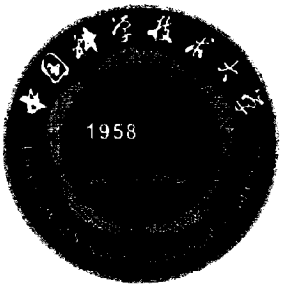
中国科学技术大学

硕士学位论文



基于 PCM 的 B+-Tree 索引的研究

作者姓名：　　李　璐

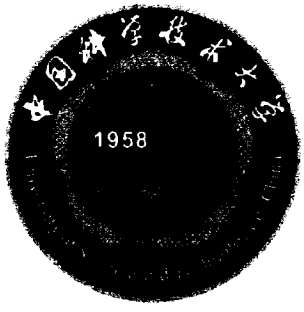
学科专业：　　计算机应用技术

导师姓名：　　岳丽华 教授

完成时间：　　二〇一六年四月

University of Science and Technology of China

A Dissertation for Master's Degree



Research on the PCM-based B+-tree Index

Author's Name: Lu Li

Specialty: Computer Application Technology

Supervisor: Prof. Lihua Yue

Finished time: April, 2016

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：李杨　　签字日期：2016.5.27

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☑公开　□保密（____年）

作者签名：李杨　　导师签名：

签字日期：2016.5.27　　签字日期：2016.5.27

摘 要

随着科技的高速发展和大数据时代的到来，数据的存储需求和对响应时间的要求都在不断提高，仅仅从软件层面上解决存储问题比较困难，需要有新技术来摆脱磁盘的机械特性限制。近年来，材料技术的进步促进了新型存储器的飞速发展，为高效能大数据存储和管理研究带来了新机遇。继 Flash 之后，国内外许多研究者开始关注以相变存储器 PCM 为代表的非易失存储器。

可字节寻址、非易失、低能耗和大容量等特点，让 PCM 成为当下可替代传统内存介质 DRAM 的热门新型存储器。但由于 PCM 与 DRAM 特性的不同，已有的数据管理方法不适用于 PCM。首先，DRAM 是易失型介质，因此一般用来存储计算过程中的动态数据；而 PCM 是非易失介质，既可以存储动态计算数据，也可以存储文件等持久的静态数据。其次，主存级 PCM 的引入并不能完全取代 DRAM，虽然 PCM 读性能较好，但写代价相对较高，而且写次数有限，因此在存储架构中依然需要借助 DRAM 来保证其高效性。因此传统的内存数据管理机制，包括空间分配、存储组织、索引等不能直接适用于 PCM。本文针对内存数据管理机制中的索引算法，探讨如何优化传统的索引机制，以让其适用于基于 PCM 的内存数据管理。

具体而言，本文的主要贡献有：

（1）提出了溢出类型 B+-tree（OB+-tree）索引的读访问优化算法 CB+-tree。本文首先对结点的读写倾向性进行了定义，分析了读写倾向性判断方法，然后详尽阐述了利用结点的读写倾向进行溢出链调整的策略，以及优化后索引的查询、插入和删除操作方法。最后通过对比实验，证明溢出链优化方法可以有效的减少索引对 PCM 的读次数，与 OB+-tree 相比减少约 25%读次数，平均整体时间性能提高约 15%。

（2）提出了利用 DRAM 作为辅助存储的溢出类型 B+-tree（OB+-tree）索引的优化算法 XB+-tree。在 DRAM 和 PCM 混合主存架构中，以 DRAM 的高写性能来弥补 PCM 的缺点，结合索引结点具有读写倾向性的特点，将少量具有写倾向的结点迁移至 DRAM 上；同时，为小容量的 DRAM 提出了索引结点管理算法。实验证明，提出的优化算法可以有效减少 PCM 的读写次数，与 OB+-tree 相比，对 PCM 的读写次数分别降低约 31%和 64%，混合主存架构的整体性能也有较大提升，平均约为 38%。

关键词： B+-tree　索引　PCM　读写倾向性

Abstract

With the high-speed technology development and the advent of big data era, data storage requirements and response time requirements are constantly improving. However, it is difficult to solve storage problems from the software perspective, and we need a new kind of storage medium to get rid of the restrictions of the mechanical properties of the disk. In recent years, with the progress in materials science, the new memory technologies develop rapidly. The emergence of new storage technology has brought new opportunities for the research on the efficient big data storage and management. Many researchers began to pay attention to non-volatile memory, which is represented by PCM, following the Flash study heat.

PCM, as a current alternative new memory to DRAM, has attractive properties such as byte addressable, non-volatile, low energy consumption and big capacity, and causes great concern of scholars. However, the DRAM-based data management method is not available for the PCM-based memory database because the differences between PCM and traditional DRAM memory medium. First, DRAM is volatile storage medium and can be used to store the dynamic data in calculation process; while PCM is non-volatile, both dynamically calculated data and persistent static data such as files are suitable to PCM. Secondly, PCM cannot completely replace DRAM as main memory medium. Although PCM has fast read speed, its slow write operations and a limited number of write cycles make it still need DRAM to guarantee the performance in the storage infrastructure. Thus, the traditional memory data management mechanism cannot be used on PCM, including spatial distribution, storage organization and index. In this paper, we focus on the indexing algorithm, which is one of the in-memory data management mechanism, to discuss how to optimize the traditional indexing mechanism in order to make it efficient for the PCM-based memory data management.

The main contribution of this paper can be summarized as follows:

(1)We proposed the overflow B+-tree (OB+-tree) index read access optimization algorithm CB+-tree. This paper first defines the read and write tendency of leaf node, and analyze the methods of read and write tendency judgment. Then we elaborate the overflow chain adjustment policies using read and write tendency judgment mechanism, and present the query, insert and delete operations. Finally, comparative experiments

show that the CB+-tree can effectively reduce the number of PCM read by 25%, and improve overall performance by approximately 15% on average, compared with OB+-tree.

(2)We proposed an OB+-tree index optimization algorithm called XB+-tree using the auxiliary storage DRAM. Based on DRAM and PCM hybrid memory architecture, we use DRAM, which has high write performance, to compensate for the PCM shortcomings. The write tendency nodes need to migrate to DRAM with the using of read and write tendency mechanism. Meanwhile, we proposed the node management algorithm for small-capacity DRAM. Experimental results show that the XB+-tree can effectively reduce the PCM frequency of reading and writing. Compared with OB+-tree, the number of read and write to PCM decrease 31% and 64% respectively, the overall performance of the hybrid memory architecture also has improved by 38% on average.

Key Words: B+-tree, index, PCM, read and write tendency

目 录

图目录

表目录

第1章 绪论

1.1 研究背景及意义

在社会飞速发展的今天，人们生活中的方方面面都有计算机的踪影，大量的计算机应用导致了数据爆炸式的增长。随着大数据时代的到来[2]，数据中心的作用越来越重要，其需要处理的数据量也快速增长，全球数据中心的数量与规模都在不断扩大，如 Google、Amazon、阿里巴巴等企业皆拥有规模庞大的数据中心，并且正以惊人的速度扩张。目前，研究者普遍认为大数据具有 4V（volume，variety，value 和 velocity）特征[1]，分别表现了大数据在数据规模、数据类型、数据价值和数据处理速度方面的特点，这些特性让现有大数据管理技术面临新的挑战：大数据存储技术和高效能处理技术成为让大数据管理性能满足日益增长的超大规模数据要求的必要条件。在这样的环境下，高性能计算（HPC）系统必须以高性能大容量的存储系统来支撑运作，以满足大规模数据采集技术和处理技术的飞速发展。目前，高性能计算机系统的存储容量已达到甚至超过 PB 量级[3]，不断扩大的存储系统规模为高性能计算机系统的性能带来了严峻的挑战。

而仅仅从软件层面上解决这些问题比较困难，我们需要利用一种新型存储介质来建立一种新的存储架构，以解决大数据存储和高性能处理的问题。大量的科学计算负载特征研究表明：小粒度随机 I/O 访问是科学计算应用程序中造成 I/O 瓶颈的一个重要原因。在 I/O 性能方面，并行 I/O 技术的优化效果较好，它提供了较高的访问带宽，对大块连续数据的访问性能有较大提升，但对于小粒度随机分布的数据的访问延迟却束手无策。

最直接有效的优化 I/O 性能的方法，是利用新型存储介质来摆脱磁盘的机械特性限制[3]，减少、甚至从根本上免去数据寻道时间，早在 2003 年，美国的“高端计算复兴之路”（Revitalizaiton of High-End Computing）就对未来 15 年推动存储和 I/O 发展的使能技术进行了预测，指出了研究新型存储机理是未来的一个重要课题，新型存储器将会扮演越来越重要的角色，随着材料科学的发展，一些新型非易失存储设备相继问世，其中包括相变存储器 PCM。

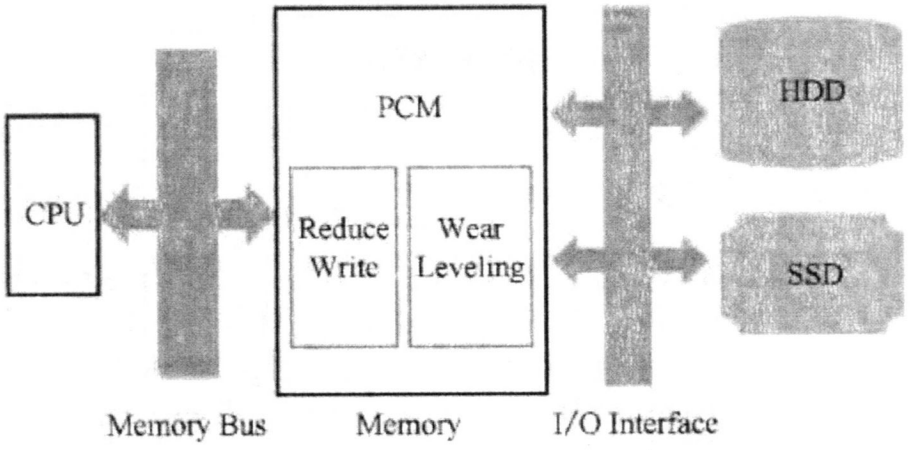


图 1.1 基于 PCM 的主存系统架构[1]

近年来，学术界和工业界对相变存储器（Phase Change Memory）的研究热度逐渐增加。PCM 诞生于 20 世纪 60 年代，作为一种非易失型存储器，PCM 比传统主存设备 DRAM 具有更广泛的应用前景。PCM 具有大容量和低功耗的特性，非常适用于大数据存储管理，拥有可以替代 DRAM 成为大容量非易失主存的发展趋势。然而已有的基于闪存和传统内存的数据管理技术都不适用于 PCM，作为数据管理的重要数据结构——索引，也存在同样的问题。所以建立以 PCM 为主存介质的数据库索引算法，成为数据库领域重点研究内容之一。

可字节寻址、非易失、低能耗和大容量等特点，让 PCM 迅速成为当下热门的新型存储器。但由于 PCM 与传统内存介质 DRAM 特性的不同，基于 DRAM 的数据管理方法不适用于基于 PCM 的内存数据库。首先，DRAM 是易失型介质，因此一般用来存储计算过程中的动态数据；而 PCM 是非易失介质，既可以存储动态计算数据，也可以存储文件等持久的静态数据。其次，主存级 PCM 的引入并不能完全取代 DRAM，虽然 PCM 读性能较好，但写代价相对较高，而且写次数有限，因此在存储架构中依然需要借助 DRAM 来保证其高效性。因此不能采用传统的内存数据管理机制，包括空间分配、存储组织、索引等。因此，本文围绕内存数据管理机制中的索引算法，探讨如何优化传统的索引机制，以让其适用于基于 PCM 的内存数据管理。

在本章余下的小节中，第 1.2 节首先简要概述了存储器的相关研究和不同介质上的 B+-tree 索引及相关研究工作。第 1.3 节介绍了本文所做的主要工作。最后，第 1.4 节介绍本文的组织结构和论文安排。

1.2 国内外研究现状

1.2.1 存储器研究

1984 年东芝公司率先提出了闪存的概念，而英特尔于 1988 年成为第一个生产闪存并将其投放市场的公司。闪存 Flash 是一种固态存储器[4]。闪存芯片上的数据组织管理单位是“块-页”的两级结构，其支持三种基本操作：读、写和擦除，其中读写基本单位是“页”，擦除操作的基本单位是“块”，一个块通常由多个（64~128）页构成。闪存有如下几个特点：（1）写前擦除机制：对闪存的每个页进行写操作之前，都要将数据页所在块的全部内容先擦除，所以写一个页会导致目标块上的所有页被擦除，产生数据丢失，所以，需要在擦除前读出块上的全部有效页面，在擦除操作后再重新写回，这种方式带来的额外写操作被称为“写放大”；（2）异位更新：鉴于闪存的写前擦除机制，闪存数据写操作采用异位更新的方式，避免了“写放大”问题；（3）读写不对称：闪存芯片上读取一个页面的延时约为 25~60us，而写一个页面的延时约为 200~1000us[5]，写延时远大于读延时，当芯片磨损，擦除操作的速度会降到相当慢；（4）磨损寿命有限：闪存芯片中每个块的可擦除次数是有限的，一旦某个块的擦除次数达到上限，该块就会损坏，整个闪存芯片无法继续工作。由于闪存芯片的读写不对称性特点和写前擦除机制，在设计基于闪存的算法时，应该重点关注写操作对整体性能的影响，优化写操作以发挥闪存的高速访问性能。

动态随机存储器 DRAM（Dynamic Random Access Memory）是当下最常见的系统内存存储介质。DRAM 利用电容来存储数据，数据存储时间较短。电容内存储电荷的多寡代表一个二进制比特是 1 还是 0，其每一个比特只需要一个晶体管另加一个电容，结构简单高效。但是电容不可避免的存在漏电现象，如果电荷不足会导致数据出错，因此电容必须被周期性的刷新（预充电）。由于这种需要定时刷新的特性，被称为“动态”存储器。DRAM 有如下两个特点：（1）易失型存储：由于 DRAM 是利用电容存储数据，一旦系统断电，数据会全部丢失；（2）空闲能耗高：由于 DRAM 使用的电容需要周期性的充电，其空闲状态也需要较高的电能来维持数据的有效性。随着科技的发展，我们需要越来越大的内存空间来保证大数据环境下的系统高效能，但是 DRAM 使用的晶体管-电容结构的体积已经趋于极限，很难继续缩小，这导致 DRAM 的容量有限；若一味的增大体积来获得大数据容量的 DRAM 会导致其能耗大大增加。所以，在大数据环境下的 DRAM 主存系统正面临很大的挑战。

相变存储器 PCM（Phase Change Memory）是一种新型非易失存储器，它是

利用硫化物在不同状态下的电阻不同来表示比特的 1 与 0。随着存储技术的发展，PCM 有望代替 DRAM 成为主存的重要组成部分，甚至可以取代外存，模糊主存与外存的概念[6]。PCM 主要有如下几个特点：（1）非易失型存储：由于 PCM 是利用材料的不同电阻来表示数据，与是否有电能供应无关，所以掉电数据不丢失；（2）可字节寻址与原位更新：PCM 的读写操作可以精确到字节粒度，且写操作可以直接在原位修改材料的电阻值，不需要闪存那样的写前擦除操作，避免了“写放大”问题；（3）空闲能耗低：由于硫化物在处于某一状态后，只有加热操作会改变其状态，所以 PCM 空闲状态下不需要周期性的提供电能刷新，这让其空闲能耗非常低；（4）读写不均衡：PCM 所使用的硫化物需要经过高温加热才能在不同的状态间转变，更改一次所需要的时间较长，导致其写延时高于读延时，且写能耗较高；（5）磨损寿命有限：一块 PCM 所能承受的写次数是有限的，一旦某一块达到了写次数上限，会让整个 PCM 芯片无法适用。PCM 的这些独有的特点，让传统的基于 Flash 和 DRAM 的索引算法无法发挥出 PCM 的优势。

1.2.2 B+-tree 索引研究

索引技术是现代搜索应用、信息检索、数据挖掘的关键技术之一。而 B+-tree 索引适合随机和顺序处理的应用环境，在数据库系统、文件系统建立索引等方面有广泛的应用。B+-tree 结构中，所有记录存储在叶子结点，可以显著减少定位记录时所经历的中间过程，从而加快存取速度。探讨和优化现有的 B+-tree 索引算法，让其适用于 PCM，对于数据存储管理的发展具有重大意义。

写优化的 B+-tree 索引已经成为过去十几年中学者们深入研究的课题[7]，对于硬盘上的 B+-tree 索引，许多人提出了索引写性能的优化办法。文章[8]提出了 LSM-tree，它为数据库维护了一个实时的低代价索引，对于频繁更新的数据具有较好的性能。文章[9]提出了一个优化 I/O 性能的缓冲树索引。文章[10]基于日志结构的文件系统提出了一个新的写优化的 B+-tree 索引。这些算法让页面迁移更高效，保留了细粒度锁的同时也保证了高并发度和快速查询性能。

对于内存索引，其中一个优化方向是让索引成为 cache 敏感索引，因为 CPU 访问主存是以 cache-line 为单位，而不是页面，更细的访问粒度可以设计更精细的结点结构以达到性能提升。文章[11]提出的 CSB+-tree 是针对主存系统 cache 进行优化的，CSB+-tree 在每个结点中只存储第一个孩子结点的地址，剩余的孩子结点则通过第一个孩子结点地址与偏移量来访问，这样就提高了一个 cache-line 的利用率，让索引成为 cache 敏感索引。文章[12]提出的 FPB+-tree 通过增加结点大小，使一个结点由多个 cache-line 组成，在访问该结点时将预读取所有的 cache-line。

对于闪存上的 B+-tree 索引的研究，主要思路是减少随机写操作，因为闪存不能原位更新，需要写前擦除，而擦除的操作单位为块级别，如果大量的随机写到达闪存，那么必然造成大量的未被更新的数据被擦除重写，降低了整体性能。文章[13]提出了 BFTL 算法，用来管理细粒度的 B+-tree 索引更新，它的优势在于算法的执行是在 FTL 层，所以不需要修改现有的应用程序。文章[14]提出的 FD-tree 主要由两部分组成：头 B+-tree 和 SSD 上的若干层有序段。它降低了操作对头 B+-tree 的随机写，并且通过迁移操作将许多小的随机写合并成连续写操作。还有一种方法是利用溢出结点结构的“以读换写”方式，比如 uB+-tree[6]和 MB-tree[15]。uB+-tree 提出让 B+-tree 叶子结点可以拥有 1 个溢出结点，算法证明了确实可以减少 B+-tree 写操作，但是对于带来的额外读代价却束手无策。MB-tree 是基于溢出机制设计，溢出链长度可以比 uB+-tree 的更长，它记录了溢出链中每个溢出结点的最大键值，使查找操作更便捷，这样做可以在一定程度上优化读代价，但它的有效性明显依赖于键值的插入顺序[5]：如果键值以一个随机顺序插入，那么 MB-tree 的效果就比较差。

