中国科学技术大学

# 读书报告


## 布隆过滤器及其衍生的新型数据结构

作者姓名： 柳枫

学科专业： 网络空间安全

导师姓名： 薛开平教授

完成时间： 二〇二四年九月十一日

# 目　录

# 符 号 说 明

| | |
|---|---|
| $a$ | The number of angels per unit area |
| $N$ | The number of angels per needle point |
| $A$ | The area of the needle point |
| $\sigma$ | The total mass of angels per unit area |
| $m$ | The mass of one angel |
| $\sum_{i=1}^{n} a_i$ | The sum of $a_i$ |

# 第 1 章　简　　介

## 1.1　布隆过滤器

在构造网络安全协议时，我们通常会使用到许多不同类型的数据结构。其中，布隆过滤器作为一种经典的数据结构，在诸如隐私集合求交、可搜索加密、隐私信息检索等密码学协议中有着广泛的应用。布隆过滤器是一种用于快速判断元素是否存在于某一集合的数据结构，它具有空间效率高、判断速度快的特点。以大小为 $n$ 的集合 $S$ 为例，对应的布隆过滤器构造只需要 $O(n)$ 的存储开销以及 $O(1)$ 的判断复杂度。布隆过滤器的构造如下图所示，它是使用 $k$ 个哈希函数 $\{h_1, \cdots, h_k\}$ 构造的哈希表结构。布隆过滤器上分为插入（Insert）和查找（Lookup）两个算法，在构造过程中，对于每个在集合 $S$ 中的元素 $x$，首先使用这 $k$ 个哈希函数计算出 $k$ 个位置，然后对过滤器中该位置上的比特置为 1。在判断元素是否属于集合 $S$ 时，只需要通过哈希函数计算该元素的 $k$ 个位置，然后检查过滤器上这 $k$ 个位置上的比特是否全为 1。如果是，返回 True，否则返回 False。从布隆过滤器的构造可以看出，

文献[1]：

布隆过滤器（Bloom filter，BF）是一种空间效率高的概率型数据结构，它可以用来快速判断元素是否属于某一集合。布隆过滤器是由 0-1 比特组成的比特向量结构，包含插入（Insert）和查找（Lookup）两个基本算法。以包含 $n$ 个元素的集合 $S = \{x_1, x_2, \cdots, x_n\}$ 为例，假设构造的布隆过滤器长度为 $m$，使用的哈希函数为 $\{h_1, h_2, \cdots, h_k\}$，其中每个哈希函数 $h_i : \{0,1\}^* \to [0, m-1]$ 为任意长度的输入到布隆过滤器上某一位置的映射。首先我们将布隆过滤器 $m$ 个位置上的比特都置为 0，然后再插入集合 $S$ 中的每一个元素。在插入元素 $x$ 时，需要使用 $k$ 个哈希函数计算出 $k$ 个位置信息，即 $\{h_1(x), h_2(x), \cdots, h_k(x)\}$。再将布隆过滤器上这 $k$ 个位置上的比特都置为 1。在判断某个元素是否属于集合 $S$ 时，只需要计算该元素对应的 $k$ 个位置，然后检查这 $k$ 个位置上的比特是否都为 1。只要有一个位置上出现了 0，那么判断结果就是不属于；否则，布隆过滤器认为该元素属于集合 $S$。

从布隆过滤器的构造和判断算法中可以看出，如果一个元素属于集合 $S$，那么判断结果一定是正确的；但是如果一个元素不属于集合 $S$（False 的情况），布隆过滤器也有可能会认为该元素属于 $S$（输出结果为 Positive），此时判断错误的概率也称为假阳性率（False Positive Rate）。尽管布隆过滤器存在误判的问题，但在实际应用场景中，只要将误判率控制在较小的值，一般认为以一定的误判换取低空间开销和高效判断是值得的。

