

学校代号 10532
分 类 号 TP393

学 号 S1510W0662
密 级 普通



工程硕士学位论文

面向键值对存储的布鲁姆过滤器查询算法设计

学位申请人姓名 潘海娜
培 养 单 位 信息科学与工程学院
导师姓名及职称 谢鲲 教授 文吉刚 高级工程师
学 科 专 业 软件工程
研 究 方 向 计算机网络
论文提交日期 2018 年 5 月 8 日

学校代号: 10532
学 号: S1510W0662
密 级: 普通

湖南大学工程硕士学位论文

面向键值对存储的布鲁姆过滤器查询 算法设计

学位申请人姓名: 潘海娜

导师姓名及职称: 谢鲲 教授 文吉刚 高级工程师

培 养 单 位: 信息科学与工程学院

专 业 名 称: 软件工程

论文提交日期: 2018 年 5 月 8 日

论文答辩日期: 2018 年 5 月 20 日

答辩委员会主席: 邝继顺 教授

The Design of Bloom Filter Algorithm for Key-value Storage

by

PAN Haina

B.E. (Yanbian University) 2015

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of engineering

in

Software Engineering

in the

Graduate school

of

Hunan University

Supervisor

Professor XIE Kun, Senior Engineer WEN Jigang

May, 2018

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：潘海娜 签字日期：2018年5月28日

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

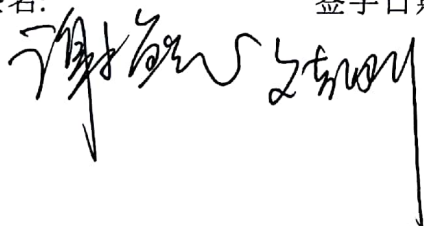
1、保密□，在____年解密后适用于本授权书

2、不保密☒。

(请在以上相应方框内打“√”)

作者签名：潘海娜 签字日期：2018年5月28日

导师签名： 签字日期：2018年5月28日



摘 要

随着互联网科学技术的迅猛发展,物联网、云计算以及移动通信技术的兴起,每时每刻,数以亿计的用户产生着数量巨大的信息,海量数据时代随之而来。布鲁姆过滤器(Bloom Filter, BF)是一种时间效率和空间效率都很高的概率型数据结构,它是用来检测某个元素是否属于一个集合,可以使用它来处理海量数据,进行快速集合元素查询。键值对(key-value)存储是数据库最简单、最常见的一种组织方式,基本上全部的编程语言都带有应用在内存中的键值对存储。传统的键值对存储对于海量数据处理极具挑战,因此,本文对布鲁姆过滤器结构进行扩展,设计了两种面向键值对存储的布鲁姆过滤器查询算法,本文的主要研究点如下:

在分析了目前多种使用布鲁姆过滤器结构进行键值对存储算法的基础上,提出了一种动态存储键值对的可扩展布鲁姆过滤器树(Scalable Bloom Filter Tree, SBFT)结构,并设计了该结构对于键值对的插入、查询、添加新value值的算法。SBFT结构将多个布鲁姆过滤器向量分布在一棵树上,并且本文使用了一种 H_3 哈希函数进一步扩展了这种结构,这种新颖的结构不仅提高了键值对处理速度,而且支持动态数据处理,更适用于现实网络环境中。最后通过理论分析和仿真实验验证了SBFT结构在处理键值对数据方面的有效性。

在分析了多种处理键值对数据时会产生冲突情况的结构特点的基础上,提出了一种基于 B_h 序列的键值对布鲁姆过滤器(B_h -Bloom Filter, B_h -BF),并设计了该结构对于键值对的插入、查询、删除和更新算法。本文选用了一种特殊的编码方式—— B_h 序列,解决了键值对数据插入时产生冲突的多种情况,提高查询效率,降低了误判率。最后使用真实的网络数据集进行仿真实验,实验结果显示,与现有的两种基于布鲁姆过滤器使用单元格cell处理键值对的结构相比, B_h -BF结构能容忍更多冲突情况,错误率更低,处理效率更高,更适合在真实网络应用中使用。

关键词: 布鲁姆过滤器; 键值对存储; 布鲁姆过滤器树; B_h 序列

Abstract

With the development of Internet technology, the rise of cloud computing, Internet of things and the mobile communication technology, hundreds of millions of users generate huge amounts of information at every moment. The age of mass data has come. The Bloom filter is a random data structure with high spatial efficiency and time efficiency. It is used to detect whether an element belongs to a set, which can be used to process massive data and query fast collection elements. The key-value storage is the simplest form of organization of database. Basically all programming languages have key-value pairs stored in memory. Traditional key-value storage are very challenging for mass data processing. Therefore, this paper extends the structure of Bloom filter and designs two query algorithms for key value pairs of Bloom filters. The main research points of this paper are as follows:

Based on the analysis of the key value pairs of Bloom filter structure, an extensible Scalable Bloom Filter Tree structure is proposed, and an algorithm for inserting, querying and adding new values for key value pairs is designed. The structure distributes a number of Bloom filter vectors on a tree, and this paper uses a H_3 hash function to further expand the structure. This novel structure not only improves the processing speed of key value pairs, but also supports dynamic data processing, and is more suitable for the real network environment. Finally, theoretical analysis and simulation experiments verify the effectiveness of the structure in dealing with key values and data.

On the basis of the analysis of the structural characteristics of a variety of processing key values resulting in conflict, a key-value Bloom filter based on B_h sequence is proposed, and the algorithm for inserting, querying, deleting and updating key value pairs is designed. In this paper, a special encoding method, B_h sequence, is used to solve the multiple situations of the conflict when the key value is inserted into the data, so as to improve the efficiency of the query and reduce the rate of misjudgment. Finally, a real network data set is used to carry out simulation experiments. The experimental results show that the B_h -BF structure can tolerate more conflicts, lower error rate, higher processing efficiency and more suitable for use in real network applications to handle key value pairs than the existing two kinds of structure based on the Bloom filter using cell.

Key Words: Bloom Filter; Key-value storage; Bloom Filter Tree; B_h sequence

目 录

学位论文原创性声明和学位论文授权使用授权书	I
摘 要	II
Abstract	III
目 录	IV
插图索引	VII
附表索引	VIII
第 1 章 绪论	1
1.1 研究背景及意义	1
1.2 国内外研究现状	2
1.2.1 可逆的布魯姆过滤器	3
1.2.2 键值对布魯姆过滤器	3
1.2.3 基于布魯姆过滤器的可扩展缓存共享方案	4
1.2.4 面向键值对存储的组合布魯姆过滤器	4
1.2.5 面向键值对存储的布魯姆过滤器树	4
1.2.6 一种存在噪音的布魯姆过滤器	4
1.2.7 面向键值对存储的矩阵索引布魯姆过滤器	4
1.2.8 面向键值对存储的状态布魯姆过滤器	5
1.3 本文主要工作与论文组织结构	5
1.3.1 本文主要工作	5
1.3.2 论文组织结构	6
第 2 章 布魯姆过滤器概述	7
2.1 布魯姆过滤器查询算法	7
2.1.1 布魯姆过滤器插入操作	7
2.1.2 布魯姆过滤器查询操作	7
2.1.3 布魯姆过滤器示例	8
2.2 布魯姆过滤器理论分析	9
2.3 布魯姆过滤器扩展	10
2.3.1 计数式布魯姆过滤器	11
2.3.2 增量可变的计数式布魯姆过滤器	11
2.3.3 布魯姆树	12
2.4 小结	13

第 3 章 动态存储键值对的布鲁姆过滤器树结构	14
3.1 引言	14
3.2 研究背景	14
3.3 键值对存储	15
3.3.1 键值对存储概述	15
3.3.2 键值对存储操作	16
3.4 结构设计	16
3.4.1 布鲁姆过滤器树	16
3.4.2 H_3 哈希函数	17
3.4.3 基于 H_3 哈希函数的 SBFT 结构设计	20
3.5 算法设计	21
3.5.1 Value 值编码设计	21
3.5.2 插入操作	21
3.5.3 查询操作	22
3.5.4 增加新 value 操作	24
3.6 性能分析	25
3.6.1 内存大小分析	25
3.6.2 误判率分析	25
3.6.3 查询速率	26
3.7 实验仿真	26
3.7.1 实验数据集来源	26
3.7.2 实验环境设置	27
3.7.3 实验结果和分析	28
3.8 小结	33
第 4 章 基于 B_h 序列的键值对布鲁姆过滤器	34
4.1 引言	34
4.2 问题来源	34
4.3 结构设计	35
4.3.1 B_h 序列	35
4.3.2 基于 B_h 序列的布鲁姆过滤器设计原则	36
4.3.3 相关介绍	36
4.4 算法设计	37
4.4.1 插入操作	37
4.4.2 查询操作	37
4.4.3 删除操作	39

4.4.4 更新操作	39
4.5 性能分析和仿真实验	39
4.5.1 B_h -BF结构相关参数计算	39
4.5.2 仿真实验	40
4.6 小结	43
结 论	45
参考文献	47
致 谢	52
附录A 发表论文和参加科研情况说明	53
附录 B 攻读学位期间所参与的科研项目	53

插图索引

图 2.1	布鲁姆过滤器插入操作	8
图 2.2	布鲁姆过滤器假阳性误判	8
图 2.3	布鲁姆过滤器示例	9
图 2.4	计数式布鲁姆过滤器	11
图 2.5	增量可变的计数式布鲁姆过滤器	12
图 2.6	布鲁姆树	13
图 3.1	H_3 哈希函数逻辑实现	19
图 3.2	逻辑移位操作	20
图 3.3	布鲁姆过滤器树结构	23
图 3.4	SBFT 节点未满足情况下增加新value操作	24
图 3.5	SBFT 动态增加新value操作	25
图 3.6	静态数据包平均处理时间	29
图 3.7	假阳性误判率	30
图 3.8	动态数据包平均处理时间	31
图 3.9	动态数据内存消耗	32
图 4.1	B_h -BF 结构	36
图 4.2	假阳性误判率	42
图 4.3	假阴性误判率	43

附表索引

表 3.1	实验数据集.....	27
-------	------------	----

第1章 绪论

本章节先介绍本文的研究背景及研究意义，接着对布鲁姆过滤器(Bloom Filter, BF)查询算法的国内外研究情况进行分析，最后概述了本文的主要工作及论文组织结构。

1.1 研究背景及意义

近年来，伴随着互联网信息技术飞速发展，普适计算^[1]广泛应用，在信息检索，交互通信，交易记录，网络监控等各类应用当中产生了大量的数据，这就使得数据密集型计算成了工业和科研界关注的主要领域。与此同时，网络信息空间的飞速增长使得网络中信息存储、表示、搜索和管理等方面面临着巨大挑战。如何选择合适的结构表示海量数据，如何迅速、高效、安全地在这些海量数据中找到自己需要的信息，已经成为众多互联网服务提供商提供服务的基础，也是普通网络使用者关注的热门话题。目前对海量数据进行深度挖掘已经成为一种很普遍的模式，它不仅可以有效了解用户的一些行为趋向，进一步理解用户需求，从中获取用户集体智慧，为研发人员决策提供帮助，不断提升用户对产品体验感受，提高产品市场占有率。因此，当前国内外学术界研究者以及各大互联网公司研究都将重点放在海量数据处理上，但是，如果仅仅寄希望于对硬件进行扩容，很难满足海量数据处理需求，因此，需要设计精简结构的查询算法来保存海量数据，以此提升网络软件体系结构，完成对海量数据的有效管理，如何设计简洁的结构来处理海量数据已经成为国内外学术界研究的关键。

BF^[2]是一种空间节俭、查询高效的随机数据结构，它能迅速检测某个元素是否属于这个集合，满足目前人类生活中各类海量数据信息相互交换以及查找的需求。布鲁姆过滤器的基本原理是一个位串（即位数组）与哈希函数的联合使用来表示集合，进行集合元素查询，并且能够检测出不属于这个集合的元素。BF结构实质上是一个 m 位长的位数组，这个数组中的每一位都初始化为0；然后，选择 k 个不同的哈希函数，每个哈希函数都可以将集合中的各个元素映射到位数组中的某一位。当向BF向量中插入一个元素时，计算 k 个哈希函数，可以得到位数组中的 k 个位，将这些位置依次设置为1，元素插入完成；如果需要检测某个元素是否属于这个集合，那么计算 k 个哈希函数，得到位数组中的 k 个位，查看 k 个位置中的值，如有有一个位置不为1，那么可以肯定这个元素不属于集合，如果 k 个位置全部为1，那么该元素可能属于这个集合，这是因为BF结构可能会产生误判，也就是假阳性误判(False Positive)^[2,3]。布鲁姆过滤器是以牺牲正确率为前提换取

空间效率和时间效率的提高，当它判断某个元素不属于集合时，那么该元素一定不属于这个集合，即不会存在假阴性误判(False Negative)；而当它判断某元素属于这个集合时，该元素不一定属于这个集合。因此，BF是一种可以容忍一定误判的概率型数据结构，适用于可以容忍低错误率的场合。

布鲁姆过滤器在进行元素查找时的时间复杂度是常数，而且它消耗的存储空间也比较小，使得这种算法的实用价值极高。BF是1970年由布鲁姆首次提出^[2]，因此被命名为布鲁姆。它可以用来表示海量数据、进行元素过滤以及提高元素的查询效率，现已被广泛应用到各种网络应用当中。目前布鲁姆过滤器比较常见的网络应用主要集中在字典查询、数据库操作和文件操作等一些方面^[4-12]。最近十几年里，随着计算机网络研究迅猛发展，各种新型网络应用技术随之出现，将BF算法的研究推向了前端，并将它应用到网络领域各方面^[13,14]：包括资源路由^[15]、覆盖网络以及在P2P网络节点上协作交互^[16-22]、网络测量管理^[23-25]、流数据管理^[26,27]、网络入侵检测^[28,29]、数据帧标签^[30]等各种网络应用领域中。A.Brooder和M.Mitzenmacher曾预言^[14]：虽然目前各领域对BF结构的网络应用开发还是寥寥无几，但是现今国内外研究学者对BF结构投注了越来越多地关注和研究，它必然会在新的学术领域和未来计算机网络系统中得到更深一步的研究和更为广泛的应用。

在高速的网络环境中，流量分类（可以根据源IP地址、目的IP地址、源端口号、目的端口号、协议ID等进行流量的分类）和数据包处理等问题都涉及到快速查表操作，即键值对操作。实际应用就是对于一个已知的键值key，比如说IP地址、MAC地址或者URL等，通过查表操作，快速返回与之对应的value值，比如说相关的端口号，这就是典型的键值对操作，相关的网络应用包括近似状态机^[13]，2层交换机中帧转发^[31]，网络流量分类^[31]，IP路由选择^[32]等等。针对这些应用，已经有许多面向键值对存储的布鲁姆过滤器查询算法被设计出来。它们设计的目标就是构建一种简洁的数据结构，能够进行快速的键值对操作并且它们结构自身内存消耗低，适用于现实的网络环境。对于研究布鲁姆过滤器查询算法来说，不管是对算法本身进行扩展，还是将它应用于当前网络中，都具有非常重大的意义。

1.2 国内外研究现状

目前，国内外学者对BF算法的研究成果较多，在它的应用方面也取得了不错的成绩，在此进行概括和分析。近几年对于BF的研究已经引起国内外学者的关注，并且在一些国际顶级网络和算法研究会议及杂志(Transaction on Networking, Infocom, Sigcomm, Sigmod等)上发表了论文。本文对目前国内外文献中一些关

于布鲁姆过滤器的改进算法做了整理，主要有如下几种：计数式布鲁姆过滤器(Counting Bloom filter)^[33]，光谱布鲁姆过滤器(Spectral Bloom filter)^[34]，压缩式布鲁姆过滤器(Compressed Bloom filter)^[3]，可逆布鲁姆过滤器(Invertible Bloom Filter)^[35]，动态布鲁姆过滤器(Dynamic Bloom filter)^[36]，键值对布鲁姆过滤器(k Bloom Filter)^[37]，变量增加布鲁姆过滤器^[38]，多维布鲁姆过滤器^[36]，联合多维布鲁姆过滤器查询算法^[39]，布鲁姆树(Bloom Tree)^[40]等。标准的BF多种扩展结构的出现，极大推动了BF的发展、研究和应用。

标准的BF只可以对元素是否在集合中进行查询以及过滤掉不属于集合的元素。然而，目前学术界对键值对存储的问题进行了大量的研究和实践，取得了很有启发性的成果。近年来有不少学者将布鲁姆过滤器这种良好的数据结构应用在键值对存储过程中^[13,31,32,40-44]，对标准的布鲁姆过滤器进行了扩展，取得了良好的成果。本文是面向键值对存储的布鲁姆过滤器查询算法的设计，在查阅了大量国内外相关参考文献的基础上，下面从国内外两个方面介绍多种面向键值对存储的布鲁姆过滤器查询算法设计结构，其中前六种是国外的相关研究现状，后两种是国内的相关研究现状：

1.2.1 可逆的布鲁姆过滤器

David等人提出了一种可逆的布鲁姆过滤器(Invertible Bloom Filter, IBF)^[35,45]结构，可以通过一些可逆的运算还原部分甚至所有插入的元素值。IBF包含了一组单元格(cells)，使用单元格代替标准BF向量中的比特位，每个单元格都是一个计数器。插入数据时，对插入同一个单元格的键值key先进行哈希操作，然后进行异或运算，将结果放入单元格中；查询时，先将单元格计算器中只有1的位置的值直接识别出来，然后使用异或运算依次将剩下的元素还原出来，重复这样的过程直到所有元素识别完。IBF主要的缺陷就是内存损耗较大，因为它使用的是单元格而不是比特位；并且如果单元格中没有单独存在的值，那么会有冲突情况发生。

1.2.2 键值对布鲁姆过滤器

Xiong等人提出了键值对布鲁姆过滤器(kBF)^[37]，它也是使用单元格cell，每个单元格包括一个计数器和一个存储结果的空间。kBF与IBF最大的不同就是，它为每一个value值选择了一个特殊的编码，使得每个单元格中只要存在3个元素以下个数元素时，都可以将键值对还原出来。kBF只可以推测cell中不超过3个元素的编码，否则也会有冲突产生，这种结构的查询成功率也比较低。

