# UpBit 位图索引的改进

## 《软件设计与开发实践》课程答辩

马玉坤

哈尔滨工业大学计算机科学与技术系

2017 年 6 月 15 日

# 位图索引

**位图索引 (Bitmap Index)**

使用位图来索引数据库信息的数据结构，广泛用于数据库系统中。

# 位图索引 (Cont'd)

| 姓名 | 性别 | 身高 |
|---|---|---|
| 小 A | 男 | 167 |
| 小 B | 女 | 165 |
| 小 C | 男 | 180 |

表 1: 一个普通的数据库中的表

- 对于“性别”这个属性来说，只有“男”、“女”两种取值。[1]

[1]根据性染色体判断生理性别，不考虑性染色体异常情况。

# 位图索引 (Cont'd)

| 姓名 | 性别 | 身高 |
|---|---|---|
| 小 A | 男 | 167 |
| 小 B | 女 | 165 |
| 小 C | 男 | 180 |

表 1: 一个普通的数据库中的表

- 对于“性别”这个属性来说，只有“男”、“女”两种取值。[1]
- 于是，对于”男“、”女“两种取值，我们可以分别获得两个位向量，分别是 101 和 010。

[1]根据性染色体判断生理性别，不考虑性染色体异常情况。

# 位图索引 (Cont'd)

| 姓名 | 性别 | 身高 |
|---|---|---|
| 小 A | 男 | 167 |
| 小 B | 女 | 165 |
| 小 C | 男 | 180 |

表 1: 一个普通的数据库中的表

- 对于“性别”这个属性来说，只有“男”、“女”两种取值。[1]
- 于是，对于”男“、”女“两种取值，我们可以分别获得两个位向量，分别是 101 和 010。
- 对不同值对应的位向量进行位操作（与、或、异或），可以进行各种各样的复杂的数据库查询。

[1]根据性染色体判断生理性别，不考虑性染色体异常情况。

# 位图索引的更新问题

## 原地更新

如果使用传统的原地更新 (In-place)，对于每次修改（例如修改某个人的身高），最坏情况下将需要修改整个位向量，效率较低。[a]

[a] 只考虑使用压缩算法对位图索引进行压缩的情形。

# UpBit

UpBit 对于更新操作的效率极高，这归功于 UpBit 的两大利器。

# UpBit

UpBit 对于更新操作的效率极高，这归功于 UpBit 的两大利器。

## Update BitVectors

由于压缩算法的存在，如果位向量比较简单（例如 1 的个数极少），那么修改操作效率就较高。
因此对于每个位向量，定义一个辅助位向量，扮演缓存的角色，用以保存更新内容。当辅助向量较为复杂后，将辅助向量与位向量进行合并。

# UpBit

UpBit 对于更新操作的效率极高，这归功于 UpBit 的两大利器。

## Update BitVectors

由于压缩算法的存在，如果位向量比较简单（例如 1 的个数极少），那么修改操作效率就较高。
因此对于每个位向量，定义一个辅助位向量，扮演缓存的角色，用以保存更新内容。当辅助向量较为复杂后，将辅助向量与位向量进行合并。

