華中科技大學

课程设计报告

题目：Cuckoo-driven Way

课程名称：大数据存储系统与管理

专业班级：校交 1902 班

学　　号：U201916202

姓　　名：杨志军

指导教师：华宇

报告日期：2022.4.18

计算机科学与技术学院

目 录

1 引言

1.1 课题背景与意义

哈希算法常用于数据的插入与查找。理想情况下，哈希算法的时间复杂度为O(1)，算法效率非常高。但传统的哈希算法在解决哈希冲突时采用的线性探测或链表存储会将算法的最坏情况下时间复杂度提升到O(n)。随着数据快速增长，用户对存储系统的查询性能的要求日益增长，传统哈希算法在某些特定场景下已无法满足快速查询的需求。Cuckoo Hash 算法的出现在一定程度上缓解了这个问题。然而，在实际应用中，Cuckoo Hash 算法中存在的多次“踢出”甚至无限循环现象给存储系统带来了很大的不稳定性。国内外许多有关“确定 Cuckoo Hash 中的循环、减少无限循环的概率、有效存储”的研究被积极开展。

1.2 Cuckoo Hash 算法介绍

Cuckoo Hash 是 Hash 算法的一种变体。它在哈希表中为每个元素提供了 n 个候选桶，即候选哈希值。Cuckoo Hash 的插入操作为：若这 n 个候选桶中存在至少一个空桶，即该桶对应的哈希值当前没有哈希冲突，就将该元素哈希到这个桶上；若 n 个候选桶都存在哈希冲突，则该元素将任意一个桶上的当前元素“踢出”并哈希到该桶上，被“踢出”的元素将重新进行上述插入操作，即重哈希。由于 Cuckoo Hash 的插入操作保证了每个元素一定在其 n 个候选桶中，因此查询某个元素时，只需要在其 n 个候选桶中查询，查询效率为 O(1)。

高效的查询效率是以复杂的插入操作作为代价的。插入操作中的“踢出”意味着数据迁移。且被“踢出”的元素在重哈希时，需要注意不能重哈希到原本被“踢出”的位置，以避免两个元素“互踢”使系统进入无限循环；同时，系统还需要注意“踢出”路径不能存在多元素“互踢”，这同样会使系统进入无限循环，也是 Cuckoo Hash 在实际应用中经常需要面临的难题。

1.3 课程设计的主要研究工作

本次课程设计的主要目的是探索一种“确定 Cuckoo Hash 中的踢出循环，减少无限循环的概率，有效存储”的 Cuckoo Hash 变体。我参考了目前广泛应用的

几种 Cuckoo Hash 变体，这些变体都在一定程度对传统 Cuckoo Hash 的弊端进行了优化。由于没有开源的代码，我对这些已经被提出的变体进行了实现，以此作为基础模型，并在此基础上对模型进行修改，尝试探索出一种更良好的变体。

我发觉到基础模型中选择踢出元素时的随机性和盲目性，于是构思出“最少迁移次数启发式选择策略”和“最多空桶启发式选择策略”，并实现了基于这两种踢出元素启发式选择策略的 Cuckoo Hash 模型，对基础模型和两个优化模型进行性能测试，并对测试结果进行比对，结论是在一定使用条件下基于踢出元素启发式选择策略的 Cuckoo Hash 模型的性能相比于基础模型具有巨大的优势。

2 基础模型实现

2.1 模型选择

这部分应该写的是用户需求，明确你做的系统要实现的目标，能处理一些什么样的事务、事务处理流程等。

实验开始之前，我参考了 ChunkStash 和 Necklace 分别提出的两种 Cuckoo Hash 变体。

ChunkStash 中提出的 Cuckoo Hash 中设置了一个单次插入操作中的数据迁移阈值，当“踢出”操作导致的数据迁移次数达到阈值时，就不再尝试将最后被“踢出”的元素继续插入到哈希表中，而是插入另外的辅助链表中。辅助链表的存在会使系统产生额外的空间开销。但是经实践证明，在 Cuckoo Hash 算法参数设置得当的情况下，这个辅助链表的长度会非常短，所占用空间在可接受范围内。我认为这得益于这个 Cuckoo Hash 变体提供了足够多的哈希函数——ChunkStash 系统推荐使用 24 个哈希函数。同时，ChunkStash 使用了两个哈希函数的线性组合来生成 24 个哈希值，因此系统在哈希计算上的时间开销几乎不随着哈希函数数量的增大而提升。