布隆过滤器的误判率 $f_r$ 由布隆过滤器的长度 $m$，使用的哈希函数个数 $k$ 和集合中的元素个数 $n$ 所决定。理论上，误判率 $f_r$ 与它们的关系如公式 (1.1)所示：

$$f_r = \left[1 - \left(1 - \frac{1}{m}\right)^{nk}\right]^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k, \tag{1.1}$$

其中，$(1 - 1/m)^{nk}$ 近似成 $e^{-kn/m}$ 的形式。为了尽可能降低误判率 $f_r$，那么就需要尽可能降低 $e^{-kn/m}$ 的值，这样一来，$k$ 的最优取值为：

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n}. \tag{1.2}$$

此时，误判率大约为 $0.5^k \approx 0.6185^{m/n}$。通常在实际应用中的误判率要比理论分析上的更高。也有一些工作对误判率做了更精确的刻画。

布隆过滤器本身是不支持删除的，因为如果是简单将需要删除元素所对应位置的比特置为 0，那么就会对其他元素的判断造成影响。

布隆过滤器的高效体现在两个方面：

- 空间利用率：布隆过滤器的大小与元素的大小无关，只与集合中元素的数量有关。比如，当给定 $m$ 与 $n$ 的比值为 5 时，根据公式 (1.2) 可以计算出需要的哈希函数数量为 3 或 4。因为布隆过滤器中每个位置上存储的都是比特，所以整体长度也就是 $m$ 比特。
- 恒定时间的查询时间：因为只需要检查 $k$ 个位置上的比特是否全为 1，因此检查一个元素的时间复杂度为 $O(k)$。相比于树形结构的查询效率（$O(\log(n))$）或列表结构的查询效率（$O(n)$）都要高。而对于具体的布隆过滤器实例来说，$k$ 的值在初始化阶段就是常数，因此插入和查询的复杂度都为 $O(1)$。
- 无漏判：尽管布隆过滤器在查询时会存在误判的情况，但是它不会出现漏判（假阴性）的情况。也就是只要是布隆过滤器判断元素 $x$ 不属于 $S$，那么该论断一定是正确的。

布隆过滤器的局限性：

- 误判，如果减少误判带来的影响
- 实现，实际实现中需要考虑访问的优化，哈希计算的优化
- 灵活性，哈希函数不能变，集合一开始就是确定的
- 功能有限，只能做成员存在性测试

降低误判率的方法（思路）：首先，

## 1.2　分类

在介绍布隆过滤器相关衍生的数据结构之前，我们首先需要对这些数据结构进行分类。对此，我们对这些数据结构进行了如下统一的定义。

**定义 1.1**　令 $\mathcal{U}$ 表示元素的集合，$\mathcal{H}$ 为哈希函数的集合。过滤器一般包含以下两个算法：

- Construct$(S,\mathcal{H}) \to F/\perp$：输入集合 $S \subseteq \mathcal{U}$ 和预先给定的哈希函数集合 $\mathcal{H}$，输出构造的过滤器 $F$（或者以可忽略的概率输出错误指示符 $\perp$）。
- Evaluate$(x,\mathcal{H},F) \to$ True/False：输入元素 $x$，预先给定的哈希函数集合 $\mathcal{H}$，输出结果 True 或者 False。

**正确性：**对于任意的 $S \subseteq \mathcal{U}$，都有：1）构建过程中，输出 $\perp$ 的概率是可忽略的；2）如果 $F \leftarrow$ Construct$(S,\mathcal{H})$，且 $F \neq \perp$，那么在判断过程中，对于任意的 $x \in S$，Pr[Evaluate$(x,\mathcal{H},F)$ = True] = 1；对于任意 $x' \notin S$，Pr[Evaluate$(x,\mathcal{H},F)$ = True] 为可忽略的。

从以上定义中可以看出，如果一个元素在原本输入的集合中，那么过滤器在判断过程中一定能返回正确的结果，即过滤器中不存在假阴性的情况；反之，如果一个元素不存在于输入的集合中，过滤器会大概率返回正确的结果，即过滤器中会存在一定的假阳性率。

