
HF-Tree: 一种闪存数据库的高更新性能索引结构

周大 梁智超 孟小峰

(中国人民大学信息学院 北京 100872)

(cadizhou@gmail.com)

HF-Tree: An Update-Efficient Index for Flash Memory

Zhou Da, Liang Zhichao, Meng Xiaofeng

(Information school, Renmin University of China, Beijing 100872)

Abstract With the recent development of electronic technologies, flash memory emerges as new data storage media with high access speed and no mechanical latency. Flash memory drives have been envisioned to be widely used in laptops, desktops, and data servers in place of hard disks in the years to come. However, due to the expensive write cost of flash memory, traditional disk-based indexes have a poor update performance when directly applied to flash drives. In this paper, we propose a novel index called *HF tree* to improve the update performance, which integrates B^F -tree with *Tri-hash*. B^F -tree is adapted from the traditional B^+ -tree, yet it avoids excessive updates of B^+ -tree by adopting a block-based storage model and a lazy split and coalesce algorithm. *Tri-hash* uses three cascaded hash structures to reduce update costs by gracefully deferring and grouping writes in main memory and flash memory. Our HF-tree index addresses the gap between access characteristics of flash memory and disk-based indexes. Performance evaluation against the existing BFTL and IPL methods shows superior update and query performance of the proposed HF tree index. Moreover, the HF tree index incurs less erase operations and extends the lifetime of flash memory.

Keywords Flash Memory; Database; Index; Update; Erase

摘要 随着电子技术的发展, 闪存作为一种新型的电子存储设备具有高速的访问速度和无机械延迟的特性。但是由于闪存高昂的写操作代价, 传统的基于磁盘的索引结构如果直接应用在闪存上的话会导致极差的更新性能。文中提出一种新颖的索引结构 HF-tree, 通过组提交、更新合并, 以及多级延迟的方式来提高更新性能。HF-tree 能够有效地克服闪存和现有基于磁盘索引之间的不匹配性的问题。通过和经典的 BFTL 及 IPL 索引的性能比较, 实验结果充分显示了 HF-tree 优越的更新和查询性能。此外 HF-tree 能够有效地减少擦除次数, 从而延长闪存的使用寿命。

关键词 闪存; 数据库; 索引; 更新; 擦除

中图法分类号 TP311

引言

在过去的 20 年中, 由于磁盘的机械延迟特性使得 CPU 和磁盘之间速度的不一致性变得越来越明显。所以对 IO 带宽有大量需求的数据密集型应用, 比如大型数据中心, 都努力寻求更快的存储设备。幸运的是, 闪存作为一种新型的电子存储设备能够提供高速的 IO 性能, 其读速度是磁盘的上百倍。原因在于闪存是一种纯电子设备, 在读取数据时不存在磁盘中效率低下的机械延迟。此外其容量从 1999 年开始每年都翻一番, 三星已经发布了 256GB 的闪存硬盘^[1]。由

于高速 IO 性能和迅速增加的容量, 闪存被普遍认为是取代磁盘成为新一代数据存储设备的理想选择。

数据库是广泛地应用在大型数据中心用来管理数据的重要工具。随着闪存取代磁盘成为新的数据库存储设备, 数据库不可避免的需要移植到闪存上。但是如果直接将目前的数据库应用在闪存上, 其性能, 特别是更新性能, 并不能获得相应于闪存和磁盘 IO 性能比值而带来的提高^[2,3]。在最坏的情况下, 比如小的随机写密集的应用, 数据库在闪存上的性能还不如在磁盘上。其中重要的原因是现有数据库索引结构和闪存物理特性之间的不匹配性。B+树索引是一种被广泛地应用在现有的数据库中的索引结构。然而传统

*本文得到国家自然科学基金项目(课题号: 60833005, 60573091); 国家 863 计划(课题号: 2007AA01Z155, 2009AA011904); 教育部博士点基金项目(200800020002)的支持。

本文的通讯作者为孟小峰

的 B+树索引是针对磁盘的物理特性而设计的。B+树为了维持其本身的平衡结构会产生大量的随机写。众所周知，随机写是导致闪存写性能低下的主要原因，因为随机写容易导致闪存的擦除和大量数据的移动。当 B+树应用在闪存上时，其产生的大量随机写导致索引更新性能的低下。总的来说，索引更新特点和闪存特性之间的不匹配性导致了目前数据库不能充分发挥闪存应有的高速物理特性。