由于 PCM 具有读写不对称和寿命有限的特性，频繁修改索引会大大降低基于 PCM 的数据库访问性能，也会导致 PCM 提前达到寿命极限，这些问题都限制了 PCM 成为新型主存介质的发展。如何对 B+-tree 索引修改，减少其读写操作，提高整体的访问性能，是当前对于 PCM 上 B+-tree 索引研究的主要关注点。

对于 B+-tree 索引的优化，其中一个研究方向是减少索引对 PCM 的写操作。因为 PCM 的写代价高于读代价，如果可以用适量的小代价读操作代替写操作，那么就可以提升整体性能。由于 B+-tree 的所有记录都存储在叶子结点中，每次对数据的修改都对应至少一个叶子结点的更新，这是无法避免的。所以如何减少叶子结点的变化对上层父结点的影响，成为 B+-tree 优化的主要方法之一。

但是随着 PCM 技术的发展，其读写代价的差异在逐渐缩小，一味的以读换写已经不能使整体性能提高，以大量的读操作增加换取少量写操作减少，可能反而导致整体性能的下降。在这些以读换写的优化算法的基础上，如何减少其读次数，也成为了提升 PCM 性能的方法之一。

现阶段的主存介质主要以 DRAM 为主，其优点在于读写代价较均衡。相较于读写代价不均衡且寿命有限的 PCM，以两者结合的方式构建混合主存，会比单纯使用 PCM 代替 DRAM 更具有发展空间。混合主存可以体现 DRAM 的低写延迟的特点的同时，发挥 PCM 的非易失和大容量的优势。对于 B+-tree 来说，把需要频繁写的结点转移到 DRAM 上，就可以大大减少对 PCM 的写操作，同时也可以提升主存的整体性能。基于混合主存的 B+-tree 索引优化也成为了一种主要研究方向之一。

传统 B+tree 的结点内部记录是有序的，查询时可以采用二分查找方法，因此查询性能较好；但是，为了保持记录有序，插入、删除等操作时会带来大量额外排序写操作。无序结点结构[6][16]是让 B+-tree 的结点内部记录无序化，这样可以大量减少排序造成的叶子结点内部的额外写操作。其中文章[6]提出了三种无序方式：全部结点无序，仅叶子结点无序和带有位图信息的叶子结点无序。其中仅叶子结点无序方案在仅插入操作时有较好的表现，在仅删除操作时性能最好的是带有位图的叶子结点无序方案。但是，结点内部记录无序让二分查找算法无法使用，增加了每次查询的读操作；在结点分裂时，仍避免不了对记录的重新排序，写操作增加无法避免。

溢出结构 B+-tree（OB+-tree）[16][17]是针对减少叶子结点对父结点结构的影响而提出的。文章[17]中首次提出溢出结点的概念，其溢出链的长度只有 1。溢出链机制可以让一部分叶子结点的分裂与合并操作消化在溢出链内部，而不影响父结点的结构，从而减少整体的写操作。文章[16]将溢出机制引入 PCM 中，并增加了溢出链的长度，让溢出链内部可以解决更多的结点分裂或合并操作，进而更大程度减少整体对父结点的写操作。然而过长的溢出链，会导致查询操作需要读取过多的溢出结点。在查询密集型的数据操作中，溢出链机制的表现很差；随着 PCM 的读写不对称性的缩小，大量的额外读反而会降低整体性能。

Bp+-tree[7]是基于 DRAM 和 PCM 混合主存设计，采用了预测机制。Bp+-tree 将新插入的记录以普通的 B+-tree 结构存放在 DRAM 中。每次有新记录到达，维护其概率分布柱状图，根据预测算法，可以估算出在特定区间内的数据总量。当 DRAM 满时，就把结点迁移到 PCM 上，迁移时根据该结点内数据所在区间的总数据量估计值，决定是否要将结点分裂成 2 个新结点，为将来可能到达的数据预留空间。这样，如果预测得准确，那么结点预留的空间就可以减少叶子结点分裂和合并所造成的对上层结点的写操作。但是这种预测方法需要事先知道总数据量，在实际应用中，事先知道数据总量的情况少之又少，而且预测算法较简单，准确性无法保证，实际的使用效果无法估计。

1.3 本文主要工作

基于 Flash 的 B+-tree 索引的问题研究已经进展多年，取得了非常丰富的成果。然而 Flash 是块设备，写前需要擦除操作，且读写粒度为页级，这些特点让闪存上的 B+-tree 索引算法无法在内存中发挥高效能的工作能力。而传统内存

DRAM 由于容量和能耗的瓶颈，也已经无法满足当今大数据环境下的存储系统的要求了。新型存储器 PCM 的出现，成功吸引了研究者的关注点。而当前基于 PCM 的 B+-tree 索引管理算法相对较少，本文围绕基于 PCM 和 DRAM 混合主存的内存数据库索引管理算法展开，探讨如何优化现有的 B+-tree 索引算法。具体而言：

（1）基于读写预测算法的 OB+-tree 索引读访问优化算法

OB+-tree 可以有效的减少整体写次数，但是以读换写的策略在 PCM 读写不对称性逐渐缩小的趋势下，已经不再适用。所以我们在 OB+-tree 的基础上对其读次数进行优化。OB+-tree 的大量读操作主要来源于对溢出链的访问，溢出链越长，读次数越多。由于数据的读写访问倾向是具有集中性的，如果频繁读的结点在溢出链的底端，那么每次访问该结点都需要读取整个溢出链，造成高额的读代价。因此，我们将频繁读的结点从溢出链中取出，作为首层叶子结点；而频繁写的结点仍然以溢出链结构存储，这样，既发挥了溢出链较少写次数的优势，又减少了大量的读操作，从而提升整体的性能。而结点的读写访问倾向是根据结点的访问信息计算得出的。我们记录结点的历史访问统计以及最近一段时间的访问状态，通过计算和预测的方式得到结点未来的读写访问倾向。

（2）基于混合主存的 OB+-tree 索引读写优化算法

DRAM 与 PCM 混合主存结构，是用 PCM 代替 DRAM 成为主存的主要存储介质，但由于 PCM 的高写代价和寿命有限等特性，完全代替 DRAM 成为单一主存介质的可能性较低。DRAM 的读写代价相当且都较低，作为主存介质的一部分，虽然容量较小，但我们可以将少部分的写频繁数据存储在 DRAM 上，而 PCM 上存储大容量的读频繁数据。由于数据库对索引结点的读写访问倾向具有集中性，这样可以减少索引对 PCM 的大量的读写操作，同时也提升整体的性能。

1.4 本文组织结构

本文共分为五章，各部分的内容组织如下：

第一章：对新型存储器 PCM 的研究背景与意义做了介绍，并简要概述了 B+-tree 索引的研究现状。

第二章：介绍 PCM 的发展状况，并详细说明了国内外对基于 PCM 的 B+-tree 索引的研究状况，阐述了将已有的索引算法应用到 PCM 上的不适应性，并指出了已有研究的不足。

第三章：研究基于 PCM 与 DRAM 的溢出类型 B+-tree 索引的读访问优化算法。针对数据库对索引结点的读写访问倾向具有集中性，提出了基于 PCM 的

OB+-tree 索引优化算法 CB+-tree。首先，介绍了 CB+-tree 的结构框架，并详细给出了插入、删除、查询等操作的执行方法。然后详细阐述了结点读写倾向的计算以及预测算法。最后展示了实验的参数设置以及实验结果，证明我们提出的优化算法是合理的，可以提升了整体的性能。

第四章：研究基于 PCM 与 DRAM 混合主存的溢出类型 B+-tree 索引的结点迁移算法。由于引入了小容量高读写性能的 DRAM，且考虑到数据库对索引结点的读写访问倾向具有集中性，我们将频繁写的结点转移到 DRAM 上，而 PCM 仍然作为主要的存储介质，存储大部分读倾向叶子结点和内部结点。最后展示的实验结果证明，混合主存的结构下，B+-tree 索引的性能有巨大的提升。

第五章：对本文的工作内容进行回顾和总结。提出了研究中的不足之处，并为下一步工作的开展做铺垫。

第2章 相变存储器及其索引管理的研究现状

2.1 引言

材料技术的进步促进新型存储器的飞速发展，国内外许多研究者开始关注以相变存储器 PCM 为代表的非易失存储器。近年来，闪存（flash memory）、磁性存储器（magnetic RAM，MRAM）、铁电存储器（ferroelectric RAM，FRAM）、相变存储器（phase change memory，PCM）等新型存储技术的出现，为研究适合高效能大数据存储和管理的新型存储架构带来了新的机遇[1]。有文章统计[18]，研究 PCM 的文章数量已经超出了对闪存研究的论文数目。PCM 是一种非易失型存储器，与传统主存介质 DRAM 相比，具有非易失性、可字节寻址、空闲能耗低，集成度高和容量大等特点。作为最有发展前途的新型存储器，PCM 有望代替 DRAM 成为新的主存介质，创造出新的存储体系架构。

对数据库索引的研究一直都是数据库领域的研究热点之一。由于索引存放在主存，数据库的访问性能和主存介质的特性息息相关。找到适合主存介质的索引管理算法，可以大大提高数据库的访问性能。由于 PCM 与 Flash 的不同特性，原有基于闪存的索引管理算法并不适用于 PCM。当前的基于 PCM 的索引研究着重于减少索引对 PCM 的写次数，主要有调整索引结构和利用 DRAM 作为辅助存储两种途径。

本章讨论 PCM 的特点以及当前已取得的有关 PCM 的 B+-tree 索引的国内外研究成果。在第 2.2 节中首先介绍了 PCM 的发展及其特点；在第 2.3 节中介绍了以 PCM 为主存介质的 B+-tree 索引的优化算法的国内外研究成果，并指出他们的优缺点；最后在第 2.4 节中做本章小结。

2.2 相变存储器 PCM

2.2.1 PCM 概述

相变存储器（Phase Change Memory, PCM）是一种由硫系玻璃材质构成的非易失类型的存储器。硫系玻璃材质可以通过电脉冲加热的方式在无定形和结晶这两种状态之间进行变化，从而可以实现二进制数据的记录和存储。在无定形状态，

材料粒子是高度无序的，会产生高电阻率，也就是二进制中的“0”；在结晶状态，材料具有长程原子序，所以电阻率极地，也就是二进制中的“1”。两种电阻率的差异多达 6 个数量级，因此理论上每个记忆单元可以保存除 0 和 1 以外的第三种状态，这使 PCM 对内存应用程序非常有吸引力。每个记忆单元在被改变状态后会持续存在，因此关闭电源也可以保持信息。

早在四十多年前，PCM 就已经被发明出来了，但由于材料与电力消耗的问题，PCM 一直未被普及。直到近几年，随着技术的不断提高，PCM 的生产成本大大降低、集成度不断提高，加之其非易失低能耗等优点，PCM 才被业界广泛的关注。

表 2.1 新型存储器件的特性比较[19]

Storage Devices	Technology Node/nm	Memory area/μm^2	Write Energy	Read Energy	Number of rewrites	Write speed/ns	Read speed/ns	VCC/V
Flash	27	0.00375	17.5J/GB	1.5J/GB	10^5	106~10^5	50	2.7~3.6
FeRAM	130	0.252		5~10J/GB	10^{10}	83	43	1.9
STT-RAM	54	0.041		4~20J/GB	10^{15}	<15	<20	1.8
DRAM	130	0.168	1.2J/GB	0.8J/GB	10^{15}	1~10	100	3~4
PCM	45	0.015	6J/GB	1J/GB	10^{10}	100~500	50	1.8

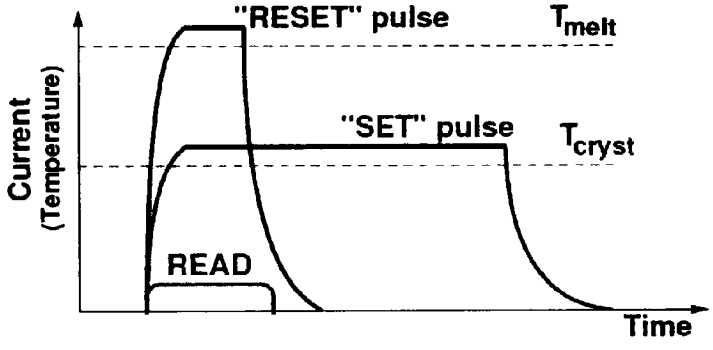


图 2.1 PCM 单元读、set 和 reset 操作所需时间和温度

表 2.1 是 PCM 与其它新型性存储器件特性的对比数据。由表中可见[19]，PCM 在集成度、读速度方面表现良好，在主存和存储系统的构建方面具有很大优势，是潜在新型存储器件的候选对象。但是，其缺点也是显而易见的：PCM 的写延时比DRAM大一个数量级[20]且PCM的写寿命有限。要想有效的使用PCM，当下最大的挑战就是解决其写操作的相关问题[7]：（1）高能耗：由图 2.1 可知，与读取一个 PCM 单元相比，写操作需要更高的电流和电压，并且消耗更多的时间，通常写操作能耗是读操作能耗的 6-10 倍。（2）高延时和低带宽：在 PCM 设备中，写操作延时主要由相对时间更长的 set 操作决定，约是读操作的 3 倍（见图 2.1）。此外，PCM 设备写操作是通过加热材料来更改存储单元的电阻，许多 PCM 原型机限制每次写的比特数，以控制加热所需的瞬时电流的峰值。这种控制技术在未来也一样存在，这是由 PCM 材料特性决定的，在有能耗限制的应用平台上尤为重要。由于写带宽的限制，写 64 字节的数据已经需要好几个写周期，加重了 PCM 写延迟问题。（3）写寿命有限：现存的 PCM 设备每单元的写次数寿命约为10^{10}。当磨损均衡效果较好时，在一般的工作负载下 PCM 可以工作几年的时间。然而，磨损均衡算法必须在内存控制器模块内执行，要求磨损均衡算法代价小且速度快，实际使用的磨损均衡算法都非常简单，而这又必然降低了均衡效果。比如[21]，在一块 16GB 容量的 PCM 上频繁更新一个计数器，以 4GHz 的机器性能，1 分钟内就可以将 PCM 磨损殆尽，而使用了磨损均衡算法后，可以正常工作 4 个月。

现有的计算机存储体系结构是层次结构，越往上级的存储芯片访问时钟周期延时越小，如一级缓存使用 SRAM；越往底层的芯片访问延时越长，通常外存使用 Flash 或者 HDD。图 2.2 给出了不同的存储介质的访问性能及其在层次存储体系架构中的角色[22]。由图中可以看出，DRAM 和 HDD 之间存在着巨大的性能差异，而 Flash 拥有比 DRAM 更高的集成度和比 HDD 更小的访问延时和更低的能耗[19]，衔接二者使性能差异缩小，因此基于 Flash 的 SSD 已经得到广泛应用；但是 Flash 的读速率仍然比 DRAM 低 2~3 个数量级，因此需要加大主存容量来减少对 FlashHDD 的访问。然而 DRAM 正面临集成度和能耗问题的瓶颈。

PCM 的出现刚好为解决这一问题提供了契机。PCM 具有与 DRAM 相当的读速度，但 PCM 具有高集成度，可以作为大容量主存介质；且 PCM 空闲能耗低，大容量主存的能耗问题得以解决，这些都恰好弥补了 DRAM 的缺点。所以 PCM 是代替 DRAM 作为主存介质的最好的选择。

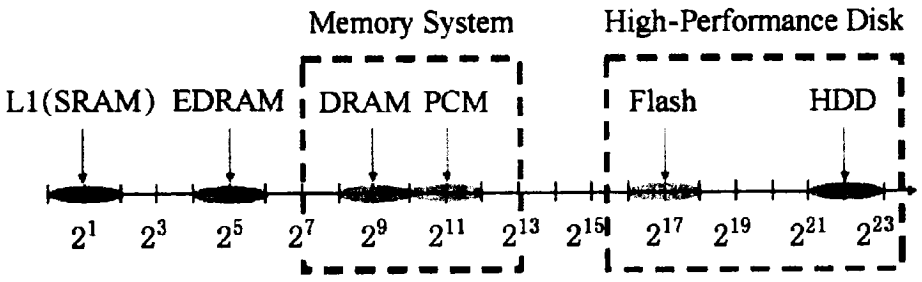


图 2.2 典型存储器件的访问时钟周期延时[22]

2.2.2 PCM 主存系统架构

目前，构建 PCM 主存主要有以下 3 种结构：仅仅以 PCM 作为主存、PCM 和 DRAM 作为平级介质的混合主存以及以 DRAM 作为 PCM 的缓存方式的混合主存。

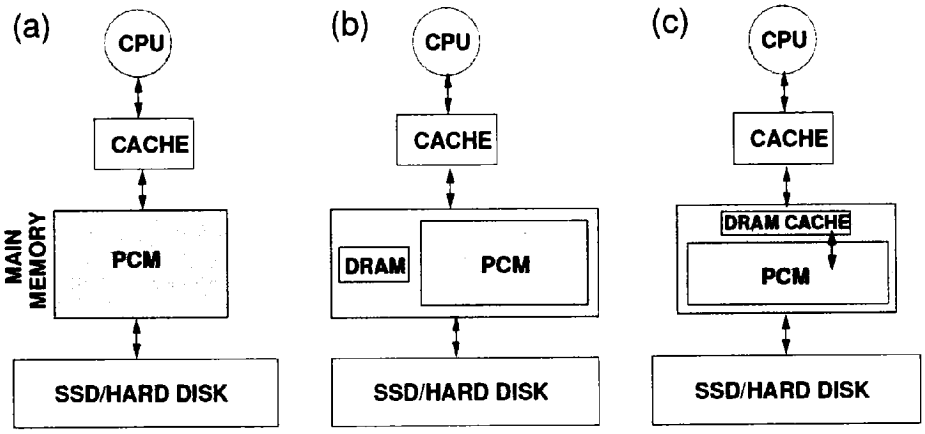


图 2.3 PCM 做主存的 3 种结构[6]

第一种以 PCM 完全替代 DRAM 是最早讨论的架构方法。在这种架构下，B+-tree 索引优化的着重点在于减少 PCM 的写和磨损均衡。为了尽量减少写，可以利用 PCM 的字节访问特点，文章[23]通过写前读的方式仅仅修改有改变的比特位，使得 SLC PCM 的寿命延长了 4.5 倍；也可以通过改变 B+-tree 索引结构的方式来减少对 PCM 的写。

第二种是以 PCM 和 DRAM 同时作为主存的架构。由于 DRAM 和 PCM 的不同性质，索引优化可以从不同读写访问类型的数据划分方法出发，如果能将频繁写的数据存放在 DRAM 上，那么就能大大降低 PCM 的写负载，从而提升整体

