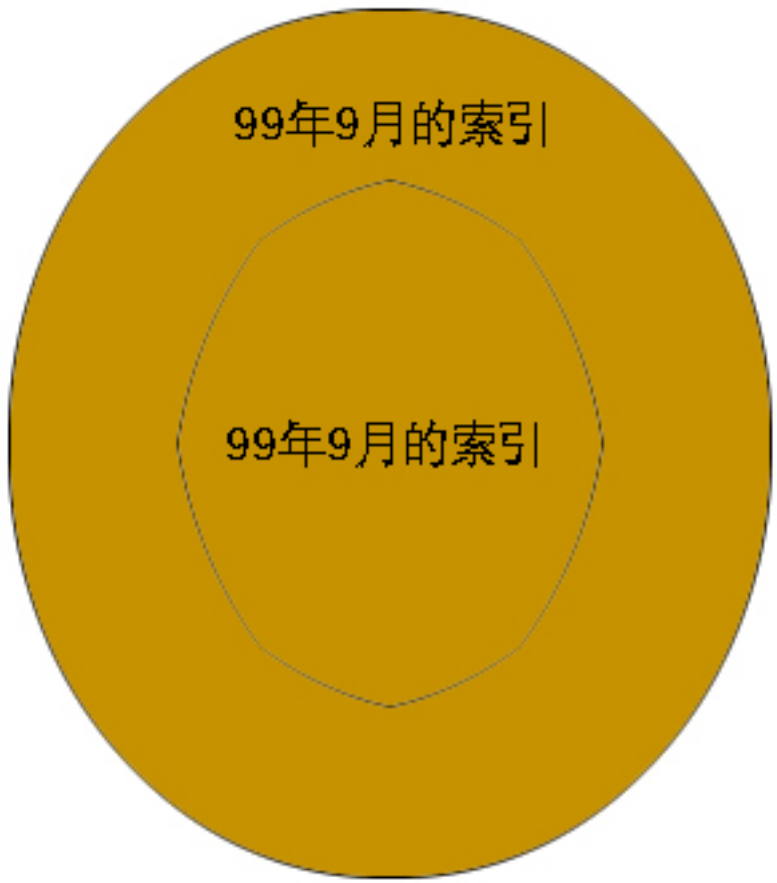
1. 把新的索引复制到磁盘内圈（老的索引继续工作）
2. 重新启动到使用新的索引
3. 擦掉老索引



索引更新(1998-1999)

- 索引服务器硬盘：
  - 磁盘外圈的带宽比较高

1. 把新的索引复制到磁盘内圈（老的索引继续工作）
2. 重新启动到使用新的索引
3. 擦掉老索引
4. 把新的索引复制到硬盘外圈



索引更新(1998-1999)

- 索引服务器硬盘：
  - 磁盘外圈的带宽比较高

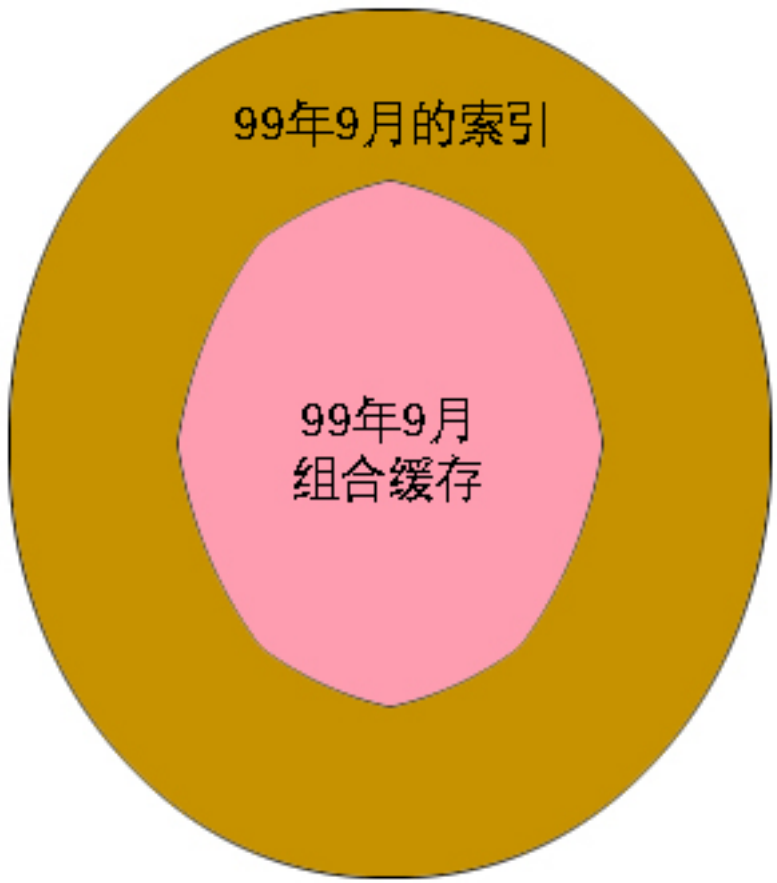
1. 把新的索引复制到磁盘内圈（老的索引继续工作）
2. 重新启动到使用新的索引
3. 擦掉老索引
4. 把新的索引复制到硬盘外圈
5. 擦掉新索引的第一个副本

索引更新(1998-1999)

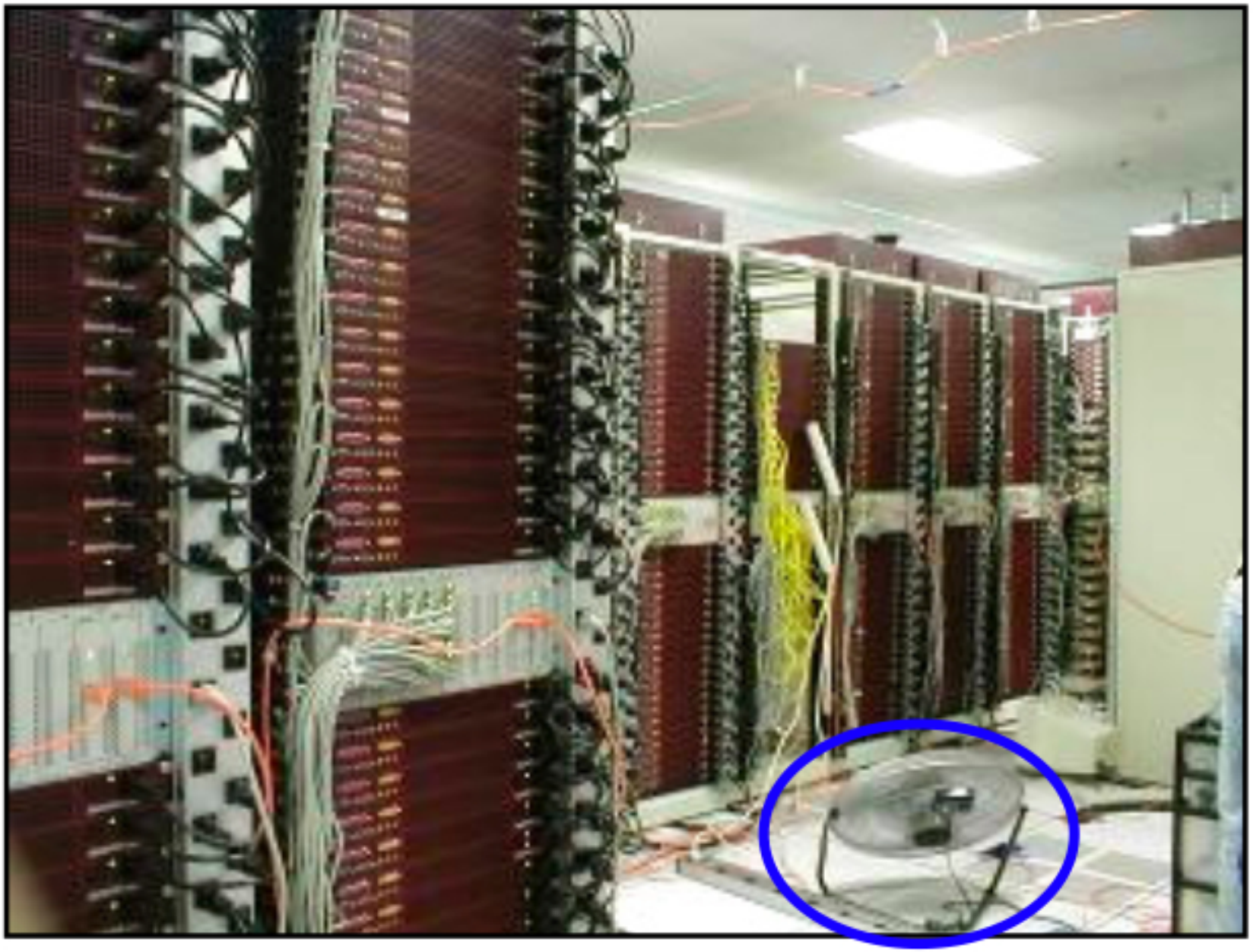
- 索引服务器硬盘：
  - 磁盘外圈的带宽比较高

1. 把新的索引复制到磁盘内圈（老的索引继续工作）
2. 重新启动到使用新的索引
3. 擦掉老索引
4. 把新的索引复制到硬盘外圈
5. 擦掉新索引的第一个副本
6. 内圈现在就空余出来可以构建几种可以提高性能的数据结构了

组合缓存：常用组合词Posting lists的交集(例如，“new”和“york” 或者“barcelona”和“restaurants”)