为了解决闪存索引的更新问题，本文提出一种高更新效率的索引结构 HF-tree，同时保持现有索引结构高效查询效率。HF-tree 由 BF-tree 和 Tri-hash 两部分组成。本索引结构能够充分利用 B+tree 索引的高速查询性能和 hash 索引的高速更新性能。BF-tree 通过对传统的 B+tree 应用块存储模型及懒惰分裂合并机制从而有效地支持查询和减少更新的代价。Tri-hash 通过三层级联 hash 结构来有效地从内存到闪存减少和延迟写的操作，从而提高更新的性能。本方法能够有效地解决目前基于磁盘的索引结构与闪存物理特性之间存在的问题，从而获得高速的更新性能和查询性能。本文主要的贡献如下：

- 高速更新性能。相比较于将数据逐个更新到索引中，Tri-hash 通过块提交和写延迟的方式能够有效地提高更新的性能。

- 高效的查询性能。本方法通过 Tri-Hash 来获取最新数据，通过 BF-tree 来查询未修改数据，从而有效地提高了查询性能。

- 低垃圾回收成本。本方法将更新以块为单位写入到闪存中，同时在数据移动时也以块为单位，从而有效地避免了小的写操作，减少了垃圾数据。

本文组织结构如下：第一节介绍了闪存特殊的物理特性给索引结构设计带来的挑战；第二节介绍了闪存数据库索引研究的相关工作；第三节详细地介绍 HF-tree 索引；第四节详细比较了 HF-tree 和典型的索引结构的性能；最后第五节进行了总结。

1. 索引设计挑战

这一节中我们简单回顾一下闪存和磁盘 IO 操作上的不同，并分析其给索引设计带来的挑战。

1.1 读写速度不一致

磁盘具有相同的读写速度，但是闪存的读写速度不一致。根据 Samsung K9WAG08U1A 数据手册^[4]，闪存读的速度是写的 8 倍，而写又是擦除的 8 倍。因此基于闪存的索引结构应该尽量减少写，适当增加读来提高索引的性能。

1.2 重写之前必须擦除

不像磁盘可以原地更新数据，闪存在重写一个页的数据前必须先将该页所在的块全部擦除。擦除操作不仅代价较高而且粒度较大，容易导致数据的大量移动，因此闪存通常采取异地更新的策略。异地更新给索引设计带来最大的挑战是级联更新，即叶子节点的更新会导致所有祖先节点的更新，引发大量的随机写操作。因此如何避免因为级联更新而导致写性能的大幅下降是闪存数据库索引设计的一个重要方面。

1.3 磨损平衡

众所周知，闪存的每个块具有有限的擦除次数，一般为 10000~100000 次。磨损平衡技术采取在一定时间范围内将冷数据和热数据存储位置进行交换的策略来使得每个块具有相近的擦除次数，从而提高闪存的使用寿命。但是数据的移动会导致大量的写，使得索引的性能下降。所以闪存数据库索引设计要在保证磨损平衡的基础上充分提高读写性能。

2. 相关工作

为了克服闪存独特物理特性给索引设计带来的挑战，大量的工作研究如何提高闪存的读写性能。这些工作主要分为块映射技术和纯闪存索引两个方面。

块映射技术主要来自 FTL 技术^[5]。FTL 技术通过将闪存的物理地址映射为逻辑地址，从而提供和磁盘相同的 IO 接口，掩盖了闪存的异地更新。许多工作致力于提高 FTL 的性能^[6-8]。这些工作主要从以下两个方面进行改进：组提交策略和调整上层索引以适应 FTL 的策略。组提交策略将随机写在内存中进行缓冲，在适当的时刻将随机写批量写到闪存中。这方面典型的工作有 BF²L^[6]，NFTL^[7]和 FAST^[8]。BF²L 在随机写达到一个页的时候将数据写入到闪存；NFTL 在 BF²L 的基础上将内存对同一个数据的多次操作进行合并；FAST 将冷热数据存储在不同的地方，通过减少冷数据的移动从而提高写的性能。其它一些索引结构则在 FTL 之上调节现有索引结构来充分利用 FTL，以提高性能^[9-12]。FlashDB^[10]将数据分为写密集型和读密集型数据，写密集型数据采取日志存储模式，读密集型数据采取磁盘存储模式。日志存储模式充分利用了 FTL 的特性在读写速度上取得较好的均衡。尹少宜^[13]则提出了针对日志的删除和恢复方法。

块映射技术性能不足的主要原因是其将闪存模拟成磁盘，导致其高速访问特性不能充分的发挥。因此纯闪存索引直接通过闪存物理地址来访问数据。纯闪存索引一般采用日志^[14]的方式来记录更新，从而减少写的代价。JFFS3^[15]在内存中维护一个日志树来延迟和减少写的次数。数据在闪存中采取异地更新的方

