数据库第一次报告兼总体概述

本实验是综合3次作业一起的角度去设计的，我想要达到的目的是尽可能具备可以和开源的sql环境相接的能力，同时又要尽可能的简单（考虑到时间问题），权衡了很多因素，维护并发数据库的代价太大，个人作业仅支持单线程，但由于rust语音的特殊性，以后扩展为并发也不难，我尽可能把内容分成3次。

支持的数据类型

在写之前有过很多的想法，最终还是决定仅支持定长记录，即不支持varchar类型，这样可以提高性能，方便设计，并且不影响数据库的完备性和可用性，只是不太适合一些使用场景,支持的数据类型如下：

```
pub enum DATATYPE{
    U8(u8),
    U16(u16),
    U32(u32),
    U64(u64),
    I8(i8),
    I16(i16),
    I32(i32),
    I64(i64),
    USIZE(usize),
    ISIZE(isize),
    F32(f32),
    F64(f64),
    STR(Db_str)
}
```

其中Db_str是特别设计的针对char（?）的数据结构

```
pub struct Db_str{
    capacity:usize,
    pub str:String
}
```

支持的类型均实现了以下的Attribute的trait，分别返回持久化的大小粒度，录入，和读取

```
pub trait Attribute {
    fn bin_size(&self) -> usize;

    fn encode(&self, buf: &mut [u8]);

    fn decode(&self, buf: &[u8]) -> Vec<u8>;
}
```

单个类型的序列支持的功能还不够强大，想要接应sql，还需要Data_item结构，按顺序把基本类型组合起来，当然，它也实现了Attribute的trait，其中每一个元组的bool类型代表是否该数据项的具体属性是否真的序列化，因为btree的update操作当中可能只要修改部分属性，未涉及的属性不需要改动， 通过bool标志可以判别，这样update操作可以和insert操作复用btree的同一个set接口，优化整个btree的操作

```
#[derive(Debug)]
pub struct Data_item{
    pub attributes: Vec<(DATATYPE,bool)>
}
```

```
fn encode(&self, buf: &mut [u8]) {
    check_len(buf, self.bin_size());
    let mut bin_size = 0;
    for attribute in self.attributes.iter() {
        if attribute.1 {//这是具体bool标志用法的体现，attribute.1就是上面说的bool标志，
决定具体属性是否序列化，有利于update和insert操作的复用set接口
            &attribute.0.encode(&mut buf[bin_size..]);
        }
        bin_size += &attribute.0.bin_size();
    }
}
```

Data_item_info结构还具有相应基本类型对应的属性名，并且它具有返回相应的无属性名信息的Data_item结构，这样才真正的和sql语句的信息可以对应起来

```
#[derive(Debug)]
pub struct Data_item_info{
    pub attributes: Vec<(DATATYPE,String,bool)>//String为属性名
}
```

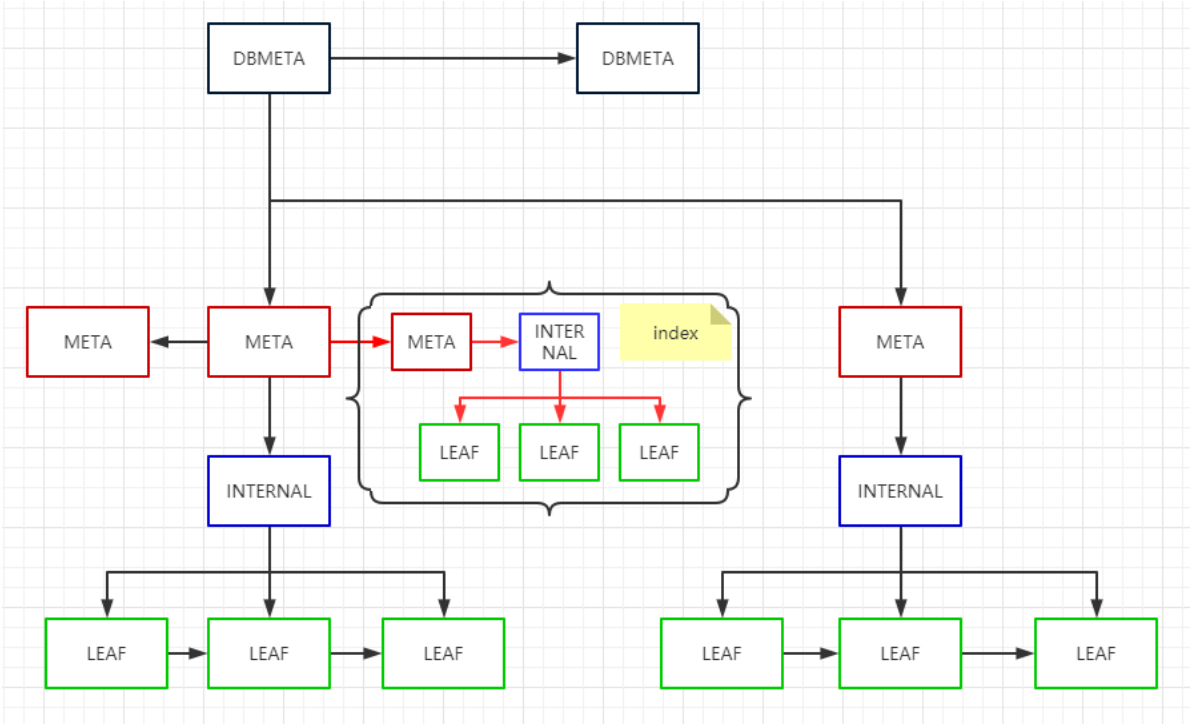
记录格式和页的设计

在谈论这个问题前，应该先说明，本设计采用与innodb引擎相识的方式，表数据内容内嵌b+树内，索引内容作为单独的b+tree，且值为主b+tree的primary_key，强制有primary_key，如果create table的时候未设定，则自行加入隐藏自增属性（u32）

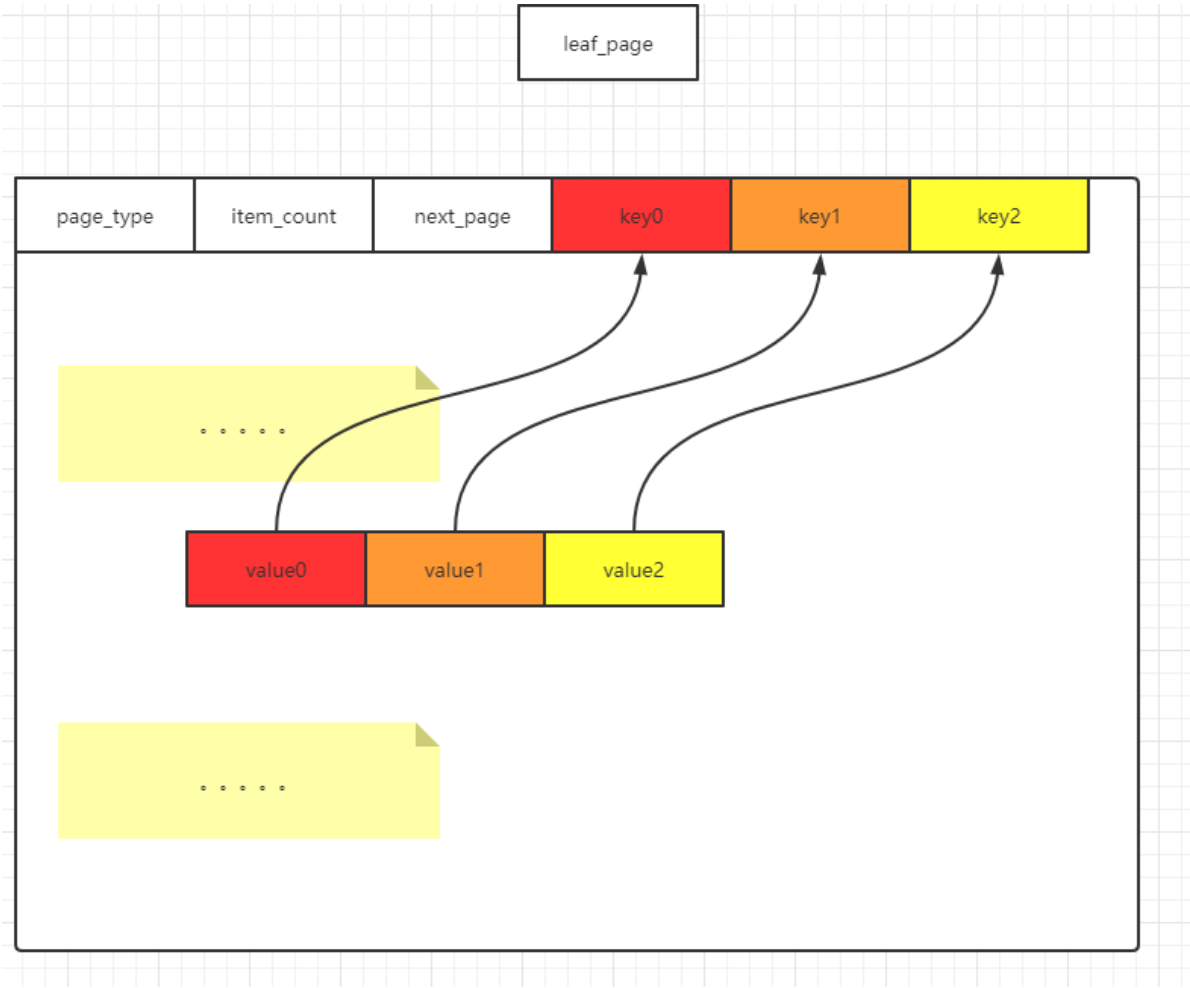
可以知道：每个表有属性名，属性类型，索引信息，是否有隐藏自增属性（如果有，当前的自增值是多少）等元信息，整个数据库有表的元信息，当然可以用单独的表来存储这些元信息，而我这里采取的策略不是这样的，一个数据库的所有信息存储在同一个一个文件中

