MeSQL 总体报告

实验目标

1. 设计并实现一个简易的数据库系统
2. 此系统应融入本学期所学的各种数据库技术，能高效地处理大量数据，支持基本的 SQL 指令
3. 此系统应模块化、低耦合、易拓展

命名

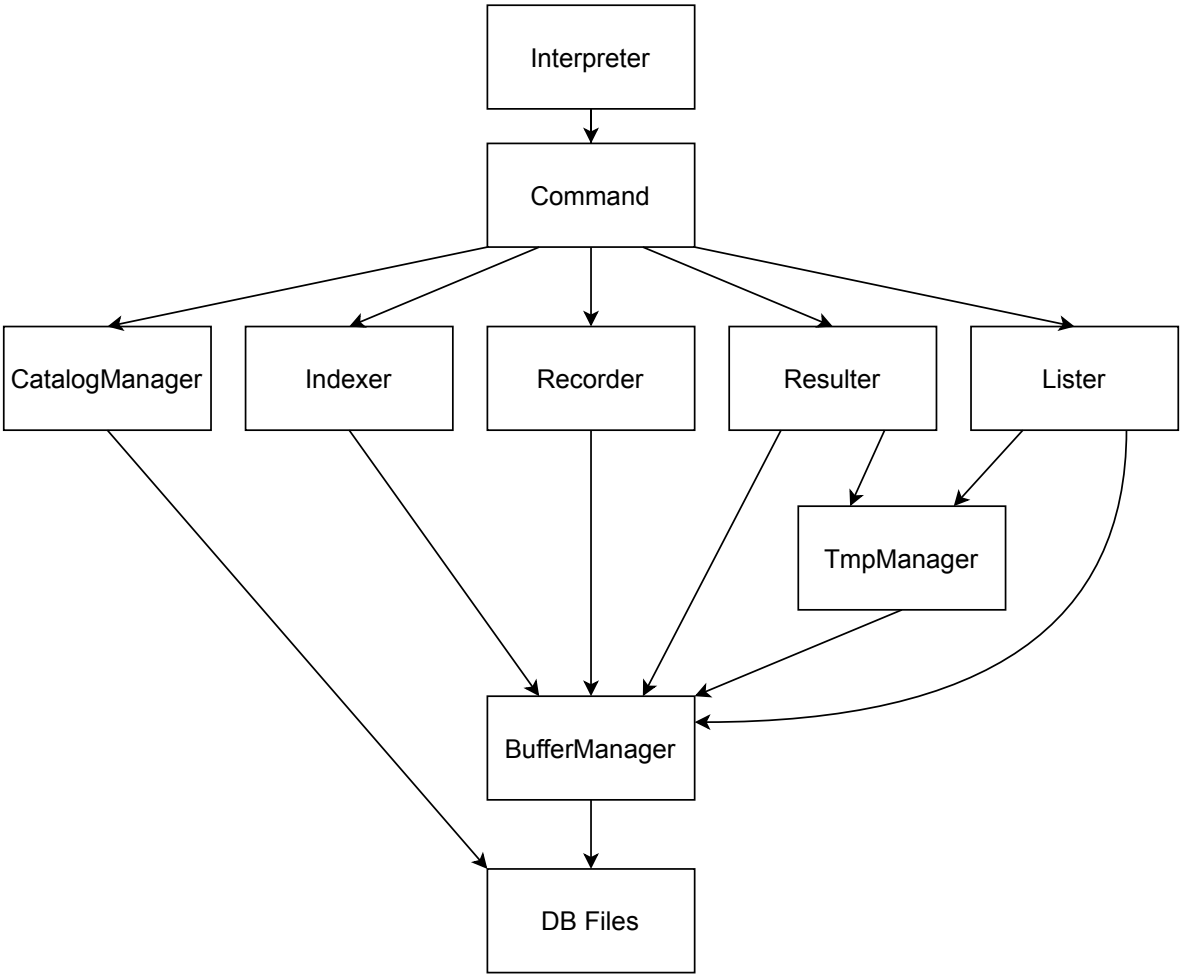
以下将此次设计并实现的数据库系统称为 **MeSQL** ，这有三个原因：

1. 这是我自己设计并实现的，所以 Me
2. 与指导名称 MiniSQL 谐音
3. 与知名的 MySQL 相映成趣

实验环境

1. Ubuntu 20.04 (Ubuntu 18.04 兼容)
2. GCC 9.3.0
3. Flex 2.6.4
4. GNU Bison 3.5.1

整体架构



我的能力不足以在开始实现前就设计好整个系统，因此整个系统设计是随着一个个我能掌握的模块的实现浮出水面的。这可以称为“实现驱动设计”。

整体采用模块化设计，以 `Command` 模块为中心，以其需要进行的操作主导设计了许多直接功能类；对于一些会复用的功能也实现为类，提高抽象性。我设计的体系结构与指导中的有所区别，这将在模块设计部分说明。

由于我是单人完成，按要求个人报告中会详细讨论 IndexManager (我这里的名称是 `Indexer`) 的实现，所以其他部分的一定程度的设计细节会在本报告中展示。

编程语言

根据“高效处理大量数据”这个预定目标，性能瓶颈处，例如 `BufferManager` 、 `Indexer` 等的实现，选择编译型语言是毫无疑问的。由于本次工程很大，用面向过程的 C 语言实现必然带来很多麻烦，所以这里选择了 C++。然而对于其他部分选择什么语言，有一些考虑。

主要问题集中在 `Interpreter` 模块，它基本上不是性能瓶颈（除了读入大量操作的时候），而由于 Python 的字符串处理优势，这部分选择 Python 实现是合理的。然而这样选择实际上就成了混合语言编程，涉及语言连接的问题。经过一番考察和尝试，我发现这非常不方便，存在许多问题。所以最终我抛弃了混合语言编程的想法，整个工程都用 C++ 来实现。

模块设计

Base 组件

Base 组件含有 `base` `color` `error` `timer` 四个小组件和 `manager` 大组件，这里介绍前四个，后面将展开介绍 `manager` 组件。

base

这个小组件非常重要，定义了三个基本类型：

```
enum class DataType {
    INT,FLOAT,CHAR,UNKNOWN
};
```

这是此数据库系统支持的三种数据类型以及作为初始化的 UNKNOWN 类型。

```
enum class CompareOp {
    EQ,NE,L,G,LE,GE,UNKNOWN
};
```

这是 where 子句中可用的六种比较运算符以及作为初始化的 UNKNOWN 类型。

```
class Literal {
public:
    DataType dtype;
    int int_val;
    float float_val;
    string char_val;

    Literal();
    Literal(int _val);
    Literal(float _val);
    Literal(const string &_val);

    ... 其他成员函数声明 ...
};
```

字面值类型。所有输入的整数、浮点数、字符串以及数据库系统中储存的数据，在进入内存后都会被包装为 `Literal` 类。`Literal` 类定义了运算符和其他成员函数，使得数据处理中所有“值”能被统一地处理。例如，要表示一个插入的元组，只需 `vector<Literal>` 。

color

实现了函数 `colorful(string,COLOR)` ，用于给输出加颜色，体现在错误信息上。

error

定义了类 `MeError` ，这是我在整个实现中抛出异常的唯一类型。

timer

利用 C++ 11 以上提供的 `<chrono>` 库，采用其最高精度的时钟 (`high_resolution_clock`) 实现了计时器，用于语句执行时间的测量。

Interpreter 模块

这个模块是我在整个实验中花时间最久的，一共做了三周（其他加起来做了两周多）。设计的时候有这样一些考虑：

首先我以前就知道有一套叫做 Lex & Yacc (其 GNU 替代为 Flex & Bison) 的工具，可以生成语法解析器，作为编译器/解释器实现的一个模块。实际上 `Interpreter` 模块就是要实现一个解释器，如果能用这套工具再好不过。其实准备的过程中还发现了一些第三方的，更加现代的语法分析工具，但它们大多不是基于 C++ ，而是基于 Python 、Java 等的，因此最后我放弃了其他语法分析工具。

Flex & Bison 的确合适，但学习使用它们需要一定时间，由于我单人完成，所以不能把所有时间都花在这上面。然后我考虑了自己实现语法分析，有两种可能：

1. 自己实现一套基于 BNF 的语法解析器，只需支持本模块所需的非常有限的功能，也应该不会太难
2. 从“用户基本上只会输入正确命令”的假设出发实现“语法分析”，再加上少量的错误处理应对一些简单的情况

我思考了一下，觉得第二种方式也不会太容易，而且有违"易拓展"的实验目标，所以放弃了；对于第一种方式，我需要花时间学习自动机的相关理论，我感觉所需时间不比学 Flex & Bison 短，所以也放弃了。最终选择了学习并使用 Flex & Bison 来做解释器。这花了我三周时间。

这套工具实际上十分复杂，我学完基础知识后找了一个 C++ 的 Demo (实际上 Flex & Bison 的有意义的 C++ Demo 非常少，幸好找到了一个；这个 Demo 实现了形如 `f(2,3,5)` 的函数调用的解析) 来研究，最后模仿他的结构写出了自己的 `Interpreter` 。

模块功能

解析命令行输入，将其**结构化**并执行。

以创建表语句为例。当最终识别到一条**创建表语句 (create_table_statement)** 时，会传入已知信息生成一个 `CreateTableStatement` 类 (属于 `Command` 模块) 的实例；之后这条创建表语句会被规约为一条**语句 (statement)** 执行。这样做是为了简化代码，不需要为每种语句都写一遍一样的调用代码。这里利用了 C++ 类的虚函数。

