2021 级

《大数据存储系统与管理》课程

课 程 报 告

姓　名　余泽坤

学　号　U202014698

班　号　计算机 2005 班

日　期　2024.4.22

目　　录

一、实验目的

1. 了解 Cuckoo Hashing 算法实现及性能。
2. 尝试改进 Cuckoo Hashing 算法。

二、实验背景

Cuckoo Hashing（布谷鸟哈希算法）是一个尝试使用直观方法解决哈希碰撞问题的算法。

算法的基本思想是在插入时使用两种哈希函数计算一个值在表中的两个键值。如果存在一个键值为空，则插入该键值。否则尝试将任意一个键值挤出其原本的位置，使被挤出的值移动到另一个可能的键值，以此类推递归进行处理，直到所有的值都找到至少一个空位为止，如图 2-1 所示。

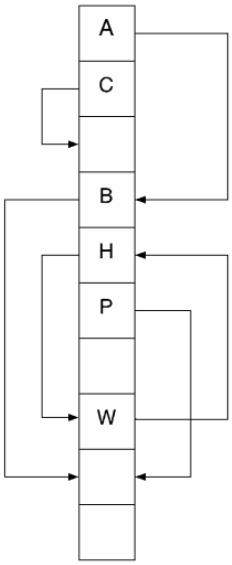

图 2-1 Cuckoo Hashing 算法示意图

常见的 Cuckoo Hashing 的算法的变种将一张表分为两张，使用两个 Hash 函数分别计算在两张表上的键值以期望减少哈希碰撞的问题。额外的，也可以将 Cuckoo Hashing 的表从两张变为更多，以使用更多哈希函数计算更多的键值来减少哈希碰撞的发生概率。

这些变种具有的一个严重问题就是在一个值尝试取代另一个值的位置时，这样的可替代关系构成的有向图可能会包含循环（即连通分量）。在表为两张时每个点的出度为 1，这样的图将会变为一个基环树，而在表更多的情况下问题将变得更加复杂。在这样的替代环/替代链增多的情况下，基于搜索算法进行键值取代的插入操作复杂度可能会急剧恶化。但算法具有一个优点，即查询的复杂度是$O(1)$的。

为了方便下面的报告对 Cuckoo Hashing 算法进行描述，我们令$Buckets$为表的

数目，令$BucketSize$为每张表的大小。为了简化问题，我们令 Cuckoo Hashing 使用的哈希函数为朴素的线性映射函数，如下方公式所示：

$$Hash(Value) = (a \times Value + b)\%(BucketSize - 1) + 1$$

为了实现的方便，我们将$Value$映射到范围$[1, BucketSize\text{-}1]$，其中a, b为哈希函数的参数。对每张表我们将随机在一个范围内设定a, b的值来为每张表设定不同的哈希函数。

三、实验环境

实验的测试环境使用 Intel i5-13600KF 作为 CPU，VirtualBox 下的 Ubuntu 22.04 LTS 作为系统环境，Clang 14.0.0-1ubuntu1.1 作为编译器，使用 C++完成课程报告的所有实验内容。

四、实验内容

为了验证 Cuckoo 算法的性能与改进 Cuckoo 算法的复杂度恶化情况，我们将使用多种算法来实现 Cuckoo 算法的插入部分，并尝试分析、比较其性能在实际情况下的差异。

4.1 Cuckoo Hashing 基本组成

本节介绍 Cuckoo Hashing 算法的基本组成部分。我们将为 Cuckoo Hashing 分配大小为 $BucketSize$ 的 $Buckets$ 张表，则哈希表的总大小为 $TotalSize = BucketSize \times Buckets$。我们将为每一张表分配一个参数随机的朴素线性映射函数 $Hash_i(Value)$ 作为哈希函数，则值 $Value$ 在表 i 的对应键值则为 $Key_i = Hash_i(Value)$。

在插入值$Value$到表中时，我们会尝试找到一个对应键值Key_e，在表e的该键值下没有值被插入。若不存在这样的表e，则尝试取出一个键值Key_f与对应的值$Value_f$，将$Value$插入Key_f以取代值$Value_f$，然后继续递归尝试插入值$Value_f$到表中，直到所有的值都找到空位为止。

若插入失败（在不可能为所有的值找到空位的情况下），为了简化问题，我们会使用朴素的方式尝试将一个环上的值取出到额外的一张线性表中，以腾出一个空位，再尝试使用相同的方式插入值$Value$。

在查找值$Value$时，我们会查找$Value$所有的可能键值Key_i，若没有找到$Value$，则说明$Value$被移动到了额外的线性表中，需要遍历线性表与$Value$进行比对。

