UpBit 的改进

《算法设计与分析》课程第四次作业

计算机科学与技术学院
马玉坤
1150310618

2017 年 6 月 21 日

1 对 UpBit 优化的动机

1.1 时间效率

1.1.1 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 1，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 $\in \{1,2,...,d\}$ **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 1: UpBit 对行查询

1.1.2 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。

1.2 空间效率

1.2.1 对 Update BitVectors 的正确认识

实际上，Update BitVectors 在 UpBit 中起了缓存的作用。而在计算机中，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。

2 对 UpBit 优化的方法

2.1 树状数组

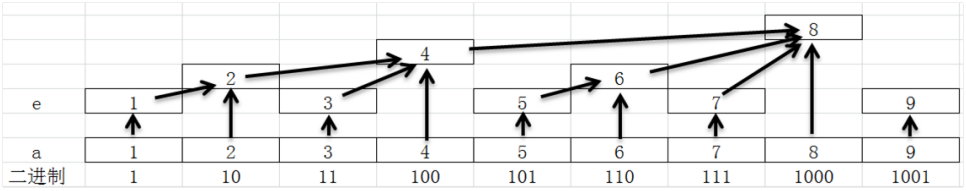

图 2: 树状数组 (**Fenwick Tree**)

如图 2，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。每个节点所管辖的区间长度恰好为**节点编号的二进制表示中最低的 1 的位置代表的 2 的幂**。例如 6 的二进制表示为 **0110**，6 号节点管辖的区间长度就是 2。

2.2 使用树状数组作为 UpBit 的组织形式

- 对行查询：

1. 我们要做的是：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。实际上 val[k]（第 k 行的值）即为 i+1。
2. 使用树状数组，我们可以从值的二进制表示中，由最高位到最低位依次确定。
3. ```
   int get_value(int row_id) {
     int col_id = 0;
     for (int i = 0; i <= log2(n); i++) {
       if (ub[col_id+(1<<i)][row_id] ^ vb[col_id+(1<<i)][row_id] == 0) {
         col_id += (1<<i);
       }
     }
     return col_id;
   }
   ```

- 范围查询：

1. 树状数组求前缀和
2. ```
   while (k > 0) {
     res |= val[k];
     k -= lowbit(k);
   }
   ```

