

# 随机跳跃索引:一种支持随机插入的可信赖索引

刘凤晨<sup>1)</sup> 黄 河<sup>2)</sup> 刘庆文<sup>3)</sup> 丁永生<sup>1),4)</sup>

<sup>1)</sup>(东华大学信息科学与技术学院 上海 201620)

<sup>2)</sup>(北京航空航天大学软件学院 北京 100083)

<sup>3)</sup>(北京科技大学信息工程学院 北京 100083)

<sup>4)</sup>(数字化纺织服装技术教育部工程研究中心 上海 201620)

**摘 要** 跳跃索引是一种可信赖性索引,但只能为严格单调递增的序列建立索引,不能处理非顺序序列.为了解决这个问题,文中提出了一种新的索引,它可以对任意顺序的序列建立索引,并且依然保证索引的可信赖性.通过在原有跳跃索引结构中加入左侧跳跃指针的方法,索引节点可以根据待加入节点值的大小将其纳入自己的左侧或右侧指针以处理随机序列;索引结构中的每一个节点到根节点的路径固定且唯一,保证了索引的可信赖性.实验结果和理论证明都表明该索引是可以处理随机序列的可信赖索引,相对原有索引,索引建立复杂度明显降低且具有相同的查找复杂度.文中的创新之处是在保证索引的可信赖性的基础上解决了跳跃索引不能为随机序列建立索引的问题.

**关键词** 可信赖性;倒排表;索引;B+树;检索;算法

**中图法分类号** TP391

**DOI号**: 10.3724/SP.J.1016.2009.00974

## A Trustworthy Index for Inserting Random Sequences

LIU Feng-Chen<sup>1)</sup> HUANG He<sup>2)</sup> LIU Qing-Wen<sup>3)</sup> DING Yong-Sheng<sup>1),4)</sup>

<sup>1)</sup>(College of Information Sciences and Technology, Donghua University, Shanghai 201620)

<sup>2)</sup>(College of Software, Beihang University, Beijing 100083)

<sup>3)</sup>(School of Information Engineering, University of Science and Technology Beijing, Beijing 100083)

<sup>4)</sup>(Engineering Research Center of Digitized Textile & Fashion Technology of Ministry of Education, Shanghai 201620)

**Abstract** Jump Index is a kind of trustworthy index which can only build index for strictly monotonically increased sequences rather than unordered sequences. To solve this problem, this paper proposes a novel index supporting building index for random sequences, without losing the trustworthy of jump index. By adding left pointers on the node of jump index, the candidate node in random sequences can be indexed into one of the left or right pointers of a node in the index. The left jump pointers are used to index nodes with less value than current node's value, while right jump pointers point to those nodes with higher value. Every node in the index has the unique and permanent path from root node, which ensures the trustworthy in the index. Experiment results and proofs in this paper show that the index is trustworthy and supports indexing for random sequences, also it has more efficiency of building index and same complexity of search compared with jump index. The contribution of this paper is that the new index with trustworthy solve the problem that jump index cannot insert and build index for random sequences.

**Keywords** trustworthy; inverted index; index; B+tree; retrieval; algorithm

收稿日期:2007-09-04;最终修改稿收到日期:2009-01-03. 本课题得到国家项目(A2120061061)、上海市科学技术委员会重点基础研究项目(08JC1400100)、上海市人才发展资金(001)、上海市领军人才后备人选专项资金资助. 刘凤晨,男,1982年生,博士研究生,主要研究方向为信息检索、智能算法、检索算法. E-mail: aric@mail.dhu.edu.cn. 黄 河,男,1972年生,博士,副教授,主要研究方向为网络与信息安全. 刘庆文,男,1966年生,博士,讲师,主要研究方向为数据库、分布式信息处理. 丁永生,男,1967年生,博士,教授,博士生导师,从事智能系统、网络智能、DNA计算、人工免疫系统、生物网络结构、生物信息学、数字化纺织服装、智能决策与分析等研究.

# 1 引言

在商业运作以及一些重大事务的关键决策过程中,电子邮件、会议记录、财务状况以及项目计划书等诸如此类的文件起着关键作用,属于重要而且基本的信息.因此这些信息必须以一种可信赖(trust-worthy)的方式保存,从而使它们不能被恶意地删除或者篡改,并且易于访问.然而,随着文档被大量的电子化,这些信息也变得更加容易被篡改、删除,且不留任何痕迹.因此,确保数据信息的易访问性、准确性、可靠性和不可抵赖性已经成为势在必行的趋势.

在这种趋势的推动之下,WORM(Write Once Read Many,一次写入多次读取)存储设备便成了此类信息存储的主要方式.但是在 WORM 保存数据并不能充分保证数据的可信赖性<sup>[1]</sup>,例如,它不能为以前所发生的事件提供不可抵赖性证据.此外,即便是数据保存在 WORM 设备中,如果访问这些数据的索引可以被改动的话,数据还是可以被隐藏或者删除的.例如,索引中指向某一数据的指针被删除或者改动后指向另一数据,这时原来的数据会再也访问不到了.因此,仅仅是数据以可信赖的方式保存是不够的,索引本身也要以可信赖的方式保存.

为了解决可信赖性索引的问题,近几年的研究也提出了一些方法,其中跳跃索引(jump index)<sup>[2]</sup>是目前解决索引可信赖性最好的方法,但它要求被索引文档 ID 有序递增,不支持以随机插入的方式建立索引.本文针对跳跃索引不能支持以随机插入的方式建立索引的不足,设计了新的跳跃索引结构,该索引支持被索引文档的非顺序插入,而且具有和原有索引相同的可信赖性.

