Project阶段3: Smart LSM with HNSW

强烈建议：在开始实验之前，先**认真**阅读本文档，确保了解了实验的目的、内容和要求后再开始动手编写代码。

Introduction

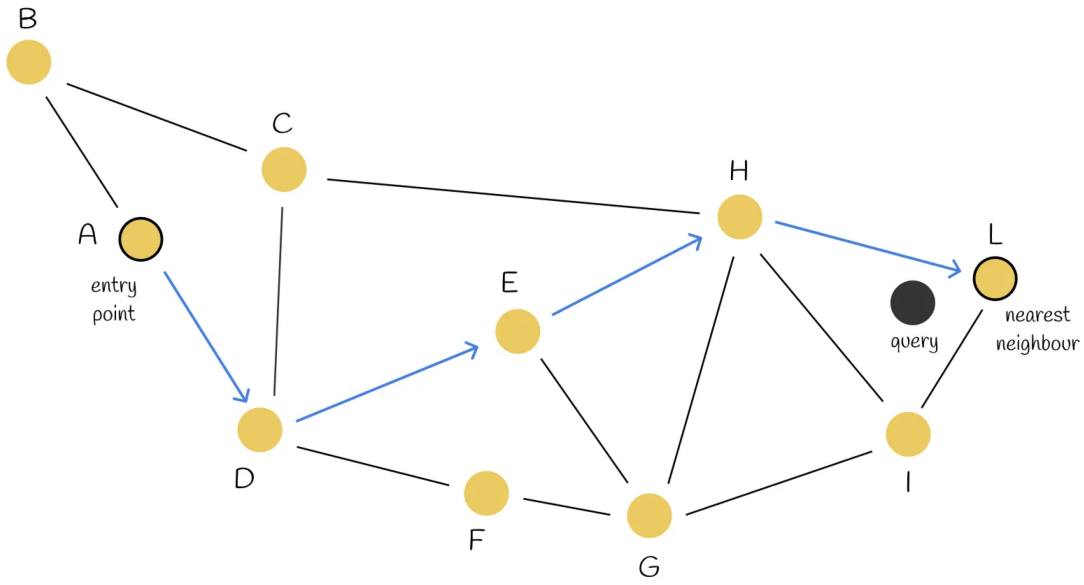
现代存储系统面临非结构化数据处理的重大挑战。**LSM-tree（Log-Structured Merge Tree** 作为高性能存储引擎，通过顺序写优化和层级合并机制，在键值存储领域占据重要地位。然而传统LSM-tree仅支持精确键值查询，难以应对**语义相似性搜索**需求。

本阶段将基于上一阶段的工作，实现一个高效的近似最近邻搜索系统。我们将使用`HNSW（Hierarchical Navigable Small World）`算法来提高搜索效率。

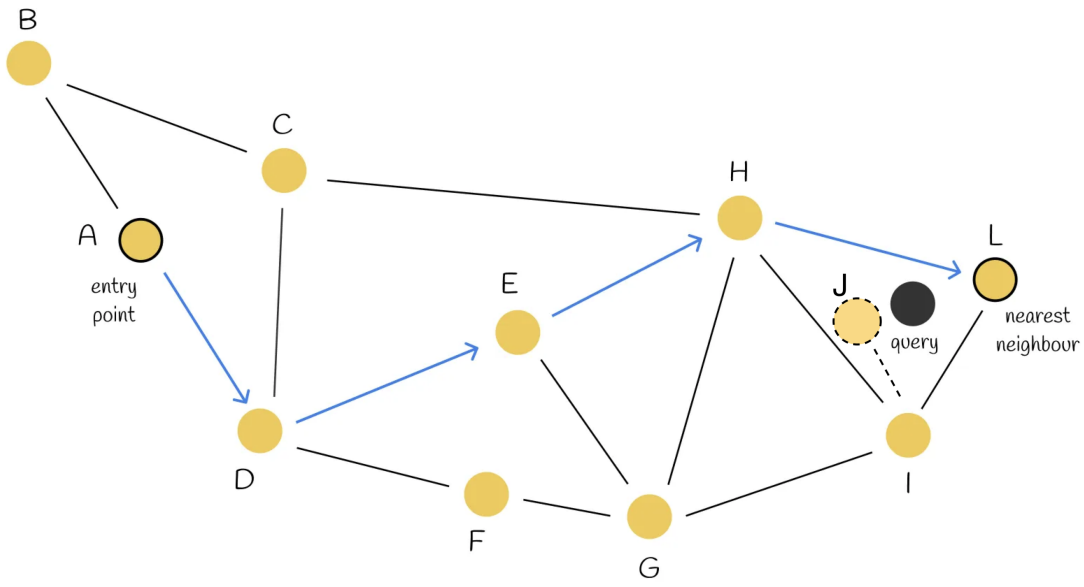
Background

本节介绍HNSW的基本思想与结构。HNSW是基于NSW（Navigable Small World）进行优化的。为便于理解，我们首先介绍NSW的结构，并以此为基础介绍HNSW的结构。

