大数据存储系统与管理报告

姓　名：　巢新科

学　院：　计算机科学与技术学院

专　业：　计算机科学与技术

班　级：　CS2104

学　号：　U202111248

指导教师：　施展

分数	
教师签名	

2024 年 4 月 21 日

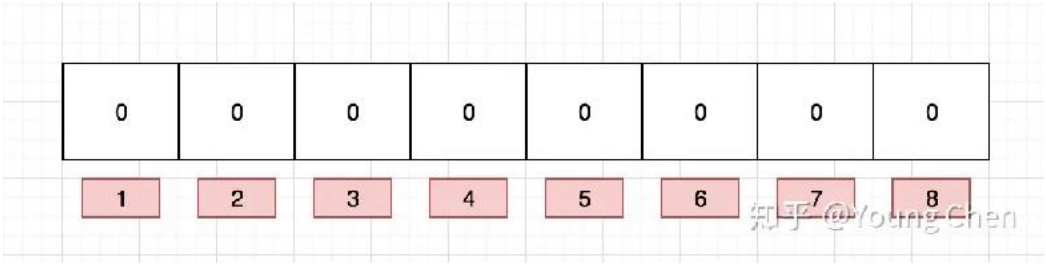
目 录

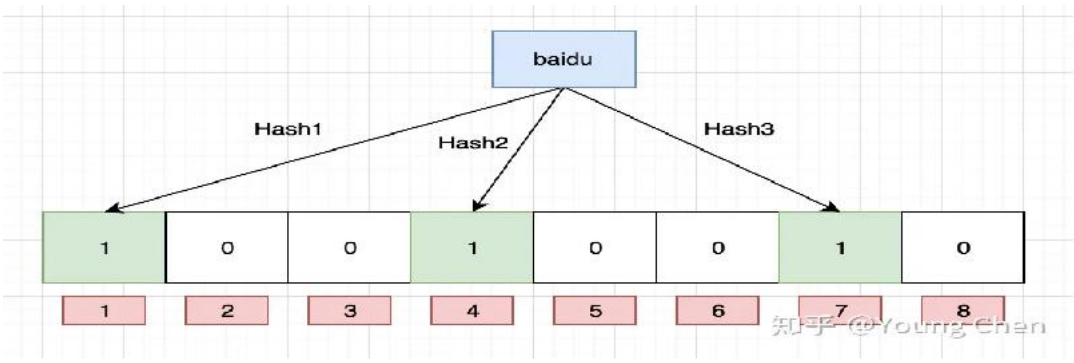
1 数据结构的设计

1.1 bloom filter 概述

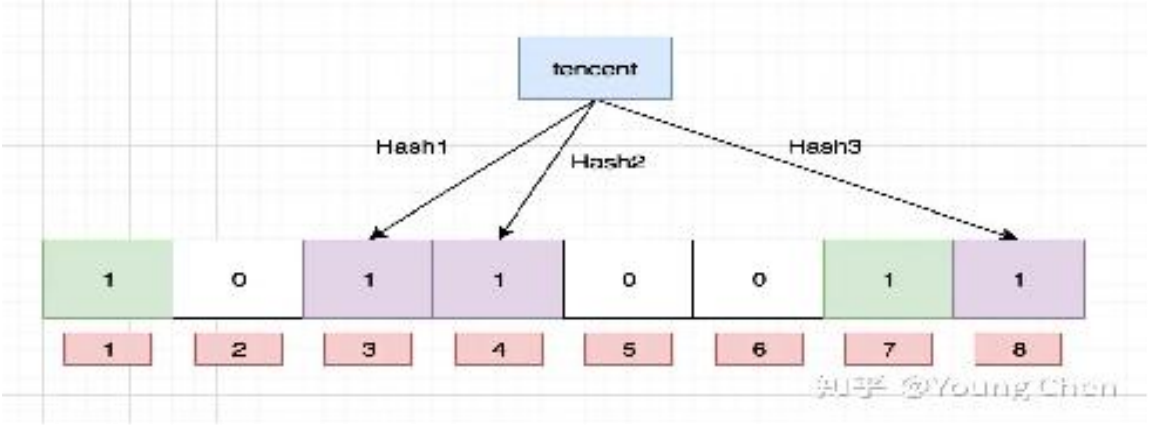
传统的 List、Set、Map 等数据结构，返回的结果是确定的，但是占用空间较大，布隆过滤器可以视为一种概率型数据结构，特点是高效地插入和查询且占用空间少，可以用来判断 “某样东西一定不存在或者可能存在”。

