论文独创性声明

本论文是我个人在导师指导下进行的研究工作及取得的研究成果．论文中除了特别加以标注和致谢的地方外．不包含其他人或其它机构已经发表或撰写过的研究成果．其他同志对本研究的启发和所做的贡献均已在论文中作了明确的声明并表示了谢意．

作者签名：谢涛　　日期：2007.6.6

论文使用授权声明

本人完全了解复旦大学有关保留、使用学位论文的规定．即：学校有权保留送交论文的复印件．允许论文被查阅和借阅；学校可以公布论文的全部或部分内容．可以采用影印、缩印或其它复制手段保存论文．保密的论文在解密后遵守此规定．

作者签名：谢涛　　导师签名：周东华　　日期：2007.6.8

摘 要

XML (eXtensible Markup Language)已逐渐成为互联网信息的主要表示和交换标准。为了有效地分析和处理 XML 数据，研究者们已经提出了各种 XML 数据处理技术，其中 XML 的存储、索引、查询等受到了广泛的关注。在 XML 的存储技术方面，原生 XML 数据库由于可按原结构不经任何转换就可存储 XML 数据，从而实现高效存储和无数据损失，并且可以通过精心设计，实现比在关系数据库或文件系统存储的文档更快的查询处理等，因此原生 XML 数据库在管理 XML 数据方面有其独特的优势。

路径表达式查询（如 XPath）是 XML 上查询语言的重要组成部分，除此之外，XML 上关键词查询技术由于查询方式简单，用户无需了解数据内在格式，是近来研究的热点。另外，XML 数据上基于结构相似度的查询由于有其实际的应用场景，也是一个重要的研究课题。因此，在原生 XML 数据库中实现上述多种查询方式，具有较大的实际意义。

本文对 XML 数据的存储和检索技术进行了研究，主要贡献和创新之处在于：

1）XML 数据存储技术的研究，设计实现了一个 XML 数据存储的原型系统 Sheepdog，提供了有效的数据存储，支持多种索引技术，能够有效地处理 XPath 查询。

2）提出了一种更为快速的 XML 数据上关键词检索的技术，该技术在求解树结点的共同祖先问题时采用范围最小值查询算法。此技术应用在上述 XML 数据存储系统 Sheepdog 中，并和现有的关键词查询技术 XKSearch 做了比较，证明了其优越性。

3) 研究了 XML 上的结构相似度检索，提出了一种快速的算法进行相似 XML 文档的查找，也实现在上述 XML 存储系统 Sheepdog 中。

本文研究 XML 数据的存储、检索技术，更详细的说，是对 XML 数据的原生存储方式和对 XML 数据上的关键词检索技术和结构相似度检索进行了深入的探讨和研究，提出了有效的算法和新的技术，并且实现了原型系统，通过实验证明了本文所提出方法的有效性。研究成果将可直接用于 XML 数据库的项目开发和产品研制中，具有重要的理论和现实意义。

关键词：XML　原生 XML 数据存储　关键词检索 结构相似度检索

Abstract

XML (eXtensible Markup Language) has been becoming the de facto standard for information representation and exchange on the Web. Researchers have proposed much XML storage, quering, indexing techniques to process XML data. As for the storage of XML data, native XML storage can store XML documents without any transformation, thus having good efficiency and no data lost. What's more, well designed and implemented query processing engine makes search in native xml database more efficient than in relational database or file system. So native XML databases are more suited to manage XML data。

Path expression query, e.g., XPath, is the core of XML query languages. Besides this, keyword search is a hot research topic as this query style is simple and does not force users to learn the inner schema of their data. Also, structure similarity search on XML data is becoming an important topic because it has real scenarioes. So, it has great significance to implement above query styles in native XML databases.

This dissertation studies the issues of storage and retrieval of XML data. The main contributions of this dissertation can be summarized as following:

1) Study native XML storage strategies. Design and implement a native XML database prototype named Sheepdog which exploits efficient data storage and index techniques to support fast XPath processing.

2) Propose a fast keyword search technique for XML documents. This technique exploits the Range Minimum Query algorithm when answering Lowest Common Ancestor problems of tree nodes. This technique is implemented in our Sheepdog system and comparison with exsiting technique XKSearch shows its efficiency.

3) Study the structure similarity search problem on XML. Propose a fast algorithm to find structure similar XML documents. This technique is also implemented in our native XML database Sheepdog.

This dissertation studies XML storage and retrieval techniques. Concretely, it focuses on native XML storage, keyword search and structure similarity search on XML. It proposes efficient algorithms and implements them in our prototype Sheepdog. Through experiments, the algorithms are proved to be correct and effective. The study results can be directly applied to XML database products and thus has important academic and practical meanings.

Keywords: XML, native XML Storage, Keyword Search, Structure Similarity Search

第一章 绪论

1.1 背景介绍

1970年，IBM公司的研究员E.F.Codd发表了题为《大型数据库的数据关系模型》[Cod70]著名文章。从此，数据库系统的发展进入了“关系型数据库系统(relational database)”时期。关系型数据库系统以关系代数为坚实的理论基础，经过几十年的发展和实际应用，技术越来越成熟和完善。其代表产品有Oracle、IBM公司的DB2、微软公司的MS SQL Server以及Informix、ADABASD等等。在传统的以关系模型为基石的关系数据库中，人们对数据库的直观印象是一张张结构完整的表格。随着应用需求的发展，关系型数据库系统虽然技术很成熟，但其局限性也是显而易见的：它能很好地处理所谓的“表格型数据”，却对技术界出现的越来越多的复杂类型的数据（例如：XML数据）无能为力。

XML是可扩展标记语言（eXtensible Markup Language）的简称，它自产生就扮演着越来越重要的角色，事实上已经成为数据交换的标准、SOA架构的基石，比如微软Office 2007中就将以XML格式存储文档。作为数据存储的格式，XML具有许多优点：

1）XML简单，自我描述而易于解析。使得XML具有机器可读性，适于电子数据交换（EDI）和处理。同时，XML/EDI模式极大地降低了EDI交换的费用。

2）XML实现了内容，结构和表现三者的分离。文档类型定义（DTD）描述了文档中元素和子元素间的嵌套结构，而不同的用户可以通过XSL按不同显示方式显示全部或部分的文档内容。

3）各种格式的数据都可以容易地转化为XML数据，使得XML非常适于Web信息发布和集成。

由于具有以上优点，学术和研究机构纷纷采用XML来表示各种科学数据，并展开了对XML的深入研究。各个行业如金融机构、海关、媒体产业正制订各自行业的XML DTD（Document Type Definition，文档类型定义），以利于数据以公认的格式交换和集成。国外已形成了XML的一系列标准，如BizTalk、ebXML。在我国，中国科学院电子商务研究中心也联合国内软件厂商制定了cnXML标准。INTERNET上已经涌现了大量的XML页面、站点和应用开发工具。XML已成为Web信息发布和交换的事实上的标准。XML在电子信息发布、电子商务（EDI交换）、数字图书馆、Web信息搜索和集成等领域具有广阔的应用前景。总之，

对 XML 的深入研究将有力促进企业的信息化和电子商务，具有巨大的应用前景和经济效益,而传统的关系数据库不能很好的处理 XML 数据，也促进了 XML 数据库的蓬勃发展。

1.2 研究现状与不足

虽然 XML 格式的数据应用越来越广泛，却给企业数据管理系统带来了不小的麻烦,其原因就在于 XML 数据模型与传统的关系模型之间存在着较大的区别：关系模型是以关系（表）、属性（列）为基础，而 XML 数据模型则是以节点（元素、属性、备注等）和节点间存在着的相互关系为基础的。在管理和处理 XML 数据上，按照存储方式的不同，基本分为四种类型: 文件系统、半结构化数据仓库、数据库管理系统和原生 XML 数据库，如图 1.1 所示。

使用文件系统来存储和检索 XML 数据是管理 XML 的最直接的方式。基于文件的 XML 系统将 XML 直接存储为文本文件，由于文件系统本身不具备查询处理 XML 数据的能力，在处理查询时需要将 XML 文档解析（parse）为内存中的 DOM 树结构。基于文件的 XML 系统简单而容易实现，无需使用底层的数据库或对象存储管理，同时，由于 XML 文档被直接存储为文本文件，这种存储方式无需存储转换和重构查询结果。然而，这种存储方式在查询处理方面有明显的弱点，首先，该方法在每次浏览和查询文档时都要重复地解析文档。其次，整个文档在查询处理过程中都要驻留内存。尽管我们可以在内存中为文档建立索引，通过索引来定位查询所需的部分，维护这种索引的代价仍十分昂贵。

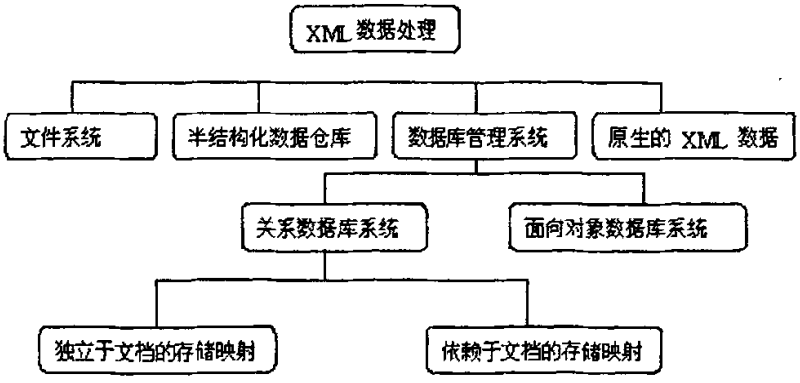


图 1.1 处理 XML 数据的基本方式

由于 XML 数据与半结构化数据十分相似，利用半结构化数据仓库来管理 XML 数据似乎是一种比较自然的方式。在这种方法中，XML 数据被聚簇存储为有向图。斯坦福大学的 Lore[QWG+96] 项目在这方面作了初步的尝试，然而，

当前的半结构化数据库技术尚不成熟，利用半结构化数据仓库（如：Lore）处理XML 查询的性能仍然难以让人满意。

剩下的两种方法就是发展 XML 数据库。第一种方法是使用数据库管理系统来管理 XML 数据，这样的数据库系统称为使能 XML 数据库（Enable XML Database）。根据数据库管理系统的不同，这种存储方式又可以分为基于关系的 XML 数据库系统和基于面向对象的 XML 数据库系统。在这两种方式中，由于当前的面向对象数据库系统的性能仍不足以支持对大规模数据的复杂查询，基于关系数据库的 XML 存储管理是一种更有前景的方式。按照将 XML 数据转存为关系的映射方式的不同，我们又可以将基于关系的 XML 数据存储分为两类：第一类方法将 XML 文档树中的节点和边映射为关系模式，得到的存储模式与 XML 文档的结构无关，我们将其称作独立于文档（document-independent ）的关系存储；第二类方法根据 XML 的结构（DTD 或 XML Schema ）来生成关系模式，不同的文档具有不同的存储模式，我们将其称作依赖于文档（document-dependent ）的关系存储，也分别称为基于模型（DOM ）的存储和基于结构的存储方法。利用数据库系统，特别是关系数据库系统，来处理 XML 数据的方式具有如下的优点：一方面，当前的关系数据库的技术已十分成熟，商用的关系数据库系统都具有高性能的查询引擎，良好的可扩展性、安全性和健壮性，因此，利用关系数据库系统管理 XML 数据可以重用数据库的查询优化器和事务处理机制，能够保证 XML 数据的一致性和完整性；另一方面，目前大量的 WEB 数据主要存放在关系数据库中，XML-关系系统便于在关系数据库上建立适于二者的应用，使关系数据库进入 Web 领域成为可能。在不变动关系型数据库内核层的基础上，将 XML 的树型结构数据拆散、重组转换成关系型表格数据存入数据库。在提取 XML 数据时，利用 SQL 语言的优化将库内的表格型数据取出并还原成 XML 结构型数据。文献[WBD+00]详细讨论了有关表格型数据与 XML 数据的转换及优化的问题。像甲骨文（Oracle）、微软等数据库厂商都在走这条路线，在其成熟的关系数据库产品中提供对 XML 的支持，如 ORACLE 9i 中的 XML SQL Utility，MICROSOFT SQL SERVER 2000 中的 XML and INTERNET Support,，IBM DB2 中的 XML Extender 等。缺点则是：关系数据库的存储模型并不适合 XML 文档，将 XML 文档分开放到关系表中，或者直接将其看作一个大的 blob，都比较耗时而且两种办法都不能建立索引和进行快速查询；实际上，分解文档会造成表格数量过多和大量空值(null)字段，导致浪费空间，通常还会造成细节损失，如元素顺序、处理指令、注释、空白和其他在很多应用程序中非常重要的成份，正是这些因素，首先使 XML 文档看起来不像是序列化的表。字段和记录的边界和 XML 文档的边界不匹配。

第二种方法，也是被业界普遍认为是代表发展方向的方法，就是发展“原生XML 数据库系统(Native XML Database)”。这一概念由德国软件股份公司(Software AG)首次提出并实施于其新型数据库 Tamino[URL3]之中。在这一数据库系统中，从数据库核心层直至其查询语言都采用与 XML 直接配套的技术。由于专门针对 XML 设计的存储模型和查询实现，使得原生 XML 数据库理所当然地更适合管理 XML 数据。它具有以下优点。

1）存取速度。根据 XML 数据库存储数据的物理方式的不同，数据的读出速度可以做到比关系型数据库快得多。其原因是，原生 XML 数据库对整个文件一起进行物理存储，和文件各个部分的物理(而不是逻辑)指针可采用同一存储策略。这就可以不使用连接(joins)或只使用物理连接读取文件，无论哪种情况都比关系型数据库所用的逻辑联结要快。以一个销售订单文件为例。在关系型数据库中，它可能被存为四个表格，SalesOrders, Items, Customers, 和 Parts。读取文件时需要将这些表格结合起来。在原生 XML 数据库中，整个文件可被存储在磁盘的一个地方，在读取文件或其片断时只需要一次查找和一次读取操作。关系数据库在读取数据时则需要四次查找以及至少四次读取操作。

2）查询性能。原生数据库可以建立各种索引提高操作的速度，比如，可以维护一个表来保存文档中所有的 ID 值，这样就可以直接跳到具有特定 ID 的元素而不必遍历树来查找，一种非数据库工具 Jaxen XPath 引擎[URL4]就是这样做的。数据库可以为每个节点分配顺序号，这样就知道每个节点的位置，可以在常数时间内比较两个节点的文档顺序。其次，数据库在存储的时候可以对每个文档进行基本的预解析。这样就不需要检查查询要访问的每个文档的结构良好性，或者构建表示那个文档的对象模型。所有这些细节都以查询引擎能够使用的方式保存在数据库内部。XML 数据库还使用其他大量的技巧(如查询改写)来优化性能。

3) 数据完整性。一些（不是全部）原生 XML 数据库的优点是可以保证数据无损。即能够原样（甚至逐字符甚至逐字节）还原存储的文档。这种功能对于涉及到法律问题的情况非常重要，因为需要重新生成完全一致的原始文档，包括每一个字节。这种功能对于软件开发可能也很重要，特别是对于 bug 跟踪和性能优化。这种情况下，似乎无关的细节有时候非常重要。一定要保证数据库不会改变 10 MB 文档中的可能真正引起问题的两个字节。基于解析器的解决方案，包括存储到关系数据库之前分解 XML 文档的系统，往往会损失一些东西，如标签中的空白、数值字符引用和其他通常无关紧要的细节。对于这种需求，就需要寻求一种能够保留这些细节的原生数据库。

原生 XML 数据库有以上关系型数据库所不具备的优点，因此被认为是支持 XML 的数据库的一个发展方向，这方面的研究也成为热点，表 1.1 是一些现有

原生 XML 数据库项目。但现有的原生 XML 数据库同 20 世纪 90 代的关系数据库产品一样尚存很多缺陷（当时关系数据库运行很慢，bug 成堆，没有标准，而且占用大量内存），一些产品，在支持存储的数据量上、以及存储的效率上差强人意典型的，像 Xindince[URL5]，只能支持较小的文档，Berkeley DB XML[URL7]则对磁盘的开销较大。另外，现有产品的全文检索能力不高，而且都不支持结构相似度的检索。

表 1.1 部分原生 XML 数据管理系统研究项目

项目名称	研究机构	主要研究内容/实现特点	支持的查询
Berkeley DB XML 开源软件	Sleepycat	（1）它是一个库而不是一个数据库服务器，它与应用程序运行在同一个进程； （2）支持大用户量的并发访问，存储容量可达 256TB； （3）支持 XML Schema 验证； （4）对外提供编程接口 API 支持 C++、Java、Perl、Python、PHP、Tcl、Ruby 等语言； （5）开放源代码、底层采用 C/C++语言实现； （6）支持操作系统包括 Windows、Linux、BSD、UNIX、Mac OS/X 和任何 POSIX 兼容的操作系统； （7）提供命令行方式与数据库进行交互操作。	XQuery 1.0 XPath 2.0
Xindice 开源软件	Apache	（1）完全用 Java 语言写成，因此需要 JDK 的支持； （2）支持 DOM 和 SAX 编程接口； （3）仅支持 well formed 的 XML 文档，不支持 Schema 验证。 （4）提供 Java 语言的 XML:DB API 接口和其它语言的 XML-RPC 接口； （5）提供命令行管理工具。 （6）不支持 XML 校验和不能处理较大的 XML 文档（1M）	XPath 1.0
eXist 开源软件	Wolfgang Meier	（1）完全用 Java 语言实现； （2）不支持 XML Schema 模式验证； （3）支持可插拔的存储后端，即后端既可以纯 XML 数据库也可以是关系数据库； （4）支持 HTTP、XML-RPC、SOAP 和 WebDAV 接口访问和 Java 专用的 XML：DB API；	XQuery 1.0
Tamino 商业软件	Software AG	（1）支持 HTTP 方法 GET、PUT、DELETE 和 HEAD 读取文档、存储或替换文档、删除文档那个和取得文档的有关信息，不支持文档内容更新； （2）提供 Java、ActiveX、JavaScript 和.net 等语言的 API 访问； （3）支持 XML Schema 模式验证； （4）也可存储其他类型的对象，如图像、声音文件、Word 文档、HTML 页面文件等； （5）方便与其他数据库进行集成与数据转化； （6）提供图形界面和命令行方式的交互。	XQuery 1.0

