高速缓存敏感 B+树的设计与实现

学　　院 计算机科学与技术

专　　业 计算机科学与技术

年　　级 2003 级

姓　　名 包武

指导教师 张坤龙

2007 年 6 月 15 日

摘　要

随着主存的价格逐渐降低，主存数据库的应用变得越来越广泛。但是，由于CPU和内存的发展速度极不平衡，两者的运行速度的差异日益增大。为了解决这个问题，人们在CPU和内存之间增加了一个高速缓存层。要想使主存数据库的运行速度得到优化，就必须使它采用的数据结构有更好的高速缓存行为。获得好的高速缓存行为最主要的方法就是用指针消除技术来降低数据结构的高速缓存缺失, 提高数据结构高速缓存的敏感性。

论文设计和实现了一种索引结构，称作高速缓存敏感B+树(CSB+树)。CSB+树在每个缓存块中能存储更多的关键字，因此比B+树更加高速缓存感知。CSB+树必须保持兄弟结点是相邻存储的特性，甚至在增量更新的时候也是一样。论文对B+树、CSB+树、以及CSB+树的两个变种（段式CSB+树和满CSB+树）进行了分析比较，而且用实验的方式对分析结论进行了验证。最终结果表明CSB+树有着比B+树更加优越的性能，因为它有着更好的高速缓存行为。

关键词：高速缓存；指针消除；CSB+树

目　录

第一章　绪论…… 1
1.1 高速缓存和指针消除技术 …… 1
1.2 B+树 …… 2
1.2.1　B+树的概念 …… 2
1.2.2　B+树上的操作 …… 3
1.3 索引树的发展 …… 3
1.4 论文安排 …… 4
第二章　CSB+树…… 5
2.1 CSB+树的结构特点…… 5
2.2 CSB+树的数据结构…… 5
2.3 CSB+树上的操作…… 6
2.3.1　批量装载 …… 6
2.3.2　查找 …… 8
2.3.3　插入 …… 9
2.3.4　删除 …… 9
2.4 满的 CSB+树 …… 12
2.5 分段的 CSB+树 …… 12
第三章　性能分析…… 14
3.1 查找性能分析…… 14
3.2 分裂性能分析…… 15
3.3 空间需求分析…… 15

第四章　实验结果…… 17
4.1 测试方法…… 17
4.2 实验结果…… 17
4.3 实验结论…… 22
第五章　结论…… 23
参考文献…… 24
致谢

第一章 绪论

1.1 高速缓存和指针消除技术

随着主存的价格逐渐降低，主存数据库的应用变得越来越广泛。但是，由于CPU和内存的发展速度极不平衡，两者的运行速度的差异日益增大。为了解决这个问题，人们在CPU和内存之间增加了一个高速缓存层。要想使计算机的运行速度得到优化，就必须使计算机有更好的高速缓存行为。而要获得好的高速缓存行为最主要的方法就是用指针消除技术来降低高速缓存缺失, 提高高速缓存的敏感性 [1]。

高速缓存是很小、很快的静态 RAM 存储器，它通过保存最近用到的数据来提高程序的性能。一个高速缓存可以用容量、高速缓存行的大小和高速缓存行的个数来进行参数化，容量就是高速缓存的大小，高速缓存行的大小就是在主存和缓存间进行传递的数据单元，高速缓存行的个数就是缓存中最多可以容纳高速缓存行的个数，典型的缓存行可以从 32 字节到 128 字节。当前的高速缓存一般用 SRAM 来实现，存储容量一般在几十 KB 到几 MB。

存储器需要的数据被高速缓存满足时叫做命中，不被满足的叫做缺失，缺失时就必须从主存中取得相应的缓存模块。现在的计算机体系结构中通常有两种高速缓存，L1 和 L2，它们处于 CPU 和主存之间。在 L1 可以满足 CPU 的数据存取需求的时候，L2 和主存通常分别按顺序导入 10 或 100 个时钟周期的最可能用到的数据。高速缓存只有在要求的数据可以在缓存中找到的时候才能降低因访问存储器而造成的延迟。这主要依赖于应用程序对存储器的访问模式。因此，除非采用特殊的措施，否则存储器的延迟将成为提高系统性能的瓶颈，阻止我们对现代硬件的充分利用[2]。

高速缓存的速度大概是主存速度的 3 到 10 倍。下面以 10 倍速度的高速缓存为例来看一下一个高速缓存缺失会造成什么影响，假设进行 10 次读写操作，从高速缓存中读写的时间为 t，从主存中读写的时间为 10t。当 10 次读写操作全部命中的话，所用时间为 10t。当其中有一次缓存缺失的时候，所用的时间是 10t+9t=19t，时间消耗比全部命中的时间消耗多了近一倍。当缓存缺失更多的时候，将耗费更多的时间来进行数据的读写操作[3]。

指针消除是一项在缓存优化方面很重要的技术，因为它增加了高速缓存行的利用率。在传统的索引结构中，指针占用了很大的空间。相对地，一个高速缓存行所能容纳的关键字的个数就变少了，高速缓存的缺失率就会比较高。如果能将这些指针消除，就可以释放出很大一部分空间存储关键字，这样一个高速缓存行

就可以容纳更多的关键字，可以使更多的数据存储到高速缓存中，从而降低高速缓存缺失率，提高机器运行的性能。指针的消除取决于关键字的大小。如果关键字的大小远大于指针的大小，则可能降低指针消除的影响。如果出现关键字的大小远大于指针的大小情况，可以把全部的关键字值映射成一定范围内的特定的ID值，并且仅存储这个ID值，从而减少关键字的大小。现在已经有人实现了64位的操作系统，这意味着每个指针将有8位，而不是4位，这意味着指针将比数据占用更多的空间。所以指针的消除在将来将会更加地重要。完全地消除指针会引入一系列的约束。比如，PLI树要求数据要按顺序插入[4]，CSS树和V树[5]不能很好的进行增量更新。本文在高速缓存敏感B+树（CSB+树）中采用的是局部的指针消除技术。这样，在优化缓存行为的同时，避免了引入新的约束。

1.2 B+树

本文所要讨论的CSB+树是B+树的一个变种，因此下面先介绍一下B+树的结构特点。

1.2.1 B+树的概念

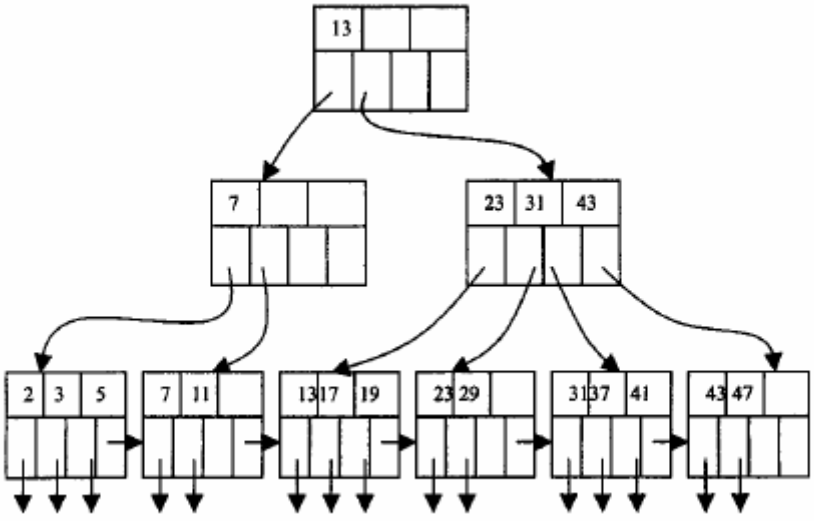

图1-1　一个简单的B+树结构

1970年，R.Bayer和E.mccreight提出了一种适用于外存查找的树，它是一种平衡的多叉树，称为B树，其定义如下：

一棵m阶的B树满足下列条件：

⑴ 树中每个结点至多有m个孩子；

⑵ 除根结点和叶结点外，其他每个结点至少有m/2个孩子；

⑶ 若根结点不是叶结点，则至少有2个孩子；

⑷ 所有叶结点都出现在同一层；

⑸ 有k个孩子的非终端结点恰好包含有k-1个关键字。

在B树中，每个结点中关键字从小到大排列，并且当该结点的孩子是非叶结点时，该k-1个关键字正好是k个孩子包含的关键字的值域的分划。

B+树是B树的一种变形树，它与B树的差异在于：

⑴有k个子结点的结点必然有k个关键字；

⑵非叶结点仅具有索引作用，跟记录有关的信息均存放在叶结点中 [6]。

图1-1显示的就是一个简单的B+树的例子。

1.2.2 B+树上的操作

⑴ B+树的查找

由于跟记录有关的信息存放在叶结点中，查找时若在上层已找到待查的关键字，并不停止，而是继续沿指针向下一直查到叶结点层的关键字。此外，B+树的所有叶结点构成一个有序链表，可以按照关键字排序的次序遍历全部记录。上面两种方式结合起来，使得 B+树非常适合范围检索。

