華中科技大學

课 程 实 验 报 告

课程名称： 物联网数据存储与管理

选题名称： Cuckoo-driven Way

专业班级：物联网 1801 班

学　　号：U201814500

姓　　名：王英嘉

指导教师：华宇

报告日期：2021 年 6 月 21 日

计算机科学与技术学院

目 录

摘要

Cuckoo Hashing，又称布谷鸟哈希，是一种查询操作非常高效的哈希方法，适合对读性能要求高、写性能要求低的实际场景。本文基于传统 Cuckoo Hashing 进行了研究，结合 Cuckoo Hashing 在 Load Factor 达到 50%时就会导致无法插入的问题，提出了一种 Dense Cuckoo Hashing 结构，在相同的空间占用情况下，通过扁平化哈希散列，牺牲一部分查询和删除效率，可以有效降低插入中无限循环和重哈希的概率，并显著提高最大空间利用率。

关键词：Cuckoo Hashing, Hash

1 选题背景与意义

目前已经有许多提出并证明比较高效的哈希算法，如链式哈希、Cuckoo Hashing 等等。其中链式哈希结构如图 1 所示，链式哈希把所有发生哈希碰撞的元素通过链表连接在一起保存在一个桶中，因此这种结构的优点在于插入很快，且空间利用率比较高，但由于链表的空间不连续，导致每次查询或删除可能需要遍历链表。

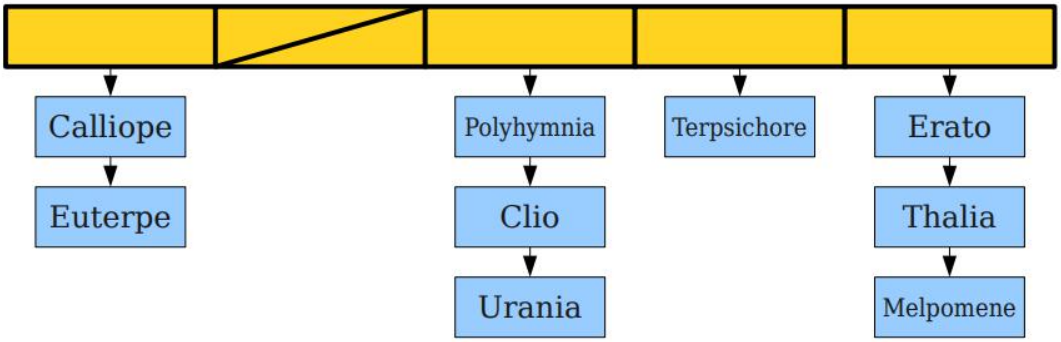

图 1 链式哈希结构

Cuckoo Hashing 结构如图 2 所示，一般使用两个哈希函数 h1(x)和 h2(x)允许 x 映射到两个哈希散列中的对应位置，与链式哈希相比，查询和删除最多只需要检查两个位置的元素，但 Cuckoo Hashing 也有一些缺点，例如插入过程很可能因为碰撞链过长而导致插入效率低下，甚至反复发生无限循环和重哈希过程。通过图算法可以证明在 Load Factor 达到 50%时，就很难再插入元素，因此 Cuckoo Hashing 的空间利用率比较低。

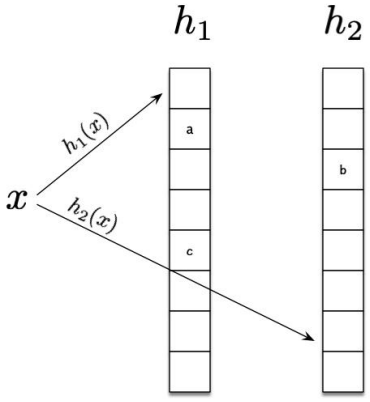

图 2 Cuckoo Hashing 结构

本文综合考虑了链式哈希和 Cuckoo Hashing 的优点和缺点，在 Cuckoo Hashing 的基础上，提出了一种 Dense Cuckoo Hashing，将一维结构扁平化成多维结构，实验证明 Dense Cuckoo Hashing 与 Cuckoo Hashing 相比性能损失很小，最大空间利用率在维度为 4 时可以提升到 76%左右，在维度为 16 时可以提升到 85%左右。

