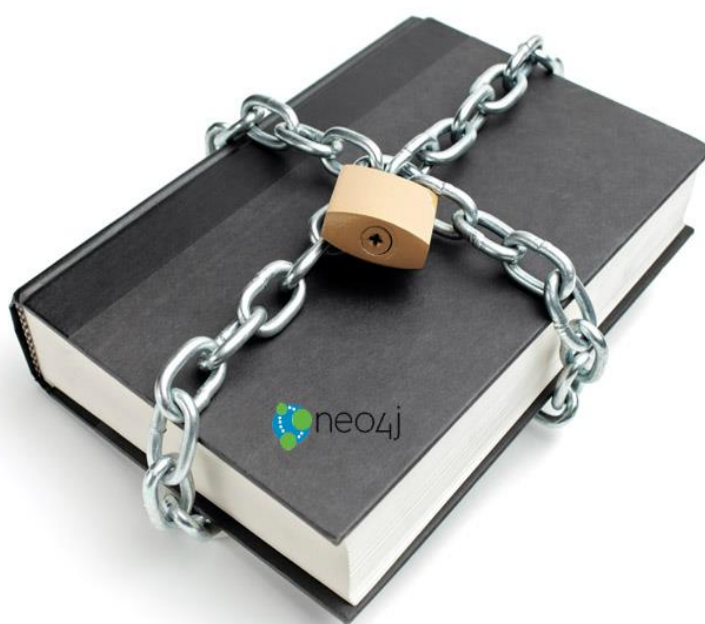


2017 年全国大学生信息安全竞赛

作品报告



作品名称：_____ 基于图数据库的可搜索加密系统 _____

电子邮箱：_____ 572682252@qq. com _____

提交日期：_____ 2017 年 5 月 29 日 _____

填写说明

1. 所有参赛项目必须为一个基本完整的设计。作品报告书旨在能够清晰准确地阐述（或图示）该参赛队的参赛项目（或方案）。
2. 作品报告采用A4纸撰写。除标题外，所有内容必需为宋体、小四号字、1.5倍行距。
3. 作品报告中各项目说明文字部分仅供参考，作品报告书撰写完毕后，请删除所有说明文字。（本页不删除）
4. 作品报告模板里已经列的内容仅供参考，作者可以在此基础上增加内容或对文档结构进行微调。
5. 为保证网评的公平、公正，作品报告中应避免出现作者所在学校、院系和指导教师等泄露身份的信息。

目录

摘要	3
第一章 作品概述.....	4
1.1 背景分析	4
1.1.1 云计算的发展现状	4
1.1.2 云存储安全的发展现状	4
1.1.3 可搜索加密的发展现状	5
1.2 核心项目工作	5
1.3 特色描述	6
1.4 应用前景分析	6
第二章 作品设计与实现.....	8
2.1 系统概述	8
2.1.1 设计思路	8
2.1.2 系统功能	8
2.1.3 系统设计	9
2.2 技术原理	10
2.2.1 可搜索加密的技术原理	10
2.2.2 动态可搜索加密模型技术原理	13
2.2.3 基于图数据库的可搜索加密技术原理	15
2.3 系统实现	20
2.3.1 系统客户端	20
2.3.2 数据库服务端	23
第三章 作品测试与分析.....	29
3.1 测试环境	29
3.1.1 客户端	29
3.1.2 服务器端	30
3.2 实验数据	30
3.3 索引构建的性能实验	32
3.4 索引更新性能试验	34

3.5 结果分析	35
第四章 创新性说明	36
4.1 实现了大数据时代云空间隐私保护	36
4.2 使用图数据库来模拟树状结构	36
4.3 设计了动态可搜索加密引擎框架	37
4.4 平衡树的创新性	37
4.5 可搜索加密算法的创新性	37
第五章 总结	38
5.1 作品技术难点	38
5.1.1 基于图数据库的可搜索加密系统	38
5.2 作品应用前景	39
参考文献	40

摘要

在当今的大数据时代，由于成本低、资源利用率高等优势，云计算得到了大规模的运用，但是在云数据安全存储方面，依然面临着众多需要解决的问题，这些问题吸引了众多研究者的兴趣。当下以对云服务的不信任为前提，加密数据成为必选项，如何让加密数据仍然得到有效的云计算处理成为主要问题，其中最常用的处理是关键词搜索。围绕这个问题也有诸如全同态加密，ORAM等解决方案，但都存在计算成本问题，可搜索加密算法因为对效率和安全的综合考虑，得到了广泛的研究。

但是，随着用户的增加和文件数的增多，传统可搜索加密将索引全部载入内存的方案存在很大挑战，而且串行检索的效率也会衰减，同时，原有加密方案中没有考虑索引的可持久化问题。

针对上述问题，本项目构建了基于图数据库的动态可搜索加密系统。图数据库作为一种新型的分布式NoSQL数据库，可以将内存索引转化为分布式的图节点，具有优异的图形数据处理能力和分布式存储特性，与传统关系型数据库相比，有更强的图形遍历能力，对于实现树状索引具有更强优势。

本文对比分析了目前典型的可搜索加密算法，总结这些工作的优缺点，通过实验对比图数据库和传统数据库在处理图形数据上的性能差异。以关键词替罪羊树为基础，提出基于图数据库的动态可搜索加密方案，描述了索引构建、并行搜索、动态更新的过程和算法。最后，建立了客户端与服务端完成系统实现。

通过对比图数据库系统和传统模型系统，可以得到构建的模型在实现索引持久化的同时，检索和更新效率均优于传统模型。在针对该模型的大数据性能测试中说明了该模型处理大数据量的有效性和可靠性。

关键词：可搜索加密、图数据库、平衡树、云计算、云存储安全

第一章 作品概述

1.1 背景分析

1.1.1 云计算的发展现状

云计算是融合了分布式计算、网络存储技术、并行计算等传统计算机技术和网络技术的一项新的技术。它既提供计算服务，也包括云存储服务，能够将计算和存储功能迁移到云服务器上进行。随着 Internet 网络应用技术的发展和普及，网络用户和网络数据量高速增长，从而对数据的处理能力提出了更高的要求。

在海量数据爆发的当今，为了减轻数据存储和维护的负担，越来越多的用户选择将本地的数据迁移到云端服务器上。与此同时，网络资源的需求和利用也出现失衡状态，大量的网络资源没有得到充分利用。因此，资源的整合和优化是当代网络发展的必然趋势，在这种背景下，云计算应运而生。

在过去的十年中，各种基于云计算的技术已经广泛运用到各行各业中。云计算服务有多种服务框架提出和应用，如软件即服务、平台即服务、基础设施服务等。构建在云上的基于云存储方案的各种存储工具也得到了大规模的应用，如Google云盘、DropBox、百度云等，截止目前，百度云的用户数已经超过4亿，用户可以轻松的上传、检索、分享文章，实现了内容与硬件的分离^[1]。

1.1.2 云存储安全的发展现状

随着云计算的逐步发展，云数据存储安全性问题变成了影响和制约云计算水平的关键因素，也对用户造成了严重的不良影响，甚至还影响到用户对云技术的认可和青睐。通过对云环境下数据存储的现状进行剖析，可以将面临的问题主要分为数据加密存储、数据安全审计、数据残留以及数据隔离这4个方面。

近年来，与云存储内容泄密相关的事件也时有发生，据报道，2011年黑客入侵Sony公司造成数亿用户个人信息泄露，同年Google旗下的产品Gmail用户信息也遭到了恶意泄露。这引起了人们对云端数据安全性的担忧，同时阻止了云计算技术在隐私数据处理中的应用，如大规模在线病例存储等。

目前常用的隐私保护方案是在数据文件离开用户控制前将其加密，并将密文存储在云端。这种解决方案可以实现端到端的安全，但是却无法应用在云计算场景中。因为数据被加密后，云端无法对它进行任何的有效计算，例如对文本文件进行关键词搜索，对文本内容的机器学习等^[2]。

1.1.3 可搜索加密的发展现状

通过关键词搜索并下载文件是云存储最基础的功能，为了同时实现文件加密和关键词搜索，可搜索加密的概念被提出，并迅速得到了广泛的研究。根据实现方式的不同，可搜索加密可以分为基于对称密钥的可搜索加密和非对称密钥的可搜索加密，同时也有学者提出了全同态加密和 ORAM，后两者从安全的角度看是问题的最优解，可以确保在存储和查询过程中，不泄露任何的明文信息甚至是搜索结果。

但是全同态加密基于数学难题，算法设计难度大或者计算量大，距离可以实用尚远。ORAM 具有同样的问题，在查询及更新过程中效率尚达不到实用要求。可搜索加密实现了安全性与效率、可用性之间的平衡，更具有实践意义，得到了广泛的研究。

目前主要有两种实现可搜索加密的方式，一种由Goh提出^[1]并在文献^[2]中使用，为每一个文档构造加密的数据结构，这种结构支持对关键词的检查能力，时间复杂度为 $O(n)$ ，其中 n 为文档的个数；第二种由Curtmola提出^[3]，为整个文档集合构建倒排索引，通过搜索关键词找到对应的文档位置，这种方式的时间复杂度是 $O(k)$ 。其中 k 为关键词的个数。由于第二种方案是时间复杂度上的最优解，所以有许多构建算法采用第二种方案。

1.2 核心项目工作

在倒排索引的构建方案中存在以下几个问题：索引是静态的，当文件发生变化时需要重新构建索引，当索引较大时，重新构建的开销会很大；索引的搜索过程是串行化的，因为索引结构的唯一性导致无法很好的实现并行搜索；没有考虑索引的大小问题和持久化问题。

本项目研究了国内外现有的可搜索加密构建方案，总结了现有技术的缺陷。同时，借鉴优秀的索引设计方案，提出了基于图数据库的动态化可搜索加密技术方案，最后通过实验和性能测试检验了系统设计的可行性和效率。

项目的核心工作主要有以下几点：

1. 关键词平衡树实现
2. 文件 AES 加密与关键词口令生成
3. 关键词检索
4. 动态可搜索加密服务端 API 接口实现：
 - a) 处理 HTTP 请求：包括 GET, POST, PUT, DELETE 请求的对应接口。
 - b) 用户认证接口：确保只有认证用户才可以调用服务端 API 接口。
 - c) 索引持久化：与 SQLite 或 Neo4j 连接。
 - d) 关键词检索灵活化
5. 客户端界面与后端实现
6. 客户端与服务端通信模块
7. 动态增删关键词索引
8. 分布式图数据库支持

1.3 特色描述

1. 提出了一种新的随机化加密关键词平衡树算法；
2. 用图数据库实现了加密索引的持久化存储；
3. 实现了并行搜索，提高了存储与读写、查询效率；
4. 设计了对称可搜索加密引擎框架，为其实用化做了基础研究；
5. 实现了支持文件检索的数据隐私保护存储原型系统；
6. 分析了可搜索加密文件存储的安全特性及读写性能。

1.4 应用前景分析

本项目的主要应用人群为企业用户以及安全性需求较强的个人用户，针对适用人群对云存储安全的高要求性及时效性。可搜索加密凭借其诸多优点，已经成为当今云存储中数据隐私保护的重要研究方向，最新的研究成果动态可搜索加密系统大幅度的提升了效率与使用灵活性。然而还是没有解决索引只能加载在内存中的问题，无法实现理想的可持久化存储。

本项目创新的提出了使用图数据实现“动态可搜索加密系统”的思想，解决了可

搜索加密系统中的隐私保护、持久化存储以及并行扩展等问题。本项目所设计的方案支持索引的动态更新，支持关键词的状态增加与删除，该模型检索和更新效率均优于传统模型，具有一定的理论价值同时有较高的实用价值。

第二章 作品设计与实现

本章分为三个部分，第一个部分系统概述，说明了设计思路并简单说明客户端和服务端；第二部分是技术原理，介绍了可搜索加密相关理论，给出动态可搜索加密的定义；第三部分是技术实现，提出了基于图数据库的动态可搜索加密方案，介绍了方案使用的核心数据结构，并围绕第二章的定义，详细描述了索引的构建、更新算法和关键词的搜索算法。该方案解决了现有方案中无法处理大索引的问题，同时实现了索引的可持久化和搜索的并行化。