文件分页后有4种页,其中DB_META存储（【表名+表META页指针】），META（顺序存放【属性名+属性类型】【索引信息+索引META页指针】），DB_META, META如果满了，则线性存储（有判满标志），默认第一页为DB_META页（且只有第一页的DB_META的total_page字段有效）。

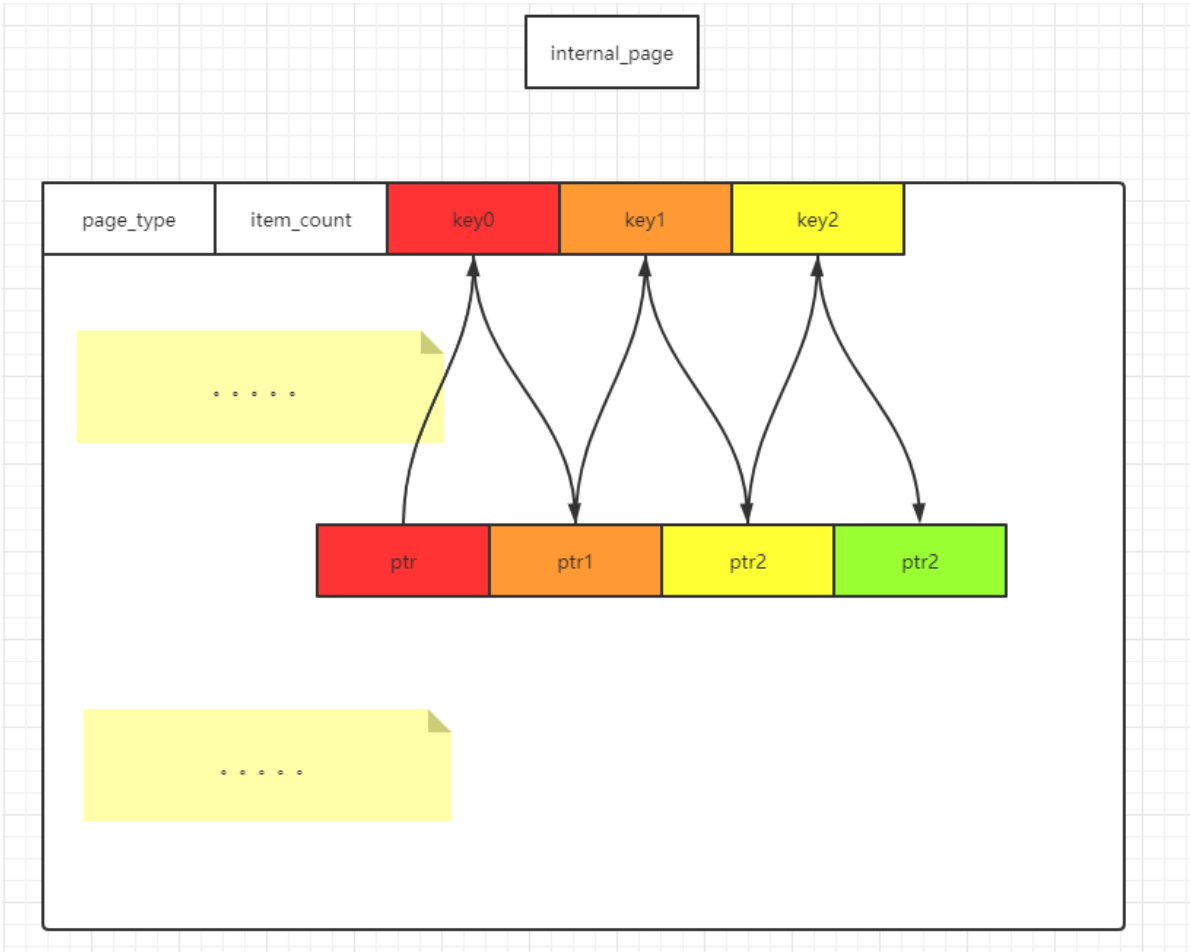
```
pub enum PageType {
    DB_META,
    META,
    INTERNAL,
    LEAF,
}
```



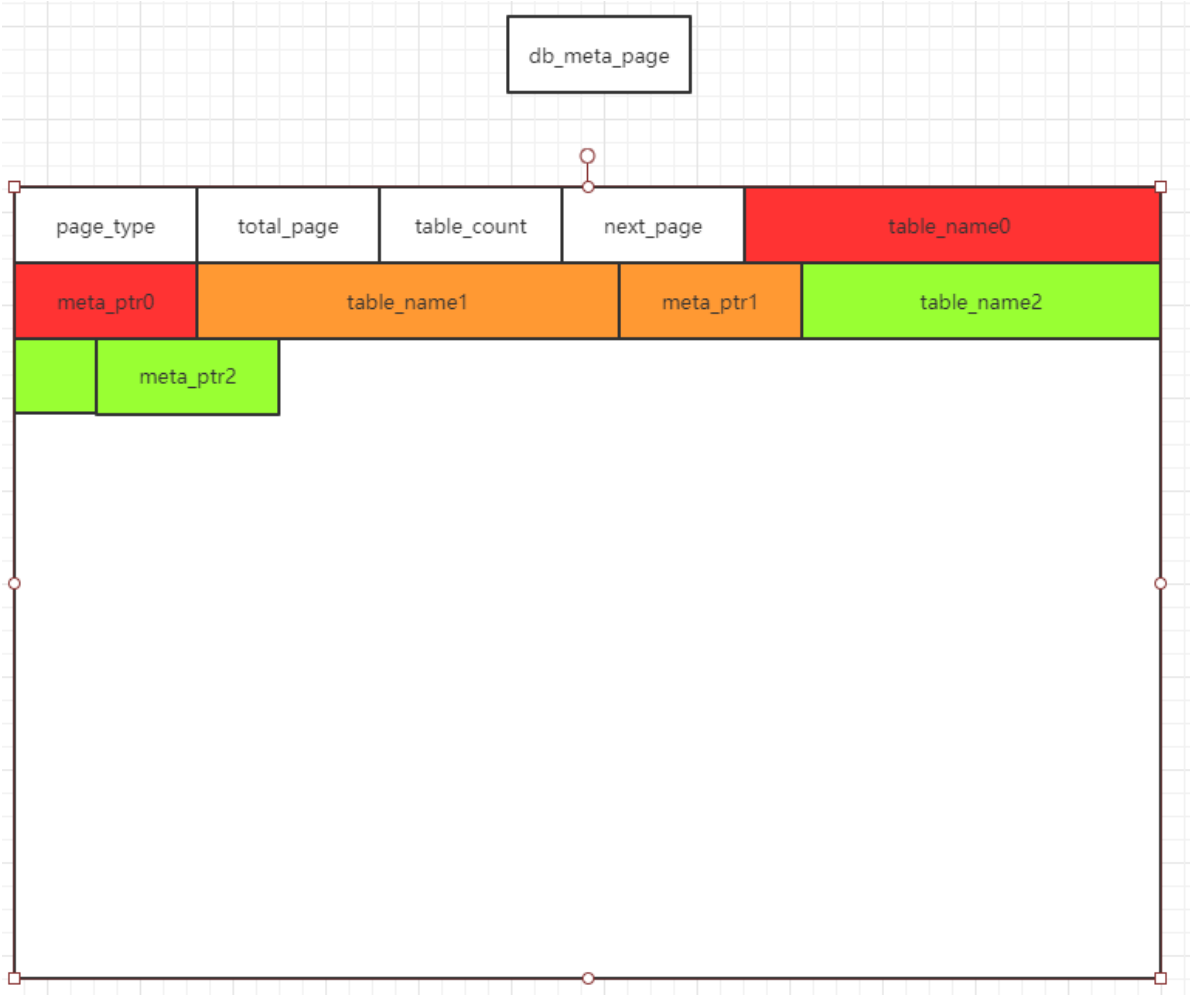
LEAF:



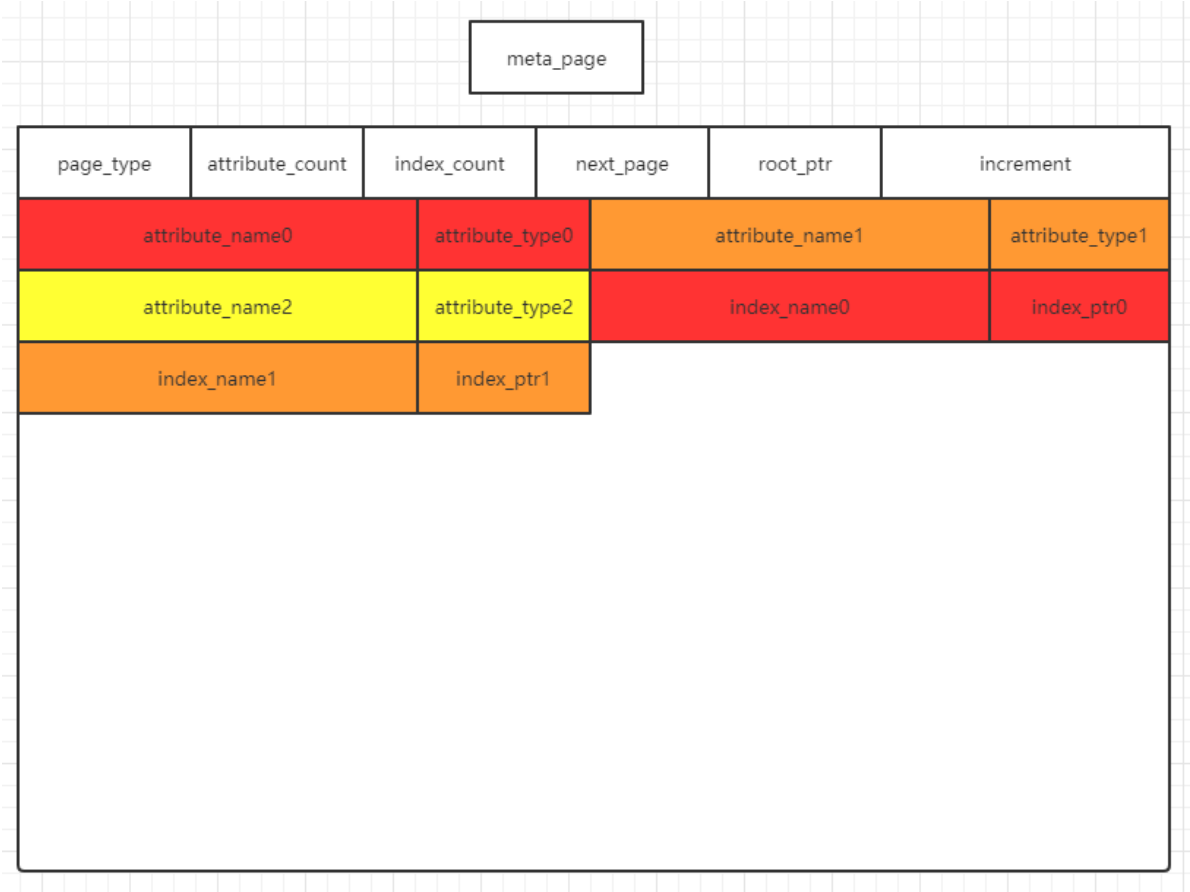
INTERNAL:


internal_page
page_type
item_count
key0
key1
key2
ptr
ptr1
ptr2
ptr2

DB_META:


db_meta_page
page_type
total_page
table_count
next_page
table_name0
meta_ptr0
table_name1
meta_ptr1
table_name2
meta_ptr2

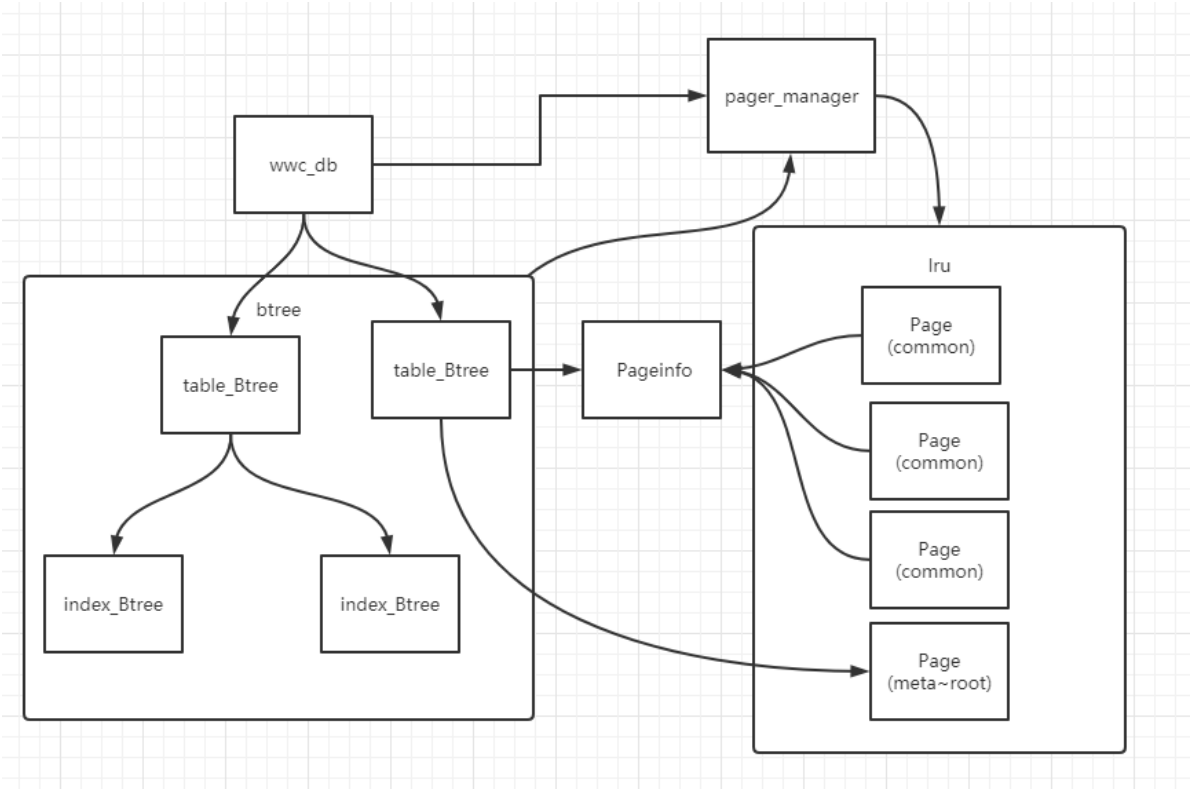
META:



DB_META,META的所有字节大小是固定的的，也限制表名的大小等，而INTERNAL,LEAF的数据字节是灵活大小的（~INTERNAL的ptr大小固定4字节~），根据页所属的Btree不同而改变，这些灵活的信息保存在相应meta页中，加载生成内存中对应的Pageinfo结构（叶子节点的max_item_num等）

整体对象的关系

画的非常的粗略，最好是看源码理解关系



为了方便理解数据体之间的关系，虽然有些繁杂，但是下面还是列出了每一个涉及到的重要对象的数据属性信息：

```
pub struct wwc_db{
    pager_manager: Rc<RefCell<Pager_manager>>,
    pub table_btrees: HashMap<String,Rc<RefCell<BTree>>>
}
```

```
#[derive(Debug)]
pub struct BTree
{
    // fd: Rc<RefCell<File>>,
    pager_manager: Rc<RefCell<Pager_manager>>,
    // keys_pos: usize,
    // keys_internal_pos: usize,
    // values_pos: usize,
    // ptrs_pos: usize,
    // max_item_count: usize,
    // max_index_count: usize,
    pager_info: Rc<RefCell<Page_info>>,
    // _key: Data_item,
    // _value: Data_item,
    // meta_page: Option<Page>,
    root_page: Rc<RefCell<Page>>,
    meta_page_index: u32,//元数据页其实不太用，不必拿page引用，拿记录页号就行，减少缓存压力
    index_btrees: Option<HashMap<String,Rc<RefCell<BTree>>>>
}
```

```
#[derive(Debug)]
pub struct Page_info{
    keys_pos: usize,
    keys_internal_pos: usize,
    values_pos: usize,
    pub ptrs_pos: usize,
    max_item_count: usize,
    max_index_count: usize,
    _key: Data_item_info,
    _value: Data_item_info,
    key_size: usize,
    value_size: usize
}
```

```
#[derive(Debug)]
pub struct Page
{
    pub index: u32,
    pub(crate) buf: [u8; PAGE_SIZE],
    pub page_type: PageType,
    // keys_pos: usize,
    // values_pos: usize,
    // ptrs_pos: usize,
    // max_item_count: usize,
    dirty: bool,
    pub(crate) fd: Option<Rc<RefCell<File>>>,
    pub page_info: Option<Rc<RefCell<Page_info>>>
}
```

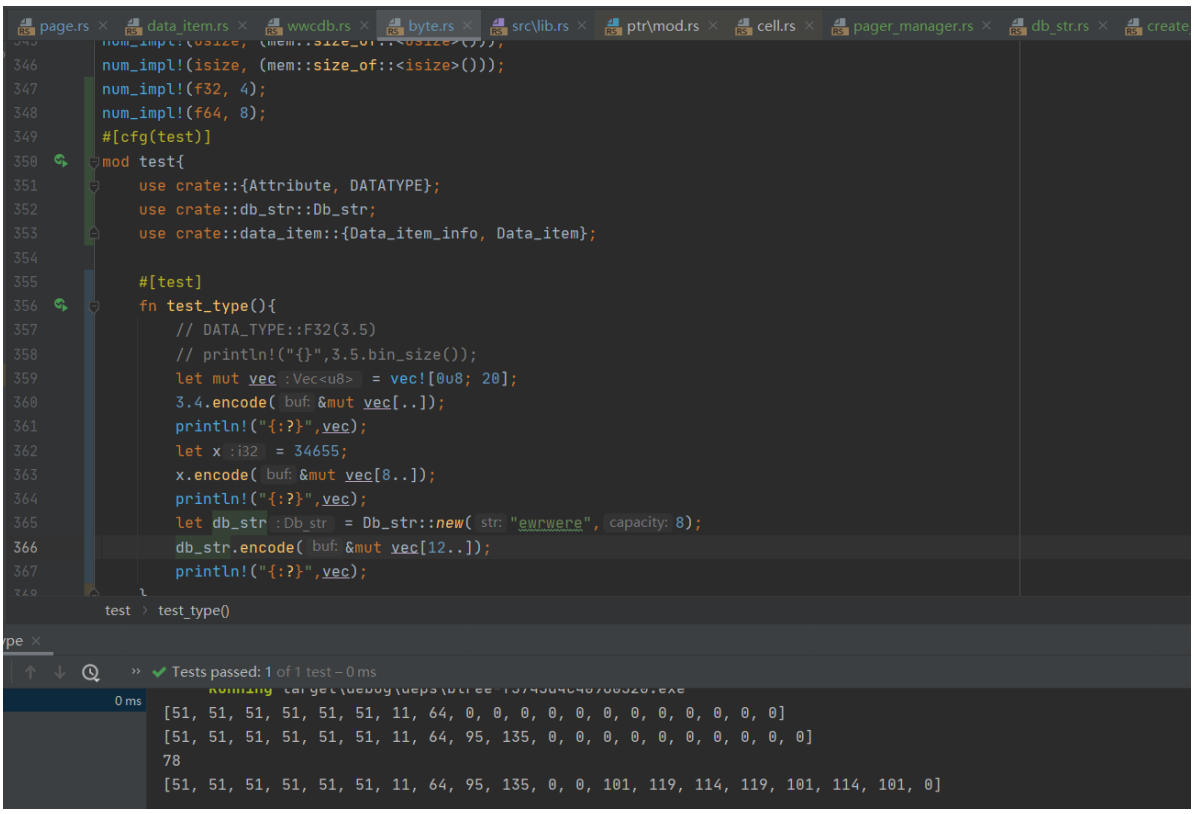
```rust
#[derive(Debug)]
pub struct Pager_manager{
    lru:LruCache<u32,Rc<RefCell<Page>>>,
    pub db_meta_page: Rc<RefCell<Page>>,
    fd: Rc<RefCell<File>>
}
```

```rust
pub struct LruCache<K, V, S = DefaultHasher> {
    map: HashMap<KeyRef<K>, Box<LruEntry<K, V>>, S>,
    cap: usize,

    head: *mut LruEntry<K, V>,
    tail: *mut LruEntry<K, V>,
}
```