在判断过程中，需要使用 $\mathcal{H}$ 中的哈希函数计算出元素在 $F$ 中对应的位置，再对这些位置上记录的结果进行计算，最后根据计算结果与事先定义的 $f(x)$ 进行比较返回 Ture 或者 False。计算过程也被称为探测（probing），文献[2]将探测方式分为以下三种类型：

- AND 型：在通过哈希函数计算出的位置中，如果所有位置上结果都与 $f(x)$ 相等，那么就输出 True；
- OR 型：在通过哈希函数计算出的位置中，如果至少有一个位置上的值与 $f(x)$ 相等，那么就输出 True，否则输出 False。
- XOR 型：在通过哈希函数计算出的位置中，如果所有位置上的值的异或结果与 $f(x)$ 相等，那么就输出 True，否则输出 False。

从以上分类可以看出，布隆过滤器的判断方式属于 AND 型。OR 型的典型代表是布谷鸟过滤器（Cuckoo filter），这种构造的特点是支持元素的动态插入和删除。XOR 型的典型代表是异或过滤器（Xor filter），这类过滤器在结构上非常紧凑，具有较高的存储空间利用率，但受限于构建方式，这类构造通常不能支持元素的动态更新。

不同的过滤器中对 $f(x)$ 的定义也略有不同。一般来说分为以下几种情况：

- $f(x)$ 为比特 1，最典型的是布隆过滤器，即要求 $x$ 对应所有位置上的结果都为 1，过滤器才会返回 True。
- $f(x)$ 等于元素 $x$ 本身，典型的是混淆布隆过滤器（garbled Bloom filter），即计算的结果为 $x$，过滤器才会返回 True。
- $f(x)$ 为 $x$ 的指纹（fingerprint），一般指的是 $x$ 的哈希值，典型代表为布谷

鸟过滤器，即当有一个位置上的值与 $x$ 的指纹相同，过滤器才会返回 True。

- $f(x)$ 为任意函数，典型的是 Bloomier 过滤器，也就是说 $f(x)$ 的形式并不重要，或者说只要满足是 $x$ 一种映射关系即可。

从上述分类可以看出，对 $f(x)$ 的定义可以是简单的比特 1，也可以是关于 $x$ 的任意一种映射关系。从另一个角度来看，键值型数据也可以看作是从键到值的映射，这样一来，键值型数据也可以编码成过滤器的形式。利用这种思路构造的结构称作不经意的键值存储（Oblivious Key-Value Store, OKVS），与过滤器不同，OKVS 返回的不是 True 或者 False，而是与输入对应的数值。OKVS 的定义如下：

**定义 1.2**　令 $\mathcal{K}$ 和 $\mathcal{V}$ 分别表示键和值的集合。OKVS 包含两个算法：

- $\mathsf{Encode}(\{(k_1,v_1),(k_2,v_2),\cdots,(k_n,v_n)\})$：输入一组键值对 $\{(k_i,v_i)\}_{i\in[n]}\subseteq\mathcal{K}\times\mathcal{V}$，输出 OKVS 的编码结果 $D$ 或以可忽略的概率输出错误指示符 $\perp$。
- $\mathsf{Decode}(D,k)$：输入 OKVS 的编码结果 $D$ 和键 $k\in\mathcal{K}$，输出值 $v\in\mathcal{V}$。

**正确性：**对于任意键值对集合 $A\subseteq\mathcal{K}\times\mathcal{V}$，如果 $\mathsf{Encode}(A)=D$ 且 $D\neq\perp$，那么对于任意的 $(k,v)\in A$，都有 $\mathsf{Decode}(D,k)=v$。

**不经意性：**随机给定两个不同的集合 $\mathcal{K}_0=\{k_1^0,k_2^0,\cdots,k_n^0\}$ 和 $\mathcal{K}_1=\{k_1^1,k_2^1,\cdots,k_n^1\}$，再随机构造集合 $\{v_i\}_{i\in[n]}\leftarrow\mathcal{V}$，若 $D\neq\perp$，则分布 $\{D|v_i\leftarrow\mathcal{V},i\in[n],\mathsf{Encode}(\{(k_1^0,v_1),(k_2^0,v_2),\cdots,(k_n^0,v_n)\})\}$ 与分布 $\{D|v_i\leftarrow\mathcal{V},i\in[n],\mathsf{Encode}(\{(k_1^1,v_1),(k_2^1,v_2),\cdots,(k_n^1,v_n)\})\}$ 统计不可区分。