1.2.3 基于布魯姆过滤器的可扩展缓存共享方案

Fan 等人提出了一种可扩展的缓存共享方案(Summary Cache)^[33], 它本质上是一种直接使用布魯姆过滤器解决键值对存储操作的方案。这种方案在构造时为每一个value值分配一个布魯姆过滤器, 插入元素时, 在对应value值的布魯姆过滤器上进行 k 次哈希操作; 查询元素时, 需要查找所有的布魯姆过滤器, 只要有一个满足条件, 这个布魯姆过滤器对应的value就是要查找的value。

1.2.4 面向键值对存储的组合布魯姆过滤器

Hao等人进一步提出了一种组合的布魯姆过滤器(Combinatorial Bloom filter, COMB)方案^[31], 这种结构在元素编码时, 将元素编码为一个 L 位二进制向量, 其中只有 θ 位bit为1, 剩余 $L-\theta$ 位都是0。COMB会生成 L 个布魯姆过滤器, 编码时, 只需要将二进制向量位为1的对应位置的布魯姆过滤器进行编码插入即可, 这样, 就只需要进行 $k \times \theta$ 次哈希, 这种方法又进一步加快了数据处理速度。

1.2.5 面向键值对存储的布魯姆过滤器树

Yoon等人总结了上述方法以后, 发现COMB结构在查询元素对应组时, 需要对所有布魯姆过滤器执行 k 次哈希计算, 计算开销非常巨大。为了改善这一点, Yoon等人第一次使用树这种数据结构来进行键值对相关操作, 提出了布魯姆树(Bloom Tree, BT)^[40]这种巧妙的数据结构。根据已知的数据, 得出value组数, 这个数值就是所搭建树的叶节点数量, 根据这个数值就可以把树搭建出来, 每一个叶节点表示一组value。编码一项(key,value)时, 根据value值, 会在树中生成一条唯一的路径, 只需要沿着这条路径对key值进行哈希计算即可。这种树的结构不仅操作简单, 存储容量大, 而且内存访问量较低。

1.2.6 一种存在噪音的布魯姆过滤器

由于使用BF结构进行键值对存储操作已经较为普遍, 因此提出了一种存在噪音的布魯姆过滤器(Noisy Bloom Filters, NBF)和错误纠正的噪音布魯姆过滤器(NBF-E)结构^[46], 它使用一种特殊的ID编码value, 插入键值对时将噪音插入, 查询过程就变成了一个降噪的过程。NBF-E结构使用了一种不对称的纠错编码技术, 提高了查询的正确率。

1.2.7 面向键值对存储的矩阵索引布魯姆过滤器

以MySQL, Oracle为代表的传统关系型数据库在过去为大家所熟识并被广泛应用, 但在现代化海量数据处理应用中仍存在大量缺陷。如果使用B+树搭建索引结构, 会使得查询复杂度偏高, 文献 [47] 提出了一种面向键值对存储的矩阵索引

布鲁姆过滤器查询算法，这种算法为布鲁姆过滤器结构搭建了一种索引，这样的方式不仅可以降低结构所需的存储空间，提高索引结构搭建效率，而且可以大大减少查询时间复杂度。

1.2.8 面向键值对存储的状态布鲁姆过滤器

为了解决高速网络中TCP流-状态的记录查询问题，实际上也就是键值对操作问题，设计了一种面向键值对存储的状态布鲁姆过滤器查询算法(stateBF)^[48]。这种结构不仅内存消耗低，而且查询快速。stateBF也是使用单元格取代布鲁姆过滤器中的比特位，使用一种独热码的编码方式对流状态信息进行编码，解决了一些冲突情况。

1.3 本文主要工作与论文组织结构

1.3.1 本文主要工作

基于上述文献分析，目前已经存在大量使用布鲁姆过滤器查询算法进行扩展的应用，它们的目的就是进行键值对存储等操作。本文查阅了大量相关文献，发现这些相关结构都存在不同程度的缺陷：内存消耗大、查询速度慢、错误率高等。本文针对这些问题，以提高面向键值对存储的布鲁姆过滤器查询算法各方面性能为目的，设计了两种相关结构以及相应操作的算法，解决了现有结构内存开销大、查询速率低、误判率较高等缺陷。本文的主要创新点如下所述：

1、在分析了目前多种使用布鲁姆过滤器进行键值对存储的算法特点的基础上，提出了一种动态处理键值对的可扩展布鲁姆过滤器树(Scalable Bloom Filter Tree, SBFT)结构，并设计了该结构对于键值对的插入、查询、添加新value值算法。本文将布鲁姆过滤器分布于搜索树结构中，使用 H_3 哈希函数^[49]以及相关的移位操作，巧妙地减少哈希计算量；并且可以在不知道数据量的情况下，对布鲁姆树进行扩展，实现动态数据插入。使用该算法后，键值对操作速度大大加快，并且可以动态处理数据。最后通过理论分析和仿真实验验证了SBFT结构在处理键值对数据方面的有效性。本文研究可以突破现有键值对存储操作的局限，大大扩展了现有大数据研究的领域，具有十分重要的理论意义和实践价值。

2、在分析了多种处理键值对数据时会产生冲突情况的结构特点的基础上，提出了一种基于 B_h 序列的键值对布鲁姆过滤器(B_h -BF)，并设计了该结构对于键值对的插入、查询、删除和更新算法。针对 B_h 序列^[50]中 h 的选择，本文解决了处理键值对数据时产生冲突的情况。最后使用真实的网络数据集进行实验，实验显示与现有的两种基于布鲁姆过滤器使用单元格cell处理键值对的结构比较， B_h -BF结构能容忍更多冲突，错误率更低，处理效率更高，更适合在网络应用中使用。

1.3.2 论文组织结构

本文一共由五章节组成，具体内容安排如下：

第1章，绪论。首先本文介绍了相关的研究背景及研究意义，然后对布魯姆过滤器查询算法的国内外研究情况作了简单的介绍，最后概述了本文的主要工作以及论文组织结构。。

第2章，布魯姆过滤器概述。首先对布魯姆过滤器查询算法的原理作了阐述，先是进行理论分析，然后做了具体实例分析，最后介绍了本文参考的主要结构的一些布魯姆过滤器的扩展技术。

第3章，介绍了动态存储键值对的可扩展布魯姆树结构。本文设计了一种可扩展的布魯姆过滤器树结构(Scalable Bloom Filter Tree, SBFT)，并设计了该结构对于键值对的插入、查询、添加新value值算法，该算法不但查询速率快，而且可以处理动态数据，性能极优。

第4章，介绍了基于 B_h 序列的键值对布魯姆过滤器。本文设计了一种基于 B_h 序列的布魯姆过滤器(B_h -BF)，用来处理键值对操作，并设计了该结构对于键值对的插入、查询、删除和更新算法，该算法不但解决了冲突问题，而且误判率较低，查询速度也很快。

最后，总结本论文工作，展望今后的研究方向。对本文工作进行归纳总结并对面向键值对存储的布魯姆过滤器查询算法设计这个研究方向进行展望。本文最后是参考文献和致谢部分。

第2章 布鲁姆过滤器概述

布鲁姆过滤器(Bloom Filter, BF)是一种可以精简地表示集合并且支持高效快速集合查询的数据结构,它能够满足目前人类生活中海量数据相互交换及查找的需求,现已被广泛应用在各种网络应用中。BF是一种允许产生一定误判的数据结构,适用于对较低错误率可以接受的一些场合。本章首先介绍了标准BF的原理和基本操作,然后对BF的性能进行分析,最后介绍了几种与本文相关的布鲁姆过滤器扩展。

2.1 布鲁姆过滤器查询算法

BF实质上就是使用一个位串向量来表示集合,并且支持元素的哈希查找。首先,位串向量中的每一位都初始化为0,为了存储一个包含 n 个元素的数据集合 $X = \{x_1, x_2, \dots, x_n\}$,可以使用 k 个哈希函数将每一个元素映射到位串向量 V 中, k 个哈希函数结果分别是 h_1, h_2, \dots, h_k ,位串向量 V 就是布鲁姆过滤器向量,用BF表示。向量 V 的总长度为 m , k 个哈希函数是相互独立,各不相关,并且它们的取值范围是 $0, 1, 2, \dots, m-1$ 。BF查询算法的基本操作主要涉及两个方面:插入操作和查询操作。

2.1.1 布鲁姆过滤器插入操作

BF对集合中元素进行插入操作,就是将数据集合中的所有元素插入到位串向量 V 中,以便于后面查询元素时,可以利用这个布鲁姆过滤器向量查询元素是否属于这个集合中,过滤掉不属于集合的元素。

首先,对于一个需要插入布鲁姆过滤器的元素 $x \in X$,先将布鲁姆过滤器向量初始化,即把位串向量 V 中的每个比特位初始化为0;然后,使用选取的 k 个互不相关的哈希函数进行计算,得到元素 x 的 k 个哈希地址,分别是 $h_1(x), h_2(x), \dots, h_i(x), 1 \leq i \leq k$;最后,将得到的这 k 个地址在BF向量对应的 k 个比特位置上置1,即:

$$BF[h_1(x)] = BF[h_2(x)] = \dots = BF[h_i(x)] = 1, 1 \leq i \leq k \quad (2.1)$$

如图 2.1 所示,将元素 x 插入到布鲁姆过滤器向量的操作完成。

2.1.2 布鲁姆过滤器查询操作

布鲁姆过滤器查询操作与插入操作类似,对于要查询的元素 x ,使用与上方相同的 k 个哈希函数,计算元素 x 的 k 个哈希地址,得到

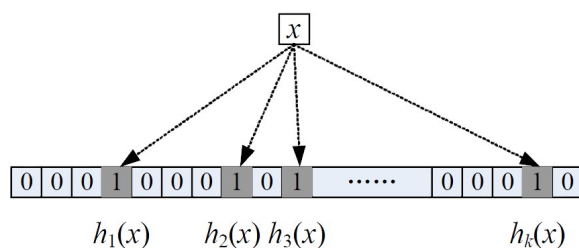


图 2.1 布魯姆过滤器插入操作

$h_1(x), h_2(x), \dots, h_i(x), 1 \leq i \leq k$; 然后, 在布魯姆过滤器向量上检查这 k 个位置

$$BF[h_1(x)], BF[h_2(x)], \dots, BF[h_i(x)], 1 \leq i \leq k \quad (2.2)$$

是否都是1。如果这 k 个比特位上有一个为0, 可以肯定这个元素 x 不属于这个集合 X ; 如果都为1, 那么元素 x 可能属于这个集合 X , 但不一定。这是因为布魯姆过滤器有些情况会出现假阳性误判, 即将不属于这个集合的元素即没有插入过的元素误判为属于这个集合^[2,3]。如图 2.2 所示, 元素 a, b, c, d 是属于集合的元素, 它们插入布魯姆过滤器之后, 发现不属于这个集合的元素 x 的所有映射位置都为1, 导致元素 x 被误判为属于集合中, 这是元素 a, b, c, d 插入引起的, 这就是假阳性误判。布魯姆过滤器不存在假阴性误判, 即将属于集合的元素误判为不属于集合, 因为元素插入到布魯姆过滤器向量中后, 对应置位肯定会为1, 不会存在其中有一个位置为0的情况。

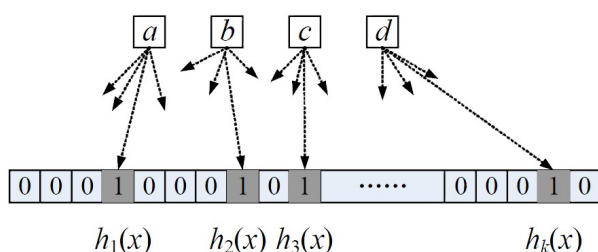


图 2.2 布魯姆过滤器假阳性误判

2.1.3 布魯姆过滤器示例

如图 2.3 所示, 给出了传统BF查询算法的具体实例。BF初始化参数: 向量长度 $m = 5bit$, 选用的哈希函数个数 $k = 2$, 选择的两个哈希函数分别为 $h_2(x) = (2x + 3) \bmod 5$ 和 $h_1(x) = x \bmod 5$ 。如图 2.3 所示, 已经插入到布魯姆过滤器中的集合 $X = \{9, 11\}$, 对15和16两个元素进行查询。

现有各种BF查询算法的扩展和应有都是在传统BF的研究基础上进行的, 本文也是以此为基础, 接下来会对标准BF的各项参数进行分析。

	$h_1(x)$	$h_2(x)$						
初始化:			<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0
0	0	0	0	0				
插入元素 9:	4	1	<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	0	0	1
0	1	0	0	1				
插入元素 11:	1	0	<table><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	1	1	0	0	1
1	1	0	0	1				
	$h_1(x)$	$h_2(x)$						
查询元素 15:	0	3	No,15不在集合中(正确的回答)					
查询元素 16:	1	0	Yes,16在集合中(错误的回答: <i>false positive</i>)					

图 2.3 布鲁姆过滤器示例

2.2 布鲁姆过滤器理论分析

BF结构实质上就是把集合中的元素通过 k 个不同的哈希函数映射到比特位串向量中，一个元素只需保存 k 个比特位。正是因为BF查询算法结构简单、易于实现，使它具有很好的实用价值，虽然它存在一些误判的情况，但也正是因为它的优良性能，使得这些缺陷可以被容忍。BF最大的缺陷就是存在假阳性误判(False Positive)，即将不属于一个集合的元素误判为属于这个集合，但是BF不会存在假阴性误判(False Negative)，即把属于一个集合中的元素误判为不属于这个集合^[2,3]。

使用布鲁姆过滤器最重要的一点就是对它的误判率做正确的评估，将误判率控制在能接受的范围内。BF误判率大小主要跟三个参数有关：布鲁姆过滤器向量大小 m ，使用的哈希函数个数 k ，以及需要插入的集合元素个数 n 。选用合适的哈希函数个数以及布鲁姆过滤器大小，都可以有效控制误判率。本文使用三元组 m, k, n 的形式来表示一个BF，更加便于下文说明。

首先，在本文的讨论中，需要假设哈希函数的取值全部服从均匀分布。当插入一个元素到BF向量时，这个位串向量中的任意一个比特位通过一个哈希函数未被置1的概率是：

$$1 - \frac{1}{m} \quad (2.3)$$

那么，使用 k 个哈希函数，这个位串向量 V 中的任意一个比特位为0的概率是：

$$\left(1 - \frac{1}{m}\right)^k \quad (2.4)$$

集合中 n 个元素全部插入完成后，使用 p 表示位串向量 V 中任意一个比特位为0的概率，得到的结果是：

$$p = \left(1 - \frac{1}{m}\right)^{kn} \quad (2.5)$$

因此，任意一个比特位为1的概率就是 $(1 - p)$ 。在检验一个元素是否属于集合时，如果经过 k 个哈希函数计算后，这 k 个位置在位串向量对应位置的都为1，BF就认为这个元素属于集合，此时，如果这个元素不属于集合，就产生了误判，即一个元素查询误判率为^[3]：

$$f(m, k, n) = (1 - p)^k \quad (2.6)$$

又因为 $e^{-kn/m}$ 非常接近于 p ^[3]，因此：

$$f(m, k, n) = (1 - p)^k = (1 - e^{-kn/m})^k \quad (2.7)$$

BF结构在实际应用过程中往往会给出一个可以容忍的误判率 f_0 ，假设BF长度 m 以及哈希函数个数 k 大小固定不变，这样根据式(2.7)就可以通过计算得到BF可以表示的集合元素个数是 n_0 ^[51]：

$$n_0 = -\frac{\ln(1 - e^{\ln(f_0/k)}) \cdot m}{k} \quad (2.8)$$

接下来，针对计算得到的误判率即式(2.7)进行分析：布鲁姆过滤器向量越大，误判率会越低；插入布鲁姆过滤器元素个数越多，它的误判率也会越高。如果想减小误判率，就需要选择合适的哈希函数个数 k 。对式(2.7)求导，使得到的结果等于0，可以得到 k 的取值^[52]：

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n} \quad (2.9)$$

后文在实验过程中往往需要根据BF向量的大小 m 来搭建结构，因此需要计算出 m 的大小，根据式(2.7)，可得^[53]：

$$m = -\frac{kn}{\ln(1 - f(m, k, n)^{1/k})} \quad (2.10)$$

标准的BF在进行集合元素插入时需要计算 k 次哈希函数，因此元素插入时一次哈希计算的时间复杂度 $O(k)$ 。查询操作与插入操作类似，因此元素查询时一次哈希计算的时间复杂度也是 $O(k)$ 。当需要插入 n 个元素到BF中时，BF长度为 m ，所以它的空间复杂度是 $O(m)$ 。因此，可以发现BF表示集合时，一个元素平均只需要存储 $\frac{m}{n}$ 位，这也就验证了BF是一种空间简洁的数据结构。

本文大多数仿真实验中，都是根据上面式子选取合适的布鲁姆过滤器参数，使它误判率在一定范围内，从而使布鲁姆过滤器效率得到极大提高。

2.3 布鲁姆过滤器扩展

布鲁姆过滤器最大的优势就是空间简洁和查询简便，但是它只能够进行元素的插入和查询操作，这就限制了布鲁姆过滤器的使用。现如今出现了各种各样布鲁姆过滤器的扩展，使得它的使用得到普及。

2.3.1 计数式布鲁姆过滤器

传统的BF查询算法只可以进行集合元素的插入和查询操作，不能进行集合元素删除操作。由于BF向量在插入集合元素时，不同的元素可能会在相同的位置置1，BF中所有位置都被集合中元素共享，如果按照元素插入操作的相反操作进行删除，将要删除的元素对应 k 个映射位置置0，就可能将其它元素的相应置位改变，导致后续查询操作错误。因此，为了使BF能够进行删除操作，文献 [33] 给出了一种布鲁姆过滤器扩展-计数式布鲁姆过滤器(Counting Bloom Filter, CBF)，如图 2.4 所示，将标准布鲁姆过滤器向量 V 中的每个比特位替换为count计数器。

