基数树项目报告模板

陈启炜 522031910299

2024年 4月 16日

1 背景介绍

基数树（英语：Radix Trie，也叫基数特里树或压缩前缀树）是一种数据结构，是一种更节省空间的Trie（前缀树）。每个内部节点的子节点数最多为基数树的基数r ，其中r为正整数，是2的x次方，x≥1，这使得基数树更适用于对于较小的集合（尤其是字符串很长的情况下）和有很长相同前缀的字符串集合 [1]。

本实验构建了`Radix Tree`的基本结构, `Compressed Radix Tree`的基本结构, 以及添加、删除、查找等基本操作。

2 系统实现

2.1 数据结构

```
  class RadixTreeNode /* the node of radix tree */
{
public:
    RadixTreeNode* children[4];  // the children of the node
    uint8_t bit;  // the bits of the node
    uint8_t height;  // the height of the node
    bool isLeaf;  // whether the node is a leaf
    // .. functions
};
  class CompressedRadixTreeNode /* the node of compressed radix tree */
{
  public:
    uint32_t bits;  // the bits of the node
    uint8_t startBit, endBit;  // if endBit == 32, the node is a leaf
    uint8_t height;  // the height of the node
    CompressedRadixTreeNode *parent; // the parent of the node
    // the children of the node
    CompressedRadixTreeNode *children[4];
```

```
    // .. functions
};
```

2.2 基本操作

2.2.1 查找操作:find

- Radix Tree的查找操作:
  从根节点开始，根据当前节点的bit值，选择对应的子节点，直到找到对应的节点或者到达叶子节点。
- Compressed Radix Tree的查找操作:
  定义函数findCompressedRadixTreeNode，从根节点开始，根据当前节点的bit值，选择对应的子节点，选择后根据当前节点的startBit和endBit，判断是否需要继续向下查找以及如何向下查找。直到找到对应的节点或者到达叶子节点。 find函数调用findCompressedRadixTreeNode函数，返回查找结果。(这个包装方便insert和remove操作)

2.2.2 插入操作:insert

- Radix Tree的插入操作:
  从根节点开始，根据当前节点的bit值，选择对应的子节点，直到找到对应的节点或者到达叶子节点，将新节点插入到对应的位置。如果插入的位置已经有节点，将其替换为新节点。（在有value的情况下，本实验无value）
- Compressed Radix Tree的插入操作:
  insert函数调用findCompressedRadixTreeNode函数，找到插入的位置，然后根据插入的位置，将新节点插入到对应的位置。插入时，判断是否需要分裂节点，如果需要分裂节点，进行分裂操作。如果无需分裂，将新节点插入到对应的位置。（不必拆成2-bit长度的节点，直接将后续长度的节点插入即可）

2.2.3 删除操作:remove

- Radix Tree的删除操作:
  从根节点开始，根据当前节点的bit值，选择对应的子节点，直到找到对应的节点或者到达叶子节点，将对应的节点删除。删除时，判断如果其父节点无其他子节点，将其父节点删除。递归向上删除。
- Compressed Radix Tree的删除操作:
  remove函数调用findCompressedRadixTreeNode函数，找到删除的位置，然后根据删除的位置，将对应的节点删除。删除时，判断是否需要合并节点，如果需要合并节点，进行合并操作。如果无需合并，将对应的节点删除即可。删除后，判断如果其父节点无其他子节点，将其父节点删除。递归向上删除。如果其父节点只有一个子节点且父节点不是根节点，将其父节点与子节点合并。向上递归。

2.2.4 实验难点

笔者在实现Compressed Radix Tree的时候，对于节点的插入和删除操作，需要考虑节点的合并和分裂，这一部分的实现比较复杂。

笔者在开始实现的时候，对于节点的合并和分裂的逻辑没有理清楚，对于一些边界情况没有考虑到，导

致实现的时候出现了一些问题。在Debug过程中，利用提供的测试用例`test_a.txt`进行最初简易版本的代码开发。针对`test_b.txt`逐步加入边界情况的处理，最终实现了`Compressed Radix Tree`的基本操作。而在`test_c.txt`中，针对数据量稍大的情况进行补充处理，保证了代码的正确性。（但是数据量较大的测试用例只有这个，可能还有一些边界情况没有考虑到，笔者自己补充一些测试用例以保证正确性）

