项目文档

在 levelDB 中实现二级索引

小组成员：陈予曈、朱陈媛

1. 项目背景

1.1 项目概述

本实验旨在实现高效的数据检索功能，特别是针对特定字段的查询场景。首先，要求实现字段查询功能，以便灵活存储多字段的数据。

但在未使用索引的情况下，FindKeysByField 函数需要遍历整个数据集逐个判断字段是否满足条件，这种线性查找方法在数据量较大时效率较低。为了提升检索性能，本实验将实现二级索引的功能。

二级索引是一种通过对特定字段或属性建立索引结构的技术，使得查询操作能够快速定位目标数据，而无需遍历整个数据集。通过实现二级索引，可以显著提高特定字段的查询效率，从而优化整体系统的性能。

1.2 项目目标

本项目涵盖以下三个方面：

（1）在 LevelDB 的 value 中实现字段功能。

（2）实现二级索引。

（3）实现 Benchmark，测试并分析性能。

2. 多字段功能

2.1 多字段功能设计

2.1.1 设计目标

基于 levelDB，扩展 value 的结构，使其可以包含多个字段，并通过这些字段实现类似数据库列查询的功能。

（1）字段存储

将 LevelDB 中的 value 组织成字段数组，每个数组元素对应一个字段；字段会被序列化为字符串，然后插入 LevelDB；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。

（2）查询功能

通过字段实现类似数据库列查询的功能。

2.1.2 编码格式设计

（1）方案一：定长

分析：便于实现，但是由于字段任意可变，对于 address 这类长度变化很大的字段，很容易超出固定的长度范围，因此舍弃该方法。

（2）方案二：变长，并记录长度

节省空间且灵活，无需手动记录长度，可以使用 levelDB 内置的 PutLengthPrefixedSlice 函数和 GetLengthPrefixedSlice 函数。

（3）方案三：用特殊字符分隔

相对容易实现，但是需要处理字段值中出现分隔符的情况。value 需要将字段之间分隔开，同时每个字段内部，字段名和字段值还需要分隔。

编码格式如下：

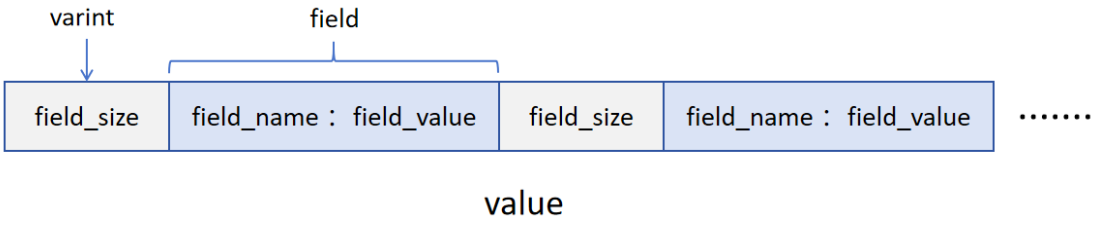

图 1 多字段的编码格式

2.2 多字段功能实现

最终，我们实现了方案二与方案三的两版设计，具体如下：

（1）方案二：变长，并记录长度

①定义类型别名（db/db.h）

```
using Field = std::pair<std::string, std::string>;
using FieldArray = std::vector<std::pair<std::string, std::string>>;
```

②序列化函数 SerializeValue（db/NewDB.cc）：

```
std::string NewDB::SerializeValue(const FieldArray& fields){
    std::string s;
    for(auto& field : fields){
      PutLengthPrefixedSlice(&s, Slice(field.first));
      PutLengthPrefixedSlice(&s, Slice(field.second));
    }
    return s;
}
```

③反序列化函数 ParseValue（db/ NewDB.cc）：

```
FieldArray NewDB::ParseValue(const std::string& value_str){
    Slice input(value_str);
    Slice v1,v2;
    FieldArray fields;
    while(!input.empty()){
      GetLengthPrefixedSlice(&input, &v1);
      GetLengthPrefixedSlice(&input, &v2);
```

```
        fields.push_back(std::make_pair(v1.ToString(), v2.ToString()));
      }
    return fields;
}
```

④FindKeysByField 函数

db/ NewDB.cc：

```
std::vector<std::string> NewDB::FindKeysByField(Field &field){
  //get keys with field
  std::vector<std::string> matching_keys;
  leveldb::DB* db = data_db_.get();
  matching_keys = data_db_->FindKeysByField(db, field);
  return matching_keys;
}
```

db/ impl.cc：

```
std::vector<std::string> DB::FindKeysByField(leveldb::DB* db, Field &field) {
 std::vector<std::string> matching_keys;
 leveldb::Iterator* it = db->NewIterator(leveldb::ReadOptions());
 for (it->SeekToFirst(); it->Valid(); it->Next()) {
  std::string key = it->key().ToString();
  std::string value = it->value().ToString();
  NewDB* db;
  FieldArray fields = db->ParseValue(value);
  for(const auto& each_field : fields){
   if(field.first == each_field.first){
    if(field.second == each_field.second){
     matching_keys.push_back(key);
    }
   }
  }
 }
 delete it;
 return matching_keys;
}
```

（2）方案三：用特殊字符分隔

①序列化函数 SerializeValue（db/NewDB.cc）：

```
std::string NewDB::SerializeValue(const FieldArray& fields){
  std::string value_str;
  for(const auto& pair : fields){
```

```
        std::string field = pair.first + ":" + pair.second;
        uint32_t field_size = field.size();
        char buffer[4];
        EncodeFixed32(buffer, field_size);
        value_str.append(buffer, 4);
        value_str.append(field);
    }
    return value_str;
}
```

②反序列化函数 ParseValue（db/NewDB.cc）：

```
FieldArray NewDB::ParseValue(const std::string& value_str){
    FieldArray fields;
    const char* data = value_str.data();
    size_t length = value_str.size();
    while (length >= 4) {
        uint32_t field_size = DecodeFixed32(data);
        if (length < 4 + field_size) {
        break;
        }
        std::string field(data + 4, field_size);
        size_t colon_pos = field.find(':');
        std::string field_name = field.substr(0, colon_pos);
        std::string field_value = field.substr(colon_pos + 1);
        fields.push_back(std::make_pair(field_name,field_value));
        data += 4 + field_size;
        length -= 4 + field_size;
    }
    return fields;
}
```

最终，出于对查询性能的考量，我们选择了**方案三：用特殊字符分隔**。具体的数据分析与对比请见 **4.性能测试**。

2.3 多字段功能测试

（1）基础测试

插入多字段数据，再读取，检测数据结构改变后是否能正常读写。

（2）FindKeysByField 函数功能测试

在测试中创建多个键值对,验证 FindKeysByField 能否正确地查找并返回匹配的键；使用 FindKeysByField 查找字段值，检查它是否能够找到匹配的键。

```
#include "gtest/gtest.h"

```

```
#include "leveldb/env.h"
#include "leveldb/db.h"
// 这个头文件是为了 EncodeFixed32
#include "util/coding.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);
}


//测试多字段
TEST(TestSchema, Basic) {
    DB *db;
    if(OpenDB("testdb", &db).ok() == false) {
      std::cerr << "open db failed" << std::endl;
      abort();
    }
    std::string key = "k_1";

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

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

    db->Get_fields(ReadOptions(), key, &fields);

    //清理数据库
    delete db;
}
// 测试 FindKeysByField 函数
 /*
 1.在测试中创建多个键值对,验证 FindKeysByField 能否正确地查找并返回匹配的键。
 2.使用 FindKeysByField 查找字段值，检查它是否能够找到匹配的键。
 */
```

```
47.     TEST(TestSchema, FindKeysByFieldTest) {
48.         DB *db;
49.         if (OpenDB("testdb", &db).ok() == false) {
50.             std::cerr << "open db failed" << std::endl;
51.             abort();
52.         }
53.
54.         // 插入多个键值对
55.         std::string key1 = "k_1";
56.         FieldArray fields1 = {
57.             {"name", "Customer#000000001"},
58.             {"address", "IVhzIApeRb"},
59.             {"phone", "25-989-741-2988"}
60.         };
61.         db->Put_fields(WriteOptions(), key1, fields1);
62.
63.         std::string key2 = "k_2";
64.         FieldArray fields2 = {
65.             {"name", "Customer#000000002"},
66.             {"address", "IVhzIApeRb"},
67.             {"phone", "25-123-456-7890"}
68.         };
69.         db->Put_fields(WriteOptions(), key2, fields2);
70.
71.         std::string key3 = "k_3";
72.         FieldArray fields3 = {
73.             {"name", "Customer#000000003"},
74.             {"address", "TXkjZEdIrZ"},
75.             {"phone", "25-555-888-1234"}
76.         };
77.         db->Put_fields(WriteOptions(), key3, fields3);
78.
79.         // 测试 FindKeysByField
80.         Field search_field = {"address", "IVhzIApeRb"};
81.         std::vector<std::string> matching_keys = db->FindKeysByField(db, search_field);
82.
83.         EXPECT_EQ(matching_keys.size(), 2);
84.         EXPECT_TRUE(std::find(matching_keys.begin(), matching_keys.end(), key1) != matching_keys.end
    ());
85.         EXPECT_TRUE(std::find(matching_keys.begin(), matching_keys.end(), key2) != matching_keys.end
    ());
86.
87.         // 清理数据库
88.         delete db;
```

```
89.     }
90.     int main(int argc, char** argv) {
91.       // All tests currently run with the same read-only file limits.
92.       testing::InitGoogleTest(&argc, argv);
93.       return RUN_ALL_TESTS();
94.     }
```

（3）测试结果

```
chen@chen-None:~/leveldb_project2/build$ /home/chen/leveldb_project2/build/field_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TestSchema
[ RUN      ] TestSchema.Basic
[       OK ] TestSchema.Basic (43 ms)
[ RUN      ] TestSchema.FindKeysByFieldTest
[       OK ] TestSchema.FindKeysByFieldTest (29 ms)
[----------] 2 tests from TestSchema (82 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (94 ms total)
[  PASSED  ] 2 tests.
chen@chen-None:~/leveldb_project2/build$
```

图 2 多字段测试结果

3. 二级索引

3.1 二级索引的总体设计

3.1.1 编码设计

（1）方案选择

索引字段的值可能出现重复，为了保证索引键的唯一性，将原本键值对中的 key 附加在索引键的后面，这样可以确保每个索引键只能对应到一条键值对。有以下两种设计方案：

①存储字段名

Key：FieldName_FieldValue_Key Value：null

这种方式适用于将不同的字段名存储在一个 LSM tree 的情况。

②不存储字段名

Key：FieldValue_Key Value：null

这种方式适用于将不同的字段名分别存储在不同 LSM tree 的情况。

如果将不同字段名分别存储在不同 LSM tree，可能出现大量不同字段名，此时就需要维护很多 LSM tree，而且每个 LSM tree 可能只包含很少的几条索引，这样做的性价比比较低。

因此考虑使用**第一种设计方案**。

（2）功能实现

类似于 **2.2 多字段功能实现**的编码设计，我们同样采取了两种实现方式：

方案二：变长，并记录长度

①构造索引键函数 ConstructIndexKey（db/ NewDB.cc）

```
// 构造索引的 key
std::string NewDB::ConstructIndexKey(const Slice& key, const Field& field){
    std::string s;
    PutLengthPrefixedSlice(&s, Slice(field.first));
    PutLengthPrefixedSlice(&s, Slice(field.second));
    PutLengthPrefixedSlice(&s, key);
    return s;
}
```

②解析索引键函数 ExtractIndexKey（db/ NewDB.cc）

```
// 从索引键中提取原数据的键
std::string NewDB::ExtractIndexKey(const Slice& key){
    Slice input = key;
    Slice v;
    GetLengthPrefixedSlice(&input, &v);
    GetLengthPrefixedSlice(&input, &v);
    GetLengthPrefixedSlice(&input, &v);
    return v.ToString();
}
```

方案三：用特殊字符分隔

①构造索引键函数 ConstructIndexKey（db/ NewDB.cc）

```
std::string NewDB::ConstructIndexKey(const Slice& key, const Field& field){
    std::ostringstream oss;
    oss << field.first << ":" << field.second << "_" << key.ToString();
    return oss.str();
}
```

②解析索引键函数 ExtractIndexKey（db/ NewDB.cc）

```
std::string NewDB::ExtractIndexKey(const Slice& key){
    //extract key of dataDB from key of indexDB
    std::string fullKey(key.data(), key.size());
    size_t pos1 = fullKey.find('_');
    return fullKey.substr(pos1 + 1);  // 提取 'Key' 部分
}
```

最终，出于对查询性能的考量，我们选择了**方案三：用特殊字符分隔**。具体的数据分析与对比请见 **4.性能测试**。

3.1.2 数据结构设计

（1）存储方案的选择

我们对原始数据与二级索引数据的存储方式进行了探讨。大致可以分为以下两种方式：

①**分开存储**：原始数据和二级索引分别存储在两个 LevelDB 数据库中。

②**合并存储**：原始数据和二级索引共存于一个 LevelDB 数据库中，通过键的前缀区分数据类型。

以下是两种方案的对比分析表格：

表 1 存储方式的对比

评价指标	分开存储	合并存储
代码复杂度	较高：需管理多个数据库实例	较低：单一数据库，使用前缀区分键
存储空间	两个数据库文件，占用略大	单个数据库文件，占用略小
查询性能	二级索引查询效率较高，读写互不干扰	索引与数据共用存储，性能稍低
一致性维护	需跨数据库事务性操作，不容易维护	使用 WriteBatch 原子操作，容易维护

此外，我们还查阅了现有的数据库设计，以下是一些实际使用中的常见例子：

①索引数据与原始数据分开存储（现代分布式或多模系统）：

Elasticsearch + 原始数据数据库：Elasticsearch 用于二级索引，原始数据存储在 MySQL、Cassandra 等系统中。索引和数据分开存储以优化全文检索性能。

HBase + Solr：HBase 用于存储原始数据，Solr 用于存储和查询索引。

②索引数据与原始数据存储在同一个数据库（大多数传统关系型数据库）：

MySQL：MySQL 的 InnoDB 存储引擎支持二级索引，数据和索引存储在同一个物理文件中，使用 B+ 树结构管理索引。

PostgreSQL：PostgreSQL 的索引和数据存储在一起，但索引可以选择不同的类型（如 B+ 树、GIN、GiST 等）。

最后，由于考虑到二级索引通常比原始数据频繁访问。如果将索引和数据混合存储，会导致查询性能下降，特别是在大量数据和索引字段更新的场景中。我们决定采用**分开存储**的方式。

（2）存储方案的实现

在确定要将索引数据与原始数据分开存储后，经过讨论，我们产生了以下两种实现方法：

①方法一：新建类 NewDB，其中包含两个小数据库，一个是存放索引数据，一个存放原始数据。当用户想在某个字段建立二级索引时，NewDB 会将原始数

据存入 DataDB，索引数据存入 IndexDB。同时维护两个数据库意味着数据一致性与读写性能将需要被取舍。

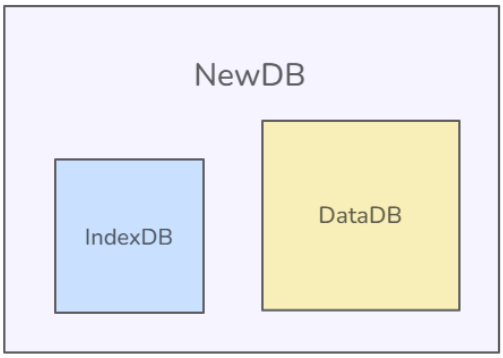

图 3 存储方式一

插入数据时，采用拼接的方式将索引信息拼接为新的 k'，以<k', ">的形式插入索引表中。插入索引时，只需要写入对应的键值对即可。

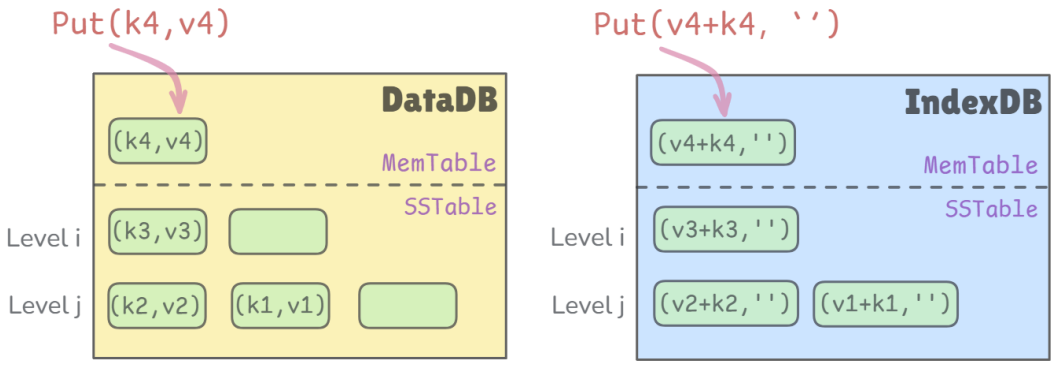

图 4 插入数据的图示

读取索引时，需要扫描整个索引表，索引表中的 k 满足字典序关系，因此利用范围扫描查询以特定前缀开头的键值对即可。

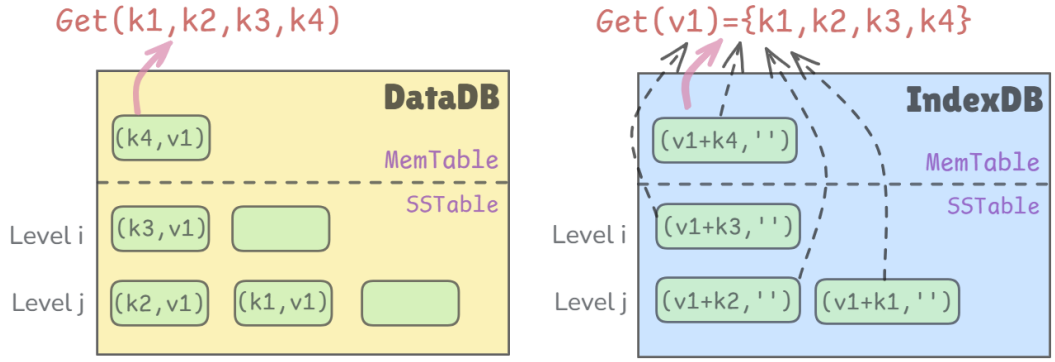

图 5 查找索引的图示

Compaction 时，不需要特定的合并操作，因为索引表中的 k 必然不同(带有删除标记的键值对除外)，直接合并写入到下一层即可。

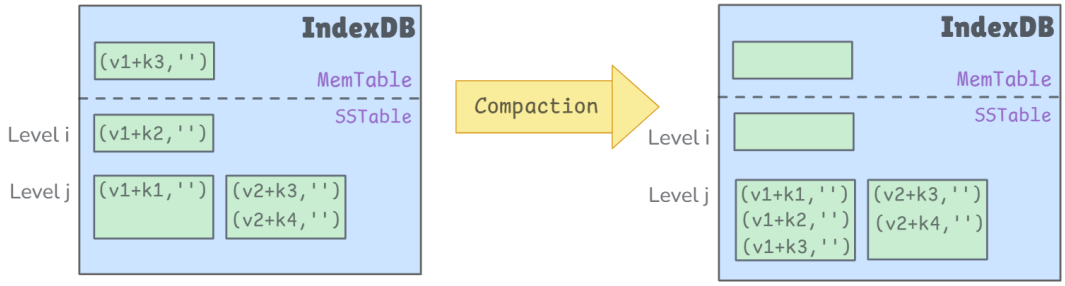

图 6 合并的图示

这种方案整体的写放大较低，而且 Compaction 操作比较简单，但是读取时需要对整个索引表进行范围查询，开销较大。

②方法二：将索引数据作为一个虚拟数据库，直接存放在原始数据库中。这样在一致性处理上会比较简单，但缺点是代码结构不清晰。

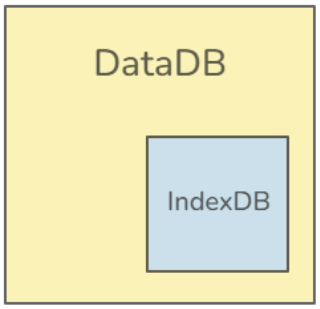

图 7 存储方式二

最终我们选取了**第一种实现方法**。

3.1.3 接口设计

（1）初版设计

①NewDB 类：包含两个实例的大数据库，包括 Open、Put_fields、Get_fields、Delete 等方法。

②IndexDB 类：继承原来 LevelDB 的 db 类，并添加了与二级索引功能相关的函数。

③DataDB 类：复用原来 LevelDB 的 db 类，可以调用 db_impl.cc 的接口。

（2）优化方案

在代码实现过程中，发现可以将二级索引的功能函数全部添加在 NewDB 类里，这样就不用专门新建 IndexDB 类，可以让索引数据库与原始数据库都复用原来的 db 类，保证了结构的清晰。

