文章编号：1674－9057(2012)04－0547－06 doi:10.3969/j.issn.1674－9057.2012.04.018

支持 MMDB 缓存优化的索引结构研究

张 少 娴

（南京邮电大学 计算机学院，南京 210046）

摘 要：针对目前内存数据库中索引缓存失配的问题，提出一种基于缓存优化的 MCTH 树索引结构。它不仅考虑到缓存失配对系统性能的影响，同时也考虑到 TLB 失配的因素，对 MCTH 树索引结构的分析和实验测试表明：MCTH 树索引结构与相关索引结构相比具有更好的有效性和优越性。

关键词：内存数据库；MCTH 树；索引；缓存优化

中图分类号：TP311 **文献标志码**：A

0 引 言

自 20 世纪 80 年代提出内存数据库（MMDB）概念以来，得到了研究者的广泛关注。在 MMDB 中，由于没有磁盘的 I/O 操作，因此提高查询效率的关键变成了处理器的计算时间和缓存的有效利用率。处理该问题的方法很多，其中一种方法就是建立合理的数据索引结构，在缓存容量有限的情况下，减少查询过程中的匹配失效，从而缩短处理器的执行时间，使性能得到最大优化。一些基于缓存的索引结构陆续被提出来，这些针对缓存的访问方法包括基于树的索引结构和基于 Hash 算法的索引结构[1]，前者包括目前最常用的 T 树、B + 树和 CSB + 树等，后者则包含可扩展 Hash 和通链 Hash。这几种基于缓存敏感技术的索引结构要么是没有考虑缓存对索引性能的影响，要么就是不够周全，没有考虑 Hankins 等的实验结果带给索引的影响因素。Hankins 等通过大量的实验，提出了索引遍历时的执行时间计算公式[2]，公式中有 4 个重要的指标：执行指令的总条数（I）、数据缓存的失配次数（M）、分支预测失败的次数（B）和 TLB 失配的次数（T）。

$$t = I \times cpi + M \times miss_\ \text{latency} + B \times pred_\ \text{penalty} + T \times tlb_\ \text{penalty}。 \quad (1)$$

式中：t 为查询的总执行时间（CPU 时间），cpi 为执行一条指令所需的 CPU 时间，$miss_$ latency 为缓存失配所损失的 CPU 时间，$pred_$ penalty 是分支预测失败所损失的 CPU 时间，$tlb_$ penalty 是 TLB 失配所损失的 CPU 时间。该式对缓存索引树的设计有重要的参考价值。

本文针对缓存索引性能的不足，综合考虑式（1）中影响索引执行时间的指标和实验发现的结果，提出了一种全新的索引结构 MCTH 树。

1 MCTH 树索引结构

根据缓存索引的相关研究可以得知，T 树设计的重点在于降低空间占用率，没有为缓存做优化；Hash 索引访问速度快，容易实现，但不支持范围检索[3]。综合考虑两种索引结构，借鉴 Hash 索引查询速度快的优点，并对 T 树进行缓存优化，进而提出一种新的索引结构 MCTH 树，假设每个 Hash 表由 3 个 Hash 桶组成，如图 1 所示。