具体的概念在下一章中介绍，关于关系数据库和 XML 数据库的简单对比如表 1.2。

表 1.2 关系数据库和 XML 数据库的比较

关系数据库	XML 数据库
关系数据库是关系表（table）的集合	XML 数据库是元素（Element）的集合
关系表是具有相同模式的记录的集合	XML 元素的模式可以相同或不同
SQL 查询返回一组无序的记录	XML Query 返回有序的节点序列

1.3 Sheepdog 项目背景

由于上述对支持 XML 高效存储和查询的数据库管理系统的迫切需求，研发原生 XML 数据库有着巨大的商业价值和应用前景，各大科研机构和数据库领域的厂商纷纷在此方面投入了大量的研究。

在这一背景下，复旦大学 WebDB&P2P 实验室与数据库厂商 Sybase 公司共同开展了 XML 存储系统方面的研究，我们的目标是设计一个高效的 XML 原生存储系统，支持大量 XML 数据的快速存储和多种 XML 索引和 XPath 查询，并且支持全文关键词检索和结构相似度检索。其中高效存储和快速 XPath 查询是现有 XML 数据库都在关注的特性，而对于高效的关键词检索和结构相似度检索却是它们的不足之处。

在该项目中，我们需解决的前沿问题主要有：

- XML 数据的灵活存储机制。XML 的数据存储是处理 XML 数据的首要问题。如何找到优化的存储方法，如何建立存储的代价模型都是可以继续探讨的问题，包括和 XML 压缩的结合。
- XML 数据的查询技术。不同于关系数据上的 SQL 查询，XML 数据上常见的查询有路径查询、关键词查询和相似度查询三种形式。第一种像 XPath、XQuery 查询，查询往往牵涉到路径信息。关键词查询类似于传统的信息检索（Information Retrieval）技术，不过返回结果的粒度不同。相似度查询可以用于查找与给定 XML 文档结构相似的文档，可作为寻找相似数据的近似方式，这是 XML 数据上特有的查询方式。

项目开始于 2006 年 2 月，结束于 2006 年 12 月，用 VC6.0 实现，通过实验证明实现的系统达到了预期的目标。项目提供不依赖模式的 XML 文档的存储，这是因为很多情况下文档本身并无相关联的文档定义存在，同时很多 XML 开发者习惯在完成文档后再设计 DTD 或者模式。实际情况中，DTD 也有丢失的可能。

1.4 论文结构

本文共分为五章，第一章总体介绍 XML 数据库系统的研究现状，论述了本论文的研究背景、研究现状和研究内容。第二章介绍了 XML 的一些基本概念及原生数据库相关概念。第三章介绍了 XML 原生数据库 Sheepdog 及其存储设计，在该系统中我们实现了高效的关键词查询和结构相似度查询，这两项技术分别在第四章和第五章进行介绍。最后，第六章为结束语，总结本文的工作并展望未来的研究方向。

第二章 XML 及原生 XML 数据库相关概念

2.1 XML 相关概念

本节介绍在后边的章节中将用到的 XML 的相关概念，包括 XML 标准、文档类型定义 DTD 等。

2.1.1 XML 简介

W3C 提出的标准超文本标记语言 XML（eXtensible Markup Language ）[BPS+98]是一种用于 Internet 上交换和表示数据的格式。一个 XML 文档由嵌套的元素层次结构构成。每个文档有一个唯一的根节点。一个元素有一个标记（tag），描述该元素的含义。一个元素由从起始标记到终止标记的区域构成。该区域可以是嵌套的子元素，也可以是属性或文本串值。图 2.1 显示了一个描述一个研究项目信息的 XML 文档样例。

```
<?xml version="1.0" standalone="yes"?>
  <project>
      <projname>XML</projname>
      <member memberID="&member3">
          <name>M.Franklin</name>
          <email>Franklin@fudan.edu.cn</email>
          <publication>
              <author>M.Fanklin</author>
              <title> Query XML ...</title>
              <year>2001</year>
          </publication>
      </member>
      <member memberID="&member24">
          <name>J.Smith</name>
          <project>
              <projname>data mining</projname>
          <project/>
          <publication>
              <author>J.Smith</author>
              <title> An Algorithm ... </title>
              <year>1999</year>
          </publication>
      </member>
  </project>
```

图 2.1 一个 XML 文档样例

该文档介绍了以项目 project 元素为根的一个 XML 文档。从图 2.1 中，我们可以看出 XML 文档的如下几个特点：首先，一个元素可以由其一个或多个子元素描述。其次，XML 数据是半结构化的数据，一个元素可以有多个标记相同的子元素，且标记相同的子元素可能具有不同的结构，如：project 有两个异构的 member 子元素。

一个 XML 文档可以解析为一棵树。DOM 模型（www.w3.org/DOM/）定义了一个 XML 文档在内存中的树型表示方式和遍历文档的基本操作的接口。应用程序可以通过 DOM 接口方便地处理 XML 文档。

2.1.2 XML 数据模型

XML 数据和半结构化数据是类似的，但 XML 也存在许多一般的半结构化数据所没有的特性（比如 ID,IDREF 等）。本文考虑到 XML 本身的特点，采用了 XML 图(XML Graph) [DFF98]来表示 XML 数据。

定义 2.1 一个 XML 图 G 符合下列条件：

G 是一个有向图，G 的每个顶点都有唯一的标识(OID)

G 的边用元素的标识(Tag Name)来标注

G 的顶点包含一个二元组的集合，其中每个元组形如(属性名,属性值)

G 的叶节点包含字符串型的值

G 有唯一的根节点 Root。

用 XML 图表示图 2.1 的 XML 数据的结果如图 2.2 所示。可以看到，元素的属性包含在节点中而边上的标记为元素自身的标识。XML 图允许两个节点间存在多条边，但边上的标记必须不同。

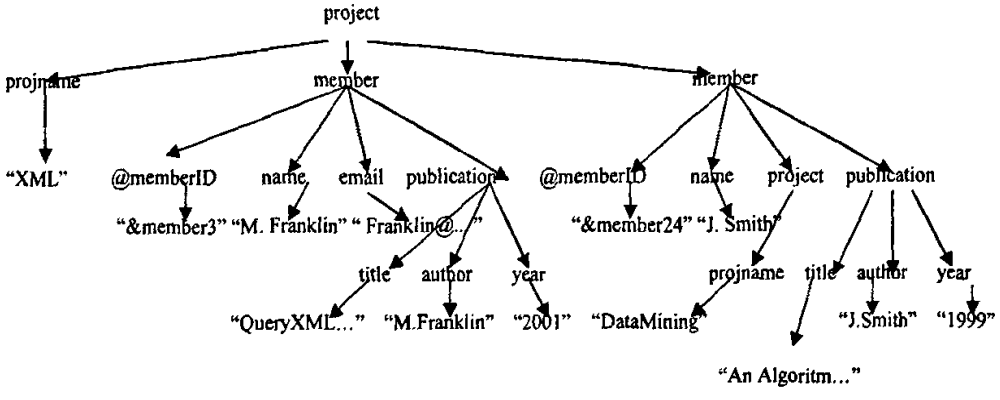


图 2.2 XML数据图示例

元素标识，IDs 和 IDREFs

为了支持元素共享，XML 保留了一个称为“ID”的属性。XML 要求元素的

ID 值必须唯一，因此 ID 可以看作是元素的唯一标识。XML 还有另一个“IDREF”的属性，它允许一个元素引用另一个或多个元素。

从 XML 图的定义可知，每个节点都有唯一的标识(OID),这个标识可以由 ID 属性来提供，也可以由系统自动生成，这是因为在 XML 规范中 ID 属性不是必须的。

和其它的属性不同，IDREF 属性并不是表示为形如（属性名，属性值）的二元组，而是表示为从引用元素到被引用元素的一条边，并且该边用属性名来标识。ID 属性也有不同，它直接成为节点的 OID。

通常，在进行 XML 上的研究时，由于 IDREF 不是很常见，通常把 XML 数据图可以看作树来对待。

元素顺序

在很多情况下，XML 图是一个无序 XML 数据模型，即 XML 元素以什么样的顺序出现在文档中是不可知的。“元素无顺序”的优点是简化了数据模型和查询语言，并允许在查询处理中作顺序无关的查询优化。

在实际应用中，有时顺序是不可或缺的，这时需要扩充 XML 图以记录元素顺序信息。这样的扩充是很简单的，只需在每个节点保留其所有后继节点的顺序即可。有序模型保留了更多的语义但会给存储和查询处理带来额外的开销。

2.1.3 文档类型定义 DTD

文档类型定义 DTD（Document Type Definition）描述 XML 文档的结构，可以将 DTD 看作是 XML 数据的模式或类型。与传统数据库中的模式相比，DTD 的定义要灵活和复杂得多。DTD 以上下文无关文法的方式描述文档中的元素和属性间的嵌套关系。DTD 使用几个操作符：*（表示 0 个或多个）、+（表示 1 个或多个）、？（表示 0 个或 1 个）和 |（表示选择）来描述元素和子元素间的关系。DTD 描述中的所有值类型均被假定为是字符串值，除非由关键词 ANY 定义，此时，值类型可以是任意的文档片段。一个元素类型可以由多个子元素类型或属性定义。其中，ID 和 IDREF 是两种特殊的属性类型，一个元素至多只能有一个 ID 属性，而一个 ID 属性唯一标识了一个元素。一个元素的 ID 属性可以被同一文档中的另一个元素的 IDREF 属性引用。IDREF 属性没有类型。在 DTD 描述中没有根的概念，符合于一个 DTD 描述的 XML 文档的根节点可以为 DTD 中的任意一个元素。例如，图 2.3 给出了图 2.1 中的文档的 DTD 描述。可以看出，图 2.1 中的文档片段的根节点为图 2.3 中的 DTD 中定义的 project 元素。

图 2.3 的 DTD 样例定义了一个实验室的文档类型。一个 laboratory 元素有

一个子元素 labname ，0 或多个子元素 member 。一个 project 有 3 个子元素 projname 、0 或多个 member 、0 或多个 publication 子元素。Member 由一个子元素 name 、0 或一个 email 、0 或多个 publication 、0 或多个 project 子元素和 memberID 属性构成。这里的 memberID 属性由关键词#REQUIRED 描述，表示每个 member 必须有一个 memberID 属性。Publication 又由 0 或多个 author，一个 title 和一个 year 子元素定义。其它元素都为原子元素，被定义为文本串 #PCDATA 。关于 XML 和 DTD 规范的详细描述见 W3C 的相关标准。

```
(http://www.w3.org/XML/)
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE Research>
<!ELEMENT laboratory(labname,member*)>
<!ELEMENT project(projname, (member|publication)*)>
<!ELEMENT projname(#PCDATA)>
<!ELEMENT member (name,email?,publication*, project*)>
<!ATTLIST member ID memberID #REQUIRED>
<!ELEMENT name(#PCDATA)>
<!ELEMENT email(#PCDATA)>
<!ELEMENT publication (author+,title,year)>
<!ELEMENT author(#PCDATA)>
<!ELEMENT title(#PCDATA)>
<!ELEMENT year(#PCDATA)>
```

图 2.3 一个 DTD 样例

2.1.4 XML 查询语言

XML正成为INTERNET上数据表示是数据交换的标准，为了抽取和重构XML文档的内容，研究者们提出了多种新的查询语言，比如UNQL、LOREL、XQL、XML-QL、XML-GL、Quilt 和XQuery，这些语言都使用路径表达式来遍历XML文档中的元素嵌套结构。其中一些与传统的数据库查询语言（如SQL，OQL）类似，其他更多的是针对XML而开发的。2002年，推进Web相关技术标准化的W3C（World Wide Web Consortium）标准制定工作组公布了有关XML查询语言的标准XQuery 1.0。

XQuery的核心是XPath，XPath被用来定位XML文档中的结点。XQuery最强大的新特性是FLWR表达式,。FLWR是 For-Let-Where-Return 的首字母缩略词，这些子句都允许在这些表达式的任何一个中。FLWR 表达式可以完成很多在 XSL 样式表中很难完成的任务。

每个 FLWR 表达式都有一个或多个 for 子句、一个或多个 let 子句、一个

可选的 where 子句以及一个 return 子句。

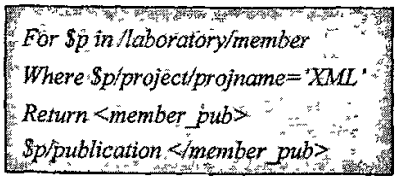


图2.4 XQuery 查询

图2.4显示了该查询的XQuery 语句。For 语句用于描述目标元素（用变量表示），Let 语句用于描述对整个路径表达式的变量绑定，Where 语句设置在目标元素上的条件，Return 语句构造返回的结果。在图2.4中，For 语句中的变量$p 首先绑定 member，然后，Where 语句在变量$p 上设置条件"$p/project/projname='XML'"，这是一个XPath语句，最后，Return 语句构造返回的结果"<member_pub> $p </member_pub>"。For 和Let 语句都是可缺省的，如：图2.4中的XQuery 查询就没有Let 语句。

XQuery是专门从XML文档和含有XML的数据库中，分解和提取信息的一种语言。正如SQL一直是查询基于表结构的数据库的标准，因此，XQuery也因为访问层次性的XML数据库，而成为较为流行的标准语言。特别是，XQuery可以分析XML标记的大量数据库，而不是文件类型的。这样商业就可以充分地检索和分析大量的相关信息，并且对于不同异构系统的数据，XQuery处理起来也是得心应手。XQuery还可以在XML文档中创建结构内容。

XQuery 为 XML 文档中的数据操作提供了一种强大的语法。它最适合于那些同时包含叙述性文本和量化数据的文档。在对这些类型的文档上使用 XQuery 时，为达到最佳性能，可以将这些文档装入一些已建立索引的 XML 资源库中。因为XML的查询语言通常针对于特定类型的数据，因此现有的XML查询语言的规范大多对某种类型的数据源能够很少的进行操作，而对其他的类型却不能体现出优势了。XQuery作为一种比较新颖的查询语言，对几乎所有类型的XML数据都有很好的查询效果。

2.1.5 XML 全文检索

从上一节我们知道已经有多种针对XML的查询语言，可以利用XML查询语言从XML文档中查询所需要的内容。但是在存在大规模来自异构数据源的文档的环境下（例如Internet），普通用户不可能根据各个数据源撰写不同的、准确的查询语句。另外，由于XML文档的来源不同，可能产生XML文档集中标识符存在异名同义的情况。现有的XML查询语言无法判断出这种标识符存在异名同义的情况。同时，由于XML文档查询不同于传统的HTML查询或信息检索，查询语

句中包括路径搜索信息，一般用户难以掌握复杂的查询语言。所以，在用户仅给出简单查询或有限次反馈的情况下，在保证高查准率和查全率的前提下，从大量异构XML文档中找到用户感兴趣的XML文档元素就变得尤为重要。与XML文档查询语言比如XQuery，XPath，XQL等相比，XML的关键词检索技术具有如下特点：首先，不同于查询语言，基于关键词的XML检索简单易操作，用户不需要学习复杂的查询语言，也不需要对XML文档底层的数据结构有深入的了解，用户仅仅需要输入与他感兴趣内容相关的关键词（有时需要输入关键词的类型）。其次，正是因为利用了传统信息检索简单查询信息的优势，其返回的结果和传统信息检索一样也不具有很高的准确性，这是不同于查询语言的方面。查询语言提交后的返回结果，可能并没有多余信息，而恰恰是用户所需要的那部分信息。针对关键词检索非精确的特点，这就要求好的搜索引擎要有效的检索结果评估算法（rank算法，或称为打分算法），为那些与用户搜索最相关的结果赋予较高的分值，为用户检索提供方便。当然，随着查询语言的发展，新兴的XML检索技术并不能完成所有查询语言所能完成的工作，比如复杂谓词和包含复杂语义的查询就不能通过简单的关键词检索技术来实现。

和搜索引擎根据关键词返回整个网页不同，XML上的全文关键词返回的是包含所查询关键词的最小片段。比如对图2.1所示XML文档进行查询关键词“XML 2001”时，结果是如图2.5的片段。这样的查询使得返回结果更加精确，更符合XML文档前套型树结构的特点。当然，如果用户需要，也应该能够返回整个XML文档。XML全文检索所面临的主要挑战存在于以下两个方面：1）如何快速的找到满足查询的片段（子树）集；2）如何对结果集进行排序，即如何评价结果的优劣程度。我们对Sheepdog系统中在XML全文检索方面提供了一个快速的解决方案。

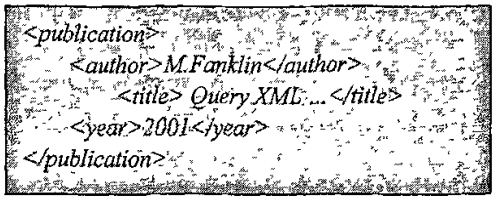


图 2.5 关键词查询结果片段

2.2 原生 XML 数据库

本节介绍原生数据库的概念以及分类，并具体考察了一个现有的原生 XML 数据库的实例——eXist。

2.2.1 原生 XML 数据库概念

Ronald Bourret 在其《XML 与数据库》[Bou99]一文中对原生数据库（native XML database）这个术语采用了以下定义：

“它为 XML 文档（而不是文档中的数据）定义了一个(逻辑)模型，并根据该模型存取文件。这个模型至少应包括元素、属性、PCDATA 和文件顺序。这种模型的例子有 XPath 数据模型、XML Infoset 以及 DOM 所用的模型和 SAX 的事件。

它以 XML 文件作为其基本(逻辑)存储单位，正如关系数据库以表中的行作为基本(逻辑)存储单位。它对底层的物理存储模型没有特殊要求。例如，它可以建在关系型、层次型或面向对象的数据库之上，或者使用专用的存储格式，比如索引或压缩文件。”

