基于对等网络的多关键词搜索技术

邓宇善 2010-05

摘 要

对等网络和搜索引擎都是当前最受关注的研究领域，而对等搜索引擎则是一种试图将两者结合的创新技术，但目前该技术的实现仍然有较大的困难。因此，研究最基本的对等搜索技术 — 多关键词搜索显得尤为重要。为实现高效的多关键词搜索，本文将使用结构化对等网络的分布式散列表和基于关键词的索引划分方法，建立分布式的倒排索引结构。但由于原始的结构只对单个关键词进行索引，所以在处理多关键词查询时需要大量的文档集合传输和求交运算操作，增加了查询响应的延迟。为解决该问题，本文提出层次化的分布式索引结构，该结构以用户查询作为驱动，在多个层次上创建关键词集索引，从而提高查询命中率；并且，我们还引入了索引的删除机制，使索引规模总是保持较小。初步的实验结果显示，带删除机制的层次化索引结构可在原来 10% 的索引规模下减少超过 76% 的求交操作，这同时也意味着数据传输的减少。此外，本文还初步讨论了负载均衡和搜索排序的解决方案。最后，我们基于 Tapestry 结构化网络模型，给出多关键词搜索的原型系统及其应用程序接口，为后续的研究和开发提供必要的基础设施。

关键词: 对等网络；对等搜索；Tapestry 覆盖网络；分布式倒排索引；多关键词查询

目 录

第 1 章 绪论

计算机网络的诞生，给人类的生活和工作方式带来巨大的影响，且每一次的技术革新，都将影响推向更广更深。同样，对等网络 (Peer-to-Peer Networks，简称 P2P) 作为一种拥有巨大潜力的网络技术，被认为是“改变互联网 (Internet) 未来”的技术之一，在过去十年，它一直是计算机产业界和学术界关注的热点；相对于不太成熟的对等网络应用，搜索引擎技术在过去的十年已成为了互联网的主流应用，并开辟了新的商业模式。尽管如此，随着网络信息的继续膨胀和人们对搜索质量要求的不断提高，基于传统架构的搜索技术将遇到前所未有的挑战，而对等网络技术恰好为解决该问题提供了新的思路。在讨论具体解决方案之前，本章简单介绍对等网络和对等搜索技术的概念和进展。

1.1 对等网络概述

人们的需求往往是推动技术革新的原动力。自从万维网 (World Wide Web) 出现至今，客户端/服务器 (Client/Server，简称 C/S) 一直是网络应用的主流架构模式，这种模式固然有它的合理之处，然而，其缺陷也是明显的，例如，随着用户规模的膨胀、功能的不断扩展以及恶意攻击的增多，服务器最终会成为处理能力或安全性的瓶颈。为了进一步提高网络资源的利用率和避免单点 (服务器) 失效的问题，同时亦得益于客户端性能和 Internet 基础设施的不断提升，对等网络技术应运而生。

对等网络打破传统 C/S 的集中式架构，以平等的 (Equal)、自治的 (Autonomous) 实体 (对等端) 构成一个自组织的系统，其目的是在一个联网的环境中共享分布式的资源，避免中心化的服务 [1]。也就是说，参与网络的对等端既能使用资源又能提供资源，故不妨将对等端称为 Servent(Server+Client)。如果每个 Servent 都能提供一定的资源，那么网络规模的增大意味着可用资源的增多；另外，因为 Servent 是网络的主体，整个网络是没有中心服务器的自组织系统，这种去中心化的结果是避免了单点失效。因此，相对于 C/S 模式而言，对等模式具有更好的扩展性 (Scalability) 和可靠性 (Resilience)。

虽然对等网络技术最初只被设计为用于文件共享应用，如 Napster[2]、Gnutella[3] 和 BitTorrent [4] 等，而事实上，可以利用对等机制来共享任何类型的分布式资源，从而为传统网络应用提供新的可能性。除了文件共享，目前基于对等网络的应用领域主要有：多媒体传输、实时通信、协同办公、广域储存、分布计算和搜索引擎等，文献 [5] 列出了一份不完全但颇具代表性的 P2P 应用清单。

1.2 基于对等网络的搜索技术

1.2.1 基本概念

搜索引擎 (Search Engine) 是搜集互联网资源的利器，从上世纪 90 年代初的简单资源检索工具开始，发展到今天以超链接为基础的机器自动搜索引擎 [6]，并造就了 Google、百度等成功的互联网企业。根据中国互联网络信息中心 (CNNIC) 的统计，在不到 20 年时间，搜索引擎已成为与网络新闻和电子邮件并列的 3 大网络应用。为区别下文讨论的对等搜索技术，不妨把当前主流的搜索技术称为传统搜索技术，传统搜索引擎是采用 C/S 结构的集中式搜索引擎，此类搜索引擎的基本工作原理已经较为成熟稳定，其主要由三个部分构成：

- 收集信息 (Crawling);
- 索引信息 (Indexing);
- 搜索信息 (Searching).

这三个步骤都在中心服务器进行。根据 Google 的官方数据，2008 年 Google 索引的网页已超过 1 万亿个 [7]，而且互联网上的网页每年都已惊人的速度在增长，这对中心服务器的计算和储存压力都是巨大的。为了应对压力，搜索服务供应商通常采用分布式的服务器，整个系统是由遍布各地的多个服务器集群组成的分布式系统，可在一定程度上避免单点失效的问题。但部署和维护这样的系统成本高昂，因为数十万台的服务器和耗电量惊人的数据中心会远远超出大部分企业和团体的成本预算，而且这样的系

统并不能解决所有的问题。例如，系统必须先用网络爬虫 (Web Crawler) 抓取网页，然后索引 (Indexing)、排序 (Ranking)，这需要耗费一定的处理时间；而结果排名又取决于具体的搜索引擎排序算法。因此，难以确保总能获取到最新、最有价值的信息。

为此，研究人员提出对等搜索引擎 (P2P Search Engine) 技术，该技术是建立在对等网络架构上的搜索技术。在对等搜索系统中，每个参与的对等端既是客户端 (信息搜索者) 也是服务器 (信息提供者)，共享信息 (包括索引信息) 被分布到整个网络当中。显然，建立在此网络上的搜索系统可继承对等网络的优势 — 扩展性和可靠性；更重要的是，这两个好处的获得是廉价的，只需使用现有设施，无需另外投入巨大的建设和维护成本，或者说，成本被分摊到各个对等端上；另外，由于信息可由对等端主动提供并在本地索引，可省去了抓取信息的过程，所以信息的发布更主动、更即时。

1.2.2 主要挑战

虽然，对等搜索有众多的好处，但技术上并不成熟，目前主要还是处在理论研究阶段，是对等技术应用的一个重要研究课题，文献 [8][9] 概述了其面临的主要挑战，可总结为如下几点：

(1) 搜索效率：构成对等网络的大多数节点通常处于网络边缘，这些节点的计算、存储和网络带宽等资源往往比较有限，因此，设计简单有效的搜索算法是当前面临的主要挑战。

(2) 搜索排序：传统搜索引擎在逻辑上仍然是一个中心节点，搜索结果可以根据其文档集合的所有统计信息进行排序计算，但在对等环境中，搜索涉及多个节点，且节点具有较大的自治性和动态性，全局的统计信息难以收集，故传统的排序策略在对等环境中难以实现，设计合适的排序策略仍然有较大的困难。

(3) 负载均衡：Web 搜索中固有的“热点 (Hot spots)”问题在对等搜索中同样存在，与使用高性能服务器的传统搜索引擎相比，单个对等端的性能相对较低，故对等搜索的负载均衡显得尤为重要；另外，各个对等端的性能常常也有较大的差异，开发节点的异构性是负载均衡策略必须考虑的。

(4) 复杂查询：相对于键值查询 (单关键词查询) 这样的简单查询，用户更希望搜索系统能够支持多关键词查询、聚合查询，甚至 SQL(Structured Query Language) 查询、语义查询等的复杂查询，从而更准确地定位信息。目前，无论是传统搜索系统还是对等搜索系统，主要的研究都是针对多关键词查询 [8]，但相对于技术较为成熟的传统搜索技术，在对等网络中支持高效的复杂查询将会遇到更大的挑战。

(5) 激励机制：对等网络中的节点具有较高的自治性，除了必要的共享资源，节点没有贡献更多资源的义务，好的激励机制可以提高节点共享资源的积极性，从而有助于增强系统性能，但双赢的方案不容易实现。

(6) 系统安全：对等系统虽然能够在一定程度上避免单点失效的问题，但也存在其他的安全问题 — 如何避免节点遭受 DoS 攻击、如何对文件进行认证以保证数据的真实性、如何确保匿名性、如何进行访问控制以保护某些内容的版权等 [8]。

另外，在面对高度动态的对等环境以及海量信息时，搜索系统的算法设计必须考虑可靠性和可扩展性，以充分发挥对等网络的优势。

1.2.3 研究进展

在对各项关键技术的研究基础上，研究人员提出了一些对等搜索的原型系统，如 ODISSEA[10]、GALANX[11]、MINERVA[12]、Coopeer[13] 和 PlanetP[14] 等，详细介绍可参考其相关论文，或参考论文 [8] 的综述。大致上，这些系统可以分成两类，一类是基于结构化对等网络 (见 2.4 节) 的搜索技术 (ODISSEA，GALANX 和 MINERVA)，另一类是基于无结构对等网络 (见 2.3 节) 的搜索技术 (Coopeer 和 PlanetP)。两种技术都有各自优势和局限，能同时具有两种技术优点的系统目前仍不存在，因此，当前对两种技术的深入研究和比较都是相当有意义的，本文的研究属于前一种技术。

除了学术界，产业界亦颇关注对等搜索技术，不过由于缺乏完善的法律法规和成功的商业模式，大部分的商业机构都处于观望状态，只有极少数的企业进行了开发尝试。

早在 2002 年，美国新兴搜索引擎设计公司 i5 Digital 推出了其依据 P2P 理念开发的商业搜索引擎 Pandango[15]，根据论文 [16] 的介绍，用户安装 Pandango 的客户端

后，每次的搜索是通过递归地查询多个邻居节点来获取最终结果 (即基于无结构对等网络的搜索)，至于其响应时间和搜索质量，论文 [16] 并没介绍，现在其主页上也没再提供客户端的下载。

值得注意的是 2008 年在英国成立的 Faroo 公司推出的 Faroo 实时搜索 (Real Time Search)[17]，他们的目标是以低廉的成本获得更好的搜索结果。Faroo 是一个真正意义的对等搜索引擎，它将传统的搜索过程 (见 1.2.1 节) 转变为对等网络上的分布式计算：

- 分布式的爬虫 (Distributed Crawler);
- 分布式的索引 (Distributed Index);
- 分布式的排序 (Distributed Ranking).

从而提供更即时、更广泛和更适合用户的搜索结果。Faroo 使用 C# 开发，不公开源代码，目前只提供 Windows 平台下的版本，Mac OS 和 Linux 下的版本即将推出。由于使用人数不多，Faroo 的并未充分展现出其优越性，故用户的反映普遍不佳，尤其是响应速度。因此，减少响应延迟是他们当前的首要任务。

在开源社区中，Lucene 和 Nutch 在传统搜索技术上的成功引起了大家对开源搜索技术的关注，事实上，已有不少开源项目已在尝试对等搜索技术。如 GPU(Global Processing Unit) 搜索引擎项目 [18] 和 GRUB 项目 [19]，他们实现了分布式的爬虫技术，这有利于传统搜索引擎实现更广更深的资源搜集；除了爬虫，YaCy 项目 [20] 还实现了内容分析和管理功能，是完整的 P2P 搜索引擎，与 P2P 文件共享应用类似，YaCy 的对等端之间可以共享网页索引，从而避免审查。YaCy 使用 JAVA 开发，主要由四部分组成：网页爬虫，索引引擎，数据库引擎和网页形式的用户界面，其详细实现可参见开发文档和源代码。

1.3 本文研究内容及其意义

对等搜索系统是一个巨大而且复杂的系统，层次化的方法有利于降低系统设计和分析的复杂性，同时优化策略也可以在各个层次上考虑。与上述介绍的原型系统相比，本文研究的并不是系统的整体实现，而是专注于如何在结构化对等网络上实现高

效的多关键词搜索，并结合具体的结构化模型进行论述。本文的研究内容主要分成两方面：

(1) 底层对等网络架构的选择。对等网络发展至今，已经涌现出多种的体系结构，通常可分为三类：混合式对等网络 (2.2 节)、无结构对等网络 (2.3 节) 和结构化对等网络 (2.4 节)，本文第 2 章将讨论三类体系的特点。在本文提出的搜索技术中，总体结构采用结构化的对等网络，并以其快速的键值查询 (即简单查询) 服务作为多关键词搜索的基础；在局部上，为解决系统中可能出现的局部性负载过重问题 (或称为“热点”)，多个节点将结成共同承担较重任务的子系统，该子系统将使用无结构对等网络的组织和通信方式。

(2) 分布式的多关键词搜索策略。由于对等网络的特殊性质，如动态性、自治性和边缘性等，传统的多关键词搜索技术无法完整地移植到对等环境中，文档索引划分 (3.2.2 节)、文档集合求交 (3.3 节) 和排序 (3.6 节) 等关键技术都是本文需要讨论的；此外，用户查询的特征 (3.3.3 节) 也是本文的研究对象，一方面是因为查询特征将极大地影响到索引的划分方式和分布式索引的结构，另一方面，基于查询记录的索引策略可大幅提升搜索效率，所以，本文提出的搜索系统在很大程度上是以用户查询驱动的。

上述内容的划分是基于层次化的设计思想，本文将多关键词搜索系统建立在结构化对等网络层之上，使其可以独立于底层网络的实现细节。具体地，我们选择的是 Tapestry 对等网络 (见 2.4 节)，但也可容易地移植到其他遵循接口规范 (见论文 [21]) 实现的对等网络。另外，考虑到多关键词查询是较为基本的查询，可作为其他复杂查询的基础支持，因此有必要将多关键词搜索作为一项底层服务供上层使用。本文第四章介绍的原型系统展示了多关键词搜索的具体实现架构 (见 4.2 节)，并给出了简单的应用程序接口，方便后续的研究与开发。

本文的研究意义在于：对等搜索技术无需通过中心服务器，也不受信息文档格式和宿主设备的限制就能够深度搜索文档，且可达到传统搜索引擎 (只能搜索到 20%–30% 的网络资源) 无可比拟的深度 (理论上将包括网络上所有开放的信息资源)[5]。而其中多关键词搜索是其他更为复杂的搜索技术基础，故本文的研究对 P2P 搜索具有基础性的

作用。另外，因为对等网络系统本质上是一个纯分布式的系统，所以对等搜索也属于分布式搜索技术的一种，但与传统的分布式搜索相比，对等搜索将更具挑战性 — 动态性、负载平衡、局部性和异构性等问题将会十分突出，因此对等搜索系统解决的问题将对其他分布式系统的研究有重大的启发意义。

1.4 论文章节安排

本文的内容安排如下：第二章简单介绍主流的对等网络体系结构，且重点介绍本文搜索系统使用结构化体系；第三章讨论基于结构化对等网络的多关键词搜索技术，提出层次化的分布式倒排索引结构，并研究基于该结构的搜索策略和增强特性；第四章将综合前两章的技术，给出多关键词搜索的原型系统和基本应用接口；第五章总结全文和展望未来的研究工作。

第 2 章 对等网络的体系结构

自从第一个对等网络应用出现至今，已提出了多种的体系结构，本章介绍三类主要的对等网络体系：混合式对等网络、无结构对等网络和结构化对等网络。重点介绍本文对等搜索系统所使用的结构化对等网络体系，为后面的讨论提供基础。

2.1 对等网络概述

对等网络的思想起源较早，可以追溯到计算机网络诞生之初。基本而言，它开始于 20 世纪 60 年代后期 ARPANET 的建立 [1]，ARPANET 的目标是为了共享分布在美国各地的科研机构的计算机资源。在原始的系统当中，并无所谓服务器或客户端，所有主机的地位都是平等的，在这个意义上，ARPANET 具备了基本的对等网络特性；由于起初的网络规模不大，故整个网络的拓扑结构较为简单，通常由管理人员进行维护，也就是说，ARPANET 没有自组织性。经过多年的演变，ARPANET 发展为规模庞大的全球性 Internet，所以自组织性成为网络的必要特性。为了降低设计的复杂性，层次化的参考模型被广泛使用，如图 2-1 所示的 TCP/IP 参考模型，Internet 的自组织性通过互联网络层及其 IP 协议 (Internet Protocol) 实现。通过传输层接口，应用程序无需了解网络拓扑结构就可实现与 Internet 上任意节点的通信，此时可在逻辑上认为任意节点间都存在连接 (实际上可能是经过多跳的连接)。为了叙述方便，不妨将 Internet 称为物理网络 (Physical Network)，而将逻辑上形成的网络称为覆盖网络 (Overlay Network)。对等网络建立在应用层上，是多个共享资源节点所形成的覆盖网络，由于该网络通常具有较大的规模，所以自组织性也是该网络的必要特性。

虽然网络的分层模型可以给开发和分析带来方便，但物理网络与覆盖网络的分离却为对等网络的路由设计带来困难，如拓扑一致性 (或局部性) 的问题 (2.4.4 节)：在覆盖网络上相邻的节点不见得在物理网络上相邻。因此，覆盖网络组织方式一直是被反复研究的重要课题，研究者亦提出了不少的体系结构模型，按其出现的先后顺序可大概分为：混合式对等网络、无结构对等网络和结构化对等网络。下面简要回顾近十年来，对等网络体系结构的演变和发展。

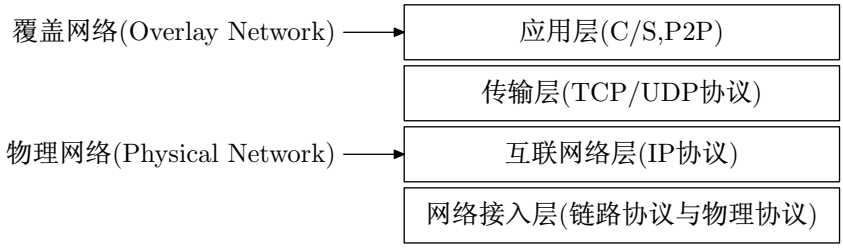


图 2-1 TCP/IP 参考模型.

2.2 混合式对等网络体系

集中式对等网络 (Centralized P2P Networks) 是最早的 P2P 应用模式。该模式多数使用 C/S 与 P2P 的混合结构，故又称混合式对等网络体系 (Hybrid P2P Architecture)。使用该模式的系统仍将面临扩展性和单点失效的问题，后来引入超节点的技术可在一定程度上缓解该问题。无论如何，该模式取得的商业成就是当前所有对等网络体系中最大的；同时该技术也可看作是连接 C/S 与 P2P 的桥梁。

1999 年，18 岁的 Shawn Fenning 开发了世界上第一个对等网络应用 Napster[2]，它是混合式 P2P 体系的典型代表，后来发展的 BitTorrent[4] 网络实际上是它的推广，相当于分散的多个 Napster 网络。Napster 的拓扑结构如图 2-2 所示，在该网络中，对等端通过向服务器递交查询 Q，然后从响应 R 中获得希望与之通信的对等端地址，于是接下来对等端之间的通信 (在 Napster 中就是文件的上传与下载) 就不再需要服务器的参与，从而实现对等通信。在 BitTorrent 中，其本身不提供查询机制，BT 网站维护大量的文件索引 (即后缀名为.torrent 的种子文件)，对等端通过种子文件以及在 Tracker 服务器的协助下与一个或多个文件拥有者获得联系，然后实现对等通信。本质上，BitTorrent 与 Napster 无异，因为 BT 网站和 Tracker 就是服务器；但作为后起之秀的 BitTorrent 采用了一些增强机制来提高网络效率，如分片机制 (Pipelining and Piece Selection) 和阻塞算法 (Choking Algorithms)[22] 等。

无论 Napster 还是 BitTorrent，网络仍然是以服务器为核心的星形拓扑结构。这种结构使得查询简单直接，查询路由在服务器集群或分布式系统的高速网络内进行；只要参与用户提供足够多的索引信息给服务器，实现更为复杂的查询或全文检索是可

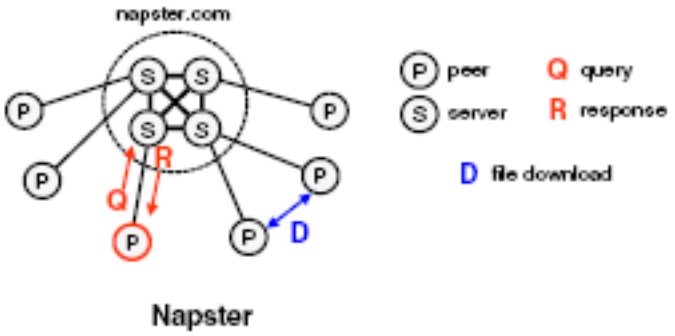


图 2-2 Napster 网络拓扑, 图片来自文献 [23].