2.1 系统概述

2.1.1 设计思路

可搜索加密技术的目标是在不影响数据检索功能的条件下，保护用户外包数据的安全与查询隐私。

图 2-1 展现了本系统的原型设计架构。本系统由一台服务端和多台客户端组成。服务端作为云端，存储用户保存的可搜索的加密数据，并接受用户发送的 GET, POST 等 HTTP 请求，对数据进行对应的获取，添加，查询，删除操作。我们用关系型数据库来保存每个文档密文，而将存储每篇文档的关键词的信息储存在图数据库中，并构成一棵自平衡二叉树，用于进行关键词检索操作。服务端再将搜索结果返回给客户端，完成一次请求处理。

用户登录客户端后，通过关键词搜索含有关键词的文档。其他操作如上传文件，删除文件，查询关键词等操作，都将通过网络请求发送到云端。客户端关闭后，所有数据将在本地消失，而云端数据将持久化保存。

2.1.2 系统功能

表 2.1 描述了系统原型包含的基本功能。

表 2-1 系统功能设计

序号	功能描述
1	云端存储可搜索的加密数据
2	云端接受客户端发出的搜索、添加、删除等请求
3	客户端发起搜索、添加、删除等操作
4	客户端阅读解密后的数据

2.1.3 系统设计

1) 客户端

用于给用户进行搜索，阅读，上传等操作。上传的本地数据和搜索的关键词组会先在客户端先进行加密，将密文传输给云端，云端既不会接触到数据的明文，也不会知道搜索的关键词组。当数据返回给客户端时，客户端需要将加密的数据解密后呈现给用户。

同时，由于可搜索加密可能泄露用户的搜索模式（长时间统计），客户端将在发送关键词组时按照泊松分布随机添加一些伪装关键词，以此来掩盖可能泄露的搜索模式。客户端本地存储伪装关键词的映射表，用来筛选服务端返回的多余数据，最终在客户端界面显示正确的搜索结果。

2) 服务端

用于持久化的保存数据和接受客户端的搜索请求。服务端获取的加密数据包含文档的基本信息（标题，正文）和该文档的关键词序列。数据将以平衡树的结构存储在图数据库中，通过引入平衡树提高查询的效率。

同时，为了保证系统的安全性。云端需要实现用户身份认证（User Authorization）。认证通过的用户请求才被接受处理，而对于认证未通过的请求，返回 403 错误。

云端处理请求时，将对应索引和密文信息存储在本地或数据库中，实现索引和密文的持久化，防止信息丢失。

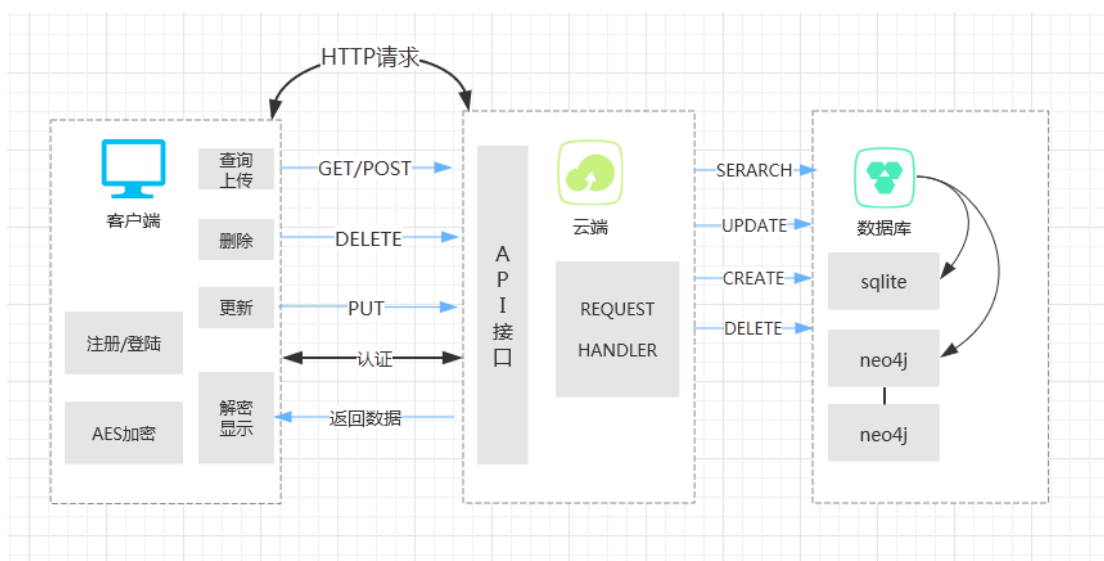


图 2-1 系统设计原型架构图

2.2 技术原理

2.2.1 可搜索加密的技术原理

为了能够实现用户检索存储在第三方服务器上的加密数据,我们选择性能较好的加密方法

a. 全同态加密和 ORAM

如果存在一种加密方式使得在密文空间上进行关键词搜索等操作,而且过程中不需要解密密文不泄露任何信息,那么问题就可以得到完美解决。全同态加密技术便是能够实现这种假设的一种重要技术。

全同态加密问题是一个密码学难题,被称作“密码学的圣杯”,最初这个概念也被称作隐私同态,由 Rivest, Adleman 和 Dertouzos 在 RSA 算法发明后不久提出。直到 2009 年,IBM 研究员 Gentry 在他的论文中首次从数学角度提出了全同态加密的可行方法。他证明在该方案中,对密文进行一个操作后得到输出,对这个输出进行解密后的结果与对明文进行同样操作后的结果相同,即任何可以对明文进行的操作也可以应用在密文空间。这被视作全同态加密领域的一个重要突破。

全同态加密非常适合非授信云计算场景,用户可以上传加密后的文件,云服务器可以对密文做排序、关键词搜索等任何操作,就像在明文中那样,但是无法得到明文结果。

然而，在 Gentry 的方案中设计使用了“理想格”数据结构，用矩阵和向量对明文进行加密，在计算过程中需要对每个元计算，在计算完矩阵后还需要再计算数据本身，巨大的计算量限制了方案的可实践性。Smart 在 Gentry 的基础上提出改，将使用矩阵和向量加密的方式改为使用数字和多项式加密。极大的简化了计算过程，使得方案可以在电脑上实现和测试。

但是 Smart 的方案也存在局限性，在多次连续加密后结果质量衰减，同时并不能针对任意计算，不能算是真正的全同态加密。所以全同态加密距离真正实用化还需要更多的努力。

用户访问模式指的是在一段历史时间内用户的内存访问路径。云服务器通过统计一段时间内用户的访问行为，结合现有的数据挖掘技术，可以分析出用户的行为模式，有概率获得有价值的信息。不同于全同态加密，ORAM 从云计算隐私保护的角度考虑问题，它是隐藏用户访问模式的一种重要方法。

可搜索加密以及其他技术手段可以解决存储隐私保护，但是 ORAM 可以彻底解决访问模式的隐私保护。当执行一个对内存随机读写的程序时，ORAM 的做法是将一次 RAM 操作变成一组 RAM 操作来掩盖真实的访问路径。最简单的 ORAM 思想是，当用户请求一个数据时，服务器顺序读取所有数据块发送给用户，用户解密找到真实需要的数据，然后将数据重加密后再返回给服务器。这样服务器就无从得知用户真实需要的数据，实现了对访问路径的掩盖。但是这样的想法需要每次请求都传送所有数据，并将所有数据加解密。计算和通信开销都是巨大的，没有实践意义。

1996 年 Goldreich 和 Ostrovsky 提出了 ORAM 的概念和两种实现方案。但是即使其中最高效的方案，服务器的存储复杂度为 $O(n \log n)$ ，单次请求的通信次数为 $O(\log^3 n)$ ，最差情况为 $O(n^2 \log n)$ ，其中 n 为数据项的个数。由于该方案的构建复杂，效率较差，在 2010 年，Pinkas 改进了其中的分层 ORAM 方案，提高了效率。在该方案中，数据以层状结构存储，当用户请求数据时，服务器使用特殊的散列方法计算该数据在每层中的两个位置发给用户，用户找到需要的数据后需要将数据重新加密发给服务器，同时需要对数据进行混淆。Pinkas 的方案将空间复杂度改进到 $O(n)$ 计算复杂度降低到 $O(\log^2 n)$ 。2011 年，Shi 提出了基于二叉树的方案，避免了复杂的混淆过程。

但是，ORAM 方案目前还是存在构建复杂难以实现等问题，同时给系统增加的额外开销也很大，这些都限制了在实践中的应用。研究人员开始通过降低部分安全要求来

获得安全和效率的平衡，各种可搜索加密方案也陆续提出。

b. 可搜索加密

可搜索加密技术被认为是可以平衡隐私保护和数据检索的重要手段，得到了国内外学者的大量研究。根据可搜索加密的加密方式可分为基于对称密钥的可搜索加密和基于公私钥的可搜索加密。

公钥加密体制由于其公私钥非对称的性质，非常适合多用户的收发场景。考虑云存储中实现多用户的信息共享，用公钥加密体系更容易实现，因为收发方并不需要提前建立安全通道协商密钥，只要发送方使用对方公钥加密，接收方可以自行使用私钥解密。公钥可搜索加密技术就是建立在这之上的技术。

Boneh 于 2003 年首次提出了公钥可搜索加密的概念和实现方案。该系统以邮件系统为应用场景，提出了对加密邮件进行关键词搜索的方法。Boneh 的实现方案的核心是使用了基于椭圆曲线上的对数问题的公私钥加密算法，在检索关键词的过程中需要对每个关键词做对运算，计算复杂度较高。虽然后面有很多工作改进了最初的设计，减少了计算量，但服务器的计算压力仍比较大，无法适应大规模应用的场景。

对称可搜索加密（SSE）相对于公钥可搜索加密的优势在于可以构建更灵活的方案，令检索在小于线性时间内完成，缺点在于如果需要信息共享，则接收方和发送方需要同时建立安全信道并分发密钥。

对于实现高效的 SSE，大致可以分为两类方法。第一种方法由 Goh 提出，在文献^[2]中使用，给每个文档都关联一个加密的数据结构，该数据结构可以检测每个关键词的出现位置。这种方法自然导致对于文件集合的检索复杂度为 $O(n)$ ， n 为文件集合的大小。

得到广泛研究的另一种方案即为安全索引方案，其中索引是一种数据结构，储存了文件集合和关键词的某种映射关系。当提供某一关键词时，索引可以返回包含该关键词的文件指针。如果在没有关键词“陷门”时，索引不泄漏任何关键词和文件集的映射信息，则称该索引为安全的。Goh 在文献^[1]中提出了构建安全索引的方法论：客户端构建文件集合的索引并将文件加密，然后将索引和密文上传到服务器端。搜索关键词 w 时，客户端生成关键词 w 的“陷门”并发送到服务器端，服务器端将陷门传入搜索方法，得到对应的密文指针，最后根据指针找到密文并发送给客户端，客户端解密得到包含关键词 w 的文本。

Curtmola 提出了为整个文件集合构建加密的倒排索引的方法,这种方法较第一种更为高效,时间复杂度为 $O(r)$, 其中 r 为文件所包含的关键词的个数。 $O(r)$ 的时间复杂度是最优解, 由于它的高效, 倒排索引的方法也有很多后续的研究。

Song 是第一个明确提出可搜索加密问题的人,并在文献中给出了一个非互动式的、时间复杂度为 $O(n)$ 的解决方案。随后 Goh 为 SSE 提出了正式的安全定义,并给出了一个基于 Bloom 滤波器的时间复杂度为 $O(n)$ 的方案,但是没有明确的语义安全性。Chang 和 Mitzenmacher 提出了另一个安全定义,并给出了一个具有明确语义安全性的 $O(n)$ 的解决方案。然而 Goh 和 Chang 的方案都不是并行的,或者说需要 n 个内核的并行。

Curtmola 首次给出了接近最优时间复杂度 $O(r)$ 的构建方案 (SSE-1 和 SSE-2)。SSE-1 被证明是 CKA1-secure 的,然而 Curtmola 指出,CKA1-secure 是无法满足实践需要的,并给出了安全性更强的 CKA2-secure 的构建方案 SSE-2。

但是以上这些方法都只能适应静态文件集合,无法实现关键词和文章的动态变化。2010年Kamara提出了一个CKA2-secure并且是最优复杂度的动态SSE构建,但是会在更新时泄露关键词信息。Kamara在之后又提出了一个改进版本,解决了泄露问题,同时简化了索引的构建过程。

2.2.2 动态可搜索加密模型技术原理

a. 动态可搜索加密模型

动态可搜索加密模型是一个多项式时间算法的八元组：
(Gen, Enc, SrchToken, Search, UpdHelper, UpdToken, Update, Dec), 其中：

