DB课程设计：MiniSQL团队设计报告

一、 实验目的

1. 设计并实现一个精简型单用户 SQL 引擎(DBMS)MiniSQL，允许用户通过字符界面输入 SQL 语句实现表的建立/删除;
2. 索引的建立/删除以及表记录的插入/删除/查找。
3. 通过对 MiniSQL 的设计与实现，提高学生的系统编程能力，加深对数据库系统原理的理解。

二、 系统需求

1. 基本需求：表的创建、删除，记录的创建、删除，索引的创建删除，基本的错误显示，重点部分是记录的查询（要求支持等值查询、区间查询、以及使用and链接的多个条件的查询）
2. 数据类型：要求至少支持int、float、char
3. bonus：中文支持、GUI、更加丰富的错误显示、更多的语句、多线程执行、物理优化、使用 yacc 工具进行parser 生成、网络交互、更多的数据类型、底层并发。

在本次，我们完全覆盖了基本需求和几乎全部的Bonus。

三、 实验环境

1. 操作系统

三位同学都使用 Windows 系统进行开发

以一位的为例

CPU Name: Intel CoreTM i7-9750H CPU @ 2.60GHz

Property	Value
Base Frequency	2.59 GHz
Max Turbo Frequency	4.10 GHz
Cache	12 MB Intel® Smart Cache
Cores Number	12
Threads	12
TDP	45W

RAM: 16GB DDR4 2666MHz with Two channel memory

SSD: SAMSUNG MZVLB1T0HBLR-000L2

2. 开发环境

软件包	版本号
Goland	2020.01.3
go	1.16.3
python	3.9.1
node	14.16.0

3. 语言与语言规范

本次主要使用go和python作为主要语言进行开发。go用于实现DBMS本体，python用于实现产生大量测试数据。

4. 项目运行方法

四、系统设计

1. 分工情况

姓名	分工
王振阳	BufferManger、GUI（frontend和backend）
陈旭征	BufferManager、CatalogManager、Interpreter的设计
龙静毅	IndexManager

2. IndexManager的设计

我们的教材上其实给出了 B+ 树操作的写法，但我在实现的时候还是参考了算法导论上的方法。其主要区别是，数据库教材上 B+ 树的操作是自顶向下、再自底向上的两遍操作，而我实现的 B+ 树只需要自顶向下的一遍操作。这两种方法的优劣我会在后面进行比较说明

为了实现自顶向下一遍操作，我必须保证当在插入或者删除到前节点的时候，绝对不会产生节点的上溢或者下溢。这就需要在这个节点的上一层节点就进行检查，保证在当前节点不会出现问题。而根节点没有上一层节点，所以根节点的情况需要特别判断，综合起来，我的伪代码如下：

Insert:

```
function insert(key, value)
    if root is full
        split_root()
    cur_root := root
    while (cur_root is not leaf)
        find the first key[i] > key
        if (cur_root.child[i] is full)
            split(cur_root, i)
        cur_root = cur_root.child[i]
    find the first key[i] > key
    insert (key, value) into ith position
```

Delete: 由于我们只在 unique 上建索引，所以只需要提供 key 就好了。这里记 danger 表示 某个节点拥有的 key 值个数等于 $\lceil Order/2 \rceil$

```
function delete(key)
    if root is danger
        delete_root()
    cur_root := root
    while (cur_root is not leaf)
        find the first key[i] > key
        cur_root = cur_root.child[i]
        if (cur_root.child[i] is danger)
            handle(cur_root, i)
    find the first key[i] = key
    delete (key[i], value[i]) from cur_node
```

如果节点满了，我会进行分裂操作，分裂操作的核心操作在于：对于叶子节点，我们需要把分裂之后的第二个节点的第一个元素添加到父节点上作为 pivot；对于非叶子节点，我们需要把中间的元素提上去作为父节点中的 pivot

```
function split(cur_root, k)
    if cur_root.child[k] is leaf
        create a new node new_node
        copy the latter half of cur_root.child[k] into new_node
        insert (new_node.key[0], pointer to new_node) into cur_root
    else
        create a new node new_node
        copy the latter half of cur_root.child[k] into new_node
        insert (cur_node.child[i].key[median], pointer to new_node) into the
middle of cur_root
```

根节点的分裂稍有不同，我采取的做法是直接新建一个node，让它指向原来的 root，这样后面的操作就可以正常执行了

```
function split_root()
    create a new node new_node
    new_node.P[0] = root
```

这样，在后面的循环中我们会把根节点作为这个新建的节点的唯一一个子节点进行分裂，解决了根节点没有父节点就无法分裂的问题。

合并和分裂的操作类似，但区别在于合并未必每次都会进行，因为有可能左右两个节点与它的节点大小之和大于一个节点能够承受的，所以在这个时候我们首先会进行调整

```
function handle(cur_root, k)
    if cur_root.child[k + 1].size() + cur_root.child[k].size() < Order
        copy cur_root.child[k + 1] into the tail of cur_root.child[k]
        delete cur_root.child[k]
    else
        move the cur_root.key[k] into cur_root.child[k]
        move cur_root.child[k + 1].key[0] into cur_root.child[k].key[last]
```

这里我做了一些简化，实际上还要考虑$k = size$ 的情况，此时需要和前面的节点做合并或者调整

Search：

针对一共五种不同类型的查找，我的处理方法分别如下：

1. 小于、小于等于：直接找到第一个节点，利用叶子节点之间相互连接的特性，顺序查找，直到第一个不满足条件的被遇到
2. 等于：顺着 B+ 树找到第一个大于等于次元素的元素，如果是大于，返回空
3. 大于、大于等于：顺着B+树找到第一个大于等于此元素的元素，如果是大于，继续找到第一个大于此元素的元素

由于 Search 相对简单，就不提供伪代码了。

相比于书上提供的那种自顶向下再自底向上的处理方法，我的写法只需要一边扫描，单次操作的常数更小，但是考虑到我的写法会分裂或者合并不必要分裂和合并的节点，所以平均下来时间复杂度可能差不多。此外就是我的写法可能会产生更多半满的节点（因为分裂比书上的写法更频繁），相比书上的写法，空间消耗可能更大。

Index 物理层面的技术细节

由于上面涉及到的都是在内存中的操作，而实际上我们的 index 是要持久化、写入存储的，因此这里面涉及到了和 Buffer 的交互，我采用一个 block 记录一个节点的方法，内部的信息如下：

叶子节点：

block	**信息**
[0]	IsLeaf(uint8)
[1:2]	Size(uint16)
[3:4]	NextNode(uint16)
[5:8]	Pointer[0]
[9:?]	Key[0]
…	…

非叶子节点：

block	**信息**
[0]	IsLeaf(uint8)
[1:2]	Size(uint16)
[3:4]	Pointer[0]
[5:?]	Key[0]
….	

其中，叶子节点的 pointer 需要记录 block + offset， 而非叶子节点的 pointer 只需要记录 block，因为一个 block 就是一个节点。

利用这样的组织，我实现了getXXX系列的函数和 setXXX 系列的函数，分别用来获取和设置各个字段。

2. Catalog Manager

i. 功能描述

Catalog Manager 负责管理数据库的所有模式信息并提供访问及操作的接口，包括：

1. 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
2. 表中每个字段的定义信息，包括字段类型、是否唯一等。

> 3. 数据库中所有索引的定义，包括所属表、索引建立在哪个字段上等。
> 4. 根据api的需要，提前算出若干量并返回，加速RM的计算

ii. 主要数据结构

每个表的数据如下

```
type TableCatalog struct {
    TableName   string
    ColumnsMap  map[string]Column
    PrimaryKeys []Key
    Cluster     Cluster
    Indexs      []IndexCatalog
    RecordCnt   int //RecordCnt means the now record number
    RecordTotal int //RecordTotal means the total number
    RecordLength int//RecordLength means a record length contains 3 parts, a
vaild part , null bitmap, and record . use byte as the unit
}
```

注意这里为了加速查找，我们对每个列的记录使用的是map进行存储，因此column中需要记录下该column是第几列，整个column如下所示

```
type Column struct {
    Name    string
    Type    ColumnType
    Unique bool
    NotNull bool
    ColumnPos int   //the created position when table is created, this value is
fixed
    StartBytesPos int //the start postion in record bytes array
}
type ColumnType struct {
    TypeTag ScalarColumnTypeTag
    Length  int
    IsArray bool
}
```

可以看到我们定义了Type、Unique、NotNull属性等用于记录该列上的属性

而index的设计如下所示

```
type IndexCatalog struct {
    IndexName     string
    Unique        bool
    Keys          []Key
    StoringClause StoringClause
    Interleaves   []Interleave
}
```

为了之后的拓展考虑，我们的key是一个数组，即考虑到了之后做多属性索引的情况，因此我们设计为数组，同时我们引入了目前还没有实现的Interleaves、StoringClause属性，为了之后该属性的拓展考虑

