第 35 卷 第 5 期 计 算 机 工 程 2009 年 3 月
Vol.35 No.5 Computer Engineering March 2009
·软件技术与数据库· 文章编号：1000—3428(2009)05—0025—03 文献标识码：A 中图分类号：TP311.12

一种处理 B^+树重复键值的方法

徐逸文[1,2]，方 钰[1,2]，陈闵中[1,2]
(1. 同济大学计算机科学与技术系，上海 201804；2. 同济大学嵌入式系统与服务计算教育部重点实验室，上海 201804)

摘 要：当前 B^+树索引结构常采用“溢出页”处理重复出现的键值，易于实现，但存在严重的空间浪费。该文通过改进索引结构的定义，使 B^+树能够显式支持重复键值，并给出相应算法。实验证明，采用该方法的 B^+树能有效减少索引文件的大小，使 B^+树满足更多数据库索引的需求。

关键词：B^+树；重复键值；溢出页

Approach for Handling Duplicate Keys in B^+Tree

XU Yi-wen[1,2], FANG Yu[1,2], CHEN Hong-zhong[1,2]
(1. Department of Computer Science and Technology, Tongji University, Shanghai 201804;
2. Key Lab of Embedded System and Service Computing, Ministry of Education, Tongji University, Shanghai 201804)

【Abstract】 Overflow page is commonly used to deal with the duplicate keys of a B^+ tree index, although it is easy to implement, it introduces great space waste. This paper adopts an alternative approach to make the B^+ tree explicitly support the duplicate keys by revising the index structure, and provides the basic algorithms to implement the idea. Experiment indicates that the revised B^+ tree can significantly reduce the size of the index file, which broadens the application area of B^+ tree.

【Key words】 B^+ tree; duplicate keys; overflow page

B 树[1]可以看作是“二叉搜索树”在外存中多路扩展的一般形式，它被设计用来管理和维护大规模数据的索引，具有随机查询效率高、更新开销小和自平衡等特点。B^+树[2]在 B 树的基础上，进一步规定所有的“键值”只能出现在叶子节点中，并用链表的方式把所有叶子节点串连在一起，提高了顺序查询的效率。随着信息数据量的日益增加，数据库系统对磁盘空间的要求也越来越高，这一点在以移动设备为载体的嵌入式数据库中尤为突出[3]：有限的存储空间既要存放大量的数据表文件，又要存放大量与数据表对应的索引文件。因此，为了让 B^+树索引的适用范围更广，应尽可能减小 B^+树索引空间复杂度。

1 “溢出页”方法

为了简化问题，B^+树算法的研究一般都假设不存在键值重复的情况[4]。现有处理重复键值的方法主要采用“溢出页”[5]：当某个数据键值对应的记录数大于 1 时，分配一个“溢出页”用来存放所有的重复键值及其对应记录的偏移量，如图 1 所示。

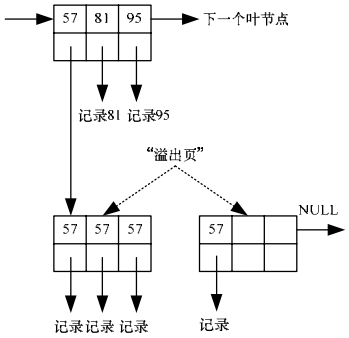

图 1 “溢出页”方法

“溢出页”方法尽管简单，但不可避免地会浪费溢出页中空闲的空间，尤其是当数据键值所占空间相对于溢出页容量较小时，空间浪费的程度加剧，整个索引文件对磁盘的需求量随之明显增加。

2 改进的索引结构及其算法

2.1 索引键值的重新定义

对于经典 B^+树索引结构，中间节点的索引键值仅表示它包含键值范围的一个下界，不能有效地表示索引结构中键值重复的情况。所以，为了使 B^+树能够支持重复键值，可以适当修改中间节点中索引键值的定义。

定义 1 中间节点中第 k 个索引键值是以它第 k+1 个子节点为根的子树中第 1 次出现的“新”数据键值，其中，k 从 0 开始。

按照上述定义，该“新”数据键值在其左边的叶子节点中不存在。一种特殊情况需要注意，当子树包含的相同键值已经出现在左边叶节点中时，称“新”(数据)键值不存在，应将该索引值定义为“空”(NULL)。

