基数树 项目报告

李若彬 522031910747

2024年 4月 24日

1. 背景介绍

基数树是一种多叉搜索树，能够有效地存储和检索集合（`set`）或映射（`map`）的数据。基数树通过共享相同的前缀来节省空间，并提供快速的查找操作。由于该特点，基数树被广泛运用在IP路由、信息检索等领域中。在`Linux`中基数树被广泛应用于`IDR`机制、`pagecache`索引、路由查找等实现中。在本实验中要求实现一个存储`int32_t`（32位整型）类型集合（`Set`）的4叉基数树。在基数树中，同一子树下的节点具有共同的前缀（存储字符串时有相同的字符前缀；存储整型时有共同的整型的二进制前缀）。

2. 系统实现

2.1 基数树的节点实现

```
struct Node
    {
        Node *children[4];
        bool isLeaf;
        Node()
        {
            children[0] = nullptr;
            children[1] = nullptr;
            children[2] = nullptr;
            children[3] = nullptr;
            isLeaf = true;
        }
    };
```

每个节点有四个子节点以及一个布尔变量表明其是否是叶子节点

2.2 基数树的基本操作

```cpp
void RadixTree::insert(int32_t value)
{
    // To Be Implemented
    Node* current = root;
    for (int i = 30; i >= 0; i -= 2) {
        int index = (value >> i) & 0x3;  // 获取两个比特
        if (current->children[index] == nullptr) {
            current->children[index] = new Node();
        } else {
            current->isLeaf = false;  // 设置父节点为非叶子节点
        }
        current = current->children[index];
    }
    current->isLeaf = true;  // 设置新插入的节点为叶子节点
}
```

1. 创建一个指向根节点的指针，我们将使用这个指针来遍历树。
2. 从最高位开始，每次取出整数的两个比特。
3. 检查当前节点的子节点数组中对应的位置是否已经有节点。如果没有，就创建一个新的节点。
4. 将当前节点移动到子节点。
5. 重复步骤2-4，直到遍历完整数的所有比特。
6. 在最后一个节点上设置`isLeaf`为`true`，表示这是一个叶子节点

```cpp
void RadixTree::remove(int32_t value)
{
    // To Be Implemented
    Node* current = root;
    Node* nodes[16];  // 存储路径上的节点
    int indices[16];  // 存储路径上的索引
    int depth = 0;

    for (int i = 30; i >= 0; i -= 2) {
        int index = (value >> i) & 0x3;  // 获取两个比特
        if (current->children[index] == nullptr) {
            return;  // 整数不在树中
        }
        nodes[depth] = current;
        indices[depth] = index;
        depth++;
        current = current->children[index];
    }

    if (!current->isLeaf) {
        return;  // 整数不在树中
    }

    // 删除路径上的节点
    for (int i = depth - 1; i >= 0; --i) {
        delete nodes[i]->children[indices[i]];
        nodes[i]->children[indices[i]] = nullptr;
```

```
28
29          // 检查是否需要删除父节点
30          bool allNull = true;
31          for (int j = 0; j < 4; ++j) {
32              if (nodes[i]->children[j] != nullptr) {
33                  allNull = false;
34                  break;
35              }
36          }
37          if (!allNull) {
38              break;
39          }
40      }
41  }
```

1. 创建一个指向根节点的指针，我们将使用这个指针来遍历树。
2. 从最高位开始，每次取出整数的两个比特。
3. 检查当前节点的子节点数组中对应的位置是否已经有节点。如果没有，就直接返回，因为这意味着整数不在树中。
4. 将当前节点移动到子节点。
5. 重复步骤2-4，直到找到叶子节点。
6. 删除叶子节点，并将父节点的对应子节点指针设置为`nullptr`。
7. 如果父节点的所有子节点都是`nullptr`，则删除父节点，并将父节点的父节点的对应子节点指针设置为`nullptr`。重复这个步骤，直到遇到一个节点，它的子节点不全是`nullptr`。

```
1   bool RadixTree::find(int32_t value)
2   {
3       // To Be Implemented
4       Node* current = root;
5       for (int i = 30; i >= 0; i -= 2) {
6           int index = (value >> i) & 0x3;  // 获取两个比特
7           if (current->children[index] == nullptr) {
8               return false;  // 整数不在树中
9           }
10          current = current->children[index];
11      }
12      return current->isLeaf;
13  }
```