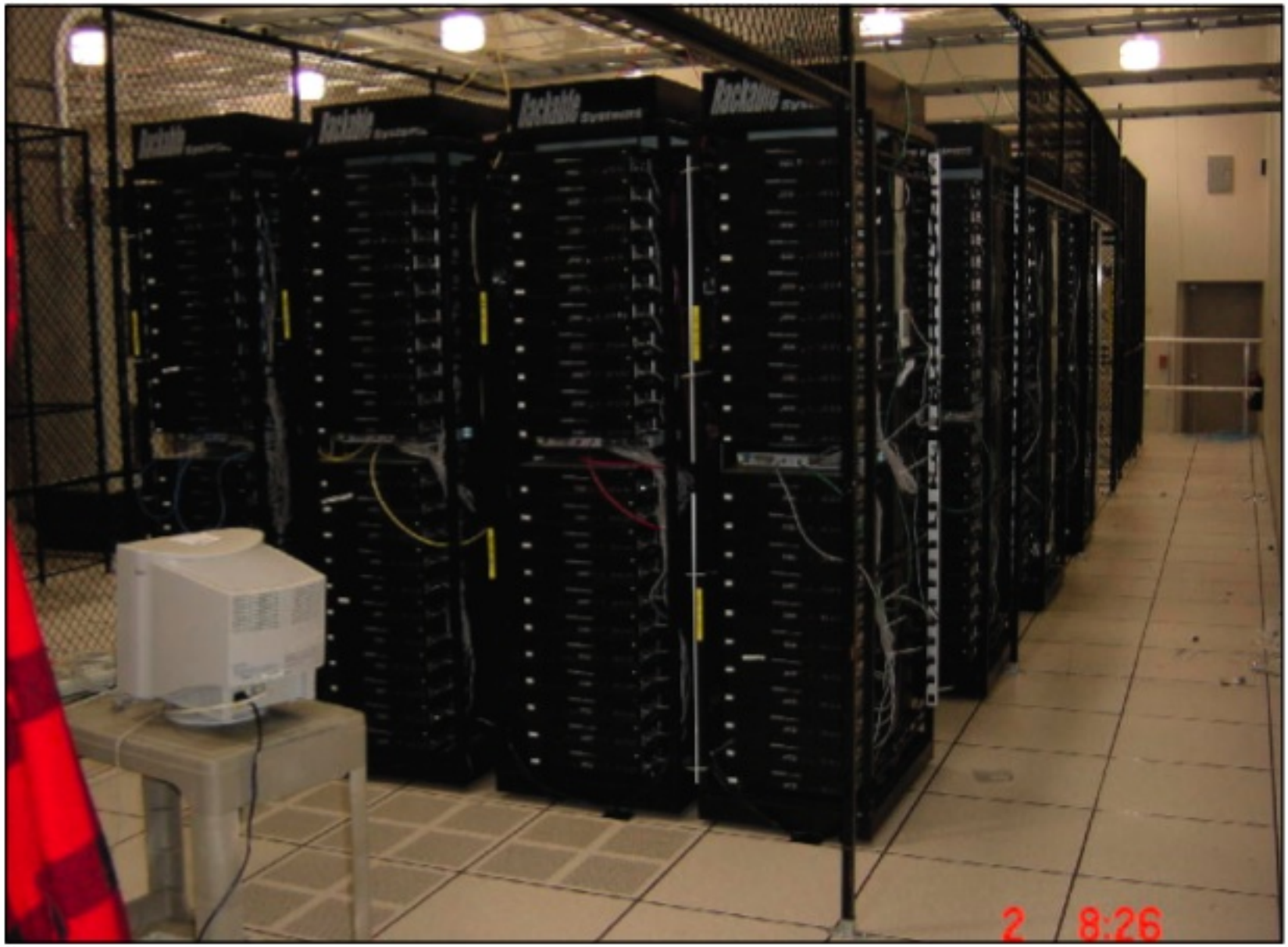
Google数据中心(2000)



译注：我估计蓝圈圈住的是一台风扇

Google 新数据中心2001

Google数据中心(三天后)

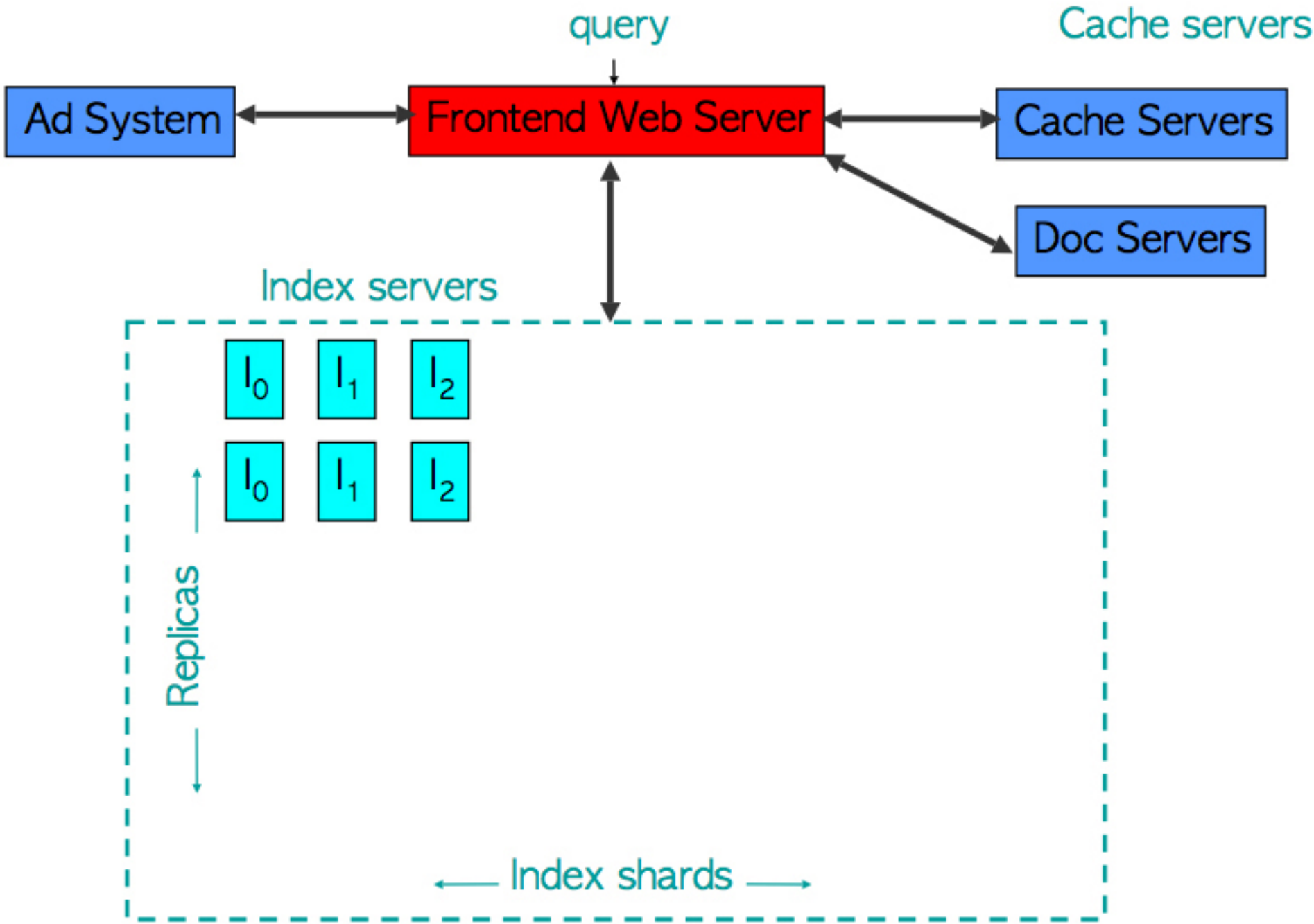


译注：这两张照片揭示了Google数据中心2001年上线的日期，呵呵，你看出来了么？

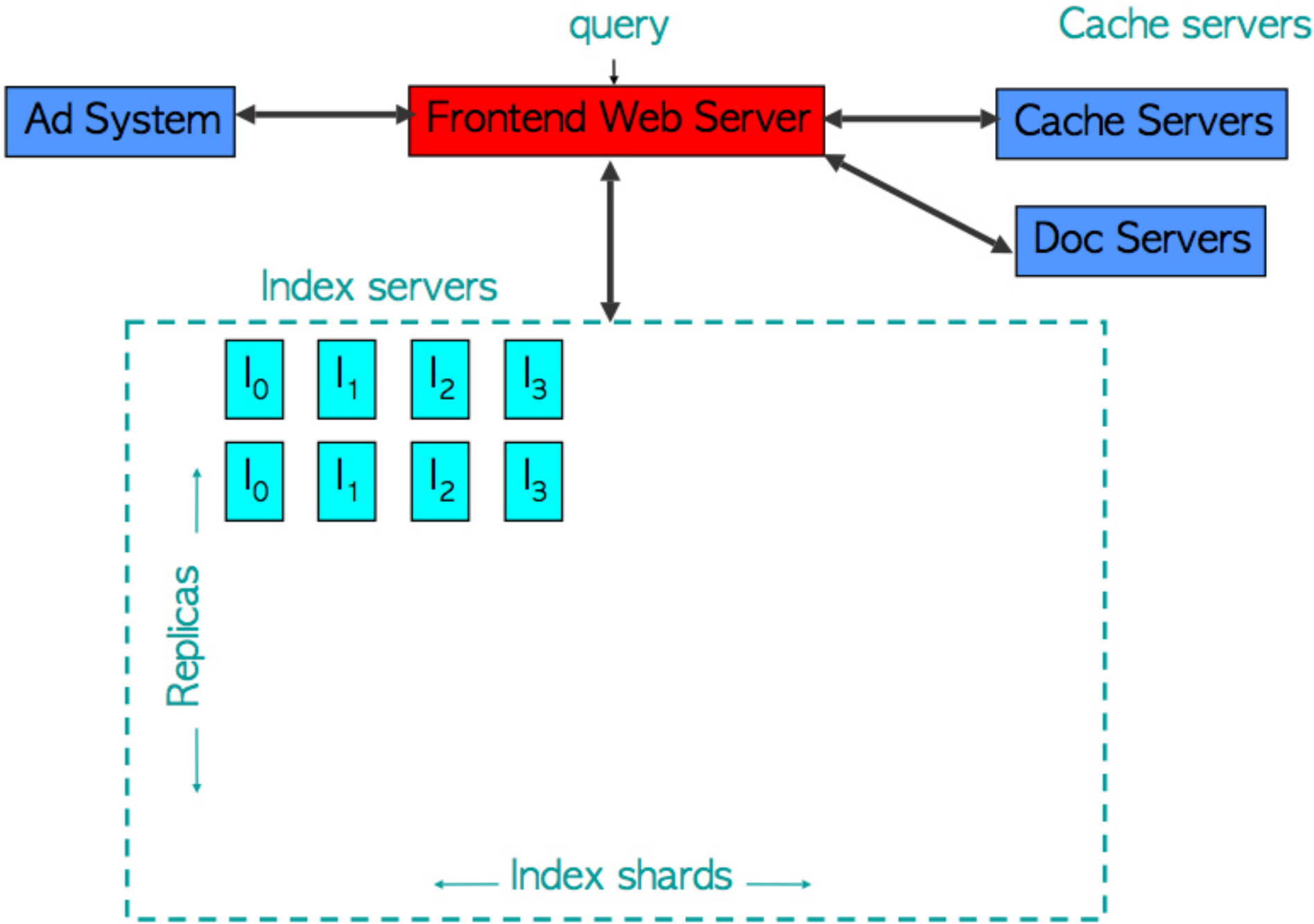
增长的索引尺寸和查询能力

- 索引尺寸在99年，2000年，2001年…间的巨大增长
  - 从约5000万页面到接近10亿页面
- 与此同时巨大的流量增长
  - 1999年到200年，每月增长约20%
  - …加上新的合作伙伴（例如，Yahoo在2000年7月几乎一夜间流量增长了一倍）
- 索引服务器的性能是首要问题
  - 持续部署更多的机器，但是…
  - 每月需要约10%到30%的基于软件的性能改善