## Fence Pointers

将每个未压缩的位向量进行分块，把每个块的起始位置对应的压缩后的位置保存到一个指针数组中。这些指针就是 Fence Pointers。

# UpBit (Cont'd)

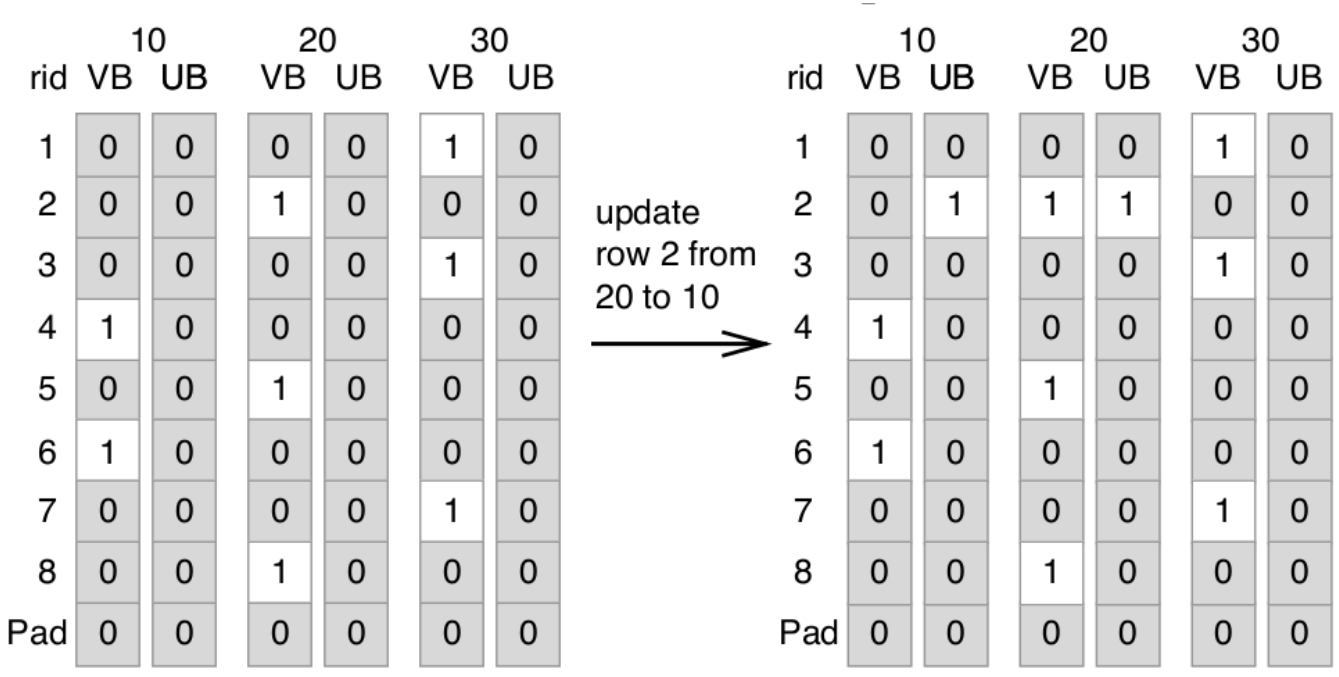


图 1: UpBit 更新操作

# 时间效率

## 对行查询

- 更新第 k 行的值，需要找到第 k 行修改之前的值。

**get_value (index: *UpBit*, row: $k$)**

1: **for** each i $\in \{1,2,...,d\}$ **do**
2: $temp\_bit = V_i.get\_bit(k) \oplus U_i.get\_bit(k)$
3: **if** $temp\_bit$ **then**
4: Return $val_i$
5: **end if**
6: **end for**

图 2: UpBit 对行查询

# 时间效率

## 对行查询

- 更新第 k 行的值，需要找到第 k 行修改之前的值。
- 然而，如图2，寻找第 k 行的值最坏情况下要遍历所有的位向量。

**get_value (index: *UpBit*, row: $k$)**

1: **for** each i $\in \{1,2,...,d\}$ **do**
2: $temp\_bit = V_i.get\_bit(k) \oplus U_i.get\_bit(k)$
3: **if** *temp_bit* **then**
4: Return $val_i$
5: **end if**
6: **end for**

图 2: UpBit 对行查询

# 时间效率

## 对行查询

- 更新第 k 行的值，需要找到第 k 行修改之前的值。
- 然而，如图2，寻找第 k 行的值最坏情况下要遍历所有的位向量。
- 实际上，作者实验证明，在基数 (**distinct cardinality**) 等于 1000 时，get_value 函数耗时占 update 总耗时的 93%。

**get_value (index:** *UpBit*, **row:** $k$**)**

```
1: for each i ∈ {1,2,...,d} do
2:     temp_bit = V_i.get_bit(k) ⊕ U_i.get_bit(k)
3:     if temp_bit then
4:         Return val_i
5:     end if
6: end for
```

图 2: UpBit 对行查询

# 时间效率 (Cont'd)

## 范围查询

- 在对数据库查询（例如使用 SQL 语句）时，我们经常会用到范围查询，比如`SELECT * FROM Persons WHERE Year>1965`。

# 时间效率 (Cont'd)

## 范围查询

- 在对数据库查询（例如使用 SQL 语句）时，我们经常会用到范围查询，比如`SELECT * FROM Persons WHERE Year>1965`。
- 然而，直接使用 UpBit 进行范围查询的效率是较低的。最坏情况下需要遍历所有的位向量。