1. 创建一个指向根节点的指针，我们将使用这个指针来遍历树。
2. 从最高位开始，每次取出整数的两个比特。
3. 检查当前节点的子节点数组中对应的位置是否已经有节点。如果没有，就直接返回`false`，因为这意味着整数不在树中。
4. 将当前节点移动到子节点。
5. 重复步骤2-4，直到遍历完整数的所有比特。
6. 检查最后一个节点的`isLeaf`是否为`true`。如果是，就返回`true`，否则返回`false`。

```cpp
uint32_t size(Node* node)
    {
        if (node == nullptr) {
            return 0;
        }
        uint32_t count = 1;  // 计算当前节点
        for (int i = 0; i < 4; ++i) {
            count += size(node->children[i]);  // 递归计算子节点
        }
        return count;
    }
```

先检查传入的节点是否为`nullptr`。如果是，那么函数立即返回`0`。否则，函数会递归地调用`size`来计算所有子节点的数量，然后加上`1`（代表当前节点）并返回结果。

```cpp
uint32_t height(Node* node)
    {
        if (node == nullptr) {
            return 0;
        }
        uint32_t maxDepth = 0;
        for (int i = 0; i < 4; ++i) {
            maxDepth = std::max(maxDepth, height(node->children[i]));
        }
        return maxDepth + 1;  // 根节点有高度
    }
```

首先检查传入的节点是否为`nullptr`。如果是，那么函数立即返回0。否则，函数会递归地调用`height`来计算所有子节点的最大深度，然后加上`1`（代表当前节点的深度）并返回结果。

3. 测试

3.1 YCSB测试

3.1.1 测试配置

系统概述

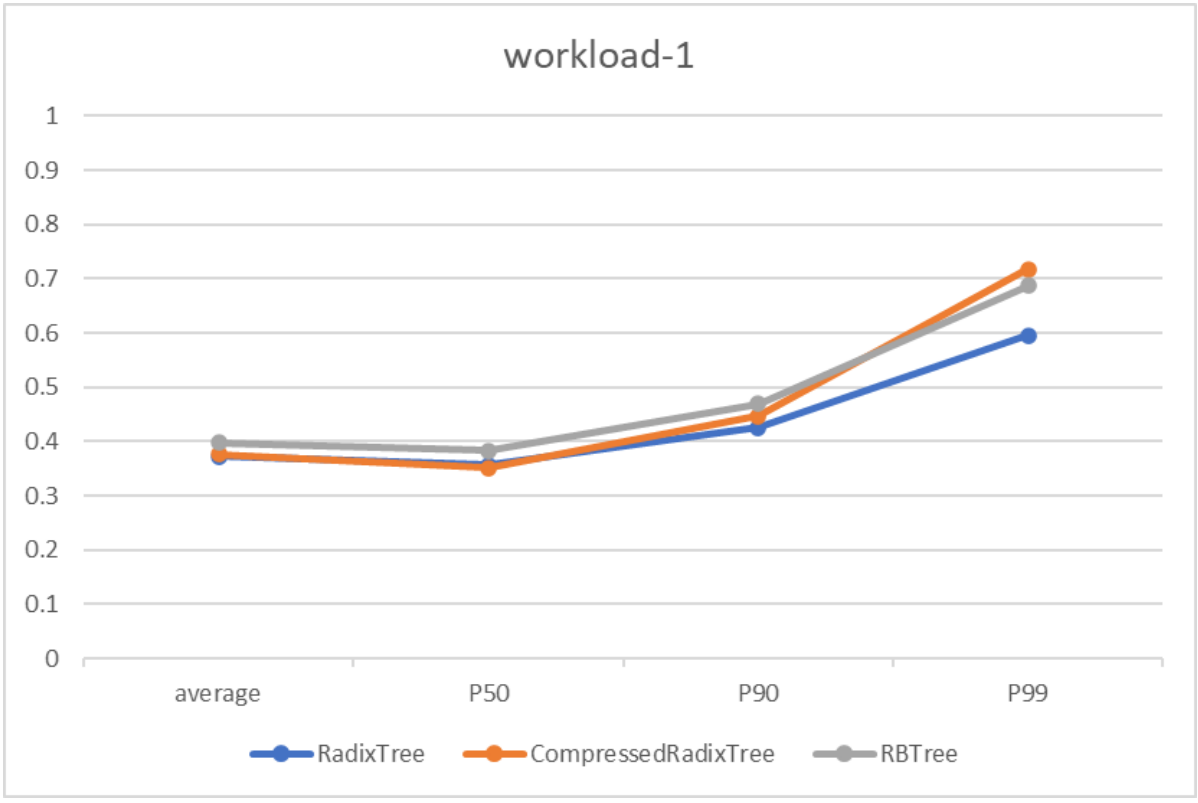
版本	22.03 LTS
	版权所有 © 2009-2021 麒麟软件 保留所有权利。
内核	linux 5.10.0-60.18.0.50.oe2203.x86_64
CPU	12thGenIntel(R)Core(TM)i7-12700H
内存	2 GB
桌面	UKUI
用户名	