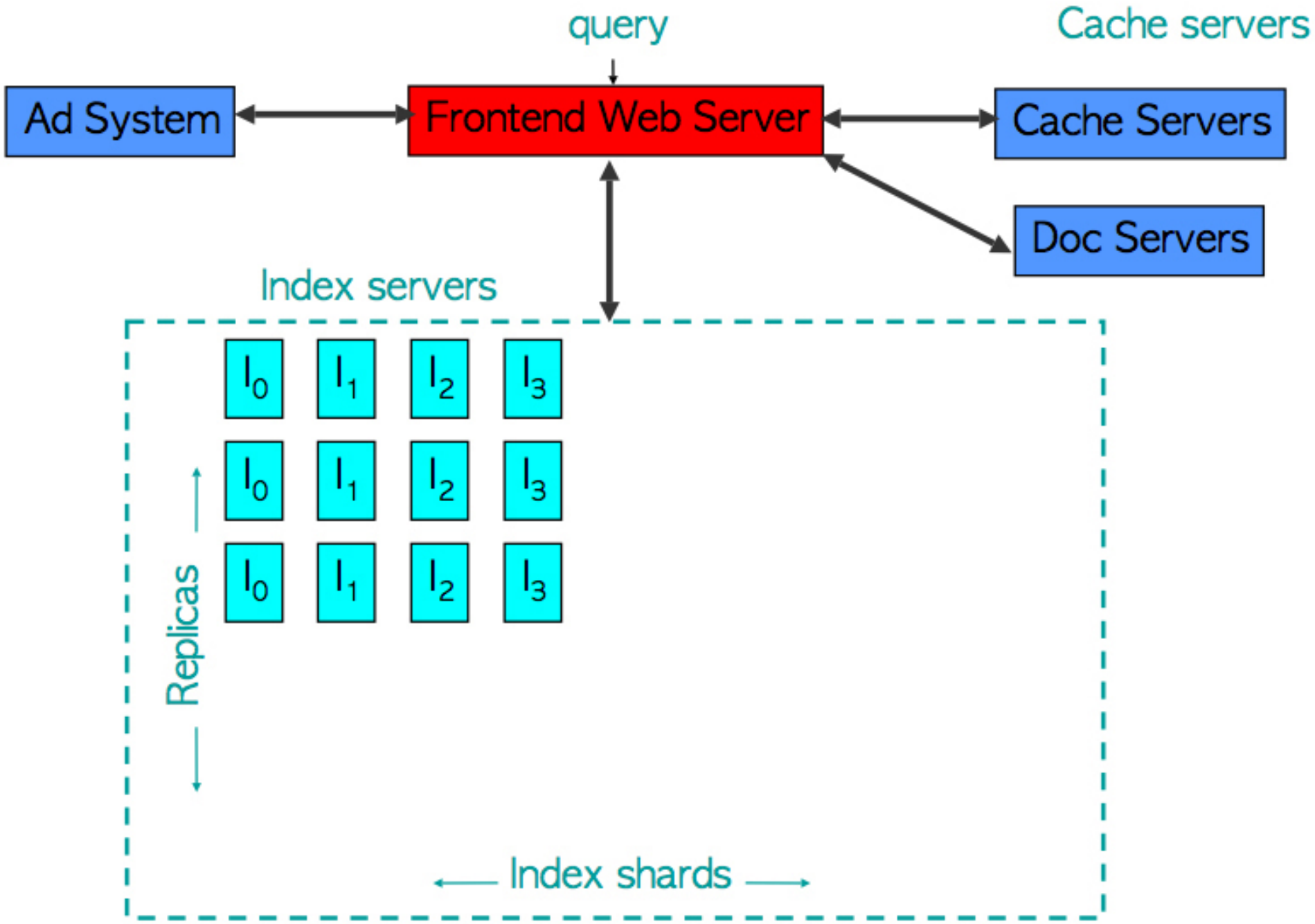
处理增长


query
Cache servers
Ad System
Frontend Web Server
Cache Servers
Doc Servers
Index servers
I_0 I_1 I_2
I_0 I_1 I_2
Replicas
Index shards

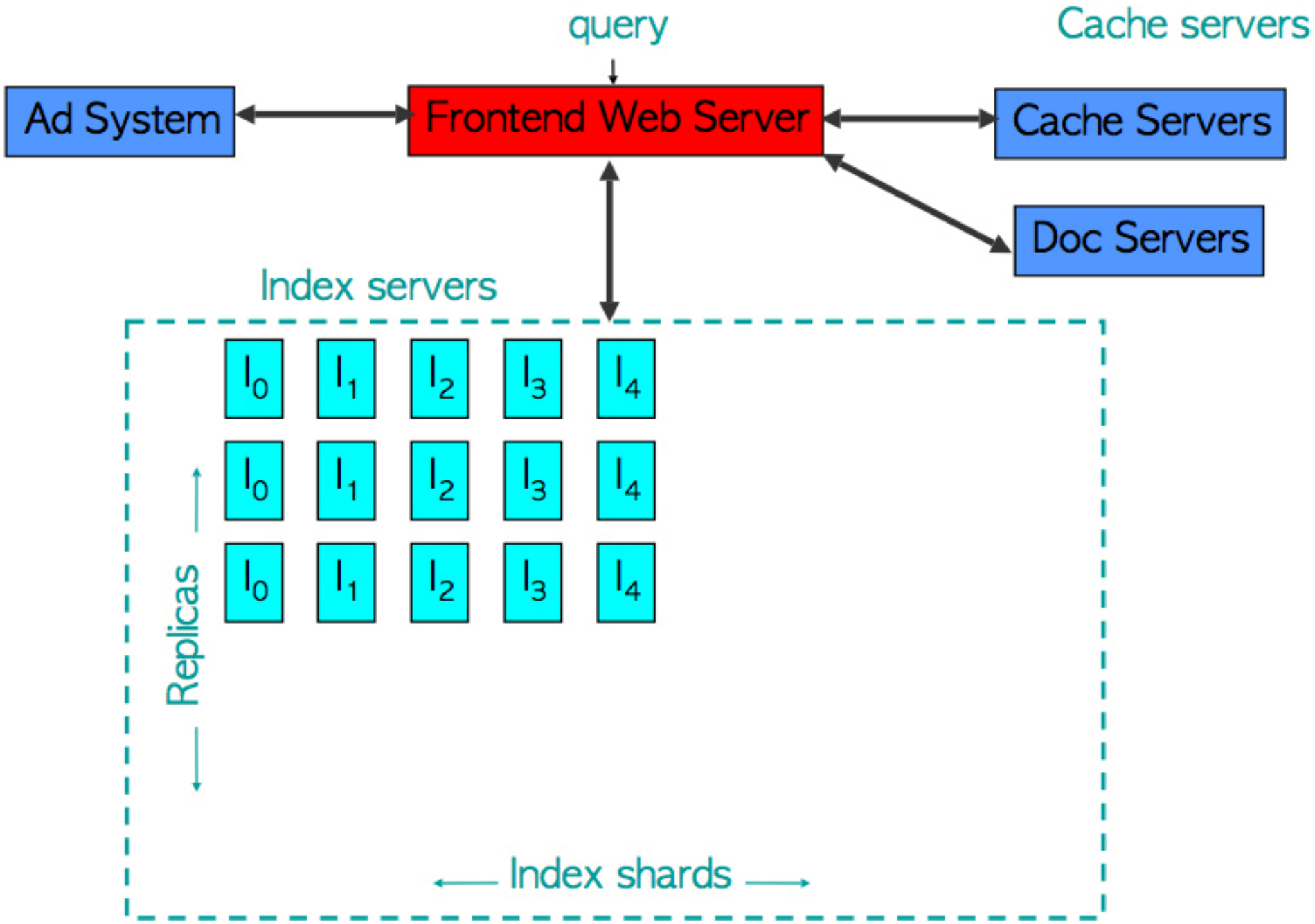
处理增长


query
Cache servers
Ad System
Frontend Web Server
Cache Servers
Doc Servers
Index servers
I_0 I_1 I_2 I_3
I_0 I_1 I_2 I_3
Replicas
Index shards