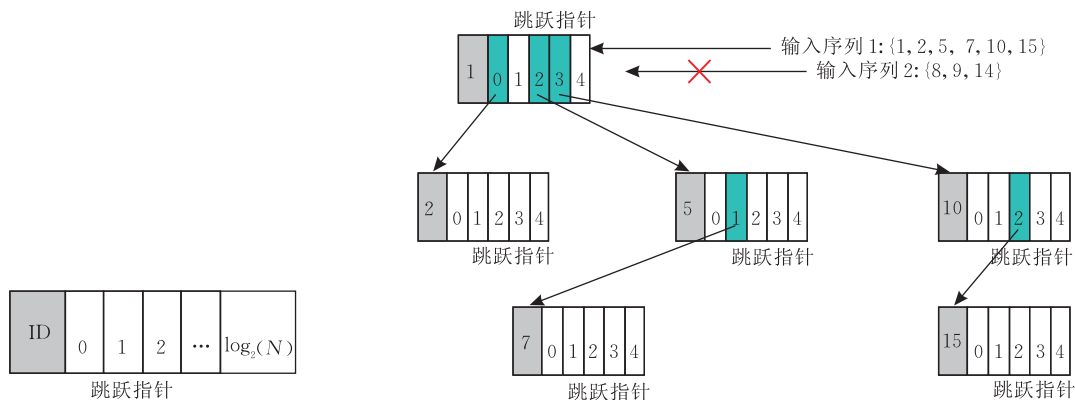
# 2 相关工作

针对索引的可信性,目前的一些研究中提出了化石索引(fossilized index)<sup>[1]</sup>的概念,主要思想是不允许对索引进行操作.GHT(Generalized Hash Tree)<sup>[1,3-4]</sup>就是一种化石索引,但这种方法对数据信息的结构要求严格,不具有通用性.它可以根据数据的属性提供精确查找,适用于结构化的数据.但是大部分商业数据,例如邮件内容、会议记录、备忘录等都是非结构化或半结构化的,这类信息查找最适合的方法是关键词查找.基于关键词的查找最典型的方法就是倒排表(Inverted Index)<sup>[5-6]</sup>.Jump Index

就是在每一条记录的倒排表(Posting lists)<sup>[7-9]</sup>中以 Jump Index 方式建立索引.该索引的建立是为了提高不同倒排表之间的合并和连接(Merge and Join)<sup>[2]</sup>操作.图 1 中是 Zigzag Join<sup>[2,10]</sup>算法,它可以将两个倒排表以时间复杂度  $O(n_1 + n_2)$  合并与连接, $n_1, n_2$  分别为两个倒排表的长度,是一种效率很高的算法.Jump Index 就是为了支持 Zigzag Join 算法中的 FindGeq()方法.图 2(a)中是 Jump Index 铬索引的节点的结构,Jump Index 每一个节点保存着倒排表中文章 ID,铬指针指向下一个节点.例如, $l$  节点的第  $i$  个铬指针指向节点  $l'$ ,  $l + 2^i \leq l' < l + 2^{i+1}$ .Jump Index 的结构保证了索引中从根节点到任意一个节点的路径是唯一的,这就确保了索引的可信性.但是,Jump Index 要求待建立索引的序列是严格单调递增的,这样的约束条件为 Jump Index 带来了很大的局限性.例如,对一个随机的序列首先要对其排序后才能建立索引,之后,如果新的序列到来只能将原有序列与新序列合并排序后重新建立索引,对较小的序列来说尚且可行,但是对非常大的序列来说,满足这种约束条件后再建立索引所付出的代价是非常大的.图 2(b)的例子中,序列 1{1,2,5,7,15}满足约束条件,可以插入到索引中;序列 2{8,9,14}不满足约束条件,因为 9 和 14 要插入的位置已经被 10 和 15 占据.遇到这种情况,Jump Index 只能将两个序列合并为{1,2,5,7,8,9,10,14,15}后重新建立索引.

```
ZIGZAG(list1, list2)
1. top1 ← list1.Start()
2. top2 ← list2.Start()
   {Initialize the iterators to point to list heads}
3. loop
4. if ((top1=list1.End()) OR (top2=list2.End())) then
5. return
6. end if
7. if (( *top1) < ( *top2)) then
8. top1 ← list1.FindGeq( *top2)
   {Find an element greater than or equal to top2}
9. continue
10. end if
11. if (( *top2) < ( *top1)) then
12. top2 ← list2.FindGeq( *top1)
   {Find an element greater than equal to top1}
13. continue
14. end if
15. if (( *top2) = ( *top1)) then
16. OUTPUT (top1)
   {Next element in join}
17. top1 ← list1.FindGeq( *top1+1)
18. top2 ← list2.FindGeq( *top2+1)
19. continue
20. end if
21. end loop
```

图 1 Zigzag Join 算法



(a) Jump Index 跳跃索引的节点结构

(b) 索引的插入过程

图 2 Jump Index 跳跃索引结构与插入过程

### 3 可随机插入的跳跃索引

#### 3.1 随机跳跃索引

能否让 Jump Index 摆脱严格单调递增的束缚呢? 为了克服原有跳跃索引的不足, 我们设计了一种新的可信赖的且支持随机插入的索引——随机跳跃索引(Random Jump Index). 这种索引不要求倒排表中的文章 ID 必须严格单调递增, 这些文章 ID 可以是任意序列(单调递增、递减或无序). Random jump index 支持 Zigzag Join 算法中的 FindGeq() 方法, 并且能在最大时间复杂度为  $O(\log(N))$  找到结果, 其中  $N$  是倒排表中文章 ID 序列中最大值. 该索引的时间复杂度的范围相比 B+ 树查找的时间复杂度  $O(\log(n))$  要宽松一些, 这里  $n$  为 B+ 树中当前叶节点的个数, 但这保证了索引中每个节点的路径唯一且保持不变. 事实上, 在实际应用当中, 文章 ID 是随着文章总数增长而增加的,  $N$  相当于数据库中文章的总数目. 因此, 随机跳跃索引的时间复杂度是文章总数的对数级.

任何数  $k$ ,  $k < N$ , 在以 2 为底数时, 都可以在  $O(\log(N))$  步之内到达. 跳跃索引的方法就是受这种将一个数与二进制相互转换的启发而得来的. 假设一个数字序列为  $0, 1, \dots, N-1$ , 其中  $N=2^p$ . 任意数  $k$  ( $0 \leq k < N$ ) 可以表示成二进制数  $b_1 \dots b_p$ , 若由此序列得到  $k$ , 可以从序列第一个位置开始, 向前跳跃  $b_1 \times 2^{p-1}$ , 然后再向前跳跃  $b_2 \times 2^{p-2}$  直至  $b_p \times 2^0$ , 最后得到  $k$ .