Necklace 的想法为：从一个空桶出发，寻找一条有向路径，这条路径上除了起点为空桶外，剩余点都是非空桶，且终点是需要插入元素的候选桶；我们姑且称这样的路径为增广路径；寻找到一条增广路径后，就可以沿着增广路径移动非空桶上的元素，最终将终点腾空，并将本次需要插入的元素哈希到这个桶上。为了提升算法的速度，需要维护一个有向图，图中的边 a→b 表示当前放在 b 桶上的元素也可以放在 a 桶；若不维护这个有向图，寻找增广路径时，对于每一条边都需要遍历所有非空桶，判断上面的元素是否能到达该桶。维护这样的一个有向图所需的空间复杂度为 O(m*n)。Necklace 算法找到一条增广路径并沿着这条路径插入新元素后，路径上的桶包含的元素发生改变，需要重新更新路径上所有桶在有向图中的边，且插入后原本选择的空桶将变成非空桶，下次寻找增广路径时需要重新选定一个空桶并重新执行算法，此前的工作成果无法复用。我认为这种做法的时间复杂度不亚于以插入元素的候选位置作为起点的广度优先搜索。当哈希函数的个数较大时，算法的复杂度是非常高的。

综上所述，我认为，ChunkStash 适用于哈希函数数量 n 较大的情况，而

Necklace 算法适用于哈希函数数量 n 较小的情况。由于课设完成时间限制，本次课设只能选择其中一个模型进行实现。经过反复斟酌后，我选择实现 ChunkStash 中提出的 Cuckoo Hash 变体作为本次课设的基础模型。

2.2 系统总体设计

- 系统功能

为了观察该Cuckoo Hash变体模型的性能，需要将该模型运用到一个实际的存储系统中并进行插入与查询测试。为此我设计了一个简单的存储系统，这个系统可以将用户输入的字符串存储到哈希表中。在实际运用中，Cuckoo Hash可以用在Key-Value存储系统中，例如ChunkStash中的哈希表存储的每一个元素是一个chunk-id: pointer的键值对。

- 哈希函数

本系统要存储的元素是字符串，因此所设计的哈希函数需要能把字符串映射到整数值上。Cuckoo Hash中哈希函数的数量n决定了每个元素的侯选位置的数量。本系统中的哈希函数的设计参考了ChunkStash中的设计，即Cuckoo Hash中实际只有两个哈希函数$h_1(x)$和$h_2(x)$，但通过两个哈希函数的线性组合生成任意n个哈希函数，即$h_i(x) = h_1(x) + i * h_2(x)\ (i = 1,2,\cdots,n)$。基于该设计，系统每次计算一个元素的n个候选哈希值时，实际只需要计算两个哈希值h1和h2，要计算下一个哈希值时只需让h1自增h2的值即可，从而使在哈希计算上的时间开销不随着哈希函数数量n的增大而线性增大。

- 查询操作

系统对于一个请求查询的字符串，需要遍历该字符串的n个哈希值对应的n个候选桶。若存在一个桶，桶上存储的字符串与输入的字符串相等，则返回查询成功；否则返回查询失败。系统使用模运算将哈希值映射到桶序号上。

- 插入操作

系统对于一个请求插入的字符串，先查询该字符串是否已在系统中。若在，则返回。若不在，需要根据该字符串的n个哈希值以此访问n个候选桶，若找到一个空桶，则将该元素放置在该空桶上，插入操作结束；若所有候选桶都非空，则根据Cuckoo Hash的想法需要选择一个桶上的字符串进行踢出，将当前所要插入的字符串放置在这个桶上，并通过重哈希为被踢出的字符串重新寻找桶，即对被踢出

的字符串重新执行插入操作。这个过程中，有两个地方值得思考：一、应该如何选择被踢出的元素？二、如何避免多次踢出操作过程中可能存在的无限循环？这两个问题分别对应了本次课设重点讨论的**“踢出元素选择策略”**和**“避免循环策略”**。

- 踢出元素选择策略和避免循环策略

这两个策略需要一起讨论的原因是，踢出元素的选择会影响到是否出现循环，因此需要在踢出元素选择策略上来避免循环的出现。同时，踢出元素的选择很可能会影响一次插入中“踢出”操作的次数——若选择策略好，则系统可以更快地结束插入操作。