测试部分

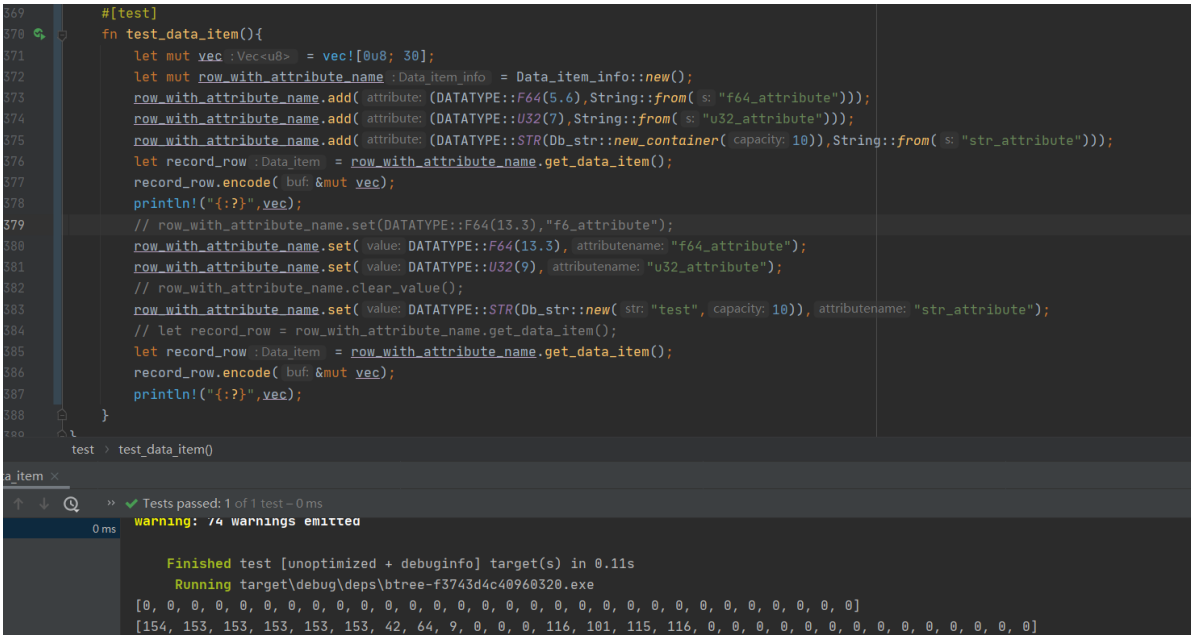
类型测试：

将f64，i32, char(8)在分别在内存中映射到vec数组的不同位置上，结果显示成功



综合数据项测试：

综合数据项测试，根据属性名设置值，并整个数据项映射到vec，结果显示成功



数据库第二次报告

缓冲管理的实现

lru缓冲的实现并不难，但是实现一个性能良好的多线程的缓冲却比较的困难，因为多线程的的缓冲必须加入引用计数的概念，缓存页一旦被获取被某个线程获取，在该线程主动释放该页前，必须确保该缓存页还在内存中，即不能被释放掉。

我对缓冲池的实现的参照了level_DB的源码：其内部自我实现了自动扩增的hash_table，除此之外，还需要维护两个双向链表used_list，freed_list,其中used_list的链表，每一个缓存页除了要放在hash_table上，还需要根据缓存页是否被正在被某些线程引用而将其放在used_list，或freed_list上，而采取替换策略时，会优先淘汰freed_list上的页，如果freed_list为空，不得已要淘汰used_list上的页时，也仅仅是将其引用计数减一，并不会真正的free掉，而是将free的工作交给引用它的最后一个线程在归还引用时调用deletehandler闭包释放。

因此从某种意义上来说，就算你给一个缓冲池固定的大小，真正在内存上的缓冲页的数目上限也可能是动态的，缓冲的目的不在于限制其数量固定，更多的意义在于防止两种极端：每次不释引起的内存爆炸，有或是次次都释放产生的IO代价。

level_DB的另一个令有趣的地方是，它的缓冲池并不是一个，而是16个，然后用一个统一的ShardedLRUCache去管理16个 LRUCache，这样的好处是作为一个多线程的缓冲池，访问必然需要mutex，会影响一定的效率，而如果事先根据key求hash，在根据hash%16到属于自己的key所在的LRUCache去获得mutex，这样就两个线程就不一定会发生锁等待了，因为他们访问的可能是不同的LRUCache，极大的提升了效率。

rust的特点导致了它不用特意去维护引用计数，实现起来非常的方便，不需要有很高的指针技巧。

服务器内存空间的划分

题目要求把空间划分成执行过sql的访问计划、数据字典信息、数据处理缓存、日志缓存等。

sql的访问计划是以执行结果的形式存在的。

而数据字典信息是以对象结构的形式存在的，比如Bree，Page_info等结构。

数据处理缓存是以pager_manager的成员lru（下图）存在的。

而日志缓存并没有实现，因为现在没有打算实现并发，也不提供单线程的事务机制，会提供exit（）接口安全退出（会把缓冲中的脏页全flush），如果通过别的方式退出程序（包括意外断电等），不保证内容的持久化

```rust
pub struct Pager_manager{
    lru:LruCache<u32,Rc<RefCell<Page>>>,
    pub db_meta_page: Page,
    fd: Rc<RefCell<File>>
}
```

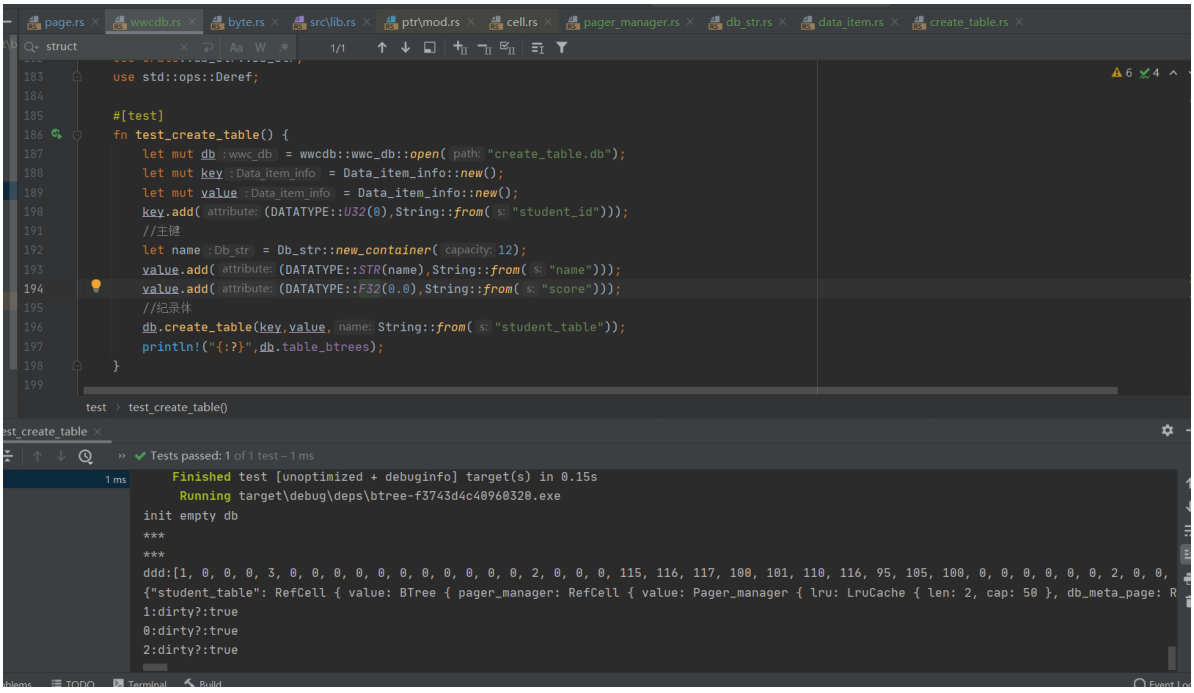
测试部分：

页缓存测试（通过创建数据库实例）：

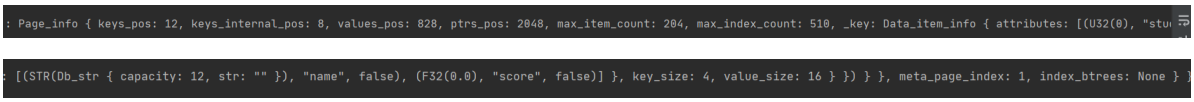
关于缓存的测试，由于缓存层已经嵌入到设计中，并没有专门的测试部分，这里通过创建一个新的数据库来演示它的功能。

