中图分类号：TP391

论文编号：10006ZY1306107

北京航空航天大學

硕 士 学 位 论 文

搜索引擎查询结果缓存技术研究

作 者 姓 名　　欧 韬

学 科 专 业　　计算机技术

指 导 教 师　　李 建 欣　副 教 授

王 丽 宏　教 授 级 高 工

培 养 学 院　　计 算 机 学 院

Research of Search Engine Query Results caching Technology

A Dissertation Submitted for the Degree of Master

Candidate：Ou Tao

Supervisor：Associate Prof. Li Jianxin

Prof. Wang Lihong

School of Computer Science & Engineering

Beihang University, Beijing, China

中图分类号：TP391

论文编号：10006 ZY1306107

硕 士 学 位 论 文

搜索引擎查询结果缓存技术研究

作者姓名	欧韬	申请学位级别	专业硕士
指导教师姓名	李建欣	职　　称	副 教 授
学科专业	计算机技术	研究方向	信息安全
学习时间自　　年　　月　　日		起至　　年　　月　　日	
论文提交日期　　年　　月　　日		论文答辩日期　　年　　月　　日	
学位授予单位	北京航空航天大学	学位授予日期	年　　月　　日

关于学位论文的独创性声明

本人郑重声明：所呈交的论文是本人在指导教师指导下独立进行研究工作所取得的成果，论文中有关资料和数据是实事求是的。尽我所知，除文中已经加以标注和致谢外，本论文不包含其他人已经发表或撰写的研究成果，也不包含本人或他人为获得北京航空航天大学或其它教育机构的学位或学历证书而使用过的材料。与我一同工作的同志对研究所做的任何贡献均已在论文中作出了明确的说明。

若有不实之处，本人愿意承担相关法律责任。

学位论文作者签名：_____ 日期：　　年　　月　　日

学位论文使用授权书

本人完全同意北京航空航天大学有权使用本学位论文（包括但不限于其印刷版和电子版），使用方式包括但不限于：保留学位论文，按规定向国家有关部门（机构）送交学位论文，以学术交流为目的赠送和交换学位论文，允许学位论文被查阅、借阅和复印，将学位论文的全部或部分内容编入有关数据库进行检索，采用影印、缩印或其他复制手段保存学位论文。

保密学位论文在解密后的使用授权同上。

学位论文作者签名：_____ 日期：　　年　　月　　日

指导教师签名：_____ 日期：　　年　　月　　日

摘　要

缓存查询结果是网络搜索引擎提高效率的关键技术。网络搜索引擎利用庞大的倒排索引在紧张的时延约束下处理用户查询。在这种条件下，搜索引擎系统需要庞大的基础设施，这包括上千台计算机和持续地投资维护这些设备。优化搜索引擎效率对网络搜索系统来说非常重要，即使微小的改进也可能为搜索引擎节省下巨额的设备费用。查询结果缓存技术广泛地被搜索引擎采用来提高搜索引擎效率,减少查询响应时间和降低搜索引擎后台负载。本文首先从两个角度进行研究。针对索引更新缓慢而容量受限的查询结果缓存研究缓存策略；针对缓存容量足够大的缓存研究面向实时检索的缓存更新策略。然后为了进一步提高效率，本文研究开销感知的缓存替换算法以达到节约搜索引擎后台处理资源的目的。

本文的主要工作如下：

(1)针对容量受限的缓存，本文为一个先进的查询结果缓存框架 SDC 选择动态缓存算法来达到更高的缓存命中率。将九种动态缓存算法分别作为 SDC 动态缓存部分的替换策略，在真实的搜索引擎日志下，比较它们应用于 SDC 的缓存命中率，时间、空间复杂度为选择合适的动态缓存策略提供依据。实验结果表明，采用 ARC 替换策略能够使 SDC 缓存达到最高的缓存命中率。

(2)随着存储技术的进步，存储硬件设备的容量不断增大、访问速度不断提高、价格越来越便宜，廉价的存储设备容量大到足够存储所有过去用户查询的结果, 而一个无穷大的缓存面临着缓存内容陈旧的问题。针对容量足够大的缓存，本文研究面向实时检索的缓存更新策略。

(3)为了进一步提高效率，本文提出一种开销感知的查询结果缓存替换算法。比 LRU 缓存替换算法减少了约 1.5%的总查询计算开销，且在缓存命中率方面仅比 LRU 缓存替换算法少了不到千分之一。

关键词：搜索引擎， 查询结果，缓存，查询开销

Abstract

Caching the query results is the key technology of network search engines to improve efficiency. Web search engines process user query using huge inverted index under the strain of timing constraint. In this condition, the search engine system requires a huge infrastructure, this includes thousands of computer and maintaining of computers. Search engine efficiency optimization is very important for web search system. Even small improvements can save huge equipment costs for search engine. The query results caching technology is widely used to improve the efficiency of search engine for reducing background load of the search engine and query response time. To solve above problems, this paper studied caching on the index updating slowly and limited capacity of query, caching on cache capacity is large enough as the cache update strategy for real-time search and cost-aware cache replacement policy.

In this paper, the main work is as follows:

(1) Choosing dynamic cache algorithm for an advanced results of the query cache framework SDC to achieve a higher cache hit ratio. Implement SDC with the nine dynamic cache algorithms respectively. This paper compare the cache hit ratio, time and space complexity under the real search engine log to provide the basis for choosing appropriate dynamic caching strategies. Experimental results show that SDC with ARC replacement policy achieve highest hit ratio.

(2) With the development of storage technology, storage capacity of the hardware is growing, the access speed is improving constantly, and the price is cheaper and cheaper. Cheap storage capacity is large enough to store all the user query results, and an infinite cache facing cache results stale problems. This paper studies the cache update strategies for real-time retrieval, to achieve the purpose of save search engine background processing resources.

(3) This paper proposed a novel query result cache replacement algorithm. The new algorithm reduces the computational overhead approximately 1.5% of the total, comparing to LRU cache replacement algorithm. The hit rate differences is less than 0.1%.

Keywords: Search engine, Query results, Cache， Query cost

目 录

图　目

表　目

第一章　绪论

1.1　研究背景

据中国互联网信息中心（CNNIC）研究报告[1]显示，截止到 2014 年 7 月，中国网民达到 6.32 亿，其中 Web 搜索引擎用户规模为 5.07 亿，网络搜索引擎使用率为 80.3%，搜索引擎已经是人们获取信息的主要工具。一方面，互联网网民增多，搜索引擎工作负载变大；另一方面，互联网网页也逐年增多，其规模已经超过 1500 亿，搜索引擎需要尽可能多地索引这些网页以提供准确和丰富的查询结果。在如此海量信息中，高效获取所需查询对学术界和工业界提出挑战。

缓存技术通过存放将来可能被请求的数据来提高数据访问速度。被存放的数据可能是计算结果或者数据的副本。如果请求的数据在缓存中被查找到，就直接获取该结果或者副本，否则，请求需要再次计算或者从其他位置获得该数据。如果更多的请求能够被缓存所满足，则意味着系统的整体效率会被提高。缓存被广泛应用于计算机系统及应用软件的各个层次，譬如 CPU 缓存、磁盘缓存、Web 缓存。

网络搜索引擎使用庞大的倒排索引在紧张的时延约束下处理用户查询，需要包含数千台计算机的大量设施和不停的投资来维护这些设施。优化网络搜索引擎效率对节省设备费用至关重要。即使微小的效率提高也可能为搜索引擎节约巨额的费用。最近的研究表明，缓存技术是优化搜索系统效率的关键技术。搜索引擎缓存给搜索系统带来令人满意的好处：它减少查询用户所察觉的时延，提升了用户体验。它减少后面查询处理器的负荷，提升了搜索系统的效率，进而可以减少搜索集群的数目，从而节省硬件资源，和降低能源消耗。缓存查询结果是大型信息检索系统和网络搜索引擎用来提高效率的关键技术。搜索引擎通常缓存查询结果为服务后继的查询。

随着 Web2.0 时代的到来，网民对信息的实时性有了更高的需求，如方便用户查找筛选新闻、微博等时效性高、更新速度快的文档。实时检索对搜索引擎缓存技术提出了新的挑战。搜索引擎为了更好地满足实时检索需求，需要频繁地更新倒排索引，容易使后台按索引计算的查询结果与缓存中的查询结果不一致，会引发搜索引擎缓存的查询结

果陈旧，使有效的缓存命中率下降。

传统的缓存替换策略面向相对固定的索引未考虑到倒排索引更新引发的搜索引擎缓存页面陈旧。因此，传统的缓存更新策略不能及时更新缓存内容使其与倒排索引一致，无法更好地适应实时检索系统。

针对以上问题，本文首先从两个角度进行研究，针对索引更新缓慢而容量受限的查询结果缓存研究缓存策略，针对缓存容量足够大的缓存研究面向实时检索的缓存更新策略。然后为了进一步提高效率，本文研究开销感知的缓存替换算法以达到节约搜索引擎后台处理资源的目的。

1.2 国内外研究现状

1.2.1 搜索引擎日志分析

搜索引擎用户查询日志记录了搜索引擎收到的用户查询请求信息，是研究和分析搜索引擎工作负载的重要载体，各大商业搜索引擎公司把查询日志作为重要战略资源。用户查询日志分析在查询推荐、用户相关反馈等研究领域被广泛采用。Cockburn[2]和Tauscher[3]等人对 Web 用户的浏览行为进行分析。Silversein[4]等人对商业搜索引擎的用户日志进行了大规模分析。Baeza-Yates[5]等人对英文搜索引擎 Excite, Vivisimo,Fast 的用户查询日志进行了分析。Xie[6]等人对西班牙文搜索引擎 TodoCL 的用户查询日志进行了分析。李亚楠[7]等人对中文商业搜索引擎用户查询日志进行了大规模分析和研究。马宏远[8]等人基于真实的中文搜索索引擎大量用户查询日志,针对搜索引擎缓存进行了研究。

1.2.2 面向有限容量缓存和固定索引的缓存技术研究

2000 年，Markatos[9]比较了 LRU 缓存替换策略及其若干变种和静态缓存策略的命中率，得出静态缓存策略仅适用于缓存容量较小的情况。在大容量缓存上的性能很差，操作系统里使用的基于替换策略如 lRU 或 LFU 的动态缓存能够达到相对更好的性能。可以认为静态缓存利用了用户查询分布的二八定律，将经常被用户发起的那 20%查询结果存入静态缓存。而 LRU 策略利用了用户查询的时间局部性。2006 年，Fagni[10]等人设计了一个双层缓存(Static-Dynamic Caching (SDC))，包括一个存储历史热门查询结果的静态缓存和一个采用替换策略的动态缓存。SDC 同时考虑到了查询的时间局部性和查询历史的统计特性。在 Altavista 日志上的实验表明，SDC 缓存达到了超过 30%的缓存命中

率。

2002 年，Xie[11]等人讨论了缓存位置的影响，分别讨论了客户端缓存、代理服务器缓存和搜索引擎服务器缓存技术。2003 年，Lempel 和 Moran[12]计算衡量查询在将来被发起概率的得分，这个技术被称为概率驱动缓存(Probability Driven Caching (PDC)).2007 年,Baeza-Yates[13]等人为搜索引擎结果缓存设计了许可策略(Admission Policies)，来阻止不频繁的查询通过替换策略获得频繁的查询在缓存中的空间，并在对用户查询日志的实验中得到了较好的缓存命中率。

查询结果并非唯一能可以被搜索引擎缓存的数据。2001 年，Saraiva[14]等人提出一种两级缓存方案，结合了缓存查询结果和缓存被频繁访问的倒排记录表。2006 年，Long 和 Suel[15]扩展了这一思想，同时缓存一对总是在查询中一起出现的词项倒排记录表的交集。2008 年，Baeza-Yates[16]等人看出，缓存存储倒排记录表可以比只存储查询结果和词项获得更高的缓存命中率。应当注意的是，在包含搜索引擎、倒排记录表缓存和查询结果缓存的大规模分布式系统中，它们可能不需要在同一台机器上竞争内存资源。2008 年，Skobeltsyn[17]等人的工作描述了一个 ResIn 体系，缓存查询结果和修剪过的索引。2008 年，Zhang[18]等人通过倒排表压缩结合缓存技术，取得了较好的缓存命中率。

以上的搜索引擎缓存技术研究对缓存记录活跃度、缓存位置、缓存数据类型等方面进行了研究，但没有考虑到搜索引擎索引更新对缓存内容有效性的影响。

1.2.3 面向无限容量缓存和持续更新索引的缓存技术研究

随着存储技术的进步，存储硬件设备的容量不断增大、访问速度不断提高、价格越来越便宜。与早期的工作聚焦于有限容量的缓存不同，越来越多的工作[19,20,21,22,23,24]假设结果缓存有无限的容量。Jonassen[19]等人认为廉价的存储设备容量大到足够存储所有过去用户查询的结果。在缓存容量无穷大的假设下，除了特定查询的第一次发起，其它的查询都能由缓存提供答案。然而，一个无穷大的缓存面临着缓存结果陈旧的问题。在倒排索引更新频繁，实时性要求高的搜索系统中，缓存结果的新鲜度极大地影响用户体验。

2010 年，Cambazoglu[21]等人提出一个新颖的算法，用生存时间值来设置缓存实体(cache entries)的过期时间并选择性地通过向搜索引擎后台发送更新查询请求来更新缓存结果，这个算法的一个实现目前被 yahoo 所采用。2010 年，Blanco[20]等人提出一种缓

存内容失效预测(Cache Invalidation Predictor CIP)方案。搜索引擎后台为增、删或修改的文档产生文档概要，并发送给缓存。缓存根据文档概要判断文档的修改是否引起缓存中某查询结果陈旧。2011 年,Alici[24]等人提出一种较为简单易于搜索引擎实现的缓存内容失效方案中，在文档被增、删和修改时仅需要相应调整倒排记录表和文档的版本时间戳。但在同等假正率的情况下未能达到采用 CIP 的缓存所达到的内容新鲜度。2012 年, Jonassen[19]等人提出线下和线上两种预取策略来更新缓存中可能陈旧的查询结果。2013 年，Sazoglu[23]等人考虑三种设置查询结果生存时间 TTL 的方式，基于时间的 TTL、基于频率的 TTL 和基于用户点击的 TTL，并考虑这三种方式的析取和合取。实验表明基于时间的 TTL 和基于频率的 TTL 以析取的方式结合能达到最好的性能。

1.2.4 查询开销在缓存技术中的应用

在以往的大多数研究中，所有的查询被假设需要相同的计算开销。从 CPU 处理时间、网络要求等方面来看，向搜索引擎提交的查询的成本开销差异很大。因此假设所有的缓存未命中带来同样的开销是不现实的。Cao[25]等人提出一种查询开销感知的代理服务器页缓存算法。Ozcan[26]等人基于查询开销提出一种多内容层次的缓存。Ismail[27]等人于 2009 提出一种查询开销感知的静态缓存策略。Ismail[27]等人于 2011 年讨论了多种查询开销感知缓存策略在查询结果缓存上的性能。

1.3 研究目标和内容

本论文的研究目标是通过研究搜索引擎查询结果缓存技术，提高搜索引擎查询结果缓存的命中率，提升搜索引擎缓存内容的新鲜度，减少搜索引擎处理查询的计算开销。因此，为实现上述研究目标，本文的主要内容包括以下三个方面：

本文的主要工作如下：

(1)针对容量受限的缓存，本文为一个先进的查询结果缓存框架 SDC 选择动态缓存算法来达到更高的缓存命中率。将九种动态缓存算法分别作为 SDC 动态缓存部分的替换策略，在真实的搜索引擎日志下，比较它们应用于 SDC 的缓存命中率，时间、空间复杂度为选择合适的动态缓存策略提供依据。实验结果表明，采用 ARC 替换策略能够使 SDC 缓存达到最高的缓存命中率。

(2)随着存储技术的进步，存储硬件设备的容量不断增大、访问速度不断提高、价格

越来越便宜，廉价的存储设备容量大到足够存储所有过去用户查询的结果，而一个无穷大的缓存面临着缓存内容陈旧的问题。针对容量足够大的缓存，本文研究面向实时检索的缓存更新策略。

(3)为了进一步提高效率，本文提出一种查询开销感知的查询结果缓存替换算法。

1.4 本文的组织结构

本文围绕搜索引擎查询结果缓存技术展开，结构如下：

第一章 绪论：介绍研究背景、国内外研究现状、研究目标和研究内容。

第二章 搜索引擎查询结果缓存的相关理论与技术：主要介绍了网络搜索引擎的体系结构、传统的针对有限容量缓存和相对固定倒排索引的缓存技术、针对更新频率的倒排索引的缓存技术、基于查询开销的缓存技术、用来预测查询请求重现时间的回归分析。

第三章 替换策略对 SDC 查询结果缓存的影响研究：介绍了九种缓存替换策略，并分别用这些缓存替换策略作为 SDC 动态缓存部分的替换策略。按真实的搜索引擎查询日志模拟用户查询，比较它们的性能。

第四章 基于历史查询的结果缓存预取策略研究：介绍了查询预取模块在搜索引擎缓存系统中的作用。通过周期性规律或回归预测查询发起时间的方法选择待预取的查询。然而后台的查询计算资源有限，本文依据查询开销和频率来选择更值得预取的查询结果。

第五章 一种开销感知的查询结果缓存替换算法：提出了一种带开销感知的查询结果缓存替换算法。在缓存容量变化的情况下始终能节约更多的查询处理时间，它的查询处理总耗时比 LRU 算法少约 1.5%。

最后是结论部分，对全文进行了总结，并展望下一步的工作。

第二章　相关理论与技术分析

本章将介绍本文研究相关的理论与技术。本章首先介绍带缓存的网络搜索引擎的体系结构，得出查询结果缓存的重要性大于索引缓存，文档缓存等，这是本文选择研究查询结果缓存的依据。本章还会介绍用户查询日志分析，日志里查询请求的规律是设计缓存算法的依据。本章介绍传统的查询结果缓存技术。索引更新频繁造成缓存陈旧与索引不一致而需要缓存更新技术。本章介绍缓存更新技术和查询开销感知的缓存技术的相关理论和预测查询请求时间需要用到的回归算法。

2.1　网络搜索引擎体系结构

在这一部分，本文给出先进的搜索引擎及其缓存的概要。如图 1 所示，搜索引擎通常由三类服务器组成：网络服务器，索引服务器和文档服务器。

网络服务器是搜索引擎与用户交互的终端。当它接收到一个查询 q 时，网络服务器负责检查 q 是否在查询结果缓存中。查询结果缓存维护一部分查询的查询结果。如果 q 的查询结果在查询结果缓存中被发现，网络服务器就直接向用户返回被存储的 q 的查询结果。通常来说，查询结果大致占用相同的缓存空间。一个查询结果是包括标题，URL 和与查询相关性排名前 k 的文档摘要。如果查询 q 的结果在查询结果缓存中未被发现，那个查询请求会被提交给索引服务器。

索引服务器负责计算出与查询 q 相关性排名前 k 的结果。索引服务器按查询 q 里的词项检索相应的倒排表来获得包括 q 中所有词项的文档 id，并根据一个排序模型将文档排序。最后将与查询 q 最相关的排名前 k 的文档发送给网络服务器。在索引服务器中，有一个在内存里的倒排表缓存被用来缓存一些词项的倒排记录表。当从网络服务器接到一个查询请求时，如果在倒排记录表缓存中发现了词项的倒排记录表，就跳过检索全体倒排索引的步骤。

文档服务器负责为网络服务器提供文档。网络服务器从索引服务器按收文档 id 列表，就会向文档服务器转发查询 q 和文档 id 的列表。文档服务器负责产生最后的结果。最后的结果是包括标题，URL,和排名前 k 的文档的摘要的一个网页。文档的摘要特定于一个查询，它是文档的中最能匹配查询 q 中词项的部分文本。在文档服务器上，先是检

