程序框架说明

> By 李易非

一、设计简介

本次工程中需要实现 Interpreter，API，Record Manager，Catalog Manager，Buffer Manager，Index Manager；

- Interpreter
- API 是程序的核心，它直接接收 Interpreter 提供的数据信息，调用各个 Manager 来完成数据库的插入 / 删除 / 更新；
- Record Manager 管理各个表的具体记录；
- Catalog Manager 管理数据库的元信息；
- Buffer Manager 管理数据库与文件层的交互；
- Index Manager 管理数据库中的索引 (B+ 树) ；

  各个 Manager 相互合作，共同维护数据库的数据；

二、重要类简介

在工程中，各个 Manager 的具体介绍可详见各 Manager 部分，这个部分主要介绍一些其它的重要类；

3.1 MiniSQL

工程中面临的一大难题是 Manager 之间的互相调用，以及 API 对于各个 Manager 的调用。我们的解决方案是实现一个 MiniSQL 类，它拥有各个 Manager 的静态成员变量，这样做的理由如下：

- 各个 Manager 并非 API 的成员变量，也并非 Interpreter 的成员变量，要在不同函数内部调用不同 Manager 是一件棘手的事情；
- 整个程序共享一份 Manager，一定程度上反映了 **全局性**

  具体类定义如下：

```
class MiniSQL
{
    private:
    static API & api;
    static CatalogManager & catalog_manager;
    static RecordManager & record_manager;
    static BufferManager & buffer_manager;
    static IndexManager & index_manager;
```

```
    public:
    static API & get_api();
    static CatalogManager & get_catalog_manager();
    static RecordManager & get_record_manager();
    static BufferManager & get_buffer_manager();
    static IndexManager & get_index_manager();

};
```

- 使用范例
  - `API & api = MiniSQL::get_api()` ，Interpreter 可以正常调用 API ,实现对应功能；
  - `CatalogManager & catalogmanager= MiniSQL::get_catalog_manager()`, API或者其它需要元数据的 Manager 可以轻松调用 Catalog Manager，提取需要信息；

3.2 Method

工程中面临的一大难题是对于数据文件的读写，数据库文件都是二进制文件，不同的解释方式决定了不同的信息，我们实现了 `Method` 类，其中包含整个数据库都需要的函数；

具体类定义如下：(简化版)

```
class Method
{
    public:
    //这个函数内部有new

    static int rawdata2int(const char * rawdata);
    static float rawdata2float(const char * rawdata);
    static void rawdata2string(const char * rawdata, int length, string &
out_str);
    static void string2rawdata(const string & str, const int type, char *
rawdata);
    static bool isSatisfyConditon(const char * rawdata, const int cond,
const string & operand, const int type);
    static const int getLengthFromType(int type);
};
```

具体说明

- `rawdata2int` 将一段 rawdata 翻译为整型变量；
- `rawdata2float` 将一段 rawdata 翻译为 float 型变量；
- `rawdata2string` 将一段 rawdata 翻译为 string;

- `string2rawdata` 将 string 翻译为 rawdata;
- `isSatisfyCondition` 给定 rawdata 与它的类型，判断这段record是否符合对应的条件与操作数;
- `getLengthFromType(type)`
  - 介绍这个函数之前，有必要介绍工程中对数据类型的处理，利用宏定义
    - `#define TYPE_INT 256`
    - `#define TYPE_FLOAT 257`
    - `#define TYPE_CHAR 255`
    - `#define MIN_TYPE_LENGTH 4`
  - 因为 varchar 类型变量最长为 255 bytes，小于255的类型一定是 varchar 类型变量，此时它的 data_type 就是它的实际长度;
  - 因为 free_list 的维护至少需要 4 bytes来标记下一条删除记录的位置，所以最短的类型长度为4，varchar(j) j < 4 均被看作 varchar(4)
  - Int / float 正常翻译即可;

```
switch(type)
    case TYPE_INT : 长度 = 4;
    case TYPE_FLOAT : 长度 = 4;
    default :
        if(type < 4)
            长度 = 4;
        else
            长度 = type;
```

