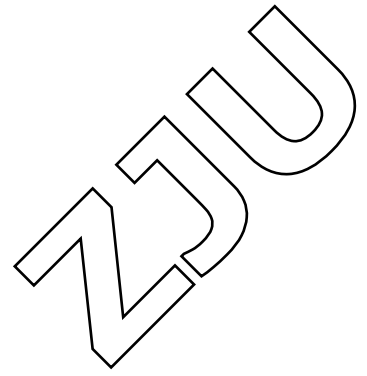


MiniSQL 详细设计报告



参与人员：管铮 B+ Tree, API

张海燕 Buffer

张一品 Interpreter, Catalog

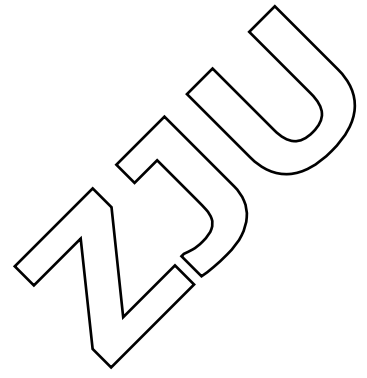
邬雪松 Catalog

许灵 Record, Buffer

时间：2007 年 6 月

指导老师：孙建伶

目录



1. [引言](#)

2. [系统结构](#)

2.1 [结构图](#)

2.2 [Interpreter](#)

2.3 [API](#)

2.4 [Catalog Manager](#)

2.5 [Index Manager](#)

2.6 [Buffer Manager](#)

2.7 [Record Manager](#)

3. [文件清单](#)

1 引言:

1.1 项目名称:

小型关系数据库管理系统的设计与实现。

1.2 编写目的:

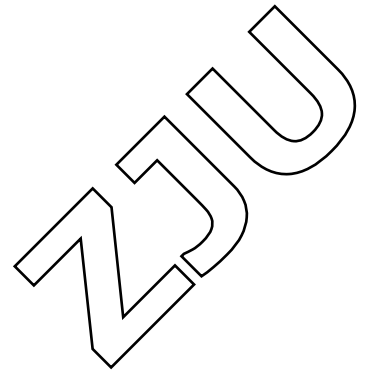
在文件系统之上设计并开发一个小型的关系数据库管理系统,深入了解模式定义、查询处理、存储管理、索引管理等相关实现技术,通过实验加深对 DBMS 及其内部实现技术的理解。实践系统软件开发的工程化方法。

1.3 项目背景:

数据库系统设计与实现实验。

1.4 参考资料:

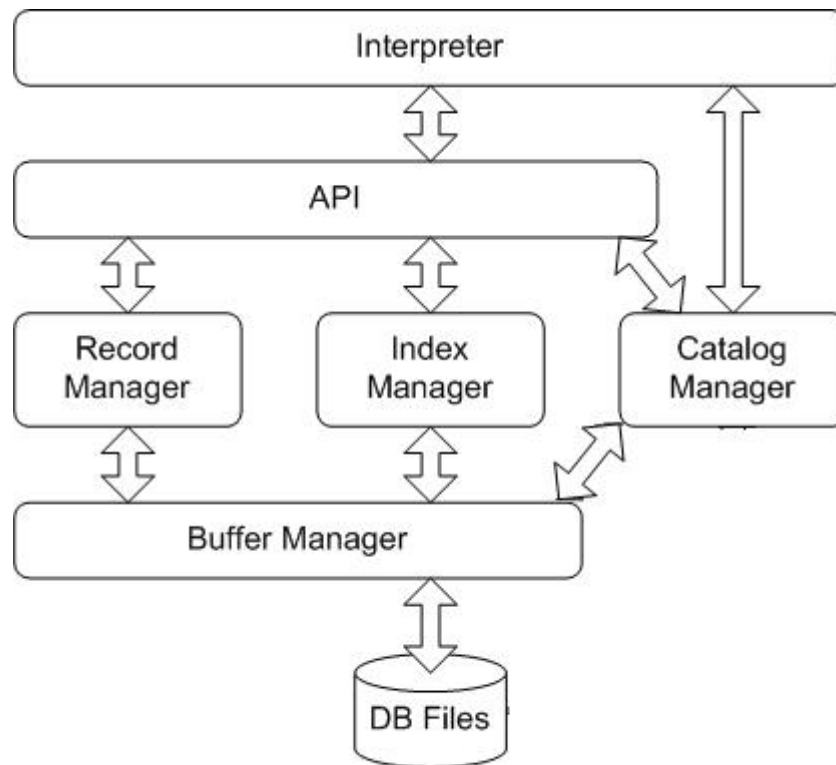
Database system concepts .



2 系统结构:

2.1 系统结构图:

2.1.1 结构图:



各个部分的大致结构关系

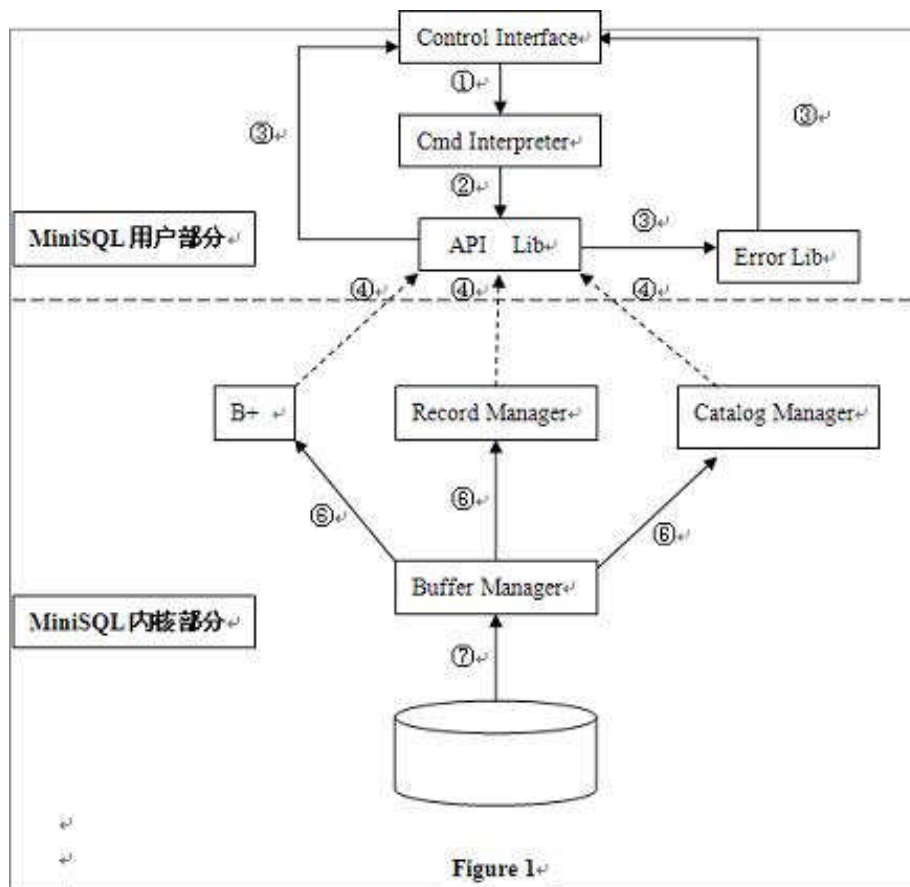


Figure 1

各个部分的详细模式分类及关系

- ①判断并接受用户字符输入，使做为解释器的输入。
- ②解释器对用户输入进行翻译，产生操作数（所需 API 编码以及参数数组）。
- ③执行选定的 API，返回用户所需的输出。
- ④BPlus、Record、Catalog 类方法注册至 API LIB,并整合生成适合于用户调用的 API。
- ⑤BPlus、Record、Catalog 类调用 Buffer 类的方法实现自己各自的方法。
- ⑥BPlus、Record、Catalog 类调用 Buffer 类的方法实现自己各自的方法。
- ⑦Buffer 类方法对数据库文件进行直接操作。

2.1.2 各模块简要说明:

1. **Interpreter** 负责解释命令，并输出相关错误信息。
2. **API** 负责各模块之间的接合及相互间的数据传递和命令执行，并帮助 **Record Manager** 实现了不等值查找和区间查找。
3. **Record Manager** 负责实现数据文件的创建与删除（由表的定义与删除引起）、记录的插入、删除与查找操作，并对外提供相应的接口，支持等值查找。
4. **Index Manager** 负责 B+树索引的实现，实现 B+树的创建和删除（由索引的定义与删除引起）、等值查找、插入键值、删除键值等操作，并对外提供相应的接口。
5. **Catalog Manager** 负责管理数据库的所有模式信息，并提供访问及操作上述信息的接口，供 **Interpreter** 和 **API** 模块使用。
6. **Buffer Manager** 负责缓冲区的管理。
7. **DB Files** 指构成数据库的所有数据文件，主要由记录数据文件、索引数据文件和 **Catalog** 数据文件组成。

2.2 ① Interpreter:

2.2.1 概述: 所有操作由一个类来完成，将得到的信息封装再传给其他模块。每执行一

条命令都由一个对象完成读取解析，传给其他模块执行，并将有返回结果的输出。

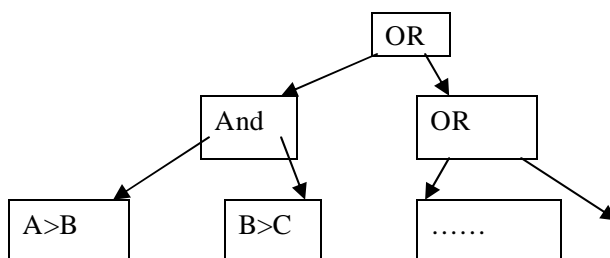
功能: 将用户的输入命令进行语法分析和语义解析并得到需要的命令参数,最后将该

命令参数封装成对应命令的参数类对象,传给 **API** 模块,供其他模块读取使用,或在操作错误是给出提示。

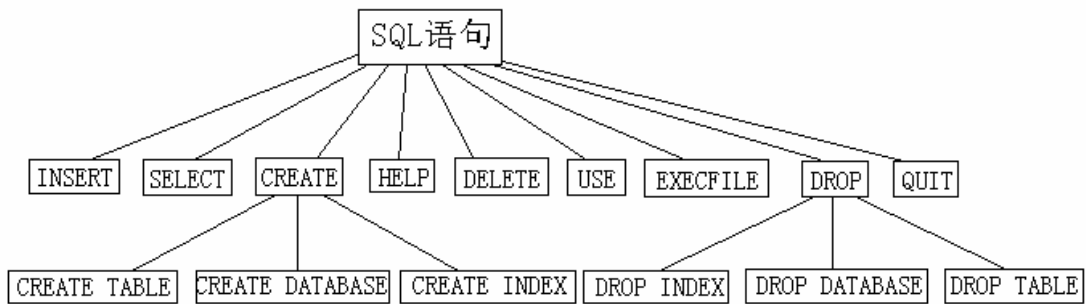
2.2.1.1 解析命令过程:

将命令读入，并把括号，标点去掉，保留有用的信息。如: `create table student(id int, name char(10));`该语句读入后变为:`create table student id int name char`。然后解析该命令,读到 `create table` 时，判断出命令类型，交由相关函数处理，提取语句中与 `table` 有关的字符串。交给 **API** 调用相关函数执行操作。

如果操作是包含条件的语句，将条件放入条件树中，大致结构如下：



其语句结构如下：



其具体实现方式是将各语句逐次进行分类，再根据各语句进行分别处理和实现：

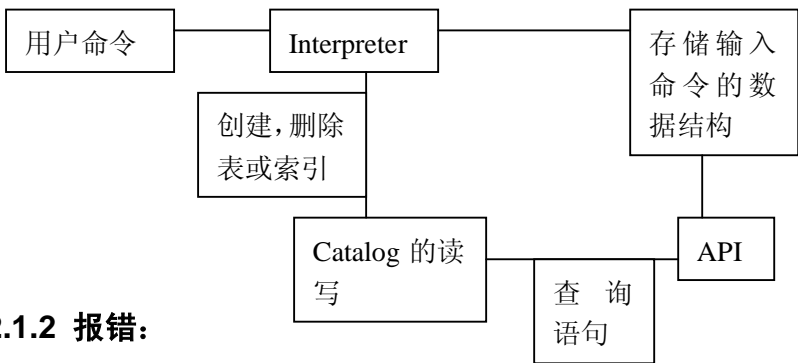
第一步：根据语句中的第一个关键字进行分类，可分为：1.CREATE； 2.DROP；
3.SELECT； 4.DELETE； 5.INSERT； 6.USE； 7.EXECFILE；
8.HELP； 9.QUIT。

第二步：可将 CREATE 类型进行进一步的分类，可分为：1.CREATE DATABASE；
2.CREATE TABLE； 3.CREATE INDEX。相似地，可将 DROP 类型分类如下：

