

3 网络信息主动获取与处理

2019年4月24日 16:19

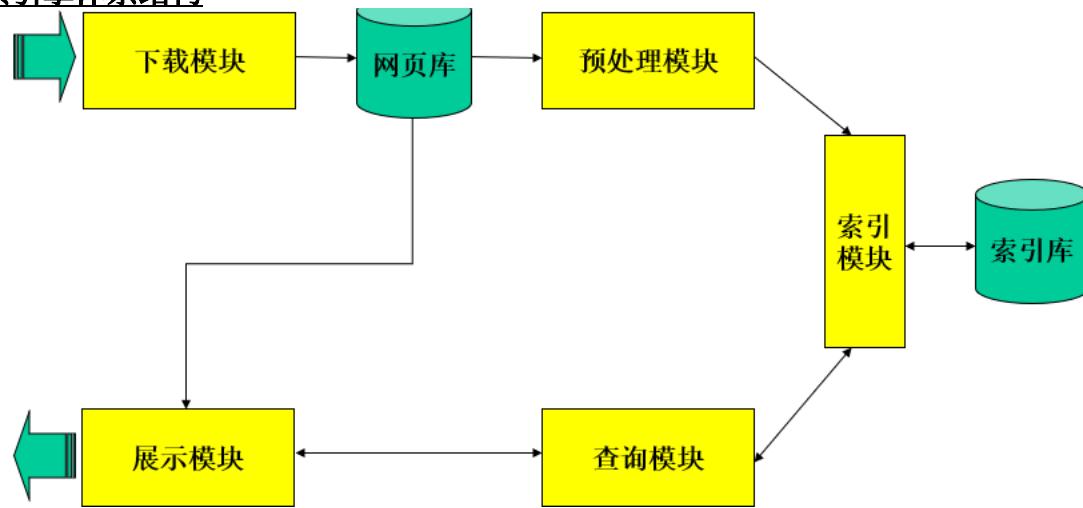
1 搜索引擎体系结构

2 网络爬虫技术

3 PageRank技术

4 索引技术

1 搜索引擎体系结构

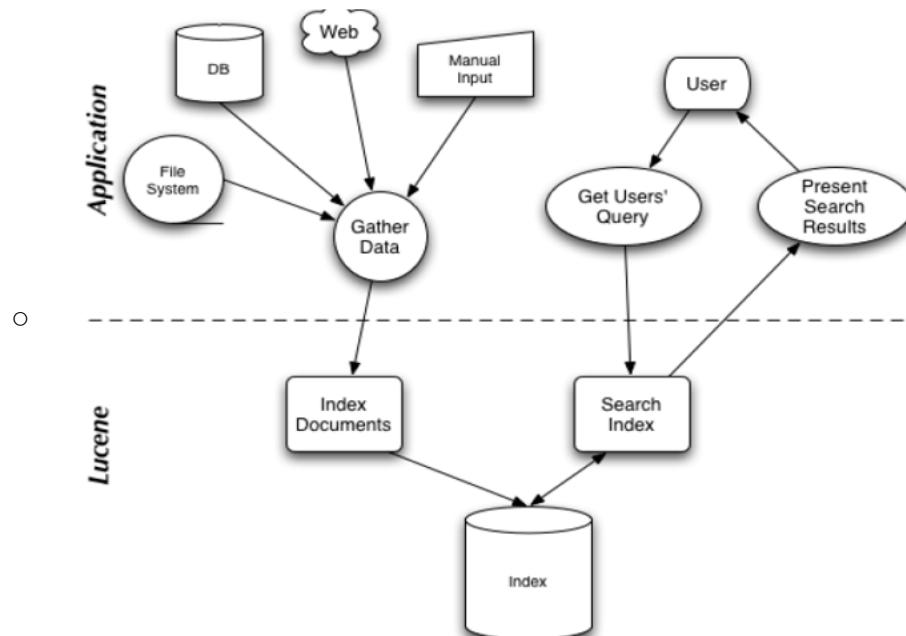


- 搜索引擎关键技术

- 下载模块
 - 爬虫: Cralwer、Robot、Spider
- 预处理模块
 - 去除噪音
 - 信息析取
- 索引模块
 - 分词、去停用词
 - 正排索引、倒排索引
 - 去重

- 查询与展示模块

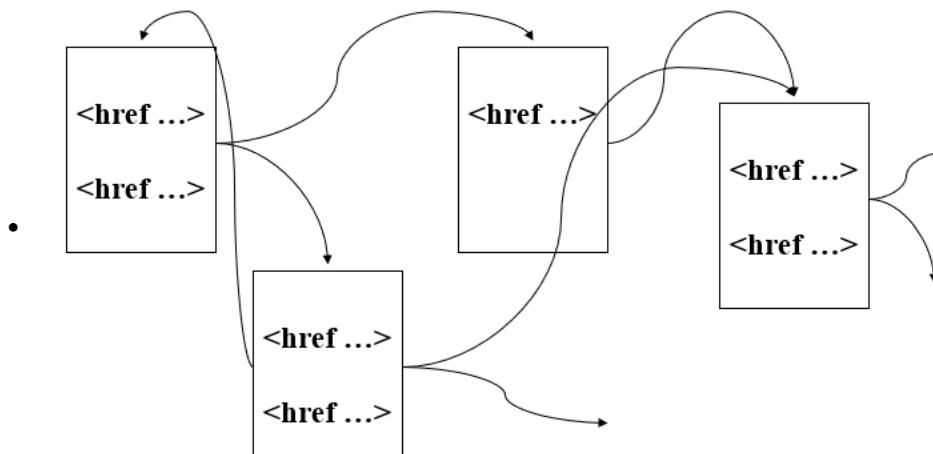
- 搜索引擎执行过程



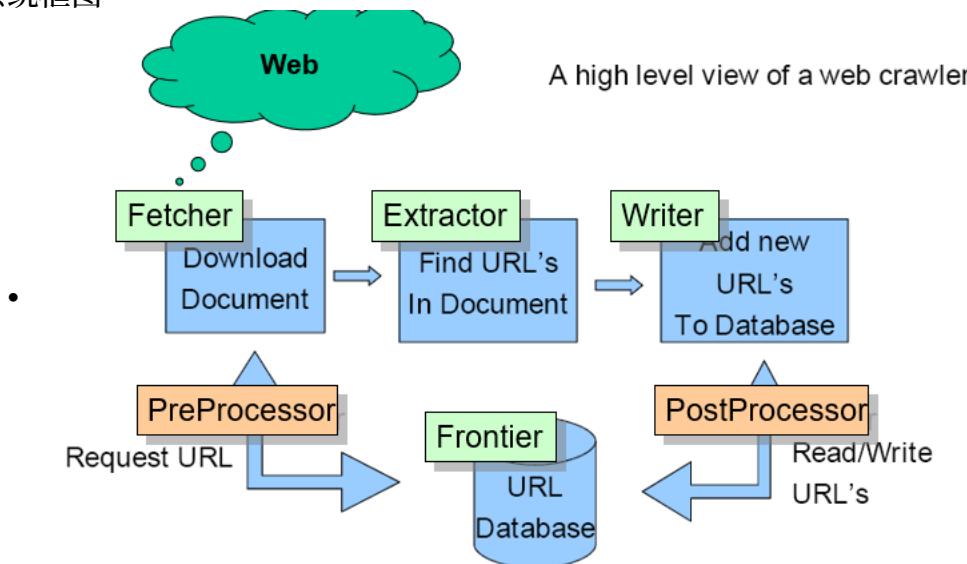
2 网络爬虫技术

怎样搜集?