式来减少原地更新的代价。IPL^[2]则采取将数据和其相关的日志存储在同一个块中。此外基于嵌入式设备的 LGeDBMS^[16]也是采取基于日志的方式来记录数据的更新,让 DBMS 绕开文件系统直接管理闪存上的数据,从而提高数据库的性能。

总的来说,块映射和纯闪存索引技术无法发挥闪存的高速访问性能,因此本文提出一种无日志的纯闪存索引结构,解决了闪存索引中存在的问题。

3. HF-Tree

在本节中,我们首先整体介绍 HF-tree 的结构,然后分别介绍 BF-tree 和 Tri-Hash 结构,最后展示了 HF-tree 如何支持通用的索引操作。

3.1 总体结构

如图 1 所示, HF-tree 由 BF-tree 和 Tri-hash 两部分组成。数据的更新首先存储在 Tri-hash 中,并最终合并到 BF-tree 中。执行查询时也是首先查找 Tri-Hash,如果在 Tri-Hash 中不能得到结果,那么将继续查询 BF-tree。插入和删除操作都将转化为更新操作,并在合并的过程中对原始数据进行修改。

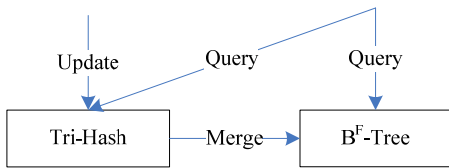


Fig. 1 Architecture and operations of HF-Tree

图 1 HF-Tree 总体结构及操作

3.2 BF-tree

根据之前的讨论可知,传统的 B+tree 因为异地更新带来的级联更新问题而导致写性能的不足。因此,本文提出一种基于闪存特性而优化的 BF-tree,其对传统 b+tree 改进的方面如下:

1. BF-tree 采取如图 2 所示的块存储模型。不同层的节点存储在不同类型的块中。根节点,索引节点和叶子节点分别存储在根块,索引块和叶子块中。叶子块的前 48 个页用来存储叶子节点,后 16 个页用来存储叶子的更新。相比较于传统的 B+tree, BF-tree 能够有效地缓冲叶子节点的更新,从而能够推迟并减少因为叶子节点更新带来的级联更新。

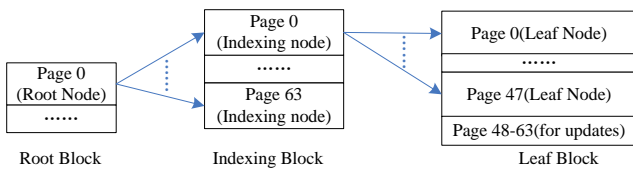


Fig. 2 The Block-based Storage Model of BF-Tree

图 2 BF-tree 的块存储模型

2. 不同层的 BF-tree 采取不同的扇出。假设一个索引项为 8 个字节,那么一个 2KB 的闪存页能够存储 256 个索引项。当叶子节点的扇出小于 256 时,会导致空间利用率较低。另一方面,如果扇出大于 256 的话,一个叶子节点将存储在多个块上,那么一个索引项的更新将会导致多个页的更新,从而产生高昂的更新代价。在我们的方法中,根节点扇出为 64,二级索引节点刚好位于同一个块中。二级索引节点的扇出为 48,这样对应于叶子块中的 48 个叶子节点。而叶子节点的扇出则为 256,这样可以充分利用闪存的存储空间而且减少更新时闪存的写次数。

3. 节点的合并和分裂只有在更新和原始数据合并时才进行。在叶子块中,前 48 个页用来存放叶子节点的原始数据,后面 16 个页用来存放叶子节点的更新。当更新产生时,并不马上将更新和叶子节点进行合并,只有当 16 个更新页全部写满的时候,才将更新和原始数据进行合并。在合并的过程,进行节点的分裂和合并。

3.3 Tri-Hash

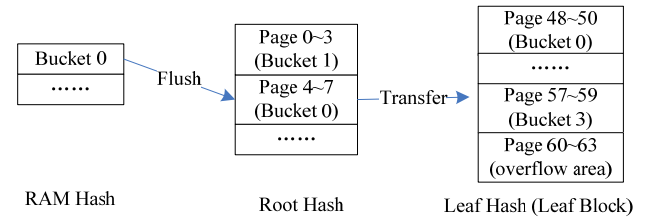


Fig. 3 Storage model and work flow of Tri-Hash index

图 3 Tri-Hash 存储模型及更新流程

在 BF-tree 的叶子块中,最后的 16 个页用来记录叶子节点的更新。但是如果一旦节点有更新就马上写入到叶子块中的话,会导致叶子块的迅速消耗光,从而触发代价高昂的合并操作。为了避免这种情况,我们设计了 Tri-Hash 索引结构来有效地延迟和合并更新。如图 3 所示, Tri-Hash 由三个级联的 hash 结构组成。下面将分别介绍三个 hash 结构。