1.DROP DATABASE； 2.DROP TABLE； 3.DROP INDEX。

第三步：对各语句进行分别处理，封装后返回相应的数据结构并交给 API 进行相关判断操作。

大致结构如下：



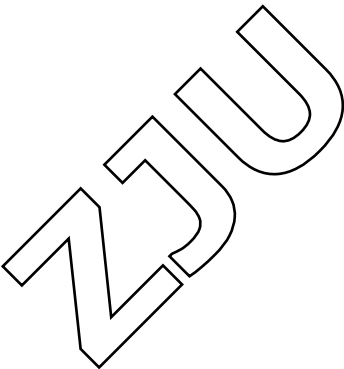
2.2.1.2 报错：

对于在解析时命令格式的错误，给出尽可能详细的错误报告。由于不是语言编译器所以，可能会出现一些无法指出的错误，这时，做到不执行命令，报语言格式出错。

2.2.2 输入输出：

输入：sql 命令；

输出：若 sql 命令出错，则输出出错信息，否则将数据传给 API 执行命令。



2.2.3 数据结构:

2.2.3.1 表的数据结构:

名称	标识符	类型	使用方式	访问方式	描述
表名	Name	String			记录表的名字
表里的属性	attributes[32]	Attribute (自定义)			记录存储的属性
主键	primarykeys[32]	String			记录表的主键
插入值	value[32]	String			插入操作时记录表的插入值

2.2.3.2 属性的数据结构:

属性的数据结构: 名称	标识符	类型	使用方式	访问方式	描述
属性名	Name	String			存储属性的名字
属性类型	Attributetype	Int			存储属性的类型
是否unique	Unique	Bool			该属性是否unique
如果是char记下char的个数	Charnum	Int			记录char的个数
索引名	Idxname	String			记录index的名字

2.2.3.3条件树的数据结构:

名称	标识符	类型	使用方式	访问方式	描述
左子树	Left	condition_tree			左子树
右子树	Right	condition_tree			右子树

运算符判断	is_op	Bool			运算符判断 判断是运算符还是and, or等符号
运算类型	Optype	Int			运算符类型
运算名	Opname	String			运算的名字
左算子	Leftone	String			左边的算子
右算子	Rightone	String			右边的算子
结束节点判断	End	Bool			判断是否是叶节点

2.2.3.4 Interpreter 类的主要结构:

主要变量:

判断操作类型:

```
bool is_create_table;
bool is_create_index;
bool is_select;
bool is_insert;
bool is_drop_table;
bool is_drop_index;
bool is_delete;
bool is_quit;
bool is_execfiles;
```

存储输入命令:

```
vector<string> command;//store the command stream in this
```

相关存储结构:

```
//this is store the selected items of the table
vector<string> selected_items;
//this is store the names of the tables where selected items come
vector<string> selected_tables;
//this store the condition tree when insert or select
condition_tree *condition;
//if create table , store the information of table in this
table T;
```

主要函数及功能说明:

```
void getcommand();//get command
```

```
bool parsingcommand();//
```

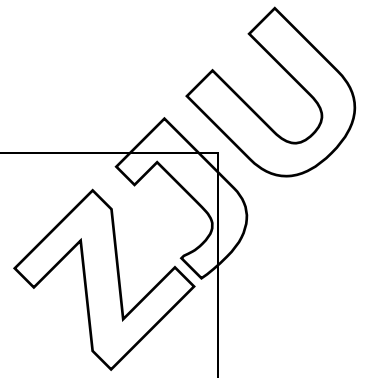
```
void getcondition
```

```
(vector<string> command,condition_tree *condition);//construct condition tree
```

```
void if_create_table();//do create tabel stuff
```

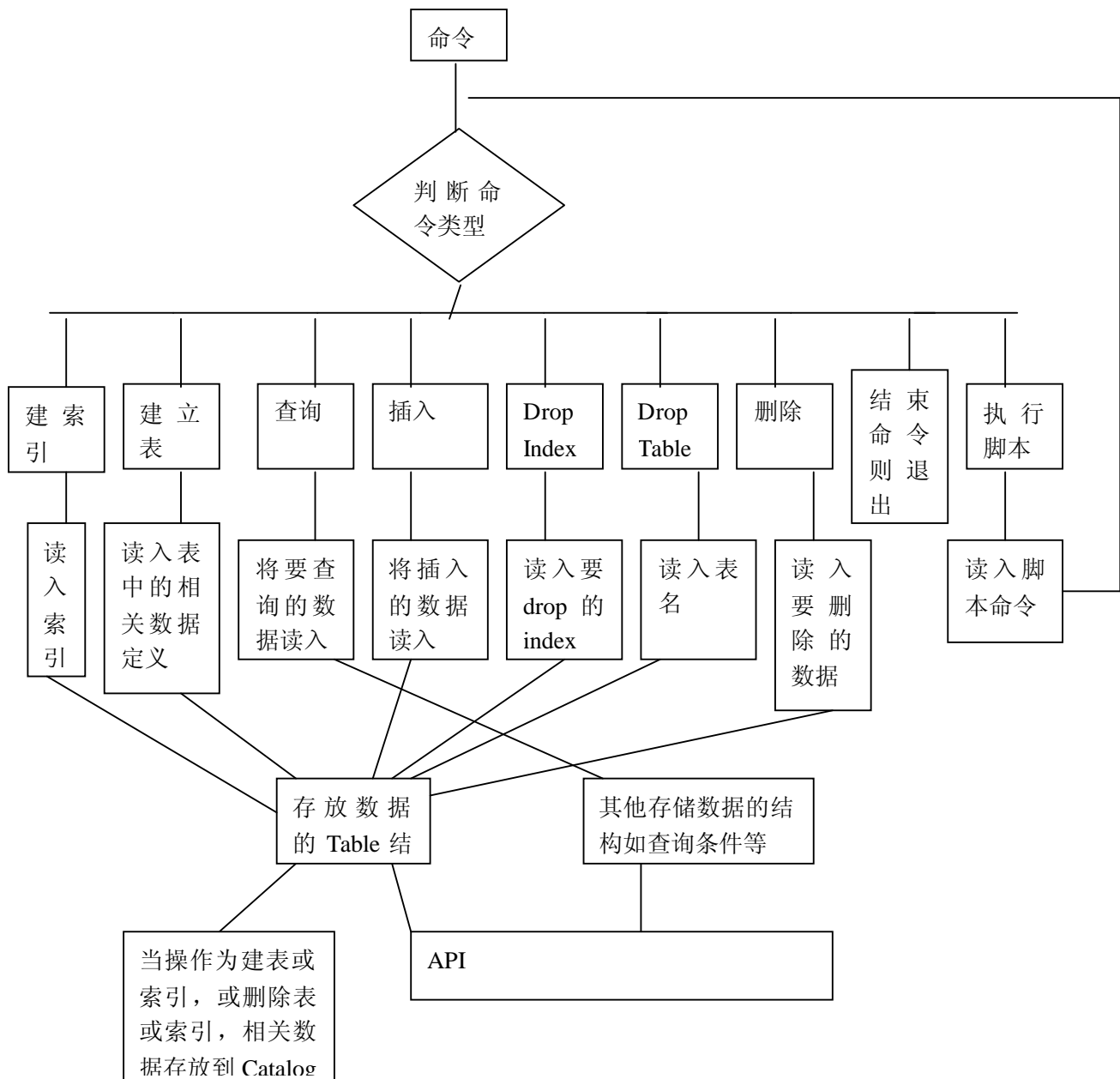
```
void if_create_index();//do create index stuff
```

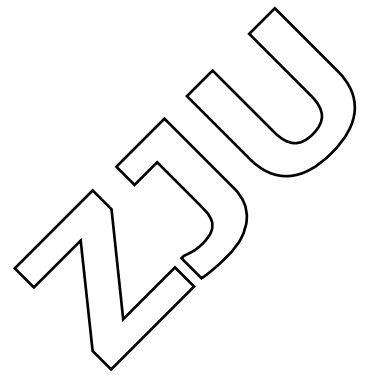
```
void if_select();//do select stuff
```



```
void if_insert();//do insert stuff
void if_drop_table();//do drop table suff
void if_drop_index();//.....
void if_delete();
void if_quit();
void if_execfiles();
```

2.2.4 主要算法:





报错操作在流程中会跳出。

2.2.5 接口说明:

主要有表内容的传递（定义在表的结构体中），其他涉及表相关操作的也有此结构体传递，并有 API 读取或写入，详细说明见[数据结构](#)说明：

传递给其他模块的关于表的结构：

```
struct table
{
    string name;
    struct attribute{
        string name;
        int attributetype;
        bool unique;
        int charnum;
        bool isidx;
        string idxname;
    }attributes[32];
    string primarykeys[32];
    string value[32]; //for insert
};
```

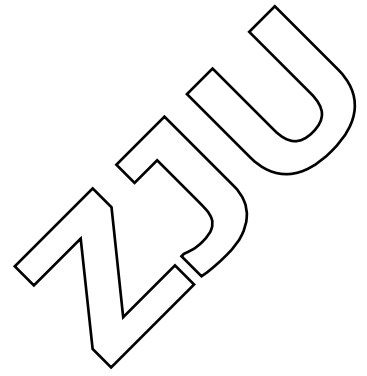
一些 select 或 delete 操作的条件内容由结构体传递：

```
struct condition_tree //this tree is store the conditions by select or insert command
{
    condition_tree *left;
    condition_tree *right;
    bool is_op;
    int optype;
    string opname;
    string leftone;
    string rightone;
    bool end;
};
```

2.2.6 测试要点:

主要测试命令解析和报错功能，查看解析后的命令是否正确。输入错误命令，是否能报

错或退出。

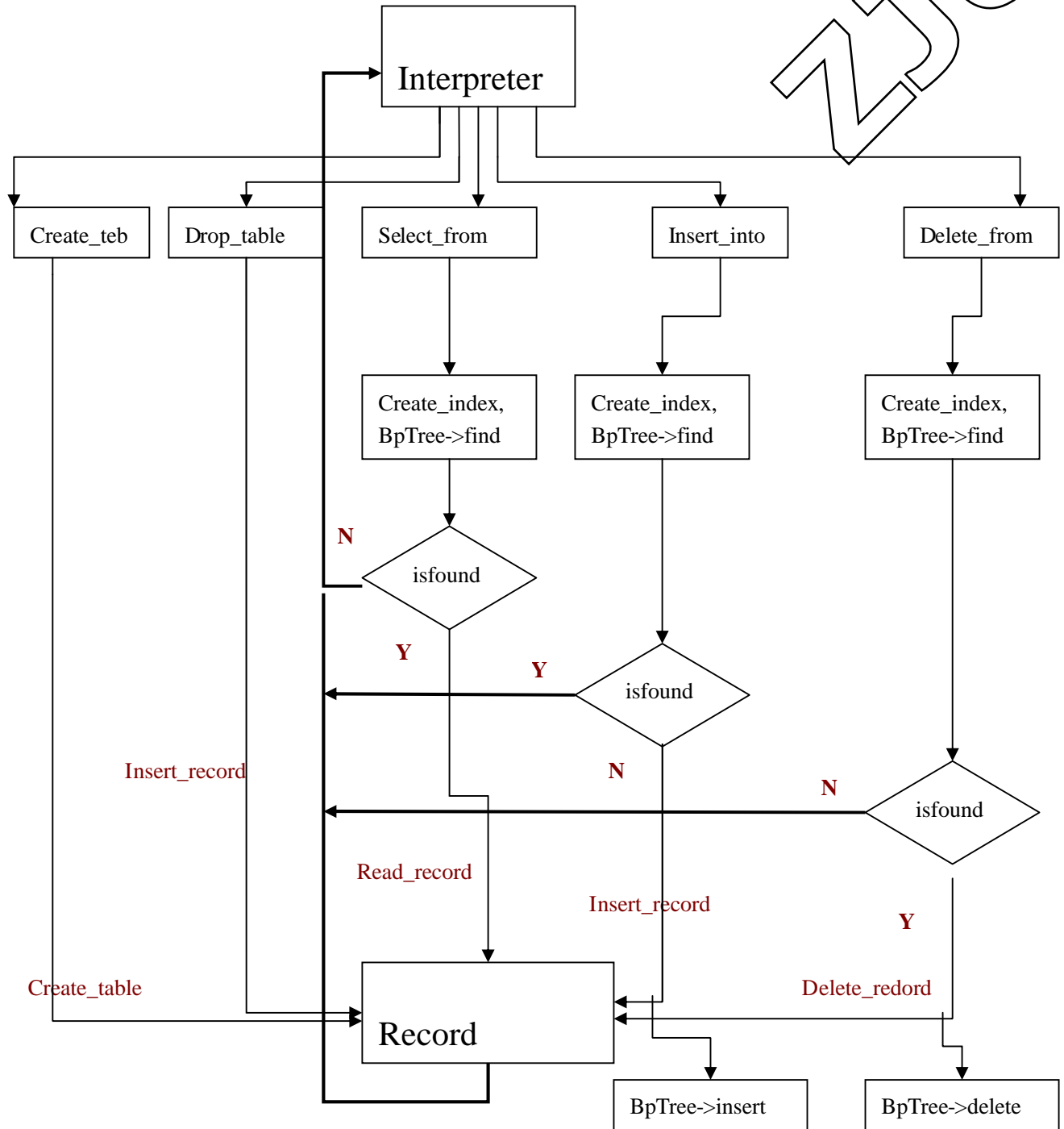
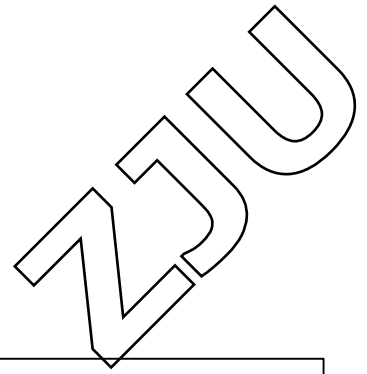