如果我们将这种方法运用到倒排表中, 由于倒排表不一定包含所有的文章 ID, 所以必须告诉跳跃指针(jump pointer)下一步跳跃的步长. 图 3(a) 中是随机跳跃索引的节点的结构, 中间的 ID 表示文章

ID, ID 左边的指针(left pointer)指向比它小的文章 ID, 右边的指针(right pointer)指向比它大的文章 ID.  $l$  节点右边的第  $i$  个指针指向比  $l$  大的节点  $l'$ ,  $l+2^i \leq l' < l+2^{i+1}$ . 图 3(b) 中, 第 1 个节点的右边第 0 个指针指向 2, 因为  $1+2^0 \leq 2 < 1+2^1$ ; 右边第 2 个指针指向 5, 因为  $1+2^2 \leq 5 < 1+2^3$ , 右边其它的指针都是这样计算得出的.  $l$  节点左边的第  $j$  个指针指向比  $l$  小的节点  $l''$ ,  $l-2^{j+1} < l'' \leq l-2^j$ . 图 3(c) 中, 第 15 个节点的左边第 0 个指针指向 14, 因为  $15-2^{0+1} < 14 \leq 15-2^0$ ; 第 10 个节点的左边第 0 个指针指向 9, 因为  $10-2^{0+1} < 9 \leq 10-2^0$ .

下面讨论更一般的情况: 假设倒排表中的记录为  $n_1, \dots, n_N$ . 从随机跳跃索引的根节点开始, 通过左右跳跃指针, 序列中的每一个记录都可以查找到. 查找过程如下: 设待查找的节点为  $n$ , 首先从根节点  $n_1$  开始, 找到节点的跳跃指针  $i_1$ , 当  $n_1 > n$  时,  $n_1 - 2^{i_1+1} < n \leq n_1 - 2^{i_1}$ ,  $i_1$  为左指针; 当  $n_1 < n$  时,  $n_1 + 2^{i_1} \leq n < n_1 + 2^{i_1+1}$ ,  $i_1$  为右指针, 然后通过跳跃指针  $i_1$  从  $n_1$  跳到另一个节点  $n_{i_1}$ . 在  $n_{i_1}$  中找到节点的跳跃指针  $i_2$ , 当  $n_{i_1} > n$  时,  $n_{i_1} - 2^{i_2+1} < n \leq n_{i_1} - 2^{i_2}$ ,  $i_2$  为左指针; 当  $n_{i_1} < n$  时,  $n_{i_1} + 2^{i_2} \leq n < n_{i_1} + 2^{i_2+1}$ ,  $i_2$  为右指针, 通过跳跃指针  $i_2$  找到节点  $n_{i_2}$ , 之后重复上述步骤直至找到  $n$ . 在图 3(c) 中, 查找节点 9, 首先从根节点 1 跳到节点 10, 此时跳跃右指针  $i_1 = 2$ ; 然后从节点 10 跳到节点 9, 此时跳跃左指针  $i_2 = 0$ .

图 4 中, 以伪代码的形式列出了插入算法 Insert( $k$ )、查找算法 Lookup( $k$ )、FindGeq( $k$ ) 算法返回第一个大于等于  $k$  的节点, FindGeqRec( $k, s$ ) 算法是以  $s$  为根节点在其子树中寻找第一个大于等于  $k$  的节点. 在 Zigzag Join 算法中执行 FindGeq( $k$ ) 可直接调用 FindGeqRec( $k, s$ ) 算法.  $s.rptr[i]$  表示  $s$  节点中右指针  $i$ ,  $s.lptr[j]$  表示  $s$  节点中左指针  $j$ . assert 用来检查所得节点是否满足约束条件.

