华中科技大学

2024

大数据存储系统与管理

课程报告

题　　目：	CuckooMap 算法性能测试
专　　业：	计算机科学与技术
班　　级：	CS2109
学　　号：	U202110415
姓　　名：	卢舒愉
电　　话：	13870341681
邮　　件：	Lushuyu-chan@qq.com

目 录

1 Cuckoo Map 数据结构的设计

1.1 Cuckoo 的基本工具函数

此部分代码位于 include/CuckooMap/CuckooHelpers.h 中。

1. 快速 64 位哈希

fasthash64 是 CuckooHelpers.h 文件中定义的一个辅助哈希函数。

```
1. uint64_t fasthash64(const void *buf, size_t len, uint64_t seed) {
2.     uint64_t const m = 0x880355f21e6d1965ULL;
3.     uint64_t const *pos = (uint64_t const *)buf;
4.     uint64_t const *end = pos + (len / 8);
5.     const unsigned char *pos2;
6.     uint64_t h = seed ^ (len * m);
7.     uint64_t v;
8.
9.     while (pos != end) {
10.        v = *pos++;
11.        h ^= mix(v);
12.        h *= m;
13.    }
14.
15.    pos2 = (const unsigned char *)pos;
16.    v = 0;
17.
18.    switch (len & 7) {
19.        case 7:
20.            v ^= (uint64_t)pos2[6] << 48;
21.        case 6:
```

```
            v ^= (uint64_t)pos2[5] << 40;
        case 5:
            v ^= (uint64_t)pos2[4] << 32;
        case 4:
            v ^= (uint64_t)pos2[3] << 24;
        case 3:
            v ^= (uint64_t)pos2[2] << 16;
        case 2:
            v ^= (uint64_t)pos2[1] << 8;
        case 1:
            v ^= (uint64_t)pos2[0];
            h ^= mix(v);
            h *= m;
    }

    return mix(h);
}
```

这是一个快速哈希函数，用于计算给定缓冲区 buf 中数据的哈希值。它采用 64 位哈希值，并使用 Cuckoo Hashing 策略来生成哈希值。

- 首先，函数声明了一些变量，包括常量 m、指向 buf 数据的指针 pos 和 end，以及指向字节数组 pos2 的指针。
- 接下来，函数通过将种子 seed 与 len（缓冲区长度）和常量 m 进行异或操作来初始化哈希值 h。
- 然后，函数通过遍历 buf 中的每个 64 位数据，将其与哈希值 h 进行混合运算，并将结果保存在 h 中。这是通过调用 mix 函数来完成的。
- 如果缓冲区的长度不能整除 8，那么函数需要处理剩余的字节。它将这些字节按顺序与 v 进行异或操作，并将结果通过 mix 函数混合到 h 中。
- 最后，函数返回混合后的哈希值 h。

2. 哈希随机化

在 fasthash64 函数中，mix 函数的作用是增加哈希的随机性，并减少哈希碰撞的可能性。它通过对哈希值进行一系列的位操作，改变位的排列方式，从而使得输入数据的微小变化能够在哈希值中得到充分体现。

```
1. static inline uint64_t mix(uint64_t h) {
2.     h ^= h >> 23;
3.     h *= 0x2127599bf4325c37ULL;
4.     h ^= h >> 47;
5.     return h;
6.
```

1.2 Cuckoo Filter 的实现

此部分代码位于 include/CuckooMap/CuckooFilter.h 中。

CuckooFilter.h 定义了 CuckooFilter 类模板，用于实现过滤器数据结构。

1. 类模板定义

CuckooFilter 具有以下模板参数：

- Key：表示键的类型
- HashKey1：表示用于第一个哈希函数的类型，默认为 HashWithSeed<Key, 0xdeadbeefdeadbeefULL>
- HashKey2：表示用于第二个哈希函数的类型，默认为 HashWithSeed<Key, 0xabcdefabcdef1234ULL>
- HashShort：表示用于短哈希函数的类型，默认为 HashWithSeed<uint16_t, 0xfedcbafedcba4321ULL>
- CompKey：表示用于键比较的类型，默认为 std::equal_to<Key>

