华中科技大学

大数据存储系统与管理报告

姓　名：　陈栩民

学　院：　计算机科学与技术学院

专　业：　计算机科学与技术

班　级：　计卓 2101

学　号：　U202115306

指导教师：　华宇

分数	
教师签名	

2024 年 4 月 20 日

目 录

1 数据结构的设计

本次实验固定使用两个哈希函数，通过设置门限值的方式来检测无限循环，并探究多路相联数 slot 的提高对无限循环产生概率的影响。依据以上思路，设计了 CuckooHash 的相应数据结构。

1.1 CuckooHashingSlots 类

将 CuckooHash 封装成一个类，并且为了方便后续实验的顺利进行，提供了相联数 slot 维度的拓展性。

- 该类包含如下的私有成员：

1. int unitsNum, unitsLen, slotsNum;

分别记录 hash 函数数量，单个哈希表长度，哈希表多路相联数；

2. Hash h;

Hash 为自定义类，用于提供哈希函数，并在出现死循环时提供重新哈希的接口 rehash()；

3. int crash;

记录测试过程中的无限循环后的 hash 重建次数；

4. std::unordered_set<int> elements;

存储一系列随机生成且不重复的数据，作为测试数据；

5. int* units;

将所有的 hash 表全部压缩为一个一维数组，通过 units 指针访问该数组；

6. int& Value(int unit, int pos, int slot);

提供 unit、pos、slot 三维属性向 units 下标一维属性转化的函数接口；

7. bool findElementInSlots(int unit, int pos, int value);

unit 表示选取的哈希函数编号，pos 表示哈希出的偏移地址值，value 表示要查询的值，该函数表示在对应的多路 slot 中寻找 value 元素，函数返回找到与否的状态；

8. bool tryInsertInSlots(int unit, int pos, int value);

尝试插入元素；

9. bool tryDeleteInSlots(int unit, int pos, int value);

尝试删除元素；

10. bool InsertElement(int value);

实际插入元素；

- 该类包含的公共接口：

  1. CuckooHashingSlots(int len, int num)；

  CuckooHash 类构造函数

  2. ~CuckooHashingSlots()；

  CuckooHash 类析构函数

  3. int count() const;

  查看当前 Cuckoo 中的元素数目

  4. int cap() const;

  查看 Cuckoo 对象的元素容量

  5. bool Insert(int value);

  插入元素接口；

  6. bool Delete(int value);

  删除元素接口；

  7. bool Search(int value);

  搜索元素接口；

  8. bool rebuild(int value);

  产生无限循环时，重新 hash 后的 value 重建接口；

2 操作流程分析

本次实验固定使用两个哈希函数，通过设置门限值的方式来避免无限循环，并探究多路相联数 slot 的提高对无限循环产生概率的影响。为了保证实现的 CuckooHash 各操作结果确实符合理论预期，处于篇幅考虑，下文将挑选部分核心模块作为例子，从函数执行流程和子模块测试方法两个方面，即逻辑和实际数据的层面，说明实现的正确性。