2 整体设计

2.1 Cuckoo Hashing 设计

图 3 展示了 Cuckoo Hashing 的一般结构，一般使用两个哈希函数将元素映射到两个哈希散列中的对应位置中，因此每一个元素只可能存在于两个可能位置。

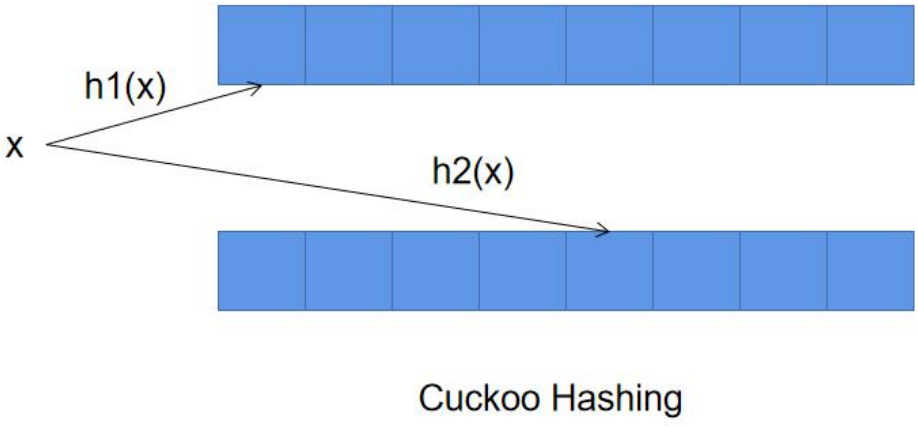

图 3 Cuckoo Hashing 结构

对于查询和删除过程，Cuckoo Hashing 只需要检查两个哈希散列中对应位置的元素。

对于插入过程，相对比较复杂，Cuckoo Hashing 首先检查两个哈希散列中对应位置是否存在空位，如果存在，则将元素插入，如果不存在空位，则将某一个位置的元素踢走到它在另一个散列中对应的位置，当然如果被踢走的元素到达的新位置仍然有元素，则循环此过程，直到被踢走的元素找到空位插入或无限循环。

举例如图 4 所示，当 x 插入时，两个位置均有空位，则任选一个位置插入；当 y 插入时，发现其中一个位置 z 已经有了元素，但另一个位置有空位，于是插入到另一个位置；当 m 插入时，发现两个位置都有元素，于是随机踢走元素 y，

并插入到 y 的位置，y 踢走另一个哈希散列中位置上的 z，并插入到 z 的位置；z 最终插入到 z 的另一个空位中。

至于无限循环的问题，综合考虑时空因素，一般在上述踢元素过程循环一定次数之后即认为出现了无限循环，这样只需要一个计数器就能实现，在本文这一参数取为 200。

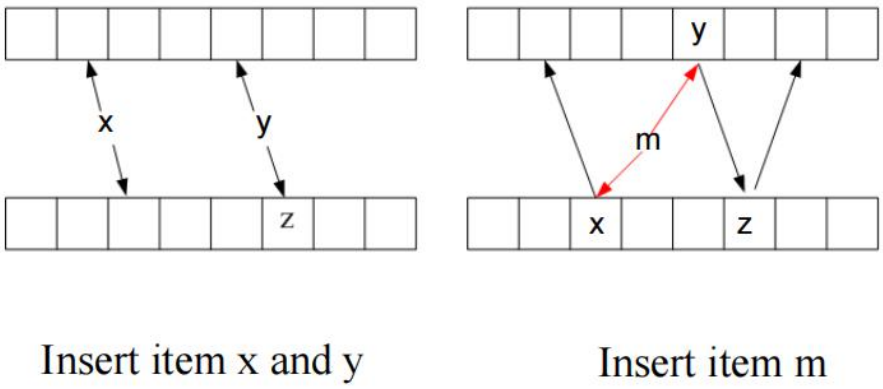

图 4 Cuckoo Hashing 插入举例

当出现无限循环时，需要进行重哈希，重哈希的流程说明如下：

（1）使用随机数重新生成哈希函数的参数，并清空两个散列中的所有元素

（2）根据保存的全部元素集合将集合里的元素重新依次哈希进散列中

（3）如果（2）中又出现了冲突，则回到（1）重新进行，直到（2）成功或超过了最大重哈希次数，本文这一参数取为 100，即重哈希超过了 100 次即认为将无法再插入。

