江苏大学

硕士学位论文

海量存储系统中元数据管理机制的研究

姓名：吴婷

申请学位级别：硕士

专业：计算机应用技术

指导教师：鞠时光

20100601

摘　要

海量存储系统中需要保存 Terabyte、Petabyte 级别甚至更大规模的数据。数据的元数据如文件的名字、属性、保存地址和访问授权等信息一般由元数据服务器进行管理。在访问海量存储系统的数据前，需要首先查找和获得元数据。因此元数据管理机制将直接关系到海量存储系统的 I/O 性能。

现有的海量存储系统一般采用目录层次结构和哈希算法管理元数据，存在修改元数据和查询目录等操作所需时间和空间开销大等问题，也没有针对海量存储系统中元数据访问特性的优化机制，严重制约了海量存储系统的 I/O 性能。

本文在分析海量存储系统中元数据管理特性的基础上，引入 DBMS 技术以及数据分级的方法，提高管理元数据的效率。论文的具体工作包括：

首先引入二维表保存系统中的元数据信息，提出了基于 DBMS 的新型元数据管理策略，给出了各类元数据操作的流程；分析了在海量存储系统中用于管理元数据信息时所需的时间和空间开销以及适应不同运行环境的能力；实现了基于 DBMS 元数据管理策略的原型系统，采集实际文件系统中的元数据，构建多种测试环境进行测试与分析，结果表明基于 DBMS 的元数据管理策略能有效地减少管理元数据所需的时间和空间开销，提高管理元数据的灵活性，增强适应能力。

在分析海量存储系统中元数据时间特性的基础上，依据元数据的生命周期，设计了元数据分级算法，将元数据分为活跃元数据和非活跃元数据；设计了分区索引算法，提高查询活跃元数据的性能；改进了基于哈希函数的索引方法，设计了非活跃元数据的索引算法，减少了管理非活跃元数据所需的时间与空间开销；从查找元数据与更新索引所需的时间与空间开销两方面进行了分析，验证了其能有效地减少了查询元数据和更新索引所需的时间和空间开销；实现了元数据分级索引算法的原型系统，采集实际文件系统中的元数据，构建多种测试环境进行测试与分析，结果表明元数据分级索引算法能有效地提高查询元数据的性能。

关键词：海量存储系统，元数据管理，数据库管理技术，数据生命周期，索引算法

ABSTRACT

The mass storage systems need to be saved Terabyte and Petabyte level even more massive data. The metadata, such as file names, attributes, saved address and access authorization information is generally managed by the metadata servers. Before access to the data of mass storage systems, you need to find and access the metadata. Therefore, the metadata management mechanism is directly related to the mass storage systems' I/O performance.

The mass storage systems generally exploit the hierarchy architecture or hashing scheme to manage the metadata, which requires more time and memory consumption for some operations, such as modifying metadata and querying directory, and it doesn't optimize the data access feature ,thus, it seriously affects the I/O performance of the system.

This paper introduces DBMS technology and data classification methods based on the analysis of the mass storage system metadata management features to improve the efficiency of metadata management. The concrete works of this dissertation can be summarized as follow:

First, we introduce the two-dimension table to preserve the metadata information and propose a new metadata management strategy based on DBMS and then give the various operating processes of metadata. We analyze the time and space consumption and adaptability to different operating environment for this strategy and realize this metadata management strategy prototype system. We collect actual file system metadata and build a variety of test environment for testing and analysis. The results prove that: using two-dimensional table preserve metadata can effectively reduce the time and space consumption, improve the flexibility of metadata management and enhance adaptability.

In the analysis of mass storage system access request based on the characteristics of time, we introduce the data life cycle technology and design the metadata classification algorithm and divide metadata into active and non-active

metadata. Then, we design the partition index algorithm for active metadata to improve query performance of active metadata and design the improved hash functions index method for non-active metadata to reduce non-active metadata management time and space consumption. Through analysis, we validated that the algorithm can reduce the time and memory consumption for query and index update. We collect actual file system metadata and build a variety of test environment for testing and analysis to realize this metadata index algorithm prototype system. The results show that: the classification index algorithm can effectively improve the performance of metadata query.

KEY WORDS: mass storage systems, metadata management, database management technology, data lifecycle, index algorithm

学位论文版权使用授权书

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

保　密　□，在　　年解密后适用本授权书。

本学位论文属于

不保密　☑。

学位论文作者签名：吴婷　　指导教师签名：鞠时光

2010 年 6 月 17 日　　2010 年 6 月 17 日

独创性声明

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

学位论文作者签名：吴婷

日期：2010 年 6 月 17 日

第一章 绪论

1.1 研究背景及意义

随着存储系统的建设进程不断加快，信息资源正朝着数字化、智能化、网络化和多媒体化的方向发展。在网络环境下，信息量的剧增、存储需求的高速增长令人难以预测。信息技术使得人们能够足不出户的交流、查询和获得信息，但是需要不断增长的大量数据作为支撑。因此，存储系统的规模在不断增大，最近几年对存储的需求量都以100%的速度在增长。在这样的海量存储系统内，往往存在成百万、上千万个文件。在访问海量存储系统的数据前，需要查找和获得数据的元数据之后，才能读取相应的数据。

从存储服务的发展趋势来看，其应用出现如下特点[1]：

第一，数据规模大，且呈现出持续的海量式的增长速度，而且这些数据都需要保存起来，以备访问修改。

第二，对数据的有效管理提出了更高的要求，对查询操作的事务处理能力要求高，响应时间要求苛刻。

根据上述存储系统的发展需求，需要采用高效的手段对存储系统的元数据服务器(Metadata Server，MDS)进行统一、简洁的管理和维护，而管理的对象从本质上来说就是系统的元数据。在大规模的海量存储系统中[2]，元数据的访问和处理是一个潜在瓶颈，因此避免瓶颈，对元数据进行高效的管理是系统取得高性能的关键。尽管元数据的数据规模比该存储系统的整个数据规模要小得多，但是元数据通常都是那些可以有效支持系统管理的数据，因此该系统所有的访问基本上都与元数据息息相关。

目前元数据管理通常使用目录子树法和哈希法两种方法。目录子树法是根据子目录确定所在的MDS，目录子树保持了文件系统的层次结构，由于不同目录中的文件数量和被访问的热度存在着很大的差异，在执行改名、更改权限和目录内容列表等操作时，需要迁移或修改大量的元数据，时间和空间开销很大。哈希法是一种基于哈希函数的构造方法，它利用文件标识符映射到相应的存储单元来实现负载均衡。其中元数据的直接定位是根据文件全路径名将元数据平均分布到

不同的MDS中，文件全路径名的更改会导致大量元数据的迁移；元数据的间接定位则是通过文件全路径名的哈希值与 MDS 建立一张元数据查找表(Metadata Look-up Table，MLT)，客户端通过MLT确定MDS，更名时只需更新MLT。但是在PB级存储系统中，当更名或者MLT结构发生改变时，MLT未能及时更新，导致哈希值所映射的MDS与实际不相符，需要逐个遍历所有MDS中元数据进行查找定位，引起性能的急剧下降；同时由于MLT的查询复杂度为O(log n)，n为系统中文件的个数，当系统有并行访问大量不同文件的访问请求任务时，同样极其影响元数据的访问效率。

文件系统中大约有50%到80%的操作是对元数据进行的[2]，因此实现元数据服务的高性能、高可靠性、负载均衡性以及可扩展性至关重要，如何管理元数据已经成为一个重要的研究热点。

为了提高元数据的管理性能，本文引入二维表保存元数据信息，提出一种基于DBMS的新型元数据管理策略，能有效地减少查询和更新所需的时空开销，实现高效、灵活的元数据管理功能。由于使用通用B树建立索引，没有针对海量存储系统中元数据访问特性的优化机制，所以我们提出将元数据根据其生命周期进行分级，并分别建立索引的新型元数据索引算法来提高元数据的查询效率。下面我们介绍一下元数据的管理以及相关研究。

1.2 元数据管理概述

对于大型的海量存储系统，避免瓶颈对获得高性能和高可扩展性是至关重要的。在海量存储系统中，大约有 50%到 80%的操作是对元数据进行的，元数据的访问就是一个潜在的瓶颈，而且对元数据的管理也是海量存储系统中一个重要而复杂的部分，所以采取怎样的方式来有效地管理好元数据是获得系统高性能和高可扩展性的前提。我们首先回顾一下元数据管理的发展历程，然后介绍国内外元数据管理的研究现状。

1.2.1 元数据

元数据最本质、最抽象的定义为关于数据的数据(data about data)[3]，它是一种广泛存在的现象，在许多领域有其具体的定义和应用。

在软件构造领域，元数据被定义为：在程序中不是被加工的对象，而是通过其值的改变来改变程序的行为的数据。它在运行过程中起着以解释方式控制程序行为的作用。在程序的不同位置配置不同值的元数据，就可以得到与原来等价的程序行为。

在图书馆与信息界，元数据被定义为：提供关于信息资源或数据的一种结构化的数据，是对信息资源的结构化的描述。其作用为：描述信息资源或数据本身的特征和属性，规定数字化信息的组织，具有定位、发现、证明、评估、选择等功能。

而我们所研究的存储系统领域，元数据则被定义为描述数据及其环境的数据。其作用为：提供基于用户的信息，例如记录数据项的业务描述信息的元数据能帮助用户使用数据；支持系统对数据的管理和维护，如关于数据项存储方法的元数据能支持系统以最有效的方式访问数据。如表 1.1 所示。

表 1.1 元数据示例

数据	元数据
文档	项目名称，创建者，创建日期，…
E-mail	发送者，主题，发送日期，…
音乐	专辑，艺术家，格式，…

如上表所示的文档，它的创建标志着一个项目的结束，那么在这里文档就是数据，而元数据包括项目名称，文档创建者以及创建日期。如果把该文档归档到文件柜中，那么元数据用来决定它归档的位置。例如，项目文档按照时间归档。文档在计算机系统的归档时，元数据的作用是类似的。当文档以电子的形式保存在磁盘或磁带时，元数据被记录在文件系统或数据库中并与该文档相关联，所以在需要时可以很容易的被找到。

1.2.2 海量存储系统的发展

从二十世纪五十年代国际商用机器公司(International Business Machines，IBM)发明的第一块硬盘以来，硬盘在存储容量、存储密度和性能等方面不断地快速发展，存储设备的接口也稳健而缓慢的演化。正是由于这些接口的稳定和标

准化，从而促进存储设备和存储系统的发展[4]。

随着大规模集成电路的发展，计算机系统中 CPU 速度以每年 40%到 100%提高，而磁盘寻道时间每年仅以 7%左右提高[5]，因此 CPU/主存的速率与 I/O 子系统的速率差距越来越大，使得I/O子系统对整个系统性能的影响显得更为突出。为了解决这个问题，提出并行存储的研究[6]，典型实例就是磁盘阵列技术[7]，它通过对多个磁盘的并行 I/O 访问，利用时间和空间上的重叠，实现高的数据传输率、I/O 吞吐率和扩大存储容量。

随着应用的拓展，人们对存储系统的可靠性研究提出了更高的要求，在研究提高存储系统容量和速度的同时，美国加州大学 Berkeley 分校的 David A.Patterson 等人在 1988 年首先在磁盘阵列中使用了冗余容错技术，提出了廉价磁盘冗余阵列（Redundant Arrays of Inexpensive Disks，RAID），后又称为独立磁盘冗余阵列（Redundant Arrays of Independent Disks，RAID）[8-9]。由于 RAID 采用了数据分块的技术，即在多个磁盘上交叉存放数据，使得多个磁盘进行并行的操作，I/O 响应时间得到改善；同时采用冗余容错技术，极大地提高了磁盘阵列的可靠性和可用性，并在阵列 Cache、多 SCSI（Small Computer System Interface）串设备的并行实现等方面也有较为深入的研究，使 RAID 成为了一种重要的海量存储系统结构。

在二十世纪八十年代后期，随着网络技术的兴起和普及，各种海量数据的应用，如高性能计算、数字博物馆、流媒体、气象服务、地理信息系统等，促使存储需求呈指数增长。因此，计算机系统的设计重点从传统以计算处理为中心转移到以数据应用为中心，并将网络服务和存储服务相结合，构成了网络存储系统。正是这些海量数据的应用需求推动了海量存储系统不断的发展和性能不断的改善，推出新的存储体系结构，从传统的直接存储系统（Direct Access Storage，DAS）、附网存储（Network Attached Storage，NAS），发展到新一代的存储区域网（Storage Area Network，SAN）。

存储区域网是一种高速专用网络，用在 MDS 和不同种类的数据存储设备之间提供连接[10]。在主机与各种智能磁盘之间用一个专用交换机进行连接。图 1.1 是典型的海量存储区域网系统结构。

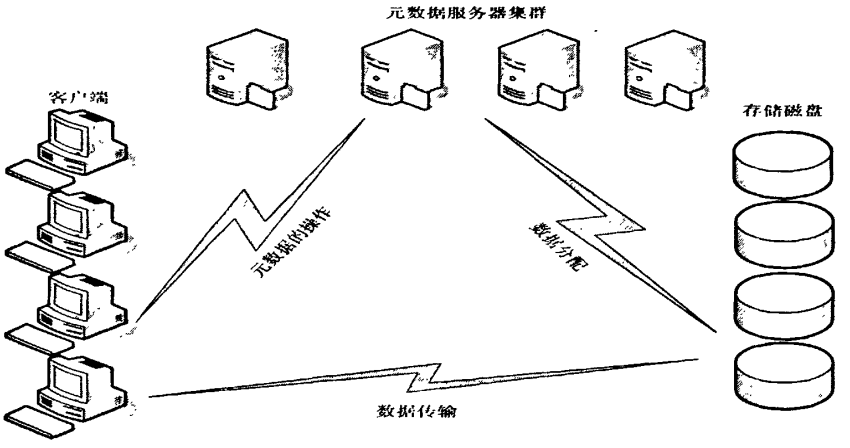

图 1.1 典型的海量存储区域网系统结构

主机用于接受用户的访问请求，提交给存储系统；元数据服务器负责管理存储磁盘中所有数据的元数据信息，为每个访问请求提供被访问数据所在存储设备的信息，亦称为存储“数据的数据”；智能磁盘负责保存数据，响应主机的每个访问请求。

海量存储系统通常用元数据服务器集中管理数据的元数据信息，如文件的名字、属性和访问授权等。访问海量存储系统中的数据时[11]，首先需要访问元数据服务器，利用文件名等信息进行查询，在获得数据的属性和访问授权等信息后，才能读取相应的数据。海量存储系统需要处理用户大量的访问请求，在元数据服务器集群中，要想准确、快速、高效地管理元数据，就需要一种良好的元数据管理机制，所以元数据管理性能的优劣对海量存储系统的I/O性能有着很大的影响。

1.2.3 元数据管理的发展历程

信息社会中我们每天都面对大量的数据和信息，而且越来越多的信息被数据化，尤其是伴随着 Internet 的发展，由此产生的各种数据呈几何级数爆炸式的增长，使得存储技术产品受到业界及专家们的高度重视。访问存储系统中的数据时，需要获得数据属性和访问授权等元数据信息后才可以访问相应的数据，所以元数据管理性能的好坏对海量存储区域网系统的性能影响很大，元数据管理的发展大致分为四个阶段[12-13]。

第一阶段：静态型元数据管理阶段

在数据库管理系统出现以前，为代码重用而开发的软件管理器就可以用来管理元数据。不过它既用来处理数据也用来处理元数据。软件管理器开发至今仍然很成功并且还在广泛使用，例如标准 C 库、C++类库和 MFC 库。尽管这种把元数据与业务数据联合起来管理的方式并不是很复杂，但这些库自始至终都是静态的，所以一旦一个软件库被开发出来并且发布使用，要更改它的结构和接口会相当困难。

