中国高校第一本闪存数据库研究专著

历时两年，倾心之作　内容原创，品质保证
网络发布，免费共享　版权所有，侵权必究
243 页，　40 余万字，　系统研究闪存数据库

闪存数据库概念与技术

（版本号：2015 年 9 月 13 日首发）

林子雨 编著

厦门大学数据库实验室

http://dblab.xmu.edu.cn

2015 年 9 月 13 日

作者介绍

林子雨博士，厦门大学计算机科学系助理教授，中国高校首个“数字教师”提出者和建设者（http://www.cs.xmu.edu.cn/linziyu）。于 2001 年获得福州大学水利水电专业学士学位，2005 年获得厦门大学计算机专业硕士学位，2009 年获得北京大学计算机专业博士学位。主要研究方向为数据库、数据仓库、数据挖掘、大数据和云计算，发表期刊和会议学术论文多篇。曾作为志愿者翻译了 Google Spanner、BigTable 和《Architecture of a Database System》等大量英文学术资料，与广大网友分享，深受欢迎。2013 年在厦门大学开设《大数据技术基础》课程，并因在教学领域的突出贡献和学生的认可，成为 2013 年度厦门大学教学类奖教金获得者。编著出版了中国高校第一本系统介绍大数据知识的专业教材《大数据技术原理与应用——概念、存储、处理、分析与应用》，2015 年 8 月由人民邮电出版社正式出版发行，并成为当当、京东等网店畅销书籍。

E-mail: ziyulin@xmu.edu.cn

个人主页：http://www.cs.xmu.edu.cn/linziyu/

数据库实验室网站：http://dblab.xmu.edu.cn

扫一扫访问个人主页

著作权声明

声明人：林子雨

2015 年 9 月于厦门大学数据库实验室

推荐教材

《大数据技术原理与应用——概念、存储、处理、分析与应用》，由厦门大学计算机科学系助理教授林子雨博士编著，是中国高校第一本系统介绍大数据知识的专业教材。2015 年 8 月由人民邮电出版社出版发行，并成为京东、当当等网店畅销书籍。

本书定位为大数据技术入门教材，为读者搭建起通向“大数据知识空间”的桥梁和纽带。本书系统梳理总结大数据相关技术，介绍大数据技术的基本原理和大数据主要应用，帮助读者形成对大数据知识体系及其应用领域的轮廓性认识，为读者在大数据领域“深耕细作”奠定基础、指明方向。在本书的基础上，感兴趣的读者可以通过其他诸如《Hadoop 权威指南》等工具书，继续深入学习和实践大数据相关技术。

全书共有 13 章，系统地论述了大数据的基本概念、大数据处理架构 Hadoop、分布式文件系统 HDFS、分布式数据 库 HBase、NoSQL 数据库、云数据库、分布式并行编程模型 MapReduce、流计算、图计算、数据可视化以及大数据在互联网、生物医学和物流等各个领域的应用。在 Hadoop、HDFS、HBase 和 MapReduce 等重要章节，安排了入门级的实践操作，让读者更好地学习和掌握大数据关键技术。

本书可以作为高等院校计算机专业、信息管理等相关专业的大数据课程教材，也可供相关技术人员参考、学习、培训之用。

欢迎访问教材官方网站：http://dblab.xmu.edu.cn/post/bigdata

扫一扫访问教材官网

公益学术行为历史

林子雨将自己花费大量时间整理的研究资料，全部免费发布到网络与广大网友分享：

- 2010 年 7 月，翻译《Google BigTable》英文学术论文，并免费发布
- 2012 年 5 月，翻译第 1 届、第 2 届和第 5 届超大数据库会议英文大会报告（XLDB），并免费发布；
- 2012 年 9 月，全国第一个翻译了《Google Spanner》英文学术论文，并免费发布；
- 2013 年 9 月，全国第一个翻译了《Architecture of a Database System》英文学术论文，并免费发布。

更多资源欢迎访问林子雨个人主页和数据库实验室网站获取：

个人主页：http://www.cs.xmu.edu.cn/linziyu/

数据库实验室网站：http://dblab.xmu.edu.cn

前 言

在过去的几十年里，传统的机械式硬盘（或称为普通硬盘）一直都是企业广泛采用的存储介质。但是，随着闪存技术的不断发展，基于闪存的存储设备被认为具有很大的潜力可以取代机械式硬盘，并为企业的各种应用获得更高的性能。闪存具有速度快、体积小、质量轻、能耗低、抗震等特点，而且是非易失的，即使断电也不会丢失信息。由于闪存的优良特性，它已经广泛应用于消费类电子产品中，比如 PDA、MP3 播放器、移动电话和数码相机。闪存芯片还被封装成不同的产品，比如 CF 卡、SD 卡、迷你 SD 卡、微型 SD 卡和 USB 棒，或者有些闪存芯片会被封装成闪存存储设备（比如基于闪存的固态盘），配备了标准的 ATA 总线，可以连接到其他宿主设备上面。市场上也已经出现一些新型的个人计算机产品，完全抛弃了机械式硬盘，转而采用基于闪存的固态盘。

图灵奖得主 Jim Gray 在 2008 年曾经作出预测：“闪存是磁盘，磁盘是磁带，磁带将消亡”，目前，这个预测在许多应用中已经成为现实。闪存被认为具有很大潜力可以取代机械式硬盘，主要有以下几个方面的原因：

（1）**机械式硬盘自身的局限性**：机械式硬盘包含了磁头和转动部件，在读取数据时有一个寻道的过程，通过转动盘片和移动磁头的位置，来找到数据在机械式硬盘中的存储位置，然后，才能进行读写。因此，机械式硬盘具有很高的顺序读写性能，但是，在面对随机读写负载时，性能就会变得很差，因为，随机读写需要不断反复移动磁头进行寻址。而且，顺序读写速度和随机读写速度二者之间的差距还在不断扩大。在 I/O 开销中，机械式硬盘的寻址时间是最耗时的部分。基于机械式硬盘存储的操作系统都会采用不同的优化策略来分摊寻址代价，比如预抓取和磁盘调度计划，但是，无法从根本上解决这个问题。因此，对于企业应用而言，如果包含明显的随机读写负载或者缺少数据访问的局部性，那么，采用机械式硬盘存储数据将会严重制约应用系统的整体性能。

（2）**闪存的优良特性**：与机械式硬盘相比，基于闪存的存储设备具有更快的数据访问速度，更小的体积，更低的能耗，抗震性也更好。闪存在这个方面的优良特性，是机械式硬盘所无法媲美的，很好地满足了掌上电子产品对存储介质的要求。

（3）**闪存的容量增加和价格下降**：随着技术和制造工艺的改进，闪存的存储容量不断提高。据预计，直到 2012 年，NAND 闪存的存储密度都会每年翻一倍。闪存芯片之所以遵循摩尔定律，是因为它和集成电路一样都使用了类似的制造技术和工艺。在容量不断提高的同时，闪存的“每字节价格”却在不断下降，这使得闪存和低端、低容量的机械式硬盘设备的竞争力逐渐加强。

数据库是企业构建各种高级应用的基础，经过多年的发展，市场上已经存在可以满足不同企业应用需求的各种数据库产品，比如 Teredata、Oralce、SQL Server 和 MySQL 等等。当前的数据库产品大都采用基于磁盘的存储系统。随着闪存技术的发展，一些存储系统制造商，开始提供 TB 级别的、基于闪存的存储解决方案，其中一个主要目的就是应用于大规模数据库服务器[LeeMPKK08]。另外，随着闪存容量的不断提高，在移动设备中使用嵌入式 DBMS 已经变得越来越普遍[GanesanMS07][NathK07]，可以用来高效地对设备中的文件进行存储、检索和导航。

但是，由于闪存的读写特性和磁盘具有很大的区别，如果直接把传统的数据库应用到基于闪存的存储设备上，是无法获得好的性能的。虽然在实际应用中，硬件都是隐藏在接口后面，比如 SCSI 协议或块设备 API，但是，在过去三十年里，数据库应用都是为机械式硬盘这类旋转磁盘设备而优化的，这类旋转设备的特点是：具有固定的顺序带宽，但是，具有很大的机械延迟，它严重影响了随机 I/O 的性能。数据库系统被认为是专门为磁盘量身定制的一种非常典型的应用，从查询优化到 SQL 操作，再到底层的磁盘管理，都假设底层存储采

用了具有较长随机访问延迟的旋转磁盘设备。比如，对于基于磁盘行为而设计的查询优化器而言，它所做出的查询优化决定，可能很不适合用在闪存上。因此，数据库自身的特性和闪存的特性，决定了必须开展相关的研究，使得数据库应用在闪存上能够取得好的性能。

作者多年来在国家自然科学基金、福建省自然科学基金和中央高校基金科研业务费的支持下，以企业级数据库和数据仓库为应用背景，针对闪存环境下的数据库技术进行了深入研究。同时，作者一直承担厦门大学计算机科学系本科生专业课程《数据库系统原理》和研究生专业课程《分布式数据库技术》《大数据技术基础》的教学工作。本书是在丰富的教学实践和扎实的科研工作的基础上撰写的。

本书重点介绍闪存数据库的基本理论和关键技术。本书共分为 12 章，内容包括闪存和固态盘、闪存文件系统、闪存转换层、闪存数据库概述、闪存数据库存储管理、闪存数据库缓冲区管理、闪存数据库索引、闪存数据库查询处理、闪存数据库事务管理、基于混合存储系统的数据库、闪存数据库实验环境的搭建和基于闪存的键值存储。

第 1 章主要介绍闪存和固态盘的基础知识，包括闪存的原理、分类、特性和应用，固态盘的特性、产品、应用，固态盘的内部并行结构，以及固态盘性能分析。

第 2 章主要介绍闪存文件系统，包括 JFFS2 和 Yaffs。

第 3 章主要介绍闪存转换层（FTL），包括 FTL 的基本功能，页级别、块级别和混合 FTL 机制，以及双模式 FTL。

第 4 章给出闪存数据库概述，包括设计面向闪存的 DBMS 的必要性、闪存特性对 DBMS 设计的影响、面向闪存的 DBMS 的设计考虑因素和技术路线。

第 5 章主要介绍闪存数据库存储管理，包括基于页的方法、基于日志的方法、基于页差异的日志方法和 StableBuffer 方法。

第 6 章主要介绍闪存数据库缓冲区管理，包括重新设计面向闪存的缓冲区替换策略的必要性、设计面向闪存的缓冲区替换策略的考虑因素和关键技术，并描述一些具有代表性的方法，比如 CFLRU、CFDC、LRU-WSR、CCF-LRU、AD-LRU、FOR、CASA 等。

第 7 章主要介绍闪存数据库索引，包括基于日志的 B-树索引、B+-树索引、自适应 B+-树索引、FD-树和 LA-树等。

第 8 章主要介绍闪存数据库查询处理，包括 PAX 页布局模型、连接索引和 Jive 连接算法、基于 PAX 的 RARE 连接算法、基于 PAX 模型和连接索引的 FlashJoin、DigestJoin 等。

第 9 章主要介绍闪存数据库事务管理，描述传统的事务恢复机制及其在闪存数据库中的表现，并给出一些具有代表性的闪存数据库事务管理方法，比如 Transactional FTL、TxFlash、Flag Commit、IPL 和 OPL 等。

第 10 章主要介绍基于混合存储系统的数据库，包括闪存可以代替磁盘作为特种数据的存储介质、闪存可以作为介于磁盘和内存之间的缓存、基于数据访问模式的混合存储系统、基于语义信息的系统框架 hStoreage-DB 等。

第 11 章主要介绍闪存数据库实验环境的搭建，包括如何使用闪存模拟器开展实验和如何在 PostgreSQL 开源数据库的真实环境下开展实验。

第 12 章介绍基于闪存的键值存储。

闪存数据库是一个比较新的领域，目前缺乏关于这个领域的全面系统的介绍，作者通过阅读和分析国内外大量相关论文和课题资料，并结合自己的研究，撰写了本书，以期尽力跟踪该领域的最新技术进展情况。但由于作者学识有限，本书难免存在一些不足之处，敬请专家和学者批评指正。

林子雨
厦门大学数据库实验室
2015 年 9 月

目 录

第一篇 基础篇

第 1 章 闪存和固态盘

闪存是一种新型的存储技术，已经在社会生产和生活的各个领域得到广泛应用。闪存具有速度快、体积小、质量轻、能耗低、抗震等特点，可以很容易被封装成各种存储产品，比如固态盘，取代传统硬盘作为各种应用系统的底层存储介质。

本章内容首先简要回顾计算机存储技术的发展历史，然后介绍闪存的分类、特性和应用，最后，介绍基于闪存的固态盘。

[概念区分]

一般意义上的硬盘，包括机械式硬盘和固态硬盘。机械式硬盘，也称为普通硬盘，是磁盘的一种，靠盘片表面的磁性材料存储信息。当前 PC 机中配置的硬盘大都是机械式硬盘，这种机械式硬盘中包含了盘片、旋转轴和磁头，通过盘片的转动和磁头的移动来读取数据。固态硬盘，也可以简称为“固态盘”，是一种纯粹的电子存储设备，是一种半导体存储器，不包含任何机械部件，靠电路控制存储数据。固态盘可以采用多种不同类型的半导体存储芯片，比如闪存或者相变内存（PRAM），但是，当前市场上的绝大多数固态盘产品都采用闪存作为存储介质。因此，本书中的固态盘，特指采用闪存的固态盘。此外，为了避免两种不同类型的硬盘——机械式硬盘和固态硬盘——发生概念混淆，本书中，经常会用磁盘来指代机械式硬盘。

1.1 计算机存储技术

计算机已经广泛应用于人们的生产和日常生活，为人类社会的进步做出了巨大的贡献。从第一台现代意义的通用计算机 EDVAC (Electronic Discrete variable Automatic Computer)，到当前最先进的计算机，都采用了冯.诺依曼体系结构，即一台计算机是由五个基本部分组成的，包括运算器、控制器、存储器、输入装置、输出装置，程序和数据放在一起存储，在程序的控制下自动完成操作。

存储器是计算机系统中的记忆设备，用来存放程序和数据，包括输入的原始数据、计算机程序、中间运行结果和最终运行结果等。存储器根据控制器指定的位置存入和取出信息。存储器为计算机赋予了记忆功能，保证计算机系统正常开展各项工作。

存储器按照用途来划分，可以分为主存储器（内存）和辅助存储器（外存）。内存访问速度快，容量小，价格高，属于易失性存储，一旦断电，就会丢失所有信息，因此，通常用来暂时存放当前正在运行的程序和数据。外存访问速度要比内存慢许多，但是，容量大，价格低，属于非易失性存储，断电后也不会丢失信息，通常用来长期保存数据。

存储器按照存储介质来划分，可以分为半导体存储器和磁表面存储器。半导体存储器是由半导体器件组成的，主要包括随机存储器（RAM：Random Access Memory）、只读存储器（ROM：Read Only Memory）和高速缓存（Cache）：

- 随机存储器 RAM：包括静态随机存储器 SRAM（Static RAM）和动态随机存储器 DRAM（Dynamic RAM）。SRAM 不需要刷新电路就可以保存它内部存储的数据，相反，DRAM 则要求每隔一段时间就要刷新充电一次，否则，内部的数据就会丢失，因此，SRAM 具有较高的性能。但是，SRAM 存在一个明显的缺点——集成度较低，这意味着，在设计产品时，在相同的容量下，DRAM 内存产品体积较小，而 SRAM 却具有较大的体积，而且功耗较大。
- 只读存储器 ROM：对于只读存储器 ROM 而言，在制造的时候，信息（数据或程序）就会被永久保存到 ROM 中，一般只能读出，不能写入，即使发生断电，ROM 中的信息也不会丢失。ROM 一般用于存放计算机的基础程序和数据，比如，BIOS 就是一个永久保存在 ROM 中的一个软件，是操作系统输入输出管理系统的一部分，

包含了自检程序、基本启动程序和基本的硬件驱动程序等，主要用来负责机器的启动和系统中重要硬件的控制和驱动，并为高层软件提供基层调用。

- 高速缓存(Cache)：是介于内存与 CPU 之间的一级存储器，速度接近于 CPU，一般而言它不会采用 DRAM 技术，而是使用昂贵但较快速的 SRAM 技术。

磁表面存储器是由磁性材料做成，简称“磁盘”。当前比较常见的磁盘设备就是传统的机械式硬盘（HDD:Hard Drive Disk），或称为普通硬盘。机械式硬盘是计算机主要的存储媒介之一，既可以作为输入设备，也可作为输出设备。绝大多数机械式硬盘中，包括磁盘驱动器、适配器及盘片（如图[Hard-disk-drive](b)所示）在内的各个部件，被永久性地密封固定在硬盘驱动器中（如图[Hard-disk-drive](a)所示），避免和外界物体直接基础，确保数据的安全存取。机械式硬盘通常由一个或者多个铝制或者玻璃制的盘片组成，这些盘片外表面覆盖着一层铁磁性材料，用来记录数据。盘片的表面会被划分成多个磁道，每个磁道又被划分成多个扇区，每个扇区的大小是固定的 512 字节。如图[Hard-disk-drive](c)所示，在存取数据时，通过磁头的移动和盘片的转动这二者相配合，就可以到达指定的扇区读取数据，这个过程被称为“寻址”。机械式硬盘的寻址过程是硬盘 IO 中最耗时的部分，因为，需要通过硬盘中的马达，带动磁头移动和盘片转动。因此，对于机械式硬盘而言，往往具有很高的顺序 IO，而随机 IO 的性能却很差，因为，随机读写需要不断反复移动磁头进行寻址。

(a)外观 (b)内部 (c)存取原理示意图

图[Hard-disk-drive] 机械式硬盘

机械式硬盘凭借着较高的性价比（即容量大、每字节价格较低），一直以来都是永久性存储领域的首要选择，已经广泛地应用于个人和企业级存储解决方案中。但是，随着闪存技术的不断发展，机械式硬盘的地位开始动摇，基于闪存的固态盘已经有逐渐取代机械式硬盘的趋势。闪存是一种新兴的半导体存储器，从 1989 年诞生第一款闪存产品开始，闪存技术不断获得新的突破，并逐渐在计算机存储产品市场中确立了自己的重要地位。闪存是电可擦除可编程只读存储器（EEPROM：Electrically Erasable Programmable Read-Only Memory）的变种，比 EEPROM 的更新速度更快。闪存是一种非易失性存储器，即使发生断电也不会丢失数据，因此，可以作为永久性存储设备。但是，闪存不能像 RAM 一样以字节为单位改写数据，因此，闪存不能取代 RAM。

1.2 闪存的原理、分类、特性和应用

本节将介绍闪存的工作原理、闪存的分类、闪存的结构、闪存的特性、闪存产品和应用。

1.2.1 闪存的工作原理

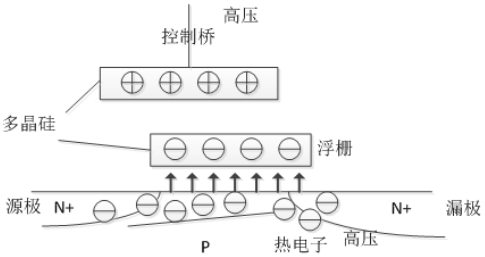
闪存是电可擦除可编程只读存储器（EEPROM）的变种，最初是由日本东芝公司开发出来去代替磁盘的。关于闪存的名字中为什么使用“闪”字，有一种说法是[LeeK07]，一般的 EEPROM 需要逐一字节进行写入和擦除，而闪存可以在一次操作行为中同时写入一个页（包含 512 字节）或者擦除一个块（包含 16 或 32 个页）。

为了更好地理解闪存的工作原理，有必要首先了解可擦除可编程只读存储器（EPROM: Erasable Programmable Read-Only Memory）和电可擦除可编程只读存储器 EEPROM。

EPROM 由以色列工程师 Dov Frohman 发明的，是一种非易失性的半导体存储器，即断

电后仍能保留数据。EPROM 具有可擦除功能，擦除后即可进行再编程，写入数据。因此，在写入数据前，必须用强紫外线照射来擦除里面的内容，采用的方法是：用紫外线照射芯片上面的透明擦除窗口（如图[EPROM]所示）。EPROM 的编程是借助于编程器来完成的。编程器是用于产生 EPROM 编程所需要的高压脉冲信号的装置。在编程时需要将等待写入 EPROM 的数据首先送到随机存储器中，然后启动编程程序，编程器会把数据逐行地写入 EPROM 中。数据被写入到 EPROM 中以后，可以大约保持 10～20 年，并能无限次读取。同时，为了确保数据安全，擦除窗口必须保持覆盖，以防偶然被阳光擦除（阳光中包含紫外线）。

图[EPROM] 一款 EPROM 产品



图[EPROM-theory] EPROM 工作原理

图[EPROM-theory]给出了 EPROM 的工作原理。EPROM 常采用浮栅雪崩注入式 MOS（Metal Oxide Semi-conductor）电路，它与 MOS 电路相似，在 P 型基片上生长出两个高浓度的 N 型区，通过欧姆接触分别引出源极和漏极。在源极和漏极之间有一个多晶硅栅极浮空在绝缘层中，与四周无直接电气联接，称为“浮栅”。这种电路以浮栅是否带电来表示存储 1 或者存储 0。当需要往 EPROM 中写入数据时，在漏极加高压，电子从源极流向漏极，沟道充分开启。在高压的作用下，电子的拉力加强，能量使电子的温度极度上升，变为热电子，在控制栅施加高压时，热电子可以注入到浮栅中，使得浮栅带电。当浮栅带电以后，沟道就会处于关闭状态，在没有别的外力的情况下，电子会很好地保持在浮栅中。在需要清除电子时，可以利用紫外线进行照射，给电子足够的能量，使电子逃逸出浮栅，浮栅就不带电了。

EEPROM 与 EPROM 相似，它是在 EPROM 基本单元电路的浮栅上面再生成一个浮栅，前者称为第一级浮栅，后者称为第二级浮栅。可以给第二级浮栅引出一个电极，使得第二级浮栅接某一电压 VG。若 VG 为正电压，第一级浮栅与漏极之间产生隧道效应，使得电子注入第一级浮栅，即编程写入。若使 VG 为负电压，就会使得第一浮栅的电子散失，即擦除，擦除后可重新写入。

闪存通常包括 NOR 和 NAND 两种类型，其中，NAND 闪存的基本单元电路与 EEPROM 类似，也是由双层浮栅 MOS 管组成。但是，NAND 闪存的第一级浮栅的介质很薄，作为隧道氧化层。NAND 闪存的写入方法与 EEPROM 相同，也是在第二级浮栅加正电压，使电子进入到第一级浮栅。当需要擦除 EEPROM 中的内容时，需要在源极加正电压，利用第一级浮栅与漏极之间的隧道效应，将注入到第一级浮栅的负电荷吸引到源极。由于利用源极加正电压擦除，因此，各单元的源极是连接在一起的，这样，擦除操作不能以字节为单位进行擦除，而是全片或者分块擦除，因此，NAND 闪存的擦除代价较高。

1.2.2 闪存的分类

根据所采用的逻辑门类型的不同，闪存通常包括 NOR 和 NAND 两种类型。NOR 闪存是由 Intel 公司开发的，是一种随机访问设备，具有专用的地址和数据线（和 SRAM 类似），以字节的方式进行读写，允许对存储器当中的任何位置进行访问，这使得 NOR 闪存是传统的只读存储器（ROM）的一种很好的替代方案，比如计算机的 BIOS 芯片。而 NAND 闪存则没有专用的地址线，不能直接寻址，是通过一个间接的、类似 I/O 的接口来发送命令和地址来进行控制的，这就意味着 NAND 闪存只能够以页的方式进行访问。相对于 NOR 闪存而

言，NAND 闪存只需要更少的逻辑门就可以存储相同数量的位，因此，NAND 闪存比 NOR 闪存体积更小，存储密度更大。就读取速度而言，NOR 闪存要比 NAND 闪存稍快一些；就写入速度而言，NAND 闪存要比 NOR 闪存快许多。NAND 闪存执行擦除操作比较简单，只需要擦除整个块即可。NOR 闪存进行擦除时，需要把所有的位都写为 1。NOR 闪存虽然具备更快、更简单的访问过程，但是，存储能力比较低，因此，比较适合用来进行程序的存储。NAND 闪存可以提供极高的单元存储密度（当前单个芯片具备了 32GB 的存储能力），比较适合存储大量的数据，并且写入和擦除的速度也很快；此外，NAND 闪存的读写操作单元通常是一个扇区的大小（即 512KB），这使得 NAND 闪存和磁盘的行为非常类似[InoueW04]。

在闪存发展的初期阶段，NOR 闪存在市场上占据统治地位，但是，后来随着各种手持设备（比如 MP3、手机和数码相机等）对数据存储量需求的快速增加，NAND 闪存的市场占有率开始逐渐超越 NOR 闪存。作为 NOR 闪存的开发者，Intel 公司也和生产 NAND 闪存的 Micron 科技公司合作成立了一个新的公司——IM 闪存科技公司，涉足 NAND 闪存的生产。一些手机生产商也开始抛弃 NOR 闪存转而选择 NAND 闪存，而且，NAND 闪存提供了和磁盘类似的标准访问接口，因此，目前市场上的存储产品很多采用了 NAND 闪存。

表[NAND-NOR] NAND 和 NOR 闪存的比较

	NOR 闪存	NAND 闪存
访问模式	线性随机访问	以页的方式进行访问
存储密度	存储能力比较低	单元存储密度高
擦写次数	10~100 万次	1~10 万次
擦写速度	写入和擦除的速度较慢	写入和擦除的速度很快
主要用途	比较适合用来进行程序的存储	比较适合存储大量的数据

就可靠性而言，可以从位反转和坏块处理这两个方面来比较 NOR 闪存和 NAND 闪存[Xiang09]：

（1）位反转现象：是指某些情况下闪存芯片上的一个位(bit)发生了反转，这种现象在 NOR 闪存和 NAND 闪存上都可能出现，不过出现的概率很低，而且前者发生反转现象的概率要远小于后者。为了增强 NAND 闪存的可靠性，系统往往使用错误探查和更正算法。

（2）坏块问题：主要与 NAND 闪存有关。NAND 闪存在生产过程中，由于生产环节的各种因素，会不可避免地产生坏块，坏块在闪存产品中是随机分布的，这些坏块性能很不稳定，如果在成品中使用，会造成用户数据的丢失。因此，产品在发布使用前，需要对整个闪存空间进行初始化扫描，如果发现坏块，就把它标记为无效不可用，这些坏块会被事先预备的区域来替代，或者可以直接改变映射表，隔离这些坏块。

虽然 NOR 闪存和 NAND 闪存在特性上存在很多差异，但是二者也存在共同点，比如，对于二者而言，写代价和擦除代价都要明显高于读代价，都需要进行“写前擦除”操作，都存在擦除次数的限制等。

NAND 闪存包含两种类型：SLC（Single-Level Cell）和 MLC(Multi-Level Cell)。SLC 的全称是“单层式存储”，即在每个存储单元中存储 1 位；MLC 的全称是“多层式存储”，它采用较高的电压驱动，通过不同级别的电压在一个块中记录两组位信息，这样就可以在理论上将原本 SLC 的存储密度提升一倍。因此，MLC 比 SLC 的存储密度更高。由于 MLC 比 SLC 价格更低，容量更高，因此，它已经被用于很多低端的消费类电子产品中，比如移动电话、MP3 播放器、PDA 和数码相机等。但是，SLC 结构简单，在写入数据时电压变化的区间小，所以寿命较长，传统的 SLC NAND 闪存可以经受 10 万次的读写。由于 SLC 比 MLC 具有更高的速度和寿命（寿命通常是 MLC 的 10 倍[ChenKZ09]），因此，SLC 大多用于对性能和可

靠性要求较高的工业化应用中。

1989 年，日本东芝公司发布了 NAND 闪存的结构，此后，NAND 闪存的发展非常迅速。从 1996 年开始，NAND 闪存芯片的存储密度每年都会翻翻。Chang-gyu Hwang[Hwang03]曾在 2003 年做出预测，到 2012 年，每个 NAND 闪存芯片的容量可以达到 250GB，到 2014 年，每个 NAND 闪存芯片的容量可以达到 1TB。Chang-gyu Hwang 对闪存发展的预测，到 2012 年为止一直都是成立的。

MLC NAND 闪存的市场仍然在不断扩大，主要是因为它的容量迅速扩大，而且性能也不断得到提升。如果没有特殊说明，在本文后面的内容中，闪存专门是指 MLC NAND 闪存。

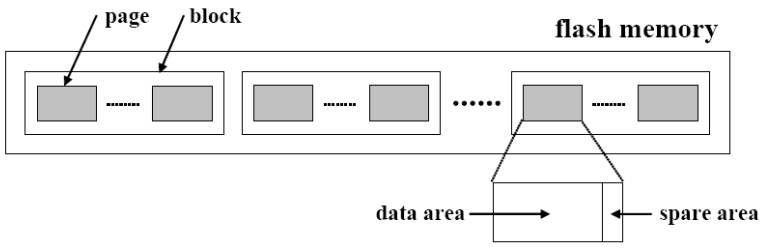
1.2.3 闪存的结构

闪存是一种纯粹的电子设备，不包含任何机械部件，靠电路控制进行存取操作，具备良好的物理特性，体积小，质量轻，能耗低，抗震性能好。闪存的能耗较低，一般假设闪存中一个操作的时间和它的电能消耗成正比。图[flash-chip-sumsung]显示了一款三星闪存芯片产品 SAMSUNG KLMBG8EEGM-8001 的外观。

图[flash-chip-sumsung] 一款三星闪存芯片产品外观

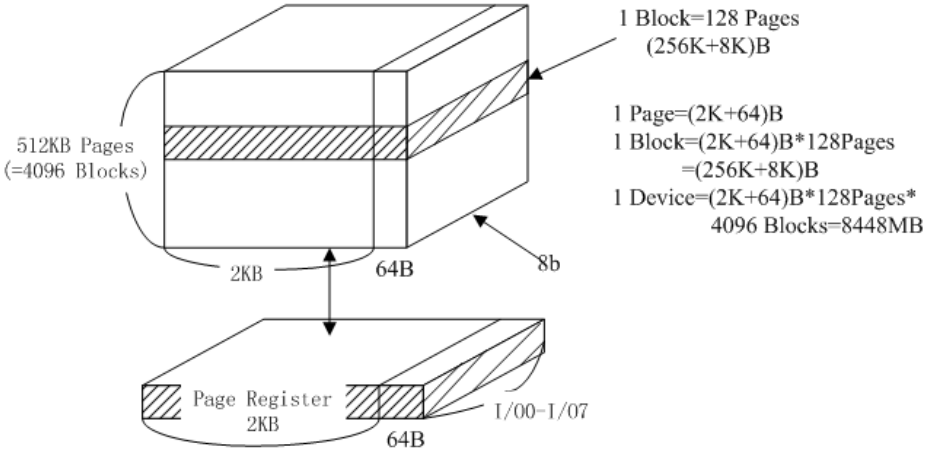
单个闪存芯片可以提供很高的性能，读操作速率可以达到 40MB/秒，写操作的速率可以达到 10MB/秒，并且具有很低的能耗。因此，几十个闪存芯片并行连接，可以提供每秒成百上千个 IO。一个闪存设备是由闪存芯片构成的，这些芯片并行连接到控制器，控制器中通常包含一定数量的缓存，比如 16MB 到 32MB。

如图[flash-structure]所示，闪存芯片是由一组数据存储单元阵列组成的，包含许多个“块”，每个块又包含许多个“页”（通常是 32 个页），一个页通常是 512 字节，因为闪存被开发出来时的最初目的就是为了取代磁盘，因此，一个闪存页的大小和磁盘扇区大小保持了一致。一个页不仅包括数据区域（通常是 512 字节），还包含了一个额外的、小的备用区域（通常是 16 字节），或者称为“带外数据（OOB：Out of Band）”区域，它是用来存储一系列的管理信息，包括：（1）错误纠正码；（2）和存储在数据区域中的数据对应的逻辑页面编号；（3）页面状态[KimKNMC02]。在写数据的同时，就可以顺便把管理信息写入这些备用区域，额外开销有时候可以忽略不计。每个闪存页的状态可以是以下三种状态中的一种：（1）有效；（2）无效；（3）自由/擦除。当没有数据被写入一个页时，这个页就处于“擦除”状态，这时，页中的所有位都是 1。一个写操作只能针对处于擦除状态的页，然后把这个页的状态改变为“有效”。异地更新会导致一些页面不再有效，它们被称为“无效页”。



图[flash-structure] 闪存的结构

闪存的一个块通常包含 32 个页，因此，一个块的大小通常是 16KB，一般称这种闪存为“小块 NAND 闪存”。但是，一些高端应用需要更快的写和擦除速度，因此，闪存生产商开始生产“大块 NAND 闪存”，这类闪存中，一个块包含了 64 个页，每个页 2212 字节，因此，一个块的大小就是 128KB。



图[Flash-chip] K9G8G08U0M 芯片的阵列组织结构

图[Flash-chip]给出了一款来自三星公司的闪存芯片产品（K9G8G08UOM 芯片）的阵列组织结构图[Samsung2006]，从图中可以看出，这款闪存芯片一共包含 4096 个块，每个块包含 128 个页，每个页的数据区域大小是 2KB，备用区域的大小是 64B，因此，可以计算得到这款闪存芯片的容量是 8448MB。

1.2.4 闪存的特性

闪存在最初始阶段，所有位都被设置成 1，闪存的三种典型操作是读操作、写操作和擦除操作，三者对位的操作具体如下：

- 读操作：返回被读取目标页的所有位；
- 写操作：把目标页中选中的一些位从 1 变成 0；
- 擦除操作：把目标块的所有位都设置为 1。

闪存中的每个操作的能耗分别大约是:24uJ(读操作)、763uJ(写操作)和 425uJ(擦除操作)[ZhengGSZJKW03]。

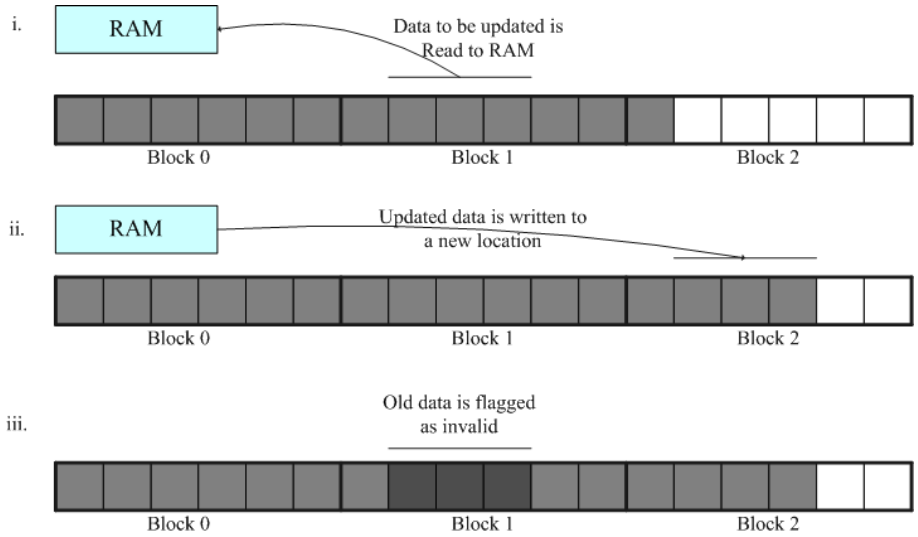
从技术角度而言，NAND 闪存具有不同于磁盘的几个显著特点：

（1）**没有机械延迟**：和磁盘不同，闪存中不存在任何机械部件，就是一个纯粹的电子设备，可以通过电路实现直接寻址，读取数据，不需要像磁盘那样进行耗时的寻址。对于闪存而言，不管是顺序读取，还是随机读取，性能都没有区别，都可以获得较快的速度。因此，在闪存中，访问数据的时间开销几乎和要访问的数据量成正比，而与数据在闪存中的物理位置无关。

（2）**写操作最小单元是页**：一个写操作的最小单元是一个页，只对一个页的一部分进行写操作是不可能的。对于不同的产品，一个页的大小可能是 2KB 或者 4KB[LeeKWCK09]。页面在块内部会被顺序写入，即当一个块中的第 i 页被写入后，块中的第 j 页（$1<j<i$）就不能被写入，直到这个块被擦除。

（3）**写前擦除**：闪存和磁盘在更新时采用两种不同的方式，前者采用“就地更新”，后者采用“异地更新”，两种方式的主要区别在于逻辑页是否总是被写入到同一个物理地址。磁盘采用“就地更新”的方式，在更新一个数据项时，首先找到这个数据项的存储位置，然后，就在原地执行更新操作，可以直接覆盖原来的数据。但是，闪存的更新方式则不同，并不支

持对原来数据的直接覆盖，为了对存储在闪存中的现有数据项进行更新，必须在写入数据之前执行一个耗时的擦除操作，然后在这个擦除过的页面上写入新数据，擦除操作的平均延迟是 1500 微秒[AgrawalPWDMP08]，比写操作慢得多。闪存的这个特性，严重制约了写操作的性能。为了避免每次更新操作都带来代价高昂的擦除操作，闪存一般采用“异地更新”的方式（如图[out-of-place-update]所示），即在更新数据时，把更新操作引导到其他空闲页执行，原来的旧数据所在的页可以暂时不擦除，只要简单设置为“无效”即可，只需要等到垃圾回收的时候才统一执行擦除操作。



图[out-of-place-update] 异地更新

为了提高闪存性能，还应该通过各种其他方式尽量减少由写操作引起的擦除操作的数量，这个问题通常是由 FTL 机制来解决的（见本书第 3 章关于 FTL 的介绍）。对于封装了闪存芯片的固态盘而言，FTL 作为核心组件会被固化到 ROM 中，而对于直接使用闪存芯片的掌上设备（如手机）而言，则是由操作系统（比如 Windows Mobile）中的软件来实现 FTL 功能。一旦一个闪存设备中已经写入数据的页需要被更新，并且没有可用的擦除页，FTL 必须执行下列操作：（1）确定一个要被擦除的块；（2）从这个块中读取所有有效的页到内存；（3）擦除这个块；（4）把有效页写回这个块；（5）把更新的页写入这个块。擦除操作是很慢的，同时，转移有效页也会增加额外的开销。因此，一个擦除操作的总代价取决于两个方面的因素：（1）擦除操作的代价被分摊到多少个写操作上面和（2）在每次擦除操作中，有多少个有效页被转移。比较理想的情况是，不需要拷贝任何有效页，在更新一个完整的擦除块时只需要进行一次擦除操作。

（4）**擦除操作粒度比写操作粒度大很多**：擦除操作的粒度通常要比写操作大许多，因为，写操作的最小单元是页，而擦除操作并不是有选择性地针对某个数据项或者页，而是必须擦除包含该数据项的整个块，这个块被称为“擦除块”，擦除块的尺寸要比页大得多，通常一个擦除块包含 64 个页或者 128 个页。此外，对于一个块而言，擦除次数是有限的。如果一个块被擦除的次数超过一定数量，这个块就会老化掉，变得不稳定。

（5）**读写速度不对称**：在闪存中，读取数据时，只需要获得闪存中某个存储单元的状态即可；相反，在写入数据时，则需要往相应的存储单元中填充电子直到达到一个稳定的状态。由此可知，闪存的读取数据和写入数据的速度是不相同的，前者要比后者快一个数量级。一般来说，读操作和写操作的访问延迟分别为 25 微秒和 200 微秒[AgrawalPWDMP08]。

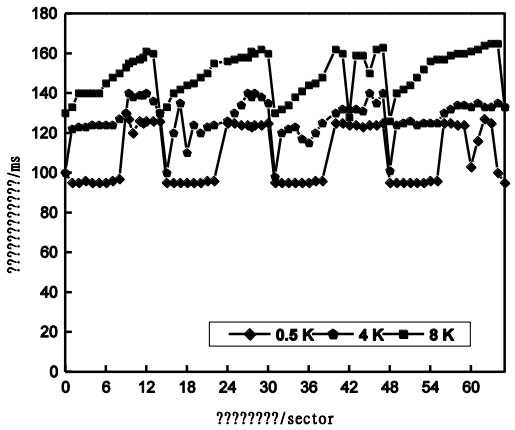
表[flash-HDD-DRAM]中给出了不同类型的存储介质的每种操作的时间开销，可以看出，

DRAM 和磁盘都具有读写操作对称性，即读操作和写操作的时间开销都相同，而闪存则表现出了读写操作的不对称性，写操作的时间开销要比读操作高一个数量级。而且闪存中存在耗时的擦除操作，擦除操作的时间开销比读操作开销高两个数量级，而 DRAM 和磁盘中则不存在擦除操作。

表[flash-HDD-DRAM] 不同类型的存储介质的每种操作的时间开销

	读操作	写操作	擦除操作
闪存	25 微秒/512 字节	200 微秒/512 字节	2 毫秒/2KB
DRAM	100 纳秒/字节	100 纳秒/字节	无
磁盘	12.4 毫秒/512 字节	12.4 毫秒/512 字节	无

（6）**存在快慢块的现象**：即不是所有块的读写速度都一样。图[SSD-3-test-result]给出了一款由 OCZ 公司生产的固态盘产品在不同块大小时的测试结果[FanLM12]，该款固态盘容量为 60GB，采用了 MLC NAND 闪存芯片，接口类型为 SATA2。图[SSD-3-test-result]中曲线显示了该固态盘产品的请求响应延迟随着请求偏移位置的变化而变化的情况。从图中可以看出，随着请求偏移大小的变化，请求响应延迟都会随着请求位置的增加而呈现出周期性的变化，并且在一个变化周期内，出现快慢两个阶段，这就说明这款固态盘产品具有快慢块的现象，即不是所有块的读写速度都一样。



图[SSD-3-test-result] 某款固态盘在不同块大小时的测试结果

1.2.5 闪存产品和应用

闪存由于具备体积小、质量轻、能耗低、抗震性能好等优良特性，已经被广泛应用于各类消费电子设备中作为存储介质，包括数码相机、手机、PDA、MP3 等小型数码产品。在这些设备中，闪存被制作成一张小的“卡片”，直接插入到设备插槽里，所以被称为“闪存卡”。根据不同的生产厂商和不同的应用，闪存卡的类型也不大相同（如图[flash-device]所示），主要包括 SmartMedia（SM 卡）、Compact Flash（CF 卡）、MultiMediaCard（MMC 卡）、microSD（TF 卡）、Secure Digital（SD 卡）、Memory Stick（记忆棒）、XD-Picture Card（XD 卡）和微硬盘（MICRODRIVE）。虽然这些闪存卡在外观和规格上存在着很大的区别，但是底层的技术原理都是相同的。此外，闪存芯片还别封装制作成 U 盘和固态盘，前者主要用于个人新的存储，后者则越来越多地应用于企业级别的数据存储。闪存由于具有很好的抗震性能，因此，它在一些对抗震和防热要求较高的领域也得到了广泛的应用，比如应用在坦克和飞机中携带的嵌入式设备中。

目前市场上生产闪存产品的主要厂商包括金士顿、索尼、晟碟、Kingmax、鹰泰、创见、爱国者、纽曼、威刚、联想、台电等。

图[flash-device] 采用闪存的电子设备

需要指出的是，不同类型的磁盘产品的性能差别不大，因为，它们内部都是包含类似的磁头和碟片，寻址时间和旋转速度差异不大。但是，对于基于闪存的存储产品而言，不同产品的性能差异却是比较大的，如表[HDD-flash-performance]所示。廉价的低端闪存存储产品，比如 USB 存储棒或者相机存储卡，只能提供适中的读带宽，但是写性能很差。具有 SATA 接口的固态盘产品则能够提供更好的性能，比最好的磁盘速度还要快 3 倍以上，这主要归功于固态盘的内部设计，比如智能的块映射机制、对多个闪存芯片的并行 IO 访问和写缓冲等等。一些高端的固态盘产品的速度更快，采用了 PCI-e 连接接口，并且具有专用的设备驱动，而不是采用现有的 SATA 驱动。不同闪存产品的性能差异主要来自两个方面：首先，固态盘可以充分利用内部并行特性提高吞吐量；其次，固态盘可以利用智能的 FTL 机制来减少闪存“写前擦除”的代价。

表[HDD-flash-performance] 磁盘和不同闪存产品的性能比较

设备	顺序(MB/s)		随机 4K-IO/s	
	读	写	读	写
磁盘	80	70	120-300/s	
USB 闪存	11.7	4.3	150/s	20/s
固态盘	250	170	35K/s	3.3K/s
PCI-e 闪存	700	600	102K/s	101K/s

1.3 基于闪存的固态盘

闪存芯片可以被封装成固态盘产品，提供标准的访问接口和大容量的存储能力。本节首先介绍固态盘的特性、产品和应用，然后，介绍固态盘的内部结构，揭示固态盘内部并行特性，并给出探测固态盘内部特性的方法，最后，介绍了固态盘的性能，并给出了一些固态盘 IO 性能实验测试分析结果。

1.3.1 固态盘特性

固态盘是硬盘的一种，硬盘包括机械式硬盘（HDD：Hard Disk Drive）和固态硬盘（SSD：Solid State Disk，简称固态盘）。机械式硬盘属于磁盘设备，而固态盘则是半导体存储器。

固态盘是指用固态电子存储芯片阵列而制成的硬盘，由控制单元和存储单元组成。固态盘的接口规范和定义、功能及使用方法上与机械式硬盘完全相同，在产品外形和尺寸上也完全与机械式硬盘一致。固态盘已经广泛地应用于军事、车载、工控、视频监控、网络监控、网络终端、电力、医疗、航空等、导航设备等领域。虽然在当前阶段固态盘的成本较高，但是，它已经表现出广阔的应用前景，已经逐渐开始在各个应用领域取代传统的机械式硬盘。

固态存储技术在过去这些年不断取得新的进步，比如 NAND 闪存、磁性 RAM、相变内存（PRAM）和铁电存储器（FRAM）等。其中，基于 NAND 闪存的固态盘（下文简称“固

态盘”），更是得到了广泛的应用。由于采用闪存技术，固态盘继承了闪存的诸多优点，拥有一些传统机械式硬盘（或称为普通硬盘）无法比拟的优势：

（1） 读写速度快。机械式硬盘包含了机械部件，需要较长的寻址时间。固态盘采用了闪存技术，不包含机械部件，寻址开销几乎是 0，可以忽略不计。固态盘的存取速度也要比机械式硬盘快许多。

（2） 能耗低、噪音小、体积小和抗震。固态盘是靠电路控制的设备，不包含任何机械部件和风扇，因此，体积比机械式硬盘更小，在使用过程中不会产生噪音，发热量和能耗也比机械式硬盘低，同时不怕外力冲击，具有较好的抗震性能。固态盘比机械式硬盘具有更大的工作温度范围，后者的正常工作温度范围在 5~55 摄氏度之间，而前者则可以在-10~70 摄氏度之间正常工作。

（3） 竞争力不断增强。虽然从每字节价格而言，在目前阶段，固态盘要比机械式硬盘贵，但是，从每秒钟每个随机 I/O 的价格而言，固态盘要明显比机械式硬盘便宜得多[CanimMBRL10]。而且，固态盘的价格每年都在不断地迅速下降，它相对于机械式硬盘的竞争力会不断增强。

1.3.2 固态盘产品

固态盘的发展已经走过了二十几年的历程。1989 年，世界上第一块固态盘产品出现。2006 年 3 月三星公司发布了一款采用 32GB 容量的固态盘的笔记本电脑。2008 年 9 月，以忆正 MemoRight SSD 的正式发布为标志，中国企业开始加速进军固态盘行业。2009 年，固态盘行业进入高速发展时期，各大厂商纷纷加入固态盘研发和制造的队伍。2011 年，固态盘的容量完成了从 32GB 到 256GB 的跨越，读取速度高达 500MB/s。2012 年，苹果公司在笔记本电脑上应用容量为 512GB 的固态盘[BaiduBaikeSSD]。

由于固态盘技术与机械式硬盘技术不同，所以产生了不少新兴的存储器厂商。厂商只需购买 NAND 存储器，再配合适当的控制芯片，就可以制造固态盘了。目前市场上比较常见的固态盘主要使用 Indilinx、SandForce、JMicron、Marvell、Samsung 和 Intel 等各种主控芯片。主控芯片是固态盘的核心部件，负责合理调配数据在各个闪存芯片上的存取，并且承担了整个数据中转任务，连接闪存芯片和外部接口（比如 SATA 接口）。不同的主控芯片之间的能力差异很大，在数据处理能力、算法、对闪存芯片的读取写入控制上，都存在非常大的区别，由此导致固态盘产品在性能上的差距可能会高达数十倍。表[SSD-control-chip]列出了一些主要的固态盘主控芯片品牌、型号和产品[BaiduBaikeSSD]。

表[SSD-control-chip] 固态盘主控芯片品牌、型号、产品一览表

品牌	型号	代表产品
Intel	PC29AS21AA0、PC29AS21BA0	Intel 320 Series G3(80G)
SandForce	SF-1500/SF-1200、SF-2000 系列	OCZ Agility 3 SATA3 60G
JMicron	JMF602、JMF612、JMF618	金士顿的 SSD Now V 系列
Marvell	88SS9174-BJP2、88SS9174-BKK2	Intel 的 510 系列、镁光 C400、浦科特 PX-128M2S
Indilinx	IDX110M00-LC、IDX110M01-LC	SOLIDATA K5-64Me
三星	S3C49RBX01-YH80、S3C29RBB01-YK40	三星 SLC 3.5 100GB
东芝	TC58NCF602GAT、	金士顿 SSDNow V+100 系列

	TC58NCF618GBT、T6UG1XBG	

固态盘具有和机械式硬盘一样的 I/O 接口，比如 SATA、PATA 或 PCMCIA 接口，新一代的固态盘普遍采用 SATA-2 接口及 SATA-3 接口。例如，Samsung Standard Type MCAQE32G8APP-0XA 就设计了 1.8 英寸的 PATA 接口，具备 32GB 的存储容量，该产品内部配置了 Samsung K9WAG08U1A 16 Gbits SLC NAND 闪存芯片（如图[SSD-PATA]所示）[LeeMPKK08]。

图[SSD-PATA] 采用 PATA 接口的固态盘产品

图[SSD-SATA]给出了一款 KINGDISK(金典)公司的固态盘产品 KDSF12MXXX，外形尺寸为 2.5 寸，采用了 SATA-2 接口和 MLC NAND 闪存，存储容量包括 60GB\120GB\240GB，缓存容量为 64MB，4KB 随机读 IOPS 为 30000，4KB 随机写 IOPS 为 10000，平均存取时间为 0.1ms，连续读速度和连续写速度分别可以达到 283MB/s 和 271MB/s，采用了动态和静态的平衡擦除算法，读寿命为无限期，对于 60GB 容量的产品，每天写 100GB 资料，可以使用 16 年，工作温度范围在 0-70 摄氏度，存放温度为-40~85 摄氏度，在 25 摄氏度下可以正常保存数据 10 年。该产品可以应用在所有笔记本及 SATA 接口的设备中。

图[SSD-SATA] 采用 SATA 接口的固态盘产品

1.3.3 固态盘应用

由于采用相同的 I/O 接口标准，因此，对于上层操作系统而言，可以用完全相同的方法访问机械式硬盘和固态盘。因此，现有的基于磁盘的各种应用，不做任何修改就可以直接转移到固态盘上面。

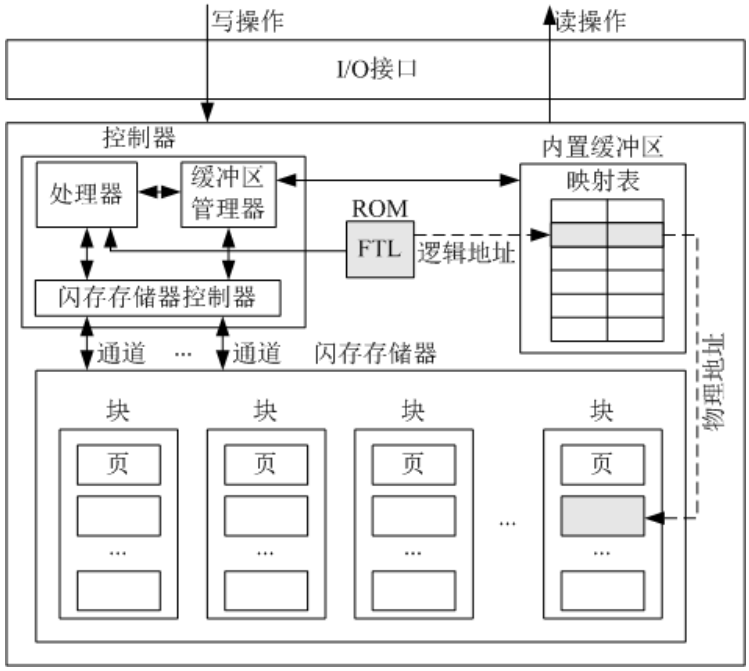
固态盘可以帮助企业构建大型存储系统，提高数据访问速度。市场上已经有一些支持大型企业应用的固态盘产品，比如 Texas Memory System 的 RamSan-500 和 EMC 的 Symmetrix DMX-4。三星和戴尔公司已经发布了完全采用固态盘的笔记本电脑[Samsung06]。目前，已经有一些知名企业开始在固态盘上部署各种企业应用。比如，苹果公司曾在部分音乐播放器上用闪存代替磁盘，Myspace.com 已经在数据中心采用固态盘作为主要存储介质[Myspace]，Facebook 也发布了 Flashcache，可以把数据缓存在固态盘中加速读写操作[Facebook]。全球领先的在线搜索引擎服务提供商——谷歌，宣布计划把现有的基于磁盘的存储系统，转移到

基于固态盘的平台[Claburn08]。2008 年 8 月，中国领先的在线搜索引擎服务提供商——百度公司，宣布以闪存及其配套技术全面代替磁盘存储，该公司承载全球检索及索引存储的运算集群中的硬盘已经全部拆除，百度也成为全球首个使用闪存技术代替磁盘并大规模商用的互联网公司。百度认为，全面采用闪存及其配套技术，可以大幅提高其服务能力和检索速度。根据 IDC 的预计，将有超过 300 万的固态盘被部署到企业中，仅在 2011 年，固态盘的市场规模就已经达到 12 亿美元[DatacenterSSD]。国际数据集团 IDC 公司曾在 2009 年作出分析，随着基于 Web 和云计算公司的需求的不断增加，直到 2013 年为止，企业级闪存设备都会保持每年 165%的增长，并且有越来越多的数据库应用会运行在闪存设备上[Lawson09]。

1.3.4 固态盘结构

图[SSD-structure]给出了固态盘的内部结构，通常包括 I/O 接口、控制器、FTL、内置缓冲区和闪存存储器。

- I/O 接口：负责接收来自外部的读写请求，并返回结果。I/O 接口一般采用和机械式硬盘类似的标准接口，比如 SATA、PATA 或 PCMCIA 接口，因此，固态盘可以直接被集成到那些原来采用机械式硬盘的系统中。
- 控制器：负责管理闪存空间，完成数据读写请求。控制器中包含三个主要组件，即处理器、缓冲区管理器和闪存存储器控制器。处理器负责从 ROM 中读取加载 FTL，实现 FTL 各种功能；缓冲区管理器负责管理内置缓冲区；闪存存储器控制器负责闪存存储器的连接、控制、读写命令传输、地址传输和数据传输等等[FanLM12]。闪存存储器和闪存存储器控制器之间使用通道进行连接，由通道来实现命令、地址和数据的传输。不同通道之间可以实现并行操作。
- FTL：FTL 被固化到 ROM 中，ROM 是非易失性存储，即使断电也不丢失信息，因此 FTL 可以一直保存在 ROM 中，当系统启动时，控制器就从 ROM 中读取加载 FTL。FTL 是最核心的组件，隐藏了闪存的特性，可以让线性的闪存设备看起来像一个虚拟磁盘。FTL 的功能包括提供逻辑地址到物理地址的映射、断电恢复、垃圾回收和磨损均衡等。
- 内置缓冲区：用来存储地址映射表等信息，加快地址转换过程。有些固态盘采用专用的 DRAM 作为内置缓冲区来保存元数据或数据，而有些固态盘则采用成本相对低一些的、较小的 SRAM， SRAM 读写速度比闪存芯片快许多，但是，相对于闪存芯片而言，SRAM 价格仍然较高，因此，通常一个固态盘配置的 SRAM 容量为闪存空间的 3%到 5%左右。此外，SRAM 是易失性存储，一旦断电就会丢失信息。
- NAND 闪存芯片：是最终用来存储数据的物理介质，包含许多个块，一个块中又包含许多个页。固态盘在接收到读写命令、逻辑地址和数据大小等信息以后，FTL 会把读写命令转换成一系列的闪存内部命令（读、写、擦除），并通过查找保存在 SRAM 中的映射表来实现逻辑地址到物理地址的转换。这个 SRAM 中的映射表，最初是通过对闪存可用的空间进行扫描后构建起来的，此后，更新操作会不断修改映射表条目，记录最新版本数据的物理存储位置。



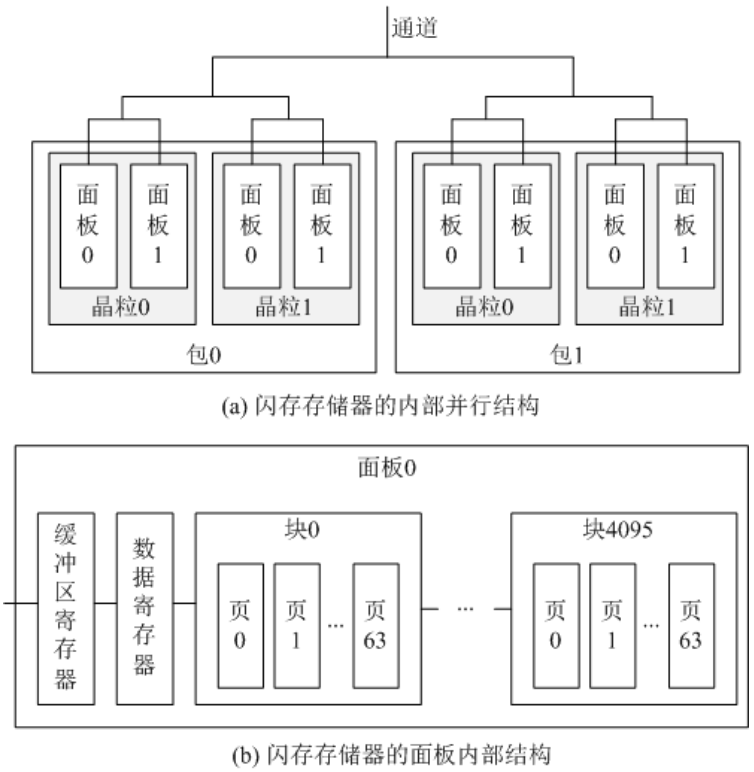
图[SSD-structure] 基于 NAND 闪存的固态盘内部结构示意图

1.3.5 固态盘内部并行特性

在固态盘中，闪存存储器采用多级并行结构，每个级别的操作都可以并行执行或者交叉执行，从而可以拥有很高的数据存取效率。一般而言，固态盘可以包括以下几个级别的并行[ParkSSML10][ChenLZ11]：

- **通道级别并行**：在一个固态盘中，闪存存储器包（flash memory package）会通过多个通道连接到控制器上，每个通道都可以独立并行操作。有些固态盘会采用多个 ECC（Error Correction Code）引擎和闪存控制器，给每个通道都分配一个，从而可以获得更好的性能。
- **包级别的并行**：为了优化资源利用率，一个通道会被多个闪存存储器包所共享，每个闪存存储器包可以独立操作。附加在同一个通道上的闪存存储器包，也可以交叉工作，因此，总线的利用率可以达到最优化。
- **晶粒级别的并行**：一个闪存存储器包通常包含两个或者多个晶粒（die），每个晶粒可以被单独选择，并且独立于其他晶粒而执行自己的命令，从而大大增加了吞吐量。
- **面板级别的并行**：一个闪存存储器晶粒通常包含两个或者多个面板。大多数闪存存储器支持在多个面板上并行执行相同的操作，包括读、写和擦除操作。

在固态盘中，第 1 个级别的并行是多个通道之间的并行操作，即多个通道之间可以同时操作，如果通道数为 *n*，我们就说第 1 级并行的并行度为 *n*。图[SSD- structure-parallelism]给出了闪存存储器在通道上的组织方式的一个实例，其中，一个通道可以存取两个包，每个包中包含两个晶粒，每个晶粒中包含两个面板，每个面板中包含缓冲区寄存器、数据寄存器和 4096 个块，每个块包含了 64 个页，块是擦除操作的基本单元。从图中可以看出，这种组织方式实际上是一个 4 级并行结构，按照并行粒度从大到小的顺序分别是通道级并行、包级并行、晶粒级并行和面板级并行。在这种 4 级并行结构中，面板是最小粒度的并行操作单元，多个并行访问的面板可以构成一个晶粒，多个并行访问的晶粒可以构成一个包，多个并行访问的包可以连接到一个通道上。



图[SSD- structure-parallelism] 固态盘的内部并行结构

固态盘的内部并行特性，可以用来优化闪存数据库的各个模块，提高数据库整体性能，比如缓冲区管理、查询处理与优化、事务处理和索引等。比如，范玉雷等人[FanLM12]充分利用固态盘内部并行特性，对传统数据库表扫描操作进行了相应的改进，提出了一种并行表扫描模型 ParaSSDScan，并在此基础上设计了一种高效的并行聚集操作模型 ParaSSDAggr，通过此模型来实现几种常见的聚集操作；实验结果表明，这种并行表扫描和并行聚集操作相对于传统的数据库表扫描和聚集操作而言，性能分别提升了 3 倍和 4 倍，这充分说明了固态盘内部并行特性相对于传统机械式硬盘的优越性。

1.3.6 固态盘内部特性的探测

了解固态盘内部结构特性，比如所采用的映射策略、块大小等，对于设计高效的数据存取策略，提高固态盘 IO 性能具有重要的意义。比如，如果固态盘采用了基于写操作顺序的映射策略——根据写操作到达的先后顺序把写操作分发到相应的闪存物理位置，那么，一些由于较差的映射而形成的物理数据布局，就会严重降低固态盘 IO 性能，因此，必须尽量避免形成这种较差的物理数据布局；再比如，如果能够知道固态盘中通道的数量，我们就可以设置一个正确的并发级别，避免过度并行化。

但是，就目前而言，获取固态盘内部结构信息又是一件非常棘手的事情。因为，固态盘内部架构细节，通常是生产商的核心技术，往往不会对外公开。尽管固态盘生产商通常会提供一些标准参数，比如峰值带宽，但是，许多重要的信息都是缺失的，比如，包括映射策略等一些关键信息，都无法在生厂商提供的产品说明文档中找到，尽管这些信息对于充分了解和利用固态盘的性能而言是至关重要的。

针对这个问题，一些研究通过设计特定的实验来探测发现固态盘内部的一些技术细节，比如，文献[ChenLZ11]在固态盘生产商已经公开的文档的基础上，定义了一个通用模型来抽象固态盘的内部结构。

1.3.6.1 探测模型

在文献[ChenLZ11]提出的通用模型中，一个域（domain）是一个共享了一组特定资源（比如通道）的闪存存储器的集合。一个域可以被进一步分区成多个子域，比如包。一个块（chunk）是一个被连续分配到一个域内的数据单元。多个不同的块可以通过映射策略，被交织放置在 D 个域中。比如，假设固态盘有 2 个域 Dom_1 和 Dom_2，现在要把 4 个块 b_1,b_2,b_3,b_4 写入到这 2 个域中，则采用交织方式进行存储的过程是，把 b_1 放入 Dom_1，把 b_2 放入 Dom_2，把 b_3 放入 Dom_1，把 b_4 放入 Dom_2。

我们可以把固态盘看成一个“黑盒子”，并且假设固态盘中的映射遵循了一些可重复但是对于我们而言是未知的模式。通过向固态盘中注入精心设计的 IO 模式，就可以观察到固态盘的反应，然后记录一些关键指标，比如延迟和带宽。基于这些探测信息，我们就可以推测固态盘的内部结构和所采用的映射策略。

目前，文献[ChenLZ11]提出的通用模型可以通过实验方法探测到固态盘的以下三个关键技术参数：

- 块大小：一个被连续分配到一个域内的最大数据单元的大小。
- 交织度：在同一个层次的域的数量。交织度通常是由资源的冗余度来决定的，比如通道的冗余度。
- 映射策略：该策略决定了逻辑数据块被映射到哪个域，决定了物理数据布局。

在开展探测实验时需要注意以下事项[FanLM12]：

（１）主板ＢＩＯＳ需开启ＡＨＣＩ（Ｓｅｒｉａｌ ＡＴＡ Ａｄｖａｎｃｅｄ Ｈｏｓｔ ＣｏｎｔｒｏｌｌｅｒＩｎｔｅｒｆａｃｅ，串行ＡＴＡ高级主控接口／高级主机控制器接口）模式，以支持固态盘接口逻辑中最新融入的ＮＣＱ技术，该技术对挖掘利用固态盘内部并行特性至关重要；

（２）固态盘文件访问需设置为ＤｉｒｅｃｔＩＯ模式，以避免操作系统文件缓冲区的影响。

NCQ（Native Command Queuing）是 SATA II 标准引入的一个新特性。有了 NCQ 的支持，设备可以接收多个到达的命令，然后在内部对作业进行排序。NCQ 对于固态盘而言，是尤其重要的，因为，当固态盘可以接收多个并行 IO 作业时，有了 NCQ 的支持，就可以充分利用固态盘较高的内部并行结构，获得很高的吞吐量。早期的固态盘产品不能支持 NCQ，因此，无法从并行 IO 中获益。

1.3.6.2 探测方法和结果

1.3.6.2.1 块大小的探测

作为一个基本的映射单元，一个块只能被映射到一个域中，而两个连续的块则可以被映射到两个不同的域中。假设块的尺寸是 S，对于任何读操作请求而言，可能包括以下两种情形：

（1）第一种情形：如果读操作的请求起始位置的偏移量是 0，并且请求数据量大小不超过 S，那么，这个读操作只需要涉及一个域；

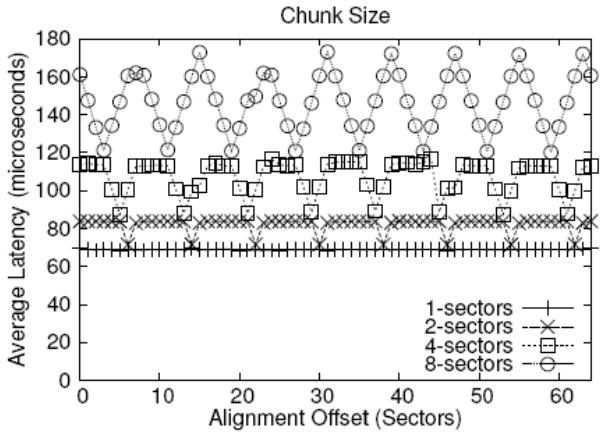
（2）第二种情形：如果读操作的请求起始位置的偏移量是 $S/2$，那么，这个读操作就需要跨越两个域。

很显然，在第二种情形下的读取速度要比第一种情形更快，因为，第二种情形跨越两个域，可以充分利用固态盘内部并行性，在两个域上并行执行读操作。基于这个特性，可以设计一个实验来确定块的大小。算法[SSD-chunk-size-probe]给出了探测块大小的伪代码。首先，需要对固态盘进行初始化，即采用顺序写操作写满整个固态盘。然后，设计不同 IO 请求大

小的情形，即让 IO 请求大小从 1 个扇区变化到 *M* 个扇区，其中，*M* 是预估的最大可能的块大小；接着，针对每种 IO 请求大小，分别测试在不同的 IO 请求偏移位置开始读取指定大小的数据量所需要的平均时间延迟，并绘制平均延迟曲线。随着 IO 请求起始位置偏移量从 0 开始逐渐增加，实验过程会不断分别经历上述的第一种和第二种情形，而每次经历第二种情形时，平均时间延迟都会降低，因此，平均延迟曲线会出现周期性的波动，而两个连续的波谷底部之间的距离就是块的大小。

算法[SSD-chunk-size-probe]：探测固态盘的块大小
init_SSD(): 顺序写满固态盘
rand_pos(*A*): 获取一个和第 *A* 个扇区对齐的随机偏移量
read(*P*, *S*): 从偏移量 *P* 位置开始读取 *S* 个扇区；
plot(*X*,*Y*,*C*): 为曲线 *C* 在(*X*,*Y*)位置绘制出一个点；
M: 预估的最大可能的块大小；
//开始探测块大小
init_SSD(); //初始化固态盘空间
for (*n* = 1 sector; *n* <= *M*; *n* *= 2) //IO 请求的大小
  for (*k* = 0 sector; *k* <= 2**M*; *k* ++)// IO 请求的偏移量
  {
    for (*i* = 0, *latency*=0; *i* < 100000; *i* ++)
    {
      pos = *rand_pos*(*M*) + *k*;//读取数据操作的起始位置
      latency += *read* (*pos*, *n*);//读取数据并计算延迟，即从 *pos* 位置开始读取 *n* 个扇区的数据
    }
    plot (*k*, *latency*/100000, *n*); //绘制平均延迟曲线
  }

图[SSD-chunk-size-experiment]显示了在某款固态盘上的测试结果，其中，每条曲线都代表了一个特定的 IO 请求大小，对于每个 IO 请求所对应的曲线，请求的起始位置偏移量在 0-64 个扇区之间进行变化。从图中可以看出，除了 IO 请求大小是 1 个扇区的情形以外，在所有其他情形下，随着请求起始位置偏移量的增加，曲线会出现周期性的波动。两个连续的波谷底部之间的距离就是块的大小。在这种情形下，探测到的块大小就是 8 个扇区（4KB）。



图[SSD-chunk-size-experiment] 探测块大小时的平均延迟曲线

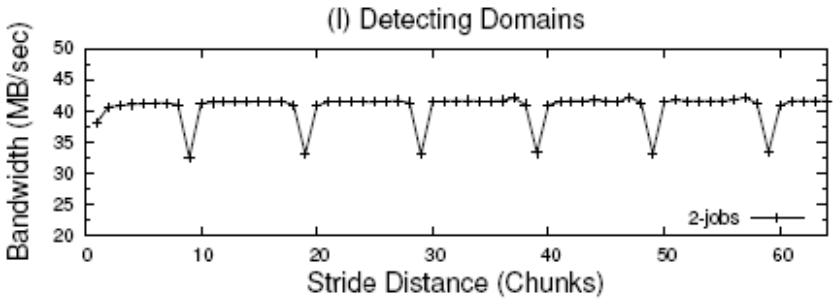
1.3.6.2.2 交织度的探测

当把多个不同块分配到多个域中时，对不存在资源共享机制的多个域中的数据执行并行访问，要比在一个域内部执行上述操作具有更高的带宽。基于这个特性，就可以设计实验来确定交织度，即处于同一层次上的域的数量。假设域的数量是 *D*，数据被交织分布在这些域中，并且假设同时发起 2 个并行线程进行数据读取操作，每次读取 *n* 个块（*n* 表示 IO 请求的大小，以块为单位），其中，对于第 1 个线程 s_1 而言，从偏移量为 0 的位置开始读取 *n* 个块，另一个线程 s_2 读操作跳跃 *d* 个块的偏移量后再读取 *n* 个块，这里的 *d* 被称为“跳读距离”。当 s_2 读操作的跳读距离 *d* 的值增加到一定程度时，线程 s_2 的读操作会跳跃经过若干个域，落入到和线程 s_1 所处的相同的域中。由于两个线程并行执行，并行的数据访问就会竞争同一个资源，带宽就会降低。而采用其他跳读距离时，两个并行的线程就会分布到两个域中执行，因此，就会带来更高的带宽。由此，我们可以设计实验，让跳读距离在某个区间内变化，观测带宽曲线，曲线就会表现出周期性的波动，两个连续的波谷底部的距离就是交织度的大小。算法[SSD-interleaving-degree-probe]给出了探测固态盘交织度的伪代码。

算法[SSD-interleaving-degree-probe]：探测固态盘的交织度

init_SSD(): 顺序写满固态盘；
D：预估的最大可能的交织度；
stride_read(*d*): 2 个并发作业执行跳读操作，跳读距离为 *d*
//开始探测交织度
init_SSD(); //初始化固态盘空间
for (*d* = 1 chunk; d < 4**D*; *d* ++)://跳读距离
{
bw = *stride_read* (*d*);//跳读数据并计算 IO 带宽
plot (*d*, *bw*, *j*); //绘制带宽曲线
}

图[SSD-interleaving-degree]显示了在某款固态盘上的交织度测试结果，可以看出，随着跳读距离的变化，带宽曲线呈现周期性波动，周期是 10，因此，可以得出这个固态盘的交织度是 10，即拥有 10 个域。



图[SSD-interleaving-degree] 固态盘交织度测试结果

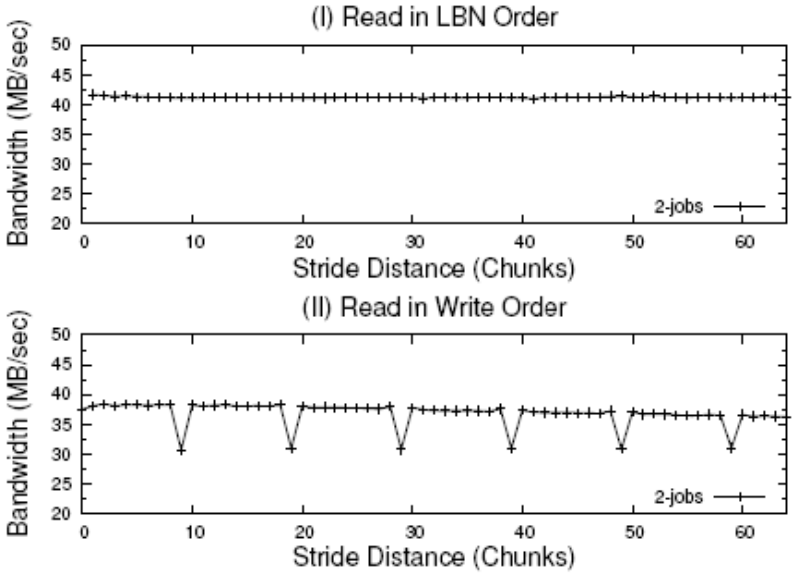
1.3.6.2.3 映射策略的探测

映射策略决定了一个逻辑页如何被映射到一个物理页。当前的固态盘产品中通常采用的映射策略主要包括以下两种：

（1）基于 LBA（逻辑块地址）的映射：对于一个给定的块，如果固态盘的交织度为 *D*，那么，这个块就会被映射到一个域编号（LBA mod *D*）。

（2）基于写顺序的映射：对于第 i 个写操作，这个块会被分配给一个域编号（$i \bmod D$）。

为了确定固态盘采用了哪种映射策略，可以首先向固态盘中随机写入 1024MB 数据，IO 请求的大小为 4KB，即一个块的大小。经过这种随机化写入操作之后，这些块会被分配到不同的域中。如果固态盘采用了基于 LBA 的映射，映射结果应该不会受到这种随机写操作的影响。但是，在经过几次反复实验后，原本随着跳读距离变化而周期性波动的带宽曲线的特征如果发生了变化，不再和原来测试结果相同，就说明随机写操作改变了块映射，那么这个固态盘肯定没有使用基于 LBA 的映射。比如，图[SSD-map-policy-test]（I）给出了采用上述方法后的固态盘映射策略测试结果，从中可以看出，带宽曲线的周期性波动特征消失了，因此，可以说明这个固态盘肯定没有使用基于 LBA 的映射。



图[SSD-map-policy-test] 固态盘映射策略测试结果

可以进一步设计实验来确认一个固态盘是否采用了基于写顺序的映射策略。首先，我们向固态盘随机写入 1024MB 空间，每个块都唯一写入一次。然后，我们采用和块被写入的顺序相同的顺序，来发起针对固态盘的读操作，并且重复这个实验过程。例如，如果随机写操作的 LBN（逻辑块编号）的顺序是（11,25,67，9……），那么，我们读数据的顺序也是（11,25,67,9……）。如果一个固态盘使用了基于写操作顺序的映射，这些块应该会被以写操作的顺序交织地分布到不同的域中，比如块 11,25,67,9 分别被交织分配到域 0,1,2,3 中。这个时候，采用相同顺序的读操作，就会重现我们之前观察到的相同的周期性波动带宽曲线。图[SSD-map-policy-test]（II）中的实验结果验证了这种假设，说明这个固态盘确实采用了基于写顺序的映射策略。

1.3.7 固态盘性能

表[HD-SSD-comparion-01] 固态盘与磁盘的特性参数表一[Chenjianqiang10]

存储介质	固态硬盘	传统磁盘
内部组成	NAND Flash、控制 IC	马达、碟片、碟盘、转轴
外观尺寸（cm3）	60～67	65～68
重量（g）	48～90	98～115
读写功耗（w）	0.13～1.0	2.4～2.5
使用中抗震度（G/ms）	>1000/0.5	250～275/2
未使用中抗震度（G/ms）	1500～2000/1	800～900/1
使用中噪音（dB）	0	2.9

表[HD-SSD-comparion-01]给出了固态盘与磁盘的特性参数，从中可以看出，固态盘没

有机械部件，体积小，重量轻，能耗低，抗震好，噪音低，相对于磁盘而言具有明显的优势。比如，在能耗方面，固态盘明显小于磁盘，固态盘在系统空闲期间的能耗甚至可以达到零。一般而言，当固态盘处于活动状态时，能耗在 0.15 到 2 瓦之间，当处于空闲状态时，可以低到 0.06 瓦。相反，SATA 磁盘的能耗在 13 到 18 瓦之间，要比固态盘高 6 到 10 倍。按照每度电 0.5 元计算，一个 SATA 磁盘持续工作三年的电费消耗将会达到大约 235 元，如果再把空调制冷等费用计算进来，总计费用将会达到大约 500 元。而对于一个固态盘而言，持续工作三年的费用只需要大约 50 元。因此，如果考虑总体的设备能耗，固态盘相对于磁盘而言具有明显的优势。

表[HD-SSD-comparison] 机械式硬盘和固态盘的特性参数表二

	SATA Disk	**SATA Flash**	**FC Flash**	**ioD Flash**
GB	500	32	146	320
$/GB	$ 0.12	$ 15.62	$ 85	$ 30
Watts（W）	13	2	8.4	6
seq.read（MB/s）	60	80	92	700
seq.write（MB/s）	55	100	108	500
ran.read（IO/s）	120	11,200	54,000	79,000
ran.write（IO/s）	120	9,600	15,000	60,000
IO/s/$	2.0	11.2	4.4	8.3
IO/s/W	9.2	5,600	6,430	13,166

表[HD-SSD-comparison]给出了一些机械式硬盘和固态盘的特性参数，从中可以看出，所有的固态盘都提供了比传统的机械式硬盘快 10-100 倍的随机读 I/O，许多企业级的固态盘也可以提供和机械式硬盘相媲美的顺序读和写带宽。但是，固态盘的随机写性能，都要比读性能差。总体而言，固态盘在以下方面超越了机械式硬盘：价格性能指标 IO/s/$（每美元的 I/O 速率）和能效指标 IO/s/W(消耗每瓦特电能产生的随机 I/O 速率)。闪存固态盘的每 GB 的价格，正在以每年 50%的速度下降，这要比机械式硬盘价格下降速度快许多。因此，我们可以预期，在每 GB 的价格方面，闪存固态盘将会和机械式硬盘逐渐缩小差距；在价格性能指标 IO/s/$和能效指标 IO/s/W 方面，固态盘会逐渐超越机械式硬盘；在许多分析应用中，固态盘会逐步取代机械式硬盘。

但是，也应该注意到，固态盘并非在所有方面都优于传统的机械式硬盘，比如，在顺序读取方面，机械式硬盘的性能仍然要好于中低端的固态盘[LeeM07]，只有一些高端的固态盘，比如 FusionIO[FusionIO]，在随机访问方面，比机械式硬盘的随机和顺序访问都具有更好的性能。因此，如果不考虑负载类型，盲目采用固态盘取代机械式硬盘，有时候会适得其反，恶化系统的性能。为了更好地说明这个问题，表[HDD-SSD-performance-comparision]给出了一款希捷的机械式硬盘产品和一款三星固态盘产品的性能比较，从中可以看出，在平均读写延迟方面，固态盘具有明显的优势，但是，在连续传输速率方面，机械式硬盘仍然要好于固态盘，也就是说，机械式硬盘具备更好的顺序读写性能。所以，在近期一段时间内，把固态盘应用于任何类型的企业应用负载是不现实的，由于固态盘可以让随机 I/O 为主的负载带来明显的性能改进，因此，固态盘当前应该重点服务于随机 I/O 负载[CanimMBRL10]。

表[HDD-SSD-performance-comparision] 机械式硬盘和闪存固态盘的性能比较

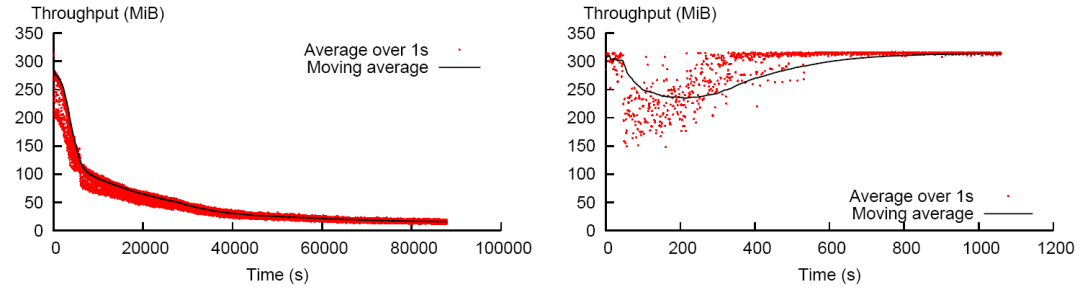
存储	磁盘	固态盘
平均延迟	8.33 毫秒	0.2 毫秒(读)

		0.4 毫秒(写)
持续传输速率	110MB/秒	56MB/秒(读) 32 MB/秒(写)

对于闪存固态盘而言，由于采用 NAND 闪存作为存储介质，而一个 NAND 闪存块被擦除的次数是有限制的，通常可以擦除 1 万到 10 万次[AgrawalPWDMP08]，因此，闪存固态盘的寿命通常比机械式硬盘的寿命要短，不过，在实际应用中，二者的寿命不会有太大差距，都可以满足应用对数据可靠性的要求。例如，在 6 年时间内，以 100MB/秒的速率对一个 250GB 的机械式硬盘进行持续的写操作，覆盖整个机械式硬盘的次数不会超过 8 万次。如果在闪存固态盘上采用一个日志结构的文件系统，以 30MB/秒的速度对一个 32GB 的闪存固态盘进行连续的覆盖写操作，达到 10 万次大概需要 3 年半的时间[Graefe07]。因此，在平均使用情况下，固态盘的寿命可以达到 5-10 年，这是企业用户可以接受的。可以看出，闪存固态盘的寿命和可靠性，还是和机械式硬盘有得一拼的。

1.3.8 固态盘 IO 特性实验测试分析

闪存的诸多特性决定了，闪存固态盘在不同的负载下会表现出和机械式硬盘不同的性能特性。文献[StoicaAJA09]在一个高端闪存固态盘 FusionIO[FusionIO]上利用一个微型的测试基准 uFLIP[BouganimJB09]进行了相关实验，展示了在大量随机写和顺序写负载下闪存吞吐量的变化情况。从图[Exp-On-FusionIO](a)可以看出，在对不同的闪存物理位置进行大量的随机写操作以后，闪存的吞吐量明显下降，性能下降幅度竟然达到了一个数量级以上，因为随机写操作会带来代价高昂的块擦除操作。在图[Exp-On-FusionIO](b)中，第 0 秒之前已经对 FusionIO 执行了大量随机写操作，此后开始执行大量顺序写操作。可以看出，在顺序读操作开始后的一段时间内（大约从第 0 到 200 秒），闪存吞吐量依然继续在下降，但是，此后慢慢增长回到了最初的高吞吐量阶段的水平，并保持稳定的性能状态。由此可以看出，闪存设备的性能高度依赖于 IO 历史，随机写操作会引起响应时间上的巨大变化，导致性能恶化，即使在随机写操作停止后的一段时间内也会如此。该文作者由此得出结论，对于闪存设备而言，如果能够把随机写操作转换成顺序写操作，并且在进行写操作的时候增大 IO 尺寸，就可以获得明显的性能改进。



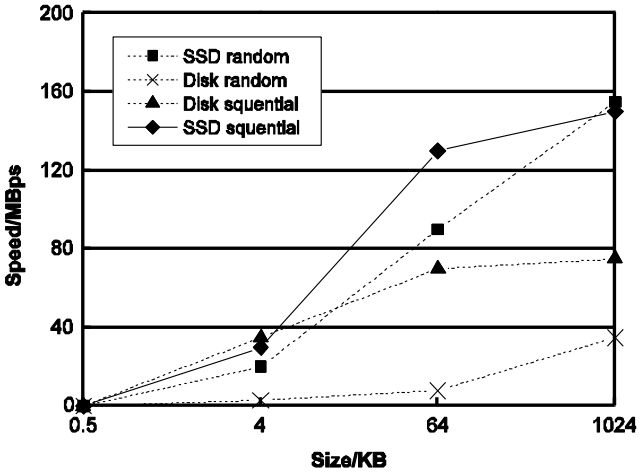
(a) Random writes (b) Sequential writes

图[Exp-On-FusionIO] 在固态盘 FusionIO 上面的随机写和顺序写

Jim Gray 等人[GrayF06]也用大量实验验证了闪存的读写特性。实验环境配置如下：32 位的 Windows Vista RC2 Build 5744 平台，双核 3.2GHz Intel x86 CPU，1 GB 的 RAM，32GB 的三星 NAND 闪存盘。实验结果显示，在以 512KB 为单位进行顺序读写请求时，闪存盘可以为顺序读写请求提供很好的服务，每秒可以执行 6,528 个读操作，每秒可以执行写操作的数量是 1,644 个，可以看出，读性能要明显比写性能好。对于随机读写请求，闪存盘的性能表现令人失望。在以 8KB 为单位进行随机读写请求时，每秒可以执行 2,800 个随机读操作，于此形成鲜明对比的是，每秒只能执行 27 个随机写操作，可见，闪存上面的随机写操作的

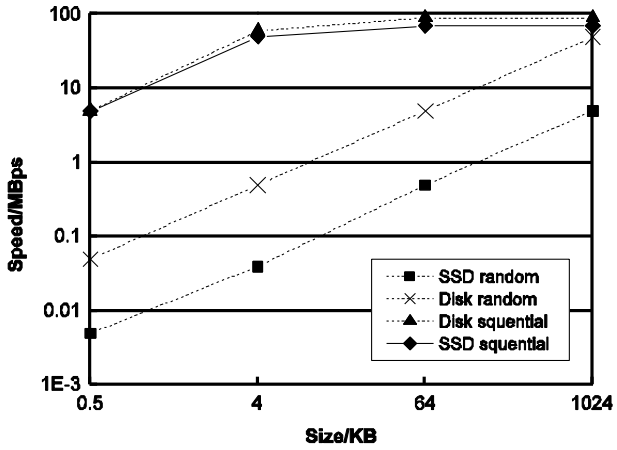
性能是比较差的。当然，导致随机写操作性能如此之差也是和实验平台相关的，Windows Vista 采用了同步写技术，即在确认写操作完成之前，必须确保数据已经被写入到稳定的存储器上，如果采用缓冲区技术，把写操作先在缓冲区里排队，达到一定数量后一次性写入到稳定存储器上，会大幅度改进写操作的性能。

文献[LvCC09]对固态盘和磁盘的相关 IO 性能进行了实验对比分析，作者选取了 PQI 32G SSD 和西部数据的机械式硬盘产品 WD5000AAKS，在 Windows XP 操作系统下完成了大量测试工作。图[cuibin-01]显示了固态盘与磁盘的顺序与随机读取速度对比情况，可以看出，固态盘的顺序读取速度高于磁盘，随机存取性能高于磁盘 1～2 个数量级，主要是因为磁盘存在机械部件，寻址时间较高。随着块大小的增加，无论固态盘还是磁盘的读取速度都显著提高；在某个特定的块大小情况下，固态盘的顺序与随机读取速度相差不多。



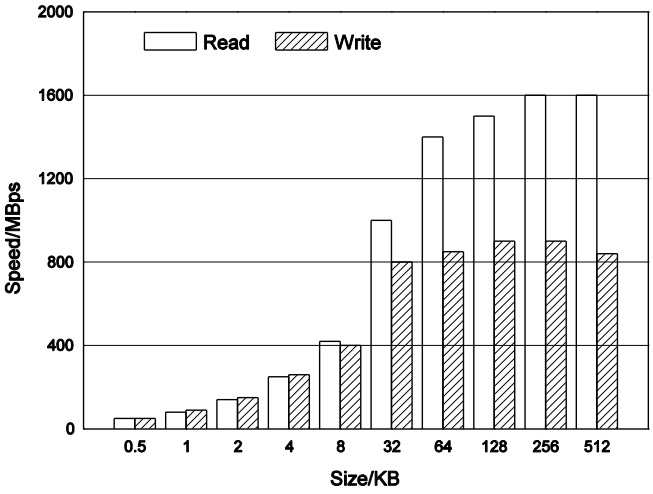
图[cuibin-01] 固态盘与磁盘的顺序与随机读取速度对比

图[cuibin-02]给出了固态盘的顺序与随机写操作速度的对比结果，从中可以看出，固态盘的顺序写操作速度远远快于随机写操作，原因在于随机写操作会引起更多的擦除操作。特别要注意的是，在这个实验结果中，固态盘的随机写操作速度甚至要比磁盘慢一个数量级。可以看出，固态盘的随机写操作代价较高，因此，在设计基于固态盘的各种应用时（比如索引机制和缓冲区替换策略），应该尽量避免随机写操作，甚至有时候不惜以增加读操作为代价。



图[cuibin-02] 固态盘的顺序与随机写入速度对比

图[cuibin-03]显示了文件系统下固态盘的读取与写入操作的速度。在块大小小于 64 KB 时，读写速度都会随着块大小的增加而显著提高，但是，当块大小超过 64KB 时，读写速度的提高幅度明显减小，不过，读与写操作依然存在比较大的速度差距。



图[cuibin-03] 文件系统下固态盘的读取与写入速度

1.4 本章小结

本章内容主要围绕闪存和固态盘，是更好地理解闪存数据库的基础。首先，介绍了计算机存储技术，给出了存储介质的类别划分和不同存储介质的特点；然后，介绍了闪存的工作原理、分类、结构、特性、产品和应用；接下来，介绍了固态盘的相关知识，包括固态盘的特性、产品和应用，展示了固态盘的内部结构，从而揭示固态盘内部并行特性，并给出探测固态盘内部特性的方法；最后，介绍了固态盘的性能，并给出了一些固态盘 IO 性能实验测试分析结果。

1.5 习题

1、 阐述计算机存储介质的分类以及不同类型存储介质的特点。
2、 阐述闪存的特性，并比较闪存与传统的机械式硬盘的特性差异。
3、 阐述固态盘结构以及各个功能组件的功能。
4、 阐述固态盘内部不同级别的并行特性的具体含义。
5、 比较不同厂家的固态盘产品和传统的机械式硬盘在不同性能参数之间的差异。

第 2 章 闪存文件系统

上层应用需要借助于闪存文件系统或者 FTL 机制，来完成对闪存芯片的直接管理和控制。虽然闪存文件系统和 FTL 机制的设计并非直接服务于数据库，但是，了解二者的工作机制，对于解决闪存数据库中的诸多问题，具有重要的借鉴作用，因此，本章重点介绍闪存文件系统，并介绍两款典型的闪存文件系统产品 JFFS2 和 Yaffs，然后，在接下来的第 3 章，将会详细介绍 FTL 机制。

2.1 闪存文件系统和闪存转换层的比较

闪存存储管理的主要目的就是为上层应用提供硬件抽象，主要包括两种方式，即闪存文件系统和闪存转换层（FTL：Flash Translation Layer）。如图[FTL-JFFS-comparison]所示，前者是直接针对闪存设计的文件系统，比如 JFFS2[Woodhouse01]（Journalling Flash File System Version 2）和 YAFFS[One08]，后者则是通过一个中间层来隐藏闪存的底层硬件细节，把闪存设备模拟成一个块设备，上层应用可以和访问磁盘一样的方式来访问闪存设备。闪存文件系统只能针对特殊厂商的闪存设备，无法广泛应用关于各种不同类型的闪存设备，通用性不强；闪存转换层可以将现有的各种常用的磁盘管理技术移植到闪存设备上，可以把各个不同厂商的闪存产品模拟成类似磁盘的块设备，具有广泛的应用范围。闪存文件系统比 FTL 具有更高的性能，一般用于固定的、非插拔的 NAND 闪存管理。表[FTL-JFFS-comparison-table]给出了闪存文件系统和闪存转换层的特点比较。

大多数闪存文件系统都借鉴了日志文件系统的设计思想，实践证明，这种设计非常适用于闪存存储设备。在传统的、不是基于日志的文件系统中，为了提高文件系统的效率，通常采用缓冲区来进行数据的读写。但是，这种方式也存在一定的隐患，比如，当发生断电事故时，如果缓冲区中的数据还没有全部刷新写入到磁盘，就会引起部分数据的丢失，而且这种数据丢失是无法恢复的。缓冲区中可能会包含一些关键数据，比如文件系统的管理信息，丢失这些数据会导致文件系统数据组织的混乱，甚至引起文件系统的崩溃。

日志文件系统可以很好地克服上述缺陷，它会在磁盘中维护一个日志文件，所有数据更新都以追加的方式写入到日志中，每个更新操作都对应于日志中的一条记录。日志文件的尾部包含了文件系统中的最新数据。当系统发生断电事故后，只需要扫描日志就可以恢复还原文件。

当前针对闪存文件系统的研究主要包括文件系统日志管理、文件系统快速初始化、文件系统崩溃恢复、垃圾收集和页面分配技术等。

图[FTL-JFFS-comparison] 闪存转换层和闪存文件系统的工作方式区别

表[FTL-JFFS-comparison-table] 闪存转换层和闪存文件系统的特点比较

	闪存文件系统	闪存转换层
原理	直接针对闪存设计的文件系统	通过一个中间层把闪存设备模拟成一个块设备
通用性	通用性不强	具有广泛的应用范围
代表产品	JFFS/JFFS2/JFFS3、LFM、YAFFS	各种 FTL 机制，比如 BAST、LAST

2.2 闪存文件系统JFFS2

JFFS2 是一个典型的基于日志结构的日志文件系统，在嵌入式 Linux 系统中得到了广泛的应用，帮助上层应用完成对闪存芯片的直接管理和控制。JFFS2 文件系统针对闪存的特性，提出了有效的管理策略，从而获得了较好的运行效率。

本节简要介绍闪存文件系统 JFFS2，并指出它的不足之处。

2.2.1 JFFS2 概述

JFFS2 是根据 JFFS 改进得到的，最初的 JFFS 是一个纯粹的基于日志的文件系统，包含数据和元数据的节点会被顺序地存放到闪存芯片上，在可用的闪存空间里严格地、按照线性方式使用存储空间。在 JFFS 当中，在日志中只有一种类型的节点，使用了结构体类型 *jffs_raw_inode* 来表示。JFFS2 的节点类型更加灵活，可以定义新的节点类型。

JFFS2 文件系统主要包括三个功能模块：块分配模块、垃圾回收模块和磨损均衡模块，各个模块的功能如下：

- 块分配模块：负责维护闪存中的可用空闲块，决定可以使用的下一个空闲块；
- 垃圾回收模块：负责跟踪闪存中的处于“无效”状态的块，当闪存空间不足时，启动垃圾回收过程，回收无效块，产生新的空闲块；
- 磨损均衡模块：负责在闪存不同的物理块之间均匀地分布写操作，延长闪存寿命。

JFFS2 把闪存块组织成多个链表（如表[JFFS2-block-list]所示），主要包括：干净块链表、脏块链表、空闲块链表。每个链表实际上就是一个按照“先进先出”原则组织的队列。当干净块链表中的某个块中的数据发生更新时，就从干净块链表中摘掉这个块，对块中的相应节点做标记后，把该块再放入到脏块链表的尾部。脏块链表中的块，都是要执行擦除操作的候选块，在执行垃圾回收时，可以选择脏块链表头部的块进行擦除。空闲块链表中的块，则是已经执行过擦除操作的空白块，可以被分配给后续到达的写操作。

表[JFFS2-block-list] JFFS2 文件系统的链表类型

链表名称	说明
clean_list	干净块链表，块内包含了没有发生更新的有效数据
dirty_list	脏块链表，块内包含了部分已经被更新过的无效数据（或脏数据）
free_list	空闲块链表，已经被擦除过的、可用的空闲块

JFFS2 文件系统被挂载时，会调用一个函数 listType()扫描每个块，并根据具体情况把每个块分别放入相应的链表中，即放入到干净块链表、脏块链表或者空闲块链表。

```
function listType()
{
    for each 闪存块 b do
    {
        if（b 是擦除过的可用空闲块）then 把 b 添加到空闲块链表尾部；
```

else if（*b* 中有过时数据）then 把 *b* 添加到脏块链表尾部；

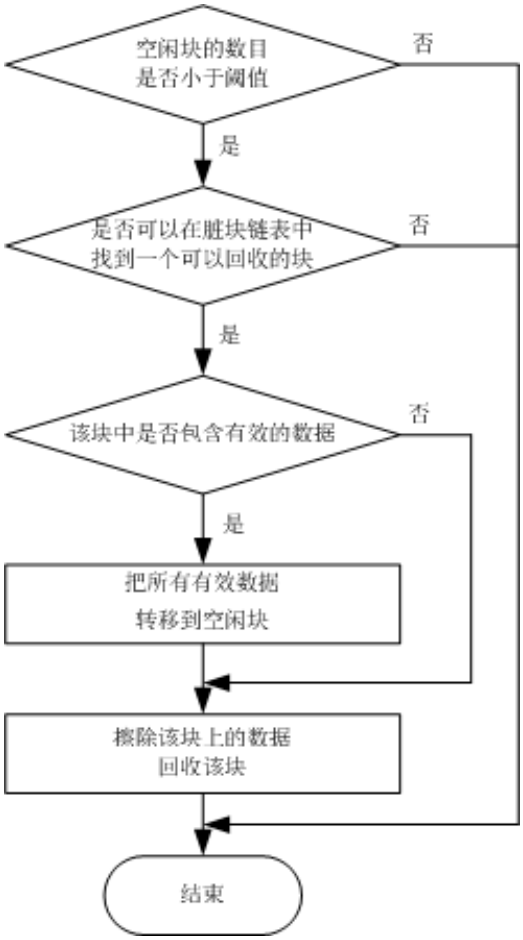
else if（*b* 中包含的数据全部都是有效数据）then 把 *b* 添加到干净块链表尾部；

}

}

随着应用程序对闪存空闲块的不断消耗，可用的空闲块的数量会变得越来越少，当减小到低于某个事先设定的阈值时，JFFS2 文件系统会调用 garbageCollection()函数，启动垃圾回收过程。如图[JFFS2-garbage-collection]所示，垃圾回收过程开始以后，系统从脏块链表中选择一个可以回收的块，如果该块中包含有效数据，还需要把有效数据复制合并到其他空闲块中，然后对该块执行擦除操作。需要特别指出的是，为了实现块的均衡磨损，garbageCollection()函数在选择要擦除的块时，会以 99%的概率从脏块链表中选择一个块，以 1%的概率从干净块链表中选择一个块。

```
function garbageCollection()
{
    产生一个随机数 random；
    if（random mod 100 不等于 0）then
        从脏块链表中选中链表头部的块；
    else 从干净块链表中选中链表头部的块；
    擦除选中的区块；
}
```



图[JFFS2-garbage-collection] JFFS2 文件系统的垃圾回收过程

2.2.2 JFFS2 的不足之处

JFFS2 文件系统的不足之处包括以下几个方面：

- 具有较长的挂载时间：JFFS2 的挂载过程需要从头到尾扫描闪存块，需要耗费较长的时间。
- 磨损平衡具有随机性：JFFS2 在选择要擦除的块时，会以 99%的概率从脏块链表中选择一个块，以 1%的概率从干净块链表中选择一个块。这种概率的方法，很难保证磨损的均衡性。在某些情况下，甚至可能造成对块的不必要的擦除操作，或者引起磨损平衡调整的不及时。
- 可扩展性较差：JFFS2 在挂载时，需要扫描整个闪存空间，因此，挂载时间和闪存空间大小成正比。另外，JFFS2 对内存的占用量也和闪存块数成正比。在实际应用中，JFFS2 最大能用在 128MB 的闪存上。

此外，JFFS2 在 NAND 闪存上可能无法取得很好的性能，主要原因包括：

（1） NAND 闪存设备通常要比 NOR 闪存设备的容量大许多，因此，在和闪存管理相关的数据结构方面，前者也会比后者大许多。而且，JFFS2 的挂载时间和闪存空间大小成正比，因此，对于容量普遍较大的 NAND 闪存设备，JFFS2 的扫描时间会较长，无法取得好的性能。

（2） NAND 闪存设备是以页为单位访问数据，如果只想访问页中的一部分数据，也必须顺序读取整个页的全部数据，而无法跳过不需要的数据。这就减慢了扫描和挂载的过程。

2.3 闪存文件系统Yaffs

2.3.1 Yaffs 概述

Yaffs 是一个专门为 NAND 和 NOR 闪存设计的开源文件系统[Yaffs]，它已经被广泛应用于 Linux 和 RTOSs 等操作系统中。Yaffs 是 Yet Another Flash File System 的缩写，是由查尔斯.曼宁（Charles Manning）在 2001 年提出的。Yaffs 的第一个版本是在 2001 年年末到 2002 年年初开发的，到了 2002 年中期的时候，Yaffs 开始采用不同的产品。在 2002 年 5 月，Yaffs 被正式宣布，并产生了更多的衍生项目；6 月，开始提供对其他操作系统的支持；9 月，Yaffs 被宣布可以应用于 Linux 设备上。

Yaffs 代码包括两种版本，即最初版本的 Yaffs 代码和当前版本的 Yaffs2 代码。Yaffs2 代码支持由最初版本的 Yaffs 代码提供的功能，并且提供了扩展的功能，可以支持额外的操作模式，这些操作模式对于更大、更现代的 NAND 闪存而言是必需的。Yaffs2 提供了和 Yaffs 代码的前向兼容性。Yaffs、Yaffs1 和 Yaffs2 三种术语的区分如下：

（1） Yaffs1 是一种更加简单的操作模式，使用了“删除标记位”来记录闪存页的状态。

（2） Yaffs2 是一种更加复杂的操作模式，可以支持更大类型的闪存，这类闪存不能使用删除标记位。

（3） Yaffs 是 Yaffs1 和 Yaffs2 这二者的公共操作模式。

Yaffs1 作为 Yaffs 的最初版本，通常只会工作在页大小为 512 字节的闪存设备上，采用了一个类似于 SM 卡（SmartMedia）的内存布局，但是，Yaffs1 已经被扩展为可以支持更大的闪存页，比如页大小为 1K 的 Intel M18 闪存。Yaffs 是 Yaffs1 的升级产品，可以支持更大的、不同的设备。Yaffs 很容易进行封装，目前已经被应用到不同的产品中，比如 POS 机、电话和航空设备等。Yaffs 也已经被应用到多个不同的操作系统中，可以支持的操作系统包括 Linux、Windows CE 和 eCOS 等。

注：SmartMedia(SM 卡)是一种快闪存储器，由 Toshiba（东芝）公司在 1995 年夏季推出，

用来对抗 MiniCard、CompactFlash 和 PC Card 等存储卡标准。SM 卡曾是数码相机普遍支持的存储格式，得到了富士相机和奥林巴斯相机的大力支持，并在 2001 年左右达到巅峰，当时差不多占据了 50%的市场份额。但是，2001 之后 SM 卡开始走下坡路，这其中包含多个方面的原因：（1）无法提供大容量的存储，几乎看不到 128MB 以上容量的 SM 卡；（2）随着数码相机尺寸的不断缩小，SM 卡的尺寸相对而言就显得太大了；（3）奥林巴斯放弃支持 SM 卡，转而支持 SD 卡。总之，由于缺少更多的新设备支持 SM 卡，SM 卡从市场上消失只是时间问题。

Yaffs 被设计成可以在多种环境下工作，这就催生了对可移植性的需求。对可移植性的支持，也改进了测试工作，因为，在应用环境中要比在一个操作系统内核中更加容易进行代码的开发和测试工作。可移植性使得 Yaffs 可以在具有更少资源的情况下，获得更快的发展。Yaffs 改进可移植性的主要策略包括：

（1） 在主体代码中不使用与操作系统相关的特性；

（2） 在主体代码中不使用与编译器相关的特性；

（3） 使用抽象类型和函数来支持 Unicode 和 ASCII 操作。

为了增强鲁棒性，便于集成和开发，Yaffs 采用了尽可能简单的设计理念，主要策略是：

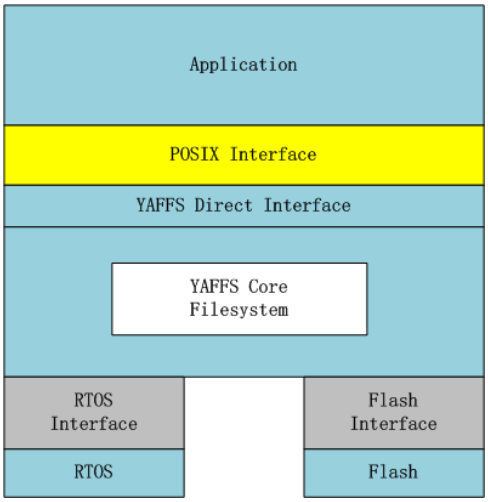
（1） Yaffs 采用单线程模型；Yaffs 的加锁粒度是分区，这要比在更低粒度（比如块或页）上使用锁更加简单。

（2） 基于日志结构的文件系统使得垃圾回收更加简单。

在 Yaffs 中，空间分配的单元是“厚片”（chunk）。通常，一个厚片和一个 NAND 闪存页是一致的，但是，厚片更加灵活，可以包含多个页，这种灵活性使得系统可以根据需要进行配置。许多个厚片构成一个块，一个块是执行擦除操作的基本单元。NAND 闪存可能在出厂时就有坏块，而且在设备使用过程中也会出现坏块，因此，为了可以探测到坏块，Yaffs 必须对坏块进行标记和检测。为了实现这个目的，NAND 闪存通常会使用一些错误探测机制和错误纠正码。Yaffs 可以使用闪存自带的 ECC 或者使用自己的方法，来实现闪存错误检测。

2.3.2 Yaffs 体系架构

图[Yaffs-structure]给出了 Yaffs 体系架构图，从中可以看出 Yaffs 文件管理系统在整个应用环境中所处的位置，以及 Yaffs 文件系统、实时操作系统（RTOS）、闪存设备、Yaffs 接口（Yaffs Direct Interface）和 POSIX 之间的关系。使用 Yaffs 接口，可以把 Yaffs 移植到一个嵌入式环境或者实时操作系统中。



图[Yaffs-structure] Yaffs 体系架构图

2.3.3 Yaffs1 如何存储文件

Yaffs1 和 Yaffs2 在存储文件的方式上，存在一些差别。Yaffs2 采用了真正的日志结构，即任何时候都只能顺序追加写入日志记录。而 Yaffs1 则采用了修改过的日志结构，使用了“删除标记位”，破坏了顺序写入日志的规则，而 Yaffs2 则不使用删除标记位。

Yaffs1 并没有给任何文件分配指定的数据存储区域，因此，每个文件的数据并非被写入到与该文件相关的区域，而是以顺序日志的形式写入闪存。日志中的每条日志记录，都占用闪存的一个厚片（注意，如果“厚片”这个概念不好理解的话，可以直接把“厚片”理解成“页”）。Yaffs1 把厚片分成两种不同的类型：

（1）数据厚片：包含了常规数据内容，也就是文件中保存的数据；

（2）对象头部：是关于一个对象的描述符，包括父目录的标识符、对象名称等。Yaffs 中的每个对象可以是一个常规文件、目录、硬链接、符号链接或特定链接。对于一个文件对象而言，当文件创建、文件裁减或文件关闭时，都可能会创建一个对象头部，通过对象头部中存储的元数据，就可以获得对象名称和文件长度等信息。

