基于leveldb的单机二级索引

成员：林子骥 郭夏辉

1.项目背景

- LevelDB自身缺陷：

leveldb 是一个高性能的键值存储引擎，但其基础功能主要集中在简单的键值存储和范围查询上，**对于非key字段只能遍历整个SSTable实现。**

此外，LSM树是为块设备设计的并针对写性能进行了优化的，对于需要高搜索性能的二级索引来说，它不是胜任的数据结构。首先，由于二级索引通常只存储主键而不是完整的记录作为值，因此二级索引中的KV对很小。其次，辅助键不是唯一的，并且可以具有多个相关联的主键。LSM-树的**错位写入模式**会将这些非连续到达的值（即相关联的主键）分散到不同级别的多个片段中。因此，查询操作需要搜索基于LSM的二级索引中的**所有级别**以获取这些值片段。

- 现有的实现方式：

TIDB有对二级索引的创建与维护进行代码的开源。不过其由于是分布式系统，数据的一致性性是依靠于事务的处理，会进行多次的log维护。在本次单机实验中是可以忽略的。

2.功能设计

2.1字段设计

```
    字段设计
    //  field_num(32位变长)| fields
    // fields: field1_name_len | field1_name | field1_val_len | field1_val |
    //          field2_name_len | field2_name | field2_val_len | field2_val |.....
```

```
uint64_t MultiDB::SerializeValue(const FieldArray &fields, std::string &res_) {
        PutVarint32(&res_, fields.size());
        for(const auto& field : fields){
            PutLengthPrefixedSlice(&res_, field.first);  // 索引字段名
            PutLengthPrefixedSlice(&res_, field.second); // 索引字段值
        }
        return 0;
    }
```

设计原因：
不像TIDB中，KV数据库只作为存储节点，其元数据有专门的数据点存储。
本次实验的leveldb对于用户输入的字段值，字段名字段数量，字段能否为空等元信息，在真正获取到具体数据前，都是未知的。
因此，需要特意的存储所有元数据，作为一个key的value部分、

2.2根据字段查询具体数据

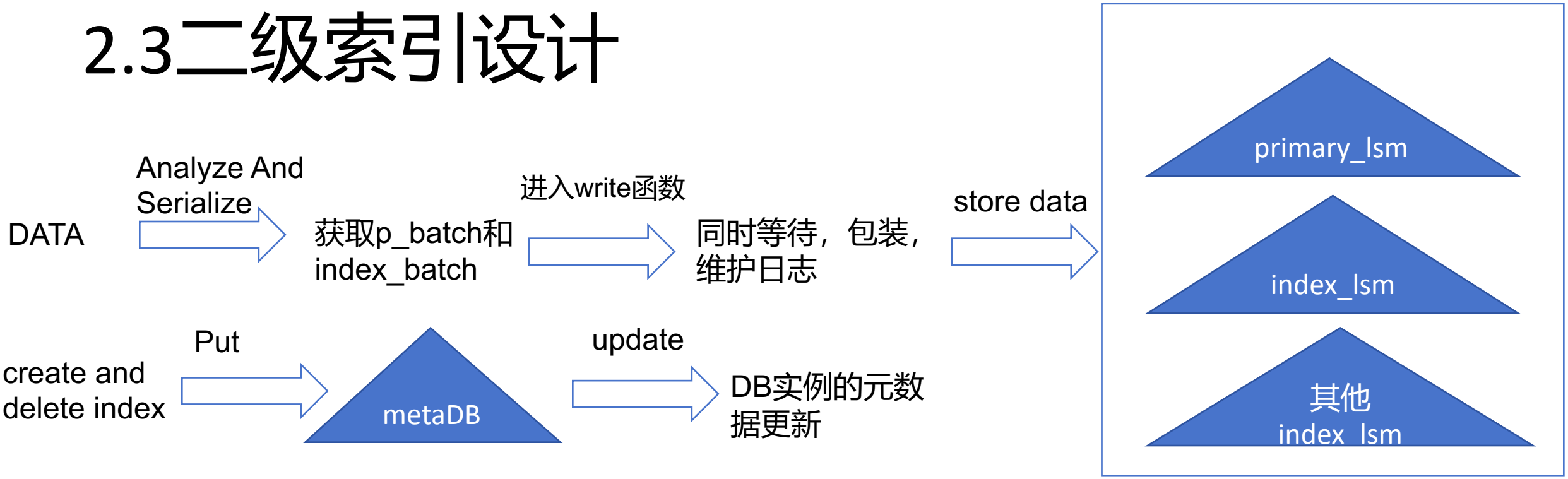
```
        Status MultiDB::Get_keys_by_field(const ReadOptions &options, const Field& field,
 std::vector<std::string> *keys) {
        for(const auto &item_pair : index_db_mp_){
            if(item_pair.first == field.first.ToString()){
                *keys= QueryByIndex(options,field);
                return Status();
            }
        }
        MutexLock l(&mutex_);
        auto snaphot = main_db_->GetSnapshot();
        auto it=main_db_->NewIterator(options);
        mutex_.Unlock();
        it->SeekToFirst();
        keys->clear();
        while(it->Valid()){
            auto val=it->value();//value format:
field_num|field1_len|field_val|field2_len|field2_val
            FieldArray arr;
            auto str_val=std::string(val.data(),val.size());
            ParseValue(str_val,&arr);
            for(auto pr:arr){
                if(pr.first==field.first&&pr.second==field.second){
                    Slice key=it->key();
                    keys->emplace_back(key.data(),key.size());
                    break;
                }
            }
            it->Next();
        }
        main_db_->ReleaseSnapshot(snaphot);
        delete it;
        return Status::OK();
    }
```

如果该字段**已经**建立索引，则走**索引查询**(下面会讲到)；若无，则需要**遍历整个lsm-tree.**

这里我们需要调用DB的NewIterator构造，这个iter为我们提供了扫描整个数据库的可能。并且，它会自动跳过所有已经delete掉的值。

此外，我们所查询的应该是此时此刻的数据，因此为了避免新输入的数据会删除掉原有数据，(防止delete时，触发gc，然后删除之前的数据)，对它建立一个快照，在查询结束后，释放快照。

2.3二级索引设计



Analyze And Serialize：包括从现有DB实例的元数据中查看是否包含索引信息，若包含，则提取出对应的key，然后同时序列化。

同时等待，包装，维护日志：意味着我们是同时对p_batch和index batch做处理的。
p_batch表示初始数据，index_batch表示构建的二级索引数据

DB实现的主体成员

```
class LEVELDB_EXPORT MultiDB:leveldb::DB {
public:
    // Constructor: initializes the main database and index databases.
    //。。。功能函数，包含leveldb原有的代码

private:
    Options op_;
    FileLock* db_lock_;
    std::string db_name_;
    DBImpl *main_db_;                           // The main DB storing primary keys
    DBImpl *meta_db_;
    //dbimpl的模仿
    port::Mutex mutex_;
    std::deque<Writer*> writers_ GUARDED_BY(mutex_);
    std::deque<Writer*> tmp_writers GUARDED_BY(mutex_);
    std::deque<Writer*> pending_list_ GUARDED_BY(mutex_);
    MultiWriteBatch* tmp_batch_ GUARDED_BY(mutex_);
    DBStatus db_status_ GUARDED_BY(mutex_);
    bool is_creating_index_ GUARDED_BY(mutex_);

    //index锁
    //std::vector<std::string> index_list_;
    port::RWLock meta_lock_;//在这里新增了读写锁，具体代码路径在util/port_stdcxx.h
    std::vector<std::string> new_index_field_ GUARDED_BY(meta_lock_);
    std::unordered_map<std::string, DBImpl*> index_db_mp_ GUARDED_BY(meta_lock_);
};
```

- mutex用于实际写入。
- meta_lock_用于与index相关的元数据的写入和读取。由于leveldb不支持c++17版本，因此我们自己写了一个读写锁的实现。

2.3.1索引key值的设计

```
/**
 * @param key : primary key
 * @param val : 索引字段对应的值
 * @param composed_key val_len|val|p_key
 */
void BuildComposedKey(const Slice &key,const Slice &val,std::string* composed_key){
    PutLengthPrefixedSlice(composed_key,val);
    composed_key->append(key.data(),key.size());
}
```

设计理由：我们期望存储的lsm-tree按照我们的如下期望排序，在memtable、filemeta以及sstable中，比较方式按照，先比较val值的大小，若相等，则继续比较key值的大小，若再相等，则比较sequence。这样就能保证全部有序，这样子就可以很方便的查询，一旦在查询的过程中，出现val不相同的情况，那么就可以马上退出。

那为什么根据字段值读取数据，还需要对key进行排序?