这里利用create_table.db这个文件创建一个数据库，因为事先不存在这个文件，所以是创建新文件并运行“init empty db”这条路径，（由于debug打印信息时把4096字节的具体页信息全打印了，所以不方便给查阅），注意到“lru: LruCache { len: 2, cap: 50 }”的打印信息，表示缓存支持50个缓存页大小，目前有仅有2个页，（不要奇怪，为什么程序退出时显示0，1，2三个页，而这里只有2个页，因为我的程序没有把0页放在lru里面，因为考虑它的调用非常频繁，直接将它作为 Pager_manager的一个成员，所以程序退出时虽然自动调用了3个页的drop闭包[判断是否为脏页，若是则回写磁盘]，这里这3个页都是脏页，但是lru的len为2是合理）

下面解释一下为什么是3页，创建数据库时，db_meta（0）页，这就创建了student_table的表，所以还得meta（1）页和root（2： leaf）页，所以一共3页

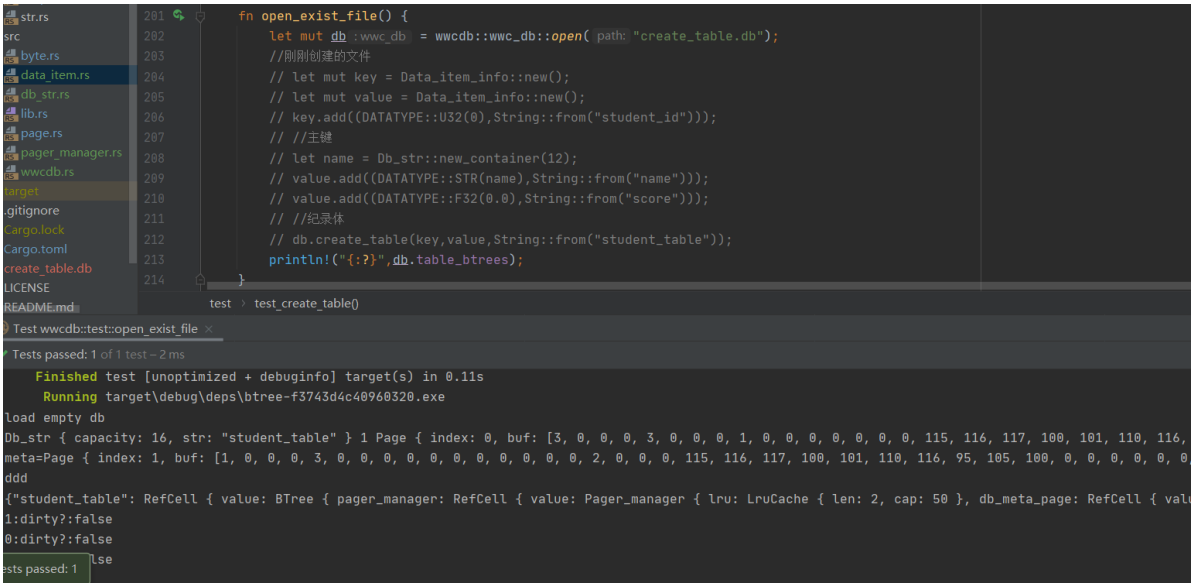


这里虽然是验证缓存的测试，但还是把上面的测试的创建的表的有关的动态的page_info信息打出来，之后的测试将不再显示关于 page_info截图



可知，lru工作正常，测试成功

加载数据库实例测试



紧接着上次的测试，这次打开的上次创建的数据库，走的是“load empty db”路线，所得到的信息与预期一致，没有出现差错，测试成功

数据库第三次报告

重复key的b+tree问题

对于内嵌有table数据的以primary_key作为key的的table_btree当然没有重复key的问题

而对于index_tree就有key值重复的问题，解决的办法是提供不同于table_btree的操作接口和内部实现

因为index_btree只有插入，删除，查询，没有修改操作，因为修改时是只需要修改table_btree的内容就可以了

index_btree的插入不会覆盖原来已有的key，允许多个key，除非插入操作的key和value同时相等，则会发生覆盖

index_btree的删除操作不但需要key，还需要value，因为只有这样，才能唯一确定要删除的索引纪录

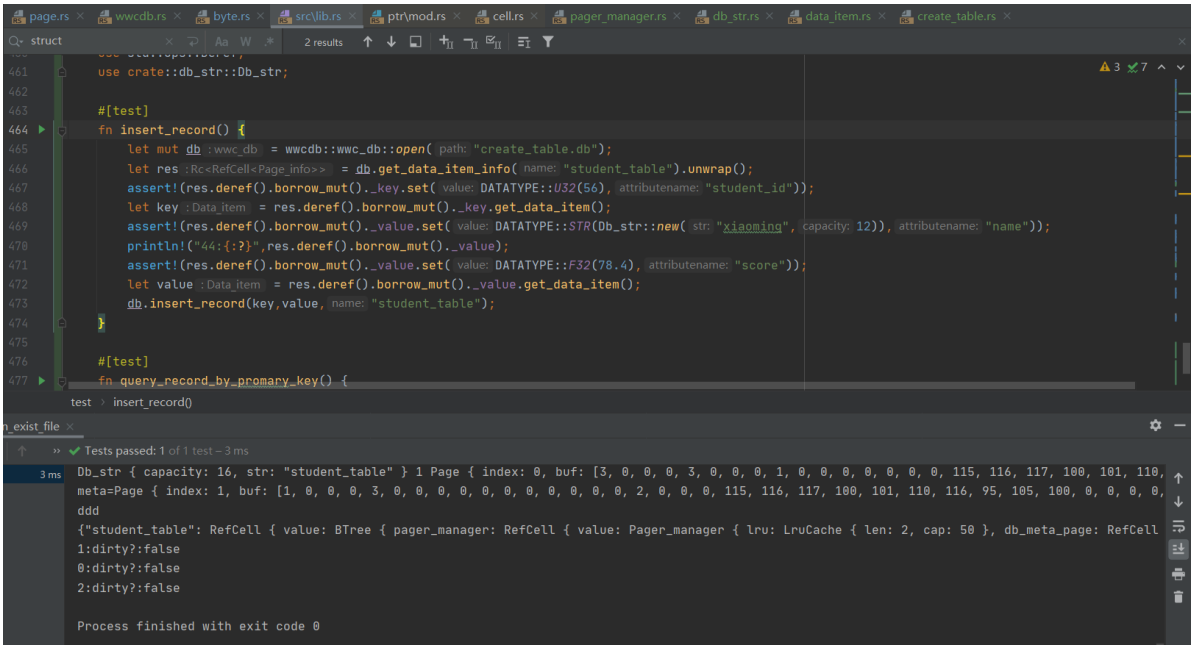
index_btree的即便查询单个key也可以放回多个value（Vec<Data_item>）

测试部分：

//由于，后期时间问题，暂时没有做没有index_btree，但并不影响作业要求

btree操作：不具体解释

插入：



```
use crate::db_str::Db_str;

#[test]
fn insert_record() {
    let mut db :wwc_db = wwcdb::wwc_db::open( path: "create_table.db");
    let res :Rc<RefCell<Page_info>> = db.get_data_item_info( name: "student_table").unwrap();
    assert!(res.deref().borrow_mut()._key.set( value: DATATYPE::U32(56), attributename: "student_id"));
    let key :Data_item = res.deref().borrow_mut()._key.get_data_item();
    assert!(res.deref().borrow_mut()._value.set( value: DATATYPE::STR(Db_str::new( str: "xiaoming", capacity: 12)), attributename: "name"));
    println!("44:{:?}",res.deref().borrow_mut()._value);
    assert!(res.deref().borrow_mut()._value.set( value: DATATYPE::F32(78.4), attributename: "score"));
    let value :Data_item = res.deref().borrow_mut()._value.get_data_item();
    db.insert_record(key,value, name: "student_table");
}

#[test]
fn query_record_by_promary_key() {
```

