華中科技大學

物联网数据存储与管理

课 程 报 告

选题： 基于 **Bloom Filter** 的
多维数据属性表示和索引

专业班级：**IOT1601**
学　　号：**U201614897**
姓　　名：潘　越
指导教师：华　宇
报告日期：**2019** 年 **6** 月 **3** 日

计算机科学与技术学院

目 录

I 摘要

在存储系统中，元数据是用于描述文件系统组织结构的数据，在访问文件数据之前必须先访问其元数据以获取要访问数据的描述信息和空间组织信息，然后才能根据这些信息访问到数据。现有元数据管理方式中，使用的非常广泛的一种就是结合布隆过滤器（Bloom-filter）结构的高速查询，本文基于布隆过滤器（Bloom-filter）结构进行了相关整理研究，并结合目前市面上已经存在的多种布隆过滤器（Bloom-filter）结构进行了改进，提出了一种多路并发哈希计数器的改进结构来解决多维数据属性表示和索引的问题，并辅以一些代码实验和相关的性能研究，以验证改进结构的合理性和高效性。

关键词：元数据，布隆过滤器结构，并发哈希

II 前言

II.1 课题的来源、目的、意义

Bloom-Filter，即布隆过滤器，1970 年由 Bloom 中提出。它可以用于检索一个元素是否在一个集合中，在存储中则是一种高效快速的元数据管理方案。

Bloom Filter 是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。它是一个判断元素是否存在集合的快速的概率算法。Bloom Filter 有可能会出现错误判断，但不会漏掉判断。也就是 Bloom Filter 判断元素不在集合，那肯定不在。如果判断元素存在集合中，有一定的概率判断错误。因此，Bloom Filter 不适合那些“零错误的应用场合”。而在能容忍低错误率的应用场合下，Bloom Filter 比其他常见的算法（如 hash 折半查找）极大节省了空间。

它的优点是空间效率和查询时间都远远超过一般的算法，通过极少的错误换取了存储空间的极大节省，而缺点是有一定的误识别率和删除困难。

通过本课题的研究，可以优化 Bloom Filter 的性能，以及在其缺点上寻求解决方案并给出一个具体的实验范例。

II.2 主要解决的问题

本文给出如图 1 所示的 Bloom Filter 结构的改进，来优化多维数据属性表示和索引的问题。

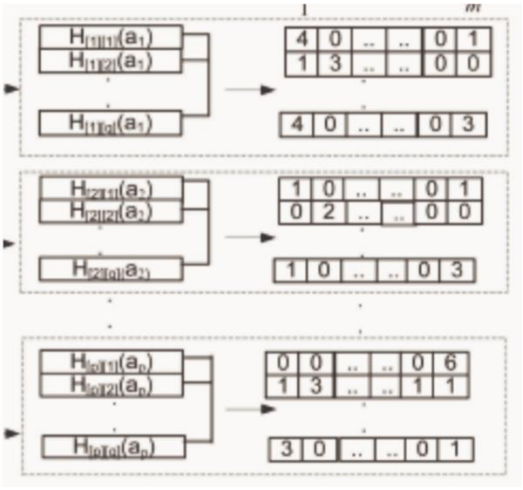

图 1 多维 Bloom Filter 结构

本文将从以下几个部分进行组织叙述：

（1）Bloom Filter 的基本思想和性能分析

（2）改进的数据结构与流程设计

（3）理论分析与代码验证实验

II.3 符号一览

下文中用到的几个符号说明如下：

表 1 符号说明

序号	符号	说明
1	m	Bloom Filter 位数组的宽度
2	n	集合元素的个数
3	t	测试集元素的个数
4	k	使用 Hash 函数的个数
5	p	Bloom Filter 位数组中某一位为 0 的概率
6	f	False Positive 的比率
7	g	改进后的 False Positive 的比率

III 核心设计

III.1 Bloom Filter 的基本思想

首先我们简单介绍一下 Bloom Filter 的基本思想和其性能。Bloom-Filter 算法的核心思想就是利用多个不同的 Hash 函数来解决“冲突”。

III.1.1 Bloom Filter 的流程

为了检验一个元素是否在集合中，最简单的方法就是使用 Hash 函数把每一个元素映射到二进制数组中的某一位，这样既节省空间又运行高效。但 Hash 函数存在的问题就是“冲突”，两个不同的元素经过同一个 Hash 函数运行所得到的结果可能相同。因此，为了减少冲突，Bloom Filter 引入了多个 Hash 函数来解决这一问题，每次加入元素时，通过多个 Hash 函数把相应的位置 1。在判断时，只要有一个 Hash 函数判断该元素不在集合中，那么该元素肯定不在集合中，只有在所有的 Hash 函数告诉我们该元素在集合中时，才能确定该元素存在于集合中，这便是 Bloom-Filter 的基本思想。

