基于 LSM-TREE 的键值存储的优化（HNSW）报告

523031910852 朱子墨

2025 年 4 月 27 日

1 背景介绍

LSM-Tree（Log-Structured Merge-Tree）是一种适用于高吞吐量写入的存储结构，由 Patrick O’Neil 等人于 1996 年在论文中首次提出。主要用于数据库（如 LevelDB、RocksDB）和键值存储系统（如 Cassandra）。它通过批量写入和分层合并（Compaction）来优化磁盘 I/O，从而提高写性能。

LSM-Tree 尤其适合写密集型场景（如日志存储、物联网设备数据、时序数据库），其设计哲学体现了” 写优化” 与” 读优化” 的权衡，成为大数据时代高性能存储系统的基石之一。现代存储系统面临非结构化数据处理的重大挑战。LSM-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。

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。

2 测试

2.1 实验设置

由于在 windows 系统上进行的环境配置遇到了很多困难，因此最终在 VMware Workstation 的 linux 系统虚拟机上完成实验。实验环境基本参数如下：

在实验第二阶段的基础上进行代码的修改，我构建了 SearchLayers 类数据结构用于分层的查询。核心思想类似于跳表（SkipList）和无向图的结合。其核心操作包括插入（insert）和查询（query）。

项目	参数
内存大小	13.1GB
硬盘大小	50.0GB
处理器数量	2
操作系统	Ubuntu 22.04.1 LTS
主要计算支持	CPU

表 1: 实验环境基本参数

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　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。

query 操作同样分为两步：

第一步：从顶层向第 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。

第二步：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　排序，选取前 k 个最接近 q 的节点作为最终结果。

完善数据结构之后开始补充对应的接口代码，然后进行正确率和搜索操作平均时间的测量。

2.2 预期结果

相比较于原来的简单的遍历查找，HNSW 算法可以提高时间性能，但是正确率会出现下降。这是由于在搜索过程中可能出现“局部最优但非全局最优”的现象，造成正确率下降。

理论上讲，正确率和时间性能受到 M（每个节点加入时默认连接的节点数量），M_max（最多连接的节点数量），m_L（层数），efConstruction（维护的邻近节点集合大小）四个参数影响。

对于正确率而言，M，M_max，efConstruction 均与其成正相关关系；这是由于连接的节点数增加可以在 query 时连接到更多近距离的节点，减少“距离很近但是没有连接”情况的出

现；而 efConstruction 较大时可以减少“局部最优但非全局最优”现象的出现。m_L 则很可能与正确率关系不大。

对于时间性能而言，M,M_max,efConstruction 与其成负相关关系，因为更多的连接意味着在层级搜索过程中，寻找下一层的入口会花费更多的时间；而保留维护更大的临近集合会付出更大的时间开销。对于 m_L，过小的 m_L 会造成难以跳过不相关的节点；过大的 m_L 会造成在层级查询时花费更多时间；同时，m_L 的效果可能和测试集规模有关。

此外，数据本身的性质、分布、内容也会极大影响算法的时间性能和正确率。

2.3 实验结果与分析

使用旧版本的测试集数据,默认参数设置选择 M = 6,M_max = 8,m_L = 6,efConstruction = 50，测试集大小 = 1100。由于生成的 level 具有随机性，因此正确率和时间可能出现浮动，这里我们采取多次测量取平均值的方法。

首先我们测试 HNSW 算法的优化效果：

测试参数	30	40	50	60	80	100	200	500
测试集大小	330	440	550	660	880	1100	2200	5500
通过率	优化前							
	57.58%	60.91%	72.36%	81.67%	89.20%	77.00%	60.32%	54.75%
	优化后							
	57.58%	60.91%	72.18%	73.94%	78.30%	71.27%	58.55%	54.22%
平均每次查找用例耗时（微秒）	优化前							
	29.33	39.43	50.14	55.51	68.29	93.06	172.76	440.66
	优化后							
	49.27	72.75	78.05	79.03	82.29	92.06	87.17	87.89
吞吐率	优化前							
	34094.78	25361.40	19944.16	18014.77	14643.43	10745.76	5788.38	2269.32
	优化后							
	20296.33	13745.70	12812.30	12653.42	12151.98	10862.48	11471.84	11377.86

表 2: 正确率和时间性能的对照

再使用新版本的测试文件，根据实际调试情况，修改参数 M = 10，M_max = 15，m_L = 4，efConstruction = 50，同时修改 E2E_test 文件，得到以下表格：

测试参数	30	50	80	120	200
测试集大小	90	150	240	360	600
通过率	优化前				
	26.67%	40.67%	60.83%	83.33%	25.83%
	优化后				
	26.67%	40.67%	57.50%	70.88%	24.67%
平均每次查找用例耗时（微秒）	优化前				
	26.10	42.80	61.26	102.69	162.56
	优化后				
	57.50	82.36	92.95	95.66	89.50
吞吐率	优化前				
	3831.42	2336.45	1632.39	973.80	615.16
	优化后				
	1739.13	1214.18	1075.85	1045.37	1117.32

