申请上海交通大学学士学位论文

学术搜索引擎大规模查询系统的建立与优化

论文作者 彭乾旸

学　　号 5130309751

导　　师 甘小莺副教授

专　　业 电子信息科学类

答辩日期 2017 年 6 月 15 日

Submitted in total fulfillment of the requirements for the degree of Bachelor
in IEEE Honor Class

The Implementation and Optimization of Large-Scaled Academic Searching Platform

Qianyang Peng

Advisor
Xiaoying Gan

Depart of Computer Science and Engineering, School of Electronics, Information and Electrical Engineering
Shanghai Jiao Tong University
Shanghai, P.R.China

Jun. 15th, 2017

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：_____

日　期：_____年_____月_____日

上海交通大学
学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于

保　密 □，在 _____ 年解密后适用本授权书。

不保密 □。

(请在以上方框内打 ✓)

学位论文作者签名：_____　　指导教师签名：_____

日　期：_____年 _____月 _____日　　日　期：_____年 _____月 _____日

学术搜索引擎大规模查询系统的建立与优化

摘　要

本毕业论文的题目为学术搜索引擎大规模查询系统的建立与优化。学术搜索引擎是一种针对学术资源进行搜索的搜索引擎，它可以通过论文标题，论文作者，会议期刊等关键词对论文或其它学术资源进行检索。针对这一课题，本论文利用已有的拥有超过1.2 亿篇论文的学术数据库，从两个方面研究了课题的相关问题。在一方面，本论文讨论了如何为学术搜索网站建立一个高效稳定的大规模查询系统。首先，论文从系统架构与索引结构上出发，对查询系统完成了初步的设计与实现，并通过索引文件映射到内存的方法，实现了未预热系统上查询速度的飞跃式的提高；在此基础上，论文通过提出文件导入法这一全新的索引建立手段，成功地大大加速了文档的索引速度，解决了之前困扰已久的索引建立速度严重不足的问题；最后，论文讨论了如何对查询系统进行功能扩展与分布式部署，不仅为学术搜索网站添加了关键词高亮与结果统计的功能，而且用两台计算机完成了分布式云搜索服务的搭建。在另一方面，论文基于查询系统研究了根据搜索条件动态生成知识层级图的相关算法，从实现角度进行了知识层级图的网页端可视化，并从理论角度给出了知识层级图的规模控制算法，为查询系统添加了一个新颖而实用的扩展功能。

关键词： 搜索引擎　数据导入　分布式系统　知识图谱

The Implementation and Optimization of Large-Scaled Academic Searching Platform

ABSTRACT

The title of this thesis is The Implementation and Optimization of Large-Scaled Academic Searching Platform. An academic search engine is a kind of search engine that aims at indexing and searching academic resources, and it can search papers and other relevant academic resources using their titles, authors, venues and keywords. Here I studied this thesis using a academic paper database with over 120 million papers from two aspects. On the one hand, this paper discussed how to build a high-performance and robust large-scaled searching platform. Firstly, we described how to implement a basic searching platform from the prospective of system architecture and index structure and projected indexed file to memory thus achieving a huge leap in the query performance. Secondly, the paper put forward an innovative data import method – file import method for academic papers, which had greated accelerated the index building speed and solved a long-standing problem. Lastly, the paper discussed about the function expansion and how to make a distributed deployment to the platform. On the other hand, this paper studied the relevant algorithm of generating dynamic knowledge graph based on searching platform and searching keywords, and implemented a user interface of the knowledge graph, which acted as a new function for our searching platform.

KEY WORDS: search engine, data import, distributed system, knowledge graph

目 录

第一章　绪论

1.1　课题研究的目的和意义

本毕设的课题为学术搜索引擎大规模查询系统的建立与优化，此课题的设立，是为了给学术网站 Acemap 部署一个功能完善的搜索平台。准确的说，是为该学术网站部署一个 2.0 版本的搜索平台。在本课题开始之前，Acemap 学术网站中有一个简单的搜索平台，虽然各项功能可以正常使用，但是其索引的论文只有 100 万篇，仅占数据库论文总数的百分之一，且结果的展示效果也较简单粗糙，扩展功能几乎没有。究其原因，一是因为作为数据源的数据库结构复杂，建立索引时大量表连接操作的 I/O 密集型操作过多，导致合理时间内导入论文的数量上限很低；二是由于 1.0 版的搜索平台完全使用开源工具的示例代码构建，没有经过完善的设计，因此表现出功能缺乏的特征。

为了解决上述问题，部署一个 2.0 版本的搜索平台的需求便呼之欲出，这也是本课题设立的原因。一方面，论文需要找出一种算法，将文档的导入速度至少提升两个数量级，以完成全部超过一亿篇论文的索引建立。如果完整的索引得以建立完成，由于我们对如此大规模的查询系统在服务器上运行的经验几乎为零，因此届时系统中可能会出现各种各样的问题，也需要我们去进一步解决。另一方面，论文需要抛弃使用示例代码构建的查询系统，而需要从头开始配置与部署一个符合需求的搜索平台，并实现搜索平台的分布式部署。

为了增加课题理论方面的意义，也为了更紧密地结合我的科研小组的研究重心，我在本次毕设中还计划开发一个基于查询系统的知识图功能。本功能的目标是在用户输入关键词进行查询的时候，能自动基于查询结果的关键词等信息生成一张知识脉络图。例如对于 energy saving 这一输入，系统能在环境科学、经济学、电气、化学、数学等领域梳理出相关研究的层级状知识体系图，以帮助用户更好地确定自己的研究方向。

1.2　课题的理论基础

与查询系统的 1.0 版本一样，我们使用开源搜索服务器工具 Solr[1] 作为系统搭建的平台。Solr 基于著名的全文检索引擎架构 Lucene[2] 开发，其本质是使用倒排索引技术实现的全文检索工具。倒排索引的“倒排”二字，即为将原数据的文档 ID 对应关键词的对应关系倒转为关键词对应文档 ID 列表的过程。经过此步处理后，根据关键词或关键词组合查找文档的过程即可以用传统的索引方式完成。一般要增加搜索引擎的可用性，我们需要用一些自然语言处理的技术对文档和查询语句进行处理，例如去除停用词 (filter stopwords) 和词干提取 (stemming)[3] 等。

在分布式系统中，我们使用开源工具 Zookeeper[4] 作为分布式程序协调服务平台。它在提供了集中的配置文件托管平台的同时，也解决了分布式应用中的死锁和数据同步的问题。死锁问题可能使多个节点同时提交请求修改同一内容时，导致所有的请求都不能成功提交。Zookeeper 采用选举领导（Leader) 的方式解决了这一问题，在 Zookeeper 管理的节点集合中，只有领导节点可以进行请求的提交操作。Zookeeper 的内部采用四种算法进行领导节点的选举，但是一般采用依赖逻辑时钟的 FastLeaderElection 的算法。算法的思想基于 Leslie Lamport 的论文 [5]，采用竞选请求加入队列，节

点投票式选举的方法选出领导节点。

在知识图部分，很多论文对我的想法的构思起到了很大的启发作用，也有论文针对关键词的节点聚合与主题提取算法为我的算法设计提供了思路。针对这一部分，在论文的第六章中会有详细的说明。

1.3 论文结构

本课题的工作主要分为两个部分，一是查询系统主体的构建，二是知识图功能的相关研究。查询系统主体部分的工作可以列举如下：

1. 系统架构与索引结构的设计
2. 大规模数据的高效导入
3. 搜索后台与前端的接口设计
4. 查询系统的分布式部署

本论文在第二到五章依次讨论了这些问题。事实上，本文也是一个偏设计类的论文。在研究的过程中，本人阅读了大量的官方与非官方的著作与相关文档，但由于学术搜索这一应用较为特殊，数据库中总条目数很多且数据结构复杂，这导致参考文档只起到了极为有限的参考作用。在大部分的成果中，本人的工作内容都是自己设计，实现并测试优化的。因此，这项建立学术搜索平台的工作，不仅在设计上具有原创性和完整性，也针对遇到的棘手问题有独到的解决方案，这也坚定了我将自己的工作整理写作成论文的信念。第二部分中，论文讨论了如何在搜索结果的基础上建立层级化知识图，这一部分的工作在第六章中有详细的说明。

第二章　学术搜索引擎查询系统架构与索引结构的设计

2.1　工作平台

查询系统采用的软硬件工作平台如下：

2.1.1　硬件部分

本论文讨论的查询系统的构建方式分为两个阶段。论文的第 2-4 章中着重讨论了在单台服务器上独立部署服务器的相关内容。在第五章中讨论了包含两台服务器的搜索平台分布式集群的建立方法。

在单台服务器版本中使用的服务器硬件配置如下：

- CPU: 64 位 Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz * 32
- 内存大小：约 128GB
- 硬盘空间：约 800TB

在分布式服务器中，除了用到了上述服务器之外，还使用了一台备用服务器用以建立集群。该服务器的硬件配置如下：

- CPU: 64 位 Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz * 40
- 内存大小：约 128GB
- 硬盘空间：约 4TB

论文中所有功能的实现与验证都基于该硬件配置。由于查询系统索引规模较大，在单台服务器中索引总大小约为 60G，分布式服务器中总大小约为 120G，因此将该系统移植到配置较低的计算机时可能会出现问题。

2.1.2　软件部分

服务器的操作系统为 Ubuntu 14.04.1，Linux 内核版本号为 3.19.0-25，本论文实验部分对操作系统版本无严格要求。同时，论文采用了 Solr-6.5.0 作为搜索服务器的实现方案。Solr 是一个基于 Lucene 编写的开源搜索平台。由于查询系统建立与优化的重点并不在从倒排索引开始的底层实现上，而该平台的特性很适合我们对问题的研究，因此论文基于该平台进行系统的实现与部署。

在分布式集群中，查询系统采用了 Zookeeper 作为多个 Solr 内核的管理平台，该平台可以自动管理内核的协调工作，包括配置文件统一管理，任务队列，计算资源分配，主服务器选举，宕机处理等。

2.2　系统架构

查询系统分为两个部分，一部分在网站后台 PHP 代码中实现，主要控制用户输入的处理与分析，并将分析后的结果传给搜索服务器后台；一部分在 Solr 平台上实现，主要负责文档的处理，索引的

建立与请求的标记解析。这一阶段我们暂时只考虑单台服务器架构下的系统架构。单台服务器上的系统架构见表（2–1）。

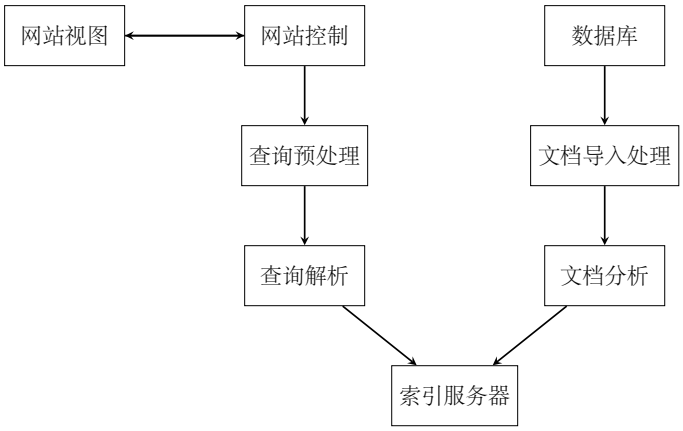

图 2–1 单服务器系统架构

Fig 2–1 System Architecture (Singled)

该架构各部分的作用如下：

- 网站视图：负责网页的显示效果与用户交互。显示效果包括搜索结果展示、统计结果展示、关键词高亮效果等；用户交互包括查询输入框、翻页功能等。
- 网站控制：负责用户输入的处理与网站实际功能的实现。包括查询预处理，网站与搜索服务器的通信机制与结果处理等。
- 查询预处理：该模块为网站控制的一部分。负责对用户输入的预处理，过滤特殊符号并使之变为 html 格式的编码。
- 查询解析：该模块为查询平台的一部分。其目前只对英文内容进行处理，包括过滤停用词，大小写统一，过滤敏感词，单词词根化等。
- 数据库：索引内容的数据源，包含了全部超过 1.2 亿条论文的信息。所有信息被保存在 7 张表中，需要通过数据库的连接得到一篇论文的全部信息。
- 文档导入处理：该模块也是查询平台的一部分。其决定要如何从数据库中取出信息和需要取出数据库的哪些信息。
- 文档分析：该模块和查询解析模块的功能类似。负责对文档部分字段（例如标题字段）的处理，同样地，包括过滤停用词，大小写统一，过滤敏感词，单词词根化等步骤。
- 索引服务器：数据库中的论文数据最终以索引文件的形式保存在索引服务器中。同时，用户的查询请求也在索引服务器中被处理。同时，索引服务器还承担了关键词高亮，结果统计等扩展功能的工作。