2.2 额外索引键值的增加

因为中间节点的儿子节点指针比索引键值多一个，所以为了让索引结构更完整对称，本文增加了一个额外索引值，对应于以第 0 个子节点为根的子树中包含的第 1 次出现的新

基金项目：上海市国际科技合作基金资助项目“智能交通信息服务技术及移动终端导航系统”(062107037)；上海市国际科技合作基金资助项目“车辆网络感知导航的关键技术与高性能支撑平台”(075107005)

作者简介：徐逸文(1983－)，男，硕士研究生，主研方向：空间数据库；方 钰，讲师、博士；陈闵中，教授、博士生导师

收稿日期：2008-10-08 **E-mail：**michaelxumail-nor@yahoo.com.cn

数据键值。考虑到原先的索引键值的序号从 0 开始，把这个额外的索引键值称为第“ -1 ”个索引键值。通过这个额外的索引键值能够大大简化 B^+树算法的设计。

一个采用上述定义的 B^+树的例子如图 2 所示。可以看出，因为 6 号叶节点中的所有键值 9 已经在之前的节点中出现过，所以第 1 次出现的新键值不存在，用“ # ”表示该索引键值为“ 空 ”。另外，图中用下划线标注的数字表示每个非叶子节点的额外键值，例如，因为 8 号叶子节点中第 1 次出现的新键值为 25，而它又是 3 号节点的第 0 个儿子节点，所以 3 号节点的额外索引值为 25。图中其他的结构和元素与经典 B^+树相同。

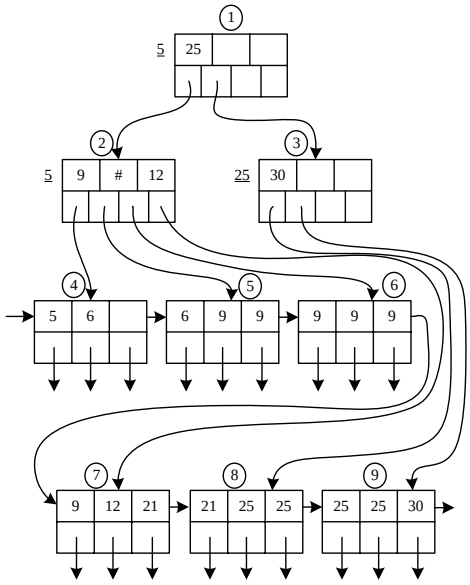

图 2　B^+树索引结构

从中可以进一步发现，各层次中间节点的索引键值之间存在一种关系，可以由如下性质描述：

性质 在改进 B^+树索引结构中，当某个中间节点的子节点不是叶节点，则此中间节点的第 k 个索引键值等于其对应子中间节点中从小到大第 1 个非空的索引键值；如果子中间节点中所有的索引键值均为空，则第 k 个索引键值也为空，其中，k 从-1 开始。

例如，图 2 中的 1 号根节点满足上述性质的要求，所以，其第-1 个索引键值等于 2 号节点中最小索引键值 5；第 0 个索引键值等于 3 号节点中最小的索引键值 25。该性质可以极大地简化改进 B^+树基本算法的设计。

2.3　查询

针对上述重新定义的 B^+树索引结构，修改经典算法得到如下查询算法：

Algorithm offset SEARCH(node, key)

(1)如果 node 是叶节点，则在 node 和其右邻居中顺序查找 key，如果找到则返回 key 对应的偏移量。

(2)如果 node 是中间节点，逐个比较索引键值与 key 的大小，确定下一步遍历的子节点；如果索引键值不为空且小于等于 key，则下一步遍历位置加 1。

(3)读入下一步遍历位置的子节点，并递归调用 SEARCH。

改进 B^+树查询算法的特点是，增加了处理“ 空 ”索引键值的情况。根据 SEARCH 算法的第(2)步：只有在当前索引键值不为空且小于等于待查询键值时，下一步遍历子节点的位置才加 1。从算法 SEARCH 可以看出，对 B^+树的改进没有增加查询的复杂度。

2.4　插入

改进 B^+树索引结构的插入算法与经典算法的思路基本相同。唯一的区别是由于改变了 B^+树索引键值的定义，因此插入操作不仅会影响待插入叶子节点与其父节点 2 个节点中的索引键值，还可能影响再上层父节点中的索引键值。为了保持 B^+树索引结构的完整性，本文增加了 UPDATE_INDEX 操作，按需更新查询路径向上每个节点的索引键值。

Algorithm UPDATE_INDEX (node,oldKey,newKey,pos)