这个定义涵盖了原生数据库的存储模型、存储单位和存储格式。其中存储模型这一概念类似于关系数据库；存储单位（即可容纳一份数据的最低级的上下文）则限定了是 XML 文件，在关系数据库中则是行。看起来似乎也可存储 XML 文件片断，但几乎所有的原生 XML 数据库都是以文件方式存储的；原生数据库底层的数据存储格式并不重要，正如关系数据库所使用的物理存储格式与数据库是不是关系型之间毫无关系。

2.2.2 原生 XML 数据库的分类

原生 XML 数据库的结构可分为两大类：基于文本的和基于模型的。

基于文本的原生 XML 数据库 基于文本的原生 XML 数据库(Text-Based Native XML Databases)将 XML 作为文本存储。它可以是文件系统中的文件、关系数据库中的 BLOB 或特定的文件格式。（事实上，就其能力来说，一个增加了支持 CLOB(Character Large Object)字段的 XML 处理功能的关系数据库也可以是原生 XML 数据库了。）

索引对所有基于文本的原生 XML 数据库来说都是一样的，它可以使查询引擎很方便地跳到 XML 文件内的任何地方。这就可以大大提高数据库存取文件或文件片断的速度。这是因为数据库只需进行一次检索、磁头定位，再假如所读的文件在磁盘上是连续存储的话，只需一次读盘就可读出整个文件或文件片断。相反，如果像关系数据库或基于模型的原生 XML 数据库那样，文件由各个部分组合而成，就必须要进行多次查找定位和多次读盘动作。

从这个意义上讲，基于文本的原生 XML 数据库与层次结构的数据库很相似，

当存取预先定义好层次的数据的时候，它比关系数据库更胜一筹。和层次结构的数据库一样，当以其他形式比如转置层次存取数据时，原生 XML 数据库也会遇到麻烦。

基于模型的原生 **XML** 数据库 第二类原生 XML 数据库是基于模型的原生 XML 数据库(Model-Based Native XML Databases)。它们不是用纯文本存储文件，而是根据文件构造一个内部模型并存储这个模型。至于模型究竟怎样存储取决于数据库。有些数据库将该模型存储于关系型和面向对象的数据库中，例如在关系型数据库中存储 DOM 时，就会有元素、属性、PCDATA、实体、实体引用等表格。其他数据库使用了专为这种模型作了优化的专有存储格式。

建立在其他数据库之上的基于模型的原生 XML 数据库的文件存取性能与这些数据库相似，很明显，它的存取要依赖这些数据库。但是这个数据库，特别是建立在其他数据库之上的原生 XML 数据库的设计有很大的变化余地。例如直接以 DOM 方式进行对象-关系映射的数据库系统在获取节点的子元素时必须单独执行 SELECT 语句。另一方面，这种数据库大多对存取模型和软件作了优化。例如 Richard Edwards 在 “System for Storing the DOM in a Relational Database” 一文中曾经描述只用一个 SELECT 语句就可获取任意文件片断(或整个文件)。

使用专用存储格式的基于模型的原生 XML 数据库如果以文件的存储顺序读取文件，其性能与基于文本的原生 XML 数据库相似。这是因为这种数据库大多在节点间使用了物理指针，这样其读取性能和读取文本差不多。(究竟哪个快一些要取决于数据格式。如果返回文本格式，显然基于文本的系统要快一些；如果希望返回的是 DOM，假如该模型很容易映射到 DOM，则基于模型的系统更快。)

2.2.3 原生 XML 数据库的存储设计要则

存储粒度 一般来说有两种存储 XML 文档的方式：完整存储(intact storage)和分散存储(non-intact storage)。前者的存储粒度是整个文档，后者的粒度比整个文档小，比如可以是子树、大多数时候是元素。对于前者，优点在于能够 100%地取回原来的文档，满足 round-tripping 的性质，缺点是要花费较多的时间和内存去进行文档的解析和查询处理；对于后者，在 round-tripping 的程度上自然比不上前者，但它能够采取更灵活高效地存储方式，以及对数据建立索引，便于查询处理。

存储模型 对完整存储的粒度来说，不需要什么存储模型，存储模型对细粒度的存储才需要。选择怎样的存储模型，对元素、属性分别怎样进行存储，物理存储格式怎样都需要进行仔细的设计。

索引创建 使用索引可以使得查询的效率大幅度提高，但是索引也会占据磁

盘空间和 cache 开销。一个经典的空间和时间问题的折中。选择索引的类型（值、结构、全文索引），索引的范围（文档、文档集），索引的目标（文档、结点）都会对索引的效率产生重大影响。

2.2.4 原生 XML 数据库的代表 - eXist

eXist 是一个开放源码的 XML 数据库系统，于 2001 年 1 月开始研究，还在不断的发展中。它提供了无模式定义的 XML 文档存储功能，即文档只要格式正确，就可以被系统接受。eXist 对文档是以层次的集合概念组织的。从用户的角度来说，这种策略和文件系统中的文件系统中的文件策略是类似的，数据集合可以随意嵌套。在一个数据集合中，任意类型的文档都可以混和存储。eXist 尤其强调了有效的基于索引的查询处理过程。eXist 的检索引擎可以提供快速的 XPath 查询，通过使用文档中所有的元素、文本和属性结点的索引实现。在路径连接算法的基础上，仅仅使用索引信息就可以处理很大一部分查询表达式。该索引模式不进把 XML 存储中实际的 DOM 结点与索引入口连接起来，而且还提供了对文档结点树中的结点之间可能关系（包括父子结点关系或祖孙关系等）的快速识别功能，而不像传统的方法那样，使用自顶向下或者自底向上的文档树遍历方式。从而使得性能要高一个数量级。

举例来说，XPath 表达式

/book//section [contains (title, 'xml')]

要在一个有关书目内容的大集合中找书名中包括字符串 XML 的书目信息。在传统的自顶向下算法中，XPath 处理器必须查找 book 结点下所有的子结点路径以发现可能存在的 section 子孙结点。这意味着算法必须遍历文档树中根节点 book 下的所有元素，因为系统无法预先提供 section 子孙结点在文档中的可能位置。当文档的内容完全在内存中时，遍历文档树的计算开销还可以接受，然而当文档内容存储在持久性存储设备中时，性能就会大幅度降低。为了判断结点的名称和类型，很多不包括 section 元素的结点也被装载到内存中，造成了大量的磁盘 I/O 开销。eXist 使用编号方案来辨别 XML 结点，并且判断在文档树中结点之间的关系。编号系统为文档中的每个结点提供数字标识符（可以通过层次顺序或者预先定义的其他顺序遍历文档树实现）。索引中使用这些生成的标识符代表实际的结点。在 eXist 中，所有 XML 结点在内部由一对文档标识和结点标识来表示。很多查询处理只用这两个标识符完成。除非系统需要，eXist 不必寻找真正的 DOM 结点。

XML 数据存储是 eXist 的核心部分，它由一个包括了所有根据 DOM 存储的文档结点的分页文件组成。数据存储被一个多根结点的 B+树支持，它也存储在

这个文件种，用于把一个结点的标识符与它在数据部分的结点的存储地址联系起来。

eXist 作为一个开源的项目，取得了很大的成功，在很大程度上依靠用户的反馈和参与，并在不断的进展。它的某些弱点（比如索引速度和存储要求）正在进行重新设计。另外，它不支持 XQuery、不支持 XUpdate。

2.3 小结

本章简要介绍了一下 XML 以及原生 XML 数据库的基本概念，为后面进行设计和实现 Sheepdog 系统打下基础。

第三章 Sheepdog 系统介绍

本章介绍了 Sheepdog 系统的需要实现的功能特性和系统架构图，着重对其存储模块的设计进行了介绍。

本章结构如下：3.1 节简要介绍了 Sheepdog 实现的功能特性；3.2 节给出了系统的架构图，3.3 节对各个模块、存储方案和存储效率进行了详细介绍；Sheepdog 系统中两个重要的模块即全文关键词查询和结构相似度查询也是本文的重点，因此分别在第四章和第五章分别单独介绍；3.4 节介绍了系统实现及界面；最后 3.5 节作了本章小结。

3.1 Sheepdog 的功能特性

Sheepdog 被设计成具有丰富功能特性的 XML 数据管理系统。它可以作为一个单机版的 XML 数据库，同时，它提供的 API 函数使其又可作为一个嵌入式的系统存在，即可以被作为一个库链接在程序中，这一点与 Berkeley XML DB 类似。该系统提供了以下功能：

- 设计良好的存储模型，提供高效的 XML 存储，提供方便的 XML 数据访问策略。
- 支持多种索引创建，采用有效的索引策略来加速 XML 数据上的查询。
- 提供对 XML 数据的多种查询方式，包括 XPath 查询，全文关键词查询，以及结构相似度查询。

由于该系统为研究型项目，主要关注 XML 数据的存储和查询，所以并未在数据库的一些特性比如事务机制上进行较多的探究。这也是将来要进行的后续工作。

3.2 Sheepdog 的系统架构图

系统架构如图 3.1 所示。对其各个功能模块描述如下。

- 输入模块：这个模块接受三种输入，XML 文档和 XPath 查询，关键词查询，XML 文档名。XML 文档经过解析（XML Parser）模块验证解析后被系统接受并存储在系统内，后面三种查询输入，分别调用查询引擎模块中的 XPath 处理模块，全文检索模块和结构相似度查询模块处理。
- 输出模块：对应输入查询的结果，当输入 XPath 查询或者关键词查询时，输出结果是 XML 结点序列，对应结构相似度查询，会返回和指定文档（通过文

档名指定）结构最相似的整个 XML 文档。

- 查询处理引擎：负责查询解析，查询优化和查询处理，具体对上面提到的三种查询输入调用三个查询处理模块执行。同下层的存储引擎联系，进行数据的获取。
- 存储引擎：系统的核心部分，负责分解 XML 文档并按照存储模型进行存储，同时创建 XML 索引，缓冲数据管理等。
  - 索引管理模块：负责对存储的文档自动建立索引。
  - 数据管理模块：负责对 XML 文档进行存储，访问。
  - 缓冲管理模块：对使用频繁的数据进行 cache 管理。

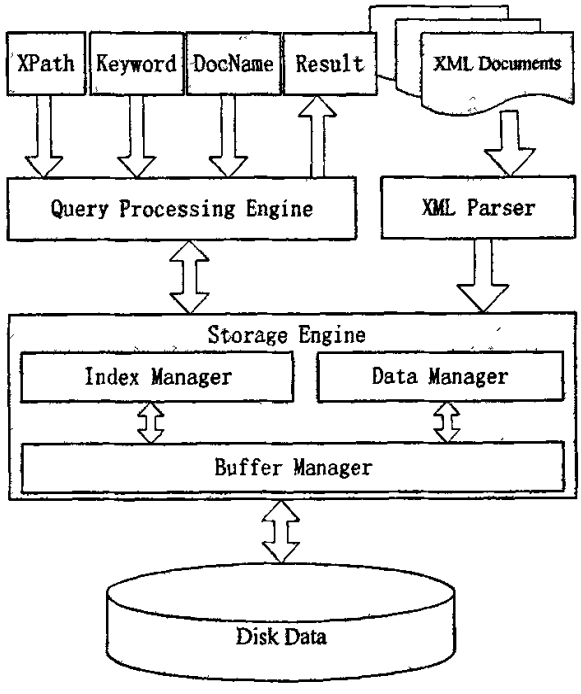


图 3.1 系统架构图

3.3 Sheepdog 系统存储模块设计

3.3.1 存储方案

在我们的 XML 数据库系统中，XML 文档在存储时，一个副本拷贝被直接存储，系统中有个文件数计数器，新加入的文档的编号由计数器加 1 获得。同时我们用一棵 B 树存储了当前 XML 文件名（键）和文件号（值）这样一个映射。由于用了文件名

作为键，所以我们要求存储的 XML 文件不能有重名现在，而且存储的文件是没有层次的，相当于都在一个容器之中。存储副本是基于两个考虑。一种是为了直接需要查找此文档时，能够快速返回，或是在需要文档的片断时，我们仅需要知道片断在该文件中的起始和结束位置就可以快速返回该片断。这总的来说，是为了 round-tripping 的问题，一些 XML 数据库很难完全 100%的返回原来的 XML 文档。另一种考虑是为了第五章我们要探讨的进行 XML 文档结构相似度查询的需要，保留副本在一定程度上方便了提取 XML 的结构信息。虽然存储了副本，数据的存储耗费也只是增加了原文档大小，还是可以接受的。存储原文档的同时，文档被解析，分解成元素、属性和文本结点。在分解的过程中，每个结点被赋予一个结点号(Node ID)，类似于 eXist 系统中的标识号。实例中，我们使用的是杜威（Dewey）编码，这种编码是对 XML 文档的结点按图 3.2 所示的方式进行编号，它的好处是可以对每个结点保存路径信息，然后凭编码就能断定结点间的父子/祖孙后代关系，从而对查询处理时的连接运算特别有效。

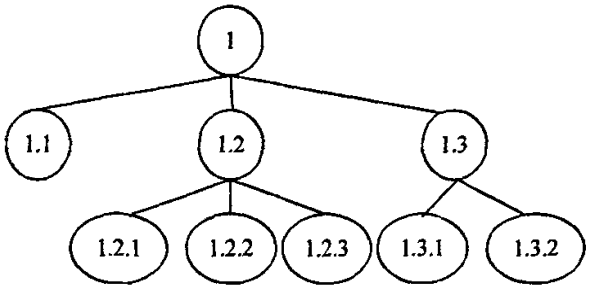


图 3.2 杜威编码

所有的 XML 数据存储在一个基于磁盘的 B 树上。B 树的键是结点 ID，值是结点。每个结点数据由以下几部分组成：结点的类型、结点的标签、结点在所在文档起始位置、结点在所在文档结束位置和结点的孩子数目。具体见图 3.3 所示。

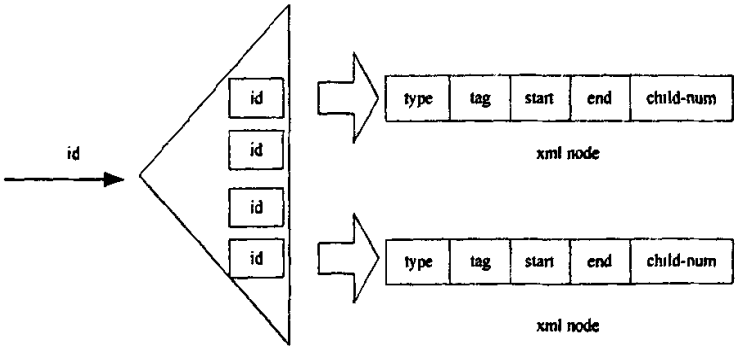


图 3.3 存储在磁盘 B 树上的 XML 结点

具体地，这棵 B 树，在磁盘上是一个分页文件。如图 3.4 所示。文件被分成众多的虚拟页。页有两种，一种是存放 B 树结点的页，一种是存放结点数据的页。后者是

可以支持变长数据的，即当结点数据过大，可以分散到几个页中存储，几个页之间用指针链进行链接。这样一棵 B 树是部分加载到内存的，只有用到的结点才会被加载到内存。

在存储 XML 数据或者回答查询的时候，我们有时需要频繁地访问这个文件。例如，当新的数据加进来的时候 B 树的结点会频繁分裂，当处理查询的时候一些结点也会被频繁访问，此时 cache 策略就很有必要。我们在 B 树的结点和虚拟页之间有一个两层的 cache 策略，B 树结点 cache 和虚拟页 cache。更新过的 B 树结点置为脏并且只在它们被换出 B 树结点 cache 的时候写回虚拟页的 cache。更新过的虚拟页被置为脏，并且只在它们被置换出页 cache 的时候才会被真正写回磁盘。Cache 置换的策略可以有很多，我们这里用最常见的 LRU 策略。

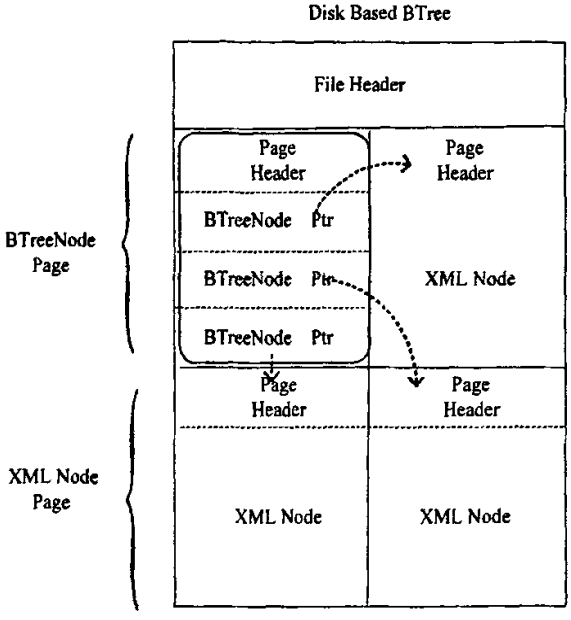


图 3.4 基于磁盘的 B 树

存储杜威编码要比 eXist 系统中存储一个整数作为编码耗费大，因此当存储 B 树结点的时候，为了节省空间开销，我们用存储压缩编码的方式。将邻近的几个结点的共同前缀求出来，然后只单独存储他们不同的部分，可以减少空间消耗。具体如图 3.5 所示。

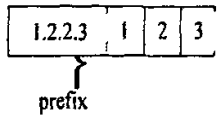


图 3.5 压缩存储 Dewey 编码

除了上述 B 树，还有一个 B 树用于快速的根据 XML 元素的 tag 访问 XML 元素结点。比如，我们要找所有的“book”元素，使用字符串“book”做为键在 B 树上进行