⑵ B+树的插入

B+树的插入在叶结点上进行，如果叶结点中的关键字个数超过 m（m 为一个结点所能容纳的最大的关键字个数），就必须分裂成关键码数目大致相同的两个结点，并保证上层结点中有这两个结点的最大关键字。

⑶ B+树的删除

B+树中的关键字在叶结点层删除后，其在上层的副本可以保留，作为一个"分界关键字"存在，如果因为删除而造成结点中关键码数小于 m/2 ，对树的结构进行调整，保证 50%以上的利用率[7]。

B+树不只适合随机检索，B+树同时支持随机检索和顺序检索，在实际中应用比较多[8]。

1.3 索引树的发展

二叉树：二叉树(BinaryTree)是 n(n≥0)个结点的有限集，它或者是空集(n=0)，或者由一个根结点及两棵互不相交的、分别称作这个根的左子树和右子树的二叉树组成。它查找的时候采用的折半查找，查找的速度比传统的逐个比较的方式有了很大提升。

B+树：这种树在上一节中已经详细介绍过了，这里不再重复。

T树：T树 [9]是 To binJ .Lehman和MichaelJ. Carey在1986年提出的一种内存数据库的索引结构，现在T树已经应用于多个内存数据库中，包括:Dali, Starburst, fastdb等，已经成为内存数据库中最主要的一种索引方式。它是AVL树(平衡二叉

树)的一个变种，吸收了AVL树和B树的优点。一般来说，T树可以定义为一棵在一个结点内包含多个关键词的平衡二叉树。

CSS 树：在文献 [RR99][10]中提出了一种新的索引结构，命名为 CSS 树。它有比 B+树还要好的高速缓存行为。CSS 树对每个高速缓存行有更好的利用。虽然这种方法改进了寻找的速度，但是它不支持增量更新，这在很大程度上限制了 CSS 树在实际中的应用。

PLI树：在文献[TMJ98][11]中提出了一种指针较少的插入树（PLI树）。PLI树是B树的一种变种，它按一定的顺序分配结点的存储空间，因此它的子结点可以通过数学计算来找到，因此它不用明确的储存子指针。PLI树只能进行添加，比如订货，这种情况下数据总是按处理的时间先后顺序进行添加的。所有的添加应该都在最右边的叶结点进行，而不会有不规则的插入发生。

V树：在[Ker89][[12]中，作者把一个二叉树用非传统的方法存储成一个队列，把这种结构称为V树。V树采用一种简单的查找方式，它不是采用指针的方式来查找，而是把查找的信息隐藏起来，通过一定量的数学计算来找到要查找的关键字的位置。虽然它有很好的寻找的性能，但它索引数据的数量存在一个上界。在存储索引的区域快满的时候，维护开销开始恶化。

CST树[13]：最近通过对T树的改进，又衍生出一种新的树——CST树。CST树在T树的基础上增加了一个数组，虽然给插入和删除时维护带来了一定的复杂性，但是它给查找定位节省了大量的时间，减轻了缓存的延迟，充分发挥了CPU的效率。

CSB+树：全名是高速缓存敏感B+树，也就是本文要讨论的主要内容。这种树用几个结点构成一个结点组，把在同一个结点组的结点连续存储在一起，这样每个结点只需要存储一个子指针。通过消除了大量的指针，CSB+树提高了索引的性能。

1.4 论文的结构

在第二章将介绍高速敏感B+树的存储结构和操作。第三章将分析比较B+树、高速缓存敏感B+树、段式高速缓存敏感B+树、满高速缓存敏感B+树四种数据结构的优缺点。第四章用实验的方法来比较它们的优缺点。第五章做全面的总结。

第二章 CSB+树

2.1 CSB+树的结构特点

在这里，将介绍一种新的索引结构，叫做CSB+树。一个CSB+树有着和B+树类似的结构。但是它不是把所有的子指针都明确的存储起来，而是把子结点存储在给定的连续的空间中，仅仅记录第一个子结点的指针。通过在头指针上加上一个偏移量的方法找到其他结点。这种结构对高速缓存行有很好的利用。另外，CSB+树还可以支持类似于B+树的增量更新。

CSB+树必须保持兄弟结点是相邻存储的特性，甚至在增量更新的时候也是一样。这样的一组兄弟结点叫做一个结点组。有很多方法保持结点组是相邻的，但这些方法在分裂过程中都要进行结点的复制。本文将展示CSB+树的几种不同的变种，解释它们是怎样完成邻近特性的保持的。最简单的方法是先分配1个新的更大一点的结点组，把原结点组的数据复制过来，最后原结点组释放自身存储单元，同时。段式CSB+树通过只复制结点组中相应的段，降低了更新的开销。满CSB+树分配额外的空间给结点组，使存储器的管理变的简单，复制操作变的开销更小。

满CSB+树相对于B+树，无论是更新增量数据，还是寻找数据都占有很大优势，虽然它会花费多一点的空间。另一方面，当工作中有大量的寻找和很少的更新时，CSB+树占用的空间比B+树要少的多，同时表现得比B+树要好。

创建CSB+树的目的就是要得到更好的高速缓存性能，同时又能够进行B+树的增量更新。完成这个目标是通过平衡两种索引结构的优点来达到的。CSB+树处理更新的方法和B+树类似。但是CSB+树有着比B+树少的指针，通过在每个结点上采用较少的指针，可以为关键字留下更多的空间，得到更好的高速缓存性能。

下面介绍一下本章的安排。在2.2中将介绍一个结点只有一个指针的CSB+树，本文简单地用CSB+树来表示。在2.3中，将介绍这种树是如何进行批量装载、查找、插入和删除等操作的。在2.4中，将描述一种具有较多指针结点的CSB+树变种，它每个结点指针的数量是一个参量，可以使系统在运行特定的工作的时候获得很好的性能。在2.5中将介绍CSB树的另一个变种，它可以进一步节省分裂所需要的开销。

2.2 CSB+树的数据结构

CSB+树是一种平衡多路查询树，它的每个结点包含m个关键字，其中d<=m<=2d。CSB+树把每个结点的所有子结点都放进一个结点组中。这些被放进

结点组的结点连续地存储在一起，可以通过偏移地址和首地址进行运算得到的地址进行访问。在CSB+中的每个结点都有下面的构造

num：在结点中关键字的个数。

fc：指向第一个子结点的指针。

a[2d]：所有的关键字的列表。

（每个叶结点存储一些<key，id>对，和这些数据对的数量，以及两个兄弟指针。）

下面是程序中对内部接点和叶子接点的定义：

```
#define  maxlen  14//内部结点中关键字的最大个数
#define  lmaxlen 6//叶子接点中关键字的最大个数
struct    node//定义结点
{
    struct    node     *fc;//第一个子结点指针
    int  num;//关键字个数
    int  a[maxlen];//关键字
};
struct    data//定义数据
{
    int  key;//关键字
    int  id;//假定的数据，可根据实际情况对它进行修改，如果数据太大不便于
//复制到主存中，可以把这项改为数据的指针，再利用指针访问数据
};
struct    lnode//定义叶结点
{
    struct    lnode    *left;//左结点指针
    struct    lnode    *right;//左结点指针
    int  num;//关键字个数
    struct    datadata1[lmaxlen];//关键字
};
```

[14]

因为CSB+树的结点中只需要明确地存储一个子结点的指针，所以它可以存储比B+树更多结点。比如，假设一个结点和一个高速缓存行的大小都是64位，指针和关键字都需要4个字节，B+树每个结点只能存储7个关键字而CSB+树每个结点可以存储14个关键字。这使CSB+树的一个数据块可以满足比原来多一倍的比较，因此在比较时需要的高速缓存行变少了。同时因为CSB+树连续存储，结构比较紧密，所以它占用较少的空间来存储数据。

2.3 CSB+树上的操作

2.3.1 批量装载

批量装载操作的具体流程图见图 2-1。一种典型的创建 B+树的方法是保证所有的插入都在最右边的结点上进行。但是如果把这个方法用在 CSB+树上会开销很大，因为在同一个结点组的结点不是连续创建的，更有效的方法是逐级的创建索引结构。首先为所有的叶结点分配空间，然后计算更高等级所需要的结点数，分配一大块连续的空间给这个级别的结点。通过复制比它低一级结点中最大的数值来填充这些结点的数据。同时在高一级的结点中创建头子结点指针。重复这个过程，直到在高一级只有一个结点。把这个结点指定为根结点。

