2019 级

《大数据存储系统与管理》课程

课 程 报 告

选　题	**Cuckoo-driven Way**
姓　名	张家荣
学　号	**U201915084**
班　号	**ACM1901 班**
日　期	**2022.04.20**

目 录

一. 研究背景

在万物互联的大数据时代的当下，数据规模呈指数级扩张、内容结构日趋复杂，为云存储系统实现高效查询服务带来了巨大挑战。哈希表是一种根据键直接访问在内存存储位置的数据结构，它通过计算哈希函数，将所需查询的键映射到表中一个位置来访问记录，使得查询操作的时间复杂度能够达到 O(1)级别，因而在高效查询服务中得到了广泛应用。

哈希函数可能把不同的键映射到相同的位置，即存在哈希冲突。哈希冲突的传统解决方案有：开链法，即把所有被映射到同一个位置的元素以链表形式连接在一起，哈希表中每个位置存储链表的表头；开放寻址法，即当映射位置不为空时，按照特定规则逐一检查下一个位置是否为空，直到找到第一个空位置时插入元素，查询时按照同样的规则进行逐一比较。这些传统解决方案都只使用了一个哈希函数，且由于查询时需要比较的元素数量不定，查询操作的时间复杂度不一定能达到 O(1)。

与传统解决方案不同，Cuckoo Hashing 使用 d 个哈希函数，每个元素被映射到 d 个不同的候选位置，且每个元素只能存放在这 d 个候选位置之一，因此查询时只需要与至多 d 个位置进行比较，最坏情况下的查询时间复杂度为常数级。Cuckoo Hashing 一般设置 d 的值为 2，本文仅考虑 d=2 的情况，即给定两个哈希表 T_1、T_2 和两个哈希函数 h_1、h_2，元素 x 只能存放在 $T_1[h_1(x)]$或 $T_2[h_2(x)]$这两个候选位置之一。Cuckoo Hashing 插入元素的传统算法如下：

1．将待插入元素 x 通过对应哈希函数分别进行计算，得到映射到两个哈希表 T_1 和 T_2 的位置分别为 h1(x)和 h2(x)；

2．若两个映射位置都为空，则任意选择一个位置插入 x；

3．若两个映射位置中只有一个为空，则选择空位置插入 x；

4.当两个映射位置都不为空,则随机选择其中一个位置踢出存储在其中的元素 y，并将 x 插入该位置。接着对 y 使用另外一个哈希表对应的哈希函数进行计算出在另外一个表的映射位置，若新映射位置为空则将 y 插入该位置；若新映射位置仍不为空，则踢出存储在其中的元素 z，并将 y 插入该位置，依此循环执行直到找到空位置。如果踢出路径形成环路（如 x 踢出 y，y 踢出 z，z 踢出 x），即找不到空位置，则本轮插入

失败，执行 rehash 操作，即更换哈希函数 h_1、h_2，清空哈希表 T_1、T_2，重新将所有已插入元素和本轮欲插入元素 x 逐一插入哈希表中，若仍遇到插入失败，则反复执行 rehash 操作，直到插入全部成功为止。

由此可见，Cuckoo Hashing 插入元素的操作与查询操作相比需要更大的时间开销，是 Cuckoo Hashing 改进和优化的关键。判断踢出路径是否形成环路是 Cuckoo Hashing 插入元素算法的关键。一种直观的方法是对踢出路径上经过的位置打上标记，当遇到已标记的位置时，表明踢出路径形成了环路。由于存储标记信息需要额外的空间开销，现有的方法多是通过设置踢出阈值，当踢出次数超过踢出阈值后，才能判定踢出操作陷入环路，但踢出阈值设置的越大会导致判定环路的开销越大，设置的偏小又有可能导致较长的非环路被误判成环路引发不必要的 rehash 操作，Cuckoo 图中可能既有很长的非环路又有很短的环路，设置统一的踢出阈值显然会带来很多不必要的开销。

无论是打标记或是设置踢出阈值，都需要实际执行踢出操作，才能判断是否会陷入环路。一旦踢出操作陷入环路，便意味着本轮所做的踢出操作都是无用功，因为无论踢出多少次都找不到空位置。踢出操作包含大量的内存写操作，因此将会导致较大的时间延迟和系统资源消耗。

本文的设计目标是：对 Cuckoo Hashing 插入元素的传统算法进行改进，使之无需实际执行踢出操作就能提前判断踢出路径中是否会形成环路，并能提前判断哪一个候选位置具有更短的踢出路径，从而做出比“随机”更合理的选择。