核心代码片段

这里展示一下创建表语句的语法定义——解释器就该这么写！

```
create_table_statement: //创建表语句
    "create" "table" IDENTIFIER "(" create_table_col_defs ","
create_table_pk_def ")"
        {
            const string &table_name = $3; // IDENTIFIER
            const vector<TableColumnDef> &cols = $5; // create_table_col_defs
            const string &primary_key = $7; // create_table_pk_def
            // 根据已知信息，生成"创建表语句"类
            $$ = new CreateTableStatement(table_name,cols,primary_key);
        }
    ;

create_table_col_defs: //多列定义
    create_table_col_def
        {
            $$ = vector<TableColumnDef>{$1};
        }
    | create_table_col_defs "," create_table_col_def
        {
            $$.swap($1);
            $$.push_back($`3);
        }
    ;

create_table_col_def: //单列定义
    IDENTIFIER create_table_col_spec
```

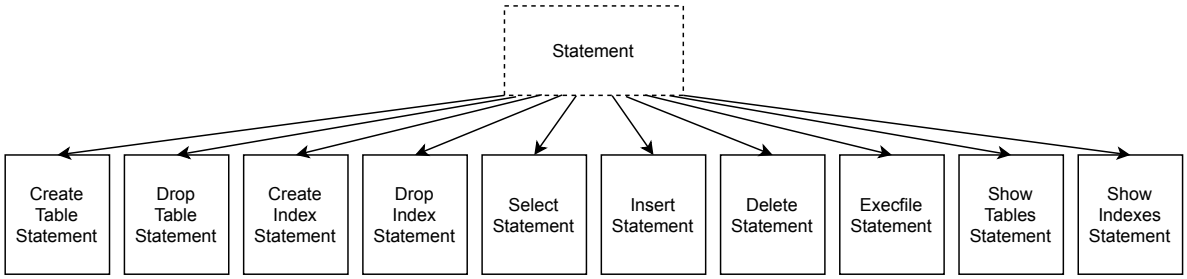
```
        {
            $$ = TableColumnDef(0,$1,$2);
        }
    ;

create_table_col_spec: //单列性质定义
    create_table_col_type
        {
            $$ = $1;
            $$.is_unique = false;
        }
    | create_table_col_type "unique"
        {
            $$ = $1;
            $$.is_unique = true;
        }
    ;

create_table_col_type: //单列类型定义
    "int" { $$ = TableColumnSpec(DataType::INT,4,false,false); }
    | "float" { $$ = TableColumnSpec(DataType::FLOAT,4,false,false); }
    | "char" "(" NUMBER ")"
        {
            此处略去一大段，包含错误处理和规约;
        }
    ;

create_table_pk_def: //primary key 定义
    "primary" "key" "(" IDENTIFIER ")"
        {
            $$ = $4;
        }
    ;
```

Command 模块

模块功能

1. 向 `Interpreter` 模块提供用于包装信息、执行语句的类，实际上成为给解释器的 API。
2. 此模块也是各种语句实际执行的位置，它调用下层模块实现语句的实际功能。

对象设计



这里 `Statement` 是一个抽象类，其最重要的两个成员函数是

1. 纯虚函数 `virtual void _execute() = 0;`
2. 实函数 `void execute();` ：

```
void Statement::execute() {
    Timer t; t.start();
    try {
        _execute();
        t.stop(); printer << t.paren_str() << endl;
    } catch (异常) { 异常处理 }
}
```

即将 `_execute()` 包装上计时器和异常处理，如此一来每个派生类只需实现各自的 `_execute()` 函数执行对应语句，避免了代码冗余。

所有运行过程中出现的错误，包括程序错误 (程序本身的bug导致的) 和语句错误 (如要创建的表名已经存在)，都通过 C++ 的异常机制来处理。遇到错误时 `throw` 出一个带有错误信息的对象，最终会在上述 `execute()` 函数的异常处理部分被捕获，然后错误信息会被输出。一个简单的抛出异常的例子：

```
if (man->cat.tables.count(table_name)) throw MeError::MeError(
    "Invalid Operation",
    "table '" + table_name + "' already exists"
);
```

实现细节

每种语句实现的重点在于实现 `_execute()` 函数，其框架是类似的：

1. 判断语句合法性，若不合法则抛出异常。目的：确保语句可以执行，提前发现问题，避免**执行过程中**发现语句不合法，带来原子性问题。
2. 执行语句。
3. 输出。

得益于下层组件的良好抽象，大部分语句的实现是直接的。这里只提两个有意思的语句。

select 操作

select 操作作为子过程用在 Select 语句、 Delete 语句和 Insert 语句的判重中。

我对 select 操作进行了小小的查询优化：

1. 如果条件中有**索引列上的等值条件**，则使用此索引、此条件查询
2. 否则，如果条件中有**索引列上的大小关系条件(<,<=,>=,>)** ，则使用此索引、此条件查询
3. 否则遍历整个表查询

这个优化是简单、直接、有效的。

Execfile 语句

最开始的想法是执行 execfile 语句时直接切换输入流，将 `cin` 换成文件的 `ifstream` 。但这样会遇到问题——命令行交互时每行输入前都会有 prompt ('<<<' 或 '...') ，这在执行文件的时候便也会出现，显然与 execfile 的意义不符。

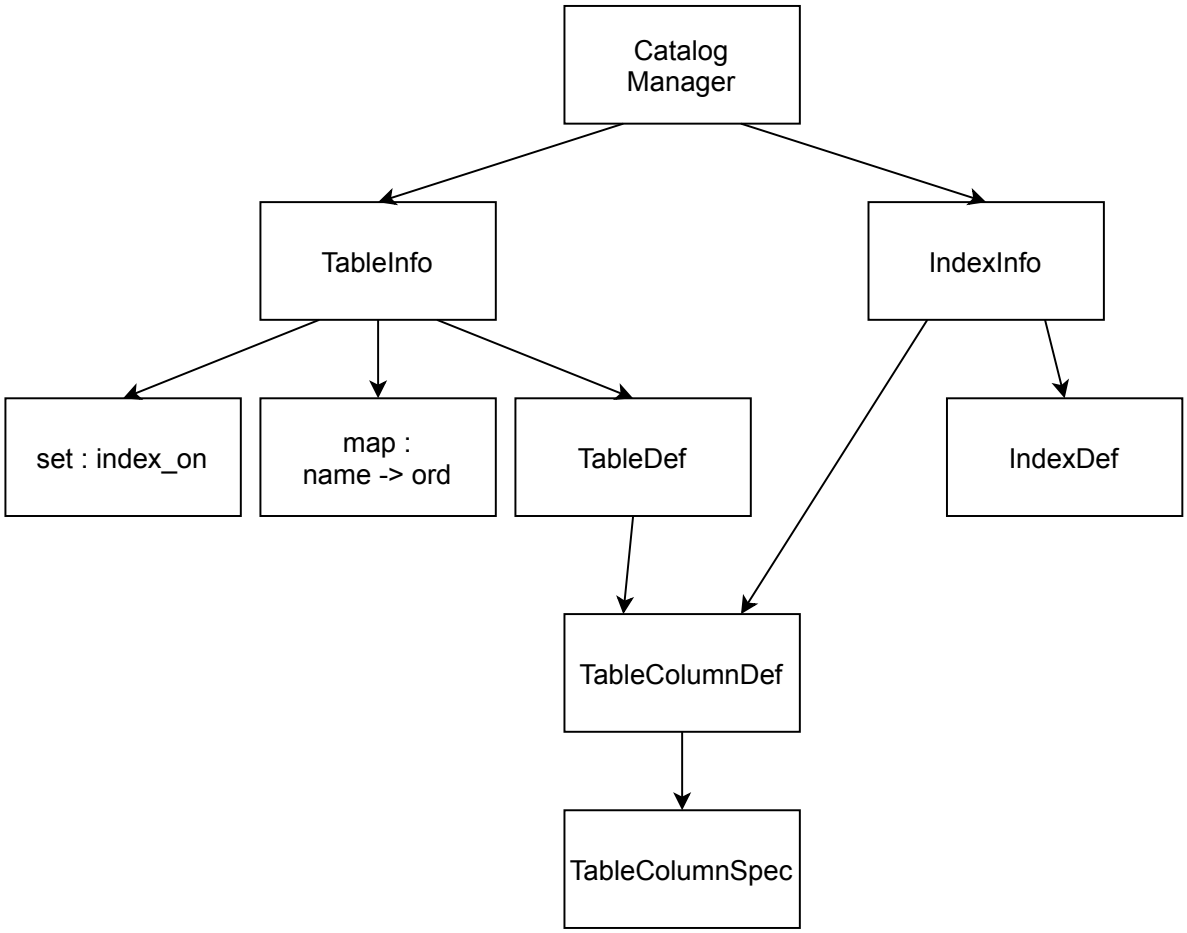
最终的办法是，调整了 `Interpreter` 的实现，把 prompt 分离出来；Execfile 的时候只需直接生成一个 `Interpreter` 类的新实例，在这个新的解释器中指定输入流为文件流，然后执行 `parse` 即可。这也自然地支持了嵌套 execfile 。