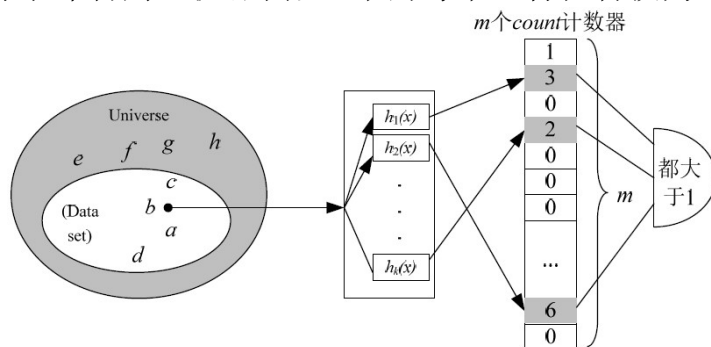


图 2.4 计数式布鲁姆过滤器

使用计数布鲁姆过滤器CBF时，首先将向量 V 中 m 个count计数器全部初始化为0，然后进行后续的集合元素插入和查询操作，并且还能进行删除操作。将集合元素插入CBF时，只需要将对应的 k 个映射位置上的count计数器分别加1；而删除元素时，与插入操作相反，将对应 k 个映射位置上的count计数器分别减1即可；查询一个元素是否属于集合时，查看对应 k 个映射位置上的count计数器是否都大于1。

CBF对标准布鲁姆过滤器查询算法进行了改进，由静态的元素数据集转换为动态数据集，集合元素操作不再仅仅限制于插入和查询，还可以进行删除。CBF使用count计数器代替向量中的比特位，使得集合中的元素可以进行动态删除。但是，CBF的存储空间与count计数器的长度成正比，如果count计数器长度较长会导致布鲁姆过滤器向量需要占用的存储空间成倍增长；而count计数器长度较小会导致集合元素插入时计数器会溢出，产生错误。当在实际的网络应用中使用CBF时，为了避免不必要的错误，提前预估CBF中count计数器长度对于后续元素查询会有很大的帮助。

2.3.2 增量可变的计数式布鲁姆过滤器

CBF现已被广泛应用于各种网络设备算法中，它可以快速表示一个集合并支持集合元素查询，相比于传统的布鲁姆过滤器，它还可以进行元素删除操作。然

而，CBF会消耗大量内存，因此，文献 [38] 提出了一种增量可变的计数式布魯姆过滤器(VI-CBF)，改善了原计数式布魯姆过滤器的效率，如图 2.5 所示。

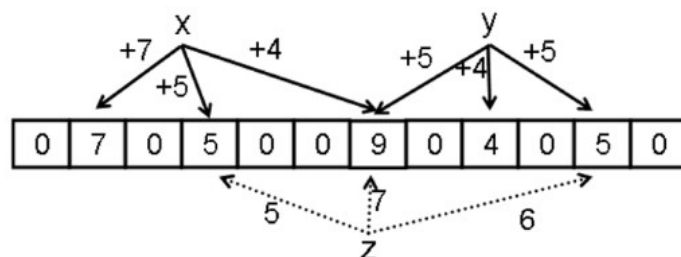


图 2.5 增量可变的计数式布魯姆过滤器

这种VI-CBF结构不像CBF结构那样，插入元素时每个计数器加1，它会增加一个特殊的值，这个特殊的值是从一种 B_h 序列中挑选出来的，后文会给出 B_h 序列的详细信息。这种VI-CBF结构插入元素时，首先计算 k 个哈希函数，在 B_h 序列的对应 k 个位置挑选出 k 个值，将这 k 个值插入到布魯姆过滤器向量的 k 个位置中，如图 2.5，此时选取的 B_h 序列 $D=4,5,6,7$ ，元素 x 经过 k 个哈希函数计算，得到的三个相加的值分别为：7,5,4，再将它们插入到布魯姆过滤器对应位置中，对于元素 y 也是相同的操作。查看元素 z 是否属于集合时，元素 z 经过 k 个哈希函数计算，得到的三个相加的值分别为：5,7,6，如图 2.5， z 计算的第二个位置计数器值为9，因为 $9 - 7 = 2$ ，因此7不可能和序列中的其它元素相加得到9，因此可以判定 z 不属于集合。元素删除操作和插入操作类似，将对应位置的计数器值减掉序列中的值即可。

VI-CBF结构是对CBF结构的一大改进，这种算法不仅可以减小计数式布魯姆过滤器需要的内存空间，还可以降低布魯姆过滤器的误判率。如图 2.5 所示，可以准确得到 z 不属于集合，而原来的CBF结构则仍然会将 z 判定为属于集合，因为对应位置的计数器数都是大于1。这种VI-CBF结构最重要的步骤，就是要选取合适的 B_h 序列，下文会给出详细介绍。

2.3.3 布魯姆树

对于计数式布魯姆过滤器来说，仅仅还是停留在集合元素的一些相关操作上。存储和查询键值对($key, value$)是计算机系统中常见的任务，这就需要设计对应的键值对存储的数据结构，支持快速的键值对处理操作。文献 [40] 提出了一种布魯姆树结构，这种结构不仅可以实现键值对操作，而且相较于其他具有相同功能的结构而言，它的准确率更高，空间也更加简洁。如图 2.6 所示就是布魯姆树结构。

布魯姆树结构是一棵完全 d 叉树，本文选择 $d = 2$ 即二叉树作为说明。其特征在于，每一棵完全二叉树中的每一个节点都是一个布魯姆过滤器；每一棵完全二叉树的每个叶子节点表示一个值 $value$ ；每一个节点的存储单元大小是该节点的父

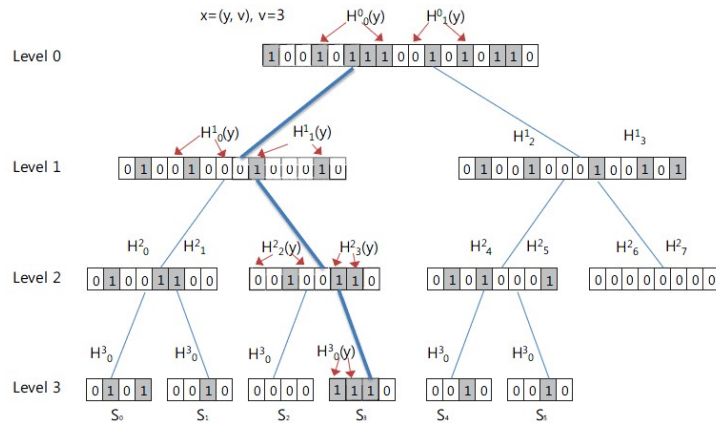


图 2.6 布鲁姆树

节点的存储单元大小的一半，每个节点需要选取 $2 * k$ 个不同的哈希函数，即每个节点包含两个哈希组，每组中有 k 个哈希函数。首先需要将所有要插入键值对的值分配给树的叶节点，插入键值对时，根据插入键值对的值 $value$ ，得到一条由根节点到叶节点的唯一路径，沿着这条路径，选择合适的哈希组，将 key 值插入到每一层布鲁姆过滤器中；查询 key 对应的 $value$ 时，由根节点开始，计算 2 组哈希函数，如果第一组满足，接下来查询左节点，如果第二组满足，则再查询右节点，如果两组都不满足，直接判定 key 值不存在，继续上述方法，直到达到叶节点为止，查询到的叶节点对应的 $value$ 就是 key 值对应的 $value$ 。

布鲁姆树结构不再局限于集合元素的简单操作，进一步对布鲁姆过滤器进行扩展，使它能够支持键值对操作。布鲁姆树结构不仅查询效率高，而且易于在网络应用中实现。本文第三章节的工作也是使用布鲁姆树结构作为基础进行改良设计和仿真。

2.4 小结

本章介绍了标准布鲁姆过滤器查询算法的基本原理和操作，给出了一个简单布鲁姆过滤器的示例，然后对布鲁姆过滤器性能进行了分析，最后介绍了几种与本文相关的布鲁姆过滤器的扩展算法。

第3章 动态存储键值对的布魯姆过滤器树结构

3.1 引言

近年来,伴随着计算机科学技术迅猛发展,计算机网络系统、数据库以及其他一些网络应用中的集合数据规模飞速上升。对于海量数据来说,高效的存储、管理和处理等各方面技术都面临着前所未有的挑战。键值对(*key, value*)存储是计算机网络系统中最普遍的任务之一,因其存储模式简单、易于扩展的特性在近几年受到广泛的关注和应用。键值对存储系统对于键值对处理操作非常简便,但是为了快速进行键值对存储,往往会将它们放置于内存中,这样做虽然查询的精度很高,但是是以消耗内存和CPU时间为代价的。

传统的键值对存储往往依赖于数据库进行操作,这也就限制了键值对必须存储于磁盘中。在过去几十年间,CPU处理速度增加了600倍,而磁盘速度只提高了10倍^[47],如此缓慢的磁盘速度必然限制了数据快速读取。因此,设计高效的键值对存储结构算法已经引起国内外研究学者的广泛关注。为了提高(*key, value*)存储系统的查询速度,近年来大量国内外文献对布魯姆过滤器结构进行扩展,设计了多种基于(*key, value*)存储的布魯姆过滤器结构,不仅提高了查询速度,而且大大优化了内存空间。

3.2 研究背景

在一个2层交换机中,一个MAC地址会与一个唯一的端口号相关联。当要转发一个数据帧时,搜索引擎会查询这一数据帧要转发的目的地址的MAC表,如果这个目的地址在表中不存在,这一帧就会被发送到所有输出端口。因此,将一个MAC地址映射到一个端口的问题就转换成了一个键值对查询问题,此时,MAC地址被看成键值*key*,而要查询的端口号就变成了值*value*。由于MAC地址是持续添加到列表中的,因此,元素的大小未知。使用完全哈希(*perfect hashing*)^[54]是解决这类问题的一种相应的方法,但是这种方法计算非常密集,不适合在硬件上实现。

另一种键值对存储的网络应用就是使用布魯姆过滤器进行IP转发。文献 [55]介绍了在IP网络中使用布魯姆过滤器结构进行数据包转发的方法。这种结构叫做BUFFALO,它为每一个输出端口分配了一个布魯姆过滤器向量,当一个数据包到达路由器时,它的目的IP地址就被映射到每个输出端口对应的布魯姆过滤器向量中。如果找到了匹配的端口,这个数据包就会转发过去;如果这个数据包跟

多个布鲁姆过滤器匹配，因为存在假阳性误判，这篇文献提出了一种概率型方案：将数据包发送到概率为1目的地址。

在现今高速发展的计算机网络系统中，如何高效存储键值对信息，实现快速查询操作已经成为一大挑战。为了支持键值对存储相关操作，在查阅大量面向键值对存储的布鲁姆过滤器查询算法的文献的基础上发现：有将一个单独的比特空间划分为多个独立单元格(cell)的，每个单元格中放置不同的值，但是在提取数据时，由于单元格中放了多个数据，会使得数据提取时存在错误，造成空间的浪费^[35,37,45]；有使用多个布鲁姆过滤器向量组合起来进行键值对存储的结构，他们为每一个value组分配一个布鲁姆过滤器^[33,56]或者根据一些编码来分配布鲁姆过滤器的数量^[31]，这些结构在查询元素时内存访问量特别大，查询误判率高，消耗的内存也较大；还有一种方法是将布鲁姆过滤器应用在一棵搜索树中^[40]，每一个叶子节点对应一组，每一个插入对象对应一条唯一的路径，这种结构大大降低了查询元素时内存访问量，容量增加，并且误判率也降低了，但是这种结构的哈希计算量巨大，并且不能进行扩展，只有在数据量已知的情况下才能使用。

本文发现布鲁姆树(Bloom Tree, BT)^[40]这种特殊的结构是第一次将树这种数据结构应用到了此类问题中，并且大大降低了内存访问量，加大了存储容量，降低了误判率。然而，本文发现布鲁姆树在处理数据的过程中需要进行大量哈希计算，增加了处理数据需要的时间；而且在构造布鲁姆树结构时，需要提前知道键值对组的大小，而在现实网络环境中，网络数据往往是不可预测的，数据都是持续添加到系统中，这种不能支持动态数据插入的结构特别不实用，正因为这些问题存在，使得对存储系统进行扩展成为计算机研究的热点。本文在此结构的基础上提出了一种可扩展的布鲁姆过滤器树结构(SBFT)，这种动态可扩展的布鲁姆过滤器树结构对于需要键值对存储的网络应用领域中，比如高速网络中进行资源定位、数据库交互查询等产生海量数据的应用，不仅可以减少集合查询所需的时间，而且可以处理动态数据，降低资源消耗，更适应真实的网络环境。

3.3 键值对存储

本节对键值对存储的概念进行简单说明，对键值对存储的基本操作进行简单介绍。

3.3.1 键值对存储概述

键值对(key, value)存储是NoSQL存储的一种形式，它是按照键值对的组织形式处理数据，对数据进行存储和索引。(key, value)存储特别适合那些与过多数据关系的业务数据没有关联的情况，同时读写磁盘的次数也比较少，相比于SQL数据库存储，(key, value)存储具有更好的读写能力。(key, value)存储是数据库中最常

见并且最简单的一种组织形式，基本上所有编程语言都会有键值对存储应用于内存中：C++ STL库中的map和hash map等映射容器，Java中的hash table和hash map，以及Python中的字典类型等等。这些组织形式往往只支持内存操作，并且查询效率比较低，因为这些都是简单的数据结构，不能对大规模存储进行实现，并且对数据进行修改很复杂，因此，本文使用布魯姆过滤器进行扩展，实现键值对操作，提高性能。

3.3.2 键值对存储操作

键值对存储一般会有如下三个接口：

1. Set(key,value): 将value存储到存储空间中标识符key下，如果key下已经保存了数据，使用value替代旧数据；
2. Get(key): 通过查询标识符key，查找key下保存的数据，如果没有数据则报错；
3. Delete(key): 通过查询标识符key，将key下保存的数据删除。

本文设计的算法往往会依据这3个接口进行设计，并在这3个接口的基础上扩展键值对存储应用。

3.4 结构设计

本章此部分首先概述布魯姆过滤器树设计原则，然后介绍 H_3 哈希函数以及移位操作的详细信息，最后描述动态可扩展的布魯姆过滤器树结构的设计思路。

3.4.1 布魯姆过滤器树

本文找到了一种使用布魯姆过滤器进行键值对操作的结构——布魯姆过滤器树^[40]，这种结构不仅空间简洁，而且比其它具有相同功能的结构效率上更高：内存访问次数少、存储容量大、误判率低。而布魯姆树能有这么好的性能得益于它的这种新型结构的组织方式——多个布魯姆过滤器组织到一棵树上。

布魯姆过滤器树结构就是一棵完全 d 叉树，本文为了说明方便，选取 $d = 2$ ，即一棵完全二叉树进行说明。图 2.6 是布魯姆树的结构。这棵完全二叉树的每一个节点都是一个布魯姆过滤器，叶节点表示不同的value值。给定一组键值对(key, value)，根据value值的编码会得到一条从根节点到叶节点的唯一路径，这棵树的高度取决于value组数。当要插入一组键值对(key, value)时，这条路径上的每个节点都会将key插入到布魯姆过滤器中；当需要查询key对应的value时，则需要从根节点到叶节点之间不断进行检验，直到得到一条唯一的路径，找到相应的value值编码，再通过解码操作得到对应的value值。其中，根节点包括 $2 * k$ 个不相同的哈希函数，即根节点包括2个哈希组，每组中包含 k 个哈希函数。

布鲁姆过滤器树结构虽然减少了内存访问次数，但是在使用时，每组键值对都需要计算多组哈希函数，大大降低了数据处理速度；并且，布鲁姆树只有在value组数已知的情况下才能搭建，即只能处理静态数据，不能动态添加键值对数据，这也就限制了它的实际应用。下文会对这些限制提出解决方案。

3.4.2 H_3 哈希函数

标准的BF结构在搭建时往往会使用设定的参数大小，因此它通常会用来表示一些静态的数据集合，BF向量在使用时通常会提前知道插入的集合元素大小和实际应用中的能接受的最大误判率，根据这些就可以计算它所需要的布鲁姆过滤器向量的大小和哈希函数个数。在布鲁姆树中，每一层过滤器向量大小可以不一致，这样就可以应用一种特殊的哈希函数，它可以在新的布鲁姆过滤器向量范围内，以较低的开销得到哈希值。

Wegman和Carter在文献 [57] 中提出了一种普遍的哈希函数(Universal Hash)—— H_3 哈希函数，它自身具有较强的散列性，是布鲁姆过滤器结构使用的一种较为普遍的实现函数^[58,59]。 H_3 哈希函数在使用过程中，对每个输入元素进行计算时，仅需要一些简单的“与”和“异或”运算，这也使得它在硬件上易于实现， H_3 哈希函数是计算机硬件经常见使用的哈希函数之一^[60,61]。

H_3 哈希操作时实际上是一次线性转换 $B^T = Q_{r \times w} A^T$ ，它是将一个 $w - bit$ 长度的二进制地址元素 $A = a_1 a_2 \cdots a_w$ 转换为一个 $r - bit$ 的二进制地址 $B = b_1 b_2 \cdots b_r$ ，即：

$$\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_r \end{pmatrix} = \begin{pmatrix} q_{11} & q_{12} & \cdots & q_{1w} \\ q_{21} & q_{22} & \cdots & q_{2w} \\ \cdots & \cdots & \cdots & \cdots \\ q_{r1} & q_{r2} & \cdots & q_{rw} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_w \end{pmatrix} \quad (3.1)$$

其中转换矩阵 $Q_{r \times w}$ 是一个0,1矩阵， H_3 哈希函数实质上就是一个转换矩阵。计算过程中，乘法运算是采用二进制与 $AND(\bullet)$ 运算，而加法运算则是采用二进制或 $XOR(\oplus)$ 运算，即：

$$b_i = (a_1 \bullet q_{i1}) \oplus (a_2 \bullet q_{i2}) \oplus \cdots \oplus (a_w \bullet q_{iw}) (i = 1, 2, \cdots, r) \quad (3.2)$$

下面给出 H_3 哈希函数两个具体示例。第一个示例中选用的 H_3 哈希函数是：

$$Q_{3 \times 8} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \quad (3.3)$$

其中， $w = 8, r = 3$ ，输入的元素经过 H_3 哈希函数进行转换，由 $\{0, \dots, 2^8 - 1 = 255\} \rightarrow \{0, \dots, 2^3 - 1 = 7\}$ 。经过此 H_3 哈希函数计算，元素179由式(3.3)计算可得：

$$h_1(179) = h_1(10110011) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \quad (3.4)$$

其中， $\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}^T = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix} = 6(decimal)$ 。元素179经过哈希函数 h_1 计算后得到 $h_1(179) = 6$ 。

第二个示例中选用的 H_3 哈希函数是：

$$Q_{4 \times 8} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad (3.5)$$

其中， $w = 8, r = 4$ ，与上述计算过程类似，经过此 H_3 哈希函数计算，元

素179由式(3.5)计算可得:

$$h_2(179) = h_2(10110011) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \quad (3.6)$$