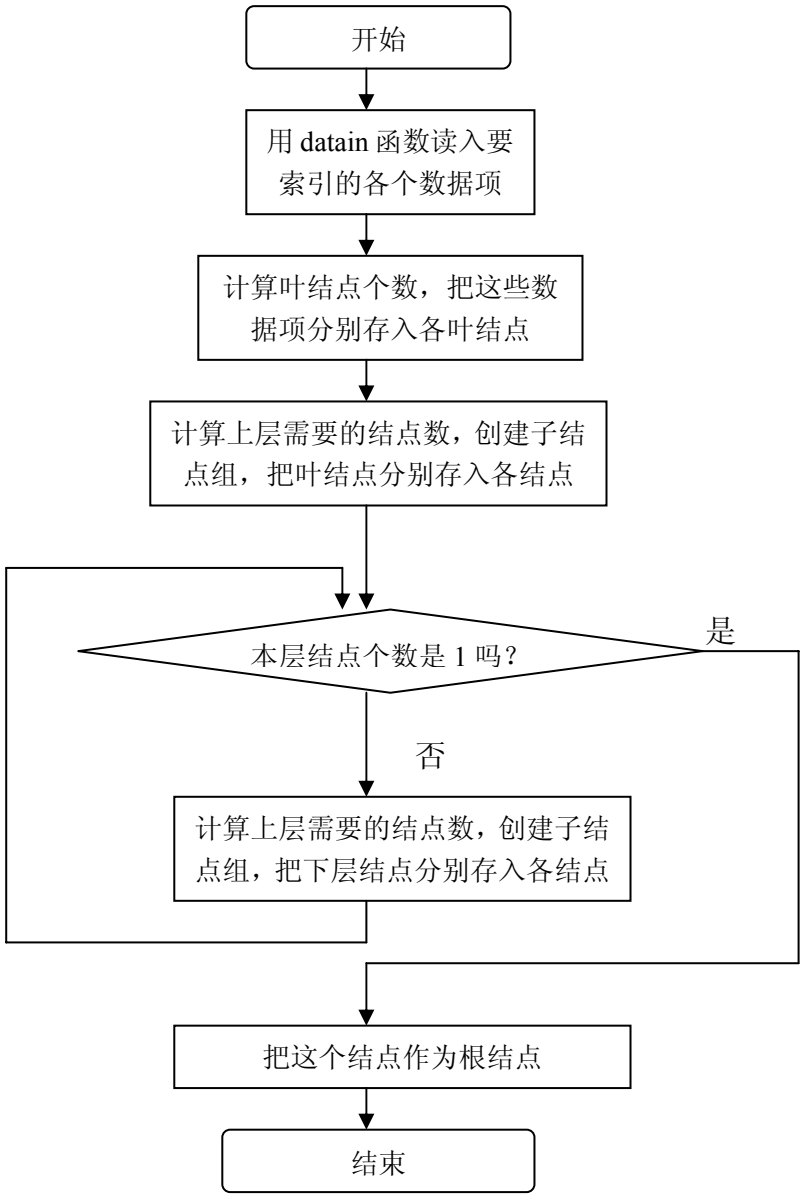

图2-1　批量装载程序流程图

2.3.2 查找

查找操作的具体流程图见图2-2。CSB+树搜索和B+树搜索很类似。当确定了不大于要查找的关键字的最大的关键字K时，只要简单的在头结点指针上加上K的偏移量（当要查找的关键字不大于最左边的关键字时，偏移量是0）。比如K是结点中的第5个数值，就可以用头指针加上5来找到相应的子结点。

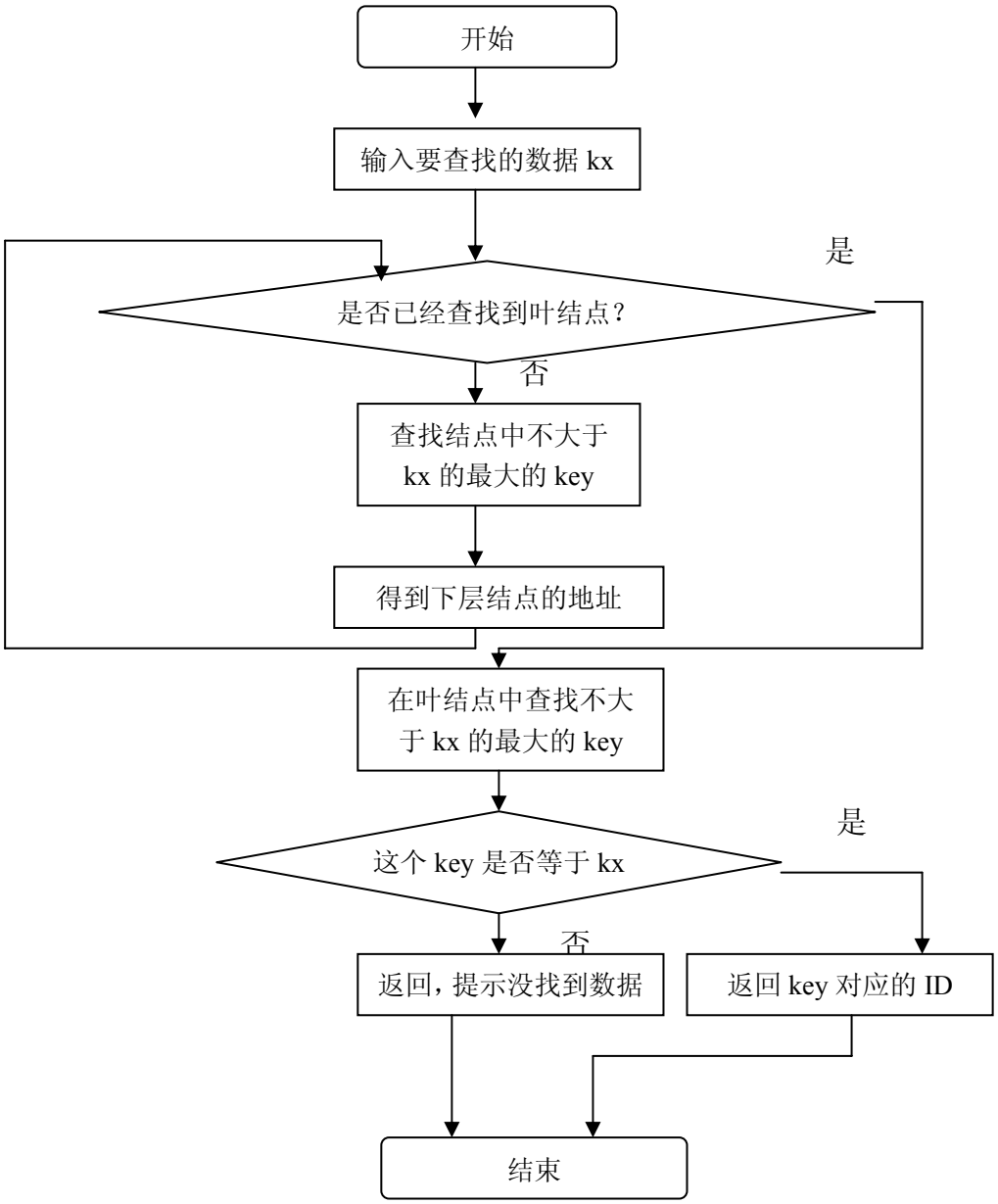

图 2-2　查找程序流程图

2.3.3 插入

插入操作的流程图见图2-3。插入同样和B+树的插入很类似，首先在树中查找要插入值的位置，找到后看结点中是否有足够的空间来存储这个值，如果足够就直接存储进这个结点，否则就要进行结点的分裂。

在叶结点进行分裂的时候，根据父结点p是否有足够的空间来存储一个新的关键字值，分裂可以分为两种。一种情况是假设有足够空间，这时令f是p中的头结点指针，g是f所指向的结点组，首先创建一个新的比g多一个结点的结点组g0，把g中所有的结点（插入关键字的结点分裂成两个结点）按顺序复制到g0中，把p的头指针f更新成指向g0，然后回收g。

还有一种更复杂的情况就是父亲结点p满了，在这种情况下，p本身也要分裂。同样，令f是p中的头结点指针，g是f所指向的结点组。在这种情况下要建立结点组g0，然后把g中的结点均匀分配到g和g0中，一半的关键字被转移到p0中（p0指向g0结点组），为了完成p到p+p0的转化，包含p的结点组需要用前一种情况时采用的方法进行复制，或者如果那个结点组也满了的话，就需要对父亲结点进行同样的操作。