第二阶段：动态协作型元数据管理阶段

在这个元数据管理的阶段，数据库管理系统的发展较为成熟，于是提出了对元数据进行更加细致管理的新的研究目标。为了达到这个目标，以前静态的软件库已经不能满足需求，用户要求能非常方便地检索、增加、删除及更新软件库的内容，要实现软件库的动态化，就需要使用软件库自身的元数据。这种技术需要把代码的元数据的收集与实际的库的应用结合在一起。主要的方式是把元数据存储在数据库中，当软件库升级的时候就去修改元数据库中相应的地方。目前已经有很多工具使用了这种技术，如 IBM 公司开发的信息管理工具 ReDiscovery。

第三阶段：纽带型元数据管理阶段

这个阶段标志着元数据及其管理的成熟，元数据成为联接多种业务资源的纽带。这个阶段的第一个尝试就是引入数据字典，它集中提供了一个数据仓储中所存储的数据的一些信息，例如格式、意义、关系、来源以及域等等。80 年代末，IBM 公司开始着手建立综合型数据仓储，并且提供统一的数据接口，使得需要这些数据的工具能很方便地从中获取。为了理解数据仓储的组成，我们来研究一下数据仓储的重要特征。数据仓储的一个重要特征就是它所管理的元数据的类型。包括数据库元数据、数据模型元数据、数据移动元数据、业务规则元数据、应用组件元数据、数据访问元数据和数据仓库相关元数据。其他的特性包括描述仓库核心元数据类型的信息模型、形式说明语言、支持不同产品互操作的语言以及标准查询语言等等。Microsoft Repository Service 就是众多数据仓储产品中的一个。

第四阶段：元数据集成管理阶段

这个阶段的任务主要是管理多种元数据的集成。集成一般分为两种形式，即工具集成和数据集成。不同的工具都会产生各自特定的元数据，元数据的集成是

为了建立工具之间的相互联系。类似的，来自不同部门的数据格式也有可能不相同。为了帮助决策支持，这些数据也需要集成。只要元数据遵从某种通用的数据模型，在第三阶段描述的数据仓储技术就可以支持元数据集成。然而不同的工具来自不同的公司，有不同的规范和不同的侧重点，因此不具备通用模型。于是Metadata Coalition(MDC)和 Object Management Group(OMG)两大组织在 90 年代开始着手制定元数据标准。其中 MDC 在 90 年代末制定出了 Open Information Model(OIM)标准，而另一方面，OMG 在 2000 年代初制定出了 Common Data Warehouse Metamodel(CWM)标准。不过现在并没有强健的工具能完全支持 CWM 模型。例如 Microsoft 公司的 Metadata Services 和 Teradata 公司的 MDS，它们并不能完全进行决策支持问题的解决。

1.2.4 元数据管理的研究现状

当前国内外对元数据管理的研究处于快速发展阶段，海量存储系统一般采用目录子树分区法和文件哈希法来管理系统中的元数据[14]，随着存储系统规模的不断扩大，改进的元数据管理策略成为新的研究热点。国内外一些大学和研究机构主要在提高访问效率、负载均衡和提高可扩展性这三个方面进行改进，下面我们分别进行介绍。

1986 年美国麻省理工学院的 Popek 等人[15-17]提出的目录子树分区法是以子目录为单位，把根目录下面的各个子目录或者下层的某些子目录分配到不同的 MDS 上进行存储。该策略可以维持一个完整的传统文件系统层次目录结构。用户对不同子目录的访问，将被引向不同的 MDS，从而达到了分流的目的。这种结构便于执行与层次目录结构相关性较大的操作，如“ls”。但是，当某一子目录的内容变成热点数据时，会有很多用户同时访问该子目录，这就会使得存放该子目录的服务器负载加重，成为系统的性能瓶颈。Coda、NFS、Sprite 都属于这一类文件系统。

1996 年 IBM 的 P.F.Corbett 等人[18]提出的纯哈希(Hash)法是通过对文件的某一特定键值(如路径名、文件名等)进行哈希，根据哈希的结果把文件分布到各个服务器上面去。Hash 法可以把同一目录下的文件均匀分布到各个服务器上，具有更好的负载均衡性，也可以避免目录热点问题。但 Hash 法也有不足。首先，它不

能提供标准的目录层次结构，处理类似“ls”之类的指令会比较复杂。其次，Hash法的可扩展性比较差。Hash法的关键是其Hash函数。一旦Hash函数确定，其输出范围也就确定了。此时如果系统进行扩展，Hash函数就要增大输出范围，这样就必须修改Hash函数。新的Hash函数建立后，又会导致很多元数据的分布和修改之前不一致，于是又要在不同的MDS之间进行大量的数据迁移。这些繁琐的操作使得在扩展以Hash法进行元数据分布的存储系统时显得非常困难。zFS、Vesta、Lustre都属于这一类文件系统。

LH法是2003年美国加利福尼亚圣克鲁兹大学存储系统研究中心Scott A.Brandt等人[2]提出的。它通过结合静态子树划分法和纯哈希法的优点并消除了他们的瓶颈，采用路径名哈希来存放文件的元数据，采用延迟懒惰的元数据更新策略，对某些耗时较多的元数据更新操作进行延迟更新，提高了系统的整体运行速度。

随着对大规模存储系统的需求逐渐递增，海量存储系统的元数据管理策略正逐渐成为研究的热点，而经过改进的目录子树分区法和纯哈希法则成为新的研究趋势。国内外一些大学和研究机构在这方面已开展了很多研究工作，主要在提高访问效率、负载均衡和提高可扩展性三方面进行了研究，提出了改进的方法。下面我们分别来介绍一下。

(1)提高访问效率

2008年韩国的Jong-Hyeon Yun等人[19]提出字典分配法把文件目录的层次结构转化成线性结构，按顺序分配到每个MDS里。同时每个MDS都保存一份位置分配记录。当客户访问MDS时，如果没有这份位置分配记录，会先向MDS索取一份位置分配记录。这样客户就可以直接定位到保存自己所需要的元数据的MDS，提高了访问效率。

2004年美国缅因大学的Zhu Yifeng等人[20]提出了将布鲁姆过滤器（Bloom Filter，BF)用在了元数据的查找上，设计了HBA(Hierarchical Bloom filter Arrays)。该设计的思想是每个MDS包括两级bloom filter array。第一级bloom filter叫LRU BF，呈现出该MDS上LRU列表中被访问次数最高的文件；第二级bloom filter呈现出该MDS上包含的所有文件元数据的分布。该MDS也保存了所有其他MDS上的两级bloom filter的副本。通过这种两级结构的布鲁姆过

滤器结构能进行高效的元数据的查找。

针对布鲁姆过滤器会出现假阳性误判的现象，即将某些集里不存在的元素误判为存在，2006 年华中科技大学的冯丹等人[21]提出了改进型的两级布鲁姆过滤器的结构：第一级：存储对象的一些属性，判断客户访问请求的某些属性是否存在； 第二级：用来验证对应于某个对象的这些属性的内在关联性，判断这些属性是否对应于这个对象，提高了访问的准确率。

(2) 负载均衡

2008 年国防科技大学的肖农等人[22]提出了动态相对负载均衡策略来解决文件元数据的分配问题。该策略首先采用元数据分级管理的方法，将元数据分为目录元数据和文件元数据来分别进行管理。而对于文件元数据的管理采用动态相对负载均衡策略，通过当前负载水平，MDS 性能，内存容量，网络吞吐量，移动元数据的总量等一些指标计算服务器的性能来重新分配文件元数据以保持负载的相对平衡。

2008 年新加坡的 Bharadwaj Veeravalli 等人[23]提出了基于各 MDS 的负载大小进行元数据分配的负载均衡策略。该策略根据各 MDS 负载值的大小构建 B+ 树来进行元数据的分配和查找。针对 MDS 可能出现“热点”的情况，提出了基于窗口的自适应元数据复制策略，可以根据 MDS 集群的负载状况动态地创建和删除元数据的复制。

(3) 提高可扩展性

2004 年新加坡数据存储中心的 Jie Yan 等人[24]提出了一种哈希分区(HAP)的元数据分布策略。该策略将元数据的存储和管理分成两层，所有的 MDS 节点共用一个独立的大型的公共存储空间来存取元数据。该空间分为多个逻辑分区，每个 MDS 可以管理一个或多个分区。该策略通过哈希法和安装/卸载分区的方法，极大地减少了 MDS 机群内部的元数据迁移。另外，还通过动态的负载调节机制，保证系统的负载均衡。

2007 年华中科技大学计算机学院外存储国家重点实验室的吴伟等人[25]提出了一种目录哈希(Directory Hash)的元数据分布法，简称 DH 算法。该算法与其它哈希法有很大的区别。其它的哈希法都是对文件进行哈希，以文件粒度来分布元数据。而 DH 算法则是对目录进行哈希，以目录粒度集中分布元数据，引入了目

录存储单元(DSU)和目录存储单元索引管理服务器(DIMS)的概念，以目录为单位集中分布元数据。DH 算法在不降低系统性能的情况下，有效地解决了传统Hash 法中存在的 MDS 的增加与删除导致大量元数据迁移等许多难题，极大地提高了存储系统的可扩展性。

1.3 研究目标

现有海量存储系统一般采用目录子树分区法和文件哈希法管理系统中的元数据，由于采用层次结构保存元数据，访问文件时需要遍历目录路径上的所有文件，在执行改名、更改权限和目录内容列表等操作时，需要迁移或修改大量的元数据，时间和空间开销很大；而利用哈希方法保存元数据，同样在执行改名、更改权限和目录内容列表等操作时，需要迁移或修改大量的元数据，给海量存储系统的性能带来很大影响。因此，本文的研究目标就是实现高效的海量存储系统元数据的管理，其中我们着重需要解决以下两个关键问题：

(1) 研究新型的元数据管理策略，提高元数据管理的效率

元数据管理策略对海量存储区域网系统的性能影响很大，现有的元数据管理策略通常使用无结构或半结构化的形式来保存系统中的元数据信息，缺乏数据组织的有效性，普遍存在遍历开销大，在执行目录改名和更改访问权限等操作时引起大量元数据迁移的问题，严重影响了海量存储系统的 I/O 性能，因此为了实现高效、灵活的元数据管理功能，针对现有的传统元数据管理策略存在的不足之处，用合理有效的组织形式来保存元数据，研究新型元数据管理策略具有重要的意义。

(2) 元数据索引算法的研究

很多情况下，人们并非对这海量数据本身感兴趣，而是需要从这些海量数据中提取出对自己有用的信息，这就涉及数据的查询技术。欲实现高效的查询必须建立合理的索引机制，能否在海量存储系统中实现数据的精确、方便、快速查询，采用何种索引方法以及索引性能的优劣直接影响到整个系统的性能。而采用现有的树型索引算法查找元数据时需要遍历该文件访问路径上的所有目录，响应时间长；采用哈希法，则消除了同一目录下的访问元数据的存储局部性，数据更新所引起的哈希函数的改变，导致大量元数据的迁移，给海量存储系统的性能带来很

大影响。所以本文提出将元数据根据其生命周期进行分级，分级后根据不同级别元数据各自的使用特性分别建立索引的新型元数据索引算法，有效地减少元数据查询和更新的时空开销，实现高效的元数据查询功能，从而有效地提高海量存储系统的性能。

1.4 本文主要工作及组织结构

随着信息存储容量不断地成爆炸性的增长，为了保证海量存储系统元数据管理的效率，必须研究新型的元数据管理机制，其中如何有效地组织保存元数据和进行高效的元数据查询是研究的主要内容。本文主要工作包括以下几个方面：

(1)针对现有的元数据管理策略在执行改名、更改权限和目录内容列表等操作时存在时间和空间开销大等一些问题，本文引入 DBMS 的相关技术，提出基于二维表的元数据结构。

(2)利用上述的多个二维表的结构保存文件的元数据信息后，在分析元数据管理需求的基础上，设计一系列元数据的操作算法来完成用户访问请求中的元数据操作要求。实现基于 DBMS 元数据管理策略的原型系统，采集实际文件系统中的元数据，构建多种测试环境进行测试与分析。

(3)在我们设计的基于DBMS的新型元数据管理策略中使用通用B树建立索引，没有针对海量存储系统中元数据的访问特性进行优化。为此，我们依据海量存储系统中元数据的时间特性，提出依据元数据的生命周期，对海量存储系统中的元数据进行分级，根据各级元数据不同的使用特性使用不同的方法建立索引，实现高效的元数据查询功能。实现元数据分级索引算法的原型系统，采集实际文件系统中的元数据，构建多种测试环境进行测试与分析。

本论文共分为五章，其组织结构具体安排如下：

第一章：介绍研究背景及意义，对元数据管理技术的发展进行概述。针对目前所存在的问题，明确研究目标，最后给出本文的主要工作及组织结构。

第二章：首先分析海量存储系统中元数据管理特点，接着设计基于 DBMS 的海量存储系统元数据管理结构。

第三章：在分析现有元数据管理策略存在的不足的基础上，设计基于DBMS的元数据管理策略，给出各类元数据操作的流程。并分析使用该策略管理元数据

所需的时间和空间开销，实现策略的原型系统，进行测试与比较。

第四章：设计基于数据生命周期的元数据分级算法，依据元数据的生命周期对元数据进行分级，根据各级元数据不同的使用特性使用不同的方法建立索引。并分析使用该索引算法查询元数据及更新索引所需的时间和空间开销，实现算法的原型系统，测试和分析查询元数据所需比较的平均次数和最大次数。

第五章：首先对已做的工作做一个总结，接着对将来的进一步工作进行展望。

第二章 海量存储系统元数据管理的特点及分析

我们首先对海量存储系统元数据管理进行特性分析，在对目前海量存储系统元数据管理特性分析的基础上，设计基于 DBMS 的海量存储系统元数据管理结构。

2.1 海量存储系统元数据管理特性分析

在存储系统中，避免瓶颈对获得系统的高性能和高可伸缩性的影响是非常巨大的。虽然元数据的大小在存储容量上仅占很小比例，有 50%到 80%的存储系统的访问是访问元数据，但是在大规模 PB 级海量存储系统中，元数据可达 GB 级，甚至到 TB 级，这些元数据是用来描述一个存储系统特征的数据，元数据的访问就是系统的一个潜在的瓶颈，而且元数据的管理也是海量存储系统中一个重要而复杂的部分，所以采取何种方式来有效管理元数据是获得系统高性能和高可伸缩性的前提。

归结起来，海量存储系统元数据管理的重要性主要表现在两个方面：

第一，元数据是最重要的系统数据。客户读写海量存储系统中的文件，首先要对数据进行定位，只有先获得文件的元数据后，才能将客户的请求转发到正确的 I/O 服务器进行数据访问。如果不能进行正确的定位，基于文件数据的并行应用程序就无法执行。因此，必须保证系统中元数据的正确性和可靠性。

第二，元数据的访问性能影响着整个海量存储系统的性能。在海量存储系统中，元数据的访问很频繁，而元数据文件通常又很小，这样对大量小文件的访问，会对系统性能造成瓶颈。

海量存储系统提供的元数据管理主要包括海量存储系统元数据存储的元数据存储管理和海量存储系统元数据访问请求的元数据访问请求管理这两种类型的管理。元数据存储管理是元数据访问请求管理的基础，元数据存储管理的好坏影响整个海量存储系统元数据管理的质量，是整个海量存储系统元数据管理的核心问题。下面我们分别介绍一下这两种管理：

(1) 元数据存储管理

存储资源合理的组织结构为系统中存储资源的有效管理提供有力的支持，影响整个元数据服务的质量。只有通过对存储资源的合理组织，才能有效地管理和利用系统的存储资源，提高系统的整体性能。随着存储系统规模的日益增长，存储资源管理和使用的参与者规模也变得非常庞大，而数据的有效组织和存储却成了一个非常严重的问题，所以需要通过有效的存储资源管理策略进行管理，避免出现制约系统性能扩展的瓶颈。存储资源的使用模式是有效应用海量存储系统元数据的保证。只有通过有效的存储资源使用模式支持，存储资源用户间才能以较低的代价实现元数据的有效利用，提高元数据利用的效率。