CatalogManager 模块

模块功能

1. 集成、包装元数据，包括表信息、索引信息
2. 从 catalog 文件中载入元数据，将元数据写入 catalog 文件
3. 支持元数据的访问，提供 创建/删除 表/索引 元数据的接口

对象设计



- `TableColumnSpec` ：表的一列的类型说明，包括数据类型、是否 unique 、是否是主键
- `TableColumnDef` ：表的一列的定义，包括列名、序号、类型说明
- `TableDef` ：表定义，包括表名、各列的定义
- `TableInfo` ：表信息，包括表定义、该表上的所有索引名称 (用 STL 的 set 存储加速访问)，以及一个列名到列序号的映射 (用 STL map ，即红黑树存储加速访问)。后二者是为了快速获取相关信息而设计的，载入文件时预处理出这两个数据结构，运行过程中动态维护，将元数据写回文件时不会写入，因为它们可以从 `TableDef` 和 `IndexInfo` 计算出来。
- `IndexDef` ：索引定义，包括索引名、表名、索引列编号
- `IndexInfo` ：索引信息，包含索引定义、索引列的信息。后者是为了加速信息访问而设计的冗余结构。
- `CatalogManager` ：元数据管理器，内含 `map<string,TableInfo>` 和 `map<string,IndexInfo>` 两个结构，以表名/索引名为关键字，存储了所有表和索引的信息。STL map 对于加速元数据访问至关重要。

实现细节

从整体架构图中可以看出，CatalogManager 的特别之处在于它是唯一一个绕过 `BufferManager` 直接处理文件的模块。这样设计主要有两个原因：

1. 元数据是最频繁访问的信息，它应该常驻内存，因此我设计为系统一开始运行，所有元数据就从文件中载入，之后不再对元数据文件操作，直到系统退出时把元数据写回文件。既然 `CatalogManager` 只在程序开始、结束时操作文件，那么不走缓冲区的代价是可接受的。

2. 元数据较为复杂，按块存储的实现有一定复杂性，既然读写量不大，那么索性不按块存储，直接一路写进文件。不按块存储的代价是可接受的。

创建/删除 表/索引 的接口是容易实现的，只需对数据结构做相应修改，并维护相关信息即可。

Load & Dump

这里比较有趣的是从文件载入元数据 (`load()` 函数) 和将元数据写入文件 (`dump()` 函数) 的实现，应用了 C++ 模板及其全特化。以下代码示例取自 `catalog/catalog.cpp` 。

首先定义两个模板：

```
template<typename T> static void parse_from_stream(T &val,stringstream &is);
template<typename T> static void embed_to_stream(T &val,stringstream &os);
```

然后对每种可能出现的数据类型，实现全特化的版本，例如对 `string` ：

```
template<> void parse_from_stream<string>(string &val,stringstream &is) {
    size_t len;
    read_from_stream(len,is); // 读取长度
    char *str = new char[len+1];
    is.read(str,len),str[len] = 0; // 读取串内容
    val.assign(str);
    delete str;
}
template<> void embed_to_stream<const string>(const string &val,stringstream
&os) {
    size_t len = val.length();
    write_to_stream(len,os); // 写入长度
    os.write(val.c_str(),len); // 写入串内容
}
```

再如 `TableColumnDef` ：

```
template<> void parse_from_stream<TableColumnDef>(TableColumnDef
&val,stringstream &is) {
    read_from_stream(val.ord,is);
    // val.col_name 的类型是 string, 这里实际上调用了上面为 string 特化的函数
    parse_from_stream(val.col_name,is);
    read_from_stream(val.col_spec,is);
}
template<> void embed_to_stream<const TableColumnDef>(const TableColumnDef
&val,stringstream &os) {
    write_to_stream(val.ord,os);
    embed_to_stream(val.col_name,os); // 这里同理
    write_to_stream(val.col_spec,os);
}
```

如此一来整个逻辑非常清晰，每次只用关心当前的数据如何解析/写入。

虽然没有经过 `BufferManager` ，我还是用 `stringstream` 作为文件的临时缓冲区。

Recorder 模块

模块功能

管理表的记录文件：

1. 在创建表时初始化记录文件，删除表时删除记录文件
2. 提供插入记录、删除记录、获取记录的接口

具体实现

在元组定长的假设下，我采用了按块存储、块内按行存储、Free-Linked-List 结构处理空位的设计。

文件开头的 0 号块只有一条记录，而且只有指针，没有数据。对于从 1 号开始的块，设 `in_len` 为元组的字节数，则 `out_len=3*8 + in_len` 为一个元组在文件中占用的空间，其中 3*8 是 3 个 8 字节的无符号数 (实现为 `std::size_t`)，它们实际上是指针：

1. `pre_tup` ：上一个元组的开头在整个文件中的地址 (即文件的 `[pre_tup,pre_tup+out_len)` 字节为上一个元组) 。空位的这个指针为 0 。
2. `nxt_tup` ：下一个元组的开头在整个文件中的地址。空位或链尾的这个指针为 0 。
3. `nxt_spa` ：下一个空位开头的地址。非空位的这个指针为 0 。

这三个指针构造了两个链表，一是记录的双向链表，二是空位的单向链表。有了这两个链表，就能支持记录的插入、删除和遍历。记录的开始地址也可以作为此记录的指针，用在索引中，因为 `Recorder` 的实现保证记录插入之后删除之前位置不会改变。

代码片段

这里展示简化过的 插入记录 的(伪)代码：

```
size_t Recorder::place_record(vector<Literal> &rec) { // rec 是要插入的记录

    合法性判断,不合法则抛出异常;

    // 获取头指针指向的空位
    Block head = man.buf.get_block(ti.path,0,false);
    head.ink();
    assert(head.data);
    ptr head_p,the_p;
    head.seek(0),head.read(head_p);
    size_t pos = head_p.nxt_spa;
    size_t seg = pos / block_size; //
    size_t off = pos % block_size;

    // 调用缓冲区,载入空位所在块
    Block the = man.buf.get_block(ti.path,seg,true);
    the.ink();
    the.seek(off),the.read(the_p);
    if (the_p.nxt_spa == 0) the_p.nxt_spa = next_valid_pos(pos);
    // 写入记录
    size_t siz = rec.size();
    for (size_t i=0;i<siz;++i)
embed(rec.at(i),ti.def.col_def.at(i).col_spec.len,the);

    // 调整头和当前记录的 nxt_spa 指针
    head_p.nxt_spa = the_p.nxt_spa;
    the_p.nxt_spa = 0;

    // 调整头和当前记录的 pre_tup, nxt_tup 指针
    the_p.pre_tup = 0;
    the_p.nxt_tup = head_p.nxt_tup;
    head_p.nxt_tup = pos;
```

```
    // 将当前记录插入链表前端
    if (the_p.nxt_tup != 0) {
        size_t nxt_seg = the_p.nxt_tup / block_size;
        size_t nxt_off = the_p.nxt_tup % block_size;
        Block nxt = man.buf.get_block(ti.path,nxt_seg,false);
        assert(nxt.data);
        nxt.ink();
        ptr nxt_p;
        nxt.seek(nxt_off),nxt.read(nxt_p);
        nxt_p.pre_tup = pos;
        nxt.seek(nxt_off),nxt.write(nxt_p);
        nxt.unpin();
    }

    // 写入修改后的指针并在缓冲区中解除锁定
    head.seek(0),head.write(head_p);
    the.seek(off),the.write(the_p);
    head.unpin();
    the.unpin();
    return pos;
}
```

TmpManager 模块

模块功能

系统中有模块会用到这样一种文件，它只用一次，用完就删掉。这种文件称为临时文件。

本模块管理临时文件，提供两个简单的功能：

1. 创建临时文件
2. 销毁指定的临时文件

具体实现

实现这个模块有两种选择，一是利用系统命令 `mktemp` 创建临时文件，二是自己实现，最后我选择了后者，原因是：

1. 我不想在程序中引入新的组分
2. 自己实现很简单，也更容易控制

最终就写了一个简单的类：

```
class TmpManager {
private:
    Manager &man;
    size_t cnt;
    string tmp_name(size_t x) const;
public:
    TmpManager(Manager &_man);
    ~TmpManager();
    string new_tmp();
    void ret_tmp(const string &);
};
```

其中 `cnt` 是计数器，初始为 0 。 `new_tmp()` 获取新临时文件的文件名，那就是 `1.tmp` `2.tmp` `3.tmp` ，以此类推。 `ret_tmp(name)` 销毁临时文件。

这里可以体现自己实现这个类的另一个好处——如果用 `mktemp` ，不可避免地这个文件要创建出来，而自己实现可以不真的创建这个文件——BufferManager 会管理好的。

Resulter 模块

模块功能

Resulter 管理表的输出，用于 Select 语句、Show Tables 语句和 Show Indexes 的输出。

定义这个模块主要是为了方便地解决两个问题：

1. 输出一个表的时候，必须一开始就知道每列的宽度，但每列的宽度必须知道所有行的内容后才能算出来。也就是说，得到要输出内容和实际输出是不同步的——过程应该是 得到输出内容 - 缓存输出内容并计算列宽 - 从缓存中提取输出内容并格式化地输出。
2. 由于我的 Select 语句支持选择一些列，而这些列可能重复，所以为了效率应只缓存列的集合 (即去除重复的) ，输出时再按要求输出重复的列。