1. RAM Hash. RAM Hash 位于内存中,用来在内存中对更新进行延迟与合并,并最终将更新批量提交到闪存中。RAM Hash 桶的数量由 BF-tree 的根节点的索引项数目决定。在本文中 RAM Hash 一共包含 64 个桶,每个桶大小为 2KB,和闪存的一个页大小一致。其 Hash 函数如公式一所示。这样属于两个键值之间的更新将位于 RAM Hash 同一个桶中,也就是将属于一个块的数据更新放在同一个桶内。这样 RAM Hash 不仅可以对更新进行延迟和合并,而且在更新以桶为单位进行批量提交时,每个桶的更新只涉及到一个闪存块,有效地减少了垃圾数据的产生量。

$$H(key) = i; \quad Ki < key < Ki+1; Ki \in root\ node \quad (1)$$

2. Root Hash。Root Hash 用来对更多的更新进行延迟和批量提交。因为 RAM Hash 位于内存中，其大小受到限制，所以缓冲的更新数量有限。如果直接将 RAM Hash 中的更新写入到 BF-tree 中的话，仍然会导致大量随机写。所以 Root Hash 利用更大的闪存空间来进行更新的缓冲。如图 3 所示，每个 Root Hash 桶包含 4 个连续的页，其 Hash 函数和 RAM Hash 一样，每个 RAM Hash 桶对应唯一的一个 Root Hash 桶。当 Root Hash 桶中的数据满的时候，就将数据提交到下一级的 Hash 函数 Leaf Hash 中。

2. Leaf Hash。Leaf Hash 用来接收从 Root Hash 提交而来的更新，并在桶内更新满的时候将更新合并到 BF-tree 中。如图 3 所示，Leaf Hash 位于 BF-tree 的叶子块的最后 16 个页中。和前两级 Hash 索引不同的是，Leaf Hash 只是面向所在的块的数据，也就是说每个 BF-tree 的叶子块都有一个 Leaf Hash 函数。其 Hash 函数公式 2 所示。Leaf Hash 包含 4 个桶，每个桶都包含本块内 BF-tree 相应叶子节点的更新。其中溢出区域用来记录某些桶中更新过于频繁的数据。

$$H(key) = i; K_{12^*i} < key < K_{12^*(i+1)}; K_{12^*i} \in \text{indexing node} \quad (2)$$

3.4 Tri-Hash 与 BF-Tree 的集成

在我们设计中，每个叶子块的前 48 个页用来存储 BF-tree 的叶子节点，后面 16 个页用来存储 Leaf Hash。如图 4 所示，当 Leaf Hash 满的时候，合并操作将被执行。在合并的过程中，将 Leaf hash 中的更新和原先的 BF-tree 中叶子节点的数据进行合并，新版本的数据替代老版本的数据。在叶子块中，每 12 个连续的页对应 Leaf hash 的一个桶，在合并时，用桶中的数据替换原来的数据，这样桶又重新变成空。

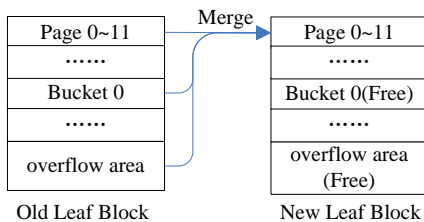


Fig.4 Merge flow during the integration of BF-Tree and Tri-hash

图 4 BF-Tree 与 Tri-Hash 集成过程中的合并流程

合并结束后，新的叶子块仍然保留 Leaf Hash 桶，但是数据已经与原始数据进行合并。通过合并操作，可以将最初从内存中的更新最终写入到 BF-Tree 中。

3.5 HF-Tree 索引基本操作的实现

前面我们介绍了 HF-Tree 的索引结构，这里详细介绍如何执行索引的基本操作，如更新和查询。

1.更新。一个索引项由<Key, pointer>组成，首先我们讨论 pointer 值的更新。当索引项所指数据发生改变时，比如元组的非主键值发生改变，由于异地更