所以，如何有效地组织海量存储系统中的元数据，并加以有效的管理和使用，提供具有较强扩展能力的存储服务是海量存储系统元数据存储管理的关键问题。只有合理有效地解决存储服务的这个关键问题，为海量存储系统元数据服务提供具有较强扩展能力的元数据存储管理，为有效解决元数据请求管理的关键问题提供基础，才能有效地提供海量存储系统元数据管理，提高海量存储系统元数据管理的扩展能力。

(2) 元数据访问请求管理

进入新世纪以来，随着现代科学技术与信息产业的迅猛发展，网络环境下的应用领域不断扩展，越来越多的应用呈现出数据量大，持续倾灌等特点，其庞大的数据规模，往往使得查询性能低下。海量存储系统每天面对着用户大量的访问请求。从海量存储系统整体而言，系统中的元数据表现出部分活跃性。文件系统大部分的文件和数据不会被经常访问，活跃的文件和数据占整个系统的比例较少。活跃文件的比例大约为 4%到 20%。从活跃的数据比例看，活跃数据的比例大约为 10%到 30%。而且单个用户的元数据请求表现出局部性的特征，在一定时间内，单个用户的文件访问往往集中在某些目录下，不会发散到整个海量存储系统。

所以，我们需要针对这些海量数据的访问请求的特性，充分发掘海量存储系统元数据自身的特点，提出更好的查询优化的解决方案，建立合理的索引结构，提高整个海量存储系统元数据访问请求管理的质量，优化系统的整体性能。

2.2 基于 DBMS 的海量存储系统元数据管理结构

现有的海量存储系统一般采用目录层次结构和哈希法管理元数据，存在遍历

开销大，在执行目录改名和更改访问权限等操作时引起大量元数据的迁移以及未针对元数据的访问特性进行优化等问题，本文针对海量存储系统元数据管理的特点和要求，引入 DBMS 的相关技术和元数据分级的方法，用二维表保存系统中的元数据并把元数据分级然后分别使用不同的方法建立索引。引入 DBMS 的海量存储区域网系统结构如图 2.1 所示。

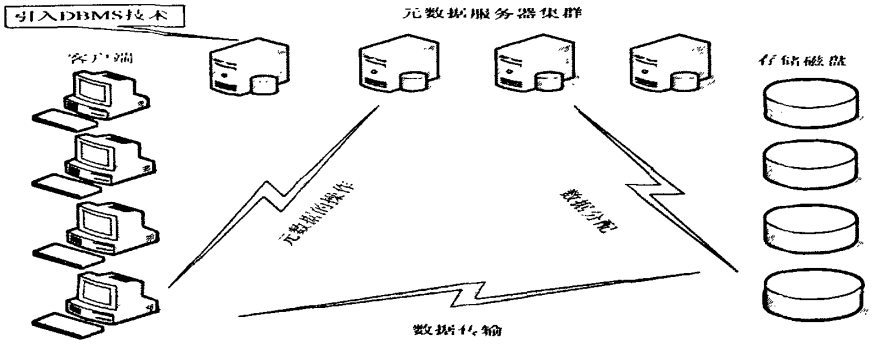

图 2.1 引入 DBMS 的海量存储区域网系统结构

我们在海量存储系统的元数据管理中引入 DBMS 技术，提出使用二维表保存文件和目录的元数据及它们之间的层次结构，并设计对这些二维表的操作算法，用于完成用户访问请求中的元数据操作要求；同时把元数据依据其生命周期进行分级，将元数据分为活跃元数据和非活跃元数据，并根据各级元数据不同的使用特性使用不同的方法建立索引。基于 DBMS 的海量存储系统元数据管理结构如图 2.2 所示。

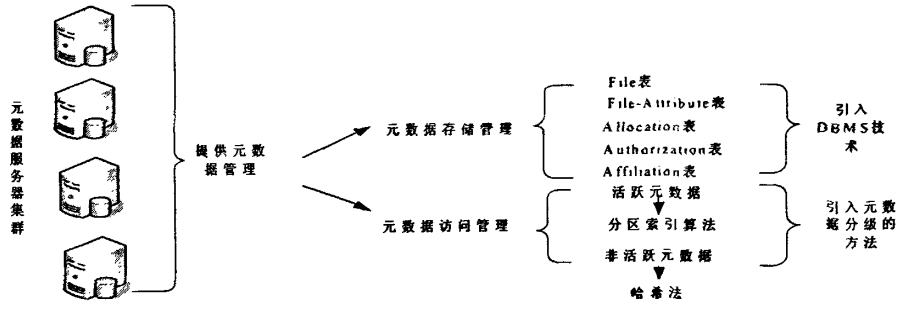

图2.2 基于DBMS的海量存储系统元数据管理结构

在图 2.2 中，我们引入 DBMS 的相关技术，根据元数据存储的需求将海量存储系统中的元数据用二维表来保存，设计 File 表、File-Attribute 表、Allocation

表、Authorization 表和 Affiliation 表来保存文件和目录的元数据及它们之间的层次结构和访问授权等信息，同时设计针对这种结构化的元数据存储结构的元数据操作算法，用于完成用户访问请求中的元数据操作要求，消除目录属性修改对元数据的影响，有效避免元数据的更新与迁移。同时海量存储系统每天必须面对用户的海量访问请求，我们依据元数据的生命周期进行分级，使用不同的方法对不同级别的元数据分别建立索引，减少查询元数据所需的时间和空间开销，提高海量存储系统元数据访问的效率。

2.3 本章小结

本章中我们分析了海量存储系统元数据管理的特性，根据其特点和运行要求分析了海量存储元数据管理的重要性。高效的海量存储系统元数据的管理必须有效地组织元数据并能便捷快速地进行元数据查询。设计了基于 DBMS 的海量存储系统元数据管理结构。

第三章 基于 DBMS 的元数据管理策略

海量存储系统中，元数据管理由元数据的存储管理和元数据的访问请求管理两部分构成。海量存储系统所提供的元数据的存储管理是元数据访问请求管理的基础。如何合理有效地来组织、管理和使用系统中的元数据资源对元数据存储管理性能的影响至关重要。针对现有的元数据管理策略存在的一些缺陷，研究新型管理策略来提高元数据存储管理的质量；同时，针对元数据的存储结构建立合理的索引算法是实现高效的元数据管理的关键问题。在本章节中，我们着重讨论如何针对现有元数据管理策略存在的缺陷来设计新型的元数据管理策略，提高海量存储系统元数据存储管理的质量，实现高效、灵活的元数据管理性能。

海量存储系统中的元数据是指存储系统中用来描述文件属性、层次目录结构、访问授权等信息的数据。这些元数据对于海量存储系统来说至关重要，因为它们直接关系着整个海量存储系统的正确性、可用性和一致性等性能。海量存储系统元数据管理的设计与实现的优劣程度直接影响着海量存储系统整体性能的好坏，是系统评价和优化的一个重要部分。现有的海量存储系统一般采用目录子树分区法和文件哈希法两类元数据管理策略来管理系统中的元数据。下面我们来介绍一下海量存储系统中现有的几种元数据管理策略，针对这些策略存在的不足，引入 DBMS 的相关技术，提出一种新型的基于 DBMS 的元数据管理策略。

3.1 现有元数据管理策略的分析

加州伯克利大学和卡内基·梅隆大学的研究人员分别对运行 NFS v2 的 Auspex 服务器和运行 AFS 的 SPARC 工作站进行了专门的负载研究统计[26-27]，希望能够从收集的负载数据中获得一些文件系统访问负载的共性。表 3.1 综合了这些负载测试的统计结果。

表 3.1 NFS、AFS 运行记录

测试环境 / 操作		NFS		AFS	
		操作请求（%）	所用时间（%）	操作请求（%）	所用时间（%）
元数据操作	读	73.9	47.3	70.9	50.2
	写	1.5	1.7	11.3	13.3
数据操作	读	20.4	31.6	13.9	27.9
	写	4.2	19.3	3.8	8.5

从表 3.1 可以看出，在这两个系统运行记录中，元数据的读和写的操作比例分别占到了文件系统操作请求总数的 75.4%和 82.2%，分别占用 49%和 63.5%的系统资源；而数据操作请求的比例仅占文件系统操作请求总数的 24.6%和 17.7%。可见在常规的应用模式中，元数据读取操作的数量远远超过了数据读写操作的数量，这种频繁发生的元数据服务请求是否能及时得到响应，在很多应用场合下决定了文件系统的整体性能，制约着整个系统的性能。

海量存储系统需要处理用户大量的访问请求，因此元数据的管理性能对海量存储区域网系统性能的影响很大，设计高性能的元数据管理策略是提高存储区域网性能的重要手段之一。常用的元数据管理策略包括：目录子树分区法和文件哈希法。

1986 年美国麻省理工学院的 Popek 等人[15-17]提出的目录子树分区(Directory Sub-tree Partitioning)法，将层次式的目录结构划分为若干子树，把根目录下面的各个子目录或者下层的某些子目录树分布到不同的元数据服务器中进行存储，从而提高元数据管理的性能。该策略可以维持一个完整的传统文件系统层次目录结构。用户对不同子目录的访问，将被引向不同的 MDS，从而能达到分流的目的。这种结构便于执行与层次目录结构相关性较大的操作，如“ls”。但层次式的目录结构存在遍历开销大的问题，在执行目录改名和更改访问授权等操作时需要移动大量的元数据，影响了海量存储系统的性能。

1996 年 IBM 的 P.F.Corbett 等人[18]提出的哈希 (Hash)法，通过设计相应的 Hash 函数，对文件的某一特定键值（如文件名、路径名等）进行哈希，可以将同一目录下的文件均匀地分布到不同的元数据服务器中，能减少元数据操作中的瓶颈，具有更好的负载均衡性，也可避免目录热点问题。但该策略破坏了目录的

层次结构，在执行与目录有关的目录列表和访问授权管理等指令时，需要遍历所有的元数据信息；同样存在执行目录改名和更改访问授权等操作时，需要移动大量的元数据，造成大量元数据的迁移，所需时间与空间开销较大的问题，降低了海量存储系统的性能。

随着对大规模存储系统的需求逐渐递增，海量存储系统中的元数据管理策略正逐渐成为研究热点。国外一些大学和研究机构纷纷对目录子树分区法和哈希法进行改进，在这方面已开展了很多研究工作，比较知名的有懒惰混合(LH)法和元数据共享存储池的管理策略。

2003 年美国加利福尼亚圣克鲁兹大学存储系统研究中心的 Scott A.Brandt 等人[2]提出的 LH 法，在保持元数据的层次目录结构的同时，使用 Hash 函数计算路径名的哈希值来确定存放文件元数据的位置；在文件的权限访问方面，将文件的访问授权与目录的访问授权进行区分管理，提高了更改访问授权等操作的性能。但由于文件的访问授权未分散保存在访问路径中所有目录与文件中，在更改某一个目录的访问权限后，需更新该目录下的所有子目录和文件的访问权限，此外还会引发元数据一致性的问题。

2007 年华中科技大学的苏勇等人[28]提出元数据共享存储池的管理策略。由网络存储器构建共享存储池，元数据采用 Hash 函数进行分布。该策略将元数据的存储和管理分成两层，所有的 MDS 节点共用一个独立的大型的公共存储空间来存取元数据。该空间分为多个逻辑分区，每个 MDS 可以管理一个或多个分区。该策略通过哈希法和安装/卸载分区的方法，可以避免当某个目录成为访问热点时所导致的瓶颈问题。但存在 Hash 函数确定困难，适应性差等问题。

目录子树分区法与 Hash 法是目前海量存储系统中保存元数据的两大类方法，在执行改名、移动文件、更改权限和列表目录内容等操作时，会引发时间与空间开销过大等问题从而影响了海量存储系统的性能。本文在元数据管理中引入 DBMS 技术，提出使用二维表保存文件和目录的元数据及它们之间的层次结构，并设计相应的元数据操作管理算法，消除目录属性修改对元数据的影响，有效避免了元数据的更新与迁移。

3.2 基于二维表的元数据结构

随着信息的爆炸式增长[29]，数据的有效组织和存储却成为一个非常严重的问题，如图 3.1 所示。全球的信息量以每年 30%到 70%的速度在增长，而结构化有序有效被组织起来的数据只有 30%到 50%，没有有效组织起来可以被方便利用的数据信息达到 35%到 55%，甚至超过了“有用”数据的数量。因此，有效的数据组织管理成为当今存储领域一个极具挑战性的问题。

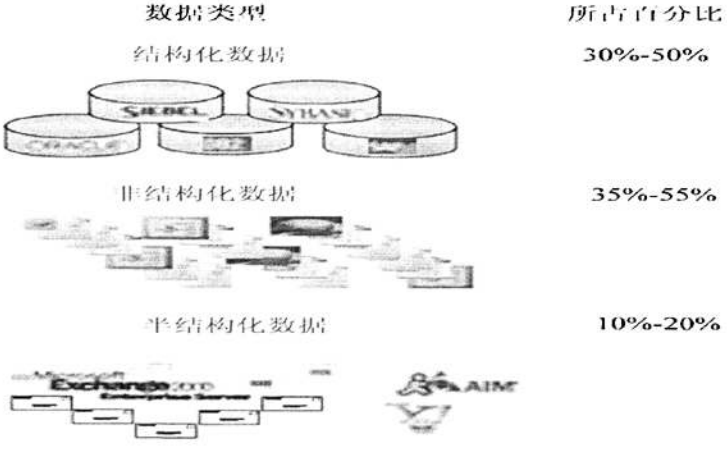

图3.1数据信息的结构化

现有的元数据管理策略通常使用无结构或半结构化的形式来保存系统中的元数据信息，缺乏数据组织的有效性，所以我们引入 DBMS 技术，用二维表这样一种结构化的形式来保存元数据信息，提高海量存储系统中元数据使用的有效性。

海量存储系统中通常将文件的元数据和数据分开存储，数据直接保存在存储设备中。元数据包括文件的属性、访问授权和数据的存储位置等信息，一般由专门的元数据服务器进行管理。主机在接收到用户访问数据的请求后，首先向元数据服务器发送请求，根据元数据服务器返回的数据地址信息，访问相应的存储设备。此时，元数据服务器需要完成元数据的查找，比较访问授权信息和提取数据地址等一系列操作。此外还有不少用户的访问请求只需要对元数据进行处理，如文件的改名、移动、修改属性（文件大小、所有者、创建时间和修改时间等属性）的操作。

为了解决文件重名的问题，仅仅使用文件名查找元数据是不够的，还需要文

件的访问路径等信息。另外，在访问文件的元数据时，需要查找的不仅是该文件的元数据，还有该文件访问路径上所有目录的元数据。因此，我们在设计基于二维表的元数据结构时，将文件名和访问路径结合用于查找文件，有效避免了文件重名的问题。

采用层次结构的元数据组织方式在访问某一文件之前，需要依次、逐层地访问该文件访问路径上的所有目录，很难快速便捷地查找到某个文件。而我们设计多个二维表用于保存元数据，建立索引，可以减少查找文件所需的时间与空间开销。

不同的文件和目录需要保存的属性类型与个数各不相同，传统元数据管理策略中，使用单一格式保存文件和目录的元数据，存在冗余、灵活性差等问题。我们设计动态属性结构，使得能根据各文件和目录的不同需求，灵活地保存不同数量和类型的属性。

目录作为海量存储系统中一类特殊的文件[30]，与一般的文件有所不同，保存了目录所属的文件和子目录信息，还保存了目录和文件之间的层次信息，传统文件系统中采用文本文件保存。我们设计的基于二维表的元数据结构中，目录和文件之间的层次信息由元组之间的关联进行表示。此外，由于目录与文件访问授权管理的需求各不相同，文件的访问授权仅需针对单个文件，而目录的访问授权则针对该目录下的所有文件，修改目录的访问授权时需要逐个修改该目录下所有文件的访问授权信息，需要较大的时间与空间开销。我们设计的基于二维表的元数据结构中，保存了目录和文件的访问授权信息。