每个厚片都有和它关联的标签，包括以下字段：

（1） 对象编号：用来确定一个厚片属于哪个对象；

（2） 厚片编号：用来确定一个厚片属于文件的哪个部分；厚片编号是 0，表示这个厚片存储了一个对象头部。厚片编号是 1，表示这个厚片是文件中的第一个厚片，厚片编号是 2，表示这是文件中的下一个厚片，依此类推。

（3） 删除标记位：只在 Yaffs1 中存在，Yaffs2 中不存在，表示这个厚片不再有用。

（4） 字节计数：如果这是一个数据厚片，就表示数据的字节数。

（5） 序列号：只在 Yaffs1 中存在，Yaffs2 中不存在，当几个厚片具有相同的对象编号和厚片编号时，就可以使用序列号加以区分。

为了更好地理解 Yaffs1 的文件存储方式，这里给出一个简单的实例。这里假设每个块中包含了 4 个厚片，开始的时候，所有的厚片都是可用的。下面分成六步执行各种操作，在每步结束后，这里都会给出各个块中的各个厚片的状态，并给予简单描述。

第一步：创建一个文件。文件系统为新文件分配了第 1 块第 0 厚片，用来存储新文件的对象头部，该厚片的“删除标记位”是“有效”，说明厚片中包含的数据当前是有效的，没有被删除。新文件的对象编号是 300，由于当前文件中不包含任何数据，因此，对象头部中关于文件长度的元数据的值是 0。由于第 1 块第 0 厚片存储的是对象头部，因此，厚片编号是 0。

块	厚片	对象编号	厚片编号	删除标记位	备注
1	0	300	0	有效	文件的对象头部 (长度是 0)

第二步：写一些数据到文件中。这里连续写入数据到三个厚片中，因此，第 1 块的第 1、2、3 厚片，都是数据厚片，保存了刚写入的数据，这些厚片的“删除标记位”是“有效”。新写入的这三个厚片的数据，都属于对象编号为 300 的文件，因此，这三个厚片的“对象编号”值都是 300。第 1 块第 1 厚片是文件的第一个数据厚片，因此，厚片编号是 1；第 1 块第 2 厚片是文件的第二个数据厚片，因此，厚片编号是 2；第 1 块第 3 厚片是文件的第三个数据厚片，因此，厚片编号是 3。

块	厚片	对象编号	厚片编号	删除标记位	备注
1	0	300	0	有效	文件的对象头部 (长度是 0)
1	1	300	1	有效	第一个数据厚片
1	2	300	2	有效	第二个数据厚片

1	3	300	3	有效	第三个数据厚片

第三步：关闭文件。由于在第二步中为文件写入了新数据，因此，在关闭文件时，必须为这个文件创建一个新的对象头部，使得它能够反映这种变化。由于第 1 个块只能容纳 4 个厚片，已经满了，因此，需要在第 2 块第 0 厚片存储这个新的对象头部，并把“对象编号”值设置为 300，删除标记位设置为“有效”，并且要在对象头部中记录新的文件长度 *n*。原来保存在第 1 块第 0 厚片中的对象头部，就会废弃，因此，把这个厚片的删除标记位设置为“删除”。由于第 2 块第 0 厚片存储的是对象头部，因此，厚片编号是 0。

块	厚片	对象编号	厚片编号	删除标记位	备注
1	0	300	0	删除	废弃的对象头部(长度是 0)
1	1	300	1	有效	第一个数据厚片
1	2	300	2	有效	第二个数据厚片
1	3	300	3	有效	第三个数据厚片
2	0	300	0	有效	新的对象头部(长度为 *n*)

第四步：打开文件进行读写，把文件中的第一个数据厚片（即第 1 块第 0 厚片）进行重写，然后关闭文件。由于闪存采用异地更新，因此，重写操作并非直接对第 1 块第 0 厚片的数据进行修改，而是把重写后的新数据写入到新分配的第 2 块第 1 厚片中，并把原来旧的第 1 块第 0 厚片废弃掉，即把它的删除标记位设置为“删除”。由于第 2 块第 1 厚片存储的是文件的第一个厚片，因此，厚片编号是 1。此外，由于文件发生了更新，因此，必须为这个文件创建一个新的对象头部，使得它能够反映这种变化。这里在第 2 块第 2 厚片中写入新的对象头部，原来旧的对象头部所在的厚片（第 2 块第 0 厚片），就会被废弃掉，即把该厚片标记为“删除”。

块	厚片	对象编号	厚片编号	删除标记位	备注
1	0	300	0	删除	废弃的对象头部
1	1	300	1	删除	废弃的第一个数据厚片
1	2	300	2	有效	第二个数据厚片
1	3	300	3	有效	第三个数据厚片
2	0	300	0	删除	废弃的对象头部
2	1	300	1	有效	新的第一个数据厚片
2	2	300	0	有效	新的对象头部(长度为 *n*)

第五步：把文件的大小变为 0，方法是：使用 O_TRUNC 打开文件，然后关闭文件。由于文件长度发生了变化，必须为这个文件创建一个新的对象头部，这里在第 2 块第 3 厚片中写入新的对象头部，原来旧的对象头部所在的厚片（第 2 块第 2 厚片），就会被废弃掉，即把该厚片标记为“删除”。由于文件已经被裁减成长度为 0，不包含任何数据内容，因此，需要把几个数据厚片（即第 1 块第 2 厚片，第 1 块第 3 厚片，第 2 块第 1 厚片），都废弃掉，即把它们的删除标记为设置为“删除”。通过这种方式，文件就实现了裁减，数据厚片中的数据就被裁减掉了。

块	厚片	对象编号	厚片编号	删除标记位	备注
1	0	300	0	删除	废弃的对象头部
1	1	300	1	删除	废弃的第一个数据厚片
1	2	300	2	删除	第二个数据厚片

1	3	300	3	删除	第三个数据厚片
2	0	300	0	删除	废弃的对象头部
2	1	300	1	删除	删除的第一个数据厚片
2	2	300	0	删除	废弃的对象头部
2	3	300	0	有效	新的对象头部(长度为 0)

第六步：这个时候，第 1 块中的所有厚片都已经被标记为“删除”，这就意味着第 1 块不再包含有用的信息，因此，现在可以擦除第 1 块并且重用空间。这里可以重命名文件，为了完成这个事情，可以在第 3 块第 0 厚片中为这个文件写入一个新的对象头部，并把原来旧的第 2 块第 3 厚片中的对象头部废弃掉。

块	厚片	对象编号	厚片编号	删除标记位	备注
1	0				擦除
1	1				擦除
1	2				擦除
1	3				擦除
2	0	300	0	删除	废弃的对象头部
2	1	300	1	删除	删除的第一个数据厚片
2	2	300	2	删除	废弃的对象头部
2	3	300	0	删除	废弃的对象头部
3	0	300	0	有效	新的对象头部显示了新的文件名称

通过观察可以发现，第 2 块现在只包含标记为“删除”的厚片，因此，也可以被擦除和重用。需要指出的是，厚片中的标签告诉我们一些非常有用的信息，包括哪个厚片属于哪个对象、厚片在文件中的位置、哪个厚片是当前的等等。具备这些信息以后，我们就可以在系统挂载的时候通过扫描各个厚片来重新创建文件的状态。这就意味着，当系统失败的时候（失败可能发生在顺序写入的任何时间点），我们就可以把系统恢复到失败发生之前的那个点。

2.3.4 垃圾回收

由于闪存采用异地更新，这导致一个闪存块中会同时包含一些有效厚片和删除厚片。随着系统的运行，被标记为删除的厚片会越来越多，导致可用的闪存空间会越来越少。由于这些删除厚片分散在不同的闪存块中，而且和有效厚片夹杂在一起，因此，必须采用有效的垃圾回收机制来回收这些被删除厚片占据的闪存空间。垃圾回收对于闪存文件系统而言，通常是影响性能的决定性因素。许多闪存文件系统，需要在一次垃圾回收过程中做许多工作。Yaffs 在每次只需要处理一个块，这就减少了在一次垃圾回收过程中所需要完成的工作量，因此，减少了系统阻塞时间。

Yaffs 垃圾回收过程如下：

（1） 利用启发式算法寻找一个值得回收的块，如果没有找到，就退出垃圾回收过程；

（2） 遍历块中的每个厚片，如果这个厚片正在使用，就为这个厚片创建一个新的副本，并且删除原来的厚片。修改 RAM 数据结构，从而反映这种变化。

一旦块中的所有厚片都已经被删除，这个块就可以被擦除重用。注意，上面的步骤（2）中，虽然在对某个厚片执行复制以后删除了原来厚片，但是，实际上没有必要执行真正的删除操作，因为，整个块会被执行擦除操作。

为了改善系统整体性能，Yaffs 会尽可能延迟垃圾回收过程，从而减少垃圾回收的次数。Yaffs 用来确定一个块是否值得回收的启发式算法如下：

（1） 如果还存在许多可用的擦除块，那么，Yaffs 就不需要竭尽全力执行垃圾回收过程，

而是只会尝试对那些只包含很少有效厚片的块执行垃圾回收，这个过程称为“被动垃圾回收”，它把一个块的垃圾回收工作分摊到多个垃圾回收过程中。

（2） 当擦除块已经很少时，Yaffs 就需要竭尽全力执行垃圾回收过程，来回收更多的空间，这时会对那些包含较多有效厚片的块也进行回收，这个过程称为“积极垃圾回收”，整个块就会在单个垃圾回收过程中被收回。

2.3.5 Yaffs1 序列号

Yaffs1 在发生文件修改时，经常需要执行“删除旧厚片和写入新厚片”的操作，如果这个执行过程发生故障，就会导致系统中同时存在多个具有相同标签值得厚片，那么，该如何判断哪个厚片是旧厚片，哪个厚片是当前厚片呢？Yaffs1 采用序列号解决这个问题。

为了说明序列号的重要作用，现在假设没有序列号，看看会发生什么问题。首先考虑一个文件的重命名操作，也就是说，要把一个对象头部厚片用一个新的厚片（包含新名称）来代替。为了完成这个重命名操作，Yaffs 需要完成以下工作：

（1） 删除旧的厚片；

（2） 写一个新的厚片；

如果上述执行过程中系统部发生任何故障，整个过程可以顺利执行结束，重命名操作就可以顺利完成。但是，这里假设系统在完成上述工作的过程中会发生故障，在删除操作执行后，文件系统失败了，这时，系统就会缺少相应的厚片与相应的标签值对应，这就可能会引起文件丢失。为了避免这个问题，我们必须在删除旧的厚片之前写入新的厚片。但是，这种做法仍然无法彻底解决问题，这种做法的操作序列如下：

（1） 写入一个新的厚片；

（2） 删除旧的厚片；

再次假设系统在完成上述工作的过程中会发生故障，当写入一个新的厚片这个操作完成以后，发生了系统失败，这时，系统中就会存在两个厚片和同一个标签值对应，同样无法判定哪个厚片是当前值。

为了解决上述问题，Yaffs1 引入了序列号。在 Yaffs1 中，每个块都被标记为一个 2 位的序列号，每次当一个具有相同标签的厚片的值被更新替换时，序列号都会递增，可以通过检查序列号来确定哪个是当前的厚片。由于序列号只会在删除旧的厚片之前增加 1，一个 2 位的序列号就足够了。表[old-current-chunk]给出了有效的<旧厚片,当前厚片>配对情况。

表[old-current-chunk] 旧厚片序列号和当前厚片序列号配对情况

旧厚片序列号	当前厚片序列号
00	01
01	10
10	11
11	00

因此，根据表[old-current-chunk]，就可以通过附加在旧厚片和新厚片上面的序列号，来确定哪个是旧的厚片，哪个是新的厚片，就可以删除旧的厚片。

2.3.6 Yaffs2 NAND 模型

Yaffs2 是 Yaffs 的一个扩充，用来实现一个新的目标集合，包括：

第一，在一个块内执行顺序厚片写操作。更多现代 NAND 闪存的可靠性规范，通常都假设顺序写。由于 Yaffs2 没有写入删除标记，一个块内部的厚片写操作是严格顺序执行的。

第二，零重写。Yaffs1 和 Yaffs2 的很大区别就是，前者有删除标记位，而后者没有，后者是真正的顺序日志结构。虽然 Yaffs1 只需要在厚片的“带外区域”修改一个字节，就可以设置删除标记位，但是，更多现代闪存设备都不允许重写，因此，为了更好地支持更现代、更大的闪存设备，Yaffs2 根本就不执行重写，真正实现了“零重写”。不过，在 Yaffs2 中仍然会

使用厚片删除的概念，只不过被删除的厚片会在内存数据结构中设置删除标记，这个删除标记不会被写入到闪存中。但是，这里有一个问题，如果闪存中的厚片没有删除标记位，那么 Yaffs2 在执行扫描时是怎样识别哪个厚片是当前的、哪个厚片是已经被删除的呢？为了能够区分当前厚片和旧厚片，Yaffs2 提供了另外一种机制，具体如下：

（1）顺序号。Yaffs2 的顺序号和 Yaffs1 的序列号不是一个概念。随着每个块被分配，Yaffs2 文件系统的顺序号会递增，块中的每个厚片都会被标记为该顺序号。顺序号可以确定日志的时间先后顺序，帮助 Yaffs 恢复文件系统状态，这是通过以时间的顺序后向扫描日志来实现的，即从最高顺序号扫描到最低顺序号。由于采用后向扫描，最近被写入的“对象编号:厚片编号”配对，会被首先扫描，所有后续的相应厚片一定是废弃的，将会被视为删除。在对象头部中的文件长度，可以被用来跟踪重新修改文件长度后的文件。如果一个对象头部已经被读取，那么，那些超出这个文件长度的后续的数据厚片，很显然就是废弃的，会被视为删除。

（2）缩减头部标志。用来标记对象头部，写入这种标志后，可以缩减文件大小。后面会详细阐述这个问题。缩减头部标记的用途，解释起来会有点绕。缩减头部标记是用来表明一个文件的长度已经发生缩减，可以防止这些对象头部被垃圾回收过程删除。为了更好地理解这个问题，这里假设如果不存在缩减头部标记的话，将会发生什么情况？

为了简化起见，这里假设在下面操作执行过程中没有发生垃圾回收，这种假设是完全可以保证做到的。

现在考虑一个文件，它经历了下面的操作：

第 1 步：f = open(“xmu”,O_CREAT| O_RDWR, S_IREAD|S_IWRITE); /*创建文件*/

第 2 步：write(f,data,6*1024*1024); /*写入 6 MB 数据*/

第 3 步：truncate(f,2*1024*1024); /*把文件长度裁减到 2MB */

第 4 步：lseek(f,3*1024*1024, SEEK_SET); /*把文件访问位置设置到 3MB 的位置*/

第 5 步：write(f,data,1024*1024); /* 写入 1MB 数据*/

第 6 步：close(f);/*关闭文件*/

经过上述操作以后，文件的长度将会是 4MB，但是，在 2MB 和 3MB 之间会存在一个“洞”，其中包含的数据是已经被裁减掉的，不应该被读取，根据 POSIX 标准，这个洞应该总是被读取为 0。

注：POSIX 表示可移植操作系统接口（Portable Operating System Interface，缩写为 POSIX 是为了读音更像 UNIX）。POSIX 标准最初是由美国电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）最初开发的，目的是为了提高 UNIX 环境下应用程序的可移植性。然而，POSIX 并不局限于 UNIX。许多其它的操作系统，例如 DEC OpenVMS 也支持 POSIX 标准。

Yaffs2 在 NAND 闪存中处理上面的文件操作过程时，会产生下面的厚片序列：

（1） 文件创建后的对象头部（文件长度是 0）；

（2） 6MB 的数据厚片（0 到 6MB）；

（3） 发生文件裁减后的对象头部（文件长度是 2MB）；

（4） 1MB 的数据厚片（3MB 到 4MB）；

（5） 文件关闭后的对象头部（文件长度是 4MB）。

在上面这些厚片中，只有下面这些厚片包含了当前最新值：

（1） 在上面第 2 步中创建的第 1 个 1MB 数据；

（2） 在上面的第 4 步中创建的 1MB 数据；

（3） 在上面的第 5 步中创建的对象头部。

Yaffs 在读取文件内容时，应该只去读取包含当前数据的厚片，而不能去读取“洞”中的数据，否则，就会得到不正确的结果。为了避免读取“洞”中的数据从而获得正确的结果，Yaffs 必须记住在第 3 步中发生的文件裁减操作。为此，当创建了一个洞的时候，Yaffs 会首先写入一个“缩减头部”，用来表明洞的开始，并且写入一个常规对象头部，用来表明洞的结束。“缩减头部标记”改变了垃圾回收过程的行为，从而确保这些缩减头部不会被垃圾回收过程给擦除掉，直到确认安全时才会被擦除。这对于那些大量使用带洞的文件系统而言，这样做可能会带来负面性能影响。此外，缩减头部也可以表明一个文件已经被裁减，但是文件中被裁减部分的记录没有丢失。

2.3.7 坏块和 NAND 错误的处理

NAND 闪存被设计成具有很低的代价和很高的密度。为了获得更高的产出，从而减小产品成本，NAND 闪存通常都会在出厂时包含一些坏块，此外，在闪存设备使用过程中，也会出现一些新的坏块。因此，对于任何闪存文件系统而言，如果不具备坏块处理机制，就不适合用来管理闪存。

Yaffs1 使用 SM 卡类型的坏块标记，它会检查闪存页的带外区域的第 6 个字节（字节 5），如果是一个好块，这个字节应该是 0XFF，如果一个块在出厂时已经被标记为坏块，这个字节应该是 0X00。当闪存设备在后续使用过程中出现坏块时，Yaffs1 会使用自己的方式对坏块进行标记，从而和工厂标记的坏块加以区分。当读或写操作失败的时候，或者当三个 ECC 错误被探测到的时候，Yaffs 就会把一个块标记为“坏块”。ECC 可以通过硬件实现，也可以通过软件驱动实现，或者在 Yaffs 自身内部实现。需要再次强调的是，任何缺乏有效 ECC 处理机制的闪存文件系统，是不适合作为闪存管理的。如果 Yaffs1 确定一个块是坏块，它就会把该块标记为 0X59（’Y’）。一个块被标记为坏块以后，就会退出使用，从而改进了文件系统的鲁棒性。

Yaffs2 模式被设计成支持更大范围的设备和带外区域布局。因此，Yaffs2 没有确定带外区域的哪个字节作为标记位，而是调用驱动函数来确定一个块是否是坏块以及标记它为坏块。Yaffs2 没有提供内置的 ECC，而是需要由驱动程序提供 ECC。

2.3.8 内存数据结构

在理论上，只需要很少的内存数据结构就可以实现一个基于日志结构的文件系统，但是，这会降低系统性能。因此，需要设计有效的内存数据结构，来存储足够的信息，从而提供较高的性能。

在介绍具体的内存数据结构之前，需要首先指出 Yaffs 中的各种内存数据结构到底服务于什么目的。主要的内存数据结构是在 yaffs_guts.h 中定义的，它们的主要目的是：

（1） 设备/分区：名称是 yaffs_dev，用来维护和 Yaffs 分区或挂载点相关的信息，可以允许 Yaffs 同时支持多个分区和不同类型的分区。实际上，几乎其他内存数据结构都是这个数据结构的一部分，或者通过这个数据结构来访问。

（2） NAND 块信息：名称是 yaffs_block_info，维护了闪存块的当前状态。每个 yaffs_dev 都有一组这类信息。

（3） NAND 厚片信息：这是一个附加在 yaffs_ dev 上面的“位字段”，维护了系统中每个厚片的当前使用情况。分区中的每个厚片存在与之对应的一个位。

（4） 对象：名称是 yaffs_obj，维护了一个对象的状态。在文件系统中，每个对象都有一个 yaffs_obj，每个对象可以是一个常规文件、目录、硬链接、符号链接或特定链接。每个对象通常使用对象编号进行唯一标识。yaffs_obj 的主要功能是，存储绝大多数的对象元数据。这就意味着，关于一个对象的绝大多数信息，都可以被立即访问，而不需要读取闪存。元数据包括对象编号、指向父目录的父指针、短名称、对象类型和授权等。特别需要说明的是，短名称具有很好的用途，如果对象名字足够短，

就可以放入一个小的固定尺寸的数组中，不需要每次在请求的时候都从闪存中读取出来。

（5） 文件结构：对于每个文件结构，Yaffs 会为其维护一棵树，可以定位一个文件中的数据厚片的位置。树中包含了称为 yaffs_tnode 的节点。

（6） 目录结构：目录结构允许对象采用名称进行查找，提供了一种快速的对象查找机制。

（7） 对象编号哈希表：它提供了一种根据对象编号来寻找对象的机制，在垃圾回收、扫描和类似操作中，都需要用到这个特性。每个 yaffs_dev 使用一个哈希表，哈希表有 256 个桶，每个对象编号会被使用一个哈希函数来找到与之对应的哈希桶。每个哈希桶都有一个属于该桶的对象的双向链表，每个哈希桶同时包含了该桶中对象数目的计数器。Yaffs 会尽量采用合理的策略来分配对象编号，从而使得每个哈希桶中的对象数量尽可能低，这样可以防止任何哈希桶中的双向链表过长。

（8） 缓存：Yaffs 提供了一个读/写缓存，它可以明显改进短操作的性能。缓存的尺寸可以在运行时进行设定。

下面的内容会重点介绍两种内存数据结构的使用方法，即目录结构和文件对象。

2.3.8.1 目录结构

目录结构的目的，就是通过名字快速访问一个对象。对于执行文件操作而言，比如打开、重命名，这是必须的。Yaffs 目录结构是采用一个对象树来组织的。每个 yaffs_obj 对象，都有一个称为兄弟的双向链表节点，它会把一个目录中的所有兄弟节点链接到一起。每个 Yaffs 分区都有一个根目录，根目录下面可以创建子目录。每个 Yaffs 分区也都有一些特定目的的“伪目录”，它不会被保存到 NAND 闪存中，而是在挂载的时候创建的：

（1） “丢失+查找”目录：这个目录是用来存储任何丢失的文件部分，它们已经不能被保存到正常的目录中。

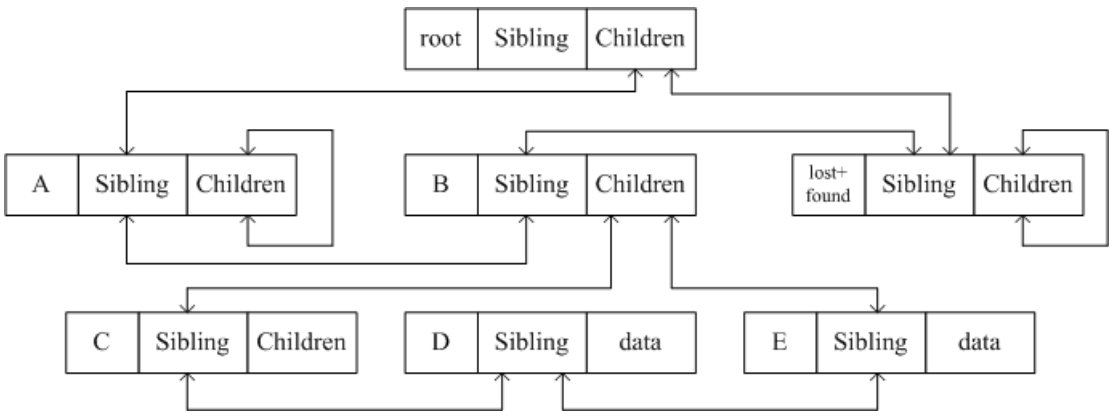
（2） “解链(unlinked)和删除”目录：这些都是伪目录，不会出现在目录树中。在这些目录中放置对象，表明它们是处于解链和删除的状态。

每个对象都有一个名称，可以通过对象名称在目录中查找一个对象。因此，一个目录中的对象名称必须是唯一的。这个规则对于已经解链或删除的伪目录中的对象不适用，因为，我们从来不会使用对象名称来查询解链或删除的文件（可能会存在许多名称相同的、已经被删除的文件）。

可以通过两种方式来加速名称的解析：

（1） 把短名称保存到 yaffs_obj 的目录中，从而不必从闪存中加载。

（2） 每个对象都有一个“名称概要”，这是一个关于对象名称的简单的哈希结果，可以加速匹配的过程。因为，比较哈希结果要比比较对象名称来得快。



图[Yaffs-directory-tree] 一个 Yaffs 目录结构的实例

图[Yaffs-directory-tree]给出了 Yaffs 目录结构的一个实例，它对应的目录详细信息如表[Yaffs-directory]所示。可以看出，根目录下存在两个一级子目录 A 和 B，其中，子目录 A 是空目录，不包含任何子目录和文件，目录 B 下存在一个二级子目录 C 和两个数据文件 D,E，目录 C 可能会继续包含更多的子目录，但是，为了简化起见，图中没有给出。

表[Yaffs-directory] 一个 Yaffs 目录结构的实例

/	根目录
/A	目录 A 是空目录
/B	目录名称是 B
/lost+found	空目录
/B/C	目录名称是 C
/B/D	D 是数据文件
/B/E	E 是数据文件

2.3.8.2 文件对象

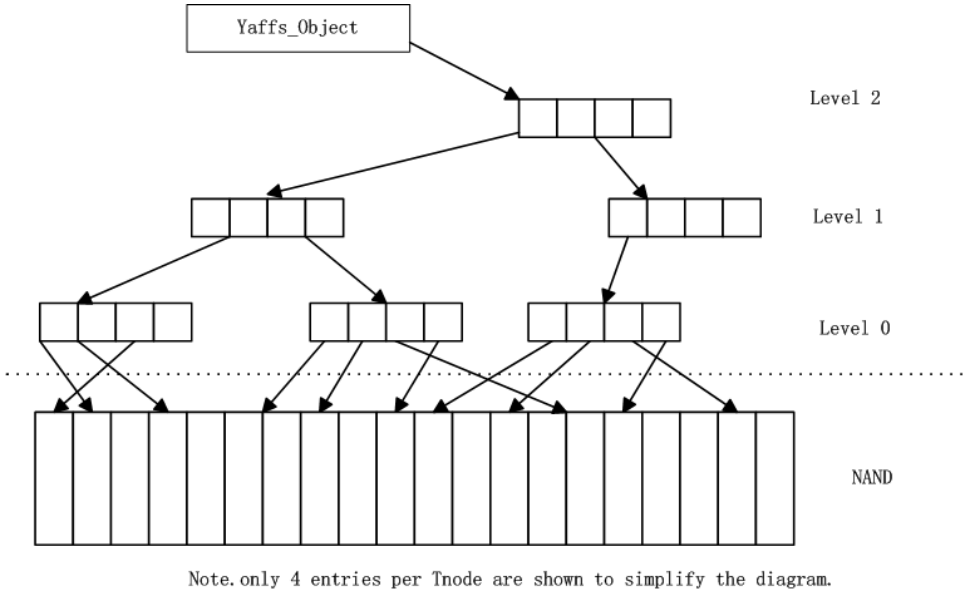
每个文件对象都有一棵 tnode 树，它是一个分层的索引结构，用来提供从文件逻辑位置到实际的 NAND 闪存厚片物理地址的映射。文件对象存储了下面的主要信息：

（1） 文件尺寸；

（2） topLevel：Tnode 树的深度；

（3） top：指向 Tnode 树顶端的指针。

top 和 topLevel 一起工作来控制 tnode 树。



图[yaffs-tnode-tree] Yaffs 文件对象的 tnode 树

如图[yaffs-tnode-tree]所示，tnode 树是由 tnode 节点的层次结构构成的，每个 tnode 节点都保存了以下信息：

（1） 在第 0 层及其以上的层：这些层上的 tnode 节点也被称为内部节点，一个内部节点具有 8 个指针，指向其它在它下面层的 tnode 节点；

（2） 在第 0 层：一个 tnode 具有 16 个 NAND 闪存厚片编号，用来确定 RAM 内存中的厚片位置。

当文件刚被创建的时候，它只会被分配一个低层的 tnode。当文件的内容超出了单个 tnode 节点可以容纳的范围的时候，系统会为它分配第二个 tnode 节点，同时创建一个内部

节点指向这两个 tnode 节点。随着文件尺寸的增加，更多的 tnode 节点会被增加进来，当一个层满时，会产生更多的层。如果文件被裁减，或者如果一个文件中的厚片被替换（即数据被重写或者被垃圾回收过程复制走了），那么，tnode 节点树必须被更新，从而反映这种变化。

2.3.9 不同机制是如何工作的

前面内容已经详细介绍了 Yaffs 的数据结构，现在来阐述 Yaffs 的一些工作机制。

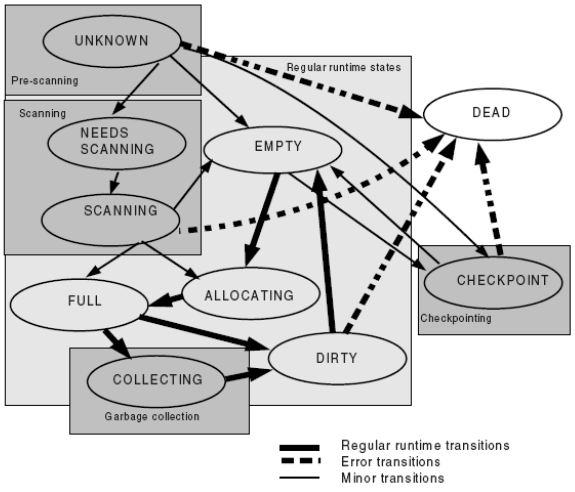
2.3.9.1 块和厚片的管理

2.3.9.1.1 块状态

Yaffs 会跟踪操作中的每个块和厚片的状态，这些信息首先是在扫描过程中（或从检查点恢复中）被构建的。表[yaffs-block-state]给出了一个块可能处于的各种状态。

表[yaffs-block-state] Yaffs 中块的各种状态

状态	说明
UNKNOWN	块的状态还是未知的
NEEDS_SCANNING	在预扫描过程中，已经确定这个块上面包含一些需要的东西，需要被扫描
SCANNING	这个块当前正在被扫描
EMPTY	这个块上面什么都没有（已经擦除过的），可以用来分配。当块被擦除过以后，就可以进入这种状态
ALLOCATING	这个块当前正被厚片分配器用来进行分配
FULL	这个块中的所有厚片都已经被分配，至少一个厚片还包含有用信息，并且没有被删除
DIRTY	块中的所有厚片，都已经被分配，所有都已经被删除。在这种状态之前，这个块可能已经在 FULL 或者 COLLECTING 状态。这个块现在可以被擦除，并且返回到 EMPTY 状态
CHECKPOINT	这个块中包含了检查点数据，当检查点失效的时候，这个块就可以被擦除,并且返回到空状态
COLLECTING	这个块正在被垃圾回收，只要所有有效数据已经被检查到，这个块就变成 DIRTY 状态
DEAD	这个块已经退休，被标记为坏块，这是一个块的最终状态



图[yaffs-block-state-transform] Yaffs 块的状态转换图

图[yaffs-block-state-transform]给出 Yaffs 块的状态转换图。常规的运行时状态是：EMPTY，ALLOCATING,FULL,COLLECTING,DIRTY。COLLECTING 状态是为了给块设置检查点。DEAD 状态是块的最终状态，或者损坏，或者发生了错误。

2.3.9.1.2 块和厚片的分配

所有可用的厚片，必须在重写之前被分配。厚片的分配机制是由 yaffs_AllocatedChunk() 提供的，非常简单。每个分区都有一个当前块用来进行分配，这个块被称为“分配块”。厚片是从分配块中顺序进行分配的。当厚片被分配的时候，块和厚片的管理信息会被更新。当块被分配完毕的时候，另外一个空块就会被选择成为分配块。

2.3.9.1.3 关于磨损均衡

闪存块只能执行有限次数的擦除操作，一个闪存文件系统需要确保不让少数块被过分擦除。这对一个文件系统（比如 FAT）来说是至关重要的，因为，这些文件系统只会用有限数量的逻辑块来存储文件分配表条目，而这些条目会被频繁地修改。如果 FAT 采用了静态的逻辑到物理块的映射，那么，用来存储文件分配表的块，会磨损得更频繁。因此，对于 FAT 而言，有一点是非常重要的，那就是，需要适当变化逻辑到物理的映射，从而使得文件分配表被写入到不同的物理块中，以此来达到均衡磨损的目的。

磨损均衡对于基于日志结构的文件系统而言，通常不是个问题。因为，基于日志结构的文件系统总是把日志记录顺序地追加写入到尾部，因此就不需要总是磨损同一个块。

通常有两种方式可以获得磨损均衡：

（1） 采用一个特定函数集合来执行磨损均衡；

（2） 把磨损均衡作为其他操作的副产品，即其他操作完成的过程也同时提供了一定程度的均衡磨损。

Yaffs 采用了第二种方案。首先，由于是基于日志的结构，Yaffs 通过顺序写入日志就已经内在地实现了磨损均衡。其次，块是从分区中已经擦除的块中顺序地分配的，因此，擦除的过程和分配块的过程，也提供了一定程度的磨损均衡。因此，在 Yaffs 中，即使没有代码来执行专门的磨损均衡策略，磨损均衡也是作为其他操作的副产品发生了。这种策略执行地很好，已经在模拟器和真实的 NAND 闪存上都表现出了较好的性能。

2.3.9.2 内部缓存

在实际应用场景中，一些应用会执行许多小数据量的读和写操作，比如一次读或写几个字节的操作。这种操作行为对于闪存性能而言会产生严重的负面影响，而且会影响闪存寿命。因为，闪存写操作是以页为单位，小数据量的写操作会浪费大量的闪存空间，导致闪存空间利用率很低，而且，小数据量的写操作也会增加垃圾回收过程的开销，因为，需要进行大量的擦除操作。

Yaffs 内部缓存的主要目的就是减少应用对 NAND 闪存的访问次数。这种缓存机制非常简单，主要是为那些不怎么复杂的、又缺乏自身文件缓存层的操作系统而设计的。

Yaffs 缓存被保存在一个缓存条目数组中，缓存条目的数量是在 yaffs_Device 配置中设置的，当缓存条目数量为 0 时，就意味着让缓存失效。缓存管理算法也很简单，很可能无法扩展到较大的数量，最好保持 5 到 20 个缓存条目。每个缓存条目保存了以下信息：（1）对象编号，厚片编号，被缓存的厚片的标签信息；（2）实际的数据类型；（3）缓存状态（计数器）。

在 Yaffs 缓存中找到一个可用的缓存条目是一个非常简单的操作。我们只需要遍历缓存条目，找到一个不忙的条目即可。如果所有的缓存条目都是忙的，那么就会执行一个缓存驱逐操作，把一个已有的缓存条目中的数据驱逐出缓冲区，腾出空间，Yaffs 使用 LRU 算法来选择 LRU 缓存条目。一旦一个缓存条目被访问，它的计数器就会增加 1，这个计数器用来给 LRU 缓存替换算法提供参考信息。已经被修改过的、并且还没有被写回闪存的缓存条目，

会被添加一个脏标记，从而告诉我们在刷新或驱逐缓存页的时候，是否需要把页写回到闪存中。这种缓存替换策略非常简单，却提供了很好的性能。

2.3.9.3 扫描

当挂载一个 Yaffs 分区时，需要通过扫描来建立状态信息，这个过程需要耗费一定的时间，因为需要从系统中所有有效的厚片中读取标签值。对于 Yaffs1 和 Yaffs2 而言，二者的扫描过程是不同的。

2.3.9.3.1 Yaffs1 扫描

Yaffs1 进行扫描工作时，会首先从分区中的所有厚片中读取标签值，然后据此判定厚片是否有效，如果没有设置删除标记位，那么它就是有效的，扫描过程中各个块的读取顺序是不重要的，可以采用任何顺序来读取块。对于任何有效的厚片，根据厚片的类型不同（可能是数据厚片或对象头部），可以采用以下方式把它添加到文件系统中：

（1） 如果厚片是一个数据厚片（厚片编号大于 0），那么，这个厚片必须被增加到相应的文件对象的 tnode 树中；

（2） 如果厚片是一个对象头部（厚片编号等于 0），那么，对象必须被创建；

（3） 如果在扫描过程中的任何时刻，发现正在读取具有相同的“对象编号:厚片编号”的另一个厚片，那么可以使用 2 位的序列号来处理冲突。

2.3.9.3.2 Yaffs2 扫描

Yaffs1 采用了删除标记位，因此，它的扫描过程相对 Yaffs2 而言就显得更加简单。但是，Yaffs2 没有删除标记位，这就使得扫描的过程更加复杂。Yaffs2 进行扫描工作时需要做的第一件事情是，对块进行预扫描，从而确定块的顺序号。块的链表会根据顺序号进行排序，形成一个按照时间顺序排列的链表。然后，Yaffs 会对这些块进行后向扫描（也就是反向时间顺序），因此，第一个遇到的“对象编号:厚片编号”配对，就是当前的厚片，后面的具有相同标签值的厚片都是过期的数据。

Yaffs 会读取每个块的每个厚片中的标签，如果它是一个数据厚片（即厚片编号大于 0），那么：

（1） 如果此前有一个具有相同的“对象编号:厚片编号”的厚片已经被找到，那么，现在这个厚片肯定不是当前数据，删除这个厚片；

（2） 否则，如果这个厚片被标记为删除或污染，就删除它；

（3） 否则，如果对象还不存在，那么这个厚片是从上次对象头部写入之后写入的，因此，这个厚片一定是文件中最后写入的厚片，这种情形一定是在一个“不彻底关闭（unclean shutdown）”之前发生的，而这时文件仍然是保持打开的。我们可以创建这个文件，然后使用厚片编号和厚片标签信息来设置文件内容；

（4） 否则，如果对象确实存在，厚片大小超过了对象文件的扫描长度，那么，它就是一个裁减后的厚片，可以被删除；

（5） 否则，把它放入到文件的 tnode 树中。

如果它是一个对象头部，即厚片编号等于 0，那么：

（1） 如果这个对象在被删除的目录里，那么就把这个对象标记为删除，并且把这个对象的缩减长度设置为 0；

（2） 否则，如果之前还没有发现这个对象的对象头部，那么，现在这个对象头部就是当前的对象头部，并且保留了当前的名称。如果之前还没有发现针对这个厚片的数据厚片，那么，在这个对象头部中的长度就是文件的长度，这个文件长度用来确定文件的扫描长度；

（3） 否则，如果文件长度比当前的扫描长度更小，就只利用文件长度作为扫描长度；

（4） 否则，这是一个废弃的对象头部，可以删除。

这里需要对文件扫描长度做更多的说明，它的目的是确定应该被忽略的数据厚片，这是由于文件裁减引起的。

2.3.9.3.3 检查点

挂载扫描需要耗费大量时间，也减慢了挂载的过程。检查点是一种加速挂载的机制，它通过获取 Yaffs 在卸载时的运行时状态的一个快照，然后在再次挂载时重新构建运行时状态。有了检查点机制以后，只有在以下情形中需要执行扫描工作：（1）挂载一个 Yaffs 分区时，如果不存在检查点数据；（2）挂载过程被告知忽略检查点数据。

实际的检查点机制非常简单。一系列的数据会被写入到一个块的集合，这个块集合被标记为专门用来保存检查点数据，重要的运行时状态会被写入到数据流。并非所有的状态都需要被保存，只有需要用来重构运行时数据结构的状态才需要被保存。例如，文件元数据就不需要保存，因为，很容易通过“延迟加载”进行加载。

检查点采用以下顺序存储数据：开始标记（包括检查点格式编号）、Yaffs_Device 设备信息、块信息、厚片标记、对象（包括文件结构）、结尾标记、校验和。

检查点的有效性是通过下面的机制来保证的：

（1） 采用任何可用的 ECC 机制来存储数据；

（2） 开始标记包含了一个检查点的版本信息，从而使得一个废弃的检查点（如果这个检查点代码发生了变化），不会被读取；

（3） 数据被写入到数据流，作为结构化的记录，每个记录都具有类型和大小；

（4） 当需要的时候，结束标记必须被读取；

（5） 在整个检查点数据集合中，需要维护两个校验和。

如果任何检查失败，检查点就会被忽略，状态必须被重新扫描。这里需要注意写检查点块时的块状态变化，那么，如何才能正确地重建块状态呢？这是通过下面的机制来实现的：

（1） 当写检查点的时候，会给检查点块的分配一个 EMPTY 块。当正在写检查点时，Yaffs 并不会改变块的状态，因此，它们被写入到检查点的状态为 EMPTY 或 CHECKPOINT，至于到底是哪个状态，其实并不是很重要；

（2） 在读取检查点之后，就可以知道哪个块已经被用来存储检查点数据，因此，可以更新这些块来反映 CHECKPOINT 状态。

任何修改（写或擦除）都可以让检查点失效，因此，任何修改路径都会检查是否存在一个检查点，如果存在就擦除它。常规的 Yaffs 块分配器和垃圾收集程序，也必须知道检查点的正确大小，从而确保有足够的空间可以保存检查点。

2.4 本章小结

本章内容首先对闪存文件系统和闪存转换层进行了特点比较，并指出了二者在工作方式上的差异；然后介绍了闪存文件系统 JFFS2，它是一个典型的基于日志结构的日志文件系统，在嵌入式 Linux 系统中得到了广泛的应用，帮助上层应用完成对闪存芯片的直接管理和控制；最后，介绍了闪存文件系统 Yaffs，详细阐述了它的体系架构、文件存储、垃圾回收、坏块和 NAND 错误处理、内存数据结构及其各种工作机制。

2.5 习题

1、 阐述闪存文件系统和闪存转换层的各自特点。

2、 说明闪存文件系统 JFFS2 的不足之处。

3、 说明闪存文件系统 Yaffs 的各种内存数据结构及其作用。

4、 阐述闪存文件系统 Yaffs 的垃圾回收过程。

5、 阐述闪存文件系统 Yaffs 的检查点机制的工作原理。

第 3 章 闪存转换层

闪存转换层（Flash Translation Layer），简称 FTL，位于文件系统与闪存之间，为文件系统提供了虚拟的磁盘，可以使得固态盘表现出类似磁盘的行为，从而使得面向磁盘开发的应用程序（比如数据库应用）不需要做任何修改，就可以直接运行在以固态盘作为存储介质的存储系统上。

理解 FTL 机制是设计出性能优良的闪存数据库的关键，根据是否采用 FTL 机制以及采用什么样的 FTL 机制，闪存数据库在设计上就会衍生出许多不同的技术路线，比如，设计面向 DBMS 的 FTL、采用 FTL 并修改面向磁盘的 DBMS 的部分模块和抛弃 FTL 并修改面向磁盘的 DBMS 的部分模块等等。因此，在介绍闪存数据库之前，有必要首先介绍 FTL 机制的概念和相关方法。

本章将详细介绍 FTL 的基本功能，并且重点阐述 FTL 的多种映射机制。

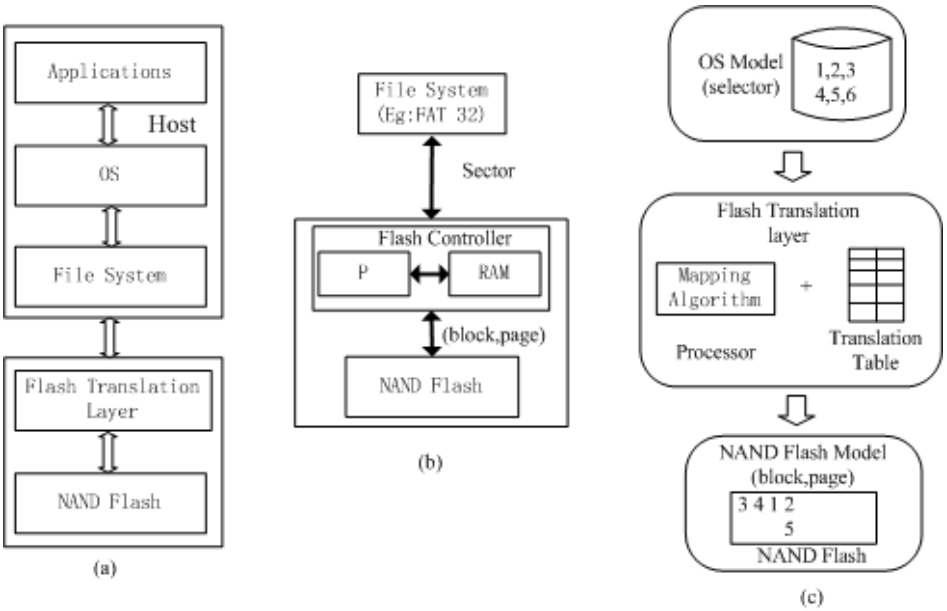
3.1 FTL的功能

FTL 的基本功能包括地址映射、垃圾回收、磨损均衡和断电恢复等，本节将分别给予简要介绍。

3.1.1 功能概述

NAND 闪存的读写特性和磁盘有很大区别，比如 NAND 闪存存在“写前擦除”的限定，闪存中擦除操作的粒度要比写操作的粒度大许多等等，那么，类似固态盘的闪存设备是如何做到表现出类似磁盘的行为的呢？这个功能是通过 FTL（Flash Translation Layer）实现的。

FTL 位于文件系统与闪存之间（如图[FTL](a)所示[ChoudhuriG07]），为文件系统提供了虚拟的磁盘，文件系统可以像使用磁盘一样来使用闪存。FTL 的实现方式主要有两种：（1）由宿主系统直接在闪存上以软件方式实现 FTL，比如存储棒（Memory Stick）就采用了这种方式，这种方式可以使得宿主系统能够根据应用特点优化存储管理策略，从而获得更好的性能；（2）对于许多基于闪存的设备而言（比如闪存固态盘），FTL 会被设计成一个控制器（如图[FTL](b)所示），封装在设备里面，控制器包含一个低端的处理器或微控制器以及少量的 SRAM。如图[FTL](c)所示，控制器主要负责完成两种功能：（1）把来自文件系统的读写请求，转换成针对闪存的读写请求，这里需要进行地址映射；（2）进行垃圾回收。



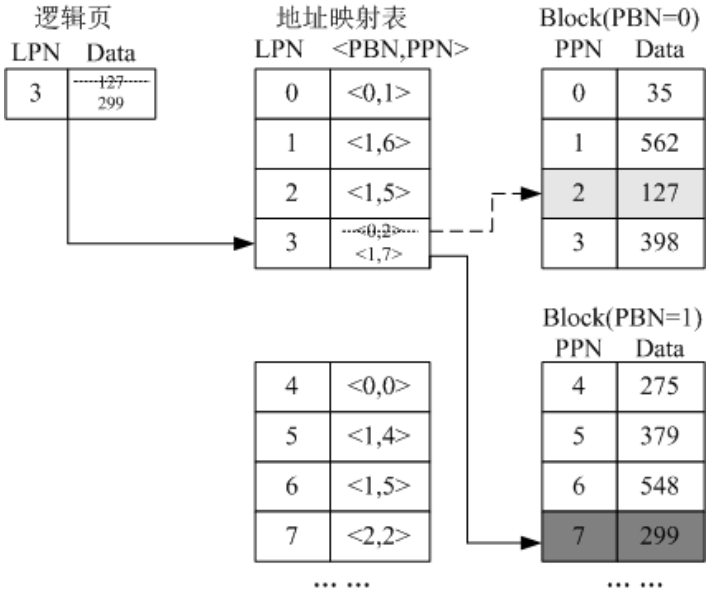
图[FTL] FTL 机制原理

具体而言，作为固态盘的最核心的组件，FTL 的主要功能是提供逻辑地址到物理地址的映射、断电恢复、垃圾回收和磨损均衡。FTL 机制通过提供上述功能，隐藏了闪存的特性，提供了一个虚拟的磁盘，使得上层应用不需要任何修改就可以直接使用基于闪存的存储设备。但是，这也增加了额外的代价，比如，FTL 为了对更新（重写）操作隐藏“写前擦除”的限定，提供了地址映射功能，这就需要消耗更多的存储空间来维护地址映射表，而且需要及时准备好闪存中可用的空闲块，一般而言，FTL 机制会在闪存中保留一定数量的日志块作为更新（重写）操作的临时存储。同样，垃圾回收和断电恢复机制也都涉及一定的代价。

3.1.2 地址映射

FTL 扮演的一个主要角色就是提供 LBA（逻辑块地址）到 PBA（物理块地址）的映射，这个功能是由 FTL 映射表来实现的。FTL 映射表需要维护两种类型的数据结构：一个是直接的映射（即从 LBA 到 PBA 的映射），另一个是反向映射，用来在失败恢复的时候重新构建得到直接映射。也就是说，直接映射是从逻辑块地址到物理块地址的映射，而反向映射则是物理块地址到逻辑块地址的映射。反向映射是存储在闪存上的，直接映射也是存储在闪存上，并且有一部分会被放入 RAM 内存用来提高地址映射的查找速度。如果访问请求所需要的那部分直接映射不在 RAM 中，就必须在需要的时候把它从闪存交换到 RAM 中。对于采用页级别映射的 FTL 机制而言，在地址映射表中，与一个逻辑页 s 对应的映射表条目 $T[s]$ 的值为$<b_0,p_0>$，其中，b_0 表示闪存中的物理块号，p_0 表示闪存中的物理页号，物理页 p_0 的 OOB 区域中会同时保存逻辑页 s 的逻辑页号和页面有效标志位。

图[FTL-address-mapping]给出了 FTL 的地址映射的一个实例。从图中可以看出，在页映射表中，左边一列表示了 LPN（Logical Page Number：逻辑页面编号），右边一列表示了<PBN,PPN>，其中，PBN（Physical Block Number）表示物理块编号，PPN(Physical Page Number)表示物理页编号。地址映射表中的一个映射条目{0,<0,1>}，表示 LPN 为 0 的逻辑页被映射到闪存中的第 0 块（PBN=0）中的第 1 页（PPN=1）。有一个逻辑页（LPN=3），原来的旧值是 127（图中用虚线划掉表示是旧值），原来的映射条目是{3,<0,2>}（图中用虚线划掉表示是旧的映射条目内容），表示该页被映射到闪存中的第 0 块第 2 页。当这个页（LPN=3）发生了更新以后，新的值是 299，闪存采用异地更新，因此，需要把新的值 299 写入到新的空闲页中，这里假设新值 299 被写入到第 1 块（PBN=1）第 7 页（LPN=7）中，然后去修改地址映射表，把 LPN=3 对应的映射条目，由{3,<0,2>}更新成为{3,<1,7>}。



图[FTL-address-mapping] FTL 的地址映射

每次设备重新启动时，都会扫描闪存物理块中的信息，读取 OOB 区域中的数据，重新构建 FTL 映射表。因为逻辑页会被不断更新，因此，逻辑页到物理页的映射不是固定不变的，也需要不断更新，这就导致映射表是不断变化的。当逻辑页 s 发生更新时，FTL 会把更新后的逻辑页写入到新的闪存物理页 p_1（假设 p_1 属于闪存物理块 b_1），并把映射条目 $T[s]$ 的值修改为$<b_1,p_1>$。FTL 接受各种读写数据请求以后，通过查找映射表，就可以把被请求数据的逻辑地址映射转换成这些数据在闪存中的物理存储地址。这样，FTL 就可以把写操作引导到闪存中那些已经擦除过的空闲页面，从而对上层应用隐藏了闪存的“写前擦除”的限制，也大大减少了写操作之前的擦除操作的次数，提升了闪存性能。地址映射的方式包括页级别映射、块级别映射和混合映射，每种映射方法都有不同的优缺点。为了加快逻辑地址到物理地址的映射，在系统运行过程中，FTL 映射表通常都会被读取到 SRAM 中。但是，FTL 映射表通常较大，往往无法一次性全部放入 SRAM。比如，对于 1GB 的 NAND 闪存而言，如果采用页级别映射，每个映射表条目为 8 字节，那么，FTL 映射表的大小就是 1GB/512B*8=16MB。

闪存设备每次启动时，都会扫描闪存物理块中的信息，读取 OOB 区域中的数据，重新构建 FTL 映射表。在早期闪存容量较小时，这种地址映射表的重构过程不会影响闪存设备的性能。但是，随着闪存容量的不断增大，扫描整个闪存空间的时间开销越来越大，从而使得地址映射表的重构过程时间越来越长。为了加速闪存设备的启动过程，可以采用一些改进策略，主要包括[Xiang09]：

（1）快照技术：一旦系统关闭，就把 SRAM 中的地址映射信息的快照写入闪存，为了能够在系统启动时能够迅速读取快照的内容来重构地址映射表，还需要在闪存中专门开辟一个区域来存储快照的存储地址；当系统正常关闭时，这种方式可以明显提高系统重启的速度，但是，当系统发生故障后重新启动时，为了保证地址映射表的正确构建，仍然需要重新扫描整个闪存物理空间。

（2）日志技术：基于日志的技术可以提高系统故障时的地址映射表重构速度，在该技术中，任何针对 SRAM 中的地址映射信息的修改，都会导致相应地生成日志记录来记录这些修改，当日志记录的数量累积到一定程度的时候，就可以批量写入到闪存的专门区域。当系统正常关闭后启动时，就可以直接到闪存中为日志存储专门开辟的区域去读取日志信息，完成地址映射表的重构。当系统发生故障后重启时，地址映射表的重构需要扫描两个方面的内容，即日志区域和一部分闪存物理空间。基于日志的技术，可以有效提高系统发生故障后的重启速度，因为，它不需要扫描整个闪存物理空间，而只需要扫描一部分闪存物理空间和日志区域。不过需要指出的是，虽然只需要扫描一部分闪存物理空间，但是，扫描的工作量仍然和闪存中的数据量成正比，当数据量较大时，扫描的时间开销也会相应增加。

这里还需要进一步指出的是，把闪存存储设备上的块以线性方式进行映射，可能会带来副作用，即导致不同闪存块之间的不均衡磨损。某些块会被较多地执行擦除操作，另一些块则可能会被较少地执行擦除操作。被频繁擦写的块，会很快到达擦除次数上限，成为坏块，导致整个闪存设备访问速度较慢或者寿命缩短。因此，必须设计相应的“磨损均衡”策略。

3.1.3 垃圾回收

闪存采用“异地更新”的方式，当对某个闪存页面进行更新时，只需要把新数据写入到一个新的“空闲块”（已经擦除过的块）中，并通过更新元数据，把原来的物理页面设置为“无效”。这样，新数据可以快速写入到新块中，不会涉及代价高昂的擦除操作。但是，闪存的空间是有限的，可用的空闲块的数量也是有限的，随着系统的运行，闪存中的无效页的数量会越来越多，空闲页就变得越来越少，当空闲块消耗到一定程序时，FTL 就必须启动垃圾回收过程，回收那些处于无效状态的页，对一些块执行擦除操作从而生成可用的空闲块。垃圾回收结束后，需要在适当的时机更新 FTL 映射表。

如果一个块只包含有效页，则垃圾回收过程可以不用考虑这个块。垃圾回收过程主要针对两种情形：第一，块中全部是无效页；第二，块中有一部分是有效页，有一部分是无效页。如果块中只包含无效页，那么，只要简单执行擦除操作即可，实时上，由于异地更新，上面的第二种情形更加常见。也就是说，如果在页上执行更新，那么，在任何时间点，闪存的擦除块往往会同时包含有效页和无效页。如果一个擦除块包含了有效页，那么，执行擦除操作可以有两种选择：（1）把擦除块 *b* 中的有效页首先拷贝到内存中，对该块 *b* 执行擦除操作，然后再把内存中的有效页回写到已经擦除过的这个块 *b* 中；（2）把擦除块 *b* 中的有效页直接拷贝到其他具有空闲页的块中，然后对块 *b* 执行擦除操作。

对于第一种选择，如果擦除块 *b* 中的有效页被拷贝到内存中、并且已经对块 b 执行了擦除操作以后，在把内存中的这些有效页再次拷贝回块 *b* 之前，发生了系统故障（比如断电），那么，这些有效页就会丢失，无法恢复。因此，为了保证数据库的一致性，通常会在闪存数据库中使用第二种选择。由于第二种选择需要额外的空闲块来放置有效页，因此，执行垃圾回收操作时，必须保证闪存中还有一定数量的空闲块。

对于第二种选择，在擦除操作执行之前，必须把这些有效页拷贝到闪存的其他位置。对于一个轻度负载的设备而言，这种额外的拷贝开销可以忽略不计。但是，在一个过载的设备上，需要大量的被擦除过的可用擦除块，这时这种开销就必须被考虑进来。对于擦除块的回收而言，一个最好的情形是：在擦除时刻，擦除单元中的所有页都是无效的。当数据访问模式具有高度局部性时，这种情况有可能发生，比如，当一个文件被一页一页地顺序更新。在这种情况下，每个页写（page-write）操作，会引起大约 *P*/*E* 个擦除操作，其中，*P* 是页的大小，*E* 是擦除单元的大小。对于擦除单元回收而言，一个最坏的情形是：所有待回收的擦除单元中都只包含一个无效页。当设备快满、数据访问模式分散在不同的擦除单元中时，就有可能发生这种情况。在这种情况下，几乎每个页写操作，都会引起一个擦除操作。在最好和最差的情形之间，还存在许多中间情况，因此，如果仅仅给定页写操作的频率，是无法准确判断擦除操作的频率的。这时，就需要同时考虑更多的关于工作负载的特性信息。

3.1.4 磨损均衡

一个闪存块被擦除的次数是有限制的，通常可以擦除 1 万到 10 万次[AgrawalPWDMP08]，一旦超过这个限定次数，闪存就会变得不稳定甚至损坏，会引起频繁的数据读写错误。一般而言，一旦某个块的擦除次数达到预定的门槛，厂商为了保证产品存储数据的安全性，都会把该块设置为“无效”，即使这个块还可以继续使用。因此，为了最大程度地延长固态盘的寿命，应该尽量延迟第一个损坏块的到来时间，这就需要有效的磨损均衡策略[GalT05][JungCJKL07]，即让擦除操作均匀地分布在不同的闪存块上，而不让某一部分块被过多地执行擦除操作。从设备的寿命来说，如果特定的算法减少了擦除次数 *f* 倍，设备的预期寿命在理论上就会增加 *f* 倍。典型的磨损均衡策略主要包括[Xiang09]：（1）基于阈值的控制方法[LofgrenNT03]：当块之间的擦除次数差距超过事先设定的阈值时，就启动回收过程，收回擦除次数最少的闪存块，从而实现更好的磨损均衡，这种做法的代价是，可能会在一定程度上降低垃圾回收的效率；（2）移动数据页的方法[KimL99]：在各擦除块之间周期性地移动数据页，避免一些具有较低更新频率的块的擦除次数过低，从而使得不同块之间具有比较均衡的擦除次数；（3）基于双队列的损耗均匀控制方法[Chang07]：当某个块的擦除次数过多时，就把冷数据（很少被访问的数据）存储在该块中，由于冷数据的访问频率很低，因此，就可以降低该块在未来一段时间内的擦除频率。

3.1.5 断电恢复

一方面，固态盘中通常都会配置少量易失的 SRAM，用来存储地址映射表，加快逻辑地址到物理地址的转换过程，提高请求响应速度。但是，当发生断电的时候，SRAM 中的

信息会立即丢失。另一方面，闪存存在擦除操作，这不仅降低了闪存性能，还可能引起数据一致性问题。因为，在对某个块执行擦除操作时，需要把该块中有效的数据复制出来，然后写入到其他新块中，但是，如果这个过程发生断电，就有可能发生数据丢失，对于一些掌上电子产品而言，发生这种情形的概率并不小。此外，当闪存块的大小和系统所采用的块大小不同时，断电故障也可能出现副作用。假设闪存块大小为 128KB，存储系统使用 4KB 大小的块，为了把 4KB 的数据更新到 128KB 的闪存块中，就需要首先把闪存块中的 128KB 数据全部拷贝到内存中，在内存中完成 4KB 数据的更新，然后，再把更新后的 128KB 数据写入到闪存。如果在内存更新的过程中发生了断电，那么，内存中的数据就会全部丢失。因此，FTL 必须设计相应的断电恢复机制。

3.2 FTL的映射机制

固态盘的硬件设计细节（尤其是 FTL 机制），通常是生产商的商业机密，不会对外公开。但是，在学术界，研究人员已经提出了许多 FTL 机制[ChungPPLLS06][LeePCLPS07] [KimKNMC02] [KangJKL06]，其中，地址映射方法是研究的核心内容。

根据映射单元的粒度，现有的 FTL 机制可以分成四大类：页级别、块级别、混合和变长映射：

- 页级别的 FTL 机制：比如 LazyFTL[MaFL11]、DFTL[GuptaKU09]；
- 块级别的 FTL 机制：比如文献[ChoudhuriG07]中的方法；
- 混合 FTL 机制：比如 LAST[LeeSKK08]、BAST[KimKNMC02] 、A-SAST (Adaptive SAST) [KooS09] 、 HFTL[LeeYL09] 、 SAST[LeePCLPS07] 、 SuperBlock FTL[KangJKL06]和文献 [ChungPPLLS06][KangJKL06]中的方法；
- 变长映射机制：比如文献[ChangK04]中的方法。

3.2.1 页级别的 FTL 机制

本节介绍页级别的 FTL 机制的基本原理，并介绍典型的相关研究。

3.2.1.1 机制概述

在采用页级别的 FTL 机制[KawaguchiNM95]中，请求的逻辑页面可以被映射到闪存空间中的任何物理页面，因此，这种机制非常灵活，而且具有很高的闪存页面利用率。图[FTL-page-mapping]给出了页级别映射示意图，为了简化问题描述，这里省略了块号，实际上，当每个块中所包含的页数确定以后，可以很容易根据页号计算出块号，比如，如果每个块包含 4 个页，那么，逻辑页 0,1,2,3 就在第 0 块中，逻辑页 4,5,6,7 就在第 2 块中，逻辑页 *i* 就在第 *i*/4 块中，其中，*i* 表示页号，*i*/4 表示整除操作。从图中可以看出，对于一个逻辑页 *i*（即逻辑页号 LPN 为 *i*）而言，可以在页映射表中找到与该页对应的映射条目，映射条目<逻辑页号 *i*,逻辑页号 *k*>就表示逻辑页 *i* 被映射到物理页 *k* 上。

图[FTL-page-mapping] 页级别映射示意图

对于包含很多小数据量的写操作而言，这种细粒度的地址映射方法是灵活高效的，但是，

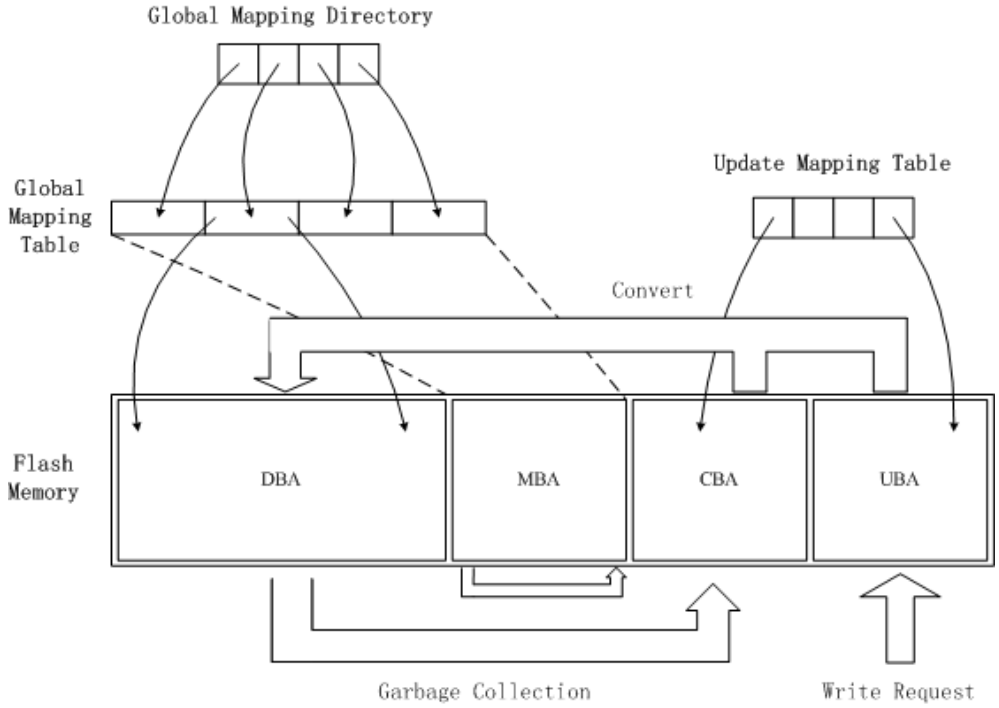
它也存在明显的缺陷，就是页面映射表会很大，导致映射表无法全部放入 SRAM。例如，一个 16GB 的闪存需要大约 32MB 的 SRAM 空间，来存储一个页级别的映射表。在固态盘产品中，通常会设计一个较小容量的 SRAM，用来存储 FTL 的映射表，SRAM 的访问速度要比闪存快许多，把映射表存放到 SRAM 中可以大大提高页面映射过程的速度，从而提高固态盘的读写速度。但是，SRAM 的价格要比闪存贵一个数量级，因此，固态盘中配备的 SRAM 的容量通常很有限。而闪存的容量正随着技术进步不断增加，让固态盘中的 SRAM 的容量也随着闪存容量的增加而保持线性增加，是不经济也不现实的。

一旦页面映射表太大，无法一次性全部放入 SRAM，就会严重影响固态盘的读写速度。因为，只有一部分映射表条目可以放入 SRAM，其他映射表条目必须保存到闪存空间中。当来自系统的数据读取请求到达时，如果被请求的逻辑页面编号恰好已经在 SRAM 的映射表中，就可以直接根据逻辑页面到物理页面的映射，找到数据在闪存中的物理存储位置后读取数据；但是，如果被请求的逻辑页面编号不在 SRAM 的映射表中，FTL 机制就必须到闪存空间中找到相应的映射表条目，读取后获得逻辑页面对应的物理页面地址，然后去该物理地址读取数据。为了获得最好的性能，可能还需要采用合适的缓存替换策略，来决定是否用刚才访问过的映射表条目来替换 SRAM 中的映射表条目，这就带来了额外的开销。

3.2.1.2 典型研究

第一个页级别 FTL 机制是由 Ban[Ban95]在 1995 年提出的，同时申请了专利，并且在几年后被 PCMCIA 采纳为 NOR 闪存的标准[Intel98]。这里需要指出的是，目前的很多 FTL 机制都被申请了专利，不仅美国如此，在其他大多数欧洲和亚洲国家也都是如此。但是，Ban 提出的页级别 FTL 机制，是针对 NOR 闪存的，无法直接应用到 NAND 闪存上，因为，NOR 闪存是以字节为单位进行读写操作，而 NAND 闪存则只能以页为单位进行读写操作。

对于一个页级别的映射机制而言，它是很难在 NAND 类型的闪存上布置的，因为，它们只能以页为单位进行读写操作。只要映射表的任何部分发生修改，就需要被立即写入到闪存，这会严重影响性能。为了降低对性能的影响，一种可能的处理方式是可以考虑把脏数据暂时保存在 SRAM 中，只有当它将被交换出去的时候，才把它写入闪存。但是，这样做就会带来丢失关键信息的风险，可能会使系统处于不一致的状态，因为，SRAM 是易失性存储器，一旦断电就会丢失全部信息。为了解决这个问题，文献[MaFL11]提出了 LazyFTL，这种机制会在闪存中保留两个小区域，并且采用惰性方式对页级别的映射表进行更新，即经过一段时间满足特定条件时才进行更新，这样就有效减少了更新映射表的次数，降低了对性能的影响。图[LazyFTL]显示了 LazyFTL 机制的体系结构，从中可以看出，LazyFTL 把整个闪存划分成四个区域：数据块区域（DBA）、映射块区域（MBA）、冷块区域（CBA）和更新块区域（UBA）。除了 MBA 以外，所有分区都存放用户数据。



图[LazyFTL] LazyFTL 机制

在 LazyFTL 机制中，DBA 中的页面是通过页级别的映射表进行跟踪的，整个映射表被称为全局映射表 GMT（Global Mapping Table）。GMT 是以页的方式组织，并且存储在 MBA 中。在 SRAM 中会开辟一块小的缓存，它采用 LRU 或者类似的算法进行缓存管理，从而可以为 GMT 中被频繁访问的部分提供快速引用。SRAM 中存储的另外一个表被称为全局映射目录 GMD（Global Mapping Directory），它记录了 GMT 的所有有效映射页的物理地址。CBA 被用来容纳冷块，UBA 被用来容纳更新块。

LazyFTL 和原来的页级别的 FTL 机制[Ban95]的主要区别在于，LazyFTL 保留了两个小分区，即 CBA 和 UBA，用来延迟对 GMT 的更新操作，这些更新操作是由写请求和无效页移动导致的。和整个闪存的容量比起来，CBA 和 UBA 是很小的。此外，LazyFTL 会在 CBA 和 UBA 这两个区域之上，构建另外一个页级别的映射表——更新映射表 UMT（Updating Mapping Table）。UMT 可以被实现成一个哈希表，或者一个二叉搜索树，从而支持高效的插入、删除和修改。UMT 中的条目的数量是很小的，因此，这些操作不会带来多少额外的开销。由于 UMT 是保存在 SRAM 中的，因此，CBA 和 UBA 不能太大，这就意味着，CBA 和 UBA 总会出现写满的情形，这时就需要执行一个转换操作，在所有 CBA 或者 UBA 中已经填满的块中，选择一个转换代价最小的块作为牺牲品，然后把它转换成正常的 DBA。

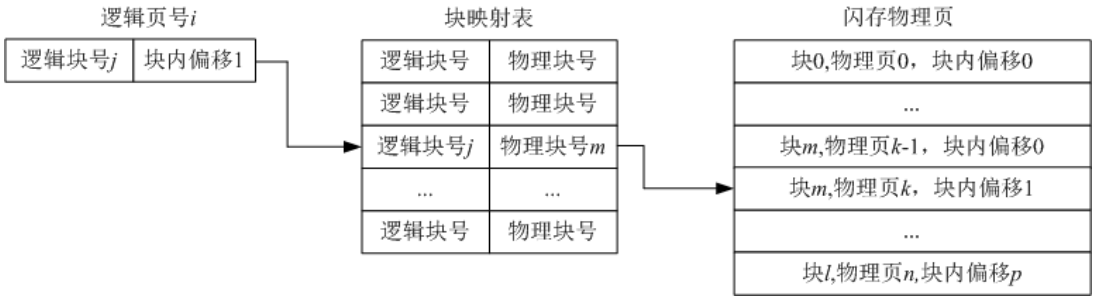
UBA 中的一个块，被称为当前更新块 CUB，是用来处理写请求的。当一个 CUB 被写满时，另外一个空闲块会被分配成为新的 CUB。类似地，在 CBA 中，会有一个当前的冷块 CCB，用来处理移动的数据页。事实上，LazyFTL 把 CBA 中填满的冷块和 UBA 中填满的更新块这二者进行同等对待和处理。换句话说，CBA 和 UBA 的相对大小是可以进行动态调节的。如果热数据的比例上升，UBA 就会扩展。如果空间利用率增加，CCB 就会比 CUB 更快地被填满，这样，CBA 就会扩大。通过这种方式，LazyFTL 就可以根据访问机制的变化而动态自我调整。

3.2.2 块级别的 FTL 机制

3.2.2.1 机制概述

对于块级别的 FTL 而言，一个逻辑页面编号通常首先被分解成一个逻辑块编号和一个

块内偏移量，然后，只需要把这个逻辑块编号转换成一个物理块编号，块内偏移量是不变的，最后，采用一些搜索算法来寻找目标页。图[FTL-block-mapping]给出块级别地址映射示意图，对于一个逻辑页 *i* 而言，首先需要计算得到该页的逻辑块号 *j* 和块内偏移量（假设为 1），然后，到块映射表中，找到相应的映射条目<逻辑块号 *j*，物理块号 *m*>，表示逻辑块 *j* 被映射到物理块 *m* 上，接着到物理块 *m* 中找到块内偏移量为 1 的物理页 *k*，因此，逻辑页 *i* 最终被映射到物理页 *k* 上。



图[FTL-block-mapping] 块级别地址映射示意图

可以看出，这种方法和传统的页式虚拟内存中的地址映射机制[SilberschatzC98]是类似的。块级别映射机制的优点是，映射表的规模要比页级别的 FTL 机制小很多，很容易放入到 SRAM 中，比如一个 16GB 的闪存只需要大约 512KB 大小的映射表；此外，小规模的映射表也降低了转换代价和能耗，很适合应用在掌上电子产品中。此外，块级别映射表的大小，不会像页级别映射表那样随着闪存容量的增加而线性增加，因为，更高存储容量的闪存一般也会采用更大尺寸的块。但是，由于块内的逻辑页码偏移量是固定的，因此，对于一个给定的逻辑页，它在物理块中只能位于具有相同偏移量的、某个特定的物理页内，这就会导致为一个逻辑页在物理块中找到合适的物理页的概率降低，由此将产生两个问题：（1）空间浪费；（2）垃圾回收的代价增加[GuptaKU09]。此外，这种机制的读、修改和写操作的开销都很大，即使只是对块的一部分进行更新操作的时候，也需要很大的代价。比如，假设在原来的地址映射表中，有一个逻辑块 *L* 被映射到闪存中的物理块 P_A 上面，一个写操作打算更新逻辑块 *L* 中的某个页 *p*，并假设页 *p* 在块 *L* 内的块内偏移量是 *d*。为了完成这个更新操作，FTL 机制会更改逻辑块到物理块的映射，把逻辑块 *L* 重新映射到闪存中新的空闲物理块 P_B 上面，然后，在物理块 P_B 中找到具有相同的块内偏移量 *d* 的一个页 *q*，在页 *q* 上面执行更新操作。与此同时，块 P_A 中其他页都要被复制到这个新的块 P_B 中。可以看出，块级别的 FTL 机制的更新操作代价是较大的。因此，和采用其他映射方法的 FTL 机制相比，块级别 FTL 机制的性能是很有限的。

总的来说，块级别映射大大减少了映射表的大小，但是，它的挑战在于如何最小化在一个块内部寻找一个页的开销。因此，块映射不支持随机写操作，也不能支持高效的更新操作。

3.2.2.2 相关研究

为了消除块级别 FTL 机制中更新操作的昂贵代价，文献[Ban95][MTD]提出了 NFTL 机制，该机制采用了替换块的策略，采用 NFTL 这个名称，是因为这种 FTL 机制是为 NAND 类型的闪存设计的。Ban 在 1999 年为两种块级别的 NFTL 机制申请了专利[ChoudhuriG07]，即 NFTL-1 和 NFTL-N。 NFTL-1 要求每个闪存页都存在 OOB 区域，而 NFTL-N 则是针对那些没有 OOB 区域的闪存。

3.2.2.2.1 NFTL-1 机制

在 NFTL-1 机制中，一个逻辑页号（LPN）会被分成两个部分：逻辑块号（LBN）和块内页偏移量，LBN 和 LPN 都从 0 开始编号，块内第 1 个页的偏移量为 0。比如，假设闪存

有 4 个块，每个块包含 4 个页。对于 LPN=5 的页而言，它的 LBN 为 1，块内页偏移量为 1。

当一个逻辑块第一次执行写操作时，系统会为这个逻辑块第一次被分配一个物理块，这个物理块被称为“主块”。逻辑块中的每个逻辑页的第一次写操作，就在与该逻辑块对应的主块中执行，逻辑页会被写入到主块相应的位置，即要求一个逻辑页在逻辑块中的块内偏移量应该和它对应的物理页在物理块中的块内偏移量相同。当一个页被重写（overwritten）的时候，NFTL-1 首先为相关的逻辑块分配一个替换块（如果还没有分配到替换块的话），然后从替换块的第一个可用空闲页位置开始，按照顺序写入重写页。由于页面是以异地（out-of-place）方式写入替换块，因此，NFTL-1 需要在替换块中以反向的顺序查找所有 OOB 区域，从而找到一个被请求页的最新版本。需要指出的是，在 NAND 闪存中，OOB 区域采用了不同的寻址算法，它专门为快速引用进行了优化处理，因此，这个搜索过程的开销较低。

这里给出一个实例解释 NFTL-1 机制的工作过程（如图[NFTL-1]所示）。假设闪存有 4 个块，每个块包含 4 个页。现在有一个操作序列 op_1,op_2,op_3,op_4,op_5,op_6，每个操作的格式为“操作类型+LPN(逻辑页号)+数据”。比如，op_1 表示把数据 A 写入到 LPN=5 的页面中。这个操作序列的执行过程如下：

（1）操作 op_1 首先到达，由于每个块的大小为 4，因此，LPN=5 的页所属的块的 LBN(逻辑块号)为 1。系统查找映射表，发现不存在 LBN=1 对应的映射条目，因此，立即为 LBN=1 的逻辑块分配一个物理块，物理块的 PBN（物理块编号）为 b_0，并把该条映射写入到映射表。物理块 $b0$ 是逻辑块 1 的“主块”，op_1 的写操作针对的页的 LPN 是 5，该页在逻辑块内的页面偏移量是 1（即块中的第 2 个页），NFTL-1 要求把数据写入到主块中具有相同块内偏移量的物理页上，因此，op_1 的数据 A 会被写入到物理块 b_0 的第 2 个页中，并在该页的 OOB 区域中写入 LPN 的值（即 5），同时把页面有效标志位设置为“有效”。

（2）当 op_2 到达以后，系统根据 LPN=5 计算得到其所属的逻辑块的 LBN 是 1，查找映射表，发现已经存在 LBN=1 对应的映射条目，读取映射条目，找到 LBN=1 的逻辑块对应的物理块为 b_0，然后，根据块内偏移量，找到物理块 b_0 内的第 2 个页，读取该页的 OOB 区域，发现该页的有效标志位已经被设置，说明已经被写入数据，由此可以判定 op_2 操作是针对同一个页的更新操作。这时，系统会为 LBN=1 的逻辑块分配一个物理块 b_1，这个物理块被称为“替换块”，主块 b_0 中会记录与之关联的替换块 b_1 的信息；然后，系统把 op_2 操作的数据 A_1 写入到替换块 b_1 的第一个空闲页上，由于这时块 b_1 的所有页都为空闲页，所以数据 A_1 会被写入到块 b_1 中的第 1 个页（该页在块 b_1 内的块内偏移量为 0）。

（3）依此类推，op_3 到达以后，由于 NFTL-1 要求把数据写入到主块中具有相同块内偏移量的物理页上，因此会把数据 B 写入到主块 b_0 的第 4 个页中。op_4 的数据 A_2 会被写入到替换块 b_1 的第一个空闲页上，由于块 b_1 的第 1 个页此前已经被写入数据 A_1，因此，数据 A_2 会被写入到替换块 b_1 中的第 2 个页。同理，op_5 的数据 B_1 会被写入到替换块 b_1 中的第 3 个页。

（4）当读取操作 op_6 到达后，它要读取 LPN=7 的页，该页对应的逻辑块的 LBN 为 1，查找映射表，可以得到对应的主块的 PBN 为 b_0，根据主块 b_0 中记录的信息，可以找到其对应的替换块的 PBN 为 b_1，系统会在替换块 b_1 中反向搜索每个页的 OOB 区域，检查其中记录的 LPN 的值是否为 7，块 b_1 中第一个被找到的页中就包含了 LPN=7 的页的最新版本的数据。如果块 b_1 中没有找到 LPN=7 的页，系统就到主块 b_0 中寻找并读取数据。这里在块 b_1 中正好可以找到 LPN=7 的页的最新版本的数据，即块 b_1 中的第 3 个页，因此，从该页中读出数据 B_1。

图[NFTL-1] NFTL-1 机制的一个实例

3.2.2.2.2 NFTL-N 机制

NFTL-N 是针对那些没有 OOB 区域的闪存而设计的。在 NFTL-N 机制中，映射表负责执行从逻辑块地址到物理块地址的映射。当一个逻辑块第一次被分配一个物理块时，这个物理块被称为“主块”。当以后再对这个逻辑块的某个页执行更新操作时，该 FTL 机制会为相应的逻辑块分配一个新的物理块，这个新的物理块被称为“替换块”，然后在替换块的相应位置写入该页，即页在逻辑块中的块内偏移量应该和它在物理块中的块内偏移量相同。每个主块都会有一个替换块，主块上的更新都被写入到替换块上。由于一个页会被多次更新，因此，对于一个替换块而言，它自身也会有相应的替换块。

为了方便确定最新版本的数据和可用的空闲页，NFTL-N 机制会为所有属于同一个逻辑块的替换块建立一个链表。对于读操作而言，通过在链表中进行查找，就可以找到最近一次更新的页；对于写操作而言，可以在链表中找到具有相同的块内偏移量的第一个空闲页。对于包含频繁更新操作的负载而言，该 FTL 机制会频繁地生成相应的替换块，而闪存的空间是有限的，可能会很快被消耗殆尽。因此，当更新操作找不到可用的空闲页时，就会执行一个合并操作，回收那些没有包含最新版本数据的页，具体方法是：找到所有链表中最长的链表，把所有最近更新的页从各个替换块中复制到链表中的最后一个替换块中，这个替换块就成为原来的逻辑块所对应的新的物理块。这个合并操作过程完成以后，原来逻辑块对应的旧的物理块和链表中的替换块（除了最后一个替换块），都会被擦除，成为可用的空闲块。

这里给出一个实例解释 NFTL-N 机制的工作过程（如图[NFTL-N]所示）。仍然假设闪存有 4 个块，每个块包含 4 个页。操作序列 op_1,op_2,op_3,op_4,op_5,op_6 的含义与上面的 NFTL-1 的实例完全相同。这个操作序列的执行过程如下：

（1）操作 op_1 首先到达， LPN=5 的页所属的块的 LBN 为 1。系统查找映射表，发现不存在 LBN=1 对应的映射条目，因此，立即为 LBN=1 的逻辑块分配一个物理块，物理块的 PBN（物理块编号）为 b_0，并把该条映射写入到映射表，同时，为 LBN=1 的逻辑块建立一个“块链表”，把主块 b_0 增加到块链表中。物理块 b_0 是逻辑块 1 的“主块”，op_1 的写操作针对的页的 LPN 是 5，该页在逻辑块内的页面偏移量是 1（即块中的第 2 个页），NFTL-1 要求把数据写入到物理块中具有相同块内偏移量的物理页上，因此，op_1 的数据 A 会被写入到物理块 b_0 的第 2 个页中

（2）然后，操作 op_2 到达，系统根据 LPN=5 计算得到其所属的逻辑块的 LBN=1，查找映射表，发现已经存在 LBN=1 对应的映射条目，读取映射条目，找到 LBN=1 的逻辑块对应的块链表，链表中找到第一个块为 b_0，然后，根据块内偏移量，找到物理块 b_0 内的第 2 个页，发现该页已经被写入数据，由此可以判定 op_2 操作是针对同一个页的更新操作。这时，系统会为 LBN=1 的逻辑块再分配一个物理块 b_1，这个物理块被称为块 b_0 的“替换块”，并把替换块 b_1 添加到为 LBN=1 的逻辑块建立的“块链表”中，即添加到链表中块 b_0 的后面；然后，系统把 op_2 操作的数据 A_1 写入到替换块 b_1 中具有相同块内偏移量的页上，即写入到块 b_1 中的第 2 个页。

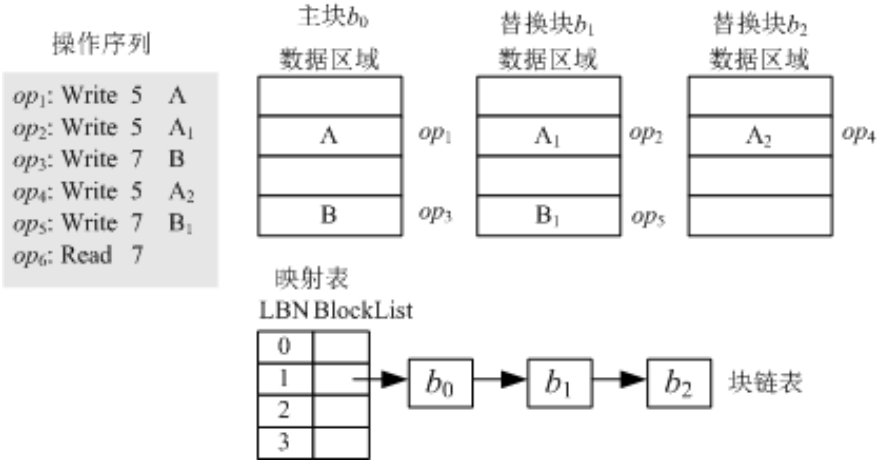
（3）接着，操作 op_3 到达，是针对 LPN=7 的页的写入操作，该页对应的 LBN 为 1。根

据映射表，找到 LBN=1 的逻辑块对应的块链表，链表中找到第一个块为为 b_0，块 b_0 中与 LPN=7 的页具有相同块内偏移量的页是块 b_0 的第 4 个页，该页仍然处于空闲状态，没有写入数据，因此把数据 B 写入块 b_0 的第 4 个页中。

（4）然后，操作 op_4 到达，LPN=5 的页所属的块的 LBN 为 1，查找映射表，找到 LBN=1 的逻辑块对应的块链表，链表中找到第一个块为 b_0，但是，块 b_0 中具有相同块内偏移量的页（即第 2 个页），已经写入数据，因此，继续沿着链表查找到第二个块 b_1，但是，块 b_1 中具有相同块内偏移量的页（即第 2 个页），也已经写入数据，而这时已经搜索到链表的尾部，已经找不到其他可用的块，因此，系统会再新分配一个替换块 b_2，加入到链表中，即放在链表中块 b_1 的后面，然后，在块 b_2 中具有相同块内偏移量的页（即第 2 个页）中，写入数据 A_2。

（5）当操作 op_5 到达，查找到与 LPN=7 的页对应的块链表，链表中找到第一个块为 b_0，块 b_0 中与 LPN=7 的页具有相同块内偏移量的页是块 b_0 的第 4 个页，但是，已经写入数据，因此，继续沿着链表寻找到第二个块 b_1，块 b_1 中与 LPN=7 的页具有相同块内偏移量的页是块 b_1 的第 4 个页，该页仍然处于空闲状态，没有写入数据，因此，把数据 B_1 写入块 b_1 的第 4 个页中。

（6）最后，当操作 op_6 到达时，查找到与 LPN=7 的页对应的块链表，然后，从链表的尾部 b_2 开始，在链表中反向搜索，寻找第一个具有相同块内偏移量并且不为空的页。可以看出，块 b_2 中找不到满足该条件的页，块 b_1 中正好可以找到满足该条件的页，即块 b_1 中的第 4 个页，这个页就是 LPN=7 的页的最新版本的数据，因此，从该页中读取出数据 B_1。



图[NFTL-N] NFTL-N 机制的一个实例

3.2.2.2.3 针对 NFTL-1 的改进

文献[ChoudhuriG07]的研究发现，在 NFTL-1 机制中，有两类重要的“元数据”信息，会被频繁地访问。第一，要频繁地检查一个给定的页是否包含有效的数据，这是一个布尔型信息，存放在闪存页的 OOB 区域中，因此，如果把反映一个页是否有效的状态信息，从 OOB 区域中复制出来放入到 RAM 中，就可以避免大量的 OOB 数据读取。第二，要确认与一个主块对应的替换块的信息。RAM 中的数据读写速度，要明远远高于闪存中的数据读写速度。

从闪存页的 OOB 区域中读写数据的开销，虽然要比从闪存的数据区域中读写数据的开销要低很多，但是，在频繁访问的情况下，OOB 数据读取的累积代价是不能被忽略的。表[OOB]显示了两种闪存芯片的各种性能参数，可以看出，OOB 读写代价是一个需要考虑的因素，它会影响 NFTL-1 机制的整体性能。

表[OOB] 两款 NAND 闪存产品的特性参数

Characteristics	Toshiba 16MB	Samsung 16MB
Block size	16384（bytes）	16384（bytes）
Page size	512（bytes）	512（bytes）
OOB size	16（bytes）	16（bytes）
Read Page	52（usec）	36（usec）
Read OOB	26（usec）	10（usec）
Write Page	200（usec）	200（usec）
Write OOB	200（usec）	200（usec）
Erasc	2000（usec）	2000（usec）

因此，把这些频繁被访问的信息放入到 RAM 中，可以有效提高 FTL 机制的整体性能。为了降低对上述两类元数据信息频繁访问带来的开销，作者引入了两种技术方案，即查找表和页缓存：

- 查找表：维护一个 RAM 中的数据结构，可以实现快速的元数据信息查找；
- 页缓存：充分利用时间局部性，缓存最近访问的页。

3.2.2.2.3.1 查找表

查找表技术中，采用了两种类型的 RAM 数据结构：替换块表和页状态位图。其中，替换块表提供了对替换块信息的快速访问，可以加快为一个指定的主块寻找对应的替换块的过程，而页状态位图则可以看成是一个关于每页状态的指示器，可以加快检查一个页是否包含有效数据的过程。

替换块表是根据虚拟块地址来进行索引的。对于一个给定的虚拟块，它在替换块表中的条目（entry），是一个替换块的物理地址（如果该虚拟块的替换块存在的话）。有了替换块表以后，与替换块相关的元数据信息，就可以直接从高速的 RAM 中读取，节省了大量的 OOB 数据读取开销。具体而言，替换块表可以在以下情形中避免闪存的 OOB 数据读取开销：

（1）为了检查替换块的存在，每个页读操作会导致 OOB 数据读取；

（2）针对一个页的重写，需要从主块的 OOB 区域中获取到替换块的物理地址；

（3）在垃圾回收过程中，每个需要被合并的主块，都需要从 OOB 区域中获取到替换块地址。

页状态位图是根据每个页的块内偏移量进行索引的。对于一个给定的页偏移量，页状态位图中的“1”，就表示相应的页已经被至少写入一次，“0”则表示相应的页从来没有被写入过。也就是说，页状态位图就是每页有效标志位在 RAM 中的一份拷贝。

页状态位图可以避免很多 OOB 数据读取开销。具体而言，主要包括两类 OOB 数据读取开销：

（1）每个写请求会读取 OOB 数据，来确定写请求应该到主块还是替换块中去执行；

（2）每个读请求会检查 OOB 数据，来确定一个非法读请求的可能性。

由于页状态位图避免了上述两种 OOB 数据读取开销，因此，它加速了读操作和写操作。

替换块表的空间开销和 FTL 映射表的空间开销是一样的。页状态位图的空间开销是最小的，每个页只需要一位。因此，对于一个给定大小为 S 的闪存，假设它的块大小为 B，页大小为 P，那么，替换块表就有（S/B）个条目。页状态位图就有(S/P)位。例如，对于一个 1GB 闪存，它的页大小是 512KB，块大小是 8KB，每个条目是 4B，那么，就需要$(2^{30}/2^{13})*4=512KB$的 RAM 内存空间，页状态位图需要$（2^{30}/（2^9*8）=256KB$的 RAM 内存空间。

3.2.2.2.3.2 页缓存

页缓存是一个可配置的缓存，它会保存最近写入的页的物理地址<块号,页号>。这个映射信息可以用来直接定位一个页，而不需要读 OOB 数据，这就避免了可能产生的 OOB 数据读取开销。实际上，为了定位一个页，有时候需要很多次 OOB 数据读取操作，在最坏的

情况下，可能需要读取替换块中每个页的 OOB 数据（OOB 中记录了一个页的 LPN），来确定是否包含一个指定的页，使得 OOB 数据读取次数会等于块中的页数。但是，和查找表方法不同，页缓存依赖于时间局部性。和查找表类似，页缓存的条目也是那些早已经存在于 OOB 区域中的数据的备份。因此，就没有必要把这些信息刷新到闪存中。页缓存可以改进一个页的逻辑地址到物理地址映射的时间，主要发生在以下情形：（1）在一个页的读请求期间；（2）由一个合并操作引发的读操作期间。而且要注意的是，在这两种情形下的收益，都来自于时间局部性。

3.2.3 混合 FTL 机制

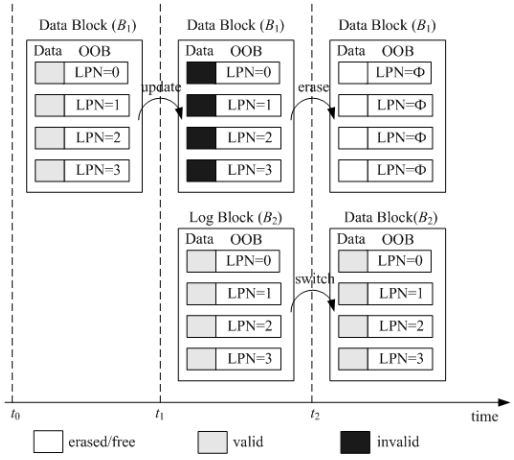
3.2.3.1 机制概述

混合 FTL 机制充分吸收了页级别和块级别映射的优点，既具备了页级别的高效和灵活性，又融合了块级别的映射表小、管理开销小的特点，从而可以有效处理比一个块小的写操作，并降低页级别地址映射的存储开销。混合 FTL 机制把逻辑块分成两类：数据块和日志块，前者采用块级别映射，用来存储数据，后者采用页级别映射，用来存储更新。数据块的数量较多，而日志块的数量较少，通常只占到闪存总空间的 3%左右[LeeSKK08]，因此，日志块采用页级别映射后，映射表的规模也很小，完全可以放入到 SRAM 中。所有日志块构成日志缓冲区，当一个更新操作需要对数据块进行更新时，只需要把更新写入到日志缓冲区中，并把数据块中相应的数据设置为“无效”，而不需要擦除原来的数据块，这就可以大大减少对数据块所要执行的擦除操作的总次数。在把逻辑更新页写入到日志块中的物理页的时候，逻辑更新页所对应的逻辑页面编号也会被同时写入到物理页的备用区域，而且写入备用区域的代价是几乎可以忽略不计的。当一个读请求到达时，必须首先检查日志块中是否已经存在被请求的页，如果日志块中已经存在该页，就直接将该页内容返回给读请求，该页数据在数据块中相应的旧版本就会被隐藏掉。当一个逻辑页被多次反复更新时，日志块中就会存在与该逻辑页对应的多个物理页，按照更新操作的先后顺序依次记录每一次更新操作，由于后面发生的更新会被记录在日志块中靠后的物理页中，因此，只要在日志块中从后往前扫描，就可以找到最新版本的物理页。

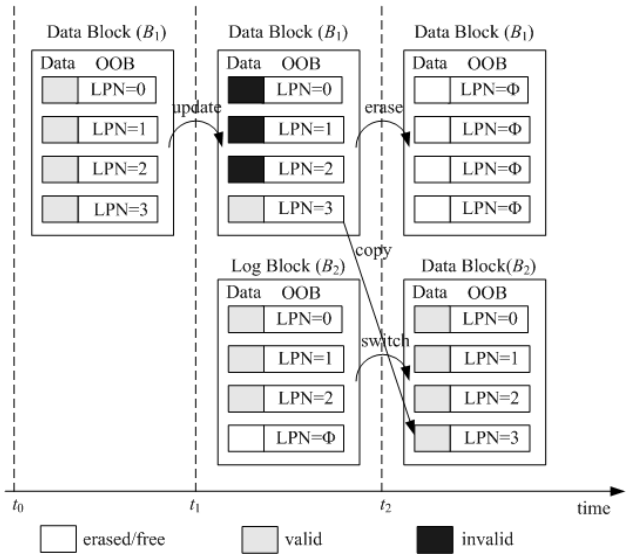
在混合 FTL 机制中，需要用日志块记录更新操作，当日志块消耗殆尽时，日志块中的一些数据就必须被刷新到数据块中，从而释放出可用的日志块空间。这就需要一个合并操作，即对数据块中的有效数据和日志块中的日志数据进行合并，写入到一个空的数据块中。

合并操作通常包含三种类型：切换合并、部分合并和完全合并。

（1）**切换合并**



图[switch-merge] 切换合并

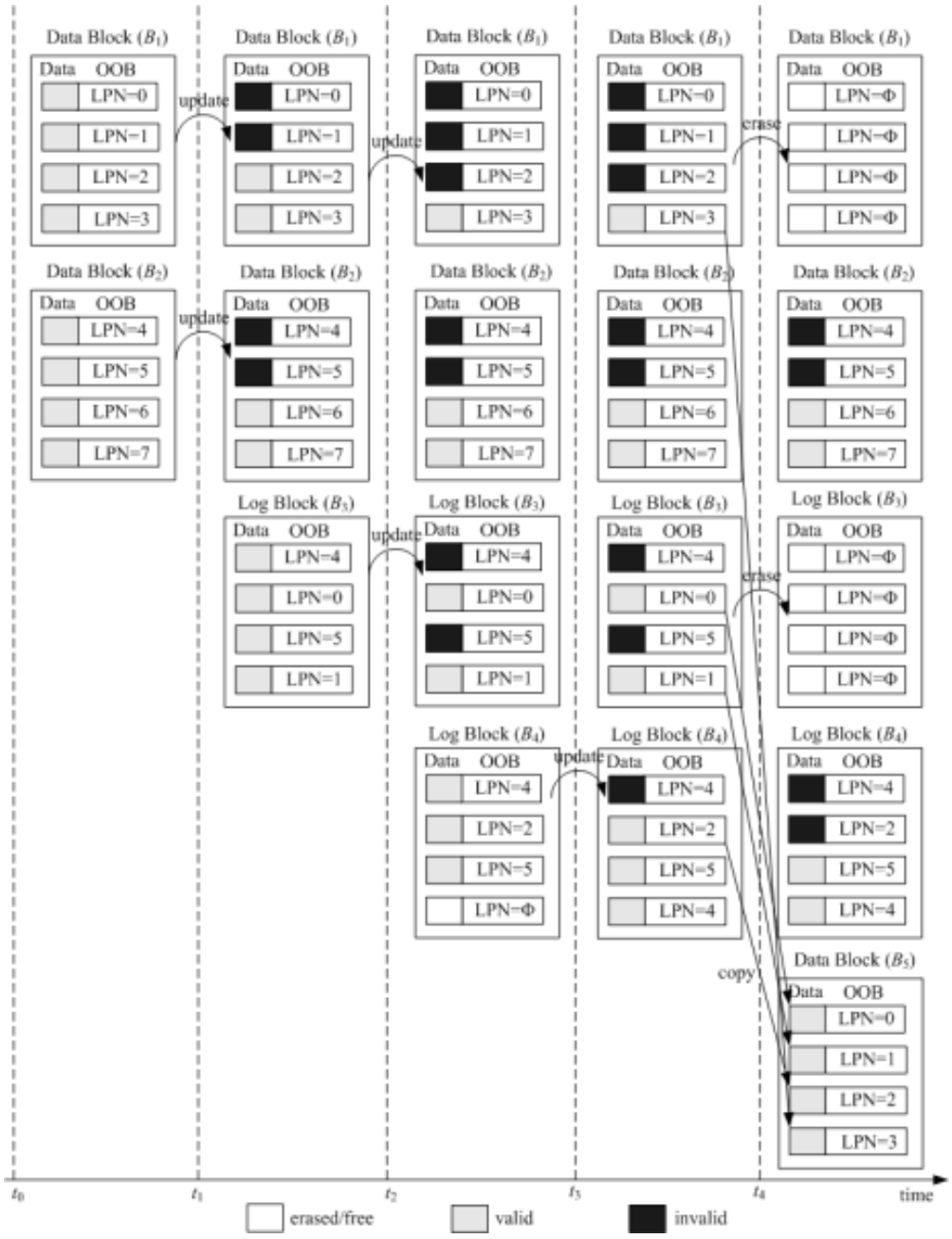


图[partial-merge] 部分合并

切换合并通常针对“顺序写入”的情形，图[switch-merge]显示了切换合并的过程。在 t_0 时刻，有一个数据块 B_1，包含了四个页面，逻辑页面编号（LPN）分别是 0、1、2 和 3，每个页面都已经写入了数据，因此，B_1 中每个页面的数据区域都用灰色背景进行标识，表示该页当前处于“有效”状态。在 t_0 和 t_1 时刻之间，发生了针对数据块 B_1 的四个页面的四次顺序更新操作。混合 FTL 机制中，对数据块的更新操作都会被记录在日志块中，因此，在 t_1 时刻，日志块 B_2 记录了上述四个顺序更新操作；这时数据块 B_1 中每个页面的数据区域都用黑色背景进行标识，表示该页当前处于“无效”状态，日志块 B_2 中的每个页面的数据区域都用灰色背景进行标识，表示该页面当前处于“有效”状态。在 t_1 时刻之后，发生了切换合并，即直接把块 B_2 从日志块变成数据块，同时，要擦除数据块 B_1，使得 B_1 再次成为空闲的数据块，可以再次用来写入其他数据，这里 B_1 中的每个页面的数据区域用白色背景标识，表示该页面当前处于“擦除/空闲”状态，擦除以后每个页面的逻辑页面编号都变成了 Φ。图[switch-merge]显示了在 t_2 时刻切换合并操作结束后各个块的状态。从上述过程可以看出，切换合并的代价很低，只需要一个擦除操作，但是，需要严格的前提条件，即只有当一个数据块中的所有页面都是顺序更新（即从第一个逻辑页到最后一个逻辑页依次进行更新）的时候，才可以进行切换合并。

（2）部分合并

部分合并针对“部分顺序写入”的情形，图[partial-merge]显示了部分合并的过程。在 t_0 时刻，有一个数据块 B_1，包含了四个页面。在 t_0 和 t_1 时刻之间，发生了针对数据块 B_1 的前三个页面的三次顺序更新操作，即分别更新了逻辑页面编号为 0、1 和 2 的三个页面，而逻辑页面编号为 3 的页面并没有发生更新。在 t_1 时刻，日志块 B_2 记录了上述三个顺序更新操作。在 t_1 时刻之后，发生了部分合并操作，即把数据块 B_1 中的有效页（LPN=3）复制到日志块中，然后，执行一个切换操作，让日志块 B_2 变成一个数据块，同时对数据块 B_1 执行擦除操作，成为空闲页。图[partial-merge]显示了在 t_2 时刻部分合并操作结束后各个块的状态。从上述过程可以看出，部分切换不仅需要一个擦除操作，还需要一个额外的复制操作，因此比切换合并代价高。使用部分合并的前提条件是，顺序更新操作没有执行满一个块。



图[full-merge] 完全合并

（3）完全合并：

完全合并通常是针对“随机写入”的情形，图[full-merge]显示了完全合并的过程。在 t_0 时刻，有两个数据块 B_1 和 B_2。在 t_0 和 t_1 时刻之间，发生了针对数据块 B_1 的两个页面（LPN=0 和 LPN=1）的更新操作，同时还发生了针对数据块 B_2 的两个页面（LPN=4 和 LPN=5）的更新操作，这四个更新操作的先后顺序是依次更新 LPN 为 4、0、5 和 1 的页面。更新结束后，在 t_1 时刻，日志块 B_3 按照更新的先后顺序依次记录了上述四个随机更新操作，这时，B_3 中的四个页面都处于“有效”状态，而数据块 B_1 的两个页面（LPN=0 和 LPN=1）和数据块 B_2 的两个页面（LPN=4 和 LPN=5），则因为已经被更新过而处于“无效”状态。在 t_1 时刻之后，又先后发生了针对不同三个页面（LPN=4、LPN=2 和 LPN=5）的更新操作，由于这里假设一个块只包含四个页，而日志块 B_3 已经填满，因此，必须把这三个更新操作依次记录到新的日志块 B_4 当中。需要注意的是，在第二次更新 LPN 为 4 的页面时，因为此前已经对该页

更新过一次，它的最新版本的数据已经在日志块 B_3 中（而不是数据块 B_2 中），因此，需要在日志块 B_3 中把 LPN=4 的页面更改为“无效”，并把最新版本的数据写入到日志块 B_4 中。对于 LPN=5 的页面而言，也是发生了二次更新，处理方法和 LPN=4 的页面相同。在 t_2 时刻，日志块 B_4 记录了上述三个更新操作，因此，B_4 中的前三个页面都处于“有效”状态，而 B_4 中还剩余一个页面，该页面没有写入数据，因而处于“擦除/空闲”状态。在 t_2 时刻之后，又发生了针对 LPN=4 的页面的第三次更新操作，该操作会被记录到日志块 B_4 中的最后一个空闲页里面，同时，日志块 B_4 中第一个页面（LPN=4）被设置为“无效”。在 t_3 时刻，日志块 B_4 中已经记录了刚才发生的更新。在 t_3 时刻之后，发生了完全合并操作，具体方法是：选择日志块 B_3 作为牺牲日志块，同时为本次完全合并操作分配一个新的空闲数据块 B_5，然后，把数据块 B_1 中的有效页（LPN=3）、日志块 B_3 中的有效页（LPN=0，LPN=1）、日志块 B_4 中的有效页（LPN=2），分别复制到新的空闲数据块 B_5 中，最后，擦除数据块 B_1 和日志块 B_3，这两个块又成为空闲块。需要注意的是，把日志块 B_4 中的有效页（LPN=2），复制到新的空闲数据块 B_5 中以后，需要把日志块 B_4 中的这个页（LPN=2）更改为“无效”。图[full-merge]显示了，在 t_4 时刻完全合并操作完成以后，各个数据块和日志块的状态。从上述过程可以看出，完全合并操作需要多个复制操作和擦除操作，代价比较高，要明显高于切换合并操作和部分合并操作。完全合并操作应用于随机更新操作较多的场合。

上面描述的只是关于完全合并的一个非常简单的实例，实际上，完全合并可能要更加负载，代价更高。比如，在经过一系列的各种合并操作以后，可能会出现一个日志块和数据块之间的“循环关联”，也就是说，一个被选中牺牲的日志块可能包含和多个数据块相关的更新，而这些数据块又包含和多个日志块相关的更新过的页面。这种情形下，完全合并的代价就会变得非常高，反过来，它又会影响到后续其他操作的性能，不管后续操作是顺序操作还是随机操作。

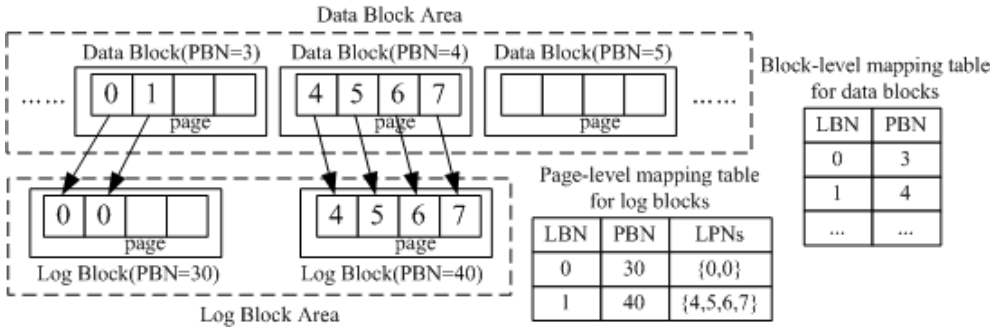
正因为完全合并具有很高的代价，大多数基于日志缓存的混合 FTL 机制重点研究解决的问题，就是如何减少完全合并操作的数量。比较典型的方法是，对数据块进行划分，分成冷块和热块[HsiehK06]。被频繁访问的数据被保存在热块中，通常会包含大量的无效页，而冷块则包含被很少访问的数据。在垃圾收集时使用热块，减少了有效页面的复制开销，因此，也就降低了完全合并的开销。不过，这种做法也只是在一定程度上缓解完全合并的问题，并没有从根本上解决这个问题，因此，文献[GuptaKU09]的大量实验测试表明，在一些比较复杂的负载类型下，各种混合 FTL 机制的性能表现都不尽人意。

3.2.3.2 典型研究

3.2.3.2.1 BAST

文献[KimKNMC02]在 2002 年提出的 BAST (Block-Associative Sector Translation)，是第一个混合 FTL 机制。BAST 的主要目标就是，在有限的 SRAM 下，能够实现高效的小数据量类型的写操作和较长的顺序写操作。为了达到这个目的，作者采用了页级别映射的日志块和块级别映射的数据块，针对数据块的更新操作都被记录到日志块中。BAST 采用“块关联映射”（block associative mapping），每个数据块都与唯一一个特定的日志块关联，即每个日志块只能容纳来自同一个逻辑块的页。为了提高性能，BAST 把日志块的页级别映射表存放到 SRAM 中，同时，BAST 还可以在发生断电时保证所存储的数据的一致性。BAST 的读性能要比块级别 FTL 好，因为，SRAM 的速度要比闪存的速度高好几个级别。当访问模式主要是顺序写时，BAST 可以提供高效的垃圾回收过程，因为，这种情形涉及大量的低代价的切换合并。