2.3 API:

2.3.1 概述: API 模块是整个系统的核心，其主要功能为提供执行 SQL 语句的接口，供 Interpreter 层调用。该接口以 Interpreter 层解释生成的命令内部表示为输入，根据 Catalog Manager 提供的信息确定执行规则，并调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行，最后返回执行结果给 Interpreter 模块。

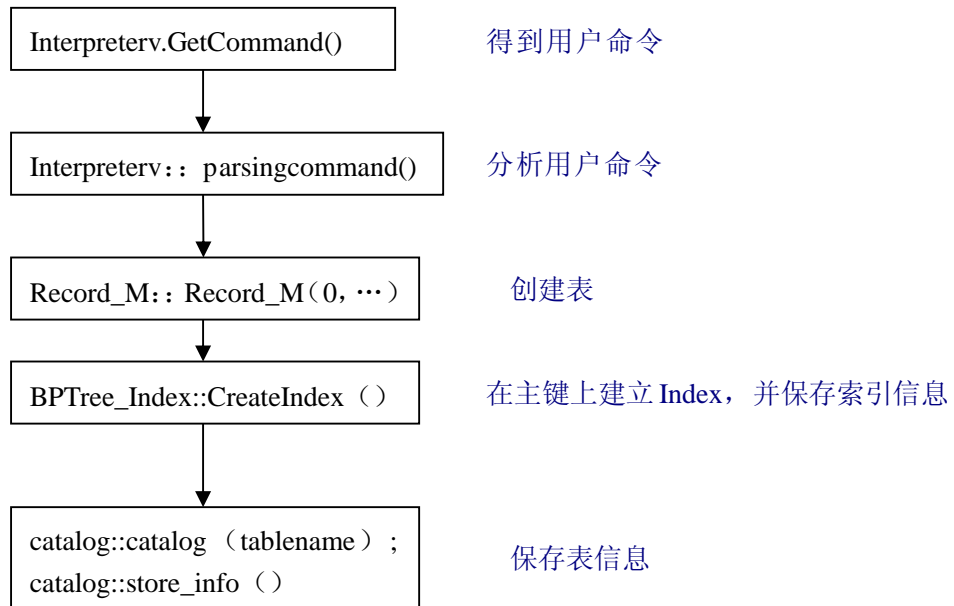
2.3.2 结构图:

注：图中红色的都是 Record Manager 模块中的函数。

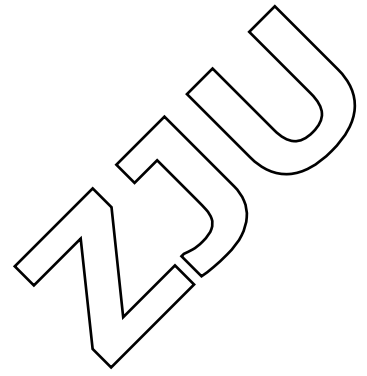
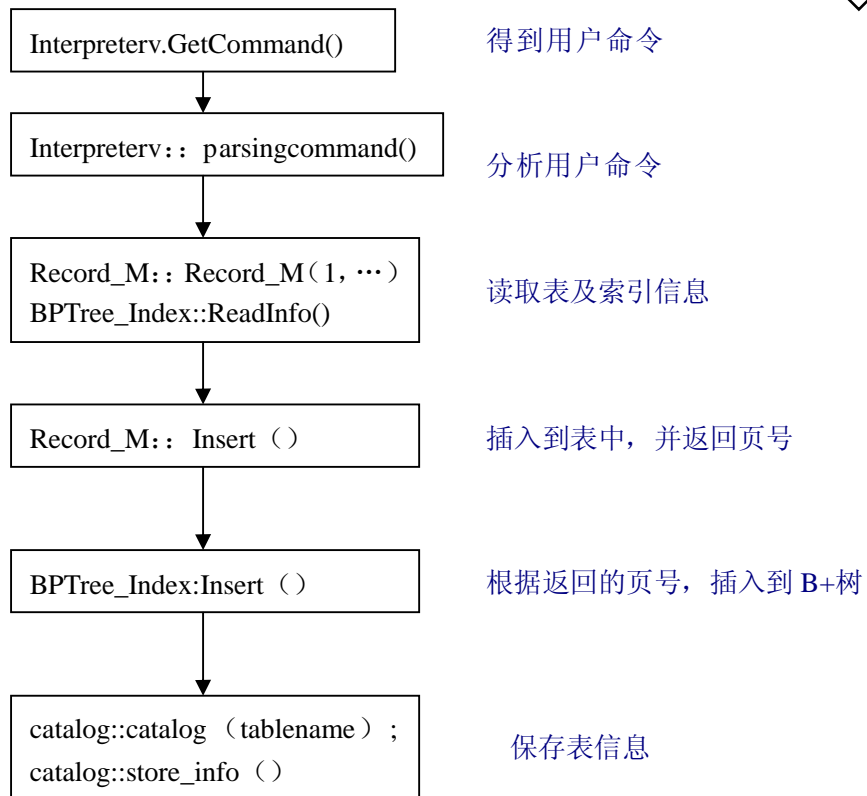


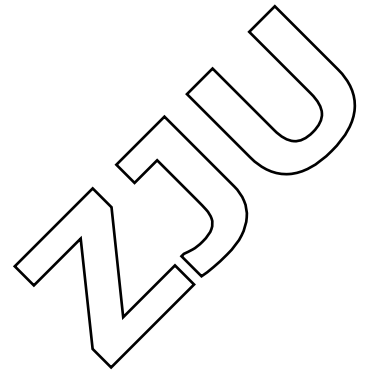
2.3.3 API 模块:

2.3.3.1 CreateTable:

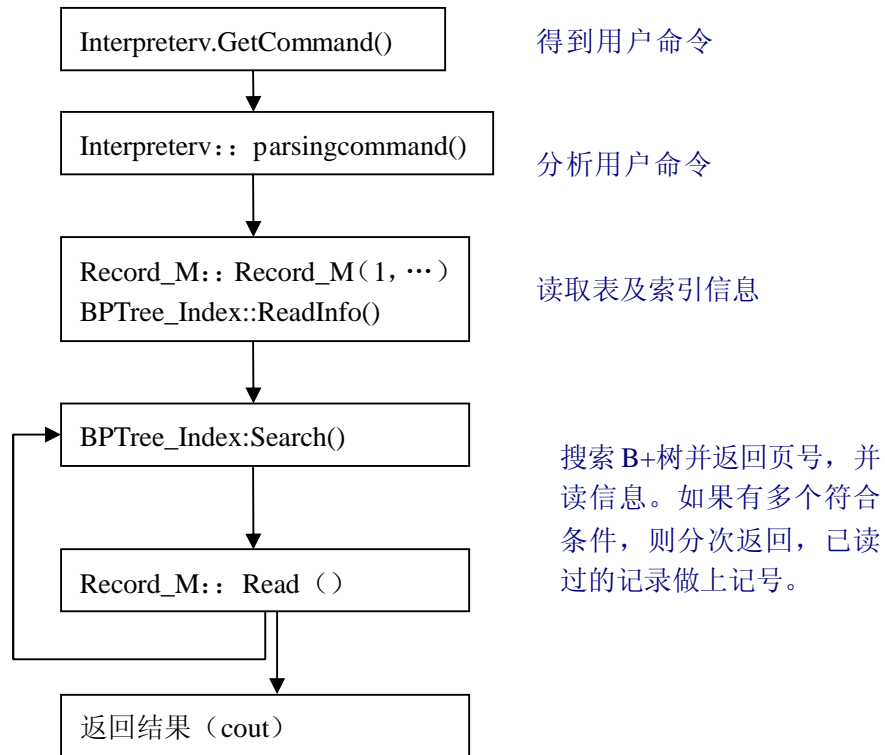


2.3.3.2 Insert:

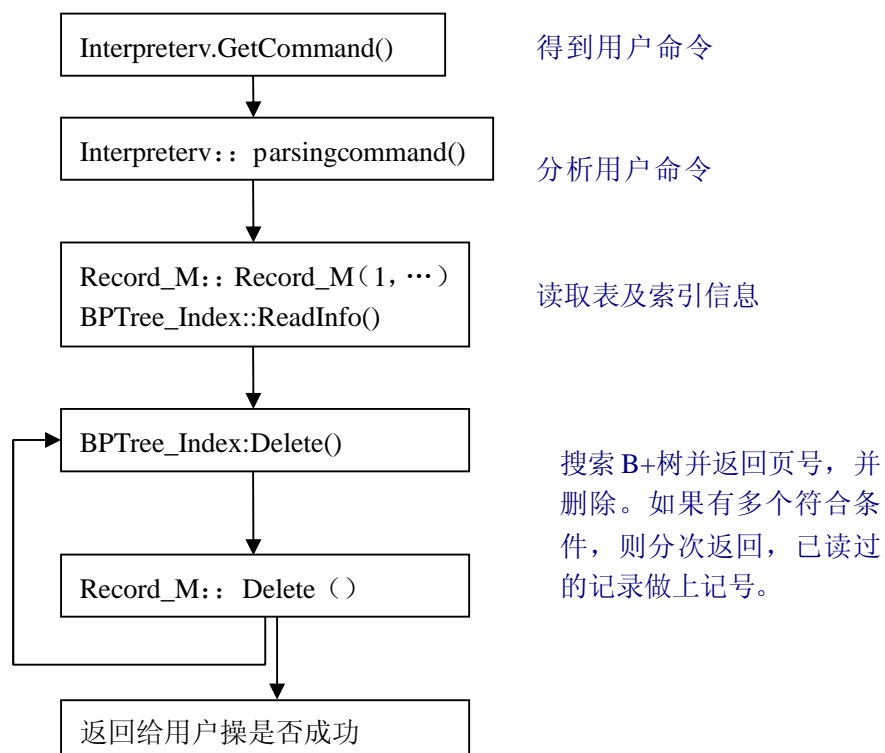




2.3.3.3 Select:



2.3.3.4 Delete:



2.3.3.5 实现:

2.3.3.5.1 在 Insert 时, 在主键上 Create_index, 再执行 Find, 如果存在则给用户提示—‘已存在’, 否则返回值是要插入的位置 (这里的 B+树中的 Find 在查找时如没找到, 则返回最后指针所在的叶结点, 这也即该 Tuple 要插入的地方), 插入记录。

2.3.3.5.2 在进行区间查找时, 比如 $a < \text{table.x} < b$, 执行 find(a), find(b) 然后只在取中间的叶结点就是所要查询结果。

2.3.3.5.3 在执行完每个句时都会把结果放在一个 struct 中返回给 Interpreter。

```
Struct retnbuff
{
    Bool issucc;      是否成功
    String info;      返回给用户的信息
    Bool tuplenum;    返回记录数
    Struct *tuple;    记录集
}
```

2.4 Catalog Manager:

2.4.1 主要功能:

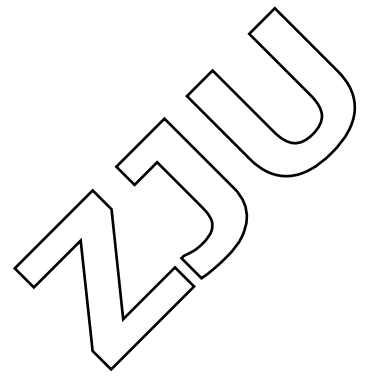
以下信息存放到文件中:

2.4.1.1 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。

2.4.1.2 表中每个字段的定义信息, 包括字段类型、是否唯一等。

2.4.1.3 数据库中所有索引的定义, 包括所属表、索引建立在那个字段上等。

Catalog Manager 还必需提供访问及操作上述信息的接口, 供 Interpreter 和 API 模块使用。



2.4.2 输入输出:

输入: 创建或删除表, 索引时的信息, 要读取的信息。

输出: 相关信息写入文件, 当其他模块需要信息时, 输出需要的信息。

2.4.3 数据结构:

(一) 异常代号

```
int ErrorCode;
char* ErrorMessage[]; //相关异常信息
typedef struct Column_Info *pColumn_Info; //字段指针类型
typedef struct Constraint_Info *pConstraint_Info; //约束条件指针类型
```

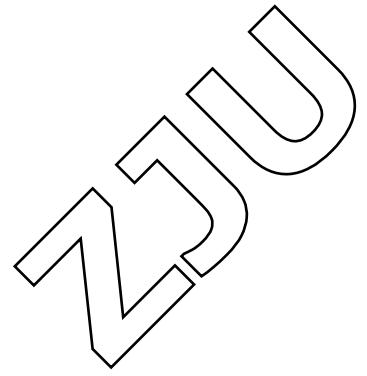
(二) 表信息

```
typedef struct
{
    char        TableName[32]; //表名 (限定最长为 32 位)
    int         TotalColumn;   //总字段数
    int         TotalLength;   //每个记录长度
    int         KeyAttrNum;    //多属性主键中属性个数
    union{
        pColumn_Info    PrimaryKey; //指向主键的指针
        FilePtr         filePrimaryKey;
    };
} Table_Info;
```