3.3 Attribute

Attribute 的信息对于数据库至关重要，Attribute 类中含有 属性名称，属性类型，属性长度，是否 Primary, 是否 Unique;

```
class Attribute
{
    string name;
    short type;
    int length;
    bool isPrimary;
    bool isUnique;

};
```

3.4 Table

Table 类存储在 CatalogManger 中，通过 CatalogManger 的 get_table 函数，数据库可以获得一个表的所有信息；

```
class Table
{
    string table_name;
    int record_length;                                  // 记录的逻辑长度
    unordered_map <string, Attribute> Name2Attri; //名字到属性的hash
    unordered_map <string, int > Name2Pos;  //属性名字到属性位置的hash
};
```

3.5 Block

Block 类中包含文件名，Block ID，Block内容信息；它由 BufferManager 进行集中管理；

```
class Block
{
    string filename;            // Block 所在的文件
    int block_id;               // Block 在文件中的位置
    bool dirty;                 // Block 是否被修改过
    char * content;             // Block 中的真实内容

};
```

三、文件存储方式

3.1 设计简介

本次实验中，有3种类型的文件

1. 表的 metadata；

2. 索引的 metadata;
3. 数据文件；
4. 索引文件；

   前三种文件我们进行了**同质化处理**，可以用相同的处理方式来读对应数据，第四种文件因为 B+ 树的特性，采取了不同的存储方式；

3.2 具体实现

3.2.1 数据文件基本约定

3.2.1.1 物理存储说明

- 采用**定长记录**方法，`varchar(n)` 将在数据文件中分配 n 个字节的空间，不会因为具体记录长度的变化而变化；
- 采用 **Heap File** 方法，即文件内部 record 没有固定的存储顺序，通过 **free list** 来标记所有被删除的记录。free list header 在文件中占据4位 (int)，若为 -1 则说明到了 free list 尾部，若不为1，则代表下一条无效 record 在文件中的**绝对地址**。其余被删除记录的前四位与上述逻辑相同，若为 -1 则说明到了 free list 尾部，若不为1，则代表下一条无效 record 在文件中的**绝对地址**；
- 每一条数据后新增一个 **有效位** ，在遍历文件过程中，仅仅通过 **free list** 来判断一个 record 是否被删除非常耗时，最终我们决定 **空间换时间**， 在每一条记录后都添加一个有效位，若为1则表示有效记录，若为0则表示无效记录；
- 记录 **不会跨块存储** ，在一个 Block 不足以容纳一条记录时，直接前往下一个 Block；

3.2.1.2 逻辑存储说明

- 每个数据文件都有自己的 metadata，存储在所有文件的 **第一个 Block** 中，metadata包含
  - Record Length (物理长度，包含了有效位)
  - free list header，free list 链表头，指向一个被删除的记录；
  - Record Count，代表文件中记录总数，注意这里包含无效记录。这是为了确认文件 **EOF** 的具体位置；
- 其余的 Block 正常存储数据，若为无效记录，则前四位指向 free list 下一条无效记录；若为有效记录，则是一个 record 的正常存储单元；

3.2.1.3 注意事项

- 逻辑 Record Length 为物理 Record Length 减1；
- 因为要维护一个 free list，我们为每一个指针都分配了四个字节的空间，所以对于 varchar(j) (j < 4) 类型，我们都按照 varchar (4) 进行处理；
- Record Count，代表文件中记录总数，注意这里包含无效记录。这是为了确认文件 **EOF** 的具体位置；

3.2.2 具体数据文件存储内容

3.2.2.1 Table Meta 存储说明

Table Meta 存储在特定文件夹 `TableMeta/` 中，其中包含一个提供所有 Table 名字的 `TableMeta/tables` 文件，以及各个 Table 对应的 Meta Data 文件 `TableMeta/Table`

- `TableMeta/tables` 存储数据库中所有表名 (32 bytes) ；
- `TableMeta/Table` 存储特定 table 的属性信息，文件中的一条记录存储属性名 (32 bytes) 、数据类型 (2 bytes) 、是否主键 (1 byte) 、是否唯一 (1 byte) ；

  把 Table Meta 又分成 tables 与 特定 table 是为了保持同一个文件的 **record length 一致**，因为不同表的属性个数是不同的，如果存储在一起，很难在不浪费空间的前提下保持记录长度的一致性；