具体实现

此模块建立在 `TmpManager` 和 `BufferManager` 上，其中 `TmpManager` 为它分配新的临时文件用于缓存行内容， `BufferManager` 则提供抽象的文件块操作。

主要实现四个函数： `init` `add_tuple` `print` `finish` ，分别支持初始化、放入元组、输出、清理。放入的元组是顺序地、按块地存放在文件中。

可以发现，本模块实现的功能与 `Recorder` 有些类似，都支持把元组高效地放入文件中；然而本模块又有特殊性，插入和读取都是顺序的，而且需要支持格式化输出。因此本模块完全重新实现，没有应用 `Recorder` 。

由于实现直接而简单，所以细节不展开。

Lister<T> 模块

模块功能

维护一个数据类型为 `T` 的链表，支持顺序访问、后端插入。

为什么要额外实现一个这样的链表？STL 提供的 `list<T>` 不好吗？

问题在于，正如实验目标所确定的，本系统会处理非常大量的数据。面对海量的数据，不可能像 `list<T>` 那样把点全部存在内存中。换言之，我实现的 `Lister<T>` 是存在文件里的链表，它利用 `BufferManager` 实现高效的文件读写。

具体实现

非常简单，就是记录一些指针，通过 `BufferManager` 操作文件块。总共就 60 行，这里不展开了。

BufferManager 模块

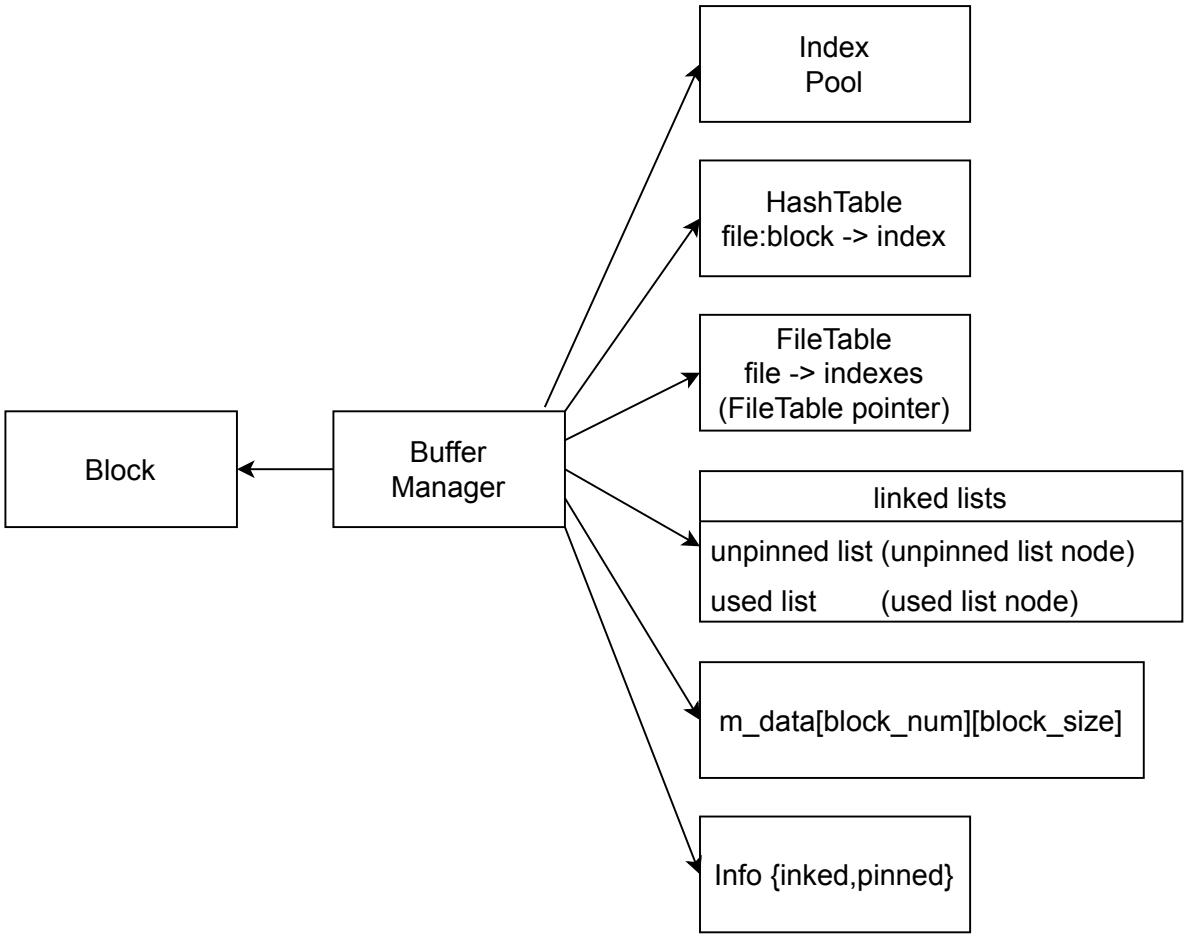
模块功能

作为数据库文件和大部分组件之间的中间层，`BufferManager` 提供**对文件系统中的一个文件的某个块的抽象**，支持：

1. 加载文件块进内存并返回包装类 (`get_block` 函数) 、将块写回对应的文件 (`flush_index` 函数，注意 `BufferManager` 中的 "index" 意为下标，与 `IndexManager` 中的 "index" 没有任何关系)
2. 记录块是否被修改过 (`inked`) ，固定 (`pin`) 块、取消固定 (`unpin`) 块

上述功能是对使用者展示的；为了高效实现这些外部功能，`BufferManager` 内部设计了一系列数据结构，支持 LRU 替换算法等内部功能。

内部设计



右侧是 `BufferManager` 内部数据结构，左边是 `BufferManager` 提供给使用者的接口—— `Block` 类。

- 块大小设置为 `block_size=4096` ，缓冲区中的块数为 `block_num` 。
- 一个文件块被载入缓冲区时会被分配一个下标 (即 index) i。这个下标来自下标池 IndexPool ，初始时其中有 `0...block_num-1` 的所有值，它类似一个栈，分配新下标就从栈顶取出值，丢弃下标就把值放回栈顶。
- HashTable 是**某文件中的某块**到其在 BufferManager 中被分配的下标的映射，用哈希表实现。
- FileTable 是 **某文件** 到 **这个文件的所有已经在 Buffer 中的块的下标**列表的映射，也用哈希表实现。同时每个有用下标会存储一个指向 FileTable 中此下标的指针，为了便于删除节点。FileTable 这个数据结构主要是为了满足这样一个需求：当一个文件被删除时 (例如删除表、删除索引、删除临时文件时) ，BufferManager 必须把缓冲区中这个文件对应的块都删除 (至少标记为删除) ，否则它们之后可能会被写出，产生文件没有被删掉的效果。
- unpinned list 是**所有在 Buffer 中但没有固定的块的下标**链表，按照从前到后从远到近的顺序排列。新取消固定的下标将被插在最后，需要腾出空间时开头的下标会被丢弃。这就实现了 LRU 替换算法。每个下标有一个指针指向其在 unpinned list 中的点 (如果这个下标没有使用过或被固定了，这个指针就是空指针)，这是为了便于删除。

- used list 是所有 Buffer 中的块的下标链表，同样每个下标有一个指针指向它在此链表中的点。
- m_data 是一个二维数组，若某块的下标是 i ，则 `m_data[i][]` 就是此块在缓冲区中的副本。
- 对每个下标还保存了 Info ，含有这个下标对应的块是否被修改过 (inked) 、是否被固定 (pinned) 。如果块准备写出时发现没有修改过，那么就无需写出，提高了效率。

小细节

如何哈希

这里有两个哈希表，其键值分别为 "(文件名,块编号)" 和 "文件名" 。如何哈希就成了一个问题。我对哈希没什么研究，在网上查到了一个据说比较好的，也很简单的字符串哈希算法 "djb2" 。那块编号怎么办呢？我的做法是在文件名后加上一个冒号，跟上块编号的十进制表示，作为字符串键值。例如 "文件 `dir/abc.cat` 的第 15 个块" 这条信息要哈希，那么先转化成字符串 `str = "dir/abc.cat:15"` ，再对 `str` 使用 djb2 算法哈希。实现起来很简单：

```
pair<size_t,size_t> BlockSpec::hash() const {
    static stringstream ss;
    ss.str("");
    // 这里没有取模, 利用 size_t 作为无符号数的自然溢出"取模"
    size_t ret1 = 5381;
    for (char c:file_path) ret1 = ret1 * 33 + c; // ret1 是直接对文件名哈希
    size_t ret2 = ret1 * 33 + ':'; // 接上一个冒号
    ss << ord; // 数字转字符串
    for (char c:ss.str()) ret2 = ret2 * 33 + c; // 继续把十进制表示的块号哈希
    return {ret1,ret2}; // ret2 就是 "文件路径文件名:块编号" 的哈希
}
```

多重使用一个块

如果向 BufferManager 索要的块已经在缓冲区中了 (这可以利用 HashTable 判断出来) ，那么直接包装并返回这个块即可。如果由于某些原因，上层多次向 BufferManager 索要同一个块，而其中一个索要者用完后把这个块取消固定，那么这个块可能会被换出去，而这个块实际上还被其他索要者引用着！因此，**pinned 不只一个 `bool` 值表示此块是否固定，它应该是一个引用计数** 。每次上层索要一个块，就给它的 `pinned` 加一；上层 "取消固定" 一个块，就给其 `pinned` 减一；块的下标在 unpinned list 中当且仅当其 `pinned=0` 。