3.2.3.2.1.1 BAST 机制的写操作过程



图[BAST] BAST 机制的写过程

下面重点介绍 BAST 机制中的写操作执行的过程，从而对该机制的技术细节有更好的理解。

假设闪存中只有两个日志块和若干数据块，每个块包含四个页。现在假设有一个更新序列 U_0,U_1,U_0,U_0,U_4,U_5,U_6,U_7,U_4,U_5,U_6,U_7，其中，U_i 表示对 LPN=i 的页面进行更新操作。U_0 执行时，首先根据 LPN=0 计算出该逻辑页所在的逻辑块的 LBN（逻辑块编号）为 0，计算方法是：LPN 除以每个块包含的页数，得到的商就是 LBN；其次，在面向数据块的块级别映射表中找到 LBN=0 的逻辑块所对应的物理块的 PBN（物理块编号）为 3；再次，计算出物理块内的物理页的块内偏移量为 0，计算方法是：用 LPN 除以每个块包含的页数，得到的余数就是偏移量。最后，根据 PBN=0 和块内偏移量 0，把 U_0 写入到 PBN=0 的物理块内的第 1 个页内（第 1 个页的块内偏移量为 0）。同理，可以把第二个到达的更新 U_1 写入到 PBN=0 的物理块内的第 2 个页内。

当第三个更新 U_0 到达时，是执行对 LPN=0 的逻辑页的第二次更新操作，因为此前在数据块中已经保存了该逻辑页中旧版本的数据，因此，本次更新操作会被记录到日志块中，具体方法是：BAST 首先查找是否已经存在和 LBN=0 的逻辑块对应的日志块，发现不存在，就会把第一个日志块（PBN=30）分配给该逻辑块，专门用于记录与该逻辑块相关的更新操作，接着把更新操作记录到该日志块的第一个空闲页内，由于当前日志块的所有页面都是空闲页，因此，更新会被记录到第一个页中；最后，对面向日志块的页级别的映射表进行更新，写入一个新条目，即 LBN=0,PBN=30,LPNs={0}，表示与 LBN=0 的逻辑块相关的所有更新操作都被记录在 PBN=30 的日志块中，并且该日志块中当前已经记录了一次针对 LPN=0 的逻辑页的更新操作。

当第四个更新操作 U_0 到达时，又是针对 LPN=0 的逻辑页的第三次更新操作。因此，会被记录到 PBN=30 的日志块中的第二个页中，然后，对页级别映射表进行更新，即把条目“LBN=0,PBN=30,LPNs={0}”更新为即“LBN=0,PBN=30,LPNs={0,0}”。

当第五、六、七和八个更新操作 U_4,U_5,U_6,U_7 依次到达时，会被依次记录到 PBN=4 的数据块中。当第九个更新操作 U_4 到达时，这是对 LPN=4 的逻辑页的第二次更新操作，此前已经在数据块中存在旧版本数据，因此，本次更新会被记录在日志块中。LPN=4 的逻辑页所在的逻辑块的 LBN 为 1，BAST 首先查找是否已经存在和 LBN=1 的逻辑块对应的日志块，发现不存在，就会把第二个日志块（PBN=40）分配给该逻辑块，专门用于记录与该逻辑块相关的更新操作。这里需要特别注意的是，BAST 机制的设计中，只允许一个日志块容纳来自同一个逻辑块的更新操作，因此，第一个日志块（PBN=30）是专门分配给 LBN=0 的逻辑块使用的，不能用来记录针对 LBN=1 的逻辑块的更新操作，所以，这里才会分配第二个日志块（PBN=40）给 LBN=1 的逻辑块使用。分配到日志块后，更新操作 U_4,会被记录到 PBN=40 的日志块中，并对页级别映射表进行更新。依此类推，接下来的另外三个更新操作 U_5,U_6,U_7 也会被依次记录到 PBN=40 的日志块，更新完成后的页级别映射表相应的条目为

“LBN=1,PBN=40,LPNs={4,5,6,7}”。

假设此后还有其他的更新操作到达，PBN=5 的数据块会被写满。PBN=5 的数据块中的物理页所对应的逻辑页，如果发生了再次更新操作，就又需要把这些操作记录到日志块中，需要为它分配一个新的日志块。但是，之前已经假设最多只有两个数据块，因此，这里就必须从 PBN=30 和 PBN=40 的日志块中选择一个日志块作为牺牲品，具体过程如下：首先分配一个空闲的数据块（假设 PBN=9），然后把牺牲日志块（假设 PBN=30）和它对应的数据块（假设 PBN=3）中的最新版本的数据都复制合并到该空闲数据块中（根据情况不同，可能发生切换合并、部分合并或完全合并），然后擦除 PBN=30 的数据块和 PBN=3 的日志块，让这两个块成为空闲块，最后，要修改块级别的映射表，即把“LBN=0，PBN=3”这个条目修改为“LBN=0，PBN=9”，同时，还要修改页级别的映射表，即删除即“LBN=0,PBN=30,LPNs={0,0}”这个条目。

3.2.3.2.1.2 BAST 的缺陷

BAST 的缺陷主要体现在以下几个方面：

- BAST 在面对小数据量的随机写请求时，日志缓冲区的利用率就会大大降低，因为，即使更新数据块的一个页面，也需要使用整个日志块。日志块对于减少“写前擦除”次数的作用很大，降低了日志块的利用率，也就等于降低了日志块对于闪存性能提升的作用。文献[LeePCLPS07]的研究发现，对于 BAST 而言，当日志块的数量少于“热块”（被频繁更新的块）的数量时，日志块的利用率很低，因为，日志块会很快成为被替换掉的牺牲品，根本没有时间让其他更新来把块中剩余的页面填满；另外，在日志块的数量保持不变时，负载类型越随机，BAST 的日志缓冲区利用率就越低。
- 对于 BAST 而言，由于采用“块关联映射”，当对数据块中的某个页进行更新时，这个更新操作只能被引导到与该数据块关联的特定数据块上面去执行，而不能引导到其他日志块上面。这会导致日志块的利用率较低，同时也增加了完全合并的开销。而且，在有限的闪存空间限制下，BAST 会很容易耗光可用的替换块，从而不得不启动回收进程，回收那些还没有被占用的块。因此，BAST 中替换块的利用率在理论和实践上都是很低的。而且，在一个给定的时间窗口内对一个块进行频繁更新操作，会导致频繁发生合并操作，这也会导致写操作数量的增加。
- 由于日志块数量有限，频繁随机写操作，还可能会导致“块抖动”问题[LeePCLPS07]（和内存页面抖动问题是类似的，请参见例子 1），因而无法获得好的性能。当对一个数据块执行一个更新操作时，如果不存在与该数据块对应的日志块，就需要从已有的日志块中选择一个作为牺牲品，把该日志块和与之关联的数据块执行合并操作（很多情况下会发生完全合并操作），然后擦除该日志块，供当前的更新操作使用。如果日志块的数量有限，那么这种选择牺牲品的替换操作会频繁地发生。

例子 1：这里以一个实例来阐述 BAST 存在的“块抖动”问题。假设闪存中只有两个日志块，每个日志块包含四个页。这里假设这八个逻辑页面对应的数据都已经保存在闪存的物理块中，而且这些块都已经填满。现在假设有一个更新序列 $U_0,U_4,U_8,U_{12},U_0,U_4,U_8,U_{12}$，其中，$U_i$ 表示对 LPN=i 的页面进行更新操作，并且不同的 i 分别来自不同的块。执行完 U_0 以后，日志块 B_0 记录了本次更新，B_0 被放入到 SRAM 缓存中。接着执行完 U_1 以后，需要使用另一个日志块 B_1 来记录本次更新，因为即使更新数据块的一个页面，也需要使用整个日志块；B_1 被放入到 SRAM 缓存中以后，这时缓存已经填满，没有剩余空间。接着执行完 U_8 以后，需要使用一个新的日志块 B_3 来记录本次更新，B_3 应该被放入缓存，但是，此前缓存已经填满，因此，就必须采用适当的替换策略，把 B_1 和 B_2 中的其中一个驱逐出缓存，然后把 B_3 放入缓存。在执行后续的每个更新以后，都会发生类似的块替换问题，这就导致了“块抖动”问题。

3.2.3.2.2 FAST

为了解决块抖动问题，文献[LeePCLPS07]提出了一种新的混合 FTL 机制——FAST(Fully Associative Sector Translation)。和 BAST 不同的是，FAST 没有采用“块关联映射”，而是采用“完全关联映射”（fully associative mapping），即允许日志块被所有的数据块共享，从而改进了日志块的利用率，而且只有在日志缓冲区没有剩余空间的时候，才会启动垃圾回收过程。这种方法有效地消除了块抖动问题，并且在面对随机负载时也可以提高垃圾回收效率。为了增加部分和切换式合并的比例，减少合并操作的开销，FAST 为顺序更新专门确定了一个顺序日志块，而其他日志块则用来执行随机写操作。作者的实验表明，在最好的情况下，在存取时间和擦除操作的次数方面，FAST 都要比 BAST 降低 50%以上。

FAST 的缺陷体现在以下几个方面：

- 这种方式带来的性能优化效果也十分有限，因为，在现代的多线程环境下，一个顺序写操作通常会被随机写和其他顺序写操作打断。因此， FAST 不能容纳多个数据流，并且它也没有提供任何特定的机制来处理随机数据流中存在的时间上的局部性。
- 虽然 FAST 尽可能地延迟了垃圾收集的时间，但是，回收一个单个的日志块所耗费的时间，会比 BAST 更长[MaFL11]。

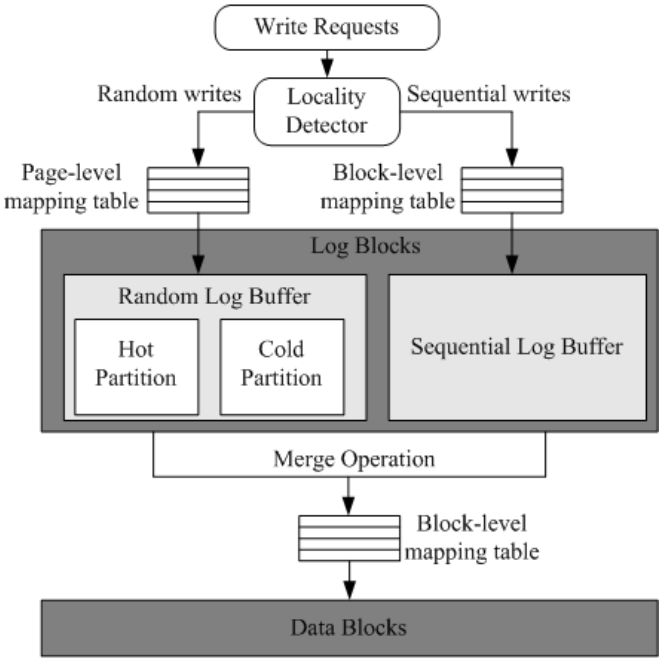
3.2.3.2.3 LAST

通常而言，在通用的计算机系统中存在的典型的负载，是随机写和顺序写的混合。因此，LAST[LeeSKK08]机制从混合负载中分离出顺序负载，从而可以对顺序负载执行更多的、低代价的切换合并和部分合并，提高整体性能。图[LAST]显示了 LAST 机制的体系架构。LAST 机制对日志块区域进行了功能分段，分成顺序日志缓冲区（包含多个顺序日志块）和随机日志缓冲区（包含多个随机日志块）。

在顺序日志缓冲区，LAST 采用了和 BAST 类似的“块关联映射”，即每个顺序日志块只和一个数据块对应，因为，块关联映射比较适合顺序访问类型的负载。LAST 设计多个顺序日志块可以充分利用负载的空间局部性，较长的请求会被写入到顺序日志块，从而有利于执行部分或者切换式合并，这就可以尽力缓解 FAST 的缺陷，因为在 FAST 中只有一个顺序日志块。

在随机日志缓冲区，LAST 采用了和 FAST 类似的“完全关联映射”，因为，这种方式更加适合随机访问类型的负载，尤其是对于那些具有很高的时间局部性的随机负载而言更是如此。LAST 进一步把随机日志块分成热块和冷块，来减少完全合并代价，即将被写入的热数据，会被放入热块中，其他的写请求则由冷块来服务。LAST 依赖一个外部的局部性探测机制来确定顺序写和随机写。

LAST 机制的缺陷是：作者自己也意识到，当小数据量的写操作具有顺序的局部性时，他们提出的局部探测器可能无法有效地确定这种顺序写操作。实际上，即使一个探测器非常智能，在测试基准上可以有效区分顺序和随机 IO，但是，在运行时刻，面对各种复杂的应用环境时，也会遇到大量的问题导致无法有效区分顺序和随机 IO。例如，在缓冲池中缓存数据就会极大改变物理数据访问模式。



图[LAST] LAST 机制的体系架构

3.2.3.2.4 SuperBlock

SuperBlock 机制[KangJKL06]和 FAST 不同，并不允许日志块被所有的数据块共享，而是在 *N* 个数据块中共享最多 *K* 个日志块。该机制还充分利用了工作负载中存在的块级别的空间局部性，会把连续的逻辑块合并成一个超级块，在超级块内部，采用页级别的映射。该机制还把超级块内部的热块和冷块做了区分，从而可以充分利用请求数据流中的时间局部性。但是，采用超级块机制后，属于 *N* 个逻辑块的页可能被分布到 *N*+*M* 个物理块中（*M* 表示额外分配给超级块的更新块的数量），因此高效地维护映射表信息就成为一个具有挑战性的问题。为此，该机制把映射表被保存到超级块的备用区域，当用户数据被写入到闪存的数据区域时，最新的页面地址映射信息也会被写入到闪存的备用区域，因此，不会引起额外的闪存空间开销和操作开销。

SuperBlock 机制的缺陷体现在以下几个方面：

- 由于备用区域的大小限制，Superblock 采用了三级地址映射方法，从而可以把映射信息保存到有限的备用区域中，这样就需要最多搜索三个备用区域来寻找一个被请求的页，需要一定的代价。
- 该机制采用固定的超级块大小，为了适应不同的工作负载类型，就需要显式地对超级块大小进行调整。
- 该机制只对冷块和热块进行了区分，却没有对块内的冷页和热页进行区分。
- 需要花费一定的代价来维护一个超级块内部的页级别的映射信息。

3.2.3.3 其他相关研究

与 SuperBlock 一样，SAST[LeePCLPS07]也是在 *N* 个数据块中共享最多 *K* 个日志块，而且不同的数据块集合，会竞争日志块。SAST 包含了调优的过程，在使用之前需要进行调优，这意味着，当访问模式发生变化时，它的性能就会恶化。A-SAST (Adaptive SAST) [KooS09]是一个针对 SAST 的优化版本，它放松了一个数据块集合中可以共享的日志块的最大数量的限制，可以对数据块集合进行动态划分和合并。

HFTL[LeeYL09]首先利用热数据探测方法（基于哈希的热数据区域标识技术）对冷热数据进行区分，然后，对热数据采用扇区映射机制，而对冷数据采用基于日志块的机制。但是，

当访问模式变化时，一些热页需要被交换出来，这就需要额外的开销。

3.2.4 变长映射

文献[ChangK04]的作者对两种典型负载进行了连续跟踪：（1）普通用户电脑上的操作（包括网页浏览、邮件收发、文档编辑和游戏等）和（2）多媒体文件存储系统。通过对跟踪结果的分析，作者观察到：（1）小尺寸的写操作具有很强的空间局部性；（2）批量写操作一般都会顺序访问磁盘。对于多媒体存储而言，绝大多数写操作都是批量的顺序操作。因此，作者认为闪存管理机制需要可变的粒度来适应不同的访问模式，而传统的方法通常采用固定的粒度，即采用两个固定的表来执行地址映射和空间管理。由此，文献[ChangK04]在 2005 年提出了一种变长的映射机制。变长映射机制会把变长的、连续的逻辑页映射到闪存中的物理页。当访问模式发生变化时，可以进行粒度的动态调整。

变长映射机制的缺点是：由于不同映射单元的尺寸并不相同，而且是动态变化的，映射表只能存放在一些特定类型的搜索树中，由此导致的后果就是，相对于其他 FTL 机制而言，变长映射表的查找代价比较大，因为，在其他 FTL 机制中，映射表就是一个非常简单的地址数组。

3.2.5 关于不同映射机制的讨论

对于一个采用了固态盘的存储系统而言，不管它采用什么样的 FTL 映射方式，这个存储系统都应该保证操作的可靠性，也就是说，脏数据（即发生了更新的数据）在操作返回前应该被刷新到闪存中。因此，设计 FTL 机制必须以保证操作可靠性作为提高性能的前提。在此前提下，为了设计一个高效的 FTL 机制，首先应该考虑映射粒度。在所有的 FTL 机制中，页面级别的映射，是最有效和高效的映射粒度，但是，映射表开销较大。块级别的映射都无法对冷数据和热数据进行区别对待，因此，在垃圾回收过程中，必须对冷数据进行不必要的移动，增加了存储系统的整体开销。变长映射虽然可以动态调整映射的粒度，但是，地址翻译的高复杂性是一个不容忽视的致命缺陷。混合映射比较灵活，因为通过对 LBA 进行分区，或者共享日志块，可以尽量避免代价高昂的完整合并操作。但是，需要指出的是，无论设计得多么细致，混合映射机制都无法完全消除完整合并。

3.3 双模式FTL

在过去几十年里，磁盘在存储系统中占据了举足轻重的地位，这是因为磁盘遵循了两条简单的标准，从而对上层应用表现出了稳定的性能：第一，数据存储如果在逻辑地址空间内具有局部性，那么在物理地址空间上也可以保持这种局部性；第二，顺序访问要比随机访问快许多。由于 Unix 的出现，接口的稳定性和磁盘特性的稳定性，确保了主要的数据库系统设计原则的长久有效性，这些设计原则包括：（1）以页作为基本的 IO 单元，在磁盘和内存中都用页来访问数据；（2）避免随机访问（比如查询处理算法），尽量采用顺序访问（比如聚类）。

随着闪存的出现，闪存存储设备已经开始在企业存储系统中发挥着越来越重要的作用。但是，与磁盘设备具有一致稳定的性能不同的是，闪存设备不仅自身无法提供稳定一致的性能，而且不同闪存设备之间的性能差异也较大。例如，对于 FusionIO[FusionIO]而言，随机写的速度要比读速度快。而在许多三星闪存设备上，随机写的速度要比读速度慢许多。对于一些闪存设备而言，性能会随着时间变化，因为，这些闪存设备的性能和历史 IO 模式有关。比如，INTEL X25-M 的性能的变化幅度会达到一个数量级，而其性能的高低取决于设备是否采用随机写。如果让一个数据库的设计建立在一个性能不断变化的存储系统上，要想获得较好的性能是很难的。

但是，数据库设计者和闪存设备设计者毕竟属于两个不同的领域，二者有着各自的设计目标。对于数据库设计者而言，他们会竭力控制由自己发起的 IO 标准，通常会首先对高效

和低效的 IO 模式进行明确的界定和区分，从而使得他们可以调整分配策略、数据表示或者查询处理算法，以便更好地适应底层的存储系统。为了获得更好的性能和稳定的行为，数据库群体甚至不惜以增加设计复杂性为代价。对于闪存设备设计人员而言，比如 SSD 设计者，会竭力隐藏闪存的各种内部特性，提供和磁盘一样的标准接口，从而和磁盘供应商进行竞争。不同闪存设备设计者之间也会相互竞争，设计更好的 FTL 来改善性能，并且隐藏自己的设计决策来保护他们自己的优势。实际上，闪存芯片早已经提供了对高效模式和低效模式之间的明确区分，页读操作和一个块内部的顺序页写操作，就属于高效模式，而就地更新操作就属于低效模式。闪存设备应该对上层应用暴露这种区分，而不是竭力以高效模式的性能为代价来减少低效模式带来的负面影响，比如以降低读性能为代价来获得改进的随机写性能。

由此可以看出，数据库设计者和闪存设备设计者之间缺乏有效的合作，导致采用闪存存储设备的数据库系统无法获得最佳的性能。闪存设备隐藏了闪存内部的特性，为数据库系统提供了标准的接口，数据库系统无法了解闪存设备的内部工作机制，因此，无法为闪存设备提供最优的 IO 模式，而闪存设备自身进行性能优化时，又完全不考虑上层数据库应用的 IO 模式特性，块分配和垃圾回收工作可能存在一定的盲目性，无法为数据库系统提供稳定的 IO 行为。

因此，当前闪存设备的复杂性和透明性对于数据库管理而言是很不合适的。数据库设计者和闪存设备设计者之间必须建立合作，保证闪存数据库获得较好的性能。现在的问题就是二者如何合作。也就是说，DBMS 如何为闪存设备优化自己的 IO 模式？闪存设备如何才能保证为 DBMS 提供稳定的 IO 行为？

针对上述问题，Bonnet 等人[BonnetB11]提出了双模式闪存设备的概念，即采用双模式 FTL 的闪存设备，并认为当前的闪存设备应该朝着双模式闪存设备发展，这种发展方向对于数据库机器的设计者而言是十分重要的，比如 Oracle Exadata 或者 Neteeza TwinFin，这些数据库机器必须采用适合自己的闪存设备，才能提供高性能的数据库服务。实际上，如果按照双模式闪存设备的理念，数据库群体是可以使得闪存设备的发展朝着符合自己商业化数据库系统利益的方向进化的。

双模式 FTL 包括两种模式：

- 第一种模式：禁止更新和随机写操作，可以为顺序写、顺序读和随机读操作保证提供最优的性能；
- 第二种模式：可以支持更新和随机写操作，但是不能保证提供最优的性能，只能尽量提供好的性能。

双模式 FTL 的两种模式都是和闪存芯片存在的 4 种约束有关的。闪存芯片存在严重的性能约束：

- 约束 C1：写操作必须以页为单位执行；
- 约束 C2：写前擦除；
- 约束 C3：在一个块内必须进行顺序写操作；
- 约束 C4：有限的生命周期。

尽管闪存芯片的发展趋势是，闪存芯片的每个单元可以存储更多的位，处理时间更短（比如 25 纳秒），页尺寸更大，一个块内包含更多的页数量，生命周期更短，但是，所有这些发展趋势都没有违反性能约束 C1 到 C4。

如图[Bimodal-FTL]所示，如果采用第二种模式的 FTL，那么 DBMS 就不必处理 C1 到 C3 的约束，也就是说，DBMS 可以向闪存设备提交任何模式的 IO，必须由闪存设备自己来处理 C1 到 C4 的约束。如果采用第一种模式的 FTL，那么 DBMS 就必须处理 C1 到 C3 的约束，也就是说，DBMS 不能自己提交包含就地更新和随机写操作的 IO。而 FTL 则会保证把遵循 C1 到 C3 的约束的 IO，直接提交给底层的闪存芯片。在这种模式下，闪存设备的 FTL

会处理约束 C4，即提供磨损均衡功能。实际上，一个 DBMS 是无法执行磨损均衡（约束 C4）的，磨损均衡功能必须由 FTL 来提供。磨损均衡对于确保闪存设备的寿命而言，绝对是必须的。闪存设备生产商绝对不会提供一个只经过几分钟频繁的写入操作后就会报废的产品。为了支持磨损均衡，闪存设备的 FTL 机制必须实现某种形式的块映射，从而让擦除操作均匀地分布到不同的块上。那么，FTL 可以取得的最优映射是什么呢？

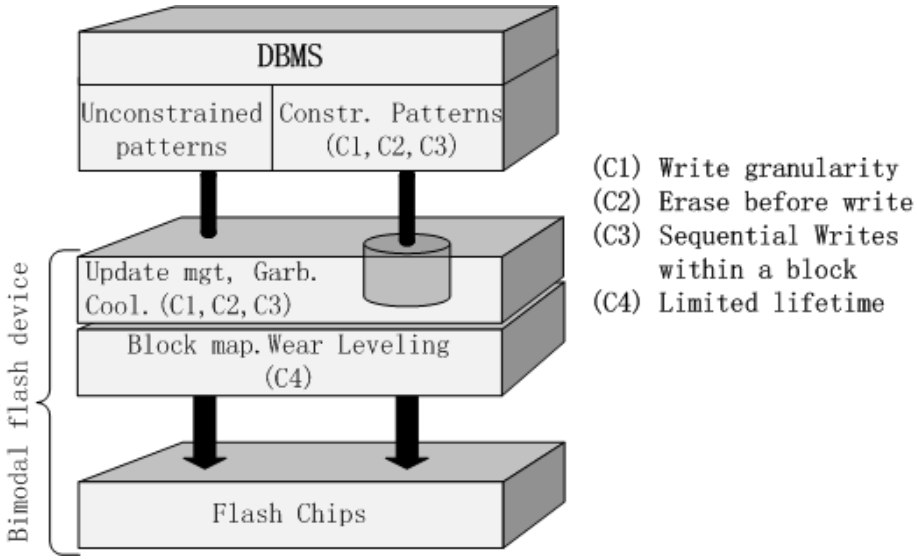
一个映射是最优的，当以下条件成立时：

(a)块的查找操作是在闪存控制器的缓存中执行的；

(b)块内部的页的偏移量可以直接根据逻辑地址计算得到，即在一个块内会顺序写入连续的逻辑地址。

很显然，顺序写操作会导致最优的映射。另外，半随机写操作，也就是随机写操作被顺序地映射到不同的逻辑块上面，也会导致一个最优的映射。一个闪存块，如果它的映射是最优的，那么，就称它为“最优块”。一个最优块从来不会被垃圾回收，因为，在最优映射中，顺序写操作不会在一个块中产生任何废弃的页。

当采用第一种模式时，DBMS 已经可以保证处理 C1 到 C3 的约束，那么，FTL 就必须保证为遵循 C1 到 C3 约束的这些逻辑块提供最优的映射。当采用第二种模式时，DBMS 提交给闪存设备的逻辑块不必遵循 C1 到 C3 约束，允许任意的 IO 模式，因此，这些逻辑块会被映射到一个或者多个物理闪存块中，并非采用最优的映射。



图[Bimodal-FTL] 双模式 FTL 机制示意图

双模式闪存设备可以为 DBMS 设计提供一个稳定和最优的性能基础。现有的一些研究工作，着力研究提供数据库顺序写这种 IO 模式，这样就可以弥补低端固态盘的较差的随机写性能。这些技术可以被应用在双模式 FTL 中，从而充分发挥顺序写操作的最优性能。同时，一些专门为闪存芯片设计的技术[2][17]，对于当前的固态盘而言仍然存在许多不足之处，但是，这些技术可以很自然地应用到双模式 FTL 中，因为它们只会生成最优块，符合双模式 FTL 中的第一种模式标准，双模式 FTL 可以保证为这种模式提供最优的性能。

3.4 本章小结

本章内容首先介绍了 FTL 的功能，包括地址映射、垃圾回收、磨损均衡、断电恢复等；然后，详细介绍了 FTL 的各种映射机制，包括页级别 FTL 机制、块级别 FTL 机制、混合 FTL 机制和变长映射机制；最后介绍了一种双模式 FTL，它可以使得数据库设计者和闪存设备设计者之间建立合作，保证闪存数据库获得较好的性能。

FTL 机制对于闪存数据库而言非常重要，它可以把一个闪存设备模拟成一个类似磁盘的

块设备，从而使得以前面向磁盘开发的 DBMS，不用任何修改就可以直接运行在具备 FTL 功能的闪存设备上。但是，由于大多数 FTL 机制都是针对文件系统开发的，并没有考虑数据库系统的 IO 特点，因此，简单地把传统的 DBMS 移植到闪存设备上，是无法充分发挥闪存的优良特性的，比如针对闪存特性，对 DBMS 或 FTL 机制进行相应的修改，从而获得最优的性能。这些内容将在后续章节中介绍。

3.5 习题

1、 阐述 FTL 的主要功能并说明每种功能的具体含义。

2、 分别说明页级别、块级别和混合 FTL 机制以及变长映射的各自优缺点。

3、 说明 BAST 机制中为什么会出现“块抖动”问题，应该设计什么机制来缓解该问题？

4、 阐述双模式闪存设备的概念以及双模式 FTL 中的两种模式的具体含义。

第二篇 闪存数据库篇

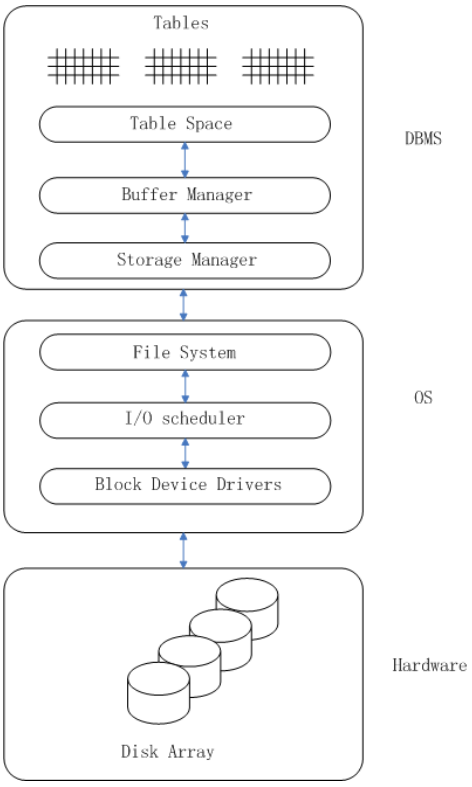
第 4 章 闪存数据库概述

闪存数据库与磁盘数据库的显著区别在于，前者采用闪存作为数据库系统的底层存储介质，而后者采用磁盘作为存储介质。存储介质的变化，会影响到 DBMS 的性能发挥，其中影响最大的就是 DBMS 存储性能，而存储性能又是数据库整体性能的关键环节，较差的存储性能必然导致较差的数据库性能。如果直接把面向磁盘设计的 DBMS 直接移植到闪存上面，不仅不能充分利用闪存优良特性来获得最佳的数据库性能，甚至在某些情况下会导致数据库性能的下降。因为，面向磁盘设计的 DBMS，在各种存储优化机制中都以磁盘技术特性为依据，在优化机制设计上都是为了达到两个方面的目的，一方面，充分利用磁盘高顺序读写带宽，另一方面，尽量避免随机读写操作带来的高延迟。而闪存和磁盘具有截然不同的技术特性，这意味着之前的各种数据存储优化机制在闪存数据库上将无法取得好的性能，必须重新设计面向闪存的数据库存储优化机制。此外，存储介质的变化对于 DBMS 其他方面也会产生影响，需要重新设计能够充分利用闪存特性的相关机制，比如重新设计事务管理机制。

本章首先介绍基于磁盘的 DBMS 存储性能优化技术，然后阐述设计实现面向闪存的 DBMS 的必要性、考虑因素和技术路线，接下来介绍采用三种不同技术路线的代表性研究，即**设**计面向 DBMS 的 FTL 的方法、采用 FTL 并修改面向磁盘的 DBMS 的部分模块的方法和抛弃 FTL 并修改面向磁盘的 DBMS 的部分模块的方法。

4.1 基于磁盘的DBMS的存储性能优化技术

图[DBMS-storage-hierarchy]给出了数据库系统的存储层次架构。在数据库中，多个关系和索引在逻辑表空间上会被聚集到一起，然后保存到存储介质上。缓冲区管理器维护了一个内存缓冲区池，可以把数据缓存在内存中。存储管理器负责针对底层存储设备的 IO 操作。存储管理器可以直接访问设备，或者可以通过文件系统来执行 IO 操作。



图[DBMS-storage-hierarchy] 数据库系统的存储层次架构

大多数具备存储性能优化机制的 DBMS，一般都采用以下几种技术来改进磁盘的存储性能：

- 缓冲区：在内存中设置数据库页的缓存可以有效提升数据库的性能，通过把那些经常被访问的数据库页放置到缓冲区中，当外部请求到达时，如果请求页正好在缓冲区中，那么，就不需要到辅助存储设备（比如磁盘或闪存）中读取该页，由于内存读写速度要远远高于辅助存储设备，因此，缓冲区可以明显减少 I/O 开销。
- 页布局：使用基于行或列的页布局或者它们的组合，来减少磁盘 IO 代价。存在两种主要的页布局方式，即基于行的页布局和基于列的页布局。如果需要读取整行数据，那么，行布局方式比较好，如果只需要读取一部分属性，那么，采用列布局更加合适。此外，也存在其他类型的混合布局方式，比如 PAX 和 DMG，这些页布局方式混合了行和列两种方式。
- 数据压缩：使用数据压缩，以增加 CPU 开销为代价来改进磁盘有效带宽。列式存储可以更好地支持压缩，因为，它把属于同一个属性的值会被保存在一起，这些值具有相同的数据类型，可以获得更高的压缩率。因此，在数据库的物理设计阶段，数据库管理员会频繁使用不同的压缩模式，来优化性能和存储空间。
- 索引：是数据库系统组织管理数据的一种重要方式，可以有效提升数据库查询的性能。
- 预抓取：使用数据库和文件系统预抓取来分摊寻址代价，即数据库存储管理器会预先抓取一些目前还暂时不会立即用到的数据，通常是从磁盘中提前读取额外的与当前数据相邻的一些连续的页。磁盘的预抓取提供了两种好处：第一，一次读取更多的数据，可以把很高的随机读取延迟的代价进行分摊；第二，当数据最终被计算所需要时，它已经在内存中了，这样可以加快计算速度。在现代数据库系统（例如 SQL Server Enterprise）的存储管理器中，可能会预抓取超过 1024 页。
- IO 调度：重新排序、调度和延迟 IO 请求，从而在不同的数据请求之间最小化磁盘延迟。在操作系统内，IO 调度器通过合并、重新排序和延迟请求，来优化底层存储设备的性能。存储设备位于数据库系统存储层次架构的最后一层，也可以优化 IO 调度。对于磁盘而言，控制器可能会根据当前的磁头位置，再次重新排序或者延缓请求，从而改进磁盘性能。

总的来说，在数据库系统的存储层次架构中，包含了在不同层面的不同优化。存储管理器和文件系统都会采用数据预抓取来减少访问延迟，并且通过优化数据布局来减少寻址次数。最后，磁盘调度器和设备驱动会重新排序操作顺序，从而最小化寻址开销。

4.2 设计面向闪存的DBMS的必要性

文件系统的负载通常具有一定的局部性和顺序性，但是对于一个典型的数据库负载而言（比如 TPC-C），在数据访问方面，具有很低的局部性和顺序性，在执行数据库负载时，随机写操作会分散到磁盘的整个物理空间。因此，类似预抓取和写缓冲这些延迟写入技术，对于这种负载类型就没有什么效果。经过多年的发展，磁盘已经成为一种非常成熟的存储技术，具有很大的存储带宽，但是，由于磁盘中存在机械部件，包含盘片转动和磁头移动的寻址开销，因此，磁盘的随机读写操作开销比较大，延迟比较高。可以说，磁盘的较大的随机读写延迟，已经成为基于磁盘存储的数据库应用的性能瓶颈。

基于闪存的存储设备（比如固态盘），相对于磁盘而言，具有很低的读写延迟，前者的读写延迟要比后者小一个数量级，这使其成为可以替代磁盘的新一代存储介质，并且可以明显改进数据库事务的吞吐量。随着闪存技术的不断发展和普及，可以考虑把整个数据库运行在基于闪存的计算平台上，或者把一个嵌入式数据库系统运行在一个轻量级的、基于闪存的

计算设备上。尤其是对于采用闪存的便携式设备而言，为这些设备开发的应用变得越来越复杂，所需要管理的数据量也越来越大，由此，更加需要有一个嵌入式的、基于 SQL 的 DBMS 而不是文件系统来支持应用开发和提供运行时数据管理引擎。当前，几乎所有主流 RDBMS 厂商都为手持设备或嵌入式操作系统（包括 Symbian OS,Palm OS,Window/CE 和嵌入式 Linux），提供了一个移动、嵌入式、轻量级的 DBMS，包括 Oracle Lite、Sybase iAnywhere、Berkeley DB、Microsoft SQL Server 2005 CE 和 IBM DB2 Everyplace 等。基于闪存的 DBMS 具有广阔的应用前景，除了应用于便携式设备，它还可以应用到很多其他领域。比如，微软的 MyLiftBits 项目[GemmellBL06]实践表明，个人信息管理系统需要管理更加复杂的数据类型和查询，闪存可以很好地应用于这类系统中。闪存是可以支持下一代智能卡系统的数据管理需求的有力候选者。此外，无线传感器网络设备也都需要闪存作为数据存储介质，为了提供一些复杂的查询能力，这些设备也需要嵌入式 DBMS 提供数据管理功能[Zeinalipour-YaztiLKGN05]。

但是，当前市场上的各种主流 DBMS 产品都是针对磁盘设备开发的，如果直接应用到闪存存储设备上，将无法获得好的性能，因为闪存的读写特性和磁盘具有很大的区别。首先，传统的 DBMS 在设计上，都尽量避免大量的随机读写，而采用顺序读写来加快速度，但是，对于闪存固态盘而言，顺序读和随机读几乎具有相同的高性能，传统 DBMS 将无法充分利用闪存的这个特性来提升数据库整体性能。其次，传统的 DBMS 都假设读写代价是相同的，但是，这点对于闪存并不成立，闪存的读速度要比写速度高 1 个数量级，因此，传统 DBMS 无法发挥闪存具备较高读性能这个优势。最后，闪存存在“写前擦除”的限制，而擦除操作的延迟通常较长，可以达到 1500ms，传统的 DBMS 并没有考虑到闪存的“写前擦除”问题，很多小数据量的数据库更新操作会带来大量的闪存擦除操作，严重恶化数据库整体性能。比如，对于一个零售商店的商品数据库而言，在每个购物车完成结算以后，都需要去更新售出的各种商品的库存数量，这就需要更新许多个页。如果 DBMS 直接使用了基于 FTL 的闪存存储，就会包含大量的垃圾回收操作，从而严重影响系统整体性能。

可以说，不考虑闪存特性的各种数据结构和算法，是不可能让 DBMS 获得好的性能的。就以查询优化器为例，当前的主流的 DBMS 中，存储介质的特性是在创建表空间的时候就已经确定的，基于代价的查询优化器会根据磁盘的特性来预估查询执行时间，查询优化器也是根据磁盘的访问特性来更好地选择查询执行计划，比如，如果一个设备具有较小的平均延迟，那么一个查询就可以采用读取记录列表的方式执行（即根据一系列不连续的记录 ID 来分别读取不同的记录）；如果一个设备具有较大的平均延迟，就可以采用全表扫描的方式执行查询（即连续读取表中的所有记录）。如果底层采用闪存设备，则查询优化器就无法做出最优的查询执行计划，因为优化器都是假设以磁盘作为存储设备而做出查询计划的，这种为磁盘量身订制的查询计划当然无法在闪存设备上获得好的性能。因此，如果没有设计出结合闪存特性的数据结构和算法，现有的数据库系统是无法充分发挥闪存的优点的。虽然 Lee 等人[LeeMPKK08]用大量实验证明，如果直接用闪存固态盘替换磁盘，那么，即使没有任何的软件修改或者其他优化措施，某些特定的数据库操作的性能也可以改进 2-10 倍。但是，他们在实验测试时的负载并没有包括小数据量的随机写操作(即每个随机写操作只涉及很小的数据量)，而实际情况是，在企业 OLTP 应用中，会存在大量的小数据量随机写操作，而且这种写操作会严重恶化闪存的性能。因此，如果闪存设备要最终替换磁盘，包括小数据量随机写在内的许多问题都是不可忽视的，必须有相应的解决方案。

因此，设计面向闪存的 DBMS 具有很强的必要性和重要性，使得我们需要重新审视现有的许多 DBMS 实现技术。目前，已经有一些研究提出了一些比较简单的闪存数据库原型系统，比如 PicoDBMS[PucheralBVB01]、LGeDBMS[KimBLLJ06]和 FlashDB[NathK07]等。PicoDBMS 是一种面向 Smartcard 的小型数据库系统，它的结构和功能都比较简单，为了能

够应用到资源受限的 SmartCard 环境中，PicoDBMS 对关系按照列的方式进行分解，并采用了基于指针的存储模型来降低数据存储量。LGeDBMS 是一个应用于基于闪存的嵌入式设备上的、微缩版本的 DBMS，它借鉴了基于日志的文件系统[RosenblumO92]的设计原理，对闪存空间进行直接管理，它把整个数据库看成是一个单独的日志，所有的数据库更新操作都会被顺序地追加到日志的尾部。在提供存储管理能力的基础上，LGeDBMS 同时也提供了一定的查询处理和恢复功能。LGeDBMS 还充分考虑了嵌入式设备应用环境中事务执行频率较低且无并发的特点，采用传统的影子页技术来实现简单的事务处理和恢复功能。FlashDB 是一个为传感器网络而优化的、自适应的、基于闪存的数据库，提供了存储管理、缓冲区管理和查询处理等功能。FlashDB 使用了一个新的、自适应的 B+-树索引，可以动态地根据负载和底层的存储设备来调整它的树结构。

总体而言，闪存数据库系统的研究具有很强的必要性，但是，当前的研究还只是处在初级阶段，只出现了一些简单的原型系统（比如上面论述的 PicoDBMS、LGeDBMS 和 FlashDB），尚无可以大规模商用的成熟产品出现，因此，该研究领域在未来具有很大的发展空间[MengJCY12]。

4.3 闪存特性对DBMS设计的影响

闪存主要包括三个方面的明显特性：（1）异地更新；（2）没有机械部件；（3）读写速度不对称。闪存的这些特性对 DBMS 各个模块设计都会产生影响，包括 IO 单位、数据分簇、缓冲区、页布局、索引、查询处理和优化以及事务管理。

4.3.1 对 IO 单位的影响

在一个操作系统或者 DBMS 中，IO 操作是以块为单位。随着内存大小、CPU 处理能力和 IO 带宽的不断增加，IO 操作单位的大小一直在持续增加。例如，默认的块大小是 16KB，在一些读操作比较集中的负载中比如数据仓库，一般推荐采用 64KB 的块大小。但是，对于闪存数据库而言，需要重新审视 IO 操作最优块大小的问题，因为，在闪存数据库中采用一个较大的块作为 IO 单位不一定是最好的。一般而言，采用较大的块作为 IO 操作单位的理论基础是，采用较大的块可以帮助实现数据预抓取，加快数据读取速度。但是，在闪存中数据预抓取策略变得不再重要，相反，一个较大的块会引起缓冲区空间的浪费。更加严峻的问题是，如果对较大的块执行小数据量的更新操作，会引起擦除操作。

4.3.2 对页布局的影响

绝大多数基于磁盘的 DBMS 都支持基于记录的页布局（或称为行式页布局），即一个记录的各个属性会被连续地存储在页中。这种页布局对于 OLTP 而言可以获得很好的性能，因此，在以 OLTP 应用为主的商业时代，这种页布局方式得到了广泛的应用。但是，对于一个闪存数据库而言，采用这种页布局方式可能会导致一些性能问题。闪存数据库的一个技术难题就是，如何高效地处理针对闪存的小数据量的随机写操作，因为数据库负载中包含大量这种只涉及很少数据量的随机写操作。小数据量随机写操作会引起代价高昂的擦除操作，恶化闪存数据库的整体性能。针对元数据的更新操作，是小数据量随机写操作的一个重要来源。比如，一个数据块通常会包含一个头结构和记录区域。任何记录更新（包括插入、删除和修改），都需要改变头结构。在闪存数据库中，这就意味着需要为头结构执行很多擦除操作。

4.3.3 对数据分簇的影响

磁盘设备包含机械部件，需要在不同位置之间移动磁头来读取数据，频繁的磁头移动会严重降低数据读取的性能，因此，在基于磁盘的 DBMS 中，为了加快数据读取速度，通常采用数据分簇技术，即把相关的数据都存储在磁盘中相邻的位置，这样就可以最小化磁头移动距离，很容易实现对相关数据的批量预抓取。

由于可以帮助实现预抓取，因此，分簇技术可以有效提高磁盘数据库中的读操作性能。但是，在闪存数据库中，数据预抓取的作用已经变得不再重要，因为闪存读操作非常快，不需要通过提前抓取数据来提高读操作速度。相反，对于闪存而言，写操作和擦除操作的性能很低，写操作速度要比读操作慢一个数量级，擦除操作速度要比读速度慢两个数量级。因此，对于闪存而言，面向写操作的数据分簇技术可能更加合适，也就是把相关的对象尽量写入到同一个物理块中。

4.3.4 对查询处理和优化的影响

在基于磁盘的 DBMS 中，哈希连接和排序合并连接方法，可以比嵌套循环连方法接取得更好的性能。这是因为，磁盘中的读写操作速度是相同的，并且有足够的 RAM 来运行哈希或排序合并连接算法。在采用哈希连接或者排序合并连接方法时，需要把中间结果写入到磁盘中，而嵌套循环连接则采用了只读的工作模式，不需要为连接操作开辟额外的辅助存储器存储空间。因此，当采用闪存作为辅助存储器时，嵌套循环连接算法会比哈希连接和排序合并连接方法更加合适；而且，如果嵌套循环连接算法的内关系上存在索引，那么，就可以获得更好的性能。对于一个给定的查询而言，DBMS 中的查询优化器会枚举不同的执行计划，评估每个执行计划的代价，并且选择代价最小的计划来执行查询。既然从磁盘更换到闪存以后，不同连接算法的相对代价会发生变化——磁盘中更适合采用哈希、排序合并连接而闪存中更适合采用嵌套循环连接，那么，就需要重新审视查询优化技术中的代价评估机制。

查询优化技术中的代价评估机制需要重新审视的另外一个方面，就是索引扫描的代价。在闪存中，索引扫描更适合采用全表扫描，因为闪存具有一致的随机访问速度和快速的读取时间。这就意味着，闪存中访问一个索引的代价要比磁盘中低许多，而且我们可以更加准确地评估它的代价，因为闪存具有一致的随机访问速度。

4.3.5 对缓冲区替换策略的影响

在操作系统或者 DBMS 中使用的缓冲区替换算法，通常假设读操作和写操作的速度是一致的。但是，这种假设在闪存中不再成立，因为，闪存的读写速度不对称，这就意味着，在闪存数据库中，传统的性能评价指标，比如缓冲区命中率，就不再适用，需要重新设计面向闪存的缓冲区替换算法。传统的替换策略只考虑缓存的命中率，即在给定的缓冲区尺寸下，最小化缓存脱靶的概率。但是，最小化缓存脱靶率，在基于闪存的数据库系统中，可能会带来较差的 I/O 性能。这是因为，和磁盘相比，闪存有一个内在的特性——读写不对称性，即写操作会比读操作慢一个数量级，而且由于写操作可能会引起擦除操作，这就进一步提高了写操作的代价。由于这种非对称性，缓存管理策略必须对写操作和读操作进行区分。面向闪存的缓冲区替换算法一般采用的思路是：尽量减少写和擦除操作的数量，哪怕是以增加读操作数量为代价。

4.3.6 对索引的影响

在基于磁盘的 DBMS 中，会使用 B-树索引来支持对列值进行有选择性地快速访问。当被索引的列值发生变化、新的记录插入、已有的记录被删除时，索引树的节点必须被更新，这就会引起索引节点的分裂或者合并，其中会涉及许多小数据量的随机写操作。由于闪存采用异地更新的方式，因此，每个 B-树中节点的更新，都会引起闪存写操作，而且，某个节点的更新可能会不断传播扩散到其他节点，引起其他节点的更新，从而导致更多的闪存写操作。因此，必须针对闪存特性设计相应的索引结构。

4.3.7 对事务管理的影响

事务管理有两个关键的组件，即并发控制和恢复。对于闪存数据库中的并发控制而言，快照隔离方法可能比较合适，而不是采用加锁技术。在快照隔离方法中，会同时维护同一个数据的多个版本，来满足不同查询的要求，因为，在不同时间点的查询需要访问不同的快照。闪存可以有效地支持这种并发控制模型，因为闪存具有异地更新的特性，更新后原来的旧版

本数据不会被立即删除。闪存异地更新的特性，允许同一个数据可能存在多个版本，我们可以被分利用这点来替代传统的基于日志的恢复机制。也就是说，每个版本都可以看成是某种形式的日志。

4.4 面向闪存的DBMS的设计考虑因素

面向闪存的 DBMS 的设计，必须克服闪存的缺陷，并充分利用闪存的独特特性，从而使得面向闪存的 DBMS 可以获得更好的性能，主要需要考虑以下几个方面的因素：

- 读写速度不对等：闪存的读速度要比写速度快许多，因此，在设计 DBMS 时，应该重点考虑减少写操作。
- 一致的随机访问速度：对于闪存而言，由于不存在可移动的机械部件，因此，随机访问和顺序访问的速度是一致的，对于随机访问没有长时间的寻址开销，因此，没有必要再执行顺序读操作。这个特性也会影响到一些设计上的策略，比如在设计数据对象和日志在闪存中的存储位置的时候可以更加自由。文献[LeeM07]在设计 IPL 时，就是利用了闪存的这个特性，把日志记录分散存储到闪存中的任何位置，而没有采用顺序存储。
- “写前擦除”的限制：这是设计 DBMS 时需要考虑的关键因素。在闪存中，如果数据库系统还坚持采用就地更新的方式，那么，由于存在“写前擦除”的限制，就必须首先把闪存中包含该更新的整个擦除单元读取到内存中，在内存中完成就地修改，然后把该擦除单元进行完整的擦除操作，最后，把内存中的修改后的内容写入到这个擦除单元中。这种就地更新的方式，代价太高，这就要求在闪存中必须采用异地更新的方式。
- 读操作代价比写操作和擦除操作代价低很多：这个闪存特性要求在设计 DBMS 时，必须尽量有效减少写操作的数量，从而减少擦除操作的数量，即使这可能会导致读数量的增加。因为，读操作代价要比写操作和擦除操作的代价低很多，在减少写和擦除操作的同时，尽管会带来读操作的增加，但是，最终的整体性能仍然可以获得提高。
- 即刻代价和延迟代价问题：针对一个闪存页的更新操作，由于闪存采用了异地更新的方式，会把数据页写入到一个新的闪存页中，让包含旧数据的页失效。写入到新的闪存页就会引起一个即刻代价，也就是说，这个代价在写操作发生的时候就产生了。但是，同时要注意到，包含旧数据的无效页，不会被立即擦除，它会在以后的垃圾回收过程中被执行擦除操作，在那时才会产生代价，因此，可以认为，这是一个由于更新操作引起的延迟代价。因此，在设计算法时，不仅要考虑即刻代价，也要充分预估到由此产生的延迟代价。
- 随机写操作严重恶化闪存性能，应该尽量改变 DBMS 的负载模式。首先，应该尽量避免在闪存上面进行随机写操作，即使以增加读操作或其他额外开销为代价。其次，所有的写操作都应该具有足够大的尺寸，从而在每个 IO 操作中，可以完全覆盖一个或多个擦除块。
- 对现有的 DBMS 体系架构的影响最小化：当前主流的基于磁盘的 DBMS 已经是一个非常成熟的技术，产品的各种功能（比如查询编译、事务机制、灾难恢复和性能优化等）已经相当完备，面向闪存的 DBMS 在设计时，应该充分继承和利用已有的架构和技术，减少设计工作量，充分发挥已有成熟技术的优势，尽快获得成熟的产品。而且，当前主流的基于磁盘的 DBMS，都采用了模块化的设计，对其中的部分模块（比如存储模块）进行替换是可以实现的。

4.5 设计面向闪存的DBMS的技术路线

通过对大量相关研究的综合分析，我们总结得到，为了提高 DBMS 在闪存设备上的性能，目前基于闪存的 DBMS 的设计所采用的技术路线主要包括以下几个方面：

（1）**设计面向 DBMS 的 FTL**。直接采用原来的针对磁盘设计开发的 DBMS，在 DBMS 和闪存存储二者之间，增加一个中间软件层 FTL，在充分考虑数据库负载特性和闪存特性的基础上，设计符合这两个特性的、面向 DBMS 的 FTL 机制。这类方法的典型代表是 DFTL 机制[GuptaKU09]。

（2）**采用 FTL 并修改面向磁盘的 DBMS 的部分模块**。继续采用中间软件层 FTL 来访问闪存设备，但是，对面向磁盘的 DBMS 的部分模块（比如索引）进行修改，从而获得更好的性能。这类方法的典型代表是基于日志的 B-树索引[WuCK03]、B+-树索引[OnHLX09][WuKC07]和采用自适应 B+-树的 FlashDB[NathK07]。

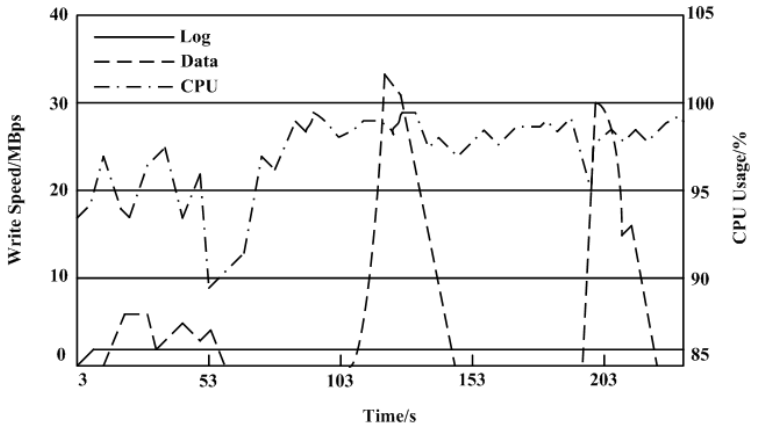
（3）**抛弃 FTL 并修改面向磁盘的 DBMS 的部分模块**。抛弃中间软件层 FTL，直接访问闪存，对原来面向磁盘的 DBMS 的部分模块（比如缓冲区管理和存储管理模块）进行重新设计，使其能够在闪存上发挥最好的性能。目前主流的商业 DBMS 都采用了模块化的设计，对其中一些模块进行替换是可行的。这类方法的典型代表是基于日志的方法（比如 IPL[LeeM07]）和基于影子页的方法[LeeKWCK09]。在 IPL 方法中，作者设计开发了 IPL 模拟器，实现了 DBMS 中的缓冲区和存储管理模块的功能。在基于影子页的方法[LeeKWCK09]中，作者设计的 DBMS 原型系统被称为“AceDB Flashlight”，它就是对 AceDB（一个面向磁盘的 DBMS）的事务模块和存储管理模块进行了修改。

（4）**完全重新设计面向闪存的 DBMS**。抛弃中间软件层 FTL 和原来面向磁盘的 DBMS，完全重新设计面向闪存的 DBMS。由于设计和实现代价过高，开发周期长，不利于在短期内把 DBMS 迅速部署到闪存平台上，因此，目前没有相关研究采用这类方法。

4.6 进一步讨论

当数据库采用磁盘作为存储介质时，磁盘 IO 性能往往成为整个系统的瓶颈。闪存比磁盘具有更高的 IO 性能，在数据库系统中会被越来越多地作为底层的存储介质。由于闪存 IO 的高性能，可能某些情况下系统的性能瓶颈会发生转移，即闪存 IO 将不再是瓶颈，而 CPU 则会成为系统的主要瓶颈。

文献[LvCC09]用实验验证了这种瓶颈转移的可能性。作者采用 TPC-B 测试基准和 SQL Server 2008 进行了数据库负载测试，图[cuibin-04]显示了数据和日志文件的实时写入速度以及 CPU 占用情况。从中可以看出，在整个数据库负载运行过程中，日志写入操作一直贯穿始终，速度变化很小。数据文件的写入操作则具有突发性的特点，而且写入量比较大，这主要是由数据库缓存回写数据造成的。当数据写入量较小时，CPU 占用率很高，因为，这时 CPU 在集中执行事务操作。当写入操作执行时，CPU 占用率会下降。由此可以看出，当采用闪存存储时，在数据库运行过程中，如果存在较多的小数据量的写操作，写操作会快速完成，闪存 IO 不会是系统的瓶颈，但是，这种小数据量写操作会大大增加 CPU 占用率，导致 CPU 称为系统的性能瓶颈。



图[cuibin-04] 数据和日志的实时写入速度以及 CPU 占用情况

因此，在闪存数据库系统中，闪存和 CPU 之间可以互相促进，共同提高系统整体性能。一方面，CPU 性能每年都在不断提升，使用闪存可以更好地发挥出 CPU 的高性能。另一方面，在设计数据库时，应该尽量优化事务处理过程，降低 CPU 负载。

4.7 本章小结

本章内容首先介绍了基于磁盘的 DBMS 的存储性能优化技术，然后指出了设计面向闪存的 DBMS 的必要性；接下来，阐述了闪存的不同特性对 DBMS 设计的影响，包括对 IO 单位、页布局、数据分簇、查询处理与优化、缓冲区替换策略、索引和事务管理等方面的影响；然后，又介绍了面向闪存的 DBMS 的设计考虑因素，包括读写速度不对等、一致的随机访问速度、“写前擦除”的限制、读操作代价比写操作和擦除操作代价低很多、即刻代价和延迟代价问题、应该尽量改变 DBMS 的负载模式以及对现有的 DBMS 体系架构的影响最小化等等；最后，介绍了设计面向闪存的 DBMS 的技术路线，包括设计面向 DBMS 的 FTL、采用 FTL 并修改面向磁盘的 DBMS 的部分模块和抛弃 FTL 并修改面向磁盘的 DBMS 的部分模块等等。

4.8 习题

1、 说明设计面向闪存的 DBMS 的必要性。
2、 分别阐述闪存特性对 DBMS 设计的影响，包括对 IO 单位、页布局、数据分簇、查询处理和优化、缓冲区替换策略、索引和事务管理等方面的影响。
3、 阐述在设计面向闪存的 DBMS 时需要考虑的因素。
4、 阐述基于闪存的 DBMS 的设计所采用的技术路线。

第 5 章 闪存数据库存储管理

和磁盘相比，闪存设备形状小，能耗低，抗震好，而且，由于闪存中不存在可转动的机械部件，因此，闪存设备具有更快的随机读速度。但是，闪存的写操作性能不具备明显的优势，尤其是随机写操作，会表现出较差的性能，因为闪存存在“写前擦除”的限制。频繁的随机写操作，会引起大量的擦除操作，不仅会严重恶化闪存写操作性能，还会大大缩短闪存的寿命。因此，面向闪存数据库的存储管理的主要任务，就是优化写操作模式，减少频繁随机写操作对闪存的影响。

目前已经有许多方法被提出来，用来优化闪存设备的写操作性能。由于把数据库中的数据存储到闪存中可以分为两种方法，即基于页的方法和基于日志的方法，因此，面向闪存数据库的存储管理方法也主要包括两大类：基于页的方法和基于日志的方法。

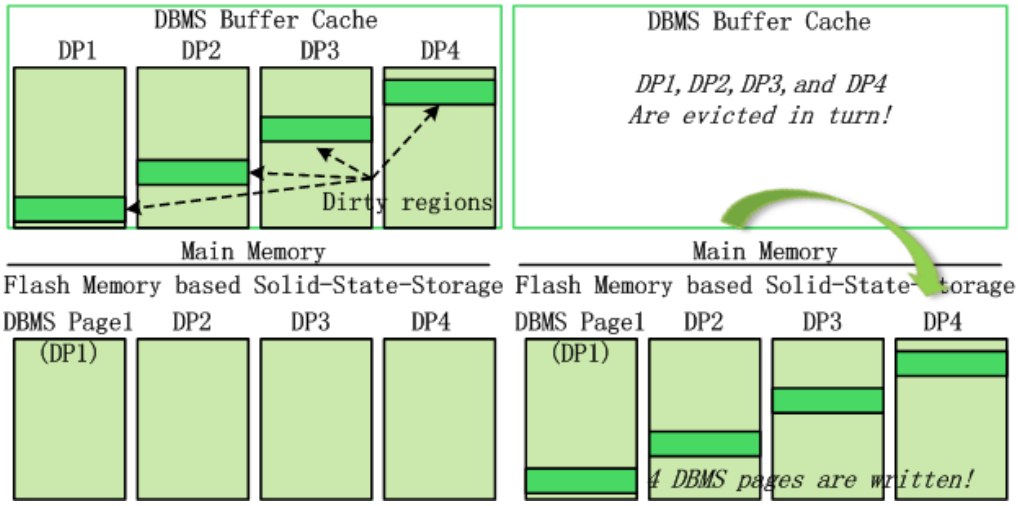
本章首先给出数据存储方法概述，然后分别介绍基于页的方法和基于日志的方法及其代表性研究，最后介绍两种其他方法，即基于页差异的日志方法和 StableBuffer 方法。

5.1 数据存储方法概述

把数据库中的数据存储到闪存中，可以采用两种方法：基于页的方法和基于日志的方法。

5.1.1 基于页的方法

在基于页的方法中[Ban95][LeePCLPS07][LeeSKK08]，一个逻辑页会被存储到一个物理页中。内存中的逻辑页会不断接受更新，当一个更新后的逻辑页需要被刷新到闪存中时，整个逻辑页（包括变化的数据和没有发生变化的数据）都会被写入到一个物理页中，因此，基于页的方法的写操作性能比较差。当需要重构一个逻辑页时——把该页从闪存中读取到 DBMS 内存缓冲区中，就可以根据 FTL 中逻辑页到物理页的地址映射表，找到闪存中和该逻辑页对应的那个物理页，读取到内存中，也就是说，重构一个逻辑页时只需要读取一个物理页，因此，基于页的方法具有很好的读操作性能。



图[page-based-method] 基于页的方法示意图

图[page-based-method]演示了基于页的方法的基本过程。在初始阶段，闪存固态盘中存在 4 个 DBMS 数据页 DP1,DP2,DP3,DP4，这 4 个页被读入内存的 DBMS 缓冲区中。随后，这 4 个页发生了更新，变成脏页。当需要把缓冲区内容刷新到闪存中时，需要把 4 个页驱逐出缓冲区，刷新到闪存中，因此，缓冲区中的 4 个 DBMS 数据页，会被依次驱逐出缓冲区，固态盘上的 4 个页会被依次更新。总体而言，这个过程更新了 4 个数据页，导致了 4 个数据页回写到闪存的写操作。

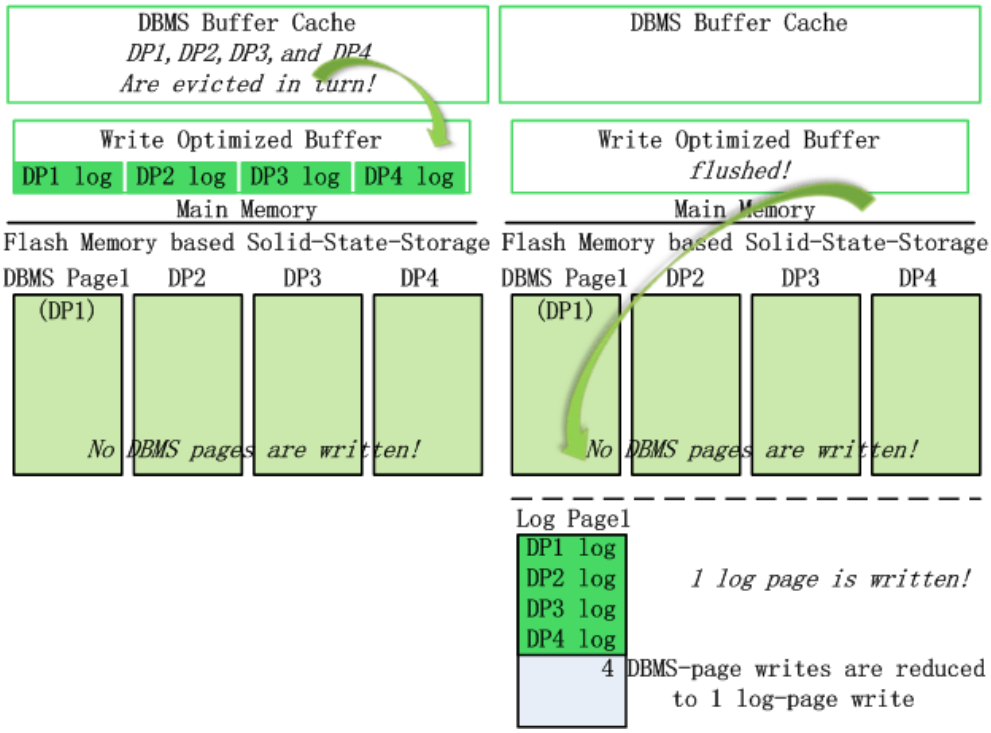
基于页的方法和 DBMS 的存储管理模块是松耦合的，不需要修改存储管理模块，而是

在 DBMS 和闪存之间增加一个中间层 FTL 来实现，FTL 会负责维护从逻辑页到物理页之间的地址映射。FTL 可以作为固态盘控制器中的一个硬件来实现，也可以作为操作系统的一个软件来实现。

在基于页的方法中，有两种类型的更新机制[NathG10]——就地更新和异地更新，二者的主要区别在于逻辑页是否总是被写入到存储介质的同一个物理地址中。当一个逻辑页需要被刷新到存储介质中时，对于就地更新而言，逻辑页会写入到存储介质中它原来被读取的位置，但是，对于异地更新而言，逻辑页会写入到一个新的物理页中。对于闪存而言，一般采用异地更新方式。

5.1.2 基于日志的方法

在基于日志的方法中，一个逻辑页通常被保存到多个物理页中。一旦逻辑页被更新，这些更新对应的日志就会首先被放入到内存的写缓冲区中。当内存的写缓冲区满的时候，这些更新会被写入到一个物理页中。因此，当一个逻辑页被多次更新时，它的更新日志可能会被保存到多个物理页中。相应地，当需要创建一个逻辑页时，需要从存储介质中读取多个物理页，然后进行合并得到一个逻辑页。



图[log-based-method] 基于日志的方法示意图

图[log-based-method]演示了基于日志的方法的基本过程。在初始阶段，闪存固态盘中存在 4 个 DBMS 数据页 DP1,DP2,DP3,DP4，这 4 个页被读入内存的 DBMS 缓冲区中。随后，这 4 个页 DP1,DP2,DP3,DP4 发生了更新，更新操作会被分别记录到日志 DP1Log, DP2Log, DP3Log，DP4Log 中。当需要把缓冲区内容刷新到闪存中时，只需要把日志 DP1Log, DP2Log, DP3Log，DP4Log 依次写入到闪存中的日志页中即可，缓冲区中的数据页 DP1,DP2,DP3,DP4 直接被删除，不需要写入到闪存。因此，这个过程虽然更新了 4 个数据页，但是，并不需要把 4 个数据页回写到闪存，而只需要向闪存中写入一个日志页，这就大大减少了闪存写操作的数量。

基于日志的方法和 DBMS 存储模块是紧耦合的，必须对 DBMS 存储模块进行修改，从而能够探测到一个逻辑页的更新，因为，只有 DBMS 内部的存储模块才能够及时探测到数据更新。

基于日志的方法，只会把一个逻辑页中发生变化的数据以更新日志的形式写入到缓冲区中，当缓冲区满时被刷新到闪存中。因此，和基于页的方法相比，当更新操作不是非常频繁时，基于日志的方法具有较好的写操作性能，但是，当更新操作非常频繁时，性能会降低许多，甚至还不如基于页的方法。此外，基于日志的方法的读操作性能较差，因为，当需要重构一个逻辑页时，由于更新日志很可能分散在多个不同的物理页中，这就需要从闪存中读取多个物理页进行合并得到一个逻辑页。

典型的基于日志的方法包括 LFS(Log-structured File system) [RosenblumO92], JFFS (Journaling Flash File System) [Woodhouse01], YAFFS (Yet Another Flash File System) [Yaffs]和 IPL(In-Page Logging) [LeeM07]，其中 JFFS、YAFFS 和 IPL 是面向闪存的方法。在 LFS、JFFS 和 YAFFS 中，一个逻辑页的更新日志可以被写入到闪存中任意的日志页中。而在 IPL[LeeM07]中，更新日志必须被写入到闪存中特定的日志页中。

5.2 基于页的方法

基于页的方法和存储系统是松耦合的，它们通常借助于一个中间层 FTL 来实现，FTL 会负责维护从逻辑页到物理页之间的地址映射。但是，现有的闪存 FTL 机制都是面向文件系统的，如果直接应用于 DBMS，将无法发挥好的性能。因此，在闪存数据库应用中，必须研究面向 DBMS 的 FTL 机制。

5.2.1 面向文件系统的 FTL 机制无法直接应用于 DBMS

FTL 机制可以隐藏闪存的特性，让固态盘看起来像是一个虚拟的磁盘。因此，通过使用 FTL 方法，一个传统的基于磁盘的 DBMS 可以直接运行在固态盘上，而不需要任何修改。FTL 机制提供了逻辑页到物理页的映射，在接收到来自 DBMS 发出的写请求以后，FTL 会把该写请求导向到闪存中的一个处于擦除状态的空闲页上去执行；收到 DBMS 发出的读请求时，FTL 会查找地址映射表，找到闪存中的物理页并读取数据。

在前面的第 2 章内容中，介绍了许多 FTL 机制[ChungPPLLS06][LeePCLPS07] [KimKNMC02] [KangJKL06]。但是，其中很多 FTL 机制并非是针对数据库负载模式而开发的，而是必须具备足够的通用性，从而能够支持各种类型的应用（尤其是文件系统）。这类主要面向文件系统的 FTL 算法如果直接应用到数据库负载上，将无法获得很好的性能。原因主要包括以下几个方面：

- DBMS 的负载和文件系统的负载具有很大的区别，前者具有更多的随机写操作。在许多情形下，文件系统的 IO 模式主要是读操作比较集中的负载，并伴有不频繁的文件顺序写操作，并且会表现出较好的空间和时间上的数据局部性。相反，数据库负载，尤其是 OLTP 负载，通常是随机分布的、很小数据量的写操作。虽然面向文件系统的 FTL 机制也考虑了随机写操作，但是，这些研究的重点还是在于从负载中区分出随机操作和顺序操作，然后对顺序操作进行重点性能优化。此外，虽然面向文件系统的 FTL 机制对随机写操作也设计了相应的优化策略，但是，文件系统中的随机写操作大都只是针对元数据，而且往往会表现出空间的局部性（LAST 和 SuperBlok 等机制就充分利用了这个特性），而在执行典型的数据库负载时，DBMS 所执行的写操作会分散到磁盘的整个物理地址空间[LeeM07]，由此也会带来更多的擦除操作。
- MLC NAND 闪存中存在特有的“耦合页问题”（paired page problem）[LeeKWCK09]。在 MLC NAND 闪存中，同一个存储单元中的多个位，会属于不同的页。对于一个两位的 MLC NAND 闪存而言，在文献[LeeKWCK09]中，作者把包含了来自同一个存储单元中的位的两个页称为“耦合页”。所谓的“耦合页问题”是指，在改变一个存储单元的某个位的状态时，如果发生断电，那么，这个存储单元中的其他位也会改变。这就意味着，在对一个页执行写操作的过程中如果发生了断电，不仅该页的写

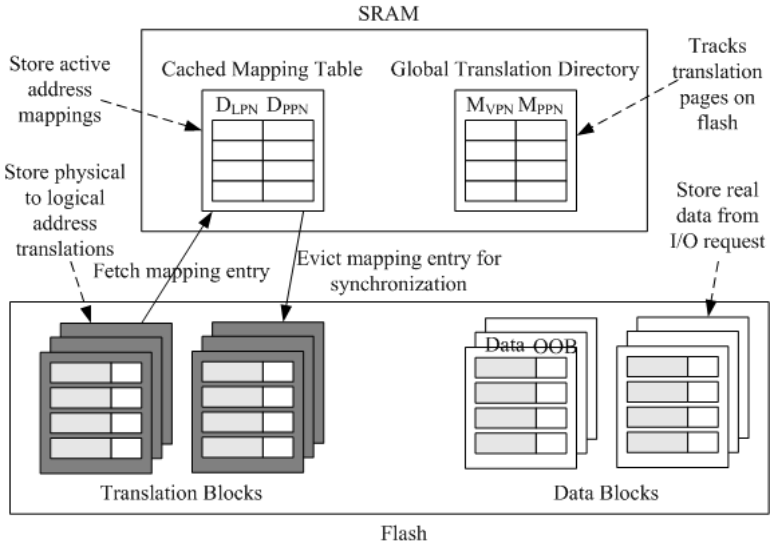
操作可能执行失败，由其他已经完成的写操作所写入到闪存中的其他页，如果与该页存在耦合页关系，也可能会被损坏，这破坏了数据库写操作的持久性。传统的针对磁盘的 DBMS 都可以保证写操作的持久性，也就是说，一旦写操作已经完成提交，此后即使发生了断电等意外情况，该写操作写入的结果都会被持久地保存。由于传统的 DBMS 是针对磁盘而不是闪存设计的，因此，也就不会考虑 MLC NAND 闪存中的“耦合页问题”（paired page problem）[LeeKWCK09]。因此，如果直接应用到闪存平台上，传统的 DBMS 就无法保证写操作的持久性，这会大大降低数据库系统的可用性。

- 为了减少擦除操作的次数，FTL 机制通常采用异地更新的方式，即对某数据项进行更新时，会把更新操作写到到新的空闲页中，并简单地把该数据项所在的原来的页设置为“无效”，旧数据就会被保留在原地，等到启动垃圾回收过程时才会被统一擦除。而对于传统的基于磁盘的 DBMS 而言，为了实现事务机制和灾难恢复机制，一般都会在执行更新操作的同时，把旧版本的数据记录在日志中。因为，这些基于磁盘的 DBMS 在设计时并不是针对闪存的特性设计的，不知道旧版本的数据可能仍然被保存在闪存的“无效”页中。由此带来的问题是，旧版本数据的重复存储，这既增加了不必要的写操作，也浪费了宝贵的闪存空间。
- 数据库负载中包含了大量针对元组的更新操作，对于比较常用的混合 FTL 机制而言，每次更新操作都需要记录到日志块中，由于一个写操作的最小单元是一个页，因此，每次更新都要记录到一个空闲页中，由此带来的闪存空间开销也是很大的。而且，数据库元组的数据量通常都很小，每次对元组的更新都用空闲页来记录，也浪费了大量的闪存空间。

就目前的技术发展趋势来看，企业的大量数据库应用将会越来越多地部署到基于固态盘的存储系统上面。为了获得更好的数据库性能，非常有必要在成熟的、针对文件系统的 FTL 机制基础上，进一步研究针新的、对数据库负载的 FTL 机制。本书前面用大量篇幅介绍 FTL 机制，也是希望这些 FTL 基础研究的介绍能够有助于研究人员更好地设计开发面向 DBMS 的 FTL 机制。

5.2.2 面向 DBMS 的 FTL 机制

数据库负载的一个典型特性就是包含大量随机写操作，采用面向文件的 FTL 机制会带来大量的完全合并操作，需要高昂的垃圾回收代价。Gupta 等人受到 Translation Lookaside Buffer(TLB) [HennessyP03]的启发，提出了完全采用页级别映射的 DFTL[GuptaKU09]。DFTL 的核心思想很简单：绝大多数企业级别的工作负载，虽然包含大量分布在整个闪存物理空间的随机写操作，但是往往会表现出时间局部性，DFTL 就是充分利用了闪存上有限的 SRAM 来存储最常用的映射，而其他比较少用的映射则保存在闪存上。



图[DFTL] DFTL 机制

图[DFTL]给出了 DFTL 机制的设计方法。DFTL 完全废除了日志块，只在闪存空间中设计了数据块和转换块。数据块中的“数据页”存储了数据，在读写操作期间会被不断更新；转换块中的“转换页”存储了逻辑地址到物理地址的映射信息。转换块大约只占到 0.2%的闪存空间，从来不和数据块进行合并。整个逻辑地址到物理地址的映射集合，通常保存在闪存中的转换页内，被称为全局映射表。全局映射表通常较大，无法一次性放入 SRAM，DFTL 只把当前使用到得映射信息存放到 SRAM 中，这部分活跃的映射表被称为缓存映射表，每个映射表条目包含逻辑数据页编号 D_{LPN} 和物理数据页编号 D_{PPN}。随着数据更新的不断发生，逻辑地址到物理地址的映射信息会不断发生变化，就需要修改转换页，而闪存实行的是异地更新，因此，各个新的转换页就会被分散存储到闪存中的不同位置。为了跟踪这些分散存储的转换页，DFTL 设计了全局转换目录，并且保存在 SRAM 中，每个转换目录条目包含虚拟转换页编号 M_{VPN} 和物理转换页编号 M_{PPN}。当外部读写请求到达时，如果请求的地址信息已经保存在 SRAM 中，就可以直接利用这个映射信息找到物理地址，到闪存中去读取数据。如果信息不存在于 SRAM 中，那么就需要从闪存中读取映射信息到缓存映射表中。随着读写操作负载的不断到达，读写操作所涉及的数据的地址空间分布也会不断演化，这就需要一些替换算法把 SRAM 中的一些映射条目替换掉，DFTL 采用了分段 LRU 数组缓存算法。

DFTL 具有以下几个方面的良好特性：（1）彻底消除了混合映射机制中存在的、代价高昂的“完全合并”操作，改进了随机写操作的性能；（2）充分利用了页级别的时间局部性来存储页面，这些页会被放在同一个数据块中被一起访问，这种机制就内在具备了对冷块和热块的区分，而不需要额外的机制来区分冷块和热块，因此可以更好地适应负载类型的变化；（3）更新操作可以在任何数据块上进行，因而提高了块利用率。作者利用一个金融机构的 OLTP 应用负载（包含大量随机写操作）进行测试，实验结果显示，平均响应时间改进了 78%左右。作者还利用以读操作为主的 TPC-H 测试基准进行测试，实验结果显示，DFTL 虽然引入了额外的开销，但是，仍然在平均响应时间方面改进了 56%。

DFTL 机制的缺点是：虽然高效，却面临一个严重的可靠性问题，因为，如果系统发生崩溃，那么，所有在 SRAM 中被修改的信息将会丢失。在这种情形下，所有数据页中的备用区域，都需要被扫描，直到系统恢复到一个一致性状态。而且，随着闪存容量和密度的增加，这个重启延迟会变得令人无法接受。因此，当闪存要作为一个永久可靠的存储设备来用的时候，DFTL 是不合适的。

5.3 基于日志的方法

各种面向闪存数据库的、基于日志的方法，都借鉴了已有的各种日志文件系统的原理（比如 LFS、JFFS 和 YAFFS）。下面首先介绍日志文件系统的原理，然后介绍几种面向 DBMS 的、基于日志的方法，包括 LGeDBMS、A/P 方法和 IPL 方法。

5.3.1 日志文件系统原理

日志文件系统根据原理可以划分成两大类：第一类是基于日志结构的日志文件系统，第二类是基于元数据的日志文件系统。表中给出了两种日志文件系统的特点比较。

表[log-file-system-comparison] 两种日志文件系统的特点比较

	实现原理	优点	缺点
基于日志结构的日志文件系统	在磁盘上创建一个日志，并且跟踪记录文件内容的变化，任何文件的修改操作，都会以日志的形式追加写入到日志文件的尾部	当发生系统崩溃时，可以直接查找日志文件的尾部日志记录来处理崩溃	实现复杂，读性能差，存储空间的垃圾回收代价较大
基于元数据的日志文件系统	在磁盘上使用一个单独的区域来记录文件内容的更新，而且只会记录针对元数据的更新	日志是对正常的文件系统的扩展，容易实现；当系统发生崩溃时，只需要扫描少量日志记录就可以迅速恢复	当发生系统崩溃时，没有办法避免缓冲区内没有回写到磁盘的数据发生丢失的问题

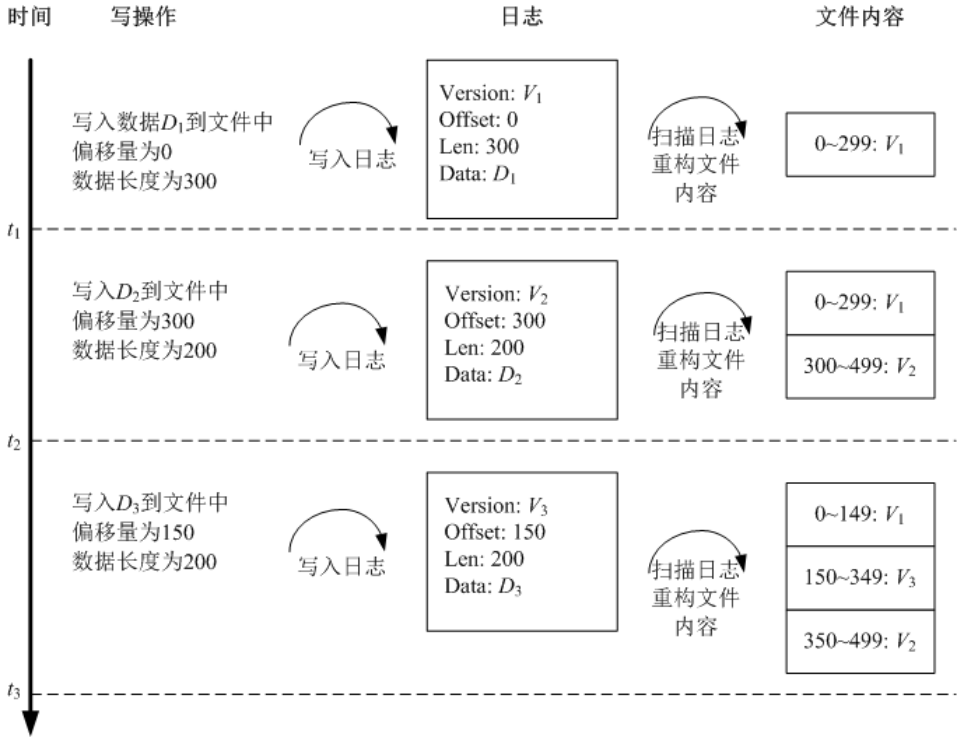
上述两种日志文件系统中，基于日志结构的日志文件系统，采用了追加的方式写入日志记录，这种顺序写入方式比较适合闪存“异地更新”的特性，可以在闪存数据管理中得到较好的应用，因此，下面论述中的日志文件系统，仅指基于日志结构的日志文件系统。

基于日志结构的日志文件系统[RosenblumO92]采用日志的方式对文件进行管理，它会在磁盘上创建一个日志，并且跟踪记录文件内容的变化，任何文件的修改操作，都会以日志的形式被记录下来，即每次文件修改操作都会产生相应的日志，这些日志采用追加的方式写入磁盘中的日志文件，即每次都在日志文件的末端写入新的日志记录。通过遍历日志，就可以获得日志对应的文件的内容。

图[log-file-system]解释了日志文件系统的原理。假设这里按照时间顺序先后对文件内容进行了三次写操作，每次写操作都会产生一条日志，用来记录本次写操作的所有相关信息。下面按照时间顺序进行简单分析：

- 在 t_1 时刻，数据 D_1 已经被写入到文件中，并产生了相应的日志记录（版本号为 V_1），这时通过扫描日志记录，可以重构得到文件的内容，即地址空间 0~299 里面的内容都来自版本号为 V_1 的日志记录中的数据。
- 在 t_2 时刻，数据 D_2 已经写入到文件中，并产生了相应的日志记录（版本号为 V_2），通过扫描日志记录，可以重构得到 t_2 时刻文件的内容，即地址空间 0~299 里面的内容都来自版本号为 V_1 的日志记录中的数据，而地址空间 300~499 里面的内容则来自版本号为 V_2 的日志记录中的数据。
- 在 t_3 时刻，数据 D_3 已经写入到文件中，并产生了相应的日志记录（版本号为 V_3），通过扫描日志记录，可以重构得到 t_3 时刻文件的内容，即地址空间 0~149 里面的内容都来自版本号为 V_1 的日志记录中的数据，而地址空间 150~349 里面的内容则来自版本号为 V_2 的日志记录中的数据，地址空间 350~499 里面的内容则来自版本号为 V_3 的日志记录

中的数据。



图[log-file-system] 日志文件系统原理

日志文件系统最大的优点是，具有极高的写性能，它可以把文件的修改操作以日志的形式顺序地写入磁盘，不需要额外的磁盘寻址开销，不仅加快了文件写入的速度，也加快了文件系统发生崩溃时的恢复速度。但是，日志文件系统也具有显著的缺点：（1）读性能差，因为文件的内容被分散存储到不同的日志记录中，需要在磁盘中扫描日志才能重构得到文件内容；（2）存储空间的垃圾回收代价较大。

5.3.2 LGeDBMS

文献[KimBLLJ06]提出了一个可用在基于闪存的嵌入式设备上的、微缩版本的DBMS——LGeDBMS，它借鉴了基于日志的文件系统[RosenblumO92]的设计原理，把整个数据库看成是一个单独的日志，所有的数据库更新操作都会被顺序地追加到日志的尾部。由于闪存实行的是异地更新，因此，基于日志的方法不会违反闪存的这种操作特性。而且，把固态盘用来写入追加日志，使得在执行写操作时可以充分利用固态盘优良的顺序写带宽。从而时间局部性就被转换成空间局部性，继而顺序读就被转换成随机读操作。这种转换有助于提升整体性能，因为，和磁盘不同的是，闪存可以提供相同的顺序读和随机读性能。下面解释上述两种转换的具体含义。如图[time-to-space-locality]所示，在 t_0 时刻，日志中已经写入了三个物理闪存块 B_0,B_1 和 B_2，假设每个块只有四个页，在时间窗口[t_0,t_1]内，发生了一系列地更新 U_0,U_1,U_6,U_9，其中，U_i 中的 i 表示被更新页的 LPN。所有的更新都会被顺序地追加到日志的尾部，更新 U_0,U_1,U_6,U_9 会被顺序追加到新的块 B_3 中。四个更新 U_0,U_1,U_6,U_9 所涉及的页(LPN=0,1,6,9)，原本分别属于三个不同的块，LPN=0 和 LPN=1 的页属于块 B_0，LPN=6 的页属于块 B_1，而 LPN=9 的页则属于块 B_2。但是，可以看出，在更新后，属于时间窗口[t_0,t_1]内的更新 U_0,U_1,U_6,U_9 都被记录在同一个块 B_3 中，所以说，时间局部性就被转换成空间局部性。对于一个顺序读取序列 R_0,R_1,R_2,R_3 而言（R_i 表示读取 LPN=i 的页），原本 LPN=0,1,2,3 的四个页都属于同一个块 B_0，但是，经过四个更新 U_0,U_1,U_6,U_9 以后，LPN=2,3 的两个页仍然属于块 B_0，而 LPN=0,1 的两个页目前已经属于另一个块 B_3，要完成顺序读取 R_0,R_1,R_2,R_3，必须从块 B_0 和块 B_3 中分别读取对应的数据，所以说，顺序读被转换成随机读操作。

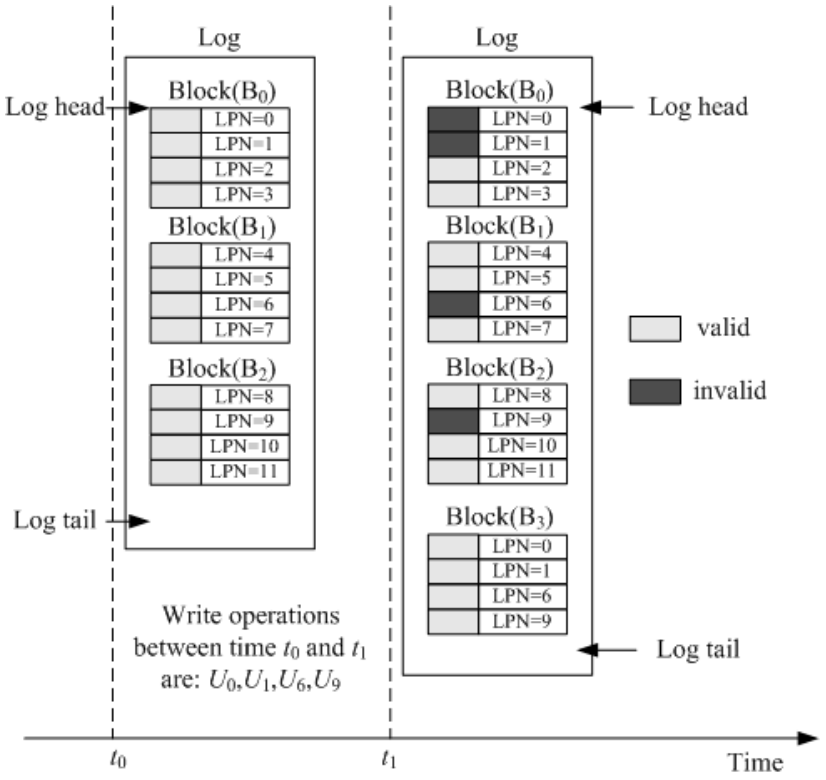


Fig. [time-to-space-locality] The transformation from temporal locality to space locality

图[time-to-space-locality] 时间局部性转换成空间局部性

5.3.3 A/P 方法

为了让 DBMS 在闪存设备也可以获得理想的性能，文献[StoicaAJA09]也提出了一个基于日志的存储数据库页的追加打包算法（Append and Pack，简称“A/P”）。A/P 方法在 DBMS 存储管理器的内部创建了一个抽象层，这个层位于逻辑数据组织和物理数据存储之间，可以为上层应用提供一个虚拟的块设备抽象。A/P 方法把整个闪存设备看成一个日志结构，从缓冲区中被驱逐出来的、需要持久性存储的脏页，就会被顺序地追加写入到闪存中，并在内存中维护一个索引，这个索引就是数据库页 ID 和日志中的存储位置之间的映射。在发生数据更新时，闪存中之前版本的数据库页，不会被就地覆盖，而是在内存索引中在逻辑上被设置为“无效”即可。为了避免频繁的写操作，A/P 方法会把脏页缓冲起来，等到满一个擦除块大小的时候，就顺序写入到和擦除块尺寸一样的闪存块中，从而优化写操作性能。

当可用闪存空间很低时，就无法继续执行追加页的操作，这时就必须回收那些被无效页占据的空间。A/P 会把最近最少更新的逻辑页进行打包合并，打包方法是：从追加日志结构的头部开始，读取有效数据并打包，然后顺序地写回到追加结构的尾部。在通常情况下，一些有效页会分散到许多擦除块中，这就使得一个单独的打包操作可以回收大量的空间。

对于典型的 OLTP 负载而言，会包含频繁发生更新的“热数据”和很少发生更新的“冷数据”。作者研究发现，如果把具有类似更新频率的数据放置在一起，空间回收时就可以实现所需移动的页的数量最小化。为此，A/P 方法设计了两个级别的数据集。第一个数据集存储了热的数据库页，而第二个数据集则存储了冷的数据库页。打包的方法是：从热数据集的头部开始读取有效数据，然后顺序地追加到冷数据集中，因为，对于那些已经到达热数据集头部的页而言，已经很长时间没有被更新过了，因此，最有可能成为冷数据。另外，冷数据也会存在更新，只不过更新频率比较低而已，当这种更新发生时，这个页会被再次像其他页一样被追加到热数据集的尾部。

A/P 方法的缺陷：该方法通过使用一个新的数据布局，把随机写操作替换成顺序写操

作，并且会连续地把被驱逐的 DBMS 数据页写入到闪存空间中。DBMS 数据页在闪存中的写入位置，会通过映射表被映射到具体的 DBMS 数据页。但是，这种方法不足以解决由于频繁写操作引起的闪存固态盘寿命问题，它只是简单地替换访问模式，并没有减少些操作的数量，因此，不能延长闪存固态盘的寿命。

5.3.4 IPL 方法

5.3.4.1 IPL 方法概述

正如文献[LeeKWCK09]的研究所指出的那样，基于日志的方法的优点是，不管什么写操作模式，都可以取得较好的性能，但是，也有明显的缺陷，主要表现在以下几个方面：

- 无法避免许多小数据量的日志写操作。因为数据库负载中会包含很多的元组级别的更新，这类更新操作所涉及的数据量通常都很小；当频繁地发生这种小数据量的更新时，很快就会耗光空闲页，因此，就需要频繁地调用垃圾回收程序，带来许多擦除操作。一个更新操作的平均延迟，会随着代价高昂的擦除操作的频繁执行而增加，这就会成为整个数据库服务器的性能瓶颈；
- 以牺牲读操作性能换取写操作性能的优化。当从数据库中读取的一个数据页的时候，当前版本的数据页必须被重新创建，方法是把日志中的更新记录全部应用到之前版本的旧数据页上面。因为和数据页相关的日志记录会被分散到日志的各个部分，因此，创建数据库页的最新版本时，需要扫描整个日志，代价很大；因此，这种顺序日志方法，虽然优化了写性能，却是以牺牲读性能为代价的，数据库的整体性能未必可以得到改善；
- 基于日志的方法无法解决前面介绍过的“耦合页问题”。

为了克服上述基于日志的方法的缺陷，文献[LeeM07]提出了“页内日志”方法——IPL（In-page logging）方法，把数据和日志放在一起存储，即把一个数据页和它的日志记录存放在闪存的同一个物理地址空间中（存放在同一个擦除单元中），从而避免了在写事务日志时的额外写操作。IPL 把每个块中的页划分成固定数量的数据页和日志页。一个逻辑页的更新日志，只会被写入到包含了这个逻辑页所对应的物理页的块中。因此，当重新创建一个逻辑页时，IPL 就只需要读取数据页（物理页）和位于同一个物理块中的日志页。当块中没有可用的空闲日志页时，IPL 会把块中的数据页和日志页进行合并，然后，把合并后的页写入到一个新的块中，旧的块就会被擦除和垃圾回收。IPL 减少了重建一个逻辑页时需要从闪存中读取的日志页的数量，因为，由于存在合并操作，日志页的数量不会无限增长。IPL 的性能和其他基于日志的方法是类似的，因为 IPL 继承了基于日志的方法的优点和缺点。

5.3.4.2 IPL 方法的核心思想

IPL 方法的核心思想主要体现在以下两个方面：

第一，把一个数据页和它的日志记录存放在闪存的同一个物理地址空间中，加快数据页重构速度。

由于闪存存在“写前擦除”的限制，即使对一个页中的某条记录进行更新，都会导致包含该更新的页的整体失效，需要写一个新版本的页到闪存中可用的已经擦除的空闲页中。为了避免这一点，IPL 只会把页的更新以日志的形式写入到闪存中，而不是每次更新都写入整个页。

在传统的顺序日志方法中 (比如 LFS)，所有日志记录都必须被顺序地写入到一个存储介质。但是，这种日志类型需要考虑的一个问题是，每当需要从数据库中读取一个页时，必须重构该页的最新版本数据，重构的方法是，把存储在日志中的更新记录应用到这个页的前一个版本上，从而得到该页的最新版本数据。由于属于这个数据页的日志可能是分散存储的，只有对整个日志进行扫描才可以找到所有相关的更新日志，因此，重构数据页的操作代价很大。

但是，闪存是纯粹的电子设备，不存在任何机械部件，分散的日志写操作不会产生明显的性能惩罚，因而，也就没有必要对日志记录进行顺序存储。所以，与传统的顺序日志方法不同的是，在 IPL 方法中，更新操作的日志记录并不会被顺序地追加到日志的尾部，而是分散地存储到不同的闪存空间中。IPL 把数据页及其对应的日志记录保存在同一个闪存物理空间中，也就是放在同一个擦除块中，这种设计方法可以带来明显的好处——加快了当前版本数据页的重构速度，因为，只需要扫描同一个物理空间中的日志记录，而不需要扫描所有的日志记录。

第二，闪存读带宽比写带宽大得多，写操作和擦除操作比读操作代价高很多，可以设计方法尽量避免写操作和擦除操作，即使以增加读操作的代价为前提。

根据数据库中的“写前记录日志”（wrtie-ahead-logging，简称 WAL）协议，更新操作的日志记录必须在写操作完成之前被写入到持久稳定的存储介质上（比如闪存）。而闪存的写操作的最小单元是页，即使是小数据量的更新也需要使用整个空闲页，空闲页用尽后还会启动垃圾回收过程，包含大量写和擦除操作。因此，为了记录日志而进行的频繁的、小量的闪存写操作，会带来明显的闪存开销。为了避免频繁的日志写操作，从而减少写操作和擦除操作的次数，IPL 在内存中设置了缓冲区，更新日志会临时存放到内存缓冲区中，只有当缓冲区写满一页或者一个脏数据页已经被驱逐出缓冲区的时候，该日志记录才会被写入到闪存中。多个日志记录可以被一次性执行写操作，写入到闪存日志区域中，因此，就可以避免频繁的写操作和擦除操作。但是，在 IPL 方法中，需要对读操作进行重新定义，增加了读操作的代价。当一个数据页需要从闪存中读取时，该数据页的当前版本必须被临时创建，创建的方法是：从闪存中读取该数据页对应的更新日志记录，然后，把这些更新应用到之前版本的数据页上。这种针对读操作的新定义，很显然会增加读操作的开销。但是，大量实验证明，这种方法最终是值得的，它可以大幅度改善数据库的整体性能，因为大大减少了写和擦除操作。

5.3.4.3 IPL 的设计

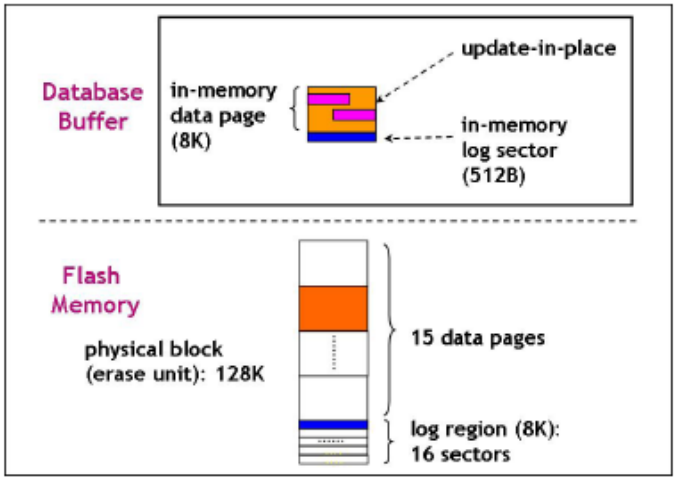


Figure 2: The Design of In-Page Logging

图[IPL] IPL 的设计

如图[IPL]所示，在 IPL 方法中，闪存的每个擦除单元都被分割成两个部分，即数据区域和日志区域，前者包含若干数据页，后者包含若干日志页。在内存缓冲区中，每个数据页（逻辑页）在“变脏”的时候（即发生了更新），IPL 存储管理器就会为之临时分配一个日志页，用来记录该页发生的更新。如果内存日志页中的日志记录满了，或者当一个脏数据页被驱逐出缓冲区时，内存中与该页对应的日志记录就会被写入到闪存中，即被写入到闪存中与该数据页对应的日志页中，这样，数据页和它的更新日志记录就可以存放到闪存的同一个擦

除单元中，加快数据页的重构速度。由于有了内存中的日志页作为缓冲区，属于一个数据页的多个更新日志记录可以被一次性写入到闪存中，因此，就可以避免频繁的擦除操作。当内存中的日志记录被写到闪存的日志区域以后，这个内存中的日志页就可以释放掉。需要注意的是，当一个脏页被驱逐出内存时，没有必要把这个脏页的内容写入到闪存中，因为，所有针对这个脏页的更新，都已经写入闪存中的日志。因此，闪存中前一版本的数据页没有发生任何变化，只是在日志中增加了更新日志记录。当一个数据页需要从闪存中读取时，需要临时创建该数据页的当前版本，方法是，从闪存中读取该数据页对应的更新日志记录和前一版本的数据页，然后，把这些更新应用到前一版本的数据页上。

一个内存中的日志页，只能存储有限数量的日志记录，当一个日志页满时，其内容就会被刷新到闪存中相应的日志页中。由于闪存中每个擦除单元的可用日志页的数量都是有限的，如果某个擦除单元中的数据页更新非常频繁，那么，这个擦除单元就会很快消耗完日志页。这个时候，IPL 存储管理器就会触发一个进程，对日志页和与之关联的数据页进行合并。如果在一个擦除单元中已经不存在可用的日志页，IPL 存储管理器就会分配一个空闲的擦除单元，然后，把更新日志应用到之前版本的数据页，从而得到最新版本的数据页，并把它写入到新的擦除单元，最后，释放掉旧的擦除单元。

5.3.4.4 IPL 方法的缺陷

IPL 方法的缺陷表现在以下几个方面[LeeKWCK09]：

- 首先，没有解决小数据量的日志随机写操作问题。虽然 IPL 方法设置了缓冲区避免了频繁的日志写问题，改进了性能，但是，由于日志和对应的数据页要一起存放，这意味着把顺序日志写操作转换成了随机日志写操作，留下了更大的随机写问题，而相关研究[BouganimJB09]已经证明随机写会严重恶化闪存性能；
- 其次，当写请求集中在特定的块中时，IPL 方法可能表现出很差的性能。因为，这种情况下，一个块的更新操作会很快耗光日志区间中的空闲页，然后启动垃圾回收过程，执行数据块和日志块的合并操作，就会涉及大量的写操作和擦除操作，代价很大；
- 再次，IPL 要求针对某个数据页的更新操作的日志记录只能被写入到闪存中的日志区域，而且必须被写入到与该数据页相关联的日志块中。由于 MLC NAND 闪存要求对一个页进行顺序写入，因此，IPL 这种日志写入方式就会导致一个块中的许多空闲页可能无法被利用，造成不必要的空间浪费；
- 另外，IPL 提出了它自己的存储管理器，必须为每个数据页分配一个称为“内存日志页”的内存空间。如果一个数据页被缓冲区管理器驱逐，只需要把内存中的日志页写入到闪存的日志区域，而不需要写入整个数据页。因为日志页要比数据页小，因此，IPL 存储管理器可以明显减少页写操作。但是，如果针对一个数据页的多个日志页被写入到闪存的日志区域，那么，当需要从闪存中读取一个数据页时，就需要从闪存中读取多个日志页。
- 最后，无法解决“耦合页问题”。

5.3.5 ICL 方法

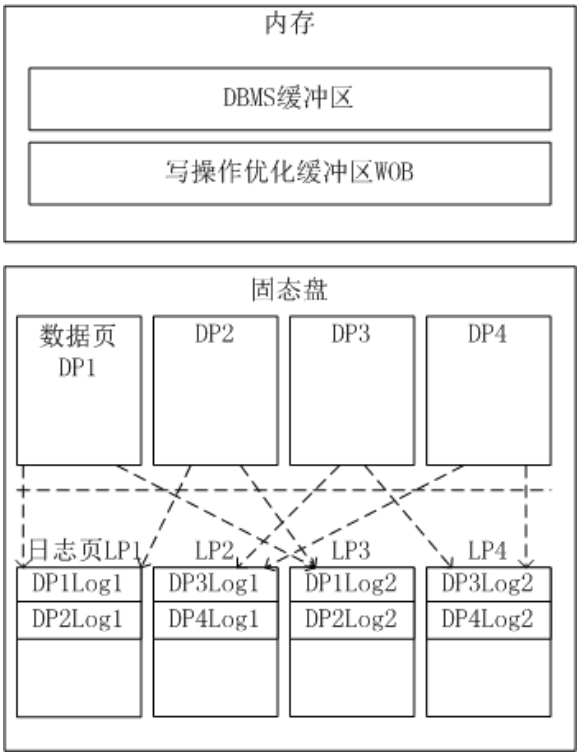
Roh 等人[RohLP10]提出了一种新的基于日志的方法，这里我们称它为“ICL 方法”，因为，该方法采用了 ICL（InCremental Logging）日志。ICL 方法是为了解决 OLTP 应用环境下存在的“面向写的问题”，即频繁的写操作会缩短闪存固态盘寿命的问题。ICL 方法的目标就是尽量减少针对闪存的写操作的数量，而且以适当增加读操作为代价。该方法通过 ICL 日志机制把随机写操作转换成顺序写操作，同时减少了页写操作。

5.3.5.1 ICL 方法概述

ICL 方法设计了一个 DBMS 写操作优化层，这个层包含了一个日志缓冲区（称为“写操作优化缓冲区”）、相应的日志空间和一个内存映射表。写操作优化缓冲区 WOB（Write Optimized Buffer）位于 DBMS 缓冲区和固态盘之间，可以用来减少页写操作。写操作优化缓冲区会收集每个 DBMS 数据页的脏区域，并把这个脏区域转变为日志。这个日志包含了两个部分，即日志头部和日志内容。日志头部包含了用来确定被驱逐 DBMS 数据页的信息，以及 DBMS 数据页内部被更新区域的<位移,尺寸>。日志的内容，就是 DBMS 数据页中被更新的元组或者被更新的部分。当写操作优化缓冲区满时，它会把日志刷新到称为“ICL 日志空间”的闪存日志空间中。在 ICL 机制中，和一个 DBMS 数据页对应的 ICL 日志中的日志内容是递增的，这样，在读取一个最新的 DBMS 数据页时，只需要读取一个日志页，这个日志页包含了针对这个 DBMS 数据页的最新日志。ICL 方法设计了映射表来维护从每个 DBMS 数据页到与之对应的日志页的映射。很显然，每个被驱逐的 DBMS 数据页所生成的更新日志的大小，会明显小于整个 DBMS 页的大小。因此，写操作优化缓冲区在有限的内存空间内就可以缓存大量的 DBMS 页写操作。

5.3.5.2 ICL 日志

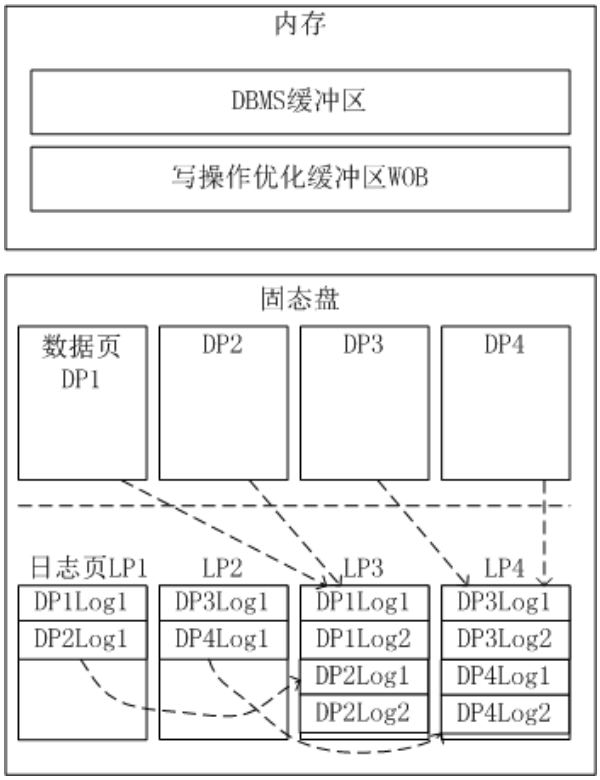
这里将通过对传统日志和 ICL 日志的比较，来阐释 ICL 日志的优点。假设对于缓冲区中的 4 个数据页 DP_1,DP_2,DP_3,DP_4，每个数据页上面都发生了一次更新，采用传统日志的方法，会为 4 个数据页分别生成 4 个日志 DP1Log1, DP2Log1, DP3Log1, DP4Log1，然后，DP1,DP2,DP3,DP4 上又发生一次更新，又生成 4 个日志 DP1Log2, DP2Log2, DP3Log2, DP4Log2。如图[ICL-log-traditional]所示，当需要把缓冲区中的内容刷新到闪存中时，系统把 DP1Log1, DP2Log1 一起写入到闪存中的日志页 LP1 中，把 DP3Log1, DP4Log1 一起写入到闪存中的日志页 LP2 中，把 DP1Log2, DP2Log2 一起写入到闪存中的日志页 LP3 中，把 DP3Log2, DP4Log2 一起写入到闪存中的日志页 LP4 中。这里假设日志页 LP1,LP2,LP3,LP4 中剩余的空间被上述 4 个数据页以外的其他数据页的日志所占用。可以看出，当采用传统的日志方法时，在对 4 个数据页发生 8 次更新时，只会导致 4 个日志页的写操作。



图[ICL-log-traditional] 传统的日志写入过程示意图

采用传统的日志时，与某个 DBMS 数据页对应的日志，可能会被分散到不同的日志页中，引起额外的读取代价。比如，在图[ICL-log-traditional]中，与数据页 DP1 对应的日志 DP1Log1 和 DP1Log2，分别被分散存储在两个日志页 LP1 和 LP3 中。这样，在读取数据页 DP1 的最新版本时，就需要从闪存中读取原来旧版本的 DP1 和两个日志页 LP1 和 LP3，从而获得旧版本的 DP1 和两个更新日志 DP1Log1 和 DP1Log2，然后对三者进行合并得到新版本的数据页。这种方法在读取一个数据页的当前版本时，需要较高的读取代价。