根据上述对海量存储系统中元数据管理需求的分析，我们建立五张二维表：File 表、File-Attribute 表、Allocation 表、Authorization 表和 Affiliation 表。

表 3.2 File 表

字段	类型
ID	int
PathName	char
DPID	int

File 表保存文件和目录的基本信息，ID 是文件或目录的唯一标识，PathName 为文件或目录的访问路径和文件名，DPID 为该文件上层目录的标识。

表 3.3 File-Attribute 表

字段	类型
ID	int
AttributeName	char
AttributeType	char
AttributeValue	char

File-Attribute 表保存文件和目录的属性，ID 是文件或目录的唯一标识，AttributeType 为文件属性的类型，AttributeValue 为文件各属性所对应的值。此时文件所有属性均以字符串的形式保存，在使用时再转换为其对应的类型。如 ID 是 1 的文件包含 Size，Type，Owner 和 ModifyTime 属性，此时 File-Attribute 表中包含 (1,Size,int,'100')、(1,Type,char,'normal')、(1,Owner,char,'root') 和 (1, ModifyTime,time,'20090626')等元组。

表 3.4 Allocation 表

字段	类型
ID	int
SegmentID	int

Allocation 表保存文件中数据所在的数据块信息，ID 是文件的唯一标识，SegmentID 为文件中数据所在数据块的标识。

表 3.5 Authorization 表

字段	类型
ID	int
AC	char

Authorization 表保存文件和目录访问授权的信息，ID 是文件或目录的唯一标识， AC 是文件或目录的访问授权。

表 3.6 Affiliation 表

字段	类型
ID	int
PID	int

Affiliation 表保存文件和目录与所有上层目录之间的所属关系，ID 是文件和目录的唯一标识，PID 为文件或目录的所有上层目录标识。

3.3 元数据操作算法设计

在 3.2 节中我们使用多个二维表保存文件的元数据信息，将元数据信息用二维表这种结构化的形式来组织存储，在这一节中我们设计对这些二维表的操作算法，用于完成用户访问请求中的元数据操作要求。我们分成对目录的操作、文件的创建与删除、文件属性的更改、文件的访问授权和文件的移动等几个方面给出具体的算法。

3.3.1 目录操作

由于使用层次结构保存文件与目录的结构信息，传统元数据管理策略只能依次打开访问路径中的各目录，逐级访问后才能找到对应文件的元数据，若访问路径的层次较多，则使得访问文件元数据需要较多的时间与空间开销。而文件哈希法会破坏目录树的层次结构，使得处理目录列表等操作命令较困难；同一目录所属的各文件与子目录之间缺乏排序机制，因此查找文件较困难。

在我们设计的基于二维表的元数据结构中，文件与目录之间的从属关系蕴含在 File 表中元组之间的关联中，对这个表进行查找和操作即能完成有关于处理目录的一系列操作命令。我们先给出操作命令--目录列表的处理算法：File 表中包含了所有文件和目录的访问路径以及父目录的信息，因此处理目录列表时仅需要查询 File 表；首先根据用户需要对目录内容列表的目录访问路径设为 *dp_name*，查找该目录所对应的标识 *id*，再依据获得的目录标识 *id* 查找 File 表中 DPID 与其相同的元组，即可获得该目录的内容列表。目录操作命令对应的 SQL 查询语句如下：

```
SELECT PathName
  FROM File
    WHERE DPID = (SELECT ID
                    FROM File
                  WHERE PathName = dp_name)
```

当处理修改目录属性的操作命令时，需要查询 File 表和 File-Attribute 表。首先依据访问路径查询 File 表获得目录标识 *id*，再更新 File-Attribute 表中该目录的属性。如更新目录访问路径为 *dp_name* 的修改时间为现在的时间（由 *now* 函数获得），则对应的 SQL 语句如下：

```
UPDATE File-Attribute
SET AttributeValue = now()
WHERE AttributeName = 'ModifyTime' and ID =
 (SELECT ID
    FROM File
      WHERE PathName = dp_name)
```

3.3.2 文件操作

对文件的操作包括创建、删除、查询、移动和复制等，我们分别给出相应的操作算法。创建文件时，其步骤如下：

步骤 1：首先在 File 表中插入一条元组，记录文件的访问路径、标识和父目录的标识等信息。

步骤 2：再将文件的每个属性作为一个元组插入到 File-Attribute 表中。

步骤 3：依据文件的访问路径，将文件的每个上层目录作为一个元组插入到 Affiliation 表中。

步骤 4：最后将文件数据所在的数据块信息，作为一个元组插入到 Allocation 表中。

当我们进行删除文件的操作时，所做的工作与创建文件相反。

快速查找文件是海量存储系统重要的要求之一，元数据以层次方式组织，通常需要以依次、逐层遍历目录路径内容的方式查找文件，使得查找文件所需的时

间与空间开销较大。我们在使用二维表保存文件元数据的基础上，建立索引，并利用文件访问路径缩小查找的范围，减少查找文件所需的时间与空间开销。当已知文件完整的访问路径时，可将访问路径作为关键字查询 File 表，利用索引快速获得文件的标识。当在某一目录下搜索文件时，可首先查找 File 表获得该目录的标识，再查找 Affiliation 表中 PID 值与其相同的元组，从而快速获得所需文件的标识。

移动文件时，只需要修改 File 表中 PathName 的值，用新的访问路径代替原有的访问路径即可。

复制文件时，则首先在 File 表、File-Attribute 表、Affiliation 表和 Allocation 表中查找需复制文件所对应的元组，用新的文件标识和访问路径替代后再添加到相应的表中。

3.3.3 文件属性的操作

不同类型文件需保存的属性个数、类型各不相同，File-Attribute 表中每个元组表示某个文件的一个属性，不限制文件属性的个数与类型，从而满足灵活保存文件属性的要求。查询文件的属性时，首先查询 File 表获得文件的标识，然后使用文件标识查询 File-Attribute 表，获得文件的属性。添加文件属性时，将文件标识、属性名、属性值和属性类型作为一个元组添加到 File-Attribute 表中。修改文件属性值时，在 File-Attribute 表中查找相应的元组，将新的属性值写入 AttributeValue 字段中。删除文件的属性时，查找 File-Attribute 表中 ID 值与文件标识相同，且 AttributeName 与要删除的属性名相同的元组。

3.3.4 访问授权的操作

传统元数据管理策略中，在更改访问授权后一般采用更新所属的所有文件和子目录或不更新而在访问文件时遍历访问路径上所有目录访问授权两种方法。前一种方法，在每次修改目录的访问授权后，需要大量的时间与空间开销用于逐个更新所有文件和子目录的访问授权；第二种方法则在访问文件时，需要大量的时间与空间开销，遍历文件访问路径上的各级目录，逐个比较它们的访问授权。这两类方法均需要大量的时间与空间开销。

通常系统中目录的数量远少于文件的数量，因此更新目录访问授权所需的时间与空间开销占总开销的较少部分；遍历层次型的目录树需要较多的时间与空间开销，因此应避免对文件访问路径中各级目录的遍历。

当修改目录的访问授权后，查找该目录下的所有子目录，用新访问授权更新 Authorization 表中对应的元组。在访问文件时，查找 Authorization 表中该文件和父目录的访问授权，进行比较，如果相同，则使用当前的访问授权值；如不相同，则将父目录的访问授权作为新的访问授权值，并更新该文件元组中 AC 字段的值。

3.4 性能分析

在前面的内容中，我们设计了多个二维表用于保存海量存储系统中的元数据信息，建立索引，并根据用户不同的访问请求的操作给出了对应不同操作命令的元数据操作方法，实现了高效、灵活的元数据管理策略。下面我们分别从管理元数据所需时间与空间开销和灵活性两个方面，分析基于 DBMS 元数据管理策略的性能。

3.4.1 管理元数据所需的时间与空间开销

传统的元数据管理策略中，以层次结构保存元数据信息，需要遍历访问路径中的各级目录才能获得文件的元数据，所需的时间与空间开销大。基于 DBMS 元数据管理策略中，使用多个二维表保存元数据信息，原来层次结构的文件与目录之间从属关系转换为元组之间的联系。此外针对 File 表、File-Attribute 表、Allocation 表、Authorization 表和 Affiliation 表分别建立了多个索引，改变了需遍历访问路径中各级目录的方法，有效地减少了查找元数据所需的时间与空间开销。

使用哈希法后，由于破坏了文件与目录之间的层次结构，使得执行目录列表等操作命令较困难。基于 DBMS 元数据管理策略中，未破坏文件与目录之间的层次结构，同时代之以查询更为方便的二维表结构保存文件与目录之间的关系，在执行目录列表等操作命令时，只需对 File 表进行两次查询，所需的时间与空间开销小，能有效地改变执行目录列表等操作命令所需时间与空间开销过大的问

题。

传统元数据管理策略在修改目录的访问授权后，需要大量的时间与空间开销修改所有的子目录及文件的访问授权，或者在访问文件时需要遍历访问路径中所有目录提取访问授权进行比较。基于 DBMS 元数据管理策略中，在修改目录的访问授权后，只修改所有子目录的访问授权信息，由于系统中目录数量远少于文件的数量，因此可减少大量的时间与空间开销；在访问文件时，只需要检查文件及父目录的访问授权，避免了遍历访问路径中各目录所需的大量时间与空间开销。

3.4.2 元数据管理的灵活性

不同类型文件需保存的属性个数、类型的需求各不相同，传统单一文件属性结构无法适应不同文件和应用的要求。而我们设计的基于 DBMS 元数据管理策略中，由 File-Attribute 表保存文件的属性信息，文件的每个属性对应表中一个元组，不同文件的属性个数、类型等可各不相同，从而实现了灵活保存和管理文件属性的功能。

在海量存储系统中移动文件时，使用传统元数据管理策略需要查找和修改多个目录项的内容，所需的时间与空间开销较大。而使用基于 DBMS 元数据管理策略时，文件与目录之间的从属关系由元组之间的关联进行表示，修改二维表中的少量数据项即可，所需的时间与空间开销小。

3.5 性能测试

我们实现基于 DBMS 的元数据管理策略的原型系统，构建相应的测试环境，建立测试数据集并进行测试与分析。

3.5.1 原型系统的实现

根据相关研究[31]，在文件系统中，读取文件元数据所涉及的元数据操作中，lookup 操作、getattr 操作和 read 操作所占的比重排在前三位，合起来约占 80%。其中，lookup 操作是查找文件，getattr 操作读取文件的属性，read 操作从磁盘读取数据到内存。所以，我们的设计测试实验主要是针对这三种常用的元数据操作

进行测试。

我们实现基于 DBMS 的元数据管理策略的原型系统，以二维表的形式保存元数据信息，使用 B 树建立索引，访问文件时，用文件的文件路径或标识作为关键字来查找；并实现现有的元数据管理策略的原型系统，以目录树的方式来保存元数据信息，依据目录之间的层次结构查找元数据，读取文件/usr/experiments/sam.c 的元数据，依次执行 getattr(usr)、lookup(usr)、getattr(experiments)、lookup(sam.c)和 getattr (sam.c)等操作。

3.5.2 测试环境的构建

我们是在 Linux 系统(内核版本为 2.6.9-42.14)上使用 C 语言开发基于 DBMS 的元数据管理策略和现有元数据管理策略两个原型系统，机器配置如表 3.7 所示。

表 3.7 测试环境的软硬件配置

CPU	Intel Pentium 4 2.93 GHz
内存	1024M
OS	Redhat Enterprise 4.0 (kernel:2.6.9-42.14)
硬盘	SATA Seagate 160G

依据文献[30]中的方法，编写程序，遍历 Linux 系统中各目录中的文件，获取文件和目录的元数据信息，共获取了 32557 个文件的元数据信息。由于机器配置和系统当前运行情况的不同对原型系统运行的时间影响较大，因此我们采用查找某个文件或目录元数据的过程中需比较的文件或目录次数作为衡量查找元数据所需时间开销的依据。

3.5.3 测试数据集

我们在测试原型系统时，采用随机型、正态性、多簇型和圆环型等方法构建多个人工数据集（Synthesized Datesets），构造了 10 个到 30000 个文件或目录元数据的访问请求。人工数据集主要包括以下几种类型：

（1）随机型的人工数据集：数据点以随机分布的形式分布在数据集的论域空间内。

（2）正态性的人工数据集：数据点以正态分布的形式分布在数据集的论域

空间内。

（3） 多簇型的人工数据集：数据点以集中分布的形式分布在数据及空间中的某几个部分。

（4） 圆环型的人工数据集：数据点以环型分布的形式分布，具有内外半径。

3.5.4 测试与分析

我们首先测试使用阶数不同的 B 树建立元数据索引时，原型系统查找元数据的性能；再测试当所保存的元数据总量固定而需处理数量不同的访问请求时，原型系统的性能；最后改变所保存的元数据总量，测试原型系统的性能。

3.5.4.1 改变 B 树阶数的测试与分析

元数据的索引是影响元数据管理效率的关键因素，在基于 DBMS 的元数据管理策略中使用 B 树建立索引，因此 B 树的阶数对管理元数据的性能有很大的影响。我们使用阶数为 3、4、5、6、7、8、9、10、12、15、18、20、25 和 30 的 B 树分别针对 3 万多条元数据建立索引，随机选择 30000 个访问请求，测试处理各访问请求时需读取与比较元数据的平均次数与最大次数。图 3.2 给出了测试的结果。

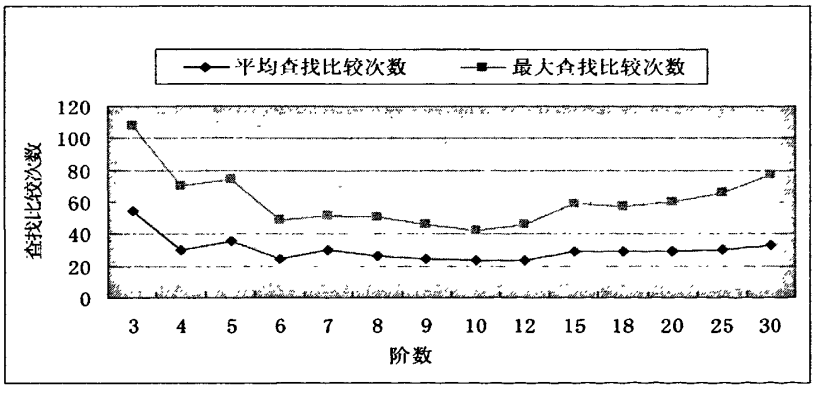

图 3.2 B 树阶数与性能比较

从图 3.2 我们可知，在阶数较小时，处理访问请求需读取与比较元数据的平均次数与最大次数较大；随着阶数的增加，处理访问请求需读取与比较元数据的平均次数与最大次数开始下降；在阶数为 10 时，出现最低点；之后处理访问请

求需读取与比较元数据的平均次数与最大次数又会随着阶数的增加而上升。

因此，针对原型系统中的 3 万多条元数据，采用 10 阶 B 树建立索引会使得原型系统具有较高的效率；在后续的测试中，B 树的阶数固定为 10。

3.5.4.2 访问请求数量不同时的测试与分析

基于 DBMS 的元数据管理策略的原型系统中，以二维表结构保存元数据，使用 10 阶 B 树建立索引，查找文件的元数据时以文件路径或标识作为关键字。现有元数据管理策略的原型系统中，依据文件系统的层次结构查找文件的元数据。先保存 1 万个元数据，分别测试访问请求数量为 10、20、50、100、200、500、1000、1500、2000、2500、3000、3500、4000、4500 和 5000 个时，查找文件元数据时需读取与比较元数据的平均次数与最大次数。结果如图 3.3 和图 3.4 所示。