我们还在 NewDB 类中增加了成员变量，来辅助实现数据库的数据一致性、元数据持久化和崩溃恢复等功能。

最终，我们新建了 NewDB.h 与 NewDB.cc 文件，并实现了如下 NewDB 类的设计（db/NewDB.h）：

```
1.  namespace leveldb{
2.
3.  class LEVELDB_EXPORT NewDB {
4.    public:
5.    NewDB() = default;
6.
7.    NewDB(const NewDB&) = delete;
8.    NewDB& operator=(const NewDB&) = delete;
9.
10.   virtual ~NewDB();
11.
12.   std::string SerializeValue(const FieldArray& fields);
13.   FieldArray ParseValue(const std::string& value_str);
14.   std::string ConstructIndexKey(const Slice& key, const Field& field);
15.   std::string ExtractIndexKey(const Slice& key);
16.   std::string ConstructRecoverKey(std::string UserOpID, std::string TinyOpID
    , std::string DBname);
17.   std::string ConstructRecoverValue(std::string TinyOp, std::string key, std
    ::string value);
18.   std::string ExtractRecoverKey(std::string s);
19.   std::pair<std::string, std::string> ExtractRecoverValue(std::string s, std
    ::string* TinyOp);
20.   std::string ConstructUserOpID(std::thread::id thread_id);
21.   static Status Open(const Options& options, const std::string& name,
22.                      NewDB** dbptr);  // 改为返回 NewDB* 类型
23.   Status Put_fields(const WriteOptions& options, const Slice& key,
24.                     const FieldArray& fields);
25.   std::pair<FieldArray, FieldArray> UpdateIndex(const WriteOptions& options,
     const Slice& key,
26.                 const FieldArray& fields, const FieldArray& current_fields);

27.   Status Get_fields(const ReadOptions& options, const Slice& key,
28.                     FieldArray* fields);
29.   bool Delete(const WriteOptions& options, const Slice& key);
30.
31.   std::vector<std::string> FindKeysByField(Field &field);
32.   bool CreateIndexOnField(const std::string& field_name);
33.   std::vector<std::string> QueryByIndex(Field &field);
34.   bool DeleteIndex(const std::string& field_name);
```

```
35.
36.   // 用于存储已经为其创建索引的字段名称。只有当字段名在这个集合中时，才会
      在 indexDB 中插入对应的索引条目。
37.   std::unordered_set<std::string> indexed_fields_read;
38.   std::unordered_set<std::string> indexed_fields_write;
39.   std::unordered_set<std::string> putting_keys;
40.
41.   private:
42.   std::unique_ptr<DB> data_db_;
43.   std::unique_ptr<DB> index_db_;
44.   std::unique_ptr<DB> recover_db_;
45.   std::mutex db_mutex_;  // 用于跨数据库操作的互斥锁
46.   uint64_t TinyOpID;
47. };
48. }
49. #endif
```

3.2 基本功能

3.2.1 函数接口设计

（1）创建索引

函数名：newDB::CreateIndexOnField

函数功能：用户在一个字段名上创建索引。根据字段名在 data_db_中查找到所有存在该字段名的键值对的 key 和 field_value，然后构造出所有索引数据的 k-v 对，将索引数据存储到 indexDB。

（2）插入数据

函数名：newDB:: Put_fields

函数功能：用户插入一条数据，首先判断是新增的情况还是更新到情况，之所以将这两种情况分别处理，是因为如果在更新的时候只是在 index_db_中插入一条新数据，会导致在查找索引时查找到更新前的索引数据，出现错误。如果是新增需要分别在 dataDB 和 indexDB 中插入；如果是更新，需要调用 newDB::UpdateIndex 函数处理。

（3）更新数据

函数名：newDB::UpdateIndex

函数功能：被 newDB::put_fields 函数调用，比对待更新的这条记录和 data_db_当中当前最新的一条记录，得到所有被删除的 fields 和所有新增的 fields。对于删除的 fields，在 index_db_中删除该条索引数据；对于新增的 fields，在 index_db_中增加一条相应的索引数据。

（4）查询数据

①函数名：newDB::Get_fields

函数功能：用户用 key 查询 value，直接调用 data_db_的 Get_fields 函数即可。

②函数名：newDB::FindKeysByField

函数功能：没有索引情况下根据字段查找 keys，直接调用 data_db_的 FindKeysByField 即可。

③函数名：newDB::QueryByIndex

函数功能：有索引的情况下根据字段查找 keys。根据 field 利用索引键的前缀进行范围查找，提取出找到的索引键里面真正的 keys，即为 matching_keys。

（5）删除数据

①函数名：newDB::delete

函数功能：用户删除一条键值对，分别删除 data_db_的键值对和 index_db_的索引数据。

②函数名：newDB::DeleteIndex

函数功能：用户删除一个字段上的索引。对 index_db_利用索引键的前缀进行范围查找，查找到所有该字段的键值对，将它们删除。

3.2.2 基本功能实现

（1）创建索引

CreateIndexOnField 函数

该函数的功能是在用户选择某个字段创建索引时，对 dataDB 中现有的数据进行遍历，批量构建该字段的索引数据并插入到 indexDB 中。

该函数主要分为两个阶段，一是索引元数据的写入，二是遍历并构建索引的过程。在遍历过程中，需要确保每个批次的索引写入不超过特定的限制，以减少批量写入过程中的内存开销，提高效率。

首先，通过向 indexDB 写入 index_field_write 标识，持久化索引字段元数据，确保字段的索引创建请求被记录。接着，遍历 dataDB 中所有的键值对，解析数据中的字段，查找目标字段的值。如果找到匹配字段，就构造相应的索引键并批量插入 indexDB 中。

批量插入过程中，每次写入的键值对数量不得超过设定的批次大小 BATCH_SIZE_LIMIT，超过限制时立即执行写入操作，并清空批次，确保内存占用可控。如果遍历过程中迭代器发生错误，则返回 false，表示索引创建未完成。

在遍历结束后，执行最后一批未写入的索引数据，并再次向 indexDB 写入 index_field_read 标识，表示该字段的索引数据已完全构建。整个过程采用互斥锁 db_mutex_确保线程安全，避免在索引创建过程中发生竞态条件。

该函数确保了索引创建过程中数据一致性与完整性，即使遍历过程中发生错误，也能通过返回 false 及时终止，避免错误索引的产生。

```
1.  bool NewDB::CreateIndexOnField(const std::string& field_name) {
2.    // 将索引字段元数据持久化到 indexDB
3.    index_db_->Put(WriteOptions(), "index_field_write:" + field_name, "1");
```

```
    std::unique_lock<std::mutex> lock(db_mutex_, std::defer_lock);
    lock.lock();
    indexed_fields_write.insert(field_name);
    lock.unlock();

const size_t BATCH_SIZE_LIMIT = 1000;  // 每个批次限制 1000 条记录
  size_t batch_count = 0;

      // 遍历 data_db_ 中的所有键值对
      leveldb::ReadOptions read_options;
      leveldb::WriteBatch batch;
      std::unique_ptr<leveldb::Iterator> it(data_db_->NewIterator(read_options));

      for (it->SeekToFirst(); it->Valid(); it->Next()) {
        // 获取主数据库的键值对
        std::string key = it->key().ToString();
        std::string value = it->value().ToString();

        // 解析值，提取字段数组
        FieldArray fields = ParseValue(value);

        // 查找指定字段
        for (const auto& field : fields) {
          if (field.first == field_name) {
            // 构造索引键
            std::string index_key = ConstructIndexKey(key, field);

            // 插入到索引数据库
            batch.Put(index_key, Slice());

            batch_count++;
            // 如果达到批次大小限制，执行写入
            if (batch_count >= BATCH_SIZE_LIMIT) {
              leveldb::Status status = index_db_->Write(leveldb::WriteOptions(), &batch);
              if (!status.ok()) {
                return false;
              }
              batch.Clear();  // 清空批次，准备下一个批次
              batch_count = 0;
            }
          }
        }
      }

```

```
48.         // 如果迭代器出错
49.         if (!it->status().ok()) {
50.             // 从头开始创建，覆盖掉原本失效的东西
51.             return false;
52.         }
53.         // 批量写入
54.         leveldb::WriteOptions write_options;
55.         Status status = index_db_->Write(write_options, &batch);
56.         if (!status.ok()) {
57.             return false;
58.         }
59.
60.         // 将索引字段元数据持久化到 indexDB
61.         index_db_->Put(WriteOptions(), "index_field_read:" + field_name, "1");
62.         lock.lock();
63.         indexed_fields_read.insert(field_name);
64.         lock.unlock();
65.         return true;
66.     }
```

（2）插入数据

①Put_fields 函数

该函数的功能是在用户向 dataDB 中插入一条数据时，判断是否需要向 indexDB 中插入对应的索引数据。

该函数分为两种情况，一种是新增的情况，另外一种是更新的情况。之所以要区分这两张情况，是因为更新 dataDB 的数据时，如果只是向 indexDB 中相应的插入新的索引数据，那么当利用旧的字段信息查找 key 时，还是可以查找到的，这样就不满足一致性了。

首先根据输入的 key，在 dataDB 中查找是否有这样的数据，如果有的话，就说明是更新的情况，如果没有，就说明是新增的情况。

对于新增的情况，遍历 value 当中所有的字段。如果这个字段已经存在，那么就需要构造相应的 indexDB 当中的 key，然后将 key 和 value 插入到准备插入 index DB 的 index batch 中。同时需要将 key 和 value 插入到准备写入 data DB 的 data batch 中。

如果是更新的情况，那么调用 UpdateIndex 函数。

该函数输入两个 Field array，一个是当前 dataDB 当中存在的 current fields，另外一个是更新后的 fields。准备两个空的 field array，分别用来存储需要删除的字段和需要更新的字段。

首先遍历 fields 看看哪些字段是在 fields 中存在，但是 current fields 中不存在的，那么这些字段就是新增的字段，添加到 insert fields 中。再看看在 current field 中同样存在但是值不同的字段，那么这些字段既需要添加到 insert fields 中，也需要添加到 deleted fields 中。然后再遍历 Current fields，看看哪些字段是在 current

fields 中存在，但是在 fields 中不存在的，那么这些字段需要插入到 deleted fields 中。

对于更新的情况，得到 insert fields 和 deleted fields 之后：首先遍历 Insert fields 中的字段，如果某个字段存在索引，那么需要构建 index Key，并且将 key 和 value 插入到 index Batch 中。然后将 key 和 value 插入到 data batch 中。最后再遍历 deleted fields 中的字段，如果某个字段存在索引，那么同样需要构建 index key，并插入 index batch 中。

最后，不论是新增还是更新的情况，都需要将准备好的 index batch 和 data batch 分别写入 Index DB 和 dataDB。

```
1.  Status NewDB::Put_fields(const WriteOptions& options, const Slice& key, const FieldArray& fields){
2.      FieldArray current_fields;
3.      Status s = Get_fields(leveldb::ReadOptions(), key, &current_fields);
4.
5.      leveldb::WriteBatch data_batch;
6.      leveldb::WriteBatch index_batch;
7.
8.      if(!s.ok()){
9.          for (const auto& field : fields) {
10.             // 如果字段名在 indexed_fields_ 中，才插入二级索引
11.             if (indexed_fields_write.find(field.first) != indexed_fields_write.end()) {
12.                 // 构造索引的 key，结构：FieldName_FieldValue_Key
13.                 std::string index_key = ConstructIndexKey(key, field);
14.                 index_batch.Put(index_key, Slice());
15.             }
16.         }
17.         std::string value = SerializeValue(fields);
18.         data_batch.Put(key.ToString(), value);
19.     }
20.
21.     else{
22.         std::pair<FieldArray, FieldArray> fieldarray_pair = UpdateIndex(options, key, fields, current_fields
    );
23.
24.         // put ops in indexdb
25.         for (const auto& field : fieldarray_pair.first) {
26.             if (indexed_fields_write.find(field.first) != indexed_fields_write.end()) {
27.                 // put into index_batch
28.                 std::string index_key = ConstructIndexKey(key, field);
29.                 index_batch.Put(index_key, Slice());
30.             }
31.         }
32.
33.         // put ops in datadb
```

```
            std::string value = SerializeValue(fields);
            data_batch.Put(key.ToString(), value);

            // delete ops in indexdb
            for (const auto& field : fieldarray_pair.second) {
                if (indexed_fields_write.find(field.first) != indexed_fields_write.end()) {
                    // delete in index_batch
                    std::string index_key = ConstructIndexKey(key, field);
                    index_batch.Delete(index_key);
                }
            }
        }

        // write into indexDB
        Status status = index_db_->Write(write_options, &index_batch);
        if(!status.ok()){
            return status;
        }

        // write into dataDB
        Status data_status = data_db_->Write(write_options, &data_batch);
        if (!data_status.ok()) {
            return data_status; // 主数据写入失败，直接返回
        }

        return Status::OK();
    }
```

②UpdateIndex 函数

```
    std::pair<FieldArray, FieldArray> NewDB::UpdateIndex(const WriteOptions& options, const Slice& key, const FieldArray& fields, const FieldArray& current_fields){
        // 构建 current_fields 的映射
        std::unordered_map<std::string, std::string> current_map;
        for (const auto& field : current_fields) {
            current_map[field.first] = field.second;
        }

        // 构建 fields 的映射
        std::unordered_map<std::string, std::string> fields_map;
        for (const auto& field : fields) {
            fields_map[field.first] = field.second;
        }

        // 删除的字段
        FieldArray deleted_fields;
```

```
        // 新增的字段
        FieldArray insert_fields;

        // fields 中的字段
        for (const auto& field : fields) {
            // fields 中存在但 current_fields 中不存在
            if (current_map.find(field.first) == current_map.end()) {
                insert_fields.push_back(field);
            } else if (current_map[field.first] != field.second) {
                // current_fields 中存在但是值不同
                insert_fields.push_back(field);
                deleted_fields.push_back(std::make_pair(field.first, current_map[field.first]));
            }
        }

        // current_fields 中的字段
        for (const auto& field : current_fields) {
            if (fields_map.find(field.first) == fields_map.end()) {
                deleted_fields.push_back(field);
            }
        }

        return std::make_pair(insert_fields, deleted_fields);
    }
```

（3）查询数据

QueryIndex 函数

该函数的功能是根据用户给出的 field 找出 indexDB 中存在该字段的所有的 key。

首先需要判断这个字段是不是存在索引，如果存在索引的话，就构造一个查找的前缀，这个前缀由字段名加字段值组成，然后用一个迭代器遍历 indexDB 中所有符合这个前缀的 Key，然后对于找到的 key 提取出当中的 data DB 中的 key，将这个 key 插入到 Matching keys 中。

```
std::vector<std::string> NewDB::QueryByIndex(Field &field){
    std::vector<std::string> matching_keys;

    if(indexed_fields_read.find(field.first) != indexed_fields_read.end()){
        std::string prefix = ConstructIndexKey(Slice(), field);  //前缀-字段名+字段值
        leveldb::Iterator* iter = index_db_->NewIterator(leveldb::ReadOptions());
        for(iter->Seek(prefix); iter->Valid() && iter->key().starts_with(prefix); iter->Next()){
            Slice index_key = iter->key();
            std::string data_key = ExtractIndexKey(index_key);  //解析索引键中的 key
```

```
            matching_keys.push_back(data_key);
        }
        delete iter;

    return matching_keys;
}
```

（4）删除数据

①Delete 函数

该函数的功能是当用户删除一条 dataDB 中的数据时，判断是否需要删除 indexDB 中相应的索引数据。

删除前同样需要判断该数据是否在 datadb 中存在，如果不存在的话，直接返回 false 即可。如果存在的话，首先删除 dataDB 中的数据。然后遍历 value 中所有的字段，如果某字段存在索引，那么需要构建 index key 并将 key 和 value 插入 indexBatch 中。最后将 data batch 和 index batch 分别写入 dataDB 和 index DB。

```
bool NewDB::Delete(const WriteOptions& options, const Slice& key){
    leveldb::WriteBatch data_batch;
    leveldb::WriteBatch index_batch;

    FieldArray fields;
    Status s = data_db_->Get_fields(leveldb::ReadOptions(), key, &fields);

    if(s.ok()){
        // delete in datadb
        data_batch.Delete(key.ToString());

        // delete in indexdb
        for (const auto& field : fields) {
            // 如果字段名在 indexed_fields_ 中，才插入二级索引
            if (indexed_fields_write.find(field.first) != indexed_fields_write.end()) {
                // 构造索引的 key，结构：FieldName_FieldValue_Key
                std::string index_key = ConstructIndexKey(key, field);
                index_batch.Delete(index_key);
            }
        }

        // write dataDB
        s = data_db_->Write(leveldb::WriteOptions(), &data_batch);
        if(!s.ok()){
            return false;
        }

```

```
            // write indexDB
            s = index_db_->Write(leveldb::WriteOptions(), &index_batch);
            if(!s.ok()){
                return false;
            }
        }
        else{
            return false;
        }

        return true;
    }
```

②DeleteIndex 函数

该函数的功能是删除某个字段的索引。

删除索引时，首先需要将该字段从存储有索引的字段的集合中删除。然后以字段名为前缀构建一个新的迭代器，迭代 IndexDB 中所有前缀是该字段名的键值对。如果找到了一个键值对，就将它的 Key 加入到 delete batch 中。每一个 delete Batch 最多能存储 1000 条键值对，当一个 delete batch 满了之后就会清空它，然后将后来的数据存储到下一个 delete batch 中。每次 delete batch 满了后都会将这批键值对写入 indexdb 中。

```
bool NewDB::DeleteIndex(const std::string& field_name){
    // 将索引字段元数据持久化到 indexDB
    index_db_->Delete(WriteOptions(), "index_field_read:" + field_name);
    index_db_->Delete(WriteOptions(), "index_field_write:" + field_name);
    std::unique_lock<std::mutex> lock(db_mutex_, std::defer_lock);
    lock.lock();
    indexed_fields_read.erase(field_name);
    indexed_fields_write.erase(field_name);
    lock.unlock();

    const size_t BATCH_SIZE_LIMIT = 1000;  // 每个批次限制 1000 条记录
    size_t batch_count = 0;

    leveldb::WriteBatch delete_batch;

    std::string prefix = field_name;
    leveldb::Iterator* iter = index_db_->NewIterator(leveldb::ReadOptions());
    for(iter->Seek(prefix); iter->Valid() && iter->key().starts_with(prefix); iter->Next()){
        Slice index_key = iter->key();
        delete_batch.Delete(index_key);
        batch_count++;
        if (batch_count >= BATCH_SIZE_LIMIT) {
            leveldb::Status status = index_db_->Write(leveldb::WriteOptions(), &delete_batch);
```

```
            if (!status.ok()) {
                return false;
            }
            delete_batch.Clear(); // 清空批次，准备下一个批次
            batch_count = 0;
        }
    }
    delete iter;

    Status s = index_db_->Write(leveldb::WriteOptions(), &delete_batch);
    if(!s.ok()){
        return false;
    }

    return true;
}
```

3.2.3 基本功能测试

在整个测试中，我们使用辅助函数 GenerateRandomString 随机生成测试数据。该函数利用全局的随机数引擎随机生成某个长度的字符串，随机数引擎的随机种子基于当前时间生成。

（1）测试用例 CreateIndexAndQueryTest

该测试用例测试了创建索引，并且利用索引查询的功能。首先插入 100 条随机生成的数据进入 dataDB，然后在 address 字段上创建索引。检验数据是否正确插入，然后检验利用索引是否能够找到正确的键。

具体的检验流程是，对于插入的 100 条数据进行遍历，然后对于每一条数据的 key 检验其是否在 dataDB 中。然后遍历这条 kv 对的所有字段，检验每个字段的字段名以及字段值是否与插入的数据一致。