避免循环踢出，实际上就是不能踢出之前已经被踢过的元素。因此我产生一个朴素的想法：维护一个集合，保存历次被踢出的元素；当选择一个元素进行踢出时，需要保证被踢出的元素不在该集合中。这显然可以用课程中介绍的Bloom Filter来解决，且Bloom Filter算法中可能存在的False Positive在这个使用背景下几乎没有不良影响——当一次查询出现False Positive时，即查询元素本不在集合中，但算法返回的结果是元素在这个集合中，这会导致系统误以为选择这个元素进行踢出会造成循环，进而不选择这个元素，并考虑下一个元素。在这个系统中使用Bloom Filter可以带来高效的查询效率、集合维护效率、空间使用效率。实际实现时，由于集合中的元素个数不会超过系统设置的重哈希次数阈值（在ChunkHash中这个阈值被设置为100），集合中的元素个数是比较少的，所以我没有用Bloom Filter，而使用了C++ STL中的unorder_set，其底层实现同样是哈希算法，在元素较少的情况下同样可以达到常数级复杂度的查询效率。

ChunkStash中没有提到其Cuckoo Hash变体使用的踢出元素选择策略。因此这部分由我个人设计。首先容易想到的是，选择所有候选桶中**第一个**不在“历次踢出元素集合”中的元素进行踢出。起初我对这种设计有一些顾虑，由于每次选择的行为都是固定的，即优先考虑$h_i(x)$中i较小的桶，会不会导致一次插入操作的踢出次数增加。实际上这个问题是不存在的，因为哈希函数的复杂性，同一个元素的n个哈希值之间没有绝对的大小关系，即第一个哈希值经过模运算后可以出现在哈希表中的较低位置，也可以出现在较高位置，可以实现哈希表的负载均衡；任意两个不相同的元素的n个哈希值是大相径庭的，因此对不同元素x优先考虑$h_i(x)$中i较小的桶实际上不会产生有固定位置规律的踢出行为，不会给系统带

来不良影响。因此，要踢出某个元素b以插入输入元素a时，选择的策略为依次访问$h_i(a)$ $(i = 1,2,\cdots,n)$，若$h_i(a)$上的元素不在“历次踢出元素集合”中，则踢出这个元素。另外一种策略是随机生成一个i的初值，并以该初值为起点依次遍历n个候选桶，如i的初值为6，则遍历顺序为$h_i(a)$ $(i = 6,7,\cdots,n,1,2,\cdots)$。实际测试结果证明，这两种策略的效果几乎一致，这也证明了我们此前顾虑的问题是不存在的。

- 辅助链表

ChunkStash系统为了避免Cuckoo Hash踢出操作时陷入无限循环，设置了一个踢出次数阈值，当踢出次数达到该阈值时就将最后被踢出的元素插入到辅助链表中。这会导致系统产生额外的空间开销，也会给降低查询效率——当某个被查询的元素在哈希表中不命中时，需要到辅助链表中继续查找。但是经过实际证明，在合适的系统参数设置的前提下，辅助链表的长度是非常短的，因此对系统的不良影响很小。

尽管我在踢出操作过程中维护了一个集合以避免陷入无限循环，但是为了保证系统踢出操作次数上限和集合所占空间大小的可控制性、系统行为的可预测性，我保留了踢出次数阈值和辅助链表的设计。

需要注意的是，除了踢出次数达到阈值时需要将最后踢出的元素加入到链表内外，还有一种情况需要考虑，即最后被踢出的元素的所有候选位置上的元素都在本次插入操作中被踢出过，则该元素没有可选择的踢出对象，需要将该元素加入到辅助链表中。

2.3 系统实现

本次课程设计我使用C/C++作为实现语言。编程实现采用面向对象的思想，将一个根据Cuckoo Hash算法维护的哈希表封装成一个类，其哈希表大小size、哈希函数个数hashFuncNum、踢出次数阈值maxRelocateNum需要在调用构造函数时传入，构造函数将返回一个符合上述设置的Cuckoo Hash Table对象。这么做可以方便后续测试和调整。代码的编写实现严格按照系统设计思路，此处不再赘述。实现环境如下。

CPU：12th Gen Intel(R) Core(TM) i7-12700H

内存：LPDDR5 16GB

操作系统：Windows 11

GCC版本：9.2.0 (tdm64-1)