对于部分安全需求较高的协议，OKVS 还需要额外满足随机性的性质，其定义如下。

**随机性：**对于任意的集合 $A=\{(k_i,v_i)\}_{i\in[n]}\subseteq\mathcal{K}\times\mathcal{V}$ 和 $k'\notin\{k_1,k_2,\cdots,k_n\}$，如果 $\mathsf{Encode}(A)=D$ 且 $D\neq\perp$，则 $\mathsf{Decode}(D,k')$ 的输出与随机选择的 $v\leftarrow\mathcal{V}$ 统计不可区分。

根据上述讨论内容，表 1.1对这些过滤器以及 OKVS 结构进行了总结，接下来我们将对这些结构进行详细介绍。

**表 1.1　不同过滤器及相关结构总结**

| 名称 | 探测类型 | $f(x)$ |
|---|---|---|
| 布隆过滤器 | AND 型 | 1 |
| 布谷鸟过滤器 | OR 型 | $x$ 的指纹值 |
| 异或过滤器 | XOR 型 | $x$ 的指纹值 |
| Binary Fuse 过滤器 | XOR 型 | $x$ 的指纹值 |
| Bloomier 过滤器 | XOR 型 | $f(x)$ |
| 混淆布隆过滤器 | XOR 型 | $x$ |
| Random-Band OKVS | XOR 型 | $x$ 对应的值 |

# 第 2 章　新型数据结构

在这一章，我们将重点介绍由布隆过滤器衍生出来的几个特殊数据结构，分别是布谷鸟过滤器，异或型过滤器以及不经意的键值存储。

## 2.1　布谷鸟过滤器

从上一章的分类我们可以知道，布谷鸟过滤器是一种 OR 型的过滤器，且 $f(x)$ 为 $x$ 的指纹信息。布谷鸟过滤器的概念最早是由 Fan 等人[3] 于 2014 年提出的，其构造方式受到了布谷鸟哈希表（Cuckoo Hash Table）[4]的启发。在介绍布谷鸟过滤器之前，我们首先介绍布谷鸟哈希表的构造。

### 2.1.1　布谷鸟哈希表

布谷鸟哈希表可以看作一个由多个桶（bucket）组成的数组结构。对于每个元素 $x$ 来说，它在哈希表中对应两个候选位置，分别由两个哈希函数 $h_1(x)$ 和 $h_2(x)$ 决定。在插入元素 $x$ 时，首先检查 $x$ 对应的两个位置上的桶中是否有多余位置。如果两个桶都有多余空间，则直接将 $x$ 放入桶中；如果两个桶都已满，则随机在一个候选位置上踢出一个元素并将 $x$ 放入该位置上。踢出的元素则重新插入到它对应的另一个候选位置上。如图所示，假设每个桶中都只能存储一个元素，当插入元素 $x$ 时，首先计算出它的两个候选位置分别是 2 和 6。因为 2 和 6 两个位置都已满，这里选择踢出位置 6 上的元素 $a$，并将 $x$ 放入其中。被踢出的 $a$ 则重新插入到它的另一个候选位置，也就是 4 上。由于位置 4 上已有元素 $c$，则把 $c$ 踢出，将 $a$ 存储在位置 4 中，并将 $c$ 重新插入到它的候选位置 1 上，最终结果如图所示。这种 “踢出-插入” 的思路与布谷鸟下蛋时会将蛋放入其他鸟的巢穴，并挤出原本巢中蛋的行为很相似，因此而得名。

