学校代号 10532 学 号 B1310S0004
分 类 号 TP391 密 级 普通

湖南大學
HUNAN UNIVERSITY

博士学位论文

基于多核系统的并发哈希表的设计与应用研究

学位申请人姓名 陈志文
培 养 单 位 信息科学与工程学院
导师姓名及职称 陈浩 教授
学 科 专 业 计算机科学与技术
研 究 方 向 高性能计算
论文提交日期 二〇一七年七月二十日

学校代号: 10532

学　　号: B1310S0004

密　　级: 普通

湖南大学博士学位论文

基于多核系统的并发哈希表的设计与应用研究

学位申请人姓名: 陈志文

培　养　单　位: 信息科学与工程学院

导师姓名及职称: 陈浩 教授

专　业　名　称: 计算机科学与技术

论文提交日期: 二〇一七年七月二十日

论文答辩日期: 二〇一七年七月二十日

答辩委员会主席: 待定

TODO

By

ZHIWEN Chen

M.S. (Hunan University)2013

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of engineering

in

Computer Science and Technology

in the

Graduate School

of

Hunan University

Supervisor

Professor HAO Chen

May, 2013

湖 南 大 学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名：_____　　签字日期：_____

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权湖南大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。本学位论文属于

□ 不保密　□ 保密（____ 年）

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

目　录

学位论文原创性声明和学位论文版权使用授权书 ····· **I**
插图索引 ····· **III**
附表索引 ····· **IV**
第 1 章　支持动态更新的并发 Cuckoo 过滤器设计 ····· **1**
1.1　布鲁姆过滤器 ····· 2
1.1.1　基本原理 ····· 2
1.1.2　误判率估计 ····· 3
1.1.3　最优哈希函数个数 ····· 4
1.1.4　最优位数组长度 ····· 5
1.2　Cuckoo 过滤器的参数 ····· 6
1.2.1　指纹信息的长度 ····· 7
1.2.2　空间效率 ····· 9
1.3　并发 Cuckoo 过滤器 ····· 10
1.3.1　加锁与解锁 ····· 11
1.3.2　插入操作 ····· 14
1.3.3　删除操作 ····· 16
1.3.4　查询操作 ····· 17
1.4　性能优化与评估 ····· 17
1.5　本章小结 ····· 17
参考文献 ····· **19**
附录 A　读学位期间所发表的学术论文 ····· **20**
附录 B　读学位期间所参加的科研项目 ····· **21**
致　谢 ····· **22**

插图索引

图 1.1　布鲁姆过滤器 ····· 2

附表索引

表 1.1 相关符号及其含义 ····· 7
表 1.2 HashTable 类的成员函数列表 ····· 13

第 1 章　支持动态更新的并发 Cuckoo 过滤器设计

查找或判断一个元素是否存在于一个指定集合中，这是计算机科学中一个基本问题。通常会采用线性表（数组或链表）、树（二叉树、堆、红黑树、B+/B-/B*树）等数据结构存储所有元素，对数据进行排序和查找。这里的查找时间复杂性通常都是 O(N) 或 O(log(N))。如果集合元素非常庞大，不仅降低了查找的效率，同时对内存空间的需求也非常大。

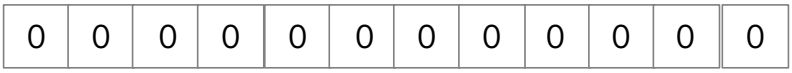
在网络安全领域有一个简单的应用场景：判断 URL 是否链接到存在安全隐患的网站。用户在浏览器内输入 URL，浏览器需要判断该 URL 是否是恶意的，它将该 URL 与本地缓存的 URL 进行匹配，如果匹配失败，则说明该 URL 是安全的链接可以正常访问；否则，说明该 URL 可能存在安全隐患。此时，提交请求给远程客户端进行验证，并警告用户该 URL 存在风险。在这个应用场景中，如果缓存的 URL 数量很少，那么使用上述的数据结构都可以达到较高的查找效率，同时对内存空间的要求也不高。假设现在需要缓存的 URL 的数量为 10 亿条 (这在当前是很常见的一个数量级)，每条 URL 的大小为 8 个字节，那么存储所有的 URL 大约需要 8 GB 的内存。使用哈希表是一种可能的解决方案. 哈希表的查询时间复杂度为 O(1)，可以节省查找的时间，但是没有降低对内存的需求。

事实上，除非有特别的需求，否者判断元素是否在一个指定集合内，并不需要把所有元素的原始信息都保存下来，而只需要保存该元素的“存在状态”，存储存在状态只需要几个 bit。使用哈希函数可以将元素映射成位数组中的一个点，采用 k 个哈希函数将元素映射成 k 个点。这样，经过映射之后，查找元素是否存在时只需看看特定的几个位点的值就能判断某个元素是否存在于集合当中，如果 k 个位置都为 1，则说明该元素可能存在，如果有 1 个位置上为 0，则可以肯定该元素不存在。这样不仅可大大缩减内存空间，查找速度非常这就是布鲁姆过滤器 (Bloom Filter) 的基本思想。它的名字源自其发明者 Burton.H Bloom[1]。布鲁姆过滤器最初应用于拼写检查和数据库系统。但是，随着互联网的爆炸式发展，海量数据中快速检索目标数据的需求使得布鲁姆过滤器的应用焕发新生，涌现出新的应用和变种[2–5]。