3.2.2.2 Index Meta 存储说明

Index Meta 存储在特定文件夹 `IndexMeta/` 中，其中包含一个提供所有 Index 信息的 `IndexMeta/indices` 文件。

- `IndexMeta/indices` 存储 索引名 (64 bytes) 、表名 (32 bytes)、属性名 (32 bytes)；

👇是一个例子

```
+--- TableMeta
|        |--- tables
|        |--- book
|        |--- student
|
+--- IndexMeta
         |--- indices
```

3.2.2.3 具体记录存储说明

具体记录文件存储在特定文件夹 `data/` 下，每一张表占据一个文件，存储表中真实的记录。

👇是一个例子

```
+--- data
        |--- book
        |--- student
```

四、API 模块设计报告 

> by 李易非

4.1 初步介绍

API是程序架构的中间层，API接受 Interpreter 解释完成、初步检测语法的命令，调用 CatalogManager 检验一致性、确定命令的具体执行规则；调用 RecordManager 来进行记录文件的更新；调用 IndexManager 进行索引文件的更新；在本次实验中，我们的API 类有如下几个函数。

```
class API
{
  public:
    bool create_table(表名, 属性);
    bool drop_table(表名);
    bool insert(表名, 插入数据, 插入数据类型);
    int Delete(表名, 属性名, 条件, 操作数);
    int select(表名, 属性名, 条件, 操作数);
    bool create_index(表名, 属性名, 索引名);
    bool drop_index(索引名, 表名);
};
```

4.2 具体实现

Create Table

- 调用 CatalogManager 的 Create Table 函数；
- 调用 RecordManager 的 Create Table 函数；
- 遍历 Interpreter 提供的属性，如果是 Primary / Unique，调用 IndexManager 建立索引；

Drop Table

- 调用 CatalogManager 获得所有的索引，通过获得的索引，调用 IndexManager 的 Drop Index 函数；
- 调用 CatalogManager 的 Drop Table 函数；
- 调用 RecordManager 的 Drop Table 函数；

Insert

- 调用 CatalogManager 获得表信息，通过表信息来判断插入数据的格式是否合法，同时把插入数据 (string) 为对应类型的 rawdata (char 数组) ；
- 判断各个属性的性质
    - 若为 Unique 且 Indexed —> 调用 IndexManager 的 Find 函数进行一致性检验；
    - 若为 Unique 但 UnIndexed —> 遍历数据文件进行一致性检验；

    - 若非 Unique —> 直接调用 RecordManager 进行插入；
- 一致性检验完毕后，分别调用 RecordManger / IndexManager 进行数据文件 / 索引文件的更新；

Delete

- 调用 CatalogManager 获得表信息，通过表信息来属性名是否存在，表是否存在；
- 对 `=` 条件检测是否存在索引，如果存在索引，调用 IndexManager 查找到记录位置，再调用 RecordManager 删除数据文件中的记录；
- 对于其它条件，直接利用 RecordManager 在文件中进行遍历搜索来删除记录；

Select

- 调用 CatalogManager 获得表信息，通过表信息来属性名是否存在，表是否存在；
- 对 `=` 条件检测是否存在索引，如果存在索引，调用 IndexManager 查找到记录位置，再调用 RecordManager 找到数据文件中的记录，在通过 CatalogManager 检验找到的记录是否符合所有的条件；
- 对于其它条件，直接利用 RecordManger 来查找记录，同时调用 CatalogManager 来检验找到的记录是否符合所有的条件；
- 找到符合条件的记录后，即时输出；

Create Index

- 调用 Catalog Manager 来判断表名是否存在，同时为索引提供对应属性的数据类型、以及在一条记录中的相对位置；
- 调用 Catalog Manager 的 Create Index 函数，判断属性是否 Unique，属性是否存在，更新 Metadata;
- 调用 Index Manager 的 Create Index 函数，建立一个新的 B+ 树；

Drop Index

- 调用 Catalog Manager 的 Drop Index 函数，判断是否存在这个索引，更新 Metadata;
- 调用 Index Manager 的 Drop Index 函数，删除对应索引文件；

