2018 级

《物联网数据存储与管理》课程

课 程 报 告

姓　名　李嘉辉

学　号　U201890060

班　号　物联网 1801 班

日　期　2021.06.28

目　　录

一、摘要

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。Bloom Filter 的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。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 通过极少的错误换取了存储空间的极大。

本文基于 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 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：Bloom-filter，hash

二、选题背景与意义

Bloom filter（布隆过滤器）是 Howard Bloom 在 1970 年提出的二进制向量 数据结构，具有良好的空间和时间效率，用于检测某元素是否为集合的成员。

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。

三、总体设计

3.1 Bloom Filter 基本思想

3.1.1 BloomFilter 原理：

当一个元素被加入集合时，通过 k 个散列函数将这个元素映像成一个位数组中的 k 个点，把它们置为 1。检索时，我们只要看看这些点是不是都是 1 就（大约）知道集合中有没有它了,如果这些点有任何一个 0，则被检元素一定不在；如果都是 1，则被检元素很可能在。

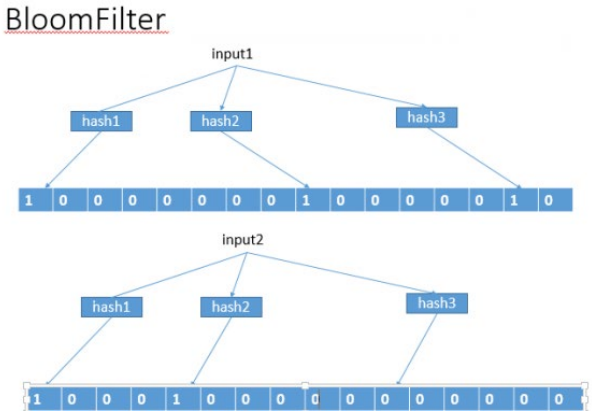

图 3-1 BloomFliter 原理图

如上图所示，我们定义了一个 16 位的二进制向量，3 个 hash 函数，这个 3 个函数 hash 的结果为 0 或者 1，该结果存放的位置为 0~15 之间，将 hash 的结果的位置映像到二进制向量的 index，并保存结果.

对于输入数据 input1，得到的的结果存在于 0，8，14，结果全部为 1，那么说明 input1 可能存在于指定的集合。

对于 input2，得到的结果存在于 0，4，10，有一个是 0，那么说明 input2 一定不存在于指定的集合。

3.1.2 Bloom Filter 的性能分析

分析 Bloom Filter 的性能问题即 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:

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

对上式做变换，并利用利用自然指数极限的代换，有：

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

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：

$$\mathrm{f}=(1-p)^k\approx\left(1-e^{-\frac{nk}{n}}\right)^k=e^{kln\left(1-e^{-\frac{nk}{m}}\right)}$$

$$=e^{-k\cdot\frac{m}{nk}lne^{-\frac{nk}{m}}\ln(1-e^{-\frac{nk}{m}})}=e^{-\frac{m}{n}\ln(p)\ln(1-p)}$$

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 - p)取最大值，此时有 $\mathrm{p}=\frac{1}{2}$，即$e^{-\frac{nk}{m}}=\frac{1}{2}$，$\mathrm{k}=\frac{m}{n}ln2\approx0.7\frac{m}{n}$，此时给出的 f 为:

$$\mathrm{f}=(\frac{1}{2})^k\approx0.6185^{\frac{m}{n}}$$

3.1.3 Bloom Filter 的不足

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$\mathrm{k}=\ln2\times\frac{m}{n}$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。

3.2 数据结构设计

3.2.1 Counter Bloom Filter 结构

如下图，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。

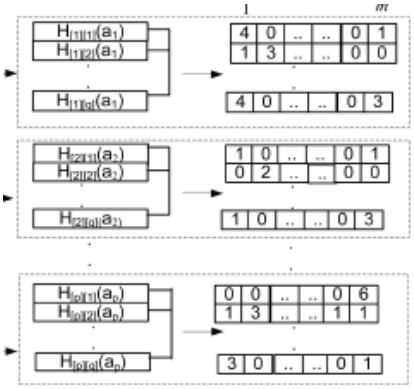

图 3-2 多维 Bloom Filter 结构

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。

3.2.2 并发哈希 Bloom Filter 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~$\frac{m}{k}$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。其结构下图所示：

