LSM-Tree Phase5 报告

2025 年 5 月 18 日

1 背景介绍

本阶段引入了并行算法，对 LSM-tree 的各个部分进行加速。具体来说，本阶段并行化了跳表的 delete 和 insert 方法、sstable 相关的 compaction、向量持久化部分。

优化之后，本阶段测试了并行化带来的性能提升。

2 并行算法设计

考虑到增删改查等用户操作不会同步进行，所以将全部线程分配给 KVstore 类，在有需要时，由 KVstore 下发给其它类。

2.1 跳表部分

在插入和删除时，需要在每一层修改节点前后邻居的指针。这一操作可以并行化完成。

2.2 sstable 部分

本实验对于 sstable 的加载部分和 compatcion 函数进行了并行化处理。

加载部分需要顺序地读取多个 key 和 offset。将总文件分成若干块，并行读取每个块的数据，保证块的数据是有序的。此后按照块的顺序，对所有数据进行排序。compaction 函数中，需要遍历所有相关的 table，整理出需要合并的数据，这一过程可以并行化处理。

2.3 向量持久化

本实验为向量的加载函数提供了并行化处理。此外，还并行化了两个工具函数 get 和 isVecDelete，分别用于找到某个键的对应向量、判断某个向量是否被删除。search_knn 函数不适合直接利用并行化提高效率（如果并行化这个函数，需要维护多个哈希表），但是优化这两个函数可以起到间接作用。

2.4 HNSW 部分

HNSW 部分中，并行化了持久化相关的两个函数。对多个 node 的读取/写入可以同时进行，但读取时要注意顺序。

3 测试

3.1 实验设置

代码运行在 Linux 虚拟机环境下，CPU 为 8 核。

实验进行了 10k 次的插入、删除、查找操作；进行了 10k 次 search_knn 和 search_hnsw_knn 查找；在存储数据数量 10k 的情况下进行了数次持久化测试。实验测试了总体的性能提升，以及每个模块优化前后的性能提升。

3.2 预期结果

在加入并行化后，所有操作耗时均减小；同时每个模块耗时、总体耗时也相应减少。

3.3 实验结果与分析

运行数次对应操作，优化前和优化后的平均耗时等如表 1。

操作名	优化前平均耗时（ms）	优化后平均耗时（ms）	提升比例
put	151459	142239	6.09%
get	34.576	30.639	11.39%
delete	1064989	917167	13.89%
search_knn	43708	44413	-
search_hnsw_knn	33370	33961	-
初始化（持久化读入）	7268.3	22283	-
持久化存储	476.86	335.10	29.7%

表 1: 优化前后各个操作性能对比

表 1中，持久化以外操作的平均耗时指单次操作的平均耗时；持久化相关的平均耗时指读/写单个数据的平均耗时。

可以看到，持久化读入带来了负面影响；并行化对 search_knn、search_hnsw_knn 两个函数基本没有影响；对其它部分带来一定提升。

由于操作中需要频繁地获取向量，而这一操作需要调用 embedding 函数或从大文件中读取，性能都较差，会显著地减弱总体性能提升效果。

某些并行化代码在修改前后的平均耗时等如表 2。

操作名	优化前平均耗时（ms）	优化后平均耗时（ms）	提升比例
跳表插入	121.76	3036.0	-
跳表删除	2550.3	2894.6	-
compaction	205369	169099	17.66%

表 2: 优化部分性能对比

在更具体的测试中，跳表和持久化读入的并行化起到了负面效果。跳表本身就是高效的结构，而并行化增加了线程创建、销毁、调用等等的开销。持久化读入时，为了维持结果的顺序需要额外处理，这部分带来的开销超过了读取文件时节省的开销。

总体来看，增删改查的优化来源于 compaction 函数和维护向量的相关函数。在分块并行化时，只有最后合并的操作足够简单，并行化才能带来优化的效果。

4 结论

对于大量数据和需要高效的系统，并行化可以带来一定性能提升，但需要注意线程安全、防止死锁、注意语句执行顺序等等。

此外，lsm-tree 原本的定义是“高效的键值对存储系统”，但在添加了语义搜索等功能后，需要计算向量，大大降低了其基本的、增删改查功能的效率。在实践中，或许更合理的方式是将这两个系统分离，保留原有 lsm-tree 的高效特点。

5 致谢

感谢知乎、维基百科等博客、网站提供的参考；感谢 deepseek、kimi 等大模型提供的思路与帮助。

感谢提供支持的朋友们。

6 其他和建议