基于 LSM TREE 的智能键值存储项目报告
(Phase 2: Smart LSM)

5230309900223

2025 年 4 月 12 日

1 背景介绍

在电商推荐系统等典型场景中，商品特征常以高维向量形式存在，用于捕捉物品间的语义相似性。传统的基于 LSM-Tree (Log-Structured Merge Tree) 的键值存储系统，虽然通过顺序写优化和层级合并机制在高性能存储领域表现出色，但其核心功能仅限于精确的键值查询。面对诸如 “查找与‘水果’语义相似的商品” 这类基于内容相似性的查询时，传统 LSM-Tree 存在局限，因为它无法直接处理和比较值的语义信息。

为了克服这一限制，本项目在 Phase 1 实现的 LSM-Tree 基础上，进入 Phase 2 阶段，目标是构建一个 `Smart LSM-tree`。通过集成语义嵌入模型 (Embedding Model) 和向量相似度计算（具体为余弦相似度），赋予 LSM-Tree 存储引擎进行近似最近邻 (Approximate Nearest Neighbor, ANN) 搜索或在此简化场景下的 K 最近邻 (K-Nearest Neighbor, KNN) 语义搜索能力。这使得系统不仅能存储键值对，还能理解和检索语义上相似的值。

2 实现细节

为了实现 `search_knn` 功能，我对原有的 LSM-Tree 进行了扩展和修改：

- **`search_knn` 接口实现**: 在 `KVStore` 类中添加了 `search_knn(std::string query, int k)` 方法。该方法接受一个查询字符串 `query` 和一个整数 `k` 作为参数，返回类型为 `std::vector<std::pair<std::uint64_t, std::string>>`。

- **嵌入向量计算与缓存**:
  - 使用 `embedding` 模块提供的 `embedding_single` 函数计算查询字符串 `query` 的嵌入向量。
  - 为了避免每次搜索都重新计算所有存储值的向量，在 `KVStore` 类中增加了一个内存缓存。该缓存是一个 `std::map`,其键为 `uint64_t` 类型，值为 `std::vector<float>` 类型，变量名为 `embeddings`。它用于存储键 `key` 到其对应值 `value` 的嵌入向量的映射。
  - 在 `KVStore::put` 操作时，当插入或更新一个键值对后，会调用 `embedding_single` 计算新值的嵌入向量，并将其存入或更新到 `embeddings` 缓存中。
  - 在 `KVStore::del` 操作时，会尝试从 `embeddings` 缓存中移除对应键的向量。
  - 在 `KVStore::reset` 操作时，会清空 `embeddings` 缓存。
- **相似度计算与遍历**:
  - 实现（或使用 `embedding` 模块提供的）`cosine_similarity` 函数，用于计算两个嵌入向量之间的余弦相似度，公式如下：

$$\text{cosine\_similarity}(A, B) = \frac{A \cdot B}{\|A\| \cdot \|B\|}$$

  - 在 `search_knn` 中，需要遍历当前存储中的所有有效键值对（包括 MemTable 和 SSTable 中的）。对于每个键 `key`：
    - 检查其向量是否已存在于 `embeddings` 缓存中。
    - 如果向量不存在（例如，来自 SSTable 且未被缓存），则需要先通过 `get(key)` 或类似方式获取其字符串值 `value`。
    - **（优化点）为了提高效率，将所有来自 SSTable 且需要计算向量的值收集起来，调用 `embedding_batch` 进行批量计算，然后将结果存入缓存**。
    - 计算该向量与查询向量 `query_embedding` 的余弦相似度。
  - 将计算得到的（`key, similarity`）存储起来。
- **结果排序与返回**:

  - 收集完所有有效键值对的相似度后，根据相似度得分进行降序排序。如果相似度相同，则按键 `key` 升序排序以确保结果稳定。
  - 选取排序后的前 `k` 个结果。
  - 对于选中的每个 `key`，再次调用 `get(key)` 获取其最新的字符串值 `value`（以处理可能的并发更新或确保获取的是未被删除的值）。
  - 将最终的 `k` 个 `(key, value)` 对作为 `std::vector<std::pair<std::uint64_t, std::string>>` 返回。

- **性能计时**:
  - 为了满足报告中的性能分析要求，在 `search_knn` 函数内部，使用 `std::chrono` 库对关键步骤进行了计时，包括：查询向量计算、Memtable 相关处理（遍历、检查/计算向量、计算相似度）、SSTable 相关处理（遍历、检查/计算向量、计算相似度）、结果排序、结果获取（调用 `get`）以及总耗时。
  - 这些计时结果在每次 `search_knn` 调用结束时输出到控制台。

3 测试与性能分析

3.1 测试环境

- 操作系统：Windows 11 64 位
- CPU：[Intel(R) Core(TM) i9-13900H]
- 内存：32GB DDR4
- 编译器/构建系统：通过 CMake 和 MinGW 编译 C++20 代码。
- 嵌入模型: `./model/nomic-embed-text-v1.5.Q8_0.gguf` (通过 llama.cpp 加载)

3.2 测试方法

本次性能分析主要基于运行项目提供的 `E2E_Test` (`./build/test/E2E_Test.exe`)。该测试执行以下步骤：

1. 向 KVStore 中插入 `trimmed_text.txt` 文件中的前 120 行文本数据（作为值），键为 0 到 119。
2. 对插入的数据进行基本的 `put` 操作验证 (对应输出中的 Phase 1)。
3. 读取 `test_text.txt` 文件中的查询语句。
4. 对每个查询语句，调用 `search_knn(query, k=3)`
5. `search_knn` 函数内部自动记录并输出各子任务的耗时。
6. 将 `search_knn` 返回的结果与 `test_text_ans.txt` 中的预期答案进行比较，并在所有查询结束后报告 Phase 2 的通过率（考虑 15% 容错）。

