CN43-1258/TP
ISSN 1007-130X

计算机工程与科学
COMPUTER ENGINEERING & SCIENCE

2012 年第 34 卷第 2 期
Vol. 34, No. 2, 2012

文章编号:1007-130X(2012)02-0019-06

一种基于树形结构的布鲁姆过滤器*
Bloom Filters Based on the Tree Structure

程 聂,黄 昆,苏 欣,张大方
CHENG Nie, HUANG Kun, SU Xin, ZHANG Da-fang
(湖南大学软件学院,湖南 长沙 410082)
(School of Software, Hunan Universty, Changsha 410082, China)

摘 要:本文提出一种基于多层次结构的树形布鲁姆过滤器 TBF。多层次结构是近年来布鲁姆过滤器及相关数据结构研究的热点。这一结构使得多层次的存储方式得以实现,减轻了片上存储的负担,而且也加快了片上查找的速度。TBF 是针对 BloomingTree 算法存在的缺陷所改进的一种更高效的算法,它能够在低于 CBF 的空间需求的条件下实现与 CBF 相同的功能。实验证明:与 BloomingTree 算法相比,TBF 能够有效地解决 BloomingTree 算法在逻辑索引时的错误问题,而且比 BloomingTree 算法时间上更加高效:在层数不变假阳性相同条件下,查询时间平均提高 13.4%;在假阳性不变层数相同条件下,插入时间平均提高 17.9%,删除时间平均提高 12%。

Abstract: This paper presents a multi-level structure called the Tree-based Bloom Filter (TBF). Multi-level structure is the hot spots of Bloom filters and related data structure research in recent years. This structure achieves multiple levels of storage and reduces the burden of on-chip memory, but it also accelerates the speed of on-chip search. TBF is a more efficient algorithm which is the improvement based on the drawbacks of the BloomingTree algorithm, and TBF can reduce the conditions of the space requirements and achieve the same function of CBF under the same conditions. Our experiments show that compared with the BloomingTree algorithm, the TBF algorithm can effectively solve the index error in the logic problem of the BloomingTree algorithm, and show more time efficiency: under the conditions of the same false positiveness and unchanged layers, the query time improve on an average of 13.4%; under the conditions of the fixed false positiveness and the same layer changes, the time of insertion improves on an average of 17.9%, and 12% average improve the time of deletion.

关键词:布鲁姆过滤器;多层次结构;数据结构;集合元素查询
Key words: Bloom filter; multi-level structure; data structure; set element search
doi: 10.3969/j.issn.1007-130X.2012.02.004
中图分类号:TP393 **文献标识码**:A

1 引言

随着计算机科学的不断进步特别是网络技术的飞速发展,网络规模的增长日新月异,这就使得数据库和网络中数据规模也在不断地增大,网络资源的合理利用成为计算机科学研究的新一轮挑战。在网络中的数据流日趋庞大的今天,如何合理地表示和查找一个数据流的相关信息从而对它进行正确的处理,成为了计算机应用领域的核心问题。

布鲁姆过滤器 BF[1]是一种高效的数据存储结构,它主要用于在存在一定的假阳性概率下对数据

* 收稿日期:2010-05-20;修订日期:2010-10-26
基金项目:国家发改委信息安全专项(发改办高技[2009]1886 号文);湖南省科技计划重点项目(2009JT1018)
通讯地址:410082 湖南省长沙市湖南大学软件学院
Address: School of Software, Hunan Universty, Changsha, Hunan 410082, P. R. China

成员的查找。同时,布鲁姆过滤器也是一种用于信息表示的精简数据结构,它通过一个由“0”与“1”组成的比特位串来表示数据元素集合,并能够支持随机的哈希查找。布鲁姆过滤器算法的实质就是通过 K 个哈希函数将原本需要存储的数据元素集合传换成仅由一定比特位就能够表示的位串,它改善了传统查询算法(如哈希查询、树形查找)中由数据大小决定存储空间的不利局面,在数据规模飞速膨胀的今天其意义显得尤为突出。