检验能否利用索引查找到对应的键时，对于插入的 100 条数据中的每个键值对，构造出字段名和字段值的 field 的对。然后调用 query by index 函数得到 matching keys，判断 matching key 是否为空，并且检验 Key 是否在 matching keys 当中。

```
#include "gtest/gtest.h"
#include "db/NewDB.h"  // NewDB 的头文件
#include "leveldb/env.h"
#include "leveldb/db.h"
#include <iostream>
#include <random>
#include <ctime>

using namespace leveldb;

```

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


    // 全局的随机数引擎
    std::default_random_engine rng;

    // 设置随机种子
    void SetGlobalSeed(unsigned seed) {
        rng.seed(seed);
    }

    // 生成随机字符串
    std::string GenerateRandomString(size_t length) {
        static const char alphanum[] =
            "0123456789"
            "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            "abcdefghijklmnopqrstuvwxyz";

        std::uniform_int_distribution<int> dist(0, sizeof(alphanum) - 2);

        std::string str(length, 0);
        for (size_t i = 0; i < length; ++i) {
            str[i] = alphanum[dist(rng)];
        }
        return str;
    }

    std::string testdbname = "dbtest26";

    TEST(TestNewDB, CreateIndexAndQueryTest) {
        // 创建 NewDB 实例
        NewDB* db;
        ASSERT_TRUE(OpenDB(testdbname, &db).ok());

        // 批量插入数据
        std::vector<std::pair<std::string, FieldArray>> bulk_data;
        const int num_records = 100;

        for (int i = 0; i < num_records; ++i) {
            std::string key = "k_" + GenerateRandomString(10);
```

```
55.             FieldArray fields = {
56.                 {"name", GenerateRandomString(5)},
57.                 {"address", GenerateRandomString(15)},
58.                 {"phone", GenerateRandomString(11)}
59.             };
60.             bulk_data.emplace_back(key, fields);
61.
62.             ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
63.         }
64.
65.         // 创建索引字段
66.         db->CreateIndexOnField("address");
67.
68.         // 验证插入的数据和索引
69.         for (const auto& [key, fields] : bulk_data) {
70.             // 获取并验证字段数据
71.             FieldArray retrieved_fields;
72.             ASSERT_TRUE(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
73.             EXPECT_EQ(retrieved_fields.size(), fields.size());
74.
75.             // 验证插入的每个字段
76.             for (size_t i = 0; i < fields.size(); ++i) {
77.                 EXPECT_EQ(fields[i].first, retrieved_fields[i].first);  // 字段名
78.                 EXPECT_EQ(fields[i].second, retrieved_fields[i].second); // 字段值
79.             }
80.
81.             // 验证索引是否能正确找到对应的键
82.             Field field_to_query{"address", fields[1].second}; // 使用 address 字段进行查询
83.             std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
84.             EXPECT_FALSE(matching_keys.empty()); // 应该能找到对应的键
85.             EXPECT_NE(std::find(matching_keys.begin(), matching_keys.end(), key), matching_keys.end());
86.         }
87.
88.         delete db;
89.     }
```

（2）测试用例 PutFieldsAndDeleteIndexTest

该测试用例测试了批量插入数据然后删除索引的情况。首先，插入 100 条随机生成的数据。然后检验是否正确插入了数据，并且检验是否在 address 字段上创建了对应的索引数据。

然后选取其中的 20 条数据删除。删除之后，同样构造字段名和字段值，利用 query by index 函数进行查找，判断找到的 matching keys 是不是空的。然后删除 Address 这个索引，检验删除索引之后，是否还能利用索引找到数据。同样构造字段名和字段值的 field 对，然后用 query by index 函数进行查找，判断找到的 matching keys 是不是空的。

```
TEST(TestNewDB, PutFieldsAndDeleteIndexTest) {
    // 创建 NewDB 实例
    NewDB* db;
    ASSERT_TRUE(OpenDB(testdbname, &db).ok());

    db->CreateIndexOnField("address");

    // 批量插入数据
    std::vector<std::pair<std::string, FieldArray>> bulk_data;
    const int num_records = 100;

    for (int i = 0; i < num_records; ++i) {
        std::string key = "k_" + GenerateRandomString(10);
        FieldArray fields = {
            {"name", GenerateRandomString(5)},
            {"address", GenerateRandomString(15)},
            {"phone", GenerateRandomString(11)}
        };
        bulk_data.emplace_back(key, fields);

        ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
    }

    // 验证插入的数据和索引
    for (const auto& [key, fields] : bulk_data) {
        // 获取并验证字段数据
        FieldArray retrieved_fields;
        ASSERT_TRUE(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
        EXPECT_EQ(retrieved_fields.size(), fields.size());

        // 验证插入的每个字段
        for (size_t i = 0; i < fields.size(); ++i) {
            EXPECT_EQ(fields[i].first, retrieved_fields[i].first);  // 字段名
            EXPECT_EQ(fields[i].second, retrieved_fields[i].second); // 字段值
        }

        // 验证索引是否能正确找到对应的键
        Field field_to_query{"address", fields[1].second}; // 使用 address 字段进行查询
        std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
        EXPECT_FALSE(matching_keys.empty()); // 应该能找到对应的键
        EXPECT_NE(std::find(matching_keys.begin(), matching_keys.end(), key), matching_keys.end());
    }

```

```
44.         // 删除部分数据
45.         const int num_updates = 20;
46.
47.         for (int i = 0; i < num_updates; ++i) {
48.             auto& [key, fields] = bulk_data[i];
49.             ASSERT_TRUE(db->Delete(WriteOptions(), key));
50.         }
51.
52.         for (int i = 0; i < num_updates; ++i) {
53.             Field query_field{"address", bulk_data[i].second[1].second};
54.             std::vector<std::string> matching_keys = db->QueryByIndex(query_field);
55.             EXPECT_TRUE(matching_keys.empty());
56.         }
57.
58.         // 删除索引
59.         ASSERT_TRUE(db->DeleteIndex("address"));
60.
61.         // 验证删除索引后无法再通过索引查询到数据
62.         for (const auto& [key, fields] : bulk_data) {
63.             Field field_to_query{"address", fields[1].second}; // 使用 address 字段进行查询
64.             std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
65.             EXPECT_TRUE(matching_keys.empty()); // 索引已被删除，应该找不到任何匹配的键
66.         }
67.
68.         delete db;
69.     }
```

（3）测试用例 UpdateFieldsAndQueryIndexTest

该测试用例测试了更新部分数据，然后再进行查询的情况。

首先，批量插入 100 条随机生成的数据。然后验证数据是否正确插入，以及 indexDB 中是否正确的创建了对应的索引数据，是否能用 query by index 函数找到正确的 matching keys。

然后选取其中的 20 条数据进行更新，只更新 address 字段的字段值。检验更新后贝塔 DB 中的数据是否正确更新，然后检验新的 address 字段值在 indexDB 中能够通过 query by index 函数查找到正确的 key。最后检验旧的 address 字段值已经不能通过 query by index 函数查找到。

```
1.     TEST(TestNewDB, UpdateFieldsAndQueryIndexTest) {
2.         // 创建 NewDB 实例
3.         NewDB* db;
4.         ASSERT_TRUE(OpenDB(testdbname, &db).ok());
5.
6.         // 创建索引字段
7.         ASSERT_TRUE(db->CreateIndexOnField("address"));
8.
```

```
9.      // 批量插入数据
10.     const int num_records = 100;
11.     std::vector<std::pair<std::string, FieldArray>> initial_bulk_data;
12.
13.     for (int i = 0; i < num_records; ++i) {
14.         std::string key = "k_" + GenerateRandomString(10);
15.         FieldArray fields = {
16.             {"name", GenerateRandomString(5)},
17.             {"address", GenerateRandomString(15)},
18.             {"phone", GenerateRandomString(11)}
19.         };
20.         initial_bulk_data.emplace_back(key, fields);
21.
22.         ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
23.     }
24.
25.     // 验证初始数据的索引查询
26.     for (const auto& [key, fields] : initial_bulk_data) {
27.         Field query_field{"address", fields[1].second}; // 使用 address 字段进行查询
28.         std::vector<std::string> matching_keys = db->QueryByIndex(query_field);
29.         ASSERT_FALSE(matching_keys.empty()); // 应该能找到对应的键
30.         EXPECT_NE(std::find(matching_keys.begin(), matching_keys.end(), key), matching_keys.end());
31.     }
32.
33.     // 更新部分数据
34.     const int num_updates = 20;
35.     std::vector<std::pair<std::string, FieldArray>> updated_bulk_data;
36.
37.     for (int i = 0; i < num_updates; ++i) {
38.         auto& [key, fields] = initial_bulk_data[i];
39.         FieldArray updated_fields = {
40.             {"name", fields[0].second},
41.             {"address", GenerateRandomString(15)}, // 更新 address
42.             {"phone", fields[2].second}
43.         };
44.         updated_bulk_data.emplace_back(key, updated_fields);
45.
46.         ASSERT_TRUE(db->Put_fields(WriteOptions(), key, updated_fields).ok());
47.     }
48.
49.     // 验证更新后的数据和索引
50.     for (const auto& [key, updated_fields] : updated_bulk_data) {
51.         // 获取并验证字段数据
52.         FieldArray retrieved_fields;
```

```
53.             ASSERT_TRUE(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
54.             EXPECT_EQ(retrieved_fields.size(), updated_fields.size());
55.
56.             // 验证插入的每个字段
57.             for (size_t i = 0; i < updated_fields.size(); ++i) {
58.                 EXPECT_EQ(updated_fields[i].first, retrieved_fields[i].first);  // 字段名
59.                 EXPECT_EQ(updated_fields[i].second, retrieved_fields[i].second); // 字段值
60.             }
61.
62.             // 验证新的 address 字段值存在于索引中
63.             Field query_field_new{"address", updated_fields[1].second};
64.             std::vector<std::string> matching_keys_new = db->QueryByIndex(query_field_new);
65.             ASSERT_FALSE(matching_keys_new.empty()); // 新地址应该能找到对应的键
66.             EXPECT_NE(std::find(matching_keys_new.begin(), matching_keys_new.end(), key), matching_keys_new.end());
67.         }
68.         for (int i = 0; i < num_updates; ++i) {
69.             // 验证旧的 address 字段值不再存在于索引中
70.             Field query_field_old{"address", initial_bulk_data[i].second[1].second};
71.             std::vector<std::string> matching_keys_old = db->QueryByIndex(query_field_old);
72.             EXPECT_TRUE(matching_keys_old.empty()); // 旧地址不应该再找到对应的键
73.         }
74.
75.         delete db;
76.     }
```

（4）测试结果

基本功能测试的测试结果如下图所示。

```
chen@chen-None:~/leveldb_project2/build$ /home/chen/leveldb_project2/build/index_test_random
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from TestNewDB
[ RUN      ] TestNewDB.CreateIndexAndQueryTest
[       OK ] TestNewDB.CreateIndexAndQueryTest (2297 ms)
[ RUN      ] TestNewDB.PutFieldsAndDeleteIndexTest
[       OK ] TestNewDB.PutFieldsAndDeleteIndexTest (840 ms)
[ RUN      ] TestNewDB.UpdateFieldsAndQueryIndexTest
[       OK ] TestNewDB.UpdateFieldsAndQueryIndexTest (458 ms)
[----------] 3 tests from TestNewDB (3596 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (3596 ms total)
[  PASSED  ] 3 tests.
chen@chen-None:~/leveldb_project2/build$
```

图 8 基本功能测试结果

3.3 一致性

3.3.1 一致性设计

最初对于一致性的保证，我们构想了以下三个方案：

①方案一：使用事务

在 Put_fields 操作中：

- 先将 data_db_ 和 index_db_ 的更新操作缓存到一个事务中。
- 如果所有操作都成功，则提交事务。
- 如果有任意一个操作失败，则回滚事务。

②方案二：先写主数据库，再写索引数据库

为了避免跨数据库的写操作不一致，可以采用 "主数据库优先" 的策略。

- 将主数据写入 data_db_。
- 如果写入成功，再更新索引数据到 index_db_。
- 如果索引写入失败，删除主数据，回滚到初始状态。

③方案三：延迟索引更新

如果实时更新索引的开销较高，可以使用延迟策略。例如，将索引更新操作延迟到后台任务中进行，写入时只写主数据。

- 先写主数据，data_db_ 的写操作完成后立即返回成功。
- 将索引写入的任务推到异步线程中执行，并使用日志来跟踪未完成的索引更新。

经过进一步的讨论，我们得出了以下结论和实现一致性时需要考虑的一些问题：

①事务

在 levelDB 中实现事务会导致并发性能低、且实现的复杂性太高，因此不能用事务保证一致性。

②并发控制

写入时一致性的保证？在写入时加锁，保证在多线程情况下同一时间只有一个线程能够执行数据库的写操作；或者利用 batch 写入时本身的原子性，来保证写入的一致性。

创建索引和 put_fields 可以同时执行（并发）吗？创建索引时，如果有对 datadb 的写入，要阻塞住；或者创建索引时，先把前面的 writebatch 处理完后再 create，跑在单独线程中。

如何尽可能减少创建索引对其他 k-v 写入的阻塞？创建索引时，先读 datadb，把锁放在中间，最后写 indexdb。

③崩溃恢复

涉及到日志。但是利用日志进行跨数据库恢复，首先要能保证两个数据库日志的一致性。

3.3.2 一致性的实现

（1）writebatch 和性能

原本一条数据一个 writebatch，为了保证一致性，需要当前线程较长时间持有锁，性能较差，因此为了减少一个线程写入时一次性持有锁的时间，现在我们将写操作聚集起来作为一个（或多个）writebatch 写入。

对于创建（删除）索引而言，一次性插入的数据量可能很大，为了进一步减少这两个操作阻塞其他操作，如果一次性写入的数据量较大，将 writebatch 分割成多个，这样可以减少每个 writebatch 占有锁的时间。

以 CreateIndex 函数为例：

```
1.      // 插入到索引数据库
2.      batch.Put(index_key, Slice());
3.
4.      batch_count++;
5.      // 如果达到批次大小限制，执行写入
6.      if (batch_count >= BATCH_SIZE_LIMIT) {
7.          leveldb::Status status = index_db_->Write(leveldb::WriteOptions(), &batch);
8.          if (!status.ok()) {
9.              return false;
10.         }
11.         batch.Clear();  // 清空批次，准备下一个批次
12.         batch_count = 0;
13.     }
```

（2）将字段名记为“有索引”的时机

createindex 函数在写入 indexDB 后向 indexed_fields 中插入字段名，deleteindex 在写入 indexDB 前向 indexed_fields 中插入字段名，这样不会导致 queryindex 的错误，而且避免了 createindex 操作、deleteindex 操作完全阻塞其他操作。

但是在 createindex 时，如果并发地进行 put，那么某个字段“有索引”对于这个 put 不可见，如果 put 先完成、createindex 后完成，那么 put 的这条数据错误地没有向 indexDB 中插入索引数据。也就是说，在 createindex 时进行 put，如何保证新插入的数据也能创建索引。

为了解决这个问题，应该设计两个集合，分别是 indexed_fields_read 和 indexed_fields_write，前者在 createindex 结束时写、在 deleteindex 开始时写，后者在 createindex 开始时写、在 deleteindex 开始时写。相当于在创建索引的动作结束前，这个索引对读不可见、但对写已经可见了，而对于删除索引而言，这个索引可以在删除的开始就不可见、不管这个索引有没有删除成功。

①创建索引的函数（部分代码）：

```
1.      bool NewDB::CreateIndexOnField(const std::string& field_name) {
2.          // 将索引字段元数据持久化到 indexDB
```

```
        index_db_->Put(WriteOptions(), "index_field_write:" + field_name, "1");
        std::unique_lock<std::mutex> lock(db_mutex_, std::defer_lock);
        lock.lock();
        indexed_fields_write.insert(field_name);
        lock.unlock();

    ......

        // 将索引字段元数据持久化到 indexDB
        index_db_->Put(WriteOptions(), "index_field_read:" + field_name, "1");
        lock.lock();
        indexed_fields_read.insert(field_name);
        lock.unlock();
        return true;
    }
```

②删除索引的函数（部分代码）：

```
    bool NewDB::DeleteIndex(const std::string& field_name){
        // 将索引字段元数据持久化到 indexDB
        index_db_->Delete(WriteOptions(), "index_field_read:" + field_name);
        index_db_->Delete(WriteOptions(), "index_field_write:" + field_name);
        std::unique_lock<std::mutex> lock(db_mutex_, std::defer_lock);
        lock.lock();
        indexed_fields_read.erase(field_name);
        indexed_fields_write.erase(field_name);
        lock.unlock();
    ......

    }
```

（3）并发控制和最终一致性

首先，考虑在 Put_fields 的过程中，并发了一个 QueryIndex 的情况。在这种情况下，如果 dataDB 以及插入了数据、但 indexDB 没有插入，那么会导致读取到的 key 缺少。

因此，在 Put_fields 未完成的情况下，indexDB 中的索引数据只能多于 dataDB（对查询到的 keys 再做检查和筛选，能保证读取的正确性），但是不能少于 dataDB，因此可以先写 indexDB，再写 dataDB，这样既能保证在写的过程中读取不会出错、又不需要在写的过程中完全阻塞读取操作，在一致性和性能之间可以有一个权衡。

当 Put_fields 正常完成后，则能保证最终一致性。

①Put_fields 函数：

需要处理新增数据和更新数据。对于新增数据，先写入 indexDB，再写入 dataDB，保证 dataDB 中存在该数据时 indexDB 中一定存在相应的索引数据。对于更新数据，同样地，先写入 indexDB，再写入 dataDB，最后删除 indexDB。

Put_fields 函数的部分代码如下：

```
Status NewDB::Put_fields(const WriteOptions& options, const Slice& key, const FieldArray& fields){

.....
  // write into indexDB
  Status status = index_db_->Write(write_options, &index_batch);
  if(!status.ok()){
    return status;
  }

  // write into dataDB
  Status data_status = data_db_->Write(write_options, &data_batch);
  if (!data_status.ok()) {
    return data_status; // 主数据写入失败，直接返回
  }

  return Status::OK();
}
```

②Delete 函数：

需要先写入 dataDB，再写入 indexDB，保证 dataDB 中的数据已经删除，查找时如果没有删除 indexDB 也可以通过筛除不在 dataDB 的 key 来保证查找到正确性（因此需要修改 QueryIndex 的逻辑）。

Delete 函数的部分代码如下：

```
bool NewDB::Delete(const WriteOptions& options, const Slice& key){
  leveldb::WriteBatch data_batch;
  leveldb::WriteBatch index_batch;
.....

