第 7 卷第 5 期
1996 年 5 月

软 件 学 报
JOURNAL OF SOFTWARE

Vol. 7, No. 5
May. 1996

e—B$^+$树：面向多用户数据库系统优化的索引技术*

龚育昌　　王卫红

（中国科学技术大学计算机科学技术系　合肥　230027）

摘要　B$^+$树在数据库系统中已成为一种标准的索引结构，其上的并发控制机制对多用户数据库系统的性能有很大的影响. 本文提出了一种变种 B$^+$树——弹性 B$^+$树—e—B$^+$树(elastic B$^+$—tree)，定义了其上的安全点和操作及并发控制方法，对 e—B$^+$树的重构时机也进行了调整，降低了 e—B$^+$树上结点的合并/分裂频率，减少了 e—B$^+$树的维护开销，也缩短了封锁时间，从而使得其上操作的并发度和系统的效率得以提高.

关键词　并发度，合并，分裂，安全点，封锁.

B$^+$树在数据库系统中已成为一种标准的索引结构，在支持多用户并发操作的数据库系统里，B$^+$树上结点的合并/分裂可能并发进行，研究 B$^+$树上的并发控制具有重要的实际意义. 研究的目的主要在于：(1)保证各用户互不干扰，使各个用户在 B$^+$树的导航下可以正确地存取数据，并保证 B$^+$树结构的正确；(2)提高系统的并发度和系统的效率，减少用户的等待时间.

本文从以上 2 点出发，仔细研究了 B$^+$树的结构和其上的操作以及它在多用户环境下的特点，提出了一种变种 B$^+$树——弹性 B$^+$树—e—B$^+$树(elastic B$^+$—tree)，定义了其上的操作和安全点，进而提出了行之有效的并发控制方法，并证明了其正确性.

1　传统的控制方法及其特点

B. Samadi 最早提出了解决 B$^+$树上并发操作同步问题的算法.[1] R. Bayer 等对 B. Samadi 所提方法进行了改进[2]，引入了随着并发类型和并发度变化而设定的参数. P. L. Lehman 和 S. B. Yao 提出了一种变种 B 树——B^{link}树，以提高并发性能.[3] 另外，还有人引入了 Side—branching 等技术.[4] 这些方法有以下共同点：

(1)定义 B$^+$树上结点中键个数$<2n$ 时为插入安全结点(n 为 B$^+$树的阶)；结点中键个数$>n$ 时为删除安全结点.

---

* 作者**龚育昌**，女，1943 年生，副教授，主要研究领域为数据库管理系统，软件开发环境. **王卫红**，1969 年生，硕士生，主要研究领域为数据库管理系统的实现.

本文通讯联系人：龚育昌，合肥 230027，中国科学技术大学计算机科学技术系

本文 1995—03—23 收到修改稿

(2)在从根结点沿着内结点向目标叶结点搜索的过程中,确定最深安全点(离叶结点最近的安全点),封锁从最深安全点到目标叶结点的整个路径.

(3)在叶结点上完成相应修改和对结点的调整后,沿着原路径向上返回直到最深安全点,沿途完成对结点的调整和 B+树的重构.

综上所述,整个操作需要进行 2 遍扫描,即从根到叶,又从叶到最深安全点.在此期间,从最深安全点到目标叶结点的整个路径都要加以适当的封锁,不仅封锁时间长,而且封锁的结点也多.在最坏情况下,一个操作期间,整个 B+树都要被封锁,致使一个事务独占 B+树,从而退化到了串行的情况.

这些方法还隐含着另一个缺点,即:插入不安全结点(键个数为 $2n$)的分裂会产生删除不安全结点(键个数为 n);而删除不安全结点的合并又要产生插入不安全结点.在多操作并发的情况下,很可能造成结点的不断合并和分裂,使得系统维护开销过大,封锁时间过长,降低了系统的并发度和效率,这是由于对安全点的定义不当造成的.

另外,L. J. Guibas 等提出了预操作的思想[4],目的在于使整个操作在一遍中完成.基于这种思想,Y. Mond 和 Y. Raz 提出了一种变种 B+树——PO−B+树[5],定义了其上的操作和安全点,使得在从根向目标叶结点搜索期间就完成不安全结点的分裂/合并,以使操作在一遍中完成.但由于其安全点和操作定义的不恰当,仍然存在着从一种不安全结点生成另一种不安全结点的问题,而且为了进行预操作导致合并/分裂的结点增多,使得合并/分裂的频率加大,维护费用上升,反而使得系统效率下降,并发度降低.

针对上述情况,本文提出了一种变种 B+树——e−B+树,定义了其上的安全点和操作及并发控制.e−B+树有效地克服了上述各种方法的缺点,提高了系统的效率和并发度,是一种面向多用户数据库系统优化设计的索引结构.