能的，不过，这又将回到过去的 C/S 架构。克服 C/S 的缺点才是对等网络技术的目标，而且应该注意到的是，Napster 和 BitTorrent 取得的巨大成功更多是得益于对等网络技术，它们都以低廉的成本获得了高效的文件传输性能，并大大地丰富了网络资源。Napster 和早期的 BitTorrent 实现都未能完全突破传统架构，故仍需考虑服务器可用性的问题；另外，存放大量文件索引的服务器还可能面临诸如版权问题等法律纠纷，事实上，这正是导致 Napster 最终破产的原因。

混合式体系的进一步改进是将服务器层对等化，使得服务器的部署有更大的灵活性。所谓的服务器层对等化，其实就是开发节点的异构性，区别不同能力的节点，使能力强的节点处理更多、更重要的任务。这个观点认为，不妨将服务器也看成是对等网络中的节点，但因为这些节点拥有比一般节点 (Ordinary Node，ON) 多的资源，所以称为超节点 (Super Node，SN)，如图 2-3 所示。由超节点连接而成的子系统处理整个对等网络的大部分任务，这将极大地提升了效率，确保系统的性能。

基于 FastTrack 协议 [24] 的 KaZaA[25] 是最早开发网络异构性的 P2P 网络，它将网络节点分成超节点和普通节点两类。超节点通常有高的带宽、高处理能力、大储存容量和不受 NAT 限制，而普通节点则不具备这些优势。当普通节点加入 KaZaA 网络时，它会选择一个超节点作为其“父超节点”，并且与父超节点维持一条半永久的 TCP 连接，将它所共享的文件元数据信息上传给这个超节点 [5]。KaZaA 超节点的作用与 Napster 服务器的作用其实是一样的，不同的是，KaZaA 中的超节点不是专门的、永久的，它只是一个普通的拥有较高能力的网络节点而已，并经常会由 KaZaA 中的普

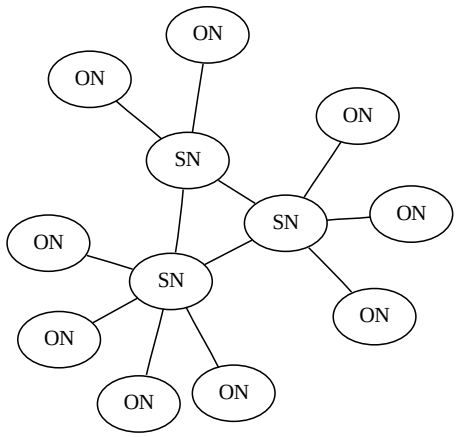


图 2-3 引入超节点的层次化覆盖网.

通节点转变过来 [5]。与 KaZaA 类似，Gnutella(在 2.3 节介绍) 协议 0.6 版、eDonkey(电驴) 以及其后继者 eMule(电骡) 都是采用节点分类的层次化组织方式，以开发异构性。

在 KaZaA 网络中，用户的查询首先发送到其父超节点上，超节点搜索自己的数据库，然后将匹配的信息所在节点 IP、服务端口号和元数据返回给用户；另外，由于每个超节点还会和其他的一些超节点保持着长期的 TCP 连接 [5]，故超节点还会将查询转送到与它相连的超节点。若要遍历所有的超节点，查询就得递归地转发，这就是 2.3 节介绍的洪泛算法；若查询不递归进行下去，查询只能发生在较小的超节点集里，因而具有局部性。KaZaA 采用了折衷的办法：对查询只作有限步的转发，此外还周期性地更新节点连接关系以缓解查询局部性的问题 [5]。

在实际中，超节点通常都是商业机构提供的服务器，因此企业掌握了该对等网络的关键设施，这也是混合式对等网络较易实现商业化的原因，但这并不是对等网络要实现的最终目标，故对等网络的研究并没有因此而停止，更为纯粹、自由和开放的对等网络不断涌现，下面介绍的无结构和结构化对等网络就是典型的代表。

2.3 无结构对等网络体系

上面介绍的混合模式实际上是 C/S 到 P2P 的过渡，在 Napster 出现之后的很短时间内，纯粹对等网络 (Pure P2P Networks) 就出现了。第二代的对等网络通常被称为无结构的对等网络，而所谓的“无结构”，指的是覆盖网络组织的不确定性，资源在

网络上的位置并不遵循特定的结构。实质上，无结构 P2P 网络是用户自发形成的一个随机网络。有研究表明，从统计意义上看，该网络的连接方式大致上符合小世界模型 (Small World)[26] 或幂律分布 (Power Law Distribution)[27][28] 的。

第一个无结构对等网络 Gnutella[3] 诞生于 2000 年，由 NullSoft 公司开发，Gnutella 是最简单、最有代表性的无结构对等网络，其拓扑结构如图 2-4 所示，在原始的模型中，当对等端要查询未知资源时，可使用洪泛式 (Flooding) 的路由方法：每个收到查询消息的节点将消息转发给除上一跳节点外的所有邻居节点，直到消息到达目标节点或跳数限制参数 TTL(Time to live) 用完为止。洪泛式的路由方法简单直接，但缺陷却是显而易见的，指数式增长的查询消息会耗费大量网络带宽，尽管 TTL 对此问题会有一定的帮助，但却又大大限制了查询范围，造成查询局部性，故洪泛式路由很少应用到实际当中。

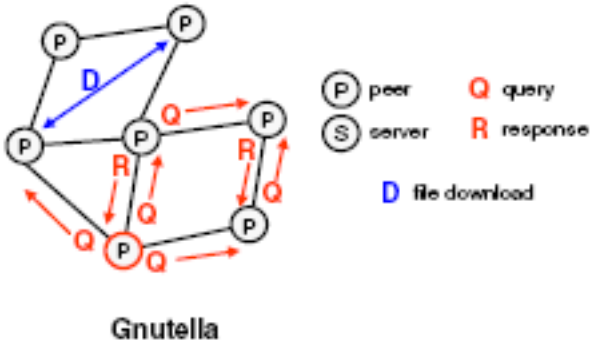


图 2-4 Gnutella 网络拓扑, 图片来自文献 [23].

为提高查询路由的效率，0.6 版的 Gnutella 将网络组织成类似 KaZaA 的层次化覆盖网络 (见 2.2 节)，相互连接的超节点构成覆盖子网络。当超节点数目较少时，则在超节点网络中使用洪泛式路由查询还是可行的，否则，就需要使用启发式方法提高查询命中率，如下面将要介绍的 Freenet。

虽然无结构对等网络的查询效率不高，但松散的结构却体现自由平等的精神，它可以避免网络资源的垄断和管制。与 Gnutella 同年，倡导自由、安全、匿名的无结构对等网络 Freenet 发布第一版 (其概念早在 1999 年就被提出 [5])，该网络具有复杂的安全机制确保信息不受审查，所以 Freenet 在大多数国家受到严格限制。在 Freenet 当

中，查询路由采用“深度优先”的搜索策略：每个收到查询的节点将先检查自己是否拥有查询信息，有则返回，否则将查询转发到它认为最好的下一跳节点，由下一跳的节点继续完成查询；但如果无法发送请求给下一跳节点 (可能该节点已不在线，或如果发送消息给它将会导致循环路径)，则该节点会尝试第二好的节点，若再不行，再尝试第三好的，如此类推，直到成功后将结果返回上一跳节点，或直到所有尝试都失败则将失败信息返回上一跳。若不使用跳数限制，Freenet 的查询在最坏情况还是跟简单洪泛一样要遍历所有的节点，但每次选择最优下一跳的启发式策略常常可避免最坏情况的发生。另外，Freenet 的这种查询方式还会使用一些加密的策略，使得路径上的节点只知道其上一跳和下一条节点，而对路径上的其他节点一无所知，这种安全的“隧道路由”是 Freenet 匿名性的核心 [5]。

从以上的介绍可以看到，无结构对等网络的组织方式是松散的，节点间的连接关系完全按照随机的方式或用户选择而形成，这种组织方式有其独到的优势：

- 由于不需要保持特定的结构，网络的维护开销降低了；
- 网络的灵活性较大，从而获得良好的容错性和自适应性；
- 拓扑结构的不确定性可以带来非常高的安全性和匿名性。

但是，从上面介绍的 Gnutella 和 Freenet 的查询路由方式可以看到，它们的路由效率都不高，又或者具有随机性，这个问题却恰恰又是其“无结构”的特性所带来的，尽管该特性也为它带来上面的三个优点。路由效率的低下导致了可扩展性不高，提高路由效率的算法往往又使得查询具有局部性，而局部性又可能会导致查询无法准确定位。这一连串的问题再次引发人们对 P2P 网络体系结构的重新思考，最终导致下一节介绍的结构化对等网络体系的诞生。

2.4 结构化对等网络体系

在混合式和无结构对等网络体系相继取得成功后，学术界开始真正关注和重视 P2P，许多的知名学术团体和研发机构成立专门的 P2P 研究组，他们提出了几个 P2P 领域最具代表性的经典模型：MIT 和 U. C. Berkeley 的 Chord[29]、U. C.

Berkeley 和 AT&T 的 CAN[30]、Microsoft Research 和 Rice Universty 的 Pastry[31] 以及 U. C. Berkeley 的 Tapestry [32][33]。它们都是结构化对等网络体系的杰出代表，而且都在 2001 年提出，所以 2001 年是结构化对等网络的起点；同时，结构化对等网络体系的研究掀起了 P2P 研究的热潮，因此 2001 年也是 P2P 发展史上重要里程碑。除了刚才提到的四个最著名的结构化模型外，在它们之后，还有很多优秀的模型被提出：Kademlia[34](2002 年)、Viceroy[35](2002 年)、Koorde[36](2003 年)、Cycloid[37](2004 年) 等，它们各自在理论上或实践上取得成功。

本文的讨论将基于 Tapestry 结构化对等网络模型，Tapestry 是由加州大学伯克利分校 (U. C. Berkeley) 的 Ben Y. Zhao 等人设计开发的一个面向广域分布式数据存取、容错的超立方体结构对等网络模型。在四大经典模型中，Tapestry 和 Parsty 是类似的，它们的设计都基于 Plaxton 网 (在 2.4.1 节介绍)，使用前缀逐位匹配的路由算法 (在早期的设计中，Tapestry 使用的是后缀匹配 [32])；其次，它们在构建覆盖网络时就考虑到拓扑一致性问题 (见 2.4.2 节介绍)，故路由具有较好的局部性 (见 2.4.4 节)，而 Chord 和 CAN 在建网时则没有考虑，只能在后期采用辅助的方法来提高局部性。Tapestry 和 Pastry 的不同之处在于数据对象的放置：Tapestry 帮助用户寻找最近的数据副本，而 Pastry 则通过主动复制数据、将副本随机地存放在网络中来提高数据可用性，并期望通过复制来使用户获得较近的副本 [5]。早期的 Tapestry 使用 JAVA 开发，直到 2004 年发布最后的版本 2.0.1 为止，开发者在 04 年的论文 [33] 对 Tapestry 作了较多的改进，新的模型在路由表中多加入了叶集 (Leafset，是从 Pastry 中借鉴过来的)，使路由更简单和健壮；新的实现也不再使用 JAVA，而是使用更高效的 C，并改名为 Chimera[38]，目前的最新版本是 1.20。

之所以使用 Tapestry 作为讨论的基础，主要是由于其实现 Chimera 严格遵守论文 [21] 提出的 API 规范，尽管该规范并不具有权威性和最终性，但其努力是值得肯定的，因为统一的接口可以鼓励上层应用的开发，开发者可以轻松地移植应用程序到合适的底层结构化网络；反过来，大量的应用可以使底层模型得到大量的测试和比较，促使模型设计更加合理。除此之外，Tapestry 对局部性的充分考虑和 Chimera 的高效实现对加快对等搜索速度都是十分有用的。在下面的介绍中，我们将包含多数的结构化对等网络模型，且详细介绍 Tapestry。

2.4.1 覆盖网络拓扑结构

结构化对等网络的最大特点是它们都有一个严格的覆盖网络拓扑结构，这是与无结构对等网络的本质区别，以下简单介绍结构化对等网络的几种主要拓扑结构，详细的介绍可参考相关论文或文献 [5]：

(1) 带弦环结构：所有的节点被组织在一个环上，环只提供两个基本功能 — 取得当前节点的前驱和后继节点，整个网络结构的正确性依赖于前驱或后继关系的正确性。为了加快查询路由，每个节点的路由表还维护一些离自己较远的节点，在 Chord 中称这些节点集合为指取表 (Finger Table)，这就相当于在环上加入“弦”。在 Chord 和 Kademlia 中，这些弦指向的节点与自己在环上的距离是指数间隔的。

(2) 多维空间结构：所有的节点组织在一个 d 维的笛卡儿空间里，每个节点管辖空间里的某个区域，并在路由表中维护自己在空间中的邻居。CAN 是采用多维空间结构的典型代表。

(3) 超立方体 (或 Plaxton Mesh[39]) 结构：采用此结构的有 Tapestry、Pastry 和它们的前驱 Plaxton。实际上，它们并不是严格的超立方体结构，它们采用的是 Plaxton Mesh 的拓扑结构，路由表中的每一层 (level) 维护与当前节点标识 (ID) 匹配一定长度前缀 (或后缀) 的节点集合，因为其基于前缀 (或后缀) 逐位匹配的路由方式 (2.4.2 节会详细讲解) 与超立方体路由类似 — 两个节点间可存在多条不同的路径，所以称为超立方体结构。而 Cycloid 则是综合带弦环和超立方体的优点，形成了每个节点都具有常数度的“带环超立方体”(Cube Connected Cycles，CCC) 结构。

(4) 蝴蝶形：每个节点都有它的“层”，每层的节点通常维护两个下边、一个上边和两个同层边。Viceroy 是采用蝴蝶结构的代表，除了基本的蝴蝶网外，每个 Viceroy 节点还保存一个前驱和一个后继的地址信息，从而形成一个全局的环结构。

(5)de Bruijn 图：所谓的 de Bruijn 图，就是图中的每个标识为 m 的节点有两条出边，一条指向节点 $2m(\mathrm{mod}\ 2^b)$，一条指向节点 $2m+1(\mathrm{mod}\ 2^b)$，假设 b 是节点标识的二进制长度。Koorde 在 Chord 环的基础上引入 de Bruijn 图来加速路由。

2.4.2 查询路由

为了实现准确、高效的查询路由，除了严格的拓扑结构以外，还要为覆盖网上的每个节点赋予全局唯一的标识 (Node ID)，通常可以由安全的一致散列函数 (如 SHA-1，MD5 等) 作用在节点的某些属性 (如 IP 地址和端口号等) 上得到，ID 值通常取 128 位 (如 CAN 和 Pastry) 或 160 位 (如 Chord 和 Tapestry)，因此标识符空间是足够大的；同样，使用一致散列函数将数据对象映射到同样的标识符空间中，称为键值 (Key) 或对象标识 (Object ID)。因为节点的动态性 (见 2.4.3 节)，系统动态地为每个数据对象分配一个负责其索引的节点，该节点称为该对象的根 (Root)：如果当前网络中恰有一个节点的标识与对象的键值相等，那么它就是对象的根；若不存在恰好相等的节点，则选择标识符与键值最接近的作为根。结构化对等网络体系提供的查询服务称为基于键值的查询路由 (Key-Based Routing, KBR[21])，而将对象索引映射到覆盖网络的技术则称为分布式散列表 (Distributed Hash Table)，高效的 KBR 服务是 DHT 的实现基础。

为实现高效的 KBR，网络中的每个节点需要维护一个不太大的路由表，并采用分布式、局部性的贪心路由算法，逐步缩小当前节点 ID 值与查询目标键值之间的差异。不同的结构化对等网络模型采用的具体路由定位方式会有所不同，通常取决于其采用的覆盖网络拓扑结构和路由表结构。另外，值得注意的是，在计算节点标识符与键值的接近程度时，度量方式是可以有多种的，除了按数值差的绝对值大小 (如 Chord) 外，还可以按空间位置的距离大小 (如 CAN)，按前缀或后缀逐位匹配的长度 (如 Pastry 和 Tapestry)，按异或值的大小 (如 Kademlia) 等。下面具体介绍 Tapestry 模型的路由方式。

Tapestry 路由定位

上小节介绍过，Tapestry 是基于 Plaxton 网的，因此路由定位机制与 Plaxton 类似，但比 Plaxton 更容错和优化，因而能适应动态的网络环境。在 Tapestry 中，除了节点和数据对象有全局唯一的标识 GUID(Globlly Unique ID) 外，每条消息都有其特定应用的 AID(Application ID)，用来区分应用的类型，这与 TCP 协议中端口号的

作用是类似。Tapestry 采用逐位匹配前缀的路由方式，每一跳匹配更多位的前缀，例如，GUID 有 4 位，采用 16 进制，则路由到节点 42AD 的匹配过程为：

$$4*** \Longrightarrow 42** \Longrightarrow 42A* \Longrightarrow 42AD \qquad (* \text{ 号表示通配符})$$

为适应前缀匹配路由，每个 Tapestry 节点维护一个层次化的路由表。表中的第 i 层包含与该节点 ID 恰好匹配前 $i-1$ 位的节点信息，而且这些节点与当前节点在物理网络上应该是尽可能邻近的；另外，在第 i 层中，第 j 项是 ID 值第 i 位等于 j 的节点。例如，在节点 325AE 路由表中，第 4 层第 9 项为 ID 值以 3259 开头且与节点 325AE 通信延迟较小的节点。或者换个角度看，如图 2-5 所示，节点指向满足一定前缀规则的邻居，并都是与自己的通信延迟较小的节点。由此可见，Tapestry 在构建覆盖网时就考虑到拓扑一致性问题。事实上，节点的路由表还会在运行中不断地修正和优化，以保持良好拓扑一致性 (2.4.4 节)。

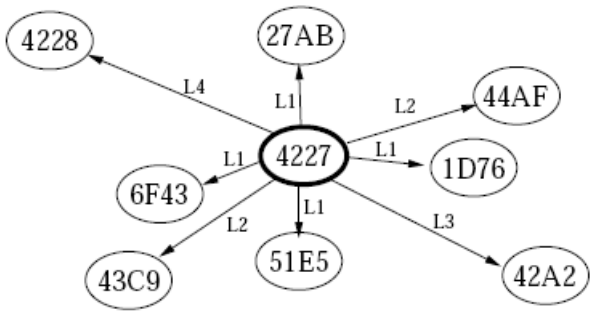


图 2-5 节点 4227 的邻居图，边上的 Li 表示所指向的节点位于路由表的第 i 层, 图片来自文献 [33].