收稿日期：2012－08－22

基金项目：国家自然科学基金面上项目（60973046）；中国博士后科学基金（20100471353）

作者简介：张少娴（1972—），女，硕士，讲师，研究方向：数据库、信息安全，zhangsx@njupt.edu.cn。

引文格式：张少娴. 支持 MMDB 缓存优化的索引结构研究［J］. 桂林理工大学学报，2012，32（4）：547－552

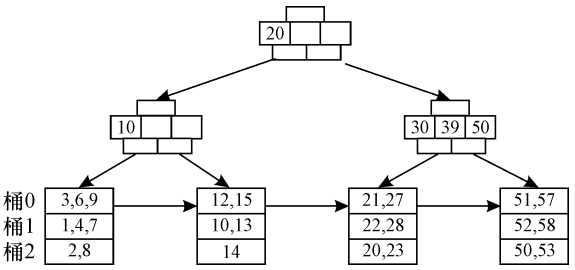


图 1 MCTH 树实例

Fig. 1 MCTH-tree structure

MCTH 树第一层结构是一棵 T 树，T 树是一种一个节点中包含多个关键字的平衡二叉树，具有以下特点[4]：① 左子树与右子树之差不超过 1；② 在一个存储结点可以保存多个键值，它的最左与最右键值分别为这个节点的最小与最大键值，左子树仅仅包含那些键值小于或等于最小键值的记录，同理右子树只包括那些键值大于或等于最大键值的记录；③ 同时拥有左右子树的节点被称为内部节点，只拥有一个子树的节点被称为半叶，没有子树的节点被称为叶子；④ 为了保持空间的利用率，每一个内部节点都需要包含一个最小数目的键值；⑤ 每个节点中拥有最大最低边界值和最小最高边界值。对内部节点来说，最大最低边界值是中序遍历时前一个节点中数据的最大值；最小最高边界值是中序遍历时后一个节点中数据的最小值。第二层是 T 树的叶子结点，每个叶子结点的数据以 Hash 方式组织，桶内存储关键码和指针，当然有些桶可能是空桶[5]。叶子结点内的数据是无序的，但是树的叶子结点之间是有序的，将每个叶子结点链接起来（即对每个叶子结点加上一个后继指针），这样可以避免进行范围查询时指针的上下移动。由于 Hash 方式访问非常快，于是可以将叶子结点增大，增大叶子结点对于 Hash 访问的速度几乎没有影响，但是对提升索引结构的性能将会产生很大的影响。MCTH 树对缓存作了优化，这是通过 MCTH 树结构特点来实现的。根据式（1），本文提出一个简化后的索引遍历执行时间的计算公式

$$t = I \times cpi + M \times miss_\text{latency} + T \times tlb_\text{penalty}, \quad (2)$$

上式中的 3 项重要指标和式（1）类似，只是不考虑分支预测失败的次数。因此影响索引遍历的时间主要是执行指令的总条数（I）、缓存失配次数（M）以及 TLB 失配次数，如果通过建立某种索引能够减少缓存失配的次数和 TLB 失配的次数，那么这种索引就是优化的。下面通过对 MCTH 树定义和结构的分析，验证 MCTH 树索引是优化的索引结构。

1）MCTH 树将内部结点设置为处理器缓存块（cache line）的大小，这样每个内部结点的访问最多导致一次缓存失配；同时将叶子结点每个桶的大小设置为缓存块的 1.2 倍，这样可以最大程度的减少缓存失配的次数[6]。

2）叶子结点增大，并且 Hash 桶链结构可以存储很多键值，这就使得 MCTH 树所需的内部结点更少，树的深度变小，缓存失配次数和 TLB 失配次数都会减少。这是因为，缓存失配减少后，L2 级缓存命中的概率就会增加，当树的结点还在缓存中的时候，TLB 失配的概率就会小很多；同时，对于单次查询，TLB 失配的次数等于树的高度，对于接下来的查询，TLB 失配的次数就会比树的高度小很多，因此，当树高度降低，并且缓存命中概率增加的时候，TLB 失配的概率也会相应的减少，MCTH 树索引结构促进了这种良性循环。

3）尽管 Hash 表（叶子节点）内数据是无序的[7]，但是各个叶子节点之间是有序的，这弥补了桶内键的无序性给范围查询带来的不便，同时可以借助 Hash 索引查询速度快的优势。这就把 T 树和 Hash 索引各自的优势结合起来，可以使查询优化达到最佳效果。