新的策略，数据会存储在新的位置，所以索引项的 pointer 必须改变以指向新的位置。如算法 1 所示，当一个索引项更新时，我们首先将更新写到 RAM Hash 中，如果 RAM Hash 满了的话，就将 RAM Hash 中的更新写入到 Root Hash 中。同理 Root Hash 满的话，就将其中的更新写入到 Leaf hash 中。最后当 Leaf Hash 满的时候，启动合并操作将 Leaf hash 中更新写入到 BF-Tree 中。这样就将一个更新从最初的 Tri-Hash 最终写入到了 BF-Tree 中。这个过程中，由于经过三层级联的 Hash 索引结构的缓冲，更新被有效地延迟与合并，从而减少写的次数。此外数据的移动始终以块为单位，所以在数据移动的过程中，本方法有效地减少垃圾数据的产生。

其次考虑 key 值的更新。一个 key 值的更新可以分解为 key 的删除和插入操作。对于 Key 的插入操作，我们可以使用 key 值和新的地址表示。对于 key 的删除操作，我们用 key 值和一个空地址表示。所以两者都可以使用前一种基于 pointer 更新的方式来表示。

Algorithm 1: UpdateHF-tree

```

Public UpdateOfHFtree(int updateType){
    Generate update dataset;
    If(bucket of RAM hash is not full) write update;
    else{
        If(bucket of Root hash is not full) flush updates;
        else{
            If(leaf hash is not full) transfer updates;
            else{
                Merge leaf nodes with leaf hash;
            }
        }
    }
}

```

2.查询。HF-Tree 能够有效地支持对等查询和范围查询。对于对等查询，我们首先查找 Tri-hash。在 Tri-Hash 中的查找次序依次 RAM hash, Root Hash 和 Leaf Hash。如果一旦找到该数据，就马上结束查找过程。如果 Tri-hash 中不能得到该数据，那么在 BF-Tree 中查询该数据。在 BF-Tree 中查找方式和 B+tree 一致。

对于范围查询，其算法过程如算法 2 所示。假设需要查找 key 值范围为[a, b]。首先我们查找 Tri-Hash 已得到最新的数据。首先得出 a、b 所在的桶号 n_a、n_b，然后在获得 RAM Hash 和 Root Hash 桶号在 n_a 和 n_b 之间所有记录。其次读取[n_a, n_b]个桶所对应的 Leaf Hash 索引中所有记录。最终将两个结果进行合并，就可以得到从 Tri-Hash 中所有的数据。第二步是从 BF-Tree 进行范围查询，查询的方法和传统 B+tree 一致。再将从 Tri-hash 和 BF-Tree 中查询的结果进行合并就可以得到最终的查询结果。

Algorithm 2: Rang query of HF-tree

```
Public RangQueryOfHFtree(int a, int b){
    Get bucket NO. n_a, n_b for a, b;
    For each bucket n_i from n_a to n_b {
        Get result from RAM: R_RAM;
        Get result root hash: R_root;
        For each bucket of leaf hash related to n_i{
            Get result: R_leaf;
        }
        Merge R_root into R_leaf;
        Merge R_RAM into R_leaf;
    }
    Get result R_B from BF-tree where key item in [a, b];
    Merge R_root with R_B;
}
```

4. 实验结果与分析

本节中，我们将 HF-Tree 和 BFTL 及 IPL 两个典型的闪存索引进行了详细的比较。本实验主要根据不同的数据流从更新和查询性能来评估索引的性能。此外我们也比较了索引导致的擦除次数。

4.1 实验环境

本实验中使用的 PC 配置为：Intel Core 2 Pentium 4 Duo CPU 2.83GHz，2GB 内存。操作系统为 Linux fedora 8，编程语言为 C 语言。HF-Tree 为一种纯闪存索引，通过闪存的物理地址来访问数据。所以我们采取国际上通用的闪存模拟器来进行性能的评估。本文中使用的模拟器为 Linux 系统中的 NandSim 模拟器，其参数按照三星 K9WAG08U1A 数据手册来配置。一个块包含 64 个页，每个页为 2KB。

实验中使用四个常见数据模型和测试标准来产生数据集：随机分布，正态分布，TPCC 和 Postmark。利用这些测试集来测试索引的更新性能。对于查询，则进行了对等查询和范围查询的对比实验。需要注意的是图 5 的前 8 个子图的 Y 轴坐标为对数形式。