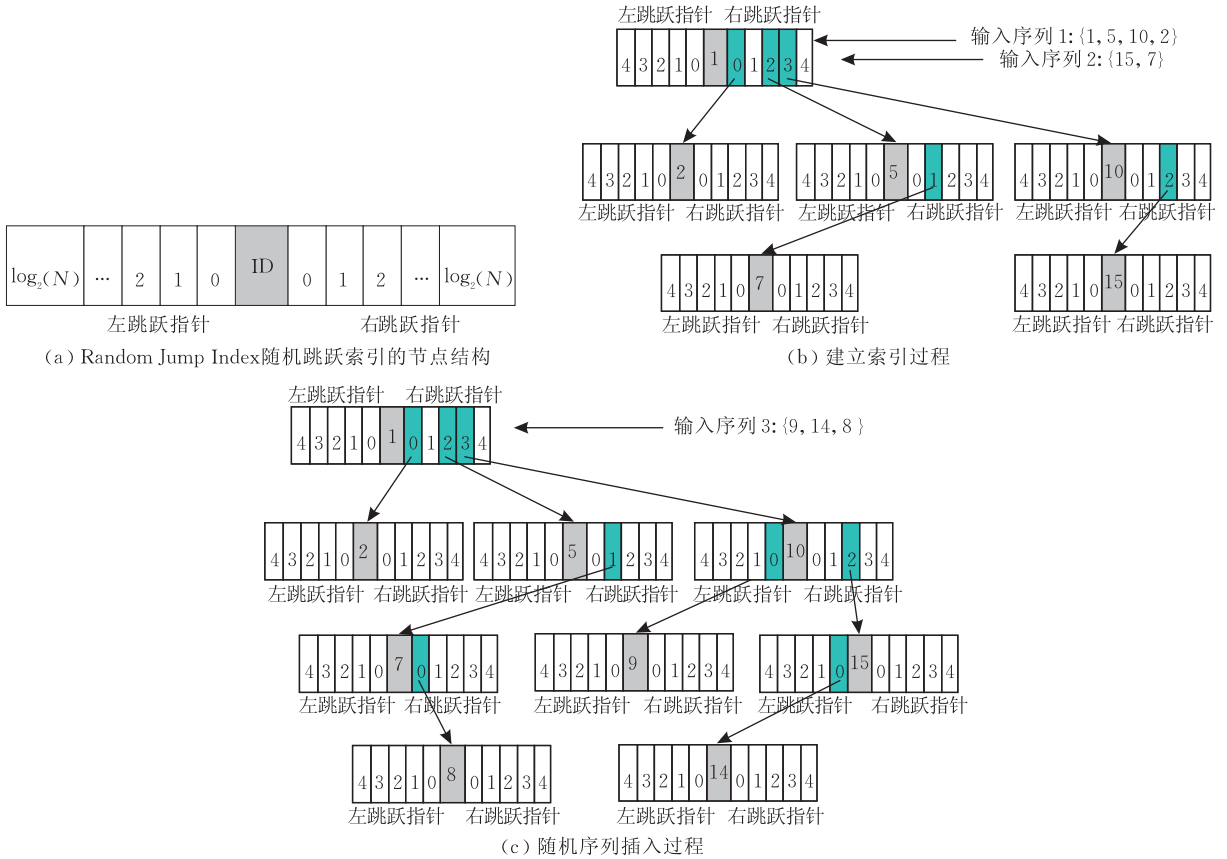


图 3 Random Jump Index 随机跳跃索引结构与索引建立过程

### 3.2 复杂度分析

为了清晰地说明随机跳跃索引的复杂度首先给出下列命题,然后证明之.

**命题 1.** 设  $i_1, \dots, i_j$  是算法  $\text{Lookup}(k)$  执行过程中循环执行第 7 或 20 步所得  $i$  和  $j$  的序列,则  $i_1, \dots, i_j$  序列中  $i_1 > \dots > i_j$ .

**证明.** 在随机跳跃索引中执行  $\text{Lookup}(k)$  依次跳过的文章 ID 序列为  $s_1, \dots, s_j$ ,  $s_1$  为 Random Jump Index 中的根节点. 在查找  $k$  的过程中从  $s_1$  跳到  $s_2$  会出现以下 4 种情况: (1)  $s_1 < s_2 < k$ ; (2)  $s_1 < k < s_2$ ; (3)  $k < s_2 < s_1$ ; (4)  $s_2 < k < s_1$ . 现在分别对这 4 种情况进行证明:

(1)  $s_1 < s_2 < k$ , 由算法  $\text{Lookup}(k)$  第 20 步可得  $s_1 + 2^{i_1} \leq k < s_1 + 2^{i_1+1}$  (a), 因为  $s_1$  的右指针  $i_1$  指向  $s_2$ , 所以  $s_1 + 2^{i_1} \leq s_2 < s_1 + 2^{i_1+1}$ ,  $s_2 \geq s_1 + 2^{i_1}$  (b). 同理得到  $s_2 + 2^{i_2} \leq k < s_2 + 2^{i_2+1}$  (c). 由 (a) 和 (c) 得到  $s_2 + 2^{i_2} < s_1 + 2^{i_1+1}$  (d). 再由 (b) 和 (d) 得  $s_1 + 2^{i_1} + 2^{i_2} < s_1 + 2^{i_1+1}$ . 因此  $i_1 > i_2$ .

(2)  $s_1 < k < s_2$ , 由算法  $\text{Lookup}(k)$  第 20 步可得  $s_1 + 2^{i_1} \leq k < s_1 + 2^{i_1+1}$  (a), 因为  $s_1$  的右指针  $i_1$  指向  $s_2$ , 所以  $s_1 + 2^{i_1} \leq s_2 < s_1 + 2^{i_1+1}$ ,  $s_2 < s_1 + 2^{i_1+1}$  (b). 由于  $k < s_2$ ,  $i_2$  是  $s_2$  的左指针, 且  $s_2 - 2^{i_2+1} < k \leq s_2 - 2^{i_2}$  (c). 由 (a) 和 (c) 得到  $s_1 + 2^{i_1} \leq s_2 - 2^{i_2}$  (d). 再由 (b)

和 (d) 得  $s_1 + 2^{i_1} + 2^{i_2} < s_1 + 2^{i_1+1} - 2^{i_2}$ . 因此  $i_1 > i_2$ .

(3)  $k < s_2 < s_1$ , 由算法  $\text{Lookup}(k)$  第 7 步可得  $s_1 - 2^{i_1+1} < k \leq s_1 - 2^{i_1}$  (a), 因为  $s_1$  的左指针  $i_1$  指向  $s_2$ , 所以  $s_1 - 2^{i_1+1} < k \leq s_1 - 2^{i_1}$ ,  $s_2 \leq s_1 - 2^{i_1}$  (b). 同理得到  $s_2 - 2^{i_2+1} < k \leq s_2 - 2^{i_2}$  (c). 由 (a) 和 (c) 得到  $s_1 - 2^{i_1+1} < s_2 - 2^{i_2}$  (d). 再由 (b) 和 (d) 得  $s_1 - 2^{i_1+1} < s_1 - 2^{i_1} - 2^{i_2}$ . 因此  $i_1 > i_2$ .

(4)  $s_2 < k < s_1$ , 由算法  $\text{Lookup}(k)$  第 7 步可得  $s_1 - 2^{i_1+1} < k \leq s_1 - 2^{i_1}$  (a), 因为  $s_1$  的左指针  $i_1$  指向  $s_2$ , 所以  $s_1 - 2^{i_1+1} < k \leq s_1 - 2^{i_1}$ ,  $s_2 > s_1 - 2^{i_1+1}$  (b). 由于  $k > s_2$ ,  $i_2$  是  $s_2$  的右指针, 且  $s_2 + 2^{i_2} \leq k < s_2 + 2^{i_2+1}$  (c). 由 (a) 和 (c) 得到  $s_2 + 2^{i_2} \leq s_1 - 2^{i_1}$  (d). 再由 (b) 和 (d) 得  $s_1 - 2^{i_1+1} + 2^{i_2} < s_1 - 2^{i_1}$ . 因此  $i_1 > i_2$ .

以上 4 种情况下均有  $i_1 > i_2$ , 同理可以证明  $i_2 > i_3 \dots i_{j-1} > i_j$ , 最后得出  $i_1 > \dots > i_j$ . 证毕.

由上面的证明可以看出  $\text{Lookup}(k)$  执行过程中的循环次数不会超过  $i_1$ , 而  $i_1 \leq \lceil \log_2(k) \rceil$ , 因此最多跳跃  $\lceil \log_2(k) \rceil$  步就可以找到  $k$ . 算法  $\text{Insert}(k)$  和  $\text{FindGeq}(k)$  时间复杂度也为  $O(\log_2(k))$ . 如果  $N$  为所有文章的总数, 以上算法的最大时间复杂度为  $O(\log_2(N))$ . 因此, 可以得出这种随机跳跃索引的查找性能是和原有跳跃索引性能是一样的.