  // write dataDB
  s = data_db_->Write(leveldb::WriteOptions(), &data_batch);
  if(!s.ok()){
    return false;
  }

  // write indexDB
  s = index_db_->Write(leveldb::WriteOptions(), &index_batch);
  if(!s.ok()){
    return false;
  }

  return true;
```

```
}
```

③QueryIndex 函数：

在查找到 matching_keys 后，还需要对 keys 进行筛选，将 dataDB 中不存在的 key、以及 key 对应的 value 中不存在该字段值的 key 删除。

```
std::vector<std::string> NewDB::QueryByIndex(Field &field){
    std::vector<std::string> matching_keys;

    if(indexed_fields_read.find(field.first) != indexed_fields_read.end()){
        std::string prefix = ConstructIndexKey(Slice(), field);  //前缀-字段名+字段值
        leveldb::Iterator* iter = index_db_->NewIterator(leveldb::ReadOptions());
        for(iter->Seek(prefix); iter->Valid() && iter->key().starts_with(prefix); iter->Next()){
            Slice index_key = iter->key();
            std::string data_key = ExtractIndexKey(index_key);  //解析索引键中的 key
            matching_keys.push_back(data_key);
        }
        delete iter;

        // check these matching_keys and remove keys not in datadb
        for (auto& key:matching_keys) {
            FieldArray fields;
            Status s = data_db_->Get_fields(leveldb::ReadOptions(), key, &fields);

            // if field not in this k-v?
            int tag = 0;
            if(s.ok()){
                for(auto& each_field : fields){
                    if(each_field.first == field.first){
                        if(each_field.second == field.second){
                            tag = 1;
                        }
                    }
                }
            }
            if(tag == 0){
                matching_keys.erase(std::remove(matching_keys.begin(),matching_keys.end(),key),matching_keys.end());
            }
            // if(!s.ok()){
            //     matching_keys.erase(std::remove(matching_keys.begin(),matching_keys.end(),key),matching_keys.end());
            // }
        }
    }

```

```
    return matching_keys;
}
```

测试用例验证 QueryIndex 会将不符合条件的 key 筛除：

该测试用例首先将一条随机生成的数据插入 dataDB。此时，Put fields 函数应该会相应的在 indexDB 中插入一条索引数据。然后更新这条数据中 address 字段的字段值。最后再检验能否通过之前的 address 字段值查找到相应的 key。

```
TEST(TestNewDB, QueryIndexTest) {
  // 创建 NewDB 实例
  NewDB* db;
  ASSERT_TRUE(OpenNewDB(testdbname, &db).ok());

  db->CreateIndexOnField("address");

  std::string key = "k_" + GenerateRandomString(10);
  FieldArray fields = {
    {"name", GenerateRandomString(5)},
    {"address", GenerateRandomString(15)},
    {"phone", GenerateRandomString(11)}
  };
  ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());

  delete db;

  // 创建 DB 实例
  DB* data_db;
  ASSERT_TRUE(OpenDB(testdbname + "_data", &data_db).ok());
  // update the address
  FieldArray fields_new = {
    {"name", fields[0].second},
    {"address", GenerateRandomString(15)},
    {"phone", fields[2].second}
  };
  ASSERT_TRUE(data_db->Put_fields(WriteOptions(), key, fields_new).ok());
  delete data_db;

  // 创建 NewDB 实例
  NewDB* db_2;
  ASSERT_TRUE(OpenNewDB(testdbname, &db_2).ok());

  // 验证索引是否能正确找到对应的键
  Field field_to_query{"address", fields[1].second}; // 使用 address 字段进行查询
  std::vector<std::string> matching_keys = db_2->QueryByIndex(field_to_query);
  EXPECT_TRUE(matching_keys.empty());

```

```
39.     delete db_2;
40.   }
```

但是，当两个并发的线程同时通过 put_fields 操作更改同一个 key 的值时，以上方式不能保证最终一致性。

例如以下场景（name 字段上有索引）：

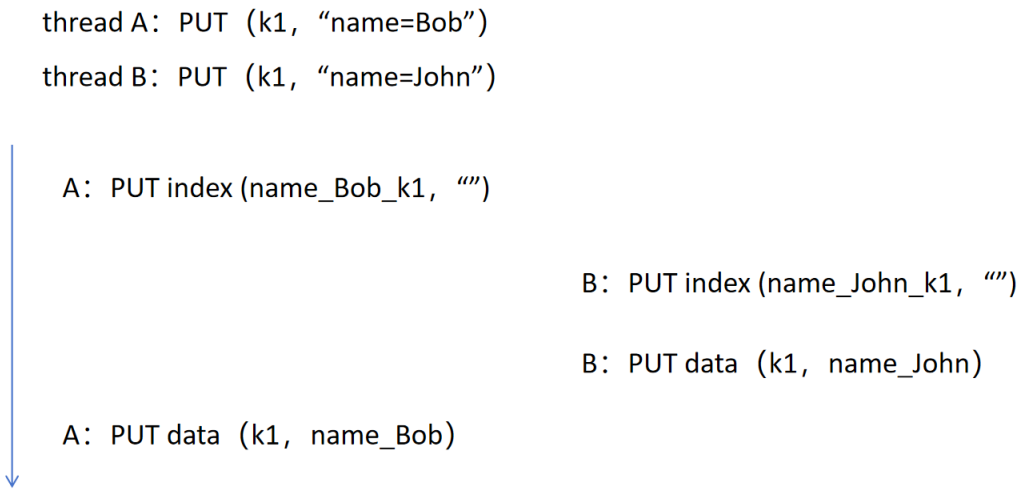

图 9 插入相同 key 的数据时的不一致场景

最终 dataDB 中 k1 对应的值是 Bob，但是 name=Bob 这个字段在 indexDB 中并没有对应的索引数据，因为在 B 线程写入 indexDB 时已经 name_Bob_k1 这条索引数据删除了。

因此，需要一个方法来保证两个操作相同 key 的 put_fields 函数不能并发，必须等待其中一个执行完，另一个才能开始执行。

所以增加了“行锁”的机制：

- cv 是一个条件变量，用于线程间的通信。
- m 是一个互斥锁，用来保护对 ready 的访问。
- ready 表示某个 key 是否已经可以被访问。
- putting keys 存储了当前在被其他线程操作的 keys。

```
1.    std::condition_variable cv;
2.    std::mutex m;
3.    bool ready = false;
4.
5.    std::unordered_set<std::string> putting_keys;
```

当一个 put field 函数开始执行时，会先检查一下 putting keys 集合中是否已经有当前 key。如果有的话，那么就需要等待，直到正在操作当前 key 的线程结束执行，将 key 从 putting keys 中移除，将 ready 改为 true，并且 cv 通知其他线程。

```
1.    Status NewDB::Put_fields(const WriteOptions& options, const Slice& key, const FieldArray& fields){
2.      std::string UserOpID = ConstructUserOpID(std::this_thread::get_id());
3.
4.      // row lock
5.      std::unique_lock<std::mutex> lock(db_mutex_, std::defer_lock);
```

```
6.          std::unique_lock<std::mutex> row_lock(m, std::defer_lock);
7.          ready = (putting_keys.find(key.ToString()) == putting_keys.end()); // no another thread putting the sa
    me key
8.
9.          if (!ready) {
10.             cv.wait(row_lock, []{return ready;});
11.         }
12.
13.         lock.lock();
14.         putting_keys.insert(key.ToString());
15.         lock.unlock();
16.         // row lock
17.       ......
18.
19.         // row lock
20.         lock.lock();
21.         putting_keys.erase(key.ToString());
22.         lock.unlock();
23.
24.         ready = true;
25.         cv.notify_all();
26.         // row lock
27.
28.         return Status::OK();
29.     }
```

（4）崩溃恢复

①基于日志恢复

为了避免两个数据库的两个日志之间不一致的问题，在 newDB 类新创建一个日志，记录两个数据库的 log，基于此做崩溃恢复。

修改 newDB::open 的逻辑，在打开一个数据库时先初始化一个日志。

修改 put_fields 的逻辑，先进行遍历字段的操作，将所有需要创建索引的字段的索引键构造出来并暂存起来，然后写日志，最后再写入 indexDB 和 dataDB。

恢复时基于 newDB 的 log 恢复两个 log，但是需要 newDB 的 log 文件能区分每条记录是属于哪个 db，因此涉及到 log 内部的结构处理，而且 log 当中存在时间戳和 log_num 等信息，如果在恢复时不加以变化直接复制给两个 log，可能会在后续的数据恢复时出现问题（可能需要读取时间戳和 log_num 等信息对 log 进行筛选）。

②新的崩溃恢复方法

由于以上日志的设计有如上问题和不足，可能会导致实现的复杂性和数据错误的问题。

所以新设计了一种崩溃恢复的方式：新建了第三个 db，命名为 RecoverDB，以键值对的形式存储用于崩溃恢复的数据。

名词解释 UserOP 和 TinyOP：一个用户操作（UserOP）包含若干个微小操作（TinyOP），例如用户创建一个索引，这个操作包含若干个向 indexDB 当中 put 的操作。

该数据库存储了恢复时需要重新执行的所有操作，该数据库的生成逻辑：

类似日志的原理，在执行一个 UserOP 中的所有 TinyOP（例如先向 indexDB 插入 10 条数据、再向 dataDB 插入一条数据）之前，在 RecoverDB 中插入所有操作的内容。

如果 UserOP 成功执行完毕，那么删除该数据库中所有属于当前 UserOP 的数据；

如果 UserOP 的过程中发生崩溃，在整个数据库重启时，按照该数据库中的记录 replay 即可恢复数据。

RecoverDB 的键值对格式为：

Key：UserOpID_TinyOpID_DBname Value：TinyOp_key_value

UserOpID 由 UserOp 开始的时间戳和当前用户线程的 thread_id 组成，保证该字段的唯一性。其中时间戳是为了在多线程情况下，在恢复时按顺序恢复多个 UserOp。thread_id 是为了防止时间戳相同导致删除了 RecoverDB 中其他 UserOp 的数据（因为删除时是以 UserOpID 来识别不同 UserOp 的数据）。

每个微小操作（例如 put_K1_nameAlice）一个 TinyOpID，用于保证恢复时同一个 UserOp 内部恢复的顺序。

DBname 即为该操作写的数据库（dataDB 或 indexDB）。

TinyOp_key_value：具体的操作内容，例如 put_K1_nameAlice。

一些辅助函数：

ConstructRecoverKey 函数：构造 recoverDB 的 key。

```
1.  std::string NewDB::ConstructRecoverKey(std::string UserOpID, std::string TinyOpID, std::string DBname){
2.      std::string s;
3.      PutLengthPrefixedSlice(&s, Slice(UserOpID));
4.      PutLengthPrefixedSlice(&s, Slice(TinyOpID));
5.      PutLengthPrefixedSlice(&s, Slice(DBname));
6.      return s;
7.  }
```

ConstructRecoverValue 函数：构造 recoverDB 的 value。

```
1.  std::string NewDB::ConstructRecoverValue(std::string TinyOp, std::string key, std::string value){
2.      std::string s;
3.      PutLengthPrefixedSlice(&s, Slice(TinyOp));
4.      PutLengthPrefixedSlice(&s, Slice(key));
5.      PutLengthPrefixedSlice(&s, Slice(value));
6.      return s;
```

```
7.          }
```

ExtractRecoverKey 函数：从 recoverDB 的 key 当中提取出数据库名称（dataDB/indexDB）。

```
1.          std::string NewDB::ExtractRecoverKey(std::string s){
2.              Slice input(s);
3.              Slice v;
4.              GetLengthPrefixedSlice(&input, &v);
5.              GetLengthPrefixedSlice(&input, &v);
6.              GetLengthPrefixedSlice(&input, &v);
7.              return v.ToString();
8.          }
```

ExtractRecoverValue 函数：从 recoverDB 的 value 中提取出操作名称（PUT/DELETE）、字段名、字段值。

```
1.          std::pair<std::string, std::string> NewDB::ExtractRecoverValue(std::string s, std::string* TinyOp){
2.              Slice input(s);
3.              Slice v;
4.              GetLengthPrefixedSlice(&input, &v);
5.              *TinyOp = v.ToString();
6.              GetLengthPrefixedSlice(&input, &v);
7.              std::string key = v.ToString();
8.              GetLengthPrefixedSlice(&input, &v);
9.              std::string value = v.ToString();
10.             return std::make_pair(key, value);
11.         }
```

ConstructUserOpID 函数：利用时间戳和线程号构造 UserOpID。

```
1.          std::string NewDB::ConstructUserOpID(std::thread::id thread_id){
2.              auto now = std::chrono::system_clock::now();
3.              std::time_t now_t = std::chrono::system_clock::to_time_t(now);
4.
5.              std::ostringstream oss;
6.              oss << thread_id;
7.              std::string UserOpID = std::to_string(now_t) + oss.str();
8.              return UserOpID;
9.          }
```

在所有 newDB 类的接口函数中，只有 put_fields 和 delete 两个函数内部有 dataDB 和 indexDB 两个数据库的写入操作，因此只有这两个函数需要利用 recoverDB 恢复，其他函数利用原本 leveldb 的日志机制即可恢复。

Put_fields 函数：

先构造出当前用户操作的 userOpID，然后在构造 index batch 和 data batch 的时候，分别构造出 recoverDB 的键值对，向 recover batch 中插入相应的数据。

每向 recover batch 中插入一条数据，就要将 tinyOpID 加一。

当 data batch、index batch 和 recover batch 都构造完成后，将数据写入三个数据库时，顺序是先写入 recoverDB，再写入 IndexDB，最后写入 dataDB。

如果写入数据都成功的话，就要删除 recoverDB 中与这个用户操作相关的所有数据。首先构造一个前缀 userOpID，然后创建一个新的迭代器，遍历 recover DB 中所有前缀为这个 userOpID 的数据，并将其删除。

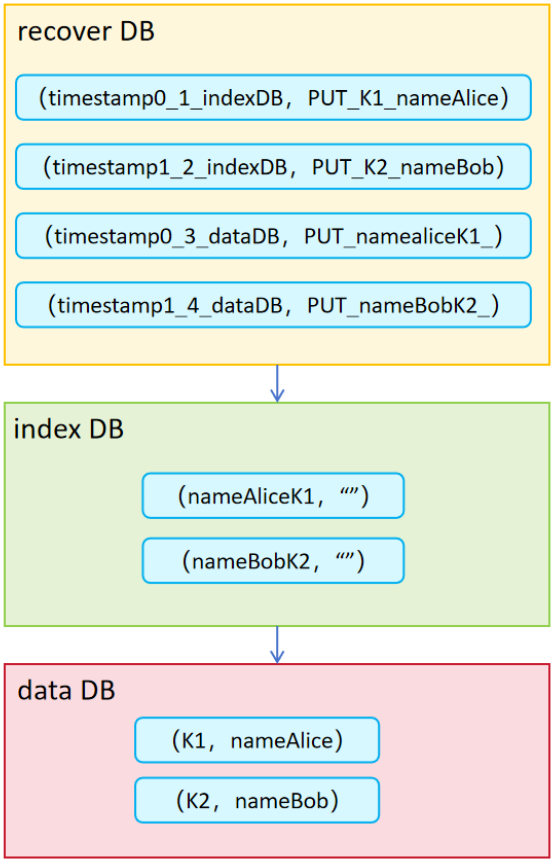

图 10 Put_fields 函数的插入流程

```
1.Status NewDB::Put_fields(const WriteOptions& options, const Slice& key, const FieldArray& fields){
2.    std::string UserOpID = ConstructUserOpID(std::this_thread::get_id());
3.
4.    // row lock
5.    std::unique_lock<std::mutex> lock(db_mutex_, std::defer_lock);
6.    std::unique_lock<std::mutex> row_lock(m, std::defer_lock);
7.    ready = (putting_keys.find(key.ToString()) == putting_keys.end()); // no another thread putting the same key
8.
9.    if (!ready) {
10.        cv.wait(row_lock, []{return ready;});
11.    }
12.
```

```
13.  lock.lock();
14.  putting_keys.insert(key.ToString());
15.  lock.unlock();
16.  // row lock
17.
18.  FieldArray current_fields;
19.  Status s = Get_fields(leveldb::ReadOptions(), key, &current_fields);
20.
21.  leveldb::WriteBatch data_batch;
22.  leveldb::WriteBatch index_batch;
23.  leveldb::WriteBatch recover_batch;
24.
25.  // uint64_t TinyOpID = 0;
26.
27.  if(!s.ok()){
28.      for (const auto& field : fields) {
29.          // 如果字段名在 indexed_fields_ 中，才插入二级索引
30.          if (indexed_fields_write.find(field.first) != indexed_fields_write.end()) {
31.              // 构造索引的 key，结构：FieldName_FieldValue_Key
32.              std::string index_key = ConstructIndexKey(key, field);
33.              index_batch.Put(index_key, Slice());
34.              // prepare recover_batch
35.              std::string Recover_key = ConstructRecoverKey(UserOpID, std::to_string(TinyOpID), "indexDB");
36.              std::string Recover_value = ConstructRecoverValue("PUT", index_key, "");
37.              recover_batch.Put(Recover_key, Recover_value);
38.
39.              TinyOpID = TinyOpID + 1;
40.          }
41.      }
42.      std::string value = SerializeValue(fields);
43.      data_batch.Put(key.ToString(), value);
44.
45.      // put dataDB's k-v into recover_batch
46.      std::string Recover_key = ConstructRecoverKey(UserOpID, std::to_string(TinyOpID), "dataDB");
47.      std::string Recover_value = ConstructRecoverValue("PUT", key.ToString(), value);
48.      recover_batch.Put(Recover_key, Recover_value);
49.
50.      TinyOpID = TinyOpID + 1;
51.  }
52.
53.  else{
54.      std::pair<FieldArray, FieldArray> fieldarray_pair = UpdateIndex(options, key, fields, current_fields);
55.
56.      // put ops in indexdb
```

```
57.     for (const auto& field : fieldarray_pair.first) {
58.       if (indexed_fields_write.find(field.first) != indexed_fields_write.end()) {
59.         // put into index_batch
60.         std::string index_key = ConstructIndexKey(key, field);
61.         index_batch.Put(index_key, Slice());
62.         // put into recover_batch
63.         std::string Recover_key = ConstructRecoverKey(UserOpID, std::to_string(TinyOpID), "indexDB");
64.         std::string Recover_value = ConstructRecoverValue("PUT", index_key, "");
65.         recover_batch.Put(Recover_key, Recover_value);
66.
67.         TinyOpID = TinyOpID + 1;
68.       }
69.     }
70.
71.     // put ops in datadb
72.     std::string value = SerializeValue(fields);
73.     data_batch.Put(key.ToString(), value);
74.
75.     // put dataDB's k-v into recover_batch
76.     std::string Recover_key = ConstructRecoverKey(UserOpID, std::to_string(TinyOpID), "dataDB");
77.     std::string Recover_value = ConstructRecoverValue("PUT", key.ToString(), value);
78.     recover_batch.Put(Recover_key, Recover_value);
79.     TinyOpID = TinyOpID + 1;
80.
81.     // delete ops in indexdb
82.     for (const auto& field : fieldarray_pair.second) {
83.       if (indexed_fields_write.find(field.first) != indexed_fields_write.end()) {
84.         // delete in index_batch
85.         std::string index_key = ConstructIndexKey(key, field);
86.         index_batch.Delete(index_key);
87.         // delete in recover_batch
88.         std::string Recover_key = ConstructRecoverKey(UserOpID, std::to_string(TinyOpID), "indexDB");
89.         std::string Recover_value = ConstructRecoverValue("DELETE", index_key, "");
90.         recover_batch.Put(Recover_key, Recover_value);
91.
92.         TinyOpID = TinyOpID + 1;
93.       }
94.     }
95.   }
96.
97.   // write into RocoverDB
98.   leveldb::WriteOptions write_options;
99.   s = recover_db_->Write(write_options, &recover_batch);
100.   if(!s.ok()){
```

```
101.     return s;
102.   }
103.
104.   // write into indexDB
105.   Status status = index_db_->Write(write_options, &index_batch);
106.   if(!status.ok()){
107.     return status;
108.   }
109.
110.   // write into dataDB
111.   Status data_status = data_db_->Write(write_options, &data_batch);
112.   if (!data_status.ok()) {
113.     return data_status; // 主数据写入失败，直接返回
114.   }
115.
116.   //delete TinyOps of this UserOp in RecoverDB
117.   leveldb::WriteBatch batch;
118.   std::string prefix;
119.   PutLengthPrefixedSlice(&prefix, Slice(UserOpID));
120.   leveldb::Iterator* iter = recover_db_->NewIterator(leveldb::ReadOptions());
121.   for(iter->Seek(prefix); iter->Valid() && iter->key().starts_with(prefix); iter->Next()){
122.     Slice Recover_key = iter->key();
123.     batch.Delete(Recover_key);
124.   }
125.   delete iter;
126.   recover_db_->Write(leveldb::WriteOptions(), &batch);
127.
128.   // row lock
129.   lock.lock();
130.   putting_keys.erase(key.ToString());
131.   lock.unlock();
132.
133.   ready = true;
134.   cv.notify_all();
135.   // row lock
136.
137.   return Status::OK();
138.}
```

Delete 函数：

和 put_fields 函数的逻辑基本类似，不同在于将当 data batch、index batch 和 recover batch 都构造完成后，将数据写入三个数据库时，顺序是先写入 recoverDB，再写入 dataDB，最后写入 IndexDB。

崩溃恢复的流程：

在 newDB 类的 open 函数中，增加一段恢复代码：

按顺序遍历 recoverDB 中的所有数据，提取出写入的数据库名称、操作名称、字段名和字段值，根据以上信息进行相应的恢复操作，恢复完成后删除 recoverDB 中的数据。