(1)如果 node 是根节点，则更新结束，返回。

(2)得到 node 在其父节点中对应的索引键值 keyInParent。

(3)如果 keyInParent 等于 oldKey：

1)如果 newKey 为空，则更新为在 node 中从 pos+1 位置开始第 1 个不为空的索引键值，如果找不到，则更新为空；否则，更新为 newKey；

2)递归向上更新 UPDATE_INDEX。

(4)如果 oldKey 为空并且 newKey 小于 keyInParent，则递归向上更新 UPDATE_INDEX。

图 3 和图 4 是在图 2 中插入键值 29 后，引起的先分裂 9 号叶节点然后更新 3 号节点索引键值的例子。

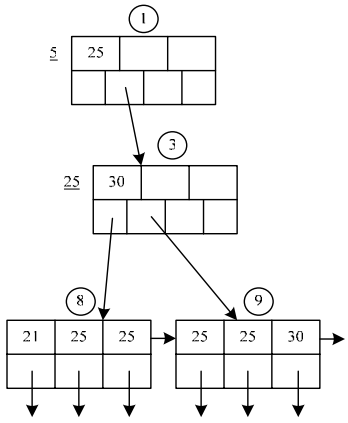

图 3　分裂 9 号叶节点前的情形

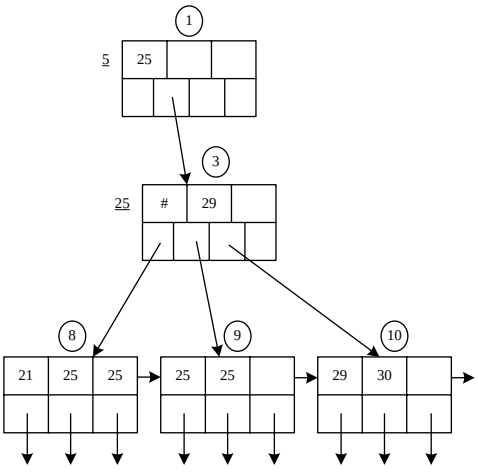

图 4　分裂 9 号叶节点且插入键值 29 后的情形

由图 3 可以看出，由于需要插入键值 29，原来包含键值 25, 25 和 30 的 9 号叶节点首先被分裂成 2 个叶节点：其中一个包含 2 个键值 25；另一个包含键值 30。节点分裂完成后，

需要在 9 号节点的父节点 3 号节点中增加一个索引键值对应由分裂产生的 10 号节点，还要判断对应 9 号节点的索引键值是否需要更新。

图 4 显示，因为 9 号节点包含的键值 25 已在 8 号节点中出现过，所以父节点中的索引键值应该从 30 修改为#(“ 空 ”)，表示第 1 个子节点中不包含新出现的键值。而且，因为 9 号节点中的第 1 个非空索引键值 25 没有改变，所以由性质可以推得，无须再更新父节点(1 号节点)。

2.5 删除

与插入算法类似，删除操作可能会影响待删除节点与其父节点之外节点中的索引值，所以，在删除操作完成后，对节点进行 UPDATE_INDEX 操作。删除算法的其余过程与经典算法相同，这里不再给出。

事实上，删除操作与插入操作互为逆过程。删除键值 29 前后，改进 B^+树索引结构的变化同样可以参照图 3 和图 4。

3 实验及性能分析

本实验用“ 键值比重 ”考察 B^+树索引结构中键值的存储情况，定义如下：

定义 2 一个重复出现键值的“ 键值比重 ”等于它重复的个数与 B^+树节点最大度的比值：

KR=键值重复次数÷B^+数最大度

由上式不难看出，“ 键值比重 ”由 3 参数决定：节点大小，键值大小及键值重复次数。键值比重反映了当前 B^+树中重复键值对索引文件大小的影响程度，键值比重越大则影响越大。为了简化问题，本实验规定每个重复键值的重复个数均相同。

本实验按如下步骤进行：

(1)插入 10 000 个不重复的键值。

(2)插入一定比例(10%, 20%, 25%, 33%, 50%)的重复键值，每个键值重复相同的次数，并考察键值比重为 6%, 11% 和 24% 3 种情况。

(3)记录 B^+树索引文件大小。

实验结果如表 1 和图 5~图 7 所示。

表 1 索引文件大小比较

插入的重复键值/(%)	*KR*=6%		*KR*=11%		*KR*=24%	
	改进 B^+树/Byte	经典 B^+树/Byte	改进 B^+树/Byte	经典 B^+树/Byte	改进 B^+树/Byte	经典 B^+树/Byte
10	640 000	1 628 160	640 000	1 628 160	641 024	1 628 160
20	640 000	2 652 160	641 024	2 652 160	1 280 000	2 652 160
25	640 000	3 164 160	792 576	3 164 160	1 430 528	3 164 160
33	641 024	4 018 176	1278 976	4 018 176	1 917 952	4 018 176
50	943 104	5 724 160	1581 056	5 724 160	2 856 960	5 724 160