- 网页为节点
- 网页中的HyperLink为有向边
- Crawl==图遍历,right?



系统框图



A More Complete Correct Algorithm

```
PROCEDURE SPIDER4(G, {SEEDS})
    Initialize COLLECTION <big file of URL-page pairs>
    Initialize VISITED <big hash-table>
    For every ROOT in SEEDS
        Initialize STACK <stack data structure>
        Let STACK := push(ROOT, STACK)

        While STACK Until URLcurr is not in VISITED
            Do URLcurr := pop(STACK)
                Until URLcurr is not in COLLECTION

                insert-hash(URLcurr, VISITED)
                PAGE := look-up(URLcurr)
                STORE(<URLcurr, PAGE>, COLLECTION)
                For every URLi in PAGE,
                    push(URLi, STACK)
    Return COLLECTION
```

STACK

用disk-based heap结构实现

还存在什么问题呢?

- Web规模在不断增长, 容量巨大
 - 必须具备高效率

- 1 billion pages / per month → 385 pages/sec
- Crawler 系统的瓶颈在哪里?
 - 加更多收集机器是否能解决问题?
 - 在 I/O 部分, look-up(), Store() → 特别是需要提高网络带宽利用率
 - 当数据量大到那些 data structure 不能够在内存中放下时 → 优化 Is url or pages VISITED

URL 不唯一性

- 不同 url 指向同一个网页
 - IP 地址和域名之间的多对多关系
 - 大规模网站用于负载平衡的技术: 内容镜像
 - “virtualhosting” 和 “Proxypass”: 不同的主机名映射到同一个 IP 地址, 发布多个逻辑网站的需要 (Apache 支持)
- 动态网页的参数
 - Sessionid
 - 上一页/下一页

对 URL 进行规范化

用一个标准的字符串表示协议 (http)

利用 canonical 主机名字

查 DNS 会返回 IP 和一个 canonical 名字

显式加上一个端口号 (80 也加上)

规范化并清理好文档路径

例如将 /books/..//papers/sigmod1999.ps 写成 /papers/sigmod1999.ps

Robot exclusion

- 检查
 - 在服务器文档根目录中的文件, robots.txt, 包含一个路径前缀表, crawlers 不应该跟进去抓文档, 例如


```
# AltaVista Search
User-agent: AltaVista Intranet V2.0 W3C Webreq
Disallow: /Out-Of-Date
# exclude some access-controlled areas
• User-agent: *
Disallow: /Team
Disallow: /Project
Disallow: /Systems
```
- 限制只是对 crawlers, 一般浏览无妨
 - “君子协定” (你的 crawler 可以不遵守)

防止系统异常

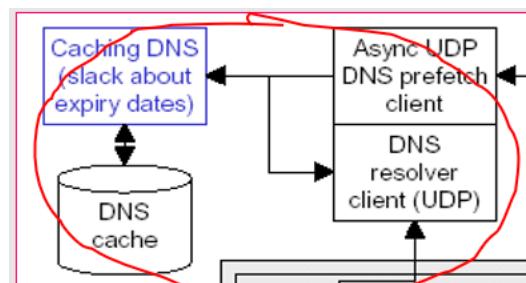
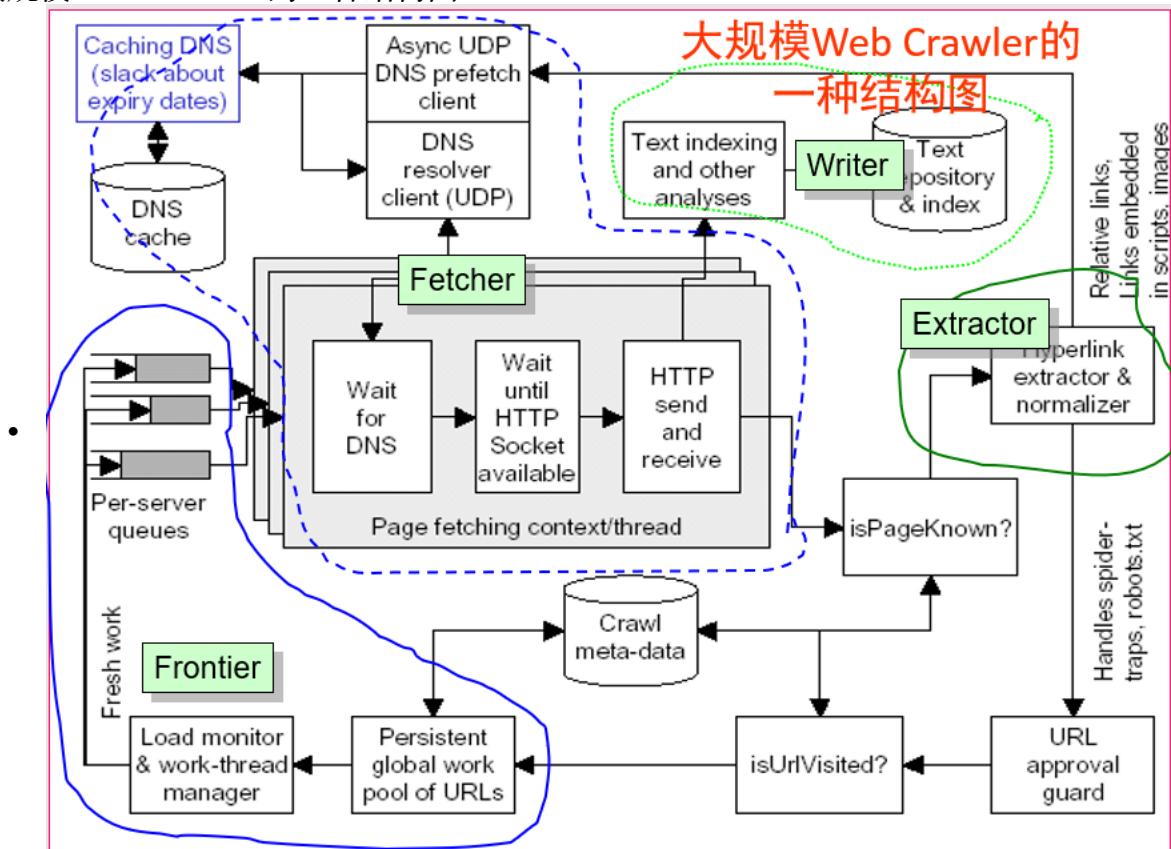
- 病态 HTML 文件
 - 例如, 有的网页含有 68kB null 字符
- 误导 Crawler 的网站
 - 用 CGI 程序产生无限个网页
 - 用软目录创建的很深的路径
 - www.troutbum.scom/Flyfactory/hatchline/hatchline/hatchline/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/hatchline
 - HTTP 服务器中的路径重映射特征

Web Crawler need...