4.3 技术细节

错误管理

本次工程中，共建 `Error` 类来进行错误管理，所有底层函数都利用 **throw** 机制抛出错误，

在 API 层集中进行 **try catch** 。以 Drop Index 为例，API 要做的事很简单，调用对应的函数并处理错误即可。

```
bool API::drop_index(const string &index_name, const string &table_name)
{
    try
    {
        CatalogManager &catalogmanager = MiniSQL::get_catalog_manager();
        IndexManager &indexmanager = MiniSQL::get_index_manager();
```

```
        catalogmanager.drop_index(index_name, table_name);
        indexmanager.dropIndex(index_name);
        return true;
    }
    catch (Error err)
    {
        err.print_error();
        return false;
    }

    return true;
}
```

五、File Manager 设计报告 👹

> by 李易非

5.1 设计简介

由于 Record Manager 与 Catalog Manager 都要对数据文件进行操作，我们在 Record Manager / Catalog Manager 与 Buffer Manager 之间加入 File Manager，理由如下：

- 减少各个 Manager 交互；
- 逻辑意义上更加清晰，Record Manager 与 Catalog Manager 不必亲自提供 / 调用各个文件的块位置，逻辑上只对对应 " 文件 " 进行操作。

  File Manager 具有的功能如下 (简化版) ：

```
class FileManager
{
    private:
    string file_name;
    int record_length;          // 记录长度
    int first_free_record_addr; // free list 表头
    int record_count;           // 记录总数
    int pointer;                // 文件中的"记录指针"

    public:

    FileManager(文件名);

    const char* get_record(记录地址); // 通过地址获得记录内容

    const int add_record(记录的rawdata); // 添加一条记录

    bool delete_record_ByAddr(记录地址); // 通过地址删除记录
```

```
    int getNextRecord(char * rawdata);  // 获得下一条record


};
```

5.2 具体说明

5.2.1 成员变量说明

- `file_name`: file_name 为该 FileManager 正在接管的文件；
- `Record_length`: record_length 为 FileManager 管理的文件的记录长度；
- `First_free_record_addr` 为 free_list 链表头；
- `Record_count` 为文件中记录总数；
- `pointer` 为文件内部指针，指向一条记录；

5.2.2 成员函数说明

- `FileManager(文件名)`
  - 通过文件名打开一个文件，读取第一个 Block 的 Metadata信息，存入成员变量中；
  - Pointer 指向第一条记录的 “ 前一条记录 ”，因为 `getNextRecord` 的函数需要我们不断获得下一条记录，所以 Pointer 的初始值是第一条记录的 “ 前一条记录 ” 。
- `const char* get_record(记录地址);`
  - 先判断记录地址是否大于 **EOF** ，若超出文件范围，弹出错误；
  - 通过记录地址与记录长度，判断记录存在的块，以及记录在块中的相对位置；
  - 调用 Buffer Manager 获得记录内容并返回。

  ```
  if 记录地址 > EOF
      throw err

  Block ID = 记录地址 / 4096;
  相对位置 = 记录地址 - Block ID * 4096;

  Block = BufferManager.GetBlock( Block ID );
  return Block.( content + 相对位置 )
  ```

- `const int add_record(记录的rawdata);`
  - 若当前 free_list 为空，则插入文件末尾；
  - 若当前 free_list 不为空，则插入 free_list 所指向的地址，并更新 free_list
  - 插入完毕后，更新文件 Meta Data (第一个 Block)

```
if first_free_record_addr = -1
    Add to file EOF;
else
    Add by free_list;

Update Meta Data;
```

- `bool delete_record_ByAddr(记录地址);`
  - 先判断记录地址是否大于 **EOF** ，若超出文件范围，弹出错误；
  - 通过记录地址与记录长度，判断记录存在的块，以及记录在块中的相对位置；
  - 调用 Buffer Manager 获得记录内容，重置记录内容的前四位，将其指向 free_list_head 指向的地址，free_list_head 指向本次调用所需删除的记录；
  - Block 标记 dirty;
  - 更新文件 Meta Data (第一个 Block)

```
if 记录地址 > EOF
    throw err

Block ID = 记录地址 / 4096;
相对位置 = 记录地址 - Block ID * 4096;

Block = BufferManager.GetBlock( Block ID );

Block.Content 前四位 = fisrt_free_record_addr;
first_free_record_addr = 记录地址;
Block.setDirty();

Update Meta Data;
return true;
```