在路由过程的第 n 跳，节点与目标 ID 至少匹配 n 位的前缀，这时可从路由表中的第 $n+1$ 层获得下一跳节点，该方法确保在 $\log_\beta N$ 跳内到达覆盖网上的任何节点，其中 β 是 ID 采用的进制数，N 是网络的节点总数。图 2-6 为从节点 5230 到 42AD 的路由过程。

在 Tapestry 的最新实现中，借鉴了 Pastry 的优点，加入了叶集的路由信息。叶集中包含与当前节点 ID 值最近邻的节点信息，并分为两部分，一部分的节点值是比当前节点的小，另一部份则比当前节点的大。当有消息到达并需要被路由时，节点首先检

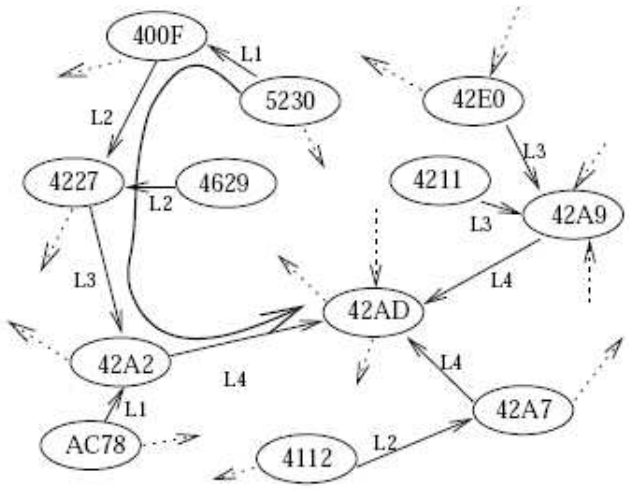


图 2-6 Tapestry 覆盖网中从节点 5230 到节点 42AD 的路由路径, 图片来自文献 [33].

查其目标是否落在叶集 ID 值的范围内，如果是则被路由到叶集中最接近的节点，否则使用刚才介绍的路由表选择下一条节点。所以，叶集使得路由变得更加简便快捷，也方便网络的维护，这是 Tapestry 的一个重要改进。

路由定位的理论分析

尽管具体的路由定位方式有所不同，但所有结构化对等网络模型提供的查询路由效率都是高效的，且都能达到 $O(\log N)$ 跳的复杂度，其中 N 是覆盖网络的总节点数。若不考虑拓扑一致性，就覆盖网而言，该效率是接近最优的，我们可以通过论文 [36] 给出的引理更好地理解这点。

引理 2.1 *在一个有 N 个节点的网络中，节点的最大度为 d，且 $d > 2$，那么在最坏情况下路由定位至少需要 $\log_d N - 1$ 跳，而平均路由跳数为 $\log N_d - O(1)$。*

证明 因为节点的最大度为 d，故以某一节点 A 为中心，在其 h 跳范围内的节点数最多是

$$1 + d + d(d-1) + \cdots + d(d-1)^{h-1} = \frac{d(d-1)^h - 2}{d-2}$$

假设 A 到达最远的节点需要 h_m 跳，那么在 h_m 跳范围内的节点数应该为 N，故有

$$N \leq \frac{d(d-1)^{h_m} - 2}{d-2} \leq d^{h_m+1}$$

因此 $h_m \geq \log_d N - 1$，即在最坏情况下，路由定位至少需要 $\log_d N - 1$ 跳。

若 A 为节点集群的中心，并假设每个节点的度都接近 d 的情况下，大多数节点分布在与 A 距离 $(\log_d N - 1)$ 跳左右，因此平均路由跳数为 $\log_d N - O(1)$；只要不是有太多节点的度小于 d，平均路由跳数还是近似符合上述规律 [5] [36]。 ■

大多数模型，如 Chord，Kademlia，Pastry 和 Tapestry 等都以 $O(\log N)$ 的度数获得 $O(\log N)$ 的平均路由效率；CAN 以维数 d 获得 $O(d\sqrt[d]{N})$ 的平均跳数，若 $d = \log N$，其实也就是 $O(\log N)$。在渐近的意义上，它们都接近理论上的路由效率。当然，存在更好的理论模型是可能的，如论文 [36] 给出的 Koorde 模型，它能够在 $O(\log N)$ 的节点度下达到 $O(\log N / \log\log N)$ 的路由效率；论文 [40] 设计的基于不完备 Kautz 有向图 (Incomplete Kautz Digraph) 的常数度对等网络模型 Moore，可达到 $O(\log_d N)$ 跳的最佳平均路由。

2.4.3 动态性与容错性

结构化对等网络的动态节点算法是实现自组织性和容错性的关键，它包括两个部分：节点的加入和退出。几乎所有的结构化对等网络的节点加入算法都是类似的。首先，新节点 N 必须已知至少一个网络中的现存节点 B，以该节点作为入口 (Bootstrap)；N 通过 B 发送以 N 的节点值为目标的消息，该消息最终到达 ID 值与 N 最接近的节点 R；R 和消息路径上的每个节点都将它们的路由表信息以及应由 N 负责的数据索引返回给 N，并更新各自的路由表；N 利用收到的信息初始化路由表，然后再作修正和优化；最后，N 通知其它相关节点更新路由表，以反映它的加入。

节点的离开也类似：将索引数据移交给合适的节点，并通知其它相关的节点更新路由表以反映其离开，这是节点正常离开的情况；若节点突然失效，无法执行正确的离开程序，则系统需要提供必要的检测手段，定期检查周围情况，一旦发现异常则马上修复路由表并通知相关节点。

在 Tapestry 中，为了使路由表保持更新和增加容错性，其路由表还加入了一些反向指针 (Back Pointer)，它们指向那些将自己作为路由表项的节点 (称为反向节点)。在节点 N 的加入过程中，N 的加入消息通过已知节点 B 到达根节点 R 后，R 先计算它

与 N 的匹配前缀位数 p，然后使用反向指针告诉那些与 R 也匹配 p 位前缀的反向节点，如果 N 适合它们，它们就在路由表中添加 N 或用 N 替换原来的项。在系统运行时，每个节点周期性地发送“心跳”信息给反向节点，以表明自己的存在，若反向节点在某个周期里没有收到“心跳”信息，则认为该路由表项失效，应更新路由信息。另外，在论文 [33] 给出的 Tapestry 模型中，路由表中的每项可保存多个节点信息，并选择最优的作为主节点，一旦主节点失效，就由备用节点替代，但在目前 Chimera 的实现中，还是只使用一个节点。除了反向指针，Tapestry 中的对象索引都使用“软状态”(Soft State) 的发布方式，即对象索引都是暂时性的，对象拥有者应定期重新发布索引信息，以更新对象的根节点。

结构化对等网络的节点加入和退出算法要比混合式和无结构对等网络复杂，原因是结构化的系统需要保持严格的覆盖网络拓扑结构，其开销通常是 $O(\log N)$ 或 $O(\log^2 N)$[5]。

2.4.4 拓扑一致性

拓扑一致性，是衡量覆盖网络与物理网络一致程度的指标，又称为局部性。因为由图 2-1 的分层模型可以知道，当覆盖网络映射到物理网络时，结构可能会有很大的差异，例如，对等网络中的一对邻居，在覆盖网络中只相隔一跳，但在物理网络中却可能相隔多跳；又或者两个节点在物理网络中距离很近，但在覆盖网络中却相隔多跳路由。所以，要提高结构化对等网络的路由效率，仅在覆盖网络中达到 2.4.2 节给出的最优路由跳数是不够的，拓扑一致性也是决定通信延迟的重要因素。

与 Chord 和 CAN 不同，Tapestry 和 Pastry 在构建网络时就考虑这个问题。如 2.4.2 节所述，Tapestry(或 Pastry) 在构建路由表时，表中的每一项都选择通信延迟最小的节点，这可以获得一定程度的拓扑一致性。为进一步增强一致性，Pastry 的每个节点还维护一个邻居集 (Neighborhood Set)，集合中的节点在物理网络上与当前节点最近邻，节点根据不断变化的邻居集动态修正路由表。虽然最新的 Tapestry 实现参考了 Pastry，但却没有使用邻居集来增强局部性，而是采用复制缓存的办法。具体做法是，拥有数据对象的节点除了将对象索引发布到其根节点外，还在发布路径的每一条跳上缓存索引，如图 2-7 所示。

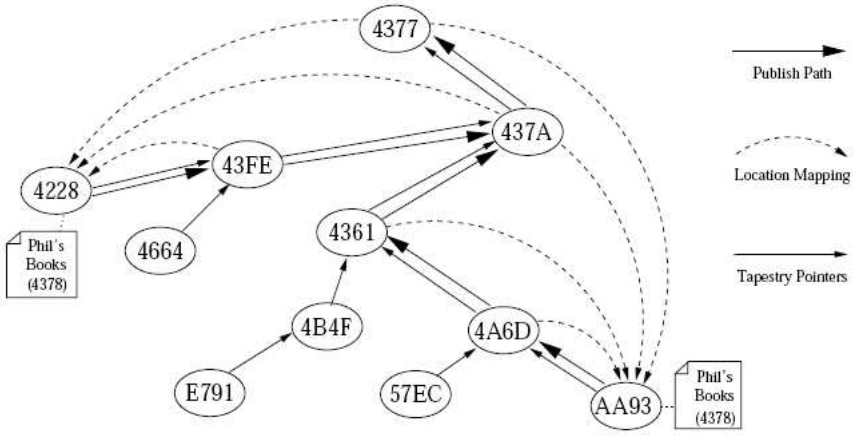


图 2-7 对象 4378 的两个索引副本分别被发布，从源节点到根节点 4377 的每一跳上，索引都被缓存，图片来自文献 [33].

使用这种路径缓存的好处是：当有节点需要查询对象时，通常不必到达根节点就可获得对象的索引，然后迅速定位到在覆盖网上最近的对象副本上。这种策略之所以能提前定位对象，且是离自己最近的对象，原因是覆盖网上的节点越近邻，它们到达同一节点的路径重叠可能性就越大，如图 2-8 所示。

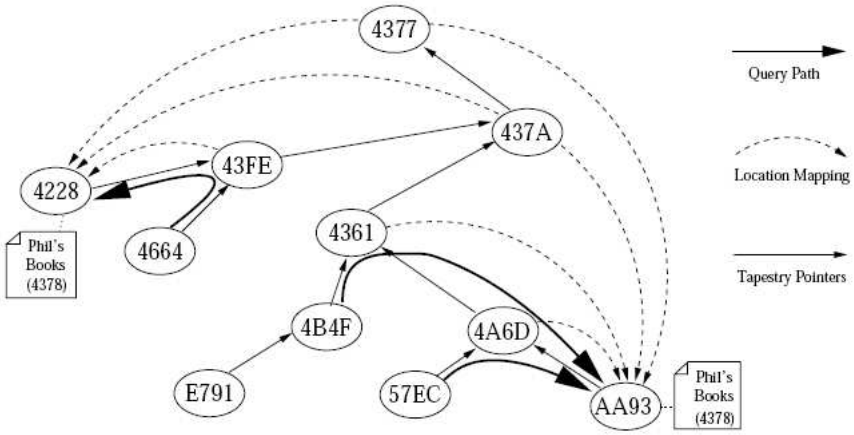


图 2-8 查找对象 4378 索引的两个节点被分别定位到其最近邻的源节点上，图片来自文献 [33].

2.4.5 Tapestry 体系结构

因为本文的原型系统使用 Tapestry 作为底层的对等网络架构，所以有必要简单介绍 Tapestry 的体系结构。图 2-9 展示 Tapestry 系统的功能层次，下面从底往上介绍各层的作用：

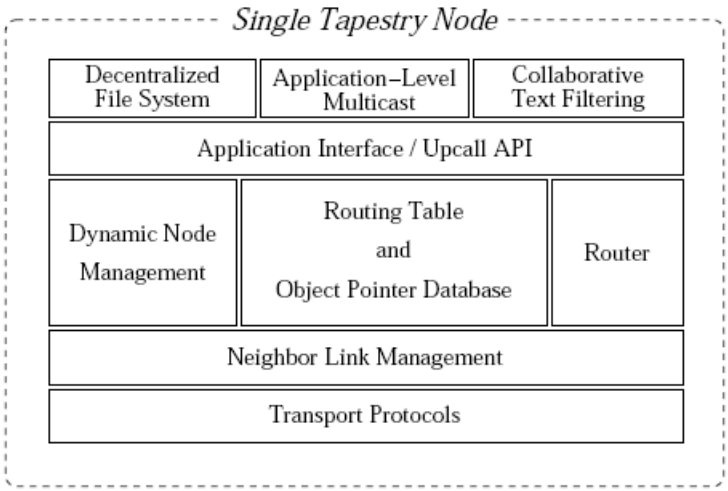


图 2-9 Tapestry 组件结构，图片来自文献 [33].

最底层是传输层，它封装了覆盖网络中节点间的通信信道，相当于覆盖网络和物理网络的中间层 (如图 2-1 所示)，通常使用 TCP 或 UDP 协议，理论上 Tapestry 模型支持两种协议的使用，但 Chimera 实现只使用 UDP 进行通信。该层对应 Chimera 的 Network 模块。

第二层是邻居链接管理 (Neighbor Link Management) 层，它向上层提供安全但不可靠的数据报服务，包括长消息的分片和整合。该层的另一个重要功能是对持续的链接进行管理和更新，如通过软状态 (见 2.4.3 节) 重发布消息检测失效节点，估计延迟和丢失率等，必要时将这些信息提交给上一层处理。另外，为了节省系统资源，该层会周期性地关闭一些连接，在需要时才重新开启。这一层对应 Chimera 实现的 Message Layer 和 Performence Monitoring 模块。

第三层是 Tapestry 的路由层，该层由三个部分组成：动态节点管理器 (Dynamic

Node Management)、路由器 (Router)、路由表和对象指针数据库。动态节点管理器负责节点的加入和离开 (见 2.4.3 节)，路由器负责路由定位 (见 2.4.2 节)。这一层是 Tapestry 的核心实现，Chimera 中的对应模块为 Routing System。

有了以上三层，Tapestry 系统基本成形，剩下的就是要向上层的应用提供开发接口 (API)，这就是第四层的作用。主要的接口都是以底层事件驱动，故又称为唤醒接口 (Upcall API)，用户只要编写相应事件的响应函数，然后以其函数指针作为参数传递给接口函数。当收到某一类型的消息时，系统就会使用函数指针调用响应函数进行处理。在 Chimera 中，有三个不同的唤醒接口：

typedef void (*chimera_update_upcall_t)(Key *key, ChimeraHost *host, int joined);

void chimera_update(ChimeraState *state, chimera_update_upcall_t func);

当有一个 ID 值为**key**的主机**host**加入或离开时节点的叶集 (见 2.4.2 节) 时，响应 update 事件的函数**func**被调用，参数**joined**为 0 时表示节点离开，1 表示加入。其中，**ChimeraState**是记录节点状态信息的结构体，详细介绍见 4.1.1 节。

typedef void (*chimera_forward_upcall_t)(Key **key, Message **msg, ChimeraHost **host);

void chimera_forward(ChimeraState *state, chimera_forward_upcall_t func);

forward 唤醒的作用是通知上层应用：Chimera 将要把目标为**key**的消息**msg**转发到下一跳**host**，应用程序可根据需要作适当的处理，它允许应用程序修改路由层的路由决定，因此可以修改消息和目标地址。

typedef void (*chimera_deliver_upcall_t)(Key *key, Message *msg);

void chimera_deliver(ChimeraState *state, chimera_deliver_upcall_t func);

当节点是目标对象**key**的根节点时，deliver 唤醒通知上层应用程序接收消息**msg**并作相应处理。

事实上，通过修改函数指针的声明，可以适当修改响应函数的参数，在本文的给出的原型系统中，为了方便多线程的实现，增加了一个额外的参数 (见 4.1.2 节)。除了这三个重要的唤醒接口外，Chimera 还提供了一些更底层的调用，允许应用程序直接访问路由信息和实施路由决策，这样既可以使接口有更大的灵活性，又方便对 Tapestry

做一些实验性的测试。本文将在 4.1.1 节对 Chimera 库的使用作进一步介绍。

2.5 本章小结

本章介绍了三种主要的对等网络体系：混合式对等网络、无结构对等网络和结构化对等网络。通过考察其各自的特点，我们选定结构化体系作为搜索系统的基础，因为快速的路由定位是实现高效搜索的必要条件，而这却是混合式和无结构体系所不具备的；在众多的结构化模型中，我们重点介绍了 Tapestry 模型，Tapestry 对拓扑一致性有较多的考虑，并具有高效且规范的实现 Chimera，故 Tapestry 是本文原型系统所选用的对等网络，事实上，原型系统可以容易地移植到其它符合论文 [21] 接口规范的结构化模型中。在下一章，我们将会在结构化对等网络的基础上讨论高效的多关键词搜索技术。

第 3 章 对等网络上的多关键词搜索技术

上一章介绍的结构化对等网络可以支持高效的键值查询，但键值查询本身是最平凡的简单查询，难以满足用户的需求。正如我们在绪论中指出，在对等网络上支持复杂查询是当前研究的一大挑战，而在复杂查询当中，多关键词查询又是当前研究的重点，因此本章将在结构化对等网络提供的键值查询基础上，尝试提出一种支持多关键词查询的搜索技术。在提出该技术之前，先给出相关的概念和技术。

3.1 向量空间模型

向量空间模型 (Vector Space Model，简称 VSM)[41][42] 是信息检索领域的经典模型之一，至今仍有广泛应用。VSM 的成功之处在于将信息检索转化为数值线性代数的有关计算问题，而数值线性代数本身已有较为成熟的理论和技术。具体地，VSM 将文档和查询都表示成 n 维向量空间中的一个向量，其中 n 是被索引的关键词总数 (或词典的词汇量)。向量中的每一个分量表示对应关键词在文档或查询中的权重 (3.6 节讨论权重的确定)。对于文档集 D 中的任意一个文档 $d_j = (d_{j1}, d_{j2}, \ldots, d_{jn})^T$，和用户提交的一个查询 $q = (q_1, q_2, \ldots, q_n)^T$，它们之间的相关性通常使用向量夹角的余弦值来度量，定义如下：

$$sim(d_j, q) = \frac{d_j^T q}{||d_j||_2||q||_2} = \frac{\sum_{i=1}^{n} d_{ji} q_i}{\sqrt{\sum_{i=1}^{n} d_{ji}^2}\sqrt{\sum_{i=1}^{n} q_i^2}}. \tag{3.1}$$

同理，文档与文档间的相似性也可以使用向量夹角余弦值来度量。若将文档向量作为矩阵中的行向量，则整个文档集合可构成一个矩阵

$$\begin{pmatrix} d_{11} & d_{12} & \cdots & d_{1n} \\ d_{21} & d_{22} & \cdots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{|D|1} & d_{|D|2} & \cdots & d_{|D|n} \end{pmatrix} \tag{3.2}$$

其中 $|D|$ 是文档集合的规模。于是，计算集合中的文档与给定查询的相关性就转化为计算文档矩阵与 q 的乘积。但文档矩阵通常是巨大且稀疏的，因为每个文档和查询仅使

用部分的词汇，故其储存和计算都需要优化。

事实上，高效的算法往往建立在良好的数据结构上，因此，快速的相关性计算很大程度依赖于文档矩阵的储存结构，如下一节将要介绍的倒排索引结构。该结构在传统搜索技术中可以良好地工作，为移植到对等网络的环境中，必须实现分布式的倒排索引结构，以及其他的增强特性，这正是本章要重点解决的问题；与此对应，相关性的计算也应该有分布式的计算方法，但这不是本文讨论的内容，可参考有关的数值线性代数知识；另外，我们还将初步讨论了对等环境下的负载均衡 (见 3.5 节) 和相关性排序的问题 (见 3.6 节)。

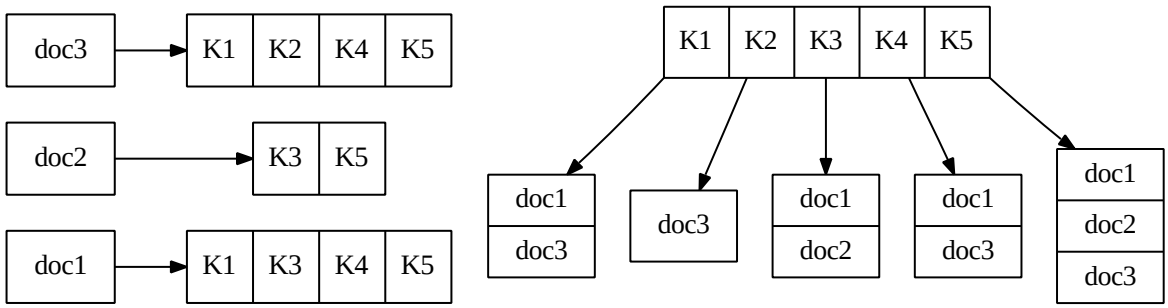
3.2 文档索引技术

3.2.1 倒排索引

为了实现快速的关键词匹配，大部分搜索引擎都使用倒排索引的数据结构，尽管具体的实现形式可能不一样，但基本原理是一致的。倒排索引维护两个索引列表：文档表 (Document Table) 和关键词表 (Term Table)，如图 3-1 所示。实际上，倒排索引记录的就是对文档矩阵 (3.2 式) 的行和列，文档表的记录相当于矩阵的行，关键词表的记录相当于矩阵的列，而且它们都考虑到稀疏性 — 仅记录相关的关键词和文档。由于文档表较容易构造，而关键词表则需要全局的信息，所以人们往往将关键词表等同于倒排索引，事实上，在 3.2.2 节我们将会看到，关键词表确实是较为重要的。

采用倒排索引，可以避免使用整个文档集合计算相关性，从而提高搜索效率。例如，对于一个含有关键词 k_1，k_2 和 k_3 的查询 Q，先从关键词表中分别取出关键词对应的文档集合 D_{k_1}、D_{k_2} 和 D_{k_3}，然后取其交集 $D_Q = D_{k_1} \cap D_{k_2} \cap D_{k_3}$，此时，搜索的范围大大变小，只要从文档表中取出集合 D_Q 中对应的文档向量计算相关性即可。

由上面的介绍可以看到，倒排索引既可解决稀疏矩阵 (3.2 式) 的储存问题，又可加快搜索的速度，是一种简单有效的数据结构。但在面对海量数据时，倒排索引规模将会非常巨大，因此有必要对其进行划分，以分布到不同的储存单元上。目前，传统的搜索引擎已有成熟的划分技术，并配有相应的并行算法，可惜这些技术并不能直接地



(a) 文档表 (b) 关键词表

图 3-1 倒排索引结构.

