B + 树索引机制的研究及优化 *

施　恩，顾大权†，冯　径，赵章明
（解放军理工大学 气象海洋学院，南京 211101）

摘　要：当数据庞杂时，B + 树索引在查找效率和空间利用率方面还存在不足。针对该问题提出一种改进的 B + 树结构，首先通过调整叶子节点与非叶子节点的数量关系，以降低树的深度；然后优化原插入算法，在分裂节点前进行平衡处理（BP），以提高树的空间利用率。经实验，改进后的 B + 树与传统 B + 树相比，在查找效率和空间利用率上分别提高了 10% 和 6%，证明对 B + 树的改进具有可行性。

关键词：数据库；索引；B + 树；算法优化
中图分类号：TP391.3　　**文献标志码**：A　　**文章编号**：1001-3695(2017)06-1766-04
doi:10.3969/j.issn.1001-3695.2017.06.036

Research and improvement of B + tree indexing mechanism

Shi En, Gu Daquan†, Feng Jing, Zhao Zhangming
(*College of Meteorology & Oceanography, PLA University of Science & Technology, Nanjing* 211101, *China*)

Abstract: B + tree is short in search efficiency and space utilization when processing mass data. This paper proposed an improved structure of B + tree. Firstly, the method reduced the depth of B + tree by adjusting the number of leaf nodes and non-leaf nodes. Then, it increased the utilization of B + tree by optimizing the original insertion algorithm, which taking balance process (BP) before splitting the non-leaf nodes. Through experiment, compared with the traditional B + tree, searching efficiency and space utilization of the improved B + tree got increased by an average of 10% and 6%. It shows that the improvement is feasible.

Key words: database; indexing; B + tree; algorithm optimization

数据库作为目前最有效的管理数据的方式被广泛应用于各类应用系统中。在数据库运行过程中，使用频率最高的操作是查询操作。如何提高查询效率是数据库优化的关键问题。随着数据库数据量的增大，使用索引机制能有效提高查询效率。

索引包含了关键字和指向关键字的位置指针，是一种树状结构，其存储了数据库表中一列或几列的信息。对表中的某一列建立索引后，在执行查询操作时，无须遍历整个表，只需在对应列的索引中查找，能够快速寻找具有特定值的记录[1]。索引机制的优化工作主要基于查询效率和空间利用率两方面。对查询效率的改进主要通过降低读取磁盘数据的次数；对空间利用率的优化主要通过调整数据分配机制，使内存使用更为高效。

B + 树索引是数据库中常用的一种索引技术[2-5]。B + 树索引支持顺序查找和随机查找，能够较好地满足数据库应用需求；但是当数据量庞杂时，B + 树的空间利用率和查找效率降低。本文通过调整现有 B + 树结构，降低树的深度，以求改善查询效率；优化原有插入算法，调整叶子节点分裂的时机，采用平衡处理（BP）使叶子节点的关键字近似均匀分布，减少分裂操作次数，提高 B + 树的空间利用率。

1　B + 树索引

1.1　B + 树的特点

B + 树是一种多路搜索树，其节点分为根节点、内部节点、叶子节点三类。B + 树作为经典的数据结构，其定义在已有的教材和文献中有详细的说明[6-11]。图 1 为一棵 4 阶 B + 树的结构示意图。

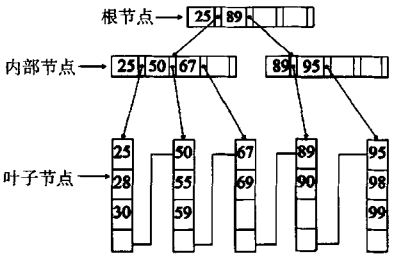

图 1　一棵 4 阶 B + 树的结构示意图

B + 树的这一结构，使其具有以下特点：a）非叶子节点的关键字等于其对应子树的最大关键字（或最小关键字）；b）具有两种查找方式，即从最小关键字所在的叶子节点开始的顺序查找和从根节点开始的多路查找；c）所有的关键字都存储在叶子节点中，非叶子节点则作为索引部分。

1.2　B + 树的基本操作

B + 树的基本操作包括传统的查找、插入和删除[6-11]。

B + 树中可以采取顺序查找和多路查找两种查找方式。顺序查找在叶子节点中进行，由于叶子节点之间存在链指针，所有叶子节点构成一个单链表，B + 树中的顺序查找与单链表的查找操作相同；多路查找从根节点开始，查找过程中若在根