2. 成员变量和构造函数

- _size：表示过滤器的大小，即存储槽位的数量
- _nrUsed：表示当前使用的槽位数量

- _useMmap：表示是否使用内存映射文件的方式进行分配和释放内存
- _allocSize：表示分配的内存大小
- _allocBase：指向分配的内存的基地址
- _base：指向对齐后的分配内存地址，保证 64 字节对齐
- _tmpFileName：临时文件的文件名，用于内存映射文件时使用
- _tmpFile：临时文件的文件描述符，用于内存映射文件时使用
- _valueSize：表示值的大小，默认为 sizeof(bool)
- _valueAlign：表示值的对齐，默认为 alignof(bool)
- _slotSize：槽位的大小，即键的大小加上值的大小，用于在内存块中定位槽位的位置
- _valueOffset：值相对于槽位的偏移量
- _hasher1：哈希函数 1 的实例
- _hasher2：哈希函数 2 的实例
- _filters：存储过滤器数据的 char 数组

3. 成员方法

- lookup：查找给定键是否在过滤器中存在，它的返回值是一个布尔类型，表示查找结果。如果键存在于过滤器中，返回 true；如果键不存在于过滤器中，返回 false。
- insert：向过滤器中插入给定键，它的返回值是一个布尔类型，表示插入操作的结果。如果成功插入键，返回 true；如果键已经存在于过滤器中，返回 false。
- remove：从过滤器中移除给定键，它的返回值是一个布尔类型，表示移除操作的结果。如果成功移除键，返回 true；如果键不存在于过滤器中，返回 false。
- check：检查给定指针是否超出内存范围，它的返回值是一个布尔类型，表示检查结果。如果指针超出了内存范围，返回 true；否则返回 false。

1.3 Cuckoo Map 的实现

此部分代码分别位于 CuckooMap.h 和 InternalCuckooMap.h 中。

InternalCuckooMap 是 Cuckoo 哈希表中的子表，用于存储键值对。它使用两个哈希函数和一种键值对的比较方式对键进行哈希和查找。

CuckooMap 是基于 Cuckoo 算法实现的哈希表，它由多个 InternalCuckooMap 组成，用于处理哈希冲突。CuckooMap 提供了对外的接口，包含插入、查找和删除等操作。它在执行这些操作时，会根据键的哈希值将键值对存储到相应的 InternalCuckooMap 子表中。

1. innerLookup 方法的实现

```
1. void innerLookup(Key const &k, Finding &f, bool moveToFront) {
2.     char buffer[_valueSize];
3.     // f 必须初始化为 _key == nullptr
4.     for (int32_t layer = 0; static_cast<uint32_t>(layer) < _tables.size();
5.          ++layer) {
6.         Subtable &sub = *_tables[layer];
7.         Filter &filter = _useFilters ? *_filters[layer] : _dummyFilter;
8.         Key *key;
9.         Value *value;
10.        bool found = _useFilters ? (filter.lookup(k) && sub.lookup(k, key, value))
11.                                 : sub.lookup(k, key, value);
12.        if (found) {
13.            f._key = key;
14.            f._value = value;
15.            f._layer = layer;
16.            if (moveToFront && layer > 0) {
17.                uint8_t fromBack = _tables.size() - layer;
18.                uint8_t denominator = (fromBack >= 6) ? (2 << 6) : (2 << fromBack);
19.                uint8_t mask = denominator - 1;
20.                uint8_t r = pseudoRandomChoice();
21.                if ((r & mask) == 0) {
22.                    Key kCopy = *key;
23.                    memcpy(buffer, value, _valueSize);
```

```
                    Value *vCopy = reinterpret_cast<Value *>(&buffer);

                    innerRemove(f);
                    innerInsert(kCopy, vCopy, &f, layer - 1);
                }
            }
            return;
        }
    }
}
```