2.3.1为什么还要比较key

- 方便数据存放和gc过程。

如果不比较，就会产生一种情况一个SSTable内部数据有序，两个SStable之间不一定有序。我们可以假设一个场景，现在有文件f1<L1,1_1,1_6>,L0层为空，memtable为<_,1_3,1_7>.（<level,index值_primary值，index值_primary值>）。其中f1表示文件的元数据信息。

如果只比较value，那么就有可能在memtable刷盘的时候，会认为memtable与文件f1没有重叠，就会放在同一层，生成f2文件。然后在删除的时候，由于构建的是twoLevel迭代器，就会导致在遍历完f1后，发现f2的首个index值_primary值和要删除的值不同，就会停止在该层继续遍历。但实际上，f2内也有可能存在要删除的数据。

查找的正确性保证：

我们所构建的DBIter，对于level1等高层，也是构建twoLevel迭代器，倘若要删除的数据和Delete数据，在迭代器中是不相邻的，那很容易读到已经被删除的值。

2.3.2具体的写操作

```
    Status MultiDB::Put_with_fields(const WriteOptions &options, const Slice &key, const
FieldArray &fields) {
        //index插入
        ReadLockGuard l(&meta_lock_);//加读锁
        std::unordered_set<int> match;
        //....从indexdb_map中获取主要数据
        Status s;
        MultiWriteBatch multi_batch;
        for(auto &idx_pos : match){
            //item entry:  <index positon in fields,>
            std::string composed_key;
            BuildComposedKey(key,fields[idx_pos].second,&composed_key);
            multi_batch.Put(composed_key,Slice(),fields[idx_pos].first.ToString());
        }

        //主lsm-tree插入
        std::string value;
        SerializeValue(fields,value);//TODO 在serialLize时，获得composedKey
        return Write(options,&multi_batch);//writer后释放meta_write lock
    }
```

这里通过读写锁保护索引的创建和读取。然后我们这里还引进了一个新的MultiWriteBatch对象。
与普通batch不同的是，他这里会有多个batch，会提前将索引写入与主数据写入数据分离，为之后的并发写入memtable提供可能。
此外，因为要同时维护两者的log日志，导致需要提前解析出两者具体的内容。同时，读写锁的构造，为在建立索引时，也允许数据库继续读取数据提供了可能。

Write函数

1.进入我们定义的新writer_队列,等待批量写
2.MakeRoomForWrite，确保index-tree和p-tree都有空间写数据
3.维护index和primary tree的版本号
4.得到批量数据multi_batch，
5.维护index和primary tree的log
6.写入对应的memtable文件
7.返回并且跟新writers队列

```
struct MultiDB::Writer {
    explicit Writer(port::Mutex* mu)
            : batch(nullptr), sync(false), done(false), cv(mu) {}

    Status status;
    MultiWriteBatch* batch;
    bool sync;
    bool done;
    port::CondVar cv;
};
```

```
class LEVELDB_EXPORT MultiWriteBatch {
public:
    MultiWriteBatch();
    // Intentionally copyable.
    MultiWriteBatch(const MultiWriteBatch&) = default;
    MultiWriteBatch& operator=(const MultiWriteBatch&) = default;

    ~MultiWriteBatch();

    // Store the mapping "key->value" in the database.
    void Put(const Slice& key, const Slice& value,const std::string &index_name);

    // If the database contains a mapping for "key", erase it.  Else do nothing.
    void Delete(const Slice& key,const std::string &index_name);

    //........其余功能与writerBatch类似

private:
    friend class MultiWriteBatchInternal;
    friend class WriteBatch;
    friend class MultiDB;
    BatchStatus s;
    WriteBatch * main_batch;
    std::unordered_map<std::string,WriteBatch *> index_batch_map_;
```

重写write函数的原因

- 因为本实验没有事务机制来保证主数据与索引数据的一致性。如果还使用leveldb自带的put和write，会出现下述错误。

在高并发的情况下，主数据和索引数据都先写入各自的writer队列中，通过BuildGraph函数来获得一个含有较多数据的batch。但是因为索引数据会远小于主数据，因此会出现有大量索引数据已经写入数据库，但主数据还没有log持久化的情况。同时因为锁的竞争，无法保证主数据与索引数据是否会在同一批次中写入，这是不符合原子性的。

2.3.3读操作

QueryByIndex:因为要读取元数据，因此要加读锁。在这里，我们会通过GetRawResult来获取可能的结果值，然后到maindb中**验证**该值是否存在。需要注意的是，整个读取的过程是无锁操作，在GetRawResult时，会对maindb创建一个快照，并且获取对应的版本号到new_op内，这样就可以保证在验证时，不会被新加入的数据所干扰。然后对于要删除的值(即验证失败的值)，打包成batch加快处理，并将BatchStatus设置为ForIndexEntryDelete。

设计原因：实际上我们也可以在delete primary key时，从主lsm-tree中查看对应的字段值，然后往index lsm-tree中插入delete记录，但这样子会大大阻塞写的性能。而现在index lsm-tree的delete只在未命中时发生，并且以批处理的形式更新，速度会更快。相比前者，只会轻微的牺牲一些读性能。

```
std::vector<std::string> MultiDB::QueryByIndex(const ReadOptions &options, const
Field &field) {
        std::unordered_set<std::string> tmp_res;
        ReadOptions new_op = options;
        ReadLockGuard read_guard(&meta_lock_);
        this->GetRawResult(new_op,field,tmp_res);
        std::vector<std::string> res;
        MultiWriteBatch multi_batch;
        for (const auto& key : tmp_res) {
            std::string value;
            auto get_s = this->main_db_->Get(new_op,key,&value);
            assert(get_s.ok() || get_s.IsNotFound());
            if(!get_s.IsNotFound() || Validate(value,field)){
                res.push_back(key);
            }else{
                std::string composed_key;
                BuildComposedKey(key,field.second,&composed_key);
                multi_batch.Delete(composed_key,field.first.ToString());
            }
        }
        this->main_db_->ReleaseSnapshot(new_op.snapshot);
        multi_batch.s = BatchStatus::ForIndexEntryDelete;
        if(!multi_batch.index_batch_map_.empty()){
            auto status = Write(WriteOptions(),&multi_batch);
```

```
    void MultiDB::GetRawResult(ReadOptions &options, const Field &field,
std::unordered_set<std::string> &tmp_res){
        auto index_db = index_db_mp_[field.first.ToString()];
        Status s;
        MutexLock l(&mutex_);
        auto iter = index_db->NewIterator(ReadOptions());//index_db的相关写都会由mutex_控制
        options.snapshot = main_db_->GetSnapshot();
        //这里还需要获得main_db的sequence
        mutex_.Unlock();
        std::string lkey;
        PutLengthPrefixedSlice(&lkey,field.second);
        iter->Seek(lkey);//输入应该是val_len|val|user_key,其中user_key为None
        //iter->SeekToFirst();
        for (; iter->Valid(); iter->Next())
            //。。。获取对应值
        delete iter;
    }
```

2.3.4Index创建

- 首先我们要考虑的是，在index创建这个时间点，其数据由哪几部分组成。
- 我们可以假设有一个场景是，在插入大量数据的过程中，我们开始createIndex，获取mutex锁。此时有三种数据。一是已经存在在maindb的数据，二是已经存在在writer函数中writers队列内的batch的数据，但还没有写入leveldb；三是接下来put操作内新写入的数据。

- 这里我们为了保证数据能够全部有序插入，通过写锁来维护元数据，注意这里获取写锁的前提，是读锁全部释放。因此当获取写锁时，能够保证Write函数内的所有数据已经写入到db中了。然后接下来就只剩下maindb内的数据已经Put的新数据需要处理。

- 这里put时需要获取读锁，会被当前写锁所阻塞。因此我们现在也只需要处理maindb内的数据。

具体代码

```
Status MultiDB::CreateIndexOnField(const std::string &field_name) {

    //meta_lock_.WriteLock();
    WriteLockGuard l_meta(&meta_lock_);//写锁
    is_new = true;
    MutexLock l(&mutex_);

    Options op = op_;
    op.index_mode = true;//标志该数据库为indexdb
    //op.abandon_log = true;
    Status status;
    DB *field_tmp_db;
    status = DB::Open(op, db_name_ + "/index_" + field_name , &field_tmp_db);//若mainDB已经打开，则无需Open

    auto field_db_impl = static_cast<DBImpl *>(field_tmp_db);
    index_db_mp_[field_name] = field_db_impl;

    //。。。。。元数据持久化。。。。。

    ReadOptions read_op;
    read_op.snapshot = main_db_->GetSnapshot();//保证该快照内的数据不会被main_db gc掉，读取不超过该seq的数据到index_db中，支持无锁并
发

    //已经这里获得快照时会获取到锁
    auto it=main_db_->NewIterator(read_op);
    mutex_.Unlock();//现在保证了接下来的最新数据可以继续解析

    MultiWriteBatch index_batch;
    index_batch.s = BatchStatus::ForIndexCreate;
    it->SeekToFirst();
    //-------------------------------------------------------------------------------遍历main数据库
    std::string composed_key;
    while(it->Valid()){
        //...........遍历maindb，将结果都放入index_batch
        it->Next();
    }
    delete it;
    main_db_->ReleaseSnapshot(read_op.snapshot);
    //把index_batch依次放入tmp_writer队列，然后该队列还会放入正在写入的

    status = Write(WriteOptions(),&index_batch);
    //..........元数据存储
    return status;
}
```