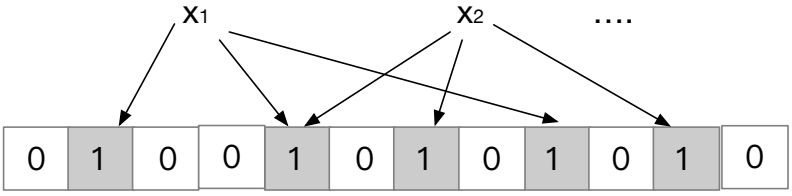
1.1 布鲁姆过滤器

1.1.1 基本原理

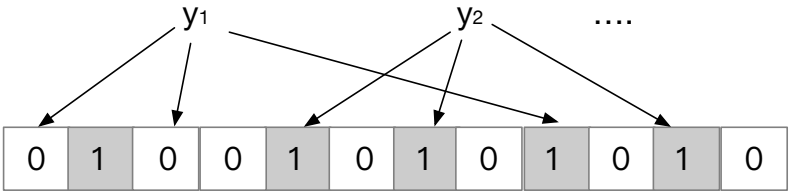
布鲁姆过滤器使用位数组表示元素集合 S，并使用 k 个哈希函数 ($h_1, h_2, ..., h_k$) 来对元素进行位映射。初始状态下的布鲁姆过滤器是一个包含 m 位的位数组，每一位都置 0，图 1.1(a) 所示为 m = 12 的布鲁姆过滤器。当需要将集合 S 中的 n 个元素 x_1，x_2, ... x_n 用位数组表示时，对该元素分别使用 k 个相互独立哈希函数进行计算，得到位数组上 k 个位置的索引值，随后将映射到位数组的相应位置 1。值得注意的是，如果一个位置被多次置为 1，只有第一次的置位是有效的。图 1.1(b) 表示 k = 3 时，将元素映射到位数组的过程，其中元素 x_1 和 x_2 都对第 5 位置位。图 1.1(c) 表示判断元素 y_i(i = 1,2, ..., n) 是否属于集合。与插入过程类似，同样先对 y 进行 k 次哈希，如果计算得到的索引值对应的位上有任何一位为 0 则表示 y 元素绝对不存在于集合中，只有当所有映射位均为 1 时才表示该元素**有可能**存在于集合当中。换句话说，如果布鲁姆过滤器判断一个元素不在集合中，那



(a) 初始化的m=12的布鲁姆过滤器



(b) 存储集合内的元素



(c) 查询集合内的元素

图 1.1 布鲁姆过滤器

肯定就不存在；而如果判断存在，则不一定存在，下文将对这种不一定存在的原因进行说明。这种不能确定元素一定存在的问题是由哈希函数可能发生碰撞的特性所决定的。这个错误率可调整位数组大小或者哈希函数个数进行控制。由此可见，布鲁姆过滤器的高效是以一定的误报为代价的，它通过容忍一定的错误发生

的概率换取存储空间的极大节省。布鲁姆过滤器不适合那些“零错误”的应用场合。

标准的布鲁姆过滤器不支持删除操作。原因很简单，考虑在图 1.1(b) 中删除元素 x_2，意味着将位数组内的第 5、7、11 位置 0，此时如果查询 x_1 会发现也不存在，因为它对应的第 5 位上的值在删除 x_2 时被置 0 了。

标准布鲁姆过滤器的实现中有几个重要参数：误判率ϵ、负载因子α、哈希函数的个数 k、位数组大小 m 和集合中元素的个数 n。下面将对这些参数进行推导，以确定实现标准布鲁姆过滤器的参数选取原则。

1.1.2 误判率估计

在进行正式的误判率估计前先明确几个定义和重要的参数：

定义 1.1 假阳性 (false positives) 也叫误判是指当前元素不在集合内，但由于哈希冲突的缘故存在其它元素被映射到部分相同 bit 位上，从而有一定的概率导致在判定该元素时认定其对应的所有位置都为 1，从而判定其在集合内，造成一次误判。这个概率本文称为误判率，误判率用ϵ 表示。

定义 1.2 假阴性 (false negatives)，也叫漏报，是指在位数组内删除某个元素，导致该元素对应的比特位置 0，造成本来存在的元素会被漏报成不存在。

定义 1.3 负载因子是指集合中的元素的个数 n，布鲁姆过滤器位数组的位数 m 之间的比值，它用α 表示，其中$\alpha = n/m$。当α 为 0 时，表示布鲁姆过滤器为空，α 为 1 表示布鲁姆过滤器满载。

下面进行误判率ϵ 的推算. 首先假设布鲁姆过滤器中使用的 k 个哈希函数的计算结果都是均匀分布的，即每个元素都等概率地被哈希到 m 个 bit 位上的任何一个，与其他元素被哈希到的位置无关。则对某一特定 bit 位在一个元素由特定哈希函数插入时没有被置位为 1 的概率 p_1 为：

$$p_1 = 1 - 1/m \tag{1.1}$$

则 k 个哈希函数中都没有一个对其置位的概率 p_2：

$$p_2 = {p_1}^k \tag{1.2}$$

如果插入 n($n \leqslant m$) 个元素，但都未对其置位的概率 p_3:

$$p_3 = {p_2}^n = {p_1}^{kn} \tag{1.3}$$

则此位被置位的概率 p_4 为:

$$p_4 = 1 - (1 - 1/m)^{kn} \tag{1.4}$$

现在考虑判定阶段，若对应某个判定元素的 k 个位全部置位为 1，则可判定其在集合中。因此将某元素误判的概率为:

$$\epsilon = \left(1 - (1 - 1/m)^{kn}\right)^k \tag{1.5}$$

由 $(1+x)^{1/x} \approx e$ 可知，但 m 很大时，满足 $-1/m \to 0$，可将公式1.4转化为:

$$\epsilon = \left(1 - (1 - 1/m)^{-m\frac{-kn}{m}}\right)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k \tag{1.6}$$

由公式1.6可以初步断定ϵ 与元素的个数 n 和位数组的长度 m 决定，增大 n 或者减小 m 都会导致ϵ 的升高。这种计算方法不严格，因为前面假设哈希函数和散列后值的分布是相互独立的。但是，这个假设随着 m 和 n 的增大误判率更接近真实的误判率。Mitzenmacher 证明无假设情况下的误判率的期望值相同[6]。

1.1.3 最优哈希函数个数

哈希函数的选择对于布鲁姆过滤器的性能以及空间利用率都有至关重要的作用。对于选取什么样的哈希函数已经在前文中有过介绍，这里不再缀述。而对于哈希函数的个数，直观的认为越多越好。实际上，哈希函数越多，用于表达集合中每一个元素所需要的位数就越多，这与布鲁姆过滤器用较低的误判率换取空间的高效利用的初衷相悖。那么哈希函数的个数应该满足什么条件才能有最佳性能呢?

