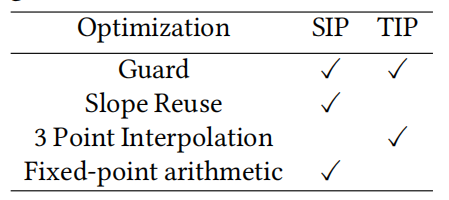
**《高级数据库技术及实践》期末报告**

论文题目：Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search? (SIGMOD 2019)

报告人：16337009 曾恺翔 16337110 匡乾

1. **论文概要回顾**

目前，广泛使用的，效率最高的搜索内存中的排序数组的算法是二分查找；本文使用插值查找的算法有性能与其它查找方法相比较差，在非一致数据集上表现差等缺点，但其可以通过复杂计算来减少内存访问的特点很吸引人。因此为利用其优点同时克服其弱点，本文引入了两个优化的插值查找方法，SIP(Slope reuse Interpolation)坡度再利用插值法和TIP (Three point Interpolation)三点插值法。其中SIP运用在一致数据集上，TIP运用在非一致数据集上。我们组将目光集中在了TIP上，主要对TIP进行了改进。在TIP中，相对于常规的插值查找，其优化点主要在Guard和3 Point Interpolation上。因此，出于改进TIP的考虑，我们分别对这两点进行了研究，其中，曾恺翔负责3 Point Interpolation方向的研究，匡乾负责Guard方向上的研究。

1. **复现、测试、实验环境**

复现：

测试：

本文的测试方法为：

从数据集中选出1000个值作为一个测试集

记录在数据集中查找这1000个值的时间，这称为一次测试

对每个算法，进行1,000,000次测试

为减小由缓存或者是一些硬件的因素带来的时间上的波动，在测试中会舍弃前30%的测试结果

最后对其取平均数，得到一个算法在一个数据集上的性能

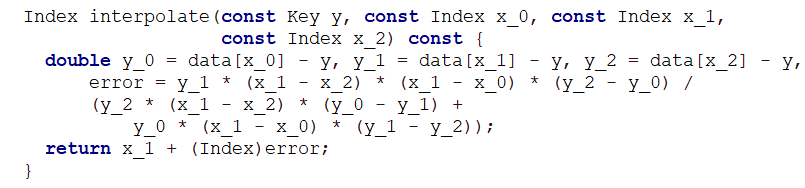
实验环境：

我们分别使用了各自的电脑进行了实验，因为是对不同算法进行比较，因此只要避免因为硬件的问题造成的数据波动即可，所以没有刻意地设置相同的实验环境，只是在运行不同算法时保证了各硬件处于相同的状态。

1. **三点迭代法的研究和尝试**

**3.1 原方法**

原方法代码：



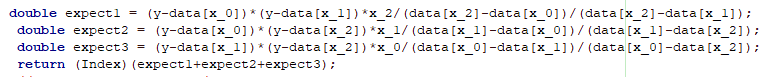
中心思想是将搜索看作一个解方程的问题。将其化成迭代方法，获得出用三个点来预测新的点的位置的方法。

**3.2 尝试其他方法**

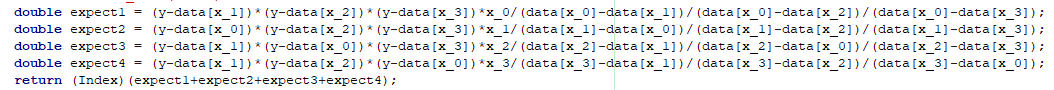
我们尝试了诸多方法，包括通过插值法、迭代法，来拟合出目标预测点。主要是改变了上述中的interpolate函数。