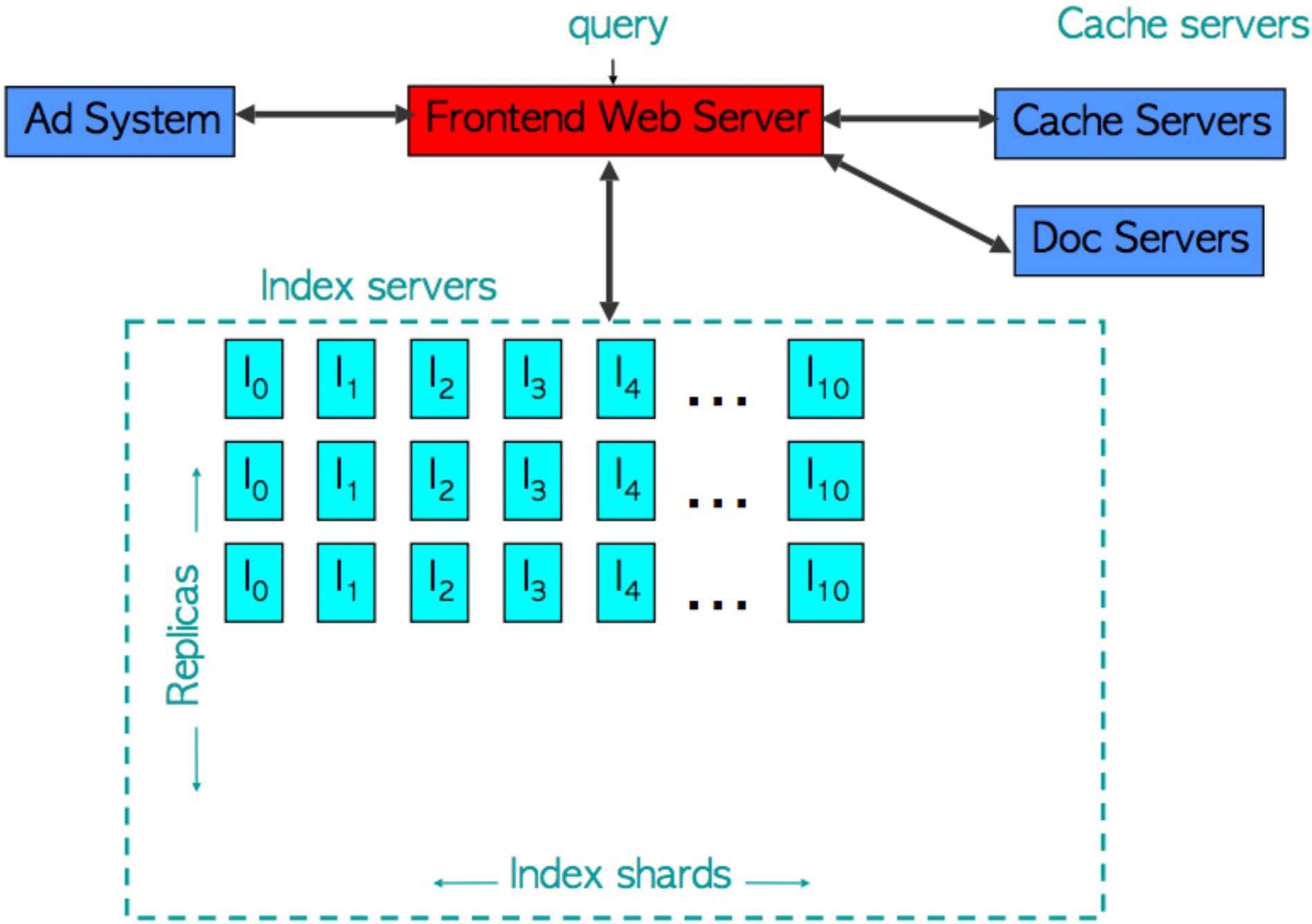
处理增长


query
Cache servers
Ad System
Frontend Web Server
Cache Servers
Doc Servers
Index servers
I_0 I_1 I_2 I_3
I_0 I_1 I_2 I_3
I_0 I_1 I_2 I_3
Replicas
Index shards

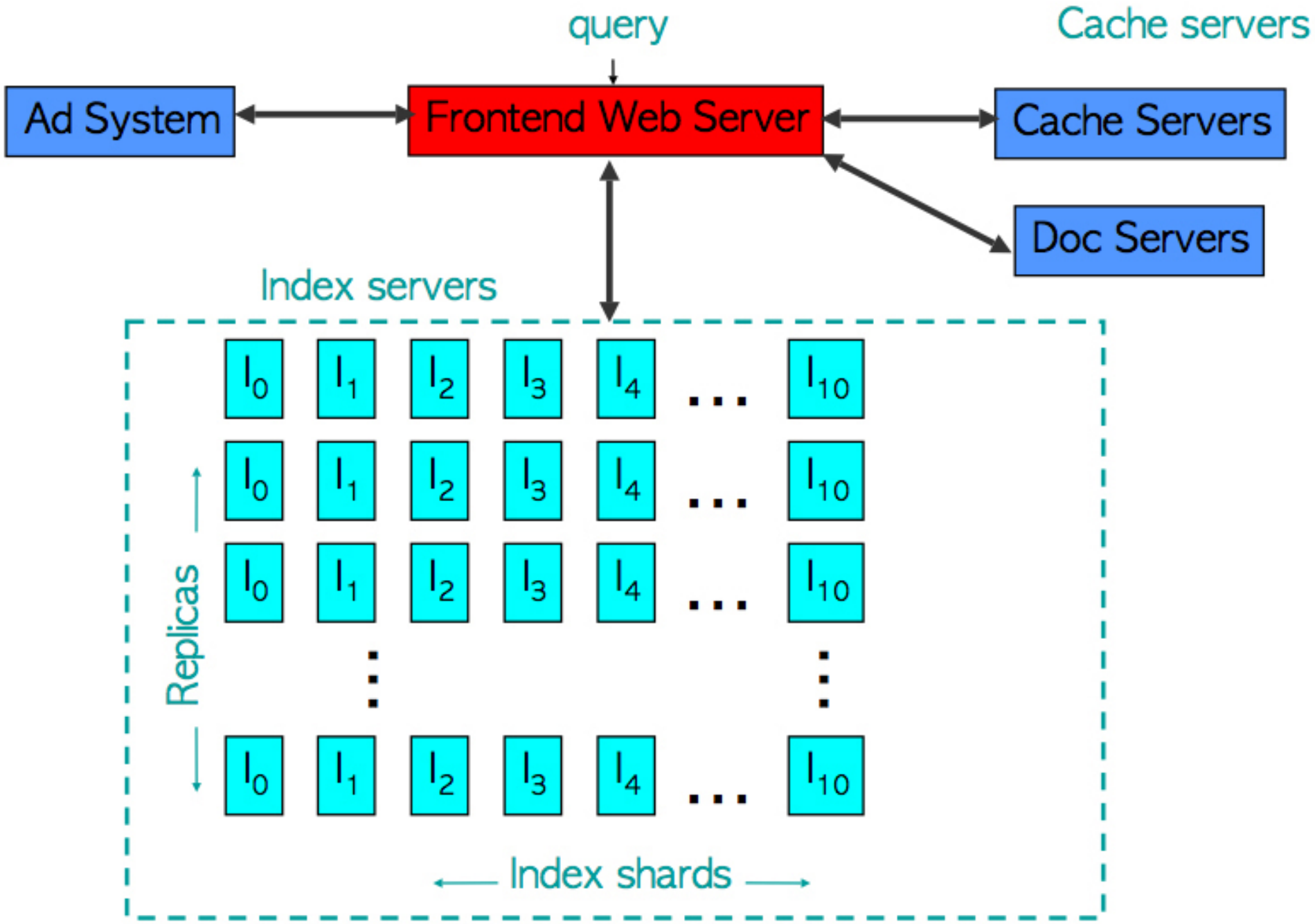
处理增长


query
Cache servers
Ad System
Frontend Web Server
Cache Servers
Doc Servers
Index servers
I_0 I_1 I_2 I_3 I_4
I_0 I_1 I_2 I_3 I_4
I_0 I_1 I_2 I_3 I_4
Replicas
Index shards

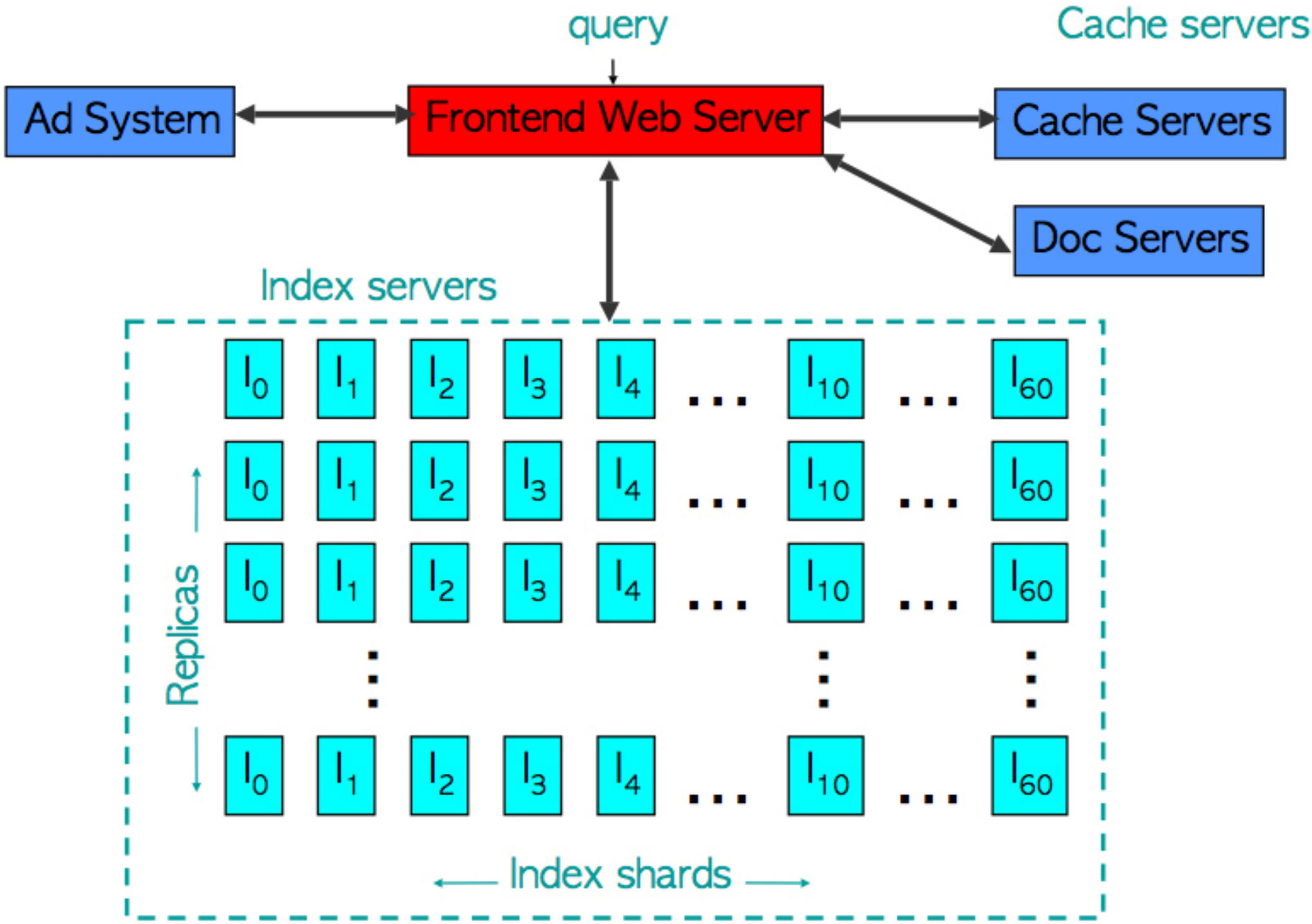
处理增长


query
Cache servers
Ad System
Frontend Web Server
Cache Servers
Doc Servers
Index servers
I_0 I_1 I_2 I_3 I_4 ... I_{10}
I_0 I_1 I_2 I_3 I_4 ... I_{10}
I_0 I_1 I_2 I_3 I_4 ... I_{10}
Replicas
Index shards