下面推导对给定的 α，k 满足什么条件可以使 ϵ 最小化。

令:

$$f(k) = \left(1 - e^{-nk/m}\right)^k \tag{1.7}$$

取 $b = e^{n/m}$，得:

$$f(k) = \left(1 - b^{-k}\right)^k \tag{1.8}$$

两边取对数得:

$$lnf(k) = kln\left(1 - b^{-k}\right) \tag{1.9}$$

函数两边对 k 求导得:

$$1/f(k)\cdot f'(k) = ln(1-b^{-k}) + k\frac{b^{-k}\cdot lnb}{1-b^{-k}} \tag{1.10}$$

对式 1.10右边求最值。
令:

$$\begin{aligned}\frac{1}{f(k)}\cdot f'(k) = ln(1-b^{-k}) + k\cdot\frac{b^{-k}\cdot lnb}{1-b^{-k}} = 0\\ \Rightarrow (1-b^{-k})ln(1-b^{-k}) = -k\cdot b^{-k}\cdot lnb\\ \Rightarrow 1-b^{-k} = b^{-k}\\ \Rightarrow e^{-\frac{kn}{m}} = \frac{1}{2}\\ \Rightarrow k = ln2\cdot m/n \approx 0.7m/n\end{aligned} \tag{1.11}$$

因此，对于固定的 α，当 $k = 0.7m/n$ 时具有最低的误判率，此时 ϵ 等于:

$$(1-1/2)^k = 2^{-ln2\cdot\frac{m}{n}} \approx 0.62m/n \tag{1.12}$$

1.1.4 最优位数组长度

下面进行给定一定的误判率上限，布鲁姆过滤器至少需要多少位才能表示全集中任意的 x 个元素的集合。假设全集中元素的个数为 n，最大误判率为 *epsilon*。以此为前提展开对位数组大小的推导。

假设 S_n 为全集中任取 n 个元素的集合，B 是表示 S_n 的位数组。那么对于集合 S_n 中任意一个元素 x，在 B 中查询 x 都能得到肯定的结果，即 B 能够接受 x。显然，由于布鲁姆过滤器引入允许误判，B 能够接受的不仅仅是 S_n 中的元素，它还允许最多 $\epsilon\cdot(u-n)$ 个误报。因此，对于一个确定的位数组来说，它能够接受总共 $n+\epsilon\cdot(u-n)$ 个元素。在 $n+\epsilon\cdot(u-n)$ 个元素中，B 真正表示的只有其中 n 个，所以一个确定的位数组可以表示:

$$\binom{n+\epsilon\cdot(u-n)}{n} \tag{1.13}$$

个集合。m 位的位数组一共有 2^m 个不同的组合，可以进一步推导 m 位的位数组可以表示:

$$2^m\binom{n+\epsilon*(u-n)}{n} \tag{1.14}$$

个集合。全集中包含 n 个元素的子集总共有:

$$\binom{u}{n} \tag{1.15}$$

个。因此，要让布鲁姆过滤器的位数组大小能够满足所有包含 n 个元素的子集，必须满足:

$$2^m \cdot \binom{n+\epsilon\cdot(u-n)}{n} \geqslant \binom{u}{n} \tag{1.16}$$

即 m 需要满足:

$$m \geqslant log_2(\binom{u}{n}/\binom{n+\epsilon\cdot(u-n)}{n}) \geqslant log_2(\binom{u}{n}/\binom{\epsilon u}{n}) \approx log_2\epsilon^{-n} = nlog_2(1/\epsilon) \tag{1.17}$$

式 1.17中近似相等有个重要的前提条件：n 远小于 $\epsilon\cdot u$，这个前提在实际的问题中也是常见的。根据式 1.17中的不等式，得到如下结论：在误判率上限为 ϵ 的情况下，m 至少要等于 $nlog_2 1/\epsilon$ 才能表示任意 n 个元素的集合。

在本章 1.1.3中推导出哈希函数的个数 k 等于 $k = 0.7m/n$ 时可以得到最小误判率，此时的误判率为 $(\frac{1}{2})^{0.7m/n}$。令 $(\frac{1}{2})^{0.7m/n} \leqslant \epsilon$，可以进一步推导:

$$m \geqslant n\frac{log_2(1/\epsilon)}{ln2} = nlog_2e * log_2(1/\epsilon) \approx 1.44n\cdot log_2(1/\epsilon) \tag{1.18}$$

式 1.18说明当 k 取到最优值时，要保证误判率不超过给定的上限 ϵ，m 至少要取到最小值的 1.44 倍。可以验证，当给定 $\epsilon = 0.01$ 时，存储每个元素需要 9.6 比特。而将 $\epsilon = 0.001$ 时，每个元素需要额外的增加 4.8 比特。所以，在实际的应用中，对于误判的容忍度不同，要求误判率越低，则存储每个元素需要的比特位越多，相同容量下存储的元素个数就越少。

1.2 Cuckoo 过滤器的参数

传统的布鲁姆过滤器的空间效率高，对插入和查询元素的处理也相当快。但是它也存在缺陷——存在一定的误报率，不支持元素的删除操作。消除误报率除非能实现没有碰撞的哈希函数，但是不发生碰撞的哈希函数至今没有被设计出来。研究人员能做的就是尽量选择均匀的哈希函数，并且借助一些数据结构的特性有效的对碰撞进行处理。而在上一节的中介绍到误判率每缩小到原来的十分之一，至少要增加 4.8 个比特位用于表示一个元素。另外，实现元素删除操作的一个方法是引入计数器，将每个比特位都扩张成一个计数值，降低了空间效率。为了支持对元素的删除操作，出现了很多标准布鲁姆过滤器的扩展版本[2,3,7]。所以，