查找，返回的是所有的“book”元素。具体如图 3.6 所示。

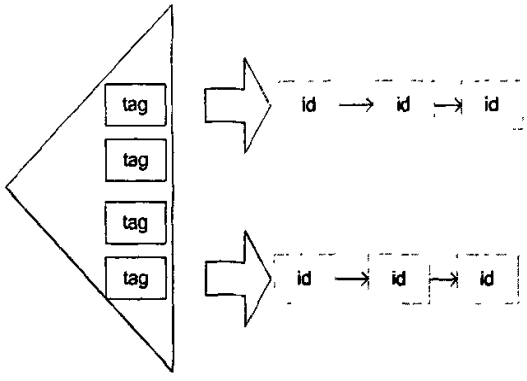


图 3.6 用于根据 tag 查找元素的 B 树

另外为了支持关键词索引，我们同样要有一个倒排表文件。如图 3.7 所示，对每个 XML 文档的每个单词，它在哪些元素中出现过都被记录下来。这和传统的信息检索领域（Information Retrieval）领域不同的地方就是它们记录的是单词出现的位置，频度等信息。更具体的介绍对关键词查询的处理在第四章有详细介绍。

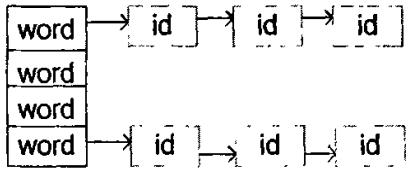


图 3.7 倒排索引

在源文件中，我们实现的系统顶层的类叫做 Collection，它作为一个 XML 的容器存在，对其详细设计如图 3.8 所示。该类使用一个 DataManager 来管理数据，LCAManager 来进行关键词查询，IndexBuilder 来创建各种索引，该类本身提供方法支持用户创建 XML 容器、存储或删除 XML 文档。

3.3.2 数据存储详细介绍

在我们的 XML 存储中，所有的数据存储使用了一个文件夹和四个文件：

（1）native 文件夹：存放所有的 XML 文档副本。

（2）doc_id.idx 文件，管理文件名到文件号的映射。

（3）id_node.idx 文件，管理结点号到结点的映射。

（4）tag_id.idx 文件，管理结点标签到结点号的映射。

其中后三个文件分别对应三棵 B+树，用来实现 O(logn)的查询。其中 Doc_ID 实现通过 XML 文件的文件名找到其 XML 文件的文档代号，ID_Node 实现通过结点的 ID 来获取该结点相关信息，Tag_ID 实现获取所有标签与查询字符串相同的结点的 ID。

```
class Collection
{
public:
    /*创建容器*/
    Collection(const string& dbname, bool allowCreate = true);
    virtual ~Collection();
public:
    void          open();
    void          close();
    void          drop();
    /* 存储 XML 文档 */
    void          loadXMLDocument(const string& xmlfile);
    /* 判断 XML 文档是否存在 */
    bool          isDocExist(const string& docName);
    /* XPath 查询处理，可以指定目标文档，缺省为整个容器 */
    vector<Node*>   evaluateXPath(char* xmlfile, char* xpath);
    /* 删除 XML 文档 */
    bool          deleteXMLDocument(const string& xml);
    /* 全文检索 */
    vector<Node*>   keywordSearch(string key);
public:
    /* 数据存储管理 */
    DataManager*    db;
    /* 全文检索查询处理*/
    LCAManager*     lb;
    /* 文档计数器 */
    int           docNum;
    /* 索引创建器 */
    IndexBuilder*   ib;
    /* 容器名 */
    string        dbname;
};
```

图 3.8 Collection 类的设计

三个 B+树的存储都是以 Value 类的对象作为关键字和值进行存储，Value 类的定义包括两个部分，data_len 和 data，data_len 是 data 的长度。B+树的节点和内容都是存进以链表形式组织起来的固定大小的页里（每个页中都有一个 long 型的 nextPage，表示

它下一个页的页号，-1 表示没有）。B+树中比较关键字有两种顺序，一种是 ID_Order，一种是字典序，其中 ID_Node 使用的是 ID_Order，Tag_ID 和 Doc_ID 使用的是字典序。一般来说，存储数据的过程是，先通过 B+树找到存储内容的第一个页的页号，读出第一个页，第一个页中有记录最后一个页的页号，然后把最后一个页的内容读出来，在后面添加新加入的内容，再从最后一页开始存回去，当前页不够用的话取一个新的页并更新。B+树节点分裂的条件是当节点的大小超过一个页的大小时，把节点拆成具有相同数目关键字的两个节点。

数据的存储依赖对这些 B+树的操作，具体的实现举例如下：

（1）ID_Node 中添加结点

因为文档编号是递增的，而且 ID 的编号也是递增的，所以每个新加入的 ID 必是当前 ID_Node 中 ID 最大的一个，考虑到这点特殊性，可以直接定位 B+树中最后的一个节点，然后把 ID 加在关键字数组最后即可。此时放弃使用 cache 来做缓存，采用以 vector 为数据结构的 cacheStack 模拟一个栈。加入第一个 ID 时，先把 ID_Node 中最后一个节点和从根节点到这个节点所经过的所有节点按读取顺序存进 cacheStack 中，因为每个节点都是它父节点的儿子节点中最大的一个，故不用二分查找即可直接定位。然后把 Node 加入最后一个节点中。以后每加入一个 Node，只要取 cacheStack 中的最后一个节点加进去即可。如果节点要分裂，做法是：1、把节点从 cacheStack 中弹出来；2、右节点分一个关键字，其余的都分给左节点（因为以后的内容都是往右节点上添加），此时左节点就可以写入硬盘并释放掉了；3、如果还要父结点要分裂，则在父结点中重新做 1；4、把右节点加进 cacheStack 中。

因为 ID_Node 中每个 ID 是唯一的，即只有一个 Node 与它对应，如果用一个页来存一个 Node 的话，会造成很大的空间浪费。故采用另一个文件来存储数据。ID_Node_data.idx 包括文件头和文件内容。数据是以块来分割的，文件头有三个数：firstFreeBlock，lastFreeBlock，freeBlock，firstFreeBlock 表示回收链表中第一个空闲块的位置，lastFreeBlock 表示回收链表中最后一个空闲块的位置，freeBlock 表示文件的尾位置。一个块的组织形式是：data+状态数 pos。data 是一个有固定长度（DATA_STORE_SIZE）的数据块,是存储数据的地方，pos 是表示当前块状态的参数。一个数据是存储在以链表形式组织的块中。pos>0 时表示当前块不是数据的结尾，pos 表示下一个块的位置; pos<0 表示当前块已经是数据的结尾了，length = -pos-1 表示数据在当前块存储的长度，即从块头 p 到 p+length 是数据。而在 B+树中对应 ID 的指针存储的是数据在 ID_Node_data.idx 中第一个块的位置。比如 Value = 10abcdefghij 要存储，则设 ID_Node 的指针为 10，则有：（设 DATA_STORE_SIZE 为 5，[][][][]表示 4 个字符代表的是 long 型数据,*表示任意字符）

文件位置：	10	11	12	13	14	15	16	17	18	19
文件数据：		[]	[]	[]	[]	a	[]	[]	[]	[]
真实数据：		1	0			a			20	
文件位置：	20	21	22	23	24	25	26	27	28	29
文件数据：		b	c	d	e	f	[]	[]	[]	[]
真实数据：		b	c	d	e	f			30	
文件位置：	30	31	32	33	34	35	36	37	38	39
文件数据：		g	h	i	j	*	[]	[]	[]	[]
真实数据：		g	h	i	j	*			-5	

（2）对 Tag_ID 的处理技巧

考虑到关键字查询的方便，存进 Tag_ID 的内容并不是 ID 本身，而是只存进去 3 个 fields：docNum、size、offset。表示在以 docNum 为文件序号的 XML 文件中有 size 个标签为 tag 的节点，他们 eulor 序号（见第四章介绍）的按顺序连续存在 NodeData（docNum）.idx 中，开始位置为 offset.在解析 XML 文件的时候，使用一个以 tag 为关键字，存储相应节点的 eulor 序号的 vector*为值的 map 型 Recordcache，每遇到一个以 tag 为标签的节点，则在 Recordcache 中找出相应的 vector*，再把它的 eulor 序号存在最后一个位置。解析完后遍历一次 Recordcache，把它存进 Tag_ID 里面去。

（3）删除数据

在 Doc_ID 中删除时，只要通过文件名找到存储内容的第一页，进而得到最后一页的页号，然后令回收链表中 lastFreePage 指向第一页，然后令 lastFreePage 为最后一页即可。再在 B+树中删除节点。在 ID_Node 和 Tag_ID 中删除时，先通过文件名得到对应的文档号，然后以文档号为关键字找到存储在 ID_Node 中的 XML 文档的根节点，按顺序遍历 ID_Node，直到节点不属于 XML 文件,把所有节点都存进一个 vector IDRecord。对每个节点，取出他的 Node 内容，接着得到对应的 Tag，用 Tag 在 Tag_ID 中找到其存储的数据，在数据中如果 docNum 与要删除的文档的序号相同，则把相应的 docNum、size、offset 删掉，然后重新从第一个页开始存回去，如果有剩余页，则得到最后一页，再按照 Doc_ID 中回收页的办法做。如果删除后没有数据，则把 Tag_ID 中 Tag 对应的节点删除掉。最后遍历 IDRecord，从 ID_Node 中删除 IDRecord 中的每个节点和其对应的内容。在 ID_Node 中的数据以块的形式存储。删除的过程与页的删除过程相同，通过 ID_Node 得到第一个块的位置 P_1 后，遍历链表，得到最后一个块的位置 P_2，然后令 lastFreeBlock 的下一个块是 P_1,更新 lastFreeBlock 为 P_2 即可。

（4） 查询数据

Sheepdog 使用 map 来做缓存，减少硬盘操作，装载完文件后再写回硬盘。BTree 有一个以页数为关键字，BTreeNode*为值的缓存 cache 和以页数为关键字，Page*为

值的缓存 pages。如果要得到 B+树节点，则在 cache 中找，如果找不到，则在 pages 中找相应的页，再找不到，则从硬盘中读取。通过 key 找到存储的位置后，把内容读出来即可。ID_Node_data 中的数据按照和存储相同的方法读出来。而 Tag_ID 则是读出若干组 docNum、size、offset。则在 NodeData（docNum）.idx 中 offset 位置开始读取 size 个 int 型参数，即其节点所对应的 eulor 序号。然后就可以通过 eulor 序号得到相应的 ID。对每个 eulorPos 在 eulorPos_ID(docNum).idx 中 eulorPos*sizeof(int)位取出 ID 所存储的位置 offset,再到那个位置取 ID 即可。

表 3.1 DBLP 上的测试

docsize	nodes	time(s)	space(k)	Qtime(s)
1k	79	0.078	9	0.00
10k	850	0.172	94	0.00
100k	9,415	1.39	1,075	0.01
1M	95,170	14.25	10,851	0.27
10M	905,240	102.27	109,774	6.57

表 3.2 SigmodRecord 上的测试

docsize	nodes	time (s)	space	Qtime
60k	2,267	0.95	635	0.00
125k	4,157	1.59	1,172	0.00
250k	8,459	3.01	1,779	0.01
500k	18,986	6.63	5,591	0.27
1M	37,970	13.23	10,863	6.57

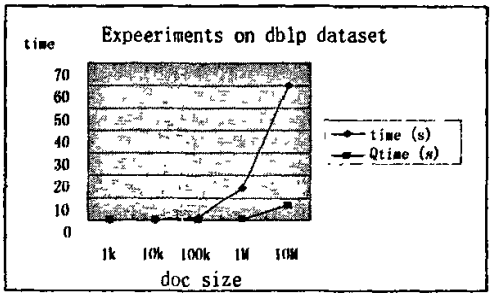


图 3.9 DBLP 上的存储和查询

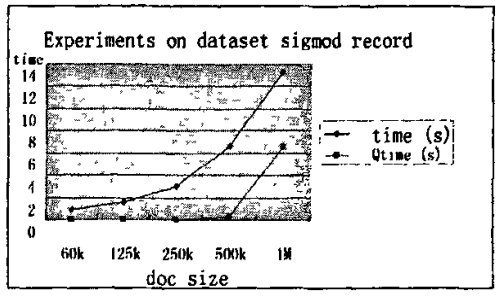


图 3.10 SimgodRecord 上的存储和查询

3.3.3 存储效率

我们做了几组实验来验证我们存储系统的效率。所有我们的实验都是在一台奔三 1.8GHz CPU，256MB 内存的 PC 上进行。操作系统时 Windows XP，我们用 Visual C++ 6.0 实现了上述系统。实验用的数据集包括 DBLP 和 SimgodRecord [URL]。我们用以下四个标准来衡量系统的效率：存储消耗的时间(time)，存储的结点数目(DocNodes)，磁盘的开销（space），执行查询的时间（qtime）。我们测试随着文档大小变化这些指标的变化情况。对每一次测试，我们都创建 3.3.1 节里提到的新的 B 树结构。测试结果在表 3.1 和 3.2 中显示。查询时间指不用索引的执行时间。所用的查询在表 3.3 中列出，每个查询被执行 10 次取平均时间。从图 3.9 和图 3.10 中存储的时间相比文档大小都是可以接受的。对于中等大小的文档，我们的系统工作地很有效。但对大文档，

性能就会下降（图 3.9 和图 3.10）。这是因为文档越大，要存储结点的数目会激增，存储的效能就会下降。

表 3.3 使用的查询集

Data set	XPath
Dblp	/dblp/article
	//www/url
	//dblp/www
	//article[year='1989']
Sigmod record	//issue/number
	//article/title
	//article/author
	/issue[volume='12']

3.4 系统实现及界面

系统采用 MicroSoft Visual C++6.0 开发，XML 解析器采用 Xerces-c++[URL6]，可执行文件为 Sheepdog.exe，使用时，需要先安装 Xerces-c++，然后直接运行 Sheepdog.exe 即可。系统启动后界面如图 3.11 所示。启动成功，系统会创建一个默认名为 XMLDB 的文件夹做为顶层 Collection 容器，容器中生成如下几个文件。XMLDB.cfg，容器配置文件；Doc_ID.idx、Tag_ID.idx、ID_Node.idx、ID_Node_Data.idx，这些文件的作用前面已经介绍了；BlockTable.idx，这个文件用于存储关键词检索中频繁计算的数据值(在第四章中，求解结点的公共祖先算法中，会频繁的进行对数值的计算，比较耗时)，在第四章介绍；native 文件夹，用于存储加载进来的 XML 文档的备份。

系统界面主要划分为三部分，第一部分，进行 XML 文档的加载和存储；第二部分进行 XML 数据上的查询，第三部分是展示查询结果。这些功能在界面上都以按钮的形式反映出来。

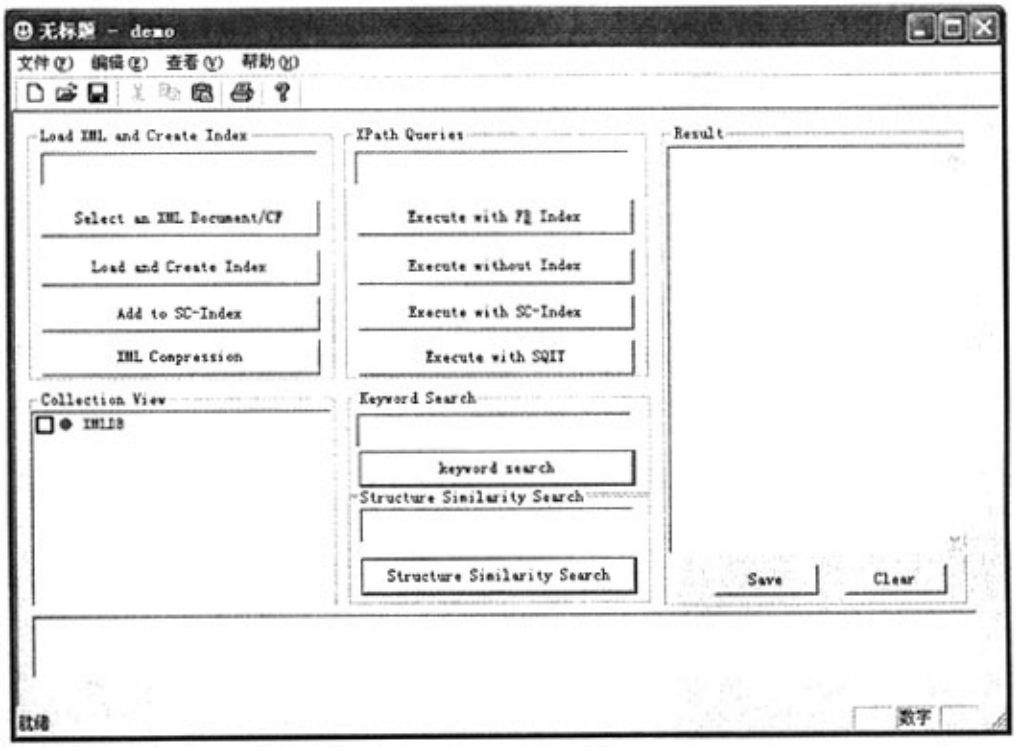


图 3.11 Sheepdog 系统启动界面

用户可以选择并且加载 XML 文档进行存储，并且存储时系统支持创建几种高级的索引（比如 F&B 索引和 SC 索引），存储时存储容器的结构在左下角以树视图的形式展现。如图 3.12 所示，容器中新存储了一个 country.xml 的文档。