ICL 日志可以较好地解决这个问题。一旦写操作优化缓冲区 WOB 要把缓冲区中的日志刷新到闪存中的一个新的日志页中，它就会把闪存中与该 DBMS 数据页对应的之前的日志拷贝到内存中。然后，WOB 把当前的日志和原来的日志一起写入到闪存中新的日志页中去。如图[ICL-log]所示，WOB 在执行第一次刷新操作时，把属于 DP1 的日志 DP1Log1 和属于 DP2 的日志 DP2Log1 写入到闪存中的 LP1 日志页中。同理，还需要把分别属于 DP3 和 DP4 的日志 DP3Log1, DP4Log1 写入到闪存中的 LP2 日志页中。在第一次刷新操作中，不会引起对之前日志的拷贝，因为，DP3 和 DP4 不存在之前的日志。但是，在第二次刷新操作中，WOB 会拷贝之前的日志，因为，它必须把分别属于 DP1 和 DP2 的日志 DP1Log2, DP2Log2 写入到闪存中，而 DP1 和 DP2 在 LP1 中已经存在之前的日志 DP1Log1 和 DP2Log1。因此，需要首先把闪存中之前的日志 DP1Log1 和 DP2Log1 拷贝出来，然后再把 DP1Log1 和新日志 DP1Log2 一起写入到闪存中的日志页 LP3 中，并把 DP2Log1 和新日志 DP2Log2 也一起写入到闪存中的日志页 LP3 中。同理，在日志页 LP2 中的 DP3 和 DP4 的日志，即 DP3Log1, DP4Log1，也会被拷贝出来，DP3Log1 和 DP3Log2 会被一起写入到闪存的日志页 LP4 中，DP4Log1 和 DP4Log2 也会被一起写入到闪存的日志页 LP4 中。



图[ICL-log] ICL 日志写入过程示意图

虽然一个 ICL 日志的大小是不断增加的，但是仍然具备减少页写操作的能力。减少页写操作的机制，就要把可能多的 DBMS 数据页写操作转换成相对较小的日志，并且一次性把这些日志刷新到闪存的一个日志页中。通过这种方式，许多针对 DBMS 数据页的写操作，都可以被缩减成为一个日志页写操作。在 ICL 中唯一发生变化的是，对于每次刷新操作，

日志的大小都是不断增加的。只要一个 ICL 日志的大小还小于一个 DBMS 数据页，就仍然可以获得性能收益。如果一个 ICL 日志的大小增加到不会带来收益的时候，就应该执行 ICL 合并操作。已经增加到足够大的 ICL 日志，会被合并到相应的 DBMS 数据页中，并释放回收 ICL 日志空间以供后续使用。释放回收 ICL 日志空间的过程，并不需要实际的页写操作，因为，日志页只需要被逻辑地设置为失效，根本不需要物理的重写操作。

这里需要关注的一个核心问题是，拷贝之前的日志这个过程到底需要多大的页读代价。在最坏的情形下，在每次需要把新的日志刷新到闪存中时，可能在闪存中都已经存在与之对应的之前的日志，而且都是存储在不同的日志页中。即使在这种最坏的情形下，由拷贝操作引起的额外读取代价，不会大于日志页的最大读取数量，即 WOB 刷新到闪存中的新日志的总数量（等于被驱逐的 DBMS 数据页的数量）。和采用传统日志的方法以及 IPL 方法相比，ICL 方法额外的读代价相对较小。

5.3.5.3 采用 ICL 日志以后读取 DBMS 数据页的过程

采用 ICL 日志以后，读取一个 DBMS 数据页的最新版本的过程，可以按照下面的方法进行。首先，写操作优化层 WOB 读取 DBMS 数据页，如果这个 DBMS 数据页存在相应的 ICL 日志，它就读取包含这个 ICL 日志的日志页。其次，它会从这个日志页中抽取属于该 DBMS 数据页的 ICL 日志。再次，检查 WOB 中的日志，如果存在属于该 DBMS 数据页的日志，则这些日志也会被抽取出来，它们和来自日志页的 ICL 日志一起被刷新到这个 DBMS 数据页中。最后，DBMS 数据页被读取到 DBMS 缓冲区中，得到该页的最新版本。

采用 ICL 日志以后，读取最新版本 DBMS 数据页的过程的代价，要比采用传统的日志低许多。在采用 ICL 日志的情况下，这种读取过程只需要读取一个日志页。在图[ICL-log] (b)中，读取每个 DBMS 数据页最新版本时，都只需要读取一个日志页，比如，读取数据页 DP1 时，只需要读取一个日志页 LP3，因为，属于 DP1 的日志 DP1Log1 和 DP1Log2 都存放在日志页 LP3 中；同理，读取数据页 DP2，DP3 和 DP4 的最新版本时，也只需要分别读取日志页 LP3，LP4 和 LP4。

5.4 基于页差异的日志方法

Kim 等人[KimWS10]提出了基于页差异的日志方法——PDL（page-differential logging）。一个页差异是指闪存中的原始页和内存中的当前页之间的差异。下面首先给出 PDL 方法概述，并给出一个实例，然后介绍该方法的设计原则。

5.4.1 PDL 方法概述

在 PDL 方法中，一个逻辑页会被存储成闪存中的两个物理页，即一个原始页和一个差异页。原始页包含了整个逻辑页的数据（它可能是旧版本数据，因为逻辑页可能会在得到原始页后发生更新），差异页包含了原始页和当前逻辑页之间的差异部分。闪存中的一个差异页，是一个物理页，但是，这个物理页可以包含来自多个逻辑页的差异部分。因此，多个逻辑页对应的差异页（假设每个差异页的大小都小于一个闪存物理页），可以被存放在同一个物理页中，构成一个大的差异页。因此，某个逻辑页对应的差异页，可能并不是一个完整的物理页，而只是一个物理页中的一个片段。

当需要计算差异页时，只需要从闪存中读取原始页，并把原始页和内存缓冲区中的当前逻辑页进行比较，计算得到差异部分，然后把差异部分写入到内存写缓冲区中，当缓冲区满时再把这些差异内容写入到闪存中。

当需要从闪存中重构一个逻辑页时，需要从闪存中读取两个页，即一个原始页和一个差异页，然后，把二者进行合并得到一个逻辑页。

PDL 方法和基于页的方法、基于日志的方法都有很大的不同，主要体现在以下几个方面：

（1）基于页的方法会把整个逻辑页（包括变化和没有变化的数据）都写入闪存。基于

日志的方法会把针对这个逻辑页的所有更新都以更新日志的形式写入闪存，这意味着，如果一个数据项发生了多次更新，就会向闪存写入多条更新日志。实际上，在重构一个逻辑页时，如果按照日志写入闪存的时间顺序进行反向扫描，最先扫描到的更新日志肯定对应着数据的最新版本，其他旧版本的更新日志就是无用的，根本就不需要扫描。存储这些旧版本的更新日志，不仅浪费更多的闪存空间，也增加了重构逻辑页时的闪存扫描开销。PDL 方法则和基于页的方法、基于日志的方法这二者具有较大的不同。PDL 方法可以通过计算得到差异页，而不需要维护所有的更新日志，也就是说，即使一个逻辑页被更新过许多次，PDL 方法也只会把逻辑页和原始页的差异部分写入到差异页中。而且，PDL 方法只计算一次差异，也就是当更新后的页需要被刷新到闪存中时才去计算一次差异，这要比基于日志的方法节省很多闪存存储空间。例如，假设一个逻辑页中的某个数据项在内存中更新过了两次，先从 aaaaaa 更新到 bbbbba，然后再更新到 bcccba。基于日志的方法会记录两个变化 bbbbb 和 ccc，而差异页中只会包含差异内容 bcccb。此外，PDL 方法在计算差异时，只需要从闪存中读取原始页到内存缓冲区中，把原始页和内存中的当前页进行比较，就可以计算得到差异内容，这个过程的代价相对较小，因为，闪存中读操作的速度要远快于写操作和擦除操作的速度。

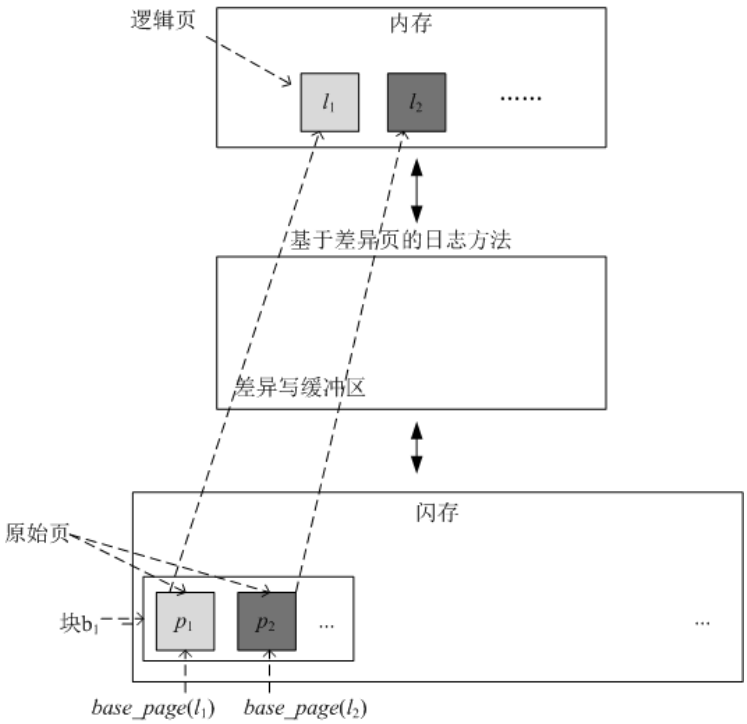
（2）当从闪存中重构一个逻辑页时，PDL 方法比基于日志的方法需要更少的读操作，因为，PDL 方法至多需要读取两个页，即一个原始页和一个差异页（即包含差异的一个页）。而基于日志的方法则可能需要读取多个物理页，因为，一个逻辑页对应的更新日志内容可能被分散存储在多个物理页中，从闪存中重构一个逻辑页时，需要从多个物理页中读取更新日志的内容，然后进行合并得到逻辑页。

（3）当需要把一个更新后的逻辑页刷新到闪存中时，PDL 方法比其他两种方法需要更少的写操作，因为，PDL 方法只写入差异部分的内容。此外，PDL 也改善了闪存的寿命，因为，更少的写操作意味着更少的擦除操作。

（4）PDL 方法和 DBMS 存储管理模块是松耦合的，而基于日志的方法是紧耦合的。基于日志的方法需要修改 DBMS 的存储模块，这样才能保证及时探测到系统中发生的数据更新，因为，只有 DBMS 的内部存储模块才能够探测到这些数据变化。而 PDL 方法不需要修改 DBMS 的数据存储模块，因为，它是在 DBMS 存储模块的外部计算数据差异，只需要对闪存中的原始页和内存中的当前页进行比较，即可计算得到差异。

5.4.2 PDL 方法的一个实例

图[PDL]显示了一个 PDL 的例子，假设有一个逻辑页 p，它在闪存中的原始页用 *base_page*(*p*)表示，闪存中的原始页和内存中的当前页之间的差异用 *differential*(*p*)表示，写入到闪存中的差异页用 *page_differential*(*p*)表示。图[PDL(a)]显示了在内存中的逻辑页 l_1 和 l_2 的初始状态，即从闪存中读取物理页 p_1 和 p_2，放入内存，分别得到逻辑页 l_1 和 l_2。



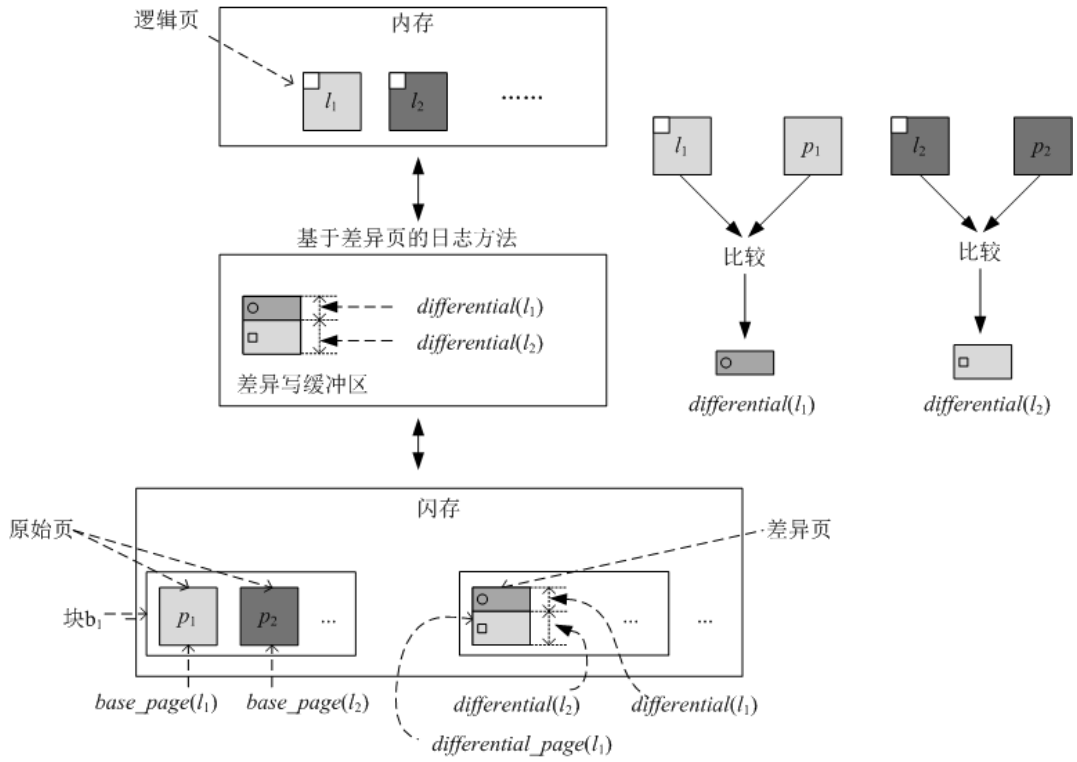
图[PDL(a)] 内存中逻辑页的初始状态

图[PDL(b)]显示了逻辑页 l_1 和 l_2 发生更新后计算和存储差异页的过程。当 l_1 和 l_2 需要被刷新到闪存中时，就需要计算和存储差异页，具体而言，需要执行下面的三步操作：

（1）从闪存中分别读取与逻辑页 l_1、l_2 对应的原始页 *base_page*(l_1)（即 p_1）、*base_page*(l_2)（即 p_2）；

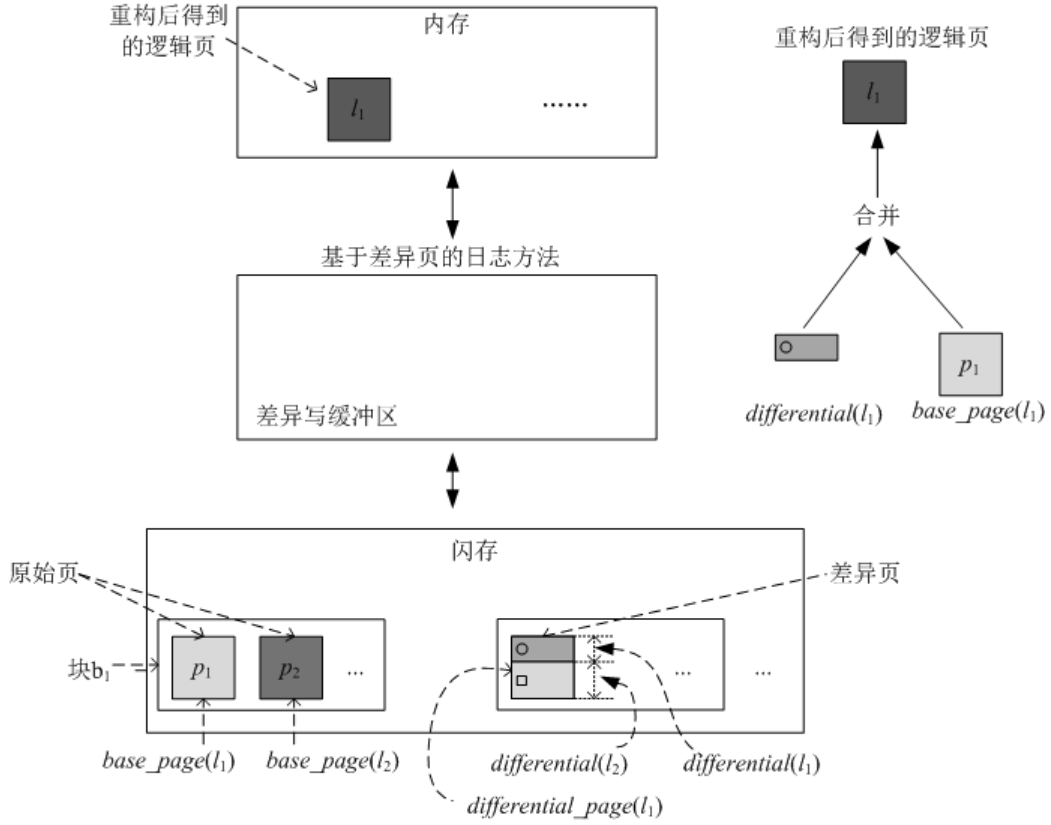
（2）把逻辑页 l_1 与它的原始页 *base_page*(l_1)进行比较，计算得到逻辑页 l_1 的差异部分 *differential*(l_1)；把逻辑页 l_2 与它的原始页 *base_page*(l_2)进行比较，计算得到逻辑页 l_2 的差异部分 *differential*(l_2)；

（3）把差异内容 *differential*(l_1)和 *differential*(l_2)分别写入到内存的写缓冲区中；当缓冲区满的时候，缓冲区中的这些差异内容就会被写入到物理页 p_3 中，把差异内容 *differential*(l_1)写入 p_3 以后可以得到与逻辑页 l_1 对应的差异页 *page_differential*(l_1)，同理，把差异内容 *differential*(l_2)写入 p_3 以后可以得到与逻辑页 l_2 对应的差异页 *page_differential*(l_2)，也就是说，同一个物理页 p_3 中包含了来自两个不同逻辑页 l_1 和 l_2 的差异页，即 *page_differential*(l_1)和 *page_differential*(l_2)。



图[PDL(b)] 逻辑页发生更新后计算和存储差异页的过程

图[PDL(c)]显示了从闪存中重构逻辑页 l_1 的过程，即把逻辑页 l_1 对应的原始页 *base_page*(l_1)（即 p_1）和物理页 P_3 中与逻辑页 l_1 对应的差异页内容 *page_differential*(l_1)从闪存中读取出来，进行合并后就可以得到逻辑页 l_1。



图[PDL(c)] 重构逻辑页的过程

5.4.3 PDL 方法的设计原则

为了保证获得好的读写操作性能，PDL 方法在设计时遵守了三个原则，这些原则克服了基于页的方法和基于日志的方法的缺点，具体如下：

- 原则 1（只写入差异）：当一个逻辑页需要被刷新到闪存中时，只写入差异部分；
- 原则 2（至多写入一个页）：当一个逻辑页需要被刷新到闪存中时，至多写入一个物理页，即使这个逻辑页在内存中已经被多次更新；
- 原则 3（至多读取两个页）：当从闪存中重新创建一个逻辑页时，至多需要读取两个物理页。

PDL 方法完全符合上述三条设计原则，具体如下：

首先，在 PDL 方法中，当一个更新过的逻辑页需要被刷新到闪存中时，通过比较内存中的当前逻辑页和闪存中的原始页之间的差异，就可以计算得到差异页，然后把差异页写入到写缓冲区中，当缓冲区满时再写入到闪存。因此，PDL 方法符合“只写入差异”的原则。

其次，当一个逻辑页只是被简单地更新时，PDL 方法只会更新内存中的逻辑页，而不会记录日志，直到被更新的逻辑页需要被写入到闪存中时，才去计算得到差异页并写入闪存。因此，PDL 方法满足了“至多写入一个页”的原则。理论上，差异部分的大小可能会大于一个物理页。但是，实际上只有当一个页中的大部分都被更新时，相应得到的差异页的大小才会比一个物理页大。这种情况还是可能发生的，因为，差异页不仅包含更新后的数据，而且包含一些元数据，比如位移和长度。在这种情形下，PDL 会抛弃被创建好的差异页，并且把更新过的逻辑页自身写入到闪存中作为一个新的原始页，从而满足“至多写入一个页”的原则。在这种情形下，PDL 就会演变成为和基于页的方法一样。

最后，当需要从闪存中重新构建一个逻辑页时，PDL 方法可以从闪存中读取该页对应的原始页和差异页，然后，把原始页和差异页进行合并得到逻辑页。在一些情况下，如果逻辑页在原始页生成后没有发生过更新，那么，PDL 方法只需要读取一个物理页，即原始页，而不需要读取差异页。也就是说，PDL 方法至多需要读取两个物理页，因此，PDL 符合“至多读取两个页”的原则。

当闪存中不存在更多空闲空间的时候，废弃的页就会被垃圾回收，PDL 方法会选择一个块进行垃圾回收。由于块中可能包含有效的原始页或者差异页，因此，在擦除块之前，PDL 方法会把那些有效页转移到一个新的块中，这些新块是为垃圾回收过程保留的。但是，对于差异页而言，PDL 方法只把多个有效的差异页转移到一个新的差异页中，也就是相当于执行了压缩。PDL 方法比基于页和基于日志的方法需要更少的写操作，因为，它满足了“只写入差异”和“至多写入一页”的原则。因此，PDL 方法要比其他方法包含更少的垃圾回收操作，从而可以取得更好的性能。

5.5 StableBuffer

在闪存存储管理中，如果要使用基于日志的方法（比如 IPL）和 PDL 方法，通常需要通过下面两种方式中的一种：第一种方式，绕过 FTL 机制，直接对底层的闪存芯片进行访问；第二种方式，修改闪存设备的驱动程序，也就是需要修改 FTL 机制，从而让 FTL 机制提供对日志的支持。但是，对于商业化的闪存设备而言，这两种方式并非总是可以做到的，因为，FTL 机制通常固化封装在闪存设备内部，无法绕开 FTL 机制直接访问底层的闪存芯片，也无法对 FTL 机制进行修改。因此，这就使得基于日志的方法的应用受到一定的限制。

针对这个问题，Li 等人[LiXCH10]提出了 StableBuffer，它可以充分利用独特的闪存设备写模式，来优化写操作集中的 DBMS 应用（比如 OLTP）的性能。StableBuffer 可以作为 DBMS 缓冲区管理器的一个插件模块，它对于底层的闪存设备而言也是透明的，因此，可以直接应用到闪存数据库中。

5.5.1 StableBuffer 的基本原理

一些关于闪存数据库的 IO 性能的研究（比如文献[BouganimJB09][ChenKZ09]）表明，闪存设备的写操作性能，不仅和简单的参数有关，比如访问类型（顺序或随机）和粒度（页的大小），而且还和“写模式”紧密相关。所谓的写模式，是指一个写操作序列所具备的特征，比如，如果一个写操作序列属于顺序写模式的一个实例，那么，这个序列中的所有写操作在地址空间上是顺序执行的。测试基准 uFLIP 的结果[BouganimJB09]显示，一个普通的写操作，如果随机地写入到整个闪存空间，那么就会表现出较差的性能，比如 OLTP 应用中的写操作通常就是随机的，会很容易成为闪存设备上的性能瓶颈。但是，确实存在几种“不是那么随机”的写模式，通过利用闪存设备的写操作局部性和写操作并行性，仍然可以获得较高的性能。典型的两种模式如下：

- 集中写：是指只针对一个集中区域的随机写操作。这里所谓的“集中区域”，是指逻辑地址空间中的一个小区域。例如，一个预分配的小于 4MB 的文件内部，就可以看成是一个集中区域。
- 分区写：是指分区顺序写操作。例如，一个写操作序列“6,21,7,22,8,23,…”，就是分区写模式的一个实例。这种写操作序列表面上看起来类似于随机写，但是，实际上，它是两种顺序的写操作序列的混合，即序列“6,7,8…”和序列“21,22,23…”的混合，这两个序列对应着地址空间中的两个分区。

表[uFLIP-IO-pattern]给出了在两种不同的写模式下面的写操作响应时间，包含了在两种闪存设备上测量得到的顺序和随机写操作时间[BouganimJB09]。可以看出，和普通的、非高效的随机写操作相比较而言，这些闪存设备具有三种高效的写模式，即顺序写、集中写和分区写。比如，对于 Mtron SATA 7035-016 而言，普通的随机写操作的延迟高达 9ms，而顺序写的延迟只有 0.4ms，集中写和分区写这两种特殊类型的随机写操作的延迟也分别只有 0.8ms 和 0.6ms。

表[uFLIP-IO-pattern] 两种固态盘上的写操作响应时间

固态盘产品品牌型号	顺序写	随机写		
	(ms)	普通写(ms)	集中写(ms)	分区(ms)
Mtron SATA 7035-016	0.4	9	0.8	0.6
Samsung MCBQE32G5MPP	0.6	18	0.9	1.2

为了充分利用上述三种高效的写模式的优势，StableBuffer 方法在设计时（如图[StableBuffer-overview]所示），采用了一个闪存中预分配的专用临时存储空间，称为“StableBuffer”，每当 DBMS 发生页写操作时，不是立即把该页写入到闪存中的实际目标地址，而是把该页临时写入到闪存中的 StableBuffer 中。然后，采特定的方法对 StableBuffer 中的写模式进行识别，当探测到高效的写模式时，再把高效的写模式刷新到闪存中的目标地址中。但是，这里需要注意一个问题，有一些写操作序列可能本来就属于高效的写模式，很显然，这些写操作不应该被缓存到 StableBuffer 中来探测发现高效的模式，而是应该直接被写入到闪存的目标地址中。对于这个问题，StableBuffer 方法假设一个 DBMS 缓冲区管理器是足够智能的，可以识别这些属于高效写模式的事务，并且确定如何高效地响应它们的读写请求。已有的研究文献（比如 DBMIN[ChouD85]），已经证明了这点是可以做到的，即可以在 DBMS 缓冲区管理器中捕获这些属于高效写模式的事务。因此，和足够只能的缓冲区管理器进行合作，就可以避免在采用 StableBuffer 时对顺序写操作模式造成的性能恶化。

现在来解释一下为什么 StableBuffer 方法可以获得更好的性能。让我们来考虑一个在 Mtron SATA7035-016 闪存设备上的页写操作，该设备的写操作性能参数可以参见表[uFLIP-IO-pattern]。采用 StableBuffer 方法以后，把一个页 p 写入到闪存的目标地址，需要包含两个属于高效模式的写操作和一个随机读操作：

- 第一个写操作是指把页 *p* 写入到闪存中的 StableBuffer 中。由于 StableBuffer 是一个比较小的集中区域，随机写入和顺序写入一样快，因此，这个写操作是一种高效的写操作。
- 第二个写操作是指把页 *p* 刷新写入到闪存中的目标位置。因为在 StableBuffer 方法中，发生刷新操作时，会把探测到高效的模式刷新写入到闪存中，因此，这个写操作也是一种高效的写操作。
- 一个随机读操作是指从 StableBuffer 中读取页 *p*。因为，在 StableBuffer 方法中发生刷新时，必须从闪存的 StableBuffer 中读取一个页，然后把它写入到闪存的目标地址。

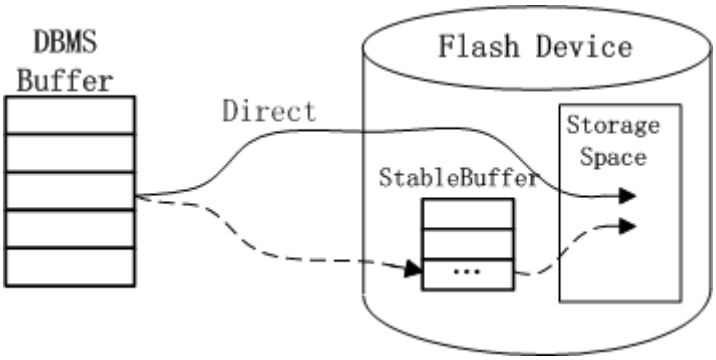
基于表[uFLIP-IO-pattern]的测量结果，两个高效的写模式的最差的代价是 0.8+0.8=1.6ms。随机读操作通常要比一个顺序写操作代价小，也就是小于 0.4ms。因此，上述过程的总代价大约是 1.6+0.4=2.0ms。另一方面，如果我们没有采用 StableBuffer 方法，而是直接把页 *p* 从 DBMS 缓冲区中刷新到闪存的目标地址，它往往很可能是一个随机写操作，代价会高达 9ms。

综上所述，虽然 StableBuffer 把同一个页执行了两次写入操作和一个随机读操作，但是，由于这两个写操作遵循了高效的写模式，因此，总代价仍然要比不采用 StableBuffer 直接刷新的策略改进了 4.5 倍。

5.5.2 StableBuffer 的设计位置

StableBuffer 的作用就是临时缓冲和收集写操作，从而探测和发现一些高效的写模式，比如，顺序写、集中写和分区写，因此，一个很容易想到的思路是，在内存中设置一个写操作缓冲区，对随机写操作进行临时缓存，也就是说，写缓冲区用作一个“写操作收集器”来收集写操作，并记录它们的顺序，用来进行性能优化。但是，这种方法存在两个方面的局限性。首先，需要消耗可观数量的缓冲区空间，而这些空间本来可以分配给 DBMS 的，这就会明显牺牲读操作的性能，并且会让缓冲区管理变得更加复杂。其次，由于内存是易失性存储介质，因此，如果采用强制的缓冲区策略，那么，当一个事务提交时，缓冲区中的写操作必须被强制地写入到稳定的存储器中；而如果采用非强制的缓冲区策略，则又必须提供一些额外的机制（比如日志），来保证写操作的持久性。

基于上述原因，StableBuffer 在设计时，并没有被放在内存缓冲区中，而是在闪存中专门开辟一个小的临时空间。StableBuffer 可以放在为 DBMS 存储保留的空间中，或者，如果 DBMS 存储是以文件的形式进行管理的话，StableBuffer 也可以被简单地实现为一个预先分配的临时文件。这个小的临时空间，就可以被看作是一个“集中区域”，测试基准 uFLIP 的结果[BouganimJB09]显示，对于大多数闪存设备而言，针对一个集中区域（大小为 4MB-16MB）的随机写操作，几乎和顺序写操作的速度一样快。因此，为了获得高效的写操作性能，StableBuffer 的大小被限制为一个集中区域的大小，即 4MB-16MB。



图[StableBuffer-overview] StableBuffer 方法示意图

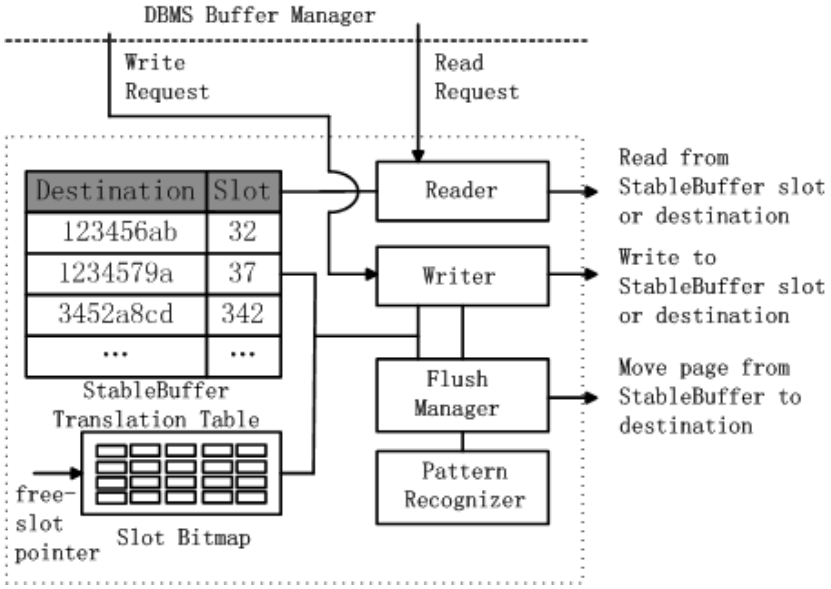
5.5.3 StableBuffer 管理器

5.5.3.1 StableBuffer 管理器的体系架构

图[StableBuffer-Manager-Architecture]显示了 StableBuffer 管理器的体系架构，主要包括以下几个组成部分：读取器（Reader）、写入器（Writer）、刷新管理器（Flush Manager）、模式识别器（Pattern Recognizer）、StableBuffer 转换表（StableBuffer Translation Table）和槽位图索引（Slot Bitmap）。

StableBuffer 管理器的各个组成部分的主要功能如下：

- **读取器**：负责响应从闪存中读取数据的请求；
- **写入器**：负责完成把 DBMS 缓冲区中被驱逐的页写入到闪存的过程；
- **模式识别器：负责对**临时存储在 StableBuffer 中的页进行模式识别，从中探测发现一些高效的写模式，比如顺序写、集中写和分区写；
- **刷新管理器：**负责把临时存储在 StableBuffer 中的页，以高效的写模式，刷新到闪存的目标地址中；
- StableBuffer 转换表：负责维护页的目标地址 *D* 到 StableBuffer 中的槽 *O* 的映射；
- **槽位图索引**：用来维护 StableBuffer 中的所有槽的状态信息，即可用或不可用。



图[StableBuffer-Manager-Architecture] StableBuffer 管理器的体系架构

5.5.3.2 StableBuffer 管理器的数据结构

在介绍 StableBuffer 管理器的具体操作之前，需要了解一下 StableBuffer 管理器中使用到的两种数据结构，即 StableBuffer 转换表和槽位图索引。

（1）StableBuffer 转换表

StableBuffer 包括许多个槽（slot），每个槽可以容纳一个页，例如，给定一个 4MB 的 StableBuffer，一个页大小是 4KB，那么，StableBuffer 就可以包含 1024 个槽。StableBuffer 转换表是一个内存中的表，维护了 StableBuffer 空间里的槽编号和目标页地址这二者之间的映射。例如，如果一个页的目标地址是 0x123456ab，它被存储在 StableBuffer 的第 32 个槽，那么，StableBuffer 转换表中就会有一个条目为 <0x123456ab,32>。图[StableBuffer-Manager-Architecture]中的 StableBuffer 转换表示例性地给出了三个条目，即<0x123456ab,32>、<0x1234579a,37>和<3452a8cd,342>。在执行页的读写操作时，StableBuffer 读取器和写入器经常需要根据一个页的目标地址在 StableBuffer 转换表中进行查找，以判定该页是否在 StableBuffer 中以及在什么位置。为了支持基于目标页地址的高效的页查找，

StableBuffer 转换表被设计成一个哈希表，哈希表的键是一个页的目标地址。

（2）槽位图索引

槽位图索引负责维护 StableBuffer 中所有槽的状态信息，从而可以快速查找 StableBuffer 中的可用槽。槽位图索引中的每个位，代表了一个 StableBuffer 槽的状态，“1”表示相应的槽是可用的，“0”表示已经被占用。为了进一步加速可用槽的查找过程，StableBuffer 还维护了一个可用槽的指针来指向下一个可用槽。

（3）额外的元数据

StableBuffer 转换表和槽位图索引等数据结构都是在内存中维护的，但是，内存是易失性存储介质，断电或系统崩溃时就会丢失信息，因此，为了保证数据库数据的一致性，能够在系统发生失败时恢复到正确的状态，StableBuffer 还设计了一些额外的元数据，比如目标地址和时间戳。

额外的元数据会和每个页一起被存放在 StableBuffer 中，因此，当系统发生失败的时候，就可以恢复数据。在把一个页写入到 StableBuffer 中之前，实际的目标地址和时间戳，将会被嵌入到页头中。一旦系统发生了失败，就可以扫描 StableBuffer，读取这些元数据来重构 StableBuffer 转换表。具体而言，在需要重构 StableBuffer 转换表时，需要一个槽一个槽地扫描 StableBuffer。对于槽 O 中的每个页，假设它的时间戳是 t_{SO}，它的目标地址是 D，目标地址 D 中的当前页的最近更新的时间戳是 t_{SD}，需要对 t_{SO} 和 t_{SD} 进行比较，如果 $t_{SO} \leq t_{SD}$，就说明这个页一定已经被刷新到目标地址中，因此，就可以把第 O 个槽的位图索引信息设置为可用状态。否则，即 $t_{SO} > t_{SD}$，这就意味着该页在系统崩溃时还没有被刷新到闪存的目标地址 D 中，因此，就需要把这个槽标记为占用状态，并且在 StableBuffer 转换表条目中插入一个新的条目<D,O>。

5.5.3.3 StableBuffer 管理器的读写操作

（1）页读操作

StableBuffer 管理器的读取器负责执行页读操作，在收到针对目标地址 D 的页的读操作请求时，读取器会通过哈希方式查找 StableBuffer 转换表。如果在 StableBuffer 转换表找到一个条目<D,O>，就读取 StableBuffer 的槽 O 中的页；否则，就直接到闪存的目标地址 D 中读取页。可以看出，在任何一种情况下，都只需要一次闪存 IO，唯一增加的开销是 StableBuffer 转换表的查找操作；但是，由于 StableBuffer 转换表是以哈希表的形式实现的，因此，这个查找操作的开销很小。

（2）页写操作

StableBuffer 管理器的写入器负责执行页写操作。当 DBMS 写缓冲区要把一个目标地址为 D 的脏页 p 驱逐出 DBMS 缓冲区时，写入器会首先在该页头部中增加一些元数据。然后，它会查找 StableBuffer 转换表，如果在某个槽 O 中找到目标地址 D，就更新这个页。否则，即在 StableBuffer 转换表中找不到目标地址为 D 的条目，则写入器会在 StableBuffer 中找到一个空的可用槽，这可以通过查找可用槽指针来实现。如果可用槽指针是 *null*，说明 StableBuffer 是满的，就会激发一个刷新进程来释放一些已经被占用的槽。当一个槽 O’被释放以后，这个页 p 就会被写入到槽 O’中，并生成一个新的条目<O’,D>插入到 StableBuffer 转换表中。最后，如果位图索引中还可以找到可用槽的话，可用槽指针会指向下一个可用槽，否则，可用槽指针设置为 *null*。可以看出，在写一个新页时，只会引起一次闪存 IO，但是，需要额外的开销来释放 StableBuffer 转换表中已经被占用的槽来获得可用槽。

5.5.3.4 StableBuffer 管理器的模式识别操作

StableBuffer 中收集了大量的写操作序列，StableBuffer 管理器的模式识别器需要对这些写操作序列进行分析，探测发现高效的写模式，比如顺序写、集中写和分区写。

StableBuffer方法采用了两种不同的方法来识别高效的写模式，即按需识别方法和增量

识别方法。按需识别模式，是在需要执行刷新操作的时候才去识别确定高效的写模式。而增量识别模式的方法，则需要不断维护关于写模式的辅助信息，每次执行针对闪存设备的写操作的时候，都需要去更新这些信息，在接收到一个页刷新请求时，可以使用辅助信息来确定高效的写模式。

（1）按需识别方法

按需识别的方法，可以很容易通过在StableBuffer转换表上的排序扫描来实现。对于不同的写模式而言，按需方法的识别不同写模式的过程如下：

- **识别顺序写模式**：通过对StableBuffer转换表中记录的目标地址进行扫描和排序，就可以很容易确定顺序写模式。多个写操作的地址相邻时，就可以构成一个地址连续的写操作序列，由此，可以形成若干个地址连续的写操作序列。具备最长连续地址的多个写操作序列，就可以被判定为“顺序写”。
- **识别分区写模式**：对StableBuffer转换表中记录的目标地址进行扫描和排序，通过对排序后的目标地址进行分析，就可以识别分区写模式。每个连续的地址序列，就是一个分区。大小相同的分区会被分到一个组，具有最多写操作数量的组，就可以在执行刷新操作时成为优先被选择的组。
- **识别集中写模式**。采用一个滑动窗口对排序后的页目标地址进行扫描，就可以探测到集中写模式。滑动窗口的大小，是和闪存设备的集中写区域大小相同的。在扫描的时候还同时需要维护一些额外的信息，包括：（1）滑动窗口的起始位置；（2）滑动窗口内的目标地址的数量。最后，包含最密集目标地址的滑动窗口就可以被确定下来，这个窗口内的操作序列就属于集中写模式。

（2）增量识别方法

增量识别方法的目标是，把确定高效写模式的代价分摊到大量写操作之中。这种方法会为增量识别高效写模式的候选实例维护一些辅助数据结构。每当一个页被插入到StableBuffer中，或者从StableBuffer中删除时，辅助数据结构都会发生更新。有了这些辅助数据结构的帮助，任何时候都可以高效地发现高效写模式。增量识别方法对于后面将要介绍的StableBuffer管理器的主动刷新操作而言是必须的，因为，主动刷新操作在任何时候一旦被触发，就要求必须可以立即获得高效写模式的实例，把它们刷新到闪存目标地址中，而按需识别方法是无法支持这一点的，只有增量识别方法才有能力做到立即为刷新操作提供高效写模式的实例。增量方法识别不同写模式的具体过程如下：

- **识别顺序写模式**：为了识别顺序写模式的实例，需要维护一个集合$SL=\{S_1,S_2,\ldots,S_i,\ldots\}$，它包含了连续地址区间，其中，$S_i=(addr_{min},addr_{max})$，表示一个地址区间。集合$SL$中的各个$S_i$是根据$S_i$中的$add_{min}$的升序顺序进行排列的。每当插入或者删除一个地址$addr$，就会触发一个在$SL$上的二分搜索，找到距离$addr$最近的$S_i$。对于一个插入操作，如果可以在已有的地址区间的基础上形成一个更长的地址区间，那么，可以把$addr$插入到距离它最近的S_i；否则，就单独创建一个只包含$addr$的新的地址区间。对于一个删除操作，包含$addr$的S_i会被分裂成两个地址区间。可以看出，每个S_i都是一个顺序写操作模式的候选实例。
- **识别分区写模式**：为了增量地识别分区写模式的候选实例，需要维护一个基于集合SL的额外的数据结构$\{P_1,P_2,\ldots,P_l,\ldots\}$，其中，$P_l$会跟踪$SL$中的所有大小为$l$的$S_i$，例如，$P_3$维护了一个指针集合，这些指针指向大小为3的所有$S_i$。每个$P_l$代表了分区写模式的一个候选实例。在发生一个页的插入或者删除操作时，在更新完SL以后，还需要继续更新受到影响的P_l。
- **识别集中写模式**：为了识别集中写模式的实例，需要维护一个关于集中写簇的集合$FL=\{F_1,F_2,\ldots,F_i,\ldots\}$，其中，$F_i=(addr_{min},addr_{max},set_{addr})$，表示落入区间$[addr_{min},addr_{max}]$

的地址的集合为set_{addr}，并且，$addr_{min}$和$addr_{max}$之间的距离，小于闪存设备集中区域的大小，因此，每个F_i是集中写模式的一个候选实例，多个F_i会根据$addr_{min}$进行升序排序。当插入一个新的地址*addr*时，就在*FL*上执行一次二分搜索，来找到最近的簇F_i。然后，如果可能的话，需要把地址*addr*合并到找到的F_i中，否则，为*addr*创建一个新的F_i。相反，对于一个删除操作而言，会把*addr*从相应的簇中删除。在StableBuffer方法中，集中写簇的识别，是采用贪婪算法实现的，这样可以降低计算代价，不过会牺牲簇的质量。为每个插入和删除操作更新辅助数据结构的时间复杂度是$O(\log n)$，其中，n是存储在StableBuffer中页的数量。每个数据结构的空间复杂度是$O(n)$。由于StableBuffer的大小通常很小，因此，空间需求是可以接受的。

（3）两种模式识别方法的比较

和按需识别方法相比，在执行刷新操作时，增量识别方法可以更加快速地确定高效的写模式，因此，增量识别方法可以取得更短的写操作响应时间。但是，增量识别方法为每个写操作都引入了一定的维护辅助信息的额外代价，因此，增量识别方法的总体代价往往高于按需识别方法，吞吐量也不如按需识别方法。此外，增量识别方法是在增量维护写模式的信息，而这个过程中，每次增量维护时，始终只能获得在维护发生时间点之前已经产生的写操作序列的信息，而无法获得在维护时间点之后产生的时间序列信息，因此，总体而言，它只是根据局部信息而不是全局信息来确定高效的写模式实例，因此，返回结果的质量可能不如按需识别方法。相反，按需识别方法在刷新操作时才会执行，相对于增量识别方法而言，在这个时间点，按需识别方法已经可以获得所有写操作序列的信息，因此，它可以根据全局信息而不是局部信息来确定高效的写模式实例，返回结果的质量较高。

5.5.3.5 StableBuffer 管理器的刷新操作

StableBuffer管理器的刷新操作，负责把临时存储在StableBuffer中的页，以高效的写模式，刷新到闪存的目标地址中。这里需要重点解决的问题就是，什么时候把StableBuffer中的页刷新到闪存的目标地址中？

对于页刷新操作执行的时机的选择，StableBuffer方法采用了两种方式，即被动方式和主动方式。

（1）被动方式

在被动方式中，对于一个写操作而言，如果在StableBuffer中已经没有额外的空间，那么，就可以把最佳写模式的实例刷新到目标地址中。一旦刷新操作被启动，被动方式会从候选实例集合中选择某些写模式的一个实例，然后对实例中包含的页执行刷新操作，写入到闪存的目标地址中。当采用按需写模式识别方法时，高效写模式的候选实例是通过扫描StableBuffer转换表来临时生成的。当采用增量写模式识别方法时，高效写模式的候选实例已经被增量地维护，因此，可以立即获得最佳写模式的实例。

（2）主动方式

在主动方式中，在任何写操作执行时，如果满足一定的条件，就可以把高效写模式的实例刷新到闪存目标地址中。在主动方式中，刷新管理器会一直在后台运行，用来探测发现高效写模式的候选实例。因此，主动方式内在地要求模式识别器采用增量识别的工作方式。对于任何写模式实例，它都会被增量地维护，一旦发生了变化，刷新管理器就会被触发。刷新管理器会检查受到影响的写模式实例是否符合刷新操作的条件，主要是根据该写模式实例的大小和类型来决定的。对于写模式 x 的某种类型，会采用一个刷新门槛值 θ，只有大小高于门槛值 θ 的实例才会被刷新到闪存的目标地址中。

5.6 本章小结

本章内容首先给出了数据存储方法概述，主要介绍了基于页的方法和基于日志的方法；

然后，介绍基于页的方法，即 DFTL，它采用了面向 DBMS 的 FTL 机制，充分利用了页级别的时间局部性来存储页面，并且更新操作可以在任何数据块上进行，提高了块利用率；接下来，介绍了基于日志的方法，包括 LGeDBMS、A/P 方法、IPL 方法和 ICL 方法等；再接下来，介绍了基于页差异的日志方法，即 PDL 方法；最后，介绍了 StableBuffer，它可以充分利用独特的闪存设备写模式，来优化写操作集中的 DBMS 应用（比如 OLTP）的性能。

5.7 习题

1、 把数据库中的数据存储到闪存中，可以采用两种方法，即基于页的方法和基于日志的方法，请阐述这两种方法的各自特点。

2、 分析为什么需要设计面向 DBMS 的 FTL 机制。

3、 阐述 PDL 方法的基本原理，以及它和基于页的方法、基于日志的方法的不同之处。

4、 分析为什么 StableBuffer 方法可以获得较好的性能。

第6章 闪存数据库缓冲区管理

在内存中设置数据库页的缓存可以有效提升数据库的性能，通过把那些经常被访问的数据库页放置到缓冲区中，当外部请求到达时，如果请求页正好在缓冲区中，那么，就不需要到辅助存储设备（比如磁盘或闪存）中读取该页，由于内存读写速度要远远高于辅助存储设备，因此，缓冲区可以明显减少 I/O 开销。

缓冲区替换策略的有效性，会严重影响到数据库系统性能。虽然，闪存比磁盘具有更好的 I/O 性能，但是，闪存仍然比内存的速度低两个数量级。一个有效的策略可以减少代价高昂的磁盘访问，从而有效提高数据库的整体性能。缓冲区替换策略是充分发挥缓冲区作用的关键，它可以在数据库服务器对外提供服务的过程中，动态决定把哪些数据放入缓冲区，哪些数据驱逐出缓冲区，从而使得数据库服务器随着负载特性的不断变化，仍能够提供高性能的服务。设计不恰当的缓冲区替换策略，不仅无法提升数据库性能，反而会导致数据库性能的恶化，因为，替换策略如果做出错误的放置和驱逐页的决定，可能会带来更多的 I/O 开销。因此，必须设计合理高效的闪存数据库缓冲区管理策略。

本章首先概要介绍缓冲区管理策略，然后介绍面向闪存的缓冲区管理策略，最后介绍一些代表性的研究成果。

概念区分：从严格意义上讲，缓存（Cache）和缓冲区（Buffer）是两个不同的概念。缓存的作用是加速“读”操作，即在内存中开辟一块空间，用来保存从磁盘上读出的数据，这些数据通常是被频繁访问的数据，由于内存访问速度要比磁盘访问速度快许多，因此，把频繁访问的数据保存到内存中，可以提高整体的数据读速度。缓冲区的作用是缓冲“写”操作，即在内存中开辟一块空间，把即将要写入到磁盘上的数据都暂时保存在这里，等到某个特定的时刻，才一次性地批量刷新到磁盘中，这样做可以降低 IO 开销，因为，每次磁盘写操作都包含了固定的磁盘启动开销，如果每次都只写入少量数据，就会带来很大的磁盘反复启动开销，而批量写入则只要一次启动开销就可以写入大量数据，从而可以把启动开销分摊到大量的数据中。在很多情况下，缓存和缓冲区这两个名词并没有严格区分，因为，很多时候缓存或缓冲区会是读写混合类型，因此，本章内容后续的论述中，统一称为“缓冲区”。

6.1 缓冲区管理策略概述

当缓冲区替换发生时，通常包括两种类型的替换代价。一种是当一个被请求的页从辅助存储中抓取到缓冲区中的代价；另一种是当一个脏页从缓存中被驱逐出来写入到辅助存储（比如闪存和磁盘）中的代价，因为，为了保证数据的一致性，一个脏的缓冲区页，当被缓冲区替换策略选中为驱逐页时，必须把该脏页回写到辅助存储。通过选择一个干净页来驱逐，可以最小化第二种代价。因为，一个干净页和闪存中原来的数据页的内容是一样的，因此，从缓冲区中被驱逐出去即可，不需要执行写入闪存的操作。

上述两种替换代价中，只要减少其中一种代价，就可以使得数据库性能获得收益，但是，两种代价不是完全独立的，即其中一种代价的增加或减少，可能在远期会对另外一种代价产生影响。例如，一个替换策略为了减少针对闪存的写操作和擦除操作代价，可能会决定把尽可能多的脏页放入缓冲区。但是，这种做法可能会让缓冲区逐渐耗尽空间，访问缓冲区时脱靶（即被请求的页不在缓冲区中）的次数就会大量增加，一旦访问缓冲区页时发生脱靶，就需要到辅助存储设备上读取相应的页，这意味着会增加从辅助存储中读取被请求页的替换代价。再比如，一个替换策略，如果只关注增加命中率，就会尽量驱逐脏页，这就会增加把脏页写入到辅助存储中的替换代价。因此，一个好的缓冲区替换策略必须综合考虑上述两种类型的代价，从而获得最好的整体性能。

最具有代表性的缓冲区替换策略就是 LRU（Least Recently Used）策略，或称为最近最少使用策略。LRU 策略有许多种不同的实现方式，最常见方式是链表法。当采用链表法实现 LRU 策略时，缓冲区中的所有页被组织成一个链表，称为 LRU 链表。LRU 链表按照页的访问频度来排列页，最近最频繁使用的页被放置在链表的 MRU（Most Recently Used）位置，最近最少使用的页被放置在链表的 LRU（Least Recently Used）位置。当访问一个页面 P 时，就到缓冲区的 LRU 链表寻找 P，如果 P 已经在缓冲区中，就把 P 从链表的当前所在位置移除，放置到链表的 MRU 位置；如果 P 不在缓冲区中，并且缓冲区还有额外的空间，则把 P 放入链表的 MRU 位置；如果缓冲区已满，则需要进行页的替换，把处于链表 LRU 位置的页作为驱逐页，驱逐出缓冲区，腾出空间后，把 P 放置到链表的 MRU 位置。一个页第一次从辅助存储（比如磁盘或闪存）读取到缓冲区时，没有经过任何修改，被称为“干净页”，一旦缓冲区中的页被修改过，就变成“脏页”。在进行缓冲区页的替换时，当一个驱逐页是干净页的时候，可以直接驱逐出缓冲区，不需要其他操作，当一个驱逐页是脏页的时候，为了保证数据一致性，必须把脏页“回写”到辅助存储中。

图[LRU]给出了一个 LRU 算法的实例。假设缓冲区最多只能容纳 5 个页。在初始阶段（见图[LRU](a)），缓冲区中包含了 P_0,P_1,P_2,P_3,P_4 共 5 个页，P_0 处于 LRU 位置，P_4 处于 MRU 位置。这时，一个针对页 P_2 的读操作到达，LRU 算法检查 LRU 链表后发现 P_2 已经在缓冲区中，就直接从缓冲区读取 P_2，P_2 被访问后，LRU 算法就把 P_2 从链表的当前位置移除，重新放置到 MRU 位置，如图[LRU](b)所示。然后，一个针对页 P_5 的读操作到达，LRU 算法检查 LRU 链表后发现 P_5 不在缓冲区中，需要到辅助存储中读取 P_5 放入缓冲区。但是，这时缓冲区已满，LRU 算法选择当前处于 LRU 位置的页 P_0 作为驱逐页，驱逐出缓冲区，然后，把从辅助存储中读取到得 P_5 放入到 LRU 链表的 MRU 位置，如图[LRU](c)所示。

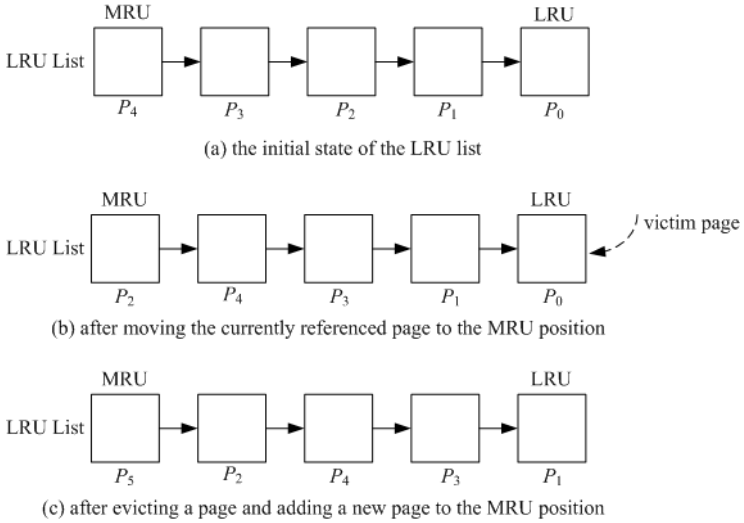


Fig.[LRU] An example of the LRU algorithm

图[LRU] LRU 算法的一个实例

6.2 面向闪存的缓冲区管理

本节介绍重新设计面向闪存的缓冲区替换策略的必要性，以及设计面向闪存的缓冲区管理策略的考虑因素和关键技术。

6.2.1 重新设计面向闪存的缓冲区替换策略的必要性

此前，许多数据库系统都使用一个近似的 LRU（Least Recently Used：最近最少使用）算法作为替换策略，取得了较好的性能。但是，以前这些替换策略都是针对磁盘这种块访问设备而设计的，替换策略只考虑缓冲区的命中率，即在给定的缓冲区尺寸下，最小化缓冲区脱靶的概率。但是，最小化缓冲区脱靶率，在基于闪存的数据库系统中，可能会带来较差的

I/O 性能。这是因为，和磁盘相比，闪存有一个内在的特性——读写不对称性，即写操作会比读操作慢一个数量级，而且由于写操作可能会引起擦除操作，这就进一步提高了写操作的代价。由于这种非对称性，缓冲区管理策略必须对写操作和读操作进行区分。例如，在缓冲区中保留一个写操作比较集中的脏页所能带来的收益，要比保留一个读操作集中的干净页的收益大。而在传统的 LRU 策略下，往往会把很多脏页替换出去，因为，LRU 替换的是链表尾部的页，而在实际应用中脏页往往容易集中在 LRU 链表的尾部。替换脏页会带来大量的 I/O 开销，这就恶化了基于闪存的数据库的性能。

因此，对于使用闪存作为辅助存储的数据库系统而言，当需要替换缓冲区页来释放缓冲区空间时，一方面，应该充分考虑闪存具有读和写操作代价不对称这个特点，尽量减少代价高昂的写操作和擦除操作的数量；另一方面，还要尽量减少缓冲区“脱靶”的数量，因为脱靶会导致大量的读操作。此外，闪存要比磁盘快一个数量级，这种低的访问延迟，不允许采用复杂的、具有高计算代价的数据结构和算法，比如，在缓冲区中选择一个驱逐页的算法应该尽可能简单，从而不会抵消由闪存低访问延迟带来的 I/O 收益。因此，设计充分考虑闪存特性的数据库缓冲区替换策略，是一件具有挑战性的工作。

6.2.2 设计面向闪存的缓冲区替换策略的考虑因素

设计面向闪存的缓冲区替换策略的一个主要目标就是，尽量减少针对闪存的写操作和擦除操作的数量。需要注意的是，缓冲区替换策略生成的 I/O 序列只会包含读写操作，而不会包含擦除操作，因为擦除操作是由闪存内部自己控制的。不过，一般来说，来自缓冲区层的写操作的数量，通常和针对闪存的物理写和擦除操作的数量成正比。因此，面向闪存的缓冲区替换策略的目标都是尽量降低来自缓冲区层的写操作数量，由写操作引发的擦除操作的数量自然就会相应降低。

针对闪存的写操作，通常都是由于替换缓冲区中的脏页引起的。当把一个脏页驱逐出缓冲区时，为了保证数据的一致性，必须把这个脏页“回写”到闪存中。因此，替换策略应该尽量让脏页在缓冲区中停留更长的时间。但是，保留更多的脏页在缓冲区中，也就意味着分配给干净页的空间就会相应减少，可能会导致缓冲区命中率的下降，引起更多的闪存读操作，恶化整体 I/O 性能。因此，缓冲区替换策略应该综合考虑脏页和干净页的驱逐代价，一个基本原则是：由于缓冲区脏页带来的写操作代价的减少，一定要大于由此带来的读操作代价的增加。比如，如果一个干净页 p 必须被反复读取 n 次，一个脏页 q 必须被反复修改 m 次，如果 n 和 m 值相差不大，那么就应该最好去替换 p 而不是 q，因为，直接在内存中服务这 m 个写操作的收益，要比 n 个闪存读操作的代价大许多。

设计面向闪存的缓冲区替换策略的基本原则是：

- 最小化物理写操作的数量，尤其是最小化随机写操作的数量；
- 调整写模式来改进写操作效率；
- 保持一个相对较高的命中率。

同时，缓冲区替换策略应该充分考虑不同操作类型对不同类型缓冲区页的操作代价，具体如下：

（1）读取干净页：直接从缓冲区中读取一个干净页，不需要到闪存读取，因此，代价非常小；

（2）读取脏页：直接从缓冲区中读取一个脏页，不需要到闪存读取，因此，代价非常小；

（3）写干净页：直接对缓冲区中的干净页进行写操作，这个页变成一个脏页，如果以后被驱逐出缓冲区，就会引起脏页回写到闪存的写操作，代价较大；

（4）写脏页：直接对缓冲区中的脏页进行写操作，相当于和之前对该页的写操作进行了合并，不管这个脏页在缓冲区停留期间发生过多少次写操作，当这个脏页被驱逐出缓冲区

时，只需要一次回写到闪存的代价，节省了若干次回写到闪存的写操作代价。因此，缓冲区替换策略让脏页在缓冲区中停留更长时间，可以获得收益。

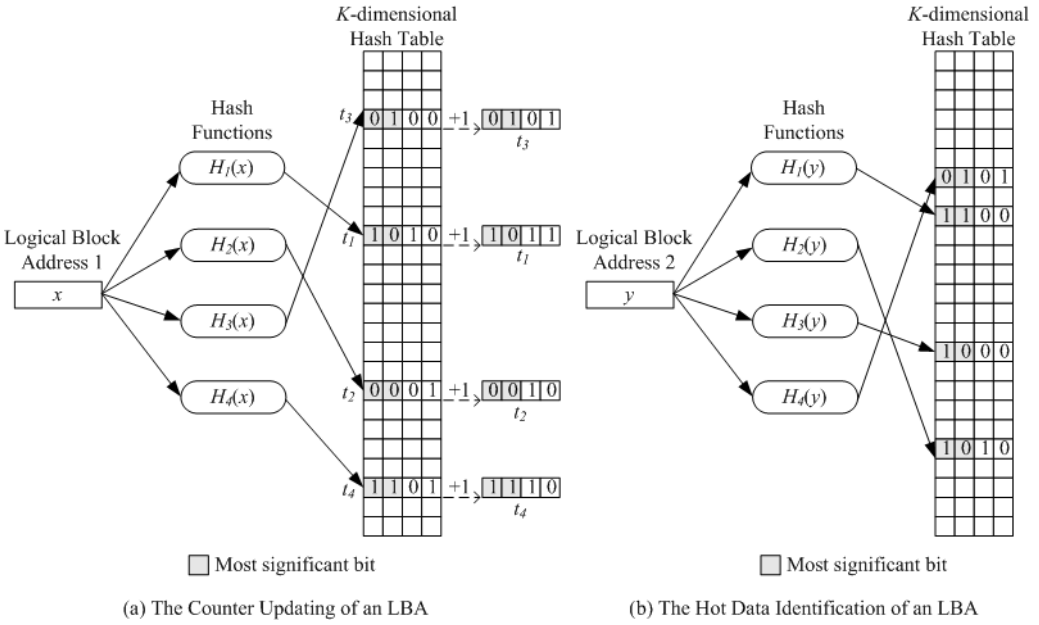
此外，每个页被访问的频度也是不同的，被频繁访问的页称为“热页”，很少被访问的页称为“冷页”。相应地，脏页可以分为“热脏页”和“冷脏页”，干净页也可以分为“热干净页”和“冷干净页”。对于不同类型的页，缓冲区替换代价也是不同的。通常有如下关系成立：Cost(CC) < Cost(CD) < Cost(HC) < Cost(HD)，其中，Cost(CC)表示替换冷干净页的代价，Cost(CD)表示替换冷脏页的代价，Cost(HC)表示替换热干净页的代价，Cost(HD)表示替换热脏页的代价。也就是说，替换热脏页的代价最大，替换冷干净页的代价最小。因此，替换策略应该优先替换冷干净页，尽量延缓替换热脏页。

6.2.3 设计面向闪存的缓冲区替换策略的关键技术

许多性能较好的缓冲区替换策略，都需要考虑页的访问频度，确定一个页属于冷页还是热页，然后，根据热脏页、冷脏页、热干净页和冷干净页的不同替换代价，设计最优的缓冲区替换策略。因此，判定一个页冷热情况的“热探测”（或也可称为“冷探测”）技术，是许多缓冲区替换策略必须采用的基本技术。

文献[HsiehCK05]提出了一种面向闪存的、基于哈希的热探测机制，是在 FTL 中实现的，可以实时跟踪记录针对各个逻辑块地址（LBA：Logical Block Address）的访问，确定哪些 LBA 包含了热数据，为数据访问的空间局部性提供高效的在线分析。大量实验表明，这种热探测技术可以获得较好的性能，为其他面向闪存的缓冲区替换策略提供辅助。

在这种热探测机制中，对于每个 LBA，都使用 K 个独立的哈希函数映射到哈希表的 K 个条目上。哈希表的每个条目（entry），都包含了一个 C 位的计数器，来跟踪针对某个 LBA 的写操作的数量。当 FTL 接收到针对某个 LBA 的写请求时，这个 LBA 会被 K 个独立的哈希函数映射到哈希表的 K 个条目上，这 K 个条目中的计数器的值就增加 1。由于计数器最多只有 C 个二进制位，因此，最大计数值为 2^C-1。计数器增加到最大值以后，就不会继续增加。另外，如果只是简单地把写操作数量累加到计数器中，那么，经过一段较长的时间后，许多计数器的值都可以累加到足够大的值，这对于判定冷热数据是没有意义的，因为，对于缓冲区替换算法而言，只有最近一段时间范围内的统计信息才是有价值的。因此，为了让计数器中统计到的写操作数量具有阶段时效性，计数器的值会进行阶段性衰减，衰减的方法是，每经过一段时间，计数器的值都会被除以 2，即所有写操作数量都随着时间流逝呈现指数级递减。由于计数器的值采用 C 个二进制位进行存储，因此，只要进行简单的向右移位操作，即可以实现除以 2 的效果。计数器一共有 C 个二进制位，其中，左边有 M 个二进制位被称为“最明显位”，它们用来辅助判定一个 LBA 是否包含热数据。一个 LBA 被判定包含热数据的条件是：对于与这个 LBA 对应的 K 个哈希表条目中的 K 个计数器而言，必须保证每个计数器的最明显位中出现至少一个 1。可以看出，最明显位的个数 M 是用来确定是否包含热数据的门槛值，也就是说，只有当所有 K 个计数器中累加得到的写操作的数量大于 $2^{(C-M)}$-1 时，才可能判定一个 LBA 包含热数据。如果其中某个计数器的最明显位都是 0，没有出现 1，就说明计数器中累计的写操作数量还没有超过这个门槛值，即 $2^{(C-M)}$-1。



图[hot-detection] 基于哈希表的热探测机制

图[hot-detection]是一个关于基于哈希表的热探测机制的实例。图[hot-detection](a)演示了计数器的更新，其中，K=4，C=4，M=2，也就是说，采用 4 个不同的哈希函数，计数器用 4 个二进制位存储，每个计数器包括 2 个最明显位，即只有当所有 4 个计数器中累加得到的写操作的数量大于 3（即 $2^{(4-2)}$-1）时，才可能判定一个 LBA 包含热数据。x 是一个 LBA，当一个针对 x 的写请求到达时，使用 4 个独立的哈希函数 H_1,H_2,H_3 和 H_4 分别把 x 映射到哈希表的 4 个不同条目 t_1,t_2,t_3 和 t_4 上，需要为这 4 个条目上的计数器增加 1。在计数器没有增加 1 之前，这 4 个计数器的 4 个二进制位分别是：t_1=(1,0,1,0), t_2=(0,0,0,1), t_3=(0,1,0,0), t_4=(1,1,0,1)。在增加 1 以后，这 4 个计数器的 4 个二进制位分别是：t_1=(1,0,1,1), t_2=(0,0,1,0), t_3=(0,1,0,1), t_4=(1,1,1,0)。计数器 4 个位中，左边 2 位是最明显位，因此，t_1 的最明显位是 1 和 0，t_2 的最明显位是 0 和 0，t_3 的最明显位是 0 和 1，t_4 的最明显位是 1 和 1。可以看出，t_1、t_3 和 t_4 的最明显位中至少有一个 1，但是，t_2 的最明显位都是 0，没有出现 1。一个 LBA 被判定包含热数据的条件是：对于与这个 LBA 对应的 K 个哈希表条目中的 K 个计数器而言，必须保证每个计数器的最明显位中出现至少一个 1。可以看出，t_2 不符合条件，因此，x 不会被判定为包含热数据。

图[hot-detection](b)演示了一个 LBA 被判定为包含热数据的情形，从图中可以看出，y 被 4 个独立的哈希函数映射到 4 个不同的哈希表条目上，这 4 个条目上的计数器分别增加 1 以后，它们的二进制位分别是(1,1,0,0)、(1,0,1,0)、(1,0,0,0)和(0,1,0,1)。每个计数器的左边 2 位是最明显位，可以看出，这 4 个计数器的最明显位中都至少出现一个 1，因此，可以判定 y 包含热数据。

基于哈希的热探测机制之所以采用 K 个独立的哈希函数，是为了减少错误识别率。哈希方法通常是把一个很大的地址空间映射到一个较小的空间上，因此，就会存在映射冲突，即不同的 LBA 被哈希函数映射到同一个哈希表条目上。当只采用一个哈希函数时，一个 LBA 被哈希函数映射到一个哈希表条目上以后，如果仅仅根据这个条目上的计数器超过事先规定的门槛值，就认为这个 LBA 包含热数据，那就有可能做出错误的识别。因为，由于存在映射冲突，多个不同的 LBA 可能被映射到同一个条目上，导致该条目上计数器数值的增加，反过来说，计数器值的增加是由多个 LBA 一起贡献的，因此，当计数器值超过门槛值时，自然就不能确切地断定是映射到该条目的其中某个 LBA 包含了热数据。而通过 K 个独立哈希函数映射得到的 K 个条目，一起来判断 LBA 是否包含热数据，就会大大降低错误识别率。

为了进一步减少错误识别率，基于哈希的热探测机制还采用了其他改进策略。在访问一个 LBA 时，并不是把与该 LBA 对应的所有 *K* 个计数器都增加 1，而只是对这 *K* 个计数器中具有最小值的那个计数器增加 1。因为，如果一个 LBA 包含热数据，那么，上面这种方式也可以最终让与这个 LBA 对应的全体 *K* 个计数器的值都超过门槛值，从而判定该 LBA 包含热数据。

6.3 代表性方法

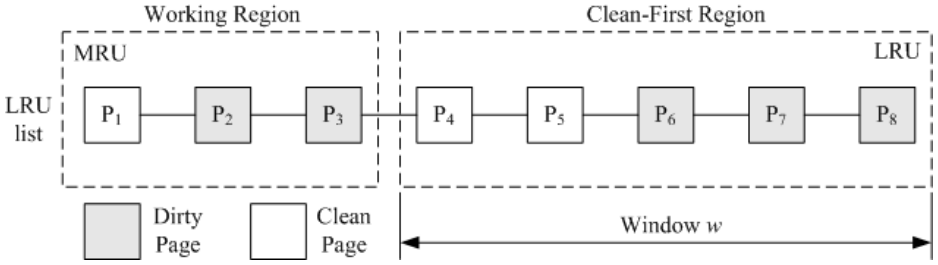
为了解决闪存盘中的读写不对称问题，研究人员已经提出了许多缓冲区管理策略[JungYSPKC07] [OnLHWLX10][ParkJKKL06][JinOHL12][KimA08] [YangWHGC11] [TangMLL11]。许多方法都继承了面向磁盘的缓冲区替换策略（比如 LRU 和 LIRS[JiangZ02]），但是，对脏页给予更高的优先级，让它有更多的机会保留在缓冲区中。因为，替换一个脏页会引起闪存的写操作，比驱逐一个干净页的代价高得多。例如，CFLRU[ParkJKKL06]替换策略把 LRU 链表分成两个区域，即工作区域和干净优先区域，干净优先区域中的干净页会被优先选择作为驱逐页，而只有当干净优先区域内没有干净页时，才会考虑驱逐脏页。CFDC[OuHJ09] [OuH10]替换策略是针对 CFLRU 替换策略的改进版本，可以获得更好的性能，包括：（1）减少了查找干净页作为驱逐页的时间；（2）采用页簇最小化脏页回写到闪存的代价；（3）尽可能避免了顺序访问模式对缓冲区性能的负面影响。CFDC 中的页簇技术，是以页簇方式驱逐脏页，可以减少总代价，提高缓冲区性能。页簇技术和缓冲区管理策略，是两个相对独立的问题，前者可以被集成到后者，来进一步改善后者的性能，因此，和文献[OuHJ09]类似，文献[SeoS08] 也采用簇脏页技术进行批量驱逐。FD-Buffer[OnLHWLX10]也采用了和 CFLRU 和 CFDC 类似的策略，把缓冲区分成两个部分，分别服务于干净页和脏页；但是，CFLRU 和 CFDC 会高度依赖于现有的特定替换策略，而 FD-Buffer 则可以对两个不同的缓冲区区域采用不同的替换策略，因此，可以灵活地直接利用不同类型的传统替换策略。LIRS-WSR[JungYSPKC07]和 LRU-WSR[JungSPKC08]分别继承自 LIRS 和 LRU，都对脏页给予了更高的优先级，如果一个脏页第二次被选中作为驱逐页，就把该脏页驱逐出缓冲区。但是，CFLRU 和 LRU-WSR 等机制都没有考虑干净页的访问频度，有时候会导致较差的 I/O 性能。因此，CCF-LRU[LiJSCY09]对 LRU-WSR 进行了改进，它根据访问的频率把干净页分成热页和冷页，在驱逐时，首先驱逐冷干净页，当冷干净页不存在时才驱逐热干净页。AD-LRU[JinOHL12]替换策略是针对 CCF-LRU 替换策略的改进版本，其主要改进体现在设法控制缓冲区中冷区的大小，避免“干净页刚读入缓冲区就被选择为驱逐页”的情况，从而提高了缓冲区的命中率。相对于 CCF-LRU 算法而言，AD-LRU 获得了更好的性能，并且在不同访问模式的工作负载下都能获得较好的性能。文献[LvCHC11]在设计替换策略 FOR 时，不仅充分考虑了闪存的特性，而且同时考虑了页面状态和未来的写操作，从而获得了更好的缓冲区性能改进。文献[OuH10]指出，其他算法都是通过优先替换干净页来减少替换脏页引起的物理写操作开销，因为，闪存的写代价明显高于读代价，但是，这些算法都没有充分考虑不同闪存设备具有不同读写代价比的情况，无法保证在不同的闪存设备上都可以获得较好的性能。因此，文献[OuH10]在设计面向闪存的缓冲区替换策略时，充分考虑了不同闪存设备的读、写代价比，以此适应不同读、写代价比的闪存设备。

6.3.1 CFLRU

文献[ParkJKKL06]在传统的 LRU 替换策略的基础上进行适当修改，设计了充分考虑闪存读写代价不对称特性的替换策略 CFLRU（Clean-First LRU），可以用在包括数据库系统在内的各种系统的缓冲区管理中。CFLRU 不仅考虑缓冲区命中率，而且考虑了驱逐脏页的替换代价，因为，从访问时间和能量消耗的角度来衡量，脏页的替换代价会比干净页的替换代价高。CFLRU 的基本思想是在缓冲区中特意维护一定数量的脏页，这就等于增加了缓冲区中脏页的空间，使得脏页在闪存中比干净页停留更长的时间（文献[KoltsidasV08s]中的缓冲

区替换策略也采用了类似的思路），从而减少闪存写操作和擦除操作的数量；但是，这会带来缓冲区命中率的恶化，因此，CFLRU 的另一个设计目标就是要保证恶化的缓冲区命中率不会降低系统的总体性能。从大量实验结果来看，CFLRU 在命中率方面有时候确实会低于普通的 LRU 算法，但是，在大多数情况下，CFLRU 确实有效地减少了写操作和擦除操作的数量。

图 [CFLRU] 给出了关于 CFLRU 算法的一个实例。LRU 链表中的页包括 P_1,P_2,P_3,P_4,P_5,P_6,P_7 和 P_8，其中，P_1,P_4,P_5 是干净页， P_2,P_3,P_6,P_7 和 P_8 是脏页。CFLRU 替换策略采用了“双区域”机制，把 LRU 链表划分成两个区域，即工作区域和干净优先区域，两个区域可以采用各种成熟的替换算法（比如 LRU）。工作区域包含了最近最常使用（MRU：Most Recently Used）的页，即 P_1,P_2 和 P_3，大多数缓冲区命中都发生在这个区域。干净优先区域包含了作为驱逐候选的页，即 P_4,P_5,P_6,P_7 和 P_8，这些页有可能被驱逐出缓冲区。干净优先区域的窗口大小是 w，必须选择合适的 w 值，因为，如果这个值太大，那么，缓冲区的命中率就会急剧下降。当请求某个页时，如果发生缓冲区脱靶，就需要启动驱逐和替换过程，CFLRU 会首先在干净优先区域中选择一个干净页来驱逐，从而减少闪存写代价。如果干净优先区域中不再有干净页，那么就在 LRU 链表中的末尾选择一个脏页来驱逐。例如，在 LRU 替换算法下，LRU 链表中的最后一个页经常被首先驱逐，因此，作为驱逐页的优先级顺序是 P_8,P_7,P_6,P_5 和 P_4。但是，在 CFLRU 替换策略下，作为驱逐页的优先级顺序是 P_5,P_4,P_8,P_7 和 P_6，也就是说，干净页 P_5,P_4 被首先驱逐，当干净页都被驱逐出去以后，才会考虑驱逐脏页 P_8,P_7 和 P_6。干净优先区域中的某个页被驱逐出去以后，就有了多余的空间来存放从工作区域驱逐出来的页，这时就可以从工作区域的 LRU 链表的尾部选择一个页，转移到干净优先区域，放在干净优先区域的 LRU 链表的头部，最后，从闪存中把被请求的页读取到缓冲区的工作区域中。



图[CFLRU] CFLRU 算法的一个实例

CFLRU 替换策略的缺陷表现在以下几个方面：

（1）需要动态调整窗口大小 w，否则就很难适应不同类型的负载，但是，根据不同类型的负载确定一个合理的窗口大小，并不是一件容易的事情，可能需要根据一些采样数据进行大量实验从而确定获得最优性能的窗口大小。

（2）没有考虑缓冲区页的访问频度，在进行驱逐替换操作时，容易保留较老的脏页，而驱逐热的干净页，这就会导致命中率的降低，会比普通的 LRU 策略带来更多的读操作，恶化总体 I/O 性能。

（3）寻找干净页的开销较大。从图[CFLRU]中可以看出，干净页并不总是出现在 LRU 链表的尾部，而是倾向于待在“干净优先区域”中靠近工作区域的位置，比如，干净页 P_4,P_5 就不在 LRU 链表的尾部，而是出现在工作区域附近。因此，在选择驱逐一个干净页时，算法就必须反向搜索 LRU 链表来获得一个干净页，而且需要反向搜索较长的 LRU 链表，时间开销比较大。

（4）没能充分利用“干净优先区域”中的脏页所占据的内存资源。在 LRU 算法的假设中，处于“干净优先区域”中的脏页被重新访问的概率更低，因此，这部分宝贵的内存资源应该被

更好地利用，用来最小化回写脏页的代价。

（5）顺序访问模式会恶化算法性能。CFLRU 具有和 LRU 同样的问题，当负载中混杂了很多较长的顺序访问模式时，它的效率会变得很低，因为，顺序访问模式所访问过的页，会把之前缓冲区中的热页都排挤出缓冲区。

（6）替换脏页时，没有区分热脏页和冷脏页，有时候会恶化性能。在替换脏页时，替换冷脏页和替换热脏页的效果是不同的，通常而言，替换冷脏页的代价要小于替换热脏页的代价，因此，应该优先替换冷脏页。但是，CFLRU 没有对脏页进行冷热区分，因此，无法获得最优的 I/O 总体性能。

6.3.2 CFDC

文献[OuHJ09]提出的 CFDC（Clean-First Dirty-Clustered）替换策略，用在数据库系统的缓冲区管理层，是针对 CFLRU 替换策略的一种改进版本，它克服了 CFLRU 存在的一些缺陷，从而获得更好的性能。

CFDC 有效解决了 CFLRU 中存在的三个问题：（1）寻找干净页的开销较大；（2）没能充分利用“干净优先区域”中的脏页所占据的内存资源来实现脏页回写代价最小化；（3）顺序访问模式会恶化算法性能。

针对 CFLRU 替换策略中存在的第一个问题，文献[OuHJ09]提出了 CFLRU 的改进算法——CFDC。如图[CFDC]所示，与采用“双区域”机制的 CFLRU 策略类似，CFDC 策略也需要管理两个区域：（1）工作区域：保存那些被频繁访问的热页；（2）优先区域：通过为不同的页赋予不同的优先级，可以优化替换代价。优先区域中包含两个队列，即干净队列和脏队列，前者用于存放干净页（即 P_4,P_5），后者用于存放脏页（即 P_6,P_7 和 P_8）。当一个页被驱逐出工作区域时，如果是一个干净页，就会进入优先区域的干净页队列，否则进入脏页队列。当发生缓冲区脱靶时，会从优先区域的干净队列的尾部选择一个干净页作为驱逐页，如果干净队列为空，就从脏队列的尾部选择一个脏页作为驱逐页。由于 CFDC 把干净页和脏页分开保存到两个不同的队列中，而不是像 CFLRU 那样统一保存在一个 LRU 链表中，因此，CFDC 查找驱逐页的过程要比 CFLRU 快许多，前者只需要找到队列的尾部就可以确定驱逐页，而后者需要沿着 LRU 链表反向查找很长的距离。

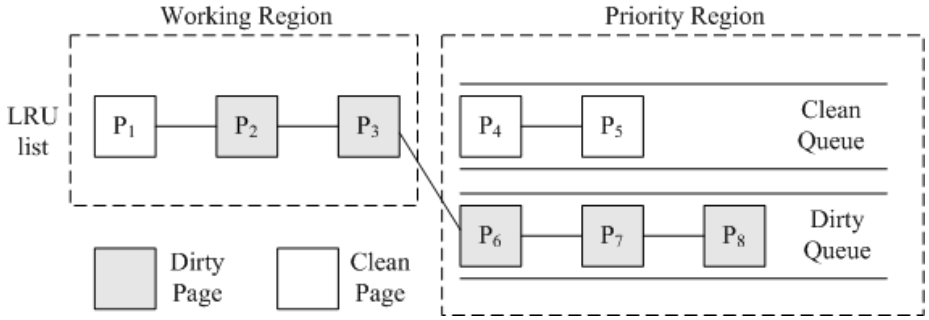


Fig. [CFDC] An example of CFDC algorithm

图[CFDC] CFDC 算法的一个实例

针对 CFLRU 替换策略中存在的第二个和第三个问题，CFDC 采用的方法是，在“脏页”队列中支持页簇，也就是说，并不是维护一个脏页的队列，而是为页簇维护一个优先级队列。一个页簇是一个页的列表，具有位置相近性，也就是一个簇内的各个页的页号彼此接近，不同簇会被赋予不同的优先级，优先级较低的簇会首先被驱逐。CFDC 设计了一个哈希表来管理这些脏页的簇，哈希表的键是簇号。当一个脏页进入优先区域时，CFDC 会首先生成它的簇号，并且使用这个簇号执行一个哈希查询。如果这个簇已经存在，这个脏页就会被加入到该簇的尾部，该簇在脏页优先级队列中的位置就会发生调整。如果这个簇还不存在，包含该脏页的一个新簇就会被创建，插入到脏页队列中，并且在哈希表中注册这个新建的簇。当访

问命中一个簇中的某个页的时候，这个页就会被转移到工作区域。在请求某个页的时候，如果发生缓冲区脱靶，就需要进行页的驱逐和替换，CFDC 会首先从优先区域的干净队列的尾部选择一个干净页作为驱逐页，驱逐出缓冲区，如果优先区域中的干净队列是空的，就选择脏页队列中优先级最低的簇中的第一个页作为驱逐页，驱逐该页后得到的空间，可以用来存放从工作区域驱逐出来的页，这时可以从工作区域的 LRU 链表的尾部选择一个页，移动到优先区域，如果这个页是个干净页，就进入干净页队列，如果这个页是脏页，就进入脏页队列的某个簇中，转移结束后，工作区域就有了剩余的空间，这时就可以把被请求的页从闪存读入工作区域。

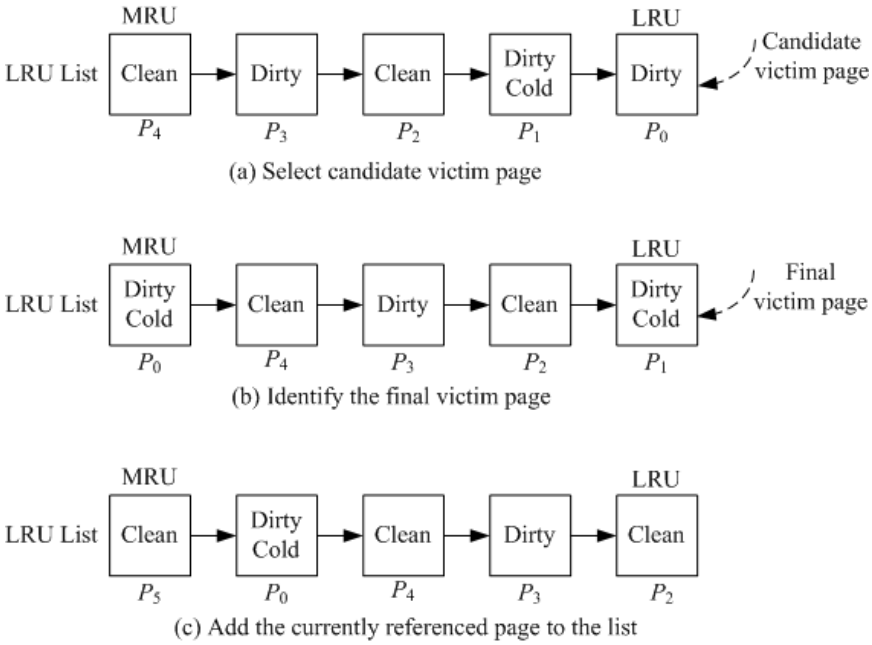
在 CFDC 替换策略中，簇越大，优先级越低，当发生缓冲区脱靶时，越会被首先驱逐。CFDC 采用这种设计策略的原因是：（1）越大的簇，空间局部性越好，写入闪存时的效率越高，因此，被驱逐出缓冲区时，会具有更小的 I/O 开销；（2）簇越大，该簇内的页以顺序访问模式为主的可能性越大，而把顺序访问模式为主的页放在缓冲区中，会恶化缓冲区的性能，为了提升缓冲区性能，应该让以随机访问模式为主的页在缓冲区中停留更长的时间。从上面论述可以看出，CFDC 替换策略可以最小化回写脏页的代价，并尽可能避免了顺序访问模式对缓冲区性能的负面影响，因此，CFDC 替换策略可以有效解决 CFLRU 替换策略中存在的第二个和第三个问题。

6.3.3 LRU-WSR

LRU-WSR 替换策略是为了克服 CFLRU 策略的缺陷而提出的，目标也是减少写操作的数量。CFLRU 算法的一个缺陷是，它会驱逐干净页，在缓冲区中尽量保留脏页，但是，没有考虑这些脏页的访问频度。对于一个很少被访问的冷脏页而言，留在缓冲区的收益很小，会降低缓冲区命中率，恶化总体 I/O 性能，因此，应该被优先驱逐。

为了克服 CFLRU 的这个缺陷，LRU-WSR 采用的策略是，延迟驱逐一个具有较高访问频度的脏页，具体而言就是：（1）使用和文献[HsiehCK05]中类似的“冷探测”技术来判断一个页是冷页还是热页；（2）如果一个页被判定为“非冷”，即该页不是冷页，就延迟把该脏页回写到闪存。使用这种策略，LRU-WSR 的命中率会比普通 LRU 算法低，会带来更高的物理读操作。但是，这种算法有效减少了写操作和擦除操作的数量。因此，它提高了基于闪存的存储系统的整体 I/O 性能。需要注意的是，LRU-WSR 是基于二次机会算法[Sliberschantz04]的启发式算法。在理论上，很难证明驱逐脏页可以提高总体 I/O 性能，但是，在实验结果上，确实可以看出 LRU-WSR 能够有效减少写操作和擦除操作的数量，同时不会导致命中率的严重恶化。

在 LRU-WSR 中，缓冲区中的所有页构成一个链表，每个页都有“冷页标记位”，但是，替换算法只会检查脏页的“冷页标记位”，而不会检查干净页的“冷页标记位”。如果一个页是冷页，就设置冷页标记位，如果一个页不是冷页（或称为“非冷页”），就清除冷页标记位。在进行缓冲区页替换时，当一个脏页 *P* 被选中作为候选驱逐页时，需要首先检查 *P* 的冷页标记位。如果 *P* 的冷页标记位没有被设置，*P* 就会被移动到链表的 MRU 位置，并且设置它的冷页标记位，然后把链表的 LRU 位置的页作为候选驱逐页。如果 *P* 是脏页，并且已经设置了冷页标记位，说明 *P* 是一个冷脏页，就会被驱逐出缓冲区，回写到闪存。如果 *P* 是一个干净页，就不会考虑它的冷页标记位，直接被驱逐。当链表中的一个页被访问时，这个页就会被转移到链表的 MRU 位置，如果该页是脏页，还要清除它的冷页标记位。



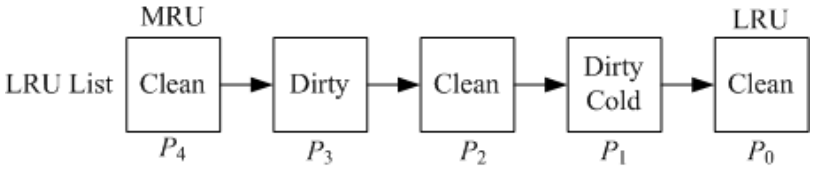
图[LRU-WSR] LRU-WSR 算法的一个实例

图[LRU-WSR]给出了 LRU-WSR 算法的一个实例。如图[LRU-WSR](a)所示，缓冲区的 LRU 链表包括 5 个页，即 P_0,P_1,P_2,P_3,P_4，其中，P_2 和 P_4 是干净页，P_0,P_1,P_3 是脏页，P_1 被设置了冷页标记位，是一个冷脏页，P_0 和 P_3 还未设置冷页标记位，都是非冷页。假设缓冲区只能容纳 5 个页，当访问第 6 个页 P_5 时，由于 P_5 不在缓冲区中，就会发生缓冲区页的替换。在替换缓冲区页时，从链表的 LRU 位置选择 P_0 作为候选驱逐页（见图[LRU-WSR](a)），P_0 的冷页标记位还未设置，说明 P_0 是非冷页，于是就把 P_0 移动到 MRU 位置，并且设置 P_0 的冷页标记位，P_1 变成了 LRU 位置，如图[LRU-WSR](b)所示。然后，LRU-WSR 算法选择处于 LRU 位置的页 P_1 作为候选驱逐页，由于 P_1 已经为设置了冷页标记位，说明 P_1 是一个冷脏页，因此，会被驱逐出缓冲区，回写到闪存中。最后，把 P_5 放入缓冲区的 LRU 链表的 MRU 位置。

LRU-WSR 的缺陷是没有考虑干净页的访问频度。在进行替换驱逐的时候，如果一个候选驱逐页是干净页，就不会考虑它的冷页标记位，直接被驱逐，很有可能被驱逐的是热干净页。和驱逐冷干净页相比，驱逐热干净页的代价更大。驱逐热干净页可能会引起额外的读操作，因为，热干净页是未来很可能会被再次访问的页，被驱逐出去以后，下次再被访问时，就需要再次从闪存读取。此外，在 LRU-WSR 中，可能出现一些最近只被访问一次的冷干净页污染缓冲区的情况，导致冷脏页和热干净页更快地被驱逐出去。而已有的研究[LiJSCY09]表明，驱逐冷脏页和热干净页的代价都要比驱逐冷干净页的代价大。

[LRU-WSR-shortcoming]是说明 LRU-WSR 算法缺陷的一个实例。在 LRU 链表中，一共包括 5 个页，即 P_0,P_1,P_2,P_3,P_4，其中，P_0,P_2 和 P_4 是干净页，P_1,P_3 是脏页，P_1 被设置了冷页标记位，是一个冷脏页，P_3 还未设置冷页标记位，都是非冷页。假设缓冲区只能容纳 5 个页。当访问第 6 个页 P_5 时，由于 P_5 不在缓冲区中，就会发生缓冲区页的替换。在替换缓冲区页时，根据 LRU-WSR 算法，首先从 LRU 链表中选择 LRU 位置的页 P_0 作为候选驱逐页，P_0 是一个干净页，可以直接驱逐，然后，把 P_5 放入到链表的 MRU 位置。接下来，假设需要再次访问 P_0，由于刚才已经把 P_0 驱逐出缓冲区，因此，就会再次发生页的替换。算法会选择 P_1 作为候选驱逐页，P_1 是一个脏页，需要检查冷页标记位，发现是一个冷脏页，就可以直接驱逐出去，把 P_1 回写到闪存，并从闪存中读取 P_0 到缓冲区中，放到链表的 MRU 位置。从上述过程可以看出，为了完成对 P_5 和 P_0 的访问，总共发生了一次写操作（把脏页

P_1 回写到闪存）和一次读操作（从闪存读取 P_0 到缓冲区）。实际上，在驱逐 P_0 时，如果能够考虑到它是一个热干净页，就不应该直接驱逐，因为驱逐热干净页会带来额外的开销，代价较大；更好的做法是，在驱逐 P_0 时，如果发现它是一个热干净页，就不驱逐，而是继续寻找下一个候选驱逐页，找到 P_1 后，发现是一个冷脏页，就可以直接驱逐出去。这样，下次再访问 P_0 时，P_0 仍然在缓冲区中，不需要到闪存读取 P_0。在这种新的替换策略下，完成对 P_5 和 P_0 的访问就只需要一次写操作（把脏页 P_1 回写到闪存），很显然，代价比 LRU-WSR 算法更低。



图[LRU-WSR-shortcoming] LRU-WSR 算法缺陷的一个实例

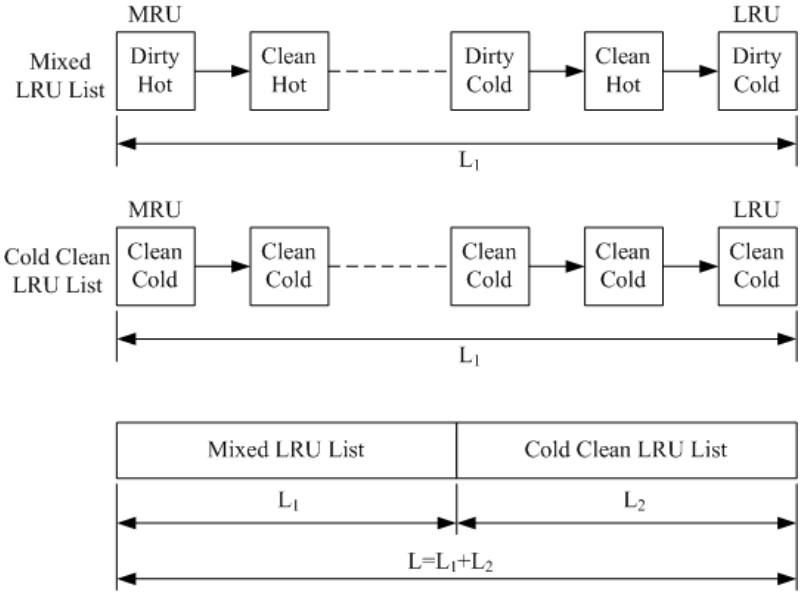
6.3.4 CCF-LRU

CFLRU 和 LRU-WSR 都使用干净优先机制来首先驱逐缓冲区中的干净页。虽然这些方法比传统的 LRU 策略性能好，但是，它们都没有考虑干净页的访问频度，有时可能会导致较差的 I/O 性能。为了解决这个问题，文献[LiJSCY09]提出了新的缓冲区替换策略 CCF-LRU（Cold-Clean-First LRU），它可以改善之前的 CFLRU 和 LRU-WSR 算法的性能。CCF-LRU 的主要目标就是通过减少写操作的次数来改进整体 I/O 性能。

CCF-LRU 算法的主要思想是：使用“冷探测”机制来判断一个页是热页还是冷页；在替换操作时，尽可能优先驱逐冷干净页，尤其是那些最近只被访问过一次的干净页；如果不存在上述冷干净页，就驱逐冷脏页，而不是驱逐热干净页。

CCF-LRU 采用上述策略是因为，驱逐一个热脏页的代价是最大的，驱逐一个冷干净页的代价是最小的，驱逐热干净页的代价，比驱逐冷干净页的代价大。为了提高整体 I/O 性能，就必须尽量减少驱逐热脏页和热干净页的次数。

图[CCF-LRU]显示了 CCF-LRU 算法的数据结构，包括两个链表，即 ML 链表（Mixed LRU List）和 CCL 链表（Cold Clean LRU List）。ML 链表用于维护热干净页和脏页（包括热脏页和冷脏页），CCL 链表用于维护冷干净页。ML 链表可以容纳 L_1 个页，CCL 链表可以容纳 L_2 个页，并且有如下关系成立：$L_1+L_2=L$，其中 L 表示缓冲区可以容纳的页的数量。也就是说，ML 链表和 CCL 链表共享缓冲区空间，二者的尺寸 L_1 和 L_2 都位于区间[0,L]内，在算法运行过程中，二者的尺寸会在区间[0,L]内动态变化，但是，一直都会满足条件 $L_1+L_2=L$。



图[CCF-LRU] CCF-LRU 算法的数据结构

在 CCF-LRU 算法中，第一次被访问的页都默认被看成是冷页，每个页都被插入到 CCL 链表中，并设置冷标志位。当 CCL 链表中的页再次被访问或者变成脏页时，它就会被移动到 ML 链表的 MRU(Most Recently Used)位置。当 ML 链表中的某个页被访问的时候，它会被移动到 ML 链表的 MRU 位置。当需要进行页的替换时，就需要选择一个驱逐页，CCF-LRU 确定驱逐页的过程如下：

（1）如果 CCL 链表不为空，就把 CCL 链表中处于 LRU 位置的页作为驱逐页；

（2）如果 CCL 链表为空，就把处于 ML 链表的 LRU 位置的页作为候选驱逐页，如果这个候选驱逐页是一个冷脏页，就选定作为最终驱逐页，把它从缓冲区中驱逐出去；如果这个候选驱逐页是一个热脏页，就把它标记为“冷”，并且移动到 ML 链表的 MRU 位置。如果候选驱逐页是一个热干净页，就标记为“冷”，从 ML 链表中移除，移动到 CCL 链表的 MRU 位置，然后继续检查 ML 链表的 LRU 位置。如果在遍历一次 ML 链表以后没有找到驱逐页，就需要再次调用 CCF-LRU 算法重新寻找驱逐页。

CCF-LRU 算法的缺陷是无法探测热干净页。CCF-LRU 在设计上采用了两个链表，即 CCL 链表（Cold Clean LRU List）和 ML 链表（Mixed LRU List）。在进行页的替换时，算法会首先选择 CCL 链表中的冷干净页进行驱逐，随着系统的运行，CCL 链表会逐渐缩小，甚至最终可能变成空。在这种情况下，当一个新的读页插入到 CCL 链表中以后，当再有一个新的页需要缓冲区空间的时候，这个读页就会被立即驱逐。因此，除非一个页被连续两次读取，否则，一个干净页根本没有机会成为热页。由此导致的结果是，当脏页完全占据缓冲区以后，干净页是很难再获得缓冲区的。

6.3.5 AD-LRU

相对于 CCF-LRU 算法而言，AD-LRU[JinOHL12]算法在性能有了进一步的提升，该算法设法控制缓冲区中冷区的大小，以此避免“一个干净页刚读入缓冲区就被选择为驱逐页”的情况。AD-LRU 主要思想如下：

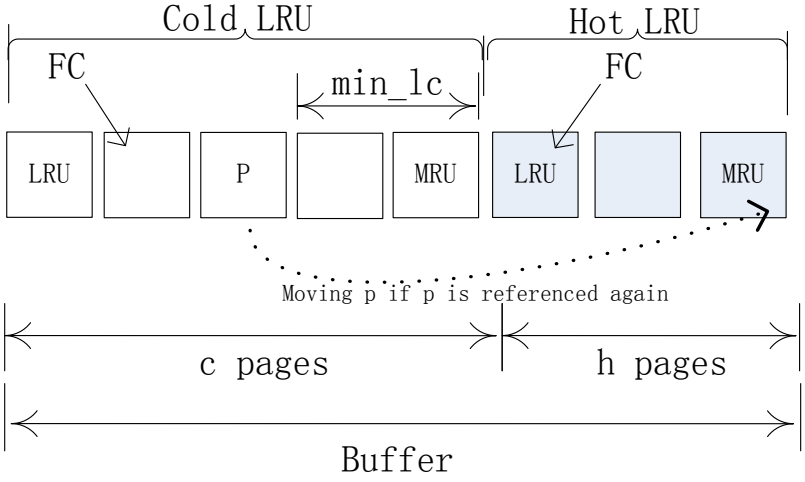
(1) 将缓冲区分为冷区和热区，冷区中存放那些只访问过一次的页，热区中存放那些至少访问过两次的页；

(2) 冷、热区的大小是动态调整的。当冷区中的页被命中时，就将该页转移到热区，此时，冷区容量减少，热区容量增加；当替换操作发生在热区时，AD-LRU 会在热区中选择驱逐页，并将新读入的页存到冷区的 MRU 位置，此时，热区容量

减少，冷区容量增加；

(3) 冷区容量有一个下界(*min_lc*)，当冷区的容量大于等于 *min_lc* 时，替换操作发生在冷区，当冷区的容量小于 *min_lc* 时，替换操作发生在热区；

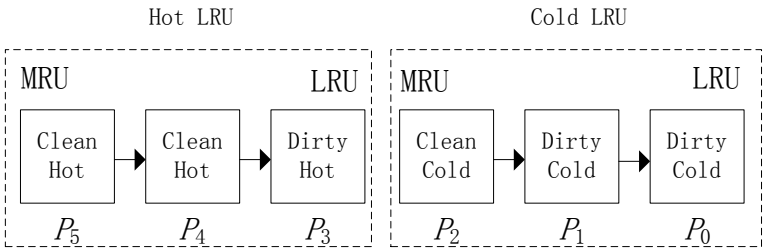
(4) 替换操作无论发生在冷区还是在热区，都优先替换干净页，若冷区中没有干净页，则其它页的替换顺序和 LRU 算法一样，若热区中没有干净页，则使用二次机会策略替换脏页。



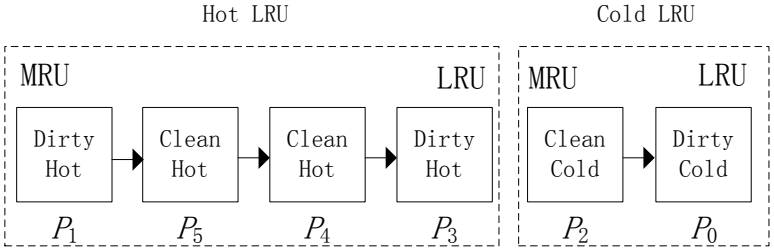
图[AD-LRU] AD-LRU 算法的数据结构

如图[AD-LRU]所示，AD-LRU 算法将缓冲区分为冷区和热区，且冷、热区的大小是动态调整的。第一次引用一个新页时，它将会被插入到冷区的 MRU 位置，缓冲区中的页被再次引用时，它将会被转移到热区的 MRU 位置。参数 *min_lc* 用于防止冷区不断缩减，最后出现冷区为空的情况，从而发生干净页刚读入缓冲区就被选择为驱逐页的情况。变量 *FC* 用于指向最近最少使用的干净页，以此加快驱逐页的查找速度。

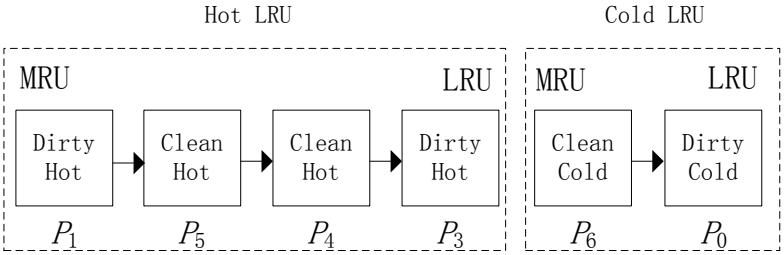
图[AD-LRU-example]给出了一个 AD-LRU 算法的实例。假设缓冲区最多只能同时容纳 6 个页，且冷区的下界 *min_lc* 为 2。在初始阶段（见图[AD-LRU-example](a)），缓冲区中包含了 P_0,P_1,P_2,P_3,P_4,,P_5 共 6 个页，其中，P_0 和 P_1 是冷脏页，P_2 是冷干净页，P_3 是热脏页，P_4 和 P_5 是热干净页。这时，一个针对页 P_1 的读操作到达，AD-LRU 算法检查缓冲区后发现 P_1 已经在缓冲区中，就直接从缓冲区读取 P_1，P_1 被访问后，AD-LRU 算法将 P_1 转移到热区的 MRU 位置，如图[AD-LRU-example](b)所示。然后，一个针对页 P_6 的读操作到达，AD-LRU 算法检查缓冲区后发现 P_6 不在缓冲区中，需要到辅助存储中读取 P_6 放入缓冲区。但是，这时缓冲区已满，需要驱逐一页以腾出空间存放 P_6，AD-LRU 算法先判断冷区是否达到下界，以确定是在冷区中选择驱逐页还是在热区中选择驱逐页，由于此时冷区中依然有两个页，大于等于冷区的下届 *min_lc*，所以 AD-LRu 算法在冷区中选择驱逐页。AD-LRU 算法总是在冷区中优先选择干净页，只有冷区中没有干净页时，才会选择脏页作为驱逐页，在本例中，AD-LRU 算法将会选择冷干净页 P_2 为驱逐页，驱逐出缓冲区，然后，把从辅助存储中读取到得 P_6 放入冷区的 MRU 位置，如图[AD-LRU-example](c)所示。



(a) The initial state of AD-LRU



(b) after moving the currently referenced page to the MRU position of hot region



(c) after evicting a page and adding a new page to the MRU position

图[AD-LRU-example] AD-LRU 算法的一个实例

AD-LRU 虽然获得了比 CCF-LRU 更好的性能，但是，AD-LRU 算法存在以下三个问题：（1）它并没有彻底解决 CCF-LRU 的缺陷，这是因为 AD-LRU 在冷区中总是优先替换干净页，随着系统的运行，脏页可能完全占据冷区。当脏页完全占据冷区时，会出现与 CCF-LRU 一样的问题，即一个干净页刚读入缓冲区就被选择为驱逐页，从而导致该页没有机会成为热页，降低了缓冲区的命中率；（2）在系统运行一段时间以后，冷热区的容量趋于稳定，此时，缓冲区中冷、热区的容量不再是动态调整的；（3）AD-LRU 也很难根据不同访问模式的工作负载确定一个合理的冷区下界（*min_lc*）。确定一个合理的冷区下界并不是一件容易的事情，可能需要根据一些采样数据进行大量实验，从而确定获得最优性能的冷区下界（*min_lc*）。

6.3.6 FOR

现有的面向闪存的缓冲区替换算法，虽然改善了基于闪存的数据库系统的性能，但是，它们只是简单地考虑了当前页面的状态，忽略了未来操作的时间局部性。对于那些不存在未来写操作或者已经长时间没有活动的页，应该被驱逐出去，从而释放宝贵的缓冲区资源。因此，页面状态和未来操作对于缓冲区管理而言都是非常重要的。

因此，文献[LvCHC11]提出了一个新的缓冲区替换算法——FOR（Flash-based Operation-aware buffer Replacement），核心思想是基于操作感知的页权重策略来进行缓冲区替换。缓冲区中的每个页都会被赋予一个权重，权重较低的页面会首先被驱逐。在计算页权重时，综合考虑了页面的状态（干净页还是脏页）以及页的热度，而页的热度是与页的操作

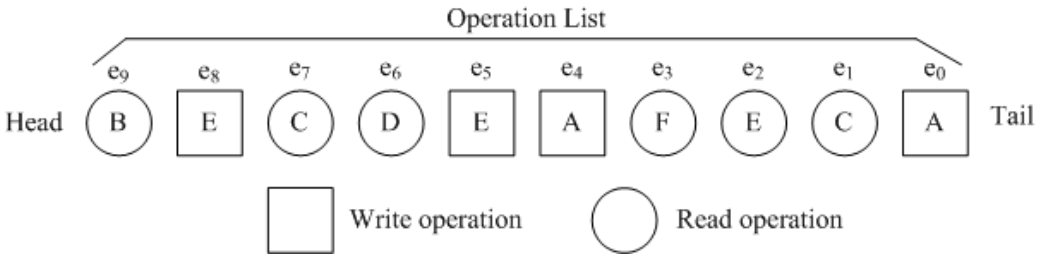
统计信息有关的。一个页 p 的权重 W_p 的计算公式如下：

$$W_p=\begin{cases}\frac{C_r}{H_r}+\frac{C_r}{H_w}, p\ is\ clean\\ \frac{C_r}{H_r}+\frac{(C_r+C_w)}{H_w}, p\ is\ dirty\end{cases}$$ 公式(FOR)