在有一个分裂需要进行时，CSB+树需要创建一个结点组，而B+树只需要创建一个结点。因此当出现过多的分裂的时候，CSB+树会需要更大的开销。在另一方面，CSB+树可以更加快速地定位右边的结点。这意味着可以在结点组中预留更多的空间来减少复制[15]。

2.3.4 删除

删除操作的流程图见图2-4。删除可以采用和插入类似的方法来进行。但实际上，人们在执行删除的时候只是简单地在找到后删掉，而不进行相关的调整来保证50%以上的使用率。

本文采用的是和插入类似的方法。先删除相应的关键字，然后看结点中的关键字个数是不是少于d，是的话要做相应的调整。按照兄弟结点关键字个数是不是大于d，调整分为两种。

当兄弟结点关键字个数大于d的时候，结点就向兄弟结点借一个关键字，然后对两个结点的子结点组进行调整，以保证CSB+树的特性。

当兄弟结点关键字个数不大于d的时候，结点就和兄弟结点合并，然后在父亲结点中删除由于合并的减少的关键字，如果删除使父亲结点的关键字个数少于d了，那么就用同样的方法继续进行调整。

直接删除的方式虽然被大多数人所使用，不过采用这种方式不能保证结点50%以上的利用率，可以使查找的性能降低，所以本文没有采用直接删除的方式。

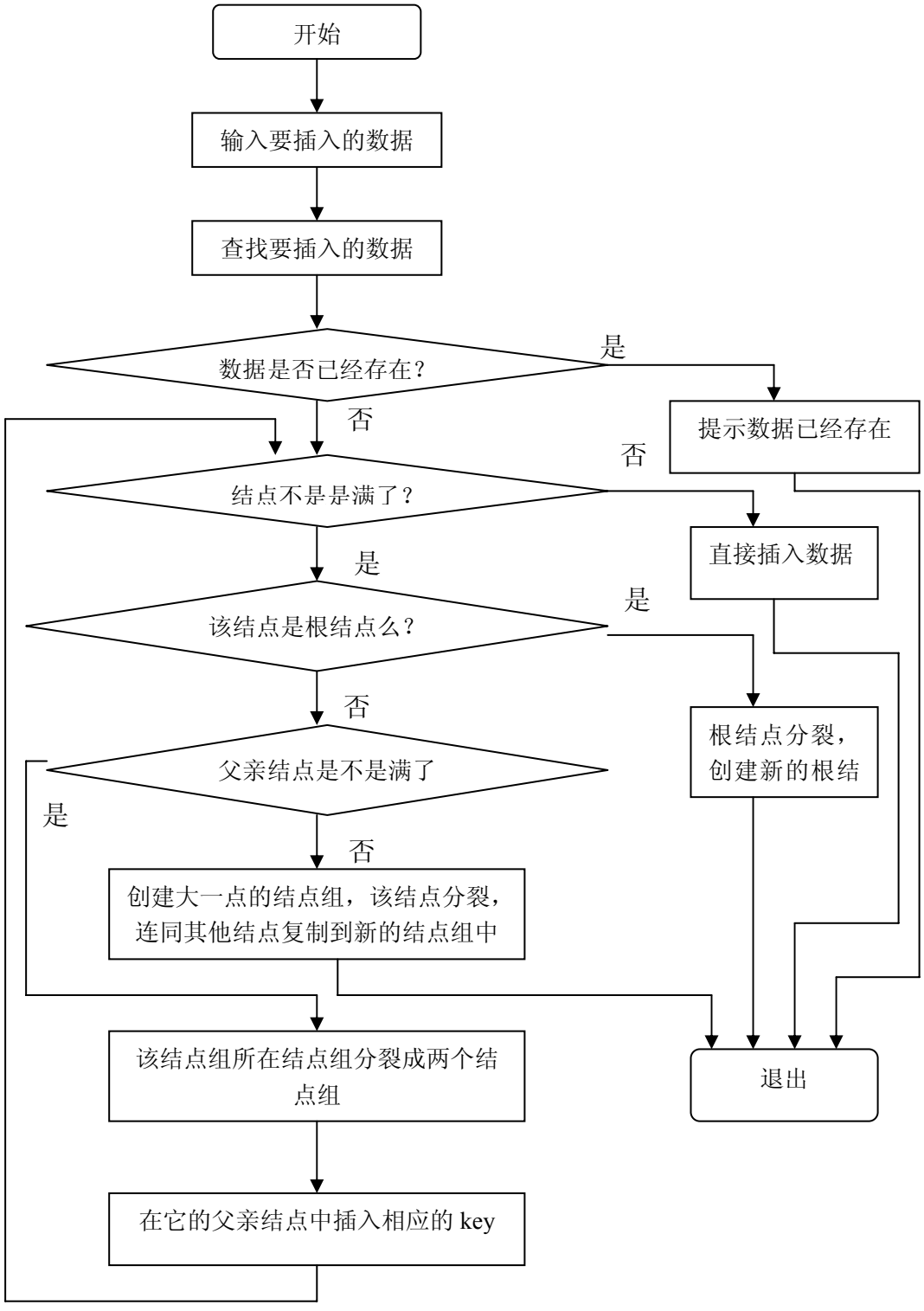
开始
输入要插入的数据
查找要插入的数据
数据是否已经存在？
是
提示数据已经存在
否
结点不是是满了？
否
直接插入数据
是
该结点是根结点么？
是
根结点分裂，
创建新的根结
否
父亲结点是不是满了
是
否
创建大一点的结点组，该结点分裂，
连同其他结点复制到新的结点组中
该结点组所在结点组分裂成两个结
点组
退出
在它的父亲结点中插入相应的 key

图 2-3　插入程序流程图

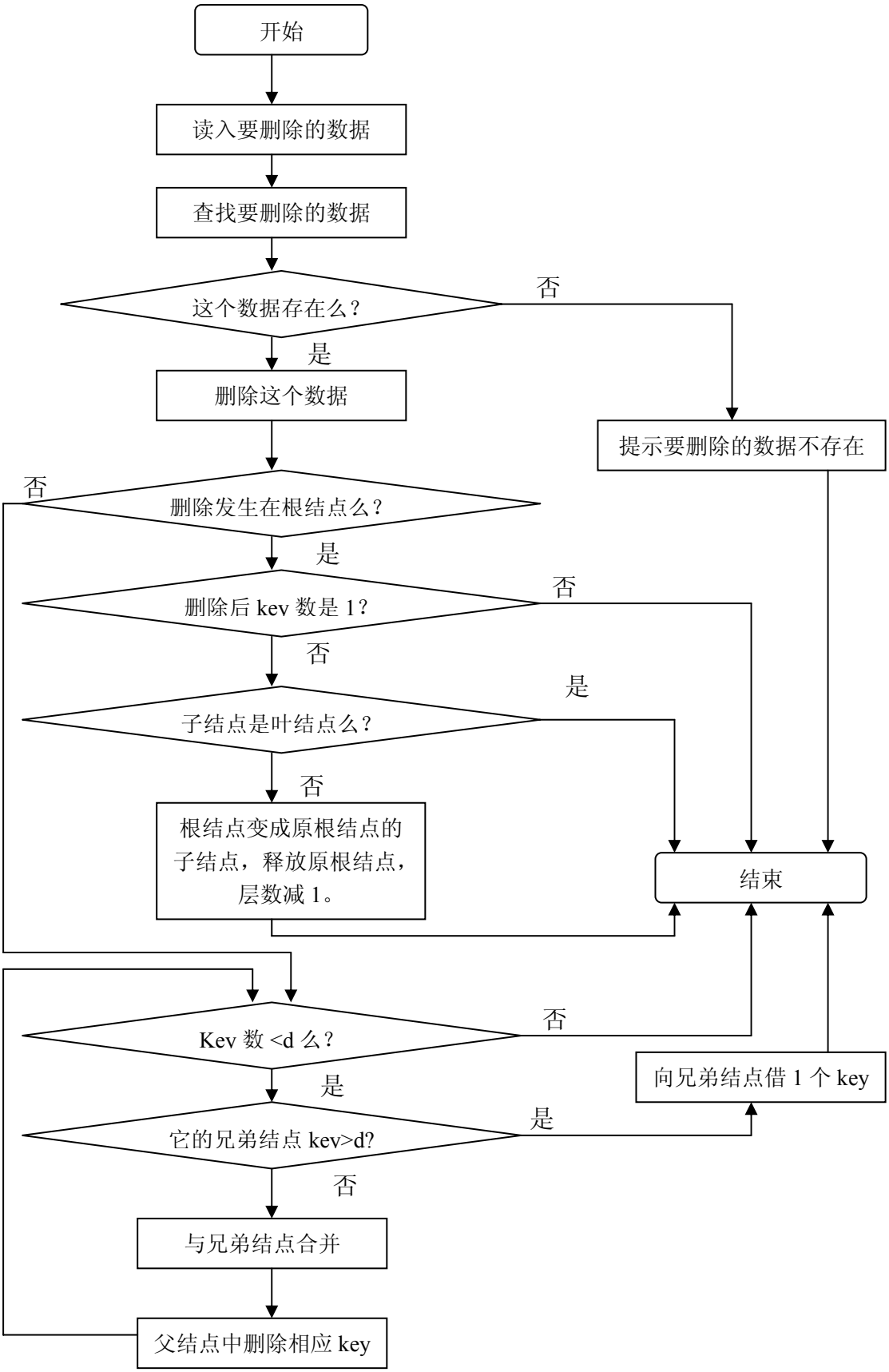
开始
读入要删除的数据
查找要删除的数据
这个数据存在么？
否
是
删除这个数据
提示要删除的数据不存在
否
删除发生在根结点么？
是
删除后 kev 数是 1？
否
否
是
子结点是叶结点么？
否
根结点变成原根结点的子结点，释放原根结点，层数减 1。
结束
Kev 数 <d 么？
否
向兄弟结点借 1 个 key
是
是
它的兄弟结点 kev>d?
否
与兄弟结点合并
父结点中删除相应 key