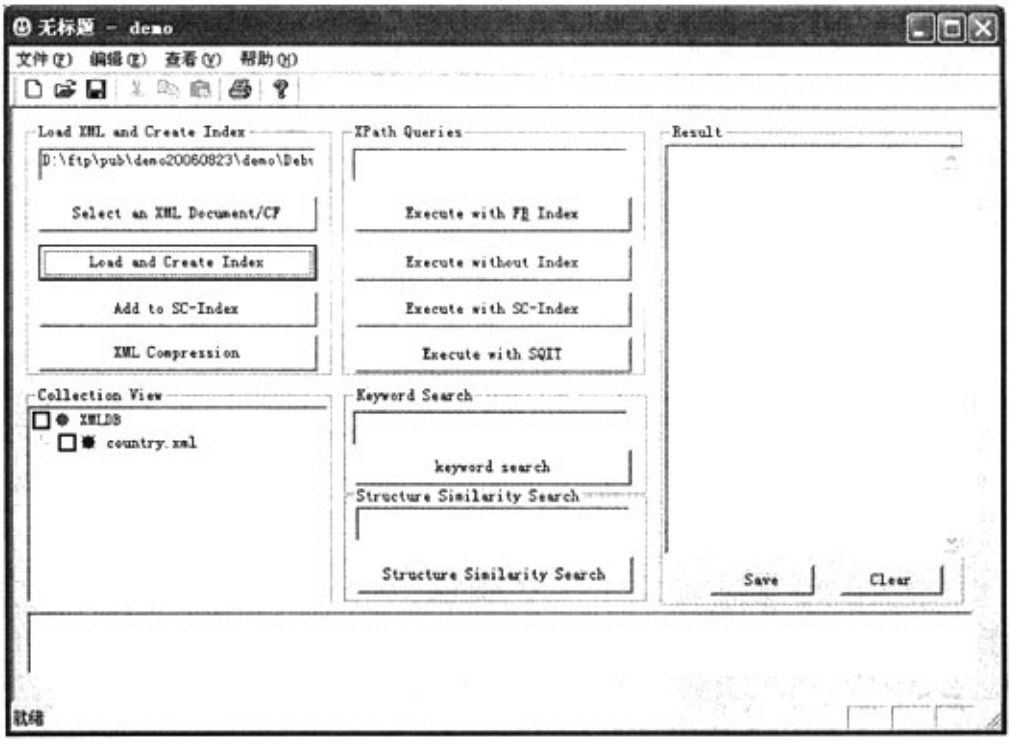


图 3.12 加载并存储一个 XML 文档

在查询区域可以在存储的 XML 数据上进行 XPath 查询，在进行查询时，可以选择利用或不利用以上高级索引技术进行查询，这部分功能是为了测试和展示索引技术对查询速度的提高。同时用户可以在左下角容器视图中选择要在哪些 XML 文档上进行查询，查询结果在右侧 Result 面板里显示并可以保存下来。底部信息栏显示查询结果的一些统计信息。查询的一个例子见图 3.13。

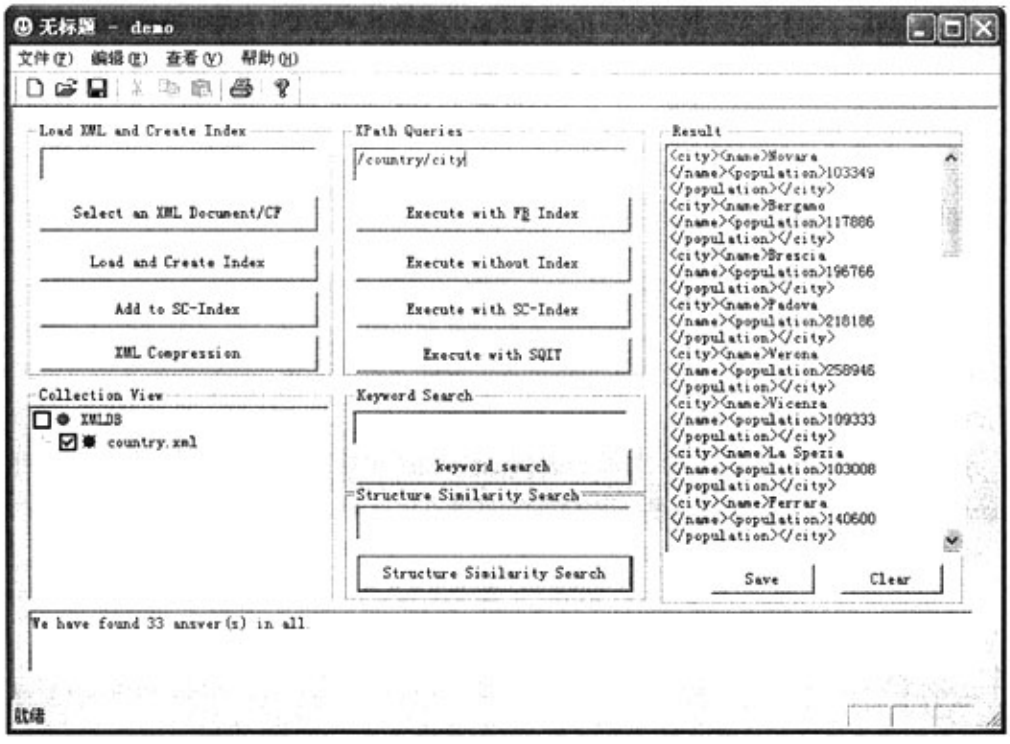


图 3.13 执行 XPath 查询

3.5 小结

本章主要介绍了 Sheepdog 系统的功能特性、系统架构以及存储模块的设计，并详细介绍了数据存储的方式。提供的实验结果证明我们这个系统在存储上和查询上的效率都是相当高效的。

第四章 Sheepdog 中关键词检索技术

4.1 引言

绪论中提到，相比传统关系数据库的 SQL 查询，XML 数据库有 XPath 或 XQuery 查询、全文关键词检索和结构相似度查询等多种查询方式。关键词的查询是信息检索领域一项重要的研究，常常体现在搜索引擎的研究上。搜索引擎基本上是对网页信息中关键词的检索，随着互联网上 XML 文档的不断增多，对这些数据的使用越来越依赖于互联网搜索引擎强大的检索能力，对检索 XML 文档的搜索引擎的研究也就越来越迫切[ZLD+01]。与信息检索领域内的关键词检索不同，XML 数据实现关键词检索必须以 XML 元素为粒度进行。相对而言，传统的搜索引擎是以文档为粒度的检索，也就是说如果某个文档包含了某个关键词，就将该文档返回给查询用户。在基于 XML 文档的关键词检索中，对于符合检索条件的文档，不但要求该文档包含特定的关键词，而且要求该关键词的出现符合一定的条件，即关键词必须被该文档中指定的元素所包含。在返回检索结果的时候，并不需要将整个文档返回给用户，而只需返回用户感兴趣且符合关键词包含关系的元素集合或者 XML 子树集合，该集合可以看作是原文档的一个片段。因此，XML 文档中的关键词检索提供了内容和结构上的双重检索要求。XML 文档检索和 HTML 文档检索在以下几个方面有着明显的区别：

- 用户检索对象的单元是元素粒度而不是文档粒度；
- 用户检索时所能利用的不仅有文本原始信息（就如传统信息检索），还有 XML 文档的结构信息。
- 为了在准确性上进一步提高，在定义元素相似度的时候不仅需要关键词之间的空间距离，还需要元素间的“结构距离”，即在结构上的关系。

于是 XML 检索技术就不仅仅是 HTML 搜索技术的移植。所以需要针对 XML 检索开发不同于传统信息检索和 HTML 搜索的方法和技术。

本章主要介绍了 Sheepdog 系统中实现的 XML 文档的关键词检索技术。本章结构如下，第 4.2 节介绍 XML 关键词检索的一些相关工作；第 4.3 节介绍本章所用到的一些技术定义；第 4.4 节介绍本章所提出的关键词查询技术中的核心——基于 RMQ 的 LCA 算法；第 4.5 节展现了做为 Sheepdog 一个查询处理模块的 XML 关键词检索部分的示意图；最后第 4.6 节给出了实验分析，第 4.7 节做了总结。

4.2 相关工作

近来“灵活查询（Flexible Query）”的概念在 XML 查询中盛行，所谓“灵活查询”就是指对 XML 文档不仅能够通过查询语言进行查询，还应该有适用于非专家用户的 XML 关键词检索。原有的查询语言越来越多地着眼于为关键词检索提供扩展，比如说，XQuery 制定工作组正在考虑如何将全文检索技术和为查询结果打分扩展到 XML 查询中来[CFF+02]。几种 XML 查询语言已经实现了对检索的兼容，[FKM00]为 XML-QL 扩展了关键词查询，并且给出了实验分析。XIRQL 是对 XQL 扩展，他支持谓词，关键词权重和最小结构抽象。XXL 搜索引擎[TW02]具有类 SQL 的语法，并且扩展了打分方法和本体的概念。但是总的说来，这些扩展了的 XML 查询语言并不适用于底层用户，因为查询语言的语法仍然很复杂。EquiX 语言[CKK+02]是对 XML 搜索引擎的一个简单扩展，但是 EquiX 仅仅能够处理带有 DTD 的 XML 文档。在[CMM+02]中，提出了另外一个 XML 检索语言，这种语言由 XML 文档片段组成，仅仅需要从查询到文档的近似匹配，然而，他们的查询结果是整个 XML 文档，而不是 XML 文档片段或元素，这和基于 HTML 的搜索引擎所查询出来的结果没有本质区别。另外有些研究成果还提出在为 XML 搜索结果打分的时候，应当像 IR 那样重点考虑空间距离，但是由 XML 文档本身的特性已经证明了关键词之间的距离仅仅是衡量搜索结果好坏的其中一个因素，其他方面比如元素之间的包含关系等必须考虑而且更加重要。

早期将结构化的 XML 查询与关键词检索相结合的研究工作还包括：Xyleme[ACW01]查询结构，K. Bohm 等人的在 HyperStorM 中做的一些研究，L. J. Brown 等在对象关系数据库中对结构化文本 ADT 的研究等，这些工作当时仅仅停留在设想和计划阶段。后来，[SKW01]专门为 XML 提出了 meet 算子，也就是返回最近似查询结果，他们同样提出了具体的算法用关系数据库方式的联接和索引来计算 meet 算子。DBXplorer[SSG02]和 DISCOVER[HP02]支持关系数据库上的关键词查询，但是不支持信息检索风格的评分，并且对模式（Schema）的要求严格，进一步说，他们并不适用于未给出固定模式的 XML 和 HTML 文档的检索。BANKS[BHN02]，DataSpot[DEG+98]和 Lore 支持图结构数据的关键词检索，其中 BANKS 用超链接结构，BANKS 和 Lore 利用简单近似度作为对查询结果的评定标准，他们的弱点也同样明显，他们不能进行 HTML 页面的检索，也没有充分利用 XML 文档本身的内部结构来评定查询结果。而且，DataSpot 没有提出任何查询评估的算法，Lore 仅仅能够返回预定义类型的查询结果，BANKS 因为过分依赖内存，使得它在处理大文档的时候效率低下，显然这是实用搜索引擎所不能容忍的。

在本节我们着重介绍的是公认各方面比较出色的 XML 搜索引擎框架

XRANK[GSB+03]。

XRANK 是 Connell 大学的研究者提出的用于 XML 搜索的引擎框架，它首先提出了在 XML 元素粒度进行搜索，把问题进行了明确定义，并且利用试验证实了这种方法的有效性。它的主要贡献在于：首先，形式化定义了问题，提出了系统结构；其次，提出了结合 XML 内部结构信息和传统 IR 技术的新的 rank 方法，同时考虑了超链接和包含关系两个方面；XRANK 还在具体实现中提出了新的倒排索引结构和相关算法；最后，XRANK 对几种索引结构的算法性能给出了试验比较，证明了算法的有效性。XRANK 不仅能够进行 XML 元素粒度的检索，还兼容平面 HTML 检索，这使得 XRANK 即能够检索 XML 文档又能检索 HTML 页面。值得一提的是，XRANK 不仅能够处理 XML 文档内部普通的父子包含关系，在处理 IDREF 和 XLINK 方面它也有一定优势，对于部分文档 内引用和文档间引用 XRANK 在评分方法上赋予了相应的权重，比较准确的反映了这类文档的内部结构关系。

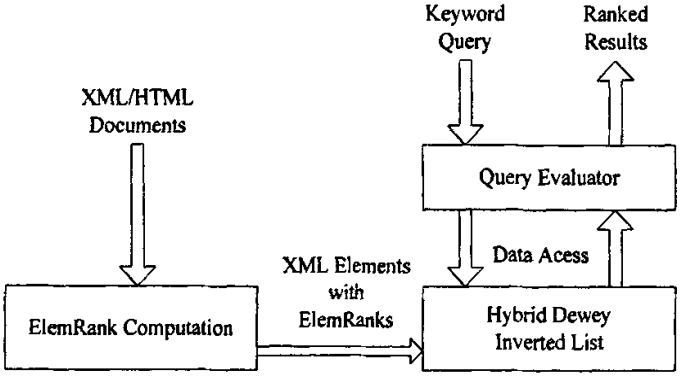


图 4.1 XRANK 系统结构图

图 4.1 是 XRANK 的系统结构图，从图中我们可以看到，对于给定的 XML 或者 HTML 文档以及查询关键词系统返回符合关键词查询的结果。需要指出的是， 信息检索中通常采用为每一个可能的关键词建立一个倒排索引，当用户提交包含关键词的检索以后，系统找到每个所提交的关键词对应的倒排表，并把找到的倒排表做联接，联接的结果就是用户需要的包含所有关键词的文档内容，这种方法在文本检索上是可行的，但是对于 XML 文档却是十分低效，XRANK 的改进倒排索引则依据了 Dewey 编码的特性，空间得到了很大的节省。Dewey 表示 XML 的方法在很多以前的研究方法中被采用，例如 H. V. Jagadish 等人在查询网络目录方面的工作，以及 I. Tatarinov 等提出的利用关系数据库存储和查询有序XML文档。XRANK 根据每个元素的 Dewey 编码、ElemRank 以及在文档中的位置，建立倒排索引，节省了存储空间，也提高了检索效率。但大量使用 Dewey 编码也带来一些问题，首先，频繁的逐段比较 Dewey 编码比较耗时；其次，存储 Dewey 编码也比较浪费空间。所以，本文提出了一种有效

的基于范围最小值查询的方法来解决 XML 关键词检索中的最低公共祖先问题。进一步地，我们将此关键词检索算法应用到我们的 Sheepdog 系统中，作为一个独立的检索模块存在。

4.3 问题定义

通常，一个 XML 文档 D 被建模成有序的带标记树 T(N,E)。其中树结点集 N 包含文档中的所有元素、属性或者值。而边集 E 表示元素之间的包含关系。这里简便起见，我们忽略了结点间可能的引用边。

给定关键词集合 $S=\{k_1, k_2, k_3, \ldots k_n\}$，对每个关键词 k_i 都会有一个结点集 S_i，里面的每一个结点都直接包含关键词 k_i。对每一种可能的结点组合 $\{e_1, e_2, \ldots, e_n\}$，其中 $e_i \in S_i$，都会有一个相应的 LCA 结点 v，即 $v=lca(e_1, e_2, \ldots, e_n)$。这里我们用 $lca(S_1,S_2,\ldots S_n)$来代表所有可能组合的 LCA 结点集。更进一步，按[GSB+03]中定义，对 $lca(S_1,S_2,\ldots S_n)$ 集合中的结点 v 来说，如果集合中没有其他结点 u 满足 $v < u$，(这里 $v < u$ 表示 v 是 u 的祖先)，则 v 是 $S_1,S_2,\ldots S_n$ 的一个 SLCA 结点，记做 $v=slca(S_1,S_2,\ldots S_n)$，所有这样的 v 结点即组成了查询结果集 R，记做 $R= SLCA(S_1,S_2,\ldots S_n)$。

根据问题定义，我们的工作首先引入 RMQ 算法来解决求解 LCA 的问题，然后提出非阻塞的算法来解决 SLCA 的问题。

4.4 基于 RMQ 的 LCA 算法

4.4.1 将 LCA 问题转化为 RMQ 问题

这一节中，我们将关注如何将 LCA 问题归结为 RMQ 问题。所谓 RMQ 问题，其描述如下。

定义（RMQ 问题）：对一个长度为 n 的数组，任意给出序号 i 和 j，返回在数组中值最小的元素的下标。

对此问题，根据下面这个 LCA 的关键性质，文献[BF00]中提出了可以将 LCA 归结为 RMQ 的算法。

性质：任意两结点 u 和 v 的 LCA 结点，是从 u 到 v 的深度优先遍历过程中遇到的深度最小的结点。

这里深度是指从根结点算起的路径长度，而且我们假设在树的优先遍历中，u 在 v 前面。这个性质是显而易见的。在从 u 到 v 的遍历过程中，必然会经过他们的 LCA 结点，而且该 LCA 结点必然是遍历过的结点中深度最小的。根据这条性质，利用欧

拉序列(Euler Sequence)来存储树的深度优先遍历序列。欧拉序列的定义如下：

定义（欧拉序列）： 在深度优先遍历树的时候，从根结点开始，每次经前向边或者后向边访问到的结点都记录下来，最后回到根结点，所得到的序列就是欧拉序列。

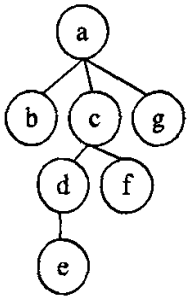


图 4.2 一个简单的 XML 片断

如图 4.2 中树的欧拉序列（用 E 表示）如表 4.1 所示，同时给出了对应的深度序列（用 L 表示）。

表 4.1 欧拉序列和深度序列

E	a	b	a	c	d	e	d	c	f	c	a	g	a
L	0	1	0	1	2	3	2	1	2	1	0	1	0

有了欧拉序列 E 和深度序列 L，现在我们可以描述基于 RMQ 的 LCA 算法。假设要求 F 和 G 的 LCA 结点。首先在 E 中分别找到两个结点第一次出现的位置 i 和 j，然后在 L 序列从 i 到 j（假设 i<j）的范围内找深度最小的值，该值所对应的结点即我们要求的 LCA 结点。为了方便找到 i 和 j，我们同时维护一个索引表 IndexTable (IT)，记录每个结点在欧拉序列中第一次和最后一次出现的位置（后面会用到）。示例的索引表如表 4.2。