处理增长


query
Cache servers
Ad System
Frontend Web Server
Cache Servers
Doc Servers
Index servers
I_0 I_1 I_2 I_3 I_4 ... I_{10}
Replicas
Index shards

处理增长


query
Cache servers
Ad System
Frontend Web Server
Cache Servers
Doc Servers
Index servers
I_0 I_1 I_2 I_3 I_4 ... I_{10} ... I_{60}
Replicas
Index shards

推论

- 分片服务器的响应时间受到下列因素的影响：
  - 需要完成的磁盘寻道数
  - 从磁盘读取的数据量
- 重大性能改进可能来自于：
  - 更好的磁盘规划
  - 改进索引编码格式

1997-1999的索引编码格式

- 原始编码('97)很简单：

  - hit:位置加上属性(字体尺寸，标题，等等)
  - 对大的posting lists加上跳表
- 简单，字节对齐格式
  - 解码开销小，但是不够紧凑
  - ...需要大量的磁盘带宽

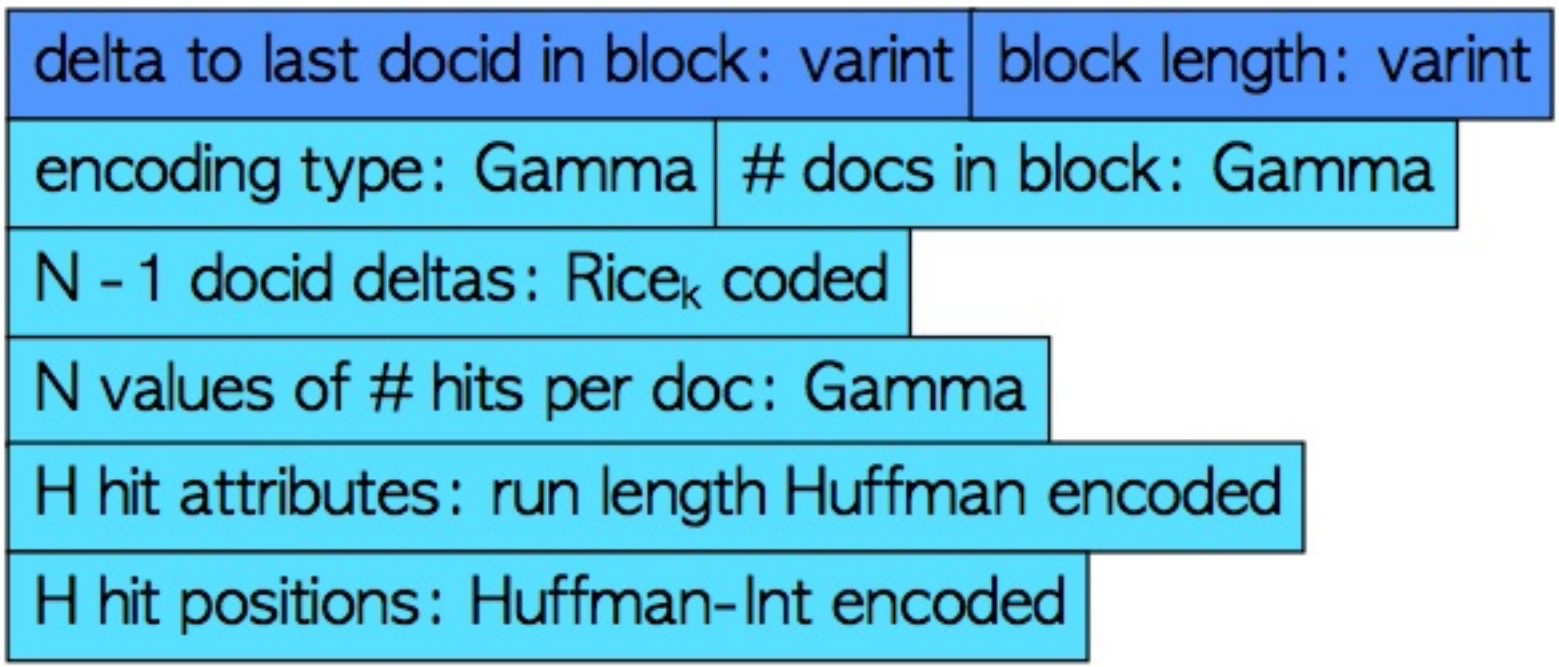
编码技术

- 位级编码：
  - Unary(一进制编码)：N个‘1’后面跟着个‘0’
  - Gamma: log2(N)以一进制编码，后跟余数的二进制表示
  - Ricek:floor(N/2^k)以一进制编码，N mode 2^k在k位中
    - 这个编码是当底为2的幂时，Golomb编码的一个特例
  - Huffman-Int:类似Gamma，只是log2(N)以Huffman编码来编码，而不是用Unary来编码
- 字节对齐编码：
  - 变体：每个字节7位，加一个校验位
    - 0-127:一个字节，128-4095:两个字节

基于块的索引格式

- 基于块的变长格式减少空间占用和Cpu开销

块格式（N个文档，H个hit）

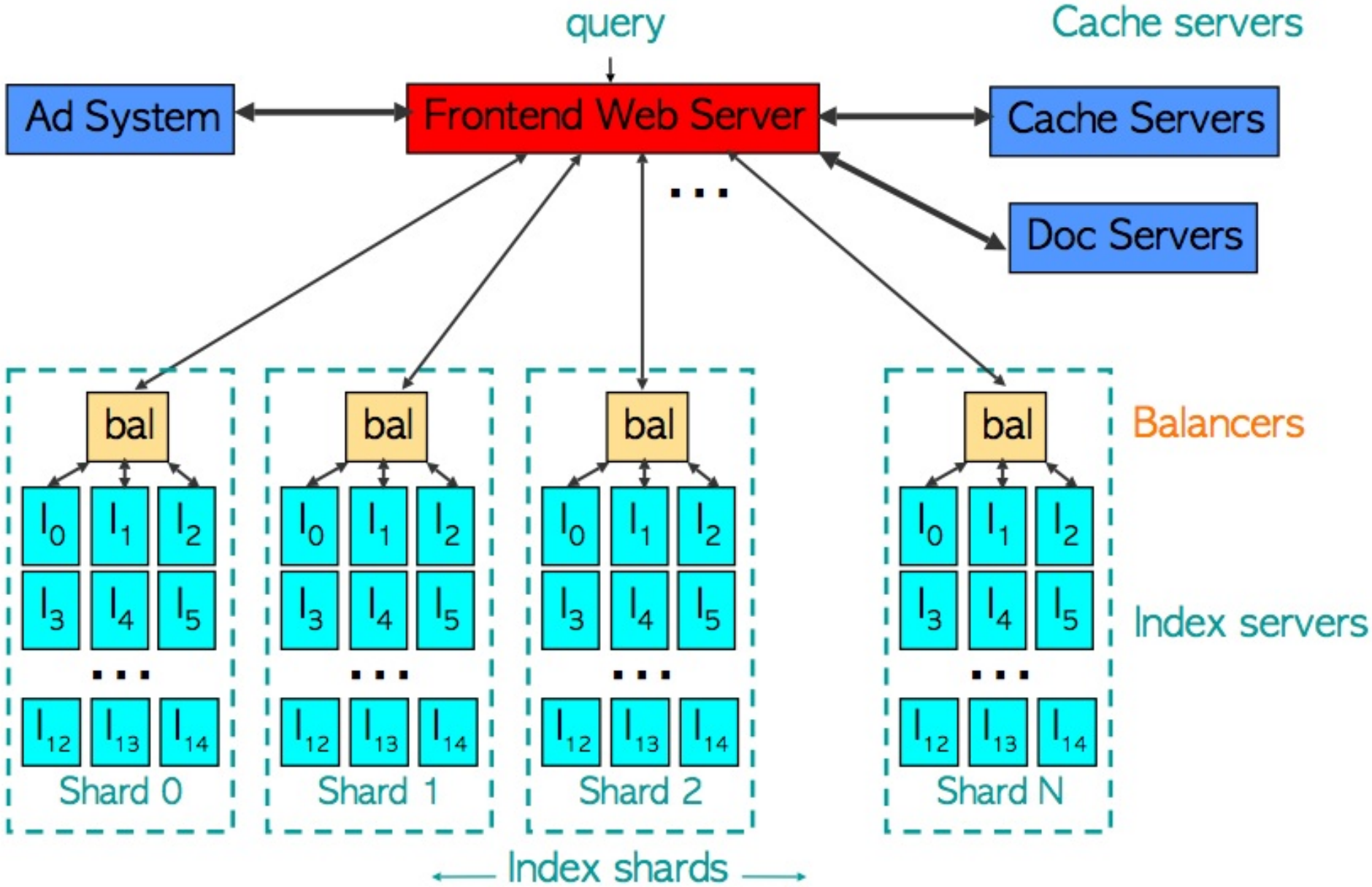


- 降低索引尺寸约30%，解码更快

更深入的分片的推论

- 索引尺寸增大时必须加入新的分片来保持快速响应
- …查询开销随着分片数的增加而增加
  - 一般 >= 1个磁盘寻道/分片/查询词
  - 甚至是对于那些冷门的词
- 随着副本数量增加，可用内存总量也增加了
  - 终于，有足够的内存保存整个索引的副本在内存中
    - 从本质上改变了许多设计参数