其中，C_r 表示从闪存中读取一个页的代价，C_w 表示向闪存写入一个页的代价。H_r 表是一个读操作的热度，H_w 表示一个写操作的热度。

一般而言，把热页（被频繁访问的页）保存在缓冲区中，可以明显提高命中率。在已有的研究中，已经有许多种方法可以用来衡量缓冲区页的热度，最常见的衡量方法是频度（frequence）和新颖度（Recency）。文献[LvCHC11]的作者采用了全新的缓冲区页热度计算方法，综合考虑了频度和新颖度。由于闪存读写操作具有不同的特性，因此，作者对一个页分别定义了读操作热度和写操作热度。在操作热度中综合考虑频度和新颖度。作者定义了两个概念，即操作间距离（IOD:Inter-operation Distance）和操作新颖度（OR:Operation Recency）来分别反映一个页操作的频度和新颖度。

在一个由多个操作构成的操作序列中，一个操作 e 的操作间距离 IOD_e 是指，针对一个页的两个同样的操作之间所间隔的不同操作的数量，如果这个操作序列中只包含一个操作 e，即操作 e 只出现过 1 次，那么，IOD_e 的值就为空。*IODe* 的值越小，操作 e 的热度越高。一个操作 e 的操作新颖度 OR_e 是指，在操作 e 出现以后执行的不同操作的数量。OR_e 的值越小，说明操作 e 越新，即操作 e 执行的时刻距离当前时刻越近。



图[FOR] 采用 FOR 算法计算 IOD 和 OR 的一个实例

图[FOR]显示了采用 FOR 算法计算 *IOD* 和 *OR* 的一个实例，图中的操作链表显示了一系列针对不同的页的读写操作。对于某个操作 e 而言，OR_e 就是位于表头和 e 之间的所有不同操作的数量，比如对于操作序列中的第一个操作 e_0 而言，是对页 *A* 的写操作，这时，LRU 链表只有一个元素 e_0，OR_{e0} 的值为 0 和 IOD_{e0} 为空(NULL)，当操作序列中的第二个操作（对页 C 的读操作）e_1 到达以后，LRU 链表的表头变成 e_1，表尾是 e_0，因为从表头到操作 e_0 之间的所有不同操作就只有 e_1，因此，OR_{e0} 的值就变为 1，由于此时还没有再次出现和 e_0、e_1 相同的操作，因此，这时不需要计算 e_0 和 e_1 的 *IOD* 值。当操作 e_2 到达时，OR_{e0} 的值变成 2，OR_{e1} 的值变成 1。可以看出，每个新操作到达后，之前操作的 *OR* 值都会增加 1，*OR* 的值越大，表示这个操作越老。依此类推，每操作 e_4 到达时，这个操作是针对页 *A* 的再次写操作，即 e_4 和 e_0 是针对同一个页 *A* 的两个同样的操作，因此，这时就可以计算 IOD_{e4} 的值，由于操作 e_0 和 e_4 之间所包含的所有不同操作是 e_1、e_2 和 e_3，因此，IOD_{e4} 的值就是 3，同时还要更新 e_3、e_2、e_1 和 e_0 的 *OR* 值。依此类推，每当一个新的操作到达时，都可以去调整相应的 *OR* 或 *IOD* 的值。当 e_7 到达时，可以计算得到 IOD_{e7} 的值为 5，因为 e_7 和 e_1 都是针对页 *C* 的读操作，属于针对同一个页面的相同操作，e_7 和 e_1 之间间隔了 e_2、e_3、e_4、e_5 和 e_6 共 5 个操作。当 e_8 到达时，可以计算得到 e_8 的 *IOD* 值为 2，因为，e_5 和 e_8 都是针对页 *E* 的写操作，属于针对同一个页面的相同操作，e_5 和 e_8 之间包括 e_6 和 e_7 共计 2 个操作。当最近的一个操作 e_9 到达时，当前各个操作的 *IOD* 值是：IOD_{e0}=*NULL*，IOD_{e1}=*NULL*，IOD_{e2}=*NULL*，IOD_{e3}=*NULL*，IOD_{e4}=3，IOD_{e5}=*NULL*，IOD_{e6}=*NULL*，IOD_{e7}=5、IOD_{e8}=2，IOD_{e9}=*NULL*；

当前各个操作的 *OR* 值是：OR_{e0}=9，OR_{e1}=8，OR_{e2}=7，OR_{e3}=6，OR_{e4}=5，OR_{e5}=4，OR_{e6}=3，OR_{e7}=2，OR_{e8}=1，OR_{e9}=0。

IOD 和 *OR* 在衡量相关页的热度方面是互补的，需要综合考虑。*OR* 只反映这个页操作的新颖度，而忽略了长期负载模式。而只使用 *IOD* 也是有问题的，因为，没有包含最近的操作信息。因此，针对一个操作 *e*，综合考虑 *IOD* 和 *OR* 后，计算得到操作 *e* 的操作热度 H_e 是：

$$He = \alpha * IODe + (1-\alpha) * ORe$$

其中，α 是加权系数，用来调整 IOD_e 和 OR_e 的相对重要性。

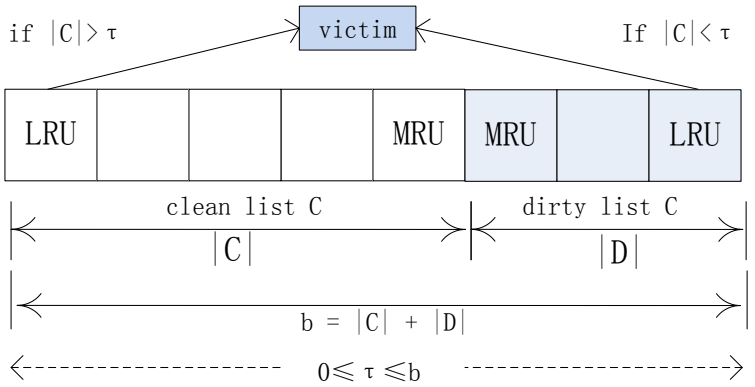
FOR 克服了 CCF-LRU 的明显缺陷。CCF-LRU 在设计上的一个不足之处就是无法探测热干净页。相反，FOR 则不存在这个问题，因为即使在该页被驱逐以后，算法仍然通过 IOD 和 OR 值来维护着与它相关的操作信息。因此，对于 FOR 而言，当一个干净页再次被读取的时候，它就有机会成为热页。

6.3.7 CASA

文献[OuH10]指出，其他算法都是通过优先替换干净页来减少替换脏页引起的物理写操作开销，因为，闪存的写代价明显高于读代价，但是，这些算法都没有充分考虑不同闪存设备具有不同读写代价比的情况，无法保证在不同的闪存设备上都可以获得较好的性能。因此，文献[OuH10]在设计面向闪存的缓冲区替换策略时，充分考虑了不同闪存设备的读、写代价比，以此适应不同读、写代价比的闪存设备。

现有的面向闪存的缓冲区替换算法都是优先替换干净页，让脏页有更多的机会驻留缓冲区。但是，这些算法都没有充分考虑不同闪存设备具有不同读写代价比的情况，无法保证在不同的闪存设备上都可以获得较好的性能。对于不同的设备而言，读、写代价的比例差异很大。当把以上方法应用于读、写代价差异较小的闪存时，在不考虑干净页操作代价的情况下就无条件优先置换干净页是不合理的，甚至可能出现性能下降的问题，从而无法适用于不同种类的闪存磁盘。

文献[OuH10]提出了一种基于代价的自适应缓冲区替换策略，该策略能够适用于不同读、写代价比的闪存设备。CASA 算法的主要思想如下：（1）将缓冲区分为干净页链表(clean list C)和脏页链表(dirty list D)，干净页链表和脏页链表的大小是动态调整的；（2）维护干净页链表的目标大小(τ)，如果干净页链表的目标大小大于实际大小（|C|），就在干净页链表中选择驱逐页，否则，在脏页链表中选择驱逐页，以此来动态调整干净页链表和脏页链表的大小；（3）在干净页链表和脏页链表中都使用 LRU 算法选择驱逐页；（4）根据数据的访问模式和闪存的读、写代价比共同决定 τ 值的变化。在 CASA 算法中，当缓冲区满时，干净页链表的大小（|C|）和脏页链表的大小(|D|)满足如下等式：|C|+|D|=b(缓冲区的容量)，$0 \le |C| \le b$，$0 \le |D| \le b$。如图所示：



图[CASA] CASA 算法的数据结构

在 CASA 算法中，τ 为干净页链表的目标大小，且有 0≤τ≤b，|C|为干净页链表的实际大小；b–τ 为脏页链表的目标大小，|D|为脏页链表的实际大小。CASA 算法选择驱逐页的过程如下：先判断干净页的实际大小是否大于目标大小，如果干净页的实际大小大于目标大小，即满足|C|>τ，则在干净页链表中选择驱逐页，否则，就在脏页链表中选择驱逐页（此时有|D|>b-τ）。

CASA 算法的自适应性主要体现在能够动态调整干净页链表的目标大小 τ 的值，并通过 τ 的值确定在干净页链表中选择驱逐页还是在脏页链表中选择驱逐页，以此来控制干净页链表和脏页链表的实际大小。该算法通过相对成本收益(relative cost effectiveness)适应不同的工作负载，通过归一化闪存的读、写代价，适应不同读、写代价比的闪存设备。

根据相对成本收益适应不同的工作负载的原理如下：请求页在干净页链表中命中时，说明对于当前的工作负载，对干净页的操作更多，就增加干净页链表的目标大小，即增加 τ 的值。此时的相对成本收益为|D|÷|C|。同理，当脏页命中时，说明对于当前工作负载，脏页的操作更多，则减少 τ 的值，此时的相对成本收益为|C|÷|D|。相对成本收益决定了每次调整 τ 值的多少，当干净页链表很小时，|D|÷|C|是一个很大的值，则此时会迅速增加 τ 的值，当干净页链表很大时，|D|÷|C|是一个很小的值，此时就微调 τ 的值。同理，在脏页链表中命中时，需要减少 τ 的值，也受到脏页链表的相对成本收益|C|÷|D|的控制。

CASA 算法通过归一化闪存的读、写代价来控制 τ 值，以此适应不同读、写代价比的闪存设备。该算法通过归一化闪存的读代价(C_R)和写代价(C_W)，使得 C_R+C_W 等于 1。读写代价比和相对成本收益共同决定了 τ 值的调整。当请求页在干净页链表中命中时，CASA 算法增加 τ 的值，且每次增加 C_R*(|D|÷|C|)；当请求页在脏页链表中命中时，CASA 算法减少 τ 的值，且每次减少 C_W*(|C|÷|D|)。

CASA 算法引入了相对成本收益的概念，所以能够避免干净页链表或脏页链表为空的极端情况。比例，在一段时间内，算法持续只读操作，那么缓冲区中命中的页都是干净页。干净页的持续命中会导致 τ 值的持续增加，τ 增加以后会有干净页链表的实际大小大于目标大小，即|C|<τ，这将导致接下来的替换操作一直发生在脏页链表中，从而使得干净页链表越来越大，脏页链表越来越少，即干净页链表占据缓冲区的大部分空间，此时，干净页链表的实际大小|C|是一个很大的值，脏页链表的实际大小|D|是一个很小的值。当脏页链表很小时，就会出现最近刚被访问的脏页马上就被替换的情况，从而降低了命中率。但是，CASA 算法能够快速的恢复到混合工作负载模式。假设在持续一段时间的只读操作以后，干净页链表占据了缓冲区的大部分空间，此时如果有一个脏页命中，就会使得 τ 值减少，且减少 C_w*(|C|÷|D|)，因为此时干净页链表占据了缓冲区中大部分空间，即|C|比|D|大很多，所以|C|÷|D|是一个很大的值，因此，C_w*(|C|÷|D|)也是一个很大的值，当 τ 的值减少 C_w*(|C|÷|D|)以后，τ 就恢复到一个适当大小的值，即算法快速地恢复到了混合工作负载的状态，避免脏页链表太小而导致最近刚使用的脏页马上就被替换的情况。同理，算法在持续一段只写操作以后，也能够快速地恢复到混合工作模式的状态。

CASA 算法充分考虑了不同闪存设备的读、写代价比，以及不同工作负载的读写比例，所以能够适应于不同的工作负载和不同读、写代价的闪存设备。该算法的主要缺陷是没有考虑页的访问频度，每个页的访问频度是不同的，任何缓冲区替换策略都应该尽可能将访问频度高的页保留在缓冲区中，以获得较高的命中率。

6.3.8 综合实例

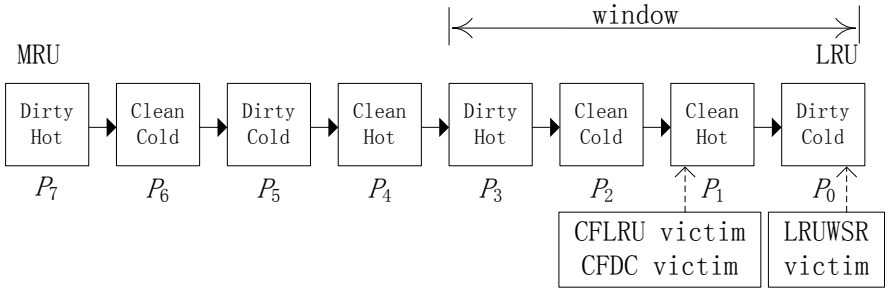
图[replace-example]展示了不同的缓冲区替换算法选择驱逐页的过程，包括 CFLRU、CFDC、LRU-WSR、CCF-LRU 和 AD-LRU。在这个例子中，假设缓冲区最多只能同时容纳 8 个页，它们最后的访问顺序是 P_0，P_1，P_2，P_3 ，P_4 ，P_5 ，P_6 ，P_7。其中，P_0 和 P_5 是冷脏页，P_1 和 P_4 是热干净页，P_2 和 P_6 是冷干净页，P_3 和 P_7 是热脏页。假设此时一个访问 P_8

的请求到达，由于缓冲区已满，所以需要将缓冲区中的一个页驱逐出去，以腾出空间存放 P_8。

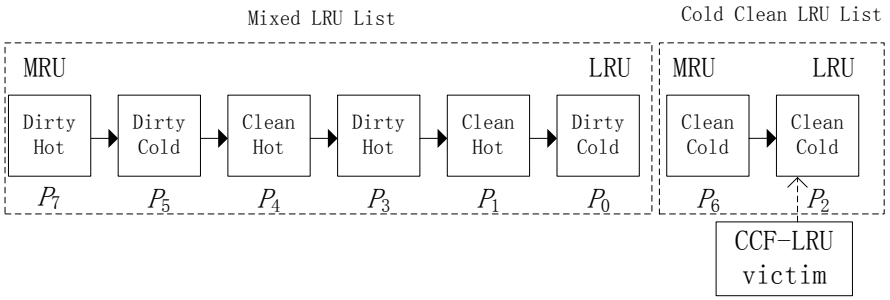
图[replace-example] (a)展示了 CFLRU、CFDC 和 LRU-WSR 算法选择驱逐页的结果。CFLRU 算法将缓冲区分为工作区域和干净优先区域，该算法在选择驱逐页时，总是在干净优先区域中优先选择干净页，如果干净页优先区域中没有干净页，则选择 LRU 位置的数据页为驱逐页。在本例中，CFLRU 算法将选择热干净页 P_1 为驱逐页。CFLRU 算法选择一个热页作为驱逐页，是因为该算法没有考虑数据访问的频度，虽然 P_1 是热页，但是它是干净优先区域中最旧的干净页，所以被 CFLRU 选择为驱逐页。

CFDC 算法是对 CFLRU 算法的改进。相对于 CFLRU 算法而言，CFDC 有以下三方面的优势：（1）能够以更快的速度选择驱逐页；（2）通过页簇技术，减少脏页回写代价；（3）能够避免一次扫描操作对缓冲区的污染。CFDC 中的页簇技术只有在替换脏页时有效，在本例中，CFDC 的选择和 CFLRU 一样，即选择热干净页 P_1 为驱逐页（如图[replace-example] (a)所示）。

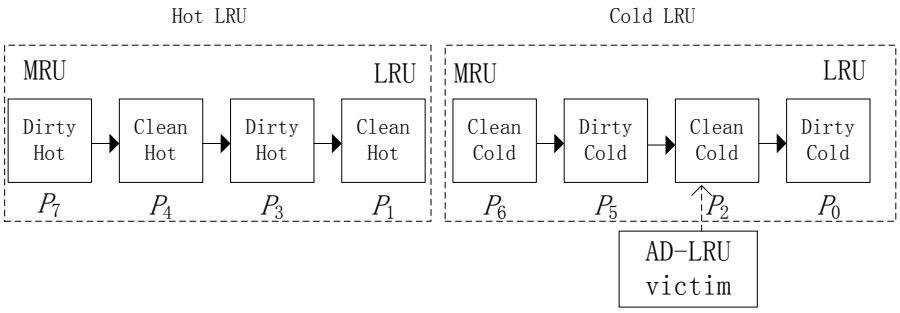
LRU-WSR 算法相对于 CFLRU 有了进一步的改进，该算法考虑了脏页的访问频度，使得热脏页能够更长时间地驻留缓冲区，因此，提高了缓冲区命中率，减少了写操作次数，从而获得了整体性能的提升。LRU-WSR 算法在选择驱逐页时，先获取 LRU 位置的数据页:（1）如果该页是冷标记位为 0 的脏页，则认为该页是热脏页，就给该页第二次机会，将其冷标记位设置为 1，并转移到 MRU 位置，然后，再次选择 LRU 位置的数据页进行判断；（2）如果选择的数据页是冷标记位为 1 的脏页或干净页，就直接替换。根据上述替换过程，在本例中，LRU-WSR 算法会选择冷脏页 P_0 为驱逐页（如图[replace-example] (a)所示）。



(a) Victims selected by CFLRU ,CFDC and LRU-WSR



(b) Victim selected by CCF-LRU



(c) Victim selected by AD-LRU

图[replace-example] 缓冲区替换算法的一个综合实例

图[replace-example] (b)展示了CCF-LRU算法选择驱逐页的结果。CCF-LRU算法将缓冲区分为ML链表（Mixed LRU List）和CCL链表（Cold Clean LRU List），其中，ML链表中存放热干净页和脏页，CCL中存放冷干净页。CCF-LRU算法总是优先替换CCL链表中的页，只有CCL链表为空时，才会在ML链表中选择驱逐页，在本例中CCF-LRU会选择冷干净页P_2为驱逐页。

图[replace-example] (c)显示了AD-LRU算法的选择驱逐页的过程。AD-LRU将缓冲区分为热区(Hot LRU)和冷区(Cold LRU)，热区中存放至少访问过两次的页，冷区中存放只访问过一次的页。AD-LRU算法无论在冷区还是热区，总是优先替换干净页，当冷区中没有干净页时，其他页的替换顺序与LRU算法一样，当热区中没有干净页时，则使用二次机会策略替换其他页。在本例中，AD-LRU算法会在冷区选择冷脏页P_2为驱逐页。

根据前文所述，在面向闪存的缓冲区替换算法中，数据页的替换代价有如下关系：Cost(CC) < Cost(CD) < Cost(HC) < Cost(HD)。所以，一个好的缓冲区替换算法应该优先替换冷干净页，在不存在冷干净页时，再替换冷脏页，并尽可能让热数据页更长时间地驻留内存。根据这个判断标准，在本例中，数据页被替换的先后顺序依次是 P_2，P_6，P_0，P_5，P_1，P_4，P_3 和 P_7，即 P_2 应该最先被替换，P_7 应该最后被替换。由此可以看出，CF-LRU 和 CFDC 作出了最差选择，而 CCF-LRU 和 AD-LRU 作出了最优的选择。

6.4 本章小结

本章内容首先给出了缓冲区管理策略概述，描述了最具有代表性的 LRU 算法的基本原理；然后，指出了重新设计面向闪存的缓冲区替换策略的必要性，因为，闪存和磁盘具有截然不同的技术特性，面向磁盘特性开发的缓冲区替换算法无法在闪存设备上获得好的性能；接下来，分别介绍了设计面向闪存的缓冲区替换策略的考虑因素和关键技术；最后，介绍了闪存数据库的缓冲区管理方面的代表性方法，包括 CFLRU、CFDC、LRU-WSR、CCF-LRU、AD-LRU、FOR、CASA 等，并给出了一个综合实例。

6.5 习题

1、 说明重新设计面向闪存的缓冲区替换策略的必要性。

2、 阐述设计面向闪存的缓冲区替换策略的考虑因素。

第 7 章 闪存数据库索引

索引是数据库系统组织管理数据的一种重要方式，可以有效提升数据库查询的性能。当前主流的商业化数据库管理系统，大都采用了 B-树及其变种索引结构，但是，如果把这些索引结构直接移植到闪存数据库上，将无法获得好的性能，因为，闪存具有和传统磁盘截然不同的物理特性。闪存采用异地更新的方式，每个 B-树中节点的更新，都会引起闪存写操作，而且，某个节点的更新可能会不断传播扩散到其他节点，引起其他节点的更新，从而导致更多的闪存写操作。因此，必须针对闪存特性设计相应的索引结构。

本章介绍修改索引模块的相关代表性研究，包括基于日志的 B-树索引[WuCK03][LeeL10]、B+-树索引[OnHLX09][WuKC07]和自适应 B+-树索引、FD-树[LiHLY09]以及 LA-树[AgrawalGSDS09]等。

7.1 基于日志的B-树索引

7.1.1 B-树

B-树是一种平衡的多路查找树，主要用于文件的索引。B-树结构特性：一棵 m 阶 B-树，或为空树，或为满足下列特性的 m 叉树(m≥3)：

(1)根节点只有 1 个，键(key)的个数的范围是[1,m-1]，分支数量的范围是[2,m]；

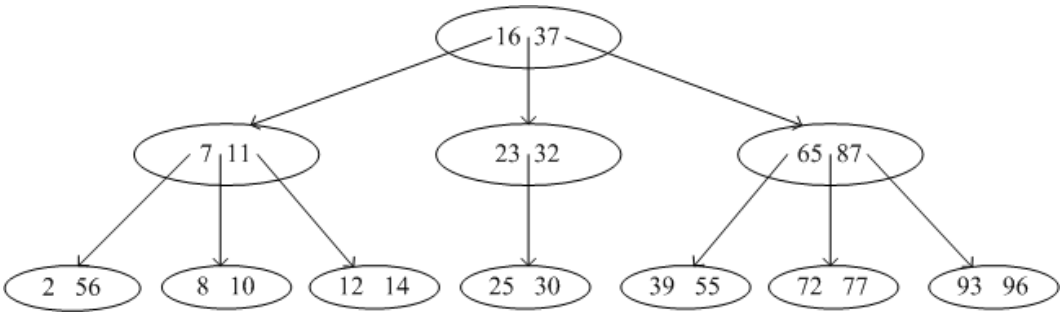
(2)除了根节点以外的所有非叶子节点，每个节点包含分支数范围是[[m/2],m]，即键的个数范围是[[m/2]-1,m-1]，其中，[m/2]表示取大于 m/2 的最小整数；

(3)非叶子节点是由叶子节点分裂而来的，所以，叶子节点的键的个数也满足[[m/2]-1,m-1]；

(4)所有的非叶子节点包含信息：(n，A_0，K_1，A_1，K_2，A_2，……，K_n，A_n)，其中，K_i 为键，A_i 为指向子树根节点的指针，并且 A_{i-1} 所指向子树中的键均小于 K_i，而 A_i 所指的键均大于 K_i（i=1，2，……，n）。n+1 表示 B-树的阶，n 表示键的个数；

(5)所有叶子节点都在同一层，并且指针域为空。

图[B-tree]给出了一棵 3-阶 B-树的实例，从中可以看出，被索引的键值分散在所有节点中，可以为每个节点中的每个键值增加一个指向该键值所对应的数据记录的指针，从而可以在 B-树索引中找到某个键值以后，根据该指针找到对应的数据记录。由此也可以看出，在数据库或文件系统中，B-树索引和被其索引的数据记录是分开存放的。



图[B-tree] 一棵 3-阶 B-树的实例

B-树中的查找、插入和删除操作的基本过程如下：

- 查找：从根节点出发，自顶向下遍历，当到达一个节点时，如果该节点中不存在查找值，就根据查找值和键值的关系确定一棵子树，继续到子树中查找，直到找到为止。
- 插入：首先，选择一个合适的叶子节点 u，在 u 中添加要插入的键值，如果 u 中键的个数不超过 m-1 个，则插入成功。否则，就需要把 u 分裂为两个节点，并把中间

的一个键拿出来插到节点 *u* 的父节点中去。父节点也可能是满的，就需要再分裂，继续再往上插。在最坏的情况下，这个过程可能一直传递到根节点，如果需要分裂根节点，由于根节点是没有父节点的，这时就需要建立一个新的根节点。由此可以看出，插入操作可能导致 B-树朝着根节点的方向生长。

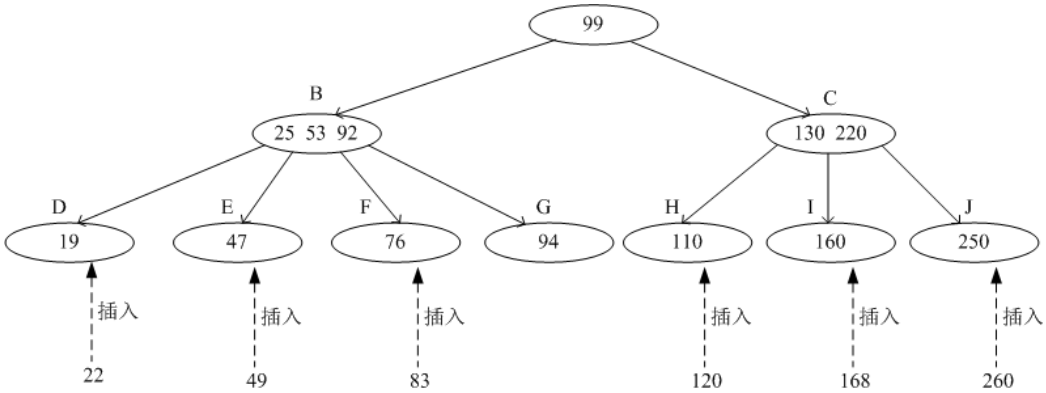
- 删除：B-树中的删除操作与插入操作类似，但要稍微复杂些。如果删除的键不在叶子节点层，那么，就需要先把此键与它在 B-树里的后继对换位置，然后再删除该键。如果删除的键在叶子节点层，则把它从它所在的节点里去掉，这可能导致此节点所包含的键的个数小于[*m*/2]-1。这种情况下，考察该节点的左或右兄弟，从兄弟节点移动若干个键到该节点中来，使两个节点所含的键的个数基本相同。如果兄弟节点的键个数也很少，刚好等于[*m*/2]-1 时，这个移动就不能进行。在这种情况下，要把将删除键所在的节点、它的兄弟节点以及它们的父节点中的一个键，合并为一个节点。

可以看出，B-树中进行插入和删除操作，都可能会引起节点的分裂或合并。

7.1.2 闪存中基于日志的 B-树索引

大多数主流的基于磁盘的 DBMS 产品都采用了 B-树作为索引结构，因为 B-树的特点就是适合在磁盘等存储设备上组织动态查找表。采用 B-树索引时，B-树索引结构自身和被其索引的数据，是分开存储的，并且都保存在磁盘上。

闪存的特性与磁盘具有很大的区别，当前的 DBMS 中所用的索引都是针对磁盘设备设计的，如果直接应用到采用 FTL 机制的闪存设备上，不仅会导致数据库性能的恶化，也会降低闪存设备的可靠性（不合理的频繁擦除操作会让闪存变得不稳定），因此，索引机制就会成为数据库性能的瓶颈。比如，在 B-树中，记录插入、记录删除和 B-树重组织，都会引起相关的写操作。如图所示，假设现在要在这棵 B-树中插入新的键值 22、49、83、120、168 和 260，这些键值会被分别插入到节点 D、E、F、H、I 和 J 中，因此，一共有六个节点的内容发生了更新。由于 B-树索引和被其索引的数据记录是分开存放的，因此，这里只关注 B-树索引。假设每个 B-树节点都单独存放在一个页中，那么，上述六次更新操作，修改了六个 B-树节点，因此，就需要对六个页进行更新。在一些情形下，如果因为插入操作导致节点的分裂，则会涉及更多的节点更新，那么就需要更新更多的页。



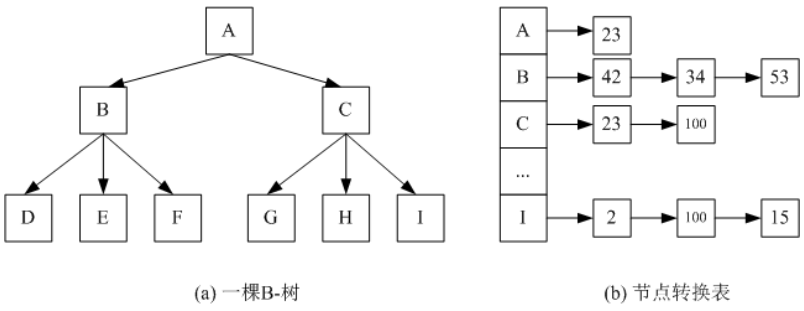
图[B-tree-insert] 在意棵 B-树中插入新的键值

由于闪存采用异地更新方式，就需要把那些没有发生变化的数据和相关节点中的树指针，也一起复制到新的闪存页中，这会导致大量的数据复制操作。而且 B-树常常会在同一个位置发生频繁更新，这进一步恶化了闪存的性能，因为，这会导致很多代价高昂的擦除操作。FTL 机制虽然可以隐藏闪存特性，但是，其核心功能是提供逻辑地址到物理地址的映射，当 B-树索引对某些逻辑地址空间进行频繁更新时，FTL 机制也无法避免闪存整体性能的恶化[LeeL10]。

因此，文献[WuCK03]提出了面向闪存的、基于日志的 B-树索引，该工作的目的就在于尽量减少索引结构造成的冗余的闪存写入操作，从而改进系统性能并且降低电能消耗。基于日志的 B-树索引方法不仅可以用于 DBMS 的索引，也可以应用在其他采用 B-树索引的应用系统中。

在闪存上直接建立 B-树索引，面临的一个棘手的问题就是级联更新[Xiang09]。闪存通常采用异地更新的方式，当 B-树中的某个节点被更新时，并非使用新版本的数据来覆盖旧版本的数据，而是把新版本的数据写入到一个新的空闲闪存页中，然后，修改上层节点指向该节点的指针，让它指向最新版本的数据。但是，当上层节点指针发生修改时，又会导致再上层节点的指针的修改，依此类推，直到最终修改了根节点的指针，也就是说，每次某个节点的更新操作，都会导致从该节点到根节点的路径上的所有节点都发生更新，这就是“级联更新”。可以看出，级联更新会引发多次闪存写操作，代价较大。针对级联更新的问题，Kang 等人[KangJKK07]提出了的一种可行的策略，即 μ-树索引，其核心思想是：把 B-树中属于同一条路径上的所有节点都存储在闪存的同一个页中，这样，每次更新这条路径上的某个节点时，就只需要写入一个闪存页，从而在一定程度上避免了节点的级联更新问题，同时也减少了索引的更新代价。但是，B-树中存在很多条路径，因此，这种方法的存储空间开销较大，管理起来也比较复杂。

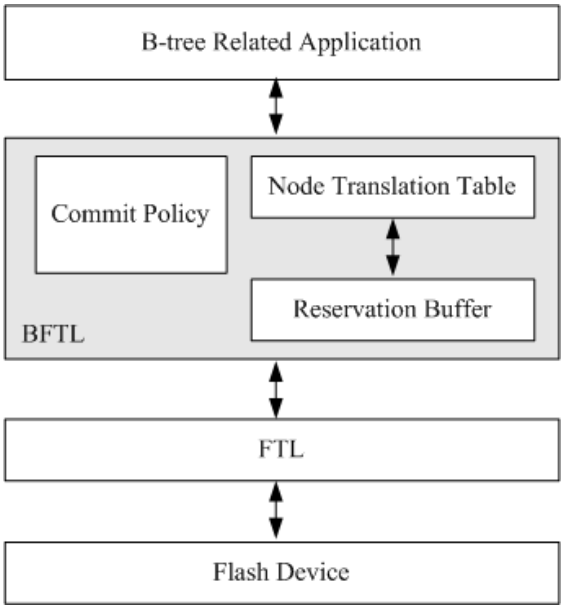
图[BFTL]给出了一种基于日志的 B-树索引方法的体系架构——BFTL[WuKC03]，该方法可以被封装成一个单独的、可插拔的模块，因为该模块处于 FTL 层之上，并且基于 B-树索引结构，因此被命名为 BFTL。BFTL 模块可以为那些采用 B-树索引的应用提供服务，因此，也可以用在现有的采用 B-树作为索引结构的 DBMS 产品中，改善 DBMS 在闪存平台上的性能。如图[BFTL]所示，上层应用提出的数据访问请求，会被 BFTL 处理和转换后发送到 FTL。BFTL 包含了一个保留缓冲区和一个节点转换表。保留缓冲区是一个为脏记录设置的缓冲区空间，可以避免在记录更新时引起 B-树索引结构对闪存的频繁写操作，减小了闪存开销。当上层应用执行插入、删除或修改记录的操作时，新生成的记录会被保存到 BFTL 的保留缓冲区，由于保留缓冲区只能容纳一定数量的记录，因此，必须及时刷新到闪存中，同时更新节点转换表。为了把保留缓冲区中的脏记录刷新到闪存中，BFTL 为每个脏记录构建了相应的索引单元。索引单元的大小比一个闪存页要小很多，如果每个索引单元都分开存储，会浪费大量的闪存空间。因此，作者设计了提交策略(Commit Policy)，可以把许多索引单元会智能地封装成一些页，从而减少物理页的数量，减低了闪存读写开销。但是，封装操作会使得属于同一个 B-树节点的多个索引单元被分散存储到不同的闪存页中。BFTL 中的节点转换表的作用，就是帮助 BFTL 识别来自同一个 B-树节点的索引单元，如图[BFTL-node-translation-table]所示，转换表中记录了 B-树中每个节点所对应的页的逻辑页号，在执行 B-树的查询时，只需要查找节点转换表就可以直接查询到节点被存储到的所有页面的逻辑页号，并通过 FTL 机制找到相应的物理页，依次读取每个闪存物理页就可以获得相应的 B-树节点，不需要到闪存中进行遍历，从而提高了闪存数据库 B-树索引的性能。



图[BFTL-node-translation-table] BFTL 的节点转换表

图[BFTL-node-translation-table](b)给出了一个可能的节点转换表实例，从中可以看出，每个 B-树节点都可以包含若干个索引单元，这些索引单元可能来自不同的页，这些页的逻辑页号被组织成一个链表，被链接到与该节点对应的转换表条目中，比如节点 B 的索引单元链表中包含了 42、34 和 53 三个链表元素，与节点 B 对应的转换表条目就保存一个指向该链表的指针。当访问某个 B-树节点时，通过扫描各个链表中所保存的逻辑页号，就可以获得属于该节点的所有索引单元。例如，为了构建图[BFTL-node-translation-table](a)中的关于节点 C 的逻辑视图，就可以通过扫描节点转换表，获得逻辑页号 23 和 100，并通过 FTL 机制找到相应的物理闪存页，然后读取相应的索引单元，从而最终构建得到节点 C 的逻辑视图。由此也可以看出，一个逻辑页中可以包含若干个来自不同 B-树节点的索引单元，比如，逻辑页号为 100 的逻辑页，包含了来自 B-树节点 C 和 I 的索引单元。

BFTL 的提交策略在设计时追求的另一个目标就是，让属于同一个 B-树节点的多个索引单元被分散存储到尽可能少量的闪存页中，从而提高索引单元的访问效率。从以上论述可以看出，BFTL 通过采用缓冲区和提交策略，有效减少了记录更新引起的 B-树索引结构对闪存的读写开销。



图[BFTL] 基于日志的 B-树索引方法的体系架构

在 BFTL 方法中，读取记录的过程如下：

（1） 一个应用发起一个读取某个记录的请求；

（2） 如果能够从保留缓冲区中找到这个记录，就把这个记录返回给应用；

（3） 否则，在节点转换表中，从根节点开始遍历整个 B-树，来搜索这个记录

（4） 如果找到这个记录，就把这个记录返回给应用。

BFTL 方法通过在应用层和 FTL 之间增加一个 B-树转换层，有效降低了闪存数据库中建立 B-树索引时存在的级联更新问题，减少了由于 B-树节点更新而引起的大量闪存写入问题。但是，基于日志的 B-树索引方法也存在一些缺陷，具体如下：

（1）虽然 BFTL 中的节点转换表可以记录 B-树节点及其对应的闪存页，但是，在查询某个 B-树节点时，由于一个 B-树中的节点可能被分散存储到不同的闪存页中，因此，为了访问一个节点，就需要进行多次读操作来访问多个闪存页，而且，由于闪存页中的索引记录对应的键值不是按照顺序存储的，无法采用二分法查找，需要遍历闪存页才能找到对应的索引记录。

（2）需要使用额外的开销来管理节点转换表，尤其是保留缓冲区中内容变化比较频繁

时，需要更多的节点转换表管理维护开销。此外，对闪存数据库的 B-树索引进行查询操作时，读取转换表对于提高查询效率的作用并不是很大。

（3）BFTL 中的缓冲区也仅仅起到了缓冲的作用，当缓冲区被索引单元填满时，仍然需要把缓冲区中所有的索引单元都刷新写入到闪存中。

（4）保留缓冲区中可能会包含冗余的数据，这也增加了刷新到闪存时的开销。比如，某索引记录先执行了插入操作，然后又执行了删除操作，此时，该索引记录在 B-树中并不存在，因此，与插入和删除相对应的两个索引单元不需要写入到闪存，而 BFTL 并无法避免这种冗余数据。

（5）没有对所有可用的各类闪存设备和负载类型进行优化。对于板上闪存芯片而言，或者在写负载比较集中时，该方法会表现出较好的性能。但是，当负载类型是读写类型时，或者应用在一个 CF 卡类型的闪存存储设备上时，该方法的性能就很差。

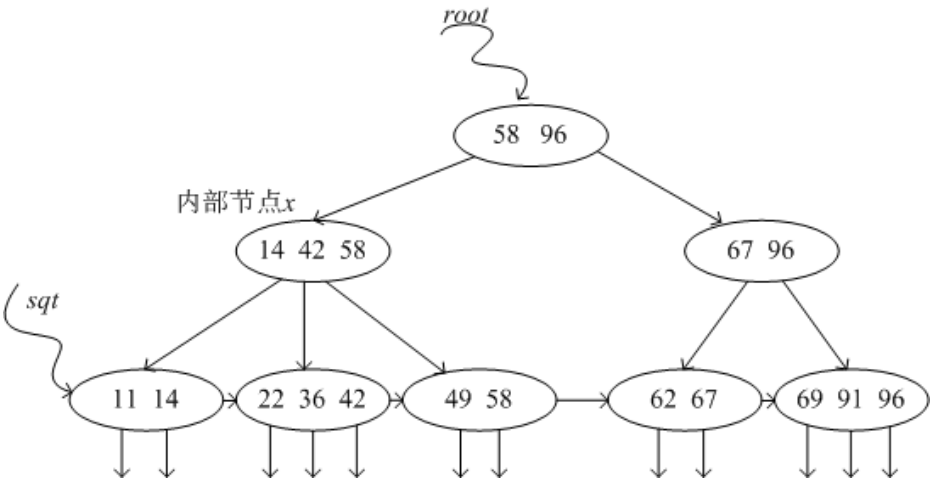
为了克服上面的第四个缺陷，文献[NathK07]提出了采用自适应 B+-树索引的 FlashDB（在下文将会介绍）。为了克服上面的前三个缺陷，文献[LeeL10]提出了针对 BFTL 的改进方法——IBSF，它的核心思想是：（1）把所有属于同一个 B-树节点的索引单元都写入到同一个页中，因此，IBSF 就不再需要节点转换表，也就省去了节点转换表的管理维护开销；（2）删除了保留缓冲区中冗余的索引单元，这就可以有效减少刷新到闪存时所涉及的写操作的数量。和 BFTL 一样，IBSF 也是一个位于存储层和应用层之间的一个软件模块，因此，也可以以可插拔的方式应用于 DBMS 中。

7.2 B+-树索引

7.2.1 B+-树

B-树索引在随机查询方面具备较好的性能，但是，无法提供较好的顺序查询性能，而作为 B-树的变种，B+-树则既可以具备较好的随机查询性能，也可以提供较好的顺序查找性能，因为，在 B+-树中，叶子节点是连接到一起的，很容易实现顺序查询。

B+-树是在 B 树基础上改进得到的动态数据结构，可以很好地适应应用结构的变化，不仅易于管理，而且具有较高的效率，通常用于数据库和操作系统的文件系统中。如图[B+-tree]所示，在 B+-树中包含两类节点，即叶子节点和内部节点（非叶子节点）。所有的内部节点可以看成是索引部分，起到指引的作用，用来定位找到某个叶子节点。每个内部节点的元素充当分开它的子树的分离值。例如，在图[B+-tree]中，对于内部节点 *x* 而言，有三个子节点（或子树），那么 *x* 的两个分离值是 14 和 42。在 *x* 的最左子树中所有的值都小于 14，在中间子树中所有的值都在 14 和 42 之间，而在最右子树中所有的值都大于 42。叶子节点中包含了记录的键的值、指向记录的指针和指向下一个叶子节点的指针，叶子节点本身按照键的大小自小而大顺序链接。叶子节点中包含了指向记录的指针，记录会被保存到磁盘的其他区域，因此，B+-树索引的存储和被其索引的记录的存储是分离的。B+-树可以支持顺序查找和随机查找。在进行顺序查找时，只需要从相应的叶子节点出发（比如从图[B+-tree]中 *sqt* 指针指向的叶子节点出发），沿着每个叶子节点中包含的指向下一个叶子节点的指针，逐个进行查找。在进行随机查找时，需要从根节点（图[B+-tree]中 *root* 指针指向的节点）出发，沿着内部节点中包含的指针，一步步定位至下一层节点，直至到达叶子节点。



图[B+-tree] 一棵 3-阶 B+-树实例

B+-树的查询、插入和删除操作的基本过程如下：

- 查询：B+-树的所有叶子节点包含了所有记录的键，因此，要查询一个记录是否存在，只需要看是否能够最终在叶节点中找到待查询的记录的键。
- 插入：插入一个数据 *d* 时，首先需要找到 *d* 所属的叶子节点，如果叶子节点有多余的存储空间，就直接把 *d* 插入到该叶子节点，否则，就将该叶子节点进行分裂，由一个叶子节点分裂成两个叶子节点，并重新分配相关数据。如果根节点发生了分裂，由于一棵树中只能有一个根节点，则会产生一个新的根节点，树的高度增加 1。
- 删除：删除一个数据 *d* 时，同样需要首先找到 *d* 所属的叶子节点，然后删除该叶子节点。在删除该叶子节点以后，如果该叶子节点包含的键的数量过少，则会发生相邻叶子节点的合并或者在相邻叶子节点之间重新分配数据。

7.2.2 闪存中的 B+-树索引

通常情况下，B+-树节点会保存在辅助存储介质上，同时，为了避免重复从辅助存储介质上读取数据，一般会把最近访问的节点缓存在内存中，这样，在某段时间内，节点的更新就可以一起被提交。但是，内存容量是有限的，只能缓存少量节点，因此，被缓存的节点通常在接收到足够多的请求一起提交之前就会被交换出内存。因而，仅仅依赖于这样一个缓存机制不大可能节省许多写操作。

文献[OnHLX09]提出了惰性更新 B+-树来高效地利用内存资源。在这种方法中，内存被分成两个部分：一部分用来缓存被访问的 B+-树节点的相应的页面（被称为页缓存），另一部分用来缓存更新请求（被称为惰性更新池）。

```
while a request R arriving do
    if R is an update request then
        if lazy-update pool is full then
            Use a commit policy to select a group of requests as victims;
            Commit victims to the B+-tree in bulk;
        Buffer R in the lazy-update pool;
        Use cancel-out policy to eliminate redundant requests;
    else
        /*R is aquery request*/
        Searching over lazy-update pool to get query result Q1;
        Apply traditional algorithm on B+-tree to get query result Q2;
        Merge Q1 and Q2 to get the final query result;
```

算法[lazy-update-B-tree]描述了惰性更新 B+-树方法。当一个更新请求到达时，并非立即提交到 B+-树，而是临时保存到惰性更新池中。在惰性更新池内，如果新到达的请求与池中已经存在的更新请求具有相同的键值但是具有不同的动作类型，就会采用一定的策略来消除冗余。此外，更新请求会被组织成不同的分组，每个更新相同叶子节点的请求构成一个分组。当惰性更新池无法容纳更多更新请求时，就会产生一个提交操作，选择其中一个组作为牺牲品，提交到 B+-树中，然后释放空间。对于查询而言，除了搜索惰性更新 B+-树以外，还需要搜索惰性更新池。

采用惰性更新池可以节省写操作数量。考虑一个更新顺序$\{U_1,U_2,U_3,U_4\}$，其中，U_1 和 U_3 将会插入到叶子节点 a 中，U_2 和 U_4 将会插入到叶子节点 b 中。在传统的方法下，两个叶子节点分别会更新两次。在惰性更新 B+-树中，这些更新请求将会被临时存储在惰性更新池中，这样做可以带来两个方面的收益：第一，被缓存的更新请求可以在后来某个时刻以批量的方式提交到 B+-树中，因此，可以在多个更新请求之间分摊定位被更新的叶子节点的读取代价；第二，更新顺序可以被重新排序成不同的分组，比如，把$\{U_1,U_3\}$分成一组，把$\{U_2,U_4\}$分成一组。然后，通过基于分组的提交，叶子节点 a 和 b 都只需要被更新一次。也就是说，可以节省一半的写操作。

对于惰性更新 B+-树而言，当一个新的更新请求到达并且惰性更新池已满时，应该采取一个提交策略，来选择一组更新请求来提交，从而为新的请求腾出空间。对于惰性更新 B+-树而言，一个高效的提交策略是很重要的，因为它对更新分组的效果的影响很大。比较理想的情况是，一个最优的提交策略应该总是选择满足下面条件的分组：这些分组没有更多后续的更新请求可以提交。这样做可以导致写操作数量的最小化。但是，这在实践中是不可能实现的，原因包括两个方面：第一，没有后续更新请求的分组并不是总是存在的；第二，后续的更新请求是无法预先知道的。因此，必须设计一个在线提交策略，使得它对更新分组的影响和写代价都达到最小化。可以考虑采用两种启发式在线提交策略：

（1）最大分组策略：对于一个新到达的更新请求而言，如果惰性更新池中已经存在一个它可以加入的分组，那么，我们就说发生了“命中”。为了提高命中率，就应该在惰性更新池中保存尽可能多的分组。因此，把最大分组驱逐出惰性更新池，相对于驱逐许多较小分组而言，可以获得更多的收益。而且，由于较大的分组包含了更多的更新请求，那么分摊到每个更新请求上面的代价就会比较小。

（2）基于代价的策略：最大分组策略是为了最大化命中率，基于代价的策略则目的在于使得驱逐分组引起的代价最小化。

7.2.3 自适应 B+-树索引

7.2.3.1 FlashDB 概述

文献[NathK07]提出了一个针对闪存的 DBMS——FlashDB，它是一个为传感器网络而优化的、自适应的数据库。FlashDB 使用了一个新的、自适应的 B+-树索引，可以动态地根据负载和底层的存储设备来调整它的树结构。在这种自适应的 B+-树中，包含两种类型的节点，即日志节点和磁盘节点，可以满足不同的性能要求。在不同类型的负载下，为了取得最好的性能，一个树节点可以在日志节点和磁盘节点之间进行切换，作者把这个切换问题建模成一个“双状态”任务系统，并提出了高效的在线算法。

如图[FlashDB]所示，FlashDB 包括两大功能组件：数据库管理系统（DBMS）和存储管理器（Storage Manager）。DBMS 组件负责实现数据库管理的各项功能，比如查询编译和索引；存储管理器组件负责实现基于闪存的高效存储功能，比如数据缓冲区和垃圾回收。索引管理器(Index Manager)是 DBMS 组件中的核心模块，它负责实现自适应的 B+-树。因此，作者在文中重点介绍了存储管理器和索引管理器。

存储管理器组件中各个模块的功能如下：

- 逻辑存储(Logical Storage)：负责实现逻辑扇区地址到物理闪存地址的映射。
- 节点转换表（Node Translation Table: NTT）：记录了自适应 B+-树中的逻辑节点的类型信息（日志节点还是磁盘节点）和物理地址信息。
- 缓冲区：专门为日志节点设置，当日志节点发生了更新时，首先被记录到缓冲区中。
- 垃圾回收：回收闪存中的无效页，执行擦除操作，得到一些可用的空闲块。

索引管理器组件中各个模块的功能如下：

- 设备配置(Device Configuration)：负责获得底层设备的信息，因为设备类型多种多样（比如闪存芯片、SD 卡、CF 卡、固态盘等），不同设备类型需要采用不同的算法配置。
- 节点切换算法(Node Switch Algorithm): 负责执行节点模式的切换，即在日志节点和磁盘节点这两种模式之间切换。
- 节点尺寸调整器（Node Size Tuner）：负责根据不同的设备来调整节点的大小，从而获得最佳性能。

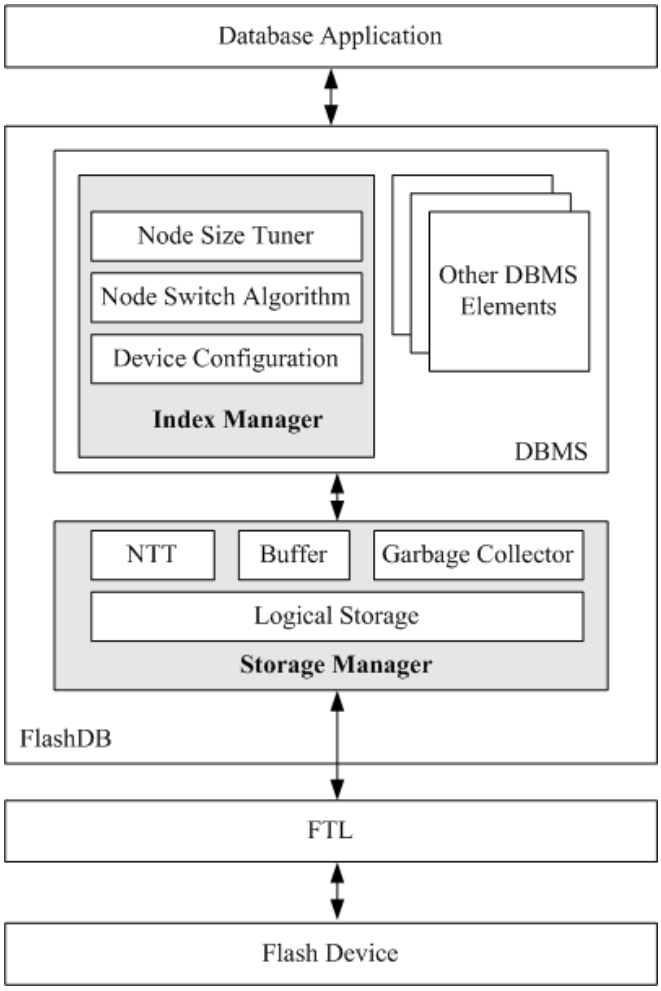


Fig.[FlashDB] The FlashDB Architecture

图[FlashDB] FlashDB 的体系架构

7.2.3.2 自适应 B+-树

FlashDB 的最重要创新点就是自适应 B+-树。文献[NathK07]分析了两种不同类型的 B+-树，即磁盘 B+-树和日志 B+-树：

（1）磁盘 B+-树：主要是面向磁盘设备，自然也就可以用在采用 FTL 的闪存设备上（可

以看成是一个虚拟的磁盘）。根据 B+-树的节点大小，每个节点会被保存到若干个连续的闪存页中。读取节点时，只要找到闪存中相应的页面读取即可。更新一个节点时，需要首先把相应的闪存页面读取到 RAM 中，进行更新，然后写回到闪存中。磁盘 B+-树的优点是：兼容性较好，可以不作任何修改直接应用到采用了 FTL 的闪存设备上。它的缺点也很明显，就是需要高昂的更新代价。

（2）日志 B+-树：基本思想是，把索引组织成事务日志的形式。当对一个树中的节点进行更新，首先把更新作为一个日志条目写入到缓冲区中，等缓冲区的日志可以写满一个闪存页时，再一次性把缓冲区中的日志条目全部写入到闪存中。很显然，日志 B+-树的更新代价要比磁盘 B+-树小很多，但是，读操作的代价较高。为了获得最新版本的数据，就必须对分散在不同闪存页中的日志条目进行扫描，然后和之前版本的旧数据进行合并更新得到最新版本的当前数据。

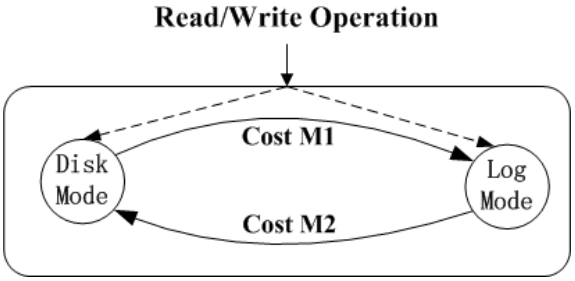
通过进一步分析，该文的作者发现，对于不同的负载类型和设备类型，同一棵 B+-树中的不同节点可以使用不同的 B+-树节点类型（磁盘 B+-树节点和日志 B+-树节点），从而获得最优性能。比如，对于写操作集中的负载以及那些写操作代价明显高于读操作代价的设备（比如 SD 卡）而言，就比较适合采用日志节点，因为，日志 B+-树节点可以减少闪存的写入次数，虽然这会增加闪存读操作的次数，但是，减少了总体读写代价；而对于读操作比较集中的节点，则适合设置为磁盘节点。如果使用 SD 卡或者写操作负载比较集中，采用日志 B+-树节点的效率要比采用磁盘 B+-树节点的效率高 80%。但是，如果负载类型或者设备类型发生了变化，结果就可能截然不同，前者效率甚至可能不如后者。比如，CF 卡的读写代价是相当的，如果采用 CF 卡，即使仍然是写负载比较集中，采用磁盘 B+-树节点的效率还是要比日志 B+-树节点的效率高 32%，因为，后者减少写入次数产生的收益可能小于该机制自身所增加的读取操作的代价。实验结果显示，当写操作次数大于读操作次数，并且每次写操作代价都是读操作代价的 5 倍以上时，采用日志 B+-树节点的效率更高，否则，应该选用磁盘 B+-树节点。

此外，负载类型和设备类型都可能会随着时间动态变化，因此，需要动态调整 B+-树中节点的类型。一般而言，数据库负载类型和设备类型都无法提前得知，而且，一个数据库上可能会同时接收到不同类型的工作负载，工作负载在不同时间点的模式也会发生变化，比如，某个时刻读操作比较集中的负载，到了另一个时刻可能就变成写操作集中的负载。因此，如果采用固定的 B+-树节点类型，就无法适应不断变化的负载类型和设备类型，会严重影响到数据库的总体性能。

实际上，这种 B+-树节点类型的动态调整机制，不仅对于固定负载类型和设备类型而言，可以获得较好的数据库性能，而且对于固定的负载和设备特性而言，同样可以获得较好的性能。比如，数据库负载是比较固定的写操作集中的负载，闪存的写代价远高于读代价。对于这种情形，按照前面的分析，很自然地应该采用日志 B+-树节点。但是，这里实际上还应该考虑另外一个因素，那就是每次写操作都需要通过一系列的查询（读操作）来确定插入点的位置，因此，这个过程会伴随着大量的读操作。由于查询操作都是从根节点开始的，因此，根节点附近的节点会被频繁地读取，这些节点就不适合采用日志 B+-树节点，而更应该采用磁盘 B+-树节点。综上所述，如果能够对每个树节点都进行独立的、动态的节点类型调整，就可以让数据库获得最优的整体性能。

自适应 B+-树节点类型切换问题，可以被建模成一个“双状态”任务系统[BlackS89]。如图[FlashDB-two-task-system]所示，一个节点可以有两种模式，即磁盘 B+-树节点模式和日志 B+-树节点模式。一个读操作 *R* 或者写操作 *W*，可以采用任何模式的节点来为它们提供服务，但是，不同节点提供服务的代价是不同的。一个节点可以从一种模式切换到另一种模式，但是，切换操作需要一定的代价。节点模式切换问题，就是使用一种在线算法，可以让节点动

态改变模式，并且使得节点切换和服务于读写操作的总代价达到最小化。这个问题就是一种经典的双状态任务系统的实例。



图[FlashDB-two-task-system] 在两种 B+-树节点模式之间的切换

节点模式动态调整算法，需要完成 B+-树节点在不同模式之间的切换：

- 当节点从日志模式切换到磁盘模式时，需要读取这个节点相关的日志内容，构造得到逻辑节点，然后作为磁盘模式写回闪存。
- 当节点从磁盘模式转换到日志模式时，就需要把逻辑节点改写成一系列的日志记录，存放在日志缓冲区中。

根据前面的分析，对于写操作集中的负载，比较适合采用日志节点，而对于读操作集中的负载，则比较适合采用磁盘节点。当负载类型发生变化时，为了获得最优性能，应该进行节点模式切换。但是，这里必须注意一个问题，节点转换会包含一定的开销，有时候这种开销甚至会大于转换操作带来的收益，对于这种“得不偿失”的转换操作应该尽量给予避免，因此，必须设计合理的节点模式切换算法。下面算法描述了节点模式切换的过程。

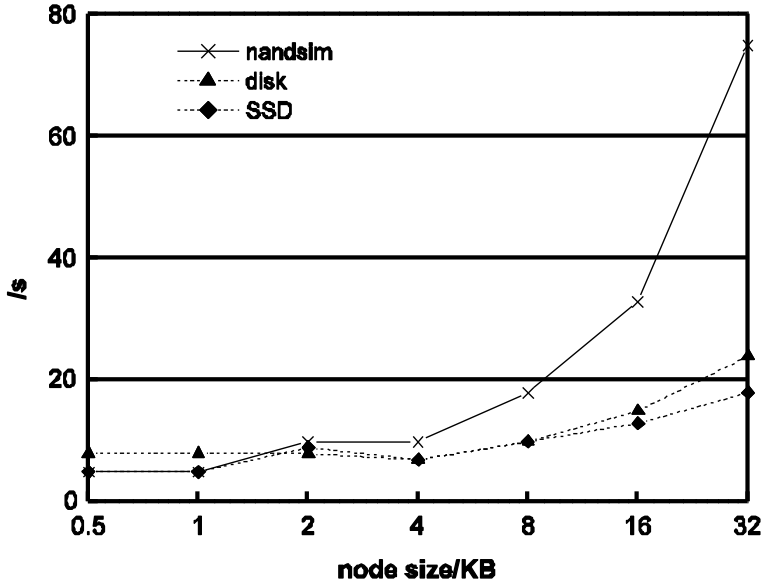
算法：节点模式切换
（下面的算法过程会针对每个 B+-树节点都执行一遍）
1. S←0； //当迁移到当前的模式时，就初始化 S
2. 对于每个读操作 O
假设 c_1 是当前模式下服务操作 O 的代价，c_2 是其他模式下服务操作 O 的代价;
S←S+(c_1-c_2);
3. 假设 M_1 和 M_2 是在两种模式之间进行转换的代价，如果 S≥M_1+M_2，那么就切换到其他模式;

7.2.3.3 B+-树节点大小的选择

7.2.3.3.1 B+-树节点大小对性能的影响

对于数据库而言，不仅 B+-树节点类型是影响其性能的一个因素，B+-树节点大小也会对性能产生影响。采用较大的 B+-树节点的优势是，可以减小树的高度，从而缩短了从根节点到叶子节点的路径长度，加快了查找过程。但是，较大的 B+-树节点的劣势也同样明显，它会增加单个节点存储的数据，增加了每次读取闪存的开销。因此，必须选择一个比较合适的 B+-树节点。在理论上，一个 B+-树节点可以使用任意数量的页，但是，实际上，通过一些实验测试还是可以确定一个比较合理的大小取值的。对于磁盘数据库而言，Gray[GrayG97]通过研究显示，采用 16KB 的节点大小可以获得较好的性能。对于闪存数据库而言，B+-树节点大小的最优值可以采用下面的方法加以确定。

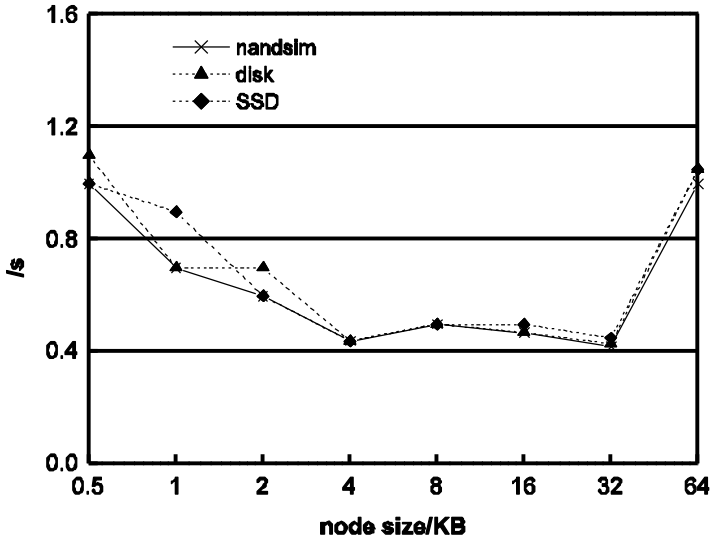
崔斌等人[CuiLC10]分别采用固态盘、磁盘和闪存模拟器等存储介质，对 B+-树节点大小的变化进行了性能影响测试。测试环境为 Fedora10，并使用闪存模拟器 nandsim 模拟了 512MB 的闪存，闪存页的大小为 2KB，块大小为 128KB，读、写和擦除操作的时间分别为 25us、200us 和 2ms。测试对象是一棵包含 20 万条数据的 B+-树，并采用 2 万条数据对这棵 B+-树进行顺序或随机的插入和查询操作。



图[cuibin-test-01] B+-树节点大小对插入操作性能的影响

测试之前需要关闭缓存，从而确保所有的数据操作都是针对外存进行的。图[cuibin-test-01]给出了在磁盘、固态盘和闪存模拟器上采用不同的 B+-树节点大小时对插入操作性能的影响。从图中可以看出，磁盘和固态盘上面的性能变化趋势比较一致，插入操作所需要的时间，随着节点大小的增加，表现出先减少后增加的特点，当节点大小为 4KB 时，插入时间达到最小值。这个测试结果比较符合前面的理论分析，因为，采用较大的 B+-树节点可以减小树的高度，缩短路径长度，从而加快了插入操作过程。但是，节点增大，也意味着它会增加单个节点存储的数据，增加了每次更新操作的开销，因此，节点大小超过一定程度后，节点越大，插入时间反而会不断增加。图[cuibin-test-01]也表明了另外一个重要的结果，即在闪存模拟器上面，插入时间会随着节点大小的增加表现出单调增加的趋势。因此，对于闪存模拟器而言，最优的树节点大小要比磁盘小得多。

图[cuibin-test-02]给出了 B+-树节点大小对查询操作性能的影响。在实验中使用 2 万个顺序查询对 B+-树进行性能测试，从图中可以看出，在固态盘、磁盘和闪存模拟器上的测试结果都表现出了大致相同的规律。采用较大的 B+-树节点可以减小树的高度，从而缩短了从根节点到叶子节点的路径长度，加快了查找过程，因此，三者的查找时间都随着节点大小的增加而不断减小。但是，查找时间减小到一定程度以后，又开始表现出上升的趋势，也就是说，当节点大小在 16KB 附近时，查询时间达到了最小值，此后又开始不断增加。这是因为，较大的 B+-树节点，会增加单个节点存储的数据，增加了每次读取节点中的数据的开销。



图[cuibin-test-02] B+-树节点大小对查询操作性能的影响

7.2.3.3.2 FlashDB 中确定 B+-树节点大小的方法

FlashDB 中 B+-树节点的大小是通过以下方法确定的。假设一个 B+-树索引有 N 个数据单元，一个 B+-树节点大小为 NodeSize，每个节点可以包含 EntiresPerNode 个索引条目，那么，这棵树的高度就是：

Height=$\log_2$ (N) / $\log_2$ (EntriesPerNode)

首先，来定义一个节点的“有用性”，它被定义为这个节点可以把一个查询指引到距离目标叶子节点多近的距离，计算公式如下：

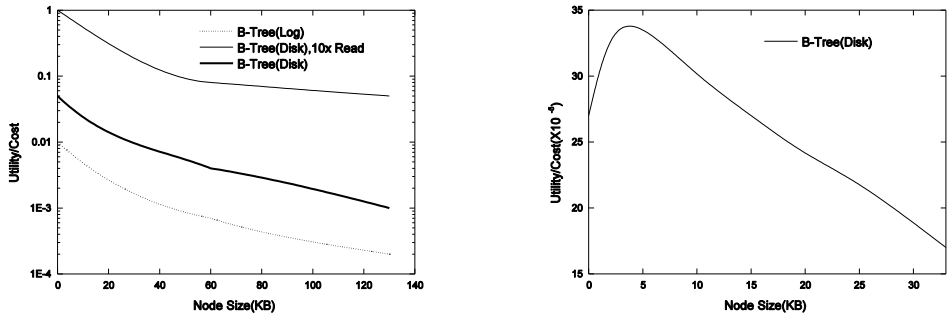
NodeUtility = $\log_2$(EntriesPerNode)

例如，如果每个索引条目（index entry）是 16 个字节，那么，对于一个只有七成满的索引页而言，就会包含 44 个索引条目。这样一个节点的有用性就是 5.5（即 $\log_2$44），只有一个 48KB 节点的有用性的一半。因此，从直观上看，节点越大，树的高度越小，到达底层叶子节点所需要访问的节点数目就越少。

然后，再来考虑一下访问一个节点的代价。假设所有的 B+-树操作都是写操作，并且所有的节点都是磁盘 B+-树节点。一个写操作需要读取从根节点到叶子节点的路径上面的所有节点，然后至少需要写入叶子节点，当叶子节点的更新传播到其他节点时，则还需要写入更多的其他非叶子节点。因此，对于一个单个节点而言，分摊后的访问代价就是：

$$Cost = \frac{Height}{Height+1}\mathrm{Re}adCost + \frac{1}{Height+1}WriteCost$$

最后，就可以计算得到一个节点的“有用性/代价”的值，用来作为性能评测指标。图[FlashDB-utility-cost]显示了针对三星闪存产品进行的性能测试，这里的 B+-树具有 30000 个 16 字节的索引条目，每个索引节点是 7 成满。从图中可以看出，对于闪存芯片而言，节点越小，性能越好（即有用性/代价的值越大）；对于 CF 卡而言，当节点大小为 4KB 时，可以获得最好的性能。



(a) 有用性/代价的比值（闪存芯片） （b）有用性/代价的比值（CF 卡）

图[FlashDB-utility-cost] 采用不同的 B+-树节点大小时，有用性/代价的值

因此，对于不同的设备而言，需要采用不同的 B+-树节点大小，以期获得最好的性能。比较好的做法是，让 B+-树节点大小随着设备特性的不同而自动调整其大小。

7.3 FD-树

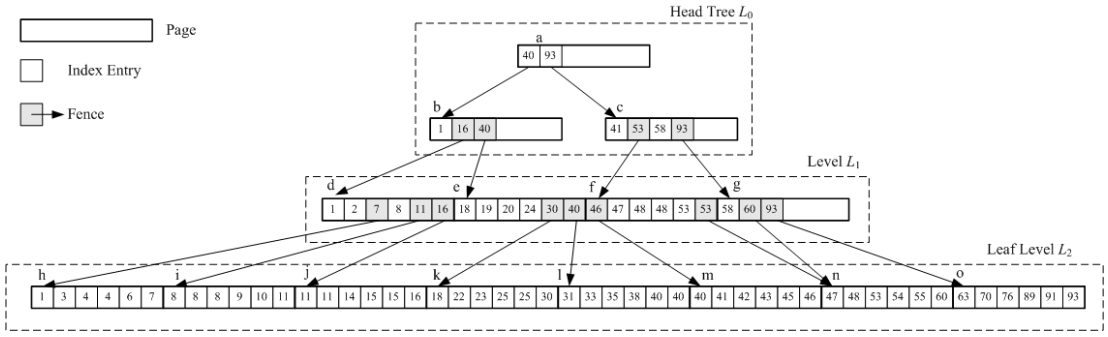
FD-树[LiHLY09]的设计目标是最小化小数据量的随机写操作的数量，同时维持一个较高的搜索效率，它采用了对数方法[Bentley79]和分散层叠技术（fractional cascading）[ChazelleG86]来实现高效地搜索和更新。一棵 FD-树包含了多个级别，被称为 $L_0,L_1,\ldots,L_{l-1}$，其中，l 表示级别的数量。最顶层 L_0 是一个小的 B+-树，称为“头树”。每个其他的级别 L_i，是一个存储在连续的页中的有序段。为了加快在多个有序段中的搜索速度，FD-树采用了分散层叠技术。图[FD-tree]显示了一棵 FD-树的结构，这棵 FD-树有三个层次：头树和两个有序段。头树是一棵两层的 B+-树。采用分散层叠技术后，头树的叶子节点具有指向有序段 L_1 的指针。每个非叶子节点，也都有指针指向相邻的下一层的有序段。

FD-树的每层都有一定的容量（能够容纳的元素的个数），FD-树根据对数方法在不同的层采用阶梯式的容量，即$||L_{i+1}||=k.||L_i||$（$0\le i\le l-2$），其中，$||L_i||$表示第 L_i 层的容量，k 是 L_i 和 L_{i+1} 之间的对数尺寸比。因此，存在如下关系：$||L_i||=k^i*||L_0||$。更新操作最先针对头树进行，然后当每层的容量达到上限时，就会被逐渐批量迁移到下面层的有序段中。头树的最大尺寸$||L_0||$远远小于可用的内存容量，因此，可以放入内存。如果设置$||L_0||$为闪存的擦除块的大小，就可以使得头树可以被保存到一个擦除块中。因为头树可以放置到内存中，许多针对闪存盘的随机写操作，就可以通过合并操作转换成顺序写操作，从而减少了小数据量的随机写操作的数量。

FD-树设计了索引项(Index Entry)和栅栏(fence)：

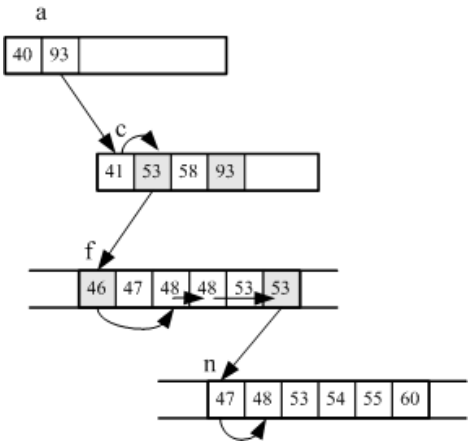
（1）索引项：一个索引项包含三个字段，即索引键 *key*、记录号和类型，“类型”字段用来确定它在“对数删除”FD-树中的角色。

（2）栅栏：一个栅栏包含了指向更低层次的有序段中的页的指针，具有三个字段，即键值、类型和 pid，pid 是紧邻的更低层次的页的 ID，用来指导进行搜索。采用栅栏以后，一个 FD-树的搜索操作，就可以首先在一个更小的树上执行，然后，在栅栏的引导下，向下一层一层地进行搜索。



图[FD-tree] FD-树的实例

在一棵 FD-树上进行搜索，需要从顶向下搜索每个层。首先在头树上进行查找，这和在传统的 B+-树中查找类似。接着，沿着栅栏的 *pid* 在每个级别上进行查找。在级别 L_i 的一个页的内部，会采用二分法搜索来寻找到第一个匹配的索引项。然后，可以扫描有序段，从第一个匹配的索引项开始，找到所有匹配的索引项。然后继续扫描，直到发现一个栅栏的键值等于或者大于搜索的键值。所有匹配的键值都会被加入到结果中。图[FD-tree-b]显示了在图[FD-tree]的 FD-树中进行查找 *key*=48 的记录的过程。在每个层次都会搜索整个页，一旦碰到栅栏，就去搜索在下一个层次的有序段中的页。



图[FD-tree-b] 在 FD-树中进行查找

7.4 LA-树

基于日志的 B-树索引[WuCK03]和 FlashDB[NathK07]中的自适应 B+-树索引等面向闪存的索引结构的设计思想，都是采用页级别的写优化策略。也就是说，这些方法并没有执行一个代价高昂的、针对原始闪存页的就地更新操作，而是把数据页发生变化的部分，以日志的形式存储到闪存中一个单独的位置。这种方式有效地降低了页写操作的代价，但是，与此同时也大大增加了读操作的代价，因为，当一个页需要被读回到内存中时，需要读取原始页和日志页，然后重构得到最新版本的数据页。因此，Agrawal 等人[AgrawalGSDS09]认为这种优化方式并不是非常适合面向闪存的索引结构，他们提出了一种新的面向闪存的索引结构——LA-树（Lazy Adaptive tree），它采用了完全不同于其他方法的思路，即惰性更新技术，它借鉴了批量更新索引树缓冲区的相关研究工作[ArgeH02]。

7.4.1 LA-树的设计思想

LA-树的主要思想是，使用了基于级联缓冲区的惰性更新技术，来减少更新一个面向闪存的树结构索引所产生的高昂代价，把树结构的高昂更新代价在多个更新操作之间进行分摊，使得单个更新操作具有较低的代价，同时还采用了一种自适应的在线算法，可以根据工作负载动态调整缓冲区大小，从而可以同时为更新和查询都获得较好的性能。

（1）级联缓冲区

为索引树的多个层次上的每个节点都附加一个缓冲区，每个缓冲区包含了将要在这个节点及其后代节点上执行的操作。选择一个合适的时机，缓冲区中的所有元素都会以批量的方式，被推到下一层的缓冲区中。这个操作可能会继续向下一层缓冲区传播，直到被推到叶子节点，因此，取名为“级联缓冲区”。由于 LA-树缓冲区中缓存了多个更新操作，因此，多个更新操作之间共享从根节点到叶子节点的访问代价，也就是说，每个更新都只需要分担一部分代价，从而使得每个更新都具有较低的代价。

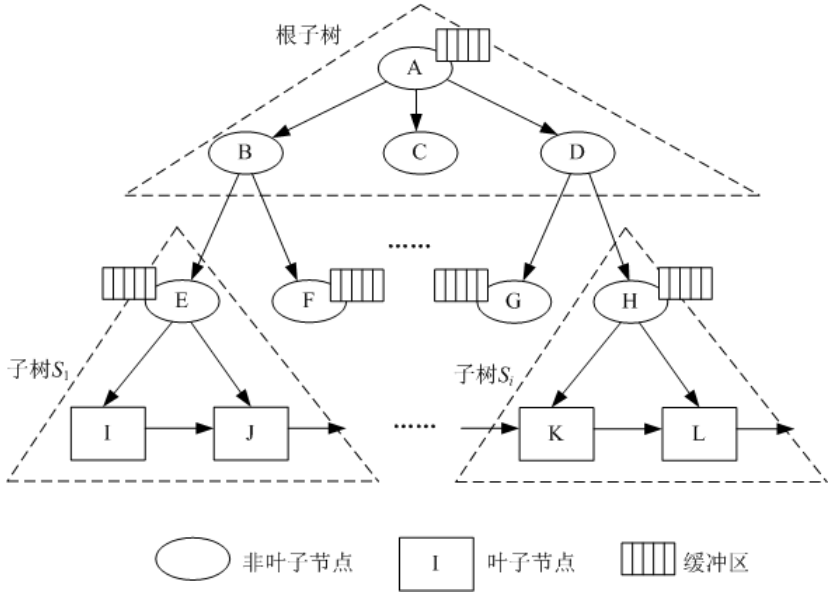
（2）在线自适应算法

惰性更新技术需要设计级联缓冲区，但是，采用固定大小的缓冲区设置，会引起更新和查询性能之间的矛盾，即提高更新操作性能可能意味着查询性能的降低。这里考虑一个具备固定尺寸 *B* 的根缓冲区。当缓冲区中已经累积了 *B* 个条目时，它就会被清空，把条目分发给下一层的缓冲区。清空代价包括两个部分：(a)对缓冲区和子树的一次扫描，和(b)把缓冲区元素写入到下一级的缓冲区中。因此，在所有更新操作之间分摊清空代价（尤其是分摊的子树读取代价），是和 B 成反比的。相反，在一个查询操作中，必须扫描一个缓冲区中的所有条目，引起的代价和 B 成正比。因此，缓冲区的大小设置，在更新操作和查找操作之间表现出了矛盾性。针对这个问题，一些其他方法都是以较高的查询开销为代价，换取较低的更新代价。相反，LA-树使用一个在线算法动态地自适应任意的工作负载，可以根据工作负载的特性控制缓冲区的大小，从而可以为更新和查询同时提供较高的性能。

7.4.2 LA-树索引结构

和 B+树类似，LA 树是一棵出度为 *F* 的平衡树。B+树不同的地方在于，在 LA-树中，每个第 K 层上的节点都附加了一个常驻闪存的缓冲区，这个缓冲区用来批量地执行将要在这个节点及其后代节点上执行的更新操作（插入或删除）。在一棵 LA-树中，一棵“子树”是指树中的一个片段，它从附加了缓冲区的节点开始，到下一个附加了缓冲区的节点所在的层之上结束，因此，一个子树有 *K* 层。

图[Lazy-adaptive-tree]显示了当 *K*=2 时的一棵 LA-树的实例。A 是根节点，A、B、C、D、E、F、G、H 是非叶子节点，I、J、K、L 是叶子节点，由节点 A、B、C 和 D 这四个节点构成一个一棵根子树，由节点 E、I 和 J 构成一棵子树 S_1，由节点 H、K 和 L 构成一棵子树 S_i。可以看出，每棵子树都包含两层。这棵 LA-树一共包含了 4 层，第 1 层中的根节点 A，以及第 3 层中的所有节点（比如 E、F、G 和 H），都附带了一个缓冲区。



图[Lazy-adaptive-tree] LA-树的一个实例

（1）惰性插入和删除

LA-树采用了惰性更新技术，也就是说，与新的插入和删除相对应的缓冲区条目，只会简单地被插入到根节点的缓冲区中。然后，在线自适应算法会被调用，用来判断是否需要清空缓冲区。如果决定需要清空缓冲区，就会唤醒缓冲区清空进程，以批量的方式把缓冲区中的所有条目，都推动到处于 LA-树中下一个层级的缓冲区中。这个过程会产生级联效果，直到这些缓冲区条目最终到达叶子节点。

（2）即时查找

当一棵 LA-树接收到一个查找请求时，它会从上自下查找索引树。LA-树和 B+树的一个主要区别是，如果在从根节点到叶子节点的路径上的某个节点被附加了缓冲区，LA-树会同时扫描缓冲区，寻找是否存在还没有被传播到叶子节点的更新操作。当执行一个缓冲区扫描时，在线自适应算法会综合考虑历史的工作负载，从而判断清空缓冲区的操作是否可以改善性能。如果缓冲区清空操作被判定为可以带来收益，缓冲区中的所有条目都会被推动到下一个层级的缓冲区中；否则，会扫描缓冲区，来寻找和搜索谓词匹配的结果。这个过程会一致持续，直到到达叶子节点。

（3）关于清空缓冲区的动态操作

在线自适应算法是 LA-树的核心智能模块，它记录了针对每个缓冲区的更新和查找操作序列，并且通过决定何时清空缓冲区来控制缓冲区的大小，从而使得缓冲区的大小始终可以保证获得较好的更新和查找操作性能。更新请求不会引发一个缓冲区清空操作，除非缓冲区大小超过了预先定义的上限值。对于查找操作而言，在线自适应算法会同时记录扫描缓冲区的代价和清空缓冲区的预估代价，然后，它会衡量下面事项：（1）在这个时间点清空缓冲区的代价；（2）由于清空操作导致的后续的查询操作可能获得的收益。经过衡量以后，如果收益超过清空代价，那么就会决定清空缓冲区。

（4）缓冲区清空操作

在决定清空缓冲区时，在线自适应算法会唤醒缓冲区清空函数。在一个中间子树中的缓冲区清空操作过程如下：在搜索完这棵子树以后，对缓冲区条目进行排序，以排序后的顺序把缓冲区中的所有条目都发送到下一个级别的缓冲区中。在叶子子树中的缓冲区的清空操作，也是从排序缓冲区条目开始的，然后，缓冲区的条目会以排序后的顺序被发送到叶子节点。叶子节点会从左到右被一次性处理，对于每个叶子节点 N，接收到的缓冲区条目会被合并到 N 中，即执行这些条目中包含的插入和删除操作。如果 N 溢出了，它就会被分裂成两个或多个节点。这种分裂可能会传播到它的祖先节点；附加在这些节点上的非空的缓冲区也会相应地发生分裂。由于在实际工作负载中，插入和删除操作是混合的，因此，N 很少会发生溢出。但是，当确实发生溢出时，N 首先采用的调整方式是，从隔壁节点借用条目，然后，如果有需要，它会和 B+树一样，和邻居节点发生合并。在很少发生的合并操作中，除了被合并的节点的缓冲区也会被相应地合并之外，父节点和祖先节点的调整和 B+树一样。

7.5 本章小结

本章内容首先指出了针对闪存特性设计相应的索引结构的必要性；然后，介绍了基于日志的 B-树索引，给出了一种基于日志的 B-树索引方法的体系架构——BFTL，该方法可以被封装成一个单独的、可插拔的模块，并且该模块处于 FTL 层之上；接下来，介绍了面向闪存的 B+-树索引，包括惰性更新 B+-树方法和自适应 B+-树方法；然后，又介绍了 FD-树，它的目标在于最小化小数据量的随机写操作的数量，同时维持一个较高的搜索效率，它采用了对数方法和分散层叠技术来实现高效地搜索和更新；最后，介绍了 FA-树，它使用基于级联缓冲区的惰性更新技术，来减少更新一个面向闪存的树结构索引所产生的高昂代价，把树结构的高昂更新代价在多个更新操作之间进行分摊，使得单个更新操作具有较低的代价，同时还采用了一种自适应的在线算法，可以根据工作负载动态调整缓冲区大小，从而可以同时

为更新和查询都获得较好的性能。

7.6 习题

1、 说明针对闪存特性设计相应的索引结构的必要性。

2、 在闪存上直接建立 B-树索引，面临的一个棘手的问题就是级联更新，请说明什么是级联更新。

3、 阐述 BFTL 方法中的节点转换表的作用。

4、 分析磁盘 B+-树和日志 B+-树的各自不同特点。

5、 阐述惰性更新 B+-树方法中采用的两种在线提交策略。

第 8 章 闪存数据库查询处理

虽然固态盘可以快速改善那些侧重于随机读操作的应用的性能，但是，对于那些长时间运行大量分析查询的数据库应用而言，固态盘却无法立即改善其整体性能。数据库查询处理引擎在设计时主要考虑的是磁盘的随机和顺序 I/O 的速度失配问题，传统的各种算法都强调针对磁盘数据的顺序访问。而固态盘的随机读和顺序读操作几乎具有一致的高性能，因此，数据库的一些数据结构和算法，应该充分利用闪存的特性提高数据库性能。

在关系数据库中，查询操作往往涉及大量的连接操作，而连接操作需要大量的 IO 开销和计算资源，因此，许多研究都关注如何在闪存环境下对连接操作进行性能优化，从而可以充分利用闪存的特性，提高数据库查询的性能。比较典型的方法是，采用 PAX 页布局模型和连接索引，实现高效的闪存数据库连接操作。

本章首先介绍 PAX 页布局模型，然后介绍连接索引的概念，接下来介绍面向闪存数据库的、基于 PAX 和连接索引的 RARE 连接算法以及 FlashJoin 算法，最后，介绍采用类似连接索引理念的 DigestJoin 方法。

8.1 PAX页布局模型

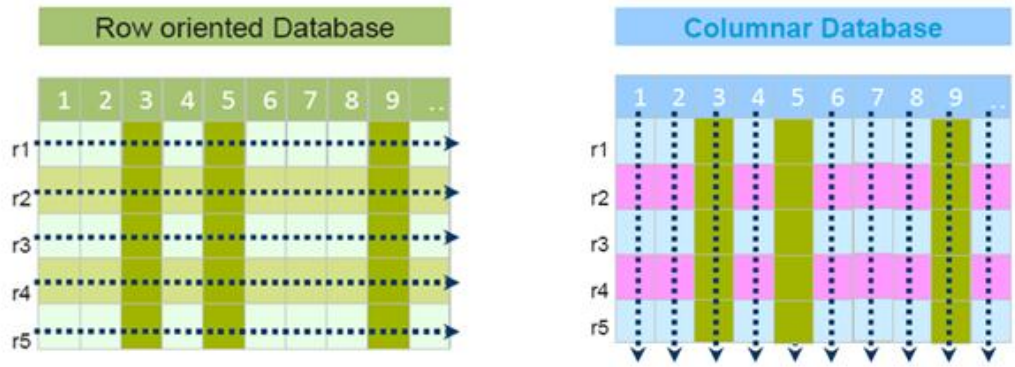
有一些研究通过修改传统的查询处理技术，来充分利用闪存的快速随机读能力，从而达到提升数据库性能的目的。比如，文献[ShahHWG08][TsirogiannisHSWG09]的研究目标就是调查分析一些可以改进复杂数据分析查询性能的查询处理技巧，这类复杂查询主要出现在较少发生更新的商务智能和数据仓库应用中。这类研究主要采用 PAX 页布局模型和连接索引，来充分利用闪存特性，实现高效的数据库连接操作。

本节首先介绍与 PAX 模型紧密相关的 NSM、DSM 模型，然后介绍 PAX 模型。

8.1.1 NSM 和 DSM 模型

下面将首先给出关于 NSM 和 DSM 模型的简要介绍，然后，将分别介绍 NSM 模型存储原理和 DSM 模型存储原理。

8.1.1.1 NSM 和 DSM 模型概述



(a) 行式数据库　　(a) 列式数据库

图[row-column-database] 行式数据库和列式数据库示意图

传统上，数据库系统使用 NSM(N-ary Storage Model)存储模型[RamakrishnanG00]，也就是采用一个基于页的存储布局，其中，元组或行会被连续地存储在页中(如图[row-column-database](a)所示)。在读取数据时，需要顺序扫描每个元组，然后，从中筛选出查询所需要的属性。如果每个元组只有少量属性的值对于查询是有用的，那么 NSM 就会浪费许多磁盘空间和内存带宽。采用 NSM 模型的数据库，通常被称为“行式数据库”，主要适合于小批量的数据处理，常用于联机事务型数据处理。

DSM(Decomposition Storage Model)存储模型（DSM）[CopelandK85]是在 1985 年提出来的，目的是为了最小化无用的 I/O。DSM 采用了不同于 NSM 的思路，DSM 会对关系进行垂直分解，并为每个属性分配一个子关系，因此，一个具有 *n* 个属性的关系，会被分解成 *n* 个子关系，每个子关系只有当其相应的属性被请求时才会被访问。也就是说，它是以关系数据库中的属性或列为单位进行存储，关系的元组中的同一属性值被存储在一起，而一个元组中不同属性值则分别存放于不同的页中(如图[row-column-database](b)所示)。采用 DSM 的数据库通常被称为“列式数据库”，主要适合于批量数据处理和即席查询，它的优点是：（1）可以降低 I/O 开销，支持大量并发用户查询，其数据处理速度比传统方法快 100 倍，因为仅需要处理可以回答这些查询的列，而不是分类整理与特定查询无关的数据行；（2）具有较高的数据压缩比，较传统的行式数据库更加有效，甚至能达到五倍的效果。列式数据库主要用于数据挖掘、决策支持和地理信息系统等查询密集型系统中，因为一次查询就要得出结果，而不能每次都要遍历所有的数据库。所以，列式数据库大多都是应用在人口统计调查、医疗分析等行业中，这种行业需要处理大量的数据统计分析，假如采用行式数据库，势必导致消耗的时间会无限放大。

DSM 的缺陷是，需要存储元组 ID 的额外存储开销以及执行连接操作时需要昂贵的元组重构代价。因此，在过去很多年里，主流商业数据库大都采用了 NSM 而不是 DSM。但是，随着市场需求的变化，尤其是企业对分析型应用（比如数据仓库）的大量需求，从最近几年开始，DSM 模型重新受到青睐，并且出现了一些采用 DSM 的商业产品和学术研究原型系统，比如 Sybase IQ、ParAccel、Sand/DNA Analytics、Vertica、InfiniDB、INFOBright、MonetDB 和 LucidDB。类似 SybaseIQ 和 Vertica 这些商业化的列式数据库，已经可以很好地满足数据仓库等分析型应用的需求，并且可以获得较高的性能。

下面给出一个关于行式存储和列式存储的非常简单的实例。表[row-column-database]是一个数据库的关系 *R*，包含了四个属性（EmpId、Name、Age 和 Salary）和三个元组。数据库必须把这个二维表存储在一系列一维的“字节”中，由操作系统写到内存或磁盘中。行式数据库把一行中的数据值串在一起存储起来，然后再存储下一行的数据，依此类推，存储结果如下：

1,Wang,35,4000;2,Lin,37,5000;3,Xue,42,4400;

列式数据库把一列中的数据值串在一起存储起来，然后再存储下一列的数据，依此类推，存储结果如下：

1,2,3;Wang,Lin,Xue;35,37,42;4000,5000,4400;

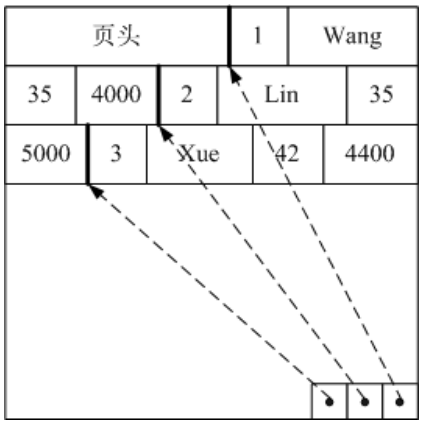
表[row-column-database] 一个数据库关系的实例

EmpId	Name	Age	Salary
1	Wang	35	4000
2	Lin	37	5000
3	Xue	42	4400

8.1.1.2 NSM 模型存储原理

传统上，数据库元组都是遵循 NSM 模型[RamakrishnanG00]被顺序地存放到磁盘上。NSM 会在磁盘页上连续存储元组。图[NSM]给出了表[row-column-database]中的关系采用 NSM 模型存储的效果。每个页包含一个页头、元组存储区域和页尾。每个新的元组，通常会被插入到从页头开始的第一个可用的自由空间里。页尾包括许多个槽（slot），每个槽都存储了一个指向元组存储区域的某个元组起始位置的指针。当在一个页内新增加一个元组时，元组可能是可变的长度，因此，一个指向新元组的起始位置的指针（偏移量），会被存储到

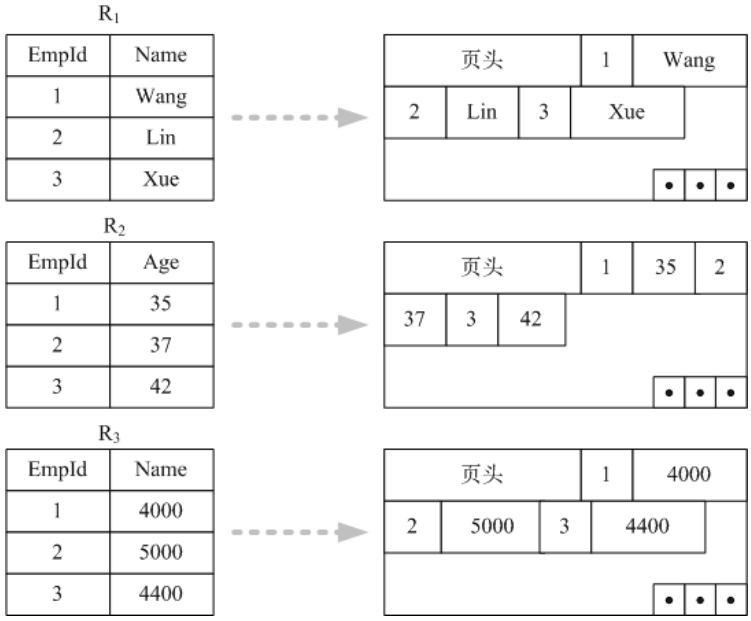
从页尾开始的下一个可用的槽里面。从页尾开始的第 *n* 个槽当中的指针，就指向该页中的第 *n* 个元组。



图[NSM] 采用 NSM 模型存储的一个实例

8.1.1.3 DSM 模型存储原理

图[DSM]给出了表[row-column-database]中的关系采用 DSM 模型存储的效果。DSM[CopelandK85]把具有 3 个属性（除了 EmpId）的关系 *R* 分解成 3 个子关系 R_1、R_2 和 R_3，每个子关系都包含两个属性。子关系会被存储到一个常规的页中，从而允许对每个属性进行独立的扫描。



图[DSM] 采用 DSM 模型存储的一个实例

8.1.2 PAX 模型

8.1.2.1 PAX 模型存储原理

PAX（Partition Attributes Across）[AilamakiDH02]是一种混合的方法，在本质上是一个在 NSM 页内部采用类似于 DSM 的组织方式。PAX 背后的机理是：像 NSM 一样，把每个元组的属性值都保存在同一个页中，同时使用一个缓冲区友好的算法来把数据放置到页中。PAX 会在每个页内部对元组进行垂直分区，把所有属于同一个属性的值都一起存储在“迷你页”（minipage）中。也就是说，PAX 把第一个属性的所有值都存储到第一个迷你页中，第二个属性的所有值都存储到第二个迷你页中，依此类推。在每个页的开始处，有一个页头，它包含了到每个迷你页起始处的偏移量。每个迷你页的结构如下：

（1）固定长度的属性值被存储到“F-迷你页”中。在每个 F-迷你页的尾部，有一个存在

位向量，每条记录对应一个条目（entry），对于空属性，其对应的存在位会记录一个空值。

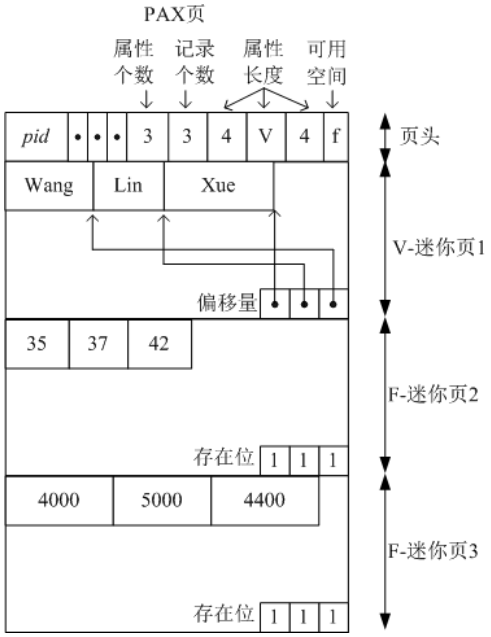
（2）变长的属性值会被存储在“V-迷你页”中，V-迷你页尾部包含许多个槽（slot），每个槽都包含一个指针，指向每个属性值的末尾，空值就用空指针表示。

图[PAX]给出了用 PAX 模型存储表[row-column-database]中的记录的效果。在页头部分，包含了页号 *pid*、属性个数、记录个数、属性长度和可用空间等信息。为了存储这个具有 3 个属性（除了 *EmpId* 以外的 3 个属性：*Name*、*Age*、*Salary*）的关系，PAX 把一个页分解成 3 个“迷你页”（minipage），即 V-迷你页 1、F-迷你页 2 和 F-迷你页 3，然后，它把第一个属性 *Name* 的所有值都存储到第一个迷你页中，第二个属性 *Age* 的所有值都存储到第二个迷你页中，依此类推。在每个 PAX 页的开始处都有一个页头，它包含了到每个迷你页起始处的偏移量。因此，当使用 PAX 的时候，每个元组会驻留在相同的页上，就像使用 NSM 时一样。但是，所有的 *Name* 属性值、所有的 *Age* 属性值和所有的 *Salary* 属性值，都会被分组存储在各自的迷你页内。通过这种方式，PAX 增加了元组之间的空间局部性，因为，它把属于不同元组的相同属性的值都保存在同一个迷你页中，同时还不会影响元组内部的空间局部性。此外，尽管 PAX 采用了页内垂直分区，但是，它只会带来很小的元组重构代价，因为，它不需要执行连接操作来重构一个元组。

每个新分配的 PAX 页，都包含了一个页头和许多迷你页，迷你页的数量和关系的属性数量相等。页头中存储的信息包括属性的数量、属性的长度（对于固定长度的属性）以及到迷你页起始位置的偏移量、页中当前已经存储的元组的数量和可用的总空间。对于属于同一个页的元组而言，可以被顺序地访问，或者也可以被随机地访问。

在存储一个关系时，PAX 的空间开销和 NSM 是一样的。NSM 会连续存储每个元组的属性值，因此，对于 NSM 而言，每个元组都需要存储一个偏移量，并且需要为每个元组中的可变长度的属性提供一个额外的偏移量。而对于 PAX 而言，需要为每个可变长度的属性值存储一个偏移量，另外还需要提供一个关于每个迷你页的偏移量。因此，无论一个关系使用 PAX 还是 NSM，都需要占用相同数量的页。

PAX 和其他设计是独立的，因为它只影响单个页上的数据布局方式，也就是说，在同一个数据库中，可以让一个关系使用 PAX 模型进行存储，而另一个关系则使用 NSM 模型进行存储。



图[PAX] 采用 PAX 模型存储的一个实例

8.1.2.2 PAX、NSM 和 DSM 的特性比较

一般而言，可以从两个方面比较 NSM、DSM 和 PAX 的特性，即元组之间的空间局部性和元组重构代价。首先，当元组之间具有较好的空间局部性时，可以最小化与 CPU 缓存(cache)相关的数据读取延迟，尤其是在循环扫描多个元组的一个属性子集的值时，空间局部性变得更加重要。其次，当元组重构代价较小时，可以使得与检索同一个元组的多个属性值相关的延迟达到最小化。

表[NSM-DSM-PAX]总结了 NSM、DSM 和 PAX 三种模型在元组之间的空间局部性和重构代价方面的特性。DSM 提供了较好的元组间的空间局部性，因为它可以对属性值进行连续存储，而 NSM 则无法提供这种空间局部性，因为 NSM 以元组为单位进行连续存储，属于不同元组的同一个属性上的值会被分散存储到不相邻的位置。当顺序扫描一个属性值时，由于 DSM 可以提供更高的空间局部性，因此，DSM 可以表现出很高的 I/O 性能和缓存性能。当查询访问每个关系中不到 10%的属性时，DSM 性能要好于 NSM。但是，当执行一个涉及多个属性的查询时，数据库系统必须连接相关的子关系，从而重构元组。连接子关系所消耗的时间，会随着结果中包含的属性的数量的增加而增加。此外，DSM 引入了明显的空间开销，因为，每个元组 ID 都需要被复制到任何一个子关系中。总体而言，DSM 虽然具有较好的空间局部性，但是，元组的重构代价较高，抵消了由于空间局部性带来的性能收益，因此，当前主流的商业数据库系统都使用 NSM 而不是 DSM。

但是，NSM 在缓存行为方面的表现不尽人意。NSM 会从每个磁盘页的起始位置开始连续存储元组，并且使用页末尾的“偏移量表”来定位每个元组的起始位置[RamakrishnanG00]。但是，绝大多数查询操作，只会访问每个元组的一少部分属性（或字段），这将导致无法获得最好的缓存性能。例如，对一个特定属性的顺序扫描时，将会发生缓存脱靶，每次访问一个属性值时，都需要访问一次内存。这个过程会把无用的数据加载到 CPU 缓存中，直接带来以下负面结果：（1）浪费了带宽；（2）无用的数据污染了缓存；（3）可能会强制替换未来可能用得到的数据，引发更多的延迟。

PAX 是对 NSM 和 DSM 两个模型的巧妙结合，既能够提供空间局部性，也具备较小的重构代价。而且，在现有的 DBMS 上实现 PAX，只需要修改页级别的数据操作代码。

表[NSM-DSM-PAX] NSM、DSM 和 PAX 之间的特性比较

特性	NSM	DSM	PAX
元组之间的空间局部性	否	是	是
较低的元组重构代价	是	否	是

PAX 相对于 DSM 的优势：和 PAX 模型相比，DSM 模型存在两个明显的局限性。首先，传统的关系数据库引擎的许多组件，比如存储层、IO 子系统、缓冲池、恢复系统、索引和运算符等，都被设计成在固定大小的页上执行操作。因此，DSM 模型会涉及到所有这些组件，需要对这些组件进行重新设计。但是，PAX 模型只需要重新实现存储层（负责从页中读取数据），因为，在 PAX 模型中，只有页的组织方式和传统不一样，而页的内容则没有发生变化。其次，采用 DSM 模型的列式存储，可能需要多个 IO 来更新一个行中的多个列。而采用 PAX 模型的时候，只需要一个 IO 就可以完成对一个行中的多个列的更新，因为，一个行中的所有的列都存储在相同的页内。

PAX 相对于 NSM 的优势：采用 NSM 模型时，对于一个只需要一部分列的查询而言，必须扫描所有的列。PAX 基于列的布局，为不同列的属性值之间提供了物理上的隔离，把同列的属性值都放置在相同的迷你页内，就可以允许一个操作只需要访问查询所涉及的那些属性，主要是通过在迷你页中“寻址”来实现。当从一个迷你页跳到下一个迷你页的寻址时间，要小于读取无用的迷你页的时间的时候，那么，执行随机读就会获得更好的性能。另外，虽

然 PAX 的磁盘访问模式是无法和 NSM 进行区分的，但是，它确实改进了内存带宽需求。对于一个常规的扫描器而言，通常会从磁盘中访问完整的页，因此，PAX 对磁盘带宽不会产生影响。但是，一旦一个页已经进入了内存，PAX 就允许 CPU 只去访问查询所涉及的那些迷你页，这就减小了内存带宽需求[AilamakiDH02]。

8.1.2.3 PAX 模型在闪存上可以发挥更好的性能

企业磁盘驱动器可以以较高的性能顺序地读取磁盘页，通常的速度为 100MB/s。同时，由于磁盘存在机械部件，磁盘中的寻址操作代价比较高，一个较短的寻址往往也会消耗 3-4ms 的时间。在采用 PAX 模型时，收益主要来自于跳过一些无用的迷你页，到达查询所需要的迷你页，只读取查询需要的数据。因此，要想获得收益，必须要求跳过无用的迷你页所节省的读取无用页的开销，必须大于由于跳过无用页而引起的磁盘寻址开销。就这个方面而言，在磁盘中采用 PAX 模型不会获得收益，而在闪存固态盘中采用 PAX 模型可以获得很大的收益：

第一，在磁盘中采用 PAX 模型不会获得收益。对于在磁盘上跳过迷你页的一个寻址而言，至少要一次跳过 300KB-400KB(100MB/s*3-4ms)的迷你页，才可以获得收益，也就是说，在磁盘中采用 PAX 模型时，需要把一个迷你页的大小设计为 300-400KB。但是，采用 PAX 模型时，磁盘中的一个页包含多个迷你页，如果一个迷你页都已经达到 300KB，那么，一个页的大小会达到好几个 MB。但是，对于一个关系型 DBMS 而言，在设计页的大小时，会综合考虑诸多因素，比如缓冲池尺寸、缓冲区命中率、更新频率和每页的算法开销等。而在考虑了上述这些因素以后，大多数关系型 DBMS 往往都倾向于使用更小的页，通常是 8KB 或者 64KB，而不是几个 MB。因此，在磁盘中采用 PAX 模型，不会为关系型 DBMS 带来性能收益。

第二，在固态盘中采用 PAX 模型可以获得很大的收益。从上面论述可以看出，PAX 模型对于磁盘而言，没有改进读性能，但是，在闪存上却可以获得明显的性能收益。闪存是一个纯粹的电子设备，不包含任何机械部件，因此，闪存上的寻址开销很小，随机读和顺序读操作几乎具有相同的高性能。假设一个闪存存储设备具备 28MB/秒的顺序读带宽和 0.25ms 的寻址时间，那么，它只要跳过 7KB（28MB/s*0.25ms）的迷你页即可。因此，一个页的大小就只需要 32KB-128KB，这恰好是当前许多 DBMS 中采用的数据库页的大小。因此，在闪存上采用 PAX 模型是可行的。

此外，在固态盘上，PAX 和更快的寻址时间相结合，允许只读取查询所需要的那些列，因此，既保留了现有的 NSM 的功能，也具备了 DSM 较高的读效率。而且，内存中的数据传输单元，可以小到 32-128B，而一个数据库页通常是 8KB 到 128KB，固态盘的最小传输单元是 512B 到 2KB，这就允许我们可以选择性地读取常规页的一部分，减少了读操作数量。

8.2 连接索引和Jive连接算法

8.2.1.1 概述

两个关系 R_1 和 R_2 之间的连接索引，包含了一系列元组“标识对”，表示为<t_1,t_2>，其中，t_1 表示来自关系 R_1 的元组的 ID，t_2 表示来自关系 R_2 的元组的 ID，并且 t_1 和 t_2 之间在连接条件上是匹配的，即 t_1 和 t_2 可以发生连接操作成为连接结果的一部分。可以把连接索引看成是一个关系，用 J 表示。并且假设连接索引是根据 t_1 或 t_2 进行排序的，为了不失一般性，可以假设根据 t_1 进行排序。

在许多情形下，采用连接索引的连接算法可以比最好的即席连接算法（比如混合哈希连接算法[DeWittKOSSW84]）获得更好的性能。Patrick Valduriez[Valduriez87]在 1987 年设计了使用连接索引的连接算法，这个算法可以很简单地实现，而且对于选择性连接非常有效。但是，对于 Valduriez 算法而言，当输入关系的大小大于内存时，需要对其中一个关系执行多趟扫描。因此，就性能而言，Valduriez 的算法可能导致许多重复的 IO，有时候只为了访问

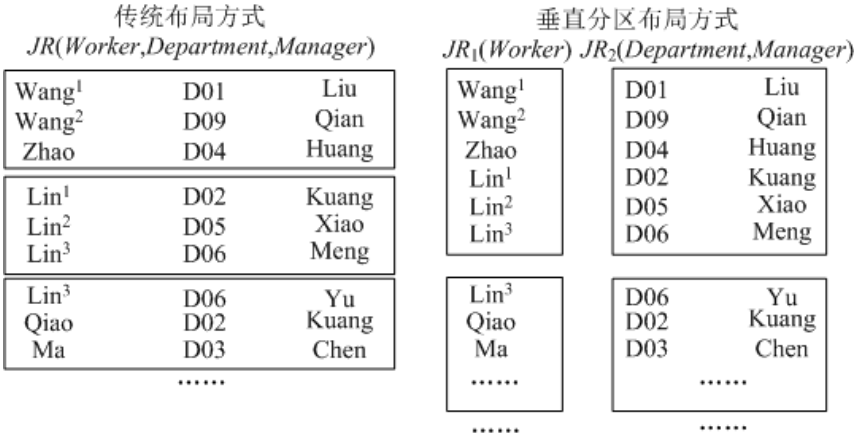
一少部分元组就需要访问整个块。而且，相同的块可能会被多次读取。

Li 等人在 1999 年提出了针对 Valduriez 的算法的改进算法——Jive(Join Index with Voluminous relation Extensions)连接算法[LiR99]。Jive 连接算法会对第二个输入关系的元组 ID 根据值域进行分区，然后单独处理每个分区。Jive 算法的特性包括：

(a) 所有的 IO 操作都是顺序执行的；
(b) 一个输入关系的块会被读入，当且仅当它包含了一个参与连接的记录；
(c) 索引连接只被读取一次；
(d) 在内存容量够大的情况下，可以只对每个输入关系执行一次扫描；
(e) 唯一需要存储的中间结果就是元组 ID，通常是很小的。