移植到对等网络环境中，下一小节将讨论这个问题。

3.2.2 索引划分

要使对等搜索系统具有可扩展性，以应对海量的数据，索引划分是必要的。由倒排索引的结构可以知道，文档表的每条记录与对应文档是密切相关的，其划分必然基于文档。在对等环境中，文档表已有一个自然的划分，那就是每个对等端只需维护其共享信息的文档表，因此划分问题归结为关键词表的划分。

目前，对于关键词表主要有两种划分方法：基于文档的划分 (Document-Based Partitioning) 和基于关键词的划分 (Keyword-Based Partitioning)，有的文献将前者称为水平划分 (Horizontal Partitioning)，将后者称为垂直划分 (Vertical Partitioning)[43]，如图 3-2 所示。

基于文档的索引划分方法将关键词表按文档划分成多个子集，每个节点负责一个或多个文档子集的索引。这种划分方法容易实现负载均衡，因为文档可以平均地分配到各个节点；另外，该划分也有利于文档更新，只需要更新负责文档对应的节点即可 (还未考虑备份节点)。但是在这种划分下，若文档集没有分类而随机地分布在整个系统中，查询就必须遍历所有的节点才能确保查全，故此时的系统并不具备良好的可扩展性。在传统的搜索引擎中，由于具有中心管理的特点，文档和节点都容易实现分类，查询可以被转发到具有相关文档的节点上，从而避免盲目的遍历。因此，传统搜索引擎可以采用基于文档的划分，如 Google[44]。相反，在对等环境下，由于节点具有高度

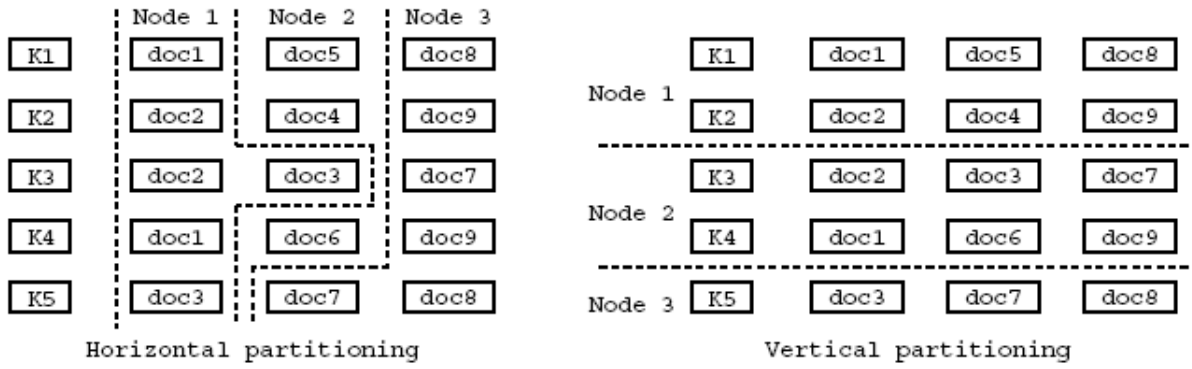


图 3-2 索引划分，doc 表示文档，K 表示关键词，图片来自文献 [43].

的自治性，不容易实现文档或节点的分类，每个节点通常只维护其共享的文档集，该集合反映用户的兴趣，但往往不属于某一特定类型。因此，要实现可扩展性，基于文档的索引划分方法并不是对等搜索系统的最佳选择。

基于关键词的索引划分方法将关键词表按关键词划分成多个部分，每个节点负责一个或多个关键词的索引。与基于文档的划分不同，该种划分方法能够解决遍历的问题：查询请求先被发送到含有所需关键词索引的节点上，再将这些节点返回的文档集合取交集，就能定位到与查询相关的文档集合上。由于用户的查询通常只有少量的关键词 (见 3.3.3 节)，这个处理过程只需要少数的节点参与，相比之前的划分将更为高效，更具有扩展性。因为可扩展性是对等系统最受人青睐的特点，同时，高效的搜索又是本文所关注的重点，所以，我们设计的对等搜索系统将使用基于关键词的索引划分方法。

3.2.3 分布式倒排索引

利用结构化对等网络的分布式散列表 (2.4.2 节) 和基于关键词的划分方法，可构造分布式的倒排索引，具体方法如图 3-3 所示：先用散列函数计算出关键词 k 的键值 $hash(k)$，然后将文档的索引信息发布到关键词的根节点 (ID 值与键值最接近的节点) 上。因此，每个关键词对应唯一的根节点，根节点负责维护该关键词的文档集合。

在实际当中，每个文档去掉应删去词后可能仍有较多的关键词，因而要发布的索引信息 P 也较多，故此应考虑优化策略。首先，发布信息 P 应该尽量精简，在我们的

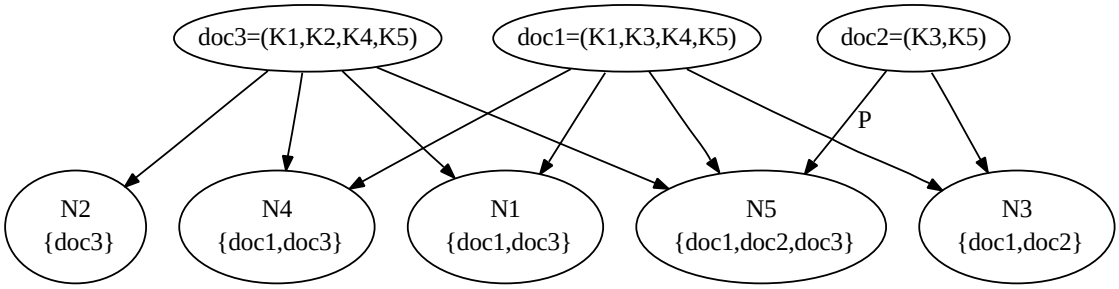


图 3-3 文档索引发布，P 为发布信息.

设计中，P 的消息类型为 PLSH(见 4.2.1 节)，除了头部信息，它包含的必要信息为关键词、文档 ID(或文档的检验和)、资源地址和该关键词在文档中的权重 (局部权重)，如图 3-4 所示； 除了 P 的大小外，发布信息的数量也应尽量地少，这是可能的，因为

Keyword	Document ID	Document URL	Keyword Rating

图 3-4 发布信息的内容.

通常使用较少的关键词就可概括文档的语义信息，例如，当前语义网 (Semantic Web) 技术 [45] 是一个可能的发展方向，该技术利用人工添加的元信息 (Metadata) 或标签 (Tags) 来帮助计算机更好地理解文档的语义。但这不是本文讨论的内容，故我们将把关键词选取的策略留给更上层实现，显然，好的关键词选取策略不仅减少发布信息的数量，而且可以提高文档被检索到的概率。

虽然基于关键词的划分策略使得系统在整体上具有可扩展性，但在实际环境中，原始的方法并未能实现高效的多关键词搜索，譬如当查询含有多个关键词时，处理文档集合求交运算的开销是不能忽略的，尤其是文档集合较大时；此外，基于关键词划分会造成负载不均衡的问题，一方面是因为关键词的使用频率不同；另一方面是由于关键词的受关注程度不同，所以各节点承受的储存或计算负载将有巨大差异。在本章的余下部分，我们将详细讨论以上的问题。

3.3 分布式搜索策略

3.3.1 原始策略

有了分布式倒排索引，就可以进行多关键词搜索。对于查询 $Q=(k_1,k_2,\ldots,k_n)$，当 $n=1$ 时，处理十分简单，查询被路由到关键词的根节点，根节点再返回相关的文档集合，这个过程实质上只是结构化对等网络的键值查询；真正的多关键词查询是 $n>1$ 的情况，在这种情况中，假设关键词 k_i 的根节点拥有的文档集合为 D_{k_i}，而满足查询 Q 的文档集合应为所有相关文档集合的交集

$$D_Q=\bigcap_i^n D_{k_i}. \tag{3.3}$$

图 3-5 展示了 $n=3$ 的一个搜索过程，假设使用图 3-3 的分布式索引，完成查询 $Q=(k_1,k_2,k_5)$ 需要计算两次的求交运算。

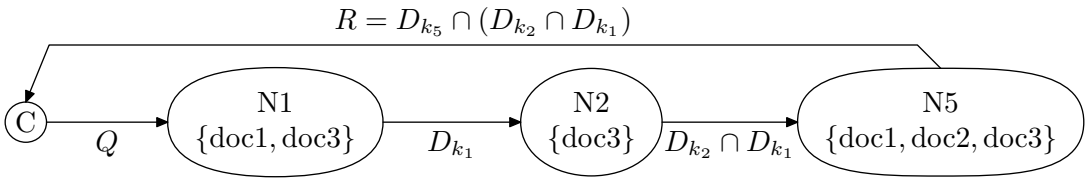


图 3-5 客户端 C 发出查询 Q，经过 N2，N5 的两次求交运算后，返回结果 R.

但是在对等环境中，实现求交运算需要较大延迟：因为倒排索引是分布式的，负责不同关键词的节点可能相隔较远，如果传输的数据量较大，则会消耗较多的带宽和产生较长的通信延迟，这对系统总体性能与用户体验都是不利的。所以，原始的搜索策略并不可行，下面讨论优化的策略。

3.3.2 减少传输数据

调整求交次序

要减少取交集时的传输数据量，有一个简单的策略，那就是调整文档集合的求交

次序：从文档集合最小的节点开始，依次将其发送给下一个节点进行求交处理，而每次求交运算得到的新列表都比原来的要小。因此，从最小的文档集合开始求交的策略可以减少整个查询处理过程的传输数据量。例如，在图 3-5 展示的例子中，如果客户端先将查询请求发送到 N2(文档集合最小)，然后 N2 再将文档集合发送到其余两个节点求交集，则每次传输的文档集合中都只有一个文档。但这种方法需要事先知道节点的文档集合大小，为此可以在查询之前先向相关节点询问列表大小 (原型系统的 CK 消息 4.2.1)，若某个节点返回文档集合大小为 0，则满足查询的结果显然为空，否则采取上述的优化策略进行搜索。当然，这个策略的可行性依赖于底层对等网络的路由效率，结构化对等网络自然是当前最好的选择。

采用 Bloom Filter 压缩集合

上述策略要发挥作用，必须要有一个节点的文档集合特别小，但是，在实际当中，要索引的文档是海量的，即使关键词的使用频率较低，其对应的文档集合也可能是较大的，所以，上面的策略不足以解决问题，压缩文档集合是必要的。研究者对此提出了不少的解决方案，最典型的做法是使用 Bloom Filter[46] 进行压缩，下面先对 Bloom Filter 作简单介绍。

Bloom Filter 是一种简单的随机数据结构，可用来精简集合数据的表示，并实现高效的元素检测。例如，对于集合 $S = \{x_1, x_2, \ldots, x_n\}$，用 m 位的 Bloom Filter 来表示：首先，Bloom Filter 被初始化为 m 个元素都是 0 的位数组；然后使用 k 个值域为 $\{1, \ldots, m\}$ 的散列函数 $h_1, \ldots, h_k$，对任意 $x \in S$，分别用 k 个散列函数作用它，并将数组的将第 $h_i(x)$ 位置成 1(其中 $1 < i < k$)，之前已被置成 1 的位保持不变；这样，对 S 的所有元素都做以上操作，最终就得到集合 S 的 Bloom Filter 表示，记为 $F(S)$。当要使用 $F(S)$ 确定元素 y 是否为集合 S 中的元素时，继续使用刚才的 k 个散列函数，检查其第 $h_i(y)$ 位是否为 1，若检查的 k 个位都是 1，则 Bloom Filter 判断 y 为 S 中的元素，否则 y 一定不是 S 的元素。图 3-6 是一个 Bloom Filter 的例子，前两步是构造 Bloom Filter 的过程，最后是 Bloom Filter 的使用，可以看到，y_1 在 Bloom Filter 中对应的位不全为 1，所以它一定不是集合中的元素，而 y_2 对应的位都为 1，故 Bloom Filter 判断 y_2 为集合中的元素，但这个判断有时会出错，称为误判 (False Positive)。

从刚才的例子可以知道，Bloom Filter 压缩信息的代价是出现误判，但这个缺陷不

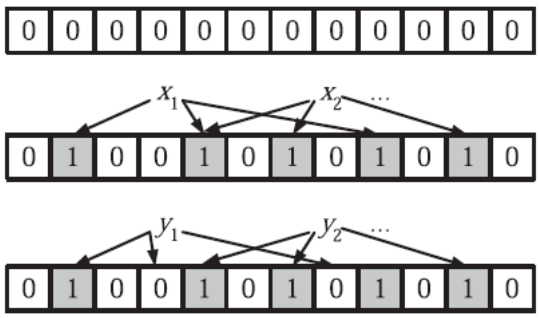


图 3-6 Bloom Filter 示例，图片来自文献 [47].

足以令人放弃 Bloom Filter 的使用，下面简单给出误判率分析的结论，详细分析可参考文献 [47]。Bloom Filter 误判率的近似公式为

$$f \approx \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k . \tag{3.4}$$

其中 k，m 和 n 的意义同上面的介绍一样。在分析中，通常使用 (3.4) 式右端的近似。利用该近似容易求得，当 $k = (m/n)\ln 2$ 时，误判率取最小值

$$f_{min} = (0.6185)^{m/n} . \tag{3.5}$$

由上式可知，当集合元素个数 n 一定时，Bloom Filter 的误判率可由其长度 m 来控制，因此，在实际应用中，可先选择一个能够接受的误判率，然后用 (3.5) 式计算最小的 m 值；如果再使用无损的压缩工具对 Bloom Filter 进行压缩，则会得到更加可观的压缩比例。考虑元素为 ID 值的文档集合，每个 ID 长 160 位 (bit)，共有 $1,000$ 个，则整个文档集合需要 $160,000$ 位，而如果使用误判率为 0.001 的 Bloom Filter，则只需 $14,378$ 位，长度不足原来的 1/10，使用压缩工具还可以进一步减少这个比例。

既然 Bloom Filter 能以更紧凑的形式表示集合，使用它进行集合运算是很自然的想法。考虑两个集合 S_1 和 S_2，分别使用同样的散列函数族计算它们的 Bloom Filter，记为 $F(S_1)$ 和 $F(S_2)$，再对它们按位作与 (AND) 运算，即 $F(S_1)\&F(S_2)$，那么得到的位数组是否等于 $F(S_1 \cap S_2)$ 呢？论文 [48] 详细研究了这个问题，结论是不一定的，而且使用 $F(S_1)\&F(S_2)$ 测试元素关系时，误判率可能要比 $F(S_1 \cap S_2)$ 稍高，因此，虽然使用该方法可提高求交运算速度，但代价是进一步增加误判率。论文 [43] 提到了使用该种方法的近似搜索策略，但没有考虑误判率上升的影响。

论文 [43] 着重讨论用 Bloom Filter 进行精确搜索，虽然其重点讲述取一次交集的情形，但容易推广到多次的情况。继续以三个关键词的查询为例来展示他们的方法，如图 3-7 所示，用户 C 将查询 Q 提交给节点 N1，N1 将其文档集合压缩成 $F(D_1)$ 后发送给 N2，N2 将收到的 $F(D_1)$ 和自己的 $F(D_2)$ 做与运算，得到一个 $F(D_1 \cap D_2)$ 的近似 Filter，然后再发送给 N3，N3 将经过 $F(D_1)\&F(D_2)$ 过滤后的集合返回给 N2，N2 剔除误判的项后得到 $D_2 \cap D_3$ 的列表，再使用 $F(D_1)$ 过滤后返回给 N1，N1 剔除误判项后返回最终结果 $R = D_1 \cap D_2 \cap D_3$。

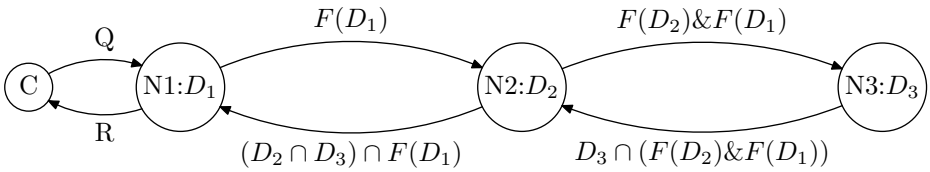


图 3-7 使用 Bloom Filter 的多关键词搜索，其中 $F(D)$ 为文档集合 D 的 Bloom Filter，& 为与运算符，$D_3 \cap (F(D_2)\&F(D_1))$ 表示 D_3 经 $F(D_2)\&F(D_1)$ 过滤后的集合.