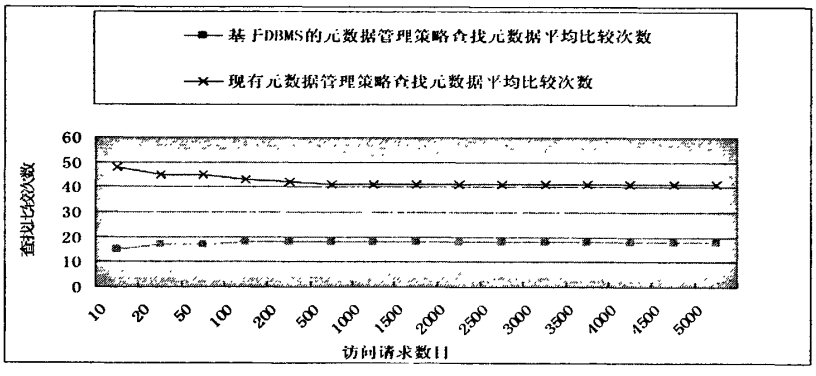

图 3.3 元数据个数为 10000 时，平均比较次数性能比较

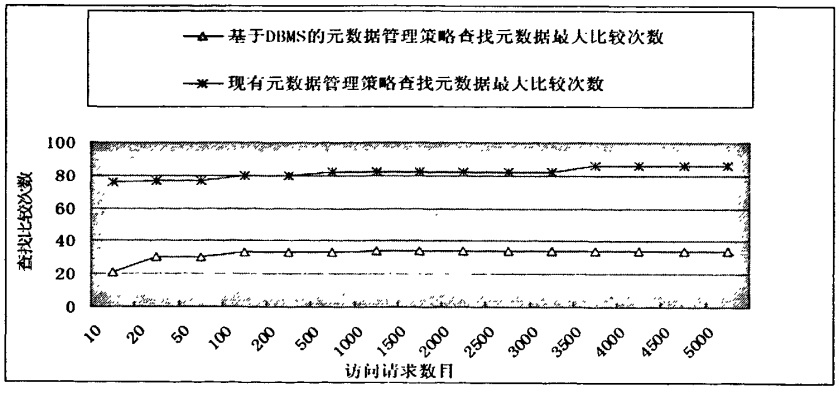

图 3.4 元数据个数为 10000 时，最大比较次数性能比较

保存 2 万个元数据，再分别测试访问请求数量为 10、20、50、100、200、500、1000、1500、2000、2500、3000、3500、4000、4500、5000、8000 和 10000 个时，查找文件元数据时需读取与比较元数据的平均次数与最大次数。结果如图 3.5 和图 3.6 所示。

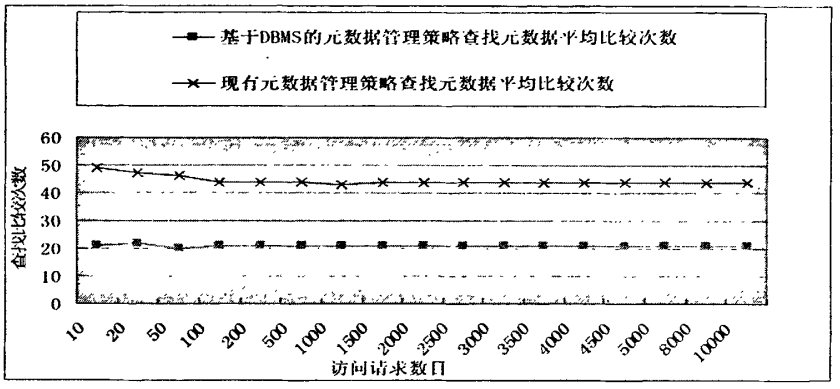

图 3.5 元数据个数为 20000 时，平均比较次数性能比较

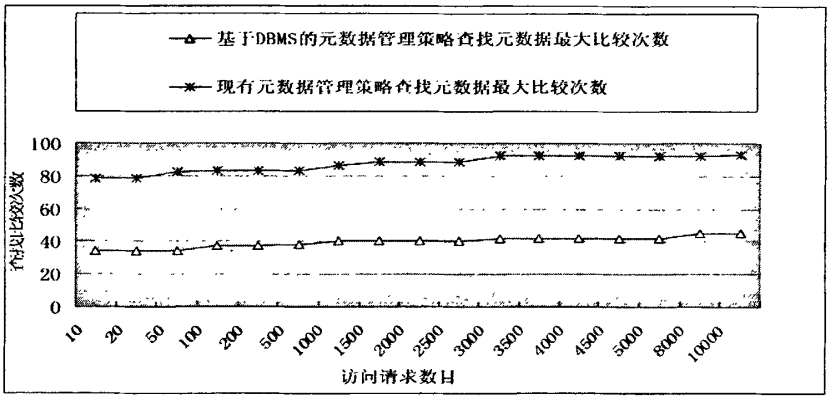

图 3.6 元数据个数为 20000 时，最大比较次数性能比较

保存 3 万个元数据，再分别测试访问请求数量为 10、20、50、100、200、500、1000、1500、2000、2500、3000、3500、4000、4500、5000、8000、15000 和 20000 个时，查找文件元数据时需读取与比较元数据的平均次数与最大次数。结果如图 3.7 和图 3.8 所示。

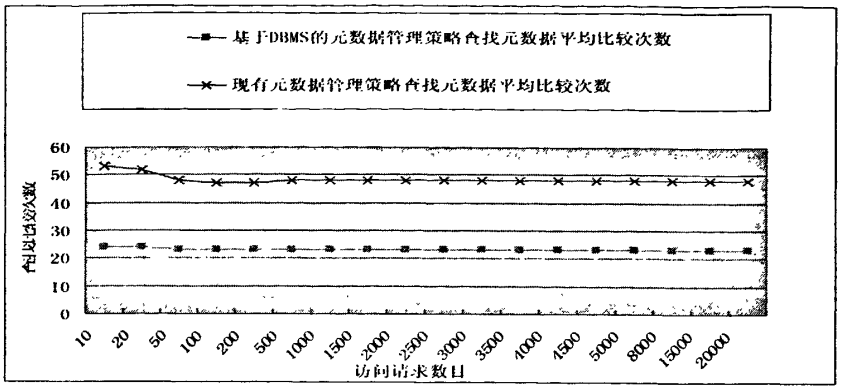

图 3.7 元数据个数为 30000 时，平均比较次数性能比较

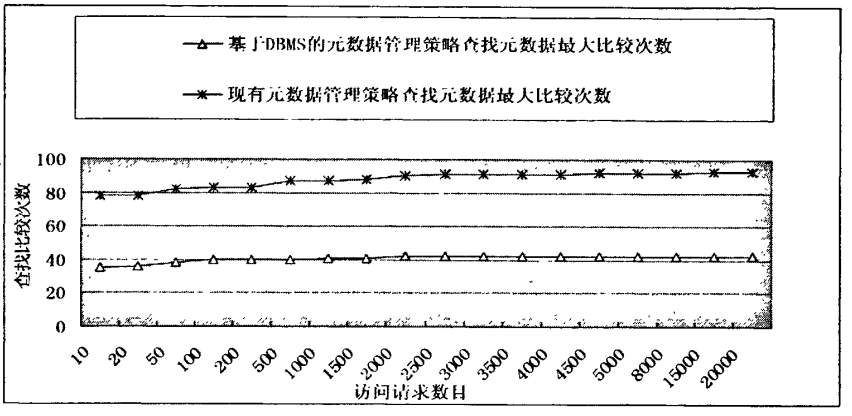

图 3.8 元数据个数为 30000 时，最大比较次数性能比较

从图 3.3-图 3.8 我们可知，使用基于 DBMS 元数据管理策略，当访问请求的数量不同时，处理访问请求需读取与比较元数据的平均次数与最大次数始终远小于现有元数据管理策略，说明基于 DBMS 元数据管理策略能有效的提高查找元数据的效率。

在访问请求数量较少时，由于目录的层次较深，会出现处理访问请求需读取与比较元数据的平均次数较大的问题，但随着访问请求数量的增加，很快趋向合理值；之后随着访问请求数量的增加，处理访问请求需读取与比较元数据的平均次数与最大次数也随之增加，但使用基于 DBMS 元数据管理策略时增加速度明显小于使用现有元数据管理策略，说明基于 DBMS 元数据管理策略有更好的适应性，具有更强的保持性能稳定的能力。

在元数据数量不同的多种测试环境中，在访问请求小于 2000 时，处理访问

请求需读取与比较元数据的平均次数与最大次数变化较大；在访问请求为 2000 左右时，处理访问请求需读取与比较元数据的平均次数与最大次数趋向稳定，变化较小，因此在 3.5.4.3 节中我们使用访问请求个数是 2000 个时进行测试。

3.5.4.3 元数据量不同时的测试与分析

依据 3.5.4.2 节中的测试结果，我们选择 2000 个访问请求，通过改变元数据的数量，测试改变元数据数据量时系统的性能。结果如图 3.9 和图 3.10 所示。

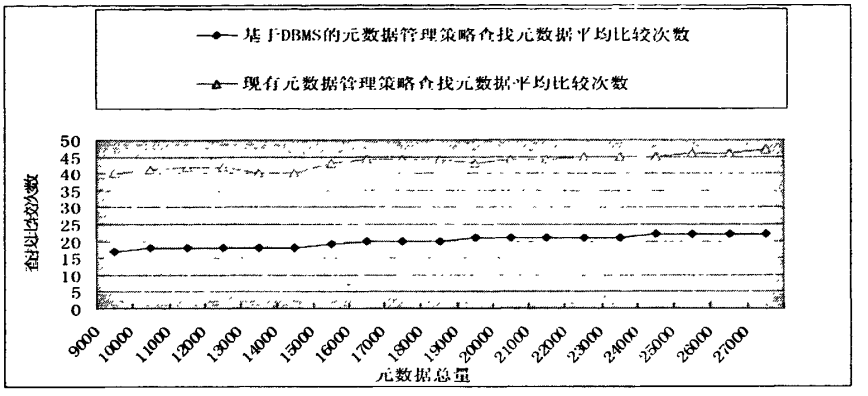

图 3.9 访问请求数为 2000 时，平均比较次数性能比较

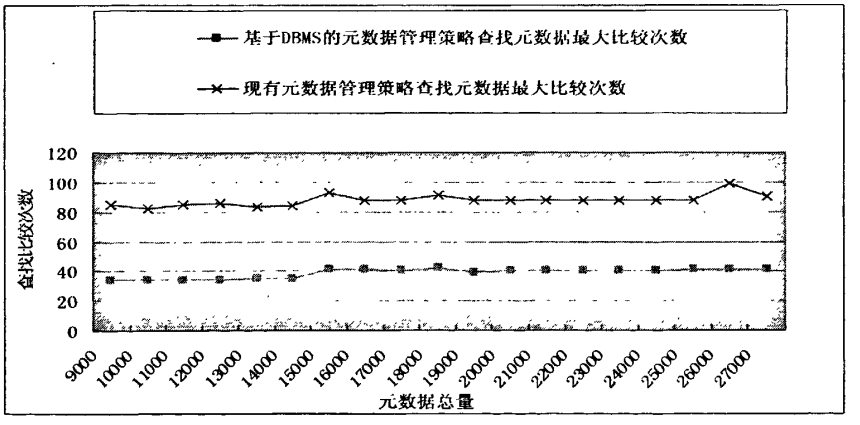

图 3.10 访问请求数为 2000 时，最大比较次数性能比较

从图 3.9 和 3.10 可知，随着元数据数量的增加，使用基于 DBMS 元数据管理策略处理访问请求需读取与比较元数据的平均次数与最大次数远小于使用现有元数据管理策略，说明基于 DBMS 元数据管理策略能有效的减少查找元数据所需的时间与空间开销；此外使用基于 DBMS 元数据管理策略时，处理访问请

求需读取与比较元数据的平均次数与最大次数变化幅度也小于使用现有元数据管理策略，说明基于 DBMS 元数据管理策略具有更强的适应能力，用于管理元数据能更好的保持性能的稳定。

3.6 本章小结

本章首先分析海量存储系统元数据管理的特性和要求，根据数据结构化存储的要求，设计了基于二维表的元数据保存结构，并给出用于完成用户访问请求的元数据操作算法。 分析了在海量存储系统中用于管理元数据信息所需的时间和空间开销以及适应不同运行环境的能力。实现了基于 DBMS 元数据管理策略原型系统，采集实际文件系统中的元数据，构建多种测试环境进行测试与分析，结果表明该策略能有效地减少管理元数据所需的时间和空间开销，提高管理元数据的灵活性，增强适应能力。

第四章 元数据的分级索引算法

随着存储技术的发展，海量存储已经越来越普遍。如何管理并使用好这些海量信息，成为海量存储面临的一个新的挑战。海量存储系统每天响应大量用户的海量元数据访问请求，所以元数据查询性能的优劣直接影响海量存储系统的性能。依据元数据的生命周期，对海量存储系统中的元数据进行分级，根据各级元数据不同的使用特性使用不同的方法建立索引。

本章在分析现有的元数据索引算法缺点和不足的基础上依据元数据的生命周期来对海量存储系统中的元数据进行分级设计，然后对不同级别的元数据分别建立索引来提高海量存储系统元数据的查询效率。

4.1 现有元数据索引算法的分析

信息社会每天都会产生大量需保存的数据，这使得海量存储系统成为当前研究和应用中的热点问题。但很多情况下使用者并不需要访问海量存储系统中保存的数据本身，而只需使用海量存储系统中的元数据即可。据统计，在 NFS 和 AFS 中，元数据的读和写的操作比例分别占到了文件系统操作请求总数的 75.4%和 82.2%，分别占用 49%和 63.5%的系统资源。此外在访问海量存储系统中保存的数据之前首先需要访问元数据服务器，依据文件名等信息查询和获得数据的大小、访问授权和保存位置等元数据信息后，才能读取相应存储设备中保存的数据，因此研究新型的元数据管理和查找方法是提高海量存储系统性能的重要手段。

海量存储系统中目前一般使用树型结构和基于哈希的元数据管理方法。在使用树型结构的元数据管理方法时，查找元数据需要逐层访问文件访问路径中的每个目录，时间与空间开销大；使用基于哈希的元数据管理方法时，在海量存储系统中保存的数据量不确定的情况下，确定哈希函数是非常困难的，在改变哈希函数后需要更新元数据的索引时，则需要大量的时间与空间开销。索引可以有较地提高查询海量存储系统中元数据的性能，使用树型结构的元数据管理方法时，可使用 B 树建立元数据的索引，但当海量存储系统保存的数据量很大时，借助于 B 树索引查找元数据所需要的时间与空间开销仍然较大，此外维护包含大量结点的

B 树索引也需要很大的时间和空间开销。海量存储系统中，数据被访问的频率、时间等因素有其自身的特性，因此针对传统元数据管理算法存在的查询元数据所需时间与空间开销大的问题，研究新型的元数据索引算法是提高海量存储系统元数据管理性能的重要方法。

1986 年美国麻省理工学院的 Popek 等人[15-17]提出的目录子树分区算法，由不同的元数据服务器分别管理一个或多个目录子树。由于采用目录层次树结构来管理元数据，查找元数据时需按照文件路径逐层访问，所需的时间与空间开销大。

1996 年 IBM 的 P.F.Corbett 等人[18]提出了基于哈希的元数据管理算法，使用文件路径计算哈希值，实现元数据的管理和查找。但哈希函数难以满足海量存储系统中保存不同数量数据时的要求，而修改哈希函数后需要大量的时间与空间开销调整元数据。