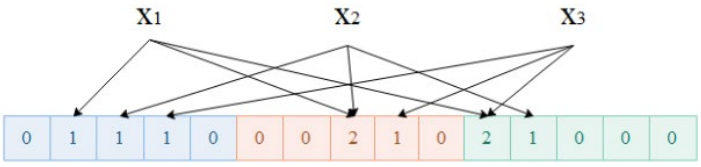

图 3-3 并发 Bloom Filter

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。

3.3 操作流程

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'= { $a_1, a_2 \ldots a_i$}，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''= {$b_1, b_2 \ldots b_j$ }，其中b_j (1≤i≤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。

四、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 $p' = (1 - \frac{k}{m})^n$

很显然，有：

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

并发哈希 Bloom Filter 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。

对于 Counter Bloom Filter，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，在接下来的实验测试中将会给出两个 Bloom Filter 的速度对比。

另一方面，并发哈希的结构使得 Hash 数组中的元素分布变得均匀，高效利用了 Hash 数组。而 Counter Bloom Filter 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。

五、实验测试

5.1 实验说明

实验中，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。

Counter Bloom Filter 和五路并发 Bloom Filter 代码可在附录中找到。

5.2 实验理论结果

利用 3.1.2 中推导的公式，我们可以针对 Counter Bloom Filter 和五路并发 Bloom Filter 计算其 False Positive 的比率，结果如下：

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

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

表 5-1 False Positive 比率理论近似值

5.3 实验结果与分析

调整参数 m 从 3000 到 7500，运行 count_bloomfilter 和 five_ways_bloomfilter 程序，图 5-1 和 5-2 为两个程序 m 取 3000 时的结果：

```
D:\Users\Administrator\CLionProjects\
Results
      Add time: 5841
     Find time: 759
   Delete time: 5769
False Positive: 121
```

图 5-1 count_bloomfilter 程序结果

```
D:\Users\Administrator\CLionProjects\
Results
      Add time: 3203
     Find time: 681
   Delete time: 2848
False Positive: 135
```

图 5-2 five_ways_bloomfilter 程序结果

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

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
count	0.605	0.550	0.395	0.365	0.320	0.285	0.200	0.125	0.110	0.130
Five ways	0.675	0.585	0.520	0.485	0.390	0.250	0.225	0.200	0.19	0.175