元数据的更新

```
    enum IndexStatus{
        CreateIndex,
        DeleteIndex,
        DoneCreate,
        DoneDelete
    };
        /**
         *
         * @param s
         * @param index_name
         * @param result  s（32位）|index_name
         */
        static void PackageIndexStatus(IndexStatus s,const std::string
&index_name,std::string *result){
            result->clear();
            PutFixed32(result,s);
            result->append(index_name);
        }
        //..............打开indexdb
        //元数据持久化
        std::string meta_key_create;
        PackageIndexStatus(IndexStatus::CreateIndex,field_name,&meta_key_create);
        meta_db_->Put(WriteOptions(),meta_key_create,Slice());
        //遍历maindb.................,写入indexdb
        //元数据持久化
        std::string meta_key_done;
        PackageIndexStatus(IndexStatus::DoneCreate,field_name,&meta_key_done);
        std::string delete_key_done;
        PackageIndexStatus(IndexStatus::DoneDelete,field_name,&delete_key_done);
        WriteBatch meta_batch;
        meta_batch.Put(meta_key_done,Slice());
        meta_batch.Delete(meta_key_create);
        meta_batch.Delete(delete_key_done);
```

存储到数据库的<k,v>格式为<IndexStatus | fieldName,None>.然后在开始转移数据时，往metadb内写入`IndexStatus::CreateIndex`,代表正在创建数据库。然后当写入完成后，告诉数据操作已经完成。

值得注意到时，这里是一个小批量操作，用于删除之前的数据以及记录当前数据库状态，保证其原子操作，方便接下来的数据恢复。

2.3.5index删除

```
Status MultiDB::DeleteIndex(const std::string &field_name) {
    std::string meta_key;

    WriteLockGuard w_l(&meta_lock_); //保证删除其对应db实例时，不会有数据再读
    //assert(index_db_mp_.find(field_name) != index_db_mp_.end());
    if(index_db_mp_.find(field_name) == index_db_mp_.end()){
        return  Status::NotFound("nothing to delete",Slice());
    }
    auto impl = index_db_mp_[field_name];
    auto db_name = impl->dbname_;
    delete impl;
    index_db_mp_.erase(field_name);
    meta_lock_.WriteUnlock();

    DestroyDB(db_name,Options());

    PackageIndexStatus(IndexStatus::DeleteIndex,field_name,&meta_key);
    meta_db_->Put(WriteOptions(),meta_key,Slice());
    WriteBatch meta_batch;
    std::string meta_key_done;
    std::string create_key;
    PackageIndexStatus(IndexStatus::DoneDelete,field_name,&meta_key_done);
    PackageIndexStatus(IndexStatus::DoneCreate,field_name,&create_key);
    meta_batch.Put(meta_key_done,Slice());
    meta_batch.Delete(meta_key);
    meta_batch.Delete(create_key);

    return meta_db_->Write(WriteOptions(),&meta_batch);//表示已经更新完成
}
```

元数据的写入和index创建时类似，一开始都是告知数据库，正在创建，当结束后，告知数据库创建完成。

与index的创建不同的是，我们这里无需一个个删除对应的键值对。而是删除其对应的元数据，`index_db_mp_`操作，并且在内存中删除后，马上解锁，这样就可以运行新的数据继续写入。

2.3.6数据崩溃与数据恢复

这里数据的恢复主要分为两个部分：

1. 一是元数据的恢复，即在打开数据库时，也能够恢复其具备哪些索引；
2. 二是恢复各个数据的具体内容。

元数据恢复

- 读到createDone标志的可能：1.index成功创建并被使用；2.deleteIndex没有完成，该情况因为原子的batch写入操作，不会出现
- 读到createIndex的可能：还没有创建成功，需要重新创建
- 读到DeleteDone的可能：成功删除
- 读到deleteIndex的可能：1.已经成功删除目录，但是元数据还没有跟新；2.还没有成功删除目录。这里再次删除，对于报错信息不理会

```
case IndexStatus::DoneDelete:
    break;
case IndexStatus::DoneCreate:
    if(index_db_mp_.find(field_name) != index_db_mp_.end())break;
    options.index_mode = true;
    status = DB::Open(options, db_name_ + "/index_" + field_name, &field_tmp_db);//若mainDB已经打开，则无需Open

    if (!status.ok()) {
        std::cerr << "Failed to open index DB: " << status.ToString() << std::endl;
        abort();  // Handle the error as appropriate
    }
    assert(static_cast<DBImpl *>(field_tmp_db)!=nullptr) ;
    index_db_mp_[field_name] = static_cast<DBImpl *>(field_tmp_db);
    break;
case IndexStatus::DeleteIndex:
    //继续进行删除操作，即使不存在也没关系，不理会报错信息
    DestroyDB(db_name_ + "/index_" + field_name,options);
    PackageIndexStatus(IndexStatus::DeleteIndex,field_name,&meta_key);
    PackageIndexStatus(IndexStatus::DoneDelete,field_name,&meta_key_done);
    PackageIndexStatus(IndexStatus::DoneCreate,field_name,&create_key);
    meta_batch.Clear();
    meta_batch.Put(meta_key_done,Slice());
    meta_batch.Delete(meta_key);
    meta_batch.Delete(create_key);
    meta_db_->Write(WriteOptions(),&meta_batch);
    break;
case IndexStatus::CreateIndex:
    //没有创建成功，需要恢复日志，在挂机点继续创建。
    status = DB::Open(options, db_name_ + "/index_" + field_name, &field_tmp_db);//若mainDB已经打开，则无需Open
    if (!status.ok()) {
        std::cerr << "Failed to open index DB: " << status.ToString() << std::endl;
        abort();  // Handle the error as appropriate
    }
    assert(static_cast<DBImpl *>(field_tmp_db)!=nullptr) ;
    index_db_mp_[field_name] = static_cast<DBImpl *>(field_tmp_db);
    RebuildIndexDB(field_name,static_cast<DBImpl *>(field_tmp_db));
    break;
```

如何在creatindex崩溃时恢复

这里通过`RebuildIndexDB`函数实现。

我们需要注意`main_db_->NewIterator`的性质，它是将key从小到大返回的。因此我们可以获得最后一次的写入的key值，然后再次在maindb中查找，达到了能够在数据库构建索引崩溃一半时，高校重建的功能，这对于特别庞大的数据库构建索引是十分有必要的。

同时，能够进行上述操作，需要满足，在createIndex时，与索引相关的数据没有完全写入数据库中，这是显然的。因为读取相关索引信息时，会被我们的读锁阻塞。

```
void MultiDB:: RebuildIndexDB(const std::string &field_name,DBImpl *field_db_impl){
        mutex_.AssertHeld();
        ReadOptions read_op;
        read_op.snapshot = main_db_->GetSnapshot();//保证该快照内的数据不会被main_db gc掉，读取不超过该seq的数据到index_db中，支持无锁并发
        auto it=main_db_->NewIterator(read_op);
        auto tmp_it = field_db_impl->NewIterator(ReadOptions());
        tmp_it->SeekToLast();
        if(tmp_it->Valid()){
            auto key = tmp_it->key();//这个现在是
            Slice p_key;
            ExtractUserkeyFromComposedKey(key,p_key);//得到 primary key
            it->Seek(p_key);
        }else{
            it->SeekToFirst();//若数据库为空，则都第一个数据
        }

        std::string composed_key;
        while(it->Valid()){
            composed_key.clear();
            auto val=it->value();
            FieldArray arr;
            auto str_val=std::string(val.data(),val.size());
            ParseValue(str_val,&arr);
            for(auto &pr:arr){
                if(pr.first== field_name){
                    BuildComposedKey(it->key(),pr.second,&composed_key);
                    field_db_impl->Put(WriteOptions(),composed_key,Slice());//TODO :也许批量写速度会更快。是否需要转为multi_batch?
                    break;
                }
            }
            it->Next();
        }
        delete it;
        main_db_->ReleaseSnapshot(read_op.snapshot);
```