字段信息

```
typedef struct
{
    char        ColumnName[32]; //字段名 (限定最长为 32 位)
    int         IsPrimary;      //是否主键的标志: 1—是; 0—否
    Column_Type ColType;        //字段类型—int, char, float
    int         ColLength;      //字段长度
    union{
        pColumn_Info    next; //下一个字段指针
        FilePtr         filenext;
    };
    union{
        Constraint_Info constraint; //指向此字段上的约束条件指针
        FilePtr         fileconstraint;
    };
} Column_Info;
```

(三) Catalog 类定义:

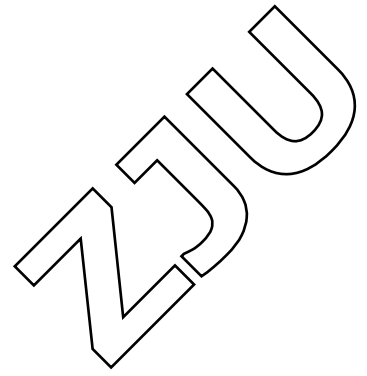


```
class Catalog
{
public:
    Catalog();
    bool Table_Exist(char*);                //检查表是否已存在
    pColumn_Info Find_Column(char* )       //查找并定位字段信息
    bool Column_Exist(char*);              //检查字段是否已存在
    //检查类型是否匹配
    bool Type_Check(pColumn_Info,Column_Type);
    //检查是否满足约束条件
    bool Constraint_Check(pColumn_Info ,pValue_Info);
    void Create(TB_Create_Info&,int); //表的创建 (int 为 index 所要的主键属性个数)
    void Select(TB_Select_Info&,        //查询 (Interpreter 给出 TB_Create_Info&,
                Condition_Info&,        // Condition_Info&给 Index
                Select_Rec_Info&);       // Select_Rec_Info&给 Record )

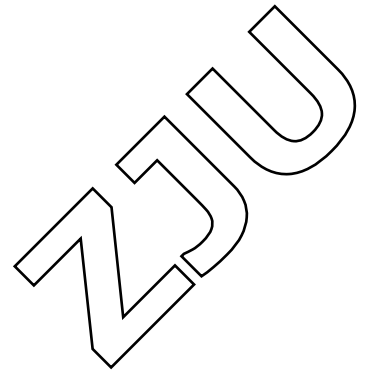
    void Insert(TB_Insert_Info&,        //插入 (Interpreter 给出 TB_Insert_Info&,
                pKey_Attr,              // pKey_Attr 给 Index,
                Rec_Info&);             // Record_Info&给 Record )
    void Update(TB_Update_Info&,        //更新 (Interpreter 给出 TB_Update_Info&,
                Condition_Info&,        // Condition_Info&给 Index,
                Rec_Info&);             // Rec_Info&给 Record )
    void Delete_check(TB_Delete_Info&   //删除 (Interpreter 给出 TB_Delete_Info&,
                Condition_Info&);       // Condition_Info 给 Index)
    ~Catalog();
private:
    typedef Value_Info  *pValue_Info;
    typedef struct
    {
        char          ColumnName[32];
        Limit_Value    value;
        Column_Type     ColType;
    }Value_Info;
};

(四) bool Table_Exist(char* pTableName)
{
    if (能打开&pTableName)
        return True;
    else
        return False;
}

(五) pColumn_Info Find_Column (char* pColumnName)
{
    取出第一个字段信息;
```



```
while (不是最后一个字段信息){
    if (&pColumnName == 字段信息.字段名)
        return 指向此字段的指针;
    else
        取下一个字段信息;
}
return Null;
}
(六) bool Column_Exist(char* pColName)
{
    if ( Find_Column(pColName) == Null)
        return False;
    else
        return True;
};
(七) bool Type_Check(pColumn_Info ptr , Column_Type ColType)
{
    if (ptr->ColType == ColType)
        return True;
    else
        return False;
}
(八) bool Constraint_Check(pColumn_Info ptr , pValue_Info ValueInfo)
{
    if ((ValueInfo->value) 满足 ptr->constraint 中的条件)
        return True;
    else
        return False;
}
(九) void Create(TB_Create_Info&,int)
{
    分析 TB_Create_Info;
    if( Table_Exist() )                //表已存在
        throw error code;
    创建新表;
    Index 初始化信息放入 int 参数传出;
    return;
}
(十) void Select(TB_Select_Info&,Condition_Info&,Select_Col_Info&)
{
    分析参数 TB_Select_Info;
    if (!Table_Exist() )                //表不存在
        throw error code;
```



```
else if (!Column_Exist())           //字段不存在
    throw error code;
else if (不是主键上的查找)
    throw error code;
else if (!Type_Check(Primary Key))  //主键类型不匹配
    throw error code;
else if (where 中查找条件越界)
    throw error code;
生成 Index 查找所需要的信息——Condition_Info;
生成 Record 查找所需要的信息——Select_Col_Info;
return;
}
(十一) void Insert(TB_Insert_Info&,    pKey_Attr, Rec_Info&)
{
    分析参数 TB_Insert_Info;
    //正确性检查
    if (!Table_Exist())               //表不存在
        throw error code;
    else if (!Column_Exist())         //字段不存在
        throw error code;
    else if (!Type_Check(values))     //插入值类型不正确
        throw error code;
    else if (!Constraint_Check())     //插入值不满足约束条件
        throw error code;
    将记录填充完整;
    生成 Index 查找所需要的信息——pKey_Attr;
    生成 Record 查找所需要的信息——Rec_Info;
    return;
}
(十二) void Delete_check(TB_Delete_Info&, Condition_Info&)
{
    分析参数 TB_Delete_Info;
    if (!Table_Exist())               //表不存在
        throw error code;
    else if (!Type_Check(Primary Key)) //主键类型不匹配
        throw error code;
    生成 Index 查找所需要的信息——Condition_Info;
    return;
}
```

2.4.4 Catalog 类的说明:

定义的变量说明:

table_record table_info;//table 的信息

attribute_record attribute_info;//属性的信息

string name;//文件名, 与表名相同

定义的函数说明:

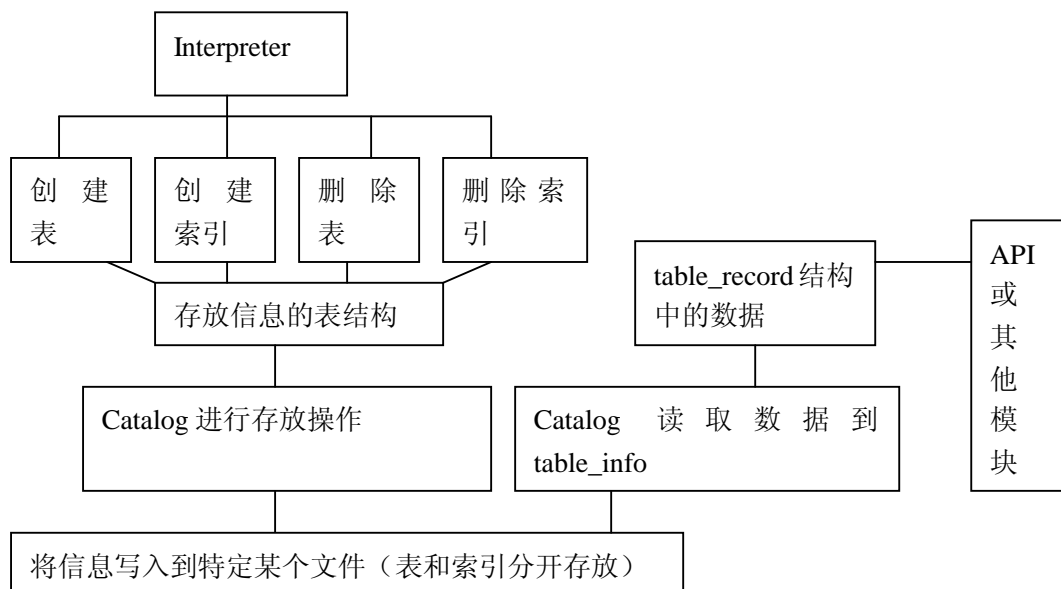
//将 table t 信息写入文件

//type =1 时存放创建表的信息,type=2 时存放创建 index 的信息,type=3 时,drop index

bool store_info(table t, int type);

bool get_info();//把文件里的信息存入 table_info

2.4.5 算法和流程:



2.4.6 接口概述:

主要在 interpreter 创建表时将以上信息存放在文件中, 供 API 进行相关如查询等操作读取。

表定义、字段定义信息和索引定义存放在不同文件中。

完成数据传递的几个结构都已在数据结构中说明。

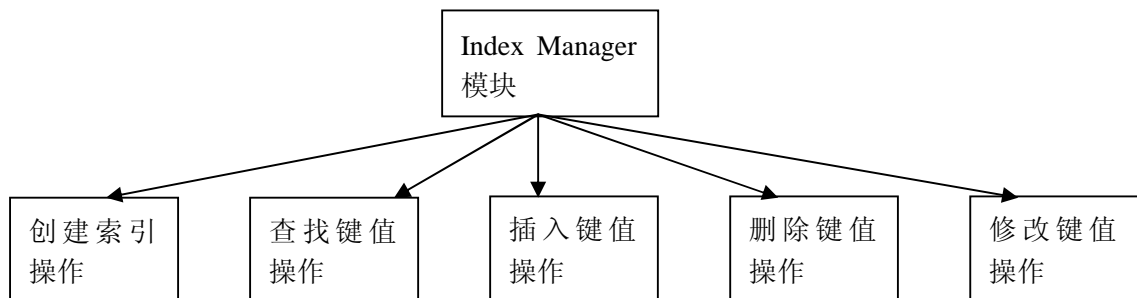
2.4.7 测试要点:

主要测试文件存放及读取信息是否正确。

2.5 Index Manager:

2.5.1 软件结构:

Index Manager 模块结构图:



2.5.2 模块描述:

2.5.2.1 功能:

对于表的主属性自动建立 B+树索引，对于声明为 unique 的属性（char, float, int 类型）可以通过 SQL 语句由用户指定建立/删除 B+树索引。

2.5.2.1.1 B+树的新建:

创建记录文件 TableName 的同时，新建一个名为 TableName 的 B+树索引文件，并保存到硬盘。

2.5.2.1.2 B+树的查找:

对于给定的关键字进行记录搜索。从 B+树根节点往下搜索各节点。如找到该关键字，返回对应的记录地址，否则返回错误信息。

2.5.2.1.3 B+树的插入:

对于要插入的记录，先查找到插入节点，如有空位，插入，否则分裂节点，并调整 B+树。如关键字已存在，则返回错误信息。

2.5.2.1.4 B+树的删除:

对于要删除的记录,先查找到该记录对应关键字,删除并调整 B+树。如找不到记录,则返回错误信息。

2.5.2.1.5 B+树的修改:

对于要修改的记录,先查找到该记录并删除,然后插入修改后的记录。如找不到记录,返回错误信息。

2.5.2.2输入项目:

2.5.2.2.1 创建索引:

关键字信息 (char *KeyInfo): "i"、"f"、"c"+"string size" (如"c50")。

索引名 (char *TableName): 如 "customer"。

2.5.2.2.2 查找键值:

键值 (int iKey、float fKey、char *cKey)。

索引名 (char *TableName)。

2.5.2.2.3 插入键值:

键值 (int iKey、float fKey、char *cKey)。

记录号 (long RecordID)。

索引名 (char *TableName)。

2.5.2.2.4 删除键值:

键值 (int iKey、float fKey、char *cKey)。

索引名 (char *TableName)。

2.5.2.2.5 修改键值:

修改前键值 (int iKey、float fKey、char *cKey)。

修改后键值 (int iKey_、float fKey_、char *cKey_)。

索引名 (char *TableName)。

2.5.2.3 输出项目:

查找键值: 记录号 (long RecordID)

2.5.2.4 实现:

1. 1) 通过 API 调用 CATALOG 分析建表信息后生成的主键信息建立 B+树索引文件
: 需使用信息 “char *KeyInfo;”
- 2) 通过 API 调用 CATALOG 分析查找信息后生成的查找范围信息查找记录
: 需使用信息

```
typedef struct Condition_Info {  
    OperatorType OperType;  
    pKeyAttr      min;  
    pKeyAttr      max;  
};
```

- 3) 通过 API 调用 CATALOG 分析插入信息后生成的插入记录主键指针更改 B+树
: 需使用信息 “pKeyAttr pPrimaryKey”
- 4) 通过 API 调用 CATALOG 分析删除信息后生成的查找范围信息更改 B+树
: 需使用信息

```
typedef struct Condition_Info {  
    OperatorType OperType;  
    pKeyAttr      min;  
    pKeyAttr      max;  
};
```

2.5.2.5 主要数据结构:

1) 索引文件头结构 Index_Head

该文件头结构中定义了索引文件在内存中的第一个空块地址, 最后一个空块地址, 第一个新块地址, 根节点地址, 第一个叶节点地址, 以及主键的属性个数, 主键的大小, 一个节点中所能容纳的最多节点数。

```
typedef struct {  
    FilePtr FirstEmptyBlock;  
    FilePtr LastEmptyBlock;  
    FilePtr FirstNewBlock;  
    FilePtr root;  
    FilePtr start;  
    int     KeyAttrNum;  
    int     KeySize;  
    int     MaxItemNum;  
} Index_Head;
```