性能。但是考虑到 DRAM 的容量有限，不能将大量数据迁移去 DRAM，所以对迁移数据的选择非常重要。本文的后续研究正是基于第二种架构。

而第三种是以 DRAM 作为 PCM 的缓存的架构，利用 DRAM 的特性来提升性能并延迟对 PCM 写，通常 DRAM 存在于 PCM 设备内部。

2.3 基于 PCM 的 B+-tree 索引的研究

基于 Flash 设备的 B+-tree 索引的研究成果丰富，但是基于 PCM 为主存的 B+-tree 索引的研究目前较少。在大数据环境下，DRAM 的能耗和容量的限制，让它无法适用于大容量主存系统。而 PCM 作为新型存储器，具有很大的发展潜力，低能耗和大容量让它有望代替 DRAM 成为下一代主存介质。

传统的主存算法的设计目标主要有以下两点[7]：（1）低复杂度；（2）良好的 CPU 性能。基于以 PCM 为主存的架构，内存数据索引优化的主要目标是减少 PCM 写。由于没有其他存储介质，我们可以利用其字节读写特性，比如通过减少每次写的比特位来延长 PCM 寿命，但这种方法只是单纯的从材料特性角度出发，并没有考虑到索引结构的特点。对 B+-tree 索引来说，我们可以通过改变其结构，进而优化读写操作对索引带来的修改，最终减少 PCM 写。

传统 B+-tree 索引的记录全部存储在叶子结点中。其内部结点和叶子结点的记录都是有序的，因此在访问结点时，内部可以使用二分查找算法提高效率。对于查询操作，从根结点开始，每次使用二分查找方法找到当前结点中比给定 *key* 大的第一个值，再继续查询该值对应的子结点，直到找到叶子结点，并判断给定 *key* 是否存在。在结点内部都有序的情况下，查询操作的效率非常高。对于插入操作，首先用查询的方法找到要插入的 *key* 所在的叶子结点，插入后若结点满，则进行分裂操作，一次分裂操作会造成对原叶子结点、新叶子结点和父结点的写操作；若插入后不需要分裂，为了保持结点内部记录有序，也需要进行排序操作，平均写长度为 $L/2$（L 为叶子结点最大记录数）。对于删除操作，在找到对应的叶子结点删除对应的记录后，若结点内记录数不足一半，则进行合并操作或者与邻居结点进行调整，这两种操作都会造成对当前结点、邻居结点和父结点的写操作；若删除后不需要调整结构，结点内部排序操作仍不可避免。由此可见，在 PCM 这样的高写代价的主存上，传统 B+-tree 具有较高的查询效率，而插入/删除操作效率及其低下。

由于排序操作在插入和删除记录时都无法避免，文章[6]提出了无序结点的思想。让每个结点内部的记录不必有序，插入操作直接将记录并入结点尾部，而

删除操作将结点尾部记录移动到被删除记录的位置即可，而分裂与合并等操作仍与传统 B+-tree 相同。这样做的好处是完全避免了记录排序带来的写操作，从一定程度上减少了写次数。对于无序结点，其第一个字节可以记录结点中的记录个数，也可以将这 8 个比特当做位图，比特为“1”表示其对应的位置有记录，称为位图无序结点。插入记录时，位图无序结点找到第一个比特为“0”的地方作为插入位置，而删除操作仅仅需要把对应的比特置为“0”。文中提出了 3 种无序结点方案：全部结点无序、仅叶子结点无序、仅叶子结点位图无序。实验通过与传统 B+-tree 的比较，证实了无序的方案确实可以降低写次数。仅叶子结点无序方案在仅插入操作中表现最好，可以大大降低写次数；而仅叶子结点位图无序方案在仅删除操作中性能最优，因为对单独一个结点的删除操作只需要修改其位图信息，仅造成一个比特的更改。但是这些无序方案却让查询时的二分查找方法无法适用。结点内部的记录无序，导致每次访问一个结点都需要将结点的全部信息读入缓存，在结点较大时会造成巨大的额外读操作。这种方法的思想是以读换写。

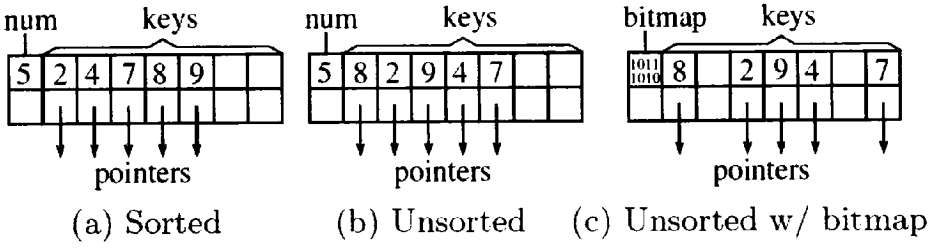


图 2.4 B+-tree 结点的组织方式

而文章[16]提出，无序结点的方法仍然无法避免结点分裂时的排序操作，且当结点较小（结点少于 4 个 cacheline）时，频繁的分裂让这种方法几乎无法降低写次数。针对这样的问题，文章[16]提出了一种半平衡机制。在结点分裂时，传统 B+-tree 需要对结点内所有记录进行排序，再平分到两个新结点中以保证每个结点至少是半满状态。为了减少排序操作，半平衡机制允许分裂后的结点不达到半满状态，只需要在分裂时找到一个中间值，让小于该值的记录留在原结点，而大于该值的记录才写入新结点。这种方法降低了结点分裂时的排序操作，在仅插入操作的实验中表现良好。但是，允许不半满的结点存在就意味着会浪费大量空间。文中提出的分裂时的中间值仅仅是简单取最小和最大记录的中点值，没有理论根据，且结点中需要时刻记录最小值和最大值，浪费了存储空间。

文章[16]还提出了溢出结点的思想 OB+-tree。溢出结点方法是对插入/删除操作造成的对上层结点的写进行优化的。由于每次分裂/合并/借元素操作都会对上层结点造成写操作，那么降低这些操作的次数也可以降低写次数。让每个叶子结

点在分裂时产生的新结点放在该结点后面，成为一条溢出链，而不需要对父结点进行修改；在合并/借元素时，优先从溢出链内部寻找结点进行操作，同样也避免了向上传递写操作，如图 2.5 所示。只有当溢出链长度达到上限时才将溢出链拆分成若干个传统的叶子结点，那么对父结点的多次写操作被合并为一次写，从而降低了写次数。文中的实验证明了这种方法对降低写次数的有效性，并且溢出链越长，写次数减少的越多。然而每次访问溢出链中的结点时需要读取其前面的所有叶子结点，平均额外读取的叶子结点数目为*(L+1)/2*（*L* 为溢出链长度），增加了许多额外读，并且会随着溢出链长度的增加而逐渐降低查询性能。这也是一种以读换写的思想。

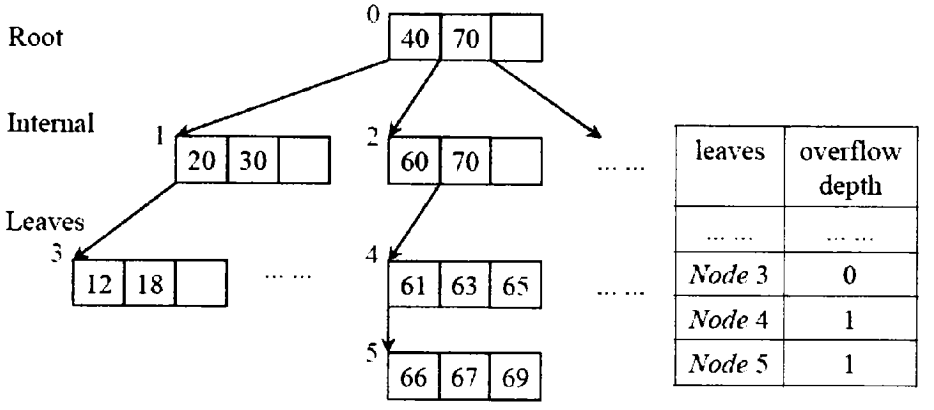


图 2.5 溢出链长度为 2 的 OB+-tree 索引结构

在 PCM 发展初期，写代价远远高于读代价，以读换写是一种有效提升整体性能的方法。但是随着技术的发展，写代价在逐渐降低，读写差异正在缩小，一味的以读换写可能会导致读代价太大，系统性能无法提升，甚至下降。

PCM 具有非易失、大容量、低能耗、可字节寻址和良好的读性能等优点让 PCM 有望代替 DRAM 成为下一代主存。但是，PCM 的缺点也非常明显：写寿命有限，而且写代价太高，读写不均衡。虽然其写寿命可达到10^8次，相比 Flash 已经延长许多，但是我们仍然希望能够尽可能减少 PCM 的写磨损状况，这样可以提高性能并延长使用寿命。而 DRAM 具有良好的写性能，且不具有写寿命限制。我们理所当然的想到让 DRAM 与 PCM 相结合，构成混合主存结构，这样既可利用 PCM 的大容量和低能耗的优点，也可以发挥 DRAM 的高效写优势，从而使系统性能提升。

在 DRAM 与 PCM 的混合主存架构中，DRAM 是作为 PCM 的上级缓存，或者与 PCM 共同作为平级主存介质。目前还没有人研究以 DRAM 和 PCM 作为同级主存介质的 B+-tree 索引优化，所以我们接下来讨论以 DRAM 作为 PCM 缓存

的索引优化方法。

文章[7]提出了 Bp+-tree，是一种使用预测机制的 B+-tree 索引。首先，使用一个小容量的 DRAM 来维护一个小的传统 B+-tree 树，这个 B+-tree 记录的是当前的若干插入操作。除了传统 B+-tree，DRAM 上还维护了一个小型柱状图。柱状图将所有的 *key* 的范围平分成 *B* 个部分，每次插入或者删除后，都要更新相应范围内的 *key* 的数量 *ANO*。同时，也要根据总的 *key* 个数来预测将来落在每个范围内的 *key* 的总数 *PNO*。这个柱状图主要用于后续的预测算法。Bp+-tree 结构如图 2.6 所示。

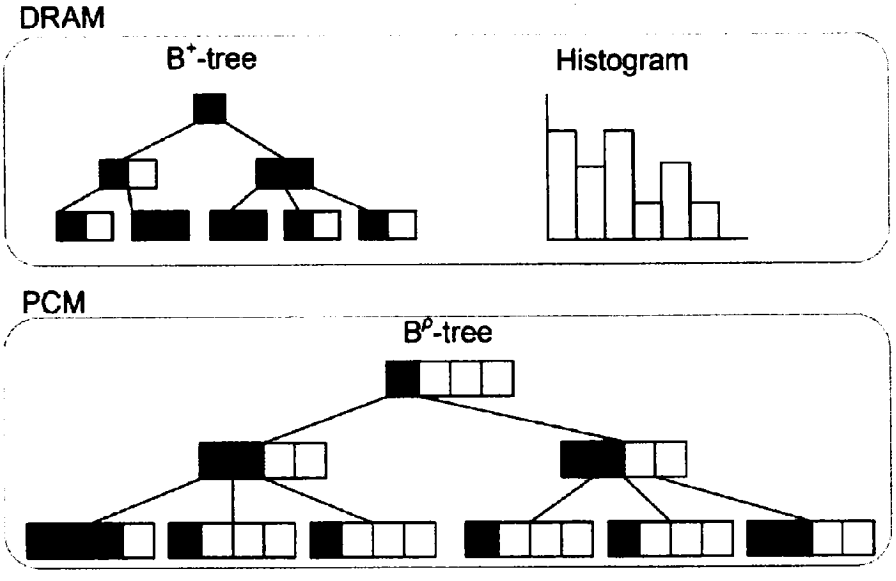


图 2.6 Bp+-tree 结构图

当 DRAM 满时，就要将 DRAM 上的 B+-tree 结点迁移到 PCM 上。但是每次迁移一个结点，都要根据 ANO 和 PNO 的值来确定如何迁移。如果 *ANO>PNO*，表示未来落在这个区域内的 *key* 比现在已经有的少，所以不用分裂结点直接迁移至 PCM；若 *ANO<PNO*，表示未来很有可能还有很多 *key* 会插入到该结点，那么我们就将该结点提前分裂，作为两个新的结点迁移到 PCM 中，这样就可以减少未来由于分裂操作而造成的写操作。

查询操作需要同时查找 DRAM 上的 B+-tree 和 PCM 上的 Bp+-tree，如果找到则返回结果，否则返回“NULL”。而对于删除操作，首先在 B+-tree 中寻找要删除的记录，若找到则直接删除即可，由于是在 DRAM 上操作，不会对 PCM 造成读写负载；若 B+-tree 中没有，则从 PCM 上的 Bp+-tree 中寻找对应记录并删除，若删除操作造成结点不满足半满条件，并不执行合并/借元素操作，理由有二：

（1）合并/借元素会造成对上层结点的写负载；（2）结点的空闲位置大，可以减少以后插入分裂操作，进而减少分裂写代价。

实验证明，Bp+-tree 可以有效的减少 PCM 的能耗和写次数。但是，Bp+-tree 的预测机制需要事先知道所有 key 的范围和总数，实际应用范围较窄。PCM 上较多的未达半满的结点会造成空间的浪费，且 Bp+-tree 的叶子结点内部记录是无序的，查询操作需要读取整个叶子结点，这些都会大大增加查询代价。系统以 DRAM 作为缓存，数据一致性无法保证，一旦发生系统掉电，DRAM 上的数据将丢失。

2.4 本章小结

本章首先介绍了相变存储器 PCM 的发展及特点，包括当下大数据环境下的对主存介质的要求以及 DRAM 所面临的瓶颈，PCM 代替 DRAM 成为新一代主存介质的合理性及可能的体系架构，并分析了各种架构的主要特点和索引的优化方向。

然后，总结了目前在仅以 PCM 作为主存的架构下的 B+-tree 索引优化算法，主要是通过调整索引结点内部结构和索引整体结构这两种方法，并给出了这些方法的优缺点。然后讨论了以 DRAM 作为 PCM 缓存的主存架构下的 B+-tree 索引优化算法，主要是通过预建立索引和预测方法的结合，减少叶子结点的分裂与合并造成的写操作。指出方法的使用局限性和数据一致性问题。

以 PCM 作为新的主存介质，突破原有的主存体系架构，是存储管理领域的研究热点。在后续章节，本文将针对已有的 B+-tree 索引优化算法的问题，提出在 DRAM 和 PCM 作为同级存储介质架构下的解决方案，并验证方法的可行性。

第3章 基于混合主存的 B+-tree 索引读访问优化

3.1 引言

索引技术一直是数据库领域的研究热点。而数据库索引优化对数据库性能的提升有着至关重要的作用。目前已经有许多对基于 Flash 的 B+-tree 索引优化的研究，但是随着大数据时代的到来，传统的 DRAM 主存无法满足大容量低能耗的需求，研究人员将目光转向以 PCM 代替 DRAM 构成新型主存架构的研究。而 PCM 作为主存的主要问题是写代价高，读写不均衡。目前国内外已经有一些研究者对 PCM 为主存介质架构下的 B+-tree 索引优化提出了自己的办法，主要使用的是“以读换写”的策略。在读写代价差异巨大的前提下，实验结果表明这些方法确实可以降低算法的写次数，提升整体性能。然而，随着技术的发展，PCM 的写代价在逐渐降低，读写不均衡的差异正在逐步缩小。事实上，以增加读次数来换取写次降低的策略已经渐渐失去了合理性，算法性能提升很小甚至略有下降。也有研究者提出以 DRAM 作为 PCM 的缓存，通过结构上的优化减少 PCM 的写。但是这种方法的前提条件较苛刻，实际应用范围受到局限。一般情况下，在单 PCM 的主存系统中，B+-tree 写次数降低必然带来读次数提升。那么我们考虑使用 DRAM 作为辅助存储，来帮助提升 PCM 主存索引的优化。

当前，在 DRAM 与 PCM 结合的平级存储架构下的索引优化还相对缺乏。这种主存架构的优势在于：可以充分利用 DRAM 的高效写性能和 PCM 的大容量低功耗优势。从这个角度出发，本章提出了基于 DRAM 和 PCM 作为平级主存介质的 B+-tree 索引优化算法，其中 DRAM 作为 PCM 的辅助存储。在前人的研究基础上我们发现，OB+-tree[16]可以有效减少索引写次数，但是缺点是读次数大量增加。在保证写次数降低的前提下，我们提出了优化读次数的索引算法。本章的主要内容有：

（1）鉴于 OB+-tree 的写负载低，并结合结点的读写访问倾向性的判断，提出了一种新的索引结构调整算法 CB+-tree，以优化其读次数，提升整体性能。

（2）结合线性模拟预测方法，提出一种结点读写倾向性的判定算法。通过记录结点的最近若干次访问状况和历史读写信息，判断结点在未来的读写倾向性，为索引结构调整提供参考信息。

（3）利用 BenchmarkSQL 来产生在 PostgreSQL 上运行的 TPC-C 工作负载实验使用的数据，并与若干 B+-tree 索引算法进行对比。实验验证了算法是可行的有效的。

本章的后续内容安排如下：在第 3.2 节中，分析 B+-tree 和溢出结点机制的性能；在第 3.3 节中，详细介绍了 CB+-tree 索引结构；在第 3.4 节中介绍结点读写倾向性的判断预测算法；在第 3.5 节中介绍了 CB+-tree 算法的具体操作；在第 3.6 节中给出实验设计与结果分析。第 3.7 节总结本章工作。

3.2 B+-tree 和溢出结点机制

传统 B+-tree 是面向 HDD 设计的[5]，每个内部结点或叶子结点都是一个页面，刚好为系统的访问粒度。在查询过程中，B+-tree 每向下查找一次只读取一个结点（页面），所以每次查询操作的读次数就是树的高度。B+-tree 的写操作主要包括插入和删除，通常更新操作是由“删除-插入”操作组合完成的。插入/删除操作发生在叶子结点，当叶子结点需要分裂/合并时会将写操作向上传递到内部结点，所以更新叶子结点会造成原发写操作，而由叶子的分裂/合并操作触发的额外写操作是内部结点的主要更新方式。

溢出机制设计的出发点是减少 B+-tree 内部结点的写操作。每个叶子结点可以拥有自己的溢出链，相当于将叶子结点扩容好几倍。当叶子结点发生分裂/合并操作时，优先从溢出链内部选择邻居结点，这样就不必向内部结点传递写信息，从而降低写次数。但是，查询操作将变得低效，为了访问到目标记录所在的溢出结点，必须要先读取连接到它的前面所有的溢出结点，因为溢出链只能顺序访问，而不能直接定位。这样，会造成大量的额外读操作，也就是“以读换写”思想。设最大溢出链长度为 K，那么每次查询的读代价期望可以由公式 3.1 计算得出[5]，可见这会造成非常高的读代价。这是在以页面为读写单位下的读代价期望，在基于 PCM 的 B+-tree 索引中，每次访问索引结点只读取需要的 cache-line 而不是整个结点，这样会让读代价进一步增加。在 PCM 读写不对称性减弱的趋势下，过多的读操作仍然会降低整体性能。实现读写代价均衡，成为“以读换写”的溢出机制所不可避免的问题。

$$E_{overflow} = \sum_{i=1}^{K} \frac{i}{K} = \frac{K+1}{2} \tag{3.1}$$

在上述的 B+-tree 算法中，设计者一直没有把 cache-line 的概念引入 B+-tree 索引。本章讨论的是基于 PCM 作为主存的索引算法，而 PCM 是可字节读写设备，系统对 PCM 的读写都是以 cache-line 为单位。所以，将结点的设计与 cache-line 结合，建立更具体的索引结点结构，代替以页面为单位的结点访问粒度，也是优化整体性能的方法之一。