2.4 系统测试

本次课设的系统测试分为正确性测试和性能测试。正确性测试即为测试系统对于一系列的输入流能否正确插入和查找。性能测试主要测试系统在不同负载系数下的插入延迟、平均单次插入操作的数据迁移次数，并观察辅助链表的长度。

- 正确性测试

为了测试系统对于一系列输入流能否正确执行插入或查找操作，我设计并实现了正确性测试。测试思路为：提前设置好负载系数Load Factor，构造出一个包含Load Factor * size个互不相同字符串的插入测试集；并构造出一个查询测试集，它是插入测试集的超集，不同点在于每隔3个元素插入一个不在插入测试集中的字符串，使用这个测试集可以判断出系统能否对于不同元素返回正确的查询结果。

为了更好地观察测试结果，我在代码中添加了一些输出文本，生成系统操作日志，下列几种情况都会被输出：1.一个元素插入到哈希表中；2.一个元素被踢到了辅助链表中；3.当前查找的元素在哈希表中；4.当前查找的元素在辅助链表中；5.当前查找的元素不在系统中。同时，我将程序的输出重定向到文件中，以方便观察测试结果。

由此查询测试集是由插入测试集每个3个元素插入一个新元素构成的，因此若系统的插入、查询功能正确，查询结果的输出应以“3个查询成功、1个查询失败”的模式循环。测试结果是否正确可以通过输出的日志来观察，同时我也编写程序对系统查询时的输出轨迹进行了跟踪，可以自动化验证查询结果是否符合预期，若符合预期，测试结束后将输出”0 error”；否则会输出”query result error”并终止测试。

因为程序输出会拖慢程序运行的时间，因此正确性测试时测试规模不宜太大。在本次测试中，我设置的参数如下。根据设置的参数，插入测试集的规模为9100，查询测试集的规模为12133。测试结束后，观察输出文本文件output.txt，观察系统生成的操作日志，可以观察到结果符合预期，系统功能实现正确。

{size=10000, hashFuncNum=24, maxRelocateNum=100, load factor=0.91}

```
21225   9134 in hash table.
21226   9135 in hash table.
21227   9136 in hash table.
21228   19136 not found.
21229   9137 in hash table.
21230   9138 in hash table.
21231   9139 in hash table.
21232   19139 not found.
21233   9140 in hash table.
21234   0 error.
```

- 性能测试

性能测试主要测试系统在不同负载系数下的插入延迟，并观察辅助链表的长度。为了确保性能测试结果不被文本输出影响，在进入性能测试时程序进入“静默模式”，即测试过程没有任何输出，测试结束后输出插入延迟(ms)、平均单次插入操作的数据迁移次数、辅助链表的长度。为了更好地测试性能，性能测试时可以扩大测试集规模。在size=1e7，hashFuncNum=24, maxRelocateNum=100时，我分别在最大负载系数为0.91, 0.94, 0.97的情况下进行测试，测试结果如下。

```
{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.91 }
Total run time: 11878
Total relocate num: 45615
Relocations per insert: 0.00501264
Length of auxiliary linked list: 0
```

```
{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.94 }
Total run time: 12276
Total relocate num: 110488
Relocations per insert: 0.011754
Length of auxiliary linked list: 0
```

```
{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.97 }
Total run time: 13040
Total relocate num: 297752
Relocations per insert: 0.0306961
Length of auxiliary linked list: 0
```

可以观察到系统插入将近1e7（一千万）个元素时仍然可以在十几秒左右的时间完成，并且每次插入操作的平均迁移次数也非常小。观察到辅助链表的长度始终为0，这是因为系统的哈希函数数量和最大踢出阈值设置得足够大。若将哈希函数数量设置为8，最大踢出阈值减少为30，则可以观察到开始有24个元素被踢入辅助链表。相比于数量接近于1e7的系统插入的元素个数，辅助链表的长度是很小的。

```
{ size=10000000  hashFuncNum=8  maxRelocateNum=30  load factor=0.91 }
Random selection strategy.
Total run time: 8745
Total relocate num: 1090857
Relocations per insert: 0.119874
Length of auxiliary linked list: 24
```

