Smart-LSM-Tree 的性能评估和并行化报告

523031910852 朱子墨

2025 年 5 月 20 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。

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。

在前面的阶段中，我使用了 HNSW（Hierarchical Navigable Small World）算法来提高搜索效率。然而，对于项目的各个接口依然缺乏系统的性能测试。本阶段的目的在于寻找算法中的时间性能瓶颈，利用并行化思想进行优化，从而优化接口的时间性能。

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：

项目	参数
内存大小	13.1GB
硬盘大小	50.0GB
处理器数量	8
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

考虑到本阶段并行化和测试性能的需求，我增加了虚拟机的处理器数量。

为了寻找到各个阶段、主要接口的性能瓶颈，我对于 put，search_knn，search_knn_hnsw 等接口进行了分步测试性能。设置参数 M = 14,M_max = 18,m_L = 6,efConstruction = 70。

2.2 测试结果与分析

考虑到前阶段中 get 操作的实现，较难实现并行化；而 del 操作经过测试，发现主要的时间开销来源于 put 接口的调用。del 操作实际可以看成 put 和 get 的结合，因此着重考虑 put 操作的性能优化。

2.2.1 put 接口的性能测试

对于 put 接口分为以下阶段测试：插入内存和 compact 阶段，插入 vectorMap 阶段，插入 hnsw 结构阶段。经过测试，发现相比较于 hnsw 结构的插入，前两者的时间消耗接近可以忽略不计（hnsw 的插入和数据规模有关，但是大约花费几百毫秒，而前两者的时间消耗不足 1 毫秒）。

因此，我对于 hnsw 的节点插入接口 insertNode 进行了性能测试，分为五个阶段：线性查找节点向量验证当前插入的键是否存在；创建节点；从顶层向下搜索入口；各层进行节点连接；更新入口点和系统状态。通过测试发现，单次操作时间开销如下：

阶段	平均消耗时间
线性查找节点向量验证当前插入的键是否存在	时间复杂度为 $O(n)$，但时间消耗较小，$n = 10000$ 时约为 0.1ms
创建节点	接近 0ms
从顶层向下搜寻入口	接近 0ms
各层进行节点连接	与数据规模正相关，理论时间复杂度为 $O(M \cdot m_L \cdot \log(n))$，$n = 10000$ 时约为 120ms
更新入口点和系统状态	接近 0ms

表 2: insertNode 接口的时间性能测量

显然，插入操作的开销主要来源于对于各层进行连接。在原有代码中，使用了一个简单的的循环结构，从节点最高层向最底层依次建立层内的节点连接。这个过程中不同层级之间是没有数据竞争发生的，因此可以较为简便地实现并行化过程。

2.2.2 search_knn 和 search_knn_hnsw 接口的性能测试

在 search_knn 和 search_knn_hnsw 接口内进行分步测试，并将二者进行不同测试集大小情况下的对比测试，每次查询 3 个最接近的值。可以得到以下数据：

阶段	平均消耗时间
embedding 向量化目标字符串	起伏较大，多次测量，一次向量化的耗时平均值约为 205ms
依次计算所有向量与目标值的相似度	与数据规模呈正相关，理论上时间复杂度为 $O(n)$ ，实际上 $n = 10000$ 时约为 27ms，$n = 30000$ 时约为 86ms
对向量依据相似度排序	与数据规模呈正相关，$n = 10000$ 时约为 1.8ms，$n = 30000$ 时约为 2.3ms
选前 n 个有效的值返回	平均约为 1ms 左右

表 3: search_knn 接口的时间性能测量

阶段	平均消耗时间
embedding 向量化目标字符串	起伏较大，多次测量，一次向量化的耗时平均值约为 205ms
初始化阶段	接近 0ms
从顶层向底层查找阶段	多次测量取平均，一次操作平均耗时 0.22ms 左右
最底层的扩展搜索阶段	多次测量取平均，一次操作平均耗时 0.26ms 左右
结果处理阶段	平均约为 0.05ms 左右

