522031910747+李若彬+lab2

基本实现细节

类的声明与成员变量以及方法实现

`HNSW`类继承自`AlgorithmInterface`，其主要成员变量包括：

- `enter_point`和`enter_level`：记录了当前图的入口点和入口层级。
- `neighbors`：三维向量，存储了每个节点在各层级的邻居信息。
- `Node`：一个哈希表，存储了节点的向量数据。

插入方法

`void HNSW::insert(const int *item, int label)`

查询方法

`std::vector<int> HNSW::query(const int *query, int k)`

搜索层方法

`std::vector<int> HNSW::search_layer(const int *q, int ep, int ef, int lc)`

搜索层方法在指定层级上搜索最近邻：

1. 维护两个优先队列，一个用于扩展节点（`C`），一个用于存储当前最近的节点（`W`）。
2. 从入口点开始，扩展其邻居节点，并更新两个优先队列。
3. 返回优先队列`W`中的节点。

选择邻居方法

`std::vector<int> HNSW::select_neighbors(const int *q, std::vector<int> &W, int M, int lc)`

选择邻居方法从候选集`W`中选择`M`个最近的节点作为邻居：

1. 将所有候选节点按距离排序。
2. 选择前`M`个节点作为邻居并返回。

初始化邻居方法

`void HNSW::init_neighbors(int L, int maxL, int label)`

初始化邻居方法用于初始化新节点的邻居向量：

1. 为新层级分配空间。
2. 确保每一层的邻居向量大小足够容纳新节点。

并行优化

引入多线程查询

`query_thread` 函数

这个函数用于在独立的线程中执行查询操作。它接收如下参数：

- `HNSW *hnsw`：指向HNSW对象的指针，用于执行查询。
- `std::vector<std::vector<int>> *test_gnd`：指向存储查询结果的二维向量。
- `const int *query`：指向查询向量的指针。
- `int k`：查询返回的最近邻数量。
- `int start` 和 `int end`：指定当前线程处理的查询向量的起始和结束索引。

函数内部使用循环遍历 `start` 到 `end` 范围内的查询向量，并将查询结果存储到 `test_gnd` 中对应的位置。

`createAndJoinThreads` 函数

这个函数负责创建和管理多个线程以并行执行查询操作。

- 它将所有查询向量分成若干子集，每个子集由一个线程处理。
- 使用 `std::thread` 创建线程，并将 `query_thread` 作为线程执行的函数。
- 在所有线程启动后，通过调用 `join()` 方法确保主线程等待所有子线程完成。
- 由于Linux虚拟机核心数为4，因此选择分配4个线程。

```
void createAndJoinThreads(HNSW& hnsw, std::vector<std::vector<int>>&
test_gnd, const int* query, int gnd_vec_dim) {
    std::vector<std::thread> sub_threads;
    for(int i = 0; i < 4; i ++) {
        int start = i * 32;
        int end = (i + 1) * 32;
        sub_threads.push_back(std::thread(query_thread, &hnsw, &test_gnd,
query, gnd_vec_dim, start, end));
    }
    for(auto& thread : sub_threads) {
        thread.join();
    }
}
```

并行化的优势

- **降低延迟**：多线程允许多个查询同时进行，大大减少了总查询时间。
- **提高吞吐量**：并行处理可以同时处理更多的请求，增加了系统的整体处理能力。
- **充分利用多核CPU**：现代服务器通常有多个CPU核心，通过多线程可以更好地利用这些计算资源。

实验测试结果

5.2 参数M的影响

	查询时延(ms)	召回率
M=M_max=10	0.268	0.740
M=M_max=20	0.320	0.976
M=M_max=30	0.399	0.953
M=M_max=40	0.442	0.987
M=M_max=50	0.470	0.990