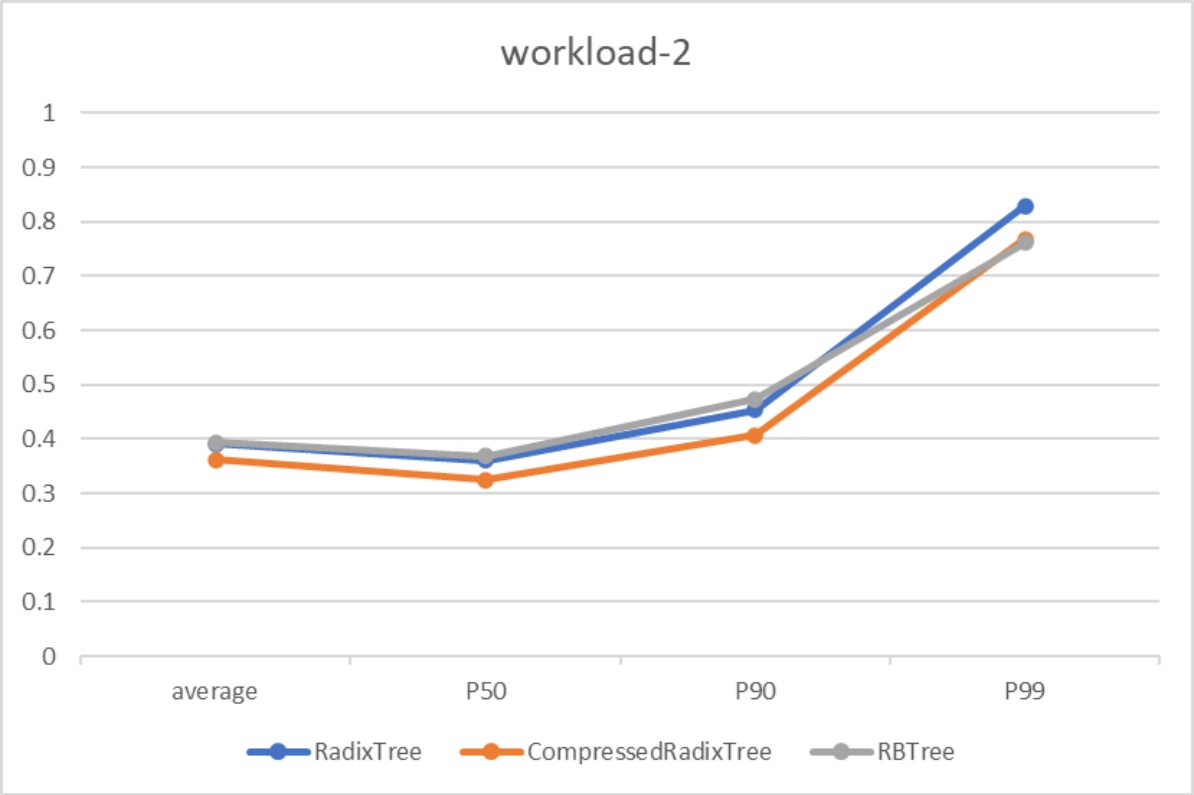
3.1.2 测试结果

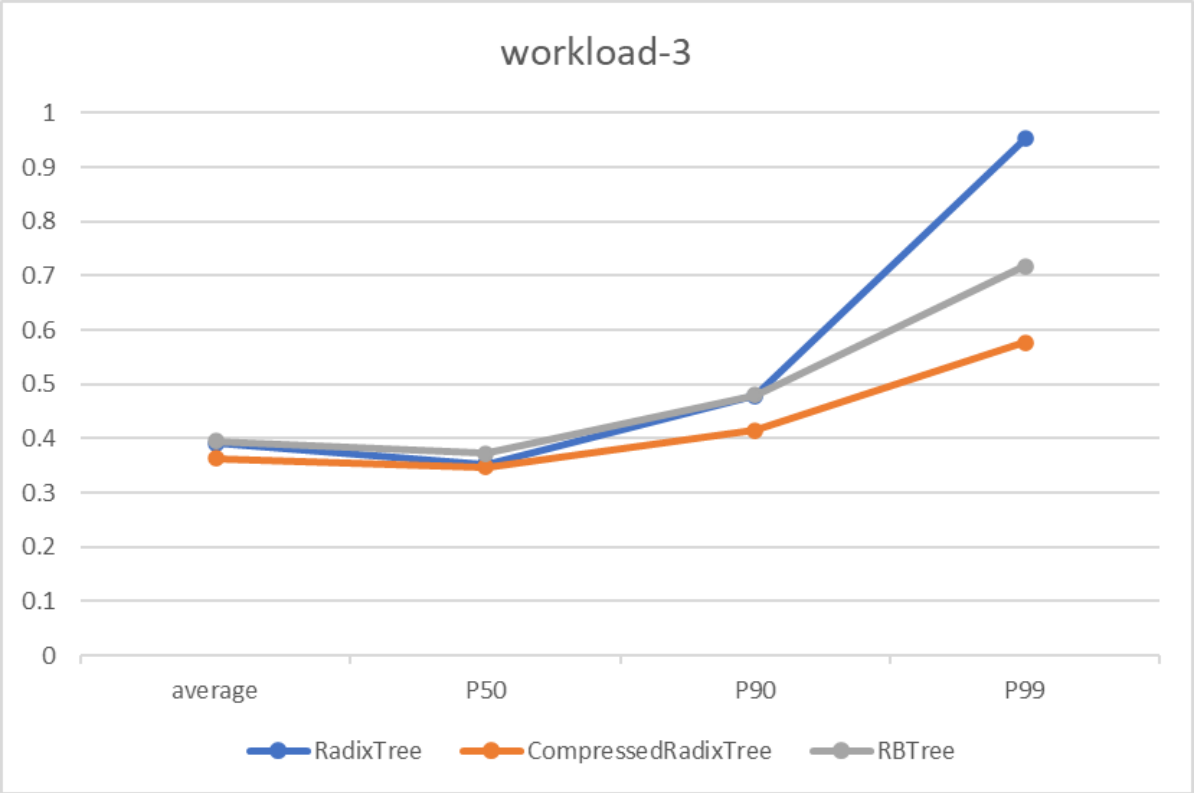
```
[lrb@localhost Lab1-handout]$ ./test
RadixTree
workload-1
average:0.372518 P50:0.356 P90:0.425 P99:0.595
workload-2
average:0.390624 P50:0.361 P90:0.454 P99:0.829
workload-3
average:0.390637 P50:0.352 P90:0.479 P99:0.953

CompressedRadixTree
workload-1
average:0.377146 P50:0.351 P90:0.447 P99:0.717
workload-2
average:0.362073 P50:0.325 P90:0.407 P99:0.768
workload-3
average:0.362804 P50:0.347 P90:0.415 P99:0.577

RBTree
workload-1
average:0.397566 P50:0.383 P90:0.47 P99:0.687
workload-2
average:0.393384 P50:0.368 P90:0.474 P99:0.761
workload-3
average:0.394875 P50:0.372 P90:0.48 P99:0.718
```







3.1.3 结果分析

- **平均时延比较**:
  - 在所有工作负载下，压缩优化后的基数树（`CompressedRadixTree`）的平均时延都略低于基数树（`RadixTree`），而红黑树（`RBTree`）的平均时延则稍高于基数树和压缩优化后的基数树。这可能是由于压缩优化后的基数树在内部节点上具有更高的空间效率，减少了访问延迟。