二. 设计思路

2.1 Cuckoo 图

为了便于分析 Cuckoo Hashing 的踢出路径，引入 Cuckoo 图对哈希表中的元素及其候选位置进行描述。哈希表中的每个元素 x 对应于 Cuckoo 图中的一条有向边，边的起点是 x 的实际存储位置，边的终点是 x 的备用候选位置。

当某个位置为空时，其节点出度为 0；当某个位置存有元素时，由于一个位置至多存储一个元素，所以其节点出度为 1。当要踢出某个位置存储的元素时，该元素将被踢到该位置节点的出边的终点位置，因此 Cuckoo 图的有向边构成的路径即为踢出路径。

一个直观的想法是：在每次插入元素前，使用 BFS（广度优先搜索）计算待插入

元素的两个候选位置的踢出路径长度，并在 BFS 过程中对已访问的节点打标记，从而判断环路，根据 BFS 计算得到的信息选择踢出路径最短且不形成环路的候选位置执行踢出操作。但 BFS 过程需要遍历踢出路径上的每一个节点，意味着在每次实际执行踢出操作之前，需要模拟执行一遍踢出操作，需要较大的时间开销，并且这一时间开销会随着插入元素数目的增多、Cuckoo 图规模的扩大、负载率的升高而增长。

为降低这一时间开销，一种可能的解决思路是：把所有位置到空位置的最短路径长度存储在一张表中，每插入一个元素时对这张表进行动态更新，即期望通过利用已有信息来降低插入元素时最短路径长度的计算开销，例如使用动态的最短路径算法。但动态最短路径算法的计算开销依然不小，且这种思路意味着除了需要记录完整的 Cuckoo 图信息和为判断环路记录的标记信息外，还需要记录所有位置到空位置的最短路径长度，其引入的额外空间开销将变得很大。

由于在实际的大数据存储系统中，哈希表规模往往非常大，所以我们必须寻求一种计算开销和空间开销较低的提高插入效率的方法。计算最短路径虽然能保证选出最优的候选位置并把插入过程开销降到最低，但预判过程需要较高的计算和空间开销；而随机选择一个候选位置虽然在预判过程不需要任何计算和空间开销（因为没有预判过程），但插入过程开销较大。我们需要在二者之间寻找折中的方案，希望在预判过程中使用较少的计算和空间开销，消除插入过程中的大部分（而非全部）不必要开销。

Cuckoo 图的边的总数目一定小于等于节点的总数目，因为插入元素的总数目一定小于等于所有候选位置的总数目。因此，Cuckoo 图可能不是一个连通图，Cuckoo 图的极大连通子图个数至少为 1。同理可知，Cuckoo 图的每个极大连通子图也必然都满足边的总数目小于等于节点总数目，又因为连通性，所以 Cuckoo 图的每个极大连通子图一定要么满足“边数=节点数-1”，要么满足“边数=节点数”。我们称满足“边数=节点数-1”的子图是 non-maximal 的，意为该子图还有空位置能容纳新插入的元素；我们称满足“边数=节点数”的子图是 maximal 的，意为该子图没有空位置能容纳新插入的元素。因此，我们只能把待插入元素插入到 non-maximal 的子图中，如果待插入元素的某个候选位置位于 maximal 的子图中，那么我们不应试图把元素插入到该候选位置。

2.2 插入元素的情况分类

根据上述建模，我们可以根据待插入元素的候选位置所在子图的性质，把插入元

素的情况分为以下几类：

1．待插入元素的两个候选位置都不在现有的任何子图中。

这种情况下，待插入元素的两个候选位置都是空位置，因此可以任选一个位置插入元素。

待插入元素及其两个候选位置构成了 Cuckoo 图的一个新的子图，该子图恰有 2 个节点和 1 条边，因而是 non-maximal 的。

2．待插入元素的一个候选位置 a 在现有的某一子图中，另一个候选位置 b 不在现有的任何子图中。

这种情况下，待插入元素的候选位置 b 一定是空位置，因此选择候选位置 b 插入元素。

待插入元素及其候选位置 b 为候选位置 a 所在的子图恰添加了 1 个节点和 1 条边，因而如果插入元素前的子图是 non-maximal 的，插入元素后的子图仍是 non-maximal 的；如果插入元素前的子图是 maximal 的，插入元素后的子图仍是 maximal 的。即插入元素不改变候选位置 a 所在子图的性质。