- `int getNextRecord(char * rawdata)`
  - 首先将 `pointer` 自增，判断是否 **EOF** ，若是，则返回 -1；
  - 判断 `pointer` 所指的记录是否是有效记录 (判断句尾有效位)；
  - 一直进行上述过程，直到找到一个有效记录，将有效记录赋给 `rawdata`；

```
PointerIncrement();

while(记录无效)
    PointerIncrement();
    if(Pointer > EOF)
        return -1;

rawdata = Pointer.Content;
return Pointer;
```

六、Buffer Manager 设计报告

> by 李易非

6.1 设计简介

Buffer Manager 是数据库与数据文件进行交互的中间层，所有数据库调用 / 更新文件的行为，都必须通过 BufferManager 来管理。这样可以尽量减少内存与磁盘的速度鸿沟，把一些较为常用的 Block 存入 Buffer 中，为了尽量减少 I / O 操作，应该把一些 Block 存在 Buffer 中，数据库需要访问速度更快；

我们采取了 **时钟算法** 作为块替换策略，时钟算法的效率接近块替换策略 **LRU算法** ，但是实现却简单很多，所以我选择了时钟算法作为Buffer的块替换策略；为了实现时钟算法，我们在Buffer内部实现了一个循环链表，循环链表的 Node 是 BufferNode 类，BufferNode 中含有一个 Block 实体，一个标记位，以及指向下一个 BufferNode 的指针；

BufferNode 具有的功能如下 (简化版)：

```
class BufferNode
{
    Block * block;                  // 实际的Block内容
    bool reference_bit;             // 标记位，来确定时钟算法替换的块
    BufferNode * next;              // 指向下一个BufferNode

    Block * get_block();            // 获得BufferNode中的块
    void set_block(Block * block);  // 重置BufferNode中的块

};
```

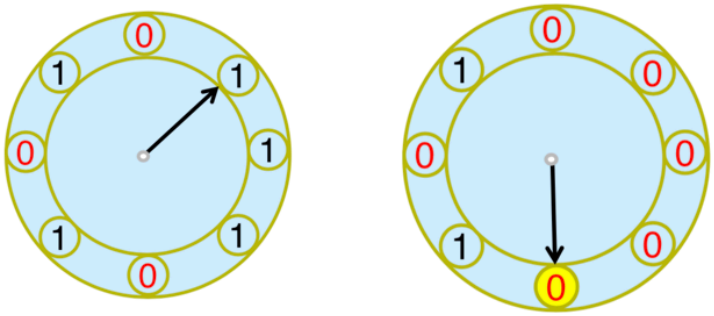
Buffer Manager 具有的功能如下 (简化版)：

```
class BufferManager
{
    const int max_block_count;      // Buffer 容量
    BufferNode * clock_ptr;         // ptr of the 时钟算法


    void WriteBlockBack(Block * block);                 // 写回一个 Block
    Block* getBlock(文件名, Block ID);                  // 通过文件名，Block
ID 获得对应块
    void addBlock(Block * block);                       // 在Buffer中加入一个
块
    void DeleteBlockByFile(文件名);                     // 删除所有与文件名相关
的块
    void WriteAllBack();                                // 写回Buffer里的所有
块

};
```

6.2 时钟算法介绍



Buffer中的每一个结点都有一个参考位，在需要替换Buffer中的块时，

- 从时钟算法指针开始遍历，若参考位为1，则将该节点参考位置为0，前往下一块；
- 重复上述过程直到找到一个参考位为0的节点，将其替换；

  每当有新块加入时，将其参考位变为1；

6.3 具体说明

6.3.1 成员变量说明

- `max_block_count`：Buffer容量，可以容纳的最大块数量；
- `clock_ptr`：时钟算法指针，用于遍历循环链表；

6.3.2 成员函数说明

- `WriteBlockBack(Block * Block)`
  - 通过 Block 对象附带的信息：文件名，Block ID，计算 Block 应该写入的位置，写入文件对应位置；

```
OpenFile(Block.FileName);
写入的相对地址 = Block ID * 4096;
移动文件指针到相对地址;
写入 Block.Content;
```