3 模型优化探索

3.1 优化思路

在基础模型的设计中，我提到**“踢出元素选择策略”**和**“避免循环策略”**可能是模型优化的方向。

在基础模型的“避免循环策略”中，我通过维护一个“历次踢出元素集合”并且在每次选择踢出元素时确保该元素之前未被踢出过，从而避免循环踢出操作。可以使用Bloom Filter优化这个过程——选择一个元素进行踢出时，查询该元素是否在Bloom Filter内；若不在，则将该元素踢出，并加入到Bloom Filter中；若在，则考虑下一个元素。且Bloom Filter算法中可能存在的False Positive在这个使用背景下几乎没有不良影响——当一次查询出现False Positive时，即查询元素本不在集合中，但算法返回的结果是元素在这个集合中，这会导致系统误以为选择这个元素进行踢出会造成循环，进而不选择这个元素，并考虑下一个元素。使用Bloom Filter可以使系统的空间使用效率、运行时间效率得到一定程度提升。

在基础模型的“踢出元素选择策略”中，我采用的选择策略是：依次访问$h_i(a)\ (i = 1,2,\cdots,n)$，若$h_i(a)$上的元素不在“历次踢出元素集合”中，则踢出这个元素。这个方法的优点是：由于每次考虑的元素a不同，因此基于采用的哈希函数值的均匀性，被踢出的元素的位置是均匀分布的，哈希表具有负载均衡的表现。但缺点也很明显，这个选择策略具有一定的盲目性。因此我在思考如何优化时尝试探索出一个具有一定指导性、并同样能够使哈希表负载均衡的“踢出元素选择策略”。经初步构思，我产生了两个优化思路：一、每次选择迁移次数最少的元素进行踢出；二、每次选择剩余空桶数量最多的元素进行踢出。由于这个选择策略具有一定的指导性，姑且称之为**“最少迁移次数启发式选择策略”**和**“最多空桶启发式选择策略”**。优化灵感来源很朴素：若一个元素的迁移次数非常多，那么与它相关的踢出路径可能会比较长，应尽量避开选择该元素；若一个元素的剩余空桶较多，那么与它相关的踢出路径可能会比较短，应尽量选择该元素。

由于课设时间有限，我在“避免循环策略”优化和“踢出元素选择策略”优化中只选择了更具有挑战性的后者进行实现。

3.2 优化设计

该节只针对踢出元素选择策略的设计进行优化。系统的其他组成部分与基础模型一致，该节不再赘述。

- 最少迁移次数启发式选择策略

若一个元素的迁移次数非常多，那么与它相关的踢出路径可能会比较长，应尽量避开选择该元素。考虑了迁移次数的元素踢出选择策略可能会带来性能优化。为此，我们需要维护每个元素的迁移次数。我们构建一个辅助表，表的容量与哈希表相等，表上的位置记录的是哈希表上同一个位置的元素的迁移次数。当哈希表上元素被踢出时，对应的该辅助表上的值也要被踢出；所有元素的初始迁移次数计数器的值都是0，当元素被踢出时，计数器自增1；当某个元素放到哈希表上某个位置时，对应的该元素的迁移次数也要放置在辅助表上对应的位置。

当我们在选择踢出的元素时，可以选择迁移次数最少的元素进行踢出。这种选择策略的有四个好处：一、具有一定的指导性，不盲目；二、灵感来源朴素，直观上可能可以产生优化效果；三、易于实现；四、每次选择最小值，具有迁移次数负载均衡的效果。

设计时还需要考虑的一点是，这个策略会不会给程序带来无法接受的额外的运行空间和时间开销。实际上并不会。首先考虑额外的空间开销，从基础模型的测试中可知，平均每个插入元素的迁移次数是非常小的，因此为每个元素维护的计数器的大小不需要太大，比如可以给一个计数器分配四个比特，这个计数器可以表示0~15的迁移次数；为了防止溢出，只需在程序发现某个元素的所有候选位置上的元素的迁移次数的最小值为15时，对所有元素的计数器值进行一次减值调整操作（最极端地，可以直接清零）即可；实际上由于这个方法具有负载均衡的特性，计数器的值几乎不可能发生溢出；其次考虑额外的时间开销，在基础模型的随机选择策略中，只需找到第一个不在“历次踢出元素集合”中的元素就可以确定所选元素，而“最少迁移次数”选择策略需要完整遍历输入元素a的所有候选位置上的元素的迁移次数才能确定选择元素，但实际系统中由于哈希函数的个数一般只有数十个，因此每次执行这个选择策略所需的时间开销并不会太大。