2.1 部分核心函数流程分析

2.1.1 Insert 流程

Insert 包含了插入失败时的重构逻辑，流程如下图 2.1 所示，其中提到的子函数会在下文提及。

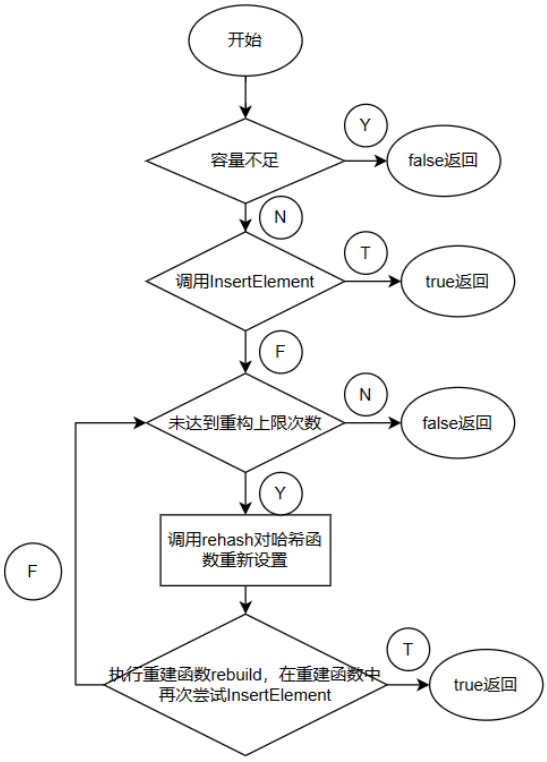

图 2.1 Insert 流程图

2.1.2 InsertElement 函数

该函数包含了直接插入失败时的 Kick Out 逻辑，流程图如图 2.2 所示。

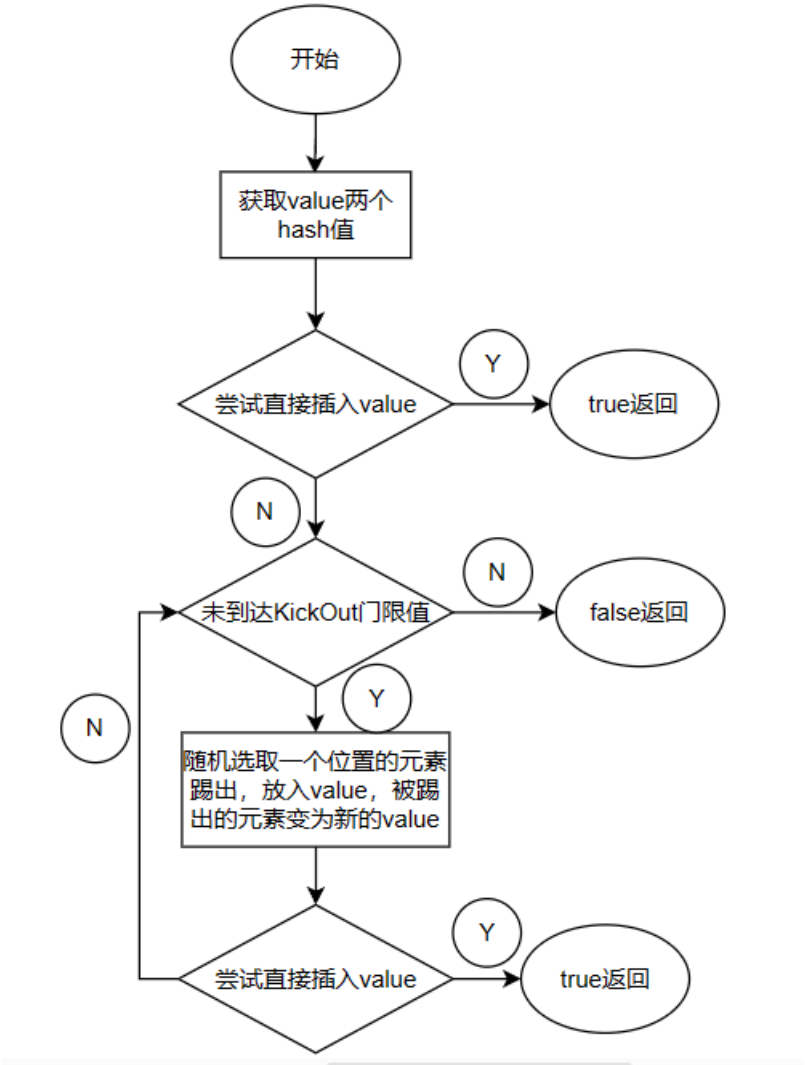

图 2.2 InsertElements 流程图

2.1.3 rebuild 函数

rebuild 函数实现了某个元素 value 插入失败时的重构功能，如下代码所示，需要遍历当前的所有元素，通过复用 InsertElement 函数插入这些元素，当有元素插入失败时，则 false 返回，说明重构失败，需要重新设计 hash 函数。

```
1.  bool CuckooHashingSlots::rebuild(int value) {
2.      memset(units, 0, sizeof(int) * cap());
3.      crash++;
4.
5.      for(int elem : elements)
```

```
            if(!InsertElement(elem)){
                std::cout << elem << " Crash!" << std::endl;
                return false;
            }
        return InsertElement(value);
    }
```

