2020 级

《物联网数据存储与管理》课程

课 程 报 告

选题：Cuckoo-driven Way

姓　名　范亚锋

学　号　U201814580

班　号　物联网 2001 班

日　期　2023.5.16

目　　录

一、 Cuckoo-driven Way 简介

Cuckoo-driven Way 是一种基于 Cuckoo Hashing 算法的散列表技术。Cuckoo Hashing 算法是一种高效的散列表算法，它通过两个散列表（或者称为桶）来减少散列冲突。

Cuckoo Hashing 算法的基本思想是：使用两个散列表，每个元素在其中一个散列表中占据一个桶，如果在另一个散列表中发现该桶已经被占用，则将该元素插入到该桶所对应的位置，同时将原来占据该位置的元素插入到另一个散列表中。这样，即使散列函数存在冲突，也可以保证元素可以被正确地插入散列表中。

Cuckoo-driven Way 则是在 Cuckoo Hashing 算法的基础上进行了改进，双哈希函数，同时使用两个哈希表存储关键字，以减少哈希冲突的概率。在此基础上，Cuckoo-driven Way 还提供了一种循环移位的策略，以减少 cuckoo 操作中的无限循环的概率和有效存储。

总体来说，Cuckoo-driven Way 是一种高效的散列表技术，可以在大规模数据处理中发挥重要作用，尤其适用于需要高速查询的应用场景。

二、 数据结构设计

Cuckoo-driven Way 数据结构是一种高效的哈希表实现，它的设计灵感来自于鹊鸟的繁殖方式。这种数据结构使用了两个哈希函数和多个哈希表，每个哈希表都有自己的一个哈希函数，并且每个键值对存储在两个哈希表中的一个桶中。

Cuckoo-driven Way 采用了两个哈希表来存储关键字。哈希表的大小也是一个关键的设计参数。通常情况下，哈希表的大小是关键字数量的两倍。此外，每个哈希表都有一个默认的装填因子，当哈希表中的元素数量达到装填因子时，哈希表将进行扩容操作。

当插入一个新的键值对时，它首先根据第一个哈希函数计算出一个桶的位置，并将其插入到该桶中。如果该桶已经被占用，则该键值对会被替换到另一个哈希表中的相应桶中。如果另一个桶已经被占用，则该过程会继续，直到插入成功或者达到最大尝试次数。

当查找一个键值对时，它首先根据第一个哈希函数计算出一个桶的位置，并在该桶中查找。如果该桶中没有该键值对，则根据第二个哈希函数计算出另一个桶的位置，并在该桶中查找。如果仍然没有找到，则该键值对不存在于哈希表中。

Cuckoo-driven Way 数据结构的优点是它具有常数级别的插入和查找时间复杂度，并且能够高效地处理哈希冲突。缺点是它需要使用多个哈希表，并且在插

入时可能需要不断地迭代，这可能会导致性能下降。

三、 操作流程分析

Cuckoo-driven Way 的核心操作是插入和查找。

1. 插入操作

插入操作的基本思路是：首先将关键字插入到第一个哈希表中，如果发生了哈希冲突，就将冲突的元素移动到第二个哈希表中。如果在第二个哈希表中也发生了哈希冲突，就将冲突的元素移动回第一个哈希表中，以此类推，直到插入成功或者达到最大迭代次数。

具体来说，插入操作的流程如下：

1）使用两个哈希函数将关键字分别映射到第一个和第二个哈希表中。

2）如果第一个哈希表中的对应位置为空，就将关键字插入到该位置中，插入完成。

3）如果第一个哈希表中的对应位置已经被占用了，就将该元素移动到第二个哈希表中，并将第一个哈希表中的对应位置设置为空。

4）如果第二个哈希表中的对应位置为空，就将关键字插入到该位置中，插入完成。

5）如果第二个哈希表中的对应位置已经被占用了，就将该元素移动回第一个哈希表中，并将第二个哈希表中的对应位置设置为空。

6）重复上述步骤，直到插入成功或者达到最大迭代次数。

2. 查找操作

查找操作的基本思路是：首先使用两个哈希函数将关键字分别映射到第一个和第二个哈希表中，然后在两个哈希表中查找关键字。如果关键字在其中一个哈希表中存在，就返回 true；否则返回 false。

具体来说，查找操作的流程如下：

1）使用两个哈希函数将关键字分别映射到第一个和第二个哈希表中。

2）在第一个哈希表中查找关键字，如果存在，就返回 true。

3）在第二个哈希表中查找关键字，如果存在，就返回 true。

4）如果在两个哈希表中都没有找到关键字，就返回 false。

为了减少 Cuckoo-driven Way 中无限循环的概率，可以采取以下措施：

1.设定最大重试次数：在进行 Cuckoo 操作时，可以设置一个最大重试次数，如果超过该次数则认为当前操作失败。这可以防止出现死循环的情况。

2.增加哈希表容量：增加哈希表的容量可以减少哈希冲突的概率，从而降低 Cuckoo 操作失败的概率。

3.使用更多的哈希函数：使用更多的哈希函数可以增加元素与哈希表项之间的映射关系，从而减少哈希冲突的概率。

4.使用随机化：在 Cuckoo-driven Way 中，可以使用随机化来选择哈希函数，从而降低哈希冲突的概率，减少 Cuckoo 操作失败的概率。

为了有效存储，在 Cuckoo-driven Way 中，可以采取以下措施：

1.压缩哈希表：可以使用一些压缩技术来减少哈希表的存储空间，如哈希表压缩。

2.合并哈希表项：可以将一些相邻的哈希表项合并成一个大的哈希表项，从而减少哈希表的存储空间。

3.使用紧凑的哈希表：可以使用紧凑的哈希表来减少哈希表的存储空间，如线性探测哈希表。

总之，Cuckoo-driven Way 可以通过调整各种参数来优化其性能，包括减少无限循环的概率和有效存储。

四、 理论分析

Cuckoo-driven Way 具有较高的查找效率和较低的空间开销，但也存在一定的错误率。主要的错误率包括 false positive 和 false negative。

False positive 是指一个不存在的关键字被错误地判断为存在。它的概率可以通过哈希表的大小和元素数量共同决定。

False negative 是指一个存在的关键字被错误地判断为不存在。它的概率可以通过哈希函数的设计和哈希表的大小共同决定。

为了降低错误率，Cuckoo-driven Way 采用了双哈希函数的设计，同时还提供了一种循环移位的策略，以减少 cuckoo 操作中的无限循环的概率和有效存储。

五、 实验测试的性能

针对 Cuckoo-driven Way 的实验测试主要包括查询延迟、空间开销和错误率等性能指标的评估。

查询延迟：查询延迟是指在哈希表中查找一个关键字所需的时间。在实验测试中，我们可以通过在哈希表中插入一定数量的关键字，然后在哈希表中查找这些关键字的时间来评估查询延迟。

空间开销：空间开销是指哈希表所占用的内存空间。在实验测试中，我们可以通过在哈希表中插入一定数量的关键字，然后统计哈希表所占用的内存空间来评估空间开销。

错误率：错误率是指在哈希表中查找关键字时出现错误的概率。在实验测试中，我们可以通过在哈希表中插入一定数量的关键字，然后查找不在哈希表中的关键字的错误率来评估错误率。

综上所述，Cuckoo-driven Way 是一种用于解决哈希冲突的算法，它采用了双哈希函数和循环移位的策略，以提高哈希表的性能和减少错误率。在实验测试中，我们可以通过评估查询延迟、空间开销和错误率等性能指标来评估 Cuckoo-driven Way 的效果。

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.