$K \leftarrow \text{Gen}(1^k)$: 概率算法, 输入安全参数 k , 输出密钥 K 。

$(\gamma, c) \leftarrow \text{Enc}(K, \delta, f)$: 概率算法, 输入是密钥 K , 索引 δ , 文件集 f , 输出加密后的索引 γ 和密文集。

$\tau_s \leftarrow \text{SrchToken}(K, w)$: 概率算法, 输入是密钥 K 和一个关键词 w , 输出搜索令牌 τ_s

$i_w \leftarrow \text{Search}(\gamma, c, \tau_s)$: 确定性算法, 输入是加密后的索引 γ , 密文集 c 和搜索令牌 τ_s , 输出是一组标记 i_w 。

$\text{info}_{i,u} \leftarrow \text{UpdHelper}(i, u, \gamma, c)$: 确定性算法, 其中 i 表示文件标示符, u 表示更新

的类型 $\{add, delete\}$, 表示加密后的索引, c 表示密文集。输出 $info_{i,u}$ 包含了此次更新的信息。

$\tau_u \leftarrow UpdToken(K, f_i, info_{i,u})$: 概率算法, 输入三个参数, 其中 K 为私钥, f_i 为标示符为 i 的文件, $info_{i,u}$ 是由 $UpdHelper$ 产生的更新信息。输出一个更新令牌 τ_u , 其中 u 属于 $\{add, delete\}$ 。

$(\gamma', c') \leftarrow Update(\gamma, c, \tau_u)$: 确定性算法, 输入 γ, c, τ_u , 输出更新后的索引和密文集。

$f \leftarrow Dec(K, c)$: 确定性算法, 输入密钥 K 和密文 c , 输出明文 f 。

此定义是 Kamara 提出^[4]的动态可搜索加密的形式化定义, 对比第 1 节 Goh 提出的 SSE 的定义, 主要的区别在于 2.2.2 的模型中定义了更新索引的方法, 通过产生一个更新陷门发送到服务器, 服务器调用更新操作修改索引。

根据上述定义, 可以给出动态可搜索加密算法 (DSSE) 的正确性的定义。

Kamara 提出的动态可搜索加密已经达到了较高的安全标准^[6], 但我们基于此定义提出了一个算法理论的更新。我们在产生索引即生成 δ 的过程中, 提出了随机化索引原则。即在原来确定性算法的基础上, 引入概率算法。我们在动态可搜索加密的形式化定义第一, 二条中插入随机化索引规则, 在第八后插入筛选规则。具体定义为:

$\delta \leftarrow P(K, salt)$: 概率性算法, 输入密钥 K 和随机化参数 $salt$, 输出索引 δ 。其中, P 为概率性函数, 概率性插入无关关键词, 打乱索引顺序, 产生满足Poisson分布的增强索引。

$f_T \leftarrow Filter(f, salt, K)$: 确定性算法, 输入明文 f , 随机化参数 $salt$ 和密钥 K 。输出真明文 f_T 。

通过引入这两条规则, 更新的动态可搜索加密的算法可以不泄露一些信息, 包括关键词的数目, 搜索得到的文件数目, 部分用户查询模式。这相对于传统动态可搜索加密算法在安全性上达到了更高的标准^[6]。付出的代价是, 降低了关键词查询效率, 同时额外增加了随机化索引生成和明文筛选的开销。

b. 动态可搜索加密的正确性

令 D 表示一个拥有八元组的动态可搜索机密模型, 如上定义。称 D 是正确的, 如

果满足以下条件：对于所有的 $k \in N$ ，所有的由 $K \leftarrow Gen(1^k)$ 产生的 K ，所有的二元组 (δ, f) 和所有的由 $Enc(K, \delta, f)$ 输出的元组 $Enc(K, \delta, f)$ ，所有成功执行的 $Update(\gamma, c, \tau_u)$ ，其中 τ_u 是 $UpdToken(K, f_i, info_{i,u})$ 由产生的更新令牌，所有的文件 f_i 和所有的 $u \in \{add, delete\}$ ，所有的关键词 w ，所有的由 $SrchToken(K, w)$ 产生的令牌 τ_s ，所有的由 $Search(\gamma, c, \tau_s)$ 产生的标识 id ，明文集 $f_w = (Dec(K, c_i); i \in i_w)$ 等于在原文件集合 f 中所有含关键词 w 的文件的集合。

c. 动态可搜索加密的安全性

对于一个安全的可搜索对称加密模型，需要完全实现对用户的隐私保护，其中又分为访问用户的隐私保护、关键词搜索的隐私保护和访问模式的隐私保护。访问用户的隐私保护类似常规的服务器架构，可以使用访问控制列表、权限控制等常规安全手段实现。对于可搜索加密模型构建方案而言，更看重如何实现后两者。

根据[6]指出，传统的动态可搜索加密模型实现了下面两个功能：

(1) 给出一组密文集 c 和为它构建的加密索引 γ ，没有攻击者能够获得明文集 f 与内容相关的任何信息。

(2) 在(1)的基础上，额外任意产生一组关键词序列 $q = (q_1, \dots, q_t)$ 并给出基于它产生的搜索令牌集 $\tau = (\tau_1, \dots, \tau_t)$ 没有攻击者可以了解到明文集 f 或者关键词序列 q 与内容相关的任何信息。

第一个功能确保了加密内容的安全性，即密文与索引不泄露任何明文信息；第二个功能确保了用户与服务器端交互的安全性，即服务器无法在交互过程中获得任何有价值的信息。传统的动态可搜索加密达到了 CKA2 级别的安全等级，从而成功的实现了关键词的隐私保护和访问模式的隐私保护。

然而，根据[6]的证明，传统动态可搜索加密泄露了以下信息，包括关键词的个数，密文的个数、密文的大小、密文与加密后的关键词对应信息、用户搜索模式，用户获取模式。而本节 a 部分提出的优化算法减少了关键词个数，密文大小，部分密文与加密后关键词的对应信息和部分用户搜索模式。

2.2.3 基于图数据库的可搜索加密技术原理

a. 关键词替罪羊树的原理

下面，我们来介绍一下我们所选择的平衡树的基本原理以及一些实现细节。

上文中提到过，我们需要将所有文章信息存储在一棵平衡树的叶子结点，并自底向上维护每颗子树所包含的关键词情况。

经过反复思考，考虑到信息只存储在叶子结点会导致插入文件时节点的旋转操作十分繁琐，并且可能带来一些性能问题，最终我们决定采用一种不通过旋转来完成自平衡的平衡树——替罪羊树。

替罪羊树是一种不需要旋转的平衡树，它保持平衡的关键在于重建操作。我们可以定义每棵子树的不平衡度 α 为：

$$\alpha = \frac{\max(Size_{LeftSon}, Size_{RightSon})}{Size_{total}}$$

即不平衡度为左右子树里面较大的那棵子树所包含的节点数占整棵子树所包含结点数的比例。

当这个比率超过一个预设的阈值(我们采用的是 70%)时，我们认为这棵子树不平衡。而每次插入删除以后，都有可能导致替罪羊树中出现一个甚至多个节点，以它们为根的子树被定义为不平衡，并且这些节点一定是祖孙关系。此时，替罪羊树为了保证平衡，会找出这些节点中深度最浅的那个(也就是子树范围最大的那个节点，该节点也被称为“替罪羊”)，将以该点为根的子树重建成完全平衡的二叉树。

众所周知，替罪羊树插入、删除节点的平均最坏时间复杂度为 $O(\log n)$ (其中 n 为整棵替罪羊树所包含的节点数)，并且从实际测试上来看，替罪羊树由于其常数小的特点，在性能上领先于其他平衡树，甚至与红黑树相差无几。因此综合各方面需求之后，我们选择了这种平衡树来维护文章信息。

综上所述，修改后的替罪羊树能够满足我们的所有需求，并且其还具备常数小，编程复杂度较低的特点，是一个很不错的选择。

由于设计的需求，我们需要把信息只存储在叶子结点上，所以需要对原版的替罪羊树做一些修改。下面我们会对几个操作过程分别进行说明。

b. 索引的构建

将原子树中所有的叶子结点取出，然后以它们为叶子结点，重新建立一棵深度为 $\log(2 * \text{叶子数量})$ 的平衡二叉树。

其中构建算法的伪代码如表 2-2 所示

ReBuild: 将以 root 为根的子树重建成一棵高度平衡的二叉树，并返回根节点。

Build: 将 List 中区间[1, r]的节点作为叶子结点构建成一棵高度平衡的二叉树，并返回根节点，该函数是一个递归的过程。

表 2-2 关键词平衡树构建算法

```
1  def Build(l, r, List, father):
2      if l == r: # List[l] is a leaf of new tree
3          List[l].father <- father
4          return List[l]
5      root <- new node;
6      node.father <- father
7      mid <- (l + r) / 2
8      node.LeftSon <- Build(l, mid, List, node)
9      node.RightSon <- Build(mid + 1, r, List, node)
10     UpdateNodeImformationWithSons(node)
11     return node
12  def ReBuild(root):
13      List <- FindAllLeaf(root)
14      return Build(1, List.size, List, root's father)
```

c. 索引的更新

插入:

与其他平衡树一样，替罪羊树的插入也是在叶子结点的下方新建儿子节点来完成插入。此处，我们的信息都只存储在叶子上，因此我们在走到叶子结点后，新建两个节点，分别作为该叶子结点的左、右儿子。然后根据插入文章与原先叶子结点上存储文章 id 的大小关系，将两篇文章放进对应的两个新建的节点，然后删除原先叶子节点上的信息，标记其为“中间节点”。接着，开始逐步向上返回根节点，路上调整好每个节点对应的键值来保证满足平衡树中序遍历键值序列不降的性质(叶子节点的键值就是它们的 id，这是固定的)。最后，找出可能存在的替罪羊节点，重建子树。

其中构建算法的伪代码如表 2-3 所示

Insert: 将 article 作为一个叶子结点插入以 NowRoot 为根的子树中，该函数是一个递归的过程。如果函数结束后 NeedRebuildNode 不是空的，那么就调用重建函数重建。

表 2-3 插入操作中树的结构变化

```

1  def Insert(NowRoot, article)
2      if NowRoot is a NULL node:
3          NowRoot <- new TreeNode(article)
4          NowRoot.key <- article.id
5          return
6      if NowRoot is a leaf:
7          NowRoot.LeftSon <- new TreeNode
8          NowRoot.RightSon <- new TreeNode
9          if NowRoot.article.id < article.id:
10             NowRoot.LeftSon.article <- NowRoot.article
11             NowRoot.LeftSon.key <- NowRoot.article.id
12             NowRoot.RightSon.article <- article
13             NowRoot.RightSon.key <- article.id
14          else:
15             NowRoot.RightSon.article <- NowRoot.article
16             NowRoot.RightSon.key <- NowRoot.article.id
17             NowRoot.LeftSon.article <- article
18             UpdateNodeImformationWithSons(NowRoot)
19             return
20  if article.id < NowRoot.key:
21      Insert(NowRoot.LeftSon, article) # Insert article in left subtree
22  else:
23      Insert(NowRoot.RightSon, article)#Insert article in right subtree
24  UpdateNodeImformationWithSons(NowRoot)
25  if max(NowRoot.LeftSon.size, NowRoot.RightSon.size) > Threshold(0.7) *
    NowRoot.size:
26      NeedRebuildNode <- NowRoot

```

删除:

由于信息一定在叶子节点上, 因此这里要删除的一定是叶子节点。我们找到对应节点后, 直接删除, 然后开始逐步返回根节点, 如果在过程中发现某个中间结点没有任何一个儿子, 那么这个节点就也可以删除了。最后, 查看返回根的路径上是否存在

替罪羊节点，如果有，则重建对应子树。

其中构建算法的伪代码如表 2-4 所示

Remove: 将存放着 article 的叶子结点从以 NowRoot 为根的子树中删除，该过程是递归调用的。如果函数结束后 NeedRebuildNode 不是空的，那么就调用重建函数重建。