此外我们还定义了很多用于拓展、转换的类型，这里不再一一赘述

iii. 存储数据结构

我们因为考虑了多database之间的使用，因此每个database都会保存一个catalog文件，而且每个database只会有一个catalog文件，在using不同的database的时候属于该database的内容才会被load。

因此Catalog不和Buffer打交道，属于自己维护自己的东西，因此我们没有特别的为catalog设计序列化方法，而是使用了`msgp`提供的go高性能序列化工具，大大简化了我们自己需要序列化的东西，同时也可以支持数组、map、指针等一系列类型的序列化。如下所示

```
// DecodeMsg implements msgp.Decodable
func (z *Cluster) DecodeMsg(dc *msgp.Reader) (err error) {

}

// EncodeMsg implements msgp.Encodable
func (z Cluster) EncodeMsg(en *msgp.Writer) (err error) {

}
```

我们使用`go:generate msgp`后，它就会自动帮我们将每个定义的struct生成我们需要的序列化和反序列化方法，同时保证了非常高的性能，在使用时我们仅需要调用改方法即可

而在整个CM大框架的存储设计上，我们分了两级进行处理，最外层存储整个Minisql的各个数据库信息，包括数据库名字和表的多少，类型如下所示

```
type MiniSqlCatalog struct {
    Databases    []DatabaseCatalog
}
```

该类型在启动时加载，同时在创建、删除数据库时刷新。

而当我们使用using database切换指定数据库时候，我们读取该database的文件同时反序列化，使用如下的数据结构

```
type TableCatalogMap map[string]*TableCatalog
var TableName2CatalogMap TableCatalogMap
```

其实就是一个表名string到表信息catalog的哈希表，msgp支持哈希表的序列化，这一点让我们非常意外，也极大的减少了我们的工作量，`TableName2CatalogMap`的刷新只在针对table、index等的创建和删除的时候会进行，因此整体上刷新的次数不会很多。

iiii.对外提供的接口如下：

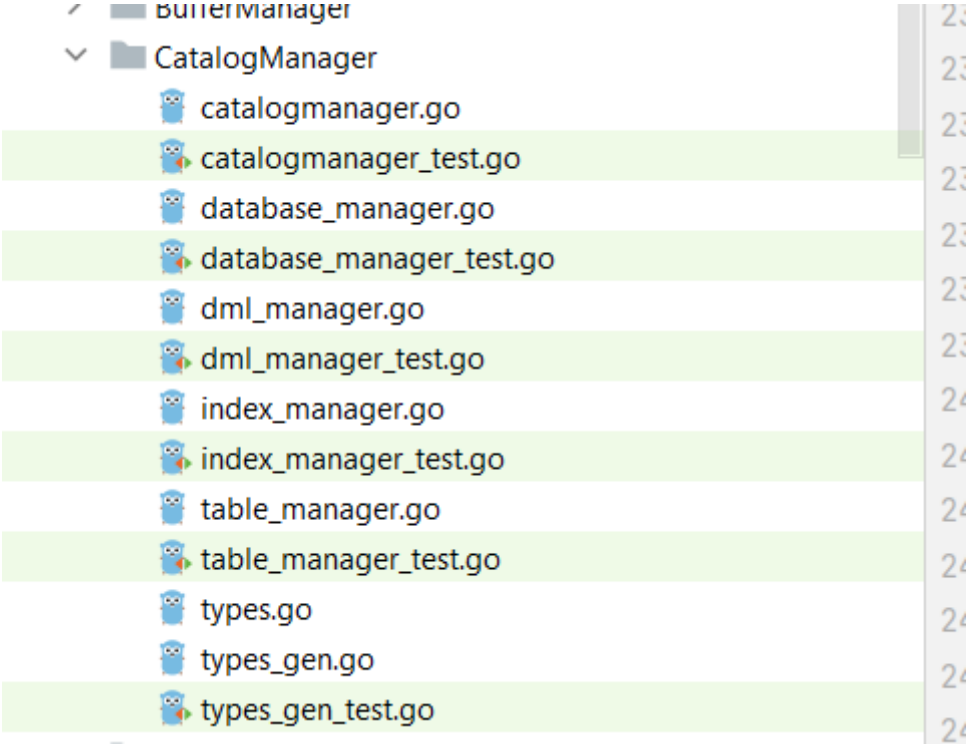
```
//LoadDbMeta is used for init the database catalog
//And if file is not exits, it will create it and return nil
func LoadDbMeta() error {

}

//InsertCheck Already do NULL CHECK, if a Value is null, I will check it and
throw a error !
func InsertCheck(statement types.InsertStament) (error,[]int,[]int,
[]UniquesColumn) {

}

//FlushDbMeta 用来flush整个db的数据库记录
func FlushDbMeta() error {

}

//GetDatabaseCatalog 用来获取该数据库的catalog
func GetDatabaseCatalog(databaseId string) (DatabaseCatalog,bool) {

}

//CreateDatabase 创建新的数据库
func CreateDatabase(databaseId string) error {

}

//GetDBTablesMap 获取某数据库下的所有table信息，返回值为 TableCatalogMap
func GetDBTablesMap(databaseId string)  (TableCatalogMap,error) {

}

//UseDatabase 使用某个数据库，加载其文件catalog
func UseDatabase(databaseId string) error  {

}

//DropDatabaseCheck 删除某database前的检查
func DropDatabaseCheck(databaseId string)error  {

}

//DropDatabase 直接删除某数据库的文件
func DropDatabase(databaseId string) error  {
```

```
}

//FlushDatabaseMeta will write the TableName2CatalogMap datas to storage
func FlushDatabaseMeta(databaseId string) error  {

}

//InsertCheck Already do NULL CHECK, if a Value is null, I will check it and
throw a error !
func InsertCheck(statement types.InsertStament) (error,[]int,[]int,
[]UniquesColumn) {

}

//DeleteCheck check the statement, and return the index if exist
func DeleteCheck(statement types.DeleteStatement)
(error,*types.ComparisonExprLSRV)  {

}

//UpdateCheck check the statement, and return the update columns name, values and
index if exist
func UpdateCheck(statement types.UpdateStament)  (error,[]string,[]value.Value,
*types.ComparisonExprLSRV)  {

}

//SelectCheck check the statement, and return index if exist
func SelectCheck(statement types.SelectStatement)
(error,*types.ComparisonExprLSRV)  {

}

func CreateIndexCheck(statement types.CreateIndexStatement)
(error,*IndexCatalog) {

}

func DropIndexCheck(statement types.DropIndexStatement)  error  {

}

func DropIndex(statement types.DropIndexStatement) error {

}

//CreateTableCheck 用来检查table，并返回所有的应该建的索引
func CreateTableCheck(statement types.CreateTableStatement) (error,
[]IndexCatalog)  {

}

//DropTableCheck don't delete the map[id] and the catalog file, just check the
legal
func DropTableCheck(statement types.DropTableStatement) error{

}
```

```
//DropTable 真正删除table文件与catalog
func DropTable(statement types.DropTableStatement) error  {

}
```

iiiii.模块测试

为了保证正确性我们进行了大量的测试，具体测试样例这里不再赘述，测试文件Catalog Manager文件夹下的*_test.go即是我们的测试文件



可以看到我们为每个文件都配备了测试数据，测试结果也完全正确

3. Buffer Manager

i. 功能描述

Buffer Manager 负责缓冲区的管理，主要功能有：

1. 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文件
2. 实现缓冲区的替换算法，当缓冲区满时选择合适的页进行替换
3. 记录缓冲区中各页的状态，如是否被修改过等
4. 提供缓冲区页的 pin 功能，及锁定缓冲区的页，不允许替换出去
5. 创建新的文件

Bonus：本次实验实现了**并发支持**的Buffer Manager，在Buffer Manger中也提供了对应的清空缓冲区及强制写回缓冲区内容的函数。

ii. 主要数据结构

buffer manager控制与文件交互的 block 的控制，可以看做一个 cache。

buffer内部编号为int类型，外界输入的filename和blockId拼接会唯一对应一个int索引，该索引并不是buffer内部的编号，因为buffer内部为链表+map，并非数组，所以该索引只是起到代替string的作用，加速查找

Block类

```
//Block 为缓冲区的块，只对外保留Data切片
type Block struct {
   filename string
   blockid  uint16
   dirty    bool
   pin      bool
   Data     []byte
   next     *Block  //后继
   prev     *Block  //前驱
       sync.Mutex
}
```

- FileName、block_id初始时会写入
- dirty代表该块是否被写过，如果写过请使用SetDirty方法
- pin代表该块是否被pin住，如果要pin某个块，请使用PinBlock方法，同样使用UnPinBlock方法解锁
- Data为对外暴露的数据，为大小4KB的切片，可以直接以你喜欢的方式修改，但是修改时请使用SetDirty置为脏
- mutex为互斥锁，当使用getBlock拿到该block的指针时，bm会自动给block上锁，因此使用完一个block后**务必**使用FinishRead释放锁，不然就会产生死锁再也无法拿到

