要在网络上爬行非常简单，基本的算法是：（a）取得一个网页（b）解析它以提取所有的链接URLs（c）对于所有没有见过的URLs重复执行 （a）-（c）。但是，网络的大小（估计有超过40亿的网页）和他们变化的频率（估计每周有7%的变化）使这个计划由一个微不足道的设计习题变成一个非常 严峻的算法和系统设计挑战。实际上，光是这两个要素就意味着如果要进行及时地，完全地爬行网络，步骤（a）必须每秒钟执行大约1000次，因此，成员检测 （c）必须每秒钟执行超过10000次，并有非常大的数据储存到主内存中。这个要求有一个分布式构造，使得成员检测更加复杂。  
 一个非常重要的方法加速这个检测就是用cache（高速缓存），这个是把见过的URLs存入主内存中的一个（动态）子集中。这个论文最主要的成果就是仔细 的研究了几种关于网络爬虫的URL缓存技术。我们考虑所有实际的算法：随机置换，静态cache，LRU，和CLOCK，和理论极限：透视cache和极 大的cache。我们执行了大约1800次模拟，用不同的cache大小执行这些算法，用真实的log日志数据，获取自一个非常大的33天的网络爬行，大 约执行了超过10亿次的http请求。  
 我们的主要的结论是 cache是非常高效的-在我们的机制里，一个有大约50000个入口的cache可以完成80%的速率。有趣的是，这cache的大小下降到一个临界 点：一个足够的小一点的cache更有效当一个足够的大一点的cache只能带来很小的额外好处。我们推测这个临界点是固有的并且冒昧的解释一下这个现 象。  
1.介绍  
皮尤基金会最新的研究指出：“搜索引擎已经成为互联网用户不可或缺的工具”，估计在2002年中期，初略有超过1半的美国人用网络搜索获取信息。因此，一 个强大的搜索引擎技术有巨大的实际利益，在这个论文中，我们集中于一方面的搜索技术，也就是搜集网页的过程，最终组成一个搜索引擎的文集。  
 搜索引擎搜集网页通过很多途径，他们中，直接提交URL，回馈内含物，然后从非web源文件中提取URL，但是大量的文集包含一个进程叫 crawling 或者 SPIDERing，他们递归的探索互联网。基本的算法是：  
Fetch a page  
Parse it to extract all linked URLs  
For all the URLs not seen before，repeat（a）-(c)  
网络怕从一般开始于一些 种子URLs。有些时候网络爬虫开始于一个正确连接的页面，或者一个目录就像：yahoo.com，但是因为这个原因相关的巨大的部分网络资源无法被访问到。（估计有超过20%）  
如果把网页看作图中的节点，把超链接看作定向的移动在这些节点之间，那么网络爬虫就变成了一个进程就像数学中的图的遍历一样。不同的遍历策略决定 着先不访问哪个节点，下一个访问哪个节点。2种标准的策略是深度优先算法和广度优先算法-他们容易被实现所以在很多入门的算法课中都有教。  
但是，在网络上爬行并不是一个微不足道的设计习题，而是一个非常严峻的算法和系统设计挑战因为以下2点原因：  
网络非常的庞大。现在，Google需要索引超过30亿的网页。很多研究都指出，在历史上，网络每9-12个月都会增长一倍。  
网络的页面改变很频繁。如果这个改变指的是任何改变，那么有40%的网页每周会改变。如果我们认为页面改变三分之一或者更多，那么有大约7%的页面每周会变。  
这2个要素意味着，要获得及时的，完全的网页快照，一个搜索引擎必须访问1亿个网页每天。因此，步骤（a）必须执行大约每秒1000次，成员检测 的步骤（c）必须每秒执行超过10000次，并有非常大的数据储存到主内存中。另外，网络爬虫一般使用一个分布式的构造来平行地爬行更多的网页，这使成员 检测更为复杂：这是可能的成员问题只能回答了一个同行节点，而不是当地。  
 一个非常重要的方法加速这个检测就是用cache（高速缓存），这个是把见过的URLs存入主内存中的一个（动态）子集中。这个论文最主要的成果就是仔细 的研究了几种关于网络爬虫的URL缓存技术。我们考虑所有实际的算法：随机置换，静态cache，LRU，和CLOCK，和理论极限：透视cache和极 大的cache。我们执行了大约1800次模拟，用不同的cache大小执行这些算法，用真实的log日志数据，获取自一个非常大的33天的网络爬行，大 约执行了超过10亿次的http请求。  
