

发帖

回复

返回

查看: 16 | 回复: 0

?????v4467



组别 论坛元老
生日
帖子 471
积分 4956
性别 保密
注册时间 2017-10-11

超酷算法：基数估计 - 数据库 - ITeye资讯 [\[复制链接\]](#)

发表于 2017-10-16 10:46 | 只看楼主

字体大小: 1

假设你有一个很大的数据集，非常非常大，以至于不能全部存入内存。这个数据集中有重复的数据，你想找出有多少重复的数据，但数据并没有排序，由于数据量太大所以排序是不切实际的。你如何来估计数据集含有多少无重复的数据呢？这在许多应用中是很有用的，比如数据库中的计划查询：最好的查询计划不仅仅取决于总共有多少数据，它也取决于它含有多少无重复的数据。

在你继续读下去之前，我会引导你思考很多，因为今天我们要讨论的算法虽然很简单，但极具创意，它不是这么容易就能想出来的。

一个简单的朴素基数估计器

让我们从一个简单的例子开始吧。假定某人以下列方式来生成数据：

[1]生成 n 个充分分散的随机数

[2]任意地从中选择一些数字，使其重复某次

[3]打乱这些数字

我们怎么估计结果数据集中有多少非重复的数字呢？了解到原来的数据集是随机数，且充分分散，一个非常简单的方法是：找出最小的数字。如果最大的可能的数值是 m ，最小的值是 x ，我们可以估计大概有 m/x 个非重复的数字在数据集里面。举个例子，如果我们扫描一个数字在 0 到 1 之间的数据集，发现最小的数字是 0.01。我们有理由猜想可能数据集里大概有 100 个非重复的数字。如果我们找到一个更小的最小值的话，可能包含的数据个数可能就更多了。请注意不管每个数字重复了多少次都没关系，这是很自然的，因为重复多少次并不会影响 \min 的输出值。

这个过程的优点是直观，但同时它也很不精确。不难举出一个反例：一个只包含少数几个非重复数字的数据集里面有一个很小的数。同样的一个含有许多非重复数字的数据集含有一个比我们想像中更大的最小值，用这种估计方法也会很不精确。最后，很少有数据充分分散充分随机的数据集。但是这个算法原型给了我们一些灵感使得我们有可能达到我们的目的，我们需要更精致一些的算法。

基于概率的计数

第一处改进来自 Flajolet 和 Martin 的论文

[[ur="http://www.cse.unsw.edu.au/~cs9314/07s1/lectures/Lin_CS9314_References/fm85.pdf"](http://www.cse.unsw.edu.au/~cs9314/07s1/lectures/Lin_CS9314_References/fm85.pdf)] Probabilistic Counting Algorithms for Data Base Applications [url]。进一步的改进来自 Durand-Flajolet 的论文 LogLog counting of large cardinalities 和 Flajolet et al 的论文 HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm。从一篇论文到另一篇论文来观察想法的产生和改进很有趣，但我的方法稍有不同，我会演示如何从头开始构建并改善一个解决方法，省略了一些原始论文中的算法。有兴趣的读者可以读一下那三篇论文，论文里面包含了大量的数学知识，我这里不会详细探讨。

首先，Flajolet 和 Martin 发现对于任意数据集，我们总可以给出一个好的哈希函数，使得哈希后的数据集可以是我们需要任意一种排列。甚至充分分散的(伪)随机数也是如此。通过这个简单的灵感，我们可以把我们之前产生的数据集转化为我们想要的数据集，但是这远远不够。

接下来，他们发现存在更好的估计非重复数个数的方法。部分方法比记录最小的哈希值表现得更好。Flajolet 和 Martin 用的估计方法是计算哈希后的值的首部的 0 字的个数。显然在一个随机的数据集中，平均每 2^k 个元素就出现一个长度为 k 的全为 0 的比特序列。我们要做的就是找出这些序列并记录最长的来估计非重复元素的个数。然而这仍然不是一个很棒的估计器。它最多只能给我们一个 2 的幂的数量的估计。而且不像基于最小值的估计方法，这个方法的方差很大。但在另一个方面，我们的估计需要的空间非常小：为了记录最长 32 比特的先导 0 比特序列，我们只需要一个 5 比特的数字就可以了。

引用附注：Flajolet-Martin 原先的论文在这里继续讨论了一种基于 bitmap 的过程来获得一个更精确的估计。我不会讨论

个细节因为它马上就会在随后的方法中得到改进。更多细节对于有兴趣的读者可以阅读原文。

现在我们得到了一个确实比较糟糕的比特式估计方法。我们能做出一些什么改进呢？一个直接的想法是使用多个独立的哈希函数。如果每个哈希函数输出它自己的随机数据集，我们可以记录最长的前导 0 比特序列。然后在最后我们就可以对其求一个平均值以得到一个更精确的估计。

从实验统计上来看这给了我们一个相当好的结果，但哈希的代价是很高的。一个更好的方式是一个叫做随机平均的方法。相比使用多个哈希函数，我们仅仅使用一个哈希函数。但是把它的输出进行分割然后使用它的一部分作为桶序号来放到许多桶中一个桶里去。假设我们需要 1024 个值，我们可以使用哈希函数的前 10 个比特值作为桶的序号，然后使用剩下的哈希值来计算前导 0 比特序列。这个方法并不会损失精确度，但是节省了大量的哈希计算。