Indexer 模块

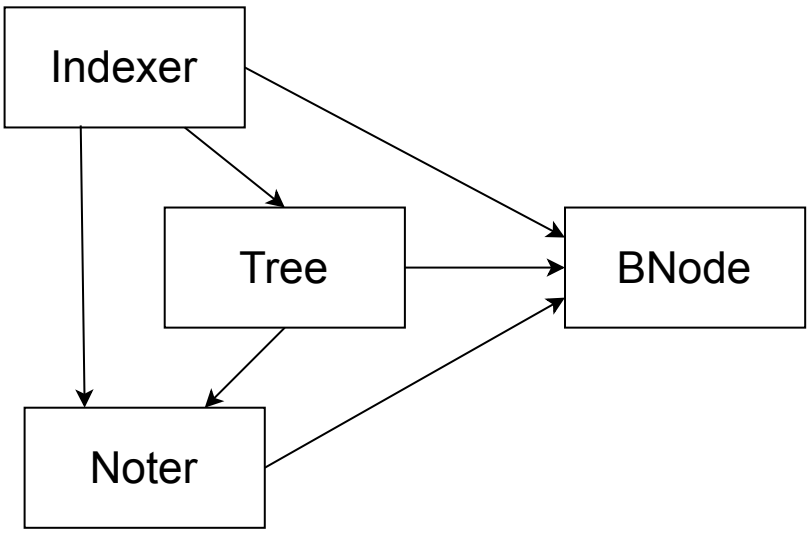
模块功能

维护索引，向上提供功能：

1. 初始化索引文件、删除索引文件
2. 插入记录、删除记录
3. Find : 最左边的叶子、第一个大于给定值的叶子位置、第一个大于等于给定值的叶子位置
4. 根据地址取出一个 B+ 树节点

`Indexer` 不是一个一直运行着的索引管理器，而是每次需要用索引的时候就地创建的类。这个类本身不保存信息，而是作为操控 Index 文件中信息的把手。这也是为什么我不把它叫做 IndexManager 而叫做 `Indexer` 。

对象设计



(其实这里画的不是依赖图也不是继承关系图，只是表达类的使用关系)

实现细节

`Indexer` 模块算是一个比较复杂的模块了，但其目标很明确，所以也并不难。

这部分细节放在我的个人报告中详细讨论。

运行分析

这部分对每种语句分析其执行流程，展示运行示例和相关错误提示。

执行流程分析会省略部分无关紧要的细节。

Create Table Statement

执行流程

1. 做各种错误检查
2. 请求 `CatalogManager` 创建表的元数据
3. 请求 `Recorder` 初始化表文件
4. 为主键自动创建索引

运行示例

正常运行：

```
>>> create table tab (
... x int,
... y float,
... z char(10) unique,
... u int,
... v float unique,
... primary key (x)
... );
OK ( 1.874 ms )
```

表已存在：

```
>>> create table tab (a int,primary key (a));
Invalid Operation : table 'tab' already exists
```

列数过多：

```
>>> create table two (x1 int,x2 int,x3 int,x4 int,x5 int,x6 int,x7 int,x8 int,x9
 int,x10 int,x11 int,x12 int,x13 int,x14 int,x15 int,x16 int,x17 int,x18 int,x19
 int,x20 int,x21 int,x22 int,x23 int,x24 int,x25 int,x26 int,x27 int,x28 int,x29
 int,x30 int,x31 int,x32 int,x33 int,primary key (x1));
Invalid Table Definition : too many (33 > 32) columns
```

超出一块：

```
>>> create table two (x1 char(200),x2 char(200),x3 char(200),x4 char(200),x5 cha
r(200),x6 char(200),x7 char(200),x8 char(200),x9 char(200),x10 char(200),x11 cha
r(200),x12 char(200),x13 char(200),x14 char(200),x15 char(200),x16 char(200),x17
 char(200),x18 char(200),x19 char(200),x20 char(200),x21 char(200),x22 char(200)
,x23 char(200),x24 char(200),x25 char(200),x26 char(200),x27 char(200),x28 char(
200),x29 char(200),x30 char(200),primary key (x1));
Invalid Table Definition : tuple total len = 6024 cannot fit into a block
```

列名重复：

```
>>> create table three (x int,y float,x char(10),primary key (x));
Invalid Table Definition : repeat column name
```

主键非列：

```
>>> create table four (x int,primary key (y));
Invalid Table Definition : primary key not in column list
```

Drop Table Statement

执行流程

1. 错误检查
2. 向该表上的所有索引的 `Indexer` 请求删除此索引
3. 请求 `Recorder` 删除表文件
4. 请求 `CatalogManager` 删除表元数据

执行示例

```
>>> drop table tab;
OK 1 table dropped ( 0.760 ms )

>>> drop table tab;
Invalid Drop Table Statement : table 'tab' not exists

>>> drop table tab if exists;
OK 0 table dropped ( 0.004 ms )
```

Create Index Statement

执行流程

1. 错误检查

2. 请求 `CatalogManager` 创建索引元数据
3. 请求 `Indexer` 初始化索引文件
4. 利用 `Recorder` 遍历整个表，逐个把表中的元组插入索引中 (这里没有实现自底向上 B+ 树构建，因为时间不够了)

运行示例

```
>>> create table tab (x int,y float unique,z char(10),primary key(x));
OK ( 1.232 ms )

>>> create index idx on tab (y);
OK ( 1.536 ms )

>>> create index idx on tab (x);
Invalid Create Index Statement : index 'idx' already exists

>>> create index two on tbb (x);
Invalid Create Index Statement : table 'tbb' not exists

>>> create index two on tab (u);
Invalid Create Index Statement : column 'u' not in table 'tab'

>>> create index two on tab (z);
Invalid Create Index Statement : column 'z' is not defined as unique
```

Drop Index Statement

执行流程

1. 错误检查
2. 请求 `Indexer` 删除索引文件
3. 请求 `CatalogManager` 删除索引元数据

运行示例

```
>>> drop index idx if exists;
OK 1 index dropped ( 0.856 ms )

>>> drop index idx if exists;
OK 0 index dropped ( 0.006 ms )

>>> drop index idx;
Invalid Drop Index Statement : index 'idx' not exists
```

Select Statement

执行流程

1. 错误检查
2. 获取符合要求行在表文件中的地址
    1. 根据查询条件选择查询方式：若有索引列上的等值条件，则用此索引此条件驱动查询；否则，若有索引列上的大小比较条件，则用此索引此条件驱动搜索；否则暴力查询

2. 执行选择的查询方式：若是 "等于"、"大于等于" 驱动查询，则找到 B+ 树上第一个大于等于给定值的，从那里开始往后遍历 B+ 树的叶子，直到驱动查询的条件不满足；若是 "大于" 驱动查询，则从第一个大于给定值的叶子位置开始往后遍历；若是 "小于"、“小于等于” 驱动查询，则从 B+ 树上最左边的叶子开始遍历叶子，直到驱动条件不满足。
3. 将满足所有查询条件 (and) 的记录的地址存进 Lister 并返回。

3. 利用 `Recorder` 取出所有结果元组，放入 `Resulter`
4. 根据投影列，用 `Resulter` 进行输出

运行示例

用验收的数据 (debug 模式)：从运行时间上可以看到明显的索引优化效果

```
>>> select * from student2 where id = 1080100005;
+------------+-------+---------+
| id         | name  | score   |
+------------+-------+---------+
| 1080100005 | name5 | 72.5000 |
+------------+-------+---------+
1 rows in result ( 2.296 ms )

>>> select * from student2 where id < 1080100005;
+------------+-------+---------+
| id         | name  | score   |
+------------+-------+---------+
| 1080100001 | name1 | 99.0000 |
| 1080100002 | name2 | 52.5000 |
| 1080100003 | name3 | 98.5000 |
| 1080100004 | name4 | 91.5000 |
+------------+-------+---------+
4 rows in result ( 3.274 ms )

>>> select * from student2 where id >= 1080109996;
+------------+-----------+---------+
| id         | name      | score   |
+------------+-----------+---------+
| 1080109996 | name9996  | 65.5000 |
| 1080109997 | name9997  | 61.0000 |
| 1080109998 | name9998  | 84.5000 |
| 1080109999 | name9999  | 69.5000 |
| 1080110000 | name10000 | 80.5000 |
+------------+-----------+---------+
5 rows in result ( 2.767 ms )
```

```
>>> select * from student2 where name > 'name999';
+------------+----------+---------+
| id         | name     | score   |
+------------+----------+---------+
| 1080109999 | name9999 | 69.5000 |
| 1080109998 | name9998 | 84.5000 |
| 1080109997 | name9997 | 61.0000 |
| 1080109996 | name9996 | 65.5000 |
| 1080109995 | name9995 | 59.5000 |
| 1080109994 | name9994 | 97.5000 |
| 1080109993 | name9993 | 67.5000 |
| 1080109992 | name9992 | 50.5000 |
| 1080109991 | name9991 | 69.0000 |
| 1080109990 | name9990 | 75.0000 |
+------------+----------+---------+
10 rows in result ( 18.619 ms )

>>> create index idx on student2 (name);
OK ( 322.090 ms )

>>> select * from student2 where name > 'name999';
+------------+----------+---------+
| id         | name     | score   |
+------------+----------+---------+
| 1080109990 | name9990 | 75.0000 |
| 1080109991 | name9991 | 69.0000 |
| 1080109992 | name9992 | 50.5000 |
| 1080109993 | name9993 | 67.5000 |
| 1080109994 | name9994 | 97.5000 |
| 1080109995 | name9995 | 59.5000 |
| 1080109996 | name9996 | 65.5000 |
| 1080109997 | name9997 | 61.0000 |
| 1080109998 | name9998 | 84.5000 |
| 1080109999 | name9999 | 69.5000 |
+------------+----------+---------+
10 rows in result ( 1.979 ms )
```

```
>>> select id,name,id from student2 where name < 'name100';
+------------+--------+------------+
| id         | name   | id         |
+------------+--------+------------+
| 1080100001 | name1  | 1080100001 |
| 1080100010 | name10 | 1080100010 |
+------------+--------+------------+
2 rows in result ( 2.359 ms )

>>> drop index idx;
OK 1 index dropped ( 1.163 ms )

>>> select id,name,id from student2 where name < 'name100';
+------------+--------+------------+
| id         | name   | id         |
+------------+--------+------------+
| 1080100010 | name10 | 1080100010 |
| 1080100001 | name1  | 1080100001 |
+------------+--------+------------+
2 rows in result ( 17.650 ms )

>>> select id,name,id from student2 where name < 'name100' and id < 1080100100;
+------------+--------+------------+
| id         | name   | id         |
+------------+--------+------------+
| 1080100001 | name1  | 1080100001 |
| 1080100010 | name10 | 1080100010 |
+------------+--------+------------+
2 rows in result ( 0.982 ms )
```