因此 MCTH 树索引结构可以很好的减少缓存失配次数和 TLB 失配次数，从而缩短索引的遍历时间，提高查询的速度，进而可以达到查询优化的目的。

MCTH 树含有多个关键值 N，查找的关键值需要跟节点的最大值和最小值之间进行比较，而不是与单个值比较，因此可知包含 N 个关键值的 MCTH 树，第一层的 T 树将具有的节点数为 N/K，而 Hash 算法的最高时间复杂度是 $O(1)$，因此得到 MCTH 树的时间复杂度为 $O(\log 2N + (1/2) * \log 2N/K + 1)$，这与 T 树的时间复杂度基本一样，MCTH 树的时间复杂度与 T 树是同一个数量级。从空间复杂度来说，Hash 表空间复杂度为 $O(N)$，T 树的空间是 N/K 个节点的空间，因此，MCTH 树的空间复杂度为 $O(N + N/K)$。虽然，MCTH 树的空间复杂度有所增加，但是第一层 T 树空间的减少可以弥补

MCTH 树空间复杂度的增加，MCTH 树索引结构的空间复杂度与 B 树、CSB + 树等相比是一个数量级的。

2　MCTH 树数据结构

根据 MCTH 树索引结构的特点建立索引对象的数据结构如下：

typedef struct {
DBHANDLE idxHandle;//索引句柄
DBHANDLE idxHandle;//拥有本索引的表句柄
BYTE idxType;//索引类型
BYTE idxKey[IDXKEYNUM];
DWORD idxKeyDim[IDXKEYNUM];//一次哈希索引关键字范围
DWORD dwModulus;//散列 Hash 模值，MCTH 树非叶子节点总数
DWORD idxItemLen;//索引项长度（MCTH 树非叶子节点索引项长度）
DWORD idxItemNum;//当前索引项数
DWORD idxItemNo;//当前索引号，散列哈希时保存哈希入口值
DWORD idxKeyLen;//表关键字长度
DWORD tupleKeyLen;//表各个关键字长度之和
LPSTR idxAddr;//索引表项地址，MCTH 树索引非叶子节点地址
LPSTR idxBulkAddr;//索引桶地址，指向散列索引桶地址
DBHANDLE idlequeHandle;//MCTH 树非叶子空闲索引节点队列句柄
DBHANDLE idlequeHandleLeaf;//MCTH 树叶子空闲索引节点队列句柄
LPSTR idxAddrLeaf;//MCTH 树索引，叶子节点地址
DWORD dwModulusLeaf;//MCTH 树索引，叶子节点总数
DWORD idxRoot;//MCTH 树索引的根节点号
DWORD idxBTKeyNo;//MCTH 树当前关键字在当前节点上的位置
KEYHASH lpKeyHash;//哈希方法
HASHKEYMERGE lpHashKeyMerge;//哈希关键字合成方法
} IDXRT, * LPIDXRT;

① 创建 MCTH 树索引

原型：DBHANDLE-CreateIndex（DBHANDLE hDB, BYTE idxMatchType, BYTE idxMethodType, DWORD idxKeyLen…）
输入：Hdb//被索引的表对象句柄
idxMatchType//索引匹配类型
idxMatchType//索引方法类型
idxKeyLen//索引关键字长度
返回值：成功返回索引句柄，失败返回 0
{
1 输入各个索引字段，根据输入的字段找到该字段的信息
2 调用-AddIndex（hDB, idxMatchType, idxMethodType, ucKeyLen, dmKeyNum）注册一个索引，该函数对索引对象做初始化
3 给索引字段加上关键字段标志
4 返回索引句柄
}

② 关键字合成

原型：DBBOOL-MergeIdxKey（LPSTR lpKey, DBHANDLE hIDX…）
输入：hIDX//索引句柄
输出：lpKey//合成后的关键字
{
switch（字段类型）
case 字符型：
if（lpDmInfo - > dmLen = = 1）//为一个字符的长度
{
调用 va-arg 获取当前参数内容；
将内容拷贝至 lpKey + offest 处；
将字段偏移量加上当前字段长度，break；
}
else//字符数组
{
调用 va-arg 获取当前参数内容；
并将内容拷贝至 lpKey + offest 处；
将字段偏移量加上当前字段长度，break；
}
case…//其他数据类型
}

③ 根据关键字获得记录指针

原型：DBBOOL-FetchIndex（DBHANDLE hIDX, LPSTR TupleKey, LPDWORD lpTupleNo）
输入：hIDX//索引句柄
TupleKey//索引关键字
输出：lpTupleNo//指向记录指针
//根据关键字，利用索引查找记录的记录号，输出指向记录的指针

