代码设计

1.项目概述

1.1 实现字段查询功能

LevelDB 的基本数据结构是由一个 key 和对应的 value 组成，其中 value 是一个简单的字节序列（可以是字符串或二进制数据）。默认情况下，LevelDB 不支持像关系型数据库那样的字段查询功能。然而，在实际应用中，用户可能需要对存储的数据进行更加精细的操作，特别是当值包含多个逻辑字段时，直接使用现有的 LevelDB 接口难以满足需求。
在本实验中，我们的目标是扩展 LevelDB 的功能，使其 value 支持多字段结构，并实现通过字段值查询对应的 key 的功能。

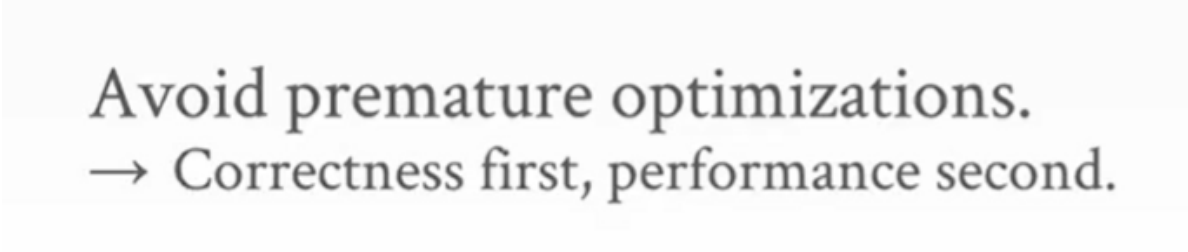
1.2 KV 分离

在 LevelDB 及其采用的 LSM 树结构中，性能挑战之一在于 Compaction 操作的效率。Compaction 是指将内存中的数据合并到磁盘上的过程，此过程中涉及大量的读写操作，对于系统的整体性能有着重要影响。在 Compaction 时，所有涉及到的旧 sstable 中的键值对都将被写入到新 sstable 中，而 Value 通常比 Key 大得多。如果将 Key 和 Value 分离存储，合并时只涉及 key 写入 sstable 的过程，可以显著减少 Compaction 的开销，从而提升性能。
基于此我们计划实施键值分离策略。具体而言，键将保持原有的排序方式，而值将被独立存储。这样做可以在不影响查询性能的前提下，大幅降低 Compaction 过程中的数据迁移量，进而减少不必要的磁盘 I/O ，提升系统的合并效率。

2.功能设计

Andy Pavlo在15445课程中说，完成一个项目，应先写出能够完成正确性要求的代码，再在此基础上提升性能，避免不成熟的优化方式。

因此，我们的项目流程将保持每周推进代码进度，在完成目标要求的代码的基础上，不断迭代优化性能。



2.1.字段设计

- **设计目标**:
    - 将 LevelDB 中的 `value` 组织成字段数组，每个数组元素对应一个字段（字段名：字段值）。
    - 字段会被序列化为字符串，然后插入LevelDB。
    - 这些字段可以通过解析字符串得到，字段名与字段值都是字符串类型。
    - 允许任意调整字段。

- 实现通过字段值查询对应的 `key`。

- **实现思路**:

函数 `Put_with_fields` 负责插入含字段的数据。原字段数据经过序列化函数 `SerializeValue` 处理后，函数 `Put_with_fields` 调用 `Put` 将序列化后的字段插入 leveldb。

函数 `Get_with_fields` 负责获得含字段的数据。使用 `Get` 从 leveldb 中获取 `key` 和序列化后的 `value`，调用 `ParseValue` 可以将字段反序列化。

函数 `Get_keys_by_field` 遍历数据库中的所有键值对，解析每个 `Value`，提取字段数组 `FieldArray`。检查字段数组中是否存在目标字段，如果匹配，则记录其对应的 `Key`。将所有匹配 `key` 汇总到 `keys` 中返回。

初步实现（第一周 已完成）：在 leveldb 内部实现以上功能。内部实现会导致读取时无法区分多字段类型和原生 kv 对，扩展性不足。

后续改进（第二周）：为了解决无法区分多字段类型和原生 kv 对的问题，将以上函数功能实现在用户层级，使 leveldb 内部对多字段类型无感知。

2.2.KV分离

- **设计目标**:
  - **KV 分离设计**

    a. 将LevelDB的key-value存储结构进行扩展，分离存储key和value

    b. Key存储在一个LevelDB实例中，LSM-tree中的value为一个指向Value log文件和偏移地址的指针，用户Value存储在Value log中。
  - **读取操作。**