布隆过滤器一般是一个位数组，数组每一位是一个 bit，只有 0 或者 1 两个值，再加上多个不同的 hash 函数，当我们传入一个数据时，分别带入多个 hash 函数将 bitset 相应位置值设置为 1。一个示例如下(取自知乎 https://zhuanlan.zhihu.com/p/43263751)：





为方便演示，我们再插入一个数据：



现在整个位数组有部分位置的值已经被设置为 1，那现在我们想查询“baidu”这个值是否存在，首先获取多个 hash 函数的返回值，利用这些返回值判断位数组中相应位置的值是否为 1，显然是满足的，那么我们说“baidu”这个词可能存在于我整个数据集中。

为什么是可能？一般而言使用 bloom filter 时我们会现将数据集都传入进去将 bloom filter 初始化，然后我们再拿着要判断的数据进行比对，就拿上面的例子而言，假如我要搜索的数据是“bloom”，三个 hash 函数的返回值分别是 1、3、4，那么对照位数组发现这个值是存在的，但是实际并不存在，因此我们说 bloom filter 是会出现误判的也就是有可能出现 false positive。与之类似，我们还可以看到搜索数据集存在的数据一定是能通过 bloom filter 的检查的，因此 bloom filter 不存在 false negative，它判断某个值不存在，那么该值就一定不存在，这也是精华所在。

1.2 单个 bloom filter 数据结构设计

按照 1.1 所说，bloom filter 有这两个核心结构，一个是 bitset 位数组，另一个是 hash 函数族，也就是多个 hash 函数，因此可以设计如下简单的 bloom filter 结构：

```
1 implementation
pub struct BloomFilter {
    bitset: Vec<bool>,
    hash_functions: Vec<Box<dyn Fn(usize) -> usize>>,
}

impl BloomFilter {
    // size代表位数组大小，hash_count代表hash函数数量
    pub fn new(size: usize, hash_count: usize) -> Self {
        let mut hash_functions: Vec<Box<dyn Fn(usize) -> …>> = Vec::with_capacity(hash_count);
        for i: usize in 0..hash_count {
            // 获取当前时间作为种子
            let now: SystemTime = SystemTime::now();
            let since_the_epoch: Duration = now.duration_since(earlier: UNIX_EPOCH).expect(msg: "Ti
            let seed: usize = since_the_epoch.as_millis() as usize;

            let mut rng: ThreadRng = thread_rng();

            let up: usize = rng.gen_range(1111..9999);

            hash_functions.push(create_hash_function(seed: seed + up * i));
        }
        BloomFilter {
            bitset: vec![false; size],
            hash_functions,
        }
    }
```

new 函数初始化一个 bloom filter，传入的参数分别是位数组长度 size 和哈希函数个数 hash_count。

接着我们可以看到两个核心函数 insert 和 contains:

```rust
pub fn insert(&mut self, item: usize) {
    for hasher: &mut Box<dyn Fn(usize) -> …> in &mut self.hash_functions {
        let hash: usize = hasher(item);
        let index: usize = hash % self.bitset.len();
        self.bitset[index] = true;
    }
}

pub fn contains(&mut self, item: usize) -> bool {
    for hasher: &mut Box<dyn Fn(usize) -> …> in &mut self.hash_functions {
        let hash: usize = hasher(item);
        let index: usize = hash % self.bitset.len();
        if !self.bitset[index] {
            return false;
        }
    }
    true
}
```