- `Block* getBlock(文件名, Block ID)`
  - 通过文件名与Block ID 来得到对应的 Block
    - 若 Block 在 Buffer 中，则直接返回；
    - 若 Block 不再 Buffer 中，则从对应文件中读入；

```
if Block is already in Buffer
    找到Block并返回;
else
    从文件中读取 Block;
    加入 Buffer;
    返回读取的 Block;
```

- `void addBlock(Block * block)`
  - 把 Block 加入时钟算法的循环链表中，进行必要的替换；
  - 时钟算法指针开始遍历，若参考位为1，则将该节点参考位置为0，前往下一块；
  - 重复上述过程直到找到一个参考位为0的节点，将其替换；

```
while clock_pointer.refbit = 0
    clock_pointer = clock_pointer->next;

if clock_pointer.Block is dirty
    Write Block Back;

Replace the found Block by new Block;
```

- `void DeleteBlockByFile(文件名)`
  - 删除与文件名相关的所有 Block;
  - 遍历Buffer，找到相关的Block，删除即可；

```
for i = 1 : BufferSize
    if BufferNode.Block.FileNmae = 文件名
        if Block is dirty
            Write Block Back;
        Delete Block from Buffer;
    BufferNode = BufferNode.next;
```

- `void WriteAllBack()`
  - 把 Buffer 中所有脏块写入文件中；

```
for i = 1 : BufferSize
    if BufferNode.Block.FileNmae = 文件名
        if Block is dirty
            Write Block Back;
    BufferNode = BufferNode.next;
```

七、Catalog Manager 模块设计报告 

> by 李易非

7.1 设计简介

Catalog Manager 管理数据库的 Meta Data，在数据库启动阶段，Catalog Manager 将读取 Meta Data并把它们存入内存当中，这样做的理由如下：

- minisql 应用场景：表/索引的数量不会很多，且表的属性最多只有32个；
- 在程序中我们需要大量访问数据 Meta Data，直接存入内存，可以加快访问速度；

  File Manager 具有的功能如下 (简化版) ：

```
class CatalogManager
{
    unordered_map <string, Table *> Name2Table; // 表名到表的hash
    unordered_map <string, Index *> Name2Index; // 索引名到索引的hash

    CatalogManager();

    bool create_table(表名, 属性集);          // 创建表，更新Meta data
```

```
    bool drop_table(表名);                    // 删除表，更新Meta data
    const Table* get_table(表名);             // 通过表名获得表
    bool create_index(索引名，表名，属性名);   // 创建索引，更新Meta data
    bool drop_index(索引名，表名);            // 删除索引，更新Meta data

};
```

7.2 具体说明

1、成员变量说明

- `Name2Table`：表名字到具体表的 `unordered_map` ，可以通过表名直接找到对应表的实体对象；
- `Name2Index`：索引名字到具体索引的 `unordered_map` ，可以通过索引名直接找到对应索引的实体对象；

2、成员函数说明

- `CatalogManager()`
  - 读取 `TableMeta/` 、`IndexMeta/` 中的元信息，构建 Table，Index对象，并加入 `Name2Table` 中；
- `create_table(表名，属性集)`
  - 首先检查该表是否已经存在，若存在，抛出错误；
  - 保证属性集中没有重复的属性，若有重复属性，抛出错误；
  - 保证属性集没有超过32个属性，若超过32个属性，抛出错误；
  - 通过属性信息，构建一个 Table 实体并加入 `Name2Table` 中；
  - 更新 `TableMeta/tables` 信息；
  - 创建 `TableMeta/表名` 文件，并写好第一个Block (该文件对应的元信息)；

```
if table already exists
    throw err
if attribute_set is duplicated
    throw err
if there are more than 32 attributes
    throw err
Name2Table.insert({表名，表实体});
Update Metadata;
CreateFile("TableMeta/表名");
```

- `drop_table(表名)`
  - 检查要删除的表是否存在，若不存在，抛出错误；
  - 更新 `Name2Table`

- 更新 `TableMeta/tables` 文件
- 删除 `TableMeta/表名` 文件

```
if table does not exist
    throw err;
Name2Table.erase(表名);
Update Metadata;
DeleteFile("TableMeta/表名");
```