DB具体数据恢复

恢复各个数据库的具体数据，主要依靠其对应的log日志，其主要功能都在AddRecord功能内。在这里，我们可以注意到，我们是先写index log，再写main log的，这主要是为了应对index log写完后马上崩溃的情况。

对于插入操作，即使我们多写了index log部分，那么再后续查找时，会大概率会因为数据不匹配而delete 当前的index 数据。

对于删除操作，若原有main log写入的是 delete k1，逻辑上，它会期望indexdb将无法找到k1对应的数据。但在崩溃后leveldb会认为maindb的数据完全丢失了，要求重新输入。因为indexdb不会对delete k1做日志处理，因此log内也不会出现重复数据。

```
    // 写入索引数据库的日志
    status = index_db->log_->AddRecord(WriteBatchInternal::Contents(index_batch));
    if (!status.ok()) {
        // 处理日志写入错误
        last_s = status;
    }
}

// 写入主数据库的日志
if(main_batch != nullptr){
    status = main_db_->log_->AddRecord(WriteBatchInternal::Contents(main_batch));
    if (!status.ok()) {
        // 处理日志写入错误
        last_s = status;
```

2.3.7索引字段范围查询

```
std::vector<std::string> MultiDB::RangeQueryByIndex(ReadOptions &options, const std::string &index_name,
                                                    const Slice &begin, const Slice& end) {


    meta_lock.ReadLock()
    auto index_db = index_db_mp_[index_name];
    meta_lock.ReadUnLock()

    Status s;
    MutexLock l(&mutex_);
    auto iter = index_db->NewIterator(ReadOptions());//index_db的相关写都会由mutex_控制
    options.snapshot = main_db_->GetSnapshot();
    //这里还需要获得main_db的sequence
    mutex_.Unlock();
    std::string start_lk;
    PutLengthPrefixedSlice(&start_lk,begin);
    auto limit_val = end.ToString();
    iter->Seek(start_lk);//输入应该是val_len|val|user_key,其中user_key为None

    std::unordered_map<std::string, std::unordered_set<std::string>> val_map;
    for (; iter->Valid(); iter->Next()) {
```

这个和我们之前的点查比较类似，唯一的区别就是，我们的终止条件，变为判断是否大于end，而不是是否等于自身。

能够实现该功能的前提则是，整个leveldb的数据排序，会按照index值有序排列。其具体的实现功能可以看我们的细节讲解。

可以实现的扩展功能：

因为我们的设计里，不仅仅是value是有序的，其组合键(val_len|val|p_key)内部的p_key也是有序的，因此它实际上也还支持对p_key和index同时进行范围查询。在一个value下，一旦有一个iter.key不符合范围，可以马上退出。这对于在一个索引下，有许多key值存在的情况，可以加快查询速度。

2.3.8索引数据库的手动压缩

这里和原先的leveldb的功能实现比较相似，需要注意的就是我们的user key从原来的 primaryKey变成了val_len|val|p_key的组合。因此在构造时，我们也需要将初始的输入begin和end(这里的格式都是val)，改为对应形式。

```
void MultiDB::CompactRange(const std::string &field_name,const Slice *begin, const Slice *end) {
    MutexLock l(&mutex_);
    assert(index_db_mp_.find(field_name) != index_db_mp_.end());
    auto index_db = index_db_mp_[field_name];

    std::string b;
    std::string e;
    Slice b_final;
    Slice e_final;
    if(begin != nullptr){
        PutLengthPrefixedSlice(&b,*begin);
        b_final = b;
    }
    if(end != nullptr){
        PutLengthPrefixedSlice(&e,*end);
        e_final = e;
    }
    index_db->CompactRange(&b_final,&e_final);
}
```

3.细节讲解

- 之前我们说期望存储的lsm-tree按照我们的如下期望排序，在memtable、filemeta以及sstable中，比较方式按照，先比较val值的大小，若相等，则继续比较key值的大小。
- 但其具体实现方式，并不是仅仅将val_len|val|key作为一整个key来比较，这里我们对数据库的比较操作进行了修改

```
enum ValueType { kTypeDeletion = 0x0, kTypeValue = 0x1,kTypeWriteIndex = 0x2,kTypeDeleteIndex=0x3 };
```

```
int InternalKeyComparator::Compare(const Slice& akey, const Slice& bkey) const {
  // Order by:
  //    increasing user key (according to user-supplied comparator) ,there userkey may be composed by val_len|val|p_key
  //    decreasing sequence number
  //    decreasing type (though sequence# should be enough to disambiguate)
  Slice a,b;

  //a = has_index_ ? ExtractPrefixDeComposedUserKey(akey) : Slice(akey.data(),akey.size() - 8);

  auto v1 = static_cast<ValueType>(DecodeLastByte(bkey.data() + bkey.size() - 8));
  auto v0 = static_cast<ValueType>(DecodeLastByte(akey.data() + akey.size() - 8));
  int r = 0;
  switch (v0) {
      case kTypeWriteIndex:
      case kTypeDeleteIndex:
          if(v1 == kTypeWriteIndex || v1 == kTypeDeleteIndex){
              //复合键比较，先比val，再比key
              auto p = akey.data();
              auto a_ptr_key_limit = ParseComposedUserKey(akey,a);
              auto b_ptr_key_limit = ParseComposedUserKey(bkey,b);
              r = user_comparator_->Compare(a, b);
              r = r == 0 ?
                      user_comparator_->Compare(Slice(a.data() + a.size(),a_ptr_key_limit - a.data() - a.size()),
                                                Slice(b.data() + b.size(),b_ptr_key_limit - b.data() - b.size())) :r;
          }
          break;
      default:
          a = Slice(akey.data(),akey.size() - 8);
          b = Slice(bkey.data(),bkey.size() - 8);
          r = user_comparator_->Compare(a, b);
  }
  if (r == 0) {
    const uint64_t anum = DecodeFixed64(akey.data() + akey.size() - 8);
    const uint64_t bnum = DecodeFixed64(bkey.data() + bkey.size() - 8);
    if (anum > bnum) {
      r = -1;
    } else if (anum < bnum) {
      r = +1;
    }
  }
  return r;
}
```

因为当前InternalKey的比较方式与原来的leveldb的比较方式有所不同，这里我们需要更改其比较器。但是若全部通过新增一个比较器来实现，代码的工程会非常庞大，这里我们通过对valueType进行扩展来实现，kTypeWriteIndex表示与index相关的写入，kTypeDeleteIndex表示与index相关的删除操作。

测试情况

功能测试

Basic

- Basic.TestParse:表示是否正常序列化
- Basic.SIMPLE_KEY_FIELD:表示

TestIndex1：

- TestIndex1.QueryKeyByFieldUnderLargeInsert:在大数据下，扫描整个数据库查找对应field字段所对应的数据
- TestIndex1.TestQueryIndexInMemory ： 在小数据插入下，index查询是否能正确查看memtable的值
- TestIndex1.QueryByIndexUnderLargeInsert：在大数据插入下，index查询是否能正常返回
- TestIndex1.QueryByIndexUnderDelete：在大数据插入和少量删除操作下，index查询是否正确
- TestIndex1.IndexCompaction：指定需要合并的范围，能否对index对应的数据库正确合并

TestIndex2

- TestIndex2.RangeQueryByIndexUnderLargeInsert：在大数据插入下，执行field范围查询
- TestIndex2.CreateIndexDuringInsert ： 在并行写线程种，能否正确createIndex