其中, $\begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}^T = \begin{pmatrix} 1 & 1 & 0 & 1 \end{pmatrix} = 13(decimal)$ 。元素179经过哈希函数 h_2 计算后

得到 $h_2(179) = 13$ 。

H_3 哈希函数是使用二进制逻辑与 $AND(\bullet)$ 运算, 和二进制逻辑异或 $XOR(\oplus)$ 运算, 如图 3.1 所示。

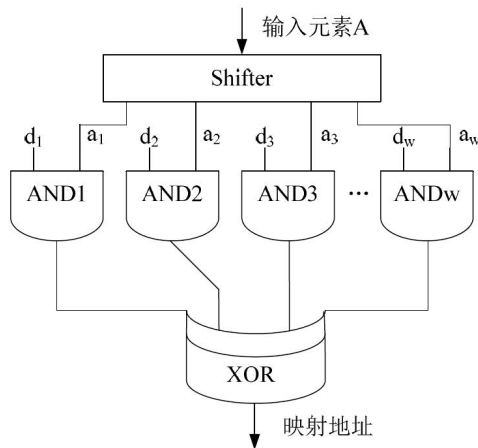


图 3.1 H_3 哈希函数逻辑实现

H_3 哈希函数在计算过程中需要 W 个AND与门以及一个XOR异或门, 其中移位器是用来获取输入元素的每一位, 每个 H_3 哈希函数的列向

量 $d_1, d_2, \dots, d_w \in [0, 2^r - 1]$ 相互独立，所以经过异或门得出的结果就是最后计算得到的哈希地址。

这两个示例使用不同的哈希函数计算元素179，转换后得到了两个不同的值，比较两个 H_3 哈希函数(3.3)和(3.5)，发现式(3.5)比式(3.3)多了一行，并且它们前3行是一模一样的。接下来，本文调查了这些相似矩阵之间的关系，并且为布鲁姆过滤器树设计了一种轻量级的查询算法。

3.4.3 基于 H_3 哈希函数的SBFT结构设计

基于 H_3 哈希函数，本文发现，通过一些简单的移位操作，可以得出一些规律。经过哈希计算发现，最终得到的哈希值的范围往往取决于相应转换矩阵的行数。因此，本文可以选取一个基矩阵，再根据不同布鲁姆过滤器向量的长度范围，从基矩阵中选择合适的行数，以此来得到不同的哈希值范围。布鲁姆树中的每个布鲁姆过滤器向量需要 $2 * k$ 个哈希函数，而本文的SBFT结构仅仅只需要生成一组 $2 * k$ 个的哈希函数，也就是 $2 * k$ 个基矩阵即可。根据布鲁姆树顶层过滤器向量大小 m ，可以得出树高 $l = \log_2 m$ ，因此，只需要在 $2 * k$ 个基矩阵中提取每个基矩阵的前 l 行作为顶层的哈希函数即可，接下来就可以通过移位操作直接得到下一层的哈希值，依次类推，下面介绍移位操作的具体步骤。

如图3.2所示，是逻辑移位操作的两个具体示例。” $<<$ ”是符号左移位，将运算数的二进制整体左移指定位数，低位用0补齐。” $>>$ ”，有符号右移位，将运算数的二进制整体右移指定位数，整数高位用0补齐，负数高位用1补齐（保持负数符号不变）。

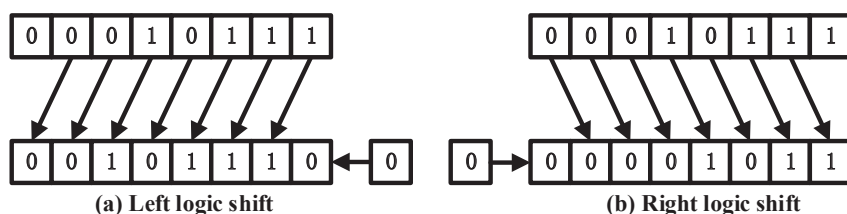


图 3.2 逻辑移位操作

上一节通过两个示例得出哈希值的推导方法，第一个例子中， $h_1(179) = (110)$ ，第二个例子 $h_2(179) = (1101)$ ，其中对应的哈希矩阵分别是(3.3)和(3.5)， h_1 的前三行与 h_2 一样。根据逻辑移位操作发现， $h_1(179) = h_2(179) >> 1 = (1101) >> 1 = (110)$ ，那么元素179经 h_1 哈希函数计算得到的值可以直接使用 h_2 哈希函数计算得到的值向右移一位得到。

如果在布鲁姆树的顶层选取合适的 H_3 基函数，那么得到的值再经过简单的向右移位操作就可以得到下一层的值，这样的操作不再需要选取大量哈希函数，而且计算速度也迅速提升，这也为后面动态数据处理做了铺垫。

3.5 算法设计

3.5.1 Value值编码设计

本文设计的SBFT结构实质上是一棵多叉树，为了便于说明本文选用二叉树进行说明。首先介绍一下哈弗曼编码^[62]：对一棵具有 n 个叶子节点的哈夫曼树，若对树中的每个左分支设置为编码0，右分支设置为编码1，则从根节点到每个叶子节点的通路上，各分支的赋值会分别构成一个二进制串，该二进制串就称为哈弗曼编码。哈夫曼树被广泛地应用，其中最典型的就是在编码技术上的应用，本文对value值使用的就是哈弗曼编码。

3.5.2 插入操作

本文设计的这棵SBFT树结构中每一个布鲁姆过滤器节点的存储单元大小是该节点的父节点存储单元大小的一半。当需要插入一个键值对($key, value$)时，首先查看value是否已经插入到布鲁姆过滤器树结构中，如果未插入，则需要进行后续的增加新value操作；如果value已经存在于布鲁姆过滤器树中，则直接进行键值对插入。根据value查找该value值对应的叶节点，即获取此叶节点位置的哈弗曼编码，确定一条从根节点到这个叶字节点的唯一路径；然后，计算根节点上的两组哈希函数，即对key使用 k 个哈希函数 $h_{(i,1)}, h_{(i,2)}, \dots, h_{(i,k)}$ ，计算 $h_{(i,1)}(key), h_{(i,2)}(key), \dots, h_{(i,k)}(key)$ ，其中 i 表示选取哈希函数的组号， $i = 1$ 或 2 ；将 $i = 1$ 计算的哈希值保存在数组A中， $i = 2$ 计算的哈希值保存在数组B中；接着，根据叶节点的第一位编码，在根节点选取A、B数组中的一组值进行插入，即对A或B数组进行右移零位操作，然后根据叶节点的第一位编码，得到下一个需要进行插入操作的布鲁姆过滤器，再根据叶节点的第二位编码，选取A、B数组中的一组值进行右移一位操作，得到插入位置，进行插入；继续进行上述操作，对每一层布鲁姆过滤器插入key，直到插到value对应的叶节点为止。

上述插入操作中，在根节点选取待插入的数组的方法为：如果编码值为0，则选取A数组，如果编码值为1，则选取B数组。根据叶节点的第一位编码，得到下一个需要进行插入操作的布鲁姆过滤器的方法为：如果编码值为0，则操作左节点，如果编码值为1，则操作右节点。算法3.1描述了这个过程：

下面给出一个具体示例：图3.3是SBFT结构示意图，它将布鲁姆过滤器与二叉树结构相结合，每一个节点都是一个BF。根节点有两组 H_3 哈希函数 H_1^2, H_2^2 ，每组有 k 个哈希函数。一个叶节点就表示一个value，根据叶节点的编码可以得到一条由根节点到叶节点的唯一路径。根据叶节点编码，如果编码值为0，则对第一组哈希函数进行移位操作；如果编码值为1，则对第二组哈希函数进行移位操作。图3.3中 H_1^1 就是 $H_1^2 \gg 1$ 得来的， H_2^1 就是 $H_2^2 \gg 1$ 得来的，而 H_1^0 是 $H_1^2 \gg 2$ 得来的，

算法 3.1 *SBFT* 的插入操作**Input:**

键值对(key,value)

Output:更新后的 $B_h - BF$

```

1:  $S = GroupId(value)$ ;
2: for  $1 \leq i \leq h$  do
3:    $number[i] = compute(S)$ ;
4: end for
5: for  $0 \leq i \leq k - 1$  do
6:    $A[i] = H_1 i(key)$ ;
7:    $B[i] = H_2 i(key)$ ;
8: end for
9: for  $1 \leq j \leq h$  do
10:  if  $number[j] == 0$  then
11:    for  $0 \leq i \leq k - 1$  do
12:       $C[i] = (A[i] \gg (j - 1))$ ;
13:       $CBF[C[i]] ++$ ;
14:    end for
15:    转到左节点;
16:  else
17:    for  $0 \leq i \leq k - 1$  do
18:       $C[i] = (B[i] \gg (j - 1))$ ;
19:       $CBF[C[i]] ++$ ;
20:    end for
21:    转到右节点;
22:  end if
23: end for

```

H_2^0 是 $H_2^2 \gg 2$ 得来的。

3.5.3 查询操作

与插入操作相反，当需要查询一个键值key对应的value值时，首先，计算根节点的两组哈希函数 $h_{(i,1)}(key), h_{(i,2)}(key), \dots, h_{(i,k)}(key) (i = 1 \text{ or } 2)$ ，将它们的值分别保存在A、B两组数组中；在根节点分别对A、B两数组值对应的位置单元进行查询操作，查到第一位编码值；根据得到的编码值，转到下一个节点继续查询，此时，对A、B两数组进行右移操作，在对应的位置单元继续布鲁姆过滤器查询操作，如果A数组满足则编码值为0，如果B数组满足则编码值为1，此时得到一个编码值；

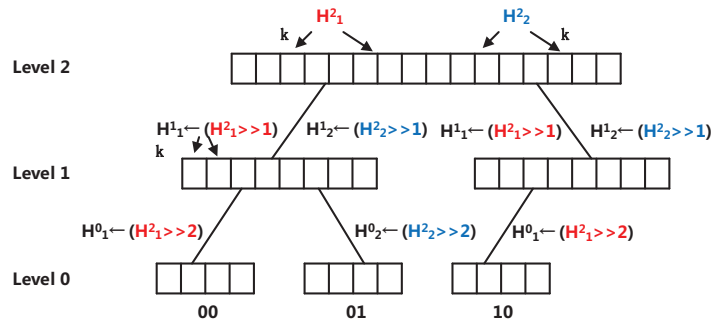


图 3.3 布鲁姆过滤器树结构

根据这一个编码值，确定下一步进行操作的节点，如果编码值为0，则操作左节点，如果编码值为1，则操作右节点；继续进行上述操作，直到查找到叶节点为止，最后得到一个完整的编码值，根据这个叶节点编码进行相应的解码操作得到对应的value值。算法3.2描述了这个过程：

算法 3.2 *SBFT* 的查询操作

Input:

键值key

Output:

value值

```

1: for  $0 \leq i \leq k-1$  do
2:    $A[i] = H_1 i(key);$ 
3:    $B[i] = H_2 i(key);$ 
4: end for
5: for  $1 \leq j \leq h$  do
6:   for  $0 \leq i \leq k-1$  do
7:      $C[i] = (A[i]/B[i] >> (j-1));$ 
8:     if  $CBF[C[i]] > 0$  then
9:       if  $i \neq k$  then
10:        continue;
11:       else
12:        number[j] = 0或1;
13:        转到左或右节点;
14:       end if
15:     else
16:       break;
17:     end if
18:   end for
19: end for

```

3.5.4 增加新value操作

要添加一个新的value到SBFT中，分以下两种情况：

1. 如果原来的布魯姆过滤器树是一棵未滿二叉树，当需要添加新的value时，为这个value分配一个新的哈夫曼编码，然后直接在布魯姆过滤器树的末尾添加新的叶节点，这个叶节点表示新添加的value；
2. 如果原来的布魯姆过滤器树是一棵滿二叉树，则在根节点的上方增加一个新布魯姆过滤器，这个新增加的布魯姆过滤器的大小是原布魯姆过滤器树根节点的2倍，此时，这个新的布魯姆过滤器就变成了新的布魯姆过滤器树的根节点，原来的布魯姆过滤器树就变成了这个新根节点的左子树，再创建一棵比原布魯姆过滤器树高少一层的滿二叉树作为新的根节点的右子树，把原根节点上置1的位置全部左移一位，插入到新的根节点中，此时新的根节点的两组 H_3 哈希函数比原来的两组 H_3 哈希函数多一层，即在基函数中多选取一行，再把原来的所有哈夫曼编码前加一位0；接下来可以继续在新的布魯姆过滤器树中按照第一种情况增加新的叶节点。

图3.4是在未滿二叉树中增加新的value操作示意图。当一个未存在于叶节点的value需要插入时，如果二叉树不是滿二叉树，则可以直接在树的末尾添加叶节点，表示插入的value。如图3.4中编码为11的叶节点就是新添加的value值的位置。

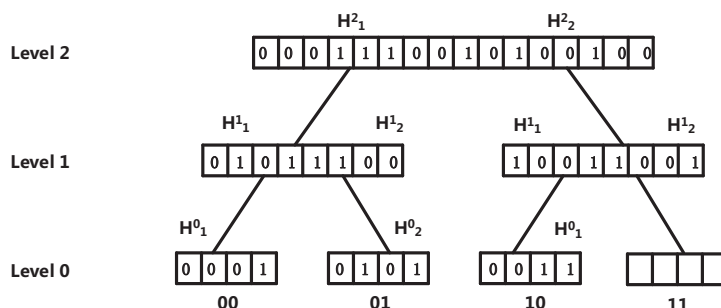


图 3.4 SBFT节点未滿情况下增加新value操作

图3.5是在滿二叉树中增加新的value操作示意图。当一个未存在于叶节点的value需要插入时，如果二叉树是滿二叉树，则不可以在原树上添加叶节点，此时需要往上扩展一层，如图3.5中新添加的level 3。原二叉树就变成了level 3的左子树，再构建一棵比原二叉树少一层的滿二叉树作为level 3的右子树。此时，level 3也存在两组 H_3 哈希函数 H_1^3, H_2^3 ，其中 H_1^3 是由 $H_1^2 \ll 1$ 得到，将原二叉树根节点中所有置位向左移一位，插入到level 3中；而 H_2^3 则是从原来的基矩阵中比 H_2^2 多选取一行即可。此时构建了一棵新的未滿二叉树，可以直接在树的末尾添加叶节点，表示插入的value。如图3.5中编码为100的叶节点就是新添加的value。

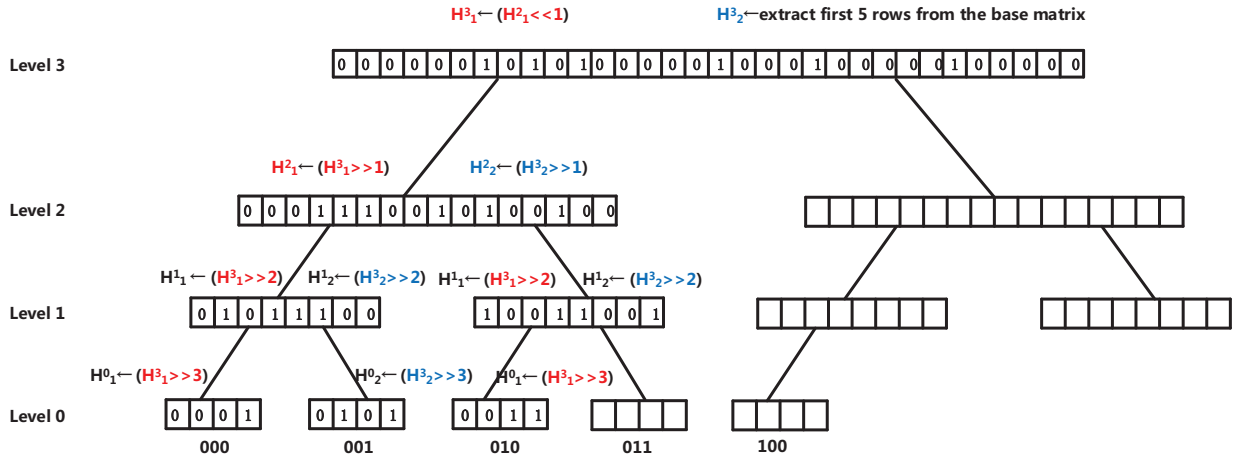


图 3.5 SBFT动态增加新value操作

3.6 性能分析

这一节会给出动态布鲁姆树的理论分析，针对基于布鲁姆过滤器进行键值对操作的结构，主要有三种性能指标：内存大小，错误率以及查询速率，本章会从这三个方面来验证SBFT的特征。

3.6.1 内存大小分析

SBFT占用的内存是原始树结构所有节点的内存加上扩展之后增加的节点内存。首先，本文定义原始的布鲁姆树使用的内存大小是 m ，树高是 h ，由此可以计算出顶层布鲁姆过滤器向量大小是： $\frac{m}{h+1}$ 。假设动态布鲁姆树扩展了 s 次，那么树高就变为 $H = h + s$ ，这样就可以通过累加每层布鲁姆过滤器向量大小，直接计算出SBFT使用的内存大小：

$$M = 2^s \cdot \frac{m}{h+1} \cdot H + \frac{m}{h+1} \quad (3.7)$$

3.6.2 误判率分析

SBFT误判率就是对于一个未被插入的键值key，经过SBFT查询后返回了一个真实的value。SBFT结构较好的性能就是每经过树的一层，误判率就会降低一半。假设插入树中的键值对一共有 n 组，SBFT共扩展了 s 次，上方计算出了总共使用的内存大小 M ，SBFT中插入元素后，一个比特位仍旧为0的概率是：

$$p_0 = (1 - \frac{1}{M})^{kn} \approx e^{-\frac{kn}{M}} \quad (3.8)$$

其中， k 是SBFT中使用的总哈希函数个数。 $e^{-\frac{kn}{M}}$ 非常接近于 $(1 - \frac{1}{M})^{kn}$ ，因此，可以得出一个比特位为1的概率是：

$$p_1 = 1 - p_0 \quad (3.9)$$

当一个从未被插入到树中的键值key在查询过程中返回了一个value值，此时，误判率就会发生。SBFT扩展了 s 次，相应的组数扩展到了 G 个，因此，误判率发生的概率就是：

$$p_f = 1 - (1 - p_1^k)^G \quad (3.10)$$

3.6.3 查询速率

在高速的网络环境中，SBFT的查询速率是一项非常重要的性能。本文通过计算SBFT的数据访问时间验证它的查询速率。SBFT数据访问时间包括哈希函数计算时间，查询时间以及进行移位操作的时间。