同时,布鲁姆过滤器是允许存在一定假阳性的,所谓假阳性即将不属于集合的元素误判断成属于集合中。出现这一误判的原因是因为布鲁姆过滤器中的每一个单独的比特位并不只是仅表示数据集合中的某一个元素,而是有可能与集合中的其它元素也存在关联,所以当我们改变布鲁姆过滤器中的比特位时,对集合中其它元素的正确表示与否也可能会产生影响,从而导致误判的发生。由此可知,布鲁姆过滤器的缺陷就是它并不支持删除操作。基于以上情况,我们需要在位串存储空间和误判率之间根据实际需要做出最优选择。

计数布鲁姆过滤器(CBF)[2]正是针对布鲁姆过滤器不能支持删除操作这一缺陷而设计的。CBF 与 BF 同样存在假阳性的问题,它是由固定大小的计数器(Couter)来替代布鲁姆过滤器中的比特位,通过对计数器的增减来表示元素的插入与删除。由于计数器的长度决定了 CBF 所需要的存储空间的大小,如果计数器过长,则 CBF 空间需求过大,可能导致空间的浪费;如果计数器过短,则插入元素时可能出现数据溢出的现象。

本文提出的 TBF(Tree-based Bloom Filters,简称 TBF)是一种多层的数据结构,TBF 能够实现与计数布鲁姆过滤器(CBF)同样的功能,根据实际情况确定所需要的假阳性和数据溢出的概率。TBF 通过为置“1”的比特位增加由固定大小的比特位所组成的叶子结点的方式实现分层的数据结构,并通过一定的算法使得各层之间产生关联,在纵向上是类似于二叉树的逻辑结构,我们称它为“树形(Tree-based)”。与 CBF 相比,TBF 在空间存储方面有更好的空间效率。与 BloomingTree 算法相比,TBF 在操作时间方面更加高效。

本文介绍了布鲁姆过滤器和计数布鲁姆过滤器算法及其相关研究——其它数据集合成员查询数据结构重要的改进;介绍了 Tree-based Bloom Filter 算法的数据结构、具体工作原理及实验结果;最后对 Tree-based Bloom Filter 算法进行了总结。

2 相关工作

存储空间的利用率,数据集查询和更新的效率这两个方面是布鲁姆过滤器及其相关研究的重点关注所在,特别是在网络带宽资源依然宝贵,网络传输数据越来越大,网络负担越来越重,线上实时处理速度要求越来越高的今天。尽管提供存储空间的硬件设施随着技术的发展出现了一定程度上的富余,但是一个高效的数据结构的研究便于空间和时间的合理利用,同时也能够达到减少计算机处理器对片外存储器的操作,提高工作效率的目的。本文所提出的树形布鲁姆过滤器算法也是基于这一需求所提出的,是为了更好地满足空间和时间的高效性而针对布鲁姆过滤器算法所进行的研究。

布鲁姆过滤器是基于网络发展的这一现状所提出的高效的查询算法,但由于布鲁姆过滤器不支持元素删除操作的局限性,计数布鲁姆过滤器 CBF 被提出以解决之一问题。CBF 使用长度为 m 的计数器 Counter 来代替布鲁姆过滤器中的比特位,以记录元素映射到向量 U 中的情况,通过计数器的增减来表示元素的插入与删除。但是,当所取的 m 过小时 CBF 存在数据溢出的问题,而且使用固定比特的计数器会造成存储空间的浪费。

各种各样的计数布鲁姆过滤器算法的改进被研究者提出,其中比较重要的包括:光谱布鲁姆过滤器(Spectral Bloom Filters 简称 SBFs)[3]是标准布鲁姆过滤器结构的拓展,它能够对每一个单独对象的多样性(Multiplicity)——集合中的元素出现存在多个副本的可能——进行评估。SBF 动态地改变计数器的大小,以便使得所用的比特数最少。但是,SBF 会出现额外的惰性比特位,另外复杂的索引结构也成为了它的缺陷。动态计数过滤器(Dynamic Count Filters 简称 DCFs)[4]需要用到两个向量(Vector),第一个向量是由固定大小的计数器组成的 CBF,第二个向量由能够动态改变大小的计数器组成,这些计数器用来跟踪记录第一个向量中每一个对应的计数器出现数据溢出的次数,之所以需要能够动态改变大小的计数器是为了避免出现饱和,我们称这第二个向量为溢出向量(Overflow Vector)。这一结构的不足之处是每一次操作都可能导致溢出向量元素的改变,这就需要重建整个数据结构。而且在溢出向量中保持同样大小的过滤器空间会导致大量的空间多余开销。D-left