TestMeta

- TestMeta.SimpleCreateDelete：能否正常进行createindex和deleteindex操作

TestRecovery

- TestRecovery.metaField ： meta数据能否正常恢复
- TestRecovery.recoveryAll ： 在插入大量数据，建立index，能否正常恢复。

```
[==========] Running 12 tests from 5 test suites.
[----------] Global test environment set-up.
[----------] 2 tests from Basic
[ RUN      ] Basic.TestParse
[       OK ] Basic.TestParse (45 ms)
[ RUN      ] Basic.SIMPLE_KEY_FIELD
[       OK ] Basic.SIMPLE_KEY_FIELD (24 ms)
[----------] 2 tests from Basic (69 ms total)

[----------] 5 tests from TestIndex1
[ RUN      ] TestIndex1.QueryKeyByFieldUnderLargeInsert
[       OK ] TestIndex1.QueryKeyByFieldUnderLargeInsert (3928 ms)
[ RUN      ] TestIndex1.TestQueryIndexInMemory
[       OK ] TestIndex1.TestQueryIndexInMemory (82 ms)
[ RUN      ] TestIndex1.QueryByIndexUnderLargeInsert
[       OK ] TestIndex1.QueryByIndexUnderLargeInsert (7438 ms)
[ RUN      ] TestIndex1.QueryByIndexUnderDelete
[       OK ] TestIndex1.QueryByIndexUnderDelete (6339 ms)
[ RUN      ] TestIndex1.IndexCompaction
[       OK ] TestIndex1.IndexCompaction (7379 ms)
[----------] 5 tests from TestIndex1 (25169 ms total)

[----------] 2 tests from TestIndex2
[ RUN      ] TestIndex2.RangeQueryByIndexUnderLargeInsert
[       OK ] TestIndex2.RangeQueryByIndexUnderLargeInsert (6331 ms)
[ RUN      ] TestIndex2.CreateIndexDuringInsert
[       OK ] TestIndex2.CreateIndexDuringInsert (11491 ms)
[----------] 2 tests from TestIndex2 (17822 ms total)

[----------] 1 test from TestMeta
[ RUN      ] TestMeta.SimpleCreateDelete
[       OK ] TestMeta.SimpleCreateDelete (55 ms)
[----------] 1 test from TestMeta (55 ms total)

[----------] 2 tests from TestRecovery
[ RUN      ] TestRecovery.metaField
[       OK ] TestRecovery.metaField (94 ms)
[ RUN      ] TestRecovery.recoveryAll
[       OK ] TestRecovery.recoveryAll (42 ms)
[----------] 2 tests from TestRecovery (136 ms total)

[----------] Global test environment tear-down
[==========] 12 tests from 5 test suites ran. (43254 ms total)
[  PASSED  ] 12 tests.
```

性能测试

```
static const char* FLAGS_benchmarks =
        "fillseq,"
        "fillsync,"
        "fillrandom,"
        "readAfterCreateIndex,"
        "readInCreateIndex,"
        "writeAfterCreateIndex,"
        "writeAfterDeleteIndex,"
        "writeInCreateIndex,"
        "writeInDeleteIndex,"
        "fill100K,";

// Number of key/values to place in database
static int FLAGS_num = 100000;

static int FLAGS_dup_num = 100;

// Number of read operations to do.  If negative, do FLAGS_num reads.
static int FLAGS_reads = -1;

// Number of concurrent threads to run.
static int FLAGS_threads = 1;

// Size of each value
static int FLAGS_value_size = 100;

static int FLAGS_each_field_siez = 50;

static int FLAGS_index_size = 16;
```

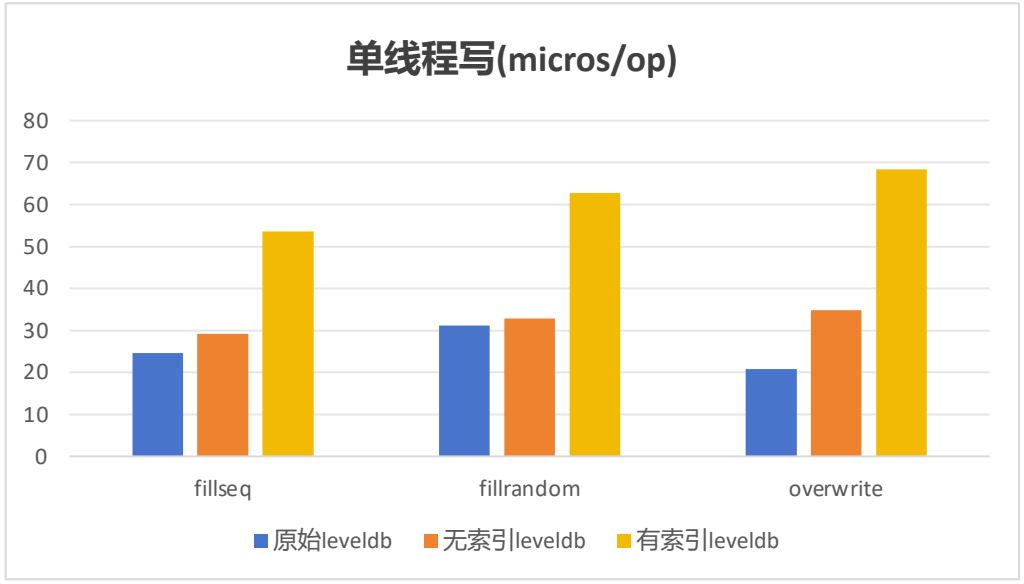
性能测试主要是仿照leveldb固有的bench进行修改。修改后的文件在benchmarks/db_bench_field.cc中。

它支持下述测试样例 "fillseq,"：在索引和非索引下的单线程顺序写 "fillsync,"：在索引和非索引下的单线程sync写。 "fillrandom,"：在索引和非索引情况下的单线程随机写 "readAfterCreateIndex,"：在索引创立后的随机读 "readInCreateIndex,"：在索引创建中的随机读 "writeAfterCreateIndex,"在索引创建后的随机写(这主要用于比较索引创建前后) "writeAfterDeleteIndex,"在索引删除后的随机写 "writeInCreateIndex,"在索引创建中的随机写 "writeInDeleteIndex,"在索引删除时，随机写。 "fill100K,";单线程下的大批量写

支持下述测试情况：

- 不同数据量下的性能、不同读写比例下的性能、不同键值对大小下的性能。
- 其中FLAGS_num表示插入的数据量；FLAGS_reads表示可以并发的读线程，若值为负数，其并发量会自动置为FLAGS_num;FLAGS_each_field_size表示非索引部分的field值的大小；FLAGS_index_size表示二级索引的大小设置。

性能测试



我们可以看到这里单线程的写操作(除了overwrite)，其几乎吞吐量几乎降低了一倍，这是因为这里的主要开销都在log的维护上，因为当前有一个索引，加上要写maindb的log，要进行两次IO的写。同时我们也可以看到，就比如fillseq，P75延迟在50-60左右，而P99延迟则在43131 micros，这主要是因为在数据写入的增加，后续的写入操作很容易产生写放大，因此延迟增高。(这里的吞吐量两次测量优点不一样)

```
------------------------------------------------
fillseq      :      62.551 micros/op;    1.3 MB/s
Microseconds per op:
Count: 100000  Average: 62.5495  StdDev: 293.43
Min: 33.0000  Median: 52.6771  Max: 43131.0000
------------------------------------------------------
[      30,      35 )     649   0.649%   0.649%
[      35,      40 )   15601  15.601%  16.250% ###
[      40,      45 )   11601  11.601%  27.851% ##
[      45,      50 )   11926  11.926%  39.777% ##
[      50,      60 )   38187  38.187%  77.964% ########
[      60,      70 )   14275  14.275%  92.239% ###
[      70,      80 )    2842   2.842%  95.081% #
[      80,      90 )    1232   1.232%  96.313%
[      90,     100 )     642   0.642%  96.955%
[     100,     120 )     546   0.546%  97.501%
```

```
fillsync     :    2750.230 micros/op;    0.0 MB/s (100 ops)
Microseconds per op:
Count: 100  Average: 2727.1300  StdDev: 2397.30
Min: 1522.0000  Median: 2226.1905  Max: 18900.0000
------------------------------------------------------
[    1400,    1600 )       3   3.000%   3.000% #
[    1600,    1800 )      10  10.000%  13.000% ##
[    1800,    2000 )      18  18.000%  31.000% ####
[    2000,    2500 )      42  42.000%  73.000% ########
[    2500,    3000 )      11  11.000%  84.000% ##
[    3000,    3500 )       6   6.000%  90.000% #
[    3500,    4000 )       3   3.000%  93.000% #
[    4000,    4500 )       4   4.000%  97.000% #
[   12000,   14000 )       1   1.000%  98.000%
[   14000,   16000 )       1   1.000%  99.000%
```

```
fillrandom   :      75.284 micros/op;    1.0 MB/s
Microseconds per op:
Count: 100000  Average: 75.2838  StdDev: 236.67
Min: 36.0000  Median: 62.6701  Max: 25331.0000
------------------------------------------------------
[      35,      40 )     548   0.548%   0.548%
[      40,      45 )    5411   5.411%   5.959% #
[      45,      50 )   10428  10.428%  16.387% ##
[      50,      60 )   26143  26.143%  42.530% #####
[      60,      70 )   27977  27.977%  70.507% ######
[      70,      80 )   17284  17.284%  87.791% ###
[      80,      90 )    5482   5.482%  93.273% #
[      90,     100 )    1820   1.820%  95.093%
[     100,     120 )    1253   1.253%  96.346%
[     120,     140 )     498   0.498%  96.844%
[     140,     160 )     239   0.239%  97.083%
```