```
1.          // recover dataDB&indexDB using recoverDB
2.          leveldb::Iterator* iter = recoverDB->NewIterator(leveldb::ReadOptions());
3.          Slice recover_key;
4.          for(iter->SeekToFirst(); iter->Valid(); iter->Next()){
5.              recover_key = iter->key();
6.              std::string DBname = (*dbptr)->ExtractRecoverKey(recover_key.ToString());
7.              std::string TinyOp;
8.              std::pair<std::string, std::string> k_v = (*dbptr)->ExtractRecoverValue(iter->value().ToString(), &Tin
    yOp);
9.              if(DBname == "dataDB"){
10.                 if(TinyOp == "PUT"){
11.                     dataDB->Put(leveldb::WriteOptions(), k_v.first, k_v.second);
12.                 }
13.                 else{
14.                     dataDB->Delete(leveldb::WriteOptions(), k_v.first);
15.                 }
16.             }
17.             else{
18.                 if(TinyOp == "PUT"){
19.                     indexDB->Put(leveldb::WriteOptions(), k_v.first, k_v.second);
20.                 }
21.                 else{
22.                     indexDB->Delete(leveldb::WriteOptions(), k_v.first);
23.                 }
24.             }
25.             recoverDB->Delete(leveldb::WriteOptions(), recover_key);
26.         }
27.         // std::cout << "recover " << i << " rows, end key is " << keystr << std::endl;
28.         delete iter;
```

3.3.3 一致性测试

（1）崩溃恢复的测试（consistency_test.cc）

①测试用例 PUTRecoveryTest

该测试用例测试了新增数据后模拟崩溃，然后进行恢复的情况。首先，批量插入 100 条随机生成的数据，Address 字段有索引。然后模拟数据库崩溃，在 dataDB 和 indexDB 中分别删除部分数据。

模拟数据库崩溃之后，重新创建一个 new DB 指针，重新打开数据库。遍历 100 条插入的数据，检验 dataDB 中是否有相应的数据存在、字段名和字段值是否一致。最后再检验 indexDB 中是否有相应的索引数据。

```
TEST(TestNewDB, PUTRecoveryTest) {
// put_fields-新增
    // 创建 NewDB 实例
    NewDB* db;
    ASSERT_TRUE(OpenNewDB(testdbname, &db).ok());

    db->CreateIndexOnField("address");
    // 批量插入数据
    std::vector<std::pair<std::string, FieldArray>> bulk_data;
    std::vector<std::string> bulk_index;
    int num_records = 100;
    for (int i = 0; i < num_records; ++i) {
        std::string key = "k_" + GenerateRandomString(10);
        FieldArray fields = {
            {"name", GenerateRandomString(5)},
            {"address", GenerateRandomString(15)},
            {"phone", GenerateRandomString(11)}
        };
        bulk_data.emplace_back(key, fields);
        bulk_index.emplace_back(db->ConstructIndexKey(key, fields[1]));

        // std::cout << key << "..." << db->SerializeValue(fields) << std::endl;

        ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
    }
    // for(auto& key:bulk_index){
    //     std::cout << key << std::endl;
    // }
    delete db;

// 模拟崩溃
    // 创建 DB 实例
    DB* data_db;
    ASSERT_TRUE(OpenDB(testdbname + "_data", &data_db).ok());
    // 删除部分数据
    int num_updates = 20;
    for (int i = 0; i < num_updates; ++i) {
        auto& [key, fields] = bulk_data[i];
        ASSERT_TRUE(data_db->Delete(WriteOptions(), key).ok());
    }
    delete data_db;
```

```
42.
43.     // 创建 DB 实例
44.     DB* index_db;
45.     ASSERT_TRUE(OpenDB(testdbname + "_index", &index_db).ok());
46.     for (int i = 0; i < num_updates; ++i) {
47.       auto& key = bulk_index[i];
48.       ASSERT_TRUE(index_db->Delete(WriteOptions(), key).ok());
49.     }
50.     delete index_db;
51.
52.   // 测试是否能通过 recoverDB 恢复
53.     // 创建 NewDB 实例
54.     NewDB* db_2;
55.     ASSERT_TRUE(OpenNewDB(testdbname, &db_2).ok());
56.
57.     // 验证插入的数据和索引
58.     for (const auto& [key, fields] : bulk_data) {
59.       // 获取并验证字段数据
60.       FieldArray retrieved_fields;
61.       ASSERT_TRUE(db_2->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
62.
63.       // 验证插入的每个字段
64.       for (size_t i = 0; i < fields.size(); ++i) {
65.         EXPECT_EQ(fields[i].first, retrieved_fields[i].first);  // 字段名
66.         EXPECT_EQ(fields[i].second, retrieved_fields[i].second); // 字段值
67.       }
68.     }
69.     delete db_2;
70.
71.     // 创建 DB 实例
72.     DB* index_db_2;
73.     ASSERT_TRUE(OpenDB(testdbname + "_index", &index_db_2).ok());
74.     for (const auto& key : bulk_index) {
75.       // 获取并验证字段数据
76.       std::string value;
77.       ASSERT_TRUE(index_db_2->Get(ReadOptions(), Slice(key), &value).ok());
78.     }
79.     delete index_db_2;
80.   }
```

②测试用例 UpdateRecoveryTest

该测试用例测试了更新数据后模拟崩溃并检验是否能够恢复的情况。同样先批量插入 100 条随机生成的数据。然后更新其中 20 条的 address 字段。然后模拟数据库崩溃，删除 dataDB 和 indexDB 的部分数据。

在模拟崩溃之后，重新打开数据库。检验插入的 100 条数据是否在 dataDB 中存在，并且检验 indexDB 中是否存在相应的索引数据。

```
TEST(TestNewDB, UpdateRecoveryTest) {
// put_fields-更新
  // 创建 NewDB 实例
  NewDB* db_3;
  ASSERT_TRUE(OpenNewDB(testdbname, &db_3).ok());

  db_3->CreateIndexOnField("address");
  // 批量插入数据
  std::vector<std::pair<std::string, FieldArray>> bulk_data;
  std::vector<std::string> bulk_index;
  int num_records = 100;
  for (int i = 0; i < num_records; ++i) {
    std::string key = "k_" + GenerateRandomString(10);
    FieldArray fields = {
      {"name", GenerateRandomString(5)},
      {"address", GenerateRandomString(15)},
      {"phone", GenerateRandomString(11)}
    };
    bulk_data.emplace_back(key, fields);
    bulk_index.emplace_back(db_3->ConstructIndexKey(key, fields[1]));

    // std::cout << key << "..." << db->SerializeValue(fields) << std::endl;

    ASSERT_TRUE(db_3->Put_fields(WriteOptions(), key, fields).ok());
  }

  // 更新数据
  std::vector<std::pair<std::string, FieldArray>> bulk_data_update;
  std::vector<std::string> bulk_index_update;
  num_records = 20;
  for (int i = 0; i < num_records; ++i) {
    auto& [key, fields] = bulk_data[i];
    FieldArray fields_update = {
      {"name", fields[0].second},
      {"address", GenerateRandomString(15)},
      {"phone", fields[2].second}
    };
    bulk_data_update.emplace_back(key, fields_update);
    bulk_index_update.emplace_back(db_3->ConstructIndexKey(key, fields_update[1]));

    ASSERT_TRUE(db_3->Put_fields(WriteOptions(), key, fields_update).ok());
  }
```

```
43.
44.             delete db_3;
45.
46.
47.         // 模拟崩溃
48.             // 创建 DB 实例
49.             DB* data_db_2;
50.             ASSERT_TRUE(OpenDB(testdbname + "_data", &data_db_2).ok());
51.             // 删除部分数据
52.             int num_updates = 10;
53.             for (int i = 0; i < num_updates; ++i) {
54.                 auto& [key, fields] = bulk_data_update[i];
55.                 ASSERT_TRUE(data_db_2->Delete(WriteOptions(), key).ok());
56.             }
57.             delete data_db_2;
58.
59.             // 创建 DB 实例
60.             DB* index_db_3;
61.             ASSERT_TRUE(OpenDB(testdbname + "_index", &index_db_3).ok());
62.             for (int i = 0; i < num_updates; ++i) {
63.                 auto& key = bulk_index_update[i];
64.                 ASSERT_TRUE(index_db_3->Delete(WriteOptions(), key).ok());
65.             }
66.             delete index_db_3;
67.
68.         // 测试是否能通过 recoverDB 恢复
69.             // 创建 NewDB 实例
70.             NewDB* db_4;
71.             ASSERT_TRUE(OpenNewDB(testdbname, &db_4).ok());
72.
73.             // 验证插入的数据和索引
74.             for (const auto& [key, fields] : bulk_data_update) {
75.                 // 获取并验证字段数据
76.                 FieldArray retrieved_fields;
77.                 ASSERT_TRUE(db_4->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
78.
79.                 // 验证插入的每个字段
80.                 for (size_t i = 0; i < fields.size(); ++i) {
81.                     EXPECT_EQ(fields[i].first, retrieved_fields[i].first);  // 字段名
82.                     EXPECT_EQ(fields[i].second, retrieved_fields[i].second); // 字段值
83.                 }
84.             }
85.             delete db_4;
86.
```

```
        // 创建 DB 实例
        DB* index_db_4;
        ASSERT_TRUE(OpenDB(testdbname + "_index", &index_db_4).ok());
        for (const auto& key : bulk_index_update) {
            // 获取并验证字段数据
            std::string value;
            ASSERT_TRUE(index_db_4->Get(ReadOptions(), Slice(key), &value).ok());
        }
        delete index_db_4;
    }
```

③测试用例 DeleteRecoveryTest

该测试用例测试了删除数据后模拟崩溃，然后检验是否能正确恢复的情况。

首先同样批量插入 100 条随机生成的数据，然后删除 20 条 dataDB 中的数据。再模拟崩溃，将 dataDB 和 indexDB 的部分数据恢复（没有删除成功）。

最后，测试是否能够正确的恢复，将数据正确的删除：检验 dataDB 中的数据和 indexDB 中的数据是否都正确删除。

```
    TEST(TestNewDB, DeleteRecoveryTest) {
        // 创建 NewDB 实例
        NewDB* db_3;
        ASSERT_TRUE(OpenNewDB(testdbname, &db_3).ok());

        db_3->CreateIndexOnField("address");
        // 批量插入数据
        std::vector<std::pair<std::string, FieldArray>> bulk_data;
        std::vector<std::string> bulk_index;
        int num_records = 100;
        for (int i = 0; i < num_records; ++i) {
            std::string key = "k_" + GenerateRandomString(10);
            FieldArray fields = {
                {"name", GenerateRandomString(5)},
                {"address", GenerateRandomString(15)},
                {"phone", GenerateRandomString(11)}
            };
            bulk_data.emplace_back(key, fields);
            bulk_index.emplace_back(db_3->ConstructIndexKey(key, fields[1]));

            ASSERT_TRUE(db_3->Put_fields(WriteOptions(), key, fields).ok());
        }

        // 删除数据
        std::vector<std::pair<std::string, FieldArray>> bulk_data_delete;
        std::vector<std::string> bulk_index_delete;
        num_records = 20;
        for (int i = 0; i < num_records; ++i) {
```

```
29.             auto& [key, fields] = bulk_data[i];
30.             bulk_data_delete.emplace_back(key, fields);
31.             bulk_index_delete.emplace_back(db_3->ConstructIndexKey(key, fields[1]));
32.
33.             ASSERT_TRUE(db_3->Delete(WriteOptions(), key));
34.         }
35.
36.         delete db_3;
37.
38.     // 模拟崩溃
39.         // 创建 DB 实例
40.         DB* data_db_2;
41.         ASSERT_TRUE(OpenDB(testdbname + "_data", &data_db_2).ok());
42.         // 恢复部分数据
43.         int num_updates = 10;
44.         for (int i = 0; i < num_updates; ++i) {
45.             auto& [key, fields] = bulk_data_delete[i];
46.             ASSERT_TRUE(data_db_2->Put_fields(WriteOptions(), key, fields).ok());
47.         }
48.         delete data_db_2;
49.
50.         // 创建 DB 实例
51.         DB* index_db_3;
52.         ASSERT_TRUE(OpenDB(testdbname + "_index", &index_db_3).ok());
53.         for (int i = 0; i < num_updates; ++i) {
54.             auto& key = bulk_index_delete[i];
55.             ASSERT_TRUE(index_db_3->Put(WriteOptions(), key, "").ok());
56.         }
57.         delete index_db_3;
58.
59.     // 测试是否能通过 recoverDB 恢复
60.         // 创建 NewDB 实例
61.         NewDB* db_4;
62.         ASSERT_TRUE(OpenNewDB(testdbname, &db_4).ok());
63.
64.         // 验证插入的数据和索引
65.         for (const auto& [key, fields] : bulk_data_delete) {
66.             // 获取并验证字段数据
67.             FieldArray retrieved_fields;
68.             ASSERT_FALSE(db_4->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
69.         }
70.         delete db_4;
71.
72.         // 创建 DB 实例
```

```
73.             DB* index_db_4;
74.             ASSERT_TRUE(OpenDB(testdbname + "_index", &index_db_4).ok());
75.             for (const auto& key : bulk_index_delete) {
76.               // 获取并验证字段数据
77.               std::string value;
78.               ASSERT_FALSE(index_db_4->Get(ReadOptions(), Slice(key), &value).ok());
79.             }
80.             delete index_db_4;
81.         }
```

④测试结果

崩溃恢复的测试结果如下：

```
chen@chen-None:~/leveldb_project2/build$ /home/chen/leveldb_project2/build/consistency_test
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from TestNewDB
[ RUN      ] TestNewDB.PUTRecoveryTest
[       OK ] TestNewDB.PUTRecoveryTest (229 ms)
[ RUN      ] TestNewDB.UpdateRecoveryTest
[       OK ] TestNewDB.UpdateRecoveryTest (250 ms)
[ RUN      ] TestNewDB.DeleteRecoveryTest
[       OK ] TestNewDB.DeleteRecoveryTest (233 ms)
[ RUN      ] TestNewDB.QueryIndexTest
[       OK ] TestNewDB.QueryIndexTest (109 ms)
[----------] 4 tests from TestNewDB (822 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (822 ms total)
[  PASSED  ] 4 tests.
chen@chen-None:~/leveldb_project2/build$
```

图 11 崩溃恢复测试结果

（2）并发的测试（Concurrency_test.cc）

①测试用例 ConcurrencyALLTest

该测试用例测试了很多 put delete 函数，和创建或删除索引并发时能否保证数据被正确的插入或删除。

向 dataDB 中插入 1 万条随机生成的数据，并创建一个向量 threads，用于保存所有当前正在执行的线程。首先，将一个创建索引的线程插入到 threads 当中，并且让它开始运行。接下来，在创建索引的线程运行的同时，并发 100 个线程，每个都随机的进行 put 或者 delete 的操作。

在每个线程的内部，会对于数据是否正确的插入或删除进行检验，检验的结果会插入到一个向量 results 当中（结果为 true or false）。等待所有的 100 个线程完成，然后遍历 results 当中的每一个结果，如果有结果为 false，那么就表明测试不成功。

删除索引和 100 个线程的并发也是类似的测试方法。

```
1.  TEST(TestNewDB, ConcurrencyALLTest) {
2.    // 创建 NewDB 实例
3.    NewDB* db;
4.    ASSERT_TRUE(OpenNewDB(testdbname, &db).ok());
```

```
5.
6.          // 批量插入数据
7.          const int num_records = 10000;
8.
9.          for (int i = 0; i < num_records; ++i) {
10.             // std::string key = "k_" + GenerateRandomString(20);
11.             std::string key = "k_" + std::to_string(i);
12.             FieldArray fields = {
13.                 {"name", GenerateRandomString(5)},
14.                 {"address", GenerateRandomString(15)},
15.                 {"phone", GenerateRandomString(11)}
16.             };
17.             ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
18.         }
19.
20.         // 创建线程向量用于保存所有线程对象
21.         std::vector<std::thread> threads;
22.         std::vector<std::thread> threads2;
23.
24.         threads.emplace_back([=]() { thread_task_createindex(db, 0); });
25.
26.         // 启动 100 个线程，每个线程都运行 thread_task 函数
27.         int thread_num = 100;
28.         for (int i = 1; i <= thread_num; ++i) {
29.             // 创建随机设备用于种子
30.             std::random_device rd;
31.
32.             // 使用随机设备创建一个均匀分布的随机数生成器
33.             std::uniform_int_distribution<int> dist(0, 1);
34.
35.             // 根据随机数选择执行哪个代码块
36.             if (dist(rd) == 0) {
37.                 threads.emplace_back([=]() { thread_task_delete(db, i); });
38.             } else {
39.                 threads.emplace_back([=]() { thread_task_put(db, i); });
40.             }
41.             // threads.emplace_back([=]() { thread_task_put(db, i); });
42.         }
43.
44.         // 等待所有线程完成
45.         for (auto& th : threads) {
46.             if (th.joinable()) {
47.                 th.join();
48.             }
```

```
        }

        // check if all the results are true
        for(const auto& result : results){
            EXPECT_EQ(result, true);
        }

        threads2.emplace_back([=]() { thread_task_deleteindex(db, 200); });

        // 启动 100 个线程，每个线程都运行 thread_task 函数
        for (int i = 201; i <= thread_num+200; ++i) {
            // 创建随机设备用于种子
            std::random_device rd;

            // 使用随机设备创建一个均匀分布的随机数生成器
            std::uniform_int_distribution<int> dist(0, 1);

            // 根据随机数选择执行哪个代码块
            if (dist(rd) == 0) {
                threads2.emplace_back([=]() { thread_task_delete(db, i); });
            } else {
                threads2.emplace_back([=]() { thread_task_put(db, i); });
            }
            // threads2.emplace_back([=]() { thread_task_put(db, i); });
        }

        // 等待所有线程完成
        for (auto& th : threads2) {
            if (th.joinable()) {
                th.join();
            }
        }

        // check if all the results are true
        for(const auto& result : results){
            EXPECT_EQ(result, true);
        }

        delete db;
    }
```

创建索引的线程函数 thread_task_createindex：

```
    void thread_task_createindex(NewDB* db, int thread_id) {
        db->CreateIndexOnField("address");
```

```
}
```

删除索引的线程函数 thread_task_deleteindex：

```
void thread_task_deleteindex(NewDB* db, int thread_id) {
    db->DeleteIndex("address");
}
```

插入数据的线程函数 thread_task_put：

该函数首先随机构造一个键值对插入 dataDB 中，然后检验这个数据是否成功插入 dataDB，并且构造出 IndexDB 中的字段名和字段值，利用这个 field 对来查找 IndexDB 中的索引对应的 key。检验是否能够成功找到，并且检验刚刚插入的 key 是不是 matching keys 中的一个。

```
void thread_task_put(NewDB* db, int thread_id) {
    std::string key = "k_" + GenerateRandomString(20);
    FieldArray fields = {
        {"name", GenerateRandomString(5)},
        {"address", GenerateRandomString(15)},
        {"phone", GenerateRandomString(11)}
    };

    // while (db->indexed_fields_write.find("address") == db->indexed_fields_write.end()) {
    //     std::this_thread::sleep_for(std::chrono::milliseconds(1));
    // }
    db->Put_fields(WriteOptions(), key, fields);

// check the consistency
    FieldArray retrieved_fields;
    if(!db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok()){
        InsertResult(false);
        return;
    }

    Field field_to_query{"address", fields[1].second}; // field_value of field address

    if (db->indexed_fields_read.find("address") == db->indexed_fields_read.end()) { // index not ready
        // std::this_thread::sleep_for(std::chrono::milliseconds(1));
        std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
        if(matching_keys.empty()){
            InsertResult(true);
            return;
        }
    }

    // index ready
```

```
        std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
        if(matching_keys.empty()){
            // std::cout << thread_id << std::endl;
            InsertResult(false);
            return;
        }

        int tag = 0;
        for(auto& matching_key : matching_keys){
            if(key == matching_key){
                tag = 1;
                break;
            }
        }
        if(tag != 1){
            InsertResult(false);
            return;
        }

        InsertResult(true);
    }
```

删除数据的线程函数 thread_task_delete：

这个函数首先先从 0~9999 中随机选择一个数字 i，然后将其作为要删除的 key。删除 Data DB 中的这个键值对。然后同样判断删除之后是否还能找到这个键值对，以及 indexDB 中是否还有相应的索引数据。

```
    void thread_task_delete(NewDB* db, int thread_id) {
        int i = getRandomInRange(0,9999);

        std::string key = "k_" + std::to_string(i);
        std::string fieldvalue;

        FieldArray retrieved_fields;
        if(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok()){
            fieldvalue = retrieved_fields[1].second;
        }

        // while (db->indexed_fields_write.find("address") == db->indexed_fields_write.end()) {
        //     std::this_thread::sleep_for(std::chrono::milliseconds(1));
        // }
        db->Delete(WriteOptions(), key);