2.3 平台配置

在设计完系统的整体架构后，需要对平台进行合适的配置，才能进行之后的索引结构设计与文档导入的操作。在 solr 平台中，主要需要配置的是 solrconfig.xml 文件，它管理了平台的大部分工作配

置,包括启动时加载的模块,针对不同的请求的对应逻辑与系统备份、系统维护的操作等等。在一个刚刚下载好的 solr 服务器中,该配置文件里已经有了一些最基本的配置,我们需要通过添加一些配置以实现我们需要的功能。我们选择和本课题目标最为接近的安装包自带的 data_driven_schema_configs 文件夹下的 solrconfig.xml 文件的基础上进行修改。参考 Solr 官方参考书 Solr in action 的内容 [6],需要进行修改的地方如下:

1. 由于文档导入涉及到数据库操作,因此要在软件启动时声明加载链接 mysql 数据库相关的.jar 包与数据导入处理相关的.jar 包。在 solrconfig.xml 中添加以下内容:

代码 2.1 solrconfig.xml 改动 1

```
<lib dir=${solr.install.dir:../../../..}/dist/ regex=mysql-connector-java-.*jar />
<lib dir=${solr.install.dir:../../../..}/contrib/dataimporthandler/lib/ regex=
.*jar />
<lib dir=${solr.install.dir:../../../..}/dist/ regex=solr-dataimporthandler-.*jar
/>
<lib dir=${solr.install.dir:../../../..}/contrib/dataimporthandler-extras/lib/
regex=.*jar />
```

2. 需要配置数据导入使用的包和默认采用的配置文件。对于以 HTML 参数形式传入的请求,都可以在 solrconfig.xml 中进行处理配置。要实现此处的配置,需要在文件中添加以下内容:

代码 2.2 solrconfig.xml 改动 2

```
<requestHandler name=/dataimport class=
   org.apache.solr.handler.dataimport.DataImportHandler>
<lst name=defaults>
  <str name=config>db-data-config.xml</str>
  <str name=clean>false</str>
</lst>
</requestHandler>
```

此处意为,当 HTML 请求传入的第一个参数为 dataimport 时,使用 solr 中的 DataImportHandler 包对其进行处理。当不指定额外参数的时候,文档导入的配置文件读取文件名为 db-data-config.xml 的文件,且进行后一次导入时默认不清空前一次导入的索引内容。

3. 对于大部分请求的默认域,系统默认配置为查询 _text_ 字段。由于本平台对默认查询字段有特殊的要求,因此用 _entext_ 字段代替 _text_ 字段。此处,entext 意为 exglish text。关于该字段的具体说明在下一个章节中有详细解释。我们需要修改配置文件中以下对应部分的 text 为 entext。

代码 2.3 solrconfig.xml 改动 3

```
<initParams path=/update/**,/query,/select,/tvrh,/elevate,/spell,/browse>
<lst name=defaults>
  <str name=df>\_text\_</str>
</lst>
</initParams>
```

4. 在搜索引擎的扩展功能，即代码高亮功能与结果统计功能中，也需要对此处的配置文件进行相应的修改。在配置文件中，我们需要修改配置文件中处理 select 请求的部分，打开默认的高亮与统计功能的接口开关。

代码 2.4 solrconfig.xml 改动 4

```
<requestHandler name=/select class=solr.SearchHandler>
<lst name=defaults>
  <str name=echoParams>explicit</str>
  <int name=rows>10</int>
  <bool name=hl>true</bool>
  <str name=hl.fl>OriginalVenueName OriginalPaperTitle</str>
  <str name=hl.simple.pre>&lt;font color=#196600&gt;</str>
  <str name=hl.simple.post>&lt;/font&gt;</str>
  <bool name=facet>true</bool>
  <str name=facet.field>PaperPublishYear</str>
  <str name=facet.field>KeywordID</str>
  <str name=facet.field>AuthorID</str>
  <str name=facet.field>ConferenceSeriesID</str>
  <str name=facet.field>JournalID</str>
</lst>
</requestHandler>
```

其中 hl 为高亮功能，hl.simple.pre 为希望设计的被高亮的字段的前缀，hl.simple.post 为高亮字段后缀。由于最后高亮效果要在 html 页面中呈现，因此此处用了 html 格式的更改样式代码。facet 为统计功能，facet.field 声明了需要被统计的域有哪些。

2.4 索引结构

本段内容描述了本查询系统的索引结构。索引的结构包括索引中需要包含哪些字段，字段是否需要被索引 (index)，字段是否需要被保存 (store) 等。被索引的字段表示我们可以通过请求查询该字段的内容，被保存的字段表示返回的结果中我们可以看到这部分的完整内容。考虑到既不需要索引也不需要保存的字段完全不需要被索引，而在本学术搜索平台的实际应用中，并不存在只需要索引且不需要保存的字段，因此，该索引中所有的字段都是需要被保存的。

本项目中需要索引的字段共有 13 个。其分别为：作者 ID、作者姓名、论文作者顺序、会议 ID、会议简称、研究领域、期刊 ID、关键词 ID、关键词、论文标题、论文发表位置、论文出版年份和论文被引用数。其中，对于每一篇文章，作者 ID 与姓名、作者顺序、关键词 ID 与名称这 5 个字段是多值的（以列表表示），其余所有字段均是单值的；期刊 ID 和会议 ID 对于每一篇文章至多只有一项不为空；论文出版年份以数的形式表示（这代表该字段可以在搜索条件中以区间作限制），其余所有字段均以字符串的形式做表示。

这 13 个字段都是被保存 (STORE) 的字段，它们虽然有一些并不会在网页中被直接显示，但是有可能会在统计功能的图表绘图中被使用（例如会议 ID，期刊 ID，关键词 ID 等）。同时，这 13 个字段中只有 5 个字段是被索引的，这五个字段分别是论文标题，作者姓名，发表处名称，出版年份和发表会议简称。也就是说，用户只能通过这 5 个字段查询想要的内容，通过关键词搜索论文，是

不能搜索到关键词位于这 5 个字段之外的字段的内容的。在这 5 个被索引字段中，作者姓名、出版年份和发表会议不会被文档分析模块处理，单词词根化等操作不会影响到这些字段的内容。其余字段会被文档分析模块处理后再被建立到索引中。

同时，这 5 个字段的内容在索引中都被引用到了同一个字段（_entext_ 字段）上。平台设立了 _extext_ 的字段用以表示这五个字段的所有内容集合。在默认查询中，这五个字段都会被等价地共同被系统作为默认字段检索，系统在用户不指定的时候，会返回在 _entext_ 字段的查询结果。如果用户指定了只在某个字段进行查询，例如，只在论文标题中查询关键字的时候，系统会返回论文标题字段的搜索结果。

2.5 索引文件映射到内存

索引文件映射到内存是寻求查询速度优化的一种最终手段。在 Linux 系统中，内存映射文件技术可以把文件夹挂载在内存上, 从而通过访问内存来读写文件。这种技术能带来文件读写速度相比机械硬盘文件的飞跃式的提升。在我们的问题中，将索引文件映射到内存，并将高访问密度的索引访问工作直接在内存上完成，是内存映射文件技术的一个典型应用。

在 Solr 应用中，索引文件的存放文件夹目录为./collectionname/data/index，我们可以先将文件夹中的所有内容备份，再用 Linux 的 tmpfs 或 ramfs 指令将该路径挂载到内存上（此时文件夹会被清空），再将备份的文件放回原目录中。此时，我们再进行查询时，会直接在内存中访问索引文件。附录 B 中，描述了将索引文件映射到内存的详细操作步骤。经过测试，将索引映射到内存后的新系统上的查询速度与已充分预热的原系统相比，查询速度有着很大的提升。

表格2–1用五个典型的查询语句对比了新系统与原系统的查询速度:

表 2–1 索引文件映射到内存前后性能对比

Table 2–1 performance contrast

查询语句	men	wang	emotion	world war	france
返回结果数	158937	1698117	115435	36179	291669
查询时间 (新系统，单位：ms)	18	71	13	35	24
查询时间 (原系统，单位：ms)	143	198	134	185	153

在该表格中可以看出，除了查询姓氏“Wang”这种返回结果超过一百五十万的特殊语句，对于大部分查询语句，在将索引文件映射到内存后的新系统中查询时间均在 50 毫秒以内，而在老系统中的查询时间则大多在 150 毫秒左右徘徊。要注意的是，此时的新系统是一个刚刚启动的系统，而老系统已经经过较长时间的运行，系统中已经有许多缓存的加速文件。对于一个新启动的传统文件查询系统，新系统与之的性能对比将会更加明显。

第三章　大规模数据的索引建立

在系统使用的完整的数据库中，论文的总量超过一亿两千万，因此索引建立的速度是建立方式选取的主要考量因素。由于论文数据库的内容在不断更新，随着时间的推移，数据库中会添加新的论文，现有论文的被引用量等信息也会改变，如果建立一次索引的时间过长，将会大大降低该系统的实时性与实用性。考虑到如果要在 12 小时内建立完整的索引，每秒导入条目的数量必须要大于 2500 条，这对建立索引的方法就有了很高的要求。经过多种方法的尝试后，本论文最终用文件导入法代替了之前的数据库导入法，成功将索引条目导入的速度从最初的每秒 30 条左右增加到了每秒 4000 条左右，成功做到了索引的实时建立与更新。

下面通过介绍并分析新旧两种导入方式，以解释这一问题的解决方式：

3.1　数据库导入法

数据库导入法是向 Solr 导入数据并建立索引的一种较为直观的方式，在 Solr 中的 DataImportHandler 包中有该方式的相关接口。由于数据源是 MySQL 数据库，因此导入过程采用 JAVA 提供的 MySQL 库中的 jdbc.driver 包进行驱动。

正如上一章中提到的，我们只要针对 DataImportHandler 包设计合适的 db-data-config.xml 文件，就可以实现数据库导入索引的操作。经过设计后，该文件的内容如下：

代码 3.1 从数据库中导入索引的 db-data-config.xml

```
<dataConfig>
    <dataSource type=JdbcDataSource
      driver=com.mysql.jdbc.Driver
      url=jdbc:mysql://*.*.*.*:**/database
      user=******
      password=****** />
    <document>
        <entity name=Papers query=select * from Papers>
            <field column=PaperID name=id />
            <field column=OriginalPaperTitle name=OriginalPaperTitle />
            <field column=OriginalVenueName name=OriginalVenueName />
            <field column=PaperRank name=PaperRank />
            <field column=PaperPublishYear name=PaperPublishYear />
            <field column=ConferenceSeriesIDMappedToVenueName name=ConferenceSeriesID />
            <field column=JournalIDMappedToVenueName name=JournalID />
            <entity name=Information
                    query=select * from PaperAuthorAffiliationswhere PaperID='$Papers.PaperID'>
                <field name=AuthorID column=AuthorID />
                <entity name=Authors
                  query=select AuthorName from Authorswhere AuthorID = '$Information.AuthorID'>
                    <field name=AuthorName column=AuthorName />
                </entity>
            </entity>
            <entity name=Conference
```

```
                query=select ShortName from ConferenceSerieswhere ConferenceSeriesID =
        '$Papers.ConferenceSeriesIDMappedToVenueName'>
                <field name=ConferenceShortName column=ShortName />
            </entity>
            <entity name=Keywords
                query=select * from PaperKeywordswhere PaperID='$Papers.PaperID'>
                <field name=KeywordName column=KeywordName />
                <field name=KeywordID column=FieldOfStudyIDMappedToKeyword />
                <entity name=FieldsOfStudy
                    query=select FieldsOfStudyName from FieldsOfStudywhere FieldsOfStudyID =
        '$Keywords.FieldOfStudyIDMappedToKeyword'>
                    <field name=FieldsOfStudyName column=FieldsOfStudyName />
                </entity>
            </entity>
            <entity name=PaperReferencesCount2
                query=select PaperReferenceCount from PaperReferencesCount2where
        PaperReferenceID = '$Papers.PaperID'>
                <field name=PaperReferenceCount column=PaperReferenceCount />
            </entity>
        </entity>
    </document>
</dataConfig>

</requestHandler>
```