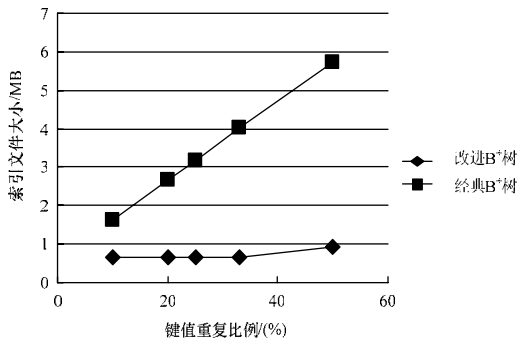

图 5 索引文件大小比较(*KR*=6%)

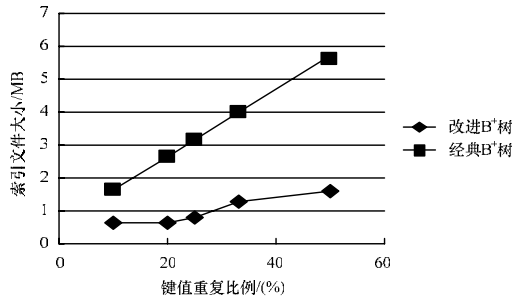

图 6 索引文件大小比较(*KR*=11%)

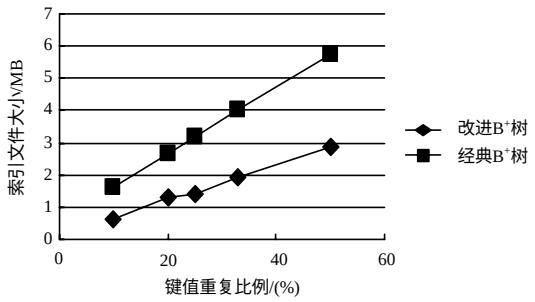

图 7 索引文件大小比较(*KR*=24%)

从实验结果可以看出，在 3 种不同的“ 键值比重 ”情况下，本文的 B^+树索引文件大小均明显小于采用溢出页的 B^+树，这是因为当 B^+树中存在重复键值时，溢出页中未使用的空间会导致空间资源的浪费，随着重复键值的比例上升，空间浪费的程度在加剧，改进 B^+树节省的空间也越多。同时，由于 3 种“ 键值比重 ”都未超过 100%，即重复键值大小之和小于一个节点空间，因此经典 B^+树在这 3 种情况下消耗的磁盘空间大小相同；相反，本文改进型 B^+树消耗的磁盘空间随着“ 键值比重 ”的增加而增加，表明其适用于“ 键值比重 ”较小而键值重复比例较大的情况。尽管查询算法并不引入额外的 I/O 开销，但由于本文的 B^+树的更新操作需要处理重复键值，因此在总体上需要更多的 I/O 操作。这主要体现在：与采用溢出页的 B^+树相比，本文的改进 B^+树额外读磁盘的总数比较多，而写磁盘总数略高。不过，因为额外读磁盘的开销主要集中在一条查询路径上的上下遍历，所以利用 B^+树更新操作的空间局部性，使用磁盘缓存技术可以有效减少 I/O 的操作次数。

4 结束语

本文通过改进 B^+树的数据结构及相应算法，得到一种处理索引中出现的重复键值的方法。实验证明，该方法能够有效减小索引文件大小，使 B^+树满足更多数据库的索引需求。

参考文献

[1] Bayer R, McCreight C. Organization and Maintenance of Large Ordered Indexes[J]. Acta Information, 1972, 1(3): 173-189.

[2] Comer D. The Ubiquitous B Tree[J]. Computer Surveys, 1979, 11(2): 121-147.

[3] Ortiz S. Embedded Databases Come out of Hiding[J]. IEEE Computer, 2000, 33(3): 16-19.

[4] Garcia-Molina H, Ullman J D, Widom J D. Database Systems: The Complete Book[M]. [S. l.]: Prentice Hall, 1999.

[5] Ramakrishnan R, Gehrke J. Database Management Systems[M]. Singapore: McGraw-Hill, 2002.

编辑 张 帆