2001年初: 内存内索引



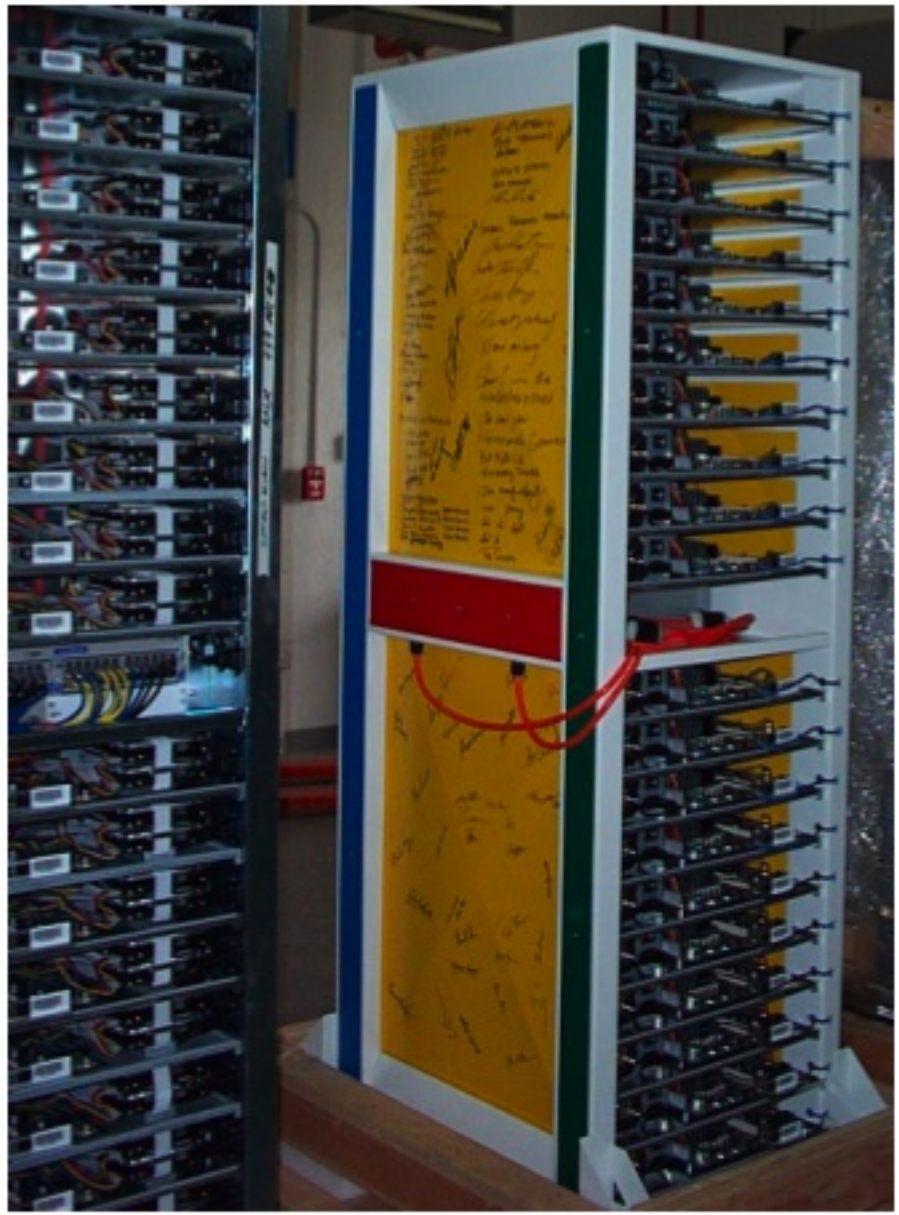
内存索引系统

- 许多优点：
  - 大幅增加吞吐量
  - 大大降低延迟
    - 尤其是尾部：以前需要数GB磁盘I/O的开销较大查询变得更快了（例如：“circle of life”）
- 一些问题：
  - 变化：需要触及上千台服务器，而不是十几个
    - 例如，随机化的定时任务曾给我们带来很多麻烦
  - 可用性：每个文档的索引数据只有1个或少数几个副本
    - 查询的死亡可以一次性杀死所有的后端：非常不好
    - 当有机器失效的时候索引数据的可用性（尤其是重要文档）：复制重要文档

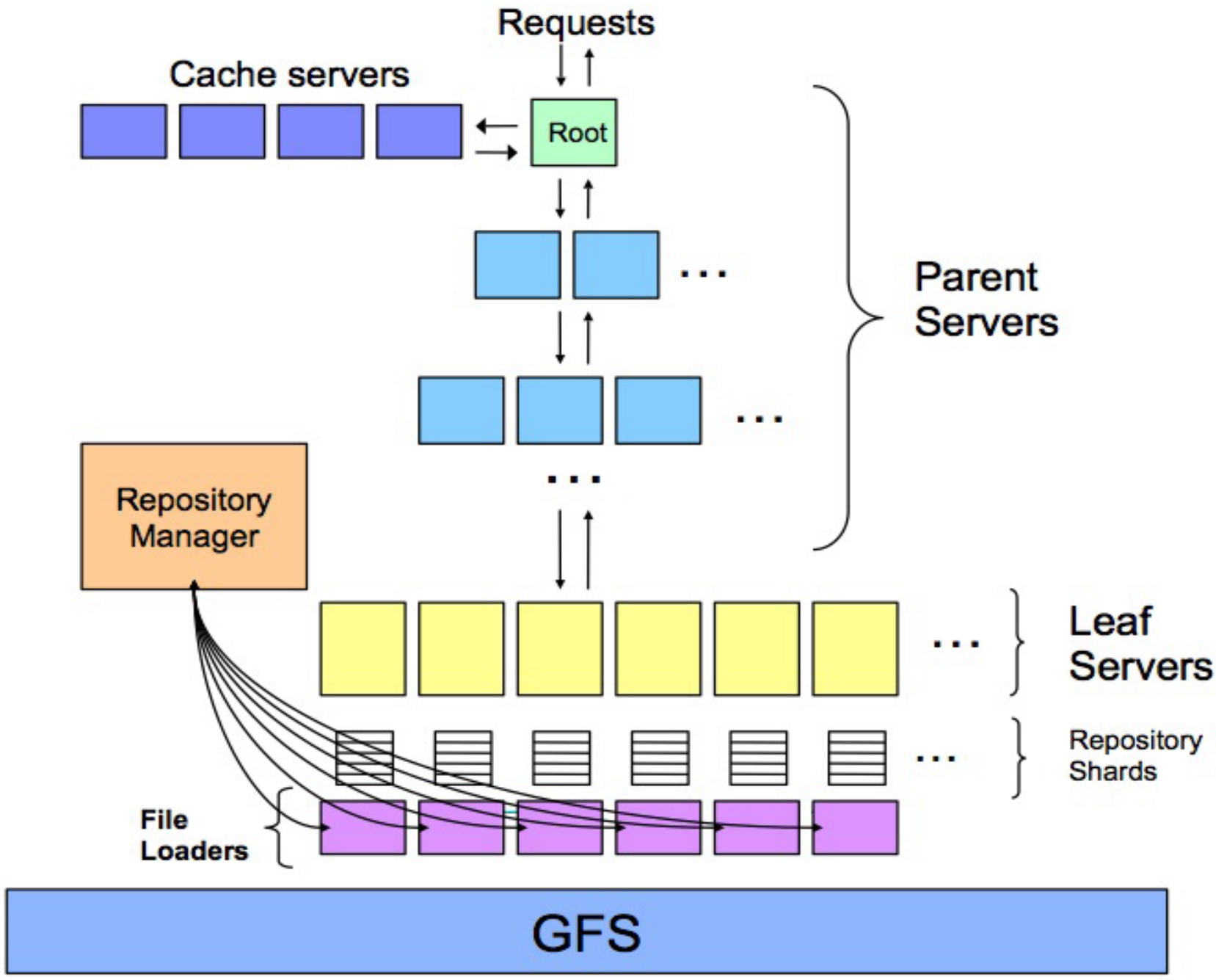
大规模计算

现在的机器

- 机架自行设计
- PC级主板
- 低端存储和网络硬件
- Linux
- +自行设计的软件



服务设计，2004版本

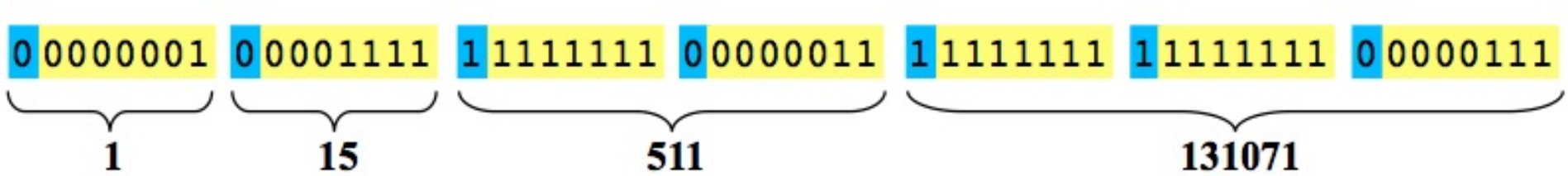

Requests
Cache servers
Root
Parent Servers
Repository Manager
Leaf Servers
Repository Shards
File Loaders
GFS

新的索引格式

- 基于块的索引格式采用两级索引方案
  - 每个hit编码为(文档编号，词的位置)形式的对
  - 文档编号的偏移用Rice编码来编码
  - 压缩率很高(原本为基于硬盘的索引设计)，但是解码较慢，对CPU性能很敏感
- 新格式:新的平面位置空间
  - 旁边的数据结构保持文档的边界
  - Posting list仅是用偏移编码的位置
  - 必须紧凑（不能每次出现都使用32位值）
  - …