采用该方法，可以实现论文的顺利导入，经过实际测试，该方法可以在 12 小时内导入 200 万篇左右的文章。不过不幸的是，因为数据导入速度过慢，对于 1.2 亿条的数据，系统预计的导入时间超过了 30 天，而在服务器上将如此占用资源的进程连续运行 30 天是不可容忍的。由上一章可知，对于每一篇文章，需要索引的域共有 14 个，共涉及到数据库的七张表。由于该方法的索引建立的过程是逐条建立，而不是按表建立，因此需要通过表的连接（JOIN）方式将多张表的内容整合起来，再导入索引中。要整合得到所需的全部 14 个域，其中作者 ID、会议简称、关键词名称、关键词 ID 需要进行一次表的连接操作；作者姓名、研究领域名称需要进行两次表的连接操作。由于大部分数据库表的规模很大，进行表的连接操作将会花费大量的时间。在实际操作中，使用该方法虽然可以完整建立功能完备索引，但是索引的速度只有每秒 30 条左右。考虑到服务器资源的限制与索引建立方式的可操作性，我们不得不抛弃这种方法，而寻找其它的解决方式。

3.2 文件导入法

在数据库导入法暴露了其致命缺陷之后，我尝试了很多改进方式，然而都没有从根本上解决这一问题。究其原因，是数据库表本身连接运算的效率过低导致的。在理论上，计算笛卡尔积的连接操作的复杂度达到了 O(M*N)，经过 SQL 内部优化的 JOIN 操作复杂度也在 O(M+N) 之上。由于这一部分涉及到大量硬盘读写，而硬盘的读写速度远远慢于内存与寄存器内的运算速度，这一典型的 I/O 密集型 (I/O bound) 工作的工作效率会被硬盘的读取速度大大限制。无论如何优化算法，或是给以程序更多的内存，只要读入数据的方式不改变，运行的速度就不会有本质上的提高。

弄清了问题的本质以后，改进该部分的重点就到了如何减少硬盘的读写上。其中一个可行的方式，也是论文中采用的方式，就是完全抛弃调用数据库操作处理数据，而将全部的数据操作都放在

内存中的哈希表中进行，我把这种方法叫做文件导入法。该方法的核心，是先将数据库中的内容处理为 XML 格式的标记语言文件，再用该文件作为数据源建立索引。由于处理得到的文件中每条论文信息可以被逐条导入，因而这种方法完全避免了 I/O 密集的数据库操作，经过测试，使用该方法的导入速度相比数据库导入法提升了 100-200 倍，使大规模索引快速建立这一目标得到了实现。

文件导入法的第一步是将数据库中的数据重构为标记语言文件数据，算法描述如下：

- 链接数据库，选取与搜索平台需要索引的内容相关的表，将表中的内容导出到文件中；
- 依次读入这些文件，用二级字典的形式保存文件内容。其中，字典的第一级为表的主键，字典的第二级为表的其它键，字典中保存的内容为表的键值；
- 遍历保存 Papers 表的字典，用字典的查询操作代替数据库表的链接操作，扩展保存 Papers 表的内容。例如，当查询到论文作者的姓名后，即在 Papers 字典的二级键中加入一个名为 AuthorName 的键，并保存相应的键值；
- 将扩展完整的 Papers 字典保存为 xml 文件，为了后续引用方便，此处文件命名为 transed_all.xml。

该算法先将数据库保存到文件中，再读入文件建立字典，而不是直接从数据库建立字典。这是因为数据库的多行导出比逐行导出的速度要快很多，因此这里不将数据库逐行导出到字典，而是添加了一个中间步骤。

在按此方法从数据库读取并重构了数据之后，我们得到的 transed_all.xml 文件便以逐条呈现的形式存储了所有需要被索引的信息。此时我们以该 xml 文件为数据源进行索引创建的时候，就可以避免耗时的数据库读取与链接操作，而可以通过顺序读文件的方法完成索引的建立。此时，db-data-config.xml 的配置也会变得简单很多。注意，此时要以流形式 (stream = true) 导入数据，可以让系统逐行读取目标文件，避免文件过大导致资源不足。

代码 3.2 从文件导入索引的 db-data-config.xml

```
<dataConfig>
    <dataSource encoding=UTF-8 type=FileDataSource />
    <document>
       <entity
           name=paper
           processor=XPathEntityProcessor
           forEach=/root/paper
           url=/foo/bar/transed_all.xml
           stream=true
           >
       <field column=id xpath=/root/paper/PaperID />
       <field column=OriginalPaperTitle xpath=/root/paper/OriginalPaperTitle />
       <field column=OriginalVenueName xpath=/root/paper/OriginalVenueName />
       <field column=PaperRank xpath=/root/paper/PaperRank />
       <field column=PaperReferenceCount xpath=/root/paper/CitationCount />
       <field column=PaperPublishYear xpath=/root/paper/PaperPublishYear />
       <field column=ConferenceSeriesID xpath=/root/paper/ConferenceSeriesIDMappedToVenueName />
       <field column=JournalID xpath=/root/paper/JournalIDMappedToVenueName />
       <field column=ConferenceShortName xpath=/root/paper/ConferenceShortName />
       <field column=AuthorID xpath=/root/paper/Author/AuthorID />
       <field column=AuthorName xpath=/root/paper/Author/AuthorName />
       <field column=AuthorSequenceOrder xpath=/root/paper/Author/AuthorSequenceOrder />
       <field column=FieldsOfStudyName xpath=/root/paper/Keyword/FieldsOfStudyName />
```

```
        <field column=KeywordID xpath=/root/paper/Keyword/KeywordID />
        <field column=KeywordName xpath=/root/paper/Keyword/KeywordName />
    </entity>
    </document>
</dataConfig>
</requestHandler>
```

该导入配置文件比数据库导入法的配置文件要简单很多。这是因为，此时需要导入的内容都在 xml 树的同一个根节点下平铺展开，所以只需要顺序读入该文件的 xml 树并获取对应域的内容，就可以完成文档的导入工作。

这是一个典型的用空间换时间的算法，其本质为将硬盘中的内容映射到内存中，再利用内存比硬盘高得多的读写速度完成任务。这种空间换时间以提高效率的方法在本平台中也有许多其它的运用场景。如论文 2.5 章中谈到的，在索引的构建过程中，如果将 60G 的索引文件映射到内存中，并直接在内存上执行索引查询请求，会使查询的速度相比于将索引存放于文件中大大提高。但是在服务器中，内存空间是一种比磁盘空间宝贵的多的资源，长时间占用 60G-120G（60G 为单机版索引总大小，120G 为分布式索引总大小）的内存资源也是一项很大的花费。因此，这种方法的使用需要经过慎重考虑。

3.3 两种导入方式的性能对比

我们在单机版服务器上对两种导入方式的速度进行了测试，如图3–1所示，我们可以在数据导入监视器上查看当前的数据导入情况。图中可以看出，当前的索引建立速度为每秒钟 4025 条，我们当前已从数据源处取来了 108,681 条数据，其中的 108,680 条数据已经被成功建立索引。这是一次文件导入法开始约 30 秒后的截图，此时索引的建立速度很快。

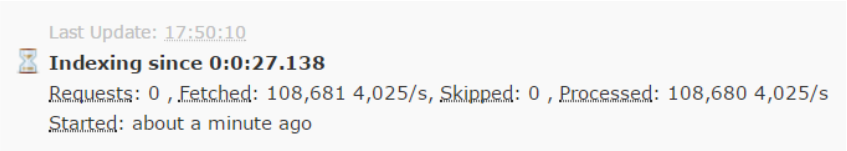

图 3–1 数据导入监视器

Fig 3–1 data import monitor

如图3–2所示，我们分别用数据库导入法和文件导入法建立索引，并在索引开始第 1 小时，第 2 小时，第 8 小时和第 40 小时的时刻记录当前数据导入监视器上的索引建立速度。重复进行三次实验取速度的平均值，可以得到两种导入方式的对比折线图。

从表格中我们可以观察到，当索引导入开始时，文件导入法的导入速度超过每秒 4000 条，数据库导入法的速度约为每秒钟 100 条，此时文件导入法的导入速度是数据库导入法的 40 倍。当索引导入进行到第 8 小时的时候，文件导入法的导入速度仍然稳定在每秒 4000 条左右，而此时数据库导入法的速度已不到文件导入法的百分之一。在第 40 小时的时候，文件导入法早已进行完毕，而数据库导入法的数据导入过程还在以每秒十条左右的速度缓慢进行。我们可以发现，相比于文件导入法，数据库导入法不仅导入速度慢，而且其导入速度随时间衰减的现象也较之文件导入法更为严重。事

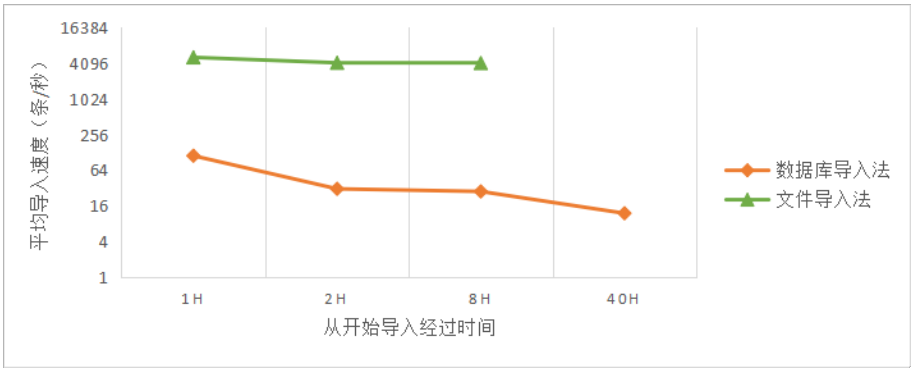

图 3–2 两种导入方式的对比

Fig 3–2 file import and database import

实上，两种导入方式最本质的区别在于数据导入形式的区别，从这个角度，我们分析此现象的原因如下：

- 在数据导入阶段，文件导入法只进行了读取文件流的操作，数据库导入法进行了连接数据库，发送表连接请求，等待服务器应答，接收数据四个阶段。而在等待数据库服务器应答阶段，索引服务器将会花费大量的时间在等待服务器进行表连接操作上，在这一阶段中索引服务器处于空闲状态。因此，数据库导入法的速度相对于文件导入法会慢很多。
- 在数据导入进行过程中，文件导入法每次只处理读入的当前文件流，之前读入到内存的数据在处理完成后可以直接销毁；而在数据库导入法中，数据库服务器的运行速度会随着时间的推移产生越来越多的缓存，运行速度也会越来越慢，此时，索引服务器在等待每次表连接操作时会等待更长的时间。因此，文件导入法的速度随时间变化的波动不大，而数据库导入法的导入速度会随着时间推移严重衰减。

第四章　搜索后台与前端的接口设计

在实现了搜索引擎的后台之后，接下来的步骤就是将其上线到网站上，并实现相应的用户交互接口与界面。这一部分的功能在网站代码的 php 文件中实现。其实现流程图如下：

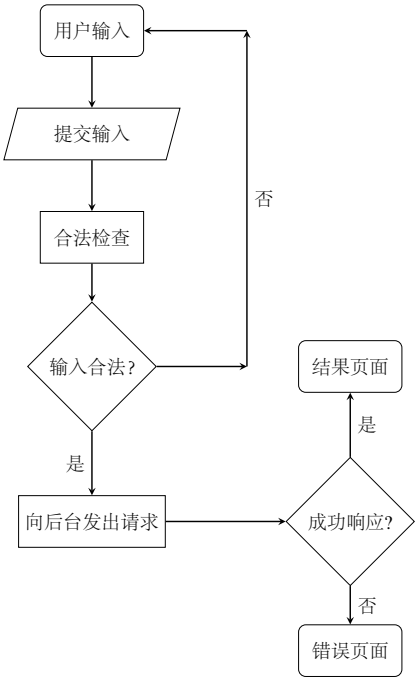

图 4–1 网页处理流程图
Fig 4–1 Website Handler Procedure

当用户输入被确认合法（即去除特殊符号与停用词后不为空字符串）后，用户输入被封装到传入 solr 服务器的 html 请求的 query 字段中，并访问 Solr 服务器以寻求其答复。发送给 Solr 的 HTTP 请求基本格式如下：

http://***.***.***.***:****/solr/collection/select?indent=on&q=content&wt=json&page=2&sort=PaperPublishYear desc&facet=false

事实上，我们还可以对 solr 服务器提出更多的要求，例如利用 edismax 改变打分规则等。由于在本课题中没有得到应用，故略去不讲。在上面那个例子中，HTTP 请求声明的参数如下：