limit 子句和查询条件错误提示：

```
>>> select * from student2 where id > 10801000235 limit 10;
cannot convert '10801000235' to INT
>>> select * from student2 where id > 1080100235 limit 10;
+------------+---------+---------+
| id         | name    | score   |
+------------+---------+---------+
| 1080100236 | name236 | 86.5000 |
| 1080100237 | name237 | 95.5000 |
| 1080100238 | name238 | 56.0000 |
| 1080100239 | name239 | 89.5000 |
| 1080100240 | name240 | 96.5000 |
| 1080100241 | name241 | 84.5000 |
| 1080100242 | name242 | 98.5000 |
| 1080100243 | name243 | 84.0000 |
| 1080100244 | name244 | 84.0000 |
| 1080100245 | name245 | 62.5000 |
+------------+---------+---------+
10 rows in result ( 52.698 ms )
```

"empty set"：

```
>>> select * from student2 where name = '';
empty set ( 16.772 ms )
```

其他错误提示：

```
>>> select * from tab;
Invalid Select Statement : table 'tab' not exists

>>> select idd from student2;
Invalid Select Statement : project column 'idd' not exists

>>> select * from student2 where name = 123;
Invalid Select Statement : cannot compare column 'name' (CHAR) and literal [123] (INT)

>>> select * from student2 where idd = 123;
Invalid Select Statement : column 'idd' not in table 'student2'

>>> select * from student2 limit -10;
Invalid Select Statement : limit number must be nonnegative

>>> select * from student2 limit '123';
Invalid Select Statement : limit number must be INT
```

Insert Statement

执行流程

1. 错误检查
2. 对插入的元组尝试做类型转换 (例如 `FLOAT` 变 `INT`) 使之适于该表
3. 对插入的元组检查完整性约束 (即 primary key & unique 的限制)
4. 请求 `Recorder` 插入元组，得到元组在表文件中的地址
5. 请求该表上的每个索引的 `Indexer` ，将该记录与地址插入索引中 (内部调用 B+ 树的插入函数)

运行示例

```
>>> create table tab (x int,y int unique,primary key (x));
OK ( 0.798 ms )

>>> insert into tab values (1,1);
OK 1 tuple inserted ( 1.250 ms )

>>> insert into tab values (2,1);
Integrity Constraint Violated : tuple (2,1,) violated at least one uniqueness constraint

>>> insert into tab values (1,2);
Integrity Constraint Violated : tuple (1,2,) violated at least one uniqueness constraint
```

```
>>> insert into tab values (5,4.3);
Invalid Insert Tuple : tuple (5,4.30000,) does not fit into this table

>>> insert into tab values (5,4,3);
Invalid Insert Tuple : tuple (5,4,3,) does not fit into this table
```

这里顺便说一下，写报告的过程中我发现了一个关于 unique 约束检查的 bug。之前验收的时候只用 600ms 就插入了所有验收数据，其实是这个 bug 导致的，实际上应该没那么快。目前我在 release 模式下测试插入所有数据用时 12s 。这么长的时间主要是 unique 约束检查导致的。我想到一种优化策略，但没时间实现了，那就是：

1. 如果所有 unique 列上都有索引，那么所有 unique 约束检查都用索引进行，这非常快
2. 否则信息不足以优化这个操作，必须整个表遍历一遍来检查

Delete Statement

执行流程

1. 错误检查

2. 像 Select Statement 中那样，找出所有待删除记录在文件中的地址
3. 对每条应删除的记录，请求 `Recorder` 在表文件中删除这条记录；请求该表所有索引的 `Indexer` ，在索引中删除这条记录

运行示例

```
>>> delete from student2 where name < 'name100';
OK 2 rows deleted ( 6.785 ms )

>>> delete from student2 where id > 1080100050;
OK 9950 rows deleted ( 99.031 ms )

>>> delete from student2;
OK 48 rows deleted ( 1.511 ms )

>>> delete from student3;
Invalid Delet Statement : table 'student3' not exists
```

Execfile Statement

执行流程

1. 检查文件是否存在，是否可以打开
2. 建立新 `Interpreter` ，启动语法解析

运行示例

```
drop table tab if exists;

create table tab (x int,y char(10),primary key (x

insert into tab values (1,'a');
insert into tab values (2,'b');
insert into tab values (3,'c');
insert into tab values (4,'d');
insert into tab values (5,'e');
insert into tab values (6,'f');
insert into tab values (7,'g');
insert into tab values (8,'h');
insert into tab values (9,'i');
insert into tab values (10,'j');
insert into tab values (11,'k');
insert into tab values (12,'l');
insert into tab values (13,'m');
insert into tab values (14,'n');
insert into tab values (15,'o');
insert into tab values (16,'p');
insert into tab values (17,'q');
insert into tab values (18,'r');
insert into tab values (19,'s');
insert into tab values (20,'t');
insert into tab values (21,'u');
insert into tab values (22,'v');
insert into tab values (23,'w');
insert into tab values (24,'x');
insert into tab values (25,'y');
insert into tab values (26,'z');

select * from tab where x < 10 and y >= 'd';

drop table tab;
~
~
"a.sql" 34L, 993C written
```

```
>>> create table tab (x int,primary key (x));
OK ( 1.278 ms )

>>> execfile "a.sql";
OK 1 table dropped ( 0.573 ms )

OK ( 0.479 ms )

OK 1 tuple inserted ( 0.360 ms )

OK 1 tuple inserted ( 0.174 ms )

OK 1 tuple inserted ( 0.169 ms )

OK 1 tuple inserted ( 0.169 ms )

OK 1 tuple inserted ( 0.170 ms )

OK 1 tuple inserted ( 0.181 ms )

OK 1 tuple inserted ( 0.143 ms )

OK 1 tuple inserted ( 0.143 ms )

OK 1 tuple inserted ( 0.142 ms )

OK 1 tuple inserted ( 0.145 ms )

OK 1 tuple inserted ( 0.151 ms )

OK 1 tuple inserted ( 0.148 ms )

OK 1 tuple inserted ( 0.149 ms )

OK 1 tuple inserted ( 0.150 ms )

OK 1 tuple inserted ( 0.157 ms )

OK 1 tuple inserted ( 0.185 ms )

OK 1 tuple inserted ( 0.152 ms )

OK 1 tuple inserted ( 0.152 ms )

OK 1 tuple inserted ( 0.153 ms )

OK 1 tuple inserted ( 0.164 ms )

OK 1 tuple inserted ( 0.156 ms )
```

```
OK 1 tuple inserted ( 0.157 ms )

OK 1 tuple inserted ( 0.159 ms )

OK 1 tuple inserted ( 0.159 ms )

OK 1 tuple inserted ( 0.160 ms )

OK 1 tuple inserted ( 0.169 ms )

+---+---+
| x | y |
+---+---+
| 4 | d |
| 5 | e |
| 6 | f |
| 7 | g |
| 8 | h |
| 9 | i |
+---+---+
6 rows in result ( 0.706 ms )

OK 1 table dropped ( 0.260 ms )

Execfile Finish ( 8.697 ms )
```