2003 年美国加利福尼亚圣克鲁兹大学存储系统研究中心的 Scott A.Brandt 等人[2]提出的 LH 法，混合了树型结构和基于哈希两类算法，使用文件路径计算哈希值，再使用该哈希值查询元数据查找表获得元数据。但同样存在选择哈希函数困难的问题，此外在文件或目录更名后需要更新所有客户机中保存的元数据查找表也使得所需的时间与空间开销较大。

海量存储系统中各类数据被访问的频率各不相同，有些数据会在短期内被连续访问，有些数据则可能在很长时间内从未被访问过，因此针对数据的时间特性，研究新型的索引算法是提高海量存储系统中元数据管理性能的重要方法。

4.2 基于数据生命周期的元数据分级算法

树型元数据索引算法对所有元数据都是同等对待的，查询元数据时需要遍历文件目录路径上的所有文件；哈希法虽然可以通过文件路径名来定位元数据，可是它破坏了目录的层次结构，在执行与目录有关的目录列表和访问授权管理等指令时，同样需要遍历所有的元数据信息，这些索引算法都未考虑元数据的时间特性，缺乏相应的性能优化机制。我们依据元数据的生命周期进行分级，为减少元数据管理所需的时间和空间开销奠定基础。

产品生命周期是 1966 年美国哈佛大学教授雷蒙德·弗农（Raymond Vernon）在其《产品周期中的国际投资与国际贸易》一文中首次提出的。产品生命周期

（PLC）是产品的市场寿命[32]，即一种新产品从开始进入市场到被市场淘汰的整个过程。弗农认为：产品和人的生命一样，也要经历形成、成长、成熟、衰退这样的周期。就产品而言，也就是要经历一个开发、引进、成长、成熟、衰退的阶段，产品生命周期的模型如图 4.1 所示。

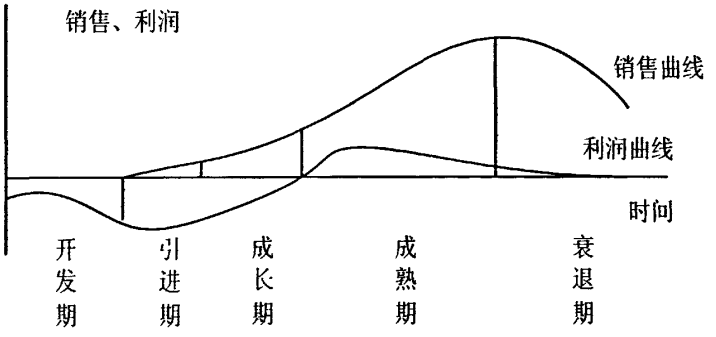

图 4.1 产品生命周期的模型[33]

无论是人、生态系统，还是企业和技术，在各自的生命周期内，都要经历从出生到成熟再到衰败的不同过程。数据同样有着产生、发展和消亡的过程。数据从它产生之日起就自然而然地进入了一个循环，经过收集、复制、访问、迁移、删除等多个步骤，最终完成了一个完整的生命周期，周而复始。数据刚生成时，处于生命周期的初期，这个阶段的数据被频繁使用，需要昂贵的存储技术和方式来保护数据。如 RAID 磁盘、定时拷贝、复制、多级备份等。随着数据“年龄”的增长，访问频率逐步降低。这时候，应该将这类数据从昂贵的磁盘迁移到较为便宜的存储设备上，不必再进行磁盘卷以及保留几份数据快照等。最后，当数据“老化”到不再被访问时，就要考虑将其删除或迁移，采用最便宜、最安全的存储方式，方便需要时的调用。这就是数据所经历的整个生命周期。数据生命周期技术是以成本与数据不断变化的价值相对应的方法，从创建到最终处置对数据进行全程管理。

数据周期与数据价值的关系如图 4.2 所示。数据由创建阶段开始，经过加工和处理进入发布使用阶段，此时数据价值达到顶峰，随后进入长期保存归档阶段，最后随着时间的推移信息价值降低、信息被删除。在我们所研究的海量存储系统的元数据管理中，系统中的元数据从其价值涵义上来说，也和产品生命周期一样，会经历产生、成熟和消亡的一个过程，会随着所处的“年龄”阶段的不同而在访

问频率、管理模式上起起落落。因此我们设想，是否可以在海量存储系统元数据索引的建立过程中引入数据生命周期的方法，减少查询元数据及更新索引所需的时间和空间开销，提高查询元数据的效率。

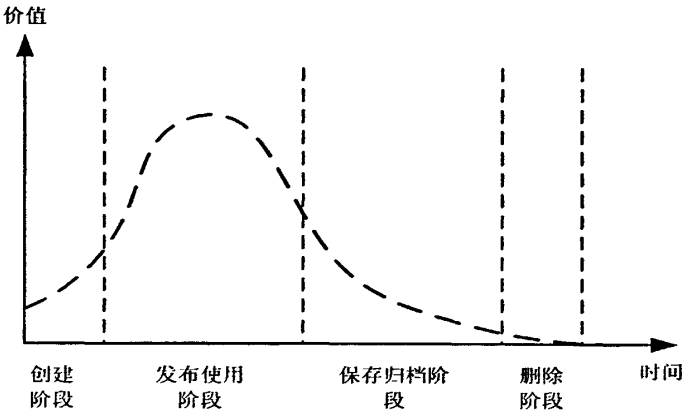

图4.2 数据周期与数据价值关系[34]

我们提出的元数据分级算法是鉴于海量存储系统中元数据的形成、成长、成熟、衰退这样的生命周期，又不同于数据生命周期技术中数据的在线、近线到离线的存储过程。文件系统的大部分文件和数据是不会被经常访问的，活跃文件和数据占整个系统的比例很小。这一部分文件和数据所占比例较小，但是由于它们被访问的频率很高，所以如何建立合理的索引，减少查找元数据所需的时间和空间开销是关键问题；而另一部分文件和数据虽然所占比例较大，但是它们被访问的频率较小，因此在满足元数据管理性能的基础上，如何合理减少元数据管理所需的额外时间和空间开销是关键问题。

树型和基于哈希的元数据管理算法中没有考虑元数据在被访问过程中与时间相关的特性，对所有元数据采用单一的管理算法是造成元数据管理与查询时所需时间与空间开销大的重要原因。大量的研究表明，当数据刚刚被保存到海量存储系统中时，对数据的访问和修改会很频繁；随着时间的推移，这些数据的访问频率会大大下降，直到在很长一段时间内几乎不被访问。在这个过程中，元数据的操作频率也同样随着数据被访问频率的变化而变化。对访问频率较高的元数据，查找所需的时间与空间开销小是关键因素，同时可适当牺牲管理元数据所需的额外时间与空间开销；而对访问频率较低的元数据，如何提高管理元数据时的

经济性是一个重要因素，同时可适当牺牲部分查询所需的时间与空间开销。因此在研究海量存储系统中的元数据索引算法时，对元数据分级是一个重要的方法。

我们依据海量存储系统中元数据的生命周期，设计基于生命周期的元数据分级算法，将元数据分为活跃元数据和非活跃元数据两级，为减少元数据管理与查询所需的时间与空间开销奠定基础。

定义元数据活跃度 A，作为衡量元数据活跃程度的依据，使用公式 4.1 计算 A 的值。

公式 4.1：$A=1-\frac{1}{f_r}+\frac{1}{t-t_c}$

其中 t_c 是创建元数据的时间，t 是系统当前的时间，f_r 是该元数据在单位时间内被访问的次数。

定义元数据活跃阀值 A_d，作为对元数据分级的依据。

在此基础上，我们依据元数据的活跃度，设计元数据分级算法，如公式 4.2 所示。

公式 4.2：$\begin{cases} 活跃, L(A,A_d)=1 \\ 非活跃, L(A,A_d)=-1 \end{cases}$

其中 L 为元数据等级判断函数，计算方法如公式 4.3 所示。

公式 4.3：$L=\mathrm{sgn}(A-A_d)$

使用元数据分级算法，我们可以将元数据分为活跃和非活跃两个级别：

（1）活跃元数据：该类元数据当前被访问的频率很高，因此减少查询元数据所需的时间与空间开销是关键。

（2）非活跃元数据：该类元数据当前被访问的频率较低或基本不被访问，因此如何使得管理元数据所需的时间与空间开销较小是关键。

假设当前系统中某个目录树如图 4.3 所示，该目录树中每个文件在单位时间内被访问的次数都基本相等，计算文件或目录的创建和访问时间时以年为单位，文件或目录名旁的括号中表示的是文件或目录的创建时间。

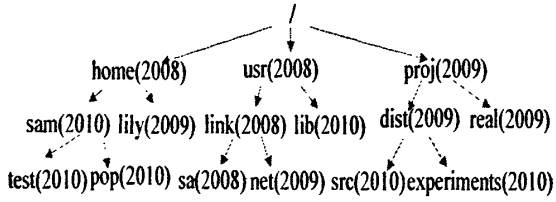

图 4.3 元数据未分级时的示意图

设当前的系统时间是 2010 年，元数据活跃阀值 A_d 的值为 1，使用元数据分级算法，可将图 4.3 中的目录树分解为图 4.4 中的活跃元数据和非活跃元数据两个子树。

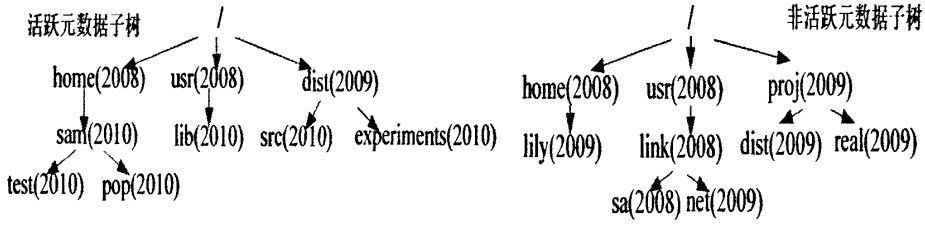

图 4.4 元数据分级后的示意图

4.3 活跃元数据分区索引算法

文件访问应包含对文件的元数据和数据访问两个步骤。在获得文件的数据属性和访问授权等元数据信息后，才能读取相应的数据。所以，文件访问时间由元数据访问时间和数据访问时间两部分组成，即：

Total_Time(file) = Time(metadata) + Time(data),

Ratio(metadata) = Time(metadata) / Time(file)

在大文件应用中，由于一次元数据访问可以支持大量的数据访问，元数据的访问效率对访问请求的影响不是很明显。但是，随着存储系统规模的不断扩大，大文件应用的聚合元数据请求数目也将非常可观，要求有效的元数据服务来支持。比如，ASCI[Lustre-SGSRFP2001]要求文件系统能支持 1.8*107 个目录、4.5*108 到 1.0*1010 个文件的文件系统规模，这将导致规模非常庞大的文件系统元数据请求。

活跃元数据当前被访问的频率很高，因此减少查询活跃元数据所需的时间与空间开销，是提高海量存储系统性能的重要问题。使用树型结构管理元数据时，

查询元数据需要依次、逐层遍历访问路径上的所有目录，所需的时间和空间开销大且不稳定；使用基于哈希的元数据管理方法时，由于破坏了文件之间的层次关系，使得执行目录列表等操作时所需的时间与空间开销大。当元数据的数量很大时，无论采用何种管理方法均无法避免查找性能下降的问题。因此我们首先设计元数据的分区方法，降低查找元数据的复杂度；同时引入 DBMS 的索引技术，对分区中的元数据建立索引，减少查找元数据所需的时间和空间开销。下面我们给出活跃元数据分区索引算法和查找元数据的流程。

文件之间的层次关系较好的体现了文件之间的关联，也使得元数据的操作具有较好的局部性。但单纯依据文件层次结构进行分区则无法避免各分区中元数据数量不均衡的情况，从而导致查找元数据性能的不稳定。我们在文件之间层次关系的基础上，综合考虑元数据数量的因素，设计分区索引算法。我们首先将活跃元数据进行分区，然后使用 Bloom Filter 生成分区中元数据的摘要串，最后使用 B 树以文件路径或标识为关键字建立元数据的索引。

由此将活跃元数据分成了若干分区，每个分区中仅包含部分活跃元数据，从而为减少查找活跃元数据所需的时间与空间开销奠定了基础；此外对每个分区使用 B 树建立了索引，保证了客户端能快速的查找到所需的活跃元数据。在此基础上我们给出活跃元数据的查找流程如图 4.5 所示。

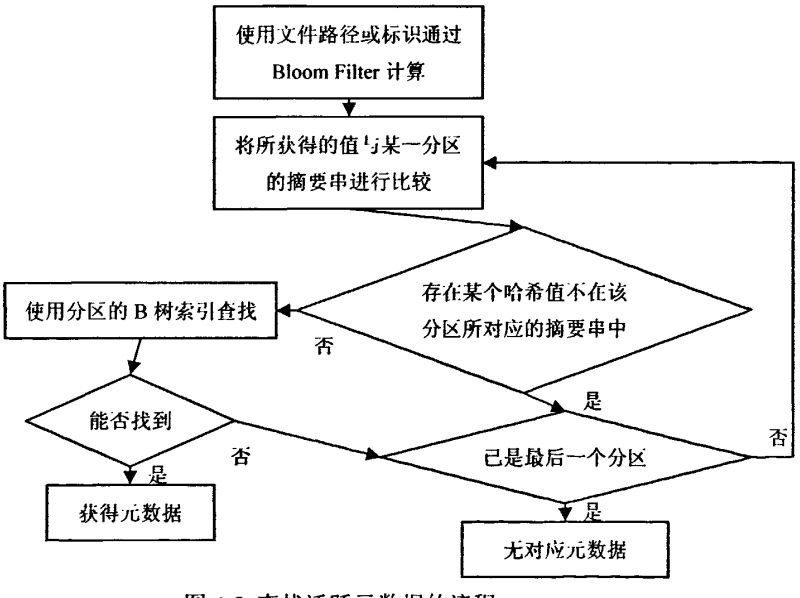

图 4.5 查找活跃元数据的流程

在分区索引算法的设计中，活跃元数据的分区机制和索引定位机制这两个步

骤至关重要。下面我们来具体介绍一下这两个步骤的设计过程。

4.3.1 分区机制

元数据的空间局部特性意味着元数据的属性值通过命名空间来聚集分布。例如，有关于 sam 的一些文件大都在/home/sam 这棵目录子树上而不会分散到整个命名空间。保持元数据的目录层次结构性，将整个大的命名空间分割为若干个小的独立部分并整合为若干独立的分区，这种层次划分方法可以有效地提高查询效率，因为满足一个查询要求的文件通常是只聚集在命名空间的一个部分。例如，查询 sam.pdf 这个文件，不再需要查询其他用户的主目录或系统文件目录。利用层次划分的方法使得查找时只考虑与查找内容相关的子树，这样减小了查找空间，相比现有的树型索引算法，提高了查找效率，缩短了查询的响应时间。但是这种划分法也存在一些问题，例如经过这种方法划分后可能导致分区目录文件的不均衡性，某些分区的目录文件过多，而某些分区的目录文件较少。如图 4.6 所示。从而引起目录文件较多的那些分区访问请求过多，响应时间长；而目录文件较少的那些分区访问请求少，严重影响了系统的性能。

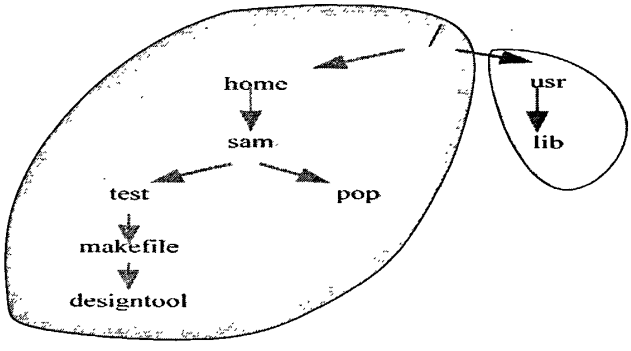

图4.6 元数据层次划分不均衡性示意图

如何利用元数据的空间局部特性来提高查询效率但又能避免因层次划分带来的不均衡性对我们研究活跃元数据索引算法尤为重要。在第三章中，我们提出引入 DBMS 技术将元数据用二维表这种结构化的形式来保存。而在一个实际应用系统上，对一个大表做聚合查询，当查询涉及的数据规模在 1 亿条记录内时，查询的响应时间在 10 分钟以内；当数据规模超过这个数字，查询的响应时间剧增，在一个小时内都无法返回结果。说明针对这类查询的优化处理需要想办法降