Jive 算法具有较好的特性是因为连接结果使用了垂直分区的数据结构。不过，Jive 并没有对输入关系进行垂直分区，而是对连接结果进行垂直分区。存储连接结果的垂直分区数据结构，被称为“转置文件(transposed file)”[Batory79]。来自输入关系 R_1 并且存在于连接结果当中的属性，会被存储到一个单独的文件中，称为 JR_1，同理，来自输入关系 R_2 并且存在于连接结果当中的属性，会被存储到另一个单独的文件中，称为 JR_2。属于两个输入关系 R_1 和 R_2 的公共属性，即连接属性，会被存储到 JR_1 或 JR_2 中的任意一个中。每个文件（JR_1 或 JR_2）中的第一个条目（entry），都对应着第一个连接结果元组，第二个条目，对应着第二个连接结果元组，依此类推。由于每个垂直分区采用相同的顺序，因此，两个分区中的每个连接结果元组都存在一一对应关系，这就意味着，不需要额外存储元组 ID 或代理键，就可以直接对两个垂直分区进行组合得到最终连接结果。

图 Jive-example-transposed-file 演示了连接结果的垂直分区，图中左边显示了采用传统的布局方式时的连接结果 *JR*（包含了 *Worker*、*Department* 和 *Manager* 属性），右边显示了采用垂直分区布局时的连接结果 JR_1（包含 *Worker* 属性）和 JR_2（包含 *Department* 和 *Manager* 属性）。可以看出，对于传统的布局方式和垂直分区布局方式而言，连接结果占用的空间开销都是相同的。



图[Jive-example-transposed-file] 连接结果的垂直分区

对连接结果采用了垂直分区方式以后，Jive 连接算法就可以分成两趟输出连接结果，每一趟的连接结果都对应于来自其中的一个输入关系的属性，最后对两个垂直分区进行组合就可以得到最终连接结果。因此，进行垂直分区方式的一个明显优点是，不需要同时把全部元组都存放在内存中，而是在不同的阶段分次写入每个垂直分区，这样可以更好地利用内存。

8.2.1.2 Jive 连接算法的步骤

Jive 连接算法会把来自 R_1 的元组和来自连接索引的记录进行匹配，同时，R_2 的记录会被分区，每个分区都包含了 R_2 元组 ID 的一个区间，并且根据 R_2 中元组的 ID，让 R_1 和 R_2 的元组进行匹配。发生匹配的 R_1 记录以及与它匹配的 R_2 元组 ID，会被存放到不同的输出文

件中。发生匹配的 R_1 记录会输出到一个输出文件 JR_1 中，这个输出文件包含了连接操作中发生匹配的 R_1 记录的集合，而与这些 R_1 记录相匹配的 R_2 元组 ID，则会输出到一个临时文件中，这个临时文件包含了与 R_1 记录相匹配的 R_2 元组 ID 的集合。然后，临时文件会和关系 R_2 中的记录一起被处理，来生成另一半连接结果 JR_2。

Jive 算法包括三个步骤：

- 第 1 步：确定分区；
- 第 2 步：生成一半连接结果 JR_1；
- 第 3 步：生成另一半连接结果 JR_2。

（1）第 1 步：确定分区

选择 x-1 个 R_2 元组 ID 作为分区元素，这些分区元素可以确定 R_2 元组 ID 的 x 个分区。每个分区在内存中都有与其相关联的输出文件缓冲区，并且有与其关联的临时文件缓冲区。当缓冲区满的时候，每个缓冲区都会把内容刷新到一个相应的输出文件和磁盘临时文件中，然后接收新的记录。

（2）第 2 步：生成一半连接结果 JR_1

以顺序的方式扫描连接索引 J 和关系 R_1（注意：前面已经假设 J 是以 R_1 元组的顺序存储的），由于连接索引 J 中包含了 R_1 元组 ID，因此，可以对 J 和 R_1 中的元组进行匹配。在读取 R_1 的过程中，如果决定读入关系 R_1 的一个块，当且仅当这个块包含了连接索引中涉及的元组。在每次匹配时，首先需要确定一个发生匹配的元组所属的分区，这个可以根据 R_2 元组的 ID 来判定，因为在第 1 步中就是根据 R_2 元组 ID 来分区的，连接索引 J 的每个条目都是一个元组标识对<t_1,t_2>，根据属于关系 R_2 的 t_2 就可以判定 t_1 属于那个分区。在确定一个发生匹配的元组所属的分区以后，就可以把关系 R_1 中 ID 为 t_1 的元组中被连接结果所需要的属性，写入到内存中为这个分区准备的缓冲区文件中。同时，发生匹配的 R_2 元组 ID 会被写入到为这个分区准备的临时文件缓冲区中。当连接索引 J 都已经读取完毕的时候，所有的文件缓冲区都被刷新到磁盘上，为文件缓冲区分配的内存被回收重新分配。在完成第 2 步以后，已经生成了一半的连接结果，即 JR_1。JR_1 的分区会被连接到一个文件中。同时会生成一个临时文件，在下面的步骤 3 中被用来生成另一半连接结果 JR_2。

（3）第 3 步：生成另一半连接结果 JR_2

对于临时文件的每个分区，分别执行以下操作：把整个分区读入内存，并且对 R_2 元组 ID 进行升序排序，消除重复元组 ID。需要指出的是，临时文件的原来版本一直保存着，排序得到的结果会单独存放在新的临时文件中。然后，从 R_2 中一直顺序地读取元组，并且根据已经排序的临时文件，只从 R_2 中读取包含了一个匹配记录的块。当从 R_2 中读取的 ID 比分区中的 ID 大时，就暂时停止读取 R_2 元组。到了这个点，已经得到了连接结果 JR_2 的一个分区，可以采用以下方式把这个分区写入文件：查看为这个分区准备的临时文件的原始版本（即未经排序的版本），把相应的 R_2 元组中属于连接结果的属性，以原始版本临时文件中的顺序写入到 JR_2 的分区中。然后，对于临时文件的下一个分区，继续执行上面的操作。当已经写完最后一个分区到 JR_2 中时，就生成了所有的 JR_2 分区。JR_2 的不同分区可以被连接成单个文件。

最后，对第 2 步产生的连接结果 JR_1 和第 3 步产生的连接结果 JR_2 进行合并，就得到所需要的连接结果。注意，在第 2 步和第 3 步中，如果 R_1 或 R_2 的某个块根本不会包含参与连接操作的元组，可以不用读取这个块。因此，如果输入关系中只有一少部分参与了连接操作，这种方式可以让连接算法获得更好的性能。

8.2.1.3 Jive 连接算法的一个实例

为了更好地理解 Jive 连接算法，这里给出一个实例。如图[Jive-example]所示，两个关系 R_1 和 R_2 分别是 *Employee*（见图[Jive-example](a)）和 *Management*（见图[Jive-example](b)），

其中，*Employee* 关系包含了两个属性 *Worker* 和 *Department*，分别表示雇员姓名和所在部门编号；*Management* 关系包含了两个属性 *Department* 和 *Manager*，分别表示部门编号和部门管理员。这里把每个关系中的元组出现的序号，作为元组 ID，比如 *Employee* 关系中的第 1 个元组<Wang,D01>，它的元组 ID 就是 1，第 2 个元组<Wang,D08>，它的元组 ID 就是 2，依次类推。需要注意的是，为了对属性值相同的元组进行区分，从而便于跟踪连接算法的执行过程，这里为相同属性值添加了上角标。对 *Employee* 关系和 *Management* 关系在 *Department* 属性上进行连接操作，可以得到连接结果，如图[Jive-example](c)所示。

Worker	Department
Wang[1]	D01
Wang[2]	D09
Zhao	D04
Lin[1]	D02
Lin[2]	D05
Lin[3]	D06
Qiao	D02
Ma	D03
Qin	D07

(a)Employee关系

Department	Manager
D01	Liu
D02	Kuang
D03	Chen
D04	Huang
D05	Xiao
D06	Meng
D06	Yu
D08	Luo
D09	Qian

(b)Management关系

Worker	Department	Manager
Wang[1]	D01	Liu
Wang[2]	D09	Qian
Zhao	D04	Huang
Lin[1]	D02	Kuang
Lin[2]	D05	Xiao
Lin[3]	D06	Meng
Lin[3]	D06	Yu
Qiao	D02	Kuang
Ma	D03	Chen

(c)连接结果

Employee 元组ID	Management 元组ID
1	1
2	9
3	4
4	2
5	5
6	6
6	7
7	2
8	3

(d)连接索引

图[Jive-example] Jive 连接算法的一个实例

为了执行 Jive 连接算法，需要创建一个连接索引，如图[Jive-example](d)所示。连接索引的每个条目都是一个元组标识对<t_1,t_2>，并且 t_1 和 t_2 在连接条件上可以匹配。因此，对于 *Employee* 关系而言，第 1 个元组的部门编号是 D01，元组 ID 是 1，在 *Management* 关系中与之匹配的元组是第 1 个元组<D01,Liu>，元组 ID 是 1，因此，可以得到连接索引的第 1 个条目是<1,1>。同理，对于 *Employee* 关系而言，第 2 个元组的部门编号是 D08，元组 ID 是 2，在 *Management* 关系中与之匹配的元组是第 9 个元组<D09,Qian>，元组 ID 是 9，因此，可以得到连接索引的第 2 个条目是<2,9>。依此类推，可以得到连接索引的其他所有条目。

在整个构建连接索引的过程中可以发现，*Management* 关系不存在部门编号为 D07 的元组，因此，*Employee* 关系中的第 9 个元组<Qin,D07>，在 *Management* 关系中不存在与之匹配的元组。另外，*Management* 关系中，部门编号为 D06 的元组有两个，分别是第 6 个元组<D06,Meng>和第 7 个元组<D06,Yu>，因此，*Employee* 关系中的第 6 个元组<Lin[3],D06>，在 *Management* 关系中存在两个与之匹配的元组，即第 6 个元组<D06,Meng>和第 7 个元组<D06,Yu>。

下面将演示按照 Jive 连接算法的三个步骤依次执行连接操作。

（1）第 1 步：确定分区

假设这里把 *Management* 关系（R_2）分成三个分区，即 x=3，并且假设每个缓冲区每次

只能容纳一个元组。这里选择 *Management* 关系的元组 ID 值 3 和 6 作为分区点，即 ID 小于 3 的元组构成一个分区 *a*，ID 大于等于 3 并且小于 6 的元组构成一个分区 *b*，ID 大于等于 6 的元组构成一个分区 *c*。

对于这三个分区而言，每个分区在内存中都有与其相关联的输出文件缓冲区，每个输出文件缓冲区满了以后，都会把缓冲区的内容刷新到输出文件 JR_1 的对应分区中，因此可以得到三个输出文件分区 $JR_1(a)$、$JR_1(b)$和 $JR_1(c)$，并且有与其关联的临时文件缓冲区，临时文件缓冲区满了以后，也会输出到临时文件对应的分区中，得到三个临时文件分区 *Temp*(*a*)、*Temp*(*b*)和 *Temp*(*c*)。随着连接操作的不断进行，缓冲区内容会不断输出到文件分区中，因此，这些文件分区 $JR_1(a)$、$JR_1(b)$、$JR_1(c)$、*Temp*(*a*)、*Temp*(*b*)和 *Temp*(*c*)，会不断增加内容。在 Jive 算法的全部步骤都完成以后，可以对输出文件分区 $JR_1(a)$、$JR_1(b)$、$JR_1(c)$进行合并，得到一个输出文件 JR_1。

（2）第 2 步：生成一半连接结果 JR_1

以顺序的方式扫描连接索引 *J* 和关系 *Employee* 关系（R_1），对连接索引和 *Employee* 关系中的元组进行匹配。为了生成一半的连接结果 JR_1，在 *Employee* 关系中找到与连接索引 $<t_1,t_2>$的左边元素 t_1 相对应的元组后，需要把该元组中出现在连接结果中的属性 *Worker* 的值写入到 JR_1 中（实际上是首先写入输出文件缓冲区中，但是，最终会被合并成一个 JR_1 文件）。比如，连接索引的第 1 个条目<1,1>的左边元素是 1，因此，需要把 *Employee* 关系（R_1）的第 1 个元组<Wang[1],D01>的 *Worker* 属性值“Wang[1]”，作为部分结果输出。但是，需要注意的是，属性值“Wang[1]”必须输出到指定的输出文件分区中。

Employee 关系（R_1）中的元组被输出到哪个输出文件分区，是由连接索引中属于 *Management* 关系（R_2）的元组标识符 t_2 来决定的，t_2 属于哪个分区，就把 t_1 对应的 Employee 关系（R_1）的元组写入到这个分区。连接索引的第 1 个条目<1,1>的右边元素 t_2 的值是 1，Management 关系（R_2）的元组标识符为 1 的元组，会被划分到分区 *a*，因此，t_1 对应的元组的 Worker 属性值“Wang[1]”会被写入到输出文件分区 JR_1(a)中（实际上是首先输出到为输出文件分区 $JR_1(a)$准备的内存缓冲区中，缓冲区满后才会被刷新到输出文件分区 $JR_1(a)$中）。连接索引的第 1 个条目的右边元素 t_2 的值 1，会被写入到为分区 *a* 对应的临时文件 *Temp*(*a*)中（实际上是首先输出到为临时文件 *Temp*(*a*)准备的内存缓冲区中，缓冲区满后才会被刷新到临时文件 *Temp*(*a*)中）。

连接索引的第 1 个条目处理结束之后，继续扫描连接索引的第 2 个条目<2,9>，*Management* 关系（R_2）的元组标识符 t_2 为 9 的元组，会被划分到分区 *c*，因此，把 t_1=2 对应的 Employee 关系（R_1）的元组的 *Worker* 属性值“Wang[2]”写入到输出文件分区 $JR_1(c)$中。依此类推，继续顺序扫描连接索引的其他条目，并把相关的 Worker 属性值写入到对应的输出文件分区中。

最终，可以得到图[Jive-example-join-JR1]中的三个输出文件分区 $JR_1(a)$、$JR_1(b)$和 $JR_1(c)$，同时得到三个临时文件分区 *Temp*(*a*)、*Temp*(*b*)和 *Temp*(*c*)。$JR_1(a)$、$JR_1(b)$和 $JR_1(c)$会被合并成一个文件 JR_1，这时，就得到了一半的连接结果。*Temp*(*a*)、*Temp*(*b*)和 *Temp*(*c*)会在下面的步骤 3 中使用，被用来生成另一半连接结果 JR_2。

需要注意的是，属性 *Department* 属于两个输入 *Employee* 关系（R_1）和 *Management* 关系（R_2）的公共属性，即连接属性，可以被存储到 JR_1 或 JR_2 中的任意一个中，这里假设存储到 JR_2 中，因此，在前面一半的连接结果 JR_1 中不存在 *Department* 属性。

(a) ID<3		(b) 3≤ID<6		(c) 6≤ID	
Wang[1]	1	Zhao	4	Wang[2]	9
Lin[1]	2	Lin[2]	5	Lin[3]	6
Qiao	2	Ma	3	Lin[3]	7
JR1(a)	Temp(a)	JR1(b)	Temp(b)	JR1(c)	Temp(c)

图[Jive-example-join-JR1] Jive 连接算法的一半结果 JR1

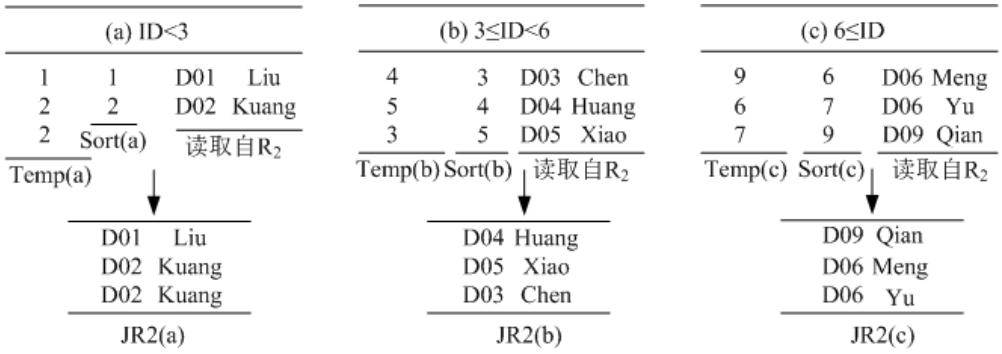
（3）第 3 步：生成另一半连接结果 JR_2

对于临时文件的分区 *Temp*(*a*)，把整个分区读入内存，并且对 *Management* 关系（R_2）元组 ID 进行升序排序，消除重复元组 ID，因此，可以得到排序后的分区 *Sort*(*a*)。需要注意的是，*Sort*(*a*)会被保存到新的文件中，原来的分区文件 *Temp*(*a*)仍然存在。然后，从 *Management* 关系（R_2）中一直顺序地读取元组，并且根据已经排序的临时文件 *Sort*(*a*)，只从 *Management* 关系（R_2）中读取包含了一个匹配记录的块。*Sort*(*a*)中的最大元组 ID 是 2，因此，在 *Management* 关系（R_2）中读取完第 2 个元组以后，就可以停止读取，这时，*Management* 关系（R_2）中的第 1 个<D01,Liu>和第 2 个元组<D02,Kuang>已经被读入到内存中。

接下来要把这些元组输出到连接结果文件分区 $JR_2(a)$中，也就是说，输出文件 JR_2 也是根据分区 *a*、*b* 和 *c* 被分成三个输出文件分区 $JR_2(a)$、$JR_2(b)$和 $JR_2(c)$，R_2 中元组 ID 为 1 和 2 的两个元组在进入连接结果 JR_2 中时，都会被写入到 $JR_2(a)$中（实际上是先写入为 $JR_2(a)$准备的缓冲区，缓冲区满后再写入到文件 $JR_2(a)$中），其他连接结果也都会按照相同的办法被写入到各自所属的输出文件分区中。在把这些元组输出到连接结果文件分区 $JR_2(a)$中时，采用的是未经排序的原始版本的 *Temp*(*a*)中的元组顺序，*Temp*(*a*)中的第 1 个元组 ID 是 1，因此，首先把已经读入内存的 *Management* 关系（R_2）的元组<D01,Liu>输出到 $JR_2(a)$中，*Temp*(*a*)中的第 2 个元组 ID 是 2，因此，需要把内存中的第 2 个元组<D02,Kuang>输出到 $JR_2(a)$中，*Temp*(*a*)中的第 3 个元组 ID 仍然是 2，因此，需要再次把内存中的第 2 个元组<D02,Kuang>输出到 $JR_2(a)$中，由此可以得到输出文件分区 $JR_2(a)$，如图[Jive-example-join-JR2]左边所示。

接下来，把临时文件分区 *Temp*(*b*)全部读入内存，按照上面的方式，计算得到连接结果的第 2 个分区文件 $JR_2(b)$，如图[Jive-example-join-JR2]中间所示。然后，把临时文件分区 *Temp*(*c*)全部读入内存，按照上面的方式，计算得到连接结果的第 3 个分区文件 $JR_2(c)$，如图[Jive-example-join-JR2]右边所示。

已经得到 JR_2 的不同分区 $JR_2(a)$、$JR_2(b)$和 $JR_2(c)$以后，可以把 $JR_2(a)$、$JR_2(b)$和 $JR_2(c)$连接成单个文件 JR_2，如图[Jive-example-join-JR]所示。



图[Jive-example-join-JR2] Jive 连接算法的另一半结果 JR_2

	Worker	Department	Manager	
JR1(a)	Wang[1]	D01	Liu	JR2(a)
	Lin[1]	D02	Kuang	
	Qiao	D02	Kuang	
JR1(b)	Zhao	D04	Huang	JR2(b)
	Lin[2]	D05	Xiao	
	Ma	D03	Chen	
JR1(c)	Wang[2]	D09	Qian	JR2(c)
	Lin[3]	D06	Meng	
	Lin[3]	D06	Yu	

图[Jive-example-join-JR] Jive 连接算法的最终结果

从上面实例的执行过程可以发现，Jive 连接算法具备几个重要的特性：（1）算法只会读取参与连接的元组；（2）参与连接的元组只会被读取一次，即使该元组可能会参与多次连接运算；（3）对每个输入关系都只要进行一趟扫描。

8.3 基于PAX的RARE连接算法

文献[ShahHWG08]提出了面向闪存的、采用 PAX 模型和连接索引的 RARE 连接算法，该算法借鉴了 Jive 算法的设计思路，当输入关系太大无法一次性放入内存时，采用连接索引，先生成一半的连接结果，然后再生成另一半连接结果，并且，在扫描输入关系的过程中，不仅只会访问那些参与到连接操作中的列，而且只访问那些包含了连接结果所需要的值的迷你页，而不是访问所有的输入关系，以此来达到节省 IO 开销的目的，提高查询性能。需要注意的是，这种 IO 开销的节省需要额外的代价，即会增加寻址开销（从一个迷你页跳到另一个迷你页），并且需要计算连接索引。但是，RARE 连接算法节省的 IO 开销，超过了额外的代价，因此，在理论和实际应用中都是可行的。

下面将以经典的 Grace 哈希连接算法作为参照对象，来讨论 RARE 连接算法在两种情形下的性能：（1）当存在足够的内存时，只需要对输入关系扫描一趟来计算连接；（2）当输入关系太大无法一次性放入内存时，需要多趟输入的时候。

需要指出的是，前面已经通过实例详细介绍了 Jive 连接算法的执行过程，由于 RARE 连接算法基本采用了 Jive 连接算法的设计思路，因此，下面只会介绍 RARE 连接算法的基本过程，不再给出实例演示。

8.3.1.1 一趟扫描

首先考虑采用 NSM 模型的 Grace 哈希连接算法。假设有两个参与连接操作的关系 R_1 和 R_2，并且为 R_2 建立的哈希表 h 可以一次性放入内存，即$|h(R_2)|<|M|$，其中，$h(R_2)$表示为 R_2 建立的哈希表 h 的空间开销，|M|表示内存空间大小。这时，Grace 哈希连接算法只需要对输入关系执行一次扫描，就可以计算得到结果，具体方法如下：

- 首先，读取 R_2，并且在内存中为 R_2 构建一个哈希表 h；
- 然后，读取 R_1，探测哈希表 h，并且把结果输出到 S 中。

在上述过程中，所有的访问都是顺序执行的，因此，总体的 IO 代价的计算方法很简单，即$|R_1|+|R_2|+|S|$。

现在考虑采用 PAX 模型的 Grace 哈希连接算法，简称“Grace-PAX”。和 Grace 哈希连接算法相比，Grace-PAX 连接算法可以取得更好的性能，主要是由于两个方面的原因：

第一，Grace-PAX 需要更少的内存来执行一趟操作，因为，R_2 中只有与连接和投影相关的列才会被读入内存，并且可以满足$|h(J_2)|+|h(V_2)|<|M|$，其中，J_2 表示关系 R_2 中参与连接的列，简称“连接列”，V_2 表示 R_2 中会被投影到连接结果的列，简称“投影列”。比如，对于图[Jive-example]中 Jive 连接算法实例而言，在输入关系 Management（R2）中（如图[Jive-example](b)所示），*Department* 列就是参与连接的列，即 J_2，Manager 列就是会被投影到连接结果的列，即 V_2。

第二，由于它跳过了不需要的列。因此，对于 Grace-PAX 而言，总共的 IO 代价会少于 $|J_1+V_1|+|J_2+V_2|+S$。

再来考虑采用 PAX 模型的 RARE 连接算法（如算法[1-pass-RARE]所示）。对于 RARE 连接算法而言，它只会读取关系 R_2 中那些参与连接的列 J_2 和行 ID 号（即 id_2），并且构建哈希表。然后，它使用 J_1 来探测哈希表，输出连接结果。在采用了 PAX 模型后，每个页都存储了相同数量的行（或元组），在每个页内部，这些行会以列的方式被存储到不同的迷你页中。因此，与 Grace-PAX 不同的是，RARE 并不是读取 V_1 和 V_2 中的所有内容，而是只会读取那些可以产生连接结果的迷你页，即从 V_2 中只读取那些包含行 ID 号（即 id_2）的迷你页，从 V_1 中只读取那些包含行 ID 号（即 id_1）的迷你页。

需要指出的是，RARE 连接算法在执行输入关系的扫描时，会对 R_1 进行顺序扫描，因此，在连接算法执行过程中，新扫描进来的 V_1 迷你页可以立即替换掉内存中旧的迷你页，不会引起内存空间消耗的不断增加，在整个过程中只需要在内存中保留 V_2 迷你页。最终，RARE 连接算法会把结果输出到 S 中。

可以看出，采用 PAX 模型的 RARE 连接算法的内存需求是$(|h(J_2)|+|h(id_2)|)+|\sigma_{p2}(V_2)|<|M|$，其中，$\sigma_{p2}(V_2)$表示 V_2 中会被连接计算使用到的那些元组。RARE 连接算法的总 IO 代价是：$|J_1|+|\sigma_{p1}(V_1)|+|J_2|+|\sigma_{p2}(V_2)|+|S|$。由此可以看出，在采用大约相同的内存的情况下，单趟 RARE 连接算法要比单趟 Grace-PAX 连接算法减少了 IO 代价。

1. Read J_2 and build hash-table;
2. Read J_1 and probe hash-table;

foreach join result <id_1, id_2> do
{
Read projected values of row id_1 from V_1;
Read projected values of row id_2 from V_2;
Write result into S;
}

算法[1-pass-RARE] 单趟 RARE 连接算法

8.3.1.2 多趟扫描

当内存空间有限，或者关系 R_2 太大时，为 R_2 构建的哈希表就无法一次性放入内存，这时，Grace 哈希连接算法就需要执行多趟（pass）扫描。另外，对于 Grace-PAX 连接算法而言，如果 J_2 和 V_2 无法一次性放入内存，它也会退化成多趟扫描。第一趟会在连接列上对两个输入关系进行分区，从而使得单个分段可以一次性放入内存。第二趟会读取每个分段到内存中，计算连接结果。

因此，对于多趟 Grace 哈希连接而言，总的 IO 开销就是：$3(|R_1|+|R_2|)+|S|$，因为，第一趟分区操作，需要一次读入（IO 开销为$|R_1|+|R_2|$）和一次输出（IO 开销为$|R_1|+|R_2|$），第二趟的 IO 开销和原来的“1 趟 Grace 哈希”连接算法相同，都是$|R_1|+|R_2|+|S|$，因此，总共的 IO 开销是 $3(|R_1|+|R_2|)+|S|$。

类似地，对于 Grace-PAX 而言，IO 总开销就是：

$3(|J_1+V_1|+|J_2+V_2|)+|S|$ 公式(Grace-PAX)。

下面将分两种情况介绍 RARE 连接算法可以比 Grace-PAX 连接算法获得更好的性能。

- 第一种情况：J_2 能够一次性放入内存，但是 V_2 无法一次性放入内存，这时，对于 RARE 算法而言，需要采用(1+ε)趟 RARE 连接算法;
- 第二种情况：J_2 和 V_2 都无法一次性放入内存，这时，对于 RARE 算法而言，需要采用“2 趟 RARE 连接算法”。

8.3.1.3 (1+ε)趟 RARE 连接算法

当 J_2 可以一次性放入内存而 V_2 无法一次性放入内存时，RARE 连接算法（如算法[1+pass-RARE]所示）仍然可以只对输入关系进行一趟扫描来计算得到连接结果。

RARE 连接算法执行第 1 步时，首先读取 J_2，并在 J_2 上面构建一个哈希表。然后，读取 J_1，并使用 J_1 对哈希表进行探测，探测这个哈希表的结果就是“连接索引”（和 Jive 连接算法中介绍过的连接索引是完全相同的概念，在 RARE 算法中也会发挥相同的作用）。对于出现在连接结果 S 中的每一行而言，在连接索引中都有与之对应的一个条目。由于探测哈希表时采用了 J_1 的顺序，因此，那些需要的 V_1 迷你页就可以被顺序地读取，并且直接被写入到结果 S 中。由于使用了 PAX 模型，因此，第 2 步只会把 V_1 中的投影列（即 V_1 中那些出现在连接结果中的列）写入到 S 中，并为每个连接结果中的属于 V_2 的列预留一个空位置，直到第 3 步的时候，再用 V_2 中的列值来填充这个空位置。从这里可以看出，RARE 连接算法借鉴了 Jive 连接算法的思想，因为，Jive 连接算法也是先生成一半的连接结果，然后再生成另一半的连接结果。需要指出的是，这种针对 S 的写模式，在闪存上是十分高效的，因为，在这个步骤已经付出了闪存擦除操作的代价，在第 3 步中只需要在预留的空位置写入值即可。

和一趟排序不同的是，这里无法把 V_2 一次性放入内存。一个简单的方法是，根据需要读取所需要的 V_2 迷你页。但是，由于在第 2 步中采用 J_1 的顺序对哈希表进行探测来生成连接索引，因此，在执行第 3 步的时候，采用这种方法可能会导致多次去读取同一个 V_2 迷你页。为了避免多次读取同一个 V_2 迷你页，RARE 算法借鉴了 Jive 连接算法的思想，在第 2 步中把连接索引分区成多个分段，使得每个分段中引用到的 V2 迷你页，都可以一次性放入内存。前面介绍 Jive 连接算法的时候，已经知道，Jive 连接算法[LiR99]会创建连接索引的排序分段，然后进行合并，从而以后可以顺序地获取 R_2 的行。但是，这里需要指出的是，当采用闪存时，并不需要像 Jive 连接算法一样以顺序的方式访问 V_2，而是只需要确保所有针对单个页的数据值都可以从一个页读操作中获得。这主要是由于闪存和磁盘的不同特性决定的。Jive 连接算法是面向磁盘设计的，因此，必须尽量执行顺序读取操作，所以，对 R_2 的扫描都是顺序进行的。但是，对于闪存而言，顺序读取和随机读取操作具有同样高的性能，因此，就不需要对 R_2 执行顺序读取操作。所以，为了确保所有针对单个页的数据值都可以从一个页读操作中获得，在步骤 2 当中，RARE 连接算法简单地采用了 R_2 的页号来对连接索引进行分区，从而，所有属于同一个页的 R_2 行号（即 id_2）都会进入同一个连接索引分段。这样，在对某一个分段执行连接操作时，需要读取的 R_2 的行就会属于同一个页，只需要一个页读操作，就可以读取到连接操作所需要的 R_2 中的行。

在连接条件上发生匹配的 R_2 中的元组的 ID 会被写入到一个临时文件 I_2 的对应分区中（相当于 Jive 连接算法的第 2 步中把 R_2 的元组 ID 写入到临时分区 $Temp(a)$、$Temp(b)$或 $Temp(c)$ 中，从而在第 3 步用来生成另一个连接结果）。

对于步骤 3 而言，属于同一个分段的整个 V_2 页的集合，必须一次性放入内存，这点与 Jive 连接算法是相同的，Jive 连接算在执行第 3 步时也有类似要求。需要的分段的数量就是 $|V_2|/|M|$，采用的分区函数可以是一个哈希函数，或者域分区机制。RARE 连接算法对 S 采用相同的分区方式，从而在步骤 3 中，可以使用相应的 V_2 值来对填充 S 中的空白部分（相当于 Jive 连接算法中的第 3 步生成另一半连接结果 JR_2）。

由于需要为 S 和 I_2 的每个分段提供一个缓冲页，并且最多有$|V_2|/|M|$个分段，因此，(1+ε)趟 RARE 连接算法的内存需求就是$(|h(J_2)|+|h(id_2)|)+2*(|V_2|/|M|)<|M|$。所有三个步骤的总共 IO 代价就是：

$|J_1|+ \sigma_{p1}|V_1|+|J_2|+\sigma_{p2}|V_2|+|S|+2|I_2|$ 公式(RARE-1+-pass-cost)。

由前面公式(Grace-PAX)可知，Grace-PAX 的 IO 代价是 $3(|J_1+V_1|+|J_2+V_2|)+|S|$。因此，把 Grace-PAX 的 IO 代价（公式(Grace-PAX)）和(1+ε)趟 RARE 算法的 IO 代价（公式

(RARE-1+-pass-cost)）进行相减，当二者差值大于 0 时， RARE 算法的性能更好。也就是说，当以下条件成立时， RARE 算法的性能要比 Grace-PAX 更好：

$2|J_1|+ (3-\sigma_{p1})|V_1|+2|J_2|+(3-\sigma_{p2})|V_2|>2|I_2|$。

条件的左边是节省的开销，包括只读取连接涉及的列和 V_1、V_2 中所需要的迷你页，只需要读取一次，而不是三次。这种节省的开销，必须能够超过下面的额外代价：即从连接索引中读取行 ID（id_2）和物化的开销。

1. Read J_2 and build hash-table;
2. Read J_1 and probe hash-table;

foreach join result <id_1, id_2> do
{
Read projected values of row id_1 from V_1;
/*S and I_2 are both partitioned by id_2*/
Write projected values into partition of S;
Write id_2 into partition of I_2;
}

3. Read I_2 and process it.

foreach partition of I_2 do
  foreach id_2 in partition do
    {
      Read projected values of row id_2 from V_2;
      Write values into partition of S;
    }

算法[1+pass-RARE] (1+ε)趟 RARE 连接算法

8.3.1.4 2 趟 RARE 连接算法

算法[2-pass-RARE]显示了当 J_2 和 V_2 都无法一次性放入内存的情况下 RARE 算法的伪代码。在这种情形下，步骤 1 和步骤 2 和 Grace 哈希类似。RARE 哈希会把两个表的连接列进行分区，从而，每个 J_2 分段都可以一次性放入内存。在步骤 3 中，它为每个分段计算和物化连接索引<id_1,id_2>。

在步骤 4 中，RARE 会把 J_1 的分段合并成 R_1 的顺序，并且读取需要的投影列 V_1。第 5 步和(1+ε)算法的第 3 步一样。需要注意的是，连接索引的尺寸是 I_2 大小的两倍，因为，它包含了 id_1 和 id_2。此外还要注意，RARE 只会读取 V_1 和 V_2 中所需要的迷你页。

整个过程的代价构成如下：

（1） 步骤 1 中，读取 J_2 并进行分段后写入到存储介质上，需要一次读取（IO 开销是$|J_2|$）和一次写入操作（IO 开销是$|J_2|$）；

（2） 步骤 2 中，读取 J_1 并进行分段后写入到存储介质上，需要一次读取（IO 开销是$|J_1|$）和一次写入操作（IO 开销是$|J_1|$）；

（3） 步骤 3 中，需要读入 J_2 的所有分段，总的 IO 开销是$|J_2|$；同时需要读入 J_1 的所有分段，进行哈希表探测，因此，整个过程需要的 IO 开销是$|J_1|$。另外，写入连接索引 JI 的 IO 开销是 $2|I_2|$（因为连接索引的大小是 I_2 大小的两倍）。

（4） 步骤 4 中，需要读出和写入连接索引，IO 开销是 $4|I_2|$；另外，需要从 V_1 中读取需要的数据，IO 开销是 $\sigma_{p1}|V_1|$；把投影属性值写入 S 的 IO 开销是$|S|$，把 id_2 写入 I_2 的开销是$|I_2|$。

（5） 步骤 5 中，读取 I_2 的 IO 开销是$|I_2|$，另外，需要从 V_2 中读取需要的数据，IO 开销是 $\sigma_{p2}|V_2|$；把这些值写入 S 的 IO 开销可以不用再次计算，因为在步骤 4

中已经计算一次。

因此，可以计算得到总共的 IO 代价是：

$3|J_1|+ \sigma_{p1}|V_1|+3|J_2|+ \sigma_{p2}|V_2|+|S|+6|I_2|$ 公式(RARE-2-pass-cost)。

由前面公式(Grace-PAX)可知，两趟 Grace-PAX 的 IO 代价是 $3(|J_1+V_1|+|J_2+V_2|)+|S|$。因此，把两趟 Grace-PAX 的 IO 代价（公式(Grace-PAX)）和两趟 RARE 算法的 IO 代价（公式(RARE-2-pass-cost)）进行相减，当二者差值大于 0 时，两趟 RARE 算法的性能更好。也就是说，当以下条件成立时，两趟 RARE 算法的性能要比两趟 Grace-PAX 更好：

$(3-\sigma_{p1})|V_1|+ (3-\sigma_{p2})|V_2|>6|I_2|$。

左边的代价节省是由于：只需要访问 V_1 和 V_2 中所需要的页，只需要访问一次，而不是三次。右边是来自对连接索引的多次读取和物化的额外代价。

```
1. Read J2 and partition it (hash on join value);
2. Read J1 and partition it (same hash function);
3. Compute JI;
foreach partition p of J2 do
{
    Read partition p and build hash-table;
    Read partition of J1 and probe hash-table;
    foreach row in join result do
        Write id1,id2 in JI partition;
}
4. Merge partitions of JI on id1;
foreach join result <id1,id2> do
{
Read projected values of row id1 from V1;
/*S and I2 are both partitioned by id2*/
Write projected values into partition of S;
    Write id2 into partition of I2;
}
5. Read I2 and process it.
foreach paritition of I2 do
   foreach id2 in partition do
   {
       Read projected values of row id2 from V2;
       Write values into partition of S;
   }
```

算法[2-pass-RARE] 两趟 RARE 连接算法

8.4 基于PAX模型和连接索引的FlashJoin

Tsirogiannis 等人提出的 FlashJoin 连接算法[TsirogiannisHSWG09][GraefeHKSTW10]，也是一种专门针对闪存固态盘设计的数据库连接算法，与 RARE 类似，也都采用了连接索引和 PAX 模型，在此基础上，作者又实现了 FlashScan，这是一个扫描操作，它只会从固态盘中读取那些参与到一个查询中的属性。FlashScan 在从一个指定的行中抓取额外的属性之前，会先对谓词进行分析，因此，可以进一步减少数据读操作的数量。

FlashScan 充分利用了固态盘的较小的传输单元，然后，只需要读取那些查询所需要的

属性的迷你页。假设有一个不包含选择谓词的扫描，它要投影出关系的第 1 列和第 3 列。对于每个数据库页，FlashScan 首先会读取第一个属性的迷你页，然后寻找第三个迷你页并且读取。接下来，它会再次寻找下一个页的第一个迷你页，然后寻找第三个迷你页……。这个过程会一直持续，直到扫描完整个关系，这就会导致一个随机的访问模式。总体而言，每次寻找都会导致一个随机读操作。而闪存固态盘的随机读操作和顺序读操作一样，具有很高的性能，因此，FlashScan 可以获得很高的性能。

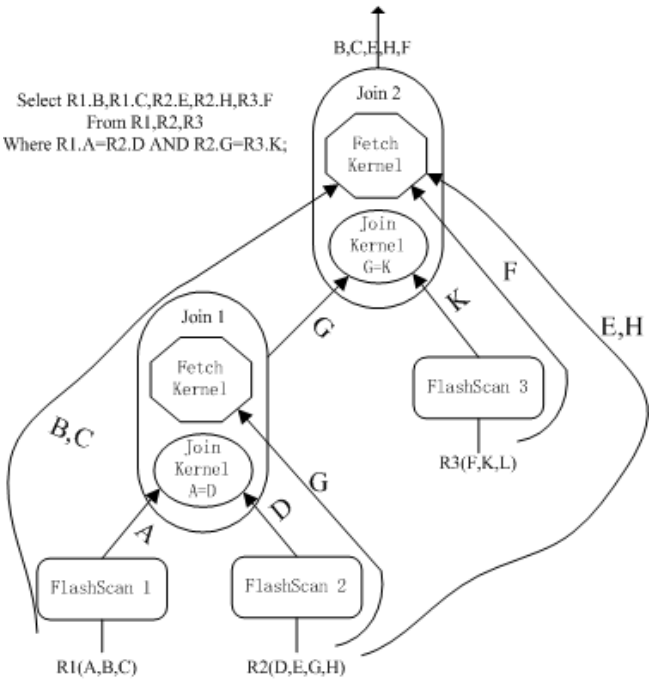
有了 FlashScan 以后，就可以高效地抽取需要的属性，在此基础上，作者进一步提出了 FlashJoin（如图[FlashJoin-example]所示），这是一个通用的、流水线结构的连接算法，通过尽量推迟检索并且只检索那些需要的属性，可以最小化对关系页的访问。FlashJoin 包括了一个二路连接（binary join）内核和一个单独的抓取（fetch）内核。多路连接会被分解成一系列流水线结构的二路连接。FlashJoin 的连接内核只会访问连接所涉及的属性，并为每个连接节点生成连接索引形式的部分结果。FlashJoin 的抓取内核，为会查询计划中的后续节点获取属性，并且会根据连接的选择性（selectivity）和可用的内存来选择不同的抓取算法。实验结果显示，FlashJoin 不仅可以明显减少内存的使用数量，还可以减少查询中每个连接的 I/O 数量。

在仔细分析了一些算法和数据结构以后，Tsirogiannis 等人还发现：

（1） 对于排序算法而言，不需要对算法做任何改变，就可以直接从固态盘中受益；

（2） 对于汇总操作而言，不会从固态盘中受益；

（3） 可以利用闪存的快速随机读，来加速关系查询处理中的选择、投影、连接操作。



图[FlashJoin-example] 采用 FlashJoin 时的三路连接实例

8.5 DigestJoin

8.5.1 概述

传统的连接算法，在不存在索引的情形下，算法的目的是为了最小化寻址操作的数量，因为，这些算法是针对磁盘设计的，磁盘存在机械部件，寻址开销很大，减少寻址操作可以有效提升连接算法的整体性能。但是，在闪存数据库系统中，最小化寻址开销并不会带来明显的收益，因为，闪存的随机读操作和顺序读操作具有同样高的性能。

为了充分利用闪存快速的随机读能力，Li 等人[LiOXCH09]提出了一种摘要连接算法

DigestJoin，假设参与连接的两个原始关系（或表）为 *R* 和 *S*，该算法把 *R* 和 *S* 的连接过程分成两个阶段来执行：

- 第一阶段：得到摘要连接结果。首先，把关系 *R* 和 *S* 的元组 ID 以及那些与连接操作相关的属性投影出来，投影得到的表称为“摘要表”，即分别得到关系 *R* 和 *S* 的摘要表 *digest_R* 和 *digest_S*。摘要表的大小要比原始表小很多，可以减少后续的 IO 开销；然后，在较小的摘要表 *digest_R* 和 *digest_S* 上运行传统的连接算法（比如嵌套循环连接和哈希连接等），来生成“摘要连接结果”。摘要连接结果只包含了来自两个连接关系的元组 ID 和连接属性，可以最小化中间连接结果。可以看出，摘要连接结果和连接索引（参见“连接索引和 Jive 连接算法”）有些类似。
- 第二阶段：构建完整连接结果。在摘要连接结果的基础上，DigestJoin 算法会从原始表中加载完整的元组来生成最终的结果。在连接索引中，这就是经典的“页抓取问题”。

DigestJoin 算法的 IO 代价收益，主要来自第一个阶段较小的摘要表和第二个阶段的随机读操作。在第一个阶段中，在较小的摘要表上进行连接可以节省很多后续的 IO 开销，尤其是可以节省连接过程中临时写入闪存的开销，由于闪存写操作往往会引起延后的擦除操作，因此，闪存写操作的减少同时也意味着需要更少的闪存擦除操作。在这个阶段，如果采用 PAX 存储模型，扫描原始表得到摘要表的过程会更加高效。采取 PAX 模型存储时，在进行连接操作时，只需要读取参与连接操作的属性（列）值，从而减少了数据的读取量，提高了读取的效率。此外，在 PAX 模型中，属于同一个属性的值都是存储在相邻的位置，因此，可以对数据进行连续读取，进一步降低了数据读取的代价。在第二个阶段，与摘要连接结果中的元组相关的、原始关系 *R* 和 *S* 中的元组，会分散在闪存的不同物理空间。因此，页抓取会引发大量随机读操作，这对于磁盘来说需要很大的 IO 开销，对于磁盘而言，这种随机读的开销甚至会超过第一个阶段节省的 IO 开销。但是，闪存具有很高的随机读性能，因此，页抓取引发的随机读操作，对于闪存而言，不会导致高昂的寻址开销。

8.5.2 算法代价分析

假设两个参与连接的关系为 $R=\{A_{r1},A_{r2},\ldots,A_{rm}\}$，$S=\{A_{s1},A_{s2},\ldots,A_{sn}\}$，其中，$A_r$ 和 A_s 分别表示关系 *R* 和 *S* 的属性，这里分别用 tid_r 和 tid_s 表示 *R* 和 *S* 的元组 ID。为了简化起见，这里讨论 $R\infty_{A_{r1}=A_{s1}}S$。在 DigestJoin 算法的第一个阶段，首先，需要扫描原始表 *R* 和 *S* 得到摘要表 R' 和 S'，这两个摘要表只包含了元组 ID 和连接属性，即 $R'=\{A_{r1},tid_r\}$ 和 $S'=\{A_{s1},tid_s\}$。摘要表要比原始表小很多，可以明显减少实际连接操作时的 IO 开销。然后，利用一个传统的连接算法，比如嵌套循环连接、归并排序连接或哈希连接，对两个摘要表进行连接，得到摘要连接结果 $\{A_{r1},tid_r,tid_s\}$。如果摘要连接结果大于内存可用空间，就可以把它们顺序写入到闪存中。但是，摘要连接结果 $\{A_{r1},tid_r,tid_s\}$ 只是告诉我们哪些元组在连接条件上发生了匹配。为了得到完整的连接结果，必须根据 *tid* 从原始表中抓取相应的元组。在 RDBMS 中，从原始表中抓取元组通常都是以页为单位，因此，DigestJoin 算法的第二个阶段被称为“页抓取”。虽然页抓取需要一定的随机读开销，但是，如果能够设计合适的页抓取策略，这种开销只是第一个阶段（即得到摘要连接结果阶段）所节省的开销的一部分，因此，DigestJoin 算法在整体上可以比传统的连接算法节省代价。

页抓取策略对 DigestJoin 第二个阶段的 IO 开销影响较大，这里以一个实例来说明这个问题。假设从关系 *R* 和 *S* 可以得到以下几个摘要连接结果：$\{r_1,tid_{r1},tid_{s1}\}$、$\{r_2,tid_{r2},tid_{s2}\}$、$\{r_3,tid_{r3},tid_{s3}\}$和$\{r_4,tid_{r4},tid_{s4}\}$。在闪存中，元组 tid_{r1} 和 tid_{r3} 被存储在页 P_1 上，元组 tid_{r2} 和 tid_{r4} 被存储在页 P_2 上，元组 tid_{s1} 和 tid_{s3} 被存储在页 P_3 上，元组 tid_{s2} 和 tid_{s4} 被存储在页 P_4 上。

当有足够的内存空间时，就可以抓取所有的 P_1,P_2,P_3,P_4 四个页，然后，在构建最终连接结果的整个过程中都把这四个页保存在内存中，在整个连接过程中，就只需要对四个页进行一次读取的 IO 开销。但是，实际上内存空间是有限的，这就需要设计合理的页抓取策略，从而最小化页的读取开销，当页抓取策略不合理时，会导致一个页刚被驱逐出内存，很快又要再次到闪存中抓取该页进入内存的情况，带来不必要的 IO 开销。这里假设内存空间只能最多同时容纳两个页，如果以 r_1,r_2,r_3,r_4 的顺序构建得到最终连接结果，那么页抓取过程如下：

（1）从摘要连接结果$\{r_1,tid_{r1},tid_{s1}\}$中得到与 r_1 对应的完整的连接结果：由于 tid_{r1} 对应的元组在页 P_1 上，tid_{s1} 对应的元组在页 P_3 上，因此，为了得到 r_1 对应的完整结果，就必须从闪存中抓取页 P_1 和 P_3 进入内存；

（2）从摘要连接结果$\{r_2,tid_{r2},tid_{s2}\}$中得到与 r_2 对应的完整的连接结果：由于 tid_{r2} 对应的元组在页 P_2 上， tid_{s2} 对应的元组在页 P_4 上，因此，为了得到 r_2 对应的完整结果，就必须从闪存中抓取页 P_2 和 P_4 进入内存；但是，由于内存中已经存在 P_1 和 P_3，而且内存只能最多同时容纳两个页，因此，需要把 P_1 和 P_3 驱逐出内存，腾出空间，把 P_2 和 P_4 放入内存；

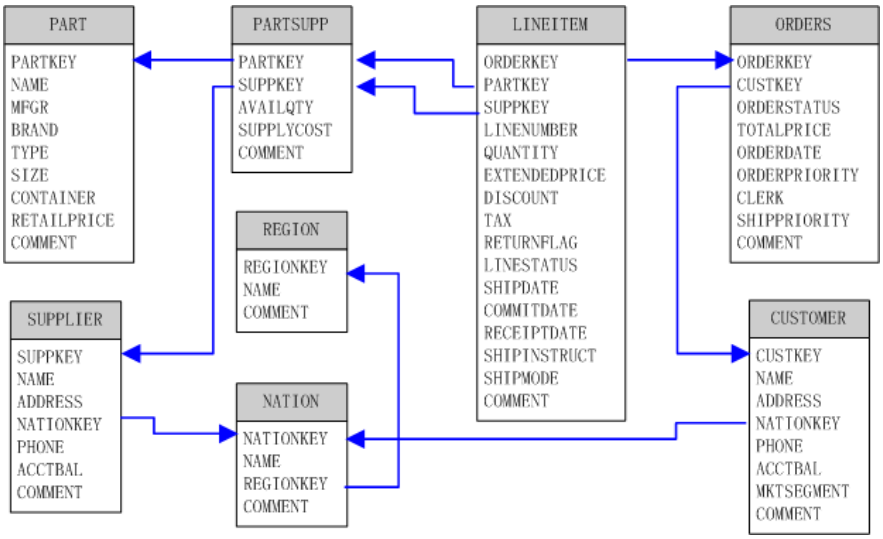
（3）从摘要连接结果$\{r_3,tid_{r3},tid_{s3}\}$中得到与 r_3 对应的完整的连接结果：和上面的过程类似，必须从闪存中抓取页 P_1 和 P_3 进入内存，因此，需要把 P_2 和 P_4 驱逐出内存，腾出空间来存放 P_1 和 P_3；

（4）从摘要连接结果$\{r_4,tid_{r4},tid_{s4}\}$中得到与 r_4 对应的完整的连接结果：和上面的过程类似，需要把 P_1 和 P_3 驱逐出内存，腾出空间来存放 P_2 和 P_4。

综上所述，在为 r_1,r_2,r_3,r_4 构建完整连接结果的整个过程中，P_1,P_2,P_3,P_4 每个页都需要被抓取两次。现在假设把 r_2 和 r_3 的构建顺序进行对调，即以 r_1,r_3,r_2,r_4 的顺序来构建完整的连接结果，那么，整个过程就会按照以下的顺序抓取页：首先抓取 P_1 和 P_3 进入内存，可以完成 r_1 和 r_3 对应的完整连接结果的构建，然后，把 P_1 和 P_3 驱逐出内存，抓取 P_2 和 P_4 进入内存，可以完成 r_2 和 r_4 对应的完整连接结果的构建。可以看出，整个过程中，P_1,P_2,P_3,P_4 每个页只需要被抓取一次，节省了 IO 开销。因此，页抓取策略对于 IO 开销影响很大。

这里给出一个基于 TPC-H 表的连接实例，来说明 DigestJoin 可以比传统的连接算法取得更好的性能，这里选择归并排序连接算法作为传统连接算法的代表。TPC-H[TPC-H]是一个决策支持测试基准，它包含了一整套面向商业的即席查询和并发数据修改。TPC-H 数据库共定义了 8 个基本表（如图[TPC-H]所示），包括 PART、PARTSUPP、LINEITEM、ORDERS、SUPPLIER、NATION 和 REGION。这里考虑在两个 TPC-H 表 CUSTOMER 和 ORDERS 上面的连接，二者是通过键 C_CUSTKEY 进行连接的：

```
SELECT *
FROM CUSTOMER, ORDERS
WHERE CUSTOMER.C_CUSTKEY=ORDERS.C_CUSTKEY;
```



图[TPC-H] TPC-H 数据库模式

根据 TPC-H 的数据库模式，一个 CUSTOMER 表的摘要表的大小，大约是 CUSTOMER 表大小的 6%，一个 ORDERS 表的摘要表的大小，大约是 ORDERS 表大小的 9%。假设 CUSTOMER 表和 ORDERS 表分别包含 10000 和 5000 个页。

首先考虑采用传统的归并排序连接算法，执行 CUSTOMER 表和 ORDERS 表的连接，并假设内存容量为 20 个页。由于归并排序连接算法包含了两个阶段，即归并排序阶段和连接阶段，因此，代价计算方法如下：

（1）归并排序阶段的代价：首先考虑较小的 ORDERS 表。在进行归并排序时，由于 ORDERS 表的大小远远大于内存容量，无法一次性放入内存，因此，需要把 ORDERS 表进行分段，把每个分段调入内存，采用内存排序算法对分段进行排序，然后把排序后的分段写入到外部存储设备中（磁盘或闪存），得到排序后的若干个初始归并段。由于内存容量为 20 个页，因此，每个分段的大小也是 20 个页，这样才能够保证把每个分段一次性放入内存进行排序。因为，ORDERS 表一共有 5000 个页，每个分段大小是 20 个页，因此，这里一共可以得到 ORDERS 表的 250 个分段。

可以看出，得到初始归并段的过程，需要把 ORDERS 表的所有未经排序的元组都读入内存一次，然后再把排序后的元组都写入到外部存储设备中一次，因此，得到初始归并段的代价是 2*5000=10000 次 IO。

得到初始归并段以后，就开始了初始归并段的合并过程。对于 ORDERS 表而言，一共有 250 个初始归并段（每个初始归并段包含 20 个页），内容容量为 20 个页，每个内存页被分配一个初始归并段使用，因此，一次只能对 20 个初始归并段进行合并，即从每 20 个初始归并段合并得到一个归并段，一共可以得到 13 个归并段（除了最后一个归并段只包含 200 个页以外，其他每个归并段都包含 400 个页），在归并的过程中，需要一边合并一边把合并结果写入到外部存储设备中。第一趟归并过程需要把 ORDERS 表的所有 250 个初始归并段的元组都读入到内存中一次，然后把合并后的归并段写入外部存储设备一次，因此，第一趟归并过程的代价是 2*5000=10000 次 IO。

然后开始执行第二趟归并，内存容量为 20 个页，每个内存页被分配一个归并段使用，由于当前只有 13 个归并段，只需要 13 个内存页用来执行归并过程，因此，可以在这一趟内完成对所有归并段的合并操作，并且一边合并一边把结果写入到外部存储设备中。第二趟归并过程需要把 ORDERS 表的所有 13 个归并段的元组都读入到内存中一次，然后合并后的归并段会被写入外部存储设备一次，因此，第二趟归并过程的代价是 2*5000=10000 次 IO。

可以看出，对于 ORDERS 表而言，从原始表得到排序后的表的代价是，得到初始归并

段的代价再加上两趟归并操作的代价，即 10000+10000+10000=30000 次 IO。同理，对于 CUSTOMER 表而言，需要一次得到初始归并段的开销和三趟归并的开销，即 4*2*10000=80000 次 IO。因此，归并排序阶段，对 CUSTOMER 表和 ORDERS 表排序的总开销是 110000 次 IO。

（2）连接阶段的代价：CUSTOMER 表和 ORDERS 表经过排序以后，就可以执行连接操作，只需要把两个表顺序地读入内存执行连接输出结果即可，因此，连接过程，两个表只需要读入内存一次，开销是 10000+5000=15000 次 IO。

综上所述，采用传统的归并排序连接算法时，CUSTOMER 表和 ORDERS 表的连接代价是归并排序阶段代价与连接阶段代价之和，即 110000+15000=125000 次 IO。

现在假设采用 DigestJoin 执行 CUSTOMER 表和 ORDERS 表的连接。在 DigestJoin 算法的第一个阶段，即得到摘要连接结果的阶段，需要首先分别从两个表投影得到各自的摘要表，然后在两个摘要表上采用传统连接算法（这里采用归并排序连接算法）进行连接。因此，DigestJoin 算法的第一个阶段的代价计算方法如下：

- 从两个表投影得到各自的摘要表的代价：从 CUSTOMER 表和 ORDERS 表投影得到的摘要表的大小分别是 600（即 10000*6%）页和 450（即 5000*9%）页，构建这两个摘要表时，需要对两个表分别顺序扫描读入内存一次，然后选择出摘要表需要的属性，写入到外部存储设备一次，因此，构建两个摘要表的代价是 10000+5000+600+450=16050 次 IO。
- 在两个摘要表上采用传统的归并排序连接算法进行连接的代价：这里包括归并排序的代价和连接的代价。（1）归并排序的代价：在得到两个摘要表的初始归并段以后，两个摘要表都可以在两趟（得到初始归并段的一趟不算在内）内完成归并，因此，归并排序的代价是 3*2*600+3*2*450=6300；（2）连接的代价：在连接阶段，需要把两个摘要表读入内存一次来执行连接，因此，连接阶段的代价是 600+450=1050 次 IO。综上所述，使用 DigestJoin 算法执行两个摘要表的连接的总代价是：6300+1050=7350 次 IO。

因此，DigestJoin 算法的第一个阶段（即生成摘要连接结果）的总共代价是 16050+7350=23400 次 IO。因此，只要 DigestJoin 算法的第二个阶段（构建完整连接结果）的 IO 少于 101600（即 125000-23400）次，DigestJoin 算法的性能就要好于传统的归并排序连接算法。

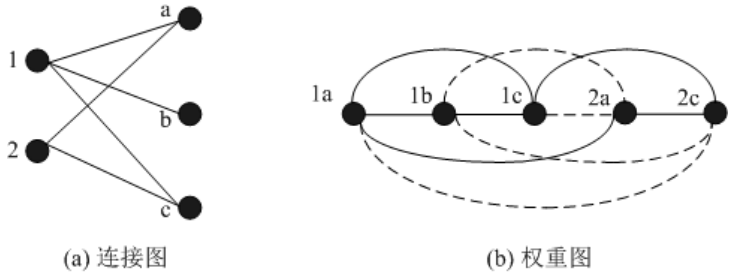
8.5.3 页抓取问题复杂度分析

从上面的论述可以看出，DigestJoin 算法的第二个阶段（即构建完整连接结果阶段）的代价，直接关系到 DigestJoin 算法是否可以比传统连接算法获得更好的性能。DigestJoin 算法的第二个阶段的核心问题就是页抓取问题，即如何合理调度页抓取顺序，从而用最小数量的页访问，就可以读取连接结果中涉及到的所有元组。

为了更好地分析页抓取问题，这里把该问题建模为图问题。一个连接图，表示了连接结果中所涉及的页之间的关系。连接图被定义为一个无向的二分图 $G=(V_1 \cup V_2, E)$，其中，V_1 和 V_2 分别表示来自两个原始表的页的集合，V_1 内部各顶点之间不存在边，V_2 内部各顶点之间也不存在边，只有 V_1 中的顶点和 V_2 中的顶点这二者之间才会存在边。$E \subseteq V_1 \times V_2$ 表示了连接结果所涉及的“页对”的集合。对于每条边 $(v_a,v_b) \in E$，表示存在一个属于页 v_a 上的元组，它可以和属于页 v_b 上的元组在连接属性上发生匹配。这个连接图可以用来动态地表示剩余需要抓取和连接的页。一条边 (v_a,v_b) 会被从 E 中删除，如果 v_a 和 v_b 已经被抓取到内存，并且相应的元组已经完成连接。一个顶点 v 会被从 G 中删除，如果该顶点的出度变为 0。

图[DigestJoin](a)给出了一个连接图的实例，其中，顶点 1 和 2 代表了来自同一个表的页，而顶点 a,b,c 则代表了来自其他表的页。边 $(1,a)$ 表示页 1 上面有个元组，可以和页 a 上

面的元组发生连接。



(a) 连接图 (b) 权重图

图[DigestJoin] DigestJoin 算法的连接图和转换后的权重图

一个页抓取的过程，就等价于把所有的边都从连接图中移除的过程。正如之前描述的那样，一条边会被移除，当且仅当相应的页会被抓取到内存（从而构建相应的最终连接结果）。因此，一个最优的页抓取顺序，就是一个从图中删除所有边的顺序，并且要具有最少的页访问次数。

下面将说明页抓取问题是一个 NP 问题，为了阐述该问题的复杂度，这里将采用下面的思路：

- 首先按照一定的方法，从连接图转换得到一个带有边权重的权重图；
- 然后将说明，页抓取问题就相当于在权重图中采用最短路径遍历所有顶点的问题，该遍历问题已经被证明是一个 NP 问题，因此可以判定页抓取问题也是一个 NP 问题。

首先介绍如何从连接图中转换得到另外一个权重图。对于一个连接图 $G=(V_1 \cup V_2, E)$，可以为之构建一个相应的包含边权重的图 $G'=(V', E')$。也就是说，连接图中的每条边，都会被转换成权重图中的一个顶点。因此，权重图中的 v'，就代表了需要把 v_a 和 v_b 抓取到内存中用来构建得到最终的连接结果。权重图中的每条边的权重，是两个连接操作之间在内存中交换页的代价。在内存中交换页是指，第一个连接操作结束后，进行第二个连接操作时，由于两个连接操作所需要的页可能不同，如果内存容量有限，就必须驱逐一些与第一个连接操作相关的页，把与第二个连接操作相关的页抓取到内存中，这就涉及内存中数据页的交换。图[DigestJoin](b)显示了图[DigestJoin](a)中的连接图对应的权重图。图[DigestJoin](b)中的顶点名称，都是用图[DigestJoin](a)中相应的边的名称来表示的，比如，图[DigestJoin](b)中的顶点 1*a* 表示图[DigestJoin](a)中的边（1,*a*）。现在讨论边的权重，假设内存只能最多同时容纳两个页，则边（1*a*,1*b*）的权重是 1，因为，边(1*a*,1*b*)涉及了两个连接操作，第一个连接操作需要抓取页 1 和 *a*，第二个连接操作需要抓取页 1 和 *b*。当执行完第一个连接操作以后，内存中包含了页 1 和 *a*，在执行第二个连接操作时，需要页 1 和 *b*，1 已经在内存中，*b* 不在内存中，需要把一个新页 *b* 调入内存，但是，由于内存最多只能同时容纳 2 个页，因此，需要把页 *a* 驱逐出内存，把页 *b* 放入内存，发生了 1 个页的交换，边(1*a*,1*b*)的权重就是表示两个连接操作之间在内存中交换页的代价，因此，边(1*a*,1*b*)的权重就是 1。同理，边(1*b*, 2*a*)的权重是 2，因为，执行第一个连接操作需要页 1 和 *b*，在执行第二个连接操作时，需要把 1 和 *b* 驱逐出内存，抓取两个新页 2 和 *a* 进入内存，内存中页交换的数目是 2。

接下来将说明页抓取问题是个 NP 问题。通过上面这种转换以后，页抓取的顺序就等价于在权重图中对各个顶点遍历一遍。遍历过程所需要抓取的页的总数，就是遍历路径上各条边的权重之和。因此，页抓取问题就被转换成“在一个图中寻找一个代价最小的路径的问题”，即哈密顿路径问题。Merrett 等人[MerrettKY81]已经证明，在权重图中寻找一个哈密顿路径是一个 NP 问题。因此，可以判定页抓取问题也是一个 NP 问题。

8.5.4 解决页抓取问题的启发式算法

上面已经证明了，页抓取问题是一个 NP 问题。对于 NP 问题，通常可以采用启发式算

法得到近似最优解。如果内存可以容纳所有的以连接图形式表示的摘要连接结果，就可以找到一种比较好的启发式方法来遍历图中的所有边。启发式算法[ChanO97][Omiecinski89]的基本思想是：

首先，选择一个关于某个顶点的子图，这个子图包含了该顶点所有相邻的边，并且需要最少地抓取页。换句话说，可以在当前的连接图中检查所有的子图，并明确两个方面的内容：（1）确定是否每个子图都包含了这个子图中的所有顶点的所有边；（2）计算有多少个顶点不在内存中。

然后，满足第一个条件并且第二个条件的值最小的子图，会被选中。由于对一个连接图的所有子图进行枚举的代价很高，因此，一种近似的方法是，选择一个具有最少的“非驻留内存邻居”（即邻居顶点不在内存中）的顶点，并且它的所有邻居也会被一起选中。这种具有最少的“非驻留内存邻居”的顶点和它的所有邻居，一起构成一个“分段”。例如，在图[DigestJoin](a)的连接图中，对于顶点 1 而言，它有三个邻居 *a*,*b*,*c*，因此，顶点 1 对应的候选分段是{1,*a*,*b*,*c*}，同理可以得到其他顶点对应的候选分段，因此，图[DigestJoin](a)中的候选分段包括：{1,*a*,*b*,*c*}、{2,*a*,*c*}、{*a*,1,2}、{*b*,1}和{*c*,1,2}。如果内存中还没有抓取放入任何页，此时，顶点 1 对应的分段{1,*a*,*b*,*c*}的非驻留内存邻居数目是 3，即 *a*,*b*,*c*，顶点 2 对应的分段{2,*a*,*c*}的非驻留内存邻居数目是 2，即 *a*,*c*，同理，分段{*a*,1,2}、{*b*,1}和{*c*,1,2}的非驻留内存邻居的数量分别是 2,1,2。可以看出，应该选择顶点 *b* 调入内存，因为，顶点 *b* 对应的分段{*b*,1}具有最少的非驻留内存邻居数目（只有页 1 不在内存中），因此需要最少的页访问。现在假设一些页已经被抓取到内存中，比如 *c*，那么应该选择顶点 2 或 *b*，因为，这时顶点 2 和 *b* 都只有 1 个非驻留内存邻居，即分别是 *a* 和 1。

通过上述方法，就可以每次选中相应的顶点，并把其对应的闪存页抓取入内存，完成连接结果构建。

上面描述的是内存可以容纳连接图的情形，但是，往往在实际应用中，连接图都比较大，无法一次性放入内存，这时，就需要更加复杂的处理机制，关于这个问题，可以参考 Li 等人[LiOXCH09]的解决方案，这里不再细述。

8.6 本章小结

本章内容首先介绍了 NSM、DSM 和 PAX 页布局模型，分别描述了各种模型的存储原理，并比较了三种模型的特性；然后，介绍了连接索引和 Jive 连接算法，许多面向闪存的查询连接操作优化算法都借鉴了这两者的思想；接下来，介绍了基于 PAX 的 RARE 连接算法以及基于 PAX 模型和连接索引的 FlashJoin 方法；最后，介绍了一种摘要连接算法 DigestJoin，该算法把 *R* 和 *S* 的连接过程分成两个阶段来执行，第一阶段得到摘要连接结果，第二阶段构建完整连接结果，DigestJoin 算法的 IO 代价收益，主要来自第一个阶段较小的摘要表和第二个阶段的随机读操作。

8.7 习题

1、 阐述 PAX、NSM 和 DSM 三种页布局模型的工作原理，并比较三者的特性。

2、 分析为什么 PAX 模型可以在闪存上发挥更好的性能。

3、 说明什么是连接索引及其作用。

4、 分析单趟 RARE 连接算法的代价。

5、 分析 DigestJoin 算法的主要 IO 代价收益来自那些地方。

第9章 闪存数据库事务管理

事务处理是 DBMS 的重要组成部分，事务机制可以保证数据库数据处理的原子性、一致性、隔离性和持久性，从而推动了数据库在商业领域的大规模成功应用。Jim Gray 正是因为提出了著名的事务理论，从根本上解决了实际应用中的数据一致性和完整性问题，因而获得了 1998 年的图灵奖。不少研究人员也在事务方面的研究做出了巨大的努力，使得事务理论在不同的环境下（比如分布式数据库）不断得到完善。

事务恢复是数据库管理系统的重要功能，可以保证数据库的正确性和一致性。数据库在实际运行过程中，会不可避免地发生各种故障，比如操作系统故障、内存故障、CPU 故障、外部设备故障、操作失误和断电等等，必须采取有效的措施才能够保证数据库的一致性和完整性。比较常见的事务恢复方法就是建立日志文件和建立镜像文件，生成原始数据的冗余备份，一旦发生故障，系统就可以从日志或镜像文件中恢复数据。

在绝大多数传统的面向磁盘的 DBMS 中，通常采用日志的方式进行事务恢复，当数据库发生比较频繁的更新时，会产生大量的小数据量的日志写入操作。磁盘可以以字节为单位进行写入，而闪存的最小写入单位是页，每次写入少量日志记录时，无法占满一页空间，下次写入新的日志记录时，需要再分配新的空白页进行日志写入。因此，频繁的小数据量的日志写入操作，不仅会浪费大量的闪存空间，也恶化了闪存的整体性能。此外，传统的 DBMS 都采用三阶段故障恢复方法，往往包含了大量的数据读取和反复更新操作，当采用闪存作为数据库的底层存储介质时，会导致大量的写操作和擦除操作，进一步恶化了数据库的性能。

综上所述，如果把现有的面向磁盘的 DBMS 的事务恢复方法直接应用到闪存数据库中，将无法获得好的数据库性能。

本章内容首先讨论了数据库事务恢复的概念和常用技术，介绍了两种应用最为广泛的传统数据库事务恢复方法，即基于日志技术的事务恢复方法和基于影子页技术的事务恢复方法；然后，分析了传统的数据库事务恢复方法在闪存数据库中的表现情况；最后，重点介绍了几种典型的闪存数据库事务恢复方法，包括基于日志技术的闪存数据库事务恢复方法，日志恢复技术与影子页恢复技术相结合的闪存数据库事务恢复方法。在介绍不同的闪存数据库事务恢复方法的同时，还分析了不同的事务恢复方法的优缺点。

9.1 事务的概念

数据库事务是数据库管理系统执行过程中的一个逻辑单元，是由有限个操作构成的一个操作序列。事务处理用以保证事务的顺利执行，即保证事务的原子性、一致性、隔离性和持久性。事务的原子性是指一个事务中的所有操作要么全部执行，要么全部都不执行；事务的一致性是指，如果在事务执行之前数据库是一致的，那么在执行事务之后数据库还是一致的；事务的隔离性是指即使多个事务并发执行，每个事务都感觉不到系统中有其他事务在执行；事务的持久性是指事务成功执行以后，它对数据库的修改是永久的，即使系统出现故障，数据库中的数据也不受影响。

事务恢复用以保证事务的原子性和持久性，是提高数据库可靠性的重要技术之一。事务恢复是指在系统发生故障或人工中止事务时，能够回滚到事务执行之前的状态，以此来确保事务的原子性与一致性。为了在系统发生故障或中止事务时，事务管理能够回滚到事务执行之前的状态，事务管理器需要跟踪事务的状态，以判断事务是否已经执行完毕，对于已经提交的事务，不用作任何特殊处理，对于没有执行完毕的事务，需要撤销中止事务所做的更改。

9.2 传统的事务恢复机制

在基于磁盘的数据库系统中有两种常用的事务恢复方法，即基于日志的恢复方法（write-ahead-logging，简写为 WAL）和基于影子页技术（shadow paging）的恢复方法。

9.2.1 基于日志的恢复方法

基于日志的恢复方法要求在磁盘上的数据被更新之前先写入其更新日志，事务提交时先写入该事务的更新日志，再写入事务的提交日志。事务恢复时，通过是否存在事务提交日志来判断该事务是否已经提交。虽然基于日志的恢复方法中日志的写入较为频繁，但是，由于数据库中日志一般连续存储在一个独立区域，日志写入时磁盘几乎不需要寻道和旋转，因此，记录日志对系统性能的影响不大；另一方面，由于磁盘可以原位更新且更新操作代价和读取操作相比并不突出，因而故障恢复时 Undo 和 Redo 操作的代价不大。

9.2.2 基于影子页的恢复方法

基于影子页的恢复方法采用了与基于日志的恢复方法完全不同的方式处理数据的更新。在使用影子页技术作为事务恢复的数据库管理系统中，需要维护一个逻辑页与物理页之间的地址映射。当一个事务需要修改逻辑页时，系统将分配一个新的空闲页作为该物理页的影子页，将修改后的数据保存在影子页中，并在地址映射表中，将逻辑页映射到影子页上。提交事务时，将地址映射表写入到永久的存储设备中，并在之后回收无用的物理页。中止事务时，只需要简单地抛弃影子页和地址映射表的更新。相对于基于日志的恢复方法，基于影子页技术的恢复方法需要更少的写操作。

在使用磁盘作为外部存储设备的数据库系统中，基于影子页的恢复方法没有基于日志的恢复方法应用广泛，这是因为：1）影子页技术需要维护地址映射表；2）影子页技术采用异地更新的方式，而各个影子页分散在磁盘的各个位置，加大了磁盘的寻道开销；3）影子页技术会产生很多垃圾页，需要专门的垃圾回收机制回收这些垃圾页。

9.3 传统的事务恢复机制在闪存数据库中的表现

虽然在传统的数据库系统中，基于日志的恢复方法比基于影子页的恢复方法应用广泛，但是，在闪存数据库系统中，情况却正好相反，这是由闪存的内在特性决定的。因为闪存的读写代价不一致、异地更新和读写操作只能以页为单位等特性，WAL 方法在闪存中会带来新的问题，而影子页的恢复方法中存在的问题却能迎刃而解。

传统的日志恢复方法并不适合直接应用在闪存上。一方面，闪存的写操作代价远高于读代价，频繁写入日志会严重影响系统性能，故障恢复时 Undo 和 Redo 操作的代价也较高；另一方面，事务提交日志通常需要及时写入，提交日志的长度一般很小，而闪存的读写操作是以页为单位，及时写入提交日志的代价相对较高，而且会降低日志空间的利用率。

闪存是使用影子页技术恢复方法的理想设备，这是因为：第一，影子页技术需要维护地址映射表，而闪存本身就需要维护地址映射表，所以，维护地址映射表的代价可以忽略不计；第二，影子页技术采用异地更新的方式，而闪存支持快速的随机访问，不存在寻道开销，能够显著提高影子页恢复方法的效率；第三，影子页恢复方法会产生很多垃圾页，需要垃圾回收机制回收这些垃圾页，而垃圾回收是闪存责无旁贷的事情。综上所述，闪存数据库系统更适合采用影子页方法来处理事务恢复。

9.4 代表性方法

目前，相关研究已经提出了一些面向闪存文件系统或闪存数据库的事务恢复方法，这些方法大多借鉴了影子页的思想，利用影子页技术在闪存设备中的天然优势，设计面向闪存的事务恢复方法，以此提高故障恢复的性能。与此同时，有部分研究针对数据库中存在的随机少量写操作提出了基于日志提交的存储管理办法，这类存储管理办法更适合采用基于日志提交的事务恢复方法。

基于影子页的事务恢复中，代表性方法有 Transactional FTL、TxFlash 和 Flag Commit。其中，Transactional FTL 是一种针对嵌入式数据库系统而设计的事务恢复方法，TxFlash 是针对文件系统或数据库系统，设计了一种通用的事务处理机制。在 Transactional FTL 和

TxFlash 中，事务处理由底层存储负责，底层存储提供了与事务相关的接口，上层应用通过调用这些接口以实现事务操作。Flag Commit 使用了 SLC(Single Level Cell)型闪存支持部分页编程（partial page programming）技术的特点，在物理页的带外区(spare area)中存储事务提交标识，通过带外区中的事务提交标识跟踪事务的状态，判断一个该事务是否已经提交。

基于日志的事务恢复方法中，代表性方法有 IPL(In-Page Logging)和 OPL(Out-Page Logging)。IPL 与 OPL 使用日志技术的存储管理办法，虽然它们记录日志的主要目的是进行数据更新，但由于这些日志被存储在非易失的闪存中，故同样可以用于事务恢复。IPL 与 OPL 采用采用了相同的方法处理事务，但是，IPL 需要底层存储设备提供部分页编程支持，OPL 在提高日志空间利用率的同时，不需要部分页编程技术的支持，所以 OPL 也可以应用于不支持部分页编程的固态盘，因此，相对于 IPL，OPL 的可用范围更加广泛。

此外，HV-Recovery 通过使用一种全新的日志记录方法，充分利用了闪存设备异地更新产生的逻辑页的历史版本，显著地提高了数据库恢复操作的性能。因为 HV-Recovery 需要写入日志，并且利用了闪存异地更新产生的逻辑页历史版本，所以，HV-Recovery 可以看作是一种日志事务恢复方法和影子页事务恢复方法相结合的恢复方法。

9.4.1 Transactional FTL

文献[KimLJB08][LeeKWCK09]提出了一种新的 FTL(Flash Transaction Layer)，即 Transactional FTL，适用于使用 MLC(Multilevel Cell Layer) NAND 型闪存作为外部存储设备的嵌入式数据库系统。Transactional FTL 在基本的 FTL 基础之上，提供了 3 个新的 API 以支持事务操作，这 3 个 API 分别是 FTL_BeginTxn()，FTL_CommitTxn()和 FTL_AbortTxn()。FTL_BeginTxn()返回一个事务 ID，这个事务 ID 将作为 FTL_CommitTxn() 和 FTL_AbortTxn()调用的参数，FTL_CommitTxn()能够原子地执行事务。当系统非正常关闭或调用 FTL_AbortTxn()时，Transactional FTL 将自动撤销中止事务所做的修改，回滚到事务执行之前的状态。有了以上 3 个 API 以后，数据库系统不再需要使用日志或影子页技术，就能够方便地实现数据库级别的事务操作。

Transactional FTL 借鉴了影子页的思想，为逻辑页的更新创建一个影子页，将更新后的数据写入到影子页中，事务提交时，在专门区域写入一条提交日志，通过是否存在提交日志来判断事务是否已经提交。该方法避免了频繁的小数据量日志写操作，从而大大减少了由此引起的 MLC NAND 闪存开销。在基于影子页的方法中，物理页的分配是基于事务的。当一个事务开始时，一个物理块被分配给这个事务，然后，该事务所有后续的写请求都会被导向到这个块中。这样，来自某个事务的逻辑上随机的写操作，会被转换成在一个块内物理上连续的写操作。这种做法和之前的其他方法具有很大的区别。在其他方法中，针对某个逻辑页的更新操作，不管是属于哪个事务，都会被引导到同一个物理页上面去执行，因此，会带来“耦合页”问题，即属于某个事务的写操作在执行过程中发生断电，会破坏同一个物理页中属于另一个事务的写操作已经写入的数据。而对于基于影子页的方法而言，就不存在耦合页问题，因为，所有不同的事务都写入不同的物理块中。某个事务所写入的块，和其他事务所写入的块没有任何关系。因而，一个事务的写操作在发生断电时，自然也就不会对其他事务写入的数据产生任何影响。

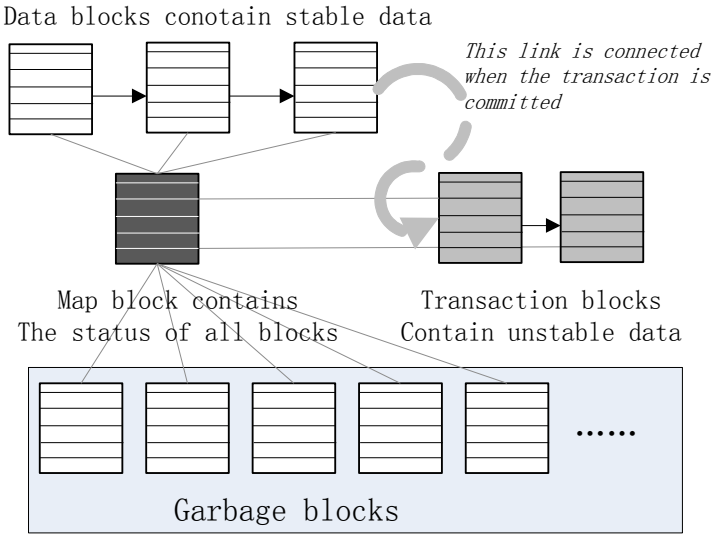
基于影子页的方法可以提供一致的写操作性能，即写操作性能和写操作的模式无关，因为，所有属于同一个事务的写操作都是在分配给该事务的块内按照先后顺序进行的。

下面简单介绍该方法的技术细节。如图[TransitionalFTL]所示，在该方法中，闪存中的块被分成四种类型：数据块、事务块、映射块和垃圾块。四种类型的块的具体作用如下：

- **数据块**以列表的方式组织在一起，称为数据块列表，每次新创建的数据块都会被追加到列表的尾部。一个数据块只包含已经提交的数据，这就意味着，数据块中的数据代表了数据库在某个时刻的一致性状态。一个数据块包含若干个数据页，每个数

据页包含数据区域和带外区，带外区域中存储了该物理页对应的逻辑页的编号 LPN。往数据块中的写入一个新的页 p_{new} 时，如果该新页的 LPN 与已经存在的某个页 p_{old} 的 LPN 相同，那么，p_{old} 会被设置为“无效”，从而使得系统中永远只有最新版本的数据页是处于“有效”状态的。

- **事务块**包含了未提交的数据。当一个事务正在执行时，系统会为它分配一个事务块，该事务的写操作会在这个新分配的事务块上进行，而不是在数据块上进行。事务提交后，所有分配给该事务的事务块，都会变成数据块，并且追加到数据块列表的尾部。如果一个事务中止了，所有分配给该事务的块，都会变成垃圾块。
- **垃圾块**只包含处于“失效”或“空闲”状态的页。当系统中不再有空闲块可以使用时，就会启动垃圾回收过程，回收后，就可以继续用作事务块、映射块或者数据块。
- **映射块**是专门用来存放块映射表的，块映射表是块状态记录的集合，包含了闪存中每个块的当前状态，当一个事务提交或者中止时，块映射表就会被更新。



图[TransitionalFTL] Transitional FTL 中块的组织方式

Transactional FTL 为每一个更新事务分配一个“空闲”状态的垃圾块作为其事务块，事务提交时，将其更新过的所有逻辑页写入其事务块，然后在闪存的映射块中写入一条提交日志，将事务块转换为数据块。

如果在事务执行过程中系统发生故障，则需要重建闪存的地址映射表，在重建闪存的地址映射表时，直接忽略事务块，只考虑数据块，从而将逻辑页恢复到最近的已提交版本。Transactional FTL 在故障恢复时不需要执行 Undo 和 Redo 操作，但仍然需要写入单独的提交日志，当系统中短事务较多时，提交代价仍然较大，且闪存空间利用率较低。

9.4.2 TxFlash

文献[PrabhakaranRZ08]通过修改现有的固态盘，提出了一种新的、支持事务操作的存储设备——Transactional Flash，简称 TxFlash。TxFlash 提供了一个支持原子地读写多个页的 API，即 WriteAtomic(p_1……p_n)，有了这个 API 以后，上层应用不再需要使用日志或影子页技术，就能够方便地实现数据库级别的事务操作。

TxFlash 的原理如图[TxFlash]所示。TxFlash 与普通的闪存设备一样，需要 FTL、垃圾回收机制和地址映射表等信息，但是，TxFlash 在普通的闪存设备基础上，提供了一个支持原子地操作多个页的 API（WriteAtomic），因此，文件系统与数据库系统可以通过调用该 API，方便地实现事务操作。TxFlash 为了提供事务支持，增加了一个事务提交逻辑（Commit Logic）

和一个事务恢复逻辑（Recovery Logic），并且提供了事务的隔离性保证，确保没有冲突的写一个页。一旦事务提交，TxFlash 将更新与该事务相关的所有页的地址映射信息。

TxFlash 的核心就是事务的提交逻辑与事务的恢复逻辑。事务的提交逻辑用于跟踪事务的状态；事务的恢复逻辑用于在系统发生故障时，判断系统中哪些页是已经提交的，哪些页是未提交的，在系统恢复过程中，重建已提交页的地址映射，忽略未提交的页，以此保证事务的原子性与一致性。

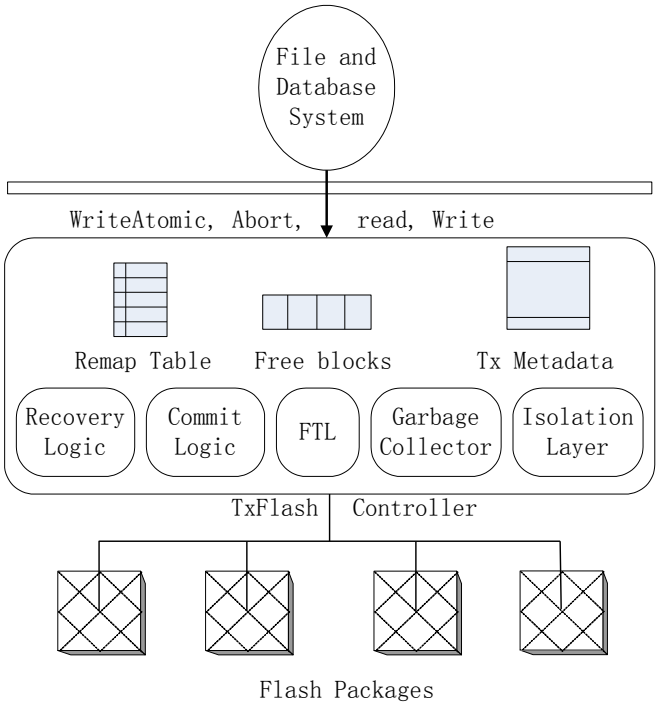
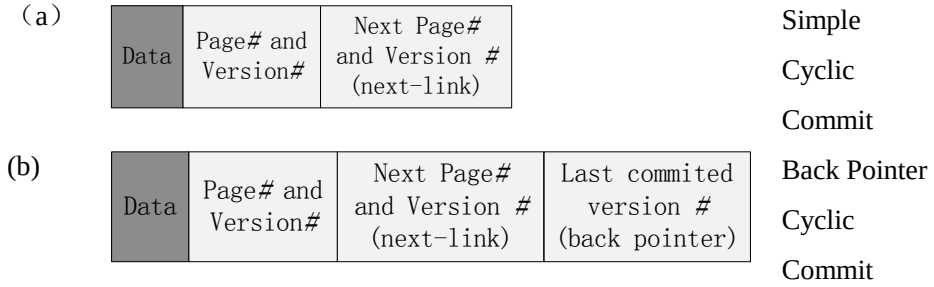


Fig. [TxFlash] Architecture of a TxFlash Device

图[TxFlash] TxFlash 设备的体系架构

TxFlash 采用了两种事务提交逻辑，即 SCC(Simple Cyclic Commit)算法和 BPCC(Back Pointer Cyclic Commit)算法，这两种事务提交算法的思想基本是一样的。下面首先介绍 SCC 算法及其不足，然后再介绍 BPCC 算法。

SCC 借鉴了影子页的思想，为每一个要修改的逻辑页创建一个影子页（影子页的数据结构如图[SCC-BPCC] (a)所示），将修改过的数据存在影子页中，在影子页的带外区中存放下一个影子页的物理地址(事务预先申请少量闪存页，从而可以在影子页被写入前就事先确定其在闪存上的位置)；事务提交时，在最后一个影子页的带外区中写入第一个影子页的物理地址，从而使得属于同一个事务的各影子页形成了一个环。通过是否存在环来判断事务是否已经提交，而不用写入事务提交日志，减少了日志的数量，因此，提高了事务提交的效率。



图[SCC-BPCC] SCC 与 BPCC 算法中物理页格式

SCC 算法中有一个重要的特性，即逻辑页的非最新版本一定是已提交的。也就是说，当要创建逻辑页的影子页时，必须确保闪存中没有中止事务所留下的未提交版本。SCC 的

这个特性对事务恢复以及写操作的性能有较大的影响。

当系统发生故障时，需要重新建立地址映射表，建立地址映射表的关键就是判断哪些页是已提交的（有效的），哪些页是未提交的（无效的）。在 SCC 中，逻辑页的非最新版本一定是已提交的，所以，事务恢复的关键就是将逻辑页映射到具有最高版本号的物理页或映射到拥有第二高版本号的物理页。如果拥有最高版本号的物理页是已提交的，则将逻辑页映射到该物理页；如果拥有最高版本号的物理页是未提交的，则说明该事务没有执行完毕，需要回滚到事务执行之前的状态，SCC 只需要简单地将逻辑页映射到具有第二高版本号的物理页即可。综上所述，SCC 中事务恢复的关键就是判断拥有最高版本号的物理页是否是已提交的。

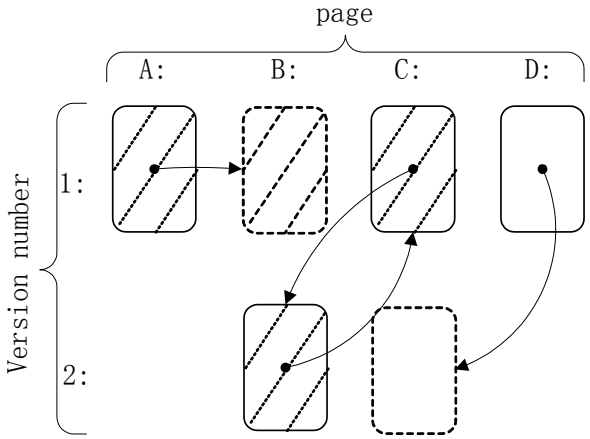
下面通过一个例子来说明 SCC 算法中的事务恢复。如图[SCC-example]所示，A_1、B_2、C_1 是普通的已提交的物理页，B_1 是一个已经过时的物理页，C_2 是一个不存在的页，D_1 是一个已写入了数据、但其所在的事务还未提交的页，C_2 是与 D_1 相关的事务下一次要写入的位置。令 h_p 为 P 的最高版本号，P_h 为拥有最高版本号的页，Q_l 为 $P_h.next$，l 为 Q_l 的版本号，h_Q 为拥有最高版本号的页。

拥有最高版本号的页 P_h 是否是已提交版本，需要分三种不同的情况讨论。

第一种情况($h_Q > l$)： h_Q 大于 l 说明对于 Q_l 而言，有更新的提交版本 Q_h，因此，P_h 是已提交的。这是因为 P_h 与 Q_l 在一个环中，一个环中的页，要么都是已提交的，要么都是未提交的。如图 4 所示，A_1 的 *next-link* 域指向 B_1，而 B 的最新版本为 B_2，在 SCC 中，非最新版本的页一定是已提交的，所以 B_1 是已提交的，而 A_1 的 *next-link* 域指向 B_1，所以 A_1 也是已提交的。

第二种情况（$h_Q < l$）： 因为 l 比 Q 的最高版本号 h_Q 还大，所以，Q_l 是一个不存在的页，说明此次事务还未执行完，因此，P_h 是未提交的。考虑图[SCC-example]中的 D_1 与 C_2，D_1 的 *next-link* 域指向 C_2，而 C 的最新版本为 C_1，所以，D_1 是未提交的。

第三种情况（$h_Q = l$）： P_h 的下一个影子页是拥有最高版本号的页，P_h 是否已提交取决于 Q_h 是否已提交。通过递归地判断 Q_h 是否已提交来决定 P_h 是否已提交。考虑图 4 中的 C_1 与 B_2，C_1 的 *next-link* 域指向 B_2，所以，C_1 是否已提交取决于 B_2 是否已提交，而 B_2 的 *next-link* 域又指向了 C_1，B_2 与 C_1 形成了一个环，根据 SCC 算法的定义，B_2 与 C_1 都是已提交的页。



图[SCC-example] SCC 算法系统状态的一个实例

SCC 算法实现相对比较容易，只需要简单地修改固态盘的垃圾回收机制和恢复机制就能实现 SCC 算法。但是，SCC 算法要求所有非最新版本的页都是已提交的，也就是说，在一个事务写一个页时，必须擦除未提交的版本。这个要求增加了写操作的延迟，因此，TxFlash 还可以采用另一种事务恢复算法 BPCC。

BPCC 的数据结构如图[SCC-BPCC] (b)所示。BPCC 在 SCC 算法的基础上，增加了一个

back-pointer 域，用于指向逻辑页的上一个已提交版本，以此来跳过未提交的版本。BPCC 算法的事务提交与 SCC 算法一样，通过判断属于同一个事务的影子页是否形成环来判断事务是否已经提交。BPCC 的事务恢复也与 SCC 类似，BPCC 使用与 SCC 一样的方法来判断拥有最高版本号的页是否已提交。如果拥有最高版本号的页是已提交的，则将逻辑页的地址映射到拥有最高版本号的页；如果拥有最高版本号的页是未提交的，则将逻辑页的地址映射到该页的上一个已提交版本。BPCC 算法可以通过 *back-pointer* 获得逻辑页的上一个已提交版本，然而，在 SCC 算法中，逻辑页的非最新版本一定是已提交的，所以，如果拥有最高版本号的页是未提交的，则将逻辑页地址映射到拥有第二高版本号的物理页，这就是 BPCC 算法与 SCC 算法的主要区别。

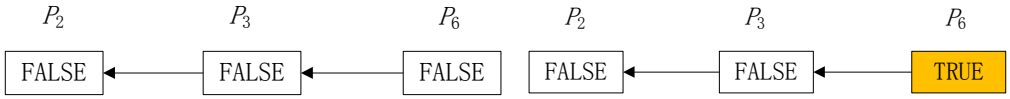
TxFlash 向上层应用提供了一个支持原子地操作多个页的 API，上层应用可以通过调用该 API 方便地实现事务操作。但是，TxFlash 存在如下问题：1）为了在影子页的带外区中存放下一个影子页的物理地址，需要预先申请少量闪存页，从而可以在影子页被写入前就确定将来其在闪存上的位置；2）为了保证数据的一致性，SCC 算法在写入逻辑页的新版本前，需要擦除未提交的版本，增加了写操作的延迟，虽然 BPCC 算法通过在带外区中存储更多的信息解决了这个问题，但是，BPCC 算法也增加了事务恢复与垃圾回收的复杂性；3）SCC 与 BPCC 算法都由 TxFlash 实现，支持文件系统或数据库系统级别的事务处理，没有专门针对数据库系统进行优化，例如，没有考虑并发控制，而并发控制是数据库系统中的关键技术。

9.4.3 Flag Commit

文献[OnXCHH12]是第一篇专门研究闪存数据库事务恢复的文章，作者提出了应用于不同工作场景的两个事务恢复方法，分别是 CFC（Commit-based Flag Commit）和 AFC(Abort-based Flag Commit)。它们的基本思想都是使用影子页(shadow paging)技术实现事务的更新操作，当一个事务需要修改逻辑页时，系统将分配一个新的空闲页作为该逻辑页的影子页，将修改后的数据保存在影子页中，并在影子页的带外区保存逻辑页号、事务的 ID 和事务的提交标识等信息。提交事务或中止事务时，通过影子页的带外区中事务提交标识，就可以跟踪事务的状态。在事务的更新、中止、提交过程中，都没有写入任何日志，减少了日志的数量，因此也减少了写操作和擦除操作的次数，最后提高了事务恢复的整体性能。

CFC 和 AFC 算法之所以能够利用影子页的带外区来跟踪事务的状态，是因为它们利用了 SLC 型闪存具有部分页编程的特征。所谓部分页编程，是指在不擦除物理页的情况下，修改物理页的带外区中的数据，即原地修改了物理页的带外区中的数据，从而实现了原地更新事务的状态。

在 CFC 与 AFC 算法中，事务的状态由属于这个事务的所有影子页的事务提交标识共同决定。CFC 算法约定，当且仅当至少有一个影子页的事务提交标识为 *TRUE*，这个事务才是已提交的，否则，该事务是未提交的。在 CFC 算法执行过程中，将所有影子页初始时的事务提交标识都设置为 *FALSE*，事务提交时，将最后一个影子页的事务提交标识设置为 *TRUE*。一个事务可能包含很多影子页，在最后一个影子页的事务提交标识由 *FALSE* 改为 *TRUE* 后，就标识该事务已顺利提交。如图[CFC-true]所示，图[CFC-true] (a)表示一个正在进行的事务或者一个已经终止的事务，图[CFC-true] (b)表示一个已提交的事务。

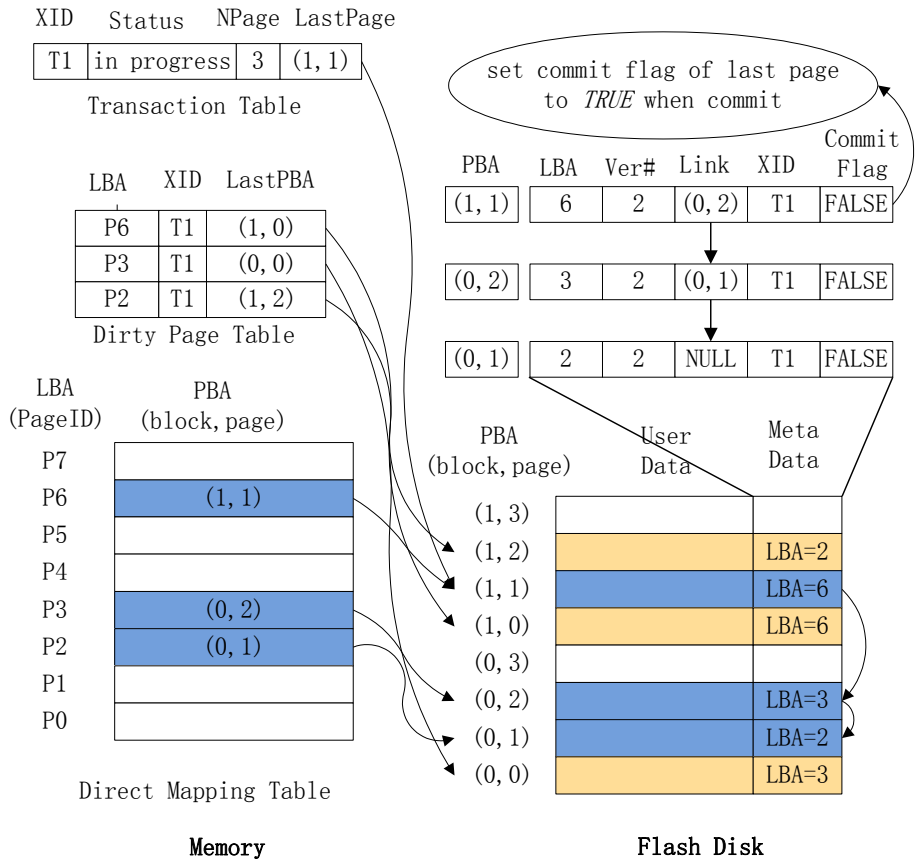


(a) An in-progress/aborted transaction (b) A committed transaction

图[CFC-true] CFC 算法的事务提交标识链表

为了实现 CFC 算法，内存中需要维护几个与事务相关的表，即事务表、脏页表和地址

映射表，如图[CFC-example]所示。



图[CFC-example] CFC 算法的一个实例

事务表(Transaction Table)维护每个事务的状态(*Status*)、该事务所涉及的影子页个数(*NPage*)以及最后一个影子页的地址(*LastPage*)。一个事务可能修改多个逻辑页，那么，就会创建多个影子页。在 CFC 中，将多个影子页通过影子页带外区的 *Link* 域链接成一个链表。在事务表中，用 *NPage* 跟踪影子页的个数，以方便在事务中止时，回收影子页；用 *LastPage* 跟踪最后一个影子页的地址，以方便在事务提交时，将 *LastPage* 的事务提交标识设置为 *TRUE*。在实际运行过程中，事务表只保存活动的事务和已中止的事务。

脏页表(Dirty Page Table)保存了在事务执行之前，逻辑页所对应的物理页地址。保存逻辑页在事务执行之前的物理页地址，可以方便地在事务中止时，将逻辑页映射到事务执行之前的物理页，以此实现事务的回滚。

地址映射表(Direct mapping table)保存了逻辑页到物理页的映射信息。

在 CFC 算法中，带外区将包含以下数据域：逻辑页号（*LBA*）；逻辑页的版本号（*Version #*）；事务 ID（*XID*）；指向前一个影子页的指针（*Link*）；事务提交标识（*Flag Commit*）。

下面将详细介绍 CFC 算法中，事务的更新、提交、中止过程。

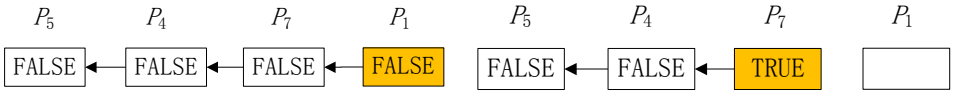
更新事务：当一个事务更新逻辑页时，将执行下面 4 步。1）创建一个与逻辑页对应的影子页，并在影子页的带外区写入逻辑页号、版本号、事务 ID、*Link* 和事务的提交标识（初始时设置为 FLASE）；2）如果事务表中没有该事务，就在事务表中创建一条记录，并初始化 *NPage* 为 1，*LastPage* 为影子页的地址。如果事务表中已经存在该事务，在事务表中修改该事务的影子页计数 *NPage*，并将该事务的 *LastPage* 设为最新的影子页地址；3）在脏页表中写入一条记录，记录逻辑页在执行事务之前的物理页地址；4）在地址映射表中，修改逻辑页的映射信息，将逻辑页映射到影子页。重复执行上面的步骤，直到事务执行完成，或者中止事务。

提交事务：当提交事务时，将事务的 *LastPage* 的提交标识置为 *TRUE*（在 CFC 中，事务中有一个影子页的提交标识为 *TRUE*，则这个事务是已提交事务），在事务表中删除该事务，移除脏页表中的信息，在移除脏页表中信息的同时，将逻辑页之前对应的物理页标记为垃圾页。

中止事务：人工中止事务时，CFC 将会撤销该事务所做的修改。CFC 首先修改事务表中事务的状态信息，然后将该事务所创建的影子页标记为垃圾页，并将地址映射表中，逻辑页映射到事务执行之前的物理页地址。

CFC 算法使用影子页技术实现事务的更新操作。当事务提交时，标记逻辑页在该事务执行之前所对应的物理页为垃圾页；当事务中止时，执行事务回滚操作，以保证数据的一致性与事务的原子性，与此同时，中止事务所产生的影子页将被标记为垃圾页。在数据库系统的运行过程中，空闲页逐渐减少，垃圾页逐渐增多，当空闲页少到一个预先设定的门槛时，将触发垃圾回收操作。垃圾回收操作将回收 1）中止事务所产生的垃圾页，这些页是未提交的；2）已经提交的页，但是，之后又被其他事务标记为垃圾的页。对于第一种情况，直接回收即可，对于第二种情况，还需要维护事务的状态。

下面将详细分析 CFC 算法中垃圾回收的第二种情况。考虑这样一个场景，如图[CFC-garbage]所示，有一个事务修改了 P_5，P_4，P_7 和 P_1，并且事务提交成功。根据 CFC 算法的定义，属于这个事务的所有影子页的事务提交标识至少有一个为 *TRUE*，该事务为已提交的事务。根据 CFC 的事务更新过程，P_1 的提交标识设置为 *TRUE*。假设现在有另外一个事务将 P_1 标记为垃圾页，那么，垃圾回收机制就可以回收 P_1。垃圾回收机制回收 P_1 以后，就剩下 P_5，P_4 和 P_7，并且这三个页的事务提交标识都为 *FALSE*。如果此时发生故障，事务恢复管理器将认为 P_5，P_4，P_7 属于一个中止事务。为了避免这种情况，CFC 算法需要在回收 P_1 的时候，将 P_7 的事务提交标识改为 *TRUE*。如果回收的不是 P_1，而是位于链表中间的 P_7，那么，还需要将 P_4 的事务提交标识设置为 *TRUE*，把链表拆分成两个子链表，将它们当作两个不同的事务对待，并且要分别维护它们的状态。



Before P_1 is garbage collected *After P_1 is garbage collected*

(a) Case I: P_1 is garbage collected

Before P_7 is garbage collected *After P_7 is garbage collected*

(b) Case II: P_7 is garbage collected

图[CFC-garbage] CFC 算法中的垃圾回收

从上面的分析可以看到，维护事务状态的代价很高，每次回收一个页，CFC 都要判断是否需要修改该事务中其它页的事务提交标识，以维护事务的状态。考虑到事务的维护代价，文献[OnXCHH12]还提出了另一个算法——AFC，以降低已提交事务的维护代价。

AFC 算法的基本思想如下：当且仅当一个事务的所有影子页中没有一个影子页的事务提交标识为 *FALSE* 时，该事务是已提交的。AFC 算法中，事务的更新过程与 CFC 类似，只是在初始化时，将第一个影子页的事务提交标识设置为 *FALSE*，将其他影子页的事务提交标识设置为 *TRUE*，当事务提交时，将第一个影子页的提交标识改为 *TRUE*，以此作为事务已提交的判断依据。

采用 AFC 算法以后，所有属于已提交事务的物理页的事务提交标识都为 *TRUE*，当回

收已提交事务的页时，不用任何额外的代价，就能维护该事务的状态，因为该事务的其他物理页的事务提交标识都为 *TRUE*。虽然 AFC 算法不用维护已提交事务的事务状态，但是需要维护已中止事务的事务状态。如图[AFC-algorithm]所示，P_5、P_4、P_7 和 P_1 是已中止事务的影子页，回收这些页。假设首先回收的是 P_5，如果直接回收 P_5，那么物理页 P_4，P_7 和 P_1 将会被认为是已提交事务的数据页，这是因为 P_4、P_7 和 P_1 的提交标识都为 *TRUE*。所以，在回收 P_5 时，需要将 P_1 的提交标识设为 *FALSE*。同理，在回收 P_4 时，也需要将 P_1 的事务提交标识设置为 *TRUE*，将链表拆分成两个链表，并分别维护它们的事务状态。

Before P_5 is garbage collected *After P_5 is garbage collected*

(a) Case I: P_5 is garbage collected

Before P_4 is garbage collected *After P_4 is garbage collected*

(b) Case II: P_4 is garbage collected

图[AFC-algorithm] AFC 算法中的垃圾回收

CFC 算法和 AFC 算法的主要区别在于 CFC 的维护已提交事务的代价较大，AFC 算法维护已中止的事务代价较大，在实际应用中，可以通过不同的工作场景以及不同的事务中断比例来选择。

CFC 算法和 AFC 算法还可以通过记录日志的方式来提供 No-Force 策略的支持。数据库的 No-Force 策略是指在事务提交时，不用强迫刷新闪存中的脏页到外部存储器中。CFC 和 AFC 算法支持 No-Force 策略的方法如下：将逻辑页的修改记录在日志中，在提交事务之前，先将缓冲区中的日志刷新到外部存储器中，此时，就算系统发生故障，也可以通过日记记录，将已提交的事务恢复。

CFC 和 AFC 算法还能够支持数据库的并发操作，即支持多个事务同时读写同一条记录。CFC 和 AFC 算法将每一个事务对数据所做的修改都记录在日志中，在并发的几个事务当中，其中一个事务提交时，都将缓冲区中的日志刷新到外部存储中，并写入一条事务提交日志。如果某个事务中止了对数据的修改，则通过日志文件中的记录，撤销该事务对数据的修改，保留其他事务对数据的更新。

CFC 和 AFC 算法通过在事务的影子页中跟踪事务的状态，减少了日志数量，从而减少了写操作和擦除操作的次数，在一定程度上提高了事务恢复的性能。但是，CFC 和 AFC 算法只能用于支持部分页编程的 SLC 闪存设备中，然而，现在市面上的闪存设备大部分都采用容量更大、价格更低、但是不支持部分页编程的 MLC 闪存。此外，CFC 和 AFC 算法为了支持 No-Force 策略和事务的并发操作，也需要写入日志记录，当写入的日志很多时，CFC 和 AFC 策略将不能获得很好的事务恢复效率。

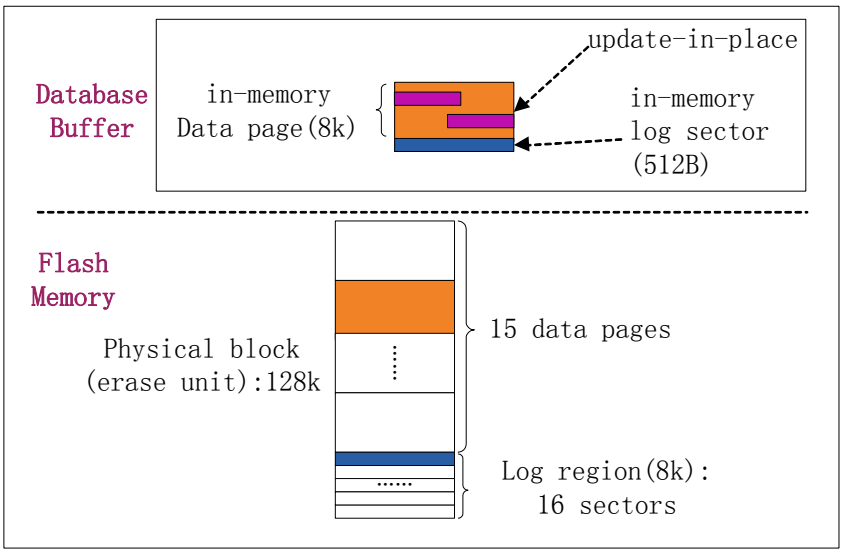
9.4.4 IPL

针对数据库中存在的“数据的细微随机更新”的特点，文献[LeeM07]提出了一种新的、适合闪存数据库的存储管理方法——IPL（In-Page Logging）。IPL 通过采用基于日志的数据更新方法，能够较好地处理对数据的细微随机更新。虽然 IPL 记录日志的主要目的是进行数据更新，但由于这些日志被存储在非易失的闪存中，故同样可以用于事务恢复。