3.3 CB+-tree 索引结构

本小节首先介绍 CB+-tree 算法设计的理论依据，然后描述 CB+-tree 索引的基本结构和结构调整方法，最后讨论查询、插入、删除等操作如何进行。

3.3.1 CB+-tree 结构设计理论依据

表 3.1 CB+-tree 变量

项目	描述
w_p, r_p	PCM 的总写/读次数
l_{wp}, l_{rp}	PCM 的写/读延时
e_{wp}, e_{rp}	PCM 写/读一个比特的能耗
T	主存总耗时
E	主存总耗能
α	l_{wp}/l_{rp}
β	e_{wp}/e_{rp}

B+-tree 中的每个操作耗时都包括 CPU 时间、缓存延时和主存延时。在本章接下来的讨论中，我们以 DRAM 和 PCM 作为混合主存介质，其中索引存放在 PCM 上，而辅助信息存放在 DRAM 上。与主存代价相比，CPU 和 cache 的访问操作代价几乎可以忽略不计，所以我们只计算主存代价，也就是：

$$T = w_p * \alpha * l_{rp} + r_p * l_{rp} \tag{3.2}$$

$$E = w_p * \beta * e_{rp} + r_p * e_{rp} \tag{3.3}$$

从表 2.1 中我们可以看出，l_{wp}大约是 l_{rp}的 2 到 10 倍，在代价计算中我们令 α为 6（取 2 与 10 的平均数）来大致估算 PCM 的读写代价差别；而e_{wp}大约是 e_{rp}的 6 倍，所以令β也为 6。那么公式 3.2 和公式 3.3 就可以写成如下形式：

$$T = (6w_p + r_p) * l_{rp} \tag{3.4}$$

$$E = (6w_p + r_p) * e_{rp} \tag{3.5}$$

从公式 3.4 和公式 3.5 中我们可以看出，在以读换写的策略中，如果增加的读次数超过了减少的写次数的 6 倍，那么整体的能耗与时间反而会增加。换句话说，如果我们能够减少大量的读次数，而代价仅仅是很少的写次数的增加，只要读写变化比不超过 6：1，就可以让整体性能提高，降低时间和能耗。因此，我们的目标是设计一个基于 PCM 的混合主存系统的的读优化 B+-tree 索引。

3.3.2 CB+-tree 索引结构

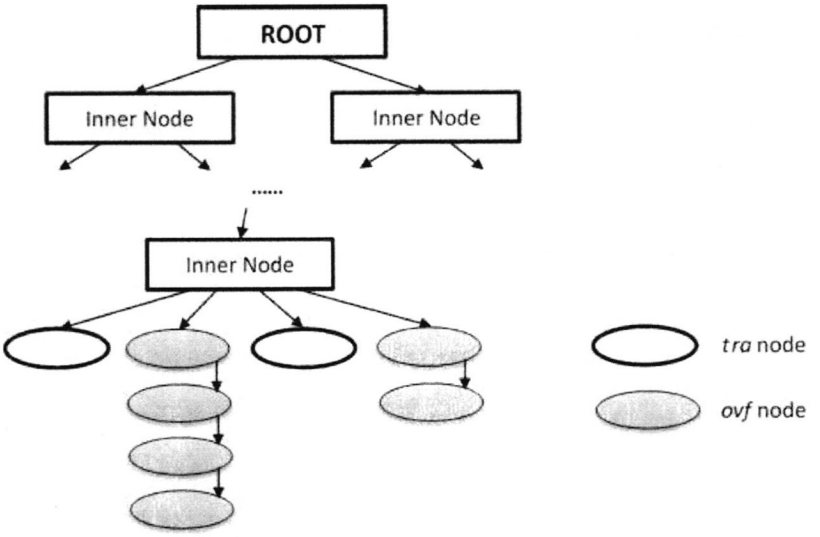


图 3.1 CB+-tree 结构示意图

图 3.1 表示一个 CB+-tree 的结构。CB+-tree 的结构与传统的 B+-tree 结构类似，不同之处在于一些叶子结点可以拥有其溢出结点，存在溢出链结构。我们将带有溢出链的叶子结点称为 *ovf*，而不带有溢出链的叶子结点称为 *tra*。

CB+-tree 中，所有的数据都存在叶子结点，这和传统 B+-tree 索引是一样的。所有结点内部记录都是有序的，所以查询操作可以使用二分查找方法。

CB+-tree 中所有结点都是由若干个 cache-line 组成的。我们设计每个结点的第一个 cache-line 为 *cacheline_h*，其他的 cache-line 称为 *cacheline*。在 *cacheline* 中，我们只存储<*key, value*>对；而在 *cacheline_h* 中，我们要存储一些辅助信息，余下的空间存储<*key, value*>对。在 CB+-tree 中，辅助信息包括 *num_keys*，*is_leaf*，*is_ovf*，*parent* 和 *brother*，分别代表结点元素个数、是否是叶子结点、是否为 ovf 类型结点、父结点和兄弟结点。这些辅助信息占用共占用 20 字节。

文章[23]中证实了 B+-tree 结点由若干 cache-line 组成时，在各方面表现最佳。对于现今的电脑，cache-line 一般是 32 字节、64 字节或 128 字节[16]。在本章的讨论中，我们令 cache-line 大小为 64 字节，那么 *cacheline* 中可以存储 3 对<*key, value*>对，而 *cacheline_h* 中可以存放 2 对。我们设一个结点共有 a（a≥2）个 cache-line，其中包括 1 个 *cacheline_h* 和 a-1 个 *cacheline*。可以算出，结点共有 2+3*(a-1)=3a-1 个元素，辅助信息占用的比例为 20/(a*64)=5/(16a)。

算法 3.1 展示了索引结构调整算法。

在 CB+-tree 中，我们认为 *ovf* 结点具有写倾向，而 *tra* 结点具有读倾向（读写访问倾向判断在 3.3 节叙述）。当我们判断一个 *ovf* 结点具有读倾向时，我们就需要进行结构调整。当一个 *ovf* 结点被判断具有读倾向时，我们将它从它所在的

溢出链中取出，作为单独的叶子结点插入到 CB+-tree 中，而原溢出链则从该结点位置断成两部分，分别在该结点的左边和右边。这样，我们就可以直接访问这个结点，而不需要读取它前面的溢出结点；且对该结点的读访问越多，我们节省的读次数也就越多。而随着访问的进行，该结点又被判断具有读倾向，那么我们只需要直接修改其类型标识，在下次分裂时就会产生溢出结点，它成为溢出链的头结点，从而减少对上层结点的写操作。

算法 3.1.结点类型转变函数 *change(ROOT, NODE)*。

输入：CB+-tree 根结点 ROOT 和目标结点 NODE
输出：CB+-tree 的根结点 ROOT

```
/*change type of NODE to ovf if it is tra*/
1. if NODE is tra then
2.     set its is_ovf to true;
/*change type of NODE to tra if it is ovf*/
3. else if NODE is ovf then
4.     set its is_ovf to false;
5.     find its first leaf node of the overflow chain;
6.     if NODE is not the only one node int the overflow chain then
7.         remove NODE from the overflow chain;
8. end if
9. return ROOT;
```

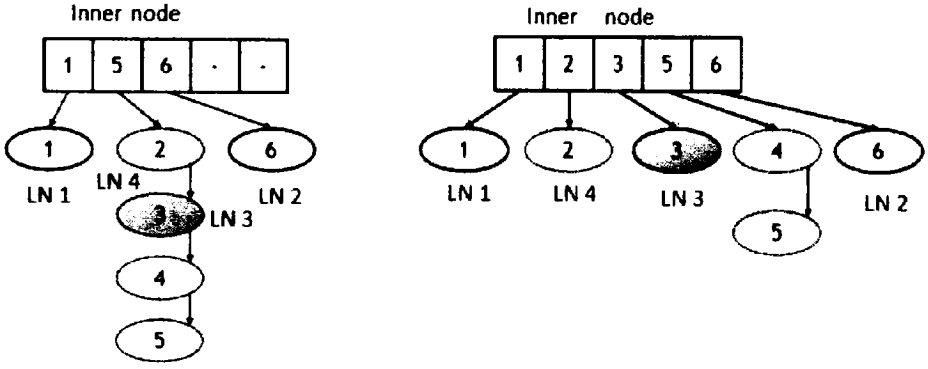


图 3.2 CB+-tree 结构调整

如图 3.2 左部所示，结点 1、结点 2 是 *tra* 结点，结点 3 是 *ovf* 结点，它在溢出链中处于中间位置。现在我们确定结点 3 具有了读倾向，那么就将结点 3 从溢出链中取出作为单独的叶子结点，而原溢出链则断成了两部分并插入到 CB+-tree 中，如图 3.2 右部所示。若此后结点 3 需要分裂，那么就不会产生溢出结点。而

原来的溢出链中的结点（比如结点 4）仍然会按照 *ovf* 结点的方式以溢出链内部调整为优先选择，进行分裂/合并操作。

3.4 读写倾向性判断

CB+-tree 索引中的 *tra* 结点和 *ovf* 结点决定了整体的结构。那么接下来我们讨论如何判断叶子结点的读写倾向性。

3.4.1 访问信息

在 CB+-tree 索引中，我们需要为每个叶子结点记录访问信息，这些信息是用来判断其读写倾向性的。每次访问都会涉及到至少一个结点的信息修改，这些数据的写频率很高。我们选择 DRAM 来存储这些数据，一来可以发挥 DRAM 的高写速度的优势，二来可以减少 PCM 的写。

表 3.2 叶子结点的访问信息

项目	描述
Q	滑动访问窗口
L	滑动访问窗口长度
r	历史读次数
w	历史写次数
is_ovf	结点类型

我们为每个叶子结点维护的访问信息如表 3.2 所示。滑动访问窗口是用来记录最近 *L* 次访问的读写顺序，其中比特“1”表示写访问，比特“0”表示读访问；*r* 和 *w* 分别用来记录结点的总读访问次数和总写访问次数；*is_ovf* 是用来记录结点类型，*is_ovf* 为“true”表示结点是 *ovf*，否则表示结点为 *tra*。

当读访问到达叶子结点，我们在滑动访问窗口尾部加入比特“0”，并将总读访问次数 *r* 加 1；若到达的是写访问，则在滑动访问窗口尾部加入比特“1”并将总写访问次数 *w* 加 1；当结点的类型改变时，修改 *is_ovf* 变量即可。对 CB+-tree 的每次操作都涉及到一个目标叶子结点 *t_l*，若有分裂或者合并/借元素操作，则还涉及到邻居结点 *n_l*。我们只对 *t_l* 和 *n_l* 叶子结点进行访问信息的更新。CB+-tree 的内部结点没有维护访问信息；对于溢出链中的叶子结点，如果是目标结点前面的叶子结点，虽然也会有读访问到达，但是我们不更新其访问信息。因为 CB+-tree 结构的合理之处在于，数据库中的数据读写访问具有集中性，也就是说，

对某个范围内的数据，其读写倾向性通常有很大概率是一样的；一个结点内的数据是相近的，所以我们可以判断出读倾向结点和写倾向结点。为了访问到目标叶子结点而读取的其他结点，并不是数据库操作本来要访问的数据所在的结点，只是索引结构引起的访问，所以对其读写倾向并没有影响。

3.4.2 读写倾向性

在维护了访问信息后，我们可以计算出叶子结点的写比例 *ratio*。而叶子结点的类型则与 *ratio* 有关。*ratio* 是以一定的权值来综合历史写比例和当前滑动窗口写比例计算得出的。

定义 3.1 叶子结点的写比例 *ratio* 的定义为

$$ratio = ratio_h * (1-\alpha) + ratio_c * \alpha \tag{3.6}$$

其中

$$ratio_h = w/(w+r) \tag{3.7}$$

$$ratio_c = qw/l \tag{3.8}$$

$$\alpha = l/L * \omega \tag{3.9}$$

ratio 可以由公式 3.6 计算得出。其中$ratio_h$和$ratio_c$分别代表历史写比例和当前写比例；公式 3.8 中 qw 代表滑动访问窗口 Q 中的比特“1”的个数（也就是最近 L 次访问中写访问次数），l 代表滑动访问窗口 Q 的当前长度；α代表当前写比例所占整体的比重，由滑动访问窗口 Q 的实际长度决定：若窗口是满的则为最大值ω（ω>0.5），否则按比例缩小比重，若当前 Q 为空，则$ratio_c$为 0。

这种计算方法可以将历史访问信息和最近访问信息结合在一起，并给予最近访问信息较高的权重α。这是因为数据的读写访问倾向可能会随着时间改变，因此近期访问记录具有较高的参考价值，故而给予较高的权重；而历史访问特点的参考性会随着时间的推移而降低，因此给予较低的权重，这样可以避免历史信息对结点访问特性判断的误导。而最近访问信息的权重α并不是固定不变的，它随着滑动访问窗口中的有效位个数的增加而增加，最小为 0，最高为ω，这样，当滑动访问窗口中数据过少导致了$ratio_c$计算不准确时，可以降低$ratio_c$对整体计算结果的影响。

定义 3.2 叶子结点的类型定义如下：

$$\text{叶子结点类型} = \begin{cases} tra, & ratio < \theta_1 \\ ovf, & ratio > \theta_2 \\ predict \begin{cases} tra\text{结点且 } 0.5 \ll ratio \ll \theta_2 \\ ovf\text{结点且 } \theta_1 \ll ratio \ll 0.5 \end{cases} \end{cases} \tag{3.10}$$

其中$0 < \theta_1 < 0.5 < \theta_2 < 1$。

定义 3.2 给出了叶子结点类型判断方法。当我们计算出的写比例低于较小阈值θ_1时，认为该结点写不频繁，故为 *tra* 传统型结点；当写比例高于较大阈值θ_2时，认为该结点是写频繁的，所以为 *ovf* 溢出型结点；而当写比例 *ratio* 介于两个阈值之间时（对于 *tra* 结点，其写比例 *ratio* 高于 0.5，但又不是非常大，仍然低于θ_2，达不到成为写倾向结点的条件；对于 *ovf* 结点，其写比例 *ratio* 低于 0.5，但又不是非常小，仍然高于θ_1，达不到成为读倾向结点的条件），读写倾向不明显，所以不能武断的为其判断类型，所以需要经过预测，来得知未来一段时间该结点的访问状况，然后根据预测信息再判断访问类型。

读写访问窗口 Q 为结点类型预测提供了主要数据。总的来说，我们是通过读写访问窗口 Q 中记录的读写顺序，预测出未来 l 次访问的读写情况，再根据未来 l 次访问的写比例与 Q 中访问写比例对比，判断未来的结点访问倾向：如果未来 l 次写比例 *ratio_*大于当前 *ratio*，表示结点未来的写访问会增加，所以判断为写倾向 *ovf* 结点；若 *ratio_*小于当前 *ratio*，表示未来写访问次数会降低，可以作为读倾向 *tra* 结点在索引中。

我们首先将滑动访问窗口 Q 中的 l 个值 a 做一次变换，得到 l 个新值 y，称为访问函数值：

$$y_i = \begin{cases} 0, & i = 1 \\ y_{i-1} + 2a_i - 1, & i = 2,3,\dots,l \end{cases} \tag{3.11}$$

如公式 3.11 所示，y 的初始值为 0；当下一次访问为读访问时，则将 y 值减 1；若为写访问，则 y 值加 1。而另一组数值 x 是从 1 开始逐渐递增 1 的长度为 l 的数组。将 x 和 y 对应起来，我们得到了 l 对<x, y>数值对。将这 l 对二元组作为线性拟合的输入数据，可以通过最小二乘法得到拟合函数表达式。拟合函数曲线图可以反映结点的访问趋势变化：当曲线上升时，表示滑动访问窗口中的“1”比较密集，该段时间内写倾向较高；曲线下降则表示读倾向较高；而曲线呈波浪形上下浮动时，表示读写访问较均衡，访问倾向性不明显。

拟合函数表达式可以准确拟合最近 l 次访问的情况，同样可以计算出未来 l 次的访问函数值，并画出曲线。如果计算出的未来 l 次函数均值比最近 l 次函数均值低，可以认为未来的写次数比当前的少，那么就预测结点在未来一段时间为读倾向性结点 *tra*；反之，则认为未来的写次数与当前相比有所增加，预测为写倾向性结点 *ovf*。

算法 3.2 展示了结点读写倾向性预测算法。

值得注意的是，我们并没有在 *ovf* 结点的写*比例 ratio* 低于θ_2时立刻进行预测，而是要等到 *ratio* 掉到 0.5 以下才进行，是因为我们不认为 *ovf* 会立刻进入读

倾向状态，等待一段时间可以得到更多最近的访问信息，再进行预测判断会更加准确；也许在等待时间内，结点写比例 *ratio* 又恢复到θ_2以上，这样也可以避免频繁触发预测机制。

算法 3.2.结点读写倾向性预测 *predict(Q)*。

输入：被预测结点的滑动访问窗口 Q
输出：被预测结点的类型
1. *Get y value from Q;*
2. *Calculate avg_o: the average y value;*
3. *Calculate the fitting curve according to x and y value;*
4. *Calculate avg_n: the average y value after the length of Q within l;*
5. **if** *avg_n* < *avg_o* **then**
6. **return** *tra;*
7. **else**
8. **return** *ovf;*
9. **end if**

3.5 CB+-tree 索引操作

在确定了 CB+-tree 索引结构后，下一步就是讨论索引的插入、删除和查询访问的具体操作方法。CB+-tree 与 B+-tree、溢出类型 B+-tree 结构较为相似，但在各种操作上需要考虑到叶子结点的读写倾向性，并及时更新结点的读写信息，判断结点的读写访问倾向。

3.5.1 插入操作

插入操作对索引结构的影响是在叶子结点分裂时造成的。传统 B+-tree 索引在叶子结点分裂时，直接将新的叶子结点作为原结点的兄弟结点，并更改上层父结点的信息，在频繁插入的环境下，会对上层结点造成大量的写操作。溢出 B+-tree 索引在叶子结点分裂时，将新的叶子结点作为原结点的溢出结点，存在溢出链中，这样就对父结点没有写操作的要求了，只有当溢出链长度达到上限，才会将溢出链拆开，对父结点的写一次性完成。但是溢出链太长会导致读代价太高。CB+-tree 的叶子结点分裂操作会根据叶子的不同类别而做不同的处理。