暴露方法：

- SetDirty
- PinBlock
- UnPinBlock
- FinishRead

内部方法（无需外部调用）：

- read 读取filename和blockid指定的一段内容，如果该block为dirty，则不会read，而是会flush
- flush 如果不为dirty，直接返回，否则写入data
- mark 置filename和bid位
- reset 重置filename和bid

BM对外暴露函数

以下内容全部不进行严格的错误检查

```
//BlockRead 读byte，不检查block id和filename， 同时加互斥锁!!
func BlockRead(filename string, block_id uint16) (*Block, error)
```

```
//获取当前文件有多少块，拿到后0--t-1为可用区间
func GetBlockNumber(fileName string) (uint16,error)
```

```
//NewBlock 返回的 block id 是指新的块在文件中的 block id
//该函数是插入用的，已经支持并发操作
func NewBlock(filename string) (uint16, error)
```

```
//BlockFlushAll 刷新所有块，一般不使用，拿锁， 同时协程处理
func BlockFlushAll() (bool, error)
```

内部LRU算法实现

我们使用的是双向链表+map实现的LRU版本，通过map我们可以用O(1)的时间找到块，同时通过双向链表又可以用O(1)的时间将块移到最前面，如下是具体实现细节

```
type LRUList struct {
   root Block  // dummy header
   len  int
}
//LRUCache is the cache struct
type LRUCache struct {
    Size     int
    root     *LRUList
     sync.Mutex
    blockMap map[int]*Block
}

//PutBlock 将block插入buffer中，并返回block的指针
func (cache *LRUCache) PutBlock(value *Block, index int) *Block {
    cache.Lock() //写锁
    defer cache.Unlock()
    if item, ok := cache.blockMap[index]; ok {
        //fmt.Println(index)
        cache.root.moveToBack(item)
        return item
    }
    //maybe it's wrong, I'm not sure
    if len(cache.blockMap) >= cache.Size {
        var temp = cache.root.Front()

        for deleteNum:=0;deleteNum<DeleteSize;deleteNum++ {
            if temp.pin{
                temp=temp.next
            } else {
                temp.Lock()
                temp.flush()
                cache.root.remove(temp)
                oldIndex := Query2Int(nameAndPos{fileName: temp.filename,
blockId: temp.blockid})
                delete(cache.blockMap, oldIndex)
                temp.Unlock()
                temp=temp.next
            }
```

```
        }
    }
    cache.root.appendToBack(value)

    //fmt.Println(index)
    cache.blockMap[index] = value
    return  value
}
//GetBlock 获取buffer中的缓存，如果没找到就返回false
func (cache *LRUCache) GetBlock(pos int) (bool, *Block) {
    cache.Lock() //读锁
    defer cache.Unlock()
    if node, ok := cache.blockMap[pos]; ok {
        cache.root.moveToBack(node)
        return true, node
    }
    return false, nil
}
```

其他细节函数省略。

可以看到当我们发现Cache满了之后，我们不是仅将最后一个移出去，而是直接移除`DeleteSize`个，这样也较好的提高了整体的效率。

我们发现字符串的比较非常费时费力，因此在buffer内部我们使用int类型的统一标识符来标志该块，一个块有如下的struct

```
type nameAndPos struct {
    fileName string
    blockId  uint16
}
```

而我们维护一个map用于将该struct转成相应的int标识符，加快在buffer manager内部处理的时间

并发实现

经过分析我们发现有如下的并发需求

- 两个协程同时获取一个块
- 两个协程同时New新的块
- 两个协程同时想将自己的块移到双向链表尾部（即同时访问）

因此我们实现了三把互斥锁，用于上述的若干需求。

- 首先我们在block内部设置一把互斥锁，当一个block被一个协程占据时，该锁属于该协程，当block处理完毕后，我们使用FinishRead释放锁，这样两个协程同时读某块的时候，其他协程由于没有锁就会被阻塞等待。
- 其次我们在LRUCache内部也设置了一把锁，因为涉及到双向链表的修改，如果没有改锁，两个协程同时处理双向链表就会导致nil指针的出现，因此在一个协程等待修改链表的时候比较尝试获取LRU锁

- 最后我们在NewBlock的时候也可能会冲突，因为我们首先要拿到该文件有多少块，而两个协程可能拿到一样的块序号，就会导致冲突，因此在NewBlock时候我们也维护一个互斥锁，必须先拿锁才能真正的NewBLock

经过测试我们的并发做的非常好，虽然增加了8%（平均）的平均访问时间，但是相对于并发的优点，这点性能损失其实不算什么。

4. Record Manager

i. 模块总体设计

因为 API 仅起到检查和转发的作用，事实上 Record Manager 调用Catalog Manager 和 Index Manager，并与 Buffer Manager 交互。

通过在 `RMUtils.go` 中的封装，实现了对 Buffer Manager 操作的高度封装，而针对API的转发，进行函数处理。

ii. 功能描述

Record Manger 主要承担执行操作的功能，对于 table 的建立删除，index 的建立删除，record 的插入，更新和删除进行处理。

对于所有的运行时错误进行错误处理。

对于各类操作进行完整性校验。

iii. 主要数据结构

模块中最为重要的是freelist的维持，通过链表的形式实现对空余位置的快速查找。

为了简化代码结构，freelist 采用 Index Manager 中定义的链表格式。

```
var FreeList IndexManager.FreeList
```

iv. 主要函数功能

Record Manager 中的函数列表

```
const freeListFileHotFix = "_list"
var FreeList IndexManager.FreeList
func CreateIndex(table *CatalogManager.TableCatalog, newIndex
CatalogManager.IndexCatalog) error
func CreateTable(tableName string) error
func DeleteRecord(table *CatalogManager.TableCatalog, where *types.Where)
(error, int)
func DeleteRecordWithIndex(table *CatalogManager.TableCatalog, where
*types.Where, ...) (error, int)
func DropDatabase(databaseId string) error
func DropIndex(table *CatalogManager.TableCatalog, indexName string) error
func DropTable(tableName string) error
func FlushFreeList() error
func InsertRecord(table *CatalogManager.TableCatalog, columnPos []int,
startBytePos []int, ...) error
```

```
func SelectRecord(table *CatalogManager.TableCatalog, columns []string, where
*types.Where) (error, []value.Row)
func SelectRecordWithIndex(table *CatalogManager.TableCatalog, columns []string,
where *types.Where, ...) (error, []value.Row)
func UpdateRecord(table *CatalogManager.TableCatalog, columns []string, values
[]value.Value, ...) (error, int)
func UpdateRecordWithIndex(table *CatalogManager.TableCatalog, columns []string,
values []value.Value, ...) (error, int)
func checkRow(record value.Row, where *types.Where, colPos []int) (bool, error)
func columnFilter(table *CatalogManager.TableCatalog, record value.Row, columns
[]string) (value.Row, error)
func deleteRecord(table *CatalogManager.TableCatalog, recordPosition dataNode)
error
func getColPos(table *CatalogManager.TableCatalog, where *types.Where) (colPos
[]int)
func getRecord(table *CatalogManager.TableCatalog, recordPosition dataNode)
(bool, value.Row, error)
func getRecordData(tableName string, recordPosition dataNode, length int)
([]byte, error)
func loadFreeList(tableName string) error
func setRecord(table *CatalogManager.TableCatalog, recordPosition dataNode,
columnPos []int, ...) error
func setRecordData(tableName string, recordPosition dataNode, data []byte,
length int) error
func updateRecord(table *CatalogManager.TableCatalog, columns []string, values
[]value.Value, ...) (bool, error)
func updateRecordData(table *CatalogManager.TableCatalog, recordPosition
dataNode, record value.Row) error
type dataNode = IndexManager.Position
```

首先考虑对外暴露的接口

大部分结构的功能相对比较清晰，这里需要特别说明的是，在 Insert 的过程中，因为需要检查unique是否满足，实际上需要进行一次select，为了保证效率，在每一个unique的位置上建立B+树索引，以提升相对的插入效率

其次是内部使用的结构

```
func loadFreeList(tableName string) error {
    fileName := CatalogManager.TableFilePrefix() + "_data/" + tableName +
freeListFileHotFix //文件名
    if FreeList.Name == fileName {
   //已经load了
        return nil
    } else if len(FreeList.Name) > 0 {//需要把旧的flush
        err := FlushFreeList()
        if err != nil {
            return err
        }
    }
    if !Utils.Exists(fileName) {  //如果没有这个文件 新建该文件并序列化写入初始name信息
        newfile, err := Utils.CreateFile(fileName)
        defer newfile.Close()
        if err!=nil {
            return err
        }
        wt:=msgp.NewWriter(newfile)
```

```
        FreeList.Name=fileName
        err=FreeList.EncodeMsg(wt)
        if err!=nil  {
            return err
        }
        return wt.Flush()
    }
    //存在该文件 直接读取即可
    existFile, err := os.Open(fileName)
    defer existFile.Close()
    if err != nil {
        return errors.New("打开free list文件失败")
    }
    rd := msgp.NewReader(existFile)
    return FreeList.DecodeMsg(rd)
}

//FlushFreeList 退出程序时候请不要忘记
func FlushFreeList() error {
    oldList, err := os.OpenFile(FreeList.Name, os.O_WRONLY|os.O_TRUNC, 0666) //写
入旧文件
    defer oldList.Close()
    if err != nil {
        return errors.New("free list文件打开失败")
    }
    wt := msgp.NewWriter(oldList)
    err = FreeList.EncodeMsg(wt)
    if err != nil {
        return errors.New("free list文件写入失败")
    }
    return wt.Flush()

}
```