假设SBFT结构已经扩展了 s 次，使用的哈希函数个数为 k ，那么哈希函数计算时间是 $O(k)$ 。插入一组键值对到SBFT中仅需要 kH 次移位操作，最理想的情况是查询一个键值时，每次计算第一组哈希函数时就查询到，那么就仅需要 $k + kH$ 次查询操作和 kH 次移位操作；而最不理想的情况就是查询一个键值时，每次计算最后一组哈希函数才查询到，那么就需要 $k + kdH$ 次查询操作和 $kd(H - 1) + k$ 次移位操作。由此，可以计算出平均需要的数据处理时间复杂度分别是： $O(k(2 + H + dH)/2)$ 和 $O(k(H + dH - d + 1)/2)$ 。由上式可以看出，SBFT数据处理时间往往取决于树总的高度。

3.7 实验仿真

本章节会通过实验仿真来验证SBFT的性能，并将实验结果与最近几年研究成果进行了对比，其中包括：Bloom Tree^[40]是2014年INFOCOM中的论文“Bloom tree: A search tree based on bloom filters for multiple-set membership testing”提出的一种将树和布鲁姆过滤器结构结合起来进行键值对查询的结构，该算法在每一个布鲁姆过滤器节点上都使用了 d 组哈希函数，从根节点开始计算，直到找到value对应的叶节点为止，该算法只能对静态数据进行处理；COMB^[31]是2012年IEEE/ACM Transactions on Networking中的论文“Fast dynamic multiple-set membership testing using combinatorial bloom filters”提出的一种基于布鲁姆过滤器的键值对查询算法，该算法将value值进行编码后，对已知的value组数选取合适的布鲁姆过滤器数目和需要插值的布鲁姆过滤器数目，查询时则需要对所有布鲁姆过滤器向量进行查询，得到相应的编码。

3.7.1 实验数据集来源

本文的实验运行Intel(R) Pentium(R) CPU G630 @ 2.70GHz 以及8.00GB RAM的主机上，采用Java语言编写程序。为了更好地说明实验数据多样性，本章使用的实验数据集如表 3.1，表 3.1 是实验数据集详细信息，包括数据来源，数据时间，

包个数。

表 3.1 实验数据集

数据集	数据时间	包个数
MAWI 1	2016-07-09[14:00:00-14:14:59]	60494902
MAWI 2	2016-12-10[14:00:00-14:14:59]	76187324
MAWI 3	2017-01-05[13:00:00-13:15:00]	77588547
ClarkNet-HTTP	[1995-08-28-00:00:00]–[1995-09-10-23:59:59]	3328587
UMass	[2008-06-02]–[2008-06-07]	611968
TKN	2001-7-18[127 minutes]	25861

1. MAWI^[63]: MAWI是WIDE项目维护的数据集。本文使用的此类数据集捕获了15分钟从日本到美国的主干数据网上的日常流量。此类数据集对外是公开的,并且对包中的IP地址和内容都做了特殊处理,保证了隐私性。
2. ClarkNet-HTTP^[64]: ClarkNet-HTTP数据集是位于加拿大阿尔伯塔省的卡尔加里卡尔加里大学计算机系的WWW服务器中采集的HTTP请求。
3. UMass^[65]: UMass数据集是在YouTube网站流量上关于一个校园网络测量的流量的收集,包含了关于特定YouTube内容的用户请求的跟踪数据。
4. TKN^[66]: TKN数据集包含了标准电影的跟踪文件,如《侏罗纪公园》。

3.7.2 实验环境设置

本章实验先处理静态数据,选取的内存大小 $m = 2^{20}bit$,根节点占用了95325bit,选取了18行8列的H3哈希函数基矩阵,在根节点上提取了基矩阵的前16行,处理组数 $g = 1024$ 组的数据,此时树高 $h = 10$;然后,在此基础上再进行数据组添加操作,进行动态数据插入时,选取的哈希函数个数 $k = 3$,处理动态数据时内存大小会产生变化。

为了对数据包中的键值对进行处理,每当遇到一个键值对数据时,首先会对键值key进行查询。如果这个key值从未插入到SBFT结构中,那么就对这组键值对进行插入操作,将它插入到SBFT中;如果这组键值对已经插入到了SBFT结构中,SBFT会查看查找的value值是否相一致,如果一致则进行下一组键值对操作,如果不一致,则先删除原来的那组键值对,然后使用插入操作,插入新的键值对到SBFT结构中,即完成键值对的更新操作。这样就将整个数据包插入到SBFT结

构中，后续实验过程是对整个数据包插入到各个结构中所花费的平均处理时间进行比较。

本文会使用一张额外的表保存值value和它的编码值，为了后续解码操作做准备。在处理动态数据包时，添加新的叶子节点需要先查询这张表，是否value值已经存在，本文使用了一个额外的布鲁姆过滤器结构来完成这个操作，使得查询更加简便。

通过对SBFT结构深度了解，本文从四个方面对SBFT结构的性能进行评估：

1. 静态数据包平均处理时间，定义为对于已知数量的数据包全部插入到结构中，一个数据包的平均处理时间。
2. 假阳性误判率，如果一组键值对未插入到SBFT结构中，而对于一个key值，查询之后返回了一个value值，定义为这种错误发生的次数占总查询次数的比例。
3. 动态数据包平均处理时间，定义为对于未知数量的数据包全部插入到结构中，数据插入完成后，统计数据包总数量，计算一个数据包的平均处理时间。
4. 内存消耗，定义为处理动态数据包时，所搭建结构的内存消耗状态。

3.7.3 实验结果和分析

实验结果图 3.6 是固定大小的布鲁姆过滤器树 $m = 2^{20} \text{bit}$ ，在哈希函数个数 k 不断变化的情况下，三种不同结构：SBFT，布鲁姆树(BT)和COMB，处理1024组键值对数据，对数据包平均处理时间(即静态数据包平均处理时间)进行比较。图 3.6(a)–(f) 分别是使用数据集MAWI 1、MAWI 2、MAWI 3、ClarkNet-HTTP、UMass、TKN进行仿真实验得到的实验结果示意图。实验过程中，为了衡量三种结构处理数据包时所需的时间，在每一种结构插入数据过程中添加了一个计时器，得出插入数据所需要的时间。图 3.6 中所有实验结果显示，针对每种不同的数据集，本文提出的算法SBFT处理数据包所消耗的平均时间最少。与BT相比，SBFT结构只需要在根节点选取 d 组哈希函数计算，然后进行移位操作，而BT则需要每个节点都选取 d 组哈希函数进行计算，耗时较大；与COMB结构相比结果类似，COMB对每一个布鲁姆过滤器都是需要选取多个哈希函数，消耗的时间比较多，但COMB需要操作的布鲁姆过滤器数目比BT少，因此，耗时会比BT小。由于哈希函数个数 k 越多，数据操作时耗时会越多，因此，随着哈希函数个数增多，数据平均处理时间会增多。通过这个实验结果可以发现，处理静态数据包时，三种结构虽然消耗内存一致，但是本文设计的动态处理键值对的可扩展布鲁姆过滤器树结构消耗的时间是最少的，处理速度较快，更适用于真实的网络环境。

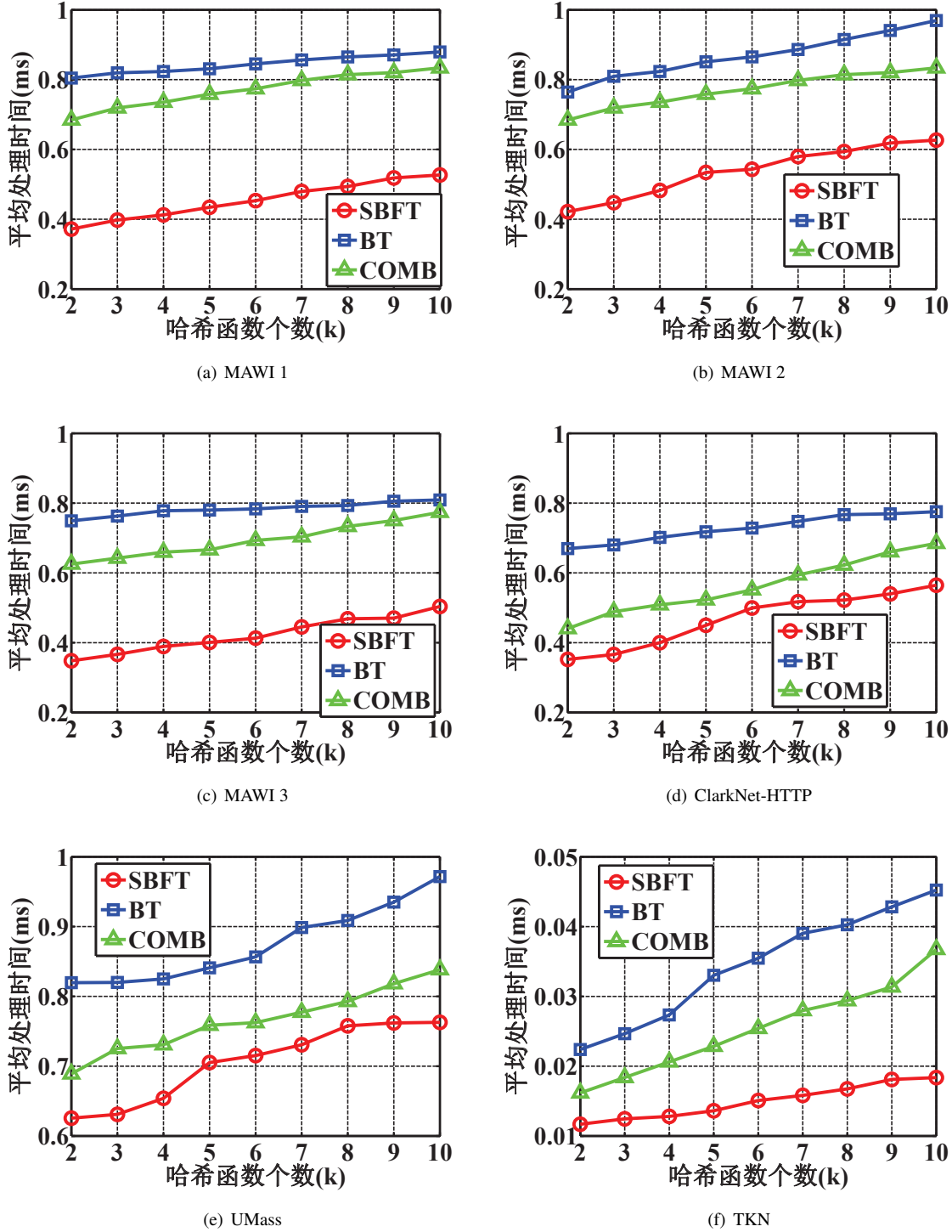


图 3.6 静态数据包平均处理时间

实验结果图 3.7 是在大小一定的布鲁姆过滤器树 $m = 2^{20} \text{bit}$ ，在哈希函数个数 k 不断变化的情况下，三种不同结构：SBFT，BT 和 COMB，插入 1024 组键值对数据之后，再选取 1024 组未插入结构的数据进行查询，查看有多少数据被误判为插入结构中，即对它们的假阳性误判率进行比较。图 3.7(a)–(f) 分别是使用数据集 MAWI 1、MAWI 2、MAWI 3、ClarkNet-HTTP、UMass、TKN 进行仿真实验得到的实验结果示意图。图 3.7 中所有实验结果显示，针对每种不同的数据集，本文

设计的SBFT结构和BT结构假阳性误判率相似，因为在处理静态数据时，它们结构是一样的，只是使用的哈希函数类型不同。与COMB结构相比，SBFT结构的假

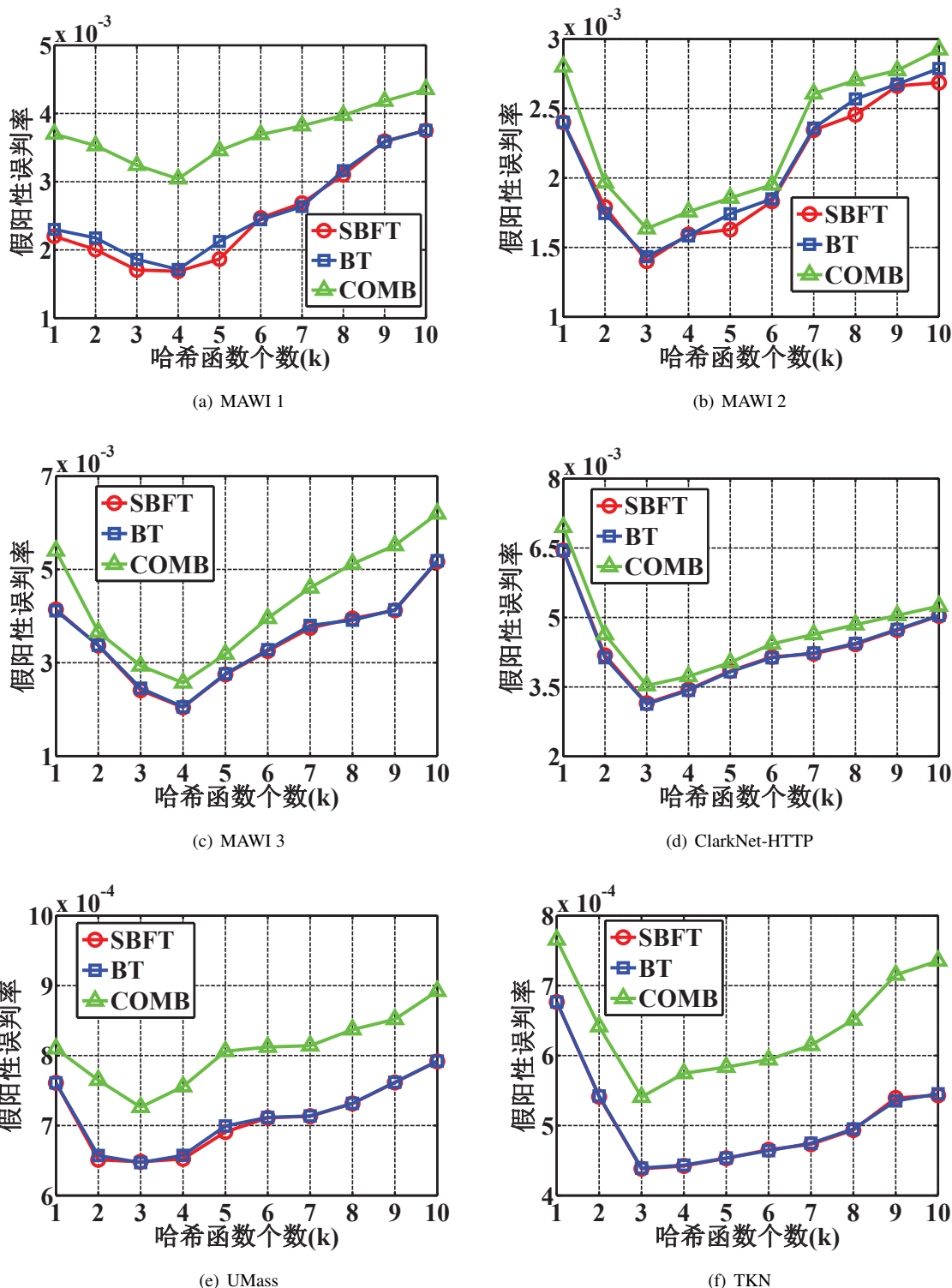


图 3.7 假阳性误判率

阳性误判率较小，因为SBFT这种树类型的结构，每一层都会对数据进行一次筛选，从而降低了误判率。由于哈希函数个数 k 会有一个最合适值，当到达这个值时，会使得布魯姆过滤器结构的假阳性误判率最小。根据图 3.7 可以看出，数据集MAWI 2、ClarkNet-HTTP、TKN选取的最优哈希函数个数 $k = 3$ ，数据

集MAWI 1、MAWI 3选取的最优哈希函数个数 $k = 4$ 。在实际操作中，根据可以容忍的误判率，选取合适的哈希函数个数 k 。

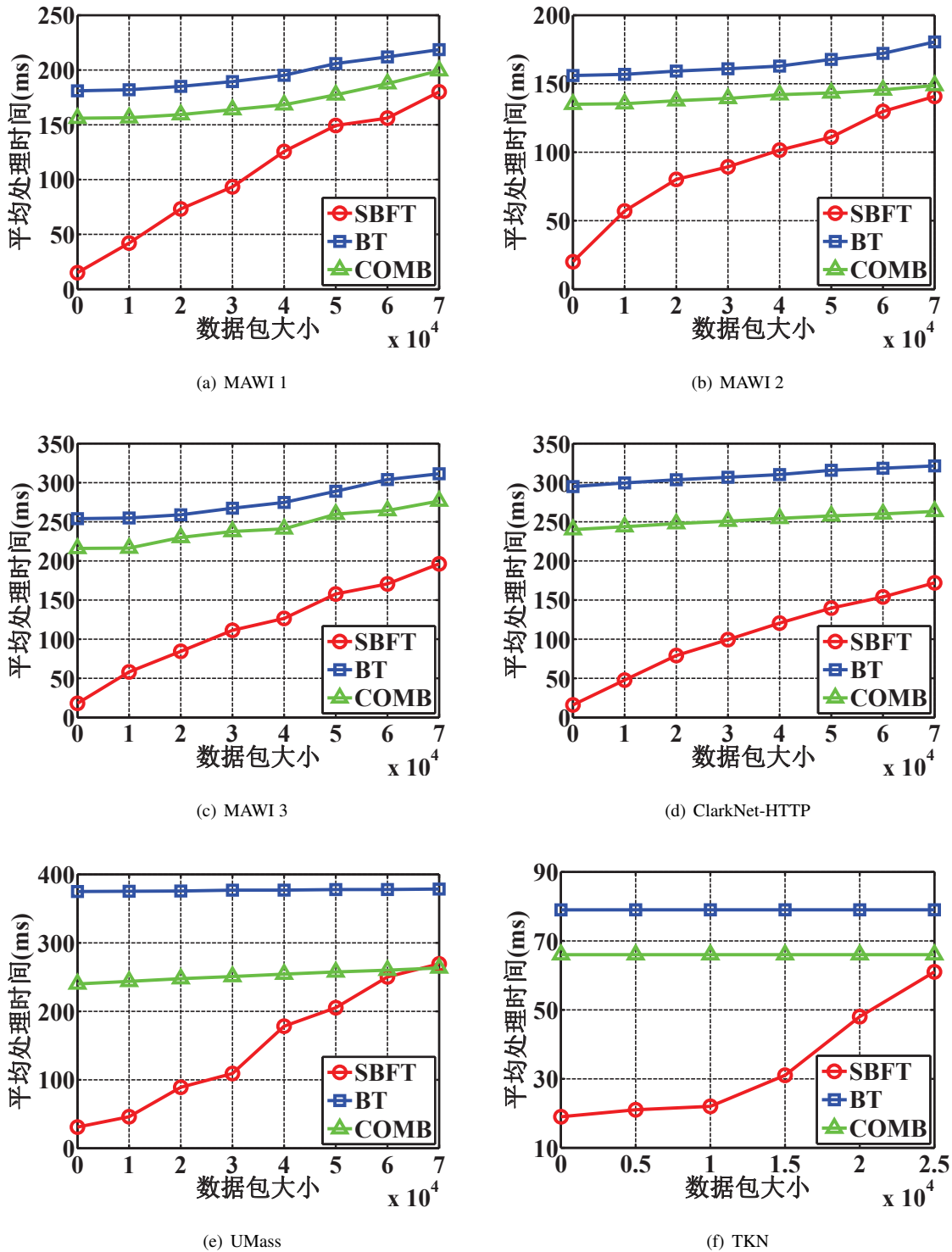


图 3.8 动态数据包平均处理时间

图 3.8 是在数据包不断增加的情况下，三种算法SBFT，BT和COMB处理数据包平均耗时情况的示意图，每组选用 $k = 3$ 个哈希函数进行实验。由于BT和COMB两种结构必须在已知数据量情况下进行操作的，即只能处理静态数据，因此当数据包还未插入时，就需要把结构搭建好，所以这两种结构本文是在