表 2-4 删除操作树的结构变化

1	def Remove(NowRoot, article):
2	if NowRoot.article is the same as article:
3	delete NowRoot
4	NowRoot <- NULL
5	Return
6	if article.id < NowRoot.key:
7	Remove(NowRoot.LeftSon,article)#remove article in left subtree
8	else:
9	Remove(NowRoot.RightSon,article)#remove article in right subtree
10	UpdateNodeImformationWithSons(NowRoot)
11	if NowRoot is not a leaf before while it's a leaf now:
12	delete NowRoot
13	NowRoot <- NULL
14	return
15	if max(NowRoot.LeftSon.size, NowRoot.RightSon.size) > Threshold(0.7) * NowRoot.size:
16	NeedRebuildNode <- NowRoot

d. 模型索引的检索

该部分由于不改变树的结构，因此不需要有什么修改，只要按照原来的需求，查询当前节点为根的子树中关键词的包含情况来决定是否继续向左儿子或右儿子询问，走到叶子节点则加入查询结果就行了。

其中构建算法的伪代码如表 2-5 所示

Query: 在以 NowRoot 为根的子树中，找出所有包含 KeyWord 关键词的叶子结点所存放的文章 id。

表 2-5 搜索操作树的结构变化

1	def Query(NowRoot, KeyWord):
2	if KeyWord is not in NowRoot.SubtreeKeyWord:
3	return
4	if NowRoot is a leaf:
5	put NowRoot.article.id into the result list
6	return
7	if NowRoot.LeftSon exists:
8	Query(NowRoot.LeftSon, KeyWord) # Search in the left subtree
9	if NowRoot.RightSon exists:
10	Query(NowRoot.RightSon, KeyWord) # Search in the right subtree

最后考虑算法复杂度，这里我们假设文章数量为 N ，来分析一下修改后的替罪羊树的时空复杂度。

(1) 空间复杂度：每次插入均新增加两个节点，重建一棵包含 K 个叶子节点的子树，可以做到重建出来的子树大小规模为 $2K$ ，因此总的空间复杂度为 $O(N)$ 。

(2) 时间复杂度：对于插入、删除操作，由于树的大小规模是 $O(N)$ 级别的，因此这两个操作的平均最坏时间复杂度为 $O(\log N)$ 。对于查询操作，不难发现时间复杂度为 O (符合要求的文章数量)。

2.3 系统实现

2.3.1 系统客户端

客户端模拟云服务提供给用户查看，查询和操作自身数据的界面。客户端采取 web 前端技术，实现了 pc 端和移动端的访问，也同时实现跨平台的用户访问需求。

2.3.1.1 React 与 Redux

客户端的前端开发框架采用了 Facebook 开发的 Reactjs。React 采用独特的 jsx 语法，采用模块化（Component）的开发方式，实现了高度可复用性。React 采用单向数据流，让每个模块根据数据自动更新，改善程序的可预测性。React 采用虚拟 DOM 的

技术，在虚拟的 DOM 上实现了一个 diff 算法，当要更新组件的时候，会通过 diff 寻找到要变更的 DOM 节点，再把这个修改更新到浏览器实际的 DOM 节点上，所以在 React 中，当页面发生变化时实际上不是真的渲染整个 DOM 树，只需要渲染需要重新渲染的部分，节省了渲染开销。

前端的数据流管理采用 Redux。Redux 采用有限状态自动机的概念，把 web 应用视为一个自动机，视图与状态对应。但状态改变时，UI 视图随之改变。Web 应用中所有的状态保存在一个对象中（在 js 里，就是一个 Object 对象）。当事件发生时，比如云端发送新的数据、用户进行交互式操作时，将会触发一个 action，action 包含类型（说明你所要进行的操作）和传递的参数。reducer 将根据 action 的类型和 action 的参数，修改现有的状态（state），并且返回一个新的 state。React 组件收到更新的 state 信息后，判断自身需要更新自身的 ui 界面。

2.3.1.2 Bootstrap

Css（层叠样式表）采用了 Bootstrap。Bootstrap 是 Twitter 推出的一个用于前端开发的开源工具包。Bootstrap 提供了全局的 CSS 设置、定义基本的 HTML 元素样式和可扩展的 class。Bootstrap 还提供了丰富的可重用的组件，用于创建图像、下拉菜单、导航、弹出框等等。

2.3.1.3 界面展示

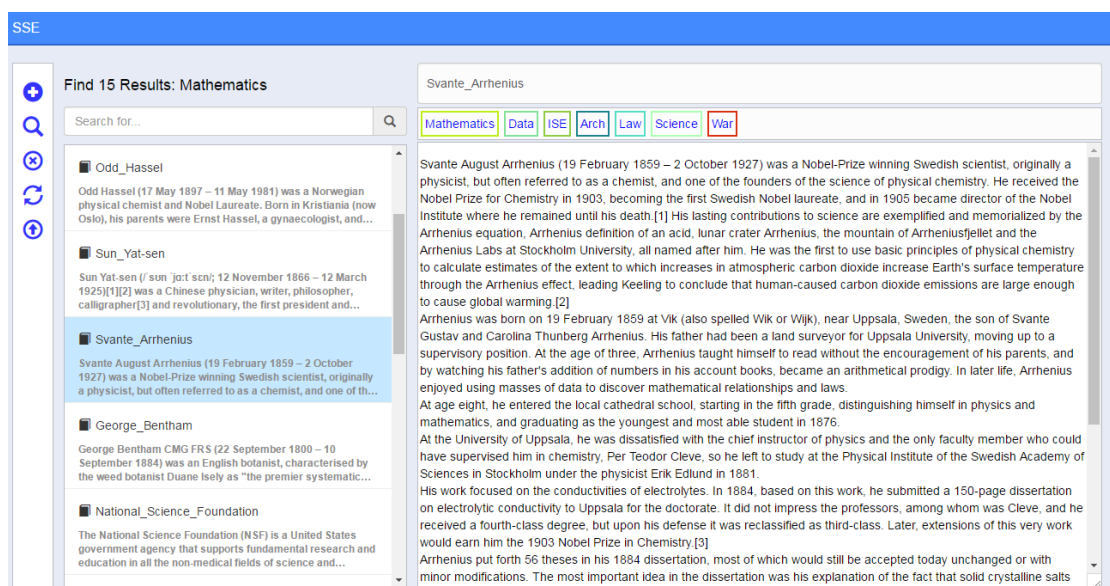


图 2-2 客户端界面

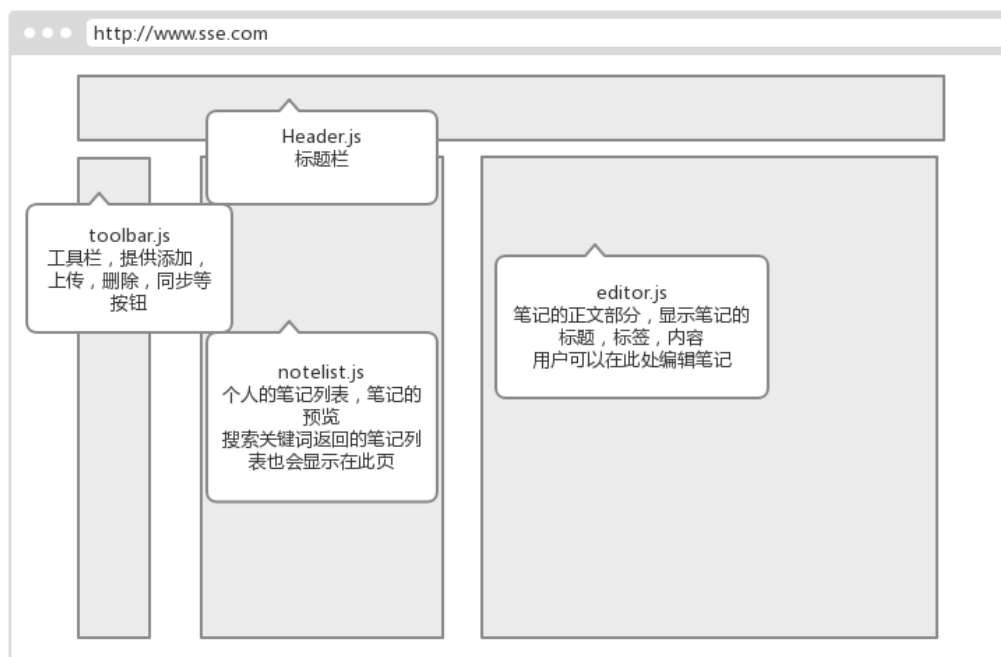


图 2-3 客户端设计原型图

2.3.1.4 客户端使用说明

(1) 搜索：搜索是我们实现的核心功能。首先你可以再搜索框中输入你说需要的关键词，比如 SHA ，然后点击搜索按钮。客户端将通过 Http 与后端进行交互数据。

搜索支持与、或操作。详细的使用说明见下表：

表 2-6 客户端关键词检索使用实例

操作	示例
搜索单个关键词的文档	SHA
搜索既包含关键词 1，又包含关键词 2 的文档	SHA & Arch
搜索包含关键词 1 或者关键词 2 的文档	SHA Arch
搜索包含关键词 1，但不包含关键词 2 的文档	SHA - Arch

(2) 上传：点击 toolbar 中的上传按钮，选择本地的文档，将以 utf8 读入文件。使用 AES 加密后将加密后的数据发送给服务端。通信采用 Https 方式保证安全性

(3) 删除：点击 toolbar 中的删除按钮，将会删除当前选中的文档

(4) 新建：点击 toolbar 中的的新建按钮，将会创建一条新的文档

2.3.2 数据库服务端

服务端为整个系统的核心部分，起到了文件存储，关键词检索与文件删除、更新的作用。服务端采用 python 通用 web 框架 Django 实现，部署在 apache 上，系统架构如图 2-4 所示。

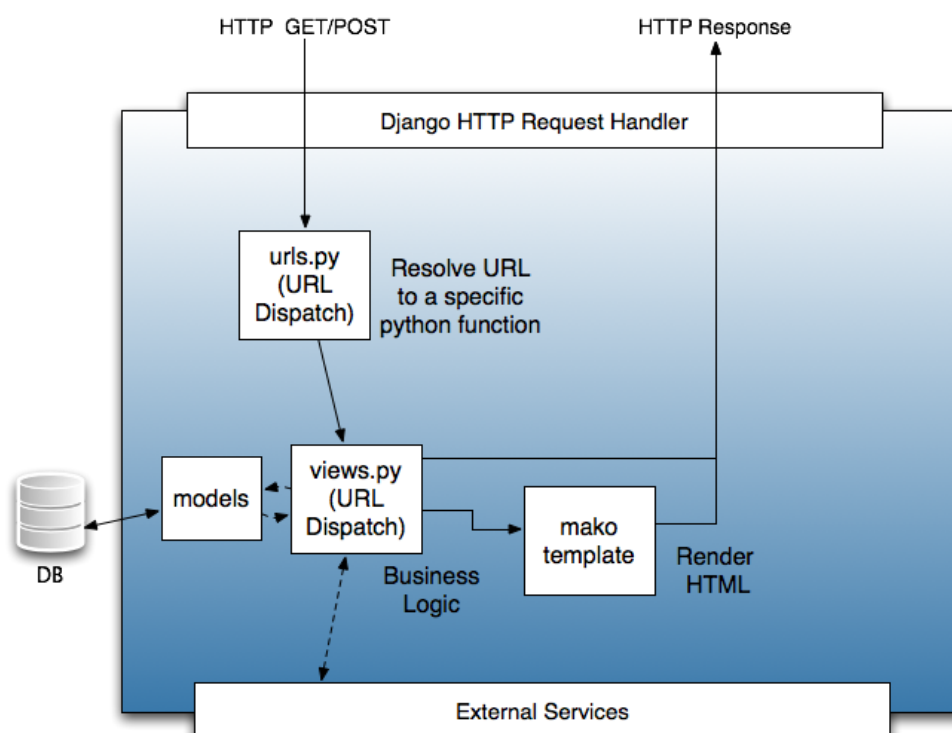


图 2-4 服务端 Django 系统框架

在数据可搜索加密存储部分，我们先后使用 C++和 Neo4j 图数据库实现了服务端关键词搜索功能。为了描述方便，以下简称为 SSE1.0(内存版)和 SSE2.0(图数据库版本)。

客户端向服务端发送一个 HTTP 请求，符合服务端 API 接口的请求经过认证后，服务器正确执行命令，实现本地的密文与加密关键词信息的存储。每篇加密的文章都对应一个 ID，该 ID 是客户端在上传密文时由服务端自动创建的。

SSE1.0 实现了文件存储，关键词检索等全部功能。为了保证关键词平衡树的效率，我们采用 C++实现关键词平衡树类，并对其用 C 接口进行封装，编译成动态链接库。之后，我们用 python 调用动态链接库，并进行进一步封装，得到 Tree 类。该 python 类提供了插入、删除、更新、查询的 API 接口。最后，我们在 Django 的 view.py 中