    KV分离后依然支持点查询与范围查询操作。
  - **Value log的管理。**

    a.当Value log超过一定大小后通过后台GC操作释放Value log中的无效数据。

    b.GC能把旧Value log中没有失效的数据写入新的Value log，并更新LSM-tree里的键值对。

    c.新旧Value log的管理功能。
  - **确保操作的原子性**

- **实现思路**:

初步实现（第一周 已完成）：使用单一Value Log简单的实现KV分离，该实现较为简单，仅需在Put/Get函数内部进行简单修改，但在大数据量场景下性能极差。

优点：实现简单，合并时开销小

缺点：大数据量下性能极差，不能作为最终方案。

第二种实现（第二周）：对每个SSTable和MemTable建立一个Value Log。该实现相比于初步实现更加复杂，需要在合并时查询所有相关Value Log，并建立新Value Log。此外还要考虑在合并结束后将废弃的Value Log异步删除。

优点：随合并自动GC，无需考虑GC。查询和插入性能同第一种实现有较大改善。

缺点：合并时开销未能减小。

第三种实现（第三周）：使用相对固定大小的Value Log，例如每个Value Log大小约为2KB。新添加的键值对依次将值计入最新Value Log，当Value Log大小满了之后就创建新Value Log。需要设计一种不改变SSTable内记录Value元数据的GC方法。

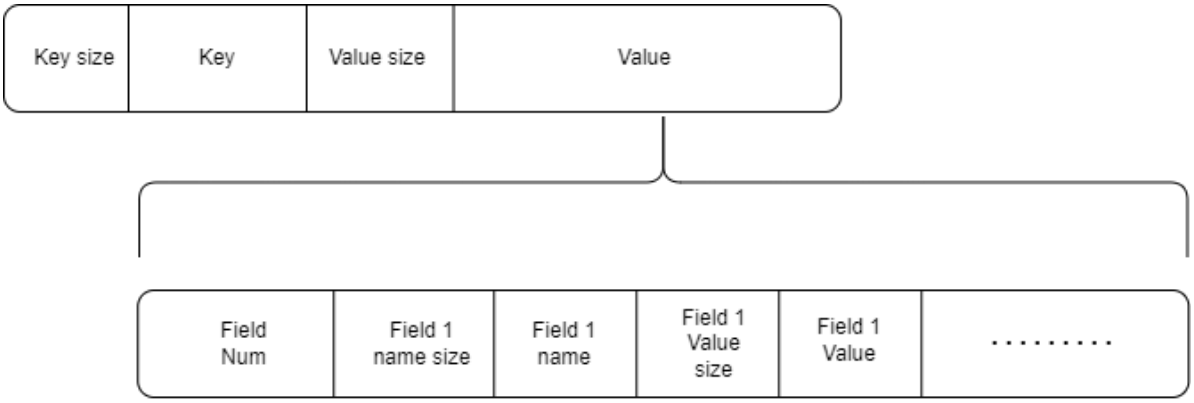
优点：合并时开销小。

缺点：需要设计一种GC方式，能够在异步GC的同时不改变SSTable。

3. 数据结构设计

KV分离后 Value 结构设计

一个Value，开头是使用Varint64存储的FieldNum，表示有FieldNum个Field组成。然后是使用Varint64存储的Field X name size，表示该field的字段名长度，然后是字段名，然后是使用Varint64存储的Field X Value size，表示该field的值长度，然后是值。