收稿日期：2016-04-26；**修回日期**：2016-06-14　　**基金项目**：江苏省自然科学基金资助项目（BK20130070）

作者简介：施恩（1992-），男，福建福州人，硕士研究生，主要研究方向为数据库索引机制优化；顾大权，男（通信作者），教授，主要研究方向为智能信息系统（gudaquan3@163.com）；冯径，女，教授，主要研究方向为网络与分布式系统；赵章明，男，硕士研究生，主要研究方向为网络与分布式系统。

节点或内部节点命中,查找继续进行,不论查找成功或失败,查找操作必须进行到叶子节点[6]。

B + 树的插入需要先找到要做插入操作的叶子节点,然后根据该叶子节点及其父节点的关键字分布情况执行相对应的操作。图 2 为 B + 树的插入操作示意图,依次插入 40、45、62。

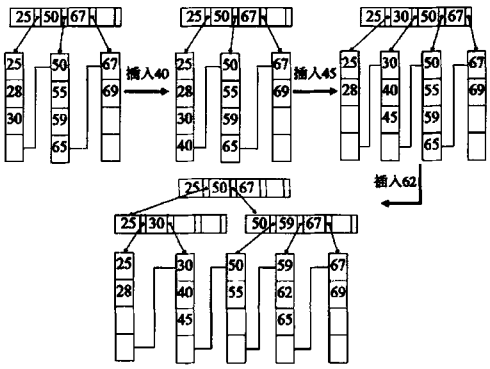

图 2　B + 树的插入操作示意图

B + 树的删除需要先找到要做删除操作的叶子节点。由于节点的数据覆盖率必须达到 50%(即节点包含的关键字个数不能小于树的阶数的一半),删除时需要用到合并节点的操作。图 3 为 B + 树的删除操作示意图,依次删除 50、69、59。

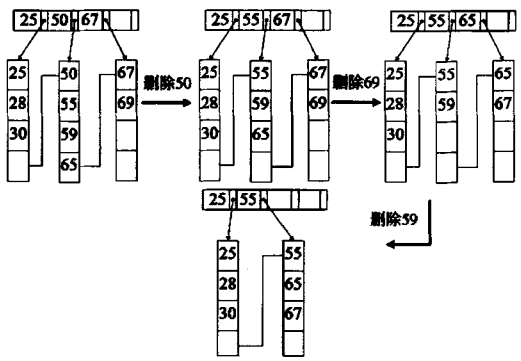

图 3　B + 树的删除操作示意图

需要注意的是,若需删除的关键字存在于祖父节点(或更高阶节点)中,则只需删除叶子节点部分,索引部分无须删除[9]。

2　改进 B + 树

2.1　改进思路

由上文总结出两点改进思路:

a) B + 树的查找效率与树的深度密切相关。B + 树的多路查找中既包含节点内部的顺序查找,也包含链接到子节点的操作,其中链接操作需要再次读取数据到内存,更为费时。由于 B + 树的查找必须进行至叶子节点,所以链接操作次数与树的深度在数值上相等。本文探讨的查找优化主要从降低树的深度入手。

b) B + 树的空间利用率和节点分裂次数存在联系,若插入数据时需要多次分裂,会导致树的空间利用率降低。定义 B + 树的空间利用率 U 为

$$U=\frac{\text{关键总数}}{\text{叶子节点数}\times N} \tag{1}$$

其中:N 为树的阶。以图 2 中的插入操作为例,在插入之前,该树的空间利用率为 0.75,由于插入三个数据的过程中进行了两次分裂节点操作,该树的空间利用率变为 0.60,空间利用率下降。本文探讨的提高空间利用率的方法主要从改进 B + 树的插入算法入手。

2.2　改进后的 B + 树定义

对 B + 树改进的目的是提高查找效率,核心思路是降低树的深度。引入枝叶比 R_{bl} 的概念,用来描述叶子节点和非叶子节点的数量关系。定义 R_{bl} 为

$$R_{bl}=\frac{\text{非叶子节点的最大关键字个数}}{\text{叶子节点的最大关键字个数}} \tag{2}$$

其中:R_{bl} 取正整数,当 $R_{bl}=1$ 时,即为传统 B + 树的结构;当 $R_{bl}>1$ 时,非叶子节点能够容纳的关键字个数增倍,树枝的数目增大,在 B + 树包含的数据总量保持不变(即叶子节点数保持不变)的前提下,能够显著降低树的深度。