通过 msgp 库进行 freelist 的持久化存储，将freelist开始程序时读入而在结束程序之后flush

```
func getRecordData(tableName string, recordPosition dataNode, length int)
([]byte, error) {
    block, err :=
BufferManager.BlockRead(CatalogManager.TableFilePrefix()+"_data/"+tableName,
recordPosition.Block)
    if err != nil {
        return nil, err
    }
    defer block.FinishRead()
    record := block.Data[int(recordPosition.Offset)*length :
int(recordPosition.Offset+1)*length]
    return record, nil
}

func setRecordData(tableName string, recordPosition dataNode, data []byte,
length int) error {
    block, err :=
BufferManager.BlockRead(CatalogManager.TableFilePrefix()+"_data/"+tableName,
recordPosition.Block)
    if err != nil {
        return err
```

```
    }
    block.SetDirty()
    defer block.FinishRead()

    record := block.Data[int(recordPosition.Offset)*length :
int(recordPosition.Offset+1)*length]
    copy(record, data)
    return nil
}
```

这里封装了对于 Buffer Manager的操作，实现一条记录的读取和写入

```
func getRecord(table *CatalogManager.TableCatalog, recordPosition dataNode)
(bool, value.Row, error) {
    data, err := getRecordData(table.TableName, recordPosition,
table.RecordLength)
    if err != nil {
        return false, value.Row{}, err
    }
    nullmapBytes:=data[0:len(table.ColumnsMap)/8+1]
    nullmap:=Utils.BytesToBools(nullmapBytes)

    if nullmap[0] == false {
        return false, value.Row{}, nil
    }
    record := value.Row{Values: make([]value.Value, len(table.ColumnsMap))}
    //思考顺序问题，Column是以什么顺序存储的
    for _, column := range table.ColumnsMap {
        startPos := column.StartBytesPos
        length := column.Type.Length //这个length是给char和string和null用的，所以其他
类型无用
        valueType := column.Type.TypeTag

        if nullmap[column.ColumnPos+1] == false {
            valueType = CatalogManager.Null
        }
        if record.Values[column.ColumnPos], err =
            value.Byte2Value(data[startPos:], valueType, length); err != nil {
            return true, value.Row{}, err
        }
    }
    return true, record, nil
}


func setRecord(table *CatalogManager.TableCatalog, recordPosition dataNode,
    columnPos []int, startBytePos []int, values []value.Value) error {
    data := make([]byte, table.RecordLength)
    nullmapBytes:=data[0:len(table.ColumnsMap)/8+1]
    nullmap:=Utils.BytesToBools(nullmapBytes)
    nullmap[0] = true
    for _, columnIndex := range columnPos {
        nullmap[columnIndex+1] = true
    }
    nullmapBytes=Utils.BoolsToBytes(nullmap)
```

```
    copy(data[:], nullmapBytes)
    for index, _ := range columnPos {
        tmp, err := values[index].Convert2Bytes()
        if err != nil {
            return err
        }
        copy(data[startBytePos[index]:], tmp)
    }
    if err := setRecordData(table.TableName, recordPosition, data,
table.RecordLength); err != nil {
        return err
    }
    return nil
}
```

更进一步的，这里实现了序列化的过程，将一条以结构存储的记录存入Buffer

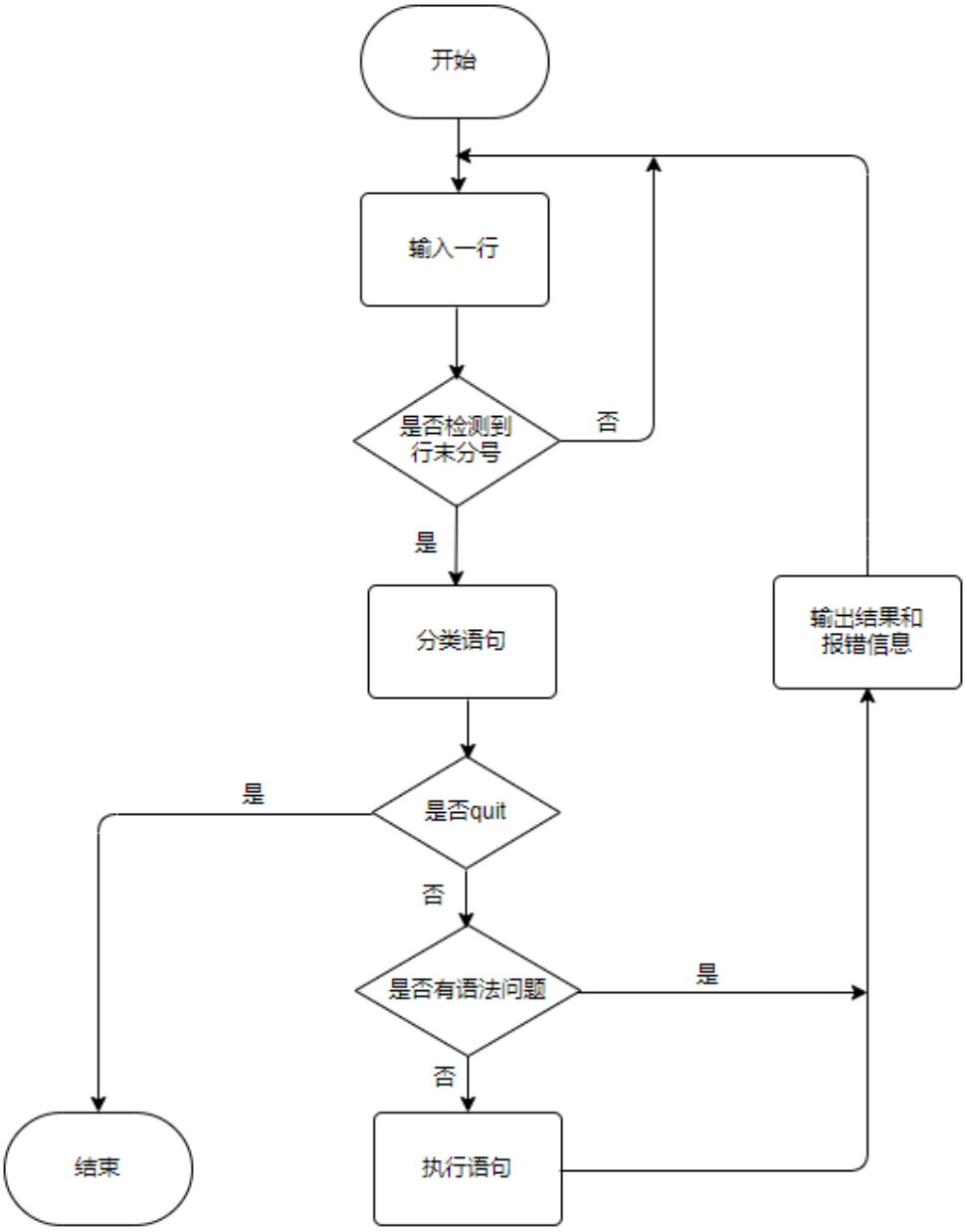
```
func columnFilter(table *CatalogManager.TableCatalog, record value.Row, columns
[]string) (value.Row, error) {
    if len(columns) == 0 { //如果select* 则使用全部的即可
        return record, nil
    }
    var ret value.Row

    for _, column := range columns {
        ret.Values = append(ret.Values,
record.Values[table.ColumnsMap[column].ColumnPos])
    }

    return ret, nil
}
```

物理优化的一部分，通过截取所需的列实现效率的优化

5. Catalog Manager

6. Interpreter

i. 程序设计流程图



###

ii. 功能描述

Interpreter 模块直接与用户交互，主要实现了以下功能：

1. 程序流程控制，接收一个reader器，不停的对该reader调用read方法并使用yacc进行语法分析，并最后将结果传给API

2. 接收并解释用户输入的命令，对输入的命令进行初步的语法检查，检查要点包括关键词是否错误、命令格式是否正确。对初步语法检查无误的命令，根据其命令类型生成命令的内部数据结构表示，并调用API的相应函数对命令进行进一步的处理。

```
3. 接收底层所有抛出的错误提示，并显示。
```

iii. 主要数据结构

因为go是强类型语言，因此我们针对Minisql设置了一套比较强大的、拓展性比较强的类型

我们将所有类型抽象成go里的interface，即接口，只要我们对某个结构体实现了如下函数，他就可以被视为是我们的Value类型，同时为各个类型实现了该接口，接口定义如下

```
//Value is the most important type which record the  true value
type Value interface {
    String()string
    //Compare is unsafe compare, if you don't know the type is same, don't use
it!
    Compare(Value, CompareType)(bool,error)
    // CompareWithoutType will return 0 if equal, -1 if less , 1 if greater
    CompareWithoutType(Value) (int,error)
    SafeCompare(Value,CompareType)(bool,error)
    //Convert2Bytes is convert value to bytes
    Convert2Bytes() ([]byte,error)
    Convert2IntType() ValueType
}
```