如图 2 所示，假设 Bloom Filter 使用一个 m 比特的数组来保存信息，初始状态时，Bloom Filter 是一个包含 m 位的位数组，每一位都置为 0，即 BF 整个数组的元素都设置为 0。

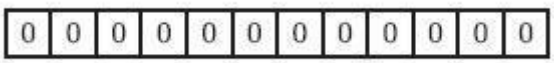

图 2 二进制数组

为了表达$S=\{x_1,x_2 \dots x_n\}$这样一个 n 元集合，Bloom Filter 使用 k 个独立的 Hash 函数，它们分别将集合中的每个元素映射到$\{1,2 \dots m\}$的范围中，如图 3 所

示。当加入一个元素时，即将 k 个 Hash 函数运算的结果位置置为 1，若运算的结果位置已经为 1，则不作任何操作。

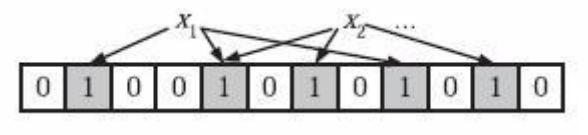

图 3 Bloom Filter 添加元素

当判断元素是否在集合中时，我们只需要对 k 个 Hash 函数得到 k 个 Hash 值，如果每个 Hash 值对应的二进制数组中的位置为 1，则认定该元素在集合中，否则就认为该元素不是集合 S 中的元素。

很显然，这个判断并不能保证结果 100%正确，例如图 3 中的情况，假设有第三个元素，其 3 个 Hash 结果分别为 4、8、10，则对应的结果也为 1，将会被判断在集合 S 中，产生了错误的判断。我们将这种情况称之为 False Positive。

III.1.2 Bloom Filter 的性能分析

接下来我们分析 Bloom Filter 的性能问题，也即 False Positive 的比率 f 的问题，探讨何时 f 才能最小。

初始状态时，如图 2，二进制数组的 m 位均为 0，此时进行一次 Hash，则某一位为 0 的概率是$\frac{m-1}{m}$（只有 1 位为 1，且假设 Hash 函数计算结果在每一位的概率均等），因此，对 n 个元素进行 k 次 Hash，则某一位为 0 的概率 p 为：

$$\mathrm{p}=\left(1-\frac{1}{m}\right)^{nk}$$

对上式做变换，并利用利用自然指数极限的代换，有：

$$\mathrm{p}=\left(1-\frac{1}{m}\right)^{nk}=\left(1-\frac{1}{m}\right)^{m\cdot\frac{nk}{m}}\approx e^{-\frac{nk}{m}}$$

一个 False Positive 发生，即一个不在集合的数却被判定在集合中的概率，是在集合中任选 k 个数，其结果均为 1 的概率，该概率即为 False Positive 的比率 f，计算如下：

$$\begin{aligned}\mathrm{f}&=(1-p)^k\\&\approx\left(1-e^{-\frac{nk}{m}}\right)^k\\&=e^{\mathrm{k}\ln\left(1-e^{-\frac{nk}{m}}\right)}\\&=e^{-\mathrm{k}\cdot\frac{m}{nk}\ln e^{-\frac{nk}{m}}\ln\left(1-e^{-\frac{nk}{m}}\right)}\\&=e^{-\frac{m}{n}\ln(p)\ln(1-\mathrm{p})}\end{aligned}$$