布鲁姆过滤器实现更低的误判率和实现删除操作都需要牺牲一定的空间效率。

为了解决上述问题，本文引入一种新的数据结构——Cuckoo 过滤器。它既可以确保该元素存在的必然性，即将“可能存在”变成“一定存在”；又支持动态的对元素的插入和删除，而不会造成漏报。

为了支持动态的插入和删除元素，Cuckoo 过滤器采用的是一种称为**不完整键值 (partial-key)** Cuckoo 哈希的技术。Cuckoo 过滤器是标准 Cuckoo 哈希表在集合元素查询算法领域的应用。在前面的章节中已经对 Cuckoo 哈希表的基本原理和概念有过详细介绍 (第 **??**节)，这里不再赘述。仅对与在这一部分密切相关的术语进行重申。Cuckoo 的数据结构与 Cuckoo 哈希表相同，其基本单元称为**实体** (entry)，不同的是每一个实体内存储的不是完整的键值，而是根据键值进行提取后的**指纹** (fingerprint)[8]，用 f 表示。哈希表由存储了多个实体的**桶** (bucket) 数组构成。

表 1.1列出了本小节所用到的一些关键参数及其含义。

表 1.1　相关符号及其含义

参数	含义
ϵ	误判率
f	指纹信息的长度 (单位：bit)
α	负载因子 ($0 \leqslant \alpha \leqslant 1$)
b	每个哈希桶内包含的实体的数量
m	哈希桶的数量
n	条目 (元素) 的数量
C	表达一个条目所需的平均位数 (单位：bit)

1.2.1　指纹信息的长度

在这一部分中，将探讨构造 Cuckoo 过滤器的几个关键参数。在 Cuckoo 过滤器中使用不完整键 Cuckoo 哈希方法存储指纹信息会导致指纹信息的下界值随着过滤器大小的增加而缓慢增加。这与用传统方法实现的过滤器相反，在标准布鲁姆过滤器中，指纹信息的大小只与预定的 ϵ 有关。这个看上去似乎是 Cuckoo 过滤器的劣势，实际上它造成的影响微乎其微。

(1)　指纹信息下界约束

在 Cuckoo 过滤器中，对于一个给定的元素，根据其当前位置和指纹信息使用不完整键哈希方法可以推导出它的备选哈希桶的索引值。这样，每个元素的候选哈希桶都不是独立的。比如，某一元素可以存放在桶 i_1 或 i_2 中，对于长度为 f

比特的指纹信息，根据公式 1.24，i_2 可能的索引值最多有 2^f 种可能。若指纹信息的长度为一个字节，对给定的 i_1，i_2 最多只能偏离 i_1 256 个位置。对于具有 m 个桶的哈希表而言，当 $2^f \leqslant m$ 时，i_2 能够选择的哈希桶的范围只是整个 m 个哈希桶的一个很小的子集。这会引起更多的碰撞。

直观上看如果指纹信息的长度足够长，不完整 Cuckoo 哈希仍然能够接近标准 Cuckoo 哈希的冲突处理能力。然而，如果哈希表非常大，而此时指纹信息的长度相对来说要远远小于哈希表的大小，这样容易引起更多的哈希碰撞，从而导致插入失败的概率升高。当 Cuckoo 过滤器需要处理大量元素，而 ϵ 设定一个 s 中等偏低的值时，可能发生上述情形。接下来，将通过分析确定插入失败的概率下限。

首先推导对于给定的 u 个元素，它们恰好映射到相同的两个哈希桶内的概率 p_1。假设第一个元素 x 位于桶 i_1 内，并且指纹为 t_x。如果其他的 $u-1$ 个元素具有与 x 相同的桶索引值，它们必然满足以下两个条件：(1) 它们的指纹都为 t_x，出现的概率为 $\frac{1}{2^f}$；(2) 它们第一个桶索引值为 i_1 或者 $i_1 \bigoplus h(t_x)$，出现的概率为 $\frac{2}{m}$。因此，u 个元素映射到相同的两个桶内的概率 $p_1 = (\frac{2}{m} * \frac{1}{2^f})^{u-1}$

现在考虑构建 Cuckoo 过滤器的随机插入 n 个元素的构建过程。假设初始化的哈希表桶数组满足 $m = cn$，每个哈希桶容纳的元素个数为 b，其中 c 为常数。当出现 $u = 2b+1$ 个元素被映射到相同的两个桶内时，插入失败。这个概率为插入失败的概率下界。由于从 n 个元素中包含 $2b+1$ 个元素的子集有 $\binom{n}{2b+1}$ 个，$2b+1$ 个元素在插入过程中发生碰撞的期望值为：

$$\binom{n}{2b+1}(\frac{2}{2^f m})^{2b} = \binom{n}{2b+1}(\frac{2}{2^f cn})^{2b} = \Omega(\frac{n}{4^{bf}}) \tag{1.19}$$

因此，由式 1.19可以做出结论，4^{bf} 必须满足 $\Omega(n)$ 才能避免一场的插入失败的概率。指纹信息的长度最好为 $f = \Omega(log(\frac{n}{b}))$ 比特。在第 1.1.4节中指出标准的布鲁姆过滤器用于表示每个元素所需的比特数为常数 (近似等于 $ln(\frac{1}{\epsilon})$)。而 Cuckoo 过滤器指纹信息所需的长度为 $\Omega(logn)$ 这个级别，这个结果看上去似乎不是特别理想。这会不会引起扩展性问题呢？实验表明桶容量 b 的在下界约束中的起决定作用：只要将 b 控制在合理的大小，指纹信息的长度仍然可以保持较小的值。

(2) 实验评估

todo

(3) 启示

todo

1.2.2 空间效率

在第 **??**节描述的 Cuckoo 过滤器的三种操作与每个哈希桶内包含多少实体无关。但是，为 Cuckoo 过滤器选择正确的参数对于空间效率具有重要意义。这一部分着重介绍如何选取合适的参数优化 Cuckoo 过滤器的空间效率。