如下是我们的自定义类型

```
type Int struct {
    Val int64
}
type Float struct {
    Val float64
}
type Bytes struct {
    Val []byte
}
type Bool struct {
    Val bool
}
type Null struct{
    length int
}
type Alien struct {
    Val interface{}
}
```

我们一共定义并且实现了6种类型，可以看到6种类型各自的基类型都是go原生类型，同时在内部我们进行了大量的优化，包括提供不安全比较函数、比较函数加速等方法尽可能弥补go的接口类型显式转换带来的较大的性能开销。

minisql内所有运行的值都会被解释成Value类型，并使用Value类型进行接收和传递，因此该类型的加速非常重要，对此我们又提供了`NewFromParquetValue`,`Byte2Value`等大量的方法去加速Value和Bytes的转换，具体可以参考`src\Interpreter\types\types.go`，这里不在一一赘述。

经过测试，我们发现各种优化后的Value类型之间的比较速度比原生的速度慢了4-5倍，但是考虑到比较次数并不算多，因此基本符合我们的要求。

iv. goyacc的使用

类型方面，所有的语句都是实现了如下接口的类型

```
type DStatements interface {
    GetOperationType() OperationType
}
```

通过该接口，我们可以方便的分析出此次语句是什么类型，之后便可以使用显式转换进行转换成我们需要的类型。

go本身就包含一个强大的语法解析器goyacc，但是需要手动安装，同时需要自定义一个lex，因此使用起来也比较麻烦，对我的挑战性也比较大，最终花费了大量的时间、参考若干文档后，最终才能写出这个功能非常完整，类型强大的Minisql解析器。

- Lex模块

  整个Lex部分最重要的就是Lex函数和Error函数，只要我们实现了这两个接口，即可认为是一个Lex解析器，在Lex函数中，我们逐字符处理Reader中的内容，同时根据内容的不同打上不同的类型标签，最后返回本次解析的结果，Error函数将处理在解析过程中的各种错误，并产生Error抛出

- yacc模块

  goyacc 内部有两个重要的 interface, 其中 `yyLexer` 需要使用者自己实现提供，yacc 会生成 `yyParser` 的实现，其使用 yyLexer 做解释操作。解释的过程和和解释前后都可以嵌入自己的代码逻辑，完成一个程序或者单纯生成一个自定义的语法树结构。

  文法定义简单说明如下

描述符	说明
%union	用来定义一个类型并映射 golang 的一个数据类型
%struct	同%union 建议使用%union
%token	定义终结符，表示不能再拆了的字符 是一个 union 中定义的类型, 可无类型
%type	定义非终结符
%start	定义从哪个终结符开始解析 默认规则段中的第一个终结符
%left	定义规则结合性质 左优先
%right	定义规则结合性质 右优先
%nonasso	定义规则结合性质 不结合
%perc term	定义优先级与 term 一致

  这里我们仅以select语句的解析进行解释

```
select_stmt:
    SELECT sel_field_list FROM table_name_list where_opt limit_opt
    {
      s:=types.SelectStatement{
        Fields: $2,
        TableNames: $4,
        Where: $5,
        Limit: $6,
       }
     yylex.(*lexerWrapper).channelSend <- s
    }
```

这是最外层的解析

```
sel_field_list:
   '*'
   {
     $$=types.FieldsName{
        SelectAll:true,
     }
   }
   | column_name_list
   {
     $$=types.FieldsName{
        SelectAll:false,
        ColumnNames:$1,
     }
   }
```

这是select column的解析

```
table_name_list: // TODO  mulitplart where condition, now only one table can
be select
    IDENT_ALL  //here we use table_name which is a string type ,not INDENT
    {
      $$ = make([]string, 0, 1)
      $$ = append($$, $1)
    }
    | table_name_list ',' IDENT_ALL
    {
      $$ =append($1, $3)
    }
```

这是选择表的解析

```
where_opt:
    {
      $$=nil
    }
    | WHERE expr_opt
    {
      $$=  &types.Where{Expr:$2}
```

```
    }
expr_opt:
    '(' expr_opt ')'
    {
        $$=$2
    }
    | IDENT_ALL compare_type Value
    {
    $$= &types.ComparisonExprLSRV{Left: $1,Operator:$2, Right:$3 }
    }
    |  Value compare_type IDENT_ALL
    {
    $$= &types.ComparisonExprLVRS{Left: $1,Operator:$2, Right:$3 }
    }
    |  Value compare_type Value
    {
    $$= &types.ComparisonExprLVRV{Left: $1,Operator:$2, Right:$3 }
    }
    |  IDENT_ALL compare_type IDENT_ALL
    {
    $$= &types.ComparisonExprLSRS{Left: $1,Operator:$2, Right:$3 }
    }
    | expr_opt AND expr_opt
    {
        left:=$1
        right:=$3

$$=&types.AndExpr{Left:left,Right:right,LeftNum:left.GetTargetColsNum(),Righ
tNum:right.GetTargetColsNum(),}
    }
    | expr_opt OR expr_opt
    {
        left:=$1
        right:=$3

$$=&types.OrExpr{Left:left,Right:right,LeftNum:left.GetTargetColsNum(),Right
Num:right.GetTargetColsNum(),}
    }
    | NOT expr_opt
    {
        left:=$2
        $$=&types.NotExpr{Expr:left,LeftNum:left.GetTargetColsNum(),}
    }
```

这是where的解析

可以看到整个语法和语义还是比较清晰的，也方便我们去实现一个真正的解析器，至少效率上比正则表达式快得多，我们测试后发现，解析100w条数据用时4.5s左右，足以看出其高效

7. API

i. 功能描述

API为执行SQL语句提供接口，供Interpreter层调用。它接受Interpreter层的输出信息，负责需要调用Catalog Manager、Record Manager 和实现的语句检查部分，并进行报错。检查通过的语句，API会根据各个manager中函数调用的需求进一步处理成需要的数据结构，并调用相应的函数执行语句，接收语句执行结果并返回。

ii. 功能实现

这里我们使用了**go的协程机制**，简单地说**interpret和api执行是一套流水线**，interpret一边解析、api一边执行，这样大大缩短了处理流程的时间，所有通信都通过信道channel完成，同时我们也提供了等待机制，即等待一条语句的执行完成后，下次一输入才会被允许，而不允许一边输入一边执行。

在main函数里我们定义了两个信道

```
    StatementChannel:=make(chan types.DStatements,500)  //用于传输操作指令通道
    FinishChannel:=make(chan Error.Error,500) //用于api执行完成反馈通道
```

而在yacc中，每次我们解析完成后都会尝试发送到操作指令信道

```
     yylex.(*lexerWrapper).channelSend <- s
```

在api中，我们接收该信道传来的操作指令，进行处理后返回执行信息。

```
//HandleOneParse 用来处理parse处理完的DStatement类型  dataChannel是接收Statement的通道,整个mysql运行过程中不会关闭，但是quit后就会关闭
//stopChannel 用来发送同步信号，每次处理完一个后就发送一个信号用来同步两协程，主协程需要接收到stopChannel的发送后才能继续下一条指令，当dataChannel
//关闭后，stopChannel才会关闭
func HandleOneParse( dataChannel <-chan types.DStatements,stopChannel chan<- Error.Error)   {
    var err Error.Error
    for statement:=range dataChannel {
        //调用各个api进行处理

        stopChannel<- err
    }
    close(stopChannel)
}
```