表 5-2 False Positive 比率理论值和真实值比较

绘制曲线图如下：

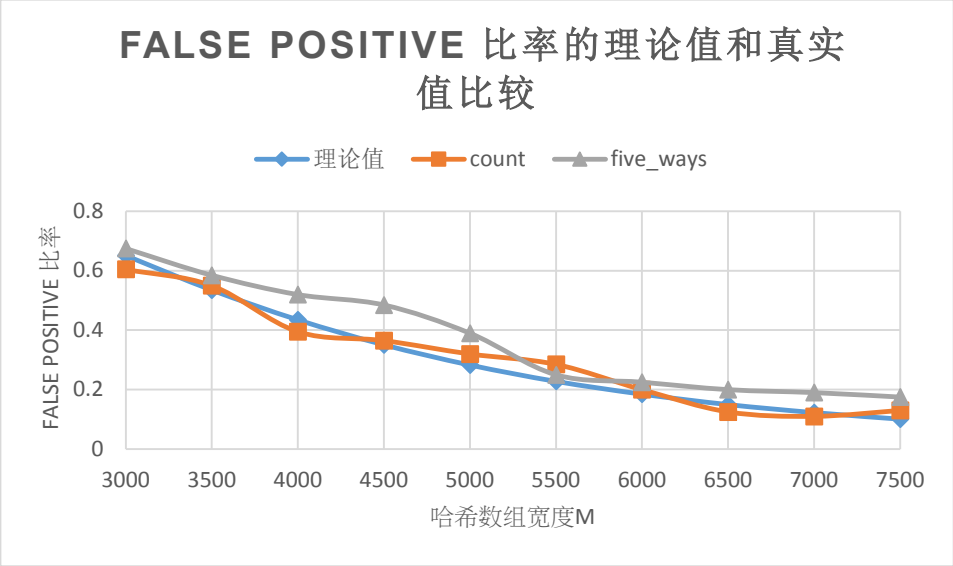

图 5-3 False Positive 的理论值和真实值曲线图

从上图中我们可以看出，五路并发 Bloom Filter 的 False Positive 是要略低于 Count Bloom Filter，这符合我们之前的理论计算，但是差的并不多。而且，两者都和真实值很接近，这也证明了我们理论推导的正确性。

而在程序的性能上，五路并发 Bloom Filter 则体现出了明显的优势。我们可以比较两者插入和删除的时间，有如下两个表格：

m	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
Count	5841	5820	5940	5810	6245	5841	5959	6134	6547	6452
Five ways	3203	3547	3795	3929	3797	3686	3934	3757	4201	4317

表 5-3 兩個程序的性能比较（插入）

m	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
Count	5841	5820	5940	5810	6245	5841	5959	6134	6547	6452
Five ways	3203	3547	3795	3929	3797	3686	3934	3757	4201	4317

表 5-4 兩個程序的性能比较（删除）

从上述两个表格中可以看到，Count Bloom Filter 的平均插入时间为 6058.9 微秒，平均删除时间为 5657.3 微秒，而五路并发 Bloom Filter 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 Count Bloom Filter 的 1.58 倍，删除速度是 Count Bloom Filter 的 1.86 倍，五路并发 Bloom Filter 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。

六、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。

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.

附录

count_bloomfilter 程序：

```
/*
 * count_bloomfilter.cpp
 */

#include <iostream>
#include <fstream>
#include <windows.h>
#include "string"
#include<stdio.h>
namespace bloom {
	// DEBUG MESSAGE FUNCTIONS
	// Arg is C style string

	// Print success message
	inline void succ_msg(const char *err) {
		printf("[SUCCEED] %s\n", err);
	}

	// Print error message
	inline void err_msg(const char *err) {
		printf("[ERROR] %s\n", err);
	}

	// Print usage
	inline void usage_err(const char *err) {
		printf("Usage: %s\n", err);
	}

	// Print error message and exit
	inline void err_exit(const char *err) {
		printf("[FATAL] %s\n", err);
		exit(1);
	}

	// println - Output the Arg
	inline void println() {
		std::cout << std::endl;
	}

	template<typename T, typename ...ARGS>
	inline void println(const T &first, const ARGS ...last) {
		std::cout << first << " ";
		println(last...);
	}

	// mylog - Output logs
	inline void mylog() {
		std::clog << std::endl;
	}

	template<typename T, typename ...ARGS>
	inline void mylog(const T &first, const ARGS ...last) {
		std::clog << first << " ";
		mylog(last...);
```

```
	}
}
using namespace std;
using namespace bloom;
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) {
	LARGE_INTEGER cpuFreq,startTime,Afteradd,Afterfind,Afterdelete;
	double runTime=0.0;
	int Time;
	QueryPerformanceFrequency(&cpuFreq);
	QueryPerformanceCounter(&startTime);
	ifstream datafile, testfile, deletefile;
```

```
	datafile.open("D:\\data.txt");
	testfile.open("D:\\test.txt");
	deletefile.open("D:\\data.txt");

	BloomFilter bf(6000);
	string str;

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

	QueryPerformanceCounter(&Afteradd);

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

	QueryPerformanceCounter(&Afterfind);
	while(getline(deletefile, str)) {
		bf.Delete(str);
	}

	QueryPerformanceCounter(&Afterdelete);
	cout << "Results" << endl;
	runTime = (((Afteradd.QuadPart - startTime.QuadPart) * 100000.0f) / cpuFreq.QuadPart);
	Time = (int)runTime;
	cout << "           Add time: " <<    Time<< endl;
	runTime = (((Afterfind.QuadPart - Afteradd.QuadPart) * 100000.0f) / cpuFreq.QuadPart);
	Time = (int)runTime;
	cout << "          Find time: " <<    Time<< endl;
	runTime = (((Afterdelete.QuadPart - Afterfind.QuadPart) * 100000.0f) / cpuFreq.QuadPart);
	Time = (int)runTime;
	cout << "      Delete time: " <<    Time<< endl;
	cout << "False Positive: " << counter << endl;

	return 0;
}
```