空间效率是通过计算在完整的过滤器中用于表示每个元素所用的平均比特数来衡量的。用哈希表的大小除以过滤器有效存储的元素的个数就是表示每个元素所用的平均比特数。尽管每个实体可以存储一条指纹信息，但是并不是所有的实体都已经存入了指纹信息——过滤器的哈希表内一定有空闲的实体。所以，每个元素实际上需要的比特数大于指纹信息的长度。如果每条指纹信息的长度为 f 比特，哈希表的负载因子为 α，则每个元素的空间开销 C 为:

$$C = \frac{\text{哈希表的大小}}{\text{元素个数}} = \frac{f \cdot \text{实体数量}}{\alpha \cdot \text{实体的数量}} = \frac{f}{\alpha}\text{bits.} \tag{1.20}$$

在前文中有过介绍，指纹信息的长度和负载因子都与哈希桶的大小有关。下面研究在给定的误判率 ϵ 前提下，如何通过选取最优的桶大小 b 使 C 最小化。

保持 Cuckoo 过滤器的总的大小为常量，改变哈希桶的大小会产生两方面的影响:

- **哈希桶容纳的实体数量越多，哈希表的空间占用率就越高。**对于使用两个哈希函数的 Cuckoo 过滤器，当桶能容纳的实体数 $b = 1$ 时，哈希表的负载因子 α 为 50%，而当 $b = 2, 4, 8$ 时，α 分别为 84%，95% 和 98%。
- **哈希桶的容量越大，维持相同 ϵ 需要的指纹信息的长度越长。**哈希桶的容量越大，进行查询时需要检查更多的实体，并且发现相同指纹信息的概率也会增加。在最坏情况下，查询一个并不存在的元素需要探测两个分别包含了 b 个实体的桶 (当然并不是所有的哈希桶内都填满了实体，这里只是考虑最糟糕的情况；当哈希表的负载因子达到 95% 时，已经很接近极限情况)。对于每一个实体而言，一次查询与存储的指纹相匹配并且返回成功匹配误报的概率最多为 $1/2^f$。在进行 $2b$ 次这样的比较之后，误报率的上界为:

  $$1 - (1 - 1/2^f)^{2b} \approx 2b/2^f \tag{1.21}$$

  该上界约束与哈希桶的容量 b 成正比。为了保证预定的误报率 ϵ 不变，必须确保 $2b/2^f \leqslant \epsilon$，保证这个条件的最小指纹信息长度为:

  $$f \geqslant \lceil log_2(2b/\epsilon) \rceil = \lceil log_2(1/\epsilon) + log_2(2b) \rceil \tag{1.22}$$

由式 1.20和1.22可以推算存储每个元素的空间开销 C 受下面条件的限制：

$$C \geqslant \lceil log_2(1/\epsilon) + log_2(2b) \rceil / \alpha \tag{1.23}$$

α 随着 b 的增加而增加。当 $b = 4$ 时，$\alpha = 0.95$，$1/\alpha \approx 1.05$。此时 $C = 1.05log_2(1/\epsilon) + 1.05 \cdot 3$。式 1.23表明，当负载因子一定时，Cuckoo 过滤器的空间开销要低于布鲁姆过滤器 $(1.44log_2(1/\epsilon))$。

为了确定最优的哈希桶容量 b，下面将通过实验比较参数 b 为不同的值时的空间效率。用不完整键 Cuckoo 哈希方法构造具有不同的指纹信息长度的哈希表，分别记录对应的平均空间开销和误判率。结果如图 **??**所示。使空间效率最好的 b 的取值依赖于预定的误判率 ϵ：当 $\epsilon > 0.002$ 时，$b = 2$ 对应的平均空间开销要略好于 $b = 4$ 对应的空间开销；而当 $10^{-5} < \epsilon \leqslant 0.002$ 时，$b = 4$ 具有最小的空间开销。

综上所述，Cuckoo 过滤器的默认参数配置为 (2，4)，即每个元素有两个候选的哈希桶，每个哈希桶最多能够容纳 4 条指纹信息。选取这组参数作为默认配置的原因是实际的应用一般都要求 $10^{-5} < \epsilon \leqslant 0.002$[9]。

(1) 性能评估

1.3 并发 Cuckoo 过滤器

无论是标准的布鲁姆过滤器还是现有的其他一些布鲁姆过滤器的扩展版本，都具有在单核平台上快速处理元素的能力以及高效的空间利用率。单核处理器的计算能力已达到瓶颈，相对而言多核计算机的计算资源和计算能力更加充裕。在当前数据呈现爆炸式增长的背景下，海量数据处理压力越来越大，单核处理器上的过滤器逐渐显得捉襟见肘。因此，设计基于多核系统的多线程并发的过滤器对于海量数据处理无疑是雪中送炭。然而，当前的相关研究中并没有一款支持多核并发查询和更新的过滤器。

并发控制在多核平台上设计并发数据结构的关键环节，它对线程扩展性和整体性能起决定作用。在前面对并发哈希表的评估与分析中通过比较各个并发哈希表的线程同步方式发现不当使用共享变量以及使用 TAS 锁不利于并发哈希表的线程扩展性。本文设计的并发 Cuckoo 过滤器其基础数据结构本质上仍然是哈希表，不同的是 Cuckoo 过滤器结合了处理集合成员查询问题的特征改用存储不完整键值替换标准 Cuckoo 哈希表中存储完整的键值信息的做法，换取存储空间的极大节省。因此，前文通过实验评估得出的结论仍然适用于并发 Cuckoo 过滤器。

在这个背景下，设计了一种支持多核并发的基于不完整键 Cuckoo 哈希方法的过滤器。这一部分主要介绍并发 Cuckoo 过滤器的实现过程和性能评估结果。

1.3.1 加锁与解锁

接下来的内容介绍如何通过基于Intel RTM的MCS锁实现多线程并发的Cuckoo过滤器。Listing 2.1给出了基于Intel RTM的MCS锁算法实现。*locklib_mutex_t*是一个包含了MCS锁字段、一个*uint8_t*类型的*mode*变量以及一个用于缓存行对齐的字符串的结构体。函数*locklib_mutex_lock*()具有两个参数，一个为互斥量*mutex*，一个为整型数*mode*。*mode*的值分别对应0、1，0表示当前持有锁的操作为读取操作，1表示当前持有锁的操作为更新操作(删除或插入)。