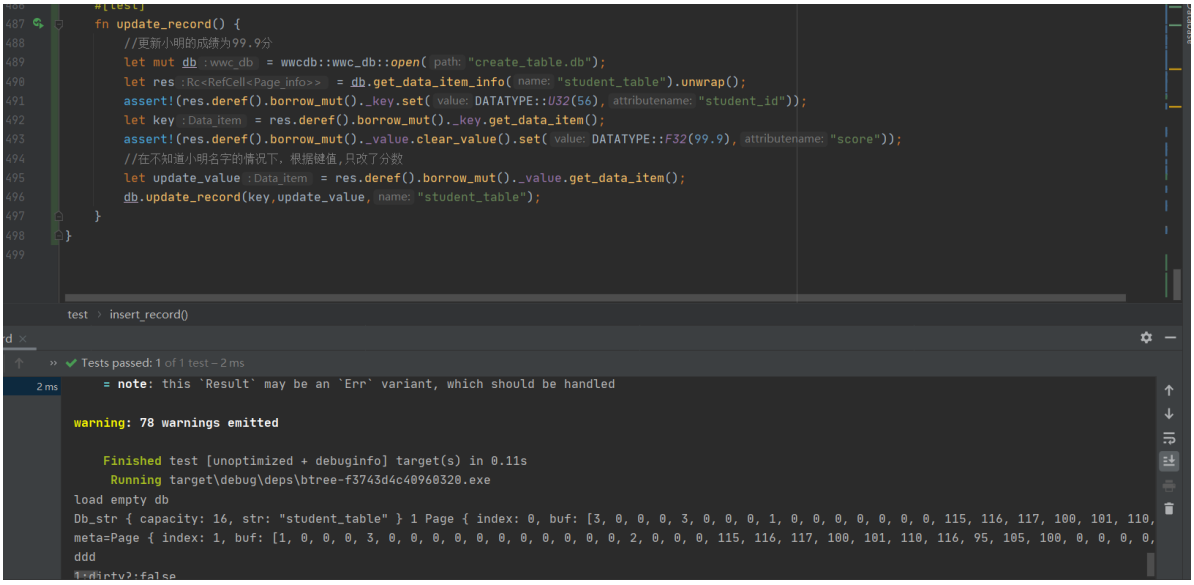
test › insert_record()

n_exist_file

Tests passed: 1 of 1 test – 3 ms

```
Db_str { capacity: 16, str: "student_table" } 1 Page { index: 0, buf: [3, 0, 0, 0, 3, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 115, 116, 117, 100, 101, 110,
meta=Page { index: 1, buf: [1, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 115, 116, 117, 100, 101, 110, 116, 95, 105, 100, 0, 0, 0, 0,
ddd
{"student_table": RefCell { value: BTree { pager_manager: RefCell { value: Pager_manager { lru: LruCache { len: 2, cap: 50 }, db_meta_page: RefCell
1:dirty?:false
0:dirty?:false
2:dirty?:false

Process finished with exit code 0
```

更新：



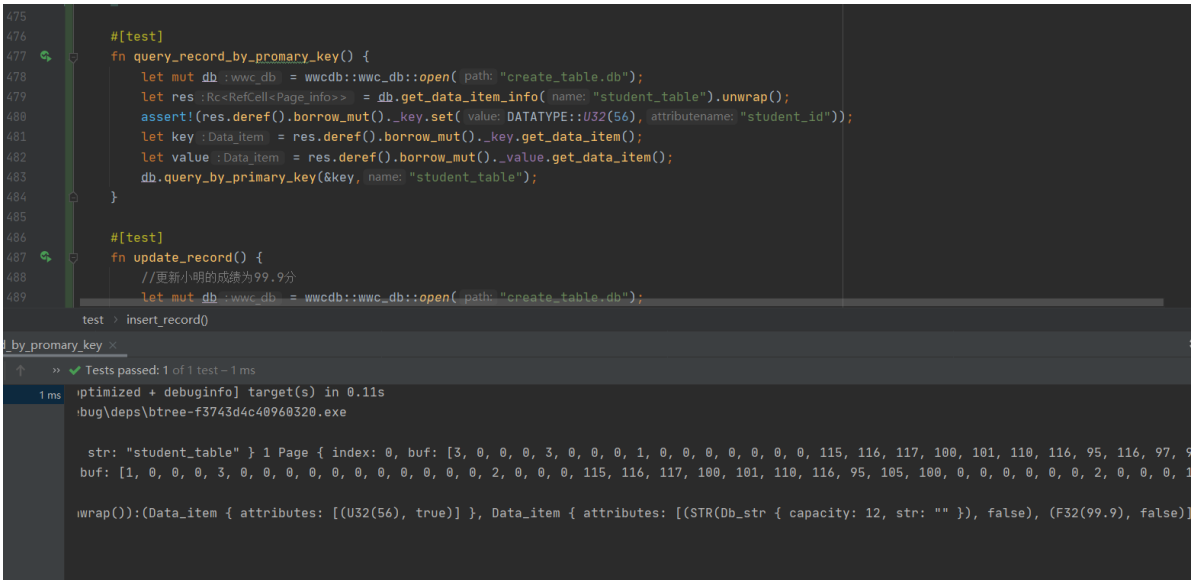
```
fn update_record() {
    //更新小明的成绩为99.9分
    let mut db :wwc_db = wwcdb::wwc_db::open( path: "create_table.db");
    let res :Rc<RefCell<Page_info>> = db.get_data_item_info( name: "student_table").unwrap();
    assert!(res.deref().borrow_mut()._key.set( value: DATATYPE::U32(56), attributename: "student_id"));
    let key :Data_item = res.deref().borrow_mut()._key.get_data_item();
    assert!(res.deref().borrow_mut()._value.clear_value().set( value: DATATYPE::F32(99.9), attributename: "score"));
    //在不知道小明名字的情况下，根据键值,只改了分数
    let update_value :Data_item = res.deref().borrow_mut()._value.get_data_item();
    db.update_record(key,update_value, name: "student_table");
}
}
```

test › insert_record()

Tests passed: 1 of 1 test – 2 ms

```
    = note: this `Result` may be an `Err` variant, which should be handled

warning: 78 warnings emitted

    Finished test [unoptimized + debuginfo] target(s) in 0.11s
     Running target\debug\deps\btree-f3743d4c40960320.exe
load empty db
Db_str { capacity: 16, str: "student_table" } 1 Page { index: 0, buf: [3, 0, 0, 0, 3, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 115, 116, 117, 100, 101, 110,
meta=Page { index: 1, buf: [1, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 115, 116, 117, 100, 101, 110, 116, 95, 105, 100, 0, 0, 0, 0,
ddd
1:dirty?:false
```

查询：



```
#[test]
fn query_record_by_promary_key() {
    let mut db :wwc_db = wwcdb::wwc_db::open( path: "create_table.db");
    let res :Rc<RefCell<Page_info>> = db.get_data_item_info( name: "student_table").unwrap();
    assert!(res.deref().borrow_mut()._key.set( value: DATATYPE::U32(56), attributename: "student_id"));
    let key :Data_item = res.deref().borrow_mut()._key.get_data_item();
    let value :Data_item = res.deref().borrow_mut()._value.get_data_item();
    db.query_by_primary_key(&key, name: "student_table");
}

#[test]
fn update_record() {
    //更新小明的成绩为99.9分
    let mut db :wwc_db = wwcdb::wwc_db::open( path: "create_table.db");
```

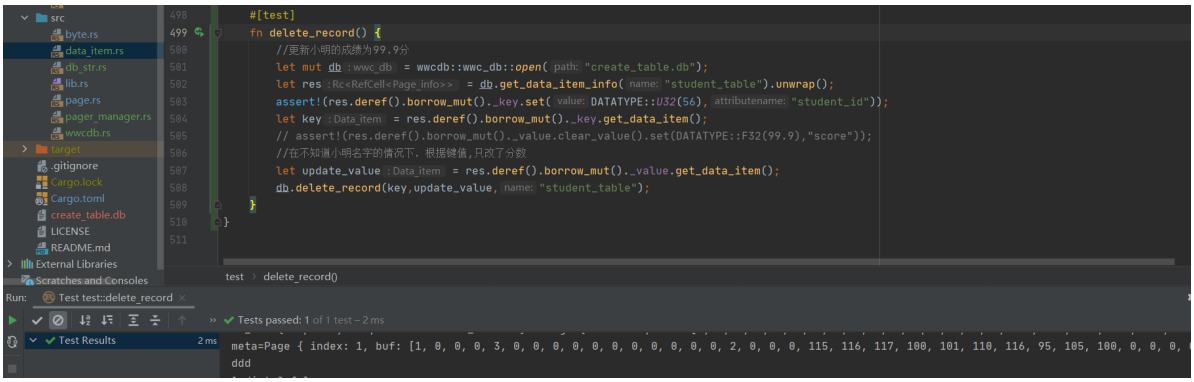
test › insert_record()

_by_promary_key

Tests passed: 1 of 1 test – 1 ms

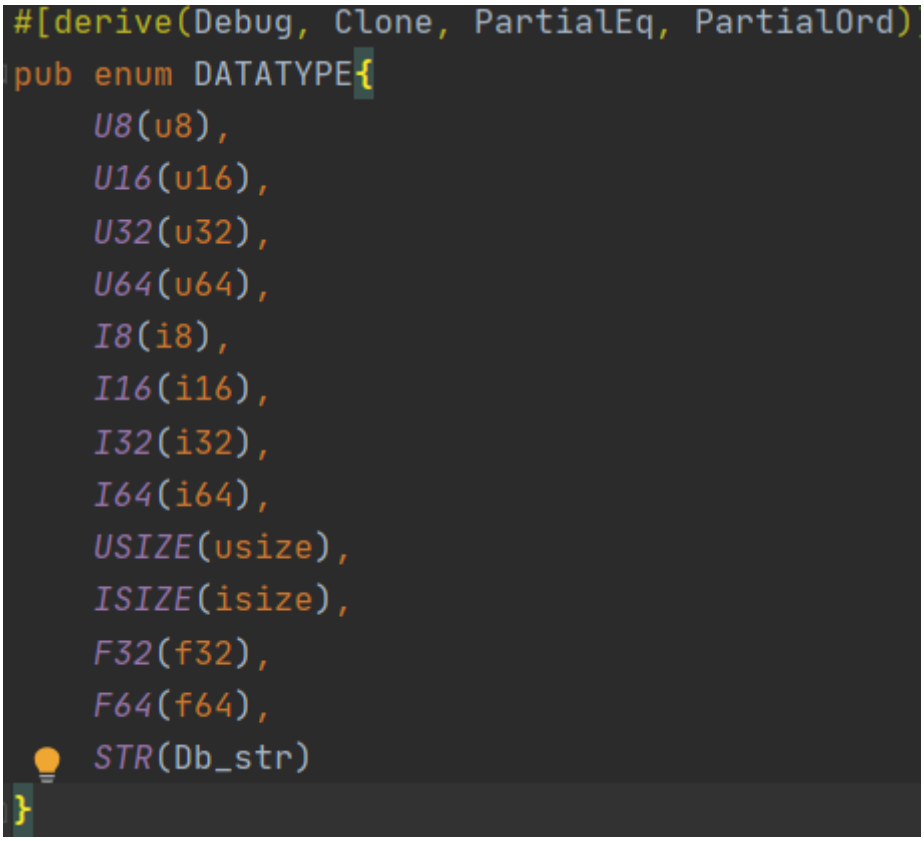
```
ptimized + debuginfo] target(s) in 0.11s
bug\deps\btree-f3743d4c40960320.exe

 str: "student_table" } 1 Page { index: 0, buf: [3, 0, 0, 0, 3, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 115, 116, 117, 100, 101, 110, 116, 95, 116, 97, 9
buf: [1, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 115, 116, 117, 100, 101, 110, 116, 95, 105, 100, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1

wrap()):(Data_item { attributes: [(U32(56), true)] }, Data_item { attributes: [(STR(Db_str { capacity: 12, str: "" }), false), (F32(99.9), false)]
```