3 测试

3.1 YCSB测试

3.1.1 测试配置

- 工作负载：在每轮测试开始前，初始化测试对象并加载 1000 个遵循$zipfian$分布的 `int32_t`到测试对象中。每组测试运行 45 秒。三种不同的工作负载描述如下：
  - WorkLoad1: 50% insert, 50% find
  - WorkLoad2: 100% find
  - WorkLoad3: 25% insert, 50% find, 25% delete
- 测试对象：Radix Tree、Compressed Radix Tree, RedBlack Tree (STL:map)
- 系统配置：VMware Workstation 17 Player: Ubuntu 22.04 LTS
- 机器配置：
  - CPU: 12th Gen Intel(R) Core(TM) i5-12500H CurrentClockSpeed:2500 MHz
  - Memory: Capacity 16GB, Speed 4800 MHz

3.1.2 测试结果

图 1: WorkLoad1-insert

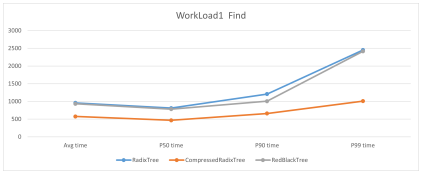

图 2: WorkLoad1-find

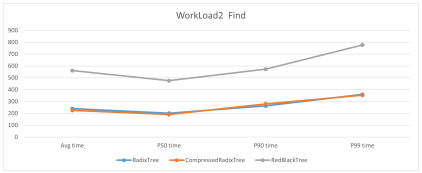

图 3: WorkLoad2-find

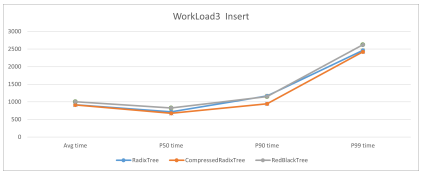

图 4: WorkLoad3-insert

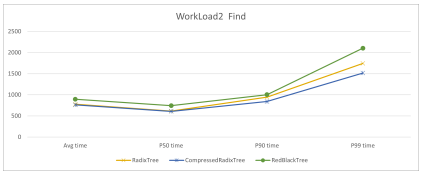

图 5: WorkLoad3-find

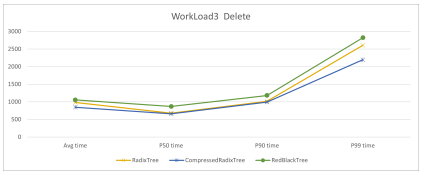

图 6: WorkLoad3-delete

3.1.3 结果分析

根据测试结果分析，可以看出在不同的工作负载下，Radix Tree、Compressed Radix Tree、RedBlack Tree的性能表现总体为：Compressed Radix Tree > Radix Tree > RedBlack Tree。但是在具体P50,P90,P99等情况下存在一些波动，可能是由于测试环境的影响，或者是由于测试数据的影响。

同时，可以看出在不同的工作负载下，Compressed Radix Tree的性能比Radix Tree更好，说明由于合并操作使节点更加紧凑，减少了空间的浪费，从而减少对内存的访问，提高了性能。

总体上，Insert/Find/Delete的时间与数据量成正相关，在WorkLoad2的情况下，find操作的时间较短说明。

4 结论

- 基数树：学习了基数树这项数据结构的基本操作和原理，并实现压缩优化。
- YCSB测试：学习了YCSB的基本使用，对于不同的工作负载下，不同数据结构的性能表现有了一定的了解。
- 绘图与报告分析：学习了如何绘制图表，如何用LaTeX撰写报告。
- 相同代码在不同测试情况下，表现变化较大，可能是由于测试数据的影响，也可能是由于测试环境的影响。

5 建议

- 可以提供YCSB测试与绘图的示例。
- 可以提供更多的测试用例，覆盖更多的边界情况。
- 增加结点对应的Value值，使得测试更符合实际应用场景。
- 提供的zipf函数将0~9999的数映射到0~INT_MAX,可能重复概率较大，存在一定的偏差。

参考文献

[1] Wikipedia, 基数树, https://zh.wikipedia.org/wiki/%E5%9F%BA%E6%95%B0%E6%A0%91,