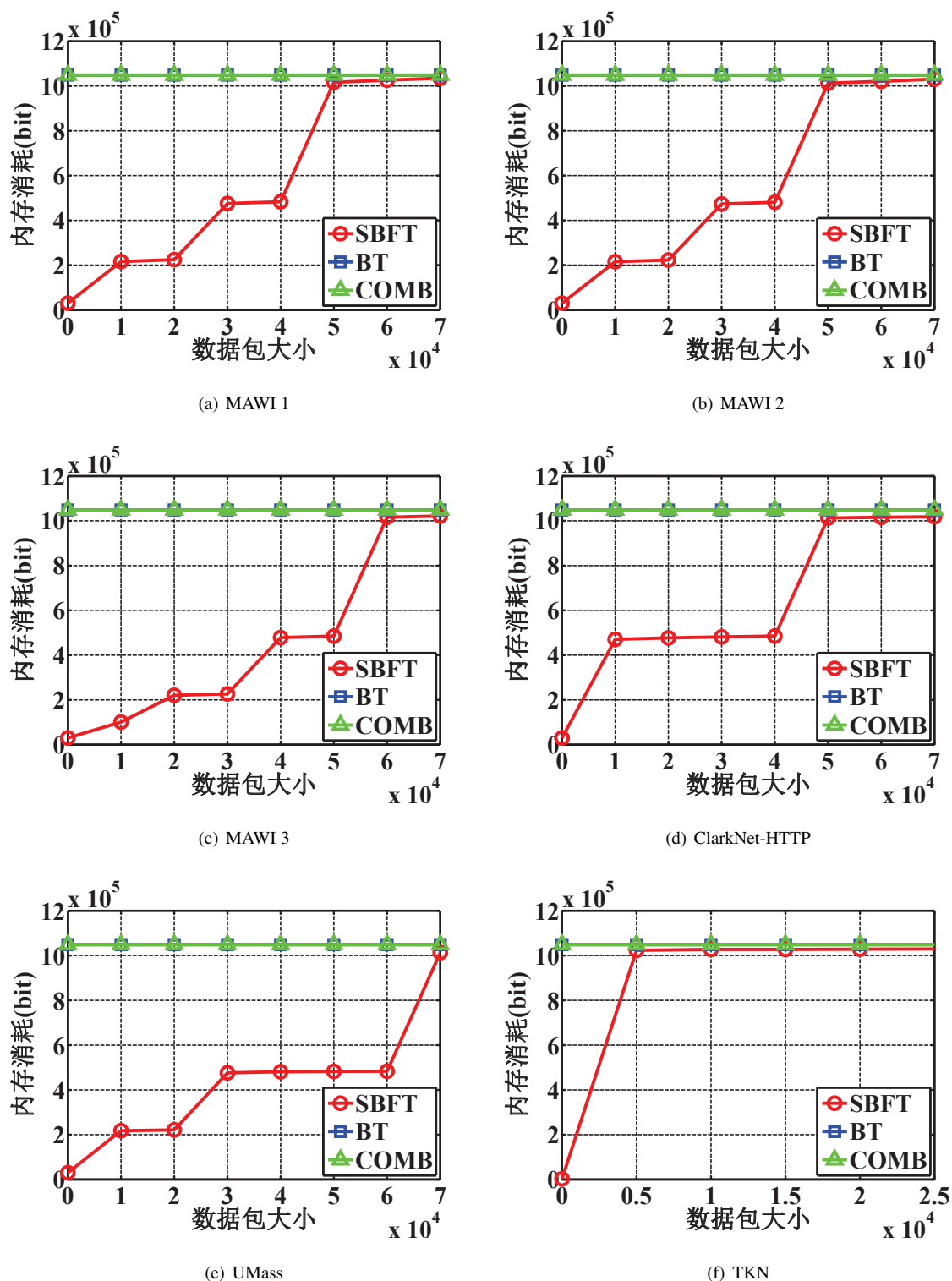


图 3.9 动态数据内存消耗

数据量已知的情况下进行的实验；而SBFT结构则可以在数据包大小未知的情况下进行操作，随着数据包动态添加，SBFT会不断变化结构，内存大小也会随之改变，以此来处理数据，这就是动态处理数据包的过程。图 3.8 (a)–(f)分别是使用数据集MAWI 1、MAWI 2、MAWI 3、ClarkNet-HTTP、UMass、TKN进行仿真实验得到的实验结果示意图。图 3.8 中所有实验结果图显示，针对每种数据集，SBFT处

理数据包是一个动态的过程，这也是因为 H_3 哈希函数的使用减少了操作时间，并且处理数据耗时比BT和COMB都要少，而BT和COMB处理数据包则是一个静态过程，并且耗时较多。通过实验结果图可以发现，本文设计的动态处理键值对的可扩展布鲁姆过滤器树结构不论是处理静态数据还是动态数据，性能更优，更适合在真实的网络环境中使用。

图 3.9 是数据包动态增加时，三种算法SBFT，BT和COMB处理数据包消耗内存情况实验图，每组使用 $k = 3$ 个哈希函数。图 3.9 (a)–(f)分别是使用数据集MAWI 1、MAWI 2、MAWI 3、ClarkNet-HTTP、UMass、TKN进行仿真实验得到的实验结果示意图。图 3.9 中所有实验结果图显示，针对每种数据集，SBFT结构处理数据包是一个动态变化的过程，是边处理数据边搭建结构的一个过程，SBFT结构的内存在不断变化，而BT和COMB处理数据包则是一个静态过程，在处理数据之前就已经把结构搭建好，这也说明它们的结构只能处理静态数据，内存是不会发生变化的。本文设计的SBFT结构在处理动态数据耗时较小的情况下也不会占用较多内存，性能较优，更适合实际的网络环境。本文设计的SBFT结构在动态处理键值对的过程中，每向上扩展一层，SBFT结构内存会在原来的基础上增加1倍，此时如果动态添加的数据较少，造成内存消耗大，也就是扩展的内存得不到充分利用，后续可以对这个问题作进一步改进。

3.8 小结

为了解决布鲁姆过滤器树只能处理静态数据的问题，本章引入了一种 H_3 哈希函数解决了这个问题，设计了一种动态存储键值对的可扩展布鲁姆树结构。SBFT结构不仅处理数据速率提高，而且可以处理动态数据，更加适应了网络环境。本章对SBFT结构基于 H_3 哈希函数进行设计，针对键值对的插入，查询，动态添加新的value操作做了详细的说明，如果将标准布鲁姆过滤器替换为计数式的布鲁姆过滤器，这个结构还可以进行删除操作。本章实验结果是采用真实的数据，各项实验结果显示，SBFT结构在数据处理时间和内存消耗等性能上更优与其它两种结构。

第4章 基于 B_h 序列的键值对布魯姆过滤器

4.1 引言

近年来，键值对存储系统已经成为一个非常重要的组件应用于各种网络应用当中，比如社交网络、线上交易系统以及云计算等等。在线上交易系统中，可以将商品的条形码作为键值key，而对应商品的价格则是value值，通过查询条形码，可以快速得到商品的价格。键值对存储系统往往可以提供一些处理键值对的操作，为了快速进行相应的操作，往往会将它们放置于内存中，这样做查询的精度会非常高。然而，之所以能达到这么高精度往往是以消耗内存和CPU时间为代价的。恰恰相反，本章设计的一种面向键值对存储的布魯姆过滤器算法比现有的大多数结构都要简洁，内存开销低，查询误判率低，这就依赖于BF这种简洁的数据结构。本章会使用一种特殊的编码方式—— B_h 序列^[50]，对键值对进行编码，这种编码的好处在于：可以使用它对布魯姆过滤器进行扩展，使布魯姆过滤器可以进行键值对操作，并且可以避免键值对插入之后发生的一些冲突情况。

4.2 问题来源

为了解决高速网络环境中键值对存储、查询等问题，可以设计一种内存消耗低、查询快速的简洁结构来进行键值对处理操作。在键值对存储过程中，往往会有误判的情况发生：当查询一组未插入的键值对时，查询key得到了一项value。发生这种错误给人的第一感觉就是很反常，但是这种情况出现在现实存在的系统中也是可以接受的：例如，在线上交易系统^[31]中，系统存储商品条形码信息以及它的类别作为一组键值对，当查询这个不存在的商品条形码时，返回了一个类别，出现这种问题其实也无关紧要，因为现实中这个虚构的商品也不会真实出现。另一种错误的情况就是对于一组已经插入的键值对，返回了错误的值信息，这种情况发生率只要能控制在一定范围内，往往也是可以接受的。键值对操作通常包括这几大类：插入、查询、删除和更新。本章的目标就是设计一种简洁的数据结构，使它在允许的误判率范围内完成这几大类键值对操作。

BF是一种空间节俭、查询高效的随机数据结构，它能在误判率一定的范围内完成快速集合元素查询，因此BF经常被使用在各种网络系统应用中，以此在内存和速率上进行平衡。本章就是使用布魯姆过滤器结构进行扩展，来完成键值对的各大操作。

文献 [35] 添加了计数器到布魯姆过滤器中，如果cell中出现碰撞，会返

回“Don't Know”。对应的 k 个位置只要有1个不“Don't Know”，就可以成功查询，但是一个位置仅能插入一个元素，冲突发生概率太高；文献 [37] 对value进行编码，这种方法改变了传统的布鲁姆过滤器，使用cell代替原来的比特位，并且使用异或操作对插入值进行编码，但是当插入的值相同或者查询的位置插入过2个以上元素时，则无法正确对其进行解码，这也是这种结构存在的最大问题。这些技术常常会出现以下冲突：1、多个相同的键值可能会映射到相同的位置上，导致原来的信息失效。2、一个位置插入的元素过多，会导致插入到相同位置的值无法还原。在现如今高速的计算机网络环境中，海量数据往往会导致这两种冲突经常发生，如果将这样的结构应用于实际的网络环境中，常常会导致相关应用效率降低。因此，本文提出了一种基于 B_h 序列的布鲁姆过滤器结构，来进行键值对的操作。而这种设计通常会面临两大问题：1、传统的布鲁姆过滤器只能进行简单的集合查询，不能进行键值对存储；2、传统的布鲁姆过滤器不能进行元素删除操作，虽然计数式布鲁姆过滤器可以进行删除操作，但是对于键值对的删除，还是需要对其结构进行进一步扩展。

4.3 结构设计

本节首先介绍键值对中值value的一种编码方式—— B_h 序列，然后介绍基于 B_h 序列的布鲁姆过滤器设计原则，最后描述如何使用这种布鲁姆过滤器结构进行键值对操作。

4.3.1 B_h 序列

近年来许多工作都用到了 B_h 序列，然而几乎没有在实际的网络应用中使用过。文献 [67] 列举了相关工作。

定义1：假设 A 是一个有限/无限的自然数集合， a_i 是集合 A 中的一个自然数，如果 $a_1 + a_2 + \dots + a_h (a_1 \leq a_2 \leq \dots \leq a_h)$ 得到的数组成的集合中都是各不相同的，那么就认为 A 是一个 B_h 序列。

示例：假设 $A = \{1, 4, 8, 13\}$ 是一个包含4个元素的自然数集合，从这个集合中任意挑选3个数可以得到20种组合： $1 + 1 + 1 = 3, 1 + 1 + 4 = 6, 1 + 1 + 8 = 10, 1 + 1 + 13 = 15, 1 + 4 + 4 = 9, 1 + 4 + 8 = 13, 1 + 4 + 13 = 18, 1 + 8 + 8 = 17, 1 + 8 + 13 = 22, 1 + 13 + 13 = 27, 4 + 4 + 4 = 12, 4 + 4 + 8 = 16, 4 + 4 + 13 = 21, 4 + 8 + 8 = 20, 4 + 8 + 13 = 25, 4 + 13 + 13 = 30, 8 + 8 + 8 = 24, 8 + 8 + 13 = 29, 8 + 13 + 13 = 34, 13 + 13 + 13 = 39$ 。这20个总和都各不相同，因此可以判定集合 A 是一个 B_3 序列，即 $h = 3$ 。又因为 $4 + 4 + 4 + 4 = 1 + 1 + 1 + 13 = 16$ ，因此 A 不是一个 B_4 序列。

定理4.1^[31]：如果 A 是一个 B_h 序列，那么 A 也是一个 $B_{h'}$ 序列，其中 $h' \in [1, h]$ 。

证明：当 $h = h'$ 时，如定义1所述；如果对于 $h' \in [1, h - 1]$ ，在集合 A 中找到

了 h' 个数相加结果相同的两组数，那么这两组数再加上集合 A 中相同的 $h - h'$ 个新元素得到的总和也会是相同的，这就和定义1相矛盾，故证明了定理4.1。

4.3.2 基于 B_h 序列的布鲁姆过滤器设计原则

假设本章需要插入 n 组键值对 $(key, value)$ ，其中 $key, value$ 可以是任意字符串，本章设计的结构需要完成的操作是：

1. 插入操作：将 $(key, value)$ 插入到结构中
2. 更新操作：对 key 对应的 $value$ 进行更新，即将原来的 $value$ 替换成新的 $value$
3. 查询操作：对于给定的 key ，找到与之对应的 $value$
4. 删除操作：删掉一组 $(key, value)$

本文选用单元格 $cell$ 代替标准布鲁姆过滤器的位串向量，其中一个单元格由两部分组成：计数器 $counter$ 和存编码的空间。如图4.1是 B_h -BF的结构。其中，计数器 $counter$ 记录插入单元格的个数，0表示这个单元格未插入元素，1表示这个单元格插入了1个元素，依次类推。存编码的部分初始化为0，存储的是插入到此单元格编码相加得到的和。接下来描述一下这个结构的设计过程：当要插入一组 $(key, value)$ 时，首先会对 $value$ 进行编码，为它分配一个 B_h 序列中的数，这个分配过程是按顺序执行的，即从一个离线获得的 B_h 序列中按顺序分配编码；分配完编码之后，使用 k 个哈希函数计算 key 值，将选取的编码映射到这 k 个位置上，这个过程在下一节会详细介绍；如果需要查询 key 对应的 $value$ ，则是查到相应的编码之后，进行一次反查询，即解码操作，就可以找到编码对应的 $value$ ，这个过程也可以使用一个辅助的 B_h -BF的结构。

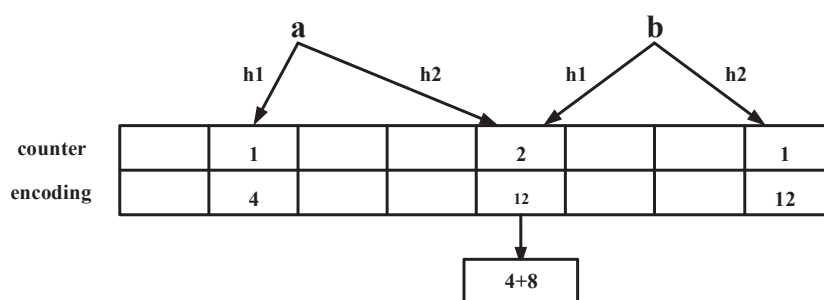


图 4.1 B_h -BF结构

4.3.3 相关介绍

文献 [68] 给出了一种寻找 B_h 序列的方法，通过寻找合适的有限域，找到对应的本原多项式，经过一系列计算，就可以得到 B_h 序列。寻找 B_h 序列的过程是离线操作，不会占用实际网络应用消耗时间。本章就是通过这种方法找到了一组 B_3 序列，本章会使用这组 B_3 序列进行仿真实验，同样也可以用这种方法进行扩展得到 B_h 序列（ h 为任意正整数）。 h 的大小决定了一个单元格 $cell$ 中允许存在的最大编

码个数，以致于不会产生冲突。

接下来介绍value与编码之间的转换：当一个value需要插入到单元格cell时，首先需要确定这个value是否已经分配到一个编码，这时可以使用一个辅助 B_h -BF结构进行查询，这个结构与原始的 B_h -BF结构一模一样，只不过它存储的键值对是(value, encoding)，如果在这个辅助结构中找到了value对应的编码，则直接使用这个编码进行下一步操作，如果没有找到，会从 B_3 序列中顺序分配一个新的编码给value，再将这个新的键值对组合插入到辅助结构 B_h -BF中；相反，如果查询到一个编码，如何转换得到value值呢？实际上，在插入键值对过程中，会将(encoding, value)组保存在一个离线查询表中，直接查询即可得到。本章在(encoding, value)查询过程中，会使用一种巧妙的方法：由于 B_3 序列是由大到小排列的，根据这个规律，可以得到一个二次多项式，后续查找过程中，将编码encoding作为 x 代入二次多项式，得到一个位置 y ，可以直接从离线表这个位置开始查询value值，不用从第一个位置顺序查询，加快了查询速度。