一个改进后的 N 阶 B + 树的定义如下:

a)每一个叶子节点至多包含 N 个关键字,与树的阶在数值上相同;

b)每一个非叶子节点至多包含 $R_{bl}\times N$ 个关键字和 $((R_{bl}\times N+1)$ 个子节点;

c)根节点至少有两个子节点,其余的内部节点至少有 $\lceil (R_{bl}\times N)/2\rceil$ 个子节点;

d)非叶子节点的关键字与其对应的子树的最小关键字相等;

e)所有关键字都在叶子节点出现,且所有的叶子节点都在同一层上;

f)叶子节点间设置链指针,所有叶子节点构成一个单链表。

图 4 为改进后 B + 树结构示意图,其中 $N=4$,$R_{bl}=2$。

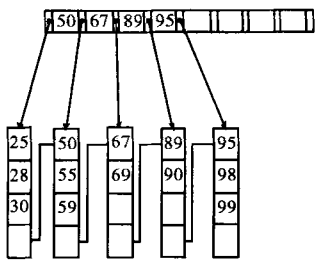

图 4　改进后的 B + 树结构示意图

改进后的 B + 树能在数据量不变的前提下,有效降低树的深度。图 4 和 1 所示的 B + 树包含的关键字相同,原 B + 树的深度为 3,改进后的 B + 树的深度仅为 2,这将有效提高查找效率。

2.3　改进后的 B + 树查找算法

改进后的 B + 树的查找与传统 B + 树基本相同,只需根据修改多路查找在非叶子节点中的循环条件。在改进后的 B + 树中查找值为 K_0 的关键字的步骤如下:

a)从根节点开始,找到小于 K_0 的最大关键字 $K_i\leqslant K_0$,此时 $i=1$;

b)在 K_i 对应的子节点中找到小于 K_0 的最大关键字 K_{i+1},由于引入了 R_{bl},此时需要判断节点类型;

c)重复步骤 b)直到 $i=\text{depth}$,其中 depth 为树的深度,即查找到叶子节点;

d)若 $K_{\text{depth}}=K_0$,则查找成功,否则查找失败。

2.4　改进后的 B + 树删除算法

改进后的 B + 树的删除与传统 B + 树类似,在涉及到节点合并时需要区分叶子节点与非叶子节点。在叶子节点中,若删除关键字后该节点的数据覆盖率小于 $\lceil N/2\rceil$,则将该叶子节点与其相邻兄弟节点合并;在非叶子节点中,若删除关键字后该节点的数据覆盖率小于 $\lceil (R_{bl}\times N)/2\rceil$,则将该节点与其相邻兄弟节点合并。B + 树的删除步骤为:

a)调用查找算法,找到理论上要进行删除操作的叶子节点,此时节点所在层数 layer = depth(树的深度)。

b)根据 layer 判断节点类型,若 layer = depth,令合并节点操作的判断参数 $C = \lceil N/2 \rceil$,否则令 $C = \lceil (R_{bl} \times N)/2 \rceil$。

c)若删除后节点的关键字数大于等于 C,则修改该节点。若需删除的关键字同时是该节点父节点的关键字,则修改相应父节点,结束。

d)若删除后节点的关键字数小于 C,则找到一个最近的兄弟节点,然后分两种情况:

(a)若 $P_{brother}$ 的关键字数大于 C,把该节点中最接近的一个数据剪切到本节点,调整父节点的键值。

(b)若 $P_{brother}$ 的关键字数等于 C,合并两个节点,调整父节点的键值。若删除键值后父节点的关键字数小于 C,则 layer = layer - 1,执行步骤 b),递归执行步骤 d)。

2.5 改进后的 B + 树插入算法

改进后的 B + 树插入算法的核心是提高同父叶子节点的空间利用率。新算法的改进之处是在父节点已满时通过平衡处理(BP)算法调整同父子节点的关键字分布,以尽量避免分裂操作。BP 算法的作用范围为一组同父叶子节点及其父节点。算法根据该组叶子节点的关键字总数调整关键字分布,使叶子节点的关键字数目大致相同(差值小于 2)。BP 算法设置一个阈值(BP_Threshold),只有当同父叶子节点的空间利用率低于该阈值时才调用算法。算法中设置的 BP_Threshold 为 $(N-1)/N$。设置此阈值的作用是保证平衡处理能够发挥作用。当同父叶子节点的空间利用率超过该阈值时,平衡处理效果不明显,而且会增加插入操作耗时。