3．待插入元素的两个候选位置都在现有的子图中。

（1）待插入元素的两个候选位置在同一个子图中，且该子图是 non-maximal 的。

这种情况下，待插入元素的两个候选位置至多有一个是空位置，也可能都不是空位置，但一定能插入成功。因此，如果某个候选位置是空位置，则选择空位置插入元素；如果两个候选位置都不是空位置，则任选一个位置插入元素，因为这种情况下不易判断哪个位置具有更短的踢出路径。

待插入元素为其候选位置所在的子图恰添加了 1 条边，因而插入元素后的子图将变为 maximal 的。

（2）待插入元素的两个候选位置在同一个子图中，且该子图是 maximal 的。

这种情况下，待插入元素候选位置所在的子图中已没有空位置能容纳新插入的元素，因此无论选择哪个候选位置都将插入失败。

（3）待插入元素的两个候选位置在不同的子图中，且两个子图都是 non-maximal 的。

这种情况下，待插入元素的两个候选位置可能都是空位置，可能只有一个是空位置，也可能都不是空位置，但一定能插入成功。因此，如果某个候选位置是空位置，则

选择空位置插入元素；如果两个候选位置都不是空位置，则任选一个位置插入元素，因为这种情况下不易判断哪个位置具有更短的踢出路径。

待插入元素将把两个子图连接起来变成一个新子图，为新子图恰添加了 1 条边。设两个子图的节点数分别为 v_1、v_2，则两个子图的边数分别为 v_1-1、v_2-1，新子图的节点数为 v_1+v_2，新子图的边数为 v_1-1+v_2-1+1=v_1+v_2-1，因此新子图是 non-maximal 的。

（4）待插入元素的两个候选位置在不同的子图中，且一个子图是 non-maximal 的，另一个子图是 maximal 的。

这种情况下，虽然选择任意一个候选位置都能踢出成功，但如果选择位于 maximal 子图的候选位置进行踢出，则踢出路径将在 maximal 子图中绕环一圈之后回到原位，并经过待插入元素的新边进入 non-maximal 子图，直至找到空位置；而如果选择位于 non-maximal 子图的候选位置进行踢出，则踢出路径将直接在 non-maximal 子图中找到空位置。因此，这种情况下应当选择位于 non-maximal 子图的候选位置进行踢出，因为这样能够获得更短的踢出路径。

待插入元素将把两个子图连接起来变成一个新子图，为新子图恰添加了 1 条边。设 non-maximal 的子图的节点数为 v_1，maximal 的子图的节点数为 v_2，则 non-maximal 的子图的边数为 v_1-1，maximal 的子图的边数为 v_2，新子图的节点数为 v_1+v_2，新子图的边数为 v_1-1+v_2+1=v_1+v_2，因此新子图是 maximal 的。

（5）待插入元素的两个候选位置在不同的子图中，且两个子图都是 maximal 的。

这种情况下，待插入元素候选位置所在的两个子图中都没有空位置能容纳新插入的元素，因此无论选择哪个候选位置都将插入失败。

2.3 理论分析

根据以上分析，在 Cuckoo Hashing 执行插入操作前，只需要知道待插入元素的候选位置所在子图的性质，就能在 O(1)时间内针对要将元素插入哪一个候选位置中做出合理选择，而且每插入一个元素后能够在 O(1)时间内对所有子图的性质进行动态更新。我们可以使用并查集数据结构来维护所有位置所属的子图编号，这样便可在近似 O(1)时间内查询某个位置所属的子图编号，可在近似 O(1)时间内合并两个位置所属的子图。我们用 unordered_map 存储当前所有子图的性质，这样便可在 O(1)时间内查询和更新某个子图编号对应子图的性质。这种改进算法引入的额外空间开销较小，且预判和动

态更新的时间复杂度可达常数级。

在情况 3（2）、3（5）下，传统算法只能通过执行踢出阈值次踢出操作才能够判定插入失败，而改进算法无需执行踢出操作就能在 O(1)时间内判定插入失败。在情况 3（4）下，传统算法可能选择具有较长踢出路径的位于 maximal 子图的候选位置进行踢出，而改进算法一定能在 O(1)时间内选择具有较短踢出路径的位于 non-maximal 子图的候选位置进行踢出。在以上三种情况下，改进算法与传统算法相比，通过在预判过程引入少量开销，节省了插入过程中的大量时间和系统资源开销。

对于其他情况，两个候选位置的踢出路径都不会形成环路，因此踢出的开销不会有较大差距，因此从中选出具有更短踢出路径的候选位置并不会显著降低踢出开销，但却需要引入大量计算和空间开销，并不划算。所以在其他情况下，改进算法和传统算法采用了相同的随机选择策略，因而性能相近。