算法 3.3 展示了 CB+-tree 的插入操作流程。其中函数 *need_change()*是判断目标结点是否需要更改类型，见算法 3.4。

算法 3.3.对 CB+-tree 的插入操作 *insert(ROOT, key, value)*。

输入：要插入记录<*key, value*>和 CB+-tree 的根结点 ROOT
输出：CB+-tree 的根结点 ROOT

```
1. find the first leaf node f_l;
2. if f_l is tra then
3.    insert <key, value> into f_l;
4. else
5.    find target leaf node t_l in the overflow chain;
6.    insert <key, value> into t_l;
7.    if NODE need to split then
8.       split NODE and the NEW NODE has the same type as NODE;
9.    end if
10. end if
11. update the metadata for NODE;/*更新结点的读写访问信息，若执
    行过分裂操作，则也为新结点更新读写信息*/
12. if NODE split and NODE is tra then
13.    if need_change(NODE) = NEED_CHANGE then
14.       change(NODE);
15.    else if need_change(NODE) = NEED_PREDICT then
16.       predict(NODE);
17.    end if
18. end if
19. return ROOT;
```

算法 3.4.判断结点类型是否要改变 *need_change(NODE)*。

输入：被判断结点 NODE
输出：结点类型是否需要改变

```
1. calculate the ratio of NODE;
2. if NODE is ovf and ratio<θ1 or NODE is tra and ratio>θ2 then
3.    return NEED_CHANGE;
4. else if NODE is ovf and ratio<0.5 or NODE is tra and ratio>0.5 then
5.    return NEED_PREDICT;
6. else
7.    return NO_CHANGE;
8. end if
```

当我们需要插入<*key, value*>键值对时，我们首先找到目标叶子结点，将键值对插入结点中。如果结点需要分裂，我们创建的新结点与原结点类型相同：若为 *tra* 类型，则将信息插入到父结点中；若为 *ovf* 类型，则将新结点作为原结点的溢出结点，插入到溢出链中，若溢出链长度达到上限，则分裂溢出链成为若干个不带有溢出链的 *ovf* 类型叶子结点。

如果插入操作引起结点分裂，不论被分裂结点是何类型，其访问信息中的滑动访问窗口 Q 需要清空，历史读写次数 r 和 w 减半。滑动访问窗口中的每个访问记录都是针对结点中某个 *key* 而产生的，但结点分裂操作可能将产生访问记录的 *key* 分裂到新结点中，那么如果仍然保留已经不在结点中的 *key* 产生的访问记录，必然会误导结点类型判断，故将滑动访问窗口 Q 清空。而历史读写次数的统计是针对整个结点的，我们粗略的认为一个结点中每个 *key* 被访问到的概率是一样的，那么分裂出去一半的 *key*，总的读写次数也需要减半。

算法 3.5.对 CB+-tree 的删除操作 *delete(ROOT, key)*。

输入：要删除记录 key 和 CB+-tree 的根结点 ROOT
输出：CB+-tree 的根结点 ROOT

```
1.  delete key and its value from the target leaf node
2.  if need to merge or distribute node then
/*For an ovf, prefer to find an ovf neighbor*/
3.     if NODE is ovf then
4.        if NODE has overflow nodes then
5.           find neighbor in overflow chain;
6.        else if NODE has ovf neighbor then
7.           find ovf-type neighbor;
8.        else
9.           find tra neighbor;
10.       end if
/*For a tra, prefer to find a tra neighbor*/
11.    else
12.       if NODE has tra neighbor then
13.          find a tra neighbor;
14.       else
15.          find an ovf neighbor;
16.       end if
17.    end if
18. end if
19. update the metadata for NODE;/*更新结点的读写访问信息，若先
    前修改过邻居结点，则也为邻居结点更新读写信息*/
20. if NODE merge or distribute and NODE is tra then
21.    if need_change(NODE) = NEED_CHANGE then
22.       change(NODE);
23.    else if need_change(NODE) = NEED_PREDICT then
24.       predict(NODE);
25.    end if
26. end if
27. return ROOT;
```

3.5.2 删除操作

对于删除操作，主要考虑在发生叶子结点合并/借元素时如何选择邻居结点，否则只需要直接删除目标结点中的对应记录即可。算法 3.5 展示了删除操作具体过程。

若要删除的记录为<*key, value*>，首先找到记录所在目标叶子结点并删除记录。若需要进行合并/借元素操作，我们优先选择与目标叶子结点类型相同的结点作为邻居结点：若结点是 *ovf* 类型，那么由于溢出链内部调整可以减少对上层结点的写操作，所以优先在溢出链内部选择邻居结点；如果该结点不含有溢出链，那么选择左右兄弟结点中是 *ovf* 类型的结点作为邻居结点；若仍然没有找到符合条件的邻居结点，那么只能选择 *tra* 类型结点作为邻居结点。若结点是 *tra* 类型，那么首先选择 *tra* 类型的兄弟结点作为邻居，若找不到则选择 *ovf* 类型邻居结点。我们优先选择相同类型的结点进行结构调整，是因为我们希望把具有相同访问倾向的数据存到同一个结点中，这样可以让结点读写访问倾向更明确。删除操作需要对目标叶子结点的读写信息进行更新：若发生合并操作，那么新结点的滑动访问窗口 Q 需要被清空，历史读写次数 r 和 w 是两个原结点的总和；若发生借元素操作，那么只需要向原结点的滑动访问窗口插入比特“1”并将总写次数 w 加 1 即可，邻居结点的读写访问数据不做修改。

3.5.3 查询操作

算法 3.6.对 CB+-tree 的查询操作 *search(ROOT, key)*。

输入：要查询记录 key 和 CB+-tree 的根结点 ROOT
输出：CB+-tree 的根结点 *ROOT*，查询结果 *value*

```
1. find the first leaf node;
2. if the first leaf node is tra then
3.     search within the target leaf node;
4. else
5.     find the target leaf node in the overflow chain;
6.     search within the target leaf node;
7. end if
8. update the metadata for target leaf node;
9. if need_change(NODE) = NEED_CHANGE then
10.    change(NODE);
11. else if need_change(NODE) = NEED_PREDICT then
12.    predict(NODE);
13. end if
14. return ROOT;
```

CB+-tree 的内部结点结构与传统 B+-tree 是一样的，所以只需要根据叶子结点的结构来调整叶子部分的查询算法即可。

算法 3.6 展示了查询操作的具体过程。当我们要查询关键字 *key* 时，首先利用二分查找方法找到第一层叶子结点。如果该叶子结点是 *tra* 类型，那么只要在该结点内部查询即可；若是 *ovf* 类型，那么目标记录可能存在溢出链中的其他叶子结点里，所以我们每次比较当前溢出结点的最大值，若 *key* 大于当前结点最大值，那么就继续访问下一个溢出结点，直到找到目标结点。我们只对目标叶子结点进行访问数据的更新，并判断目标结点是否需要更改类型。

3.6 实验设计及结果分析

在本小节中，我们首先阐述实验的环境建立及数据来源；然后对预测算法进行实验验证，证明预测机制对结点读写倾向性预测的准确性；最后通过模拟数据，比较我们的算法和已有的基于 PCM 的索引算法：B+-tree、溢出类型 B+-tree 在减少 PCM 整体写代价上的优势，其中 B+-tree 和溢出类型 B+-tree 都只使用了 PCM 介质，在本文 2.3 节中已经详细阐述过。在实验中，我们只统计 PCM 的读写数据，并给出在各种不同参数下的实验结果，分析实验结论。

3.6.1 实验环境与数据集

实验使用的机器环境参数如表 3.3 所示。

表 3.3 实验环境参数

参数	值
操作系统	Ubuntu 14.04
CPU	AMD Athlon II X2
主存	4GB RAM
硬盘	1TB Seagate

我们采用模拟实验来验证 CB+-tree 索引的性能。由于目前还没有 PCM 和 DRAM 的混合模拟平台，所以我们利用文件来模拟 PCM 和 DRAM，并分别统计其读写次数；利用内存来模拟 cache，使用 LRU 算法管理；算法时间由读写次数乘以介质对应的读写延时来得到。

实验中使用的数据集，是由数据库领域知名的 TPC-C 基准测试的官方工具产生的。TPC-C 负载包含了许多利用到底层索引的查询访问，因此我们可以用它

测试出不同索引的性能差异。运行 TPC-C 工作负载时需要配置 10 个数据集和 100 个客户端。TPC-C 工作负载共包含 8 个索引，分别建立在 8 个表上，每个表数据大小约为 1GB。我们利用 BenchmarkSQL 来产生运行在开源数据库 PostgreSQL 上的 TPC-C 工作负载，并同时记录下对这 8 个表的页面请求。

首先，我们执行插入操作来建立这 8 个索引。建立索引的插入操作大约有 540 万条，建立的索引文件大约占 135MB。在创建索引之后，我们执行对索引的访问请求，共有约 380 万次访问，其中查询访问占 74.2%，插入访问占 23.8%，删除访问占 1%。我们记录并比较分析在这个阶段各种索引算法对 PCM 的读写频繁度、执行时间等性能。

3.6.2 预测算法结果分析

CB+-tree 中，叶子结点的类型判断主要由结点的写比例 *ratio* 决定，但当写比例处于中间大小时，我们需要通过预测算法来决定叶子结点的读写访问类型。可见预测算法的准确性在整个 CB+-tree 索引结构调整中起着至关重要的作用。在本小节我们给出验证预测算法的实验结果，并证明其准确性。

预测结点的读写倾向，需要根据结点的访问滑动窗口中的数据，得到新的一组值 y，我们称之为访问函数值，在 3.4.2 节中已经详细阐述过。我们可以通过比较访问函数图像与预测函数图像的拟合程度，来判断预测函数模拟的准确性；通过预测函数图像的后期走势，判断预测算法预测的准确性。

图 3.3 中我们模拟了写倾向、读倾向和读写访问倾向不明显的结点的实际访问图像，并计算出其对应的预测函数图像。实验设计的滑动窗口长度为 22。图 3.3 (a)显示的是写倾向结点的函数图像，其访问函数是逐渐上升的趋势，表示滑动访问窗口中比特“1”出现频繁，可以被认为是写倾向结点。我们计算出的预测函数与访问函数图像非常贴近，可见预测函数拟合准确。从第 23 个数据开始，是预测函数所预测的结点访问情况，由图中可知仍然是上升的趋势，也就是说，函数预测结点未来仍然是写倾向结点，这与我们给结点的设计是相符的，所以预测函数预测准确。同理，图 3.3 (b)是读倾向结点的访问函数和预测函数图像，预测函数无论是在拟合阶段还是预测阶段都达到了理想的效果。图 3.3 (c)是读写倾向不明显的结点，访问函数始终在“0”值上下徘徊；但仔细观察仍可发现，在第 9 到第 14 个数据之间访问函数上升，这阶段可以被认为写频繁，在第 14 到第 22 个数据之间访问函数下降，这阶段可以被认为是读倾向。预测函数的前 22 个数据都在“0”值左右，从第 23 个开始呈下降趋势，是根据访问函数后半部分的读倾向判断得出的，这也是合理的结果。

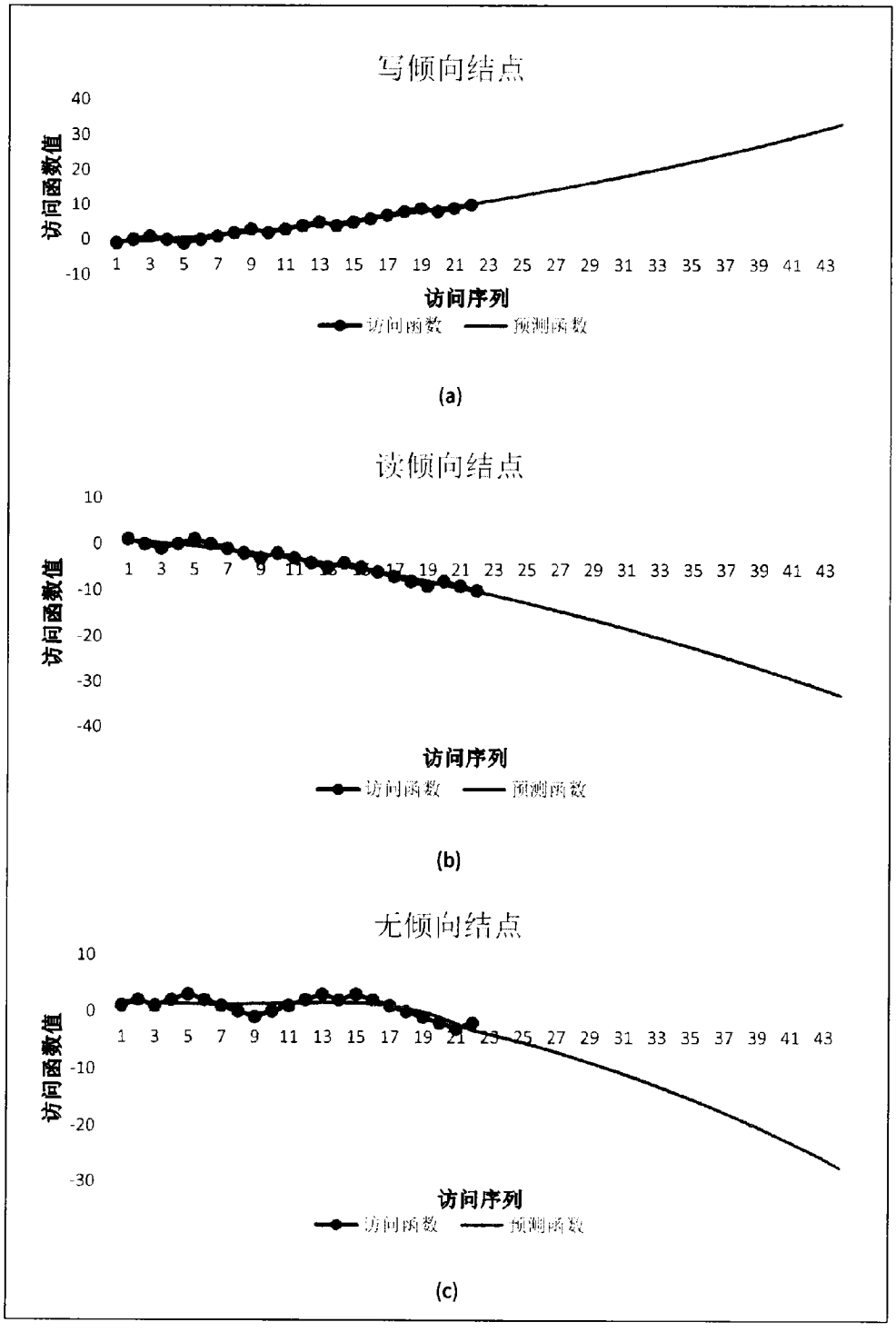


图 3.3 不同访问倾向结点的访问函数和预测函数

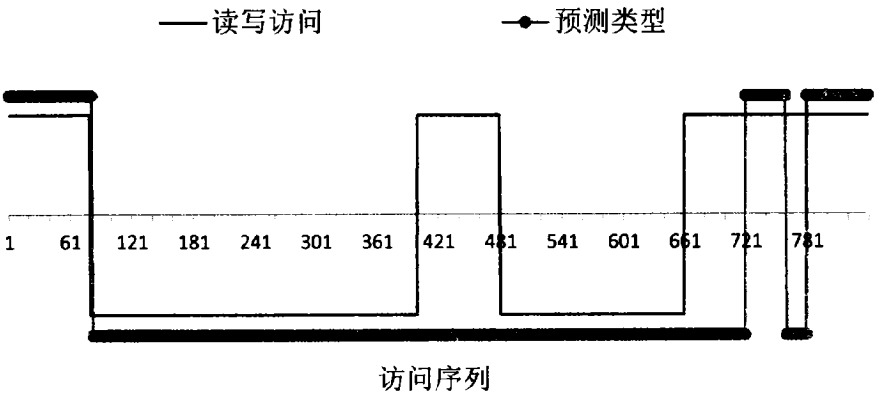


图 3.4 结点实时访问与类型预测

图 3.4 给出了一个结点的实时读写访问和实时预测类型的比较结果。对于读写访问，高于横轴表示该次是写访问，低于横轴表示该次是读访问。对于预测类型，高于横轴表示预测出该结点是写倾向类型，低于横轴则表示预测出是读倾向类型，其中预测函数所用到的数据是当前点之前的所有访问数据。如果两个图像拟合程度高，则表示预测函数准确性较高，反之则预测不准确。从图 3.4 中可以看出，预测类型在大部分时间都与原读写访问一致；读写访问在短时间内发生突变，并不会立刻影响预测算法对其整体读写倾向的预测结果，而长时间的改变则会影响预测结果。比如一直是读倾向的结点，突然接受到几次写访问请求，这并不能表示该结点就变成了写倾向结点，而如果此后长时间都接受写访问，那么结点的访问倾向就有可能发生改变，这与实际的访问情况是相符的。预测类型在 761 次访问附近发生了两次抖动，而实际访问却一直是读访问，看起来似乎表示预测函数不具有准确性，但是，CB+-tree 中的叶子结点并不总是进行类型预测，只有当结点的写比例 ratio 处于中间的模糊地带时才出发预测机制，在图 3.4 的状况下，第 761 次访问附近，结点的写比例一定是低于较小阈值θ_1的，结点会被直接判断为读倾向类型，而不会触发预测机制。

以上两种验证方法均可以证明，我们的预测机制是可以准确预测出结点的读写访问倾向的，这就为 CB+-tree 的结构调整的合理性提供了理论支持。

3.6.3 CB+-tree 性能分析

为了验证 CB+-tree 的性能优势，我们实现了传统 B+-tree 算法、溢出类型 B+-tree 算法，在执行同样的访问请求后，记录各索引对 PCM 的读写次数、执行时间等参数，并进行性能优劣分析。由于 Bp+-tree 索引算法的前提条件是知道所有

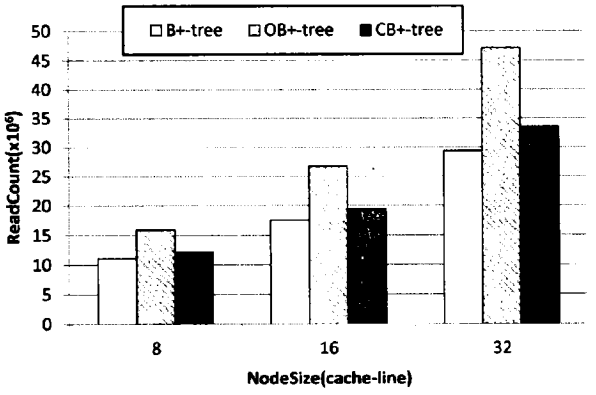
访问数据的范围和总访问记录数，实用性较差，且与我们的实验条件不相符，所以这里不做比较。实验中，各参数的设置如表 3.4 所示，其中溢出类型 B+-tree（OB+-tree）与 CB+-tree 的最大溢出链长度都设置为 6。