低其数据规模。根据上述查询优化处理的要求，我们设计活跃元数据的分区机制，具体步骤如下：

步骤 1：将 File 表内存储文件的顺序按照文件的层次结构进行分区；

步骤 2：统计各分区中元数据的数量；

步骤3：将元数据数量较多中的部分元数据移动到元数据数量较少的分区中，从而保持各分区中元数据数量基本均衡；

步骤 4：为每个子表分区建立 B 树索引。

这样的分区机制使得每个分区都有自己的索引子树，同时我们为每个子索引树建立分区元数据，它包含索引子树的路径信息，文件的统计信息以及特征文件。

特征文件简洁描述了分区的内容，引导查询指向相关分区以便缩小查询空间。我们可以将每个子表分区通过其保存的文件的文件路径或标识进行哈希，计算对应的 Bloom Filter 值（Bloom Filter Value，BFV）。Bloom Filter 最大特点就是使用短哈希串代表一个长字符串的空间，即将分区索引子树中所有值映射到同一个哈希空间，能够极大地缩短查询时间。

4.3.2 子索引定位机制

布鲁姆过滤器（Bloom Filter）是一种基于硬件的支持高速数据查询的实现方法，采用的是多个哈希函数并行计算的实现机制。它自 1970 年提出以来，已经被广泛应用到各个领域，包括路由查找[35]、数据包的分类[36]、元数据查找[20]等。

我们利用 Bloom Filter 进行子索引定位的工作步骤是：首先用一个长度为 m 的向量表示一个共有 n 个子表分区的活跃元数据集合 S={ s_1，s_2，…，s_n}。然后，假设源串 x_i 为某一活跃元数据，取值范围为{1,2,…,M}，将活跃元数据集合中所有的元素用 m 个比特单元构成活跃元数据的存储空间 S，初始时 m 个比特单元为 0。最后，使用 k 个相互独立的哈希函数 h_1，h_2，…，h_k，其值域为{0,1,2,…,m-1}，将所有源串 x_i 通过这 k 个独立的哈希函数映射到活跃元数据的存储空间 S，这样就可以通过一组哈希函数的映射过程来检验空间 S，以断定元素 x 是否属于集合 S。

利用4.2节提出的元数据分级函数L将海量存储系统的元数据划分为活跃元数据和非活跃元数据，我们利用布鲁姆过滤器来完成活跃元数据的子索引定位，将海量存储系统的全部活跃元数据视为检验空间S，将S分为n个子表分区，通过k个哈希函数来判断某一个访问请求的文件属于哪一个子表分区，其工作示意图如图4.7所示。

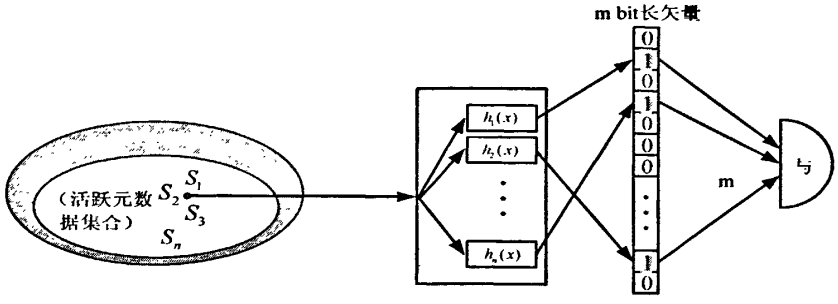

图4.7 子索引定位工作示意图

上述表示法由于哈希函数的冲突性，会导致某个查询元素不属于活跃元数据集合S，而被误判属于的可能性，简称误判率。假设hash函数是均匀随机分布的，在我们研究的海量存储系统中一个元素被误判的概率可作如下定义：令p表示位串中的某位在对n个活跃元数据集合元素表示结束时仍为0的概率，显然 $p=(1-\frac{1}{m})^{kn}\approx e^{-kn/m}$，式（4.4）中的 $f^{BF}(m,n,k)$ 就是表示算法的误判率。

$$f^{BF}(m,n,k)=(1-p)^k=(1-(1-\frac{1}{m})^{kn})^k\approx(1-e^{-kn/m})^k \qquad (4.4)$$

对式子（4.4）进行等价转化，得：

$$f^{BF}(m,n,k)=\exp(k\ln(1-e^{-kn/m})) \qquad (4.5)$$

设 $g(k)=k\ln(1-e^{-kn/m})$，并对式子两边求导，令 $\frac{dg}{dk}=0$，得到哈希函数个数的最小值。

$$k_{\min}=(\ln 2)(\frac{m}{n}) \qquad (4.6)$$

存在的错误率为：

$$f(k_{\min})=(1/2)^{k_{\min}}=(1/2)^{(\ln 2)m/n}=(0.6185)^{m/n} \qquad (4.7)$$

Bloom Filter 算法的性能主要取决于误判率，其越小，算法性能越好。因此，可以通过选择合适的哈希函数、哈希函数的个数及适合的 m/n 值，使错误率更小。根据上面的式子可以得出 k 与 m/n 之间成正比例关系，而误判率随着 m/n

或者 k 的增加而降低。

2004 年美国缅因大学的 Zhu Yifeng 等人[20]提出了将布鲁姆过滤器用在了元数据的查找上，设计了 HBA(Hierarchical Bloom filter Arrays)。该设计的思想是每个 MDS 包括两级 bloom filter array。第一级 bloom filter 叫 LRU BF，呈现出该 MDS 上 LRU 列表中被访问次数最高的文件；第二级 bloom filter 呈现出该 MDS 上包含的所有文件元数据分布。该 MDS 也保存了所有其他 MDS 上的两级 bloom filter 的副本。通过这种两级结构的布鲁姆过滤器结构能进行有效的元数据的查找，因此我们设想是否可以在之前活跃元数据的分区结果的基础上，设计基于布鲁姆过滤器的子索引定位机制。

Bloom Filter 是一种精简的信息表示方案，活跃元数据经过分区之后，每个子表分区的索引子树通过 Bloom Filter 算法生成一串子索引分区的摘要串，概括该索引分区里包含的元数据信息以及不包含的元数据信息。然后将这些摘要串传送给客户端，节省客户端大量的存储空间，减少信息传输，提高检索速度，并降低了访问延迟。如果摘要串认为该索引分区不含某个文件数据，则该分区必然没有这个文件数据，这样过滤了该文件；如该摘要串认为该索引分区可能含有某个文件数据，则该分区以很大的可能性存有这个文件数据。

查询文件 x 所属的子表分区，用基于 Bloom Filter 的查找定位机制实现算法步骤如下：

步骤 1：每个子表分区的索引子树通过 Bloom Filter 算法生成一串子索引分区的摘要串。

步骤 2：将文件 x 的文件路径或标识进行哈希计算。

步骤 3：将步骤 2 得出的哈希值通过 Bloom Filter 算法计算 BFV。

步骤 4：将步骤 3 得出的 BFV 与步骤 1 的各子表分区 BFV 进行比对，返回相应的子表分区的 ID。

在使用布鲁姆过滤器判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合，而出现假阳性误判（false positive）的情况。所以，当误判率低时采用 Bloom Filter 算法进行过滤能够一次找到文件所在的子表分区，查询复杂度为 O(1)；而在最坏情况下，即发生误判率为 100%时，则需查找出所有的子索引所组成的子表分区，查询复杂度为 O(log n)。因此，当

系统不断地扩大时，需合适地重新设计哈希函数、哈希函数的个数，以最大程度来降低误判率。虽然重新设计时增加了系统的一定开销，但能够满足系统访问请求元数据时的快速的查找和定位的要求。

4.4 非活跃元数据索引算法

非活跃元数据当前被访问的频率较低或基本不被访问，由于数量非常庞大，建立和维护索引需要大量的空间开销，因此减少额外的时间与空间开销是关键。基于哈希函数的管理方法虽然存在碰撞问题，但不需要额外的空间开销，能满足管理非活跃元数据时对额外时间与空间开销较小的要求。

但哈希函数的适应能力较差，合适的哈希函数在对非活跃元数据散列时碰撞少，使得管理和查找非活跃元数据所需的时间与空间开销小，反之会严重增加所需的时间与空间开销。而非活跃元数据所划分的不同分区有其各自的特性，单个哈希函数无法满足所有分区的要求。我们设计哈希函数查找表，为每个分区保存不同的哈希函数，满足各分区不同的要求，结构如表 4.1 所示。

表 4.1 哈希函数查找表

分区标识	哈希函数
0	哈希函数 0
1	哈希函数 1
…	…
n	哈希函数 n

我们设计非活跃元数据管理算法如下：

步骤 1：将非活跃元数据按照文件的层次结构进行分区；

步骤 2：统计各分区中元数据的数量；

步骤 3：将元数据数量较多中的部分元数据移动到元数据数量较少的分区中，从而保持各分区中元数据数量基本均衡；

步骤 4：使用 Bloom Filter，生成分区中元数据的摘要串；

步骤 5：在哈希函数选择表中查找该分区所使用的哈希函数；

步骤 6：使用该分区的哈希函数以文件路径或标识为关键字计算保存的位置。

与活跃元数据分区索引算法所不同的是，此时对单个分区使用哈希函数确定

元数据的位置，不需要额外的空间保存索引，为减少管理元数据所需的额外时间与空间开销奠定了基础。在此基础上我们给出非活跃元数据的查找流程如图 4.8 所示。

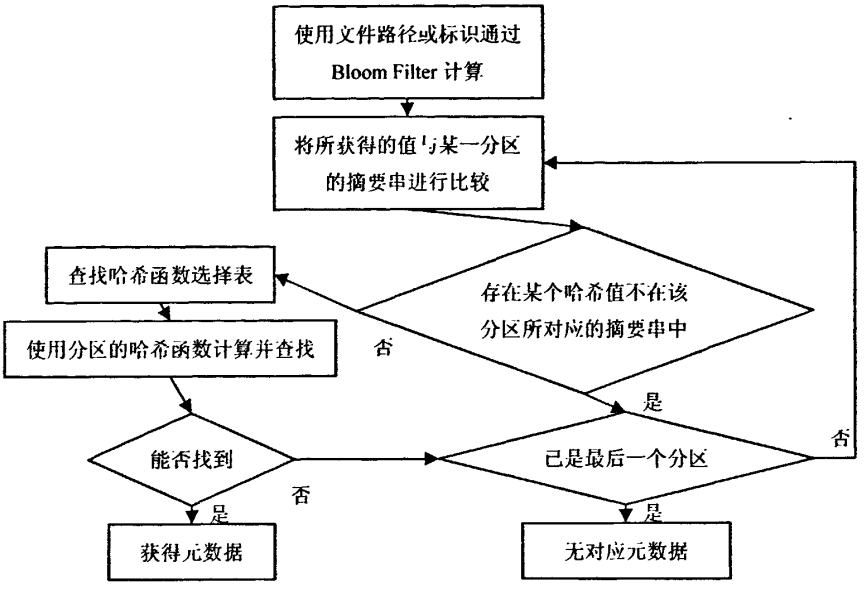

图 4.8 查找非活跃元数据的流程

4.5 性能分析

在前面几节内容中，我们设计了元数据的分级索引算法，将元数据分为活跃元数据和非活跃元数据，实现了高效、灵活的元数据查询功能。下面我们分别从查询元数据和更新索引所需时间与空间开销两个方面，分析元数据分级索引算法的性能。

4.5.1 查询元数据所需的时间与空间开销

我们依据元数据的生命周期，将元数据分为活跃元数据和非活跃元数据。访问文件元数据时，先根据活跃判别公式判断元数据级别，缩小了查询的空间，根据不同级别的元数据采用不同的索引方法，提高了查询的效率。

传统的元数据索引算法中，以层次结构保存元数据信息，需要遍历访问路径中的各级目录才能获得文件的元数据，所需的时间与空间开销大。而通过层次分区的方法将整棵目录树根据空间局部特性划分为若干子树，使得查找时只需查找与内容相关的子树，减小了查询空间，提高了查找的效率，缩短了查询时间，能

有效地改变执行查询文件命令时所需时间与空间开销过大的问题。

使用哈希法后，由于破坏了文件与目录之间的层次结构，消除了同一目录下的访问文件元数据的存储局部性，执行与目录有关的操作时仍需遍历目录路径上的所有文件。而我们在对活跃元数据索引的建立方法中，未破坏文件与目录之间的层次结构，同时代之以查询效率更高的划分子树的方法，在查询文件时，只需查找与该文件内容相关的子树，所需的时间与空间开销小。

4.5.2 更新索引所需的时间与空间开销

文件的更新必然带来索引的更新，如果采用树型索引算法，某个文件的更新可能会导致索引中大量节点的移动；而采用哈希法，某个文件的更新将会导致哈希值的改变，引起大量元数据的迁移。我们采用分区的方法，某个文件的更新只会引起相关分区索引的更新，所需的时间与空间的开销较小。

4.6 性能测试

我们实现元数据分级索引算法的原型系统，构建相应的测试环境，建立测试数据集并进行测试与分析。

4.6.1 原型系统的实现

我们实现元数据分级索引算法的原型系统，查找文件时，利用元数据分级函数判断是活跃元数据还是非活跃元数据，如果是活跃元数据则使用 Bloom Filter 定位分区，获取文件的元数据；如果是非活跃元数据使用哈希的方法获取文件的元数据；并实现现有的树型索引算法的原型系统，使用 B 树构建索引，进行元数据的查找。

4.6.2 测试环境的构建

我们在 Linux 系统（内核版本为 2.6.9-42.14）上用 C 语言开发了元数据的分级索引算法和现有的元数据索引算法两个原型系统。。原型系统运行和测试平台包括硬件和软件两个部分，机器的配置表如表 4.2 所示。

表 4.2 测试环境的软硬件配置

CPU	Intel Pentium 4 2.93 GHz
内存	1024M
OS	Redhat Enterprise 4.0 (kernel:2.6.9-42.14)
硬盘	SATA Seagate 160G

依据文献[30]中的方法，编写程序，遍历 Linux 系统中各目录中的文件，获取文件和目录的元数据信息，共获取了 32557 个文件的元数据信息。由于机器配置和系统当前运行情况的不同对原型系统运行的时间影响较大，因此我们采用查找某个文件或目录元数据的过程中需比较的文件或目录次数作为衡量查找元数据所需时间开销的依据。

4.6.3 测试数据集

我们在测试原型系统时，采用随机型、正态性、多簇型和圆环型等方法构建多个人工数据集，构造了 5-5000 个文件或目录元数据的访问请求。人工数据集主要包括以下几种类型：

（1） 随机型的人工数据集：数据点以随机分布的形式分布在数据集的论域空间内。

（2） 正态性的人工数据集：数据点以正态分布的形式分布在数据集的论域空间内。

（3） 多簇型的人工数据集：数据点以集中分布的形式分布在数据及空间中的某几个部分。

（4） 圆环型的人工数据集：数据点以环型分布的形式分布，具有内外半径。

4.6.4 测试与分析

测试时原型系统中共保存 3 万条元数据，设置访问请求的数量分别为 5、10、20、50、100、200、500、1000、1500、2000、3000、4000 和 5000 个，测试处理访问请求时需比较文件元数据的平均次数和最大次数。结果如图 4.6 和图 4.7 所示。