 这个论文像这样组织的：第2部分讨论在文学著作中几种不同的爬行解决方案和什么样的cache最适合他们。第3部分介绍关于一些cache的技术和介绍了 关于cache几种理论和实际算法。第4部分我们实现这些算法，在实验机制中。第5部分描述和讨论模拟的结果。第6部分是我们推荐的实际算法和数据结构关 于URLcache。第7部分是结论和指导关于促进研究。  
2.CRAWLING  
 网络爬虫的出现几乎和网络同期，而且有很多的文献描述了网络爬虫。在这个部分，我们呈现一个摘要关于这些爬虫程序，并讨论问什么大多数的网络爬虫会受益于URL cache。  
 网络爬虫用网络存档雇员多个爬行进程，每个一次性完成一个彻底的爬行对于64个hosts 。爬虫进程储存非本地的URLs到磁盘；在爬行的最后，一批工作将这些URLs加入到下个爬虫的每个host的种子sets中。  
 最初的google 爬虫，实现不同的爬虫组件通过不同的进程。一个单独的URL服务器进行维护需要下载的URL的集合；爬虫程序获取的网页；索引进程提取关键字和超链接；URL解决进程将相对路径转换给绝对路径。这些不同的进程通过文件系统通信。  
 这个论文的中实验我们使用的meractor网络爬虫。Mercator使用了一个独立的集合，通信网络爬虫进程。每个爬虫进程都是一个有效的web服务 器子集；URLs的分配基于URLs主机组件。没有责任通过TCP传送这个URL给网络爬虫，有责任把这些URLs绑在一起减少TCP开销。我们描述 mercator很多的细节在第4部分。  
 任何网络爬虫必须维护一个集合，装那些需要被下载的URLs。此外，不能重复地下载同一个URL，必须要个方法避免加入URLs到集合中超过一次。一般的，达到避免可以用维护一个发现URLs的集合。如果数据太多，可以存入磁盘，或者储存经常被访问的URLs。  
3.CACHING  
 在大多数的计算机系统里面，内存是分等级的，意思是，存在2级或更多级的内存，表现出不同的空间和速度。举个例，在一个典型的工作站里，有一个非常小但是 非常快的内存，一个大，但是比较慢的RAM内存，一个非常大胆是很慢的disk内存。在一个网络环境中，也是分层的。Caching就是一种想法储存经常 用到的项目从慢速内存到快速内存。  
 Caching术语就像下面：cache是内存用来储存同等大小的元素。一个cache有k的大小，那么可以储存k个项目.在每个时间段,cache接受 到来自一个项目的请求.如果这个请求项目在这个cache中，这种情况将会引发一个碰撞并且不需要进一步的动作。另一方面，这种情况叫做 丢失或者失败。如果cache没有k个项目，那个丢失的项目被加入cache。另一方面，算法必须选择驱逐一个项目来空出空间来存放那个丢失的项目，或者 不加入那个丢失的项目。Caching算法的目标是最小化丢失的个数。  
 清楚的，cache越大，越容易避免丢失。因此，一个caching算法的性能要在看在一个给定大小的cache中的丢失率。  
 一般的，caching成功有2个原因：  
 不一致的请求。一些请求比其他一些请求多。  
时间相关性或地方的职权范围。  
3.1 无限cache(INFINITE)  
 这是一个理论的算法，假想这个cache的大小要大于明显的请求数。  
3.2 透视cache（MIN）  
 超过35年以前，L?aszl?o Belady表示如果能提前知道完整的请求序列，就能剔除下一个请求最远的项目。这个理论的算法叫MIN，因为他达到了最小的数量关于丢失在任何序列中，而且能带来一个飞跃性的性能提升。  
3.3 最近被用到（LRU）  
 LRU算法剔除最长时间没用被用到的项目。LRU的直觉是一个项目如果很久都没被用过，那么在将来它也会在很长时间里不被用到。  
 尽管有警告“过去的执行不能保证未来的结果”，实际上，LRU一般是非常有效的。但是，他需要维护一个关于请求的优先权队列。这个队列将会有一个时间浪费和空间浪费。  
3.4 CLOCK  
 CLOCK是一个非常流行的接近于LRU，被发明与20世纪60年代末。一个排列标记着M0，M1，….Mk对应那些项目在一个大小为k的cache中。 这个排列可以看作一个圈，第一个位置跟着最后一个位置。CLOCK控制指针对一个项目在cache中。当一个请求X到达，如果项目X在cache中，然后 他的标志打开。否则，关闭标记，知道一个未标记的位置被剔除然后用X置换。  
3.5 随机置换（RANDOM）  
 随机置换完全忽视过去。如果一个项目请求没在cache中，然后一个随机的项目将被从cache中剔除然后置换.  