Listing 1.1 基于Intel RTM的MCS锁算法

```
1  struct {
2        mcs_lock_t lock;
3        uint32_t pad1[128/4 - sizeof(mcs_lock_t)];
4  } spec_mcs_lock_t;
5
6  struct {
7        mcs_lock_t    mcs;
8        uint8_t mode;
9        char padding[];
10 } locklib_mutex_t;
11
12 int locklib_mutex_lock(locklib_mutex_t *mutex, uint8_t mode){
13       spec_mcs_lock_t *lock = (spec_mcs_lock_t *) mutex;
14       uint32_t reason = 0;
15
16 speculative_path:
17       XBEGIN(fallback_path, reason);
18       if (lock->lock) XABORT(1);
19       return 0;
20
21 fallback_path:
22       retries++;
23       while (lock->lock)
24             cpu_relax();
25       if (retries < MAX_RETRIES)
26             goto speculative_path;
27
28       // Acquire lock in a standard manner
29       my_node.locked = true;
30       qnode_t *prev = __sync_lock_TAS(&lock->lock, &my_node);
```

```
31        if (unlikely(prev != NULL)) {
32              prev->next = &my_node;
33              while (my_node.locked)
34                    cpu_relax();
35        }
36        mutex->mode = mode;
37        return 0;}
```

对临界区的保护设置了 *speculative_path*，*fallback_path* 两条执行路径：一条为事务化推测执行路径 (第 16 行)；一条为回退路径 (第 21 行)，即当事务执行失败后，临界区申请标准锁完成本次操作。在事务化推测执行期间，线程首先不会申请获取锁，直到执行完成准备提交时再申请获取锁，如果申请的锁被占用，则该线程所执行的事务被中止，线程对系统状态所做的更改都失效，系统回退到初始状态跳转到回退路径执行 (事务内存的原子性)。

跳转到回退路径之后，如果当前重试的次数没有达到预先设定的门限值，将继续尝试进行事务化推测执行 (第 23-26 行)。如果当前重试次数已经达到了门限值，则使用标准的锁方法完成操作 (第 29-35 行)。

由于硬件事务内存不能保证每次事务化执行都能成功提交对系统状态的更改，为了避免进程悬停，在使用硬件事务内存进行锁省略编程时设置回退路径是有效的保障程序顺利执行的手段。一般的，事务代码在经过一定次数的重试之后成功提交的概率远远大于失败的概率，所以执行回退路径对性能的影响是可控的。

Listing 1.2　基于 Intel RTM 的 MCS 解锁算法

```
1  int locklib_mutex_unlock(locklib_mutex_t *mutex)
2  {
3     spec_mcs_lock_t *lock = (spec_mcs_lock_t *) mutex;
4
5     if (XTEST()) {
6        XEND();
7     }
8     else {
9        // Release lock in a standard manner
10       qnode_t *last = my_node.next;
11       if (last == NULL) {
12          if (likely(true == __sync_bool_CAS
13             (&lock->lock, &my_node, NULL)))
14                return 0;
15
16          while ((last = my_node.next) == NULL)
```

```
17                      cpu_relax();
18              }
19
20          my_node.next = NULL;
21          last->locked = false;
22      }
23
24      retries = 0;
25      return 0;
26  }
```

Listing 2.2 展示了对应的解锁过程。与加锁过程的两条路径相对应，释放锁的过程也分为两个阶段：首先使用 *XTEST* 判断当前执行的操作是否为事务执行，若为事务执行，使用 *XEND* 结束；若判断此次操作申请的锁是通过标准的方式获取的，则按照标准锁的释放过程释放锁。最后将 *retries* 变量清零。

(1) 并发访问接口

表 1.2 HashTable 类的成员函数列表

序号	API	描述
1	explicit **HashTable(num)**	构造函数
2	~ **HashTable()**	析构函数
3	size_t **NumBuckets()**	返回哈希桶数量
4	size_t **SizeInBytes()**	返回哈希表大小 (单位:Bytes)
5	size_t **SizeInTags()**	返回哈希表容纳的指纹的数量
6	string **Info()**	返回哈希表的容量信息
7	uint32_t **ReadTags(i, j)**	读取指纹信息
8	void **WriteTags(i, j, t)**	修改指纹信息
9	bool **ConFindTagInBuckets(i1, i2, tag)**	并发查找接口
10	bool **ConDeleteTagFromBucket1(i, tag)**	并发删除接口
11	ReturnCode **ConInsertTagToBucket(i, tag, kickout, &oldtag)**	并发插入接口

并发 Cuckoo 过滤器的实现使用了大约 500 行 C++ 代码。在这一部分中，将对并发 Cuckoo 过滤器 **HashTable** 类的主要 API 进行介绍。表 1.2 列出了 Cuckoo 过滤器的 *HashTable* 类的主要 API。序号 3-5 对应的 API 主要用于统计哈希表的信息，用于最终计算哈希表的负载因子，内存消耗等；序号 6 对应的 API 输出指纹信息的长度、每个哈希桶能容纳的指纹信息的数量、哈希桶的数量、哈希表的最大指纹容量等信息；序号 7、8 对应的 API 分别用于读取和修改指定的指纹信息；序号 9-11 对应的 API 实现在哈希表内并发的读取和修改元素。

下面对 Cuckoo 过滤器的三种元素操作的多线程并发实现进行描述。

1.3.2 插入操作

在标准 Cuckoo 哈希表中，在哈希表中插入新的条目时需要以某种方式读取原本存储于哈希表内的条目，以便在发生踢出原始条目时确定将该条目安置在哪个位置上。但是，对于只存储了指纹的 Cuckoo 过滤器而言，它无法根据原始条目的键计算出旧键迁移到哪个位置。这里引入不完整键 Cuckoo 哈希算法来解决无法根据指纹信息定位旧键迁移位置的问题。对任意的条目 x，使用公式 1.24计算其两个备选哈希桶的索引值:

$$\begin{aligned} h_1(x) &= hash(x) \\ h_2(x) &= h_1(x) \bigoplus hash(\text{x 的指纹信息}) \end{aligned} \tag{1.24}$$