- 快 Fast
 - Bottleneck? Network utilization
- 可扩展性 Scalable

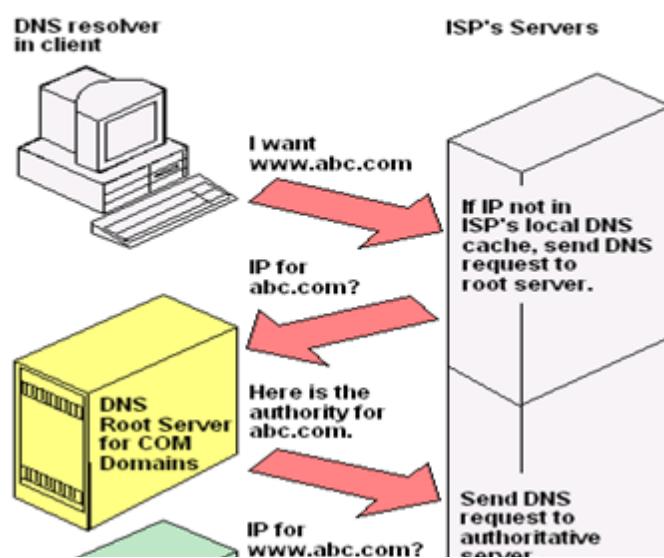
- Parallel,distributed
- 友好性Polite
 - DoS (DenyofServiceAttack) ,robottxt
- 健壮Robust
 - Traps,errors,crashrecovery
- 持续搜集Continuous
 - Batchorincremental
- 时新性Freshness

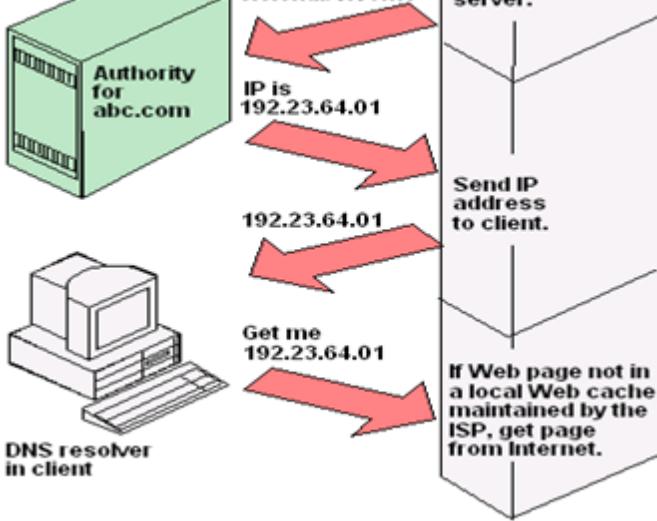
大规模WebCrawler的一种结构图



DNSresolver

From Computer Desktop Encyclopedia
© 2005 The Computer Language Co. Inc.





- Typical DNS request are synchronized!
- DNS can take long time to resolve host names
- Host name may never resolve
- 问题:同步和异步，阻塞和非阻塞调用是？

DNS解析

- 如果不采取措施，DNS地址解析会成为一个重要的瓶颈
 - 局域网中，DNS服务器通常较小，对付几百个工作站的常规操作没问题，但一个crawler产生的压力大很多
 - 通常的DNS解析系统调用是同步调用 (synchronized)
 - 避免同时从一个服务器抓许多网页也使DNS的cache能力发挥不出来
- 怎样提高DNS解析模块的性能？
 - 并发DNS client
 - 缓存cache dns results
 - 预取prefetch client

并发的地址解析client

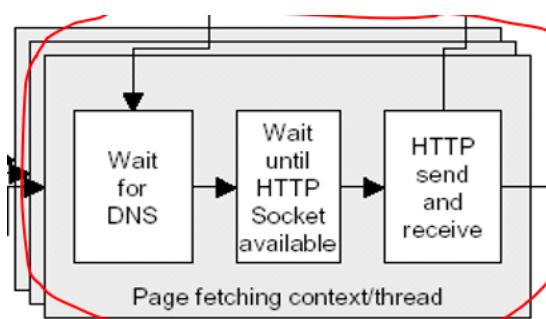
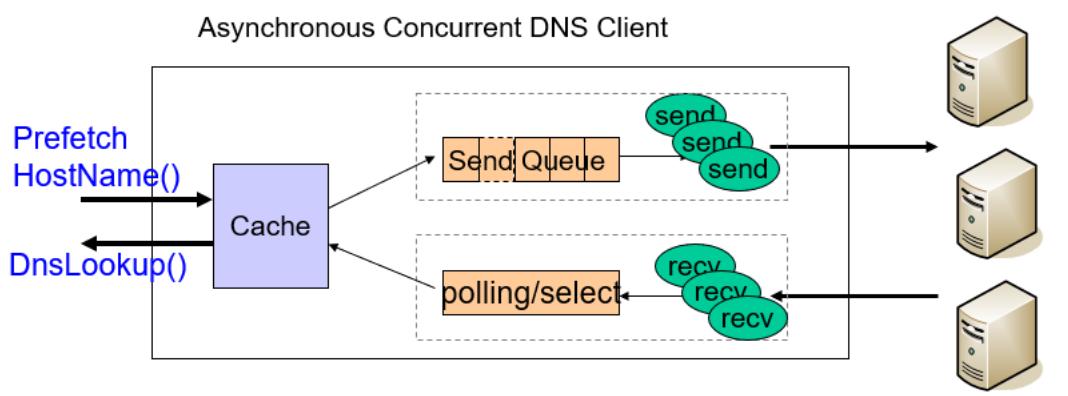
- 一般系统（例如UNIX）提供的DNSclient没有考虑cralwer的需求，带来两个问题：
 - 以gethostbyname()为基础，它不可重入非线程安全
 - 不会考虑在多个DNSserver之间分配负载。
- 因此一个customclient很必要。
 - 专门对付多个请求的并发处理
 - 容许一次发出多个解析请求(multiplexer)
 - 协助在多个DNSserver之间做负载分配（例如根据掌握的URL进行适当调度）

缓存服务器 (DNSCachingServer)

- 大缓存容量，跨DNS系统的刷新保持内容
 - Internet的DNS系统会定期刷新，交换更新的域名和IP的信息。
 - 普通的DNScache一般应该尊重上级DNS服务器带来的域名“过期”的信息，但用于爬取网页的DNScache不一定如此，以减小开销（让缓存中有些过期的无所谓，但也要注意安排适时刷新）
- 映射尽量放在内存，可以考虑用一台专门的PC

预取client

- 为了减少等待查找涉及新主机的地址的时间：尽早将主机名投给DNS系统
- 步骤
 - 分析刚得到的网页
 - 从HREF属性中提取主机名（不是完整的URL）
 - 向缓存服务器提交DNS解析请求
 - 结果放到DNScache中（后面可能有用，也可能用不上）
- 通常用UDP实现
 - 非连接，基于包的通信协议，不保证包的投递
- 用不着等待解析的完成
 - Asynchronous调用



Page fetching

- 获取一个网页需要多少时间?
 - 局域网的延迟在110ms，带宽为10~100Mbps；Internet的延迟在100~500ms，带宽为0.01~0.2Mbps
 - 在一个网络往返时间RTT为200ms的广域网中，服务器处理时间SPT为100ms，那么TCP上的事务时间就大约500ms (2RTT+SPT)
 - 网页的发送是分成一系列帧进行的，则发送1个网页的时间量级在 $(13KB/1500B) * 500ms \approx 4s$
- Problem:
 - network connection and transfer overheads
- Solutions:
 - Multiple concurrent fetches