从上面的过程可以看到，为了剔除误判项，每次经 Bloom Filter 过滤的集合都要返回给前一个节点，这会带来一定的延迟，只要查询中的关键词不太多，这问题还不太大，而“查询关键词不太多”在实际当中是基本成立的，本文将在 3.3.3 节讨论。论文 [43] 对这种方法作了进一步的优化：缓存 (Cache) 一些节点的 Bloom Filter，特别是那些与“热点”关键词对应的列表。有了对方的 Filter，就可以省掉一些等待的时间，例如在刚才的例子中，如果 N2 已经有 N1 的 Filter，它就可以立即发送 $F(D_1)\&F(D_2)$ 给 N3，N1 就可以尽快收到 $(D_2 \cap D_3) \cap F(D_1)$ 的集合，于是，C 就会有更快的响应。而且通常的情况是，少数关键词占据了大部分查询 (见 3.3.3 节)，所以即使缓存少量的 Filter 也能有很高的命中率 (Hit Rate)。

虽然上述缓存策略可以减少 Filter 传输的开销，但并没有减少太多的节点计算开销，在上述例子中，即使节点 N3 都有 N1 和 N2 的 Filter 缓存，但是它并没有缓存最终结果，当下一条同样的查询到达时，它必须重新计算。因此，简单的想法是让 N3 缓存最终结果，但这对 N3 似乎不太公平，因为相对 N1 和 N2，它承担了更多的负载。在下一节，我们将使用分布式散列表来缓存查询结果，从而减少重复的求交计算。

3.3.3 基于查询的搜索技术

用户查询特征

本节使用搜狗实验室 (Sogou Lab)[49] 提供的用户查询日志作为研究数据，完整数据包含 Sogou 搜索引擎在一个月内的查询记录及用户点击行为等信息。但我们只选取了一天的查询数据 (即 Sogou 提供的精简版数据) 进行研究，这是因为缓存通常是短暂性的，并不需要研究较长时间的数据；另外，我们的实验环境 (Intel(R) Celeron(R) CPU 2.40GHz，512MB MEM) 也不适合处理较大的数据。

我们从原始数据中提取用户查询，并用简易中文分词系统 (SCWS，1.1.1 版本)[50] 对其分词，除去标点和应删去词，将原始查询转换成多关键词查询，由于分词系统仍不够完善，所以我们还剔除了一些分词系统处理得不是太好的查询项，如超链接地址等，但这只是极少数的项目，并不影响分析。最终得到 $1,012,429$ 条多关键词查询，下面的分析基于这些数据。

首先验证上节提出的一个假设：通常每条查询含有的关键词数量不太多。图 3-8 是从日志数据中得到的统计结果，可以看到，含有不超过 4 个关键词的查询占据 95% 以上。论文 [43] 也作了类似的统计，他们统计了 $99,405$ 条英文查询，从他们的统计中也可以发现类似的特征, 但他们并没有利用这个特征。为了方便，不妨将含有 5 个以下关键词的查询称为基础查询，它们是用户较经常使用的查询。在研究优化策略时，我们将先考虑基础查询，若系统对基础查询的工作效率得到提高，则意味着系统在大多数情况下是工作良好的；而对于含有更多关键词的查询，可分解为多个基础查询，然后再求交结果，因此其余的查询都依赖于基础查询。

事实上，除了只包含一个关键词的查询，任何多关键词查询都可以分解为更小的查询。或者换一个说法，将查询看成关键词集 (假设查询中的关键词两两不相同)，若其元素个数为 n，则可将其分解为 2^n-1 个非空子集，其中元素个数为 i 的子集有 $\binom{n}{i}$ 个。虽然子集个数会随 n 呈指数上升，但由上面的分析可知，对大部分查询而言，n 都是较小的，于是它分解出来的子集个数也较小，后面将看到这对于我们的索引机制和查询策略可行性都有重要意义。

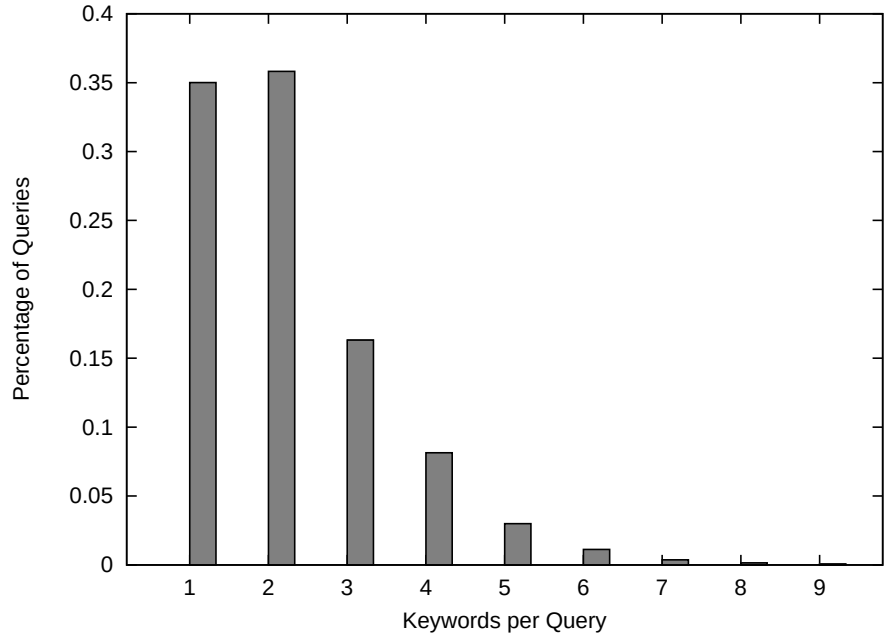


图 3-8 每条查询的关键词数量统计，其中含有 1 个到 4 个关键词的查询分别占据 35.0%，35.8%，16.3% 和 8.1%.

引入关键词集的概念，是为了将其看成一个整体，并在系统中加入一些负责关键词集索引信息的节点。在前面介绍的分布式倒排索引结构 (见 3.2.3 节) 中，每个节点负责的只是一个关键词的索引信息 (假设节点足够多，也不考虑散列函数的冲突率)，而现在我们更进一步地让节点负责某一关键词集的索引信息，称该节点为关键词集的根节点，其定义与单个关键词的类似：对关键词集 $K = \{k_1, k_2, \ldots, k_n\}$，可取其键值为各关键词的键值和，

$$hash(K) = \left(\sum_{i=1}^{n} hash(k_i) \right) \mod b^{length(ID)}. \tag{3.6}$$

取模是为了使键值落在标识符空间中，这里假设标识符使用 b 进制，长度为 $length(ID)$，ID 值与该键值最接近的就是其根节点。此外，我们根据关键词集的大小对系统节点进行划分，含有 n 个元素的词集归属第 n 层，故原本意义的根节点被划分到第 1 层，新的分布式的倒排索引被划分为多个层次。这种新的索引结构将有望加快基础查询的搜索效率，因为使用高层次的索引信息可以减少求交运算次数。在讨论具体的搜索技术之前，先分析层次化索引的构造方法。

原始的倒排索引结构已经具有一层的根节点，理论上，可以使用组合的办法从第

一层构造出其他的层次，但实际上是不可能的，因为第一层的节点已经是数以万计，任意地组合将会导致组合爆炸 (Combinatorial Explosion)；而事实上也没有必要构造所有的组合，只有当关键词集中的词语是紧密联系时才有意义。因此，简单的组合方法显然是不可行的。要让系统判断关键词的组合是否有意义，最简单直接的方法是从用户的查询中学习。因为既然用户将某一关键词集作为查询，这些关键词之间必然存在一定的联系 (至少对用户来说是有的)，所以我们认为，某一关键词集在越多的用户查询中出现，其关联性越大，在它的根节点中建立索引就越有必要。为了验证这个构造方法的可行性，我们继续对用户查询进行研究。

对于数据中的每一条查询 Q，假设含有 n 个关键词，我们将其分解为 $2^n - 1$ 个非空子集，并对所有子集进行计数，得到表 3-1 的统计数据，其中，层次是指关键词集的所属层级，后两项是该层的平均词频和词频标准差。

从表中可以看到，在 100 多万条的查询中，用户实际使用的词汇量为 $61,544$；如果使用组合的办法构造出第二层，则会有 $\binom{61,544}{2} = 1,893,801,196$ 个二元关键词集，这已经是一个相当大的数字，往上的层次就更不用说了，而如果由用户查询分解出的关键词集来构造，则第二层只有 $380,118$ 个词集，个数最多的第三层有 $394,565$ 个，更高层次的词集数逐渐递减，原因是查询的数量基本上随关键词的增多而减少 (见图 3-8)。显然，由查询驱动的构造所产生的词集数要比全组合少得多，因而是较为可行的。

从各层次的平均词频可以看到，层次越低，访问频率越高，原因很简单，因为每个关键词集都由更低层次的词集组成，所以低层的计数隐含了上层的计数，当然还跟查询关键词个数的特征有关。该数据反映的是各层关键词集索引可能的使用频率，可从一定程度上反映各层节点将要应对的平均负载。如果只使用单层的倒排索引结构，如 3.3.2 节的方法，那么每个节点平均会被访问 35.258 次，这个数字比其他层次高很多，另一方面，这也意味着节点需要做多次的求交运算，而求交通常消耗较大的系统资源。因此，我们希望通过分层，让高层的节点分担低层的负载，使得每个查询尽量在其所属的层次完成，避免重复的求交运算。

最后，关键词集的词频标准差反应的是每一层的负载均衡状况：层次越低越不均衡，也就是说，虽然低层的关键词集平均访问频率很高，但其实热点只集中在少数词集上，也就是说，少数关键词集覆盖大量查询。这个现象提醒我们，要考虑较低层次

的负载均衡问题 (见 3.5 节)，以及可适当删除较为罕用的关键词集索引，将释放的系统资源转移到负载较大的部分。

表 3-1 查询数据中的关键词集统计.

层次	关键词集数	平均词频	词频标准差
1	61,544	35.258	595.771
2	380,118	5.193	131.379
3	394,565	3.299	13.896
4	263,768	2.771	6.691
5	139,125	2.517	3.937
6	57,286	2.410	3.680
7	17,022	2.380	3.724
8	3,189	2.393	3.977
9	279	2.423	4.297

由查询驱动的搜索技术

上一小节通过分析了用户查询的特征，引入了由查询驱动构造的层次化索引结构，在新的索引结构中，多关键词搜索的效率将会得到提高。例如，查询关键词集 $\{k_1, k_2, \ldots, k_n\}$，先查询其对应的根节点 $N_{k_1 \wedge k_2 \wedge \cdots \wedge k_n}$，如果节点已经有该词集的索引信息，那么可马上返回结果，无需任何求交操作，这是最理想的情况，不妨称为查询命中；否则节点 $N_{k_1 \wedge k_2 \wedge \cdots \wedge k_n}$ 将查询分解为低一层的子查询，并分别转发给相应的根节点，假如命中的子查询并集仍不等于原查询，则继续在更低的层次搜索其补集，如此类推，直到足以构成原查询或到达第 1 层为止，然后将命中的子查询结果求交后返回最终结果。

图 3-9 展示了 $n = 4$ 的一个例子，搜索分两个阶段，首先如图 (a) 所示，从分布式索引的高层到低层试探可用索引信息，假设查询 Q 在第 4 和第 3 层都没有命中，到达第 2 层时，只有子查询 $\{k_1, k_3\}$ 命中，则将补集分解为 $\{k_2\}$、$\{k_4\}$，然后继续在第 1 层搜索。在确定了命中的子查询后，上层节点将下层返回的结果取交集后返回给用户，如图 (b) 所示，在这个过程中，没用命中查询的上层节点可作为求交运算节点，并保留

求交结果，成为新的索引节点。系统就是这样不断地从用户查询中学习，增添索引节点，使得下一个查询的命中概率增大，从而提高搜索效率，不妨将这种技术称为由查询驱动的搜索技术。

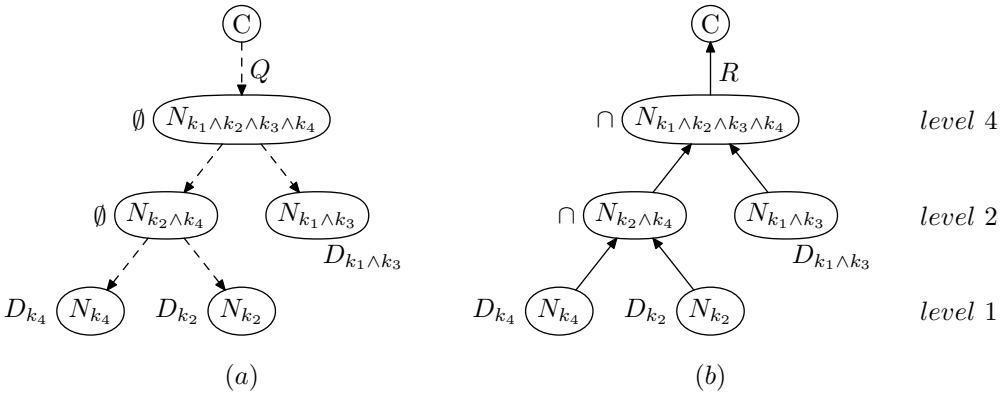


图 3-9 (a) 搜索可用节点；(b) 文档集合求交.

事实上，当 n 较小时，我们可以使用一次性试探所有子集根节点，因为索引是分布式的。这样就可知道 Q 的所有可用查询子集，不妨记为子集族 $\mathcal{F}$，若第一层的关键词总是可用，则有

$$Q = \bigcup_{S\in\mathcal{F}} S. \tag{3.7}$$

理论上，可以找到一个最小的子集族 $\mathcal{C} \subseteq \mathcal{F}$，使得

$$Q = \bigcup_{S\in\mathcal{C}} S \tag{3.8}$$

我们称 $\mathcal{C}$ 为 Q 的最小覆盖。显然，使用最小覆盖可以确保求交次数是最少的，但实际上这个问题却是具有 NP 难度 [51]，对于较小的 n，并不值得为这小小的优化而去解一个 NP 的问题。因此，我们在实际中还是使用前面给出的试探方式，该算法总是先选择高层的索引，所以是一种贪心的近似算法。

3.4 性能分析与优化

下面分析基于层次化分布式索引的搜索效率，为分析系统在每个阶段的运行状况，我们对系统运行作了简单的模拟，具体方法为：使用 Python 中的字典

(Dictionary) 模拟层次化的索引结构，因为字典是可变的 (mutable) 对象，可动态增删或修改元素，并可以嵌套使用；另外，字典具有关联数组 (Associate Array) 的特性，可方便地使用名字 (字符串) 作为索引地址。在我们的模拟中，字典 node 为整个索引，其中的层次可用 node[$level$] 引用，node[$level$] 也是一个字典，该字典记录 $level$ 层的索引节点；node[$level$][$name$] 记录节点 $name$ 的信息 — 包括节点对应的关键词集查询频率和最近一次的访问时间；系统使用 Sogou 的用户查询作为输入数据，每次处理一条查询，并以此作为一个时间单位，每隔 $10,000$ 个时间单位收集一次系统的运行状况，其中包括，

- 当前系统一共使用的求交运算次数，并计算出平均每条查询需要的求交次数；
- 当前整个分布式索引的规模，即生成的索引节点数量.

下面将分别利用以上两个指标考量系统的时间和空间复杂度。

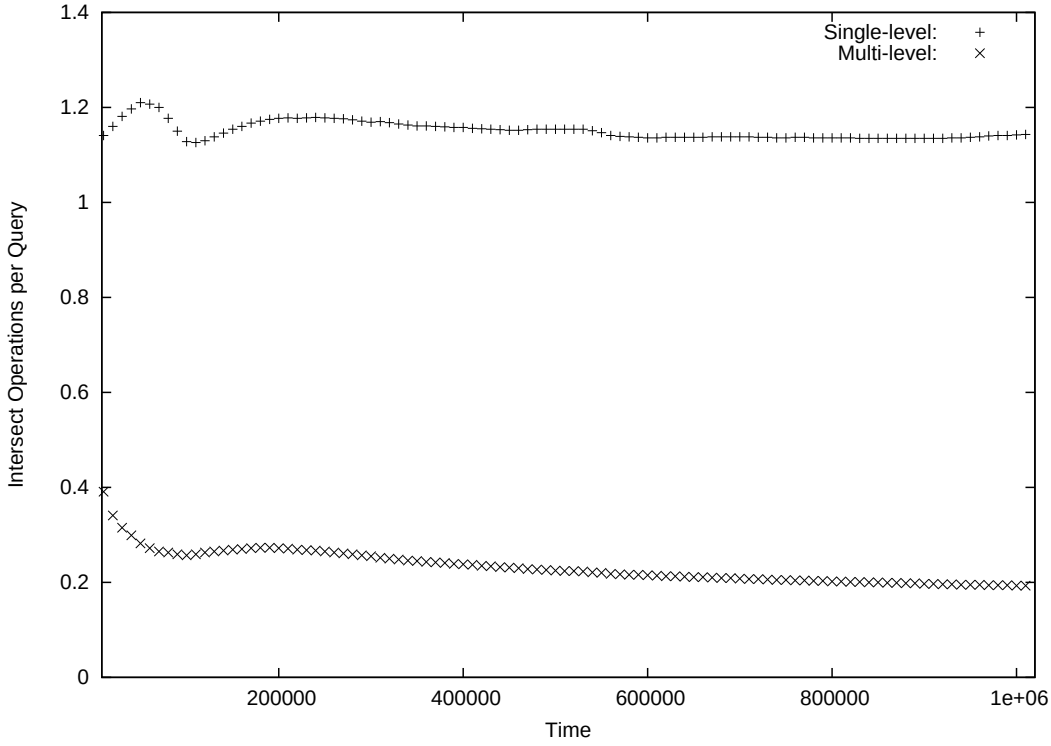


图 3-10 单层和多层索引结构的平均求交次数比较.

首先来看使用层次化的分布式索引对减少求交运算的作用，利用模拟记录的数据，可计算出系统各个阶段平均每条查询所需要的求交运算次数，我们比较了系统在

单层索引与多层索引下的运行情况。如图 3-10 所示，在多层索引结构中，因为上层索引节点的不断增多，完成搜索所需的求交次数不断下降，最终平均求交次数下降到 0.193 次；而使用单层索引结构时，查询的平均求交次数始终多于 1.12 次。因此，从减少求交运算的角度看，搜索效率得到大大提高。

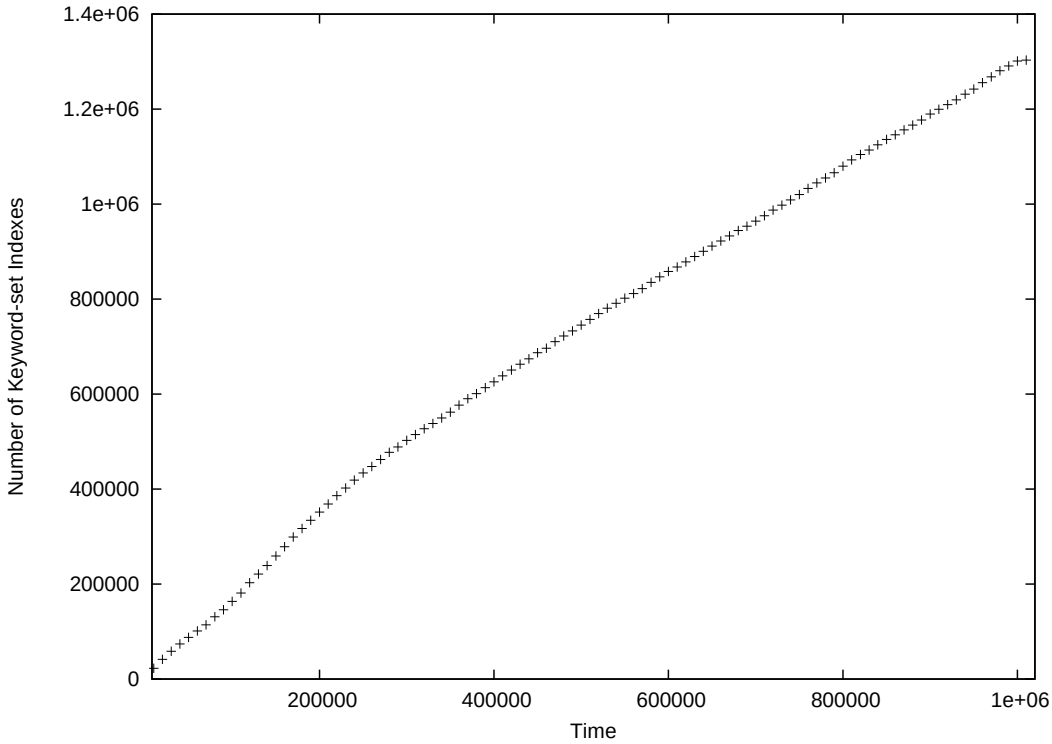


图 3-11 在无删除机制下，系统无限地增加的关键词集索引数量.

