HNSW + 并行编程实验报告

邓杰天 522031910441

2024 年 5 月 28 日

1 背景介绍

本实验基于 HNSW 来实现向量数据库，并寻找目标结点的临近结点。NSW 将数据库的向量与接近的向量相连，形成一个连通图，查询时从起始结点出发，不断跳到更靠近目标节点的邻居节点，直到无法再靠近为止，得到的终点结点即为查询结果。而 HNSW 更进一层，其借鉴了跳表的思想，采用分级的方式存储特征向量，使得导航时能在高层次快速靠近目标节点，防止数据库过于庞大带来的时延陡增。在本次实验中，主要实现了 HNSW 算法的两个基本操作，即 insert 和 query，分别用于插入新的向量和查询与目标相近的 N 个向量。

实际的向量数据库有着非常庞大的访问量，如果只是串行处理，会导致用户之间的冲突，引发不可忍受的时延。使用并行处理可以充分利用现代服务器的多核架构，同时支持多个用户同时查询。本次实验实现了查询的并行算法。

2 系统实现

2.1 HNSW 基本操作

HNSW 有两个基本的操作，分别为 insert 和 query，接下来分别介绍如何实现：

insert 分为以下几步：1. 需要计算新结点的插入层级，将 label 对应的向量放入到 vector 数组中，该数组用于存放所有向量的坐标，便于计算向量的欧氏距离；同时需要使用提供的辅助函数获取新结点的 level. 2. 从数据库的最高层搜索到 level+1，在每一层使用 search_layer 找到当前层中最接近的一个邻居节点，放入到集合 newNeighbors 中，并使用该集合最近的点作为新的入口。3. 从 level 向下搜索，这时就需要在每层都维护最近的 efConstruction 个节点并存储起来。4. 更新入口，如果 level > maxL，说明该节点更高，应当作为新的入口节点。

query 与 insert 相当类似：1. 初始化，即找到入口节点，同时为了方便后续计算，将查询结点的 label 假设为-1，并也要放入到 vector，注意这个 vector 实际上是一个 unordered_map 型。2. 从最高层搜索到-1 层，每次也是使用 search_layer 找到当前层最近的一个点放入集合

中，并找到新的入口节点以搜索下一层。3. 在 level 0 中找到最近的 efConstruction 个节点，并挑选最近的 k 个节点返回。

最后介绍两个辅助函数：search_layer 和 select_neighbors，前者用于寻找当前层临近的 n 个节点，后者用于寻找给定集合中最近的 n 个节点。之所以要区分，是因为两者的实现有一定区别，前者是借助了每个节点存储起来的邻居节点集合，不断探索已访问节点的邻居中最近的那一个来加入集合，相当于是通过了结点的链接网来查询的；后者直接遍历集合中的所有节点并计算欧氏距离。因此，先使用 search_layer 找到多于 M_max 的结点，这些节点只能说和目标节点联通的边更少，再用 select_neighbors 进一步查找，其更精确但是时间复杂度大。综合两个辅助函数，可以在更小的时延下尽量找到最近的节点。

2.2 并行优化

本实验只在 test.cpp 中提供了并行优化的查询，该优化为每个查询都分配了一个线程，这些线程被压入在 vector<std::thread> threads 中，因此 thread_id 被指定为迭代器 i，从 0 到 99。为了防止互斥锁带来的巨大开销，需要避免多个线程对同一全局变量的修改。在原来的 query 中，每次查询都会将向量存放在 vector[-1] 中，但这样就会修改同一个变量，导致互斥。为了避免这个问题，提供了一个新的查询函数 concurrent_query，和 query 基本一致，除了会将-(thread_id+1) 作为 vector 的索引使用，这样就不会修改同一变量，不必加入互斥锁。另一方面，concurrent_query 的返回值需要压入 concurrent_test_gnd_l 以便后续计算，因此也不可以再使用 emplace_back，而是在 resize 分配好空间后，再放入 thread_id 索引对应的位置。

3 测试

3.1 参数 M 的影响

3.1.1 测试配置

修改 M 和 M_max 的值，每个值测试 5 次，记录不同 M 值下的查询召回率和单次查询的时延变化。

3.1.2 测试结果

结果如下表：

M = M_max	recall	time1	time2	time3	time4	time5	average time
10	0.712	4.5	5.0	4.7	4.6	4.6	4.68
20	0.976	8.3	8.3	9.3	9.1	8.8	8.76
30	0.937	9.6	9.4	9.3	9.3	9.9	9.5
40	0.987	10.8	10.8	10.8	10.8	11.0	10.84
50	0.990	12.3	12.9	12.3	13.8	12.2	12.7

3.1.3 结果分析

注意到对于串行编程，每次测试的召回率都不会变化，同时单次查询时延也比较稳定。除了 M = 30 的情况之外，召回率基本是随 M 的增大而增大的，这是因为 M 增大后，每个节点的邻居结点集合就更大，更容易找到最近节点；但也有限制，即计算 level 的参数 mult_ 与 M 成反比，层数下降不仅会导致查询时延上升，还可能导致查询时入口节点移动次数不足，可能会是精度有很小的下降。对于单次查询时延，其与 M 成正比关系，一方面，M 的增大导致集合中需要加入的元素更多；另一方面，M 增大可能导致 level 数下降，使得跳表性能不够好，不能在高层就通过较少的移动找到更近的位置。

3.2 性能测试

3.2.1 测试配置

编写的 test.cpp 程序可以进行测试，同时并行优化也是在这里才实现的。该程序每插入 1000 个节点后，便会报告当前 1000 个节点的平均插入时延，共报告 10 次。接下来将会串行查询 100 个结点，并报告这 100 次查询的单次查询时延。最后使用并发编程，为每个查询操作都分配一个线程，并报告这 100 次查询的单次查询时延。

3.2.2 测试结果

对 M=M_max=30 的情况进行测试，输出结果如下：

load ground truth
load query
load base
inserting
from 0 to 999: single insert time 3.0 ms
from 1000 to 1999: single insert time 5.9 ms
from 2000 to 2999: single insert time 8.7 ms
from 3000 to 3999: single insert time 8.5 ms
from 4000 to 4999: single insert time 8.6 ms

from 5000 to 5999: single insert time 8.3 ms
from 6000 to 6999: single insert time 9.1 ms
from 7000 to 7999: single insert time 9.4 ms
from 8000 to 8999: single insert time 10.2 ms
from 9000 to 9999: single insert time 10.9 ms
serial querying
average recall: 0.937, single query time 9.3 ms
current querying
average recall: 0.937, single query time 6.1 ms
对于查询时延，同样测试 5 次，结果如下：

s/c	time1	time2	time3	time4	time5	average time
sequential	9.6	9.4	9.3	9.3	9.9	9.5
concurrent	6.5	7.7	5.8	6.1	6.9	6.6

3.2.3 结果分析

结果表明，随着插入节点数的上升，单次插入时延也在上升，且开始时上升的更快，这是因为前期结点少，可能填不满集合要求的 M 个结点，且此时 level 也上升的比较明显。对于查询时延，串行查询的性能比并行查询要差上不少，符合预期。并行查询让多个查询能同步进行，同时充分利用了多核的计算资源，因此性能会表现得更好。另一方面，并行查询的时延稳定性较差，这和线程执行顺序的不确定性有关，但串行和并行的召回率始终是相等且不变的，说明程序的编写正确。

4 结论

本次实验实现了 HNSW 算法，有助于了解现代的数据库查询算法。该实验也考查了并行编程，加深了对并行优化的理解。