我们首先采用了拉格朗日插值，其要求必须过所有点，由于此插值是给出y要拟合x，所以我们将各点的x、y都调换了。

三点插值代码：

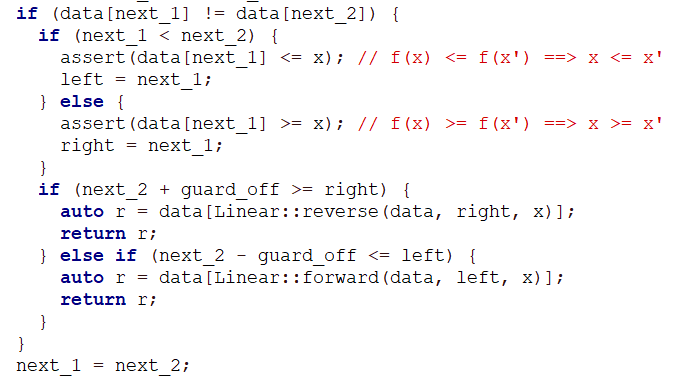


类似地，我们尝试了四点插值，代码如下：

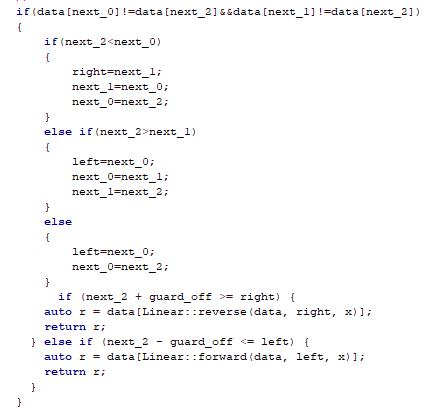


但是，从三点到四点的代码更改不只是更改函数，基本整个体系都变了。对于整个循环都是需要改变的，主要是原先的三点更新变成了四点更新。

原迭代代码：



现迭代代码：



首先我们选取了左右端点和两个四分位作为四个点，直观上比较符合逻辑。然后，我们每次将四个点进行了更新，如果预测点掉在左边，则选取最左边的四个点，否则选取最右边的四个点。最后，仍要注意预测点不能掉出新的四个点。

虽然最后此方法可以没有错误地运行，但是效率不如原方法。

同时，我们还尝试了牛顿插值、最小二乘拟合、埃特金加速收敛法、弦截法，效果都不太好，难以和原方法比拟。

**3.3 利用先验知识**

我们直接对原方法难以改进，但是，原方法仅仅是用了曲线拟合，并没有利用任何其他的先验信息。很显然地，如果我们知道数据的分布情况，并加以利用，很有可能会相较宽泛的方法更加有效率。

我们从这个角度再深入研究。

**3.3.1 完全信息**

我们先研究，当我们数据的分布相当规律并且已知，我们利用这样的有效信息应该会非常有效。

（1）线性完全信息

如果数据分布是非常线性地，我们利用这个信息，用相应的线性函数来拟合，并多次采用不同的线性函数来测试，获得其运行时间数据，结果如下图表：

（bs为二分查找法、tip为原论文中的三点拟合方法、newtip为新改进的方法，下同）

可见，利用了先验信息的方法远好于宽泛的曲线拟合，同时也优于二分查找法。

（2）指数型完全信息

如果数据分布是指数型的，我们利用这个信息，用相应的指数型函数来拟合，并多次采用不同的指数型函数来测试，获得其运行时间数据，结果如下图表：

可见，利用了先验信息的方法略优于曲线拟合，但略差于二分查找法。

（3）其他型完全信息

我们再试试其他类型的各种函数分布的数据，用相应的拟合方法，获得其运行时间数据，结果如下图表：

可见，二分查找法还是最好的，新方法波动性较大，也很难看出其有显著优于原方法。

**3.3.2含误差的先验信息**

现实中我们一般是不可能知道关于数据的所有分布信息的，数据也不可能非常一致地是某种函数分布，假如先验信息中含有误差，那么结果会如何？我们下面进行研究。

（1）线性、含误差

对于线性的分布，如果先验信息包含误差，那么利用先验信息的新方法表现如何？以下为存在20%、10%、5%、1%误差的情况下，各方法的运行表现。

（2）指数型、含误差

对于指数型的分布，如果先验信息包含误差，那么利用先验信息的新方法表现如何？以下为存在20%、10%、5%、1%误差的情况下，各方法的运行表现。

（3）其他型、含误差

对于其他类型的函数分布，如果先验信息包含误差，那么利用先验信息的新方法表现如何？以下为存在20%、10%、5%、1%误差的情况下，各方法的运行表现。

**3.3.3 先验信息拟合**

上述的先验信息都比较丰富了。一种现实的情况是，假若仅知道数据近似于某种分布，但不知道具体参数，这样的拟合，如果利用先验信息会如何？我们以指数型函数为例，做了实验。三种方法的运行时间如下图：