实例化封装过的 Tree 类，实现了服务端动态可搜索加密过程。

值得一提的是，在 SSE1.0 中我们为了解决索引持久化问题（即防止服务器断电引起的数据丢失），我们引入 SQLite 数据库，即将平衡树的信息和密文/ID 信息分开存储，而平衡树的信息以实时文件的方式存储在服务器上（如图 2-5）。当服务器断电恢复后，SQLite 根据本地存储平衡树的文件信息建立新的映射关系（重建映射关系），这样就保证了索引的持久化，同时并没有完全重建树那么大的开销。

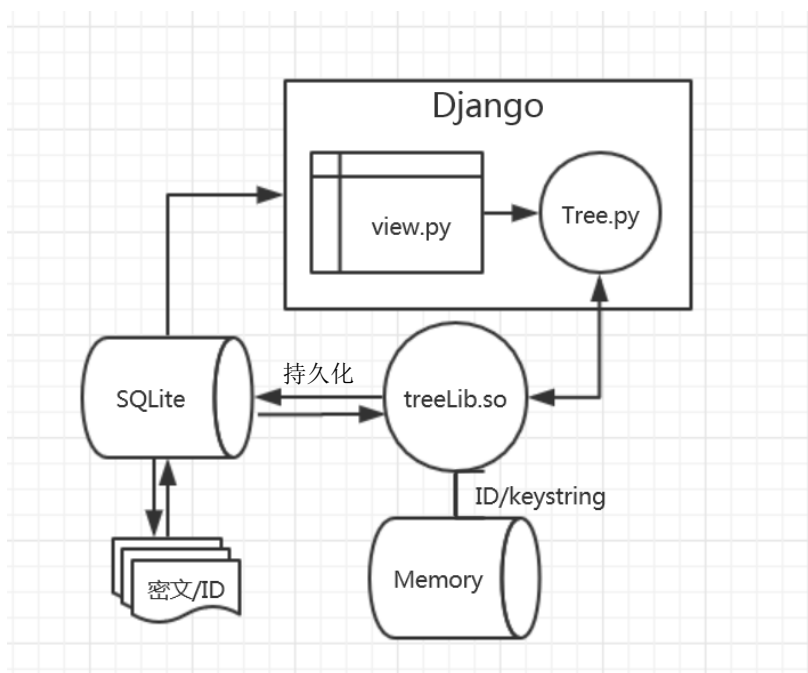


图 2-5 SSE1.0 存储结构图（支持持久化）

为了增加 API 接口的灵活性，我们又拓宽了关键词检索的接口。提供了包括任意数量关键词和排除关键词（即不包含某关键词）检索和确定序列关键词检索的接口。图 2-6 是 SSE1.0 服务端项目文件结构图。表 2-4 是 SSE1.0 一些关键代码文件的作用及接口说明。

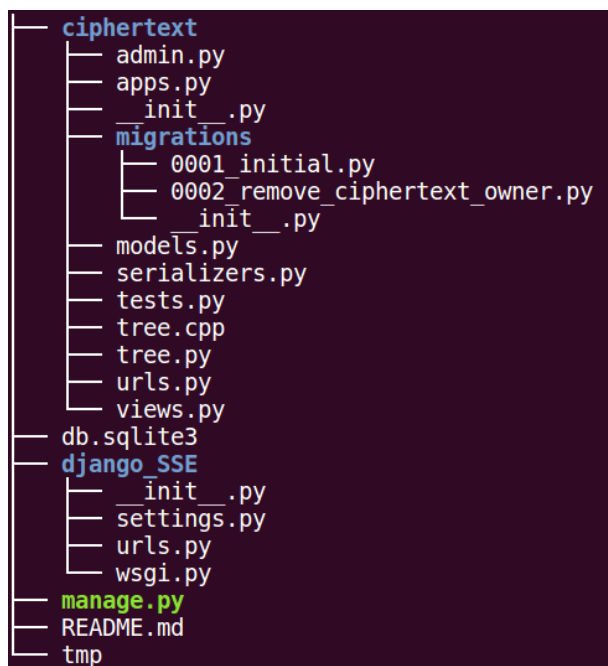


图 2-6 SSE1.0 服务端项目文件结构

表 2-7 关键代码文件功能说明

manage.py	python manage.py runserver 运行主程序, 部署应用
db.sqlite3	SQLite 数据库, 用于存储密文和对应 ID
tree.py	封装过的关键词平衡树类, 提供添加、删除、更新和搜索 API
tree.cpp	实现关键词平衡树功能
treeLib.so	使用 g++编译生成的 python 可调用的动态链接库
models.py	Django 框架下的数据库的定义文件
serializers.py	对数据库 model 序列化
views.py	实现服务端 API, 处理 POST, DELETE, PUT 和 DELETE 请求
urls.py	定义服务端 API 对应的 URL
migrations/	数据库迁移记录信息
tmp	用于存储关键词平衡树和 SQLite 映射信息, 用于持久化

SSE1.0 的缺陷主要有以下几点:

1. 索引可以动态更新, 但关键词不能动态增加删除
2. 虽然在一定程度上实现了索引的持久化, 但需要额外存储映射信息, 在断电恢复后重建映射关系, 影响性能。
3. 面对海量大数据和稀疏关键词 (而现实中文档关键词往往是稀疏的) 的性能较

差，占用空间较多。

于是我们在第二轮迭代后实现了 SSE2.0（图数据库版本）。SSE2.0 实现了关键词的动态修改，同时支持真正的索引持久化（由于图的信息就存储在图数据库中），通过 Neo4j 的高效性和可扩展性来保证对海量数据的搜索性能。Neo4j 支持分布式并行搜索，但我们只在两台服务器上实现并测试了 SSE2.0 版本，验证了系统的有效性和可行性。

图 2-7 是 SSE1.0 和 SSE2.0 在平衡树实现上的区别，同时 SSE2.0 平衡树直接使用 Neo4j 的 Node 和 Relationship 实现。由于 SSE2.0 在平衡树摒弃了加密关键词对应列表，而直接存储关键词令牌的 hash 值，因此占用空间更少（对于稀疏关键词文章更为明显）。同时也解决了动态加入删除关键词的问题，使得系统更加灵活。

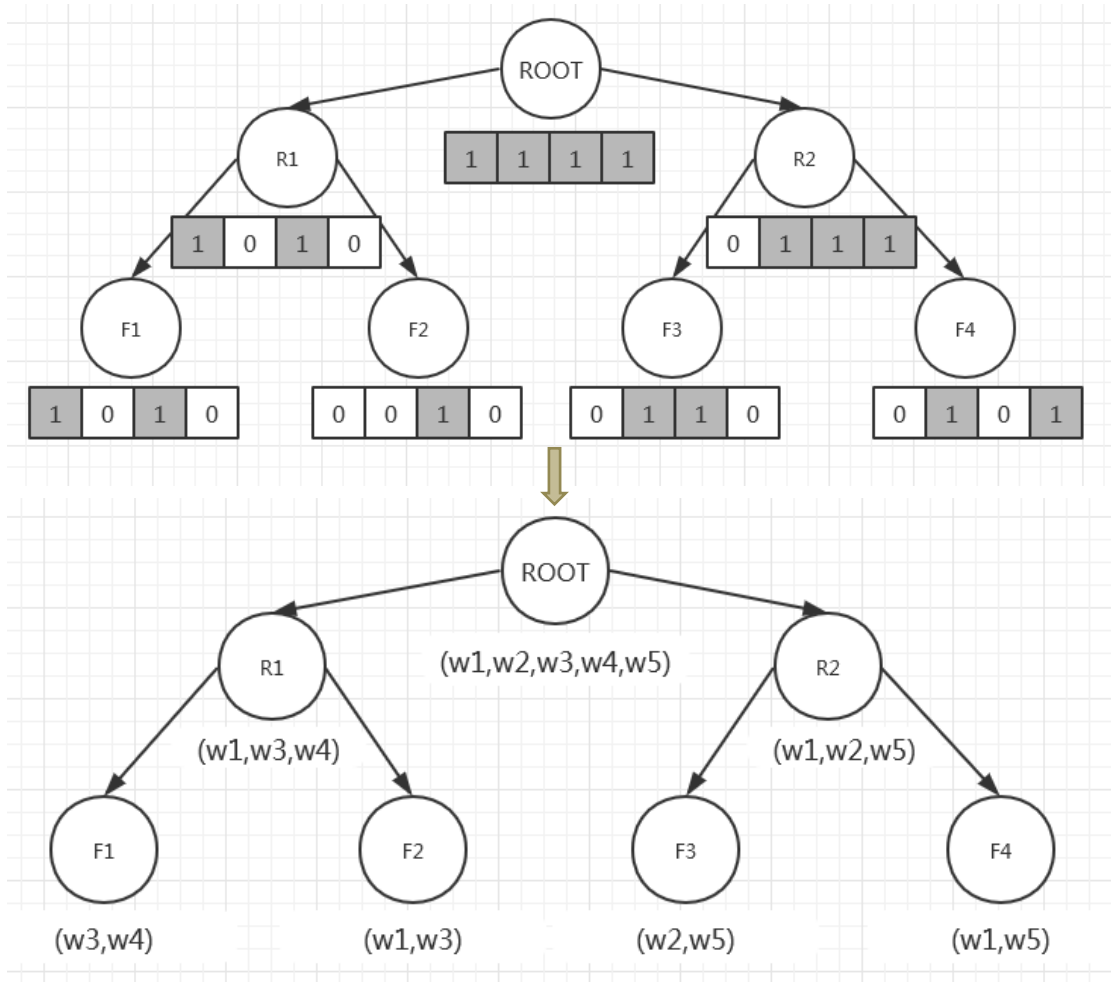


图 2-7 SSE2.0 与 SSE1.0 在关键词平衡树实现的差异

以动态插入关键词为例，图 2-8 解释了图数据库动态增删关键词的步骤。插入或删除关键词，只需要更新文件叶子节点的所有祖先节点的关键词信息即可。

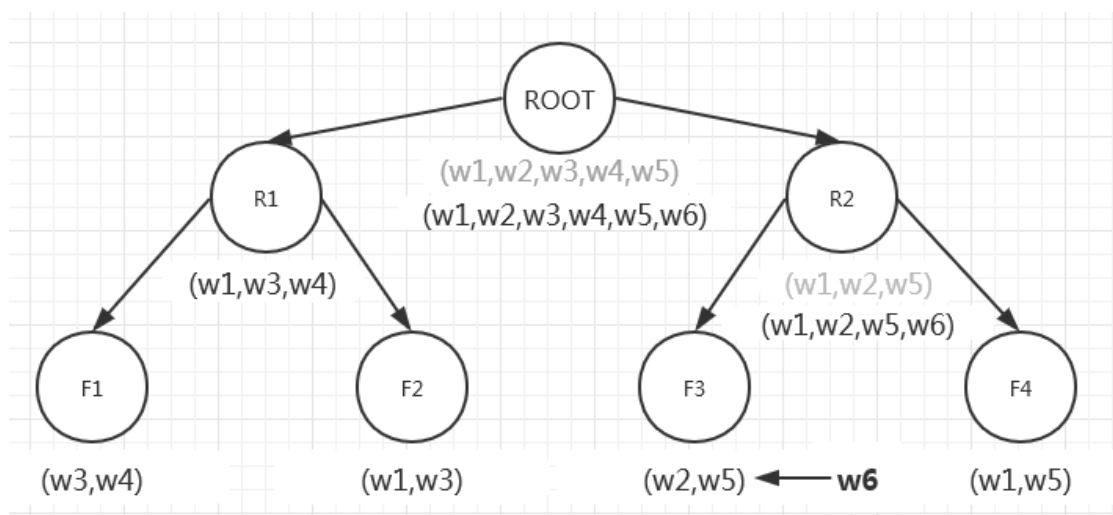


图 2-8 添加关键词时关键词平衡树更新

目前，Neo4j 可以很好地与 Django 连接。通过安装 Django Neomodel 第三方库，正确配置后可以在 Django 的 model.py 文件中实现数据库的定义，完成数据库连接。SSE2.0 中 Neo4j 直接取代 SSE1.0 版本的 SQLite 和 treeLib.so，实现了动态可搜索加密过程。图 2-9 为 SSE2.0 版本的系统结构示意图，而图 2-10 和图 2-11 为一个具有八个文件叶子节点的 Neo4j 可视化示意图（为了便于观察，我们在示意图中并未对关键词进行加密，真实系统中应为加密关键词后的 hash 值）。