- 首先，该方法会在每个子表 sub 上进行查找操作。
- 先根据 _useFilters 决定是否使用过滤器 filter 进行查找。
- 如果 _useFilters 为 true，则通过调用 filter.lookup 和 sub.lookup 进行查找。
- 如果在子表 sub 中找到了对应的键值对，则将其存储到 f 中，并根据 moveToFront 参数决定是否执行移至前端的操作。
- 移至前端操作的目的是为了优化查找，减少后续查找的时间。这里使用了一个基于伪随机数的算法，根据 layer 和随机数生成一个值，并判断是否满足特定条件，如果满足，则执行移至前端的操作。
- 如果没有在任何子表中找到对应的键值对，则说明查找不成功。

2. innerInsert 方法的实现

```
bool innerInsert(Key const &k, Value const *v, Finding *f, int layerHint) {
    // 向表中插入键值对
    // 如果键 k 已经存在于表中，则返回 -1，表不会发生改变，此时 k 和 *v 也不会改变。
    // 否则，如果在表中还没有键 k 的存在，插入成功，返回 true。如果没有冲突，则返回 0，此时 k 和 *v 也不会改变。如果需要从表中删除一个键值对，则 k 和 *v 将被覆盖为要删除的键值对的值，返回 1。
    //
```

```
    // 如果 kPtr 和 vPtr 为非空指针，并且返回值为非负数，则将 k 和 v 在表中的位置分别写入 *kPtr 和 *vPtr。
    Key *kTable;
    Value *vTable;

    uint64_t hash1 = _hasher1(k);
    uint64_t pos1 = hashToPos(hash1);
    // 我们已经在这里计算第二个哈希值，以便在第一个循环中存储结果以便忍受误判。
    uint64_t hash2 = _hasher2(k);
    uint64_t pos2 = hashToPos(hash2);
    for (uint64_t i = 0; i < SlotsPerBucket; ++i) {
        kTable = findSlotKey(pos1, i);
        if (kTable->empty()) {
            vTable = findSlotValue(pos1, i);
            *kTable = k;
            std::memcpy(vTable, v, _valueSize);
            ++_nrUsed;
            if (kPtr != nullptr && vPtr != nullptr) {
                *kPtr = kTable;
                *vPtr = vTable;
            }
            return true;
        }
        if (_compKey(*kTable, k)) {
            return -1;
        }
    }
    for (uint64_t i = 0; i < SlotsPerBucket; ++i) {
        kTable = findSlotKey(pos2, i);
```

```
34.         if (kTable->empty()) {
35.             vTable = findSlotValue(pos2, i);
36.             *kTable = k;
37.             std::memcpy(vTable, v, _valueSize);
38.             ++_nrUsed;
39.             if (kPtr != nullptr && vPtr != nullptr) {
40.                 *kPtr = kTable;
41.                 *vPtr = vTable;
42.             }
43.             return true;
44.         }
45.         if (_compKey(*kTable, k)) {
46.             return -1;
47.         }
48.     }
49.
50.     // 现在从这些插槽中删除一个键的元素：
51.     uint8_t r = pseudoRandomChoice();
52.     if ((r & 1) != 0) {
53.         pos1 = pos2;
54.     }
55.     uint64_t i = (r >> 1) & (SlotsPerBucket - 1);
56.     // 我们删除位置为 pos1 和插槽为 i 的元素：
57.     kTable = findSlotKey(pos1, i);
58.     vTable = findSlotValue(pos1, i);
59.     Key kDummy = std::move(*kTable);
60.     *kTable = std::move(k);
61.     k = std::move(kDummy);
62.     std::memcpy(_theBuffer, vTable, _valueSize);
```

```
63.     std::memcpy(vTable, v, _valueSize);
64.     std::memcpy(v, _theBuffer, _valueSize);
65.     if (kPtr != nullptr && vPtr != nullptr) {
66.         *kPtr = kTable;
67.         *vPtr = vTable;
68.     }
69.     return true;
70. }
```