  - 在不同工作负载下，基数树的平均时延在0.372到0.390之间，压缩优化后的基数树在0.362到0.377之间，而红黑树在0.393到0.398之间。
- **P50、P90、P99时延比较**:

    - 对于P50、P90和P99时延，基数树和压缩优化后的基数树表现相对接近，而红黑树的时延则普遍高于基数树和压缩优化后的基数树。
    - 在不同工作负载下，基数树和压缩优化后的基数树的P50、P90和P99时延表现类似，而红黑树的P50、P90和P99时延普遍高于基数树和压缩优化后的基数树。
- **工作负载对性能的影响**：
    - 在不同的工作负载下，基数树、压缩优化后的基数树和红黑树的性能表现有所差异。例如，在`workload-2`下，执行`find`操作的比例为100%，基数树和压缩优化后的基数树的性能优于`workload-1`和`workload-3`，而红黑树的性能则相对较差。

4. 结论

- **压缩基数树（`CompressedRadixTree`）**在所有工作负载下的平均时延、P50、P90和P99时延上通常表现最好，显示出其对于查找、插入和删除操作的高效处理能力。这可能是由于压缩基数树通过节点压缩减少了树的深度，从而在遍历时减少了访问时间。
- **基数树（`RadixTree`）**在大多数情况下表现优于红黑树，但略逊于压缩基数树。这表明基数树在处理特定类型的数据集时效率较高，但通过进一步的优化（如节点压缩）可以提高性能。
- **红黑树（`RBTree`）**在所有测试中的平均时延、P50、P90和P99时延上通常是最高的。尽管红黑树在平衡性和通用性方面表现出色，但在这些特定工作负载下，它在性能上不如基数树和压缩基数树。

压缩优化后的基数树在大多数情况下表现略优于基数树和红黑树，而红黑树的性能相对较差。在不同的工作负载下，基数树、压缩优化后的基数树和红黑树的性能差异较大，因此在选择数据结构时应根据实际需求和工作负载来进行选择。