3 实验结果

3.1 对行查询效率对比

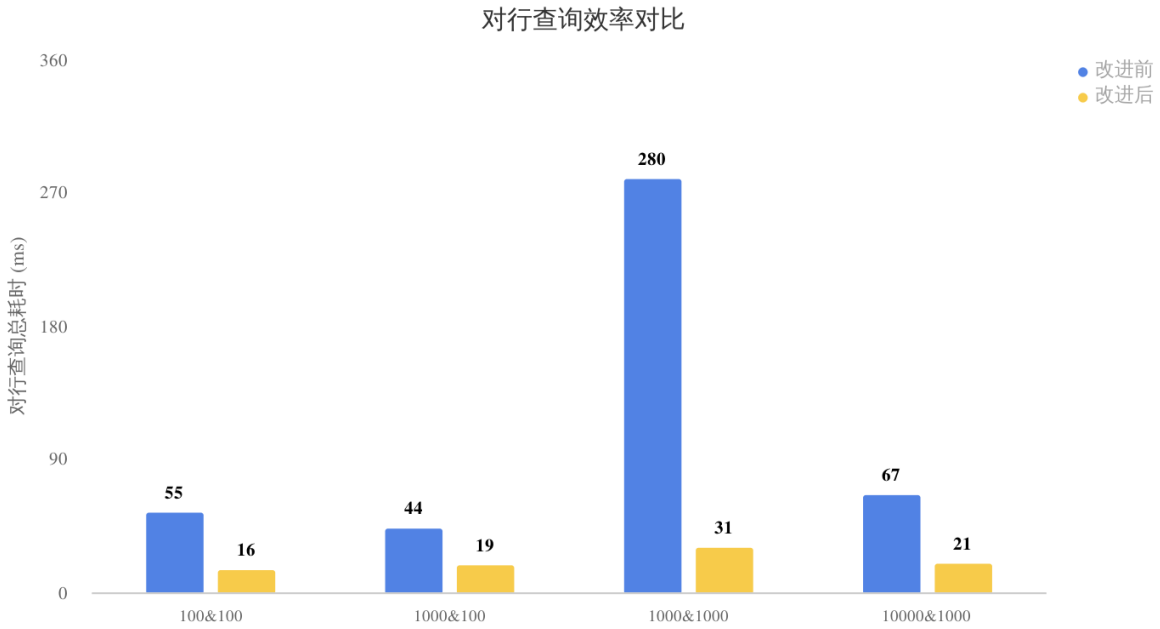

图 3: 对行查询效率比较图

3.2 范围查询效率对比

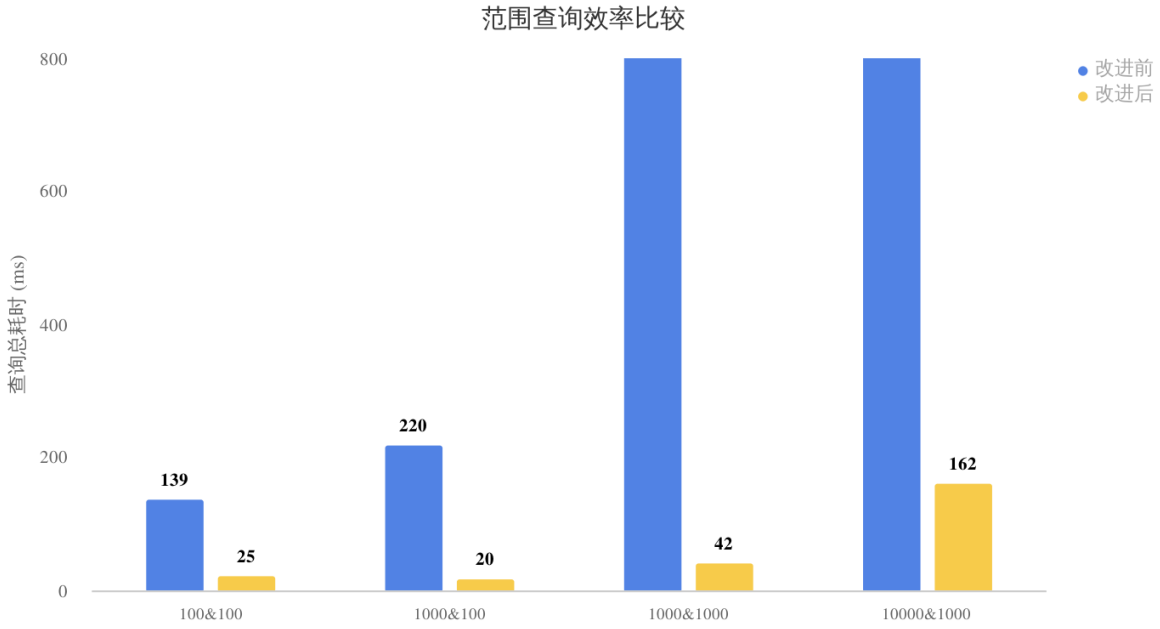

图 4: 范围查询效率比较图

3.3 单值修改效率对比

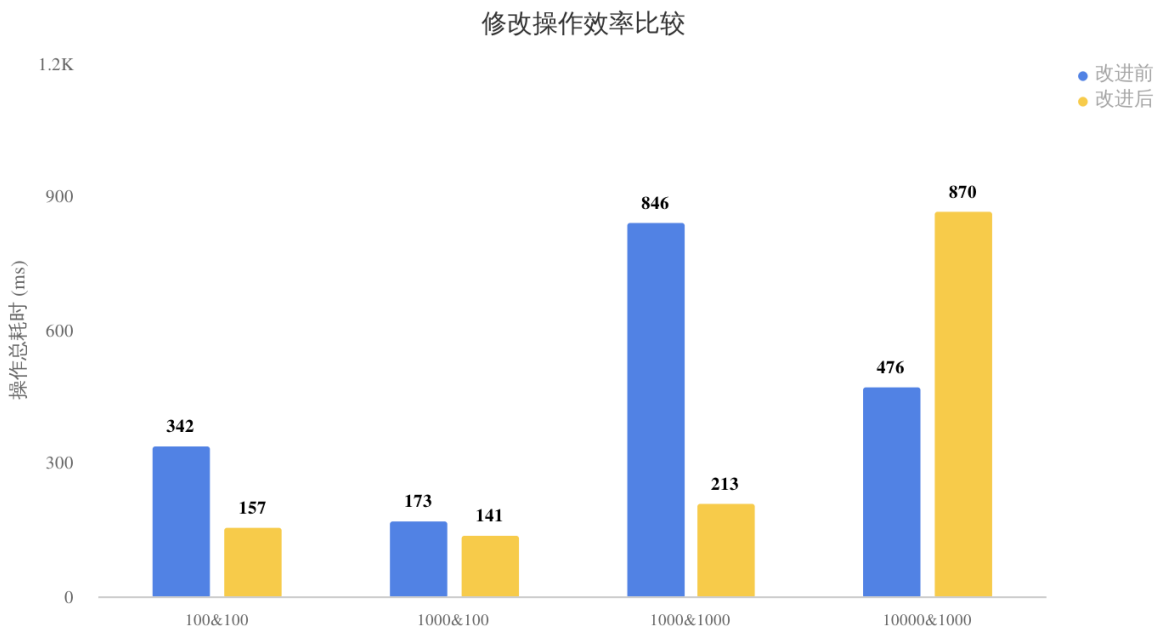

图 5: 修改操作效率比较图

4 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。