上式中，e^x为单调递增函数，因此当其指数最小时，取最小值，也即

$\ln(p)\ln(1-\mathrm{p})$取最大值，此时有$\mathrm{p}=\frac{1}{2}$，即

$$e^{-\frac{nk}{m}}=\frac{1}{2}$$

$$\mathrm{k}=\frac{m}{n}\ln 2\approx 0.7\frac{m}{n}$$

此时给出的 f 为：

$$\mathrm{f}=\left(\frac{1}{2}\right)^{k}\approx 0.6185^{\frac{m}{n}}$$

III.1.3 Bloom Filter 的缺点

通过上面的分析，我们可以看到 Bloom Filter 高效的查找与优越的性能，但是也能看到几个明显的问题。

（1）无法删除集合中的元素

Bloom Filter 在插入元素时，对于 Hash 计算结果处已经被标记为 1 的位是不做操作的，所以在删除时就会出现问题，如果直接将该元素经过 Hash 计算后的位置置为 0，则会牵动到其他元素。

（2）Hash 函数的选择会影响到算法的结果

根据前面的错误率计算，当哈希函数个数$\mathrm{k}=\ln 2\times\frac{m}{n}$时错误率最小，在实际的应用中，只有 n 是固定的，我们要综合考虑 m 和 k 的选择问题以及哈希函数的设计问题。

III.2 数据结构设计

III.2.1 改进前的结构阐述

如图 1，题目中给出的多维 Bloom Filter 的结构是在 Bloom Filter 的基础上改进的 Counter Bloom Filter。这种结构中，不再使用二进制数组来管理数据，而是为每一个位置维护了一个 Counter，每次 Hash 函数计算之后，将结果的位置计数加一。这样的改进可以支持元素的删除，同时又具有 Bloom Filter 查询的高效性能，唯一的缺陷是增加了存储空间，当然这也是必须的，因为二进制数组只能存储一位的数据。

其性能上并没有变化，仍然是在$e^{-\frac{nk}{m}}=\frac{1}{2}$时可以取到 f 的最小值。

III.2.2 改进后的结构

考虑到在 Bloom Filter 运行的过程中，有大量的 Hash 函数运行，因此考虑从程序的并发性上来优化。由于该结构引入了计数器，那么如果要直接并发 Hash 函数，那么对于计数器的值就必须引入锁机制来防止冲突，这样反倒并不能给出太大的优化，因此考虑对 Hash 函数进行分区。

假设有 k 个 Hash 函数，那么使得 k 个 Hash 函数的运算范围在$0\sim\frac{m}{k}$之间，

各自负责某一区域，此时即可使用多线程并发完成多个 Hash 函数的运算。其结构如图 4 所示：

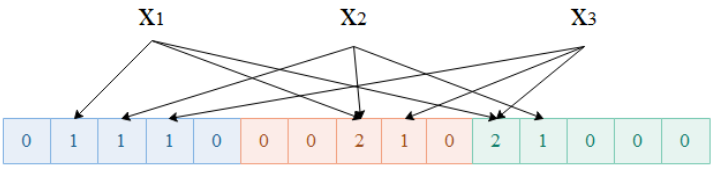

图 4 并发 Bloom Filter

上图中，每种颜色的区域由一个 Hash 函数来控制，这样在运行的时候就可以按照 Hash 函数来开启线程，各自执行自己的插入删除操作。而对数组每一个单元，仍旧采用 int 类型的值来存储，以支持数据的删除。

这种分区的操作相比直接并发，省去了创建和操作互斥锁的开销，同时相比普通的 Bloom Filter 又提高了并发度，具体的性能测试会在后面给出。

III.3 操作流程设计

为了简化问题，我们接下来考虑一维单个 Hash 数组。对于更复杂的问题，可以直接迁移考虑。

一次添加请求包括要添加的数据集合$S' = \{a_1, a_2 \dots a_i\}$，当请求到来时，根据 Hash 函数的个数 k 创建 k 个线程，每个线程针对集合中的所有元素进行一次添加操作。这个添加操作相比原本的 Bloom Filter 也有简化，只计算一个 Hash 函数，并更新 Hash 数组在这个 Hash 函数结果处的值。

一次查询请求包括一个数据a_1，当请求到来时，循环执行 k 个 Hash 函数来判断元素是否在集合中。这里不能使用并发操作，因为每一个元素是否在集合中是要根据五个 Hash 函数的结果来综合判断，如果根据 Hash 函数来开启多线程的话，会出现大量的同步问题，为解决这些同步问题所引入的互斥锁的开销非常大，反而会造成性能损失。