4.2 随机更新模型

随机分布主要用来评价当各个节点具有大致相似的更新频率时索引的更新性能。图 5 (a)和(b)展示了 BFTL, IPL 和 HF-Tree 的随机更新性能。为了能够直观的展示数据，坐标轴都是采取对数的形式来表示。经过 1000000 次更新，我们分别得出三种索引写和擦除的次数。如(a)所示，BFTL 和 IPL 分别进行了 772442 和 66867 次写，而 HF-Tree 只花费了 18664 次写。和 BFTL 及 IPL 相比，HF-Tree 在性能上分别提高了 97%和 72%。这也说明了 HF-Tree 通过三级

hash 结构能够有效地延迟和减少写的次数。此外由于数据在移动的过程中始终都是以块为单位，所以垃圾数据较少，从而有效地减少了擦除次数，如(b)所示。

4.3 正态分布更新模型

在实际的应用环境中，大多数数据符合正态分布模型。在正态分布模型中，key 的更新频率符合正态分布中的概率密度函数，如公式 3 所示：

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, x \in R \quad (3)$$

正态分布主要用来评价当各个节点更新频率不一致的情况下索引的更新性能。在不同的标准方差下都进行 1000000 次更新。其实验结果如图(c)和(d)所示。当标准差为 1 时，HF-Tree 比 BFTL 和 IPL 性能分别提高 84%和 60%；标准差为 5 时，分别提高 96%和 63%。实验结果不仅说明了 HF-Tree 更新性能的优越性，更加表示了 HF-Tree 能够更好的处理更新频率差异较大的数据流。(d)图所示的擦除次数同理说明了 HF-tree 能有效地减少垃圾数据的产生，从而减少擦除的次数。

4.3 TPCC 测试集

TPCC 是数据库事务处理性能的标准测试集。我们通过修改 PostgreSQL 得到其在 TPCC 测试时的 IO 队列，并作为我们的数据集。通过执行 TPCC 产生的写操作，本方法和 BFTL 及 IPL 的比较结果如图 5 (e)和(f)所示。TPCC 测试集所产生的 IO 操作随机性很强，数据也非常分散，而本方法能够充分处理随机性的数据。实验表明在写的次数上，本方法比 BFTL 和 IPL 分别提高 95%和 85%，充分体现本方法在索引更新上的高效性。

在擦除次数上，本方法的优势更加明显。擦除次数的减少都是在 90%以上，除了本方法的块提交策略带来的提高之外，TPCC 数据集产生的 IO 较少也是一个原因。因为本方法可以缓冲大量的更新，从而在数据量较少时优势更加明显。

4.3 Postmark 测试集

Postmark 数据集主要用来测试文件系统的 IO 性能。本方法主要利用其来产生 IO 操作，并检验索引的更新性能。我们利用 Blktrace 来获取 Postmark 运行过程中产生的 IO 操作。Postmark 在运行期间产生大量的读写操作，而且数据也是非常分散。但是局部数据访问比较连续。因此利用此数据集可以同时检测本方法在随机和连续访更新性能上的性能。

其实验结果如图 5 (g)和(h)所示。对于写性能，本方法比 BFTL 和 IPL 分别提高 95%和 75%。实验结果充分表现了本方法的高效更新性能。同时我们也可

以看到 IPL 的性能有所提升,其原因在于 IPL 能够较好的处理局部连续更新的数据。同理,对于擦除操作,本方法也能够大量地减少擦除操作。其擦除性能的提升主要来自于本方法的更新始终以块为单位进行提交,从而有效地减少了垃圾数据的产生。

4.4 查询性能

对等查询是索引最基本的查询类型。通过对三种类型的索引进行 1000000 次随机查询, IPL 平均一共进行了 6885312 次读操作,但是 HF-Tree 平均只进行了 3968960 次读操作。相比于 IPL, HF-Tree 在对等

查询性能上提高了 42%。主要得益于 HF 通过 Tri-Hash 和 BF-Tree 来提高查询速度,而 IPL 必须扫描日志来获取最新的数据。

同理对于范围查询, IPL 必须频繁而且重复地扫描日志区来得到最新版本的数据。因为 HF-Tree 将索引项在 Tri-Hash 和 BF-Tree 中都是有序存放,所以范围查询能够可以简单按照算法 2 来获得查询结果。实验结果如图(i)所示, HF-Tree 相比较于 BFTL 和 IPL 具有更好的查询性能。特别是查询的范围越大, HF-Tree 的性能越好。