忽略线性表的情况下，易分析得到 Cuckoo Hashing 的查询复杂度为 $O(Buckets)$。在线性表最大长度为$maxLen$的情况下，Cuckoo Hashing 的查询复杂度为$O(maxLen + Buckets)$，若$maxLen + Buckets$较小，我们可以近似地认为该

算法的查询复杂度为$O(1)$。

为了方便报告进行描述，我们为 Cuckoo Hashing 值的可替代关系构建一张有向图，每个节点代表已被占据的键值或未被占据的键值。所有被占据键值的点会向其替选的键值节点连接一条有向边。易知这张图上每个节点的出度为0或$Buckets-1$。

4.2 朴素深度优先搜索插入算法

根据 4.1 节的分析易知 Cuckoo Hashing 的效率主要取决于其插入部分和线性表的长度（插入失败/冲突次数）。为了实现 Cuckoo Hashing 的基本功能，我们尝试实现一个朴素的 DFS 算法，其将遍历 Cuckoo Hashing 替代图上的每一个点，直到找到一个空位为止，如下所示。

```
bool naive_dfs(int bucket, int idx) {
    lastVisited[bucket][idx] = TimeStamp;
    if(!occupied[bucket][idx]) return true;

    for(auto to:alt[bucket][idx])
        if(to.first != bucket && lastVisited[to.first][to.second] != TimeStamp) {
            if(naive_dfs(to.first, to.second)) {
                Swap_Times ++;
                occupied[to.first][to.second] = true;
                occupied[bucket][idx] = false;
                data[to.first][to.second] = data[bucket][idx];
                alt[to.first][to.second] = alt[bucket][idx];
                return true;
            }
        }
    return false;
}
```

我们将随机插入$TotalSize$个元素以尝试对哈希表上的每一个键值都插入一个值。在插入元素数目接近$0.9\times TotalSize$时，我们可以观察到插入操作明显的复杂度恶化现象，分析可以得到这是因为插入失败的情况与替代链的深度在明显增加导致，因此朴素 DFS 在哈希表接近满载时效率十分底下。

4.3 朴素广度优先搜索插入算法

根据 4.2 节的分析，朴素 DFS 算法效率底下的一个重要原因就是替代链的深度明显增加。如果使用朴素 BFS 算法来替代 DFS 算法，我们就可以找到深度最浅

的空位，以减少替代过程的计算量。

```
bool naive_bfs(int bucket, int idx) {
    queue<pair<int, int>> qu;
    qu.push({bucket, idx});
    lastVisited[bucket][idx] = TimeStamp;
    fromVisited[bucket][idx] = {0,0};
    int searched_nodes = 0;
    while(!qu.empty()) {
        searched_nodes ++;
        assert(searched_nodes < TotalSize);
        auto x = qu.front();
        qu.pop();
        for(auto to: alt[x.first][x.second])
            if(to != x && lastVisited[to.first][to.second] != TimeStamp) {
                if(!occupied[to.first][to.second]) {
                    auto nx = x;
                    auto nt = to;
                    while(nx.second) {
                        Swap_Times ++;
                        occupied[nt.first][nt.second] = true;
                        occupied[nx.first][nx.second] = false;
                        data[nt.first][nt.second] = data[nx.first][nx.second];
                        alt[nt.first][nt.second] = alt[nx.first][nx.second];
                        nt = nx;
                        nx = fromVisited[nx.first][nx.second];
                    }
                    return true;
                }
                fromVisited[to.first][to.second] = x;
                lastVisited[to.first][to.second] = TimeStamp;
                qu.push(to);
            }
    }
    return false;
}
```

通过 BFS 算法我们解决了替代链过深的问题，但是我们仍然能在哈希表接近满载时观察到明显的复杂度恶化现象。通过分析可以得到因为在哈希表接近满载时存在大量的插入失败的情况，在每次 BFS 进行搜索的时候将遍历图上的大量节点，导致插入效率急剧恶化。

为了解决这一问题我们有许多思路可以入手。由于报告篇幅与实验时间限制，我们这里仅尝试了两种解决思路。

4.4 BFS 随机重构算法

我们将在下面提出一种基于随机算法和固定参数重构替代链的方式来解决复杂度恶化的问题。

基于遍历节点的 BFS 树我们很难使用简单易懂的方式去找到一个或多个连通分量并决定环上的哪几个节点应该被取出来得到最佳的结果。但通过观察我们可以得到在进行 BFS 遍历时，能够成功插入的数据往往得到较浅的替代链，而插入失败的数据将在 BFS 遍历时付出迅速恶化到几万甚至几十万节点的搜索代价。