综合以上理论分析可知，Cuckoo Hashing 插入元素的改进算法比传统算法具有更小的时间和系统资源开销。

三. 算法实现

下文将阐述改进的 Cuckoo Hashing 的具体实现，对应代码 ImprovedCuckoo.cpp。传统的 Cuckoo Hashing 的具体实现与之类似，本文不再赘述，详见代码 TraditionalCuckoo.cpp。

3.1 数据结构设计

Cuckoo Hashing 的哈希表个数为 TABLE_CNT，单个哈希表长度为 TABLE_LEN，哈希表中元素的类型为 string。两个哈希表构成一个 string 类型的一维数组 table，数组大小为 TABLE_CNT*TABLE_LEN，两个哈希表相邻顺序存放在一维数组中。两个哈希函数都使用 MurmurHash3，但使用不同的参数，其源代码来自 https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp。

为标记哈希表中各位置是否已有元素，每个位置的标记只需要消耗 1 比特空间，1 表示该位置有元素（不为空），0 表示该位置没有元素（为空）。所有位置的标记比特构成一个长度为 TABLE_CNT*TABLE_LEN 比特的 bitset 对象 occupied。

为使用并查集数据结构维护所有位置所属的子图编号，定义长度为

TABLE_CNT*TABLE_LEN 的 int 类型的一维数组 fa，fa[x]表示位置 x 的父节点位置，每个子图的所有节点中有且仅有一个根节点，其 fa 值就是它本身。fa 数组的初值为-1，表示该位置不属于现有的任何子图。通过 findRoot 函数可在近似 O(1)时间内根据 fa 数组找到任意位置所属子图的根节点编号，这个编号同时也是该子图的编号。

为存储当前所有子图编号对应的子图的性质，定义一个 unordered_map 对象 type，键和值的类型均为 int。子图的性质用枚举类型表示，SG_MAX（值为 10）表示 maximal，SG_NON_MAX（值为 11）表示 non-maximal。

综上，改进的 Cuckoo Hashing 的所有数据结构定义如下代码所示：

```
string table[TABLE_CNT * TABLE_LEN];
bitset<TABLE_CNT* TABLE_LEN> occupied;
int fa[TABLE_CNT * TABLE_LEN];
unordered_map<int, int> type;
enum subgraph_types {
    SG_MAX = 10,
    SG_NON_MAX
};
unordered_set<string> inserted_elems;
vector<string> stash;
```

如果要执行 rehash 操作，还需定义 unordered_set 对象 inserted_elems，集合中元素的类型为 string，用于存放当前已插入的所有元素（因为直接从哈希表中不易获知此信息），便于 rehash 操作在清空哈希表后重新插入这些元素。本实验不执行 rehash 操作，原因将在后续实验章节中说明。

如果要引入 stash 结构，还需定义固定长度的 string 类型一维数组 stash。stash 中存放插入失败时找不到空位置而被踢出的元素，借助 stash 可以使得插入失败时不必进行代价高昂的 rehash 操作，减少了 rehash 操作的次数。文献[6]中证明了，使用长度为 s 的 stash 时，在插入 n 个元素后进行 rehash 操作的概率为 $O(n^{-s})$。但引入 stash 意味着每个元素可能存放的位置从 2 个候选位置变成 2+s 个位置，增大了查询操作的开销，因此 stash 长度一般很小（例如 s=5）。然而在后续的实验过程中我发现，一旦哈希表的负载率超过一定值之后，就会频繁地遭遇插入失败，而小容量的 stash 只适合解决偶尔几

次的插入失败，当负载率较高时，使用 stash 并不能显著减少 rehash 操作的次数，反而为每一次查询都引入了更多开销，因此本实验没有使用 stash。

3.2 函数接口设计

1. int findRoot(int x);

该函数是并查集操作的一部分，返回位置 x 所属子图的编号，具体实现中为提高并查集查找效率，还包含了并查集的路径压缩过程。

2. void init();

该函数将所有数据结构（inserted_elems 除外，因为 rehash 操作要用）恢复初始值状态，用于系统初始化和 rehash 操作中的清空步骤。

3. int determine_v_add(int idx1, int idx2);

该函数将判断待插入元素的两个候选位置 idx1 和 idx2 对应于 2.2 中所述的哪一大类。函数返回 2 表明属于第 1 大类，返回 1 表明属于第 2 大类，返回 0 表明属于第 3 大类。

4. void Kick(const string& s, int idx);