虽然，时间复杂度降低了，但仍未考虑其空间复杂度。如图 3-11 所示，随着输入查询的增多，整个系统的索引数量线性地递增，最终生成超过 130 万的关键词集索引，这是因为在刚才的模拟当中，我们为每一条的查询都产生其所有全组合的索引，所以这个数字也可以从表 3-1 的第二列相加得出。仅是一天的查询数据就产生了如此巨大的索引数据，我们需要考虑控制索引规模的有效措施。事实上，我们在分析表 3-1 中的词频标准差时已经指出，是少数关键词集覆盖了大部分的查询，通过检查每个索引节点的使用频率，如图 3-12 所示，除第一层外 (因为第一层是由用户发布信息时产生的)，由查询全组合产生的大部分关键词集索引使用频率都是较低的，其中有超过 55 万个是从未被使用过，维护这些索引显然是不必要的。因此，系统需要使用某种机制来

提高资源使用率，可以从两个方面解决该问题：

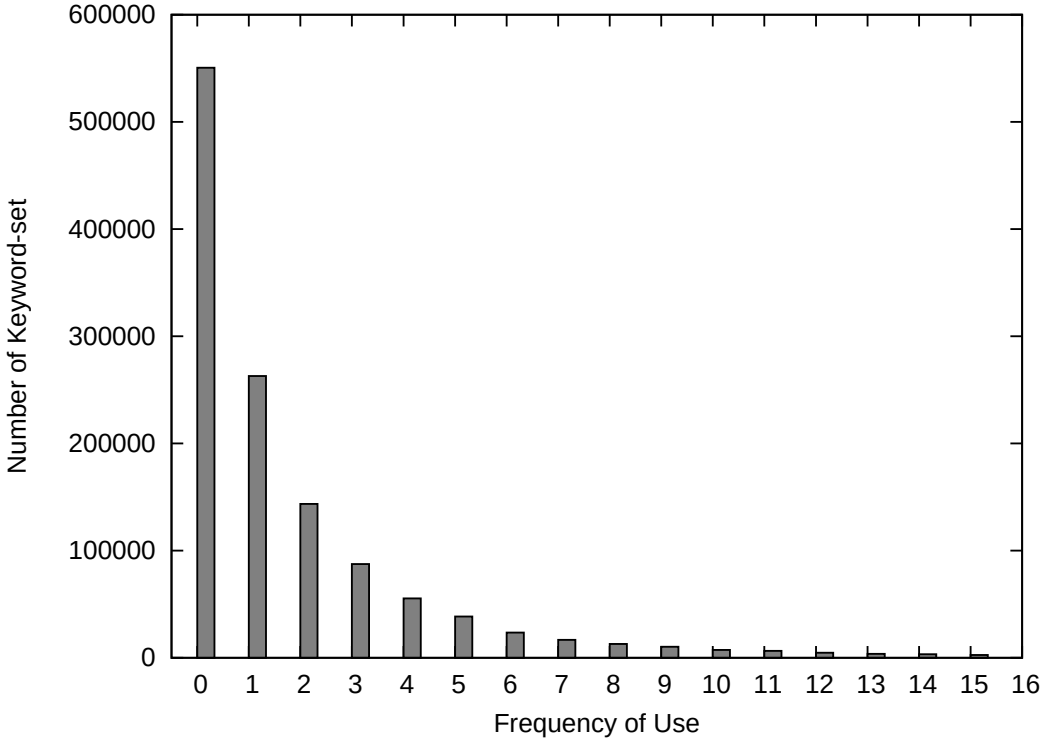


图 3-12 由查询产生的关键词集使用频率统计.

(1) 从生成索引的角度考虑：综合更多的查询的信息，有选择性地为某些关键词集生成索引，而不是查询的全组合，因为单条查询信息往往只反映单个用户对该关键词集关联性的认识，搜集足够多的信息后再触发索引节点的生成显然更为合理；

(2) 从删除索引的角度考虑：在实际的系统中，若节点较少，则每个节点可能需要维护多个的关键词集索引，而当系统的可用资源不足时，就必须选择性地删除某些不太重要的索引信息，长期没有使用的索引是可以被考虑删除的对象。

对于方法 (1)，我们可能更多地需要使用统计语言学的处理方法，如词语搭配 (Collocation) 关系分析等，这些都查出了本文的讨论范围，限于篇幅，本文只从方法 (2) 的角度考虑，但我们将看到，即使是单方面的优化策略也可以使空间效率得到显著提升。我们使用最少频繁使用 (Least Frequently Used，LRU) 删除策略，该策略会删除系统中长时间未被使用的关键词集索引 (第一层的索引除外)。为此我们设定索

引的生存时间 (Time To Live，TTL) 参数，并在每次访问时加入时间戳；仍然是每隔 10,000 个时间单位检查一次系统，若索引时间戳与当前时间相隔超过 TTL，则删除该索引，但不删除第一层的索引，因为这是用户发布的索引信息。我们模拟了 TTL 分别为 50,000、100,000、200,000 以及按层次设定的情况，其中，按层次设定的具体操作是，第 2 层 TTL 设为 50,000，往上逐层递减 10,000，直到 TTL 为 10,000 后不再递减。最终得到图 3-13 的结果，与无删除机制的索引生成策略 (图 3-11) 相比，系统的索引规模始终保持在一个较为稳定的水平上，而不是无限地增长，这将对系统稳定性产生积极影响。在较宽松的情况下 (TTL=200,000 时)，系统的索引数下降到 30 多万个，是无删除策略时的 26%；逐层递减的设定方法删除索引最多，其最终的索引规模不到原来的 10%，具体数据见表 3-2。

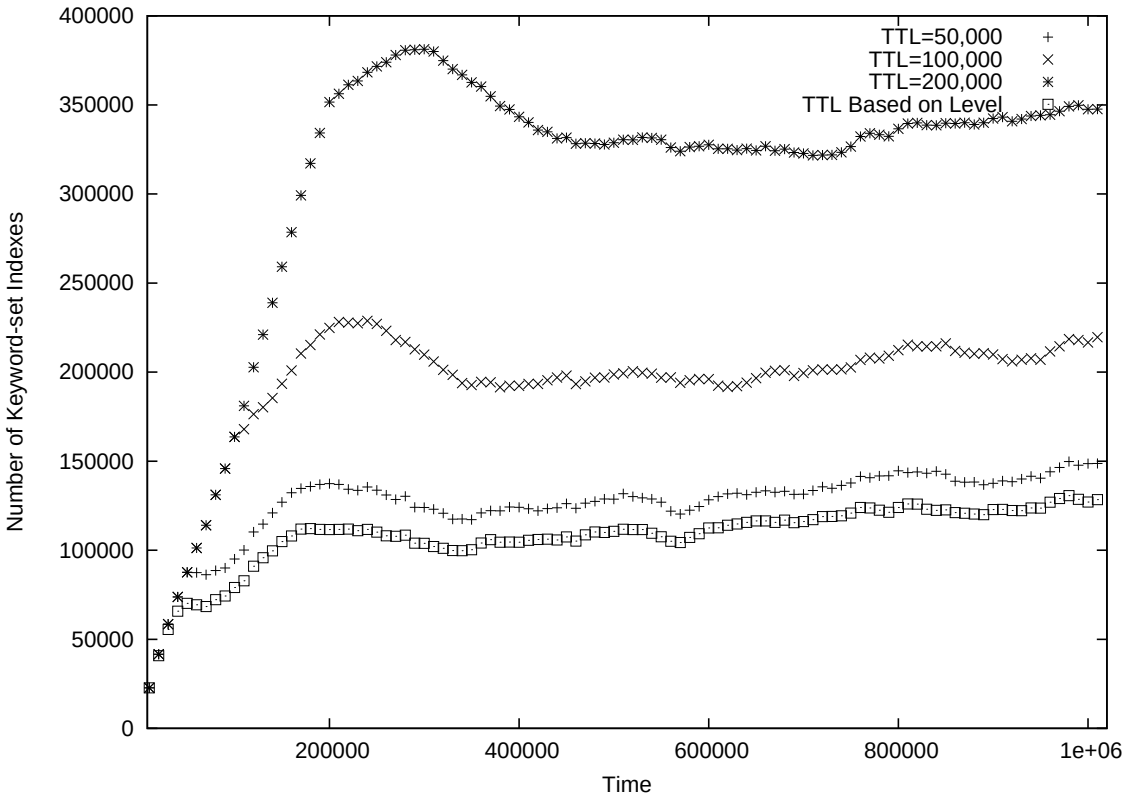


图 3-13 在不同的 TTL 设定下，系统关键词集索引的规模.

删除策略虽然降低了空间复杂度，但会牺牲少量的系统性能，因为上层索引的数量减少了，查询命中的概率会有所减小，但由于我们删除的大部分是很少用到的索引，所以系统性能不会受太大影响，图 3-14 显示使用删除策略后查询所需的平均求交

次数，正如我们的预测，平均求交次数稍有增多，但总体上系统依然能够得益于用户查询驱动的层次化索引。如表 3-2 所示，即使索引规模减少到原来的 10%(按层次设定 TTL 时)，平均每条查询所需的求交次数也只是比原来增加 0.078 次，依然处在一个较低的水平，所以 LRU 删除策略是有效的。

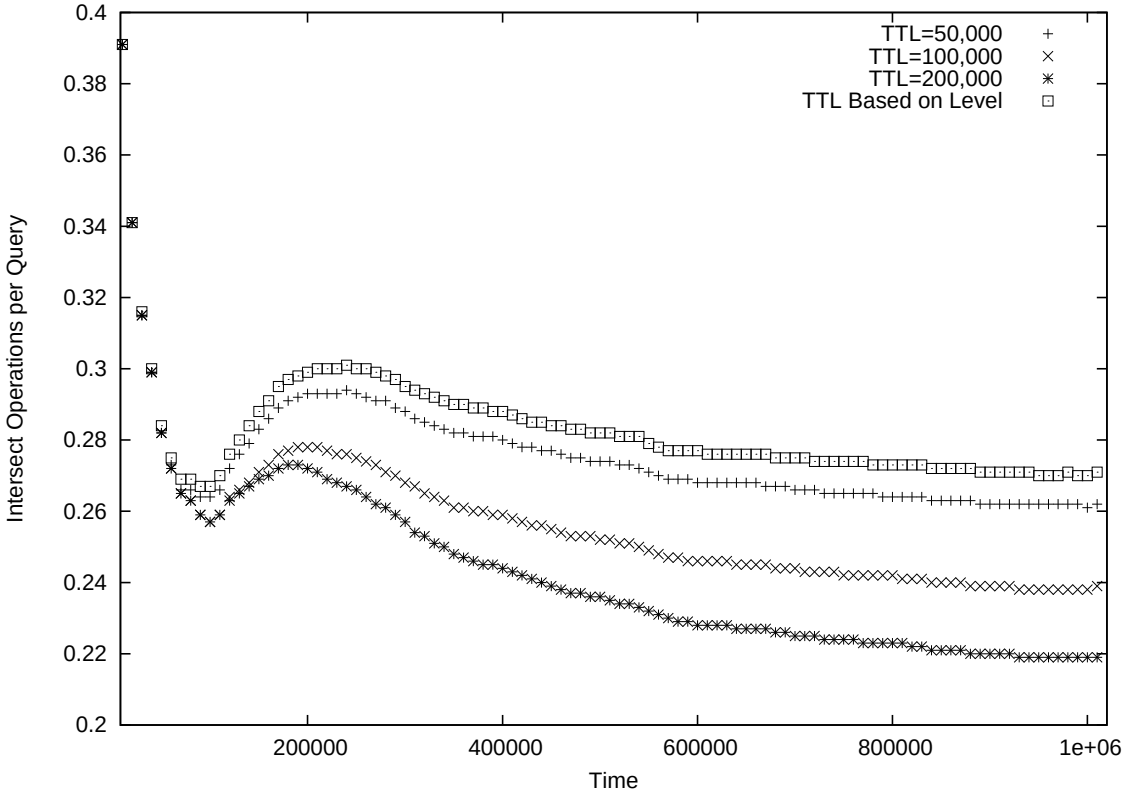


图 3-14 在不同的 TTL 设定下，查询所需的平均求交次数.

表 3-2 不同 TTL 设定的比较.

索引生存期 (TTL)	索引规模	总求交次数	平均每条查询的求交次数
$+\infty$	$1,302,981$	$195,346$	0.193
$200,000$	$348,388$	$221,771$	0.219
$100,000$	$219,962$	$241,538$	0.239
$50,000$	$148,713$	$264,890$	0.262
按层设定	$128,233$	$273,920$	0.271

在这里，我们只是为了方便模拟而设定较为固定的 TTL，事实上，可以将 TTL

设置和是否删除到期的索引的决定留给对等端，对等端通过比较其计算与储存的成本进行决定。因为小的 TTL 可以节省空间但会导致更频繁的求交运算，大的 TTL 可以减少求交运算但消耗较大储存空间，所以对等端可以作出更适合他自己的设置，又或者可以根据其系统运行时的情况动态地调整，这种通过发挥群体智慧进行决策的灵活性是传统缓存策略所没有的。另外，不强制用户删除索引的另一个原因是，当对等搜索系统增长到一定规模时，删除索引的必要性将会减少。正如我们在绪论中指出，对等计算的一个重要优势是可扩展性，理论上它允许任意多的节点加入到网络中，总体的可用资源是可以无限大的。对于百万级索引规模，只要参与节点数量也达到百万级别，那么每个节点平均亦只需负责维护少量的索引。而这样的用户规模是完全有可能的，据 Skype Journal[52] 的报道，P2P 语音通信软件 Skype[53](网络结构与 2.2 节介绍的 KaZaA 类似 [5]) 在 2010 年初录得超过 $2,200$ 万用户同时在线的记录。这个潜在扩展规模是传统搜索引擎所难以达到的，最新的数据显示，目前只有搜索引擎巨头 Google 的服务器规模达到百万级 [54]。

3.5 负载均衡

由 3.2.3 节和表 3-1 的分析可知，使用基于关键词的索引划分策略和用户的查询行为都会造成负载不均匀的现象：首先是储存负载的不均匀，由于每个关键词集的使用频率不一，有的关键词集频繁被使用，故其对应的文档集合将是巨大的，甚至有可能超出对等端提供的储存空间，相反，较为罕用的关键词集对应的文档集合将会较小，故其根节点有大量的空闲空间；另外，因为每个关键词集的受关注程度不同，某些热门的关键词集将会被大量的用户搜索，负责该关键词集的节点可能因为没有足够的响应能力和带宽而成为系统的瓶颈；相反，负责冷门关键词集的节点将会有较多的计算和带宽资源。显然，这两个极端都不利于系统资源的优化配置，系统需要引入调配资源的机制。

事实上，只要整个系统的资源是足够的，资源不足的节点可以向系统请求更多的资源，以处理繁重的任务，同时又不浪费系统的空闲资源。在结构化的对等网络中，最简单的方法就是向邻居节点请求资源，这里的邻居是指 ID 值最邻近的节点，大部分结构化系统都提供了获取邻居信息的方法。在新的 Tapestry 实现里，由于引入了叶

集的概念 (见 2.4.2 节)，使得获取邻居节点的信息变得更为方便，Chimera 库中获取叶集的函数是**route_neighbors**(见 4.1.1 节)，它返回按邻近程度从大到小排序的节点集合。

使用邻居节点来扩展资源有以下好处：首先是因为到邻居节点的路由跳数少 (在覆盖网络上只需一跳)，方便通信；此外，由于结构化对等网络都使用基于键值的查询路由方法 (见 2.4.2 节)，所以发送到某一节点的信息通常会经过其邻居节点，使用邻居节点来扩展计算或储存资源并不会影响路由方向；当资源仍然不足时，邻居节点会继续向其邻居节点申请资源，亦即是以原始根节点为中心不断往外扩大，而查询信息的路由路径却不断变短，原因是它不必到达根节点就获得了处理。对于搜索系统来说，这意味着查询越热门的关键词集，响应速度越快。

其实，扩展后的根节点相当于一个无结构的对等网络，随着扩展半径的增大，节点的通信效率将会下降，某些需要同步的全局信息 (如文档集合的规模等) 只能通过洪泛来传播，但相对于整个网络的规模，这个子网络的规模还是较小的，因此我们相信洪泛方式在这种情况下应该是可以接受的。实际上，无结构和结构化对等体系研究是可以互补的，因此，可以使用优秀的无结构对等网络模型来组织这个扩展子网，但本文不打算对此作深入研究。

3.6 搜索结果排序

上述讨论的优化策略都是针对搜索效率的，而所得到的搜索结果却是一个没有排序的文档集合，但用户往往只需要其中的少部分结果，故他们希望最有用的文档排在最前面，而忽略大部分不感兴趣的。搜索排序 (Ranking) 是颇具挑战性的研究课题，尤其在对等搜索领域中 (相关综述可参见文献 [8])。本章讨论简单的排序技术，并重点关注排序对搜索效率的影响。

如果对等搜索系统能够对文档集合排序，那么，系统的性能会在某些方面得到提升。因为搜索节点不必返回全部的搜索结果，排在最前面的 k 个结果 (Top-k) 已经基本满足用户所需，所以搜索过程消耗的带宽会相应减少，这也意味着搜索节点的负载将会减缓。但是，我们不能忽略排序所需的开销，如果要对结果作细致的排序，通常需要大量的资源，这并不符合我们保持高效性的原则，更有可能抵消了排序所带来的效

益。因此，我们更倾向的做法是：只对节点持有的文档集合作大致有序的排列，这样也可以避免传输整个文档集合，节点每次只返回一个最为相关的文档子集合，更细致的排序策略可应用在这个较小的集合上。

实际上，搜索排序就是对搜索结果按相关性大小排序，因此，首先要计算出结果的相关性大小。在 3.1 节中，我们给出了相关性的计算公式 (3.1 式)，该公式的计算很简单，但若要计算出较好的语义相关性，度量关键词权重则是较为重要的。当前的度量方法往往只是针对传统搜索系统，在对等环境下实施将会遇到较大的困难，下面简单讨论这个问题。

3.6.1 关键词权重度量

传统的 TF-IDF(Term Frequency–Inverse Document Frequency) 是公认最重要的关键词权重度量方法，该方法将从两个角度 (TF 和 IDF) 进行度量。首先看 TF 度量，该度量认为，关键词在该文档中出现的频率越大就越能反映该文档的内容，故权重越大。例如，假设关键词 k 在某一文档 d 中的出现频率为 $freq(d,k)$，则关键词 k 的在文档 d 中的 TF 值可使用下面公式计算 [55]：

$$TF(d,k)=\begin{cases}0 & \text{如果} freq(d,k)=0;\\ 1+\log(1+\log(freq(d,k))) & \text{其他.}\end{cases} \tag{3.9}$$

然而只是 TF 度量不足以去掉一些对度量相关性没有意义的应删去词 (Stopwords)，如汉语中“是”、” 的”、“了”等，因此要引入倒排文档频率 IDF，根据 IDF 的定义，如果关键词 k 出现在越多的文档中，其区分文档的能力就越弱，故其重要性也会降低。关键词 k 的 IDF 值可由如下公式计算 [55]：

$$IDF(t)=\log\frac{1+|D|}{|D_k|}. \tag{3.10}$$

其中 D_k 是包含关键字 k 的文档集合，其模表示集合的元素个数。在 VSM 中，将 TF 和 IDF 结合在一起，形成 TF-IDF 度量，关键词 k 在文档 d 中的权重为：

$$weight(d,k)=TF(d,k)\cdot IDF(k). \tag{3.11}$$

在实际的应用中，计算公式 (3.9) 和 (3.10) 会根据实际情况进行修正和优化，使相关性的度量更为准确。虽然 TF-IDF 度量是现时大部分搜索引擎计算相关性的基本方法，但该方法却较难在对等搜索中实现，难点在于 IDF 的计算。由 IDF 的定义可以知道，IDF 度量的是关键词在全局中的重要性，故必须要知道文档的全局统计信息，如公式 (3.10) 中的 $|D|$ 和 $|D_k|$。