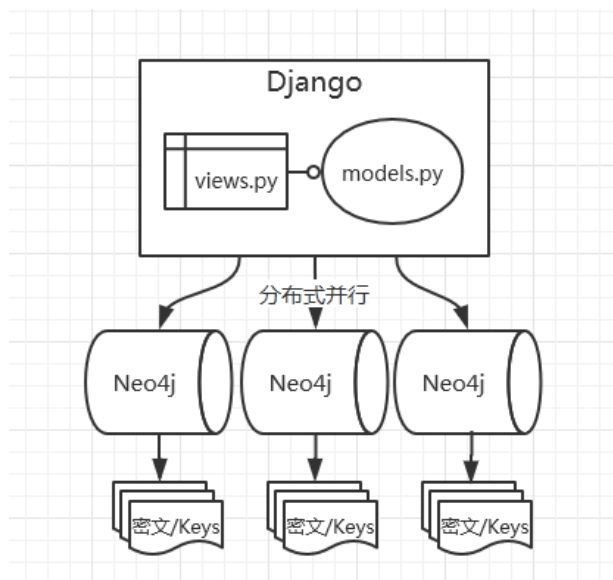


图 2-9 基于 Neo4j 数据库的服务端系统构架

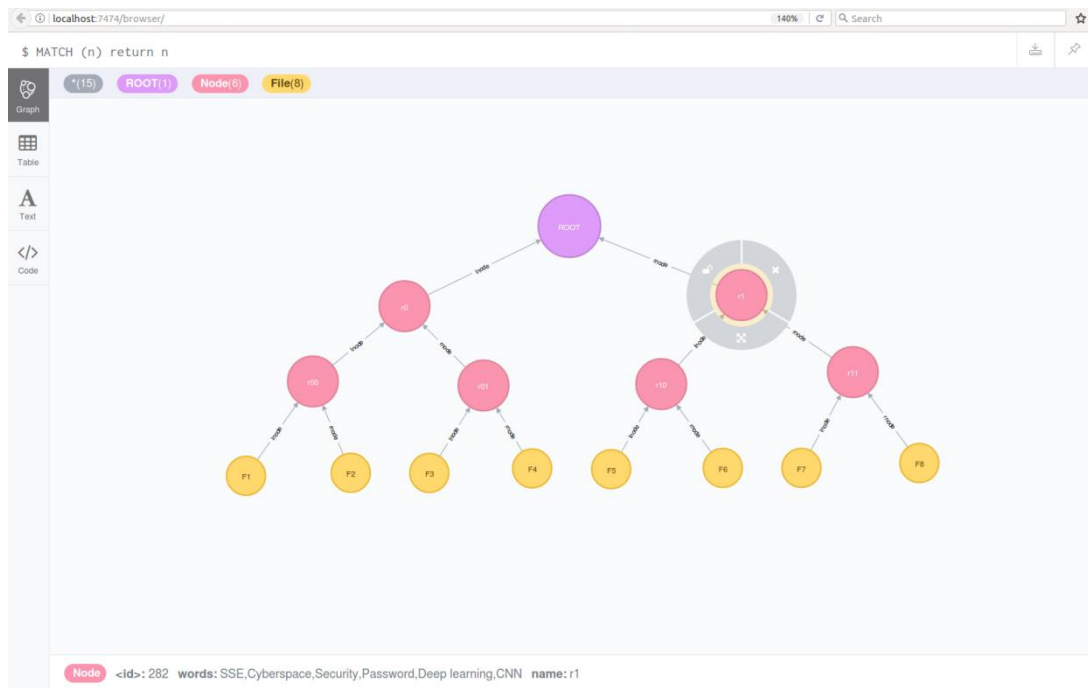


图 2-10Neo4j 可视化界面

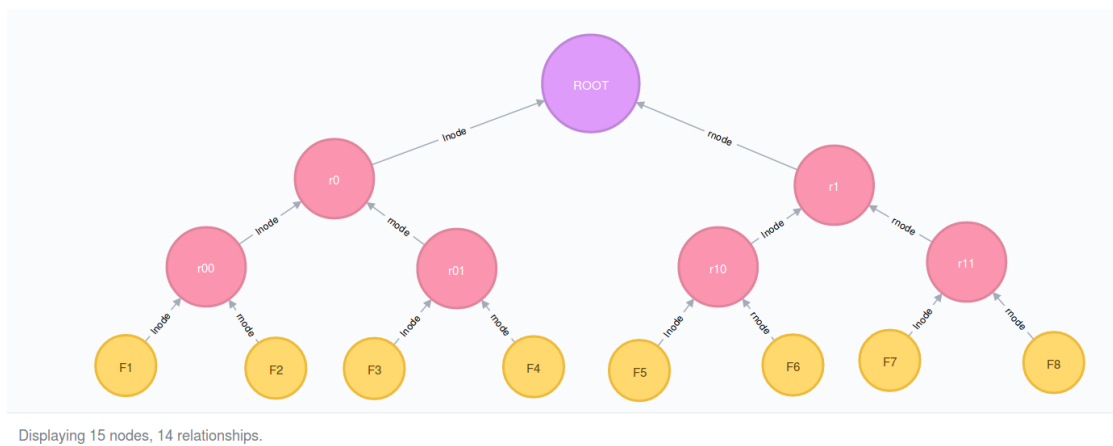


图 2-11 八个文件叶子节点的节点和关系

第三章 作品测试与分析

本项目针对动态可搜索加密共实现 SSE1.0 和 SSE2.0 两个版本，分别实现了单机内存版本和分布式图数据库版本。本章围绕 SE1.0 和 SSE2.0 具体测试内容和性能对比展开。

为了比较 SSE1.0 和 SSE2.0 两个系统的性能，我们分别构建了两套测试实验系统。我们针对索引构建，添加文件，删除文件，查询文件进行了性能的测试比较。测试结果表明 SSE2.0 版本在索引的构建，添加和删除文件方面性能略低于 SSE1.0，但在文件关键词查询方面体现了很大的性能提升（提升一倍以上）。

对于 SSE2.0 版本，我们利用三台服务器实现分布式图数据库支持。除与 SSE1.0 对比测试外，我们又额外进行了海量数据的测试，压力稳定性测试。验证了该版本的有效性、稳定性和可拓展性。

在测试系统的过程中，为了避免核心功能外的操作对时间的影响，我们缺省了一些非核心的耗时操作，如客户端将文件加密后需要发送密文集到服务器的网络传输开销等。

3.1 测试环境

在测试过程中，由于实际情况，我们在客户端使用 Windows，服务器端使用 Linux 进行测试。但实际上，SSE1.0 与 SSE2.0 均无对操作系统的依赖。只要在对系统上配置对应版本的 python，Django，neo4j，react 以及其他依赖包等，系统均可以正常运行。

3.1.1. 客户端：

表 3-1 客户端测试设备参数

项目	配置参数值
操作系统	Windows7 Professional 64 位
CPU	Intel(R) Core(TM) i7-3630QM CPU @2.40GHz

内存	8.00GB DDR_3
磁盘	500G 7200r
网络连接	以太网

3.1.2. 服务器端：

表 3-2 SSE1.0（内存单机）测试设备参数

项目	配置参数值
操作系统	Debian8.2 64 位
CPU	Intel(R) Core(TM) i5-4200H CPU @2.80GHz 2.79GHz
内存	4.00GB DDR_3
磁盘	500G 7200r
网络连接	以太网

表 3-3 SSE2.0（图数据库分布式）测试设备参数

项目	配置参数值
操作系统	Ubuntu14.04 Server
CPU	Intel(R) Core(TM) i7-4500H CPU @1.80GHz
内存	4.00GB DDR_3 x3
磁盘	500G 7200r x3
网络连接	以太网

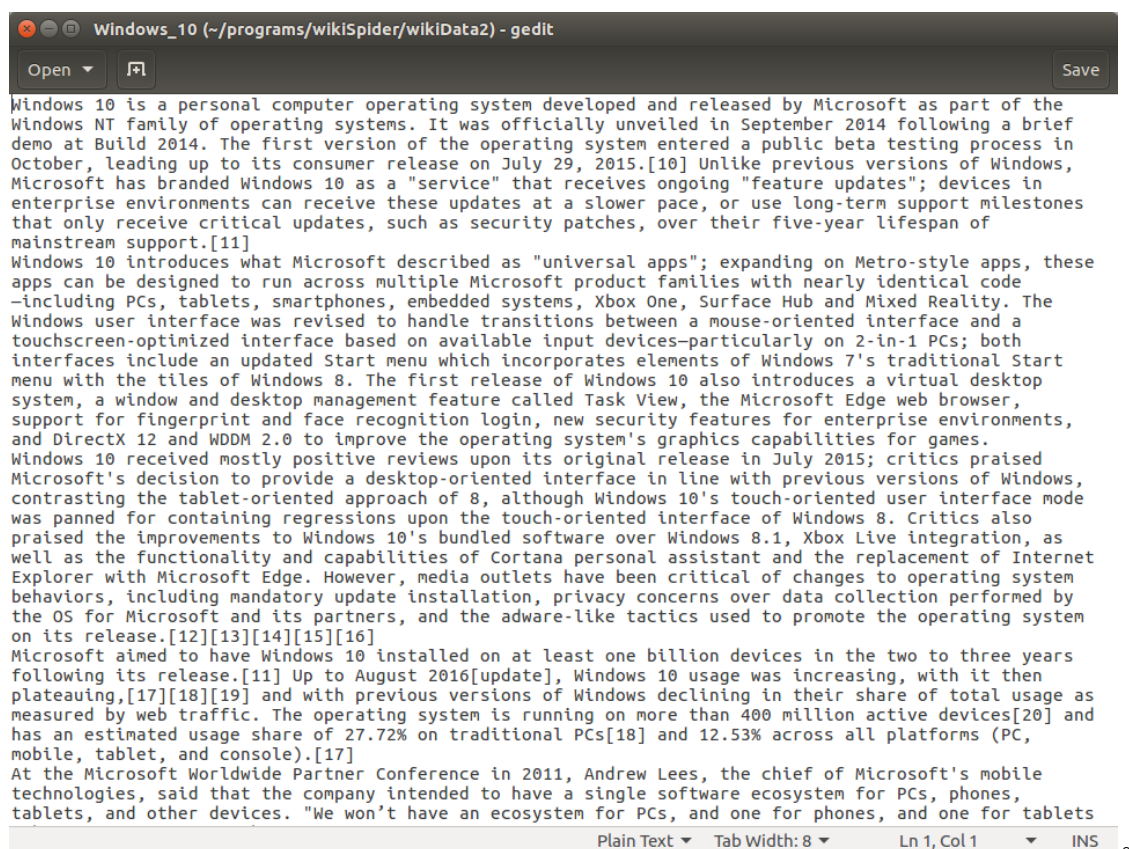
3.2 实验数据

实验中所用的数据集合为 Wikipedia 关键词数据集（英文）和 Baidu 关键词数据集（中文）。这两个数据集均是我们因测试需要从网络上获取后整理得到的。

由于维基百科和百度百科天然的关键词搜索结构，我们很容易的利用网络爬虫获取了相关的关键词和对应的纯文本文章，非常适合统计等。之后我们又重新对所获得文章进行文本处理和统计得到了最终的关键词数据集。其中 Wikipedia 关键词数据集

共有文件 607, 213 个文件, 总大小 13774. 5MB, 而 Baidu 关键词数据集共有文件 10, 534 个文件, 总大小 543. 3MB。

图 3-1 是 Wikipedia 关键词数据集中的实例文件。而表 3-4 则是我们将两个数据集结合后随机抽取的测试文件大小以及对应索引大小。文件数越多, 文件集合越大, 加密时间越长, 形成关键词索引大小也越大。而由于 SSE1.0 需要索引需要存储不相干关键词信息, 故索引较 SSE2.0 更大