# 空间效率

## 对 Update BitVectors 的正确认识

- 实际上，Update BitVectors 在 UpBit 中起了缓存的作用

# 空间效率

## 对 Update BitVectors 的正确认识

- 实际上，Update BitVectors 在 UpBit 中起了缓存的作用
- 在计算机中，缓存的大小是远远小于主存的大小的。

# 空间效率

## 对 Update BitVectors 的正确认识

- 实际上，Update BitVectors 在 UpBit 中起了缓存的作用
- 在计算机中，缓存的大小是远远小于主存的大小的。
- 类比计算机系统中的缓存，实际上不需要在内存中为每个位向量都维护一个 Update BitVector。

# 空间效率

## 对 Update BitVectors 的正确认识

- 实际上，Update BitVectors 在 UpBit 中起了缓存的作用
- 在计算机中，缓存的大小是远远小于主存的大小的。
- 类比计算机系统中的缓存，实际上不需要在内存中为每个位向量都维护一个 Update BitVector。
- 甚至可以使用计算机系统中的缓存的替换算法，来有效率地维护 Update BitVector。

# 空间效率

## 对 Update BitVectors 的正确认识

- 实际上，Update BitVectors 在 UpBit 中起了缓存的作用
- 在计算机中，缓存的大小是远远小于主存的大小的。
- 类比计算机系统中的缓存，实际上不需要在内存中为每个位向量都维护一个 Update BitVector。
- 甚至可以使用计算机系统中的缓存的替换算法，来有效率地维护 Update BitVector。
- 这样做不仅可以减小 Update BitVectors 内存的占用，还不会降低 UpBit 的性能。

# 对 UpBit 优化的方法

## 树状数组 (Fenwick Tree)

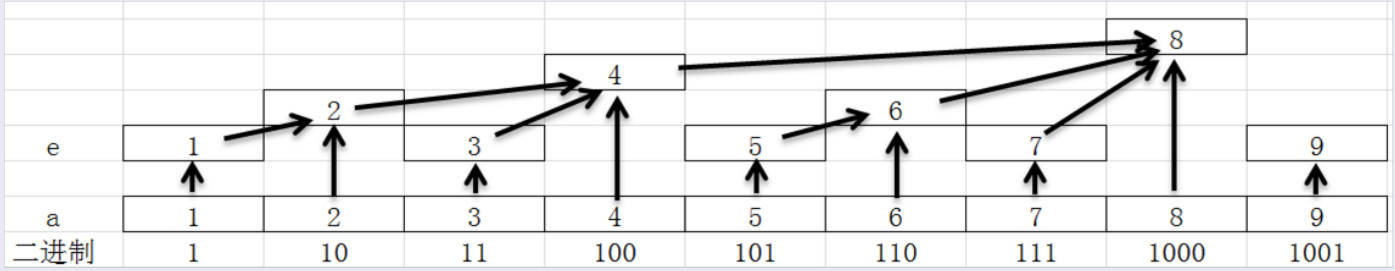


图 3: 树状数组 (**Fenwick Tree**)

如图3，每个节点对应一个位向量，该位向量为对其子节点的位向量进行或操作后的结果。

# 对 UpBit 优化的方法 (Cont'd)

## 使用树状数组作为 UpBit 的组织形式

- 对行查询：
  1. 找到最大的 i，使得前 i 个值的位向量或操作后第 k 位为 0。val[k] 即为 i+1。
  2. 从值的二进制表示中，由最高位到最低位依次确定。
- 范围查询：
  1. 树状数组求前缀和
  2. ```
     while (k > 0) {
       res |= val[k];
       k -= lowbit(k);
     }
     ```
     ```