five_ways_bloomfilter 程序：

```
/*
 * five_ways_bloomfilter.cpp
 */

#include <iostream>
#include <fstream>
#include <thread>
#include <windows.h>
#include "string"
namespace bloom {
	// DEBUG MESSAGE FUNCTIONS
	// Arg is C style string

	// Print success message
	inline void succ_msg(const char *err) {
		printf("[SUCCEED] %s\n", err);
```

```
	}

	// Print error message
	inline void err_msg(const char *err) {
		printf("[ERROR] %s\n", err);
	}

	// Print usage
	inline void usage_err(const char *err) {
		printf("Usage: %s\n", err);
	}

	// Print error message and exit
	inline void err_exit(const char *err) {
		printf("[FATAL] %s\n", err);
		exit(1);
	}

	// println - Output the Arg
	inline void println() {
		std::cout << std::endl;
	}

	template<typename T, typename ...ARGS>
	inline void println(const T &first, const ARGS ...last) {
		std::cout << first << " ";
		println(last...);
	}

	// mylog - Output logs
	inline void mylog() {
		std::clog << std::endl;
	}

	template<typename T, typename ...ARGS>
	inline void mylog(const T &first, const ARGS ...last) {
		std::clog << first << " ";
		mylog(last...);
	}
}
using namespace bloom;
using namespace std;

#define SIZE 6500

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) {
	LARGE_INTEGER cpuFreq,startTime,Afteradd,Afterfind,Afterdelete;
	double runTime=0.0;
	int Time;
	QueryPerformanceFrequency(&cpuFreq);
	QueryPerformanceCounter(&startTime);

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

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

	std::thread t2([](int *bf){
		ifstream datafile;
		datafile.open("D:\\data.txt");
		string str;
		while(getline(datafile, str)) {
			Add(bf, str, Hasher2);
		}
	}, bf);

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

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

	std::thread t5([](int *bf){
```

```
        ifstream datafile;
        datafile.open("D:\\data.txt");
        string str;
        while(getline(datafile, str)) {
            Add(bf, str, Hasher5);
        }
    }, bf);


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

    QueryPerformanceCounter(&Afteradd);

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

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

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

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

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

```
        }
    }, bf);

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

    t6.join();
    t7.join();
    t8.join();
    t9.join();
    t10.join();
    QueryPerformanceCounter(&Afterdelete);

    cout << "Results" << endl;
    runTime = (((Afteradd.QuadPart - startTime.QuadPart) * 100000.0f) / cpuFreq.QuadPart);
    Time = (int)runTime;
    cout << "           Add time: " <<    Time<< endl;
    runTime = (((Afterfind.QuadPart - Afteradd.QuadPart) * 100000.0f) / cpuFreq.QuadPart);
    Time = (int)runTime;
    cout << "          Find time: " <<    Time<< endl;
    runTime = (((Afterdelete.QuadPart - Afterfind.QuadPart) * 100000.0f) / cpuFreq.QuadPart);
    Time = (int)runTime;
    cout << "      Delete time: " <<    Time<< endl;
    cout << "False Positive: " << counter << endl;

    return 0;
}
```