3 MCTH 树算法描述

根据前文提出的 MCTH 树索引结构，设计 MCTH 树算法，先建立 MCTH 树索引结构，然后设计查询算法、插入算法、分裂算法和删除算法等几种常用的算法。MCTH 树的建立可以分为以下几个步骤：① 把根结点设置为空，并创建一个空结点，然后创建一个 Node 对象，把 Node 处的数据赋给 Data 变量；② 如果插入新结点的数据小于当前结点内的数据值，就把新结点设置为当前结点的左子树（递归插入操作），否则跳到步骤④；③ 如果当前结点的左子树结点为空，就把新结点插入并且退出循环，否则，跳到下一次循环操作中；④ 把新结点设置为当前结点的右子树；⑤ 如果当前结点的右子树为空，把新结点插入并且退出循环，否则，跳到下一次循环中；⑥ 把新结点的数据按照哈希函数规则存储到桶链结构中，这与桶链 Hash 规则一样。建立好 MCTH 树结构后，就可以对树结构进行调整，并设计 MCTH 树操作算法，下面将介绍几种操作算法的思想。

3.1 MCTH 树插入算法

对于将关键字 key 插入到 MCTH 树，插入是递归的，算法步骤描述如下：

① 通过查找算法找到 key 所应该插入的叶子结点（Hash 表），通过 Hash 计算，找到 key 所对应的 Hash 桶，如果桶未满，则将 key 插入该桶后结束。

② 如果桶已满，则分裂该结点，并把其中的键重新 Hash，并分配到两个新结点中，分裂的分界是最接近该 Hash 表所能容纳键的最大值和最小值的平均值的键。

③ 某一层的分裂在其上一层看来，相当于是要在这一较高层次上插入一个新的键（指针对），这时的操作和 T 树的插入操作相同。

3.2 MCTH 树查询算法

任何成功的索引结构都支持两类查询操作：精确查询和范围查询，精确查询是对特定键的查询，范围查询是对一系列键的查询。为了执行查询 Search［key，θ value］，要获得匹配的 Hash 表（key，table）。给出一个关键字 key < ∞，找到键值 key 和相关的 Hash 表 table，使 key 满足 key θ value，这里 θ 是一个比较操作符[8]。

MCTH 树的查找过程是递归查找，查找过程可以归纳为两点。若处于叶子节点（Hash 表），使用 Hash 方法在 Hash 表中查找，定位 Hash（key）所在的桶，在桶中做顺序查找，如果找到，则指针指向所需的记录。若处于某个内部节点，且它的键为（K_1，K_2，…，K_n），如果 key < K_1，则为第一个子节点；如果 $K_1 \leqslant$ key < K_2，则为第二个子节点，依此类推，在这一子节点上递归查找。

对于 MCTH 树的范围查询，由于叶子节点之间是有序的，仅需要扫描范围查询覆盖的那部分叶子节点。例如，在查找关键字的范围［min，max］，通过一次查找来找到 min，不论它是否存在，都将达到可能出现 min 的 Hash 表，然后在该 Hash 表中查找 min 和大于 min 的那些键，如果查找完当前的 Hash 表，仍没有发现大于或等于 max 的键，就沿着当前 Hash 表的后继指针，到达下一个 Hash 表，并继续查找，直到找到一个大于或者等于 max 的键，这时停止查找；或者到了 Hash 表的末尾，然后到达下一个 Hash 表，重复这个过程。

3.3 MCTH 树分裂算法

假定叶子节点 N 的某个 Hash 桶已满，如果试图将一个新的键插入到该 Hash 桶，这时，执行如下步骤：① 创建一个新的叶子节点（Hash 表）A，它将是 B 的兄弟，并且紧挨在 B 的右边；② 以最接近该 Hash 表所能容纳键的最大值和最小值的平均值的键为界将原有键和新插入的键重新 Hash 放到 A 或 B 中；③ 作为界的键被插入到节点 A 和 B 的父节点用于划分这两个节点之间的查找；④ 如果父节点未满，则直接插入到父节点相应位置，若如节点已满，则按照 T 树节点分裂方式分裂父节点[9]。