# 对行查询的效率

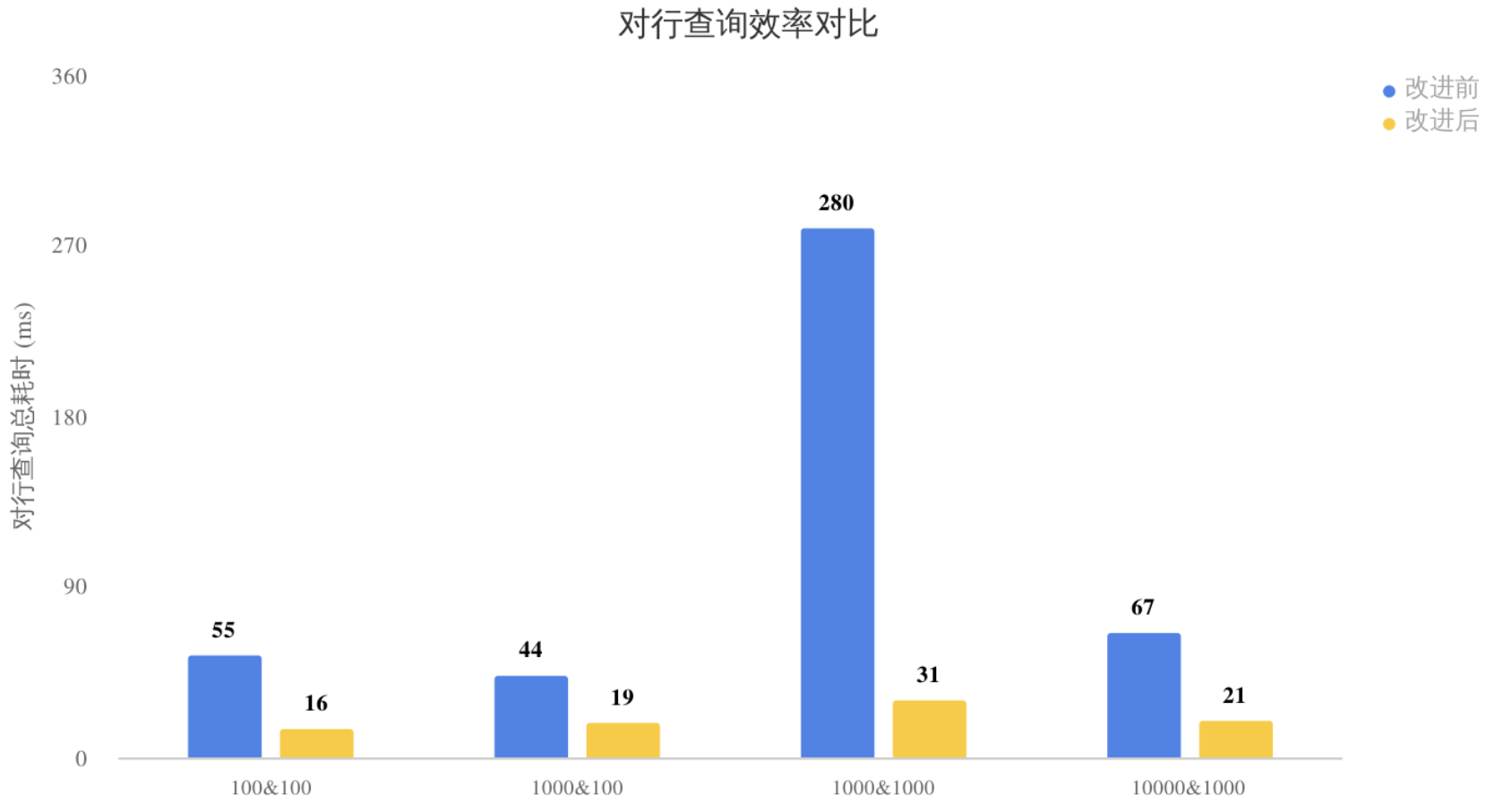


图 4: 对行查询效率比较图

# 范围查询的效率

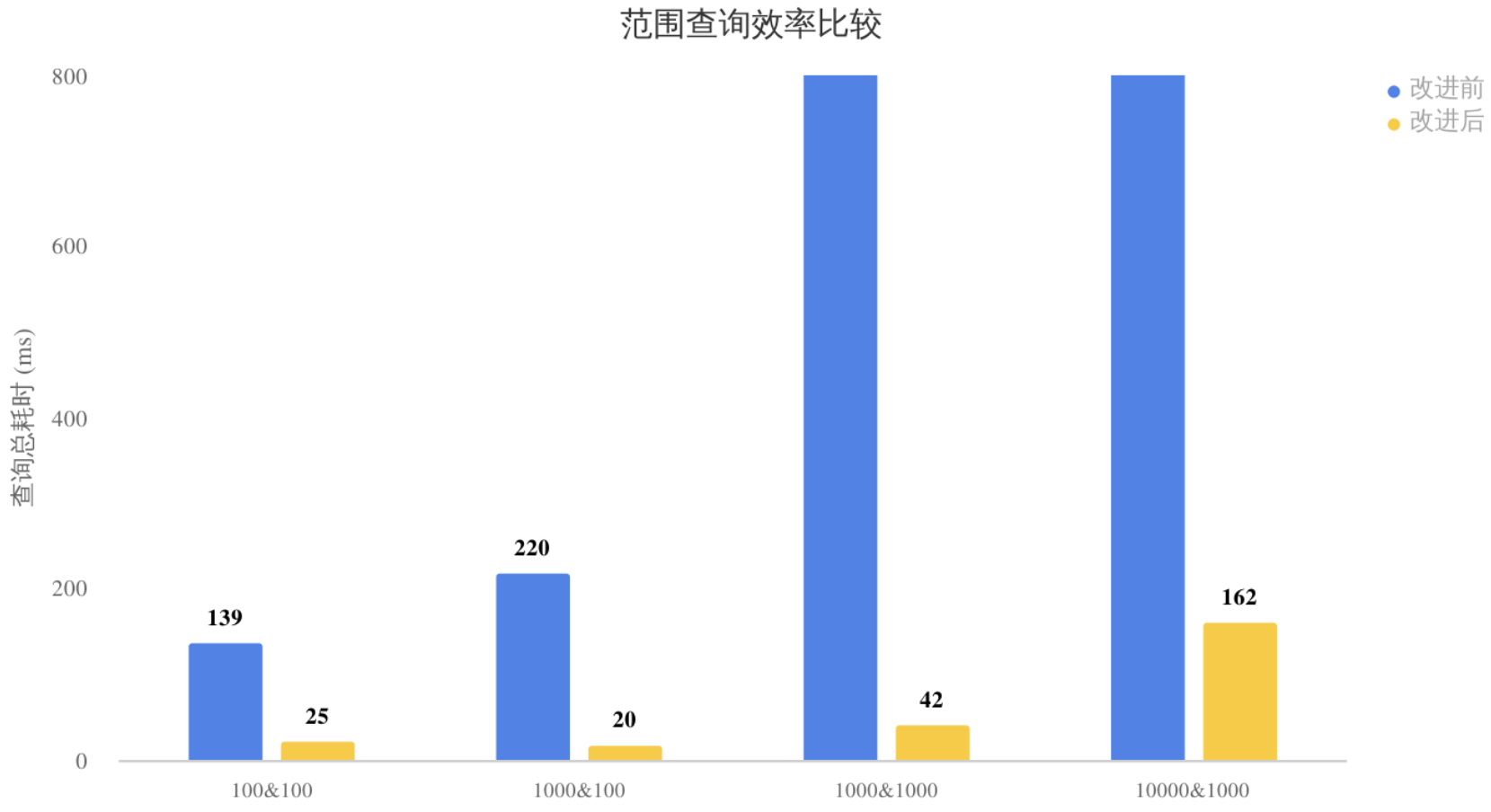


图 5: 范围查询效率比较图

# 单值修改的效率

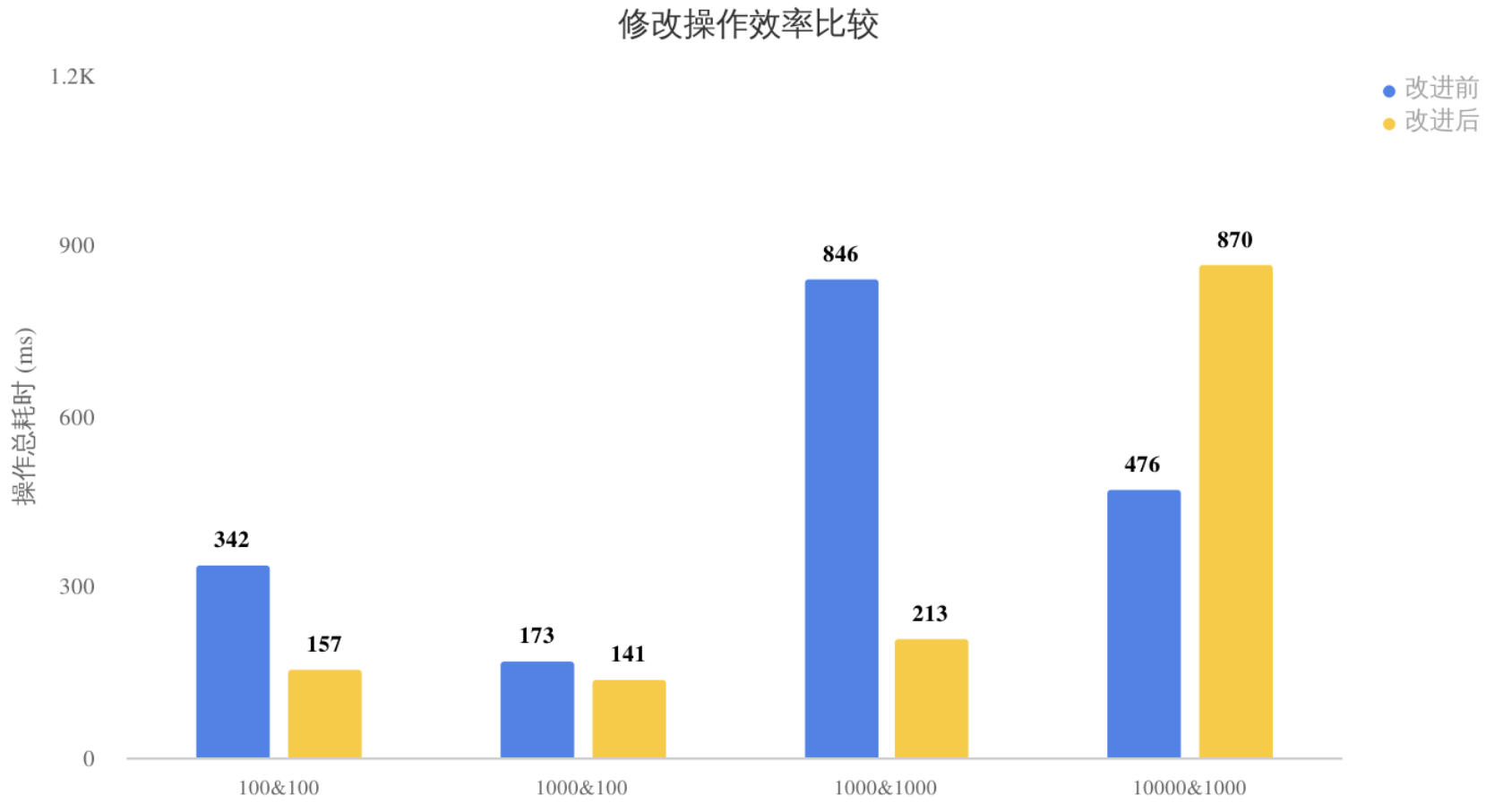


图 6: 修改操作效率比较图

# 结论

基于上述研究，我们可以得出以下结论:

- 改进后的 UpBit 对于范围查询效率更高

# 结论

基于上述研究，我们可以得出以下结论:

- 改进后的 UpBit 对于范围查询效率更高
- 改进后的 UpBit 在基数较大的情况下修改操作效率更高

# 结论

基于上述研究，我们可以得出以下结论:

- 改进后的 UpBit 对于范围查询效率更高
- 改进后的 UpBit 在基数较大的情况下修改操作效率更高
- 改进前的 UpBit 在基数较小的情况下修改操作效率较高

# 结论

基于上述研究，我们可以得出以下结论:

- 改进后的 UpBit 对于范围查询效率更高
- 改进后的 UpBit 在基数较大的情况下修改操作效率更高
- 改进前的 UpBit 在基数较小的情况下修改操作效率较高
- 如果将树状数组修改为其他平衡树（例如 Splay），将能够在取值集合未知的情况下，动态向属性的取值集合添加元素。（尽管效率可能下降。）

# 结论

基于上述研究，我们可以得出以下结论:

- 改进后的 UpBit 对于范围查询效率更高
- 改进后的 UpBit 在基数较大的情况下修改操作效率更高
- 改进前的 UpBit 在基数较小的情况下修改操作效率较高
- 如果将树状数组修改为其他平衡树（例如 Splay），将能够在取值集合未知的情况下，动态向属性的取值集合添加元素。（尽管效率可能下降。）
- 高效地利用 Update BitVector 将能在不影响效率的情况下减少其内存使用。