多个并发的抓取

- 管理多个并发的连接
 - 单个下载可能需要几秒钟
 - 同时对不同的HTTP服务器建立许多socket连接
- 过多的硬件并行好处不大
 - 抓取的性能瓶颈主要在网络和硬盘
- 两种基本方法
 - 用多线程/多进程
 - 用异步I/O:带事件处理的非阻塞sockets

多线程方法

- 逻辑线程（进程）
 - 操作系统控制的线程（例如pthread），或
 - 并发进程（地址空间分离，好处是相互干扰不大）
- 通常根据经验采用一个预先确定的线程数量
 - 平均4秒抓取时间下，按带宽100%利用计算，可完成 $4 * 100Mb / (8 * 13KB) \approx 4000$ 个网页，约启动4000个线程
- 程序设计
 - 创建一个clientsocket
 - 将该socket连接到服务器的HTTP服务上
 - 发送HTTP请求
 - 读socket(recv)直到没有更多的东西可读
 - 还可以是通过时间超时判断中止
 - 关闭socket
- 通常用阻塞的系统调用（简单；等待和计算的重叠靠并发多线程“随机”实现）

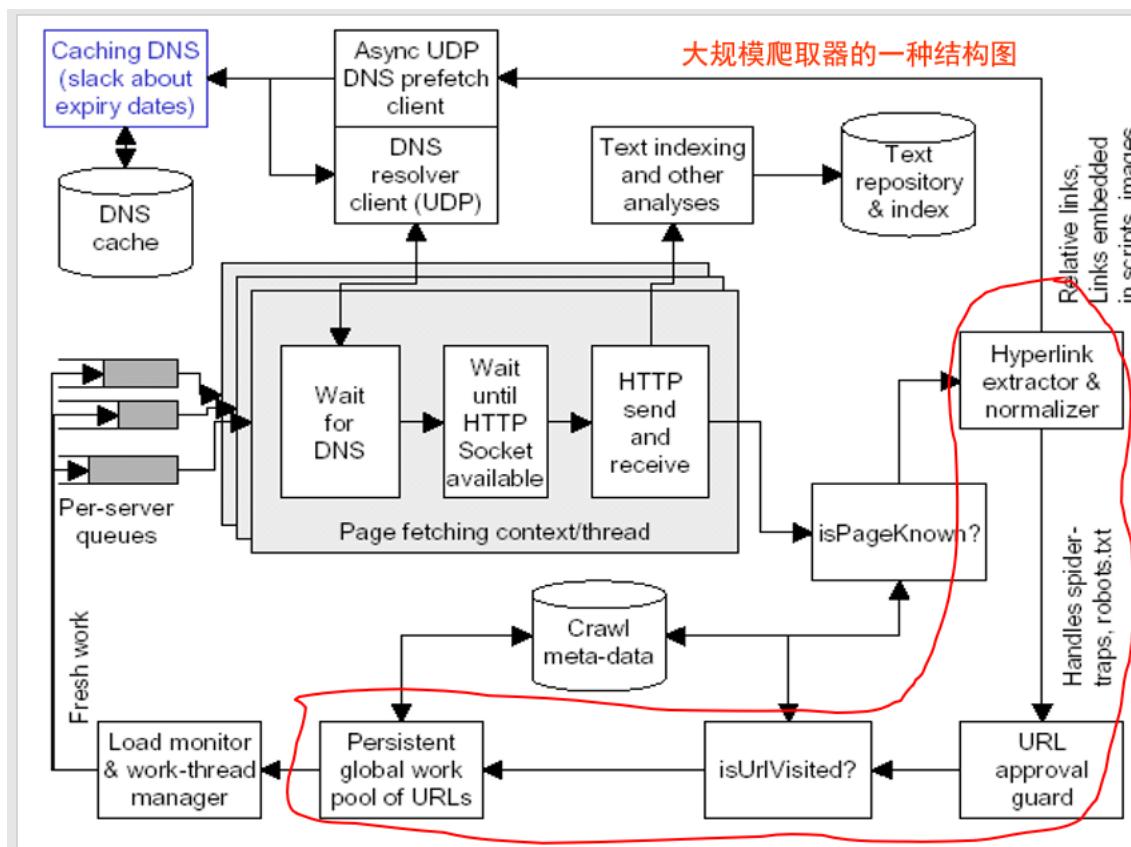
多线程：问题

多线程的性能代价 (performance penalty)

- 内存地址空间不够
- 互斥
 - 共享的工作数据区数据结构的并发访问，用程序的方式协调这种访问的开销可能会更大（线程互斥，或者进程通信，都很复杂）
- 磁盘访问代价
 - 并发线程在完成抓取后，进行文档存储时会形成大量交叉的，随机磁盘I/O，外部表现就是磁盘不断的寻道，很慢。

异步非阻塞socket和事件处理：单进程

- 异步非阻塞sockets
 - connect, send或recv调用立刻返回，不需要等网络操作的完成（因此可以连续给出多个）
 - 随后可以通过polling来检查完成的状态
- “select”系统调用：专门针对非阻塞socket
 - 让调用者挂起，直到数据可以通过socket读/写，或者期限超时
 - 可以同时监管多个sockets的状态，一个可用就返回
- select体现了更高效的存储管理：一次select返回一个socket标识
 - 网页抓取的完成不相互影响（自然地序列化了！）
 - 于是在共享的任务池上也不需要加锁或者信号灯！



消除已经访问过的URL

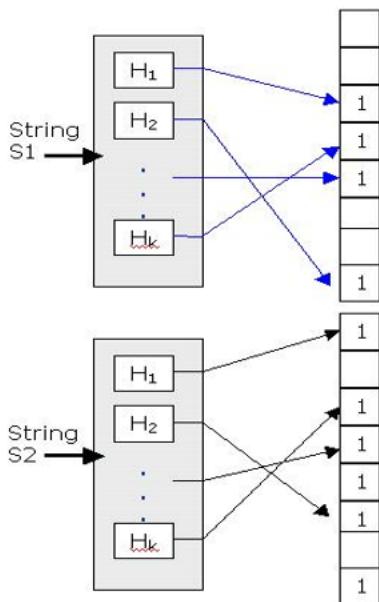
- 检查某个URL是否已经被抓过了
 - 在将一个新的URL(规范化后的)放到工作池之前
 - 要很快，不要在这里形成性能瓶颈（检查将要访问磁盘）
 - 符合条件（即未被访问过）的URLs放到crawler的任务中
- 优化方法
 - URL用fingerprint(如MD5)来记录，减少内存开销
 - 利用访问的时空局部性--Cache
 - 海量数据的高效率查找表
 - Btree
 - Bloomfilter

IsURLVisisted:Btree实现

- 一种平衡搜索树，适合基于外存的查找
- 每个节点和磁盘的block一样大
 - 每个节点可以容纳很多key
- 每个节点可以有很多子节点，例如1000
 - 树通常很“矮”，于是搜索涉及的节点个数不多
- 维护节点中数据key的有序性有助于提高查找效率
 - 顺序数据访问将有效利用I/O系统的缓存