3.4 MCTH 树删除算法

删除一个具有给定键 K 的记录，必须先定位该记录和它在树中的叶子结点（Hash 表），这部分删除过程主要是查找，然后删除记录本身，并删除 MCTH 树中的键和指针对。删除算法的关键是，当删除了给定的键之后，如果发生删除的结点在删除后还有最小数目的键和指针，那么就不需要再做什么，但是，结点有可能在删除之前正好具有最小的满度，因此，在删除后，对键数目的约束就会失去作用。这时，可以分为两种情况：

1）如果叶子结点空 Hash 桶数目少于事先规

定的约束数目，则查看与它同父结点的兄弟结点满度是否大于规定满度，如果是，则将这两个重新去定 Hash 分界值，分界值确定方法与分裂结点的分界值确定方法相同，修改其父结点的相应的值删除结束。

2）如果删除后叶子结点空 Hash 桶数目少于事先规定的约束数目，并且与它同父结点的子结点的满度也正好等于满度的最小值，则合并这两个结点，修改其父结点的相应值。如果父结点也小于满度的最小值，则分裂父结点，分裂方法和 T 树相同。

4　实验测试

4.1　实验环境

硬件环境：Intel 双核处理器，主频 2.04 GHz；内存 1 G。数据库环境是自主研发的内存数据库系统 QMDB，建立两个关系表：Customers（Cno，Cname，Cservices），Services（Sno，Sfee，Cno）。这两个表分别记录了用户和服务费用的基本信息，用于模拟某计费系统。同时，编写了数据随机生成器程序，用于产生实验操作中需要的数据。为了模拟系统的真实数据特点，实验中设定的记录数为 1 000 000，根据关系表分别建立 B + 树、T 树和 MCTH 树 3 种索引。

4.2　实验结果及分析

4.2.1　查询性能　分别建立 B + 树、T 树和 MCTH 树 3 种索引，并向各个索引中插入相同数量的索引，对 3 种索引做同一序列的查询操作，且查询操作近似服从均匀分布[10]，通过编译器自身的计时器，统计得出 B + 树、T 树和 MCTH 树在不同索引数量情况下每一次查询操作的效率，实验结果如表 1 所示。

表 1　查询效率比较

Table 1　Query efficiency comparison

索引数量/条	查询耗时/ms		
	B + 树	T 树	MCTH 树
10 000	107	108	96
20 000	312	310	291
50 000	851	867	801
100 000	1 823	1 954	1 594
500 000	10 825	11 651	8 907

可以看出，MCTH 树查询的速度比 T 树、B + 树快，而且如果记录数量越大，MCTH 树则显得更快。主要有两个原因：第一，MCTH 树是一个宽而浅的树，查找一个数据需要访问很少的结点；第二，MCTH 树具有较高的缓存敏感性，MCTH 树结点的容量较大，增加结点的扇出度，降低树的高度，减少了缓存失配的机率，同时在结点内扩大索引结构来提高缓存操作效率。

4.2.2　插入性能　在建立的 B + 树、T 树和 MCTH 树索引上进行插入操作，实验结果如表 2 所示。可以看出，索引数量较少时进行的插入操作，T 树效率最高，耗时最少，MCTH 树则没有表现出优势，这是因为 MCTH 树结构相比于 T 树要复杂，遍历的时间增加；但是当索引数量很大时，MCTH 树的优势就比较明显了，这是因为与 T 树、B + 树相比，MCTH 树在插入时要更新结点内索引以及记录地址数组，如果记录量很少，这些额外开销要大于结点内索引定位时带来的开销，所以速度稍慢。然而，当记录量很大时，MCTH 树利用结点内索引定位减少的开销弥补了更新它所造成的开销，所以 MCTH 树插入速度最快。