该函数将尝试把元素 s 存入位置 idx，如果位置 idx 不为空则循环执行踢出操作，直到找到空位置为止。在改进的 Cuckoo Hashing 插入算法中，Kick 函数只会在预判存在空位置的时候执行，因此 Kick 函数一定能找到空位置并执行成功。

5. int InsertElem(const string& s);

该函数将把元素 s 插入哈希表中，如果插入成功还将动态更新 fa、type 等子图信息。函数返回 1 表示插入情况属于第 1 大类；返回 2 表示插入情况属于第 2 大类，返回 3 表示插入情况属于第 3 大类第（1）小类；返回 4 表示插入情况属于第 3 大类第（3）小类；返回 5 表示插入情况属于第 3 大类第（4）小类；返回 6 表示插入失败，但 stash 仍有空间，元素将被放入 stash 中；返回 7 表示插入失败，且 stash 已满，将进行 rehash 操作。

6. int Insert(const string& s);

该函数将把元素 s 插入哈希表中，如果插入成功则还将动态更新 fa、type 等子图信息，如果插入失败且 stash 已满则还将进行 rehash 操作。该函数内部调用了 InsertElem 函数，之所以把插入操作拆分成两个函数，是为了把 rehash 操作分离出来写在 Insert

中。函数返回-1 表示已插入元素个数超过哈希表位置总数，哈希表容量已满；返回 0 表示待插入元素已存在于哈希表或 stash 中，不执行插入操作；返回 1~7 的含义同 InsertElem 函数。

注意后文实验中没有执行 rehash 操作，该函数 rehash 操作的相关代码被注释掉。

7. int Search(const string& s);

该函数将查询元素 s 是否存在于哈希表或 stash 中。函数返回 0 表示查询结果为不存在，返回 1 表示查询结果为存在。

四. 实验测试

4.1 实验环境

为了验证前文理论分析的正确性，我们将对前文实现的传统的 Cuckoo Hashing 和改进的 Cuckoo Hashing 进行性能观测。实验测试的软硬件环境如表 1 所示。

表 1 实验测试软硬件环境

CPU	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz，6 核
内存	16 GB (8GB DDR4 2666MHz 内存×2，双通道)
硬盘	512GB 固态硬盘
操作系统	Microsoft Windows 10 家庭中文版 (Build: 18363.1556) (64 位)
开发语言	C++
集成开发环境	Microsoft Visual Studio 2019 (版本 16.9.1)
编译器	MSVC 19.28.29912 for x64

4.2 实验内容

为了控制变量，我们让改进的插入算法和传统的插入算法的测试程序使用种子相同的随机数生成器，使用参数相同的哈希函数，执行相同的插入和查询序列操作，假设传统插入算法的踢出阈值保证不会把较长的踢出路径误判成环路，则由两算法的异同之处分析可知，二者执行相同的操作必将得到相同的结果，即都成功或都失败，唯一的区别是改进算法的时间和系统资源开销更低。

改进的插入算法和传统的插入算法相比，只是能够更快地判定插入失败，而并不能降低插入失败后 rehash 操作的开销，也并不能降低插入失败的概率。由于 rehash 操作可能需要反复多次，当哈希表的负载率较高时更是如此，会带来极大的时间开销，而根据理论分析，改进算法和传统算法的 rehash 开销相同，所以实验比较时 rehash 开销不是我们关心的对象，反而可能掩盖改进算法的性能优势，不利于实验结果的分析。因此在实验测试时对算法进行如下调整：当判定某元素插入失败时，放弃插入该元素，不执行 rehash 操作，立即继续插入下一个元素。

测试程序的流程为：首先把测试文件的每一行作为一个元素读入到内存中，然后按读入顺序逐一向哈希表中插入这些元素，最后按读入顺序逐一在哈希表中查询这些元素。程序将统计插入所有元素的总耗时、查询所有元素的总耗时、成功的插入操作数目、成功的查询操作数目、测试文件元素总数目，并分别输出。注意一个测试文件的所有行都是互异的。

为了涵盖不同规模、不同应用场景的数据，我们为测试程序准备了三个不同特征的测试文件。number20000.txt 包含 20000 个均匀分布的随机整数，每个整数的范围是[-2147483648,2147483647]，该文件是使用 Mersenne Twister 19937 随机数生成器生成的，生成程序代码见 number2000.cpp。vocab100000.txt 包含 100000 个由英文字母、数字、符号组成的有意义单词，是从 PubMed 摘要文本中提取得到的，源数据来自 http://archive.ics.uci.edu/ml/machine-learning-databases/bag-of-words/vocab.pubmed.txt 。macos500000.txt 包含 500000 个文件的哈希值，每个哈希值是由英文字母、数字组成的长度为 12 的串，是根据 Mac OS X 服务器电脑上的文件的文件内容计算得到的，源数据来自 https://tracer.filesystems.org/traces/macos/2011/macos-2011-11-24-010804.tar.bz2。