查询时延

召回率随着 M = M_max 的增加也有显著的提高。较大的 M = M_max 提供了更密集的图结构，从而提高了查询时找到相关节点的概率。具体表现如下：

- 当 M = M_max = 10 时，查询时延是 0.268 ms
- 当 M = M_max = 20 时，查询时延增加到 0.320 ms
- 当 M = M_max = 30 时，查询时延进一步增加到 0.399 ms
- 当 M = M_max = 40 时，查询时延是 0.442 ms
- 当 M = M_max = 50 时，查询时延达到了 0.470 ms

可以看出，查询时延随着 M 和 M_max 的增加呈现出线性增长的趋势。

召回率

召回率随着 M 和 M_max 的增加也有显著的提高。较大的 M 和 M_max 提供了更密集的图结构，从而提高了查询时找到相关节点的概率。具体表现如下：

- 当 M = M_max = 10 时，召回率是 0.740
- 当 M = M_max = 20 时，召回率大幅增加到 0.976
- 当 M = M_max = 30 时，召回率稍微下降到 0.953
- 当 M = M_max = 40 时，召回率又回升到 0.987
- 当 M = M_max = 50 时，召回率略微提升到 0.990

可以看出，在 M = M_max 增加到 20 时，召回率已经接近 1，之后的增加对召回率的提升效果减弱。这说明适当增加 M = M_max 能够显著提高召回率，但超过一定值后，提升效果变得不明显，甚至在某些情况下可能会略有下降。

5.3 性能测试

M=M_max	10	20	30	40	50
串行查询时延(ms)	0.268	0.320	0.399	0.442	0.470
2并行查询时延(ms)	0.078	0.090	0.098	0.106	0.123

M=M_max	10	20	30	40	50
4并行查询时延(ms)	0.076	0.095	0.135	0.127	0.133
8并行查询时延(ms)	0.114	0.105	0.122	0.229	0.133

数据分析

1. **串行查询性能**：
   - 随着M的增加（即最大邻居数的增加），串行查询时延（ms）逐渐增加。这是预期的结果，因为较大的M值意味着更多的邻居节点需要检查，从而导致查询时间的增加。
2. **2并行查询性能**：
   - 2并行查询时延显著低于串行查询时延，并且在不同M值下波动不大。这表明并行化显著提升了查询性能。
   - 但随着M的增加，查询时延仍然有所增加，这与串行查询的趋势一致，但增加的幅度较小。
3. **4并行查询性能**：
   - 4并行查询时延在M=10和M=20时与2并行查询时延相近，甚至略高，但整体趋势仍然比串行查询更快。
   - 然而，在M=30、M=40和M=50时，4并行查询时延反而比2并行查询时延要高，特别是M=30时，时延明显增大。
4. **8并行查询性能**：
   - 8并行查询时延在低M值时（M=10, M=20）仍然较低，但随着M的增加，时延大幅波动，特别是在M=40时达到了0.229ms。
   - 总体上，8并行查询的性能不稳定，在某些情况下甚至比4并行查询和2并行查询的时延还要高。

原因分析

1. **串行查询时延增加的原因**：
   - 随着M值的增加，需要处理的节点数增多，查询路径变长，从而导致查询时延增加。
2. **并行查询时延较低的原因**：
   - 并行化能有效利用多核处理器的能力，分摊查询任务，从而加快查询速度。2并行查询时延显著低于串行时延，证明了这一点。
3. **4并行和8并行查询时延反而增加的原因**：
   - **线程调度开销**：并行线程数增多时，线程间的调度和同步开销增加，可能会抵消并行化带来的性能提升。
   - **资源竞争**：更多的并行线程会导致CPU缓存和内存带宽等资源的竞争，进而影响性能。
   - **负载均衡问题**：如果数据集或查询任务不能很好地分配给各个线程，某些线程可能会比其他线程处理更多的工作，从而导致整体时延增加。
   - **处理器核数限制**：实验在4核处理器上进行，超出4线程的并行度（如8并行查询）将引发线程争用，导致性能下降。8并行查询时，系统需要处理更多的线程调度和上下文切换，从而增加了额外的开销。