The image shows a screenshot of a text editor window titled "Windows_10 (~/programs/wikiSpider/wikiData2) - gedit". The window contains a snippet of text from a Wikipedia article about Windows 10. The text describes the operating system's development, release, and features. The editor interface includes a menu bar with "Open" and "Save" buttons, and a status bar at the bottom showing "Plain Text", "Tab Width: 8", "Ln 1, Col 1", and "INS".

```
Windows 10 is a personal computer operating system developed and released by Microsoft as part of the Windows NT family of operating systems. It was officially unveiled in September 2014 following a brief demo at Build 2014. The first version of the operating system entered a public beta testing process in October, leading up to its consumer release on July 29, 2015.[10] Unlike previous versions of Windows, Microsoft has branded Windows 10 as a "service" that receives ongoing "feature updates"; devices in enterprise environments can receive these updates at a slower pace, or use long-term support milestones that only receive critical updates, such as security patches, over their five-year lifespan of mainstream support.[11] Windows 10 introduces what Microsoft described as "universal apps"; expanding on Metro-style apps, these apps can be designed to run across multiple Microsoft product families with nearly identical code—including PCs, tablets, smartphones, embedded systems, Xbox One, Surface Hub and Mixed Reality. The Windows user interface was revised to handle transitions between a mouse-oriented interface and a touchscreen-optimized interface based on available input devices—particularly on 2-in-1 PCs; both interfaces include an updated Start menu which incorporates elements of Windows 7's traditional Start menu with the tiles of Windows 8. The first release of Windows 10 also introduces a virtual desktop system, a window and desktop management feature called Task View, the Microsoft Edge web browser, support for fingerprint and face recognition login, new security features for enterprise environments, and DirectX 12 and WDDM 2.0 to improve the operating system's graphics capabilities for games. Windows 10 received mostly positive reviews upon its original release in July 2015; critics praised Microsoft's decision to provide a desktop-oriented interface in line with previous versions of Windows, contrasting the tablet-oriented approach of 8, although Windows 10's touch-oriented user interface mode was panned for containing regressions upon the touch-oriented interface of Windows 8. Critics also praised the improvements to Windows 10's bundled software over Windows 8.1, Xbox Live integration, as well as the functionality and capabilities of Cortana personal assistant and the replacement of Internet Explorer with Microsoft Edge. However, media outlets have been critical of changes to operating system behaviors, including mandatory update installation, privacy concerns over data collection performed by the OS for Microsoft and its partners, and the adware-like tactics used to promote the operating system on its release.[12][13][14][15][16] Microsoft aimed to have Windows 10 installed on at least one billion devices in the two to three years following its release.[11] Up to August 2016[update], Windows 10 usage was increasing, with it then plateauing,[17][18][19] and with previous versions of Windows declining in their share of total usage as measured by web traffic. The operating system is running on more than 400 million active devices[20] and has an estimated usage share of 27.72% on traditional PCs[18] and 12.53% across all platforms (PC, mobile, tablet, and console).[17] At the Microsoft Worldwide Partner Conference in 2011, Andrew Lees, the chief of Microsoft's mobile technologies, said that the company intended to have a single software ecosystem for PCs, phones, tablets, and other devices. "We won't have an ecosystem for PCs, and one for phones, and one for tablets
```

图 3-1 Wikipedia 关键词数据集实例文件

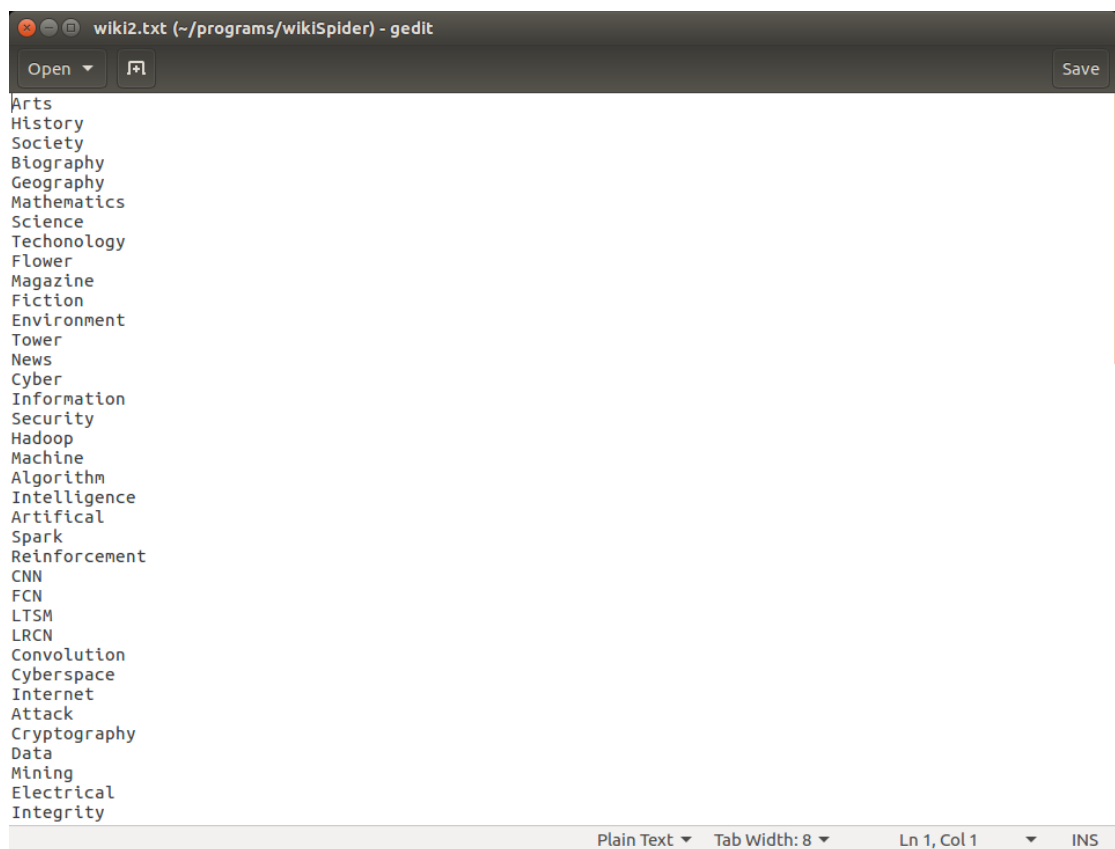


图 3-2 关键词列表文件

表 3-4 测试文件数，文件集大小与索引大小的关系

文件数	文件集大小 (MB)	SSE1.0 索引 (MB)	SSE2.0 索引 (MB)
1000	20.5	56.3	15.7
5,000	100.3	542.3	89.6
10,000	215.2	1015.2	207.2
100,000	2056.2	9845.7	2178.6
500,000	9879.8	46584.6	11253.4

3.3 索引构建的性能实验

模型的第一步为索引构建过程，根据第三章的算法主要拆分为以下几个步骤，首先对文本进行处理并生成关键词平衡树，然后对文本和平衡树做加密，加密后的平衡树被称作索引，最后将密文和索引发送到服务器端，服务器将索引转化为图数据库的

节点结构。

当密文和索引较大时，客户端发送到服务器端的网络通信过程是很耗时的，掩盖了其他部分的性能对比效果，因此在本实验中省略该步骤，改为直接在服务器端做上述操作。

本文所提的模型设计中存在一个全局参数设置，即全局关键词个数 m ，在本实验及以下各个实验中， m 等于 30000。因为根据牛津词典的词条共 80000 余条推断，英语语境下的常见词汇约为 30000。

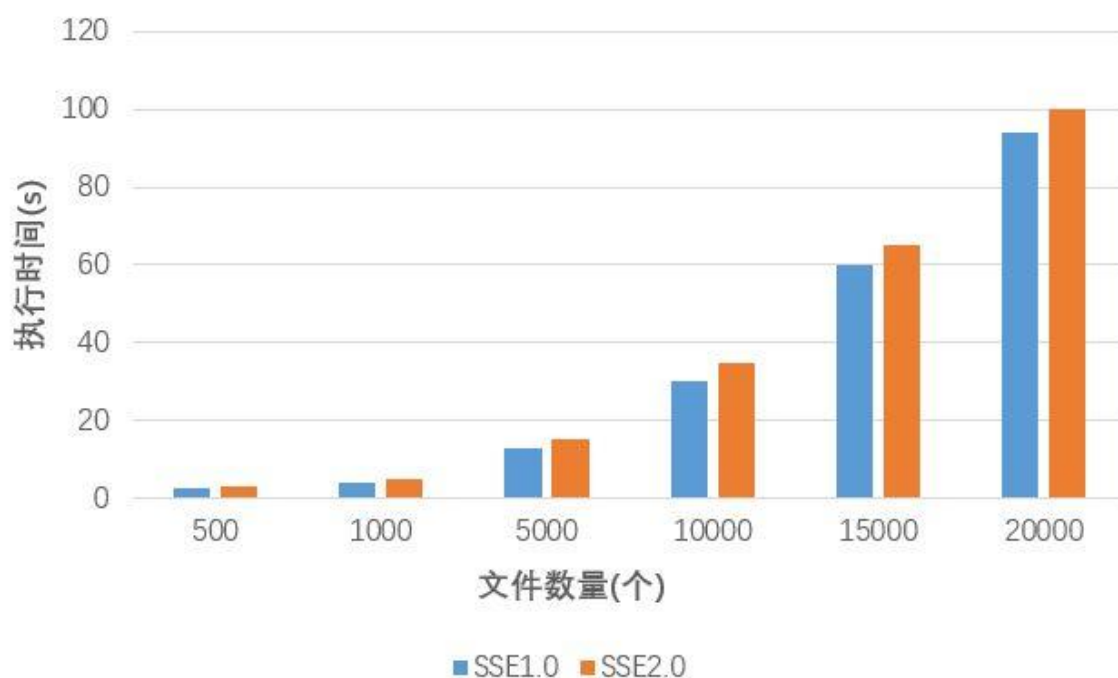


图 3-3 不同模型索引构建时间对比图

图 3-3 对比了图数据库模型和单机模型在索引构建过程中的时间开销，同时可以看到文件数的变化对构建时间的影响。在构建过程中，不仅要对每个文件的内容加密，同时需要对文件的每个关键词加密并计算哈希值确定其映射位置。所以文件个数与加密时间近似正比的同时，每个样本文件含有的关键词个数略有不同造成一定波动，当文件数增多时这些误差会被累积。

图数据库模型在获得内存索引后需要将索引持久化为图的节点和关系，这个过程中主要涉及分布式节点的创建和节点关系的创建。由实验结果可以看出，Neo4j 的这个转化过程是很快的。

3.4 索引更新性能试验

索引更新的主要过程，首先服务器端根据客户端的更新请求，插入或删除一个节点，然后对红黑树做结构调整，将子树结构发送到客户端，客户端计算子树的新内容后将子树发回，服务器端将子树内容拷贝到对应的节点。由此可见，影响索引更新效率的主要因素为被更新子树的大小和客户端计算新内容的效率。

文件个数等于红黑树中叶子节点的个数等于非叶子节点的个数。所以，文件个数决定了索引大小，图 3-4 和图 3-5 对比了在不同文件数下不同模型的插入和删除文件的时间开销情况。先用对应文件数的文件集建立索引，然后对该索引连续执行十次对应操作后取平均值。