- 最多空桶启发式选择策略

若一个元素的剩余空桶较多，那么与它相关的踢出路径可能会比较短，应尽

量选择该元素。但是，一个元素插入到某个位置后，它的插入动作很有可能会使先前已经插入到系统中的其他元素的空桶数量减少。如果我们要维护准确的元素拥有的空桶数量的话，就遇到了Necklace中类似的问题：需要确定以某个位置作为候选桶的所有元素的集合。我认为这实际上会带来较大的额外时间和空间开销。因此，我不维护准确的元素拥有的空桶数量，只维护元素在插入时拥有的空桶数量，**这实际上是一种折中，希望以较小的额外开销为代价获得一定程度的优化，即使优化的程度可能不大**。因此维护元素拥有空桶数量和基于元素拥有空桶数量选择踢出元素方法为：插入某个输入元素时，遍历该元素的所有候选桶，记录其中空桶数量和第一个空桶；若空桶数量非0，则将该元素放在第一个空桶上，并且记录该元素拥有的空桶数量；若空桶数量为0，则遍历输入元素所有侯选位置上的元素，找到空桶数计数值（尽管可能并不准确）最大的元素进行踢出，将输入元素放置在被踢出元素的位置，并设置该输入元素拥有空桶数为0。

考虑这种策略的额外空间开销，由于每个元素的最大空桶数量就是哈希函数数量n，因此为每个元素维护的空桶数量计数器只需分配$\lg(n+1)$个比特。例如在有15个哈希函数的Cuckoo Hash系统中，一个计数器的大小为4 bit。接着考虑这种策略的额外时间开销，这种策略在插入到空桶前，为了维护拥有空桶数量，需要遍历所有候选桶，而不是碰到第一个空桶就结束遍历；若没有空桶，则需要遍历所有候选桶上的元素，选择拥有空桶数量最多的元素进行踢出；以上两步可以合并在一起执行；由于实际系统中哈希函数的数量只有数十个，因此策略的每次执行带来的额外时间开销并不大。

在实际实现中，为了与上一个策略中的“最少”对应，我实际维护的是每个元素“非空桶”的数量，则策略中的操作相应地取反，例如“选择空桶最多的元素进行踢出”对应“选择非空桶最少的元素进行踢出”。

3.3 优化实现

实现语言、配置、环境与基础模型一致，此处不再赘述。在分别实现了上述两个启发式策略后，我分别实现了基于这两个策略的插入函数。加上基础模型中的一个插入函数，现在共有三个插入函数。这么做的目的是方便系统测试，要测试某个策略的性能只需调用基于该策略的插入函数。

3.4 系统测试

优化方案实现后同样要经过正确性测试和性能测试。本节的重点是基于三种踢出元素选择策略实现的插入操作的性能比较。

- 正确性测试

正确性测试的方法和基础模型一致，由于不是本节重点，此处不再赘述。测试的结果是两个优化方案均通过了正确性测试。

- 性能测试

性能测试的方法为，设置多组Cuckoo Hash系统参数，对于每一组参数的系统，都使用**同样的测试集**对三种插入方法进行性能测试，对结果进行比对和分析。首先我们在哈希表大小为1e7，哈希函数数量为24，踢出次数阈值为100，最大负载系数分别为0.91, 0.94, 0.99的三组参数设置中，测试三个模型。

{size=1e7, hashFuncNum=24, maxRelocateNum=100, load factor=0.91}测试结果如下图所示。测试结果中，三个模型的插入延迟相差不多，但“最多空桶选择策略”模型的数据迁移次数明显比其他两个模型少（比基础模型少了2.96%）。

```
{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.91 }
Random selection strategy.
Total run time: 11302
Total relocate num: 45568
Relocations per insert: 0.00500747
Length of auxiliary linked list: 0

{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.91 }
Minimum relocations selection strategy
Total run time: 11664
Total relocate num: 45520
Relocations per insert: 0.0050022
Length of auxiliary linked list: 0

{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.91 }
Maximum empty bucket selection strategy.
Total run time: 12654
Total relocate num: 44217
Relocations per insert: 0.00485901
Length of auxiliary linked list: 0
```

{size=1e7, hashFuncNum=24, maxRelocateNum=100, load factor=0.95}测试结果如下图所示。在相差较小的插入延迟的条件下，“最多空桶选择策略”模型依然展现出了更少的数据迁移次数（比基础模型少6.79%）。

```
{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.95 }
Random selection strategy.
Total run time: 12770
Total relocate num: 152205
Relocations per insert: 0.0160216
Length of auxiliary linked list: 0

{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.95 }
Minimum relocations selection strategy
Total run time: 12446
Total relocate num: 150648
Relocations per insert: 0.0158577
Length of auxiliary linked list: 0

{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.95 }
Maximum empty bucket selection strategy.
Total run time: 13399
Total relocate num: 141870
Relocations per insert: 0.0149337
Length of auxiliary linked list: 0
```