NSW的核心思想是将数据库中的向量与接近的向量相连，形成所谓的"Small World"，然后在这个连通图上从某个起始节点开始，通过不断选择能够靠近目标节点的边进行导航，直到无法再靠近目标节点为止。这个过程被称为导航（Navigation）。如下图所示为NSW的基本结构示意图。每个黄色节点代表一个特征向量，并与较为接近的向量相连。黑色节点表示被查询的目标向量，若要搜索一个目标向量的相似向量，从A向量出发，导航过程分别经过A、D、E、H，最后停在L。由于L的邻边无法再使目标节点更靠近，因此L即为最终的查询结果。

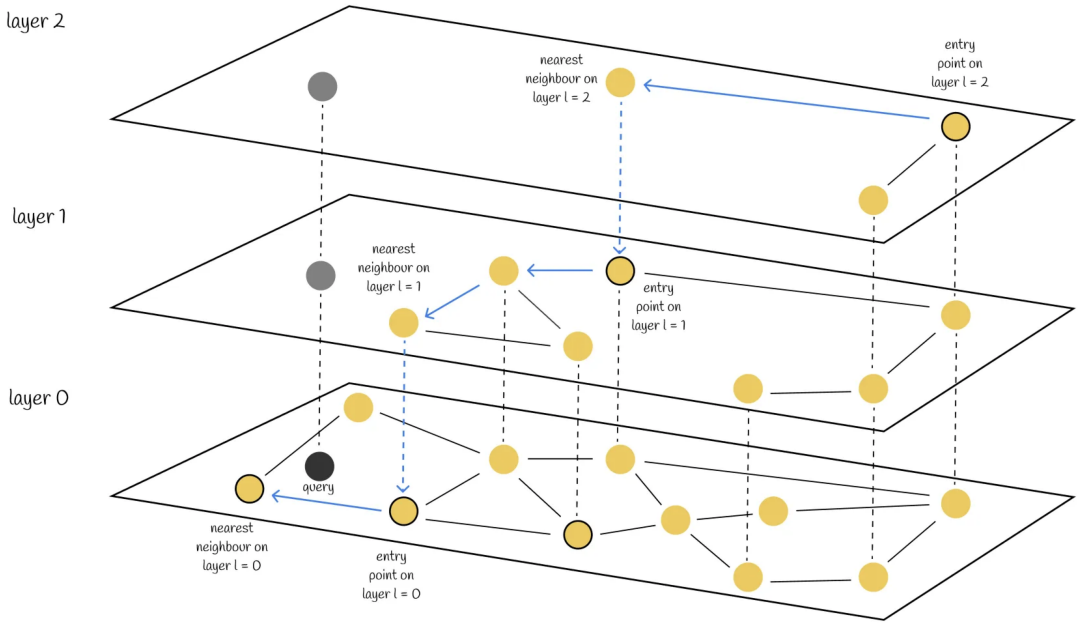


注意：通过这个算法并不一定能找到最接近的N个向量，例如下图中节点J比L更接近目标节点，然而L与其之间并没有边，则L不是最接近的向量，但仍然是较为接近的向量，这在推荐系统等应用中是可以被接受的。



上述查询方式相比精确地找到最接近的相似向量能够大大提升效率。然而，当数据库中的向量数量特别庞大时，若起始节点距离目标节点很远，导航过程中的延迟也会显著增加。在这种情况下，HNSW借鉴了跳表的思想，采用分级存储特征向量的方式。在高层级存储较少的向量，这使得导航过程能够在高层级快速地靠近目标节点。

下图展示了HNSW基本结构的示意图。导航过程从入口节点（entry point）开始，在较高层级尽可能向目标节点靠近。如果无法继续靠近，则下降到下一层级的相同节点，直到最终下降到最底层（layer 0）并完成查询。