**Insert( $k$ ) — Insert ID  $k$  into the random jump index**

```

1.  if (jump index is empty){
2.      Create a new jump index with a node containing  $k$ ;
3.      return
4.  }
5.   $s$  = the root node of jump index
6.  if ( $s = k$ ) {
7.      return  $k$  has existed
8.  }
9.  loop
10.     if ( $s < k$ ) {
11.         Find  $j \geq 0$ , such that  $s - 2^{j+1} < k \leq s - 2^j$ 
12.         if ( $s.lptr[j] = \text{NULL}$ ) {
13.             Create a new jump index node containing  $k$ ;
14.              $s.lptr[j] = k$ 's location //Set  $j$ th pointer of  $s$ 
15.             return DONE
16.         }
17.         else {
18.              $s'$  = the document ID at  $s.lptr[j]$ 
19.             {Follow the pointer to a new  $s$ }
20.              $s = s'$ 
21.             continue
22.         }
23.     }
24.     if ( $s > k$ ) {
25.         Find  $i \geq 0$ , such that  $s + 2^{i+1} < k \leq s + 2^i$ 
26.         if ( $s.rptr[i] = \text{NULL}$ ) {
27.             Create a new jump index node containing  $k$ ;
28.              $s.rptr[i] = k$ 's location //Set  $i$ th pointer of  $s$ 
29.             return DONE
30.         }
31.         else {
32.              $s'$  = the document ID at  $s.rptr[i]$ 
33.             {Follow the pointer to a new  $s$ }
34.              $s = s'$ 
35.             continue
36.         }
37.     }
38. end loop

```

**Lookup( $k$ ) — Find  $k$  in the random jump index**

```

1.   $s$  = the root node of jump index
2.  loop
3.      if ( $s = k$ ) {
4.          return FOUND
5.      }
6.      if ( $s > k$ ) {
7.          Find  $j \geq 0$ , such that  $s - 2^{j+1} < k \leq s - 2^j$ 
8.          if ( $s.lptr[j] = \text{NULL}$ ) {
9.              return NOT_FOUND
10.         }
11.         else {
12.              $s'$  = the document ID at  $s.lptr[j]$ 
13.             {Follow the pointer to a new  $s$ }
14.             assert  $s - 2^{j+1} < k \leq s - 2^j$ 
15.              $s = s'$ 
16.             continue
17.         }
18.     }
19.     if ( $s < k$ ) {
20.         Find  $i \geq 0$ , such that  $s + 2^{i+1} < k \leq s + 2^i$ 
21.         if ( $s.rptr[i] = \text{NULL}$ ) {
22.             return NOT_FOUND
23.         }
24.         else {
25.              $s'$  = the document ID at  $s.rptr[i]$ 

```

```

26.         {Follow the pointer to a new  $s$ }
27.         assert  $s + 2^{i+1} < k \leq s + 2^i$ 
28.          $s = s'$ 
29.         continue
30.     }
31. }
32. end loop

```

**FindGeqRec( $k, s$ ) — Function implementing FindGeq( $k$ )**

```

1.  if ( $s \geq k$ ) {
2.      if ( $s = k$ ) {
3.          return  $s$ ;
4.      }
5.      Find  $j \geq 0$ , such that  $s - 2^{j+1} < k \leq s - 2^j$ 
6.      if ( $s.lptr[j] \neq \text{NULL}$ ) {
7.           $t$  = document ID at  $s.lptr[j]$ 
8.          assert  $s - 2^{j+1} < t \leq s - 2^j$ 
9.           $res = \text{FindGeqRec}(k, t)$ 
10.         //Recursively call FindGeqRec() by following the  $lptr$ 
11.         if ( $res \neq \text{NOT\_FOUND}$ ) {
12.             assert  $s - 2^{j+1} < res \leq s - 2^j$ 
13.             return  $res$ 
14.         }
15.     }
16.     //No number  $\geq k$  could be found by following  $j$ th  $lptr$ ,
17.     //return the FindGeqRec( $k, *lptr$ ) of the first non-null  $lptr$ 
18.      $j = j - 1$ 
19.     while ( $j \geq 0$ ) {
20.         if ( $s.lptr[j] \neq \text{NULL}$ ) {
21.              $t$  = document ID at  $s.lptr[j]$ 
22.             assert  $s - 2^{j+1} < t \leq s - 2^j$ 
23.             return FindGeqRec( $k, t$ )
24.         }
25.          $j = j - 1$ 
26.     }
27.     return NOT_FOUND
28. }
29. if ( $s < k$ ) {
30.     Find  $i \geq 0$ , such that  $s + 2^{i+1} < k \leq s + 2^i$ 
31.     if ( $s.rptr[i] \neq \text{NULL}$ ) {
32.          $t$  = document ID at  $s.rptr[i]$ 
33.         assert  $s + 2^{i+1} < t \leq s + 2^i$ 
34.          $res = \text{FindGeqRec}(k, t)$ 
35.         //Recursively call FindGeqRec() by following the  $rptr$ 
36.         if ( $res \neq \text{NOT\_FOUND}$ ) {
37.             assert  $s + 2^{i+1} < res \leq s + 2^i$ 
38.             return  $res$ 
39.         }
40.     }
41.     //No number  $\geq k$  could be found by following  $i$ th  $lptr$ ,
42.     //return the FindGeqRec( $k, *rptr$ ) of the first non-null  $rptr$ 
43.      $i = i + 1$ 
44.     while ( $i < \log_2[N]$ ) {
45.         if ( $s.rptr[i] \neq \text{NULL}$ ) {
46.              $t$  = document ID at  $s.rptr[i]$ 
47.             assert  $s + 2^{i+1} < t \leq s + 2^i$ 
48.             return FindGeqRec( $k, t$ )
49.         }
50.          $i = i + 1$ 
51.     }
52.     return NOT_FOUND
53. }

```

**FindGeq( $k$ ) — Find number  $\geq k$** 

```

1.  return FindGeqRec( $k$ , the root node in the random jump index)

```

图 4 Random Jump Index 随机跳跃索引的基本算法

**3.3 随机跳跃索引的可信赖性**

跳跃索引的优势在于它保证了索引中数据的可信赖性,即数据不能被改动或隐藏,数据的路径唯一且固定不变.而这些特性是由跳跃索引的两条性质保证的,随机跳跃索引具有和跳跃索引相同的可信赖性,为了说明这一点,下面证明它也具有跳跃索引的两条性质.