表 3.4 实验环境参数

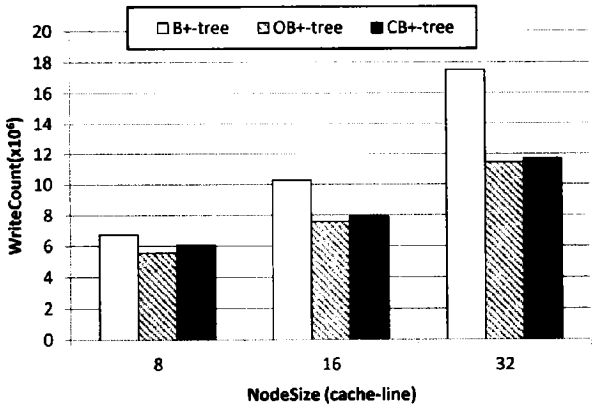
参数	值
L	16
ω	0.8
θ_1	0.3
θ_2	0.7
MAX_OVF_LEN	6

图 3.5 给出的是不同索引在不同结点大小的情况下对 PCM 造成的读次数、写次数、运行时间的比较图，实验中设置的 Cache 大小均为 1024KB。图 3.5 (a)是不同索引的读次数统计图，可以看出，在不同结点大小的情况下，传统 B+-tree 索引的读次数都是最少的，因为传统 B+-tree 不含有溢出结点。然而，从图 3.5 (b)中可以看出 B+-tree 索引对 PCM 造成的写次数都是最多的，这将会减少 PCM 的使用寿命并降低整体性能。OB+-tree 的读次数最多，因为其中每个叶子结点都含有较长的溢出链，会增加读操作次数，但溢出链可以有效减少对上层结点的写次数，所以 OB+-tree 的写次数最少。在另一方面，CB+-tree 的读次数与 OB+-tree 相比大大减少，只比传统 B+-tree 高一些，是因为访问倾向预测机制动态的把读频繁结点从溢出链中取出，减少了访问代价。注意 CB+-tree 的写次数与 OB+-tree 相比有少量增加，这是因为访问倾向预测机制造成的结构调整会带来额外写次数。但是，结构调整是不频繁的，所以带来的额外写代价也微乎其微，并没有大量降低整体性能。如图 3.5 (c)所示，CB+-tree 拥有最短的运行时间，OB+-tree 其次。CB+-tree 主要通过以小量写次数的增加来大量降低读次数，从而提升整体的运行时间。由图中可以看出，结点越大，CB+-tree 所提升的时间性能越多，在结点大小为 32cache-line 时，CB+-tree 比传统 B+-tree 节省了 22.85%的时间开销。

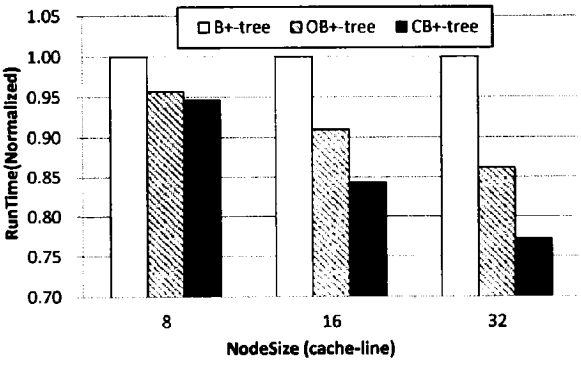
图 3.6 给出了不同 Cache 大小对于各索引性能的影响，其中每个索引结点大小都设置为 16 个 cache-line。从图中可以看出，Cache 越大，各索引对 PCM 造成的读写次数越低，但是同一 Cache 大小下，各索引之间的相对性能与图 3.5 给出的一致。CB+-tree 无论在何种情况下，都拥有较少的读次数和几乎与 OB+-tree 一样多的写次数，时间性能最好。传统 B+-tree 虽然读性能最好，但是较高的写次数仍然拖累了索引整体性能，有最长的运行时间。OB+-tree 在写次数上比 B+-tree 低很多，但是读次数太高，使得其性能与传统 B+-tree 比有所提升，但提升的幅度不大。



(a)



(b)



(c)

图 3.5 不同结点大小的 CB+-tree 与各索引算法比较

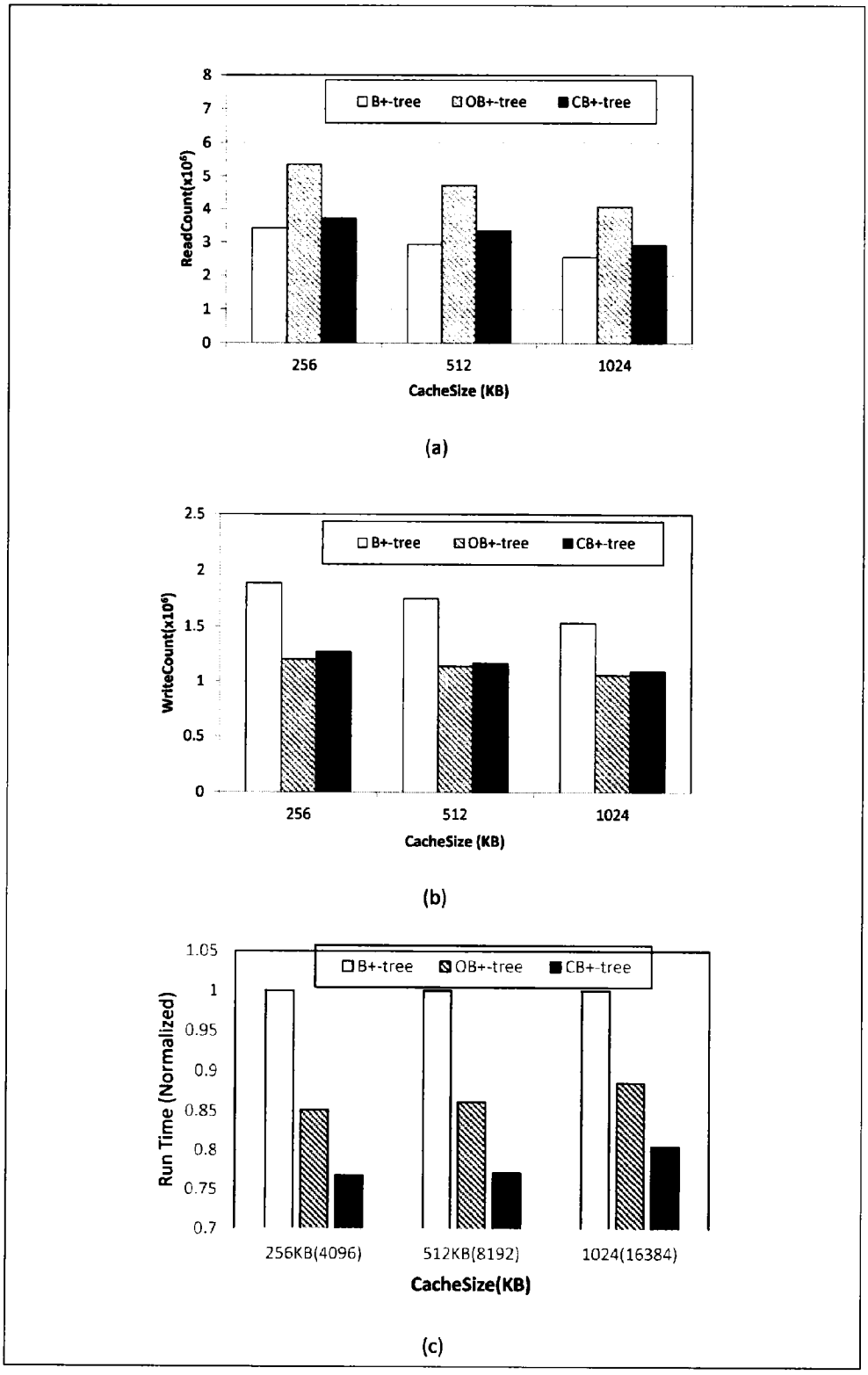


图 3.6 不同 Cache 大小的 CB+-tree 与各索引算法比较

表 3.5 给出了 CB+-tree 中预测机制触发次数与结构调整次数在不同结点大小条件下的统计数据。由于预测机制所需数据全部存储在 DRAM 上，所以对 PCM 的性能并无影响。由表中可以看出，随着结点大小成倍增大，预测机制的触发次数却是线性增长。当结点大小为 16 个 cache-line 时，索引结构调整次数最少。叶子结点的读写访问信息占索引结构的比例也随着结点的增大而降低，可见预测机制的空间代价非常小。实验中各算法对 PCM 的读写次数数量级为百万级甚至更高，而预测机制的触发次数数量级为万级，二者相差两个数量级，且预测次数是呈线性增长趋势，相比之下其对 DRAM 的读写代价可以忽略；预测算法主要采用最小二乘法线性拟合计算，时间代价也较低，所以预测机制的时间开销几乎可以忽略不计。而结点类型判断机制仅在特定条件下触发，而触发类型判断后又需满足一定的条件才会触发预测机制，所以结点类型相关信息的统计与计算的开销都非常小，与 PCM 上的读写代价相比几乎可以忽略不计。总的来说，CB+-tree 是以空间、时间低代价的结点读写访问倾向预测机制，换取写次数的大幅度降低，从而最终提升整体性能。

表 3.5 CB+-tree 预测机制相关统计

次数 / 结点大小（cache-line） / 参数	8	16	32
预测次数	10561	12472	15763
结构调整次数	6200	4480	4961
访问信息占比	2.14%	1.05%	0.53%

3.7 本章小结

在本章中，我们首先提出了一个基于 OB+-tree 的索引结构调整模型，该模型兼顾了传统 B+-tree 的优良读性能，又发挥了 OB+-tree 的低写代价的优势，并且在该模型中引入叶子结点读写访问性概念，保证了 CB+-tree 结构调整的准确性。基于结点的读写访问倾向性，我们提出了访问倾向性判断算法，其中涉及到结点访问类型的预测机制。最后，通过与传统 B+-tree 和 OB+-tree 的实验比较，测试了多种情况下的读写性能和预测代价。实验结果证明，CB+-tree 索引算法的性能优于对比算法，证明了以极小的写代价换取大量读代价降低，也可以提高 PCM 的性能，打破了以往只关注如何降低写代价来进行优化的思路。

第4章 基于混合主存的 B+-tree 索引迁移算法

4.1 引言

随着大数据时代的到来，数据管理对主存系统的要求越来越高，大容量、低能耗、高性能成为主存系统必须达到的要求。然而，传统 DRAM 的特点，让其无法构成大容量、低能耗的主存系统，新型存储器 PCM 却刚好可以弥补 DRAM 这一缺陷，构建以 PCM 和 DRAM 作为混合主存的系统架构成为当下业界研究的热点。

非易失存储器 PCM 的字节读写与高写代价特性，让传统的主存索引算法不能适用其中，所以，优化传统 B+-tree 算法的需求是适应于大数据时代的趋势。在目前已有的研究中，尚未有利用 DRAM 和 PCM 作为平级主存介质进行索引优化的方法，本章提出的 B+-tree 索引优化算法正是基于这一点。由第 3 章中提出的 CB+-tree 的实验结果可以证明，极少的结构调整次数可以带来较大的整体性能提升，表明数据库中对数据的读写访问具有集中性，相同读写访问趋势的数据通常比较集中，我们可以将不同访问趋势的数据存放在不同的介质上，让 DRAM 和 PCM 分别发挥各自的优势，以达到整体性能的提升。本章的主要内容有：

（1）基于数据读写访问集中性和结点读写类型判断机制（详见第 3.3 小节），结合 DRAM 和 PCM 各自的读写特点，提出了一种新的索引优化算法 XB+-tree，有效降低 PCM 的读写次数，提高 PCM 性能。

（2）通过对结点读写访问的记录，提出了一种 DRAM 上的结点数据管理算法，最大限度发挥 DRAM 的优势，优化 PCM 性能。

（3）对提出的索引优化算法进行实验测试评估。实验结果表明我们提出的索引优化算法可以大大提升 PCM 的读写性能。

本章的后续内容安排如下：在第 4.2 节中，详细介绍了 XB+-tree 索引算法；在第 4.3 节中介绍 DRAM 上结点数据的管理算法；在第 4.4 节中给出实验设计与结果分析。第 4.5 节总结本章工作。

4.2 XB+-tree 索引算法

本小节首先介绍 XB+-tree 算法设计的理论依据，然后描述 XB+-tree 索引的基本结构，最后讨论查询、插入、删除等操作如何进行。

4.2.1 XB+-tree 索引设计理念

在先前的讨论中，我们已经知道，溢出类型 B+-tree（OB+-tree）可以有效提高传统 B+-tree 的写性能，但其高昂的读代价降低了它的性能提升。但是，如果利用 DRAM 的高写性能，将少量频繁访问的结点存放到 DRAM 上，而剩下的大量低热度结点存放在 PCM 上，那么就可以大大提升 PCM 性能。

由于数据库的数据访问非常具有集中性，通常，相同访问特性的数据都存在同一结点中，这就导致了结点也具有读写访问倾向性。通过第 3.3 节的讨论，我们可以较准确的判断结点的访问倾向性，这就为实现 XB+-tree 索引优化提供了保证。

对 PCM 上数据进行写操作，需要先将数据读入缓存，换句话说，一次写操作至少对应一次读操作。所以，将写倾向结点迁移到 DRAM 上，以后每对该结点进行一次写操作，就可以同时减少对 PCM 的读写各一次。

4.2.2 XB+-tree 索引结构

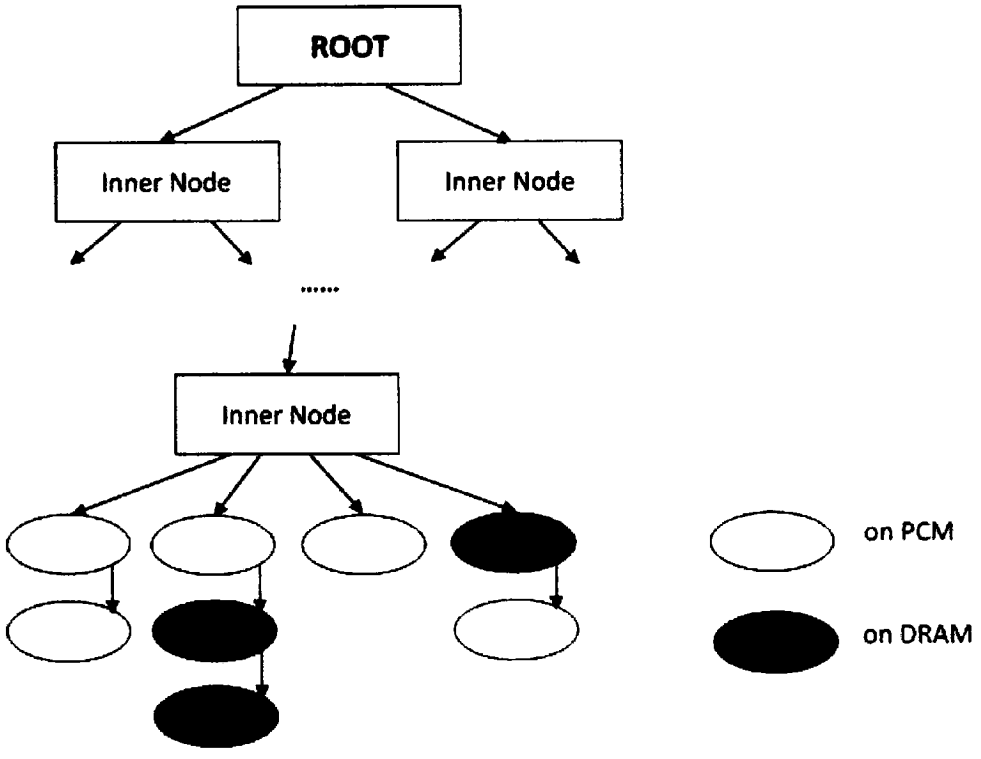


图 4.1 XB+-tree 索引结构

图 4.1 显示了 XB+-tree 索引结构。XB+-tree 索引结构是基于溢出类型 B+-tree（OB+-tree）索引。所有记录都存放在叶子结点中，每个叶子结点都可以有若干溢出结点。溢出结点结构可以有效减少叶子结点分裂/合并操作对上层结点造成的写操作。所有结点内部记录都有序，且都由若干 cache-line 组成，这与我们在第三章提出的 CB+-tree 索引结点结构一样；不同的是，在 *cacheline_h* 中，我们将 *is_ovf* 参数更改为 *on_PCM*，用来标记当前结点的位置，由 a 个 cache-line 组成的结点的辅助信息占比仍为 5/(16a)。XB+-tree 中所有内部结点都存放在 PCM 上，而叶子结点则要根据其读写倾向选择存储位置：写倾向结点存放在 DRAM 上；读倾向且访问频繁结点存放在 DRAM 上；其他类型叶子结点存放在 PCM 上。

当我们判断出某个在 PCM 上的叶子结点具有写倾向时，就要将其迁移到 DRAM 上；当 DRAM 满时，则选择合适的结点进行迁回 PCM 的操作。结点的迁移对整体 XB+-tree 结构并无影响。

下面给出 XB+-tree 的索引操作算法。其中调用 *adjust_leaf()*函数的作用是调整叶子结点位置，具体方法将在 4.3 节中给出。

算法 4.1.对 XB+-tree 的插入操作 *insert(ROOT, key, value)*。

输入：要插入记录<*key, value*>和 XB+-tree 的根结点 ROOT
输出：XB+-tree 的根结点 ROOT

```
1. find the first leaf node f_l;
2. if f_l has overflow chain then
3.     find target leaf node t_l in the overflow chain;
4.     insert <key, value> into t_l;
5. else
6.     insert <key, value> into f_l;
7. if NODE need to split then
8.     split NODE as an overflow chain;/*分裂出的新结点与原结点所
       在位置相同*/
9.     adjust the overflow chain;/*若溢出链长度达到上限，则分裂溢
       出链，成为若干独立叶子结点*/
10. end if
11. update the metadata for NODE;/*更新结点的读写访问信息，若执
       行过分裂操作，则也为新结点更新读写信息*/
12. adjust_leaf(ROOT, NODE);
13. return ROOT;
```

算法 4.1 给出插入操作的具体算法。我们首先找到需要插入的目标叶子结点 NODE。若插入记录后结点需要分裂，那么 NODE 分裂出的新结点需要和 NODE

在同一介质上，因为我们认为相近的数据具有相似的访问倾向性，在同一介质上创建新结点也可以让结点更契合存储介质的特性。最后，我们需要对 NODE 进行读写倾向性判断，让变成写倾向的 NODE 可以迁移到 DRAM 上，或已经在 DRAM 上又变为读倾向的 NODE 加入 LRR 链表中（LRR 链表将在第 4.3 节中作说明）。

算法 4.2.对 XB+-tree 的删除操作 *delete(ROOT, key)*。

输入：要删除记录 *key* 和 XB+-tree 的根结点 ROOT
输出：XB+-tree 的根结点 ROOT

```
1. delete key and its value from the target leaf node
2. if need to merge or distribute node then
/*For a node on PCM, prefer to find a neighbor on PCM*/
3.    if NODE is on PCM then
4.       if NODE has neighbor on PCM then
5.          find a neighbor on PCM;
6.       else
7.          find a neighbor on DRAM;
8.       end if
/*For a node on DRAM, prefer to find a neighbor on DRAM*/
9.    else
10.      if NODE has neighbor on DRAM then
11.         find a neighbor on DRAM;
12.      else
13.         find a neighbor on PCM;
14.      end if
15.   end if
16. end if
17. update the metadata for NODE;/*更新结点的读写访问信息，若先
    前修改过邻居结点，则也为邻居结点更新读写信息*/
18. adjust_leaf(ROOT, NODE);
19. return ROOT;
```