4.4 算法设计

本章为了说明方便，选用了 B_3 序列进行叙述，即对键值对中的值value选用 B_3 序列进行编码。下面就是 B_h -BF结构核心操作的详细介绍：

4.4.1 插入操作

当要插入一组(key, value)到 B_h -BF结构中时，首先选用 k 个不同的哈希函数对key计算，找到 k 个单元格的位置；然后在对应单元格的计数器部分加1，再将value编码值与单元格中的编码值相加，算法4.1描述了这个过程。

4.4.2 查询操作

对于给定的key，需要返回对应的value。计算 k 个不同的哈希函数对key的映射位置，获取相应的单元格信息。根据计数器counter值信息，可以分为以下几种情况进行查询：

1. 如果 k 个位置中有一个计数器counter为0，结束查询，key值不存在于布鲁姆过滤器中；
2. 如果 k 个位置中有一个计数器counter为1，取出该位置中的编码值，在离线表中查询出该编码值对应的value，返回value；
3. 如果 k 个位置中没有计数器counter为0或1，则任取两个位置的编码值进行分解，即根据编码值求出它是由哪些 B_3 序列的数相加得到的，将它们保存在一个集合中，最后将得到的这两个集合取并集，得到要求的编码值，在离线表中查询出该编码值对应的value，返回value。

算法 4.1 $B_h - BF$ 的插入操作**Input:**

键值对(key,value)

Output:更新后的 $B_h - BF$

```

1: for  $0 \leq i \leq k - 1$  do
2:    $j = h_i(key);$ 
3:   if  $B_j.counter == 0$  then
4:      $B_j.counter ++;$ 
5:      $B_j.encoding = value.encoding;$ 
6:   end if
7:   if  $B_j.counter > 0$  then
8:      $B_j.counter ++;$ 
9:      $B_j.encoding = B_j.encoding + value.encoding;$ 
10:  end if
11: end for

```

算法4.2描述了这个过程。由于本文操作都是在基于 B_3 序列上进行的工作，因此，如果单元格中加入的元素个数超过3个，可能会导致还原失败。选择哪种 B_h 序列，是根据可以接受的误判率来决定的，在下文会详细分析。

算法 4.2 $B_h - BF$ 的查询操作**Input:**

键值key

Output:

value值

```

1: for  $0 \leq i \leq k - 1$  do
2:    $j = h_i(key);$ 
3:   if  $B_j.counter == 0$  then
4:     return null
5:   end if
6:   if  $B_j.counter == 1$  then
7:     return  $encodingToValue(B_j.encoding);$ 
8:   end if
9:   if  $B_j.counter > 1$  then
10:     $addList(B_j.encoding);$ 
11:  end if
12: end for

```

4.4.3 删除操作

删除操作就是把需要删除的键值对($key, value$)从 B_h -BF中移除, 和插入操作相反, 对 key 计算 k 个不同的哈希函数的映射位置, 此时, 首先查看 k 个位置的计数器是否都大于1, 如果有1个为0, 那么这个键值对还从未被插入过, 结束操作; 否则, 才将对应位置单元格中的计数器减1, 在将单元格编码值减去 $value$ 的编码值, 完成删除操作, 这个过程可以参考算法4.3。

算法 4.3 $B_h - BF$ 的删除操作

Input:

需要删除的键值对($key, value$)

Output:

更新后的 $B_h - BF$

```

1: for  $0 \leq i \leq k - 1$  do
2:    $j = h_i(key)$ ;
3:   if  $B_j.counter == 0$  then
4:     return
5:   end if
6:   if  $B_j.counter > 0$  then
7:      $B_j.counter - -$ ;
8:      $B_j.encoding = B_j.encoding - value.encoding$ ;
9:   end if
10: end for

```

4.4.4 更新操作

更新操作需要在删除操作的基础上进行, 首先对 key 计算 k 个不同的哈希函数, 得到 k 个映射位置, 注意此时不需要对计数器做任何改变, 只需要将旧的 $value$ 值从原始的单元格中删掉, 再在这个位置插入新 $value$ 对应的新编码值即可。这个过程参考算法4.4。

4.5 性能分析和仿真实验

这一节首先分析 B_h -BF结构一些相关参数的计算, 然后使用真实的网络数据集进行仿真实验, 本文与 B_h -BF结构进行对比的两种结构是容易产生冲突的两种结构: $kBF^{[37]}$ 、 $IBF^{[35]}$, 最后对实验结果进行了分析。

4.5.1 B_h -BF结构相关参数计算

这一节, 理论分析 B_h -BF结构的容量和错误率。进行理论分析的难点在于构造布鲁姆过滤器涉及到许多参数, 其中包括布鲁姆过滤器向量的大小 m , 使用的

算法 4.4 $B_h - BF$ 的更新操作**Input:**

旧的键值对($key, value1$), 新的键值对($key, value2$)

Output:

更新后的 $B_h - BF$

```

1: for  $0 \leq i \leq k - 1$  do
2:    $j = h_i(key)$ ;
3:   if  $B_j.counter == 0$  then
4:     return
5:   end if
6:   if  $B_j.counter > 0$  then
7:      $B_j.encoding = B_j.encoding - value1.encoding + value2.encoding$ ;
8:   end if
9: end for

```

哈希函数的个数 k , 以及插入到其中的总的组数 n 。在第二章已经分析过, 为了将布鲁姆过滤器误判率控制在能接受的范围, 使它的误判率最小, 可以找到一个合适哈希函数个数 $k_{opt} = \frac{m}{n} \ln 2$ 。另一方面, 给定一个合适的误判率值 p , 可以得到布鲁姆过滤器向量的大小:

$$m = -\frac{n \ln p}{(\ln 2)^2} \quad (4.1)$$

从这个式子可以看出, 向量大小 m 与插入组数 n 是呈线性增长关系的, 即给定 m , n 会存在一个上界。

根据上式, 给定一个合适的误判率值 p , 并且在向量大小 m 确定的情况下可以得到 B_h -BF结构的容量:

$$n_{capacity} = -\frac{m(\ln 2)^2}{\ln(p)} \quad (4.2)$$

从上式可以发现, 当 $n_{capacity}$ 达到, 哈希函数个数 k 仅仅与误判率相关: $k = -\frac{\ln p}{\ln 2}$ 。

4.5.2 仿真实验

实验背景: 本章节通过仿真实验验证 B_h -BF的性能, 并将实验结果与两种相似的结构进行了对比, 其中包括: kBF^[37]是2014年INFOCOM中的论文“kBF: a Bloom Filter for Key-Value Storage with an Application on Approximate State Machine”提出的一种键值对布鲁姆过滤器结构, 这种结构使用单元格cell代替布鲁姆过滤器的比特位, 每个单元格插入的value的编码是一个二进制字符串, 插入元素时二进制编码之间是进行异或运算, 这种结构只能还原单元格中不超过2个元素的情

况；IBF^[35]是2011年论文“*Invertible bloom lookup table*”提出的一种利用布鲁姆过滤器进行键值对操作的情况，这篇论文第一次使用单元格cell代替比特位来处理键值对，插入的键值没有经过任何编码，即插入的是什么取出的就是什么，这种结构效率比较低，对于插入同一个单元格的元素无法处理。

实验准备：本文的实验运行在Intel(R) Pentium(R) CPU G630 @ 2.70GHz 以及8.00GB RAM的主机上，采用java编写程序。本文比较的主要是这三种结构的假阳性误判率和假阴性误判率：

1. 假阳性误判率(False Positive Ratio):如果一组键值对未插入到 B_h -BF中，而对于一个key值，查询之后返回了一个value值，这种错误发生的次数占总查询次数的比例。
2. 假阴性误判率(False Negative Ratio):如果对于一组已经插入 B_h -BF结构的键值对，经过查询，返回了一个空值null，或者是一个错误的值，这种错误查询次数占总查询次数的比例。

实验数据：由于都是使用布鲁姆过滤器结构进行键值对处理操作，因此本章选择的实验数据和上一章实验数据一样，在此不再赘述。

实验过程：从数据包中可以提取出很多有用的信息，本文将数据包中的转发ip地址作为key值，而它对应的端口号则是value值。每当遇到一组键值对，首先会使用查询算法查询该键值对是否插入到这个结构当中，如果已经插入，则跳过这个键值对，如果是一组新的值，就使用插入算法对这个键值对进行插入；如果遇到已经插入的key值而value值不同，则使用更新操作。

实验分析：如图4.2，哈希函数个数 k 增加，三种结构的假阳性误判率(False Positive Ratio)会减少，这是因为哈希函数个数越多，可以过滤掉不相关的键值对个数；当 k 值达到一个中心值之后，哈希函数个数 k 增加，三种结构的假阳性误判率会随之增加，这又是因为哈希函数个数增加会减少每一个单元格中可以存储的键值，从而增加其它键值映射到当前键值对应的单元格中的概率。从实验结果图4.2可以发现， B_h -BF结构的假阳性误判率始终比kBF和IBF这两种结构的要低，这也是因为 B_h -BF结构解决了其它结构会发生的冲突问题，通过这个实验也可以发现，选择合适的哈希函数个数 k 可以将假阳性误判率最小化。根据图4.2可以看出，数据集MAWI 1、MAWI 2、UMass选取的最优哈希函数个数 $k = 3$ ，数据集MAWI 3、ClarkNet-HTTP选取的最优哈希函数个数 $k = 4$ ，数据集TKN选取的最优哈希函数个数 $k = 5$ 。在实际操作中，根据可以容忍的误判率，选取合适的哈希函数个数 k 。

三种结构的假阴性误判率(False Negative Ratio)与假阳性误判率呈现规律一致，如图4.3， B_h -BF假阴性误判率之所以比kBF和IBF这两种结构低，是因为 B_h -BF结构可以还原单元格中插入超过2个元素的情况，而IBF只能处理单元格中存在一个元

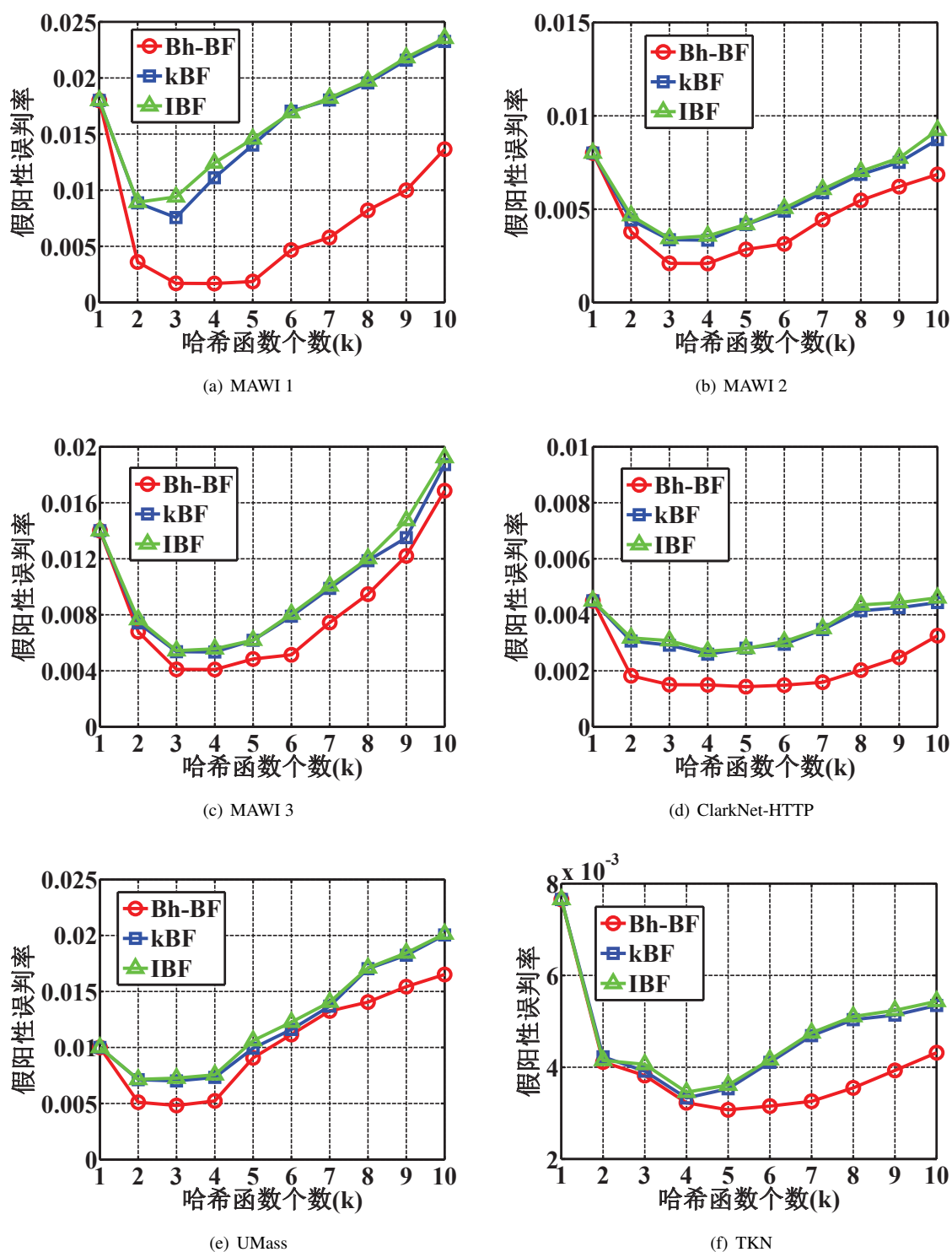


图 4.2 假阳性误判率

素的情况，kBF只能处理单元格中存在两个元素以下的情况。虽然 B_h -BF结构还原效率高，但是付出的代价即还原需要的时间会比kBF和IBF这两种结构需要的多。

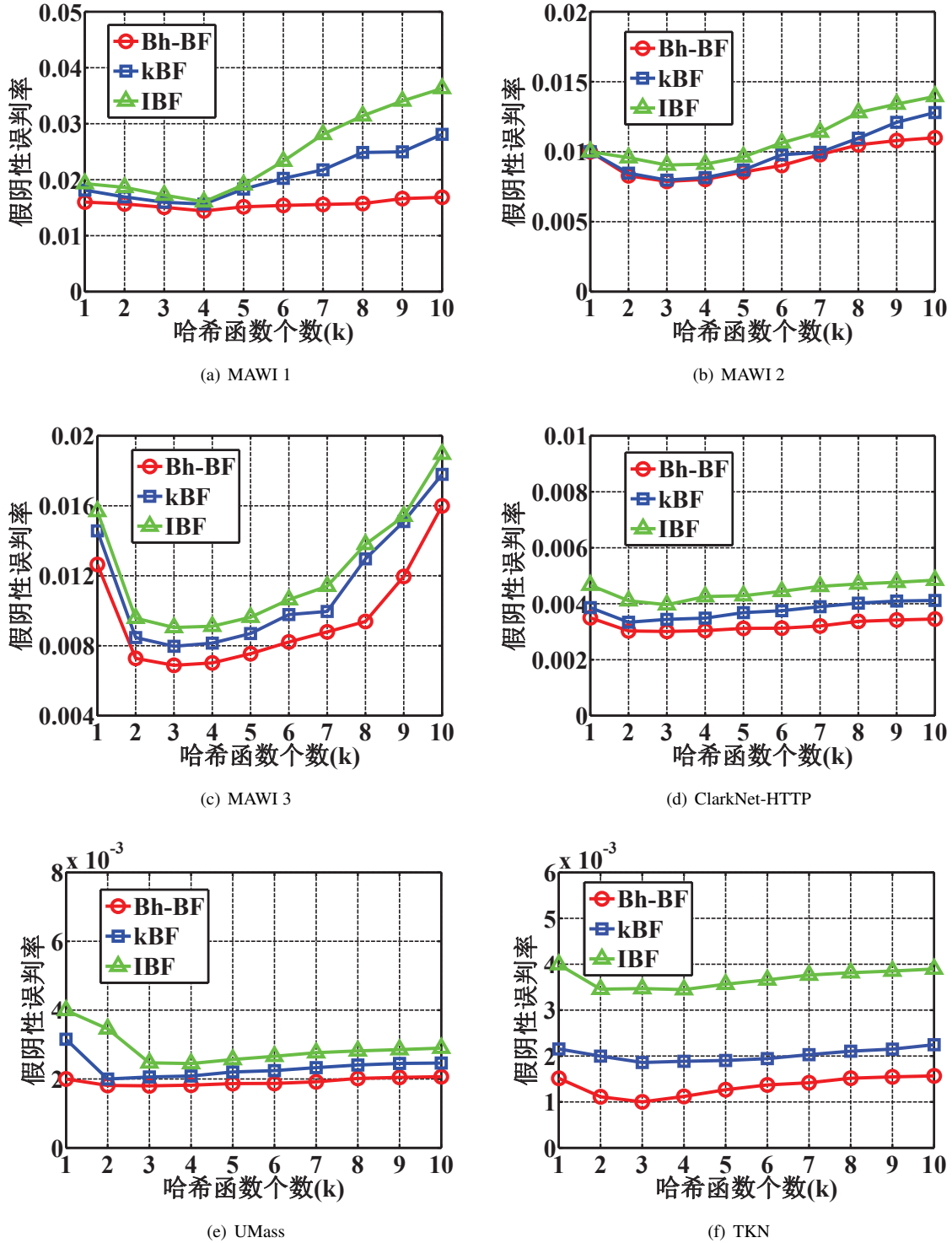


图 4.3 假阴性误判率

4.6 小结

本文针对一些面向键值对存储的布鲁姆过滤器查询算法存在冲突的情况寻找解决方案，找到了一种很特殊的编码方案— B_h 序列，这种序列有一种特殊的性质：序列中任意 h 个数相加结果各不相同。根据这种性质，本文解决了kBF，IBF两种结构产生冲突的情况。本章设计了一种基于 B_h 序列的布鲁姆过滤器查询算法用来

存储键值对，并且设计了这种结构基于 B_h 序列对键值对进行插入、查询、更新和删除操作的算法。最后使用真实有效的网络流量数据进行仿真实验，与kBF, IBF这两种结构相比较，实验结果显示 B_h -BF具有更低的假阳性和假阴性误判率。