图2-4　删除程序流程图

2.4 满 CSB+树

在一个结点分裂的过程中需要回收一个大小为s的结点组，然后分配一个大小为s+1的结点组，结果花费了过多的时间在存储空间的分配和释放上。如果需要建立一个结点组的时候，为它分配一个满的结点组空间，就能避免这种存储分配的调用。这样只有在结点组满的时候才调用存储分配，（而不是结点满的时候）。把这种预先为全部结点组分配满的结点组存储空间的CSB+树叫做满CSB+树。

在满CSB+树中，结点分裂开销可能会比CSB+树要小。在CSB+树中。当一个结点要进行分裂的时候，就要把整个的结点组复制到另一个结点组中。在满CSB+树中，可以只转移结点组的一部分，也就是说只需要处理结点组的一半就可以了。进一步讲，源地址和目标地址与这样的转移操作有很频繁的交互。一个结点组涉及到的高速缓存行的个数有着上限s。在现代计算机体系结构中，一个在缓存中的写操作缺失经常会要求在写入数据前先把相应的缓存块调入内存。在分裂发生的时候，一般满CSB+树平均要涉及到0.5s个结点，而CSB+要涉及到2s个结点。因此满CSB+树在插入的过程中比CSB+树表现要好。另一方面，预先分配空间意味着要用额外的空间来得到这种效果。

2.5 段式 CSB+树

假设一个缓存块有128字节。每个CSB+树的结点最多有30个关键字，这意味着每个结点最多有31个子结点。一个结点组的最大的空间是31*128=4KB，所以当一个分裂进行的时候就要复制4kb的数据来建立一个新的结点组，如果在将来缓存块的空间变得更大的时候，分裂的开销将会变得更大。为了解决这个问题引入了一种新的CSB+树，称为段式CSB+树。段式CSB+树把子结点分段存储，在结点中记录各段的首地址，这样在进行插入的时候，就只需要复制其中的一部分段就可以了。

段式CSB+树按照段大小是不是固定的可以分成两种。

一种方法就是允许段的大小不一致。在装载的时候，均匀的把结点分配到两个段中，同时记录每个段的大小。在进行新结点的插入时只需要为结点所在的段创建一个新的段，然后更新相应段的大小。在这种方法中，只有一个段被涉及到（当然，当父亲结点需要分裂的时候，还是要复制所有的段）。以两段的CSB+树为例进行讨论，假设一个新结点插入到两个段中的几率是一样的，需要被复制的数据大小是1/2 * 4KB = 2KB。可以看到，这种方法可以更加的减少需要复制的数据量，但是很难保证新结点插入到两个段中的几率是一样的，因为越大的段被插入新结点的可能越大，这样就可能使大的段变得更大，同时这种方法因为要

存储每个段的关键字个数，用掉了一部分空间，每个结点能存储的关键字相对变少了。当段的个数比较多的时候会降低系统的性能。

另一种解决这个问题的方法是固定段的大小，然后向第一个段中填充结点。满了之后向第二个段中填充。在这种情况下如果要想第二段中添加结点，根本不用动第一个段。尽管当需要向第一个段中添加结点的时候需要把数据从第一个段复制到第二个段中。进行随机的插入时，平均需要复制的数据可以降低到1/2 (1/2 + 3/4)*4KB = 2.5KB。另外这种方式下子结点的个数和CSB+树相同，所以几乎没有降低查找的性能。

另外，注意到满CSB+树也是用来减少插入造成的复制的，因此本文在段式CSB+树中也引入了“满”的概念。本文采用的是固定段大小的方式，每次申请空间的时候都申请一个满的段，只有当段满的时候才需要再申请新的空间。这种方法还有一个优点，就是当段的个数是偶数时，在进行分裂时对子结点组不用任何复制操作，只需要把父亲结点分裂成两个结点就可以了，因为分裂后的结点包含的子结点刚好是原来结点组中的几个段。在进行随机的插入时，平均需要复制的数据可以降低到1/2 (1/4 + 3/4)*4KB = 2KB。

所有这些操作都可以用类似于非段式的CSB+树的方法来解决。在段式CSB+树中找到相应的结点需要更大的开销，因为需要先确定结点属于哪个段。

第三章 性能分析

在这一章中，将分析比较B+树、高速缓存敏感B+树、段式高速缓存敏感B+树、满高速缓存敏感B+树等四种不同方法的时空复杂性。

为了简单起见，假设关键字、子结点指针、和ID都需要同样的空间K，用n来记录被索引的叶结点数，用c来表示缓存块的大小，用t来表示段式CSB+树段的个数，用m来记录每个结点有几个数据空间，其中m=c/K。再假定每个结点的大小和缓存块大小一样。这些参数的典型值如表3-1所示。

表3-1 参数和它们的典型值

变量	典型值
K	4bytes
n	10^7
c	64bytes
t	2
m=c/K	16

3.1 查找性能分析

结点中的查找采用的是顺序查找的方式。顺序查找的过程为：从数据的第一个记录开始，逐个和给定的关键字进行比较，从而找到要查找的关键字。查找比较的次数等于每层比较之和。每层最少比较一次，最多和结点中每个关键字都要比较，平均比较次数为结点中关键字的个数加1除以2。高速缓存缺失可以用总的比较次数除以每个结点中的平均查找次数得到，近似等于树的层数。

B+树中，每个结点中关键字的个数为(m-2)/2，它的子结点个数比关键字的个数多1，即m/2个子结点。也就是说B+树有m/2个分支。层数为$\log_{m/2}{}^{n}$，每层平均比较 [(m-2)/2+1]/2=m/4次，平均一次查找总的比较次数$m*\log_{m/2}{}^{n}/4$。缓存缺失为$\log_{m/2}{}^{n}$。

CSB+树中，每个结点中关键字的个数为m-2，它的子结点个数比关键字的个数多1，即m-1个子结点。也就是说CSB+树有m-1个分支。层数为$\log_{m-1}{}^{n}$，每层平均比较 [m-2+1]/2=(m-1)/2次，平均一次查找总的比较次数$(m-1)*\log_{m-1}{}^{n}/2$。缓存缺失为$\log_{m-1}{}^{n}$。

满CSB+树比CSB+树多申请了一部分空间，只是改变了插入和删除的性能，

对查找没有影响。因此它的这部分分析和CSB+树完全相同。

段式CSB+树中，每个结点中关键字的个数为(m-t-1)，它每段的子结点个数都比关键字的个数多1，它的子结点个数为m-1个子结点。也就是说CSB+树有m-1个分支。层数为$\log_{m-t-1}^{n}$，每层平均比较 [m-t-1+1]/2=(m-t)/2次，平均一次查找总的比较次数(m-t) *$\log_{m-t-1}^{n}$/2。但是由于分段的原因，查找的时候要先确定要查找的数据在哪个段，需要额外的平均 (t+1)/2次的比较。缓存缺失为$\log_{m-t-1}^{n}$。