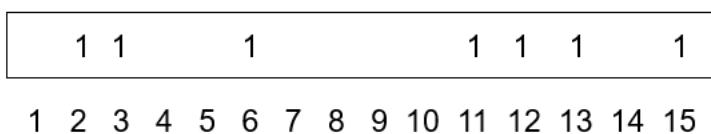
IsURL Visisted:Bloom filter

- Bloom filters are compact data structures for probabilistic representation of a set in order to support membership queries.
- 查找集合是否包含一个元素（查找表）
- False Positive
 - 可由参数n(string number), k(hash function number), m(bit-array size)调节
 - Each value of S maps to a value of $H[S_n]$
 - Typically: Size of (S) >> Size of ($H[S_n]$)
 - $H[S_n]$ evenly distributed over values of S_n

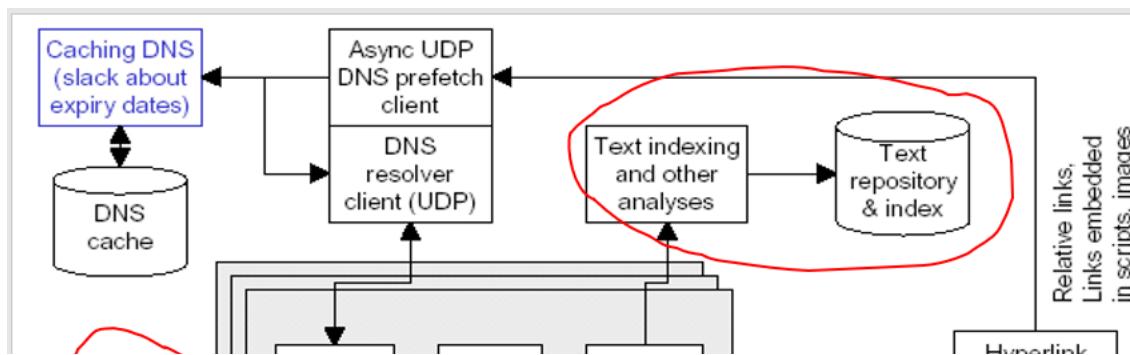


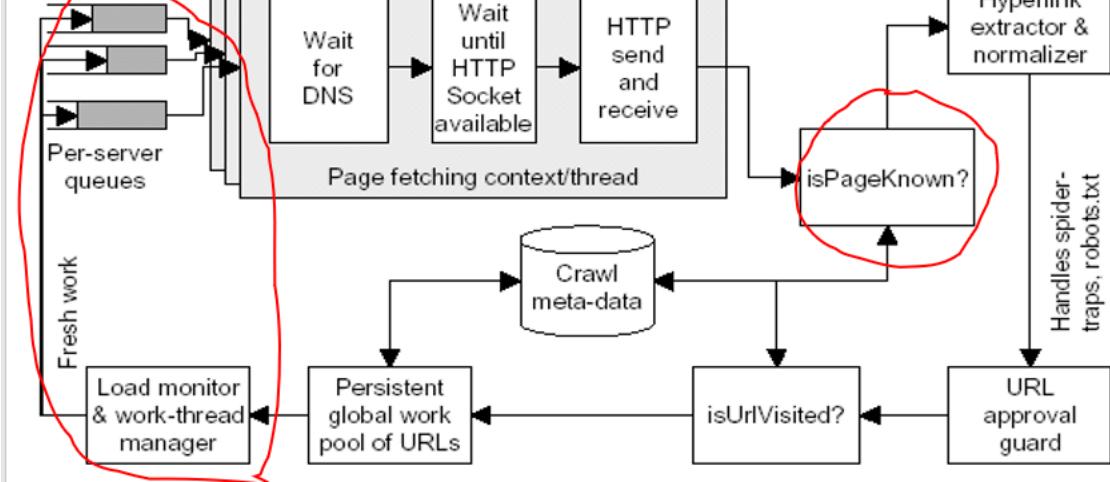
Simple Example

- Use a bloom filter of 16 bits
 - $h_1(key) = key \bmod 16$
 - $h_2(key) = key \bmod 14 + 2$
- Insert numbers 27, 18, 29 and 28



- Check for 22:
 - $H_1(22) = 6$, $H_2(22) = 10$ (not in filter)
- Check for 51
 - $H_1(51) = 3$, $H_2(51) = 11$ (false positive)





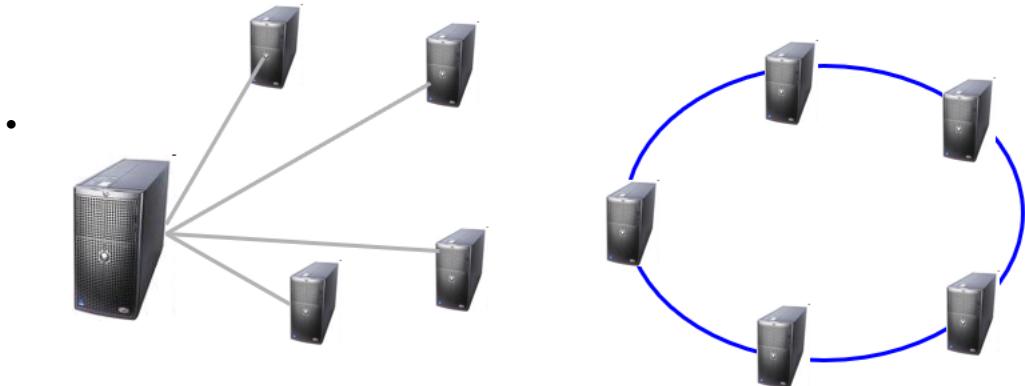
避免在重复的网页上再提取链接

- 检测完全重复的网页 (exactduplicates)
 - 对比不同URL对应网页的MD5摘要 (访问一次磁盘)
 - 或者, 将相对于网页u的链接v表示为(h(u);v), 其中h(u)是u的内容的散列。这样, 两个别名的相同相对链接就有同样的表示, 直接放到isUrlVisited中检查
- 检测接近重复的网页 (nearduplicates)
 - 即使是一个字符的改变也会完全改变MD5摘要
 - 例如, 网页的转载常伴随有日期或者网站管理者名字的变化
 - 解决方案: Shingling (分段)

分布式爬虫

分布式计算

- 中央控制节点Master-Slave
- 对等Peer-to-peer模式



任务划分问题

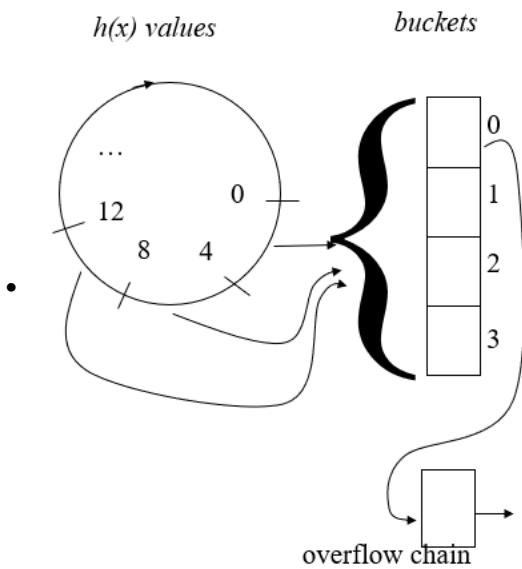
- M个节点同时执行搜集
- 问题: 如何有效的把N个网站的搜集任务分配到M个机器上去?
- 目标: 任务分配得均匀 (Balance)



Hashing

- 从一个值均匀分布的 hash 函数开始:

- $h(\text{name}) \rightarrow$ a value (e.g., 32-bit integer)
- 把 values 映射到 hash buckets
 - 一般取模 mod (# buckets)
- 把 items 放到buckets
 - 冲突, 需要 chain解决冲突



在机器间划分HashTable

- 简单划分把buckets按组分配到对应机器上去
 - 对应表可以发给用户, 或者维护一个全局目录
 - 可以控制均匀分配, 或者按capacity分配多少
 - 查找十分简单
- 问题?
 - 新增加节点, 节点crash, 节点重新加入,
 - 机器数目变化情况下……

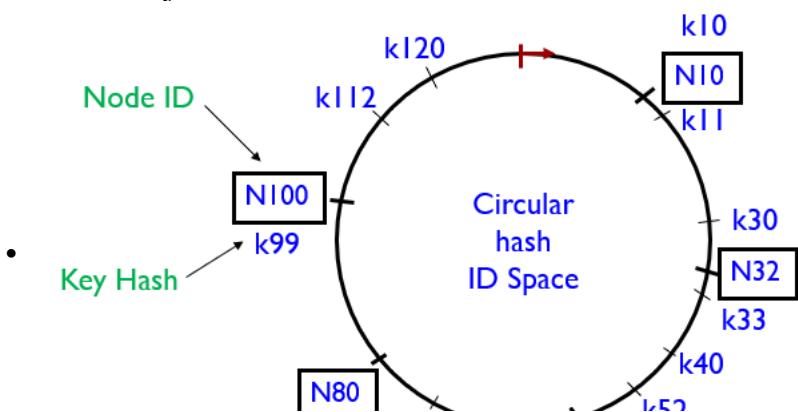
We need...

- 在一个分布式系统中, 无需集中目录, 也无需担心节点失效的一种查找元素位置的方法

Basic Ideas

- 使用一个巨大的hash key空间
 - SHA-1 hash: 20B, or 160 bits
 - 把它组织成 “**circular ring**”(在 2^{160} 处回到0)
- 我们把 **objects' keys** (这里是URL) 和**nodes' IP addresses** 都hash映射到一个hash key 空间
 - “www.pku.edu.cn” \rightarrow SHA-1 \rightarrow K10
 - 130.140.59.2 \rightarrow SHA-1 \rightarrow K12

Hashes a Key to its Successor



Incremental Crawling

问题

- 有限的资源条件下
 - 网络带宽, 存储空间
- Crawler系统怎样和变化的Web同步?
 - 如何估计网页变化频率, 来预测其更新时间?
 - 如何度量搜集结果的优劣?
 - 按预测到的更新时间去抓取是最优策略吗?

Summary

- Crawler面临的难题
 - Scalable, fast, polite, robust, continuous
- 实现高效率的基本技术
 - Cache
 - Prefetch
 - Concurrency
 - 多进程/多线程
 - 异步I/O
- 有趣的技术
 - Bloomfilter
 - ConsistentHashing

Google搜索核心算法

- 网页级别 (PageRank)
 - 按网页链接广泛程度判断网页重要性, 是Google中表示网页重要性的综合性指标。
- 页面分析 (PageAnalysis)
 - 按页面标题是否出现关键词、网页内关键词出现的频率及关键词出现的位置确定哪些网页与正在执行的搜索密切相关。

3 PageRank技术

- PageRank会通过解析一个具有5亿多个变量和20亿个条件方程, 对网页的重要性进行客观测定。PageRank会将网页A上指向网页B的链接解释为由网页A对网页B所投的一票, 而不是计算直接的链接数。然后Page根据网页收到的票数来评估其重要性。
- Page也会考虑发出投票的每个网页的重要性, 也就是某些网页的投票具有价值较大, 为该链接的页面富裕的价值也就较大。重要的网页会得到较高的PageRank, 并出现在搜索的顶部。Google技术是利用网络融合信息来确定网页的重要性。因为没有人工干涉, 也就不对结果进行操纵, 所以用户一直信任Google是一个不会因为付费而影响排名的客观信息来源。

PageRank对搜索结果的影响

- 结合了所有网页重要性和相关性指标, Google将最相关和最可靠的结果放在搜索结果的顶端。
- 一般而言, PageRank对于排名的影响比页面分析还高。

PageRank算法思想简介

基本依据:

- PageRank基于假设关系——“许多优质的网页链接的网页, 必定是优质网页”, 判定所有网页的重要性。

PageRank要点

- 链入链接数
 - 单纯意义上的受欢迎度指标
- 链入链接是否来自受欢迎程度高的页面
 - 有根据的受欢迎指标
- 链入链接源页面的链出链接数

- 被选中的概率指标

PageRank计算

- 互联网是一个有向图
- 每一个网页是图的一个顶点
- 网页间的每一个超链接是图的一个有向边
- 用邻接矩阵来表示图，即：定义邻接矩阵为G，若网页j到网页i有超链接，则 $g_{ij}=1$ ；反之， $g_{ij}=0$ 。
- 显然，如果网页有N个，则矩阵为N×N的0、1方阵。
- 定义邻接矩阵为G，若网页j到网页i有超链接，则 $g_{ij}=1$ ；反之， $g_{ij}=0$ 。
- 记矩阵G的列和、行和分别是

$$c_j = \sum_i g_{ij}$$

$$r_i = \sum_j g_{ij}$$

- 它们分别给出了页面j的链出链接数目和链入链接数目
- 假设我们在上网的时候浏览页面并选择下一个页面，这个过程与过去浏览过哪些页面无关，而仅依赖于当前所在的页面，那么这一选择过程可以认为是一个有限状态、离散时间的随机过程，其状态转移规律用Markov链描述。
- 定义转移概率矩阵 $A=(a_{ij})$

$$a_{ij} = \frac{g_{ij}}{c_j} \quad i, j = 1 \dots n$$

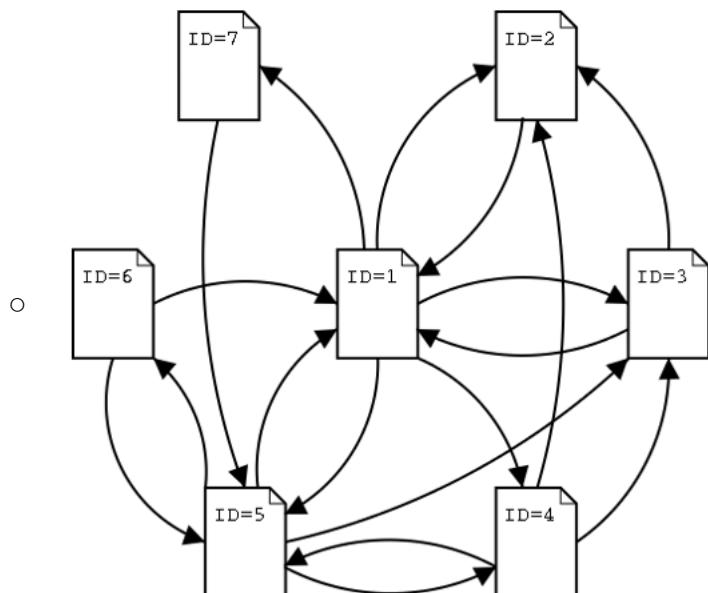
- 根据Markov链的基本性质，对于正则Markov链，存在平稳分布 $\alpha = (x_1, x_2, \dots, x_N)^T$ 满足

$$A\alpha = \alpha \quad \sum_i x_i = 1$$

- α 表示在极限状态（转移次数趋于无限）下各网页被访问的概率分布。
- α 定义为网页的PageRank向量， x_i 表示第i个网页的PageRank值

举例

- 某7个网页之间的链接关系图



- 网页链接图的邻接矩阵

$$\textcircled{G} = \begin{matrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