一次删除操作包括要删除的元素集合$S'' = \{b_1, b_2 \dots b_j\}$，其中$b_i (1 \le i \le j)$为集合 S 中的元素，这一保证由其他函数来实现，不作为删除操作的核心步骤。和添加请求一样，根据 Hash 函数的个数 k 创建 k 个线程，每个线程针对集合中的所有元素进行一次删除操作。同样的，这里的删除操作也简化到只计算一个 Hash 函数即可。

IV 理论分析

我们考虑这种情况下的 False Positive 的比率 g。此时，Hash 数组中的任何一位均只能被一个 Hash 函数选中，并且概率为$\frac{k}{m}$，所以这一位不被选中的概率

为$1-\frac{k}{m}$，对 n 个元素而言，有

$$\mathrm{p}'=\left(1-\frac{k}{m}\right)^{n}$$

很显然，有：

$$\mathrm{p}'=\left(1-\frac{k}{m}\right)^{n}\geq\left(1-\frac{1}{m}\right)^{nk}=\mathrm{p}$$

改进后的 False Positive 的比率 g 是略大于 f 的，但是大的并不多，这两个函数都是同阶无穷小，取极限时都为$e^{-\frac{nk}{m}}$，但是性能优势是很明显的。

在改进前，对集合中的每一个元素，都要循环调用五个 Hash 函数来计算，并对 Hash 数组五个位置都进行操作后，再对下一个元素进行操作。而现在 k 个 Hash 函数可以安全的并行访问 Hash 数组，在大量的插入删除请求的情形下，有着极大的性能提升。具体的加速比因环境的不同而不同，在章节 V 实验测试中将会放上改进后的加速结果对比。

另一方面，改进后的结构使得 Hash 数组中的元素分布变得均匀，高效利用了 Hash 数组。而改进前的结构必须要精心选择足够强大的 Hash 函数才能做到这一效果。

V 实验测试

V.1 测试配置

实验中，各参数的配置如下：

表 2 实验参数配置

序号	符号	配置
1	m	3000~7500，每隔 500 进行一次测试
2	n	1500
3	t	200
4	k	5

其中数据集合 n 为收集到的 reddit 网站的 1500 个不重复的 url 网址，实验将使用一般的 Counter Bloom Filter 和改进的五路并发 Bloom Filter 来对这些 url 进行插入删除实验，并计算插入和删除操作所用的时间。

测试集合 t 为 200 个一般 url 网址，并且全部和之前的 1500 个网址不同，即如果在 Bloom Filter 中查询这些 url，应该全部返回 false。据此我们就可以根据查询这些 url 返回的 true 的个数来计算 False Positive 的比率。

Hash 数组宽度 m 使用 3000~7500，每 500 进行一次测试，这是由于考虑到性能原因，大部分已有的 Hash 算法库都保证 Hash 数组的宽度至少为元素个数的两倍，因此从 3000 开始测试。