计数布鲁姆过滤器(D-left CBFs 简称 dl-CBFs)[5]是计数布鲁姆过滤器基于 d-left 哈希和指纹的简单备选假设,他们需要的空间更小,在相同假阳性概率下有时可以达到两倍的空间节省。但是,dl-CBF 存在着计数器溢出以及数据结构中的某些数据位(bin)可能会需要额外的指纹开销的潜在不足之处。此外,还有压缩布鲁姆过滤器(Compressed Bloom Filter)、拆分型布鲁姆过滤器(Split Bloom Filter)、可扩展布鲁姆过滤器(Scalable Bloom Filter)、多维布鲁姆过滤器和分档布鲁姆过滤器(Basket Bloom Filter)等诸多算法,这些布鲁姆过滤器扩展算法为布鲁姆过滤器查询算法的研究发展做出了大量贡献。

3　Tree-based Bloom Filter 算法

3.1　布鲁姆过滤器算法介绍

布鲁姆过滤器 BF 用一个比特位串来表示集合 S,集合 S 中含有 n 个元素$\{s_1,s_2,s_3,\cdots,s_n\}$,这 n 个元素由 k 个哈稀函数$\{h_1,h_2,h_3,\cdots,h_k\}$映射到长度为 m 的比特位串向量 U 中,哈希函数相互独立且函数的取值范围为$\{0,1,2,\cdots,m\}$,U 中比特位定义为 $u[i]$,即 $u[1],u[2],u[3],\cdots,u[m]$,当有元素映射到 U 中的第 i 位 $u[i]$时,将该位置“1”,其它位默认置“0”哈希后不变。最后得到的向量 U 就是 BF 的最终表示形式。

当需要查询某个元素 x 是否属于某一数据集合 S 时,需要使用 k 个哈稀函数 $h_1,h_2,h_3,\cdots,h_k$ 将 x 映射到向量 U,确认是否所有的 $u[h_i]$的值均等于 1,若是,则元素 x 是集合 S 内的元素;反之则不然。这种判断机制可能会出现误判,即产生假阳性。

假阳性即元素不在集合 S 中的概率用公式(1)表示:

$$\varepsilon=(1-(1-1/m)^{kn})^k\approx(1-e^{-kn/m})^k \quad (1)$$

为了满足给定的假阳性概率,存储空间最小应该满足公式(2):

$$m=\frac{1}{\ln 2}kn\approx 1.44kn \quad (2)$$

在这一计算中,所有向量 U 中的元素 $u[1],u[2],u[3],\cdots,u[m]$每一位等于“1”或等于“0”的概率 p 大约为 1/2。

3.2　BloomingTree 算法介绍及其缺陷

BloomingTree 算法[6]的主要思想是通过为置“1”的比特位增加由比特位组成的叶子结点的方式实现分层的数据结构,并且每一层的布鲁姆过滤器都能够呈现“树形”结构中相应叶子结点的层次特性,从而达到比 CBF 空间更加高效——空间需求减少了 39.2%,假阳性更低的目的。

BloomingTree 是由多层的布鲁姆过滤器构成的,除了第一层基本的布鲁姆过滤器之外的各层布鲁姆过滤器都与第一层布鲁姆过滤器存在联系,这种联系构成了类似于二叉树的结构,而在 BloomingTree 的最后一层则采用由计数器组成的计数布鲁姆过滤器。以 4 层 BloomingTree 为例来描述 TBF 的结构,如图 1 所示。

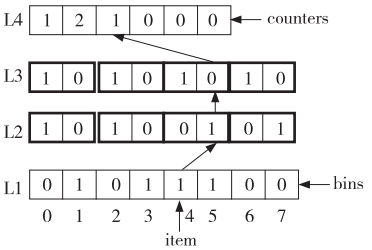

图 1　BloomingTree 基本结构

第一层是带有 k 个哈希函数 $h_1,h_2,h_3,\cdots,h_k$ 并由 m 个比特位所组成的布鲁姆过滤器;第二、三层由叶子结点所组成,这些叶子结点的大小由控制函数 *bitnode* 决定,每个结点对应着第一层置“1”的比特位;第四层也就是最后一层则是由计数器组成的计数布鲁姆过滤器,其大小为 m 比特。