    // check the consistency
        if(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok()){
```

```
            InsertResult(false);
            return;
        }

        Field field_to_query{"address", fieldvalue}; // field_value of field address
        if (db->indexed_fields_read.find("address") == db->indexed_fields_read.end()) { // index not ready
            // std::this_thread::sleep_for(std::chrono::milliseconds(1));
            std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
            if(matching_keys.empty()){
                InsertResult(true);
                return;
            }
        }

        // index ready
        std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
        if(!matching_keys.empty()){
            // std::cout << thread_id << std::endl;
            InsertResult(false);
            return;
        }

        InsertResult(true);
    }
```

②测试用例 ConcurrencyPutSameKeyTest：

该测试用例测试了两个线程并发的向数据库中插入同一个 key 的情况。

首先向 dataDB 中插入一条数据，然后删除 name 字段的索引，确保测试环境的干净，然后在 name 字段上创建索引。开启两个线程，分别将 name 字段的值改成 Bob 和 John，让这两个线程并发。等待两个线程执行完成。

检验刚才的 key 是否存在 dataDB 当中。对于这个 key 对应的 value，找出它当中 address 字段的字段值，然后构造一个字段名和字段值的 field 对。用 query by index 函数在 index DB 中查找对应的 matching keys，判断刚才的 key 是否存在于 matching keys 当中。

```
TEST(TestNewDB, ConcurrencyPutSameKeyTest) {
    // 创建 NewDB 实例
    NewDB* db;
    ASSERT_TRUE(OpenNewDB(testdbname, &db).ok());

    // 批量插入数据
    const int num_records = 1;

    for (int i = 0; i < num_records; ++i) {
        // std::string key = "k_" + GenerateRandomString(20);
        std::string key = "k_" + std::to_string(i);
```

```
12.             FieldArray fields = {
13.                 {"name", GenerateRandomString(5)},
14.                 {"address", GenerateRandomString(15)},
15.                 {"phone", GenerateRandomString(11)}
16.             };
17.             ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
18.         }
19.
20.         db->DeleteIndex("name");
21.
22.         db->CreateIndexOnField("name");
23.
24.         // 创建线程向量用于保存所有线程对象
25.         std::vector<std::thread> threads;
26.         std::string name;
27.
28.         for (int i = 1; i <= 1; ++i) {
29.             std::string ikey = "k_" + std::to_string(i);
30.             name = "bob";
31.             threads.emplace_back([=]() { thread_task_put_key(db, i, ikey, name); });
32.             name = "john";
33.             threads.emplace_back([=]() { thread_task_put_key(db, i, ikey, name); });
34.
35.             // 等待所有线程完成
36.             for (auto& th : threads) {
37.                 if (th.joinable()) {
38.                     th.join();
39.                 }
40.             }
41.
42.             // check data and index
43.             FieldArray retrieved_fields;
44.             Status s = db->Get_fields(ReadOptions(), Slice(ikey), &retrieved_fields);
45.             ASSERT_TRUE(s.ok());
46.
47.             std::string fieldvalue = retrieved_fields[0].second;
48.             Field field_to_query{"name", fieldvalue};
49.
50.             // fieldvalue should be found by index
51.             std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
52.             ASSERT_FALSE(matching_keys.empty());
53.             ASSERT_TRUE(std::find(matching_keys.begin(), matching_keys.end(), ikey) != matching_keys.en
    d());
54.         }
```

```
55.
56.             delete db;
57.         }
```

更新某个 key 的数据的线程函数 thread_task_put_key：

```
1.          void thread_task_put_key(NewDB* db, int thread_id, std::string key, std::string name) {
2.              FieldArray fields = {
3.                  {"name", name},
4.                  {"address", GenerateRandomString(15)},
5.                  {"phone", GenerateRandomString(11)}
6.              };
7.
8.              db->Put_fields(WriteOptions(), key, fields);
9.          }
```

③测试结果

并发测试的测试结果如下：

```
chen@chen-None:~/leveldb_project2/build$ /home/chen/leveldb_project2/build/concurrency_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TestNewDB
[ RUN      ] TestNewDB.ConcurrencyALLTest
[       OK ] TestNewDB.ConcurrencyALLTest (6390 ms)
[ RUN      ] TestNewDB.ConcurrencyPutSameKeyTest
[       OK ] TestNewDB.ConcurrencyPutSameKeyTest (580 ms)
[----------] 2 tests from TestNewDB (6971 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (6972 ms total)
[  PASSED  ] 2 tests.
chen@chen-None:~/leveldb_project2/build$
```

图 12 并发控制测试结果

4. 性能测试

本次实验中，性能的衡量指标有以下三种：吞吐量、延迟、写放大。

实验环境如下：Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz and 32.0 GB of RAM，an NVIDIA GeForce RTX 2080 Super with 8 GB of dedicated GPU memory。

4.1 查询性能的测试与分析

4.1.1 有无二级索引的查询性能比较

测试脚本首先随机生成批量数据，然后测试 FindKeysByField 和 QueryByIndex 两种方法的吞吐量和延迟，数据量分别为 10^3，10^4，10^5。

（1）测试用例 FindKeysByFieldPerformance

```
TEST(TestNewDB, FindKeysByFieldPerformance) {
    // 创建 NewDB 实例
    NewDB* db;
    ASSERT_TRUE(OpenDB(testdbname, &db).ok());

    // 批量插入数据
    std::vector<std::pair<std::string, FieldArray>> bulk_data;
    const int num_records = 100000;

    for (int i = 0; i < num_records; ++i) {
        std::string key = "k_" + GenerateRandomString(10);
        FieldArray fields = {
            {"name", GenerateRandomString(10)},
            {"address", GenerateRandomString(25)},
            {"phone", GenerateRandomString(11)}
        };
        bulk_data.emplace_back(key, fields);

        ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
    }

    // 创建索引字段
    db->CreateIndexOnField("address");
    // 测试查询延迟
    auto start_time = std::chrono::high_resolution_clock::now();  // 记录开始时间
    // 验证插入的数据和索引
    for (const auto& [key, fields] : bulk_data) {
        // 获取并验证字段数据
        FieldArray retrieved_fields;
        ASSERT_TRUE(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
        EXPECT_EQ(retrieved_fields.size(), fields.size());

        // 验证索引是否能正确找到对应的键
        Field field_to_query{"address", fields[1].second}; // 使用 address 字段进行查询
        std::vector<std::string> matching_keys = db->FindKeysByField(field_to_query);

        EXPECT_FALSE(matching_keys.empty()); // 应该能找到对应的键
        EXPECT_NE(std::find(matching_keys.begin(), matching_keys.end(), key), matching_keys.end());
    }
    auto end_time = std::chrono::high_resolution_clock::now();  // 记录结束时间
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time).count();
    cout << "Throughput: " << num_records * 1000 / duration << " OPS" << endl;
    cout << "Total Latency " << duration << " ms" << endl;
    delete db;
```

```
}
```

（2）测试用例 QueryByIndexPerformance

```
TEST(TestNewDB, QueryByIndexPerformance) {
    // 创建 NewDB 实例
    NewDB* db;
    ASSERT_TRUE(OpenDB(testdbname, &db).ok());

    // 批量插入数据
    std::vector<std::pair<std::string, FieldArray>> bulk_data;
    const int num_records = 100000;

    for (int i = 0; i < num_records; ++i) {
        std::string key = "k_" + GenerateRandomString(10);
        FieldArray fields = {
            {"name", GenerateRandomString(10)},
            {"address", GenerateRandomString(25)},
            {"phone", GenerateRandomString(11)}
        };
        bulk_data.emplace_back(key, fields);

        ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
    }

    // 创建索引字段
    db->CreateIndexOnField("address");
    // 测试查询延迟
    auto start_time = std::chrono::high_resolution_clock::now();  // 记录开始时间
    // 验证插入的数据和索引
    for (const auto& [key, fields] : bulk_data) {
        // 获取并验证字段数据
        FieldArray retrieved_fields;
        ASSERT_TRUE(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
        EXPECT_EQ(retrieved_fields.size(), fields.size());

        // 验证索引是否能正确找到对应的键
        Field field_to_query{"address", fields[1].second}; // 使用 address 字段进行查询
        std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);

        EXPECT_FALSE(matching_keys.empty()); // 应该能找到对应的键
        EXPECT_NE(std::find(matching_keys.begin(), matching_keys.end(), key), matching_keys.end());
    }
    auto end_time = std::chrono::high_resolution_clock::now();  // 记录结束时间
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time).count();
```

```
42.         cout << "Throughput: " << num_records * 1000 / duration << " OPS" << endl;
43.         cout << "Total Latency " << duration << " ms" << endl;
44.         delete db;
45.     }
```

（3）测试结果

每个实验总共进行三次，每次实验前将数据库清除，保证数据量一致，并取平均值记录于表格中。下面仅放出示例截图。

FindKeysByField

```
devcontainers@LAPTOP-6C5U4TQP:/mnt/e/leveldb_project2/build$ /mnt/e/leveldb_project2/build/index_test_random
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TestNewDB
[ RUN      ] TestNewDB.BulkCreateIndexAndQueryTest
Throughput: 70 OPS
Total Latency 141998 ms
[       OK ] TestNewDB.BulkCreateIndexAndQueryTest (143300 ms)
[----------] 1 test from TestNewDB (143300 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (143300 ms total)
[  PASSED  ] 1 test.
```

```
devcontainers@LAPTOP-6C5U4TQP:/mnt/e/leveldb_project2/build$ /mnt/e/leveldb_project2/build/index_test_random
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TestNewDB
[ RUN      ] TestNewDB.FindKeysByFieldPerformance
Throughput: 6 OPS
Total Latency 15311021 ms
[       OK ] TestNewDB.FindKeysByFieldPerformance (15326064 ms)
[----------] 1 test from TestNewDB (15326064 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (15326064 ms total)
[  PASSED  ] 1 test.
```

图 13 FindKeysByField 测试结果

QueryByIndex-方案二：变长，并记录长度

```
devcontainers@LAPTOP-6C5U4TQP:/mnt/e/leveldb_project2_consistency/leveldb_project2/build$
rformance_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TestNewDB
[ RUN      ] TestNewDB.QueryByIndexPerformance
Throughput: 47 OPS
Total Latency 2085368 ms
[       OK ] TestNewDB.QueryByIndexPerformance (2119732 ms)
[----------] 1 test from TestNewDB (2119732 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (2119732 ms total)
[  PASSED  ] 1 test.
```

图 14 QueryByIndex-方案二测试结果

QueryByIndex-方案三：用特殊字符分隔

```
devcontainers@LAPTOP-6C5U4TQP:/mnt/e/leveldb_project2/build$ /mnt/e/leveldb_project2/build/index_test_random
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TestNewDB
[ RUN      ] TestNewDB.BulkCreateIndexAndQueryTest
Throughput: 133333 OPS
Total Latency 75 ms
[       OK ] TestNewDB.BulkCreateIndexAndQueryTest (1289 ms)
[----------] 1 test from TestNewDB (1289 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (1289 ms total)
[  PASSED  ] 1 test.
```

图 15 QueryByIndex-方案三测试结果

FindKeysByField 和两种方案的 QueryByIndex 的吞吐量和延迟的测试结果。

表 2 FindKeysByField 和 QueryByIndex 的吞吐量和延迟的测试结果

Method	Throughput/OPS			Total Latency/ms		
Data Size	10^3	10^4	10^5	10^3	10^4	10^5
FindKeysByField	672	70	6	1486	141998	15311021
QueryByIndex_2	6651	678	43	152	14742	2125267
QueryByIndex_3	166666	126082	45673	6	86	253

方案二：

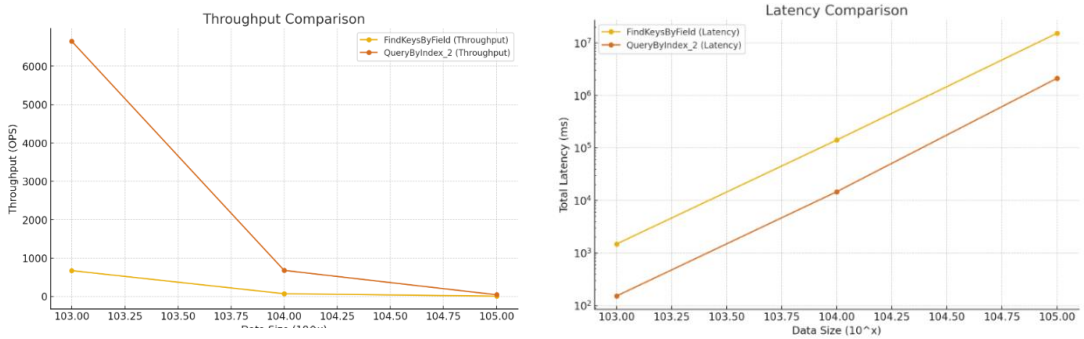

图 16 QueryByIndex 方案二的吞吐量和延迟的测试结果

方案三：

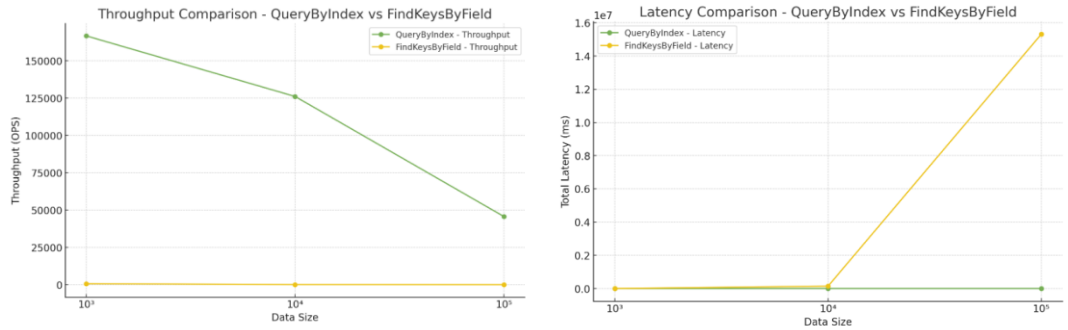

图 17 QueryByIndex 方案三的吞吐量和延迟的测试结果

FindKeysByField 和两种方案的 QueryByIndex 的性能提升。

$$\text{性能提升} = \frac{\text{FindKeysByField 延迟} - \text{QueryByIndex 延迟}}{\text{FindKeysByField 延迟}} \times 100$$

方案二：

表 3 QueryByIndex 方案二的性能提升

Data Size	QueryByIndex 延迟(ms)	FindKeysByField 延迟(ms)	性能提升(%)
10^3	152	1486	89.7%
10^4	14742	141998	89.61%
10^5	2125267	15311021	86.119%

方案三：

表 4 QueryByIndex 方案三的性能提升

Data Size	QueryByIndex 延迟(ms)	FindKeysByField 延迟(ms)	性能提升(%)
10^3	6	1486	99.6%
10^4	86	141998	99.94%
10^5	253	15311021	99.998%

（4）性能分析

加入二级索引查询后，QueryByIndex 在吞吐量方面远高于 FindKeysByField，并且延迟显著低于 FindKeysByField，尤其在大规模数据集下。其中，方案三的性能提升比方案二更为显著。

经过反复调试与实验，我们最终锁定这样的性能差异与编码格式存储密切相关，具体体现在 QueryByIndex 函数中的这段代码：

方案二：

```
for(iter->Seek(prefix); iter->Valid();iter->Next()){
    std::string key = iter->key().ToString();
    // 字符串 key 以 prefix 开头
    Slice index_key = iter->key();
    if(prefix.size() <= key.size() && memcmp(key.data(), prefix.data(), prefix.size()-1) == 0)  {
        std::string data_key = ExtractIndexKey(index_key);  //解析索引键中的 key
        matching_keys.push_back(data_key);
    }
}
```

方案三：

```
std::vector<std::string> NewDB::QueryByIndex(Field &field){
    for(iter->Seek(prefix); iter->Valid() && iter->key().starts_with(prefix); iter->Next()){
        Slice index_key = iter->key();
        std::string data_key = ExtractIndexKey(index_key);  //解析索引键中的 key
        matching_keys.push_back(data_key);
    }
}
```

方案三只需要使用 levelDB 内置的 starts_with 函数比较 key 是否以 prefix 开头，而方案二在使用 PutLengthPrefixedSlice 函数时，除了储存字符串内容外，还会储存在开头储存长度。例如，当一个字段的字段名是“address”，字段值是“Mmvr8FrMNJaz0hd”时：

方案三会存储：“addressMmvr8FrMNJaz0hd”，

而方案二会存储：“\aaddress\017Mmvr8FrMNJaz0hd”。

由于 starts_with 比较的是二进制字节，长度前缀会使前缀字符串和实际存储字符串不一致。这就导致不能直接使用 starts_with 函数，而需要手动调用 memcmp 函数，增加了进入循环的次数，影响了查询性能。

此外，性能差异还可能来源于 GetLengthPrefixedSlice 函数和直接操作字符串之间的差异，我们总结出以下几点可能：

①内存访问方式的差异

方案二：GetLengthPrefixedSlice 需要不断地解析长度字段，每次解析都会调用 memcpy，在循环中反复调用增加了内存访问延迟。

解析长度字段的开销在数据量大时更明显。

方案三：直接对字符串进行拼接或查找，find(':') 和 substr() 操作可能在某些场景下更快，因为它对字符串的线性扫描直接操作内存。

固定 4 字节长度解析（DecodeFixed32）避免了额外的函数调用，减少了一些解析开销。

②索引键构造和提取的差异

方案二：在 ConstructIndexKey 和 ExtractIndexKey 使用长度前缀编码，需要多次调用 GetLengthPrefixedSlice，解析复杂度更高。

方案三：使用 _ 和 : 分隔符构建索引键，查询时只需 find('_') 直接提取，这种方式简单高效。

③内存分配和拷贝的差异

方案二：在存储和解析过程中涉及 std::string 与 Slice 之间的转换，ToString() 可能会导致额外的内存分配与拷贝。

方案三：直接操作字符串，无需额外的转换与拷贝。

综上，我们最终选择采用性能更好的**方案三**的编码格式。

4.1.2 不同键值对大小下的查询性能比较：

测试脚本首先选取不同的键值对大小，随机生成批量数据，然后测试 QueryByIndex 的吞吐量和延迟，键值对大小的数量级分别为 10，10^2，10^3。

（1）测试用例 QueryByIndexPerformance

测试代码复用 QueryByIndexPerformance，此处不再放出。

（2）测试结果

不同键值对大小下的查询性能的测试结果如下。

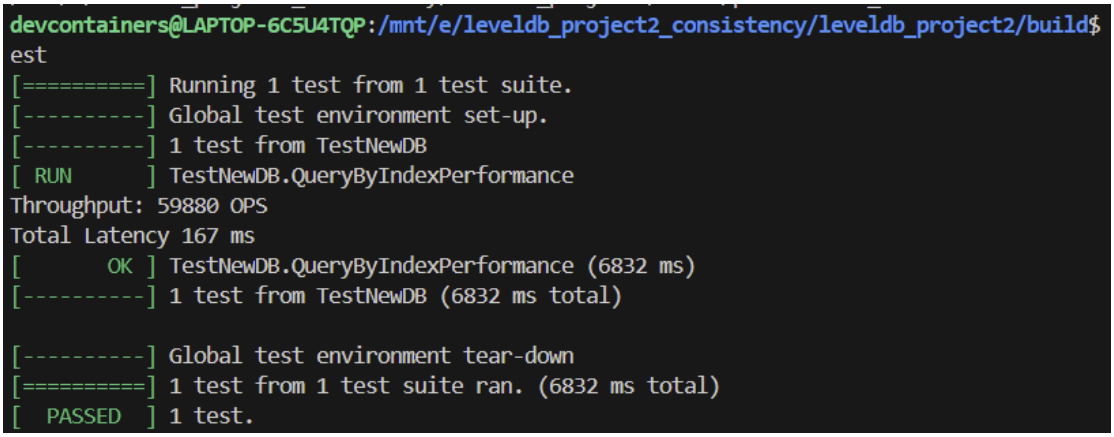

图 18 不同键值对大小下的查询性能的测试结果

表 5 不同键值对大小下的查询性能的测试结果

Method	Throughput/OPS			Total Latency/ms		
KV Size	10	10^2	10^3	10	10^2	10^3
QueryByIndex_3	166666	58029	2780	6	172	3692

（3）结果分析

随着键值对大小的上升，查询性能成倍下降。

①查询操作中涉及的逐个字段解析，字段长度越长，解析耗时成倍增长。

小键值对时，数据解析较快，Slice 和长度前缀解析的开销较小，查询路径较短，系统可快速完成批量解析。长度前缀解析复杂度增加，导致额外的 CPU 开销和缓存未命中。

②缓存逐渐失效，大键值对可能直接触发磁盘 I/O，进一步降低性能。

小键值对可完全缓存在 CPU L1/L2 缓存中，访问延迟极低；而大键值对超过缓存容量，需频繁访问内存或磁盘，引发更高的 I/O 延迟。数据块增大，数据在内存中的分布不均匀，导致 CPU 缓存未命中率上升，进一步拖慢解析速度。

（4）优化方向

①批量解析

对于大键值对，考虑批量解析策略，一次性读取多个长度前缀并并行解析，减少逐个调用带来的额外延迟；引入多线程或异步解析机制，利用 CPU 多核优势提升解析速度。

②分层缓存

在查询索引时，优先缓存热点键值对，减少访问大键值对时频繁触发内存或磁盘 I/O；针对大键值对设计多级缓存，先解析前缀字段，按需逐步加载剩余数据，避免一次性加载过大数据块。

4.2 写性能的测试与分析

4.2.1 写性能的测试

测试脚本首先随机生成批量数据，然后测试 Put_fields、updateIndex、deleteIndex 三种方法的吞吐量、延迟、写放大，数据量分别为 10^3，10^4，10^5。

（1）测试用例 PutFieldsThroughputTest