- 首先，该方法会在两个位置 pos1 和 pos2 的插槽上进行插入操作，用于解决哈希冲突。
- 在每个插槽中遍历 SlotsPerBucket 个槽位，如果找到空槽，则在其中插入键值对，并返回 true。如果找到相同的键，则返回 -1，表示插入失败。
- 如果没有找到空槽或相同的键，则需要随机选择一个插槽进行元素的替换。使用一个伪随机数 r 来选择 pos1 或 pos2 中的一个，并选择其中一个 SlotsPerBucket 槽位进行替换。
- 将要删除的键值对的键存储于 kDummy 变量中，并将要插入的键 k 替换到对应的插槽中。同时，将原始的键值对的值从 vTable 拷贝到 _theBuffer 中，再将新的值 v 拷贝回 vTable 中。
- 如果 kPtr 和 vPtr 非空指针，并且返回值非负数，则将键和值的位置分别写入 *kPtr 和 *vPtr。

3. innerRemove 方法的实现

```
1. void innerRemove(Finding &f) {
2.     if (_useFilters) {
3.         _filters[f._layer]->remove(*(f._key));
4.     }
5.     _tables[f._layer]->remove(f._key, f._value);
6.     f._key = nullptr;
7.     --_nrUsed;
8. }
```

- 首先，判断是否使用了过滤器 _useFilters，如果使用，则调用对应子表的过滤器 remove 方法来移除键的过滤器信息。
- 然后，调用 remove 方法从对应的子表中移除键值对。

- 最后，将 f._key 置为 nullptr，表示键值对已被移除，并将 _nrUsed 减一，表示表中的键值对数量减一。

2 性能测试

2.1 性能测试设计

此部分代码位于 tests/PerformanceTest.cpp 中。

在 PerformanceTest.cpp 中，通过构造随机数据，模拟三种操作（插入、查找和删除)来对比 CuckooMap 和 C++ STL 自带的 unordered_map 进行对比。还引入了 qdigest.h 以统计花费的时间成本。

根据命令行参数指定的初始大小（nInitialSize），使用 for 循环向数据结构中插入初始数据。初始数据是一个连续的从 1 开始的整数序列，根据序列号构造对应的键和值，并调用数据结构对象的插入函数将键值对插入数据结构中。

经过初始插入后，数据结构会包含 nInitialSize 个键值对。

在性能测试的执行过程中，根据命令行参数指定的操作次数（nOpCount）以及各个操作的概率（pInsert、pLookup、pRemove），使用随机数生成器 RandomNumber 生成 opCode 以选择不同的操作。

- 在插入操作中，使用 RandomNumber 类生成随机数，在指定范围内（当前最小值和最大值之间）生成一个随机的当前键值，并调用数据结构对象的插入函数将键值对插入数据结构中。
- 对于查找操作，根据当前的最大元素和最小元素以及工作集大小（nWorking），选择在工作集范围内或者整个范围内生成随机键，并调用数据结构对象的查找函数进行键的查找。
- 对于删除操作，根据当前的最大元素和最小元素，选择在整个范围内删除最小元素或者最大元素，调用数据结构对象的删除函数进行键的删除。

2.2 数据处理

在 battery.csv 文件中存放了 12 个测试基本参数，我写了 shell 脚本文件 test.sh，依次测试 battery.csv 中的参数，将结果转化为 markdown 文件输出。

2.3 测试结果

以 Result Set 0 为例：

- 测试参数

nOpCount	nInitialSize	nMaxSize	nWorking	pInsert	pLookup	pRemove	pWorking	pMiss
1000000	100000	10000000000	50000	0.09	0.90	0.01	0.90	0.00

- 空间开销

CM	UM
179545	179545

- 操作延迟

insert

	50.0p	95.0p	99.0p	99.9p
CM	208	1417	4167	7291
UM	250	375	667	5208

lookup

	50.0p	95.0p	99.0p	99.9p
CM	167	209	333	1083
UM	125	250	375	667

remove

	50.0p	95.0p	99.0p	99.9p
CM	167	708	2958	4250
UM	208	375	3000	4417

更多结果在 result.md 中。