BloomingTree 算法在逻辑索引方面的缺陷如图 1 中所标记的第 5 个比特位所示,当第 4 个比特位置“1”后,因为 BloomingTree 算法必须根据所查询的比特位的前一位是“1”还是“0”来决定下一层叶子结点置“1”的位置,而现在第 5 个比特位的前一个比特位由“0”变成了“1”,但是第 5 个比特位的下一层叶子结点并没有发生相应的变化,这种情况的发生必然会导致查询时出现误判,特别是当数据集规模增大,插入、删除操作增多时,这样的情况将会频繁出现。

3.3　Tree-based Bloom Filters 算法介绍

树形布鲁姆过滤器 TBF 的主要思想是在延用 BloomingTree 算法基本结构的基础上,采用新的算法改进了它的查询、插入和删除操作的步骤,这样不仅能够有效地解决 BloomingTree 算法在上述操作中所存在的逻辑索引时的错误,而且也有效地改进了在多层次结构中需要多次重复计算的缺点。在多层次结构中特别是当层数 L 逐渐增大时,TBF 在能够保持 BloomingTree 在空间方面优势

地位的同时,还有效地提高了时间的运算效率。下面就以 $bitnode=1$ 为例,具体地介绍 TBF 的查询、插入和删除操作。

在一个 TBF 中查询元素 a 是否属于某一个数据集合 U 如图 2 所示,需要经过 k 个哈希函数中的任意一个或多个哈希后插入布鲁姆过滤器中,我们需要通过同样的哈希函数找到第一层布鲁姆过滤器中元素的正确位置 $u_1[h(a)](L=1)$,然后需要找到在树形结构中相应的叶子结点并且返回相应的查询结果。

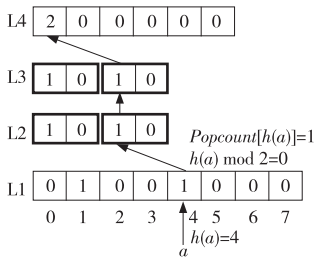

图 2 树形布鲁姆过滤器查询算法

查询的具体步骤如下:

(1)元素 a 经过哈希确定元素在第一层布鲁姆过滤器的位置 $u_1[h(a)]$,如果 $u_1[h(a)]$ 的值等于 1,则继续查询 $L+1$ 层;如果 $u_1[h(a)]$ 的值不等于 1,则返回元素 a 不属于数据集合 U 的信息。

(2)使用 *Popcount* 函数找出第一层布鲁姆过滤器中位置 $u_1[h(a)]$ 之前所有“1”的个数 x,从而找到接下来 $L-1$ 层中需要查询的叶子结点的位置,同时用得到的哈希值 $h(a)$ 对 $2^{bitnode}$(此时 $bitnode=1$)求模得到值 p,指示叶子结点中置“1”的位置 $Leaf[h(a)]$(当 $bitnode=1$ 时,$Leaf[h(a)]=0$ 表示叶子结点左结点,$Leaf[h(a)]=1$ 表示叶子结点右结点)。

(3)在 $L=2,3,4,\cdots,L-1$ 层中根据得到的参数 x 和 p 索引得到相应查询位置 $u_L[h(a)]$,如果 $u_L[h(a)]$ 等于 1,则继续查询 $L+1$ 层;如果 $u_L[h(a)]$ 不等于 1,则返回元素 a 不属于数据集合 U 的信息。

(4)在最后一层即第 L 层,使用 *Popcount* 函数找出第 $L-1$ 层中查询位置 $u_{L-1}[h(a)]$ 之前“1”的个数 y,根据 y 找到第 L 层相应的计数器 *Counter*,如果 *Counter* 不等于 0,则返回元素 a 属于数据集合 U 的信息;如果 Counter 等于 0,则返回元素 a 不属于数据集合 U 的信息。