算法 4.3.对 XB+-tree 的查询操作 *search(ROOT, key)*。

输入：要查询记录 *key* 和 XB+-tree 的根结点 ROOT
输出：XB+-tree 的根结点 *ROOT*，查询结果 *value*

```
1. find the first leaf node;
2. find the target leaf node in the overflow chain;
3. search within the target leaf node;
4. adjust_leaf(ROOT, NODE)
5. return ROOT;
```

算法 4.2 给出删除操作的具体算法。我们首先找到要删除记录所在的叶子结点并删除该记录。如果触发了结点合并/借元素操作，那么对于溢出链中的结点，要以同一溢出链中的前后结点作为邻居结点，且优先选择在同一存储介质上结点，这样做仍然是尽可能让相同访问倾向性的结点放在同一结点中。最后需要对 NODE 进行读写倾向性判断并调整结构。

算法 4.3 给出了查询操作的具体算法，这与 OB+-tree 的查询算法类似，不同之处在于当找到目标叶子结点并查找到对应的 *value* 后，需要对目标叶子结点进行读写倾向性判断和叶子结点结构调整。

4.3 DRAM 结点管理算法

DRAM 具有比 PCM 高的写性能，我们将高写访问度的结点迁移到 DRAM 上可以减少索引算法对 PCM 的磨损，也能减少执行时间。但是，DRAM 容量有限，我们只能存放少量写访问度最高的结点，且随着迁移结点的增多，DRAM 容量达到上限，再进行结点向 DRAM 迁移时，需要进行结点回迁的操作。本小节主要讨论如何来管理 DRAM 上的结点，以让结点选择最合适的存储介质。

DRAM 上结点的管理主要利用了两个链表。第一个是 LRU 链表，用来管理 DRAM 上所有结点的 LRU 链表，当该结点被访问到（无论是读或写访问），都将其前移到链表头部。第二个是 LRR(Least Recently Read)链表，用来管理 DRAM 上的读倾向结点。一个叶子结点因为写倾向而被前移到 DRAM 上后，随着访问的进行，可能又变成读倾向结点，此时我们将该结点加入 LRR 链表，当结点有读访问请求时，将其调整到链表头部；当该结点变成写倾向时，就从 LRR 链表中移除。

当某个叶子结点变成写倾向结点时，需要被迁移到 DRAM 上。若此时 DRAM 上仍有空间，则直接迁入，并将结点插入至 LRU 链表头部即可。若此时 DRAM 空间已满，则需要选择一个结点迁出 DRAM 回到 PCM，为迁入结点腾出空间：若 LRR 链表不为空，表示 DRAM 上有读倾向结点，那么选择 LRR 链表尾部结点迁回 PCM；若 LRR 链表为空，则从 LRU 链表尾部选择结点迁回，因为该结点在迁入 DRAM 后很久没有被访问了，不论它是读倾向还是写倾向的结点，在迁回 PCM 后都很少会被访问到，不会造成较多的读写次数，所以其读写特性对 PCM 性能的影响几乎可以忽略。腾出空间后再将要迁入的结点写入 DRAM，并将其加入到 LRU 链表头部。

算法 4.4 给出了结点迁入 DRAM 的操作方法。

算法 4.4.结点迁入函数 *act_modify(ROOT, NODE)*。

输入：XB+-tree 根结点 ROOT 和要迁移结点 NODE
输出：XB+-tree 的根结点 ROOT

```
1.  if DRAM is full then
2.      if LRR is not empty then
3.          move the last node of LRR to PCM;
4.      else
5.          move the last node of LRU to PCM;
6.      end if
7.  end if
8.  write NODE on DRAM;
9.  adjust LRU;
10. return ROOT;
```

算法 4.5 给出了索引叶子结点结构维护算法。

算法 4.5.叶子结点结构维护算法 *adjust_leaf(ROOT, NODE)*。

输入：XB+-tree 根结点 ROOT 和被维护结点 NODE
输出：XB+-tree 的根结点 ROOT

```
1.  if NODE is on PCM and has write tendency then
2.      act_modify(ROOT, NODE);
3.      on_PCM of the parent node ←false;
4.  end if
5.  if NODE is on DRAM then
6.      adjust LRU;
7.      if NODE has read tendency then
8.          adjust LRR;
9.      end if
10. end if
11. return ROOT;
```

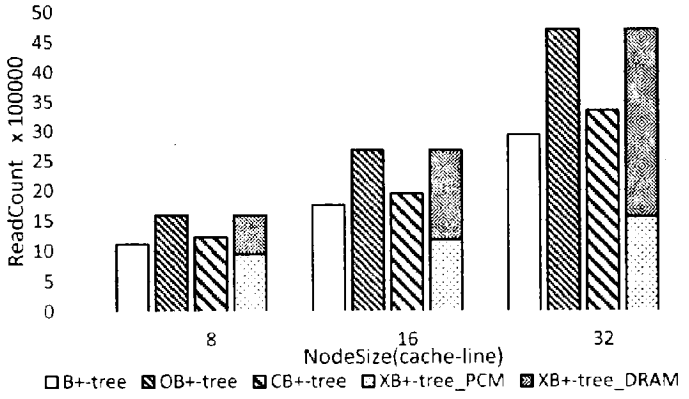
4.4 实验设计与结果分析

我们同样利用 3.5 小节中的实验环境来验证 XB+-tree 对 PCM 性能的提升。我们的对比实验有传统 B+-tree、溢出类型 B+-tree（OB+-tree）和第三章中提出的 CB+-tree，分别统计了各算法对 PCM 造成的读次数、写次数和运行时间，特别的，对于本章提出的 XB+-tree，我们还统计了算法对 DRAM 造成的读写次数。

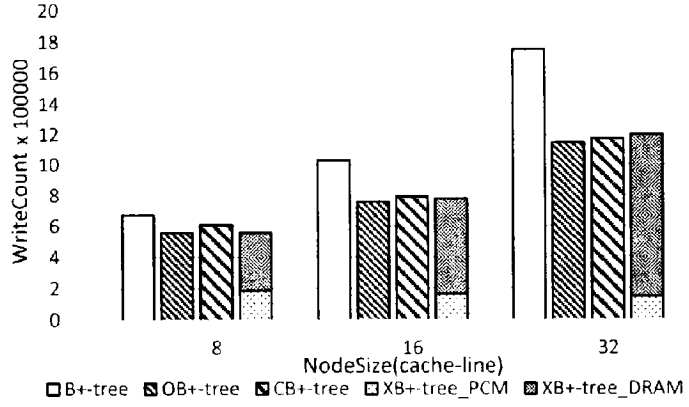
图 4.2 给出了在不同结点大小的情况下，各索引对 PCM、XB+-tree 对 DRAM 造成的读写次数以及各索引运行时间的比较图表，其中设置的 Cache 大小均为

1024KB。图 4.2 (a)统计了 PCM 的读次数，可以看出，OB+-tree 具有最多的读次数，这是因为其每个叶子结点都拥有较长的溢出链，而同样拥有溢出链的 XB+-tree 却有最少的读次数，甚至优于不含溢出链的传统 B+-tree，这是因为我们将频繁写的结点迁移至 DRAM，而写操作之前需要进行一次读操作，所以写频繁结点的迁出也有利于读次数的减少。XB+-tree 对 DRAM 造成的读次数随着结点大小的增加而增加，在结点大小为 16 个 cache-line 和 32 个 cache-line 时，大大超过了索引对 PCM 造成的读次数，这是由于迁移至 DRAM 的结点具有很高的访问频度，对迁移结点的访问越集中，XB+-tree 可以减少的对 PCM 的读次数就越多。图 4.2 (b)统计了 PCM 写次数，传统 B+-tree 由于不含有溢出链结构而具有最多的写次数，OB+-tree 和 CB+-tree 的写次数相当，且都少于传统 B+-tree，是因为溢出链结构减少了结构变化对上层结点的写访问。而同样具有溢出链的 XB+-tree 与这两种索引相比写次数大大降低，这也要归功于结点迁移机制，减少的写访问全部转移到了 DRAM 上，XB+-tree 对 DRAM 造成的写次数较多，但是 DRAM 的写代价很低，与其读代价相当，同样次数的写操作造成的延迟比在 PCM 上小很多，这就可以大大提升整体的运行时间。XB+-tree 比 OB+-tree 减少的读写次数刚好是 XB+-tree 对 DARM 造成的读写次数，而 DRAM 的高读写次数也可以体现出数据库对索引结点的读写访问具有很大的倾斜性。图 4.2 (c)是各索引的运行时间，皆以 B+-tree 时间标准化。如图所示，XB+-tree 拥有最短的运行时间，CB+-tree 其次，传统 B+-tree 时间最长。XB+-tree 对 PCM 的读写均有较大程度的降低，虽然降低的读写次数转移到了 DRAM 上，但是 DRAM 的高读写性能让整体的时间大大缩短。由图中可以看出，结点越大，XB+-tree 所提升的时间性能越多，当结点大小为 32cache-line 时，XB+-tree 比传统 B+-tree 提升了 51.55%的时间性能，最差情况下（结点大小为 8cache-line）也仍然节省了 40.02%的时间。

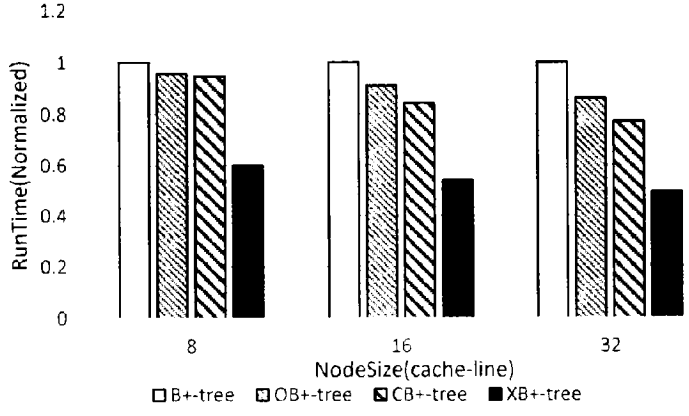
图 4.3 给出了不同 Cache 大小对于各索引性能的影响，其中结点大小均为 16 个 cache-line。从图中可以看出，Cache 越大，各索引对 PCM 造成的读写次数越低，但是各索引间的相对性能与图 4.2 中所给的一致。我们提出的 XB+-tree 在所有情况下都拥有最少的 PCM 读写次数，其 PCM 读写次数与 DRAM 读写次数之和刚好为 OB+-tree 的读写次数。时间性能上，不同 Cache 大小对时间性能的提升几乎没有影响，如图 4.2 (c)所示，从左到右时间性能的提升分别为 44.79%，46.02%和 45.91%。



(a)



(b)



(c)

图 4.2 不同结点大小的 XB+-tree 与各索引算法比较

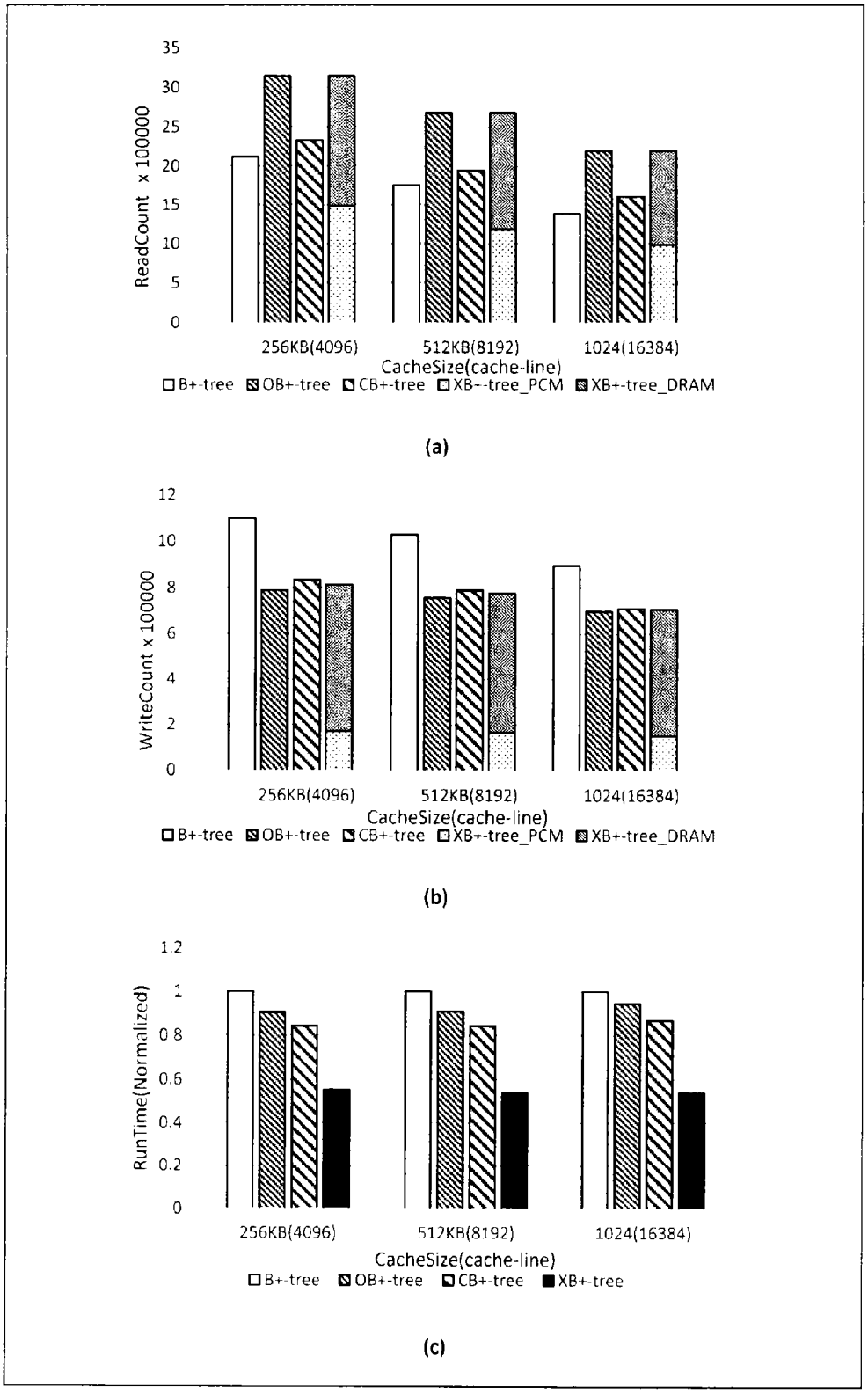


图 4.3 不同 Cache 大小的 XB+-tree 与各索引算法比较

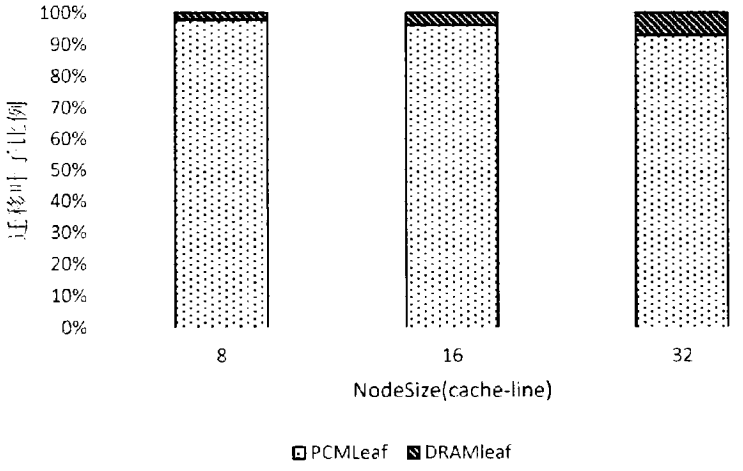


图 4.4 不同结点大小的 XB+-tree 迁移叶子数目占比

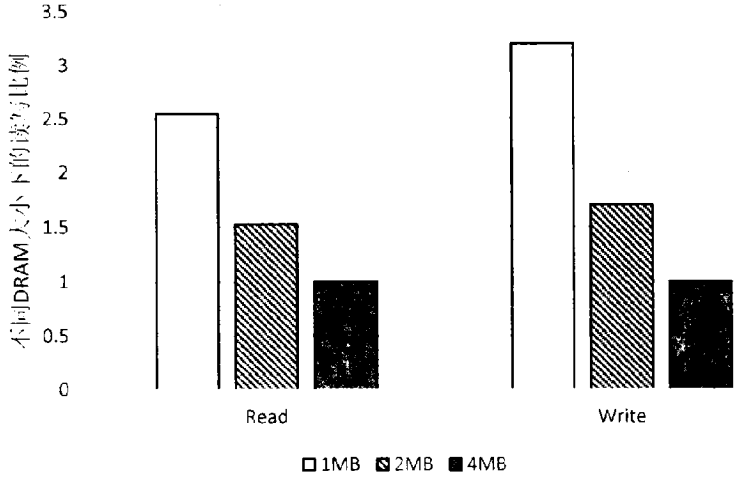


图 4.5 不同 DRAM 大小的 XB+-tree 读写次数比较

图 4.4 给出了 XB+-tree 在不同结点大小情况下，迁移至 DRAM 的叶子结点占所有叶子结点数目的比例，实验是在 DRAM 空间无限大的条件下进行的。在结点大小为 8 个 cache-line、16 个 cache-line 和 32 个 cache-line 时，迁移叶子结点占比分别为 2.17%、3.0%和 7.02%，占用 DRAM 大小分别为 2.34MB、3.97MB 和 6.89MB，可见大部分叶子结点都是读倾向且存放在 PCM 上的，迁移机制只需要很小的 DRAM 空间。而当 DRAM 大小有限时，我们统计了不同 DRAM 大小下的 XB+-tree 的读写次数，XB+-tree 结点大小均为 16 个 cache-line，如图 4.5 所示，以 DRAM 为 4MB 的情况标准化（由图 4.4 可知，该条件下的 XB+-tree 所需要 DRAM 的最大空间为 3.97MB）。当 DRAM 只有所需空间的 1/4 大小（1MB）

时，XB+-tree 的读写次数分别为标准情况下的 2.55 倍和 3.2 倍，当 DRAM 为所需空间的 1/2（2MB）时，读写次数分别为标准情况下的 1.53 倍和 1.7 倍。这是由于 DRAM 空间不足够大时，随着写倾向结点不断迁移到 DRAM，DRAM 空间满，则需要迁回部分写倾向结点，而这些迁回的结点又给 PCM 带来了较多的读写次数。

总的来说，XB+-tree 可以大大减少索引对 PCM 的读写次数，延长了 PCM 寿命；对于 DRAM 和 PCM 的混合主存来说，也可以较大程度提升整体的时间性能。