表3-2　查找时间分析

名称	分支数量	比较次数	缓存缺失	额外的比较
B+树	m/2	m*$\log_{m/2}^{n}$/4	$\log_{m/2}^{n}$	0
CSB+树	m-1	(m-1) *$\log_{m-1}^{n}$/2	$\log_{m-1}^{n}$	0
满CSB+树	m-1	(m-1) *$\log_{m-1}^{n}$/2	$\log_{m-1}^{n}$	0
段式CSB+树	m-1	(m-t) *$\log_{m-t-1}^{n}$/2	$\log_{m-t-1}^{n}$	(t+1)/2

3.2 分裂性能分析

分裂的开销一般和树的深度无关，因为一般分裂都发生在叶结点。

CSB+树需要把一个结点组的数据转移到比原来结点组大一些的结点组中，或把结点组的数据的一半转移到另一个新的结点组中。因此CSB+树分裂一共涉及到两个结点组。

满CSB+树有很少的需求，因为源地址和目的地址是交迭在一起的。因此满CSB+树平均只需要移动半个结点组就可以了。

段式CSB+树在分裂的时候平均需要移动一半的段中的数据。

表3-3　分裂开销分析

名称	分裂时涉及到的高速缓存行的个数	典型值
B+树	2	2
CSB+树	(m-1)*2	30
满CSB+树	(m-1)/2	7.5
段式CSB+树	(m-1)(t+1)/2t	11.25

3.3 空间需求分析

假定全部结点70%满，下面分别衡量内部结点和叶结点所占用的空间，假定每个叶结点有两个兄弟指针。

对于叶结点，直接用“总的索引数/每个结点可以容纳的关键字个数*每个结点所占的空间”得到。

对于内部结点，其空间是由内部结点个数乘以每个结点占用空间来获得。关键是如何获得内部结点个数。设每个内部结点平均有q个子结点，叶结点的个数为a，树的深度为h，内部结点每层的结点个数都可以用它的下层结点个数除以q来得到，因此可以得到内部结点每层的结点个数为a/q、a/q^2、a/q3……a/q^h。则内部结点的总个数为

$$a/q+a/q^2+a/q3\ldots\ldots+a/q^h=a/q(1-(1/q)^{h-1})/(1-1/q)=a/(q-1)*(1-(1/q)^{h-1})$$

当h足够大的时候，$(1/q)^{h-1}$近似为0，因此可以把式子近似转化为a/(q-1)。则内部结点占用的空间为ac/(q-1)。由此式子得出，内部所占空间是用叶结点的空间乘上1/（q-1）得到的。把各种树的q带入即可得到下表。

表3-4　空间需求分析

名称	叶结点占用空间	典型值	内部结点占用空间	典型值
B+树	2nc/[0.7(m-2)]	130.6MB	4nc/[0.7(m-2)(0.7m-2)]	28.4MB
CSB+树	2nc/[0.7(m-2)]	130.6MB	2nc/[0.7(m-2)(0.7m-1)]	12.8MB
满CSB+树	$2nc/[0.7^2(m-2)]$	186.6MB	$2nc/[0.7^2(m-2)(0.7m-1)]$	18.3MB
段式CSB+树	2nc/[0.7(m-2)]	130.6MB	2nc/[0.7(m-2)(0.7m-1)]	12.8MB

第四章 实验结果

4.1 测试方法

实验在一台个人电脑上运行（机器配置：CPU 赛扬4 1.7G；内存 DDR400 256M）。操作系统使用Windows XP。用Visual C++实现所有的方法（包括B+树、CSB+树、段式CSB+树、满CSB+树，B+树、CSB+树、段式CSB+树、满CSB+树都支持装载、查找、插入和删除，并且执行删除时采用和插入类似的方法）。选择4位整数作为关键字。同样假设一个ID和一个指针都占4位。对每次测试都进行3次，记录最小的时间。采用vtune性能分析器（由英特尔公司提供的一种工具）来收集高速缓存的缺失。利用Visual C++的时间函数才收据运行时间的数据信息。因为是比较几种树的性能，本文没有对结点中的查找做优化，几种树采用的都是最简单的顺序查找方式查找。

因为高速缓存块大小和结点大小一样对B+树最有利，所以把结点大小设置得和高速缓存行大小尽量接近。对B+树来讲，它的每个内部结点有6个关键字和7个子结点指针，还有使用的关键字的个数。每个CSB+树的内部结点包含14个关键字1个头子结点指针和使用的关键字的个数。每个满CSB+树的内部结点和CSB+树一样。段式CSB+树使用两段的分段方式，每个段式CSB+树的内部结点包含12个关键字两个子结点指针。一个B+树的叶结点由6个<key,ID>数据对、1个向前和1个向后的指针，以及关键字的个数组成。对CSB+树来说，所有的在同一个结点组的结点连续的存储在一起。因此不需要为所有的结点存储兄弟指针。在装载的时候，高一级的内部结点要用下一子树的最大的关键字来填充。采用循环的方式进行查找，避免不必要的函数调用。插入仍然递归的执行，因为要处理分裂。删除采用和插入类似的方式。

4.2 实验结果

本文一共设计了三个实验。

在第1个实验中要比较各种方法的查找功能。实验在装载过程中改变关键字的个数，测试了40000次的查找，使用了叶结点的全部空间。这个实验测试了所有提到的树，结果见图4-1，其中的数据是3次运行的结果取平均值得到的。

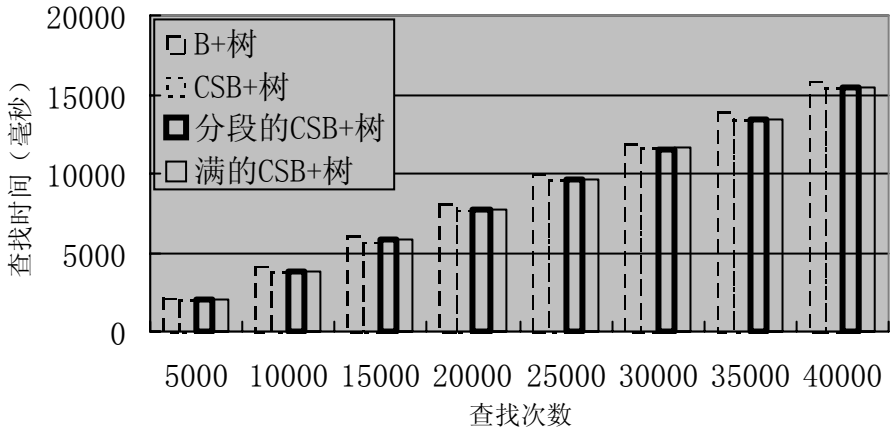

图4-1　查找实验数据

由图4-1可以看出，B+树的表现是最差的。其他3种树的搜索性能相差不大，因为满CSB+树的结构和CSB+树很类似，而段式CSB+树虽然结构上和CSB+树有较大差别，但是本文采用的段式CSB+树的每个结点的最大子结点数是相同的，导致它在查找方面也具有和CSB+树类似的性能

