基数树项目报告

赵楷越 522031910803

2024 年 4 月 11 日

1 背景介绍

在这个实验中，我要完成两个任务，分别是普通基数树的构建以及压缩优化后的基数树的构建。通过类比的思想，我也可以把他看作是字典树，前缀树的一种变体，只不过由于在基数树中存储的信息是大量的数值，因此在进行基数树的三种操作的过程中，要用到大量的位运算，对数值进行快速处理。这个实验很好地加深了我对基数树和位运算的理解。

2 系统实现

2.1 实现基数树的节点

基数树 RadixTree（Radix）的节点主要由一个大小为 4 的节点指针组成，其指向了代表四个不同前缀（00，01，10，11）的孩子。而压缩后的基数树 CompressedRadixTree（CRadix）的节点在 RadixTree 的基础上多了数值 value 和 length，分别代表该节点实际存储的数值，以及该存储数值的有效位数。

2.2 实现基数树的基本操作

基本操作有查询，插入和删除，其核心思想是利用数值的前缀两位进行快速比对处理。

1. **查询：**对于 Radix 的查询，每次取未比较位的最前两位对节点的孩子进行查询，迭代至叶子节点后便判断查询值存在。对于 CRadix，每次查询前缀是否存在之后，还要将查询值与对应实际存储值的有效位进行比对，比对成功后才能移动到下一个节点，直到移动到叶子节点。

2. **插入：**对于 Radix 的插入，每次取未比较位的最前两位对节点的孩子进行查询，当查询不到节点时，便不断对数值的剩余部分创建新节点，插入到 Radix 中，如果查询到了便不插入新节点。对于 CRadix，其主要分为三种情况：（1）当查询到插入值时，不插入新节

点；（2）当查询不到前缀值时，创建一个新节点插入当前访问到的节点；（3）当查询到前缀值，但是对应节点的实际存储值与插入值的对应位上并不相等，便要对该节点进行分裂操作。分裂操作算法具体如下：先找出当前的最长公共前缀、再将原节点拆成存储最长公共前缀和原节点的剩余部分值，然后新建一个节点存储插入值的剩余值，调整节点间孩子的关系，完成分裂。

3. **删除：** Radix 和 CRadix 的删除基本一致（父节点没有孩子时删除），除了在删除叶子节点后，在 CRadix 中若发现其父节点的子节点数变为 1，需要将其与其子节点进行合并。

2.3 印象较深的细节

第一个印象比较深的细节是对于压缩基数树节点的设计，一开始的时候，对题目理解并不到位，采用了 map 去存储该节点实际存储的值和该节点孩子的对应关系，但是在实验设计中发现，这样的方法并不能有效利用节点存储的前缀信息去快速查找一个数值是否存在于基数树当中。因此在查询了相关资料，通过类比的方法，我从压缩后的前缀字典树中得到的灵感，并重新读懂了题目中的引导，增加了 value 和 length 作为节点的属性，如此一来后面的思路便清晰许多。

第二个印象比较深的细节是对于在位运算中，当我要取到一个 int32t 类型的前 32 位时，应当对该情况做一个特殊判断，而不能直接用位运算的方法去进行处理，这个细节让我花费了不少时间。

```
41              if (i + 2 == 32) // 特判32位的情况，防止溢出
42                  current->children[index]->value = value;
43              else
44                  current->children[index]->value = (value & ((1 << (i + 2)) - 1));
```

3 测试

3.1 YCSB 测试

3.1.1 测试配置

本次测试配置基于 Ubuntu 22.04 的虚拟运行环境；虚拟机可用磁盘大小为 32GB；虚拟机内存为 8GB；虚拟机处理器内核总数为 6 个。测试对象为按要求自行实现的 RadixTree，CompressedRadixTree 以及 RedBlackTree（RedBlackTree 使用 stl 中的 set 容器实现）。在所有工作负载下，测试程序不断循环对测试对象调用某一种基础操作。查询、插入、删除的数值均服从 zipfian 分布。在每轮测试开始前，重新初始化了测试对象，加载 1000 个均匀随机分布的 int32t 类型到测试对象中。每组测试运行了 60s 从而获得了一个较为稳定的测试结果。工作负载均按照 YCSB 标准的以下三种工作负载模版进行测试。

1. workload-1: 在该负载下，每轮循环 50% 几率执行 find 操作,50% 几率执行 insert。
2. workload-2: 在该负载下，每轮循环 100% 执行 find 操作。
3. workload-3: 在该负载下，每轮循环 50% 执行 find, 25% 执行 insert, 25% 执行 del。