2) 管理索引文件的类 BPTreeNode

在这个类中声明了读写索引文件头的所有操作函数, 并且通过该类能够获得索引主键信息, 设置主键信息, 获得根节点地址, 第一个叶节点地址。



```
class BPTree_Index {
    Index_Head      IdxHead;
    char*            KeyInfo;
public:
    BPTree_Index();
    ~BPTree_Index();
    void creatHead(char* KeyStr); //produce the info for IdxHead
    void writeHeadToFile();
    void readHeadFromFile();
    void setKeyInfo(char* Key_Info); //set KeySize and KeyAttrNum
    char* getKeyInfo();
    FilePtr getStartNode(); //get start FilePtr of the leftmost leaf node
    FilePtr getRoot();
};
```

3) B+树节点结构

I. 定义了一个管理 B+树节点的类 BPTreeNode:

```
class BPTreeNode {
    bool        IsLeaf;
    int         ItemInNode;
    FilePtr     *p;
    pKey_Arr    *k;
public:
    BPTreeNode() {
        p = new FilePtr[MaxItemNum + 1];
        k = new pKey_Arr[MaxItemNum];
    }
    ~BPTreeNode();
};
```

II. 节点结构用图的形式表示如下:

IsLeaf	ItemInNode	P1	K1	P2	K2	P3	K3	...	Kn-1	Pn
--------	------------	----	----	----	----	----	----	-----	------	----

III. 各参数的地址分配:

一个树节点分配同磁盘 BLOCK 相等的地址空间, 即 4KB。主键的分配空间为 12 bytes, 磁盘指针分配 8 bytes。

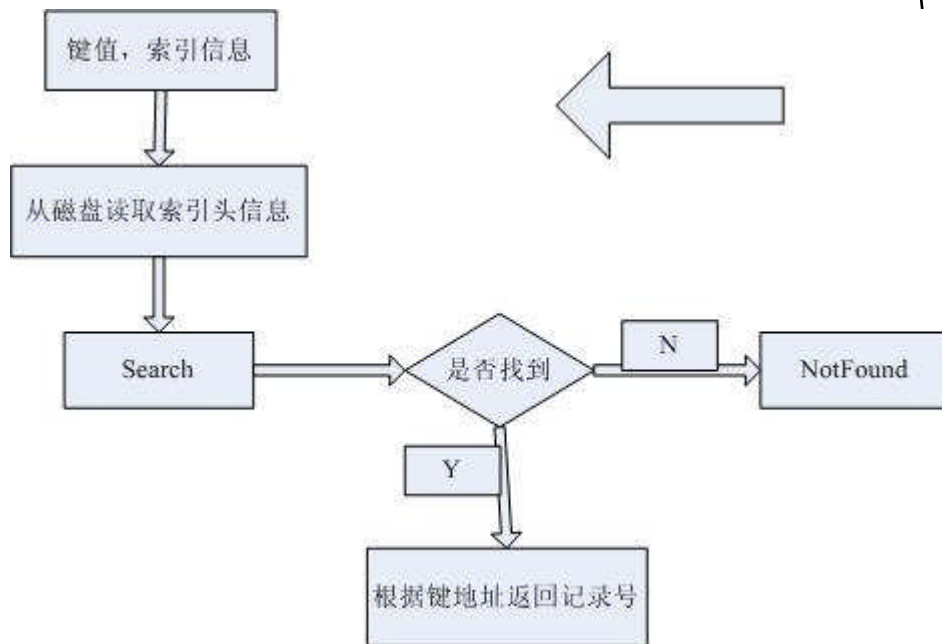
4) 管理 B+树所有操作的类 BPTree

在此类中声明了查找, 插入, 删除的所有操作函数, 可以实现 B+树的创建, 查找, 更新 (包括插入, 删除索引值)。

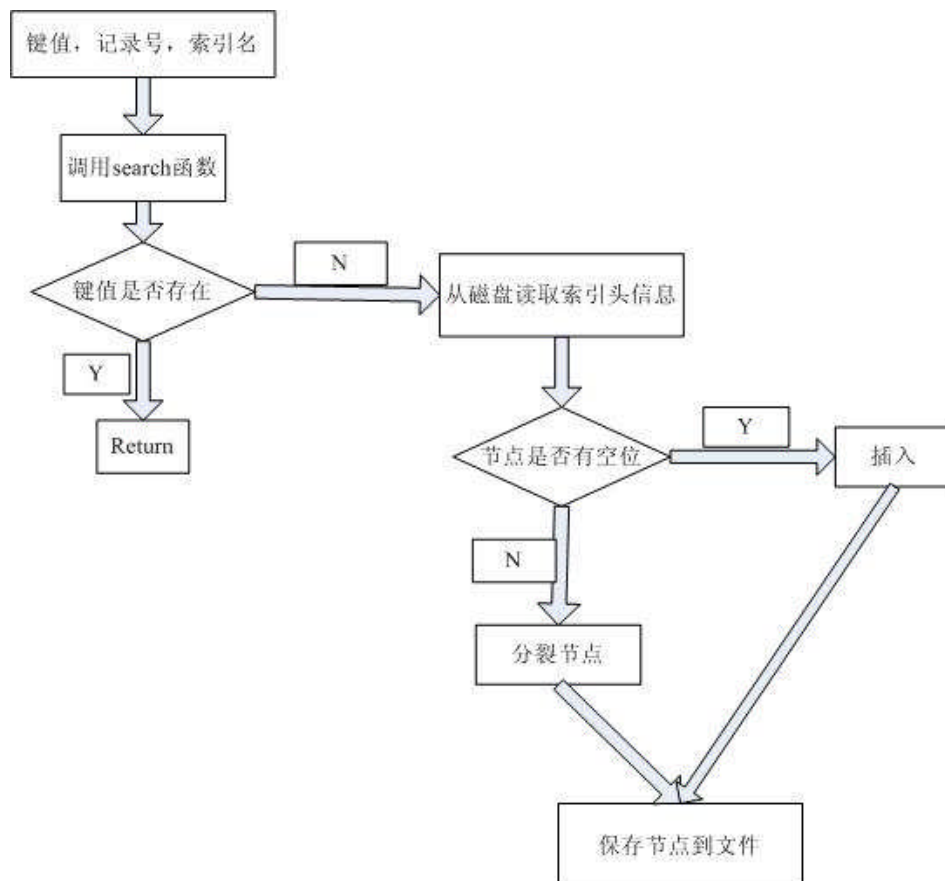
```
class BPTree {
    BPTreeNode *root; //the root node
    BPTreeNode *start; //the first leaf node
public:
    BPTree();
    ~BPTree();
    BPTreeNode* getRoot();
    BPTreeNode* getStartNode();
    FilePtr getCurRec(FilePtr *, int offset);
    bool compare(pKey_Arr, pKey_Arr); //compare two keys
    BPTreeNode* getSonNode(FilePtr);
    BPTreeNode* getParNode(FilePtr);
    void deleteEntry(BPTreeNode *, pKey_Arr, int);
    void insertEntry(BPTreeNode *, pKey_Arr, int);
    pNode_Key search(pKey_Arr);
    void select(Condition_Info&);
    void delete(pKey_Arr, int);
    void insert(pKey_Arr, int);
};
```

2.5.2.6 . 算法描述:

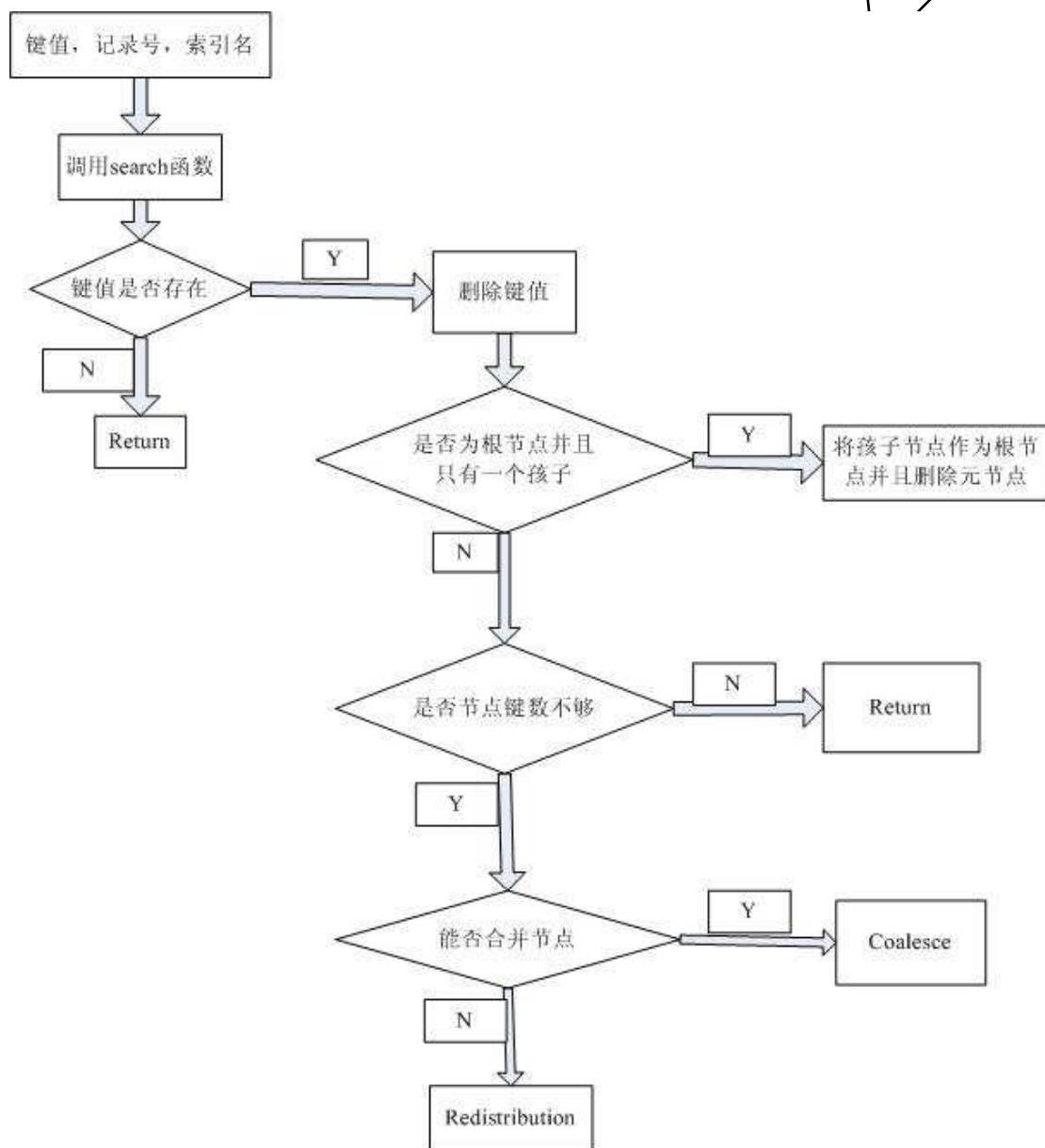
1) 查找算法流程



2) 插入算法流程



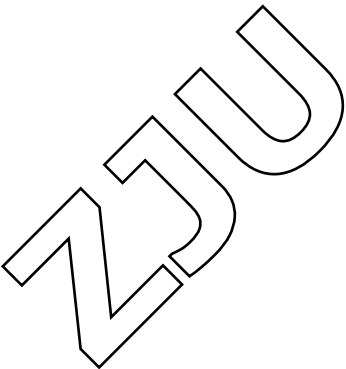
3) 删除算法流程



2.5.2.7接口:

见功能和算法说明。

与此模块存在调用关系的模块有 Buffer Manager 和 API 模块, 相关联的数据有记录数据文件。



2.5.2.8限制条件:

2.5.2.8.1 建立索引时主键必须是单属性并且唯一。

2.5.2.8.2 Block 大小为 4k。

2.5.2.8.3 整型、长整型、浮点数据均为 4 字节。

2.5.2.8.4 B+树高度不能超过 9。

2.5.2.9测试要点:

测试是否能生成索引文件并正确保存相关信息；提供连续的 key 作为输入，测试模块的主要接口是否能正确执行其功能并返回相关异常信息。

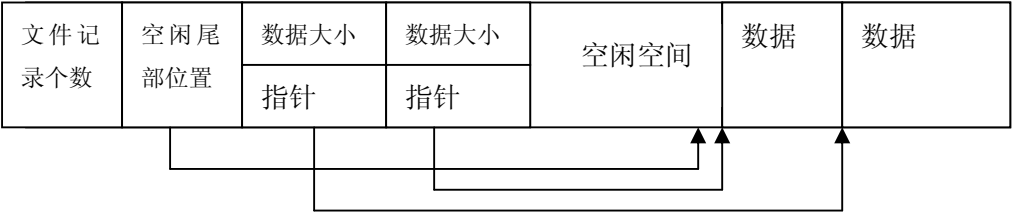
2.6 Buffer Manager

一、 功能概述

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

二、 设计思路



程序运行过程中,只存在一个 buffer,需要定义一个全局变量 MyDB,只支持单个数据库的多文件操作,若需切换数据库,重复调用 MyDB.End()和 MyDB.Start()完成参数的修改。