{size=1e7, hashFuncNum=24, maxRelocateNum=100, load factor=0.99}测试结果如下图所示。在这组测试中，两个优化模型都在更少的迁移次数下完成插入操作。其中“最少迁移选择策略”模型比基础模型少12.79%，“最多空桶选择策略”模型比基础模型少21.19%。

```
{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.99 }
Random selection strategy.
Total run time: 13846
Total relocate num: 825819
Relocations per insert: 0.0834161
Length of auxiliary linked list: 0

{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.99 }
Minimum relocations selection strategy
Total run time: 14369
Total relocate num: 720181
Relocations per insert: 0.0727456
Length of auxiliary linked list: 0

{ size=10000000  hashFuncNum=24  maxRelocateNum=100  load factor=0.99 }
Maximum empty bucket selection strategy.
Total run time: 15518
Total relocate num: 650836
Relocations per insert: 0.065741
Length of auxiliary linked list: 0
```

从测试结果的趋势可以看出，随着负载系数逐渐增大，我们的优化模型的优越性也逐渐体现。这其实可以从优化的设计直观地解释：我们优化的愿想是通过在可接受范围内的额外计算，在选择踢出元素时尽量减小进入复杂踢出路径的可能。当负载系数逐渐增大时，哈希表中越来越拥挤，踢出路径越来越长，此时我

们的踢出元素启发式选择策略的作用越来越明显。

按照这个想法，我们试图让哈希表中的元素感觉系统更拥挤——我们减少哈希函数数量和踢出次数阈值，在{size=1e7, hashFuncNum=6, maxRelocateNum=30, load factor=0.91}的参数设置下进行测试，测试结果如下。

```
{ size=10000000  hashFuncNum=6  maxRelocateNum=30  load factor=0.91 }
Random selection strategy.
Total run time: 70885
Total relocate num: 3028881
Relocations per insert: 0.332844
Length of auxiliary linked list: 13441

{ size=10000000  hashFuncNum=6  maxRelocateNum=30  load factor=0.91 }
Minimum relocations selection strategy
Total run time: 10047
Total relocate num: 1896671
Relocations per insert: 0.208425
Length of auxiliary linked list: 12

{ size=10000000  hashFuncNum=6  maxRelocateNum=30  load factor=0.91 }
Maximum empty bucket selection strategy.
Total run time: 9787
Total relocate num: 1437608
Relocations per insert: 0.157979
Length of auxiliary linked list: 0
```

测试结果让我喜出望外。可以观察到，在这组参数设置下，**原始模型的运行时间已经慢到了70秒，而我们的两个启发式优化模型的运行时间分别为10秒和9.7秒，甚至比此前哈希函数数量为24、踢出次数阈值为100的系统还要快**。相比于原始模型，两个启发式优化模型的**时间开销减少了85.83%和86.19%，迁移次数减少了37.38%和52.54%**。另外，**原始模型的辅助链表长度达到了13441，而两个优化模型的辅助链表长度分别为12和0**。

惊喜之余，我们还要考虑一个问题，为什么哈希函数数量和踢出次数阈值减少后，优化模型的运行时间更快了呢？我认为原因是：在优化设计中我们提到，优化模型在使用启发式策略选择踢出元素时，需要遍历其所有候选空桶，因此其单次踢出元素的选择花费在计算上的时间比原始模型更多，优化模型的性能需要靠“数据迁移次数少”拉回优势；当哈希函数数量减少后，每个元素的候选空桶数量少了，单次踢出元素的选择花费在计算上的时间也随之变少，若此时数据迁移次数不至于增加太多，那么插入延迟将大幅度减少；**而且，当每个元素拥有的候选桶变少后，它将以更小的概率和其他元素产生哈希冲突，这将导致后续插入哈希表中的元素将在更小规模地影响已在哈希表中的元素的剩余空桶数，使**