- q: 请求内容的主体，即用户搜索的内容。如果请求内容包含多个单词，用空格隔开的话，多个关键词之间是或的关系，服务器会返回所有含有任何一个关键词的结果。如果在每个关键词前面添加一个'+' 符号，多个关键词之间即是与的关系，服务器只会返回包含有全部关键词的结果。另外，如果不指定搜索的范围，默认会在 _entext_ 域进行搜索（见第二章，平台配置）；如果指定搜索的范围，会在指定的某个域中进行搜索。
- wt: 返回结果的格式。此处指定为 json，也可以根据需要改成 python, xml, php 等。这一部分不

影响返回的内容，但是会影响返回内容的组织格式，在网页对返回结果进行解析的时候，需要选择与此处格式配套的解析策略。

- page: 翻页情况。被索引的论文总数超过一亿篇，这意味着一般搜索一个关键词都会看到超过一万条的返回结果，而返回全部的搜索结果是不现实的。这时候就需要进行翻页。在默认情况下，每页返回十条记录，通过指定该参数，可以让服务器返回结果中的第 10*(page-1)+1 到第 10*page 条结果。
- sort：结果排序情况。默认条件下，我们设置将结果按照论文的引用数排序，如果需要按其他的字段，如按出版年份降序排序，就可以按上例所示修改排序方式。可以同时指定多个排序方式，用逗号隔开，即当前一个指定项相同时，按后一个指定项的顺序进一步排列结果。
- facet：是否统计结果。Solr 服务器提供了统计结果的功能，本搜索平台也在网页上对结果统计进行了相关的界面实现。但是统计结果功能会在一定程度上影响响应速度，所以当对返回结果速度要求较高的时候，可以将此参数设为否，即关闭统计功能。

如果 Solr 服务器正确响应，对于以上请求将会以 Http Response 的形式返回一个.json 格式的文件。该文件中即包含了网页需要显示的所有信息。该文件的结构如下：

代码 4.1 返回文件格式

```
{
  responseHeader:{
    status:0,
    QTime:1442,
    params:{
      q:content,
      indent:on,
      page:2,
      wt:json,
      facet:false}},
  response:{numFound:332899,start:0,docs:[
      {
        PaperReferenceCount:0,
        ConferenceShortName:,
        _entext_:[,
          ,
          contents contents,
          Contents-May 2015,
          2015],
        AuthorID:[7AABEB29],
        OriginalVenueName:[],
        AuthorName:[contents contents],
        ConferenceSeriesID:,
        OriginalPaperTitle:[Contents-May 2015],
        JournalID:,
        AuthorSequenceOrder:[1],
        PaperPublishYear:2015,
        id:5F993E0C,
        _version_:1554052763125547020},
    ......],
     highlighting:{
```

```
5F993E0C:{
  OriginalPaperTitle:[<font color=#196600>Contents</font>-May 2015]},
......
 }}}
```

在该响应文件中，responseheader 里显示了我们查询的参数，response 中显示了响应的内容。其中，response 中的 docs 列表中将会返回在该次查询中返回的十条结果，我们只要解析该 json 文件，并把对应字段的内容加上对应字体的 html 标签在返回页面显示，就实现了搜索结果的展示功能。除了基本内容之外，网页还有结果高亮和结果统计两个扩展功能，这些功能进一步扩展了网页的显示效果。

4.1 结果高亮

结果高亮的目的是用不同的颜色或字体表现用户搜索的关键词部分，使之更加明显。在系统构建的过程中，最初关键词高亮的功能是通过正则表达式匹配替换的方法实现的，即在返回结果中正则匹配出用户输入关键词的部分，然后用两边加入不同颜色的 HTML 标签的方式实现。然而正则表达式匹配法在高亮上存在一个两难抉择的问题:

- 如果采用宽松的匹配方式，即只匹配搜索关键词而不管关键词两边的内容，可能会出现错误匹配的情况。例如，用户搜索单词 get 的时候，和关键词无关但是包含了该单词的单词会被全部高亮，例如 together, forget 等，但是因为本搜索平台是以词为单位进行检索，因而这些结果并不是因为被高亮的部分而被查询到的，这样一来，这种高亮显示就会对用户产生误导，也偏离了关键词高亮这一功能的设计初衷。
- 如果采取严格的匹配方式，即匹配搜索关键词，而且要求关键词两端是字符串边界，空白字符或符号的话，可以避免上述的问题，但是名词或动词变形的情况就无法被匹配到。例如动词结尾 +s, +es, +ed，或以 y 结尾的动词的过去式变形等。这样一来高亮的功能就会受到一些限制，同样会影响到高亮功能的表现。

考虑到以上两个问题用正则匹配法难以同时解决，搜索平台不得不采取另一种方法解决关键词高亮的问题。平台采用的方法为词根匹配并记录位置的方法，由于在搜索引擎中，系统对用户请求和文档内容本身都进行了词根化的操作，而词根也是搜索引擎进行检索的最小单位，因此我们只要记录词根在文档中出现的位置，再于词根化之前的文本中的相同位置进行高亮标记，即可实现同时解决了上述两个问题的高亮算法。在上面的示例文件中可见，虽然用户的查询内容是 content，但是首字母大写的复数形式 Contents 也被正确高亮，同时，由于包含了 content 的其它单词，例如 subcontent，和 content 拥有不同的词根，因此不会被高亮标记。

在 json 返回结果中，我们将 highlighting 域的对应内容，代替 docs 域的对应内容放在网页上进行展示，即可达到关键词被高亮显示的效果。

4.2 结果统计

结果统计功能的目的是统计搜索结果的分布情况。本论文第二章中已经提到过，配置文件中对搜索平台搜索结果的默认统计域有五个——出版年份，关键词，作者，会议名称，期刊名称。其中

对于每篇文章，会议名称和期刊名称不会同时出现。统计功能即是统计搜索结果中出版于各个年份的论文数目是多少，拥有某关键词的论文数目是多少，还有不同作者与搜索关键词相关的论文数量数目等等。结果统计功能可以让用户不需要浏览所有的搜索结果，就可以对搜索结果的特点产生直观的印象。

如果在 html 请求中将 facet=false 改为 facet=true，即可打开结果统计功能。此时，服务器返回的 json 文件中会增加一个 facet 键对应的内容：

代码 4.2 结果统计

```
{
facet_counts:{
    facet_queries:{},
    facet_fields:{
      PaperPublishYear:[
        2013,23649,
        2014,22127,
        2012,20761,
        .....],
      KeywordID:[
        017C8A77,9940,
        08EE83EF,5945,
        09AEBB9C,4648,
        200524E7,4165,
        .....],
      .....},
    facet_ranges:{},
    facet_intervals:{},
    facet_heatmaps:{}}
}
```

如上例所示，由于统计功能的相关参数都在默认配置文件中指定，因此 facet_queries 等域的内容均为空。同时，facet_fields 的内容显示了与该关键词相关的论文的分布情况，例如出版于 2013 年的论文的数量为 23649 篇，2014 年的数量为 22127 篇等。

得到了该统计结果，接下来就是要在网页上对其进行可视化显示。在平台中，我们使用了 ACharts.js[7] 作为绘图工具绘制统计结果图表。考虑到在这五个被统计的域中，对于年份人们关心的是数量随时间变化的信息，而对于其他四个域人们关心的是数量排名信息，因此我们对年份域的结果按年份排序，对其他域的统计结果按统计数目从多到少排序，再分别用折线图，柱状图或饼状图进行分别呈现。

由于结果统计功能复杂度较高，花费时间多，因此开启了统计结果的网页响应速度会比关闭该功能的响应速度慢 500 毫秒左右。为了解决这一问题，平台采用了 Ajax 异步加载的方法，让搜索结果和统计结果异步完成呈现。在用户提交查询请求后，平台会先向搜索服务器发送一次关闭统计功能的请求，先快速显示查询结果，此时统计结果的图表显示为加载状态。查询结果显示完毕后，平台再向服务器发送一次打开结果统计的请求，得到响应后再绘制结果统计的各项图表。

第五章　查询系统的分布式部署

完成了前面几章叙述的内容后，搜索平台经过测试后就已经可以正式上线了。然而在系统实际运行的过程中，却发现系统有一些因为只有单台服务器而产生的问题。例如：

1. 当服务器负荷较大，如正在运行其它高系统占用率服务时，平台的搜索速度会有极大地减缓。
2. 当查询平台的进程因内存溢出等原因崩溃或服务器计算机重启时，网站的搜索结果会直接报错。

这些实际问题的产生也体现了搜索平台对分布式服务器的需求。对于大规模的查询系统，单台服务器往往有它显而易见的局限性，例如负载上限较低，稳定性较差，容错性低等。使用多台服务器部署分布式云架构搜索平台，是解决单台服务器局限性的一种很好的方式。这种架构有如下优点：

1. 实现了多台服务器查询时的负载均衡，使搜索服务器可以动态分配计算资源，达到性能最大化；
2. 实现了多台服务器上配置文件的集中管理，将多台服务器的配置文件统一挂在在一个节点上，避免了多份配置文件管理上的不便；
3. 实现了多台服务器运行状态的统一监管，在任意节点上可以查看所有节点的运行状态；
4. 采用了分布式的自组织架构，并没有严格意义上的主服务器，任何一个服务器挂掉，都不会影响到整个系统的正常运行；
5. 实现了服务进程的自动恢复，当任何一台电脑需要重启或关闭服务进程时，直接关闭该进程不会中断搜索服务的提供，并且在服务进程重新启动后，进程能自动恢复到云上；
6. 实现了索引文件的备份。对于索引文件的每一个分片，其备份数目都与云平台上节点数目相同，这在保证了系统容忍任意 (N-1) 个节点同时故障的同时，也对索引文件实现了多次备份，避免了某几个服务器硬盘损坏导致的数据不可恢复地丢失。

当然，要实现拥有这么多功能的云架构搜索平台，就不可避免地需要大大增加整个系统的复杂度。虽然目前互联网上有一些相关的资料，但是大部分资料都是让分布式服务运行不报错就浅尝辄止，而几乎没有资料讲述了完整地实现一个功能完整的分布式搜索服务器的方法。因此在做这一部分工作的时间里，我付出了大量的时间去摸索实现分布式服务的各个环节，也花费了比预期更长的时间才完成了分布式搜索平台的所有功能。

实现分布式搜索平台的第一步就是要设计系统的架构。架构设计的目的是要在尽量不改动之前实现的功能模块的基础上为系统引入分布式的服务，同时要使分布式的系统尽可能地稳定，功能强大，并且易于维护。

5.1　分布式系统的架构

要想实现搜索平台的分布式部署，就必须引入分布式程序的协调工具。在本平台中，我们使用 Apache Zookeeper 来进行分布式服务的协调。Zookeeper 在本应用中，显式地可以配置文件云节点的挂载，以实现配置文件的统一管理；隐式地可以完成主服务器选举、负载均衡、同步数据、锁死避免等分布式服务所需要的功能。经过初步设计后，分布式平台的架构设计如下：

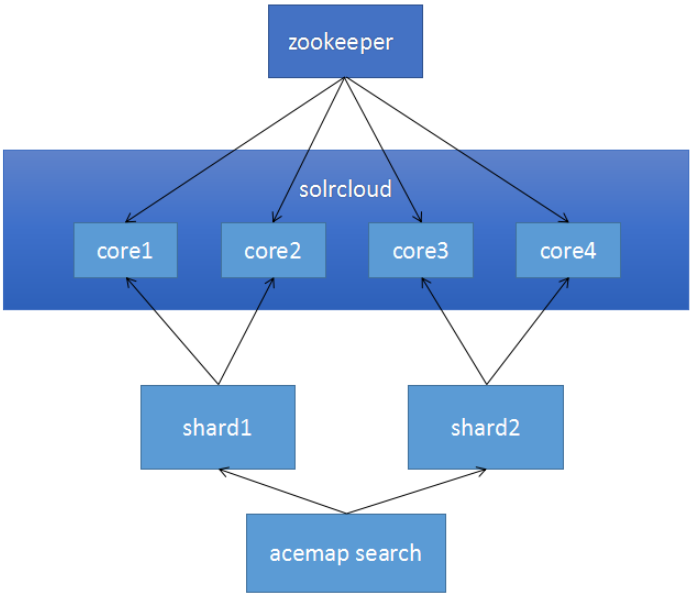

图 5–1 云服务器初始架构
Fig. 5–1 Solrcloud Simplified

在这里，我们把原来单一的 Solr 实例 (Instance，又叫 Core) 分为四个节点，并由一个 Zookeeper 服务器进行统一管理。同时，我们原本的单一索引文件被分成了两个片 (Shard)，每一个片分为两个相同的备份。这样一来，每个 Solr 节点里恰保存一份索引，并负责与该份索引相关的索引工作。我们将 Core1 与 Core3 放在同一台服务器上，Core2 个 Core4 放在另一台服务器上，四台服务器上的两片索引文件和两片备份索引文件共同组成了索引集合 (Collection)。