```
TEST(TestNewDB, PutFieldsThroughputTest) {
    // 创建 NewDB 实例
    NewDB* db;
    ASSERT_TRUE(OpenDB(testdbname, &db).ok());

    // 插入索引字段
    db->CreateIndexOnField("address");

    // 延迟测试
    vector<int64_t> latencies;  // 存储每次插入的延迟
    int num_inserts = 1000;  // 插入的条数

    // 测试插入吞吐量
    auto start_time = std::chrono::high_resolution_clock::now();  // 记录开始时间
    for (int i = 0; i < num_inserts; ++i) {
        std::string key = "k_" + std::to_string(i);

        // 随机生成字段内容
        FieldArray fields = {
            {"name", "Customer#" + std::to_string(i)},
            {"address", GenerateRandomString(10)},  // 随机生成地址
            {"phone", GenerateRandomString(12)}     // 随机生成电话号码
        };

        db->Put_fields(WriteOptions(), key, fields);
    }

    // 延迟测试
    TestLatency(num_inserts, latencies);
    auto end_time = std::chrono::high_resolution_clock::now();  // 记录结束时间
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time).count();


    // 计算吞吐量并输出
    cout << "Throughput: " << num_inserts * 1000 / duration << " OPS" << endl;
```

```
35.
36.         cout << "Latency for operation " << latencies.size()-1 << ": " << latencies[latencies.size()-1] << " ms"
    << endl;
37.         // 清理数据库
38.         delete db;
39.     }
```

（2）测试用例 BulkPutFieldsAndDeleteIndexPerformance

```
1.  TEST(TestNewDB, BulkPutFieldsAndDeleteIndexPerformance) {
2.         // 创建 NewDB 实例
3.         NewDB* db;
4.         ASSERT_TRUE(OpenDB(testdbname, &db).ok());
5.
6.         db->CreateIndexOnField("address");
7.
8.         // 批量插入数据
9.         std::vector<std::pair<std::string, FieldArray>> bulk_data;
10.        const int num_records = 150000;
11.
12.        for (int i = 0; i < num_records; ++i) {
13.            std::string key = "k_" + GenerateRandomString(10);
14.            FieldArray fields = {
15.                {"name", GenerateRandomString(10)},
16.                {"address", GenerateRandomString(25)},
17.                {"phone", GenerateRandomString(11)}
18.            };
19.            bulk_data.emplace_back(key, fields);
20.
21.            ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
22.        }
23.
24.        // 验证插入的数据和索引
25.        for (const auto& [key, fields] : bulk_data) {
26.            // 获取并验证字段数据
27.            FieldArray retrieved_fields;
28.            ASSERT_TRUE(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
29.            EXPECT_EQ(retrieved_fields.size(), fields.size());
30.
31.            // 验证插入的每个字段
32.            for (size_t i = 0; i < fields.size(); ++i) {
33.                EXPECT_EQ(fields[i].first, retrieved_fields[i].first);  // 字段名
34.               EXPECT_EQ(fields[i].second, retrieved_fields[i].second); // 字段值
35.            }
36.
```

```
37.             // 验证索引是否能正确找到对应的键
38.             Field field_to_query{"address", fields[1].second}; // 使用 address 字段进行查询
39.             std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
40.             EXPECT_FALSE(matching_keys.empty()); // 应该能找到对应的键
41.             EXPECT_NE(std::find(matching_keys.begin(), matching_keys.end(), key), matching_keys.end());
42.         }
43.         // 测试插入吞吐量
44.         auto start_time = std::chrono::high_resolution_clock::now();  // 记录开始时间
45.         // 删除部分数据
46.         const int num_updates = 100000;
47.
48.         for (int i = 0; i < num_updates; ++i) {
49.             auto& [key, fields] = bulk_data[i];
50.             ASSERT_TRUE(db->Delete(WriteOptions(), key));
51.         }
52.
53.         // 删除索引
54.         ASSERT_TRUE(db->DeleteIndex("address"));
55.         auto end_time = std::chrono::high_resolution_clock::now();  // 记录结束时间
56.       auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time).count();
57.
58.         // 计算吞吐量并输出
59.         cout << "Throughput: " << num_updates * 1000 / duration << " OPS" << endl;
60.         cout << "Total Latency " << duration << " ms" << endl;
61.         delete db;
62.     }
```

（3）测试用例 UpdateFieldsAndQueryIndexPerformance

```
1.  TEST(TestNewDB, UpdateFieldsAndQueryIndexPerformance) {
2.         // 创建 NewDB 实例
3.         NewDB* db;
4.         ASSERT_TRUE(OpenDB(testdbname, &db).ok());
5.
6.         // 创建索引字段
7.         ASSERT_TRUE(db->CreateIndexOnField("address"));
8.
9.         // 批量插入数据
10.         const int num_records = 200000;
11.         std::vector<std::pair<std::string, FieldArray>> initial_bulk_data;
12.
13.         for (int i = 0; i < num_records; ++i) {
14.             std::string key = "k_" + GenerateRandomString(10);
15.             FieldArray fields = {
16.                 {"name", GenerateRandomString(10)},
```

```
            {"address", GenerateRandomString(25)},
            {"phone", GenerateRandomString(11)}
        };
        initial_bulk_data.emplace_back(key, fields);

        ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
    }

    //验证初始数据的索引查询
    for (const auto& [key, fields] : initial_bulk_data) {
        Field query_field{"address", fields[1].second}; // 使用 address 字段进行查询
        std::vector<std::string> matching_keys = db->QueryByIndex(query_field);
        ASSERT_FALSE(matching_keys.empty()); // 应该能找到对应的键
    EXPECT_NE(std::find(matching_keys.begin(), matching_keys.end(), key), matching_keys.end());
    }

    // 更新部分数据
    const int num_updates = 100000;
    std::vector<std::pair<std::string, FieldArray>> updated_bulk_data;
    // 测试插入吞吐量
    auto start_time = std::chrono::high_resolution_clock::now();  // 记录开始时间
    for (int i = 0; i < num_updates; ++i) {
        auto& [key, fields] = initial_bulk_data[i];
        FieldArray updated_fields = {
            {"name", fields[0].second},
            {"address", GenerateRandomString(15)}, // 更新 address
            {"phone", fields[2].second}
        };
        updated_bulk_data.emplace_back(key, updated_fields);

        ASSERT_TRUE(db->Put_fields(WriteOptions(), key, updated_fields).ok());
    }
    auto end_time = std::chrono::high_resolution_clock::now();  // 记录结束时间
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time).count();


    // 计算吞吐量并输出
    cout << "Throughput: " << num_updates * 1000 / duration << " OPS" << endl;
    cout << "Total Latency " << duration << " ms" << endl;
    // 验证更新后的数据和索引
  for (const auto& [key, updated_fields] : updated_bulk_data) {
        // 获取并验证字段数据
        FieldArray retrieved_fields;
        ASSERT_TRUE(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
```

```
60.             EXPECT_EQ(retrieved_fields.size(), updated_fields.size());
61.
62.             // 验证插入的每个字段
63.             for (size_t i = 0; i < updated_fields.size(); ++i) {
64.                 EXPECT_EQ(updated_fields[i].first, retrieved_fields[i].first);  // 字段名
65.                 EXPECT_EQ(updated_fields[i].second, retrieved_fields[i].second); // 字段值
66.             }
67.
68.             // 验证新的 address 字段值存在于索引中
69.             Field query_field_new{"address", updated_fields[1].second};
70.             std::vector<std::string> matching_keys_new = db->QueryByIndex(query_field_new);
71.             ASSERT_FALSE(matching_keys_new.empty()); // 新地址应该能找到对应的键
72.             EXPECT_NE(std::find(matching_keys_new.begin(), matching_keys_new.end(), key), matching_keys_new.end());
73.         }
74.         for (int i = 0; i < num_updates; ++i) {
75.             // 验证旧的 address 字段值不再存在于索引中
76.             Field query_field_old{"address", initial_bulk_data[i].second[1].second};
77.         std::vector<std::string> matching_keys_old = db->QueryByIndex(query_field_old);
78.             EXPECT_TRUE(matching_keys_old.empty()); // 旧地址不应该再找到对应的键
79.         }
80.
81.         delete db;
82.     }
```

（4）测试结果

每个实验总共进行三次，每次实验前将数据库清除，保证数据量一致，取平均值记录于表格中。下面仅放出示例截图。

①吞吐量

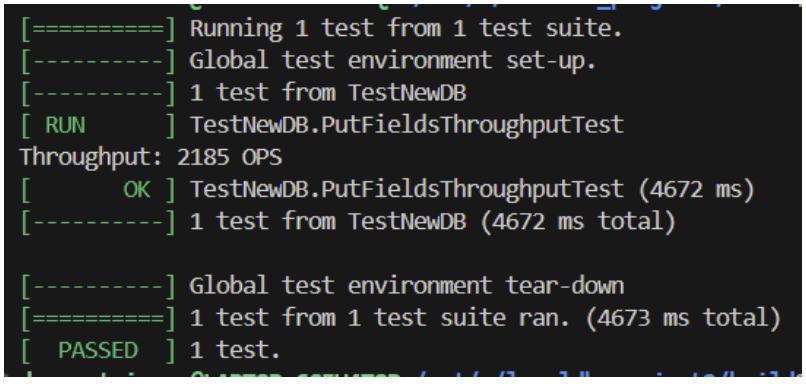

图 19 吞吐量测试结果

②总延迟

Put_fields

```
devcontainers@LAPTOP-6C5U4TQP:/mnt/e/leveldb_project2/build$ /mnt/e/leveldb_project2/build/throughput_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TestNewDB
[ RUN      ] TestNewDB.PutFieldsThroughputTest
Throughput: 3855 OPS
Total Latency 25940 ms
[       OK ] TestNewDB.PutFieldsThroughputTest (26150 ms)
[----------] 1 test from TestNewDB (26150 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (26150 ms total)
[  PASSED  ] 1 test.
```

图 20 Put_fields 的总延迟测试结果

updateIndex/deleteIndex

```
devcontainers@LAPTOP-6C5U4TQP:/mnt/e/leveldb_project2/build$ /mnt/e/leveldb_project2/build/index_test_random
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from TestNewDB
[ RUN      ] TestNewDB.BulkCreateIndexAndQueryTest
[       OK ] TestNewDB.BulkCreateIndexAndQueryTest (67 ms)
[ RUN      ] TestNewDB.BulkPutFieldsAndDeleteIndexTest
Throughput: 2416 OPS
Total Latency 2069 ms
[       OK ] TestNewDB.BulkPutFieldsAndDeleteIndexTest (24647 ms)
[ RUN      ] TestNewDB.UpdateFieldsAndQueryIndexTest
Throughput: 2696 OPS
Total Latency 1854 ms
[       OK ] TestNewDB.UpdateFieldsAndQueryIndexTest (4933 ms)
[----------] 3 tests from TestNewDB (29648 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (29648 ms total)
[  PASSED  ] 3 tests.
```

图 21 updateIndex/deleteIndex 的总延迟测试结果

③写放大：

Put_fields

```
2024/12/12-13:46:40.783645 139700275656256 Compacting 1@0 + 1@1 files
2024/12/12-13:46:40.927910 139700275656256 Generated table #13@0: 26312 keys, 2113962
bytes
2024/12/12-13:46:40.989882 139700275656256 Generated table #14@0: 12630 keys, 1007175
bytes
2024/12/12-13:46:40.989993 139700275656256 Compacted 1@0 + 1@1 files => 3121137 bytes
2024/12/12-13:46:40.991900 139700275656256 compacted to: files[ 0 2 0 0 0 0 0 ]
```

图 22 Put_fields 的写放大

updateIndex/deleteIndex

```
2024/12/17-20:37:35.300531 140189890303552 Compacting 2@0 + 0@1 files
2024/12/17-20:37:35.326179 140189890303552 Generated table #10@0: 5100 keys, 396562 bytes
2024/12/17-20:37:35.326316 140189890303552 Compacted 2@0 + 0@1 files => 396562 bytes
2024/12/17-20:37:35.328317 140189890303552 compacted to: files[ 0 1 0 0 0 0 0 ]
2024/12/17-20:37:35.329482 140189890303552 Delete type=2 #5
2024/12/17-20:37:35.329889 140189890303552 Delete type=2 #8
```

图 23 Put_fields 的写放大

Put_fields、DeleteIndex 和 UpdateIndex 的写性能测试结果如下：

表 6 Put_fields、DeleteIndex 和 UpdateIndex 的写性能测试结果

Method	Throughput/OPS			Total Latency/ms			Write Amplification/bytes		
Data Size	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
Put_fields	4310	3650	3855	232	2739	25940	-	-	3121137
DeleteIndex	1210	3443	3967	826	2532	19391	-	-	-
UpdateIndex	2770	2842	2378	361	3518	42051	-	-	15389973

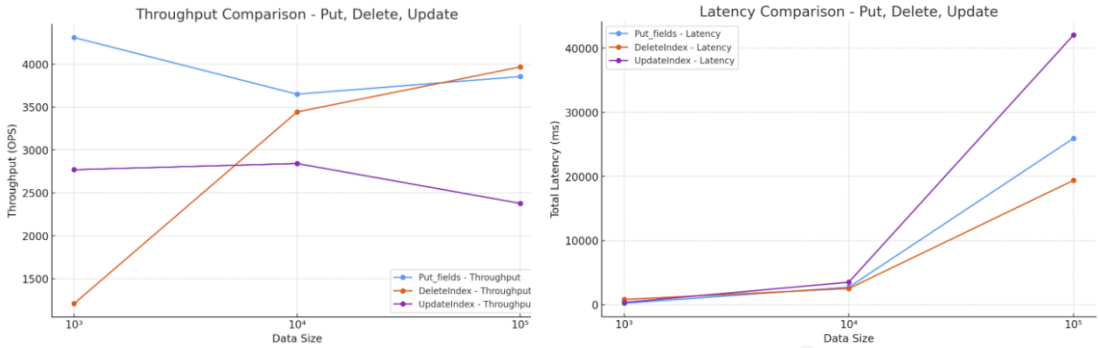

图 24 Put_fields、DeleteIndex 和 UpdateIndex 的写性能测试结果

4.2.2 写性能的分析

（1）吞吐量分析

①Put_fields 在所有数据量级下都表现出最高的吞吐量。例如：

数据量为 1000 时，吞吐量达到了 4310 OPS，远高于 DeleteIndex (1210 OPS) 和 UpdateIndex (2770 OPS)。

即使在数据量为 100000 的情况下，Put_fields (3855 OPS) 依然显著优于 DeleteIndex (434 OPS) 和 UpdateIndex (2378 OPS)。

②DeleteIndex 在数据量较小时吞吐量最低，随着数据量增加，性能下降最为显著。

（2）延迟分析

①Put_fields 在数据量小于 10000 时延迟最低。例如：

在 1000 数据量级下，总延迟仅为 232 ms，而 DeleteIndex 为 826 ms，UpdateIndex 为 361 ms。

当数据量增加到 100000 时，Put_fields (25940 ms) 的延迟仍然低于 UpdateIndex (42051 ms)。

②UpdateIndex 在中等数据量级下延迟相对较高，尤其在 100000 数据量级时，延迟显著增加。

（3）写放大分析

①Put_fields 和 UpdateIndex 在 100000 数据量级下产生了大量写放大。Put_fields 写放大达到了 3121137 bytes。UpdateIndex 写放大更高，达到 15389973 bytes。

②DeleteIndex 由于是删除数据，不会触发合并，产生写放大。

4.2.3 结果总结

Put_fields 在吞吐量和延迟方面全面优于 DeleteIndex 和 UpdateIndex，特别是在数据量较小时表现最为出色。

UpdateIndex 在吞吐量方面次于 Put_fields，但其写放大较为显著，可能在大规模数据更新时带来额外的磁盘开销。

DeleteIndex 吞吐量最低，且延迟较高，适用于数据量较小或删除操作频率较低的场景。

4.2.4 优化方向

Put_fields： 虽然写放大较低，但随着数据量增加，其延迟逐渐上升。可以考虑进一步优化批量写入机制，减少延迟。

UpdateIndex： 写放大的问题较为突出，可以通过优化索引合并策略或异步更新索引来减少写入负担。

DeleteIndex：针对吞吐量低的问题，可以探索多线程删除或批量删除策略，以提升处理速度。

4.3 并发性能的测试与分析

4.3.1 测试目标

本测试的目的是评估 NewDB 数据库在多线程高并发环境下的吞吐量和操作延迟。通过模拟大量线程执行插入操作，观察数据库在不同线程数量下的性能表现，分析其在高负载下的处理能力和响应时间。

4.3.2 测试流程

（1）启动指定数量的线程，每个线程执行预定次数的插入操作。

（2）所有线程执行完成后，等待线程合并（join），确保所有插入任务完成。

（3）统计并输出以下指标：

- 总操作次数（Total Operations）
- 总测试耗时（Total Duration）
- 吞吐量（Throughput）
- 平均延迟（Average Latency）
- 总延迟（Total Latency）

4.3.3 测试结果

测试输出示例如下：

