# 代码设计文档

## 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。

## 2.功能设计

### 2.1.字段设计

- 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。

具体而言，需要实现字段存储和字段查询的功能。

- 实现思路

1. 字段的编码格式设计

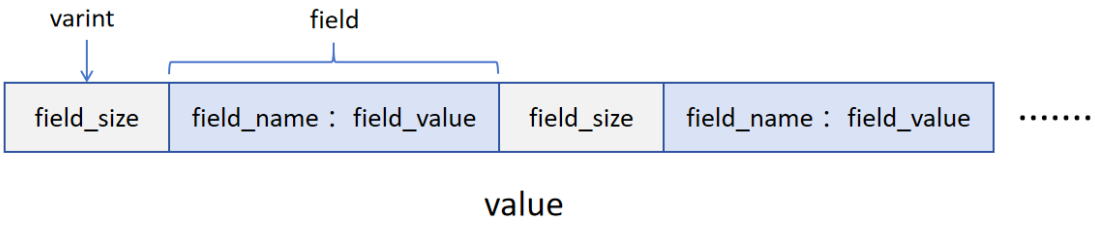
可选的编码格式：

(1) 定长：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) 变长，并记录长度：节省空间且灵活，但是需要先读取长度。

(3) 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。综合考虑，最终决定采用以下编码格式：




2. 功能实现

序列化函数：

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

std::string 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;
}
```

反序列化函数：

```
FieldArray 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(':');       //转义

    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;
}
```

FindKeysByField 函数：

```cpp
std::vector<std::string> 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();
    if (ExtractField(value, field.first) == field.second) {
      matching_keys.push_back(key);
    }
  }