2　e−B+树及其上操作的定义

2.1　e−B+树及安全点的定义

定义 1. n 阶 e−B+树($n>1$)是满足以下条件的变种 B+树:

(1)自根到任一叶结点的路径有相同的长度;

(2)根结点至少含有一个键值,至多含有 $2n+1$ 个键值;

(3)任一非根结点至少含有 $n-1$ 个键值,至多含有 $2n+1$ 个键值;

(4)任一含有 k 个键值的非叶结点,有 $k+1$ 个子树.

由定义 1 可见,一个 n 阶的 e−B+树的非根结点所含的键值个数可以在$[n-1,2n+1]$区间内,而 B+树结点所含键值个数只允许在$[n,2n]$区间内,e−B+树扩大了结点中键值个数的变化范围.

定义 2. 在 n 阶 e−B+树中($n>1$):

(1)一个所含键的个数为 $n-1$ 的结点为删除不安全结点,记为 *d_unsafe*;否则为删除安全的,记为 *d_safe*.

(2)一个所含键的个数为 $2n+1$ 的结点为插入不安全结点,记为 *i_unsafe*;否则为插入安全的,记为 *i_safe*.

由定义 2 可知,在 e−B^+树中,一个插入不安全结点(非叶结点)的分裂,只会产生 2 个键个数为 n 的结点,不会产生删除不安全结点;而 2 个删除不安全结点(非叶结点)的合并,产生一个键个数为 $2n-1$ 的结点,也不会产生插入不安全结点.

2.2 e−B^+树上的操作及并发控制

e −B^+树上的操作主要有 4 种:分别为读(Read)、修改(Update)、插入(Insert)、删除(Delete),会造成 e−B^+树结构变化的只有 insert 和 delete,因此本文对这 2 种操作进行详细的研究,read 和 update 最后给予简略的说明.

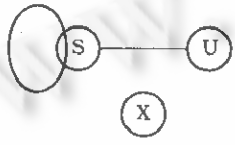
定义 3. 封锁的类型和操作:

(1)一个操作可对结点加上以下几种封锁:

S 封锁:共享性封锁,供 read,update 搜索时用;

U 封锁:更新封锁,供 insert,delete 搜索时用;

X 封锁:排它性封锁,供 insert,delete,update 对结点进行修改和树重构时用;这 3 种封锁的相容性如图 1 所示.



(a) 相容关系图(有边表示相容)

	S	U	X
S	+	+	−
U	+	−	−
X	−	−	−

(b) 相容矩阵(+表示相容, −表示不容)

图1

(2)有以下几个与锁有关的操作:

lock(v,*type*):对结点 v 加以 *type* 型的封锁(若 v 指向空结点,不作任何动作,下同);

unlock(v,*type*):解除结点 v 上 *type* 型的封锁;

convert(v,*type*1,*type*2):把结点 v 上的 *type*1 型的锁变成 *type*2 型的锁,若新锁与结点 v 上已有的锁不相容,则本事务进入阻塞状态.

2.2.1 插入操作

插入操作借鉴了预操作的思想,在从根向目标叶结点搜索的过程中,发现插入不安全结点,立即对之分裂,整个操作从上到下一遍完成,其过程如下:

(a)若根为 *i_unsafe*,则分裂之;

(b)下降至 K(要插入之键值)应插入之子树的根结点,若该结点为 *i_unsafe*,则分裂之,并调整相应父结点,转(b),直至到达叶结点;

(c)将 K 插入相应的叶结点.

上述过程中,只有当前结点和其父结点需加以封锁,下面给出递归过程的形式描述:

```
insert(k,p,q)
{把键值 k 插入到以 q 为根的 e−B+树中去,p 的初值为 nil,为 q 的父结点,并已加上 U 锁}
begin
  lock(q,U);
  if q=nil then {convert(p,U,X);ins(p,k); unlock(p,X);return}
          else {if q is i_unsafe
                  then {convert(p,U,X);convert(q,U,X);
                        temp=search(q,k);
                        q=split(p,q,k);
```

```
                    unlock(p,X);convert(q,X,U)}
                else temp=search(q,k);
            p=q;q=temp;
            insert(k,p,q)}
end.
```

下面说明在以上过程中用到的几个函数,其具体过程略:

search(p,k):在以 p 为根的树中,查找 k 应在的子树,返回该子树的根结点的指针,若 p 为叶结点,返回 *nil*;

ins(p,k):在 p 所指结点中,插入键值 k;

split(p,q,k):把 q 结点均分成2个结点,并调整其父结点 p(p 为 *nil*,生成一个新根结点),并返回一个指针,该指针指向新生成的结点中,位于 k 所在子树上的那个结点.