由于在实例中保存的配置文件难以同步修改，因此在这四个 Solr 实例中均不保存任何配置文件，所有的配置文件我们统一保存在 Zookeeper 的数据节点中。在索引构建的阶段，四个实例分别读取 Zookeeper 数据节点中的配置文件，确定自己的索引结构，并由 Zookeeper 内部算法决定将每一篇文档转发到哪个分片中进行索引；在查询阶段，Zookeeper 决定将查询请求分配给哪些较为空闲的节点，并通过配置文件决定给 Solr 实例的查询请求格式，由选举出的主实例 (Leader) 进行结果整合与发布。

当四个节点中的任意一个节点因为意外（例如索引文件损坏，进程意外终止等）无法正常工作时，Zookeeper 会自动检测到节点的异常，并将其标记为宕机（Down）状态或离线（Gone）状态，此时整个系统的服务并不会终止，而是会由剩下的节点分配承担所有任务。由于目前架构中任意一台服务器上都保存着一份所有索引的备份，所以只要不是所有的计算机服务器全部宕机，完整的服务功能都会保持。当意外关闭的节点被重新注册回 Zookeeper 时，Zookeeper 会尝试恢复该节点并使其回复正常工作。

以上的架构看似很完美，然而仍然存在一个严重的缺陷——那就是 Zookeeper 节点的单一性会导致其反而成为系统鲁棒性的短板。换句话说，虽然对于目前的架构任意 Solr 节点的宕机都不会造成服务故障，但是一旦 Zookeeper 服务本身出现故障，所有的服务都会跟着瘫痪。解决这一问题的方

法有两个，一是将 Zookeeper 服务运行在几乎不需要维护重启的专门服务器上，二是针对 Zookeeper 服务器本身建立一个分布式架构。用通俗的话说，第二种解决问题的方法，就是用 Zookeeper Cloud 来管理 Solr Cloud。该改进的架构如下图所示：此时，Zookeeper 不仅管理着四个 Solr 节点，同时也

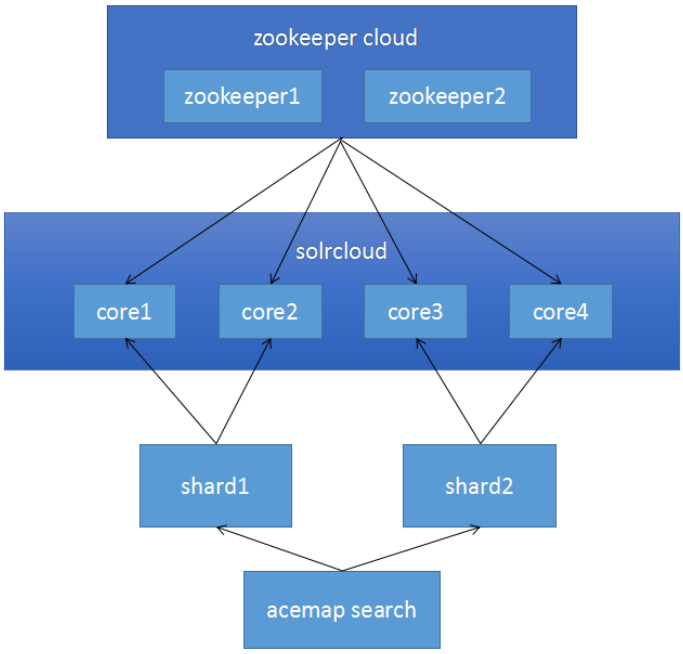

图 5–2 云服务器改进架构

Fig. 5–2 Solrcloud Advanced

管理着两个 Zookeeper 节点。与 Solr 节点自动选取 Leader 类似，Zookeeper 节点的 Leader 也是由内部自动选举得到的。两个 Zookeeper 节点的存在分担了 Zookeeper 服务器宕机的风险，使得其中任何一个 Zookeeper 服务器故障都不会影响到系统整体的正常工作。一个显而易见的好处是，在这个架构下我们可以只配置两台服务器计算机，并在每台计算机上部署一个 Zookeeper 节点和两个 Solr 节点，此时任何一台计算机的重启都不会终止服务的正常提供。而如果不配置两个 Zookeeper 节点，运行着 Zookeeper 服务的计算机是不能重启的，否则整个服务就会崩溃。

但是这个方案和第一套方案相比也不是十全十美的。第一套解决方案能保证系统在上线后，能够用一套十分简单的流程来维护系统的所有功能，而对于第二套解决方案架构的系统，其维护过程会比第一套解决方案复杂很多，在非专业人士操作维护时可能会让服务器产生更多的故障。因此这里只是在理论上说明了第二套流程的可行性及其优势，在实际搜索平台的实现中，我们并没有采用云架构部署 Zookeeper 本身。

5.2 分布式系统的部署

这一部分介绍云服务器初始架构的部署方式。具体的部署命令可以参见附录 A，上面有详细的罗列。

首先，我们需要配置并启动一个 Zookeeper 服务。Zookeeper 服务的配置需要指定 5 个参数，我们通过修改 zoo.cfg 文件以完成此处的配置。

1. tickTime：一次同步的时间周期，此处设置为 2000ms；
2. initLimit：第一次同步过程花费时间周期数的限制，此处设置为 10；
3. syncLimit：一次同步过程的时间周期数限制，即发送请求与得到相应之间的最大周期数间隔，此处设置为 5；
4. dataDir：存放快照数据的目录，根据需要设置；
5. clientPort：服务运行的端口，所有的 Solr 节点都要通过这个端口注册到 Zookeeper。

配置好这些参数后，我们就可以启动 Zookeeper 服务了。启动 Zookeeper 服务后，我们可以打开其客户端（Client）查看内部文件情况，由于是新挂载的节点，所以里面是没有任何文件的。针对之后需要挂载的 Solr 服务，需要在 Zookeeper 上挂载一个空节点，这里起名为 acemap 节点。

接下来要将 Solr 服务器注册到 Zookeeper 上。我们在两台服务器上各启动一个 Solr 服务，然后在每一个 Solr 服务上运行两个实例节点。在这一过程中，首先需要配置 Solr 服务的 host 地址。由于本平台中两台服务器并不在同一个内网上，所以我们使用公网地址作为服务的 Solr 地址。然后，我们需要指定服务以云模式启动并挂在到 Zookeeper 的 acemap 节点上，并指定为其 Java 虚拟机分配的内存。

第一台服务器上的 Solr 服务启动后，Zookeeper 的 acemap 节点上会自动保存一些基本的配置文件，打开其中的 live_nodes 文件夹，可以看到对应与启动服务 host 名称的记录文件。启动第二台服务器上的 Solr 服务后，live_nodes 文件夹里会显示两个服务各自的记录文件。

之后，我们需要在两个 Solr 服务组成的集群上创建一个新的索引集合（Collection），并以 2 的冗余度创建两个分片。此时，这四个分片会被自动分配到四个 Solr 节点中，同时每台服务器上的 Solr 服务都会自动分配到两个不同的分片。

最后，我们需要实现配置文件的统一管理功能。这一功能可以借助 zkcli.sh 进行实现。我们先将所有的配置文件统一放入文件夹中，用 zkcli.sh 的 upconfig 方法可以将本地的文件夹传入 Zookeeper 对应节点上，downconfig 方法可以将 Zookeeper 节点上的文件下载到本地。在文件夹被挂载到节点上之后，我们可以用 linkconfig 函数将节点与 Solr 云中的 collection 对应起来。

至此，Solr 服务器的分布式部署已经基本完成，我们可以通过访问任何一个 Solr 服务器来管理这个分布式的集合（collection）。

5.3 分布式系统的配置

分布式系统的配置方法和单台服务器的配置方法大同小异。在集合整体的视角下，四个子节点组成的整体可以视作一个大的单台服务器，因此，系统配置的过程和单台服务器一样，需要先配置 solrconfig.xml 文件，再进行索引结构设计与文档导入，最后进行查询指令的设计。不同的是，在配置文件与索引结构设计完成后，注意需要用 reload 指令确保指令被同步到了每个节点之中。

此外，由于索引被分成了两个独立的片，因此查询指令中新增了一条 &shard=XXX 的命令，用以只在某个片中搜索结果。更进一步地，可以指定只在某个 shard 的某个特定的备份中进行查询。这种查询方式称作分布式查询。[8]

为了进一步利用 Solr 云架构拥有分布式查询的功能，shard 的构建过程除了交给系统自动进行外，也可以人为地进一步自定义。例如，在集合创建时，可以用 router.name 属性在决定每一条文档被分发到哪一个具体的片上。例如对于本学术网站的搜索平台，router.name 可以由论文属于计算机领域或是非计算机领域决定，通过将计算机领域的论文索引到其中一个片，非计算机领域的论文索引到另一个片，就可以用分布式查询的方法在计算机或非计算机论文这两个子集中分别查询论文。但是由于实际上非计算机领域和计算机领域论文这一划分方式并不均匀，前者的数量实际上会远远多于后者，所以会导致资源分配不均的问题。我们可以通过添加服务器，并利用片分割（Shard Splitting）的方法将较大的片进一步分割存储到更多的服务器中，以达到负载均衡。

5.4 分布式系统的使用

本分布式平台的架构是一种自组织的架构，我们可以通过访问任何一个 Solr 服务器节点来访问并管理整个服务器集群。同样，对搜索平台的查询命令，也可以通过访问任何一个节点完成。假设分布式两台服务器被挂载于 192.168.0.1:8888 和 192.168.0.2:8888 两个节点上，那么执行以下两个操作，都可以等同地在分布式平台中进行查询操作。即：

- http://192.168.0.1:8888/solr/collection/select?indent=on&q=content
- http://192.168.0.2:8888/solr/collection/select?indent=on&q=content

这两个查询命令在分布式系统中基本是等价的。

分布式系统的一大功能是能够在某个节点挂掉时不影响整个系统的正常运行，但是系统的正常运行并不代表上面两个查询命令都可以正常工作。例如，当 192.168.0.1:8888 这台服务器在进行维护时，虽然整个分布式平台依然正常工作，但是 192.168.0.1:8888 这个地址本身是无法访问的。这就要求网站控制部分在其中一个服务器无法访问时，继续去尝试访问另一个服务器。只有两个服务器都无法访问，才能说明目前分布式平台工作不正常，此时再返回错误页面。此时网页处理的流程图如下：

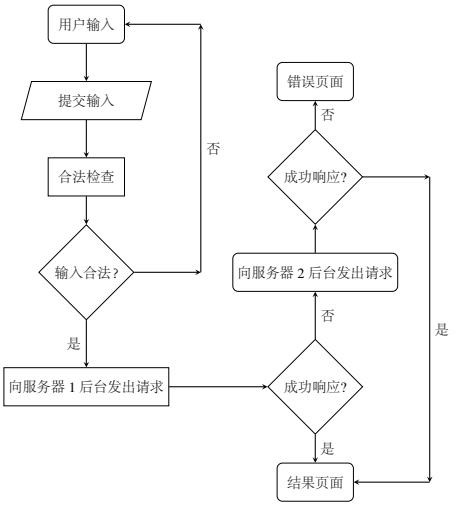

图 5–3 网页处理流程图 2

Fig 5–3 Website Handler Procedure 2

第六章　基于查询系统的知识图设计

6.1　背景知识

知识图谱是将知识整合后以图的形式表现的一种方法，也是一种可视化的知识管理工具。在毕设开题报告的时候，我就一直思考如何为我的偏工程的设计中发掘出一些理论意义，于是便有了这个结合查询系统进行知识图设计的想法。该功能的构想基于如下考量：用户使用学术网站进行查询的时候，可能只重点关注搜索结果的前几条，而隐藏在大量搜索结果之中的信息则无法很好的被用户知晓。论文希望设计出一套根据用户输入的查询关键词动态生成知识图，帮助用户发掘出这些埋藏的信息。例如对于用户可能对“节能”这一关键词感兴趣，于是他键入这一关键词并查询，系统则能在环境科学、经济学、电气、化学、数学等领域梳理出相关研究的层级状知识图，以帮助用户更好地确定与了解自己的研究方向。