两个函数都很简单，功能就是 1.1 中所描述的那样插入值和判断值是否存在。

1.3 多维 bloom filter 数据结构设计

1.2 所示的 bloom 过滤器一般来讲是一维的，也就是我们插入的值是单个的数值或者字符串等等，但是如果我所存储的数据是多维的呢？比如是一些高维向量例如(1, 2, 3 ,5 ,7 ,7 ,8 ,9)，那么单个的 bloom filter 就不好使了，这驱使我们创建一个多维 bloom filter 来解决问题。

一个很直接的想法是根据数据的维度大小来创建相应数量的 bloom filter，把它们联合在一起，如下所示：

```rust
1 implementation
pub struct MultidimensionalBloomFilter {
    bloom_filters: Vec<BloomFilter>,
```

insert 操作和 contains 操作相比于单个 bloom filter 是多出一层循环，我需要插入每个维度的数据，同时在比对的时候我也需要比对每一个维度的数据：

```rust
pub fn insert(&mut self, items: Vec<usize>) -> bool {
    if items.len() != self.bloom_filters.len() {
        return false;
    }

    for i: usize in 0..self.bloom_filters.len() {
        self.bloom_filters[i].insert(item: items[i]);
    }
```

```rust
pub fn contains(&mut self, items: Vec<usize>, flag: bool) -> bool {
    if items.len() != self.bloom_filters.len() {
        return false;
    }

    for (index: usize, bf: &mut BloomFilter) in self.bloom_filters.iter_mut().enumerate() {
        if !bf.contains(item: items[index]) {
            return false;
        }
    }

    true
```

那么这样就完成了吗？实际上并没有。思考：一个向量为向量，不是仅仅因为它有多个值，而是这多个值组成了多个维度，也就是说，一个完整向量各个值间应当有关系，而不是相互独立。回到代码，无论是 insert 还是 contains 我们只是对比单个维度的值，但是忽略了所有维度的值组成一个向量这个关系，也就是说，我们还忘了对这层关系做 bloom filter。那么缺失这一步后果是什么呢？答案显而易见，就是误判率很高，因为缺少向量这一层关系，导致我们对很多值相同但是向量形式不同的向量敞开了大门。

1.4 多维 bloom filter 数据结构优化

为了避免多维 bloom filter 带来的较高的误判率，现在在多维 bloom filter 数据结构上再加一个联合 bloom filter，其作用是保存一个向量各个维度间的关系：

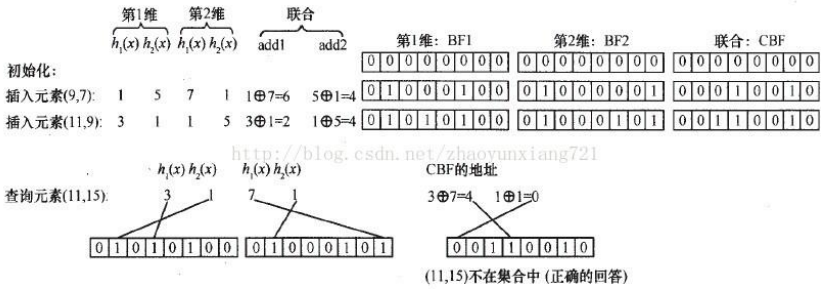
```rust
1 implementation
pub struct MultidimensionalBloomFilter {
    bloom_filters: Vec<BloomFilter>,
    union_bloom_filter: Vec<bool>,
    hash_num: usize,
}
```