- ## Show Statement

执行流程

1. 从 `CatalogManager` 处获取表或索引的信息
2. 利用 `Resulter` 输出结果

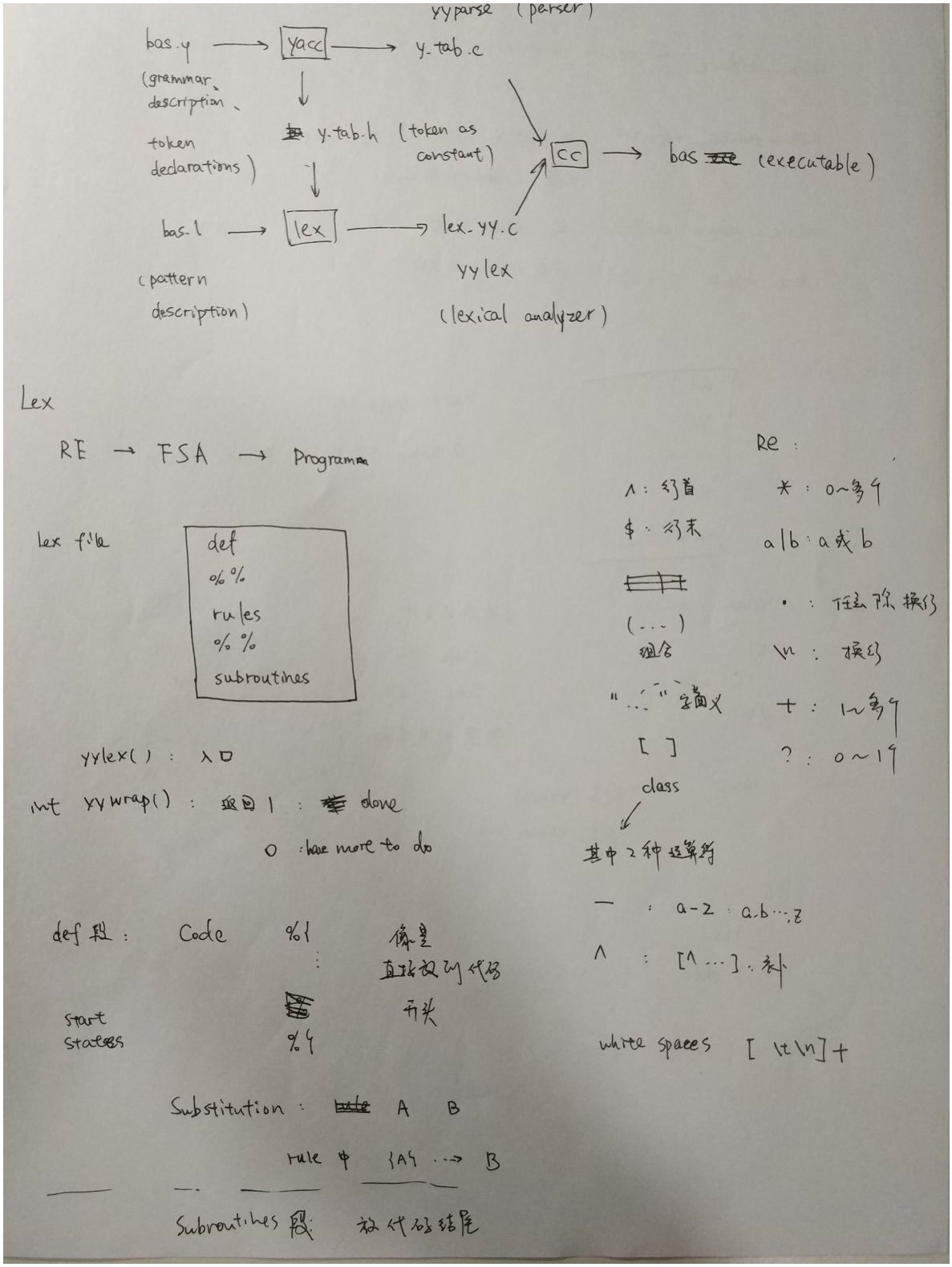
运行示例

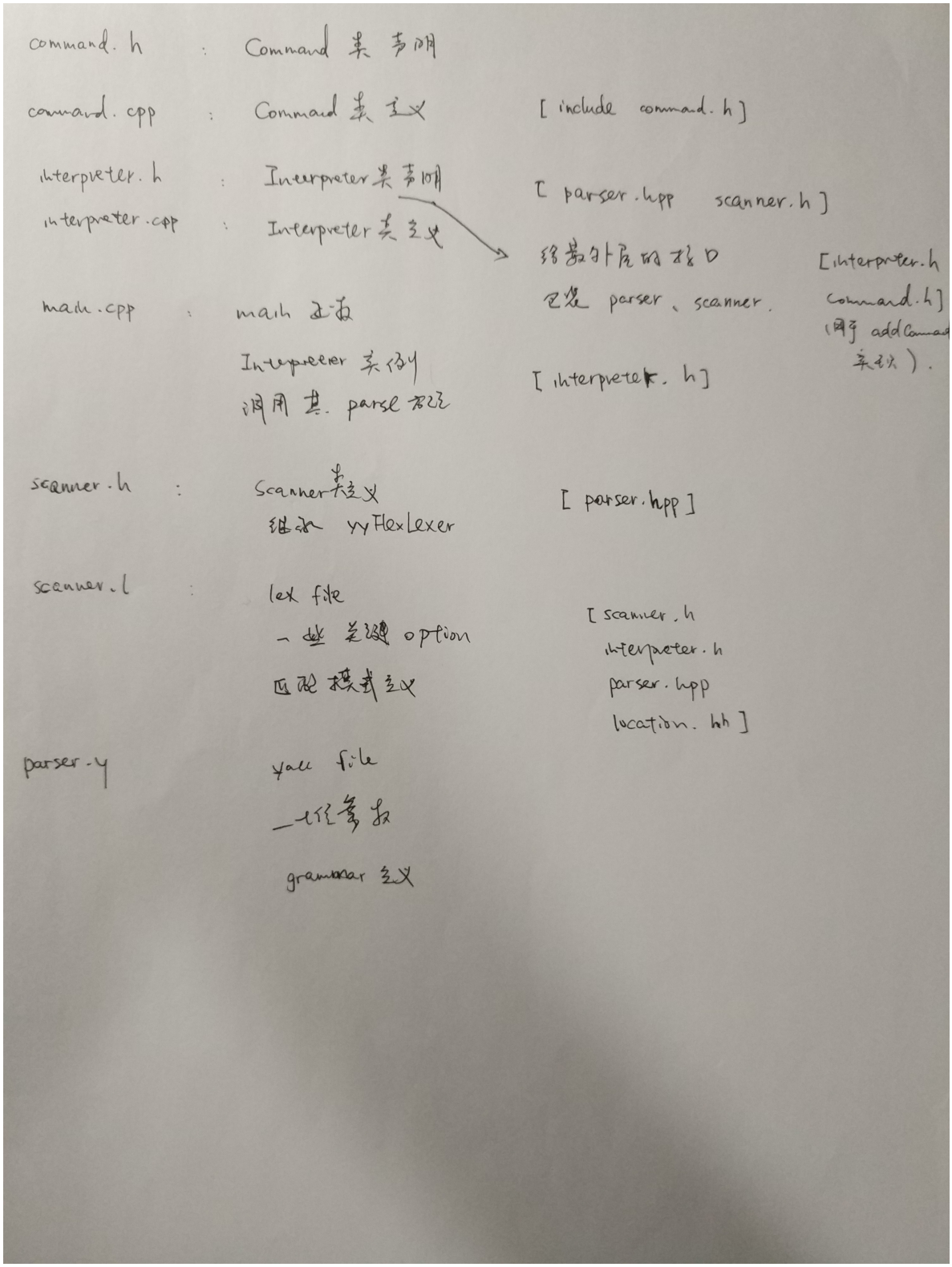
```
>>> show tables;
empty result ( 0.212 ms )

>>> create table A (x int,y char(10) unique,primary key (x));
OK ( 2.340 ms )

>>> show tables;
+--------+
| tables |
+--------+
| A      |
+--------+
1 rows in result ( 1.554 ms )

>>> show indexes;
+------------+-------+--------+
| index name | table | column |
+------------+-------+--------+
| A_PI       | A     | x      |
+------------+-------+--------+
1 rows in result ( 1.450 ms )

>>> create table B (a float unique,b char(5),primary key (b));
OK ( 1.573 ms )

>>> create index bdx on B (a);
OK ( 1.260 ms )

>>> show tables;
+--------+
| tables |
+--------+
| A      |
| B      |
+--------+
2 rows in result ( 1.068 ms )

>>> show indexes;
+------------+-------+--------+
| index name | table | column |
+------------+-------+--------+
| A_PI       | A     | x      |
| B_PI       | B     | b      |
| bdx        | B     | a      |
+------------+-------+--------+
3 rows in result ( 1.426 ms )
```

如何测试

1. 确保安装好合适版本的环境 (见 实验环境 小节) 。
2. 确保安装了较新版本的 `make` 。
3. 命令行进入 `src/` 目录，运行 `make release` 或 `make debug` ，就能自动编译。可执行文件是当前目录下的 `ui` ，即输入命令 `./ui` 就能运行 MeSQL Shell 了。
4. `make clean` 能将目录恢复到 `make` 之前的状态。

设计草稿

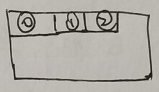

yyparse (parser)
bas.y
yacc
y.tab.c
(grammar description)
token declarations)
y.tab.h (token as constant)
cc
bas (executable)
bas.l
lex
lex.yy.c
yylex
(pattern description)
(lexical analyzer)
Lex
RE → FSA → Programa
lex file
def
%%
rules
%%
subroutines
yylex() : 入口
int yywrap() : 返回1 : done
0 : have more to do
def 段 : Code %{
%}
start states
Substitution :
rule 中 {A} --> B
Subroutines 段: 放代码结尾
Re :
^ : 行首
$: 行末
* : 0~多个
a|b : a或b
. : 任意除换行
(...) 组合
\n : 换行
"..." 字面义
+ : 1~多个
[ ]
? : 0~1个
class
其中2种运算符
- : a-z : a,b...,z
^ : [^...] : 补
white spaces [\t\n]+