2.1.4 Delete/Search 函数

都是直接获取 hash 值，简单删除/搜索即可，不再赘述。

2.2 子模块测试方法

为了保证各方法实现的正确性，我设计了一些子模块测试的方法，具体而言，代码中维护了一个 ideal_set, 每当执行相应的 insert/delete/search 操作时，ideal_set 也做相应的操作，通过简单比较返回的 true/false 值是否一致（无限循环等情况时可能不一致，已做特殊处理），来大致判定操作是否正确。当然，返回值一致，也不完全等价于各模块实现正确，所以，在固定的一段操作后，会直接遍历 CuckooHash 中的数组，查看里面实际上都有哪些元素，将这个实际的集合和 ideal_set 进行比较，从而便能够很好地验证程序的正确性，保证能够实现 CuckooHash 的设计理念和设计目标。

3 理论分析

由于在本次的实现中，并没有进行指纹的存储，而是直接对 key 进行存储，所以相应的 CuckooHash 不会有插入/删除/搜索时的误判发生，故 false positive 和 false negative 均为 0。当然，这必然给存储器带来了一些负担。

4 性能测试

测试参数如下表 4.1 所示：

表 4.1 测试参数概览

属性	值
Hash 函数个数	2
最大 KickOut 次数	200
最大 Rehash 次数	10
Hash 表总大小	2000000 * 4 bytes
测试数据随机种子	1024

测试将以 slot 为自变量，探究 slot 数为 1/2/4/8 的 CuckooHash 插入和查询性能。值得注意的是，此处 slot 的增加不会引起 hash 表总大小的增加，仍能保证一个有效的恒定存储量，而不是通过提高存储开销来提高性能。

4.1 插入性能

4.1.1 不同空间占有率下的插入时间开销

通过编写 python 脚本获取测试数据并绘图，得到图 4.1 所示的插入时间曲线，可以看到，各曲线的斜率在一开始都十分稳定，说明 CuckooHash 在空间占有率不高时，有着稳定的插入性能，但当到达某个“崩溃阈值”时，插入性能将迅速下降，并很快导致插入完全失败（曲线消失）。当 slot 数提高时，在空间占有率不高时，能够带来一定的性能提升，因为对应的 kickout 路径变短了，但是提升并不十分明显，但是 slot 带来了“崩溃阈值”的上升，使得在较高的空间占有率时，CuckooHash 仍能正常工作。如图 4.2 所示，此处也给出了各占有率下的平均插入时间，也指向类似的结论。