**命题 2.** 当文章 ID 插入到倒排表的随机跳跃

索引中之后,该文章 ID 总是可以被成功地查找到.

证明. 在执行 Insert() 时,数据是被写在 WORM 设备上的,所以数据在写入之后是不会被改动的.上一节中的序列  $i_1 > \dots > i_j$  对 Lookup() 是唯一的,因此插入的数据总是可以被找到. 证毕.

**命题 3.** 设  $v$  是一个在倒排表中的文章 ID. 如果  $k \leq v$ , 那么 FindGeq( $k$ ) 返回的值不会大于  $v$ .

证明. 设  $s_0$  是随机跳跃索引的根节点. 此时有

3 种情况: (1)  $s_0 < k \leq v$ ; (2)  $k \leq v < s_0$ ; (3)  $k < s_0 < v$ . 在分别证明之前, 设  $v$  在索引中的跳跃指针序列为  $j_1, \dots, j_o$  (例如, 执行  $\text{Lookup}(v)$  所得序列  $j_1, \dots, j_o$ ). 同样, 设  $\text{FindGeq}(k)$  返回值  $l$  的序列为  $i_1, \dots, i_p$ .

(1)  $s_0 < k \leq v$ , 首先证明  $i_1 \leq j_1$ . 在第一次调用  $\text{FindGeqRec}()$ , 由第 28 步得到  $i \leq j_1$ , 因为  $k \leq v$ . 如果  $i$  通过了第 29 和第 33 步的检查, 那么  $i_1 = i, i_1 \leq j_1$ . 如果上两步都为假, 那么  $\text{FindGeqRec}()$  将执行第 39 步. 由于  $v$  在索引中,  $s.rptr[i_1]$  不会为 NULL, 所以  $i \leq j_1, i_1 = i, i_1 \leq j_1$ . 这时又有两种情况: ①  $i_1 < j_1$  这种情况下,  $l < v$  因为  $l < s_0 + 2^{i_1+1} \leq s_0 + 2^{j_1} \leq v$ . ②  $i_1 = j_1$ . 此时直接调用  $\text{FindGeqRec}()$ .

(2)  $k \leq v < s_0$ , 首先证明  $i_1 \geq j_1$ . 第一次调用  $\text{FindGeqRec}()$ , 由第 5 步得到  $j \geq j_1$ , 因为  $k \leq v$ . 如果  $i$  通过了第 6 和第 11 步的检查, 那么  $i_1 = j, i_1 \geq j_1$ . 如果上两步都为假, 那么  $\text{FindGeqRec}()$  将执行第 17 步. 由于  $v$  在索引中,  $s.lptr[j_1]$  不会为 NULL, 所以  $j \geq j_1, i_1 = j, i_1 \geq j_1$ . 这时又有两种情况: ①  $i_1 > j_1$  这种情况下,  $l < v$  因为  $l \leq s_0 - 2^{i_1} \leq s_0 - 2^{j_1+1} < v$ . ②  $i_1 = j_1$ . 此时直接调用  $\text{FindGeqRec}()$ .

(3)  $k < s_0 < v$ , 由 (2) 得  $\text{FindGeqRec}(k)$  返回值  $l < s_0$ , 又因为  $s_0 < v$ , 所以  $l < v$ .

以上 3 种情况下都有  $\text{FindGeqRec}(k) \leq v$ .

证毕.

命题 3 保证了在连接两个倒排表时没有文章 ID 会被隐藏. 如果文章 ID  $d$  在两个倒排表中都出现, 执行 Zigzag Join 算法时从表中最小的值开始调用  $\text{FindGeq}()$ , 性质 3 可以保证在返回  $d$  之前,  $\text{FindGeq}()$  返回的值不会大于它. 换言之,  $d$  最终被两个倒排表中的  $\text{FindGeq}()$  返回, 并且出现在最终结果当中. 这两条性质保证了随机跳跃索引与跳跃索引具有相同的可信赖性.

### 3.4 广义的随机跳跃索引

索引的深度是影响查找速度的主要因素. 在文章 ID 非常多时, 例如  $2^{32}$ 、 $2^{64}$  甚至更大, 减小索引深度能明显减少查找次数. 为此, 可以用  $B$  作为底数,  $B \geq 2$ , 不仅仅以 2 为底数. 每个节点的左右跳跃指针个数为  $(B-1)\log_B(N)$ , 跳跃指针用参数  $(i, j)$  表示, 其中  $0 \leq i < \log_B(N)$ ,  $1 \leq j < B$ . 如果  $s_0$  为当前节点, 右指针  $(i, j)$  指向  $s_1$ , 那么  $s_0 + jB^i \leq s_1 < s_0 + (j+1)B^i$ ; 左指针  $(i, j)$  指向  $s'_1$ , 那么  $s_0 - (j-1)B^i < s'_1 \leq s_0 - jB^i$ . 图 5 中的例子是以 3 为底数,  $B=3$ , 文章 ID 总数  $N=60$ . 节点 13 的右跳跃指针  $(1, 1)$  指向 16, 因为  $13 + 1 \times 3^1 \leq 16 < 13 + (1+1) \times 3^1$ , 同理右跳跃指针  $(2, 2)$  指向 53, 因为  $13 + 1 \times 3^2 \leq 53 < 13 +$

$2 \times 3^3$ ; 13 的左跳跃指针  $(1, 2)$  指向 7, 因为  $13 - 1 \times 3^2 < 7 \leq 13 - 2 \times 3^1$ .

