Hash 实验报告

2021201645 赵宸

本次实验主要探究了不同种的 hash 函数构造方案（多分组和少分组）以及不同的冲突处理策略下的程序数据处理速度.六组的结果实验结果相对来说比较明显,但与我此前的猜想略有差异. ## 六组测试数据构造方法 实验指导中提到了三个方面: - 不同的数据规模 - 不同的插入/查询操作比例 - 插入和查询的不同分布方式

其中我认为第一点和第二点是有意义的,不同的冲突处理和 hash 函数最大的优缺点可能就体现在 **hash 表的填充因子**和**插入查找的速度差异**上,因此我对这两个因素做了控制变量法;而第三个因素我则认为并不能体现出不同处理方式中的差异,因此根据最不利原则,我采用了理论处理速度最慢的 **“先 0 后 1”** 的处理方法.
具体的分布是这样的:
data1 和 data2 为小样本数据,填充因子控制在 1%左右,远远小于 M,几乎不会出现冲突.以此来测量在**不会出现冲突**时,冲突处理的 debuff 效果.
data3 和 data5 为大插入样本数据,输入的数据中 `0` 的含量非常大,基本上可以达到 50%的填充因子,同时跨越两个不同的数据集,也可以测试中文输入是否会对速度产生影响.对**数据填充因子敏感**的方法影响巨大.
data4 是大输出样本数据,输入的数据中 `1` 的含量非常大,测试在 30%的填充因子下各个数据结构和冲突处理方案的查询速度,部分对**查找敏感**的方法数据波动较大.
data6 则是中等大小,中英混合,插入和查找平衡的数据集,用于做**对照组**.

数据特征总结: - data1,data2: 小样本(5MB 以内),来自 poj.txt,插入查找 4:1; - data3: 大样本(10MB),来自 poj.txt,插入查找 7:1; - data4:大样本(10MB),来自 hdu.txt,插入查找 1:3; - data5:大样本(12MB),来自 hdu.txt,插入查找 1:1; - data6:中等样本(7MB),来自 hdu.txt,插入查找 1:1; ## hash 函数与冲突处理 关于 hash 函数问题,我主要采用的是字符串哈希方法,即以**多项式加权**处理:

$$h(x) = \sum_{i=1}^{len}(x_i p^i) + key_{start}$$

其中,len 为字符串长度,p 取 `31`,M 取 `1333331`,x 为每个字符的 ascii/utf-8 对应的 int 数字,key_{start}取 `39`.特别的 utf-8 看做一个字符,即看做一个二十四位的无符号整数,并对这个数字加权.
而对于冲突处理,我采用了**链地址法**,**双向平方探测法**,和**线性试探法**.
对于链地址法,我利用链表的动态性,动态的增加对应同一个 key 值的元素,同时用链表相连接.特别的,我没有采用按序排列的方法来规范链表,考虑到时间复杂度没有差异,因此数据也应当是准确的;而双向平方探测法,顾名思义,以 hashCode 为中心向前向后按$base^2(base = 0,1,2\ldots)$探测,同时注意+M 和%M 以保证数据永远落在 M 的

范围内;线性探测法与双向平方探测基本一致,唯一的不同是线性探测仅仅向前探测,而且仅仅按$base$探测.与双向平方探测相比,线性探测要更为稳定,更容易做剪枝处理.

测试结果

hash函数与冲突处理策略	data1(s)	data2(s)	data3(s)	data4(s)	data5(s)	data6(s)	whole average time(s/MB)	1&2 average time	3&5 average time
ascii+链地址法	0.678	1.396	4.588	4.59	5.857	2.815	0.448	0.406	0.472
ascii+双向平方探测法	0.646	2.492	19.276	26.821	27.962	15.596	2.084	0.621	2.137
ascii+线性探测	0.325	1.102	2.052	2.598	2.847	1.433	0.233	0.280	0.221
UTF-8+链地址法	1.029	3.747	6.229	7.29	9.467	4.71	0.73	0.936	0.71
UTF-8+双向平方探测法	3.488	10.667	21.138	27.86	29.196	15.904	2.432	2.775	2.278
UTF-8+线性探测	0.542	1.867	3.47	4.316	4.766	2.647	0.396	0.472	0.373
average time	1.123	3.545	9.459	12.246	13.349	7.184	1.054	0.915	1.032

关于 hash 的一系列问题

Question1

Q: 将 utf-8 字符串“当作”ascii 字符串进行处理，使用针对 ascii 字符串的哈希函数，实际效果如何（相比 “针对 utf-8 字符串设计的哈希函数）？可能的原因是什么？
A: 将 utf-8 当做 ascii 处理,效果好于 utf-8 直接处理.其中在几乎没有冲突时,utf-8 要远差于 ascii;装填因子稍大时,utf-8 差于 ascii,但差距较小. **原因**:将 utf-8 实质上是把各个字符不规则加权,其中多字节字符前面的字节加权值为2^8或2^{16}.由于采用不规则加权,而且加权值是一个非常”规整"的合数,这就大大增加了冲突的概率,因此对冲突特别敏感的链地址法会发生明显变慢;另一方面,从信息熵的角度,高字节的位置在加权的时候很可能就直接被截掉了,对 hashcode 几乎没有贡献,这也极大地增大了冲突的概率.

Question2

Q: 在你的测试中，不同的冲突处理策略性能如何？可能的原因是什么？
A: 总体来看,线性探测最好,链地址次之,双向平方最差. 就插入速度而言,线性最好,链地址次之,双向平方最差;而查找操作中,链地址的效果要比线性更好. ($T_{data4} - t_{25average} * size_{4insert}$).**原因**:**插入过程**链表和线性探测效果基本一致,但是链表要多一个创建和 delete 的过程,因此链表要略慢;而**查找过程**链表要更稳定,不同的 hashcode 之间不会产生干扰,而线性中随填充因子增大,hashcode 之前会产生严重的干扰而导致查找变慢;至于**双向平方查找**,由于其跳跃的不稳定性,随着填充因子的增大,其寻找的跳跃次数会指数级的增大,不稳定且效率低,因此运行速度极慢.

Question3

Q:设计哈希函数时，我们往往假定字符串每个位置上出现字符集内每个字符的概率都是相等的，但实际的数据集往往并不满足这一点。这可能造成什么影响？
A:这可能会大大增加冲突出现的概率,进而那些对冲突敏感和对装填因子敏感的冲突函数的效率和实际效果会大大降低,进而使结果有偏.

Qusetion4

Q:对于“字符串到数字映射”问题(给定一组字符串以及它们各自对应的数字，然后多次查询某个字符串对应的数字，可能会在中途更新某个字符串对应的数字)，哈希表并不总是最优方案。请描述一种输入数据，再举出一种哈希表之外的数据结构，对这种数据，这种数据结构能比哈希表更高效地解决“字符串到数字映射”问题。
A:可以考虑 BitMap 数据结构.
数据构造:假设数据量非常大,用户名之间的相似度很高,但不重复;查询量很大,插入量相对较小.在这样的数据集上,BitMap 的效果要好于 hash 函数.