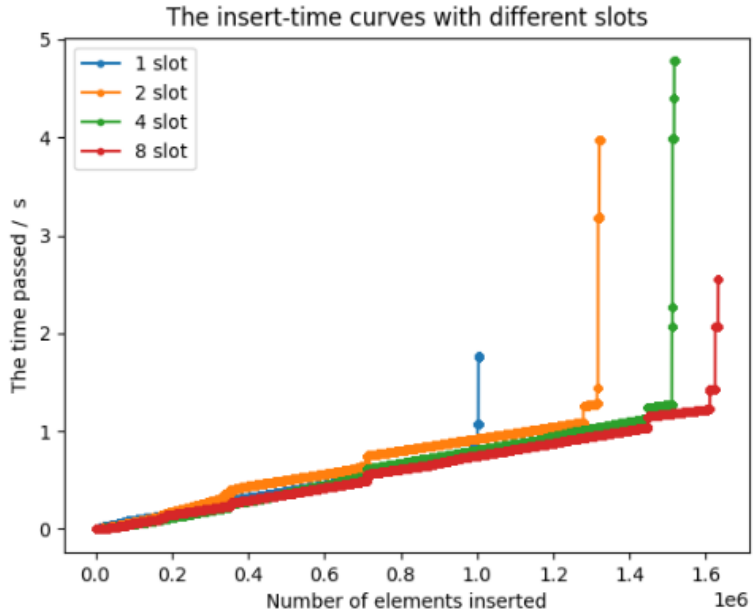

图 4.1 插入过程的时间曲线

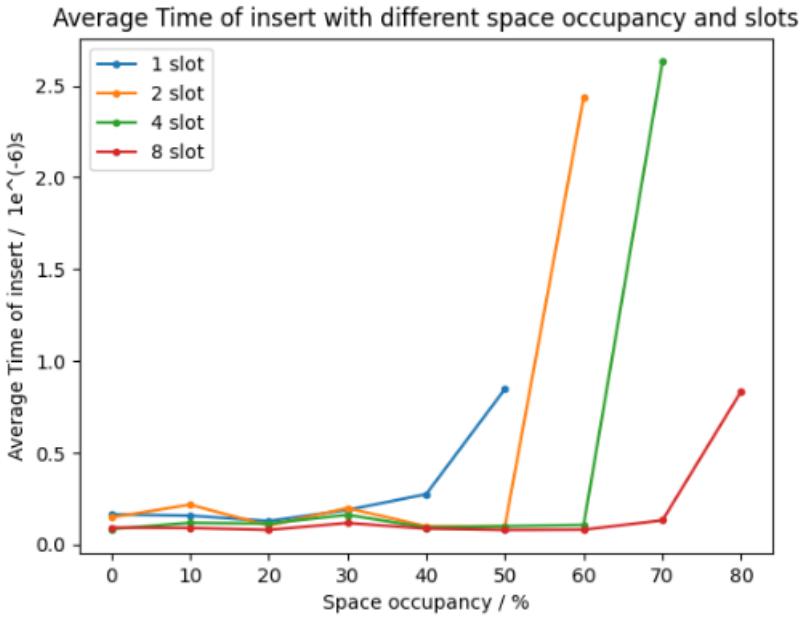

图 4.2 各空间占有率下的插入平均时间

4.1.2 第一次插入失败时的空间占有率

如果在规定的 Rehash 次数中，始终无法插入，则判定为插入失败。在 2e6 次的随机插入中，第一次插入失败的对应序号和空间占有率如下为：

	1slot	2slot	4slot	8slot
序号	1005919	1321534	1518343	1632439
空间占有率	50.30%	66.08%	75.91%	81.62%

可以看到，随着 slot 数目的上升，空间占用率将持续上升，这也反映了，增加 slot 数能够有效降低无限循环的概率。但到了 8slot 时，上升幅度已经明显放缓，所以为了兼顾查询等方面的性能，在本模型中，4slot 是一个较好的选择。

4.2 查询性能

如图 4.3 所示，测试了 2e6 次查询的总时间，随着 slot 数的上升，总时间不断上升，且实际上查询时间与 slot 数正相关，这一点在 slot 数较大时尤为明显。

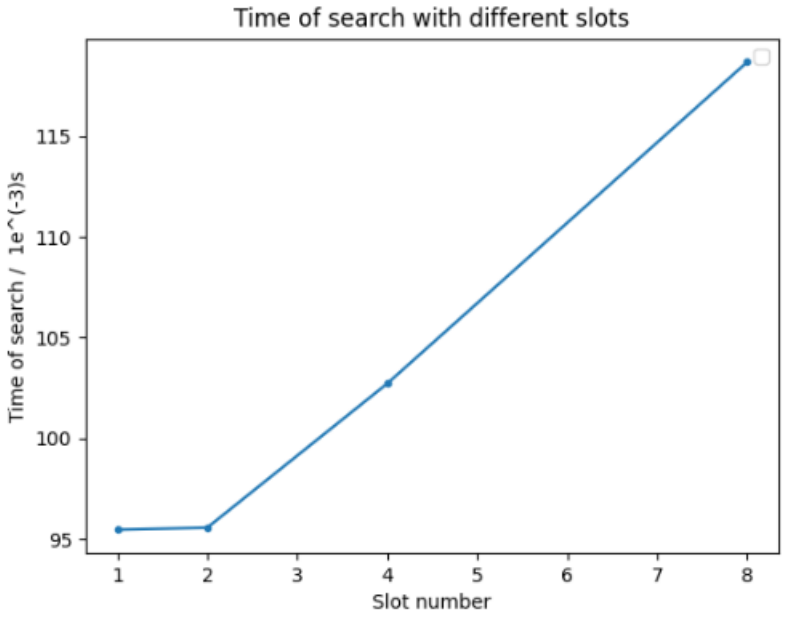

图 4.3 不同 slot 下的查询操作时间