表 4: search_knn_hnsw 接口的时间性能测量

数据规模	算法选择	平均一次查询操作耗时（毫秒）
1000	search_knn	205.53
1000	search_knn_hnsw	209.82
5000	search_knn	211.64
5000	search_knn_hnsw	220.01
10000	search_knn	225.43
10000	search_knn_hnsw	214.08
20000	search_knn	267.12
20000	search_knn_hnsw	210.01
30000	search_knn	357.65
30000	search_knn_hnsw	231.65

表 5: search_knn 和 search_knn_hnsw 接口的时间性能对比

由此可以看出，当数据规模较小时，hnsw 算法不具有时间性能优势，这是因为此时 knn 算法中的线性遍历向量表计算对应的相似度的过程消耗的时间较小，远小于 embedding 向量化过程花费的时间；此时二者的时间差异可能是 embedding 本身的运行的不稳定性造成的。而当数

据规模很大（10000 个数据点及以上）时，knn 算法中的线性遍历向量表计算对应的相似度的过程会显著增大，其对于总时间的影响超过了 embedding 的不稳定带来的误差，此时可以相当明显地看出，和数据规模无明显直接关系的 hnsw 算法的时间性能的优越性。

考虑到单次的 embedding 使用基本无法实现并行化；而 hnsw 算法内部即使进行并行化，由于线程的管理需要额外的时间成本，因此对于 hnsw 算法内部，诸如最底层计算时的并行化并无太大意义。

2.3 对应的优化

经过综合考虑，我选择着眼于对 hnsw 节点插入过程的“向不同层次添加连接”进行并行化。这一过程，各层分别在一个线程中完成计算和连接，等待所有线程完成后再进行后续操作。

在参考代码中给出了一个线程池的实现。由于在大量节点的插入过程中会不断发生线程的创建和销毁，因此考虑使用线程池进行管理。在 hnsw 的构造函数中，构造一个 m_L 个线程（我这里设置了 6 个）的线程池。当进行节点各层的连接时，使用线程池分配线程，在每个线程里并行执行每一层的连接：分别选出候选节点，再进行相似度排序，然后保留 M 个最接近的节点，先后与其建立连接，修改邻接表。等待所有线程完成后执行下一步。

2.4 优化后的结果测试

对于优化前后的插入算法进行测试，测试结果如下：

数据规模	算法选择	平均一次插入操作耗时（毫秒）
1000	优化前	13.00
5000	优化前	34.80
10000	优化前	66.20
50000	优化前	315.60
1000	优化后	8.00
5000	优化后	22.40
10000	优化后	37.10
50000	优化后	188.28