表 4.2 索引表 T

Node	a	b	c	d	e	f	g
firstOccur	1	2	4	5	6	9	12
LastOccur	13	2	10	7	6	9	12

将基于 RMQ 的 LCA 算法应用到 XML 文档上，我们就得到了在 XML 上求解任意两个结点的 LCA 的算法，如图 4.3 算法 LCA 所示。

上面的欧拉序列 E 和深度序列 L 是在解析 XML 的过程中创建，考虑到为了支持大规模的 XML 文档，用占用内存比较小的 SAX 方式进行解析，其具体事件处理函数如图 4.4 所示。

LCA 算法

输入：u 和 v，直接包含关键词的两个结点；E：XML 树的欧拉序列；L：E 对应的深度序列

输出：u 和 v 的 LCA 结点

步骤：

1. 在索引表 IT 中查找 u 和 v 第一次在 E 中出现的序号设为 i 和 j；
2. 在 L 中 L[i]和 L[j]之间查找最小值所在的序号设为 k；
3. LCA = E[k]; 返回 LCA;

图 4.3 LCA 算法

```
Node; //结点模型;
Vector eulerSeq; //欧拉序列
Hashtable invertedIndex; //倒排表
Stack stack; //初始化空
subroutine startElement(){
    Node n = new Node;
    n.level = current.leve+1; //current 指示当前上下文结点，层次加 1
        if(eulerSeq.getLastNode() != n.parent)
    //父亲结点不在欧拉序列中就加入它
            eulerSeq.add(n.parent);
        eulerSeq.add(n); //添加 n 到欧拉序列
    //把 n 入栈，在 endElement 时弹出
        stack.push(n);
    }
    subroutine endElement(){
        Node n=stack.pop(); //弹栈
        if (eulerSeq.getLastNode()!=n)
        eulerSeq.add(n); //若结点已在欧拉序列末尾则不添加
        current = current.parent; //向上一层
    }
    subroutine characters(){
    for each word w {//进行分词，建倒排表
            Vector v = invertedIndex.get(key);
            if(!v.contains(temp))v.add(temp);
    }
}
```

图 4.4 SAX 解析事件响应函数

4.4.2 优化 RMQ 算法

从上一小节我们得出 LCA 问题可以通过 RMQ 问题来求解，现在考虑如何优化 RMQ 算法。

为了快速取得任意两个结点的 LCA 结点，最直观的方法是通过预处理，计算深度序列 L 中任意序号 i 和 j 之间取最小值所在的位置，并且用 $M[i][j]= argmin_{k=i\cdots j}\{L[k]\}$ 来存储。

假设结点总数是 n，那么就要预存一张 $O(n^2)$的表，存储空间耗费比较大。因此，虽然算法 LCA 能满足在 O(1)的时间查得 LCA 结点，但预处理的耗费太大。文献[BF00]给出了一种时间复杂度为<O(n), O(1)>的快速 RMQ 算法。这里用<P(n),Q(n)>来表示预处理时间 P(n)，查询时间 Q(n)。给出具体算法之前，先看一个用<O(nlogn), O(1)>的时间复杂性构建 Sparse Table 的算法。

与上述的 M[i][j]不同，这里 M[i][j]表示深度序列中从第 i 个位置开始，长度为 2^j 的范围中的最小值对应的序号，也就是 $M[i][j] = argmin_{k=i\cdots i+2^j}\{L[k]\}$。这样 i 的变化范围是从 1 到 n，j 的变化范围是从 1 到 logn，表的大小为 O(nlogn)。因此，构建此 Sparse Table 需要的时间复杂性 O(nlogn)，利用该数据结构进行查询，时间复杂性为 O(1)。算法示意图如图 4.5。

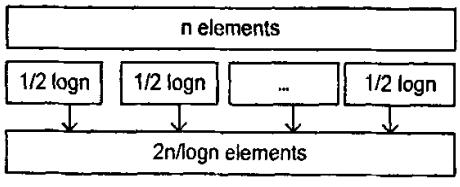


图 4.5 快速 RMQ 算法示意

4.4.3 XML 上的 SLCA 检索算法

有了求解两个结点 LCA 的算法，我们就可以构建 XML 关键词查询 SLCA ($S_1,S_2,\cdots S_n$)的算法。一种最直接的做法是对任意的结点组合$\{e_1, e_2, \cdots, e_n\}$其中 $e_i \in S_i$，都计算出它们的 LCA。对多个结点的 LCA 计算，可以按 $lca(e_1, e_2, \cdots, e_n) = lca(e_1, lca(e_2, \cdots, e_n))$公式计算，最终在所有 LCA 结点中去除掉是其他结点祖先的结点后，就得到了最终解，即 $SLCA(S_1,S_2,\cdots S_n) = removeAncestor\{lca(e_1, e_2, \cdots, e_n)\}$。显而易见，这种方法时间耗费太大，规模为 $O(kd|S_1|\cdots|S_n|)$，其中 d 是求解两个编码共同前缀的代价，一般正比于编码的长度，即树的深度，k 为结点集合的个数，$|S_i|$为集合 S_i 的大小。[GSB+03]基于下面四个性质，提出了复杂度为 的 Indexed Lookup Eager 算法，其中|S1|(|S|)是

S_1 到 S_n 中最小(最大)的集合。我们通过分析和证明，说明基于 RMQ 的 LCA 算法同样具有这样的性质。

性质 1：

$$slca(\{v\},S)=\{descendant(lca(v,lm(v,S)),lca(v,rm(v,S)))\}$$

此性质说明，在求结点 v 和结点集 S 中结点的 slca 时，只要比较从 v 和 S 中最靠近 v 的两个结点得到的 lca 即可。其中，函数 lm(v, S)返回在 S 中前序遍历顺序在 v 前面（左边）最接近 v 的结点，由于结点采用 Dewey 编码，所以就是返回 S 中小于或者等于 v 的最大的编码。函数 rm(v, S)则是返回大于或等于 v 的最小的编码。函数 desendant(v_1, v_2)则是返回两个结点里的子孙结点。

这条性质基于两个很明显的观察，即若

A. $pre(v) < pre(v_1) < pre(v_2)$，则 $lca(v, v_2) < lca(v,v_1)$;

B. $pre(v_2) < pre(v_1) < pre(v)$, 则 $lca(v, v_2) < lca(v,v_1)$

性质 2：

$$slca(\{v\},S_2,...,S_k)=slca(slca(\{v\},S_2,...S_{k-1}),S_k)$$

性质 3：

$$slca(S_1,...S_k)=removeAncestor(\bigcup_{v\in S_1} slca(\{v\},S_2,...S_k))$$

这两条性质比较直观，性质 2 给出了处理多关键词的方法；性质 3 指出需要过滤掉不是最终结果的操作。

基于这三个性质，就得出了一个对 XML 文档计算 SLCA 的算法，但这个算法要等待所有的 SLCA 结点算出来之后才能去除其中的祖先结点。SLCA 的非阻塞算法[XP05]，解决了这个问题。非阻塞算法建立在两个定理之上。

定理 1：给定两个结点 v_i 和 v_j 和集合 S，如果 $pre(v_i) < pre(v_j)$并且 $pre(slca(\{v_i\}, S)) > pre(slca(\{v_j\}, S))$，则 $slca(\{v_j\},S) < slca(\{v_i\},S)$。

定理 2：给定两个结点 v_i 和 v_j 和集合 S，满足 $pre(v_i) < pre(v_j)$并且 $pre(slca(\{v_i\},S)) < pre(slca(\{v_j\},S))$，如果 $slca(\{v_i\},S)$不是 $slca(\{v_j\},S)$的祖先，则对所有满足 $pre(v) > pre(v_j)$的 v，$slca(\{v_i\},S)$不可能是 $slca(\{v\},S)$的祖先。

定理中 pre(v)代表结点 v 的先序遍历顺序。定理 1 说明如果结点 v_i 先序遍历顺序出现在 v_j 之前，并且 $slca(\{v_i\},S)$的先序遍历出现在 $slca(\{v_j\},S)$后，则说明 $slca(\{v_j\},S)$是 $slca(\{v_i\},S)$的祖先结点，可以在计算中丢掉。子函数 get_slca 中第 5 行应用了定理 1。定理 2 则尽可能早的判定出了可以输出的结果结点，应用一个 buffer B 可以很好的利用定理 2(见查询算法)。

统观这些性质和定理，本质上都依赖先序遍历顺序的比较，例如计算函数 lm 和 rm 时 Dewey 编码的比较。而我们在索引表 IT 中记录的各元素在欧拉序列中第一次出

现的位置 firstOccu 也体现了这个次序，所以这些操作都能方便实现；另外，求解 removeAncestor 和 descendant 等需要判定祖先关系的操作时，[GSB+03]同样要依赖比较前缀编码，而我们借助 IT 表可以轻松完成。因此，我们得到性质 4 如下。

性质 4：

若结点 u 满足 u.firstOccu < v.firstOccu < u.lastOccu，则 u 是 v 的祖先。

由此，在[XP05]中工作的启发下，我们提出了基于 RMQ 的 LCA 非阻塞算法，改进后的算法如图 4.6 所示。相比原算法，原来比较耗时的核心操作如 lca，descendant，pre 的比较等已经由 dewey 编码比较变成了查表和简单整数的比较，故而可以大大加快速度。

```
算法：非阻塞的 SLCA 算法
预处理阶段：
1.  根据 2.1 节解析一个 XML 文档
2.  根据 2.2 节快速 RMQ 算法创建 sparse table 和块内表等中间结果
查询算法：
设置一个可以容纳 P 个结点的 buffer B
1.  v = null
2.  while (S1 has more nodes){
3.      load P nodes of S1 into B
4.      for I=2 -> k
5.          B = get_slca(B, Si)
6.      if(v≠null && firstNode(B) ≤ v)
7.          removeFirstNode (B);
8.      if (v≠null && v!< firstNode(B))
9.          output v
10.     v = removeLastNode(B)
11.     output B; B = empty
12. }
13. output v
subroutine get_slca(S1, S2)
// S1 and S2 is sorted by firstOccu
1.  set result R empty
2.  u = 0;
3.  for each node v∈S1 {
4.  x=descendant(lca(v,lm(v,S2)), lca(v,rm(v,S2)))
5.  if(u.firstOccu≤ x.firstOccu)
6.      if (u.lastOccu<x.lastOccu)
7.          R = R∪{u}
8.      u = x
9.  }
10. return R∪{u}
```

图 4.6 非阻塞的 SLCA 算法

由上易知整个算法的时间复杂度为 $O(|S_1|\sum_{2}^{k}\log|S|+|S_1|^2)$。

4.5 Sheepdog 系统中 XML 关键词检索模块

4.5.1 关键词检索模块示意图

如图 4.7 所示，输入的每个 XML 文档在被解析时，生成每个结点的欧拉顺序和层次，并且记录在一个多维数组中；同时创建索引表 IT。欧拉序列和索引表 IT 经由快速 RMQ 算法预处理，生成的包括所有等价块的块内表，新的序列等，把这些数据结构和索引都存放在磁盘上。另外，针对每个单词，包含这个单词的结点被建成倒排索引，也被储存到磁盘上。数据存储采用 Berkeley DB 的 B 树结构。

查询模块提供了缓存，cache 中保存了比较频繁的查询条目，用户的关键词条目会首先在 cache 中进行查找，如果命中，就直接返回结果。如果不命中，就通过引擎中的 SLCA 算法进行查询。

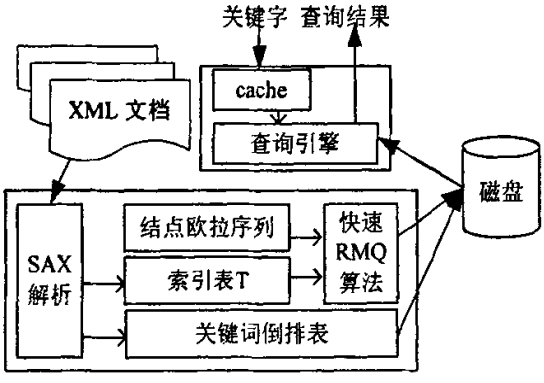


图 4.7 Sheepdog 关键词检索模块图

4.5.2 关键词查询界面

关键词查询界面如图 4.8 所示，用户输入关键词，比如“city Genova”，用户想要查找有关城市 Genova 相关的信息，以空格作为几个关键词的分隔符，然后点击“keyword search”按钮，结果显示在右边面板中。此时返回结果表示一个 city 元素，城市的名称是 Genova，城市的人口是 701032 等其他信息。可以看到此结果覆盖了查询请求中的两个关键词。

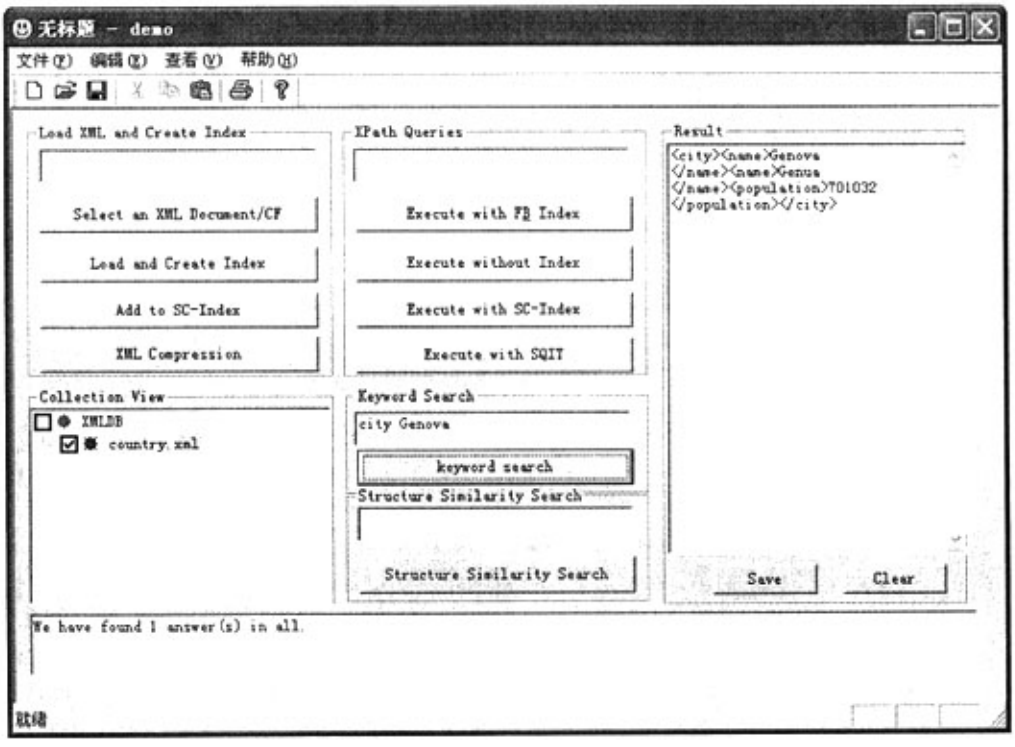


图 4.8 关键词查询界面

4.6 实验分析

实验在一台配备 Intel Pentium4 1.80GHz 处理器，512MB 内存的微机上进行了测试，数据集包括 DBLP 数据集、 SIGMODRecord 数据集和华盛顿大学提供的数据集，例如：包括 Mondial、Nasa 和 Auction；数据集的大小和特征信息见表 4.1。我们主要做了三组实验，分别进行算法性能实验和对比试验。我们采用的对比实验方法是 SIGMOD2005 上的 Xksearch 方法，该系统提出借助 Dewey 编码来求解 LCA，采用了非阻塞的 SLCA 算法，我们用 java 实现了该系统。

表 4.1 实验所用数据集

数据集	大小(MB)	最大深度	平均深度
Reed	0.252	4	3.199
Sigmod	0.458	6	5.141
Mondial	1.702	5	3.593
Nasa	2.430	8	5.580
Dblp	5.095	6	2.902
Auction	9.018	5	3.756

表 4.2 与 xksearch 的时间开销进行比较

数据集	时间开销 (ms)		
	xksearch	本文算法	节省比值
Reed	16.0	15.5	3.1%
Sigmod	27.5	15.5	43.6%
Mondial	23.5	15.5	34.0%
Nasa	31.5	15.5	50.8%
Dblp	23.5	15.5	34.0%

表 4.3 与 xksearch 的空间开销进行比较

数据集	空间开销 (MB)		
	xksearch	本文算法	节省比值
Reed	1.12	1.09	2.8%
Sigmod	2.06	1.83	11.3%
Mondial	1.92	1.82	5.2%
Nasa	17.22	13.53	21%
dblp	25.32	24.86	1.8%

第一组是观察不同数据集上我们的算法跟 Xksearch 的算法在查询时间和存储空间开销的差异。此时查询关键词都为 3 个，每个关键词的频率都是 100（实际是 ），查询是随机从文档中的关键词集（包括 XML 元素的 tag 关键词和文本中的关键词）中进行抽取的，对同一数据集的查询是相同的。查询结果如表 4.2 和表 4.3 所示。

表 4.2 列出了 5 个数据集上的时间测试结果。表中的查询时间是查询执行时间（预处理时间不计在内）。表 4.3 是对应数据集的存储开销，包括所有预处理阶段存储在磁盘上的数据。从两图可以看出，本文算法在查询时间上和存储的空间上都要优于 xksearch。时间上的性能提升更为明显，这是因为基于 RMQ 的 LCA 求解，省去了传统算法在逐段比较求解共同前缀的时间开销（比如 lm，rm，祖先关系 < 的判定，共同前缀的求解等比较耗时）。当 XML 文档的平均深度较大时，比如数据集 nasa，xksearch 方法的开销更为巨大，实验结果差距也更为明显。同时可以看到本文算法所用时间基本相同，这也跟我们算法的时间复杂度和 XML 文档的平均深度无关相符合。