为提升传统插入算法的性能表现，我们根据负载率情况对踢出阈值进行动态调整，使得在保证不会把较长的非环路误判成环路的前提下，踢出阈值尽量小。由于改进算法判定环路的方式不同，不可能发生误判，所以改进算法的插入失败次数就是传统算法不发生误判时的插入失败次数，据此指导踢出阈值的调整。

4.3 实验结果分析

在三个测试文件上分别运行测试程序，根据不同的哈希表负载率，调整程序中的单个哈希表长度（TABLE_LEN）、踢出阈值（MAX_KICK）等参数，得到测试结果如

表 2、表 3、表 4 所示。

表 2 number20000.txt 测试结果

哈希表负载率	单个哈希表长度	踢出阈值	传统算法		改进算法		插入/查询成功数
			插入耗时(ms)	查询耗时(ms)	插入耗时(ms)	查询耗时(ms)	
5%	200000	600	201	16	158	11	20000
25%	40000	600	207	13	143	11	20000
35%	28571	600	213	14	140	11	20000
40%	25000	600	202	13	135	12	20000
45%	22222	600	210	13	139	12	20000
50%	20000	600	224	14	161	13	20000
55%	18182	600	306	13	143	12	19972
60%	16667	600	406	17	162	14	19872
65%	15385	600	477	12	151	11	19707
70%	14286	601	777	15	146	11	19373
75%	13333	600	1040	13	144	13	19038
80%	12500	600	1285	13	139	14	18646
85%	11765	600	1575	12	173	16	18115
90%	11111	600	1797	13	159	14	17758
95%	10526	600	2353	14	157	17	17204
100%	10000	600	2694	12	168	13	16757

表 3 vocab100000.txt 测试结果

哈希表负载率	单个哈希表长度	踢出阈值	传统算法		改进算法		插入/查询成功数
			插入耗时(ms)	查询耗时(ms)	插入耗时(ms)	查询耗时(ms)	
5%	1000000	10	994	78	786	60	100000
25%	200000	10	986	77	722	62	100000
40%	125000	30	1002	71	759	83	100000
50%	100000	500	1127	70	765	71	100000
55%	90909	900	1265	70	721	64	99913
60%	83333	900	2159	77	812	75	99338
65%	76923	1000	4180	67	812	76	98337
70%	71429	1100	6552	70	917	73	96887
75%	66667	1200	9682	68	736	70	95174
80%	62500	1400	14394	67	742	65	92976
85%	58824	1400	19889	75	744	63	90801
90%	55556	1400	21352	70	804	77	88534
95%	52632	1400	25179	68	696	66	86143
100%	50000	1400	28582	66	758	71	83738

表 4 macos500000.txt 测试结果

哈希表负载率	单个哈希表长度	踢出阈值	传统算法		改进算法		插入/查询成功数
			插入耗时(ms)	查询耗时(ms)	插入耗时(ms)	查询耗时(ms)	
5%	5000000	10	4796	428	3668	320	500000
20%	1250000	10	4782	385	3619	331	500000
35%	714286	30	4845	381	3463	350	500000
40%	625000	30	4775	369	3499	345	500000
45%	555556	60	4994	371	3471	347	500000
50%	500000	1000	4990	391	3551	355	500000
55%	454545	2000	7266	372	3606	362	499515
60%	416667	3000	18750	376	3519	352	496993
65%	384615	3000	40945	392	3492	356	491874
70%	357143	4000	80615	389	3555	362	484419
75%	333333	4000	119213	376	3569	363	475440
80%	312500	4000	179314	417	3517	370	465625
85%	294118	-	-	-	3474	368	454290
90%	277778	-	-	-	3428	363	442813
95%	263158	-	-	-	3318	369	431072
100%	250000	-	-	-	3332	387	419263