Hash 函数使用了五个各不相同的 Hash 函数，算法囊括相加，相乘，移位，旋转和固定数字函数迭代等。

这里实现了一个普通的 Counter Bloom Filter 测试和五路并发 Bloom Filter 测试，代码如下：

代码段 1 bloomfilter.cpp

```
/*
 * bloomfilter.cpp
 *
 * TEST for HUST IOT Storage lab
 *
 * Crated by Pan Yue at 2019-06-08
 */

#include <iostream>
#include <fstream>
#include "pylib/debug.h"
#include "pylib/sys/time.h"

using namespace pylib;
using namespace std;

class BloomFilter {
private:
  int size;
  int *tables;
  int Hasher1(std::string str) {
    int ret = 0;
    for (int i = 0; i < str.length(); i++)
      ret += str[i];
    return ret % size;
  }
  int Hasher2(std::string str) {
    int ret = 1;
    for (int i = 0; i < str.length(); i++) {
      ret = ret * 33 + str[i];
    }
    if (ret < 0) ret = -ret;
    return ret % size;
  }
  int Hasher3(std::string str) {
    int ret = str.length();
    for (int i = 0; i < str.length(); i += 2) {
      ret = ((ret >> 28) ^ (ret << 4)) ^ str[i];
    }
    if (ret < 0) ret = -ret;
    return ret % size;
  }
  int Hasher4(std::string str) {
    int ret = 1;
    int p = (int)2166136261L;
    for (int i = 0; i < str.length(); i += 2) {
      ret = (ret ^ str[i]) * p;
    }
    ret += ret << 13;
```

```
    ret ^= ret >> 7;
    ret += ret << 3;
    ret ^= ret >> 17;
    ret += ret << 5;
    if (ret < 0) ret = -ret;
    return ret % size;
  }
  int Hasher5(std::string str) {
    int ret = 0;
    int a = 63689;
    int b = 378551;
    for (int i = 0; i < str.length(); i += 2) {
      ret = ret * a + str[i];
      a = a * b;
    }
    if (ret < 0) ret = -ret;
    return ret % size;
  }

public:
  BloomFilter(int size) {
    this->size = size;
    tables = new int[this->size];
    if (tables == nullptr)
      err_msg("New table error");
    for (int i = 0; i < this->size; i++)
      tables[i] = 0;
  }
  ~BloomFilter() { delete[] tables; }
  bool Add(std::string str) {
    int ret[5];
    ret[0] = Hasher1(str);
    ret[1] = Hasher2(str);
    ret[2] = Hasher3(str);
    ret[3] = Hasher4(str);
    ret[4] = Hasher5(str);
    for (int i = 0; i < 5; i++) {
      tables[ret[i]]++;
    }
    return true;
  }
  bool Find(std::string str) {
    int find = true;
    int ret[5];
    ret[0] = Hasher1(str);
    ret[1] = Hasher2(str);
    ret[2] = Hasher3(str);
    ret[3] = Hasher4(str);
    ret[4] = Hasher5(str);
    for (int i = 0; i < 5; i++) {
      if (tables[ret[i]] == 0) {
        find = false;
        break;
      }
    }
    return find;
  }
  bool Delete(std::string str) {
```

```
    int ret[5];
    ret[0] = Hasher1(str);
    ret[1] = Hasher2(str);
    ret[2] = Hasher3(str);
    ret[3] = Hasher4(str);
    ret[4] = Hasher5(str);
    for (int i = 0; i < 5; i++) {
      tables[ret[i]]--;
    }
    return true;
  }
};

int main(int argc, const char** argv) {
  Timer Start;

  ifstream datafile, testfile, deletefile;
  datafile.open("data.txt");
  testfile.open("test.txt");
  deletefile.open("data.txt");

  BloomFilter bf(7500);
  string str;

  while(getline(datafile, str)) {
    bf.Add(str);
  }
  Timer Afteradd;

  int counter = 0;
  while(getline(testfile, str)) {
    if (bf.Find(str) == true)
      counter++;
  }
  Timer Afterfind;

  while(getline(deletefile, str)) {
    bf.Delete(str);
  }
  Timer Afterdelete;

  cout << "Results" << endl;
  cout << "      Add time: " << (Afteradd - Start).AccurateMicrosecond() <<
endl;
  cout << "     Find time: " << (Afterfind - Afteradd).AccurateMicrosecond()
<< endl;
  cout << "   Delete time: " << (Afterdelete - Afterfind).AccurateMicrosecond()
<< endl;
  cout << "False Positive: " << counter << endl;

  return 0;
}
```