- PageRank的计算

状态转移概率矩阵A

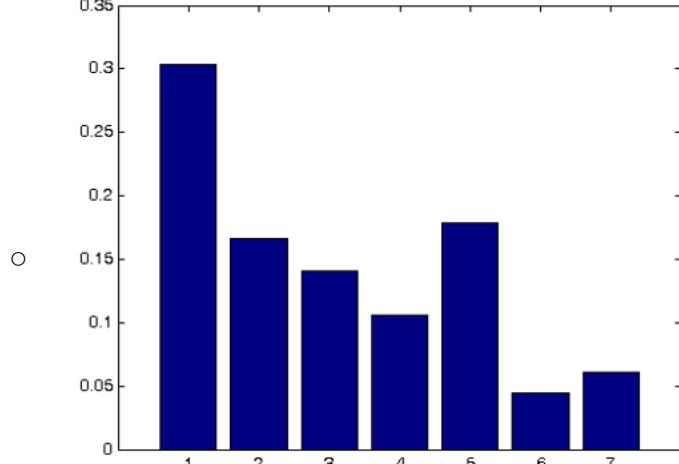
$$\textcircled{A} = \begin{matrix} 0 & 1 & 1/2 & 0 & 1/4 & 1/2 & 0 \\ 1/5 & 0 & 1/2 & 1/3 & 0 & 0 & 0 \\ 1/5 & 0 & 0 & 1/3 & 1/4 & 0 & 0 \\ 1/5 & 0 & 0 & 0 & 1/4 & 0 & 0 \\ 1/5 & 0 & 0 & 1/3 & 0 & 1/2 & 1 \\ 0 & 0 & 0 & 0 & 1/4 & 0 & 0 \\ 1/5 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

求特征值对应的特征向量
0.699456533837389
0.382860418521518
0.323958815672054
0.242969111754040
0.412311219946251
0.103077804986563
0.139891306767478

归一化 $\alpha =$ →

0.303514376996805
0.166134185303514
0.140575079872204
0.105431309904153
0.178913738019169
0.0447284345047923
0.0607028753993610

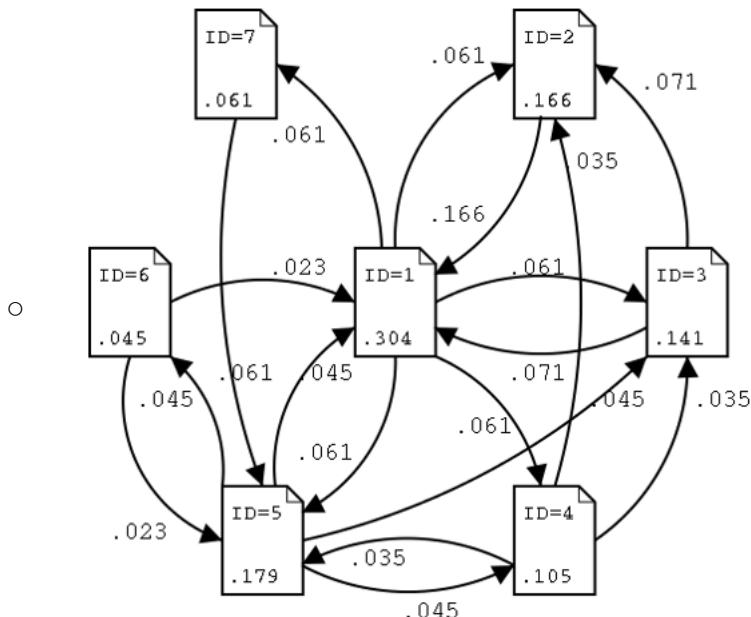
- 7个网页的PageRank值



- PageRank结果的评价
 - 将 PageRank 的评价按顺序排列(PageRank小数点3位四舍五入):

名次	PageRank	文件 ID	发出链接 ID	被链接 ID
1	0.304	1	2, 3, 4, 5, 7	2, 3, 5, 6
2	0.179	5	1, 3, 4, 6	1, 4, 6, 7
3	0.166	2	1	1, 3, 4
4	0.141	3	1, 2	1, 4, 5
5	0.105	4	2, 3, 5	1, 5
6	0.061	7	5	1
7	0.045	6	1, 5	5

- 页面之间相互关系及状态转移图



- PageRank结果的评价
 - 首先应该关注的是，PageRank的名次和链入链接的数目是基本一致的。无论链接多少链出链接都几乎不会影响PageRank，相反地有多少链入链接却是从根本上决定PageRank的大小。
 - 但是，仅仅这些并不能说明第1位和第2位之间的显著差别，在链入链接相同的情况下，链出链接数也影响PageRank的大小。(同样地、第3位和第4位，第6位和第7位之间的差别)。
 - 总之，绝妙之处在于PageRank不只是通过链入链接数来决定的。
 - 让我们详细地看一下。ID=1的页面的PageRank是0.304，占据全体的三分之一，成为了第1位。特别需要说明的是，起到相当大效果的是从排在第3位的ID=2页面中得到了所有的PageRank(0.166)数。ID=2页面有从3个地方过来的链入链接，而只有面向ID=1页面的一个链接，因此(面向ID=1页面的)链接就得到了所有的PageRank数。不过，就因为ID=1页面是链出链接和链入链接最多的页面，也可以理解它是最受欢迎的页面。
 - 反过来，最后一名的ID=6页面只有ID=1的15%的微弱评价，这可以理解为是因为没有来自PageRank很高的ID=1的链接而使其有很大地影响。总之，即使有同样的链入链接的数目，链接源页面评价的高低也影响PageRank的高低。

- 页面之间状态转移图的验证

- 实际地试着计算一下PageRank的收支。因为 $\lambda=1$ 所以计算很简单，只要将自各页的流入量单纯相加即可。

譬如 ID=1 的流入量

- $= (\text{ID=2发出的Rank}) + (\text{ID=3发出的Rank}) + (\text{ID=5发出的Rank}) + (\text{ID=6发出的Rank})$

$$= 0.166 + 0.141/2 + 0.179/4 + 0.045/2$$

$$= 0.30375$$

PageRank算法实际应用的困难

- 现实世界和假想模型不同；
- 实际计算上，80亿的网页数量，计算非常困难。

现实世界和假想模型的不同

- 现实世界：
 - 顺着链接前进的话，有时会走到完全没有链出链接的网页；
 - 同样道理，只有链出的链接而没有链入的链接的页面也是存在的。（不考虑）
 - 有时候也有链接只在一个集合内部旋转而不向外界链接的现象。
- 假想的理论模型：
 - 正则链（或称回归类，无吸收状态）

实际问题可能出现的情况

- 最大特征值不唯一，导致特征向量不唯一，无法对网页进行排序

问题的解决方法

为了解决这样的问题，PageRank考虑了这样一种浏览模型——用户虽然在许多场合都顺着当前页面中的链接前进，但时常会跳跃到完全无关的页面。

- 将「时常」这个概率固定为 15% 来计算。则用户在 85% 的情况下沿着链接前进，但在 15% 的情况下会突然跳跃到无关的页面中去。（注：PageRank 的原始参数是 87% (= 1/1.15) 和 13% (= 0.15/1.15)。）