3.1.2 测试结果

本次实验记录了不同工作负载下，不同基本操作的平均时延、P50（第 50 百分位数）、P90（第 90 百分位数）和 P99 时延（第 99 百分位数），数据由以下所作图表所示（图表中数值单位为纳秒 ns）：

图 1-3 分别记录了在三种工作负载下，RadixTree（Radix）基本操作的时延。图 3-5 分别记录了 CompressedRadixTree（CRadix）基本操作的时延。图 7-9RedBlackTree（RBT）（点击查看源码来源）基本操作的时延。图表中 Total 列代表了所有操作的对应时延的平均值。

Lat	Find	Insert	Total
Avg	311	305	308
P50	298	293	296
P90	420	411	416
P99	684	676	680

图 1: Radix 1

Lat	Find	Total
Avg	60	60
P50	55	55
P90	78	78
P99	146	146

图 2: Radix 2

Lat	Find	Insert	Delete	Total
Avg	422	501	1030	594
P50	290	451	901	483
P90	785	744	1711	1006
P99	1103	1061	2070	1334

图 3: Radix 3

Lat	Find	Insert	Total
Avg	341	344	343
P50	320	322	321
P90	462	466	464
P99	721	727	724

图 4: CRadix 1

Lat	Find	Total
Avg	80	80
P50	75	75
P90	103	103
P99	138	138

图 5: CRadix 2

Lat	Find	Insert	Delete	Total
Avg	197	235	557	297
P50	177	214	519	272
P90	283	337	718	405
P99	484	575	1043	647

图 6: CRadix 3

Lat	Find	Insert	Total
Avg	448	2075	1262
P50	424	1850	1137
P90	692	2537	1615
P99	1002	5295	3149

图 7: RBT 1

Lat	Find	Total
Avg	163	163
P50	135	135
P90	258	258
P99	406	406

图 8: RBT 2

Lat	Find	Insert	Delete	Total
Avg	532	1023	978	766
P50	487	987	945	727
P90	850	1380	1297	1094
P99	1209	1801	1681	1475

图 9: RBT 3

3.1.3 结果分析

1. **对比 Radix 和 CRadix**。通过对比 Radix 和 CRadix 在不同 workload 下的性能表现，我们可以看到，Radix 在负载 1 和负载 2 下的性能表现略好于 CRadix，Radix 在负载 3 下的性能与其在负载 1 下的性能相差不大，原因是（负载 2 由于只有 find 操作，树中节点个数较少（1000 个），因此性能表现最好）。而 CRadix 在负载 3 下的性能表现明显优于未经节点压缩的 Radix。分析原因可能是因为负载 3 含有删除操作，而由于 CRadix 在删除操作时，会一并将可以合并的节点进行合并，也能减少树中的节点个数，使其平均性能水平向负载 2 趋近（树中数据规模较少的情况），从而提高了性能，但是由于 Radix 在删除时并不会进行多余的优化操作，因此在含有 25% 删除操作的负载 3 下，CRadix 的性能优于 Radix。结果符合预期。

2. **对比 Radix（包括 CRadix）和 RBT**。我们可以从图表中看到，Radix 和 CRadix 的时延小于 RBT 的时延。因为 Radix 不用像红黑树通过自身旋转达到平衡，因此，在大规模插入的情况下，Radix 相较于红黑树，少了很多对节点的频繁旋转和染色操作，所以 Radix 的平均插入性能明显优于 RBT 的平均插入性能。结果符合预期。

3. **分析同一负载状态下的平均时延、P50、P90 和 P99 时延**。每一组负载下的平均时延和 P50 代表了对应数据结构的常规表现。而当我们将注意力放在 P90 和 P99 时延上（代表程序会出现的极端情况）时，我们可以发现，对于 CRadix，其平均时延（或 P50）与 P90（或 P99）之间的极差相对于 Radix 与 RBT 来说更小，说明经过压缩优化的 CRadix 对于不同负载下的数据处理，函数的时延更为稳定。

4 结论

通过对 Radix 和 CRadix 的代码实现，我对基数树的认识更加深入了。同时，在经过对 Radix，CRadix 和 RBT 在不同负载下的性能对比之后，可以初步得到结论：

1. 在含有插入和删除的系列组合负载下，CRadix 的性能表现优于 Radix。因为 CRadix 的删除也能压缩节点。但在没有删除的负载下，依然可以选用 Radix。

2. 在含有大量频繁插入的负载下，基数树的性能表现优于 RBT，因为基数树少了很多对节点的频繁旋转和染色操作，只需在基数树的节点中按序插入，性能水平不会因为数据规模的增大而明显下降。