式 1.24的异或操作有一个重要的特性：$h_1(x)$ 可以通过 $h_2(x)$ 和指纹信息用同样的公式推算出来。即就是说，替换编号为 i 的哈希桶中的旧键 (不论 i 对应的是 h_1 还是 h_2)，都可以通过当前桶编号 i 以及存储在该桶内的指纹直接计算出旧键的备选哈希桶的编号 j，计算方式为：$j = i \bigoplus hash$(指纹信息)。

因此，在进行插入操作时，不用检索目标哈希桶内存储的条目的完整信息，只需要存储在哈希表中的指纹信息。

另外，为了使众元素在哈希表中均匀分布，在与索引值进行异或运算之前，指纹信息已经进行过哈希运算。当指纹信息远小于哈希表的大小时，如果直接使用索引值与未经哈希的指纹信息进行异或运算推算接纳被踢出元素的哈希桶的位置，那么接纳被踢出的元素哈希桶与原来的哈希桶在位置上很近。下面举例说明。使用 8 比特的指纹信息时，接纳从 i 中被踢出的条目的哈希桶的位置距离 i 的最大距离为 $2^8 = 256$。这时因为在进行异或运算时，会选取索引值的低 8 位进行运算，而高 8 位保持不变。对指纹信息进行哈希能够确保这些条目尽量分散在哈希表的不同位置，从而可以避免哈希碰撞，提高哈希表的空间利用率。

算法 1.1给出了使用不完整键 Cuckoo 哈希方法对 Cuckoo 过滤器动态插入元素的过程。

指纹信息的长度小于 h_1 和 h_2 的长度造成的后果有两个方面：**第一**，通过公式 1.24计算出的 (h_1, h_2) 的组合的总数会远远小于使用完整的哈希值计算得到的组合的数量，这将导致哈希碰撞更严重；**第二**，允许插入两个具有相同指纹的条目 x 和 y，在一个哈希桶内可能出现多个相同的指纹信息是合法的。但是如果相同的指纹信息的数量超过 2b(b 为哈希桶的大小) 时，存储其指纹信息的哈希桶会过载。解决哈希桶过载的途径有多种。

- 第一，也是最简单的方法——不实现过滤器删除元素操作，这样每条指纹信息都只需要存储一份副本。但这显然与 Cuckoo 过滤器的设计初衷不符。
- 第二，引入适当的空间开销在哈希桶内加入计数器，进行插入/删除操作时

算法 1.1: Cuckoo 过滤器插入操作

```
1  # define UPDATE_LOCK     locklib_mutex_lock(mutex, 1)
2  # define UPDATE_UNLOCK     locklib_mutex_unlock(mutex)
3  Function Insert(x)
4  f = fingerprint(x)
5  i1 = hash(x)
6  i2 = i1 ⊕ hash(f)
7  UPDATE_LOCK
8  if 桶 i1 或 i2 内有空闲实体; then
9      将 f 存入 i1 或 i2;
10     UPDATE_UNLOCK
11     return true
12 end
   /* 当前桶内没有空闲位置                                          */
13 i = rand(i1, i2)
14 for n = 0 to MaxNumKicks − 1 do
15     从 bucket[i] 中随机的选择一个实体 e;
16     将新插入条目的指纹信息与 e 的指纹信息交换;
17     i = i ⊕ hash(f)
18     if bucket[i] 有空闲实体; then
19         将 f 存储到 bucket[i];
20         UPDATE_UNLOCK
21         return true
22     end
23 end
24 UPDATE_UNLOCK
   /* 哈希表饱和                                                    */
25 return false
```

适当的自增/自减。

- 第三，将原始的键存储在其他位置 (可以是访存速度较慢的外存上)，这样可以在插入时查看该记录，防止重复插入。但是如果哈希桶内已经存在匹配的实体，则插入的速度相对较慢。

1.3.3 删除操作

在标准的布鲁姆过滤器中删除条目需要对整个过滤器进行重建，引起惊人的性能开销。所以标准布鲁姆过滤器不支持删除操作。而将布鲁姆过滤器的比特位扩展成计数值的方法需要耗费 3-4 倍的空间开销。Cuckoo 过滤器可以直接从过滤器中移除相关条目的指纹完成删除操作。有关 Cuckoo 过滤器的删除过程如算法 1.2所示。

算法 1.2: Cuckoo 过滤器的删除操作

```
1  # define UPDATE_LOCK     locklib_mutex_lock(mutex, 1)
2  # define UPDATE_UNLOCK     locklib_mutex_unlock(mutex)
3  Function Delete(x)
4  f = fingerprint(x)
5  i1 = hash(x);
6  i2 = i1 ⊕ hash(f)
7  UPDATE_LOCK
8  if bucket[i1] 或 bucket[i2] 中含有 f then
9      从当前 bucket 内删除 f 的一个副本;
10     UPDATE_UNLOCK
11     return true
12 end
13 UPDATE_UNLOCK
14 return false
```

相比当前一些支持删除操作的布鲁姆过滤器的扩展版本，比如 *d-left* 计数过滤器，熵过滤器的实现，Cuckoo 过滤器的删除操作十分简单。对于一个需要删除的条目，Cuckoo 过滤器首先根据索引值在桶内进行查找；如果在任意的桶内有匹配的指纹信息，则删除该桶内的指纹信息的一个副本。

在删除某个条目后，不需要对这个实体进行清理。这样可以避免在同一个桶内存有两个具有相同指纹信息的条目时的“误删”。假设条目 x 和 y 都映射到了桶 i_1 内，并且具有相同的指纹信息 f。因为 $i_2 = i_1 \bigoplus hash(f)$，所以它们同样能够保存在桶 i_2 内. 在删除 x 时，不用考虑删除的指纹信息的副本是在插入 x 还是 y 是添