一个 buffer 在此 miniSQL 中规定为 100 个 MemPage,而每个 MemPage 是 4KB。考虑到不但减少写磁盘的频率,更要考虑到读取方便,将文件第一个块确定为信息块,存放几个链表的指针。数据结构如上图所示。Record Manager, category Manager 等都通过 buffer Manager 访问磁盘,这里 buffer Manager 采取 write back 的策略,即只写 buffer 里,而不直接写磁盘。对于读操作,如果文件在 buffer 里,直接读取,如果不在 buffer Manager 里,buffer Manager 负责把指定的数据块读入 buffer,如果 buffer 有空,则直接读入,如果 buffer 满了,则采用 LRU(least recently used)策略,将最少用的 MemPage 替换掉,替换时根据 modified 值来判断是否将替换的 MemPage 写入磁盘,如果没有修改过就不必写入磁盘,从而减少不必要磁盘访问的次数。

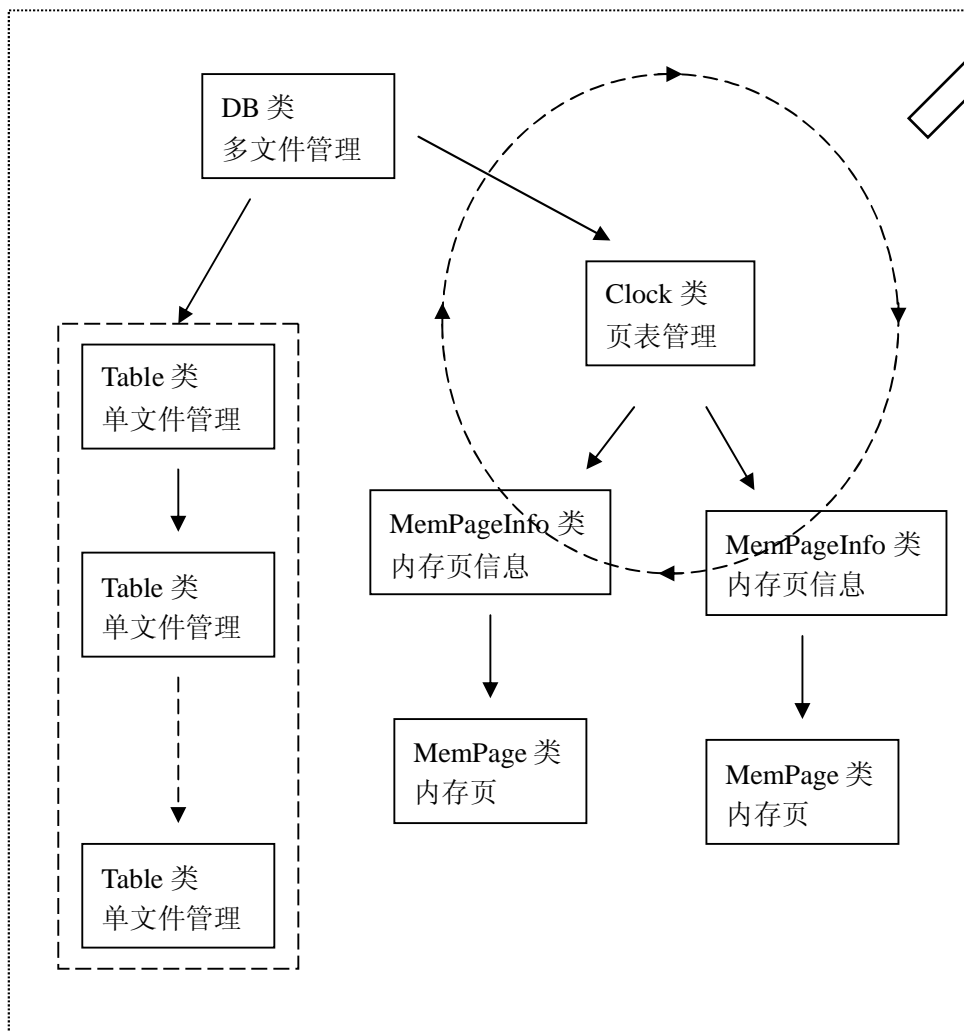
对于写操作,如果 buffer 里有空白的 MemPage,从文件中找是否有容得下写数据大小的块,并且该块不在 bufferManager 中(因为是 write back 策略,很可能文件中该块未满,而 buffer 中该数据块已经满了,只不过还未写回),将文件中该块读入 buffer。如果 buffer 中存在可以容下该数据大小的块,直接写入该块中。

Buffer Manager 封装了 dbfile, 和 MemPage 数组。Dbfile 封装了 Cfile,为了能在每次打开文件时对文件进行初始化,所以采用了用一个类封装。只要是刚创建的文件,就将第一块信息块初始化。

Buffer 模块提供上层模块 4 个接口,一是 DB 类的几个申明为 Public 的几个方法;二是 Table 类申明为 Public 的几个方法;三是 MemWrite 函数;四是 Fileptr 类,作为一个全局的类型使用。为防止其他模块访问计划外的类成员,所有计划外类成员(变量和函数)全部申明为 private。

作为 buffer 构成的基础单元 MemPage, buffer 实现的具体细节都在 MemPage 上。MemPage 采用分槽式结构,所以必须提供 addData(), rmData()等基本的接口,又因为表信息,索引信息,记录都是通过链表连起来的,所以 MemPage 还提供了相关链接功能的接口,同时 MemPage 必须保存到自己的硬盘位置,所以 save()接口是不可少的。

三、 具体实现



1. 声明宏

```
#define FILE_PAGESIZE      4096    // the size of MemPage
#define MEM_PAGEAMOUNT    1000    // the maximum number of MenPage
#define MAX_FILENAME_LEN  256     // the maximum length of filename
```

2. 声明结构体

文件物理页的头信息:

```
typedef struct {
    unsigned long PageID; //页编号
    bool IsFixed;         //是否为固定页（常驻内存页）
    void InitialPageHead(unsigned long mypageid,bool myisfixed); //初始化头信息
}TablePageHead;
```

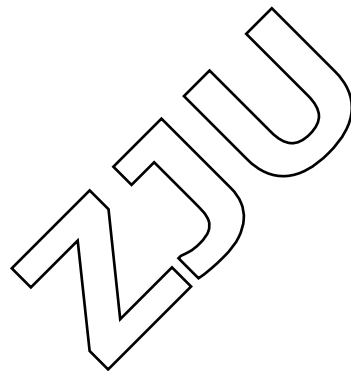
文件头信息:

```
typedef struct{
```

```

        unsigned long PageTotal; //文件已有页总数
        char LastModifiedDate[20]; //文件最近一次修改时间
        void InitialFileCond(); //文件头信息初始化
    }TableFileCond;

```



3. 声明类

数据库管理类:

```

class DB{
private:
    unsigned int TableCount; //目前已经打开的文件数
    unsigned int Currtableid; //目前正在操作的文件 ID
    Table* first; //第一个打开的文件
    Table* last; //最后一个打开的文件
    Table* CurrTable; //目前正在操作的文件
    class MyClock* MemPageClock; //CLOCK 页表
    Table* operator[](unsigned int tableid) const;
    //根据 tableid 返回相应的文件管理对象
    bool GetIsNew(unsigned int tableid) const; //根据 tableid 返回文件是否新建
    void SetIsNew(unsigned int tableid,bool isnew);
    //根据 tableid 设置文件是否新建
    unsigned long GetPageTotal(unsigned int tableid) const;
    //根据 tableid 返回文件拥有的总物理页数
    void AddPageTotal(unsigned int tableid,int add);
    //根据 tableid 设置文件物理总页数
    int GetPtr2File(unsigned int tableid) const; //根据 tableid 返回文件句柄
public:
    Table operator[](const char* filename);
    //根据文件名返回一个文件管理对象, 如果已经打开的文件, 找到相应的文件管
    //理对象, 直接返回, 反之新建一个文件管理对象并返回
    void Start();
    //数据库管理初始化, 主要是初始化 CLOCK 页表以及文件管理对象链表
    void End();
    //数据库管理对象析构, CLOCK 页表析构以及文件句柄的关闭等
};

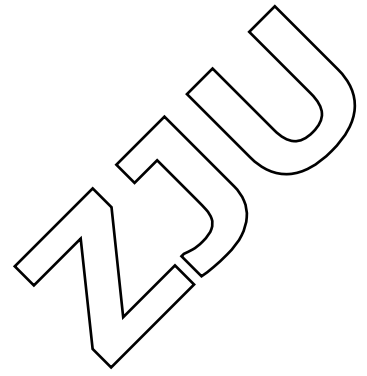
```

文件内指针:

```

class FilePtr{
public:
    void Initialize(); //指针初始化
    unsigned long FilePageID; //页编号
    unsigned int Offset; //页内偏移量
    void* MemAddr(); //转化为内存地址
    void ShiftOffset(int offset); //指针页内滑动

```

```
};
```

内存页:

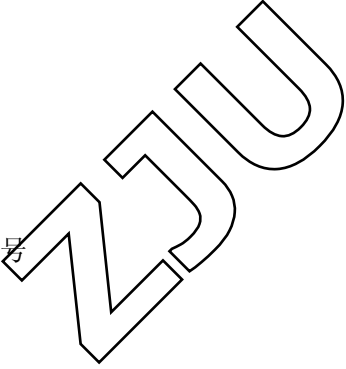
```
class MemPage{
private:
    unsigned int  TableID; //所属文件 ID
    unsigned long FilePageID; //页编码
    void* Ptr2PageBegin; //内存指针, 指向内存映射于物理页的首地址
    TablePageHead* Ptr2Head; //内存指针, 指向物理页头信息的首地址
    TableFileCond* Ptr2FileCond(); //取得文件头信息
    MemPage(); //构造函数, 开辟内存空间, 大小为物理页大小
    ~MemPage(); //析构函数、释放内存空间
    void LoadFromFile(unsigned int tableid,unsigned long filepageid);
        //从物理文件导入相应内容
    void Back2File() const; //写回物理文件, 假如该页被修改的情况下
};
```

内存页信息:

```
class MemPageInfo{
private:
    bool IsLastUsed; //最近一次访问内存页时是否被使用
    bool IsModified; //最近一次访问内存页时是否被修改
    class MemPage* Ptr2Page; //指向具体的一个内存页
    MemPageInfo(); //内存页信息初始化
    ~MemPageInfo(); //内存页信息析构
    void UpdatePageInfo(unsigned int tableid,unsigned long filepageid);
        //更新内存页信息, 通常导致所指向内存页写回和重新从物理文件导入
    TablePageHead* GetPtr2Head() const; //取得内存页头信息的首地址
    TableFileCond* GetPtr2FileCond() const;
        //取得内存页指向文件头信息的首地址
    unsigned int Gettableid() const; //取得内存页所属文件的 ID
    unsigned long GetFilePageID() const; //取得内存页的编号
};
```

CLOCK 页表:

```
class MyClock
{
private:
    unsigned int ClockSize; //页表大小
    unsigned int CurrClockPtr; //当前页表指针所指内存页
    class MemPageInfo* Ptr2MemPageInfo[MEM_PAGEAMOUNT+1];
        //页表内容
    MyClock(); //页表构造, 主要为初始化内存页信息
    ~MyClock(); //页表析构
};
```



```

void SetPageModified();
//设置是当前页表指针所指内存页信息标识被修改
unsigned int GetNullPage(); //取得未被初始化的内存页的信息编号
unsigned int NR_Search(unsigned int tableid);
//取得不属于当前处理文件的内存页的信息编号

unsigned int U_M_Search(bool islastused,bool ismodified,bool changeused);
//应用 CLOCK 算法，取得需要替换的内存页的信息编号

unsigned int GetSwapPage(unsigned int tableid)
{
    if(! this->NR_Search(tableid) ) //寻找不属于当前文件的内存页
        if(! this->U_M_Search(0,0,0) ) //寻找最近没用且未修改的内存页
            if(! this->U_M_Search(0,1,1) )
                //寻找最近没用但是被修改的页，同时把所有遇到的的使用位改为 0（未使用）
                if(! this->U_M_Search(0,0,0) )
                    //寻找最近未使用且未修改的内存页
                    this->U_M_Search(0,1,1);
                //寻找最近没用但是被修改的页，同时把所有遇到的的使用位改为 0（未使用）
}
return this->CurrClockPtr;
}
//取得需要替换的内存页的信息编号
unsigned int GetExsitPage(unsigned int tableid,unsigned long filepageid);
//取得已存在的内存页的信息编号
MemPageInfo* GetTargetPage(unsigned int tableid,unsigned long filepageid);
//取得最终所需要的内存页的信息编号
};

```

文件管理类:

```

class Table{
private:
    unsigned int TableID; //文件编号
    unsigned long PageTotal; //文件所含物理页总数
    bool IsNew; //文件是否新建
    char FileName[MAX_FILENAME_LEN]; //文件名称
    Table* next; //下一个已经打开的文件
    int Ptr2File; //指向该文件的文件句柄
    void Deconstruct(); //对象析构
    MemPageInfo* GetPageInfo(unsigned long filepageid) const;
//取得属于该文件的某一内存页信息，其所指向的内存页中所含的恰为指定的该文件中的一个物理页
    Table(const char *name,unsigned int tableid); //构造函数，文件打开和新建
public:
    FilePtr operator[](unsigned long filepageid) const;
//取得一个 FilePtr，其转换为内存地址后所指恰为指定的物理页在内存中的映像