索到文档 id 引用的原文档。然后，生成了每个文档对查询 q 的摘要。在文档服务器的 RAM 上，有两种缓存：文档摘要缓存和文档缓存。摘要缓存存放过去生成的与查询和文档相对应的文档摘要。如果一个特定的查询及文档 id 对在摘要缓存中被找到，那查询文档 id 对应的原文档这个步骤和生成查询及文档对应的文档摘要的步骤就可以被跳过。文档缓存存放一些被检索过的文档。如果被请求的文档 id 在文档缓存中被找到，那么通过文档 id 查找原文档的步骤就可以被跳过。文档服务器返回最终结果给网络服务器。最终结果的形式是一个网页，网页里是与查询 q 相关性排名前 k 的文档的摘要。网络服务器把最终查询结果存在查询结果缓存中并将最终结果返回给用户。

一旦查询请求命中查询结果缓存，就由查询结果缓存直接向用户返回查询结果而无需索引服务器和文档服务器的参与。因此，查询结果缓存对搜索引擎性能的影响远大于倒排表缓存，网页摘要缓存和文档缓存。

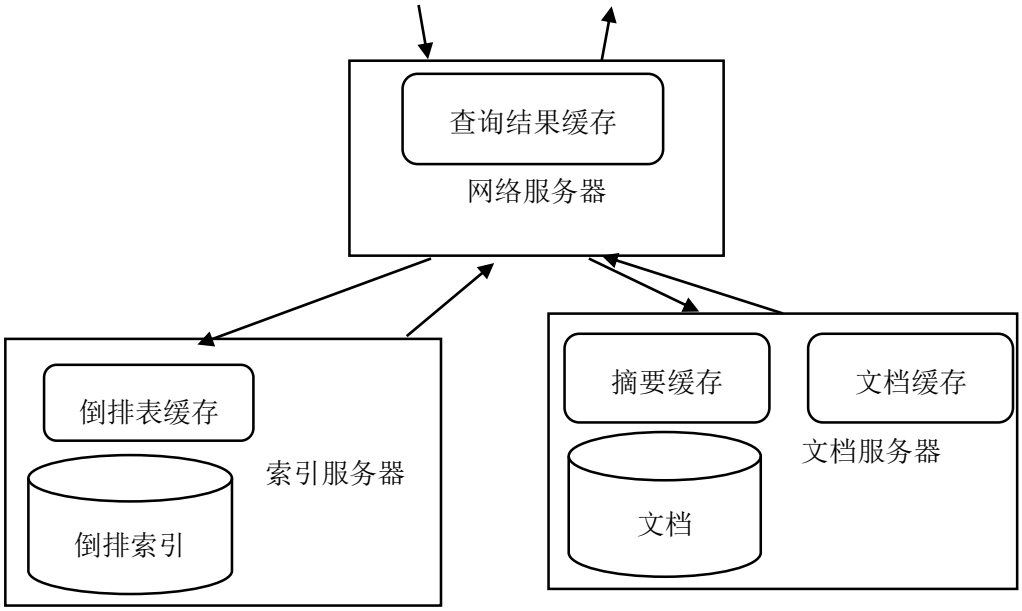

图 1 网络搜索引擎体系结构

2.2 搜索引擎用户查询日志分析

搜索引擎用户查询日志记录了搜索引擎收到的不同用户的查询请求信息，是研究和分析搜索引擎工作负载的重要载体，各大商业搜索引擎公司将查询日志作为重要战略资源。用户查询日志分析在查询推荐、用户相关反馈等研究领域被广泛采用。Cockburn[2]

和 Tauscher[3]等人对 Web 用户的浏览行为进行分析。Silversein[4]等人对商业搜索引擎的用户日志进行了大规模分析。Baeza-Yates[5]等人对英文搜索引擎 Excite,Vivisimo,Fast 的用户查询日志进行了分析。Xie[6]等人对西班牙文搜索引擎 TodoCL 的用户查询日志进行了分析。李亚楠[7]等人对中文商业搜索引擎用户查询日志进行了大规模分析和研究。马宏远等人[8]基于真实的中文搜索索引擎用户查询日志，针对搜索引擎查询结果缓存应该进行了研究。

对查询日志里用户查询的独一查询频率和查询复用距离的统计表明：查询频率符合 Zipf 定律。少数高频查询占据了大多数查询请求，在某时间段内搜狗的查询日志上的统计数据显示频率最高的 20%的独一查询占据了 80.45%的查询请求量，而 33.67%的独一查询只被提交了一次。平均查询重复次数为 3.82。用户查询还具有良好的局部性，多数查询的复用距离在 1 至 100 之间。

对查询日志里查询发起的时间进行统计发现：在一天之中，夜晚用户发起的查询量远少于白天的查询量。一天内的查询量与时间的关系大至符合正弦函数，在凌晨 4 点左右查询量达到最低值，在 15 点左右查询量到达最高值。在一周之中，周一的用户查询量最高，周二、三的查询量下降约 20%。

用户查询的特性能够被利用来优化搜索引擎的效率和资源分布。查询的重复规律能够作为设计缓存替换算法、缓存许可策略的依据。查询请求量较少的搜索引擎后台空闲时间段能够用来预取缓存内容,更新陈旧查询结果，从而提升有效的缓存命中率。

2.3　搜索引擎查询结果缓存技术

缓存作为有效减少搜索系统响应时间和系统负载的关键技术，被广泛应用于系统性能优化。Markatos[9]比较了 LRU 缓存替换策略及其若干变种和静态缓存策略的命中率，得出静态缓存策略仅适用于缓存容量较小的情况，而在大容量缓存上的性能很差，操作系统里使用的基于替换策略如 lRU 或 LFU 的动态缓存能够达到相对更好的性能。静态缓存利用了用户查询分布的二八定律,将经常被用户发起的那 20%查询结果存入静态缓存。而 LRU 策略利用了用户查询的时间局部性。Fagni[10]等人设计了一个双层缓存(Static-Dynamic Caching (SDC))，包括一个存储频繁查询的静态缓存和一个采用替换策略的动态缓存。SDC 同时考虑到了查询的时间局部性和查询历史的统计特性。在 Altavista 日

志上的实验表明，SDC 缓存达到了超过 30%的缓存命中率。

Xie 等人[11]讨论了缓存位置的影响，分别讨论了客户端缓存、代理服务器缓存和搜索引擎服务器缓存技术。Lempel 和 Moran[12]计算衡量查询在将来被发起概率的得分，这个技术被称为概率驱动缓存(Probability Driven Caching (PDC)). Baeza-Yates[13]等人为搜索引擎结果缓存设计了许可策略(Admission Policies)，来阻止不频繁的查询通过替换策略获得频繁的查询在缓存中的空间，并在对用户查询日志的实验中得到了较好的缓存命中率。

查询结果并非唯一能可以被搜索引擎缓存的数据。Saraiva 等人[14]提出一种两级缓存方案，结合了缓存查询结果和缓存被频繁访问的倒排记录表。Long 和 Suel[15]扩展了这一思想，同时缓存一对总是在查询中一起出现的词项倒排记录表的交集。Baeza-Yates[16]等人看出，缓存存储倒排记录表可以比只存储查询结果和词项获得更高的缓存命中率。应当注意的是，在包含搜索引擎、倒排记录表缓存和查询结果缓存的大规模分布式系统中，它们可能不需要在同一台机器上竞争内存资源。SKobeltsyn 等人[17]的工作描述了一个 ResIn 体系，缓存查询结果和修剪过的索引。Zhang 等人[18]通过倒排表压缩结合缓存技术，取得了较好的缓存命中率。

以上的搜索引擎缓存技术对缓存数据压缩、缓存位置、缓存记录活跃度等方面进行了研究，但没有考虑到搜索引擎索引更新对缓存内容有效性的影响。

2.4 面向实时检索的缓存更新方法

面向无限容量缓存和持续更新索引的缓存技术研究认为:廉价的存储设备容量大到足够存储所有过去用户查询的结果。在缓存容量无穷大的假设下，除了特定查询的第一次发起请求(compulsory missed)，其它的查询请求都能由缓存提供答案。一个无穷大的缓存面临的关键问题是缓存结果陈旧(stale)的问题。

在缓存更新技术中，被预测为陈旧且不久将被用户查询的缓存内容通过搜索引擎后台重新计算相关查询的结果而得到更新。这类技术利用搜索引擎后台的闲置周期（如深夜）来重新计算被猜测为结果已陈旧的查询。这类技术不需要结果缓存与搜索系统后台交互来决定哪些查询要更新。

然而，不使用后台提供的索引更新的具体信息，仅利用一些历史查询记录和查询特征来辨别查询是否陈旧和预测查询何时将再次被用户发起是一个相当困难的任务。这导致许多查询结果被不必要地重新计算而未对缓存新鲜度产生正面影响。

Cambazoglu 等人[21]提出一个新颖的算法，用生存时间值来设置缓存实体(cache entries)的过期时间并选择性地通过向搜索引擎后台发送更新查询来更新缓存结果。这个启发式算法结合了查询的频率和查询结果在缓存里的时间来给需要被更新的缓存实体排序。此外，通过设置查询更新率展示了一种将搜索引擎后台闲置周期考虑进去的机制。实际后台工作负载表明，这个算法不仅提高了缓存命中率，也减少了缓存结果的平均年龄。该算法的一个实现目前被 yahoo 所使用。

Jonassen 等人[19]提出线下和线上两种预取策略来更新缓存中可能陈旧的查询结果。线下预取策略,过去的研究[28]得出许多查询会在上次提交后的 24 小时左右再次被发起。线上预取策略利用查询的频率，查询重复发起的时间间隔，查询词数等属性用机器学习的方法来预测查询再次被发起的时间。由此确定更新特定查询缓存结果的时机。

线下预取策略利用历史查询记录来决定更新的查询。某些查询会在离上次被发起的固定一段时间 dt 后被再次发起。如，某些美剧一周更新一次，这使得许多观众会间隔一周的时间去搜索相应美剧的名字。将缓存里在与当前时刻相距 dt 的时刻被发起的查询放入等待更新的队列里，并按一定的度量计算其更新权重。根据当前搜索引擎后台负载更新队列更新权重较高的查询。

线上预取策略以历史查询记录为训练集用机器学习的方法来预测缓存里查询再次被发起的时间间隔,选择预测最准确的方法对发起时间间隔进行预测。进而求得缓存里每个查询 q 再次被发起的时刻 t。如果当前 q 的结果已陈旧，且当前时刻 t0 满足 t-t0>ttl 则将查询放入等待更新的队列里。

在缓存内容失效技术中，索引系统向结果缓存传递关于最近倒排索引的更新信息。这些信息被缓存用来辨别可能陈旧的查询结果。搜索结果缓存模块对缓存中的每个查询判断索引的变化是否会导致这个查询的结果陈旧，并将查询标记为失效，得到它的缓存结果视为未命中缓存。

与缓存更新相比，缓存模块与索引系统的信息交互将占用更多的资源。Blanco 等人[20]提出一种缓存内容失效预测(Cache Invalidation Predictor CIP)方案。搜索引擎后台为增、删或修改的文档产生文档概要，并发送给缓存。缓存根据文档概要判断文档的修改是否引起缓存中某查询结果陈旧。Alici 等人[24]提出一种较为简单易于搜索引擎实现的缓存内容失效方案中，在文档被增、删和修改时仅需要相应调整倒排记录表和文档的版本时间戳。但在同等假正率的情况下未能达到采用CIP的缓存所达到的内容新鲜度。Bai 等人[22]提出了一个查询驱动的缓存内容失效框架，失效决定只在有查询被发起且查询结果在缓存中时发生。它将索引的更新信息保存在子索引上来为失效决定提供依据。

事实上，缓存中的查询结果就是对给定查询文档得分排名前 K 的文档(链接)。通过估计被更改的文档对特定查询的得分，能大致预测此文档的改变是否影响查询的结果。文档纲要生成模块尝试发送相应文档的压缩表示，能由文档纲要大致估计对任意查询的文档得分。它的主要输出是由文档中 TF-IDF 权重高的词项组成的集合，文档可能因为这些词项而获得较高的查询文档得分。为了控制文档纲要的长度，文档生成模块只发送一定比率 n(0<=n<=1)的查询到集合中。较短的文档纲要能降低通信量，但会增加缓存内容失效预测的错误率。文档微小的修改通常不会影响文档的得分排名。因此，需要比较文档修改前后的差异，仅对差异大于阀值 di 的文档生成纲要。采用 Jaccard 距离来衡量文档修改前后的差异。增加 di 能使该模块生成更少的文档纲要，进而减少通信量，但会降低缓存内容陈旧预测的准确率。大多数情况下，一个文档的更新只可能影响到它匹配的查询。缓存内容失效模块接收到文档纲要之后，让所有缓存里的查询与文档纲要在合取模式下进行匹配。当文档纲要匹配上一个查询时，我们计算文档纲要对这个查询的文档得分。如果文档得分超过了缓存结果中排名最后的文档的得分。我们就认为文档的这次更新将会改变这条查询的结果，所以使这条查询的结果失效。

在上述两类技术中，陈旧决策由启发式算法得到而不能产生完美的准确度。某些查询结果陈旧了，却可能没有被及时鉴别出来。这样的查询的陈旧结果可能在缓存中存在很长一段时间。上述两类技术依靠补充机制：生存时间(Time-to-Live TTL)，来补救这个问题。

Sazoglu[23]等人考虑三种设置 TTL 的方式，基于时间的 TTL、基于频率的 TTL 和基

于用户点击的 TTL，并考虑这三种方式的析取和合取。通过实验表明基于时间的 TTL 和基于频率的 TTL 以析取的方式结合能达到最好的性能。

2.5　基于查询开销的缓存技术

在网络搜索引擎背景下，许多文献涉及缓存内容和如何缓存的研究。然而，缓存未命中的代价差异通常被忽略，所有的查询被假设需要相同的计算开销。Ismail Sengor Altingovde[27]等人专注于静态查询结果缓存提出一种查询开销感知的策略在决定缓存内容时考虑查询开销因素。从 CPU 处理时间、网络要求等方面来看，向搜索引擎提交的查询请求的计算成本开销差异很大。因此假设所有的缓存未命中带来同样的开销是不现实的。其次，频率并不总能精确地代表计算开销，因此用高频流行的查询或最近使用的查询来填充缓存并不总是带来最优的性能，一个查询开销感知的策略可能带来更多的收益。

对于一个查询 q,假设它的查询成本为 C(q), 将来查询频率为 f(q)，历史查询频率 F(q)。将查询放入静态缓存中的收益可以由 C(q)和 f(q)来计算为 C(q)*f(q)。研究表明，将来查询频率与历史查询频率并非线性相关。历史查询出现频率高的通常在将来仍然以高频率出现然而历史查询频率低的查询很可能出现得更少甚至在将来不会出现。因此,令 f(q)=F(q)的 k 次方，k>=1 来加强高频查询。修正收益期望为$C(q) * F^k(q)$。

2011 年，Rifat Ozcan 与 Ismail Sengor Altingovde[29]等人提出了 LFCU 缓存替换策略在 LFU 上加入时间开销感知，获得比 LFU 更好的性能。然而 LFU 本身应用于查询结果缓存的性能比 LRU 差，而改进的 LFCU 缓存在节约查询开销方面与 LRU 相差不大，在缓存容量较小时性能不如 LRU,在缓存容量大时性能略优于 LRU。

通常频繁磁盘 IO 与占用 CPU 资源多的操作是影响系统性能的关键因素，经分析搜索引擎查询检索过程，搜索引擎查询受如下因素影响：(1)从磁盘获取查询词项对应的倒排表，记作 Cost(list);(2)进行查询词项之间的倒排表求交集，记作 Cost(intersection);(3)从磁盘获取文档，记作 Cost(document);(4)生成文档摘要,记作 Cost(snippet)。不同缓存类型的节省时间开销如表 1 所示，查询结果缓存节省的查询开销高于倒排表缓存、文档摘要缓存和文档缓存，这也是本文研究查询结果缓存算法的原因。Ozcan[26]等人基于查询开销提出一种多内容层次的缓存，根据节省的开销决定缓存的内容，是缓存查询结果，缓

存查询用到的倒排表，缓存查询用到的文档摘要，还是缓存查询用到的文档。文档对查询的得分和倒排表的交集也作为缓存的内容。然而查询结果、倒排表、文档摘要和文档的缓存相互影响其节省的查询开销，Ozcan[26]等人提出多内容层次缓存方案属于贪心算法，只能在局部节省最多的查询开销。这种贪婪地启发式填充多内容层次缓存的方法保持多个队列，每个缓存内容类型一个队列，并根据缓存收益来决定特定缓存类型的优先级。这种启发式的方法分两个步骤。在第一步中，缓存各内容项目的初始收益被计算出来并插入相应的队列。在第二步中，拥有最高收益的缓存内容项目被选出来，并对当前查询请求缓存其收益最高的内容，然后以迭代的方式更新缓存各内容项目的收益。

表 1　不同缓存类型的节省的开销

缓存类型	缓存内容	节省查询开销
查询结果缓存	查询结果页面	Cost(list)+ Cost(intersection) + Cost(document) + Cost(snippet)
倒排表缓存	词项倒排表	Cost(list)
文档摘要缓存	文档和查询对应的文档摘要	Cost(snippet)+Cost(document)
文档缓存	文档	Cost(document)

2.6　回归分析技术

回归问题与分类问题一样属于有监督学习的范畴。回归问题的目标是给定多维输入矢量 x，并且训练模型的样本中每一个输入变量 x 都有其对应的目标值 y，要求对新来的未标注数据预测它对应的连续的目标值 t。回归分析反映了数据属性值之间的关系，通过函数表达数据映射关系来发现数据属性值之间的依赖关系。Python scikit-learn 库中提供的回归算法有 gradient boosting regression ,SVR, ridge regression, Lasso, Nearest Neighbor Regression, Bayesian Regression 等。本文考虑过 gradient boosting regression ,SVR, Nearest Neighbor Regression 和 Bayesian Regression 来预测第四章的查询下次发起时间间隔。然而，SVR 和 Bayesian Regression 消耗的时间太多，Nearest Neighbor 算法消耗的内存过大而不适用于本文环境下的实验，本文采用 gradient boosting regression 来预测第四章的查询下次发起时间间隔。

梯度树提升或梯度提升回归树（GBRT）是提升任意可微损失函数的推广。GBRT 有准确和有效的现成的程序，可用于回归和分类问题。梯度提升树模型被应用于许多领域，包括网络搜索排名和生态。GBRT 的优点包括：非均匀的混合数据类型的自然处理，预测能力，在输出空间中对离群值的鲁棒性（通过强大的损失函数）。GBRT 的缺点是：可扩展性较差，由于梯度软提升的序列性质而不能并行。

Boosting 方法是用来提高弱分类算法准确度的一种方法，这种方法通过构造一系列预测函数，然后以一定的方式将他们组合成一个新的预测函数。它是一种框架算法，主要通过对样本集的抽样操作获得样本子集，然后在样本子集上用弱分类算法训练生成一系列基分类器。它可以用来提高其他弱分类算法的准确度，也就是将其他弱分类算法作为基分类算法放在 Boosting 框架中，通过 Boosting 框架对训练样本集的抽样,得到不同的训练样本子集，用该样本子集去训练生成基分类器。每得到一个样本集就在该样本集上用该基分类算法产生一个基分类器。这样在给定的训练轮数 n 后，可产生 n 个基分类器，然后用 Boosting 框架算法将这 n 个基分类器加权融合，最后产生一个结果分类器。在这 n 个基分类器中，每个单独的分类器的识别率不高,但他们联合后的结果可以有很高的识别率,这样就提高了弱分类算法识别率。Boosting 算法对一份数据，建立多个模型，这种模型一般比较简单，称为弱学习者(weak learner)每次计算都将上一次算错的数据权重提高一点再进行计算，这样最终得到的模型在测试集与训练集上都可以得到较好的成绩。

Gradient Boosting 是 Boosting 的一种方法，它的主要思想是，每次建立模型在之前建立模型的损失函数梯度下降方向上。损失函数(loss function)描述是用来模型的不靠谱程度的，损失函数值越大，则说明该模型越容易出错。如果模型能够让损失函数持续下降，则说明模型在不断的改进，而最好的方式就是让损失函数沿着其梯度（Gradient)方向下降。