Cuckoo Hashing 的性能与哈希函数的设计密切相关，好的哈希函数可以使元素分布地更均匀，发生无限循环的概率更小。

2.2 改进后的 Dense Cuckoo Hashing 设计

Dense Cuckoo Hashing 的结构如图 5 所示，Dense Cuckoo Hashing 仍然使用两个哈希散列和两个哈希函数，区别在于哈希散列的维度从一维增加到了多维，也可以看作哈希散列中的每一个位置有了多个 slot。设原来的哈希散列长度为 m，新维度为 n，在保证最大空间相同的情况下，现在只需要长度为 m/n（假设 m 是 n 的倍数）。

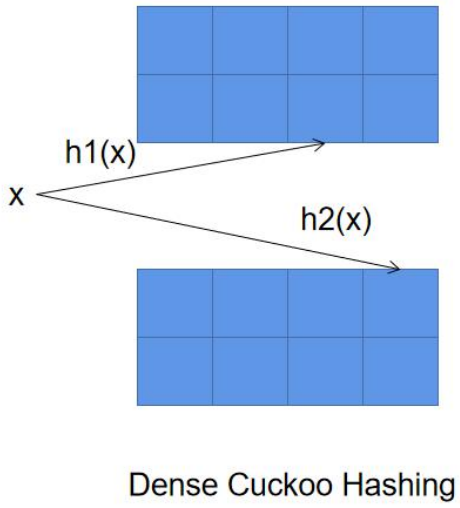

图 5 Dense Cuckoo Hashing 结构

对于查询和删除过程，Dense Cuckoo Hashing 需要检查两个哈希散列中对应位置的 slots，对于每个 slots 可能需要遍历每个 slot 直到查询成功或删除成功；

对于插入过程，Dense Cuckoo Hashing 需要检查两个哈希散列中对应位置的 slots 是否还有空位，如果有空位则选择一个空位插入，如果没有空位则踢出一个 slot 的元素，并将其踢到它在另一个哈希散列对应位置的 slots 中，如果这个 slots 仍然没有空位，则循环此过程。

同理，Dense Cuckoo Hashing 也有概率出现无限循环，但是由于每个位置有多个 slot 可以存放元素，出现无限循环的概率大大降低。

3 理论分析

Cuckoo Hashing 和 Dense Cuckoo Hashing 查询、插入和删除的平均时间复杂度和最坏时间复杂度对比分析如表 1 所示。

表 1 改进前与改进后操作时间复杂度对比分析

	平均查询	最坏查询	平均插入	最坏插入	平均删除	最坏删除
改进前	O(1)	O(1)	O(1)	/	O(1)	O(1)
改进后	O(1)	O(s)*	O(1)	/	O(1)	O(s)*

* s 为 Dense Cuckoo Hashing 中的维度

对于传统 Cuckoo Hashing，插入过程时间复杂度的期望为 O(1) (with a high probability)，主要的时间开销发生在当 Load Factor 接近理论最大值时，元素插入时由于碰撞链较长会发生大量数据移动、甚至重哈希过程；改进后的 Dense Cuckoo Hashing 主要在优化这一问题，表 2 中的 s 一般情况下仅为一个比较小的常数，比如 4 或 16，最坏查询和删除的时间复杂度仍然保持在一个常数级，但是却可以大量减少数据碰撞和重哈希概率，使得 Cuckoo 在运行过程中更加稳定可靠。

4 性能测试与分析

4.1 性能测试

性能测试环境如表 2 所示。

表 2 性能测试环境

操作系统	Ubuntu 20.04 LTS
CPU 版本	Intel® Core™ i7-8550U CPU @ 1.80GHz × 8
g++版本	9.3.0
cmake 版本	3.16.3

定义测试数据规模为 n，Cuckoo 每一个散列长度为 m，散列位置维度为 s（传统 Cuckoo Hashing 视为 s = 1），因此 Load Factor = 2ms / n。

分别在表 3 中的 1-6 条件下测试 Load Factor 的最大值，测试过程从 Load Factor = 25%开始，每次增加 5%，直到测试失败时回到上一次成功的 Load Factor 并每次增加 0.5%，直到确定 0.5%粒度下的最大值。