若我们可以找到一个参数限制广度优先搜索的深度，则可以迅速地对大概率失败的插入结果进行优化。在我们对已经被遍历的点结构一无所知的情况下，我们可以基于随机算法在 BFS 树上随机选择一个点取出到线性表中，并以这个点到 BFS 树的根节点作为替代链进行替代操作。这个点可能处在某个连通分量中，也可能没有处在某个连通分量中，我们对这个点一无所知，仅仅只是基于随机算法强制地基于 BFS 上的替代关系重新分配了链上每一个节点的位置。因此 Cuckoo Hashing 替代图上是否存在连通分量对这个算法而言并不重要。

为了简化问题，我们将 BFS 树的节点最大值设置为$Buckets^k$。其中k为一个固定的参数。在 4.1 节中我们已经说明 BFS 树上的每一个点的出度最大为$Buckets - 1$，因此设置节点最大值为$Buckets^k$可以近似地认为在最优的情况下，BFS 树的最大深度为k，最坏深度为$Buckets^k$。BFS 的搜索范围随着k的增大而增加，插入失败的概率会因此减小，但如 4.3 节所述算法的复杂度可能会急剧恶化。

为了测试算法，我们使用如下的方法对本报告的所有算法进行测试：我们将随机生成$TotalSize$个任意 Long Long 范围（范围大小为2^{63}）的有符号非负整型，并将其插入到哈希表中试图使其满载，并在打乱其插入顺序之后对所有被插入的元素进行一次查询，计算其插入/查询速度以比较算法的性能与验证算法正确性。

经过测试，在实践中令$k = 3,4$，$Buckets$范围在$[5,16]$可以得到一个比较均衡的插入/查询复杂度。理论上算法的插入复杂度最坏为$O(Buckets^k)$，但在实践中观察到插入/查询复杂度都得到了近似$O(1)$的结果。在$Buckets = 8, BucketSize = 250000, k = 3$的情况下，插入$TotalSize = 2000000$的数据使哈希表的负载率达到了 99.6%，仅 0.3%（6542 次）的插入操作存在数据被移动到线性表中（线性查询的复杂度最坏为$O(n)$），如表 5-1 所示。若使用另外的树形数据结构或哈希表替代

线性表对数据进行维护我们还可以得到更加优秀的查询复杂度如$O(\log n), O(1)$，但为了简化问题我们在此不尝试深入进行实验。

算法的 C++实现如下所示。

```
bool naive_bfs_impl_extra(int bucket, int idx) {
    assert(occupied[bucket][idx]);
    queue<pair<int, int>> qu;
    qu.push({bucket, idx});
    lastVisited[bucket][idx] = TimeStamp;
    fromVisited[bucket][idx] = {0,0};
    vector<pair<int,int>> nodes;
    while(!qu.empty() && nodes.size() < BFSReconstructThreshold) {
        auto x = qu.front();
        qu.pop();
        nodes.push_back(x);
        for(auto to: alt[x.first][x.second])
            if(to != x) {
                if(lastVisited[to.first][to.second] != TimeStamp) {
                    if(!occupied[to.first][to.second]) {
                        auto nx = x;
                        auto nt = to;
                        do {
                            assert(!occupied[nt.first][nt.second]);
                            Swap_Times ++;
                            occupied[nt.first][nt.second] = true;
                            occupied[nx.first][nx.second] = false;
                            data[nt.first][nt.second] = data[nx.first][nx.second];
                            alt[nt.first][nt.second] = alt[nx.first][nx.second];
                            nt = nx;
                            nx = fromVisited[nx.first][nx.second];
                        } while(nx.second);
                        assert(nt == (pair<int,int>(bucket, idx)));
                        return true;
                    }
                    lastVisited[to.first][to.second] = TimeStamp;
                    fromVisited[to.first][to.second] = x;
```

```
                    qu.push(to);
                }
                else {
                    nodes.push_back(to);
                }
            }
        }

        // Randomly choose an alternate point.
        int ridx = rand_int(0, nodes.size() - 1);
        auto nt = nodes[ridx];
        auto nx = fromVisited[nt.first][nt.second];
        extraData.push_back(data[nt.first][nt.second]);
        occupied[nt.first][nt.second]=false;

        while(nx.second) {
            assert(!occupied[nt.first][nt.second]);
            Swap_Times ++;
            occupied[nt.first][nt.second] = true;
            occupied[nx.first][nx.second] = false;
            data[nt.first][nt.second] = data[nx.first][nx.second];
            alt[nt.first][nt.second] = alt[nx.first][nx.second];
            nt = nx;
            nx = fromVisited[nx.first][nx.second];
        }

        return false;
    }
```