2.7　本章小结

本章首先介绍了网络搜索引擎体系结构。搜索引擎通常由三类服务器组成：网络服务器，索引服务器和文档服务器。网络服务器的 RAM 上有查询结果缓存；索引服务器 RAM 上有倒排表缓存；文档服务器上有网页摘要缓存和文档缓存。其中一旦查询请求

命中查询结果缓存，就由查询结果缓存直接向用户返回查询结果而无需索引服务器和文档服务器的参与。查询结果缓存对搜索引擎性能的影响远大于倒排表缓存，网页摘要缓存和文档缓存。这也是本文研究查询结果缓存的原因。本章然后介绍了搜索引擎用户查询日志分析，用户查询的特性能够被利用来优化搜索引擎的效率和资源分布。查询的重复规律能够作为设计缓存替换策略、许可策略的依据。查询量较少的搜索引擎后台空闲时间能够用来预取缓存内容和更新陈旧查询结果，提升缓存命中率。本章还介绍了搜索引擎查询结果缓存技术的现状，由静态缓存，动态缓存发展到静态缓存和动态缓存相结合，概率驱动缓存和多层次的缓存。本章介绍了面向实时检索的缓存更新方法。包括缓存更新技术，缓存内容失效技术和生存时间机制。本章介绍了基于查询开销的缓存技术。包括查询开销的静态缓存策略，查询开销感知的 LFU 算法改进和将查询开销分解成计算查询结果各步骤开销进而演化出多内容层次缓存框架。最后，本章介绍了回归分析技术，主要是本文第四章使用的 gradient boosting regression 方法。

第三章　替换策略对 SDC 查询结果缓存的影响研究

本章为一个先进的查询结果缓存框架 SDC 选择动态缓存算法来达到更高的缓存命中率。本章介绍九种缓存替换算法并将这九种缓存替换算法分别作为 SDC 的动态缓存替换策略，在真实的搜索引擎日志下，比较它们应用于 SDC 的缓存命中率，时间、空间复杂度，为选择合适的动态缓存策略提供依据。实验结果表明，采用 ARC 替换策略能够使 SDC 缓存达到最高的缓存命中率。

缓存被用在计算机的的各个方面，如存储系统，数据库，网络服务器，处理器，文件系统，磁盘驱动，操作系统，数据压缩等。缓存通过存放将来可能被请求的数据来提高数据访问速度。被存放的数据可能是计算结果或者数据的副本。如果请求的数据在缓存中被查找到，就直接获取该结果或者副本，否则，请求需要再次计算或者从其他位置获得该数据。研究者提出了许多缓存算法,如:CLOCK,LIRS ,ARC，以提升缓存命中率，使更多的请求在缓存中被查找到，从而提高缓存作用。

查询结果缓存是大型信息检索系统和网络搜索引擎用来提高效率的关键技术。搜索引擎通常缓存查询结果为服务后继的查询。Fagni等[30]人 2006年提出一种混合的缓存策略：SDC，包括静态缓存和动态缓存。SDC从历史数据中提取提交频率最高的查询的结果并将它们存储在静态，只读的一部分缓存中；缓存的其他部分存储不在静态缓存部分的查询的结果，并由缓存替换算法动态管理。在三种查询日志上的实验表明，它比纯粹的静态缓存或动态缓存达到更好的性能。

Fagni 等人[30]没有详尽地讨论动态缓存替换算法的选择对缓存命中率的影响。本文讨论 SDC 动态部分的缓存替换算法的选取。通过在真实查询日志上的实验，和对缓存算法的时间复杂性的分析为选择动态缓存算法提供依据。

缓存替换算法最大限度地利用最近访问和引用频率的信息来最大化缓存命中率。Fagni 等人[30]指出:SDC 简化了采取的缓存替换算法的选取。静态的只读缓存固定存储过去引用频率最高的查询，这使得最近使用信息成为我们要考虑的最重要的参数。在接下来的实验中，我们仍然会尝试利用引用频率信息的缓存替换算法，如 LFU 来证实 Fagni 等人[30]的观点。

3.1 SDC 查询结果缓存框架

Fagni[30]等人 2006 年提出一种混合的缓存策略：SDC，包括静态缓存和动态缓存。SDC 从历史数据中提取提交频率最高的查询的结果并将它们存储在静态，只读的一部分缓存中；缓存的其他部分存储不在静态缓存部分的查询的结果，并由缓存替换算法动态管理。在三种查询日志上的实验表明，它比纯粹的静态缓存或动态缓存达到更好的性能。本文讨论 SDC 动态部分的缓存替换算法的选取。通过在真实查询日志上的实验，和对缓存算法的时间复杂性的分析为选择动态缓存算法提供依据。

3.2 动态缓存替换策略

SDC 缓存，包括静态缓存和动态缓存部分。本文分别实现了九种缓存替换策略作为 SDC 查询结果缓存的动态缓存替换算法。其中 LRU, LIRS, KRANDOM, CLOCK 算法用最近使用信息来估计未来的数据请求。LFU 用频率信息来估计未来的数据请求。ARC 和 SKLRU 同时考虑了最近使用信息和使用频率信息。OPT 替换算法假设已知未来的请求序列，提供一个在特定请求序列和缓存容量的情况下，缓存命中率能达到的上界。而 RANDOM 替换算法，在缓存满时随机地移出一个缓存中的实例，提供一个可以接受的缓存命中率下界。

本文在下面将给出实验中九种缓存替换算法的时间复杂度、空间复杂度和基本流程。然而同一算法的时间复杂度也与其实现有关。如：2010 年,Prof. Ketan Shah 等人提出基本频率桶的 LFU 算法实现，使单次查询的计算复杂度降至 O(1),n 次请求的时间复杂度为 O(n)。在本文的实验中，LFU 仍利用堆来实现，时间复杂度为 O(n*log(m)) ,其中 n 为请求次数,m 为缓存大小。多数替换算法时间复杂度为 O(n)，而时间复杂度高的替换算法也仅比 O(n)多一个对数因子,难以区别它们的查询速度。因此，在实验中我们测量了在相同查询请求下，各替换算法的运行时间来区别它们的查询速度。缓存替换算法实现了 get 方法和 put 方法。其中 get 方法在缓存接到请求时更新记录最近使用信息和频率信息的数据结构，并返回查询是否命中缓存的情况。而 put 方法在缓存需要加入新元素时，更新相应的数据结构并决定被移出的元素。

3.2.1 OPT 缓存替换策略

OPT 算法需要预先已知请求序列。时间复杂度为 O(n*log(m)) ,n 为请求次数,m 为缓存大小。空间复杂度为 O(n)。当一个请求到来时，查询这个请求是否在缓存中。若在缓存中，则使用缓存中的结果。若这个请求不在缓存中，则从外存中读或计算此请求的结果 R。若缓存未满，将此请求结果 R 放入缓存。若缓存已满，根据请求序列选择缓存中下次被使用时间最晚的请求，将下次被使用时间最晚的请求的查询结果逐出缓存；将 R 放入缓存。

3.2.2 ARC 缓存替换策略

ARC 算法能够动态地调整一级 lru 缓存和二级 lru 缓存的空间。采用 4 个 lru 链表，包括一个一次频率的 lru 的链表，一个 2 次以上频率查询的 lru 链表和 2 个虚拟链表,虚拟链表只存查询不存查询的结果。它的时间复杂度和空间复杂度都是 O(n)。设 x 被请求的页，缓存总空间大小为 c。用参数 p 来衡量 T1 的空间大小趋势。T1 是一级 lru 缓存表，T2 是二级 lru 缓存表，B1 是虚拟的一级缓存表，B2 是虚拟的二级缓存表。L1 由 T1

初始化 T1，B1，T2，B2 为空集, p= 0.

Case I. x ∈T1 ∪T2 (x 命中了缓存容量为 c 的 ARC 缓存): 把 x 移至 T2 的顶端。

Case I I. x ∈B1 (x 未命中缓存容量为 c 的 ARC 缓存):

将 p 调整为 min{c,p+ max{|B2|/ |B1|, 1} } . REPLACE(p).

把 x 移至 T2 的顶端并把它的查询结果放进缓存中。

Case I I I. x ∈B2 (x 未命中缓存容量为 c 的 ARC 缓存):

将 p 调整为 max{0,p-max{ |B1|/ |B2|, 1} } . REPLACE(p).

把 x 移至 T2 的顶端并把它的查询结果放进缓存中。

Case IV. x ∉L1 ∪L2 (x 未命中缓存容量为 c 的 ARC 缓存):

case (i) |L1| = c:

If |T1| < c then 从 B1 尾部来删除一个页并 REPLACE(p).

else 从 T1 尾部删除最近最少使用的页，并将它的内容移出缓存。

case (ii) |L1| < c and |L1| + |L2| ≥ c:

if |L1| + |L2| = 2c then 从 B2 尾部删除其最近最少使用的页。

REPLACE(p).

将 x 放入 T1 的顶端，并把 x 的结果放进缓存中。

子过程 REPLACE(p):

if (|T1| ≥1) and ((x ∈B2 and |T1| = p) or (|T1| > p)) 那么将 T1 尾部最近最少使用的页放入 B1 顶端并将它的内容从缓存中删除。

else 将 T2 尾部最近最少使用的页放入 B2 顶端并将它的内容从缓存中删除。

图 2 ARC 算法

和 B1 组成，L2 由 T2 和 B2 组成。那么它的基本流程如图 2 所示。

3.2.3 LRU 缓存替换策略

LRU 算法用最近使用情况来预测将来缓存页的使用情况，许多缓存算法在它的基础上进行了改进。它的时间复杂度和空间复杂度都是 O(n)。当一个请求到来时，查询这

个请求是否在缓存中。若在缓存中，则使用缓存中的结果。并把这个被命中的请求及其结果移动链表头部。若这个请求不在缓存中，则从外存中读或计算此请求的结果 R。若缓存未满，直接将此请求结果 R 放入缓存链表头部。若缓存已满，将链表尾部的请求及其结果逐出缓存，腾出空间再将 R 放入缓存块链表头部。

3.2.4 CLOCK 缓存替换策略

CLOCK 算法使用一个标志位来记录缓存块的最近使用情况。它的时间和空间复杂度都是 O(n)。当一个请求到来时，查询这个请求是否在缓存中。若在缓存中，则使用缓存中的结果。若这个请求不在缓存中，则从外存中读或计算此请求的结果 R。若缓存未满，将此请求结果 R 放入环形缓存链表 hand 指针处，并置标志位为 1,hand 指针向前移动一步。若缓存已满，使环形缓存表的 hand 指针向前移动寻找标志位为 0 的缓存块。把经过的标志位为 1 的缓存块标志位置 0，把找到的标志位为 0 的的请求及其结果逐出缓存；将 R 放入环形缓存链表 hand 指针处，并置标志位为 1，hand 指针向前移动一步。

3.2.5 KRANDOM 缓存替换策略

KRANDOM 算法使用随机函数，缓存块被命中与否不确定，它考虑了最近使用信息，通过增加 k 值能接近于 LRU，需要存储缓存块最近被使用的时间。KRANDOM 算法的时间复杂度和空间复杂度都是 O(n)。当一个请求到来时，查询这个请求是否在缓存中。若在缓存中，则使用缓存中的结果。并更新它的最近访问时间记录为当前时间。若这个请求不在缓存中，则从外存中读或计算此请求的结果 R。若缓存未满，将此请求结果 R 放入缓存并设置它的最近访问时间为当前时间。若缓存已满，随机取三个缓存块，比较它们的最近访问时间，并将最近访问时间最早的请求逐出缓存；将 R 放入缓存并设置它的最近访问时间为当前时间。

3.2.6 SKLRU 缓存替换策略

SKLRU 算法采用 K 个固定大小的 LRU 缓存表，分别存放最近被使用 1，2，…k 次的查询结果。SKLRU 算法的时间复杂度和空间复杂度都是 O(n)。当一个请求到来时，查询这个请求是否在缓存中。若在缓存中，则使用缓存中的结果。并尝试把这个被命中的请求及其结果移动到更高一级链表头部（如果是在最高级 LRU 链表中命中则移动到最高级链表头部）。若要在 LRU 中加入请求结果时发现此 LRU 链表已满，则移出本层

LRU 队列尾部元素并将它放到下一级队列头部,如果已经是第一层队列则直接移除。若这个请求不在缓存中，则从外存中读或计算此请求的结果 R。若缓存第一层 LRU 链表未满，将此请求结果 R 放入第一层缓存链表头部。若缓存第一层队列已满，将链表尾部的请求的查询结果逐出缓存，将 R 放入缓存块链表头部。

3.2.7 RANDOM 缓存替换策略

RANDOM 算法并不考虑缓存块的最近使用情况情况或使用频率信息。它使用随机函数决定被替换出的块，每次运行缓存块被命中与否不确定。RANDOM 算法的时间复杂度和空间复杂度都是 O(n)。当一个请求到来时，查询这个请求是否在缓存中。若在缓存中，则使用缓存中的结果。若这个请求不在缓存中，则从外存中读或计算此请求的结果 R。若缓存未满，将此请求结果 R 放入缓存。若缓存已满，则随机地将一个缓存中的请求的查询结果逐出缓存并将 R 放入缓存。

3.2.8 LIRS 缓存替换策略

LIRS 缓存算法采用最近复用信息 IRR 来预测缓存块下次被使用的时间。它的时间复杂度和空间复杂度都是 O(n)。它将缓存块分成低最近复用距离缓存块 LIR 和高最近复用距离块 HIR。将大部分的缓存空间用来存放低最近复用距离块 LIR。用 p 表示 HIR 缓存块占用的总缓存空间比例，本文中 p 取 0.01。用 x 表示要被使用的块，S 表示 LIRS 队列，c 表示缓存的容量大小。Q 容纳所有驻留在缓存中的 HIR 块，用 x.flag 来表示 x 是 HIR 块还是 LIR 块。Q 中最多容纳的 HIR 块数量是 c*p。LIRS 算法如图 3 所示。

初始化 Q 和 S 为空集。

Case I. x∈S and　x.flag = LIR (命中了 LIRS 缓存):

　　将 x 移至 S 的顶端。STACKPRUNING.

Case II. x∈S and　x.flag = HIR and x∈Q (命中了 LIRS 缓存):

　　将 x 移至 S 的顶端。将 x.flag 改为 LIR 并从 Q 中删除。

　　将 S 底部的元素 y 移除。将 y 加入 Q 的末端并设 y.flag 为 HIR。

　　STACKPRUNING.

Case III. x∉S and x∈Q (命中了 LIRS 缓存):

　　将 x 移至 S 的顶端。在 Q 中将 x 移动至末端。

Case IV. x∈S and　x.flag = HIR and x∉Q (未命中了 LIRS 缓存):

　　将 Q 头部的元素移除。将 x 移至 S 的顶端。把 x.flag 改为 LIR.

　　将 S 底部的元素 y 移除。将 y 加入 Q 的末端并设 y.flag 为 HIR。

　　STACKPRUNING.

Case V. x∉S and x∉Q (未命中了 LIRS 缓存):

　　将 Q 头部的元素移除。将 x 加入 S 的顶端。将 x 放入 Q 的末端。

子过程 STACKPRUNING():

　　while S 的底部元素.flag != LIR:

　　　　移除 S 的底部元素。

图 3 LIRS 算法

3.2.9 LFU 缓存替换策略

LFU 算法考虑缓存块的历史被使用频率来预测将来缓存块的使用情况。它的时间复杂度是 O(n*log(m)) ,其中 n 为请求次数,m 为缓存大小，空间复杂度是 O(n)。当一个请求到来时，查询这个请求是否在缓存中。若在缓存中，则使用缓存中的结果。这个请求的频率记录自增 1,按频率调整此请求小根堆中的位置。若这个请求不在缓存中，则从外存中读或计算此请求的结果 R。若缓存未满，直接将此请求结果 R 放入缓存。这个请求的频率设置为 1,按频率调整此请求小根堆中的位置。若缓存已满，将堆顶访问频率最小的请求的查询结果逐出缓存；将此请求结果 R 放入缓存。这个请求的频率设置为 1,按频率调整此请求小根堆中的位置。

3.3 查询日志分析

为了评估缓存替换算法的表现，我们使用两份查询日志。中国搜狗搜索引擎的查询日志，包含约 43 545 444 条查询的三天查询记录，从 2011.12.30 到 2012.01.01; 美国 aol 查询日志，包含来自约 657 426 个用户在三个月里，从 01 March, 2006 到 31 May, 2006 的约 36 389 567 条查询。在本文中，将检索词相同的查询视为相同的查询而不考虑查询结果的页码。搜狗的查询日志有 8 935 490 条独一查询，在 aol 查询日志里除去查询词为'---'的异常查询，还剩 35 389 192 条查询记录，其中 9 687 128 条独一查询。独一查询的第一次查询请求一定不会命中缓存。在搜狗查询中约 79.5%的查询请求是之前已经被发起过。而在 aol 日志中仅 72.6%的查询是重复之前的查询。因此，搜狗的查询请求比 aol 具有更多的重复查询，更能体现缓存的性能。

我们分析了查询出现的频率分布，图 4 在对数-对数坐标下，画出了每个日志里独一查询出现的次数，而查询按频率降序排列并由查询的排名来标识。注意到两个日志里的查询的分布都服从幂律。可以看出搜狗日志的查询出现频率略高于 aol 日志的查询频率从而具有更好的大范围内的重复性。

我们同样分析相同查询被相继提交的距离，图 5 在对数-对数坐标下，描绘了每个日志里查询被重复提交的距离及其出现次数。如果查询在较短的时间内重复出现，它更可能在较小容量的缓存中被检索命中缓存。可以从图 5 看出在 aol 日志的查询复用距离小的查询比搜狗日志的多。可以推断出在 aol 日志在较小容量的缓存上的缓存命中率高于