在第二个实验中，测试在索引结构稳定时个别的查找、插入和删除情况。查找的时间数据可以在图4-1中找到，这不再重复。

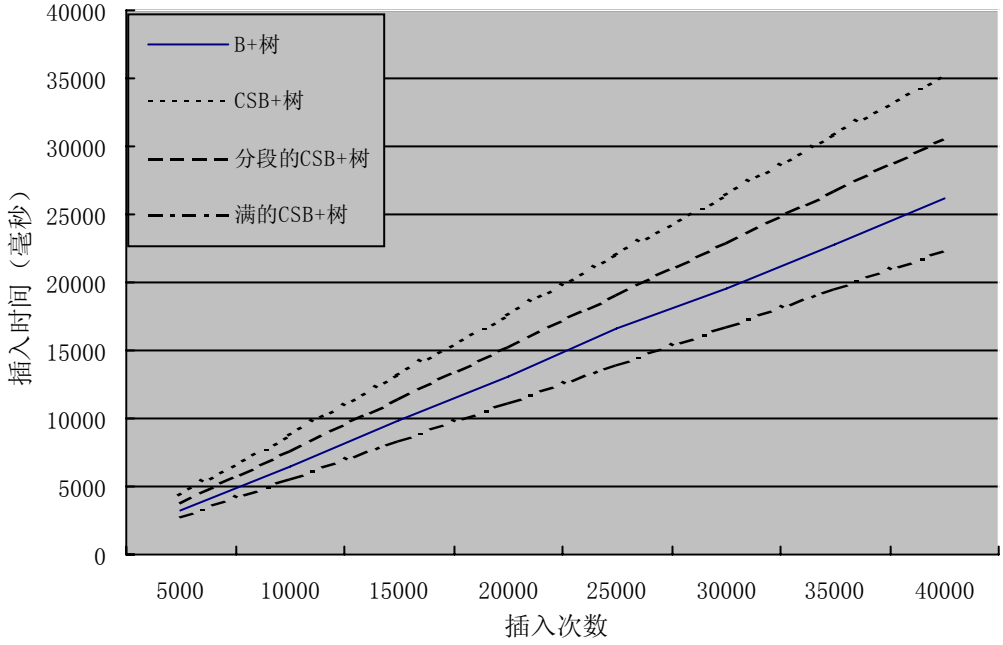

图4-2　插入时间数据

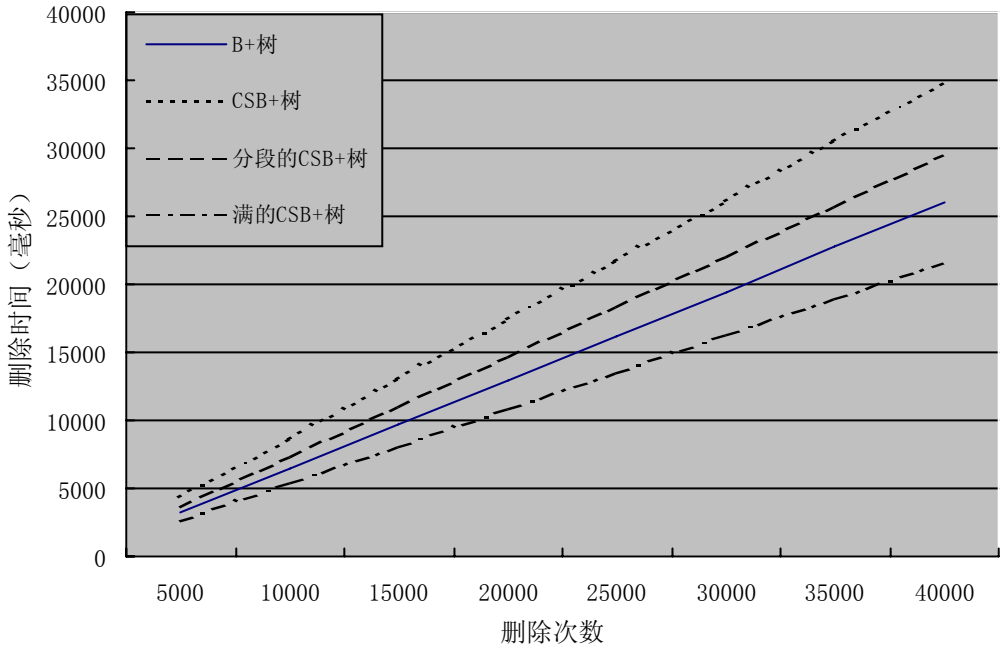

图4-3　删除时间数据

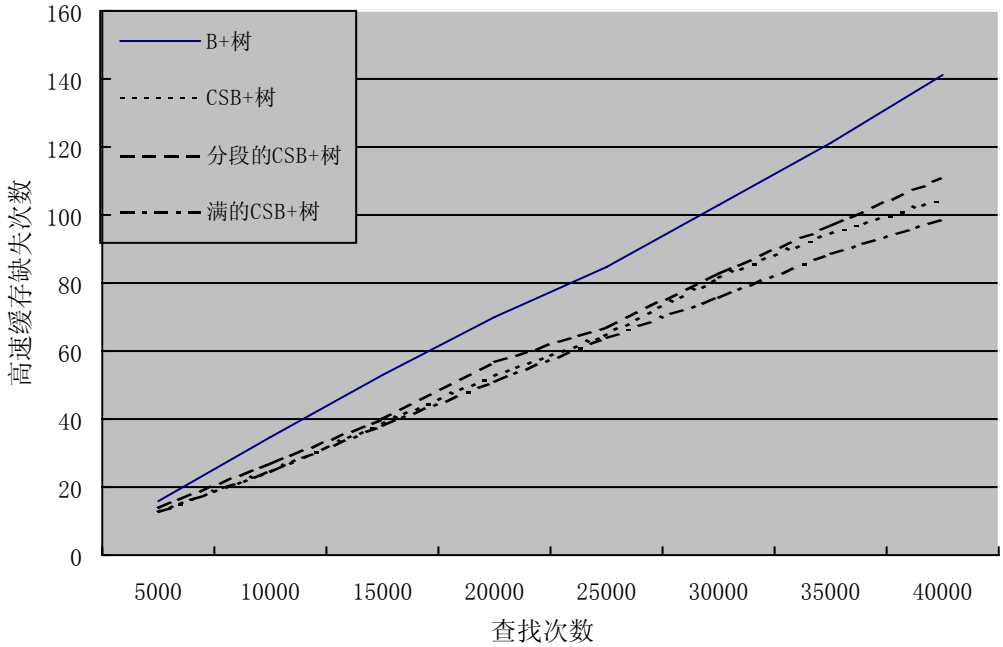

图4-4　查找高速缓存缺失

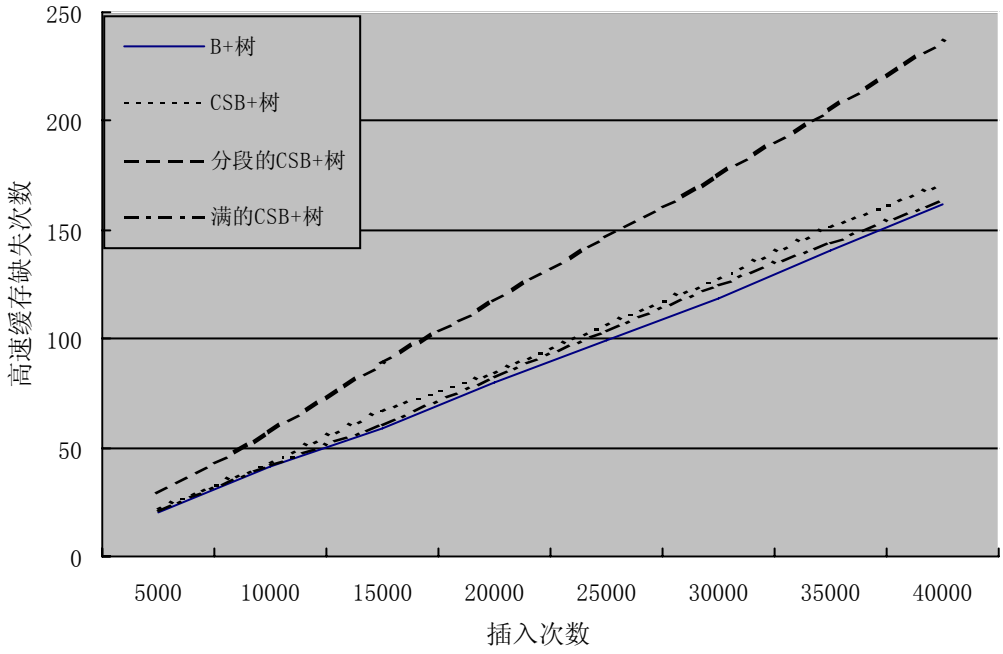

图4-5　插入高速缓存缺失

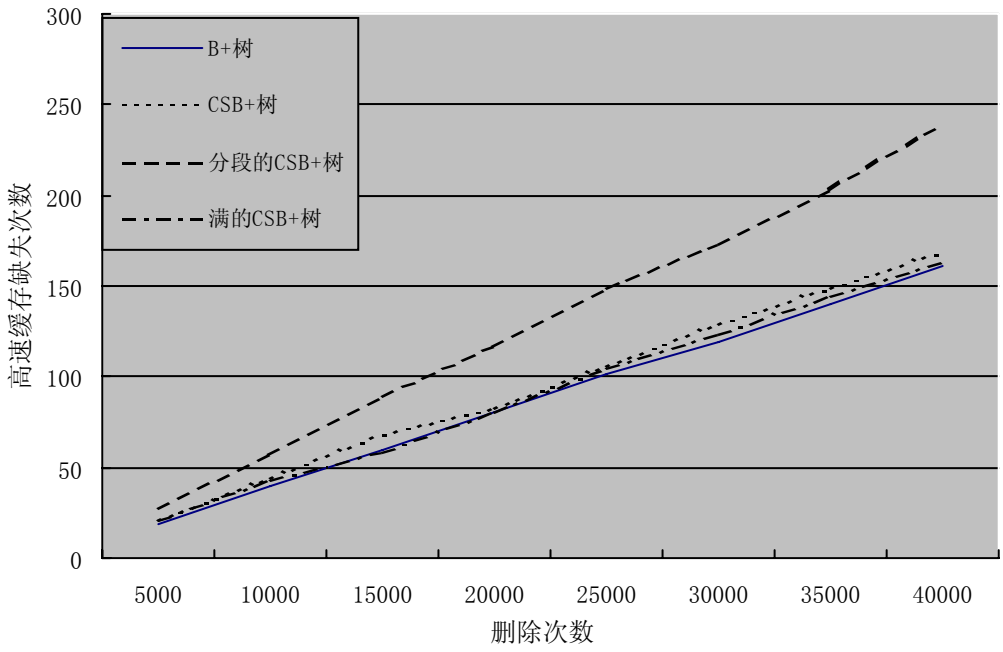

图4-6　删除高速缓存缺失

图4-1和图4-4表明对查找来讲，CSB+的性能比B+树要好，因为它充分地利

用了高速缓存，因此高速缓存缺失比较少，正如分析的一样，数据总量越大，二者的差距越大。段式CSB+树性能在B+树和CSB+树之间。段式CSB+树要比非段式的CSB+树要慢，主要原因是在查找的时候它需要额外的比较来确定数据所在的段。满CSB+树比CSB+树要好一点，它有着少一些的高速缓存缺失。