- 即

$$A' = c * A + (1-c) * [1/N]$$

- 其中， $[1/N]$ 是所有要素为 $1/N$ 的 N 次正方行列， $c = 0.85 (= 1 - 0.15)$ 。 A' 是新的状态转移矩阵。

- 也就是说，根据 PageRank 的变形，原先求矩阵 A 的特征值问题变成了求矩阵 A' 的最

方法的数学解释

- 从数学角度看，把非正则链的状态转移矩阵正则化，就是把不是强联通的图变成强联通的，是一种变换操作。
- 对全部的要素都考虑0.15的转移概率，意味着将原本非正则的状态转移矩阵转换为正则的状态转移矩阵，将原本并非强连通的图变成了强联通的。
- 相对于原来的状态转移矩阵，这样的变换操作能保证最大特性值的次数为1，也就保证了PageRank的存在。

PageRank数值计算难点

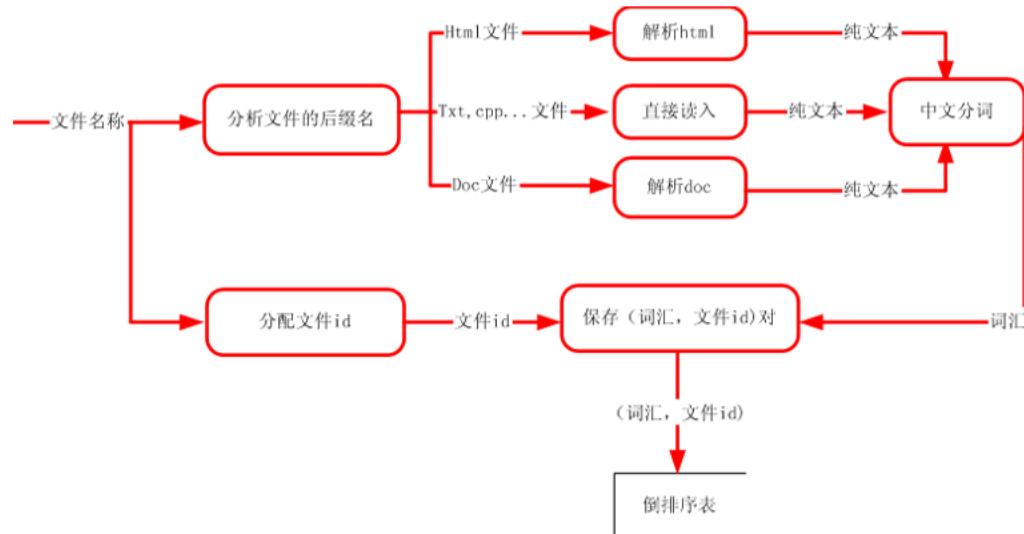
- 计算机容量限制
 - 假设N是104的order。通常，数值计算程序内部行列和矢量是用双精度记录的，N次正方行列A的记忆领域为 $\text{sizeof(double)} \times N \times N = 8 \times 104 \times 104 = 800\text{MB}$ 。N如果变成105或106的话，就变成80GB, 8TB。这样的话不用说内存就连硬盘也已经很困难了。目前，Google处理着80亿以上的页面，很显然，已知的这种做法已经完全不适用了。
- 收敛问题
 - 特征向量的求解，就是求解方程，是N元一次方程组，一般地不能得到分析解，所以只能解其数值。
 - 然而，常用的迭代求解方法会导致收敛速度很慢。

4 索引技术

一个搜索引擎需要的几件事

- 自动下载尽可能多的网页
- 建立快速有效的索引
- 根据相关性对网页进行公平准确的排序

搜索引擎中文件索引过程



布尔代数与搜索引擎索引

- 布尔 (GeorgeBoole)是十九世纪英国一位小学数学老师
 - 他生前没有人认为他是数学家
- 布尔代数简单得不能再简单了
 - 运算的元素只有两个1 (TRUE, 真)和0 (FALSE, 假)
 - 基本的运算只有“与” (AND)、 “或” (OR)和 “非” (NOT)三种
- 布尔代数提出后80多年里，它没有什么像样的应用
- 1938年香农在他的硕士论文中指出用布尔代数来实现开关电路
 - 布尔代数成为数字电路的基础
 - 所有的数学和逻辑运算，加、减、乘、除、乘方、开方等等，全部能转换成二值的布尔运算
- 查询语句 “原子能AND应用AND(NOT原子弹)”
 - 包含原子能 (TRUE, 1)
 - 包含应用 (TRUE, 1)
 - 不包含原子弹 (FALSE, 0)

- 关键字与文档表示

	docA	docB	docC	docD	docS
key1	0	1	0	1	*
key2	1	1	1	0	*
key3	1	0	0	0	*
key4	0	1	1	1	1
○ key5	0	0	0	0	*
key6	0	0	1	0	0
key7	1	1	1	1	*
key8	1	1	1	1	*
key9	0	1	0	1	1
key10	0	0	1	1	*

索引方法

● 正排序索引

● 建模

- docA=0110001100
- docB=1101001110
- docC=0101011101
- docD=1001001111
- docS= 0001000010

● 检索

- docA & docS !=docS
- docB & docS =docS
- docC & docS !=docS
- docD & docS =docS

● 倒排序索引

● 建模

- key1=0101
- key2=1110
-

● 检索

- key4 & ~key6 & key9
=0101 => docB,docD

倒排索引技术

- 为了提高对网页文本内容的查找速度，需要对网页文本内容建立索引（Index）。建立索引的本质是建立标记来指示内容的位置。
- 网络搜索通常是全文搜索，而倒排序索引就是一种高效的全文索引技术。
- 倒排序索引有两部分组成：词汇表和位置表。
- 词汇表是文本中所有词汇的集合。
- 位置表由词汇在文本中出现的地址列表构成，每个词一个列表。
- 在多文档的情况下，词的地址列表有文档标识符和文档内的位置构成。
- 即按照（词汇t, 文档d）二元组排序，获得td表，然后将相同的文档标识符合并到一个列表中，形成了倒排序索引。

索引存储

- 动态语料库

- BerkeleyDB2存储管理
 - 能够频繁的增加，修改和删除文档。
- 静态收集
 - 索引压缩技术，为了提高查找速度，将索引压缩，以便将其放在内存中
 - 对文档ID降序排序，保存第一个标识符，以后只保存差值—delta编码
- 文件中的关键词进行了压缩，关键词压缩为<前缀长度，后缀>
 - 例如：当前词为“阿拉伯语”，上一个词为“阿拉伯”，那么“阿拉伯语”压缩为<3，语>。
- 其次大量用到的是对数字的压缩，数字只保存与上一个值的差值
 - 例如当前文章号是16389（不压缩要用3个字节保存），上一文章号是16382，压缩后保存7（只用一个字节）。

倒排索引查询

- Setp1：在词汇表中查找用户中查询包含的词汇；对词典二元查找、找到该词
- Setp2：查找并取出所有查询词在位置表中的地址列表；
- Setp3：根据各查询词的地址列表计算需检索文档的或段落的标识符。