搜狗查询日志。

从对搜狗和 aol 查询日志分析来看,搜狗查询日志在大范围上有更好的重复性，而 aol 查询日志在小范围上有着更好的局部性。

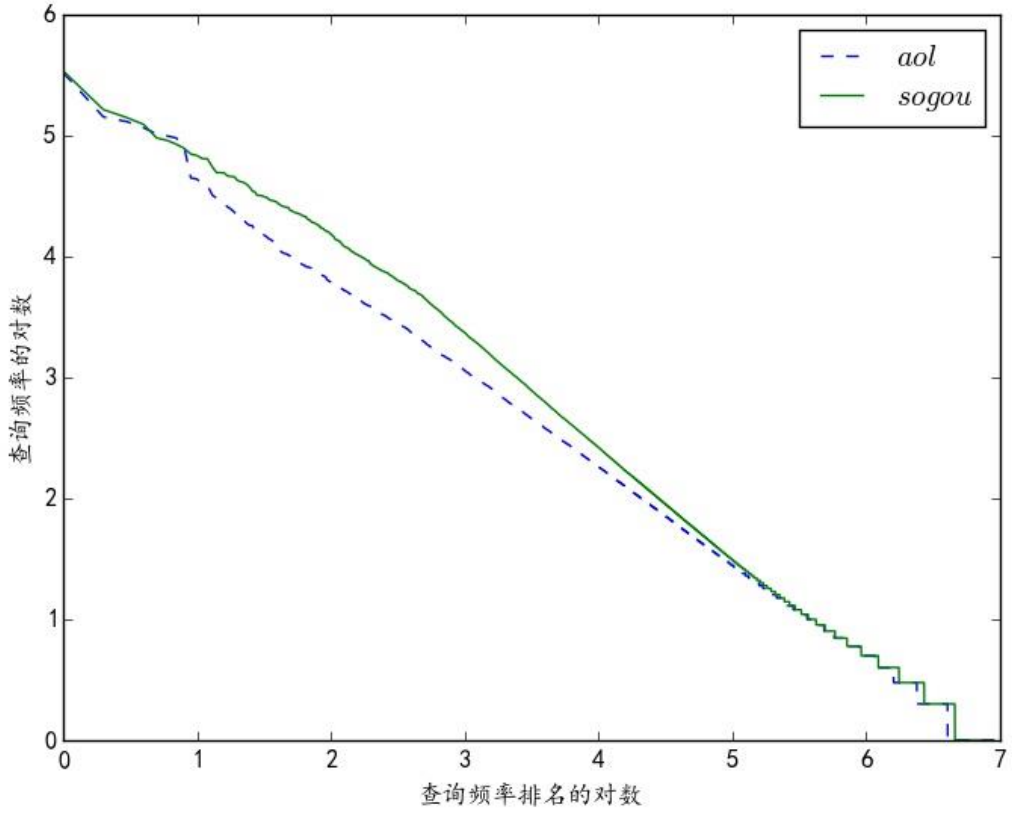

图 4 对数坐标下两种查询日志中查询频率与查询频率排名的关系

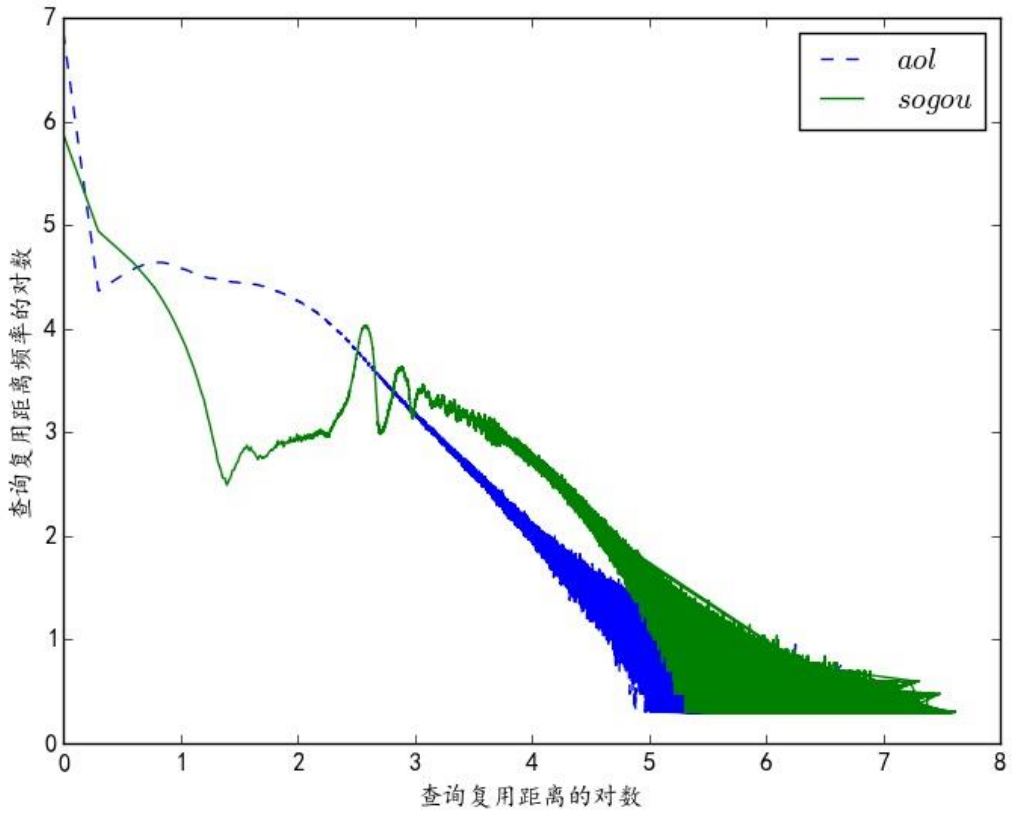

图 5 在对数坐标上两种日志里查询复用距离的频率

3.4 实验

3.4.1 实验目的

按真实的搜索引擎日志模拟用户请求，测量在九种缓存替换算法应用于 SDC 结果缓存的时的性能和效率。从而为选择合适的动态缓存策略提供依据。

3.4.2 实验设置

SDC 由静态缓存部分和动态缓存部分组成。静态缓存部分存储固定的缓存内容，填满了由过去出现频率最高的那些查询。动态缓存部分由指定的缓存替换策略来管理。参数 f, 取值从 0 到 1，表示静态缓存大小占 SDC 总缓存大小的比例。它取 0 时代表完全的动态缓存;它取 1 时代表完全的静态缓存。当查询 q 到来时，SDC 首先在静态缓存中查找 q, 然后在动态缓存中查找 q, 如果找到了，它向用户返回相关的查询结果; 否则 SDC 向搜索引擎后台请求查询结果，根据采取的替换策略，返回的查询结果可能替换动

态缓存中的实例。

由于 SDC 静态缓存部分需要被预先被填充，我们将查询日志分为两部分，一个训练集，包含日志里 2/3 的查询; 一个测试集，包含剩余的查询。我们用 N 来表示缓存能容纳查询结果实例的个数。实验开始时，我们将训练集中频率最高的 N*f 个查询结果填入静态缓存部分, 并提交训练集中的查询序列来预热动态缓存部分。

3.4.3 实验环境

1) 硬件环境：

- CPU：AMD Athlon(tm) X4 750 Quad Core Processor　3.40 GHz
- 内存：4.00 GB
- 硬盘 : 1T

2) 软件环境：

- 操作系统：Win7 64 位操作系统
- Python 环境 ： Python 2.7.6

3) 数据集：

- 搜狗 2011 年 12 月 30-31 日两天的查询日志
- 美国 AOL 查询日志，从 2006 年 3 月 1 日 到 5 月 31 日

3.4.4 实验结果

在搜索引擎的查询结果缓存采用 SDC 策略时，参数 f 可以通过用历史数据进行实验来确定而使缓存命中率尽可能提高。我们分别实现 OPT、ARC、LRU、CLOCK、kRANDOM(k=3)、SKLRU(k=4)、RANDOM、LIRS、LFU 九种算法作为 SDC 的动态缓存替换策略。对每种替换策略和缓存大小测出静态缓存比例 f 为 0.1,02….1 时的 SDC 缓存命中率,并找出静态缓存比例最优时，缓存命中率能达到的最大值。图 6 显示了搜狗日志上各替换策略实现的 SDC 能达到的最大缓存命中率。其中 OPT 实现的 SDC 达到最高的缓存命中率，LFU 的 SDC 缓存命中率最低。而其它缓存算法达到的缓存命中率相差不大。与图 6 类似，图 7 显示了 aol 日志上各替换策略实现的 SDC 能达到的最大缓存命中率。

为了进一步比较 ARC、LRU、CLOCK、kRANDOM(k=3)、SKLRU(k=4)、RANDOM、

LIRS 替换策略的 SDC 的缓存命中率。在图 8,图 9 中我们将它们与 RANDOM 策略的 SDC 缓存命中率对比。我们可以从图 8，图 9 中看到,ARC 的 SDC 达到最高的缓存命中率,而 LIRS 缓存命中率较低。从直觉上来说，RANDOM 算法在缓存满的时候随机移出一个查询结果，应该是最差的方案，然而在实验中 LIRS,甚至 SKLRU 的 SDC 缓存命中率低于 RANDOM 算法。

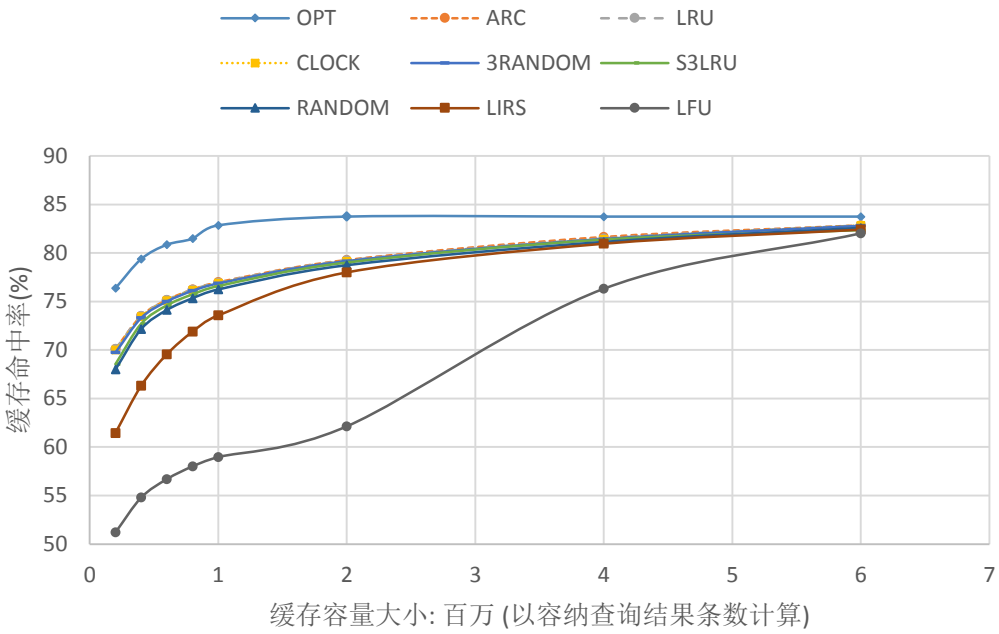

图 6 搜狗查询日志上各替换策略实现的 SDC 能达到的最大缓存命中率

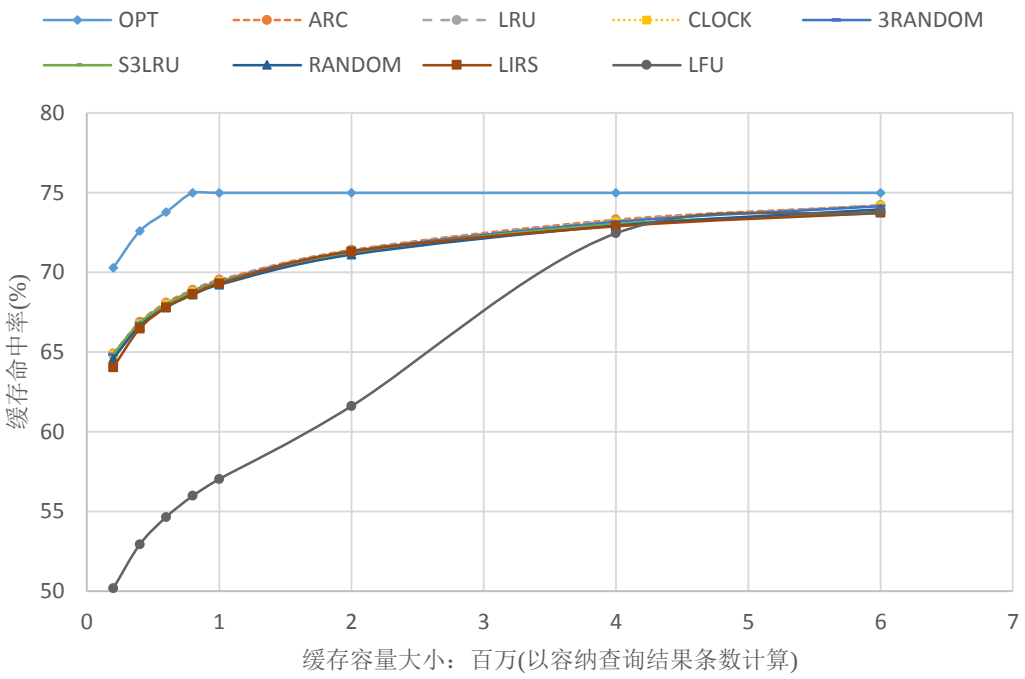

图 7 aol 查询日志上各替换策略实现的 SDC 能达到的最大缓存命中率

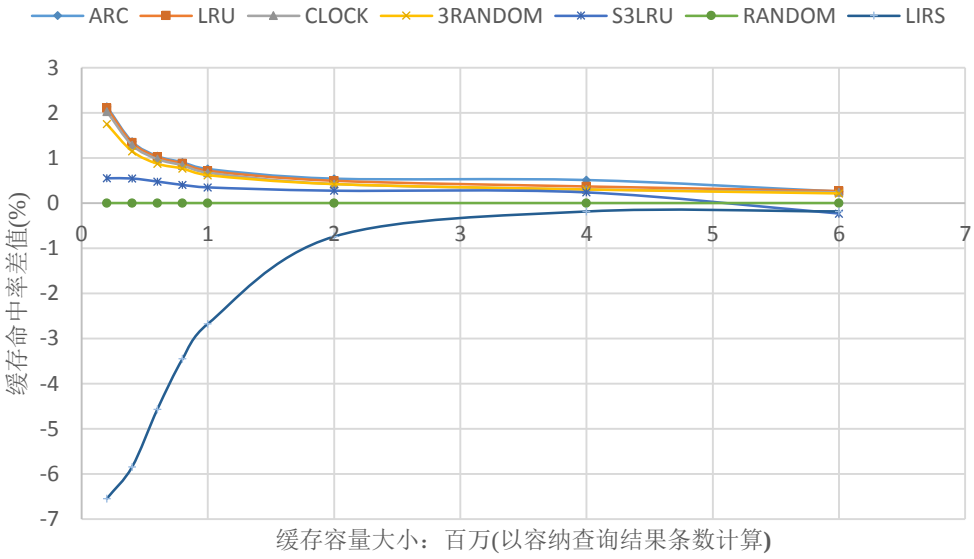

图 8 在搜狗查询日志上与 RANDOM 策略的 SDC 缓存命中率对比

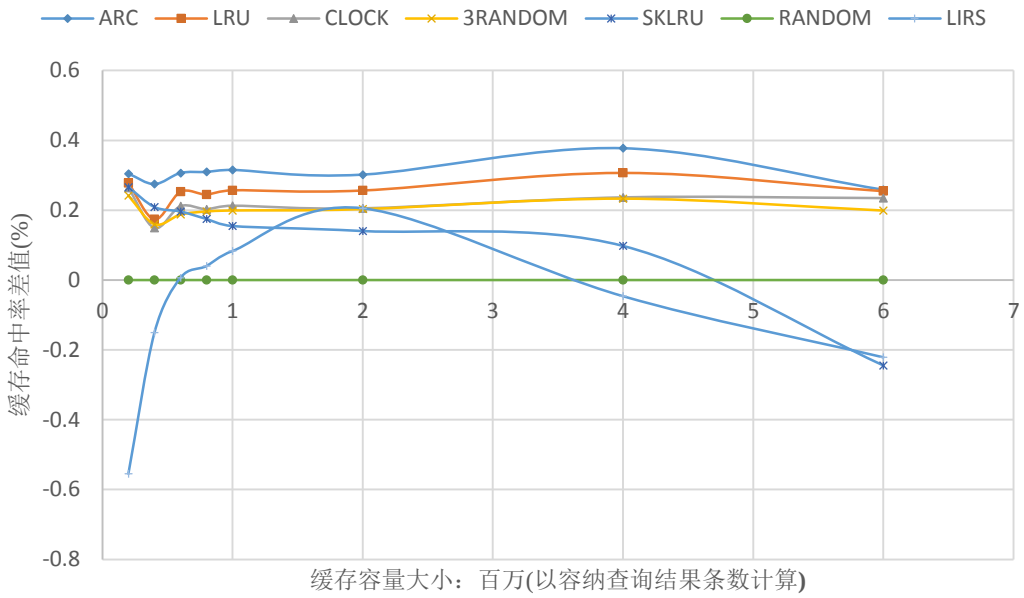

图 9 在 aol 查询日志上与 RANDOM 策略的 SDC 缓存命中率对比

图 10,图 11 画出了在两种日志上，单独的动态缓存替换策略即 f=0 时的缓存命中率。我们发现 LFU 本身作为搜索引擎结果缓存替换策略性能就比 RANDOM 策略更差。为了更清晰的比较动态缓存替换策略，图 12、图 13 画出了单独的 ARC、LRU、CLOCK、kRANDOM(k=3)、SKLRU(k=4)、RANDOM、LIRS 缓存替换策略相对于 RANDOM 策略的差值。可以看到，单独的 SKLRU(k=4),LIRS 替换策略缓存命中率优于 RANDOM 策略。然而 SKLRU 在 LRU 在基础上更多的考虑了 frequency, 而静态缓存部分已经存储了历史记录里频率最高的那些查询，因此它作为 SDC 动态缓存部分的替换策略命中率较低。