2.2.2 删除操作

删除操作也借鉴了预操作的思想,在从根向目标叶结点搜索的过程中,若发现有删除不安全结点,立即向其邻居结点查看,若紧邻的兄弟结点是删除安全的,向它移入一个键值,并调整其父结点,否则,把当前结点和它的一个邻居兄弟结点合并,整个操作从上到下一遍完成.其过程如下:

(a)查看根结点,确定要删除之键值所在的子树;

(b)下降至可能含 K(要删除之键值)之子树的根结点,若其为 *d_unsafe*,则查看它的紧邻兄弟结点,若有 *d_safe* 的紧邻兄弟结点,从中移入一个键值,并调整父结点;若其紧邻兄弟皆为 *d_unsafe*,则选一个紧邻兄弟结点与之合并,并调整父结点,转(b),直至叶结点;

(c)删除叶结点中相应键值.

上述过程中,只有当前结点和其父结点及要调整的紧邻兄弟结点需加以适当的封锁,下面给出递归过程的形式描述:

```
delete(k,p,q)
{删除以 q 为根的 e—B+树中的键值 k,p 的初值为 nil,为 q 的父结点,并已加上 U 锁}
begin
  lock(q,U);
  if q =nil
     then {convert(p,U,X);del(p,k);unlock(p,X);return}
     else {if q is d_unsafe
           then {convert(p,U,X);convert(q,U,X);
                 temp=search(q,k);
                 m=brother(p,q,1);lock(m,U);
                 if m is d_safe
                    then {convert(m,U,X);shift(p,q,m);unlock(m,X)}
                    else {n=brother(p,q,0);lock(n,U);
                          if n is d_safe
                             then {convert(n,U,X);shift(p,q,n);unlock(n,X)}
                             else if m<>nil
                                  then {convert(m,U,X);catenat(p,q,m)}
                                  else {convert(n,U,X);catenat(p,q,n)}}
                          unlock(p,X);convert(q,X,U)}
           else temp=search(q,k);
     p=q; q=temp;
     delete(k,p,q)}
end.
```

下面说明在以上过程中用到的几个函数,其具体过程略:

del(*p*,*k*):在 *p* 所指结点中,删除键值 *k*.

catenat(*p*,*q*,*m*):把结点 *m* 合并到 *q* 结点中来并调整父结点 *p*.

brother(*p*,*q*,*lr*):*p* 为 *q* 之父结点,*lr*=1,确定 *q* 之紧邻左兄弟结点;*lr*=0,确定 *q* 之紧邻右兄弟结点,返回其指针,若无,返回 *nil*.

shift(*p*,*q*,*m*):从结点 *m* 中移入一个键值到 *q* 中来,并调整相应父结点 *p*.

2.2.3 读和修改操作

read 和 update 不会影响 e−B$^+$树结构,其操作主要为从根向目标叶结点的搜索,在搜索中,需对当前结点加以 S 锁,若为修改操作,则在修改前,要把 S 锁变为 X 锁,它们的具体过程略.

2.3 算法讨论

以上讨论的 e−B$^+$树和其上的操作及并发控制,具有2个特点:①把树的重构结合到搜索过程中进行,在从根向目标叶结点搜索中,发现不安全结点立即进行结点的合并/分裂,以完成树的重构,从而整个操作在一遍中完成,不必再从下向上重整树上的结点.使得封锁时间短,封锁结点少,一次只封锁当前结点和其父结点,这提高了操作的并发度;②插入不安全结点的分裂不会产生删除不安全结点,删除不安全结点的合并不会产生插入不安全结点.因为在 e−B$^+$树中,*i_unsafe* 结点(键个数为2*n*+1)分裂产生2个键个数为 *n* 的结点,当且仅当这2个结点都变为 *n*−1时才有可能再次合并;*d_unsafe* 结点(键个数为 *n*−1)合并产生一个键个数为2*n*−1的结点,该结点只有再插入2个键值才会变为 *i_unsafe*,从而分裂与合并的频率得以降低,维护费用减少,提高了系统效率,同时由于要封锁的结点数目减少,并发度得以提高.

3 正确性证明

由于 read,update 不影响 e−B$^+$树的结构,因此,本文主要证明 insert 和 delete 操作的正确性.

定理1. 在多操作并发情况下,insert 操作完成后,e−B$^+$树的性质仍满足.

证明:(1)首先要证明,对1棵 e−B$^+$树单独作 insert 操作后,e−B$^+$树性质仍满足.

设树高为 *H*,在 insert 操作中,令 *h* 为当前结点(*q* 所指的结点)的层次,下面利用归纳法来证明.

(a)当 *h*=1时,*p*−>*nil*,*q*−>*boot*(根结点),有|*q*|<=2*n*+1(|*q*|表示结点 *q* 中键个数),|*p*|=0

若|*q*|=2*n*+1,则分裂之,产生2个键个数为 *n* 的结点,且|*p*|=1;

若|*q*|<2*n*+1,则向下继续搜索.