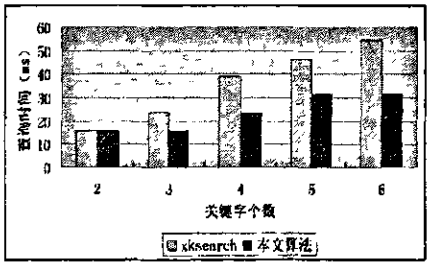


图 4.9 关键词个数对查询时间的影响

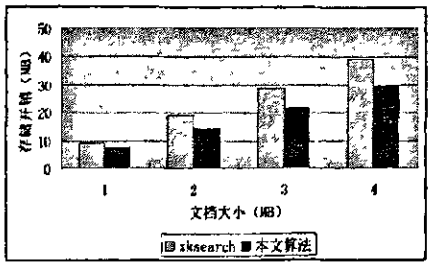


图 4.10 文档大小对存储开销的影响

第二组实验是比较关键词个数对查询时间的影响。实验所用的数据集是 dblp，查询关键词由 2 个增加到 6 个，每个关键词的频率都是 100。实验结果如图 4.9，可以发现在关键词增长的时候相比 xksearch 方法本文算法的查询时间增长比较缓慢。第三组实验主要观察 XML 文档大小对算法空间开销的影响。此空间开销主要指预处理阶段系统要存储的数据及中间结果。此组实验所用的数据来自与表 4.1 中的 auction。我们从该数据集得到大小分别为 1M、2M、3M 和 4M 的四个 XML 文件。实验结果如图 4.10 所示。可以发现，存储开销随文档增大而增大，但要比 xksearch 开销要小。

4.7 小结

本章提出在XML 关键词检索中应用基于 RMQ 的 LCA 求解算法,通过有效的预处理，可以消去 XML 上关键词检索时借助 Dewey 编码求解共同祖先的时间系数 d，同时由于仅存储结点的欧拉序列而不是 Dewey 编码，可以有效减少存储空间的开销。本章进一步证明此算法可以适用于文献[XP05]提出的 XML 上关键词检索的非阻塞算法，并且经实验证明我们的 Sheepdog 系统中的 XML 上关键词查询模块十分高效。

第五章 Sheepdog 中结构相似度查询技术

在以容器形式存储大量 XML 文档的原生 XML 数据库中，有时需要寻找和给定 XML 文档结构相似的文档（在一些应用比如数字图书馆、XML 订阅发布系统中尤其有用），这是 XML 数据库上的另一项重要的查询。以往的工作通过两个 XML 文档的树结构模型的编辑距离（edit distance）来衡量文档的相似度，但因为编辑距离计算量非常大，最近的一些工作提供了用精简的概要结构来计算编辑距离，但它们忽略了 XML 树中不同的结点可能有很大的不同，因此对所有结点采用相同的编辑操作代价是不合适的。

本章主要介绍了 Sheepdog 系统中结构相似度查询模块的实现。通过提出一种概要结构来使得计算编辑距离更为合理；进一步地，引出采用带权重的方案来区别不同的结点的重要性，同时，编辑距离的代价模型也因此而改变。同以往技术相比，这种方法对近似求解最相似 K 个文档的问题更为有效。实验证明，这种方法是有效而实用的。

本章结构如下，第 5.1 节介绍此类查询的相关工作和应用背景；第 5.2 节介绍像编辑距离等一些基本定义；第 5.3 节介绍本章的主要内容——基于新的概要结构的编辑距离算法；最后第 5.4 节给出了实验部分，第 5.5 节给出了总结。

5.1 相关工作

大多数原生 XML 数据库都有容器的概念，大量的同构的或者异构的 XML 文档被放在一个容器中，用户可以对容器提交查询。除了前面提到的 XPath 查询和关键词查询外，寻找与给定文档结构最相似的文档也是一项重要的查询（比如在大量用 XML 格式表示的电子病历中，寻找与给定病人最相似的病例）。在将 XML 文档建模成带根有序的标记树之后，相似的文档有相似的树结构。因此，寻找相似性就转化为计算树结构的相似度。但是，树结构的比较是非常昂贵的操作，所以需要一些简化有效的算法。

目前在树的相似度的计算上已经有一些相关工作，如[RPK05, K79, DJK+94, DPA+04]。他们所使用用来描述树之间的差别都是编辑距离这一标准。一棵树可以通过一个编辑操作序列（插入结点、删除结点和更名结点）转化为另一棵树。每一个编辑操作都被赋值一个非负的代价，所有操作的代价和被称为整个变换序列的代价。对给定的两棵树，可能有众多的变换序列，其中最小的编辑代价被成为它们的编辑距离。

[KD89]中的算法是目前计算树的编辑距离中最好的，在下文中简称 ZS 算法。他们的算法是严格的编辑距离，要判断源树和目的树的每一个结点对，计算的时间复杂度 O(|M||N|depth(M)depth(N))。因此计算代价昂贵在现实应用中是无法接受的。因此一些近似的算法被提出，使得计算过程迅速实用。这些工作关注在如何更改或限制允许的编辑操作。不同于他们，[TTK04]提出另一个途径，他们通过消去嵌套和重复元素来获得树的概要结构，然后再在概要结构上进行计算编辑距离。虽然时间复杂度是 O(|M||N|)，但由于是在概要上进行计算，计算代价已经降低了很多。但是，它的代价就是降低了计算结果的准确度。

本章的主要贡献如下：

- 不同于已有工作，在计算结构相似度时引进权重因子，提出了两种权重策略。实验证明，权重信息对相似度查询有很大帮助。
- 在上面基础上，提出了一个获取 XML 树概要结构的算法。不同于[TTK+04],我们消除重复结点的时候保留了他们的权重信息，这样使得我们的近似算法准确度较高。
- 提出了一个在带权重结点上进行编辑操作的代价模型，重新给出了一个新的编辑距离定义 WD，即带权重的编辑距离。并将其应用在 Top-K 相似度查询的计算当中并实现在 Sheepdog 系统中，实验结果证明这种方法的效率较高。

5.2 问题定义

首先，编辑距离中的三种编辑操作（插入、删除、更改）的详细定义如下：

1. 插入：插入一个新的结点。
2. 删除：删除一个已有结点。
3. 更改：改变已有的一个结点成为另外的结点。

举例如图 5.1，展示了如何将左边的树经过以上三种编辑操作变化到右边的树。其中，虚线连接的两个结点如果结点名称一样，则不需要任何操作；如果结点名称不同，则需要更改操作。在左树中没有虚线连接的结点表明右树中这些没有相应的结点，应该删除操作来删除；同理右树中没有虚线连接的结点表明它们应该被添加到左树。这些虚线将源树和目的树对应起来，从而构成了一个映射。每个映射对应一个编辑序列，每个编辑序列都有一个代价，显然，这样的映射可以有很多，而最小的那个代价我们称作两棵树之间的编辑距离。寻找这样的一个最小代价一般采取动态规划的算法。

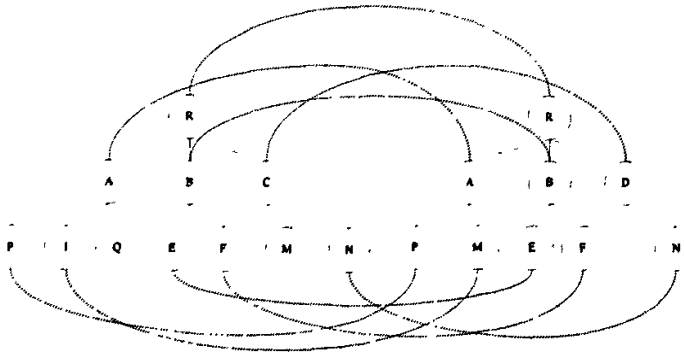


图 5.1 编辑操作

其次，为了在我们的编辑代价模型中引入权重信息。我们首先考虑如何给树上的不同结点赋值权重信息。这里我们考虑了两种权重函数：线性权重和指数权重。考虑一棵树 T，用 depth(T)表示树的深度，level(n)表示结点 n 的层次，则结点的线性权重函数表示如下：

$$weight(n) = 1 - \frac{level(n)}{depth(T)}$$

对于指数权重，我们假设树的深度是 d，即 depth(T)=d，我们赋值权重 r^d 给根节点，然后第 i 层的结点的权重是 r^{d-i}。其中 r 是个可调的参数，用来反应结点层次的重要性。指数权重函数表示如下：

$$weight(n) = \gamma^{depth(T)-level(n)}$$

这两个权重函数的区别见图 5.2 中同一棵树的结点权重值，左侧对应线性权重，右侧对应指数权重。权重函数的目的是为了反应不同层次的结点有不同的重要性，对他们的编辑代价就不应该完全相同。当然，还可以定义其他的权重函数，这里给出的两个是比较典型的。对于他们的效果会在实验部分给出。

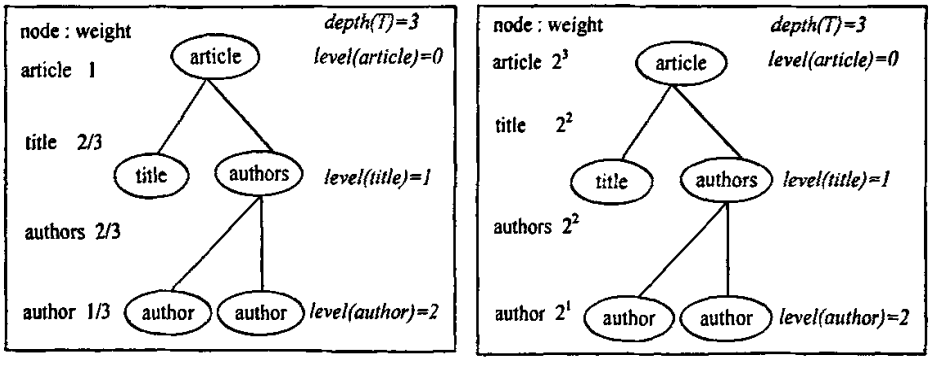
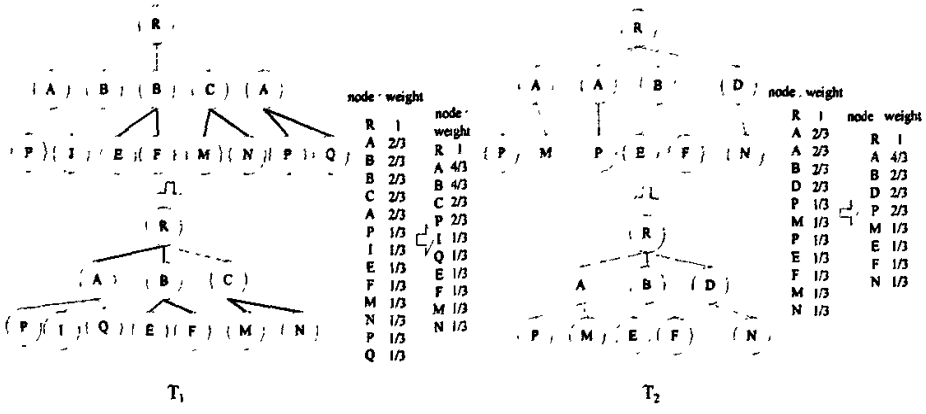


图 5.2 线性权重函数和指数权重函数

5.3 概要结构和带权重的编辑距离计算

5.3.1 概要结构

从以上可知，对于原树结构采用严格的编辑距离算法是代价昂贵的，因此我们尝试去得到能够比较准确反应原树信息的概要结构来计算编辑距离。因为XML 文档通常含有路径信息相同的的结点，我们可以压缩这些结点。用一个结点来代替，而其权重信息，则积累到这一个结点里面。如图 5.3，给出了树 T_1 和 T_2 以及他们经过压缩后的概要结构。要注意的是，重复的结点去掉了，但是他们的权重信息保留在剩余的结点上。这里用到的权重函数是上面提到的线性权重函数。



T_1 T_2

图 5.3 概要结构

5.3.2 带权重的编辑距离计算

以往的工作在计算编辑距离时认为所有编辑操作的代价都是单位 1，有了上述的权重信息后，我们重定义三种编辑操作的代价如下：

$$insert:\gamma(\wedge \to v) = weight(v)$$
$$delete:\gamma(u \to \wedge) = weight(u)$$
$$relabel:\gamma(u \to v) = \begin{cases} |weight(u) - weight(v)| & (u.label = v.label) \\ |weight\{u\} + weight(v)| & (u.label \neq v.label) \end{cases}$$

其中 γ 表示编辑操作的代价，$\wedge$ 表示不存在的结点。

定义（带权重的编辑距离）：给定两棵树 T_1 和 T_2，设 $S(T_1)$和 $S(T_2)$分别是他们的概要结构，用 $ED(T_1, T_2)$表示树 T_1 和 T_2 的编辑距离，用 weight(T)表示树 T 的所有结点权重和，则带权重的编辑距离 $WD(T_1, T_2)$表示为：

$$WD(T_1,T_2)=\frac{ED(S(T_1),S(T_2))}{weight(T_1)+weight(T_2)}$$

拿图 5.3 中的两棵树来举例说明如何计算 WD。将 T_1 的概要结构转换到 T_2 的概要结构需要如下的编辑操作：(relabel(B,B),4/3-2/3)，(relabel(C,D),2/3+2/3)，(delete(I), 1/3)，(delete(Q),1/3)，(insert(M), 1/3)和(delete(M), 1/3)。括号里面前面一项表示操作，后一项表示操作的代价。整个的操作代价是 10/3。注意到如果按传统的方法，像 relabel(B,B)的操作代价应该是 0，在我们的计算中，我们把一个结点看成是一些结点集，所以虽然结点一样，也会产生不为 0 的操作代价。再由 $weight(T_1)=21/3$，$weight(T_2)=17/3$，得到最后的带权重的编辑距离是（10/3）/（21/3 + 17/3)=10/38。这个最后的除法是归一化的过程，使得最后的结果值在[0,1]的区间内。

5.3.3 用带权重的编辑距离回答结构相似度 Top-K 查询

有了这样一个距离标准，我们可以按如下步骤回答 XML 文档集中的结构相似度问题的 Top-K 查询。

1. 对所有 XML 文档创建树模型。
2. 计算所有树的概要结构，同时计算概要结构上的结点权重信息。
3. 使用带权重的编辑距离计算作为 XML 距离的标准，可采用任意的 Top-K 算法来回答该查询。

5.4 实验分析

我们在真实数据和模拟数据上比较了[RPK05]的严格的编辑距离算法和我们 Sheepdog 系统中的结构相似度检索模块中的带权重的编辑距离算法。实验在一台配备 Intel Pentium4 3.20GHz 处理器，2GB 内存的微机上进行了测试。真实数据使用来自于[URL]的 XML 数据，虚拟数据用 IBM 的 XML Generator[AD99]生成。我们用最简单的算法来求解 Top-K 问题，默认求取前 10 个结构上与给定文档最相近的文档。为了衡量实验的效果，我们假设用[PRK05]的算法得到 m 个结果，用我们的算法得到 n 个结果，如果交集是 q 个，我们使用 q/n 来表示我们结果的准确度。由于我们只关心我们找到的结果中有多少个是合适的，所以没有

采用信息检索领域的“准确率”和“召回率”。

5.4.1 时间性能

算法时间性能的比较是在由 SigmodRecord.dtd 生成的虚拟数据上进行的，我们观察随着输入树规模的增大，算法的执行时间的变化。数据集有 1000 个文档，平均结点数目 35，最大值 78。我们改变输入树的大小，并使用 CPU 时间来作为所用时间的标准。实验对比图如图 5.4 所示。这里时间花费包括建树时间和求取编辑距离时间。对每个查询，重复 10 次求其平均值作为查询时间。从图上可以看出，当树的结点数目增多的时候，[RPK05]的算法时间快速增加，而我们的算法执行时间基本不变化。这是由于我们算法中计算的是概要结构的编辑距离，而概要结构的规模几乎不变化，所以算法执行的时间也几乎不变化。

图 5.5 中展示了随着文档数据集规模增大时算法执行时间的变化。我们从 3 个分别有 1000 个文档的数据集中取出等同数量的文档，组成大小分别为 300，600，1500，3000 的数据集。从图中可以看出，随着数据集的增大，平均时间开销都线性增长，但是我们算法的时间开销要远小于严格距离计算下的时间开销。

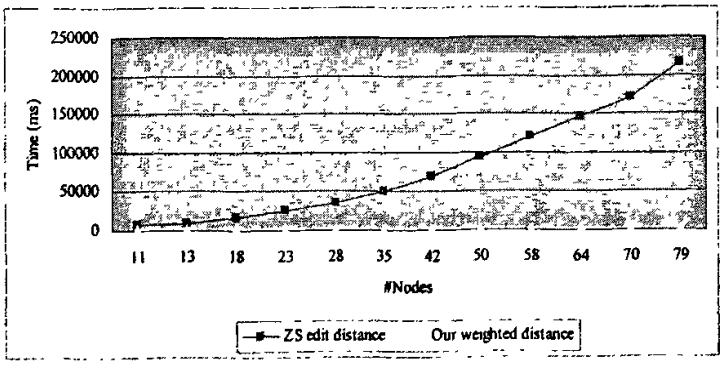


图 5.4 时间随输入 XML 树大小的变化

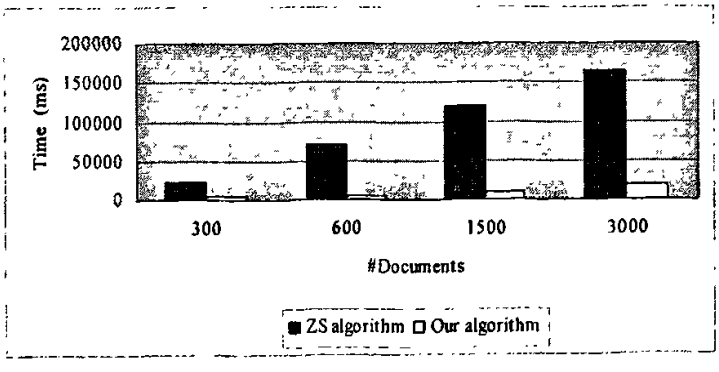


图 5.5 时间随数据集大小的变化

采用信息检索领域的“准确率”和“召回率”。

5.4.1 时间性能

算法时间性能的比较是在由 SigmodRecord.dtd 生成的虚拟数据上进行的，我们观察随着输入树规模的增大，算法的执行时间的变化。数据集有 1000 个文档，平均结点数目 35，最大值 78。我们改变输入树的大小，并使用 CPU 时间来作为所用时间的标准。实验对比图如图 5.4 所示。这里时间花费包括建树时间和求取编辑距离时间。对每个查询，重复 10 次求其平均值作为查询时间。从图上可以看出，当树的结点数目增多的时候，[RPK05]的算法时间快速增加，而我们的算法执行时间基本不变化。这是由于我们算法中计算的是概要结构的编辑距离，而概要结构的规模几乎不变化，所以算法执行的时间也几乎不变化。