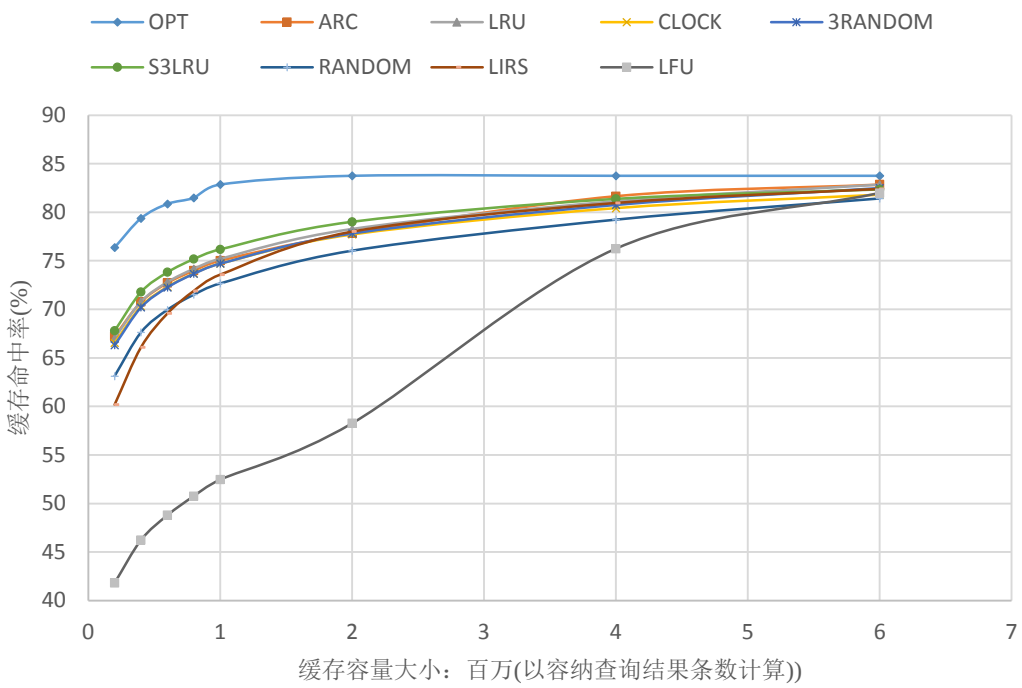

图 10 在搜狗查询日志上九种缓存替换算法的缓存命中率

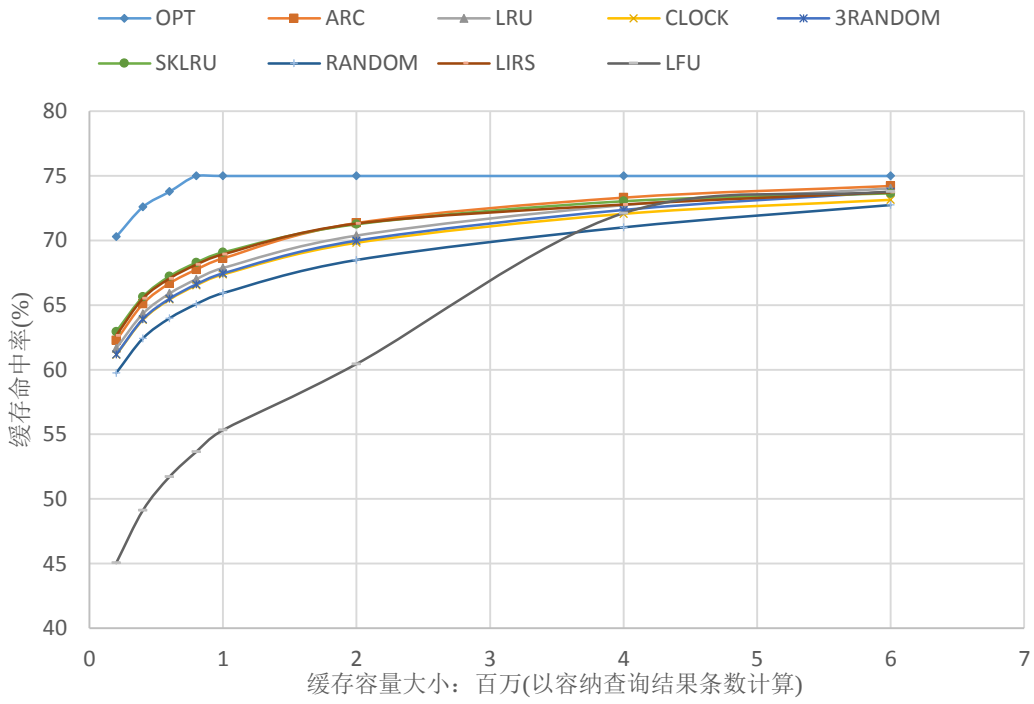

图 11 在 aol 查询日志上九种缓存替换算法的缓存命中率

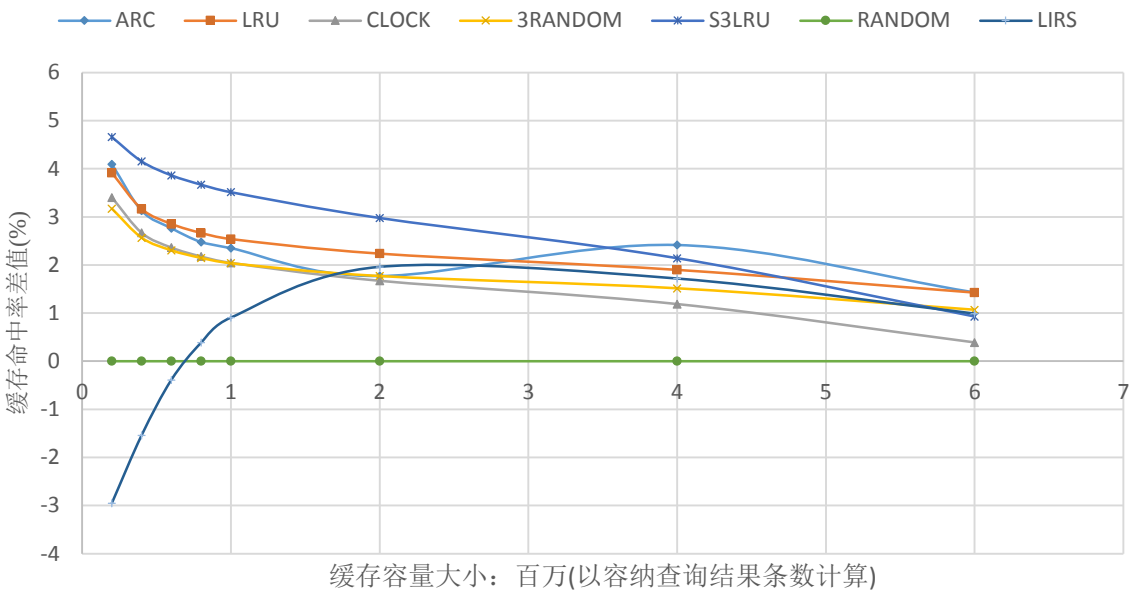

图 12 在搜狗日志上各替换算法缓存命中率与 RANDOM 算法差值

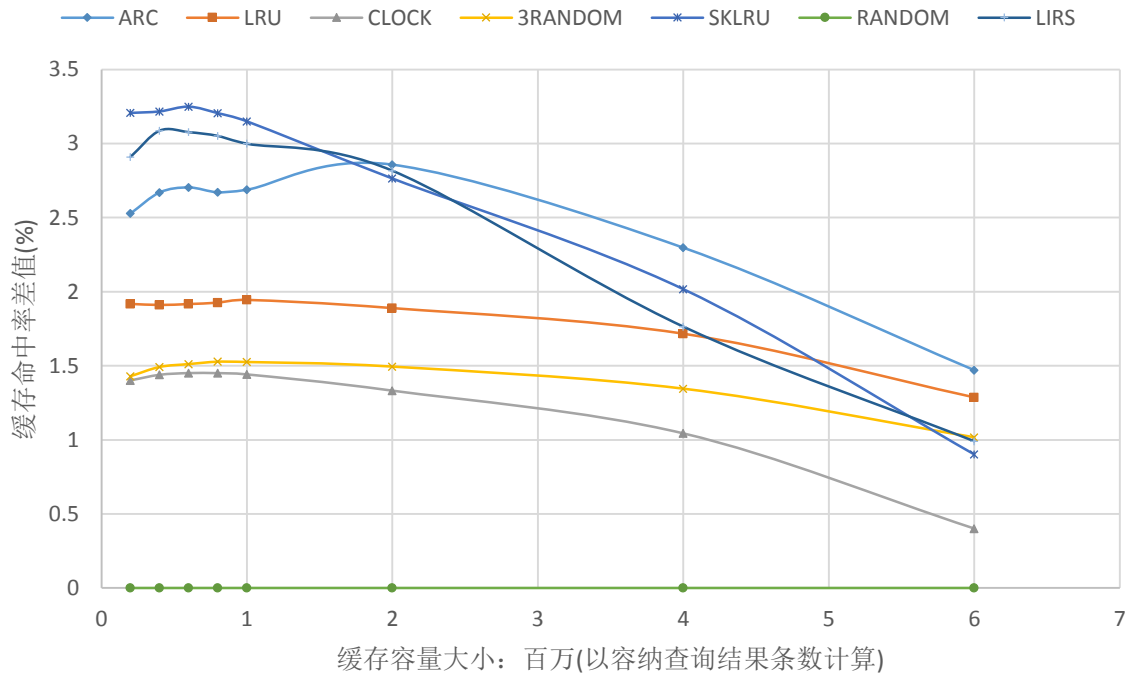

图 13 在 aol 日志上各替换算法缓存命中率与 RANDOM 算法差值

实验表明，除了 OPT 策略外，ARC、LRU、CLOCK、3RANDOM、SKLRU 替换策略在 SDC 上的命中率也高于 RANDOM 策略。ARC 作为 SDC 的替换策略命中率最高，但较复杂的替换策略往往需要使处理查询的时间变长。图 14,图 15 显示了在单独的替换

策略实现的动态缓存处理完两种日志的测试查询的运行时间。本文的实验中 SKLRU 有 4 层 LRU 链表，即 K=4。本文实验中 4 层 LRU 缓存的容量大小同为缓存大小的四分之一。SKLRU 运行最为耗时，而 CLOCK 和 RANDOM 算法运行最快。然而，作为 SDC 动态缓存部分的替换策略，对未来查询预测较好的策略将占用更多的容量使缓存命中率最大，而静态缓存比例会影响 SDC 处理查询的时间。图 16,图 17 显示了应用替换策略的 SDC 调节 f 使命中率最大时的查询处理时间。结果仍然是，SKLRU(k=4)时处理查询的需要花较多时间，而 CLOCK,RADOM 替换策略处理查询最快。

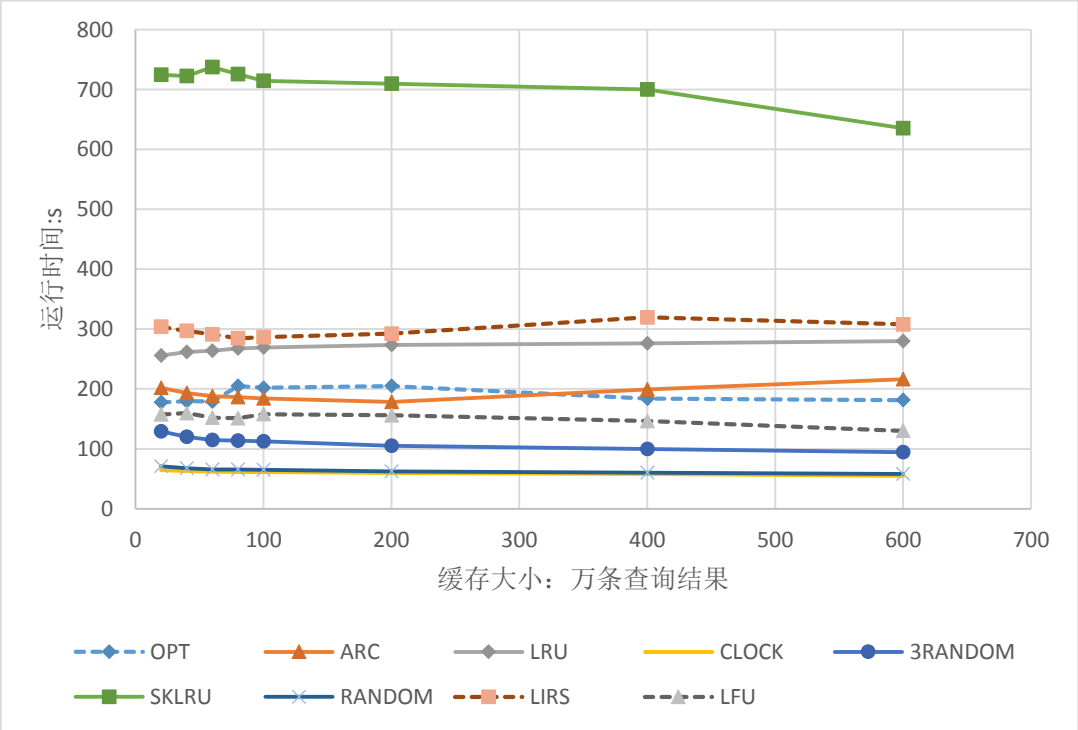

图 14 在搜狗查询日志上各缓存替换算法的运行时间

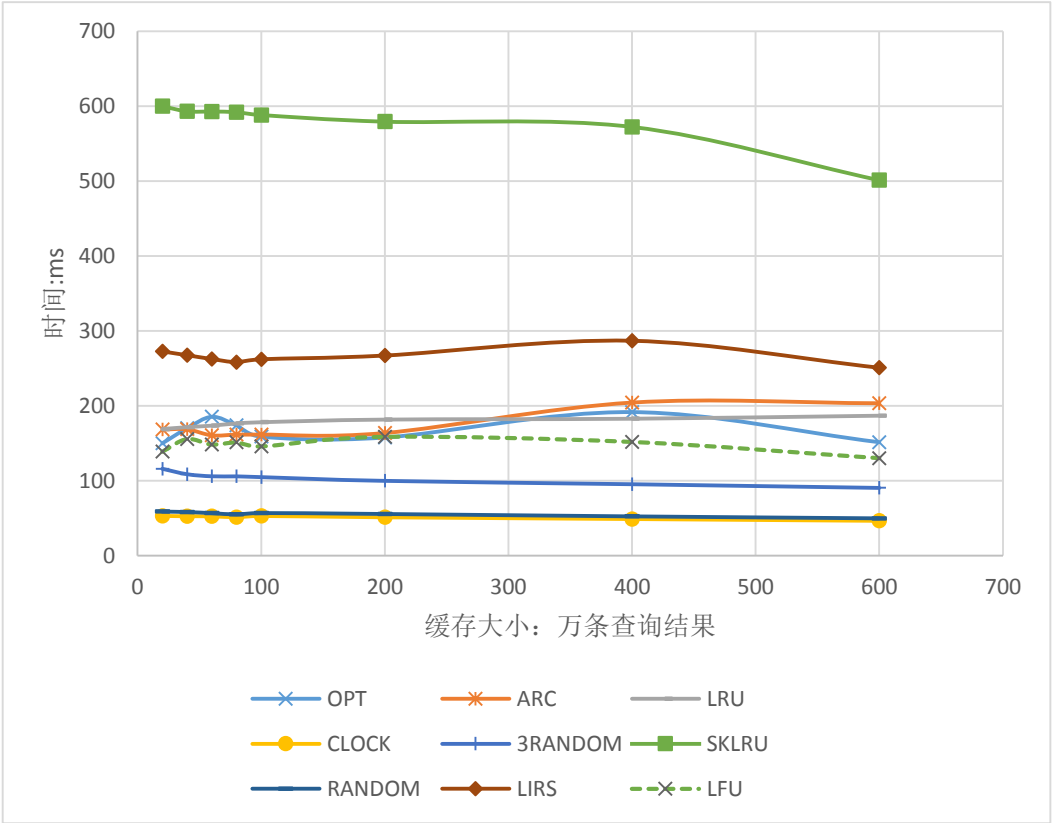

图 15 在 aol 查询日志上各缓存替换算法的运行时间

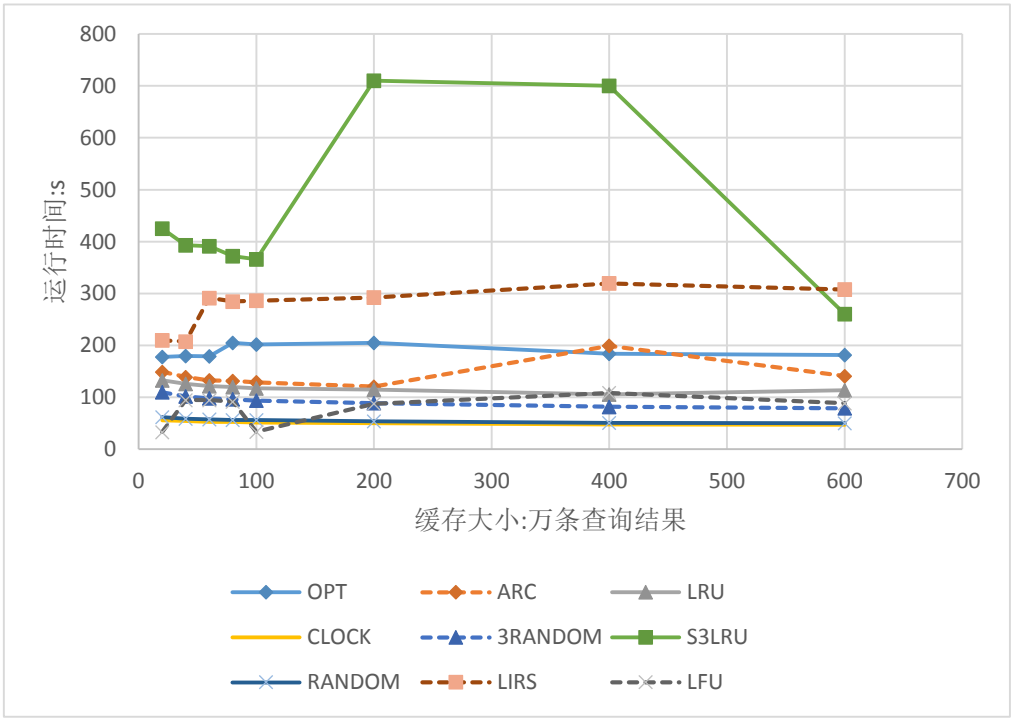

图 16 在搜狗查询日志上各缓存替换算法实现的 SDC 运行时间

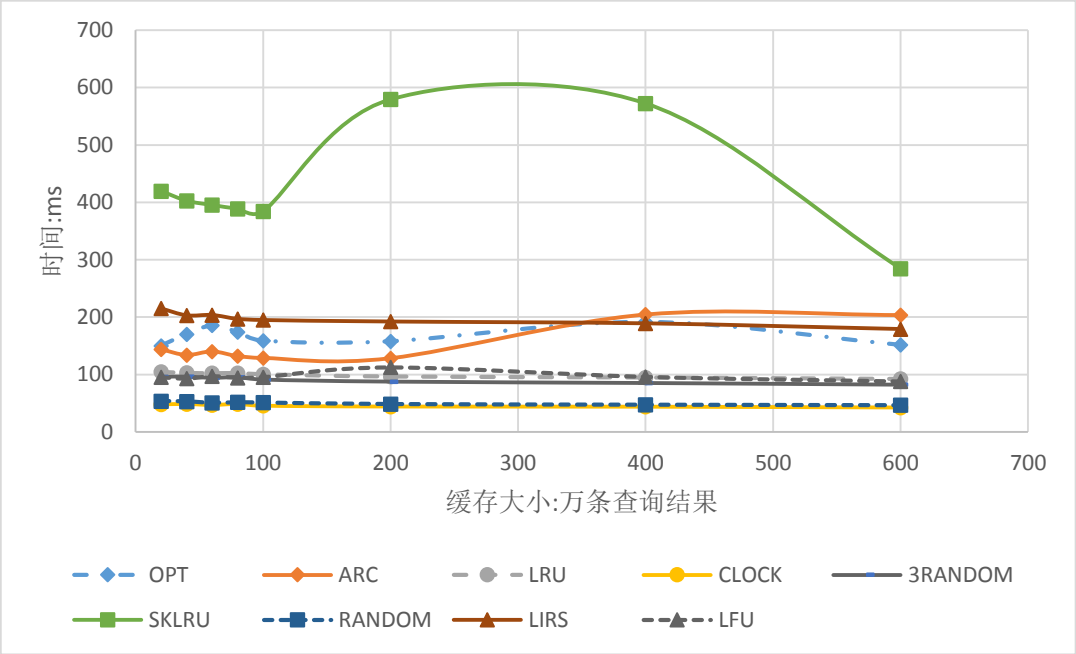

图 17 在 aol 查询日志上各缓存替换算法实现的 SDC 运行时间

随着缓存容量的不同和查询日志的不同，缓存算法的性能会略有差异，本文综合考虑不同缓存容量和不同查询日志的缓存算法性能，分别将应用 7 种缓存替换策略的 SDC 的性能分等级列于表 2，表 2 不包括 OPT 算法，因为 OPT 假设已知了未来的查询请求序列而并不能应用于实际。表 2 也不包括 LFU 算法，因为 LFU 算法在两种查询日志上的性能明显低于用于作为缓存命中率下界的 RANDOM 算法。从 3.3 节对 LIRS 的介绍可知，LIRS 对复用距离十分敏感,注意在 LIRS 算法在搜狗日志上的缓存命中率低于 RANDOM 算法而在 aol 日志上缓存命中率高于 RANDOM 算法，这与 3.4 节图 5 显示的搜狗日志与 aol 日志查询的复用距离差异相符合。

从表 2 来看，越是复杂的缓存替换策略运行速度越慢，而全动态缓存命中率也越高。而把缓存替换策略应用于 SDC 时，部分在全动态缓存命中率高的缓存替换策略却性能不佳，如 SKLRU。在 SDC 静态缓存已经充分使用频率信息的情况下，SKLRU 又过多地使用频率信息来预测将来的查询请求使局部请求频率较高而以后不再使用的查询结果长时间驻留于缓存中，因而在 SDC 框架上性能不佳。而 ARC 缓存替换策略只考虑最近二次信息，并能反馈动态调整一级 LRU 缓存和二级 LRU 缓存的大小，因而适量的使用频率信息所以应用在 SDC 上获得最高的缓存命中率。

表 2　各缓存替换策略的性能

缓存替换策略	SDC 缓存命中率	全动态缓存命中率	运行速度
ARC	最高	次高	较慢
LRU	较高	较高	较快
CLOCK	较高	较低	最快
3RANDOM	较高	较低	较快
SKLRU	较低	最高	最慢
RANDOM	低	低	次快
LIRS	低	较高	较慢

单独考虑 SDC 命中率应该采用 ARC 替换策略。而采用 CLOCK 替换策略的 SDC 能快速的处理查询而且达到较高的缓存命中率。

3.5　本章小结

在搜索引擎查询结果缓存容量有限的情况下，本章在搜索引擎查询结果缓存常用的框架 SDC 的基础上，讨论动态部分的缓存替换算法的选取对 SDC 缓存命中率的影响。通过在真实查询日志上的实验，和对缓存算法的时间复杂性的分析为选择动态缓存算法提供依据。本章先介绍了九种动态缓存算法和 SDC 缓存，然后分析了用来测试缓存算法的查询日志的频率分布和查询复用距离。最后按查询日志模拟用户查询请求进行了实验。实验表明，单独考虑 SDC 缓存命中率应该采用 ARC 替换策略。而采用 CLOCK 替换策略的 SDC 能快速的处理查询而且达到较高的缓存命中率。

第四章　基于历史查询的结果缓存预取策略研究

本章研究面向实时检索的缓存更新策略，为搜索引擎缓存加上查询预取模块。本章以查询开销作为选择被预取的查询结果的根据之一，以相同的搜索引擎后台处理能力，预取更新尽可能多的查询结果，从而达到更高的缓存命中率。