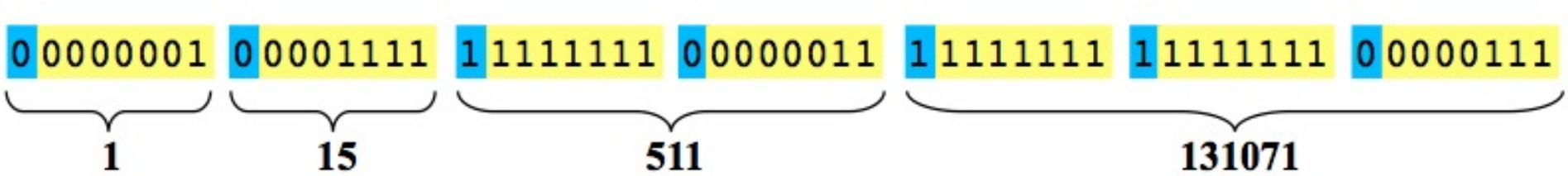
字节对齐变长编码

- 变长编码：
  - 每个字节7位，一个连续标志位
  - 连续标志位：解码需要大量分支/移位/掩码操作

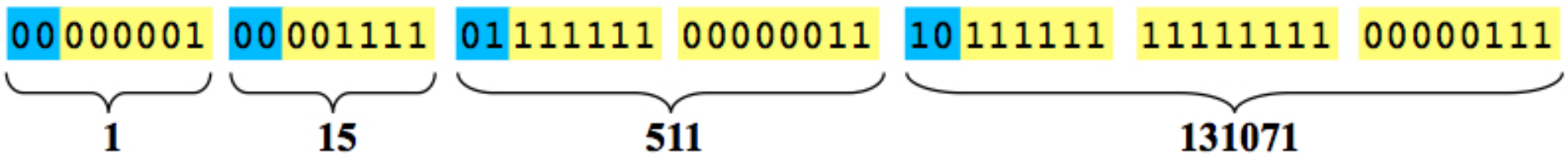


字节对齐变长编码

- 变长编码：
  - 每个字节7位，一个连续标志位
  - 连续标志位：解码需要大量分支/移位/掩码操作

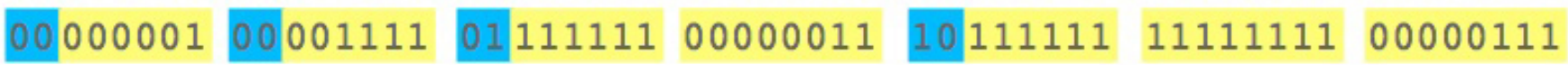


- 思想：用低二位编码字节长度
  - 更好：更少的分支，移位和掩码操作
  - 连续标志位：值只能限制在30位，解码仍旧需要移位



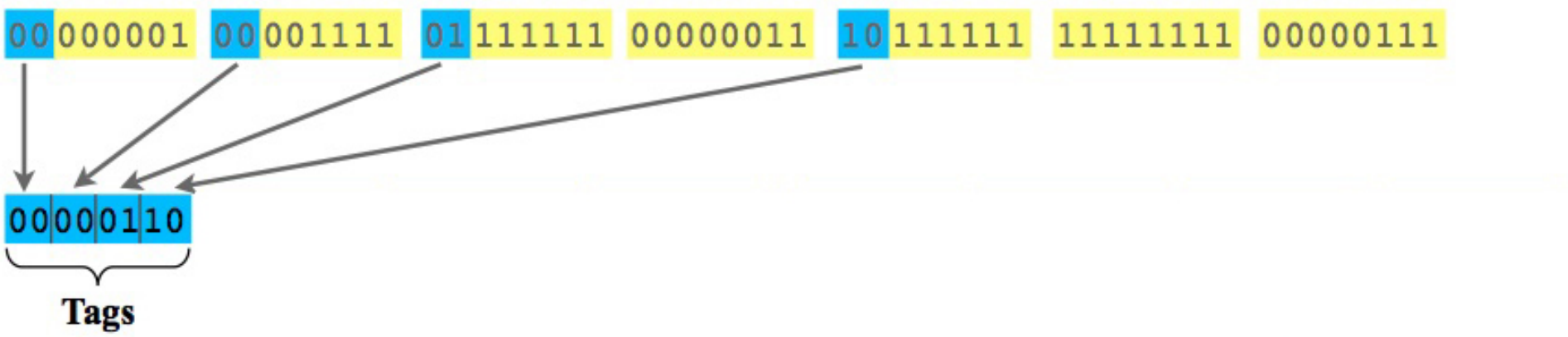
组变长编码

- 思想：用5-17字节来编码一组4个值
  - 把4个2位长度位抽出来，合成一个单字节前缀



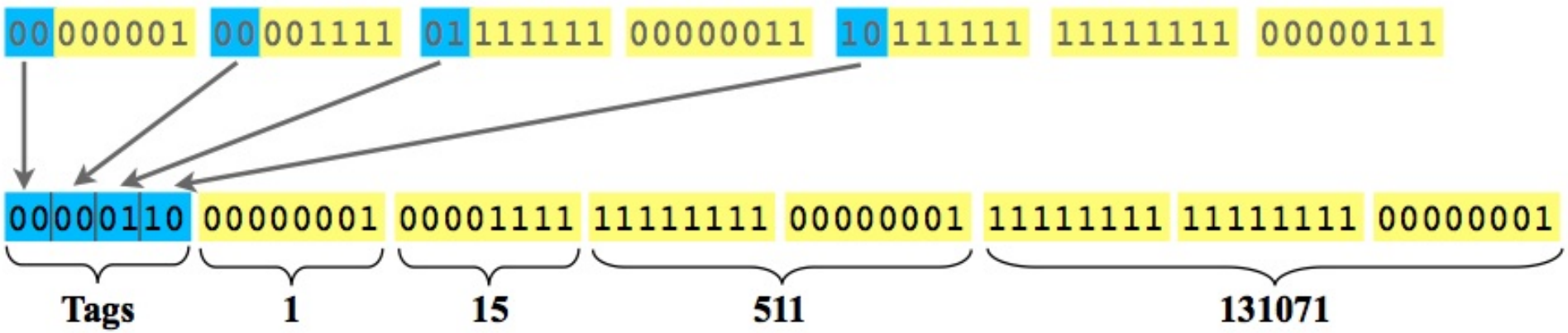
组变长编码

- 思想：用5-17字节来编码一组4个值
  - 把4个2位长度位抽出来，合成一个单字节前缀



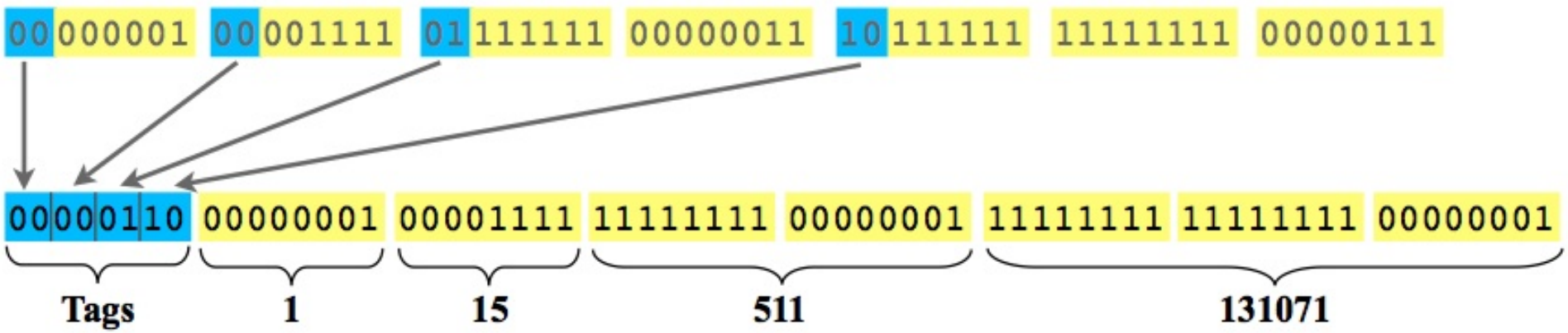
组变长编码

- 思想：用5-17字节来编码一组4个值
  - 把4个2位长度位抽出来，合成一个单字节前缀



组变长编码

- 思想：用5-17字节来编码一组4个值
  - 把4个2位长度位抽出来，合成一个单字节前缀

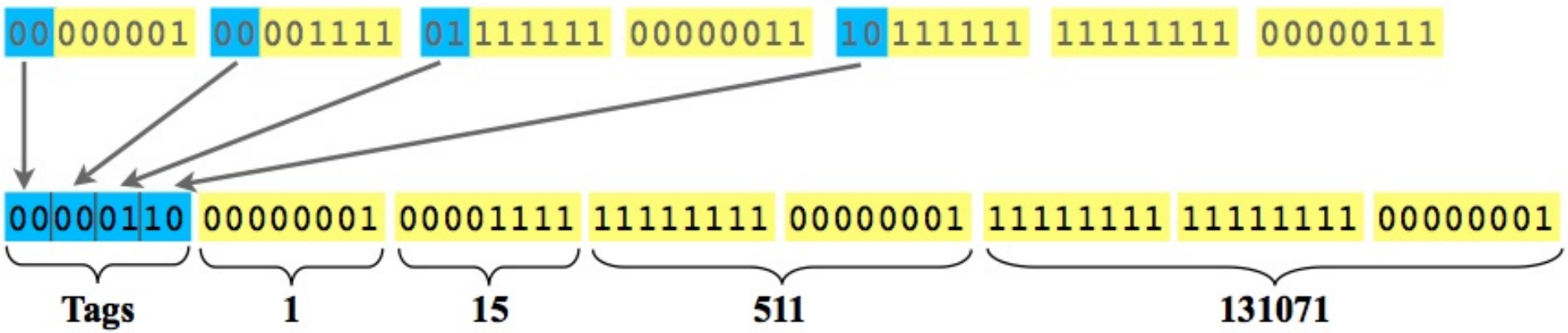


- 编码：读取前缀字节，利用它的值查找256节点的表：

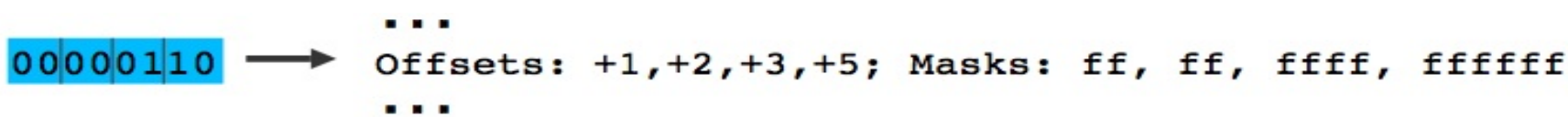
```
...
00000110 ──→ Offsets: +1,+2,+3,+5; Masks: ff, ff, ffff, ffffff
...
```