图 5.5 中展示了随着文档数据集规模增大时算法执行时间的变化。我们从 3 个分别有 1000 个文档的数据集中取出等同数量的文档，组成大小分别为 300，600，1500，3000 的数据集。从图中可以看出，随着数据集的增大，平均时间开销都线性增长，但是我们算法的时间开销要远小于严格距离计算下的时间开销。

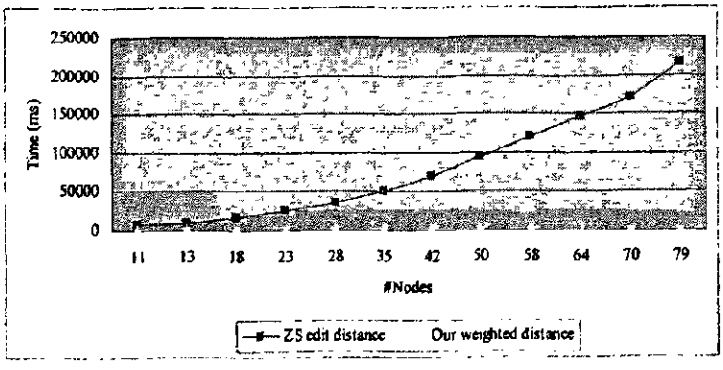


图 5.4 时间随输入 XML 树大小的变化

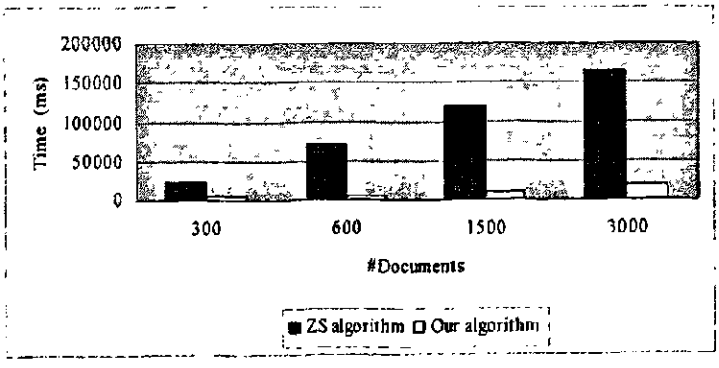


图 5.5 时间随数据集大小的变化

5.4.2 准确度

我们在真实数据上比较了两个算法的准确度。从 SigmodRecord 数据集，我们选出 51 个根元素为 issue 的 XML 文档，从 Mondial 数据集，我们得到 231 个根元素为 country 的文档，前者平均结点数目是 87，后者平均是 57。用上述定义的准确率我们比较两个算法的时间效率和准确度。

采用线性权重函数计算的结果对比如表 5.1。可以看出，我们的算法活得较高的准确度，同时时间消耗要远小于[RPK05]的算法。对于 SigmodRecord 数据来说，时间可以节省高达 98%。这是因为该数据集重复元素较多，概要结构要比原树规模小很多。对 Mondial 数据，虽然没有这么明显，但越接近原树，活得的准确率也就越高。

表 5.1 时间和准确度比较

	ZS algorithm		Our algorithm			precision
	Time (ms)	Σm	Time (ms)	Σn	Σq	
Sigmod (51 docs)	18490	108	344	103	85	0.825
Mondial (231 docs)	6592	192	707	112	107	0.955

在上述试验中，我们假设[RPK05]中编辑操作的代价是单位 1。为了观察新的代价模型对结果的影响，我们同样添加了权重信息到该算法中并重做实验。表 5.2 展示了这一次的实验结果。可以看出，加了权重信息之后，[RPK05]的算法准确度就提高了很多，这说明我们引入权重信息在编辑距离的计算中还是有很大作用的。而且，可以看出，指数权重函数能获得更好的效果。

表 5.2 添加权重信息之后的[RPK05]算法

Linear Weight Function				
	Σm	Σn	Σq	precision
SigmodRecord (51 docs)	103	103	91	0.883
Mondial (231 docs)	132	112	109	0.973
Exponential Weight Function				
SigmodRecord (51 docs)	103	103	93	0.912
Mondial (231 docs)	131	112	111	0.991

5.5 小结

本章提出在如何快速回答 XML 上的结构相似度查询。已有的工作使用编辑距离的同时忽略了结点层次信息的重要性。从语义和结构上看，我们认为结点在不同层次有不同的重要性。所以我们提出两种权重函数来给结点添加权重信息，然后计算 XML 文档概要结构的带权编辑距离，这样一方面大大减少了算法的执行时间，另一方面能够尽量增加算法的准确度。实验结果也证明了我们的方法的有效性。作为原生 XML 数据库 Sheepdog 系统中一种重要的查询，弥补了现有 XML 数据库尚无此种查询实现的不足。

第六章 总结与展望

本文首先介绍了 XML 方面和原生 XML 数据库方面发展和应用的背景，介绍了原生数据库方面的研究现状和本文的主要研究内容：在研究现有原生 XML 数据库产品的基础上，设计和实现了一个原生 XML 数据库——Sheepdog，并在其上实现了高效的关键词检索查询和结构相似度查询。

本文第二章介绍了 XML 及原生 XML 数据库的一些基本概念和相关知识。

本文第三章介绍了 Sheepdog 系统的功能特性和系统架构，并着重介绍了系统存储模块的设计。然后本文第三章和第五章分别重点介绍了 Sheepdog 系统中两个查询模块：关键词查询模块和结构相似度查询模块。第四章介绍了 XML 关键词检索相关工作，然后提出了用基于 RMQ 实现的快速 LCA 算法来改进 XML 关键词检索算法的效率。第五章介绍了结构相似度查询的相关工作，然后介绍了我们基于带权重的概要结构和在其上进行的编辑距离算法，并据此实现的 Sheepdog 系统中结构相似度查询模块。这两章都给出了具体的实验结果，并进行了详细的分析。

总结本文的贡献和创新之处如下：

1）高效的 XML 存储技术。本文设计和实现了一个原生 XML 数据库——Sheepdog，能够有效地支持 XML 文档的存取。

2）在 Sheepdog 系统中实现了一种高效的 XML 数据上关键词检索的算法。同已有的根据前缀编码来求解树上最小公共祖先的算法相比，该算法能够减少前缀比较和存储带来的开销，能够节省很多检索时间。

3）在 Sheepdog 系统中实现了一种高效的结构相似度查询算法，丰富了原生 XML 数据库的查询方式，弥补了现有数据库产品的不足。

最后，在 XML 的研究领域里还有几个方面应该值得关注，其中有些也会成为本文未来的工作：

1）如何在 XML 存储系统中支持 XML 数据的更新。一个完善的数据库系统要能够很好的支持数据的更新，在当前系统中，由于采用 dewey 编码，导致不能支持 XML 数据的更新，这将是未来待解决的一个重点问题，这其中包括对于某些问题可行性的判定。

2）单纯的依靠关键词进行查询和单纯的依靠 XML 查询语言 XPath、XQuery 都不能很好的进行全文检索。我们将考虑如何将二者进行有机结合，更好地实现 XML 数据的全文检索。

综上所述，对 XML 数据存储和查询研究是任何时候 XML 数据管理所面临

的主要问题。XML 存储的技术的发展能使原生 XML 数据库就像关系数据库一样，扎实而生命力长久，他也是其他关于 XML 研究的基石。而高效的查询技术更是原生 XML 数据库极为重要的部分，而对 XML 检索技术的研究需要融合传统的信息检索技术和数据库技术，建立新的理论框架，并最终开发出如 google 之于 HTML 检索的高效的之于 XML 的检索引擎。同时，对 XML 的转换、发布、XML 视图等的研究都具有现实意义。这些研究将构成下一代 XML 数据库（Web 信息管理系统）的基础。

参考文献

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. Data on the Web: from relations to semistructured data and XML. Morgan Kaufmann, 1999.

[ACW01] V. Aguilera, S. Cluet, F. Wattez, Xyleme Query Architecture, WWW Conf., 2001.

[AD99] A. Luis Diaz, D. Lovell. XML generator. 1999. http://www.alphaworks.ibm.com/tech/xmlgenerator

[BF00] M. A. Bender, and M. Farach-Colton. The LCA Problem Revisited. Proceedings of the 4th Latin American Symposium on Theoretical Informatics, 17: 88--94, 2000.

[BHN02] G Bhalotia,A Hulgeri,C Nakhe,et al.eyword searching and browsing in databases using BANKS. The 18th ICDE, San Jose,USA,2002

[Blo70] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422–426, 1970

[BM99] Catriel Beeri, Tova Milo: Schemas for Integration and Translation of Structured and Semi-structured Data. ICDT 1999: 296-313

[Bou99] Ronald Bourret: XML and Databases. http://www.rpbourret.com/xml/XMLAndDatabases.htm

[BPS+98] T. Bray, J. Paoli, C. Sperberg-McQueen, et al. Extensible Markup Language (XML) 1.0. W3C, 1998.

[CD99] J. Clark and S. DeRose: XML path language (XPath). W3C Recommendation 16 November 1999, http://www.w3.org/TR/XPath, 1999.

[CFF+02] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases. www.w3.org/TR/2002/WD-xmlquery-use-cases-20020816

[CKK+02] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. EquiX: A search and query language for XML. Journal of the American Society for Information Science and Technology, 53(6): 454-466, 2002.

[CMK+03] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for XML. In VLDB, pages 45—56, 2003.

[Cod70] E. F. Codd: A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13(6): 377-387, 1970

[CSF+01] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, Moshe Shadmon. A Fast Index for Semistructured Data. In Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

[DD99] De Rose, S., Daniel Jr., R.: XML Pointer Language (XPointer), W3C Working Draft 9-July-1999, W3C, http://www.w3.org/TR/xlink, Juli 1999.

[Dea99] Stephen Deach. Extensible Stylesheet Language (XSL) Specification W3C Working Draft. Technical report, World Wide Web Consortium, 21 April 1999.

[DEG+98] S Dar,G Entin,S Geva,et al.DTL's DataSpot:Database exploration using plain language. The 24th Int'l Conf on VLDB, New York,1998

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, and et al. XML-QL: A Query

Language for XML. W3C Note, 1998. Available: http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/

[DFS99] A. Deutsch, M. Fernandez, D. Suciu. Storing Semistructured Data with STORED. In Proceedings of the 28th SIGMOD International Conference on Management of Data, May, 1999.

[DOT99] DeRose, S., Orchard, D., Trafford, B.: XML Linking Language (XLink), W3C Working Draft, December 1999. http://www.w3.org/TR/xlink.

[FK99] Daniela Florescu, Donald Kossmann. Storing and Querying XML Data Using an RDBMS. Data Engineering Bulletin, Vol. 22, No. 3, 1999.

[FKM00] Daniela Florescu, Donald Kossmann, Ioana Manolescu: Integrating keyword search into XML query processing. WWW9 / Computer Networks 2000. 33(1-6): 119-135.

[GSB+03] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In SIGMOD, 2003

[GW97] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Proceedings of 23th International Conference on Very Large Data Bases (VLDB), Athens, Greece, August 1997, 436-445.

[HP02] V. Hristidis, Y. Papakonstantinou. DISCOVER: Keyword search in relational databases[C].The 28th Int'l Conf on VLDB, Hong Kong,2002

[K79] Tai, K: The Tree-to-Tree Correction Problem. J. of the ACM. No. 3. (1979) 422—433

[K96] Zhang, K.: A constrained editing distance between unordered labeled trees. Algorithmica (1996) 205—222

[KM00] Carl-Christian Kanne, Guido Moerkotte. Efficient Storage of XML Data. In Proceedings of the 16th International Conference on Data Engineering, 28 February-3 March, 2000, San Diego, California, page 198.

[KRD92] Zhang, K., Statman, R., Shasha, D.: On the editing distance between unordered labeled trees. Inf. Process. Lett. (1992) 133—139

[LWY+02] Hongjun Lu, Guoren Wang, Ge Yu, Yubin Bao, Jianhua Lv, Yaxin Yu. Xbase: Making Your Gigabyte Disk Files Queriable. In SIGMOD 2002

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. SIGMOD Record, 26(3): 54-66, September 1997.

[BF00] M.A. Bender, M. Farach-Colton: The LCA problem revisited. In: Proc. Latin American Theoretical Informatics (LATIN). Volume 1776 of LNCS., SpringerVerlag (2000) 88--94.

[MS99a] T. Milo, D. Suciu: Index Structures for Path Expressions. In ICDT'99, the 7th International Conference, Jerusale, Israel, pages 277-295, 1999.

[OR01] T. B. Ohme and E. Rahm. Xmach-1: A Benchmark for XML Data Management. In Proceedings of BTW2001, Oldenburg, 2001. 264-273.

[RPK05] Yang, R., Kalnis, P., Tung, K.: Similarity Evaluation on Tree-structured Data. In SIGMOD (2005) 754—765

[QWG+96] D. Quass, J. Widom. R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov,

A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, and J. Wiener. LORE: A Lightweight Object Repository for Semistructured Data. In Proc. Of the ACM SIGMOD International Conference on Management of Data, Montreal, Canada, June 1996.

[SKW01] A. Schmidt, M. Kersten, M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. ICDE Conference, 2001

[SSG02] Agrawal S,Chaudhuri S,Das G.DBXplorer:A system for keyword -based search over relational databases.In:Agrawal R,et al.,eds.Proc.Of the 18th Int'l Conf.on Data Engineering.San Jose:IEEE Press,2002.5-16.

[STZ+99] J. Shanmugasundaram, K. Tufte, C. Zhang, and et. Al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In Proceedings of the International Conference on Very Large Data Bases Edinburgh, Scotland, 1999.

[TTK+04] Dalamagas, T., Cheng, T., Winkel, K., Sellis, T.: Clustering XML Documents using Structural Summaries. In EDBT Workshops (2004) 547—556

[TW02] Anja Theobald , Gerhard Weikum: The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking, Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology, p.477-495, March 25-27, 2002

[URL] XMLRepository. http://www.cs.washington.edu/research/xmldatasets/www/repository.html

[URL2] XML database products: http://www.rpbourret.com/xml/XMLDatabaseProds.htm#categories

[URL3] Tamino: the Information Server for Electric Business. http://www.software-ag.com/tamino.

[URL4] Jaxen XPath engine: http://jaxen.org

[URL5] Apache Xindice. http://xml.apache.org/xindice/

[URL6] Apache Xerces. http://xml.apache.org/xerces-c/

[URL7] Berkeley DB XML. http://sleepycat2.inetu.net/products/bdbxml.html

[Woo98] Lauren Wood, et al. Document Object Model (DOM) level 1 Specification. Version 1.0 W3C Recommendation 1 October, 1998. http://www.w3.org/TR/1998/REC-DOM-level-1-19981001/DOM.ps.

[WBD+00] K. Williams, M. Brundage, P. Dengler, etc: Professional XML Databases. Wrox Press; 1st edition (January 15, 2000)

[WWL+02] W. Wang, H. Wang, H. Lu, et al. Efficient Processing of XML Path Queries using the Disk-based F&B Index. In VLDB, 2005

[XMill] www.garshol.priv.no/xmltools/prod/XMill.html

[XP05] Y. Xu, Y. Papakonstantinou: Efficient Keyword Search for Smallest LCAs in XML Databases. SIGMOD Conference 2005

[ZLD+01] Chun Zhang, Qiong Luo, David DeWitt, Jeffrey Naughton, Feng Tian. On the Use of a Relational Database Management System for XML Information Retrieval http://www.cs.wisc.edu/niagara/papers/ czhang00.pdf.

附 录

参与的科研项目

国家自然科学基金项目“基于 XML 的 WEB 数据发布、交换和集成”
实验室与 Sybase 公司合作项目“XML 数据存储系统设计实现”

已发表或录用的论文

Tao Xie, Chaofeng Sha, Xiaoling Wang, Aoying Zhou: Approximate Top-k Structural Similarity Search over XML Documents. *APWeb'2006*. Published by ***Lecture Notes in Computer Science*** 319-330.
谢涛，王晓玲，欧阳树生，周傲英. XML 关键词检索的最小公共祖先快速查找方法，《计算机研究与发展》2006 年第 43 卷增刊：477-483.

致 谢

回顾三年的学习生涯中，我有幸得到了众多师长和同学的指导、支持和帮助，在此谨对他们表示衷心的感谢。

首先，感谢我的导师周傲英教授。我由衷的尊敬他，不仅因为他治学严谨、待人诚恳宽厚，而且周教授他还千方百计为我们创造各种有利的研究条件和氛围。感谢他在研究方法和思路上给予的悉心指导；在生活中给予的无微不至的关怀和支持。在今后的人生道路上，我将铭记他的教诲。

其次，感谢本研究小组的王晓玲、钱卫宁、宫学庆和沙朝峰等老师对我的鼓励和指导。本文中的很多工作都是和他们的指导分不开的。

本文的顺利完成，得益于本实验室 XML 研究小组的集体智慧。感谢研究小组的所有成员。正是与他们的讨论与合作才使我得以完成此论文。

实验室的同学们伴随我经过了愉快、有益的三年，他们让我觉得实验室有着家庭般的温暖！在此感谢：郭志懋，黄宇凯，王伟彦等。

感谢所有帮助过我的同学和朋友！

最后，着重感谢我的父母和姐姐，在多年的求学生涯中，他们一直关心、鼓励着我，使我可以集中精力学习和研究。祝愿他们快乐健康！