随着硬件性能的提高和价格的降低，廉价的存储设备容量大到足够存储所有过去用户查询的结果。在缓存容量无穷大的假设下，除了特定查询的第一次发起，其它的查询都能由缓存提供答案。一个无穷大的缓存面临的关键问题是缓存结果陈旧的问题。

本章假设搜索引擎的体系结构如下图18所示。用户查询被提交到搜索引擎的前端。搜索引擎的前端包含一个无限容量的结果缓存，而结果缓存中缓存的实体与它的生存时间值相关联。如果一条查询在结果缓存中被查找到，而它的查询结果还没有过期，那么这条查询的结果由缓存给出。否则，这条查询被提交给搜索引擎后台。搜索引擎后台由许多节点组成，且拥有一个在文档集上建立的索引。当一条查询被后台处理后，查询的结果以及计算出结果的时间会被打包存入查询结果缓存中。这样，查询过期的时间就能够被确定下来。查询预取模块与搜索引擎前端、后台交互。它负责从查询日志或结果缓存中选择一些已过期或即将过期的查询并把这些查询提交给搜索引擎后台。这些由搜索引擎后台计算出的查询结果被存在查询结果缓存中。

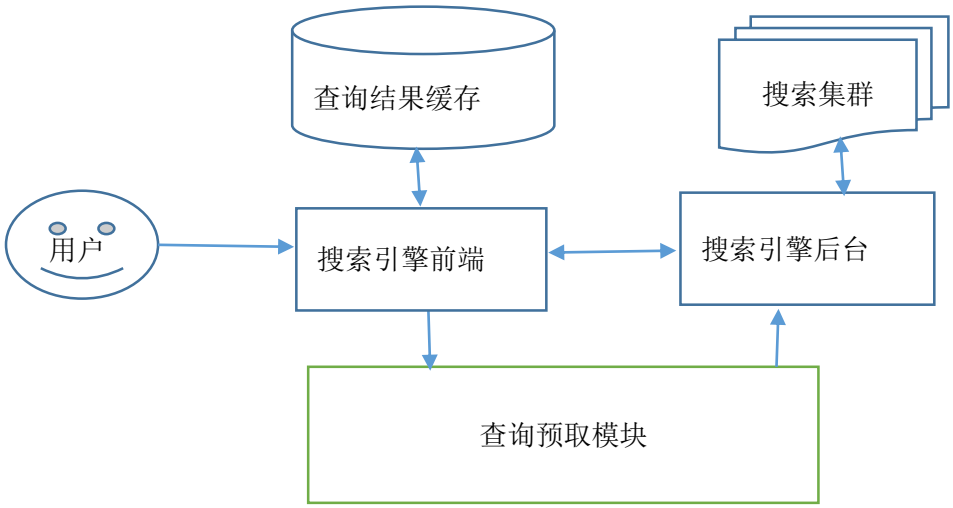

图 18 查询预取模块在搜索引擎的体系结构中

查询预取模块的意义在于尽可能地减少因为陈旧过期造成的用户查询请求未命中缓存。通过在用户发起的查询之前，预取相应的过期或即将过期的查询结果可以提高缓存命中率。然而，预取查询会消耗搜索引擎后台的计算资源而且并不是所有预取操作都是有用的。如果一条被预取了结果的查询在过期之前没有被用户再次请求，那么这次预取只是导致计算资源的浪费。

查询预取最大的好处在于提高了缓存命中率。使用缓存中的结果立即响应用户查询来减少平均查询响应时间。此外，查询预取还通过预取更新即将过期的查询结果从而提高了查询结果的新鲜度。

在过去的关于搜索引擎缓存如何调度存储什么内容的研究中，未命中缓存的代价往往被忽略，计算所有的查询的结果都被假设有相同的代价。从 CPU 处理时间、网络要求等方面来看，向搜索引擎提交的查询的处理成本开销差异很大。因此假设所有的缓存未命中带来同样的开销是不现实的，而且被提交给搜索引擎的查询的计算代价显著不同。本章基于预取查询开销来研究查询结果缓存预取策略。

4.1 查询预取模块

在不产生负荷影响用户查询处理的前提下，查询结果需要尽可能多地被预取。因此，查询预取的灵活性取决于后台负载低的时间。本节考虑了这样的问题：是否存在足够长的时期后台的访问量较低，而搜索引擎后台能够利用空闲的周期来进行预取。图 19 展示了搜狗搜索引擎后台在特定一天 0 至 24 小时收到的查询请求。最上面的曲线是缓存中查询结果生存时间为 0h 时，即无缓存时,每秒到达后台的请求数，这时所有的用户查询都会到达后台。底部的曲线是在无生存时间限制的情况下到达后台的查询流量。这时只有历史从未出现过的查询才会不命中缓存而到达后台。这时结果缓存里的内容不会更新，缓存里的结果和当前索引计算的结果很可能不一至。夹在中间的曲线是生存时间设为 1 小时的情况下到达后台的查询流量。图 19 中，在 10 至 22 点的流量高峰时间，近一半在生命时间为 1 小时情况下未命中缓存而到达后台查询是由于过期了，这些查询结果在生命时间足够的情况能由结果缓存提供。而查询结果预取能够更新过期的或即将过期的查询结果，从而提高缓存命中率，减少到达后台的查询请求。

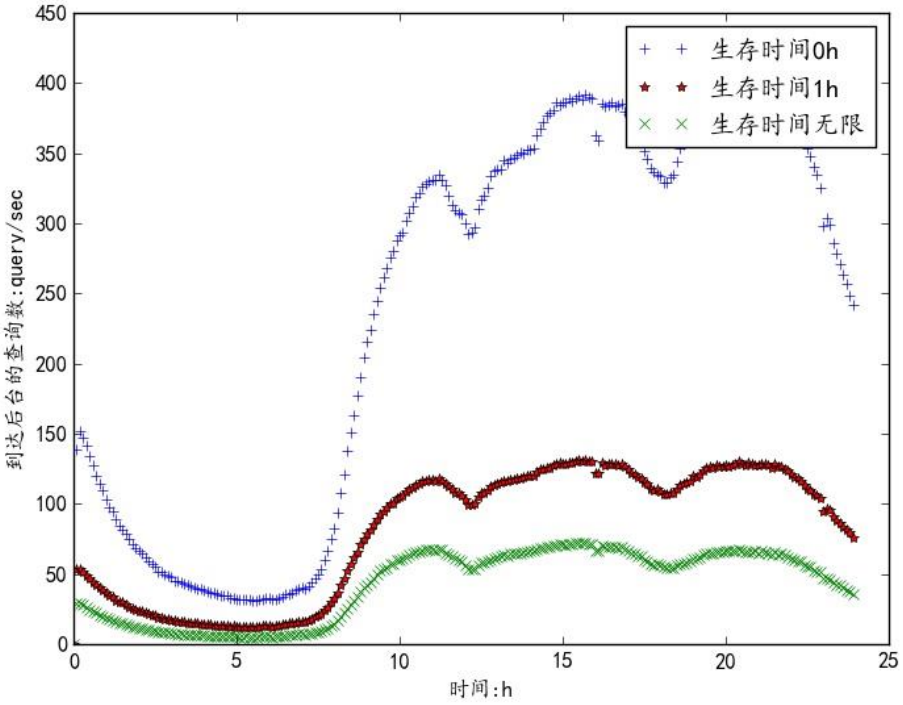

图 19 一天中到达后台的查询流量

查询预取模块的结构如图 20 虚线下的部分所示。查询选择器将查询结果缓存中的查询按估计的重现时间放入查询分桶的结构中，作为候选的被预取查询。查询分桶结构根据查询重现时间的估计值将查询分成若干个时间区间。设查询的生命时间为 T, 查询分桶时间区间大小为 s(s<T 且 T 被 s 整除),N=T/s。则在(t, t+s)区间的预取如下：在 t 时刻将查询分桶里被预测在(t, t+T)时间区间会被用户发起的查询作为候选查询。即将(t, t+s), (t+s, t+2s), …(t+Ns-s), (t+Ns)这些时间区间里的查询加入候选集。然后，将候选集里的查询按一定的权值排序，仅值的计算与查询时间开销记录有关。在接下来的时间里，搜索引擎后台优先处理用户查询，并通知查询缓存预取模块后台处理完用户查询后剩余的计算能力。查询预取模块根据后台剩余的计算能力，从预取查询候选集里选择权值大的前 k 个即将过期或已过期的候选预取查询提交给后台重新计算。

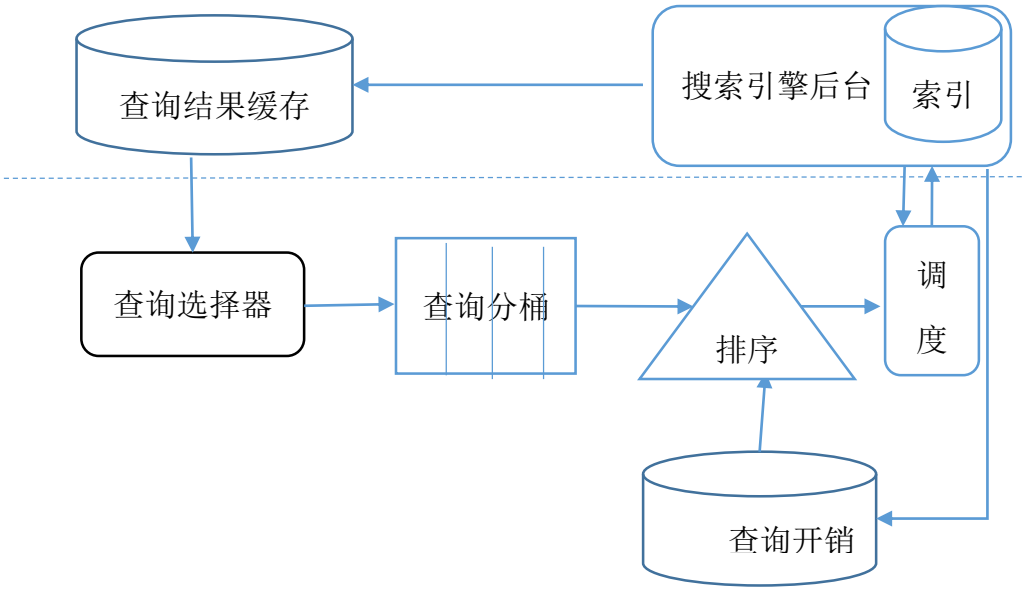

图 20 查询预取模块

4.2 查询开销计算

在过去的关于搜索引擎缓存如何调度存储什么内容的研究中，未命中缓存的代价往往被忽略，计算所有查询的结果都被假设有相同的代价。然而被提交给搜索引擎的查询的计算代价显著不同。计算查询结果的时间开销主要包括从磁盘上读取倒排表的时间开销，计算倒排表交集的时间开销，读取文档的时间开销和生成网页摘要的时间开销。由于倒排表的长度各异，所以读取倒排表的时间各不相同。由于查询的词项个数不同，各词项的倒排表长度也不一样，所以计算倒排表交集的开销各不相同。由于各查询的相关性排名前 k 的文档数量和长度不同，所以读取文档的开销差异很大，生成网页文档摘要的开销也各不相同。这使得查询的处理时间只与查询本身的词项构成有关而与查询在日志里出现的频率，顺序等无直接关联。

在本文中以搜索引擎后台处理查询请求的时间来衡量查询开销。本文采用真实的互联网语料库建立索引，在单机环境下测量出真实的查询日志里处理每条查询需要的时间来代表查询开销。

本文从约 17T 的搜狗 2012 年语料库中选取 1T 的网页文档，使用 lucence 开源类库建立索引。并让索引分别计算查询日志里的每条查询的结果，并记录下时间开销。使用

的搜狗查询日志包括两天的查询记录 43545444 条,其中 8935490 条独一查询。图 21 在双对数坐标下，描绘了每条查询在日志里出现的频率和单个计算节点按索引计算出查询结果所消耗的时间。由此可见查询处理时间各不相同且与查询出现频率并无关系。

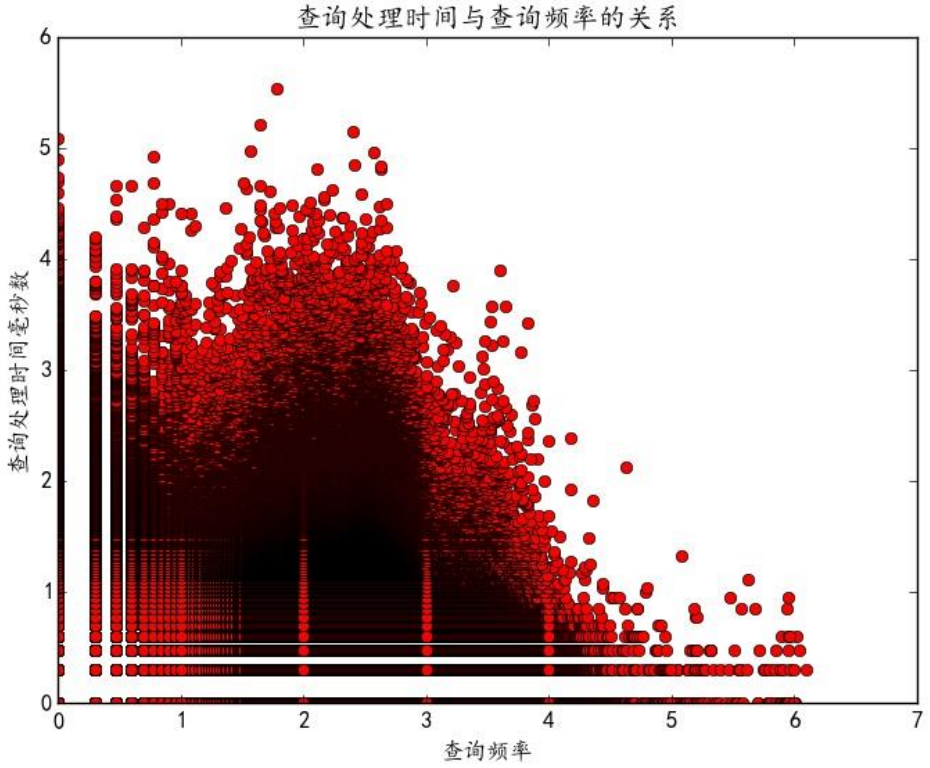

图 21 对数坐标下查询处理耗时与查询频率的关系

4.3 查询结果缓存预取候选集

4.3.1 周期性规律

从图19中最上端的曲线可以看到,用户查询请求量随时间变化呈现一条正弦曲线。晚上用户查询量低而下午高，以一天为周期变化。过去的研究表明用户查询不仅在数量而且在内容上也以天为周期有一定的重复性，有些查询在每天相近的时刻被用户发起。因此，预取前一天的日志里的特定的查询是一个合理的策略。在本节，用 T 来表示缓存中查询结果的生存时间，用 t 来表示当前的时刻用 t’ 来表求 24 小时之前的时刻。把区间 [t, t’]被分成 N 个时间区间，每个区间

的长度为 s = (t – t’) / N。则第 i 个时间区间为[t’+(i-1)*s, t’+i*s]。每个时间区间对应查询

分桶结构中的一个桶，桶中存的查询可能在指定的时间段被提交给服务器。在时间区间[t+(i-1)*s, t+i*s]预取的查询结果可以为[t+(i-1)*s,t+(i-1)*s+T]时间段发起的用户查询提供新鲜的结果,所以要预取在[t'+(i-1)*s, t'+(i-1)*s+T]时间段发起的查询的结果。将这些查询存放在 T/s 个桶里的查询作为这个时间区间的候选集，从中选择查询提交给后台预取。在 t+i*s 时刻，时间区间[t+(i-1)*s, t+i*s]结束的时候就开始预取下一个候选集里的查询。

4.3.2 回归预测下次查询发起时间

本节希望被预取的查询结果在过期之前能被使用到，即用户再次提交被预取的查询请求时，被预取的结果未陈旧过期。需要鉴别过期的或即将过期的查询结果是否会被再次发起并由于缓存结果处于过期状态而未命中缓存。通过预测查询在将来下一次被用户发起的时间 e，即可得知应该何时预取特定查询的结果。

一个经典的缓存算法 LIRS，采用当前查询请求与上次查询请求发起的栈距离来估计将来下次查询请求出现的栈距离。在查询流量平稳的情况下，栈距离与查询间的时间间隔线性相关。本节使用查询与其上次出现的时间间隔来预测下次查询出现的时间间隔。图 22 描绘了上次查询出现时间间隔与查询重现时间间隔的关系。可以看出上次查询出现时间间隔较小，小于 15000s 时，查询重现时间间隔也较小，多数小于 20000s。然而查询重现时间和上次查询出现时间间隔并无明显的正相关关系。这是由于用户查询的流量不稳定影响了查询时间间隔与查询栈距离的关系。本节加入查询发起的小时数来指示当前查询流量及查询流量变化趋势来预测查询再次发起时间。图 23 描绘了查询再次发起时间间隔与查询时间的关系。在虚线附近的查询点很稀疏，虚线的方程约为 y/3600+x=5，表示查询发起时间+查询再次时间间隔即查询再次发起时间为 5h。图 19 上方曲线所示，大约 5 点时查询流量达到最低点，这与图 23 中虚线表达的查询将来在 5 点左右被发起的概率很小相一致。

既然使用了上次查询的发起的时间间隔来预测重现时间间隔，那么上次查询发起与上上次查询发起的时间间隔也应当有相似的作用。本节将多次查询之间时间间隔 gap 简化为在固定周期内相邻查询的时间间隔平均值 avg 代替，而 avg = gap/freq，其中 freq 表示在时间间隔 gap 内查询的频率。所以，可以用查询频率来代替时间间隔均值来预测查询重现时间间隔。图 24 描绘了查询频率与查询重现时间间隔的关系。可以看到查询重现时间与查询频率反相关。即查询频率越小，查询重现时间通常越大。