### 2.1.2　布谷鸟过滤器的构造

与布谷鸟哈希表类似，布谷鸟过滤器也是由多个桶组成的数组结构。不同的是，在布谷鸟过滤器中每个桶中存储的并不是元素本身，而是元素的指纹。这就导致在将桶中的元素指纹踢出时，无法根据指纹信息确定它的另一个候选位置。因此布谷鸟过滤器采用了部分密钥布谷鸟哈希（partial-key cuckoo hashing）的技巧来解决该问题。也就是将元素的两个候选位置与元素的指纹值建立联系，这样就能在只知道元素的指纹值和其中一个位置信息的情况下计算出另一个位置信

息。具体来说，对于元素 $x$，其对应的两个候选位置计算如下：

$$h_1(x) = \mathsf{hash}(x) \tag{2.1}$$

$$h_2(x) = h_1(x) \oplus \mathsf{hash}(x\text{'s fingerprint}) \tag{2.2}$$

式 2.2中的异或操作正好满足了上述性质，即 $h_1(x)$ 也可以通过 $h_2(x)$ 和 $x$ 的指纹信息计算得出。另外，在异或操作中采用的是 $x$ 指纹的哈希而不是 $x$ 的指纹本身，这样做是因为如果只用指纹本身的话，两个候选位置之间的距离就受限于指纹的取值范围。比如使用 8 比特长度的指纹，那么两个候选位置之间最多相差 256。而使用指纹的哈希则可以确保两个候选位置可以分布在过滤器中的任意位置，从而降低哈希碰撞的概率并提高存储空间的利用率。

通过上述讨论，我们可以直接给出布谷鸟过滤器的 Construct 算法思路。以输入集合 $S$ 为例，对于每一个元素 $x \in S$，插入 $x$ 的过程描述如下：

- 首先计算 $x$ 的指纹值 $f$，以及两个候选位置信息 $h_1(x)$ 和 $h_2(x)$。
- 只要 $h_1(x)$ 和 $h_2(x)$ 两个位置上有一个桶有空余，那么直接将 $f$ 插入空余的桶中，否则进入下一步。
- 随机从 $h_1(x)$ 和 $h_2(x)$ 中选取一个位置 $i$，并从该位置上踢出一个指纹，并将 $f$ 插入。踢出的指纹再计算出它的另一个候选位置，执行插入步骤。

在布谷鸟过滤器的构造过程中，会设置一个最大踢出值，当踢出的指纹超过最大值时将直接返回错误指示符 $\perp$。

布谷鸟过滤器的 Evaluate 算法比较直接，给定输入的元素 $x$，只需要计算它对应的两个位置 $h_1(x)$ 和 $h_2(x)$，如果在这两个位置上至少有一个桶中含有 $x$ 的指纹，那么返回 True，否则返回 False。

除了在**定义 1.1**中的 Construct 和 Evaluate 两个算法之外，布谷鸟过滤器中还支持删除操作。这也是布谷鸟过滤器相比布隆过滤器的一大优势。删除算法与 Evaluate 算法类似，也比较直接，即对需要删除的元素 $x$ 计算出 $h_1(x)$ 和 $h_2(x)$ 两个位置之后，如果这两个位置上有一个包含 $x$ 的指纹，则直接移除该位置上的指纹信息。注意在删除过程中，如果找到两个位置上都存在 $x$ 的指纹时，只需要移除其中一个位置上的指纹信息即可。这是因为当两个元素具有相同的指纹信息时，这样做就不会影响对另一个元素的存在性判断。当然，只删除一个会带来假阳性的问题，但这对于大部分过滤器结构来说都是无法避免的，我们只需要将假阳性率控制在较小的值即可。

### 2.1.3　布谷鸟过滤器的性能

我们用 $|f|$ 表示指纹值的比特长度，当给定 $h_1(x)$ 的值时，也就确定了 $h_2(x)$ 有 $2^{|f|}$ 种不同取值。假设布谷鸟过滤器中包含 $m$ 个桶，当 $2^{|f|} < m$ 时，$h_2(x)$ 的