删除对应的图4-3和图4-6和分裂对应的图4-2和图4-5很像，是因为删除采用的和插入方法类似的方法进行的。CSB+在插入的时候不如B+树，插入的开销有两部分，一个是查找的开销，另一个是分裂的时间。分裂的时间包括复制1个接点组的时间，而B+树只要建立一个接点，因此CSB+树插入会比B+树花费更多的时间。段式CSB+树降低了分裂的开销，因为它只需要复制一个段。这就是两段的CSB+树比不分段要快的原因。查找的开销取决于树的大小，树越大开销越大，插入在索引结构增大的时候不会这样。满CSB+树插入的比CSB+树要快的多。更有意思的是它甚至比B+树插入的还要快。缓存的缺失并不能解释这种区别，因为满CSB+树有着更多缺失，最可能的解释是CSB+树在分裂的时候都分配空间，而满CSB+树只有在结点组满的时候才分配1次。

最后的第三个实验涉及了所有的方法的执行。首先建立像前面实验一样的稳定的树。然后对它执行40000次操作，改变查找的百分比，保持插入和删除的比率为1：1。实验结果见图4-7。

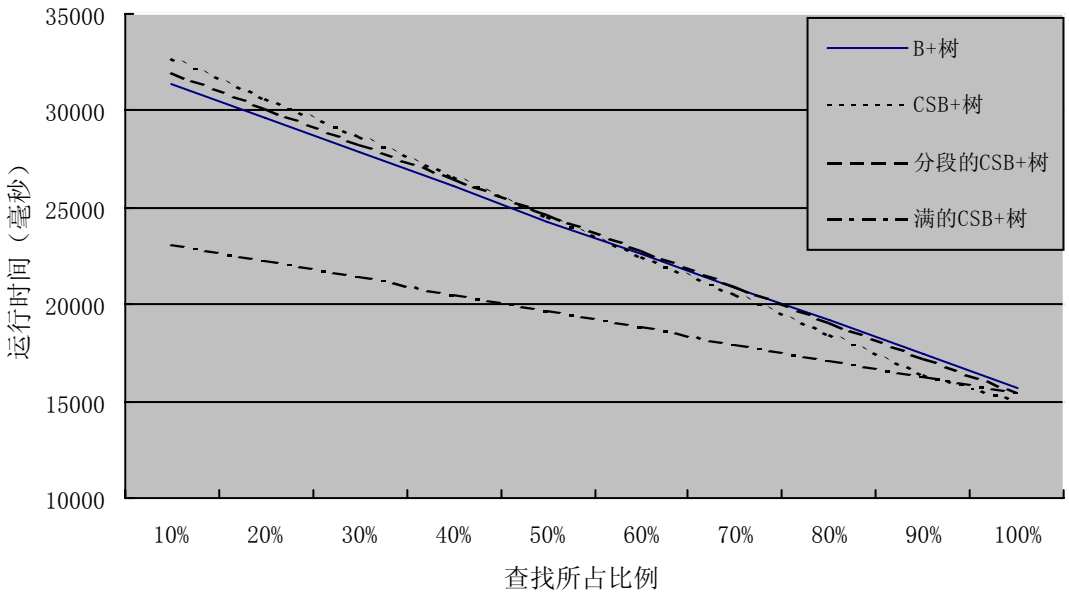

图4-7　改变查找比例后程序的运行时间

图4-7表明，满CSB+树表现的最好，虽然它使用了比其他方法更多的空间。在查找的操作比较少的情况下，B+树表现仅次于满CSB+树，但随着查找的不断增多，CSB+树都变的越来越快，当55%左右的操作是查找的时候CSB+树就开始

比B+树快了。

4.3 实验结论

除了在空间方面，满CSB+树比B+树在各方面都要好。当空间不是特别关心的时候满CSB+树是最好的选择。当空间受限制的时候CSB+树和段式CSB+树可以提供更快的查找速度，同时还支持更新。有很多应用软件，比如在线购物和数字图书馆，查找操作比更新操作要多很多。对这些应用软件，用CSB+树或段式CSB+树比B+树要好。

第五章 结论

本文设计和实现了一种索引结构，称作CSB+树。它是通过消除B+树的一部分指针得到的。CSB+树在每个缓存块中能存储更多的关键字，因此比B+树更加高速缓存感知。本文对B+和CSB+树进行了分析比较，又用实验的方式对结论进行了验证，最终结果表明了CSB+树有着比B+树更加优越的性能，因为它有着好的高速缓存行为。

随着CPU和存储器之间速度差异的增大，高速缓存的行为变得越来越重要，CSB+树最终将有可能取代B+树。

指针消除技术可以被用于其他数据结构，以改进它们的高速缓存行为，提高它们的性能。

参考文献

[1] 张新荣，于瑞国．计算机组成原理（第2版）[M]．天津：天津大学出版社，2004．53-60

[2] 王晨．内存数据库若干关键技术研究[D]． 杭州：浙江大学，2006．

[3] 孙丽梅．主存数据库中Cache敏感索引机制的研究与实现[D]． 吉林：东北大学，2005．

[4] Kristian Torp, *et al*． Efficient differential timeslice computation[R]． IEEE Transactions on knowledge and data engineering, 1998,10(4)．

[5] Martin L Kersten． Using logarithmic codeexpansion to speedup index access and maintenance[J]．Proceedings of 3rd FODO Conference．1989．228-232．

[6] 晋良颖．数据结构 [M]．北京：人民邮电出版社，2003．236-250．

[7] 魏小亮，蔡弘． B-树／B＋树的批量插入算法[J]． 北京：中央民族大学学报：自然科学版 2001，10(1)：57-61

[8] 严蔚敏，吴伟民．数据结构（C语言版）[M]．北京：清华大学出版社．238-246．

[9] 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(27)：211-213．

[10] Jun Rao，Kenneth A． Ross． Cache consciousindexing for decision-support in main memory[R] ．Proceedings of the 25th VLDB Conference, 1999．

[11] Kristian Torp, *et al*． Efficient differential timeslice computation[R]． IEEE Transactions on knowledge and data engineering, 1998,10(4)．

[12] Martin L． Kersten．Using logarithmic codeexpansion to speedup index access and maintenance[J]．Proceedings of 3rd FODO Conference．1989 ．228-232．

[13] 林鹏，李航，徐学耄．关键业务中内存数据库的 T 树索引优化[J]．计算机工程，2004，30(17)：75-97．

[14] 陈笑，陈晓霞，李维杰．程序设计技术与 C++语言简明教程[M]．清华大学出版社．

[15] Rao J，Ross K A． Making B+-Trees Cache Conscious in Main Memory[J]． In:Chen W,Naughton J F,eds． Proc of the ACM SIGMOD Conf．Dallas: ACM, 2000．475-486．

致　　谢

感谢我的导师张坤龙副教授，他严谨细致、一丝不苟的作风一直是我工作、学习中的榜样；他循循善诱的教导和不拘一格的思路给予我无尽的启迪。

感谢同组的胡正军、王乐等同学，这一段时间我们在一起讨论，一起学习，在这个过程中我学到了很多专业知识，同时我更深深的体会到协同合作的重要性。

本文在写作过程中参阅了许多前人的文献，在此，对他们表示由衷地感谢

感谢我在这四年的学习中给教授过我课程和给予我指导或帮助的各位老师。

感谢我的舍友们和我的朋友们，感谢他们在大学期间给我的关怀与帮助。

感谢我的父母，养育之恩，无以回报，你们永远健康快乐是我最大的心愿。

感谢所有的关心和帮助过我的人！谢谢！