我们维护的“近似空桶数”更加接近真实值，使“最少空桶启发式选择策略”更具有正确的指导性。

进一步增大负载系数，在{size=1e7, hashFuncNum=6, maxRelocateNum=30, load factor=0.95}的参数设置下进行测试，基础模型未能在180s内完成插入操作，因此没有其测试结果数据。两个优化模型的测试结果如下图所示，插入延迟分别为13.424s和11.959s，其中“最多空桶选择策略”模型的辅助链表长度仅为86，若增大踢出次数阈值，辅助链表中的元素将进一步减少。

```
{ size=10000000  hashFuncNum=6  maxRelocateNum=30  load factor=0.95 }
Minimum relocations selection strategy
Total run time: 13424
Total relocate num: 3348422
Relocations per insert: 0.352465
Length of auxiliary linked list: 898

{ size=10000000  hashFuncNum=6  maxRelocateNum=30  load factor=0.95 }
Maximum empty bucket selection strategy.
Total run time: 11959
Total relocate num: 2452975
Relocations per insert: 0.258208
Length of auxiliary linked list: 86
```

综合以上几个性能测试结果，我初步得出结论：基于“踢出元素启发式选择策略”的Cuckoo Hash模型可以以“可接受范围内的额外计算和空间开销”为代价，达到“以更短的踢出路径完成插入操作”的目的；在Cuckoo Hash的哈希表拥挤时，优化模型相比于基础模型的性能优越性更加突出。

4 总结与展望

4.1 全文总结

在本次课程设计中，我首先参考了几个 Cuckoo Hash 变体模型，并选择了 ChunkStash 中的模型作为本次课设的基础模型。我在基础模型中引入并维护了一个“历次踢出元素集合”，来避免插入操作出现循环踢出现象。我编写了正确性测试程序和性能测试程序，并使用这两个测试程序对基础模型进行测试。

接着，我开始探索模型的优化方法。在**“避免循环策略”**方向，我提出用 Bloom Filter 来维护集合从而优化系统。在**“踢出元素选择策略”**方向，我提出了**“最少迁移次数启发式策略”**和**“最多空桶数量启发式策略”**，并实现了基于这两种启发式策略的模型。两个新模型通过了正确性测试后，我对基础模型和两个新模型进行性能测试，并对比分析三个模型的性能，得出的结论是：基于“踢出元素启发式选择策略”的 Cuckoo Hash 模型可以以“可接受范围内的额外计算和空间开销”为代价，达到“以更短的踢出路径完成插入操作”的目的；在 Cuckoo Hash 的哈希表拥挤时，优化模型相比于基础模型的性能优越性更加突出。

4.2 工作展望

由于在本次课设中，我设计并实现的优化模型在一定条件下展现出了极大的优越性，因此在今后的工作中，我希望对此研究进行进一步的深入开展。例如，将模型应用到实际的云存储系统中，观测模型的表现。如果优化模型确实具有很大的优势，我希望能够发表相关论文。

5 体会

在本次课程设计中，我的工作历程可以概括为“阅读相关文献——实现文献基础模型——设计模型优化思路——实现优化模型——新旧模型性能测试与比对”。在过程中，我初步体会到了科学研究的流程与感受。本次课程设计的经历为我今后的学习生活打下了一定的基础，提供了可参考的经验。

虽然展现在报告上的结果是优化后的模型具有很强的优越性，但是我在实际实验过程中，由于一开始在哈希表较稀疏的条件下进行测试，得到的结果是优化模型与基础模型性能相差无几。优化设计时的激情和看到结果后的失落形成强烈反差，个中滋味只有自己知道。在我即将放弃优化模型的时候，我开始观察到优化模型相比于基础模型有着更少的迁移次数，并开始调整我的性能测试策略，在更拥挤的哈希表中进行性能测试，才有了最终的令我感到意外的优秀结果。从中我也感受到，科学研究的进展有时就是从一个小小的细节中慢慢挖掘出来，我们在进行科学研究时，应该时刻保持敏锐的眼光，从而及时发现研究的突破口。

参考文献

[1]Debnath B, Sengupta S, Li J. {ChunkStash}: Speeding Up Inline Storage Deduplication Using Flash Memory[C]//2010 USENIX Annual Technical Conference (USENIX ATC 10). 2010.

[2]Li Q, Hua Y, He W, et al. Necklace: An efficient cuckoo hashing scheme for cloud storage services[C]//2014 IEEE 22nd International Symposium of Quality of Service (IWQoS). IEEE, 2014: 153-158.

[3]Hua Y, Jiang H, Feng D. FAST: Near real-time searchable data analytics for the cloud[C]//SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2014: 754-765.