代码段 2 bloomfilternew.cpp

```
/*
 * bloomfilternew.cpp
```

```
 *
 * TEST for HUST IOT Storage lab
 *
 * Crated by Pan Yue at 2019-06-08
 */

#include <iostream>
#include <fstream>
#include <thread>
#include "pylib/debug.h"
#include "pylib/sys/time.h"

using namespace pylib;
using namespace std;

#define SIZE 5000

int Hasher1(std::string str) {
  int ret = 0;
  for (int i = 0; i < str.length(); i++)
    ret += str[i];
  return ret % (SIZE / 5);
}
int Hasher2(std::string str) {
  int ret = 1;
  for (int i = 0; i < str.length(); i++) {
    ret = ret * 33 + str[i];
  }
  if (ret < 0)
    ret = -ret;
  return ret % (SIZE / 5) + (SIZE / 5);
}
int Hasher3(std::string str) {
  int ret = str.length();
  for (int i = 0; i < str.length(); i += 2) {
    ret = ((ret >> 28) ^ (ret << 4)) ^ str[i];
  }
  if (ret < 0)
    ret = -ret;
  return ret % (SIZE / 5) + (SIZE / 5) * 2;
}
int Hasher4(std::string str) {
  int ret = 1;
  int p = (int)2166136261L;
  for (int i = 0; i < str.length(); i += 2) {
    ret = (ret ^ str[i]) * p;
  }
  ret += ret << 13;
  ret ^= ret >> 7;
  ret += ret << 3;
  ret ^= ret >> 17;
  ret += ret << 5;
  if (ret < 0)
    ret = -ret;
  return ret % (SIZE / 5) + (SIZE / 5) * 3;
}
int Hasher5(std::string str) {
  int ret = 0;
```

```
  int a = 63689;
  int b = 378551;
  for (int i = 0; i < str.length(); i += 2) {
    ret = ret * a + str[i];
    a = a * b;
  }
  if (ret < 0)
    ret = -ret;
  return ret % (SIZE / 5) + (SIZE / 5) * 4;
}

bool Add(int *bf, std::string str, int f(std::string)) {
  auto ret = f(str);
  bf[ret]++;
  return true;
}

bool Find(int *bf, std::string str) {
  int find = true;
  int ret[5];
  ret[0] = Hasher1(str);
  ret[1] = Hasher2(str);
  ret[2] = Hasher3(str);
  ret[3] = Hasher4(str);
  ret[4] = Hasher5(str);
  for (int i = 0; i < 5; i++) {
    if (bf[ret[i]] == 0) {
      find = false;
      break;
    }
  }
  return find;
}

bool Delete(int *bf, std::string str, int f(std::string)) {
  auto ret = f(str);
  bf[ret]--;
  return true;
}

int main(int argc, const char** argv) {
  Timer Start;

  ifstream testfile;
  testfile.open("test.txt");
  int bf[SIZE];

  std::thread t1([](int *bf){
    ifstream datafile;
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Add(bf, str, Hasher1);
    }
  }, bf);

  std::thread t2([](int *bf){
    ifstream datafile;
```

```
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Add(bf, str, Hasher2);
    }
  }, bf);

  std::thread t3([](int *bf){
    ifstream datafile;
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Add(bf, str, Hasher3);
    }
  }, bf);

  std::thread t4([](int *bf){
    ifstream datafile;
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Add(bf, str, Hasher4);
    }
  }, bf);

  std::thread t5([](int *bf){
    ifstream datafile;
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Add(bf, str, Hasher5);
    }
  }, bf);


  t1.join();
  t2.join();
  t3.join();
  t4.join();
  t5.join();

  Timer Afteradd;

  int counter = 0;
  string str;
  while(getline(testfile, str)) {
    if (Find(bf, str) == true)
      counter++;
  }
  Timer Afterfind;

  std::thread t6([](int *bf){
    ifstream datafile;
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Delete(bf, str, Hasher1);
    }
```

```
  }, bf);

  std::thread t7([](int *bf){
    ifstream datafile;
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Delete(bf, str, Hasher2);
    }
  }, bf);

  std::thread t8([](int *bf){
    ifstream datafile;
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Delete(bf, str, Hasher3);
    }
  }, bf);

  std::thread t9([](int *bf){
    ifstream datafile;
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Delete(bf, str, Hasher4);
    }
  }, bf);

  std::thread t10([](int *bf){
    ifstream datafile;
    datafile.open("data.txt");
    string str;
    while(getline(datafile, str)) {
      Delete(bf, str, Hasher5);
    }
  }, bf);

  t6.join();
  t7.join();
  t8.join();
  t9.join();
  t10.join();
  Timer Afterdelete;

  cout << "Results" << endl;
  cout << "      Add time: " << (Afteradd - Start).AccurateMicrosecond() <<
endl;
  cout << "     Find time: " << (Afterfind - Afteradd).AccurateMicrosecond()
<< endl;
  cout << "   Delete time: " << (Afterdelete - Afterfind).AccurateMicrosecond()
<< endl;
  cout << "False Positive: " << counter << endl;

  return 0;
}
```

V.2 测试理论结果

利用章节 III.1.2 中推导的公式，我们可以针对改进前和改进后的 Bloom Filter 计算其 False Positive 的比率，结果如下：

$$p = \left(1 - e^{-\frac{nk}{m}}\right)^k$$

表 3 False Positive 比率理论近似值

m	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
f	0.651	0.535	0.435	0.351	0.283	0.228	0.185	0.150	0.123	0.101

V.3 测试结果与分析

调整参数 m 从 3000 到 7500，运行 bloomfilter 程序，得到的结果如下列图片所示：

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 5841
     Find time: 759
   Delete time: 5769
False Positive: 121
```

图 5 改进前：m=3000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 5820
     Find time: 801
   Delete time: 5600
False Positive: 110
```