 在大多数实际的情况下，随机替换比CLOCK要差，但并不是差很多。  
3.6 静态caching（STATIC）  
 如果我们假设每个项目有一个确定的固定的可能性被请求，独立的先前的访问历史，然后在任何时间一个撞击在大小为k的cache里的概率最大，如果一个cache中包含那k个项目有非常大的概率被请求。  
 有2个问题关于这个步骤：第一，一般这个概率不能被提前知道；第二，独立的请求，虽然理论上有吸引力，是对立的地方参考目前在大多数实际情况。  
 在我们的情况中，第一种情况可以被解决：我们可以猜想上次爬行发现的最常用的k个URLs适合于这次的爬行的最常用的k个URLs。（也有有效的技术可以 发现最常用的项目在一个流数据中。因此，一个在线的步骤可以运行的很好）当然，为了达到模拟的目的，我们可以首先忽略我们的输入，去确定那个k个最常用 URLs，然后预装这些URLs到我们做实验的cache中。  
 第二个情况是我们决定测试STATIC的很大的原因：如果STATIC运行的很好，Sname结论是这里有很少的位置被涉及。如果STATIC运行的相对差，那么我们可以推断我们的数据显然是真实被提及的位置，连续的请求是密切相关的。  
4 实验机制  
4.1 Meractor 爬虫体系  
 一个Meractor爬虫体系有一组爬虫进程组成，一般在不同的机器上运行。每个爬虫进程都是总网络服务器的子集，然后由一些工作线程组成（一般有500个），他们负责下载和处理网页从这些服务器。  
 举一个例子，每个工作现场在一个系统里用4个爬行进程。  
 每个工作现场重复地完成以下的操作：它获得一个URL从URL边境里，一个磁盘数据结构维护被下载的URL集合；用HTTP协议下载对应的网页到一个缓冲 区中；如果这个网页是HTML，提取所有的超链接。把提取出来的超链接流转换为完全链接然后运行通过URL过滤器，丢弃一些基于syntactic properties的链接。比如，它丢弃那些基于服务器联系我们，不能被爬行的链接。  
 URL流被送进主机Splitter里面，主机splitter用来分配URL给爬虫进程通过URL的主机名。直到大多数的超链接被关联，大部分的URL（在我们的实验中是81。5%）将被分配给本地爬虫进程；剩下的传说通过TCP给适当的爬虫进程。  
 本地URLs流和从爬虫中得到的URLs流都要送到复制URL消除器中（DUE）。DUE会除去那些被访问过的URLs。新的URLs会被送到URL边境中去以供以后下载。  
 为了避免重复URLs，DUE必须维护发现的URLs的集合。假设今天的网络包括几十亿有效的URLs，内存就需要维护这个集合是非常重要的。 Mercator可以被认为可以维护这个集合通过一个分布式的内存中的hash table（这个地方是每个爬虫进程维护URLs的子集时分配给它）；但是DUE执行（这个强制URLs成8-byte的checksums，而且用前3 —bytes来用作hash table的索引）需要大约5.2bytes 每个URl，意思就是它会用5GB的RAM每个爬虫机器来维护一个10亿个URLs的集合每台机器。这个内存需求非常不合理在很多的设置里，而且实际上， 它对于我们超过了硬件的适用性在这个实验里。因此，我们用一个选择性的DUE来执行那个缓冲器引入URLs到内存中，但是保存大多的URLs（或者更好， 他们的8-bytes checksum）到一个排序好的队列在磁盘中。  