**3.4 总结与结论**

基于以上的研究，我们认为有以下结论：

（1）三点拟合总体上来说较二分查找的优势并不明显，其能否有效还取决于数据的分布情况。

（2）利用完全先验知识能够提高效率，但是该效果对先验知识的正确性非常敏感。不够正确的先验知识，如果盲目利用，会导致效率不增反降。

（3）在函数参数不确定的情况下，利用函数先验知识总体上对效率的改善不大，但是可以显著降低运行时间的波动率。

（4）总的来说，运算的代价相对较大，其节约的时间难以抵补所耗时间。



1. **guard\_size的研究和改进**
   1. **背景**

插值查找可以通过数据的分布信息快速定位目标值，但随着期望值不断地靠近目标值，这种收益会大大降低。当其计算的代价大于计算得到的收益时，就应该停止插值查找，改用其它更简单、代价更小的算法。在本文中，使用的是顺序查找这种方法。

在本文中，是否过于靠近目标值，需要切换算法的判断是由guard\_size决定的，而guard\_size的设置则来源于经验。

在实际实现中，guard\_size被默认设置为64。

因此，我就产生了一个想法：能否通过数据集的大小(dataset\_size)来调整guard\_size？这样会不会使得算法会有更好的性能？

* 1. **改进的实现方法**

对于不同的dataset\_size，我使用了多个guard\_size来进行对比，为了更好地检测不同的guard\_size的影响，我取了guard\_size=2n，详细信息见下表：

|  |  |
| --- | --- |
| DatasetSize:104  TimeNS  tip2 65  tip4 65  tip8 65  tip16 64  tip32 50  tip64 44  tip128 45  tip256 49  tip512 92 | Size_10000 |
| DatasetSize:105  TimeNS  tip2 119  tip4 100  tip8 88  tip16 97  tip32 99  tip64 84  tip128 86  tip256 56  tip512 78  tip1024 79  tip2048 116  tip4096 236 | Size_100000 |
| DatasetSize:106  TimeNS  tip2 213  tip4 207  tip8 202  tip16 189  tip32 205  tip64 210  tip128 185  tip256 176  tip512 142  tip1024 146  tip2048 158  tip4096 180  tip8192 312 | Size_1000000 |
| DatasetSize:107  TimeNS  tip2 277  tip4 278  tip8 281  tip16 252  tip32 223  tip64 217  tip128 212  tip256 204  tip512 190  tip1024 176  tip2048 166  tip4096 169  tip8192 180  tip16384 252 | Size_10000000 |
| DatasetSize:108  TimeNS  tip64 391  tip128 380  tip256 327  tip512 321  tip1024 310  tip2048 317  tip4096 291  tip8192 272  tip16384 283  tip32768 333  tip65536 408 | Size_100000000 |

在处理数据的过程中，我删除了一些性能不理想的值或者是不合理的值，如(DatasetSize, guard\_size)= (104, 1024), (108, 2)；没有测试DatasetSize < 104 的原因是，每次测试要选择1,000个数据作为测试集，若数据集太小，则插值查找的意义不大，甚至性能上可能不如直接进行顺序查找；没有测试DatasetSize > 108 的原因是，数据量过大，机器表现不佳，需要更好的硬件配置，同时从已有的数据结果已经能初步地得到结论，再扩大DatasetSize的必要性不大。

* 1. **实验结果**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DatasetSize | 默认guard(64)性能(ns) | 最佳guard性能(ns) | 提升性能比 | 最佳guard |
| 104 | 44 | 44 | 0% | 64 |
| 105 | 84 | 56 | 33% | 256 |
| 106 | 210 | 142 | 32% | 512 |
| 107 | 217 | 166 | 24% | 2048 |
| 108 | 391 | 272 | 30% | 8192 |

从表格中，很容易地看出，随着DatasetSize不断增大，最佳guard的取值也不断变大；同时也很容易发现，在DatasetSize足够大时，提升的性能比大概在30%左右，这是比较可观的一个数据。同时，通过分析，我也总结出一个最佳guard的公式： , 

* 1. **讨论与总结**

我得出的结果当然是一个稍显粗糙的结果，主要有以下三个方面的不足：

1. 对于大型数据集（DatasetSize>108）没有进行测试，不能保证上述经验公式在这种情形下也成立
2. 只研究了DatasetSize对guard的影响，其它影响因素，如：数据的shape，不同分布的数据，没有在实验中体现
3. 只研究了tip的guard的影响，没有去研究此结论在sip上是否成立

在将来的研究中若是补足这三个部分，相信就能算是在guard这一点上研究的透彻了。当然通过我的这一点改进，可以有效地提升大约30%的性能，这是很可观的。