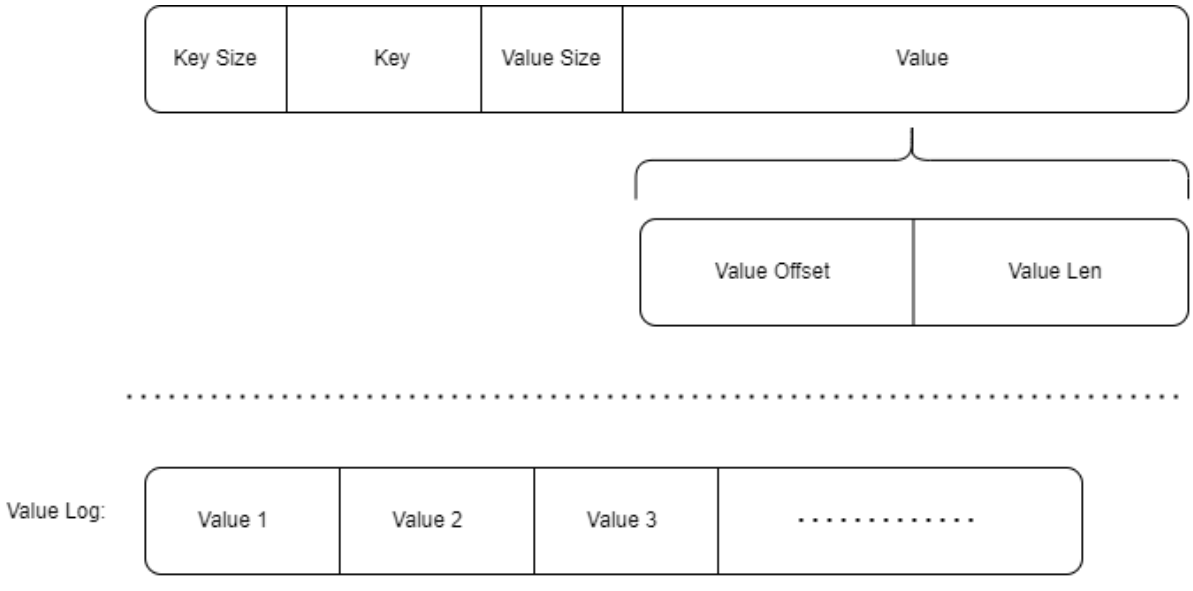


ValueLog结构设计

第一版设计

使用一个Value Log文件的设计中，我们只需记录Value在Value Log中对应的偏移量和Value长度即可。

Value Log中只记录Value值，无需记录元信息。

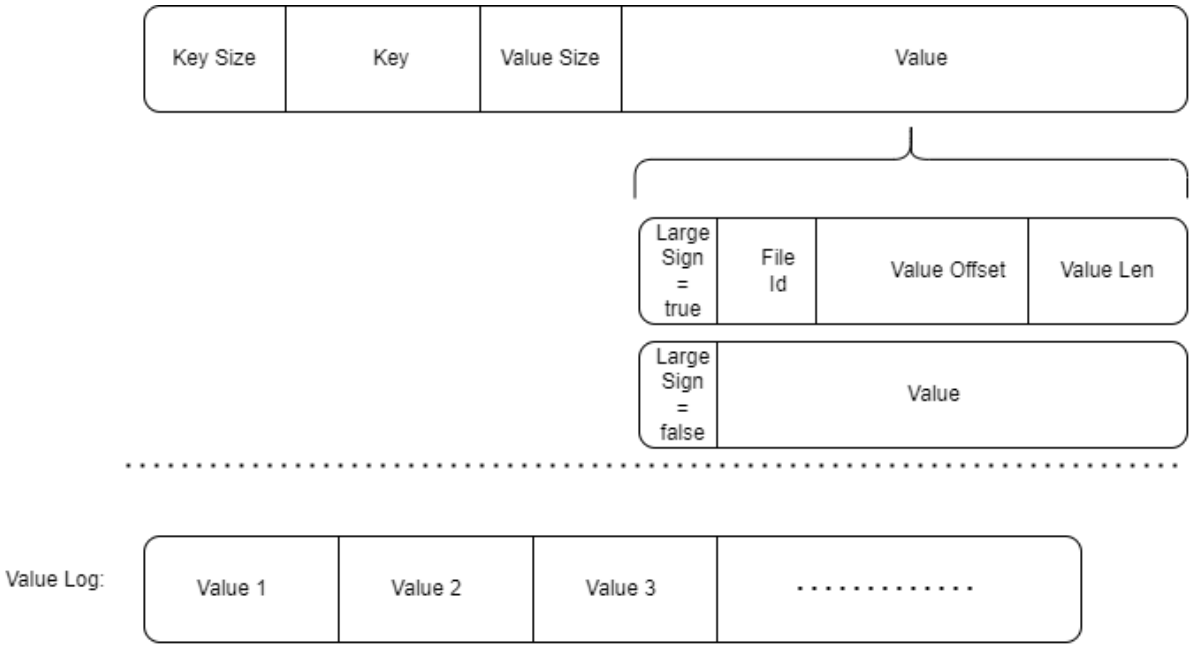


第二版设计

Value设计为：1字节标志位+Varint64文件ID+Varint64偏移量+Varint64长度。

在存储时根据Value大小是否较大选择进行KV分离。若分离则标志位为true，否则标志位为false。

日志文件中仍然只需记录Value值即可，无需记录元信息。



第三版设计

这一版设计有些复杂，但已经完成。

我们将在后续对该设计的详细信息进行公开。

粗略来讲，第三版设计的每一个Value Log将对应固定数量的键值对或大约固定大小的键值对的值。并且拥有一个异步的GC函数，且该GC函数不会影响SSTable内的数据。

4. 接口/函数设计

4. 1Value多字段设计

4.1.1 数据序列化与反序列化

序列化字段数组为字符串值

```
std::string DB::SerializeValue(const FieldArray& fields);
```

- **输入**：字段数组 `fields`。
- **输出**：序列化后的字符串。

反序列化字符串值为字段数组

```
void DB::ParseValue(const std::string& value_str, FieldArray* res);
```

- **输入**：序列化字符串 `value_str`。
- **输出**：字段数组 `res`。

4.1.2 数据存储接口

存储字段数组

```
Status DB::Put_with_fields(const WriteOptions& op, const Slice& key, const FieldArray& fields);
```

- **输入**:
  - 写入选项 `op`。
  - 键值 `key`。
  - 字段数组 `fields`。
- **输出**: 操作状态 `Status`。

读取字段数组

```
Status DB::Get_with_fields(const ReadOptions& options, const Slice& key, FieldArray* fields);
```

- **输入**:
  - 读取选项 `options`。
  - 键值 `key`。