所以 *h*=1时,insert 操作完成后,有|*q*|<=2*n*,|*p*|<=2*n*+1成立.

(b)设 insert 进行到层次 *H*′−1后(*H*′<=*H*),e−B$^+$树的性质仍满足,即:|*q*|<=2*n*,|*p*|<=2*n*+1,则 *h*=*H*′时,有|*p*|<=2*n*(由于下降一层,*p* 指向 *q* 先前所指结点)

若|*q*|=2*n*+1,则分裂后产生2个键数为 *n* 的结点,且 *p* 的键数加1,|*p*|<=2*n*+1;

若|*q*|<2*n*+1,则继续向下搜索.

所以 *h*=*H*′时,insert 操作完成后,有|*q*|<=2*n*,|*p*|<=2*n*+1成立.特别地,当 *h*=*H*

($H'=H$)时,插入一个值到叶结点后,$|q|<=2n+1$. 综上所述,insert 操作完成后,e－B⁺树的性质仍满足.

(2)接下来要说明,若一个 e－B⁺树上有多个操作并发进行,insert 操作完成后,e－B⁺树性质仍满足.

由于算法中在搜索时加 U 锁,对结点要调整时用 X 锁,而 U 与 U,X 与 X,U 与 X 之间互不相容,从而在算法中,当前结点和其父结点只能被一个可以改变其结构的操作所占有,因此 insert 操作要对当前结点和其结点加以结构调整时,这2个结点被 insert 操作独占,由(1)可知,insert 对其加以调整后,$|p|<=2n+1$,$|q|<=2n$(为叶结点时$|q|<=2n+1$). 所以 $e-B^+$树的性质仍得到保持.

定理2. 在多操作并发情况下,*delete* 操作完成后,$e-B^+$树的性质仍满足(其证明过程类似于定理1,此处略).

定理3. 并发情况下,不会导致死锁.

证明:由于是在1棵树上进行操作,前节所述算法中,总是在对当前结点之父结点已取得封锁后,才会申请对当前结点加以封锁;若是 *delete* 操作,则可能在取得当前结点的封锁后,又去申请对其邻居结点加以封锁,但不管何种操作,都不会再去申请对其任何祖先结点加以封锁,可见不会形成回路,所以不会造成死锁.

4　结束语

根据上文中提出的思想,树中结点所含键值个数的变化范围可以进一步扩大,比如使其在区间[$n-2$,$2n+3$]及[$n-3$,$2n+5$]中变化,只需改变安全点的定义,上述算法仍然适用. 这样可以进一步减少结点的合并/分裂的频率,当然也付出了一定的空间代价. 在实际中可以依据具体情况在时间和空间之间作出适当的折衷,以求得最佳的性能价格比.

参考文献

1　Samadi B. B－trees in a system with multiple users. Inf. Process Lett, Oct 1976. 107～112.

2　Bayer R, Schkolnick M. Concurrency of operations on B－trees. Acta Informatica, 1977, 9: 1～21.

3　Lehman P L, Yao S B. Efficient locking for concurrent operations on B－trees. ACM TODS, 1981, 6(4): 650～670.

4　Guibas L J, Sedgewick R. A dichromatic framework for balanced trees. In: Proc. of the 19th Ann. Symp. on Foundation of Computer Science, IEEE, 1978. 8～21.

5　Mond Y, Raz Y. Concurrency control in B⁺－trees database using preparatory operations. In: Proc. of VLDB'85, Stockholm, 1985. 331～334.

6　高庆华,高燕,崔国华. 数据库多查询任务并行处理技术研究. 见:李未,怀进鹏,白硕编,智能计算机基础研究 '94,北京:清华大学出版社,1994.

7　周龙骧. 数据库管理系统实现技术. 武汉:中国地质大学出版社,1990.

8　萨师煊,王珊. 数据库系统概论. 北京:高等教育出版社,1983.

e—B^+—TREE: AN INDEXING ORGANIZATION OPTIMIZED FOR DBMS SUPPORTING MULTI—USERS

Gong Yuchang Wang Weihong

(*Department of Computer Science and Technology University of Science and Technology of China Hefei 230027*)

Abstract B^+—trees are considered standard organization for indexes in database systems. The concurrent control mechanism on B^+—trees has a great effect on the performance of DBMS supporting multi—users. A variant of B^+—tree called e—B^+—tree(elastic B^+—tree) is presented in this paper. Then, the safe nodes and operations on e—B^+—trees are defined and the moment of reconstruction of e—B^+tree is regulated. All these measures not only reduce the overhead for maintaining the e—B^+ tree and the time spent on locking, but also decrease the frequency of splitting and catenating operation. Therefore, the degree of concurrency of operations on e—B^+tree and efficiency of the database system can increase by a big margin.

Key words Degree of concurrecy, catenat, split, safe node, lock.