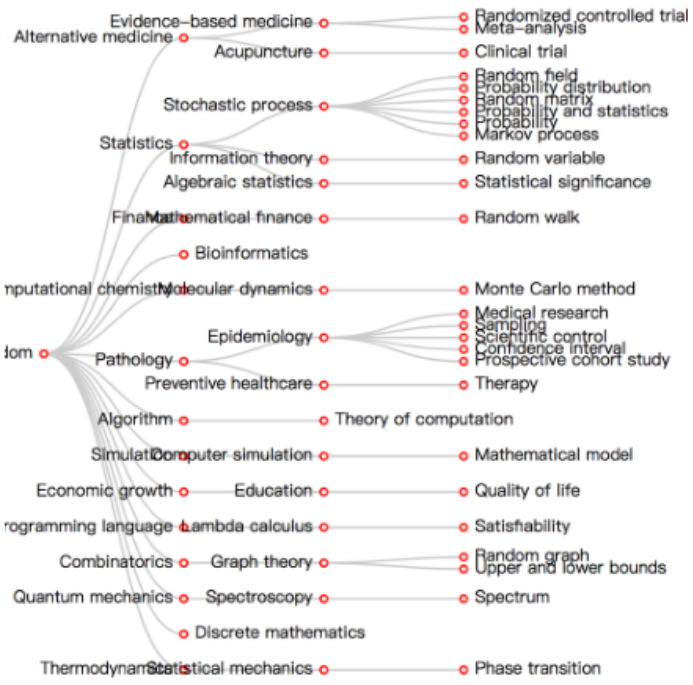

图 6–1 一个简单的层级状知识图

Fig. 6–1 A simple layered knowledge graph

为了更好地说明我的设计目标，我绘制了一个简单的层级状知识图作为示例。为了插图易于阅读，我们隐去了最高层 L0 层级，避免知识图过于复杂。如图所示，对于用户输入的关键词 random，我们可以看到几棵规模较大的子树，例如 Alternative Medicine（替代医学），Statistics（统计学），Pathology（病理学）等；也有一些规模不大但是我个人比较感兴趣的子树，例如 Algorithm（算法）和 Programming Language（编程语言）等；同时，还有 Quantum Mechanics（量子力学）等重要的学科

子树。这些子树又扩展开，给我们展示了这些领域的子领域相关知识。

此图是基于用户搜索 random 这一关键词，根据查询结果动态生成的。显然，该图相比单纯的查询结果列表能反映出更多的信息。例如，“随机”这一关键词在医学领域的应用比在计算机领域要更加地广泛，并且和统计学密切相关。如果一个研究者对“随机”这一关键词感兴趣，他如果作为一名应用科学的从业者，那么可以更多的了解到理论科学方面统计学的相关的知识点，并作为自己的研究工具；如果他是一名理论科学的从业者，也可以通过了解到随机这一关键词在医学与计算机科学方面的应用，从而给自己理论的实际应用带来灵感。这就是根据用户搜索的内容来动态生成知识图的应用。目前的主流学术网站上很少有与该设想相重复的功能，这也是我继续这一研究的一大动力。

在理论方面，有很多的论文为我研究这一问题起到了很大的启发作用。Ronen Feldman 的文章 [9] 介绍了根据实体的关键词分布进行数据挖掘的各种算法，为如何从理论的角度量化研究论文的关键词信息提供了很大的启发；Hsin-Ning Su 的论文 [10] 介绍了从关键词的共现性绘制知识结构的方法，并讨论了该方法在二维知识图绘制中的应用;Yuefeng Li 的论文 [11] 通过分析关键词的相关性和非相关性给出了关键词信息的筛选手段，对我们的图规模控制算法起到了启发作用；Stanley Loh 的论文 [12] 给出了关键词相关性的置信度算法，为我们对关键词之间相关性的计算给出了启发。

有了这些理论基础，我们就可以从最简单的层级状知识图入手设计我们的动态知识图生成功能。

6.2 知识图生成基础算法

与背景知识的示例中一样，我们利用查询结果中的论文关键词信息绘制层级状知识图。在这里，本文章的第 4.2 章中介绍的结果统计功能 (facet) 恰好能为我们所用。结果统计功能恰好统计了论文的关键词信息，其基本结构如下：

代码 6.1 关键词结果统计

```
{
facet_counts:{
    facet_queries:{},
    facet_fields:{
      PaperPublishYear:[.....],
      KeywordID:[
        017C8A77,9940,
        08EE83EF,5945,
        09AEBB9C,4648,
        200524E7,4165,
        .....],
      .....},
    facet_ranges:{},
    facet_intervals:{},
    facet_heatmaps:{}}
}
```

其中，KeywordID 虽然用不直观的十六进制字符串的形式表示，但是在数据库中能很方便的查询到 ID 对应的实际内容，经过转化后如下：

代码 6.2 关键词结果统计 _ 新

```
{
    KeywordID:[
      Water content,9940,
      Chemical composition,5945,
      Content analysis,4648,
      Nitrogen,4165,
      .....]
}
```

在此图中我们便可以很明确地看到关键词结果统计的详细信息，我们的查询语句为“content”，而查询结果论文拥有关键词排名的前几名都与化学成分分析等领域密切相关。上表也正是一张包含了“content”这一单词的论文的关键词出现频率的排行榜。

我们的数据库中还有一张记录了关键词之间层级关系的表，它记录了每个在论文中出现过的关键词, 其所在的层级（L0, L1, L2, L3）与关键词的父关键词，其中 L3 为最细分的层级，而 L0 为最概括的层级。结合这张表的内容，我们即可以实现知识图生成的基础算法。首先，我们说明如何生成一颗基础的树状知识图:

算法 6–1 生成基础的树状知识图

输出: 树状知识图 $knowledgetree$

输入: 用户查询关键词 $userinput$, 结果统计关键词列表 $keylist$, 关键词层级关系表 $hierarchy$

$knowledgetree \leftarrow \{\}$
$root \leftarrow userinput$
$knowledgetree.append(root)$
for $node$ in $keylist$ **do**
    $node.parent \leftarrow hierarchy[node]$
    if $node$ not in $knowledgetree$ and $node.parent == None$ **then**
        $node.parent \leftarrow root$
        $knowledgetree.append(node)$
    end if
    if $node$ not in $knowledgetree$ and $node.parent != None$ **then**
        $nodep \leftarrow node.parent$
        $knowledgetree.append(node)$
        $keylist.append\{nodep\}$
    end if
end for

通过该算法，我们可以得到得到一棵理论上信息完全的知识图树结构，但是其保存的数据结构还略显混乱，因此我们需要进一步将之前得到的数据结构进行格式化。一个好的树的数据结构应该是这样的:

代码 6.3 树数据结构

```
{node:{
```

```
  nodeproperty1 : aaa,
  nodeproperty2 : bbb,
  ...,
  children:{node*}
  }
}
```

在该数据结构中，树用一个字典进行表示，字典中只有一个键为根节点的元素，其值为根节点的各种属性与子节点列表。子节点字典要么为空，要么包含的元素递归地为与根节点相同的字典结构。

上述优结构的树与算法 6-1 得到的树最大的区别在于，优结构树中父节点总是在子节点之前出现的，而算法 6-1 得到的树的父节点会经常地在子节点之后出现。可以运用递归函数，用栈的思想即可将算法 6-1 得到的输出转化为上述优字典结构的树。具体实现此处略去。

得到了优结构的树后，我们即可以开始考虑知识图的绘制问题。

6.3 知识图的绘制

我们使用数据可视化工具 d3.js 来进行知识图的绘制。d3 是数据驱动文档（Data Driven Document）的简称，它以开源 Javascript 代码的形式提供了很多方便的网页数据可视化接口。d3.js 提供了许多绘制层级状树结构的示例接口。例如图 6.1 的系统树图（Dendrogram），下图所示的径向树（Radial tree）与放射图（sunburst）等。

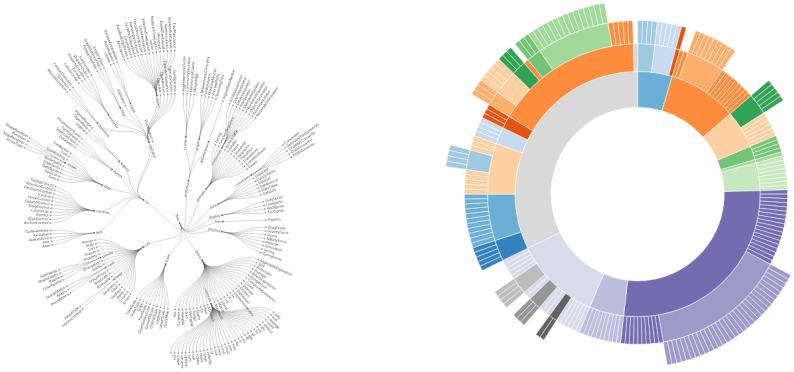

图 6–2 左：径向树；右：放射图

Fig 6–2 radial tree and sunburst

这些工具都可以用来展示层级状知识图，但是以上两幅图相比图6–1的传统树状图，对空间的利用率更高，且支持更多的用户交互动作，因此更适合在网站中使用。运用这些接口绘制我们的我们的层级状知识图，图上的圆心代表用户查询的关键词，而图上的同心圆从内到外依次代表树节点各层次的内容。实际绘制得到的效果如图6–3所示。

图6–3中左图是算法 6-1 得到的树结构化后直接绘制的知识图，右图是将左图两点钟方向的子树放大后的效果。可以看出，绘制的知识图已经有了初步的雏形与信息量，但是也存在着明显的问题。左图包含的节点数显然过多，左上角部分有大量紧密的难以阅读的节点；而右图包含的节点数又太

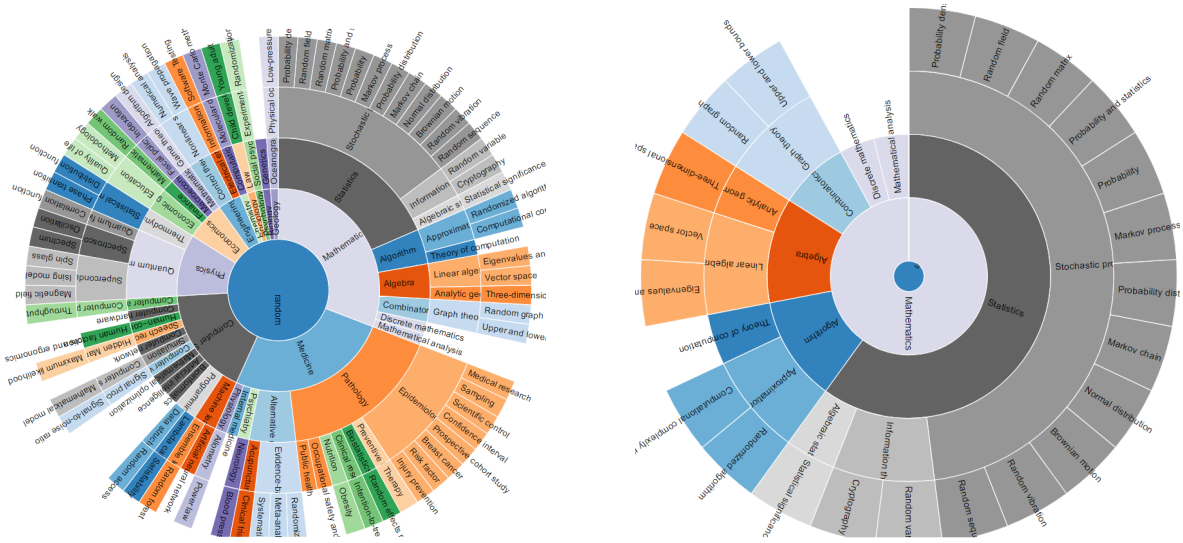

图 6–3 左：放射状知识图；右：知识图子树

Fig 6–3 sunburst and subtree sunburst

少，使得放射图的表现效果不佳。因此，下一步我们需要进行知识图的规模控制，将知识图的节点数控制在一个合适的水平。

6.4 知识图规模的控制

我们用以下算法流程实行知识图规模的控制：

6.4.1 去除信息缺失的节点

我们生成的知识图以树的形式呈现，其中树的根节点是人为加入的参考节点。去除根节点，可以得到一组树 $T_1, T_2, ..., T_n$。这些树中的节点均为论文的关键词节点，节点包含层级信息 L_0, L_1, L_2, L_3，代表关键词节点在论文关键词层级表中处于哪一层。考虑图6–3中的左半部分，其中包含了一些直接连接至根节点的 L_2, L_3 层节点，它们同时也是子树集合 $T_1, T_2, ..., T_n$ 的根节点。我们认为，直接连接至根节点的 L_2, L_3 层节点处于较为细化的层次，其子节点层数至多为一层，且很有可能是因为数据库信息缺失导致的父节点信息丢失。这些信息缺失的节点会影响对整张图层级化结构的认识，因此我们将这些节点联通其连接的树结构删去。详见算法6–2。