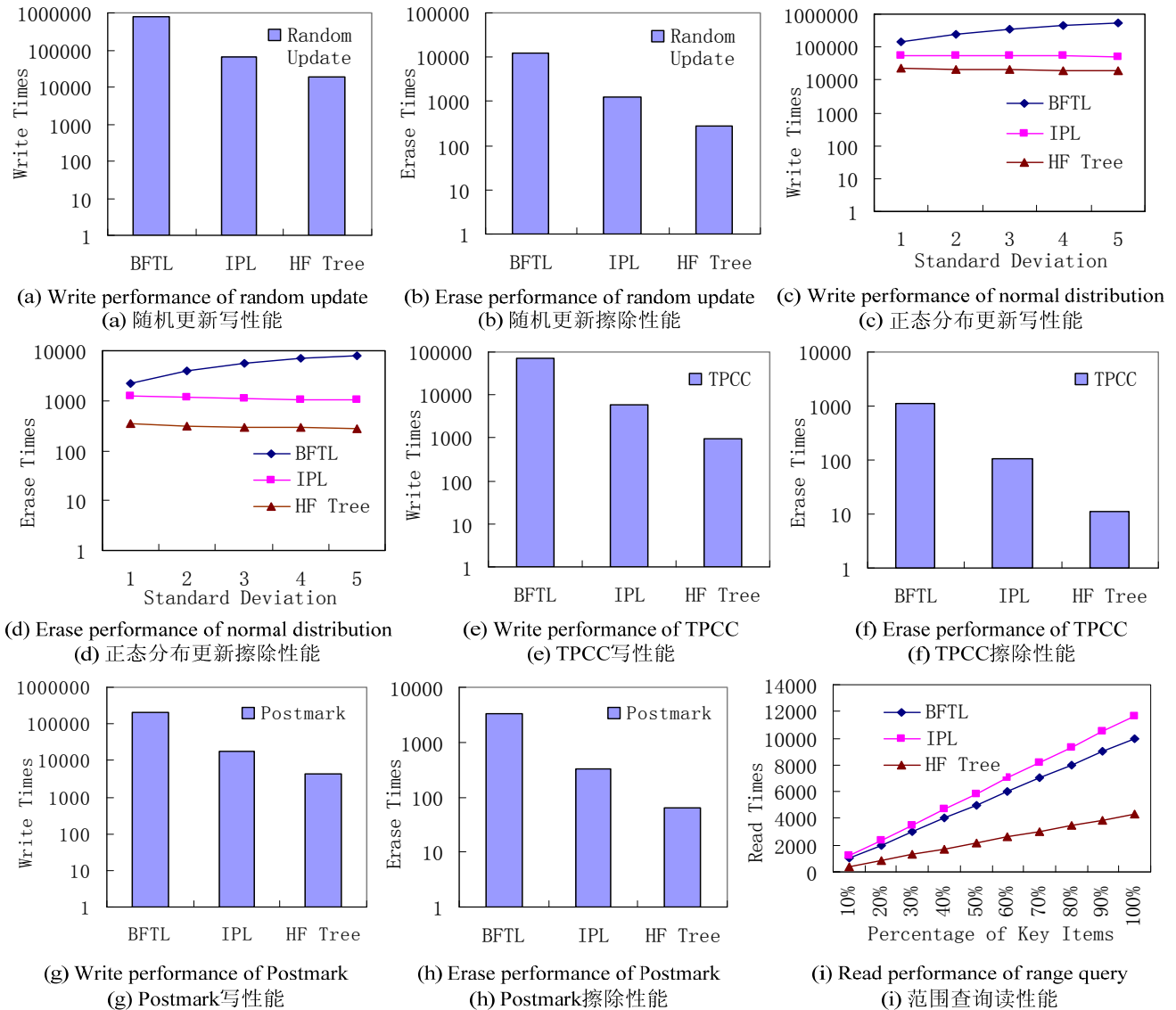


Fig.5 Update performance of BFTL, IPL and HF-Tree under random update model, normal distribution model, TPCC and postmark benchmarks, and range query performance.

图 5 BFTL, IPL 和 HF-Tree 在随机分布, 正态分布, TPCC 测试集和 Postmark 测试集下更新性能, 以及范围查询性能比较图

5. 总结

闪存独特的物理特性, 比如 IO 速度不一致性、重写之前必须擦除等, 导致目前基于磁盘设计的索引

结构不能充分利用闪存的高速读写特性。为此本文提出一种新颖的具有高更新效率的闪存索引结构 HF-Tree。HF-Tree 由两个部分组成: BF-Tree 和 Tri-Hash。索引的更新通过 Tri-Hash 的页批量提交的策略来有效地延迟和减少了写操作。此外, Tri-Hash

的三层级联 Hash 结构在更新时不需要寻址来获取更新的地址, 从而进一步提高更新性能。BF-Tree 通过块存储模型有效地解决了更新带来的级联更新的问题。通过大量和 BFTL 及 IPL 的对比实验充分说明了 HF-Tree 在更新上的卓越性能, 同时在对等查询和范围查询上也表现出优越的性能。