索引创建过程中对读性能影响

可以发现，在创建后读取，其吞吐量是读取是创建的3倍。这里主要是因为在后期创建的时候，会阻塞与index值相关的读取操作。即不支持在创建索引时读取索引数据。

但是，我们这里发现前者的P75延迟反而比后者高。后者在创建索引前，会通过扫描整个数据库的方法查找符合值。这就说明在这种情况下，走index查询的验证步骤，会耗费更多的时间。导致这种情况的原因是，我们将数据量设置为10000，但若是将数据量改为100000，索引查询将显然更快。但是因为我们的测试程序需要通过多次随机读来均摊compaction的影响的原因，会导致整个过程耗时接近半小时，很难继续其他实验。因此这两者我们将在大数据下，一次读取过程中单独比较，具体比较看FindKeysByField和QueryByIndex的比较。

```
readAfterCreateIndex :     206.370 micros/op; (100 of 100000 found)
Microseconds per op:
Count: 100  Average: 206.2300  StdDev: 1253.26
Min: 31.0000  Median: 66.0000  Max: 12668.0000
------------------------------------------------------
[      30,      35 )       1   1.000%   1.000%
[      35,      40 )       9   9.000%  10.000% ##
[      40,      45 )      10  10.000%  20.000% ##
[      45,      50 )       8   8.000%  28.000% ##
[      50,      60 )      13  13.000%  41.000% ###
[      60,      70 )      15  15.000%  56.000% ###
[      70,      80 )       5   5.000%  61.000% #
[      80,      90 )       7   7.000%  68.000% #
[      90,     100 )       3   3.000%  71.000% #
[     100,     120 )      15  15.000%  86.000% ###
[     120,     140 )       5   5.000%  91.000% #
[     140,     160 )       3   3.000%  94.000% #
[     180,     200 )       1   1.000%  95.000%
[     200,     250 )       4   4.000%  99.000% #
[   12000,   14000 )       1   1.000% 100.000%
```

```
readInCreateIndex :     710.780 micros/op; (100 of 100000 found)
Microseconds per op:
Count: 100  Average: 349.3700  StdDev: 1951.72
Min: 33.0000  Median: 62.2222  Max: 16372.0000
------------------------------------------------------
[      30,      35 )       2   2.000%   2.000%
[      35,      40 )       7   7.000%   9.000% #
[      40,      45 )      18  18.000%  27.000% ####
[      45,      50 )       6   6.000%  33.000% #
[      50,      60 )      13  13.000%  46.000% ###
[      60,      70 )      18  18.000%  64.000% ####
[      70,      80 )      16  16.000%  80.000% ###
[      80,      90 )       6   6.000%  86.000% #
[      90,     100 )       6   6.000%  92.000% #
[     100,     120 )       2   2.000%  94.000%
[     120,     140 )       2   2.000%  96.000%
[     160,     180 )       1   1.000%  97.000%
[    1400,    1600 )       1   1.000%  98.000%
[   10000,   12000 )       1   1.000%  99.000%
[   16000,   18000 )       1   1.000% 100.000%
```

索引创建过程中对写性能影响

```
writeAfterCreateIndex :      83.379 micros/op;    0.9 MB/s
Microseconds per op:
Count: 100000  Average: 83.3787  StdDev: 275.27
Min: 42.0000  Median: 67.9037  Max: 23778.0000
------------------------------------------------------
[      40,      45 )     173   0.173%   0.173%
[      45,      50 )    3739   3.739%   3.912% #
[      50,      60 )   18558  18.558%  22.470% ####
[      60,      70 )   34832  34.832%  57.302% #######
[      70,      80 )   26509  26.509%  83.811% #####
[      80,      90 )    7728   7.728%  91.539% ##
[      90,     100 )    2945   2.945%  94.484% #
[     100,     120 )    1686   1.686%  96.170%
```

```
writeInCreateIndex :      94.559 micros/op;    1.0 MB/s
Microseconds per op:
Count: 100000  Average: 79.8696  StdDev: 1464.74
Min: 27.0000  Median: 64.4350  Max: 453184.0000
------------------------------------------------------
[      25,      30 )     261   0.261%   0.261%
[      30,      35 )    6298   6.298%   6.559% #
[      35,      40 )    7182   7.182%  13.741% #
[      40,      45 )    6128   6.128%  19.869% #
[      45,      50 )    5518   5.518%  25.387% #
[      50,      60 )   13109  13.109%  38.496% ###
[      60,      70 )   25939  25.939%  64.435% #####
[      70,      80 )   22127  22.127%  86.562% ####
[      80,      90 )    6613   6.613%  93.175% #
[      90,     100 )    2077   2.077%  95.252%
[     100,     120 )    1269   1.269%  96.521%
[     120,     140 )     464   0.464%  96.985%
[     140,     160 )     239   0.239%  97.224%
[     160,     180 )     271   0.271%  97.495%
[     180,     200 )     318   0.318%  97.813%
[     200,     250 )     830   0.830%  98.643%
```

1.可以看到两者的吞吐量差异不会像read相差那么大，这可能主要是因为，批量写可以很好的分担阻塞队列对写入性能的影响。
2.但后者吞吐量还是会低一些，是因为在新写入的数据会阻塞在进入writers队列当中。
3.然后可以发现后者的延迟情况。后者的最低延迟最低，这是因为它只写了一个log。在p75时两者比较相近，这是因为此时索引还没有完全建立。而在建立后，队列阻塞，因此后续的写入并没有立马响应P99较高，在index建立完成后，才继续写入。

索引删除过程中对写性能影响

```
writeAfterDeleteIndex :      43.873 micros/op;    1.8 MB/s
Microseconds per op:
Count: 100000  Average: 43.8724  StdDev: 145.62
Min: 25.0000  Median: 38.1901  Max: 22468.0000
------------------------------------------------------
[      25,      30 )    9582   9.582%   9.582% ##
[      30,      35 )   19023  19.023%  28.605% ####
[      35,      40 )   33533  33.533%  62.138% #######
[      40,      45 )   23889  23.889%  86.027% #####
[      45,      50 )    6741   6.741%  92.768% #
[      50,      60 )    3818   3.818%  96.586% #
[      60,      70 )     907   0.907%  97.493%
[      70,      80 )     361   0.361%  97.854%
[      80,      90 )     215   0.215%  98.069%
[      90,     100 )     117   0.117%  98.186%
```

```
writeInDeleteIndex :      53.492 micros/op;    1.5 MB/s
Microseconds per op:
Count: 100000  Average: 53.3246  StdDev: 231.21
Min: 26.0000  Median: 40.9501  Max: 37926.0000
------------------------------------------------------
[      25,      30 )    4343   4.343%   4.343% #
[      30,      35 )   13487  13.487%  17.830% ###
[      35,      40 )   26215  26.215%  44.045% #####
[      40,      45 )   31339  31.339%  75.384% ######
[      45,      50 )   11706  11.706%  87.090% ##
[      50,      60 )    6352   6.352%  93.442% #
[      60,      70 )    1843   1.843%  95.285%
[      70,      80 )     763   0.763%  96.048%
[      80,      90 )     460   0.460%  96.508%
[      90,     100 )     298   0.298%  96.806%
[     100,     120 )     382   0.382%  97.188%
[     120,     140 )     210   0.210%  97.398%
[     140,     160 )     208   0.208%  97.606%
[     160,     180 )     269   0.269%  97.875%
[     180,     200 )     380   0.380%  98.255%
```

这里是删除索引后写，与写时删除的比较。可以看到，两者的吞吐量和延迟都比较相近(没有前面read相差的那么大)。

这主要是因为我们的delete操作，仅仅只是删除元数据。而为什么偏低，是因为我们在删除元数据的时候，需要获取写锁，这就需要暂时将当前的writers队列的内容全部写入memtable，并且新的数据暂时不进入writers，因此会造成一定的性能损耗。这有点类似于pg数据库在创建索引时的准备过程，等待相关数据先全部写完，再继续操作。

FindKeysByField在有二级索引的情况下性能提升情况

吞吐量（Throughput）：每秒操作次数，单位为 OPS（operations per second）。通过迭代100000次，我们测试得到的吞吐量如下所示

```
[ RUN      ] Benchmark.WithThroughput
With Secondary_Index Throughput: 512820 OPS
[       OK ] Benchmark.WithThroughput (242 ms)
[ RUN      ] Benchmark.WithoutThroughput
Without Secondary_Index Throughput: 319488 OPS
[       OK ] Benchmark.WithoutThroughput (391 ms)
```

可以看到有二级索引的吞吐量为512820 OPS，而没有的话则为319488 OPS，有二级索引可以显著增大吞吐量。全表扫描是指数据库在没有索引的情况下，需要扫描表中的每一行来找到符合条件的记录。当表的数据量较大时，全表扫描的耗时会非常高，导致系统吞吐量下降。而二级索引允许数据库通过索引快速定位到目标记录，避免了大量的不必要扫描，从而减少了查询时间，提升了系统的吞吐量。

另一方面，**磁盘I/O**操作通常是数据库性能的瓶颈。二级索引通过减少需要读取的数据量，降低了磁盘I/O操作的频率。索引可以直接定位到目标数据的物理位置，避免了大量的随机磁盘读取操作，从而提高了系统的吞吐量。