以下是 BP 算法具体流程。

输入:满足分裂条件的非叶子节点 PNode。

输出:调整子节点分布后的非叶子节点 PNode。

```
for i = 0 to (Rbl * N + 1)
  ChildNode = PNode→getChild(i)
  Sum = Sum + getcount(ChildNode)   //获取 PNode 下的数据量
  Data[ ] = getdata(ChildNode)   //获取 PNode 下的数据
end for
k = sum % N
for i = 0 to (Rbl × N + 1)
  for j = 0 to ( sum / N)
    setdata(Data[i] , PNode→getChild(i) )   //设置关键字
  end for
  if(i < k )
    setdata(Data[i] , PNode→getChild(i) )
  end if
end for
return PNode
```

引入 BP 算法后 B + 树的插入步骤为:

a)调用查找算法,找到理论上要做插入操作的叶子节点;

b)叶子节点未满,则直接在该节点中插入数据;

c)叶子节点已满,且该节点无父节点(即根节点为叶子节点),则分裂叶子节点,插入数据,生成新的根节点;

d)叶子节点已满,但其父节点未满,则分裂叶子节点,插入数据,再修改父节点的指针;

e)叶子节点已满,其父节点已满,且所有同父叶子节点的空间利用率 $U <$ BP_Threshold,则采用 BP 算法处理目标节点,再将新数据插入;

f) 叶子节点已满,其父节点已满,且所有同父叶子节点的空间利用率 $U \geqslant$ BP_Threshold,则先分裂叶子节点,插入新数据,再分裂父节点,修改祖父节点的指针。视祖父节点情况,决定是否采取递归操作。

图 5 为改进的 B + 树插入过程示意图。

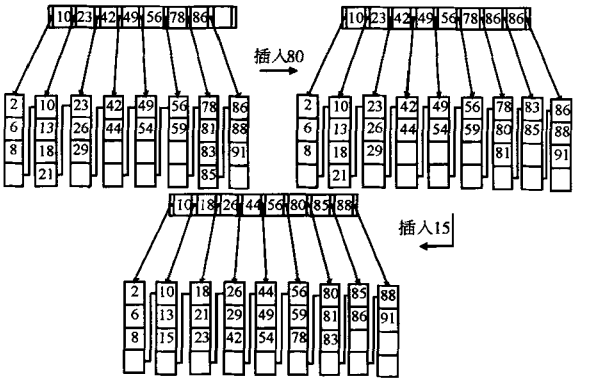

图 5 改进后的 B + 树插入过程示意图

3 实验与分析

3.1 改进后 B + 树的性能比较

为验证优化方案能否改进 B + 树索引机制,本文选择 C ++ 为编程语言,在 Visual Studio 2010 上分别实现改进前后 B + 树的构建,并通过比较两者查找耗时、树的深度以及空间利用率三方面来探讨优化方案对 B + 树性能的影响[12]。

在对比实验中,采取控制变量法以提高实验结果的可信度。一是生成 B + 树的源数据相同,这保证了改进前后 B + 树的深度、空间利用率仅由树的结构决定;二是比较查找效率时采取对同一关键字进行 20 次查找取查找耗时平均值的方法,以减小误差。实验中改进前后的 B + 树均为 8 阶,即 $N = 8$,改进方案中 $R_{bl} = 2$,得到结果如表 1、2 所示。

表 1 改进前后 B + 树的查找耗时和深度对比

类型	数据量			
	10^3	10^4	10^5	10^6
原 B + 树	799 μs /4	856 μs /5	948 μs /7	1 026 μs /8
改进后 B + 树	763 μs /3	806 μs /4	868 μs /5	924 μs /6

表 2 改进前后 B + 树的空间利用率对比

类型	数据量			
	10^3	10^4	10^5	10^6
原 B + 树	0.701	0.710	0.714	0.714
改进后 B + 树	0.748	0.757	0.758	0.760

表 1 中查找耗时的单位为 μs,"/"后的数字为该树的深度。从表 1 可以得到以下结论:a)查找耗时与树的深度成正相关,树的深度相同时,两者的查找耗时相近;b)随着数据量的增大,优化效果趋于稳定,改进后的 B + 树的深度明显减小,查找效率约有 10% 的提高。