删除：

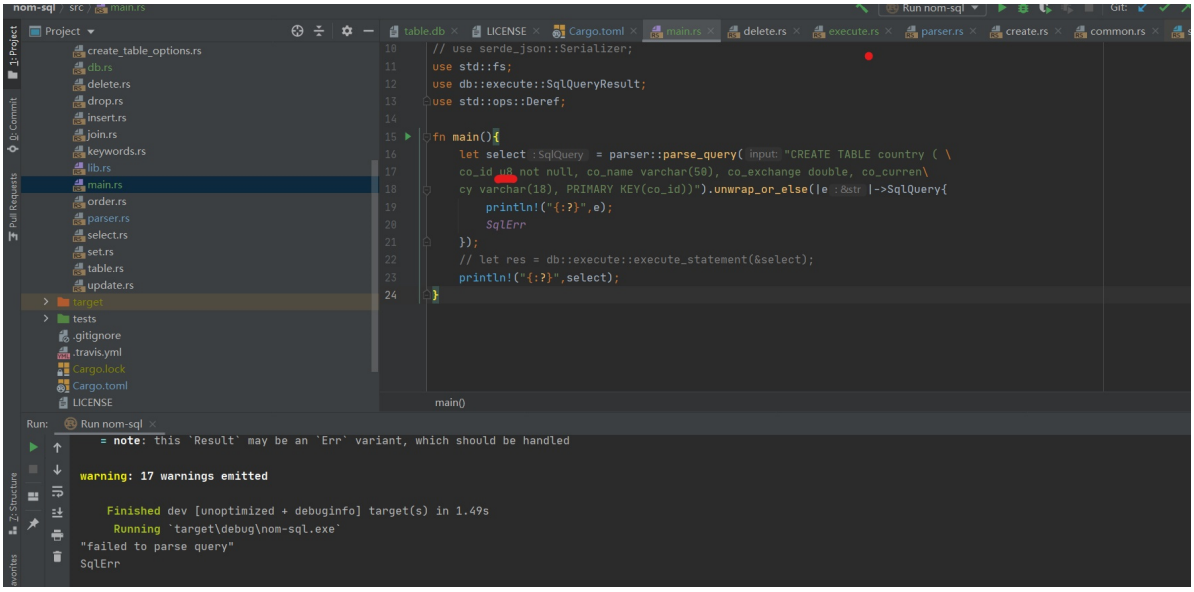


数据库第四次报告

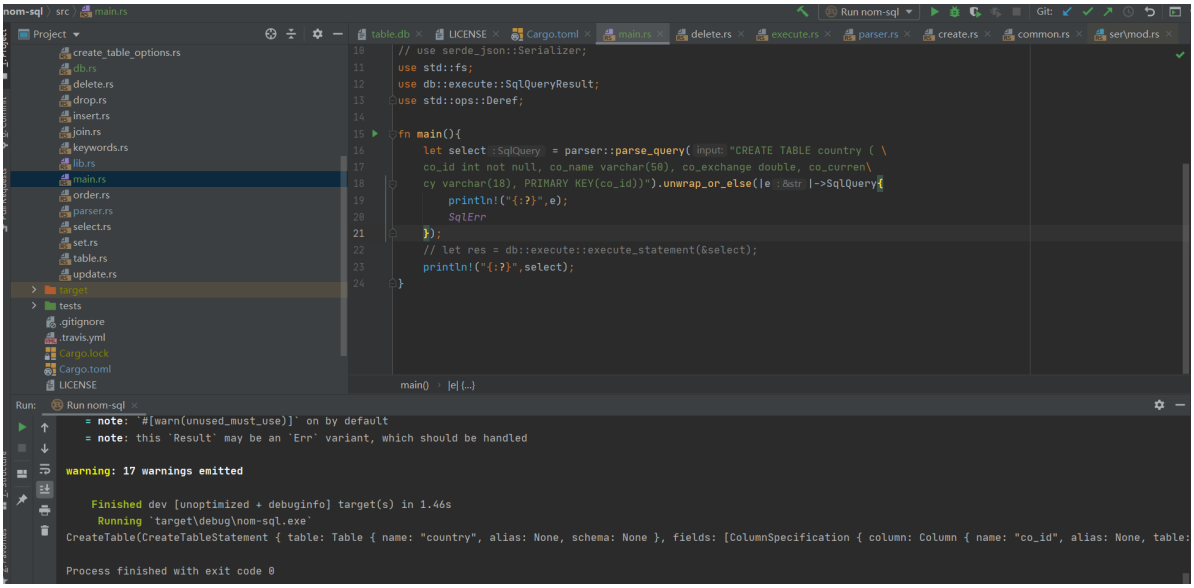
在做个之前，我了解很多sql的解析库，我遇到了一个很大的问题，让我想放弃这次的作业，我觉得我的数据库很难跟现有现有的sql融合，虽然我当初在设计之初就想到了这个问题，但是我考虑的不周全，我数据库正如我前面所讲述，类型是我自己定义的，也偷懒刚好是rust内置的类型和自己封装定义的Db_str（如下图）



当初没想到sql的解析是不支持动态类型的，否则就会报错，如下图，我拉了nom_sql库做测试，我用一个u8类型是无法解析的



而只有改成解析器默认的事先定义类型，比如int，才能解析成功，得到ast



我才发现我都忘记了sql都限定了数据类型，比如，nom_sql,或适配sqlite的lemon解析器的sql_parser都是，我甚至想着自己根据现有的nom包做一个解析器，结果代价太大了，最终也无疾而终，但是为此我付出了很多的代价。

我的现在这个程序前前后后接近2500行代码，很多东西都是自己想出来，直接根据理论写出来，很少有借鉴他人直接代码，有的话也算是凭着别的语言的逻辑（比如c），很大一部分原因是github上用rust写的一个简单且适合要求的项目实在太少，要么就是大部头的开源项目，实在不适合初学者学习借鉴，能实现一种自己头脑里的东西真的是很神奇的感觉

这样导致了我设计的数据库确实也存在自己设计的特色，也导致现在不能借用开源sql库的尴尬境遇，我不想欺骗老师强行用别的项目联合起来，应付报告，我也觉得自己在这门课程中的表现足够好

虽然不能继续实现下去，我本想的是利用别人写的数据库，并加以讲解，已作为对这次报告的弥补，但是，找了很多项目，并没有我想要的效果的项目，我发现网上很多项目其实都很水

所以作为弥补，我也不利用别人的项目了，我想直接较为详细的从理论讲述整个SQL的查询处理的过程

1)

首先我想讲述的是第一部分，针对DML语句，进行词法、语法、语义分析，输出语法分析树：

很庆幸，这个学期之前，我个人爱好学习了编译原理，没想到很快就能因为这课运用到实践中

词法分析是给词语定性，根据事先的规则（正则表达式）定义它的性质，比如234代表数字，select代表token（有语音的一个单位，事先确定好），“sfwe”可能就代表没有任何意义普通的字符串

语法分析是词法的token之间要有一定的位置序列规则，我个人认为它常常是和语义分析一起做的，所以这里就一起讲解

```
oneselect(A) ::= SELECT distinct(D) selcollist(W) from(X) where_opt(Y)
      groupby_opt(P) having_opt(Q) orderby_opt(Z) limit_opt(L). {
 A = sqlite3SelectNew(pParse,W,X,Y,P,Q,Z,D,L.pLimit,L.pOffset);
}
```

比如这个符合lemon解析器语法规则的部分select语法规则，大括号外部的部分就是语法规则的限定，而大括号内部的部分就是语义赋予分析的过程，从而可以得到最终的内存上的ast结构的语义结果，本人也曾经作为练习，用lemon写过一个计算器的语法规则，很有意思，除了lemon之外，还有很多词法和语法解释器，比如yacc，lex，bison等等之类的，但我仍认为lemon的语法是最友好的，他是sqlite只带的解析器，是这个数据库的作者专门写的一个解析器，由于历史并不古老，考虑到很多因素，它有更优势新颖的方面

解析的过程是很复杂的，但是sql的解析有很好的开源环境，现在做数据库的人，这一部分甚至可以不关心，独立于数据库的实现部分

2)