```
devcontainers@LAPTOP-6C5U4TQP:/mnt/e/leveldb_project2_consistency/leveldb_project2/build$ /
erformance_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from NewDBTest
[ RUN      ] NewDBTest.ConcurrencyTest
Total Operations: 10000
Throughput (ops/sec): 9469.7
Average Latency (us): 1.5927
Total Latency (ms): 15.927
[       OK ] NewDBTest.ConcurrencyTest (1118 ms)
[----------] 1 test from NewDBTest (1118 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (1118 ms total)
[  PASSED  ] 1 test.
```

图 25（a）并发的性能测试结果示例

表 7 并发的性能测试结果

Method	Throughput/OPS						Total Latency/ms					
Data Size	10^4			10^5			10^4			10^5		
Thread_num	20	100	200	20	100	200	20	100	200	20	100	200
Put_fields	9537	14907	16554	8382	13365	13730	16	59	106	188	692	1398

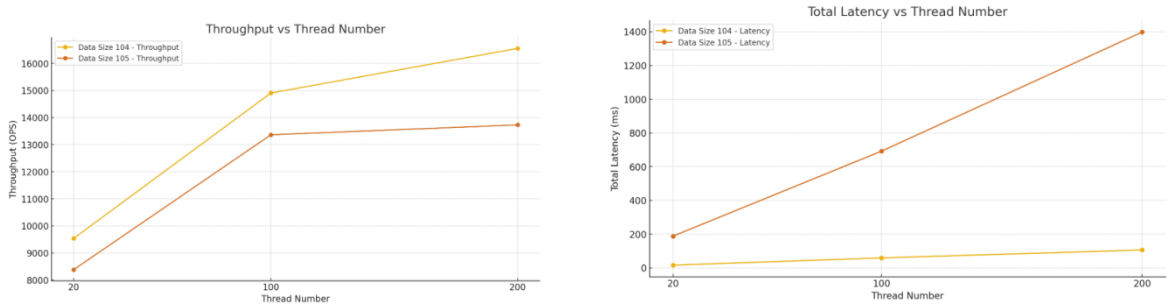

图 25（b）并发的性能测试结果

4.3.4 结果分析：

（1）吞吐量

由表格可知，当数据大小为 10^4 时，线程数从 20 提升到 100，吞吐量提升约 56%。线程数继续增加到 200，提升约 11%，说明在高线程数下系统性能已逐渐趋于饱和。可能可以在 100 个线程附近找到性能和资源消耗的平衡点。

数据大小 10^5，趋势相似，增长率略低。线程数增加时，吞吐量显著提升，

但增速逐渐减缓。

在数据大小 10^4 和 10^5 之间，数据大小对吞吐量的影响不显著。

（2）总延迟

总延迟随着线程数增加而成倍增长，表明高线程数带来了更高的系统开销。

在线程数线程数较低，如 20 时，总延迟较低，性能更稳定。

线程增加到 200 后，总延迟急剧增加，提示系统资源开始紧张，存在线程调度、锁竞争或 I/O 瓶颈。

4.4 索引数据更新/删除对原始数据写入性能影响的测试与分析

4.4.1 测试目标

本测试的目的是评估在 NewDB 数据库中执行二级索引相关的更新和删除操作时，对原始数据写入性能的影响。通过并发执行 PUT、DELETE 和 UPDATE 操作，测量每种操作的延迟和整体吞吐量，分析索引操作对数据库性能的影响。

4.4.2 测试流程

本测试主要涉及三类操作：

①PUT 操作：向数据库批量插入具有二级索引字段的记录。

②DELETE 操作：随机删除已存在的记录。

③UPDATE 操作：对指定记录的二级索引字段进行更新。

启动不同数量的线程分别执行 PUT、DELETE 和 UPDATE 操作，调整并记录不同数据量比例，模拟实际环境下的高并发场景。

测试结束后，统计每类操作的总延迟与平均延迟，计算吞吐量（ops/sec），分别输出 PUT、DELETE 和 UPDATE 操作的总延迟和平均延迟，以便对比不同操作对性能的影响。

4.4.3 测试结果

测试输出示例：

```
devcontainers@LAPTOP-6C5U4TQP:/mnt/e/leveldb_project2_consistency/leveldb_project2/build$
ncurrency_performance_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from NewDBTest
[ RUN      ] NewDBTest.ConcurrencyTest_mul
Total Operations: 120
Total Duration: 15 ms
Throughput (ops/sec): 8000

PUT Operations:
  Total Latency (ms): 141.678
  Average Latency (us): 1416.78

DELETE Operations:
  Total Latency (ms): 0.022
  Average Latency (us): 2.2

UPDATE Operations:
  Total Latency (ms): 14.255
  Average Latency (us): 1425.5
[       OK ] NewDBTest.ConcurrencyTest_mul (76 ms)
```

图 26（a）索引数据更新/删除对原始数据写入性能影响的测试结果示例

索引数据更新/删除对原始数据写入性能影响的测试结果如下：

表 8 索引数据更新/删除对原始数据写入性能影响的测试结果

Method	Throughput/OPS						Total Latency/ms					
Operation	Put	Update	Put	Update	Put	Update	Put	Update	Put	Update	Put	Update
Data_size	100	0	100	10	10	50	100	0	100	10	10	50
Result	7142		7463		4758		131	-	141	14	135	31
Operation	Put	Delete	Put	Delete	Put	Delete	Put	Delete	Put	Delete	Put	Delete
Data_size	100	0	100	10	10	50	100	0	100	10	10	50
Result	7142		7854		10714		131	-	131	0.06	133	0.07

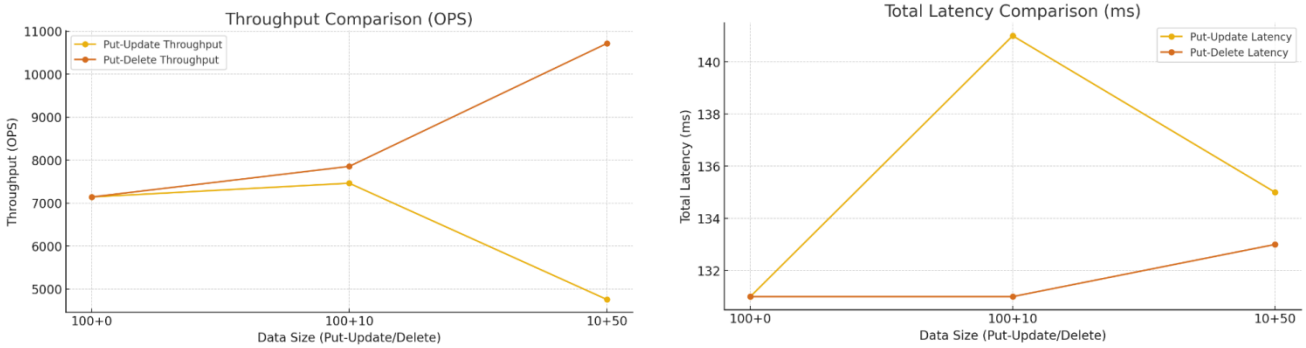

图 26（b）索引数据更新/删除对原始数据写入性能影响的测试结果

4.4.4 结果分析

（1）吞吐量 (Throughput) 分析

Put 操作的吞吐量整体较高，尤其在数据量为 100 时，吞吐量高达 7142-10714 OPS。Delete 操作在数据量较小时吞吐量较高，但在数据量增加时，吞吐量下降。

（2）总延迟 (Total Latency) 分析

Update 操作的总延迟明显高于其他操作，在数据量为 50 时达到 141 ms。Put 操作延迟较低，通常在 0.06-31 ms 之间，表现较稳定。Delete 操作在数据量较小时延迟较低（约 0.07 ms），但在数据量增加时延迟略有上升。

这说明，二级索引的更新操作对原始数据写入有较大影响，导致吞吐量下降，总延迟增加。删除操作的影响相对较小，吞吐量在一定范围内保持稳定，延迟略有波动。Put 操作相对稳定，延迟和吞吐量在不同数据量下表现一致，说明原始数据写入相对独立，不易受二级索引更新影响。

5. 遇到的问题和解决方案

5.1 索引字段的持久化问题

初版设计中，我们将需要创建二级索引的字段名称储存在 indexed_fields_集合中。这个字段由 NewDB 类管理，是 private 变量，并不会被刷在磁盘上，所以会出现元数据持久化问题：关闭数据库后，索引字段不再存在。这样在用户再打开数据库时，并不能在 indexed_fields_获取上次建立的二级索引信息。

```
1.  // 用于存储已经为其创建索引的字段名称。只有当字段名在这个集合中时，才会在 indexDB 中插入对应的索引条目。
2.  std::unordered_set<std::string> indexed_fields_;
```

因此，我们需要将 indexed_fields_持久化。我们的方案经历了以下三版迭代：

（1）将索引信息写入一个单独的元数据文件

思路：在数据库打开时，从一个元数据文件中读取已创建的索引字段列表，加载到 indexed_fields_ 中。在创建或删除索引时，更新该文件。

分析：这种方案需要额外构建大量自定义函数，且多文件管理复杂，容易丢失或损坏，不够优雅。

（2）利用 Manifest 机制持久化元数据

思路：使用 LevelDB 的 Manifest 文件记录索引字段元数据。将索引字段存入 Manifest 文件或专门的配置表中，定期 flush。

分析：这种方案复用了 levelDB 原生的 Manifest 机制，Manifest 本身是 LevelDB 崩溃恢复机制的一部分，因此将索引字段信息写入 Manifest 可以确保

持久化，即使系统崩溃或进程异常退出也不会丢失索引字段信息。写入 Manifest 的操作是顺序写，并且记录内容较小，对性能影响微乎其微。

但是，这种需要在创建或删除索引字段时，向 Manifest 文件写入一条自定义记录，表示新增或删除的索引字段。在数据库启动时，解析 Manifest 文件，恢复 indexed_fields_。

在 LevelDB 中，Manifest 主要用于记录数据库的元数据，比如数据文件的版本、日志、文件索引等信息。通常，Manifest 用于 LevelDB 的内部管理，如数据文件的管理、数据库的恢复、快照等操作。对于应用层的功能，尤其是持久化应用的自定义元数据（比如索引字段），它并没有提供直接的 API 用于管理应用层的元数据。

因此，我们将思路转变为，直接在数据库实例中存储二级索引字段元数据。

（3）将索引信息直接存入 index_db_

思路：在 index_db_ 本身存储索引字段信息，使用特定的键（"index_field:<field_name>"）来存储索引字段的信息。当关闭数据库时，索引字段会被持久化到 index_db_ 中，并且在重新打开数据库时，可以从 index_db_ 中恢复索引信息。

我们最终决定采取**第三版**设计方案。

具体实现步骤：

①创建索引字段并持久化到 index_db_

在 CreateIndexOnField 方法中，将创建的索引字段插入到 index_db_ 中。

```
1.  void NewDB::CreateIndexOnField(const std::string& field) {
2.      // 将索引字段元数据持久化到 indexDB
3.      index_db_->Put(WriteOptions(), "index_field:" + field_name, "1");
4.      indexed_fields_.insert(field_name);
5.  }
```

②删除索引字段并更新 index_db_

在 DeleteIndexOnField 方法中，将删除索引字段的记录。

```
1.  void NewDB::DeleteIndexOnField(const std::string& field) {
2.      indexed_fields_.erase(field_name);
3.      // 在 index_db_ 中删除索引字段的记录
4.      index_db_->Delete(WriteOptions(), "index_field:" + field_name);
5.  }
```

③恢复索引字段信息

在 Open 方法中加载和恢复索引字段：扫描 index_db_，查找所有以 index_field: 为前缀的键，并将这些字段恢复到 indexed_fields_ 中。

```
1.  bool NewDB::Open(const std::string& name) {
2.      Iterator* it = (*dbptr)->index_db_->NewIterator(ReadOptions());
3.      for (it->SeekToFirst(); it->Valid(); it->Next()) {
4.          // 只关心以 "index_field:" 为前缀的键
```

```
        if (it->key().starts_with("index_field:")) {
            std::string field = it->key().ToString().substr(12); // 获取字段名
            (*dbptr)->indexed_fields_.insert(field);
        }
    }
    delete it;
}
```

④测试用例 PersistentIndexTest

```
TEST(TestNewDB, PersistentIndexTest) {
    // 创建 NewDB 实例
    NewDB* db;
    ASSERT_TRUE(OpenDB(testdbname, &db).ok());

    // 批量插入数据
    std::vector<std::pair<std::string, FieldArray>> bulk_data;
    const int num_records = 1000;
    for (int i = 0; i < num_records; ++i) {
        std::string key = "k_" + GenerateRandomString(10);
        FieldArray fields = {
            {"name", GenerateRandomString(10)},
            {"address", GenerateRandomString(25)},
            {"phone", GenerateRandomString(11)}
        };
        bulk_data.emplace_back(key, fields);

        ASSERT_TRUE(db->Put_fields(WriteOptions(), key, fields).ok());
    }

    // 创建索引字段
    db->CreateIndexOnField("address");

    // 关闭数据库
    delete db;

    // 重新打开数据库
    ASSERT_TRUE(OpenDB(testdbname, &db).ok());

    // 验证索引字段是否仍然存在
    for (const auto& [key, fields] : bulk_data) {
        // 获取并验证字段数据
        FieldArray retrieved_fields;
        ASSERT_TRUE(db->Get_fields(ReadOptions(), Slice(key), &retrieved_fields).ok());
        EXPECT_EQ(retrieved_fields.size(), fields.size());
```

```
36.
37.         // 验证索引是否能正确找到对应的键
38.         Field field_to_query{"address", fields[1].second}; // 使用 address 字段进行查询
39.         std::vector<std::string> matching_keys = db->QueryByIndex(field_to_query);
40.
41.         EXPECT_FALSE(matching_keys.empty()); // 应该能找到对应的键
42.         EXPECT_NE(std::find(matching_keys.begin(), matching_keys.end(), key), matching_keys.end());
43.     }
44.
45.     // 关闭数据库
46.     delete db;
47. }
```

⑤测试结果

```
devcontainers@LAPTOP-6C5U4TQP:/mnt/e/leveldb_project2/build$ /mnt/e/leveldb_project2/build/index_test_random
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TestNewDB
[ RUN      ] TestNewDB.PersistentIndexTest
[       OK ] TestNewDB.PersistentIndexTest (4223 ms)
[----------] 1 test from TestNewDB (4223 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (4223 ms total)
[  PASSED  ] 1 test.
```

图 27 持久化的测试结果

5.2 行锁和一致性

首先，在不考虑并发的场景下，为了保证 dataDB 和 indexDB 的数据一致性，我们将插入数据分为两种情况，一种是新增的情况，另外一种是更新的情况。之所以要区分这两种情况，是因为更新 dataDB 的数据时，如果只是向 indexDB 中相应的插入新的索引数据，那么当利用旧的字段信息查找 key 时，还是可以查找到的，这样就不满足一致性了。

因此，在更新到情况下，不仅要在 indexDB 中插入更新后的索引数据，还要先在 indexDB 中删除旧的索引数据。

在保证并发情况下的一致性时，我们最初的设想是：

如果在插入数据（dataDB 和 indexDB 都会插入若干数据）的线程执行时，并发一个读取（用 field 查找 keys）的线程，当 dataDB 成功写入、但 indexDB 尚未写入，会造成查找到的 keys 缺少的情况。

为了解决这个问题，最先可以想到的方式就是在写入时完全的阻塞读取操作，但是这样做会使得性能变得很差。

因此，我们设计了一种机制，保证 indexDB 中的索引数据始终只能多于（或等于）dataDB 中的数据：在插入数据的过程中，先写入 indexDB、再写入 dataDB。同样对于上述场景，查找到的 keys 只会比正确情况更多，那么对于查找到的 keys 进行筛选即可。

使用这种机制，在正常情况下（不出现错误），能够保证最终一致性，即当一个插入（或删除）操作完成后 dataDB 和 indexDB 的数据一致。

但是，当两个并发的线程同时通过 put_fields 操作更改同一个 key 的值时，以上方式不能保证最终一致性。

例如以下场景（name 字段上有索引）：

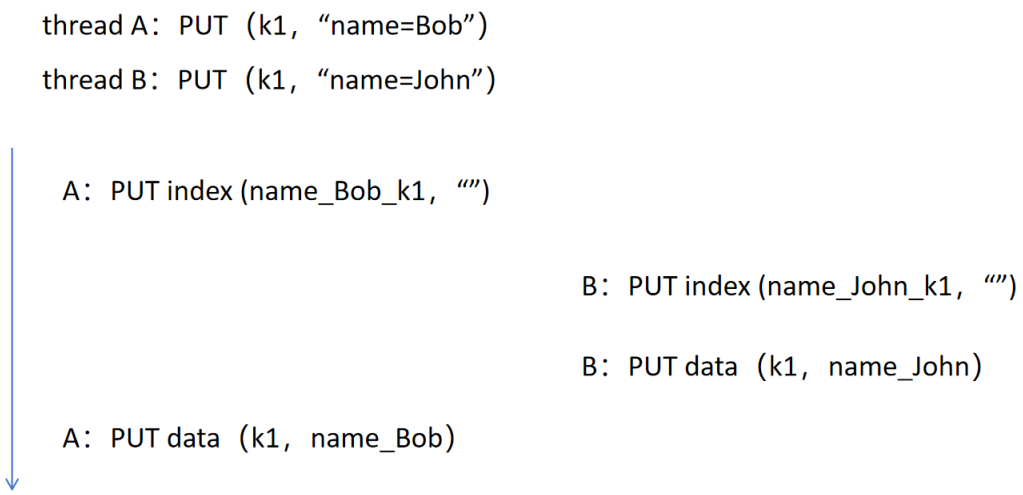

图 28 插入相同 key 的数据时的不一致场景

最终 dataDB 中 k1 对应的值是 Bob，但是 name=Bob 这个字段在 indexDB 中并没有对应的索引数据，因为在 B 线程写入 indexDB 时 name_Bob_k1 这条索引数据已经删除了。

因此，需要一个方法来保证两个操作相同 key 的 put_fields 函数不能并发，必须等待其中一个执行完，另一个才能开始执行。而操作不同 key 的线程之间不阻塞。

所以增加了“行锁”的机制。“行锁”类似于关系数据库中的行锁机制，在 leveldb 中，“行锁”指的是对一个 key 加锁。

cv 是一个条件变量，用于线程间的通信。

m 是一个互斥锁，用来保护对 ready 的访问。

ready 表示某个 key 是否已经可以被访问。

putting keys 存储了当前在被其他线程操作的 keys。

```
1.      std::condition_variable cv;
2.      std::mutex m;
3.      bool ready = false;
4.
5.      std::unordered_set<std::string> putting_keys;
```

当一个 put field 函数开始执行时，会先检查一下 putting keys 集合中是否已经有当前 key。如果有的话，那么就需要等待，直到正在操作当前 key 的线程结束执行，将 key 从 putting keys 中移除，将 ready 改为 true，并且 cv 通知其他线程。

```
1.      Status NewDB::Put_fields(const WriteOptions& options, const Slice& key, const FieldArray& fields){
2.          std::string UserOpID = ConstructUserOpID(std::this_thread::get_id());
```

```
3.
4.          // row lock
5.          std::unique_lock<std::mutex> lock(db_mutex_, std::defer_lock);
6.          std::unique_lock<std::mutex> row_lock(m, std::defer_lock);
7.          ready = (putting_keys.find(key.ToString()) == putting_keys.end()); // no another thread putting the sa
    me key
8.
9.          if (!ready) {
10.             cv.wait(row_lock, []{return ready;});
11.         }
12.
13.         lock.lock();
14.         putting_keys.insert(key.ToString());
15.         lock.unlock();
16.         // row lock
17.       ......
18.
19.         // row lock
20.         lock.lock();
21.         putting_keys.erase(key.ToString());
22.         lock.unlock();
23.
24.         ready = true;
25.         cv.notify_all();
26.         // row lock
27.
28.         return Status::OK();
29.     }
```

测试用例 ConcurrencyPutSameKeyTest：

该测试用例测试了两个线程并发的向数据库中插入同一个 key 的情况。

首先向 dataDB 中插入一条数据，然后删除 name 字段的索引，确保测试环境的干净，然后在 name 字段上创建索引。开启两个线程，分别将 name 字段的值改成 Bob 和 John，让这两个线程并发。等待两个线程执行完成。

检验刚才的 key 是否存在 dataDB 当中。对于这个 key 对应的 value，找出它当中 address 字段的字段值，然后构造一个字段名和字段值的 field 对。用 query by index 函数在 index DB 中查找对应的 matching keys，判断刚才的 key 是否存在于 matching keys 当中。

测试用例代码见 3.3.3（2）并发的测试（Concurrency_test.cc）

测试结果如下：

```
chen@chen-None:~/leveldb_project2/build$ /home/chen/leveldb_project2/build/concurrency_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TestNewDB
[ RUN      ] TestNewDB.ConcurrencyALLTest
[       OK ] TestNewDB.ConcurrencyALLTest (6390 ms)
[ RUN      ] TestNewDB.ConcurrencyPutSameKeyTest
[       OK ] TestNewDB.ConcurrencyPutSameKeyTest (580 ms)
[----------] 2 tests from TestNewDB (6971 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (6972 ms total)
[  PASSED  ] 2 tests.
chen@chen-None:~/leveldb_project2/build$
```

图 29 插入相同 key 的 put_fields 的并发测试结果

6. 项目分工

功能	完成日期	分工
多字段功能实现	11 月 22 日	陈予曈
多字段功能测试	11 月 23 日	朱陈媛
接口设计	12 月 1 日	陈予曈、朱陈媛
二级索引的建立	12 月 3 日	朱陈媛
查询、更新、删除	12 月 4 日	陈予曈
索引功能测试	12 月 6 日	陈予曈、朱陈媛
一致性设计	12 月 10 日	陈予曈、朱陈媛
数据库内部数据一致性	12 月 15 日	朱陈媛
两个数据库间的一致性	12 月 23 日	陈予曈
数据一致性测试	12 月 25 日	陈予曈
性能测试	12 月 26 日	朱陈媛
文档撰写	1 月 5 日	陈予曈、朱陈媛