取值范围也就是整个过滤器长度的子集。因此，当 $|f|$ 取值越小时，哈希碰撞的几率也会越大，构建过滤器的失败概率也会随之增大。而且当 $m$ 与 $2^{|f|}$ 之间的差距越大时，布谷鸟过滤器的空间利用率也会越低。如何设定合适的参数就显得尤为重要。

我们用负载因子 $\alpha$ （$0 \leqslant \alpha \geqslant 1$）来表示布谷鸟过滤器的空间利用率，它的定义是过滤器中已占用空间大小与过滤器大小的比值。因此当 $\alpha$ 的值越接近 1，那就表示空间利用率越高。在给定指纹长度 $|f|$ 和负载因子 $\alpha$ 的情况下，对于每个元素均摊的空间开销 $C$ 可以表示为：

$$C = \frac{\text{过滤器的存储大小}}{\text{存储的条目数}} = \frac{|f| \cdot \text{总条目数}}{\alpha \cdot \text{总条目数}} = \frac{|f|}{\alpha} \text{ bits.} \tag{2.3}$$

负载因子的大小受到桶大小（用 $b$ 表示）的影响。当 $b = 1$ 时，负载因子仅有 50%，而当 $b = 4$ 或 $b = 8$ 时，负载因子随之增长为 95% 和 98%。而当桶越大时，就越容易出现碰撞（即相同指纹信息）的情况。为了保证相同的假阳性率，就要求指纹长度越长。根据文献[3]中的推导，指纹长度 $|f|$ 与假阳性率 $\epsilon$ 和桶大小 $b$ 之间的关系如下所示：

$$|f| \geqslant \lceil \log_2(2b/\epsilon) \rceil = \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil \text{ bits.} \tag{2.4}$$

从式 2.3和式 2.4可以得出：

$$C \leqslant \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil / \alpha. \tag{2.5}$$

当 $b = 4$ 时，此时均摊空间开销约为 $(\log_2(1/\epsilon) + 3)/\alpha$，其中 $\alpha \approx 95\%$。而对于布隆过滤器，其均摊空间开销约为 $1.44 \log_2(1/\epsilon)$。因此，相比于布隆过滤器，布谷鸟过滤器可以实现更优的均摊空间开销。文献[3]中的实验结果表示，当 $b$ 取 4 的时候，布谷鸟过滤器能在假阳性率和空间开销之间取得较好的平衡。

## 2.2　异或型过滤器

Bloomier 过滤器[5] 是首个异或型结构的过滤器。在 Bloomier 过滤器中，不存在

### 2.2.1　异或过滤器

### 2.2.2　二进制引信过滤器

## 2.3　不经意的键值存储

### 2.3.1　RB-OKVS

# 第 3 章　在隐私保护上的应用

## 3.1　在可搜索加密方面的应用

## 3.2　在隐私信息检索方面的应用

## 3.3　在隐私集合计算方面的应用

# 参 考 文 献

[1] LUO L, GUO D, MA R T B, et al. Optimizing Bloom filter: Challenges, solutions, and comparisons[J/OL]. IEEE Communications Surveys & Tutorials, 2019, 21(2): 1912-1949. DOI: 10.1109/COMST.2018.2889329.

[2] DILLINGER P C, WALZER S. Ribbon filter: Practically smaller than Bloom and Xor: arXiv:2103.02515[M/OL]. arXiv, 2021. DOI: 10.48550/arXiv.2103.02515.

[3] FAN B, ANDERSEN D G, KAMINSKY M, et al. Cuckoo filter: Practically better than bloom [C/OL]//Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CONEXT). ACM, 2014: 75-88. DOI: 10.1145/2674005.2674994.

[4] PAGH R, RODLER F F. Cuckoo hashing[J/OL]. Journal of Algorithms, 2004, 51(2): 122-144. DOI: 10.1016/j.jalgor.2003.12.002.

[5] CHAZELLE B, KILIAN J, RUBINFELD R, et al. The Bloomier filter: An efficient data structure for static support lookup tables[C]//Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms. SIAM, 2004: 30-39.