  delete it;
  return matching_keys;
}

std::string ExtractField(const std::string& value, const std::string& field_name) {
  FieldArray fields = ParseValue(value);
  for(const auto& field : fields){
    if(field.first == field_name){
      return field.second;
    }
  }
  return "No such field_name";
}
```

## 2.2. 二级索引

- 设计目标

由于 FindKeysByField 函数需要遍历所有数据、查询效率低，因此构建字段的二级索引，以提升字段的查询效率。

- 实现思路

二级索引的编码设计：

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。

1. 存储字段名

| |
|---|
| Key：FieldName_FieldValue_Key<br>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<br>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式。

## 3. 数据结构设计

### 3.1 设计方案

我们将 levelDB 中的数据依据 LSM 结构分为三类，分别设计了以下三种索引：

（1）MemTable 索引

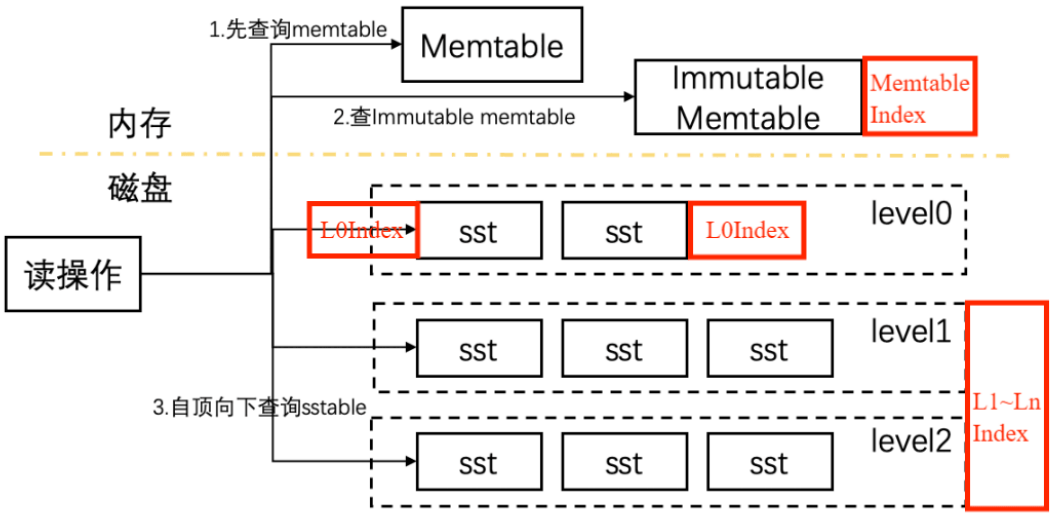
使用一个 LSMTree 构建 MemTable 索引，与内存中的 MemTable 保持一致性，确保数据在写入后立即可查。

（2）Level0 索引

针对 Level0 的 SSTable，每个表生成单独的二级索引。

（3）Level1 及以上的索引

针对 Level1 及以上的 SSTable，设计一个全局的二级索引。使用一个 LSMTree，将 Level1 及以上所有 SSTable 的索引统一存储。




### 3.2 设计思路

### 3.2.1 MemTable

MemTable 存储的是最新写入的数据，位于内存中，可以被快速读写，且按 key 值有序排列，便于范围查询。

由于 MemTable 数据实时性强，索引需要实时更新，且需要与主索引保持一致性，因此为 MemTable 单独维护一个轻量化的二级索引，便于快速插入和检索。

### 3.2.1 Level0 的 SSTable

Level0 中的 SSTable 内部数据有序，但 SSTable 之间可能存在键值范围的重叠，导致整体无序。同时，Level0 生成和合并频率较高，是数据变动最频繁的层。

由于 Level0 的 SSTable 之间数据无序且多版本并存，为每个 SSTable 单独维护一个二级索引，可以快速定位目标数据范围。这样的独立索引设计减少了查找范围交叉带来的复杂性。

### 3.2.2 Level1 及以上的 SSTable

从 Level1 开始，LevelDB 的合并操作确保每个 Level 内的所有 SSTable 数据范围无重叠，整个 Level 数据全局有序，写入频率较低，每个 Key 仅对应一个版本值，便于查询。

由于 Level1 及以上的数据全局有序且单版本，可以使用一个大型的全局索引统一存储这些 Level 的二级索引。这种设计减少了索引文件的数量，同时利用全局有序性优化了查找效率。

## 3.3 总结

（1）MemTable 索引：实时维护最新写入数据的索引，适应 MemTable 数据实时性和高频变动的特点。

（2）Level0 索引：独立为每个 SSTable 构建索引，解决范围重叠和多版本数据带来的复杂性。

（3）Level1 及以上全局索引：基于全局有序性的特点，统一维护一个全局索引，提高查询效率，降低索引维护成本。

## 4.接口/函数设计

### 4.1 建立

AddIndex(indexName, type);

功能说明：创建一个新的二级索引。

UpdateIndex(indexName, key, value);

功能说明：在指定的二级索引中插入或更新数据。

UpdateIndex(records);

功能说明：批量更新索引，用于 MemTable Dump 到 Level0 或 SSTable 合并时。

records：需要更新的记录集合，每条记录包含主键和字段值。

### 4.2 查询

SearchIndex(condition);

功能说明：符合查询条件的 key 和 value。

### 4.3 删除

DeleteIndex(indexName);

功能说明：删除指定的二级索引及其所有索引数据。

## 5.功能测试

### 5.1 单元测试

1. 插入操作：向数据库中插入键值对后，测试是否能利用某个字段查询到对应的 key

2. 更新操作：更新数据库中某个键值对的字段，再分别查询新的字段值和

旧的字段值，测试是否能查询到对应的 key

3. 删除操作：删除数据库中某个键值对的字段，查询这个字段值，测试是否能查询到对应的 key

### 5.2 性能测试

性能的衡量指标：吞吐量、延迟、写放大

性能测试的内容：

1. 创建二级索引后，FindKeysByField 函数的性能
2. 创建二级索引后的读写操作性能和二级索引的空间占用
3. 删除二级索引的用时和对读写性能的影响

## 6. 可能遇到的挑战与解决方案

### 6.1 二级索引和主索引的数据一致性问题

主索引上写操作完成后，需要保证在二级索引完成相应的写操作之前，不能执行二级索引上的查询操作。因此主索引和二级索引上的写操作应该保证是一个原子操作。

### 6.2 合并操作导致的数据不一致

在合并时，会产生多次更新操作，容易出现数据不一致的问题。为了保证所有发生变动的数据都能同步到二级索引上，可以将所有修改的键值对存储到一个临时的数据结构中，在合并之后再利用这个数据结构对二级索引中的数据进行更新和删除。

### 6.3 二级索引的存储开销问题

二级索引的存储会额外占用磁盘空间，尤其在索引字段较多或数据量较大的情况下，可能导致存储资源耗尽。可以设定机制，定期检查和清理无效索引，释放存储空间。

### 6.4 查询性能与一致性验证的冲突

二级索引在查询后仍需通过主索引校验结果的有效性，这会增加查询延迟，降低整体性能。可以设计一个高效的索引缓存，存储最近访问的索引结果，并结合主索引的 SequenceNumber 校验缓存是否有效，减少对主索引的重复查询。也

可以分层优化查询，对 MemTable 和低层级 SSTable 的查询结果优先进行校验，而对更高层级的索引结果可以采用概率校验策略，平衡性能与一致性。

## 7.分工和进度安排

| 功能 | 完成日期 | 分工 |
| --- | --- | --- |
| 二级索引的建立 | 12 月 1 日 | 朱陈媛 |
| 查询、更新、删除 | 12 月 15 日 | 陈予曈 |
| 确保数据一致性 | 12 月 20 日 | 陈予曈、朱陈媛 |
| 功能测试 | 12 月 25 日 | 朱陈媛 |
| 性能测试 | 12 月 30 日 | 陈予曈 |