下面将简单地介绍 IPL 的思想，然后重点介绍 IPL 存储管理中的事务恢复机制。

IPL 的基本思想如下：当缓冲区中的数据页被更新时，产生更新日志并在缓冲区中为其分配日志扇区（512B）来缓存日志；当数据页被驱逐出缓冲区时，并不将数据页写入闪存，而是将其日志扇区写入闪存中的日志页内，通过记录日志实现数据的更新。

在数据库运行过程中，当发生缓冲区“脱靶”，即访问的页面不在缓冲区中，需要到外部存储中读取该页时，IPL 需要根据原始的数据页，以及日志页内与该页相关的更新日志构造该页的最新版本。在构造过程中，IPL 需要扫描所有与该页有关的日志页，如果将所有的日志都存放在同一个地方，每构造一个数据页都需要扫描所有的日志，那么，数据页的构造代价将是非常巨大的。为了减少数据页的构造代价，IPL 存储管理器将每一个闪存块分为数据区和日志区，如图[IPL-design]所示。日志区只记录该块中数据页的日志，以此来减少数据页的构造代价。



图[IPL-design] IPL 的设计

图[IPL-design]给出了 IPL 的设计，假设一个闪存块为 128k，一个数据页为 8k。那么，一个闪存块包含 15 个数据页和 1 个日志页。每个日志扇区为（512B），则一个日志页包含 16 个日志扇区。这里需要注意的是，虽然闪存中的写操作的最小单位为页，但是 IPL 利用一部分 NAND 型闪存支持部分页编程技术，将一个日志页(8kb)分为 16 个扇区(512B)，每次可以只写入一个扇区，所需要的时间与写入一个页基本相同。

缓冲区中的每一个脏页都有一个与之相对应的日志扇区，一个日志扇区只能存放有限条日志记录，当日志扇区满或者与其对应的脏页被驱逐出缓冲区时，就将日志扇区写回闪存中的日志页中。如果某一块中的数据页反复被更新，那么该闪存块的日志扇区很快就会写满，此时，IPL 存储管理器就会将闪存块中的数据页与日志页进行合并。合并的过程如下：首先分配一个新的空闲块，然后，将旧的闪存块中数据页与日志页进行合并，构造出最新的数据页，并将最新的数据页写入到新分配的闪存块中，最后，回收旧的闪存块。

IPL 记录日志的主要目的是进行数据更新，但由于这些日志被存储在非易失的闪存中，故可以方便的支持事务恢复。为了支持事务操作，IPL 只需要在的更新日志中加入事务编号、事务状态等信息。

事务提交时，对缓冲区中每个该事务更新过的逻辑页，将该页的日志扇区写入闪存中的相应位置，最后在系统日志区(闪存上的专门区域)写入一条事务提交日志即可。

事务中止时，对缓冲区中每个其更新过的逻辑页，撤销该事务的更新并删除日志扇区中该事务的更新日志，最后在系统日志区写入一条事务中止日志；对于该事务已经被写入闪存的更新日志不作处理。在重构数据页的过程中，IPL 存储管理器忽略已中止的事务所产生的更新日志；在闪存块中的日志扇区写满时，需要将数据页与日志页进行合并，并写入到一个新的空闲块。在合并过程中，将已提交事务的更新日志应用到数据页上，忽略已中止事务所产生的更新日志，将当前活动事务的更新日志转移到新的日志页中。

IPL 在故障恢复时不需要执行 Undo 和 Redo 操作，只需要扫描系统日志区以获取各事务的状态，如果事务是已提交的或者已中止的，则不需要执行任何操作，对于活动的事务，则在系统日志中加入一条记录，表明该事务已被中止，在构造数据页的时候，IPL 管理器将忽略已中止事务的更新日志，以此完成事务的回滚，确保了事务执行的原子性。

IPL 提出了一种新型的存储模式，这种存储模式稍做修改，就能够方便地支持事务操作。但是，IPL 还存在以下问题：1）只能用于支持部分页编程的闪存设备；2）缓冲区中的日志扇区被相关的逻辑页独占，使大多数逻辑页在被换出时其日志扇区中的日志数量较少，故日志扇区被写入闪存时空间利用率较低，闪存中日志页的空间利用率也较低，导致需要写入的扇区数量仍然较多，而这又加快了块内日志空间耗尽的速度，引起了更多的合并和垃圾回收操作，从而影响了系统性能；3）将逻辑页与其日志存储在同一个块内，虽然有助于控制读取和合并代价，但各块内的日志空间大小相同，并未考虑数据更新频率的差异，造成空间浪费。

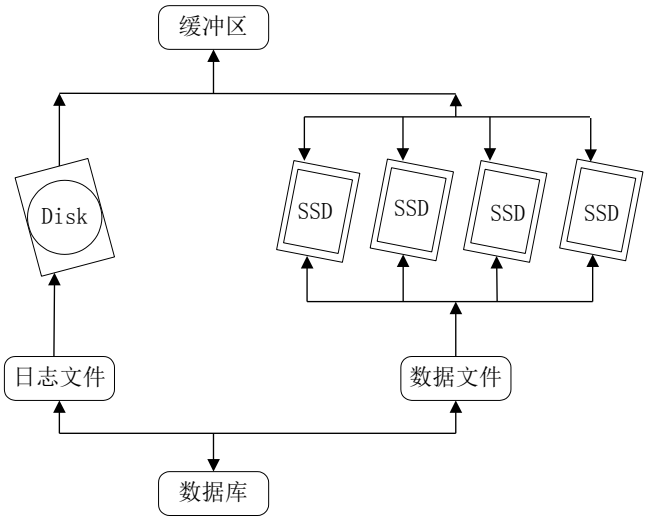
针对以上问题，文献[YueXJL10]提出了一种基于分离日志的闪存数据库系统存储管理方法，简称 OPL。OPL 令缓冲区中的日志块由多个逻辑页共享，当日志缓冲满时，集中写入到闪存的日志块中，因此，提高了空间利用率。并且，OPL 将所有逻辑页的日志集中写入闪存的日志块内，而不是像 IPL，使用部分页编程技术将日志写入逻辑页所在的块，降低了硬件依赖，因此，能够适用于更多的闪存设备。OPL 与 IPL 一样，构造数据页时，需要扫描所有与数据页有关的日志页，但是，OPL 使用打包算法将和同一个逻辑页相关的各日志存放到尽量少的日志页中，以减少读取逻辑页时需要读取的日志数量。OPL 主要是针对 IPL 存储的空间利用不足的改进，对于事务操作的支持，OPL 与 IPL 的思想一样，在此不再熬述。

IPL 与 OPL 都是给出一种新型的存储模式，以提供性能较高的对数据库的操作，它所提出的恢复技术主要是针对这种新的存储方式的一种扩展。因此，IPL 与 OPL 不能方便的扩展到现有的大量数据库中。同时，IPL 与 OPL 的存储方式是针对原始的闪存存储器的，而不是现在被简单广泛地应用在 SSD 上，因此，它们具有相当大的局限性。

9.4.5 HV-Recovery

文献[LuMZ10]提出了一种适合于闪存数据库的日志恢复方法，即 HV-Recovery。虽然 HV-Recovery 使用了日志记录数据的更新，但是，HV-Recovery 在事务恢复时，充分利用了闪存设备异地更新产生的逻辑页的历史版本，所以，HV-Recovery 其实是一种日志事务恢复方法和影子页事务恢复方法相结合的恢复方法。HV-Recovery 通过日志记录逻辑页的历史版本的地址，在事务恢复时检查逻辑页的历史版本是否可用，如果历史版本中的数据没有被破坏，在事务恢复时，就将逻辑页地址映射到历史版本中，不用写入新的数据，从而显著地提高了事务恢复操作的效率。

日志一般都是一些小的写操作，为了节省闪存空间，减少日志记录产出的大量闪存写操作以及闪存写操作引发的代价高昂的闪存擦除操作，HV-Recovery 提出了基于磁盘与闪存的混合存储系统，如图[HV-recovery]所示。HV-Recovery 将数据存放在闪存上，将日志记录存放在磁盘中，通过这样的设计，节约了日志记录所占用的闪存空间，减少了大量写日志操作，从整体上提高了数据库系统的性能。



图[HV-recovery] 混合存储架构图

HV-Recovery 之所以能够在事务恢复时利用逻辑页的历史版本，是因为它使用了一种全新的日志记录方法，即 version_list。version_list 是一个用以保存日志记录版本列表，其结构如表[version-list]所示。version_list 中的日志记录包含了事务 ID、逻辑页号、逻辑页的历史版本(PreAddress)，以及逻辑页的旧值(PreValue)。

Table[version-list] structure of version_list

表[version-list] version_list 结构图

Tid	元素	PreAddress	PreValue
T_1	X	$P(X')$	X'
T_2	A	$P(A')$	A'
T_4	D	NULL	NULL
T_4	B	$P(B')$	B'
T_4	B	DELETE	NULL
T_2	Y	$P(Y')$	Y'
T_1	X	$P(X'')$	X''
T_1	commit	NULL	NULL
……	……	……	……

HV-Recovery 遵循日志先写原则，如果事务改变了数据库元素，则日志记录必须在数据库元素的新值写到外部存储器前写出；如果事务提交，则事务提交日志记录操作必须在事务改变的所有数据库元素已写到外部存储之后再执行。在 HV-Recovery 中，事务更新与事务恢复时，对 version_list 的操作如下：

- **事务更新**需要在 version_list 中写入更新日志。任何事务对数据库的更改操作都会产生一条日志记录，日志记录包含了事务 ID、逻辑页号、逻辑页的历史版本以及逻辑页的旧值。事务修改一个逻辑页就添加一条日志记录，如果事务一次修改多个逻辑页，就添加多条日志记录，如表[version-list]中的第 2 条记录和第 7 条记录；如果事务插入一条新记录，则在日志中添加一条日志记录，这条日志的 PreAddres 和 PreValue 为空，如表[version-list]中的第 3 条记录；如果事务删除一条记录，则在日志中添加两条日志记录，第一条日志记录逻辑页的历史版本和旧值，第二条记录用一个 DELETE 标识表示删除操作，如表[version-list]中的第 4 条记录和第 5 条记录；如果事务多次修改一个逻辑页，则需要多次添加日志记录，如表中的第 1 条记录和

第 7 条记录。

- **事务提交**需要在 version_list 中写入事务提交日志。每当有事务提交时，HV-Recovery 就在日志记录文件，也就是 version_list 中对该事务添加一条新的提交记录，如表中的第 8 条记录。
- **事务恢复**的过程与传统的基于日志的恢复方法不同，HV-Recovery 能够利用闪存中天然存在的历史版本，高效地执行事务恢复操作。由于 HV-Recovery 对日志记录和数据更新的提交顺序满足日志先写原则，所以，只需要对日志记录中体现为尚未提交的事务进行恢复。事务恢复时，先从外部存储中读入日志记录，然后根据这些日志记录，读出需要恢复的各数据项的历史版本的地址，从中取出数据，判断是否与日志记录中存在的旧值相同，若相同，将已写入新更新数据内容的地址标识为无效，将原地址标识为有效，并在地址映射表中将逻辑页的物理地址映射到原地址，从而完成恢复；若不同，则只有重新写入，在闪存中新分配一页，写入日志记录中的旧值，以此完成逻辑页的恢复。

HV-Recovery 通过使用一种全新的日志记录方法，充分利用了闪存设备异地更新产生的逻辑页的历史版本，显著地提高了数据库恢复操作的性能。但是，HV-Recovery 没有考虑事务的并发性，同时，在事务更新与事务提交时，需要写入大量的日志记录，虽然 HV-Recovery 通过将日志记录存储在磁盘中，减少写日志的代价，但是，HV-Recovery 采用了基于磁盘和闪存的混合存储系统，增加了数据库系统的复杂性。

9.5 本章小节

本章内容首先介绍了事务的概念、事务的特性，并分析了传统的事务恢复机制在闪存数据库中的表现，说明了针对闪存数据库设计新的事务恢复方法的必要性；然后，重点介绍了几种典型的闪存数据库事务恢复方法，针对闪存数据库设计的事务恢复方法大都借鉴了影子页的思想（如 Transactional FTL、TxFlash 和 Flag Commit），在闪存的物理页的带外区存储事务提交标识，通过事务提交标识跟踪事务的状态，以此减少写日志操作的数量，提高事务恢复的效率；接下来，又介绍了一种基于日志记录的事务恢复方法，该方法适用于日志结构的文件系统（ Log-Structure File System）；最后，介绍了一种日志恢复技术与影子页恢复技术相结合的事务恢复方法。在介绍不同的事务恢复方法的同时，还分析了各个事务恢复方法的优缺点和应用场景。

9.6 习题

1. 什么是事务？
2. 事务有哪些特性，ACID 各指的是什么？
3. 为什么基于日志的事务恢复方法不适合闪存数据库？
4. 考虑一个事务 T3，它需要一次修改 P1，P4，P7，P2，其中 P1 的物理页地址为（1，0），为它分配的影子页地址为（0，0）；P4 的物理页地址为（1，2），为它分配的影子页地址为（0，2）；P7 的物理页地址为（1，3），为它分配的影子页地址为（0，3）；P2 的物理页地址为（1，1），为它分配的影子页地址为(0，1)。请画出 AFC 算法执行事务 T3 的快照（参考图图[CFC-example]）。
5. 下图是一个 BPCC 算法的例子，请指出逻辑页 A、B、C 和 D 的最新已提交版本。

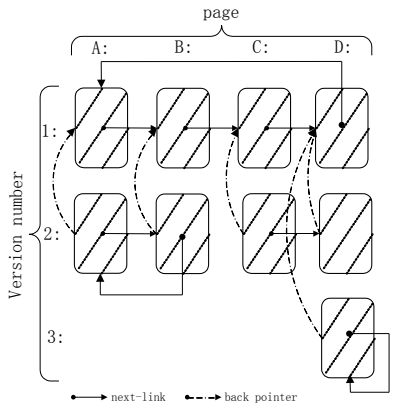


Fig. [BPCC-algorithm-example] An Example TxFlash System State with BPCC

图[BPCC-algorithm-example] BPCC 算法的一个实例

6. 文中介绍的几种事务恢复方法，有哪些方法在事务提交时不需要写入事务提交日志？

第10章 基于混合存储系统的数据库

闪存凭借其优良的特性，已经在企业级存储系统中得到了广泛的应用，大有取代磁盘之势。但是，在目前这个阶段，完全用闪存取代磁盘作为底层存储介质还不现实，主要是因为：（1）磁盘作为存储系统的主要介质已经有很久的历史，里面保存了大量的企业数据，从节约成本的角度而言，大量磁盘仍有继续发挥“余热”的必要；（2）从性能的角度而言，虽然闪存的随机存取速度要明显优于磁盘，但是，就顺序存取速度而言，磁盘仍然具有一定的优势；（3）从价格方面而言，当前闪存的价格仍然要比磁盘高出许多。

因此，在未来较长的一段时期，都会出现闪存和磁盘共存的情形，即由闪存和磁盘一起作为存储介质，数据库既不是单独运行在磁盘上，也不是单独运行在基于闪存的存储设备（比如固态盘）上，而是运行在基于闪存和磁盘的混合存储系统上。但是，由于闪存和磁盘具有明显不同的特性，因此，在混合存储系统中二者将会扮演不同的角色，以期获得最好的投资收益。从闪存和磁盘在存储系统中扮演的角色而言，可以大致包括以下三种情形：（1）闪存代替磁盘作为特种数据的存储介质；（2）闪存和磁盘并列构成混合存储系统；（3）闪存作为介于磁盘和内存之间的缓存。

本章内容将首先分别介绍闪存和磁盘在存储系统中扮演的不同角色，然后介绍一种基于语义信息的系统框架 hStoreage-DB，它可以根据收集到的语义信息，把访问请求分成不同的类型，并为每种类型分配一个合适的 QoS(Quality of Service)策略，这些策略是被底层的混合存储系统所支持的，从而使得每个请求都可以由混合存储系统中合适的存储设备来为之提供服务。

10.1 闪存代替磁盘作为特种数据的存储介质

在混合存储系统中，闪存代替磁盘作为特种数据的存储介质，继续使用磁盘存储作为其他普通数据的存储介质。

10.1.1 存储事务日志

同步事务日志是指，在事务提交之前要求必须把日志“强制写入”到稳定的存储介质上（目前一般采用磁盘），然后，事务才能提交完成，它是 DBMS 为了保证数据一致性和可恢复性的核心机制。

10.1.1.1 同步事务日志成为数据库性能的瓶颈

但是，硬件技术发展的两道“鸿沟”，使得事务日志“强制写入”过程的性能受到严重制约，从而最终严重制约了数据库的整体性能：

（1）CPU、DRAM 内存性能与磁盘性能之间的鸿沟。在过去的这些年里，CPU 和 DRAM 内存的带宽都已经增长了指数倍，但是，磁盘的性能改进的步伐却显得远远落后，磁盘的随机访问延迟和带宽，自从 1980 年以来只提升了 3.5 倍[AthanassoulisACGS10]。这道鸿沟使得数据库中的大量事务日志的“强制写入”过程，都会在与磁盘发生 I/O 操作时遭遇第一道性能瓶颈。

（2）磁盘延迟和磁盘带宽之间的鸿沟。磁盘带宽是由接口决定的，接口的速率一般是指理论最大传输速率，但是，接口提供的带宽远远高于当前的磁盘速度，因为，磁盘内部传输速率制约了整体的速度，而磁盘延迟又是内部传输速率的决定因素。由于磁盘包含磁头、碟片和转轴等机械部件，读写数据都需要一个寻址的过程，因此，磁盘延迟往往较大。在过去的十多年里，磁盘延迟改进的步伐，已经远远落后于磁盘带宽的改进速度，延迟和带宽之间的鸿沟，在未来将会变得更加明显[Patterson05]。这就意味着，磁盘延迟会成为事务日志的“强制写入”过程的第二道瓶颈，也是最大的瓶颈。可以说，事务型数据库应用的性能，更

多地是受到磁盘延迟的限制，而不是磁盘带宽或容量[ZhangYKW02]。

此外，磁盘的技术特性和同步事务日志的访问模式，也使得同步事务日志无法获得好的性能。在 OLTP 系统中，存在大量的事务，因此，会不断生成日志并被强行写入到磁盘上，也就是说，同步事务日志会生成很多小的顺序写操作。这种写操作模式很不适合磁盘，因为在前后写操作之间，磁盘的碟片会连续旋转，写完一个日志条目后，需要碟片转过完整的一圈，然后再在后面的位置写入下一个条目，所以在两个不同的写操作之间会存在一个完整的旋转延迟，这就带来了较大的日志“强制写入”延迟。

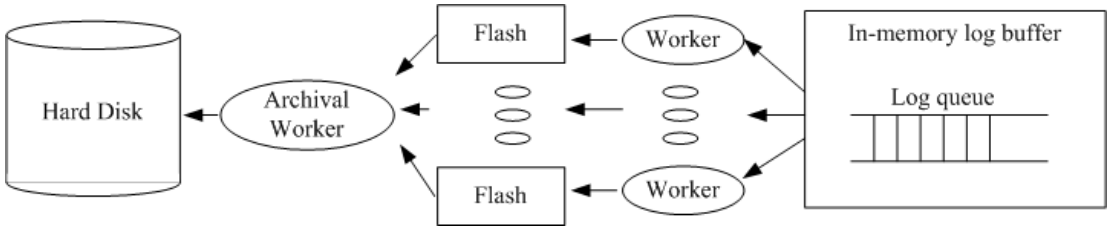
另外，数据库事务机制的特性也带来了较大的日志强制写入延迟。每个要提交事务都必须等到所有属于自己的日志记录被强制写入到日志中以后才能提交，而某个事务的日志写入操作，还必须等待队列中排在它之前的所有其他事务的日志写入操作都已经完成，然后才能开始执行。因此，随着并发事务数量的增加，日志强制写入延迟就会增加，通常不会少于几个毫秒。

随着处理器的速度变得越来越快以及内存容量的增加，由于同步日志“强制写入”而导致的提交时间延迟，会越来越成为获得事务处理高性能的一个严重瓶颈。

10.1.1.2 用闪存存储事务日志可以明显改进事务吞吐量

闪存固态盘具有明显低于磁盘的延迟，可以大大加快同步日志的“强制写入”过程速度，降低平均事务提交时间，从而明显改进事务吞吐量。文献[LeeMPKK08]通过大量的实验表明，通过把磁盘替换成闪存固态盘，会极大缓解在事务提交时间方面的瓶颈，数据库在事务吞吐量方面可以获得指数级的提升。这将会有助于加速在企业数据库应用中采用闪存固态盘，并且帮助数据库从业者重新审视数据库设计的需求和数据库服务器的调优方法。

但是，文献[AthanassoulisACGS10]的研究指出，虽然把磁盘替换成固态盘解决了磁盘的性能瓶颈问题，但是，也带来了新的挑战，因为，大量的随机写操作会恶化性能，并且会很快让闪存磨损老化掉。为了解决这个挑战难题，作者提出了一个 A/P（Append/Pack）设计技术[StoicaAJA09]，它不对 DBMS 做任何修改，可以避免对闪存的随机写操作，同时提出了 FlashLogging[Chen09]来利用闪存进行同步地写入日志。



图[FlashLogging] FlashLogging 的体系架构

图 1 显示了 FlashLogging 的体系架构，它采用了闪存阵列，即可以利用多个闪存设备进行日志记录和恢复性能，当闪存中的日志数据到达一个指定的大小时，这些日志数据会被刷新到磁盘中，因此，磁盘可以看成是一个归档磁盘。闪存阵列可以采用多个低端的 USB 闪存设备，或者也可以是一个高端的固态盘中的多个分区。FlashLogging 要实现两个目标：（1）在常规操作中优化顺序写操作的性能；（2）在恢复过程中优化顺序读操作的性能。当日志写入操作被分配到一个闪存上执行，发现闪存正在进行其他管理操作，比如擦除操作，就需要让日志写入操作处于等待状态，这会增加日志写入过程的延迟。为此，FlashLogging 一旦发现这些异常写操作情况发生，就会把它引导到其他已经准备好的闪存设备中，减少延迟。

10.1.2 存储回滚段

目前有非常多的 DBMS 产品采用了多版本并发控制技术（比如 Oracle、SQL Server 2005、PostgreSQL、 Firebird、InnoDB、Falcon、PBXT 和 Maria 等等），而不是采用单版本加锁的

方法实现并发控制。多版本并发控制技术保存了某一时刻数据的一个快照，对于一个事务而言，无论运行多久，都可以保证它能够看到一致的数据。

多版本并发控制技术中使用了一个非常重要的设计——回滚段，必须存放在稳定的存储介质上，用于存放数据修改之前的值。一个事务只能使用一个回滚段来存放它的回滚信息，而一个回滚段可以存放多个事务的回滚信息。需要特别指出的是，多版本并发控制技术中的回滚段，和数据库的事务日志具有很大的区别，前者只用于多版本并发控制，不用于事务的恢复，而后者主要用于事务恢复。在多版本并发控制下，更新一个数据对象时，需要把它之前的旧版本写入到一个回滚段，同时还要为这次更新书写相应的重做日志。

当一个事务被创建时，它会被分配给一个特定的回滚段，然后，这个事务就会把数据对象的旧版本按照追加模式顺序地写入到这个回滚段。多个并发事务会同时生成并行的回滚段写操作数据流。在当前的区间中（分配给对象的任何连续块，称为“区间”，也可称为“扩展”），如果回滚段耗尽了空间，就会给这个回滚段分配一个新的区间。因此，针对回滚段的连续写请求，对应的物理地址空间往往是不连续的。如果采用磁盘驱动器来存储回滚段，那么，针对回滚段的许多连续写请求，很可能必须把磁头移动到不同的磁道上面去执行。因此，对于多版本并发控制而言，由于磁盘寻址存在大量延迟，记录回滚数据的代价就会比较高。

另外，在多版本并发控制技术中，如果采用磁盘作为存储回滚段的介质，读取一个数据对象也具有很高的代价。当一个数据对象更新时，就要把该数据对象的旧版本写入回滚段，如果发生过多次更新，就会在回滚段中记录多个旧版本，这些旧版本构成一个数据链表，从而加速查找遍历旧版本的速度。当读取这个数据对象时，需要首先检查它是否被更新过，如果曾经发生了更新，就需要到回滚段中查找旧版本链表，找到正确版本的旧数据。可以想象得到的是，当这个数据对象被频繁更新时，旧版本链表会很长，而且，由于回滚段采用了顺序追加模式写入旧版本数据，这些旧版本会被分散到不同的磁盘物理空间内，遍历旧版本链表会带来大量的磁头移动开销，恶化数据库性能。

从上面论述可以看出，采用磁盘存储回滚段时，由于存在较高的寻址延迟，写入和读出操作的代价都很高，影响了数据库的性能。闪存的随机访问延迟很低，很适合作为回滚段的存储介质，而且，数据对象的旧版本是以追加的方式顺序写入到回滚段中的，这种方式非常符合闪存异地更新的特性。文献[LeeMPKK08]通过大量实验研究，从两个方面讨论了采用闪存盘作为回滚段的存储介质带来的性能收益：

- 对写操作的性能改善有限：虽然闪存固态盘的写延迟要比磁盘的写延迟低很多，但是，在回滚段写操作性能方面，采用固态盘和磁盘的性能差异不大。导致这个问题的主要原因在于固态盘空间有限，存在垃圾回收过程，会产生大量代价昂贵的合并操作（包含擦除操作和写操作）。
- 对读操作的性能改善较为明显：闪存的随机访问速度和顺序访问速度是一致的，比磁盘块许多，因此，读取保存在固态盘中的回滚段数据时，虽然这些数据分散在不同的物理空间，但是，仍然可以获得较快地读取这些数据。

10.1.3 存储中间结果

基于闪存的固态盘可以成为一种比较理想的中间结果的临时数据存储介质。例如，在数据库中的分析查询经常需要运行连接算法，在连接算法执行过程中，会生成大量的中间结果，这些中间结果存取的快慢会直接影响到连接算法生成最终结果的速度，从而影响查询的响应速度。在“内存-磁盘”的存储体系架构下，连接操作的大量中间结果通常需要保存到磁盘中，因为，内存空间有限，无法容纳大量的中间结果。固态盘具有很高的性价比，相比磁盘而言，具有更高的 I/O 带宽，也不会受到机械部件的限制。因此，固态盘非常适合用于临时存储中间结果。尤其对于非阻塞连接算法而言，它的全部外部 I/O 都是从临时存储中读取数据的 I/O，采用固态盘作为临时存储，可以大大减少从临时存储中读取数据的时间，从而加快了算法完

成速度。

10.1.3.1 连接算法介绍

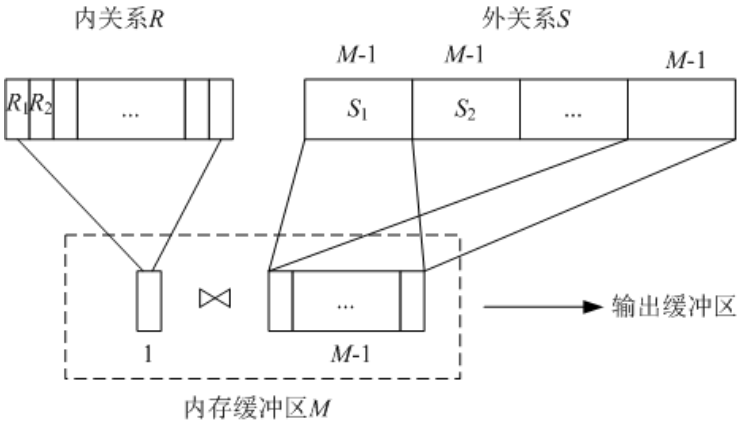
数据库系统中采用的连接算法主要包括：块嵌套循环连接、归并排序连接和哈希连接。

10.1.3.1.1 块嵌套循环连接

块嵌套连接算法的伪代码如下：

```
JoinResult={};/*初始化结果集合*/
Buffer=M;/*内存缓冲区大小为 M 个块*/
for each M-1 blocks in S /*每次从外关系 S 中读取 M-1 个块到内存缓冲区中*/
    Read M-1 blocks of S into Buffer;
    for each block in R /*每次从内关系 R 中读取 1 个块到内存缓冲区*/
        Join M-1 blocks of S and 1 block of R in Buffer;/*在内存中对块中元组
执行连接*/
        Output the join results into JoinResult;
    endfor
endfor
Return JoinResult;
```

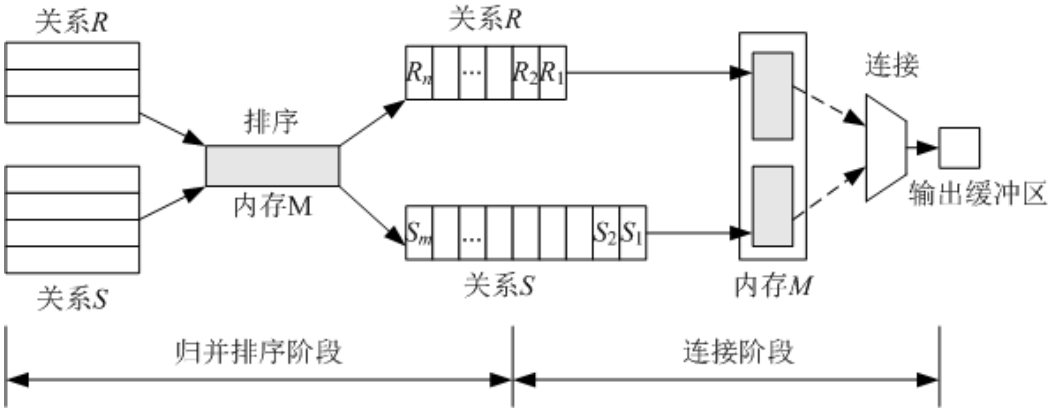
可以看出，块嵌套连接算法包含了一个二重循环，通常把处于外循环中的关系 S 称为“外关系”，而把处于内循环中的关系 R 称为“内关系”。采用块嵌套循环连接算法执行两个关系 R 和 S 的连接操作时，通常把较小的关系 S 作为“外关系”，把较大的关系 R 作为“内关系”。图[join(a)]给出了块嵌套循环连接方法的原理。假设内存空间大小为 M 个块，算法为关系 R 和 S 分配内存空间时，会把其中的 M-1 个块的内存分配给关系 S，把 1 个块的内存分配给关系 R。外关系 S 会按照 M-1 个块的大小，被切分成若干个子片段，即每个子片段（除了最后一个子片段）的大小都是 M-1 个块。当连接操作开始执行时，首先把外关系的第一批的 M-1 个块（即 S_1）读入内存，然后，读入关系 R 的第 1 个块（即 R_1），在内存中对 S_1 和 R_1 进行连接，把连接结果放入输出缓冲区中（当输出缓冲区满时，缓冲区中的连接结果会被刷新到底层的辅助存储器中，比如磁盘）。接下来，再读入关系 R 的第 2 个块（即 R_2），在内存中对 S_1 和 R_2 进行连接，把连接结果放入输出缓冲区中。依此类推，分别顺序读入关系 R 的其他块 R_i，在内存中对 S_1 和 R_i 进行连接。这个过程结束后，就完成了子片段 S_1 和关系 R 的连接操作，然后，就可以再次读入外关系 S 的第二批的 M-1 个块（即 S_2），重复上述过程，完成子片段 S_2 和关系 R 的连接操作。依此类推，可以完成每个子片段 S_j 和 R 的连接操作，从而最终完成整个外关系 S 和内关系 R 的连接操作。



图[join(a)] 块嵌套循环连接方法

10.1.3.1.2 归并排序连接

归并排序连接算法包括两个步骤（如图[join(b)]所示），即归并排序和连接。在归并排序这个步骤中，需要对参与连接的关系 R 和 S 在连接属性上进行排序。在连接这个步骤中，需要按照连接属性的顺序扫描两个关系，同时对两个关系中的元组执行连接操作，具体方法如下：把已经在连接属性上排序的关系 R 和 S 的第一个块 R_1 和 S_1，都读入内存，R_1 和 S_1 都包含了若干个元组，在内存中对 R_1 和 S_1 中的元组按照顺序执行连接操作，如果 R_1 和 S_1 的参与连接比较的两个元组相等，则把连接结果输出到缓冲区，如果两个元组不相等，则废弃值较小的元组并从该元组所属的块（R_1 或者 S_1）中输入下一个元组参与连接的比较。当 R_1 或者 S_1 中的某个块中的元组消耗殆尽时，就从该块所属的关系（R 或 S）中顺序读入下一个块，继续执行连接操作。最终，当来自关系 R 和 S 的所有块都消耗殆尽时，连接过程结束。

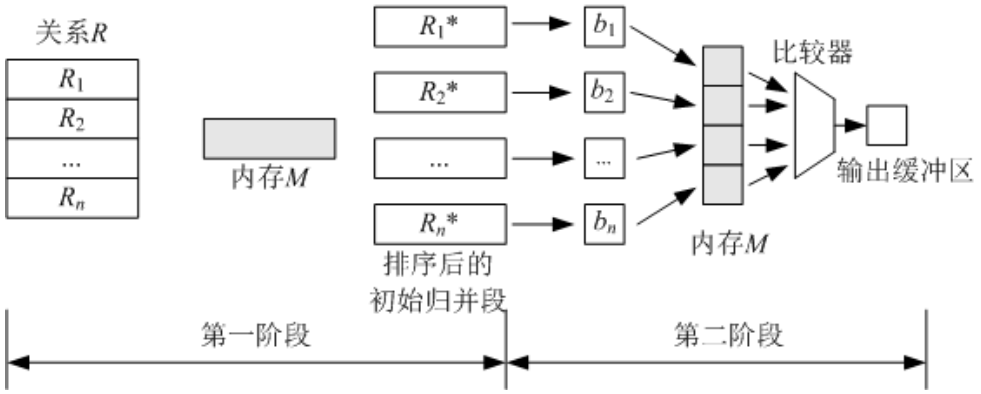


图[join(b)] 归并排序连接算法

归并排序连接算法的第一个步骤归并排序，是一种外部排序。外部排序是数据库中的一种常见操作，是指针对大数据集的排序，即待排序的元组存储在外存储器（比如磁盘）上，大量待排序的元组无法一次性放入内存，需要在内存和外部存储器之间进行多次数据交换，最终实现对整个数据集进行排序的目的。如图[join(c)]所示，外部排序一般包括两个阶段：

（1）第一阶段：**得到初始归并段**。根据内存容量的大小，把大数据集切分成若干个数据块，每个数据块都可以单独放入内存，即把图[join(c)]中的关系 R 切分成若干个块 $R_1,R_2,\ldots,R_n$，然后，采用有效的内存排序算法（比如快速排序和堆排序等）进行排序，由此可以得到若干个初始归并段，每个归并段内部的元组都是已经经过排序的；对于图[join(c)]中的关系 R 而言，经过内存排序后的初始归并段为 $R_1*,R_2*,\ldots,R_n*$。

（2）第二阶段：**归并排序。**进行逐趟归并，比如，假设有 10 个初始归并段，如果采用 2-路归并，那么第一趟由 10 个归并段得到 5 个归并段，第二趟由 5 个归并段得到 3 个归并段，第三趟由 3 个归并段得到 2 个归并段，第 4 趟（最后一趟）归并得到整个元组集合的有序序列。在执行每趟归并时，把初始归并段 R_1*中的第 1 个块 b_1，R_2*中的第 1 个块 b_2，……，R_n*中的第 1 个块 b_n，都读入内存中，每个块都包含了若干个元组，然后，在内存中，对 b_1，$b_2,\ldots, b_n$ 第一个元组进行比较，假设这里按照降序排序，则经过比较后，把具有最大值的元组放到输出缓冲区，并从内存中删除该元组；同理，可以进行后面的若干次元组比较，每次比较后，都会把具有最大值的元组放到输出缓冲区，并从内存中删除该元组，从而使得 b_1，$b_2,\ldots, b_n$ 这些块中的元组会越来越少，当某个块的元组消耗殆尽时，就从该块所属的初始归并段中读入下一个块到内存中，这就会使得初始归并段剩余的块数越来越少，直至最终每个初始归并段的所有块都被消耗殆尽时，这一趟的归并排序阶段就彻底完成，继续进入下一趟归并排序过程，再重复上面的步骤。



图[join(c)] 归并排序

10.1.3.1.3 哈希连接

哈希连接有两种输入关系，即生成输入和探测输入，查询优化器会指派这些角色，一般而言，为了能够让生成输入的哈希桶能够放入内存，通常把两个输入关系中元组较少的关系作为生成输入。这里假设生成输入是元组数量较少的关系 S，探测输入是关系 R。

哈希连接包括多种形式，主要有：内存中的哈希连接、Grace 哈希连接、递归哈希连接和混合哈希连接。生成输入和内存大小的关系，决定了采用哪种哈希连接方法。当生成输入可以一次性放入内存时，可以使用内存中的哈希连接；当生成输入大大超过内存大小时，则采用 Grace 哈希连接；当生成输入只是略微大于内存大小时，就可以采用混合哈希连接。数据库系统的查询优化器在执行具体的连接运算时，通常无法确定始终使用一种方法，比如，可能在开始阶段使用的是内存中的哈希连接，然后根据生成输入的大小逐渐转换到 Grace 哈希连接和递归哈希连接。

10.1.3.1.3.1 内存中的哈希连接

当生成输入可以一次性放入内存时，可以使用内存中的哈希连接，它的执行过程包括两个阶段：第一个阶段是生成阶段，首先在内存中生成哈希表，然后扫描生成输入 S，使用哈希函数把 S 中的每个元组在连接属性上进行映射，分别放入相应的哈希桶中；第二个阶段是探测阶段，每次从输入关系 R 中顺序读取一个元组，首先在连接属性上计算该元组的哈希键的值，然后，把该元组和内存中相应的、具有相同键值的、属于 S 的哈希桶中的元组进行比较连接，并输出结果。

10.1.3.1.3.2 Grace 哈希连接

当生成输入大于内存，无法一次性放入内存时，需要采用 Grace 哈希连接算法，它的执行过程包括两个阶段：第一个阶段是，使用相同的哈希函数 f_1 把参与连接的两个关系 R 和 S，在连接属性上分别映射到不同的哈希桶中，这个过程结束后，所有在连接属性上具有相同的键值的元组，都会被放入到同一个哈希桶中，由此可以得到关系 R 和 S 的若干个哈希桶 $R_1,R_2,\ldots,R_n$ 和 $S_1,S_2,\ldots,S_n$，这些哈希桶分别被存放在磁盘的不同文件中，每个哈希桶对应一个分区文件，从而可以有效减少单个输入的元组数量，另外，对关系 R 和 S 采用相同的哈希函数，可以保证关系 R 和 S 中在连接属性上具有相同值的元组，一定会分别被放入到配对的哈希桶$<R_i, S_i>$中；第二个阶段，对关系 R 和 S 中对应的哈希桶中的元组进行连接操作，即对每个配对的哈希桶$<R_i, S_i>$中的元组执行连接操作，需要首先读入关系 S 的一个分区文件 S_i，使用一个哈希函数 f_2 在内存中为 S_i 构建一个哈希表 h，然后，依次顺序读入分区文件 R_i 中的每个元组 t，使用元组 t 在哈希表 h 上进行探查，输出连接结果。在第二个阶段，如果要求每个配对的哈希桶$<R_i, S_i>$中的元组能够只需要一次读入内存就可以完成连接操作，那么，就必须使得 S_i 能够一次性放入内存，这就要求必须采用一个合适的哈希函数，使得哈希得到的每个哈希桶 S_i 的大小不超过内存的大小。当 S_i 能够一次性放入内存时，可以首

先把 S_i 全部读入内存，然后，把 R_i 中的元组依次顺序读入内存，每读入一个 R_i 中的元组 t，就把 t 和 S_i 中的全部元组都进行一次比较连接，输出连接结果，然后丢弃 t，再继续从 R_i 中读入下一个元组。当 R_i 中的元组全部消耗殆尽，就完成了<R_i, S_i>中的元组的连接。

需要注意的是，Grace 哈希连接中，分区文件的构建（每个分区文件包含了一个哈希桶）和连接被放入两个不同的阶段，在执行连接操作时，需要再次把这些分区文件从磁盘读入内存。

10.1.3.1.3.3 混合哈希连接

当生成输入只是略微大于内存大小时，则可以采用混合哈希连接，即把内存中的哈希连接和 Grace 哈希连接的元素结合在一个步骤中。

在第一个阶段中，首先，需要把较小的关系 S 顺序地读入内存，对于每个元组，都在连接属性上对其使用合适的哈希函数，把元组映射到相应的哈希桶 S_i 中。与 Grace 哈希连接不同的是，混合哈希连接并非把所有的哈希桶都写入到磁盘文件中，由于一部分内存缓冲区本来就是用来在第二个阶段（连接阶段）为每个哈希桶 S_i 构建哈希表的，因此，关系 S 的最后一个哈希桶 S_n，就可以不用写入到磁盘文件中，只需要把其他 n-1 个哈希桶写入文件。然后，需要把关系 R 的每个元组顺序读入内存，并在连接属性上对每个元组使用合适的哈希函数，把元组映射到相应的哈希桶 R_i 中，在这个过程中，每个读入内存的关系 R 的元组，都可以直接和那些没有写入文件而是保存在内存中的哈希桶 S_n 中的元组进行连接操作，这个过程完成后，既可以完成关系 R 和 S_n 的连接操作，同时，也可以得到关系 R 的 n-1 个分区文件。

混合哈希连接的第二个阶段和 Grace 哈希连接的第二个阶段完全相同。

10.1.3.2 采用固态盘存储连接算法的临时结果

文献[ChenGN10]提出了一个非阻塞连接算法 PR-join，并探讨了把固态盘作为临时存储的可行性和优势。PR-join 可以取得近似最优的性能，它利用固态盘作为临时存储来存放分片后的数据。PR-join 可以支持在线聚合和数据流处理。文献[LeeMPKK08]提出采用闪存固态盘存储临时表空间，主要是存储外部排序和哈希连接查询生成的临时结果，可以明显提高数据库性能。文献[DoP09]在一个单线程、轻量级的数据库引擎上，分别采用闪存固态盘和磁盘，对四个流行的即席连接算法进行了性能测试，包括块嵌套循环连接、归并排序连接、Grace 哈希连接和混合哈希连接，通过对实验结果的分析得到如下结论：

（1）在闪存固态盘上的连接操作的性能，更有可能会受到 CPU 性能的限制，而不是受限于 I/O 的性能，因此，改变 CPU 性能的方法（比如更好的缓存利用方式），对于固态盘而言非常重要；

（2）对于闪存固态盘而言，设计更多随机读操作，尽量减少随机写操作，可能是一个比较好的设计选择；

（3）和顺序写操作相比，随机写操作在闪存上产生了更多的 I/O 变化，这使得连接操作的性能更加难以预测。

当把磁盘替换成闪存固态盘时，对于上述四种连接算法而言，在性能改进方面，文献[DoP09]的测试结果给出了如下结论：

（1）把磁盘替换成固态盘，对于所有四种连接算法都是有益的。块嵌套循环连接算法的 I/O 模式是顺序读，因此，具有最大的性能改进。其他连接算法在固态盘上的性能也都要好于在磁盘上面的性能，但是改进幅度要小于块嵌套循环连接算法，这是因为在闪存固态盘上面，写传输速率要比读传输速率慢，无法预料的擦除操作可能会进一步恶化写操作的性能。

（2）混合哈希连接，无论是在磁盘上还是在固态盘上面，都要比其他连接算法的性能要好。

（3）采用固态盘时，对于较大的缓冲池尺寸而言，块嵌套循环连接比归并排序连接和

Grace 哈希连接的性能都要好，但是，比混合哈希连接要慢，因为，混合哈希连接的 CPU 效率更高（虽然需要更多的 I/O 代价）。

（4）Grace 哈希连接在所有阶段的 I/O 加速比，都要比归并排序算法低。对于 Grace 哈希连接而言，主要 I/O 模式是第一个阶段的随机写操作和第二个阶段是顺序读操作。Grace 哈希连接的第二个阶段是顺序读操作，因此，可以取得较高的加速比，但是，第一个阶段是随机写操作，在闪存上会伴有昂贵的擦除操作，因此，第一个阶段的 I/O 加速比很低。对于归并排序算法而言，第一个阶段是顺序写操作，性能要比 Grace 哈希连接好（第一阶段是随机写），第二个阶段是随机读操作，按理说，性能应该和 Grace 哈希连接的第二阶段一样，因为，闪存上面的顺序读和随机读操作具有一致的性能，但是，Grace 哈希连接在第一阶段的随机写操作，产生的闪存性能的负面影响会持续一段时间，因此会影响到第二个阶段的顺序读操作的性能，这就导致 Grace 哈希连接的第二个阶段的 I/O 加速比仍然要比归并排序算法比。这就说明，那些强调随机读并且尽量避免随机写操作的算法，在闪存固态盘上面可能获得更大的性能改进。

10.1.3.3 采用固态盘可以提升归并排序连接算法性能的原因分析

这里将分析采用闪存固态盘作为中间结果的临时存储介质是如何提高归并排序连接算法性能的，对于哈希连接查询时采用固态盘存储中间结果，也具有类似的性能提升效果，将不再赘述。

归并排序连接算法包含一个外部排序的过程，对于外部排序整个过程而言，在数据存储方面可以大致划分成三个子过程：（1）初始大数据集的存储；（2）多个数据块的外部排序中间结果的存储；（3）排序后的大数据集的存储。用闪存固态盘取代磁盘存储外部排序的中间结果，发生在第 2 个存储子过程，其他两个存储子过程的外存储器仍然采用磁盘，因此，这里只对第 2 个存储子过程所具备的 I/O 模式的特点进行简单分析，它的 I/O 模式特点可以概括为：顺序写操作和随机读操作[GraefeLS94]。每次把内存中的排序结果写入外存储器就是顺序写操作，每次从外存储器读取数据就是随机读操作。

如果采用闪存固态盘存储外部排序的中间结果，由于固态盘具有较低的读写延迟，并且随机读和顺序读操作同样快，所以，可以明显提升外部排序过程的性能。文献[LeeMPKK08]在商业数据库上运行外部排序，测试了分别采用固态盘和磁盘存储中间结果时的排序性能，这里假设大数据集被分成 n 个数据块，得到 n 个初始归并段以后，采用 n-路归并排序，这样，归并排序阶段只需要从外存储器中进行一次随机读操作（读入 n 个排序后的归并段），从而使得整个外部排序的第 2 个存储子过程的 I/O 模式，就会呈现出明显的两阶段特征，即第一个阶段的顺序写操作和第二个阶段的随机读操作。

文献[LeeMPKK08]中得到了图[external-sort]中的实验结果，其中，横坐标轴表示时间，纵坐标轴表示读写操作对应的逻辑地址空间。图[external-sort](a)和(b)这两个子图的左边，是第一个阶段生成每个初始归并段后写入外存储器时的逻辑地址空间分布情况，可以看出，对于固态盘和磁盘而言，都呈现出一条逐渐向上的“斜线段”，这正好符合顺序写操作的特征，因为初始归并段后写入外存储器就是一个顺序写操作过程；图[external-sort](a)和(b)这两个子图的右边，代表了第二个阶段归并排序时从外存储器读入数据时的逻辑地址空间分布情况，可以看出，对于固态盘和磁盘而言，都覆盖了较大范围内的逻辑地址空间，即读操作是随机分布在这些逻辑地址空间上，这正好符合归并排序时的随机读的特征。从图[external-sort]中可以看出，在第一个阶段的顺序写操作方面，采用两种不同存储介质时的时间开销差异不大，但是，在第二个阶段的随机读操作方面，采用闪存固态盘时的时间开销，明显要比采用磁盘时的时间开销少很多，这说明固态盘确实可以大幅度提升外部排序的性能。

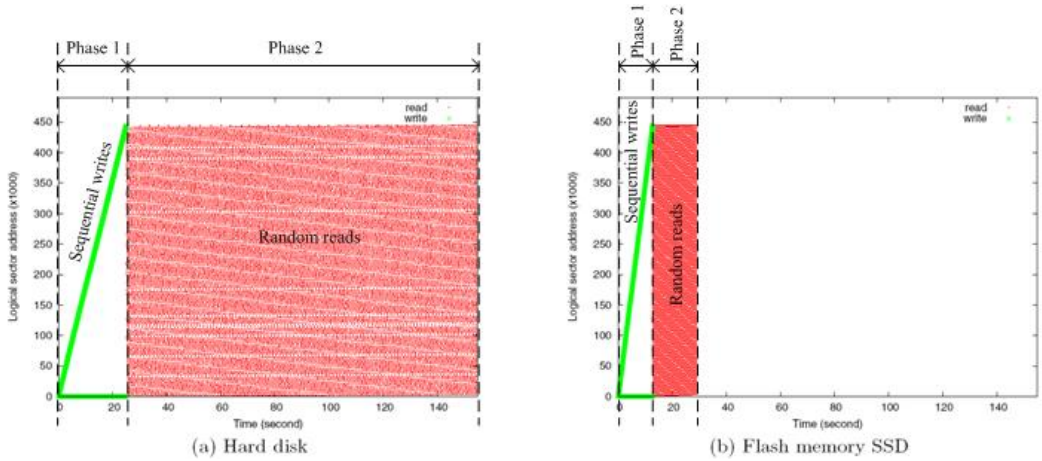


Fig. [external-sort] The I/O pattern of external sort

图[external-sort] 外部排序的 I/O 模式

10.2 闪存作为介于磁盘和内存之间的缓存

在过去的这些年里，CPU 和 DRAM 内存的带宽都已经增长了指数倍，但是，磁盘的性能却没有获得同步的增长，磁盘的随机访问延迟和带宽，自从 1980 年以来只提升了 3.5 倍[AthanassoulisACGS10]。因此，在带宽和随机访问延迟方面，磁盘和内存之间的鸿沟越来越大，磁盘已经成为数据库应用的性能瓶颈。基于闪存的固态盘很好地填补了磁盘留下来的这道鸿沟。相对于磁盘而言，固态盘读写速度快，每字节价格也在逐渐降低；相对于 DRAM 内存而言，固态盘的价格更低，容量更大。因此，随着固态盘技术的不断发展和成熟，将这种闪存设备纳入到存储体系架构中将会带来较大的收益，即把“CPU 高速缓存(cache)-内存-磁盘”这种传统的三层存储体系架构扩展为“CPU 高速缓存(cache)-内存-闪存-磁盘”。

10.2.1 典型应用

把基于闪存的固态盘作为介于磁盘和内存之间的缓存，典型的应用包括：

- 作为数据库系统的二级缓存
- 作为数据仓库的更新缓存

此外，FlashStore[DebnathSL10]也是一种用于在磁盘和内存之间的缓存设备，它采用“键/值”形式存储数据。

10.2.1.1 作为数据库系统的二级缓存

文献[CanimMBRL10]提出在数据库管理中，把闪存固态盘作为磁盘和内存之间的缓存层，如果内存视为数据库系统的“一级缓存”，那么，固态盘就可以视为数据库系统的“二级缓存”（和 CPU 的二级缓存不是一个概念）。在固态盘中缓存频繁被访问的数据，可以减少磁盘 I/O。

当前的数据库系统中，内存的配置容量越来越高，数据库服务器中配置几十 GB 内存已经比较常见。这可能会给人造成一种错觉，认为内存已经足够大，足以用来存储数据库中被频繁访问的数据。实际上，这种错觉的形成是因为忽略了另外一个重要事实，那就是磁盘中保存的数据也在迅速膨胀。在电信、金融、气象、物流、零售等领域，每天都有海量的数据生成，包括中国移动在内的各个电信企业的日常运营数据都在 TB 级别以上。可以说，内存容量增加的步伐远远落后于海量数据的增加步伐，面对磁盘中的海量数据，内存空间仍然显得捉襟见肘。当需要缓存的、频繁访问的数据集的大小，超过内存中为数据库开辟的缓冲区大小时，物理 I/O 就会稀疏地分布到大量不同的磁盘页中，这就意味着，没有什么数据局部性可以让缓存得以有效利用来改善数据库性能。因此，在内存作为“一级缓存”的基础上，采用容量更大、价格更低的固态盘作为数据库系统的“二级缓存”，具有很强的必要性，可以给数据库系统系能带来明显收益。

当前已经有一些主流商业数据库采用闪存固态盘作为数据库系统的缓存。例如，当数据从磁盘中读取出来时，Oralce Exadata 会把热数据页缓存在闪存中[Oracle10]。热数据选择是根据数据类型进行静态选择的，索引比日志和备份具有更高的优先级。类似地，IBM DB2 的扩展缓存原型系统[CanimMBRL10]，提出了一个基于温度感知的缓存策略 TAC（Temperature Aware Caching）机制，它依赖于数据访问频率。TAC 监测器会连续监测数据访问模式来确定热数据页。热数据页在从磁盘到 DRAM 缓冲区的路上，会被缓存在闪存缓存中。TAC 采用了“贯穿写”（write-through）策略，即当一个脏页被从 DRAM 缓冲区中驱逐出来时，它会被同时写入到闪存缓存和磁盘中。

Canim 等人[CanimMBRL10]在 IBM DB2 数据库上面运行 TPC-C 负载，并且跟踪记录了物理 I/O 请求在磁盘页中的位置分布情况（如图[warm-cold-page]所示）。根据对记录结果的观察，他们发现一个比较有趣的结果：如果数据访问被分组成连续的、固定尺寸的物理存储区域（或称为数据区域），那么，就会存在“温区”和“冷区”。温区对应着的数据会被适度频繁地访问，但是不足以频繁到需要放入到 RAM 缓存中，能够进入 RAM 缓存的数据都是被非常频繁访问的热数据。冷区中的数据，或者是访问不频繁，或者是数据太大以至于单位区域内的访问频度就显得相对较低。

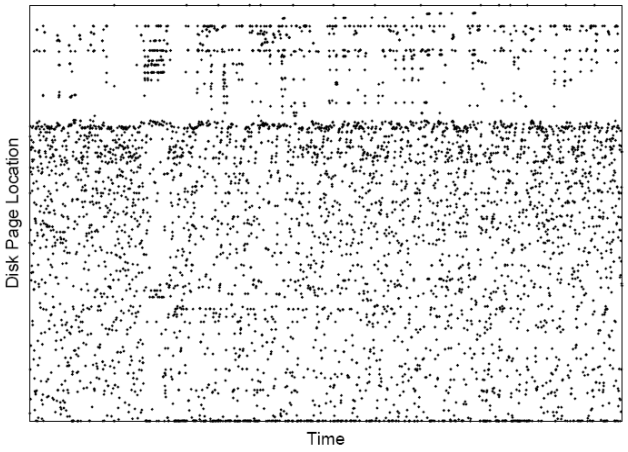


Fig.[warm-cold-page] Disk page access statistics for a TPC-C workload

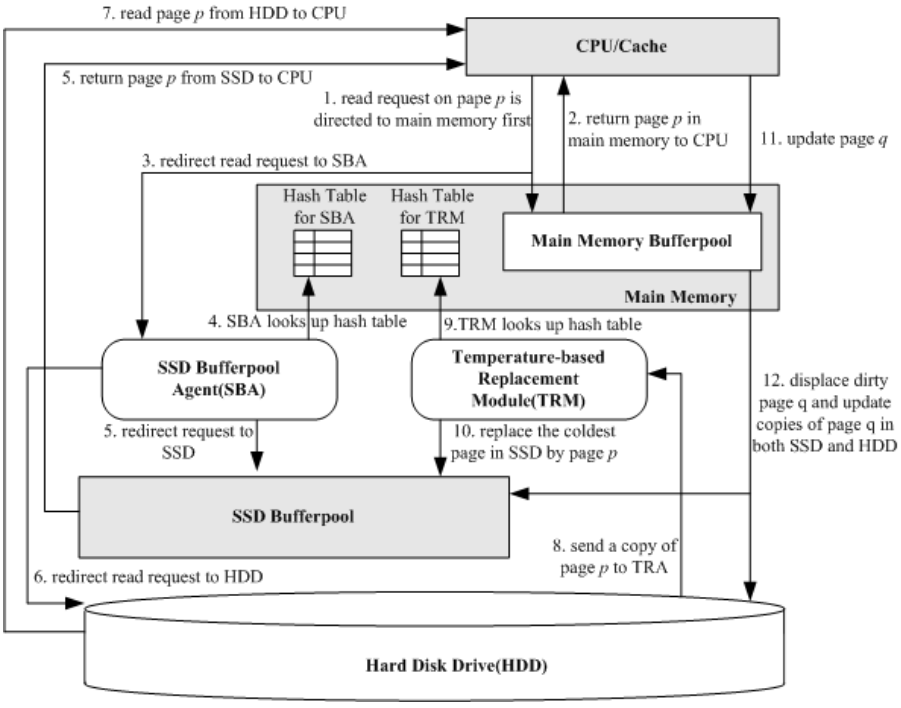
图[warm-cold-page] 在 TPC-C 负载下一个磁盘页的访问情况统计

作者对温区和冷区进行实时的温度监控，也就是统计每个数据区域的访问频度、最后访问时间等信息，用作固态盘缓存的替换算法的依据。如果一个页来自温区，就把它放入到固态盘缓存中。一个来自冷区的页也有可能会被放入到缓存中，但是前提是它不能替换缓存中来自温区的页。由于温区和冷区是以一个区域为统计单位，而不是以单个页为统计单位，因此，某个页的相关访问信息都会统计到它所属的区域上面。以区域为单位统计访问信息，相对于以页为单位统计访问信息的优点是：（1）统计信息消耗的空间更小；（2）同一个区域中，对某个页面 p 的相邻页面 q 的访问，可能会增加对 p 访问的机会，把 p 提前放入缓存会带来收益，但是，采用传统的基于页的统计方法时，由于 p 没有被访问过，就会一直是个冷页，无法进入缓存；采用基于区域的统计方法后，访问 p 的相邻页 q 后，访问次数是增加到区域上面，这会使得区域温度升高，由此，属于该区域的页 p 也会成为温区中的页，从而获得更多的进入固态盘缓存的机会。

图[SSD-as-second-level-cache]显示了采用固态盘作为数据库系统的二级缓存的体系架构，由磁盘存储数据库的所有数据，用内存缓冲区存储最频繁访问的热数据，用固态盘缓冲区存储被适当频繁访问的温数据，并设置一个基于温度的替换模块实现对两个缓冲区中的数据的管理和替换。当 CPU 需要读取数据页 p 时，首先需要查找内存缓冲区，如果 p 在内存

缓冲区中，就直接从内存读取数据。如果 p 不在内存缓冲区中，这个读请求会被转交给固态盘缓冲区代理。代理去到内存中查找哈希表来判断 *p* 是否在固态盘缓冲区中，如果在，*p* 就会被读入到内存缓冲区中；如果不在，就从磁盘中读取 *p*，并同时把 *p* 的一个副本传送给基于温度的替换模块。替换模块会确定是否用 *p* 来替换固态盘缓冲区中的页（假设之前缓冲区已经填满），如果 *p* 比固态盘缓冲区中当前最冷的干净页的温度还要高，则把它写入到固态盘缓冲区中。内存中保存了一个哈希表，记录了每个数据区域的温度信息，替换模块可以快速获得各个页所属区域的温度信息。为了迅速找到最冷的页，系统为所有在缓存中的页维护一个堆栈，以温度为基础组织堆栈中的页，这样，寻找最冷页的平均时间就是堆栈中页的数量的对数。数据页的更新会首先发生在内存缓冲区中，当内存缓冲区中的页被更新时，它会被标记为脏页。一旦脏页被驱逐出内存缓冲区，则必须把它同时写入到磁盘和固态盘缓冲区中。

基于温度的替换模块需要根据温度统计信息来确定被驱逐的页，温度信息是以数据区域为单位进行统计，并以哈希表的形式保存在内存中，哈希表中记录了在过去时间内所有访问过的数据区域的温度信息。系统会实时维护各个数据区域的温度信息，每当一个页被从固态盘或磁盘中读取时，它的标识符也会被传输给基于温度的替换模块，从而更新温度信息。为了能够反映负载访问模式随着时间变化的情况，作者对温度信息采用了标准的老化策略[ZhouCL04]，即在经过一定次数的访问以后，就对所有页的温度值进行折半处理，从而使得历史数据的重要性要比近期新鲜数据的重要性低一些。



图[SSD-as-second-level-cache] 把固态盘作为数据库系统的二级缓存时的体系架构

10.2.1.2 作为数据仓库的更新缓存

文献[AthanassoulisACGS10]提出了采用固态盘作为数据仓库的更新缓存，用来存储数据仓库的实时更新数据。随着市场竞争的加剧和企业应用的不断发展，传统的数据仓库已经无法很好地满足企业的日常应用需求。在传统数据仓库中，OLTP 系统中的数据是以一定的周期（每周一次或每月一次）批量加载到数据仓库中的，分析人员对大量历史数据进行汇总后得到有价值的信息，从而辅助管理者做出企业决策。但是，目前快速变化的市场环境，要求管理者必须能够及时获得各种 OLTP 系统中的数据，做出实时决策，尽量避免错过商机。因

此，实时数据仓库已经成为越来越多的企业的必然选择。实时数据仓库要求采用 CDC（变化数据捕捉）方法，从 OLTP 系统中实时捕捉变化数据，并立即传播到数据仓库中。数据仓库当前主要保存在磁盘中，而且，数据仓库拥有海量的数据，在未来一段时间内，仍然会把磁盘作为首要存储设备。磁盘将会同时承受来自实时数据仓库应用的两种典型负载：查询负载和更新负载。查询负载主要是对数据表的顺序扫描，而更新负载则是大量的随机写操作。这两种负载同时存在于磁盘中，而且磁盘采用的是就地更新的方式，这必然会导致数据的读写冲突，恶化查询性能和更新性能。因此，把更新操作单独保存到独立于磁盘的其他存储介质上，可以很好地解决该问题。由于内存容量有限而且价格昂贵，不适合作为大量更新操作的缓存。而且，如果把大量的内存分配给数据仓库作为更新缓冲区，会明显恶化查询操作的性能，因为，这意味着能够用作频繁访问数据和中间临时数据的缓存会变得更少，而为频繁访问数据和中间临时数据提供缓存，是提升数据仓库性能的关键措施。因此，闪存设备很自然就成为一种理想的选择。文献[AthanassoulisACGS10]提出采用闪存设备对更新进行缓冲，核心思想是：(1)在一个闪存的缓冲区中对到达的更新进行缓存；（2）在查询过程中即时考虑这些被缓存的更新，从而使得查询可以看到最新的数据；（3）当缓冲区满的时候，把缓存的更新数据迁移到磁盘中。

10.2.2 5 分钟规则

当把闪存固态盘作为磁盘和内存之间的缓存时，需要把一些数据放置到固态盘中。一般而言，把数据放到固态盘中是基于以下其中一种考虑因素：

（1）保证响应时间：固态盘比磁盘的访问速度快许多，对于一些对响应速度要求比较高的数据，可以考虑放入到固态盘中，从而提高针对这些数据的读写速度；

（2）获得经济收益：在一个存储设备上的数据访问代价可以用“每秒每次访问的价格”来衡量，而固态盘的数据访问代价要比磁盘低许多，因此，把频繁访问的数据放入到固态盘中，可以明显降低数据访问代价，带来经济收益。

出于第一种考虑因素的情形，相对较少，大部分情形都是基于第二种考虑因素，即目的在于获得经济收益。以获得经济收益为目的时，需要把频繁访问的数据放入到固态盘中，那么，到底应该把什么访问频度（即多长时间访问一次）的数据放入固态盘呢？著名的“5 分钟规则”[GrayP87]可以很好地回答这个问题。“5 分钟规则”解决了把什么访问频度的数据放入内存的问题，其中，内存被视作存储在磁盘中的数据的缓存。当我们把固态盘视作存储在磁盘中的数据的缓存时，“5 分钟规则”同样可以解决把什么访问频度的数据放入固态盘的问题。

“5 分钟规则”是由 Gray 和 Putzolo 在 1987 年提出的[GrayP87]，并在 1997 年做了修改和更新[GrayG97]。在 Gray 和 Putzolo 的方法中，首先比较两个方面的代价：（1）把一个记录（或者页）永久存放在内存中需要的代价；（2）每次访问这个记录（或者页）进行磁盘 I/O 操作的代价。在比较之后就可以做出决定，把什么访问频度的数据放入内存，从而进一步确定合理经济的设备购买预算，即在总的设备购买预算中，为内存和磁盘分配多少购买预算。Gray 和 Putzolo 的方法的计算结果，与磁盘和内存的价格、带宽、容量等参数的具体值有关，不同的设备性能和价格参数，会得到不同的结果。二人当时在 1987 年采用的是 Tandem 公司的计算机设备，根据这些设备性能和价格参数得到的结果是 400 秒，即把那些访问频度超过“每 400 秒被访问 1 次”的数据放入内存，为了便于问题阐述，二人把 400 秒简化成 5 分钟，由此得到“5 分钟规则”。下面简单了解一下 Gray 和 Putzolo 在 1987 年是如何计算得到“5 分钟规则”的，计算过程采用的是当时的设备性能和价格参数[GrayP87]。

一个磁盘一般每秒钟可以处理 15 个随机访问，磁盘的价格在 15K 美元左右，因此，每秒钟每个磁盘访问的价格就是 1K 美元，即磁盘访问代价是 1K 美元/访问/秒。为了支持磁盘数据访问所需要的额外的 CPU 和通道的开销大约是 1K 美元，即 1K 美元/访问/秒。因此，

磁盘访问的实际代价是 2K 美元/访问/秒。一个 1MB 的内存的开销大约是 5K 美元，因此，1KB 内存的开销大约是 5 美元。如果把 1KB 的数据保存到内存中，就不再需要访问磁盘，可以每秒节省一次磁盘访问（Tandom 中一次磁盘访问可以传输 4KB 数据，因此，从磁盘读取 1KB 数据只需要一次磁盘访问），也就节省了 2K 美元的开销，所需要的开销只是 5 美元的内存开销。由此，可以计算得到收支平衡点是：每隔 2000/5=400 秒访问一次。因此，对于任何 1KB 的记录，如果被访问的频度大于每 400 秒（或简化成 5 分钟）被访问一次，就应该放入内存。

收支平衡点不仅和设备的性能和价格相关，而且与页的大小相关，页越大，收支平衡点越小，反之，则越大，比如对于 100 字节的页，收支平衡点是 1 个小时，对于 4KB 的页，收支平衡点是 2 分钟。当页的大小超过磁盘每秒传输数据量时，比如，一个页是 100KB，而磁盘每秒只能传输 4KB 数据，这个时候就必须采用实际的磁盘传输数据量作为计算收支平衡点的依据，而不是采用页的大小作为计算依据。

Gray 和 Putzolo 在 1997 年对“5 分钟规则”进行了修改补充，并给出了收支平衡点的计算公式[GrayG97]：

BreakEvenIntervalinSeconds= (PagesPerMBofRAM / AccessesPerSecondPerDisk) × (PricePerDiskDrive / PricePerMBofRAM)。

其中，PagesPerMBofRAM 表示 1MB 的 RAM 内存所包含的页的数量，AccessesPerSecondPerDisk 表示计算机设备中每个磁盘每秒钟可以处理的随机访问次数，PricePerDiskDrive 表示每个磁盘的价格，PricePerMBofRAM 表示 1MB 的 RAM 内存的价格。

Gray 和 Putzolo 采用 1997 年的 DELL TPC-C 测试基准的相关设备参数进行了收支平衡点的计算，具体参数如下：

PagesPerMBoIRAM = 128 pages/MB（每页大小是 8KB）

AccessesPerSecondPerDisk = 64 access/sec/disk

PricePerDiskDrive = 2000 美元/disk (9GB + controller)

PficePerMBofRAM = 15 $/MB_DRAM

基于上述设备性能和价格参数计算得到的收支平衡点是 266 秒（大约是 5 分钟）。

“5 分钟规则”是和具体的设备性能和价格参数相关的，只有在特定参数下，收支平衡点才会是 5 分钟。随着制造工艺的不断提高，磁盘和内存的性能在不断改善，价格也在不断下降，导致设备性能和价格参数都在不断变化，因此，5 分钟规则适用的情形也会不断变化。当要求收支平衡点大约是 5 分钟时，在不同时期，需要采用的页面大小也不同。在 1987 年，以当时的设备性能和价格，页面大小是 1KB 时，收支平衡点大约是 5 分钟；在 1997 年，页面大小是 8KB 时，收支平衡点大约是 5 分钟；而在 2007 年，Graefe 根据当时的设备性能和价格参数，计算得到的结果是，当页面大小是 64KB 时，收支平衡点大约是 5 分钟（334 秒）[Graefe07]。Graefe 认为，5 分钟规则对于页面大小在 64KB 以上的闪存固态盘是可以适用的。

“5 分钟规则”只是一种确定数据放置策略的理论名称，并非一定要求收支平衡点是 5 分钟，在实际应用中，可以根据“5 分钟规则”理论的收支平衡点计算公式，计算出把什么访问频度的数据放入闪存固态盘。

10.2.3 FaCE

Kang 等人[KangLM12]提出了 FaCE（Flash as Cache Extension），它把闪存作为一个可恢复数据库的扩展缓存，不仅可以改进事务吞吐量，而且可以缩短从系统失败中恢复的时间。FaCE 采用了两种新的算法来管理闪存缓存中的数据页，即多版本 FIFO（First In First Out）和小组二次机会策略。FaCE 是在开源数据库 PostgreSQL 的基础之上实现的，在 TPC-C 测试基准的大量实验发现，对于一个 OLTP 应用而言，如果使用少量闪存作为数据库系统的缓

存，甚至要比把数据库内容全部存放到闪存中具有更高的性能。

10.2.3.1 相关研究工作的不足之处

Canim 等人[CanimMBRL10]针对 IBM DB2 数据库提出的扩展缓存原型系统，采用了贯穿写的策略，当一个脏页被从 DRAM 缓冲区中驱逐出来时，它会被同时写入到闪存缓存和磁盘中，由此可以看出，闪存缓存只是为读操作提供了缓存效果，却并没有减少磁盘写操作的数量。此外，在闪存中持久性地维护缓存元数据的高昂代价，也恶化了系统整体性能。

惰性清洗方法——LC（Lazy Cleaning），是另外一种可以考虑的缓存方法。在这种方法中，在数据页从 DRAM 缓冲区出来到磁盘的路上，会被拦截下来放置到闪存缓存中，如果该数据页是脏的，就采用“回写”（write-back）策略，即采用一个后台进程，当脏页的数量超过预设的门槛值的时候，就把脏页从闪存刷新到磁盘。LC 方法使用 LRU-2 替换算法来管理闪存缓存，因此，替换闪存缓存中的一个数据页，会引起许多随机读和随机写操作。此外，由于不能把闪存缓存融入到数据库恢复机制中，因此，就需要采用检查点机制不断把闪存缓存中的脏页更新到磁盘。而已有研究显示，检查点的额外代价是非常高的。

另外，上述方法通常需要探测确定冷热数据，或者通过静态统计工具，或者采用运行时监测方法，但是，无论哪种方法都存在一些缺点。静态方法无法处理访问模式的变化；动态方法具有较高的运行时开销，并且需要在磁盘和闪存之间迁移数据，而且，当工作负载是更新操作比较集中时，动态方法不会取得好的效果。

10.2.3.2 FaCE 的设计思想

FaCE 的基本设计思想是，使用一种统一的方式而不需要人工方法（比如使用静态统计工具）或者算法（比如运行时监测）的干预，来确定把哪些数据放置到哪个设备上，从而构建一个有效的、由磁盘和闪存构成的混合存储系统。因此，FaCE 提出把闪存固态盘用作 DRAM 缓冲区的一个扩展缓存，这样它就可以和 DRAM 缓冲区自身的数据页替换机制一起工作，而不需要提供额外的机制来区分确定冷数据和热数据，也不需要监测数据访问模式，因此，只需要很少的运行时开销，未来数据访问模式的预测质量也不会对结果产生影响。

FaCE 可以克服上面介绍过的其他方法存在的两个不足之处：（1）其他方法没有减少磁盘写操作的数量；（2）其他方法没有把闪存缓存纳入数据库恢复机制。具体而言，FaCE 可以做到以下三点：

第一，FaCE 实现了闪存写操作优化以及磁盘访问的减少。DRAM 缓冲区通常会为写操作和顺序访问操作提供同样的性能。和 DRAM 不同的是，闪存的性能会随着操作类型（读或写）和数据访问模式（顺序访问呢或随机访问）的不同而表现出较大的差别。对于许多现在的闪存固态盘而言，随机写操作要比顺序写操作慢大约一个数量级。FaCE 提供了闪存感知的策略来管理闪存缓存，它可以被设计成完全独立于 DRAM 缓冲区管理策略。FaCE 会把小的随机写转换成大的顺序写，因此可以充分利用闪存的高顺序带宽和内部并行性，从而实现高吞吐量。在 FaCE 中，当一个冷数据页被驱逐出 DRAM 缓冲区的时候，它可能获得第二次机会，继续停留在闪存缓存中一段时间。如果这个冷数据页停留在闪存缓存中时再次变得热起来，就可以快速地从闪存缓存中直接交换回 DRAM 缓冲区，而不需要从磁盘中读取。如果一些数据页确实是冷的，并且长期内都没有再被访问，那么，它最终就不会被 DRAM 缓冲区或者闪存缓存所保存。可以看出，FaCE 可以有效减少磁盘访问，并且可以优化写操作性能。

第二，FaCE 充分利用了闪存的非易失性，对数据库持久性的范畴进行了扩充，包含了闪存缓存中的数据页的情形。一个数据页被从 DRAM 缓冲区中驱逐出来，如果进入闪存缓存中，则也会被认为已经被持久地保存到数据库中。因此，闪存缓存中的数据页，可以用来最小化系统恢复开销，加速系统从失败中恢复的过程，并且以最低的代价获得事务原子性和持久性。在恢复期间需要访问到的大多数数据页，都可以在闪存缓存中得到，这就可以明显

减少系统恢复时间。FaCE 恢复管理器提供的机制，可以完全支持数据库恢复和持久性元数据管理。

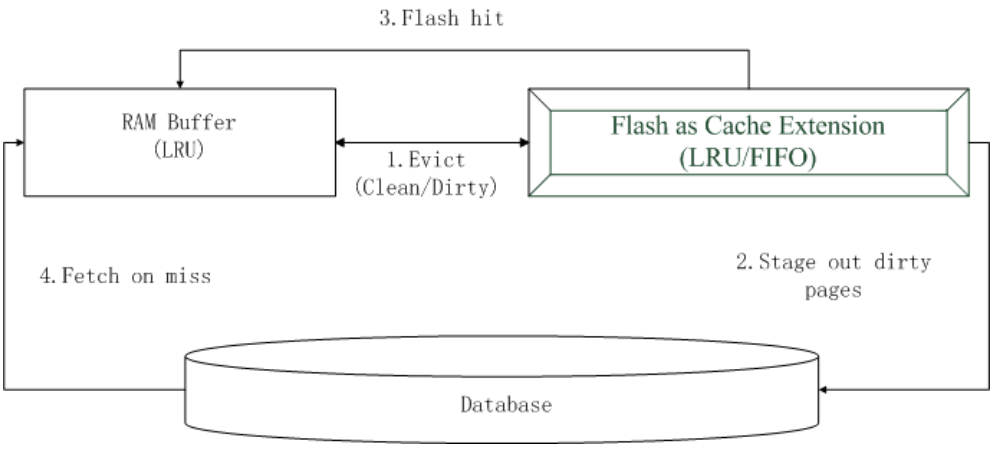
第三， FaCE 是一个具有较低开销的框架，因为，它使用闪存驱动器作为 DRAM 缓冲区的扩展，而不是作为磁盘的替代品。闪存的这种使用方式是和 DRAM 缓冲区紧密耦合的，闪存缓存管理和 DRAM 缓冲区管理可以很好地一起工作。FaCE 不需要人工或者算法的方法从冷数据中剥离出热数据，因此，运行时的开销是非常低的。FaCE 并不会增加磁盘的通信量，因为，它不会为了更高的缓存命中率而在闪存和磁盘之间迁移数据。

10.2.3.3 FaCE 的基本框架

图[FaCE-overview]给出了 FaCE 的基本框架，从中可以看出，在这种混合存储系统中，闪存充当了 DRAM 缓冲区的扩展缓存。FaCE 的不同组成部分之间的交互情况，具体如下：

- 当数据库服务器请求一个数据页时，如果该页不在 DRAM 缓冲区中，就到闪存缓存中搜索。DRAM 缓冲区维护了一个关于闪存缓存中所有页的列表信息，可以支持快速查找。如果这个页在闪存中，就从闪存中读取；否则，就从磁盘中读取。
- 当一个页被交换出 DRAM 缓冲区的时候，需要根据不同的情形采取不同的动作，这主要取决于这个页是脏页还是干净页。如果它是干净页，就会被直接丢弃或者被保存到闪存缓存中。如果它是一个脏页，就会被回写到闪存或者磁盘，或者同时回写到闪存和磁盘中。
- 当一个页被交换出闪存缓存时，也需要根据不同的情形采取不同的动作，主要取决于该页是脏页还是干净页。如果它是干净页，这个页就会直接被丢弃；如果是脏页，就会被写入到磁盘，除非它在从 DRAM 缓冲区驱逐出来时早已经写入到磁盘。

很显然，从上面描述的主要交互过程可以看出，FaCE 需要解决的基本问题是，什么时候以及哪些数据应该暂时停留或者离开闪存缓存。有许多方式可以解决这个问题，而且对于数据库的整体性能可能具有很大的影响。例如，当一个数据页发生 DRAM 缓冲区“脱靶”时（即无法在 DRAM 缓冲区中找到该页），这个页会从磁盘中被读取出来放入到 DRAM 缓冲区时，这个时候就不用把该页同时放入到闪存缓存中，因为，当 DRAM 缓冲区中存在数据副本时，闪存缓存中的副本是不会被访问的。由于这个原因，只有当一个数据页从 DRAM 缓冲区中被驱逐出来时，才有可能进入到闪存缓存中。可以看出，在这一点上，FaCE 和其他把闪存作为缓存的方法具有很大的不同，在其他方法中，一些“温数据页”（不是热数据页）被从磁盘中读取出来以后，可能会被首先保存到闪存缓存中，如果在闪存缓存中停留期间，这个温数据页变成热数据页，该页会从闪存缓存调入到 DRAM 缓冲区。而在 FaCE 中，这是不行的，FaCE 不允许把一个数据页在从磁盘到 DRAM 缓冲区的道路上被拦截下来放入到闪存缓存，而只允许一个数据页从 DRAM 缓冲区到磁盘的道路上被拦截下来放入到闪存缓存，也就是说，当 DRAM 缓冲区中的一个数据页变成冷数据页的时候，可能会在离开 DRAM 缓冲区后被拦截下来放入到闪存缓存中。而且，只有通过这种方式进入到闪存缓存中的数据页，以后才可能有机会被再次交换回到 DRAM 缓冲区中，也就是说，当闪存缓存中的数据页再次变成热数据页的时候，会获得机会被交换回 DRAM 缓冲区。



图[FaCE-overview] FaCE 的基本框架

10.2.3.4 FaCE 的设计策略

FaCE 必须要解决三个方面的设计问题：第一，当一个被从 DRAM 缓冲区驱逐出来以后，必须同时写入到闪存缓存和磁盘，还是只需要写入到闪存呢？第二，采用什么样的闪存缓存管理策略？第三，当一个数据页被从 DRAM 缓冲区驱逐出来时，是否要对脏页和干净页进行区分对待？

（1）采用回写策略而不是贯穿写策略

当一个脏页被从 DRAM 缓冲区中驱逐出来的时候，可以选择两种写入策略，即贯穿写（write-through）和回写（write-back）。所谓的贯穿写是指，把该页同时写入到磁盘和闪存缓存中，而回写则是指只被写入到闪存缓存中。当采用贯穿写方式时，可以保证磁盘和闪存缓存二者都获得最新版本的数据页；当采用回写方式时，只有闪存缓存获得了最新版本的数据页，而磁盘中的数据页原始版本并没有发生变化，这意味着磁盘中包含的是过期的数据页，直到闪存中的数据页被驱逐出来写入到磁盘中时，磁盘才可以获得最新版本的数据页。

对于读操作而言，上面这两种方式在效果上是等价的。但是，对于写操作而言，在许多情形下，贯穿写的代价明显要高于回写的代价。这里分两种情形来讨论：

第一种情形：在一段时期内，一个脏页只被读入 DRAM 缓冲区中一次，然后被驱逐出 DRAM 缓冲区以后，再也没有被访问。在这种情形下，如果采用贯穿写，当该脏页被从 DRAM 缓冲区中驱逐出来时需要写入到闪存和磁盘，就会包含一个闪存写操作和一个磁盘写操作；而如果采用回写方式，则需要暂时把该该脏页放入闪存缓存，这时需要一个闪存写操作，由于此后一段时间都没有被再次使用，该页变成冷页，会被驱逐出闪存缓存，写入到磁盘，这时又会产生一个磁盘写操作。可以看出，在这种情形下，两种写方式都包含一个闪存写操作和一个磁盘写操作，二者代价是相同的，只不过对于回写方式而言，磁盘写操作并没有和闪存写操作同时发生，而是延迟一段时间才执行磁盘写操作。

第二种情形：在一段时期内，一个脏页被驱逐出 DRAM 缓冲区，后来又变成热数据页再次被读入 DRAM 缓冲区，而且这种情况反复发生多次，最后一次才彻底变成冷数据页被最终写入到磁盘，假设总共被从 DRAM 缓冲区中驱逐了 N 次。对于这种情形，如果采用贯穿写策略，每次当这个脏页被从 DRAM 缓冲区中驱逐出来时，都需要一个磁盘写操作和一个闪存写操作。而如果采用回写策略，它会把数据页先放入闪存缓存，变成热页时，直接从闪存缓存交换回 DRAM 缓冲区，如此多次反复，直到最终彻底变成冷数据页后才被从闪存中驱逐出来写入到磁盘。可以看出，采用回写方式，只需要 N 次闪存写操作和一次磁盘写操作，而贯穿写则需要 N 次闪存写操作和 N 次磁盘写操作。也就是说，采用回写方式明显减少了写操作代价。出于这个原因，FaCE 采用了回写策略，而不是贯穿写策略。

（2）基于多版本 FIFO 的替换

闪存缓存空间是有限的，在对闪存缓存进行管理时，很容易想到利用 LRU（Least Recently Used）替换算法。尽管 LRU 算法是一种被 DBMS 广泛采用的缓存管理策略，但是，对于闪存缓存而言，并不能获得好的性能。LRU 算法会为闪存缓存中的所有数据页构建一个 LRU 链表，链表的 MRU 位置存放了最近被访问过的页，链表的 LRU 位置存放了距离当前时间最远的被访问过的页。在选择驱逐页时，算法会在 LRU 链表的 LRU 位置选择一个页作为驱逐页，而不会考虑该页在闪存中的物理位置。这就意味着，每个页替换都会引起一个闪存随机写操作。对于闪存而言，随机写这种模式相对而言是效率较低的，涉及较大的执行代价，通常比顺序写操作慢一个数量级，因为，随机写操作会引起后续的许多垃圾回收和块擦除操作。

因此，FaCE 没有采用 LRU 算法来管理闪存缓存，而是采用了 FIFO（First In First Out）策略的一个变种——多版本 FIFO。虽然 FIFO 策略通常被认为性能不如 LRU 算法，但是，FIFO 策略有其自己独特的优点，使得它被应用到闪存缓存机制中时可以获得较好的性能。因为，在 FIFO 策略中，所有新到达的数据页都会追加的方式、被顺序地压入到闪存缓存队列的尾部，这就意味着，所有的闪存写操作都是顺序写操作。FIFO 所表现出来的这种独特的写模式，和闪存特性非常匹配，可以帮助闪存缓存产生较好的性能。

多版本 FIFO 和传统的 FIFO 策略不同的地方在于，FIFO 策略只允许在队列中存在同一个数据页的唯一副本，而多版本 FIFO 策略则允许一个数据页的一个或者多个版本同时存在于闪存缓存中。在多版本 FIFO 中，当需要替换闪存缓存中的数据页时，会从闪存缓存队列头部选择一个驱逐页，新进来的数据页直接放入到闪存缓存队列的尾部。接下来，一个页是干净页还是脏页，会引起不同的操作。如果进来的数据页是脏页，那么，它就会被无条件地压入队列，不会采取额外的操作来移除可能存在的之前的版本，从而不会引起随机写操作。如果进来的数据页是干净页，那么，只有闪存缓存中还不存在这个页的同样副本，才可以把这个页放入队列。一个数据页从闪存缓存队列中移除时，它被写入到磁盘的前提条件是，该页是脏页，并且是闪存缓存中的最新版本，这样做可以避免冗余的磁盘写操作。

多版本 FIFO 还采用和二次机会同样的策略，即当一个有效页被驱逐出闪存缓存时，如果这个页在停留在闪存缓存期间时已经被访问过，它会被给予第二次机会，而不是被丢弃（对于干净页而言）或者刷新到磁盘（对于脏页而言）。

（3）缓存干净页和脏页

实际上，对于一个数据页而言，如果它是干净页，被放置到闪存缓存中以后，只要被访问过一次以上，就可以获得收益。比如说，对于一个干净页 p 而言：

（a）把页 p 放入闪存缓存时的读取代价：在采用回写策略时，数据页 p 被驱逐出 DRAM 缓冲区时，首先被放入闪存缓存，需要一个闪存写操作，读取数据页时需要一次闪存读操作，最终在某个时刻被驱逐出闪存缓存时，由于是干净页，可以直接丢弃，不用写入磁盘。因此，总代价是一个闪存写操作和一个闪存读操作。

（b）不采用闪存缓存的读取代价：当数据页 p 被驱逐出 DRAM 缓冲区时，由于是一个干净页，则可以直接丢弃，不用写入到磁盘，当需要读取这个页时，需要从磁盘中读取该页，需要一个磁盘读操作。因此，总代价只包括一个磁盘读操作。

对于现在的绝大多数闪存固态盘产品和磁盘产品而言，一个磁盘读操作的代价，要高于一个闪存写操作和闪存读操作代价之和。所以可以得出结论，干净页被放置到闪存缓存中以后，只要被访问过一次以上，就可以获得收益。

由此可以认为，是否缓存一个干净页，应该根据这个页被缓存以后到它被驱逐出缓存之前，再被访问至少一次的可能性有多大。

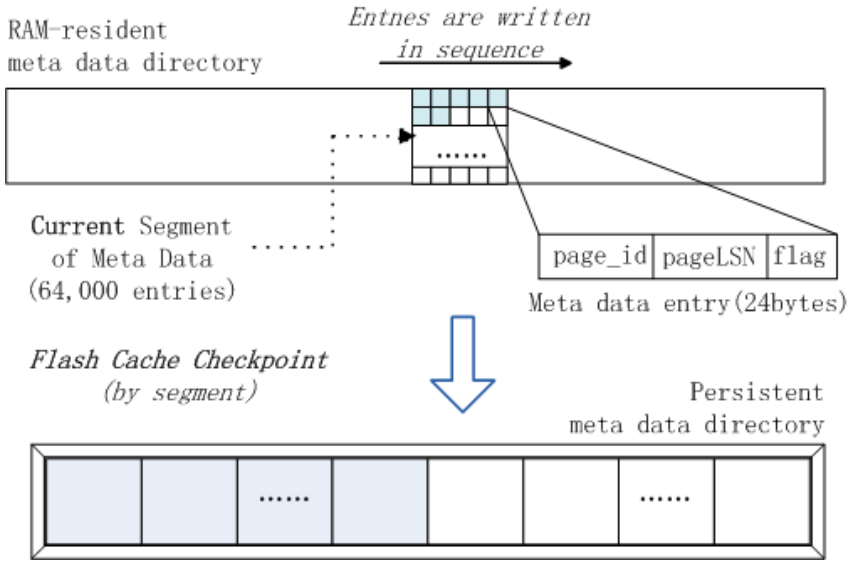
对于一个脏页而言，当它被驱逐出 DRAM 缓冲区时，把它保留在闪存缓存中总是可以带来收益的。因为，如果不保存到闪存缓存，就会立即引起一个磁盘写操作。此外，上面曾

经讨论过一种情形，即在一段时期内，一个脏页被驱逐出 DRAM 缓冲区，后来又变成热数据页再次被读入 DRAM 缓冲区，而且这种情况反复发生多次，最后一次才彻底变成冷数据页被最终写入到磁盘。对于这种情形而言，采用回写策略把脏页缓存到闪存中，可以把许多次磁盘写操作转变成闪存写操作，大大提高了系统整体性能。因此，当采用回写策略时，脏页应该比干净页具有更高的优先级进入到闪存缓存中。因此，FaCE 采用了多版本 FIFO 策略，允许一个数据页的多个版本进入闪存缓存。

10.2.3.5 FaCE 的恢复

对于数据库系统而言，一旦发生系统失败，必须保证数据库能够恢复到一致的状态，从而保证事务的原子性和持久性。数据库恢复机制中的关键技术就是“写前日志”，即在事务提交之前要求必须把日志“强制写入”到稳定的存储介质上。对于 FaCE 而言，当一个脏页被从 DRAM 缓冲区中驱逐出来的时候，它的所有日志记录都会被写入到稳定的存储器上（闪存或者磁盘）。就事务的持久性而言，一旦一个脏页被写入到磁盘或者闪存中，这个页就被认为已经永久性地保存到了数据库当中，因为，磁盘和闪存都是非易失性的存储介质。对于 FaCE 系统而言，闪存的非易失性保证了，在发生系统失败的时候，总是有可能从闪存缓存中恢复数据。因此，FaCE 利用了存储在闪存缓存中的数据页，来服务于双重目的，即缓存扩展和数据库恢复。

在 FaCE 中，当发生数据库的恢复时，存储在闪存缓存中的数据页可能比磁盘中的页的版本更新，这个时候，恢复数据库一致性就必须使用闪存缓存中的数据页。即使闪存和磁盘中的数据页是完全同步的，也应该使用闪存缓存中的数据页，因为闪存比磁盘的访问速度更快。



图[FaCE-recovery] FaCE 的元数据维护

这里唯一需要处理的问题就是，要保证在系统发生失败的时候，与闪存缓存相关的元数据能够存活下来。否则，如果丢失了元数据，当系统重启的时候，将无法从闪存缓存中获得相应数据，数据库就无法恢复到一致性状态。

图[FaCE-recovery]显示了 FaCE 的元数据维护方法。在 FaCE 中，每条元数据包含了三个属性，即 *page_id*, *pageLSN* 和 *flag*，其中，*page_id* 表示页编号，*pageLSN* 表示页的逻辑扇区编号，*flag* 是一个标记位，用来表示该页是否在闪存缓存中。元数据的变化会被收集在内存中，并且作为一个个较大的分段持久性地写入到闪存缓存中。这种类型的元数据管理对于 FaCE 而言是比较高效的，因为，进入一个闪存的数据页总是以时间顺序被放入到队列的末端，它的元数据目录也是如此，而且，元数据是以分段的形式批量更新到闪存中，而不是

一条条元数据分次更新。但是，需要指出的是，由于系统发生失败的时候，仍然会有少量元数据在内存中，这些内存中的元数据就会发生丢失。不过这个不会成为问题，因为，这部分元数据只对应着闪存缓存中的少量数据页，因此，只需要扫描闪存中的一少部分数据页，就可以快速恢复系统。

10.3 基于数据访问模式的混合存储系统

在未来较近一段时期内，磁盘仍将是企业应用的主要数据存储介质，但是，随着闪存技术的不断发展，闪存设备已经表现出逐步取代磁盘的趋势。在这个取代的过程中，一种比较大的可能性是，闪存设备和磁盘在一段时期内共存，在存储的层级结构中处在同一个级别上，共同构成混合存储系统。

10.3.1 基于对象放置顾问的混合存储系统

文献[CanimBMLR09]指出，如果固态盘和磁盘一起组成混合存储系统，把随机访问的数据转移到固态盘中，留在磁盘中的数据通常都是顺序访问的，那么，就可以获得明显的性能收益，主要包括以下几个方面：

（1）提高了数据库系统的整体性能：通过把那些频繁地、以随机方式被访问的数据存储到固态盘上面，可以大幅改善随机操作的性能，因为，固态盘不存在机械部件，随机访问速度要比磁盘快。

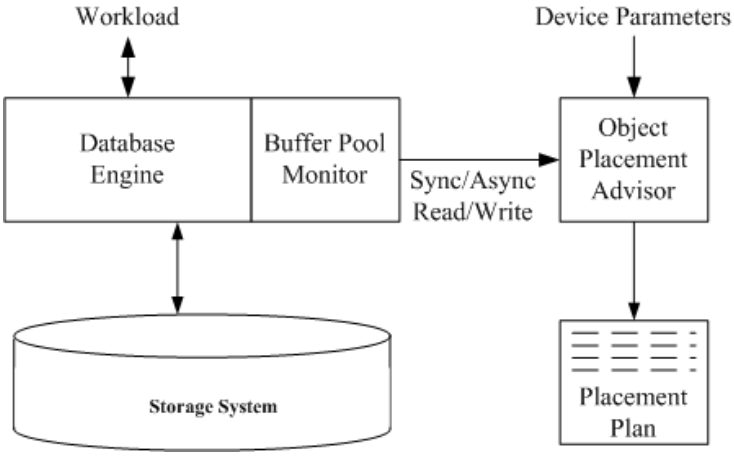
（2）提高了采用“短行程”策略的磁盘的空间利用率：在一些企业级的数据库系统实施方案中，为了提升数据库的性能，会采用 “短行程”（short-stroking）策略，即有意只使用磁盘存储空间的一部分来存储数据库数据，从而使得寻址时间尽量小，保证系统响应时间，这种以牺牲存储空间来换取高性能的做法在特定应用场合下是比较经济高效的。如果需要更多的存储空间，就需要不断增加磁盘，而不是使用磁盘中剩余的可用空间。由于多个磁盘的磁头可以并行寻址，因此，仍然可以获得较好的性能。很显然，采用 “短行程”（short-stroking）策略后，磁盘空间利用率较低。如果采用固态盘和磁盘的混合存储系统，则可以有效改善磁盘空间利用率。因为磁盘中保留下来的数据大都是顺序访问的，而磁盘具有很高的顺序访问吞吐量，顺序访问数据不需要增加额外的寻址开销，因此，在限定的系统响应时间内，就可以顺序访问更多的数据，从而可以更多地利用“短行程”策略中剩余的磁盘空间，提高了采用“短行程”策略时的磁盘空间利用率。

（3）减低了能耗：随机负载被分流到固态盘以后，就减轻了磁盘的负担，继而降低了系统总体能源开销，因为固态盘比磁盘更加节能。

虽然把数据存放到固态盘中可以带来明显的收益，但是，毕竟固态盘空间是有限的，只能把一部分数据存储带固态盘中，因此，就涉及到数据放置策略问题，即把哪些对象放置到固态盘中可以最大程度地改进系统的性能。文献[CanimBMLR09]提出了一个工具称为“对象放置顾问”，用来明智地确定数据对象的存放位置。通过收集负载信息，对象放置顾问会使用启发式算法（比如贪婪算法或动态规划算法），来确定需要放置到固态盘中的对象的列表。

对象放置顾问生成最优的数据放置策略的过程包括两个阶段：特征提取阶段和决策阶段。在特征提取阶段可以获得数据对象在运行时的访问特征统计信息，利用这些统计信息就可以计算出把一个数据对象从磁盘转移到固态盘中可以带来的性能收益。决策阶段根据这些性能收益数据生成对象放置计划。图[object-placement-advisor]给出了对象放置顾问的工作原理。数据库引擎负责处理来自各种应用的查询负载，附着在数据库引擎上面的缓冲区监控工具，执行特征提取任务，它会不断监控记录各种性能指标，包括一个页从磁盘读取到内存缓冲区所消耗的时间，以及把一个脏页写回到磁盘所消耗的时间等等。各种监控到得性能指标数据（特征信息）会被传送给对象放置顾问。对象放置顾问就会根据这些特征信息，生成一个“代价-收益图”，显示在不同的存储空间容量限制下不同的放置策略所带来的相应的性能收益。数据库管理员就可以参考“代价-收益图”来决定需要购买多大容量的固态盘。对象放

置顾问还可以生成候选放置对象列表，具体方法是：根据提取到得工作负载的特征信息和设备的特性信息，计算每个对象的预期的性能收益，除以对象的尺寸后得到单位尺寸的性能收益，对不同的对象按照单位尺寸性能收益进行降序排序；然后，使用一个启发算法从高到低依次选择数据对象，直到没有更多的对象可以放置到固态盘空间中；最终，这些被选中的对象就构成了候选放置对象列表。数据库管理员可以根据候选放置对象列表，把相关的数据对象从磁盘转移到固态盘上面，从而获得较好的性能收益[CanimBMLR09]。



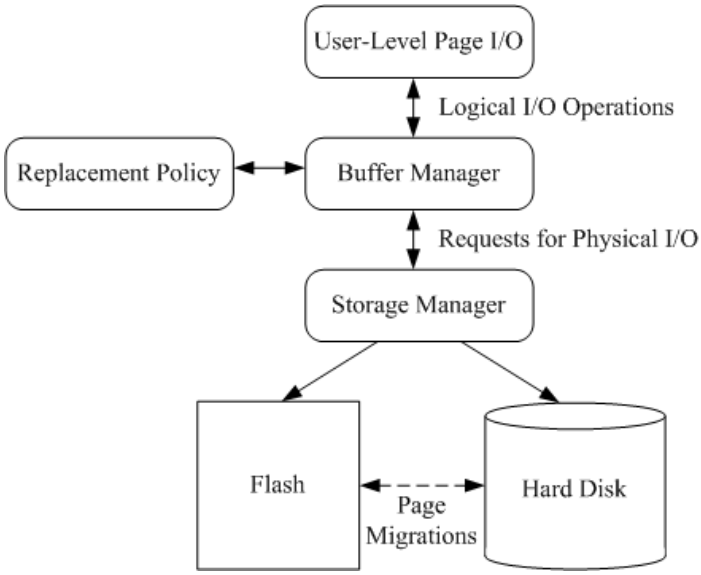
图[object-placement-advisor] 对象放置顾问示意图

但是，正如作者在其后续研究工作[CanimMBRL10]中指出的那样，基于对象放置顾问的混合存储系统具有下面的缺陷：

- 用户必须运行一个负载特征提取工具来获得统计数据，并且，随着数据在固态盘上的反复进出，又要重新进行代价昂贵的特征提取；
- 对象放置决定是以整个表或索引为粒度，而在一些情况下，只有表中的一部分数据是被频繁访问的，我们可能只希望把表中的一些片段放入固态盘。

10.3.2 基于双状态任务系统的混合存储系统

文献[KoltsidasV08s]提出在存储层次结构的同一个层面同时使用闪存和磁盘的混合存储系统。图[flash-and-HDD-hybrid-system]显示了闪存和磁盘混合存储系统的体系架构。缓冲区管理负责对分配给数据库使用的内存缓冲区进行管理和维护，这个过程需要调用高效的替换算法。存储管理器负责根据具体的负载情况来确定一个页的最优存放位置，或者放入磁盘中，或者放入闪存设备中，但是，在某一个时刻，一个数据页只能存在于其中一种存储介质中。其中，如果一个页具有比较集中的读操作负载，它会被写入到闪存中，而对于具有比较集中的写操作负载的页，则会被写入到磁盘中。通过这种方式，读操作的速度要比只采用磁盘的系统快，写操作的速度要比只采用闪存的系统快。作者设计了一系列在线算法，可以确定一个页的最优放置方式，从而可以根据工作负载的变化动态决定数据页的存放位置。

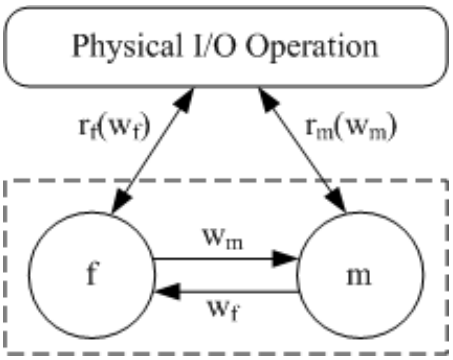


图[flash-and-HDD-hybrid-system] 闪存和磁盘混合存储系统的体系架构

混合存储系统中需要解决两个关键难题：（1）如何根据历史负载模式来预测一个页的未来负载，如果预测不准，就会带来大量不合理的 I/O 开销；（2）当一个页的负载模式发生变化时（比如，从读操作比较集中的负载变成写操作比较集中的负载，或者反过来），如何自适应地调整这个页的放置位置。

对于未来负载模式预测问题，作者为一个页建立了负载跟踪记录，并设计了相应的缓冲区替换策略，可以根据历史负载情况合理确定替换页，从而让未来可能使用到的页最大可能性地保留在缓冲区中。

对于把一个页放置在闪存还是磁盘中这个问题，作者把它建模成一个双状态任务系统[BorodinLS87]。如图[two-state-task-system]所示，双状态任务系统包含了两个状态 f 和 m，其中，f 表示数据页在闪存中，m 表示数据页在磁盘中。从闪存和磁盘中读取一个页的代价分别是 r_f 和 r_m，把一个页随机写入闪存和磁盘的代价分别是 w_f 和 w_m。从一个状态转移到另一个状态的代价，等于把一个页写入到另一个存储介质的代价，这里之所以不用考虑读取代价，是因为当状态转移发生的时候，即在决定把一个页写入到另一个存储介质的时候，这个页已经被读取出来。对于双状态系统的状态转移问题，已经有不少成熟的研究成果，比如文献[BorodinLS87][BlackS89]，作者采用了和文献[NathK07]类似的在线算法。



图[two-state-task-system] 双状态任务系统示意图

这个混合存储系统的缺陷是：（1）它假设固态盘可以容纳整个数据库，但是，在当前的企业应用中，往往可以提供足够容量的磁盘，而能够提供的固态盘的容量还是有限的；（2）没有区分顺序和随机访问模式。对于磁盘而言，顺序访问模式的代价要比随机访问模式的代价小得多，因此，把所有的访问模式都假设为随机访问模式，会导致数据更容易被存储到固

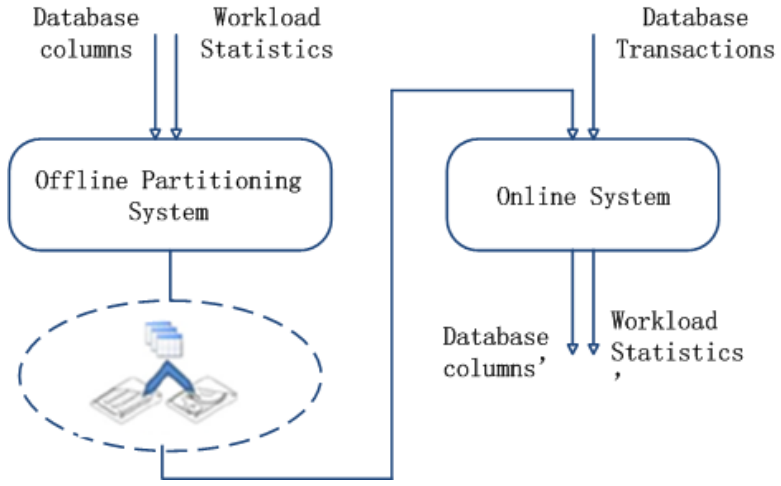
态盘而不是磁盘中；（3）没有考虑垂直分区，而实践已经证明，即使对于传统的磁盘而言，垂直分区都可以明显获得数据库性能的提升。针对这些问题，Clementsen 等人[ClementsenH10]提出了一种基于垂直分区的混合存储系统。

10.3.3 基于垂直分区的混合存储系统

10.3.3.1 系统概述

基于垂直分区的混合存储系统[ClementsenH10]可以有效克服基于双状态任务系统的混合存储系统的一些缺陷，它包括两个分离的子系统，即离线分区系统和在线系统（如图[vertical-partition-for-SDD-HDD]所示）。离线分区系统负责确定应该把数据库的每个列放置在固态盘还是磁盘中。在离线系统确定的分区的基础上，在线系统可以运行查询，但是不会转移数据。在线系统也会收集工作负载的统计信息，在后续进行分区操作时提供给离线系统使用。Clementsen 等人只研究了离线分区，而没有研究实时动态分区，也就是没有设计在线算法根据实时负载特征在磁盘和固态盘之间动态迁移数据库的列。离线系统的主要任务就是确定数据库的列在磁盘和固态盘这两种存储介质上的放置位置，并且在某一时刻，只能存储在其中一种存储介质上。

离线分区系统把数据库和负载统计信息作为输入，然后使用这些信息来把每个数据库列分配给某个存储介质。系统的输出是一个的列到存储介质的映射集合。当后续到达的工作负载和先前用于统计的负载的类型比较类似时，系统就可以获得最优的 IO 性能。分区算法运行结束后，系统会根据列到存储介质的映射集合，把各个列放置到相应的存储介质中。离线算法运行的频率，取决于许多因素，比如工作负载的变化、数据表的变化以及特定环境的性能需求等。



图[vertical-partition-for-SDD-HDD] 基于垂直分区的混合存储系统

10.3.3.2 离线分区系统

10.3.3.2.1 工作负载统计

为了把数据库分区到两个驱动器中，就需要获得数据库工作负载的统计信息。为了获得关于哪个请求需要 IO 操作的准确信息，只看数据库事务是不够的，必须同时考虑系统使用的缓存策略，因为，并非所有的读操作都会引发一个磁盘读操作。而且，离线分区系统将会考虑每个列的放置策略，因此，统计信息的粒度是列。换句话说，所需要的统计负载的信息将会包括每个列上的 IO 请求操作的不同类型的数量。

此外，数据库内容是动态变化的，这也让问题变得更加复杂。例如，一个存储在固态盘上的列可能会随着时间增长。由于固态盘的容量是有限的，因此就很有必要考虑列增长的问题。系统应该尽可能地避免把一个列同时分散在两种存储介质上。列的扩充可以通过在存储介质上扩充列文件来实现。但是，如果列是存储在固态盘上面，并且固态盘已经完全满了，

那么，该列后续的扩充内容就会被存储到磁盘上面。这就意味着，一个随着时间不断增加的列，可能会被分散存储到两种存储介质上面，尤其是新增的数据（即那些从数据库被分区以后新增加的数据）可能会被保存到磁盘上，即使离线系统最初把这个列保存到固态盘上。因为一个列新增的页可能会被保存到磁盘上，而这个列原来的数据页被保存在固态盘上，因此，对原来页和新增页进行读写的代价就会不同。为了解决这个问题，系统需要获得每个列原来数据和新增数据的工作负载统计信息。

10.3.3.2.2 NP-完全问题的证明

现在来证明分区问题是一个 NP-完全问题，证明的思路是把分区问题转换成背包问题，即证明在分区问题和背包问题二者之间存在一个多项式的映射$\Pi \to \Pi'$，而背包问题已经被证明是一个 NP-完全问题，因此，可以证明分区问题也是一个 NP-完全问题。

这里首先描述一下背包问题，然后说明如何把分区问题映射到背包问题。一旦这个映射构建完成，就可以使用背包问题的已有解决方案来解决分区问题。

背包问题已经被证明是一个 NP-完全问题。给定一个物品的集合，每个物品都有物理尺寸和价值，背包问题就是为一个限定尺寸的背包确定一个可以装入背包的物品子集，使得这些物品的总价值最大。形式化地，可以定义如下：

条件：假设有一个有限集合 U，对于每个$u \in U$，都存在一个尺寸$s(u) \in Z^+$和一个值$v(u) \in Z^+$，同时假设存在正整数 B 和 K。