对于每一次测试，测试流程分为四个步骤：

（1）插入测试：随机生成 n 个非重复数据，依次插入

（2）查询测试：对（1）中的数据依次查询，每次查询必定成功

（3）删除测试：对（1）中的数据依次删除，每次删除必定成功

（4）查询测试：对（1）中的数据依次查询，每次查询必定失败

分别记录每个步骤运行的时间，以及每次测试发生的重哈希次数，得到完整数据见附录 A，源代码及复现方法见附录 B。

表 3 测试流程参数条件

	n	m	s
1	2000000	1000000	1
2	2000000	250000	4
3	2000000	62500	16
4	10000000	5000000	1
5	10000000	1250000	4
6	10000000	312500	16

4.2 结果分析

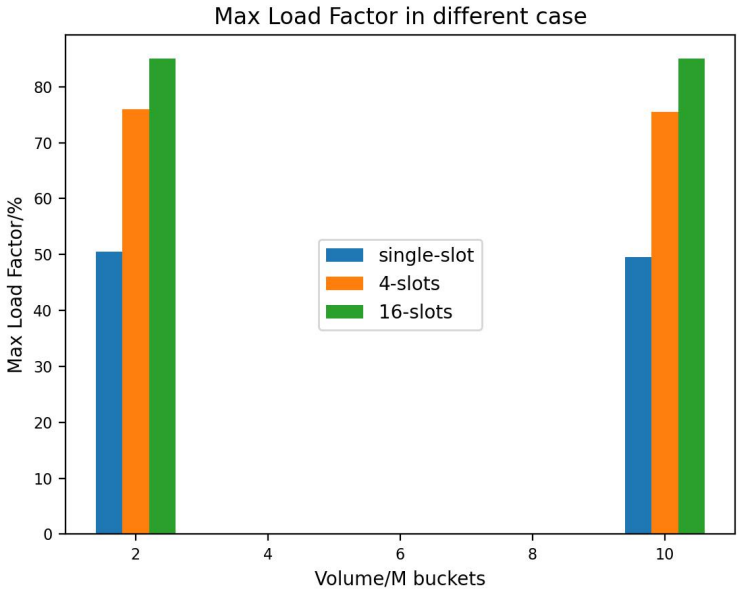

图 6 改进前后最大空间占用率对比

图 6 展示了 Dense Cuckoo Hashing 和 Cuckoo Hashing 相比在最大空间占用率上的优势，在 4-slots 条件下可以提升到 75%左右，16-slots 条件下可以提升到 85%左右。