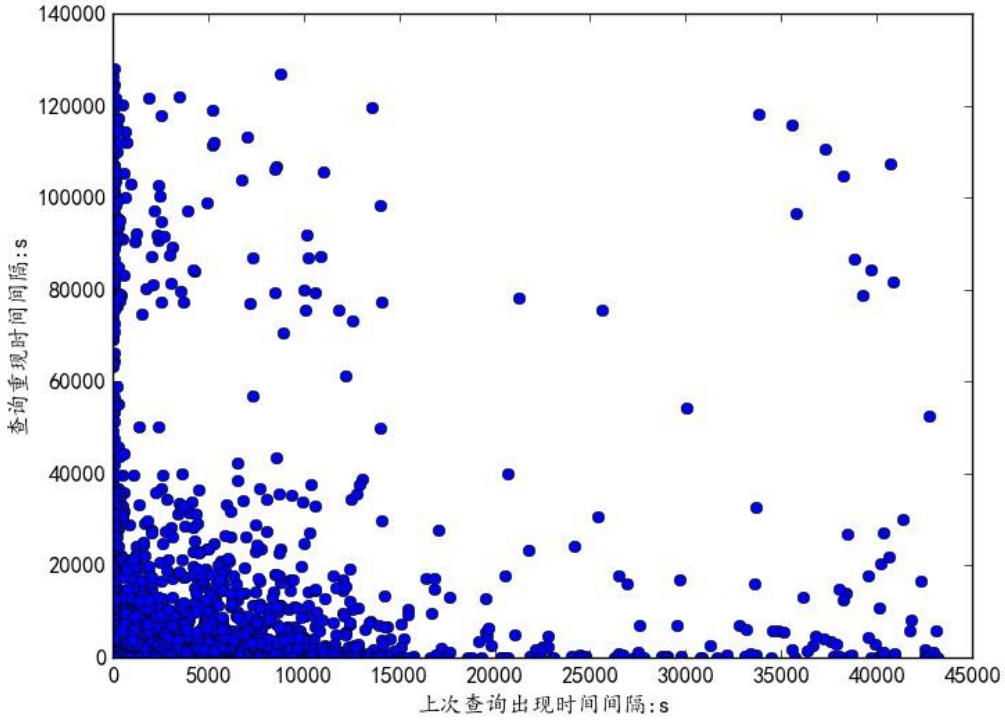

图 22 查询重现时间间隔与上次查询出现时间间隔的关系

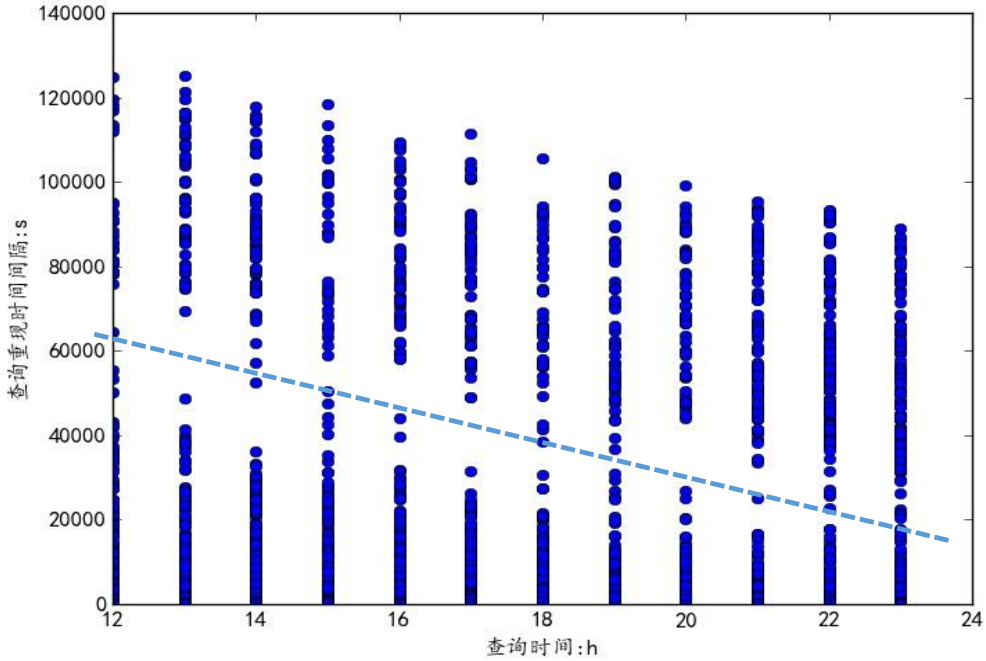

图 23 查询重现时间间隔与查询发起时间的关系

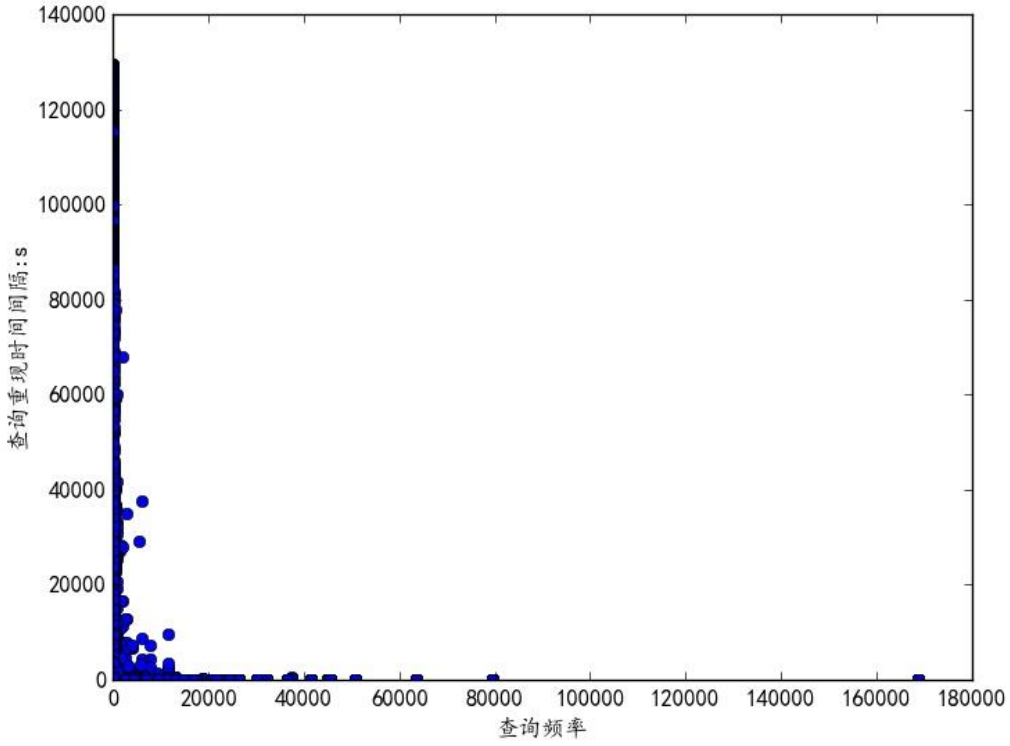

图 24 查询重现时间间隔与历史查询频率的关系

从时间效率方面考虑，本节采用梯度提升回归树训练模型来预测查询下一次出现的时间间隔 n, 然后就能用 e = t + n 来估计查询下一次被用户发起的时间。使用的属性包括此次查询与上次查询被发起的时间间隔、此次查询发起的时刻、查询在前一天出现的频率。

4.4 候选集查询排序

由于后台的处理能力限制，不可能预取所有的查询结果，需要把查询按它们的重要性排序，而优先预取重要性高的查询。给查询排序时，在候选集里的频率是一个要考虑的重要因素。频率越高则在相应时间段出现查询的概率越大。查询开销是另一个应该考虑的因素，优先预取查询开销小的查询就能用同样的后台处理能力预取更多的查询。本章将查询的重要性 w 衡量为频率 freq 除以(1+查询开销 timecost)。即 w(q) = freq(q)/1+timecost(q)。

4.5　实验

4.5.1 实验目的

按真实的搜索引擎日志模拟用户请求，验证基于历史查询的预取模块给查询结果缓存带来性能上的提升。

4.5.2 实验模拟设置

4.5.2.1　查询时间开销测量实验设置

本文从搜狗 17T 网页文档中抽取 1T 的网页文档，使用 lucence 开源搜索引擎类库建立索引。使用的字段及其存储和索引方式如表 3 所示。如果字段是索引的，则可以按字段的文本内容查询，如果字段是存储是则可以在查询后获字段的信息。如:本文的网页文本字段是索引的，所以可以按网页文本查询，但查询结果里不包括网页文本信息而是提供文档的 url 和唯一标志文档的文档 id。本文在处理中文时使用 lucence 自带的 CJK 分词模块。

表 3　创建索引参数

字段	是否索引	是否存储
网页文本	是	否
网页文档 id	否	是
网页 url	否	是

4.5.2.2　查询模拟实验设置

本文使用搜狗 2011 年 12 月 30-31 日两天查询日志来模拟用户查询。在周期性预取策略中，本文用第一天的查询记录预热结果缓存，而不计其请求次数和缓存命中率，用第二天的查询记录来测量缓存命中率。

在回归预测下次查询发起时间的预取策略中，第一天的查询记录首次发起的查询不能求出上次发起的时间间隔这一属性。本文用第一天的查询记录计算查询频率和上次发起时间间隔属性。用第二天 0-12 点查询记录训练模型，来预测第二天 12-24 点查询记录

的发起时间。因此，用第一天和第二天 12 点之前的查询记录来预热缓存，这一阶段没有采用预取且不计录其查询请求次数和缓存命中。第二天 12 点之后的查询用来评估预取策略。

本文考虑一个拥有 150 个节点的搜索引擎后台集群，每秒最秒能使用 150s 的单机 CPU 时间计算查询开销。设定使用 125s 的 CPU 时间来处理查询请求而不使后台处于满负荷状态。当到达后台的用户请求使用的 CPU 时间少于 125s 时会将剩下的计算时间用来预取。在周期性预取策略中本文将缓存实例生存时间设为 1h。而在回归预测预取策略中本文将缓存实例生存时间设为 3h,原因是回归预测的均方差为 3942 秒，接近一小时，将缓存实例生存时间设为 3h 能减轻回归预测误差对预取策略的影响。为了防止预取更新最近处理过的查询，本文限制被预取更新的查询最小年龄为生存时间的一半。对于查询分桶结果，本文设定每个查询分桶对应的时间区间长度为 6 分钟，即一天的将被预取查询按将被发起的时间分成 240 个桶。

本文用缓存命中率作为评估参数，缓存命中率越大则用户查询响应时间就短，用户获得更好的体验，同时到达后台的用户查询流量也少。本文设置了三组对比实验。无预取即不使用预取策略。频率时间开销权重策略按候选查询按重要性 w(q) = freq(q) / 1 + timecost(q) 排序并优先预取重要的查询。频率单元权重是 Simon Jonassen[19]等人提出的候选查询重要性排序方法。

4.5.3 实验环境

4.5.3.1 查询时间开销测量实验环境

1) 硬件环境：
   - 内存：64.00 GB
   - 硬盘 : 3T
2) 软件环境：
   - 操作系统：linux 64 位操作系统
   - Java 环境：Java<TM> SE Runtime Environment < build 1.7.0_51-b13 >
3) 数据集：
   - 搜狗搜索引擎 17T 网页文档
   - 搜狗 2011 年 12 月 30-31 日两天的查询日志

4.5.3.2　查询模拟实验环境

1) 硬件环境：

- CPU：AMD Athlon(tm) X4 750 Quad Core Processor　3.40 GHz
- 内存：4.00 GB
- 硬盘 : 1T

2) 软件环境：

- 操作系统：Win7 64 位操作系统
- Python 环境 ： Python 2.7.6

3) 数据集：

- 搜狗 2011 年 12 月 30-31 日两天的查询日志

4.5.4 实验结果

图 25 是无预取时的缓存命中率和在周期性预取策略下，频率时间开销权重排序预取的缓存命中率对比。图 26 是在周期性预取策略下，频率时间开销权重排序时的缓存命中率和频率单元权重排序预取的缓存命中率对比。由于频率单元权重和频率时间开销权重的缓存命中率差异太小，所以分开画在两张图上便于比较。可以看到利用周期性的预取策略相比无预取能提高约 6%的缓存命中率。而考虑了时间开销的频率时间开销权重比频率单元权重排序在白天能提高约 0.5%的缓存命中率，在夜晚后台空闲计算资源多时，能预取大多数候选查询而候选查询之间的排序方法对缓存命中率影响很小。

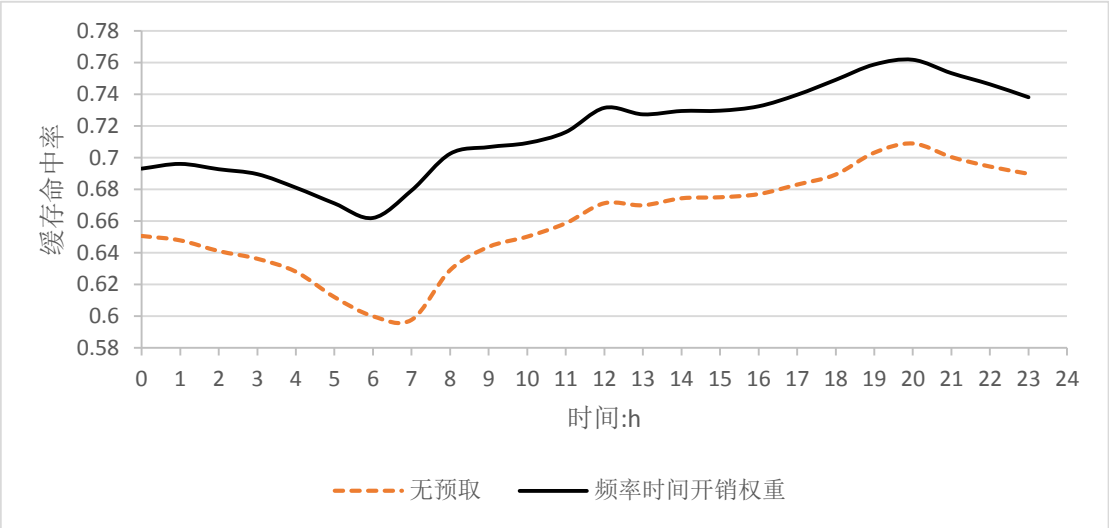

图 25　频率时间开销权重预取与无预取的缓存命中率对比

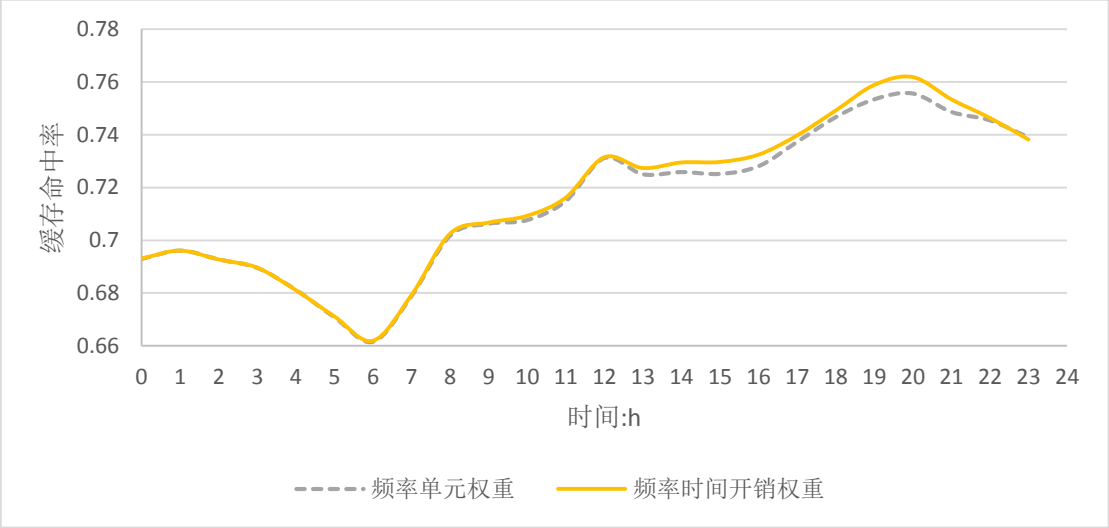

图 26　频率时间开销权重预取与频率单元权重预取的缓存命中率对比

图 27 是回归预测预取策略下采用频率开销权重对候选查询排序时的缓存命中率与不进行预取时缓存命中率的对比。12 时刚开始时候选查询分桶中刚开始填入候选查询，所以实际上预取很少。达到稳定后比无预取的缓存提高了约 3%的缓存命中率。同样采用频率开销权重对候选查询进行排序的缓存命中率与采用频率单元权重排序的缓存命中率相差很少。在图上难以展示出来，如表 4 所示，在生命周期为 3h 时，按频率单元权重排序候选预取查询时缓存命中率仅高出了约 0.01%。

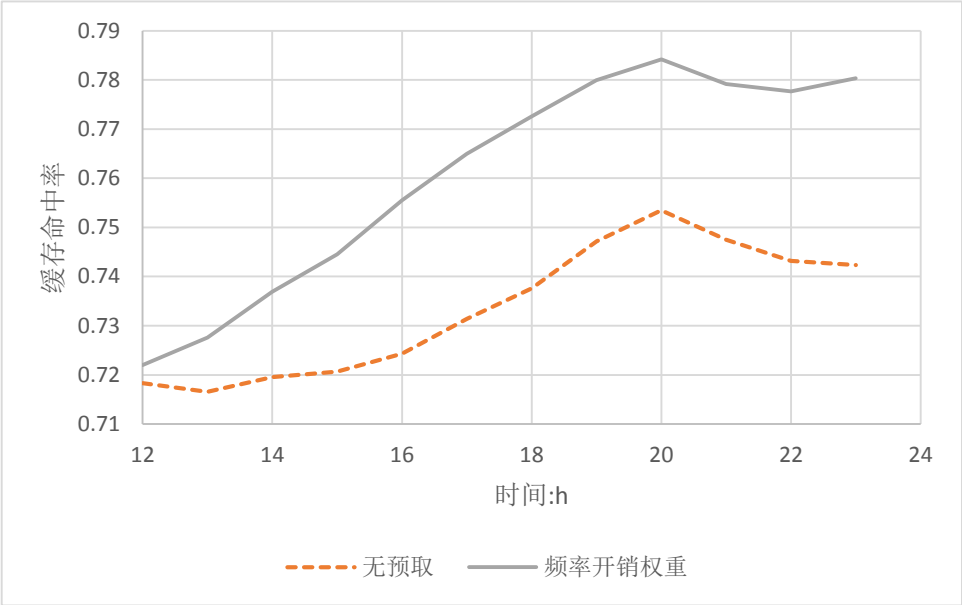

图 27　回归预测的频率开销权重预取与无预取的缓存命中率对比

表 4　回归预测的频率开销权重与频率单元权重预取的缓存命中率对比

时间:h	频率单元权重缓存命中率	频率开销权重缓存命中率
12	0.722011	0.722012
13	0.727585	0.727590
14	0.736906	0.736910
15	0.744475	0.744520
16	0.755476	0.755524
17	0.764872	0.764963
18	0.772542	0.772592
19	0.779855	0.779951
20	0.784110	0.784147
21	0.779056	0.779150
22	0.777641	0.777684
23	0.780248	0.780316

4.6　本章小结

基于历史查询的结果缓存预取策略在搜索引擎索引更新频繁的情况，利用后台的空闲计算周期来预取更新查询结果，使缓存的查询结果更新鲜而与索引保持一致，从而提高缓存的有效命中率，这里命中的陈旧过期的缓存查询结果视为未命中缓存。本文从周期性规律和回归预测两方面选择来待预取更新结果的查询。由于搜索引擎后台计算资源

有限，本文对候选查询按重要性和预取所需的代价排序，优先预取重要性高预取所需的计算资源少的查询。实验表明，相比无预取的缓存能够使缓存命中率提高 3%-5%。

第五章　一种开销感知的查询结果缓存替换算法

本章提出一种开销感知的查询结果缓存替换算法以达到节约后台处理查询时间的目的。通过按真实的搜索引擎日志模拟用户查询请求来验证此开销感知缓存替换算法的性能。

在过去的关于搜索引擎缓存如何调度存储什么内容的研究中，未命中缓存的代价往往被忽略，计算所有的查询请求都被假设有相同的代价，并使用缓存命中率来评估缓存策略。从 CPU 处理时间、网络要求等方面来看，向搜索引擎提交的查询的成本开销差异很大。因此假设所有的缓存未命中带来同样的开销是不现实的，而且被提交给搜索引擎的查询的计算代价显著不同。计算查询结果的时间开销主要包括从磁盘上读取倒排表的时间开销，计算倒排表交集的时间开销，读取文档的时间开销和生成网页摘要的时间开销。由于倒排表的长度各异，所以读取倒排表的时间各不相同。由于查询的词项个数不同，各词项的倒排表长度也不一样，所以计算倒排表交集的开销各不相同。由于各查询的相关性排名前 k 的文档数量和长度不同，所以读取文档的开销差异很大，生成网页文档摘要的开销也各不相同。这此使得查询的处理时间各不相同而且只与本身的词项构成有关而与查询在日志里出现的频率，顺序等无关。

在开销感知的查询结果缓存替换算法方面，2009 年，Ismail Sengor Altingovde[27]等人提出了针对于静态缓存的开销感知策略，能够比基于频率的静态缓存获得更好的性能，但仍不能较动态缓存替换算法得到更好的性能。2011 年，Rifat Ozcan 与 Ismail Sengor Altingovde[29]等人提出了 LFCU 缓存替换策略在 LFU 上加入时间开销感知，获得比 LFU 更好的性能。然而 LFU 本身应用于查询结果缓存的性能更 LRU 差，而改进的 LFCU 缓存在节约查询开销方面与 LRU 相差不大，在缓存容量小时，性能不如 LRU，而在缓存容量大的时候性能优于 LRU。本文提出一种兼顾查询开销和最近使用信息的开销感知的查询结果缓存替换算法。实验表明，它在缓存容量变化的情况上，性能始终优于 LRU 缓存替换算法。

5.1 开销感知的缓存替换算法

LRU 替换算法在查询结果缓存上获得较好的性能。尤其应用于动态静态混合查询结果缓存上时，由静态缓存来利用查询频率信息，LRU 能充分利用最近使用信息。本文希望在 LRU 算法上加入查询开销因素，从而减少查询计算总时间开销。然而基于链表和哈希表的 LRU 实现中，难以加入查询开销因素。本节介绍一种与 LRU 等效的缓存替换算法，维护一个小根堆使堆顶的元素与 LRU 的最近最少使用的缓存单元一致。