但是这样就会存在一个问题，在interpret过程中遇到的语法错误无法使用该信道传输回去，只能就地进行处理，这也是我们之后准备优化的一个方向。

除了Execfile之外的api我们不准备详细介绍，下面我们简单说明一下execfile的执行过程

- 语法检查，文件检查
- 分别开启parse协程和handle协程

- 进行流水线处理
- 解析完成后关闭parse、handle协程
- 返回

可以看到execfile其实可以理解为新开了一个main函数，在这个上面去执行多个语句，流水线作业。

iii. 并发支持

实际上在一个文件内的若干语句我们目前没有做到并发执行，因为控制语句执行顺序是一件非常非常麻烦的事情，go没有提供很好的方法去规定协程的执行顺序。

但是我们提供了多用户处理下的一定程度上的并发支持，得益于Buffer Manager的并发支持设计，我们的API一定程度上支持并发，具体的细节将在Buffer Manager的报告中进行阐述。

iiii. 对外提供接口

```
//HandleOneParse 用来处理parse处理完的DStatement类型  dataChannel是接收Statement的通道,整个mysql运行过程中不会关闭，但是quit后就会关闭
//stopChannel 用来发送同步信号，每次处理完一个后就发送一个信号用来同步两协程，主协程需要接收到stopChannel的发送后才能继续下一条指令，当dataChannel
//关闭后，stopChannel才会关闭
func HandleOneParse( dataChannel <-chan types.DStatements,stopChannel chan<- Error.Error)  {

}
//CreateDatabaseAPI 只调用CM，和IM、RM无关
func CreateDatabaseAPI(statement types.CreateDatabaseStatement)  Error.Error {

}
//UseDatabaseAPI 只调用CM，和IM、RM无关
func UseDatabaseAPI(statement types.UseDatabaseStatement) Error.Error  {

}
//DropDatabaseAPI  先CM的check，和IM、RM无关 ，再调用RM的drop ，  再在CM中删除并flush
func DropDatabaseAPI(statement types.DropDatabaseStatement) Error.Error  {

}

//CreateTableAPI CM进行检查，index检查 语法检查  之后调用RM中的CreateTable创建表， 之后使用RM中的CreateIndex建索引
func CreateTableAPI(statement types.CreateTableStatement) Error.Error {

}

//CreateIndexAPI CM进行检查，index语法检查 之后使用RM中的CreateIndex建索引
func CreateIndexAPI(statement types.CreateIndexStatement) Error.Error  {

}

//DropTableAPI CM进行检查，注意这个时候并不真正删除CM中的记录， 之后RM的DropTable删除table文件以及index文件， 之后让CM删除map中的记录同时flush
func DropTableAPI(statement types.DropTableStatement) Error.Error  {

}
```

```
//DropIndexAPI CM进行检查， RM中删除index 之后CM中再删除并flush
func DropIndexAPI(statement types.DropIndexStatement) Error.Error  {

}

//InsertAPI nothing to explain
func InsertAPI(statement types.InsertStament) Error.Error  {

}

//UpdateAPI nothing to explain
func UpdateAPI(statement types.UpdateStament) Error.Error  {

}

//DeleteAPI nothing to explain
func DeleteAPI(statement types.DeleteStatement) Error.Error {

}

//SelectAPI nothing to explain
func SelectAPI(statement types.SelectStatement) Error.Error  {


}
// ExecFileAPI 执行某文件  开辟两个新协程
func ExecFileAPI(statement types.ExecFileStatement) Error.Error  {
    //parse协程 有缓冲信道

}
```

9. Bonus

i. 功能描述

通过 React-Ace 框架实现数据库的GUI界面。
这一GUI界面支持用户登录鉴权，权限管理；
SQL语句自动补全；
SQL语句语法静态检查；
查询数据回显。

ii. 主要结构

实现 SQL 语句查询的模块

```
import React, {useEffect, useRef, useState} from 'react';
import {VariableSizeGrid as Grid} from 'react-window';
import ResizeObserver from 'rc-resize-observer';
import classNames from 'classnames';
import {Table, Empty, Result} from 'antd';

function VirtualTable(props) {
```

```
    const {columns, scroll, className} = props;
    const [tableWidth, setTableWidth] = useState(0);
    const widthColumnCount = columns.filter(({width}) => !width).length;
    const mergedColumns = columns.map(column => {
        if (column.width) {
            return column;
        }

        return {...column, width: Math.floor(tableWidth / widthColumnCount)};
    });
    const gridRef = useRef();
    const [connectObject] = useState(() => {
        const obj = {};
        Object.defineProperty(obj, 'scrollLeft', {
            get: () => null,
            set: scrollLeft => {
                if (gridRef.current) {
                    gridRef.current.scrollTo({
                        scrollLeft,
                    });
                }
            },
        });
        return obj;
    });

    const resetVirtualGrid = () => {
        gridRef.current.resetAfterIndices({
            columnIndex: 0,
            shouldForceUpdate: false,
        });
    };

    useEffect(() => resetVirtualGrid, []);
    useEffect(() => resetVirtualGrid, [tableWidth]);

    const renderVirtualList = (rawData, {scrollbarSize, ref, onScroll}) => {
        ref.current = connectObject;
        return (
            <Grid
                ref={gridRef}
                className="virtual-grid"
                columnCount={mergedColumns.length}
                columnWidth={index => {
                    const {width} = mergedColumns[index];
                    return index === mergedColumns.length - 1 ? width -
scrollbarSize - 1 : width;
                }}
                height={scroll.y}
                rowCount={rawData.length}
                rowHeight={() => 54}
                width={tableWidth}
                onScroll={({scrollLeft}) => {
                    onScroll({
                        scrollLeft,
                    });
                }}
            >
```

```
                {({columnIndex, rowIndex, style}) => (
                    <div
                        className={classNames('virtual-table-cell', {
                            'virtual-table-cell-last': columnIndex ===
mergedColumns.length - 1,
                        })}
                        style={style}
                    >
                        {rawData[rowIndex]
[mergedColumns[columnIndex].dataIndex]}
                    </div>
                )}
            </Grid>
        );
    };

    return (
        <ResizeObserver
            onResize={({width}) => {
                setTableWidth(width);
            }}
        >
            <Table
                {...props}
                className={classNames(className, 'virtual-table')}
                columns={mergedColumns}
                pagination={false}
                components={{
                    body: renderVirtualList,
                }}
            />
        </ResizeObserver>
    );
}


function DataTable(props) {
    const {tableData, tableColumns} = props;
    const getData = (tableData) => {
        return tableData.map(
            (x, idx) => {
                const item = {}
                x.map(
                    (_x, _idx) => {

                        item[_idx] = _x;
                        return _x;
                    }
                )
                return item;
            }
        )
    }


    const _columns = tableColumns.map(
        (x, idx) => {
            return ({
```

```
                title: x,
                dataIndex: idx,
                key: x
            })
        }
    )
    const _data = getData(tableData)
    return (
        <VirtualTable
            columns={_columns}
            dataSource={_data}
            scroll={{
                y: 300,
                x: '100vw',
            }}
        />
    )
}

function Callback(props) {
    const {status, times, rows, data} = props
    if (status === true) {
        if (data !== undefined && data !== null && data !== []) {
            const tableColumns = data[0]
            const tableData = data.slice(1)
            return <div style={{
                width: '100%',
                height: '100%'
            }}>
                <Result
                    status="success"
                    title={`操作成功！ 本次操作影响了${rows}行数据，耗时${times}s`}
                    style={{
                        backgroundColor: '#FFF'
                    }}
                />
                <DataTable tableData={tableData} tableColumns={tableColumns}/>

            </div>
        }
        return (<div style={{
            width: '100%',
            height: '100%'
        }}>
            <Result
                status="success"
                title={`操作成功!耗时${times}s`}
                style={{
                    backgroundColor: '#FFF'
                }}
            />
        </div>)
    } else if (status === false) {
        return <Result
            status="error"
            title={`操作失败！ 后端反馈：${data}`}
            style={{
                backgroundColor: '#FFF'
```

```
            }}
        />
    } else if (status === undefined) {
        return <Empty/>
    } else if (times === undefined || times === null || times === []) {
        // TODO 退出登录
        return <Result
            status="success"
            title={'登出成功! '}
            style={{
                backgroundColor: '#FFF'
            }}
        />

    } else {
        return <Result
            status="error"
            title={`啊欧，失败了！可能是你的语句不太正常。`}
            style={{
                backgroundColor: '#FFF'
            }}
        />
    }


}

export default Callback;
```