图 6 改进前：m=3500

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 5940
     Find time: 767
   Delete time: 5831
False Positive: 79
```

图 7 改进前：m=4000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 5810
     Find time: 766
   Delete time: 5594
False Positive: 73
```

图 8 改进前：m=4500

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 6245
     Find time: 824
   Delete time: 4239
False Positive: 64
```

图 9 改进前：m=5000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 5841
     Find time: 773
   Delete time: 5669
False Positive: 57
```

图 10 改进前：m=5500

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 5959
     Find time: 773
   Delete time: 5614
False Positive: 40
```

图 11 改进前：m=6000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 6134
     Find time: 770
   Delete time: 5888
False Positive: 25
```

图 12 改进前：m=6500

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 6547
     Find time: 1681
   Delete time: 6452
False Positive: 22
```

图 13 改进前：m=7000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilter
Results
      Add time: 6452
     Find time: 810
   Delete time: 5917
False Positive: 26
```

图 14 改进前：m=7500

调整参数 m 从 3000 到 7500，运行 bloomfilternew 程序，得到的结果如下列图片所示：

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 3203
     Find time: 681
   Delete time: 2848
False Positive: 135
```

图 15 改进后：m=3000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 3547
     Find time: 772
   Delete time: 2921
False Positive: 117
```

图 16 改进后：m=3500

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 3795
     Find time: 809
   Delete time: 3106
False Positive: 104
```

图 17 改进后：m=4000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 3929
     Find time: 808
   Delete time: 2987
False Positive: 97
```

图 18 改进后：m=4500

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 3797
     Find time: 794
   Delete time: 2999
False Positive: 78
```

图 19 改进后：m=5000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 3686
     Find time: 626
   Delete time: 2184
False Positive: 50
```

图 20 改进后：m=5500

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 3934
     Find time: 1406
   Delete time: 3169
False Positive: 45
```

图 21 改进后：m=6000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 3757
     Find time: 773
   Delete time: 4286
False Positive: 40
```

图 22 改进后：m=6500

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 4201
     Find time: 776
   Delete time: 3002
False Positive: 38
```

图 23 改进后：m=7000