多层次的查询能够保证比计数布鲁姆过滤器更高的正确匹配率,但是 TBF 的关键还是在于插入与删除操作是否能够在多层次索引结构中插入或者删除正确的叶子结点,以便保证查询的正确匹配率。TBF 的插入与删除操作如图 3 所示,结合图 2 来描述 TBF 的插入操作的具体步骤。图 3 表示 TBF 的插入操作,在插入一个属于数据集合 U 的元素 b 时,首先元素 b 经过哈希函数哈希之后得到哈希值 $h(b)$,根据哈希值 $h(b)$ 找到第一层布鲁姆过滤器中应该置“1”的位置并将其置“1”;其次使用 *Popcount* 函数找出第一层布鲁姆过滤器中位置 $u_1[h(b)]$ 之前所有“1”的个数 x,从而找到接下来 $L-1$ 层中需要插入叶子结点位置并插入叶子结点,同时用得到的哈希值 $h(b)$ 对 $2^{bitnode}$ 求模得到值 p 指示叶子结点中置“1”的位置 $Leaf[h(b)]$;然后在 $L=2,3,4,\cdots,L-1$ 层中根据得到的参数 x 索引得到叶子结点应该插入的位置并插入叶子结点,再根据参数 p 索引得到叶子结点中应该置“1”的位置 $u_L[h(b)]$ 并将 $u_L[h(b)]$ 置“1”;在最后一层即第 L 层,使用 *Popcount* 函数找出第 $L-1$ 层中查询位置 $u_{L-1}[h(b)]$ 之前“1”的个数 y,根据 y 找到第 L 层相应的计数器 *Counter* 并令计数器中的值加 1。

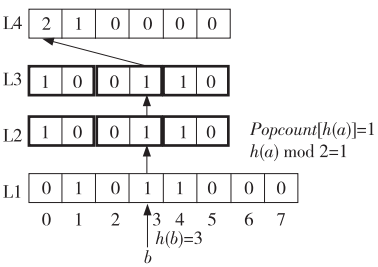

图 3 TBF 插入过程

TBF 的删除操作是插入操作的逆向过程,以删除元素 c 为例,在进行删除操作时,首先元素 c 经过哈希函数哈希之后得到哈希值 $h(c)$,根据哈希值 $h(c)$ 找到第一层布鲁姆过滤器中应该置“0”的位置;其次使用 *Popcount* 函数找出第一层布鲁姆过滤器中位置 $u_1[h(b)]$ 之前所有“1”的个数 x,从而找到接下来 $L-1$ 层中需要删除叶子结点位置;再在最后一层即第 L 层,使用 *Popcount* 函数找出第 $L-1$ 层中查询位置 $u_{L-1}[h(c)]$ 之前“1”的个数 y,根据 y 找到第 L 层相应的计数器 *Counter* 并令计数器中的值减 1;然后根据参数 p 删除 $L=L-1,L-2,L-3,\cdots,2$ 层中相应的叶子结点;最后根据哈希值 $h(c)$ 将第一层布鲁姆过滤器相应位置置“0”。

3.4 理论分析

比较 TBF 和 BloomingTree 算法在时间效率方面的性能,以一次插入操作为例,当使用 k 个哈

希函数且 BloomingTree 层数为 L 时,任意元素插入 BloomingTree 所需要的时间分别为:经过一次哈希的时间 HASH;每一层使用 *Popcount* 函数的时间 $POPCOUNT$;需要对每一层插入操作时需要移动其余比特位的时间 $SHIFT$;每一层相应比特位置"1"的时间 $BITSET$。而每一个元素需要经过 k 个哈希函数 K 次哈希后才能完成一次插入操作,由此可以得到总时间为 $K[HASH+L(POPCOUNT+SHIFT+BITSET)]$。

同理可得出当使用 k 个哈希函数,树形布鲁姆计数器层数为 L 时,任意元素插入树形布鲁姆计数器所需要的时间分别为:经过一次哈希的时间 $HASH$;第一层和第 $L-1$ 层的使用 *Popcount* 函数的时间 $2\times POPCOUNT$;需要对每一层插入操作时需要移动其余比特位的时间 $SHIFT$;每一层相应比特位置"1"的时间 $BITSET$。同样,每一个元素需要经过 k 个哈希函数经过 K 次哈希后才能完成一次插入操作,由此可以得到总时间为 $K[HASH+2\times POPCOUNT+L(SHIFT+BITSET)]$。经过分析得出 TBF 与 BloomingTree 算法在查询、插入和删除方面的时间效率比较如表 1 所示。