注意到，联合 bloom filter 只是一个位数组，不含有其他的 hash 函数，其原理在于联合 bloom filter 维护的只是关系，所需的 hash 值已经在 bloom_filters 里得到。在 insert 和 contains 函数中，我们添加有关 union_bloom_filter 的处理逻辑：

```rust
// flag表示是否查找联合布隆过滤器
pub fn contains(&mut self, items: Vec<usize>, flag: bool) -> bool {
    if items.len() != self.bloom_filters.len() {
        return false;
    }

    for (index: usize, bf: &mut BloomFilter) in self.bloom_filters.iter_mut().enumerate() {
        if !bf.contains(item: items[index]) {
            return false;
        }
    }

    if flag {
        for i: usize in 0..self.hash_num {
            let mut res: usize = 0;

            for (index: usize, bt: &mut BloomFilter) in self.bloom_filters.iter_mut().enumerate() {
                res = res ^ (bt.get_x_hash_res(x: i, item: items[index]));
            }

            if !self.union_bloom_filter[res % self.union_bloom_filter.len()] {
                return false;
            }
        }
    }

    true
} fn contains
```

```rust
pub fn insert(&mut self, items: Vec<usize>) -> bool {
    if items.len() != self.bloom_filters.len() {
        return false;
    }

    for i: usize in 0..self.bloom_filters.len() {
        self.bloom_filters[i].insert(item: items[i]);
    }

    // 联合布隆过滤器
    for i: usize in 0..self.hash_num {
        let mut res: usize = 0;

        for (index: usize, bt: &mut BloomFilter) in self.bloom_filters.iter_mut().enumerate() {
            res = res ^ (bt.get_x_hash_res(x: i, item: items[index]));
        }

        let len: usize = self.union_bloom_filter.len();
        self.union_bloom_filter[res % len] = true;
    }

    true
}
```

可以看到，我选择了将一个向量每个维度的第一个 hash 值的异或和作为联合 bloom filter 的第一个 hash 值并在 bitset 中设置相应位置值为 1，以此类推。以下是一个示例图片：



这样我就得到了一个有关向量本身多个维度关系的布隆过滤器，在判断一个值是否在多维 bloom filter 时，我先在多个 bloom filter 判断，如果回答是肯定的，我再去联合 bloom filter 中查找，如果回答依旧是肯定的那么结论就是这个值在集合中，否则上述任何一步出错都给出否定答案。

2 流程分析

2.1 插入流程分析

优化后的多维 bloom filter 插入一个向量分为两步，一步是将向量各个维度值分开，以此加入到 bloom filters 相应的 bloom filter 中，第二步是我们取出各个维度在 bloom filter 的 hash 值，将它们逐个进行亦或和作为联合 bloom filter 的 hash 值，具体代码如下：

```rust
pub fn insert(&mut self, items: Vec<usize>) -> bool {
    if items.len() != self.bloom_filters.len() {
        return false;
    }

    for i: usize in 0..self.bloom_filters.len() {
        self.bloom_filters[i].insert(item: items[i]);
    }

    // 联合布隆过滤器
    for i: usize in 0..self.hash_num {
        let mut res: usize = 0;

        for (index: usize, bt: &mut BloomFilter) in self.bloom_filters.iter_mut().enumerate() {
            res = res ^ (bt.get_x_hash_res(x: i, item: items[index]));
        }

        let len: usize = self.union_bloom_filter.len();
        self.union_bloom_filter[res % len] = true;
    }

    true
}
```

2.2 查询流程分析

优化后的多维 bloom filter 查询流程也分为两步，第一步将向量各个维度值分开，分别到 bloom filters 相应的 bloom filter 中进行判断，如果任何一个 bloom filter 返回不存在，那么这个值就不存在，否则，进入第二步，同样取出各个维度在 bloom filter 的 hash 值，将它们逐个进行亦或和，判断联合 bloom filter 中相应位置值是否为 1，如果不为 1 则原向量不存在，否则原向量存在。具体代码如下：

```rust
// flag表示是否查找联合布隆过滤器
pub fn contains(&mut self, items: Vec<usize>, flag: bool) -> bool {
    if items.len() != self.bloom_filters.len() {
        return false;
    }

    for (index: usize, bf: &mut BloomFilter) in self.bloom_filters.iter_mut().enumerate() {
        if !bf.contains(item: items[index]) {
            return false;
        }
    }

    if flag {
        for i: usize in 0..self.hash_num {
            let mut res: usize = 0;

            for (index: usize, bt: &mut BloomFilter) in self.bloom_filters.iter_mut().enumerate() {
                res = res ^ (bt.get_x_hash_res(x: i, item: items[index]));
            }

            if !self.union_bloom_filter[res % self.union_bloom_filter.len()] {
                return false;
            }
        }
    }
```