在我们的分布式索引结构中，$|D_k|$ 可从 k 的根节点获取，但前提是节点要较稳定，离开网络时要将信息转移给接管的节点；另外，在使用资源扩展 (见 3.5 节) 时，多个节点同时负责一个关键词，它们的文档集合中可能存在相同的项目，故它们之间要经常交换信息，以确保 $|D_k|$ 的正确性。而对于 $|D|$，通常就较难获取了，因为系统并没有一个负责统计全局信息的节点，即使有，恐怕 $|D|$ 的变化也将会非常大，原因是对等网络的动态性 — 网络中的节点随机地加入和离开网络，可用的文档数量也会随之变化。这就是在对等网络中使用 TF-IDF 的难处，若使用基于文档的索引划分方法 (见 3.2.2 节)，则甚至连获取 $|D_k|$ 也会相当困难。

有研究提出，可使用其他的度量绕过该问题，如 PlanetP 系统 [14] 用所谓的倒排对等端频率 (Inverse Peer Frequency, IPF) 取代 IDF 度量，关键词 k 的倒排对等端频率用公式 [14]

$$IPF(k) = \log(1 + \frac{|N|}{|N_k|}) \tag{3.12}$$

计算，其中 $|N|$ 是网络中的对等端总数，$|N_k|$ 是共享文档中含关键词 k 的对等端数量。与 IDF 类似，IPF 度量也可表示该关键词区分能力。相比海量的文档，网络中的对等端数量要少得多，而且并不会无限地增长，故 IPF 是一个较为合理的度量方式。与本文系统使用的结构化网络不同，PlanetP 系统是建立在无结构对等网络上的，它需要使用 Gossiping 算法 [14] 在网络中传播各个节点的汇总信息。而在基于结构化网络的系统中，$|N_k|$ 可由关键词 k 的根节点统计，$|N|$ 可通过一些专门的节点进行计数。除了 IPF 外，其他的度量方法也是可能的，因为这是一个开放性的问题。无论如何，新的度量似乎都难以与传统搜索引擎的方法相比，除非对等搜索系统能够获取较多的全局统计信息，因此，不妨将这些新方法称为近似度量。

3.6.2 文档集合的排序

虽然使用近似度量只能获得近似的排序结果，但这些度量的计算较为简单，消耗的系统资源较少，且最重要的是，近似的排序也有利于系统的性能提升。如图 3-15 所示，节点 $N_{k\wedge k'}$ 需要对 N_k 和 $N_{k'}$ 的文档集合取交集，但由于 N_k 和 $N_{k'}$ 的文档集合较大，需要分块传输，假设 N_k 将其文档集合 D_k 划分成 $\{D_k^{(1)}, \cdots, D_K^{(n)}\}$，则它在第 i 次传输中只给 $N_{k_1\wedge k_2}$ 发送集合 $D_k^{(i)}(1 \le i \le n)$；而如果 D_k 是近似排序的，那么我们可以期望对任意 $i < j$，有

$$rate(d^{(i)}) \ge rate(d^{(j)}), \qquad d^{(i)} \in D_k^{(i)}, d^{(j)} \in D_k^{(j)} \tag{3.13}$$

其中 $rate(d)$ 表示文档的得分 (分数越高排名越前)。值得注意的是，(3.13) 式并不一定对所有的 $d^{(i)}$ 或 $d^{(j)}$ 成立，还与排序的近似程度和列表 D_k 划分的粒度 (即子列表 $D_k^{(i)}$ 的大小) 有关：近似程度越高或划分粒度越大，(3.13) 式成立的概率越大；另外，近似排序并不确保子列表 $D_k^{(i)}$ 有序，但这并不影响求交运算。因此，只要根据排序的近似程度选择适当的划分粒度，节点 $N_{k\wedge k'}$ 不仅可以快速获得较好的文档集合，而且可以按需索取，忽略排名靠后的文档。若 $N_{k\wedge k'}$ 需要获得更好的排序结果，则可在近似排序的基础上继续使用更为复杂的算法，但这时要处理的数据通常是较少的。

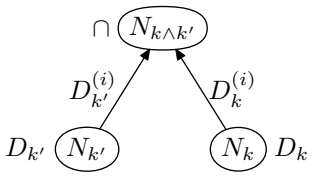


图 3-15 排序后的文档集合传输.

3.6.3 用户反馈

对等网络技术的提出，增进了用户与用户之间的交流，使系统表现得更人性化，所以用户参与是系统的重要组成部分。正如 3.3.3 节提出的查询驱动搜索策略一样，用户对搜索结果的选择也可以帮助系统改进排名结果，这是传统搜索引擎普遍使用的方

法；此外，也可以使用社会化网络服务 (Social Network Service，SNS) 的用户评分机制，让用户为感兴趣的文档打分，分数返回给查询对应的关键词集根节点，根节点根据这些信息修正文档的得分，然后将反馈信息转发给低层相关的节点，让它们也做同样的修正和转发。因此，系统的排序质量就可以在用户的参与下不断地得到改进和提升，这是当前对等搜索研究的一个重要方向，Faroo 团队 (见 1.2.3 节介绍) 已在进行相关的开发尝试。

3.7 本章小结

本章简单介绍了信息检索领域的向量空间模型 (VSM) 和倒排索引结构；讨论了结构化对等网络上的倒排索引划分策略，提出了层次化的分布式索引结构和相应的多关键词搜索策略，并使用 Python 程序模拟了系统的运行，验证了相关增强机制的有效性；此外，还初步研究了负载均衡和搜索排序的解决方案，且讨论了近似排序策略对提升搜索效率的影响。在下一节，我们将给出基于层次化索引结构的多关键词搜索系统原型实现架构，为进一步的研究和开发打下基础。

第 4 章 系统原型与应用

本章给出一个支持多关键词搜索的对等系统原型 (Multiple Keywords Search System, MKSS) 及其简单的应用接口。如图 4-1 所示，系统建立在结构化对等网络 Tapestry 之上，使用最新的 Tapestry 实现 Chimera(1.20 版本)[38]；由于 Chimera 库是用 C 语言编写的，所以应用程序必须使用其 C 接口函数，但为了加快原型系统的开发速度，我们选取了部分 Chimera 库函数，并用 SWIG[56] 转换成 Python 接口，使 Chimera 成为 Python 的扩展模块；原型系统主要分为三个模块：

- Receiver 模块用于接收底层 Chimera 消息，并提交 Manager 模块作分析处理；
- Manager 模块负责系统资源的分配和调度，资源包括索引数据和任务队列，Handler 模块从这里获取数据和任务；
- Handler 模块由一系列的处理程序组成，用于对不同的消息作响应处理；

整个搜索系统最后被封装成一个类 (Mkss)，为上层应用提供多关键词搜索服务。原型系统使用 Python(2.5 版本) 编写，并在 Xubuntu(8.04.4 版本) 系统下调试运行，下面详细介绍实现细节。

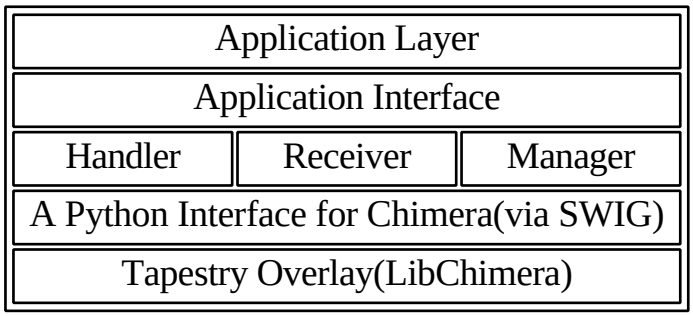


图 4-1 多关键词搜索系统架构.

4.1 覆盖网络

4.1.1 Chimera 库

Chimera 是一个实现结构化对等网络模型 Tapestry(2.4 节) 的 C 函数库，比过去的 JAVA 实现更加快速和轻量，该函数库提供的应用程序接口 (API) 符合论文 [21] 倡导的标准，本文已在 2.4.5 节简单介绍过三个唤醒接口，此外，该库还提供了很多其他的接口函数，下面只介绍在本文原型系统中使用到的，其余的接口介绍可见 Chimera 的文档 [38]。

ChimeraState *chimera_init (int port);

在加入对等网络前，先要指定通信端口，**chimera_init**函数使用端口**port**初始化对等端，并返回其**ChimeraState**结构的指针，该结构用于记录各个 Chimera 模块的状态信息，其定义在头文件“host.h”中。

void chimera_setkey (ChimeraState *state, Key key);

在获得**ChimeraState**信息后，可以使用**chimera_setkey**设定节点的 ID 值，其中**Key**结构用于保存 ID 值信息，ID 值默认使用 16 进制，长度为 40 位 (即 160 位二进制数)，可在文件“key.h”中修改这些定义；函数**str_to_key**可将字符串类型的 ID 转换成**Key**类型。

void chimera_register (ChimeraState *state, int type, int ack);

该函数用于注册消息的属性，因为 Tapestry 使用 AID 区分消息的应用类型 (见 2.4.2 节)。消息类型**type**用整型表示，值小于 10 的类型保留给系统消息使用；**ack**为确认类型，1 表示消息需要确认，2 表示无需确认。

void chimera_update(ChimeraState *state, chimera_update_upcall_t func);

void chimera_forward(ChimeraState *state, chimera_forward_upcall_t func);

void chimera_deliver(ChimeraState *state, chimera_deliver_upcall_t func);

这是我们在 2.4.5 节介绍过的唤醒接口，用户可根据需要定制响应函数**func**，当然，也可以不定义，表示应用程序无需对该事件做任何响应。因为我们的实现不涉及底层路由决策，所以系统只定制了对 update 和 deliver 的响应函数。

void chimera_join (ChimeraState *state, ChimeraHost *bootstrap);

在完成了上述的初始化工作后，对等端使用 2.4.3 节介绍的方法加入到对等网络中，函数**chimera_join**就是用来完成这个过程的，其中**bootstrap**为已知的入口节点，若为**NULL**则表示节点是第一个加入到网络中的。

void chimera_send (ChimeraState *state, Key key, int type, int size, char *data);

当对等端成功加入网络后，就可以用**chimera_send**向其他对等端发送消息，**key**为目标对象的键值，消息将最终到达其根结点；消息的类型**type**必须是初始化时用**chimera_register**注册的类型，并要指定消息的长度**size**和发送数据的指针**data**。

ChimeraHost **route_neighbors (ChimeraState *state, int count);

正如 3.5 节所述，上层应用可能要使用邻居节点的资源来解决负载过重的问题，因此需要获得邻居节点的信息，Chimera 库提供的**route_neighbors**函数可返回**count**个邻居节点的信息，且结果按近邻程度从大到小排序。

以上就是我们的原型系统使用到的 Chimera 库函数，此外，Chimera 库还有很多其它可供调用的函数，但因为上述函数已经可以满足本文原型系统的实现，所以，我们不打算对其他函数的进行介绍。事实上，Chimera 对 Tapestry 模型的实现仍不完整，很多功能还有待进一步的测试和完善。因此，本文的多关键词搜索系统连同底层对等网络系统都仍只是初步的原型，有待更进一步的研究。

4.1.2 Pchimera

早期的 Tapestry 是用 JAVA 实现，这更方便上层应用的开发，但难以确保底层网络的效率，因为对等网络已经是建立在 TCP/IP 模型的应用层之上 (如图 2-1)，且覆盖网络与物理网络可能会有不一致的情况 (见 2.4.4 节)，所以为了加快计算速度，Tapestry 的研究团队用 C 重新编写实现是有必要的，可惜他们没有将 Chimera 库

封装成其他高级编程语言的接口，这并不利于提高上层应用开发效率。为了能够使用 Python 快速建立原型系统，我们将用 SWIG[56] 将 Chimera 库转换成 Python 的扩展模块，称为 pchimera，该模块提供搜索系统所需的 Chimera 功能。

在 pchimera 中，我们并不是将 4.1.1 节介绍的 Chimera 函数原样封装，而是将其浓缩成简单的三个基本函数和一个全局变量，这样可简化原型系统的实现，令开发的重点落在搜索技术的实现上。下面具体介绍 pchimera 提供的函数和变量。

ChimeraState *pjoin(char *joinhost, int joinport, char *nodeid, int port);

函数**pjoin**实现对等端的初始化和加入网络的过程，返回值是指向对等端的**ChimeraState**结构的指针，后面可使用该指针来引用对等端。该函数需要提供四个参数，前两个参数是入口节点主机名和端口号，后两个参数是对等端将要使用的 ID 值和端口号。如果节点是第一个加入网络的，第一个参数应为一个空的字符串，而第二个参数将会被忽略。

void psend(ChimeraState *state, char *dest, char *msg);

在 pchimera 中发送信息使用函数**psend**，该函数的实现其实是调用**chimera_send**，但参数更少，只需要提供对等端的状态表**state**、目标对象键值**dest**(字符串形式) 和消息内容**msg**。参数少是因为我们的原型系统将使用自己的消息类型，所以在**pjoin**的初始化过程中，我们只注册了一种值为 18 的消息类型 (宏定义为 MKSS)，Chimera 将该类型的消息都上传给搜索系统处理；同样，**psend**也只发送 MKSS 的 Chimera 消息。

void pneighbors(ChimeraState *state);

该函数能够获取对等端的邻居信息，因为本文并未对负载均衡问题进行更为深入的研究，所以，我们的原型系统还没有做相关实现，但该函数可留待将来使用。

pchimera 除了提供上述三个函数外，还有一个全局的整型变量**RPORT**，用来定义 Chimera 与上层系统的通信端口号。为了解析该端口的作用，我们先简单介绍原型系统中的 Reveiver 模块 (其所处位置可见图 4-1)。前面已经介绍过，搜索系统将使用自己的消息类型，pchimera 会将所有的 MKSS 消息提交到搜索系统，系统再对消息进行解析处理。其中，Receiver 模块就是专门用来接收底层网络提交的消息。pchimera 与 Receiver 之间用套接字 (Socket) 通信，Receiver 是一个使用 TCP 的服务端程序，它监听着本地端口**RPORT**，为系统收取消息；在底层，pchimera 定义

的响应函数**deliver**和**update**将从对等网络中收到的 MKSS 消息发送到**RPORT**端口上。系统模块之间消息传递如图 4-2 所示。

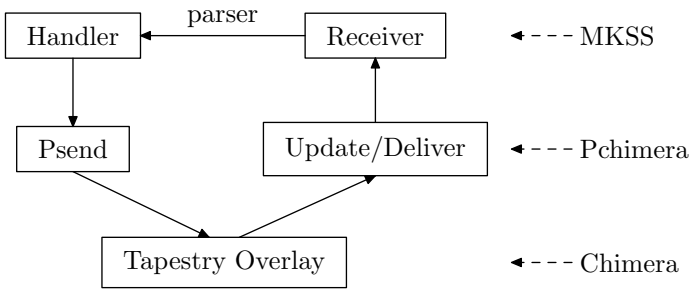


图 4-2 系统内部的消息传递.

此外，为了在单台机器上运行多个节点，我们使用多线程实现，但如果使用 Chimera 原有的响应函数格式，则无法区分究竟是那个节点在唤醒上层应用，所以，我们稍微修改了 Chimera 的代码，使得在实现 pchimera 的响应函数**update**和**deliver**时允许增加参数**ChimeraState *state**，利用**state**就可以在上传信息中加入被唤醒的对等端信息，以便系统将消息递交给相应的对等端 (线程) 处理。

4.2 多关键词搜索系统原型

本节介绍的搜索系统原型将在扩展模块 pchimera 的基础上，实现上一章提出的多关键词搜索技术，并尝试给出一些可被调用的应用程序接口，为进一步开发更为复杂的搜索系统作基础。同时为使实现保持清晰性，我们采用模块化的设计方法，将系统分为三个主要的模块 (如图 4-1 所示)，图 4-3 展示了模块之间交互，Receiver 将收到的消息发送给 Manager，Manager 对消息进行解析后将需要处理的任务放在相应任务队列上，Handler 从队列中获取任务，如果没有任务则阻塞；另外，Handler 在处理任务时会向 Manager 索取数据资源或反馈信息。下面详细介绍各个模块的工作原理。

图 4-3 模块间的通信.

4.2.1 系统消息

Receiver 模块其实是为封装 Chimera 库而产生的，因为 Chimera 的唤醒接口不能直接调用 Python 的函数，所以最好的办法就是使用 Socket 进行通信。在 pchimera 实现中，update 和 deliver 将从 Chimera 收到的消息发送给 Receiver。因此，Receiver 要处理的工作十分简单，那就是监听**RPORT**端口，获取消息。在原型系统的初期实现中，我们尝试让 Receiver 同时对获取的信息进行解析，然后直接递交给 Handler，但当消息数量较多时，Receiver 的接收速度就会成为系统的瓶颈：首先，建立 TCP 连接已有一定的开销；另外，解析消息也耗费一定的时间；并且 Receiver 为单个线程，而 Handler 是多线程，所以最终会出现供不应求的状况。为此，我们最终将解析工作留给 Manager，Receiver 只需将消息放到 Manager 的处理队列即可，Manager 中会有专门的解析线程。

在 Manager 中，parser 是解析消息的线程，它负责分析消息，并将消息放到恰当的 Handler 任务队列中。为了使 parser 更好地处理消息，系统内部使用的消息具有一定的格式，如图 4-4 所示，系统消息可分解为 5 个域，实际上在对等网络中传输的只是后四个域的内容，第一个域其实是 pchimera 上传时补上的，它是接收方 ID 值，因为我们的系统将允许在单个进程中以多线程运行多个节点，所以区分接收方是有必要的；第二个域是目标对象的键值，Chimera 利用它进行路由定位；第三个域是发送消息的节点 ID 值，方便对方回复时使用；第四个域是消息的类型，parser 用它来判断将消息递交给哪个处理程序 (handler)，类型值为四个字符，且可以进一步拥有子类型，子类型值用两个字符表示，具体类型如表 4-1 所示；最后一个域是消息的净荷，包含 Handler 需要用到的数据信息。parser 从消息中提取各个域的信息，但它只会查看 ID(dest) 和类型信息，因为这是它用来选择对等端 Handler 的依据，以下介绍各种的消息类型。

图 4-4 系统使用消息格式.

因为只是原型，目前系统的消息类型还比较少，但足以令系统运行起来。GIFO(Get Infomation) 消息用于获取信息，如子类型 CK(Check) 用于试探关键词集索引的可用性，DL(Document List) 可获取关键词集对应的文档集合；PLSH(Publish) 消息用于发布索引信息，具体格式如图 3-4 所示；RIFO(Reply Infomation) 消息是节点对 GIFO 和 SRCH 消息的回复结果；SRCH(Search) 是搜索消息，本质上也属于 GIFO 类型，但考虑到搜索是系统的重要任务，故独立成一种类型，搜索结果以 RIFO 消息的 FN 类型回复。表 4-1 的最后一列为各类消息对应的处理程序，也就是 handler(下一节介绍)；对于最终结果 (FN 类型) 的处理，可留给以后添加的 handler 或上层应用。

表 4-1 系统消息类型.