实现数据回显的模块

```
import React, {useState,useRef} from "react";
import AceEditor from "react-ace";
import {Button, Empty, Layout, Menu, message, PageHeader} from 'antd';
import Axios from "axios";
import Callback from "./Callback";

import 'ace-builds/src-noconflict/ext-language_tools';
import 'ace-builds/src-noconflict/ext-searchbox';
import 'ace-builds/src-noconflict/mode-mysql';
// theme
import 'ace-builds/src-noconflict/theme-sqlserver';
import 'ace-builds/src-noconflict/theme-github';
import 'ace-builds/src-noconflict/theme-eclipse';
import 'ace-builds/src-noconflict/theme-monokai';
import 'ace-builds/src-noconflict/theme-clouds';
import 'ace-builds/src-noconflict/theme-chrome';
import 'ace-builds/src-noconflict/theme-solarized_dark';
import 'ace-builds/src-noconflict/theme-solarized_light';
import Redirect from "react-router-dom/es/Redirect";

function MiniSQL(props) {
    const {userName} = props


    const {SubMenu} = Menu;
```

```
    const {Content, Footer, Sider} = Layout;

    const themeList = ["sqlserver", "github", "eclipse", "monokai", "clouds",
"chrome", "solarized_dark", "solarized_light"]
    const [theme, setTheme] = useState(themeList[0])
    const EditorRef = useRef()
    const [queryData, setQueryData] = useState([])
    const [checkOn,setCheckOn] = useState()
    const sqlSplit = (texts) => {
        const dtFilter = require("dt-sql-parser").filter;

        const afterFilterComments = dtFilter.filterComments(texts)
        const afterSplit = dtFilter.splitSql(afterFilterComments)
        console.log(afterFilterComments)
        console.log(afterSplit)
        let res = []
        for (let i = 0; i < afterSplit.length; i++) {
            const item = afterSplit[i]
            if (item !== "" && item !== '\n' && item !== undefined) {
                res.push(item.replace(/[\r\n]/g, '').replace('undefined', ''))
            }
        }
        console.log(res)
        return res
    }
    const syntaxCheck = (text) => {
        // if(checkOn===undefined||checkOn===false){
        //     return false
        // }

        const dtSqlParser=require("dt-sql-parser").parser;
        return dtSqlParser.parseSyntax(text);
    }


    const doQuery = (data) => {

        if (userName === 'manager') {
            if (data.indexOf('delete') !== -1 || data.indexOf('drop') !== -1) {
                message.error('权限不足! 只有root账号才能进行delete和drop操作')
                return
            }
        } else if (userName === 'customer') {
            console.log(data)
            if (data.indexOf('select') === -1) {
                message.error('权限不足! 普通用户只能使用select操作')
                return
            }
        }
        const query = async (data) => {
            try {
                const res = await Axios(
                    'api/query',
                    {
                        method: 'POST',
                        data: {
                            'query': data
                        }
```

```
                    }
                );
                setQueryData(res.data.data)
            } catch (e) {
                message.error('后端提示：啊欧，失败了！可能是你的语句不太正常。')
            }
        }

        let texts = sqlSplit(data)
        if (texts === undefined || texts === null) {
            message.error('空代码可跑不了哦！')
            return
        }

        for (let i = 0; i < texts.length; i++) {
            const check = syntaxCheck(texts[i]);
            console.log(check)
            if(check!==false){
                message.error(
                    `前端语法检查:\n
                                错误类型: ${check.token}\n
                                错误位置: \n
                                        开始行数: ${check.loc.first_line}    结束行
数: ${check.loc.last_line}\n
                                        开始列数: ${check.loc.first_column}  结束列
数: ${check.loc.first_column}\n
                                修改建议: \n
                                        改为:
${check.expected!==null&&check.expected.length>0?check.expected[0].text:'暂无'}
                                `

                )
                return
            }
            query(texts[i])
        }


    }


    return (<div>
        {
            (userName === undefined || userName === null || userName === '') ?
<Redirect to="/"/>
                : <Layout>

                    <Content style={{padding: '0 50px'}}>
                        <PageHeader
                            className="site-page-header"
                            title="MinSQL Editor"
                            subTitle={"current user: " + userName}
                        />
                        <Layout className="site-layout-background" style=
{{padding: '24px 0'}}>
                            <Sider className="site-layout-background" width=
{200}>
                                <Menu
                                    mode="inline"
```

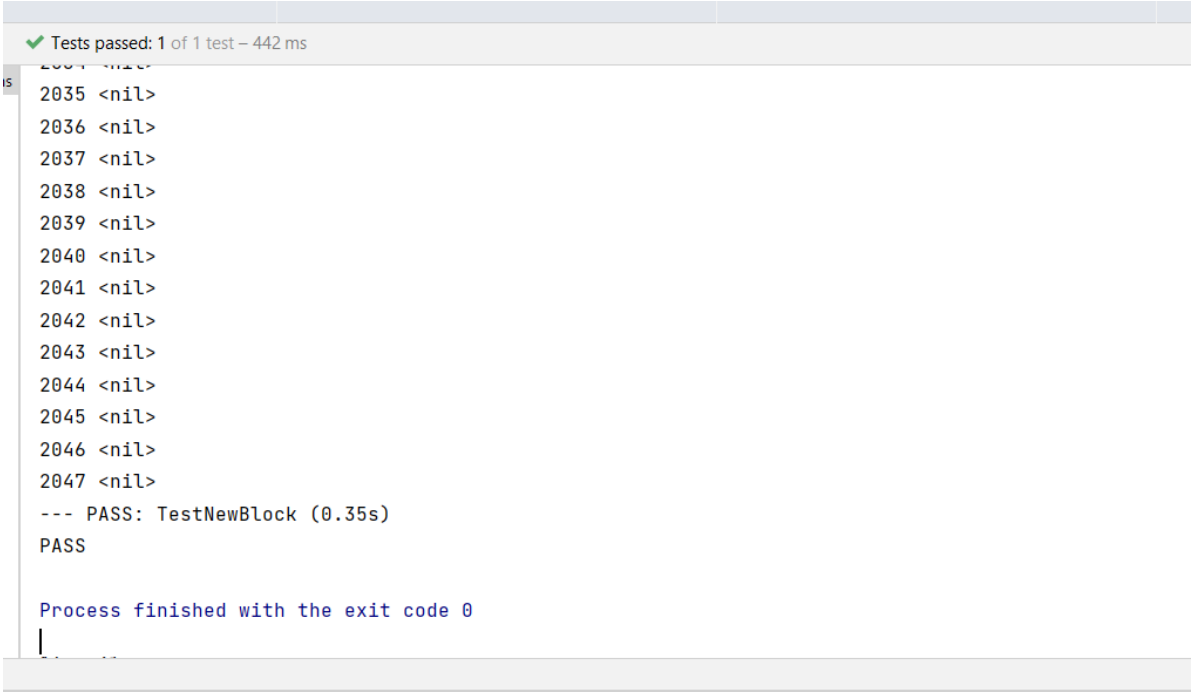
```
                                    style={{height: '100%'}}
                                    onClick={(param) => {
                                        setTheme(themeList[param["key"]])
                                    }}
                                >
                                    <Button
                                        type="primary"
                                        style={{
                                            textAlign: "center",
                                            width: "100%",
                                            marginBottom: "5px"
                                        }}
                                        onClick={(e) => {
                                            e.preventDefault();
                                            const context =
EditorRef.current.editor.getValue()
                                            doQuery(context)

 EditorRef.current.editor.setValue(context)
                                        }}
                                        ghost
                                    >

                                        Run Code
                                    </Button>
                                    <SubMenu key="theme" title="Theme">
                                        <Menu.Item key={0}>sql
server</Menu.Item>
                                        <Menu.Item key={1}>github</Menu.Item>
                                        <Menu.Item key={2}>eclipse</Menu.Item>
                                        <Menu.Item key={3}>monokai</Menu.Item>
                                        <Menu.Item key={4}>clouds</Menu.Item>
                                        <Menu.Item key={5}>chrome</Menu.Item>
                                        <Menu.Item key=
{6}>solarized_dark</Menu.Item>
                                        <Menu.Item key=
{7}>solarized_light</Menu.Item>
                                    </SubMenu>
                                </Menu>

                            </Sider>
                            <Content style={{padding: '0 24px', minHeight:
300}}>
                                <AceEditor
                                    ref={EditorRef}
                                    mode="mysql"
                                    theme={theme}
                                    fontSize={16}
                                    style={{
                                        width: '100%',
                                        height: '100%',
                                        minHeight: 300,
                                        fontFamily: "Fira Code, Consolas,
monospace"
                                    }}
                                    setOptions={{
                                        enableBasicAutocompletion: false, //关闭基
本自动完成功能
```

```
                                        enableLiveAutocompletion: true,    //启用实
时自动完成功能
                                        enableSnippets: true,
                                        showLineNumbers: true,
                                        editorProps: {$blockScrolling: true},
                                        highlightActiveLine: true,
                                        tabSize: 4
                                    }}
                                />

                            </Content>
                        </Layout>
                        {queryData === null || queryData === undefined
                            ? <Empty/>
                            :
                            <Callback status={queryData[0]} times={queryData[1]}
rows={queryData[2]} data={queryData[3]}/>}

                    </Content>
                    <Footer style={{textAlign: 'center'}}>MiniSQL ©2021 Created
by Wolfram</Footer>
                </Layout>
        }
    </div>)

}

export default MiniSQL;
```