组变长编码

- 思想：用5-17字节来编码一组4个值
  - 把4个2位长度位抽出来，合成一个单字节前缀

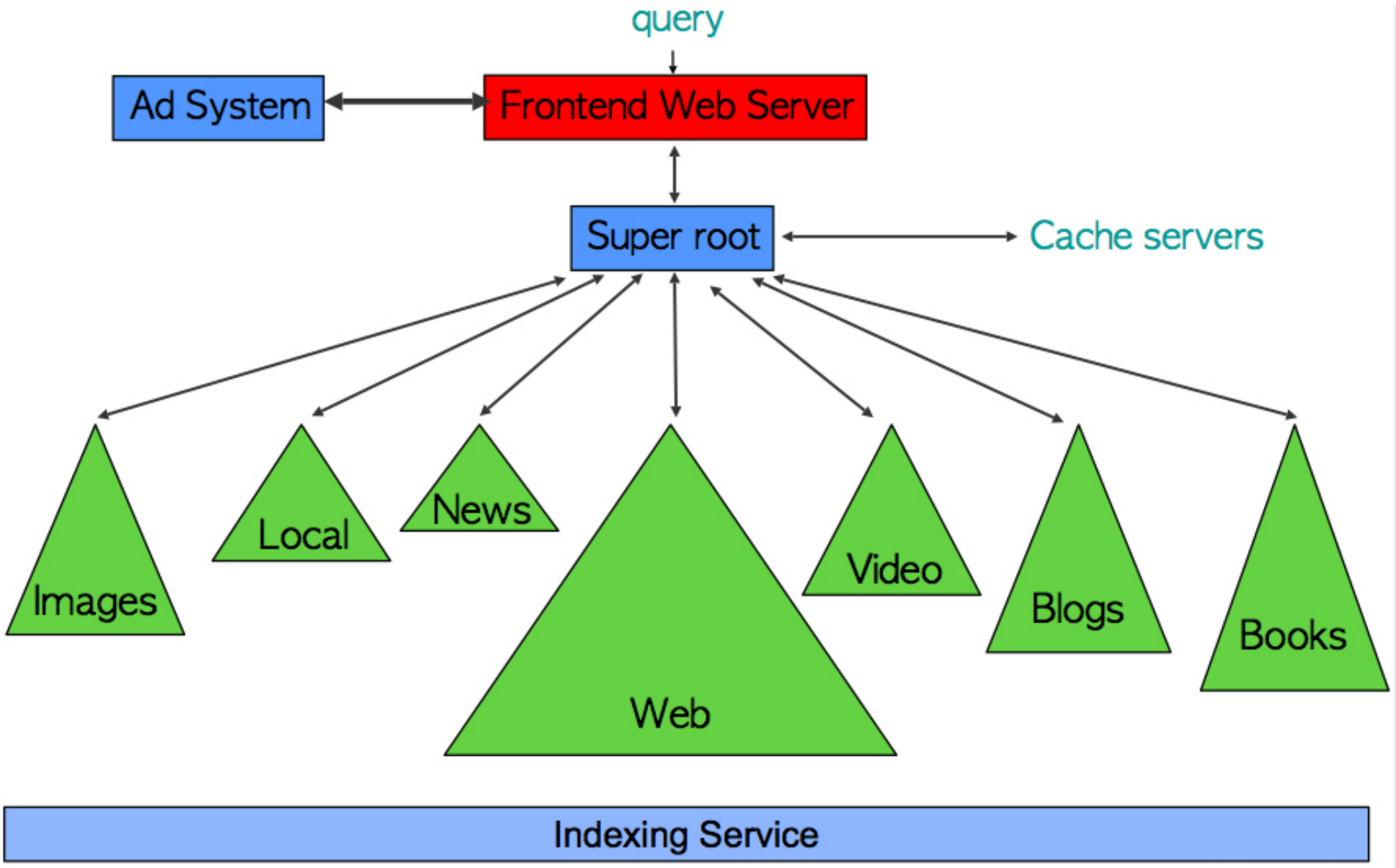


- 编码：读取前缀字节，利用它的值查找256节点的表：



- 比其他方案更快
  - 每字节7位变长：解码速度 约每秒1.8亿数字
  - 变长30位 2位长度位：解码速度 约每秒2.4亿数字
  - 组变长编码：解码速度 约每秒4亿数字

2007:全能搜索


query
Ad System
Frontend Web Server
Super root
Cache servers
Images
Local
News
Web
Video
Blogs
Books
Indexing Service

索引它？一分钟足矣！

- 低延迟爬取和索引非常困难
  - 探索式爬取：什么页面应该被爬取？
  - 爬取系统：需要快速爬取页面
  - 索引系统：依赖于全局数据
    - PageRank，指向页面的链接锚文本，等等
    - 必须拥有这些全局属性的在线近似值
  - 服务系统：在服务请求的同时，时刻准备接受更新
    - 与批量更新服务系统完全不同的

信息检索系统的弹性和实验

- 易于实验非常重要
  - 快速切换 => 更多实验 => 更多改进
- 有些实验是很容易的
  - 例如，对现有数据赋不同的权重
- 其他的难以实施：需要的数据不存在于生产索引中
  - 必须能容易的生成和组合新数据并用于实验

搜索系统的架构

- 架构的几个关键点
  - GFS: 大规模分布文件系统
  - MapReduce: 便于编写和运行大规模作业
    - 更快地生成生产索引
    - 更快的执行特定的实验
    - …
  - BigTable: 半结构化存储系统
    - 在线，任何时候可以高效地访问每个文档的信息
    - 多个进程可以异步地更新文档的信息
    - 更新文档的耗时从小时到分钟

GFS论文：英文 中文

MapReduce论文： 英文 中文

BigTable论文： 英文 中文

实验循环，第一部分

- 从新的评分思想开始
- 必须要容易实验，快速的进行实验：
  - 利用MapReduce，BigTable这些工具生成数据
  - 起初，离线实验运行和实验结果
    - …在各种人工评分的查询集
    - …在随机查询中，观察与现有评分的差别
  - 对于这类原型不考虑延迟和吞吐量
- …在实验结果上反复迭代

实验循环，第二部分

- 一旦离线实验有前途，就开始在线实现
  - 在用户流量中的很小一部分进行实验
  - 通常采用随机样本
    - 但是有时候采用特定种类的查询样本
      - 例如，英语查询，或者带有地名的查询，等等
- 对于这个部分，吞吐量不重要，延迟重要
  - 实验框架的操作延迟必须接近产品系统的延迟

实验看起来不错：然后呢？

- 实施！
- 性能优化/重新实现令其可以支持全部负载
  - 例如，预先计算好数据，而不是实时计算
  - 例如，如果近似“足够好”，而且便宜，那么就取近似值
- 发布流程要点：
  - 持续的在质量和开销之间权衡
  - 快速发布跟低延迟和整体稳定性是冲突的
    - 需要搜索质量小组和令搜索更快更稳定的小组间建立良好的工作关系

未来的方向和挑战

- 一些在有趣的方向上即将实现的想法

跨语言信息检索

- 把世界上所有文档翻译成各种语言
  - 实际上增大了索引的尺寸
  - 计算代价高昂
  - …但是一旦完成，有巨大的好处
- 挑战
  - 持续提高翻译质量
  - 大规模系统用来处理更大更复杂的语言模型
    - 翻译一句话 => 需要在数TB的模型进行100次查找

检索系统的访问控制信息

- 检索系统中混杂了私有，半私有，大量被分享，以及完全广开的文档
  - 例如，e-mail，在10个人中共享的文档，10万个用户用户组的信息，以及公开的web文档
- 挑战：构建高效处理非常巨大的访问控制信息的检索系统
  - 针对10个人共享的文档的最好解决方案一定不同于全世界都可以访问的文档
  - 文档的共享模型可能随时在变

高效信息检索系统的自动构建

- 当前使用几个信息检索系统
  - 例如，一个系统需要低于一秒的更新延迟，一个系统包含大量的文档但是只需每日更新，…
  - 相同的接口，但是为了效率采用了非常不同的实现方式
  - 工作良好，需要付出许多额外的de工作量，维护和扩展不同的系统
- 挑战：我们能否使用一个简单的参数化系统，基于不同的参数自动构建高效的信息检索系统

半结构化数据的信息抽取

- 在所有数据中，带有清晰语义信息标签的数据是非常少的
- 但是存在很多半结构化数据
  - 书，带表格的网页，以及窗体中的数据，…
- 挑战：提升从无结构或者半结构化数据源抽取信息的算法和技术
  - 噪声信息，很多冗余信息
  - 试图关联/整合/聚集不同数据源的信息

结论

- 设计和构建大规模信息检索系统是一个挑战，乐趣无穷
- 新问题需要持续演化
- 为许多用户的利益努力
- 新的检索技术经常需要新的系统
- 谢谢你的关注

感谢和提问

- 延展阅读：
  - Ghemawat, Gobioff, & Leung. Google File System, SOSP 2003.
  - Barroso, Dean, & Hölzle. Web Search for a Planet: The Google Cluster Architecture, IEEE Micro, 2003.
  - Dean & Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004.
  - Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, & Gruber. Bigtable: A Distributed Storage System for Structured Data, OSDI 2006.
  - Brants, Popat, Xu, Och, & Dean. Large Language Models in Machine Translation, EM
- 更多的参看
  http://labs.google.com/papers.html