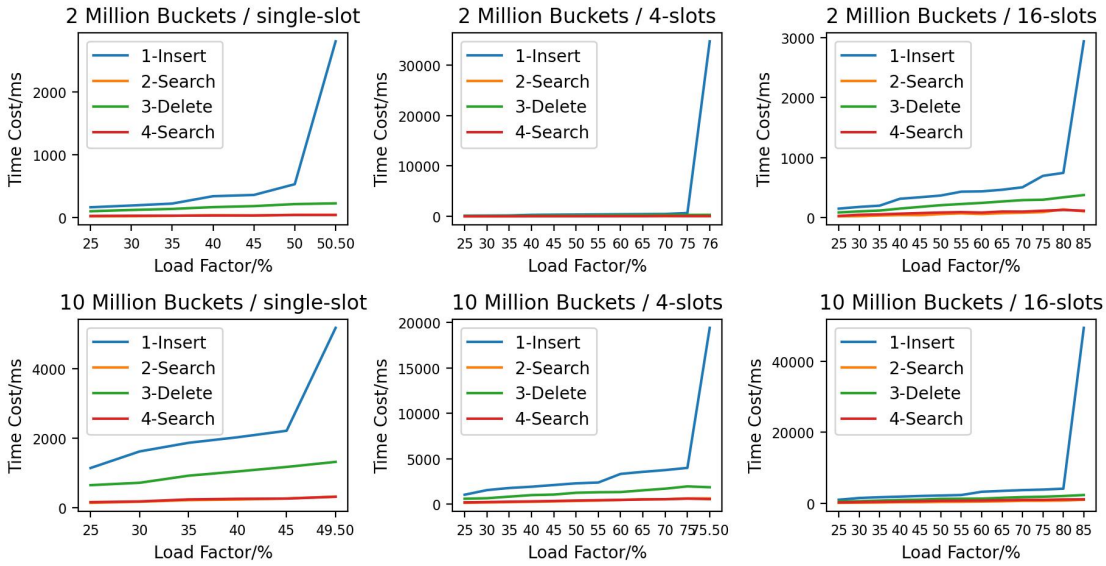

图 6 各种条件下四类操作性能比较

图 6 是在表 3 各种条件下，依次进行插入、查询、删除、查询四类操作在不同 Load Factor 下的性能比较，可以看到，随着 Load Factor 的增大，四类操作的 Time Cost 在大部分时间都在近似线性增长，在接近理论最大 Load Factor 时会迅速变大，伴随着大量的数据碰撞以及重哈希过程，并因为超过最大重哈希次数最终在实验条件下测试失败。