表 3: 正确率和时间性能的对照（新数据集）

可以从表格中看出，优化后的算法正确率要低于优化前；时间性能上，在数据测试集比较小时，由于 HNSW 数据结构本身维护的时间开销，造成其时间性能差于优化前；不过，当数据测试集很大时，优化后的时间性能优势体现明显，这是由于原有的算法的查找时间随数据规模的增大而线性增加。而优化后的算法在数据集很大时平均查找时间仍然较为稳定。

新数据集表格中，仅在参数 120 左右正确率较高；当参数增大或降低时，正确率急剧下降，低于旧数据文件的测试结果。而此时相比较于优化前，优化后的正确率下降幅度并不是很大。因此，对于正确率较低的原因，我认为很有可能和数据测试集本身、embedding 底层逻辑和 E2E_test 的更新存在关联。经过实验发现，利用新 E2E_test 测试旧的测试文件正确率非常低；而旧有的 E2E_test 测试新的文件正确率也非常低，因此推断正确率较低的原因可能和测试数据、embedding 底层逻辑关系更大。

出于避免未知因素带来的严重误差和保证一个较高的正确率的考虑，下面对于参数的影响的探讨均采用旧的测试文件和测试集进行。

使用控制变量法研究四个参数的影响：

M	3	4	5	6	7
通过率	69.09%	71.45%	70.27%	71.27%	72.64%
平均每次查找用例耗时（微秒）	76.86	88.39	91.33	92.06	94.32
吞吐率	1301.07	1131.35	1094.93	1086.25	1060.22

表 4: M 对于正确率和时间性能的影响

和预测的结果基本相符，这里可以看出 M 和通过率整体呈正相关；当通过率较大时增长不明显。而时间性能则和 M 呈现负相关关系。

M_max	6	7	8	10	12
通过率	68.36%	69.73%	71.27%	71.73%	72.09%
平均每次查找用例耗时（微秒）	75.11	91.31	92.06	93.51	101.75
吞吐率	1331.38	1095.17	1086.25	1069.40	982.80

表 5: M_max 对于正确率和时间性能的影响

和预测的结果基本相符，这里可以看出 M_max 和通过率整体呈正相关；当通过率较大时增长不明显。而时间性能则和 M_max 呈现负相关关系。

efConstruction	20	30	50	60	80
通过率	61.55%	65.18%	71.27%	73.82%	77.64%
平均每次查找用例耗时（微秒）	64.41	68.94	92.06	95.46	116.25
吞吐率	1552.55	1450.54	1086.25	1047.56	860.22

表 6: efConstruction 对于正确率和时间性能的影响

和预测的结果基本相符，这里可以看出 M_max 和通过率整体呈正相关。而时间性能则和 M_max 呈现负相关关系。且 efConstruction 的提高对于两者的影响相当显著，因此需要考虑进行权衡，平衡时间性能和正确率的关系。

m_L	2	3	6	8	10
通过率	70.45%	71.18%	71.27%	71.73%	71.64%
平均每次查找用例耗时（微秒）	65.28	77.43	92.06	100.01	101.75
吞吐率	1531.86	1291.49	1086.25	999.90	982.80

表 7: m_L 对于正确率和时间性能的影响

m_L 对于通过率的影响较小，而与时间性能呈现负相关；这里和预测存在冲突，这可能是由于测试集大小较小造成的。测试集较小时，过多的分层会造成很大的额外时间开销。

3 结论

HNSW 算法可以在有限牺牲正确率的情况下极大优化查找的时间性能。但是这种优化带来的优势更多体现在规模很大的数据集上。如果数据集规模较小，则 HNSW 算法本身维护数据结构带来的时间开销会极大削弱算法本身的时间性能优势。

在实际使用中，应该综合考虑使用两种算法，依据数据集本身的性质进行选择。

多个参数均会对于 HNSW 算法的效果产生影响。总体上来说，M，M_max，efConstruction 和正确率呈现正相关关系，和时间性能呈现负相关关系。因此，在实际使用中需要先对于数据进行分析，再进行实验性的调试，选择较为合适的参数，平衡正确性和时间性能。m_L 的选取则需要考虑数据集的大小，在数据集规模较小时尽可能避免较多的分层，以免带来过大的时间损耗。

4 致谢

在完成此实验过程中，ChatGPT 和 deepseek 等大模型给了我很多帮助。同时，知乎专栏的介绍文章也给予我了理论原理上的指导。

5 其他和建议

在实验过程中，我遇到了一些挑战和困难：

1. 不同数据集的通过率差异很大，因此需要根据实际实验找到相对合理的参数设置范围。

2. 设备本身的性能一般，测试时花费的时间更多。

3. 由于 level 产生的随机性，不同次测试直接的正确率和时间性能存在差异，应该多次测试，得到更合理的数据验证规律。

4. 新版本的 E2E_test 不能用来测试老版本的数据，测试结果受测试文件和数据集的影响比较大。

建议：

1. 同学和助教共同完善答疑解惑的微信文档，互帮互助解决在环境配置等方面的问题。

2. 习题课对已经完成的 lab 或 project 进行讲解，加深理解。