问题：是否存在一个子集$U' \in U$ 使得$\sum_{u \in U'} s(u) \le B$，并且使得$\sum_{u \in U'} v(u) \ge K$。

分区问题的目标是把数据库的每个列分配到两种存储介质（磁盘和固态盘）中的一个。实际上，可以把这个问题看成是把列的一个子集分配给固态盘，这样分区问题就比较符合背包问题。因为这里假设可以提供足够容量的磁盘来容纳所有的数据库内容，而只能提供有限的固态盘空间来容纳部分数据库内容。

这里定义一个变量 I/OR(u)，表示把列 u 放置到固态盘中带来的 IO 开销的减少值，它的计算公式如下：

I/OR(u)= $\cos t_{SSDcolumn}(u) - \cos t_{HDDcolumn}(u)$ （式[IORU]）

其中，$\cos t_{SSDcolumn}(u)$表示如果把列 u 放置在固态盘上面，在列 u 上面的工作负载的 IO 代价。它的定义如下：

$$\cos t_{SSDcolumn}(u) =$$

$$(N_{oir}(u) + N_{osr}(u)) \times (\cos t_{SSDread} + \cos t_{SSDrand}) + (N_{air}(u) + N_{asr}(u)) \times \cos t_{HDDread}$$

$$+ (N_{air}(u) + N_{acs}(u)) \times \cos t_{HDDseek}$$

$$+ N_{ow}(u) \times (\cos t_{HDDwrite} + \cos t_{HDDrand}) + N_{aw}(u) \times (\cos t_{HDDseek} + \cos t_{HDDwrite})$$

上面每个变量符号的含义如表[parameters-table-01]所示，其中，$N_{oir}(u)$, $N_{air}(u)$, $N_{bcs}(u)$, $N_{acs}(u)$, $N_{osr}(u)$, $N_{asr}(u)$, $N_{ow}(u)$, $N_{aw}(u)$都是系统收集的统计信息。

表[parameters-table-01] 符号的含义

符号	含义
$\cos t_{SSDread}$	在固态盘上的一个页读操作的代价
$\cos t_{SSDwrite}$	在固态盘上的一个页写操作的代价
$\cos t_{SSDrand}$	在固态盘上随机访问的代价
$\cos t_{HDDread}$	磁盘上的一个页读操作的代价
$\cos t_{HDDwrite}$	磁盘上的一个页写操作的代价
$\cos t_{HDDseek}$	磁盘上的一个寻址操作的代价
$N_{oir}(u)$	从索引查找中读取的原来的页的数量
$N_{air}(u)$	从为新增页构建的索引中读取的页的数量
$N_{bcs}(u)$	在对列进行任何新增数据之前对列执行的扫描的次数
$N_{acs}(u)$	在对列进行第一次新增数据之后对列执行的扫描的次数
$N_{osr}(u)$	从列扫描中读取的原来页的数量
$N_{asr}(u)$	从列扫描中读取的新增页的数量
$N_{ow}(u)$	针对原来页的写次数
$N_{aw}(u)$	针对新增页的写次数

注意，上面的代价公式是一个比实际代价更高的估计，因为，这里假设所有的新增数据都被写入到磁盘中。但是，如果固态盘中还有剩余空间时，一些新增的页可能会被写入到固态盘中，这种情况下，刚才的假设就是错误的。但是，通常情况下，分区操作会最大程度地利用固态盘，因此，固态盘中的可用空间会很小。

$\cos t_{HDDcolumn}(u)$ 表示如果把列 *u* 放置在磁盘上面，在列 *u* 上面的工作负载的 IO 代价，它的定义如下：

$$\cos t_{HDDcolumn}(u) =$$

$$(N_{oir}(u)+N_{air}(u))\times(\cos t_{HDDseek}+\cos t_{HDDread})+(N_{osr}(u)+N_{asr}(u))\times\cos t_{HDDread}$$

$$+(N_{bcs}(u)+N_{acs}(u))\times\cos t_{HDDseek}$$

$$+(N_{ow}(u)+N_{aw}(u))\times(\cos t_{HDDseek}+\cos t_{HDDwrite})$$

其中，$\cos t_{HDDread}$ 表示在找到一个页的位置后，从磁盘中读取这个页的代价。

定理 1：分区问题是一个 NP-完全问题。

证明：让 Π 表示分区问题，让 Π' 表示背包问题。映射 $T:\Pi\to\Pi'$ 是非常直观的，可以在多项式时间内完成。*U* 是数据库中所有列的集合，U' 表示放置到固态盘中的列的集合，并且把这些列放入到固态盘中以后，可以带来最大程度的 IO 开销减少，*s*(*u*)表示列 *u* 的尺寸，

$v(u)$可以采用公式[IORU]来计算。

10.3.3.2.3 解决分区问题的动态规划算法

分区问题就是要确定数据库的列最佳放置位置，即放置到磁盘上还是固态盘上。采用动态规划算法解决分区问题的基本思想是“分而治之”。首先，把一个问题分解成可能的最小子问题，并解决这个较小的问题；然后，使用最小子问题的解，来构建下一个较大问题的解；这个过程一直重复，直到找到原来问题的解。采用这种思路确定数据库列的最佳放置位置的问题，就可以采用下面的方法来解决。首先，把固态盘容量限制为 1 个页，然后在其中放入一个列集合，并且使得这个集合中的列可以在给定的工作负载下面具有最低的 IO 总代价。在许多情形下，这时的列集合中可能包含 0 个列，因为，通常所有的列都大于 1 个页。然后，我们把固态盘容量增加到 2 个页，再去做同样的事情。这个过程一直持续，直到我们到达实际的固态盘容量。采用从最小到最大的固态盘容量这种渐进的方式来求解的原因是，当求解更大固态盘容量下的问题时，可以利用基于较小的固态盘容量得到的结果。

10.4 基于语义信息的系统框架hStoreage-DB

Luo 等人[LuoLMCZ12]为 DBMS 存储管理设计和实现了一个基于语义信息的混合存储系统框架 hStorage-DB，它定义了对于存储 IO 而言至关重要的语义信息（比如内容类型和访问模式），并且传递给存储管理层。存储管理层根据收集到的语义信息，把访问请求分成不同的类型，并为每种类型分配一个合适的 QoS(Quality of Service)策略，这些策略是被底层的存储系统所支持的，从而使得每个请求都可以由合适的设备来为之提供服务。

10.4.1 其他混合存储系统的不足

混合存储系统需要在闪存和磁盘两种存储介质之间分配数据，可以采用两种方法：（1）基于数据库管理员的方法，根据数据库管理员的知识和经验，在不同的设备之间分配数据；（2）基于访问模式的方法，设计监测程序，探测发现数据访问模式，并根据访问模式确定数据在不同设备之间的分配，前面已经介绍过的多种基于数据访问模式的混合存储系统就采用这种方法。但是，上述两种方法都具有一定的不足之处。

对于基于数据库管理员的方法而言，具有以下局限性：

（1）需要大量的人工操作。数据库管理员必须根据经验知识来人工确定数据在不同设备之间的分配，比如，索引会被频繁地随机访问，有一些小表也会被频繁地使用，由于闪存固态盘具有较高的随机读性能，因此，数据库管理员通常会把索引和小表存储到固态盘中，而其他数据则存放在相对廉价的磁盘设备中。

（2）数据粒度会变得非常粗糙，无法获得好的性能。数据库管理员会同等地对待所有被分配同一个表的请求，但是，对于一个表的不同部分的访问，可能具备不同的访问模式，因此，以表为粒度无法很好地区别一些访问模式。为了改进这一点，数据库管理员可能会尝试把一个表分割成多个分区，对每个分区加以区别对待，但是，这种分区方式往往是一种静态的结果，而实际应用负载的访问模式是不断变化的，因此，这种改进仍然无法保证获得最优的性能结果。

（3）根据负载的访问模式来配置数据放置策略，是一种静态的策略，很难适应 IO 需求的动态变化。

对于基于数据访问模式的方法而言，特定的访问模式，只有通过一段时间的监测才能观察到，因此，当数据库负载表现出较长时间的稳定性的时候，它可以表现出较好的性能。但是，基于数据访问模式的方法在以下情况下不能获得好的性能：

（1）对于高度动态变化的负载和公用的短生命周期的数据（比如临时数据）来说，根本没有足够长的时间来收集数据和发现正确的访问模式；

（2）在共享缓存设备上的并发数据流之间存在比较复杂的交互，会引起一些不可预测

的访问模式，数据访问模式监测方法很难探测发现访问模式，或者探测得到的模式的准确性较低；

（3）无法确定一些重要的信息，比如内容类型和数据生命周期，这些信息是和访问模式无关的，但是，这些信息在混合存储系统中对于做出正确的数据放置策略而言是非常重要的。

10.4.2 hStoreage-DB 的设计思路

hStoreage-DB 在设计混合存储系统时，采用了不同的思路——基于语义信息的方法，可以克服基于数据库管理员的方法和基于数据访问模式的方法的局限性。

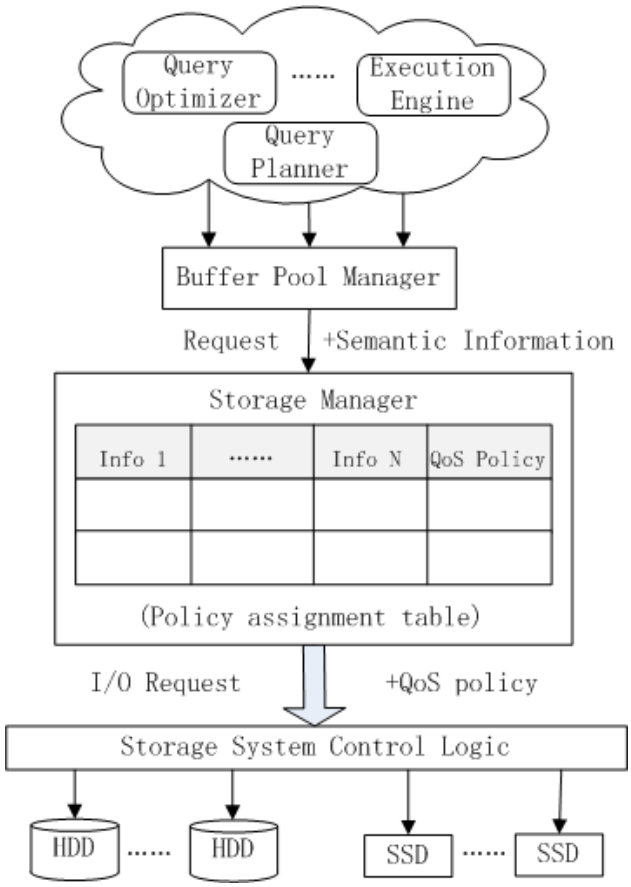
一个数据库的 IO 请求有不同的语义信息，比如内容类型和访问模式等信息可能都不尽相同。hStoreage-DB 会根据请求的语义信息对请求进行分类，然后，为不同类型的请求关联不同的 QoS 策略，并根据这个请求的 QoS 策略来选择合适的设备为之提供服务。比如，如果把 hStoreage-DB 设计成一个包含两级存储架构的系统，即最低层是磁盘，位于磁盘上面的一层是固态盘，作为磁盘的缓存，那么，对于 hStoreage-DB 而言，为了实现缓存优先级的目的，可以按照以下方式对语义信息进行分类：

- 内容类型：这里主要关注三种内容类型，即常规表、索引和临时数据。常规表定义了数据库的内容，它们消耗了大部分数据库存储空间，索引可以用来加速对常规表的访问，临时数据(比如哈希表)是在一个查询过程中产生的，在查询完成的时候就会被删除掉。
- 访问模式：它指的是一个 IO 请求所具备的特征，是由查询优化器来决定的。对一个表的访问模式可以是顺序扫描，也可以是随机访问的。对一个索引的访问模式通常是随机访问的。

hStoreage-DB 会把对于存储 IO 而言至关重要的语义信息（比如内容类型和访问模式），和请求一起传递给存储管理层。存储管理层根据收集到的语义信息，会把请求分成以下几个类型：（1）顺序请求；（2）随机请求；（3）临时数据请求；（4）更新请求。然后，上述四种不同类型的 IO 请求，会被分配不同的 QoS 策略。

图[hStorage-DB]给出了 hStoreage-DB 的体系架构，从中可以看出，hStoreage-DB 的具体实现过程如下：

- 首先，把那些对于存储 IO 而言至关重要的语义信息（比如内容类型和访问模式），和请求一起传递给存储管理层。
- 其次，存储管理层接收到一个请求以后，会先查看语义信息，然后根据预先定义的一个规则集合，为不同语义的请求分配不同的 QoS 策略，并且底层的存储系统可以支持这些 QoS 策略。然后，一个请求和与之关联的 QoS 策略一起被发送到底层的混合存储系统。
- 最后，在收到一个请求的时候，存储系统首先提取得到与之关联的 QoS 策略，然后使用一个正确的机制，根据这个请求的 QoS 策略来选择合适的设备为之提供服务。



图[hStorage-DB] hStorage-DB 的体系架构

10.4.3 hStoreage-DB 的技术挑战和解决方案

要向实现上述的设计思路，hStoreage-DB 会面临以下两个方面的主要挑战：

- 如何为每个 QoS 关联一个合适的 QoS 策略？为了使得混合存储系统能够采用正确的机制为一个请求提供相应的服务，需要设计一些规则来完成从语义信息到 QoS 策略的高效的映射，即根据收到的请求中包含的语义信息就可以立即找到与之匹配的合适的 QoS 策略。这里必须系统地考虑多个因素，包括查询类型的多样性，不同查询计划的复杂性，以及并发执行多个查询带来的问题。
- 如何把每个请求的 QoS 策略传递到底层的混合存储系统？一个 DBMS 存储管理器通常是一个接口，这个接口会把一个 DBMS 数据请求转换成一个 IO 请求。在转换期间，所有的语义信息（比如内容类型和访问模式等）都会被剥离，只会留下一个请求的物理布局信息，包括逻辑块地址、类型（读或写）、大小和实际数据（如果是一个写操作）。这就意味着，在现有的 DBMS 存储管理器实现方式下，在 DBMS 和存储系统之间存在一个“语义鸿沟”，无法传递语义信息。

为了解决第一个挑战，hStoreage-DB 针对不同的语义类型设计了 5 种规则，可以实现从语义信息到 QoS 策略的快速映射。针对第二个挑战，hStoreage-DB 把 QoS 策略嵌入到原来的 IO 请求中，并通过一个块接口分发给底层的混合存储系统，存储系统收到请求后，首先提取得到与该请求关联的 QoS 策略，然后使用一个正确的机制，根据这个请求的 QoS 策略来选择合适的设备为之提供服务。

10.4.4 QoS 策略

10.4.4.1 QoS 策略概述

QoS 策略为一个存储系统提供了一个高层次的服务抽象。通过一个良好定义的 QoS 策

略，一个存储系统可以有效地提供自己的能力，同时不需要对用户暴露设备层次的细节。一个 QoS 策略，或者可以是性能相关的，比如延迟和带宽需求，或者也可以是性能无关的，比如可靠性需求。一个存储系统的所有 QoS 策略，取决于它的硬件资源和这些资源的组织形式。与请求关联的 QoS 策略必须能够被底层的存储系统所理解和支持，如果使用一个存储管理系统无法理解的 QoS 策略，那就是毫无意义的。这就意味着，如果一个 DBMS 被转移到另外一个采用了完全不同 QoS 策略的存储系统上面，那么，就必须采用一个不同的存储管理模块，设计不同的语义信息和 QoS 策略之间的映射。

10.4.4.2 混合存储系统的 QoS 策略

这里以一个实例来演示 hStoreage-DB 是如何支持自动存储管理的。在本实例中，hStoreage-DB 是一个包含两级存储架构的系统，最低层是磁盘，位于磁盘上面的一层是固态盘，作为磁盘的缓存。由于是一个两级存储架构，就需要确定数据被存放到缓存（固态盘）中的优先级，因此，QoS 策略被定义为一个关于缓存优先级的集合。

具体而言，可以把 QoS 策略定义成一个三元组{*N*,*t*,*b*}，其中，*N*>0，0≤*t*≤*N*，并且 0%≤*b*≤100%，参数 *N* 定义了优先级的总数量，更小的值意味着更高的优先级，即获得更好地被缓存的机会。参数 *t* 是一个针对“不缓存”的优先级门槛值，也就是说，被一个优先级大于等于 *t* 的请求所访问的块，将不会有任何被缓存的机会，这里需要注意的是，优先级的值越大，优先级越低。如果把“不缓存”的优先级门槛值设置为 *t*=*N*-1，那么，就会存在两种不缓存的优先级，即 *N*-1 和 *N*，这里称优先级 *N*-1 为“不缓存和不驱逐”，称优先级 *N* 为“不缓存和驱逐”。参数 *b* 是和一个特定的优先级相关的配置参数，这个特定优先级被称为“写缓冲区”优先级。注意，写缓冲区优先级和写缓冲区不是一个概念，写缓冲区是一个缓存空间，而写缓冲区优先级并不是一个专用的空间，而是一个专门的优先级，在这种优先级下，一个更新请求可以比其他类型的请求更加优先获得缓存空间。参数 *b* 用来决定把多少缓存空间用来作为写缓冲区，当更新操作占用的数据空间超过 *b* 时，写缓冲区中的所有内容都会被刷新到磁盘中。对于 OLAP 类型的工作负载，可以设置 *b* 为 10%。对于每个到达的请求，存储系统首先抽取出与之关联的 QoS 策略，然后调整所有被访问块的放置位置。例如，如果一个块被一个关联了高优先级的请求所访问，那么，如果这个块还不在缓存中，它就会被调入缓存，当然，这个过程也会取决于已经在缓存中的其他块的优先级。因此，在实际应用中，一个请求的优先级会被最终转换成为所有被访问的块的优先级。

10.4.4.3 针对不同类型的请求的 QoS 策略

前面讲过，如果把 hStoreage-DB 设计成一个包含两级存储架构的系统，为了实现缓存优先级的目的，可以把语义信息分成四类：（1）顺序请求；（2）随机请求；（3）临时数据请求；（4）更新请求。下面介绍如何为四种不同类型的 IO 请求分配不同的 QoS 策略。

10.4.4.3.1 顺序请求

在混合存储系统 hStoreage-DB 中，缓存设备是一个固态盘，底层的存储是磁盘，磁盘可以提供较好的顺序访问性能，这意味着，把接受顺序访问的块放入缓存就不会有什么收益。因此，针对顺序请求，hStoreage-DB 按照规则 1 为之分配 QoS 策略。

规则 1：所有的顺序请求都会被分配一个“不缓存和不驱逐”优先级。

一个请求具备“不缓存和不驱逐”优先级，具有两个方面的含义：（1）如果被访问的数据不在缓存中，就不会把它分配到缓存中；（2）如果被访问的数据已经在缓存中，它的优先级是由之前的请求确定的，不会受到本次请求的影响。换句话说，具备这种优先级的请求，不会影响现有的存储数据布局。

10.4.4.3.2 随机请求

随机请求可以从缓存中获得收益，但是，最终收益的大小取决于数据重用的可能性。如果一个块被一次随机访问以后，就再也不会被随机访问，那么，就不应该为它分配缓存空间。

因此，针对随机请求，hStoreage-DB 按照规则 2 为之分配 QoS 策略。

规则 2：在查询计划树当中的较低层次的操作发出的随机请求，要比在查询计划树当中的较高层次的操作发出的随机请求，具备更高的缓存优先级。

10.4.4.3.3 临时数据请求

临时数据是在查询执行过程中生成的，查询结束以后就会被删除，生命周期较短。临时数据包含两个阶段：生成阶段和消费阶段。在生成阶段，临时数据会被一个写数据流生成；在消费阶段，临时数据会被一个或多个读数据流所访问。在消费阶段的结束，临时数据会被删除，以释放存储空间。因此，对于临时数据的这个两阶段特点，应该在临时数据被生成的时候就立即把它放入缓存，并且在生命周期结束时候立即把它驱逐出缓存。因此，针对临时数据请求，hStoreage-DB 按照规则 3 为之分配 QoS 策略。

规则 3：所有针对临时数据的读写请求，都被给予最高优先级。删除临时数据的命令，被分配“不缓存和驱逐”优先级。

一个请求具备“不缓存和驱逐”优先级，包含两个方面的含义：（1）如果被访问的数据不在缓存中，它就不会被加入到缓存中；（2）如果被访问的数据已经在缓存中，它的优先级会被改变成“不缓存和驱逐”优先级，从而使之在以后的时间里会及时被驱逐。因此，具备“不缓存和驱逐”优先级的请求，不允许数据进入缓存，只让数据尽快离开缓存。

10.4.4.3.4 更新请求

hStoreage-DB 为写操作分配一部分缓存，从而使得这些写操作不需要直接访问磁盘。使用一个写操作缓冲区以后，所有的写数据将会被首先存储到固态盘缓存中，并被异步地刷新到磁盘中。因此，针对更新请求，hStoreage-DB 按照规则 4 为之分配 QoS 策略：

规则 4：所有更新请求将会被分配“写缓冲区”优先级。

10.5 本章小结

本章内容首先介绍了闪存代替磁盘作为特种数据的存储介质，包括存储事务日志、回滚段、中间结果；然后，介绍了闪存作为介于磁盘和内存之间的缓存，包括作为数据库系统的二级缓存和作为数据仓库的更新缓存，并介绍了“5 分钟规则”，可以用来确定把哪些数据放入固态盘；接下来，介绍了基于数据访问模式的混合存储系统，包括基于对象放置顾问的混合存储系统、基于双状态任务系统的混合存储系统和基于垂直分区的混合存储系统；最后，介绍了基于语义信息的系统框架 hStoreage-DB，它定义了对于存储 IO 而言至关重要的语义信息，并且传递给存储管理层。存储管理层根据收集到的语义信息，把访问请求分成不同的类型，并为每种类型分配一个合适的 QoS 策略，从而使得每个请求都可以由合适的设备来为之提供服务。

10.6 习题

1、 分析用闪存来存储事务日志为什么可以明显改进事务吞吐量。
2、 分析当把磁盘替换成闪存固态盘时，对于块嵌套循环连接、归并排序连接、Grace 哈希连接和混合哈希连接这四种连接算法而言，在性能方面分别会带来什么影响。
3、 说明闪存作为数据库仓库的更新缓存时的主要作用。
4、 阐述 5 分钟规则的含义。
5、 阐述 FaCE 的设计思想。
6、 阐述基于语义信息的混合存储系统框架 hStorage-DB 的设计思路。

第 11 章 闪存数据库实验环境的搭建

在闪存数据库的研究中，需要对各种算法的性能进行验证。算法性能验证方法主要包括两大类：（1）在闪存模拟器这种仿真环境下进行性能验证；（2）修改开源数据库相关模块代码，在真实数据库和闪存设备环境下进行验证。本章内容分别介绍上述两种方法。

11.1 使用闪存模拟器开展实验

闪存模拟器为闪存相关研究提供了一种仿真的实验环境，简单易用。目前已经存在多种闪存模拟器，这里以国内研究人员开发的 Flash-DBSim 为实例，介绍闪存模拟器的体系架构以及如何利用闪存模拟器开展相关实验（主要是缓冲区替换算法实验）。

11.1.1 Flash-DBSim 介绍

Flash-DBSim[SuJXCY09]是由中国科学与技术大学知识与数据工程实验室开发的闪存模拟器，它为各种算法性能验证提供了仿真环境，可以模拟闪存技术研究中出现的各种实验环境。该模拟器可以根据上层应用的需要模拟出不同特性的固态盘，如闪存的读写代价不一致和写前擦除等特性。Flash-DBSim 使用的程序设计语言为 c++，开发工具为 visual studio 2008。很多已有的研究（比如缓冲区替换算法 CCF-LRU 和 AD-LRU）都采用了该模拟器进行算法性能的比较。

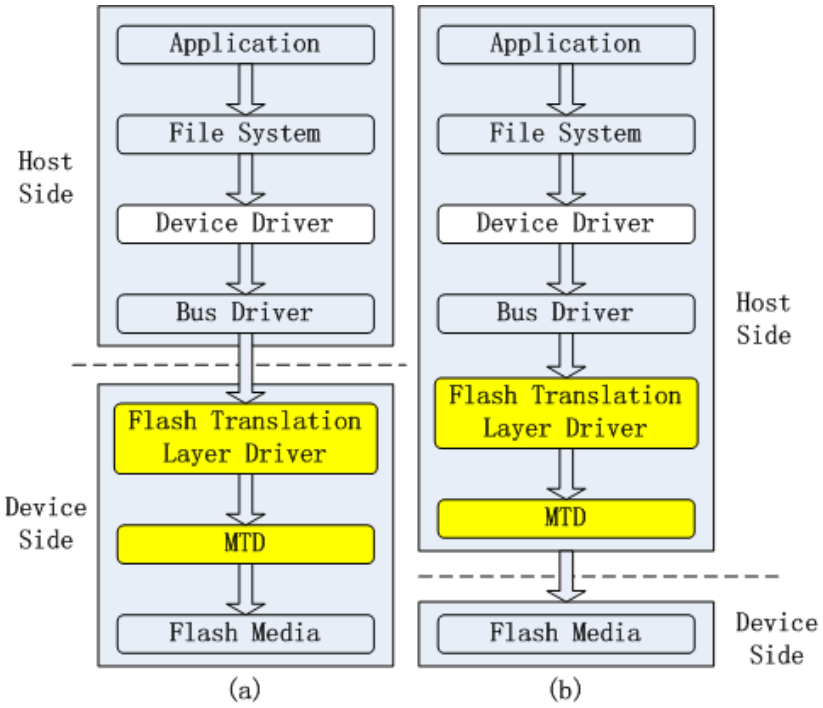
Flash-DBSim 具有如下特点：

- 模块化：采用了模块化的设计方式，自下而上包含了三个主要模块，即虚拟闪存设备模块 (Virtual Flash Device Module, VFD)，存储技术设备模块 (Memory Technology Device Module, MTD)以及闪存转换层模块 (Flash Translation Layer Module, FTL)。每个模块中都封装了一些具有代表性的算法，大大减轻了研究人员开展相关实验时的冗余工作，如果需要替换其中的一个或多个算法，只需要修改相关的模块即可。
- 接口化：所有模块之间使用通用接口进行交互，由于接口的不变性，因此，某个模块的内容修改不会影响到其他模块。
- 可扩展性：研究人员可以根据自己的实际研究需要，对模拟器的各个模块内容进行修改，只要保证遵循 Flash-DBSim 各模块的接口定义，新的代码就能够被配置到 Flash-DBSim 中运行。
- 易配置性：不同的闪存技术研究需要配置不同的实验环境，为了方便用户配置模拟器，Flash-DBSim 尽可能简化了接口数量，从而使得只需要使用少量的代码就可以对整个模拟器环境进行配置。

11.1.2 Flash-DBSim 体系架构

闪存存储系统包括两个关键组件，即闪存技术设备模块 MTD 和闪存转换层 FTL。MTD(Memory Technology Driver)提供了对闪存的读、写和擦除操作的函数，它直接控制物理闪存芯片。FTL（Flash Translation Layer）是架构在 MTD 层之上的，用来实现地址翻译、空间分配和垃圾回收等操作，从而把一个闪存设备模拟成一个块设备，使得文件系统和用户应用可以像访问磁盘一样访问闪存设备。

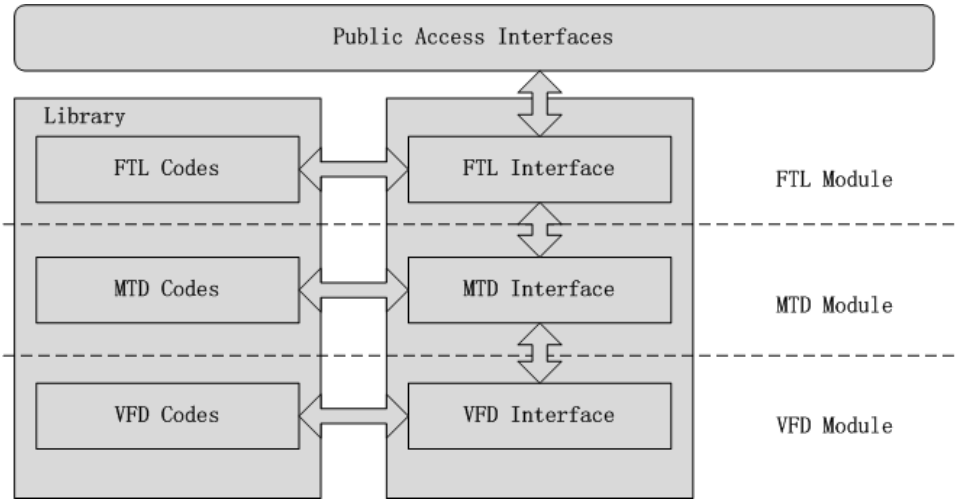
图[Flash-DBsim-two-ways]显示了实现一个闪存存储系统的两种不同方式。第一种方式是，MTD 和 FTL 被封装在闪存设备（比如固态盘）中，如图[Flash-DBsim-two-ways] (a)所示。第二种方式是，MTD 和 FTL 在闪存设备上面的软件系统中实现，如图[Flash-DBsim-two-ways] (b)所示。Flash-DBSim 采用了第一种实现方式，即封装在一个闪存设备中，只不过在 Flash-DBSim 中，这个闪存设备并非真实的设备，而是一个虚拟的设备。



图[Flash-DBsim-two-ways] 闪存存储系统的两种实现方式

图[Flash-DBsim-architecture]给出了 Flash-DBSim 的体系架构，包括以下几个组件：

- 虚拟闪存设备模块：即 VFD（Virtual Flash Device）模块，可以提供虚拟闪存的一些特性，比如 IO 延迟和擦除限制，它为其他模块提供了基本的操作接口，比如读、写和擦除。
- 闪存技术设备模块：即 MTD（Memory Technology Device）模块，可以通过接口来控制 VFD 模块。MTD 模块隐藏了 NAND 和 NOR 闪存之间的差别，因此，通过使用 MTD 模块中定义的相同的接口，上层的模块可以访问各种不同类型的虚拟闪存。
- 闪存转换层模块：即 FTL（Flash Translation Layer）模块，可以实现地址翻译、空间分配和垃圾回收。该模块通过调用 MTD 模块的接口来访问下面的虚拟闪存。
- 库函数：包含许多源代码，可以直接用于闪存研究工作，减少了研究人员的冗余工作。所有这些源代码都遵循了相关模块的定义。
- 公共访问接口：该组件是开放给所有用户的，它对用户隐藏了所有底层细节。



图[Flash-DBsim-architecture] Flash-DBSim 的体系架构

11.1.3 Flash-DBSim 的下载和简要说明

Flash-DBSim 官方网站（http://kdelab.ustc.edu.cn/flash-dbsim/）提供了关于该模拟器的详细说明和源代码下载。

Flash-DBSim 官方网站提供了两个不同版本的 Flash-DBSim 下载内容：

- “运行组件”版本：如果只是使用 Flash-DBSim 进行闪存研究实验，请下载此版本，然后，根据网站上的说明进行配置并开展相关实验；
- “源代码+示例”版本：如果需要对 Flash-DBSim 系统进行二次开发，请下载此版本，很显然，该版本也可以用来进行闪存研究实验。

这里将假设读者下载的 Flash-DBSim 文件版本为"源代码+示例”，版本号为“1.6.0.32”，从网站下载得到的原始压缩文件为“flash-dbsim-source-1.6.zip”。

Flash-DBSim 使用的程序设计语言为 c++，开发工具为 visual studio 2008，因此，读者对压缩文件 flash-dbsim-source-1.6.zip 进行解压后，可以得到名为“flash-dbsim-source-1.6”的文件夹，该文件夹下面包含一个解决方案文件和两个工程文件夹：

- FlashDBSimDll.sln 文件：Visual Studio 的解决方案文件；
- FlashDBSimDll 文件夹：包含 Flash-DBSim 源代码，可以进行二次开发；
- FlashDBSimDll_Sample 文件夹：包含了一个测试 LRU 算法性能的实例，可以在这个工程下修改相关代码，设计和实现其他闪存相关算法并测试性能。

11.1.4 使用 Flash-DBSim 模拟器开展相关实验

这部分内容首先简单介绍 FlashDBSimDll_Sample 工程中各个文件的作用，然后介绍如何设置模拟器的参数，接下来给出了一个实例来演示如何测试 LRU 算法的性能，最后，简单介绍如何测试自己的缓冲区替换算法性能。

11.1.4.1 使用 Visual Studio 开发工具打开解决方案

从 Flash-DBSim 官方网站下载得到原始压缩文件“flash-dbsim-source-1.6.zip”以后，解压缩得到名为“flash-dbsim-source-1.6”的文件夹，该文件夹下面包含一个解决方案文件 FlashDBSimDll.sln 和两个工程文件夹。

使用 visual studio 2008/2010 打开 FlashDBSimDll.sln，就可以看到一个解决方案（名称为 FlashDBSimDll）下面包括两个工程，即 FlashDBSimDll 和 FlashDBSimDll_Sample。如果需要进行二次开发，可以根据 Flash-DBSim 官方网站的详细说明来修改 FlashDBSimDll 工程下面的相关代码；如果需要进行实验，可以直接在 FlashDBSimDll_Sample 工程下面根据需要编写相关代码，运行程序，测试算法性能。这里不需要进行二次开发，只是开展相关实验，因此，只需要修改 FlashDBSimDll_Sample 工程下面的代码，不需要理会 FlashDBSimDll 工程。

在 FlashDBSimDll_Sample 工程下面，包含 4 个头文件、3 个 C++源文件和 2 个测试数据集文件：

- 4 个头文件：stdafx.h，BufferAlgorithm.h，flashdbsim_i.h 和 LRU.h；
- 3 个 C++源文件：BufferAlgorithm.cpp，LRU.cpp，main.cpp；
- 2 个测试数据集文件：trace1000000 和 trace200000。

打开工程 FlashDBSimDll_Sample 下的 main.cpp 文件，可以直接运行程序。下面介绍各个文件的作用：

- flashdbsim_i.h 和 stdafx.h：不管开展什么实验，都不应该去修改这两个文件；
- BufferAlogorithm.h 和 BufferAlgorithm.cpp：定义了所有缓冲区替换算法的基类，所有的缓存替换算法都应该从该类继承；

- LRU.h 和 LRU.cpp：给出了缓冲区替换算法 LRU 的实现，用来演示 Flash-DBSim 的使用方法；
- trace1000000 和 trace200000：是两个测试数据集文件，里面包含了若干行数据，每行数据代表一次读操作或一次写操作。每行数据包含两个列，其中，第一列是逻辑页号，第二列中用 0 和 1 分别表示读操作和写操作。
- main.cpp：程序入口，包含了一些实验参数的设置，并调用缓冲区替换算法进行算法的性能测试。

11.1.4.2 模拟器的参数设置

Flash-DBSim 是一种高效的、可重用和可配置的闪存存储系统仿真平台，可以根据上层应用的需要模拟出不同的特性的固态盘，从而可以方便地为上层应用提供仿真测试环境。Flash-DBSim 闪存模拟器通常可以采用下面表[Flash-DBSim-parameters]中的典型参数配置：

表[Flash-DBSim-parameters]. NAND 闪存的特性参数

Attribute	Value
Page Size	2,048B
Block Size	64 pages
Read Latency	25us/page
Write Latency	200us/page
Erase Latency	1.5ms/block
Endurance	100,000

Flash-DBSim 中的闪存基本参数设置是在 FlashDBSimDll_Sample 工程下面的 main.cpp 文件中完成的，相应的代码如下：

```
vfdInfo.blockCount = 1024;//设置块的数量
    vfdInfo.pageCountPerBlock = 64;//设置每个块中包含的页的数量
    vfdInfo.pageSize.size1 = 2048;//设置页的数据区域的大小
    vfdInfo.pageSize.size2 = 0;//设置页的带外数据区域的大小
    vfdInfo.eraseLimitation = 100000;//设置闪存的每个块的擦除次数上限
    vfdInfo.readTime.randomTime = 25;//设置随机读操作延迟为25微秒
    vfdInfo.readTime.serialTime = 0;//设置顺序读操作延迟为0微秒
    vfdInfo.programTime = 200;//设置写操作延迟为200微秒
    vfdInfo.eraseTime = 1500;//设置擦除操作延迟为1500微秒
```

这些闪存特性参数可以根据具体的实验要求而有所不同，可以在 main.cpp 文件中进行修改。需要指出的是，如果实验中需要修改缓冲区的大小，可以在 BufferAlgorithm.h 文件中修改变量 DEFBUFSIZE 的值，该值默认为 1536，即缓冲区可以容纳 1536 个页，每个页大小为 2048 字节，因此，缓冲区大小为 3MB。。

11.1.4.3 实例：测试 LRU 算法的性能

FlashDBSimDll_Sample 工程提供了一个测试 LRU(Least Recently Used)算法性能实验的例子。为了更好地设计实现自己的缓冲区替换算法，读者应该首先了解 LRU 算法性能测试的基本步骤、所使用的数据结构和算法成员函数。

11.1.4.3.1 LRU 算法性能测试的基本步骤

在 FlashDBSimDll_Sample 工程中，通过 main.cpp 来调用各种缓冲区替换算法，测试任何缓冲区替换算法（包括 LRU 算法和自己设计的算法）的性能时，都需要严格遵循以下 5 个步骤：

- 第 1 步：初始化配置信息；
- 第 2 步：写入数据页；
- 第 3 步：读入测试数据集，开始测试；
- 第 4 步：测试完毕，将内存中的脏页写回闪存；
- 第 5 步：获取测试结果数据。

11.1.4.3.2 LRU 算法的数据结构

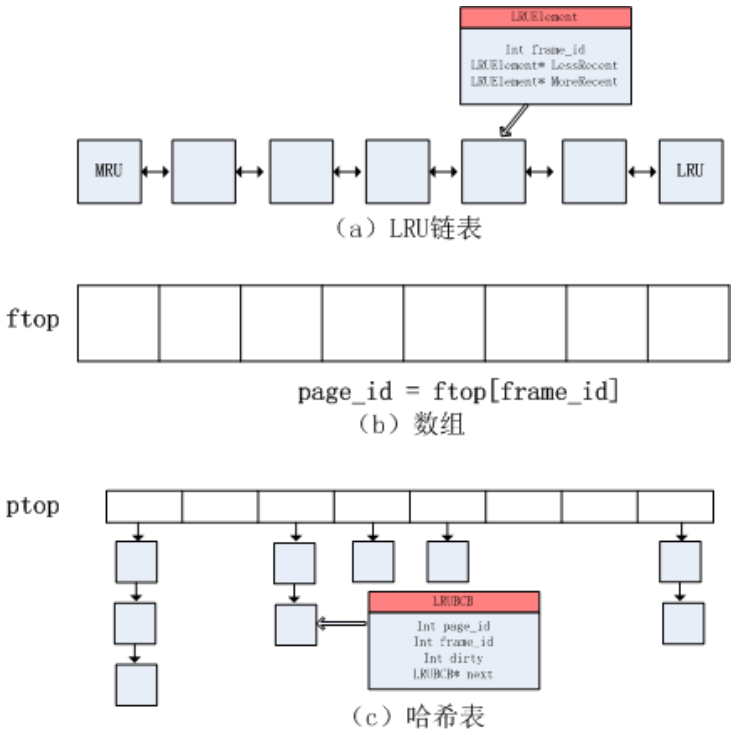
在 Flash-DBSim 自带的实例工程 FlashDBSimDll_Sample 中，包含一个头文件 LRU.h，这个头文件里定义了两个结构体 LRUBCB 和 LRUElement，以及一个数组 ftop，它们都是为 LRU 算法服务的。如图[Flash-DBSim-LRU]所示，LRU 算法使用的数据结构包含了一个 LRU 链表、一个哈希表(ptop)和一个数组(ftop)，具体作用如下：

- LRU 链表：如图[Flash-DBSim-LRU](a)所示，在 LRU 算法中，缓冲区中的所有页被组织成一个 LRU 链表，链表的 MRU（Most Recently Used）端表示最频繁访问的页，链表的 LRU（Least Recently Used）端表示最少访问的页。一个页被命中以后，会被转移到链表的 MRU 端。LRU 链表采用的数据结构是 LRUElement，包含了三个成员变量 frame_id，LessRecent,MoreRecent，其中，MoreRecent 和 LessRecent 是两个指针，分别指向左右两边的相邻元素。
- 数组 ftop：如图[Flash-DBSim-LRU](b)所示，在 LRU 算法中，内存中会有一个固定大小的空间作为缓冲区，Flash-DBSim 把每个缓冲区单元称为“帧”(frame)，每个帧都有编号 frame_id，一个帧可以保存一个逻辑页(page)，每个逻辑页都有编号 page_id，数组 ftop 记录了每个帧中存储的页号，即数组 ftop 的下表是 frame_id，数组元素 ftop[frame_id]中存储的是页号 page_id。
- 哈希表 ptop：如图[Flash-DBSim-LRU](c)所示，这个哈希表的键(key)是页号 page_id，对 page_id 使用哈希函数，可以把一个页号映射到相应的哈希桶中。多个页号可能映射到同一个哈希桶中，因此，每个哈希桶都包含了一个链表，它采用了数据结构 LRUBCB，LRUBCB 中包含了 4 个成员变量 page_id,frame_id,dirty,next，表示页号为 page_id 的页被存放在缓冲区的帧号为 frame_id 的帧中，dirty 表示是否是一个脏页，next 是一个指针，指向链表中的下一个元素。

当需要读取一个逻辑页时，需要判断该页是否在缓冲区中，方法很简单，只需要根据页号 page_id 得到哈希值，然后在哈希表 ptop 找到对应的哈希桶，每个哈希桶都存储了一个链表，如果该链表中不存在与 page_id 对应的元素，则说明该页不在缓冲区中。如果链表中存在一个与 page_id 对应的元素，则读取出该元素的 frame_id 属性的值，到相应的缓冲区单元中读取数据页。

当需要把一个页号为 page_id 的逻辑页写入缓冲区的帧号为 frame_id 的某个帧时，首先，修改数组内容，把数组元素 ftop[frame_id]的内容设置为 page_id，然后，需要根据页号 page_id 得到哈希值，在哈希表 ptop 找到对应的哈希桶，每个哈希桶都存储了一个链表，为 page_id 和 frame_id 创建一个新的元素加入到链表中。

当需要把一个逻辑页驱逐出缓冲区时，只需要根据页号 page_id 得到哈希值，然后在哈希表 ptop 找到对应的哈希桶，每个哈希桶都存储了一个链表，在链表中找到与 page_id 对应的元素，读取 frame_id 的值，然后，把内存中帧号为 frame_id 的缓冲区单元的内容清空。



图[Flash-DBSim-LRU] Flash-DBSim 中 LRU 算法使用的数据结构

11.1.4.3.3 LRU 算法的成员函数

为了快速理解 LRU 算法，读者应该重点阅读理解 LRU.cpp 中的 FixPage()函数和 SelectVictim()函数的源代码，LRU.cpp 中很多其他成员函数是不用修改的，也不需要深入理解。另外还需要注意的是，LRU.cpp 中所有与外部存储相关的操作都是“假动作”，只是模拟闪存的读和写操作，并没有实际读、写某个文件，只是统计读写次数。

下面分别介绍 LRU 算法的成员函数及其作用。

Public 成员函数：

```
Public:LRU()                                          // 构造函数，初始化数据
~LRU()                                                // 析构函数，释放资源
void Init()                  // 初始化，在 LRU 算法性能测试的基本步骤的第 2 步与第 3 步之间调用
int FixPage(int page_id)                              // 安插一个数据页
NewPage FixNewPage(LBA lba)                           // 安插一个新的数据页
int UnFixPage(int page_id)                            // 基本不会用到
void ReadFrame(int frame_id, char* buffer)            // 读一帧数据，用不到
void WriteFrame(int frame_id, const char*buffer) //  写一帧数据，用不到
int WriteDirty(void)                                  // 将内存中的脏页写回闪存
double HitRatio(void)                                 // 返回命中率
void RWInfo()                                         // 返回读写统计信息
int IsLBAValid(LBA lba)                               // 判断 LBA 是否有效
int LBAToPID( LBA lba)                                // 通过 LBA 获取 PID
```

Private 成员函数：

```
int hash(int page_id)                         // 哈希函数
```

```
LRUBCB* PageToLRUBCB(int page_id)          // 通过 page_id 获取块数据
void RemoveLRUBCB(LRUBCB* pb)                // 移除块数据
void InsertLRUEle(int frame_id)             // 在链表中插入元素
void RemoveLRUEle(int frame_id)              // 移除链表元素
void AdjustLRUList(int frame_id)            // 调整链表元素（缓冲区命中以后调整命中页在链表中的位置）
void SetDirty(int frame_id)                 // 设置 frame_id 的脏标识
int SelectVictim()                           // 选择替换页，缓存替换算法的核心函数
void RegistEntry(LBA lba, PID page_id) // 注册一页，用不到
```

Private 成员变量：

```
int ftop[DEFBUFSIZE]      // 数组
LRUBC* ptop[DEFBUFSIZE] // 哈希表
bFrame buf[DEFBUFSIZE]   // 存放帧数据，用不到
Map<LBA,PID>                  // maplist 逻辑页号与物理页号的映射关系
LRUElement* lru              // 链表的表尾指针
LRUElement* mru              // 链表的表头指针
int hit                        // 命中次数
int total                      // 读写操作的次数
int flashreadcount         // 物理读操作次数
int flashwritecount        // 物理写操作次数
```

11.1.4.4 测试自己的缓冲区替换算法的性能

假设读者需要开展实验测试自己的缓冲区替换算法的性能，建议首先阅读 FlashDBSimDll_Sample 工程下面的 main.cpp、LRU.h 和 LRU.cpp 文件，理解程序运行过程，从而更好地了解 Flash-DBSim 的使用方法以及如何在 main.cpp 中调用自己设计实现的其他缓冲区替换算法。在实现自己的缓冲区替换算法时，建议直接在 LRU 算法基础之上进行修改，这样可以避免引入未知错误。建议深刻理解 LRU 算法实例的运行细节，这样才能更好地设计实现自己的算法，知道哪些成员函数需要修改，哪些成员函数不需要修改。在实现自己的算法中，可以拷贝 LRU 算法的代码，修改关键的几个成员函数。这里需要注意的是，由于大部分成员函数不需要修改，如果读者需要实现多个不同的缓冲区替换算法，建议将那些不需要修改的成员函数移到基类 BMgr 中去。

11.2 在PostgreSQL开源数据库的真实环境下开展实验

利用闪存模拟器提供的仿真环境进行实验具有一定的局限性，因此，对于某些闪存研究实验，需要在真实的数据库和闪存设备环境下进行，因此，这里将介绍如何修改 PostgreSQL 开源数据库的源代码，从而实现真实环境下的性能测试。

11.2.1 在 DBMS 真实环境下开展实验的必要性和可行性

使用闪存模拟器（比如 Flash-DBSim）提供的仿真环境进行闪存相关算法的性能测试，这种方法的优点是，灵活易用，绝大多数算法都可以在仿真环境下进行性能测试。但是，某些算法在模拟器这种仿真环境下的性能测试结果，可能缺乏说服力，甚至准确性和真实性都会存在一定的问题。比如，缓冲区替换算法是 DBMS 内部的一个核心模块，算法的性能会受到系统内部诸多其他模块运行过程的影响，简单地采用闪存模拟器来测试缓冲区替换算法的性能，存在一定的局限性。

正是因为使用闪存模拟器进行算法的性能测试，存在一定的局限性，因此，很有必要在 DBMS 真实环境下开展相关实验。实际上，对于一些算法而言（比如缓冲区替换算法），在 DBMS 真实环境下开展实验是可以实现的，具有较好的可行性。基本思路是，根据实验的不同需求，对开源数据库（比如 PostgreSQL）的相关模块的代码进行修改。

11.2.2 在 PostgreSQL 下开展实验之前需要回答的一些问题

在使用 PostgreSQL 开展实验之前，读者可能会思考一些问题，因此，这里以问答的形式帮助读者对实验的开展有一个整体的认识。

问 1：做缓冲区替换算法的实验，是否只需要修改 PostgreSQL 缓冲区替换算法相关的源码，然后编译安装即可？

答 1：是的，在 postgresql-7.3.40/src/backend/storage/buffer 目录下修改缓冲区替换算法，修改完成以后还和以前一样安装即可。

问 2：要实现多个缓冲区替换算法，是否需要多次安装、卸载 PostgreSQL？

答 2：是的，一个系统中只有一个 PostgreSQL 数据库，如果要比较多个缓冲区替换算法的性能，则需要在每次实现一个算法的时候，都分别安装 PostgreSQL，获取测试数据，然后卸载 PostgreSQL，然后，再去实现下一个算法。

问 3：如何将数据存储到固态盘？

答 3：首先用 mount 命令挂载固态盘，然后，在 postgresql-7.4.30 目录下执行“./configure --prefix=[*安装目录*]”命令的时候指定路径。

问 4：实验的输入数据是什么？从哪里来？

答 4: 开展实验时，不需要我们自己获取输入数据，有一个名为 BenchmarkSQL 的测试工具，会自动帮我们创建表，创建索引，添加数据。

问 5:如何获取运行时间？

答 5：数据库性能测试工具—— BenchmarkSQL，会显示运行时间。

问 6：如何设置缓冲区大小？

答 6：在 PostgreSQL 的源码中，有一个名为 NBuffer 的变量，默认值是 1000，表示有几个缓冲区(数据页)，修改缓冲区大小，只需要修改这个变量的值即可。

问 7：如何统计写操作次数和命中率？

答 7：BenchmarkSQL 虽然帮助我们输入了数据，完成了数据库性能的测试，获取了数据库的执行时间，但是，没有给出命中率、物理写操作次数等信息。不过 PostgreSQL 源码中有统计这些信息，我们需要修改 PostgreSQL，把这些信息打印出来。

11.2.3 使用 PostgreSQL 开展缓冲区替换算法的步骤

使用 PostgreSQL 开展缓冲区替换算法，包括以下 4 个大的步骤：

第 1 步：下载 PostgreSQL，修改 PostgreSQL 自带的缓冲区替换算法（LRU 算法）的源代码，实现自己的缓冲区替换算法；

第 2 步：用源码安装的方式把 PostgreSQL 数据库安装到固态盘；

第 3 步：安装 BenchmarkSQL；

第 4 步：使用 BenchmarkSQL 进行性能测试，获取测试结果

以上每一步都可以分为很多小步，其中，第 1 步最复杂，需要读者首先阅读 PostgreSQL 源码，在此基础上实现自己的缓冲区替换算法。

接下来的内容，将介绍以下几个方面的细节：

- 如何下载 PostgreSQL 以及如何在 Linux 系统下把 PostgreSQL 安装到固态盘；
- 如何下载和安装 BenchmarkSQL，以及如何使用 BenchmarkSQL 对 PostgreSQL 数据库的性能进行测试；

- 如何修改 PostgreSQL 数据库的缓冲区替换算法，实现自己的缓冲区替换算法并测试性能。

11.2.4 PostgreSQL 的下载和安装

11.2.4.1 PostgreSQL 的下载

请到 PostgreSQL 官方网站（http://www.postgresql.org/ftp/source/）下载 PostgreSQLv7.4.30。下载 PostgreSQL v7.4.30 是因为这是 PostgreSQL 最后一个使用 LRU 算法的版本，更新版本的 PostgreSQL 使用了更加复杂的缓冲区替换算法，阅读起来更有难度，而且，在实现自己的缓冲区替换算法时，几乎都要和 LRU 算法进行性能比较，因此，下载这个版本的 PostgreSQL，就不用再实现一遍 LRU 算法，节省了实验工作量。将下载的 PostgreSQL 解压后，会得到一个名为 postgresql-7.4.30 的文件夹。

11.2.4.2 PostgreSQL 的安装和配置

PostgreSQL 需要在 Linux 环境下运行，可以采用虚拟机软件 VMware，在其中安装 ubuntu 系统。

在 Linux 下安装 PostgreSQL，需要遵循 Linux 下安装源码文件的三个步骤：

（1） ./configure

（2）Make

（3）Make install

通过上述三个步骤，就可以把 PostgreSQL 安装到 prefix 指定的目录下。

实际上，读者也可以根据 PostgreSQL 的安装说明文件来指导自己的安装过程。在刚才解压缩得到的 postgresql-7.4.30 目录下，有一个安装说明文件 INSTALL，这里对该说明文件中的几行关键代码做简要解释，具体如下：

```
./configure  #配置
gmake #make
su #切换到 root 用户
gmake install  #安装，以上几步是 linux 下安装软件的典型方式
adduser postgres #添加一个用户，添加完用户以后需要通过 passwd postgres 来修改该用户的密码
mkdir /usr/local/pgsql/data #新建一个目录，以后数据库的所有数据和操作都在该目录下
chown postgres /usr/local/pgsql/data #更改目录的所有者为 postgres
su - postgres  #切换用户为 postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data #初始化工作区
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test #启动服务，并新建一个名为 test 的数据库
/usr/local/pgsql/bin/psql test  #启动数据库
```

下面介绍如何将 PostgreSQL 安装到固态盘，以及如何配置 PostgreSQL，使得 BenchmarkSQL 能够顺利完成对 PostgreSQL 的性能测试。

把 PostgreSQL 安装到固态盘，可以执行以下三个步骤：

```
./configure --prefix=想要安装的路径 --without-readline --without-zlib
make
sudo make install
```

第一条语句用来指定数据库的安装路径，如果不指定，则默认安装到“ /usr/loca/pgsql” 目

录下；如果读者现在身边没有固态盘，则可以不指定--prefix 参数，让数据库安装到默认位置。需要指出的是，为了描述方便，下面假设读者安装到默认位置，如果读者安装到其他位置，在下面的语句中，请自行修改路径。

执行完上面 3 条语句以后，数据库就安装完成了，但是，还有很多工作要做。

首先，修改/usr/local/pgsql/share 目录下的 conversion_create 文件，将所有的 "$libdir" 替换成 "[*安装目录*]/lib"。

其次，在替换完成以后，需要初始化工作区：

```
mkdir /usr/local/pgsql/data
chown userName /usr/local/pgsql/data
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data #初始化工作区
```

再次，在初始化工作区以后，需要修改配置文件。进入存储空间所在文件夹(/usr/local/pgsql/data)，打开 postgresql.conf 文件，将#tcpip_socket = false 的“#”删除，并把 false 改为 true，另外，需要将#port = 5432 的“#”删除。

上面步骤完成以后，配置工作就完成了，下面只需要启动数据库服务，然后新建数据库就可以了。

```
#启动服务
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start

#创建一个名为 test 的数据库
/usr/local/pgsql/bin/createdb test
```

到这里为止，我们的数据库就创建完成了，下面就可以使用 BenchmarkSQL 来测试性能。

11.2.5 使用 BenchmarkSQL 测试性能

BenchmarkSQL 是一款采用 TPC-C 的、开源的数据库测试工具，几乎不用修改就能测试当前主流数据库的性能。这里介绍 BenchmarkSQL 的下载、安装和使用方法。

11.2.5.1 BenchmarkSQL 的下载和安装

在 Linux 下使用 BenchmarkSQL 非常方便，只需要下载到本地以后，不需要执行安装，只要运行里面的 shell 脚本即可。可以访问 BenchmarkSQL 官网（http://sourceforge.net/projects/benchmarksql/）下载该软件。

这里需要注意的是，BenchmarkSQL 的运行需要 JAVA 虚拟机，因此，需要在 Linux 下配置 JDK，读者可以参考相关书籍或网站完成 JDK 的配置，这里不再赘述。

11.2.5.2 BenchmarkSQL 的使用方法

首先，需要修改“BenchmarkSQL-2.3.2/run/”目录下面的 postgres.properties 文件，设置正确的数据库名和密码。如果之前已经创建了一个名为 test 的数据库，那么，这里只需要给出相应的账户和密码就可以了，具体配置如下：

```
driver=org.postgresql.Driver
conn=jdbc:postgresql://localhost:5432/test
user=lalor
password=123456
```

然后，在“BenchmarkSQL-2.3.2/run/”目录下运行如下命令：

```
./runSQL.sh postgres.properties sqlTableCreates
```

上面的命令用于创建我们进行 TPC-C 测试所需的数据库表。

```
./loadData.sh postgres.properties numWarehouses 10
```

上面的命令用于加载我们进行 TPC-C 测试所需的数据，numWarehouses 后的数字可以自行设置。

```
./runSQL.sh postgres.properties sqlIndexCreates
```

上面的命令用于创建我们进行 TPC-C 测试所需的索引。

```
./runBenchmark.sh postgres.properties
```

上面的命令执行后，开始执行 TPC-C 测试，这时会跳出一个对话框，用户可以根据自己的测试需要设定相关的 warehouse 数目和 terminal 数目，然后进行测试。

上面只是简单介绍了开源工具 BenchmarkSQL 的使用方法，如果更加详细的资料，读者可以参考网络资料（比如，http://blog.sina.com.cn/s/blog_448574810101a276.html）。

11.2.5.3 获取测试结果数据

使用 BenchmarkSQL 可以帮我们生成测试结果数据，从而了解数据库的性能，但是，如果要获取底层的详细信息，如缓冲区命中率、物理读操作次数和物理写操作次数等，则还需要查看 PostgreSQL 源代码，通过修改源代码来获取这些信息。 具体方法如下：

```
vim PostgreSQL-7.4.30/src/backend/libpq/pqcomm.c
```

运行上面命令，进入 pqcomm.c 文件的编辑状态，在该文件中注册一个客户端退出时的回调函数，用该函数来调用 ShowBufferUsage()函数，打印统计信息。ShowBufferUsage()函数是 PostgreSQL 自带的函数，用来打印与缓冲区相关的信息。

注册回调函数具体过程包含以下三个步骤：

第 1 步：声明回调函数 printBufferUsageInfo(void)

```
/* Internal functions */
static void pq_close(void);
static void printBufferUsageInfo(void);//新增加的回调函数声明
static int    internal_putbytes(const char *s, size_t len);
static int    internal_flush(void);
```

第 2 步：定义回调函数

```
void printBufferUsageInfo(void)
{
    FILE *fp = NULL;
    char *usage;
    usage = ShowBufferUsage();

    fp = fopen("/home/lalor/code/data.txt", "w+");
    if (fp == NULL)
    {
        fprintf(stderr, "open file error");
        fclose(fp);
    }
    else
    {
```

```
                fprintf(fp, "%s\n", usage);
                fclose(fp);
            }
        }
```

第 3 步：在合适的地方(pg_init)注册回调函数

```
void
pq_init(void)
{
        PqSendPointer = PqRecvPointer = PqRecvLength = 0;
        PqCommBusy = false;
        DoingCopyOut = false;
        on_proc_exit(printBufferUsageInfo, 0);//新增加的回调函数注册语句
        on_proc_exit(pq_close, 0);
}
```

11.2.6 删除 PostgreSQL 数据库

通过前面内容的描述，我们已经了解了在 PostgreSQL 下测试缓冲区替换算法 LRU 的性能的基本过程。在实际研究工作中，我们需要测试其他多个缓冲区替换算法的性能，从而进行性能比较，但是，一个系统中只能有一个 PostgreSQL 数据库，在测试完 LRU 算法的性能以后，我们只能删除 PostgreSQL 数据库，再吧 PostgreSQL 数据库中的缓冲区替换算法修改成其他算法，再次进行安装测试。

删除 PostgreSQL 数据库包括以下 4 个步骤：

第 1 步：关闭数据库服务

```
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile stop
```

第 2 步：删除数据库

```
/usr/local/pgsql/bin/dropdb test
```

第 3 步：卸载 PostgreSQL

```
./configure --without-readline --without-zlib
sudo make uninstall
sudo make clean
```

第 4 步：删除目录

```
sudo rm -rf /usr/local/pgsql
```

11.2.7 修改 PostgreSQL 的缓冲区替换算法

下面首先介绍修改 PostgreSQL 的预备知识，然后介绍修改 PostgreSQL 缓冲区替换算法的基本步骤，最后给出一个实例，介绍如何修改 PostgreSQL 实现缓冲区替换算法 CFLRU。

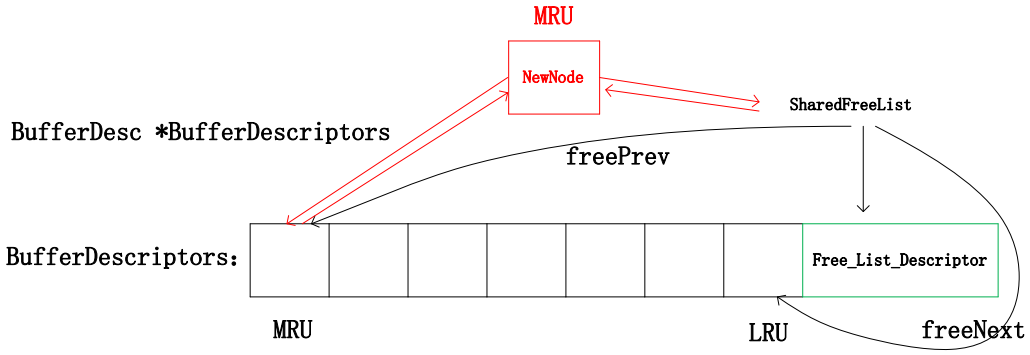
11.2.7.1 修改 PostgreSQL 缓冲区替换算法的预备知识

11.2.7.1.1 PostgreSQL 缓存替换算法的核心函数

在我们使用的 PostgreSQL 7.4.30 版本中，使用的缓存替换算法是 LRU 算法，与缓存算法相关的文件位于 postgresql-7.4.30/src/backend/storage/buffer 目录下。打开该目录下的 freelist.c 文件，可以看到几个关键的函数：AddBufferToFreeList()，GetFreeBuffer()，InitFreeList()，PinBUffer()，UnpinBuffer()，这些函数的主要功能如下：

- AddBufferToFreeList()函数：向 LRU 链表添加一个元素；
- GetFreeBuffer()函数：选择一个驱逐页，这是替换算法的核心，也是实现自己的缓存替换算法时主要修改的地方
- InitFreeList()函数：初始化 LRU 链表（LRU 链表不是真正的链表，而是一个数组）
- PinBUffer()函数 和 UnpinBuffer()函数：PostgreSQL 是一个多进程的数据库系统。当有一个进程访问 LRU 链表中的一个元素，PinBUffer()函数就将该元素从 LRU 链表中移除，并增加该元素的引用计数；在这个进程访问结束以后，需要再调用 UnpinBuffer()函数减少引用计数，在 UnpinBuffer()函数减少引用计数以后，还要判断引用计数是否为 0，如果引用计数为 0，则表示没有任何进程引用该页，此时才可以将该页再次插回到 LRU 链表的 MRU 位置。

11.2.7.1.2 PostgreSQL 中 LRU 算法的实现



图[PostgreSQL-LRU] PostgreSQL 中的 LRU 算法所使用的数据结构

图[PostgreSQL-LRU]显示了 PostgreSQL 中的 LRU 算法所使用的“LRU 链表”，不过需要注意的是，这个 LRU 链表并不是真正的链表，而是用数组来实现的，数组名为 BufferDescriptors ，数组中有 NBuffer + 1 个“描述子”，所谓“描述子”就是一个结构体（BufferDesc）类型的变量，用于表示 LRU 链表的一个元素，结构体（BufferDesc）类型定义位于头文件 buf_internals.h 中。如果我们实现的算法中需要增加新的属性，就在该结构体中添加。数组 BufferDescriptors 中，前 NBuffer 个元素用来表示空闲缓冲区（数据页），最后一个元素是“哨兵”，设置这个元素是为了方便操作，有一个指针变量 SharedFreeList 会一直指向“哨兵”位置，当满足 SharedFreeList->freeNext == Free_List_Descriptor 的时候，就说明缓冲区中的所有数据页都在被使用。有了 SharedFreeList，我们就可以很快地从 LRU 链表中选择 LRU 位置的数据页作为驱逐页，并将新的数据页插入到 LRU 链表的 MRU 位置。

在 PostgreSQL 中，有了上面的数据结构，实现 LRU 算法只用到了两个函数：

- AddBufferToFreeList()函数：使用该函数向 LRU 链表添加一个元素，新元素位于 MRU 位置；
- GetFreeBuffer()函数：该函数用于返回链表中 LRU 位置的元素。

11.2.7.2 修改 PostgreSQL 缓冲区替换算法的步骤

修改 PostgreSQL 的缓冲区替换算法主要包括两个步骤：

第 1 步：修改描述子，增加新属性。若新算法为缓冲区赋予了新的属性，则需要修改 PostgreSQL 缓冲区描述子的数据结构，在其中定义新的参数，以描述缓冲区的新属性。

第 2 步：增加新的数据结构和操作。LRU 算法要求缓冲池是一个 LRU 链表，若新算法要求缓冲池是另一种数据结构，就需要对缓冲区管理部分进行相应的修改。首先，需要修改缓冲区描述子的数据结构，增加新的数据结构，满足缓冲池的构建要求；然后，需要修改缓冲池的初始化操作，在初始化时构建新结构的缓冲池；最后，通过修改原有的缓冲池维护操作，或者替换和增加新的维护操作，保证缓冲池的数据结构始终满足新算法的要求。

11.2.7.3 修改 PostgreSQL 实现 CFLRU 算法

CFLRU 算法[ParkJKKL06]采用了“双区域”机制，把 LRU 链表划分成两个区域，即工作区域和干净优先区域，两个区域可以采用各种成熟的替换算法（比如 LRU）。CFLRU 算法不需要增加新属性，只需要判断干净页优先区域(Clean-First Region)中是否有干净页，如果有干净页，就替换干净页，如果没有干净页，就替换 LRU 位置的数据页。在 CFLRU 算法中，我们只需要增加一个变量 nWindowSize 表示干净页优先区域的大小，然后修改 GetFreeBuffer()函数即可。

（1）PostgreSQL 自带的 LRU 算法中的 GetFreeBuffer()函数

```
/*
 * GetFreeBuffer() -- get the 'next' buffer from the freelist.
 */
BufferDesc *
GetFreeBuffer(void)
{
	BufferDesc *buf;
	if (Free_List_Descriptor == SharedFreeList->freeNext)
	{
		/* queue is empty. All buffers in the buffer pool are pinned. */
		ereport(ERROR,
				(errcode(ERRCODE_INSUFFICIENT_RESOURCES),
				 errmsg("out of free buffers")));
		return NULL;
	}
	buf = &(BufferDescriptors[SharedFreeList->freeNext]);
	/* remove from freelist queue */
	BufferDescriptors[buf->freeNext].freePrev = buf->freePrev;
	BufferDescriptors[buf->freePrev].freeNext = buf->freeNext;
	buf->freeNext = buf->freePrev = INVALID_DESCRIPTOR;
	buf->flags &= ~(BM_FREE);
	return buf;
}
```

（2）采用 CFLRU 算法时修改以后的 GetFreeBuffer()函数

```
/*
 * GetFreeBuffer() -- get the 'next' buffer from the freelist.
```

```
 */
BufferDesc *
GetFreeBuffer(void)
{
	BufferDesc *buf;
	if (Free_List_Descriptor == SharedFreeList->freeNext)
	{
		/* queue is empty. All buffers in the buffer pool are pinned. */
		ereport(ERROR,
				(errcode(ERRCODE_INSUFFICIENT_RESOURCES),
				 errmsg("out of free buffers")));
		return NULL;
	}
	int n = 0;
	buf = &(BufferDescriptors[SharedFreeList->freeNext]);
	/* select a replace page in clean-first region */
	while (n < nWindowSize)
	{
		/* Is buffer dirty? */
		if (buf->flags & BM_DIRTY || buf->cntxDirty)
		{
			buf = &(BufferDescriptors[buf->freeNext]);
			/* no more buffer */
			if ( buf == SharedFreeList)
			{
				buf = NULL;
				break;
			}
		}
		else
		{/* select this page as replace page */
			break;
		}
		n++;
	}
	/*
	 *1. buf == NULL 说明缓冲区中总共也没有 nWindowSize 个 free buffer
	 *2.  n >= nWindowSize 说明缓冲区中干净页优先区域中，全都是脏页
	 * 上面两种情况，都选择位于 LRU 位置的数据页作为驱逐页
	 */
	if (n >= nWindowSize || buf == NULL)
	{
		buf = &(BufferDescriptors[SharedFreeList->freeNext]);
	}
```

```
    /* remove from freelist queue */
    BufferDescriptors[buf->freeNext].freePrev = buf->freePrev;
    BufferDescriptors[buf->freePrev].freeNext = buf->freeNext;
    buf->freeNext = buf->freePrev = INVALID_DESCRIPTOR;
    buf->flags &= ~(BM_FREE);
    return buf;
}
```

11.3 本章小结

本章内容首先介绍了如何使用闪存模拟器开展实验，详细描述了 Flash-DBSim 闪存模拟器的体系架构、参数配置和使用方法，并以 LRU 算法参考实例，演示了在 Flash-DBSim 上测试相关算法性能的基本步骤；然后，介绍了如何在 PostgreSQL 开源数据库的真实环境下开展实验。总体而言，闪存模拟器为闪存相关研究提供了一种仿真的实验环境，简单易用，但是，某些算法在模拟器这种仿真环境下的性能测试结果，可能缺乏说服力，甚至准确性和真实性都会存在一定的问题。在真实数据库环境下开展实验，可以更加准确地测试相关算法的性能，而且具有实施的可行性，比如对开源数据库的相关模块进行修改。

11.4 习题

1、 阐述闪存数据库的研究中对各种算法的性能进行验证的两大类方法及其各自优缺点。
2、 分析 Flash-DBSim 的体系架构中的各个组件的功能。
3、 阐述在 Flash-DBSim 进行算法性能测试的 5 个步骤。
4、 阐述如何修改开源数据库 PostgreSQL 从而支持对用户自己的缓冲区替换算法进行性能测试。

第三篇 基于闪存的其他存储篇

第12章 基于闪存的键值存储

本章内容首先介绍基于闪存的键值存储的应用领域，然后介绍相关的研究。

12.1 基于闪存的键值存储的应用领域

文献[DebnathSL10]中列举了键值存储的两个重要的实际应用领域：多人游戏和重复数据删除。

（1）多人游戏：目前网络在线多人游戏产业非常繁荣，吸引了大量的玩家。多人游戏中，要求处于不同地理位置的多个玩家，能够及时交互信息，这就需要服务器能够及时获得、存储和处理各个玩家的游戏状态信息。为了保证这些玩家信息处理的实时性，就要求存储这些数据的存储设备具有很高的访问速度。传统的磁盘的读写延迟比较高，无法有效满足这种需求，速度较快的 RAM 产品价格太高，无法大量使用。因此，性价比较高的闪存，就成为可以满足这类应用需求的首选产品。

（2）重复数据删除：在云存储时代，百度网盘等云存储产品越来越受到网民的欢迎，每天都有大量的用户把相关数据传输到云存储中。另外，对于企业而言，为了保证数据的可用性和可靠性，也都建立了数据备份制度。无论是个人用户数据还是企业用户数据，都可能存在重复存储的可能。在当今数据大爆炸的时代，重复数据删除的作用就尤为凸显，可以帮助企业减少重复数据存储，节约存储空间，降低企业开销。重复数据删除的方法就是，把文件分成多个块，然后对每个块使用 SHA-1 哈希，从而判定两个块是否包含相同的数据。为了加快速度，一般都是放在 RAM 中运行重复数据删除算法。但是，随着数据的暴增，RAM 有限的空间已经无法容纳如此多的数据，因此，就必须把数据保存到磁盘上，通过建立索引来快速获取磁盘中的数据。但是，磁盘的读写延迟较大，因此，如果采用性价比高的闪存来替代磁盘的角色，可以获得更好的重复数据删除性能。

12.2 相关研究

Debnath 等人于 2010 年和 2011 年先后提出了两种相似的基于闪存的键值存储方案 FlashStore[DebnathSL10]和 SkimpyStash[DebnathSL11]，两种方案很大的区别就在于，FlashStore 用于磁盘和内存之间的中间存储，而 SkimpyStash 则可以直接作为底层的数据存储。

12.2.1 FlashStore

FlashStore[DebnathSL10]在闪存中以日志结构的方式来存储“键值对”，从而可以充分获得更好的顺序写性能。同时，采用常驻内存的哈希表来为这些“键值对”提供索引，当哈希表出现冲突时，采用一种“布谷鸟哈希”的变体来解决冲突。如图[FlashStore]所示，FlashStore 包含了如下组件：

（1）RAM 写缓冲区：所有更新操作都首先进入缓冲区保存，只有缓冲区的数据量可以正好填满一个闪存页时，才一次性把缓冲区的数据刷新到闪存中。

（2）RAM 哈希表索引：这是为闪存中所存储的“键值对”建立的索引，通过这个索引，只需要一次闪存读操作，就可以获得闪存中的数据。采用了一种“布谷鸟哈希”的变体来解决哈希表冲突。

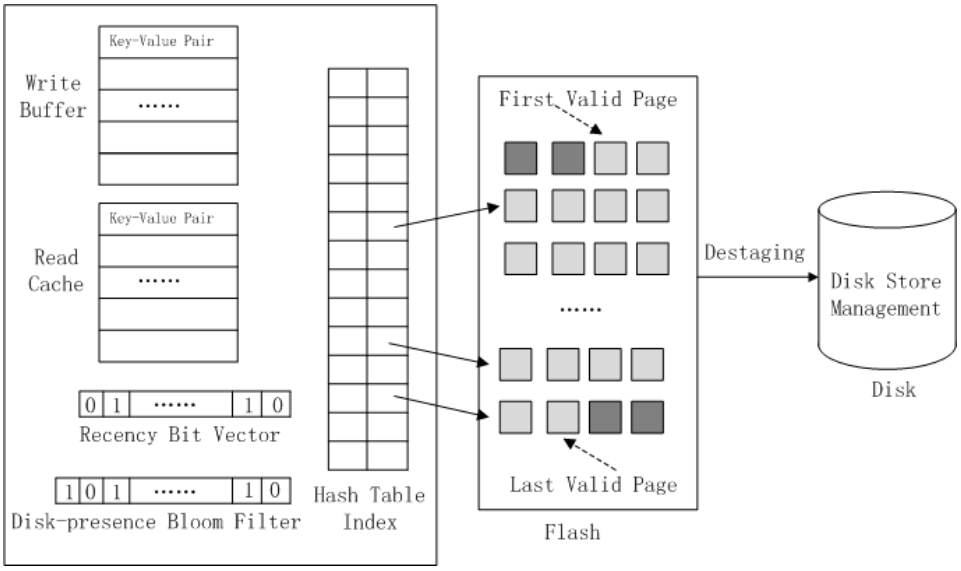
（3）RAM 读缓存：保存最近读取过的“键值对”，并且采用最近最少使用策略进行替换。

（4）近期位向量：可以记录“哈希表索引”中近期有哪些条目被访问过。

（5）布隆过滤器[BroderM02]：用来判断一个“键值对”是否已经被保存到磁盘中。

（6）闪存：用来存储经常使用的“键值对”，闪存页被组织成循环链表的形式，系统会周期性地执行“回收”过程，从链表中回收无效页和不经常使用的页（这时会用到近期位向量）。

（7）磁盘：从闪存中驱逐出来的“键值对”会被保存到磁盘中，一般都是不常用的数据。



图[FlashStore] FlashStore 的体系架构

FlashStore 采用了布隆过滤器来提升读写过程的速度，因此，这里简单介绍一下布隆过滤器的原理。布隆过滤器[BaiduBaike]是由布隆在 1970 年提出的，它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中，它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率。布隆过滤器采用的是哈希函数的方法，将一个元素映射到一个长度为 m 的阵列上的一个点，当这个点是 1 时，那么这个元素在集合内，反之则不在集合内。这个方法的缺点是，当检测的元素很多的时候可能有冲突，解决方法就是使用 k 个哈希函数对应 k 个点，如果所有点都是 1 的话，那么元素在集合内，如果有 0 的话，则判断元素不在集合内。

下面简单介绍 FlashStore 的读写操作过程：

（1）读操作过程：当使用一个键去读取数据时，首先到 RAM 读缓存中查找，如果找不到，就继续到 RAM 写缓冲区去查找，如果还是找不到，就查找 RAM 哈希表索引，如果仍然找不到，就利用布隆过滤器来判断这个键是否存在于磁盘中，如果布隆过滤器判断为存在，则到磁盘中寻找该键，如果判断为不存在，则返回空值。在上述过程中，如果在 RAM 读缓存之外的其他地方找到了该键，就把该键更新到 RAM 读缓存中，同时也会发生“近期位向量”的更新，从而记录最近被访问过的哈希表索引条目。

（2）写操作过程：首先把一个“键值对”写入到 RAM 写缓冲区，如果缓冲区中已经存在一个旧版本的数据，就让旧版本失效。如果发现缓冲区已满一页，就把数据刷新到闪存中，并更新 RAM 哈希表索引。

FlashStore 的缺点是：（1）使用变种的布谷鸟哈希函数时，需要人为设定所使用的哈希函数的数目，这个值不同，所取得的性能也不同，而对于不同的负载类型，键值分布区间不同，为达到最好性能所需要设定的哈希函数的数目也是不同的。（2）随着键的数量的增加，布隆过滤器的误识别率也会增加，这是个不容忽视的问题。

下面进一步分析一下布隆过滤器的特性对于 FlashStore 的性能影响。对于布隆过滤器而言，在采用多个哈希函数时，如果至少有一个函数判断元素不在集合中，那肯定就不在。如果它们都说元素在集合中，却有一定的误判概率，也就是说实际上元素可能并不存在。这个特性对 FlashStore 有着很大的影响。

下面我们分析四种不同的读写操作情形。对于 FlashStore 的读操作而言，存在下面两种情形：

（1）情形一：如果布隆过滤器判断某个键不存在于磁盘中时，这个键就一定不会已经保存在磁盘中，所以返回空值；如果不采用布隆过滤器，就必须到磁盘进行查找，所以，在这种情形下，布隆过滤器可以帮助节省搜索磁盘的开销；

（2）情形二：如果布隆过滤器判断某个键存在于磁盘中，这个键未必真的已经保存在磁盘中，可能是误判。不过，不管是否误判，FlashStore 都会到磁盘中去找这个键，如果找不到该键，就返回空值，如果找到了，就返回键值对。所以，在这种情形下，布隆过滤器不仅没有节省开销，而且增加了一个判断过程的开销。

对于 FlashStore 的写操作而言，也存在下面两种情形：

（1）情形三：如果布隆过滤器判断某个键不存在于磁盘中时，这个键就一定不会已经保存在磁盘中，所以，写操作必须把该键值对写入到磁盘中。如果不采用布隆过滤器，就必须先到磁盘查找该键是否存在，如果不存在就写入磁盘。可以看出，采用布隆过滤器以后，可以节省磁盘查找的开销。

（2）情形四：如果布隆过滤器判断某个键存在于磁盘中，这个键未必真的已经保存在磁盘中，可能是误判。不过，不管是否误判，FlashStore 都会到磁盘中去找这个键，如果找不到该键，就写入磁盘，如果找到了，就不执行写操作。可以看出，采用过滤器以后，不仅没有节省开销，而且增加了判断过程的开销。

从上面的四种情形的讨论可以看出，情形一和情形三可以节省开销，而情形二和情形四则会增加开销。因此，FlashStore 要想取得好的性能，必须保证情形一和情形三所节省开销，一定要大于情形二和情形四所增加的开销。也就是说，当布隆过滤器判断不存在的情形较多时，FlashStore 收益会比较大。但是，从作者给出的大量实验结果来看，并没有关于这个方面的详细分析。

12.2.2 SkimpyStash

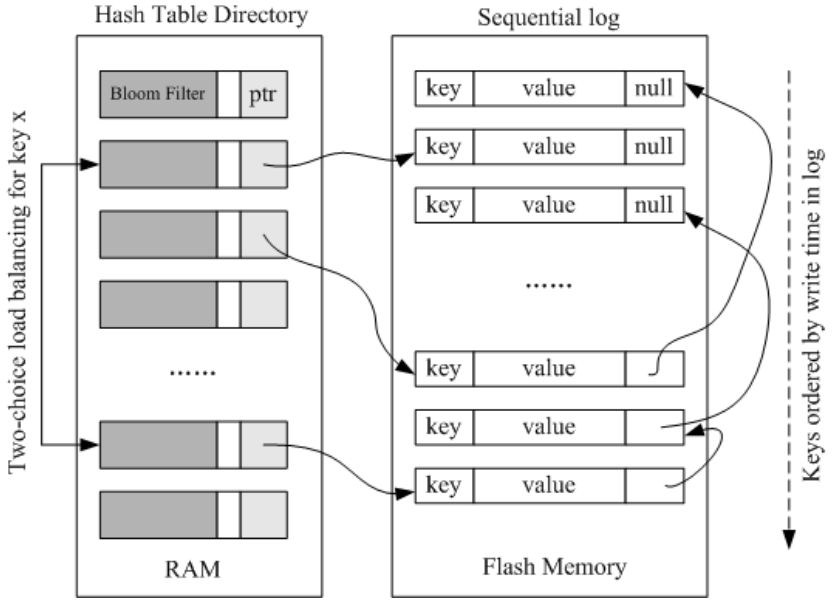
文献[DebnathSL11]提出了一个基于闪存的键值存储——SkimpyStash，只需要占用很少的 RAM 空间。如图[SkimpyStash]所示，SkimpyStash 在 RAM 中使用了一个哈希表目录来索引“键值对”，而这些键值对是以日志结构的方式存储在闪存上的。为了确定每个键值对在闪存中的位置，需要为每个键值对设置闪存指针，这会带来大量的 RAM 空间开销，为了解决这个问题，作者采用如下的方法：（1）使用线性链表解决哈希表冲突，即当有多个键被映射到同一个哈希桶中时，就会被组织成一个链表；（2）把链表存储到闪存上，并且在每个哈希桶中存储一个指向链表头部的指针。

使用哈希函数把键分配到各个哈希桶中，可能会导致每个桶中包含的键的数量不同，有时候甚至有很大的扭曲分布，这就会使得部分哈希桶对应的链表长度过大，增加了平均查找时间。为了让每个哈希桶尽量包含相近数量的键，从而减少与其对应的链表的长度，加快查找速度，作者采用了基于双选的负载均衡策略[Azar94]，该策略最初是用于把多个球均匀地投入到不同的箱子中。“双选”策略的基本思想是：每个键都使用两个不同的哈希函数 f_1 和 f_2，产生两个候选的哈希桶，然后，把该键放入到元素个数较少的那个哈希桶中。

但是，在采用双选策略后，又会增加查找过程的闪存读操作次数。因为，现在每个键都有两个可能存放的哈希桶，在查找时，如果在第一个哈希桶中找不到，就要再到第二个哈希桶中查找。在最坏的情况下，闪存读操作的次数会翻倍。为了解决查找延迟的问题，作者进一步采用了布隆过滤器（Bloom Filter）。

采用布隆过滤器以后，针对某个键的读操作过程如下：使用两个不同哈希函数找到该键可能存在于其中的两个不同的候选桶，布隆过滤器判断认为该键在哪个桶中，则到该桶指向的哈希链表中继续寻找该键。但是，布隆过去器有一定的误识别率，即当它判断认为该键存在于某个哈希桶中时，实际上可能并不存在，这意味着，扫描一遍该桶对应的哈希链表以后，却找不到该键，浪费了查找时间。幸运的是，文献[DebnathSL11]通过大量实验显示，这种

误识别率不会超过 2%。因此，采用布隆过滤器总体上加快了键的平均查找速度。



图[SkimpyStash] SkimpyStash 的体系架构

SkimpyStash 的缺点是：由于采用布隆过滤器判定一个键是否存在于某个哈希桶中，因此，不可避免地继承了布隆过滤器的缺点，即随着存入哈希桶的元素数量的不断增加，误识别率也会随之增加。

参考文献

[AilamakiDH02]Anastassia Ailamaki, David J. DeWitt, Mark D. Hill: Data page layouts for relational databases on deep memory hierarchies. VLDB J. (VLDB) 11(3):198-215 (2002)

[AgrawalGSDS09]Devesh Agrawal, Deepak Ganesan, Ramesh K. Sitaraman, Yanlei Diao, Shashi Singh: Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. PVLDB 2(1):361-372 (2009)

[ArgeH02]L. Arge, K. Hinrichs et al. Efficient bulk operations on dynamic R-trees. In Algorithmica 33(1):104-128, 2002.

[AgrawalPWDMP08]N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In USENIX ATC, 2008.

[AthanassoulisACGS10]Manos Athanassoulis, Anastasia Ailamaki, Shimin Chen, Phillip B. Gibbons, Radu Stoica: Flash in a DBMS: Where and How? IEEE Data Eng. Bull. (DEBU) 33(4):28-34 (2010)

[Azar94]Azar, Y., Broder, A., Karlin, A., Upfal, E. Balanced Allocations. In SIAM Journal on Computing (1994).

[Batory79]Don S. Batory: On Searching Transposed Files. ACM Trans. Database Syst. (TODS) 4(4):531-544 (1979)

[Ban95]A. Ban. Flash File System, Apr. 1995. United States Patent No. 5,404,485.

[Bentley79]J. L. Bentley, “Decomposable searching problems,” Inf. Process. Lett., vol. 8, no. 5, pp. 244–251, 1979.

[BaiduBaike]Bloom Filter. http://baike.baidu.com/view/4526149.htm

[BaiduBaikeSSD]http://baike.baidu.com/view/723957.htm

[BonnetB11]Philippe Bonnet, Luc Bouganim: Flash Device Support for Database Management. CIDR 2011:1-8

[BouganimJB09]Luc Bouganim, Björn Þór Jónsson, Philippe Bonnet: uFLIP: Understanding Flash IO Patterns. CIDR 2009

[BorodinLS87]Allan Borodin, Nathan Linial, Michael E. Saks: An Optimal Online Algorithm for Metrical Task Systems STOC 1987:373-382

[BroderM02]A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In Internet Mathematics, 2002.

[BlackS89]BLACK, D. L., AND SLEATOR, D. D. Competitive algorithms for replication and migration problems. Tech. Rep. CMU-CS-89-201, Carnegie Mellon University, 1989.

[Chang07] Li-Pin Chang. On Effieient Wear Leveling for Large-Scale Flash-Memory Storage Systems[C]. the 22st ACM Symposium on Applied Computing (ACM SAC), 2007.

[ChanO97]Chee Yong Chan, Beng Chin Ooi: Efficient Scheduling of Page Access in Index-Based Join Processing. IEEE Trans. Knowl. Data Eng. (TKDE) 9(6):1005-1011 (1997)

[Claburn08]T. Claburn. Google plans to use Intel SSD storage in servers. http://www.informationweek.com/news/207602745

[Chen09]Shimin Chen: FlashLogging: exploiting flash devices for synchronous logging performance. SIGMOD 2009:73-86

[Chenjianqiang10]陈坚强. 基于 NAND Flash 的数据库管理系统优化研究. 湖南大学, 硕士学位论文. 2010.

[CanimBMLR09]Mustafa Canim, Bishwaranjan Bhattacharjee, George A. Mihaila, Christian A. Lang, Kenneth A. Ross: An Object Placement Advisor for DB2 Using Solid State Storage. PVLDB 2(2):1318-1329 (2009)

[ChouD85]H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In VLDB ’85, pages 127–141. VLDB Endowment, 1985.

[ChoudhuriG07]S. Choudhuri and T. Givargis. Performance Improvement of Block Based NAND Flash Translation Layer. In CODES+ISSS’07, Salzburg, Austria, 2007.

[ChazelleG86]B. Chazelle and L. J. Guibas, “Fractional cascading: I. a data structuring technique,” Algorithmica, vol. 1, no. 2, pp. 133–162, 1986.

[ChenGN10]Shimin Chen, Phillip B. Gibbons, Suman Nath: PR-join: a non-blocking join achieving higher early result rate with statistical guarantees. SIGMOD 2010:147-158
[ChangK04]Li-Pin Chang, Tei-Wei Kuo: An efficient management scheme for large-scale flash-memory storage systems. SAC 2004:862-868
[CopelandK85]George P. Copeland, Setrag Khoshafian: A Decomposition Storage Model. SIGMOD 1985:268-279
[ChenKZ09]Feng Chen, David A. Koufaty, Xiaodong Zhang: Understanding intrinsic characteristics and system implications of flash memory based solid state drives. SIGMETRICS/Performance 2009:181-192
[CuiLC10]崔斌,吕雁飞,陈学轩. 闪存环境下B+树索引重访. 计算机应用, 2010, Vol.30(1):1-4.
[ChenLZ11]Feng Chen, Rubao Lee, Xiaodong Zhang: Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. HPCA 2011:266-277
[CanimMBRL10]Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, Christian A. Lang: SSD Bufferpool Extensions for Database Systems. PVLDB 3(2):1435-1446 (2010)
[ChungPPLLS06]Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, Ha-Joo Song: System Software for Flash Memory: A Survey. EUC 2006:394-404
[DeWittKOSSW84]David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood: Implementation Techniques for Main Memory Database Systems. SIGMOD 1984:1-8
[DoP09] Jaeyoung Do, Jignesh M. Patel: Join processing for flash SSDs: remembering past lessons. DaMoN 2009:1-8
[DebnathSL10]Biplob K. Debnath, Sudipta Sengupta, Jin Li: FlashStore: High Throughput Persistent Key-Value Store. PVLDB 3(2):1414-1425 (2010)
[DatacenterSSD]White Paper: Datacenter SSDs: Solid Footing for Growth. White Paper: Datacenter SSDs: Solid Footing for Growth. http://www.samsung.com/global/business/semiconductor/products/flash/FlashApplicationNote.html.
[Facebook]Releasing Flashcache. http://www.facebook.com/note.php?note_id=388112370932.
[FusionIO]The FusionIO drive. Technical speci¯cations available at: http://www.fusionio.com/photos/fusion-io-use-and-terminology/
[FanLM12]范玉雷,赖文豫,孟小峰.基于固态硬盘内部并行的数据库表扫描与聚集. 计算机学报, Vol35(11),2012,pp:2327-2336.
[Graefe07]Goetz Graefe: The five-minute rule twenty years later, and how flash memory changes the rules. DaMoN 2007:6
[GemmellBL06]Jim Gemmell, Gordon Bell, Roger Lueder, MyLifeBits: A Personal Database for Everything, Communications of the ACM, Vol. 49, No. 1, January 2006.
[GrayF07]Jim Gray and Bob Fitzgerald. Flash Disk Opportunity for Server-Applications. http://www.research.microsoft.com/˜gray, January 2007.
[GrayF06]Jim Gray, Bob Fitzgerald: Flash Disk Opportunity for Server Applications. ACM Queue (QUEUE) 6(4):18-23 (2006)
[GrayG97]Jim Gray, Goetz Graefe: The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. SIGMOD Record (SIGMOD) 26(4):63-68 (1997)
[GraefeHKSTW10]Goetz Graefe, Stavros Harizopoulos, Harumi A. Kuno, Mehul A. Shah, Dimitris Tsirogiannis, Janet L. Wiener: Designing Database Operators for Flash-enabled Memory Hierarchies. IEEE Data Eng. Bull. (DEBU) 33(4):21-27 (2010)
[GuptaKU09]Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar: DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. ASPLOS 2009:229-240
[GraefeLS94]Goetz Graefe, Ann Linville, Leonard D. Shapiro: Sort versus Hash Revisited. IEEE Trans. Knowl. Data Eng. (TKDE) 6(6):934-944 (1994)

[GanesanMS07] Yanlei Diao, Deepak Ganesan, Gaurav Mathur, Prashant J. Shenoy: Rethinking Data Management for Storage-centric Sensor Networks. CIDR 2007:22-31
[GrayP87]Jim Gray, Gianfranco R. Putzolu: The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. SIGMOD 1987:395-398
[GalT05]Eran Gal, Sivan Toledo: Algorithms and data structures for flash memories. ACM Comput. Surv. (CSUR) 37(2):138-163 (2005)
[Hwang03]Chang-gyu Hwang. Nanotechnology enables a new memory growth model. Proceedings of the IEEE, 2003, Vol.91(11):1765 - 1771.
[HsiehCK05]Jen-Wei Hsieh, Li-Pin Chang, Tei-Wei Kuo: Efficient on-line identification of hot data for flash-memory management. SAC 2005:838-842
[HsiehK06]Jen-Wei Hsieh and Tei-Wei Kuo, "Efficient identification of Hot Data for Flash Memory Storage Systems", ACM Transactions on Storage,Vol. 2, No. 1, pp. 22-40, 2006.
[HennessyP03]J. Hennessy and D. Patterson. Computer Architecture – A Quantitative Approach. Morgan Kaufmann, 2003.
[Intel98]Intel. Understanding the Flash Translation Layer (FTL) Specification. Technical report, Intel Corporation, Dec.1998.
[InoueW04]Atsushi Inoue and Doug Wong. NAND Flash Applications Design Guide. Technical Report Revision 2.0, Toshiba America Electronic Components, Inc., March 2004.
[JungCJKL07]Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jinsoo Kim, Joonwon Lee: A group-based wear-leveling algorithm for large-capacity flash memory storage systems. CASES 2007:160-164
[JoKPKL06]H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee. FAB: Flash-aware buffer management policy for portable media players. IEEE Transactions on Consumer Electronics, 52(2):485–493, 7 2006.
[JinOHL12]Peiquan Jin, Yi Ou, Theo Härder, Zhi Li: AD-LRU: An efficient buffer replacement algorithm for flash-based databases. Data Knowl. Eng. (DKE) 72:83-102 (2012)
[JungSPKC08]H. Jung, H. Shim, S. Park, S. Kang, and J. Cha. LRU-WSR: Integration of lru and writes sequence reordering for flash memory. IEEE Transactions on Consumer Electronics, 54(3):1215–1223, 10 2008.
[JungYSPKC07]Hoyoung Jung, Kyunghoon Yoon, Hyoki Shim, Sungmin Park, Sooyong Kang, Jaehyuk Cha: LIRS-WSR: Integration of LIRS and Writes Sequence Reordering for Flash Memory. ICCSA 2007:224-237
[JiangZ02]Song Jiang, Xiaodong Zhang: LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. SIGMETRICS 2002:31-42
[Kooi80]Robert Kooi: The Optimization of Queries in Relational Databases. Case Western Reserve University 1980
[KimA08]Hyojun Kim, Seongjun Ahn: BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. FAST 2008:239-252
[KangJKK07]Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim: mu-tree: an ordered index structure for NAND flash memory. EMSOFT 2007:144-153
[KangJKL06]Jeong-Uk Kang, Heeseung Jo, Jinsoo Kim, Joonwon Lee: A superblock-based flash translation layer for NAND flash memory. EMSOFT 2006:161-170
[KimKNMC02]J. Kim, J.M. Kim, S.H. Noh, S. Min, and Y. Cho. A Space-Efficient Flash Translation Layer for Compactflash Systems. IEEE Transactions on Consumer Electronics, 48(2):366–375, 2002.
[KimL99] Han-joon Kim, Sang-goo Lee: A New Flash Memory Management for Flash Storage System. COMPSAC 1999:284-289
[KangLM12]Woon-Hak Kang, Sang-Won Lee, Bongki Moon: Flash-based Extended Cache for Higher Throughput and Faster Recovery. PVLDB 5(11):1615-1626 (2012)
[KimLJB08] Hyojun Kim, Ki Yong Lee, JaeGyu Jung, Kyoungil Bahng:A New Transactional Flash Translation Layer for Embedded Database Systems Based on MLC NAND Flash Memory.
[KawaguchiNM95]A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In Proc. of USENIX Winter, 1995.

[KooS09]D. Koo and D. Shin. Adaptive Log Block Mapping Scheme for Log Buffer-based FTL (Flash Translation Layer). In IWSSPS'09, Grenoble, France, Oct. 2009.
[KoltsidasV08s]Ioannis Koltsidas, Stratis Viglas: Flashing up the storage layer. PVLDB 1(1):514-525 (2008)
[Lawson09]S. Lawson. Cloud computing could be a boon for flash storage. http://www.businessweek.com/technology/content/aug2009/tc20090824 219491.htm, 2009.
[LvCHC11]Yanfei Lv, Bin Cui, Bingsheng He, Xuexuan Chen: Operation-aware buffer management in flash-based systems. SIGMOD 2011:13-24
[LiHLY09]Yinan Li, Bingsheng He, Qiong Luo, Ke Yi: Tree Indexing on Flash Disks. ICDE 2009:1303-1306
[LiJSCY09]Z. Li, P. Jin, X. Su, K. Cui, and L. Yue. CCF-LRU: A new buffer replacement algorithm for flash memory. IEEE Transactions on Consumer Electronics, 55(3):1351–1359.
[LeeK07]Sang-Won Lee, Won Kim: On Flash-Based DBMSs: Issues for Architectural Re-Examination. Journal of Object Technology (JOT) 6(8):39-49 (2007)
[LeeKWCK09]Ki Yong Lee, Hyojun Kim, Kyoung-Gu Woo, Yon Dohn Chung, Myoung-Ho Kim: Design and implementation of MLC NAND flash-based DBMS for mobile devices. Journal of Systems and Software (JSS) 82(9):1447-1458 (2009)
[LeeL10]Hyun-Seob Lee, Dong-Ho Lee: An efficient index buffer management scheme for implementing a B-tree on NAND flash memory. Data Knowl. Eng. (DKE) 69(9):901-916 (2010)
[LuoLMCZ12]Tian Luo, Rubao Lee, Michael P. Mesnier, Feng Chen, Xiaodong Zhang: hStorage-DB: Heterogeneity-aware Data Management to Exploit the Full Capability of Hybrid Storage Systems. PVLDB 5(10):1076-1087 (2012)
[LeeM07]Sang-Won Lee, Bongki Moon: Design of flash-based DBMS: an in-page logging approach. SIGMOD 2007:55-66
[LeeMPKK08]Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, Sang-Woo Kim: A case for flash memory ssd in enterprise database applications. SIGMOD 2008:1075-1086
[LuMZ10] LU ZePing, MENG XiaoFeng, ZHOU Da. HV-Recovery: A High Efficient Recovery Technique for Flash-Based Database. Vol.33(12), 2010:2258-2266.
[LeePCLPS07]Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song: A log buffer-based flash translation layer using fully-associative sector translation. ACM Trans. Embedded Comput. Syst. (TECS) 6(3) (2007)
[LiR99]Zhe Li, Kenneth A. Ross: Fast Joins Using Join Indices. VLDB J. (VLDB) 8(1):1-24 (1999)
[LeeSKK08]Sungjin Lee, Dongkun Shin, Young-Jin Kim, Jihong Kim: LAST: locality-aware sector translation for NAND flash memory-based storage systems. Operating Systems Review (SIGOPS) 42(6):36-42 (2008)
[LvCC09]Lv Yanfei, Chen Xuexuan,Cui Bin. Performance Evaluation and Optimization Analysis on Flash—Based Database. Journal of Computer Research and Development.46(suppl.):307-312,2009.
[LofgrenNT03] Lofgren KM，Norman RD，Thelin GB，et al. 2003. Wear Leveling Techniques for Flash EEPROM Systems. USA，6594183[P].
[LiOXCH09]Yu Li, Sai Tung On, Jianliang Xu, Byron Choi, Haibo Hu: DigestJoin: Exploiting Fast Random Reads for Flash-Based Joins. Mobile Data Management 2009:152-161
[LiXCH10]Yu Li, Jianliang Xu, Byron Choi, Haibo Hu: StableBuffer: optimizing write performance for DBMS applications on flash devices. CIKM 2010:339-348
[LeeYL09]H.-S. Lee, H.-S. Yun, and D.-H. Lee. HFTL: Hybrid Flash Translation Layer based on Hot Data Identification for Flash Memory. IEEE Transactions on Consumer Electronics, 55(4), Nov. 2009.
[Myspace]White Paper: MySpace Uses Fusion Powered I/O to Drive Greener and Better Data Centers. http://www.fusionio.com/PDFs/myspace-case-study.pdf.
[MaFL11]Dongzhe Ma, Jianhua Feng, Guoliang Li: LazyFTL: a page-level flash translation layer optimized for NAND flash memory. SIGMOD 2011:1-12

[MengJCY12]孟小峰,金培权,曹巍,岳丽华. 闪存数据库研究进展及发展趋势. 中国科学基金, 2012 年第 3 期, 142-145.
[MerrettKY81]T. H. Merrett, Yahiko Kambayashi, H. Yasuura: Scheduling of Page-Fetches in Join Operations VLDB 1981:488-498
[MTD]MTD, "Memory Technology Device (MTD) subsystem for Linux," http://www.linux-mtd.infradead.org
[NathG10]Suman Nath, Phillip B. Gibbons: Online maintenance of very large random samples on flash storage. VLDB J. (VLDB) 19(1):67-90 (2010)
[NathK07]Suman Nath, Aman Kansal: FlashDB: dynamic self-tuning database for NAND flash. IPSN 2007:410-419
[One08]One, A., 2008. YAFFS: Yet another Flash file system. http://www.aleph1.co.uk/yaffs
[Omiecinski89]Edward Omiecinski: Heuristics for Join Processing Using Nonclustered Indexes. IEEE Trans. Software Eng. (TSE) 15(1):18-25 (1989)
[Oracle10] Oracle Corp. Oracle Databasse Concepts 11g Rel. 2. Feb 2010.
[OuH10]Yi Ou, Theo Härder: Issues of Flash-Aware Buffer Management for Database Systems. BNCOD 2010:127-130
[OuHJ09] Yi Ou, Theo Härder, Peiquan Jin: CFDC: a flash-aware replacement policy for database buffer management. DaMoN 2009:15-20
[OnHLX09]Sai Tung On, Haibo Hu, Yu Li, Jianliang Xu: Lazy-Update B+-Tree for Flash Devices. Mobile Data Management 2009:323-328
[OnLHWLX10]Sai Tung On, Yinan Li, Bingsheng He, Ming Wu, Qiong Luo, Jianliang Xu: FD-buffer: a buffer manager for databases on flash disks. CIKM 2010:1297-1300
[OuH10]Yi Ou, Theo Härder: Clean first or dirty first?: a cost-aware self-adaptive buffer replacement policy. IDEAS 2010:7-14
[OnXCHH12] Sai Tung On, Jianliang Xu, Byron Choi, Haibo Hu, Bingsheng He: Flag Commit: Supporting Efficient Transaction Recovery in Flash-Based DBMSs. IEEE Trans. Knowl. Data Eng. (TKDE) 24(9):1624-1639 (2012)
[Patterson05]David A. Patterson: Latency Lags Bandwidth. ICCD 2005:3-6
[PucheralBVB01]Philippe Pucheral, Luc Bouganim, Patrick Valduriez, Christophe Bobineau: PicoDBMS: Scaling down database techniques for the smartcard. VLDB J. (VLDB) 10(2-3):120-132 (2001)
[ParkJKKL06]Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jinsoo Kim, Joonwon Lee: CFLRU: a replacement algorithm for flash memory. CASES 2006:234-241
[PrabhakaranRZ08] Vijayan Prabhakaran, Thomas L. Rodeheffer, Lidong Zhou: Transactional Flash. OSDI 2008:147-160
[ParkSSML10]Seon-Yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, Joonwon Lee: Exploiting Internal Parallelism of Flash-based SSDs. Computer Architecture Letters (CAL) 9(1):9-12 (2010)
[RamakrishnanG00]R. Ramakrishnan, J. Gehrke (2000) Database management systems. WCB/McGraw-Hill, NewYork.
[RohLP10]Hongchan Roh, Daewook Lee, Sanghyun Park: Yet another write-optimized DBMS layer for flash-based solid state storage. CIKM 2010:1345-1348
[RosenblumO92]Mendel Rosenblum, John K. Ousterhout: The Design and Implementation of a Log-Structured File System. ACM Trans. Comput. Syst. (TOCS) 10(1):26-52 (1992)
[Sliberschantz04]A. Sliberschantz et al: Operating System Concepts. 6th ed., John Wiley & Sons, Inc. (2004)
[StoicaAJA09]Radu Stoica, Manos Athanassoulis, Ryan Johnson, Anastasia Ailamaki: Evaluating and repairing write performance on flash devices. DaMoN 2009:9-14
[SilberschatzC98]A. Silberschatz and P. B. Calvin, Operating System Concepts, John Wiley & Sons, Inc., New York, NY, 1998.
[Samsung06]Hachman, M.: New Samsung notebook replaces hard drive with flash. http://www.extremetech.com/article2/0,1558,1966644,00.asp, May 2006
[Samsung2006]Samsung．K9G8G08UOM Datasheet．Soul：Samsung，2006

[ShahHWG08]Mehul A. Shah, Stavros Harizopoulos, Janet L. Wiener, Goetz Graefe: Fast scans and joins using flash drives. DaMoN 2008:17-24
[SilberschatzKS05]Silberschatz, A., Korth, H.F., Sudarshan, S., 2005. Database System Concepts, fifth ed. McGraw-Hill.
[SuJXCY09]Xuan Su; Peiquan Jin; Xiaoyan Xiang; Kai Cui; Lihua Yue. Flash-DBSim: A simulation tool for evaluating Flash-based database algorithms. Proceedings of 2nd IEEE International Conference on Computer Science and Information Technology, 2009, pp: 185 – 189.
[SeoS08]D. Seo and D. Shin. Recently-evicted-first buffer replacement policy for flash storage devices. IEEE Transactions on Consumer Electronics, 54(3):1228–1235, 10 2008.
[TangMLL11] Tang X, Meng XF, Liang ZC, Lu ZP. Cost-Based buffer management algorithm for flash database systems. Journal of Software, 2011,22(12):2951−2964.
[TPC-H] TPC-H Benchmark. http://www.tpc.org/tpch/
[Valduriez87]Valduriez P Join indices. ACM Trans Database Syst,1987, 12(2):218–246
[Woodhouse01] David Woodhouse. JFFS: The Joumnaling Flash File System, Ottawa Linux Symposium, 2001.
[WuKC07]Chin-Hsien Wu, Tei-Wei Kuo, Li-Ping Chang: An efficient B-tree layer implementation for flash-memory storage systems. ACM Trans. Embedded Comput. Syst. (TECS) 6(3) (2007)
[WuCK03]Chin-Hsien Wu, Li-Pin Chang, Tei-Wei Kuo: An Efficient B-Tree Layer for Flash-Memory Storage Systems. RTCSA 2003:409-430
[Xiang09] 向小岩. 闪存数据库若干关键问题研究. 博士学位论文. 中国科技大学. 2009.
[Yaffs] Aleph One Ltd., Yet Another Flash File System(YAFFS), 2002. http://www.yaffs.net.
[YueXJL10] YUE Lihua, XIANG Xiaoyan, JIN Peiquan, LIU Zhanzhan.An Out-page logging approach to storage management in flash-based DBMS.JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA. Vol.40(5), 2010:526-532.
[YangWHGC11]Liang Huai Yang, Jing Wang, Zhifeng Huang, Weihua Gong, Lijun Chen: An Efficient Buffer Scheme for Flash-based Databases. JCP 6(7):1307-1318 (2011)
[Zeinalipour-YaztiLKGN05]Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, Walis A. Najjar, MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices, 4th USENIX Conference on File and Storage Technologies (FAST 2005), December 2005.
[ZhouCL04]Yuanyuan Zhou, Zhifeng Chen, Kai Li: Second-Level Buffer Cache Management. IEEE Trans. Parallel Distrib. Syst. (TPDS) 15(6):505-519 (2004)
[ZhengGSZJKW03]Fengzhou Zheng, Nitin Garg, Sumeet Sobti, Chi Zhang, Russell E. Joseph, Arvind Krishnamurthy, Randolph Y. Wang, Considering the Energy Consumption of Mobile Storage Alternatives, Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Orlando, Florida. Oct 2003.
[ZhangYKW02]Chi Zhang, Xiang Yu, Arvind Krishnamurthy, Randolph Y. Wang: Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. FAST 2002:289-304
[LvCC09]吕雁飞,陈学轩,崔斌.基于闪存的数据库性能评测与优化分析.计算机研究与发展.46(增):307-312,2009.
[YueXJL10] 岳丽华,向小岩,金培权,刘沾沾.基于分离日志的闪存数据库系统存储管理方法.中国科学技术大学学报.Vol.40(5), 2010:526-532.
[LuMZ10] 卢泽萍,孟小峰,周大. HV-Recovery: 一种闪存数据库的高效恢复方法. Vol.33(12), 2010:2258-2266.

厦门大学数据库实验室
为中国科研事业贡献力量

扫一扫访问厦门大学数据库实验室网站