延迟（Latency）：每个操作的平均响应时间，单位为毫秒（ms）。本来我们想采用平均延迟作为评判指标，但是因为字段查询的单词延迟非常小（要采用微秒才能勉强计算出来），所以最终通过迭代1000000次的总延迟时间作为标准，单位是微秒。

最终的结果如下所示：

```
[ RUN      ] Benchmark.WithLatency
With Secondary_Index Sum Latency: 1560757 microseconds
[       OK ] Benchmark.WithLatency (2166 ms)
[ RUN      ] Benchmark.WithoutLatency
Without Secondary_Index Sum Latency: 3054208 microseconds
[       OK ] Benchmark.WithoutLatency (3515 ms)
```

可以看到同样是迭代1000000次字段查询，如果有二级索引，延迟仅为1560757 微秒，但是没有的话就需要3054208 微秒，从延迟角度来看二级索引带来的提升也是巨大的。

与吞吐量层面的提升原因一致，二级索引在全表扫描和磁盘I/O角度都有着明显的优化。

build > testdb_write > main > LOG

```
2025/01/05-20:50:36.748829 140291890296576 Compacting 4@0 + 1@1 files
2025/01/05-20:50:36.918652 140291890296576 Generated table #16@0: 29580 keys, 2114799 bytes
2025/01/05-20:50:37.065784 140291890296576 Generated table #17@0: 29580 keys, 2114803 bytes
2025/01/05-20:50:37.203041 140291890296576 Generated table #18@0: 29580 keys, 2114802 bytes
2025/01/05-20:50:37.343265 140291890296576 Generated table #19@0: 29580 keys, 2114803 bytes
2025/01/05-20:50:37.490482 140291890296576 Generated table #20@0: 29580 keys, 2114798 bytes
2025/01/05-20:50:37.635251 140291890296576 Generated table #21@0: 29938 keys, 2113383 bytes
2025/01/05-20:50:37.783407 140291890296576 Generated table #22@0: 30031 keys, 2113992 bytes
2025/01/05-20:50:37.795063 140291890296576 Generated table #23@0: 2516 keys, 177180 bytes
2025/01/05-20:50:37.795209 140291890296576 Compacted 4@0 + 1@1 files => 14978560 bytes
2025/01/05-20:50:37.796598 140291890296576 compacted to: files[ 0 8 1 0 0 0 0 ]
2025/01/05-20:50:37.797797 140291890296576 Delete type=2 #7
2025/01/05-20:50:37.798285 140291890296576 Delete type=2 #9
2025/01/05-20:50:37.798785 140291890296576 Delete type=2 #11
2025/01/05-20:50:37.799256 140291890296576 Delete type=2 #13
2025/01/05-20:50:37.799693 140291890296576 Delete type=2 #15
2025/01/05-20:50:37.803575 140291890296576 Expanding@1 1+1 (2114799+3003487 bytes) to 8+1 (1497
2025/01/05-20:50:37.803699 140291890296576 Compacting 8@1 + 1@2 files
2025/01/05-20:50:37.980892 140291890296576 Generated table #24@1: 29642 keys, 2112798 bytes
2025/01/05-20:50:38.126634 140291890296576 Generated table #25@1: 29643 keys, 2112872 bytes
2025/01/05-20:50:38.276374 140291890296576 Generated table #26@1: 29641 keys, 2112735 bytes
2025/01/05-20:50:38.451144 140291890296576 Generated table #27@1: 29642 keys, 2112815 bytes
2025/01/05-20:50:38.625582 140291890296576 Generated table #28@1: 29642 keys, 2112816 bytes
2025/01/05-20:50:38.798829 140291890296576 Generated table #29@1: 29771 keys, 2113510 bytes
2025/01/05-20:50:38.947054 140291890296576 Generated table #30@1: 30090 keys, 2111413 bytes
2025/01/05-20:50:39.086716 140291890296576 Generated table #31@1: 30090 keys, 2111407 bytes
2025/01/05-20:50:39.149245 140291890296576 Generated table #32@1: 15124 keys, 1061309 bytes
2025/01/05-20:50:39.149389 140291890296576 Compacted 8@1 + 1@2 files => 17961675 bytes
2025/01/05-20:50:39.150946 140291890296576 compacted to: files[ 0 0 9 0 0 0 0 ]
2025/01/05-20:50:39.152182 140291890296576 Delete type=2 #5
2025/01/05-20:50:39.152620 140291890296576 Delete type=2 #16
2025/01/05-20:50:39.153051 140291890296576 Delete type=2 #17
2025/01/05-20:50:39.153489 140291890296576 Delete type=2 #18
2025/01/05-20:50:39.153940 140291890296576 Delete type=2 #19
```

可以看到第一次合并的写放大为14978560字节，第二次为17961675字节，两次总计写放大为32940235字节

其他实现的思路

我们在multi_db.h中新增对象

```
std::vector<std::string> new_index_field_;//用来存储当前正在创建的对象
MultiWriteBatch *tmp_index_batch_;//用来临时存放需要输入到新的indexdb的新输入的数据，
```

然后对于put_with_field，提前释放读锁,而不是等到write函数结束后才释放，然后当!new_index_field.*empty()时，意味着当前正在创建新的数据库，对于其数据，我们将它放入tmp_index_batch*中，然后继续正常的将其他数据输入到maindb或者其他index数据库。

```
meta_lock_.ReadLock();
std::unordered_set<int> match;
for (int i = 0; i < fields.size(); i++) {
    for (const auto& entry : index_db_mp_) {
        const auto& i_name = entry.first;  // 字段名称
        if (fields[i].first == i_name) {
            // 存储字段 i 和对应的数据库
            match.insert(i);
            break;
        }
    }
}
Status s;
MultiWriteBatch multi_batch;
for(auto &idx_pos : match){
    //item entry:  <index positon in fields,>
    std::string composed_key;
    BuildComposedKey(key,fields[idx_pos].second,&composed_key);
    if(!new_index_field_.empty() && (fields[idx_pos].first.ToString() == new_index_field_[0])){
        tmp_index_batch_->Put(composed_key,Slice(),fields[idx_pos].first.ToString());
    }else{
        multi_batch.Put(composed_key,Slice(),fields[idx_pos].first.ToString());
    }
}
meta_lock_.ReadUnlock();
Write()
```

然后对于createindex，也提前释放锁

```
Status MultiDB::CreateIndexOnField(const std::string &field_name) {

    //meta_lock_.WriteLock();
    WriteLockGuard l_meta(&meta_lock_);
    MutexLock l(&mutex_);
    this->db_status_ = DBStatus::CreatingIndex;
    //assert(writers_.empty());
    //WriteLockGuard l_meta(&meta_lock_);//主要针对PutwithField操作能够解析完整的multi_batch
    //assert(0);
    for (const auto& field_pair :  this->index_db_mp_) {
        if (field_pair.first == field_name) {
            return Status::InvalidArgument(field_name,"Index already exists for this field");
        }
    }

    //元数据持久化
    //..............

    new_index_field_.push_back(field_name);
    meta_lock_.WriteUnlock();
```

对应性能

```
writeAfterCreateIndex :      51.501 micros/op;    1.5 MB/s
Microseconds per op:
Count: 100000  Average: 51.5006  StdDev: 63.90
Min: 34.0000  Median: 46.1637  Max: 14470.0000
------------------------------------------------------
[      30,      35 )       1   0.001%   0.001%
[      35,      40 )   11774  11.774%  11.775% ##
[      40,      45 )   33719  33.719%  45.494% #######
[      45,      50 )   19360  19.360%  64.854% ####
[      50,      60 )   22343  22.343%  87.197% ####
[      60,      70 )    7584   7.584%  94.781% ##
[      70,      80 )    2255   2.255%  97.036%
[      80,      90 )     763   0.763%  97.799%
[      90,     100 )     245   0.245%  98.044%
[     100,     120 )     235   0.235%  98.279%
[     120,     140 )     129   0.129%  98.408%
[     140,     160 )     266   0.266%  98.674%
[     160,     180 )     391   0.391%  99.065%
```

```
writeInCreateIndex :      53.566 micros/op;    1.9 MB/s
Microseconds per op:
Count: 100000  Average: 42.2347  StdDev: 66.30
Min: 21.0000  Median: 41.5085  Max: 10966.0000
------------------------------------------------------
[      20,      25 )    8643   8.643%   8.643% ##
[      25,      30 )   19326  19.326%  27.969% ####
[      30,      35 )    5510   5.510%  33.479% #
[      35,      40 )    8316   8.316%  41.795% ##
[      40,      45 )   27196  27.196%  68.991% #####
[      45,      50 )   12935  12.935%  81.926% ###
[      50,      60 )   11799  11.799%  93.725% ##
[      60,      70 )    3359   3.359%  97.084% #
[      70,      80 )    1022   1.022%  98.106%
[      80,      90 )     333   0.333%  98.439%
[      90,     100 )     160   0.160%  98.599%
[     100,     120 )     139   0.139%  98.738%
[     120,     140 )     146   0.146%  98.884%
[     140,     160 )     269   0.269%  99.153%
[     160,     180 )     291   0.291%  99.444%
[     180,     200 )     212   0.212%  99.656%
[     200,     250 )     202   0.202%  99.858%
[     250,     300 )      59   0.059%  99.917%
[     300,     350 )      37   0.037%  99.954%
```