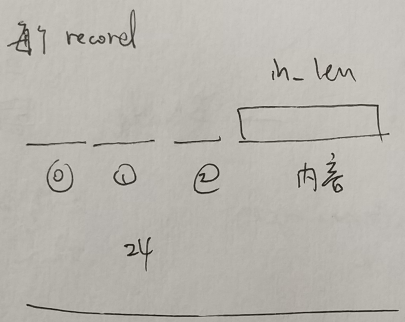

command.h : Command 类声明
command.cpp : Command 类定义 [include command.h]
interpreter.h : Interpreter 类声明 [parser.hpp scanner.h]
interpreter.cpp : Interpreter 类定义
给易外层的接口
包含 parser、scanner.
[interpreter.h command.h] (用于 addCommand 类型)
main.cpp : main 函数
Interpreter 类实例
调用其 parse 函数
[interpreter.h]
scanner.h : Scanner 类定义
继承 yyFlexLexer
[parser.hpp]
scanner.l : lex file
一些关键 option
匹配模式定义
[scanner.h interpreter.h parser.hpp location.hh]
parser.y yacc file
一些参数
grammar 定义

record manager (table Info) 每次用生成一个实例

- init_table：初始化此表（初始化文件头） 在 Catalog Manager 中
  不删表不全支 ⇒ 存引用
- ~~查~~
  place_record (vector<Literal>)
- ~~查~~ erase_record_at (size_t pt)

初始
⓪ pre_tup 0 ≥4
① nxt_tup 0
② nxt_spa block_size





pre_tup、nxt_tup
无 → 0

nxt_spa 最后一个空闲

~~文件第一~~
~~不存在块的开头~~
next_valid_position
(存得下时)
每一个位置

始终保证
最后一个 ~~free~~ space

~~存文件~~ 指向空在文件最后
(存在 / 不存在)

Index

Indexer (IndexBuff) 最外层 index 相关接口

.init. ...

↓

Tree (基于 path) B+ 树

↓

Noter: B+ 树上相关接口

生成新点、删点 → BNode 点对象

↓

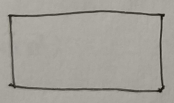
取点

和文件(BufferManager)、block 直接交互

若修改须提交给 Noter

Buffer Manager

B+ 树结点



block.

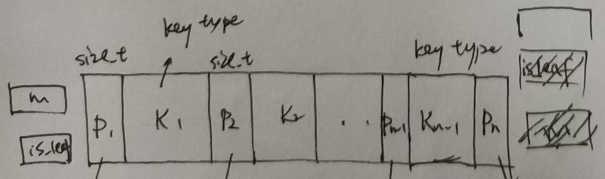
index 文件中

① 空 block

开头一个 size_t:

下一个空 block 编号

始终保证链尾一定指向不存在的第一个 block



P: 指向的 block → 某叶子的编号

指向此表 .tab 文件中此记录的文件内 pos

P=0: 空指针

sizeof(key type) ≤ 32.

$n = \left\lfloor \frac{B+K-1-5}{K+5} \right\rfloor$

m is_leaf $P_0 \cdots P_{n-1}$ $K_0 \cdots K_{n-2}$

Catalog Manager.

块大小：4096 bytes

目的：一个文件一个常驻内存的类实例

static const 的 size_t

block_size = 4096

buffer_size =

catalog 存的：

① 表定义

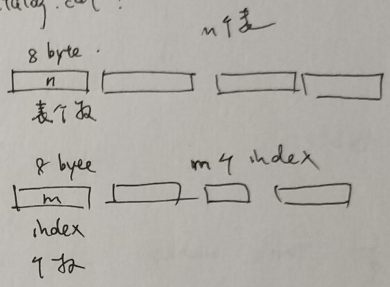
② 索引定义

实现：

dump_vec(), load_vec()

dump(), load()

catalog.cat：



Buffer manager

表：TableInfo. dump、load

index：IndexInfo. dump、load

实现：取一个文件的一个 block.

. 创建一个文件的一个 block

锁定、是否修改过.

BlockSpec { ~~file~~ path, ord }.

↑

hash

hash table. <BlockSpec*, ~~int~~ >

此 block 在 buffer 中的位置

(无：-1)

Block 对象

public: char* data

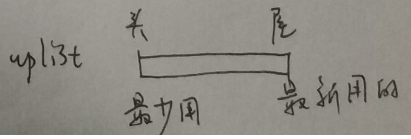
private: list_node

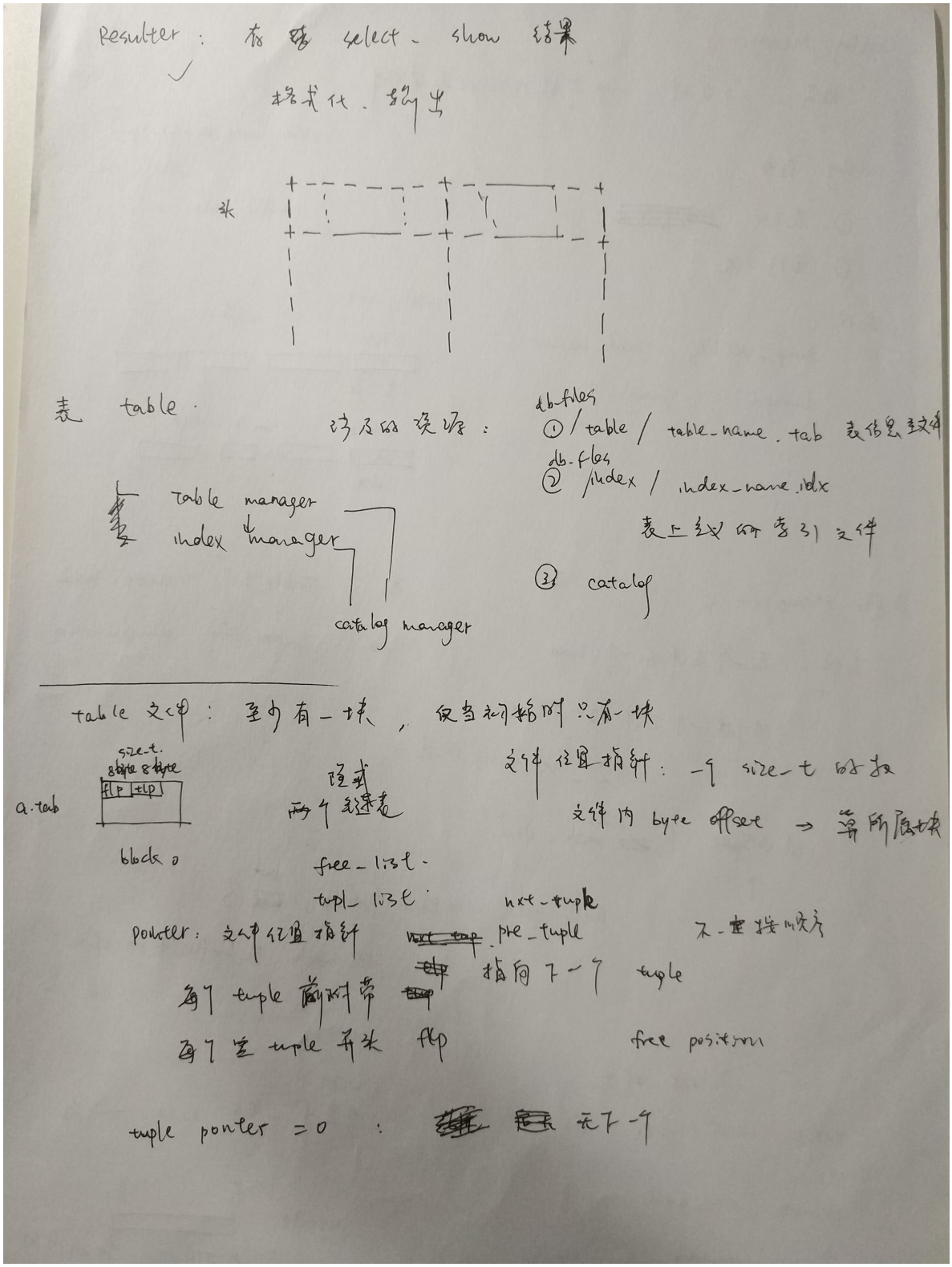
pinned

BufferManager &

List

链表中一个点 node 一般.




Resulter : 存储 select、show 结果
格式化输出
表 table
db_files
① / table / table_name.tab 表的元文件
db_files
② / index / index_name.idx
表上建的索引文件
③ catalog
table manager
index manager
catalog manager
table 文件：至少有一块，仅当初始时只有一块
size_t
a.tab
block 0
链式
两个链表
文件位置指针：一个 size_t 的数
文件内 byte offset → 第几块
free_list
tuple_list
next_tuple
pointer：文件位置指针
pre_tuple
不一定按顺序
指向下一个 tuple
每个 tuple 前对等
每个 tuple 开头 flg
free position
tuple pointer = 0 :
无下一个

Github 仓库