由表 2 可以得出,随着数据量的增大,B + 树的空间利用率趋于稳定,改进后的 B + 树的空间利用率约有 6% 的提高,优化效果较好。

3.2 枝叶比 R_{bl} 对改进后 B + 树的影响

上述实验中 $R_{bl} = 2$。为探究 R_{bl} 对 B + 树性能的影响,进行实验对比不同 R_{bl} 下的 B + 树的查找耗时和空间利用率。需要说明的是,当 $R_{bl} = 1$ 时,该树代表原 B + 树。实验结果如图 6、7 所示。由图 6 可以得出,当 $R_{bl} > 1$ 时,不同 R_{bl} 下 B + 树的空间利用率的变化较小,由此可知 BP 算法是改进 B + 树的空间利用率的主要影响因素,R_{bl} 的取值主要影响查找效率。由图 7

可以得出,引入 R_{bl} 能够降低查找耗时,优化查找效率,但是 R_{bl} 与查找耗时并非简单的线性关系。根据前文分析,查找耗时与树的深度密切相关。表 3 为不同 R_{bl} 下树的深度对比。

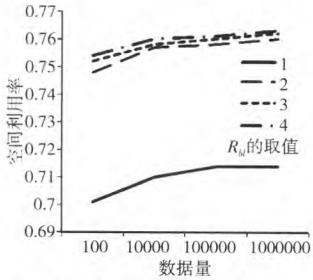

图 6　不同 R_{bl} 下 B + 树的空间利用率对比

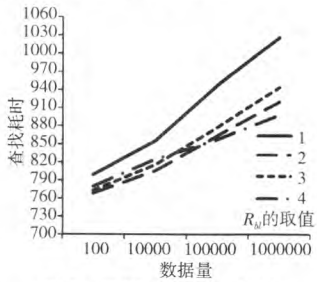

图 7　不同 R_{bl} 下 B + 树的查找耗时对比

表 3　不同 R_{bl} 下树的深度对比

R_{bl}	数据量				R_{bl}	数据量			
	10^3	10^4	10^5	10^6		10^3	10^4	10^5	10^6
1	4	5	7	8	3	3	4	5	6
2	3	4	5	6	4	3	4	4	5

结合表 3 分析图 7 的结果,当树的深度不同时,深度越小,查找耗时越小;当树的深度相同时,R_{bl} 取值越大,查找时在单个节点中遍历的耗时越长,查找耗时也就越长。此外,R_{bl} 也会影响插入和删除操作耗时。实验研究 R_{bl} 对插入和删除操作的影响。通过实验构造不同 R_{bl} 下的 B + 树,并获得在相同数据量(10^5 个关键字)下 B + 树的插入和删除耗时,实验结果如图 8 所示。

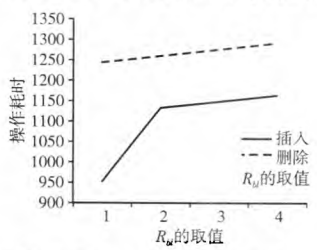

图 8　不同 R_{bl} 下 B + 树的插入和删除耗时对比

由图 8 可以得出,对于删除操作,随着 R_{bl} 的增大,删除时耗略有增大,但是涨幅很小,这说明 R_{bl} 对改进后的 B + 树的删除影响较小;对于插入操作,当 $R_{bl}=2$ 时,插入时耗有一个骤升,之后随着 R_{bl} 的增大,插入时耗略有增大,但基本平稳,这说明 BP 算法是影响插入操作的主要因素,R_{bl} 对改进后的 B + 树的删除影响较小。

上述实验中,树的阶均为 8(即 $N=8$)。为了进一步研究对于不同阶的 B + 树,R_{bl} 对 B + 树的查找、插入、删除的综合影响,探究不同 N 下的 R_{bl} 最佳值,调整参数进行对比实验。实验证明,在关键字总数不超过 100 万的前提下,通过设置合适的 R_{bl},控制树的深度小于 6,此时能够获得较好的查找效率。表 4 为不同 N 下的 R_{bl} 最优取值。