本节设置一个计数器seq，初始化为0。若查询请求未命中缓存：将查询请求的结果存入缓存，并为此缓存中的查询附带一个最近使用权重w为seq,同时seq自增1个单位。若查询命中缓存：将缓存中查询最近使用权重w更新为seq,同时seq自增1个单位。可以看到缓存中上次使用时间离现在最远的查询的最近使用权重w最小，将缓存里的查询按最近使用权重w放在小根堆中，则w最小的最近最少使用单元放在堆顶。用x表示请求的页，x.w表示最近使用权重, c表示缓存容量大小，当前缓存已存查询结果条数cn，L表示以查询为键的结果缓存。算法伪代码如图28所示。

```
初始化seq = 0，小根堆H为空
Case I. x∈L(命中缓存):
    x.w = seq
    seq += 1
    按x.w调整x在小根堆H中的位置
Case II. x∉ L(未命中缓存):
    Case (i). cn = c:
        移出H的堆顶元素
        在缓存中移除堆顶元素的查询结果
    将x的查询结果放计算出来放入缓存
    x.w = seq
    seq += 1
    把x按x.w加入小根堆H中
```

图 28　等效 LRU 的一种替换算法

本文加入查询开销因素，使查询开销高的查询的初始最近使用权重更大。对于缓存中的查询请求 x，设查询计算时间为 cost(q)。本文用 w(q)=k*cost(q)+seq。其中参数 k 应该与最大查询开销成反比，与缓存容量成正比。在本文的实验里 k 为缓存容量除以最大查询开销，将所有查询请求的最近使用权重同时缩小 k 倍 w(q) = cost(q)+ seq/k 可以便于计算得。用 x 表示请求的页，x.w 表示最近使用权重, c 表示缓存容量大小，当前缓存已存查询结果条数 cn，L 表示以查询为键的结果缓存，查询 x 的计算开销记为 cost(x)。查询开销感知的缓存替换算法如图 29 所示。

初始化seq = 0，小根堆H为空，inc = 最大查询开销/缓存容量。

Case I. x∈L(命中缓存):

x.w = seq + cost(x)

seq += inc

按x.w调整x在小根堆H中的位置

Case II. x∉ L(未命中缓存):

Case (i). cn = c:

移出H的堆顶元素

在缓存中移除堆顶元素的查询结果

将x的查询结果放计算出来放入缓存

x.w = seq + cost(x)

seq += inc

把x按x.w加入小根堆H中

图 29 一种查询开销感知的缓存替换算法

5.2 模拟实验

5.2.1 实验目的

按真实的搜索引擎日志模拟用户请求，对比查询开销感知的缓存替换策略和 LRU 缓存替换策略的性能。

5.2.2 实验设置

本章使用搜狗 2011 年 12 月 30-31 日两天查询日志来模拟用户查询。用第一天的查询预热结果缓存，而不计其请求次数和缓存命中率，用第二天 12 点之前的查询记录来测试缓存性能。本章将缓存容量从 50 万条查询结果逐步提高至 400 万条查询结果，记录缓存命中率和查询处理总时间。

5.2.3 实验环境

1) 硬件环境：

- CPU：AMD Athlon(tm) X4 750 Quad Core Processor　3.40 GHz
- 内存：4.00 GB
- 硬盘 : 1T

2)　软件环境：

- 操作系统：Win7 64 位操作系统
- Python 环境 ： Python 2.7.6

3)　数据集：

- 搜狗 2011 年 12 月 30-31 日两天的查询日志

5.2.4 实验结果

如表 5 所示，查询开销感知的缓存替换算法的缓存命中率略小于 LRU 的缓存命中率，但相差不到千分之一。所以查询开销感知的缓存替换算法很大程度保留了 LRU 的在缓存命中率上的贡献。

表 5　查询开销感知的缓存替换算法 LRU 的缓存命中率对比

缓存容量	开销感知算法缓存命中率	LRU 缓存命中率
500000	0.701094821	0.70179941
1000000	0.731891029	0.732545405
1500000	0.749690224	0.750312926
2000000	0.762349069	0.762931277
2500000	0.772163822	0.772699418
3000000	0.78023681	0.780717155
3500000	0.787437656	0.78791836
4000000	0.793739183	0.794179573

图 30 显示了在缓存容量由 50 万至 400 万条查询时，LRU 和查询开销感知的缓存替换算法的搜索引擎系统处理查询消耗的 CPU 时间。查询开销感知的缓存替换算法在缓存容量变化的情况下始终能节约更多的查询处理时间，它的查询处理总耗时比 LRU 算法少约 1.5%。

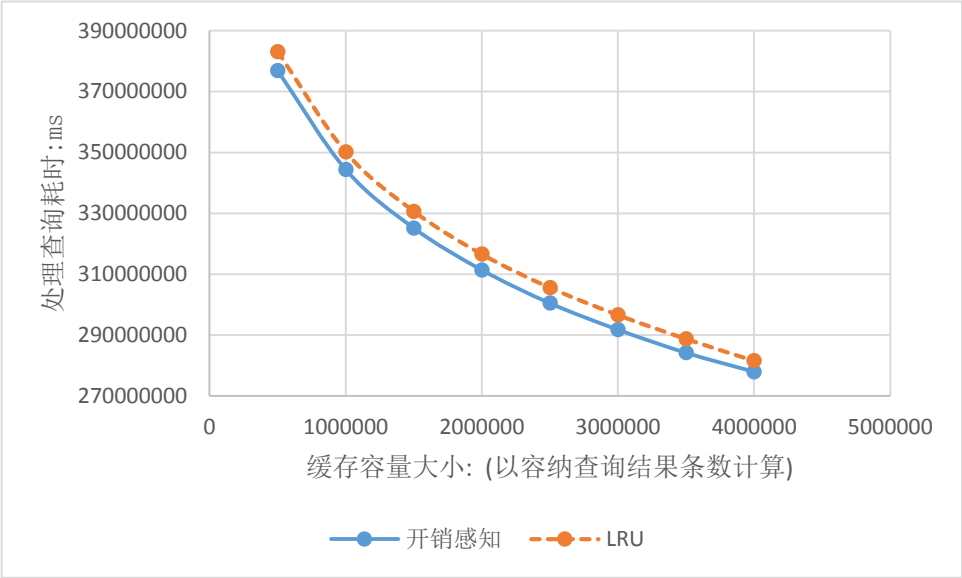

图 30　处理用户查询总耗时对比

在查询开销感知的缓存中，对于查询开销相同，而最近使用时间不同的查询结果 r1 和 r2，假设 r1 的最近使用时间早于 r2，且 r1 与 r2 之间相隔 b 个查询请求。在 r1 最近一次被访问时, r1.w = seq + cost(r1)。到 r2 最近被使用，seq 自增了 b 次。r2.w = seq + b* 最大查询开销/缓存容纳查询结果条数 + cost(r2)。由 r1 与 r2 结果开销相同知 r1.w 小于 r2.w。于是最近使用时间较早的 r1 先被逐出缓存，这与 LRU 算法一致。所以此查询开销感知的缓存替换算法能与 LRU 有着相近的缓存命中率。然而当两个查询结果 r1 与 r2 最近使用时间相近，而查询开销不同时。r1 与 r2 的最近使用权重相对大小由 r1 与 r2 的查询开销大小。查询开销越小，最近使用权重就越小，于是缓存先逐出查询开销小的查询结果。而对查询开销小的查询结果请求未命中缓存导致的计算耗时增加不大，因为查询开销少，后台计算这种查询结果很快。因此查询开销感知的缓存算法能够节省总查询耗时。

5.3　本章小结

本章提出一种开销感知的查询结果缓存替换算法，在查询计算开销相同的情况下能保持与 LRU 缓存替换算法一致。实验表明查询开销感知的缓存替换算法很大程度保留了 LRU 的在缓存命中率上的贡献，同时查询开销感知的缓存替换算法在缓存容量变化的情况下始终能显著地节约更多的查询处理时间。

总结与展望

论文总结

本文展示了在搜索引擎查询结果缓存方面的国内外研究现状和相关理论与技术。针对倒排索引更新慢而缓存容量有限的缓存，本文研究了传统的查询结果缓存策略，假设所有的缓存未命中带来同样的开销，在静态缓存与动态缓存混合的 SDC 框架上，探讨动态缓存替换策略对 SDC 查询结果缓存命中率的影响。实验结果表明，采用 ARC 替换策略能够使 SDC 缓存达到最高的缓存命中率。随着硬件性能的提高和价格的降低，廉价的存储设备容量大到足够存储所有过去用户查询的结果。在缓存容量无穷大的假设下，除了特定查询的第一次发起，其它的查询都能由缓存提供答案。一个无穷大的缓存面临的关键问题是缓存结果陈旧的问题。针对倒排索引更新快而容量足够大的缓存，本文提出了一种基于历史查询的缓存预取策略用一定的后台计算资源来预取更多查询结果更新结果缓存，从来减少查询结果陈旧失效更来的负面影响，提高了有效缓存命中率。为了进一步提高效率，本文还提出了一种开销感知的查询结果缓存替换算法，实验表明开销感知的缓存替换算法很大程度保留了 LRU 的在缓存命中率上的贡献，同时开销感知的缓存替换算法在缓存容量变化的情况下始终能显著地节约更多的查询处理时间。

工作展望

本文在以后的工作中将开展更深入的研究。对基于历史查询的结果缓存预取，在用回归预测的方法获取查询结果预取候选时，选取更多更好的属性使用更好的回归预测方法使预测的查询下次发起时间更精确从而提高预取的准确度，提升缓存的性能。评价缓存性能以节约的查询处理 CPU 时间作为指标，对候选查询的排序方式需要进一步探索。

此外，开销感知的查询结果缓存替换算法能够取代 LRU 作为动态静态结合缓存 SDC 中的动态缓存替换策略，同时用 Ismail Sengor Altingovde[27]等人提出的查询开销感知的静态缓存策略作为 SDC 中静态缓存的缓存策略，后续的工作验证查询开销感知的缓存替换算法应用于 SDC 时能够在性能上优于应用传统替换策略的 SDC 缓存。

参考文献

[1] China internet Network Information Center (CNNIC). The 34th Statistical Report on Internet Development in China[EB/OL], http: //www.cnnic.net.cn/hlwfzyj/hlwxzbg/hlwtjbg/201407/P020140721507223212132. pdf, 2014-07-21

[2] Cockburn A, Jones S. Which way now? Analysing and easing inadequacies in WWW navigation[J]. International Journal of Human-Computer Studies, 1996, 45:105-129

[3] Tauscher L, Greenberg S. How people revisit web pages: Empirical findings and implications for the design of history system. International Journal of Human-Computer Studies[J], 1997, 47:97-137

[4] Silverstein C, Henzinger M, Marais H,et al. Analysis of a very large web search engine query log[A]. SIGIR[C], 1998:6-12

[5] Baeze-Yates R. Web Usage Mining in Search Engines[M]. Reading MA: Addison Wesley, 1999: 307-322

[6] Hongyuan Ma, Wei Liu, Bingjie Wei et al. PAAP: prefetch-aware admission policies for query results cache in web search engines[A]. SIGIR[C], 2014: 983-986

[7] Xiao Bai, Flavio Paiva Junqueira, Adam Silberstein. Cache refreshing for online social news feeds[A]. CIKM[C], 2013: 787-792

[8] 马宏远，王斌.基于日志分析的搜索引擎查询结果缓存研究[J].计算机研究与发展, 2012, S1(S1):224-228

[9] E. Markatos. On caching search engine query results[J]. Computer Communications, 2001, 24(00):137-143

[10] D. Puppin, F. Silvestri, R. Perego, et al. Load-balancing and caching for collection selection architectures[A]. INFOSCALE[C], 2007: 1-10

[11] Xie Y, O'Hallaron D R. Locality in search engine queryies and its implications for caching[A]. IEEE[C], 2002: 1238-1247

[12] R. Lempel, S. Moran. Predictive caching and prefetching of query results in search

engines[A]. WWW[C], 2003: 19-28

[13] Ricardo A. Baeza-Yates, Flavio Junqueira, Vassilis Plachouras, et al. Admission Policies for Caches of Search Engine Results[A]. SPIRE[C], 2007: 74-85

[14] Patricia Correia Saraiva, Edleno Silva de Moura, Rodrigo C. Fonseca, et al. Rank-Preserving Two-Level Caching for Scalable Search Engines[A]. SIGIR[C], 2001: 51-58

[15] Xiaohui Long, Torsten Suel. Three-Level Caching for Efficient Query Processing in Large Web Search Engines[J]. World Wide Web, 2006, 9(4): 369-395

[16] R. Baeza-Yates, A. Gionis, F. Junqueira, et al. Design trade-offs for search engine caching[J]. ACMTWEB, 2008, 2(4):1-28

[17] Gleb Skobeltsyn, Flavio Junqueira, Vassilis Plachouras, et al. ResIn: a combination of results caching and index pruning for high-performance web search engines [A]. SIGIR[C], 2008: 131-138

[18] Zhang J, Long X, Suel T, et al. Performance of compressed inverted list caching in search engines[A]. WWW[C], 2008: 387-396

[19] S.Jonassen, B.B. Cambazoglu, F. Silvestri. Prefetching query results and its impact on search engines[A]. international ACM SIGIR conference[C], 2012: 631-640

[20] Roi Blanco, Edward Bortnikov, Flavio Junqueira, et al. Caching search engine results over incremental indices[A]. SIGIR[C], 2010: 82-89

[21] Berkant Barla Cambazoglu, Flavio Paiva Junqueira, Vassilis Plachouras, et al. A refreshing perspective of search engine caching[A]. WWW[C], 2010: 181-190

[22] Xiao Bai, Flavio Paiva Junqueira. Online result cache invalidation for real-time web search[A]. SIGIR[C], 2012: 641-650

[23] Fethi Burak Sazoglu, Berkant Barla Cambazoglu, Rifat Ozcan, et al. Strategies for setting time-to-live values in result caches[A]. CIKM[C], 2013: 1881-1884

[24] Sadiye Alici, Ismail Sengör Altingövde, Rifat Ozcan, et al. Timestamp-based result cache invalidation for web search engines[A]. SIGIR[C], 2011: 973-982

[25] Pei Cao, Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms[A]. USENIX Symposium on Internet Technologies and Systems[C], 1997: 193-206

[26] Rifat Ozcan, Ismail Sengör Altingövde, Berkant Barla Cambazoglu, et al. A five-level static cache architecture for web search engines[J]. Inf. Process. Manage, 2012, 48(5): 828-840

[27] Ismail Sengör Altingövde, Rifat Ozcan, Özgür Ulusoy. A Cost-Aware Strategy for Query Result Caching in Web Search Engines[A]. ECIR[C], 2009: 628-636

[28] B.B. Cambazoglu, R. Baeza-Yates. Scalability challenges in web search engines[J]. The Information Retrieval Series, 2011, 33:27-50

[29] Rifat Ozcan, Ismail Sengör Altingövde, Özgür Ulusoy. Cost-Aware Strategies for Query Result Caching in Web Search Engines[J]. TWEB, 2011, 5(2): 9-18

[30] T. Fagni, R. Perego, F. Silvestri, et al. Boosting the performance of Web search engines. Caching and prefetching query results by exploiting historical usage data[J]. ACM TOIS, 2006, 24(1):51-78

[31] Mauricio Marin, Veronica Gil-Costa, Carlos Gomez-Pantoja. New caching techniques for web search engines[A]. ACM International Symposium on High Performance Distributed Computing[C], 2010: 215-226

[32] D. Puppin, F. Silvestri, R. Perego, et al. Tuning the Capacity of Search Engines: Load-driven Routing and Incremental Caching to Reduce and Balance the Load[J]. TOIS, 2010, 28(2):890-895

[33] L. A. Barroso, J.Dean, U. Holzle. Web search for a planet: the Google cluster architecture[J]. IEEE Micro, 2003, 23(2):22-28

[34] 马宏远，王斌.一种基于预取感知接纳策略的查询结果缓存方法[J].计算机研究与发展, 2012, S1(S1):148-152

[35] 马宏远，王斌.一种基于查询特性的查询结果缓存与预取方法[J].中文信息学报,2011(5):37-43

[36] Baeze-Yates R, Gionis A, Junquerira F, et al. The impact of caching on search engines[A]. SIGIR[C], 2007:183-190

[37] Li Y, Zhang S, Wang B, et at. Characteristics of Chinese Web searching: A large-scale analysis of Chinese query logs[J]. Journal of Computational Information Systems, 2008,

4(3):1127-1136

[38] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schutze. Introduction to Information Retrieval[M]. Cambridge University Press, 2008: 1-482

[39] Jianguo Wang, Eric Lo, Man Lung Yiu, et al. The impact of solid state drive on search engine cache management[A]. SIGIR[C], 2013: 693-702

[40] S jiang, X Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance[J]. Acm Sigmetrics Performance Evaluation Review, 2002, 30:31-42

[41] G. Glass, P. Cao. Adaptive Page Replacement Based on Memory Reference Behavior[A], ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems[C], 1997: 115-126

[42] E. J. O'Neil, P. E. O'Neil, G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering[A], ACM SIGMOD Conference[C], 1993: 297-306

[43] J. T. Robinson, M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement[A], ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems[C], 1990: 134-142

[44] N Megiddo, DS Modha. Outperforming LRU with an adaptive replacement cache[J]. Computer, 2004, 37(4):58 – 65

[45] A.V. Aho, P.J. Denning, J.D. Ullman. Principles of Optimal Page Replacement[J]. ACM, 1971, 18(1):80-93

致　谢

最后感谢各位评阅老师抽出宝贵的时间对本文进行审评。不久就要结束自己在北航计算机学院的研究生生活，毕业并走向工作岗位。对在研究生阶段关心，帮助过我的老师和同学们，在此向他们表示感谢。

首先要感谢我的校内导师李建欣老师，是李老师接收我，使我开始我的研究生生涯。李老师对科学的渴望，探索精神，严谨的治学理念和孜孜不倦的工作态度令我无比敬佩。同时也深深地影响了我。感谢李老师对我在学术科研方面引领和指导，让我在项目实践方面得到锻炼，在学习生活方面的关心和照顾。在授予知识的同时，李老师也教会了我很多做人做事的道理，对我整个人生有着不同寻常意义。还记得我得我研一的时候偷偷玩游戏，是李老师严厉的批评使我意识到自己的错误。走向社会之后，我也会继续努力，用实际行动和工作成绩来回报李老师的教诲和信任。感谢 shutter 项目组的吴涛、刘昇丽、武南南、杨博睿、钟元等同学，感谢他们在项目工作中，学长学姐给我的指导，学弟学妹给我的支持和帮助以及营造出的良好的学习和工作氛围。

特别感谢我的校外导师王丽宏老师，感谢她在我读研期间给予我方方面面的帮助。在学术上她超凡的能力和开阔的思路，给我以榜样力量，她求真务实的精神，深深感染了我。此外，在很多学术研究和工作实践中，她都给了我细致入微的指导意见，祝愿她在工作生涯中能够成绩斐然，硕果累累。感谢我的校外导师马宏远老师，感谢他在百忙之中仍能抽出时间来给予我学术上的指导。感谢曹凯、杨雨龙、张雯裕同学一起度过充实和快乐时光。感谢高壮良、李宁、董元魁等网络信息安全班的同学的陪伴。

感谢 1306 大班的姚全营辅导员，以及 ZY13061 小班的班委，网络安全班班长李宁是他们对班集体的无私奉献，为我们创造了一个团结友爱的学习环境，为我们的学习和成长提供了保障。

感谢亲人，感谢父母对我的抚养和教育，感谢弟弟从小到大对我的关心和支持，在今后的时光里，我会更加努力，脚踏实地地做好自己的工作，为自己的小家庭和国家贡献自己的力量。

衷心地感谢所有帮助和支持我的老师和同学们！祝他们身体健康，学习工作顺利！