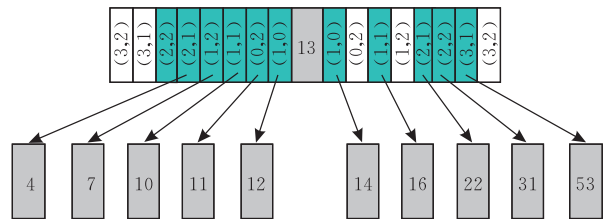


图 5 广义的随机跳跃索引

用广义的随机跳跃索引,  $\text{Insert}()$ 、 $\text{Lookup}()$  和  $\text{FindGeqRec}()$  算法的时间复杂度均为  $O(\log_B(k))$ .

## 4 性能评估

本节首先对 Random Jump Index 的索引大小做出分析; 然后比较 Random Jump Index 与 Jump Index 两种索引在建立时, 随机插入与严格单调递增插入的比较次数, 最后对这两种索引的查找性能作出比较. 影响这些性能的关键变量有底数  $B$ 、文章 ID 的总数  $N$ .

在对总数为  $N$  的文章 ID 集合建立随机跳跃索引时, 索引的大小可由下面的公式表示:  $\text{Size}_{\text{Index}} = N(2((B-1)\log_B(N) \times \text{size}_p) + \text{size}_e)$ , 其中  $\text{size}_p$  为一个跳跃指针的大小, 通常为 4 个字节,  $\text{size}_e$  为节点元素值的大小, 通常为 4 或 8 个字节. 索引大小  $\text{Size}_{\text{Index}}$  与  $B$  和  $N$  成正比. 在实际应用中对不同的  $N, B$  可以根据查找的效率与存储效率作相应调整.

下面对比二者建立索引时的效率. Random Jump Index 可以连续处理多个文章 ID 序列, 而 Jump Index 只能处理一个文章 ID 序列, 如果有新的序列到达, Jump Index 只能将原有序列与新序列按照递增顺序合并后, 毁掉以前的索引重新建立索引; 对于 Random Jump Index 则只需要将新序列插入到原有索引即可. 显然, 在处理多序列时 Random Jump Index 要大大优于 Jump Index. 现在只对比处理单个序列时二者的复杂度: 设  $l_0, \dots, l_{N-1}$  为随机序列, Random Jump Index 继续将该序列的  $N$  个元素依次插入到索引中即可, 总的比较次数为  $\sum_{k=1}^N \log_B(k)$ ; 对于 Jump Index 则需要将该序列排序后再依次插入索引, 当该序列有序时它的比较次数也是  $\sum_{k=1}^N \log_B(k)$ , 当序列无序时调整次序需要  $N \log_B(N)$  次比较, 所以 Jump Index 对该序列建立索引总比较次数为

$$\frac{1}{N!} \cdot \sum_{k=1}^N \log_B(k) + \left(1 - \frac{1}{N!}\right) \left(N \log_2(N) + \sum_{k=1}^N \log_B(k)\right).$$
可见对单个序列建立索引时, Random Jump Index 效率也高于 Jump Index.

最后, 对比两者的查找效率. Jump Index 查找时, 根节点始终为序列中最小的节点,  $l_{\min} = 1$ , 最大比较次数为  $\log_B(N)$ . Random Jump Index 查找时比较次数与根节点的值  $l_{\text{root}}$  有关, 而  $l_{\text{root}}$  等于该序列的期望值,  $l_{\text{root}} = E\{l_0, \dots, l_{N-1}\} = \lfloor N/2 \rfloor$ ; Random Jump Index 查找时的平均比较次数为  $\log_B \lfloor N/2 \rfloor$ . 命题 1 决定了二者的最坏查找效率相等, 但平均查找效率 Random Jump Index 要优于 Jump Index.

5 实验结果

我们用 MATLAB 进行了仿真实验, 针对不同

的文章 ID 总数  $N$ , 不同的底数  $B$ , 将二者的索引大小、建立索引的效率以及查找效率做了对比. 其中  $B = \{2, 4, 6, 8, 16, 32, 64\}$ ,  $N = \{2^{16}, 2^{32}\}$ .

图 6(a) 中是两者索引大小的对比结果. 实验中  $N = 2^{32}$ ,  $size_p = 4\text{Byte}$ ,  $size_e = 4\text{Byte}$ . 由于随机跳跃索引多了左跳跃指针, 所以其索引要比 Jump Index 大. 对同样的  $N$ , 索引大小随着  $B$  的增大而增加, 可见在降低索引深度的同时会增加索引大小. 图 6(b) 是对两种索引建立的效率比较. 这里  $N = 2^{16}$ , 图 6(b) 中 Jump Index 曲线是对单个序列建立索引时的最小比较次数, Random Jump Index 曲线是对单个序列建立索引时的最大比较次数, 可见, 后者最大比较次数小于前者的最小比较次数. 对相同的  $N$ , 二者的比较次数都随着  $B$  的增加在不断减小. 图 6(c) 是对查找效率的比较. 实验结果显示二者的查找效率是非常接近的, 而且随着  $B$  的增加查找效率也越来越高.