6.4.2 计算节点的规模值

在这一部分中，我将描述如何进一步地选取知识图中相对更为重要的节点。此处，我们采用一维 Kmeans 聚类 [13] 的方法选取规模值最大，也就是重要度最高的一系列节点，并将其它节点删去，从而控制整棵树的规模。在此算法构思的形成上，我们受到了文献 [14] 和 [15] 中的对关键词进行聚类的算法的启发，但是并没有照搬文献中的公式，而是针对我们自己的问题用类似的思想构建了一套自己的公式。

算法 6–2 去除信息缺失的节点

输出: 子树集合 $\{T'_1, T'_2, ...T'_m\}$

输入: 原始树集合 $\{T_1, T_2, ...T_n\}$

```
for t in {T_1, T_2, ...T_n} do
    troot ← t.root
    if troot.layer == L2 or troot.layer == L3 then
        remove t from {T_1, T_2, ...T_n}
    end if
end for
```

本算法中用到的符号定义如下:

L_i: 关键词处于的层次;
$T^Q_{L_i,j}$: 以第 L_i 层的第 j 个节点为根节点，Q 为根节点父节点的子树;
$S(T)$: 子树的规模值，此处及以下的 T 均为 $T^Q_{L_i,j}$ 的简写形式;
$N(T)$: 子树的权重值;
$D(T)$: 子树根节点所在的深度值，即 L_i 中的 i 值加上 1;
$F(T)$: 子树根节点在 Q 的所有子节点中的重要度;
$f(T)$: 子树根节点出现频率;
f_{max}: Q 的子节点的最大出现频率;
f_{min}: Q 的子节点的最小出现频率;
m_T: 传递系数

算法的中心思想，即为根据节点的相关信息计算出子树的规模值 $S(T)$，之后对子树的规模进行聚类，保留规模较大的聚类，而删除规模较小的聚类，从而达成了知识图的规模控制效果。其中，子树规模值 $S(T^Q_{L_i,j})$ 的计算公式如下:

$$S(T^Q_{L_i,j}) = \sum_{k=1}^{n} m_{T^{Q'}_{L_{i+1},k}} \cdot S(T^{Q'}_{L_{i+1},k}) + N(T^Q_{L_i,j}) \tag{6–1}$$

其中，k 从 1 到 n 遍历了子树 T 的根节点的所有子节点。该算法是一个递归算法，算法基于如下考量: 对于任意一个子树 T，只要该子树含有多余一个节点，那么删去该子树的根节点，可以得到一个树的集合 $T'_1, T'_2, ..., T'_m$。我们采用递归算法，认为子树 T 的规模值为上述树集合中所有树的规模值乘以传递系数并求和，加上该子树本身的权重值。如果子树 T 本身已经是叶子节点，那么前一项的值为 0，直接用该节点的权重值代替节点的规模值。$m_{T^{Q'}_{L_{i+1},k}}$ 为比例系数，其代表节点 $T^{Q'}_{L_{i+1},k}$ 是否在搜索统计结果里直接出现。如果直接出现则该比例系数为 1，否则该比例系数为 0。

接下来，我们定义节点的权重值 $N(T^Q_{L_i,j})$。权重值的定义方式如下:

$$N(T^Q_{L_i,j}) = (\alpha_1 \cdot f(T^Q_{L_i,j}) + \alpha_2 \cdot \frac{1}{D(T^Q_{L_i,j})}) \cdot F(T^Q_{L_i,j}) \tag{6–2}$$

其中，$f(T^Q_{L_i,j})$ 指树 $T^Q_{L_i,j}$ 根节点的出现频率，该值在查询系统的统计结果中可以得到。$D(T^Q_{L_i,j})$ 是指树 $T^Q_{L_i,j}$ 根节点的深度，也就是 L_i 中的 i 值加上 1。α_1，α_2 为比例系数，我们取 $\alpha_1 = \frac{1}{\max_{i=1}^{N}(f_i)}$，$\alpha_2 = 1$。

$F(T^Q_{L_i,j})$ 子树根节点在 Q 的所有子节点中的重要度，其计算公式为：

$$F(T^Q_{L_i,j}) = \frac{f(T^Q_{L_i,j}) - f_{min}}{f_{max} - f_{min}} \tag{6–3}$$

注意 f_{max} 与 $\max_{i=1}^{N}(f_i)$ 的区别。f_{max} 是拥有同一父节点的所有子节点的频率值的最大值，$\max_{i=1}^{N}(f_i)$ 是图中所有节点的频率值的最大值。

运用以上公式，即可计算出所有节点的规模值。

6.4.3 选取规模值最大的节点聚类

在上一节中，我们阐述了节点规模值的计算公式。节点规模值同时考虑了三个要素，一是节点的子节点的规模值会被传递到父节点，二是节点的深度值和包含论文的数量会影响到节点的重要性，三是节点与兄弟节点包含论文的相对数目会影响到节点的取舍。

在计算了节点规模值后，我们即需要确认哪些节点需要保留，而哪些节点不需要保留。我们的计算方式是按层计算，即当内层的计算结束，节点过滤工作完成后，我们才会开始计算外层的节点取舍。在节点取舍上，我们采用了 kmeans 聚类的算法。我们将同层的节点按规模值聚类为重要和不重要两类，并保留分类为重要的所有节点，删除分类为不重要的所有节点及其所有的子节点。

选用 kmeans 算法而不选用贪婪法的原因在于节点选择的公平性。例如当某层节点的规模值依次为 3,3,5,5,5 时，我们倾向于保留三个规模值为 5 的节点，而当节点的规模值一次为 3,3,3,5,5 时，我们则倾向于只保留两个规模值较大的节点。而这种动态的节点保留思想，用 kmeans 算法可以得到很好的体现。该算法的描述如下：

1. 取 a，b 初始值 a=0，b=1；
2. 计算所有节点的规模值到 a，b 的距离，即与 a，b 分别做差的绝对值。与 a 较近的节点标记为 A，与 b 较近的标记为 B；
3. 将标记为 A 的所有节点规模值的平均值作为新的 a，标记为 B 的所有节点规模值的平均值作为新的 b，重复第 2，3 步操作，直到集合 A，B 内的元素不改变为止；
4. 返回集合 A，B。

最后，我们将集合 B 中的节点作为保留节点，集合 A 中的节点作为被过滤的节点，即达成算法目标。

6.5 知识图的显示效果

我们运用以上算法保留重要度相对较高的节点，并删除了一些重要性相对较低的节点，目的就是把最终知识图的规模控制到合理的范围内。因为是动态生成层级状知识图的算法，所以我们没有采用高复杂度的算法，而是更加重视算法在系统上的响应速度，以保证该功能可以顺利上线。经过测试，本方法在知识图的规模控制上有着很好的效果。为了进一步丰富展示的内容，我们用放射图

圆弧的弧度直观显示对应关键词的对应论文数目，并将论文数量随年份的演化关系在图右侧的折线图上显示。最终的显示效果如图6–4所示。

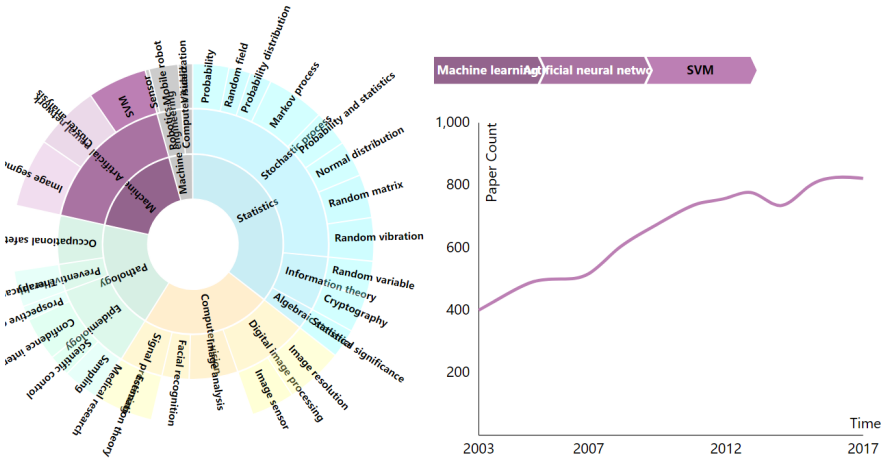

图 6–4 放射状知识图

Fig 6–4 sunburst knowledge graph

如图，运用 d3.js 提供的交互式脚本我们实现了知识图的最终效果。在知识图的规模被合理化的同时，系统会跟踪用户鼠标指针的位置，并显示鼠标指向的块的层次关系和其对应的关键词论文数随时间的变化趋势。至此，基于查询系统的知识图设计功能即完成实现。

全文总结

本次毕设课题的工作成功地完成了 Acemap 的 2.0 版搜索平台的部署。2.0 版本的查询系统成功解决了在 1.0 版本中存在的查询速度与索引速度严重不足和功能缺乏的问题。对于查询速度不足的问题，我采用了将索引文件映射到内存的方法，成功将查询的速度相对原查询系统相比提升了三倍以上；对于索引速度不足的问题，我采用了文件导入法代替数据库导入法，将索引的平均速度提高了超过一百倍，实现了完整索引的成功建立。在另一方面，我实现了查询系统的分布式部署，成功在两台服务器上搭建了系统的分布式云服务，进一步强化了搜索引擎的搜索性能与稳定性。搜索平台的部署工作到此便告一段落，基本的平台维护方式我都整理到了论文的附录中，可以参照附录进行系统的后期维护。

在知识图方面，我一开始的想法是作出一个强理论性的系统，同时也要兼顾系统的展示效果。但是在实际操作中，不同种类的图的展示效果千差万别，并没有一个固定的理论去说按什么标准组织的图的展示效果是最好的，所以后来这部分的工作中，展示效果的优化成了工作的重点，理论部分的工作反而相对随性，只是根据一些参考论文的思想构建了一套自己的规模控制算法，并在自己的系统中使用。由于最终的目标是完成知识图的动态生成，根据用户的输入查询语句直接反馈出相应的知识图，所以算法的复杂度的限制相当高，很多迭代算法因为收敛速度不够也因之被弃用。在论文使用的算法中，使用了一次递归的规模值计算算法和收敛速度较快的 Kmeans 算法，算法复杂度并不高，在实际应用中也有着较好的效果。

很高兴本次毕设课题的任务书中的都能够被圆满地解决，在解决一个个实际问题的过程中，我对搜索引擎这一应用有了更全面的认识，也学到了各种算法与软件方面的知识。感谢这次毕业设计的机会，让我有了许多收获，也激励我在今后研究生阶段的学习和工作中虚心学习，勇于面对一切困难与挑战。

附录 A　搜索平台的启动和维护命令

A.1　单机版架构

A.1.1　启动服务

单机版服务的启动可以直接使用 example 中的 data import handler 示例，启动方式为：

```
cd solr-6.5.0
bin/solr -e dih -m 20g
```

其中-e dih 指启动 data import handler 这一 example，-m 为 JAVA 虚拟机分配内存。启动后，基本功能都已经配置完成，接下来可以从索引结构设计开始进行进一步配置。

当然，用示例启动服务器这一行为并不优雅，我们可以用下面的指令启动一个正式的服务器：

```
cd solr-6.5.0
bin/solr start -h host -p port -d dir -m memory
```

用这个指令，我们可以指定 host 地址，端口地址和服务器文件地址，并开启一个新的未经过配置的服务。这时我们需要从头配置所有的内容，参见第二章中平台配置小节。

A.1.2　关闭服务

关闭某个端口的服务，可以用指令：

```
cd solr-6.5.0
bin/solr stop -p port
```

关闭本地所有端口的服务，可以用指令：

```
cd solr-6.5.0
bin/solr stop -all
```

A.2　分布式架构

A.2.1　启动服务

分布式架构中首先需要启动 Zookeeper 服务：

```
cd zookeeper-3.4.10
bin/zkServer.sh start
```

注意服务被启动到了哪个端口，这个端口号在 conf/zoo.cfg 中指定，之后需要再次用到，此处假设为 1234 端口。启动 zkCli.sh 并创建一个名为 acemap 的新节点：

```
cd zookeeper-3.4.10
bin/zkCli.sh create nodename
```