我们可以看到在createIndex的过程中，写的性能几乎不会受到影响。

延迟的表现反而更好，前者的p75在50-60之间，后者的p75在45-50之间。前者的p99在160左右，后者则在140左右。
这主要是因为，对于第一个put操作，它是以每一条数据为单位，然后写入两个数据库；而对于后者，它在建立index时，对于新的写入数据，最后都是以batch形式写入的，说明批量操作更快。

设置全局日志

- 实际上我们可以发现，这里index log的主要作用是用于建立索引时崩溃恢复。若我们接受可以在createIndex崩溃时，需要重新扫描整个数据库的话，我们可以有下述操作。

- 我们只需要记录maindb对应的log，因为indexdb的数据实际上可以从当前log中解析出来。但是需要注意leveldb中的void DBImpl::RemoveObsoleteFiles() 函数，它会自主删除过期数据，对于log数据是否过期的判断如下

```
for (std::string& filename : filenames) {
  if (ParseFileName(filename, &number, &type)) {
    bool keep = true;
    switch (type) {
      case kLogFile:
        keep = ((number >= versions_->LogNumber()) ||
                (number == versions_->PrevLogNumber()));
        break;
```

因此我们这里提出全局log日志记录方式，maindb的对应log将不再放入maindb的目录下，而是放在一个新建的LOG目录下。

global_log_file_manager是一个全局变量，用于管理文件的引用数，管理文件号，并且管理什么时候删除对应的log文件。当reference减到1时，开始删除文件。

```
namespace leveldb{
    class LogFileManager {
    public:
//        struct LogFileInfo {
//            uint64_t log_number;
//            std::string log_dir;   // 日志文件所在的目录
//            int ref_count;         // 引用计数
//        };
        void AddLogReference(uint64_t log_number);
        void AddLogReference(uint64_t log_number,uint64_t count);
        bool RemoveLogReference(uint64_t log_number);
        void unLock();
        void Lock();
        void clear();


        uint64_t getNewLogNumber() const;

        void setNewLogNumber(uint64_t newLogNumber);

        std::string log_dir;
        std::vector<uint64_t> log_to_delete_;
        std::atomic<uint64_t > new_log_number;//只在maindb中修改，在其他db中读取
        bool is_update_to_indexes_;
    private:
        port::Mutex mutex_;
        std::unordered_map<uint64_t , uint64_t> log_files_;//表示 <log_number,ref count>

        std::unordered_set<uint64_t> new_log;//在index刷盘前产生了多少个
    };
// 声明全局 LogFileManager 实例
    extern LogFileManager global_log_file_manager;
}
```

全局log的性能

```
fillseq      :      35.670 micros/op;    2.2 MB/s
fillsync     :    2405.940 micros/op;    0.0 MB/s (100 ops)
fillrandom   :      40.660 micros/op;    1.9 MB/s
readAfterCreateIndex :     168.770 micros/op; (100 of 100000 found)
readInCreateIndex :   10390.280 micros/op; (100 of 100000 found)
Index on 'address' created successfully.
writeAfterCreateIndex :      47.066 micros/op;    1.7 MB/s
Index on 'address' delete successfully.
writeAfterDeleteIndex :      27.564 micros/op;    2.8 MB/s
Index on 'address' created successfully.
writeInCreateIndex :      44.171 micros/op;    1.8 MB/s
Index on 'address' delete successfully.
writeInDeleteIndex :      28.629 micros/op;    2.7 MB/s
```

可以发现写入的性能明显比之前加快了。这里的writerInCreateIndex也是随机写，可以发现与fillrandom性能将比较接近。

待实现的优化

- 数据的验证

我们在验证时，是根据值是否相等判断它是否过期的。实际上，我们最好是根据sequence判断。但是，我们是很难做到两个数据库的sequece完全一致的。难点如下，存在maindb建立后插入索引的情况，这样就需要读取maindb对应的数据的sequence，然后手动对version设置sequence，也因为每一条数据的sequence不一定相等，因此无法进行批量写，性能堪忧。

因此，我们提出了以空间换时间的写法，原先的设计下，index的key值为val_len|value|p_key，我们可以进一步修改为 val_len|value|p_key|p_sequence,这样每一次除了记录实际数据，还会记录sequence用来加速validate操作。

具体操作则是，对于其sequence，传给ReadOptions操作，告知不能读取超过该sequence的数据。

待实现的优化

batch细粒度写

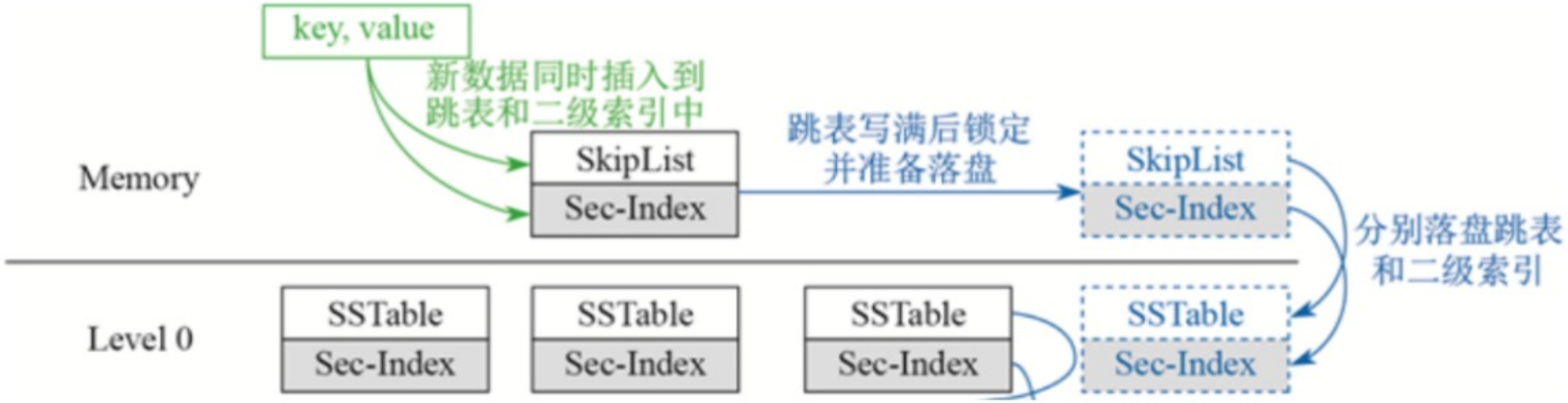
- 我们对于在createindex过程的非阻塞写，实现中，比较粗暴的把数据放到一个batch内，然后统一write，这样过于暴力，后续可以先进行一个大小的粗粒度的判断，然后决定是不是要先写一部分。

待实现的优化

- 在读取未命中的数据，然后indexdb内写delete信息时，可以是异步进行的，这样就可以不用增加在未命中数据时，read的延迟。

并且该部分也可以不进行log记录，可以加快写入速度。带来的影响则是，在数据库崩溃后，不命中的数据不会删除，还会有人读到过期数据，增加该次操作的读延迟。

与之前设计文档的比较



我们原先的打算是，在memtable中维护本地索引，然后在memtable刷盘的时候，该部分也刷盘，这样的好处是，很容易仅仅通过一次log写，保证数据的一致性。看上去比我们为两者分别写log文件要快很多。

当我们index对应的memtable是写满后才刷盘的。可以假设主键对应的数据字节长度是我们index key对应的字节长度的10倍，10次可以写满一个主memtable。也就是说，对于相同的数据量，我们的index的开销是，10次维护log的IO以及一次刷盘IO。而本地索引则是，10次刷盘IO。后者相当于少了一次IO。

但是，我们需要考虑两者的潜在性能差，我们会在序列化value的同时，序列化index对应的key，而这种方法在刷盘时还需要再次序列化一次，这种增加了cpu的cache miss。另一种情况则是，本地索引这种方法会使得level0的文件数量变得很大，就很容易触发compact操作(写放大)，同时当level0过多时，还会使线程进入睡眠状态。