4.5 本章小结

为了提高基于 DRAM 与 PCM 混合主存架构下的 B+-tree 索引性能，本章提出了一种 OB+-tree 的结点迁移策略 XB+-tree。XB+-tree 采用了叶子结点读写倾向判断机制和结点迁移机制，将不同访问倾向的结点迁移至不同的存储介质中，从而发挥了介质各自的优势，以达到整体性能的提升，并使用了两个链表的方式来管理 DRAM 上的结点数据。通过与传统 B+-tree、OB+-tree 和 CB+-tree 的比较实验，证明 XB+-tree 可以有效减少 PCM 读写次数，其性能优于对比算法，以极小的空间代价换取了较大的时间性能提升。

第5章 总结与展望

5.1 本文工作总结

随着科技的高速发展和大数据时代的到来，对数据的存储需求和对响应时间的要求都在不断提高，而计算机的发展速度已不能满足数据增长的高效能存储需求。我们不仅要从软件层面上解决存储问题，更需要建立一种更快、更高效的主存系统。传统 DRAM 已经无法满足低能耗和大容量的主存系统要求，而新型存储器 PCM 则是建立新主存架构的优选存储介质。但是，PCM 的高写能耗和写延迟以及写寿命有限等特点，让我们无法随心所欲的将现有主存索引算法直接使用，研究适用于 PCM 的索引算法成为当下 PCM 研究领域的一个关注热点。本文面向 PCM 为主要介质的主存系统，研究如何优化现有的 B+-tree 主存索引算法，让其在新型主存架构下仍然保持较高的性能。目前，对 B+-tree 索引的优化方法主要有自结构调整和利用 DRAM 为辅助存储两种方式，本文结合了数据库对索引结点访问的读写倾向性，分别探讨了 B+-tree 索引自结构调整和利用 DRAM 辅助存储的结点迁移两种方式下的索引优化策略。

本文的主要贡献可总结为如下两方面：

（1）提出了溢出类型 B+-tree 索引的读访问优化算法。我们首先提出了索引结点的读写倾向性概念，并引入了结点读写倾向性判断策略。基于结点的不同访问倾向，设计了动态调整溢出类型 B+-tree 索引溢出链的方法，以极少的写代价增加换取大量的读次数减少，以让整体性能提升。在实验部分，首先设计模拟实验验证了结点读写倾向预测机制的正确性，然后利用截取到的真实的数据库访问数据，对传统 B+-tree、溢出类型 B+-tree 和提出的算法进行对比实验，证明我们提出的算法能有效的减少溢出类型 B+-tree 的读次数，验证了算法的有效性。

（2）提出了利用 DRAM 作为辅助存储的溢出类型 B+-tree 索引的结点迁移算法。利用 DRAM 的高写性能来弥补 PCM 在写操作上的短板，结合索引结点具有读写倾向性的特点，将少量具有写倾向的结点迁移至 DRAM 上，而大容量的 PCM 存储大部分索引结点。同时，为小容量的 DRAM 提出了索引结点管理算法，尽可能让读倾向结点存储在 PCM 而有限的 DRAM 空间存储写倾向结点。利用截取到的真实的数据库访问数据，进行与传统 B+-tree、溢出类型 B+-tree 等索引的对比实验，证明了我们提出的算法对减少 PCM 的读写次数均非常有效，可以大大提高 PCM 的性能，对混合主存的整体性能也起到优化作用。

5.2 下一步工作展望

本文利用 DRAM 和 PCM 的混合主存架构，实现了溢出类型 B+-tree 索引的读访问优化算法与迁移机制下的读写性能优化算法，但是对于 B+-tree 索引的相关研究还有许多不足之处，对混合主存的利用也非常局限。在以后的工作中，主要打算从以下几个方面来开展研究工作：

一方面，B+-tree 索引的种类繁多，基于闪存的对传统 B+-tree 索引的优化算法已经有丰硕的研究成果。本文主要是以溢出类型 B+-tree 索引为基础进行优化算法的研究，而针对其他类型 B+-tree 的优化研究是下一步工作的主要方向。

另一方面，数据库索引也不仅仅局限于 B+-tree 索引，还有许多类型比如 Hash 索引等。其他类型索引有不同于 B+-tree 索引的优势，下一步工作可以基于其他数据结构来考虑索引的优化，或者利用其他数据结构与 B+-tree 相结合，而不单单从 B+-tree 角度出发。

再者，DRAM 与 PCM 的混合主存架构可以充分利用各自存储器的优点，但是，DRAM 不是非易失型存储器，在利用 DRAM 作为辅助存储的同时，没有考虑到系统掉电 DRAM 上数据丢失的问题。下一步的工作可以考虑如何设计索引，让索引不依靠 DRAM 上的数据也可以顺利执行各种操作，或者如何进行系统恢复，保证索引的正常使用。

此外，本文提出的索引结点读写倾向性判断机制，对索引数据的访问集中性具有很大的依赖性。如果一些特定的数据访问是较分散的，读写集中性较弱，那么读写性判断结果则会不够准确，索引结构的优化效果则会大大降低。下一步工作可以研究如何准确的判断索引结点的读写倾向性。另外，抛开结点的读写倾向，如何优化 B+-tree 索引，也是下一步的工作目标。

参考文献

[1] 吴章玲, 金培权, 岳丽华,等. 基于PCM的大数据存储与管理研究综述[J]. 计算机研究与发展, 2015, 52(2):343-361.

[2] 谢佳壮. 面向数据库集群的节能查询技术研究[D]. 中国科学技术大学, 2015.

[3] 刘金垒, 李琼. 新型非易失相变存储器PCM应用研究[C]// 全国信息存储技术大会. 2011.

[4] 郑文静, 李明强, 舒继武. Flash存储技术[J]. 计算机研究与发展, 2010, 47(4):716-726.

[5] 杨濮源. 基于多介质设备的混合存储系统关键技术研究[D]. 中国科学技术大学, 2014.

[6] Chen S, Gibbons P B, Nath S. Rethinking Database Algorithms for Phase Change Memory[C]// CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings. 2011:21-31.

[7] Ni J, Hu W, Li G, et al. B^p-tree: A Predictive B^+-tree for Reducing Writes on Phase Change Memory[J]. IEEE Transactions on Knowledge & Data Engineering, 2014, 26(10):1-1.

[8] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 2010, 33(4):351-385.

[9] Arge L. The buffer tree: A new technique for optimal I/O-algorithms[J]. University of Aarhus, 1996, 955:334-345.

[10] Graefe G. Write-Optimized B-Trees[C]// Thirtieth International Conference on Very Large Data Bases-volume. 2004:672-683.

[11] Rao J, Ross K A. Making B+- trees cache conscious in main memory[J]. Acm Sigmod Record, 2000, 29(2):475-486.

[12] Chen S, Gibbons P B, Mowry T C, et al. Fractal prefetching B + -Trees: optimizing both cache and disk performance[C]// ACM SIGMOD International Conference on Management of Data. ACM, 2002:157--168.

[13] Wu C H, Chang L P, Kuo T W. An Efficient B-Tree Layer for Flash-Memory Storage Systems[M]// Real-Time and Embedded Computing Systems and Applications. Springer Berlin Heidelberg, 2004:17--24.

[14] Li Y, He B, Yang R J, et al. Tree indexing on solid state drives[J]. Proceedings of Vldb Endowment, 2010, 3(12):1332-1341.

[15] Roh H, Kim W C, Kim S, et al. A B-Tree index extension to enhance response time and the life cycle of flash memory[J]. Information Sciences, 2009, 179(18):3136-3161.

[16] Chi P, Lee W C, Xie Y. Making B + -tree efficient in PCM-based main memory[C]// International Symposium on Low Power Electronics and Design. ACM, 2014:69-74.

[17] Viglas S D. Adapting the b + -tree for asymmetric i/o[C]// East European Conference on Advances in Databases and Information Systems. Springer-Verlag, 2012:399-412.

[18] Burr G W, Kurdi B N, Scott J C, et al. Overview of candidate device technologies for storage-class memory[J]. Ibm Journal of Research & Development, 2008, 52(4):449-464.

[19] 张鸿斌, 范捷, 舒继武,等. 基于相变存储器的存储系统与技术综述[J]. 计算机研究与发展, 2014, 51(8):1647-1662.

[20] Qureshi M, Gurumurthi S, Rajendran B. Phase Change Memory: From Devices to Systems[M]. Morgan & Claypool, 2011.

[21] Qureshi M K, Karidis J, Franceschini M, et al. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling[C]// Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2009:14-23.

[22] Qureshi M K, Srinivasan V, Rivers J A. Scalable high performance main memory system using phase-change memory technology[J]. Acm Sigarch Computer Architecture News, 2009, 37(3):24-33.

[23] Zhou P, Zhao B, Yang J, et al. A durable and energy efficient main memory using phase change memory technology[J]. Acm Sigarch Computer Architecture News, 2009, 37(3):14-23.

[24] 张学琴. 嵌入式数据库B+_树索引机制研究及其改进[J]. 计算机与现代化, 2009(12):68-71.

[25] Choi G S, On B W, Lee I. PB+-Tree: PCM-aware B+-Tree[J]. IEEE Transactions on Knowledge & Data Engineering, 2015, 27(9):2466-2479.

[26] Jin P, Yang P, Yue L. Optimizing B+-tree for hybrid storage systems[J]. Distributed & Parallel Databases, 2015, 33(3):449-475.

[27] Jin R, Cho H J, Lee S W, et al. Lazy-split B+-tree: a novel B+-tree index scheme for flash-based database systems[J]. Design Automation for Embedded Systems, 2013, 17(1):167-191.

[28] Athanassoulis, Manos, Ailamaki, Anastasia. BF-tree: approximate tree indexing[J]. Proceedings of the VLDB Endowment, 2014, 7(14):1881-1892.

[29] Yang J, Wei Q, Chen C, et al. NV-Tree: reducing consistency cost for NVM-based single level systems[C]// 13th USENIX Conference on File and Storage Technologies (FAST '15). 2015.

[30] Wang P, Sun G, Jiang S, et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD[C]//Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014: 16.

[31] Chen S, Jin Q. Persistent B + -trees in non-volatile main memory[J]. Proceedings of the Vldb Endowment, 2015, 8(7):786-797.

[32] Email this article, Add to my bookshelf, Add to citation manager, et al. A Survey of Phase Change Memory Systems[J]. Journal of Computer Science and Technology, 2015, 30(1):121-144.

[33] Viglas S D. Data Management in Non-Volatile Memory[C]// Acm Sigmod International Conference on Management of Data. ACM, 2015:1707-1711.

[34] Wang Q, Li J R, Wang D H. Improving the Performance and Energy Efficiency of Phase Change Memory Systems[J]. 计算机科学技术学报:英文版, 2015, 30(1):110-120.

[35] Kuan Y H, Chang Y H, Huang P C, et al. Space-efficient multiversion index scheme for PCM-based embedded database systems[C]// Design Automation Conference. IEEE, 2014:1-6.

[36] Hankins R A, Patel J M. Effect of Node Size on the Performance of Cache-Conscious Indices[J]. Acm Sigmetrics Performance Evaluation Review, 2002, 31(1):283-294.

[37] Son M, Lee S, Kim K, et al. A small non-volatile write buffer to reduce storage writes in smartphones[C]// Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015. IEEE, 2015:713-718.

[38] Moraru I, Andersen D G, Kaminsky M, et al. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory[J]. Research.microsoft.com, 2013, 1:15-22 vol.1.

[39] Zhang P, Zhou C, Wang P, et al. E-Tree: An Efficient Indexing Structure for Ensemble Models on Data Streams[J]. IEEE Transactions on Knowledge & Data Engineering, 2015, 27(2):461-474.

[40] Arulraj J, Pavlo A, Dulloor S R. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems[C]// ACM SIGMOD International Conference on Management of Data. ACM, 2015:707-722.

[41] Viglas S D. Write-limited sorts and joins for persistent memory [J]. Proceedings of the Vldb Endowment, 2014, 7(5):413-424.

[42] Levandoski J J, Lomet D B, Sengupta S. The Bw-tree: A B-tree for new hardware platforms[C]//Data Engineering (ICDE), 2013 IEEE 29th International Conference on. IEEE, 2013: 302-313.

[43] Li X, Lu K, Zhou X. SIM-PCM: A PCM Simulator Based on Simics[C]// Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on. IEEE, 2012:1236-1239.

[44] Dong X, Jouppi N P, Xie Y. PCRAMsim: System-level performance, energy, and area modeling for Phase-Change RAM[C]// Ieee/acm International Conference on Computer-Aided Design. IEEE, 2009:269-275.

[45] Gao S, Xu J, He B, et al. PCMLogging: Reducing Transaction Logging Overhead with PCM[C]// Proceedings of the 20th ACM international conference on Information and knowledge management. ACM, 2011:2401-2404.

[46] Cho S, Lee H. Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance[C]//Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on. IEEE, 2009: 347-357.

[47] Ding S, Song Z, Chen H. A novel read-while-write (RWW) algorithm for phase change memory[C]// Non-Volatile Memory Technology Symposium (NVMTS), 2011 11th Annual. IEEE, 2011:1 - 4.

[48] Xi Zhang, Qian Hu, Dongsheng Wang, et al. A Read-Write Aware Replacement Policy for Phase Change Memory[C]// Advanced Parallel Processing Technologies - International Symposium, Appt 2011, Shanghai, China, September 26-27, 2011. Proceedings. 2011:31-45.

[49] Lencer D, Salinga M, Wuttig M. Design Rules for Phase-Change Materials in Data Storage Applications[J]. Advanced Materials, 2011, 23(18):2030-58.

[50] Wang J, Dong X, Sun G, et al. Energy-efficient multi-level cell phase-change memory system with data encoding[C]// IEEE International Conference on Computer Design. IEEE Computer Society, 2011:175-182.

[51] Jiang L, Zhao B, Zhang Y, et al. Improving Write Operations in MLC Phase Change Memory[C]// Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture. IEEE Computer Society, 2012:1-10.

[52] Liu D, Wang T, Wang Y, et al. PCM-FTL: A Write-Activity-Aware NAND Flash Memory Management Scheme for PCM-Based Embedded Systems[C]// Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd. IEEE, 2011:357-366.

[53] Papandreou N, Pozidis H, Pantazi A, et al. Programming algorithms for multilevel phase-change memory[C]// IEEE International Symposium on Circuits & Systems. IEEE, 2011:329-332.

[54] Yue J, Zhu Y. Making Write Less Blocking for Read Accesses in Phase Change Memory[C]// IEEE International Symposium on Modeling. IEEE, 2012:269-277.

[55] Jeyasingh R, Liang J, Caldwell M A, et al. Phase Change Memory: Scaling and applications[C]// Proceedings of the Custom Integrated Circuits Conference. 2012:1-7.

[56] Lee B C, Ipek E, Mutlu O, et al. Architecting phase change memory as a scalable dram alternative[J]. Isca '09 Proceedings of Annual International Symposium on Computer Architecture, 2009, 37(3):2-13.

[57] Li J, Lam C. Phase change memory[J]. Sciece China Information Sciences, 2011, 54(54):1061-1072.

[58] Lee B C, Zhou P, Yang J, et al. Phase-Change Technology and the Future of Main Memory[J]. IEEE Micro, 2010, 30(1):143-143.

[59] Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory[C]// Acm Symposium on Operating Systems Principles. ACM, 2010:133-146.

[60] Dhiman G, Ayoub R, Rosing T. PDRAM: A hybrid PRAM and DRAM main memory system[C]// Design Automation Conference. ACM, 2009:664-669.

[61] Vamsikrishna M V, Su Z, Tan K L. A Write Efficient PCM-Aware Sort[J]. Lecture Notes in Computer Science, 2012, 7446:86-100.

致 谢

值此论文完成之际，短暂的三年研究生生涯即将结束，也意味着我的学生时代将画上句号。感谢科大为我的学习和成长提供了优良的资源和美好的环境，让我的学习状态从本科的接受型转变为现在的研究型，为我学习能力的提升提供了充分的锻炼机会。我非常荣幸能够进入科大这所美丽的学校，度过宝贵的三年时光，成长为即将步入工作岗位的新人。科大时光的每个点滴都非常值得回味，是我一生中难以忘怀的经历。

由衷感谢我的导师岳丽华教授！她知识渊博，对工作认真负责一丝不苟，让我非常的敬佩。她一直鼓励我们多提问多讨论，并经常以其他研究领域的内容来拓展我们的思路；在我学习上遇到困难时，她始终耐心的引导我解决问题，为我指引方向；在我不知道如何选择研究方向时，她为我详细介绍实验室的研究状况，并且始终尊重我的选择；在我生病时，她对我无微不至的关怀让我倍感温馨。再次对我的导师岳老师表示深深的谢意！祝她及她的家人幸福健康！

由衷感谢实验室的金培权副教授！在科研的各个阶段，金老师都给予了非常耐心的指导和帮助。即使身处国外，交流不便，金老师仍然会百忙之中抽出时间为我解决疑惑。无论是研究过程中的方法探讨，还是最后论文形成的修改投稿，金老师都耐心提供了诸多帮助。在此对金老师表示由衷的感谢！祝他和他的家人幸福健康！

同时，我也要感谢万寿红副教授。万老师和蔼可亲，在工作和生活上都给予了我很多帮助。万老师在科研上认真专注，是我学习和科研的榜样。在此对万老师表示由衷的感谢！祝她和她的家人幸福健康！

感谢知识和数据工程实验室的所有同学们，尤其是同一个课题组的同学，和你们并肩学习工作的时光终身难忘。感谢师兄杨程程、郝行军、师姐吴章玲和已经毕业的陈凯萌、谢佳壮等师兄在学术上给予我的帮助和指导；感谢同学张德志、王文强、王晓亮等，给予我的照顾；感谢实验室的同学：张晓翔、都江、许大洲、罗文艺、田源、唐丽丽。和你们在一起的日子里，我们是最有爱的集体，让我能在一个轻松愉快的氛围中度过研究生生涯。衷心祝福大家都有个美好的前程！

感谢我的父母！感谢他们给予了我生命，教导我做人的道理，培育我成长，为我创造了在科大学习深造的条件。感谢他们用爱关怀我的成长，在我脆弱和迷茫的时候始终做我坚强的后盾，让我奋力拼搏！

最后，谨向百忙之中抽出宝贵时间评审论文的老师们致以最诚挚地感谢！

在读期间发表的学术论文与取得的其他研究成果

已发表论文：

[1] Lu Li, Peiquan Jin, Chengcheng Yang, Zhangling Wu, Lihua Yue: Optimizing B+-Tree for PCM-Based Hybrid Memory. EDBT 2016: 662-663.

待投稿论文：

[1] Lu Li, Peiquan Jin, Lihua Yue, Chengcheng Yang: B+-tree Migration Algorithm for PCM-Based Hybrid Memory

参与的科研项目：

[1] 国家自然科学基金项目，“基于相变存储器的数据管理关键技术研究”，2015-2018