表 1　TBF 与 Blooming Tree 算法时间比较

	Blooming Tree	TBF
查询	$K[HASH+L(POPCOUNT+2\times CHECK)]$	$K[HASH+2\times POPCOUNT+L\times CHECK]$
插入	$K[HASH+L(POPCOUNT+SHIFT+BITSET)]$	$K[HASH+2\times POPCOUNT+L(SHIFT+BITSET)]$
删除	$K[HASH+L(POPCOUNT+SHIFT+BITSET)]$	$K[HASH+2\times POPCOUNT+L(SHIFT+BITSET)]$

经过对上表分析可得出 TBF 算法与 BloomingTree 算法相比在时间效率方面有着一定的改进,特别是当选用的层数 L 越大时,其改进优势将会越来越明显。

4　实验验证

4.1　实验环境

为了验证本文提出的 TBF 算法,并且为了与其他的相关算法的性能进行比较,我们使用 Microsoft Visual C++ 6.0 编写 TBF 查询算法和 BloomingTree 算法的程序,这一程序可以允许使用者根据实际的需要动态地调节决定假阳性的相关参数和 TBF 层次的多少,并且在执行查询操作时能够在输出结果中分别显示查询过程中需要查询的字符的个数、正确匹配的个数以及查询所用的时间;在执行插入、删除操作时能够显示插入数据的个数和插入操作所用的时间。

在配置为内存 2G,处理器 Intel(R) core(TM)2 DUO E7200 频率为 2.73GHz 的实验机器上运行该程序,为了比较,改进 BloomingTree 的逻辑索引机构之后,TBF 与 BloomingTree 算法在查询匹配率上的优劣,选用了包含 10 000 个随机字符所组成的数据集作为基本的存储数据,在查询时则是对于包含 10 000 个固定顺序的字符在 TBF 和 BloomingTree 中进行查询从而得出结果。在比较 TBF 和 BloomingTree 算法在时间上的差异时,查询操作执行方法与和 BloomingTree 比较的方法相同,同样选用的是包含 10 000 个字符的数据集,不同之处在于在执行插入、删除操作时则是先选用了包含 10 000 个随机字符所组成的数据集作为基本的存储数据,在此基础上插入 10 000 个随机字符所组成的数据集,再在其中删除 10 000 个随机字符所组成的数据集,最后对于包含 10 000 个固定顺序的字符进行查询从而得出结果。为了得出科学的研究数据,以上的操作均采用 10 轮取平均值的方法得出最后统计结果。

4.2　实验结果

通过改变假阳性参数 $X=-f$(X 与假阳性概率 f 成反比),比较在 $K=4,5,6,7,8,9,10,11,12,14$ 时,TBF 算法与 BloomingTree 算法在匹配率方面的比较如图 4 所示。

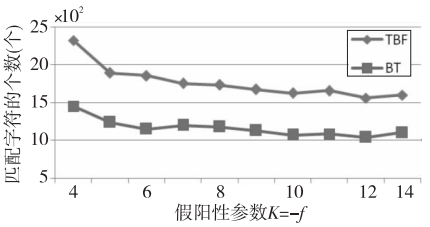

图 4　TBF 与 BloomingTree 匹配率比较

图 4 中实验结果验证的是当假阳性一致的时候,树形布鲁姆算法匹配个数要大于 BloomingTree 算法的匹配个数。假阳性相同的情况下,匹配个数多证明正确匹配的个数相比而言也就更多,实验验证了 TBF 算法在数据成员查询方面的优势。TBF 算法与 BloomingTree 算法在时间高效性方面的比较,选用最基本的层数 $L=4$,在层数 L 固定条件下,比较在 $K=2,4,6,8,10,11,12,13,14,15$ 时,分别在所构造的 TBF 和 BloomingTree

中执行查询操作后的结果如图5所示。

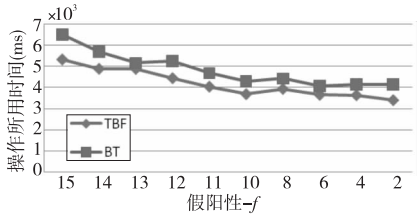

图5 TBF与BloomingTree查询时间比较

图5用于验证在层数相同条件下,假阳性发生变化时TBF算法和BloomingTree算法在查询时间上效率的优劣。经过统计实验结果后得到结论:在假阳性相同条件下,TBF算法相对BloomingTree算法而言,其查询时间平均提高了13.4%。