分别在两台服务器上以云模式启动 Solr 服务：

```
On Solr Server1:
cd solr-6.5.0
bin/solr start -h s1host -cloud -p s1port -z zkhost:1234/nodename -m 20g
On Solr Server2:
cd solr-6.5.0
bin/solr start -h s2host -cloud -p s2port -z zkhost:1234/nodename -m 20g
```

其中 s1host 与 s2host 是两台服务器的主机地址，s1port 和 s2port 是两台服务器希望启动服务的端口号，zkhost 是 zookeeper 服务的主机地址，1234 为之前指定的端口号，nodename 为之前指定的节点名称。

当服务因故退出或计算机重启后，使用以上指令可以使服务重新注册回云端。

管理云平台索引集合：

```
cd solr-6.5.0
增加collection: bin/solr create -c collectionname -shards 2 -replicationFactor 2
删除collection: bin/solr delete -c collectionname
```

其中-shards 表示索引分片数目，-replicationFactor 表示分片备份数目，一般来说，这两个值都与分布式服务器的数目相同。

更新配置文件：

```
在任何一台运行了Solr服务的计算机上:
cd solr-6.5.0/server/scripts/cloud-scripts
./zkcli.sh -z zkhost:1234/nodename -cmd upconfig -confdir configdirectory -
       confname configname
```

其中 configdirectory 为存放配置文件的本地目录，configname 为节点名字（一般与 collection 名字相同），此命令会将配置文件传到 Zookeeper 的名为 nodename 的节点上。

更新配置文件后，需要重新加载集合：

```
在任何一台运行了Solr服务的计算机上:
http://solrhost:port/solr/admin/collections?action=RELOAD&name=collectionname
```

关闭 Solr 服务：

```
cd solr-6.5.0
bin/solr stop -p port
```

关闭 Zookeeper 服务：

```
cd zookeeper-3.4.10
bin/zkServer.sh stop
```

附录 B　索引文件映射到内存的方法

B.1　定位索引文件位置

首先我们需要知道索引文件存放的位置。一般来说，Solr 的索引文件存放于文件目录的./data/index 目录下

```
Index: /home/path/to/solr/collection/data/index
```

B.2　备份索引文件夹

```
cp ./index ./index2
```

B.3　将索引文件夹挂载到内存上

```
mount -t tmpfs -o size=90g tmpfs ./index
```

该操作会花费 90G 内存空间，如果计算机内存空间不足，可酌情减少。不过，该值太小会导致 solr 服务无法启动。

B.4　将索引文件拷贝回目标目录

```
cp -Rf ./index2/* ./index
```

注意不要用 mv 指令直接剪切并粘贴文件。因为内存中保存的文件不稳定，所以请保留一份硬盘备份。

B.5　按正常方式启动 Solr 服务

```
./bin/solr start
```

参考文献

[1] Solr. Apache solr[R/OL]. Apache Software Foundation, 2012. http://digilib.unikom.ac.id/man/php/book.solr.html.

[2] MCCANDLESS E H Michael, GOSPODNETIC O. Lucene in Action: Covers Apache Lucene 3.0[M]. [S.l.]: Manning Publications Co., 2010.

[3] LOVINS J B. Development of a stemming algorithm[J]. Mechanical Translation and Computational Linguistics, 1968, 11: 22–31.

[4] Hunt, Patrick. ZooKeeper: Wait-free Coordination for Internet-scale Systems[J]. USENIX annual technical conference, 2010, 8: 9.

[5] LAMPORT L. Time, clocks, and the ordering of events in a distributed system[J]. Communications of the ACM, 1978, 21.7: 558–565.

[6] Grainger, Trey, POTTER T, et al. Solr in action[M]. [S.l.]: MANNING, 2014.

[7] Achart.js. [R/OL]. https://github.com/acharts/acharts.

[8] Apache. Apache Solr 6.5.0 Ducumentation[R/OL]. Confluence, 2017. https://cwiki.apache.org/confluence/display/solr.

[9] FELDMAN R, DAGAN I, HIRSH H. Mining text using keyword distributions[J]. Journal of Intelligent Information Systems, 2008, 10.3: 281–300.

[10] SU H.-N, LEE P.-C. Mapping knowledge structure by keyword co-occurrence: a first look at journal papers in Technology Foresight[J]. Scientometrics, 2010, 85.1: 65–79.

[11] LI Y, ZHONG N. Mining Ontology for Automatically Acquiring Web User Information Needs[J]. IEEE Transactions on Knowledge and Data Engineering, 2006, 18.4: 554–568.

[12] LOH S, WIVES L K, de OLIVEIRA J P M. Concept-based knowledge discovery in texts extracted from the web[J]. ACM SIGKDD Explorations Newsletter, 2000, 2.1: 29–39.

[13] HARTIGAN J A, WONG M A. Journal of the Royal Statistical Society. Series C (Applied Statistics)[J]. Pattern Recognition (ICPR), 1979, 28.1: 100–108.

[14] WARTENA C, BRUSSEE R. Topic detection by clustering keywords[J]. Database and Expert Systems Application, 2008: 54–58.

[15] ZHAO Q. Keyword Clustering for Automatic Categorization[J]. Pattern Recognition (ICPR), 2012: 2845–2848.

[16] MORADI K T Rizan, MIRIAN M S. Data-Driven Methods to Create Knowledge Maps for Decision Making in Academic Contexts[J]. Journal of Information & Knowledge Management, 2017, 16.01: 1750008.

致　谢

在毕业设计结束之际，我想感谢很多在毕业设计中给过我大力帮助的人。

首先感谢甘小莺副教授和王新兵教授对我的毕设课题的大力支持。二位老师都是网络方向的教授，我在大三的时候，就深受王新兵教授的关照，参与写作了两篇网络方向的发表于国际 A 类学术会议论文。得知我更大的兴趣在工程方面，且研究生阶段的研究方向是编程语言与软件工程后，二位导师大力支持我选择了与自己兴趣紧密相关的毕设课题，并在毕设过程中给了我很大的指导和帮助。

同时，感谢傅洛伊博士对我毕业设计的监督和指导。傅博士在我知识图构思的形成过程中起到了很大的帮助与指导作用，并在每周周进展审核中给了我很多宝贵的建议。在我大四为了申请美国学校助教而忙于准备托福考试时，也给了我莫大的支持和理解。

最后，感谢学校里的贾雨葶同学在文件导入法中提供的帮助和林特同学在知识图理论部分提供的思路，还有许多实验室同学在我不懂的各方面积极地给我答疑解惑。我们实验室的互帮互助，互相分享的氛围使我毕业设计的进行顺利了很多，也让我学到了很多知识。

THE IMPLEMENTATION AND OPTIMIZATION OF LARGE-SCALED ACADEMIC SEARCHING PLATFORM

This paper discussed the problem of the implementation and optimization of large-scaled academic searching platform. To be more concise, is to implement the 2.0 version of the searching platform for the academic website Acemap. On the first prototype of the searching platform, which is implemented before this study, its indexed was incomplete and the relevant functions were insufficient. The reason for these deficiencies is the database structure had been complicated, which created a huge I/O bound for the data import process. Moreover, the over-use of example code had made it difficult to develop self defined functions.

The initial motivation of this study is to settle these problems and implement a new version of well-functioned academic searching platform. On the one hand, my paper needs to find a algorithm to accelerate the data import speed by at least 100 times. On the other hand, I want to develop a complete searching platform from zero to fit in our functional demands, and to implement it on distributed architecture to enlarge its robustness and maintainability.

On the theoretical part, this study aims to develop a knowledge graph module based on the searching platform. The target is, when users input the keyword to submit their queries, our platform can automatically generate a knowledge graph indicating the relevant academic keyword hierarchy, to better assist users for their research purpose.

Alike the 1.0 version of our search platform, we use open source search platform *Solr* to develop our system. And in the distributed platform, we use *Zookeeper*, a distributed service coordination platform, to manage and deposit our server nodes. Based on these tools, we implemented the 2.0 version of our search platform. This work includes the design of system architecture and index structure; fast data import algorithms. platform background and foreground development and the platform's distributed deployment. Now I will describe these works one by one.

The first work is the design of system architecture. Our system architecture contains eight parts, including the website view, website controller, query preprocess, query tokenizer, database, document preprocesser, document tokenizer and search engine server. Compared to the old version, our new version of system is having more indexed fields, which means the index structure is more complicated but supporting more query methods and richer result fields. Also, the new version implemented a complete preprocess and tokenize to the documents and queries, which handled the problem of stopwords, singular-plural pair and language tense. To speed up the query process, I used the memory project file technology to implement the file system of solr's index file folder. It is to put all the index file into memory rather than hard disk, thus making the speed of reading the index file much more faster. As a result, this new method achieved a over 3 times boost in the query speed.

The second work is fast data import algorithm. We innovatively used a file import method in the data import process thus prevented the time-consuming I/O operations. In this part we firstly exported the whole database tables to files, then we used a two level dictionary to load these files into memory. For all the data import process involving table JOIN operation, we replaced them with similar dictionary lookup operation, and we merge our all dictionaries together to form a expanded dictionary. Finally, we write our expanded dictionary into a formatted single .xml file including all the needed fields, and used this .xml file to do our data import. By changing the hard drive reading to memory reading, the data import speed is increased by over 100 times, and the difficult problem of slow data import is solved.

The third work is the platform background and foreground development. The foreground of platform is the UI interface presented as a website, and the background is the controller of the website, together with the searching platform itself. When the users' input is valid, their input on the foreground user interface is sent to the background controller, and in the background controller the query is preprocessed and sent to the search engine. Then, a HTTP response in .json format is returned back to the website controller, processed and showed in the foreground UI. Also, this part of work includes the principle of keyword highlighting and result faceting, and how these functions are implemented into the system.

The last work is the platform's distributed deployment. A singled system just works fine, but is also having its deficiencies. When the computer resource, including memory, CPU and hard disk, is highly occupied, the servers response speed will be greatly delayed. Moreover, once the server is shut down, all the service will be suddenly down and unable to work. A distributed system can solve all these problems, and a well-developed distributed system can achieve config file centralized management, workload arrangement and can handle server nodes down and recover. We used Zookeeper to implement our distributed system. In this part, we described the architecture, deployment, configure and usage of our distributed system. Our final distributed system contains 1 collection, 2 shards and 2 replicas for each shard. Each of the two server include one copy of replica for both of two shards. So, when any server shuts down, we also have one server with a complete copy of index file that can handle all the searching work. For the most time when all the 4 server nodes are alive, Zookeeper where automatically elect a leader node and the leader node will undertake the shard management and workload distribution. When any server node is dead but started again, it will also automatically login back to the cloud without any extra configuration.

After these work, the 2.0 version of searching platform is completed and can be slated to launch on the website. Then I focused on the development of the knowledge graph module. Knowledge graph is a knowledge management tool that makes organizational processes more visible, feasible, and practicable.[16] My motivation of developing this module is stated below: When users are using an academic website, they may only focus on the top few columns of his search result. However, most information buried under thousands of more results are just ignored. So I want to develop a function to summarize and to convert these results into a visible form, and we selected the form to be a layered knowledge hierarchy graph. Detailed examples are listed in the paper. To settle this problem, we used a tree generate algorithm and tree formalization algorithm to generate the basic knowledge graph, and used a set of scale control algorithms to make sure our graph is having proper size. As we want a dynamic knowledge graph generating process, which requires a relatively

low algorithm complixity and quick response, so some of the algorithms with low converge speed are not applied in the system. On the visualization of our knowledge graph, we used the sunburst effect provided by d3.js to draw it on the website. Some other elements are also added next to the graph to provide more information.

In my graduate design I successfully implemented the 2.0 version of our search system, which achieved a over 3 times query speed and a over 100 times data-import speed compared to the 1.0 version. Moreover, I implemented the whole system in distributed mode thus achieving a higher robustness and efficiency. It is glad to see all problems I want to solve having been solved well, and in the process of solving these problems, I had a more well-rounded recognition to the application of search engine. So Thanks for the chance my school providing me to have this graduate design, which have made me learned quite a lot and motivated me to be brave to face to any challenge in front of me.