五、系统实现分析及运行截图

测试newblock的情况

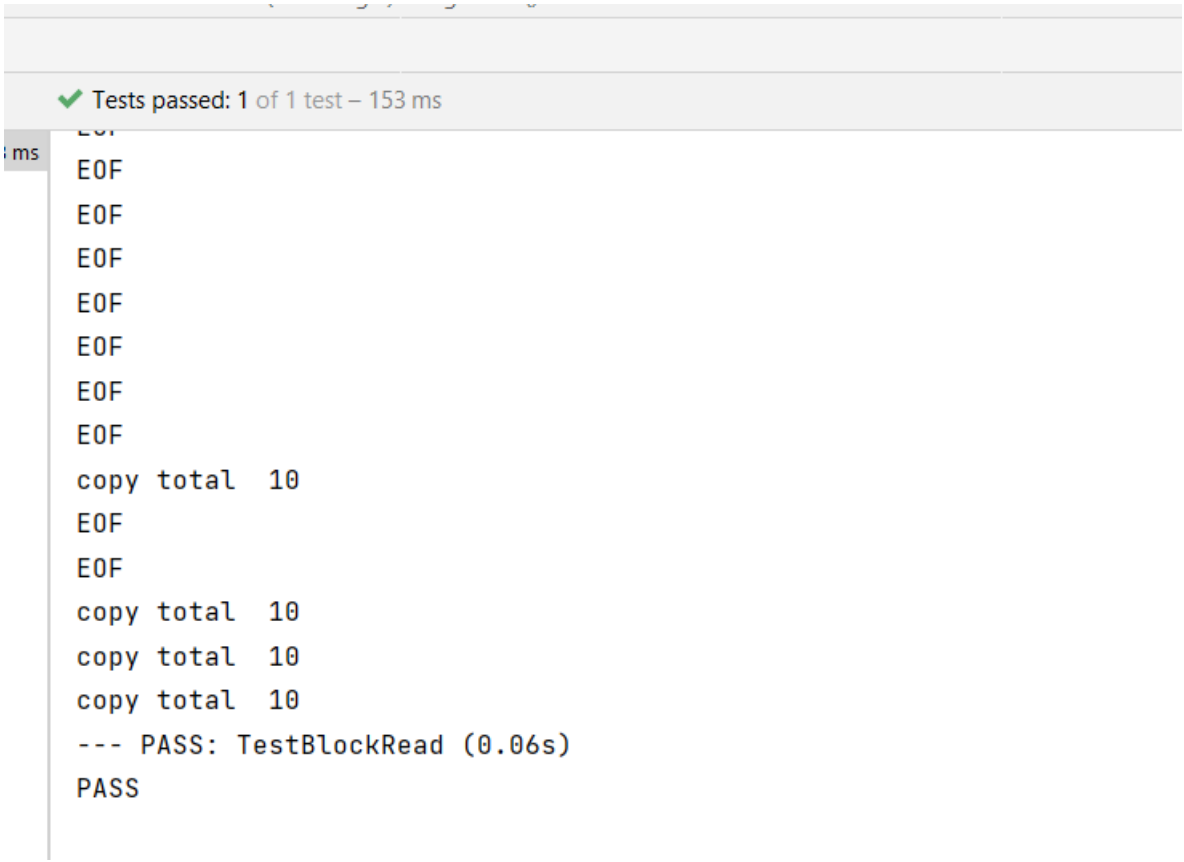
```
//
func TestNewBlock(t *testing.T) {
    InitBuffer()
    filename:="database1"
    wg:=sync.WaitGroup{}
    wg.Add(InitSize)
    for i:=1;i<=InitSize;i++ {
        go func() {
            defer wg.Done()
            fmt.Println(NewBlock(filename))
        }()
    }
    wg.Wait()
}
```



1000**次随机读取**（并发）

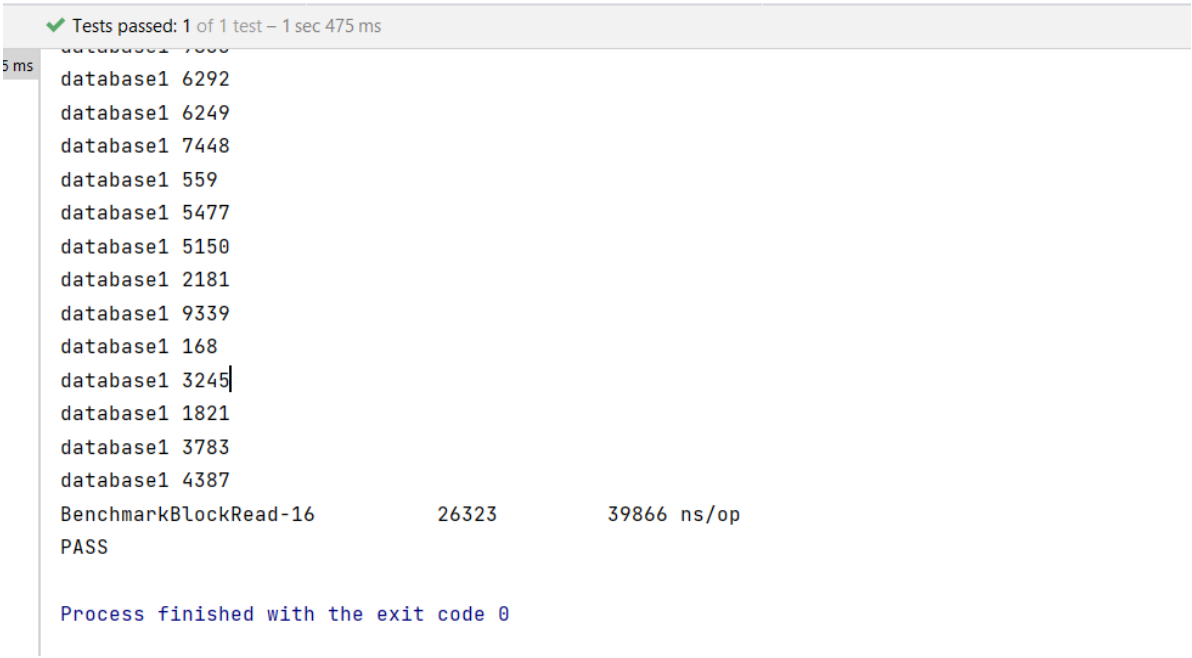
```
func TestBlockRead(t *testing.T) {
    InitBuffer()
    //t.Parallel()

    filename:="database1"
    fmt.Println(GetBlockNumber(filename))
    r := rand.New(rand.NewSource(time.Now().Unix()))
    newdata:=[]byte("I love syf")
    //for i:=1;i<2000;i++ {
    //  //tmp:=uint16(i%200)
    //  tmp:= uint16(r.Intn(2*InitSize))
    //  block,err:=BlockRead(filename,tmp)
    //  if err!=nil {
    //      fmt.Println(err)
    //      return
    //  }
    //  num:=copy(block.Data,newdata)
    //  fmt.Println("copy total ",num)
    //  block.SetDirty()
    //  block.FinishRead()
    //}
    //BlockFlushAll()
    wg:=sync.WaitGroup{}
    wg.Add(1000)
    for i:=1;i<=1000;i++ {
        go func() {
            defer wg.Done()
                tmp:= uint16(r.Intn(2*InitSize))
                block,err:=BlockRead(filename,tmp)
                if err!=nil {
                    fmt.Println(err)
                    return
                }
                num:=copy(block.Data,newdata)
```

```
                fmt.Println("copy total ",num)
                block.SetDirty()
                block.FinishRead()
        }()
    }
    wg.Wait()
}
```



读取性能测试（并发）

```
func BenchmarkBlockRead(b *testing.B) {
   InitBuffer()
   filename:="database1"
   //t:=Query2Int(nameAndPos{fileName: filename,blockId: 1})
   r := rand.New(rand.NewSource(time.Now().Unix()))

   b.RunParallel(func(pb *testing.PB) {
      for pb.Next() {
         tmp:= uint16(r.Intn(10000))
         block,_:= BlockRead(filename,tmp)
         fmt.Println(filename,tmp)
         block.FinishRead()
      }
   })

}
```



六、 遇到的问题及解决方法

1. 团队协作

   可以说这个数据库管理程序是我至今为止做过的最具有挑战性的大程，因为各个部分的接口比较复杂，接口传入的参数要求也不太一样，因此总是会出现我对于一些接口的传入参数和返回参数的数据结构理解有误的情况。在debug期间也确实出现了很多因为队友间没有充分交流而导致的问题。这也给了我对于团队合作的一些启发，遇到不太确定的问题一定要及时沟通，需要与队友间频繁交流，了解互相的进度。

2. goyacc的使用

   由于还没有学过编译原理这门课，我在学习lex和yacc的时候还是废了好大一番功夫才真正看明白了goyacc的使用方法，之后又是废了好大的功夫才写出这个比较齐全的lex和yacc，参考了很多资料，查阅了前人的实现过程，最终才有了这样一个功能齐全的interpret，不可否认我在这个过程总有许多收获。

3. 协程的使用

   虽然我们要求是一个简单的串行程序，但是既然我们底层bm支持并发，那么光是串行程序将大大降低运行速度，因此我学习了大量的通道使用方法，最终想到了这样一个二级流水线的工作方法， 最终事实证明我们的方法提升的效率还是非常明显的，也没有白费我一番功夫苦思冥想吧。

七、总结

这一次大程应该说是上大学以来做的最难的一次课程设计，不光是码力的锻炼，更是各种意义上的提升，但只要认真完成，也确实能从中学习到许多。同时要感谢孙老师课堂上的悉心指导为我们打下了良好的基础，感谢助教哥哥的耐心指导和细致的实验说明也帮助了我们许多，感谢我的两位队友与我的协助，和我一起解决了非常多的问题，很幸运能和这么可靠的两位同学合作。