表 4　不同 N 下的 R_{bl} 最优取值

取值	B + 树的阶 N					
	3	4	5	6	7	8
R_{bl}	5	4	3	3	2	2

通过实验,改进后的 B + 树在查找效率和空间利用率上都有一定的提高,具有更好的性能,能够更好地满足数据库实时性要求[13]。需要说明的是,改进后的 B + 树的插入效率有一定程度上的降低,这主要是因为在插入时采用 BP 算法一定程度上增加了插入操作时耗。考虑到数据库的实际应用场景,查询操作使用频率最高,因此优化方案是可行的。

4　结束语

提高 B + 树索引的查找效率和空间利用率是本文的研究目的。本文分析了现有 B + 树的数据结构和基本算法,提出优化方案。通过控制叶子节点与非叶子节点的阶数以降低树深度;调整节点插入策略以减少分裂操作次数。通过对比,改进后的 B + 树索引机制能够缩短查找耗时,并提高 B + 树的空间利用率。改进后的插入算法会在一定程度上增加插入操作的时耗,但能够带来较高的查找效率,这对实时性要求较高的数据库系统有现实意义。将改进后 B + 树索引机制应用于装备管理系统中,运行情况良好,能够较好地满足系统需求。

参考文献:

[1] Bohm C, Berchtold S, Kriegel H. Multidimensional index structures in relational databases[J]. Journal of Intelligent Information Systems,2000, 15(1):51-70 .

[2] Tobin J, Lehman M C. A study of index structures for main memory database management system [C]//Proc of the 12th International Conference on Very Large Database. 1986: 294-302.

[3] Locmis M E S. Data management and file structures [M]. 2nd ed. [S. l.]:Prentice Hall, 1989.

[4] Li Xianhui, Ren Cuihua, Yue Menglong. A distributed real-time database index algorithm based on B + tree and consistent hashing[J]. Procedia Engineering, 2011,24:171-176.

[5] Hasan A K M. An index structure for large order database maintenance using variants of B trees[J]. Information Technology Journal, 2008,7(7):1061-1066.

[6] Priyadarshini S, Sahoo G. A modified and memory saving approach to B + tree index for search of an image database based on chain codes [J]. International Journal of Computer Applications, 2010,9 (3):2-5.

[7] Zhou Wei, Lu Jin, Luan Zhongzhi. SNB-index: a SkipNet and B + tree based auxiliary cloud index[J]. Cluster Computer, 2014,17 (2): 453-462.

[8] Bayer R, McCreight E M. Organization and maintenance of large ordered indexes [J]. Acta Informatica, 1972,1(3): 173-189.

[9] Comer D. The ubiquitous B-tree [J]. Computing Surveys,1979, 11(2): 123-137.

[10] Cormen T, Leiserson C, Rivest R. Introduction to algorithms[M]. 2nd ed. [S. l.]:MIT Press and McGraw-Hill, 1992.

[11] Knuth D. The art of computer programming [M]. 2nd ed. [S. l.]: Addison-Wesley, 1998.

[12] Srinivasan V, Carey M J. Performance of B + tree concurrency algorithms [C]//Proc of ACM SIGMOD International Conference on Management of Data. 1993:416-425.

[13] Tozun P, Pandis I, Johnson R. Scalable and dynamically balanced shared-everything OLTP with physiological partitioning[J]. VLDB Journal, 2012,22(2):151-175.

[14] Wang Wei, Wang Tujun, Shi Baile. Dynamic interval index structure in constraint database systems [J]. Journal of Computer Science and Technology, 2000,15(6):542-551.

[15] Taniar D, Rahayu J W. A taxonomy of indexing schemes for parallel database system[J]. Distributed and Parallel Databases, 2002, 12(1):73-106.

[16] Mehedintu A, Pirvu C, Etegan C . Indexing strategies for optimizing queries on MySQL[J]. Annals of the University of Petrosani, 2010,4(1):201-210.

[17] Huang P W, Lin P L, Lin H Y. Optimizing starage utilization in R-tree dynamic index structure for spatial database[J]. Journal of System and Software, 2001,55(3):291-299.

[18] Barbar A, Ismail A. Stachastically balancing trees for file and database systems [J]. International Journal of Green Computer, 2013,4(1):50-70.

[19] Zhu Mingdong, Shen Derong, Yue Kou, *et al*. A framework for supporting tree_link indexes on the chord overlay[J]. Journal of Computer Science and Technology, 2013,28(6):962-972.

[20] Jalata I, Sippu S, Eijas S S. Concurrency control and recovery for balanced B-link trees[J]. The VLDB Journal, 2005, 14(2):257-277.