表 2　插入效率对比

Table 2　Insert efficiency comparison

索引数量/条	插入耗时/ms		
	B + 树	T 树	MCTH 树
10 000	87	73	77
20 000	210	198	172
50 000	629	588	531
100 000	1 411	1 402	979
500 000	7 930	8 127	6 519

4.2.3　删除性能　在建立的 B + 树、T 树和 MCTH 树索引上进行删除操作，实验结果如表 3 所示。删除操作与插入过程类似，当记录量较少时进行删除

表 3　删除效率对比

Table 3　Delete efficiency comparison

索引数量/条	删除耗时/ms		
	B + 树	T 树	MCTH 树
10 000	143	139	157
20 000	319	318	329
50 000	857	871	809
100 000	1 879	1 930	1 231
500 000	10 831	11 056	8 907

操作，MCTH 树树删除速度要比 T 树慢；当记录量很大时，MCTH 树的删除速度要比 T 树快。这是因为与 T 树、B + 树相比，MCTH 树在删除时要更新结点内索引以及记录地址数组，如果记录量较少，这些额外开销要大于结点内索引在定位时带来的开销，所以 MCTH 树速度慢。然而如果记录量很大，MCTH 树利用结点索引定位减少开销弥补了更新所造成的开销，所以 MCTH 树速度要快很多。

5 结束语

针对传统的内存数据库缓存索引性能的不足，提出了一个新的内存数据库索引结构 MCTH 树，给出了查询、插入和删除等操作算法的描述，最后通过实验测试，证明了 MCTH 树索引结构与 T 树、B + 树相比较，在插入、查询、删除等 3 个方面的工作效率上都具有优越性。

参考文献：

[1] 袁培森，皮德常. 用于内存数据库的 Hash 索引的设计与实现 [J]. 计算机工程，2007，32 (18)：69 - 71.

[2] Hankins R，Patel J. Effect of node size on the performance of cache conscious B + trees [C] //Proceedings of SIGMETRICS03. New York：ACM，2003：283 - 294.

[3] 钟宝荣，袁文亮. 内存数据库中存储结构的实现机制 [J]. 计算机工程与设计，2007，28 (5)：1213 - 1214.

[4] 马洪连，杨波，姚建伟. 一个适用于内存数据库系统的多维索引结构 [J]. 计算机工程与应用，2003，39 (29)：211 - 213.

[5] 杨友俊，王轫. 实时内存数据库存储结构及数据安置策略 [J]. 软件导刊，2006 (19)：37 - 39.

[6] Sun Li-mei，Song Bao-yan，Yu Ya-xin，et al . Cache-conscious index mechanism for main-memory databases [J]. Wuhan University Journal of Natural Sciences，2006，11 (1)：309 - 312.

[7] Graefe G. B-tree indexes，interpolation search，and skew [C] //Proceedings of the 2nd International Workshop on Data Management on New Hardware. New York：ACM，2006：466 - 475.

[8] 孙志宏，吴明成，谢会玲，等. 基于 C + + 的实时数据库的设计与实现 [J]. 三峡大学学报，2004，26 (6)：540 - 543.

[9] 龙玉国，耿协坚. 嵌入式实时内存数据库恢复子系统的设计 [J]. 计算机工程与应用，2004，21 (6)：41 - 43.

[10] 曹阳，方强，王国仁，等. 基于遗传算法的多连接表达式并行查询优化 [J]. 软件学报，2002 (2)：250 - 257.

Index Structure Supporting Cache Optimization in Main Memory Database

ZHANG Shao-xian

(*School of Computer*，*Nanjing University of Posts and Telecommunication*，*Nanjing* 210046，*China*)

Abstract：Aiming at the cache mismatch problem of index in main memory database，a MCTH-tree index structure is proposes based on cache optimization. It gives attention to the effects of cache misses and TLB misses. According to index structure analysis，it is concluded that MCTH-tree supports cache optimization. Detailed description on data structure and operation algorithm are given. A series of experiments show that MCTH-tree is verified to have better effectiveness and superiority compared with the relevant index structure.

Key words：main memory database；MCTH-tree；index；cache optimization