结 论

布鲁姆过滤器是一种可以简洁地表示数据集合，并且空间效率和时间效率都极高的概率型数据结构。本文对布鲁姆过滤器查询算法的应用、基本操作、研究现状等作了简单的介绍，从中发现布鲁姆过滤器查询算法本身具有很强的扩展性，而且现今该算法处于迅猛发展的时期，对它的研究以及应用也正处于火热的状态。

键值对(key, value) 存储是计算机网络系统中最常见的任务之一，因其存储模式简单、易于扩展的特性在近几年受到广泛的关注和应用。本文对键值对存储的概念和基本操作作了简单概述，从中发现传统的键值对存储往往依赖于数据库进行操作，这也就限制了键值对必须存储于磁盘中。现今设计高效的键值对存储结构算法已经引起国内外研究学者的广泛关注。

本文主要是面向键值对存储的布鲁姆过滤器查询算法设计，利用布鲁姆过滤器结构进行扩展，在这些方面展开了深入研究，设计了两种相关的处理键值对数据的算法。本文的主要贡献主要就是以下几个方面：

1. 本文针对现有的一种处理键值对的布鲁姆树结构处理速率慢、只能处理静态数据这两个缺陷，设计了一种可扩展的布鲁姆过滤器树(SBFT)结构。这种结构操作简单，误判率低，比较适合在各种应用中使用。
2. 本文在SBFT的每个节点上引入了一种 H_3 哈希函数，通过移位操作与 H_3 哈希函数搭配使用，解决了原布鲁姆树存在的问题。并且利用 H_3 哈希函数的相关性质，可以对SBFT结构进行扩展，使它可以进行动态数据处理，更加适应现今的网络环境。
3. 本文设计了SBFT结构对于键值对数据的插入，查询，动态添加新的value操作的算法，如果将标准的布鲁姆过滤器替换为计数式布鲁姆过滤器，那么这个结构还可以进一步进行删除操作。仿真实验结果显示，三者消耗内存一定的基础上，SBFT结构在数据处理时间和处理动态数据的性能上更优与BT和COMB两种结构。
4. 本文针对现有的一些面向键值对存储的布鲁姆过滤器查询算法存在冲突的问题，设计了一种基于 B_h 序列的布鲁姆过滤器(B_h -BF)结构，用来处理键值对操作。
5. 本文找到了一种很特殊的编码方案—— B_h 序列，这种序列有一种特殊的性质：序列中任意 h 个数相加结果各不相同。本文编码实现了这种 B_h 序列，并且离线保存了多个 B_3 序列数，为后续实验做准备。

6. 本文解决了kBF, IBF两种结构产生冲突的情况, 并且设计了 B_h -BF结构对键值对进行插入、查询、更新和删除操作的算法。最后使用真实有效的网络流量数据进行仿真实验, 与kBF, IBF这两种结构相比较, 实验结果显示 B_h -BF具有更低的假阳性误判率和假阴性误判率。

由于时间等一些条件的限制, 本文对面向键值对存储的布魯姆过滤器查询算法的研究还可以继续深入, 总结了多种面向键值对存储的布魯姆过滤器查询算法之后, 接下来可以进行深一步的研究:

1. 对于提出的动态可扩展的布魯姆过滤器树结构, 可以进一步研究它的精简模式, 因为在数据不断增加的情况下, 布魯姆过滤器树结构占用的内存会越来越大, 内存消耗开销大, 性能会降低, 研究精简模式的布魯姆过滤器树, 使它更适用于实际的网络环境。
2. 对于SBFT结构在动态处理键值对的过程中, 每向上扩展一层, SBFT结构内存会增加1倍, 此时如果动态添加的数据较少, 造成内存消耗大, 也就是扩展的内存得不到充分利用, 后续可以对这个问题作进一步改进。
3. 对于提出的基于 B_h 序列的布魯姆过滤器查询算法, 探索一种更合适的编码使得冲突的情况不会发生。针对这种结构, 可以继续探索一种自适应的布魯姆过滤器, 当元素集合不断增长的情况下, 可以自适应变更布魯姆过滤器向量长度, 添加更多的键值对; 如果可以设计一种动态释放并且分配内存的结构, 必然可以减小 B_h 序列布魯姆过滤器的内存消耗, 通过这种方法, 算法内存可以优化, 进一步节省了 B_h -BF的内存开销, 更适用于实际的网络环境。
4. 本文下一步可以继续深入布魯姆过滤器查询算法的研究, 设计更对布魯姆过滤器查询算法处理键值对操作的结构。结合现有的一些布魯姆过滤器研究成果, 将更多的理论研究应用于实践中, 期望各个领域进行结合得到最新的研究成果。

参考文献

- [1] 郑增威, 吴朝晖. 普适计算综述. 计算机科学, 2003, (4):18–22,29
- [2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 1970, 13(7):422–426
- [3] Michael Mitzenmacher. Compressed bloom filters. IEEE/ACM Trans. Netw., 2002, 10(5):604–612
- [4] James K. Mullin. Optimal semijoins for distributed database systems. Software Engineering, IEEE Transactions on, 1990, 16(5):558–560
- [5] M. McIlroy. Development of a Spelling List. Communications, IEEE Transactions on, 1982, 30(1):91–99
- [6] James K. Mullin. Estimating the size of a relational join. Information Systems, 1993, 18(3):189 – 196
- [7] Udi Manber, Sun Wu. An Algorithm for Approximate Membership Checking with Application to Password Security. Inf. Process. Lett., 1994, 50(4):191–197
- [8] Lee L. Gremillion. Designing a Bloom filter for differential file access. Commun. ACM, 1982, 25(9):600–604
- [9] James K. Mullin. A second look at bloom filters. Commun. ACM, 1983, 26(8):570–571
- [10] 李龙飞, 贺占庄, 史阳春. 基于布鲁姆过滤器的面向IP包识别的CPBF算法. 华南理工大学学报(自然科学版), 2017, (7):90–97,106
- [11] 李玮, 张大方, 徐冰. 面向NDN中名字查找的哈希布鲁姆过滤器. 电子科技大学学报, 2017, (5):734–740
- [12] 胡会南, 陈华辉. 大数据下近似成员关系查询方法研究进展. 数据通信, 2017, (2):27–34
- [13] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, et al. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. ACM SIGCOMM Computer Communication Review, 2006, 36(4):315–326
- [14] A Broder, M Mitzenmacher. Network Applications of Bloom Filters: A Survey. Internet Mathematics, 2004, 1(4):485–509
- [15] S.C. Rhea, J. Kubiawicz. Probabilistic location and routing. In: Proc of INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, volume 3. 2002, 1248–1257 vol.3

- [16] Jonathan Ledlie, Jacob M. Taylor, Laura Serban, et al. Self-organization in Peer-to-peer Systems. In: Proc of Proceedings of the 10th Workshop on ACM SIGOPS European Workshop. New York, NY, USA: ACM, 2002. 125–132
- [17] Francisco Matias Cuenca-Acuna, Christopher Peery, Richard P. Martin, et al. Planet-P: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In: Proc of Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing. Washington, DC, USA: IEEE Computer Society, 2003. 236–246
- [18] Peter Druschel, Antony Rowstron. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In: Proc of Proceedings of the Eighth Workshop on Hot Topics in Operating Systems. Washington, DC, USA: IEEE Computer Society, 2001. 65–70
- [19] Ion Stoica, Robert Morris, David Karger, et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. SIGCOMM Comput. Commun. Rev., 2001, 31(4):149–160
- [20] Sylvia Ratnasamy, Paul Francis, Mark Handley, et al. A Scalable Content-addressable Network. SIGCOMM Comput. Commun. Rev., 2001, 31(4):161–172
- [21] 范俊梅, 王斌, 王国仁, et al. 分布式环境下改进的Bloom Filter过滤技术. 华中科技大学学报 (自然科学版), 2005, (z1):205–208
- [22] 谢鲲, 张大方, 谢高岗, et al. 基于轨迹标签的无结构P2P副本一致性维护算法. 软件学报, 2007, (1):105–116
- [23] Cristian Estan, George Varghese. New Directions in Traffic Measurement and Accounting. SIGCOMM Comput. Commun. Rev., 2002, 32(4):323–336
- [24] Sarang Dharmapurikar, Praveen Krishnamurthy, D.E. Taylor. Longest prefix matching using bloom filters. Networking, IEEE/ACM Transactions on, 2006, 14(2):397–409
- [25] Alex C. Snoeren, Craig Partridge, Luis A. Sanchez, et al. Hash-based IP Traceback. SIGCOMM Comput. Commun. Rev., 2001, 31(4):3–14
- [26] 金澈清, 钱卫宁, 周傲英. 流数据分析与管理综述. 软件学报, 2004, (8):1172–1181
- [27] Fan Deng, Davood Rafiei. Approximately Detecting Duplicates for Streaming Data Using Stable Bloom Filters. In: Proc of Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: ACM, 2006. 25–36
- [28] M. E. Locasto, J. J. Parekh, A. D. Keromytis, et al. Towards collaborative security and P2P intrusion detection. In: Proc of Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop. 2005, 333–339

- [29] 陈伟, 何炎祥, 彭文灵. 一种轻量级的拒绝服务攻击检测方法. 计算机学报, 2006, (8):1392–1400
- [30] A. Whitaker, D. Wetherall. Forwarding without loops in Icarus. In: Proc of Open Architectures and Network Programming Proceedings, 2002 IEEE. 2002, 63–75
- [31] Fang Hao, Murali Kodialam, T. V. Lakshman, et al. Fast Dynamic Multiple-set Membership Testing Using Combinatorial Bloom Filters. IEEE/ACM Trans. Netw., 2012, 20(1):295–304
- [32] F. Chang, W. Feng, K. Li. Approximate Caches for Packet Classification. Proc. of IEEE Infocom' 04, 2004.
- [33] J Almeida AZ Broder. L Fan, P Cao. Summary cache: a scalable wide-area Web cache sharing protocol. Networking, IEEE/ACM Transactions on, 2000, 8(3):281–293
- [34] Saar Cohen, Yossi Matias. Spectral bloom filters. In: Proc of Proceedings of the 2003 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, 2003. 241–252
- [35] M. T. Goodrich, M. Mitzenmacher. Invertible bloom lookup tables. In: Proc of 2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton). 2011, 792–799
- [36] Deke Guo, Jie Wu, Honghui Chen, et al. Theory and Network Applications of Dynamic Bloom Filters. In: Proc of INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings. 2006, 1–12
- [37] S. Xiong, Y. Yao, Q. Cao, et al. kBF: A Bloom Filter for key-value storage with an application on approximate state machines. In: Proc of IEEE INFOCOM 2014 - IEEE Conference on Computer Communications. 2014, 1150–1158
- [38] Ori Rottenstreich, Yossi Kanizo, Isaac Keslassy. The Variable-increment Counting Bloom Filter. IEEE/ACM Trans. Netw., 2014, 22(4):1092–1105
- [39] 谢鲲, 文吉刚, 张大方, et al. 布鲁姆过滤器查询算法. 软件学报, 2009, (1):96–108
- [40] Myung Keun Yoon, JinWoo Son, Seon-Ho Shin. Bloom tree: A search tree based on Bloom filters for multiple-set membership testing. In Proceedings of the IEEE Conference on Computer Communications (INFOCOM), 2014. 1429–1437
- [41] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, et al. Fast hash table lookup using extended bloom filter: an aid to network processing. In: Proc of Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications. New York, NY, USA: ACM, 2005. 181–192

- [42] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, et al. Deep Packet Inspection Using Parallel Bloom Filters. *IEEE Micro*, 2004, 24(1):52–61
- [43] Yan Qiao, Tao Li, Shigang Chen. One Memory Access Bloom Filters and Their Generalization. *IEEE INFOCOM 2011*, 2011. 1745–1754
- [44] H. Song, F. Hao, M. Kodialam, et al. IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards. In: *Proc of IEEE INFOCOM 2009*. 2009, 2518–2526
- [45] David Eppstein, Michael T. Goodrich, Frank Uyeda, et al. What’s the Difference?: Efficient Set Reconciliation Without Prior Context. *SIGCOMM Comput. Commun. Rev.*, 2011, 41(4):218–229
- [46] Haipeng Dai, Yuankun Zhong, Alex X. Liu, et al. Noisy Bloom Filters for Multi-Set Membership Testing. *SIGMETRICS Perform. Eval. Rev.*, 2016, 44(1):139–151
- [47] 李玮, 张大方, 谢鲲, et al. 一种面向闪存键值存储的矩阵索引布鲁姆过滤器. *计算机研究与发展*, 2015, (5):1210–1222
- [48] 施文. 安全的布鲁姆过滤器和基于键值对的布鲁姆过滤器. 湖南大学, 2016.
- [49] J. Lawrence Carter, Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 1979, 18(2):143 – 154
- [50] S. W. Graham. Bh sequences. *Birkhäuser Boston*, Boston, MA, 1996: 431–449
- [51] 谢鲲. 布鲁姆过滤器查询算法及其应用研究. 湖南大学, 2007.
- [52] S. Tarkoma, C. E. Rothenberg, E. Lagerspetz. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys Tutorials*, 2012, 14(1):131–155
- [53] Rüdiger Schollmeier, Wolfgang Kellerer. Suitability of Bloom Filters for Network Applications. In: *Proc of Proceedings of 2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2004)*. 2004
- [54] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, et al. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM Journal on Computing*, 1994, 23(4):738–761
- [55] M. Ramakrishna, E. Fu, E. Bahcekapili. A Performance Study of Hashing Functions for Hardware Applications. *Proc. 6th Int. Conf. Comput. Inf.*, 1994.
- [56] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, et al. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In: *Proc of Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004. 30–39

- [57] J. Lawrence Carter, Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 1979, 18(2):143 – 154
- [58] M. V. Ramakrishna. Practical Performance of Bloom Filters and Parallel Free-text Searching. *Commun. ACM*, 1989, 32(10):1237–1239
- [59] I. Kaya, T. Kocak. A low power lookup technique for multi-hashing network applications. In: *Proc of IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*. 2006, 179–184
- [60] M. V. Ramakrishna, E. Fu, E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers*, 1997, 46(12):1378–1381
- [61] M. V. Ramakrishna, G. A. Portice. Perfect Hashing Functions for Hardware Applications. In: *Proc of Proceedings of the Seventh International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1991. 464–470
- [62] 耿国华. 数据结构——用C语言描述. 高等教育出版社, 年份2011
- [63] Mawi working group traffic archive. <http://mawi.nyu.edu/mawi/>.
- [64] Calgary-HTTP. <http://ita.ee.lbl.gov/html/contrib/Calgary-HTTP.html>.
- [65] UMass Trace Repository. <http://traces.cs.umass.edu/index.php/Network/Network>.
- [66] TKN. <http://www-tnk.ee.tu-berlin.de/research/trace/ltvt.html>.
- [67] Xingde Jia. Bh[g]-Sequences with Large Upper Density. *Journal of Number Theory*, 1996, 56(2):298 – 308
- [68] R. C. Bose, S. Chowla. Theorems in the additive theory of numbers. *Commentarii Mathematici Helvetici*, 1962, 37(1):141–147

致 谢

转眼间三年研究生生活即将画上句号，内心既兴奋又激动，但更多的还是不舍。即将步入社会，独立承担社会责任，研究生生活让我学会了很多。高考时就梦想着来湖南大学就读，带着这份执着如愿考上了湖大的研究生，特别高兴。在湖大求学的三年里，收获了太多太多，不仅仅是在学习上，还有生活中。三年的研究生生涯遇到了很多老师、同学、朋友，在与他们的交往中，我学到了很多东西，收获了不少情谊，在此，我想向曾经关心和帮助过我的老师、同学、朋友致以最诚挚的谢意。

首先，我要衷心的感谢我的导师谢鲲教授。感谢谢老师这三年来对我的悉心指导和帮助，在我学术研究中遇到困难时会不断鼓励我，给我信心，让我能坚持下去。谢老师在学术上的精深造诣以及严谨态度是我在科研道路上最重要的航向标杆。谢老师以身作则教导我们培养良好的生活习惯和做事风格，这不仅仅使我们在学术上严格要求自己，更使得我们在自己人生道路上也要严于律己，这一点会让我终生受益。研究生期间我的每一次进步都离不开谢老师的密切指导，在此我向我的导师表示深切的谢意与祝福！祝谢老师及家人身体健康，心想事成！

感谢湖南大学提供给我们这么好的学术氛围和环境，每一次参加学术报告都能让我学到很多东西，感谢您为我的成长提供一方乐土。感谢湖大的授课老师们给我们上的精彩课程，让我能学到更多的专业知识。

这三年交流最多的就是实验室的伙伴们，非常开心与你们在一起的这些日子。感谢我的同门陶恒对我项目实践和编码过程中专业知识的指导，这些指导对于我来说犹如雪中送炭。感谢我的同门彭灿、赵彦彦，与你们在一起的日子很开心，学到很多东西。感谢师兄师姐师弟师妹们，每一次学术讨论都能让我学到新知识，拓展视野。感谢在湖大认识的朋友们，谢谢你们对我的关心和帮助，让我的研究生生活更加精彩。

深深感谢我的父母和家人，谢谢你们这么多年对我的支持与鼓励，是你们让我更加乐观地面对生活中的一切，你们对我如此无私的付出，我会用一生去回报。感谢我的男朋友，异地真心不容易，谢谢你总是鼓励我，让我积极面对一切，努力学习，对生活充满信心，谢谢你，出现在我的人生中。

最后，我要向所有参与审阅、评议本论文以及参与本人论文答辩的各位老师表示由衷的感谢！人生道路的每一时刻都要好好珍惜，抱着感恩的心态度过这段美好岁月，因为有你们的关心和帮助，我很幸福。将来我一定会更加勤奋学习、更积极的生活，我想这也是能给你们最好的回报！

附录A 发表论文和参加科研情况说明

（一）申请及已获得的专利

- [1] 潘海娜, 谢鲲, 凌纯清. 动态存储键值对的布鲁姆过滤器树结构及键值对存储方法, 发明专利, 实审阶段, 申请号: 201710542207.5, 申请日期2017.08.02

附录 B 攻读学位期间所参与的科研项目

- [1] 基于矩阵填充的软件定义网络流量矩阵测量方法研究，国家自然科学基金(61572184)，2016.1 -2019.12，参与