```

的首地址往后滑动物理页头信息大小后的地址，此地址用于其他模块在该页中可写的首地址。

```
FilePtr GetCataPoint() const; //取得 Catalog 模块在文件中可写的首地址
FilePtr GetIdxPoint() const; //取得 B+模块在文件中可写的首地址
unsigned long GetPageTotal() const; //取得该文件目前所有的页总数
void SetPageFixed(unsigned long filepageid); //使某一物理页常驻内存
void SetPageUnFixed(unsigned long filepageid); //使某一物理页不常驻内存
};
```

4. 主要函数及其功能

内存写函数:

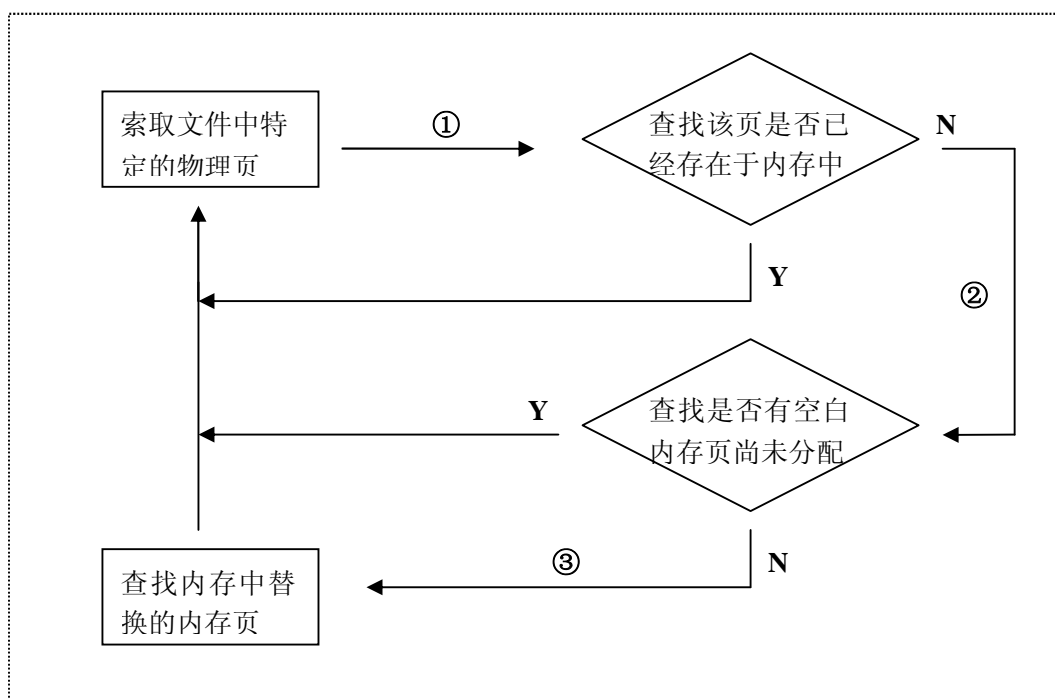
```
FilePtr MemWrite(const void*,size_t,FilePtr*);
```

//该函数进行内存拷贝，并进行内存页溢出检测，如果溢出，自动修改第三个参数至下一个可写的 FilePtr，并返回内存拷贝结束后下一个可以写的 FilePtr。

算法：CLOCK 页表管理类中有一个页替换算法，采用的是扩展了的 clock 算法。

详细算法见数据结构中 MyClock :: GetSwapPage(unsigned int tableid)成员函数。

逻辑流程图：在寻找最终所需要的内存页的时候，有流程图如下：

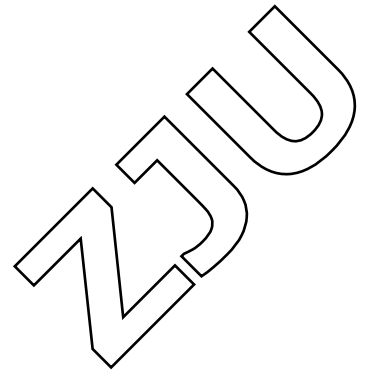


四、测试要点

1. 模拟数据写入，保证换页、页溢出判断、写回页有效。

```
FilePtr MemWriteTest(size_t,_FilePtr*);
```

根据欲写入的文件地址，测试把其他模块内存中的数据写入实际未写入。用于当整个数据对象不能跨快写的时候又不能一次写入（如链表结构(各字段)的一条记录，在内存中不连续）的场合。



Record Manager 负责管理记录表中数据的数据文件。主要功能为实现记录的插入、删除与查找操作，并对外提供相应的接口。其中记录的查找操作要求能够支持不带条件的查找和带条件的查找（包括等值查找、不等值查找和区间查找）。

数据文件由一个或多个数据块组成，块大小应与缓冲区块大小相同。一个块中包含一条至多条记录，为简单起见，只要求支持定长记录的存储，且不要求支持记录的跨块存储。

1) 程序文件: record.h record.cpp

2) 主要数据结构:

文件中被删除记录结构体

```
typedef struct{
```

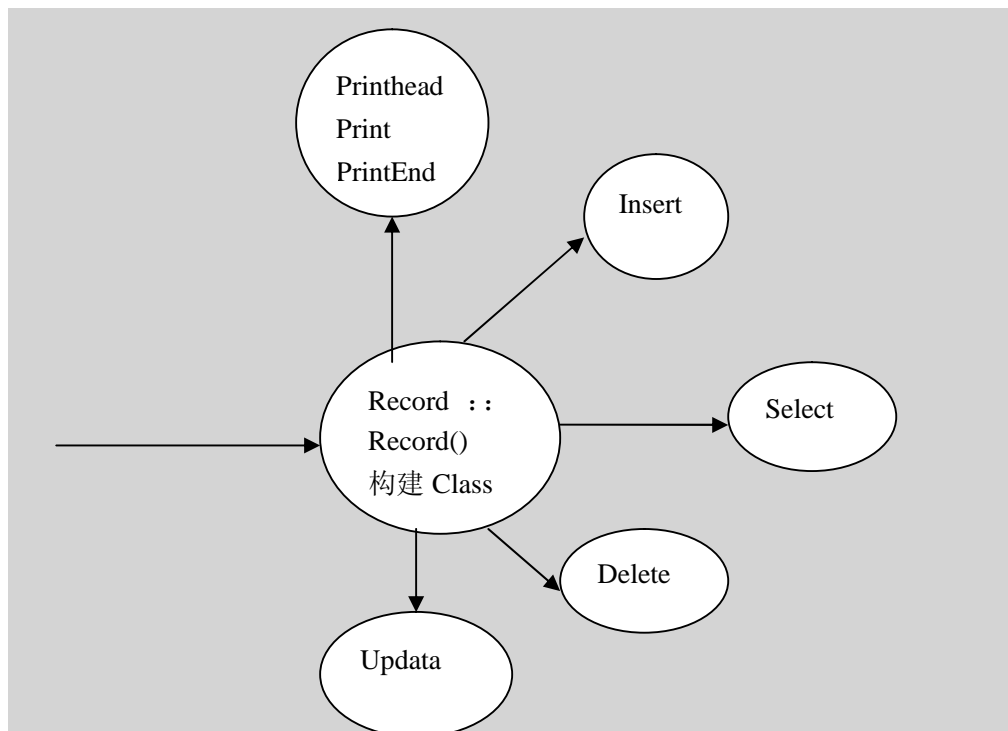
```
    _F_FileAddr DelFirst; // 第一个被删除的记录地址
```

```
    _F_FileAddr DelLast; // 最后一个被删除的记录地址
```

```
    _F_FileAddr NewInsert; // 文件末尾第一个可插入记录的地址
```

```
}_F_DELLIST;
```

主结构体



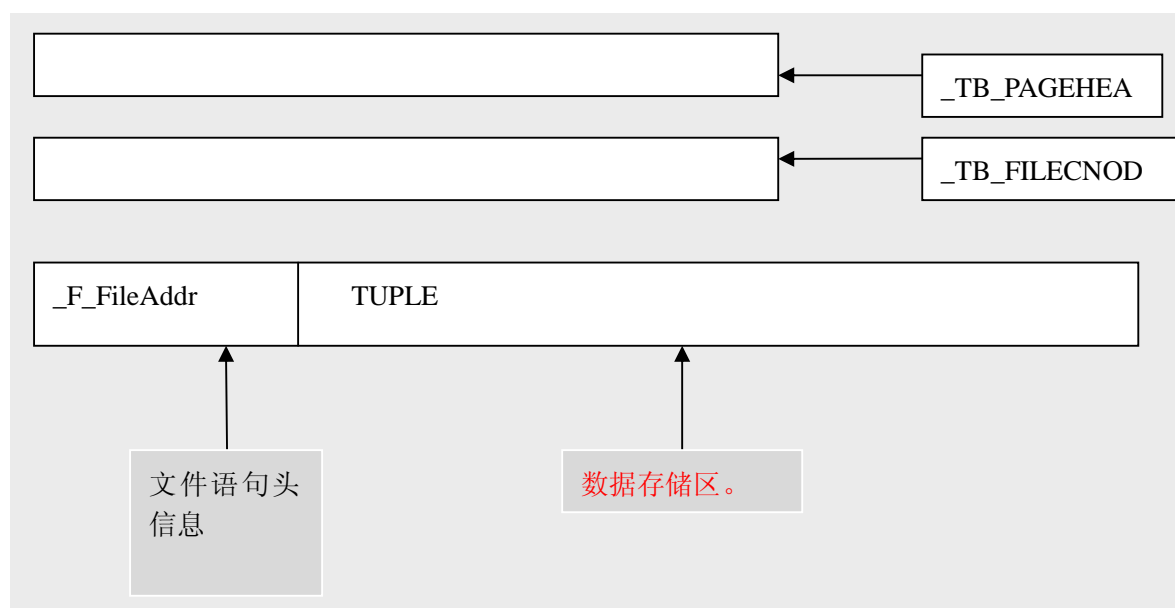
```
class record
{
public:
    Record();
    ~Record(){}
    _F_FileAddr Insert(Rec_Info&); //插入一个 Tuple
    void Delete(_F_FileAddr&); //删除一个 Tuple
    void Update(_F_FileAddr&,Rec_Info&);//更新数据
    Rec_Info* Select(_F_FileAddr&,Select_Rec_Info&) const;//选择单条 Tuple
    void PrintHead(Select_Rec_Info&) const; //打印列表头
    void Print(_F_FileAddr&,Select_Rec_Info&) const;//打印 record with index
```

```

void PrintEnd(Select_Rec_Info&) const;//打印列表尾
// void Print(Select_Rec_Info&, Select_Cond_Info &);
private:
//block 中连续文件中是否有空，用于 insert 如果可以的话，由 catalog 提供
_F_DELLIST * m_DelList;
_M_File * m_Dbffile;
}

```

3) 文件组织结构



如果为文件首页则有 `_TB_FILECOND` 不是刚没有；在文件中信息存放是连续的，并不跟上图一样。

4) 调用的其它模块：BUFFER 模块，和异常处理模块

5) 其具体实现如下：

选择语句

select * from 表名 ；

对表文件进行遍历，并对每条记录，根据选择属性名组，打印出对应的属性值。

select * from 表名 **where** 条件 ；

其中“条件”具有以下格式：列 op 值 and 列 op 值 ... and 列 op 值。

op 是算术比较符：= <> < > <= >=

若无可用的索引，则须对表文件进行遍历，并对每条记录，进行 **where** 条件匹配，若符合，则打印相应属性的值(用 **Print** 函数)。若有可用的索引，则由 **Index Manater** 模块直接查找得到并反回 **FilePtr**，并打印相应属性的值。在最后打印出被选择的记录的条数。

在这里我们只实现了单主键上的搜索，所以遍历和条件判断在 **INDEX** 中。

只写了单语句的搜索；

算法：

```

Rec_Info* Select(_F_FileAddr&,Select_Rec_Info&) const;
{
    Rec_Info* RPreInfo = new class Rec_Info;//存放 Select 出来的信息
    TCell_Info* CPTemp , * CPTemp1;
}

```

```

RPreInfo->head=NULL;
//向 Rec_Info 写信息
Select_Cell* SPTemp=SelRecInfo.head;
_F_FileAddr temp=*(_F_FileAddr*)FAddr.MemAddr();//取该语句存放地址
if(temp!=FAddr)//与要选择的信息不一致，返回
    return 0;
temp.ShiftOffset(sizeof(_F_FileAddr));
while(SPTemp)
{
    CPTemp = new Cell_Info;
    if(SPTemp == SelRecInfo.head)//第一个 Column 要链到整条信息的头信息
        RPreInfo->head=CPTemp;
    else
    {
        CPTemp1->next = CPTemp;
    }
    CPTemp1=CPTemp;
    temp.ShiftOffset(SPTemp->PriorLength);//移动 OFFSET 指针到要读的数据位置
    //不同类数据的读取
    switch(SPTemp->ColType)
    {
        //对 Cell_Info 符值
    }
    temp.ShiftOffset( -SPTemp->PriorLength);//将 OFFSET 指返回初始值，以便下次
    SPTemp=SPTemp->next;
}
return RPreInfo;
}
}