参考文献

- [1] SAMSUNG. PM800 (256/128/64GB, MLC, SATA 3.0Gbps) [EB/OL]. (2009-06-01)[2009-06-15].
<http://www.samsung.com/global/business/semiconductor/products/flash/ssd/2008/down/pm800.pdf>
- [2] Lee S, Moon B. Design of flash-based DBMS: An in-page logging approach [C] //Proc of the ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2007:55-66
- [3] Lee S, Moon B, Park C, et al. A case for flash memory ssd in enterprise database applications [C] //Proc of the ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2008:1075-1086
- [4] Samsung Electronics. 1G x 8 Bit / 2G x 8 Bit / 4G x 8 Bit NAND Flash Memory , Version 1.1 [EB/OL]. (2007-06-18)[2009-06-15]
<http://www.alldatasheet.com/datasheet-pdf/pdf/139788/SAMSUNG/K9WAG08U1A.html>
- [5] Intel-Corporation. Understanding the Flash Translation Layer(FTL) specifications [EB/OL]. (1998-12)[2009-06-15].
<http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf>
- [6] Wu C, Kuo T, and Chang L. An efficient b-tree layer implementation for flash-memory storage systems [J]. ACM Trans. on Embedded Comput. Sys., 2007, 6(3): Article 19
- [7] Tsai Y, Hsieh J, Kuo T. Configurable NAND Flash Translation Layer [C] //Proc of IEEE Int Conf on Sensor Networks, Ubiquitous, and Trustworthy Computing. Taiwan: IEEE, 2006: 118-127
- [8] Lee S, Park D, Chung T, et al. A log buffer-based flash translation layer using fully-associative sector translation [J]. ACM Trans. on Embedded Comput. Sys., 2007, 6(3): Article 18
- [9] Myers D. On the Use of NAND Flash Memory in High-Performance Relational Databases [D].Massachusetts: Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2008
- [10] Nath S, Kansal A. FlashDB: Dynamic Self-Tuning Database for NAND Flash [C] //Proc of the 6th Int Conf on Information Processing in Sensor Networks, Massachusetts:ACM, 2007:410-419
- [11] Koltsidas I, Viglas S. Flashing Up the Storage Layer [J]. PVLDB, 2008, 1(1): 514-525
- [12] Nath S, Gibbons P. Online Maintenance of Very Large Random Samples on Flash Storage [J]. PVLDB, 2008, 1(1): 970-983.
- [13] Yin S, Chen J, Meng X, et al. Storage and Recovery Techniques for PhoneDB [J]. Computer Science, 1992, Vol.32 Suppl.:358-362 (in

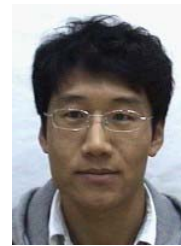
Chinese) (尹少宜, 陈继东, 孟小峰等. 手机数据库 PhoneDB 的存储和恢复技术[J]. 计算机科学, 2005, 卷 32(增刊 A): 358-362)

- [14] Rosenblum M, Ousterhout J. The Design and Implementation of a Log-Structured File System [J]. ACM Trans. Comput. Syst., 1992, 10(1): 26-52
- [15] Bityutskiy A. JFFS3 design issues [EB/OL]. (2005-11-27)[2009-06-15].
<http://www.linux-mtd.infradead.org>
- [16] Kim G, Baek S, Lee H, et al. LGeDBMS: a Small DBMS for Embedded System with Flash Memory [C] // Proc of the 32nd Int Conf on Very Large Data bases, Seoul: ACM, 2006: 1255-1258



Zhou Da, male, was born in 1980.10. Ph.D. candidate in computer science from Renmin university of China, Beijing, China. His current interest include indexing, query processing and transaction processing of Flash-based database systems

周大, 男, 1980 年 10 月出生, 博士研究生, 主要研究方向为闪存数据库存储, 索引和事务处理。



Liang Zhichao, Male, was born in 1986.12. M.Sc. Students in computer science from Renmin university of China, Beijing, China. His current interest include query processing of Flash-based database systems

梁智超, 男, 1986 年 12 月出生, 硕士研究生, 主要研究方向为闪存数据库查询处理。



MENG Xiao-feng was born in 1964. Male He received the Ph.D degree in Computer Application and Technology from Institute of Computing Technology Chinese Academy of Sciences in 1999. Now he is a professor and doctoral supervisor at Renmin University. His research interests include

web data management, native XML databases, mobile data management, etc.

孟小峰(1964-), 男, 河北邯郸人, 1999 年于中国科学院获计算机应用技术专业工学博士学位, 目前是中国人民大学教授, 博士生导师, 主要研究领域为 Web 数据管理、XML 数据库、移动数据管理。