因此，性能分析的数据直接来源于 `E2E_Test` 运行时打印的 `KNN Search Performance Analysis:` 信息块。

3.3 预期结果分析 (KNN 搜索)

- **向量计算是瓶颈**: 预期 `Query vector computation time`（包括查询向量和可能的值向量计算）将是整个 `search_knn` 操作中最耗时的部分，因为它涉及调用底层语言模型 (llama.cpp) 进行推理。其耗时应远超其他部分。
- **相似度计算开销**: 遍历所有 120 个存储项并计算余弦相似度的过程，虽然涉及向量运算，但相比模型推理，预期耗时较短。这部分耗时可能包含在 `Memtable search time` 和 `SSTable search time` 的计时中（因为需要遍历并访问向量）。
- **内存与磁盘访问**: `Memtable search time` 和 `SSTable search time` 在这里的含义与纯粹的 KV 查找不同。它们更多地反映了遍历数据结构、检查向量缓存、可能触发的批量向量计算以及计算相似度的总时间。由于数据量不大 (120 项)，且向量缓存在内存中，预期这部分时间主要消耗在 CPU 计算上，而不是磁盘 I/O。
- **排序与检索**: `Result sorting time` (对 k 个结果或所有相似度排序) 和 `Result retrieval time` (调用 `get` 获取最终 value) 预期耗时非常短，因为涉及的数据量很小。

3.4 实验结果与分析 (KNN 搜索)

`E2E_Test` 成功运行并通过，输出了详细的性能计时数据。以下是从多次 `KNN Search Performance Analysis` 输出中提取的 ** 平均耗时 **:

计时模块	平均耗时 (ms)
Query vector computation time	258.750
Memtable search time	0.151
SSTable search time	0.000
Result sorting time	0.004
Result retrieval time	0.002
Total time	**258.907**

表 1: KNN 搜索各模块平均耗时分析 (N=117 次查询)

结果分析:

- **向量计算主导耗时**: 实验结果清晰地表明，`Query vector computation time` 占据了整个 `search_knn` 操作总耗时的绝大部分（平均约 136 ms，占总耗时 >99%）。这验证了预期，即调用嵌入模型进行推理是主要的性能瓶颈。
- **LSM-Tree 相关操作极快**: `Memtable search time` 和 `SSTable search time` 的耗时非常低（亚毫秒级）。这表明在当前数据规模下，遍历内存中的 SkipList 或从 SSTable 获取少量元数据（如果需要触发批量向量计算）以及计算相似度的开销非常小。内存向量缓存 `embeddings` 起到了关键作用，避免了频繁的模型调用。
- **后处理开销可忽略**: `Result sorting time` 和 `Result retrieval time` 的耗时同样非常低，符合预期。对少量结果进行排序和最终的 `get` 操作几乎不影响整体性能。
- **整体性能**: `search_knn` 的总耗时主要取决于单次或批次的向量计算时间。

4 结论

本项目成功地在 Phase 1 实现的 LSM-Tree 基础上，扩展实现了 Phase 2 的 `search_knn` 功能，赋予了键值存储系统进行语义相似性搜索的能力。

- 通过集成 `embedding` 模块和实现 `cosine_similarity` 计算，系统能够理解查询和存储值的语义信息。
- 利用内存中的 `embeddings` map 对计算出的向量进行缓存，有效避免了每次查询都重复计算所有值的向量，显著提高了效率（尽管向量计算本身仍是瓶颈）。
- 对 `put`, `del`, `reset` 操作进行了相应修改以维护向量缓存的一致性。
- `E2E_Test` 的通过（Phase 1: 120/120 [PASS], Phase 2: 313/360 [PASS]）验证了 `search_knn` 功能的正确性（在容错范围内）以及基础 KV 操作的稳定性。
- 性能分析明确指出了 ** 向量计算是 `search_knn` 操作的主要性能瓶颈 **，其耗时远超 LSM-Tree 本身的查找、排序和数据检索开销。

5 致谢

在完成此实验过程中，`llama.cpp` 开源项目及其文档为嵌入模型的集成提供了基础。RocksDB 和 LevelDB 的设计思想对理解 LSM-Tree 的实现细节有很大帮助。此外，和同学的交流探讨和 AI 工具在解决具体编码问题和提供思路方面给予了支持。

6 其他和建议

在实验过程中，我遇到了一些挑战和困难：

- 理解和集成 `llama.cpp` 的嵌入接口需要仔细阅读其示例和 API 文档。
- 在 `search_knn` 中设计有效的向量缓存处理逻辑（何时计算、何时更新）需要一些思考。

- 调试 `CMakeLists.txt` 中不同目标（`correctness`, `persistence`, `kvstore` library, `E2E_Test`）的编译定义和链接关系，以确保嵌入逻辑只在需要时启用，花费了一些时间。
- 最初 `correctness` 和 `persistence` 测试因包含向量计算而运行极其缓慢，通过使用预处理器宏跳过这些计算，才得以快速验证基础功能。

建议：

- （已完成）提供一种方式在运行基础功能测试（`correctness`, `persistence`）时禁用耗时的嵌入计算。
- 考虑向量的持久化方案，使得重启后无需重新计算所有向量。