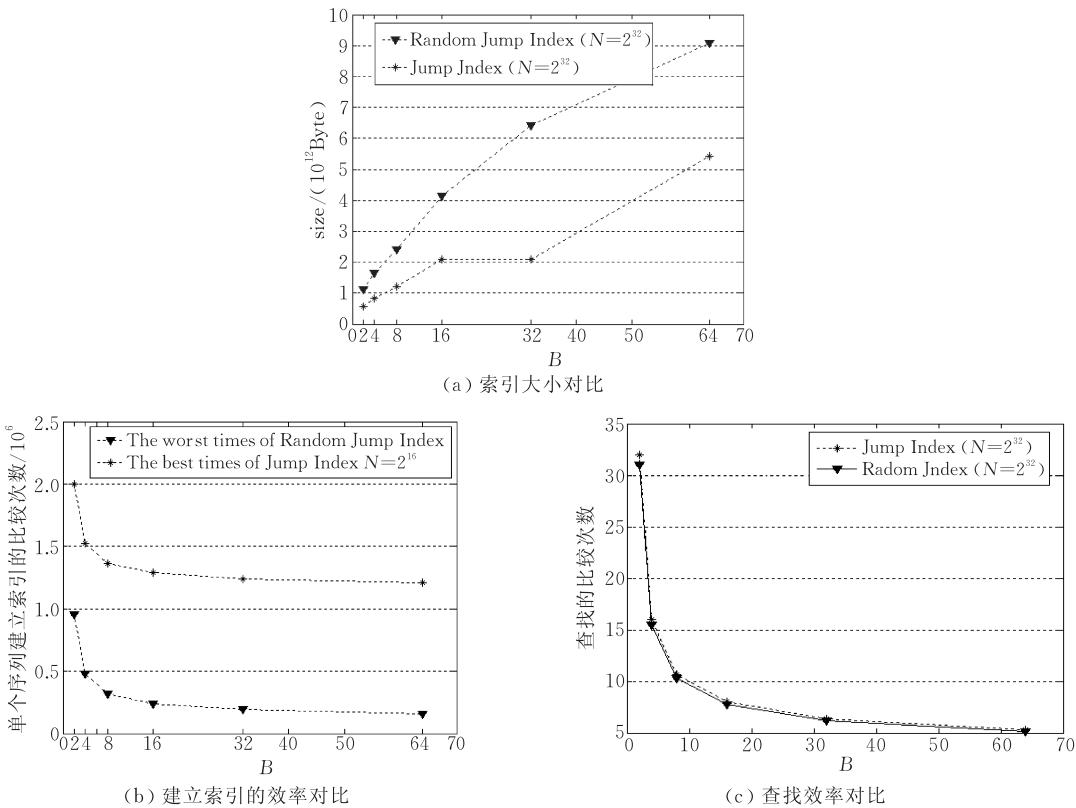


图 6 随机跳跃索引的性能实验结果

6 结束语

本文提出的 Random Jump Index 随机跳跃索引, 不再受限于 Jump Index 跳跃索引严格单调递增的约束条件; 通过在原有跳跃索引结构上加入左侧

跳跃指针的方法, 索引节点可以根据待加入节点值的大小将其纳入自己的左侧或右侧跳跃指针, 使随机跳跃索引可以支持多个序列的随机插入, 显著地提高了 Jump Index 建立索引的效率; 它与后者有着相同的查找效率; 随机跳跃索引中的每一个节点到根节点的路径固定且唯一, 保证了索引的可信赖性.

仿真实验结果表明:其建立索引的效率明显优于后者,两者的查找效率也非常接近。

## 参 考 文 献

- [1] Zhu Q, Hsu W W. Fossilized index: The linchpin of trustworthy non-alterable electronic records//Proceedings of the ACM International Conference on Management of Data, Baltimore, USA, 2005: 395-406
- [2] Mitra Soumyadeb, Hsu Windsor W, Winslett Marianne. Trustworthy keyword search for regulatory-compliant records retention//Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, 2006: 1001-1012
- [3] Pavlou Kyriacos, Snodgrass Richard T. Forensic analysis of database tampering//Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, Chicago, 2006: 109-120
- [4] Joukov Nikolai, Papaxenopoulos Harry, Zadok Erez. Secure deletion myths, issues, and solutions//Proceedings of the 2nd ACM Workshop on Storage Security and Survivability, Alexandria, 2006: 61-66
- [5] Kim M-S, Whang K-Y, Lee J-G, Lee M-J. n-Gram/2L;

- A space and time efficient two-Level n-Gram inverted index structure//Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, 2005: 325-336
- [6] Fontoura M F, Neumann A, Rajagopalan S, Shekita E, Zien J. High performance index build algorithms for intranet search engines//Proceedings of the 30th International Conference on Very large Data Bases, Toronto, Canada, 2004: 1122-1133
- [7] Silverstein Craig, Marais Hannes, Henzinger Monika, Moricz Michael. Analysis of a very large web search engine query log. ACM SIGIR Forum, 1999, 33(1): 6-12
- [8] Miller Ethan, Shen Dan, Liu Junli, Nicholas Charles. Performance and scalability of a large-scale n-gram based information retrieval system. Journal of Digital Information, 2000, 1(5): 1-25
- [9] Witten Ian H, Moffat Alistair, Bell Timothy C. Managing Gigabytes: Compressing and Indexing Documents and Images. 2nd Edition. San Francisco: Morgan Kaufmann Publishers, 1999
- [10] Garcia-Molina Hector, Ullman Jeffrey D, Widom Jennifer D. Database System Implementation. Upper Saddle River, New Jersey: Prentice-Hall, 2000



**LIU Feng-Chen**, born in 1982, Ph.D. candidate. His research interests include information retrieval, retrieval algorithm and intelligent algorithm.

**HUANG He**, born in 1972, Ph. D. , associate professor. His research interests include computer network and information security.

## Background

This research is partly supported by the Science and Technology Program of China under grant No. A2120061061, Project of the Shanghai Committee of Science and Technology (No. 08JC1400100), Shanghai Talent Developing Foundation (No. 001), Specialized Foundation for Excellent Talent from Shanghai.

In the recent few years, the trustworthy problem in write once read many devices is raised by many researches. To address this issue some researchers proposed a kind of indexes with trustworthy, such as fossilized index and jump index. The former prohibits some manipulation on index after index has been built to keep the trustworthy, later ensures that every node in the index has the unique and permanent path from root node to make sure the index is trustworthy. Although jump index is a better way to handle this problem, it still has difficulty with building index for sequences without orders. Because it requires that sequences must be strict-

**LIU Qing-Wen**, born in 1966, Ph. D. , lecturer. His research interests include cluster, security of database and distributed information system.

**DING Yong-Sheng**, born in 1967, Ph. D. , professor, Ph.D. supervisor. His research interests include intelligent systems, network intelligence, DNA computing, artificial immune systems, bio-network, bioinformatics, digitalized textile and garment, intelligent decision making and analysis.

ly monotonically increased, this requirements makes it is impossible for inserting random sequences. The authors' main object is to design an index which can insert and build index for random sequences. How to maintain the trustworthy and efficiency of query is also important. In this paper, authors proposes a solution that adding left pointers on the current node in index as the counterpart of the right pointers in jump index to record nodes with less value than current node, while right jump pointers are used to record with higher value. Therefore the candidate nodes in random sequences can be indexed into one of the left or right pointers depending on its value. Also, this changing on structure does not change the unique and permanent path from root node to any node in the index, which keeps the trustworthy. Compared with jump index, new structure makes building index and inserting records become more flexible and efficient.