另外,需要比较在 $K=8$ 条件下,选用不同层数 $L=4,5,6,7,8,9,10,11,12$ 时,分别在所构造的TBF和BloomingTree中进行10轮插入和删除操作后的结果如图6所示。

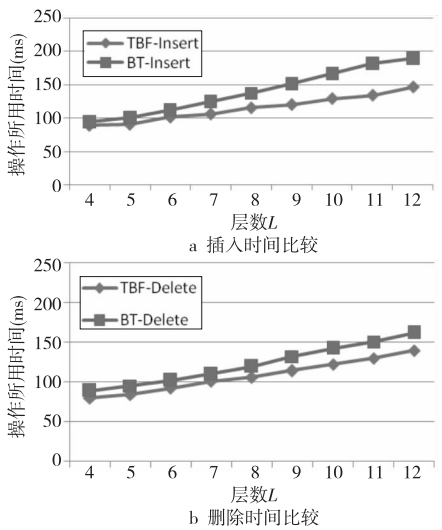

图6 TBF与BloomingTree算法10轮插入和删除的时间比较

图6用于验证在假阳性相同条件下,层数不断增大时TBF算法和BloomingTree算法在插入和删除操作时间上效率的优劣。经过统计实验结果后得到结论:在假阳性相同条件下,TBF算法相对BloomingTree算法而言,在执行插入操作时时间效率平均提高了17.9%,在执行删除操作时时间效率平均提高了12%。

5 结束语

本文提出了树形布鲁姆过滤器算法这种多层次的布鲁姆过滤器算法,它主要用于数据集成员的动态查询,能够实现计数布鲁姆过滤器同样的功能如支持数据成员信息的动态更新、有效改善数据溢出状况等等,而且在空间结构上比计数布鲁姆过滤器更加高效。它是基于BloomingTree算法的优化,改进了BloomingTree算法在逻辑索引结构方面以及在时间效率方面的缺陷。同时,通过实验验证了TBF算法与BloomingTree算法相比在时间上的高效性:在层数不变假阳性相同条件下,查询时间平均提高13.37%;在假阳性不变层数相同条件下,插入时间效率平均提高17.93%,删除时间效率平均提高12.01%。

参考文献:

[1] Bloom B H. Space/Time Trade-Offs in Hash Coding with Allowable Errors[J]. Communications of the ACM, 1970, 13(7):422-426.

[2] Fan L, Cao P, Almeida J, et al. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol[J]. IEEE/ACM Transactions on Networking, 2000,8(3):281-293.

[3] Cohen S, Matias Y. Spectral Bloom Filters[C]//Proc of the 2003 ACM SIGMOD International Conference on Management of Data, 2003:241-252.

[4] Aguilar-Saborit J, Trancoso P, Muntes-Mulero V, et al. Dynamic Count Filters[J]. SIGMOD Record, 2006,35(1):26-32.

[5] Bonomi F, Mitzenmacher M, Panigrahy R, et al. An Improved Construction for Counting Bloom Filters[C]//Proc of the 14th Annual European Symposium on Algorithms, 2006:684-695.

[6] Ficara D, Giordano S, Procissi G, et al. Blooming Trees: Space-Efficient Structures for Data Representation[C]//Proc of Global Telecommunications Conference, 2008:5828-5832.

程聂(1986-),男,湖南临澧人,硕士,研究方向为网络安全。**E-mail**:cnsk999@163.com

CHENG Nie, born in 1986, MS, his research interest includes network security.

黄昆(1978-),男,江西永丰人,博士,CCF会员(E200012780M),研究方向为网络安全和网络虚拟化。**E-mail**:huangkun09@ict.ac.cn

HUANG Kun, born in 1978, PhD, CCF member(E200012780M), his research interests include network security, and network virtualization.

苏欣(1983-),男,河南南阳人,博士,研究方向为网络安全。**E-mail**:versace0922@163.com

SU Xin, born in 1983, PhD, his research interest includes network security.