3 理论分析

3.1 false positive 和 false negative

在 1.1 概论上说到过，bloom filter 本身是不存在 false negative 的，因为一个数据不在 bloom filter 中那么它一定不在原数据集中，但是 bloom filter 存在 false positive，也就是一个数据在 bloom filter 中那么它可能存在于原数据集中。将一堆原本不存在的数据判断存在于原数据集中的比例称为误判率，接下来分析单个 bloom filter 和多维 bloom filter 的误判率。

3.2 单个 bloom filter 的误判率

d 过小的布隆过滤器随着插入元素增加，bit 位很快都被置为 1，那么查询任何元素时误判率就是增加。布隆过滤器的长度越大，其误报率就越小。此外，哈希函数个数越多，每个元素插入时置 1 的比特位越多，过滤效率越低；但如果哈希函数太少的话，误报率也会变高。

我们把哈希函数个数记为 k，布隆过滤器长度记为 m，插入元素个数记为 n，误报率为 p。那么可以给出如下公式:

$$m = -(n \ln p) / (\ln 2)^2$$

则：当 $k = (m/n) * \ln 2$ 时 p 最小

因此，我保证了单个 bloom filter 的误判率最低，体现在代码上是:

```rust
impl MultidimensionalBloomFilter {
    // dimension代表维度，m代表单个布隆过滤器数组长度，n代表预估的数据量
    pub fn new(dimension: usize, m: usize, n: usize) -> MultidimensionalBloomFilter {
        let k: usize = ((m as f64 / n as f64) * 0.693) as usize;

        let mut bloom_filers: Vec<BloomFilter> = Vec::with_capacity(dimension);
        for _ in 0..dimension {
            bloom_filers.push(BloomFilter::new(size: m, hash_count: k + 1));
        }

        MultidimensionalBloomFilter {
            bloom_filters: bloom_filers,
            union_bloom_filter: vec![false; m],
            hash_num: k + 1,
        }
    }
```

hash 函数个数由预设的位数组长度 m 和数据量 n 决定。

3.3 多维 bloom filter 误判率

在 3.1 中我尽量保证了单个 bloom filter 误判率较低，因此也保证了多维 bloom filter 误判率较低，但是在优化前多维 bloom filter 的误判率非常高，原因在于优化前的多维 bloom filter 未保存向量本身的维度的关系，仅仅依靠判断单个维度是否存在将误判率进行了放大，一个简单的例子是我们向多维 bloom filter 中存储两个向量(0, 0, 0, 0, 0, 0) 和 (1, 1, 1, 1, 1, 1)，那么任何一个由 0，1 组成的六维向量都会被误判断存在于多维 bloom filter 中，这是一个相当严重的错误，假设六维向量真全是由 0，1 组成，那么误判率将会是$(2^6 - 2) / 2^6$接近

100%，多维 bloom filter 将会完全失去功能。因此优化多维 bloom filter，增加存储向量本身关系的联合 bloom filter 是必须的，在代码中表示为在多维 bloom filter 中多增加一个联合 bloom filter 用以存储关系：

```
1 implementation
pub struct MultidimensionalBloomFilter {
    bloom_filters: Vec<BloomFilter>,
    union_bloom_filter: Vec<bool>,
    hash_num: usize,
}
```

4 误判率测试

在测试方面，我主要测试了优化前后，多维 bloom filter 的误判率，为方便起见，我设置测试参数综合考虑碰撞概率，如果说位数组长度过长，而 hash 函数较少，测试数据较少，会导致优化前后误判率相差不大，难以看到明显优化程度。最终我选择了设置每个 bloom filter 的 bitset 长度为 1000，单个数据表示为三维向量，每个维度值在 0-10 之间，因为数据规模的问题，误判次数也会相应高，插入数据总量为 150，测试数据量为 100，且保证这 250 个数据完全不同，总共测试 100 次。

进入到 bloom_filter 目录下，运行：

```
cargo test --package bloom_filter --test md_bloom_filter --test_multidimensional_bloom_filter --exact --nocapture > test.log
```

可以查看 log 中的执行情况：

可以看到，多维 bloom filter 在优化前误判率维持在 80%，而优化后的误判率维持在 5%及以下，优化效果非常明显。