4.5 BFS 可变数组算法

另一种思路是利用可变数组的思想，对哈希表进行动态扩充。算法的主要思想是在当前已插入的元素数目$Elements$大于TotalSize$/k$的情况下令$TotalSize$变为原先的k倍，然后整体重构哈希表并重新插入所有元素。通过均摊复杂度分析我们可以得知重构操作的均摊复杂度将与插入操作保持一致，因此若将重构操作视作插入操作的一部分，插入操作的最优$O(1)$复杂度将保持不变。由于篇幅问题，在此我们省略可变数组的均摊复杂度分析。

通过这个算法我们可以计算得到哈希表的最大负载率将不超过$1/k$，在这样的负载率下哈希表发生冲突的概率几乎为 0（实际上不止对 Cuckoo 算法而言），因此 4.4 节的算法对可变数组算法而言影响较小，不过我们在本次实验中仍然选择将可变数组算法基于随机重构算法进行测试。牺牲空间复杂度（最坏情况下为$O(\mathrm{kn})$）和负载率的情况下换取了插入和查询的优秀时间复杂度是这个算法的核心思想。

五、实验结果

我们将在这一部分将 4.4 节、4.5 节的算法与标准 C++ STL 实现进行比较（编译参数为-O3）。对算法的测试方法已在 4.4 节进行描述。我们固定插入的元素总数（$TotalSize$）为 2000000(2e6)，令$BucketSize = TotalSize/Buckets$，在仅改变$Buckets$的情况下对算法进行测试其 IPS（每秒插入操作数）、QPS（每秒查询操作数）、负载率（Cuckoo Hashing 表中包含元素的键值数量与$TotalSize$之比，不包含线性表）。

特别地，我们将 4.5 节算法中的重构时间视作插入时间的一部分以计算插入操作的平均速度。

由于插入失败的概率极低，发生随机重构过程的概率可忽略不计（期望负载率为 50%），因此可变数组算法将基于随机重构算法（在该场景下几乎等效于朴素 BFS 算法）进行实验。

实验中所有参数和随机数种子不固定。

表 5-1 BFS 随机重构算法效率实验结果（$k = 3$）

Buckets	**2**	**3**	**4**	**5**	**8**	**12**	**16**
IPS（百万）	**2.097**	1.626	1.519	1.433	1.338	1.289	1.265
QPS（百万）	0.128	0.599	2.401	5.934	**10.725**	9.906	9.630
负载率	80.8%	91.4%	96.2%	98.2%	99.6%	99.9%	**100%**

表 5-2 BFS 可变数组+随机重构算法效率实验结果（可变数组$k = 2$，随机重构$k = 3$）

Buckets	**2**	**3**	**4**	**5**	**8**	**12**	**16**
IPS（百万）	**2.915**	2.727	2.561	2.426	2.169	1.875	1.876
QPS（百万）	3.941	8.008	8.532	8.995	**9.351**	8.464	9.236
负载率	48.1%	**50.0%**					

表 5-3 STL(Unordered multiset)效率对比

算法	随机重构 $\boldsymbol{Buckets = 8}$	可变数组 $\boldsymbol{Buckets = 5}$	可变数组 $\boldsymbol{Buckets = 8}$	STL
IPS（百万）	1.338	**2.426**	2.169	***5.844***
QPS（百万）	**10.725**	8.995	9.351	***34.037***

课程报告的相关 Git 库见 Report_assets 压缩包，其中包含完整的实验数据日志记录与实验代码。

六、实验总结

通过观察实验结果我们可以得到一些基本的结论：数据显示随机重构算法的查询效率最高，可变数组算法的插入效率最高，而两种算法都达到了每秒百万的插入/查询速度，随机重构的算法整体效率约为 STL 的 30%。STL 相对而言的高效率实现可能与 C++ 语言本身和我们未对算法和线性表进行深入优化有关，但足以证明随机重构算法的实践效率实际上十分可行。

由于篇幅限制和实验时间限制，我们只尝试了两种解决方案与随机数据集，单词查询与操作过程而没能进行更深入的实验与探索，实验过程和算法设计也存在许多可改进之处，实为遗憾。在本次课程报告中我们深入探索了 Cuckoo 算法的实现、性能、测试与缺陷，并尝试从多种角度去解决 Cuckoo 算法的缺陷，平衡了其插入/查询效率。实验过程令我收益良多，过程十分愉悦充实。在此感谢设计课程报告课题的老师们的辛勤付出。