把我们目前学到的应用一下，这里有一个简单的实现。这和 Durand-Flajolet 的论文中的算法是等价的，为了实现方便和清晰所以我计算的是尾部的 0 比特序列。结果是完全等价的。

```
def trailing_zeroes(num): """Counts the number of trailing 0 bits in num.""" if num == 0: return 32 # Assumes 32 bit integer inputs! p = 0 while (num >> p) & 1 == 0: p += 1 return p def estimate_cardinality(values, k): """Estimates the number of unique elements in the input set values. Arguments: values: An iterator of hashable elements to estimate the cardinality of. k: The number of bits of hash to use as a bucket number; there will be 2**k buckets. """ num_buckets = 2 ** k max_zeroes = [0] * num_buckets for value in values: h = hash(value) bucket = h & (num_buckets - 1) # Mask out the k least significant bits as bucket ID bucket_hash = h >> k max_zeroes[bucket] = max(max_zeroes[bucket], trailing_zeroes(bucket_hash)) return 2 ** (float(sum(max_zeroes)) / num_buckets) * num_buckets * 0.79402
```

这很漂亮就像我们描述的一样：我们保持一个计算前导(或尾部)0个数的数组，然后在最后对个数求平均值，如果我们的平均值是 x ，我们的估计就是 2^x 乘以桶的个数。前面没有说到的是这个魔术数 0.79402。数据统计表明我们的程序存在一个可预测的偏差，它会给出一个比实际更大的估计值。这个在 Durand-Flajolet 的论文中导出的魔术常数是用来修正这个偏差的。实际上这个数字随着使用的桶的个数(最大 2^{64})而发生变化，但是对于更多数目的桶数，它会收敛到我们上面用到的算法的估计数字。大量更多的信息请看完整的论文，包括那个魔术数是怎么导出的。

这个程序给了我们一个非常好的估计，对于 m 个桶来说，平均错误率大概在 $1.3/\sqrt{m}$ 左右。所以1024个桶时()，我们大概会有 4% 的期望错误率。为了估计每篇最多 2^{27} 个数据的数据集每个桶仅需要 5 比特就够了。少于 1 kb [西安网页设计培训](#) 内存，这真的很赞($1024 * 5 = 5120$ ，即 640 字节)!

让我们在一些随机的数据上测试一下它：

```
>>> [100000/estimate_cardinality([random.random() for i in range(100000)], 10) for j in 西安java range(10)]
[0.9825616152548807, 0.9905752876839672, 0.979241749110407, 1.050662616357679, 0.937090578752079,
0.9878968276629505, 0.9812323203117748, 1.0456960262467019, 0.9415413413873975, 0.9608567203911741]
```

结果不坏，一些估计超过 4% 的预期偏差，但总而言之结果都很好。如果你自己再尝试一遍这个实验，请注意：Python 建的 hash() 函数将整数哈希为它们本身。导致运行像 estimate_cardinality(range(10000), 10) 这样的会给出偏差很大的结果，因为此时的 hash() 不是一个好的哈希函数。当然使用上述例子中的随机数是没有问题的。

改进准确度：SuperLogLog 和 HyperLogLog

虽然我们已经得到了一个非常好的估计，但它有可能做到更好。Durand 和 Flajolet 发现极端数值会很大地影响估计结果的准确度。通过在求平均前舍弃一些最大值，准确度可以得到提高。特别地，舍弃前 30% 大的桶，仅仅计算 70% 的桶的平均值，精确度可以用 $1.30/\sqrt{m}$ 提高到 $1.05/\sqrt{m}$! 这意味着在我们之前的例子中，用 640 字节的状态，平均错误率，4% 变成了大约 3.2%。但并没增加空间的使用。

最后，Flajolet et al 的论文的贡献就是使用了一个不同类型的平均数。使用调和平均数而不是几何平均数。通过这么做，我们可以把错误率降到 $1.04/\sqrt{m}$ ，同样不增加需要的空间。当然完整的算法要更复杂一点，因为它必须修正小的和大的基数误差。有兴趣的读者应该，可能你已经猜到了，就是去阅读完整的论文。

并行化

这些方案所共有的整齐性使得它们很容易就能并行化。多台机器可以独立地运行同样的哈希函数同样数目的桶。我们在最后只需要把结果结合起来，取每个算法实例中每个桶最大的值就可以了。这不仅很好实现，因为我们最多只需要传输不到 1kb 的数据就可以了，而且和在单台机器上运行的结果是完全一模一样的。

总结

就像我们刚刚讨论过的基数排序算法，使得有可能得到一个非重复数字个数的很好的估计。通常只用不到 1kb 空间。我们可以不依赖数据的种类而使用它，并且可以分布式地在多台机器上工作，机器间的协调和数据的传输达到最小。结果估计数可以用来做许多事情，比如流量监控(多少个独立IP访问过?)和数据库查询优化(我们应该排序然后归并呢还是构造一个哈希表呢?)。

相关的主题文章：

[1植发一个毛囊多少钱](#)

[植发多久恢复正常](#)

分享


转发

[上一主题](#) | [下一主题](#)

发帖

回复

返回

 高级编辑器

你需要登录后才可以发帖 [登录](#) | [注册](#)

发表回复