加的。删除 x 后，在桶 i_1、i_2 中 y 仍然可以被查询到。

值得注意的是，在上面的例子中删除条目 x 之后，过滤器的误判行为仍然存在。过滤器中的 y 会在查询 x 时误报，因为两者具有相同的桶索引值和指纹信息。误判行为仍然在近似集合元素查询数据结构接受的范围之内，误判率也仍然满足 ϵ 的上界约束条件。

安全的删除条目有一个前提条件：被删除的 x 必须是已经插入到了过滤器中的条目。否则的话，有可能造成过滤器内恰好就有相同指纹信息的条目。这条原则不仅是对 Cuckoo 过滤器，同样对其他支持删除操作的过滤器也适用。

1.3.4 查询操作

算法 1.3: Cuckoo 过滤器查询操作

```
1  # define FIND_LOCK     locklib_mutex_lock(mutex, 0)
2  # define FIND_UNLOCK     locklib_mutex_unlock(mutex)
3  Function Lookup(x)
4  f = fingerprint(x)
5  i1 = hash(x);
6  i2 = i1 ⊕ hash(f)
7  FIND_LOCK
8  if bucket[i1] 或 bucket[i2] 中含有 f; then
9      FIND_UNLOCK
10     return true
11 end
12 FIND_UNLOCK
13 return false
```

Cuckoo 过滤器的查询操作相对简单。算法 1.3对这个过程做了简单描述。对于给定的条目 x，首先计算其指纹并根据公式 1.24计算出它的两个哈希桶的索引值。然后遍历这些哈希桶：如果在任何一个桶内找到了 x 的指纹，则返回 $true$。否则返回 $false$。值得注意的是，只要哈希桶不发生溢出，就能确保不存在任何误报。

1.4 性能优化与评估

1.5 本章小结

算法 1.4: Cuckoo 过滤器的并发查询过程

```
1  # define FIND_lock     locklib_mutex_lock(mutex, 0)
2  # define FIND_unlock     locklib_mutex_unlock(mutex)
3  Function ConFindTagInBuckets(const i1, const i2, const tag)
4      char *p1 = bucktets_[i1].bits_, *p2 = bucktets_[i2].bits_
5      int v1 = *(*)p1, v2 = *(*)p2
6      FIND_LOCK
7      if f == 4   &&   k == 4 then
8          bool  ret ← hasvalue4(v1, tag) || hasvalue4(v2, tag)
9          FIND_UNLOCK
10         return ret
11     end
12     else if f == 8  &&  k == 4 then
13         bool ret ← hasvalue8(v1, tag) || hasvalue8(v2, tag)
14         FIND_UNLOCK
15         return ret
16     else if f == 12  &&  k == 4 then
17         bool ret ← hasvalue12(v1, tag) || hasvalue12(v2, tag)
18         FIND_UNLOCK
19         return ret
20     else if f == 16  &&  k == 4 then
21         bool ret ← hasvalue16(v1, tag) || hasvalue16(v2, tag)
22         FIND_UNLOCK
23         return ret
24     else
25         for j = 0 to kTagsPerBucket − 1 do
26             if (ReadTag(i1, j) == tag) || (ReadTag(i2, j) == tag) then
27                 FIND_UNLOCK;
28                 return true
29         end
30     end
31     FIND_UNLOCK;
32     return false;
```

参考文献

[1] Bloom B H. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7):422–426

[2] Bender M A, Farach-Colton M, Johnson R, et al. Don't thrash: how to cache your hash on flash. Proceedings of the VLDB Endowment, 2012, 5(11):1627–1637

[3] Bonomi F, Mitzenmacher M, Panigrahy R, et al. An improved construction for counting bloom filters. In: Proc of ESA, volume 6. Springer, 2006, 684–695

[4] Song H, Dharmapurikar S, Turner J, et al. Fast hash table lookup using extended bloom filter: an aid to network processing. ACM SIGCOMM Computer Communication Review, 2005, 35(4):181–192

[5] Yu M, Fabrikant A, Rexford J. BUFFALO: Bloom filter forwarding architecture for large organizations. In: Proc of Proceedings of the 5th international conference on Emerging networking experiments and technologies. ACM, 2009, 313–324

[6] Mitzenmacher M. Compressed bloom filters. IEEE/ACM transactions on networking, 2002, 10(5):604–612

[7] Fan L, Cao P, Almeida J, et al. Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking (TON), 2000, 8(3):281–293

[8] Fan B, Andersen D G, Kaminsky M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing.. In: Proc of NSDI, volume 13. 2013, 385–398

[9] Broder A, Mitzenmacher M. Network applications of bloom filters: A survey. Internet mathematics, 2004, 1(4):485–509

附录 A　读学位期间所发表的学术论文

1. **Zhiwen Chen**, Xin He, Jianhua Sun, and Hao Chen. Have Your Cake and Eat it (too): A Concurrent Hash Table with Hardware Transactions. NPC 2017: THE 14TH IFIP INTERNATIONAL CONFERENCE ON NETWORK AND PARALLEL COMPUTING.(CCF C 类，SCI 刊源)
2. Xin He, **Zhiwen Chen**, Jianhua Sun, Hao Chen, and Dong Li. Exploring Synchronization in Cache Coherent Manycore Systems: A Case Study with Xeon Phi. International Conference on Parallel and Distributed Systems. ICPADS 2017.(CCF C 类)
3. Wenyong Zhong, Jianhua Sun, Hao Chen, Jun Xiao, **Zhiwen Chen**, and Chang Cheng. Optimizing Graph Processing on GPUs. IEEE Transactions on Parallel and Distributed Systems (Volume: 28, Issue: 4, April 1 2017)
4. 吴蓉晖, 汪宁, 孙建华, 陈浩, **陈志文**. 一种针对 JVM 运行时库安全策略的全自动检测方法, 电子学报, 2013, 41(1)：161-165 (EI 收录)

附录 B　读学位期间所参加的科研项目

1. A A A A A A A A A
2. A A A A A A A A A
3. A A A A A A A A A

致　谢

TODO：1 页