表 6: hnsw 插入算法优化前后的时间性能对比

根据实验数据作出图像：

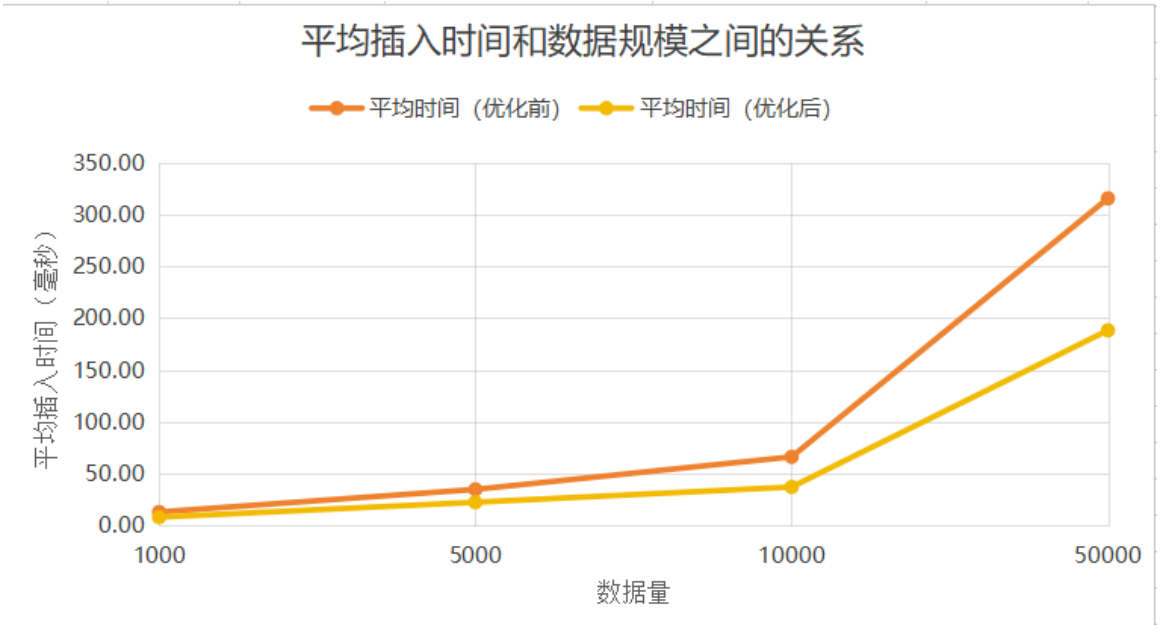

图 1: 平均查询时间和数据规模之间的关系

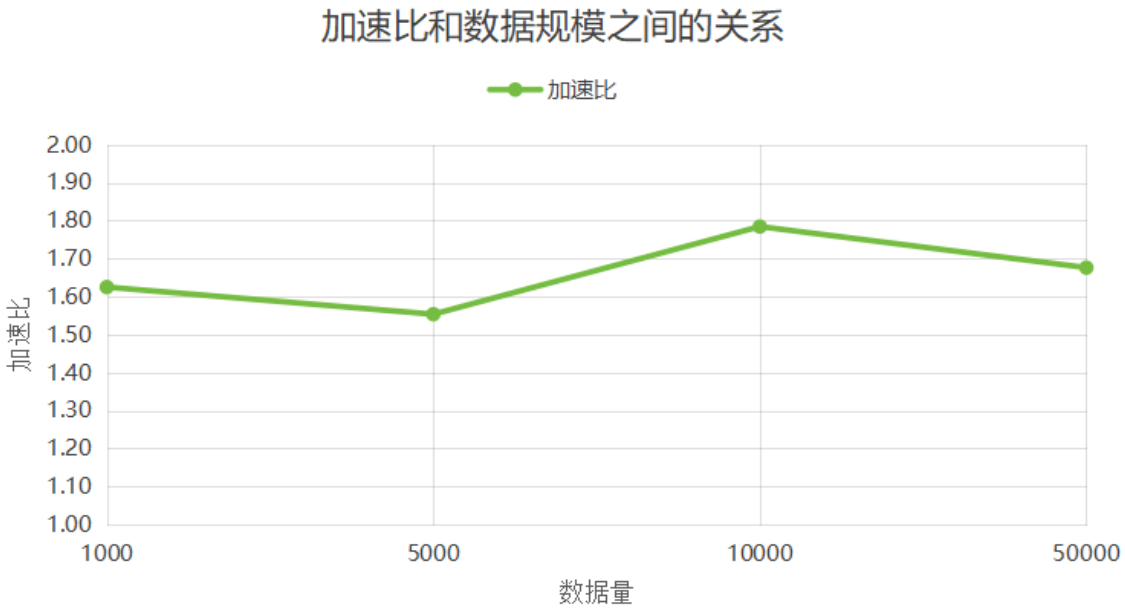

图 2: 加速比和数据规模之间的关系

可以看到，并行后的程序性能有了较大提升，加速比达到了 1.6 左右。然而，如果从理论分析，实际的加速比应该大约为 3 3.5 左右。这种差距一是由于线程的管理带来的时间损耗，二是由于线程的任务量不完全均匀，实际执行时间是执行时间最长的线程决定的。此外，虽然各层的连接占据了 put 操作的大部分时间，但是其他操作依旧会有一定比例的时间消耗，这些因素使得实际的加速比低于理论分析值。

3 结论

在本阶段，通过分步测试查找到了 put 操作的性能瓶颈，再通过分析找到了便于实现并行化优化时间性能的部分。在并行化时，需要考虑线程的创建和管理、线程池的使用，综合考虑提高程序的时间性能。同时，应当优先选择耗费时间多的部分进行并行化处理。

完成并行化后进行重复测试，降低随机性带来的评测误差，分析理论加速比和实际加速比的差距，以及造成差距的原因。

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很多指导和帮助。

5 其他和建议

新给出的 embedding 头文件无法关闭调试信息，在测试时会输出大量不必要的信息，这会带来一些麻烦。