Details of HNSW

本节详细介绍HNSW的两个基本操作及其具体算法。HNSW的基本操作包括：

- insert：用于在建立索引过程中插入新的向量。
- query：用于在索引中查询与目标向量相近的N个向量。

Insert

本小节介绍了HNSW插入的流程。首先，介绍HNSW算法中的几个配置参数，这些参数会影响HNSW搜索的精确度和性能。：

> M(推荐默认值6): 在插入过程中，被插入节点需要与图中其他节点建立的连接数
> M_max(推荐默认值8): 每个节点与图中其他节点建立的最大连接数。随着新节点的插入，图中旧节点连接数可能会超过该值，需要相应地删去部分边
> efConstruction(推荐默认值30): 由于插入、查询时需要搜索一批相似向量，因此在搜索过程中需要维护一个候选节点集合，efConstruction为候选节点集合的数量
> m_L(推荐默认值6): 在决定节点最高会被插入到HNSW哪一层级时使用的正则化参数。m_L用来控制节点的层数分布。较大的m_L会导致更多的节点被插入到高层级，从而增加搜索效率，但也会增加内存开销。

insert过程主要分为两步：

- 第一步：设被插入到节点为q，q被插入的层数l应该在0到m_L之间随机选择。接着，自顶层向被插入节点q的层数l逐层搜索，一直到l+1，在每一层导航到与节点q相对接近的节点，将其加入最近邻元素集合W，并从W中挑选最接近q的节点作为下一层搜索的入口节点，这一过程与Background中只需查找一个相似向量的图例相同。
- 第二步：自l层向第0层逐层搜索，维护当前层搜索到的与q最近邻的efConstruction个点，并且在这efConstruction个点中选取最近的M个点去和q建立连接。注意，维护efConstruction个点的过程应采取类似BFS的策略，不断从最近的点向外延着边出发，efConstruction为你此时维护的优先队列/堆的大小。如果仅直接只找M个最近邻，很容易陷入“局部最优”，因为图结构的限制，可能有些更接近的点没有直接连到入口点，只有通过多步跳转才可能发现。
- 。注意由于链接是双向的，因此需要检查被链接的邻居们的点有没有超过M_max，如果超过了M_max，则需要删除掉距离最远的点。

注意：由于大家的电脑配置不尽相同，参数可以按照自己的电脑配置进行调整。

Query

本小节介绍查询目标节点的相邻节点的算法。查询过程与插入过程相近，query过程同样分为两步：

- 自顶层向第1层逐层搜索,每层寻找当前层与目标节点q最近邻的1个点赋值到集合W，然后从集合W中选择最接近q的点作为下一层的搜索入口点。
- 假设要查找的是最近的k个节点。接着在第0层中，查找与目标节点q临近的efConstruction个节点，其中选取k个最接近q的节点作为最终结果。注意：在第0层中，查找与目标节点q临近的efConstruction个节点时，可能会找到比k个更多的节点，因此需要对这些节点进行排序，选取前k个最接近q的节点作为最终结果。

第二步得到的节点即为返回结果。

Tasks

- 你需要实现接口 `std::vector<std::pair<std::uint64_t, std::string>> KVStore::search_knn_hnsw(std::string query, int k)`，该接口需要完成上述的HNSW算法，接受一个查询字符串和一个整数 `k`，返回与查询字符串最相近的 `k`个向量的key和value。并且按照向量 余弦相似度从高到低的顺序排列。

- 你应修改E2E_test.cpp中的search_knn为search_knn_hnsw，并运行测试用例来验证你的实现。你的正确性应该与你自己上一阶段的search_knn相比accept rate降低不超过10%。
- 在这一阶段，你需要保证key、value持久化，但是这个阶段仍不需要考虑向量的持久化。
- 请把向量存储在内存中，如果每次KV操作都重新计算向量，性能会很差。尽量减少重复计算。

作为一个project，你只需要完成要求的接口以保证测试用例能够正常运行，我们不对你的实现细节要求做过多限制。

Tips

- 如果你的电脑有GPU，你可以尝试使用GPU加速的embedding模型，提升embedding速度。这一部分可以参考llama.cpp的文档。

Report

由于不同的电脑配置，模型精度与运行时间可能会有差异。我们这次报告，需要你在你的电脑上测试HNSW带来的性能提升与accept rate下降。

- 请以data/trimmed_text.txt的每一行作为一个句子输入进KVStore；以data/test_text.txt的每一行作为查询语句。以你在phase2中实现的search_knn作为基准（即标准答案），测试HNSW的性能提升与accept rate下降。由于embedding模块较为耗时，在计算性能提升时，请去除掉embedding的时间再进行比较。
- 尝试不同的M、efConstruction、m_L等参数，观察对性能和准确率的影响（请同样去除掉embedding的时间之后再进行比较）。

你需要在报告中附上你的测试结果，与其他完成过程中遇到的问题、解决方案、思考等。

注意 在phase2阶段中，很多同学反映电脑速度运行embedding模型不够快导致很难调试。所以在这个阶段，可以按照你的电脑硬件情况、运行速度，参考E2E_test.cpp，**适当**修改insert和query的次数来方便你进行实验、撰写报告。

Submission

请将你的代码打包成 zip格式，命名为 学号_姓名.zip，例如 523030912345_张三.zip。你的压缩包不应包含third_party文件夹。

此外，请将你的报告命名为 学号_姓名_report.pdf，在canvas上单独的报告提交链接进行提交。