```

使用

插入记录语句

insert into 表名 **values** (值 1 , 值 2 , ... , 值 n);

根据 Catalog Manager 处理生成的初步内部数据形式，提取表名及记录，根据信息计算插入记录的块号，并调用 Buffer Manager 的功能，获取指定的内存块，并将记录插入到内存中，更新表的信息。

```

_F_FileAddr record::Insert (Rec_Info& a)
{
    Cell_Info* temp = a.head;
    _F_FileAddr TargetPlace , InsertPlace;
    TargetPlace = InsertPlace = FindInserPlace;//取插入地址
    InsertPlace=Memwrite( (void*)insert_place,sizeof(FilePtr) ,& InsertPlace)//写头信
    while(temp)
    {

```

息

```

        Switch(ColType)
        {
            //Write the cell information 插入数据
        }
        temp = temp->next;
    }
    return TargetPlace;
}

```

删除记录语句

delete from 表名 ;

把该表中的所有记录删除并更新表的信息。在最后打印出被已删除的记录条数。

delete from 表名 **where** 条件 ;

根据 Catalog Manager 处理生成的内部数据形式，提取表名及可用的索引。若有可用的索引，则须根据索引来查找符合条件的记录，删除该记录并更新该表的信息，循环往复，直至所有符合条件的记录都被删除为止。若无可用的索引，则须对表文件中的所有记录进行一次遍历，并对每一条记录都需要判别是否符合条件。若符合条件，则删除该记录并更新表的信息。在最后打印出被已删除的记录条数。

在这里我们只实现了单主键上的搜索，所以遍历和条件判断在 INDEX 中。

只写了单语句的删除;

```

void record::Delete (_F_FileAddr &DelTarget)
{
    _F_FileAddr * temp=(_F_FileAddr *) DelTarget.Memaddr();
    if(DelFirst==DelLast==0)
    {
        m_DelList->DelFirst=temp;
        m_DelList-> DelLast_=temp;
        _F_FileAddr ZPtr;
        ZPtr.Initialize();
        //ZPtr.Status='I';如果有遍历整个文件功能
        MemWrite(&ZPtr,sizeof(_F_FileAddr),FPTemp);//将被删除处的头信息写空}
        else
        {
            MemWrite(&DelTarget,sizeof(_F_FileAddr),&m_DelList->DelLast);
            m_DelList->DelLast = DelTarget;
        }
    }
}

```

更新记录语句

Update 表名 **set** 更新信息 **where** 条件

根据 Catalog Manager 处理生成的内部数据形式，提取表名及可用的索引。若有可用的索引，则须根据索引来查找符合条件的记录，更新该记录并更新该表的信息，循环往复，直至所有符合条件的记录都被更新为止。若无可用的索引，则须对表文件中的所有记录进行一次遍历，并对每一条记录都需要判别是否符合条件，若符合条件的记录，更新该记录并更新该表的信息;

在这里我们只实现了单主键上的搜索，所以遍历和条件判断在 INDEX 中。

```
void Record::Update(_F_FileAddr& UpdateFAddr, Rec_Info& UpdateInfo)
{
    Cell_Info* temp = UpdateInfo.head;
    _F_FileAddr UFATemp = UpdateFAddr;
    UFATemp.ShiftOffset(sizeof(_F_FileAddr));
    while(temp)                                //Write the cell information
    {
        UFATemp.ShiftOffset(temp->PriorLength);
        switch(temp->ColType)
        {
            //Write the cell information
        }
        UFATemp.ShiftOffset(-temp->PriorLength);
        temp = temp->next;
    }
}
```

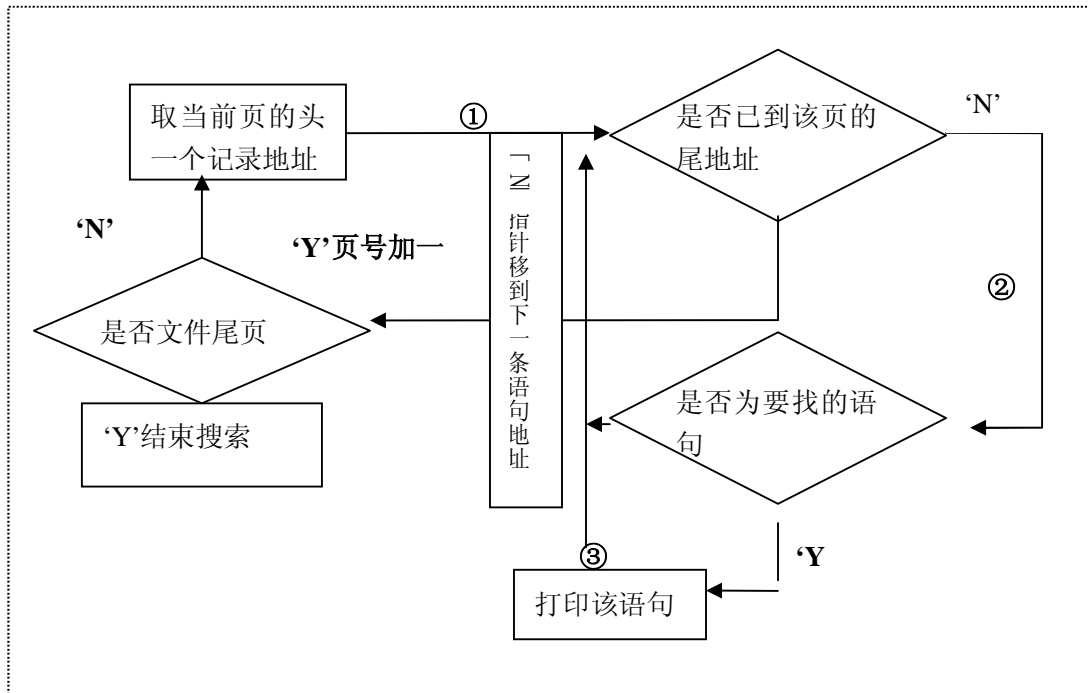
6) 扩展（多条件多属性的查找，直接遍历表，可惜我们没有实现，我写了 Record 的代码）
在 Buffer 中的 _F_FileAddr 结构中加一个 status 标记 类型为 char 值为 'u' 或 'l' 初始化为 'u'

遍历所要的和所要定义的结构体//由 catalog 提供

```
typedef struct TSelect_Cond_Cell
{
    Column_Type    ColType;    //字段类型
    int            PriorLength; //记录头到此字段之间的长度
    int            ColLength;   //字段类型的长度
    Column_Value   ColSelValue; //记录值
    Operator_Type  OperType;    //条件类
    TSelect_Cond_Cell* next;    //下一个字段信息
    TSelect_Cond_Cell():PriorLength(0),ColLength(0),ColType(I),OperType(B)
    {
        this->next = NULL;
        strcpy(this->ColumnName, "");
    }
}Select_Cond_Cell;

class Select_Cond_Info
{
public:
    Select_Cond_Cell* head;
    int                ColumnNum;
    int                RecordLength;
    Select_Cond_Info():ColumnNum(0) { this->head=NULL; }
    ~Select_Cond_Info(){};
```

```
};
```



算法：

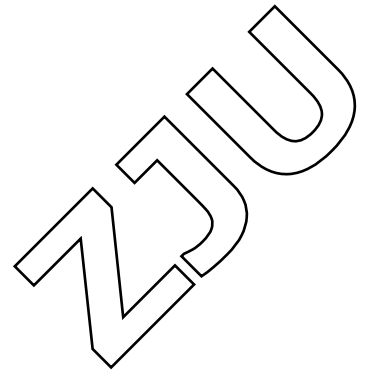
```

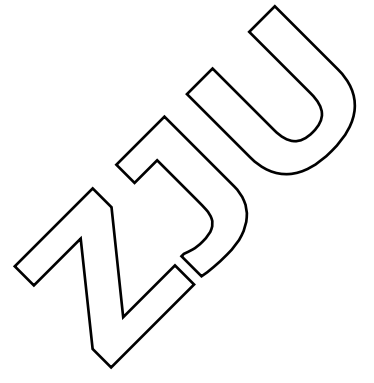
void Record::Print(Select_Rec_Info& prilist, Select_Cond_Info &SelCondInfo)
{
    _F_FileAddr *pFile;
    _F_FileAddr FileAddr;
    int TotalLength = sizeof(_F_FileAddr) + SelCondInfo.RecordLength; // 每条记录的总长度
    // 所有页进行查找
    for(unsigned long i=0; i<targettb.GetPageTotal(); i++)
    {
        // 设置每页第一条语句的页数 ID 和偏移地址以取得文件指针
        if(i==0)
        {
            // 第一页包含了一个文件头信息和一个页头信息
            FileAddr.ShiftOffset(sizeof(_TB_FILECOND) + sizeof(_TB_PAGEHEAD));
            FileAddr.ulFilePageID=0;
            pFile = (_F_FileAddr*)FileAddr.MemAddr();
        }
        else
        {
            // 其它页只有一个页头信息
            FileAddr.ShiftOffset(sizeof(_TB_PAGEHEAD));
            FileAddr.ulFilePageID=i;
            pFile = (_F_FileAddr*)FileAddr.MemAddr();
        }
    }
}
  
```

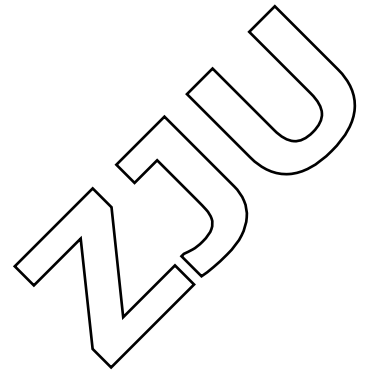
```

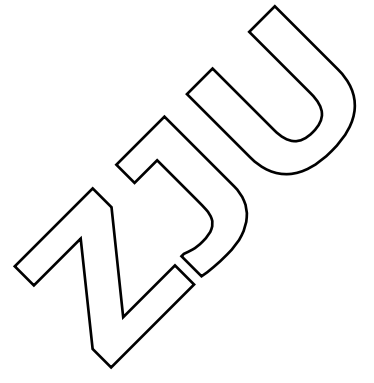
//页内搜索，
while((pFile->uiOffset+TotalLength)<FILE_PAGESIZE)//记录是否溢出，溢出则跳出
循环
{
    int IfCatch=0;//标记该语句是否为所要找的语句
    FileAddr = *pFile;
    if(pFile==&this->m_DelList->NewInsert)//到文件尾结整搜索
        return ;
    pFile=(_F_FileAddr *)((char*)pFile+TotalLength);//指到下一语句地址
    if(FileAddr.Status=='1')//判断该语句是否已删除
        continue;
    TSelect_Cond_Cell* SCCT=SelCondInfo.head;//搜索条件
    Column_Value TempVal;//用于存放临时读取的数据
    while(SCCT)//多条件循环
    {
        //匹配条件
        SCCT=SCCT->next;//下一个条件
    }
    if(IfCatch)//如果找到符合条件的语句
    {
        this->Print(FileAddr,prilist);
    }
}
return ;
}

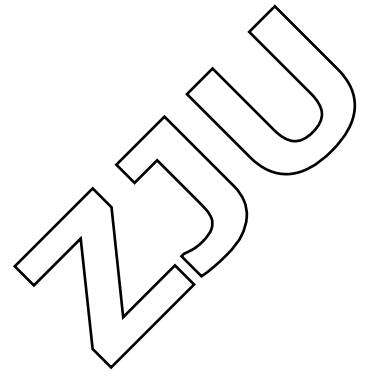
```











文件清单:

	模块名	程序文件名	运行平台	语言	简要描述
用户接口	解释器	Interpreter.h	Windows/Linux	C/C++	解释器模块头文件
		Interpreter.cpp	Windows/Linux	C/C++	解释器模块实现文件
	API 集成	Ctrl_Center.h	Windows/Linux	C/C++	API Lib 头文件
		Ctrl_Center.cpp	Windows/Linux	C/C++	API Lib 实现文件
	异常处理	Error.h	Windows/Linux	C/C++	Error Lib 头文件
		Error.cpp	Windows/Linux	C/C++	Error Lib 实现文件
	B+树	Bptree.h	Windows/Linux	C/C++	B+树模块头文件
		Bptree.cpp	Windows/Linux	C/C++	B+树模块实现文件
	Record	Record.h	Windows/Linux	C/C++	Record 模块头文件
		Record.cpp	Windows/Linux	C/C++	Record 模块实现文件
系统内核	Catalog	Catalog.h	Windows/Linux	C/C++	Catalog 模块头文件
		Catalog.cpp	Windows/Linux	C/C++	Catalog 模块实现文件
	Buffer	Buffer.h	Windows/Linux	C/C++	Buffer 模块头文件
		Buffer.cpp	Windows/Linux	C/C++	Buffer 模块实现文件
	Main	MiniSQL.h	Windows/Linux	C/C++	系统主程序头文件
		MiniSQL.cpp	Windows/Linux	C/C++	系统主程序实现文件
		Glob_Var.h	Windows/Linux	C/C++	系统全局变量头文件
	Gvariable				
系统集成	Display	Glob_Var.cpp	Windows/Linux	C/C++	系统全局变量实现文件
		Display.cpp			
		Display.h	Windows/Linux	C/C++	显示模块