消息类型	子类型	用途	处理程序
GIFO	CK	获取关键词集信息	gifo_handler
	DL	获取文档集合	
PLSH	-	发布信息	plsh_handler
RIFO	CK	回复关键词集信息	srch_handler
	DL	回复文档集合	
	FN	回复最终搜索结果	-
SRCH	-	搜索请求	srch_handler

与 Receiver 不同，parser 并不是将消息原样地递交给处理程序，因为有些域，如目标节点 ID 和类型域，它们只供 parser 选择 handler 时使用，一旦选好了 handler，handler 没必要再次查看，所以 parser 可将这些域去掉后再递交，这样可以节省一点空间。另外，随着系统任务的增多，消息类型也会越来越多，将来消息系统的设计还需要更为谨慎的考虑，虽然本文不打算作进一步讨论，但可以肯定的是，将 parser 作为独立于 Receiver 运行会使得系统更具扩展性，因为必要时可以使用更多的 parser 处理线程。

4.2.2 消息处理

由表 4-1 可以看到，系统里面有各种不同类型的 handler，而且每个节点拥有自己的 handler 集合，在实现中，我们将 handler 集合抽象为 Handler 类，节点在加入系统时需要调用 Handler.addnode() 初始化自己的 handler 集。上节简单介绍了 parser 与 handler 的关系，这只是 Manager 和 Handler 模块交互的一个方面，此外，Handler 在处理过程中还需使用到 Manager 管理的数据资源，同时也会向 Manager 反馈信息，从而使 Manager 更有效地管理数据 (如图 4-3 所示)。

与 Chimera 类似，搜索系统是由事件驱动的，parser 相当于 upcall 接口，handler 相当于响应函数，它们之间使用 Python 的队列 (Queue) 通信。parser 只要将任务放到 (put) 相应的队列，handler 就从队列中获取 (get) 任务，当队列为空时 handler 会阻塞，直至有任务到来。因为 Queue 是线程安全的，所以系统允许有多个 parser，也允许节点拥有多个同类型的 handler，这有利于系统扩展。

在当前的实现中，每个节点有三个 handler：

(1) plsh_handler 将发布信息的内容 (见图 3-4) 写入数据库，而数据库由 Manager 管理。在递交数据给 Manager 前，plsh_handler 可对发布信息作一些处理，例如检查数据库中是否有该项目，又或者是该项目的个别信息是否有所修改等；另外，plsh_handler 可以用来过滤信息，如恶意节点大量发布的垃圾信息等，这些都是将来系统可能具有的增强特性。

(2) gifo_handler 从数据库获取信息，对于 CK 消息，只需检查数据库当中是否有该项目 (或对应的文档集合长度)，响应速度较快；而对于 DL 消息，则需要从数据库中取出相应的文档集合，因而速度较慢；事实上，可以增加 BF 消息，使得节点可以获取文档集合的 Bloom Filter(见 3.3.2 节)，这时可以减少数据传输，但原型系统尚未实现该功能，实现时可以使用 pybloom 模块 [57]；另外，在以后引入排序功能后，gifo_handler 可分块传输文档集合 (见 3.6 节)。

(3) srch_handler 是搜索系统的关键处理程序，因此我们将它作为独立的线程。srch_hanlder 使用 3.3.3 节介绍的搜索方法进行工作，先检查自己是否已经拥有结果，有则直接

返回 FN 消息，否则使用 CK 消息探查可用节点 (如图 3-9(a))，这个过程可调用 srch_handler 的 check 方法实现，也就是说 CK 消息是由 check 产生的，check 返回关键词集的可用索引集合，然后节点根据可用信息；选择文档集合或求交节点 (如图 3-9(b))，该过程由方法 intersect 实现，intersect 使用 DL 消息获取文档集合 (或用 BF 获取 Bloom Filter)，为了充分利用分布式的资源，可将求交运算转移到没有命中查询的节点上，如图 3-9(b) 所示。

除了 plsh_handler 会向 Manager 递交数据外，gifo_handler 和 srch_handler 也会有信息产生，因为层次化的分布式索引是由用户查询驱动构造的 (3.3.3 节)，驱动信息可从 gifo_handler 和 srch_handler 中获取。如图 3-9(b) 所示，求交节点通常会保留求交结果，即 srch_handler 会请求 Manager 创建新的关键词集索引；另外，即使没有成为求交节点，gifo_handler 也可以请求 Manager 建立缺失的关键词集索引 (也就是 CK 结果为 NO 的)。但这些请求信息往往只作为对 Manager 的反馈，最终是否建立索引则由 Manager 决定，因为我们认为，系统应该更谨慎地建立索引，而不是完全由用户查询驱动，正如我们在 3.4 节的分析中指出，查询信息中也存在大量“噪声”，所以索引数据的管理显得尤为重要，这也是我们将资源管理独立为一个模块的原因，下小节详细介绍 Manager 模块的资源管理。

4.2.3 资源管理

在以上两小节的介绍中，我们始终离不开 Manager 模块，可见其重要性，因为 Manager 是系统资源分配和调度的核心。之前已经介绍过 Manager 里面的 parser 和数据库，但为了更好地管理节点的索引数据，我们加入了 indexer 线程，由 indexer 负责索引数据的增删。

Handler 将反馈信息递交到 indexer 的任务队列，然后 indexer 根据一定的规则判断是否接受反馈的请求；另外，除了增添索引，indexer 还会周期性地检查数据库，清除某些不符合条件的索引数据 (3.4 节讨论了 LRU 的删除策略)。在当前的系统实现中 indexer 并未完整实现，因为更好的索引增删策略还可进一步研究，但是，我们认为较有前途的实现是统计学习的方法，也就是说，indexer 应由数据驱动：所有 Handler 的

反馈信息和数据库中索引使用情况都应作为 indexer 的训练数据，不断更新判断规则，令下一次的决定更合理。

4.3 多关键词搜索系统的应用

在绪论中我们指出，多关键词搜索技术是其他复杂搜索的基础，因为不管查询如何复杂，最终都离不开带有逻辑运算 (NOT，AND，OR) 的关键词查询，而本文讨论的多关键词查询实际上就是 AND 查询，至于 NOT 和 OR 查询，搜索过程较为简单：对 NOT 来说，通常是在某个结果的基础上剔除含有某关键词集的文档，故只需在该关键词集的根节点上过滤先前结果即可；对于 OR 查询，在分布式的对等环境下则相当有优势 — 查询可以并发处理，系统只需将结果简单合并即可，当然要剔除重复的项目。因此，一旦 AND 查询的处理问题得到解决，支持逻辑运算的关键词查询基础就具备。

正因为多关键词搜索的基础性作用，我们的原型系统有必要向上层应用提供接口，以方便进一步的研究与开发。目前，原型系统有两个基本接口:

- **publish**: 该接口用于发布共享文档信息，参数为文档的索引信息：包括文档 ID，文档 URL，关键词集及其权重，publish 方法根据该信息产生多条发布消息并发送到相应的根节点 (如图 3-4)，考虑到根节点的负载，发布信息不会被回复。

- **search**: 该接口提供多关键词搜索服务，参数为查询字符串，查询中的多个关键词用空格分开，search 计算查询的键值并发送到相应的根节点，根节点将搜索结果以 FN 类型的消息回复。

除了基本接口，用户可以方便地修改系统内部的处理程序 (handler)，因为各个处理程序是较为独立的，无需顾及太多的依赖关系；用户也可以定义新的消息类型及其相应的 handler。

事实上，因为用户的查询大多是 AND 查询，所以多关键词搜索系统也可以发展成简单的对等搜索引擎，但这要求系统在自然语言处理 (如分词，搭配)、排序 (见 3.6 节) 和安全性等方面有更高层次的提升。

4.4 本章小结

本章结合具体的结构化对等网络模型函数库 Chimera，详细介绍了多关键词搜索原型系统的实现架构。为使用 Python 语言快速建立原型，我们实现了从 Chimera 库到 Python 扩展模块 (pchimera) 的封装；此外，由于使用模块化的设计方法，系统分成消息接收 (Receiver)、消息处理 (Handler) 和资源管理 (Manager) 的三大模块，本章分别介绍了各模块的功能和协作；为方便后续的研究与开发，原型系统保留了需要完善和改进的实现，并初步给出基本的应用程序接口。

第 5 章 总结与展望

5.1 本文总结

随着对等网络技术的发展，各种创新的应用将建立在对等的架构之上。其中，对等搜索技术以其可扩展、高容错、深度挖掘文档和成本低廉等优势引起了学界与业界的极大关注，但由于对等网络的动态性、自治性等特点，传统的搜索技术难以移植到对等环境之中，本文分别从对等网络体系和索引组织结构的方面讨论了多关键词搜索系统的基本架构：

首先，本文介绍了三种主流的对等网络体系结构：混合式对等网络、无结构对等网络和结构化对等网络。严格来说，原始的混合式体系不算是纯粹的对等网络系统，因为它有中心服务器支持，后来超节点的引入扩展了服务器，并极大地开发了网络的异构性，但它不再具备快速准确的查询路由，因为很大程度上，引入超节点后的网络相当于无结构的对等网络；所谓的无结构对等网络，指的是网络没有严格的拓扑特征，节点与节点间的连接是随机的，这样的网络结构难以实现快速的路由定位，故不利于快速搜索技术的实现；我们最终选择的是结构化的对等网络，它能支持高效的键值查询，因而能够快速获取对象索引信息，这是本文设计的多关键词搜索技术的基础。在我们的原型实现中，使用了结构化模型 Tapestry，该模型较多地考虑了拓扑一致性，而且有规范的接口。

在结构化对等网络的基础上，本文介绍了支持快速搜索的倒排索引结构，并分析其划分策略，认为在结构化的对等环境中，选择基于关键词的倒排索引划分策略是较为合理的，亦即是关键词对应的文档集合由其根节点维护。因此，整个对等网络系统可构成一个分布式的倒排索引。但使用原始的分布式索引结构需要大量的求交运算，既降低了搜索的速度，也消耗较多的计算资源，研究时间和空间复杂度相对合理的搜索策略是本文的重点。

在当前储存成本不断下降的情况下，用空间换取时间的想法是合理的，因此我们提出了层次化的分布式索引结构，令系统从用户查询中获得启发，自动创建一些关键

词集的索引信息，在层次化的索引结构下，搜索过程有望减少求交运算次数。本文的模拟实验显示，求交次数最多可减少近 85%。然而，随着用户查询的不断增多，系统产生的索引信息也越来越多，但有用的信息却是有限的，故系统储存效率的提升空间是较大的。模拟实验显示，周期性地删除过期的索引信息可大大减少索引规模，且系统仍然能保持较少的求交运算次数 (见表 3-2)。事实上，更智能的索引创建和删除策略是可能的，例如使用机器学习和自然语言处理等技术。

虽然基于关键词的索引划分策略有助于高效搜索的实现，但该策略会导致负载不均衡的现象，本文讨论了简单的解决办法：负载过重的节点可使用邻居节点扩展资源，扩展后的根结点实际上是一个无结构的子网络，全局信息只能依赖洪泛的方式传播。当扩展范围较小时，这种方法既简单又高效，而且还能加快热点查询的路由；但当扩展范围达到一定程度时，考虑使用更好的无结构网络组织方式是必要的。

除了索引结构和搜索策略外，我们还初步讨论了文档集合排序的问题，指出简单但近似的排序策略也能够提高系统性能。因为用户越早获得需要的结果，搜索过程就越早结束，系统就不必传输排名靠后的文档信息。如果需要向用户呈现更准确的排序结果，复杂的排序算法可用在近似排序的结果之上；此外，正如查询驱动的搜索策略一样，用户对搜索结果的反馈也可用作改进排序质量。

在以上讨论和分析的基础上，本文给出了多关键词搜索原型系统的实现架构，为后续研究和开发提供了基础的平台。

5.2 未来展望

由于本文的讨论更多地着重于搜索系统的基础方面，所以对其他的增强机制并未予以充分的考虑。对于后续工作，我们提出了以下的展望：

(1) 关键词集索引的创建和搜索。本文提出的索引创建和搜索策略主要是基于查询词集的全组合，当大部分的用户查询含有较少的关键词时，这个策略在分布式的环境下是可行的，但如果能进一步考虑关键词之间搭配关系，那么创建出来的索引可用性将会更高，搜索时选取词集也更有针对性。因此，从用户查询中进一步挖掘出词语搭配关系将是进一步增强系统性能的研究方向。

(2) 负载均衡与动态性。本文只给出了简单的负载均衡策略，而并未讨论动态性的问题。事实上，可以使用“ID 近邻复制”[5] 的办法解决动态性的问题，这与解决负载均衡的“近邻扩展”方法是相似的。关键是如何实现高效的数据同步方法，这是后续研究需要考虑的问题。

(3) 搜索结果排序。本文只是初步讨论了排序对提高搜索效率的影响，并未深入地探讨高质量的排序算法。传统的排序算法在对等环境中实施都遇到较大的困难，原因是对等网络的动态性与自治性使得全局信息难以收集，因此有效的对等搜索排序可能需要一种完全不同的权重度量方法，例如，基于社会化网络的用户评分方式或许是较有前途的一种，无论如何，对该问题的研究仍需要较大的创新。

(4) 系统开发与测试。本文给出了原型系统的基本实现架构，大量的增强特性仍有待实现。另外，本文的模拟实验只对系统的较少方面进行测评，其他方面 (如上述几点) 的实验仍需要更多的研究，但寻找更大规模的测试平台将会成为系统测试面临的挑战，因为对等网络系统是一个纯分布式的系统，其性能测试难以在较少的机器上表现出来，大规模的测试是必要的，最好是将系统投放到真实的使用环境中。因此，对等搜索系统的研究应保持开放，让更多的人可以加入到开发和测试的团队中。

对等搜索技术是一个较新的研究领域，尽管充满着大量的问题和挑战，但简单性、模块化和社会化都将是我们在今后设计中经常使用的原则。

参考文献

[1] Ralf Steinmetz and Klaus Wehrle(Eds). Peer-to-Peer Systems and Applications[M]. Berlin Heidelberg: Springer-Verlag, 2005. 王玲芳, 陈焱译. P2P 系统及其应用 [M]. 北京: 机械工业出版社, 2008.

[2] Napster[EB/OL]. http://en.wikipedia.org/wiki/Napster.

[3] Gnutella[EB/OL]. http://en.wikipedia.org/wiki/Gnutella.

[4] BitTorrent[CP]. http://www.bittorrent.com.

[5] 陈贵海, 李振华著. 对等网络：结构、应用与设计 [M]. 北京: 清华大学出版社, 2007.

[6] 王学松编. Lucene+Nutch 搜索引擎开发 [M]. 北京：人民邮电出版社, 2008.

[7] We Knew Web Was Big[EB/OL]. Official Google Blog, July, 2008. http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html.

[8] 方启明, 杨广文, 武永卫, 郑纬民. 基于 P2P 的 Web 搜索技术 [J]. 软件学报, 2008, 19(10):2706–2719.

[9] 凌波. 基于对等计算的信息检索技术 [D]. 复旦大学, 2004.

[10] T. Suel, C. Mathur, Jo-Wen Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long and K. Shanmugasundaram. ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval[R]. Technical Report, TR-CIS-2003-01, Polytechnic University, 2003.

[11] Yuan Wang, Leonidas Galanis and David J. DeWitt. GALANX: An Efficient Peer-to-Peer Search Engine System[OL]. http://www.cs.wisc.edu/ yuanwang/papers.

[12] M. Bender, S. Michel, P. Triantafillou, G. Weikum and C. Zimmer. MINERVA: Collaborative P2P Search[C]//Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005.

[13] Jin Zhou, Kai Li and Li Tang. Coopeer: A Peer-to-Peer Web Search Engine Towards Collaboration, Humanization and Personalization[C]//IPCCC04, 2004, 313–314.

[14] F. Cuenca-Acuna, C. Peery, R. Martin, and T. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities[C]//IEEE HPDC-12, 2003.

[15] Pandango[CP]. http://www.pandango.com/.

[16] 徐婕. 基于对等网络的资源搜索策略的研究 [D]. 华中科技大学, 2007.

[17] Faroo[CP]. http://www.faroo.com/.

[18] GPU Distributed Search Engine[CP]. http://gpu.sourceforge.net/search_engine.php.

[19] GRUB[CP]. http://grub.org/.

[20] YaCy Distributed Web Search[CP]. http://yacy.net/.

[21] F. Dabek, B. Y.Zhao, P. Druschel, J. D. Kubiatowicz, and I. Stoica. Towards A Common API for Structured Peer-to-Peer Overlays[C]//IPTPS, 2003, 33–44.

[22] Bram Cohen. Incentives Build Robustness in BitTorrent[C]//Workshop on Economics of Peer-to-Peer Systems, Berkeley, June 2003.

[23] Stefan Saroiu, P. Krishna Gummadi, and Steven D.Gribble. Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts[J]. **Multimedia Systems**, 2003, **9**(2):170–184.

[24] FastTrack[EB/OL]. http://en.wikipedia.org/wiki/FastTrack.

[25] KaZaA[CP]. http://www.kazaa.com/.

[26] D. J. Watts and S. H. Strogatz. Collective Dynamics of 'Small-world' Networks[J]. **Nature**, 1998, **393**(6684):440–442.

[27] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of The Internet Topology[C]//SIGCOMM, 1999, 251–262.

[28] A. -L. Barabási and R. Albert. Emergence of Scaling in Random Networks[J]. **Science**, October, 1999, **286**:509–512.

[29] I. Stocia, R. Morris,D. Karger,M. F. Kaashoek,and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications[C]//SIGCOMM, 2001, 149–160.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network[C]//SIGCOMM, 2001, 161–172.

[31] A. Rowstron and P. Druschel. Pastry, A Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems[C]//IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01), LNCS 2218, Springer-Verlag, 2001, 329–350.

[32] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-resilient Wide-area Location and Routing. Tech.Rep.UCB/CSD-01-1141[R], Computer Science Division, U. C. Berkeley, April, 2001.

[33] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment[J]. **IEEE JSAC**, January, 2004, **22**(1):41–53.

[34] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric[C]//IPTPS, 2002, 53–65.

[35] D. Malkhi, M. Naor and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly[C]//ACM PODC, 2002, 183–192.

[36] M. F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table[C]//IPTPS, 2003, 640–651.

[37] Haiying Shen, Cheng-Zhong Xu and Guihai Chen. Cycloid: A Constant-Degree and Lookup-Efficient P2P Overlay Network[C]//IPDPS04, New Mexico, USA, 2004.

[38] M. S. Allen and R. Alebouyeh. Chimera: A Library for Structured Peer-to-Peer Application Development[R/OL]. http://current.cs.ucsb.edu/projects/chimera.

[39] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in A Distributed Environment[C]//ACM SPAA, 1997, 314–326.

[40] Deke Guo, Jie Wu, Honghui Chen, and Xueshan Luo. Moore: An Extendable Peer-to-Peer Network Based on Incomplete Kautz Digraph With Constant Degree[C]//INFOCOM, 2007, 821–829.

[41] G. Salton A. Wong, and C. S. Yang. A Vector Space Model for Automatic Indexing[J]. **CACM**, November, 1975, **18**(11):613–620.

[42] Michael W. Berry, Zlatko Drmac, and Elizabath R. Jessup. Matrices, Vector Spaces, and Information Retrieval[J]. **SIAM Review**, June, 1999, **41**(2):335–362.

[43] P. Reynolds and A. Vahdat. Efficient Peer-to-Peer Keyword Searching[C] //Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware, June, 2003.

[44] Sergey Brin and Lawrence Page. The Anatomy of A Large-scale Hypertextual Web Search Engine[C]// In 7th International World Wide Web Conference, 1998.

[45] Semantics Web[EB/OL]. http://en.wikipedia.org/wiki/Semantic_Web.

[46] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors[J]. **CACM**, 1970, **13**(7):422–426.

[47] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey[J]. **Internet Mathematics**, 2004, **1**(4):485–509.

[48] 谢鲲, 张大方, 文吉刚, 谢高岗, 尤志强. 布鲁姆过滤器代数运算探讨 [J]. 电子学报, 2008, 36(5):869–874.

[49] Sogou Lab[OL]. http://www.sogou.com/labs/resources.html.

[50] Simple Chinese Words Segmentation[CP]. http://www.ftphp.com/scws.

[51] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. Introduction to Algorithms(Second Edition)[M]. Boston: MIT Press, 2001. 潘金贵, 顾铁成等译. 算法导论 [M]. 北京: 机械工业出版社, 2006.

[52] Skype Dialtone: 22 Million Online[J/OL]. **Skype Journal**, January, 2010. http://skypejournal.com/2010/01/skype-dialtone-22-million-online.html.

[53] Skype[CP]. http://www.skype.com/.

[54] A Comparison of Dedicated Servers By Company[EB/OL]. INTAC, April, 2010. http://www.intac.net/a-comparison-of-dedicated-servers-by-company_2010-04-13/

[55] J. W. Han and M. Kamber. Data Mining: Concepts and Techniques(Second Edison)[M]. Elsevier, 2006. 范明, 孟小峰译. 数据挖掘: 概念与技术 [M]. 北京: 机械工业出版社, 2007.

[56] SWIG[CP]. http://www.swig.org/.

[57] PyBloom[CP]. http://github.com/jaybaird/python-bloomfilter.