注：表 4 中空白单元格是因为传统算法在 macos500000.txt 上哈希表负载率≥85%时的插入耗时过长（＞200000ms）而没有进行测量。

由表 2、表 3、表 4，分析两算法在三个测试文件上的插入耗时表现，绘制出图 1。由图 1 可知，当哈希表负载率≤50%时，哈希表不会遇到插入失败的情况，此时传统算法和改进算法的插入耗时相近，但无论哈希表负载率有多么低，改进算法的插入耗时始终低于传统算法的插入耗时，这是因为改进算法在情况 3（4）下比传统算法表现更优。当哈希表负载率＞50%时，随着哈希表负载率的增加，哈希表遭遇插入失败的次数将变得越来越多，对传统算法而言，每遭遇一次插入失败都需要花费踢出阈值次踢出操作的时间开销，因而传统算法的插入总耗时随着哈希表负载率的增加而急剧上升；对改进算法而言，判定插入失败需要花费的时间是常数级，因此改进算法的插入总耗时与哈希表负载率几乎无关，始终维持在较低水平，表现出了很好的性能稳定性。

由图 1 还可知，随着测试文件规模的扩大，保证不发生误判的最低踢出阈值也越来越大，导致传统算法判定插入失败的开销也急剧增大，图 1 中 macos500000.txt 的传统算法曲线斜率明显大于 number20000.txt 中传统算法的曲线斜率，表明随测试文件规模扩大，改进算法开销的增加比传统算法更加缓慢。

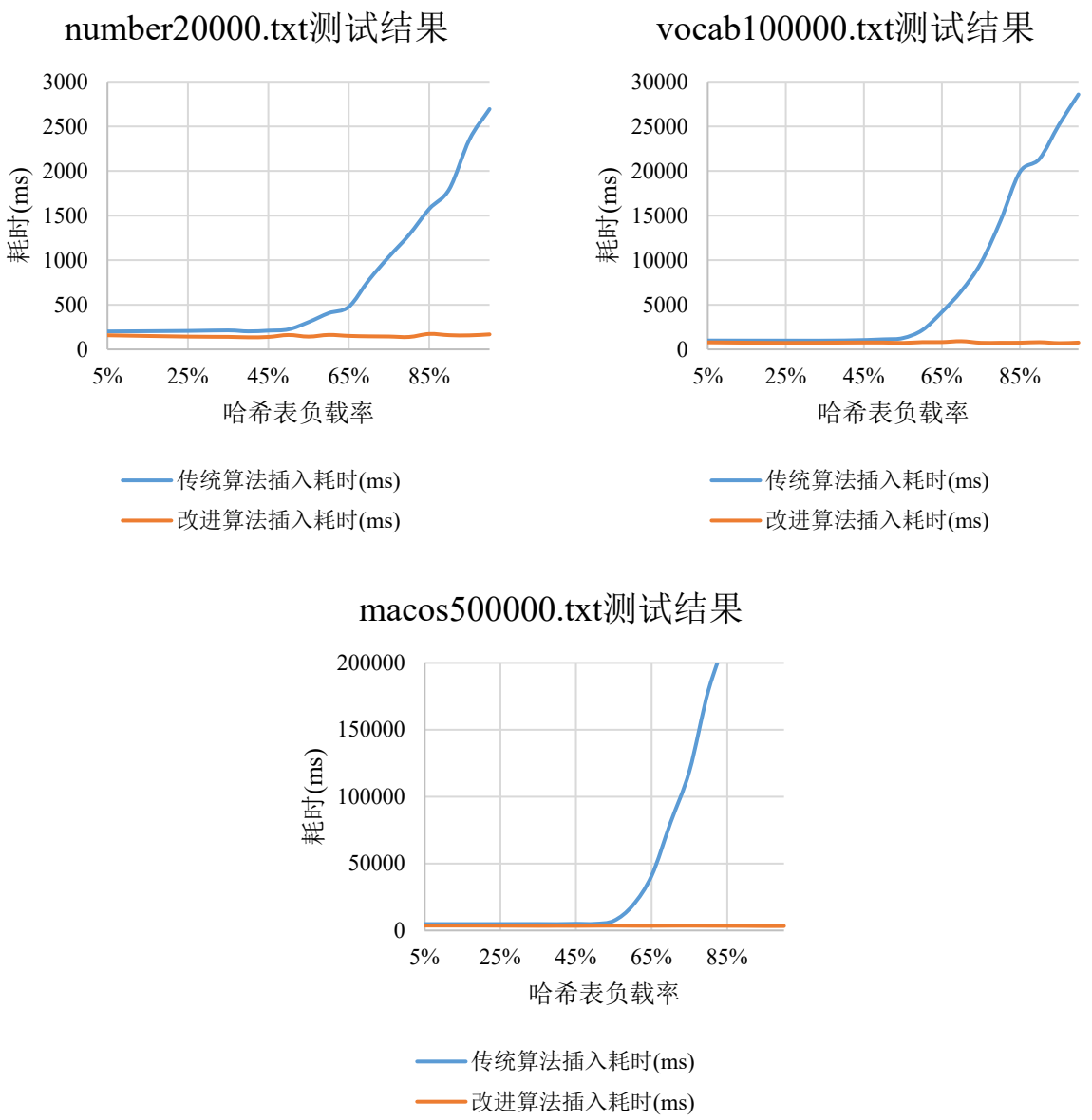
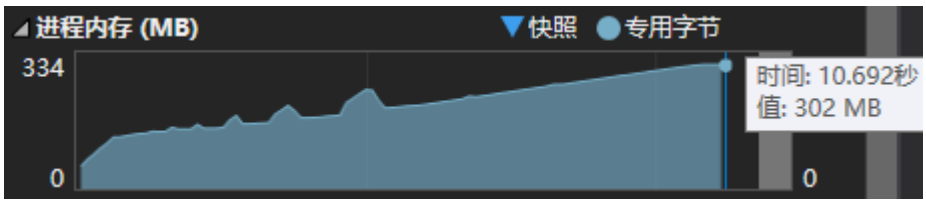