除此之外，比较不同操作的时间开销，插入操作的开销相对最大，查询操作的开销相对最小，删除操作的开销介于两者之间。

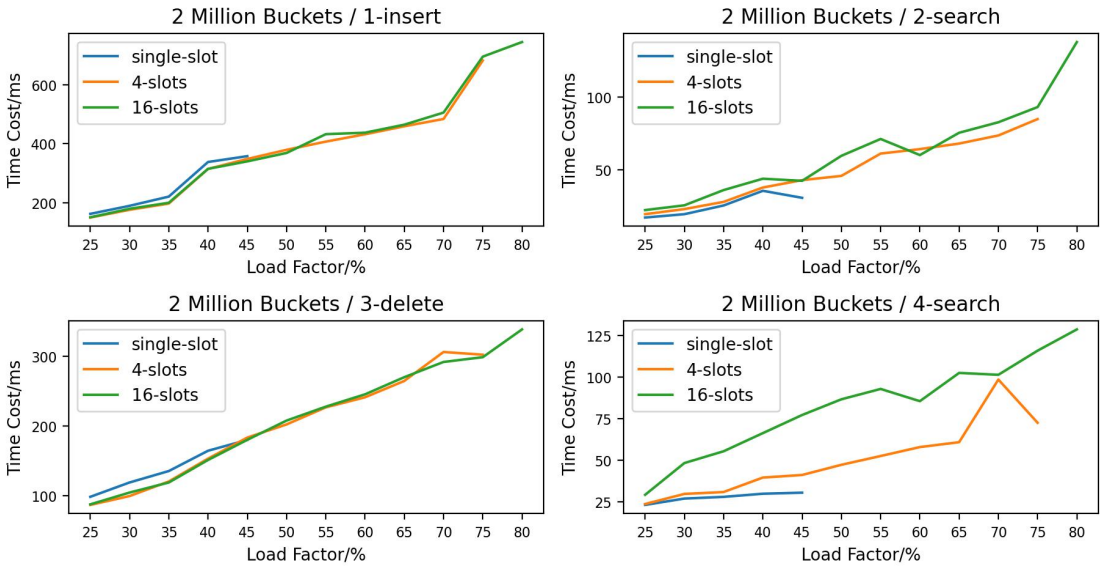

图 7 数据量为 200 万时改进前后各类操作性能对比

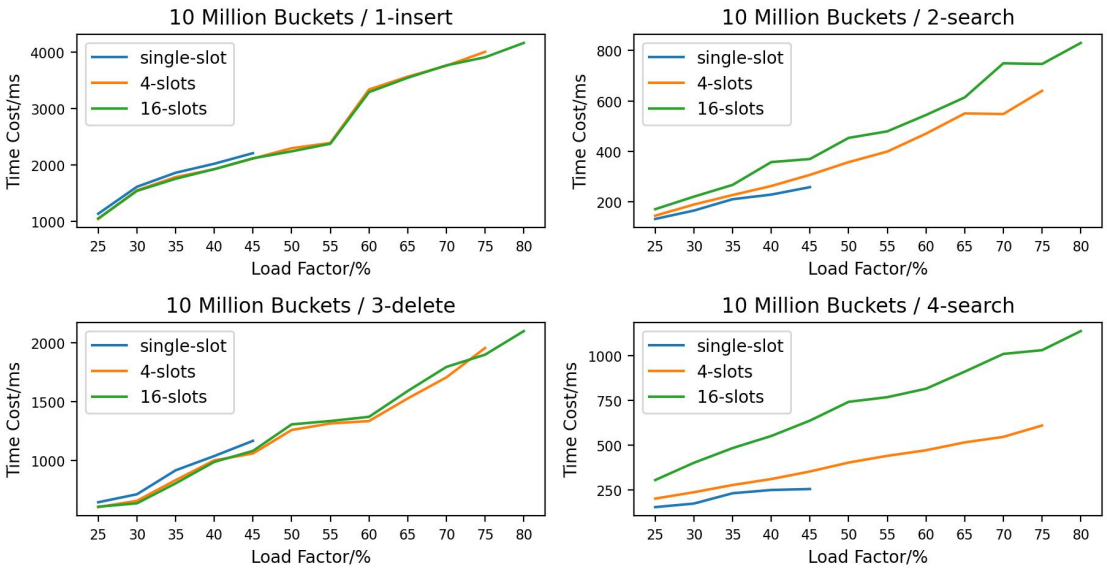

图 8 数据量为 1000 万时改进前后各类操作性能对比

图 7 和图 8 分别展示了在数据量为 200 万和 1000 万时传统 Cuckoo Hashing、4-slots Dense Cuckoo Hashing 和 16-slots Dense Cuckoo Hashing 中各类操作的性能对比（去除了 Time Cost 激增的尾部数据），可以看到：

插入过程、第一次查询过程和删除过程三类操作的时间开销差异比较小，但是 Dense Cuckoo Hashing 的最大空间利用率提升了将近一倍。

第二次查询过程由于是在全部删除元素后进行查询，每一次查询都是失败的，且需要遍历全部 slots 才判定失败，因此和传统 Cuckoo Hashing 相比较慢，且 slots 越多相对越慢。

5 结论

本文提出的 Dense Cuckoo Hashing 综合考虑了链式哈希和传统 Cuckoo Hashing 的优点，既维持了比较不错的操作性能，又显著提升了空间利用率，测试结果表明，Dense Cuckoo Hashing 在空间相对昂贵时是一种不错的优化结构。

本文也有一些不足，例如哈希函数的设计，以及没有考虑 Cuckoo Hashing 中存在重复元素的情况；除此之外，每个位置拥有多 slot 不利于多线程的并发操作，如何保证并发安全性比较困难。

参考文献

[1] R. Pagh and F. Rodler, “Cuckoo hashing,” Proc. ESA, pp. 121–133, 2001.

[2] Yu Hua, Hong Jiang, Dan Feng, "FAST: Near Real-time Searchable Data Analytics for the Cloud", Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), November 2014, Pages: 754-765.

[3] Yu Hua, Bin Xiao, Xue Liu, "NEST: Locality-aware Approximate Query Service for Cloud Computing", Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM), April 2013, pages: 1327-1335.

[4] Qiuyu Li, Yu Hua, Wenbo He, Dan Feng, Zhenhua Nie, Yuanyuan Sun, "Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage Services", Proceedings of IEEE/ACM International Symposium on Quality of Service (IWQoS), 2014.

[5] B. Fan, D. G. Andersen, and M. Kaminsky, “MemC3: Compact and concurrent memcache with dumber caching and smarter hashing,” Proc. USENIX NSDI, 2013.

[6] B. Debnath, S. Sengupta, and J. Li, “ChunkStash: speeding up inline storage deduplication using flash memory,” Proc. USENIX ATC, 2010.

[7] Slides of Cuckoo Hashing from Stanford University, https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/13/Small13.pdf