- `const Table* get_table(表名)`
  - 首先检查该表是否存在，若不存在，抛出错误
  - 通过 `Name2Table` 直接返回对应表实体；

```
if table does not exist
    throw err;

return Name2Table(表名)
```

- `bool create_index(索引名, 表名, 属性名)`
  - 首先检查该索引是否存在，若已经存在，抛出错误；
  - 再检查需要建索引的属性是否时 Unique，若不是，抛出错误；
  - 更新 `Name2Index`
  - 更新 `IndexMeta/indices`

```
if index already exists
    throw err;
if attribute is not Unique
    throw err;
Name2Index.insert({索引名, 索引实体})
Update Metadata;
```

- `bool drop_index(索引名, 表名)`
  - 首先检查该索引是否存在，若不存在，抛出错误；
  - 更新 `Name2Index`
  - 更新 `IndexMeta/indices`

```
if index does not exist
    throw err;
Name2Index.erase(索引名);
Update Metadata;
```

八、Record Manager 设计报告

> by 李易非

8.1 设计简介

Record Manager 具体管理各个表的记录，依据 API 提供的需求来进行相应数据文件的更新，因为之前 File Manager 的实现，Record Manager 的实现被大大化简，从物理层操作上升到了逻辑层操作。

Record Manager 具有的功能如下 (简化版)：

```
class RecordManager
{
    bool create_table(表名);
    bool drop_table(表名);
    int insert(表名, rawdata);
    int select(表名, 属性名, 条件, 操作数);
    int Delete(表名, 属性名, 条件, 操作数);
    const char * GetRecordByAddr(表名, 地址);
    bool DeleteRecordByAddr(表名, 地址);
};
```

8.2 具体实现

- `bool create_table(表名)`
    - 调用 Catalog Manager 获得对应表名的 Record Length;
    - 创建数据区文件 `data/表名`

```
get Record Length from Catalog Manager;
CreateFile("data/表名", 记录长度)
```

- `bool drop_table(表名)`
    - 删除数据区文件 `data/表名`

- `int insert(表名, rawdata)`
  - 通过表名构建一个 File Manager, 调用 File Manager 的 `add_record(rawdata)` 函数

```
FileManager file(表名);
file.add_record(rawdata);
```

- `int select(表名, 属性名, 条件, 操作数)`
  - 调用 Catalog Manager 通过表名获得表信息；
  - 遍历文件，判断每一条记录是否符合条件；若符合，打印对应记录；
  - 返回符合条件的记录总数；

```
Table = CatalogManager.get_table(表名);
FileManager file("data/" + 表名);

int NumOfRecord = 0;
while file.getNextRecord != -1
    if Table.isSatisfyAllCondition(记录rawdata, 属性名, 条件, 操作数);
        print the record;
        NumOfRecord++;

return NumOfRecord;
```

- `int Delete(表名, 属性名, 条件, 操作数)`
  - 调用 Catalog Manager 通过表名获得表信息；
  - 遍历文件，判断每一条记录是否符合条件；若符合，删除记录与对应的索引；
  - 返回删除的记录总数；

```
Table = CatalogManager.get_table(表名);
Attris = Table.GetAttriIndexed(); // 获得所有有索引的属性
FileManager file("data/" + 表名);

int NumOfRecord = 0;

while file.getNextRecord != -1
    if Table.isSatisfyAllCondition(记录rawdata, 属性名, 条件, 操作数);
        file.DeleteRecord;
        for all attributes that is indexed
            delete the index data;
        NumOfRecord++;

return NumOfRecord;
```

- `const char * GetRecordByAddr(表名, 地址)`
  - 构建 File Manager, 调用 File Manager 的 GetRecordByAddr 函数，并返回记录的 rawdata;

```
FileManager file("data/" + 表名);
return file.GetRecordByAddr(地址);
```

- `bool DeleteRecordByAddr(表名, 地址)`
  - 构建 File Manager, 调用 File Manager 的 DeleteRecordByAddr 函数;

```
FileManager file("data/" + 表名);
return file.DeleteRecordByAddr(地址);
```