图 1 传统算法和改进算法在三个测试文件上的插入耗时表现

由表 2、表 3、表 4，分析两算法在三个测试文件上的插入耗时表现，绘制出图 2。由图 2 可知，在误差允许范围内，可以认为传统算法和改进算法具有相同的查询效率，这是因为传统算法和改进算法的查询逻辑完全相同。此外，查询耗时随着测试文件规模的扩大而变长，因为需要查询的元素个数随之增加。

图 2 传统算法和改进算法在三个测试文件上的查询耗时表现

空间开销方面，使用 Visual Studio 2019 的调试界面的诊断工具观察内存占用情况，以使用 macos500000.txt 测试文件、50%哈希表负载率为例，如图 2 所示。观察到传统算法内存占用最大值为 302MB，而改进算法内存占用最大值为 326MB，只比传统算法高 8%，因此改进算法引入的额外空间开销较小。



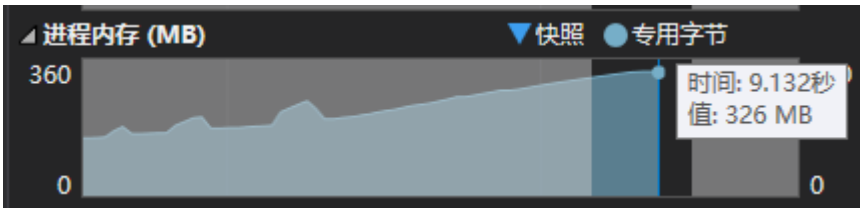


图 3 传统算法（上）和改进算法（下）的内存占用情况

五．结论

理论分析和实验结果都表明，本文提出的 Cuckoo Hashing 的改进插入算法能够通过在预判过程中使用较少的计算和空间开销，消除插入过程中的大部分不必要开销，极大减少了插入操作的总体时间和系统资源开销，并使得插入操作耗时与哈希表负载率无关，增强了 Cuckoo Hashing 性能的稳定性。

本文提出的改进算法仍有一些局限性，例如只适用于只使用 2 个哈希函数的 Cuckoo Hashing，只适用于在一个位置只存储一个元素的 Cuckoo Hashing。在今后的研究中，将考虑把本文的改进思路应用于更一般的 Cuckoo Hashing 结构中。

参考文献

[1] R. Pagh, F. Rodler, "Cuckoo hashing", Proc. ESA, 2001, pages: 121–133.

[2] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, Pengfei Zuo, "SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems", Proceedings of USENIX Annual Technical Conference (USENIX ATC), 2017, pages: 553-566.

[3] 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, pages: 50-55.

[4] 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), 2014, pages: 754-765.

［5］ 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), 2013, pages: 1327-1335.

［6］ A. Kirsch, M. Mitzenmacher, U. Wieder, "More robust hashing: Cuckoo hashing with a stash", SIAM Journal on Computing, 2010, pages: 1543-1561.

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

［8］ K. Schwarz. "Cuckoo Hashing", http://web.stanford.edu/class/archive/cs/cs166/cs166.1216/lectures/07/Slides07.pdf, 2021.

附录

实验相关的源代码、测试文件已上传至 https://gitee.com/zhang_jiarong/improved-cuckoo-hashing 仓库中。