附录

A. 完整测试数据

数据规模 n	散列长度 m	槽数目 s	空间占用率	插入时间/ms	查询时间/ms	删除时间/ms	查询时间/ms	重哈希次数
500000	1000000	1	25%	161.95	16.97	98.17	23.28	0
600000	1000000	1	30%	189.20	19.29	118.78	27.09	0
700000	1000000	1	35%	220.58	25.28	135.20	28.10	0
800000	1000000	1	40%	338.13	35.38	164.36	29.98	0
900000	1000000	1	45%	357.94	30.54	179.94	30.62	0
1000000	1000000	1	50%	529.38	40.90	212.42	38.65	1
1010000	1000000	1	50.5%	2805.07	39.11	223.96	39.50	18
500000	250000	4	25%	149.42	19.28	86.56	23.83	0
600000	250000	4	30%	175.71	22.80	99.26	29.86	0
700000	250000	4	35%	196.90	27.75	120.24	31.01	0
800000	250000	4	40%	314.52	37.63	153.03	39.67	0
900000	250000	4	45%	348.48	42.72	183.28	41.22	0
1000000	250000	4	50%	379.43	45.63	202.16	47.32	0
1100000	250000	4	55%	407.30	60.99	226.66	52.59	0
1200000	250000	4	60%	432.24	64.06	241.13	57.94	0
1300000	250000	4	65%	459.41	67.88	264.62	60.88	0
1400000	250000	4	70%	484.25	73.45	306.21	98.42	0
1500000	250000	4	75%	682.81	84.72	302.30	72.50	0
1520000	250000	4	76%	34703.6	78.00	320.57	74.78	174
500000	62500	16	25%	150.24	22.12	87.35	29.34	0
600000	62500	16	30%	179.15	25.38	104.31	48.37	0
700000	62500	16	35%	199.64	35.88	118.72	55.44	0
800000	62500	16	40%	314.90	43.70	151.10	66.31	0
900000	62500	16	45%	340.21	42.24	180.02	77.17	0
1000000	62500	16	50%	368.75	59.45	207.69	86.58	0
1100000	62500	16	55%	432.61	71.04	227.70	92.79	0
1200000	62500	16	60%	437.70	59.97	245.39	85.46	0
1300000	62500	16	65%	464.83	75.31	270.02	102.39	0
1400000	62500	16	70%	505.94	82.54	291.89	101.27	0
1500000	62500	16	75%	695.98	92.99	298.74	115.74	0
1600000	62500	16	80%	745.58	137.74	338.66	128.48	0
1700000	62500	16	85%	2934.0	104.98	376.49	116.28	8
2500000	5000000	1	25%	1135.18	131.63	641.95	153.50	0
3000000	5000000	1	30%	1611.79	164.87	709.97	173.42	0
3500000	5000000	1	35%	1860.71	209.94	914.35	231.27	0
4000000	5000000	1	40%	2020.43	228.27	1036.21	249.47	0
4500000	5000000	1	45%	2207.54	257.82	1165.92	254.82	0
4950000	5000000	1	49.5%	5169.52	311.58	1311.66	307.58	0

2500000	1250000	4	25%	1043.07	144.60	600.39	201.19	0
3000000	1250000	4	30%	1548.98	188.82	654.26	236.88	0
3500000	1250000	4	35%	1783.49	226.58	830.73	277.31	0
4000000	1250000	4	40%	1922.72	262.35	999.47	310.68	0
4500000	1250000	4	45%	2111.79	306.56	1057.86	352.99	0
5000000	1250000	4	50%	2295.07	357.20	1258.48	403.06	0
5500000	1250000	4	55%	2387.91	399.05	1314.02	440.63	0
6000000	1250000	4	60%	3334.78	470.18	1334.48	471.80	0
6500000	1250000	4	65%	3560.95	550.30	1527.52	515.95	0
7000000	1250000	4	70%	3754.31	548.07	1707.33	546.87	0
7500000	1250000	4	75%	4000.65	640.18	1958.12	610.37	0
7550000	1250000	4	75.5%	19416.6	638.66	1864.83	562.82	13
2500000	312500	16	25%	1051.01	170.33	603.95	305.08	0
3000000	312500	16	30%	1539.13	220.07	632.67	401.81	0
3500000	312500	16	35%	1754.21	266.73	803.07	483.97	0
4000000	312500	16	40%	1924.68	357.20	986.05	551.77	0
4500000	312500	16	45%	2116.00	369.48	1080.04	637.86	0
5000000	312500	16	50%	2239.80	453.32	1306.24	742.98	0
5500000	312500	16	55%	2374.39	479.32	1334.94	769.06	0
6000000	312500	16	60%	3287.82	544.22	1370.61	816.39	0
6500000	312500	16	65%	3541.35	613.95	1590.9	911.98	0
7000000	312500	16	70%	3761.51	749.42	1796.28	1011.26	0
7500000	312500	16	75%	3904.61	746.59	1900.93	1032.17	0
8000000	312500	16	80%	4158.28	830.15	2101.79	1138.92	0
8500000	312500	16	85%	49383.2	1042.77	2374.52	1174.19	27

B. 源代码

https://github.com/yingjia-git/Cuckoo-Hashing