```
panyue@zxcpyp ~/code/HUST-Storage ./bloomfilternew
Results
      Add time: 4317
     Find time: 1424
   Delete time: 2808
False Positive: 35
```

图 24 改进后：m=7500

根据上述测试结果，将 False Positive 比率的理论值和真实值作比较，绘制表格如下：

表 4 False Positive 比率理论值和真实值比较

m	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
理论	0.651	0.535	0.435	0.351	0.283	0.228	0.185	0.150	0.123	0.101
f	0.605	0.55	0.395	0.365	0.32	0.285	0.200	0.125	0.11	0.13
g	0.675	0.585	0.52	0.485	0.39	0.250	0.225	0.200	0.19	0.175

绘制曲线图如下：

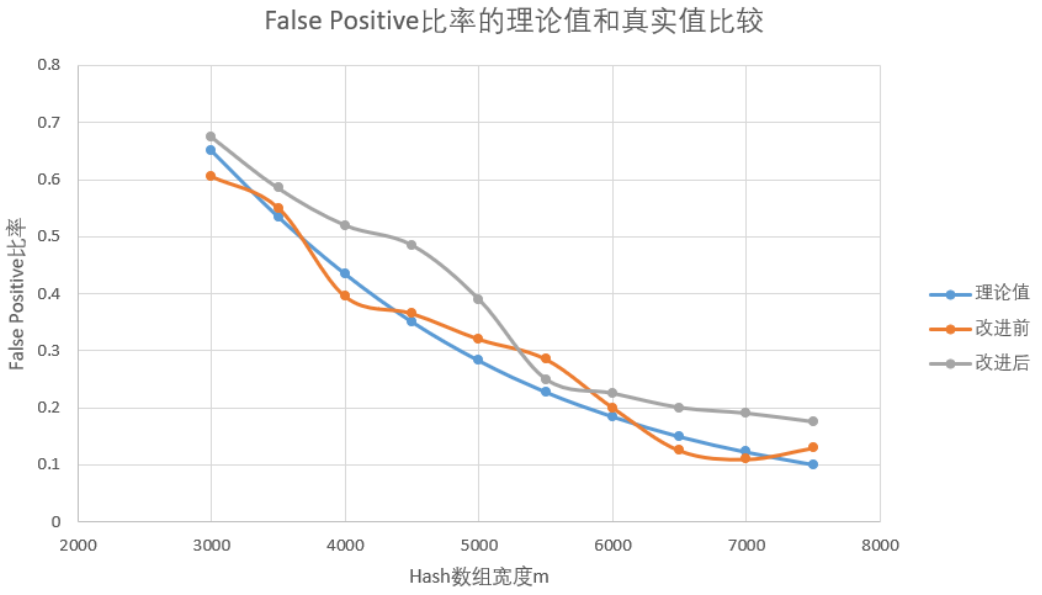

图 25 False Positive 的理论值和真实值曲线图

从上图中我们可以看出，改进后的 False Positive 是要略低于改进前的，这符合我们之前的理论计算，但是差的并不多。而且，不管是改进前还是改进后，都和真实值很接近，这也证明了我们理论推导的正确性。

而在程序的性能上，改进后的 Bloom Filter 则体现出了明显的优势。我们可以比较改进前和改进后插入和删除的时间，有如下两个表格：

表 5 改进前和改进后的性能比较（插入）

m	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
插入时间改进前(微秒)	5841	5820	5940	5810	6245	5841	5959	6134	6547	6452
插入时间改进后(微秒)	3203	3547	3795	3929	3797	3686	3934	3757	4201	4317

表 6 改进前和改进后的性能比较（删除）

m	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
删除时间改进前(微秒)	5769	5600	5831	5594	4239	5669	5614	5888	6452	5917
删除时间改进后(微秒)	2848	2921	3106	2987	2999	2184	3169	4286	3002	2808

从上述两个表格中可以看到，改进前的平均插入时间为 6058.9 微秒，平均删除时间为 5657.3 微秒，而改进后的平均插入时间为 3816.6 微秒，平均删除时间为 3031 秒。

插入速度是改进前的 1.58 倍，删除速度是改进前的 1.86 倍，改进后的性能有了极大提升。

从横向来看，总体上执行时间随着 m 的增大而增大，但增大的不是特别多，m 的规模并不是影响执行时间的主要因素。

VI 结语

本文首先通过对普通 Bloom Filter 的流程分析，计算了 Bloom Filter 的理论性能参数。接着针对 Bloom Filter 存在的问题进行改进，提出了一种 Hash 函数分区，并行计算的改进结构。并通过理论分析与编写代码测试两方面，论证了理

论性能分析的正确性和改进的优越性。

综上所述，本文中改进的多路并行的 Bloom Filter 尽管降低了一点 False Positive 的比率，但却在性能上获得了极大的提升，总体是优于题目中原先的 Counter Bloom Filter 的。并且本文中的改进模型并没有影响 Bloom Filter 本身具有的功能，插入、删除和查询的执行也都正常，是一种可行的 Bloom Filter 改进设计方案。

VIII 参考文献

[1] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, “Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines,” Proc. ACM SIGCOMM, 2006.

[2] Y. Zhu and H. Jiang, “False Rate Analysis of Bloom Filter Replicas in Distributed Systems,” Proc. Int’l Conf. Parallel Processing (ICPP ’06), pp. 255-262, 2006.

[3] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor, “Longest Prefix Matching Using Bloom Filters,” Proc. ACM SIGCOMM, pp. 201-212, 2003.

[4] L. Fan, P. Cao, J. Almeida, and A. Broder, “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol,” IEEE/ACM Trans. Networking, vol. 8, no. 3, pp. 281-293, June 2000.

[5] B. Xiao and Y. Hua, “Using Parallel Bloom Filters for Multi-Attribute Representation on Network Services,” IEEE Trans. Parallel and Distributed Systems, vol. 21, no. 1, pp. 20-32, Jan. 2010.

[6] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, “Scalable and Adaptive Metadata Management in Ultra Large-scale File Systems,” Proc. 28th Int’l Conf. Distributed Computing Systems (ICDCS ’08), pp. 403-410, 2008.

[7] D. Guo, J. Wu, H. Chen, and X. Luo, “Theory and Network Application of Dynamic Bloom Filters,” Proc. IEEE INFOCOM, 2006.