- **输出**:
  - 操作状态 `Status`。
  - 字段数组 `fields`。

4.1.3 数据查询接口

按字段查找键

```
Status DB::Get_keys_by_field(const ReadOptions& options, const Field field, std::vector<std::string>* keys);
```

- **输入**:
  - 读取选项 `options`。
  - 字段值 `field`。
- **输出**:
  - 操作状态 `Status`。
  - 符合条件的键列表 `keys`。

4.2 Value Log设计

4.2.1 WriteValueLog

将一堆键值对的值顺序写入Value Log，用于writebatch写入数据库，以及Value Log GC的时候。两者都会对多个键值对同时操作，因此设计为批处理。

函数内将使用写锁保证正确性。同一时间最多只有一个WriteValueLog可以进行。

```
std::vector<std::pair<uint64_t,uint64_t>> WriteValueLog(std::vector<const slice&> value);
```

- **输入**：一个Slice vector，表示要写入Value Log的Value们。
- **输出**：一个std::pair<uint64_t,uint64_t> vector，每个pair中：第一个uint64_t是Value Log文件ID，第二个uint64_t是处在Value Log中的偏移量。

4.2.2 ReadValueLog

通过Value Log读取目标键值对的值。

函数内将使用读锁保证正确性。在一个ValueLog正在被读取时，GC和WriteValueLog(?)无法对该ValueLog操作。

```
Status ReadValueLog(uint64_t file_id, uint64_t offset,Slice* value);
```

- **输入**：第一个uint64_t是Value Log文件ID，第二个uint64_t是处在Value Log中的偏移量，第三个是指向要传回的value的指针。
- **输出**：一个Status，表示是否成功传回对应Value。

4.2.3 从ValueLogGC

第三种KV分离方案独有的Value Log异步垃圾回收函数，当一个ValueLog中有效Value较少时将被激活，对该Value Log进行GC。

```
Status ValueLogGC(uint64_t file_id);
```