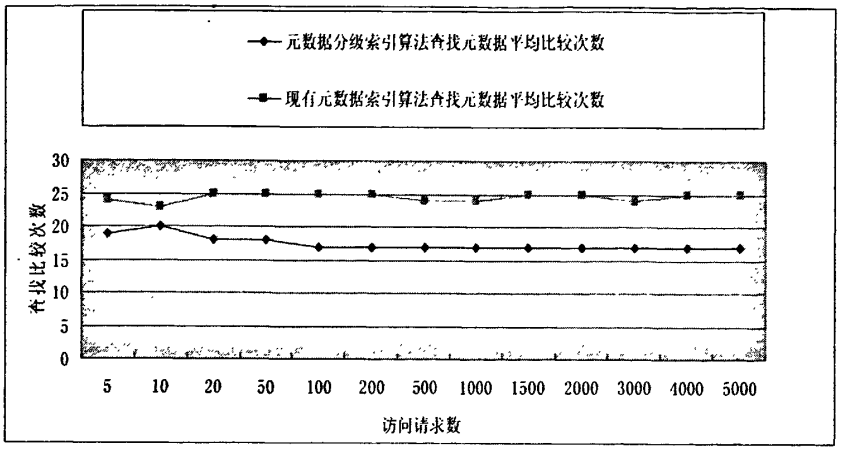

图 4.6 查找元数据平均比较次数性能比较

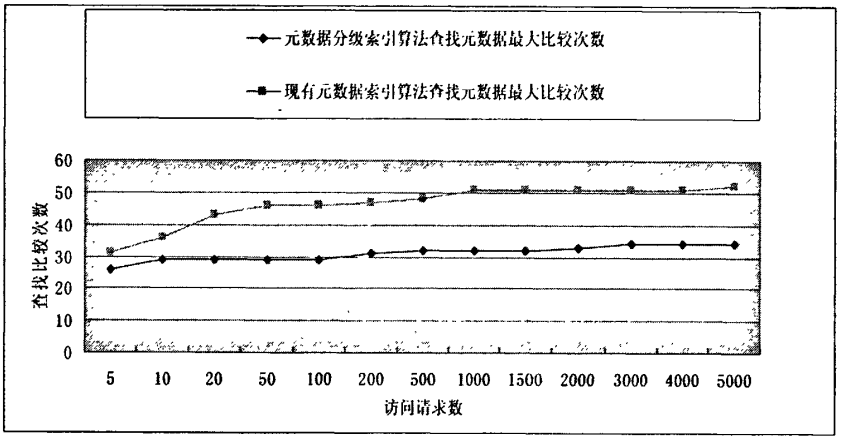

图 4.7 查找元数据最大比较次数性能比较

从图 4.6 和图 4.7 可知，使用元数据分级索引算法后，查找元数据所需比较元数据的平均次数与最大次数远小于未使用元数据分级索引算法时，说明元数据分级索引算法能有效的提高查找元数据的效率。

此外，随着访问请求数目的增加，使用元数据分级索引算法查找元数据所需比较元数据的平均次数逐渐降低并趋于平稳，而未使用分级索引算法查找元数据所需比较的平均次数始终高于或基本等于访问请求数目较小时的平均次数；随着访问请求数目的增加，使用元数据分级索引算法和未使用元数据分级索引算法查找元数据所需比较的最大次数都会随之增加，但使用元数据分级索引算法后增长幅度明显小于未使用元数据分级索引算法时，这说明元数据分级索引算法能有效地提高原型系统的适应能力，增强系统性能的稳定性。

4.7 本章小结

本章首先对现有的元数据索引算法进行了分析，针对现有算法的不足和进行高效的元数据查找的要求，发现元数据分级的方法是进行元数据查询机制优化的重要途径，为此将元数据依据其生命周期进行分级，然后根据不同级别元数据不同的使用特性使用不同的方法建立索引。通过性能分析，验证了该方法相比现有元数据索引算法能有效地减少查询元数据以及更新索引所需的时间和空间开销，提高了元数据查询的效率，具有较强的稳定性。实现了元数据分级索引算法的原型系统，采集实际文件系统中的元数据，构建多种测试环境进行测试与分析，结果表明元数据分级索引算法能有效的提高查询元数据的性能。

第五章 总结与展望

本章首先对本文所做的主要工作做简要回顾，并总结创新之处；接着指出系统有待完善之处，对未来进一步的研究工作进行展望。

5.1 工作总结

因特网的蓬勃发展、社会的数字化变革，导致网络上的数据呈爆炸式增长。根据互联网数据中心（Internet Data Center, IDC）的统计，至 2011 年，数据会以每年 60%的速度增长，仅在 2007 年就产生了 281EB（Exabyte）数据，到 2008 年底，全球数据内容的总量激增为 487EB[37-38]。如此海量的数据对存储系统提出了巨大要求，使得存储产业步入高速发展的黄金时期。海量存储系统的元数据，如文件的名字、属性和访问授权等信息集中由元数据服务器进行管理。访问存储系统中的数据时，首先需要访问 MDS，利用文件名等信息查询，获得数据属性和访问授权等信息后，才能读取相应的数据。所以元数据的管理性能的优劣对海量存储系统的 I/O 性能有着很大的影响。

本文从设计新型的元数据管理策略和元数据索引算法两方面展开研究，主要的工作如下：

（1）提出了基于 DBMS 的元数据管理策略

现有海量存储系统一般采用目录子树分区法和文件哈希法两类元数据管理策略，由于采用层次结构或哈希方法保存元数据，这些策略在执行改名、更改权限和目录内容列表等操作时，需要迁移或修改大量的元数据，会给海量存储系统的 I/O 性能造成很大影响。

本文首先分析了海量存储系统元数据管理的特性和要求，接着分析了现有元数据管理策略存在的不足，引入了 DBMS 的相关技术，提出了基于二维表的元数据保存结构并设计了各类元数据操作的流程；分析了在海量存储系统中用于管理元数据信息时所需的时间和空间开销以及适应不同运行环境的能力，并实现了基于 DBMS 元数据管理策略原型系统，采集实际文件系统中的元数据，构建多种测试环境进行测试与分析；验证了基于 DBMS 的元数据管理策略能有效地减

少管理元数据所需的时间和空间开销，提高管理元数据的灵活性，增强适应能力。

（2）提出了元数据分级索引算法

现有元数据索引算法缺乏针对元数据时间特性的优化机制，在用于查找海量存储系统中的元数据时，存在时间和空间开销大、索引维护困难等问题。如何减少查找元数据所需的时间与空间开销、提高索引的可维护性，是提高海量存储系统中元数据管理性能的重要手段。

我们依据元数据的生命周期，对海量存储系统中的元数据进行分级，根据各级元数据不同的使用特性使用不同的方法分别建立索引；分析了查找元数据与更新索引所需的时间与空间开销；实现了元数据分级索引算法的原型系统，采集实际文件系统中的元数据，构建多种测试环境进行测试与分析；验证了元数据分级索引算法能有效的减少查询元数据所需的时间与空间开销，缩短更新索引所需的时间。

5.2 下一步工作展望

本文着重于海量存储系统中元数据管理性能的研究，主要集中在元数据的保存和访问请求等关键问题。在很大程度上，系统的元数据请求处理性能和扩展能力依赖于元数据请求分布决策算法，如何提供有效的元数据请求分布决策是首先需要进一步深化的问题。除此之外，还需要在系统的实现上进行优化。

在今后的研究中，要对元数据负载均衡性能进行优化。动态的元数据请求分布管理的核心是元数据服务器负载的预测。在保证元数据请求的处理效率的前提下，尽可能地将用户的负载分布到多个服务器。元数据分布策略的优化，是需要进一步深入的研究内容。

参考文献

[1] 张江陵，冯丹．海量信息存储．北京：科学出版社，2003．1~8.

[2] Brandt S A, Lan Xue, Miller E L, *et al.* Efficient metadata management in large distributed file systems: Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage System and Technologies. San Diego:[s.n.], 2003:290-297.

[3] http://baike.baidu.com/view/107838.htm?fr=ala0_1_1.

[4] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. Communications Magazine, 2003, 41 (8): 84~90.

[5] D. Patterson, G. Gibson and R. Katz. A Case for Redundant arrays of Inexpensive Disks (RAID). In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of data. 1988. 109~116.

[6] J. Menon, M. Hartung. The IBM 3990 Disk Cache. In: Proceedings of the IEEE Computer Society International COMPCON Conference. 1988. 146~151.

[7] A. Vasudeva. A Case for Disk Array Storage System. In: Proceedings of Systems Design and Networks Conference, Mass Storage Trends and Systems Integration, ed. Kenneth Majithia. April 1988.

[8] P. M. Chen, E. K. Lee, G. A. Gibson et al. RAID: High-Performance, Reliable Secondary Storage. ACM Computing Surveys, 1994, 26(2): 145~185.

[9] R. H. Katz, G. A. Gibson and D. A. Patterson. Disk System Architectures for High Performance Computing. IEEE, 1989, 77(12): 1842~1858.

[10] （美）Marc Farley 著．SAN 存储区域网络．机械工业出版社，北京，2002.

[11] William J B, John R D, Jacob R L , *et al.* A Five-Year Study of File-System Metadata. ACM Trans. on Storage,2007,3(3):31-45.

[12] Paul T.Singley Metadata Management for Data Warehouses, University of Wisconsin Madison 1994.

[13] Arun Sen, Metadata Management: past, present and future, Mays Business School, Texas A&M University, College Station, TX 77843,USA,4 December 2002.

[14] Yifeng Zhu, Hong Jiang, Jun Wang, *et al.* HBA: Distributed Metadata Management for Large

Cluster-Based Storage Systems. IEEE Trans. on Parallel and Distributed Systems , 2008, 19(6):750-763.

[15] Popek G J, Rudisin G, Stoughton A, *et al.* Detection of Mutual Inconsistency in Distributed Systems[J]. IEEE Trans. on Software Engineering,1986,12(11):1067-1075.

[16] Satyanaray M, Kistler J J, Kumar P, *et al.* Coda: A Highly Available File System for Distributed Workstation Environments. IEEE Trans. on Computers, 1990, 39(4): 184-201.

[17] Ousterhout J K, Cherenson A R, Douglis F, et al. The Sprite network operating system. IEEE Computer, 1998, 21(2):23~36.

[18] Corbett P F, Feitelso D G. The Vesta parallel file system. ACM Trans. on Computer System, 1996,14(3):225-264.

[19] Jong-Hyeon Yun, Yong-Hun Park, Seok-Jae Lee, *et al.* Design and Implementation of a Non-Shared Metadata Server Cluster for Large Distributed File Systems. Computer Science and its Applications,2008.CSA'08 International Symposium on 13-15 Oct.2008 Page(S): 343-346.

[20] Yifeng Zhu, Hong Jiang, Jun Wang, *et al.* Hierarchical Bloom Filter Arrays (HBA): A Novel, Scalable Metadata Management System for Large Cluster-based Storage.CLUSTER 2004:165-174.

[21] Yu Hua, Feng Dan, Bin Xiao. TBF: An Efficient Data Architecture for Metadata Server in the Object-based Storage Network.Networks,2006.ICON'06.14th IEEE International Conference on Volume 1,Sept.2006 Page(s):1-6.

[22] Yinjin Fu, Nong Xiao, Enqiang Zhou. A Novel Dynamic Metadata Management Scheme for Large Distributed Storage Systems. High Performance Computing and Communications, 2008.HPPC'08. 10th IEEE International Conference on 25-27 Sept.2008 Page(s):987-992.

[23] Qingsong Wei,Bharadwaj Veeravalli, Wujuan Lin. WPAR:A Weight-based Metadata Management Strategy for Petabyte-scale Object Storage Systems. Fourth International Workshop on Storage Network Architecture and Parallel I/Os on 2008 Page(s):99-106.

[24] Yan Jie, Zhu YaoLong. A design of metadata server cluster in large distributed object-based storage. In: Proceedings of the21th IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies, 2004.

[25] 吴伟，谢长生，韩德志，等. 海量存储系统中高可扩展性元数据服务器集群设计.计算机

科学，2007,34(7):106-109.

[26] Dahlin, M. et al., Cooperative Caching: Using Remote Client Memory to Improve File System Performance, First OSDI, pp. 267-280, Nov. 1994.

[27] Gibson, G. A., Nagle, D.F., Amiri, K., *et al.* File Server Scaling With Network-attached Secure Disks. Performance Evaluation Review, vol. 25, no. 1, June 1997: 272-284.

[28] 苏 勇，周敬利，余胜生，等. 基于共享存储池的元数据服务器机群的设计研究. 小型微型计算机系统，2007，28(4):734-737.

[29] Mike Casey, Greg Forst. Storage decision: Tiered storage school. http://wp.bitpipe.com/resource/Storage_Decision/ArchTieredStorage/Tiered%20Storage%20101(MLEdit).pdf?site_cd=bp, 2006.

[30] 刘仲，周兴铭. 基于目录路径的元数据管理方法. 软件学报，2007, 18(2):236-245.

[31] Duchamp,D., Optimistic Lookup of Whole NFS Path in Single Operation, Summer USENIX Proceedings, Boston, MA, June 1994, pp 161-169.

[32] 信息生命周期管理.http://wiki.mbalib.com/wiki.

[33] Information Lifecycle Management, http://en.wikipedia.org/wiki.

[34] Akshat verma, Upendra Sharma, Jim Rubas. An Architecture for Lifecycle Management in Very Large File Systems[C].Monterey, CA, USA: Proceedings of the22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies(MSST2005), 2005.

[35] Sarang D, Praveen K, David E T. Longest p refix matching using bloom filters. In: Proceedings of SIGCOMM 2003. ACM. 2003. 201~21.

[36] Chang C C, Lee T F, Leu J J. Partition search filter and its performance analysis. Journal of Systems and Software, 1999, 47 (1): 35~ 43.

[37] IDC, EMC. The Diverse and Exploding Digital Universe. March, 2008. http://www.emc. com /collateral /analyst-reports/ diverse-exploding-digital-universe.pdf.

[38] IDC, EMC. As the Economy Contracts, the Digital Universe Expands. May, 2009. http://www.emc.com/collateral/demos/microsites/idc-digital-universe/iview.htm.

致 谢

时光飞逝，转眼间三年的硕士研究生涯已经接近尾声。值此论文结束之际，回首已逝岁月，往事历历在目，心中感慨良多。这篇论文能够得以顺利完成，我既体会到了辛勤劳动后的喜悦，又深深地体会到了这是离不开整个课题组成员的热心帮助。

值此硕士学位论文完成之际，谨向我的导师鞠时光教授致以深深的感谢和崇高的敬意。在攻读硕士学位期间，我的每一份收获都凝聚着鞠老师的心血。无论在学术还是生活上，鞠老师都给予我巨大的帮助。鞠老师严谨的治学态度、渊博的学识、敏锐的洞察力和科学的工作方法对我产生了巨大的影响，使我终身受益。

衷心地感谢张晓红老师、薛安荣老师和蔡涛老师，他们在我平时的生活和学习中给予了许多的帮助。我所研究课题的很多细节都得到了蔡老师的悉心指导。蔡老师以他严谨的学风，渊博的知识，勤恳的工作作风和实事求是的科学态度深深地感染着我，并将继续对我的成长产生很大的影响。

感谢刘扬宽师弟、李秀娟师妹以及同门师兄弟，在平时的学术探讨与交流中给我很大的启示，使我受益匪浅。

感谢培育我的江苏大学，感谢计算机学院所有的老师们!

最后，向养育我长大成人的父母致以崇高的敬意，多年来他们对我无私的关爱和精神上的支持是我不断取得成功的重要动力和保障。

发表论文

在攻读硕士学位期间，作者发表论文如下：

[1] 基于 DBMS 的元数据管理策略.《计算机应用研究》.第 27 卷第 4 期，2010 年 4 月，1297-1300.第一作者

海量存储系统中元数据管理机制的研究

作者：吴婷
学位授予单位：江苏大学