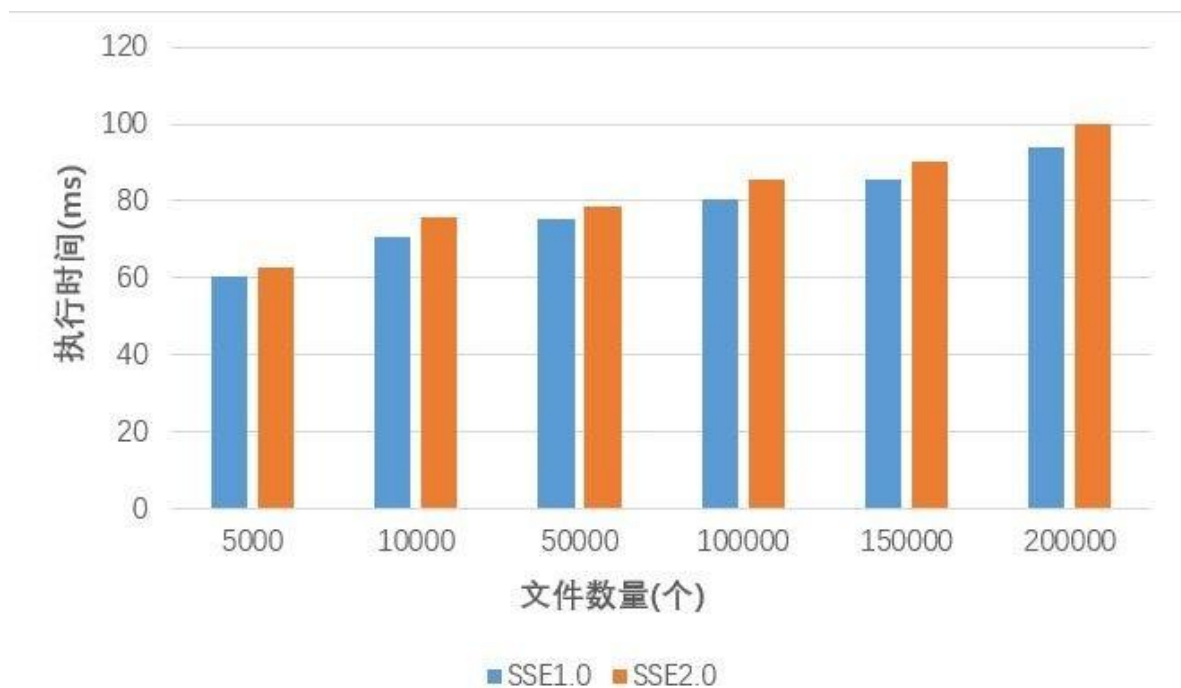


图 3-4 插入文件时间对比图

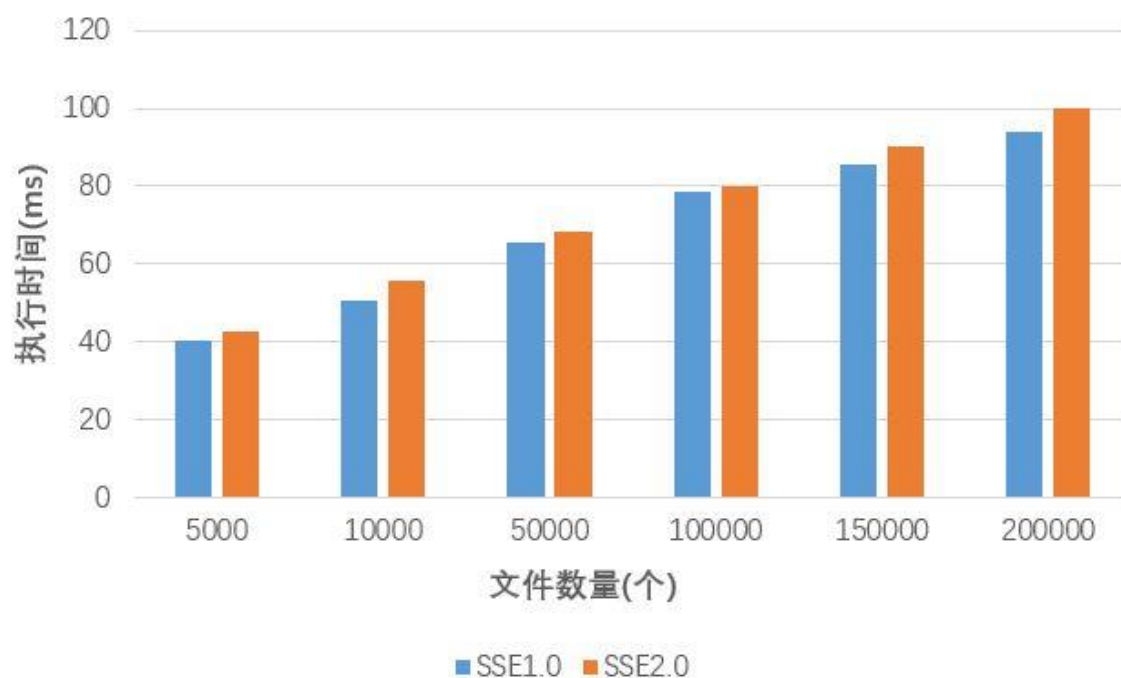


图 3-5 删除文件时间对比图

在一次插入或删除操作中，主要的耗时动作为对树结构的调整以及对子树内容的更新。其中，调整树结构时，图数据库涉及到分布式节点的关系的重新分配和内容改变，这些操作中包含多机器间的网络交互甚至是硬盘读写，这些都会导致额外开销，当索引可以完全加载进内存时，显然是比完全的内存操作要耗时。但是图数据库通过使用缓存等措施，一定程度上改善了这种劣势。

计算新子树内容的过程是另一个主要耗时操作，它与构建一个新子树的操作类似。但是更新过程中，需要重新计算的子树规模很小，根据第三章的复杂度讨论，子树的空间复杂度为 $O(\log n)$

3.5 结果分析

我们本项目针对动态可搜索加密共实现SSE1.0和SSE2.0两个版本，给出了客户端和服务端的具体实现，并针对实验系统做了详细的性能测试。通过单机内存版本和分布式图数据库版本的对比，说明了系统设计的合理性，通过压力测试检验了系统的健壮性和可用性。

第四章 创新性说明

本作品创新性的提出了“基于图数据库的动态可搜索加密”，旨在实现大数据时代云空间隐私保护。从可搜索加密到动态可搜索加密再到基于图数据库的动态可搜索加密一点点改进，进而确保云存储的安全性以及云计算的效率。

作品的几大创新点分别是：在原有树形数据结构的基础上，提出使用图数据库来模拟树状结构，设计了对应的算法模型；设计了动态可搜索加密引擎框架，良好的面向对象设计可以自由拓展其中的加密搜索模型，为模型的实用化做出基础。同时，最后的性能试验证明算法的高效性和并行搜索的巨大优势。本章针对作品的创新性做详细的说明。

4.1 实现了大数据时代云空间隐私保护

随着信息技术的发展，以Web2.0技术为基础的博客、微博、社交网络等新兴服务和物联网以前所未有的发展速度产生了类型繁多的数据，而云计算为数据的存储提供了基础平台，这一切造就了大数据时代的正式到来。

大数据中蕴藏着巨大的价值，是企业的宝贵财富。但大数据同时也带来了巨大的挑战，个人隐私保护问题就是其中之一。本项目通过构建一个安全性高，实用性强的基于图数据库的可搜索加密系统，使得大数据时代云空间隐私保护得到一定的实现^[8]。

4.2 使用图数据库来模拟树状结构

对于一般的数据库，虽然也可以满足索引动态更新，一定程度上实现了索引的持久化，但还是有着种种的不足，比如：关键词不能动态增加删除；一旦需要额外存储映射信息，断电后重建映射关系，影响性能；面对海量大数据和稀疏关键词的性能较差，占用空间较多。所以考虑基于图数据库解决上述问题，通过Neo4j的高效性和可扩展性来保证对海量数据的搜索性能，也实现了关键词的动态修改，同时支持真正的索引持久化，并行化搜索。

4.3 设计了动态可搜索加密引擎框架

当下保证云存储的安全性，文档加密已经成为必然。相比全同态函数和 ORAM 具有较高的通信复杂度或者空间复杂度。可搜索对称加密对可用性和效率做了很好的折中，提供对密文的关键词搜索功能，同时具有较高效率。在一般的可搜索加密的基础上重新定义了更新索引的方法，从而达到自由拓展其中的加密搜索模型的作用，为模型的实用化做出良好的基础。

4.4 平衡树的创新性

我们在原有的替罪羊数据结构的基础上加以改进，创新性的得到了关键词替罪羊树，减少了算法复杂度，使得系统的效率得到提升。

4.5 可搜索加密算法的创新性

我们在产生索引的过程中，提出了随机化索引原则。即在原来确定性算法的基础上，引入概率算法。通过引入两条规则，更新的动态可搜索加密的算法可以不泄露一些信息，包括关键词的数目，搜索得到的文件数目，部分用户查询模式。这相对于传统动态可搜索加密算法在安全性上达到了更高的标准

第五章 总结

本项目研究了可搜索加密技术的构建方法，分析了现有方案的不足。提出了基于图数据库的动态可搜索加密方案，并设计了一套可应用在实践中的可搜索加密编程开发接口，基于该接口开发了实验系统，并对实验系统进行了性能测试和分析，证明了方案设计的可行性和高效性。

本项目的关键技术主要在于以下几点：

1) 关键词平衡树实现；2) 文件AES加密与关键词口令生成；3) 关键词检索；4) 处理HTTP请求；5) 用户认证接口；6) 索引持久化；7) 关键词检索灵活化；8) 客户端界面与后端的实现；9) 客户端与服务端通信模块；10) 动态增删关键词索引；11) 分布式图数据库支持

本章将对于作品的技术难点以及作品的应用前景进行详细的论述

5.1 作品技术难点

5.1.1 基于图数据库的可搜索加密系统

a. 客户端的构建

客户端的前端开发框架采用了的Reactjs。React采用独特的jsx语法，采用模块化（Component）的开发方式，实现了高度可复用性。React采用单向数据流，让每个模块根据数据量自动更新，改善程序的可预测性。React采用虚拟dom的技术，将修改更新到浏览器实际的 DOM 节点上。

前端的数据流管理采用redux。Redux采用有限状态自动机的概念，把web应用视为一个自动机，视图与状态对应。当状态改变时，UI视图随之改变。

b. 服务端的构建

服务端为整个系统的核心部分，起到了文件存储，关键词检索与文件删除、更新的作用。服务端采用 python 通用 web 框架 Django 实现，部署在 apache 上。在数据可搜索加密存储部分，我们先后使用 C++和 Neo4j 图数据库实现了服务端关键词搜索功能。改进后，整个系统基于 Neo4j 图数据库，实现了关键词的动态修改，同时支持

真正的索引持久化，通过 Neo4j 的高效性和可扩展性来保证对海量数据的搜索性能。Neo4j 支持分布式并行搜索，验证了系统的有效性和可行性。

基于整个项目的完成，需要用一种数据结构(平衡树)来实现文章信息的快速插入、删除以及查询。在实现关键词平衡树时，决定使用替罪羊树来实现，替罪羊树相较于红黑树除去了繁杂结点的旋转等问题，更加容易实现。同时替罪羊树也属于平衡树，同样具有较低的算法复杂度。考虑到信息只存储在叶子结点会让通过旋转来自平衡的各类平衡树出现一定的麻烦以及可能带来的性能问题，最终我们决定采用一种不通过旋转来完成自平衡的平衡树——替罪羊树。

替罪羊树插入、删除节点的平均最坏时间复杂度为 $O(\log n)$ (其中 n 为整棵替罪羊树所包含的节点数)，并且从实际测试上来看，替罪羊树由于其常数小的特点，在性能上领先于其他平衡树，与红黑树相差无几。于是将替罪羊树进行改造，实现了重建，插入，删除，查询功能，最终得到了新的改进数据结构——关键词替罪羊树。

5.2 作品应用前景

本项目的主要应用人群为企业用户以及安全性需求较强的个人用户，针对试用人群对云存储安全的高要求性及时效性，项目创新的提出了“动态可搜索加密系统”的思想，解决了索引的动态更新机制还不够成熟，关键词不能动态增加删除的问题，确保了在安全性达到一定级别的情况下，该模型检索和更新效率均优于传统模型，具有一定的现实意义和积极意义。

可搜索加密凭借其诸多优点，已经成为当今云计算中的主要成分，得到改进的 DSSE(动态可搜索加密)更是大幅度的提升了效率，然而还是没有解决索引只能加载在内存中，虽然在一定程度上实现了索引的持久化，但需要额外存储映射信息，所以将可搜索加密系统基于 Neo4j 图数据库中，由于图的信息就存储在图数据库中，从而支持真正的索引持久化。在保证安全性和功能的同时，拓展了其开发性与拓展性，符合当今的发展主流。

参考文献

- [1] 李娟. 浅析云计算安全现状及趋势[J]. 西部资源, 2015, (5):54-56.
- [2] 许文广. 论云数据安全存储技术的发展现状[J]. 电脑迷, 2016, (7)
- [3] Goh, E.-J.: Secure indexes. IACR Cryptology ePrint Archive, 2003:216 (2003)
- [4] Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442-455. Springer, Heidelberg (2005)
- [5] Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. In: Computer and Communications Security (CCS), pp. 79-88 (2006)
- [6] Kamara S, Papamanthou C. Parallel and Dynamic Searchable Symmetric Encryption[M]// Financial Cryptography and Data Security. Springer Berlin Heidelberg, 2013:258-274.
- [7] 田琛, 范磊. 基于图数据库的可搜索加密[J]. 计算机科学与应用, 2016, 6(12):778-785.
- [8] 刘雅辉, 张铁赢, 靳小龙, 程学旗. 大数据时代的个人隐私保护[J]. 计算机研究与发展, 2015, 52(1):229-247