基于磁盘的DUE和主机Splitter都受益于URL caching。给基于磁盘的DUE加一个cache可以使它丢弃引入的URLs，发生碰撞在cache中，替代加入他们到内存缓存区中。而且有个结果 是，内存缓存区要慢些，而且不频繁地和磁盘文件链接。将cache加入到一个主机Splitter中可以丢弃引入的重复的URLs代替将它们传入每个节 点，这样可以减少总的网络通信。这个通信的减少非常重要在爬虫进程没有通过高速LAN连接的时候，但是被替代成球形分布式。在这样一个装置中，每个爬虫负 责让web servers关掉它。 Mercator执行一个遍历通过广度优先算法在网络图上。每个爬虫进程中的线程同时执行。更重要的是，下载的行程被Mercator的 politeness policy调节，它限制每个爬虫的负载咋一些特殊的网络服务器中。Mercator的politeness policy保证没有服务器不断同时收到多个请求；而且，它还保证下一个请求会在它的上一个请求几倍的时间内完成（通常是10倍）。这样一个 politeness policy基本在任何一个大量搜索的网络爬虫中，否则爬虫将会陷入繁重的处理中。  
4.2 我们的网络爬虫  
 我们的网络爬虫由4个Compaq XP1000工作站组成，每个装备一个667MHz的Alpha processor，1.5GB的RAM，144GB的磁盘，和一个100Mbit/sec的以太网连接。每个机器定位于Palo Alto Internet Exchange,十分接近于Internet的backbone。  
 这个爬虫运行从7月12到2002年的9月3日，虽然它活跃地爬行只有33天。下载时由于不同的硬件和网络故障。爬行过程中，那4个机器完成了10.4亿 的下载尝试，7.84亿成功下载。4.29亿的成功下载文档是HTML页面。这些页面包含了大约268.3亿个超链接，相当于每个页面有62.55个超链 接；但是，中间的数值每个超链接只有24个，暗示平均的超链接数被一些包含很多链接的页面扩大了。早期的论文报道每个页面平均只有8个超链或者17个超链 接。我们提供了3个解释关于为什么我们每个页面找到了更过的超链接。首先，我们认为Mercator并没有限制发现URLs在anchor tags，但是更好的是提取所有的tags在可能包含他们的地方。第二，我们认为他下载页面一直16MB的大小（一个设置显著地大于平常），让它可能遇到 上万个的超链接页面.第三，大部分的论文报道那些每个页面中唯一的超链接。如果我们只考虑每个页面中唯一的超链接，那么平均值是47.74，而中间值为 17.  
 那些超链接从这些HTML中提取出来，加上大约3800万的HTTP跳转，在这个爬行中，流入到Host Splitter中。为了去测试不同caching算法的效率，我们通过Mercator的Host Splitter组件将所有的引入URLs打日志到磁盘中.四个爬虫中的Host Splitter接收并日志记录了总共268.6亿个URLs。  
 完成爬行后，我们浓缩了Host Splitter日志文件。我们把每个URL hash化为一个64—bit的识别码。我们确信没有故意的碰撞在排序最初的URL日志文档，而且计算了唯一的URLs个数。然后我们把这些唯一的URL 数和唯一的识别码数比较，我们决定用一个内存hash table在一个内存很大的机器里。数据减少的过程，大小距离51GB到57GB，而且包含64亿和71亿个URLs。  
 为了发现caching的效率，在一个分布式的爬虫程序里的交互的进程通信，我们也获得了一个日志文件，记录那些URLs被传给每个爬虫。这个日志文件包 含49.2亿个URLs，大约相当于全部URLs的19.5%。我们也浓缩了这个日志文件用同样的方式。然后我们会用这个浓缩的日志文件到我们的模拟中。