- **输入**：一个uint64_t，表示Value Log文件ID。
- **输出**：一个Status，表示是否成功GC。

---

5. 功能测试

5.1单元测试（测试用例）：

依据我们的设计，每周的工作内容完成后，都将对当前完成的功能进行正确性检验。以下以第一周我们完成的功能为例：

第一周

字段数组的存储与读取：

验证了 `Put_with_fields` 和 `Get_with_fields` 的正确性，确保字段数组可以正确序列化存储并反序列化读取。

基于字段的键查询：

验证了 `Get_keys_by_field` 的逻辑，确保能够根据字段值查找所有匹配的键。

Key Value分离：

并未额外设计，通过上两个功能的正确运行能够证明Key Value分离的初步实现大体是正确的。

```
#include "gtest/gtest.h"
#include "leveldb/env.h"
#include "leveldb/db.h"
using namespace leveldb;

constexpr int value_size = 2048;
constexpr int data_size = 128 << 20;

Status OpenDB(std::string dbName, DB **db) {
  Options options;
  options.create_if_missing = true;
  return DB::Open(options, dbName, db);
}

TEST(TestTTL, OurTTL) {
    DB *db;
    WriteOptions writeOptions;
    ReadOptions readOptions;
    if(OpenDB("testdb_for_XOY", &db).ok() == false) {
        std::cerr << "open db failed" << std::endl;
        abort();
    }
    std::string key = "k_1";

    std::string key1 = "k_2";

    FieldArray fields = {
        {"name", "Customer#000000001"},
        {"address", "IVhzIApeRb"},
        {"phone", "25-989-741-2988"}
    };

    FieldArray fields1 = {
        {"name", "Customer#000000001"},
        {"address", "abc"},
        {"phone", "def"}
    };

    db->Put_with_fields(WriteOptions(), key, fields);

    db->Put_with_fields(WriteOptions(), key1, fields1);

    // 读取并反序列化
    FieldArray value_ret;
    db->Get_with_fields(ReadOptions(), key, &value_ret);;
    for(auto pr:value_ret){
        std::cout<<std::string(pr.first.data(),pr.first.size())<<" "
<<std::string(pr.second.data(),pr.second.size())<<"\n";
    }

```

```
    std::vector<std::string> v;
    db->Get_keys_by_field(ReadOptions(),fields[0],&v);
    for(auto s:v)std::cout<<s<<"\n";
    delete db;
}



int main(int argc, char** argv) {
  // All tests currently run with the same read-only file limits.
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

进一步设计

对KV分离实现更细粒度的测试，以及对KV分离GC操作实现测试

1.向表内插入一些value较小的键值对以及value较大的键值对，随后通过检查ValueLog内部数据（也可以是ValueLog文件长度）来判断是否对长短数据各自进行了处理。

2.向表内插入大量value较大的键值对后，查询ValueLog文件总数，删除其中绝大多数键值对，然后再查一次ValueLog文件总数，期望文件总数变少。

5.2性能测试（Benchmark）：

这一部分我们希望在完成大部分功能后再根据代码调整。

初步计划：

1.测试大数据量下短键值对和长键值对分别的插入和查询效率，与原版LevelDB作对比。

2.测试大数据量下磁盘使用率，与原版LevelDB作对比。

3.测试大数据量下合并的速率，与原版LevelDB作对比。

4.完成了多种KV分离方案后，将不同方案在Benchmark下进行测试。

6. 可能遇到的挑战与解决方案

如何处理**GC开销、数据同步**

如何实现GC

在数据结构设计中已经进行了详细说明。

第二种设计通过合并过程自动完成了GC的功能。

第三种设计通过设计了一种异步的GC操作，使GC无需改变SSTable数据。

数据同步

写Value Log的时机和写WAL的时机一致，都在写MemTable之前完成。如果用户的Sync参数设置为True，则要保证Value Log一定写入完成后才能返回给用户写入成功的信息。

减少GC开销

有一种可能的优化是仅在数据写入SSTable之后才会使用Value Log。

7. 分工和进度安排

功能	完成日期	分工
完成初步的多字段Value实现和KV分离实现	11月20日	谢瑞阳
完成设计文档	11月27日	徐翔宇&谢瑞阳
将多字段Value实现迁移至用户层级	11月27日	徐翔宇
完成第二版ValueLog的设计	11月27日	谢瑞阳
完成第二版ValueLog的测试	11月27日	徐翔宇
完成第三版ValueLog的函数接口实现以及测试	12月1日	徐翔宇
完成第三版ValueLog的函数实现	12月4日	谢瑞阳
完成BenchMark设计	12月8日	徐翔宇&谢瑞阳
完成BenchMark，对不同KV分离方案进行测试	12月11日	徐翔宇
基于测试结果进行优化，完成第四版ValueLog的设计...	12月??日	徐翔宇&谢瑞阳