制定逻辑查询计划：把语法树转换成一个关系代数表达式，输出分析树对应的逻辑计划和优化后的逻辑计划：

关系代数种的符号其实都和ast解析结果挂钩，比如以刚才的nom_sql为例

nom其实的解析过程并没有真正意义上利用任何解析器，而是利用设计模式中的解释器模式，将式子递归分解，解释

以具有代表性的select语句为例，下图是nom_sql的源代码：以我用代码另外标注的语句路径为例

```
Select(SelectStatement)
```

```
#[derive(Clone, Debug, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub enum SqlQuery {
    CreateTable(CreateTableStatement),
    CreateView(CreateViewStatement),
    Insert(InsertStatement),
    CompoundSelect(CompoundSelectStatement),
    Select(SelectStatement),
    Delete(DeleteStatement),
    DropTable(DropTableStatement),
    Update(UpdateStatement),
    Set(SetStatement),
    SqlErr
}
```

```
pub where_clause: Option<ConditionExpression>
```

```rust
#[derive(Clone, Debug, Default, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub struct SelectStatement {
    pub tables: Vec<Table>,
    pub distinct: bool,
    pub fields: Vec<FieldDefinitionExpression>,
    pub join: Vec<JoinClause>,
    pub where_clause: Option<ConditionExpression>,
    pub group_by: Option<GroupByClause>,
    pub order: Option<OrderClause>,
    pub limit: Option<LimitClause>,
}
```

`ComparisonOp(ConditionTree)`

```rust
#[derive(Clone, Debug, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub enum ConditionExpression {
    ComparisonOp(ConditionTree),
    LogicalOp(ConditionTree),
    NegationOp(Box<ConditionExpression>),
    ExistsOp(Box<SelectStatement>),
    Base(ConditionBase),
    Arithmetic(Box<ArithmeticExpression>),
    Bracketed(Box<ConditionExpression>),
}
```

```rust
#[derive(Clone, Debug, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub struct ConditionTree {
    pub operator: Operator,
    pub left: Box<ConditionExpression>,
    pub right: Box<ConditionExpression>,
}
```

把语法树转换成一个关系代数表达式其实很简单，关系代数的每一个代数运算符其实都和语法树的成员有挂钩关系，比如上述截图的其实就是对应代数关系的选择运算符，在每一个SqlQuery的SelectStatement语句，最终结果的record都要经过它本身的where_clause成员的筛选才能是真正的最终结果

同理，fields成员对应投影操作等等

```rust
#[derive(Clone, Debug, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub enum FieldDefinitionExpression {
    All,
    AllInTable(String),
    Col(Column),
    Value(FieldValueExpression),
}
```

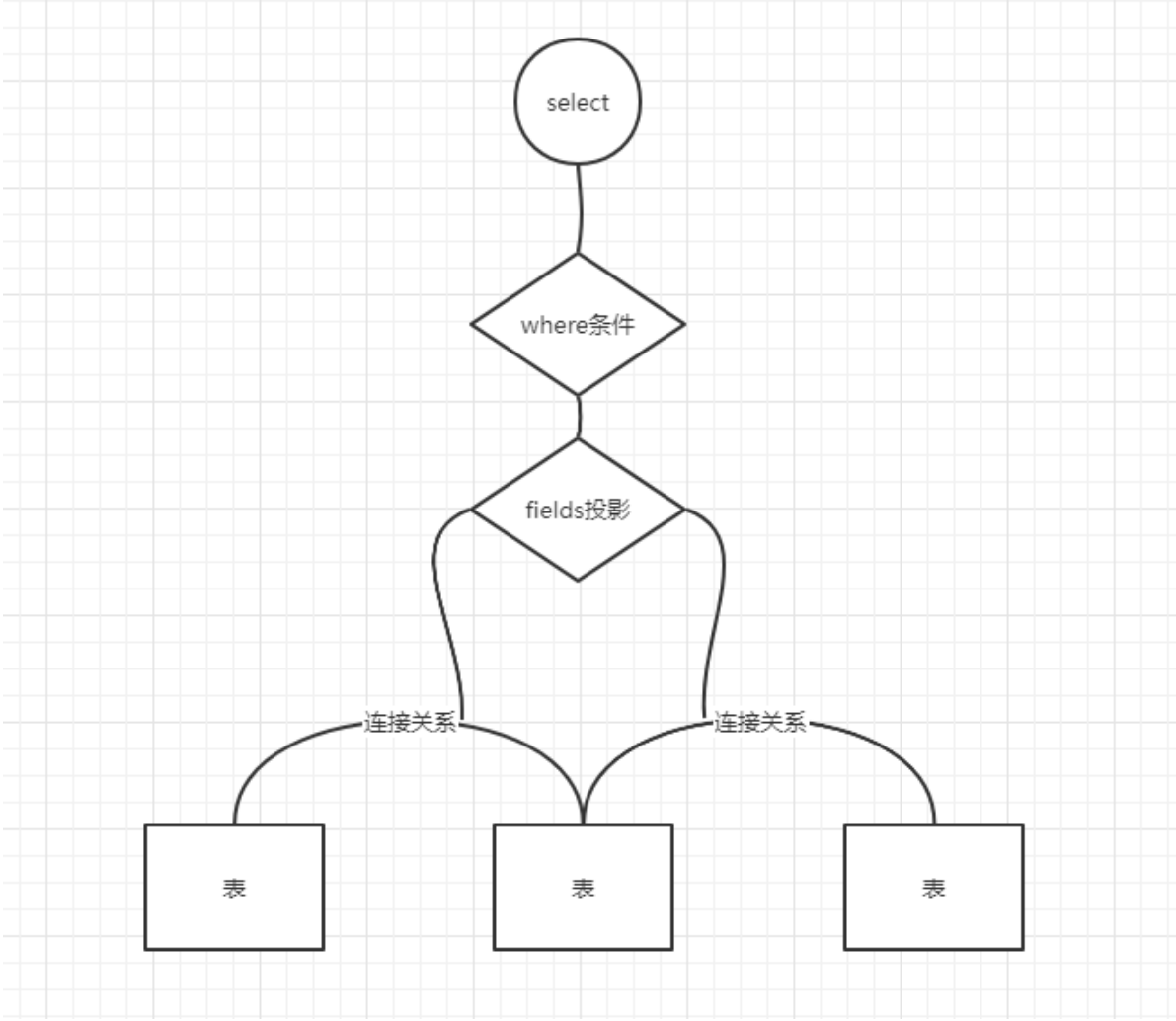
另外，可以看见pub tables: Vec

成员，每两个表之间的连接有一个pub join: Vec成员表示，可以知道我所分析的这个nom_sql库并不支持嵌套的select语句，但是基本的重要的sql性质都涵盖，这个语法书树和关系代数表达式其实一一对应的

```rust
#[derive(Clone, Debug, Default, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub struct Table {
    pub name: String,
    pub alias: Option<String>,
    pub schema: Option<String>,
}
```

```rust
#[derive(Clone, Debug, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub struct JoinClause {
    pub operator: JoinOperator,
    pub right: JoinRightSide,
    pub constraint: JoinConstraint,
}
```

根据上面的分析，解析所得的树其实是一颗下图所示的树，最底层是表，上面是一些关系代数条件（**可以用迭代责任流水线的模式实现关系代数的处理操作**），可以知道如果不对这棵树做处理，下面3个表做单纯的笛卡尔乘积，将会是数据库性能效率急剧下降



优化逻辑计划做的事情就是尽可能"下移"一些关系代数的处理操作，尽量在连接操作前做一些数据筛选处理，尽可能早的做选择运算和投影运算，尽可能早的做选择运算和投影运算，（具体到表上的操作），优化策略和下移的原则这里就不复述，很多得学习资料上都有，这里只是根据nom_sql具体分析

3)

制定物理查询计划：把优化后的逻辑计划转换成物理查询计划，要求指定操作执行的顺序，每一步使用的算法，操作之间的传递方式等。

物理查询优化就是根据现有的表的索引，决定是否可以用索引查询，增快查询速度，当然这只是最简单的一种表现形式

如果真的要在我原本的代码基础上实施物理计划，也只能依靠表的索引决定是否能通过索引加快查询，这是一个很简单的判断过程

高级的物理计划还应该根据内存大小，高级的元数据，选择是一趟扫描算法，还是二趟扫描算法，还是具体的基于排序得算法，基于排列的算法，基于散列的算法

具体的，本人非常推荐哈尔滨工业大学的战德成老师的网课，关于这方面讲的很详细，本人有非常认真听康辉老师的慕课，我认为各有特色和优点，但是我认为吉林大学的慕课可以结合哈尔滨工业大学的网课的优点，尤其是对要进行数据库底层实现的那批学生，可以提前一个学期就应该让学生从侧重了解数据库底层实现方面去教学学生，可以让学生做起来不是那么的痛苦

个人总结：

无论如何，个人阶段作业终于结束了，在此期间，学到了很多东西，也是第一次体验到“开源学习”的流程，看过很多的代码，只有少部分对自己有用，但是重要的是学会如何看别人的代码，自己写出的东西也让自己有些快乐，但总的来讲，痛苦大于快乐，因为大部分的工作和代码都是冗余的，重复的，但是还是感谢这段时间，也希望老师能对以后的学生要求不那么的严格，因为这个作业的压力太大了，最重要的是整个作业没有一种很明确的方向感和目的感，有时候看了一天的东西可能也没有用，我觉得老师应该加强对学生的具体管理（比如源码解读和推荐），而不是给一个大概的方向，让没有经验的学生做很多无用功，感到迷茫

本人不完善的代码的github地址：https://github.com/DctorWei1314/simple_db